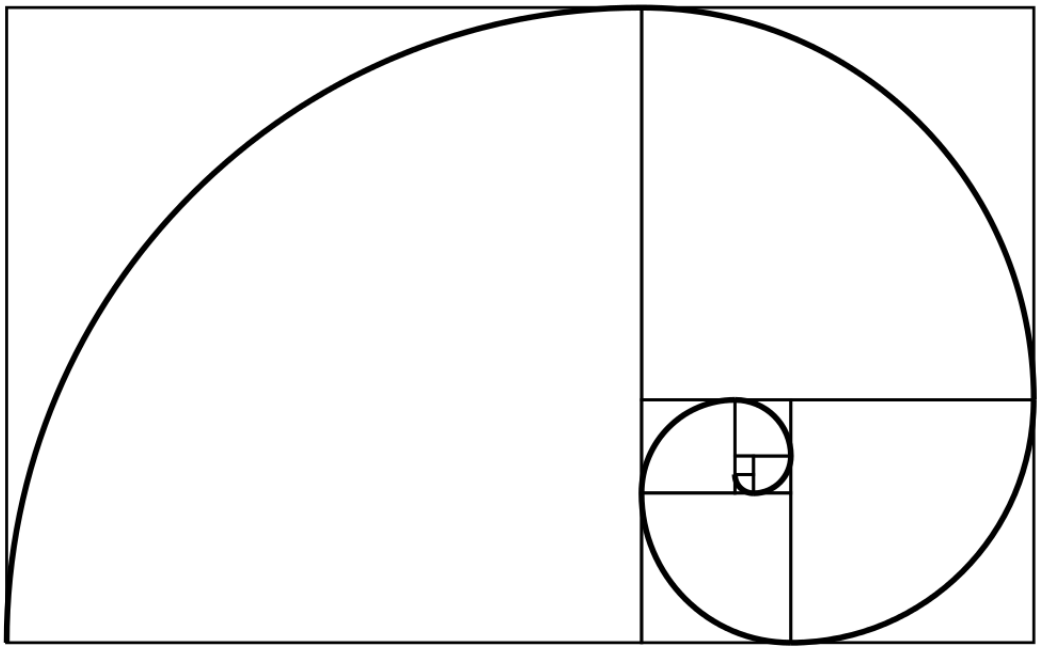
Elementary Algorithms



Larry LIU Xinyu [1]

August 25, 2018

[1]**Larry LIU Xinyu**
Version: 0.6180339887498949
Email: liuxinyu95@gmail.com

www.dbooks.org

Contents

I Preface **5**

0.1 Why? . . . 7
0.2 The smallest free ID problem, the power of algorithms . . . 7
0.2.1 Improvement 1 . . . 8
0.2.2 Improvement 2, Divide and Conquer . . . 9
0.2.3 Expressiveness vs. Performance . . . 10
0.3 The number puzzle, power of data structure . . . 12
0.3.1 The brute-force solution . . . 12
0.3.2 Improvement 1 . . . 12
0.3.3 Improvement 2 . . . 15
0.4 Notes and short summary . . . 17
0.5 Structure of the contents . . . 18

II Trees **21**

1 Binary search tree, the 'hello world' data structure **23**

1.1 Introduction . . . 23
1.2 Data Layout . . . 24
1.3 Insertion . . . 27
1.4 Traversing . . . 28
1.5 Querying a binary search tree . . . 31
1.5.1 Looking up . . . 31
1.5.2 Minimum and maximum . . . 32
1.5.3 Successor and predecessor . . . 32
1.6 Deletion . . . 34
1.7 Randomly build binary search tree . . . 38

2 The evolution of insertion sort **43**

2.1 Introduction . . . 43
2.2 Insertion . . . 44
2.3 Improvement 1 . . . 46
2.4 Improvement 2 . . . 47
2.5 Final improvement by binary search tree . . . 49
2.6 Short summary . . . 50

www.dbooks.org

3 Red-black tree, not so complex as it was thought 53
3.1 Introduction . . . 53
3.1.1 Exploit the binary search tree . . . 53
3.1.2 How to ensure the balance of the tree . . . 54
3.1.3 Tree rotation . . . 56
3.2 Definition of red-black tree . . . 58
3.3 Insertion . . . 60
3.4 Deletion . . . 63
3.4.1 The sibling of the doubly black node is black, and it has one red child . . . 65
3.4.2 The sibling of the doubly-black node is red . . . 66
3.4.3 The sibling of the doubly-black node, and its two children are all black . . . 67
3.5 Imperative red-black tree algorithm ⋆ . . . 69
3.6 More words . . . 72

4 AVL tree 75
4.1 Introduction . . . 75
4.1.1 How to measure the balance of a tree? . . . 75
4.2 Definition of AVL tree . . . 75
4.3 Insertion . . . 78
4.3.1 Balancing adjustment . . . 80
4.3.2 Pattern Matching . . . 82
4.4 Deletion . . . 83
4.5 Imperative AVL tree algorithm ⋆ . . . 83
4.6 Chapter note . . . 87

5 Radix tree, Trie and Prefix Tree 91
5.1 Introduction . . . 91
5.2 Integer Trie . . . 92
5.2.1 Definition of integer Trie . . . 93
5.2.2 Insertion . . . 93
5.2.3 Look up . . . 95
5.3 Integer prefix tree . . . 96
5.3.1 Definition . . . 97
5.3.2 Insertion . . . 97
5.3.3 Look up . . . 103
5.4 Alphabetic Trie . . . 104
5.4.1 Definition . . . 104
5.4.2 Insertion . . . 106
5.4.3 Look up . . . 108
5.5 Alphabetic prefix tree . . . 109
5.5.1 Definition . . . 109
5.5.2 Insertion . . . 110
5.5.3 Look up . . . 115
5.6 Applications of trie and prefix tree . . . 116
5.6.1 E-dictionary and word auto-completion . . . 116
5.6.2 T9 input method . . . 121
5.7 Summary . . . 124

6 B-Trees **127**
6.1 Introduction . . . 127
6.2 Insertion . . . 129
6.2.1 Splitting . . . 129
6.3 Deletion . . . 136
6.3.1 Merge before delete method . . . 136
6.3.2 Delete and fix method . . . 144
6.4 Searching . . . 150
6.5 Notes and short summary . . . 151

III Heaps **155**

7 Binary Heaps **157**
7.1 Introduction . . . 157
7.2 Implicit binary heap by array . . . 157
7.2.1 Definition . . . 158
7.2.2 Heapify . . . 159
7.2.3 Build a heap . . . 160
7.2.4 Basic heap operations . . . 162
7.2.5 Heap sort . . . 169
7.3 Leftist heap and Skew heap, the explicit binary heaps . . . 171
7.3.1 Definition . . . 172
7.3.2 Merge . . . 173
7.3.3 Basic heap operations . . . 174
7.3.4 Heap sort by Leftist Heap . . . 175
7.3.5 Skew heaps . . . 176
7.4 Splay heap . . . 178
7.4.1 Definition . . . 178
7.4.2 Heap sort . . . 184
7.5 Notes and short summary . . . 184

8 From grape to the world cup, the evolution of selection sort **189**
8.1 Introduction . . . 189
8.2 Finding the minimum . . . 191
8.2.1 Labeling . . . 192
8.2.2 Grouping . . . 193
8.2.3 performance of the basic selection sorting . . . 194
8.3 Minor Improvement . . . 195
8.3.1 Parameterize the comparator . . . 195
8.3.2 Trivial fine tune . . . 196
8.3.3 Cock-tail sort . . . 197
8.4 Major improvement . . . 201
8.4.1 Tournament knock out . . . 201
8.4.2 Final improvement by using heap sort . . . 209
8.5 Short summary . . . 210

www.dbooks.org

9 Binomial heap, Fibonacci heap, and pairing heap **213**
9.1 Introduction . . . 213
9.2 Binomial Heaps . . . 213
9.2.1 Definition . . . 213
9.2.2 Basic heap operations . . . 218
9.3 Fibonacci Heaps . . . 228
9.3.1 Definition . . . 228
9.3.2 Basic heap operations . . . 230
9.3.3 Running time of pop . . . 240
9.3.4 Decreasing key . . . 241
9.3.5 The name of Fibonacci Heap . . . 243
9.4 Pairing Heaps . . . 246
9.4.1 Definition . . . 246
9.4.2 Basic heap operations . . . 246
9.5 Notes and short summary . . . 252

IV Queues and Sequences **257**

10 Queue, not so simple as it was thought **259**
10.1 Introduction . . . 259
10.2 Queue by linked-list and circular buffer . . . 260
10.2.1 Singly linked-list solution . . . 260
10.2.2 Circular buffer solution . . . 263
10.3 Purely functional solution . . . 266
10.3.1 Paired-list queue . . . 266
10.3.2 Paired-array queue - a symmetric implementation . . . 269
10.4 A small improvement, Balanced Queue . . . 270
10.5 One more step improvement, Real-time Queue . . . 272
10.6 Lazy real-time queue . . . 279
10.7 Notes and short summary . . . 282

11 Sequences, The last brick **285**
11.1 Introduction . . . 285
11.2 Binary random access list . . . 286
11.2.1 Review of plain-array and list . . . 286
11.2.2 Represent sequence by trees . . . 286
11.2.3 Insertion to the head of the sequence . . . 288
11.3 Numeric representation for binary random access list . . . 293
11.3.1 Imperative binary random access list . . . 296
11.4 Imperative paired-array list . . . 299
11.4.1 Definition . . . 299
11.4.2 Insertion and appending . . . 300
11.4.3 random access . . . 300
11.4.4 removing and balancing . . . 301
11.5 Concatenate-able list . . . 303
11.6 Finger tree . . . 306
11.6.1 Definition . . . 307
11.6.2 Insert element to the head of sequence . . . 309
11.6.3 Remove element from the head of sequence . . . 312

11.6.4 Handling the ill-formed finger tree when removing 313
11.6.5 append element to the tail of the sequence 318
11.6.6 remove element from the tail of the sequence 319
11.6.7 concatenate 320
11.6.8 Random access of finger tree 325
11.7 Notes and short summary 337

V Sorting and Searching 341

12 Divide and conquer, Quick sort vs. Merge sort 343
12.1 Introduction 343
12.2 Quick sort 343
12.2.1 Basic version 344
12.2.2 Strict weak ordering 345
12.2.3 Partition 346
12.2.4 Minor improvement in functional partition 349
12.3 Performance analysis for quick sort 351
12.3.1 Average case analysis ⋆ 352
12.4 Engineering Improvement 355
12.4.1 Engineering solution to duplicated elements 355
12.5 Engineering solution to the worst case 362
12.6 Other engineering practice 366
12.7 Side words 367
12.8 Merge sort 368
12.8.1 Basic version 368
12.9 In-place merge sort 375
12.9.1 Naive in-place merge 376
12.9.2 in-place working area 377
12.9.3 In-place merge sort vs. linked-list merge sort 381
12.10Nature merge sort 383
12.11Bottom-up merge sort 389
12.12Parallelism 391
12.13Short summary 392

13 Searching 397
13.1 Introduction 397
13.2 Sequence search 397
13.2.1 Divide and conquer search 398
13.2.2 Information reuse 418
13.3 Solution searching 446
13.3.1 DFS and BFS 446
13.3.2 Search the optimal solution 483
13.4 Short summary 512

VI Appendix 515

Appendices

www.dbooks.org

A Lists **517**
A.1 Introduction . . . 517
A.2 List Definition . . . 517
A.2.1 Empty list . . . 518
A.2.2 Access the element and the sub list . . . 518
A.3 Basic list manipulation . . . 519
A.3.1 Construction . . . 519
A.3.2 Empty testing and length calculating . . . 520
A.3.3 indexing . . . 521
A.3.4 Access the last element . . . 522
A.3.5 Reverse indexing . . . 523
A.3.6 Mutating . . . 525
A.3.7 sum and product . . . 535
A.3.8 maximum and minimum . . . 539
A.4 Transformation . . . 542
A.4.1 mapping and for-each . . . 543
A.4.2 reverse . . . 549
A.5 Extract sub-lists . . . 551
A.5.1 take, drop, and split-at . . . 551
A.5.2 breaking and grouping . . . 553
A.6 Folding . . . 558
A.6.1 folding from right . . . 558
A.6.2 folding from left . . . 560
A.6.3 folding in practice . . . 563
A.7 Searching and matching . . . 564
A.7.1 Existence testing . . . 564
A.7.2 Looking up . . . 565
A.7.3 finding and filtering . . . 565
A.7.4 Matching . . . 568
A.8 zipping and unzipping . . . 570
A.9 Notes and short summary . . . 573

B The imperative red-black tree deletion algorithm **577**
B.1 Doubly Black node . . . 577
B.1.1 The doubly black node has a black sibling, and one of its nephew is red. . . . 578
B.1.2 The sibling of the doubly black node is red. . . . 580
B.1.3 The sibling of the doubly black node, and both nephews are black. . . . 581

C AVL tree - proofs and deletion algorithm **587**
C.1 Height increment after insertion . . . 587
C.2 Proof to the balance adjustment after insertion . . . 588
C.3 Deletion algorithm . . . 592
C.3.1 Functional deletion . . . 592
C.3.2 Imperative deletion . . . 594

D Suffix Tree **599**
D.1 Introduction . . . 599
D.2 Suffix trie . . . 600
D.2.1 Node transfer and suffix link . . . 601
D.2.2 On-line construction . . . 602
D.3 Suffix Tree . . . 606
D.3.1 On-line construction . . . 606
D.4 Suffix tree applications . . . 615
D.4.1 String/Pattern searching . . . 615
D.4.2 Find the longest repeated sub-string . . . 617
D.4.3 Find the longest common sub-string . . . 619
D.4.4 Find the longest palindrome . . . 621
D.4.5 Others . . . 621
D.5 Notes and short summary . . . 621

GNU Free Documentation License **625**
1. APPLICABILITY AND DEFINITIONS . . . 625
2. VERBATIM COPYING . . . 627
3. COPYING IN QUANTITY . . . 627
4. MODIFICATIONS . . . 628
5. COMBINING DOCUMENTS . . . 629
6. COLLECTIONS OF DOCUMENTS . . . 630
7. AGGREGATION WITH INDEPENDENT WORKS . . . 630
8. TRANSLATION . . . 630
9. TERMINATION . . . 631
10. FUTURE REVISIONS OF THIS LICENSE . . . 631
11. RELICENSING . . . 631
ADDENDUM: How to use this License for your documents . . . 632

www.dbooks.org

Part I

Preface

www.dbooks.org

0.1 Why?

'Are algorithms useful?'. Some programmers say that they seldom use any serious data structures or algorithms in real work such as commercial application development. Even when they need some of them, they have already been provided by libraries. For example, the C++ standard template library (STL) provides sort and selection algorithms as well as the vector, queue, and set data structures. It seems that knowing about how to use the library as a tool is quite enough.

Instead of answering this question directly, I would like to say algorithms and data structures are critical in solving 'interesting problems', the usefulness of the problem set aside.

Let's start with two problems that looks like they can be solved in a brute-force way even by a fresh programmer.

0.2 The smallest free ID problem, the power of algorithms

This problem is discussed in Chapter 1 of Richard Bird's book [1]. It's common that applications and systems use ID (identifier) to manage objects and entities. At any time, some IDs are used, and some of them are available for use. When some client tries to acquire a new ID, we want to always allocate it the smallest available one. Suppose IDs are non-negative integers and all IDs in use are kept in a list (or an array) which is not ordered. For example:

```
[18, 4, 8, 9, 16, 1, 14, 7, 19, 3, 0, 5, 2, 11, 6]
```

How can you find the smallest free ID, which is 10, from the list?

It seems the solution is quite easy even without any serious algorithms.

```
1: function MIN-FREE(A)
2:     x ← 0
3:     loop
4:         if x ∉ A then
5:             return x
6:         else
7:             x ← x + 1
```

Where the $\notin$ is realized like below.

```
1: function '∉'(x, X)
2:     for i ← 1 to |X| do
3:         if x = X[i] then
4:             return False
5:     return True
```

Some languages provide handy tools which wrap this linear time process. For example in Python, this algorithm can be directly translated as the following.

```
def brute_force(lst):
    i = 0
    while True:
        if i not in lst:
```

www.dbooks.org

```
            return i
        i = i + 1
```

It seems this problem is trivial. However, There will be millions of IDs in a large system. The speed of this solution is poor in such case for it takes $O(n^2)$ time, where n is the length of the ID list. In my computer (2 Cores 2.10 GHz, with 2G RAM), a C program using this solution takes an average of 5.4 seconds to search a minimum free number among 100,000 IDs[1]. And it takes more than 8 minutes to handle a million numbers.

0.2.1 Improvement 1

The key idea to improve the solution is based on a fact that for a series of n numbers $x_1, x_2, ..., x_n$, if there are free numbers, some of the x_i are outside the range $[0, n)$; otherwise the list is exactly a permutation of $0, 1, ..., n-1$ and n should be returned as the minimum free number. We have the following fact.

$$minfree(x_1, x_2, ..., x_n) \leq n \tag{1}$$

One solution is to use an array of $n+1$ flags to mark whether a number in range $[0, n]$ is free.

```
1: function MIN-FREE(A)
2:     F ← [False, False, ..., False] where |F| = n + 1
3:     for ∀x ∈ A do
4:         if x < n then
5:             F[x] ← True
6:     for i ← [0, n] do
7:         if F[i] = False then
8:             return i
```

Line 2 initializes a flag array all of False values. This takes $O(n)$ time. Then the algorithm scans all numbers in A and mark the relative flag to True if the value is less than n, This step also takes $O(n)$ time. Finally, the algorithm performs a linear time search to find the first flag with False value. So the total performance of this algorithm is $O(n)$. Note that we use $n+1$ flags instead of n flags to cover the special case that $sorted(A) = [0, 1, 2, ..., n-1]$.

Although the algorithm only takes $O(n)$ time, it needs extra $O(n)$ spaces to store the flags.

This solution is much faster than the brute force one. On my computer, the relevant Python program takes an average of 0.02 second when dealing with 100,000 numbers.

We haven't fine tuned this algorithm yet. Observe that each time we have to allocate memory to create a $n+1$ elements array of flags, and release the memory when finished. The memory allocation and release is very expensive thus they cost us a lot of processing time.

There are two ways in which we can improve on this solution. One is to allocate the flags array in advance and reuse it for all the calls of our function to find the smallest free number. The other is to use bit-wise flags instead of a flag array. The following is the C program based on these two minor improvements.

[1] All programs can be downloaded along with this series posts.

```
#define N 1000000 // 1 million
#define WORD_LENGTH (sizeof(int) * 8)

void setbit(unsigned int* bits, unsigned int i){
  bits[i / WORD_LENGTH] |= 1<<(i % WORD_LENGTH);
}

int testbit(unsigned int* bits, unsigned int i){
  return bits[i/WORD_LENGTH] & (1<<(i % WORD_LENGTH));
}

unsigned int bits[N/WORD_LENGTH+1];

int min_free(int* xs, int n){
  int i, len = N/WORD_LENGTH+1;
  for(i=0; i<len; ++i)
    bits[i]=0;
  for(i=0; i<n; ++i)
    if(xs[i]<n)
      setbit(bits, xs[i]);
  for(i=0; i<=n; ++i)
    if(!testbit(bits, i))
      return i;
}
```

This C program can handle 1,000,000 (1 million) IDs in just 0.023 second on my computer.

The last for-loop can be further improved as seen below but this is just minor fine-tuning.

```
  for(i=0; ; ++i)
    if(~bits[i] !=0 )
      for(j=0; ; ++j)
        if(!testbit(bits, i*WORD_LENGTH+j))
          return i*WORD_LENGTH+j;
```

0.2.2 Improvement 2, Divide and Conquer

Although the above improvement is much faster, it costs $O(n)$ extra spaces to keep a check list. if n is huge number this means a huge amount of space is wasted.

The typical divide and conquer strategy is to break the problem into some smaller ones, and solve these to get the final answer.

We can put all numbers $x_i \leq \lfloor n/2 \rfloor$ as a sub-list A' and put all the others as a second sub-list A''. Based on formula 1 if the length of A' is exactly $\lfloor n/2 \rfloor$, this means the first half of numbers are 'full', which indicates that the minimum free number must be in A'' and so we'll need to recursively seek in the shorter list A''. Otherwise, it means the minimum free number is located in A', which again leads to a smaller problem.

When we search the minimum free number in A'', the conditions changes a little bit, we are not searching the smallest free number starting from 0, but

www.dbooks.org

actually from $\lfloor n/2 \rfloor + 1$ as the lower bound. So the algorithm is something like $minfree(A, l, u)$, where l is the lower bound and u is the upper bound index of the element.

Note that there is a trivial case, that if the number list is empty, we merely return the lower bound as the result.

This divide and conquer solution can be formally expressed as a function :

$$minfree(A) = search(A, 0, |A| - 1)$$

$$search(A, l, u) = \begin{cases} l & : \quad A = \phi \\ search(A'', m + 1, u) & : \quad |A'| = m - l + 1 \\ search(A', l, m) & : \quad otherwise \end{cases}$$

where

$$m = \lfloor \frac{l + u}{2} \rfloor$$
$$A' = \{\forall x \in A \wedge x \leq m\}$$
$$A'' = \{\forall x \in A \wedge x > m\}$$

It is obvious that this algorithm doesn't need any extra space[2]. Each call performs $O(|A|)$ comparison to build A' and A''. After that the problem scale halves. So the time needed for this algorithm is $T(n) = T(n/2) + O(n)$ which reduce to $O(n)$. Another way to analyze the performance is by observing that the first call takes $O(n)$ to build A' and A'' and the second call takes $O(n/2)$, and $O(n/4)$ for the third... The total time is $O(n + n/2 + n/4 + ...) = O(2n) = O(n)$.

In functional programming languages such as Haskell, partitioning a list has already been provided in the basic library and this algorithm can be translated as the following.

```
import Data.List

minFree xs = bsearch xs 0 (length xs - 1)

bsearch xs l u | xs == [] = l
               | length as == m - l + 1 = bsearch bs (m+1) u
               | otherwise = bsearch as l m
    where
      m = (l + u) `div` 2
      (as, bs) = partition (≤m) xs
```

0.2.3 Expressiveness vs. Performance

Imperative language programmers may be concerned about the performance of this kind of implementation. For instance in this minimum free ID problem, the number of recursive calls is in $O(\lg n)$, which means the stack size consumed is in $O(\lg n)$. It's not free in terms of space. But if we want to avoid that , we

[2]Procedural programmer may note that it actually takes $O(\lg n)$ stack spaces for book-keeping. As we'll see later, this can be eliminated either by tail recursion optimization, for instance gcc -O2. or by manually changing the recursion to iteration

can eliminate the recursion by replacing it by an iteration [3] which yields the following C program.

```
int min_free(int* xs, int n){
  int l=0;
  int u=n-1;
  while(n){
    int m = (l + u) / 2;
    int right, left = 0;
    for(right = 0; right < n; ++ right)
      if(xs[right] <= m){
        swap(xs[left], xs[right]);
        ++left;
      }
    if(left == m - l + 1){
      xs = xs + left;
      n  = n - left;
      l  = m+1;
    }
    else{
      n = left;
      u = m;
    }
  }
  return l;
}
```

This program uses a 'quick-sort' like approach to re-arrange the array so that all the elements before $left$ are less than or equal to m; while those between $left$ and $right$ are greater than m. This is shown in figure 1.

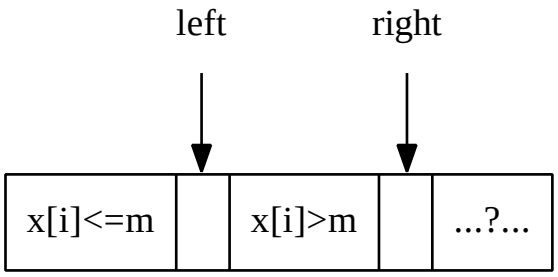


Figure 1: Divide the array, all $x[i] \leq m$ where $0 \leq i < left$; while all $x[i] > m$ where $left \leq i < right$. The left elements are unknown.

This program is fast and it doesn't need extra stack space. However, compared to the previous Haskell program, it's hard to read and the expressiveness decreased. We have to balance performance and expressiveness.

[3] This is done automatically in most functional languages since our function is in tail recursive form which lends itself perfectly to this transformation

www.dbooks.org

0.3 The number puzzle, power of data structure

If the first problem, to find the minimum free number, is a some what useful in practice, this problem is a 'pure' one for fun. The puzzle is to find the 1,500th number, which only contains factor 2, 3 or 5. The first 3 numbers are of course 2, 3, and 5. Number $60 = 2^2 3^1 5^1$, However it is the 25th number. Number $21 = 2^0 3^1 7^1$, isn't a valid number because it contains a factor 7. The first 10 such numbers are list as the following.

2,3,4,5,6,8,9,10,12,15

If we consider $1 = 2^0 3^0 5^0$, then 1 is also a valid number and it is the first one.

0.3.1 The brute-force solution

It seems the solution is quite easy without need any serious algorithms. We can check all numbers from 1, then extract all factors of 2, 3 and 5 to see if the left part is 1.

```
1: function GET-NUMBER(n)
2:     x ← 1
3:     i ← 0
4:     loop
5:         if VALID?(x) then
6:             i ← i + 1
7:             if i = n then
8:                 return x
9:         x ← x + 1

10: function VALID?(x)
11:     while x mod 2 = 0 do
12:         x ← x/2
13:     while x mod 3 = 0 do
14:         x ← x/3
15:     while x mod 5 = 0 do
16:         x ← x/5
17:     if x = 1 then
18:         return True
19:     else
20:         return False
```

This 'brute-force' algorithm works for most small n. However, to find the 1500th number (which is 859963392), the C program based on this algorithm takes 40.39 seconds in my computer. I have to kill the program after 10 minutes when I increased n to 15,000.

0.3.2 Improvement 1

Analysis of the above algorithm shows that modular and divide calculations are very expensive [2]. And they executed a lot in loops. Instead of checking a number contains only 2, 3, or 5 as factors, one alternative solution is to construct such number by these factors.

We start from 1, and times it with 2, or 3, or 5 to generate rest numbers. The problem turns to be how to generate the candidate number in order? One handy way is to utilize the queue data structure.

A queue data structure is used to push elements at one end, and pops them at the other end. So that the element be pushed first is also be popped out first. This property is called FIFO (First-In-First-Out).

The idea is to push 1 as the only element to the queue, then we pop an element, times it with 2, 3, and 5, to get 3 new elements. We then push them back to the queue in order. Note that, the new elements may have already existed in the queue. In such case, we just drop the element. The new element may also smaller than the others in the queue, so we must put them to the correct position. Figure 2 illustrates this idea.

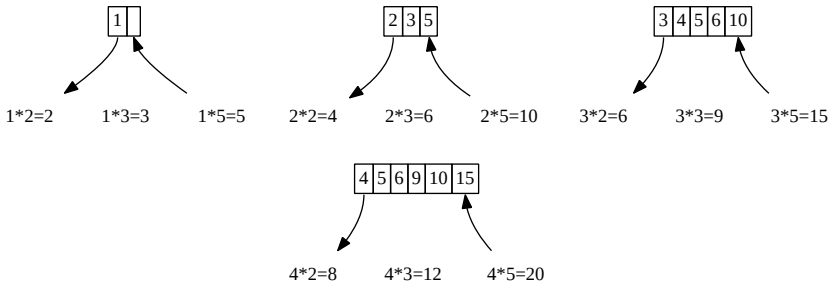


Figure 2: First 4 steps of constructing numbers with a queue.
1. Queue is initialized with 1 as the only element;
2. New elements 2, 3, and 5 are pushed back;
3. New elements 4, 6, and 10, are pushed back in order;
4. New elements 9 and 15 are pushed back, element 6 already exists.

This algorithm is shown as the following.

```
1:  function GET-NUMBER(n)
2:      Q ← NIL
3:      ENQUEUE(Q, 1)
4:      while n > 0 do
5:          x ← DEQUEUE(Q)
6:          UNIQUE-ENQUEUE(Q, 2x)
7:          UNIQUE-ENQUEUE(Q, 3x)
8:          UNIQUE-ENQUEUE(Q, 5x)
9:          n ← n − 1
10:     return x

11: function UNIQUE-ENQUEUE(Q, x)
12:     i ← 0
13:     while i < |Q| ∧ Q[i] < x do
14:         i ← i + 1
15:     if i < |Q| ∧ x = Q[i] then
16:         return
17:     INSERT(Q, i, x)
```

www.dbooks.org

The insert function takes $O(|Q|)$ time to find the proper position and insert it. If the element has already existed, it just returns.

A rough estimation tells that the length of the queue increase proportion to n, (Each time, we extract one element, and pushed 3 new, the increase ratio ≤ 2), so the total running time is $O(1 + 2 + 3 + ... + n) = O(n^2)$.

Figure3 shows the number of queue access time against n. It is quadratic curve which reflect the $O(n^2)$ performance.

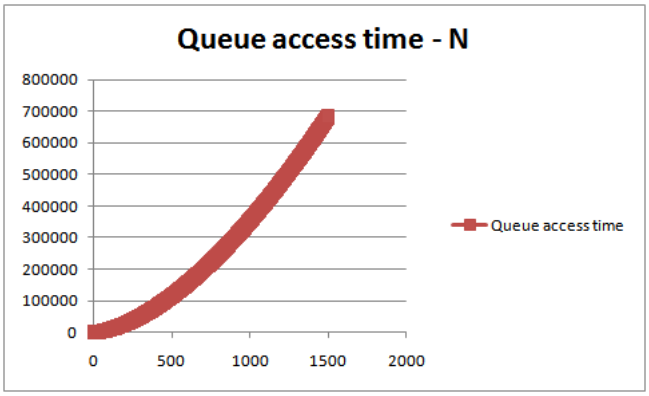


Figure 3: Queue access count v.s. n.

The C program based on this algorithm takes only 0.016[s] to get the right answer 859963392. Which is 2500 times faster than the brute force solution.

Improvement 1 can also be considered in recursive way. Suppose X is the infinity series for all numbers which only contain factors of 2, 3, or 5. The following formula shows an interesting relationship.

$$X = \{1\} \cup \{2x : \forall x \in X\} \cup \{3x : \forall x \in X\} \cup \{5x : \forall x \in X\} \tag{2}$$

Where we can define $\cup$ to a special form so that all elements are stored in order as well as unique to each other. Suppose that $X = \{x_1, x_2, x_3...\}$, $Y = \{y_1, y_2, y_3, ...\}$, $X' = \{x_2, x_3, ...\}$ and $Y' = \{y_2, y_3, ...\}$. We have

$$X \cup Y = \begin{cases} X & : \quad Y = \phi \\ Y & : \quad X = \phi \\ \{x_1, X' \cup Y\} & : \quad x_1 < y_1 \\ \{x_1, X' \cup Y'\} & : \quad x_1 = y_1 \\ \{y_1, X \cup Y'\} & : \quad x_1 > y_1 \end{cases}$$

In a functional programming language such as Haskell, which supports lazy evaluation, The above infinity series functions can be translate into the following program.

```
ns = 1:merge (map (*2) ns) (merge (map (*3) ns) (map (*5) ns))

merge [] l = l
merge l [] = l
merge (x:xs) (y:ys) | x <y = x : merge xs (y:ys)
                    | x ==y = x : merge xs ys
                    | otherwise = y : merge (x:xs) ys
```

By evaluate ns !! (n-1), we can get the 1500th number as below.

```
>ns !! (1500-1)
859963392
```

0.3.3 Improvement 2

Considering the above solution, although it is much faster than the brute-force one, It still has some drawbacks. It produces many duplicated numbers and they are finally dropped when examine the queue. Secondly, it does linear scan and insertion to keep the order of all elements in the queue, which degrade the ENQUEUE operation from $O(1)$ to $O(|Q|)$.

If we use three queues instead of using only one, we can improve the solution one step ahead. Denote these queues as Q_2, Q_3, and Q_5, and we initialize them as $Q_2 = \{2\}$, $Q_3 = \{3\}$ and $Q_5 = \{5\}$. Each time we DEQUEUEed the smallest one from Q_2, Q_3, and Q_5 as x. And do the following test:

- If x comes from Q_2, we ENQUEUE $2x$, $3x$, and $5x$ back to Q_2, Q_3, and Q_5 respectively;
- If x comes from Q_3, we only need ENQUEUE $3x$ to Q_3, and $5x$ to Q_5; We needn't ENQUEUE $2x$ to Q_2, because $2x$ have already existed in Q_3;
- If x comes from Q_5, we only need ENQUEUE $5x$ to Q_5; there is no need to ENQUEUE $2x$, $3x$ to Q_2, Q_3 because they have already been in the queues;

We repeatedly ENQUEUE the smallest one until we find the n-th element. The algorithm based on this idea is implemented as below.

```
1:  function GET-NUMBER(n)
2:      if n = 1 then
3:          return 1
4:      else
5:          Q2 ← {2}
6:          Q3 ← {3}
7:          Q5 ← {5}
8:          while n > 1 do
9:              x ← min(HEAD(Q2), HEAD(Q3), HEAD(Q5))
10:             if x = HEAD(Q2) then
11:                 DEQUEUE(Q2)
12:                 ENQUEUE(Q2, 2x)
13:                 ENQUEUE(Q3, 3x)
14:                 ENQUEUE(Q5, 5x)
15:             else if x = HEAD(Q3) then
16:                 DEQUEUE(Q3)
17:                 ENQUEUE(Q3, 3x)
18:                 ENQUEUE(Q5, 5x)
19:             else
20:                 DEQUEUE(Q5)
21:                 ENQUEUE(Q5, 5x)
22:             n ← n − 1
```

www.dbooks.org

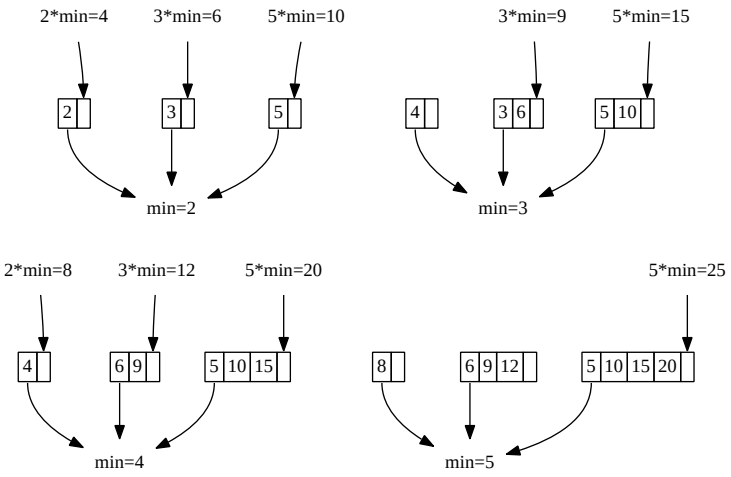


Figure 4: First 4 steps of constructing numbers with Q_2, Q_3, and Q_5.
1. Queues are initialized with 2, 3, 5 as the only element;
2. New elements 4, 6, and 10 are pushed back;
3. New elements 9, and 15, are pushed back;
4. New elements 8, 12, and 20 are pushed back;
5. New element 25 is pushed back.

23: **return** x

This algorithm loops n times, and within each loop, it extract one head element from the three queues, which takes constant time. Then it appends one to three new elements at the end of queues which bounds to constant time too. So the total time of the algorithm bounds to $O(n)$. The C++ program translated from this algorithm shown below takes less than 1 μs to produce the 1500th number, 859963392.

```
typedef unsigned long Integer;

Integer get_number(int n){
  if(n==1)
    return 1;
  queue<Integer> Q2, Q3, Q5;
  Q2.push(2);
  Q3.push(3);
  Q5.push(5);
  Integer x;
  while(n-- > 1){
    x = min(min(Q2.front(), Q3.front()), Q5.front());
    if(x==Q2.front()){
      Q2.pop();
      Q2.push(x*2);
      Q3.push(x*3);
      Q5.push(x*5);
    }
```

```
    else if(x==Q3.front()){
      Q3.pop();
      Q3.push(x*3);
      Q5.push(x*5);
    }
    else{
      Q5.pop();
      Q5.push(x*5);
    }
  }
  return x;
}
```

This solution can be also implemented in Functional way. We define a function $take(n)$, which will return the first n numbers contains only factor 2, 3, or 5.

$$take(n) = f(n, \{1\}, \{2\}, \{3\}, \{5\})$$

Where

$$f(n, X, Q_2, Q_3, Q_5) = \begin{cases} X & : \quad n = 1 \\ f(n-1, X \cup \{x\}, Q_2', Q_3', Q_5') & : \quad otherwise \end{cases}$$

$$x = min(Q_{21}, Q_{31}, Q_{51})$$

$$Q_2', Q_3', Q_5' = \begin{cases} \{Q_{22}, Q_{23}, ...\} \cup \{2x\}, Q_3 \cup \{3x\}, Q_5 \cup \{5x\} & : \quad x = Q_{21} \\ Q_2, \{Q_{32}, Q_{33}, ...\} \cup \{3x\}, Q5 \cup \{5x\} & : \quad x = Q_{31} \\ Q_2, Q_3, \{Q_{52}, Q_{53}, ...\} \cup \{5x\} & : \quad x = Q_{51} \end{cases}$$

And these functional definition can be realized in Haskell as the following.

```
ks 1 xs _ = xs
ks n xs (q2, q3, q5) = ks (n-1) (xs++[x]) update
    where
      x = minimum $ map head [q2, q3, q5]
      update | x == head q2 = ((tail q2)++[x*2], q3++[x*3], q5++[x*5])
             | x == head q3 = (q2, (tail q3)++[x*3], q5++[x*5])
             | otherwise = (q2, q3, (tail q5)++[x*5])

takeN n = ks n [1] ([2], [3], [5])
```

Invoke 'last takeN 1500' will generate the correct answer 859963392.

0.4 Notes and short summary

If review the 2 puzzles, we found in both cases, the brute-force solutions are so weak. In the first problem, it's quite poor in dealing with long ID list, while in the second problem, it doesn't work at all.

The first problem shows the power of algorithms, while the second problem tells why data structure is important. There are plenty of interesting problems, which are hard to solve before computer was invented. With the aid of computer and programming, we are able to find the answer in a quite different way.

www.dbooks.org

Compare to what we learned in mathematics course in school, we haven't been taught the method like this.

While there have been already a lot of wonderful books about algorithms, data structures and math, however, few of them provide the comparison between the procedural solution and the functional solution. From the above discussion, it can be found that functional solution sometimes is very expressive and they are close to what we are familiar in mathematics.

This series of post focus on providing both imperative and functional algorithms and data structures. Many functional data structures can be referenced from Okasaki's book[6]. While the imperative ones can be founded in classic text books [2] or even in WIKIpedia. Multiple programming languages, including, C, C++, Python, Haskell, and Scheme/Lisp will be used. In order to make it easy to read by programmers with different background, pseudo code and mathematical function are the regular descriptions of each post.

The author is NOT a native English speaker, the reason why this book is only available in English for the time being is because the contents are still changing frequently. Any feedback, comments, or criticizes are welcome.

0.5 Structure of the contents

In the following series of post, I'll first introduce about elementary data structures before algorithms, because many algorithms need knowledge of data structures as prerequisite.

The 'hello world' data structure, binary search tree is the first topic; Then we introduce how to solve the balance problem of binary search tree. After that, I'll show other interesting trees. Trie, and Prefix trees are useful in text manipulation. While B-trees are commonly used in file system and data base implementation.

The second part of data structures is about heaps. We'll provide a general Heap definition and introduce about binary heaps by array and by explicit binary trees. Then we'll extend to K-ary heaps including Binomial heaps, Fibonacci heaps, and pairing heaps.

Array and queues are considered among the easiest data structures typically, However, we'll show how difficult to implement them in the third part.

As the elementary sort algorithms, we'll introduce insertion sort, quick sort, merge sort etc in both imperative way and functional way.

The final part is about searching, besides the element searching, we'll also show string matching algorithms such as KMP.

Bibliography

[1] Richard Bird. “Pearls of functional algorithm design”. Cambridge University Press; 1 edition (November 1, 2010). ISBN-10: 0521513383

[2] Jon Bentley. “Programming Pearls(2nd Edition)”. Addison-Wesley Professional; 2 edition (October 7, 1999). ISBN-13: 978-0201657883

[3] Chris Okasaki. “Purely Functional Data Structures”. Cambridge university press, (July 1, 1999), ISBN-13: 978-0521663502

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. The MIT Press, 2001. ISBN: 0262032937.

www.dbooks.org

Part II

Trees

www.dbooks.org

Chapter 1

Binary search tree, the 'hello world' data structure

1.1 Introduction

Arrays or lists are typically considered the 'hello world' data structures. However, we'll see they are not actually particularly easy to implement. In some procedural settings, arrays are the most elementary data structures, and it is possible to implement linked lists using arrays (see section 10.3 in [2]). On the other hand, in some functional settings, linked lists are the elementary building blocks used to create arrays and other data structures.

Considering these factors, we start with Binary Search Trees (or BST) as the 'hello world' data structure using an interesting problem Jon Bentley mentioned in 'Programming Pearls' [2]. The problem is to count the number of times each word occurs in a large text. One solution in C++ is below:

```
int main(int, char** ){
  map<string, int> dict;
  string s;
  while(cin>>s)
    ++dict[s];
  map<string, int>::iterator it=dict.begin();
  for(; it!=dict.end(); ++it)
    cout<<it->first<<": "<<it->second<<"\n";
}
```

And we can run it to produce the result using the following UNIX commands [1].

```
$ g++ wordcount.cpp -o wordcount
$ cat bbe.txt | ./wordcount > wc.txt
```

The map provided in the standard template library is a kind of balanced BST with augmented data. Here we use the words in the text as the keys and the number of occurrences as the augmented data. This program is fast, and

[1] This is not a UNIX unique command, in Windows OS, it can be achieved by: `type bbe.txt | wordcount.exe > wc.txt`

www.dbooks.org

it reflects the power of BSTs. We'll introduce how to implement BSTs in this section and show how to balance them in a later section.

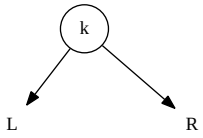
Before we dive into BSTs, let's first introduce the more general binary tree.

Binary trees are recursively defined. BSTs are just one type of binary tree. A binary tree is usually defined in the following way.

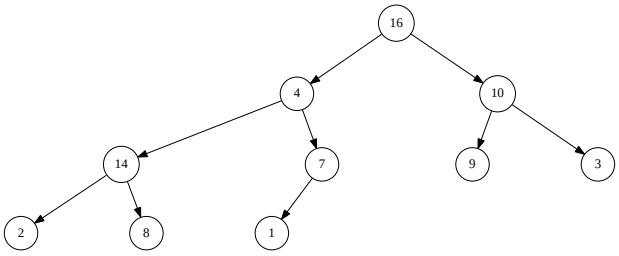
A binary tree is

- either an empty node;
- or a node containing 3 parts: a value, a left child which is a binary tree and a right child which is also a binary tree.

Figure 1.1 shows this concept and an example binary tree.



(a) Concept of binary tree



(b) An example binary tree

Figure 1.1: Binary tree concept and an example.

A BST is a binary tree where the following applies to each node:

- all the values in left child tree are less than the value of this node;
- the value of this node is less than any values in its right child tree.

Figure 1.2 shows an example of a BST. Comparing with Figure 1.1, we can see the differences in how keys are ordered between them.

1.2 Data Layout

Based on the recursive definition of BSTs, we can draw the data layout in a procedural setting with pointers as in Figure 1.3.

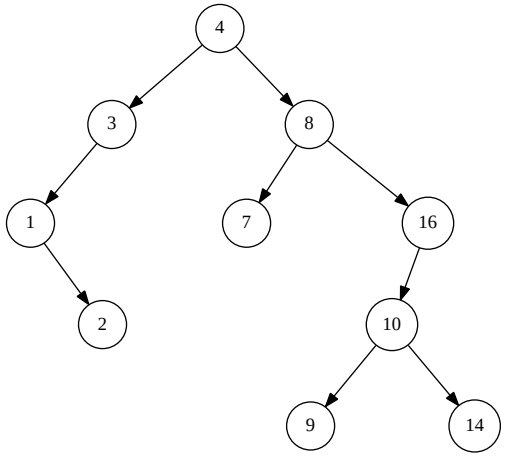


Figure 1.2: An example of a BST

The node first contains a field for the key, which can be augmented with satellite data. The next two fields contain pointers to the left and right children, respectively. To make backtracking to ancestors easy, a parent field is sometimes provided as well.

In this section, we'll ignore the satellite data for the sake of simplifying the illustrations. Based on this layout, the node of BST can be defined in a procedural language, such as C++:

```
template<class T>
struct node{
  node(T x):key(x), left(0), right(0), parent(0){}
  ~node(){
    delete left;
    delete right;
  }

  node* left;
  node* right;
  node* parent; //Optional, it's helpful for succ and pred
  T key;
};
```

There is another setting, for instance in Scheme/Lisp languages, the elementary data structure is a linked list. Figure 1.4 shows how a BST node can be built on top of linked list.

In more functional settings, it's hard to use pointers for backtracking (and typically, there is no need for backtracking, since there are usually top-down recursive solutions), and so the 'parent' field has been omitted in that layout.

To simplify things, we'll skip the detailed layouts in the future and only focus on the logic layouts of data structures. For example, below is the definition of

www.dbooks.org

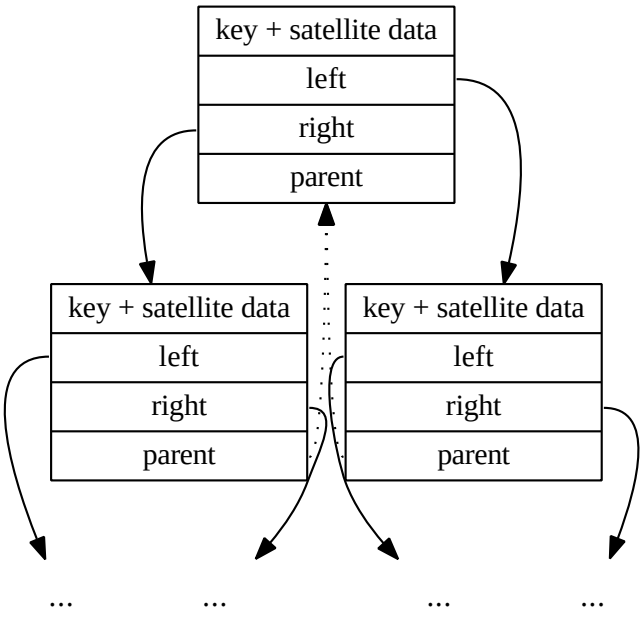


Figure 1.3: Layout of nodes with parent field.

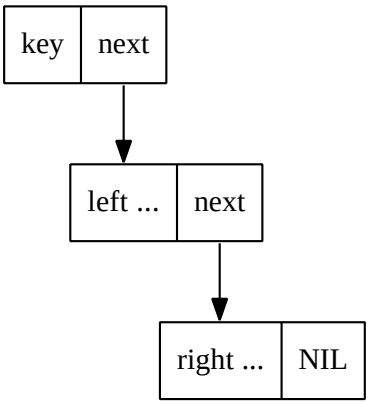


Figure 1.4: Binary search tree node layout on top of linked list. Where 'left...' and 'right ...' are either empty or BST nodes composed in the same way.

a BST node in Haskell:

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

1.3 Insertion

To insert a key k (sometimes along with a value in practice) to a BST T, we can use the following algorithm:

- If the tree is empty, construct a leaf node with key $= k$;
- If k is less than the key of root node, insert it in the left child;
- If k is greater than the key of root node, insert it in the right child.

The exception to the above is when k is equal to the key of the root node, meaning it already exists in the BST, and we can either overwrite the data, or just do nothing. To simplify things, this case has been skipped in this section.

This algorithm is described recursively. It's simplicity is why we consider the BST structure the 'hello world' data structure. Formally, the algorithm can be represented with a recursive mathematical function:

$$insert(T, k) = \begin{cases} node(\phi, k, \phi) & : \quad T = \phi \\ node(insert(T_l, k), k', T_r) & : \quad k < k' \\ node(T_l, k', insert(T_r, k)) & : \quad otherwise \end{cases} \tag{1.1}$$

Where T_l is the left child, T_r is the right child, and k' is the key when T isn't empty.

The node function creates a new node given the left subtree, a key and a right subtree as parameters. ϕ means NIL or empty.

Translating the above functions directly to Haskell yields the following program:

```
insert Empty k = Node Empty k Empty
insert (Node l x r) k | k < x = Node (insert l k) x r
                      | otherwise = Node l x (insert r k)
```

This program utilized the pattern matching features provided by the language. However, even in functional settings without this feature (e.g. Scheme/Lisp) the program is still expressive:

```
(define (insert tree x)
  (cond ((null? tree) (list '() x '()))
        ((< x (key tree))
         (make-tree (insert (left tree) x)
                    (key tree)
                    (right tree)))
        ((> x (key tree))
         (make-tree (left tree)
                    (key tree)
                    (insert (right tree) x)))))
```

This algorithm can be expressed imperatively using iteration, completely free of recursion:

www.dbooks.org

```
1: function INSERT(T, k)
2:     root ← T
3:     x ← CREATE-LEAF(k)
4:     parent ← NIL
5:     while T ≠ NIL do
6:         parent ← T
7:         if k < KEY(T) then
8:             T ← LEFT(T)
9:         else
10:            T ← RIGHT(T)
11:    PARENT(x) ← parent
12:    if parent = NIL then                          ▷ tree T is empty
13:        return x
14:    else if k < KEY(parent) then
15:        LEFT(parent) ← x
16:    else
17:        RIGHT(parent) ← x
18:    return root

19: function CREATE-LEAF(k)
20:    x ← EMPTY-NODE
21:    KEY(x) ← k
22:    LEFT(x) ← NIL
23:    RIGHT(x) ← NIL
24:    PARENT(x) ← NIL
25:    return x
```

While more complex than the functional algorithm, it is still fast, even when presented with very deep trees. Complete C++ and python programs are available along with this section for reference.

1.4 Traversing

Traversing means visiting every element one-by-one in a BST. There are 3 ways to traverse a binary tree: a pre-order tree walk, an in-order tree walk and a post-order tree walk. The names of these traversal methods highlight the order in which we visit the root of a BST.

- pre-order traversal:, visit the key, then the left child, finally the right child;
- in-order traversal: visit the left child, then the key, finally the right child;
- post-order traversal: visit the left child, then the right child, finally the key.

Note that each 'visiting' operation is recursive. As mentioned before, we see that the order in which the key is visited determines the name of the traversal method.

For the BST shown in figure 1.2, below are the three different traversal results.

- pre-order traversal results: 4, 3, 1, 2, 8, 7, 16, 10, 9, 14;
- in-order traversal results: 1, 2, 3, 4, 7, 8, 9, 10, 14, 16;
- post-order traversal results: 2, 1, 3, 7, 9, 14, 10, 16, 8, 4.

The in-order walk of a BST outputs the elements in increasing order. The definition of a BST ensures this interesting property, while the proof of this fact is left as an exercise to the reader.

The in-order tree walk algorithm can be described as:

- If the tree is empty, just return;
- traverse the left child by in-order walk, then access the key, finally traverse the right child by in-order walk.

Translating the above description yields a generic map function:

$$map(f, T) = \begin{cases} \phi & : \quad T = \phi \\ node(T'_l, k', T'_r) & : \quad otherwise \end{cases} \tag{1.2}$$

Where

$$\begin{aligned} T'_l &= map(f, T_l) \\ T'_r &= map(f, T_r) \\ k' &= f(k) \end{aligned}$$

And T_l, T_r and k are the children and key when the tree isn't empty.

If we only need access the key without create the transformed tree, we can realize this algorithm in procedural way lie the below C++ program.

```
template<class T, class F>
void in_order_walk(node<T>* t, F f){
  if(t){
    in_order_walk(t->left, f);
    f(t->value);
    in_order_walk(t->right, f);
  }
}
```

The function takes a parameter f, it can be a real function, or a function object, this program will apply f to the node by in-order tree walk.

We can simplified this algorithm one more step to define a function which turns a BST to a sorted list by in-order traversing.

$$toList(T) = \begin{cases} \phi & : \quad T = \phi \\ toList(T_l) \cup \{k\} \cup toList(T_r) & : \quad otherwise \end{cases} \tag{1.3}$$

Below is the Haskell program based on this definition.

```
toList Empty = []
toList (Node l x r) = toList l ++ [x] ++ toList r
```

www.dbooks.org

This provides us a method to sort a list of elements. We can first build a BST from the list, then output the tree by in-order traversing. This method is called as 'tree sort'. Let's denote the list $X = \{x_1, x_2, x_3, ..., x_n\}$.

$$sort(X) = toList(fromList(X)) \tag{1.4}$$

And we can write it in function composition form.

$$sort = toList.fromList$$

Where function $fromList$ repeatedly insert every element to an empty BST.

$$fromList(X) = foldL(insert, \phi, X) \tag{1.5}$$

It can also be written in partial application form[2] like below.

$$fromList = foldL \quad insert \quad \phi$$

For the readers who are not familiar with folding from left, this function can also be defined recursively as the following.

$$fromList(X) = \begin{cases} \phi & : \quad X = \phi \\ insert(fromList(\{x_2, x_3, ..., x_n\}), x_1) & : \quad otherwise \end{cases}$$

We'll intense use folding function as well as the function composition and partial evaluation in the future, please refer to appendix of this book or [6] [7] and [8] for more information.

Exercise 1.1

- Given the in-order traverse result and pre-order traverse result, can you re-construct the tree from these result and figure out the post-order traversing result?
  - Pre-order result: 1, 2, 4, 3, 5, 6;
  - In-order result: 4, 2, 1, 5, 3, 6;
  - Post-order result: ?
- Write a program in your favorite language to re-construct the binary tree from pre-order result and in-order result.
- Prove why in-order walk output the elements stored in a binary search tree in increase order?
- Can you analyze the performance of tree sort with big-O notation?

[2] Also known as 'Curried form' to memorialize the mathematician and logician Haskell Curry.

1.5 Querying a binary search tree

There are three types of querying for binary search tree, searching a key in the tree, find the minimum or maximum element in the tree, and find the predecessor or successor of an element in the tree.

1.5.1 Looking up

According to the definition of binary search tree, search a key in a tree can be realized as the following.

- If the tree is empty, the searching fails;
- If the key of the root is equal to the value to be found, the search succeed. The root is returned as the result;
- If the value is less than the key of the root, search in the left child.
- Else, which means that the value is greater than the key of the root, search in the right child.

This algorithm can be described with a recursive function as below.

$$lookup(T, x) = \begin{cases} \phi & : & T = \phi \\ T & : & k = x \\ lookup(T_l, x) & : & x < k \\ lookup(T_r, x) & : & otherwise \end{cases} \tag{1.6}$$

Where T_l, T_r and k are the children and key when T isn't empty. In the real application, we may return the satellite data instead of the node as the search result. This algorithm is simple and straightforward. Here is a translation of Haskell program.

```
lookup Empty _ = Empty
lookup t@(Node l k r) x | k == x = t
                        | x < k = lookup l x
                        | otherwise = lookup r x
```

If the BST is well balanced, which means that almost all nodes have both non-NIL left child and right child, for n elements, the search algorithm takes $O(\lg n)$ time to perform. This is not formal definition of balance. We'll show it in later post about red-black-tree. If the tree is poor balanced, the worst case takes $O(n)$ time to search for a key. If we denote the height of the tree as h, we can uniform the performance of the algorithm as $O(h)$.

The search algorithm can also be realized without using recursion in a procedural manner.

```
1: function SEARCH(T, x)
2:     while T ≠ NIL ∧ KEY(T) ≠ x do
3:         if x < KEY(T) then
4:             T ← LEFT(T)
5:         else
6:             T ← RIGHT(T)
7:     return T
```

www.dbooks.org

Below is the C++ program based on this algorithm.

```
template<class T>
node<T>* search(node<T>* t, T x){
  while(t && t->key!=x){
    if(x < t->key) t=t->left;
    else t=t->right;
  }
  return t;
}
```

1.5.2 Minimum and maximum

Minimum and maximum can be implemented from the property of binary search tree, less keys are always in left child, and greater keys are in right.

For minimum, we can continue traverse the left sub tree until it is empty. While for maximum, we traverse the right.

$$min(T) = \begin{cases} k & : & T_l = \phi \\ min(T_l) & : & otherwise \end{cases} \tag{1.7}$$

$$max(T) = \begin{cases} k & : & T_r = \phi \\ max(T_r) & : & otherwise \end{cases} \tag{1.8}$$

Both functions bound to $O(h)$ time, where h is the height of the tree. For the balanced BST, min/max are bound to $O(\lg n)$ time, while they are $O(n)$ in the worst cases.

We skip translating them to programs, It's also possible to implement them in pure procedural way without using recursion.

1.5.3 Successor and predecessor

The last kind of querying is to find the successor or predecessor of an element. It is useful when a tree is treated as a generic container and traversed with iterator. We need access the parent of a node to make the implementation simple.

It seems hard to find the functional solution, because there is no pointer like field linking to the parent node[3]. One solution is to left 'breadcrumbs' when we visit the tree, and use these information to back-track or even re-construct the whole tree. Such data structure, that contains both the tree and 'breadcrumbs' is called zipper. please refer to [9] for details.

However, If we consider the original purpose of providing *succ/pred* function, 'to traverse all the BST elements one by one' as a generic container, we realize that they don't make significant sense in functional settings because we can traverse the tree in increase order by *map* function we defined previously.

We'll meet many problems in this series of post that they are only valid in imperative settings, and they are not meaningful problems in functional settings at all. One good example is how to delete an element in red-black-tree[3].

In this section, we'll only present the imperative algorithm for finding the successor and predecessor in a BST.

[3]There is `ref` in ML and OCaml, but we only consider the purely functional settings.

When finding the successor of element x, which is the smallest one y that satisfies $y > x$, there are two cases. If the node with value x has non-NIL right child, the minimum element in right child is the answer; For example, in Figure 1.5, in order to find the successor of 8, we search it's right sub tree for the minimum one, which yields 9 as the result. While if node x don't have right child, we need back-track to find the closest ancestor whose left child is also ancestor of x. In Figure 1.5, since 2 don't have right sub tree, we go back to its parent 1. However, node 1 don't have left child, so we go back again and reach to node 3, the left child of 3, is also ancestor of 2, thus, 3 is the successor of node 2.

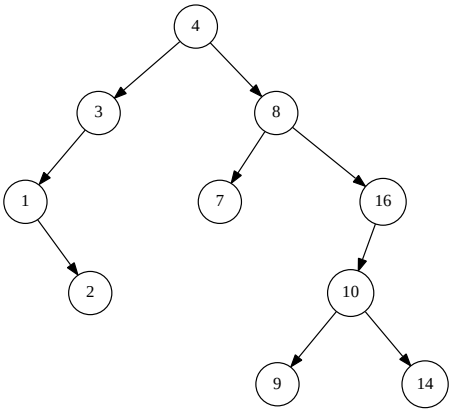


Figure 1.5: The successor of 8, is the minimum one in its right sub tree, 9; In order to find the successor of 2, we go up to its parent 1, but 1 doesn't have left child, we go up again and find 3. Because its left child is also the ancestor of 2, 3 is the result.

Based on this description, the algorithm can be given as the following.

```
1: function SUCC(x)
2:     if RIGHT(x) ≠ NIL then
3:         return MIN(RIGHT(x))
4:     else
5:         p ← PARENT(x)
6:         while p ≠ NIL and x = RIGHT(p) do
7:             x ← p
8:             p ← PARENT(p)
9:         return p
```

If x doesn't has successor, this algorithm returns NIL. The predecessor case is quite similar to the successor algorithm, they are symmetrical to each other.

```
1: function PRED(x)
2:     if LEFT(x) ≠ NIL then
3:         return MAX(LEFT(x))
4:     else
```

www.dbooks.org

```
5:      p ← Parent(x)
6:      while p ≠ NIL and x = Left(p) do
7:          x ← p
8:          p ← Parent(p)
9:      return p
```

Below are the Python programs based on these algorithms. They are changed a bit in while loop conditions.

```
def succ(x):
    if x.right is not None: return tree_min(x.right)
    p = x.parent
    while p is not None and p.left != x:
        x = p
        p = p.parent
    return p

def pred(x):
    if x.left is not None: return tree_max(x.left)
    p = x.parent
    while p is not None and p.right != x:
        x = p
        p = p.parent
    return p
```

Exercise 1.2

- Can you figure out how to iterate a tree as a generic container by using Pred/Succ? What's the performance of such traversing process in terms of big-O?
- A reader discussed about traversing all elements inside a range $[a, b]$. In C++, the algorithm looks like the below code:

  ```
  for_each (m.lower_bound(12), m.upper_bound(26), f);
  ```

  Can you provide the purely function solution for this problem?

1.6 Deletion

Deletion is another 'imperative only' topic for binary search tree. This is because deletion mutate the tree, while in purely functional settings, we don't modify the tree after building it in most application.

However, One method of deleting element from binary search tree in purely functional way is shown in this section. It's actually reconstructing the tree but not modifying the tree.

Deletion is the most complex operation for binary search tree. this is because we must keep the BST property, that for any node, all keys in left sub tree are less than the key of this node, and they are all less than any keys in right sub tree. Deleting a node can break this property.

In this post, different with the algorithm described in [2], A simpler one from SGI STL implementation is used.[4]

To delete a node x from a tree.

- If x has no child or only one child, splice x out;
- Otherwise (x has two children), use minimum element of its right sub tree to replace x, and splice the original minimum element out.

The simplicity comes from the truth that, the minimum element is stored in a node in the right sub tree, which can't have two non-NIL children. It ends up in the trivial case, the node can be directly splice out from the tree.

Figure 1.6, 1.7, and 1.8 illustrate these different cases when deleting a node from the tree.

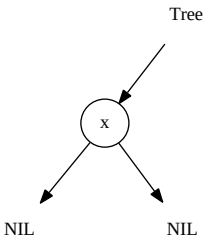
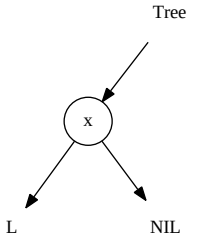
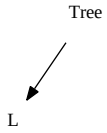


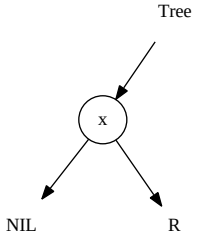
Figure 1.6: x can be spliced out.



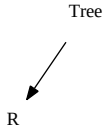
(a) Before delete x.



(b) After delete x, x is spliced out, and replaced by its left child.



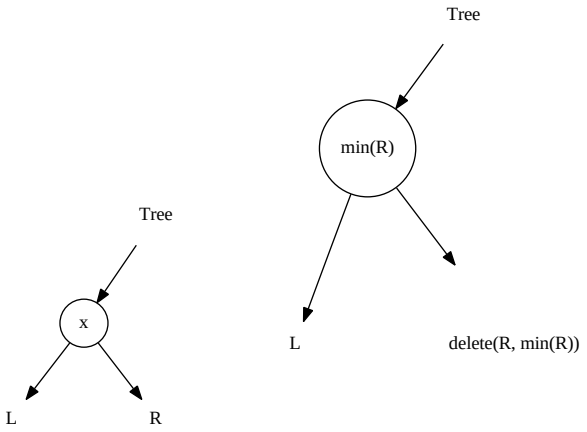
(c) Before delete x.



(d) After delete x, x is spliced out, and replaced by its right child.

Figure 1.7: Delete a node which has only one non-NIL child.

www.dbooks.org



(a) Before delete x.

(b) After delete x, x is replaced by splicing the minimum element from its right child.

Figure 1.8: Delete a node which has both children.

Based on this idea, the deletion can be defined as the below function.

$$delete(T, x) = \begin{cases} \phi & : & T = \phi \\ node(delete(T_l, x), K, T_r) & : & x < k \\ node(T_l, k, delete(T_r, x)) & : & x > k \\ T_r & : & x = k \wedge T_l = \phi \\ T_l & : & x = k \wedge T_r = \phi \\ node(T_l, y, delete(T_r, y)) & : & otherwise \end{cases} \tag{1.9}$$

Where

$$\begin{aligned} T_l &= left(T) \\ T_r &= right(T) \\ k &= key(T) \\ y &= min(T_r) \end{aligned}$$

Translating the function to Haskell yields the below program.

```
delete Empty _ = Empty
delete (Node l k r) x | x < k = (Node (delete l x) k r)
                      | x > k = (Node l k (delete r x))
                      -- x == k
                      | isEmpty l = r
                      | isEmpty r = l
                      | otherwise = (Node l k' (delete r k'))
                        where k' = min r
```

Function `isEmpty` is to test if a tree is empty (ϕ). Note that the algorithm first performs search to locate the node where the element need be deleted, after that it execute the deletion. This algorithm takes $O(h)$ time where h is the height of the tree.

It's also possible to pass the node but not the element to the algorithm for deletion. Thus the searching is no more needed.

The imperative algorithm is more complex because it need set the parent properly. The function will return the root of the result tree.

```
1:  function DELETE(T, x)
2:      r ← T
3:      x' ← x                                              ▷ save x
4:      p ← PARENT(x)
5:      if LEFT(x) = NIL then
6:          x ← RIGHT(x)
7:      else if RIGHT(x) = NIL then
8:          x ← LEFT(x)
9:      else                                                ▷ both children are non-NIL
10:         y ← MIN(RIGHT(x))
11:         KEY(x) ← KEY(y)
12:         Copy other satellite data from y to x
13:         if PARENT(y) ≠ x then                           ▷ y hasn't left sub tree
14:             LEFT(PARENT(y)) ← RIGHT(y)
15:         else                                            ▷ y is the root of right child of x
16:             RIGHT(x) ← RIGHT(y)
17:         if RIGHT(y) ≠ NIL then
18:             PARENT(RIGHT(y)) ← PARENT(y)
19:         Remove y
20:         return r
21:     if x ≠ NIL then
22:         PARENT(x) ← p
23:     if p = NIL then                                     ▷ We are removing the root of the tree
24:         r ← x
25:     else
26:         if LEFT(p) = x' then
27:             LEFT(p) ← x
28:         else
29:             RIGHT(p) ← x
30:     Remove x'
31:     return r
```

Here we assume the node to be deleted is not empty (otherwise we can simply returns the original tree). In other cases, it will first record the root of the tree, create copy pointers to x, and its parent.

If either of the children is empty, the algorithm just splice x out. If it has two non-NIL children, we first located the minimum of right child, replace the key of x to y's, copy the satellite data as well, then splice y out. Note that there is a special case that y is the root node of x's right sub tree.

Finally we need reset the stored parent if the original x has only one non-NIL child. If the parent pointer we copied before is empty, it means that we are deleting the root node, so we need return the new root. After the parent is set properly, we finally remove the old x from memory.

The relative Python program for deleting algorithm is given as below. Because Python provides GC, we needn't explicitly remove the node from the

www.dbooks.org

memory.

```
def tree_delete(t, x):
    if x is None:
        return t
    [root, old_x, parent] = [t, x, x.parent]
    if x.left is None:
        x = x.right
    elif x.right is None:
        x = x.left
    else:
        y = tree_min(x.right)
        x.key = y.key
        if y.parent != x:
            y.parent.left = y.right
        else:
            x.right = y.right
        if y.right is not None:
            y.right.parent = y.parent
        return root
    if x is not None:
        x.parent = parent
    if parent is None:
        root = x
    else:
        if parent.left == old_x:
            parent.left = x
        else:
            parent.right = x
    return root
```

Because the procedure seeks minimum element, it runs in $O(h)$ time on a tree of height h.

Exercise 1.3

- There is a symmetrical solution for deleting a node which has two non-NIL children, to replace the element by splicing the maximum one out off the left sub-tree. Write a program to implement this solution.

1.7 Randomly build binary search tree

It can be found that all operations given in this post bound to $O(h)$ time for a tree of height h. The height affects the performance a lot. For a very unbalanced tree, h tends to be $O(n)$, which leads to the worst case. While for balanced tree, h close to $O(\lg n)$. We can gain the good performance.

How to make the binary search tree balanced will be discussed in next post. However, there exists a simple way. Binary search tree can be randomly built as described in [2]. Randomly building can help to avoid (decrease the possibility) unbalanced binary trees. The idea is that before building the tree, we can call a random process, to shuffle the elements.

Exercise 1.4

- Write a randomly building process for binary search tree.

www.dbooks.org

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. ISBN:0262032937. The MIT Press. 2001

[2] Jon Bentley. “Programming Pearls(2nd Edition)”. Addison-Wesley Professional; 2 edition (October 7, 1999). ISBN-13: 978-0201657883

[3] Chris Okasaki. “Ten Years of Purely Functional Data Structures”. http://okasaki.blogspot.com/2008/02/ten-years-of-purely-functional-data.html

[4] SGI. “Standard Template Library Programmer’s Guide”. http://www.sgi.com/tech/stl/

[5] http://en.literateprograms.org/Category:Binary_search_tree

[6] http://en.wikipedia.org/wiki/Foldl

[7] http://en.wikipedia.org/wiki/Function_composition

[8] http://en.wikipedia.org/wiki/Partial_application

[9] Miran Lipovaca. “Learn You a Haskell for Great Good! A Beginner’s Guide”. the last chapter. No Starch Press; 1 edition April 2011, 400 pp. ISBN: 978-1-59327-283-8

www.dbooks.org

Chapter 2

The evolution of insertion sort

2.1 Introduction

In previous chapter, we introduced the 'hello world' data structure, binary search tree. In this chapter, we explain insertion sort, which can be think of the 'hello world' sorting algorithm [1]. It's straightforward, but the performance is not as good as some divide and conqueror sorting approaches, such as quick sort and merge sort. Thus insertion sort is seldom used as generic sorting utility in modern software libraries. We'll analyze the problems why it is slow, and trying to improve it bit by bit till we reach the best bound of comparison based sorting algorithms, $O(n \lg n)$, by evolution to tree sort. And we finally show the connection between the 'hello world' data structure and 'hello world' sorting algorithm.

The idea of insertion sort can be vivid illustrated by a real life poker game[2]. Suppose the cards are shuffled, and a player starts taking card one by one.

At any time, all cards in player's hand are well sorted. When the player gets a new card, he insert it in proper position according to the order of points. Figure 2.1 shows this insertion example.

Based on this idea, the algorithm of insertion sort can be directly given as the following.

function SORT(A)
  $X \leftarrow \phi$
  for each $x \in A$ **do**
    INSERT(X, x)
  return X

It's easy to express this process with folding, which we mentioned in the chapter of binary search tree.

$$sort = foldL \quad insert \quad \phi \tag{2.1}$$

[1] Some reader may argue that 'Bubble sort' is the easiest sort algorithm. Bubble sort isn't covered in this book as we don't think it's a valuable algorithm[1]

www.dbooks.org

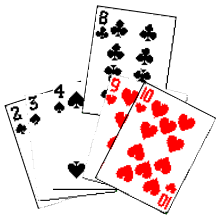


Figure 2.1: Insert card 8 to proper position in a deck.

Note that in the above algorithm, we store the sorted result in X, so this isn't in-place sorting. It's easy to change it to in-place algorithm. Denote the sequence as $A = \{a_1, a_2, ..., a_n\}$.

function SORT(A)
  for $i \leftarrow 2$ to $|A|$ **do**
    insert a_i to sorted sequence $\{a'_1, a'_2, ..., a'_{i-1}\}$

At any time, when we process the i-th element, all elements before i have already been sorted. we continuously insert the current elements until consume all the unsorted data. This idea is illustrated as in figure 8.3.

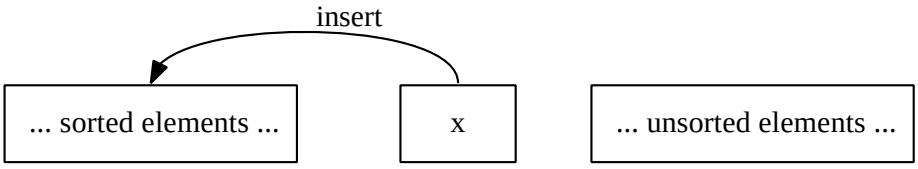


Figure 2.2: The left part is sorted data, continuously insert elements to sorted part.

We can find there is recursive concept in this definition. Thus it can be expressed as the following.

$$sort(A) = \begin{cases} \phi & : \quad A = \phi \\ insert(sort(\{a_2, a_3, ...\}), a_1) & : \quad otherwise \end{cases} \tag{2.2}$$

2.2 Insertion

We haven't answered the question about how to realize insertion however. It's a puzzle how does human locate the proper position so quickly.

For computer, it's an obvious option to perform a scan. We can either scan from left to right or vice versa. However, if the sequence is stored in plain array, it's necessary to scan from right to left.

```
function SORT(A)
    for i ← 2 to |A| do                ▷ Insert A[i] to sorted sequence A[1...i − 1]
        x ← A[i]
        j ← i − 1
        while j > 0 ∧ x < A[j] do
            A[j + 1] ← A[j]
            j ← j − 1
        A[j + 1] ← x
```

One may think scan from left to right is natural. However, it isn't as effect as above algorithm for plain array. The reason is that, it's expensive to insert an element in arbitrary position in an array. As array stores elements continuously, if we want to insert new element x in position i, we must shift all elements after i, including $i+1, i+2, ...$ one position to right. After that the cell at position i is empty, and we can put x in it. This is illustrated in figure 2.3.

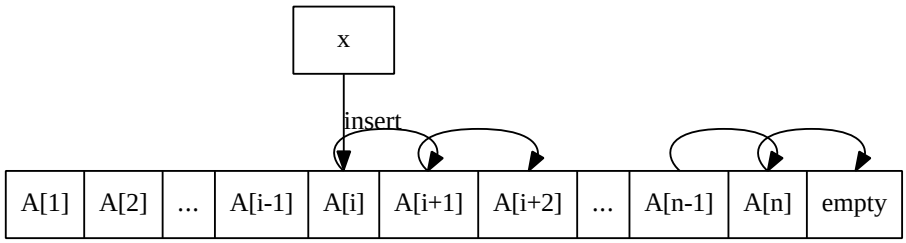


Figure 2.3: Insert x to array A at position i.

If the length of array is n, this indicates we need examine the first i elements, then perform $n - i + 1$ moves, and then insert x to the i-th cell. So insertion from left to right need traverse the whole array anyway. While if we scan from right to left, we examine i elements at most, and perform the same amount of moves.

Translate the above algorithm to Python yields the following code.

```
def isort(xs):
    n = len(xs)
    for i in range(1, n):
        x = xs[i]
        j = i - 1
        while j ≥ 0 and x < xs[j]:
            xs[j+1] = xs[j]
            j = j - 1
        xs[j+1] = x
```

It can be found some other equivalent programs, for instance the following ANSI C program. However this version isn't as effective as the pseudo code.

```
void isort(Key* xs, int n){
  int i, j;
  for(i=1; i<n; ++i)
    for(j=i-1; j≥0 && xs[j+1] < xs[j]; --j)
      swap(xs, j, j+1);
}
```

www.dbooks.org

This is because the swapping function, which can exchange two elements typically uses a temporary variable like the following:

```
void swap(Key* xs, int i, int j){
  Key temp = xs[i];
  xs[i] = xs[j];
  xs[j] = temp;
}
```

So the ANSI C program presented above takes $3m$ times assignment, where m is the number of inner loops. While the pseudo code as well as the Python program use shift operation instead of swapping. There are $m + 2$ times assignment.

We can also provide INSERT() function explicitly, and call it from the general insertion sort algorithm in previous section. We skip the detailed realization here and left it as an exercise.

All the insertion algorithms are bound to $O(n)$, where n is the length of the sequence. No matter what difference among them, such as scan from left or from right. Thus the over all performance for insertion sort is quadratic as $O(n^2)$.

Exercise 2.1

- Provide explicit insertion function, and call it with general insertion sort algorithm. Please realize it in both procedural way and functional way.

2.3 Improvement 1

Let's go back to the question, that why human being can find the proper position for insertion so quickly. We have shown a solution based on scan. Note the fact that at any time, all cards at hands have been well sorted, another possible solution is to use binary search to find that location.

We'll explain the search algorithms in other dedicated chapter. Binary search is just briefly introduced for illustration purpose here.

The algorithm will be changed to call a binary search procedure.

function SORT(A)
  for $i \leftarrow 2$ to $|A|$ **do**
    $x \leftarrow A[i]$
    $p \leftarrow$ BINARY-SEARCH($A[1...i-1], x$)
    for $j \leftarrow i$ down to p **do**
      $A[j] \leftarrow A[j-1]$
    $A[p] \leftarrow x$

Instead of scan elements one by one, binary search utilize the information that all elements in slice of array $\{A_1, ..., A_{i-1}\}$ are sorted. Let's assume the order is monotonic increase order. To find a position j that satisfies $A_{j-1} \leq x \leq A_j$. We can first examine the middle element, for example, $A_{\lfloor i/2 \rfloor}$. If x is less than it, we need next recursively perform binary search in the first half of the sequence; otherwise, we only need search in last half.

Every time, we halve the elements to be examined, this search process runs $O(\lg n)$ time to locate the insertion position.

```
function BINARY-SEARCH(A, x)
    l ← 1
    u ← 1 + |A|
    while l < u do
        m ← ⌊(l+u)/2⌋
        if A[m] = x then
            return m                          ▷ Find a duplicated element
        else if A[m] < x then
            l ← m + 1
        else
            u ← m
    return l
```

The improved insertion sort algorithm is still bound to $O(n^2)$, compare to previous section, which we use $O(n^2)$ times comparison and $O(n^2)$ moves, with binary search, we just use $O(n \lg n)$ times comparison and $O(n^2)$ moves.

The Python program regarding to this algorithm is given below.

```
def isort(xs):
    n = len(xs)
    for i in range(1, n):
        x = xs[i]
        p = binary_search(xs[:i], x)
        for j in range(i, p, -1):
            xs[j] = xs[j-1]
        xs[p] = x

def binary_search(xs, x):
    l = 0
    u = len(xs)
    while l < u:
        m = (l+u)/2
        if xs[m] == x:
            return m
        elif xs[m] < x:
            l = m + 1
        else:
            u = m
    return l
```

Exercise 2.2

Write the binary search in recursive manner. You needn't use purely functional programming language.

2.4 Improvement 2

Although we improve the search time to $O(n \lg n)$ in previous section, the number of moves is still $O(n^2)$. The reason of why movement takes so long time, is because the sequence is stored in plain array. The nature of array is continuously layout data structure, so the insertion operation is expensive. This hints

www.dbooks.org

us that we can use linked-list setting to represent the sequence. It can improve the insertion operation from $O(n)$ to constant time $O(1)$.

$$insert(A, x) = \begin{cases} \{x\} & : & A = \phi \\ \{x\} \cup A & : & x < a_1 \\ \{a_1\} \cup insert(\{a_2, a_3, ...a_n\}, x) & : & otherwise \end{cases} \tag{2.3}$$

Translating the algorithm to Haskell yields the below program.

```
insert [] x = [x]
insert (y:ys) x = if x < y then x:y:ys else y:insert ys x
```

And we can complete the two versions of insertion sort program based on the first two equations in this chapter.

```
isort [] = []
isort (x:xs) = insert (isort xs) x
```

Or we can represent the recursion with folding.

```
isort = foldl insert []
```

Linked-list setting solution can also be described imperatively. Suppose function KEY(x), returns the value of element stored in node x, and NEXT(x) accesses the next node in the linked-list.

function INSERT(L, x)
  $p \leftarrow$ NIL
  $H \leftarrow L$
  while $L \neq$ NIL $\wedge$ KEY(L) $<$ KEY(x) **do**
    $p \leftarrow L$
    $L \leftarrow$ NEXT(L)
  NEXT(x) $\leftarrow L$
  if $p =$ NIL **then**
    $H \leftarrow x$
  else
    NEXT(p) $\leftarrow x$
  return H

For example in ANSI C, the linked-list can be defined as the following.

```
struct node{
  Key key;
  struct node* next;
};
```

Thus the insert function can be given as below.

```
struct node* insert(struct node* lst, struct node* x){
  struct node *p, *head;
  p = NULL;
  for(head = lst; lst && x→key > lst→key; lst = lst→next)
    p = lst;
  x→next = lst;
  if(!p)
    return x;
  p→next = x;
```

```
  return head;
}
```

Instead of using explicit linked-list such as by pointer or reference based structure. Linked-list can also be realized by another index array. For any array element $A[i]$, $Next[i]$ stores the index of next element follows $A[i]$. It means $A[Next[i]]$ is the next element after $A[i]$.

The insertion algorithm based on this solution is given like below.

function INSERT($A, Next, i$)
  $j \leftarrow \perp$
  while $Next[j] \neq$ NIL $\wedge A[Next[j]] < A[i]$ **do**
    $j \leftarrow Next[j]$
  $Next[i] \leftarrow Next[j]$
  $Next[j] \leftarrow i$

Here $\perp$ means the head of the $Next$ table. And the relative Python program for this algorithm is given as the following.

```
def isort(xs):
    n = len(xs)
    next = [-1]*(n+1)
    for i in range(n):
        insert(xs, next, i)
    return next

def insert(xs, next, i):
    j = -1
    while next[j] != -1 and xs[next[j]] < xs[i]:
        j = next[j]
    next[j], next[i] = i, next[j]
```

Although we change the insertion operation to constant time by using linked-list. However, we have to traverse the linked-list to find the position, which results $O(n^2)$ times comparison. This is because linked-list, unlike array, doesn't support random access. It means we can't use binary search with linked-list setting.

Exercise 2.3

- Complete the insertion sort by using linked-list insertion function in your favorate imperative programming language.
- The index based linked-list return the sequence of rearranged index as result. Write a program to re-order the original array of elements from this result.

2.5 Final improvement by binary search tree

It seems that we drive into a corner. We must improve both the comparison and the insertion at the same time, or we will end up with $O(n^2)$ performance.

www.dbooks.org

We must use binary search, this is the only way to improve the comparison time to $O(\lg n)$. On the other hand, we must change the data structure, because we can't achieve constant time insertion at a position with plain array.

This remind us about our 'hello world' data structure, binary search tree. It naturally support binary search from its definition. At the same time, We can insert a new node in binary search tree in $O(1)$ constant time if we already find the location.

So the algorithm changes to this.

function SORT(A)
  $T \leftarrow \phi$
  for each $x \in A$ **do**
    $T \leftarrow$ INSERT-TREE(T, x)
  return TO-LIST(T)

Where INSERT-TREE() and TO-LIST() are described in previous chapter about binary search tree.

As we have analyzed for binary search tree, the performance of tree sort is bound to $O(n \lg n)$, which is the lower limit of comparison based sort[3].

2.6 Short summary

In this chapter, we present the evolution process of insertion sort. Insertion sort is well explained in most textbooks as the first sorting algorithm. It has simple and straightforward idea, but the performance is quadratic. Some textbooks stop here, but we want to show that there exist ways to improve it by different point of view. We first try to save the comparison time by using binary search, and then try to save the insertion operation by changing the data structure to linked-list. Finally, we combine these two ideas and evolute insertion sort to tree sort.

Bibliography

[1] http://en.wikipedia.org/wiki/Bubble_sort

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. "Introduction to Algorithms, Second Edition". ISBN:0262032937. The MIT Press. 2001

[3] Donald E. Knuth. "The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)". Addison-Wesley Professional; 2 edition (May 4, 1998) ISBN-10: 0201896850 ISBN-13: 978-0201896855

www.dbooks.org

Chapter 3

Red-black tree, not so complex as it was thought

3.1 Introduction

3.1.1 Exploit the binary search tree

We showed the power of using binary search tree as a dictionary to count the occurrence of every word in a book in previous chapter.

One may come to the idea to feed a yellow page book [1] to a binary search tree, and use it to look up the phone number for a contact.

By modifying a bit of the program for word occurrence counting yields the following code.

```
int main(int, char** ){
  ifstream f("yp.txt");
  map<string, string> dict;
  string name, phone;
  while(f>>name && f>>phone)
    dict[name]=phone;
  for(;;){
    cout<<"\nname: ";
    cin>>name;
    if(dict.find(name)==dict.end())
      cout<<"not found";
    else
      cout<<"phone: "<<dict[name];
  }
}
```

This program works well. However, if you replace the STL map with the binary search tree introduced in previous chapter, the performance will be bad, especially when you search some names such as Zara, Zed, Zulu.

This is because the content of yellow page is typically listed in lexicographic order. Which means the name list is in increase order. If we try to insert a

[1] A telephone number contact list book

www.dbooks.org

sequence of number 1, 2, 3, ..., n to a binary search tree, we will get a tree like in Figure 3.1.

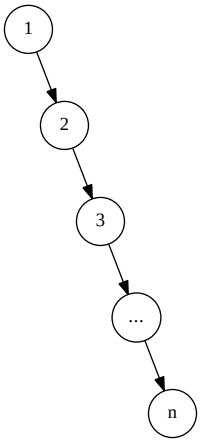


Figure 3.1: unbalanced tree

This is extreme unbalanced binary search tree. The looking up performs $O(h)$ for a tree with height h. In balanced case, we benefit from binary search tree by $O(\lg n)$ search time. But in this extreme case, the search time downgraded to $O(n)$. It's no better than a normal link-list.

Exercise 3.1

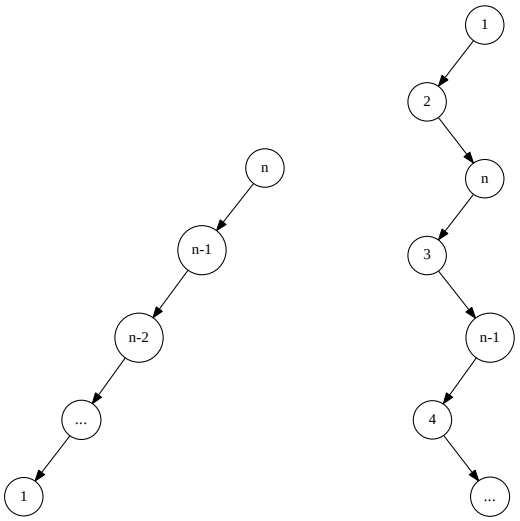
- For a very big yellow page list, one may want to speed up the dictionary building process by two concurrent tasks (threads or processes). One task reads the name-phone pair from the head of the list, while the other one reads from the tail. The building terminates when these two tasks meet at the middle of the list. What will be the binary search tree looks like after building? What if you split the list more than two and use more tasks?
- Can you find any more cases to exploit a binary search tree? Please consider the unbalanced trees shown in figure 3.2.

3.1.2 How to ensure the balance of the tree

In order to avoid such case, we can shuffle the input sequence by randomized algorithm, such as described in Section 12.4 in [2]. However, this method doesn't always work, for example the input is fed from user interactively, and the tree need to be built and updated online.

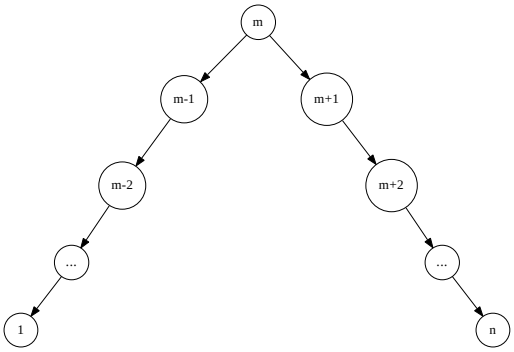
There are many solutions people found to make binary search tree balanced. Many of them rely on the rotation operations to the binary search tree. Rotation operations change the tree structure while maintain the ordering of the elements. Thus it can be used to improve the balance property of the binary search tree.

In this chapter, we'll first introduce the red-black tree. It is one of the most popular and widely used self-adjusting balanced binary search tree. In next chapter, we'll introduce another intuitive solution, the AVL tree. In later



(a)

(b)



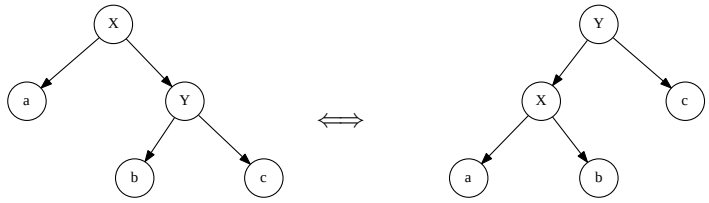
(c)

Figure 3.2: Some unbalanced trees

www.dbooks.org

chapter about binary heaps, we'll show another interesting tree called splay tree, which can gradually adjust the the tree to make it more and more balanced.

3.1.3 Tree rotation



(a) (b)

Figure 3.3: Tree rotation, 'rotate-left' transforms the tree from left side to right side, and 'rotate-right' does the inverse transformation.

Tree rotation is a set of operations that can transform the tree structure without changing the in-order traverse result. It based on the fact that for a specified ordering, there are multiple binary search trees correspond to it. Figure 3.3 shows the tree rotation. For a binary search tree on the left side, left rotate transforms it to the tree on the right, and right rotate does the inverse transformation.

Although tree rotation can be realized in procedural way, there exists simple functional definition by using pattern matching. Denote the non-empty tree as $T = (T_l, k, T_r)$, where k is the key, and T_l, T_r are left and right sub-trees.

$$rotate_l(T) = \begin{cases} ((a, X, b), Y, c) & : \quad T = (a, X, (b, Y, c)) \\ T & : \quad otherwise \end{cases} \tag{3.1}$$

$$rotate_r(T) = \begin{cases} (a, X, (b, Y, c)) & : \quad T = ((a, X, b), Y, c)) \\ T & : \quad otherwise \end{cases} \tag{3.2}$$

To perform tree rotation imperatively, we need set all fields of the node as the following.

```
1:  function LEFT-ROTATE(T, x)
2:      p ← PARENT(x)
3:      y ← RIGHT(x)                                  ▷ Assume y ≠ NIL
4:      a ← LEFT(x)
5:      b ← LEFT(y)
6:      c ← RIGHT(y)
7:      REPLACE(x, y)
8:      SET-CHILDREN(x, a, b)
9:      SET-CHILDREN(y, x, c)
10:     if p = NIL then
11:         T ← y
12:     return T
```

```
13: function RIGHT-ROTATE(T, y)
14:     p ← PARENT(y)
15:     x ← LEFT(y)                                   ▷ Assume x ≠ NIL
16:     a ← LEFT(x)
17:     b ← RIGHT(x)
18:     c ← RIGHT(y)
19:     REPLACE(y, x)
20:     SET-CHILDREN(y, b, c)
21:     SET-CHILDREN(x, a, y)
22:     if p = NIL then
23:         T ← x
24:     return T
```

Where procedure REPLACE(x, y), uses y to replace x.

```
1: function REPLACE(x, y)
2:     if PARENT(x) = NIL then
3:         if y ≠ NIL then PARENT(y) ← NIL
4:     else if LEFT(PARENT(x)) = x then
5:         SET-LEFT(PARENT(x), y)
6:     else
7:         SET-RIGHT(PARENT(x), y)
8:     PARENT(x) ← NIL
```

Procedure SET-CHILDREN assigns a pair of sub-trees as the left and right children of a given node.

```
1: function SET-CHILDREN(x, L, R)
2:     SET-LEFT(x, L)
3:     SET-RIGHT(x, R)

4: function SET-LEFT(x, y)
5:     LEFT(x) ← y
6:     if y ≠ NIL then PARENT(y) ← x

7: function SET-RIGHT(x, y)
8:     RIGHT(x) ← y
9:     if y ≠ NIL then PARENT(y) ← x
```

Compare the imperative operations with the pattern matching functions, we can found the latter focus on the structure change, while the former focus on the rotation process. As the title of this chapter indicated, red-black tree needn't be so complex as it was thought. Many traditional algorithm text books use the classic procedural treatment to the red-black tree. When insert or delete keys, there are multiple cases with a series of node manipulation. On the other hand, in functional settings, the algorithm turns to be intuitive and simple, although there is some performance overhead.

Most of the content in this chapter is based on Chris Okasaki's work in [2].

www.dbooks.org

3.2 Definition of red-black tree

Red-black tree is a type of self-balancing binary search tree[4]. [2] By using color changing and rotation, red-black tree provides a very simple and straightforward way to keep the tree balanced.

For a binary search tree, we can augment the nodes with a color field, a node can be colored either red or black. We call a binary search tree red-black tree if it satisfies the following 5 properties([2] pp273).

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Why this 5 properties can ensure the red-black tree is well balanced? Because they have a key characteristic, the longest path from root to a leaf can't be as 2 times longer than the shortest path.

Consider the 4-th property. It means there can't be two adjacent red nodes. so the shortest path only contains black nodes, any path that is longer than the shortest one has interval red nodes. According to property 5, all paths have the same number of black nodes, it finally ensures there can't be any path that is 2 times longer than others[4]. Figure 3.4 shows a red-black tree example.

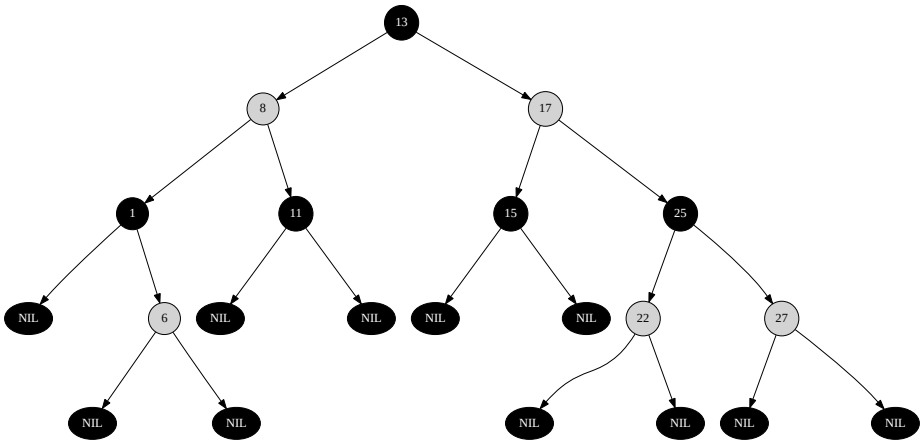


Figure 3.4: A red-black tree

As all NIL nodes are black, people often omit them when draw red-black tree. Figure 3.5 gives the corresponding tree that hides all the NIL nodes.

[2]Red-black tree is one of the equivalent form of 2-3-4 tree (see chapter B-tree about 2-3-4 tree). That is to say, for any 2-3-4 tree, there is at least one red-black tree has the same data order.

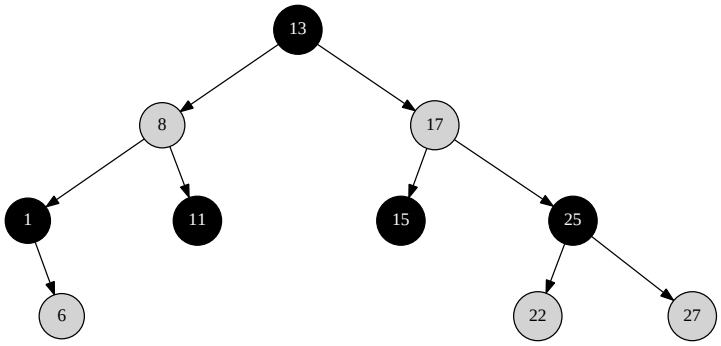


Figure 3.5: The red-black tree with all NIL nodes hidden.

All read operations such as search, find the min/max, are same as the binary search tree. The insertion and deletion are special for the red-black tree.

Many implementation of set or map container are based on red-black tree. One example is the C++ Standard library (STL)[4].

For the data layout, the only change is the color color information need be augmented to binary search tree. This can be represented as a data field. Like the below C++ example.

```
enum Color {Red, Black};

template <class T>
struct node{
  Color color;
  T key;
  node* left;
  node* right;
  node* parent;
};
```

In functional settings, we can add the color information in constructors, below is the Haskell example of red-black tree definition.

```
data Color = R | B
data RBTree a = Empty
              | Node Color (RBTree a) a (RBTree a)
```

Exercise 3.2

- Can you prove that a red-black tree with n nodes has height at most $2\lg(n+1)$?

www.dbooks.org

3.3 Insertion

The tree may be unblanced if a new node is inserted with the method we used for the binary search tree. In order to maintain the red-black properties, we need do some fixing after insertion.

When insert a new key, we can always insert it as a red node. As far as the new inserted node isn't the root of the tree, we can keep all properties except the 4-th one as it may bring two adjacent red nodes.

There are both functional and procedural fixing methods. One is intuitive but has some overhead, the other is a bit complex but has higher performance. In this chapter, we focus on the functional approach to show how easy a red-black tree insertion algorithm can be realized. The traditional procedural method will be given for comparison purpose.

As described by Chris Okasaki, there are total 4 cases which violate property 4. All of them has 2 adjacent red nodes. However, they have a uniformed structure after fixing[2] as shown in figure 3.6.

Note that this transformation will move the redness one level up. During the bottom-up recursive fixing, the last step will make the root node red. According to property 2, root is always black, thus we need finally fix to revert the root color to black.

Observing that the 4 cases and the fixed result have strong patterns, the fixing function can be defined by using the similar method we mentioned in tree rotation. Denote the color of a node as $\mathcal{C}$, it has two values: black $\mathcal{B}$, and red$\mathcal{R}$. A none empty tree can be represented as $T = (\mathcal{C}, T_l, k, T_r)$.

$$balance(T) = \begin{cases} (\mathcal{R}, (\mathcal{B}, A, x, B), y, (\mathcal{B}, C, z, D)) & : & match(T) \\ T & : & otherwise \end{cases} \tag{3.3}$$

where function $match()$ tests if a tree mathes one of the 4 possible patterns as the following.

$$match(T) = \left\{ T = \begin{array}{l} (\mathcal{B}, (\mathcal{R}, (\mathcal{R}, A, x, B), y, C), z, D) \vee \\ (\mathcal{B}, (\mathcal{R}, A, x, (\mathcal{R}, B, y, C), z, D)) \vee \\ (\mathcal{B}, A, x, (\mathcal{R}, B, y, (\mathcal{R}, C, z, D))) \vee \\ (\mathcal{B}, A, x, (\mathcal{R}, (\mathcal{R}, B, y, C), z, D)) \end{array} \right\}$$

With function $balance(T)$ defined, we can modify the binary search tree insertion functions to make it work for red-black tree.

$$insert(T, k) = makeBlack(ins(T, k)) \tag{3.4}$$

where

$$ins(T, k) = \begin{cases} (\mathcal{R}, \phi, k, \phi) & : & T = \phi \\ balance((ins(T_l, k), k', T_r)) & : & k < k' \\ balance((T_l, k', ins(T_r, k))) & : & otherwise \end{cases} \tag{3.5}$$

If the tree is empty, then a new red node with k as the key is created; otherwise, denote the children and the key as T_l, T_r, and k', we compare k and

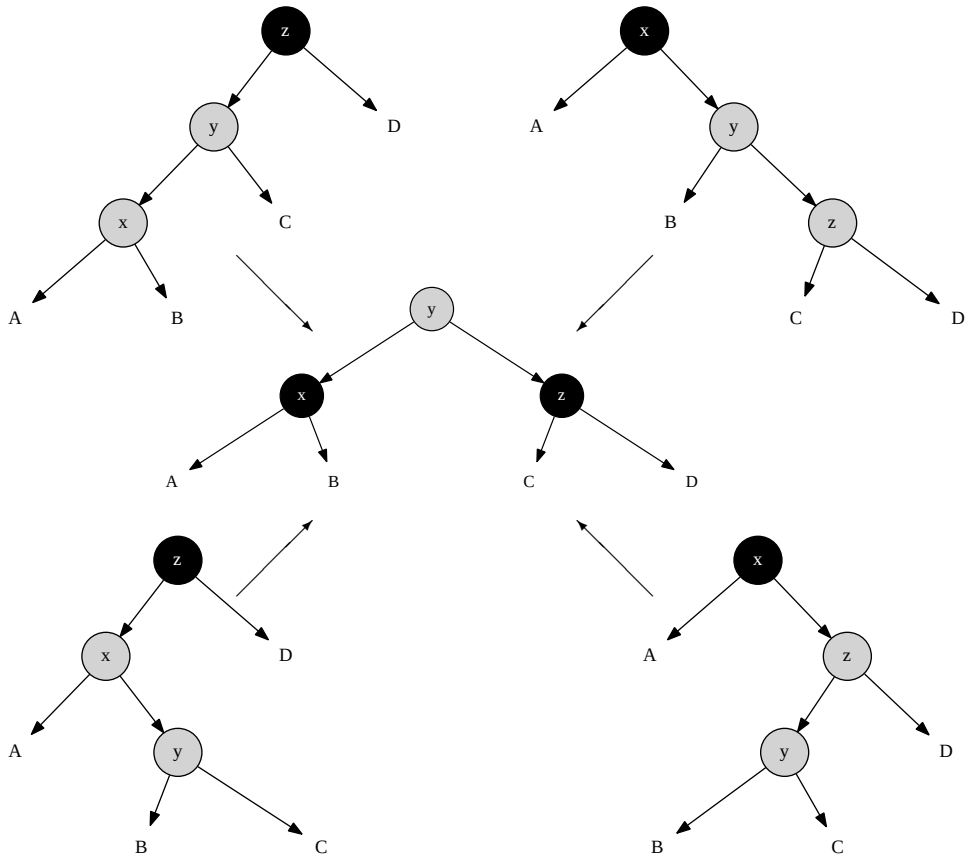


Figure 3.6: 4 cases for balancing a red-black tree after insertion

www.dbooks.org

k' and recursively insert k to a child. Function *balance* is called after that, and the root is re-colored black finally.

$$makeBlack(T) = (\mathcal{B}, T_l, k, T_r) \tag{3.6}$$

Summarize the above functions and use language supported pattern matching features, we can come to the following Haskell program.

```
insert t x = makeBlack $ ins t where
    ins Empty = Node R Empty x Empty
    ins (Node color l k r)
        | x < k     = balance color (ins l) k r
        | otherwise = balance color l k (ins r) --[3]
    makeBlack(Node _ l k r) = Node B l k r

balance B (Node R (Node R a x b) y c) z d =
               Node R (Node B a x b) y (Node B c z d)
balance B (Node R a x (Node R b y c)) z d =
               Node R (Node B a x b) y (Node B c z d)
balance B a x (Node R b y (Node R c z d)) =
               Node R (Node B a x b) y (Node B c z d)
balance B a x (Node R (Node R b y c) z d) =
               Node R (Node B a x b) y (Node B c z d)
balance color l k r = Node color l k r
```

Note that the 'balance' function is changed a bit from the original definition. Instead of passing the tree, we pass the color, the left child, the key and the right child to it. This can save a pair of 'boxing' and 'un-boxing' operations.

This program doesn't handle the case of duplicated keys. we can either overwrite the key or drop the duplicated one. Another option is to augment the data with a linked list([2], pp269).

Figure 3.7 shows two red-black trees built from feeding list 11, 2, 14, 1, 7, 15, 5, 8, 4 and 1, 2, ..., 8. The tree is well balanced even if we input an ordered list.

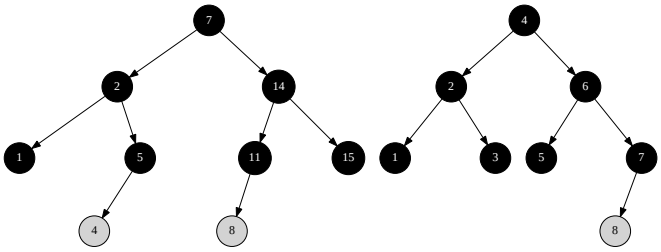


Figure 3.7: insert results generated from two sequences of keys.

This algorithm shows great simplicity by summarizing the uniform pattern from the four different unbalanced cases. It is expressive over the traditional tree rotation approach, that even in programming languages which don't support pattern matching, the algorithm can still be implemented by manually check the pattern. A Scheme/Lisp program is available along with this book can be referenced as an example.

The insertion algorithm takes $O(\lg n)$ time to insert a key to a red-black tree which has n nodes.

Exercise 3.3

- Write a program in an imperative language, such as C, C++ or python to realize the same algorithm in this section. Note that, because there is no language supported pattern matching, you need to test the 4 different cases manually.

3.4 Deletion

Remind the deletion section in binary search tree. Deletion is 'imperative only' for red-black tree as well. In many cases, the tree is often built just one time, and then performs looking up frequently[3].

The purpose of this section is to show that red-black tree deletion is possible in purely functional settings, although it actually rebuilds the tree because trees are read only in terms of purely functional data structure[3]. In real world, it's up to the user (i.e. the programmer) to adopt the proper solution. One option is to mark the node be deleted with a flag, and later rebuild the tree when the number of deleted nodes exceeds 50%.

Deletion is more complex than insertion in both functional and imperative settings, as there are more cases to fix. Deletion may also violate the red black tree properties, so we need fix it after the normal deletion as described in binary search tree.

The problem only happens if you try to delete a black node, because it will violate the last property of red-black tree. The number of black node in the path decreases so not all the paths contain the same number of black node.

When delete a black node, we can resume the last red-black property by introducing a 'doubly-black' concept([2], pp290). It means that the although the node is deleted, the blackness is kept by storing it in the parent node. If the parent node is red, it turns to black, However, if it's already black, it turns to 'doubly-black'.

In order to express the 'doubly-black node', The definition need some modification accordingly.

```
data Color = R | B | BB -- BB: doubly black for deletion
data RBTree a = Empty | BBEmpty -- doubly black empty
              | Node Color (RBTree a) a (RBTree a)
```

When deleting a node, we first perform the same binary search tree deleting algorithm. After that, if the node to be sliced out is black, we need fix the tree to keep the red-black properties. The delete function is defined as the following.

$$delete(T, k) = blackenRoot(del(T, k)) \tag{3.7}$$

[3]Actually, the common part of the tree is reused. Most functional programming environments support this persistent feature.

www.dbooks.org

where

$$del(T,k) = \begin{cases} \phi & : & T = \phi \\ fixBlack^2((\mathcal{C}, del(T_l,k), k', T_r)) & : & k < k' \\ fixBlack^2((\mathcal{C}, T_l, k', del(T_r,k))) & : & k > k' \\ \begin{cases} mkBlk(T_r) & : & \mathcal{C} = \mathcal{B} \\ T_r & : & otherwise \end{cases} & : & T_l = \phi \\ \begin{cases} mkBlk(T_l) & : & \mathcal{C} = \mathcal{B} \\ T_l & : & otherwise \end{cases} & : & T_r = \phi \\ fixBlack^2((\mathcal{C}, T_l, k'', del(T_r,k''))) & : & otherwise \end{cases} \tag{3.8}$$

The real deleting happens inside function *del*. For the trivial case, that the tree is empty, the deletion result is ϕ; If the key to be deleted is less than the key of the current node, we recursively perform deletion on its left sub-tree; if it is bigger than the key of the current node, then we recursively delete the key from the right sub-tree; Because it may bring doubly-blackness, so we need fix it.

If the key to be deleted is equal to the key of the current node, we need splice it out. If one of its children is empty, we just replace the node by the other one and reserve the blackness of this node. otherwise we cut and past the minimum element $k'' = min(T_r)$ from the right sub-tree.

Function *delete* just forces the result tree of *del* to have a black root. This is realized by function *blackenRoot*.

$$blackenRoot(T) = \begin{cases} \phi & : & T = \phi \\ (\mathcal{B}, T_l, k, T_r) & : & otherwise \end{cases} \tag{3.9}$$

The $blackenRoot(T)$ function is almost same as the $makeBlack(T)$ function defined for insertion except for the case of empty tree. This is only valid in deletion, because insertion can't result an empty tree, while deletion may.

Function *mkBlk* is defined to reserved the blackness of a node. If the node to be sliced isn't black, this function won't be applied, otherwise, it turns a red node to black and turns a black node to doubly-black. This function also marks an empty tree ϕ to doubly-black empty Φ.

$$mkBlk(T) = \begin{cases} \Phi & : & T = \phi \\ (\mathcal{B}, T_l, k, T_r) & : & \mathcal{C} = \mathcal{R} \\ (\mathcal{B}^2, T_l, k, T_r) & : & \mathcal{C} = \mathcal{B} \\ T & : & otherwise \end{cases} \tag{3.10}$$

where $\mathcal{B}^2$ denotes the doubly-black color.

Summarizing the above functions yields the following Haskell program.

```
delete t x = blackenRoot(del t x) where
    del Empty _ = Empty
    del (Node color l k r) x
        | x < k = fixDB color (del l x) k r
        | x > k = fixDB color l k (del r x)
        -- x == k, delete this node
        | isEmpty l = if color==B then makeBlack r else r
        | isEmpty r = if color==B then makeBlack l else l
        | otherwise = fixDB color l k' (del r k') where k'= min r
```

```
    blackenRoot (Node _ l k r) = Node B l k r
    blackenRoot _ = Empty

makeBlack (Node B l k r) = Node BB l k r -- doubly black
makeBlack (Node _ l k r) = Node B l k r
makeBlack Empty = BBEmpty
makeBlack t = t
```

The final attack to the red-black tree deletion algorithm is to realize the $fixBlack^2$ function. The purpose of this function is to eliminate the 'doubly-black' colored node by rotation and color changing. There are three cases. In every case, the doubly black node can either be normal node, or doubly black empty node Φ. Let's examine these three cases one by one.

3.4.1 The sibling of the doubly black node is black, and it has one red child

In this situation, we can fix the doubly-blackness with one rotation. Actually there are 4 different sub-cases, all of them can be transformed to one uniformed pattern. They are shown in the figure B.1.

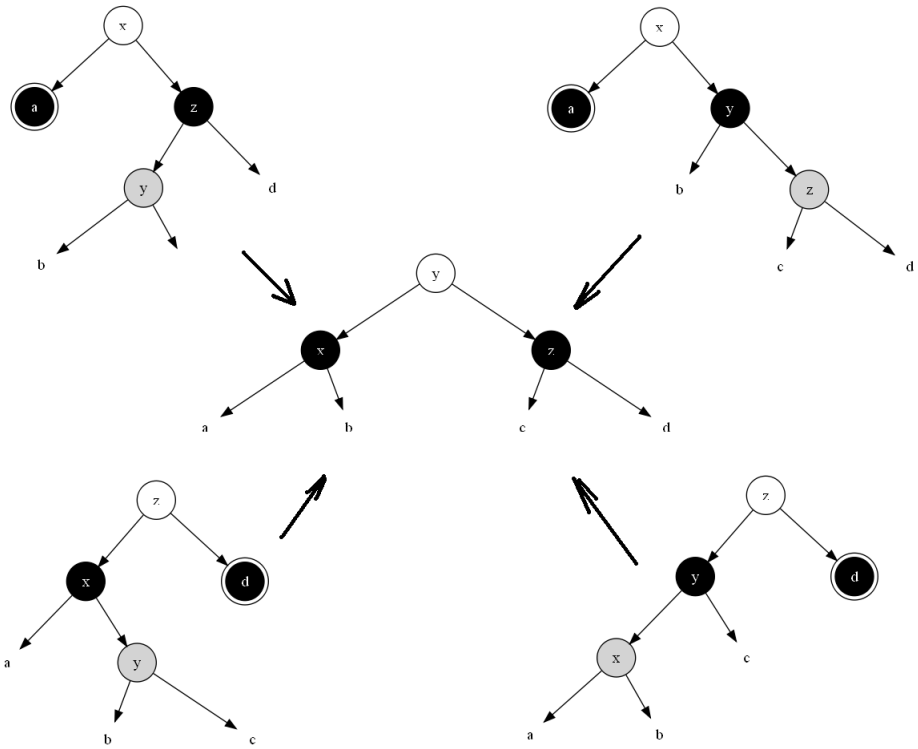


Figure 3.8: Fix the doubly black by rotation, the sibling of the doubly-black node is black, and it has one red child.

www.dbooks.org

The handling of these 4 sub-cases can be realized with pattern matching.

$$fixBlack^2(T) = \begin{cases} (\mathcal{C}, (\mathcal{B}, mkBlk(A), x, B), y, (\mathcal{B}, C, z, D)) & : & p1.1 \\ (\mathcal{C}, (\mathcal{B}, A, x, B), y, (\mathcal{B}, C, z, mkBlk(D))) & : & p1.2 \end{cases} \tag{3.11}$$

where $p1.1$ and $p1.2$ each represent 2 patterns as the following.

$$p1.1 : \left\{ \begin{array}{l} T = (\mathcal{C}, A, x, (\mathcal{B}, (\mathcal{R}, B, y, C), z, D)) \wedge color(A) = \mathcal{B}^2 \\ \vee \\ T = (\mathcal{C}, A, x, (\mathcal{B}, B, y, (\mathcal{R}, C, z, D))) \wedge color(A) = \mathcal{B}^2 \end{array} \right\}$$

$$p1.2 : \left\{ \begin{array}{l} T = (\mathcal{C}, (\mathcal{B}, A, x, (\mathcal{R}, B, y, C)), z, D) \wedge color(D) = \mathcal{B}^2 \\ \vee \\ T = (\mathcal{C}, (\mathcal{B}, (\mathcal{R}, A, x, B), y, C), z, D) \wedge color(D) = \mathcal{B}^2 \end{array} \right\}$$

If the doubly black node is a doubly black empty node Φ, it can be changed back to normal empty node after the above operation. We can add the doubly black empty node handling on top of the (3.11).

$$fixBlack^2(T) = \begin{cases} (\mathcal{C}, (\mathcal{B}, mkBlk(A), x, B), y, (\mathcal{B}, C, z, D)) & : & p1.1 \\ (\mathcal{C}, (\mathcal{B}, \phi, x, B), y, (\mathcal{B}, C, z, D)) & : & p1.1' \\ (\mathcal{C}, (\mathcal{B}, A, x, B), y, (\mathcal{B}, C, z, mkBlk(D))) & : & p1.2 \\ (\mathcal{C}, (\mathcal{B}, A, x, B), y, (\mathcal{B}, C, z, \phi)) & : & p1.2' \end{cases} \tag{3.12}$$

Where patter $p1.1'$ and $p1.2'$ are defined as below:

$$p1.1' : \left\{ \begin{array}{l} T = (\mathcal{C}, \Phi, x, (\mathcal{B}, (\mathcal{R}, B, y, C), z, D)) \\ \vee \\ T = (\mathcal{C}, \Phi, x, (\mathcal{B}, B, y, (\mathcal{R}, C, z, D))) \end{array} \right\}$$

$$p1.2' : \left\{ \begin{array}{l} T = (\mathcal{C}, (\mathcal{B}, A, x, (\mathcal{R}, B, y, C)), z, \Phi) \\ \vee \\ T = (\mathcal{C}, (\mathcal{B}, (\mathcal{R}, A, x, B), y, C), z, \Phi) \end{array} \right\}$$

3.4.2 The sibling of the doubly-black node is red

In this case, we can rotate the tree to it to pattern $p1.1$ or $p1.2$. Figure B.2 shows about it.

We can add this case on top of (3.12) to gain (3.14).

$$fixBlack^2(T) = \begin{cases} ... & : & ... \\ fixBlack^2(\mathcal{B}, fixBlack^2((\mathcal{R}, A, x, B), y, C) & : & p2.1 \\ fixBlack^2(\mathcal{B}, A, x, fixBlack^2((\mathcal{R}, B, y, C)) & : & p2.2 \\ T & : & otherwise \end{cases} \tag{3.13}$$

where $p2.1$ and $p2.2$ are two patterns as the following.

$$p2.1 : \{color(T) = \mathcal{B} \wedge color(T_l) = \mathcal{B}^2 \wedge color(T_r) = \mathcal{R}\}$$

$$p2.2 : \{color(T) = \mathcal{B} \wedge color(T_l) = \mathcal{R} \wedge color(T_r) = \mathcal{B}^2\}$$

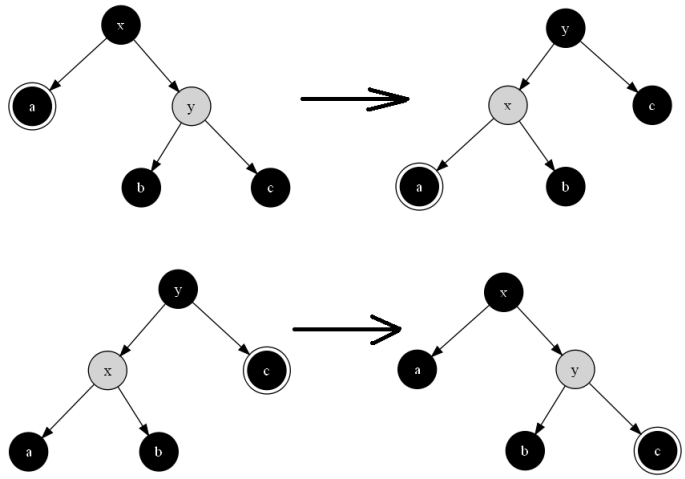


Figure 3.9: The sibling of the doubly-black node is red.

3.4.3 The sibling of the doubly-black node, and its two children are all black

In this case, we can change the color of the sibling node to red; turn the doubly-black node to black and propagate the doubly-blackness one level up to the parent node as shown in figure B.3. There are two symmetric sub-cases.

We go on adding this fixing after formula (3.14).

$$fixBlack^2(T) = \begin{cases} & \ldots & : & \ldots \\ mkBlk((\mathcal{C}, mkBlk(A), x, (\mathcal{R}, B, y, C))) & : & p3.1 \\ mkBlk((\mathcal{C}, (\mathcal{R}, A, x, B), y, mkBlk(C))) & : & p3.2 \\ & \ldots & : & \ldots \end{cases} \tag{3.14}$$

where $p3.1$ and $p3.2$ are two patterns as below.

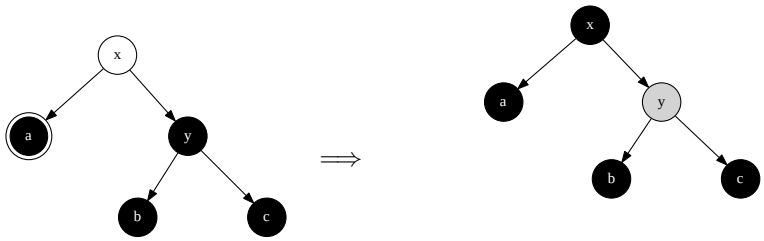
$$p3.1 : \left\{ \begin{array}{l} T = (\mathcal{C}, A, x, (\mathcal{B}, B, y, C)) \wedge \\ color(A) = \mathcal{B}^2 \wedge color(B) = color(C) = \mathcal{B} \end{array} \right\}$$

$$p3.2 : \left\{ \begin{array}{l} T = (\mathcal{C}, (\mathcal{B}, A, x, B), y, C) \wedge \\ color(C) = \mathcal{B}^2 \wedge color(A) = color(B) = \mathcal{B} \end{array} \right\}$$

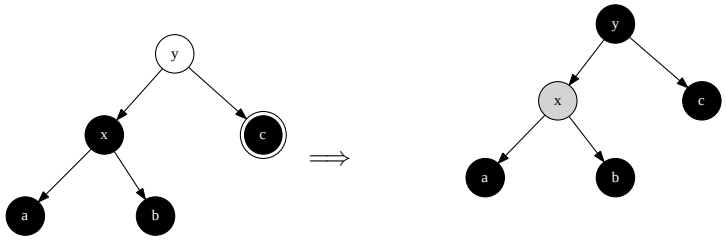
If the doubly black node is doubly black empty node Φ, it can be changed back to normal empty node after re-coloring. We add the doubly black empty node handling to (**??**) as below.

$$fixBlack^2(T) = \begin{cases} & \ldots & : & \ldots \\ mkBlk((\mathcal{C}, mkBlk(A), x, (\mathcal{R}, B, y, C))) & : & p2.1 \\ mkBlk((\mathcal{C}, \phi, x, (\mathcal{R}, B, y, C))) & : & p2.1' \\ mkBlk((\mathcal{C}, (\mathcal{R}, A, x, B), y, mkBlk(C))) & : & p2.2 \\ mkBlk((\mathcal{C}, (\mathcal{R}, A, x, B), y, \phi)) & : & p2.2' \\ & \ldots & : & \ldots \end{cases} \tag{3.15}$$

www.dbooks.org



(a) Color of x can be either black or red. (b) If x was red, then it becomes black, otherwise, it becomes doubly-black.



(c) Color of y can be either black or red. (d) If y was red, then it becomes black, otherwise, it becomes doubly-black.

Figure 3.10: propagate the blackness up.

Where pattern $p3.1'$ and $p3.2'$ are defined as the following.

$$p3.1' : \left\{ \begin{array}{l} T = (\mathcal{C}, \Phi, x, (\mathcal{B}, B, y, C)) \wedge \\ color(B) = color(C) = \mathcal{B} \end{array} \right\}$$

$$p3.2' : \left\{ \begin{array}{l} T = (\mathcal{C}, (\mathcal{B}, A, x, B), y, \Phi) \wedge \\ color(A) = color(B) = \mathcal{B} \end{array} \right\}$$

Fixing the doubly-black node with all above different cases is a recursive function. There are two termination conditions. One contains pattern $p1.1$ and $p1.2$, The doubly-black node was eliminated. The other cases may continuously propagate the doubly-blackness from bottom to top till the root. Finally the algorithm marks the root node as black anyway. The doubly-blackness will be removed.

Put formula (3.12), (3.14), and (3.15) together, we can write the final Haskell program.

```
-- the sibling is black, and it has one red child
fixDB color a@(Node BB _ _ _) x (Node B (Node R b y c) z d)
      = Node color (Node B (makeBlack a) x b) y (Node B c z d)
fixDB color BBEmpty x (Node B (Node R b y c) z d)
      = Node color (Node B Empty x b) y (Node B c z d)
fixDB color a@(Node BB _ _ _) x (Node B b y (Node R c z d))
      = Node color (Node B (makeBlack a) x b) y (Node B c z d)
fixDB color BBEmpty x (Node B b y (Node R c z d))
      = Node color (Node B Empty x b) y (Node B c z d)
```

```
fixDB color (Node B a x (Node R b y c)) z d@(Node BB _ _ _)
      = Node color (Node B a x b) y (Node B c z (makeBlack d))
fixDB color (Node B a x (Node R b y c)) z BBEmpty
      = Node color (Node B a x b) y (Node B c z Empty)
fixDB color (Node B (Node R a x b) y c) z d@(Node BB _ _ _)
      = Node color (Node B a x b) y (Node B c z (makeBlack d))
fixDB color (Node B (Node R a x b) y c) z BBEmpty
      = Node color (Node B a x b) y (Node B c z Empty)
-- the sibling is red
fixDB B a@(Node BB _ _ _) x (Node R b y c) = fixDB B (fixDB R a x b) y c
fixDB B a@BBEmpty x (Node R b y c) = fixDB B (fixDB R a x b) y c
fixDB B (Node R a x b) y c@(Node BB _ _ _) = fixDB B a x (fixDB R b y c)
fixDB B (Node R a x b) y c@BBEmpty = fixDB B a x (fixDB R b y c)
-- the sibling and its 2 children are all black, propagate the blackness up
fixDB color a@(Node BB _ _ _) x (Node B b y c) = makeBlack (Node color (makeBlack a) x (Node R b y c))
fixDB color BBEmpty x (Node B b y c) = makeBlack (Node color Empty x (Node R b y c))
fixDB color (Node B a x b) y c@(Node BB _ _ _) = makeBlack (Node color (Node R a x b) y (makeBlack c))
fixDB color (Node B a x b) y BBEmpty = makeBlack (Node color (Node R a x b) y Empty)
-- otherwise
fixDB color l k r = Node color l k r
```

The deletion algorithm takes $O(\lg n)$ time to delete a key from a red-black tree with n nodes.

Exercise 3.4

- As we mentioned in this section, deletion can be implemented by just marking the node as deleted without actually removing it. Once the number of marked nodes exceeds 50%, a tree re-build is performed. Try to implement this method in your favorite programming language.
- Why needn't enclose $mkBlk$ with a call to $fixBlack^2$ explicitly in the definition of $del(T, k)$?

3.5 Imperative red-black tree algorithm ⋆

We almost finished all the content in this chapter. By induction the patterns, we can implement the red-black tree in a simple way compare to the imperative tree rotation solution. However, we should show the comparator for completeness.

For insertion, the basic idea is to use the similar algorithm as described in binary search tree. And then fix the balance problem by rotation and return the final result.

```
1: function INSERT(T, k)
2:     root ← T
3:     x ← CREATE-LEAF(k)
4:     COLOR(x) ← RED
5:     p ← NIL
6:     while T ≠ NIL do
7:         p ← T
8:         if k < KEY(T) then
```

www.dbooks.org

```
            T ← LEFT(T)
        else
            T ← RIGHT(T)
    PARENT(x) ← p
    if p = NIL then                                  ▷ tree T is empty
        return x
    else if k < KEY(p) then
        LEFT(p) ← x
    else
        RIGHT(p) ← x
    return INSERT-FIX(root, x)
```

The only difference from the binary search tree insertion algorithm is that we set the color of the new node as red, and perform fixing before return. Below is the example Python program.

```
def rb_insert(t, key):
    root = t
    x = Node(key)
    parent = None
    while(t):
        parent = t
        if(key < t.key):
            t = t.left
        else:
            t = t.right
    if parent is None: #tree is empty
        root = x
    elif key < parent.key:
        parent.set_left(x)
    else:
        parent.set_right(x)
    return rb_insert_fix(root, x)
```

There are 3 base cases for fixing, and if we take the left-right symmetric into consideration. there are total 6 cases. Among them two cases can be merged together, because they all have uncle node in red color, we can toggle the parent color and uncle color to black and set grand parent color to red. With this merging, the fixing algorithm can be realized as the following.

```
function INSERT-FIX(T, x)
    while PARENT(x) ≠ NIL ∧ COLOR(PARENT(x)) = RED do
        if COLOR(UNCLE(x)) = RED then          ▷ Case 1, x's uncle is red
            COLOR(PARENT(x)) ← BLACK
            COLOR(GRAND-PARENT(x)) ← RED
            COLOR(UNCLE(x)) ← BLACK
            x ← GRAND-PARENT(x)
        else                                              ▷ x's uncle is black
            if PARENT(x) = LEFT(GRAND-PARENT(x)) then
                if x = RIGHT(PARENT(x)) then  ▷ Case 2, x is a right child
                    x ← PARENT(x)
                    T ← LEFT-ROTATE(T, x)
                                                  ▷ Case 3, x is a left child
```

```
13:                COLOR(PARENT(x)) ← BLACK
14:                COLOR(GRAND-PARENT(x)) ← RED
15:                T ← RIGHT-ROTATE(T, GRAND-PARENT(x))
16:            else
17:                if x = LEFT(PARENT(x)) then          ▷ Case 2, Symmetric
18:                    x ← PARENT(x)
19:                    T ← RIGHT-ROTATE(T, x)
                                                        ▷ Case 3, Symmetric
20:                COLOR(PARENT(x)) ← BLACK
21:                COLOR(GRAND-PARENT(x)) ← RED
22:                T ← LEFT-ROTATE(T, GRAND-PARENT(x))
23:    COLOR(T) ← BLACK
24:    return T
```

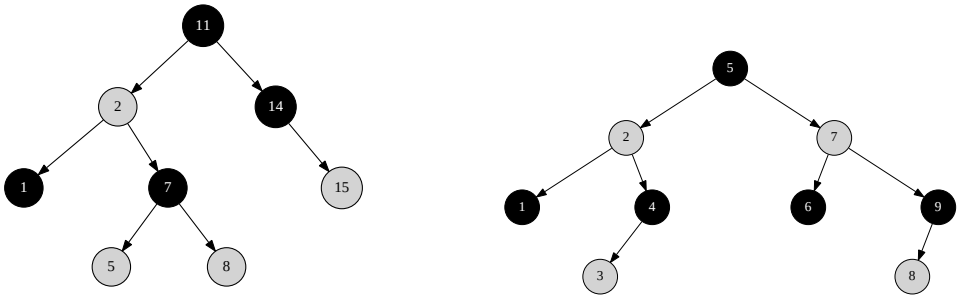
This program takes $O(\lg n)$ time to insert a new key to the red-black tree. Compare this pseudo code and the *balance* function we defined in previous section, we can see the difference. They differ not only in terms of simplicity, but also in logic. Even if we feed the same series of keys to the two algorithms, they may build different red-black trees. There is a bit performance overhead in the pattern matching algorithm. Okasaki discussed about the difference in detail in his paper[2].

Translate the above algorithm to Python yields the below program.

```
# Fix the red→red violation
def rb_insert_fix(t, x):
    while(x.parent and x.parent.color==RED):
        if x.uncle().color == RED:
            #case 1: ((a:R x:R b) y:B c:R) ⟹ ((a:R x:B b) y:R c:B)
            set_color([x.parent, x.grandparent(), x.uncle()],
                      [BLACK, RED, BLACK])
            x = x.grandparent()
        else:
            if x.parent == x.grandparent().left:
                if x == x.parent.right:
                    #case 2: ((a x:R b:R) y:B c) ⟹ case 3
                    x = x.parent
                    t=left_rotate(t, x)
                # case 3: ((a:R x:R b) y:B c) ⟹ (a:R x:B (b y:R c))
                set_color([x.parent, x.grandparent()], [BLACK, RED])
                t=right_rotate(t, x.grandparent())
            else:
                if x == x.parent.left:
                    #case 2': (a x:B (b:R y:R c)) ⟹ case 3'
                    x = x.parent
                    t = right_rotate(t, x)
                # case 3': (a x:B (b y:R c:R)) ⟹ ((a x:R b) y:B c:R)
                set_color([x.parent, x.grandparent()], [BLACK, RED])
                t=left_rotate(t, x.grandparent())
    t.color = BLACK
    return t
```

Figure 3.11 shows the results of feeding same series of keys to the above python insertion program. Compare them with figure 3.7, one can tell the difference clearly.

www.dbooks.org



(a) (b)

Figure 3.11: Red-black trees created by imperative algorithm.

We put the red-black tree delete algorithm in imperative settings in Appendix B, because it is more complex than the insertion.

3.6 More words

Red-black tree is the most popular implementation of balanced binary search tree. Another one is the AVL tree, which we'll introduce in next chapter. Red-black tree can be a good start point for more data structures. If we extend the number of children from 2 to k, and keep the balance as well, it leads to B-tree, If we store the data along with edge but not inside node, it leads to Tries. However, the multiple cases handling and the long program tends to make new comers think red-black tree is complex.

Okasaki's work helps making the red-black tree much easily understand. There are many implementation in other programming languages in that manner [5]. It's also inspired me to find the pattern matching solution for Splay tree and AVL tree etc.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. ISBN:0262032937. The MIT Press. 2001

[2] Chris Okasaki. “FUNCTIONAL PEARLS Red-Black Trees in a Functional Setting”. J. Functional Programming. 1998

[3] Chris Okasaki. “Ten Years of Purely Functional Data Structures”. http://okasaki.blogspot.com/2008/02/ten-years-of-purely-functional-data.html

[4] Wikipedia. “Red-black tree”. http://en.wikipedia.org/wiki/Red-black_tree

[5] Pattern matching. http://rosettacode.org/wiki/Pattern_matching

www.dbooks.org

Chapter 4

AVL tree

4.1 Introduction

4.1.1 How to measure the balance of a tree?

Besides red-black tree, are there any other intuitive self-balancing binary search trees? In order to measure how balancing a binary search tree is, one idea is to compare the height of the right sub-tree and left sub-tree. If they differs a lot, the tree isn't well balanced. Let's denote the difference height between two children as below

$$\delta(T) = |T_r| - |T_l| \tag{4.1}$$

Where $|T|$ means the height of tree T, and T_l, T_r are the left and right sub-trees.

If $\delta(T) = 0$ for every node, The tree is definitely balanced. For example, a complete binary tree has $n = 2^h - 1$ nodes for height h. There is no empty branches unless the leafs. Another trivial case is empty tree. $\delta(\phi) = 0$. The less absolute value of $\delta(T)$ the more balanced the tree is.

We define $\delta(T)$ as the *balance factor* of a binary search tree.

4.2 Definition of AVL tree

The AVL tree is a special binary search tree, that all sub-trees satisfying the following criteria.

$$|\delta(T)| \leq 1 \tag{4.2}$$

The absolute value of balance factor is less than or equal to 1, which means there are only three valid values, -1, 0 and 1. Figure 4.1 shows an example AVL tree.

Why AVL tree can keep the tree balanced? In other words, Can this definition ensure the height of the tree as $O(\lg n)$ where n is the number of the nodes in the tree? Let's prove this fact.

For an AVL tree of height h, The number of nodes varies. It can have at most $2^h - 1$ nodes for a complete binary tree. We are interesting about how

www.dbooks.org

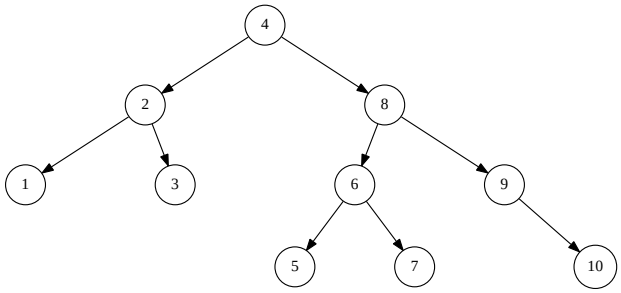


Figure 4.1: AVL tree example

many nodes there are at least. Let's denote the minimum number of nodes for the AVL tree of height h as $N(h)$. It's obvious we have the below result for the trivial cases.

- For empty tree, $h = 0$, $N(0) = 0$;
- For a singleton leaf tree, $h = 1$, $N(1) = 1$;

What's the situation for the common case $N(h)$? Figure 4.2 shows an AVL tree T of height h. It contains three parts, the root node, and two sub trees T_l, T_r. We have the following fact.

$$h = max(|T_l|, |T_r|) + 1 \tag{4.3}$$

We immediately know that, there must be one child has height $h - 1$. According to the definition of AVL tree, we have $||T_l| - |T_r|| \leq 1$. This leads to the fact that the height of other tree can't be lower than $h - 2$, So the total number of the nodes of T is the number of nodes in both children plus 1 (for the root node). We exclaim that.

$$N(h) = N(h-1) + N(h-2) + 1 \tag{4.4}$$

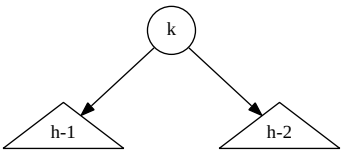


Figure 4.2: An AVL tree of height h. The height of one sub-tree is $h - 1$, the other is no less than $h - 2$

This recursion reminds us the famous Fibonacci series. Actually we can transform it to Fibonacci series by defining $N'(h) = N(h) + 1$. So equation (4.4) changes to.

$$N'(h) = N'(h-1) + N'(h-2) \tag{4.5}$$

Lemma 4.2.1. *Let $N(h)$ be the minimum number of nodes for an AVL tree of height h. and $N'(h) = N(h) + 1$, then*

$$N'(h) \geq \phi^h \tag{4.6}$$

Where $\phi = \frac{\sqrt{5}+1}{2}$ is the golden ratio.

Proof. For the trivial case, we have

- $h = 0$, $N'(0) = 1 \geq \phi^0 = 1$
- $h = 1$, $N'(1) = 2 \geq \phi^1 = 1.618...$

For the induction case, suppose $N'(h) \geq \phi^h$.

$$\begin{aligned} N'(h+1) &= N'(h) + N'(h-1) && \{Fibonacci\} \\ &\geq \phi^h + \phi^{h-1} \\ &= \phi^{h-1}(\phi + 1) && \{\phi + 1 = \phi^2 = \tfrac{\sqrt{5}+3}{2}\} \\ &= \phi^{h+1} \end{aligned}$$

□

From Lemma 4.2.1, we immediately get

$$h \leq log_\phi(n+1) = log_\phi 2 \cdot \lg(n+1) \approx 1.44 \lg(n+1) \tag{4.7}$$

It tells that the height of AVL tree is proportion to $O(\lg n)$, which means that AVL tree is balanced.

For the mutate operations such as tree insertion and deletion, if the balance factor changes to any invalid values, some fixing has to be performed to resume $|\delta|$ within 1. Most implementations utilize tree rotations. In this chapter, we'll show the pattern matching solution which is inspired by Okasaki's red-black tree solution[2]. Because of this 'modify-fix' approach, AVL tree is also a kind of self-balancing binary search tree. For comparison purpose, we'll also show the procedural algorithms.

Of course we can compute the δ value recursively, another option is to store the balance factor inside each nodes, and update them when we modify the tree. The latter one avoid computing the same value every time.

Based on this idea, we can add one extra data field δ to the binary search tree definition. The following C++ example code reflects this change [1].

```
template <class T>
struct node {
    int delta;
    T key;
    node* left;
    node* right;
    node* parent;
};
```

[1] Some implementations store the height of a tree instead of δ as in [5]

www.dbooks.org

In purely functional setting, some implementation use different constructors to store the δ information. for example in [1], there are 4 constructors, `E`, `N`, `P`, `Z` defined. `E` for empty tree, `N` for tree with negative 1 balance factor, `P` for tree with positive 1 balance factor and `Z` for zero case.

In this chapter, we'll explicitly store the balance factor inside the node.

```
data AVLTree a = Empty
               | Br (AVLTree a) a (AVLTree a) Int
```

The immutable operations, including looking up, finding the maximum and minimum elements are all same as the binary search tree. We'll skip them and focus on the mutable operations.

4.3 Insertion

Insert a new element to the tree may violate the AVL tree property that the absolute value of δ exceeds 1. To resume it, one option is to do the tree rotation according to the different insertion cases. Most implementation is based on this approach

Another way is to use the similar pattern matching method mentioned by Okasaki in his red-black tree implementation [2]. Inspired by this idea, it is possible to provide a simple and intuitive solution.

When insert a new key to the AVL tree, the balance factor of the root may *changes* in range $[-1, 1]$[2], and the height may increase at most by one, which we need recursively use this information to update the δ value in further level nodes. We can define the result of the insertion algorithm as a pair of data $(T', \Delta H)$. Where T' is the new tree and ΔH is the increment of height. Let's denote function $first(pair)$ can return the first element in a pair. We can modify the binary search tree insertion algorithm as the following to handle AVL tree.

$$insert(T, k) = first(ins(T, k)) \tag{4.8}$$

where

$$ins(T, k) = \begin{cases} ((\phi, k, \phi, 0), 1) & : & T = \phi \\ tree(ins(T_l, k), k', (T_r, 0), \Delta) & : & k < k' \\ tree((T_l, 0), k', ins(T_r, k), \Delta) & : & otherwise \end{cases} \tag{4.9}$$

T_l, T_r, k', Δ represent the left child, right child, the key and the balance factor of a tree.

$$\begin{aligned} T_l &= left(T) \\ T_r &= right(T) \\ k' &= key(T) \\ \Delta &= \delta(T) \end{aligned}$$

When we insert a new key k to a AVL tree T, if the tree is empty, we create a leaf with k as the key, set the balance factor as 0, and the height is increased by one.

If T isn't empty, we need compare the key k' with k. If k is less than the key, we recursively insert it to the left child, otherwise we insert it to the right.

[2]Note that, it doesn't mean δ is in range $[-1, 1]$, the changes of δ is in this range.

As the result of the recursive insertion is a pair like $(T'_l, \Delta H_l)$, we need do balance adjustment and update the increment of height. Function $tree()$ is defined to dealing with this task. It takes 4 parameters as $(T'_l, \Delta H_l)$, k', $(T'_r, \Delta H_r)$, and Δ. The result of this function is defined as $(T', \Delta H)$, where T' is the new tree after adjustment, and ΔH is the new increment of height. It is defined as below.

$$\Delta H = |T'| - |T| \tag{4.10}$$

This can be further detailed deduced in 4 cases.

$$\begin{aligned}\Delta H &= |T'| - |T| \\ &= 1 + max(|T'_r|, |T'_l|) - (1 + max(|T_r|, |T_l|)) \\ &= max(|T'_r|, |T'_l|) - max(|T_r|, |T_l|) \\ &= \begin{cases} \Delta H_r & : \quad \Delta \geq 0 \wedge \Delta' \geq 0 \\ \Delta + \Delta H_r & : \quad \Delta \leq 0 \wedge \Delta' \geq 0 \\ \Delta H_l - \Delta & : \quad \Delta \geq 0 \wedge \Delta' \leq 0 \\ \Delta H_l & : \quad otherwise \end{cases}\end{aligned} \tag{4.11}$$

The proof of this equation can be referred from Appendix C.

The next problem is to determine the new balance factor Δ' before performing balance adjustment. According to the definition of AVL tree, the balance factor is the height difference of the right and left sub trees. We have the following fact.

$$\begin{aligned}\Delta' &= |T'_r| - |T'_l| \\ &= |T_r| + \Delta H_r - (|T_l| + \Delta H_l) \\ &= |T_r| - |T_l| + \Delta H_r - \Delta H_l \\ &= \Delta + \Delta H_r - \Delta H_l\end{aligned} \tag{4.12}$$

With all these changes in height and the balance factor, we can define the $tree()$ function mentioned in (4.9).

$$tree((T'_l, \Delta H_l), k, (T'_r, \Delta H_r), \Delta) = balance((T'_l, k, T'_r, \Delta'), \Delta H) \tag{4.13}$$

Before we moving into details of balance adjustment, let's translate the above equations to example Haskell program.

First is the insert function.

```
insert::(Ord a)⇒AVLTree a → a → AVLTree a
insert t x = fst $ ins t where
    ins Empty = (Br Empty x Empty 0, 1)
    ins (Br l k r d)
        | x < k     = tree (ins l) k (r, 0) d
        | x == k    = (Br l k r d, 0)
        | otherwise = tree (l, 0) k (ins r) d
```

Here we also handle the duplicated keys (the key has already existed.) by overwriting.

```
tree::(AVLTree a, Int) → a → (AVLTree a, Int) → Int → (AVLTree a, Int)
tree (l, dl) k (r, dr) d = balance (Br l k r d', delta) where
    d' = d + dr - dl
    delta = deltaH d d' dl dr
```

www.dbooks.org

And the definition of height increment is as below.

```
deltaH :: Int → Int → Int → Int → Int
deltaH d d' dl dr
       | d ≥0 && d' ≥0 = dr
       | d ≤0 && d' ≥0 = d+dr
       | d ≥0 && d' ≤0 = dl - d
       | otherwise = dl
```

4.3.1 Balancing adjustment

As the pattern matching approach is adopted in doing re-balancing. We need consider what kind of patterns violate the AVL tree property.

Figure 4.3 shows the 4 cases which need fix. For all these 4 cases the balancing factors are either -2, or +2 which exceed the range of $[-1, 1]$. After balancing adjustment, this factor turns to be 0, which means the height of left sub tree is equal to the right sub tree.

We call these four cases left-left lean, right-right lean, right-left lean, and left-right lean cases in clock-wise direction from top-left. We denote the balancing factor before fixing as $\delta(x), \delta(y)$, and $\delta(z)$, while after fixing, they changes to $\delta'(x), \delta'(y)$, and $\delta'(z)$ respectively.

After fixing, we have $\delta(y) = 0$ for all four cases. The result values of $\delta'(x)$ and $\delta'(z)$ can be given as below. The proof are provided in Appendix C.

Left-left lean

$$\begin{aligned} \delta'(x) &= \delta(x) \\ \delta'(y) &= 0 \\ \delta'(z) &= 0 \end{aligned} \tag{4.14}$$

Right-right lean

$$\begin{aligned} \delta'(x) &= 0 \\ \delta'(y) &= 0 \\ \delta'(z) &= \delta(z) \end{aligned} \tag{4.15}$$

Right-left lean and Left-right lean

$$\begin{aligned} \delta'(x) &= \begin{cases} -1 & : \quad \delta(y) = 1 \\ 0 & : \quad otherwise \end{cases} \\ \delta'(y) &= 0 \\ \delta'(z) &= \begin{cases} 1 & : \quad \delta(y) = -1 \\ 0 & : \quad otherwise \end{cases} \end{aligned} \tag{4.16}$$

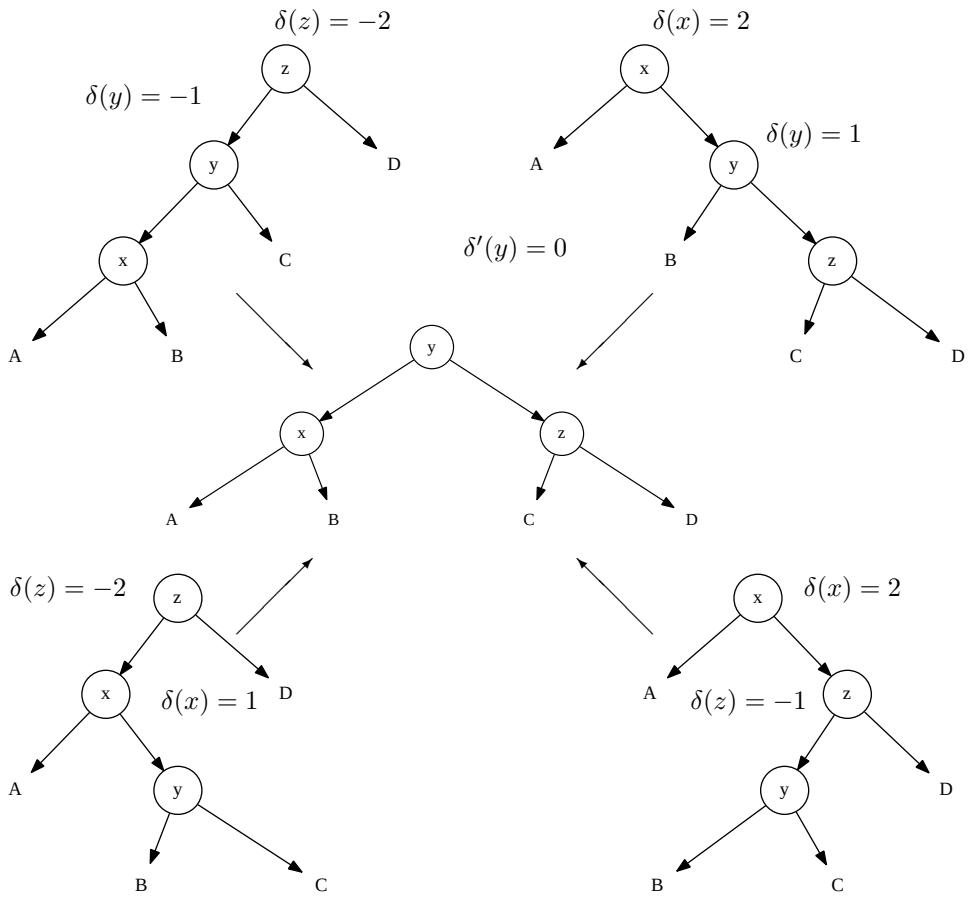


Figure 4.3: 4 cases for balancing a AVL tree after insertion

www.dbooks.org

4.3.2 Pattern Matching

The pattern matching fixing function can be given as the following.

$$balance(T, \Delta H) = \begin{cases} (((A, x, B, \delta(x)), y, (C, z, D, 0), 0), \Delta H - 1) & : & P_{ll}(T) \\ (((A, x, B, 0), y, (C, z, D, \delta(z)), 0), \Delta H - 1) & : & P_{rr}(T) \\ (((A, x, B, \delta'(x)), y, (C, z, D, \delta'(z)), 0), \Delta H - 1) & : & P_{rl}(T) \vee P_{lr}(T) \\ (T, \Delta H) & : & otherwise \end{cases} \tag{4.17}$$

Where $P_{ll}(T)$ means the pattern of tree T is left-left lean respectively. $\delta'(x)$ and $delta'(z)$ are defined in (C.16). The four patterns are tested as below.

$$\begin{aligned} &P_{ll}(T) : T = (((A, x, B, \delta(x)), y, C, -1), z, D, -2) \\ &P_{rr}(T) : T = (A, x, (B, y, node(C, z, D, \delta(z)), 1), 2) \\ &P_{rl}(T) : T = ((A, x, (B, y, C, \delta(y)), 1), z, D, -2) \\ &P_{lr}(T) : T = (A, x, ((B, y, C, \delta(y)), z, D, -1), 2) \end{aligned} \tag{4.18}$$

Translating the above function definition to Haskell yields a simple and intuitive program.

```
balance (Br (Br (Br a x b dx) y c (-1)) z d (-2), _) =
        (Br (Br a x b dx) y (Br c z d 0) 0, 0)
balance (Br a x (Br b y (Br c z d dz)    1)    2, _) =
        (Br (Br a x b 0) y (Br c z d dz) 0, 0)
balance (Br (Br a x (Br b y c dy)    1) z d (-2), _) =
        (Br (Br a x b dx') y (Br c z d dz') 0, 0) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
balance (Br a x (Br (Br b y c dy) z d (-1))    2, _) =
        (Br (Br a x b dx') y (Br c z d dz') 0, 0) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
balance (t, d) = (t, d)
```

The insertion algorithm takes time proportion to the height of the tree. As AVL is balanced according to (4.7), its performance is $O(\lg n)$ where n is the number of elements stored in the AVL tree.

Verification

When verify if a tree is AVL tree, we need verify two things, first, it's a binary search tree; second, it satisfies AVL tree property.

In order to test if a binary tree satisfies AVL tree property, we can examine the height difference between the two sub trees recursively till the leaves.

$$avl?(T) = \begin{cases} True & : & T = \phi \\ avl?(T_l) \wedge avl?(T_r) \wedge ||T_r| - |T_l|| \leq 1 & : & otherwise \end{cases} \tag{4.19}$$

Where the height can also be calculated recursively.

$$|T| = \begin{cases} 0 & : & T = \phi \\ 1 + max(|T_r|, |T_l|) & : & otherwise \end{cases} \tag{4.20}$$

The corresponding Haskell example program is given as the following.

```
isAVL :: (AVLTree a) → Bool
isAVL Empty = True
isAVL (Br l _ r d) = and [isAVL l, isAVL r, abs (height r - height l) ≤ 1]

height :: (AVLTree a) → Int
height Empty = 0
height (Br l _ r _) = 1 + max (height l) (height r)
```

Exercise 4.1

Write a program to verify if a tree is the AVL tree. Please consider both functional and imperative approaches.

4.4 Deletion

As we mentioned before, deletion will not be a major problem in purely functional settings. As the tree is read only, the use case is typically performing looking up after build.

For purely functional deletion, it actually re-builds the tree as we show in the chatper of red-black tree. We put the AVL tree deletion algorithm in Appendix C.

4.5 Imperative AVL tree algorithm ⋆

This section shows the traditional insert-and-rotate approach to realize AVL tree insertion algorithm.

Similar to the red-black tree algorithm, the strategy is to first do the binary search tree insertion, then fix the balance by rotation and return the final result.

```
1: function INSERT(T, k)
2:     root ← T
3:     x ← CREATE-LEAF(k)
4:     δ(x) ← 0
5:     parent ← NIL
6:     while T ≠ NIL do
7:         parent ← T
8:         if k < KEY(T) then
9:             T ← LEFT(T)
10:        else
11:            T ← RIGHT(T)
12:    PARENT(x) ← parent
13:    if parent = NIL then                          ▷ tree T is empty
14:        return x
15:    else if k < KEY(parent) then
16:        LEFT(parent) ← x
17:    else
18:        RIGHT(parent) ← x
19:    return AVL-INSERT-FIX(root, x)
```

www.dbooks.org

Note that after insertion, the balance factor δ may change because the height of the tree can grow. Inserting on right side can increase δ by 1, while insert on left side can decrease it. By the end of this algorithm, we need perform bottom-up fixing from node x towards root.

We can translate the pseudo code to Python example program[3].

```
def avl_insert(t, key):
    root = t
    x = Node(key)
    parent = None
    while(t):
        parent = t
        if(key < t.key):
            t = t.left
        else:
            t = t.right
    if parent is None: #tree is empty
        root = x
    elif key < parent.key:
        parent.set_left(x)
    else:
        parent.set_right(x)
    return avl_insert_fix(root, x)
```

This is a top-down algorithm. It searches the tree from root down to the proper position and inserts the new key as a leaf. By the end of this algorithm, it calls the fixing function with the root and the new inserted node.

Note that we reuse the same methods of `set_left()` and `set_right()` as we defined in chapter of red-black tree.

In order to resume the AVL tree property, we first check if the new node is inserted on left or right. If it is on left, the balance factor δ decreases, otherwise it increases. If we denote the new value as δ', there are 3 cases between δ and δ'.

- If $|\delta| = 1$ and $|\delta'| = 0$, it means the new node makes the tree perfectly balanced, the height of the parent node doesn't change, the algorithm can be terminated.
- If $|\delta| = 0$ and $|\delta'| = 1$, it means either the left or the right sub tree increases its height. We need go on checking the upper level of the tree.
- If $|\delta| = 1$ and $|\delta'| = 2$, it means the AVL tree property is violated due to the new insertion. We need perform rotation to fix it.

```
1: function AVL-Insert-Fix(T, x)
2:     while Parent(x) ≠ NIL do
3:         δ ← δ(Parent(x))
4:         if x = Left(Parent(x)) then
5:             δ' ← δ − 1
6:         else
7:             δ' ← δ + 1
8:         δ(Parent(x)) ← δ'
```

[3]C and C++ source code are available along with this book

```
        P ← Parent(x)
        L ← Left(x)
        R ← Right(x)
        if |δ| = 1 and |δ'| = 0 then          ▷ Height doesn't change, terminates.
            return T
        else if |δ| = 0 and |δ'| = 1 then          ▷ Go on bottom-up updating.
            x ← P
        else if |δ| = 1 and |δ'| = 2 then
            if δ' = 2 then
                if δ(R) = 1 then                                  ▷ Right-right case
                    δ(P) ← 0                                          ▷ By (C.5)
                    δ(R) ← 0
                    T ← Left-Rotate(T, P)
                if δ(R) = −1 then                                 ▷ Right-left case
                    δ_y ← δ(Left(R))                                  ▷ By (C.16)
                    if δ_y = 1 then
                        δ(P) ← −1
                    else
                        δ(P) ← 0
                    δ(Left(R)) ← 0
                    if δ_y = −1 then
                        δ(R) ← 1
                    else
                        δ(R) ← 0
                    T ← Right-Rotate(T, R)
                    T ← Left-Rotate(T, P)
            if δ' = −2 then
                if δ(L) = −1 then                                 ▷ Left-left case
                    δ(P) ← 0
                    δ(L) ← 0
                    Right-Rotate(T, P)
                else                                              ▷ Left-Right case
                    δ_y ← δ(Right(L))
                    if δ_y = 1 then
                        δ(L) ← −1
                    else
                        δ(L) ← 0
                    δ(Right(L)) ← 0
                    if δ_y = −1 then
                        δ(P) ← 1
                    else
                        δ(P) ← 0
                    Left-Rotate(T, L)
                    Right-Rotate(T, P)
            break
    return T
```

As rotation operation doesn't update the balance factor δ, we need update it for impacted nodes. Among the four cases, the right-right case and the left-left

www.dbooks.org

case need only one rotation, while the right-left case and the left-right case need two rotations.

The relative example python program is as the following.

```
def avl_insert_fix(t, x):
    while x.parent is not None:
        d2 = d1 = x.parent.delta
        if x == x.parent.left:
            d2 = d2 - 1
        else:
            d2 = d2 + 1
        x.parent.delta = d2
        (p, l, r) = (x.parent, x.parent.left, x.parent.right)
        if abs(d1) == 1 and abs(d2) == 0:
            return t
        elif abs(d1) == 0 and abs(d2) == 1:
            x = x.parent
        elif abs(d1)==1 and abs(d2) == 2:
            if d2 == 2:
                if r.delta == 1:  # Right-right case
                    p.delta = 0
                    r.delta = 0
                    t = left_rotate(t, p)
                if r.delta == -1: # Right-Left case
                    dy = r.left.delta
                    if dy == 1:
                        p.delta = -1
                    else:
                        p.delta = 0
                    r.left.delta = 0
                    if dy == -1:
                        r.delta = 1
                    else:
                        r.delta = 0
                    t = right_rotate(t, r)
                    t = left_rotate(t, p)
            if d2 == -2:
                if l.delta == -1: # Left-left case
                    p.delta = 0
                    l.delta = 0
                    t = right_rotate(t, p)
                if l.delta == 1: # Left-right case
                    dy = l.right.delta
                    if dy == 1:
                        l.delta = -1
                    else:
                        l.delta = 0
                    l.right.delta = 0
                    if dy == -1:
                        p.delta = 1
                    else:
                        p.delta = 0
                    t = left_rotate(t, l)
                    t = right_rotate(t, p)
```

```
            break
    return t
```

We put the AVL tree deletion algorithm in appendix C for reference.

4.6 Chapter note

AVL tree was invented in 1962 by Adelson-Velskii and Landis[3], [4]. The name AVL tree comes from the two inventor's name. It's earlier than red-black tree.

It's very common to compare AVL tree and red-black tree, both are self-balancing binary search trees, and for all the major operations, they both consume $O(\lg n)$ time. From the result of (4.7), AVL tree is more rigidly balanced hence they are faster than red-black tree in looking up intensive applications [3]. However, red-black trees could perform better in frequently insertion and removal cases.

Many popular self-balancing binary search tree libraries are implemented on top of red-black tree such as STL etc. However, AVL tree provides an intuitive and effective solution to the balance problem as well.

After this chapter, we'll extend the tree data structure from storing data in node to storing information on edges. It leads to Radix trees. If we extend the number of children from two to more, we can get B-tree. These data structures will be introduced in the next chapters.

www.dbooks.org

Bibliography

[1] Data.Tree.AVL http://hackage.haskell.org/packages/archive/AvlTree/4.2/doc/html/Data-Tree-AVL.html

[2] Chris Okasaki. "FUNCTIONAL PEARLS Red-Black Trees in a Functional Setting". J. Functional Programming. 1998

[3] Wikipedia. "AVL tree". http://en.wikipedia.org/wiki/AVL_tree

[4] Guy Cousinear, Michel Mauny. "The Functional Approach to Programming". Cambridge University Press; English Ed edition (October 29, 1998). ISBN-13: 978-0521576819

[5] Pavel Grafov. "Implementation of an AVL tree in Python". http://github.com/pgrafov/python-avl-tree

www.dbooks.org

Chapter 5

Radix tree, Trie and Prefix Tree

5.1 Introduction

The binary trees introduced so far store information in nodes. Edge can also be used to store information. Radix trees including Trie and prefix tree are important data structures in information retrieving and manipulating. They were found in 1960s. And are widely used in compiler design[2], and bio-information area, such as DNA pattern matching [3].

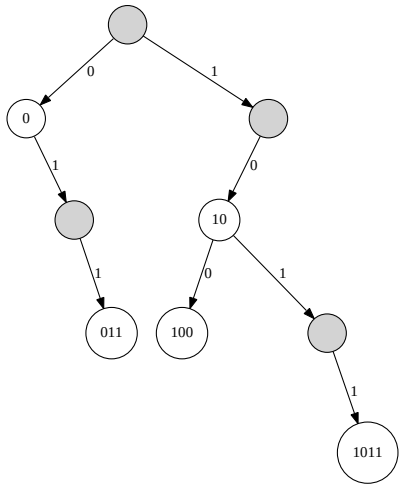


Figure 5.1: Radix tree.

Figure 5.1 shows a radix tree([2] pp. 269). It contains strings of bit 1011, 10, 011, 100 and 0. When searching a key $k = (b_0 b_1 ... b_n)_2$, we take the first bit b_0 (MSB from left), check if it is 0 or 1, if it is 0, we turn left, else turn right for 1. Then we take the second bit and repeat this search till either meet a leaf node or finish all the n bits.

The radix tree needn't store keys in node at all. The information is repre-

www.dbooks.org

sented by edges. The nodes marked with keys in the above figure are only for illustration purpose.

Another idea is to represent the key in integer instead of string. Because integer can be in binary format to save space. The speed is also fast as we can use bit-wise manipulation in most programming environments.

5.2 Integer Trie

The data structure shown in figure 5.1 is often called as *binary trie*. Trie is invented by Edward Fredkin. It comes from "retrieval", pronounce as /'tri:/ by the inventor, while it is pronounced /'trai/ "try" by other authors [5]. Trie is also called prefix tree. A binary trie is a special binary tree in which the placement of each key is controlled by its bits, each 0 means 'go left' and each 1 means 'go right'[2].

Because integer can be represented in binary format, we can use it instead of 0, 1 string. When insert a new integer to the trie, we change it to binary form, then examine the first bit, if it is 0, we recursively insert the rest bits to the left sub-tree; otherwise if it is 1, we insert into the right sub-tree.

There is a problem when treat the key as integer. Consider a binary trie shown in figure 5.2. If represented in 0, 1 strings, all the three keys are different although they are equal integers. Where should we insert decimal 3 to this trie?

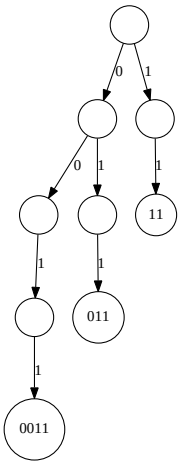


Figure 5.2: A big-endian trie.

One approach is to treat all the prefix zero as effective bits. Suppose the integer is represented with 32-bits, If we want to insert key 1, it ends up with a tree of 32 levels. There are 31 nodes, each only has the left sub-tree. the last node only has the right sub-tree. It is very inefficient in terms of space.

Okasaki shows a method to solve this problem in [2]. Instead of using big-endian integer, we can use the little-endian integer to represent key. Thus decimal integer 1 is represented as binary 1. When insert it to the empty binary trie, the result is a trie with a root and a right leaf. There is only 1 level. decimal 2 is represented as 01, and decimal 3 is $(11)_2$ in little-endian binary

format. There is no need to add any prefix 0, the position in the trie is uniquely determined.

5.2.1 Definition of integer Trie

We can use the binary tree structure to define the littel-endian binary trie. A binary trie node is either empty, or a branch. The branch node contains a left child, a right node, and optional value as the satellite data. The left sub-tree is encoded as 0 and the right sub-tree is encoded as 1.

The following example Haskell code defines the integer trie as algebraic data type.

```
data IntTrie a = Empty
               | Branch (IntTrie a) (Maybe a) (IntTrie a)
```

The below Python example provides the corresponding imperative definition.

```
class IntTrie:
    def __init__(self):
        self.left = self.right = None
        self.value = None
```

5.2.2 Insertion

Because the definition of the integer trie is recursive, it's strightforward to define the insertion algorithm recursively. If the lowest bit is 0, the key to be inserted is even, we recursively insert it to the left sub-tree; otherwise if the lowest bit is 1, the key is odd, then the recursive insertion is applied to the right. we next divide the key by 2 to get rid of the lowest bit. For trie T, denote the left and right sub-trees as T_l and T_r respectively. Thus $T = (T_l, v', T_r)$, where v' is the optional satellite data. If T is empty, then T_l, T_r and v' are defined as empty as well.

$$insert(T, k, v) = \begin{cases} (T_l, v, T_r) & : \quad k = 0 \\ (insert(T_l, k/2, v), v', T_r) & : \quad even(k) \\ (T_l, v', insert(T_r, \lfloor k/2 \rfloor, v)) & : \quad otherwise \end{cases} \tag{5.1}$$

If the key to be inserted already exists, this algorithm just overwrites the previous stored data. It can be replaced with other alternatives, such as to store the data in a linked-list.

Figure 5.3 shows an example trie. It's generated by inserting the key-value pairs $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$ to the empty trie.

The following Haskell example program implements the insertion algorithm.

```
insert t 0 x = Branch (left t) (Just x) (right t)
insert t k x
    | even k = Branch (insert (left t) (k `div` 2) x) (value t) (right t)
    | otherwise = Branch (left t) (value t) (insert (right t) (k `div` 2) x)

left (Branch l _ _) = l
left Empty = Empty

right (Branch _ _ r) = r
```

www.dbooks.org

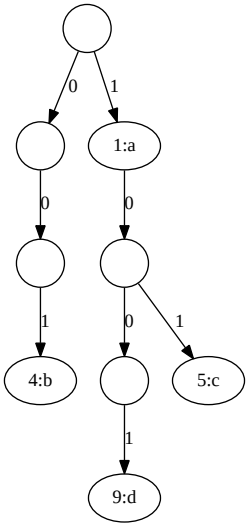


Figure 5.3: A little-endian integer binary trie for the map $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$.

```
right Empty = Empty

value (Branch _ v _) = v
value Empty = Nothing
```

We can also define the insertion algorithm imperatively. As the key is is stored as little-endian integer, when insert a new key, we extract the bit one by one from the right most. If it is 0, we go to the left, otherwise for 1, we go to the right. If the sub-tree is empty, we need create a new node, and repeat this to the last bit of the key.

```
1:  function INSERT(T, k, v)
2:      if T = NIL then
3:          T ← EMPTY-NODE
4:      p ← T
5:      while k ≠ 0 do
6:          if EVEN?(k) then
7:              if LEFT(p) = NIL then
8:                  LEFT(p) ← EMPTY-NODE
9:              p ← LEFT(p)
10:         else
11:             if RIGHT(p) = NIL then
12:                 RIGHT(p) ← EMPTY-NODE
13:             p ← RIGHT(p)
14:         k ← ⌊k/2⌋
15:     DATA(p) ← v
16:     return T
```

This algorithm takes 3 arguments, a Trie T, a key k, and the satellite data v. The following example Python program implements the insertion algorithm.

It uses bit-wise operation to test whether a number is even or odd, and shift the bit to right as division.

```
def insert(t, key, value = None):
    if t is None:
        t = IntTrie()
    p = t
    while key != 0:
        if key & 1 == 0:
            if p.left is None:
                p.left = IntTrie()
            p = p.left
        else:
            if p.right is None:
                p.right = IntTrie()
            p = p.right
        key = key >> 1  # key / 2
    p.value = value
    return t
```

For a given integer k with m bits in binary, the insertion algorithm goest into m levels. The performance is bound to $O(m)$ time.

5.2.3 Look up

To look up key k in the little-endian integer binary trie, if the trie is empty, the looking up fails; if $k = 0$, then we return the data stored in the current node; if the last bit is 0, we recursively look up the left sub-tree; otherwise we look up the right sub-tree.

$$lookup(T, k) = \begin{cases} \phi & : & T = \phi \\ d & : & k = 0 \\ lookup(T_l, k/2) & : & even(k) \\ lookup(T_r, \lfloor k/2 \rfloor) & : & otherwise \end{cases} \tag{5.2}$$

The following Haskell example program implements the recursive look up algorithm.

```
search Empty k = Nothing
search t 0 = value t
search t k = if even k then search (left t) (k `div` 2)
             else search (right t) (k `div` 2)
```

The look up algorithm can also be realized imperatively. We examine each bit of k from the lowest one. We go left if the bit is 0, otherwise, go right. The looking up completes when all bits are consumed.

```
1: function LOOKUP(T, k)
2:     while k ≠ 0 ∧ T ≠NIL do
3:         if EVEN?(k) then
4:             T ← LEFT(T)
5:         else
6:             T ← RIGHT(T)
7:         k ← ⌊k/2⌋
8:     if T ≠ NIL then
```

www.dbooks.org

```
9:          return DATA(T)
10:     else
11:         return not found
```

Below example Python program implements the looking up algorithm.

```
def lookup(t, key):
    while t is not None and k != 0:
        if key & 1 == 0:
            t = t.left
        else:
            t = t.right
        key = key >> 1
    return None if t is None else t.value
```

The looking up algorithm is bound to $O(m)$ time, where m is the number of bits of the key.

5.3 Integer prefix tree

Trie has some drawbacks. It occupies a lot of spaces. As shown in figure 5.3, the real data is mostly stored in leafs. It's very common that an integer binary trie contains many nodes only have one child. One idea is to compress the chained nodes to one. Integer prefix tree is such a data structure invented by Donald R. Morrison in 1968, who named it as 'Patricia'. It stands for **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric[3]. It is another kind of prefix tree. We call it integer tree in this book.

Okasaki provided the implementation of integer tree in [2]. If merge the chained nodes which have only one child together in figure 5.3, we can get a integer tree as shown in figure 5.4.

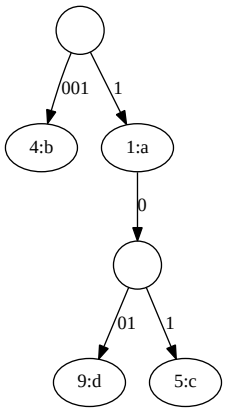


Figure 5.4: Little endian integer tree for the map $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$.

From this figure, we can find the key of the branch node is the longest common prefix for its descendant trees. They branches out at certain bit. Integer tree saves a lot of space compare to trie.

Different from integer trie, padding bits of zero don't cause issue with the big endian integer tree. All zero bits before MSB are omitted to save the space.

Okasaki list some significant advantages of big endian integer tree in [2].

5.3.1 Definition

Integer prefix tree is a special binary tree. It is either empty or a node. There are two different types of node:

- A leaf contains integer key and optional satellite data;
- Or a branch node with the left and right sub-trees. The two children share the **longest common prefix** bits for their keys. For the left child, the next bit in the key is zero, while it's one for the right child.

The following Haskell example code defines integer tree accordingly.

```
type Key = Int
type Prefix = Int
type Mask = Int

data IntTree a = Empty
               | Leaf Key a
               | Branch Prefix Mask (IntTree a) (IntTree a)
```

In the branch node, we use a mask number to tell from which bit the sub-trees differ. The mask is power of 2, which is 2^n for some non-negative integer n, all bits that are lower than n don't belong to the common prefix.

The following example Python code defines integer tree with auxiliary functions.

```
class IntTree:
    def __init__(self, key = 0, value = None):
        self.key = key
        self.value = value
        self.prefix = key
        self.mask = 1
        self.left = self.right = None

    def isleaf(self):
        return self.left is None and self.right is None

    def replace(self, x, y):
        if self.left == x:
            self.left = y
        else:
            self.right = y

    def match(self, k):
        return maskbit(k, self.mask) == self.prefix
```

Where match tests if the prefix stored in the node are same as the given key before the mask bit. It's explained in the next section.

5.3.2 Insertion

When insert a key, if the tree is empty, we create a leaf node as shown in figure 5.5.

www.dbooks.org



Figure 5.5: Left: the empty tree; Right: After insert key 12.

If the tree is a singleton leaf node x, we create a new leaf y, put the key and the value into it. After that, we need create a new branch node, set x and y as the two sub-trees. In order to determine if y should be on the left or right, we need find the longest common prefix of x and y. For example if $key(x)$ is 12 ($(1100)_2$ in binary), $key(y)$ is 15 ($(1111)_2$ in binary), then the longest common prefix is $(11oo)_2$. Where o denotes the bits we don't care about. We can use another integer to mask those bits. In this case, the mask number is 4 (100 in binary). The next bit after the longest common prefix presents 2^1. This bit is 0 in $key(x)$, while it is 1 in $key(y)$. We should set x as the left sub-tree and y as the right sub-tree. Figure 5.6 shows this example.

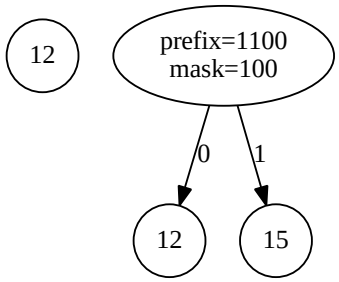


Figure 5.6: Left: A tree with a singleton leaf 12; Right: After insert key 15.

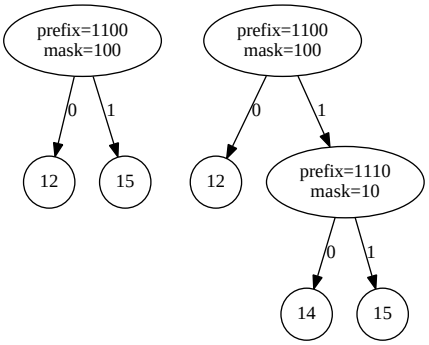
In case the tree is neither empty, nor a singleton leaf, we need firstly check if the key to be inserted matches the longest common prefix recorded in the root. Then recursively insert the key to the left or right according to the next bit of the longest common prefix. For example, if insert key 14 ($(1110)_2$ in binary) to the result tree in figure 5.6, since the common prefix is $(11oo)_2$, and the next bit (the bit of 2^1) is 1, we need recursively insert to the right sub-tree.

If the key to be inserted doesn't match the longest common prefix in the root, we need branch a new leaf out. Figure 5.7 shows these two different cases.

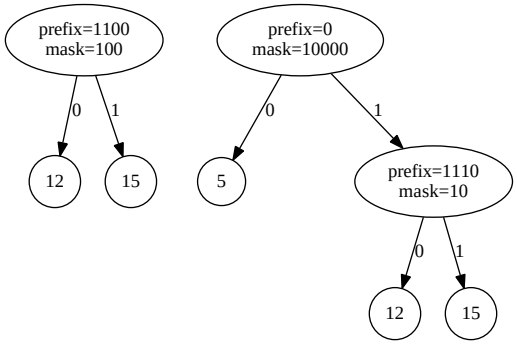
For a given key k and value v, denote (k, v) as the leaf node. For branch node, denote it in form of (p, m, T_l, T_r), where p is the longest common prefix, m is the mask, T_l and T_r are the left and right sub-trees. Summarize the above cases, the insertion algorithm can be defined as below.

$$insert(T, k, v) = \begin{cases} (k, v) & : & T = \phi \vee T = (k, v') \\ join(k, (k, v), k', T) & : & T = (k', v') \\ (p, m, insert(T_l, k, v), T_r) & : & T = (p, m, T_l, T_r), match(k, p, m), zero(k, m) \\ (p, m, T_l, insert(T_r, k, v)) & : & T = (p, m, T_l, T_r), match(k, p, m), \neg zero(k, m) \\ join(k, (k, v), p, T) & : & T = (p, m, T_l, T_r), \neg match(k, p, m) \end{cases} \tag{5.3}$$

The first clause deals with the edge cases, if T is empty, the result is a leaf node. If T is a leaf node with the same key, we overwrite the previous value.



(a) Insert key 14. It matches the longest common prefix $(1100)_2$; 14 is then recursively inserted to the right sub-tree.



(b) Insert key 5. It doesn't match the longest common prefix $(1100)_2$, a new leaf is branched out.

Figure 5.7: Insert key to the branch node.

www.dbooks.org

The second clause handles the case that T is a leaf node, but with different key. Here we branch out another leaf, then extract the longest common prefix, and determine which leaf should be set as the left sub-tree. Function $join(k_1, T_1, k_2, T_2)$ does this work. We'll define it later.

The third clause deals with the case that T is a branch node, the longest common prefix matches the key to be inserted, and the next bit to the common prefix is zero. Here we need recursively insert to the left sub-tree.

The fourth clause handles the similar case as the third one, except that the next bit to the common prefix is one, but not zero. We need recursively insert to the right sub-tree.

The last clause is for the case that the key to be inserted doesn't match the longest common prefix in the branch. We need branch out a new leaf by calling the $join$ function.

We need define function $match(k, p, m)$ to test if the key k, has the same prefix p above the masked bits m. For example, suppose the prefix stored in a branch node is $(p_n p_{n-1} ... p_i ... p_0)_2$ in binary, key k is $(k_n k_{n-1} ... k_i ... k_0)_2$ in binary, and the mask is $(100...0)_2 = 2^i$. They match if and only if $p_j = k_j$ for all j, that $i \leq j \leq n$.

One solution to realize match is to test if $mask(k, m) = p$ is satisfied. Where $mask(x, m) = \overline{m-1}\&x$, that we perform bitwise-not of $m - 1$, then perform bitwise-and with x.

Function $zero(k, m)$ test the next bit of the common prefix is zero. With the help of the mask m, we can shift m one bit to the right, then perform bitwise-and with the key.

$$zero(k, m) = x\&shift_r(m, 1) \tag{5.4}$$

If the mask $m = (100..0)_2 = 2^i$, $k = (k_n k_{n-1} ... k_i 1 ... k_0)_2$, because the bit next to k_i is 1, $zero(k, m)$ returns false value; if $k = (k_n k_{n-1} ... k_i 0 ... k_0)_2$, then the result is true.

Function $join(p_1, T_1, p_2, T_2)$ takes two different prefixes and trees. It extracts the longest common prefix of p_1 and p_2, create a new branch node, and set T_1 and T_2 as the two sub-trees.

$$join(p_1, T_1, p_2, T_2) = \begin{cases} (p, m, T_1, T_2) & : \quad zero(p1, m), (p, m) = LCP(p_1, p_2) \\ (p, m, T_2, T_1) & : \quad \neg zero(p1, m) \end{cases} \tag{5.5}$$

In order to calculate the longest common prefix of p_1 and p_2, we can firstly compute bitwise exclusive-or for them, then count the number of bits in this result, and generate a mask $m = 2^{|xor(p_1, p_2)|}$. The longest common prefix p can be given by masking the bits with m for either p_1 or p_2.

$$p = mask(p_1, m) \tag{5.6}$$

The following Haskell example program implements the insertion algorithm.

```
import Data.Bits

insert t k x
  = case t of
      Empty → Leaf k x
```

```
      Leaf k' x' → if k==k' then Leaf k x
                   else join k (Leaf k x) k' t -- t@(Leaf k' x')
      Branch p m l r
         | match k p m → if zero k m
                         then Branch p m (insert l k x) r
                         else Branch p m l (insert r k x)
         | otherwise → join k (Leaf k x) p t -- t@(Branch p m l r)

join p1 t1 p2 t2 = if zero p1 m then Branch p m t1 t2
                                else Branch p m t2 t1
    where
      (p, m) = lcp p1 p2

lcp :: Prefix → Prefix → (Prefix, Mask)
lcp p1 p2 = (p, m) where
    m = bit (highestBit (p1 `xor` p2))
    p = mask p1 m

highestBit x = if x == 0 then 0 else 1 + highestBit (shiftR x 1)

mask x m = (x .&. complement (m-1)) -- complement means bit-wise not.

zero x m = x .&. (shiftR m 1) == 0

match k p m = (mask k m) == p
```

The insertion algorithm can also be realized imperatively.

```
1:  function INSERT(T, k, v)
2:      if T = NIL then
3:          T ← CREATE-LEAF(k, v)
4:          return T
5:      y ← T
6:      p ← NIL
7:      while y is not leaf, and MATCH(k, PREFIX(y), MASK(y)) do
8:          p ← y
9:          if ZERO?(k, MASK(y)) then
10:             y ← LEFT(y)
11:         else
12:             y ← RIGHT(y)
13:     if y is leaf, and k = KEY(y) then
14:         DATA(y) ← v
15:     else
16:         z ← BRANCH(y, CREATE-LEAF(k, v))
17:         if p = NIL then
18:             T ← z
19:         else
20:             if LEFT(p) = y then
21:                 LEFT(p) ← z
22:             else
23:                 RIGHT(p) ← z
24:     return T
```

www.dbooks.org

Function BRANCH(T_1, T_2) does the similar job as what *join* is defined. It creates a new branch node, extracts the longest common prefix, sets T_1 and T_2 as the two sub-trees.

```
1: function BRANCH(T1, T2)
2:     T ← EMPTY-NODE
3:     ( PREFIX(T), MASK(T) ) ← LCP(PREFIX(T1), PREFIX(T2))
4:     if ZERO?(PREFIX(T1), MASK(T)) then
5:         LEFT(T) ← T1
6:         RIGHT(T) ← T2
7:     else
8:         LEFT(T) ← T2
9:         RIGHT(T) ← T1
10:    return T
```

The following Python example program implements the insertion algorithm.

```
def insert(t, key, value):
    if t is None:
        return IntTree(key, value)
    node = t
    parent = None
    while (not node.isleaf()) and node.match(key):
        parent = node
        if zero(key, node.mask):
            node = node.left
        else:
            node = node.right
    if node.isleaf() and key == node.key:
        node.value = value
    else:
        p = branch(node, IntTree(key, value))
        if parent is None:
            return p
        parent.replace(node, p)
    return t
```

The auxiliary functions, `branch`, `lcp` etc. are given as below.

```
def maskbit(x, mask):
    return x & (~(mask - 1))

def zero(x, mask):
    return x & (mask >> 1) == 0

def lcp(p1, p2):
    diff = p1 ^ p2
    mask = 1
    while diff != 0:
        diff >>= 1
        mask <<= 1
    return (maskbit(p1, mask), mask)

def branch(t1, t2):
    t = IntTree()
    (t.prefix, t.mask) = lcp(t1.prefix, t2.prefix)
```

```
if zero(t1.prefix, t.mask):
    t.left, t.right = t1, t2
else:
    t.left, t.right = t2, t1
return t
```

Figure 5.8 shows the example integer tree created with the insertion algorithm.

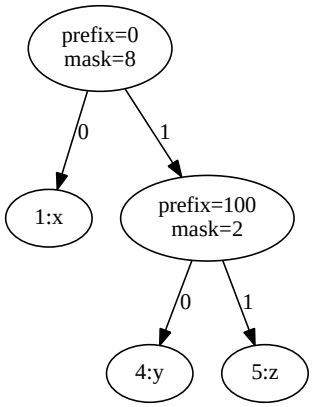


Figure 5.8: Insert map $1 \to x, 4 \to y, 5 \to z$ into the big-endian integer prefix tree.

5.3.3 Look up

If the integer tree T is empty, or it's a singleton leaf with the key that is different from what we are looking up, the result is empty. else if the key in the leaf equals, we are done. If T is a branch node, we need check if the common prefix matches the subject key, and recursively look up the sub-tree according to the next bit. If the common prefix doesn't match the key, then the lookup fails.

$$lookup(T, k) = \begin{cases} \phi & : & T = \phi \vee (T = (k', v), k' \neq k) \\ v & : & T = (k', v), k' = k \\ lookup(T_l, k) & : & T = (p, m, T_l, T_r), match(k, p, m), zero(k, m) \\ lookup(T_r, k) & : & T = (p, m, T_l, T_r), match(k, p, m), \neg zero(k, m) \\ \phi & : & otherwise \end{cases} \tag{5.7}$$

The following Haskell example program implements this recursive lookup up algorithm.

```
search t k
  = case t of
      Empty → Nothing
      Leaf k' x → if k == k' then Just x else Nothing
      Branch p m l r
             | match k p m → if zero k m then search l k
                             else search r k
             | otherwise → Nothing
```

www.dbooks.org

The look up algorithm can also be realized imperatively. Consider the property of integer prefix tree. When look up a key, if it has common prefix with the root, then we check the next bit. If this bit is zero, we then recursively look up the left sub-tree; otherwise we look up the right sub-tree if the bit is one.

When arrive at the leaf node, we check if the key of the leaf equals to the one we are looking up.

```
1: function LOOK-UP(T, k)
2:     if T = NIL then
3:         return NIL                                    ▷ Not found
4:     while T is not leaf, and MATCH(k, PREFIX(T), MASK(T)) do
5:         if ZERO?(k, MASK(T)) then
6:             T ← LEFT(T)
7:         else
8:             T ← RIGHT(T)
9:     if T is leaf, and KEY(T) = k then
10:        return DATA(T)
11:    else
12:        return NIL                                    ▷ Not found
```

Below Python example program implements the looking up algorithm.

```
def lookup(t, key):
    while t is not None and (not t.isleaf()) and t.match(key):
        if zero(key, t.mask):
            t = t.left
        else:
            t = t.right
    if t is not None and t.isleaf() and t.key == key:
        return t.value
    return None
```

5.4 Alphabetic Trie

Integer based trie and tree can be a good start point. The Glasgow Haskell Compiler (GHC) utilized the similar integer tree implementation for several years before 1998[2].

If we extend the key from integer to alphabetic value, Trie and integer tree can be very powerful in solving textual manipulation problems.

5.4.1 Definition

It's not enough to just use the left and right sub-trees to represent alphabetic keys. Taking English for example, there are 26 letters. If we don't care about the case, one solution is to limit the number of branches (children) to 26. Some simplified implementation defines the trie with the array of 26 letters. This can be illustrated as in Figure 5.9.

Not all the 26 branches contain data. For instance, in Figure 5.9, the root only has three non-empty branches representing letter 'a', 'b', and 'z'. Other branches such as for letter 'c', are all empty. We will not show empty branch in the future.

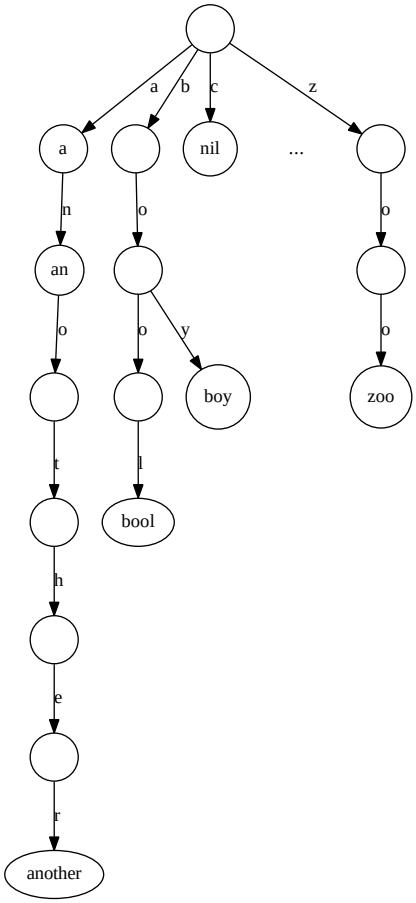


Figure 5.9: A trie with 26 branches, containing key 'a', 'an', 'another', 'bool', 'boy' and 'zoo'.

www.dbooks.org

When dealing with case sensitive problems, or handling languages other than English, there can be more letters. We can use the collection data structures, like Hash map to define the trie.

Alphabetic trie is either empty or a node. There are two types of node.

- A leaf node does not have any sub-trees;
- A branch node contains multiple sub-trees. Each sub-tree is bound to a character.

Both leaf and branch can contain optional satellite data. The following Haskell code shows the example definition.

```
data Trie a = Trie { value :: Maybe a
                   , children :: [(Char, Trie a)]}

empty = Trie Nothing []
```

Below ANSI C example code defines the alphabetic trie. For illustration purpose, it limits the character set to lower case English letters, from 'a' to 'z'.

```
struct Trie {
  struct Trie* children[26];
  void* data;
};
```

5.4.2 Insertion

When insert to the trie, denote the key to be inserted as $K = k_1 k_2 ... k_n$, where k_i is the i-th character. K' is the rest of characters except k_1, v' is the data to be inserted. The trie is in form $T = (v, C)$, where v is the data store in the trie, $C = \{(c_1, T_1), (c_2, T_2), ..., (c_m, T_m)\}$ is the collection of sub-trees. It associates a character c_i and the corresponding sub-tree T_i. C is empty for leaf node.

$$insert(T, K, v') = \begin{cases} (v', C) & : & K = \phi \\ (v, ins(C, k_1, K', v')) & : & otherwise. \end{cases} \tag{5.8}$$

If the key is empty, the previous value v is overwritten with v'. Otherwise, we need check the children and perform recursive insertion. This is realized in function $ins(C, k_1, K', v')$. It examines the (character, sub-tree) pairs in C one by one. Let C' be the rest of pairs except for the first one. This function can be defined as below.

$$ins(C, k_1, K', v') = \begin{cases} \{(k_1, insert((\phi, \phi), K', v'))\} & : & C = \phi \\ \{k_1, insert(T_1, K', v')\} \cup C' & : & k_1 = c_1 \\ \{(c_1, T_1)\} \cup ins(C', k_1, K', v') & : & otherwise \end{cases} \tag{5.9}$$

If C is empty, we create a pair, mapping from character k_1 to a new empty tree (it is not ϕ, but a node with empty value and empty sub-tree list), and recursively insert the rest characters. Otherwise, the algorithm locates the child which is mapped from k_1 for further insertion.

The following Haskell example program implements the insertion algorithm.

```
insert t []     x = Trie (Just x)  (children t)
insert t (k:ks) x = Trie (value t) (ins (children t) k ks x) where
    ins [] k ks x = [(k, (insert empty ks x))]
    ins (p:ps) k ks x = if fst p == k
                        then (k, insert (snd p) ks x):ps
                        else p:(ins ps k ks x)
```

To realize the insertion imperatively, starting from the root, we pick the character one by one from the string. For each character, we examine which child sub-tree represents that character. If the corresponding child is empty, a new node is created. After that, we pick the next character and repeat this process.

After consuming all the characters, we then store the value bound the key in the node we arrived.

```
1:  function INSERT(T, k, v)
2:      if T = NIL then
3:          T ← EMPTY-NODE
4:      p ← T
5:      for each c in k do
6:          if CHILDREN(p)[c] = NIL then
7:              CHILDREN(p)[c] ← EMPTY-NODE
8:          p ← CHILDREN(p)[c]
9:      DATA(p) ← v
10:     return T
```

The following example ANSI C program implements the insertion algorithm.

```
struct Trie* insert(struct Trie* t, const char* key, void* value) {
    int c;
    struct Trie *p;
    if(!t)
        t = create_node();
    for (p = t; *key; ++key, p = p->children[c]) {
        c = *key - 'a';
        if (!p->children[c])
            p->children[c] = create_node();
    }
    p->data = value;
    return t;
}
```

Where function `create_node` creates new empty node, with all children initialized to empty.

```
struct Trie* create_node() {
    struct Trie* t = (struct Trie*) malloc(sizeof(struct Trie));
    int i;
    for (i = 0; i < 26; ++i)
        t->children[i] = NULL;
    t->data = NULL;
    return t;
}
```

www.dbooks.org

5.4.3 Look up

When looking up a key, we start from the first character, if it is bound to some sub-tree, we then recursively search the rest characters in that child sub-tree. Denote the trie as $T = (v, C)$, the key being looked up as $K = k_1 k_2 ... k_n$ if it isn't empty. The first character in the key is k_1, and the rest characters are represented as K'.

$$lookup(T, K) = \begin{cases} v & : & K = \phi \\ \phi & : & find(C, k_1) = \phi \\ lookup(T', K') & : & find(C, k_1) = T' \end{cases} \tag{5.10}$$

Where function $find(C, k)$ examines the character-tree pairs one by one to check if any child sub-tree is bound to character k. If the list of pairs C is empty, then the subject key does not exist. Otherwise, let $C = \{(k_1, T_1), (k_2, T_2), ..., (k_m, T_m)\}$, the first sub-tree T_1 is bound to k_1, the rest of pairs are represented as C'. We repeatedly consumes each pair to located the sub-tree for further search. Below equation defines the $find$ function.

$$find(C, k) = \begin{cases} \phi & : & C = \phi \\ T_1 & : & k_1 = k \\ find(C', k) & : & otherwise \end{cases} \tag{5.11}$$

The following Haskell example program implements the trie looking up algorithm. It uses the `lookup` function provided in standard library.

```
find t [] = value t
find t (k:ks) = case lookup k (children t) of
                  Nothing → Nothing
                  Just t' → find t' ks
```

To realize the look up algorithm imperatively, we extract the character from the key one by one. For each character, we search among the sub-trees to see if there is a branch matches this character. If there is no such a child, the look up process terminates to indicate that the key does not exist. When we arrive at the last character of the key, the data stored in the current node is the result.

```
1: function LOOK-UP(T, key)
2:     if T = NIL then
3:         return not found
4:     for each c in key do
5:         if CHILDREN(T)[c] = NIL then
6:             return not found
7:         T ← CHILDREN(T)[c]
8:     return DATA(T)
```

Below ANSI C example program implements the look up algorithm. It returns NULL if the key does not exist.

```
void* lookup(struct Trie* t, const char* key) {
    while (*key && t && t→children[*key - 'a'])
        t = t→children[*key++ - 'a'];
    return (*key || !t) ? NULL : t→data;
}
```

Exercise 5.1

- Use a collection data structure to manage sub-trees in the imperative alphabetic trie. How does the collection impact the performance?

5.5 Alphabetic prefix tree

Similar to integer trie, alphabetic trie is not memory efficient. We can use the same approach to compress alphabetic trie to prefix tree.

5.5.1 Definition

Alphabetic prefix tree is a special prefix tree, each node contains multiple branches. All sub-trees share the longest common prefix string in a node. As the result, there is no node has only one child, because it conflicts with the longest common prefix property.

If we turn the trie shown in figure 5.9 into prefix tree by compressing all nodes which have only one child. we can get a prefix tree as in figure 5.10.

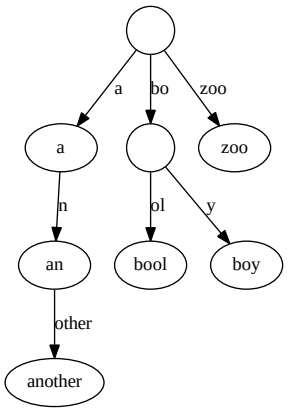


Figure 5.10: A prefix tree, with keys: 'a', 'an', 'another', 'bool', 'boy' and 'zoo'.

We can modify the alphabetic trie and adapt it to prefix tree. The tree is either empty, or a node in form $T = (v, C)$. Where v is the optional satellite data; $C = \{(s_1, T_1), (s_2, T_2), ..., (s_n, T_n)\}$ represents the sub-trees. It is a list of pairs. Each pair contains a string s_i, and a sub-tree T_i the string is bound to.

The following Haskell example code defines prefix tree accordingly.

```
data PrefixTree k v = PrefixTree { value :: Maybe v
                                 , children :: [([k], PrefixTree k v)]}

empty = PrefixTree Nothing []

leaf x = PrefixTree (Just x) []
```

Below Python example program reuses the trie definition to define prefix tree.

www.dbooks.org

```
class PrefixTree:
    def __init__(self, value = None):
        self.value = value
        self.subtrees = {}
```

5.5.2 Insertion

When insert a key s, if the prefix tree is empty, we create a leaf node as shown in figure 5.11 (a). Otherwise, we examine the sub-trees to see if there's some tree T_i bound to the string s_i, and there exists common prefix between s_i and s. In such case, we need branch out a new leaf T_j. To do this, we firstly create a new internal branch node, bind it with the common prefix; then set T_i and T_j as the two children sub-trees of this node. T_i and T_j share the common prefix. This is shown in figure 5.11 (b). There are two special cases. s can be the prefix of s_i as shown in figure 5.11 (c). Similarly, s_i can be the prefix of s as shown in figure 5.11 (d).

For prefix tree $T = (v, C)$, function $insert(T, k, v')$ inserts key k, and value v' to the tree.

$$insert(T, k, v') = (v, ins(C, k, v')) \tag{5.12}$$

This function calls another function $ins(C, k, v')$. If the children sub-trees C is empty, a new leaf is created; Otherwise we examine the sub-trees one by one. Denote $C = \{(k_1, T_1), (k_2, T_2), ..., (k_n, T_n)\}$, C' holds all the (prefix, sub-tree) pairs except for the first one. the ins function can be defined as the following.

$$ins(C, k, v') = \begin{cases} \{(k, (v', \phi))\} & : \quad C = \phi \\ \{(k, (v', C_{T_1}))\} \cup C' & : \quad k_1 = k \\ \{branch(k, v', k_1, T_1)\} \cup C' & : \quad match(k_1, k) \\ \{(k_1, T_1)\} \cup ins(C', k, v') & : \quad otherwise \end{cases} \tag{5.13}$$

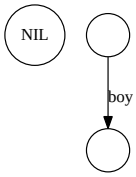
The first clause deals with the edge case of empty children. A leaf node bound to k, containing v' is returned as the only sub-tree. The second clause overwrites the previous value with v' if there is some child bound to the same key. C_{T_1} represents the children of sub-tree T_1. The third clause branches out a new leaf if the first child matches the key k. The last clause goes on checking the rest sub-trees.

We define two keys A and B matching if they have non-empty common prefix.

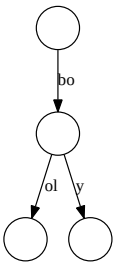
$$match(A, B) = A \neq \phi \wedge B \neq \phi \wedge a_1 = b_1 \tag{5.14}$$

Where a_1 and b_1 are the first characters in A and B if they are not empty.

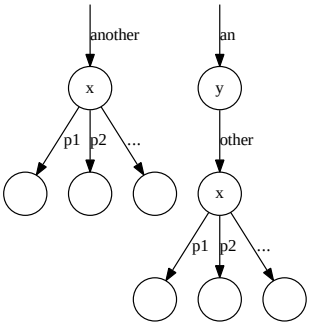
Function $branch(k_1, v, k_2, T_2)$ takes two keys, a value and a tree. It extracts the longest common prefix $k = lcp(k_1, k_2)$, and assigns the different part to $k_1' = k_1 - k$, $k_2' = k_2 - k$. The algorithm firstly handles the edge cases that either k_1 is the prefix of k_2 or k_2 is the prefix of k_1. For the former one, it creates a new node containing v, binds this node to k, and set (k_2', T_2) as the only child sub-tree; For the later one, it recursively inserts k_1' and v to T_2. Otherwise, the algorithm creates a branch node, binds it to the longest common prefix k, and



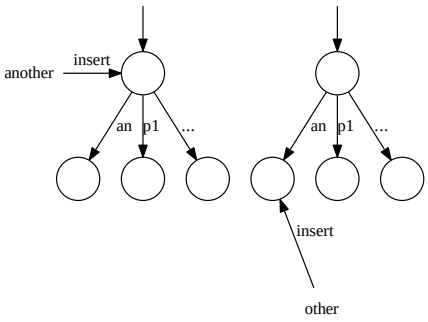
(a) Insert key 'boy' into the empty prefix tree, the result is a leaf.



(b) Insert key 'bool'. A new branch with common prefix 'bo' is created.



(c) Insert key 'an' with value y into x with prefix 'another'.



(d) Insert 'another', into the node with prefix 'an'. We recursively insert key 'other' to the child.

Figure 5.11: Prefix tree insertion

www.dbooks.org

set the two children sub-trees for it. One sub-tree is (k'_2, T_2), the other is a leaf node containing v, and being bound to k'_1.

$$branch(k_1, v, k_2, T_2) = \begin{cases} (k, (v, \{(k'_2, T_2)\})) & : \quad k = k_1 \\ (k, insert(T_2, k'_1, v)) & : \quad k = k_2 \\ (k, (\phi, \{(k'_1, (v, \phi)), (k'_2, T_2)\}) & : \quad otherwise \end{cases} \tag{5.15}$$

Where

$$\begin{aligned} k &= lcp(k_1, k_2) \\ k'_1 &= k_1 - k \\ k'_2 &= k_1 - k \end{aligned}$$

Function $lcp(A, B)$ keeps taking the same characters from A and B one by one. Denote a_1 and b_1 as the first characters in A and B if they are not empty. A' and B' are the rest characters.

$$lcp(A, B) = \begin{cases} \phi & : \quad A = \phi \vee B = \phi \vee a_1 \neq b_1 \\ \{a_1\} \cup lcp(A', B') & : \quad a_1 = b_1 \end{cases} \tag{5.16}$$

The following Haskell example program implements the prefix tree insertion algorithm.

```
import Data.List (isPrefixOf)

insert :: Eq k ⇒ PrefixTree k v → [k] → v → PrefixTree k v
insert t ks x = PrefixTree (value t) (ins (children t) ks x) where
    ins []     ks x = [(ks, leaf x)]
    ins (p@(ks', t') : ps) ks x
        | ks' == ks
          = (ks, PrefixTree (Just x) (children t')) : ps  -- overwrite
        | match ks' ks
          = (branch ks x ks' t') : ps
        | otherwise
          = p : (ins ps ks x)

match x y = x /= [] && y /= [] && head x == head y

branch :: Eq k ⇒ [k] → v → [k] → PrefixTree k v → ([k], PrefixTree k v)
branch ks1 x ks2 t2
    | ks1 == ks
      -- ex: insert "an" into "another"
      = (ks, PrefixTree (Just x) [(ks2', t2)])
    | ks2 == ks
      -- ex: insert "another" into "an"
      = (ks, insert t2 ks1' x)
    | otherwise = (ks, PrefixTree Nothing [(ks1', leaf x), (ks2', t2)])
   where
     ks = lcp ks1 ks2
     m = length ks
     ks1' = drop m ks1
     ks2' = drop m ks2
```

```
lcp :: Eq k ⇒ [k] → [k] → [k]
lcp [] _ = []
lcp _ [] = []
lcp (x:xs) (y:ys) = if x==y then x : (lcp xs ys) else []
```

The insertion algorithm can be realized imperative as below.

```
1:  function INSERT(T, k, v)
2:      if T = NIL then
3:          T ← EMPTY-NODE
4:      p ← T
5:      loop
6:          match ← FALSE
7:          for each (s_i, T_i) ∈ CHILDREN(p) do
8:              if k = s_i then
9:                  VALUE(p) ← v
10:                 return T
11:             c ← LCP(k, s_i)
12:             k_1 ← k − c
13:             k_2 ← s_i − c
14:             if c ≠ NIL then
15:                 match ← TRUE
16:                 if k_2 = NIL then                    ▷ s_i is prefix of k
17:                     p ← T_i
18:                     k ← k_1
19:                     break
20:                 else                                 ▷ Branch out a new leaf
21:                     ADD(CHILDREN(p), (c, BRANCH(k_1, LEAF(v), k_2, T_i)))
22:                     DELETE(CHILDREN(p), (s_i, T_i))
23:                     return T
24:         if ¬match then                               ▷ Add a new leaf
25:             ADD(CHILDREN(p), (k, LEAF(v)))
26:             break
27:     return T
```

In this algorithm, function LCP finds the longest common prefix of the two strings. For example, string 'bool' and 'boy' have the longest common prefix 'bo'. The subtraction symbol '-' for strings gives the different part of two strings. For example 'bool' - 'bo' = 'ol'. Function BRANCH creates a branch node and updates keys.

The longest common prefix can be extracted character by character from two strings till there is unmatch.

```
1:  function LCP(A, B)
2:      i ← 1
3:      while i ≤ |A| ∧ i ≤ |B| ∧ A[i] = B[i] do
4:          i ← i + 1
5:      return A[1...i − 1]
```

There are two cases when branch out a new leaf. BRANCH(s_1, T_1, s_2, T_2) takes two different keys and trees. If s_1 is empty, we are dealing with the case such as insert key 'an' into a child bound to string 'another'. We set T_2 as the child sub-tree of T_1. Otherwise, we create a new branch node and set T_1 and

www.dbooks.org

T_2 as the two children.

```
1: function BRANCH(s_1, T_1, s_2, T_2)
2:     if s_1 = φ then
3:         ADD(CHILDREN(T_1), (s_2, T_2))
4:         return T_1
5:     T ← EMPTY-NODE
6:     CHILDREN(T) ← {(s_1, T_1), (s_2, T_2)}
7:     return T
```

The following example Python program implements the prefix tree insertion algorithm.

```
def insert(t, key, value):
    if t is None:
        t = PrefixTree()
    node = t
    while True:
        match = False
        for k, tr in node.subtrees.items():
            if key == k: # overwrite
                node.value = value
                return t
            prefix, k1, k2 = lcp(key, k)
            if prefix != "":
                match = True
                if k2 == "":
                    # e.g.: insert "another" into "an", go on traversing
                    node = tr
                    key = k1
                    break
                else: #branch out a new leaf
                    node.subtrees[prefix] = branch(k1, PrefixTree(value), k2, tr)
                    del node.subtrees[k]
                    return t
        if not match: # add a new leaf
            node.subtrees[key] = PrefixTree(value)
            break
    return t
```

Where the `lcp` and `branch` functions are implemented as below.

```
def lcp(s1, s2):
    j = 0
    while j < len(s1) and j < len(s2) and s1[j] == s2[j]:
        j += 1
    return (s1[0:j], s1[j:], s2[j:])

def branch(key1, tree1, key2, tree2):
    if key1 == "":
        #example: insert "an" into "another"
        tree1.subtrees[key2] = tree2
        return tree1
    t = PrefixTree()
    t.subtrees[key1] = tree1
    t.subtrees[key2] = tree2
```

```
    return t
```

5.5.3 Look up

When look up a key, we can't examine the characters one by one as in trie any more. Start from the root, we need search among the children sub-trees to see if any one is bound to some prefix of the key. If there is such a sub-tree, we remove the prefix from the key, and recursively look up the updated key in this child sub-tree. The look up fails if there's no sub-tree bound to any prefix of the key.

For prefix tree $T = (v, C)$, we search among its children sub-tree C.

$$lookup(T, k) = find(C, k) \tag{5.17}$$

If C is empty, the lookup fails; Otherwise, For $C = \{(k_1, T_1), (k_2, T_2), ..., (k_n, T_n)\}$, we firstly examine if k is the prefix of k_1, then recursively check the rest pairs denoted as C'.

$$find(C, k) = \begin{cases} \phi & : & C = \phi \\ v_{T_1} & : & k = k_1 \\ lookup(T_1, k - k_1) & : & k_1 \sqsubset k \\ find(C', k) & : & otherwise \end{cases} \tag{5.18}$$

Where $A \sqsubset B$ means string A is prefix of B. $find$ mutually calls $lookup$ if a child is bound to some prefix of the key.

Below Haskell example program implements the looking up algorithm.

```
find :: Eq k ⇒ PrefixTree k v → [k] → Maybe v
find t = find' (children t) where
    find' [] _ = Nothing
    find' (p@(ks', t') : ps) ks
          | ks' == ks = value t'
          | ks' `isPrefixOf` ks = find t' (diff ks ks')
          | otherwise = find' ps ks
    diff ks1 ks2 = drop (length (lcp ks1 ks2)) ks1
```

The look up algorithm can also be realized imperatively.

```
1:  function LOOK-UP(T, k)
2:      if T = NIL then
3:          return not found
4:      repeat
5:          match ← FALSE
6:          for ∀(k_i, T_i) ∈ CHILDREN(T) do
7:              if k = k_i then
8:                  return DATA(T_i)
9:              if k_i is prefix of k then
10:                 match ← TRUE
11:                 k ← k − k_i
12:                 T ← T_i
13:                 break
14:     until ¬match
15:     return not found
```

www.dbooks.org

Below Python example program implements the looking up algorithm. It reuses the `lcp(s1, s2)` function defined previously to test if a string is the prefix of the other.

```
def lookup(t, key):
    if t is None:
        return None
    while True:
        match = False
        for k, tr in t.subtrees.items():
            if k == key:
                return tr.value
            prefix, k1, k2 = lcp(key, k)
            if prefix != "" and k2 == "":
                match = True
                key = k1
                t = tr
                break
        if not match:
            break
    return None
```

5.6 Applications of trie and prefix tree

Trie and prefix tree can be used to solve many interesting problems. Integer based prefix tree is used in compiler implementation. Some daily used software applications have many interesting features which can be realized with trie or prefix tree. In this section, we give some examples, including, e-dictionary, word auto-completion, T9 input method etc. Different from the commerial implementation, the solutions we demonstrated here are for illustration purpose only.

5.6.1 E-dictionary and word auto-completion

Figure 5.12 shows a screen shot of an E-dictionary. When user enters characters, the dictionary searches its word library, then lists the candidate words and phrases starts from what the user input.

A E-dictionary typically contains hundreds of thousands words. It's very expensive to perform a complete search. Commercial software adopts complex approaches, including caching, indexing etc to speed up this process.

Similar with e-dictionary, figure 5.13 shows a popular Internet search engine. When user input something, it provides a candidate lists, with all items starting with what the user has entered[1]. And these candidates are shown in the order of popularity. The more people search, the upper position it is in the list.

In both cases, the software provides a kind of word auto-completion mechanism. Some editors can also help programmers to auto-complete the code.

Let's see how to implement the e-dictionary with prefix tree. To simplify the problem, we assume the dictionary only supports English - English information.

[1]It's more complex than just matching the prefix. Including the spell checking and auto currection, key words extraction and recommendation etc.

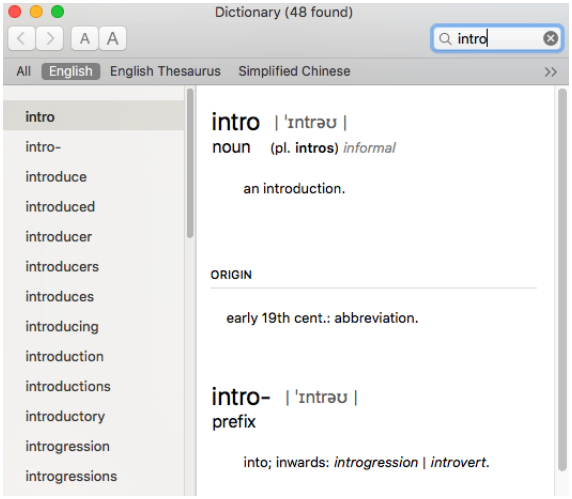


Figure 5.12: E-dictionary. All candidates starting with what the user input are listed.



Figure 5.13: A search engine. All candidates starting with what user input are listed.

www.dbooks.org

A dictionary stores key-value pairs, the key is English word or phrase, the value is the meaning described in text.

We can store all the words and their meanings in a trie, but it consumes too large space especially when there are huge amount of items. We'll use prefix tree to realize the e-dictionary.

When user wants to look up word 'a', the dictionary does not only return the meaning of 'a', but also provides a list of candidates starting with 'a', including 'abandon', 'about', 'accent', 'adam', ... Of course all these words are stored in the prefix tree.

If there are too many candidates, we can limit only displaying the top 10 candidates, and allow the user to browse more.

To define this algorithm, if the string we are looking for is empty, we expand all children sub-trees until getting n candidates. Otherwise we recursively examine the children to find one which has prefix equal to this string.

In programming environments supporting lazy evaluation. An intuitive solution is to lazily expand all candidates, and take the first n on demand. Denote the prefix tree in form $T = (v, C)$, below function enumerates all items starts with key k.

$$findAll(T, k) = \begin{cases} enum(C) & : \quad k = \phi, v = \phi \\ \{(\phi, v)\} \cup enum(C) & : \quad k = \phi, v \neq \phi \\ find(C, k) & : \quad k \neq \phi \end{cases} \tag{5.19}$$

The first two clauses deal with the edge cases that the key is empty. All the children sub-trees are enumerated except for those with empty values. The last clause finds child sub-tree matches k.

For non-empty children sub-trees, $C = \{(k_1, T_1), (k_2, T_2), ..., (k_m, T_m)\}$, denote the rest pairs except for the first one as C'. The enumeration algorithm can be defined as below.

$$enum(C) = \begin{cases} \phi & : \quad C = \phi \\ mapAppend(k_1, findAll(T_1, \phi)) \cup enum(C') & : \end{cases} \tag{5.20}$$

Where $mapAppend(k, L) = \{(k + k_i, v_i) | (k_i, v_i) \in L\}$. It concatenate the prefix k in front of every key-value pair in list L[2].

Function $enum$ can also be defined with concept of $concatMap$ (also called $flatMap$)[3].

$$enum(C) = concatMap(\lambda_{(k,T)}.mapAppend(k, findAll(T, \phi))) \tag{5.21}$$

Function $find(C, k)$ is defined as the following. For empty children, the result is empty as well; Otherwise, it examines the first child sub-tree T_1 which is bound to string k_1. If k equals to k_1 or is a prefix of k_1, it calls $mapAppend$ to concatenate the prefix k_1 in front of the key of every child sub-tree under

[2]The concept here is to map on the first thing. In some environment, like Haskell, $mapAppend$ can be expressed as $map(first(k+), L)$ by using the arrow in category theory.

[3]Literally, it results like first map on each element, then concatenate the result together. It's typically realized with 'build-foldr' to eliminate the intermediate list.

T_1; If k_1 is prefix of k, the algorithm recursively find all children sub-trees start with $k - k_1$; otherwise, the algorithm by-passes the first child sub-tree and goes on finding the rest sub-trees.

$$find(C, k) = \begin{cases} \phi & : & C = \phi \\ mapAppend(k_1, findAll(T_1, \phi)) & : & k \sqsubset k_1 \\ mapAppend(k_1, findAll(T_1, k - k_1)) & : & k_1 \sqsubset k \\ find(C', k) & : & otherwise \end{cases} \tag{5.22}$$

Below example Haskell program implements the e-dictionary application according to the above equations.

```
import Control.Arrow (first)

get n t k = take n $ findAll t k

findAll :: Eq k ⇒ PrefixTree k v → [k] → [([k], v)]
findAll (PrefixTree Nothing cs) [] = enum cs
findAll (PrefixTree (Just x) cs) [] = ([], x) : enum cs
findAll (PrefixTree _ cs) k = find' cs k
  where
    find' [] _ = []
    find' ((k', t') : ps) k
          | k `isPrefixOf` k'
              = map (first (k' ++)) (findAll t' [])
          | k' `isPrefixOf` k
              = map (first (k' ++)) (findAll t' $ drop (length k') k)
          | otherwise = find' ps k

enum :: Eq k ⇒ [([k], PrefixTree k v)] → [([k], v)]
enum = concatMap (λ(k, t) → map (first (k ++)) (findAll t []))
```

In the lazy evaluation environment, the top n candidates can be gotten like $take(n, findAll(T, k))$. Appendix A has detailed definition of $take$ function.

We can also realize this algorithm impertiavely. The following algorithm reuses the looking up defined for prefix tree. When finds a node bound the prefix of what we are looking for, it expands all its children sub-trees till getting n candidates.

```
1: function LOOK-UP(T, k, n)
2:     if T = NIL then
3:         return φ
4:     prefix ← NIL
5:     repeat
6:         match ← FALSE
7:         for ∀(k_i, T_i) ∈ CHILDREN(T) do
8:             if k is prefix of k_i then
9:                 return EXPAND(prefix + k_i, T_i, n)
10:            if k_i is prefix of k then
11:                match ← TRUE
12:                k ← k − k_i
13:                T ← T_i
14:                prefix ← prefix + k_i
```

www.dbooks.org

```
15:                 break
16:     until ¬match
17:     return φ
```

Where function EXPAND$(T, prefix, n)$ picks n sub-trees. They share the same prefix in T. It is realized as BFS (Bread-First-Search) traverse. 14.3.1 in the Chapter of search explains BFS in detail.

```
1: function EXPAND(prefix, T, n)
2:     R ← φ
3:     Q ← {(prefix, T)}
4:     while |R| < n ∧ Q is not empty do
5:         (k, T) ← POP(Q)
6:         if DATA(T) ≠ NIL then
7:             R ← R ∪ {(k, DATA(T) )}
8:         for ∀(k_i, T_i) ∈ CHILDREN(T) in sorted order do
9:             PUSH(Q, (k + k_i, T_i))
```

The following example Python program implements the e-dictionary application. When testing if a string is prefix of another one, it uses the `find` function provided in standard string library.

```
def lookup(t, key, n):
    if t is None:
        return []
    prefix = ""
    while True:
        match = False
        for k, tr in t.subtrees.items():
            if string.find(k, key) == 0: # key is prefix of k
                return expand(prefix + k, tr, n)
            if string.find(key, k) ==0:
                match = True
                key = key[len(k):]
                t = tr
                prefix += k
                break
        if not match:
            break
    return []

def expand(prefix, t, n):
    res = []
    q = [(prefix, t)]
    while len(res)<n and q:
        (s, p) = q.pop(0)
        if p.value is not None:
            res.append((s, p.value))
        for k, tr in sorted(p.subtrees.items()):
            q.append((s + k, tr))
    return res
```

5.6.2 T9 input method

When people edit text in the mobile phone, the experience is quite different. This is because the so called ITU-T key pad has much fewer keys than PC as shown in figure 5.14.



Figure 5.14: The ITU-T keypad for mobile phone.

There are typical two methods to input word or phrases with ITU-T key pad. If user wants to enter a word 'home' for example, he can press the keys in below sequence.

- Press key '4' twice to enter the letter 'h';
- Press key '6' three times to enter the letter 'o';
- Press key '6' to enter the letter 'm';
- Press key '3' twice to enter the letter 'e';

Another much quicker way is to just press the following keys.

- Press key '4', '6', '6', '3', word 'home' appears on top of the candidate list;
- Press key '*' to change a candidate word, so word 'good' appears;
- Press key '*' again to change another candidate word, next word 'gone' appears;
- ...

Compare the two methods, the second one is much easier for the user. The only overhead is the need to store a dictionary of candidate words.

The second method is known as 'T9' input method, or predictive input method [6], [7]. The abbreviation 'T9' stands for 'textonym'. It start with 'T' with 9 characters. T9 input can also be realized with prefix tree.

In order to provide candidate words, a dictionary must be prepared in advance. Prefix tree can be used to store the dictionary. The commercial T9 implementations typically use multiple layers indexed dictionary in both file system and cache. The realization shown here is for illustration purpose only.

www.dbooks.org

Firstly, we need define the T9 mapping, which maps from digit to candidate characters.

$$M_{T9} = \{ \begin{array}{l} 2 \to abc, 3 \to def, 4 \to ghi, \\ 5 \to jkl, 6 \to mno, 7 \to pqrs, \\ 8 \to tuv, 9 \to wxyz \} \end{array} \tag{5.23}$$

With this mapping, $M_{T9}[i]$ returns the corresponding characters for digit i. We can also define the reversed mapping from a character back to digit.

$$M_{T9}^{-1} = concat(\{\{c \to d | c \in S\} | (d \to S) \in M_{T9}\}) \tag{5.24}$$

Given a sequence of characters, we can convert it to a sequence of digits by looking up M_{T9}^{-1}.

$$digits(S) = \{M_{T9}^{-1}[c] | c \in S\} \tag{5.25}$$

When input digits $D = d_1 d_2 ... d_n$, we define the T9 lookup algorithm as below.

$$findT9(T, D) = \begin{cases} \{\phi\} & : \quad D = \phi \\ concatMap(find, prefixes(T)) & : \quad otherwise \end{cases} \tag{5.26}$$

Where T is the prefix tree built from a set of words and phrases. It's kind of a dictionary we'll look up. If the input D is empty, the result is an empty string. Otherwise, it looks up the sub-trees that match the input, and concat the result together.

To enumerate the matched sub-trees, we examine all the children sub-trees C_T, for every pair (k_i, T_i). We first convert string k_i to digit sequence d_i, then compare d_i and D. If either one is the prefix of the other, then this pair is selected as a candidate for further search.

$$prefixes(T) = \{(k_i, T_i) | (k_i, T_i) \in C_T, d_i = digits(k_i), d_i \sqsubset D \vee D \sqsubset d_i\} \tag{5.27}$$

Function $find$ takes a passed in prefix S, and a sub-tree T' to look up further. As S is prefix of D, it removes it from D to get a new input to $D' = D - S$ to search, then later insert S back in front of every recursive search result.

$$find(S, T') = \{take(n, S + s_i) | s_i \in findT9(T', D - S)\} \tag{5.28}$$

Where $n = |D|$ is the length of the input digits. Function $take(n, L)$ takes the first n elements from the list L. If the length of the list is less then n, or the elements are taken.

The following Haskell example program implements the T9 look up algorithm with prefix tree.

```
import qualified Data.Map as Map

mapT9 = Map.fromList [('1', ",."), ('2', "abc"), ('3', "def"), ('4', "ghi"),
                      ('5', "jkl"), ('6', "mno"), ('7', "pqrs"), ('8', "tuv"),
                      ('9', "wxyz")]
```

```
rmapT9 = Map.fromList $ concatMap (λ(d, s) → [(c, d) | c ← s]) $ Map.toList mapT9

digits = map (λc → Map.findWithDefault '#' c rmapT9)

findT9 :: PrefixTree Char v → String → [String]
findT9 t [] = [""]
findT9 t k = concatMap find prefixes
  where
    n = length k
    find (s, t') = map (take n ∘ (s++)) $ findT9 t' (k `diff` s)
    diff x y = drop (length y) x
    prefixes = [(s, t') | (s, t') ← children t, let ds = digits s in
                          ds `isPrefixOf` k || k `isPrefixOf` ds]
```

To realize this algorithm imperatively, we can perform BFS search with a queue Q. The queue stores tuples $(prefix, D, T)$. Every tuple records the possible prefix string we've searched so far; the rest of the digits to be searched; and the sub-tree we are going to search. The queue is initialized with the empty prefix, the whole digit sequence, and the prefix tree root. The algorithm keeps picking the tuple from the queue until it's empty. For every tuple popped from the queue, we extract the tree from the tuple, then examine the children sub-trees of it. for each sub-tree T_i, we convert the corresponding prefix string k_i to digits D' by looking up the reversed T9 map. If the D is prefix of D', it's a valid candidate. We concatenate k_i after the prefix in the tuple, and record this string in the result. If D' is prefix of D, we need furthur search this sub-tree. To do this, we create a new tuple consist of the new prefix ends with k_i, the rest of the digits $D - D'$, and the sub-tree. Then push this tuple back to the queue.

```
1:  function LOOK-UP-T9(T, D)
2:      R ← φ
3:      if T = NIL or D = φ then
4:          return R
5:      n ← |D|
6:      Q ← {(φ, D, T)}
7:      while Q ≠ φ do
8:          (prefix, D, T) ← POP(Q)
9:          for ∀(k_i, T_i) ∈ CHILDREN(T) do
10:             D' ← DIGITS(k_i)
11:             if D' ⊏ D then                          ▷ D' is prefix of D
12:                 R ← R ∪ { TAKE (n, prefix + k_i)}    ▷ limit the length to n
13:             else if D ⊏ D' then
14:                 PUSH(Q, (prefix + k_i, D − D', T_i))
15:     return R
```

Function DIGITS(S) converts string S to sequence of digits.

```
1:  function DIGITS(S)
2:      D ← φ
3:      for each c ∈ S do
4:          D ← D ∪ {M_T9^{-1}[c]}
5:      return D
```

The following example Python program implements the T9 input method with prefix tree.

www.dbooks.org

```
T9MAP={'2':"abc", '3':"def", '4':"ghi", '5':"jkl", λ
       '6':"mno", '7':"pqrs", '8':"tuv", '9':"wxyz"}

T9RMAP = dict([(c, d) for d, cs in T9MAP.items() for c in cs])

def digits(w):
    return ''.join([T9RMAP[c] for c in w])

def lookup_t9(t, key):
    if t is None or key == "":
        return []
    res = []
    n = len(key)
    q = [("", key, t)]
    while q:
        prefix, key, t = q.pop(0)
        for k, tr in t.subtrees.items():
            ds = digits(k)
            if string.find(ds, key) == 0: # key is prefix of ds
                res.append((prefix + k)[:n])
            elif string.find(key, ds) == 0: # ds is prefix of key
                q.append((prefix + k, key[len(k):], tr))
    return res
```

Exercise 5.2

- Realize the e-dictionary and T9 lookup with trie.
- For the alphabetic prefix tree look up algorithms that return multiple results, how to ensure the result is in lexicographic order? What is the performance?
- How to realize the e-dictionary and T9 look up without lazy evaluation?

5.7 Summary

In this chapter, we start from the integer based trie and prefix tree. The map data structure based on integer tree plays the important role in Compiler implementation. Alphabetic trie and prefix tree are natural extensions. They can manipulate text information. We demonstrate how to realize the predictive e-dictionary and T9 input method with prefix tree, although these examples are different from the commercial implementations. Other data structure, suffix tree, has close relationship with trie and prefix tree. Suffix tree is introduced in Appendix D.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. "Introduction to Algorithms, Second Edition". Problem 12-1. ISBN:0262032937. The MIT Press. 2001

[2] Chris Okasaki and Andrew Gill. "Fast Mergeable Integer Maps". Workshop on ML, September 1998, pages 77-86, http://www.cse.ogi.edu/~andy/pub/finite.htm

[3] D.R. Morrison, "PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric", Journal of the ACM, 15(4), October 1968, pages 514-534.

[4] Suffix Tree, Wikipedia. http://en.wikipedia.org/wiki/Suffix_tree

[5] Trie, Wikipedia. http://en.wikipedia.org/wiki/Trie

[6] T9 (predictive text), Wikipedia. http://en.wikipedia.org/wiki/T9_(predictive_text)

[7] Predictive text, Wikipedia. http://en.wikipedia.org/wiki/Predictive_text

www.dbooks.org

Chapter 6

B-Trees

6.1 Introduction

B-Tree is important data structure. It is widely used in modern file systems. Some are implemented based on B+ tree, which is extended from B-tree. B-tree is also widely used in database systems.

Some textbooks introduce B-tree with the the problem of how to access a large block of data on magnetic disks or secondary storage devices[2]. It is also helpful to understand B-tree as a generalization of balanced binary search tree[2].

Refer to the Figure 6.1, It is easy to find the difference and similarity of B-tree regarding to binary search tree.

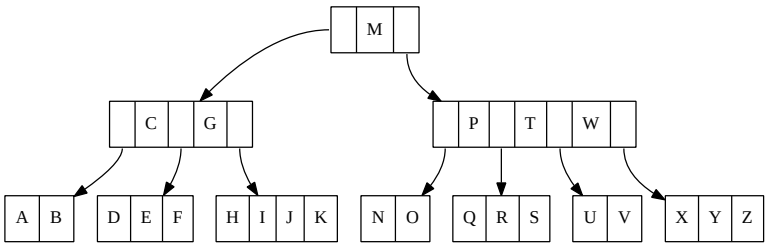


Figure 6.1: Example B-Tree

Remind the definition of binary search tree. A binary search tree is

- either an empty node;
- or a node contains 3 parts, a value, a left child and a right child. Both children are also binary search trees.

The binary search tree satisfies the constraint that.

- all the values on the left child are not greater than the value of of this node;
- the value of this node is not greater than any values on the right child.

www.dbooks.org

For non-empty binary tree (L, k, R), where L, R and k are the left, right children, and the key. Function $Key(T)$ accesses the key of tree T. The constraint can be represented as the following.

$$\forall x \in L, \forall y \in R \Rightarrow Key(x) \leq k \leq Key(y) \tag{6.1}$$

If we extend this definition to allow multiple keys and children, we get the B-tree definition.

A B-tree

- is either empty;
- or contains n keys, and $n + 1$ children, each child is also a B-Tree, we denote these keys and children as $k_1, k_2, ..., k_n$ and $c_1, c_2, ..., c_n, c_{n+1}$.

Figure 6.2 illustrates a B-Tree node.

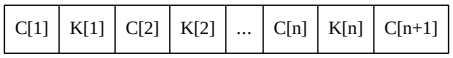


Figure 6.2: A B-Tree node

The keys and children in a node satisfy the following order constraints.

- Keys are stored in non-decreasing order. that $k_1 \leq k_2 \leq ... \leq k_n$;
- for each k_i, all elements stored in child c_i are not greater than k_i, while k_i is not greater than any values stored in child c_{i+1}.

The constraints can be represented as in equation (6.2) as well.

$$\forall x_i \in c_i, i = 0, 1, ..., n, \Rightarrow x_1 \leq k_1 \leq x_2 \leq k_2 \leq ... \leq x_n \leq k_n \leq x_{n+1} \tag{6.2}$$

Finally, after adding some constraints to make the tree balanced, we get the complete B-tree definition.

- All leaves have the same depth;
- We define integral number, t, as the *minimum degree* of B-tree;
  - each node can have at most $2t - 1$ keys;
  - each node can have at least $t - 1$ keys, except the root;

Consider a B-tree holds n keys. The minimum degree $t \geq 2$. The height is h. All the nodes have at least $t - 1$ keys except the root. The root contains at least 1 key. There are at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, ..., finally, there are at least $2t^{h-1}$ nodes at depth h. Times all nodes with $t - 1$ except for root, the total number of keys satisfies the following inequality.

$$\begin{aligned} n &\geq 1 + (t-1)(2 + 2t + 2t^2 + ... + 2t^{h-1}) \\ &= 1 + 2(t-1)\sum_{k=0}^{h-1} t^k \\ &= 1 + 2(t-1)\frac{t^h - 1}{t-1} \\ &= 2t^h - 1 \end{aligned} \tag{6.3}$$

Thus we have the inequality between the height and the number of keys.

$$h \leq \log_t \frac{n+1}{2} \tag{6.4}$$

This is the reason why B-tree is balanced. The simplest B-tree is so called 2-3-4 tree, where $t = 2$, that every node except root contains 2 or 3 or 4 keys. red-black tree can be mapped to 2-3-4 tree essentially.

The following Python code shows example B-tree definition. It explicitly pass t when create a node.

```
class BTree:
    def __init__(self, t):
        self.t = t
        self.keys = []
        self.children = []
```

B-tree nodes commonly have satellite data as well. We ignore satellite data for illustration purpose.

In this chapter, we will firstly introduce how to generate B-tree by insertion. Two different methods will be explained. One is the classic method as in [2], that we split the node before insertion if it's full; the other is the modify-fix approach which is quite similar to the red-black tree solution [3] [2]. We will next explain how to delete key from B-tree and how to look up a key.

6.2 Insertion

B-tree can be created by inserting keys repeatedly. The basic idea is similar to the binary search tree. When insert key x, from the tree root, we examine all the keys in the node to find a position where all the keys on the left are less than x, while all the keys on the right are greater than x.[1] If the current node is a leaf node, and it is not full (there are less then $2t - 1$ keys in this node), x will be insert at this position. Otherwise, the position points to a child node. We need recursively insert x to it.

Figure 6.3 shows one example. The B-tree illustrated is 2-3-4 tree. When insert key $x = 22$, because it's greater than the root, the right child contains key 26, 38, 45 is examined next; Since $22 < 26$, the first child contains key 21 and 25 are examined. This is a leaf node, and it is not full, key 22 is inserted to this node.

However, if there are $2t - 1$ keys in the leaf, the new key x can't be inserted, because this node is 'full'. When try to insert key 18 to the above example B-tree will meet this problem. There are 2 methods to solve it.

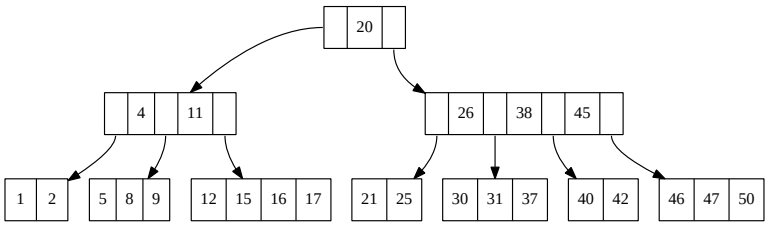
6.2.1 Splitting

Split before insertion

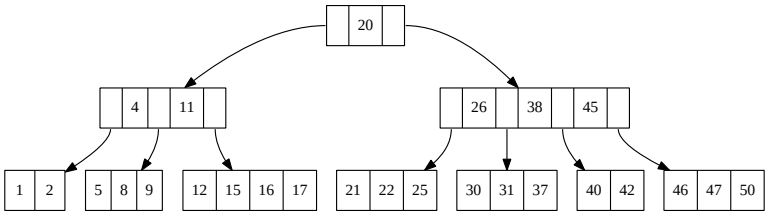
If the node is full, one method to solve the problem is to split to node before insertion.

[1] This is a strong constraint. In fact, only less-than and equality testing is necessary. The later exercise address this point.

www.dbooks.org



(a) Insert key 22 to the 2-3-4 tree. $22 > 20$, go to the right child; $22 < 26$ go to the first child.



(b) $21 < 22 < 25$, and the leaf isn't full.

Figure 6.3: Insertion is similar to binary search tree.

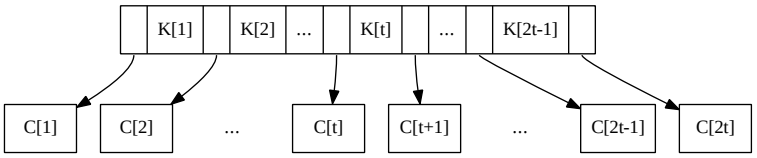
For a node with $t-1$ keys, it can be divided into 3 parts as shown in Figure 6.4. the left part contains the first $t-1$ keys and t children. The right part contains the rest $t-1$ keys and t children. Both left part and right part are valid B-tree nodes. the middle part is the t-th key. We can push it up to the parent node (if the current node is root, then the this key, with the two children will be the new root).

For node x, denote $K(x)$ as keys, $C(x)$ as children. The i-th key as $k_i(x)$, the j-th child as $c_j(x)$. Below algorithm describes how to split the i-th child for a given node.

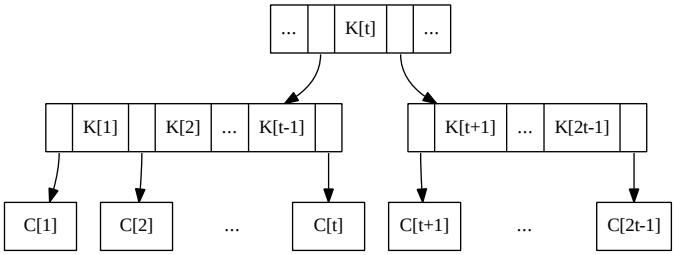
1: **procedure** SPLIT-CHILD($node, i$)
2: $x \leftarrow c_i(node)$
3: $y \leftarrow$ CREATE-NODE
4: INSERT($K(node), i, k_t(x)$)
5: INSERT($C(node), i+1, y$)
6: $K(y) \leftarrow \{k_{t+1}(x), k_{t+2}(x), ..., k_{2t-1}(x)\}$
7: $K(x) \leftarrow \{k_1(x), k_2(x), ..., k_{t-1}(x)\}$
8: **if** y is not leaf **then**
9: $C(y) \leftarrow \{c_{t+1}(x), c_{t+2}(x), ..., c_{2t}(x)\}$
10: $C(x) \leftarrow \{c_1(x), c_2(x), ..., c_t(x)\}$

The following example Python program implements this child splitting algorithm.

```
def split_child(node, i):
    t = node.t
    x = node.children[i]
    y = BTree(t)
    node.keys.insert(i, x.keys[t-1])
    node.children.insert(i+1, y)
    y.keys = x.keys[t:]
```



(a) Before split



(b) After split

Figure 6.4: Split node

```
    x.keys = x.keys[:t-1]
    if not is_leaf(x):
        y.children = x.children[t:]
        x.children = x.children[:t]
```

Where function `is_leaf` test if a node is leaf.

```
def is_leaf(t):
    return t.children == []
```

After splitting, a key is pushed up to its parent node. It is quite possible that the parent node has already been full. And this pushing violates the B-tree property.

In order to solve this problem, we can check from the root along the path of insertion traversing till the leaf. If there is any node in this path is full, the splitting is applied. Since the parent of this node has been examined, it is ensured that there are less than $2t - 1$ keys in the parent. It won't make the parent full if pushing up one key. This approach only need one single pass down the tree without any back-tracking.

If the root need splitting, a new node is created as the new root. There is no keys in this new created root, and the previous root is set as the only child. After that, splitting is performed top-down. And we can insert the new key finally.

1: **function** INSERT(T, k)
2: $r \leftarrow T$
3: **if** r is full **then** ▷ root is full
4: $s \leftarrow$ CREATE-NODE
5: $C(s) \leftarrow \{r\}$
6: SPLIT-CHILD($s, 1$)
7: $r \leftarrow s$
8: **return** INSERT-NONFULL(r, k)

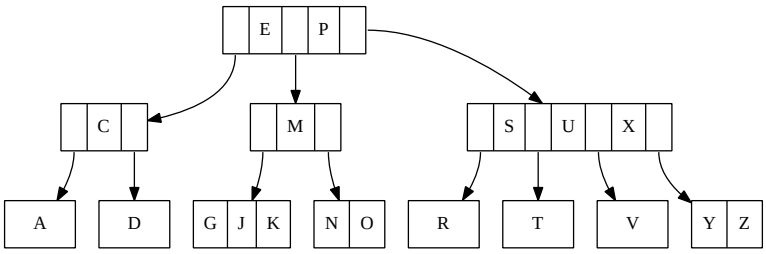
www.dbooks.org

Where algorithm INSERT-NONFULL assumes the node passed in is not full. If it is a leaf node, the new key is inserted to the proper position based on the order; Otherwise, the algorithm finds a proper child node to which the new key will be inserted. If this child is full, splitting will be performed.

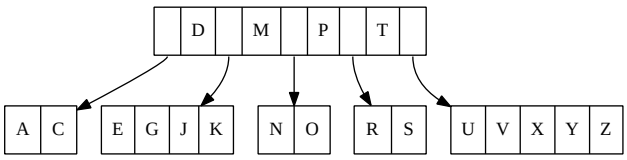
```
1:  function INSERT-NONFULL(T, k)
2:      if T is leaf then
3:          i ← 1
4:          while i ≤ |K(T)| ∧ k > k_i(T) do
5:              i ← i + 1
6:          INSERT(K(T), i, k)
7:      else
8:          i ← |K(T)|
9:          while i > 1 ∧ k < k_i(T) do
10:             i ← i − 1
11:         if c_i(T) is full then
12:             SPLIT-CHILD(T, i)
13:             if k > k_i(T) then
14:                 i ← i + 1
15:         INSERT-NONFULL(c_i(T), k)
16:     return T
```

This algorithm is recursive. In B-tree, the minimum degree t is typically relative to magnetic disk structure. Even small depth can support huge amount of data (with $t = 10$, maximum to 10 billion data can be stored in a B-tree with height of 10). The recursion can also be eliminated. This is left as exercise to the reader.

Figure 6.5 shows the result of continuously inserting keys G, M, P, X, A, C, D, E, J, K, N, O, R, S, T, U, V, Y, Z to the empty tree. The first result is the 2-3-4 tree ($t = 2$). The second result shows how it varies when $t = 3$.



(a) 2-3-4 tree.



(b) $t = 3$

Figure 6.5: Insertion result

Below example Python program implements this algorithm.

```
def insert(tr, key):
    root = tr
    if is_full(root):
        s = BTree(root.t)
        s.children.insert(0, root)
        split_child(s, 0)
        root = s
    return insert_nonfull(root, key)
```

And the insertion to non-full node is implemented as the following.

```
def insert_nonfull(tr, key):
    if is_leaf(tr):
        ordered_insert(tr.keys, key)
    else:
        i = len(tr.keys)
        while i>0 and key < tr.keys[i-1]:
            i = i-1
        if is_full(tr.children[i]):
            split_child(tr, i)
            if key>tr.keys[i]:
                i = i+1
        insert_nonfull(tr.children[i], key)
    return tr
```

Where function `ordered_insert` is used to insert an element to an ordered list. Function `is_full` tests if a node contains $2t - 1$ keys.

```
def ordered_insert(lst, x):
    i = len(lst)
    lst.append(x)
    while i>0 and lst[i]<lst[i-1]:
        (lst[i-1], lst[i]) = (lst[i], lst[i-1])
        i=i-1

def is_full(node):
    return len(node.keys) ≥ 2 * node.t - 1
```

For the array based collection, append on the tail is much more effective than insert in other position, because the later takes $O(n)$ time, if the length of the collection is n. The `ordered_insert` program firstly appends the new element at the end of the existing collection, then iterates from the last element to the first one, and checks if the current two elements next to each other are ordered. If not, these two elements will be swapped.

Insert then fixing

In functional settings, B-tree insertion can be realized in a way similar to red-black tree. When insert a key to red-black tree, it is firstly inserted as in the normal binary search tree, then recursive fixing is performed to resume the balance of the tree. B-tree can be viewed as extension to the binary search tree, that each node contains multiple keys and children. We can firstly insert the key without considering if the node is full. Then perform fixing to satisfy the minimum degree constraint.

$$insert(T, k) = fix(ins(T, k)) \tag{6.5}$$

www.dbooks.org

Function $ins(T, k)$ traverse the B-tree T from root to find a proper position where key k can be inserted. After that, function fix is applied to resume the B-tree properties. Denote B-tree in a form of $T = (K, C, t)$, where K represents keys, C represents children, and t is the minimum degree.

Below is the Haskell definition of B-tree.

```
data BTree a = Node{ keys :: [a]
                   , children :: [BTree a]
                   , degree :: Int} deriving (Eq)
```

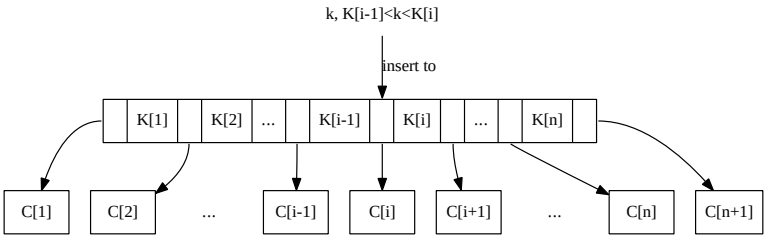
The insertion function can be provided based on this definition.

```
insert tr x = fixRoot $ ins tr x
```

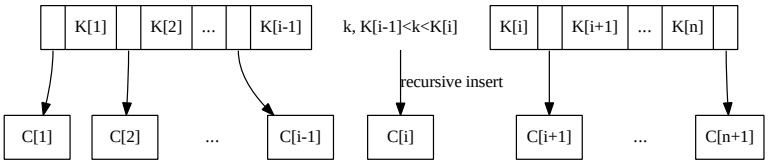
There are two cases when realize $ins(T, k)$ function. If the tree T is leaf, k is inserted to the keys; Otherwise if T is the branch node, we need recursively insert k to the proper child.

Figure 6.6 shows the branch case. The algorithm first locates the position. for certain key k_i, if the new key k to be inserted satisfy $k_{i-1} < k < k_i$, Then we need recursively insert k to child c_i.

This position divides the node into 3 parts, the left part, the child c_i and the right part.



(a) Locate the child to insert.



(b) Recursive insert.

Figure 6.6: Insert a key to a branch node

$$ins(T, k) = \begin{cases} (K' \cup \{k\} \cup K'', \phi, t) & : \quad C = \phi, (K', K'') = divide(K, k) \\ make((K', C_1), ins(c, k), (K'', C_2')) & : \quad (C_1, C_2) = split(|K'|, C) \end{cases} \tag{6.6}$$

The first clause deals with the leaf case. Function $divide(K, k)$ divide keys into two parts, all keys in the first part are not greater than k, and all rest keys are not less than k.

$$K = K' \cup K'' \wedge \forall k' \in K', k'' \in K'' \Rightarrow k' \leq k \leq k''$$

The second clause handle the branch case. Function $split(n, C)$ splits children in two parts, C_1 and C_2. C_1 contains the first n children; and C_2 contains the rest. Among C_2, the first child is denoted as c, and others are represented as C_2'.

Here the key k need be recursively inserted into child c. Function $make$ takes 3 parameter. The first and the third are pairs of key and children; the second parameter is a child node. It examines if a B-tree node made from these keys and children violates the minimum degree constraint and performs fixing if necessary.

$$make((K', C'), c, (K'', C'')) = \begin{cases} fixFull((K', C'), c, (K'', C'')) & : \quad full(c) \\ (K' \cup K'', C' \cup \{c\} \cup C'', t) & : \quad otherwise \end{cases} \tag{6.7}$$

Where function $full(c)$ tests if the child c is full. Function $fixFull$ splits the the child c, and forms a new B-tree node with the pushed up key.

$$fixFull((K', C'), c, (K'', C'')) = (K' \cup \{k'\} \cup K'', C' \cup \{c_1, c_2\} \cup C'', t) \tag{6.8}$$

Where $(c_1, k', c_2) = split(c)$. During splitting, the first $t-1$ keys and t children are extract to one new child, the last $t-1$ keys and t children form another child. The t-th key k' is pushed up.

With all the above functions defined, we can realize $fix(T)$ to complete the functional B-tree insertion algorithm. It firstly checks if the root contains too many keys. If it exceeds the limit, splitting will be applied. The split result will be used to make a new node, so the total height of the tree increases by one.

$$fix(T) = \begin{cases} c & : \quad T = (\phi, \{c\}, t) \\ (\{k'\}, \{c_1, c_2\}, t) & : \quad full(T), (c_1, k', c_2) = split(T) \\ T & : \quad otherwise \end{cases} \tag{6.9}$$

The following Haskell example code implements the B-tree insertion.

```
import qualified Data.List as L

ins (Node ks [] t) x = Node (L.insert x ks) [] t
ins (Node ks cs t) x = make (ks', cs') (ins c x) (ks'', cs'')
    where
      (ks', ks'') = L.partition (<x) ks
      (cs', (c:cs'')) = L.splitAt (length ks') cs

fixRoot (Node [] [tr] _) = tr -- shrink height
fixRoot tr = if full tr then Node [k] [c1, c2] (degree tr)
             else tr
    where
      (c1, k, c2) = split tr

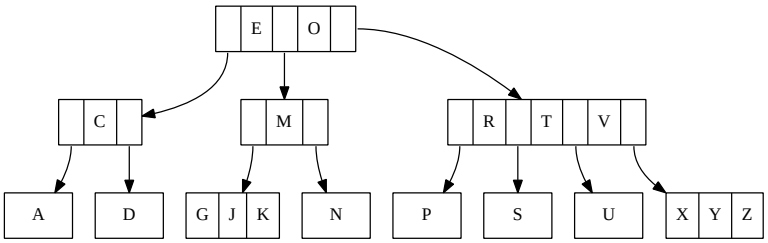
make (ks', cs') c (ks'', cs'')
    | full c = fixFull (ks', cs') c (ks'', cs'')
    | otherwise = Node (ks'++ks'') (cs'++[c]++cs'') (degree c)
```

www.dbooks.org

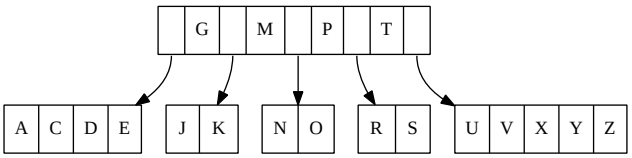
```
fixFull (ks', cs') c (ks'', cs'') = Node (ks'++[k]++ks'')
                                        (cs'++[c1,c2]++cs'') (degree c)
    where
      (c1, k, c2) = split c

full tr = (length $ keys tr) > 2*(degree tr)-1
```

Figure 6.7 shows the varies of results of building B-trees by continuously inserting keys "GMPXACDEJKNORSTUVYZ".



(a) Insert result of a 2-3-4 tree.



(b) Insert result of a B-tree with $t = 3$

Figure 6.7: Insert then fixing results

Compare to the imperative insertion result as shown in figure 6.7 we can found that there are different. However, they are all valid because all B-tree properties are satisfied.

6.3 Deletion

Deleting a key from B-tree may violate balance properties. Except the root, a node shouldn't contain too few keys less than $t - 1$, where t is the minimum degree.

Similar to the approaches for insertion, we can either do some preparation so that the node from where the key being deleted contains enough keys; or do some fixing after the deletion if the node has too few keys.

6.3.1 Merge before delete method

We start from the easiest case. If the key k to be deleted can be located in node x, and x is a leaf node, we can directly remove k from x. If x is the root (the only node of the tree), we needn't worry about there are too few keys after deletion. This case is named as case 1 later.

In most cases, we start from the root, along a path to locate where is the node contains k. If k can be located in the internal node x, there are three sub cases.

- Case 2a, If the child y precedes k contains enough keys (more than t), we replace k in node x with k', which is the predecessor of k in child y. And recursively remove k' from y.

  The predecessor of k can be easily located as the last key of child y.

  This is shown in figure 6.8.

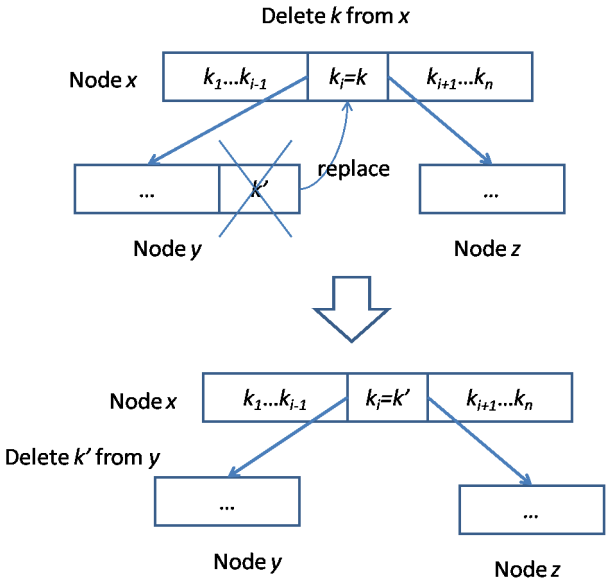


Figure 6.8: Replace and delete from predecessor.

- Case 2b, If y doesn't contain enough keys, while the child z follows k contains more than t keys. We replace k in node x with k'', which is the successor of k in child z. And recursively remove k'' from z.

  The successor of k can be easily located as the first key of child z.

  This sub-case is illustrated in figure 6.9.

- Case 2c, Otherwise, if neither y, nor z contains enough keys, we can merge y, k and z into one new node, so that this new node contains $2t - 1$ keys. After that, we can then recursively do the removing.

  Note that after merge, if the current node doesn't contain any keys, which means k is the only key in x. y and z are the only two children of x. we need shrink the tree height by one.

Figure 6.10 illustrates this sub-case.

the last case states that, if k can't be located in node x, the algorithm need find a child node c_i in x, so that the sub-tree c_i contains k. Before the deletion is recursively applied in c_i, we need make sure that there are at least t keys in c_i. If there are not enough keys, the following adjustment is performed.

- Case 3a, We check the two sibling of c_i, which are c_{i-1} and c_{i+1}. If either one contains enough keys (at least t keys), we move one key from x down

www.dbooks.org

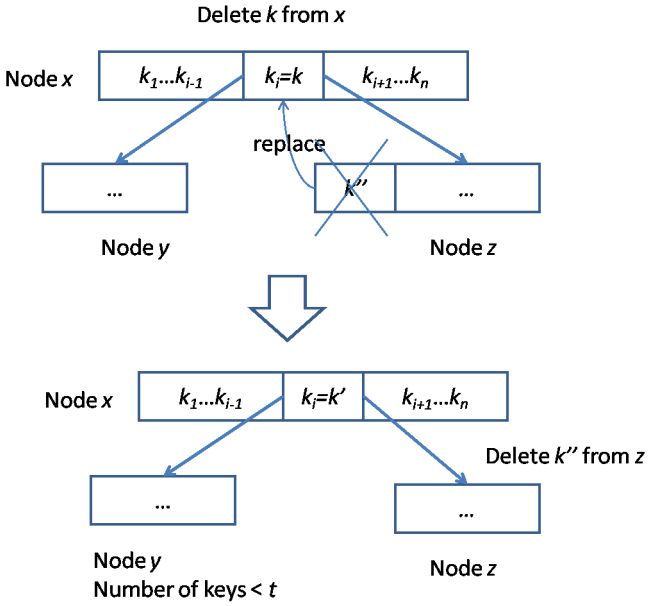


Figure 6.9: Replace and delete from successor.

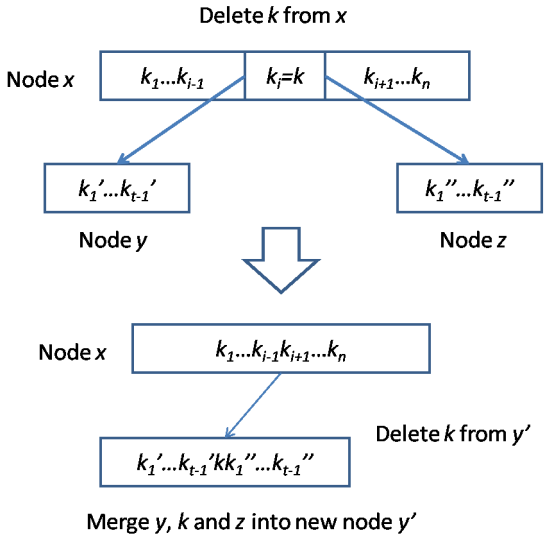


Figure 6.10: Merge and delete.

to c_i, and move one key from the sibling up to x. Also we need move the relative child from the sibling to c_i.

This operation makes c_i contains enough keys for deletion. we can next try to delete k from c_i recursively.

Figure 6.11 illustrates this case.

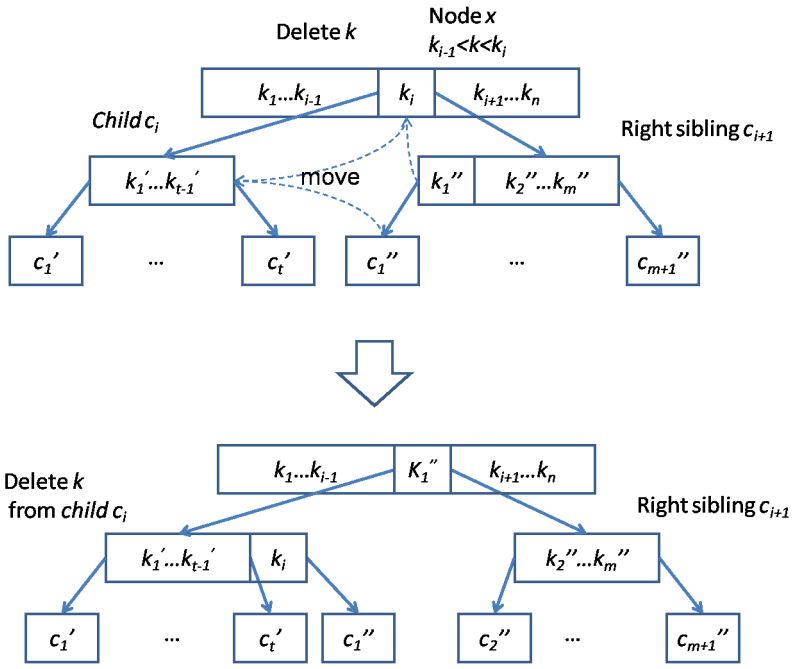


Figure 6.11: Borrow from the right sibling.

- Case 3b, In case neither one of the two siblings contains enough keys, we then merge c_i, a key from x, and either one of the sibling into a new node. Then do the deletion on this new node.

Figure 6.12 shows this case.

Before define the B-tree delete algorithm, we need provide some auxiliary functions. Function CAN-DEL tests if a node contains enough keys for deletion.

```
1: function CAN-DEL(T)
2:     return |K(T)| ≥ t
```

Procedure MERGE-CHILDREN(T, i) merges child $c_i(T)$, key $k_i(T)$, and child $c_{i+1}(T)$ into one big node.

```
1: procedure MERGE-CHILDREN(T, i)        ▷ Merge c_i(T), k_i(T), and c_{i+1}(T)
2:     x ← c_i(T)
3:     y ← c_{i+1}(T)
4:     K(x) ← K(x) ∪ {k_i(T)} ∪ K(y)
5:     C(x) ← C(x) ∪ C(y)
6:     REMOVE-AT(K(T), i)
7:     REMOVE-AT(C(T), i + 1)
```

www.dbooks.org

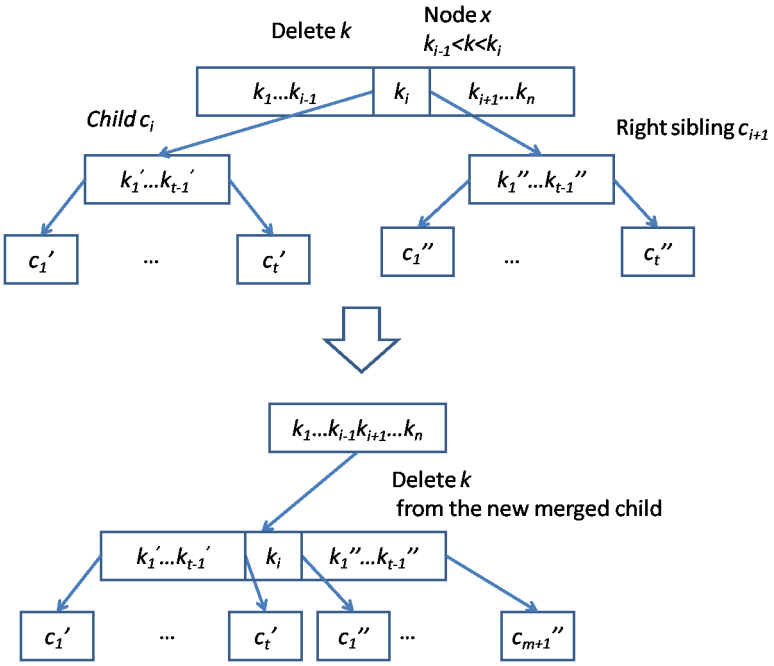


Figure 6.12: Merge c_i, k, and c_{i+1} to a new node.

Procedure MERGE-CHILDREN merges the i-th child, the i-th key, and $i+1$-th child of node T into a new child, and remove the i-th key and $i+1$-th child from T after merging.

With these functions defined, the B-tree deletion algorithm can be given by realizing the above 3 cases.

```
1:  function DELETE(T, k)
2:      i ← 1
3:      while i ≤ |K(T)| do
4:          if k = k_i(T) then
5:              if T is leaf then                                  ▷ case 1
6:                  REMOVE(K(T), k)
7:              else                                               ▷ case 2
8:                  if CAN-DEL(c_i(T)) then                        ▷ case 2a
9:                      k_i(T) ← LAST-KEY(c_i(T))
10:                     DELETE(c_i(T), k_i(T))
11:                 else if CAN-DEL(c_{i+1}(T)) then               ▷ case 2b
12:                     k_i(T) ← FIRST-KEY(c_{i+1}(T))
13:                     DELETE(c_{i+1}(T), k_i(T))
14:                 else                                           ▷ case 2c
15:                     MERGE-CHILDREN(T, i)
16:                     DELETE(c_i(T), k)
17:                     if K(T) = NIL then
18:                         T ← c_i(T)                             ▷ Shrinks height
19:             return T
20:         else if k < k_i(T) then
```

```
21:             Break
22:         else
23:             i ← i + 1

24:     if T is leaf then
25:         return T                                   ▷ k doesn't exist in T.
26:     if ¬ CAN-DEL(c_i(T)) then                      ▷ case 3
27:         if i > 1∧ CAN-DEL(c_{i-1}(T)) then         ▷ case 3a: left sibling
28:             INSERT(K(c_i(T)), k_{i-1}(T))
29:             k_{i-1}(T) ← POP-BACK(K(c_{i-1}(T)))
30:             if c_i(T) isn't leaf then
31:                 c ← POP-BACK(C(c_{i-1}(T)))
32:                 INSERT(C(c_i(T)), c)
33:         else if i ≤ |C(T)|∧ CAN-DEL(c_{i_1}(T)) then  ▷ case 3a: right sibling
34:             APPEND(K(c_i(T)), k_i(T))
35:             k_i(T) ← POP-FRONT(K(c_{i+1}(T)))
36:             if c_i(T) isn't leaf then
37:                 c ← POP-FRONT(C(c_{i+1}(T)))
38:                 APPEND(C(c_i(T)), c)
39:         else                                       ▷ case 3b
40:             if i > 1 then
41:                 MERGE-CHILDREN(T, i − 1)
42:             else
43:                 MERGE-CHILDREN(T, i)
44:     DELETE(c_i(T), k)                              ▷ recursive delete
45:     if K(T) = NIL then                             ▷ Shrinks height
46:         T ← c_1(T)
47:     return T
```

Figure 6.13, 6.14, and 6.15 show the deleting process step by step. The nodes modified are shaded.

The following example Python program implements the B-tree deletion algorithm.

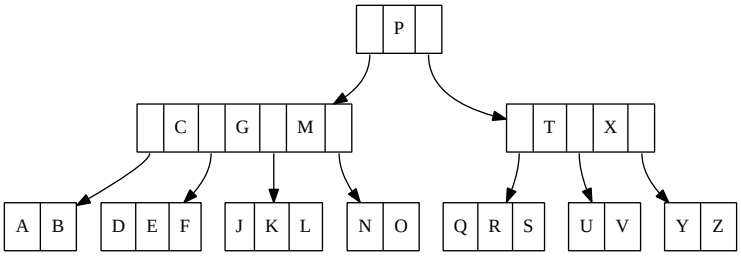
```
def can_remove(tr):
    return len(tr.keys) ≥ tr.t

def replace_key(tr, i, k):
    tr.keys[i] = k
    return k

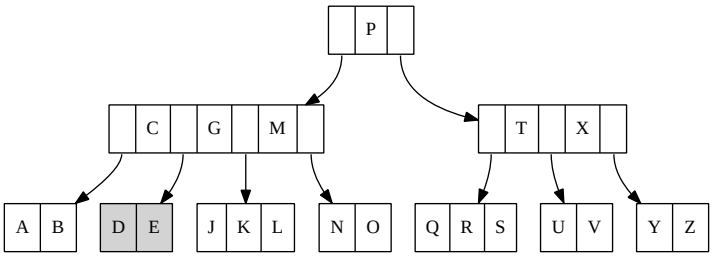
def merge_children(tr, i):
    tr.children[i].keys += [tr.keys[i]] + tr.children[i+1].keys
    tr.children[i].children += tr.children[i+1].children
    tr.keys.pop(i)
    tr.children.pop(i+1)

def B_tree_delete(tr, key):
    i = len(tr.keys)
    while i>0:
        if key == tr.keys[i-1]:
            if tr.leaf:  # case 1 in CLRS
```

www.dbooks.org

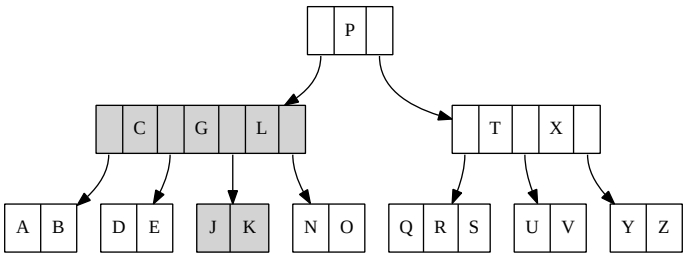


(a) A B-tree before deleting.

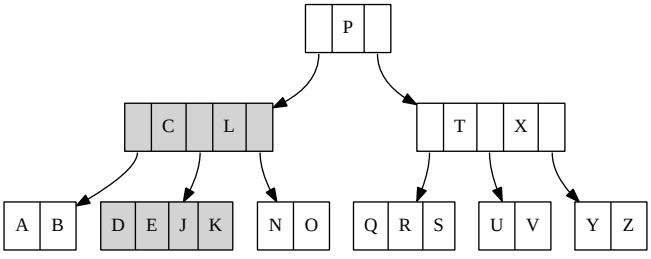


(b) After delete key 'F', case 1.

Figure 6.13: Result of B-tree deleting (1).

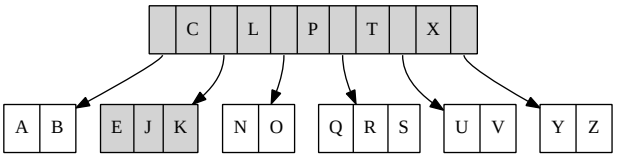


(a) After delete key 'M', case 2a.

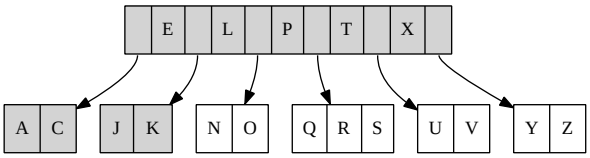


(b) After delete key 'G', case 2c.

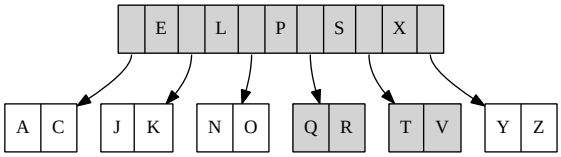
Figure 6.14: Result of B-tree deleting program (2)



(a) After delete key 'D', case 3b, and height is shrunk.



(b) After delete key 'B', case 3a, borrow from right sibling.



(c) After delete key 'U', case 3a, borrow from left sibling.

Figure 6.15: Result of B-tree deleting program (3)

```
            tr.keys.remove(key)
        else: # case 2 in CLRS
            if tr.children[i-1].can_remove(): # case 2a
                key = tr.replace_key(i-1, tr.children[i-1].keys[-1])
                B_tree_delete(tr.children[i-1], key)
            elif tr.children[i].can_remove(): # case 2b
                key = tr.replace_key(i-1, tr.children[i].keys[0])
                B_tree_delete(tr.children[i], key)
            else: # case 2c
                tr.merge_children(i-1)
                B_tree_delete(tr.children[i-1], key)
                if tr.keys==[]: # tree shrinks in height
                    tr = tr.children[i-1]
        return tr
    elif key > tr.keys[i-1]:
        break
    else:
        i = i-1
# case 3
if tr.leaf:
    return tr #key doesn't exist at all
if not tr.children[i].can_remove():
    if i>0 and tr.children[i-1].can_remove(): #left sibling
        tr.children[i].keys.insert(0, tr.keys[i-1])
        tr.keys[i-1] = tr.children[i-1].keys.pop()
        if not tr.children[i].leaf:
            tr.children[i].children.insert(0, tr.children[i-1].children.pop())
    elif i<len(tr.children) and tr.children[i+1].can_remove(): #right sibling
```

www.dbooks.org

```
            tr.children[i].keys.append(tr.keys[i])
            tr.keys[i]=tr.children[i+1].keys.pop(0)
            if not tr.children[i].leaf:
                tr.children[i].children.append(tr.children[i+1].children.pop(0))
        else: # case 3b
            if i>0:
                tr.merge_children(i-1)
            else:
                tr.merge_children(i)
    B_tree_delete(tr.children[i], key)
    if tr.keys==[]: # tree shrinks in height
        tr = tr.children[0]
    return tr
```

6.3.2 Delete and fix method

The merge and delete algorithm is a bit complex. There are several cases, and in each case, there are sub cases to deal.

Another approach to design the deleting algorithm is to perform fixing after deletion. It is similar to the insert-then-fix strategy.

$$delete(T, k) = fix(del(T, k)) \tag{6.10}$$

When delete a key from B-tree, we firstly locate which node this key is contained. We traverse from the root to the leaves till find this key in some node.

If this node is a leaf, we can remove the key, and then examine if the deletion makes the node contains too few keys to satisfy the B-tree balance properties.

If it is a branch node, removing the key breaks the node into two parts. We need merge them together. The merging is a recursive process which is shown in figure 6.16.

When do merging, if the two nodes are not leaves, we merge the keys together, and recursively merge the last child of the left part and the first child of the right part to one new node. Otherwise, if they are leaves, we merely put all keys together.

Till now, the deleting is performed in straightforward way. However, deleting decreases the number of keys of a node, and it may result in violating the B-tree balance properties. The solution is to perform fixing along the path traversed from root.

During the recursive deletion, the branch node is broken into 3 parts. The left part contains all keys less than k, includes $k_1, k_2, ..., k_{i-1}$, and children $c_1, c_2, ..., c_{i-1}$, the right part contains all keys greater than k, say $k_i, k_{i+1}, ..., k_{n+1}$, and children $c_{i+1}, c_{i+2}, ..., c_{n+1}$. Then key k is recursively deleted from child c_i. Denote the result becomes c_i' after that. We need make a new node from these 3 parts, as shown in figure 6.17.

At this time point, we need examine if c_i' contains enough keys. If there are too less keys (less than $t-1$, but not t in contrast to the merge-and-delete approach), we can either borrow a key-child pair from the left or the right part, and do inverse operation of splitting. Figure 6.18 shows example of borrowing from the left part.

If both left part and right part are empty, we can simply push c_i' up.

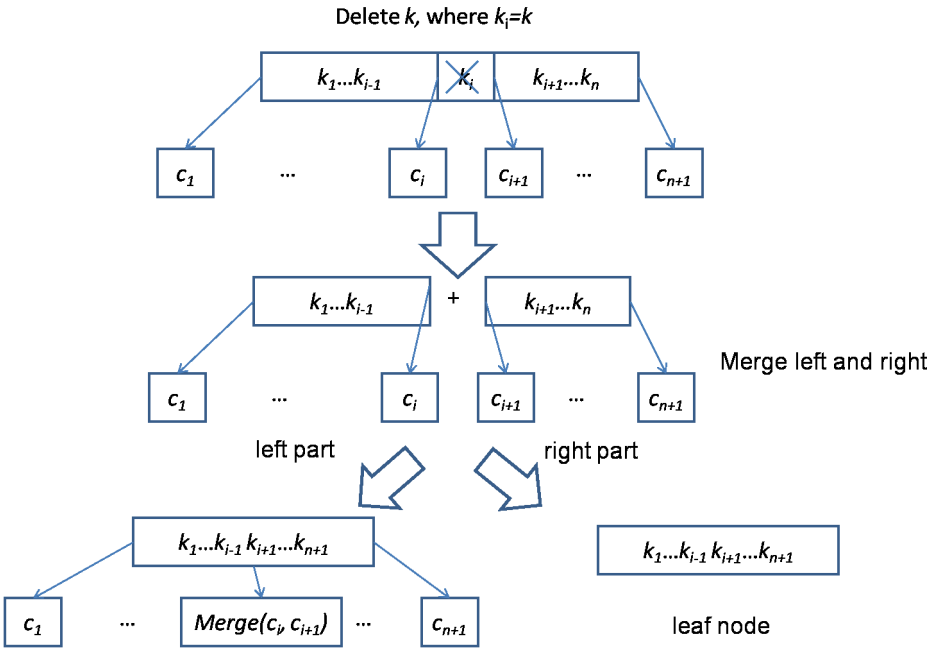


Figure 6.16: Delete a key from a branch node. Removing k_i breaks the node into 2 parts. Merging these 2 parts is a recursive process. When the two parts are leaves, the merging terminates.

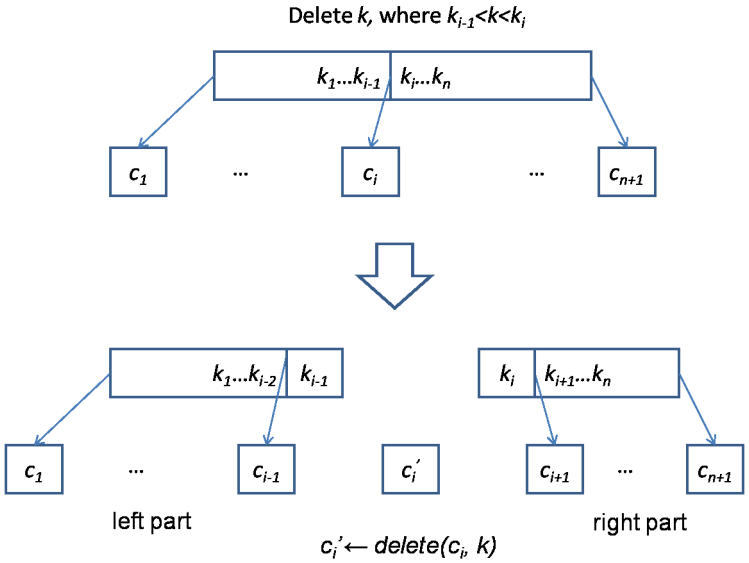


Figure 6.17: After delete key k from node c_i, denote the result as c_i'. The fixing makes a new node from the left part, c_i' and the right part.

www.dbooks.org

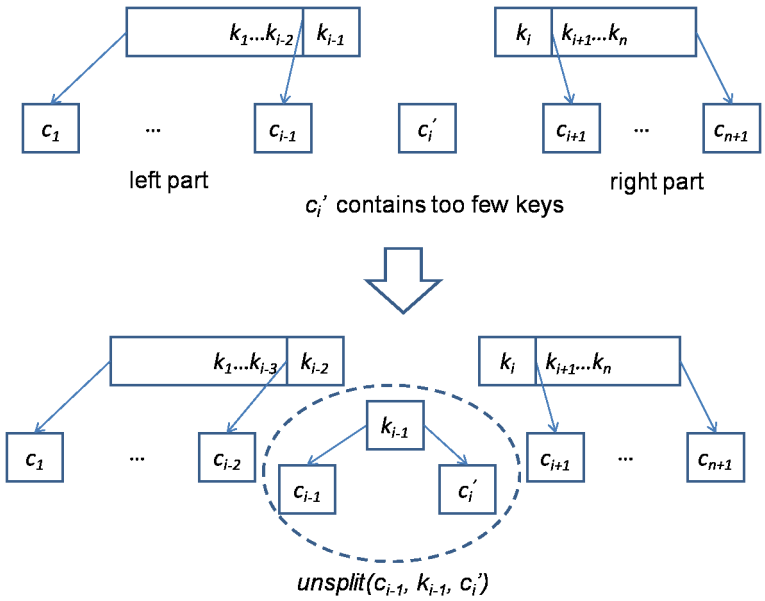


Figure 6.18: Borrow a key-child pair from left part and un-split to a new child.

Denote the B-tree as $T = (K, C, t)$, where K and C are keys and children. The $del(T, k)$ function deletes key k from the tree.

$$del(T,k) = \begin{cases} (delete(K,k), \phi, t) & : \quad C = \phi \\ merge((K_1, C_1, t), (K_2, C_2, t)) & : \quad k_i = k \\ make((K_1', C_1'), del(c, k), (K_2', C_2')) & : \quad k \notin K \end{cases} \tag{6.11}$$

If children $C = \phi$ is empty, T is leaf. k is deleted from keys directly. Otherwise, T is internal node. If $k \in K$, removing it separates the keys and children in two parts (K_1, C_1) and (K_2, C_2). They will be recursively merged.

$$\begin{aligned} K_1 &= \{k_1, k_2, ..., k_{i-1}\} \\ K_2 &= \{k_{i+1}, k_{i+2}, ..., k_m\} \\ C_1 &= \{c_1, c_2, ..., c_i\} \\ C_2 &= \{c_{i+1}, c_{i+2}, ..., c_{m+1}\} \end{aligned}$$

If $k \notin K$, we need locate a child c, and further delete k from it.

$$\begin{aligned} (K_1', K_2') &= (\{k' | k' \in K, k' < k\}, \{k' | k' \in K, k < k'\}) \\ (C_1', \{c\} \cup C_2') &= splitAt(|K_1'|, C) \end{aligned}$$

The recursive merge function is defined as the following. When merge two trees $T_1 = (K_1, C_1, t)$ and $T_2 = (K_2, C_2, t)$, if both are leaves, we create a new leave by concatenating the keys. Otherwise, the last child in C_1, and the first child in C_2 are recursively merged. And we call *make* function to form the new tree. When C_1 and C_2 are not empty, denote the last child of C_1 as $c_{1,m}$, the rest as C_1'; the first child of C_2 as $C_{2,1}$, the rest as C_2'. Below equation defines

the merge function.

$$merge(T_1, T_2) = \begin{cases} (K_1 \cup K_2, \phi, t) & : & C_1 = C_2 = \phi \\ make((K_1, C_1'), merge(c_{1,m}, c_{2,1}), (K_2, C_2')) & : & otherwise \end{cases} \tag{6.12}$$

The *make* function defined above only handles the case that a node contains too many keys due to insertion. When delete key, it may cause a node contains too few keys. We need test and fix this situation as well.

$$make((K', C'), c, (K'', C'')) = \begin{cases} fixFull((K', C'), c, (K'', C'')) & : & full(c) \\ fixLow((K', C'), c, (K'', C'')) & : & low(c) \\ (K' \cup K'', C' \cup \{c\} \cup C'', t) & : & otherwise \end{cases} \tag{6.13}$$

Where $low(T)$ checks if there are too few keys less than $t - 1$. Function $fixLow(P_l, c, P_r)$ takes three arguments, the left pair of keys and children, a child node, and the right pair of keys and children. If the left part isn't empty, we borrow a pair of key-child, and do un-splitting to make the child contain enough keys, then recursively call *make*; If the right part isn't empty, we borrow a pair from the right; and if both sides are empty, we return the child node as result. In this case, the height of the tree shrinks.

Denote the left part $P_l = (K_l, C_l)$. If K_l isn't empty, the last key and child are represented as $k_{l,m}$ and $c_{l,m}$ respectively. The rest keys and children become K_l' and C_l'; Similarly, the right part is denoted as $P_r = (K_r, C_r)$. If K_r isn't empty, the first key and child are represented as $k_{r,1}$, and $c_{r,1}$. The rest keys and children are K_r' and C_r'. Below equation gives the definition of $fixLow$.

$$fixLow(P_l, c, P_r) = \begin{cases} make((K_l', C_l'), unsplit(c_{l,m}, k_{l,m}, c), (K_r, C_r)) & : & K_l \neq \phi \\ make((K_r, C_r), unsplit(c, k_{r,1}, c_{r,1}), (K_r', C_r')) & : & K_r \neq \phi \\ c & : & otherwise \end{cases} \tag{6.14}$$

Function $unsplit(T_1, k, T_2)$ is the inverse operation to splitting. It forms a new B-tree nodes from two small nodes and a key.

$$unsplit(T_1, k, T_2) = (K_1 \cup \{k\} \cup K_2, C_1 \cup C_2, t) \tag{6.15}$$

The following example Haskell program implements the B-tree deletion algorithm.

```
import qualified Data.List as L

delete tr x = fixRoot $ del tr x

del:: (Ord a) ⇒ BTree a → a → BTree a
del (Node ks [] t) x = Node (L.delete x ks) [] t
del (Node ks cs t) x =
    case L.elemIndex x ks of
      Just i → merge (Node (take i ks) (take (i+1) cs) t)
                     (Node (drop (i+1) ks) (drop (i+1) cs) t)
      Nothing → make (ks', cs') (del c x) (ks'', cs'')
    where
```

www.dbooks.org

```
      (ks', ks'') = L.partition (<x) ks
      (cs', (c:cs'')) = L.splitAt (length ks') cs

merge (Node ks [] t) (Node ks' [] _) = Node (ks++ks') [] t
merge (Node ks cs t) (Node ks' cs' _) = make (ks, init cs)
                                             (merge (last cs) (head cs'))
                                             (ks', tail cs')

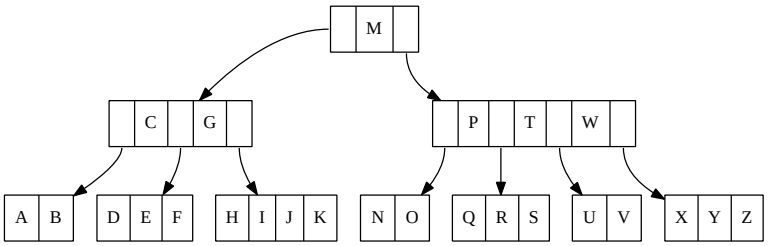
make (ks', cs') c (ks'', cs'')
    | full c = fixFull (ks', cs') c (ks'', cs'')
    | low c  = fixLow  (ks', cs') c (ks'', cs'')
    | otherwise = Node (ks'++ks'') (cs'++[c]++cs'') (degree c)

low tr = (length $ keys tr) < (degree tr)-1

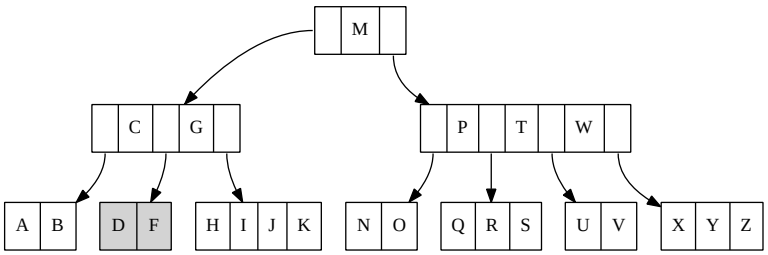
fixLow (ks'@(_:_), cs') c (ks'', cs'') = make (init ks', init cs')
                                              (unsplit (last cs') (last ks') c)
                                              (ks'', cs'')
fixLow (ks', cs') c (ks''@(_:_), cs'') = make (ks', cs')
                                              (unsplit c (head ks'') (head cs''))
                                              (tail ks'', tail cs'')
fixLow _ c _ = c

unsplit c1 k c2 = Node ((keys c1)++[k]++(keys c2))
                       ((children c1)++(children c2)) (degree c1)
```

When delete the same keys from the B-tree as in delete and fixing approach, the results are different. However, both satisfy the B-tree properties, so they are all valid.

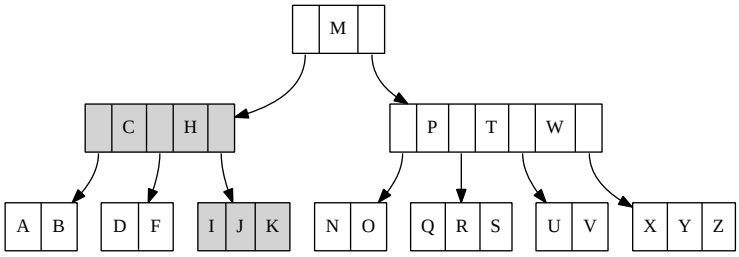


(a) B-tree before deleting

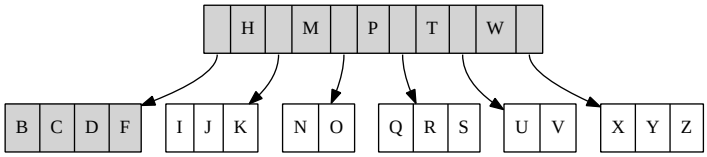


(b) After delete key 'E'.

Figure 6.19: Result of delete-then-fixing (1)

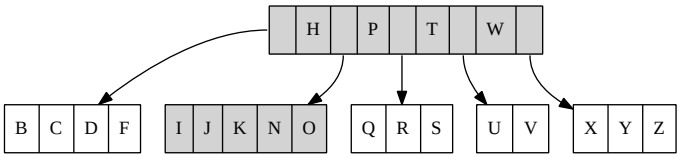


(a) After delete key 'G',

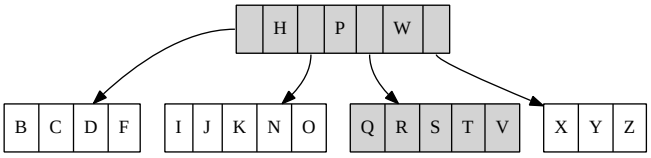


(b) After delete key 'A'.

Figure 6.20: Result of delete-then-fixing (2)



(a) After delete key 'M'.



(b) After delete key 'U'.

Figure 6.21: Result of delete-then-fixing (3)

www.dbooks.org

6.4 Searching

Searching in B-tree can be considered as the generalized tree search extended from binary search tree.

When searching in the binary tree, there are only 2 different directions, the left and the right. However, there are multiple directions in B-tree.

1: **function** SEARCH(T, k)
2: **loop**
3: $i \leftarrow 1$
4: **while** $i \leq |K(T)| \wedge k > k_i(T)$ **do**
5: $i \leftarrow i + 1$
6: **if** $i \leq |K(T)| \wedge k = k_i(T)$ **then**
7: **return** (T, i)
8: **if** T is leaf **then**
9: **return** NIL ▷ k doesn't exist
10: **else**
11: $T \leftarrow c_i(T)$

Starts from the root, this program examines each key one by one from the smallest to the biggest. In case it finds the matched key, it returns the current node and the index of this key. Otherwise, if it finds the position i that $k_i < k < k_{i+1}$, the program will next search the child node c_{i+1} for the key. If it traverses to some leaf node, and fails to find the key, the empty value is returned to indicate that this key doesn't exist in the tree.

The following example Python program implements the search algorithm.

```
def B_tree_search(tr, key):
    while True:
        for i in range(len(tr.keys)):
            if key ≤ tr.keys[i]:
                break
        if key == tr.keys[i]:
            return (tr, i)
        if tr.leaf:
            return None
        else:
            if key > tr.keys[-1]:
                i=i+1
            tr = tr.children[i]
```

The search algorithm can also be realized by recursion. When search key k in B-tree $T = (K, C, t)$, we partition the keys with k.

$$K_1 = \{k' | k' < k\}$$
$$K_2 = \{k' | k \leq k'\}$$

Thus K_1 contains all the keys less than k, and K_2 holds the rest. If the first element in K_2 is equal to k, we find the key. Otherwise, we recursively search the key in child $c_{|K_1|+1}$.

$$search(T, k) = \begin{cases} (T, |K_1| + 1) & : & k \in K_2 \\ \phi & : & C = \phi \\ search(c_{|K_1|+1}, k) & : & otherwise \end{cases} \tag{6.16}$$

Below example Haskell program implements this algorithm.

```
search :: (Ord a)⇒ BTree a → a → Maybe (BTree a, Int)
search tr@(Node ks cs _) k
    | matchFirst k $ drop len ks = Just (tr, len)
    | otherwise = if null cs then Nothing
                  else search (cs !! len) k
    where
      matchFirst x (y:_) = x==y
      matchFirst x _ = False
      len = length $ filter (<k) ks
```

6.5 Notes and short summary

In this chapter, we explained the B-tree data structure as a kind of extension from binary search tree. The background knowledge of magnetic disk access is skipped, user can refer to [2] for detail. For the three main operations, insertion, deletion, and searching, both imperative and functional algorithms are given. They traverse from the root to the leaf. All the three operations perform in time proportion to the height of the tree. Because B-tree always maintains the balance properties. The performance is ensured to bound to $O(\lg n)$ time, where n is the number of the keys in B-tree.

Exercise 6.1

- When insert a key, we need find a position, where all keys on the left are less than it, while all the others on the right are greater than it. Modify the algorithm so that the elements stored in B-tree only need support less-than and equality test.
- We assume the element being inserted doesn't exist in the tree. Modify the algorithm so that duplicated elements can be stored in a linked-list.
- Eliminate the recursion in imperative B-tree insertion algorithm.

www.dbooks.org

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. The MIT Press, 2001. ISBN: 0262032937.

[2] B-tree, Wikipedia. http://en.wikipedia.org/wiki/B-tree

[3] Chris Okasaki. “FUNCTIONAL PEARLS Red-Black Trees in a Functional Setting”. J. Functional Programming. 1998

www.dbooks.org

Part III

Heaps

www.dbooks.org

Chapter 7

Binary Heaps

7.1 Introduction

Heaps are one of the most widely used data structures–used to solve practical problems such as sorting, prioritized scheduling and in implementing graph algorithms, to name a few[2].

Most popular implementations of heaps use a kind of implicit binary heap using arrays, which is described in [2]. Examples include C++/STL heap and Python heapq. The most efficient heap sort algorithm is also realized with binary heap as proposed by R. W. Floyd [3] [5].

However, heaps can be general and realized with varies of other data structures besides array. In this chapter, explicit binary tree is used. It leads to Leftist heaps, Skew heaps, and Splay heaps, which are suitable for purely functional implementation as shown by Okasaki[6].

A heap is a data structure that satisfies the following *heap property*.

- Top operation always returns the minimum (maximum) element;
- Pop operation removes the top element from the heap while the heap property should be kept, so that the new top element is still the minimum (maximum) one;
- Insert a new element to heap should keep the heap property. That the new top is still the minimum (maximum) element;
- Other operations including merge etc should all keep the heap property.

This is a kind of recursive definition, while it doesn't limit the under ground data structure.

We call the heap with the minimum element on top as *min-heap*, while if the top keeps the maximum element, we call it *max-heap*.

7.2 Implicit binary heap by array

Considering the heap definition in previous section, one option to implement heap is by using trees. A straightforward solution is to store the minimum (maximum) element in the root of the tree, so for 'top' operation, we simply

www.dbooks.org

return the root as the result. And for 'pop' operation, we can remove the root and rebuild the tree from the children.

If binary tree is used to implement the heap, we can call it *binary heap*. This chapter explains three different realizations for binary heap.

7.2.1 Definition

The first one is implicit binary tree. Consider the problem how to represent a complete binary tree with array. (For example, try to represent a complete binary tree in the programming language doesn't support structure or record data type. Only array can be used). One solution is to pack all elements from top level (root) down to bottom level (leaves).

Figure 7.1 shows a complete binary tree and its corresponding array representation.

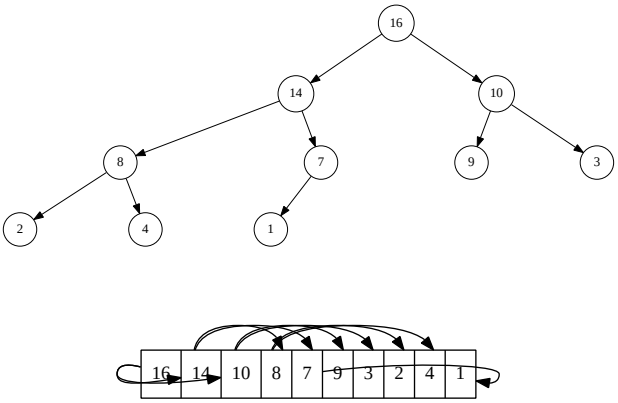


Figure 7.1: Mapping between a complete binary tree and array

This mapping between tree and array can be defined as the following equations (The array index starts from 1).

1: **function** PARENT(i)
2: **return** $\lfloor \frac{i}{2} \rfloor$

3: **function** LEFT(i)
4: **return** $2i$

5: **function** RIGHT(i)
6: **return** $2i + 1$

For a given tree node which is represented as the i-th element of the array, since the tree is complete, we can easily find its parent node as the $\lfloor i/2 \rfloor$-th element; Its left child with index of $2i$ and right child of $2i + 1$. If the index of the child exceeds the length of the array, it means this node does not have such a child (leaf for example).

In real implementation, this mapping can be calculated fast with bit-wise operation like the following example ANSI C code. Note that, the array index starts from zero in C like languages.

```
#define PARENT(i) ((((i) + 1) >> 1) - 1)

#define LEFT(i) (((i) ≪ 1) + 1)

#define RIGHT(i) (((i) + 1) ≪ 1)
```

7.2.2 Heapify

The most important thing for heap algorithm is to maintain the heap property, that the top element should be the minimum (maximum) one.

For the implicit binary heap by array, it means for a given node, which is represented as the i-th index, we can develop a method to check if both its two children are not less than the parent. In case there is violation, we need swap the parent and child recursively [2]. Note that here we assume both the two sub-trees are the valid heaps.

Below algorithm shows the iterative solution to enforce the min-heap property from a given index of the array.

```
1: function HEAPIFY(A, i)
2:     n ← |A|
3:     loop
4:         l ← LEFT(i)
5:         r ← RIGHT(i)
6:         smallest ← i
7:         if l < n ∧ A[l] < A[i] then
8:             smallest ← l
9:         if r < n ∧ A[r] < A[smallest] then
10:            smallest ← r
11:        if smallest ≠ i then
12:            EXCHANGE A[i] ↔ A[smallest]
13:            i ← smallest
14:        else
15:            return
```

For array A and the given index i, None its children should be less than $A[i]$, in case there is violation, we pick the smallest element as $A[i]$, and swap the previous $A[i]$ to child. The algorithm traverses the tree top-down to fix the heap property until either reach a leaf or there is no heap property violation.

The HEAPIFY algorithm takes $O(\lg n)$ time, where n is the number of elements. This is because the loop time is proportion to the height of the complete binary tree.

When implement this algorithm, the comparison method can be passed as a parameter, so that both min-heap and max-heap can be supported. The following ANSI C example code uses this approach.

```
typedef int (*Less)(Key, Key);
int less(Key x, Key y) { return x < y; }
int notless(Key x, Key y) { return !less(x, y); }

void heapify(Key* a, int i, int n, Less lt) {
    int l, r, m;
    while (1) {
```

www.dbooks.org

```
        l = LEFT(i);
        r = RIGHT(i);
        m = i;
        if (l < n && lt(a[l], a[i]))
            m = l;
        if (r < n && lt(a[r], a[m]))
            m = r;
        if (m != i) {
            swap(a, i, m);
            i = m;
        } else
            break;
    }
}
```

Figure 7.2 illustrates the steps when HEAPIFY processing the array $\{16, 4, 10, 14, 7, 9, 3, 2, 8, 1\}$ from the second index. The array changes to $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$ as a max-heap.

7.2.3 Build a heap

With HEAPIFY algorithm defined, it is easy to build a heap from an arbitrary array. Observe that the numbers of nodes in a complete binary tree for each level is a list like below:

$1, 2, 4, 8, ..., 2^i, ...$

The only exception is the last level. Since the tree may not full (note that complete binary tree doesn't mean full binary tree), the last level contains at most 2^{p-1} nodes, where $2^p + 1 \leq n$ and n is the length of the array.

The HEAPIFY algorithm doesn't have any effect on leave node. We can skip applying HEAPIFY for all leaves. In other words, all leaf nodes have already satisfied the heap property. We only need start checking and maintain the heap property from the last branch node. the index of the last branch node is no greater than $\lfloor n/2 \rfloor$.

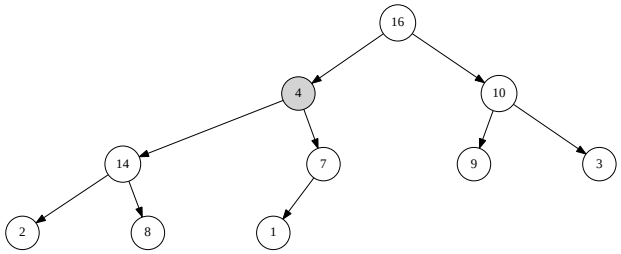
Based on this fact, we can build a heap with the following algorithm. (Assume the heap is min-heap).

1: **function** BUILD-HEAP(A)
2: $n \leftarrow |A|$
3: **for** $i \leftarrow \lfloor n/2 \rfloor$ down to 1 **do**
4: HEAPIFY(A, i)

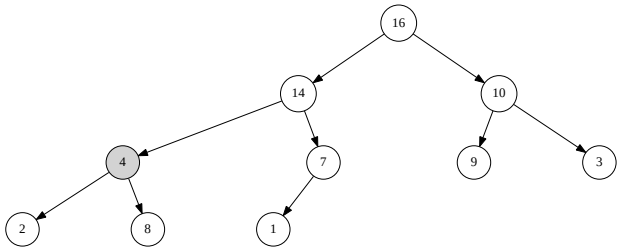
Although the complexity of HEAPIFY is $O(\lg n)$, the running time of BUILD-HEAP is not bound to $O(n \lg n)$ but $O(n)$. It is a linear time algorithm. This can be deduced as the following:

The heap is built by skipping all leaves. Given n nodes, there are at most $n/4$ nodes being compared and moved down 1 time; at most $n/8$ nodes being compared and moved down 2 times; at most $n/16$ nodes being compared and moved down 3 times,... Thus the upper bound of total comparison and moving time is:

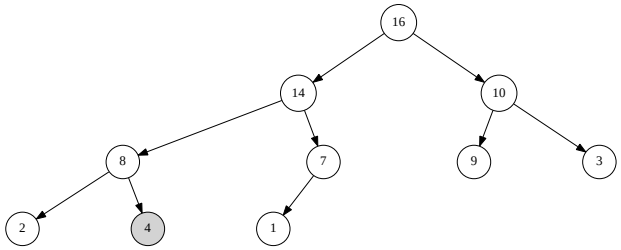
$$S = n(\frac{1}{4} + 2\frac{1}{8} + 3\frac{1}{16} + ...) \tag{7.1}$$



(a) Step 1, 14 is the biggest element among 4, 14, and 7. Swap 4 with the left child;



(b) Step 2, 8 is the biggest element among 2, 4, and 8. Swap 4 with the right child;



(c) 4 is the leaf node. It hasn't any children. Process terminates.

Figure 7.2: Heapify example, a max-heap case.

www.dbooks.org

Times by 2 for both sides, we have:

$$2S = n(\frac{1}{2} + 2\frac{1}{4} + 3\frac{1}{8} + ...) \tag{7.2}$$

Substract equation (7.1) from (7.2):

$$S = n(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...) = n$$

Below ANSI C example program implements this heap building function.

```
void build_heap(Key* a, int n, Less lt) {
    int i;
    for (i = (n-1) >> 1; i ≥ 0; --i)
        heapify(a, i, n, lt);
}
```

Figure 7.3, 7.4 and 7.5 show the steps when building a max-heap from array $\{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$. The node in black color is the one where HEAPIFY being applied, the nodes in gray color are swapped in order to keep the heap property.

7.2.4 Basic heap operations

The generic definition of heap (not necessarily the binary heap) demands us to to provide basic operations for accessing and modifying data.

The most important operations include accessing the top element (find the minimum or maximum one), popping the top element from the heap, finding the top k elements, decreasing a key (for min-heap. It is increasing a key for max-heap), and insertion.

For the binary tree, most of operations are bound to $O(\lg n)$ in worst-case, some of them, such as top is $O(1)$ constant time.

Access the top element

For the binary tree realization, it is the root stores the minimum (maximum) value. This is the first element in the array.

1: **function** TOP(A)
2: **return** $A[1]$

This operation is trivial. It takes $O(1)$ time. Here we skip the error handling for empty case. If the heap is empty, one option is to raise an error.

Heap Pop

Pop operation is more complex than accessing the top, because the heap property has to be maintained after the top element is removed.

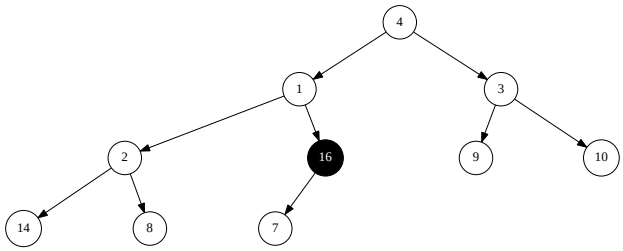
The solution is to apply HEAPIFY algorithm to the next element after the root is removed.

One simple but slow method based on this idea looks like the following.

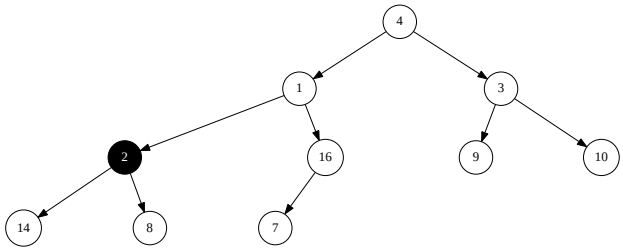
1: **function** POP-SLOW(A)
2: $x \leftarrow$ TOP(A)
3: REMOVE(A, 1)

4	1	3	2	16	9	10	14	8	7

(a) An array in arbitrary order before heap building process;



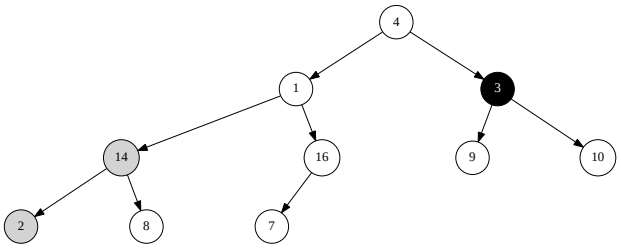
(b) Step 1, The array is mapped to binary tree. The first branch node, which is 16 is examined;



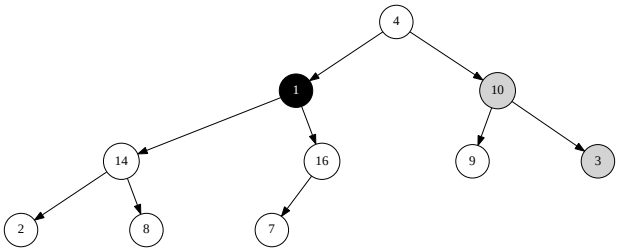
(c) Step 2, 16 is the largest element in current sub tree, next is to check node with value 2;

Figure 7.3: Build a heap from the arbitrary array. Gray nodes are changed in each step, black node will be processed next step.

www.dbooks.org

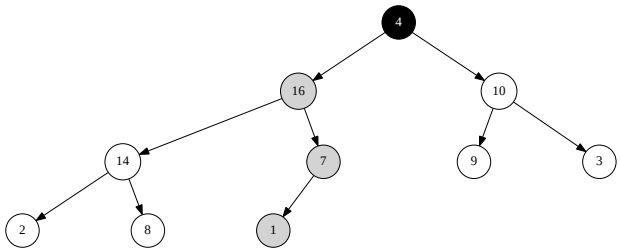


(a) Step 3, 14 is the largest value in the sub-tree, swap 14 and 2; next is to check node with value 3;

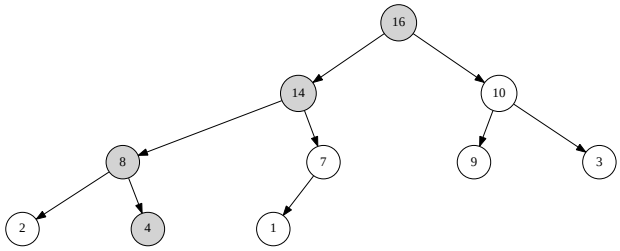


(b) Step 4, 10 is the largest value in the sub-tree, swap 10 and 3; next is to check node with value 1;

Figure 7.4: Build a heap from the arbitrary array. Gray nodes are changed in each step, black node will be processed next step.



(a) Step 5, 16 is the largest value in current sub-tree, swap 16 and 1 first; then similarly, swap 1 and 7; next is to check the root node with value 4;



(b) Step 6, Swap 4 and 16, then swap 4 and 14, and then swap 4 and 8; And the whole build process finish.

Figure 7.5: Build a heap from the arbitrary array. Gray nodes are changed in each step, black node will be processed next step.

www.dbooks.org

```
4:     if A is not empty then
5:         HEAPIFY(A, 1)
6:     return x
```

This algorithm firstly records the top element in x, then it removes the first element from the array, the size of this array is reduced by one. After that if the array isn't empty, HEAPIFY will applied to the new array from the first element (It was previously the second one).

Removing the first element from array takes $O(n)$ time, where n is the length of the array. This is because we need shift all the rest elements one by one. This bottle neck slows the whole algorithm to linear time.

In order to solve this problem, one alternative is to swap the first element with the last one in the array, then shrink the array size by one.

```
1: function POP(A)
2:     x ← TOP(A)
3:     n ← HEAP-SIZE(A)
4:     EXCHANGE A[1] ↔ A[n]
5:     REMOVE(A, n)
6:     if A is not empty then
7:         HEAPIFY(A, 1)
8:     return x
```

Removing the last element from the array takes only constant $O(1)$ time, and HEAPIFY is bound to $O(\lg n)$. Thus the whole algorithm performs in $O(\lg n)$ time. The following example ANSI C program implements this algorithm[1].

```
Key pop(Key* a, int n, Less lt) {
    swap(a, 0, --n);
    heapify(a, 0, n, lt);
    return a[n];
}
```

Find the top k elements

With pop defined, it is easy to find the top k elements from array. we can build a max-heap from the array, then perform pop operation k times.

```
1: function TOP-K(A, k)
2:     R ← φ
3:     BUILD-HEAP(A)
4:     for i ← 1 to MIN(k, |A|) do
5:         APPEND(R, POP(A))
6:     return R
```

If k is greater than the length of the array, we need return the whole array as the result. That's why it calls the MIN function to determine the number of loops.

Below example Python program implements the top-k algorithm.

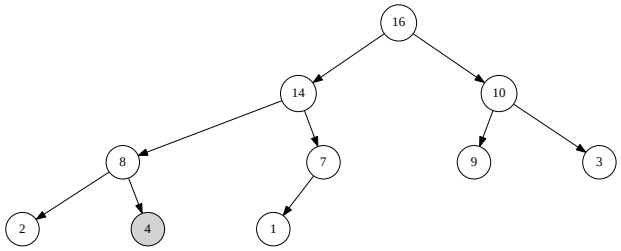
```
def top_k(x, k, less_p = MIN_HEAP):
    build_heap(x, less_p)
    return [heap_pop(x, less_p) for _ in range(min(k, len(x)))]
```

[1]This program does not actually remove the last element, it reuse the last cell to store the popped result

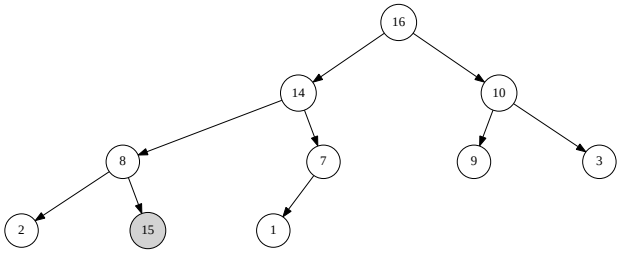
Decrease key

Heap can be used to implement priority queue. It is important to support key modification in heap. One typical operation is to increase the priority of a tasks so that it can be performed earlier.

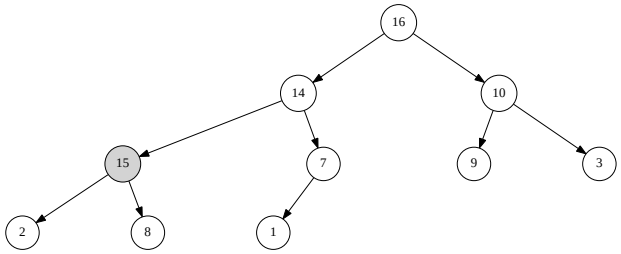
Here we present the decrease key operation for a min-heap. The corresponding operation is increase key for max-heap. Figure 7.6 and 7.7 illustrate such a case for a max-heap. The key of the 9-th node is increased from 4 to 15.



(a) The 9-th node with key 4 will be modified;



(b) The key is modified to 15, which is greater than its parent;

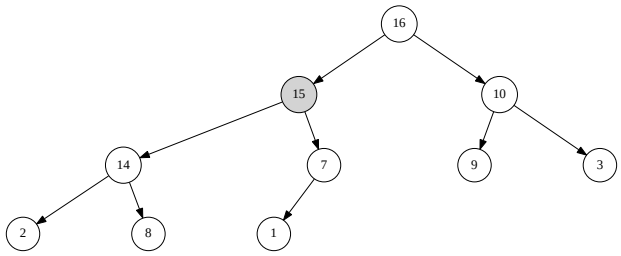


(c) According the max-heap property, 8 and 15 are swapped.

Figure 7.6: Example process when increase a key in a max-heap.

Once a key is decreased in a min-heap, it may make the node conflict with the heap property, that the key may be less than some ancestor. In order to maintain the invariant, the following auxiliary algorithm is defined to resume

www.dbooks.org



(a) Since 15 is greater than its parent 14, they are swapped. After that, because 15 is less than 16, the process terminates.

Figure 7.7: Example process when increase a key in a max-heap.

the heap property.

1: **function** HEAP-FIX(A, i)
2: **while** $i > 1 \wedge A[i] < A[$ PARENT(i) $]$ **do**
3: EXCHANGE $A[i] \leftrightarrow A[$ PARENT(i) $]$
4: $i \leftarrow$ PARENT(i)

This algorithm repeatedly compares the keys of parent node and current node. It swap the nodes if the parent contains the smaller key. This process is performed from the current node towards the root node till it finds that the parent node holds the smaller key.

With this auxiliary algorithm, decrease key can be realized as below.

1: **function** DECREASE-KEY(A, i, k)
2: **if** $k < A[i]$ **then**
3: $A[i] \leftarrow k$
4: HEAP-FIX(A, i)

This algorithm is only triggered when the new key is less than the original key. The performance is bound to $O(\lg n)$. Below example ANSI C program implements the algorithm.

```
void heap_fix(Key* a, int i, Less lt) {
    while (i > 0 && lt(a[i], a[PARENT(i)])) {
        swap(a, i, PARENT(i));
        i = PARENT(i);
    }
}

void decrease_key(Key* a, int i, Key k, Less lt) {
    if (lt(k, a[i])) {
        a[i] = k;
        heap_fix(a, i, lt);
    }
}
```

Insertion

Insertion can be implemented by using DECREASE-KEY [2]. A new node with ∞ as key is created. According to the min-heap property, it should be the last element in the under ground array. After that, the key is decreased to the value to be inserted, and DECREASE-KEY is called to fix any violation to the heap property.

Alternatively, we can reuse HEAP-FIX to implement insertion. The new key is directly appended at the end of the array, and the HEAP-FIX is applied to this new node.

1: **function** HEAP-PUSH(A, k)
2: APPEND(A, k)
3: HEAP-FIX($A, |A|$)

The following example Python program implements the heap insertion algorithm.

```
def heap_insert(x, key, less_p = MIN_HEAP):
    i = len(x)
    x.append(key)
    heap_fix(x, i, less_p)
```

7.2.5 Heap sort

Heap sort is interesting application of heap. According to the heap property, the min(max) element can be easily accessed by from the top of the heap. A straightforward way to sort a list of values is to build a heap from them, then continuously pop the smallest element till the heap is empty.

The algorithm based on this idea can be defined like below.

1: **function** HEAP-SORT(A)
2: $R \leftarrow \phi$
3: BUILD-HEAP(A)
4: **while** $A \neq \phi$ **do**
5: APPEND(R, HEAP-POP(A))
6: **return** R

The following Python example program implements this definition.

```
def heap_sort(x, less_p = MIN_HEAP):
    res = []
    build_heap(x, less_p)
    while x!=[]:
        res.append(heap_pop(x, less_p))
    return res
```

When sort n elements, the BUILD-HEAP is bound to $O(n)$. Since pop is $O(\lg n)$, and it is called n times, so the overall sorting takes $O(n \lg n)$ time to run. Because we use another list to hold the result, the space requirement is $O(n)$.

Robert. W. Floyd found a fast implementation of heap sort. The idea is to build a max-heap instead of min-heap, so the first element is the biggest one. Then the biggest element is swapped with the last element in the array, so that it is in the right position after sorting. As the last element becomes the new

www.dbooks.org

top, it may violate the heap property. We can shrink the heap size by one and perform HEAPIFY to resume the heap property. This process is repeated till there is only one element left in the heap.

1: **function** HEAP-SORT(A)
2: BUILD-MAX-HEAP(A)
3: **while** $|A| > 1$ **do**
4: EXCHANGE $A[1] \leftrightarrow A[n]$
5: $|A| \leftarrow |A| - 1$
6: HEAPIFY($A, 1$)

This is in-place algorithm, it needn't any extra spaces to hold the result. The following ANSI C example code implements this algorithm.

```
void heap_sort(Key* a, int n) {
    build_heap(a, n, notless);
    while(n > 1) {
        swap(a, 0, --n);
        heapify(a, 0, n, notless);
    }
}
```

Exercise 7.1

- Somebody considers one alternative to realize in-place heap sort. Take sorting the array in ascending order as example, the first step is to build the array as a minimum heap A, but not the maximum heap like the Floyd's method. After that the first element a_1 is in the correct place. Next, treat the rest $\{a_2, a_3, ..., a_n\}$ as a new heap, and perform HEAPIFY to them from a_2 for these $n - 1$ elements. Repeating this advance and HEAPIFY step from left to right would sort the array. The following example ANSI C code illustrates this idea. Is this solution correct? If yes, prove it; if not, why?

```
void heap_sort(Key* a, int n) {
    build_heap(a, n, less);
    while(--n)
        heapify(++a, 0, n, less);
}
```

- Because of the same reason, can we perform HEAPIFY from left to right k times to realize in-place top-k algorithm like below ANSI C code?

```
int tops(int k, Key* a, int n, Less lt) {
    build_heap(a, n, lt);
    for (k = MIN(k, n) - 1; k; --k)
        heapify(++a, 0, --n, lt);
    return k;
}
```

7.3 Leftist heap and Skew heap, the explicit binary heaps

Instead of using implicit binary tree by array, it is natural to consider why we can't use explicit binary tree to realize heap?

There are some problems must be solved if we turn into explicit binary tree as the under ground data structure.

The first problem is about the HEAP-POP or DELETE-MIN operation. Consider the binary tree is represented in form of left, key, and right as (L, k, R), which is shown in figure 7.8

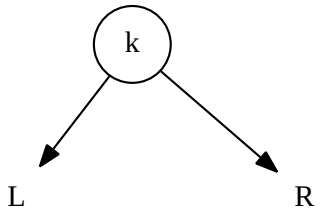


Figure 7.8: A binary tree, all elements in children are not less than k.

If k is the top element, all elements in left and right children are not less than k in a min-heap. After k is popped, only left and right children are left. They have to be merged to a new tree. Since heap property should be maintained after merge, the new root is still the smallest element.

Because both left and right children are binary trees conforming heap property, the two trivial cases can be defined immediately.

$$merge(H_1, H_2) = \begin{cases} H_2 & : & H_1 = \phi \\ H_1 & : & H_2 = \phi \\ ? & : & otherwise \end{cases}$$

Where ϕ means empty heap.

If neither left nor right child is empty, because they all fit heap property, the top elements of them are all the minimum respectively. We can compare these two roots, and select the smaller as the new root of the merged heap.

For instance, let $L = (A, x, B)$ and $R = (A', y, B')$, where A, A', B, and B' are all sub trees. If $x < y$, x will be the new root. We can either keep A, and recursively merge B and R; or keep B, and merge A and R, so the new heap can be one of the following.

- $(merge(A, R), x, B)$
- $(A, x, merge(B, R))$

www.dbooks.org

Both are correct. One simplified solution is to only merge the right sub tree. *Leftist* tree provides a systematically approach based on this idea.

7.3.1 Definition

The heap implemented by Leftist tree is called Leftist heap. Leftist tree is first introduced by C. A. Crane in 1972[6].

Rank (S-value)

In Leftist tree, a rank value (or S value) is defined for each node. Rank is the distance to the nearest external node. Where external node is the NIL concept extended from the leaf node.

For example, in figure 7.9, the rank of NIL is defined 0, consider the root node 4, The nearest external node is the child of node 8. So the rank of root node 4 is 2. Because node 6 and node 8 both only contain NIL, so their rank values are 1. Although node 5 has non-NIL left child, However, since the right child is NIL, so the rank value, which is the minimum distance to NIL is still 1.

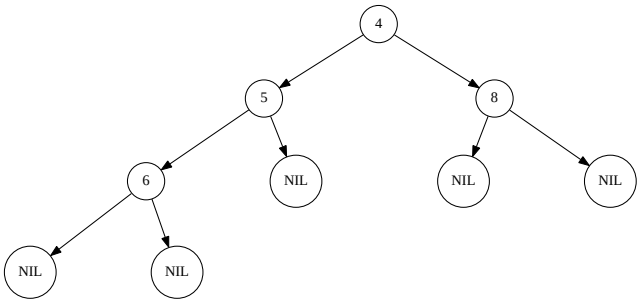


Figure 7.9: $rank(4) = 2$, $rank(6) = rank(8) = rank(5) = 1$.

Leftist property

With rank defined, we can create a strategy when merging.

- Every time when merging, we always merge to right child; Denote the rank of the new right sub tree as r_r;
- Compare the ranks of the left and right children, if the rank of left sub tree is r_l and $r_l < r_r$, we swap the left and the right children.

We call this 'Leftist property'. In general, a Leftist tree always has the shortest path to some external node on the right.

Leftist tree tends to be very unbalanced, However, it ensures important property as specified in the following theorem.

Theorem 7.3.1. *If a Leftist tree T contains n internal nodes, the path from root to the rightmost external node contains at most $\lfloor \log(n+1) \rfloor$ nodes.*

We skip the proof here, readers can refer to [7] and [1] for more information. With this theorem, algorithms operate along this path are all bound to $O(\lg n)$.

We can reuse the binary tree definition, and augment with a rank field to define the Leftist tree, for example in form of (r, k, L, R) for non-empty case. Below Haskell code defines the Leftist tree.

```
data LHeap a = E -- Empty
             | Node Int a (LHeap a) (LHeap a) -- rank, element, left, right
```

For empty tree, the rank is defined as zero. Otherwise, it's the value of the augmented field. A $rank(H)$ function can be given to cover both cases.

$$rank(H) = \begin{cases} 0 & : \quad H = \phi \\ r & : \quad otherwise, H = (r, k, L, R) \end{cases} \tag{7.3}$$

Here is the example Haskell rank function.

```
rank E = 0
rank (Node r _ _ _) = r
```

In the rest of this section, we denote $rank(H)$ as r_H

7.3.2 Merge

In order to realize 'merge', we need develop the auxiliary algorithm to compare the ranks and swap the children if necessary.

$$mk(k, A, B) = \begin{cases} (r_A + 1, k, B, A) & : \quad r_A < r_B \\ (r_B + 1, k, A, B) & : \quad otherwise \end{cases} \tag{7.4}$$

This function takes three arguments, a key and two sub trees A, and B. if the rank of A is smaller, it builds a bigger tree with B as the left child, and A as the right child. It increment the rank of A by 1 as the rank of the new tree; Otherwise if B holds the smaller rank, then A is set as the left child, and B becomes the right. The resulting rank is $r_b + 1$.

The reason why rank need be increased by one is because there is a new key added on top of the tree. It causes the rank increasing.

Denote the key, the left and right children for H_1 and H_2 as k_1, L_1, R_1, and k_2, L_2, R_2 respectively. The $merge(H_1, H_2)$ function can be completed by using this auxiliary tool as below

$$merge(H_1, H_2) = \begin{cases} H_2 & : \quad H_1 = \phi \\ H_1 & : \quad H_2 = \phi \\ mk(k_1, L_1, merge(R_1, H_2)) & : \quad k_1 < k_2 \\ mk(k_2, L_2, merge(H_1, R_2)) & : \quad otherwise \end{cases} \tag{7.5}$$

The $merge$ function is always recursively called on the right side, and the Leftist property is maintained. These facts ensure the performance being bound to $O(\lg n)$.

The following Haskell example code implements the merge program.

```
merge E h = h
merge h E = h
merge h1@(Node _ x l r) h2@(Node _ y l' r') =
```

www.dbooks.org

```
    if x < y then makeNode x l (merge r h2)
    else makeNode y l' (merge h1 r')

makeNode x a b = if rank a < rank b then Node (rank a + 1) x b a
                 else Node (rank b + 1) x a b
```

Merge operation in implicit binary heap by array

Implicit binary heap by array performs very fast in most cases, and it fits modern computer with cache technology well. However, merge is the algorithm bounds to $O(n)$ time. The typical realization is to concatenate two arrays together and make a heap for this array [13].

1: **function** MERGE-HEAP(A, B)
2: $C \leftarrow$ CONCAT(A, B)
3: BUILD-HEAP(C)

7.3.3 Basic heap operations

Most of the basic heap operations can be implemented with *merge* algorithm defined above.

Top and pop

Because the smallest element is always held in root, it's trivial to find the minimum value. It's constant $O(1)$ operation. Below equation extracts the root from non-empty heap $H = (r, k, L, R)$. The error handling for empty case is skipped here.

$$top(H) = k \tag{7.6}$$

For pop operation, firstly, the top element is removed, then left and right children are merged to a new heap.

$$pop(H) = merge(L, R) \tag{7.7}$$

Because it calls *merge* directly, the pop operation on Leftist heap is bound to $O(\lg n)$.

Insertion

To insert a new element, one solution is to create a single leaf node with the element, and then merge this leaf node to the existing Leftist tree.

$$insert(H, k) = merge(H, (1, k, \phi, \phi)) \tag{7.8}$$

It is $O(\lg n)$ algorithm since insertion also calls *merge* directly.

There is a convenient way to build the Leftist heap from a list. We can continuously insert the elements one by one to the empty heap. This can be realized by folding.

$$build(L) = fold(insert, \phi, L) \tag{7.9}$$

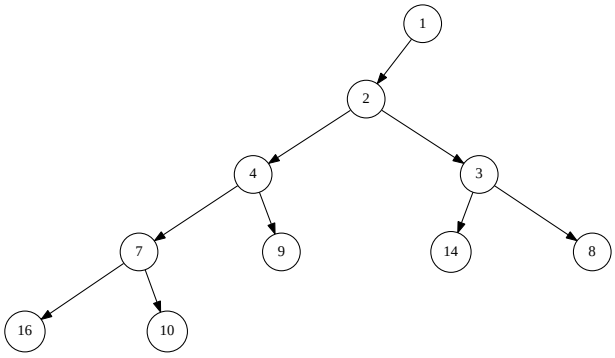


Figure 7.10: A Leftist tree built from list $\{9, 4, 16, 7, 10, 2, 14, 3, 8, 1\}$.

Figure 7.10 shows one example Leftist tree built in this way.

The following example Haskell code gives reference implementation for the Leftist tree operations.

```
insert h x = merge (Node 1 x E E) h

findMin (Node _ x _ _) = x

deleteMin (Node _ _ l r) = merge l r

fromList = foldl insert E
```

7.3.4 Heap sort by Leftist Heap

With all the basic operations defined, it's straightforward to implement heap sort. We can firstly turn the list into a Leftist heap, then continuously extract the minimum element from it.

$$sort(L) = heapSort(build(L)) \tag{7.10}$$

$$heapSort(H) = \begin{cases} \phi & : \quad H = \phi \\ \{top(H)\} \cup heapSort(pop(H)) & : \quad otherwise \end{cases} \tag{7.11}$$

Because pop is logarithm operation, and it is recursively called n times, this algorithm takes $O(n \lg n)$ time in total. The following Haskell example program implements heap sort with Leftist tree.

```
heapSort = hsort ∘ fromList where
    hsort E = []
    hsort h = (findMin h):(hsort $ deleteMin h)
```

www.dbooks.org

7.3.5 Skew heaps

Leftist heap leads to quite unbalanced structure sometimes. Figure 7.11 shows one example. The Leftist tree is built by folding on list $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$.

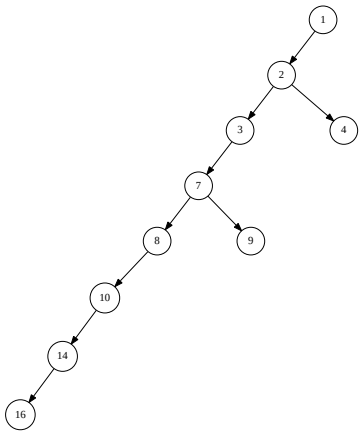


Figure 7.11: A very unbalanced Leftist tree build from list $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$.

Skew heap (or *self-adjusting heap*) simplifies Leftist heap realization and intends to solve the balance issue[9] [10].

When construct the Leftist heap, we swap the left and right children during merge if the rank on left side is less than the right side. This comparison-and-swap strategy doesn't work when either sub tree has only one child. Because in such case, the rank of the sub tree is always 1 no matter how big it is. A 'Brute-force' approach is to swap the left and right children every time when merge. This idea leads to Skew heap.

Definition of Skew heap

Skew heap is the heap realized with Skew tree. Skew tree is a special binary tree. The minimum element is stored in root. Every sub tree is also a skew tree.

It needn't keep the rank (or S-value) field. We can reuse the binary tree definition for Skew heap. The tree is either empty, or in a pre-order form (k, L, R). Below Haskell code defines Skew heap like this.

```
data SHeap a = E -- Empty
             | Node a (SHeap a) (SHeap a) -- element, left, right
```

Merge

The merge algorithm tends to be very simple. When merge two non-empty Skew trees, we compare the roots, and pick the smaller one as the new root, then the other tree contains the bigger element is merged onto one sub tree, finally, the tow children are swapped. Denote $H_1 = (k_1, L_1, R_1)$ and $H_2 = (k_2, L_2, R_2)$ if

they are not empty. if $k_1 < k_2$ for instance, select k_1 as the new root. We can either merge H_2 to L_1, or merge H_2 to R_1. Without loss of generality, let's merge to R_1. And after swapping the two children, the final result is $(k_1, merge(R_1, H_2), L_1)$. Take account of edge cases, the merge algorithm is defined as the following.

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ (k_1, merge(R_1, H_2), L_1) & : & k_1 < k_2 \\ (k_2, merge(H_1, R_2), L_2) & : & otherwise \end{cases} \tag{7.12}$$

All the rest operations, including insert, top and pop are all realized as same as the Leftist heap by using merge, except that we needn't the rank any more.

Translating the above algorithm into Haskell yields the following example program.

```
merge E h = h
merge h E = h
merge h1@(Node x l r) h2@(Node y l' r') =
    if x < y then Node x (merge r h2) l
    else Node y (merge h1 r') l'

insert h x = merge (Node x E E) h

findMin (Node x _ _) = x

deleteMin (Node _ l r) = merge l r
```

Different from the Leftist heap, if we feed ordered list to Skew heap, it can build a fairly balanced binary tree as illustrated in figure 7.12.

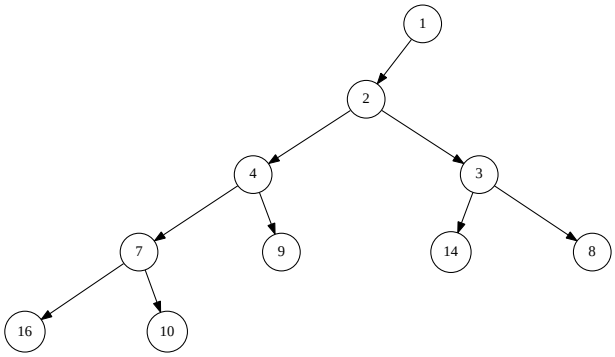


Figure 7.12: Skew tree is still balanced even the input is an ordered list $\{1, 2, ..., 10\}$.

www.dbooks.org

7.4 Splay heap

The Leftist heap and Skew heap show the fact that it's quite possible to realize heap data structure with explicit binary tree. Skew heap gives one method to solve the tree balance problem. Splay heap on the other hand, use another method to keep the tree balanced.

The binary trees used in Leftist heap and Skew heap are not Binary Search tree (BST). If we turn the underground data structure to binary search tree, the minimum(or maximum) element is not root any more. It takes $O(\lg n)$ time to find the minimum(or maximum) element.

Binary search tree becomes inefficient if it isn't well balanced. Most operations degrade to $O(n)$ in the worst case. Although red-black tree can be used to realize binary heap, it's overkill. Splay tree provides a light weight implementation with acceptable dynamic balancing result.

7.4.1 Definition

Splay tree uses cache-like approach. It keeps rotating the current access node close to the top, so that the node can be accessed fast next time. It defines such kinds of operation as "Splay". For the unbalanced binary search tree, after several splay operations, the tree tends to be more and more balanced. Most basic operations of Splay tree perform in amortized $O(\lg n)$ time. Splay tree was invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985[11] [12].

Splaying

There are two methods to do splaying. The first one need deal with many different cases, but can be implemented fairly easy with pattern matching. The second one has a uniformed form, but the implementation is complex.

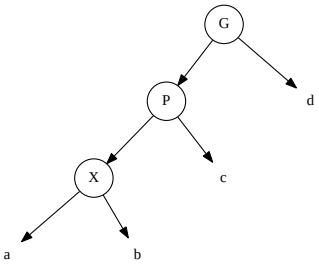
Denote the node currently being accessed as X, the parent node as P, and the grand parent node as G (If there are). There are 3 steps for splaying. Each step contains 2 symmetric cases. For illustration purpose, only one case is shown for each step.

- *Zig-zig step.* As shown in figure 7.13, in this case, X and P are children on the same side of G, either both on left or right. By rotating 2 times, X becomes the new root.

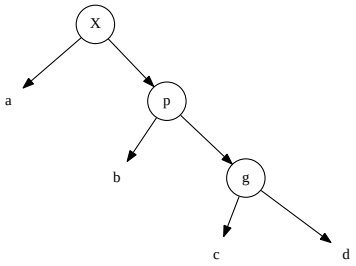
- *Zig-zag step.* As shown in figure 7.14, in this case, X and P are children on different sides. X is on the left, P is on the right. Or X is on the right, P is on the left. After rotation, X becomes the new root, P and G are siblings.

- *Zig step.* As shown in figure 7.15, in this case, P is the root, we rotate the tree, so that X becomes new root. This is the last step in splay operation.

Although there are 6 different cases, they can be handled in the environments support pattern matching. Denote the non-empty binary tree in form $T =$

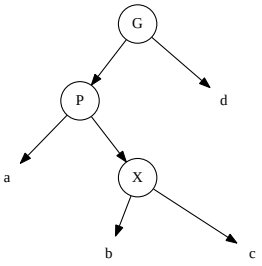


(a) X and P are both left children or both right children.

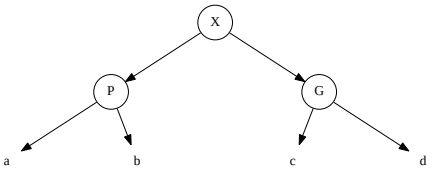


(b) X becomes new root after rotating 2 times.

Figure 7.13: Zig-zig case.

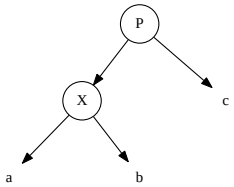


(a) X and P are children on different sides.

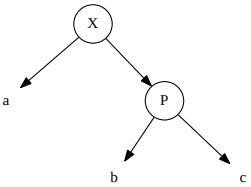


(b) X becomes new root. P and G are siblings.

Figure 7.14: Zig-zag case.



(a) P is the root.



(b) Rotate the tree to make X be new root.

Figure 7.15: Zig case.

www.dbooks.org

(L, k, R),. when access key Y in tree T, the splay operation can be defined as below.

$$splay(T, X) = \begin{cases} (a, X, (b, P, (c, G, d))) & : & T = (((a, X, b), P, c), G, d), X = Y \\ (((a, G, b), P, c), X, d) & : & T = (a, G, (b, P, (c, X, d))), X = Y \\ ((a, P, b), X, (c, G, d)) & : & T = (a, P, (b, X, c), G, d), X = Y \\ ((a, G, b), X, (c, P, d)) & : & T = (a, G, ((b, X, c), P, d)), X = Y \\ (a, X, (b, P, c)) & : & T = ((a, X, b), P, c), X = Y \\ ((a, P, b), X, c) & : & T = (a, P, (b, X, c)), X = Y \\ T & : & otherwise \end{cases} \tag{7.13}$$

The first two clauses handle the 'zig-zig' cases; the next two clauses handle the 'zig-zag' cases; the last two clauses handle the zig cases. The tree aren't changed for all other situations.

The following Haskell program implements this splay function.

```
data STree a = E -- Empty
             | Node (STree a) a (STree a) -- left, key, right

-- zig-zig
splay t@(Node (Node (Node a x b) p c) g d) y =
    if x == y then Node a x (Node b p (Node c g d)) else t
splay t@(Node a g (Node b p (Node c x d))) y =
    if x == y then Node (Node (Node a g b) p c) x d else t
-- zig-zag
splay t@(Node (Node a p (Node b x c)) g d) y =
    if x == y then Node (Node a p b) x (Node c g d) else t
splay t@(Node a g (Node (Node b x c) p d)) y =
    if x == y then Node (Node a g b) x (Node c p d) else t
-- zig
splay t@(Node (Node a x b) p c) y = if x == y then Node a x (Node b p c) else t
splay t@(Node a p (Node b x c)) y = if x == y then Node (Node a p b) x c else t
-- otherwise
splay t _ = t
```

With splay operation defined, every time when insert a new key, we call the splay function to adjust the tree. If the tree is empty, the result is a leaf; otherwise we compare this key with the root, if it is less than the root, we recursively insert it into the left child, and perform splaying after that; else the key is inserted into the right child.

$$insert(T, x) = \begin{cases} (\phi, x, \phi) & : & T = \phi \\ splay((insert(L, x), k, R), x) & : & T = (L, k, R), x < k \\ splay(L, k, insert(R, x)) & : & otherwise \end{cases} \tag{7.14}$$

The following Haskell program implements this insertion algorithm.

```
insert E y = Node E y E
insert (Node l x r) y
    | x > y     = splay (Node (insert l y) x r) y
    | otherwise = splay (Node l x (insert r y)) y
```

Figure 7.16 shows the result of using this function. It inserts the ordered elements $\{1, 2, ..., 10\}$ one by one to the empty tree. This would build a very poor result which downgrade to linked-list with normal binary search tree. The splay method creates more balanced result.

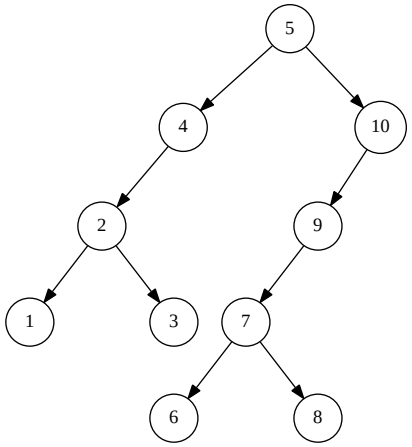


Figure 7.16: Splaying helps improving the balance.

Okasaki found a simple rule for Splaying [6]. Whenever we follow two left branches, or two right branches continuously, we rotate the two nodes.

Based on this rule, splaying can be realized in such a way. When we access node for a key x (can be during the process of inserting a node, or looking up a node, or deleting a node), if we traverse two left branches or two right branches, we partition the tree in two parts L and R, where L contains all nodes smaller than x, and R contains all the rest. We can then create a new tree (for instance in insertion), with x as the root, L as the left child, and R being the right child.

www.dbooks.org

The partition process is recursive, because it will splay its children as well.

$$partition(T, p) = \begin{cases} (\phi, \phi) & : & T = \phi \\ (T, \phi) & : & T = (L, k, R) \wedge R = \phi \\ ((L, k, L'), k', A, B) & : & \begin{array}{l} T = (L, k, (L', k', R')) \\ k < p, k' < p \\ (A, B) = partition(R', p) \end{array} \\ ((L, k, A), (B, k', R')) & : & \begin{array}{l} T = (L, K, (L', k', R')) \\ k < p \leq k' \\ (A, B) = partition(L', p) \end{array} \\ (\phi, T) & : & T = (L, k, R) \wedge L = \phi \\ (A, (L', k', (R', k, R)) & : & \begin{array}{l} T = ((L', k', R'), k, R) \\ p \leq k, p \leq k' \\ (A, B) = partition(L', p) \end{array} \\ ((L', k', A), (B, k, R)) & : & \begin{array}{l} T = ((L', k', R'), k, R) \\ k' \leq p \leq k \\ (A, B) = partition(R', p) \end{array} \end{cases} \tag{7.15}$$

Function $partition(T, p)$ takes a tree T, and a pivot p as arguments. The first clause is edge case. The partition result for empty is a pair of empty left and right trees. Otherwise, denote the tree as (L, k, R). we need compare the pivot p and the root k. If $k < p$, there are two sub-cases. one is trivial case that R is empty. According to the property of binary search tree, All elements are less than p, so the result pair is (T, ϕ); For the other case, $R = (L', k', R')$, we need further compare k' with the pivot p. If $k' < p$ is also true, we recursively partition R' with the pivot, all the elements less than p in R' is held in tree A, and the rest is in tree B. The result pair can be composed with two trees, one is $((L, k, L'), k', A)$; the other is B. If the key of the right sub tree is not less than the pivot, we recursively partition L' with the pivot to give the intermediate pair (A, B), the final pair trees can be composed with (L, k, A) and (B, k', R'). There are symmetric cases for $p \leq k$. They are handled in the last three clauses.

Translating the above algorithm into Haskell yields the following partition program.

```
partition E _ = (E, E)
partition t@(Node l x r) y
    | x < y =
        case r of
          E → (t, E)
          Node l' x' r' →
              if x' < y then
                  let (small, big) = partition r' y in
                  (Node (Node l x l') x' small, big)
              else
                  let (small, big) = partition l' y in
                  (Node l x small, Node big x' r')
```

```
  | otherwise =
      case l of
        E → (E, t)
        Node l' x' r' →
            if y < x' then
                let (small, big) = partition l' y in
                (small, Node l' x' (Node r' x r))
            else
                let (small, big) = partition r' y in
                (Node l' x' small, Node big x r)
```

Alternatively, insertion can be realized with *partition* algorithm. When insert a new element k into the splay heap T, we can first partition the heap into two trees, L and R. Where L contains all nodes smaller than k, and R contains the rest. We then construct a new node, with k as the root and L, R as the children.

$$insert(T, k) = (L, k, R), (L, R) = partition(T, k) \tag{7.16}$$

The corresponding Haskell example program is as the following.

```
insert t x = Node small x big where (small, big) = partition t x
```

Top and pop

Since splay tree is just a special binary search tree, the minimum element is stored in the left most node. We need keep traversing the left child to realize the top operation. Denote the none empty tree $T = (L, k, R)$, the $top(T)$ function can be defined as below.

$$top(T) = \begin{cases} k & : & L = \phi \\ top(L) & : & otherwise \end{cases} \tag{7.17}$$

This is exactly the $min(T)$ algorithm for binary search tree.

For pop operation, the algorithm need remove the minimum element from the tree. Whenever there are two left nodes traversed, the splaying operation should be performed.

$$pop(T) = \begin{cases} R & : & T = (\phi, k, R) \\ (R', k, R) & : & T = ((\phi, k', R'), k, R) \\ (pop(L'), k', (R', k, R)) & : & T = ((L', k', R'), k, R) \end{cases} \tag{7.18}$$

Note that the third clause performs splaying without explicitly call the *partition* function. It utilizes the property of binary search tree directly.

Both the top and pop algorithms are bound to $O(\lg n)$ time because the splay tree is balanced.

The following Haskell example programs implement the top and pop operations.

```
findMin (Node E x _) = x
findMin (Node l x _) = findMin l

deleteMin (Node E x r) = r
deleteMin (Node (Node E x' r') x r) = Node r' x r
deleteMin (Node (Node l' x' r') x r) = Node (deleteMin l') x' (Node r' x r)
```

www.dbooks.org

Merge

Merge is another basic operation for heaps as it is widely used in Graph algorithms. By using the *partition* algorithm, merge can be realized in $O(\lg n)$ time.

When merging two splay trees, for non-trivial case, we can take the root of the first tree as the new root, then partition the second tree with this new root as the pivot. After that we recursively merge the children of the first tree to the partition result. This algorithm is defined as the following.

$$merge(T_1, T_2) = \begin{cases} T_2 & : \quad T_1 = \phi \\ (merge(L, A), k, merge(R, B)) & : \quad T_1 = (L, k, R), (A, B) = partition(T_2, k) \end{cases} \tag{7.19}$$

If the first heap is empty, the result is definitely the second heap. Otherwise, denote the first splay heap as (L, k, R), we partition T_2 with k as the pivot to yield (A, B), where A contains all the elements in T_2 which are less than k, and B holds the rest. We next recursively merge A with L; and merge B with R as the new children for T_1.

Translating the definition to Haskell gives the following example program.

```
merge E t = t
merge (Node l x r) t = Node (merge l l') x (merge r r')
    where (l', r') = partition t x
```

7.4.2 Heap sort

Since the internal implementation of the Splay heap is completely transparent to the heap interface, the heap sort algorithm can be reused. It means that the heap sort algorithm is generic no matter what the underground data structure is.

7.5 Notes and short summary

In this chapter, we define binary heap more general so that as long as the heap property is maintained, all binary representation of data structures can be used to implement binary heap.

This definition doesn't limit to the popular array based binary heap, but also extends to the explicit binary heaps including Leftist heap, Skew heap and Splay heap. The array based binary heap is particularly convenient for the imperative implementation because it intensely uses random index access which can be mapped to a completely binary tree. It's hard to find directly functional counterpart in this way.

However, by using explicit binary tree, functional implementation can be achieved, most of them have $O(\lg n)$ worst case performance, and some of them even reach $O(1)$ amortize time. Okasaki in [6] shows detailed analysis of these data structures.

In this chapter, only purely functional realization for Leftist heap, Skew heap, and Splay heap are explained, they can all be realized in imperative approaches.

It's very natural to extend the concept from binary tree to k-ary (k-way) tree, which leads to other useful heaps such as Binomial heap, Fibonacci heap and pairing heap. They are introduced in the following chapters.

Exercise 7.2

- Realize the imperative Leftist heap, Skew heap, and Splay heap.

www.dbooks.org

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. The MIT Press, 2001. ISBN: 0262032937.

[2] Heap (data structure), Wikipedia. http://en.wikipedia.org/wiki/Heap_(data_structure)

[3] Heapsort, Wikipedia. http://en.wikipedia.org/wiki/Heapsort

[4] Chris Okasaki. “Purely Functional Data Structures”. Cambridge university press, (July 1, 1999), ISBN-13: 978-0521663502

[5] Sorting algorithms/Heapsort. Rosetta Code. http://rosettacode.org/wiki/Sorting_algorithms/Heapsort

[6] Leftist Tree, Wikipedia. http://en.wikipedia.org/wiki/Leftist_tree

[7] Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in Java. http://www.brpreiss.com/books/opus5/index.html

[8] Donald E. Knuth. “The Art of Computer Programming. Volume 3: Sorting and Searching.”. Addison-Wesley Professional; 2nd Edition (October 15, 1998). ISBN-13: 978-0201485417. Section 5.2.3 and 6.2.3

[9] Skew heap, Wikipedia. http://en.wikipedia.org/wiki/Skew_heap

[10] Sleator, Daniel Dominic; Jarjan, Robert Endre. “Self-adjusting heaps” SIAM Journal on Computing 15(1):52-69. doi:10.1137/0215004 ISSN 00975397 (1986)

[11] Splay tree, Wikipedia. http://en.wikipedia.org/wiki/Splay_tree

[12] Sleator, Daniel D.; Tarjan, Robert E. (1985), “Self-Adjusting Binary Search Trees”, Journal of the ACM 32(3):652 - 686, doi: 10.1145/3828.3835

[13] NIST, “binary heap”. http://xw2k.nist.gov/dads//HTML/binaryheap.html

www.dbooks.org

Chapter 8

From grape to the world cup, the evolution of selection sort

8.1 Introduction

We have introduced the 'hello world' sorting algorithm, insertion sort. In this short chapter, we explain another straightforward sorting method, selection sort. The basic version of selection sort doesn't perform as good as the divide and conqueror methods, e.g. quick sort and merge sort. We'll use the same approaches in the chapter of insertion sort, to analyze why it's slow, and try to improve it by varies of attempts till reach the best bound of comparison based sorting, $O(n \lg n)$, by evolving to heap sort.

The idea of selection sort can be illustrated by a real life story. Consider a kid eating a bunch of grapes. There are two types of children according to my observation. One is optimistic type, that the kid always eats the biggest grape he/she can ever find; the other is pessimistic, that he/she always eats the smallest one.

The first type of kids actually eat the grape in an order that the size decreases monotonically; while the other eat in a increase order. The kid *sorts* the grapes in order of size in fact, and the method used here is selection sort.

Based on this idea, the algorithm of selection sort can be directly described as the following.

In order to sort a series of elements:

- The trivial case, if the series is empty, then we are done, the result is also empty;
- Otherwise, we find the smallest element, and append it to the tail of the result;

Note that this algorithm sorts the elements in increase order; It's easy to sort in decrease order by picking the biggest element instead; We'll introduce about passing a comparator as a parameter later on.

www.dbooks.org



Figure 8.1: Always picking the smallest grape.

This description can be formalized to a equation.

$$sort(A) = \begin{cases} \phi & : \quad A = \phi \\ \{m\} \cup sort(A') & : \quad otherwise \end{cases} \tag{8.1}$$

Where m is the minimum element among collection A, and A' is all the rest elements except m:

$$m = min(A)$$
$$A' = A - \{m\}$$

We don't limit the data structure of the collection here. Typically, A is an array in imperative environment, and a list (singly linked-list particularly) in functional environment, and it can even be other data struture which will be introduced later.

The algorithm can also be given in imperative manner.

function SORT(A)
  $X \leftarrow \phi$
  while $A \neq \phi$ **do**
    $x \leftarrow$ MIN(A)
    $A \leftarrow$ DEL(A, x)
    $X \leftarrow$ APPEND(X, x)
  return X

Figure 8.2 depicts the process of this algorithm.

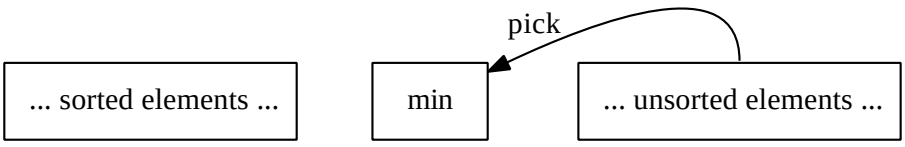


Figure 8.2: The left part is sorted data, continuously pick the minimum element in the rest and append it to the result.

We just translate the very original idea of 'eating grapes' line by line without considering any expense of time and space. This realization stores the result in

X, and when an selected element is appended to X, we delete the same element from A. This indicates that we can change it to 'in-place' sorting to reuse the spaces in A.

The idea is to store the minimum element in the first cell in A (we use term 'cell' if A is an array, and say 'node' if A is a list); then store the second minimum element in the next cell, then the third cell, ...

One solution to realize this sorting strategy is swapping. When we select the i-th minimum element, we swap it with the element in the i-th cell:

```
function SORT(A)
    for i ← 1 to |A| do
        m ← MIN(A[i...])
        EXCHANGE A[i] ↔ m
```

Denote $A = \{a_1, a_2, ..., a_n\}$. At any time, when we process the i-th element, all elements before i, as $\{a_1, a_2, ..., a_{i-1}\}$ have already been sorted. We locate the minimum element among the $\{a_i, a_{i+1}, ..., a_n\}$, and exchange it with a_i, so that the i-th cell contains the right value. The process is repeatedly executed until we arrived at the last element.

This idea can be illustrated by figure 8.3.

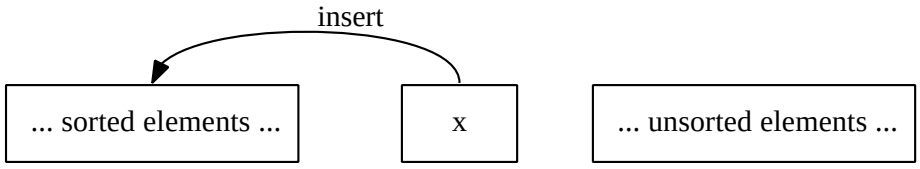


Figure 8.3: The left part is sorted data, continuously pick the minimum element in the rest and put it to the right position.

8.2 Finding the minimum

We haven't completely realized the selection sort, because we take the operation of finding the minimum (or the maximum) element as a black box. It's a puzzle how does a kid locate the biggest or the smallest grape. And this is an interesting topic for computer algorithms.

The easiest but not so fast way to find the minimum in a collection is to perform a scan. There are several ways to interpret this scan process. Consider that we want to pick the biggest grape. We start from any grape, compare it with another one, and pick the bigger one; then we take a next grape and compare it with the one we selected so far, pick the bigger one and go on the take-and-compare process, until there are not any grapes we haven't compared.

It's easy to get loss in real practice if we don't mark which grape has been compared. There are two ways to to solve this problem, which are suitable for different data-structures respectively.

www.dbooks.org

8.2.1 Labeling

Method 1 is to label each grape with a number: $\{1, 2, ..., n\}$, and we systematically perform the comparison in the order of this sequence of labels. That we first compare grape number 1 and grape number 2, pick the bigger one; then we take grape number 3, and do the comparison, ... We repeat this process until arrive at grape number n. This is quite suitable for elements stored in an array.

function MIN(A)
  $m \leftarrow A[1]$
  for $i \leftarrow 2$ to $|A|$ **do**
    if $A[i] < m$ **then**
      $m \leftarrow A[i]$
  return m

With MIN defined, we can complete the basic version of selection sort (or naive version without any optimization in terms of time and space).

However, this algorithm returns the value of the minimum element instead of its location (or the label of the grape), which needs a bit tweaking for the in-place version. Some languages such as ISO C++, support returning the reference as result, so that the swap can be achieved directly as below.

```
template<typename T>
T& min(T* from, T* to) {
    T* m;
    for (m = from++; from != to; ++from)
        if (*from < *m)
            m = from;
    return *m;
}

template<typename T>
void ssort(T* xs, int n) {
    for (int i = 0; i < n; ++i)
        std::swap(xs[i], min(xs+i, xs+n));
}
```

In environments without reference semantics, the solution is to return the location of the minimum element instead of the value:

function MIN-AT(A)
  $m \leftarrow$ FIRST-INDEX(A)
  for $i \leftarrow m + 1$ to $|A|$ **do**
    if $A[i] < A[m]$ **then**
      $m \leftarrow i$
  return m

Note that since we pass $A[i...]$ to MIN-AT as the argument, we assume the first element $A[i]$ as the smallest one, and examine all elements $A[i + 1], A[i + 2], ...$ one by one. Function FIRST-INDEX() is used to retrieve i from the input parameter.

The following Python example program, for example, completes the basic in-place selection sort algorithm based on this idea. It explicitly passes the range information to the function of finding the minimum location.

```
def ssort(xs):
```

```
    n = len(xs)
    for i in range(n):
        m = min_at(xs, i, n)
        (xs[i], xs[m]) = (xs[m], xs[i])
    return xs

def min_at(xs, i, n):
    m = i;
    for j in range(i+1, n):
        if xs[j] < xs[m]:
            m = j
    return m
```

8.2.2 Grouping

Another method is to group all grapes in two parts: the group we have examined, and the rest we haven't. We denote these two groups as A and B; All the elements (grapes) as L. At the beginning, we haven't examine any grapes at all, thus A is empty (ϕ), and B contains all grapes. We can select arbitrary two grapes from B, compare them, and put the loser (the smaller one for example) to A. After that, we repeat this process by continuously picking arbitrary grapes from B, and compare with the winner of the previous time until B becomes empty. At this time being, the final winner is the minimum element. And A turns to be $L - \{min(L)\}$, which can be used for the next time minimum finding.

There is an invariant of this method, that at any time, we have $L = A \cup \{m\} \cup B$, where m is the winner so far we hold.

This approach doesn't need the collection of grapes being indexed (as being labeled in method 1). It's suitable for any traversable data structures, including linked-list etc. Suppose b_1 is an arbitrary element in B if B isn't empty, and B' is the rest of elements with b_1 being removed, this method can be formalized as the below auxiliary function.

$$min'(A, m, B) = \begin{cases} (m, A) & : & B = \phi \\ min'(A \cup \{m\}, b_1, B') & : & b_1 < m \\ min'(A \cup \{b_1\}, m, B') & : & otherwise \end{cases} \tag{8.2}$$

In order to pick the minimum element, we call this auxiliary function by passing an empty A, and use an arbitrary element (for instance, the first one) to initialize m:

$$extractMin(L) = min'(\phi, l_1, L') \tag{8.3}$$

Where L' is all elements in L except for the first one l_1. The algorithm $extractMin$ doesn't not only find the minimum element, but also returns the updated collection which doesn't contain this minimum. Summarize this minimum extracting algorithm up to the basic selection sort definition, we can create a complete functional sorting program, for example as this Haskell code snippet.

```
sort [] = []
sort xs = x : sort xs' where
  (x, xs') = extractMin xs
```

www.dbooks.org

```
extractMin (x:xs) = min' [] x xs where
  min' ys m [] = (m, ys)
  min' ys m (x:xs) = if m < x then min' (x:ys) m xs else min' (m:ys) x xs
```

The first line handles the trivial edge case that the sorting result for empty list is obvious empty; The second clause ensures that, there is at least one element, that's why the `extractMin` function needn't other pattern-matching.

One may think the second clause of `min'` function should be written like below:

```
min' ys m (x:xs) = if m < x then min' ys ++ [x] m xs
                            else min' ys ++ [m] x xs
```

Or it will produce the updated list in reverse order. Actually, it's necessary to use 'cons' instead of appending here. This is because appending is linear operation which is proportion to the length of part A, while 'cons' is constant $O(1)$ time operation. In fact, we needn't keep the relative order of the list to be sorted, as it will be re-arranged anyway during sorting.

It's quite possible to keep the relative order during sorting[1], while ensure the performance of finding the minimum element not degrade to quadratic. The following equation defines a solution.

$$extractMin(L) = \begin{cases} (l_1, \phi) & : & |L| = 1 \\ (l_1, L') & : & l_1 < m, (m, L'') = extractMin(L') \\ (m, l_1 \cup L'') & : & otherwise \end{cases} \tag{8.4}$$

If L is a singleton, the minimum is the only element it contains. Otherwise, denote l_1 as the first element in L, and L' contains the rest elements except for l_1, that $L' = \{l_2, l_3, ...\}$. The algorithm recursively finding the minimum element in L', which yields the intermediate result as (m, L''), that m is the minimum element in L', and L'' contains all rest elements except for m. Comparing l_1 with m, we can determine which of them is the final minimum result.

The following Haskell program implements this version of selection sort.

```
sort [] = []
sort xs = x : sort xs' where
  (x, xs') = extractMin xs

extractMin [x] = (x, [])
extractMin (x:xs) = if x < m then (x, xs) else (m, x:xs') where
  (m, xs') = extractMin xs
```

Note that only 'cons' operation is used, we needn't appending at all because the algorithm actually examines the list from right to left. However, it's not free, as this program need book-keeping the context (via call stack typically). The relative order is ensured by the nature of recursion. Please refer to the appendix about tail recursion call for detailed discussion.

8.2.3 performance of the basic selection sorting

Both the labeling method, and the grouping method need examine all the elements to pick the minimum in every round; and we totally pick up the minimum

[1] known as stable sort.

element n times. Thus the performance is around $n+(n-1)+(n-2)+...+1$ times comparison, which is $\frac{n(n+1)}{2}$. Selection sort is a quadratic algorithm bound to $O(n^2)$ time.

Compare to the insertion sort, which we introduced previously, selection sort performs same in its best case, worst case and average case. While insertion sort performs well in best case (that the list has been reverse ordered, and it is stored in linked-list) as $O(n)$, and the worst performance is $O(n^2)$.

In the next sections, we'll examine, why selection sort performs poor, and try to improve it step by step.

Exercise 8.1

- Implement the basic imperative selection sort algorithm (the none in-place version) in your favorite programming language. Compare it with the in-place version, and analyze the time and space effectiveness.

8.3 Minor Improvement

8.3.1 Parameterize the comparator

Before any improvement in terms of performance, let's make the selection sort algorithm general enough to handle different sorting criteria.

We've seen two opposite examples so far, that one may need sort the elements in ascending order or descending order. For the former case, we need repeatedly finding the minimum, while for the later, we need find the maximum instead. They are just two special cases. In real world practice, one may want to sort things in varies criteria, e.g. in terms of size, weight, age, ...

One solution to handle them all is to passing the criteria as a compare function to the basic selection sort algorithms. For example:

$$sort(c, L) = \begin{cases} \phi & : \quad L = \phi \\ m \cup sort(c, L'') & : \quad otherwise, (m, L'') = extract(c, L') \end{cases} \tag{8.5}$$

And the algorithm $extract(c, L)$ is defined as below.

$$extract(c, L) = \begin{cases} (l_1, \phi) & : \quad |L| = 1 \\ (l_1, L') & : \quad c(l_1, m), (m, L'') = extract(c, L') \\ (m, \{l_1\} \cup L'') & : \quad \neg c(l_1, m) \end{cases} \tag{8.6}$$

Where c is a comparator function, it takes two elements, compare them and returns the result of which one is preceding of the other. Passing 'less than' operator ($<$) turns this algorithm to be the version we introduced in previous section.

Some environments require to pass the total ordering comparator, which returns result among 'less than', 'equal', and 'greater than'. We needn't such strong condition here, that c only tests if 'less than' is satisfied. However, as the minimum requirement, the comparator should meet the strict weak ordering as following [16]:

www.dbooks.org

- Irreflexivity, for all x, it's not the case that $x < x$;
- Asymmetric, For all x and y, if $x < y$, then it's not the case $y < x$;
- Transitivity, For all x, y, and z, if $x < y$, and $y < z$, then $x < z$;

The following Scheme/Lisp program translates this generic selection sorting algorithm. The reason why we choose Scheme/Lisp here is because the lexical scope can simplify the needs to pass the 'less than' comparator for every function calls.

```
(define (sel-sort-by ltp? lst)
  (define (ssort lst)
    (if (null? lst)
        lst
        (let ((p (extract-min lst)))
          (cons (car p) (ssort (cdr p))))))
  (define (extract-min lst)
    (if (null? (cdr lst))
        lst
        (let ((p (extract-min (cdr lst))))
          (if (ltp? (car lst) (car p))
              lst
              (cons (car p) (cons (car lst) (cdr p)))))))
  (ssort lst))
```

Note that, both `ssort` and `extract-min` are inner functions, so that the 'less than' comparator `ltp?` is available to them. Passing '<' to this function yields the normal sorting in ascending order:

```
(sel-sort-by < '(3 1 2 4 5 10 9))
;Value 16: (1 2 3 4 5 9 10)
```

It's possible to pass varies of comparator to imperative selection sort as well. This is left as an exercise to the reader.

For the sake of brevity, we only consider sorting elements in ascending order in the rest of this chapter. And we'll not pass comparator as a parameter unless it's necessary.

8.3.2 Trivial fine tune

The basic in-place imperative selection sorting algorithm iterates all elements, and picking the minimum by traversing as well. It can be written in a compact way, that we inline the minimum finding part as an inner loop.

```
procedure SORT(A)
    for i ← 1 to |A| do
        m ← i
        for j ← i + 1 to |A| do
            if A[i] < A[m] then
                m ← i
        EXCHANGE A[i] ↔ A[m]
```

Observe that, when we are sorting n elements, after the first $n-1$ minimum ones are selected, the left only one, is definitely the n-th big element, so that

we need NOT find the minimum if there is only one element in the list. This indicates that the outer loop can iterate to $n-1$ instead of n.

Another place we can fine tune, is that we needn't swap the elements if the i-th minimum one is just $A[i]$. The algorithm can be modified accordingly as below:

```
procedure SORT(A)
    for i ← 1 to |A| − 1 do
        m ← i
        for j ← i + 1 to |A| do
            if A[i] < A[m] then
                m ← i
        if m ≠ i then
            EXCHANGE A[i] ↔ A[m]
```

Definitely, these modifications won't affects the performance in terms of big-O.

8.3.3 Cock-tail sort

Knuth gave an alternative realization of selection sort in [1]. Instead of selecting the minimum each time, we can select the maximum element, and put it to the last position. This method can be illustrated by the following algorithm.

```
procedure SORT'(A)
    for i ← |A| down-to 2 do
        m ← i
        for j ← 1 to i − 1 do
            if A[m] < A[i] then
                m ← i
        EXCHANGE A[i] ↔ A[m]
```

As shown in figure 12.1, at any time, the elements on right most side are sorted. The algorithm scans all unsorted ones, and locate the maximum. Then, put it to the tail of the unsorted range by swapping.

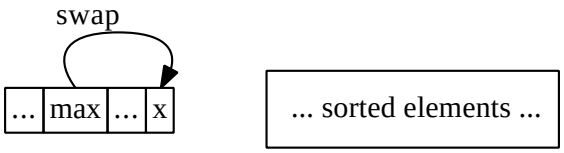


Figure 8.4: Select the maximum every time and put it to the end.

This version reveals the fact that, selecting the maximum element can sort the element in ascending order as well. What's more, we can find both the minimum and the maximum elements in one pass of traversing, putting the minimum at the first location, while putting the maximum at the last position. This approach can speed up the sorting slightly (halve the times of the outer loop). This method is called 'cock-tail sort'.

```
procedure SORT(A)
    for i ← 1 to ⌊|A|/2⌋ do
        min ← i
```

www.dbooks.org

$max \leftarrow |A| + 1 - i$
if $A[max] < A[min]$ **then**
    EXCHANGE $A[min] \leftrightarrow A[max]$
for $j \leftarrow i + 1$ to $|A| - i$ **do**
    if $A[j] < A[min]$ **then**
        $min \leftarrow j$
    if $A[max] < A[j]$ **then**
        $max \leftarrow j$
EXCHANGE $A[i] \leftrightarrow A[min]$
EXCHANGE $A[|A| + 1 - i] \leftrightarrow A[max]$

This algorithm can be illustrated as in figure 8.5, at any time, the left most and right most parts contain sorted elements so far. That the smaller sorted ones are on the left, while the bigger sorted ones are on the right. The algorithm scans the unsorted ranges, located both the minimum and the maximum positions, then put them to the head and the tail position of the unsorted ranges by swapping.

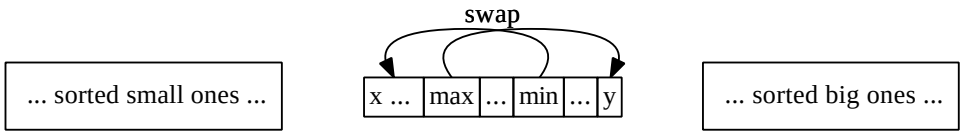


Figure 8.5: Select both the minimum and maximum in one pass, and put them to the proper positions.

Note that it's necessary to swap the left most and right most elements before the inner loop if they are not in correct order. This is because we scan the range excluding these two elements. Another method is to initialize the first element of the unsorted range as both the maximum and minimum before the inner loop. However, since we need two swapping operations after the scan, it's possible that the first swapping moves the maximum or the minimum from the position we just found, which leads the second swapping malfunctioned. How to solve this problem is left as exercise to the reader.

The following Python example program implements this cock-tail sort algorithm.

```
def cocktail_sort(xs):
    n = len(xs)
    for i in range(n / 2):
        (mi, ma) = (i, n - 1 -i)
        if xs[ma] < xs[mi]:
            (xs[mi], xs[ma]) = (xs[ma], xs[mi])
        for j in range(i+1, n - 1 - i):
            if xs[j] < xs[mi]:
                mi = j
            if xs[ma] < xs[j]:
                ma = j
        (xs[i], xs[mi]) = (xs[mi], xs[i])
        (xs[n - 1 - i], xs[ma]) = (xs[ma], xs[n - 1 - i])
    return xs
```

It's possible to realize cock-tail sort in functional approach as well. An intuitive recursive description can be given like this:

- Trivial edge case: If the list is empty, or there is only one element in the list, the sorted result is obviously the origin list;
- Otherwise, we select the minimum and the maximum, put them in the head and tail positions, then recursively sort the rest elements.

This algorithm description can be formalized by the following equation.

$$sort(L) = \begin{cases} L & : & |L| \leq 1 \\ \{l_{min}\} \cup sort(L'') \cup \{l_{max}\} & : & otherwise \end{cases} \tag{8.7}$$

Where the minimum and the maximum are extracted from L by a function $select(L)$.

$$(l_{min}, L'', l_{max}) = select(L)$$

Note that, the minimum is actually linked to the front of the recursive sort result. Its semantic is a constant $O(1)$ time 'cons' (refer to the appendix of this book for detail). While the maximum is appending to the tail. This is typically a linear $O(n)$ time expensive operation. We'll optimize it later.

Function $select(L)$ scans the whole list to find both the minimum and the maximum. It can be defined as below:

$$select(L) = \begin{cases} (min(l_1, l_2), max(l_1, l_2)) & : & L = \{l_1, l_2\} \\ (l_1, \{l_{min}\} \cup L'', l_{max}) & : & l_1 < l_{min} \\ (l_{min}, \{l_{max}\} \cup L'', l_1) & : & l_{max} < l_1 \\ (l_{min}, \{l_1\} \cup L'', l_{max}) & : & otherwise \end{cases} \tag{8.8}$$

Where $(l_{min}, L'', l_{max}) = select(L')$ and L' is the rest of the list except for the first element l_1. If there are only two elements in the list, we pick the smaller as the minimum, and the bigger as the maximum. After extract them, the list becomes empty. This is the trivial edge case; Otherwise, we take the first element l_1 out, then recursively perform selection on the rest of the list. After that, we compare if l_1 is less then the minimum or greater than the maximum candidates, so that we can finalize the result.

Note that for all the cases, there is no appending operation to form the result. However, since selection must scan all the element to determine the minimum and the maximum, it is bound to $O(n)$ linear time.

The complete example Haskell program is given as the following.

```
csort [] = []
csort [x] = [x]
csort xs = mi : csort xs' ++ [ma] where
  (mi, xs', ma) = extractMinMax xs

extractMinMax [x, y] = (min x y, [], max x y)
extractMinMax (x:xs) | x < mi = (x, mi:xs', ma)
                     | ma < x = (mi, ma:xs', x)
                     | otherwise = (mi, x:xs', ma)
  where (mi, xs', ma) = extractMinMax xs
```

www.dbooks.org

We mentioned that the appending operation is expensive in this intuitive version. It can be improved. This can be achieved in two steps. The first step is to convert the cock-tail sort into tail-recursive call. Denote the sorted small ones as A, and sorted big ones as B in figure 8.5. We use A and B as accumulators. The new cock-tail sort is defined as the following.

$$sort'(A, L, B) = \begin{cases} A \cup L \cup B & : \quad L = \phi \vee |L| = 1 \\ sort'(A \cup \{l_{min}\}, L'', \{l_{max}\} \cup B) & : \quad otherwise \end{cases} \tag{8.9}$$

Where l_{min}, l_{max} and L'' are defined as same as before. And we start sorting by passing empty A and B: $sort(L) = sort'(\phi, L, \phi)$.

Besides the edge case, observing that the appending operation only happens on $A \cup \{l_{min}\}$; while l_{max} is only linked to the head of B. This appending occurs in every recursive call. To eliminate it, we can store A in reverse order as $\overleftarrow{A}$, so that l_{max} can be 'cons' to the head instead of appending. Denote $cons(x, L) = \{x\} \cup L$ and $append(L, x) = L \cup \{x\}$, we have the below equation.

$$\begin{aligned} append(L, x) &= reverse(cons(x, reverse(L))) \\ &= reverse(cons(x, \overleftarrow{L})) \end{aligned} \tag{8.10}$$

Finally, we perform a reverse to turn $\overleftarrow{A}$ back to A. Based on this idea, the algorithm can be improved one more step as the following.

$$sort'(A, L, B) = \begin{cases} reverse(A) \cup B & : \quad L = \phi \\ reverse(\{l_1\} \cup A) \cup B & : \quad |L| = 1 \\ sort'(\{l_{min}\} \cup A, L'', \{l_{max}\} \cup B) & : \end{cases} \tag{8.11}$$

This algorithm can be implemented by Haskell as below.

```
csort' xs = cocktail [] xs [] where
  cocktail as [] bs = reverse as ++ bs
  cocktail as [x] bs = reverse (x:as) ++ bs
  cocktail as xs bs = let (mi, xs', ma) = extractMinMax xs
                      in cocktail (mi:as) xs' (ma:bs)
```

Exercise 8.2

- Realize the imperative basic selection sort algorithm, which can take a comparator as a parameter. Please try both dynamic typed language and static typed language. How to annotate the type of the comparator as general as possible in a static typed language?
- Implement Knuth's version of selection sort in your favorite programming language.
- An alternative to realize cock-tail sort is to assume the i-th element both the minimum and the maximum, after the inner loop, the minimum and maximum are found, then we can swap the the minimum to the i-th position, and the maximum to position $|A|+1-i$. Implement this solution in your favorite imperative language. Please note that there are several special edge cases should be handled correctly:

  - $A = \{max, min, ...\}$;
  - $A = \{..., max, min\}$;
  - $A = \{max, ..., min\}$.

  Please don't refer to the example source code along with this chapter before you try to solve this problem.

- Realize the function $select(L)$ by folding.

8.4 Major improvement

Although cock-tail sort halves the numbers of loop, the performance is still bound to quadratic time. It means that, the method we developed so far handles big data poorly compare to other divide and conquer sorting solutions.

To improve selection based sort essentially, we must analyze where is the bottle-neck. In order to sort the elements by comparison, we must examine all the elements for ordering. Thus the outer loop of selection sort is necessary. However, must it scan all the elements every time to select the minimum? Note that when we pick the smallest one at the first time, we actually traverse the whole collection, so that we know which ones are relative big, and which ones are relative small partially.

The problem is that, when we select the further minimum elements, instead of re-using the ordering information we obtained previously, we drop them all, and blindly start a new traverse.

So the key point to improve selection based sort is to re-use the previous result. There are several approaches, we'll adopt an intuitive idea inspired by football match in this chapter.

8.4.1 Tournament knock out

The football world cup is held every four years. There are 32 teams from different continent play the final games. Before 1982, there were 16 teams compete for the tournament finals[4].

For simplification purpose, let's go back to 1978 and imagine a way to determine the champion: In the first round, the teams are grouped into 8 pairs to play the game; After that, there will be 8 winner, and 8 teams will be out. Then in the second round, these 8 teams are grouped into 4 pairs. This time there will be 4 winners after the second round of games; Then the top 4 teams are divided into 2 pairs, so that there will be only two teams left for the final game.

The champion is determined after the total 4 rounds of games. And there are actually $8 + 4 + 2 + 1 = 16$ games. Now we have the world cup champion, however, the world cup game won't finish at this stage, we need to determine which is the silver medal team.

Readers may argue that isn't the team beaten by the champion at the final game the second best? This is true according to the real world cup rule. However, it isn't fair enough in some sense.

We often heard about the so called 'group of death', Let's suppose that Brazil team is grouped with Deutch team at the very beginning. Although both

www.dbooks.org

teams are quite strong, one of them must be knocked out. It's quite possible that even the team loss that game can beat all the other teams except for the champion. Figure 8.6 illustrates such case.

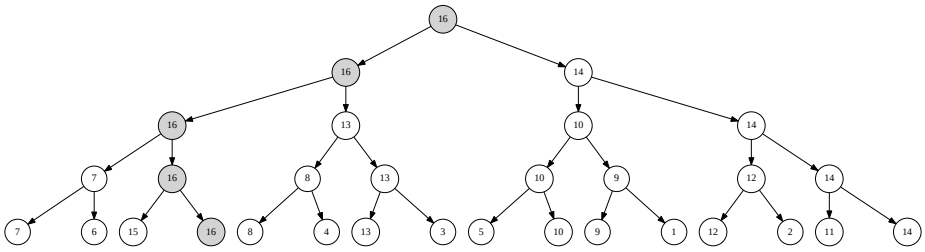


Figure 8.6: The element 15 is knocked out in the first round.

Imagine that every team has a number. The bigger the number, the stronger the team. Suppose that the stronger team always beats the team with smaller number, although this is not true in real world. But this simplification is fair enough for us to develop the tournament knock out solution. This maximum number which represents the champion is 16. Definitely, team with number 14 isn't the second best according to our rules. It should be 15, which is knocked out at the first round of comparison.

The key question here is to find an effective way to locate the second maximum number in this tournament tree. After that, what we need is to apply the same method to select the third, the fourth, ..., to accomplish the selection based sort.

One idea is to assign the champion a very small number (for instance, $-\infty$), so that it won't be selected next time, and the second best one, becomes the new champion. However, suppose there are 2^m teams for some natural number m, it still takes $2^{m-1} + 2^{m-2} + ... + 2 + 1 = 2^m$ times of comparison to determine the new champion. Which is as slow as the first time.

Actually, we needn't perform a bottom-up comparison at all since the tournament tree stores plenty of ordering information. Observe that, the second best team must be beaten by the champion at sometime, or it will be the final winner. So we can track the path from the root of the tournament tree to the leaf of the champion, examine all the teams along with this path to find the second best team.

In figure 8.6, this path is marked in gray color, the elements to be examined are $\{14, 13, 7, 15\}$. Based on this idea, we refine the algorithm like below.

1. Build a tournament tree from the elements to be sorted, so that the champion (the maximum) becomes the root;
2. Extract the root from the tree, perform a top-down pass and replace the maximum with $-\infty$;
3. Perform a bottom-up back-track along the path, determine the new champion and make it as the new root;
4. Repeat step 2 until all elements have been extracted.

Figure 8.7, 8.8, and 8.9 show the steps of applying this strategy.

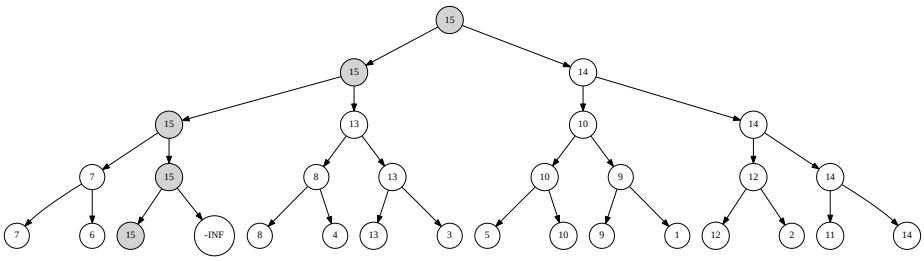


Figure 8.7: Extract 16, replace it with $-\infty$, 15 sifts up to root.

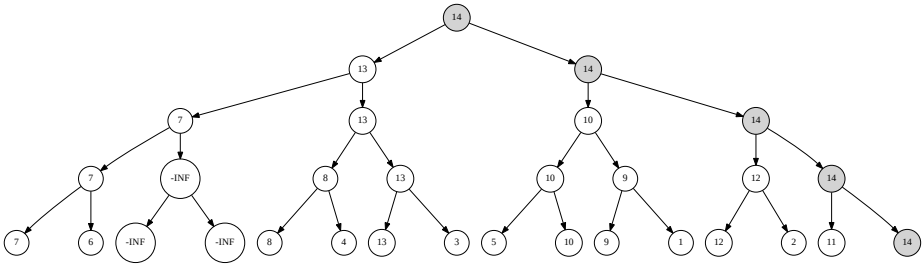


Figure 8.8: Extract 15, replace it with $-\infty$, 14 sifts up to root.

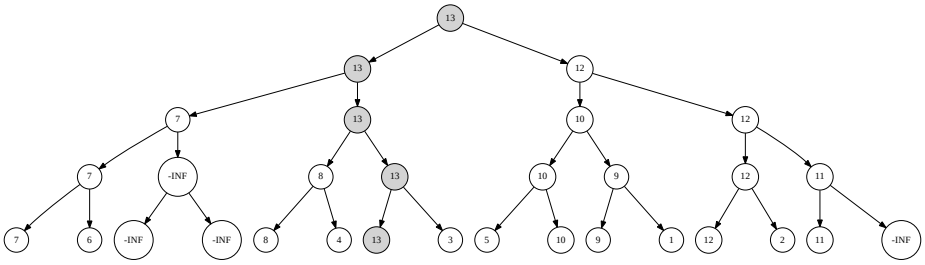


Figure 8.9: Extract 14, replace it with $-\infty$, 13 sifts up to root.

www.dbooks.org

We can reuse the binary tree definition given in the first chapter of this book to represent tournament tree. In order to back-track from leaf to the root, every node should hold a reference to its parent (concept of pointer in some environment such as ANSI C):

```
struct Node {
  Key key;
  struct Node *left, *right, *parent;
};
```

To build a tournament tree from a list of elements (suppose the number of elements are 2^m for some m), we can first wrap each element as a leaf, so that we obtain a list of binary trees. We take every two trees from this list, compare their keys, and form a new binary tree with the bigger key as the root; the two trees are set as the left and right children of this new binary tree. Repeat this operation to build a new list of trees. The height of each tree is increased by 1. Note that the size of the tree list halves after such a pass, so that we can keep reducing the list until there is only one tree left. And this tree is the finally built tournament tree.

function BUILD-TREE(A)
  $T \leftarrow \phi$
  for each $x \in A$ **do**
    $t \leftarrow$ CREATE-NODE
    KEY(t) $\leftarrow x$
    APPEND(T, t)
  while $|T| > 1$ **do**
    $T' \leftarrow \phi$
    for every $t_1, t_2 \in$ T **do**
      $t \leftarrow$ CREATE-NODE
      KEY(t) $\leftarrow$ MAX(KEY(t_1), KEY(t_2))
      LEFT(t) $\leftarrow t_1$
      RIGHT(t) $\leftarrow t_2$
      PARENT(t_1) $\leftarrow t$
      PARENT(t_2) $\leftarrow t$
      APPEND(T', t)
    $T \leftarrow T'$
  return $T[1]$

Suppose the length of the list A is n, this algorithm firstly traverses the list to build tree, which is linear to n time. Then it repeatedly compares pairs, which loops proportion to $n + \frac{n}{2} + \frac{n}{4} + ... + 2 = 2n$. So the total performance is bound to $O(n)$ time.

The following ANSI C program implements this tournament tree building algorithm.

```
struct Node* build(const Key* xs, int n) {
    int i;
    struct Node *t, **ts = (struct Node**) malloc(sizeof(struct Node*) * n);
    for (i = 0; i < n; ++i)
        ts[i] = leaf(xs[i]);
    for (; n > 1; n /= 2)
        for (i = 0; i < n; i += 2)
```

```
            ts[i/2] = branch(max(ts[i]→key, ts[i+1]→key), ts[i], ts[i+1]);
    t = ts[0];
    free(ts);
    return t;
}
```

The type of key can be defined somewhere, for example:

```
typedef int Key;
```

Function `leaf(x)` creats a leaf node, with value `x` as key, and sets all its fields, left, right and parent to `NIL`. While function `branch(key, left, right)` creates a branch node, and links the new created node as parent of its two children if they are not empty. For the sake of brevity, we skip the detail of them. They are left as exercise to the reader, and the complete program can be downloaded along with this book.

Some programming environments, such as Python provides tool to iterate every two elements at a time, for example:

```
for x, y in zip(*[iter(ts)]*2):
```

We skip such language specific feature, readers can refer to the Python example program along with this book for details.

When the maximum element is extracted from the tournament tree, we replace it with $-\infty$, and repeatedly replace all these values from the root to the leaf. Next, we back-track to root through the parent field, and determine the new maximum element.

```
function EXTRACT-MAX(T)
    m ← KEY(T)
    KEY(T) ← −∞
    while ¬ LEAF?(T) do                          ▷ The top down pass
        if KEY(LEFT(T)) = m then
            T ← LEFT(T)
        else
            T ← RIGHT(T)
        KEY(T) ← −∞
    while PARENT(T) ≠ φ do                       ▷ The bottom up pass
        T ← PARENT(T)
        KEY(T) ← MAX(KEY(LEFT(T)), KEY(RIGHT(T)))
    return m
```

This algorithm returns the extracted maximum element, and modifies the tournament tree in-place. Because we can't represent $-\infty$ in real program by limited length of word, one approach is to define a relative negative big number, which is less than all the elements in the tournament tree, for example, suppose all the elements are greater than -65535, we can define negative infinity as below:

```
#define N_INF -65535
```

We can implements this algorithm as the following ANSI C example program.

```
Key pop(struct Node* t) {
    Key x = t→key;
    t→key = N_INF;
    while (!isleaf(t)) {
```

www.dbooks.org

```
        t = t→left→key == x ? t→left : t→right;
        t→key = N_INF;
    }
    while (t→parent) {
        t = t→parent;
        t→key = max(t→left→key, t→right→key);
    }
    return x;
}
```

The behavior of EXTRACT-MAX is quite similar to the pop operation for some data structures, such as queue, and heap, thus we name it as `pop` in this code snippet.

Algorithm EXTRACT-MAX process the tree in two passes, one is top-down, then a bottom-up along the path that the 'champion team wins the world cup'. Because the tournament tree is well balanced, the length of this path, which is the height of the tree, is bound to $O(\lg n)$, where n is the number of the elements to be sorted (which are equal to the number of leaves). Thus the performance of this algorithm is $O(\lg n)$.

It's possible to realize the tournament knock out sort now. We build a tournament tree from the elements to be sorted, then continuously extract the maximum. If we want to sort in monotonically increase order, we put the first extracted one to the right most, then insert the further extracted elements one by one to left; Otherwise if we want to sort in decrease order, we can just append the extracted elements to the result. Below is the algorithm sorts elements in ascending order.

procedure SORT(A)
  $T \leftarrow$ BUILD-TREE(A)
  for $i \leftarrow |A|$ down to 1 **do**
    $A[i] \leftarrow$ EXTRACT-MAX(T)

Translating it to ANSI C example program is straightforward.

```
void tsort(Key* xs, int n) {
    struct Node* t = build(xs, n);
    while(n)
        xs[--n] = pop(t);
    release(t);
}
```

This algorithm firstly takes $O(n)$ time to build the tournament tree, then performs n pops to select the maximum elements so far left in the tree. Since each pop operation is bound to $O(\lg n)$, thus the total performance of tournament knock out sorting is $O(n \lg n)$.

Refine the tournament knock out

It's possible to design the tournament knock out algorithm in purely functional approach. And we'll see that the two passes (first top-down replace the champion with $-\infty$, then bottom-up determine the new champion) in pop operation can be combined in recursive manner, so that we needn't the parent field any more. We can re-use the functional binary tree definition as the following example Haskell code.

```
data Tr a = Empty | Br (Tr a) a (Tr a)
```

Thus a binary tree is either empty or a branch node contains a key, a left sub tree and a right sub tree. Both children are again binary trees.

We've use hard coded big negative number to represents $-\infty$. However, this solution is ad-hoc, and it forces all elements to be sorted are greater than this pre-defined magic number. Some programming environments support algebraic type, so that we can define negative infinity explicitly. For instance, the below Haskell program setups the concept of infinity [2].

```
data Infinite a = NegInf | Only a | Inf deriving (Eq, Ord)
```

From now on, we switch back to use the $min()$ function to determine the winner, so that the tournament selects the minimum instead of the maximum as the champion.

Denote function $key(T)$ returns the key of the tree rooted at T. Function $wrap(x)$ wraps the element x into a leaf node. Function $tree(l, k, r)$ creates a branch node, with k as the key, l and r as the two children respectively.

The knock out process, can be represented as comparing two trees, picking the smaller key as the new key, and setting these two trees as children:

$$branch(T_1, T_2) = tree(T_1, min(key(T_1), key(T_2)), T_2) \tag{8.12}$$

This can be implemented in Haskell word by word:

```
branch t1 t2 = Br t1 (min (key t1) (key t2)) t2
```

There is limitation in our tournament sorting algorithm so far. It only accepts collection of elements with size of 2^m, or we can't build a complete binary tree. This can be actually solved in the tree building process. Remind that we pick two trees every time, compare and pick the winner. This is perfect if there are always even number of trees. Considering a case in football match, that one team is absent for some reason (sever flight delay or whatever), so that there left one team without a challenger. One option is to make this team the winner, so that it will attend the further games. Actually, we can use the similar approach.

To build the tournament tree from a list of elements, we wrap every element into a leaf, then start the building process.

$$build(L) = build'(\{wrap(x) | x \in L\}) \tag{8.13}$$

The $build'(\mathbb{T})$ function terminates when there is only one tree left in $\mathbb{T}$, which is the champion. This is the trivial edge case. Otherwise, it groups every two trees in a pair to determine the winners. When there are odd numbers of trees, it just makes the last tree as the winner to attend the next level of tournament and recursively repeats the building process.

$$build'(\mathbb{T}) = \begin{cases} \mathbb{T} & : & |\mathbb{T}| \leq 1 \\ build'(pair(\mathbb{T})) & : & otherwise \end{cases} \tag{8.14}$$

[2] The order of the definition of 'NegInf', regular number, and 'Inf' is significant if we want to derive the default, correct comparing behavior of 'Ord'. Anyway, it's possible to specify the detailed order by make it as an instance of 'Ord'. However, this is Language specific feature which is out of the scope of this book. Please refer to other textbook about Haskell.

www.dbooks.org

Note that this algorithm actually handles another special cases, that the list to be sort is empty. The result is obviously empty.

Denote $\mathbb{T} = \{T_1, T_2, ...\}$ if there are at least two trees, and $\mathbb{T}'$ represents the left trees by removing the first two. Function $pair(\mathbb{T})$ is defined as the following.

$$pair(\mathbb{T}) = \begin{cases} \{branch(T_1, T_2)\} \cup pair(\mathbb{T}') & : & |\mathbb{T}| \geq 2 \\ \mathbb{T} & : & otherwise \end{cases} \tag{8.15}$$

The complete tournament tree building algorithm can be implemented as the below example Haskell program.

```
fromList :: (Ord a) ⇒ [a] → Tr (Infinite a)
fromList = build ∘ (map wrap) where
  build [] = Empty
  build [t] = t
  build ts = build $ pair ts
  pair (t1:t2:ts) = (branch t1 t2):pair ts
  pair ts = ts
```

When extracting the champion (the minimum) from the tournament tree, we need examine either the left child sub-tree or the right one has the same key as the root, and recursively extract on that tree until arrive at the leaf node. Denote the left sub-tree of T as L, right sub-tree as R, and K as its key. We can define this popping algorithm as the following.

$$pop(T) = \begin{cases} tree(\phi, \infty, \phi) & : & L = \phi \wedge R = \phi \\ tree(L', min(key(L'), key(R)), R) & : & K = key(L), L' = pop(L) \\ tree(L, min(key(L), key(R')), R') & : & K = key(R), R' = pop(R) \end{cases} \tag{8.16}$$

It's straightforward to translate this algorithm into example Haskell code.

```
pop (Br Empty _ Empty) = Br Empty Inf Empty
pop (Br l k r) | k == key l = let l' = pop l in Br l' (min (key l') (key r)) r
               | k == key r = let r' = pop r in Br l (min (key l) (key r')) r'
```

Note that this algorithm only removes the current champion without returning it. So it's necessary to define a function to get the champion at the root node.

$$top(T) = key(T) \tag{8.17}$$

With these functions defined, tournament knock out sorting can be formalized by using them.

$$sort(L) = sort'(build(L)) \tag{8.18}$$

Where $sort'(T)$ continuously pops the minimum element to form a result tree

$$sort'(T) = \begin{cases} \phi & : & T = \phi \vee key(T) = \infty \\ \{top(T)\} \cup sort'(pop(T)) & : & otherwise \end{cases} \tag{8.19}$$

The rest of the Haskell code is given below to complete the implementation.

```
top = only ∘ key

tsort :: (Ord a) ⇒ [a] → [a]
tsort = sort' ∘ fromList where
    sort' Empty = []
    sort' (Br _ Inf _) = []
    sort' t = (top t) : (sort' $ pop t)
```

And the auxiliary function `only`, `key`, `wrap` accomplished with explicit infinity support are list as the following.

```
only (Only x) = x
key (Br _ k _ ) = k
wrap x = Br Empty (Only x) Empty
```

Exercise 8.3

- Implement the helper function `leaf()`, `branch`, `max()` `lsleaf()`, and `release()` to complete the imperative tournament tree program.
- Implement the imperative tournament tree in a programming language support GC (garbage collection).
- Why can our tournament tree knock out sort algorithm handle duplicated elements (elements with same value)? We say a sorting algorithm stable, if it keeps the original order of elements with same value. Is the tournament tree knock out sorting stable?
- Design an imperative tournament tree knock out sort algorithm, which satisfies the following:
  - Can handle arbitrary number of elements;
  - Without using hard coded negative infinity, so that it can take elements with any value.
- Compare the tournament tree knock out sort algorithm and binary tree sort algorithm, analyze efficiency both in time and space.
- Compare the heap sort algorithm and binary tree sort algorithm, and do same analysis for them.

8.4.2 Final improvement by using heap sort

We manage improving the performance of selection based sorting to $O(n \lg n)$ by using tournament knock out. This is the limit of comparison based sort according to [1]. However, there are still rooms for improvement. After sorting, there lefts a complete binary tree with all leaves and branches hold useless infinite values. This isn't space efficient at all. Can we release the nodes when popping?

Another observation is that if there are n elements to be sorted, we actually allocate about $2n$ tree nodes. n for leaves and n for branches. Is there any better way to halve the space usage?

www.dbooks.org

The final sorting structure described in equation 8.19 can be easily uniformed to a more general one if we treat the case that the tree is empty if its root holds infinity as key:

$$sort'(T) = \begin{cases} \phi & : \quad T = \phi \\ \{top(T)\} \cup sort'(pop(T)) & : \quad otherwise \end{cases} \tag{8.20}$$

This is exactly as same as the one of heap sort we gave in previous chapter. Heap always keeps the minimum (or the maximum) on the top, and provides fast pop operation. The binary heap by implicit array encodes the tree structure in array index, so there aren't any extra spaces allocated except for the n array cells. The functional heaps, such as leftist heap and splay heap allocate n nodes as well. We'll introduce more heaps in next chapter which perform well in many aspects.

8.5 Short summary

In this chapter, we present the evolution process of selection based sort. selection sort is easy and commonly used as example to teach students about embedded looping. It has simple and straightforward structure, but the performance is quadratic. In this chapter, we do see that there exists ways to improve it not only by some fine tuning, but also fundamentally change the data structure, which leads to tournament knock out and heap sort.

Bibliography

[1] Donald E. Knuth. "The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)". Addison-Wesley Professional; 2 edition (May 4, 1998) ISBN-10: 0201896850 ISBN-13: 978-0201896855

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. "Introduction to Algorithms, Second Edition". ISBN:0262032937. The MIT Press. 2001

[3] Wikipedia. "Strict weak order". http://en.wikipedia.org/wiki/Strict_weak_order

[4] Wikipedia. "FIFA world cup". http://en.wikipedia.org/wiki/FIFA_World_Cup

www.dbooks.org

Chapter 9

Binomial heap, Fibonacci heap, and pairing heap

9.1 Introduction

In previous chapter, we mentioned that heaps can be generalized and implemented with varies of data structures. However, only binary heaps are focused so far no matter by explicit binary trees or implicit array.

It's quite natural to extend the binary tree to K-ary [1] tree. In this chapter, we first show Binomial heaps which is actually consist of forest of K-ary trees. Binomial heaps gain the performance for all operations to $O(\lg n)$, as well as keeping the finding minimum element to $O(1)$ time.

If we delay some operations in Binomial heaps by using lazy strategy, it turns to be Fibonacci heap.

All binary heaps we have shown perform no less than $O(\lg n)$ time for merging, we'll show it's possible to improve it to $O(1)$ with Fibonacci heap, which is quite helpful to graph algorithms. Actually, Fibonacci heap achieves almost all operations to good amortized time bound as $O(1)$, and left the heap pop to $O(\lg n)$.

Finally, we'll introduce about the pairing heaps. It has the best performance in practice although the proof of it is still a conjecture for the time being.

9.2 Binomial Heaps

9.2.1 Definition

Binomial heap is more complex than most of the binary heaps. However, it has excellent merge performance which bound to $O(\lg n)$ time. A binomial heap is consist of a list of binomial trees.

Binomial tree

In order to explain why the name of the tree is 'binomial', let's review the famous Pascal's triangle (Also know as the Jia Xian's triangle to memorize the Chinese methematican Jia Xian (1010-1070).) [4].

www.dbooks.org

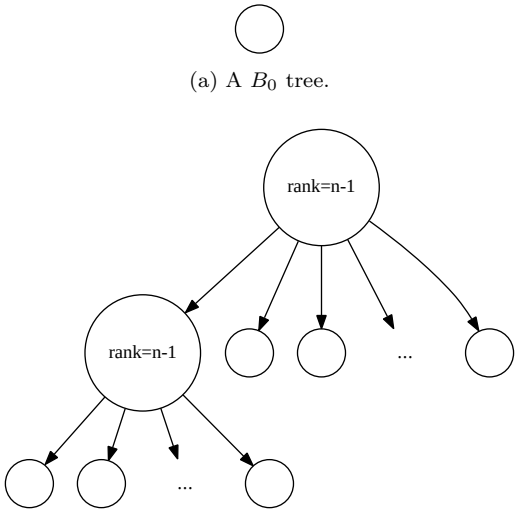
```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
...
```

In each row, the numbers are all binomial coefficients. There are many ways to gain a series of binomial coefficient numbers. One of them is by using recursive composition. Binomial trees, as well, can be defined in this way as the following.

- A binomial tree of rank 0 has only a node as the root;
- A binomial tree of rank n is consist of two rank $n-1$ binomial trees, Among these 2 sub trees, the one has the bigger root element is linked as the leftmost child of the other.

We denote a binomial tree of rank 0 as B_0, and the binomial tree of rank n as B_n.

Figure 9.1 shows a B_0 tree and how to link 2 B_{n-1} trees to a B_n tree.



(a) A B_0 tree.

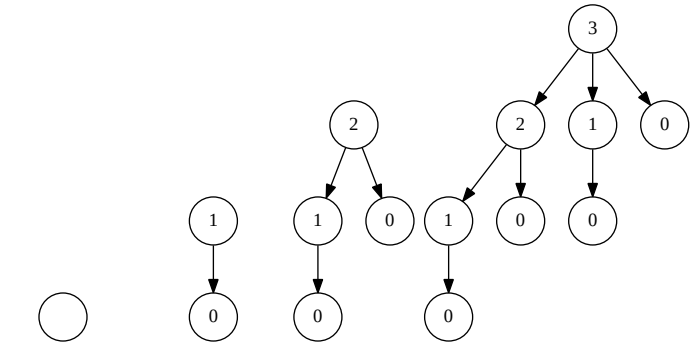
(b) Linking 2 B_{n-1} trees yields a B_n tree.

Figure 9.1: Recursive definition of binomial trees

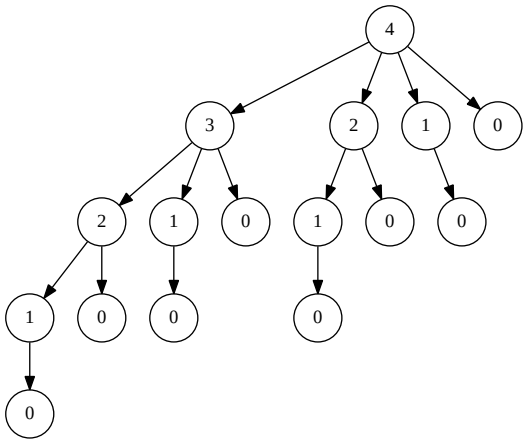
With this recursive definition, it easy to draw the form of binomial trees of rank 0, 1, 2, ..., as shown in figure 9.2

Observing the binomial trees reveals some interesting properties. For each rank n binomial tree, if counting the number of nodes in each row, it can be found that it is the binomial number.

For instance for rank 4 binomial tree, there is 1 node as the root; and in the second level next to root, there are 4 nodes; and in 3rd level, there are 6 nodes; and in 4-th level, there are 4 nodes; and the 5-th level, there is 1 node. They



(a) B_0 tree; (b) B_1 tree; (c) B_2 tree; (d) B_3 tree;



...

(e) B_4 tree;

Figure 9.2: Forms of binomial trees with rank = 0, 1, 2, 3, 4, ...

www.dbooks.org

are exactly 1, 4, 6, 4, 1 which is the 5th row in Pascal's triangle. That's why we call it binomial tree.

Another interesting property is that the total number of node for a binomial tree with rank n is 2^n. This can be proved either by binomial theory or the recursive definition directly.

Binomial heap

With binomial tree defined, we can introduce the definition of binomial heap. A binomial heap is a set of binomial trees (or a forest of binomial trees) that satisfied the following properties.

- Each binomial tree in the heap conforms to *heap property*, that the key of a node is equal or greater than the key of its parent. Here the heap is actually min-heap, for max-heap, it changes to 'equal or less than'. In this chapter, we only discuss about min-heap, and max-heap can be equally applied by changing the comparison condition.

- There is at most one binomial tree which has the rank r. In other words, there are no two binomial trees have the same rank.

This definition leads to an important result that for a binomial heap contains n elements, and if convert n to binary format yields $a_0, a_1, a_2, ..., a_m$, where a_0 is the LSB and a_m is the MSB, then for each $0 \leq i \leq m$, if $a_i = 0$, there is no binomial tree of rank i and if $a_i = 1$, there must be a binomial tree of rank i.

For example, if a binomial heap contains 5 element, as 5 is '(LSB)101(MSB)', then there are 2 binomial trees in this heap, one tree has rank 0, the other has rank 2.

Figure 9.3 shows a binomial heap which have 19 nodes, as 19 is '(LSB)11001(MSB)' in binary format, so there is a B_0 tree, a B_1 tree and a B_4 tree.

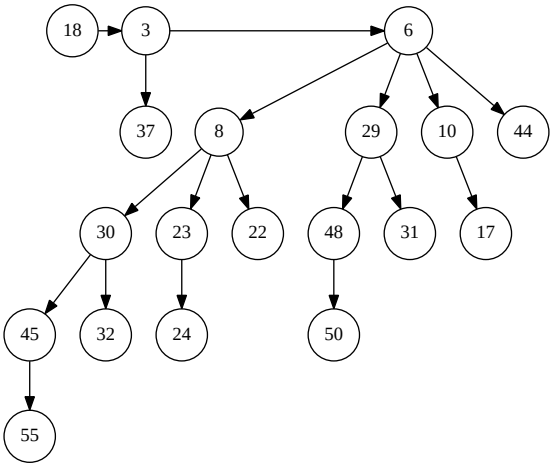


Figure 9.3: A binomial heap with 19 elements

Data layout

There are two ways to define K-ary trees imperatively. One is by using 'left-child, right-sibling' approach[2]. It is compatible with the typical binary tree structure. For each node, it has two fields, left field and right field. We use the left field to point to the first child of this node, and use the right field to point to the sibling node of this node. All siblings are represented as a single directional linked list. Figure 9.4 shows an example tree represented in this way.

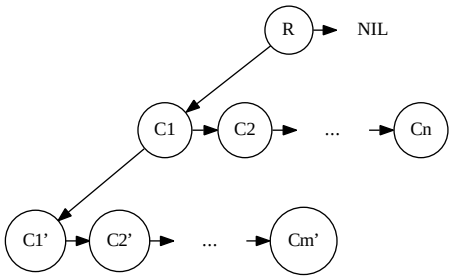


Figure 9.4: Example tree represented in 'left-child, right-sibling' way. R is the root node, it has no sibling, so it right side is pointed to NIL. $C_1, C_2, ..., C_n$ are children of R. C_1 is linked from the left side of R, other siblings of C_1 are linked one next to each other on the right side of C_1. $C'_2, ..., C'_m$ are children of C_1.

The other way is to use the library defined collection container, such as array or list to represent all children of a node.

Since the rank of a tree plays very important role, we also defined it as a field.

For 'left-child, right-sibling' method, we defined the binomial tree as the following.[1]

```
class BinomialTree:
    def __init__(self, x = None):
        self.rank = 0
        self.key = x
        self.parent = None
        self.child = None
        self.sibling = None
```

When initialize a tree with a key, we create a leaf node, set its rank as zero and all other fields are set as NIL.

It quite nature to utilize pre-defined list to represent multiple children as below.

```
class BinomialTree:
    def __init__(self, x = None):
        self.rank = 0
        self.key = x
        self.parent = None
        self.children = []
```

[1] C programs are also provided along with this book.

www.dbooks.org

For purely functional settings, such as in Haskell language, binomial tree are defined as the following.

```
data BiTree a = Node { rank :: Int
                     , root :: a
                     , children :: [BiTree a]}
```

While binomial heap are defined as a list of binomial trees (a forest) with ranks in monotonically increase order. And as another implicit constraint, there are no two binomial trees have the same rank.

```
type BiHeap a = [BiTree a]
```

9.2.2 Basic heap operations

Linking trees

Before dive into the basic heap operations such as pop and insert, We'll first realize how to link two binomial trees with same rank into a bigger one. According to the definition of binomial tree and heap property that the root always contains the minimum key, we firstly compare the two root values, select the smaller one as the new root, and insert the other tree as the first child in front of all other children. Suppose function $Key(T)$, $Children(T)$, and $Rank(T)$ access the key, children and rank of a binomial tree respectively.

$$link(T_1, T_2) = \begin{cases} node(r+1, x, \{T_2\} \cup C_1) & : \quad x < y \\ node(r+1, y, \{T_1\} \cup C_2) & : \quad otherwise \end{cases} \tag{9.1}$$

Where

$$\begin{aligned} x &= Key(T_1) \\ y &= Key(T_2) \\ r &= Rank(T_1) = Rank(T_2) \\ C_1 &= Children(T_1) \\ C_2 &= Children(T_2) \end{aligned}$$

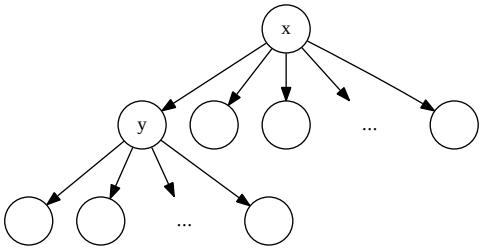


Figure 9.5: Suppose $x < y$, insert y as the first child of x.

Note that the link operation is bound to $O(1)$ time if the $\cup$ is a constant time operation. It's easy to translate the link function to Haskell program as the following.

```
link t1@(Node r x c1) t2@(Node _ y c2) =
    if x<y then Node (r+1) x (t2:c1)
    else Node (r+1) y (t1:c2)
```

It's possible to realize the link operation in imperative way. If we use 'left child, right sibling' approach, we just link the tree which has the bigger key to the left side of the other's key, and link the children of it to the right side as sibling. Figure 9.6 shows the result of one case.

1: **function** LINK(T_1, T_2)
2: **if** KEY(T_2) < KEY(T_1) **then**
3: Exchange $T_1 \leftrightarrow T_2$
4: SIBLING(T_2) $\leftarrow$ CHILD(T_1)
5: CHILD(T_1) $\leftarrow T_2$
6: PARENT(T_2) $\leftarrow T_1$
7: RANK(T_1) $\leftarrow$ RANK(T_1) + 1
8: **return** T_1

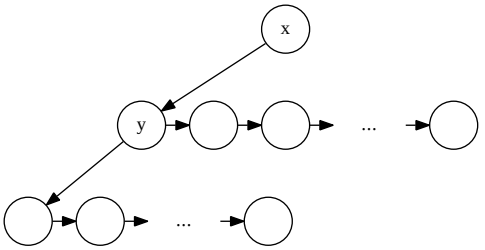


Figure 9.6: Suppose $x < y$, link y to the left side of x and link the original children of x to the right side of y.

And if we use a container to manage all children of a node, the algorithm is like below.

1: **function** LINK'(T_1, T_2)
2: **if** KEY(T_2) < KEY(T_1) **then**
3: Exchange $T_1 \leftrightarrow T_2$
4: PARENT(T_2) $\leftarrow T_1$
5: INSERT-BEFORE(CHILDREN(T_1), T_2)
6: RANK(T_1) $\leftarrow$ RANK(T_1) + 1
7: **return** T_1

It's easy to translate both algorithms to real program. Here we only show the Python program of LINK' for illustration purpose [2].

```
def link(t1, t2):
    if t2.key < t1.key:
        (t1, t2) = (t2, t1)
    t2.parent = t1
    t1.children.insert(0, t2)
    t1.rank = t1.rank + 1
    return t1
```

Exercise 9.1

Implement the tree-linking program in your favorite language with left-child, right-sibling method.

[2]The C and C++ programs are also available along with this book

www.dbooks.org

We mentioned linking is a constant time algorithm and it is true when using left-child, right-sibling approach, However, if we use container to manage the children, the performance depends on the concrete implementation of the container. If it is plain array, the linking time will be proportion to the number of children. In this chapter, we assume the time is constant. This is true if the container is implemented in linked-list.

Insert a new element to the heap (push)

As the rank of binomial trees in a forest is monotonically increasing, by using the *link* function defined above, it's possible to define an auxiliary function, so that we can insert a new tree, with rank no bigger than the smallest one, to the heap which is a forest actually.

Denote the non-empty heap as $H = \{T_1, T_2, ..., T_n\}$, we define

$$insertT(H, T) = \begin{cases} \{T\} & : & H = \phi \\ \{T\} \cup H & : & Rank(T) < Rank(T_1) \\ insertT(H', link(T, T_1)) & : & otherwise \end{cases} \tag{9.2}$$

where

$$H' = \{T_2, T_3, ..., T_n\}$$

The idea is that for the empty heap, we set the new tree as the only element to create a singleton forest; otherwise, we compare the ranks of the new tree and the first tree in the forest, if they are same, we link them together, and recursively insert the linked result (a tree with rank increased by one) to the rest of the forest; If they are not same, since the pre-condition constraints the rank of the new tree, it must be the smallest, we put this new tree in front of all the other trees in the forest.

From the binomial properties mentioned above, there are at most $O(\lg n)$ binomial trees in the forest, where n is the total number of nodes. Thus function $insertT$ performs at most $O(\lg n)$ times linking, which are all constant time operation. So the performance of $insertT$ is $O(\lg n)$. [3]

The relative Haskell program is given as below.

```
insertTree [] t = [t]
insertTree ts@(t':ts') t = if rank t < rank t' then t:ts
                           else insertTree ts' (link t t')
```

With this auxiliary function, it's easy to realize the insertion. We can wrap the new element to be inserted as the only leaf of a tree, then insert this tree to the binomial heap.

$$insert(H, x) = insertT(H, node(0, x, \phi)) \tag{9.3}$$

And we can continuously build a heap from a series of elements by folding. For example the following Haskell define a helper function 'fromList'.

[3] There is interesting observation by comparing this operation with adding two binary numbers. Which will lead to topic of *numeric representation*[6].

```
fromList = foldl insert []
```

Since wrapping an element as a singleton tree takes $O(1)$ time, the real work is done in $insertT$, the performance of binomial heap insertion is bound to $O(\lg n)$.

The insertion algorithm can also be realized with imperative approach.

Algorithm 1 Insert a tree with 'left-child-right-sibling' method.

```
1: function INSERT-TREE(H, T)
2:     while H ≠ φ∧ RANK(HEAD(H)) = RANK(T) do
3:         (T_1, H) ← EXTRACT-HEAD(H)
4:         T ← LINK(T, T_1)
5:     SIBLING(T) ← H
6:     return T
```

Algorithm 1 continuously linking the first tree in a heap with the new tree to be inserted if they have the same rank. After that, it puts the linked-list of the rest trees as the sibling, and returns the new linked-list.

If using a container to manage the children of a node, the algorithm can be given in Algorithm 2.

Algorithm 2 Insert a tree with children managed by a container.

```
1: function INSERT-TREE'(H, T)
2:     while H ≠ φ∧ RANK(H[0]) = RANK(T) do
3:         T_1 ← POP(H)
4:         T ← LINK(T, T_1)
5:     HEAD-INSERT(H, T)
6:     return H
```

In this algorithm, function POP removes the first tree $T_1 = H[0]$ from the forest. And function HEAD-INSERT, insert a new tree before any other trees in the heap, so that it becomes the first element in the forest.

With either INSERT-TREE or INSERT-TREE' defined. Realize the binomial heap insertion is trivial.

Algorithm 3 Imperative insert algorithm

```
1: function INSERT(H, x)
2:     return INSERT-TREE(H, NODE(0, x, φ))
```

The following python program implement the insert algorithm by using a container to manage sub-trees. the 'left-child, right-sibling' program is left as an exercise.

```
def insert_tree(ts, t):
    while ts !=[] and t.rank == ts[0].rank:
        t = link(t, ts.pop(0))
    ts.insert(0, t)
    return ts
```

www.dbooks.org

```
def insert(h, x):
    return insert_tree(h, BinomialTree(x))
```

Exercise 9.2

Write the insertion program in your favorite imperative programming language by using the 'left-child, right-sibling' approach.

Merge two heaps

When merge two binomial heaps, we actually try to merge two forests of binomial trees. According to the definition, there can't be two trees with the same rank and the ranks are in monotonically increasing order. Our strategy is very similar to merge sort. That in every iteration, we take the first tree from each forest, compare their ranks, and pick the smaller one to the result heap; if the ranks are equal, we then perform linking to get a new tree, and recursively insert this new tree to the result of merging the rest trees.

Figure 9.7 illustrates the idea of this algorithm. This method is different from the one given in [2].

We can formalize this idea with a function. For non-empty cases, we denote the two heaps as $H_1 = \{T_1, T_2, ...\}$ and $H_2 = \{T'_1, T'_2, ...\}$. Let $H'_1 = \{T_2, T_3, ...\}$ and $H'_2 = \{T'_2, T'_3, ...\}$.

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ \{T_1\} \cup merge(H'_1, H_2) & : & Rank(T_1) < Rank(T'_1) \\ \{T'_1\} \cup merge(H_1, H'_2) & : & Rank(T_1) > Rank(T'_1) \\ insertT(merge(H'_1, H'_2), link(T_1, T'_1)) & : & otherwise \end{cases} \tag{9.4}$$

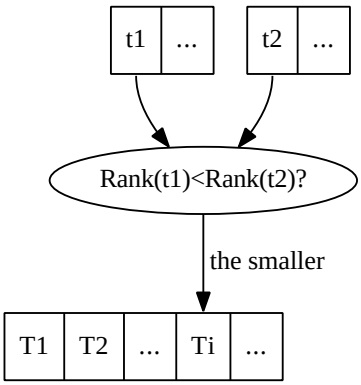
To analysis the performance of merge, suppose there are m_1 trees in H_1, and m_2 trees in H_2. There are at most $m_1 + m_2$ trees in the merged result. If there are no two trees have the same rank, the merge operation is bound to $O(m_1 + m_2)$. While if there need linking for the trees with same rank, $insertT$ performs at most $O(m_1 + m_2)$ time. Consider the fact that $m_1 = 1 + \lfloor \lg n_1 \rfloor$ and $m_2 = 1 + \lfloor \lg n_2 \rfloor$, where n_1, n_2 are the numbers of nodes in each heap, and $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor \leq 2\lfloor \lg n \rfloor$, where $n = n_1 + n_2$, is the total number of nodes. the final performance of merging is $O(\lg n)$.

Translating this algorithm to Haskell yields the following program.

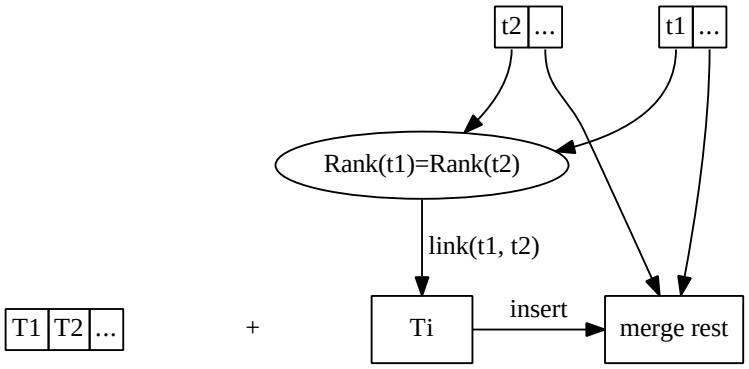
```
merge ts1 [] = ts1
merge [] ts2 = ts2
merge ts1@(t1:ts1') ts2@(t2:ts2')
    | rank t1 < rank t2 = t1:(merge ts1' ts2)
    | rank t1 > rank t2 = t2:(merge ts1 ts2')
    | otherwise = insertTree (merge ts1' ts2') (link t1 t2)
```

Merge algorithm can also be described in imperative way as shown in algorithm 4.

Since both heaps contain binomial trees with rank in monotonically increasing order. Each iteration, we pick the tree with smallest rank and append it to the result heap. If both trees have same rank we perform linking first. Consider



(a) Pick the tree with smaller rank to the result.



(b) If two trees have same rank, link them to a new tree, and recursively insert it to the merge result of the rest.

Figure 9.7: Merge two heaps.

www.dbooks.org

Algorithm 4 imperative merge two binomial heaps

```
1:  function MERGE(H_1, H_2)
2:      if H_1 = φ then
3:          return H_2
4:      if H_2 = φ then
5:          return H_1
6:      H ← φ
7:      while H_1 ≠ φ ∧ H_2 ≠ φ do
8:          T ← φ
9:          if RANK(H_1) < RANK(H_2) then
10:             (T, H_1) ← EXTRACT-HEAD(H_1)
11:         else if RANK(H_2) < RANK(H_1) then
12:             (T, H_2) ← EXTRACT-HEAD(H_2)
13:         else                                          ▷ Equal rank
14:             (T_1, H_1) ← EXTRACT-HEAD(H_1)
15:             (T_2, H_2) ← EXTRACT-HEAD(H_2)
16:             T ← LINK(T_1, T_2)
17:         APPEND-TREE(H, T)
18:     if H_1 ≠ φ then
19:         APPEND-TREES(H, H_1)
20:     if H_2 ≠ φ then
21:         APPEND-TREES(H, H_2)
22:     return H
```

the APPEND-TREE algorithm, The rank of the new tree to be appended, can't be less than any other trees in the result heap according to our merge strategy, however, it might be equal to the rank of the last tree in the result heap. This can happen if the last tree appended are the result of linking, which will increase the rank by one. In this case, we must link the new tree to be inserted with the last tree. In below algorithm, suppose function LAST(H) refers to the last tree in a heap, and APPEND(H, T) just appends a new tree at the end of a forest.

```
1: function APPEND-TREE(H, T)
2:     if H ≠ φ∧ RANK(T) = RANK(LAST(H)) then
3:         LAST(H) ← LINK(T, LAST(H))
4:     else
5:         APPEND(H, T)
```

Function APPEND-TREES repeatedly call this function, so that it can append all trees in a heap to the other heap.

```
1: function APPEND-TREES(H_1, H_2)
2:     for each T ∈ H_2 do
3:         H_1 ← APPEND-TREE(H_1, T)
```

The following Python program translates the merge algorithm.

```
def append_tree(ts, t):
    if ts != [] and ts[-1].rank == t.rank:
        ts[-1] = link(ts[-1], t)
    else:
```

```
        ts.append(t)
    return ts

def append_trees(ts1, ts2):
    return reduce(append_tree, ts2, ts1)

def merge(ts1, ts2):
    if ts1 == []:
        return ts2
    if ts2 == []:
        return ts1
    ts = []
    while ts1 != [] and ts2 != []:
        t = None
        if ts1[0].rank < ts2[0].rank:
            t = ts1.pop(0)
        elif ts2[0].rank < ts1[0].rank:
            t = ts2.pop(0)
        else:
            t = link(ts1.pop(0), ts2.pop(0))
        ts = append_tree(ts, t)
    ts = append_trees(ts, ts1)
    ts = append_trees(ts, ts2)
    return ts
```

Exercise 9.3

The program given above uses a container to manage sub-trees. Implement the merge algorithm in your favorite imperative programming language with 'left-child, right-sibling' approach.

Pop

Among the forest which forms the binomial heap, each binomial tree conforms to heap property that the root contains the minimum element in that tree. However, the order relationship of these roots can be arbitrary. To find the minimum element in the heap, we can select the smallest root of these trees. Since there are $\lg n$ binomial trees, this approach takes $O(\lg n)$ time.

However, after we locate the minimum element (which is also know as the top element of a heap), we need remove it from the heap and keep the binomial property to accomplish heap-pop operation. Suppose the forest forms the binomial heap consists trees of $B_i, B_j, ..., B_p, ..., B_m$, where B_k is a binomial tree of rank k, and the minimum element is the root of B_p. If we delete it, there will be p children left, which are all binomial trees with ranks $p-1, p-2, ..., 0$.

One tool at hand is that we have defined $O(\lg n)$ merge function. A possible approach is to reverse the p children, so that their ranks change to monotonically increasing order, and forms a binomial heap H_p. The rest of trees is still a binomial heap, we represent it as $H' = H - B_p$. Merging H_p and H' given the final result of pop. Figure 9.8 illustrates this idea.

In order to realize this algorithm, we first need to define an auxiliary function,

www.dbooks.org

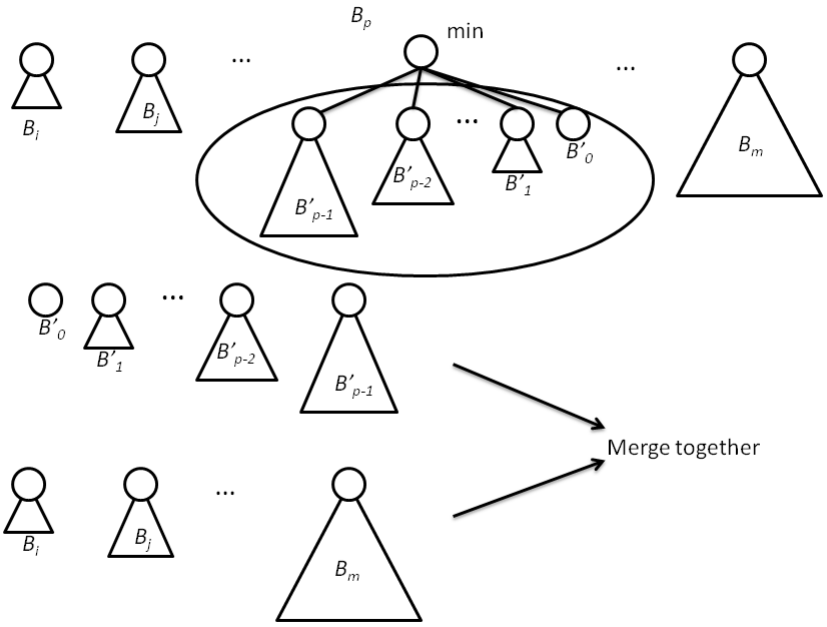


Figure 9.8: Pop the minimum element from a binomial heap.

which can extract the tree contains the minimum element at root from the forest.

$$extractMin(H) = \begin{cases} (T, \phi) & : & \text{H is a singleton as } \{T\} \\ (T_1, H') & : & Root(T_1) < Root(T') \\ (T', \{T_1\} \cup H'') & : & otherwise \end{cases} \tag{9.5}$$

where

$$\begin{array}{ll} H = \{T_1, T_2, ...\} & \text{for the non-empty forest case;} \\ H' = \{T_2, T_3, ...\} & \text{is the forest without the first tree;} \\ (T', H'') = extractMin(H') & \end{array}$$

The result of this function is a tuple. The first part is the tree which has the minimum element at root, the second part is the rest of the trees after remove the first part from the forest.

This function examine each of the trees in the forest thus is bound to $O(\lg n)$ time.

The relative Haskell program can be give respectively.

```
extractMin [t] = (t, [])
extractMin (t:ts) = if root t < root t' then (t, ts)
                    else (t', t:ts')
    where
      (t', ts') = extractMin ts
```

With this function defined, to return the minimum element is trivial.

```
findMin = root ∘ fst ∘ extractMin
```

Of course, it's possible to just traverse forest and pick the minimum root without remove the tree for this purpose. Below imperative algorithm describes it with 'left child, right sibling' approach.

```
1: function FIND-MINIMUM(H)
2:     T ← HEAD(H)
3:     min ← ∞
4:     while T ≠ φ do
5:         if KEY(T)< min then
6:             min ← KEY(T)
7:         T ← SIBLING(T)
8:     return min
```

While if we manage the children with collection containers, the link list traversing is abstracted as to find the minimum element among the list. The following Python program shows about this situation.

```
def find_min(ts):
    min_t = min(ts, key=lambda t: t.key)
    return min_t.key
```

Next we define the function to delete the minimum element from the heap by using $extractMin$.

$$delteMin(H) = merge(reverse(Children(T)), H') \tag{9.6}$$

where

$$(T, H') = extractMin(H)$$

Translate the formula to Haskell program is trivial and we'll skip it.

To realize the algorithm in procedural way takes extra efforts including list reversing etc. We left these details as exercise to the reader. The following pseudo code illustrate the imperative pop algorithm

```
1: function EXTRACT-MIN(H)
2:     (T_min, H) ← EXTRACT-MIN-TREE(H)
3:     H ← MERGE(H, REVERSE(CHILDREN(T_min)))
4:     return (KEY(T_min), H)
```

With pop operation defined, we can realize heap sort by creating a binomial heap from a series of numbers, than keep popping the smallest number from the heap till it becomes empty.

$$sort(xs) = heapSort(fromList(xs)) \tag{9.7}$$

And the real work is done in function $heapSort$.

$$heapSort(H) = \begin{cases} \phi & : \quad H = \phi \\ \{findMin(H)\} \cup heapSort(deleteMin(H)) & : \quad otherwise \end{cases} \tag{9.8}$$

Translate to Haskell yields the following program.

www.dbooks.org

```
heapSort = hsort ∘ fromList where
    hsort [] = []
    hsort h = (findMin h):(hsort $ deleteMin h)
```

Function fromList can be defined by folding. Heap sort can also be expressed in procedural way respectively. Please refer to previous chapter about binary heap for detail.

Exercise 9.4

- Write the program to return the minimum element from a binomial heap in your favorite imperative programming language with 'left-child, right-sibling' approach.
- Realize the EXTRACT-MIN-TREE() Algorithm.
- For 'left-child, right-sibling' approach, reversing all children of a tree is actually reversing a single-direct linked-list. Write a program to reverse such linked-list in your favorite imperative programming language.

More words about binomial heap

As what we have shown that insertion and merge are bound to $O(\lg n)$ time. The results are all ensure for the *worst case*. The amortized performance are $O(1)$. We skip the proof for this fact.

9.3 Fibonacci Heaps

It's interesting that why the name is given as 'Fibonacci heap'. In fact, there is no direct connection from the structure design to Fibonacci series. The inventors of 'Fibonacci heap', Michael L. Fredman and Robert E. Tarjan, utilized the property of Fibonacci series to prove the performance time bound, so they decided to use Fibonacci to name this data structure.[2]

9.3.1 Definition

Fibonacci heap is essentially a lazy evaluated binomial heap. Note that, it doesn't mean implementing binomial heap in lazy evaluation settings, for instance Haskell, brings Fibonacci heap automatically. However, lazy evaluation setting does help in realization. For example in [5], presents a elegant implementation.

Fibonacci heap has excellent performance theoretically. All operations except for pop are bound to amortized $O(1)$ time. In this section, we'll give an algorithm different from some popular textbook[2]. Most of the ideas present here are based on Okasaki's work[6].

Let's review and compare the performance of binomial heap and Fibonacci heap (more precisely, the performance goal of Fibonacci heap).

operation	Binomial heap	Fibonacci heap
insertion	$O(\lg n)$	$O(1)$
merge	$O(\lg n)$	$O(1)$
top	$O(\lg n)$	$O(1)$
pop	$O(\lg n)$	amortized $O(\lg n)$

Consider where is the bottleneck of inserting a new element x to binomial heap. We actually wrap x as a singleton leaf and insert this tree into the heap which is actually a forest.

During this operation, we inserted the tree in monotonically increasing order of rank, and once the rank is equal, recursively linking and inserting will happen, which lead to the $O(\lg n)$ time.

As the lazy strategy, we can postpone the ordered-rank insertion and merging operations. On the contrary, we just put the singleton leaf to the forest. The problem is that when we try to find the minimum element, for example the top operation, the performance will be bad, because we need check all trees in the forest, and there aren't only $O(\lg n)$ trees.

In order to locate the top element in constant time, we must remember where is the tree contains the minimum element as root.

Based on this idea, we can reuse the definition of binomial tree and give the definition of Fibonacci heap as the following Haskell program for example.

```
data BiTree a = Node { rank :: Int
                     , root :: a
                     , children :: [BiTree a]}
```

The Fibonacci heap is either empty or a forest of binomial trees with the minimum element stored in a special one explicitly.

```
data FibHeap a = E | FH { size :: Int
                        , minTree :: BiTree a
                        , trees :: [BiTree a]}
```

For convenient purpose, we also add a size field to record how many elements are there in a heap.

The data layout can also be defined in imperative way as the following ANSI C code.

```
struct node{
  Key key;
  struct node *next, *prev, *parent, *children;
  int degree; /* As known as rank */
  int mark;
};

struct FibHeap{
  struct node *roots;
  struct node *minTr;
  int n; /* number of nodes */
};
```

For generality, Key can be a customized type, we use integer for illustration purpose.

```
typedef int Key;
```

www.dbooks.org

In this chapter, we use the circular doubly linked-list for imperative settings to realize the Fibonacci Heap as described in [2]. It makes many operations easy and fast. Note that, there are two extra fields added. A *degree*, also known as *rank* for a node is the number of children of this node; Flag *mark* is used only in decreasing key operation. It will be explained in detail in later section.

9.3.2 Basic heap operations

As we mentioned that Fibonacci heap is essentially binomial heap implemented in a lazy evaluation strategy, we'll reuse many algorithms defined for binomial heap.

Insert a new element to the heap

Recall the insertion algorithm of binomial tree. It can be treated as a special case of merge operation, that one heap contains only a singleton tree.

$$insert(H, x) = merge(H, singleton(x)) \tag{9.9}$$

where singleton is an auxiliary function to wrap an element to a one-leaf-tree.

$$singleton(x) = FibHeap(1, node(1, x, \phi), \phi)$$

Note that function $FibHeap()$ accepts three parameters, a size value, which is 1 for this one-leaf-tree, a special tree which contains the minimum element as root, and a list of other binomial trees in the forest. The meaning of function $node()$ is as same as before, that it creates a binomial tree from a rank, an element, and a list of children.

Insertion can also be realized directly by appending the new node to the forest and updated the record of the tree which contains the minimum element.

1: **function** INSERT(H, k)
2: $x \leftarrow$ SINGLETON(k) ▷ Wrap x to a node
3: append x to root list of H
4: **if** $T_{min}(H) = NIL \vee k <$ KEY($T_{min}(H)$) **then**
5: $T_{min}(H) \leftarrow x$
6: N(H) $\leftarrow$ N(H)+1

Where function $T_{min}()$ returns the tree which contains the minimum element at root.

The following C source snippet is a translation for this algorithm.

```
struct FibHeap* insert_node(struct FibHeap* h, struct node* x){
  h = add_tree(h, x);
  if(h->minTr == NULL || x->key < h->minTr->key)
    h->minTr = x;
  h->n++;
  return h;
}
```

Exercise 9.5

Implement the insert algorithm in your favorite imperative programming language completely. This is also an exercise to circular doubly linked list manipulation.

Merge two heaps

Different with the merging algorithm of binomial heap, we post-pone the linking operations later. The idea is to just put all binomial trees from each heap together, and choose one special tree which record the minimum element for the result heap.

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ FibHeap(s_1 + s_2, T_{1min}, \{T_{2min}\} \cup \mathbb{T}_1 \cup \mathbb{T}_2) & : & root(T_{1min}) < root(T_{2min}) \\ FibHeap(s_1 + s_2, T_{2min}, \{T_{1min}\} \cup \mathbb{T}_1 \cup \mathbb{T}_2) & : & otherwise \end{cases} \tag{9.10}$$

where s_1 and s_2 are the size of H_1 and H_2; T_{1min} and T_{2min} are the special trees with minimum element as root in H_1 and H_2 respectively; $\mathbb{T}_1 = \{T_{11}, T_{12}, ...\}$ is a forest contains all other binomial trees in H_1; while $\mathbb{T}_2$ has the same meaning as $\mathbb{T}_1$ except that it represents the forest in H_2. Function $root(T)$ return the root element of a binomial tree.

Note that as long as the $\cup$ operation takes constant time, these *merge* algorithm is bound to $O(1)$. The following Haskell program is the translation of this algorithm.

```
merge h E = h
merge E h = h
merge h1@(FH sz1 minTr1 ts1) h2@(FH sz2 minTr2 ts2)
    | root minTr1 < root minTr2 = FH (sz1+sz2) minTr1 (minTr2:ts2++ts1)
    | otherwise = FH (sz1+sz2) minTr2 (minTr1:ts1++ts2)
```

Merge algorithm can also be realized imperatively by concatenating the root lists of the two heaps.

```
1: function MERGE(H1, H2)
2:     H ← Φ
3:     ROOT(H) ← CONCAT(ROOT(H1), ROOT(H2))
4:     if KEY(Tmin(H1)) < KEY(Tmin(H2)) then
5:         Tmin(H) ← Tmin(H1)
6:     else
7:         Tmin(H) ← Tmin(H2)
       N(H) = N(H1) + N(H2)
8:     return H
```

This function assumes neither H_1, nor H_2 is empty. And it's easy to add handling to these special cases as the following ANSI C program.

```
struct FibHeap* merge(struct FibHeap* h1, struct FibHeap* h2){
  struct FibHeap* h;
  if(is_empty(h1))
    return h2;
  if(is_empty(h2))
    return h1;
  h = empty();
```

www.dbooks.org

```
  h→roots = concat(h1→roots, h2→roots);
  if(h1→minTr→key < h2→minTr→key)
    h→minTr = h1→minTr;
  else
    h→minTr = h2→minTr;
  h→n = h1→n + h2→n;
  free(h1);
  free(h2);
  return h;
}
```

With *merge* function defined, the $O(1)$ insertion algorithm is realized as well. And we can also give the $O(1)$ time top function as below.

$$top(H) = root(T_{min}) \tag{9.11}$$

Exercise 9.6

Implement the circular doubly linked list concatenation function in your favorite imperative programming language.

Extract the minimum element from the heap (pop)

The pop operation is the most complex one in Fibonacci heap. Since we postpone the tree consolidation in merge algorithm. We have to compensate it somewhere. Pop is the only place left as we have defined, insert, merge, top already.

There is an elegant procedural algorithm to do the tree consolidation by using an auxiliary array[2]. We'll show it later in imperative approach section.

In order to realize the purely functional consolidation algorithm, let's first consider a similar number puzzle.

Given a list of numbers, such as $\{2, 1, 1, 4, 8, 1, 1, 2, 4\}$, we want to add any two values if they are same. And repeat this procedure till all numbers are unique. The result of the example list should be $\{8, 16\}$ for instance.

One solution to this problem will as the following.

$$consolidate(L) = fold(meld, \phi, L) \tag{9.12}$$

Where $fold()$ function is defined to iterate all elements from a list, applying a specified function to the intermediate result and each element. it is sometimes called as *reducing*. Please refer to Appendix A and the chapter of binary search tree for it.

$L = \{x_1, x_2, ..., x_n\}$, denotes a list of numbers; and we'll use $L' = \{x_2, x_3, ..., x_n\}$ to represent the rest of the list with the first element removed. Function $meld()$ is defined as below.

$$meld(L, x) = \begin{cases} \{x\} & : & L = \phi \\ meld(L', x + x_1) & : & x = x_1 \\ \{x\} \cup L & : & x < x_1 \\ \{x_1\} \cup meld(L', x) & : & otherwise \end{cases} \tag{9.13}$$

The $consolidate()$ function works as the follows. It maintains an ordered result list L, contains only unique numbers, which is initialized from an empty

Table 9.1: Steps of consolidate numbers

number	intermediate result	result
2	2	2
1	1, 2	1, 2
1	(1+1), 2	4
4	(4+4)	8
8	(8+8)	16
1	1, 16	1, 16
1	(1+1), 16	2, 16
2	(2+2), 16	4, 16
4	(4+4), 16	8, 16

list ϕ. Each time it process an element x, it firstly check if the first element in L is equal to x, if so, it will add them together (which yields $2x$), and repeatedly check if $2x$ is equal to the next element in L. This process won't stop until either the element to be melt is not equal to the head element in the rest of the list, or the list becomes empty. Table 9.1 illustrates the process of consolidating number sequence $\{2, 1, 1, 4, 8, 1, 1, 2, 4\}$. Column one lists the number 'scanned' one by one; Column two shows the intermediate result, typically the new scanned number is compared with the first number in result list. If they are equal, they are enclosed in a pair of parentheses; The last column is the result of meld, and it will be used as the input to next step processing.

The Haskell program can be give accordingly.

```
consolidate = foldl meld [] where
    meld [] x = [x]
    meld (x':xs) x | x == x' = meld xs (x+x')
                   | x < x'  = x:x':xs
                   | otherwise = x': meld xs x
```

We'll analyze the performance of consolidation as a part of pop operation in later section.

The tree consolidation is very similar to this algorithm except it performs based on rank. The only thing we need to do is to modify $meld()$ function a bit, so that it compare on ranks and do linking instead of adding.

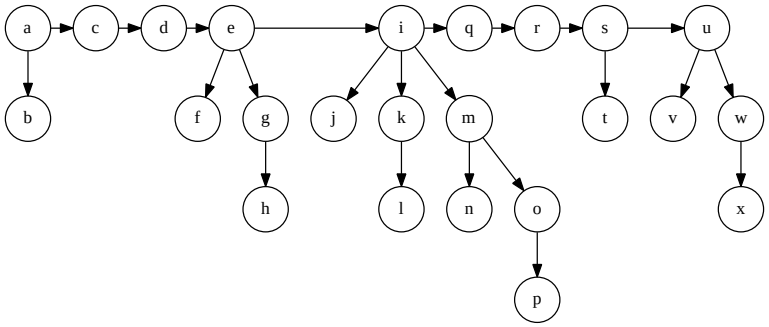
$$meld(L, x) = \begin{cases} \{x\} & : & L = \phi \\ meld(L', link(x, x_1)) & : & rank(x) = rank(x_1) \\ \{x\} \cup L & : & rank(x) < rank(x_1) \\ \{x_1\} \cup meld(L', x) & : & otherwise \end{cases} \tag{9.14}$$

The final consolidate Haskell program changes to the below version.

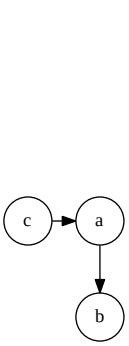
```
consolidate = foldl meld [] where
    meld [] t = [t]
    meld (t':ts) t | rank t == rank t' = meld ts (link t t')
                   | rank t <  rank t' = t:t':ts
                   | otherwise = t' : meld ts t
```

Figure 9.9 and 9.10 show the steps of consolidation when processing a Fibonacci Heap contains different ranks of trees. Comparing with table 9.1 reveals the similarity.

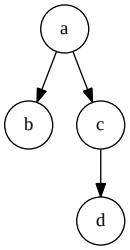
www.dbooks.org



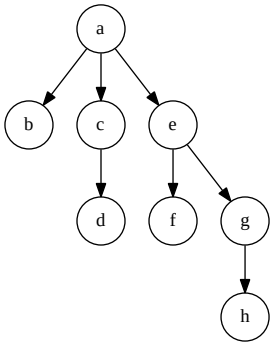
(a) Before consolidation



(b) Step 1, 2

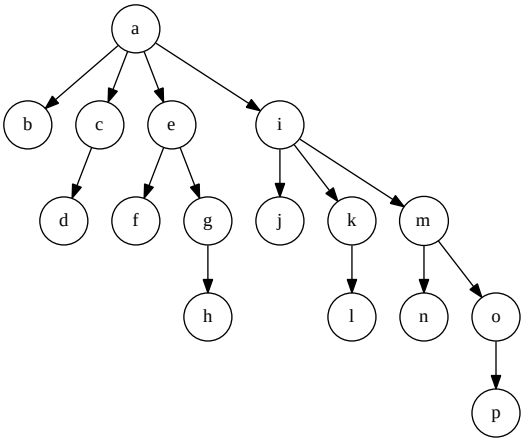


(c) Step 3, 'd' is firstly linked to 'c', then repeatedly linked to 'a'.

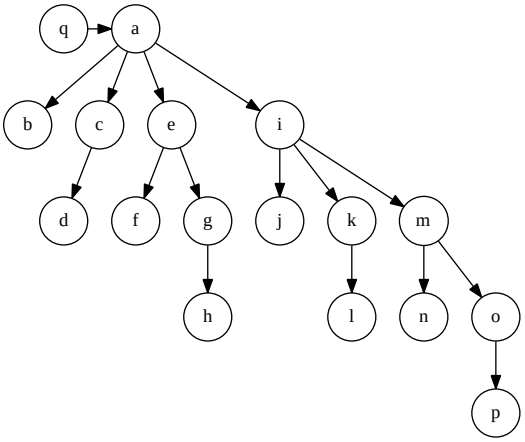


(d) Step 4

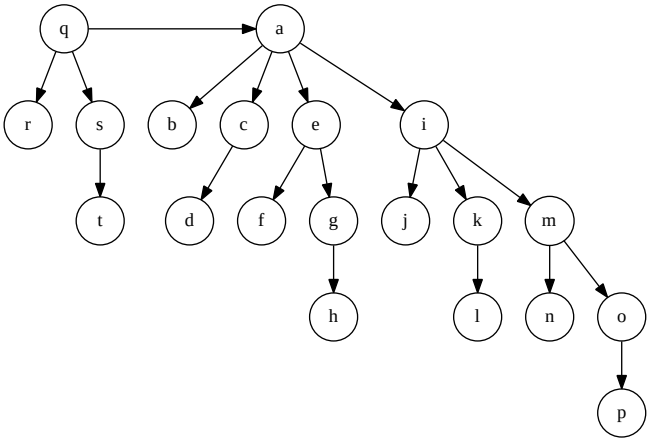
Figure 9.9: Steps of consolidation



(a) Step 5



(b) Step 6



(c) Step 7, 8, 'r' is firstly linked to 'q', then 's' is linked to 'q'.

Figure 9.10: Steps of consolidation

www.dbooks.org

After we merge all binomial trees, including the special tree record for the minimum element in root, in a Fibonacci heap, the heap becomes a Binomial heap. And we lost the special tree, which gives us the ability to return the top element in $O(1)$ time.

It's necessary to perform a $O(\lg n)$ time search to resume the special tree. We can reuse the function $extractMin()$ defined for Binomial heap.

It's time to give the final pop function for Fibonacci heap as all the sub problems have been solved. Let T_{min} denote the special tree in the heap to record the minimum element in root; $\mathbb{T}$ denote the forest contains all the other trees except for the special tree, s represents the size of a heap, and function $children()$ returns all sub trees except the root of a binomial tree.

$$deleteMin(H) = \begin{cases} \phi & : \mathbb{T} = \phi \wedge children(T_{min}) = \phi \\ FibHeap(s-1, T'_{min}, \mathbb{T}') & : otherwise \end{cases} \tag{9.15}$$

Where

$$(T'_{min}, \mathbb{T}') = extractMin(consolidate(children(T_{min}) \cup \mathbb{T}))$$

Translate to Haskell yields the below program.

```
deleteMin (FH _ (Node _ x []) []) = E
deleteMin h@(FH sz minTr ts) = FH (sz-1) minTr' ts' where
    (minTr', ts') = extractMin $ consolidate (children minTr ++ ts)
```

The main part of the imperative realization is similar. We cut all children of T_{min} and append them to root list, then perform consolidation to merge all trees with the same rank until all trees are unique in term of rank.

```
1: function DELETE-MIN(H)
2:     x ← T_min(H)
3:     if x ≠ NIL then
4:         for each y ∈ CHILDREN(x) do
5:             append y to root list of H
6:             PARENT(y) ← NIL
7:         remove x from root list of H
8:         N(H) ← N(H) - 1
9:         CONSOLIDATE(H)
10:    return x
```

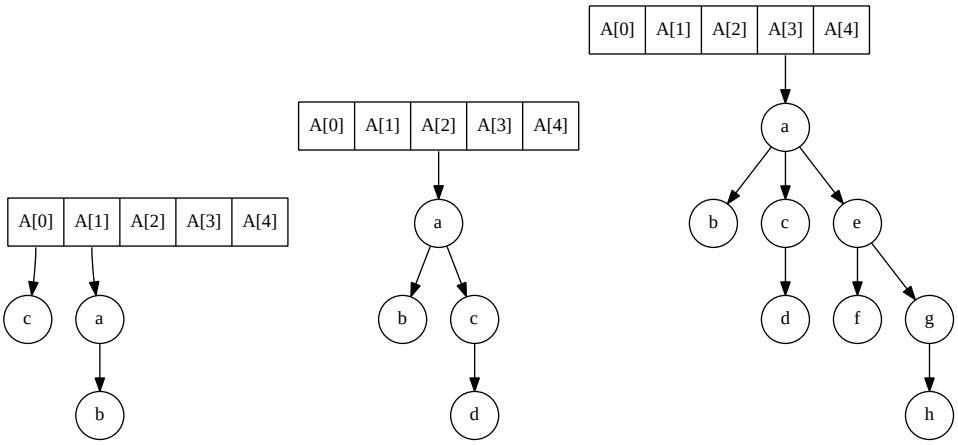
Algorithm CONSOLIDATE utilizes an auxiliary array A to do the merge job. Array $A[i]$ is defined to store the tree with rank (degree) i. During the traverse of root list, if we meet another tree of rank i, we link them together to get a new tree of rank $i+1$. Next we clean $A[i]$, and check if $A[i+1]$ is empty and perform further linking if necessary. After we finish traversing all roots, array A stores all result trees and we can re-construct the heap from it.

```
1: function CONSOLIDATE(H)
2:     D ← MAX-DEGREE(N(H))
3:     for i ← 0 to D do
4:         A[i] ← NIL
5:     for each x ∈ root list of H do
6:         remove x from root list of H
```

```
7:        d ← DEGREE(x)
8:        while A[d] ≠ NIL do
9:            y ← A[d]
10:           x ← LINK(x, y)
11:           A[d] ← NIL
12:           d ← d + 1
13:       A[d] ← x
14:   T_min(H) ← NIL                          ▷ root list is NIL at the time
15:   for i ← 0 to D do
16:       if A[i] ≠ NIL then
17:           append A[i] to root list of H.
18:           if T_min = NIL∨ KEY(A[i]) < KEY(T_min(H)) then
19:               T_min(H) ← A[i]
```

The only unclear sub algorithm is MAX-DEGREE, which can determine the upper bound of the degree of any node in a Fibonacci Heap. We'll delay the realization of it to the last sub section.

Feed a Fibonacci Heap shown in Figure 9.9 to the above algorithm, Figure 9.11, 9.12 and 9.13 show the result trees stored in auxiliary array A in every steps.



(a) Step 1, 2

(b) Step 3, Since $A_0 \neq NIL$, 'd' is firstly linked to 'c', and clear A_0 to NIL. Again, as $A_1 \neq NIL$, 'c' is linked to 'a' and the new tree is stored in A_2.

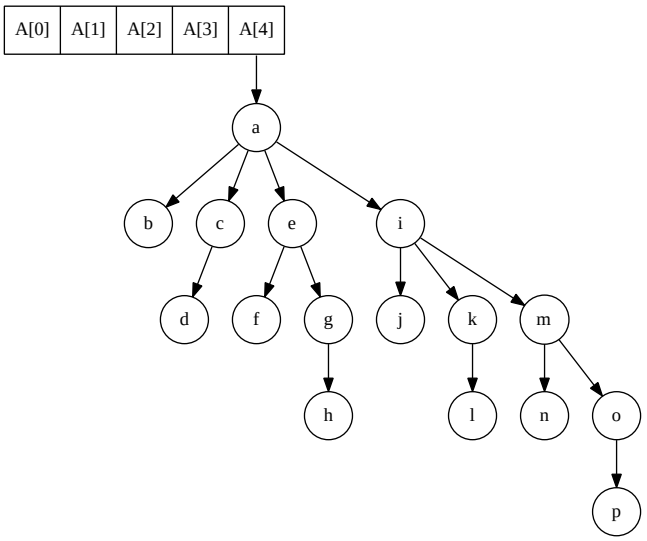
(c) Step 4

Figure 9.11: Steps of consolidation

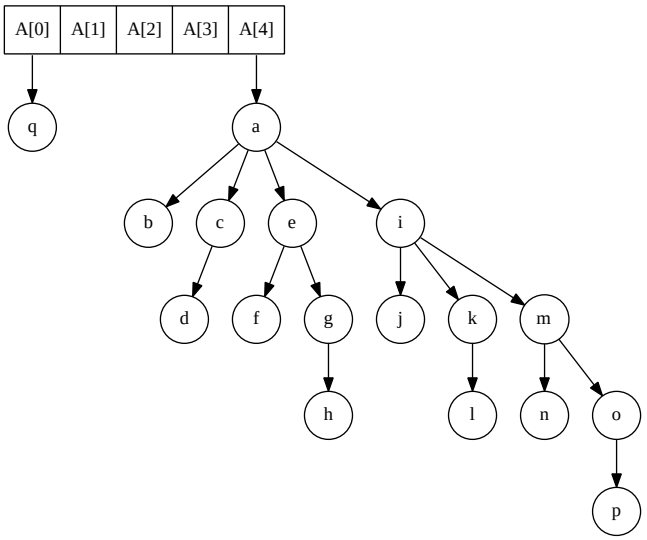
Translate the above algorithm to ANSI C yields the below program.

```
void consolidate(struct FibHeap* h){
  if(!h→roots)
    return;
  int D = max_degree(h→n)+1;
  struct node *x, *y;
  struct node** a = (struct node**)malloc(sizeof(struct node*)*(D+1));
```

www.dbooks.org

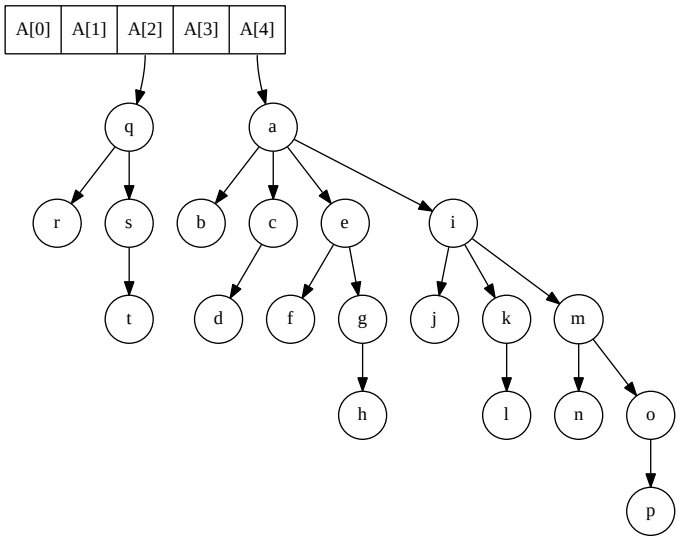


(a) Step 5



(b) Step 6

Figure 9.12: Steps of consolidation



(a) Step 7, 8, Since $A_0 \neq NIL$, 'r' is firstly linked to 'q', and the new tree is stored in A_1 (A_0 is cleared); then 's' is linked to 'q', and stored in A_2 (A_1 is cleared).

Figure 9.13: Steps of consolidation

```
  int i, d;
  for(i=0; i<=D; ++i)
    a[i] = NULL;
  while(h->roots){
    x = h->roots;
    h->roots = remove_node(h->roots, x);
    d= x->degree;
    while(a[d]){
      y = a[d];  /* Another node has the same degree as x */
      x = link(x, y);
      a[d++] = NULL;
    }
    a[d] = x;
  }
  h->minTr = h->roots = NULL;
  for(i=0; i<=D; ++i)
    if(a[i]){
      h->roots = append(h->roots, a[i]);
      if(h->minTr == NULL || a[i]->key < h->minTr->key)
        h->minTr = a[i];
    }
  free(a);
}
```

Exercise 9.7

Implement the remove function for circular doubly linked list in your favorite imperative programming language.

www.dbooks.org

9.3.3 Running time of pop

In order to analyze the amortize performance of pop, we adopt potential method. Reader can refer to [2] for a formal definition. In this chapter, we only give a intuitive illustration.

Remind the gravity potential energy, which is defined as

$$E = M \cdot g \cdot h$$

Suppose there is a complex process, which moves the object with mass M up and down, and finally the object stop at height h'. And if there exists friction resistance W_f, We say the process works the following power.

$$W = M \cdot g \cdot (h' - h) + W_f$$

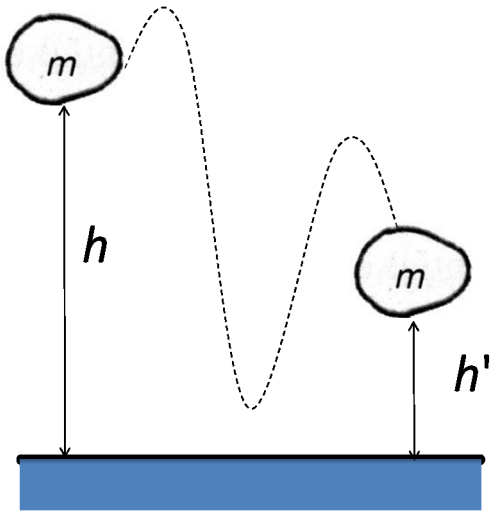


Figure 9.14: Gravity potential energy.

Figure 9.14 illustrated this concept.

We treat the Fibonacci heap pop operation in a similar way, in order to evaluate the cost, we firstly define the potential $\Phi(H)$ before extract the minimum element. This potential is accumulated by insertion and merge operations executed so far. And after tree consolidation and we get the result H', we then calculate the new potential $\Phi(H')$. The difference between $\Phi(H')$ and $\Phi(H)$ plus the contribution of consolidate algorithm indicates the amortized performance of pop.

For pop operation analysis, the potential can be defined as

$$\Phi(H) = t(H) \tag{9.16}$$

Where $t(H)$ is the number of trees in Fibonacci heap forest. We have $t(H) = 1 + length(\mathbb{T})$ for any non-empty heap.

For the n-nodes Fibonacci heap, suppose there is an upper bound of ranks for all trees as $D(n)$. After consolidation, it ensures that the number of trees in the heap forest is at most $D(n) + 1$.

Before consolidation, we actually did another important thing, which also contribute to running time, we removed the root of the minimum tree, and concatenate all children left to the forest. So consolidate operation at most processes $D(n) + t(H) - 1$ trees.

Summarize all the above factors, we deduce the amortized cost as below.

$$\begin{aligned} T &= T_{consolidation} + \Phi(H') - \Phi(H) \\ &= O(D(n) + t(H) - 1) + (D(n) + 1) - t(H) \\ &= O(D(n)) \end{aligned} \tag{9.17}$$

If only insertion, merge, and pop function are applied to Fibonacci heap. We ensure that all trees are binomial trees. It is easy to estimate the upper limit $D(n)$ is $O(\lg n)$. (Suppose the extreme case, that all nodes are in only one Binomial tree).

However, we'll show in next sub section that, there is operation can violate the binomial tree assumption.

Exercise 9.8

Why the tree consolidation time is proportion to the number of trees it processed?

9.3.4 Decreasing key

There is a special heap operation left. It only makes sense for imperative settings. It's about decreasing key of a certain node. Decreasing key plays important role in some Graphic algorithms such as Minimum Spanning tree algorithm and Dijkstra's algorithm [2]. In that case we hope the decreasing key takes $O(1)$ amortized time.

However, we can't define a function like $Decrease(H, k, k')$, which first locates a node with key k, then decrease k to k' by replacement, and then resume the heap properties. This is because the time for locating phase is bound to $O(n)$ time, since we don't have a pointer to the target node.

In imperative setting, we can define the algorithm as DECREASE-KEY(H, x, k). Here x is a node in heap H, which we want to decrease its key to k. We needn't perform a search, as we have x at hand. It's possible to give an amortized $O(1)$ solution.

When we decreased the key of a node, if it's not a root, this operation may violate the property Binomial tree that the key of parent is less than all keys of children. So we need to compare the decreased key with the parent node, and if this case happens, we can cut this node and append it to the root list. (Remind the recursive swapping solution for binary heap which leads to $O(\lg n)$)

Figure 9.15 illustrates this situation. After decreasing key of node x, it is less than y, we cut x off its parent y, and 'past' the whole tree rooted at x to root list.

Although we recover the property of that parent is less than all children, the tree isn't any longer a Binomial tree after it losses some sub tree. If a tree losses too many of its children because of cutting, we can't ensure the performance of merge-able heap operations. Fibonacci Heap adds another constraints to avoid such problem:

www.dbooks.org

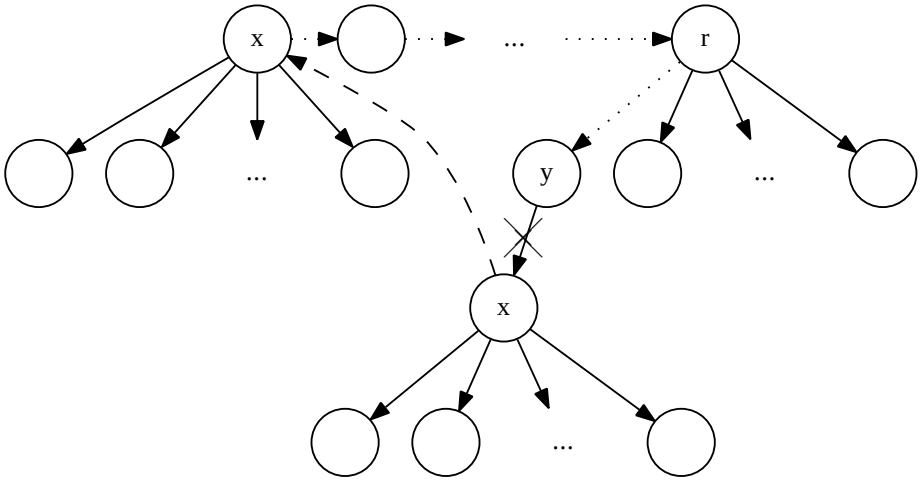


Figure 9.15: $x < y$, cut tree x from its parent, and add x to root list.

If a node losses its second child, it is immediately cut from parent, and added to root list

The final DECREASE-KEY algorithm is given as below.

```
1: function DECREASE-KEY(H, x, k)
2:     KEY(x) ← k
3:     p ← PARENT(x)
4:     if p ≠ NIL ∧ k < KEY(p) then
5:         CUT(H, x)
6:         CASCADING-CUT(H, p)
7:     if k < KEY(T_min(H)) then
8:         T_min(H) ← x
```

Where function CASCADING-CUT uses the mark to determine if the node is losing the second child. the node is marked after it losses the first child. And the mark is cleared in CUT function.

```
1: function CUT(H, x)
2:     p ← PARENT(x)
3:     remove x from p
4:     DEGREE(p) ← DEGREE(p) - 1
5:     add x to root list of H
6:     PARENT(x) ← NIL
7:     MARK(x) ← FALSE
```

During cascading cut process, if x is marked, which means it has already lost one child. We recursively performs cut and cascading cut on its parent till reach to root.

```
1: function CASCADING-CUT(H, x)
2:     p ← PARENT(x)
3:     if p ≠ NIL then
4:         if MARK(x) = FALSE then
```

```
5:          MARK(x) ← TRUE
6:      else
7:          CUT(H, x)
8:          CASCADING-CUT(H, p)
```

The relevant ANSI C decreasing key program is given as the following.

```
void decrease_key(struct FibHeap* h, struct node* x, Key k){
  struct node* p = x->parent;
  x->key = k;
  if(p && k < p->key){
    cut(h, x);
    cascading_cut(h, p);
  }
  if(k < h->minTr->key)
    h->minTr = x;
}

void cut(struct FibHeap* h, struct node* x){
  struct node* p = x->parent;
  p->children = remove_node(p->children, x);
  p->degree--;
  h->roots = append(h->roots, x);
  x->parent = NULL;
  x->mark = 0;
}

void cascading_cut(struct FibHeap* h, struct node* x){
  struct node* p = x->parent;
  if(p){
    if(!x->mark)
      x->mark = 1;
    else{
      cut(h, x);
      cascading_cut(h, p);
    }
  }
}
```

Exercise 9.9

Prove that DECREASE-KEY algorithm is amortized $O(1)$ time.

9.3.5 The name of Fibonacci Heap

It's time to reveal the reason why the data structure is named as 'Fibonacci Heap'.

There is only one undefined algorithm so far, MAX-DEGREE(n). Which can determine the upper bound of degree for any node in a n nodes Fibonacci Heap. We'll give the proof by using Fibonacci series and finally realize MAX-DEGREE algorithm.

Lemma 9.3.1. *For any node x in a Fibonacci Heap, denote $k = degree(x)$, and $|x| = size(x)$, then*

$$|x| \geq F_{k+2} \tag{9.18}$$

www.dbooks.org

Where F_k is Fibonacci series defined as the following.

$$F_k = \begin{cases} 0 & : \quad k = 0 \\ 1 & : \quad k = 1 \\ F_{k-1} + F_{k-2} & : \quad k \geq 2 \end{cases}$$

Proof. Consider all k children of node x, we denote them as $y_1, y_2, ..., y_k$ in the order of time when they were linked to x. Where y_1 is the oldest, and y_k is the youngest.

Obviously, $|y_i| \geq 0$. When we link y_i to x, children $y_1, y_2, ..., y_{i-1}$ have already been there. And algorithm LINK only links nodes with the same degree. Which indicates at that time, we have

$$degree(y_i) = degree(x) = i - 1$$

After that, node y_i can at most lost 1 child, (due to the decreasing key operation) otherwise, if it will be immediately cut off and append to root list after the second child loss. Thus we conclude

$$degree(y_i) \geq i - 2$$

For any $i = 2, 3, ..., k$.

Let s_k be the *minimum possible size* of node x, where $degree(x) = k$. For trivial cases, $s_0 = 1$, $s_1 = 2$, and we have

$$\begin{aligned} |x| &\geq s_k \\ &= 2 + \sum_{i=2}^{k} s_{degree(y_i)} \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \end{aligned}$$

We next show that $s_k > F_{k+2}$. This can be proved by induction. For trivial cases, we have $s_0 = 1 \geq F_2 = 1$, and $s_1 = 2 \geq F_3 = 2$. For induction case $k \geq 2$. We have

$$\begin{aligned} |x| &\geq s_k \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i \end{aligned}$$

At this point, we need prove that

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i \tag{9.19}$$

This can also be proved by using induction:

- Trivial case, $F_2 = 1 + F_0 = 2$
- Induction case,

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &= 1 + \sum_{i=0}^{k-1} F_i + F_k \\ &= 1 + \sum_{i=0}^{k} F_i \end{aligned}$$

Summarize all above we have the final result.

$$n \geq |x| \geq F_k + 2 \tag{9.20}$$

□

Recall the result of AVL tree, that $F_k \geq \phi^k$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. We also proved that pop operation is amortized $O(\lg n)$ algorithm.

Based on this result. We can define Function $MaxDegree$ as the following.

$$MaxDegree(n) = 1 + \lfloor \log_\phi n \rfloor \tag{9.21}$$

The imperative MAX-DEGREE algorithm can also be realized by using Fibonacci sequences.

```
1: function MAX-DEGREE(n)
2:     F_0 ← 0
3:     F_1 ← 1
4:     k ← 2
5:     repeat
6:         F_k ← F_{k_1} + F_{k_2}
7:         k ← k + 1
8:     until F_k < n
9:     return k − 2
```

Translate the algorithm to ANSI C given the following program.

```
int max_degree(int n){
  int k, F;
  int F2 = 0;
  int F1 = 1;
  for(F=F1+F2, k=2; F<n; ++k){
    F2 = F1;
    F1 = F;
    F = F1 + F2;
  }
  return k-2;
}
```

www.dbooks.org

9.4 Pairing Heaps

Although Fibonacci Heaps provide excellent performance theoretically, it is complex to realize. People find that the constant behind the big-O is big. Actually, Fibonacci Heap is more significant in theory than in practice.

In this section, we'll introduce another solution, Pairing heap, which is one of the best heaps ever known in terms of performance. Most operations including insertion, finding minimum element (top), merging are all bounds to $O(1)$ time, while deleting minimum element (pop) is conjectured to amortized $O(\lg n)$ time [7] [6]. Note that this is still a conjecture for 15 years by the time I write this chapter. Nobody has been proven it although there are much experimental data support the $O(\lg n)$ amortized result.

Besides that, pairing heap is simple. There exist both elegant imperative and functional implementations.

9.4.1 Definition

Both Binomial Heaps and Fibonacci Heaps are realized with forest. While a pairing heaps is essentially a K-ary tree. The minimum element is stored at root. All other elements are stored in sub trees.

The following Haskell program defines pairing heap.

```
data PHeap a = E | Node a [PHeap a]
```

This is a recursive definition, that a pairing heap is either empty or a K-ary tree, which is consist of a root node, and a list of sub trees.

Pairing heap can also be defined in procedural languages, for example ANSI C as below. For illustration purpose, all heaps we mentioned later are minimum-heap, and we assume the type of key is integer [4]. We use same linked-list based left-child, right-sibling approach (aka, binary tree representation[2]).

```
typedef int Key;

struct node{
  Key key;
  struct node *next, *children, *parent;
};
```

Note that the parent field does only make sense for decreasing key operation, which will be explained later on. we can omit it for the time being.

9.4.2 Basic heap operations

In this section, we first give the merging operation for pairing heap, which can be used to realize insertion. Merging, insertion, and finding the minimum element are relative trivial compare to the extracting minimum element operation.

Merge, insert, and find the minimum element (top)

The idea of merging is similar to the linking algorithm we shown previously for Binomial heap. When we merge two pairing heaps, there are two cases.

[4] We can parametrize the key type with C++ template, but this is beyond our scope, please refer to the example programs along with this book

- Trivial case, one heap is empty, we simply return the other heap as the result;
- Otherwise, we compare the root element of the two heaps, make the heap with bigger root element as a new children of the other.

Let H_1, and H_2 denote the two heaps, x and y be the root element of H_1 and H_2 respectively. Function $Children()$ returns the children of a K-ary tree. Function $Node()$ can construct a K-ary tree from a root element and a list of children.

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ Node(x, \{H_2\} \cup Children(H_1)) & : & x < y \\ Node(y, \{H_1\} \cup Children(H_2)) & : & otherwise \end{cases} \tag{9.22}$$

Where

$$x = Root(H_1)$$
$$y = Root(H_2)$$

It's obviously that merging algorithm is bound to $O(1)$ time [5]. The *merge* equation can be translated to the following Haskell program.

```
merge h E = h
merge E h = h
merge h1@(Node x hs1) h2@(Node y hs2) =
    if x < y then Node x (h2:hs1) else Node y (h1:hs2)
```

Merge can also be realized imperatively. With left-child, right sibling approach, we can just link the heap, which is in fact a K-ary tree, with larger key as the first new child of the other. This is constant time operation as described below.

```
1: function MERGE(H1, H2)
2:     if H1 = NIL then
3:         return H2
4:     if H2 = NIL then
5:         return H1
6:     if KEY(H2) < KEY(H1) then
7:         EXCHANGE(H1 ↔ H2)
8:     Insert H2 in front of CHILDREN(H1)
9:     PARENT(H2) ← H1
10:    return H1
```

Note that we also update the parent field accordingly. The ANSI C example program is given as the following.

```
struct node* merge(struct node* h1, struct node* h2) {
  if (h1 == NULL)
    return h2;
  if (h2 == NULL)
    return h1;
```

[5] Assume $\cup$ is constant time operation, this is true for linked-list settings, including 'cons' like operation in functional programming languages.

www.dbooks.org

```
  if (h2→key < h1→key)
    swap(&h1, &h2);
  h2→next = h1→children;
  h1→children = h2;
  h2→parent = h1;
  h1→next = NULL; /*Break previous link if any*/
  return h1;
}
```

Where function swap() is defined in a similar way as Fibonacci Heap.

With merge defined, insertion can be realized as same as Fibonacci Heap in Equation 9.9. Definitely it's $O(1)$ time operation. As the minimum element is always stored in root, finding it is trivial.

$$top(H) = Root(H) \tag{9.23}$$

Same as the other two above operations, it's bound to $O(1)$ time.

Decrease key of a node

There is another operation, to decrease key of a given node, which only makes sense in imperative settings as we explained in Fibonacci Heap section.

The solution is simple, that we can cut the node with the new smaller key from it's parent along with all its children. Then merge it again to the heap. The only special case is that if the given node is the root, then we can directly set the new key without doing anything else.

The following algorithm describes this procedure for a given node x, with new key k.

1: **function** DECREASE-KEY(H, x, k)
2: KEY(x) $\leftarrow k$
3: **if** PARENT(x) $\neq$ NIL **then**
4: Remove x from CHILDREN(PARENT(x)) PARENT(x) $\leftarrow$ NIL
5: **return** MERGE(H, x)
6: **return** H

The following ANSI C program translates this algorithm.

```
struct node* decrease_key(struct node* h, struct node* x, Key key) {
  x→key = key; /* Assume key ≤ x→key */
  if(x→parent) {
    x→parent→children = remove_node(x→parent→children, x);
    x→parent = NULL;
    return merge(h, x);
  }
  return h;
}
```

Exercise 9.10

Implement the program of removing a node from the children of its parent in your favorite imperative programming language. Consider how can we ensure the overall performance of decreasing key is O(1) time? Is left-child, right sibling approach enough?

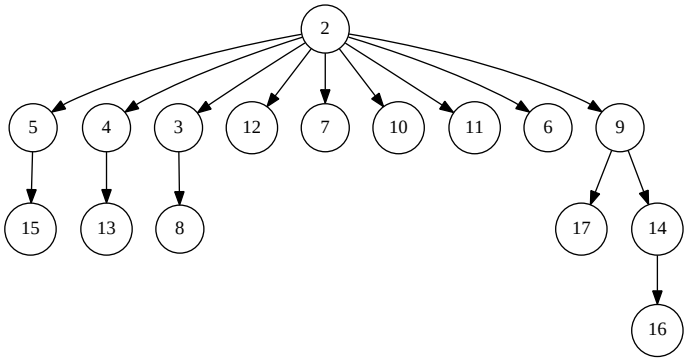
Delete the minimum element from the heap (pop)

Since the minimum element is always stored at root, after delete it during popping, the rest things left are all sub-trees. These trees can be merged to one big tree.

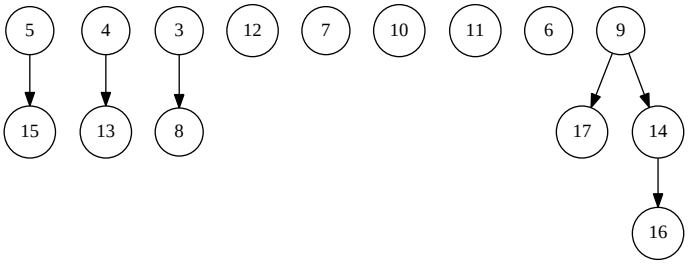
$$pop(H) = mergePairs(Children(H)) \tag{9.24}$$

Pairing Heap uses a special approach that it merges every two sub-trees from left to right in pair. Then merge these paired results from right to left which forms a final result tree. The name of 'Pairing Heap' comes from the characteristic of this pair-merging.

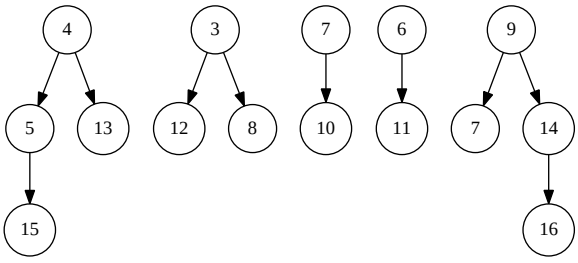
Figure 9.16 and 9.17 illustrate the procedure of pair-merging.



(a) A pairing heap before pop.



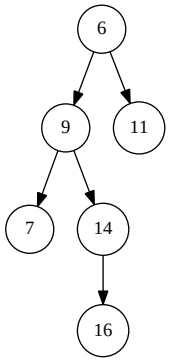
(b) After root element 2 being removed, there are 9 sub-trees left.



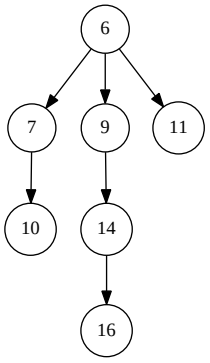
(c) Merge every two trees in pair, note that there are odd number trees, so the last one needn't merge.

Figure 9.16: Remove the root element, and merge children in pairs.

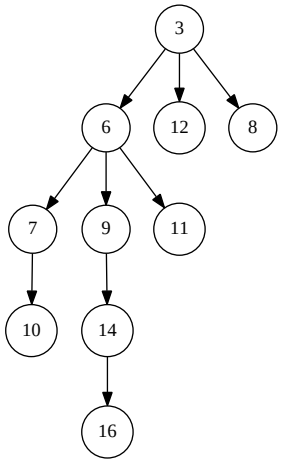
www.dbooks.org



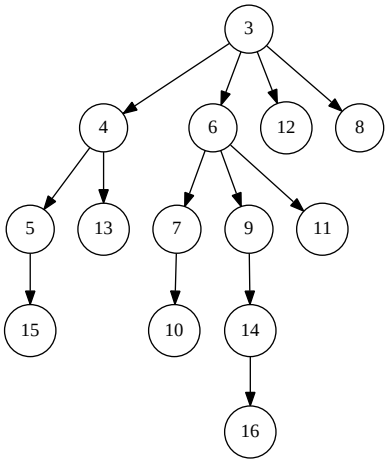
(a) Merge tree with 9, and tree with root 6.



(b) Merge tree with root 7 to the result.



(c) Merge tree with root 3 to the result.



(d) Merge tree with root 4 to the result.

Figure 9.17: Steps of merge from right to left.

The recursive pair-merging solution is quite similar to the bottom up merge sort[6]. Denote the children of a pairing heap as A, which is a list of trees of $\{T_1, T_2, T_3, ..., T_m\}$ for example. The $mergePairs()$ function can be given as below.

$$mergePairs(A) = \begin{cases} \Phi & : & A = \Phi \\ T_1 & : & A = \{T_1\} \\ merge(merge(T_1, T_2), mergePairs(A')) & : & otherwise \end{cases} \tag{9.25}$$

where

$$A' = \{T_3, T_4, ..., T_m\}$$

is the rest of the children without the first two trees.

The relative Haskell program of popping is given as the following.

```
deleteMin (Node _ hs) = mergePairs hs where
    mergePairs [] = E
    mergePairs [h] = h
    mergePairs (h1:h2:hs) = merge (merge h1 h2) (mergePairs hs)
```

The popping operation can also be explained in the following procedural algorithm.

1: **function** POP(H)
2: $L \leftarrow NIL$
3: **for** every 2 trees T_x, $T_y \in$ CHILDREN(H) from left to right **do**
4: Extract x, and y from CHILDREN(H)
5: $T \leftarrow$ MERGE(T_x, T_y)
6: Insert T at the beginning of L
7: $H \leftarrow$ CHILDREN(H) ▷ H is either NIL or one tree.
8: **for** $\forall T \in L$ from left to right **do**
9: $H \leftarrow$ MERGE(H, T)
10: **return** H

Note that L is initialized as an empty linked-list, then the algorithm iterates every two trees in pair in the children of the K-ary tree, from left to right, and performs merging, the result is inserted at the beginning of L. Because we insert to front end, so when we traverse L later on, we actually process from right to left. There may be odd number of sub-trees in H, in that case, it will leave one tree after pair-merging. We handle it by start the right to left merging from this left tree.

Below is the ANSI C program to this algorithm.

```
struct node* pop(struct node* h) {
  struct node *x, *y, *lst = NULL;
  while ((x = h->children) != NULL) {
    if ((h->children = y = x->next) != NULL)
      h->children = h->children->next;
    lst = push_front(lst, merge(x, y));
  }
  x = NULL;
  while((y = lst) != NULL) {
    lst = lst->next;
```

www.dbooks.org

```
    x = merge(x, y);
  }
  free(h);
  return x;
}
```

The pairing heap pop operation is conjectured to be amortized $O(\lg n)$ time [7].

Exercise 9.11

Write a program to insert a tree at the beginning of a linked-list in your favorite imperative programming language.

Delete a node

We didn't mention delete in Binomial heap or Fibonacci Heap. Deletion can be realized by first decreasing key to minus infinity ($-\infty$), then performing pop. In this section, we present another solution for delete node.

The algorithm is to define the function $delete(H, x)$, where x is a node in a pairing heap H [6].

If x is root, we can just perform a pop operation. Otherwise, we can cut x from H, perform a pop on x, and then merge the pop result back to H. This can be described as the following.

$$delete(H, x) = \begin{cases} pop(H) & : \quad x \quad \text{is root of} \quad H \\ merge(cut(H, x), pop(x)) & : \quad otherwise \end{cases} \tag{9.26}$$

As delete algorithm uses pop, the performance is conjectured to be amortized $O(\lg n)$ time.

Exercise 9.12

- Write procedural pseudo code for delete algorithm.
- Write the delete operation in your favorite imperative programming language
- Consider how to realize delete in purely functional setting.

9.5 Notes and short summary

In this chapter, we extend the heap implementation from binary tree to more generic approach. Binomial heap and Fibonacci heap use Forest of K-ary trees as under ground data structure, while Pairing heap use a K-ary tree to represent heap. It's a good point to post pone some expensive operation, so that the over all amortized performance is ensured. Although Fibonacci Heap gives good performance in theory, the implementation is a bit complex. It was removed in

[6] Here the semantic of x is a reference to a node.

some latest textbooks. We also present pairing heap, which is easy to realize and have good performance in practice.

The elementary tree based data structures are all introduced in this book. There are still many tree based data structures which we can't covers them all and skip here. We encourage the reader to refer to other textbooks about them. From next chapter, we'll introduce generic sequence data structures, array and queue.

www.dbooks.org

Bibliography

[1] K-ary tree, Wikipedia. http://en.wikipedia.org/wiki/K-ary_tree

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. "Introduction to Algorithms, Second Edition". The MIT Press, 2001. ISBN: 0262032937.

[3] Chris Okasaki. "Purely Functional Data Structures". Cambridge university press, (July 1, 1999), ISBN-13: 978-0521663502

[4] Wikipedia, "Pascal's triangle". http://en.wikipedia.org/wiki/Pascal's_triangle

[5] Hackage. "An alternate implementation of a priority queue based on a Fibonacci heap.", http://hackage.haskell.org/packages/archive/pqueue-mtl/1.0.7/doc/html/src/Data-Queue-FibQueue.html

[6] Chris Okasaki. "Fibonacci Heaps." http://darcs.haskell.org/nofib/gc/fibheaps/orig

[7] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. "The Pairing Heap: A New Form of Self-Adjusting Heap" Algorithmica (1986) 1: 111-129.

www.dbooks.org

Part IV

Queues and Sequences

www.dbooks.org

Chapter 10

Queue, not so simple as it was thought

10.1 Introduction

It seems that queues are relative simple. A queue provides FIFO (first-in, first-out) data manipulation support. There are many options to realize queue includes singly linked-list, doubly linked-list, circular buffer etc. However, we'll show that it's not so easy to realize queue in purely functional settings if it must satisfy abstract queue properties.

In this chapter, we'll present several different approaches to implement queue. A queue is a FIFO data structure satisfies the following performance constraints.

- Element can be added to the tail of the queue in $O(1)$ constant time;
- Element can be removed from the head of the queue in $O(1)$ constant time.

These two properties must be satisfied. And it's common to add some extra goals, such as dynamic memory allocation etc.

Of course such abstract queue interface can be implemented with doubly-linked list trivially. But this is a overkill solution. We can even implement imperative queue with singly linked-list or plain array. However, our main question here is about how to realize a purely functional queue as well?

We'll first review the typical queue solution which is realized by singly linked-list and circular buffer in first section; Then we give a simple and straightforward functional solution in the second section. While the performance is ensured in terms of amortized constant time, we need find real-time solution (or worst-case solution) for some special case. Such solution will be described in the third and the fourth section. Finally, we'll show a very simple real-time queue which depends on lazy evaluation.

Most of the functional contents are based on Chris, Okasaki's great work in [6]. There are more than 16 different types of purely functional queue given in that material.

www.dbooks.org

10.2 Queue by linked-list and circular buffer

10.2.1 Singly linked-list solution

Queue can be implemented with singly linked-list. It's easy to add and remove element at the front end of a linked-list in $O(1)$ time. However, in order to keep the FIFO order, if we execute one operation on head, we must perform the inverse operation on tail.

In order to operate on tail, for plain singly linked-list, we must traverse the whole list before adding or removing. Traversing is bound to $O(n)$ time, where n is the length of the list. This doesn't match the abstract queue properties.

The solution is to use an extra record to store the tail of the linked-list. A sentinel is often used to simplify the boundary handling. The following ANSI C [1] code defines a queue realized by singly linked-list.

```
typedef int Key;

struct Node{
  Key key;
  struct Node* next;
};

struct Queue{
  struct Node *head, *tail;
};
```

Figure 10.1 illustrates an empty list. Both head and tail point to the sentinel NIL node.

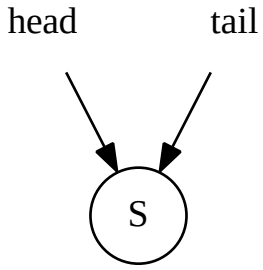


Figure 10.1: The empty queue, both head and tail point to sentinel node.

We summarize the abstract queue interface as the following.

function EMPTY ▷ Create an empty queue
function EMPTY?(Q) ▷ Test if Q is empty
function ENQUEUE(Q, x) ▷ Add a new element x to queue Q
function DEQUEUE(Q) ▷ Remove element from queue Q
function HEAD(Q) ▷ get the next element in queue Q in FIFO order

[1] It's possible to parameterize the type of the key with C++ template. ANSI C is used here for illustration purpose.

Note the difference between DEQUEUE and HEAD. HEAD only retrieve next element in FIFO order without removing it, while DEQUEUE performs removing.

In some programming languages, such as Haskell, and most object-oriented languages, the above abstract queue interface can be ensured by some definition. For example, the following Haskell code specifies the abstract queue.

```
class Queue q where
    empty :: q a
    isEmpty :: q a → Bool
    push :: q a → a → q a      -- Or named as 'snoc', append, pushλ_back
    pop :: q a → q a           -- Or named as 'tail', popλ_front
    front :: q a → a           -- Or named as 'head'
```

To ensure the constant time ENQUEUE and DEQUEUE, we add new element to head and remove element from tail.[2]

function ENQUEUE(Q, x)
  $p \leftarrow$ CREATE-NEW-NODE
  KEY(p) $\leftarrow x$
  NEXT(p) $\leftarrow NIL$
  NEXT(TAIL(Q)) $\leftarrow p$
  TAIL(Q) $\leftarrow p$

Note that, as we use the sentinel node, there are at least one node, the sentinel in the queue. That's why we needn't check the validation of of the tail before we append the new created node p to it.

function DEQUEUE(Q)
  $x \leftarrow$ HEAD(Q)
  NEXT(HEAD(Q)) $\leftarrow$ NEXT(x)
  if $x =$ TAIL(Q) **then** ▷ Q gets empty
    TAIL(Q) $\leftarrow$ HEAD(Q)
  return KEY(x)

As we always put the sentinel node in front of all the other nodes, function HEAD actually returns the next node to the sentinel.

Figure 10.2 illustrates ENQUEUE and DEQUEUE process with sentinel node.

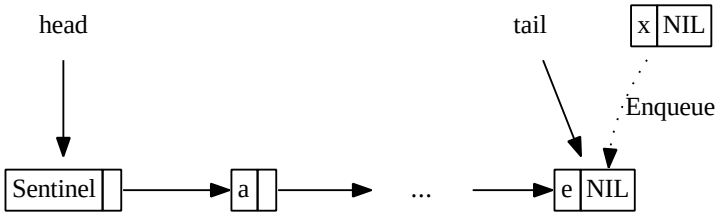
Translating the pseudo code to ANSI C program yields the below code.

```
struct Queue* enqueue(struct Queue* q, Key x) {
  struct Node* p = (struct Node*)malloc(sizeof(struct Node));
  p→key = x;
  p→next = NULL;
  q→tail→next = p;
  q→tail = p;
  return q;
}

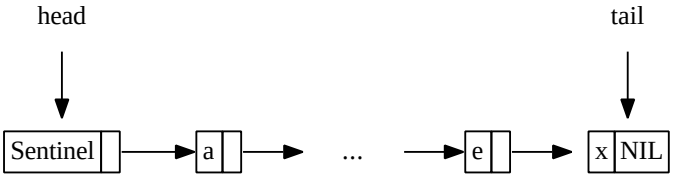
Key dequeue(struct Queue* q) {
  struct Node* p = head(q); /*gets the node next to sentinel*/
  Key x = key(p);
  q→head→next = p→next;
  if(q→tail == p)
```

[2]It's possible to add new element to the tail, while remove element from head, but the operations are more complex than this approach.

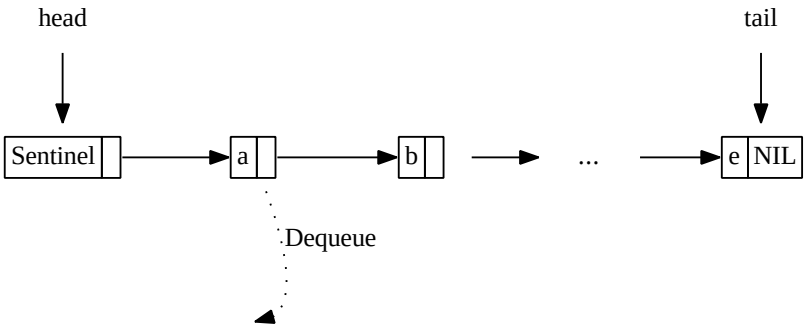
www.dbooks.org



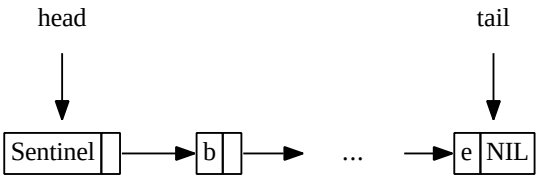
(a) Before ENQUEUE x to queue



(b) After ENQUEUE x to queue



(c) Before DEQUEUE



(d) After DEQUEUE

Figure 10.2: ENQUEUE and DEQUEUE to linked-list queue.

```
    q→tail = q→head;
  free(p);
  return x;
}
```

This solution is simple and robust. It's easy to extend this solution even to the concurrent environment (e.g. multicores). We can assign a lock to the head and use another lock to the tail. The sentinel helps us from being dead-locked due to the empty case [1] [2].

Exercise 10.1

- Realize the EMPTY? and HEAD algorithms for linked-list queue.
- Implement the singly linked-list queue in your favorite imperative programming language. Note that you need provide functions to initialize and destroy the queue.

10.2.2 Circular buffer solution

Another typical solution to realize queue is to use plain array as a circular buffer (also known as ring buffer). Oppose to linked-list, array support appending to the tail in constant $O(1)$ time if there are still spaces. Of course we need reallocate spaces if the array is fully occupied. However, Array performs poor in $O(n)$ time when removing element from head and packing the space. This is because we need shift all rest elements one cell ahead. The idea of circular buffer is to reuse the free cells before the first valid element after we remove elements from head.

The idea of circular buffer can be described in figure 10.3 and 10.4.

If we set a maximum size of the buffer instead of dynamically allocate memories, the queue can be defined with the below ANSI C code.

```
struct QueueBuf{
    Key* buf;
    int head, cnt, size;
};
```

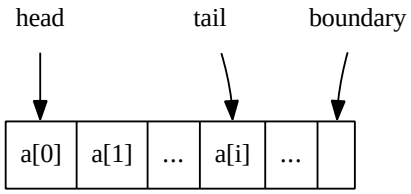
When initialize the queue, we are explicitly asked to provide the maximum size as the parameter.

```
struct QueueBuf* createQ(int max){
    struct QueueBuf* q = (struct QueueBuf*)malloc(sizeof(struct QueueBuf));
    q→buf = (Key*)malloc(sizeof(Key)*max);
    q→size = max;
    q→head = q→cnt = 0;
    return q;
}
```

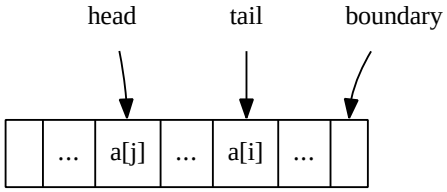
With the counter variable, we can compare it with zero and the capacity to test if the queue is empty or full.

function EMPTY?(Q)
  return COUNT(Q) = 0

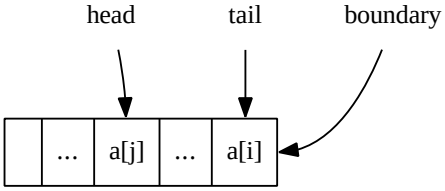
www.dbooks.org



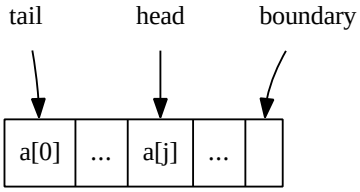
(a) Continuously add some elements.



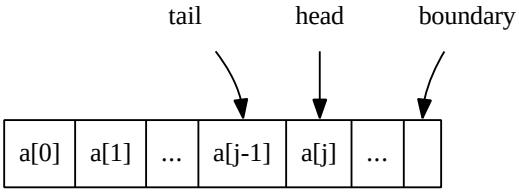
(b) After remove some elements from head, there are free cells.



(c) Go on adding elements till the boundary of the array.



(d) The next element is added to the first free cell on head.



(e) All cells are occupied. The queue is full.

Figure 10.3: A queue is realized with ring buffer.

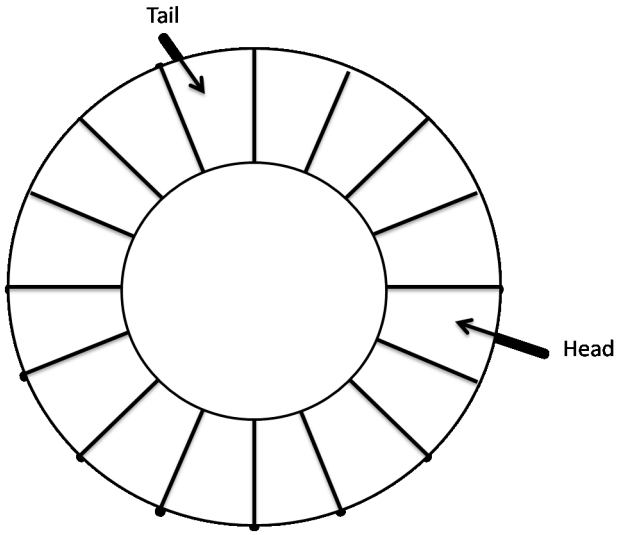


Figure 10.4: The circular buffer.

To realize ENQUEUE and DEQUEUE, an easy way is to calculate the modular of index as the following.

function ENQUEUE(Q, x)
  if $\neg$ FULL?(Q) **then**
    COUNT(Q) $\leftarrow$ COUNT(Q) + 1
    tail $\leftarrow$ (HEAD(Q) + COUNT(Q)) mod SIZE(Q)
    BUFFER(Q)[tail] $\leftarrow x$

function HEAD(Q)
  if $\neg$ EMPTY?(Q) **then**
    return BUFFER(Q)[HEAD(Q)]

function DEQUEUE(Q)
  if $\neg$ EMPTY?(Q) **then**
    HEAD(Q) $\leftarrow$ (HEAD(Q) + 1) mod SIZE(Q)
    COUNT(Q) $\leftarrow$ COUNT(Q) - 1

However, modular is expensive and slow depends on some settings, so one may replace it by some adjustment. For example as in the below ANSI C program.

```
void enQ(struct QueueBuf* q, Key x){
    if(!fullQ(q)){
        q->buf[offset(q->head + q->cnt, q->size)] = x;
        q->cnt++;
    }
}

Key headQ(struct Queue* q) {
    return q->buf[q->head];  //    I ¼»¿£
}
```

www.dbooks.org

```
Key deQ(struct QueueBuf* q){
    Key x = headQ(q);
    q->head = offset(++q->head, q->size);
    q->cnt--;
    return x;
}
```

Exercise 10.2

The circular buffer is allocated with a maximum size parameter. Can we test the queue is empty or full with only head and tail pointers? Note that the head can be either before or after the tail.

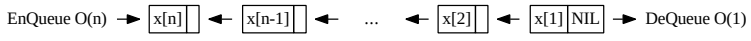
10.3 Purely functional solution

10.3.1 Paired-list queue

We can't just use a list to implement queue, or we can't satisfy abstract queue properties. This is because singly linked-list, which is the back-end data structure in most functional settings, performs well on head in constant $O(1)$ time, while it performs in linear $O(n)$ time on tail, where n is the length of the list. Either dequeue or enqueue will perform proportion to the number of elements stored in the list as shown in figure 10.5.



(a) DeQueue performs poorly.



(b) EnQueue performs poorly.

Figure 10.5: DeQueue and EnQueue can't perform both in constant $O(1)$ time with a list.

We neither can add a pointer to record the tail position of the list as what we have done in the imperative settings like in the ANSI C program, because of the nature of purely functional.

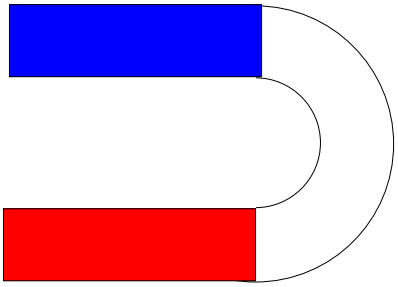
Chris Okasaki mentioned a simple and straightforward functional solution in [6]. The idea is to maintain two linked-lists as a queue, and concatenate these two lists in a tail-to-tail manner. The shape of the queue looks like a horseshoe magnet as shown in figure 10.6.

With this setup, we push new element to the head of the rear list, which is ensure to be $O(1)$ constant time; on the other hand, we pop element from the head of the front list, which is also $O(1)$ constant time. So that the abstract queue properties can be satisfied.

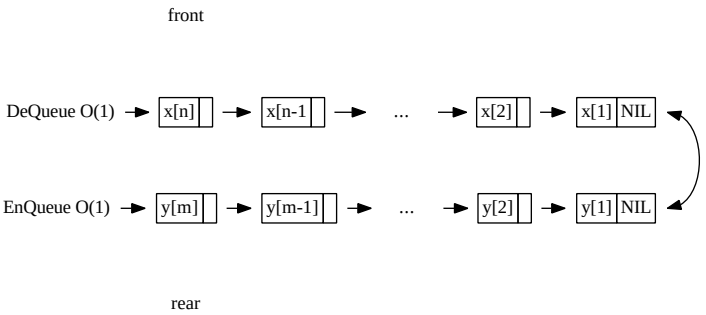
The definition of such paired-list queue can be expressed in the following Haskell code.

```
type Queue a = ([a], [a])

empty = ([], [])
```



(a) a horseshoe magnet.



(b) concatenate two lists tail-to-tail.

Figure 10.6: A queue with front and rear list shapes like a horseshoe magnet.

www.dbooks.org

Suppose function $front(Q)$ and $rear(Q)$ return the front and rear list in such setup, and $Queue(F, R)$ create a paired-list queue from two lists F and R. The EnQueue (push) and DeQueue (pop) operations can be easily realized based on this setup.

$$push(Q, x) = Queue(front(Q), \{x\} \cup rear(Q)) \tag{10.1}$$

$$pop(Q) = Queue(tail(front(Q)), rear(Q)) \tag{10.2}$$

where if a list $X = \{x_1, x_2, ..., x_n\}$, function $tail(X) = \{x_2, x_3, ..., x_n\}$ returns the rest of the list without the first element.

However, we must next solve the problem that after several pop operations, the front list becomes empty, while there are still elements in rear list. One method is to rebuild the queue by reversing the rear list, and use it to replace front list.

Hence a balance operation will be execute after popping. Let's denote the front and rear list of a queue Q as $F = front(Q)$, and $R = fear(Q)$.

$$balance(F, R) = \begin{cases} Queue(reverse(R), \phi) & : \quad F = \phi \\ Q & : \quad otherwise \end{cases} \tag{10.3}$$

Thus if front list isn't empty, we do nothing, while when the front list becomes empty, we use the reversed rear list as the new front list, and the new rear list is empty.

The new enqueue and dequeue algorithms are updated as below.

$$push(Q, x) = balance(F, \{x\} \cup R) \tag{10.4}$$

$$pop(Q) = balance(tail(F), R) \tag{10.5}$$

Sum up the above algorithms and translate them to Haskell yields the following program.

```
balance :: Queue a → Queue a
balance ([], r) = (reverse r, [])
balance q = q

push :: Queue a → a → Queue a
push (f, r) x = balance (f, x:r)

pop :: Queue a → Queue a
pop ([], _) = error "Empty"
pop (_:f, r) =  balance (f, r)
```

Although we only touch the heads of front list and rear list, the overall performance can't be kept always as $O(1)$. Actually, the performance of this algorithm is amortized $O(1)$. This is because the reverse operation takes time proportion to the length of the rear list. it's bound $O(n)$ time, where $N = |R|$. We left the prove of amortized performance as an exercise to the reader.

10.3.2 Paired-array queue - a symmetric implementation

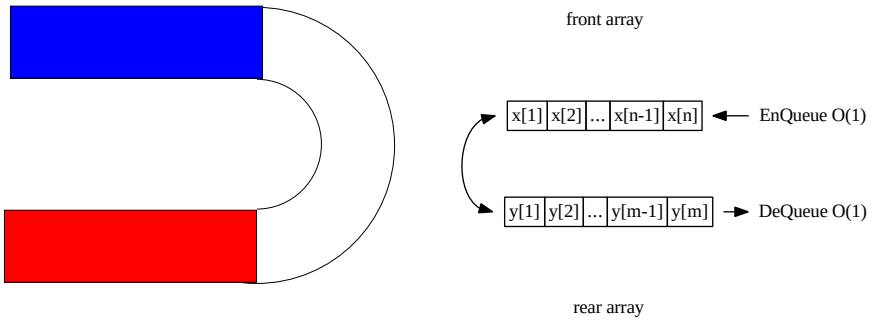
There is an interesting implementation which is symmetric to the paired-list queue. In some old programming languages, such as legacy version of BASIC, There is array supported, but there is no pointers, nor records to represent linked-list. Although we can use another array to store indexes so that we can represent linked-list with implicit array, there is another option to realized amortized $O(1)$ queue.

Compare the performance of array and linked-list. Below table reveals some facts (Suppose both contain n elements).

operation	Array	Linked-list
insert on head	$O(n)$	$O(1)$
insert on tail	$O(1)$	$O(n)$
remove on head	$O(n)$	$O(1)$
remove on tail	$O(1)$	$O(n)$

Note that linked-list performs in constant time on head, but in linear time on tail; while array performs in constant time on tail (suppose there is enough memory spaces, and omit the memory reallocation for simplification), but in linear time on head. This is because we need do shifting when prepare or eliminate an empty cell in array. (see chapter 'the evolution of insertion sort' for detail.)

The above table shows an interesting characteristic, that we can exploit it and provide a solution mimic to the paired-list queue: We concatenate two arrays, head-to-head, to make a horseshoe shape queue like in figure 10.7.



(a) a horseshoe magnet. (b) concatenate two arrays head-to-head.

Figure 10.7: A queue with front and rear arrays shapes like a horseshoe magnet.

We can define such paired-array queue like the following Python code [3]

```
class Queue:
    def __init__(self):
        self.front = []
        self.rear = []
```

[3]Legacy Basic code is not presented here. And we actually use list but not array in Python to illustrate the idea. ANSI C and ISO C++ programs are provides along with this chapter, they show more in a purely array manner.

www.dbooks.org

```
def is_empty(q):
    return q.front == [] and q.rear == []
```

The relative PUSH() and POP() algorithm only manipulate on the tail of the arrays.

function PUSH(Q, x)
  APPEND(REAR(Q), x)

Here we assume that the APPEND() algorithm append element x to the end of the array, and handle the necessary memory allocation etc. Actually, there are multiple memory handling approaches. For example, besides the dynamic re-allocation, we can initialize the array with enough space, and just report error if it's full.

function POP(Q)
  if FRONT(Q) $= \phi$ **then**
    FRONT(Q) $\leftarrow$ REVERSE(REAR(Q))
    REAR(Q) $\leftarrow \phi$
  $n \leftarrow$ LENGTH(FRONT(Q))
  $x \leftarrow$ FRONT(Q)[n]
  LENGTH(FRONT(Q)) $\leftarrow n - 1$
  return x

For simplification and pure illustration purpose, the array isn't shrunk explicitly after elements removed. So test if front array is empty (ϕ) can be realized as check if the length of the array is zero. We omit all these details here.

The enqueue and dequeue algorithms can be translated to Python programs straightforwardly.

```
def push(q, x):
    q.rear.append(x)

def pop(q):
    if q.front == []:
        q.rear.reverse()
        (q.front, q.rear) = (q.rear, [])
    return q.front.pop()
```

Similar to the paired-list queue, the performance is amortized $O(1)$ because the reverse procedure takes linear time.

Exercise 10.3

- Prove that the amortized performance of paired-list queue is $O(1)$.
- Prove that the amortized performance of paired-array queue is $O(1)$.

10.4 A small improvement, Balanced Queue

Although paired-list queue is amortized $O(1)$ for popping and pushing, the solution we proposed in previous section performs poor in the worst case. For

example, there is one element in the front list, and we push n elements continuously to the queue, here n is a big number. After that executing a pop operation will cause the worst case.

According to the strategy we used so far, all the n elements are added to the rear list. The front list turns to be empty after a pop operation. So the algorithm starts to reverse the rear list. This reversing procedure is bound to $O(n)$ time, which is proportion to the length of the rear list. Sometimes, it can't be acceptable for a very big n.

The reason why this worst case happens is because the front and rear lists are extremely unbalanced. We can improve our paired-list queue design by making them more balanced. One option is to add a balancing constraint.

$$|R| \leq |F| \tag{10.6}$$

Where $R = Rear(Q)$, $F = Front(Q)$, and $|L|$ is the length of list L. This constraint ensure the length of the rear list is less than the length of the front list. So that the reverse procedure will be executed once the rear list grows longer than the front list.

Here we need frequently access the length information of a list. However, calculate the length takes linear time for singly linked-list. We can record the length to a variable and update it as adding and removing elements. This approach enables us to get the length information in constant time.

Below example shows the modified paired-list queue definition which is augmented with length fields.

```
data BalanceQueue a = BQ [a] Int [a] Int
```

As we keep the invariant as specified in (10.6), we can easily tell if a queue is empty by testing the length of the front list.

$$F = \phi \Leftrightarrow |F| = 0 \tag{10.7}$$

In the rest part of this section, we suppose the length of a list L, can be retrieved as $|L|$ in constant time.

Push and pop are almost as same as before except that we check the balance invariant by passing length information and performs reversing accordingly.

$$push(Q, x) = balance(F, |F|, \{x\} \cup R, |R| + 1) \tag{10.8}$$

$$pop(Q) = balance(tail(F), |F| - 1, R, |R|) \tag{10.9}$$

Where function $balance()$ is defined as the following.

$$balance(F, |F|, R, |R|) = \begin{cases} Queue(F, |F|, R, |R|) & : \quad |R| \leq |F| \\ Queue(F \cup reverse(R), |F| + |R|, \phi, 0) & : \quad otherwise \end{cases} \tag{10.10}$$

Note that the function $Queue()$ takes four parameters, the front list along with its length (recorded), and the rear list along with its length, and forms a paired-list queue augmented with length fields.

We can easily translate the equations to Haskell program. And we can enforce the abstract queue interface by making the implementation an instance of the Queue type class.

www.dbooks.org

```
instance Queue BalanceQueue where
    empty = BQ [] 0 [] 0

    isEmpty (BQ _ lenf _ _) = lenf == 0

    -- Amortized O(1) time push
    push (BQ f lenf r lenr) x = balance f lenf (x:r) (lenr + 1)

    -- Amortized O(1) time pop
    pop (BQ (_:f) lenf r lenr) = balance f (lenf - 1) r lenr

    front (BQ (x:_) _ _ _) = x

balance f lenf r lenr
    | lenr ≤ lenf = BQ f lenf r lenr
    | otherwise = BQ (f ++ (reverse r)) (lenf + lenr) [] 0
```

Exercise 10.4

Write the symmetric balance improvement solution for paired-array queue in your favorite imperative programming language.

10.5 One more step improvement, Real-time Queue

Although the extremely worst case can be avoided by improving the balancing as what has been presented in previous section, the performance of reversing rear list is still bound to $O(n)$, where $N = |R|$. So if the rear list is very long, the instant performance is still unacceptable poor even if the amortized time is $O(1)$. It is particularly important in some real-time system to ensure the worst case performance.

As we have analyzed, the bottleneck is the computation of $F \cup reverse(R)$. This happens when $|R| > |F|$. Considering that $|F|$ and $|R|$ are all integers, so this computation happens when

$$|R| = |F| + 1 \tag{10.11}$$

Both F and the result of $reverse(R)$ are singly linked-list, It takes $O(|F|)$ time to concatenate them together, and it takes extra $O(|R|)$ time to reverse the rear list, so the total computation is bound to $O(|N|)$, where $N = |F| + |R|$. Which is proportion to the total number of elements in the queue.

In order to realize a real-time queue, we can't computing $F \cup reverse(R)$ monolithic. Our strategy is to distribute this expensive computation to every pop and push operations. Thus although each pop and push get a bit slow, we may avoid the extremely slow worst pop or push case.

Incremental reverse

Let's examine how functional reverse algorithm is implemented typically.

$$reverse(X) = \begin{cases} \phi & : \quad X = \phi \\ reverse(X') \cup \{x_1\} & : \quad otherwise \end{cases} \tag{10.12}$$

Where $X' = tail(X) = \{x_2, x_3, ...\}$.

This is a typical recursive algorithm, that if the list to be reversed is empty, the result is just an empty list. This is the edge case; otherwise, we take the first element x_1 from the list, reverse the rest $\{x_2, x_3, ..., x_n\}$, to $\{x_n, x_{n-1}, .., x_3, x_2\}$ and append x_1 after it.

However, this algorithm performs poor, as appending an element to the end of a list is proportion to the length of the list. So it's $O(N^2)$, but not a linear time reverse algorithm.

There exists another implementation which utilizes an accumulator A, like below.

$$reverse(X) = reverse'(X, \phi) \tag{10.13}$$

Where

$$reverse'(X, A) = \begin{cases} A & : \quad X = \phi \\ reverse'(X', \{x_1\} \cup A) & : \quad otherwise \end{cases} \tag{10.14}$$

We call A as *accumulator* because it accumulates intermediate reverse result at any time. Every time we call $reverse'(X, A)$, list X contains the rest of elements wait to be reversed, and A holds all the reversed elements so far. For instance when we call $reverse'()$ at i-th time, X and A contains the following elements:

$$X = \{x_i, x_{i+1}, ..., x_n\} \quad A = \{x_{i-1}, x_{i-2}, ...x_1\}$$

In every non-trivial case, we takes the first element from X in $O(1)$ time; then put it in front of the accumulator A, which is again $O(1)$ constant time. We repeat it n times, so this is a linear time ($O(n)$) algorithm.

The latter version of reverse is obviously a *tail-recursion* algorithm, see [5] and [6] for detail. Such characteristic is easy to change from monolithic algorithm to incremental manner.

The solution is state transferring. We can use a state machine contains two types of stat: reversing state S_r to indicate that the reverse is still on-going (not finished), and finish state S_f to indicate the reverse has been done (finished). In Haskell programming language, it can be defined as a type.

```
data State a = | Reverse [a] [a]
               | Done [a]
```

And we can schedule (slow-down) the above $reverse'(X, A)$ function with these two types of state.

$$step(S, X, A) = \begin{cases} (S_f, A) & : \quad S = S_r \wedge X = \phi \\ (S_r, X', \{x_1\} \cup A) & : \quad S = S_r \wedge X \neq \phi \end{cases} \tag{10.15}$$

Each step, we examine the state type first, if the current state is S_r (on-going), and the rest elements to be reversed in X is empty, we can turn the algorithm to finish state S_f; otherwise, we take the first element from X, put it in front of A just as same as above, but we do NOT perform recursion, instead, we just finish this step. We can store the current state as well as the resulted

www.dbooks.org

X and A, the reverse can be continued at any time when we call 'next' *step* function in the future with the stored state, X and A passed in.

Here is an example of this step-by-step reverse algorithm.

$$\begin{array}{lcl} step(S_r, \text{“hello”}, \phi) & = & (S_r, \text{“ello”}, \text{“h”}) \\ step(S_r, \text{“ello”}, \text{“h”}) & = & (S_r, \text{“llo”}, \text{“eh”}) \\ ... & & \\ step(S_r, \text{“o”}, \text{“lleh”}) & = & (S_r, \phi, \text{“olleh”}) \\ step(S_r, \phi, \text{“olleh”}) & = & (S_f, \text{“olleh”}) \end{array}$$

And in Haskell code manner, the example is like the following.

```
step $ Reverse "hello" [] = Reverse "ello" "h"
step $ Reverse "ello" "h" = Reverse "llo" "eh"
...
step $ Reverse "o" "lleh" = Reverse [] "olleh"
step $ Reverse [] "olleh" = Done "olleh"
```

Now we can distribute the reverse into steps in every pop and push operations. However, the problem is just half solved. We want to break down $F \cup reverse(R)$, and we have broken $reverse(R)$ into steps, we next need to schedule(slow-down) the list concatenation part $F \cup ...$, which is bound to $O(|F|)$, into incremental manner so that we can distribute it to pop and push operations.

Incremental concatenate

It's a bit more challenge to implement incremental list concatenation than list reversing. However, it's possible to re-use the result we gained from increment reverse by a small trick: In order to realize $X \cup Y$, we can first reverse X to $\overleftarrow{X}$, then take elements one by one from $\overleftarrow{X}$ and put them in front of Y just as what we have done in $reverse'$.

$$\begin{array}{rcl} X \cup Y & \equiv & reverse(reverse(X)) \cup Y \\ & \equiv & reverse'(reverse(X), \phi) \cup Y \\ & \equiv & reverse'(reverse(X), Y) \\ & \equiv & reverse'(\overleftarrow{X}, Y) \end{array} \tag{10.16}$$

This fact indicates us that we can use an extra state to instruct the *step*() function to continuously concatenating $\overleftarrow{F}$ after R is reversed.

The strategy is to do the total work in two phases:

1. Reverse both F and R in parallel to get $\overleftarrow{F} = reverse(F)$, and $\overleftarrow{R} = reverse(R)$ incrementally;
2. Incrementally take elements from $\overleftarrow{F}$ and put them in front of $\overleftarrow{R}$.

So we define three types of state: S_r represents reversing; S_c represents concatenating; and S_f represents finish.

In Haskell, these types of state are defined as the following.

```
data State a = Reverse [a] [a] [a] [a]
             | Concat [a] [a]
             | Done [a]
```

Because we reverse F and R simultaneously, so reversing state takes two pairs of lists and accumulators.

The state transferring is defined according to the two phases strategy described previously. Denotes that $F = \{f_1, f_2, ...\}$, $F' = tail(F) = \{f_2, f_3, ...\}$, $R = \{r_1, r_2, ...\}$, $R' = tail(R) = \{r_2, r_3, ...\}$. A state $\mathcal{S}$, contains it's type S, which has the value among S_r, S_c, and S_f. Note that $\mathcal{S}$ also contains necessary parameters such as F, $\overleftarrow{F}$, X, A etc as intermediate results. These parameters vary according to the different states.

$$next(\mathcal{S}) = \begin{cases} (S_r, F', \{f_1\} \cup \overleftarrow{F}, R', \{r_1\} \cup \overleftarrow{R}) & : \quad S = S_r \wedge F \neq \phi \wedge R \neq \phi \\ (S_c, \overleftarrow{F}, \{r_1\} \cup \overleftarrow{R}) & : \quad S = S_r \wedge F = \phi \wedge R = \{r_1\} \\ (S_f, A) & : \quad S = S_c \wedge X = \phi \\ (S_c, X', \{x_1\} \cup A) & : \quad S = S_c \wedge X \neq \phi \end{cases} \tag{10.17}$$

The relative Haskell program is list as below.

```
next (Reverse (x:f) f' (y:r) r') = Reverse f (x:f') r (y:r')
next (Reverse [] f' [y] r') = Concat f' (y:r')
next (Concat 0 _ acc) = Done acc
next (Concat (x:f') acc) = Concat f' (x:acc)
```

All left to us is to distribute these incremental steps into every pop and push operations to implement a real-time $O(1)$ purely functional queue.

Sum up

Before we dive into the final real-time queue implementation, let's analyze how many incremental steps are taken to achieve the result of $F \cup reverse(R)$. According to the balance variant we used previously, $|R| = |F| + 1$, Let's denotes $m = |F|$.

Once the queue gets unbalanced due to some push or pop operation, we start this incremental $F \cup reverse(R)$. It needs $m + 1$ steps to reverse R, and at the same time, we finish reversing the list F within these steps. After that, we need extra $m + 1$ steps to execute the concatenation. So there are $2m + 2$ steps.

It seems that distribute one step inside one pop or push operation is the natural solution. However, there is a critical question must be answered: Is it possible that before we finish these $2m + 2$ steps, the queue gets unbalanced again due to a series push and pop?

There are two facts about this question, one is good news and the other is bad news.

Let's first show the good news, that luckily, continuously pushing can't make the queue unbalanced again before we finish these $2m + 2$ steps to achieve $F \cup reverse(R)$. This is because once we start re-balancing, we can get a new front list $F' = F \cup reverse(R)$ after $2m+2$ steps. While the next time unbalance is triggered when

$$\begin{aligned} |R'| &= |F'| + 1 \\ &= |F| + |R| + 1 \\ &= 2m + 2 \end{aligned} \tag{10.18}$$

That is to say, even we continuously pushing as mush elements as possible after the last unbalanced time, when the queue gets unbalanced again, the $2m+2$

www.dbooks.org

front copy	on-going computation	new rear
$\{f_i, f_{i+1}, ..., f_M\}$	$(S_r, \tilde{F}, ..., \tilde{R}, ...)$	$\{...\}$
first $i-1$ elements popped	intermediate result $\overleftarrow{F}$ and $\overleftarrow{R}$	new elements pushed

Table 10.1: Intermediate state of a queue before first m steps finish.

steps exactly get finished at that time point. Which means the new front list F' is calculated OK. We can safely go on to compute $F' \cup reverse(R')$. Thanks to the balance invariant which is designed in previous section.

But, the bad news is that, pop operation can happen at anytime before these $2m+2$ steps finish. The situation is that once we want to extract element from front list, the new front list $F' = F \cup reverse(R)$ hasn't been ready yet. We don't have a valid front list at hand.

One solution to this problem is to keep a copy of original front list F, during the time we are calculating $reverse(F)$ which is described in phase 1 of our incremental computing strategy. So that we are still safe even if user continuously performs first m pop operations. So the queue looks like in table 10.1 at some time after we start the incremental computation and before phase 1 (reverse F and R simultaneously) ending[4].

After these M pop operations, the copy of F is exhausted. And we just start incremental concatenation phase at that time. What if user goes on popping?

The fact is that since F is exhausted (becomes ϕ), we needn't do concatenation at all. Since $F \cup \overleftarrow{R} = \phi \cup \overleftarrow{R} = \overleftarrow{R}$.

It indicates us, when doing concatenation, we only need to concatenate those elements haven't been popped, which are still left in F. As user pops elements one by one continuously from the head of front list F, one method is to use a counter, record how many elements there are still in F. The counter is initialized as 0 when we start computing $F \cup reverse(R)$, it's increased by one when we reverse one element in F, which means we need concatenate this element in the future; and it's decreased by one every time when pop is performed, which means we can concatenate one element less; of course we need decrease this counter as well in every steps of concatenation. If and only if this counter becomes zero, we needn't do concatenations any more.

We can give the realization of purely functional real-time queue according to the above analysis.

We first add an idle state S_0 to simplify some state transferring. Below Haskell program is an example of this modified state definition.

```
data State a = Empty
             | Reverse Int [a] [a] [a] [a] -- n, f', accλ_f' r, accλ_r
             | Append Int [a] [a]          -- n, revλ_f', acc
             | Done [a] -- result: f ++ reverse r
```

[4]One may wonder that copying a list takes linear time to the length of the list. If so the whole solution would make no sense. Actually, this linear time copying won't happen at all. This is because the purely functional nature, the front list won't be mutated either by popping or by reversing. However, if trying to realize a symmetric solution with paired-array and mutate the array in-place, this issue should be stated, and we can perform a 'lazy' copying, that the real copying work won't execute immediately, instead, it copies one element every step we do incremental reversing. The detailed implementation is left as an exercise.

And the data structure is defined with three parts, the front list (augmented with length); the on-going state of computing $F \cup reverse(R)$; and the rear list (augmented with length).

Here is the Haskell definition of real-time queue.

```
data RealtimeQueue a = RTQ [a] Int (State a) [a] Int
```

The empty queue is composed with empty front and rear list together with idle state S_0 as $Queue(\phi, 0, S_0, \phi, 0)$. And we can test if a queue is empty by checking if $|F| = 0$ according to the balance invariant defined before. Push and pop are changed accordingly.

$$push(Q, x) = balance(F, |F|, \mathcal{S}, \{x\} \cup R, |R| + 1) \tag{10.19}$$

$$pop(Q) = balance(F', |F| - 1, abort(\mathcal{S}), R, |R|) \tag{10.20}$$

The major difference is $abort()$ function. Based on our above analysis, when there is popping, we need decrease the counter, so that we can concatenate one element less. We define this as aborting. The details will be given after $balance()$ function.

The relative Haskell code for push and pop are listed like this.

```
push (RTQ f lenf s r lenr) x = balance f lenf s (x:r) (lenr + 1)
pop (RTQ (_:f) lenf s r lenr) = balance f (lenf - 1) (abort s) r lenr
```

The $balance()$ function first check the balance invariant, if it's violated, we need start re-balance it by starting compute $F \cup reverse(R)$ incrementally; otherwise we just execute one step of the unfinished incremental computation.

$$balance(F, |F|, \mathcal{S}, R, |R|) = \begin{cases} step(F, |F|, \mathcal{S}, R, |R|) & : \quad |R| \leq |F| \\ step(F, |F| + |R|, (S_r, 0, F, \phi, R, \phi)\phi, 0) & : \quad otherwise \end{cases} \tag{10.21}$$

The relative Haskell code is given like below.

```
balance f lenf s r lenr
    | lenr ≤ lenf =  step f lenf s r lenr
    | otherwise = step f (lenf + lenr) (Reverse 0 f [] r []) [] 0
```

The $step()$ function typically transfer the state machine one state ahead, and it will turn the state to idle (S_0) when the incremental computation finishes.

$$step(F, |F|, \mathcal{S}, R, |R|) = \begin{cases} Queue(F', |F|, S_0, R, |R|) & : \quad S' = S_f \\ Queue(F, |F|, \mathcal{S}', R, |R|) & : \quad otherwise \end{cases} \tag{10.22}$$

Where $\mathcal{S}' = next(\mathcal{S})$ is the next state transferred; $F' = F \cup reverse(R)$, is the final new front list result from the incremental computing. The real state transferring is implemented in $next()$ function as the following. It's different from previous version by adding the counter field n to record how many elements

www.dbooks.org

left we need to concatenate.

$$next(\mathcal{S}) = \begin{cases} (S_r, n+1, F', \{f_1\} \cup \overleftarrow{F}, R', \{r_1\} \cup \overleftarrow{R}) & : & S = S_r \wedge F \neq \phi \\ (S_c, n, \overleftarrow{F}, \{r_1\} \cup \overleftarrow{R}) & : & S = S_r \wedge F = \phi \\ (S_f, A) & : & S = S_c \wedge n = 0 \\ (S_c, n-1, X', \{x_1\} \cup A) & : & S = S_c \wedge n \neq 0 \\ \mathcal{S} & : & otherwise \end{cases} \tag{10.23}$$

And the corresponding Haskell code is like this.

```
next (Reverse n (x:f) f' (y:r) r') = Reverse (n+1) f (x:f') r (y:r')
next (Reverse n [] f' [y] r') = Concat n f' (y:r')
next (Concat 0 _ acc) = Done acc
next (Concat n (x:f') acc) = Concat (n-1) f' (x:acc)
next s = s
```

Function $abort()$ is used to tell the state machine, we can concatenate one element less since it is popped.

$$abort(\mathcal{S}) = \begin{cases} (S_f, A') & : & S = S_c \wedge n = 0 \\ (S_c, n-1, X'A) & : & S = S_c \wedge n \neq 0 \\ (S_r, n-1, F, \overleftarrow{F}, R, \overleftarrow{R}) & : & S = S_r \\ \mathcal{S} & : & otherwise \end{cases} \tag{10.24}$$

Note that when $n = 0$ we actually rollback one concatenated element by return A' as the result but not A. (Why? this is left as an exercise.)

The Haskell code for abort function is like the following.

```
abort (Concat 0 _ (_:acc)) = Done acc     -- Note! we rollback 1 elem
abort (Concat n f' acc) = Concat (n-1) f' acc
abort (Reverse n f f' r r') = Reverse (n-1) f f' r r'
abort s = s
```

It seems that we've done, however, there is still one tricky issue hidden behind us. If we push an element x to an empty queue, the result queue will be:

$$Queue(\phi, 1, (S_c, 0, \phi, \{x\}), \phi, 0)$$

If we perform pop immediately, we'll get an error! We found that the front list is empty although the previous computation of $F \cup reverse(R)$ has been finished. This is because it takes one more extra step to transfer from the state $(S_c, 0, \phi, A)$ to (S_f, A). It's necessary to refine the $\mathcal{S}'$ in $step()$ function a bit.

$$\mathcal{S}' = \begin{cases} next(next(\mathcal{S})) & : & F = \phi \\ next(\mathcal{S}) & : & otherwise \end{cases} \tag{10.25}$$

The modification reflects to the below Haskell code:

```
step f lenf s r lenr =
    case s' of
      Done f' → RTQ f' lenf Empty r lenr
      s' → RTQ f lenf s' r lenr
    where s' = if null f then next $ next s else next s
```

Note that this algorithm differs from the one given by Chris Okasaki in [6]. Okasaki's algorithm executes two steps per pop and push, while the one presents in this chapter executes only one per pop and push, which leads to more distributed performance.

Exercise 10.5

- Why need we rollback one element when $n = 0$ in $abort()$ function?
- Realize the real-time queue with symmetric paired-array queue solution in your favorite imperative programming language.
- In the footnote, we mentioned that when we start incremental reversing with in-place paired-array solution, copying the array can't be done monolithic or it will lead to linear time operation. Implement the lazy copying so that we copy one element per step along with the reversing.

10.6 Lazy real-time queue

The key to realize a real-time queue is to break down the expensive $F \cup reverse(R)$ to avoid monolithic computation. Lazy evaluation is particularly helpful in such case. In this section, we'll explore if there is some more elegant solution by exploit laziness.

Suppose that there exits a function $rotate()$, which can compute $F \cup reverse(R)$ incrementally. that's to say, with some accumulator A, the following two functions are equivalent.

$$rotate(X, Y, A) \equiv X \cup reverse(Y) \cup A \tag{10.26}$$

Where we initialized X as the front list F, Y as the rear list R, and the accumulator A is initialized as empty ϕ.

The trigger of rotation is still as same as before when $|F| + 1 = |R|$. Let's keep this constraint as an invariant during the whole rotation process, that $|X| + 1 = |Y|$ always holds.

It's obvious to deduce to the trivial case:

$$rotate(\phi, \{y_1\}, A) = \{y_1\} \cup A \tag{10.27}$$

Denote $X = \{x_1, x_2, ...\}$, $Y = \{y_1, y_2, ...\}$, and $X' = \{x_2, x_3, ...\}$, $Y' = \{y_2, y_3, ...\}$ are the rest of the lists without the first element for X and Y respectively. The recursion case is ruled out as the following.

$$\begin{array}{lcll} rotate(X, Y, A) & \equiv & X \cup reverse(Y) \cup A & \text{Definition of (10.26)} \\ & \equiv & \{x_1\} \cup (X' \cup reverse(Y) \cup A) & \text{Associative of } \cup \\ & \equiv & \{x_1\} \cup (X' \cup reverse(Y') \cup (\{y_1\} \cup A)) & \text{Nature of reverse and associative of } \cup \\ & \equiv & \{x_1\} \cup rotate(X', Y', \{y_1\} \cup A) & \text{Definition of (10.26)} \end{array} \tag{10.28}$$

www.dbooks.org

Summarize the above two cases, yields the final incremental rotate algorithm.

$$rotate(X, Y, A) = \begin{cases} \{y_1\} \cup A & : & X = \phi \\ \{x_1\} \cup rotate(X', Y', \{y_1\} \cup A) & : & otherwise \end{cases} \tag{10.29}$$

If we execute $\cup$ lazily instead of strictly, that is, execute $\cup$ once pop or push operation is performed, the computation of $rotate$ can be distribute to push and pop naturally.

Based on this idea, we modify the paired-list queue definition to change the front list to a lazy list, and augment it with a computation stream. [5]. When the queue triggers re-balance constraint by some pop/push, that $|F| + 1 = |R|$, The algorithm creates a lazy rotation computation, then use this lazy rotation as the new front list F'; the new rear list becomes ϕ, and a copy of F' is maintained as a stream.

After that, when we performs every push and pop; we consume the stream by forcing a $\cup$ operation. This results us advancing one step along the stream, $\{x\} \cup F''$, where $F'' = tail(F')$. We can discard x, and replace the stream F' with F''.

Once all of the stream is exhausted, we can start another rotation.

In order to illustrate this idea clearly, we turns to Scheme/Lisp programming language to show example codes, because it gives us explicit control of laziness.

In Scheme/Lisp, we have the following three tools to deal with lazy stream.

```
(define (cons-stream a b) (cons a (delay b)))

(define stream-car car)

(define (stream-cdr s) (cdr (force s)))
```

Function `cons-stream` constructs a 'lazy' list from an element x and an existing list L without really evaluating the value of L; The evaluation is actually delayed to `stream-cdr`, where the computation is forced. delaying can be realized by lambda calculus as described in[5].

The lazy paired-list queue is defined as the following.

```
(define (make-queue f r s)
  (list f r s))

;; Auxiliary functions
(define (front-lst q) (car q))

(define (rear-lst q) (cadr q))

(define (rots q) (caddr q))
```

A queue is consist of three parts, a front list, a rear list, and a stream which represents the computation of $F \cup reverse(R)$. Create an empty queue is trivial as making all these three parts null.

```
(define empty (make-queue '() '() '()))
```

Note that the front-list is also lazy stream actually, so we need use stream related functions to manipulate it. For example, the following function test if the queue is empty by checking the front lazy list stream.

```
(define (empty? q) (stream-null? (front-lst q)))
```

The push function is almost as same as the one given in previous section. That we put the new element in front of the rear list; and then examine the balance invariant and do necessary balancing works.

$$push(Q, x) = balance(\mathcal{F}, \{x\} \cup R, \mathcal{R}_s) \tag{10.30}$$

Where $\mathcal{F}$ represents the lazy stream of front list; $\mathcal{R}_s$ is the stream of rotation computation. The relative Scheme/Lisp code is give below.

```
(define (push q x)
  (balance (front-lst q) (cons x (rear q)) (rots q)))
```

While pop is a bit different, because the front list is actually lazy stream, we need force an evaluation. All the others are as same as before.

$$pop(Q) = balance(\mathcal{F}', R, \mathcal{R}_s) \tag{10.31}$$

Here $\mathcal{F}'$, force one evaluation to $\mathcal{F}$, the Scheme/Lisp code regarding to this equation is as the following.

```
(define (pop q)
  (balance (stream-cdr (front-lst q)) (rear q) (rots q)))
```

For illustration purpose, we skip the error handling (such as pop from an empty queue etc) here.

And one can access the top element in the queue by extract from the front list stream.

```
(define (front q) (stream-car (front-lst q)))
```

The balance function first checks if the computation stream is completely exhausted, and starts new rotation accordingly; otherwise, it just consumes one evaluation by enforcing the lazy stream.

$$balance(Q) = \begin{cases} Queue(\mathcal{F}', \phi, \mathcal{F}') & : \quad \mathcal{R}_s = \phi \\ Queue(\mathcal{F}, R, \mathcal{R}'_s) & : \quad otherwise \end{cases} \tag{10.32}$$

Here $\mathcal{F}'$ is defined to start a new rotation.

$$\mathcal{F}' = rotate(F, R, \phi) \tag{10.33}$$

The relative Scheme/Lisp program is listed accordingly.

```
(define (balance f r s)
  (if (stream-null? s)
      (let ((newf (rotate f r '())))
    (make-queue newf '() newf))
      (make-queue f r (stream-cdr s))))
```

The implementation of incremental rotate function is just as same as what we analyzed above.

```
(define (rotate xs ys acc)
  (if (stream-null? xs)
      (cons-stream (car ys) acc)
      (cons-stream (stream-car xs)
           (rotate (stream-cdr xs) (cdr ys)
               (cons-stream (car ys) acc)))))
```

www.dbooks.org

We used explicit lazy evaluation in Scheme/Lisp. Actually, this program can be very short by using lazy programming languages, for example, Haskell.

```
data LazyRTQueue a = LQ [a] [a] [a] -- front, rear, f ++ reverse r

instance Queue LazyRTQueue where
    empty = LQ [] [] []

    isEmpty (LQ f _ _) = null f

    -- O(1) time push
    push (LQ f r rot) x = balance f (x:r) rot

    -- O(1) time pop
    pop (LQ (_:f) r rot) = balance f r rot

    front (LQ (x:_) _ _) = x

balance f r [] = let f' = rotate f r [] in LQ f' [] f'
balance f r (_:rot) = LQ f r rot

rotate [] [y] acc = y:acc
rotate (x:xs) (y:ys) acc = x : rotate xs ys (y:acc)
```

10.7 Notes and short summary

Just as mentioned in the beginning of this book in the first chapter, queue isn't so simple as it was thought. We've tries to explain algorithms and data structures both in imperative and in function approaches; Sometimes, it gives impression that functional way is simpler and more expressive in most time. However, there are still plenty of areas, that more studies and works are needed to give equivalent functional solution. Queue is such an important topic, that it links to many fundamental purely functional data structures.

That's why Chris Okasaki made intensively study and took a great amount of discussions in [6]. With purely functional queue solved, we can easily implement dequeue with the similar approach revealed in this chapter. As we can handle elements effectively in both head and tail, we can advance one step ahead to realize sequence data structures, which support fast concatenate, and finally we can realize random access data structures to mimic array in imperative settings. The details will be explained in later chapters.

Note that, although we haven't mentioned priority queue, it's quite possible to realized it with heaps. We have covered topic of heaps in several previous chapters.

Exercise 10.6

- Realize dequeue, which support adding and removing elements on both sides in constant $O(1)$ time in purely functional way.
- Realize dequeue in a symmetric solution only with array in your favorite imperative programming language.

Bibliography

[1] Maged M. Michael and Michael L. Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms”. http://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html

[2] Herb Sutter. “Writing a Generalized Concurrent Queue”. Dr. Dobb’s Oct 29, 2008. http://drdobbs.com/cpp/211601363?pgno=1

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. The MIT Press, 2001. ISBN: 0262032937.

[4] Chris Okasaki. “Purely Functional Data Structures”. Cambridge university press, (July 1, 1999), ISBN-13: 978-0521663502

[5] Wikipedia. “Tail-call”. http://en.wikipedia.org/wiki/Tail_call

[6] Wikipedia. “Recursion (computer science)”. http://en.wikipedia.org/wiki/Recursion_(computer_science)#Tail-recursive_functions

[7] Harold Abelson, Gerald Jay Sussman, Julie Sussman. “Structure and Interpretation of Computer Programs, 2nd Edition”. MIT Press, 1996, ISBN 0-262-51087-1

www.dbooks.org

Chapter 11

Sequences, The last brick

11.1 Introduction

In the first chapter of this book, which introduced binary search tree as the 'hello world' data structure, we mentioned that neither queue nor array is simple if realized not only in imperative way, but also in functional approach. In previous chapter, we explained functional queue, which achieves the similar performance as its imperative counterpart. In this chapter, we'll dive into the topic of array-like data structures.

We have introduced several data structures in this book so far, and it seems that functional approaches typically bring more expressive and elegant solution. However, there are some areas, people haven't found competitive purely functional solutions which can match the imperative ones. For instance, the Ukkonen linear time suffix tree construction algorithm. another examples is Hashing table. Array is also among them.

Array is trivial in imperative settings, it enables randomly accessing any elements with index in constant $O(1)$ time. However, this performance target can't be achieved directly in purely functional settings as there is only list can be used.

In this chapter, we are going to abstract the concept of array to sequences. Which support the following features

- Element can be inserted to or removed from the head of the sequence quickly in $O(1)$ time;
- Element can be inserted to or removed from the tail of the sequence quickly in $O(1)$ time;
- Support concatenate two sequences quickly (faster than linear time);
- Support randomly access and update any element quickly;
- Support split at any position quickly;

We call these features abstract sequence properties, and it easy to see the fact that even array (here means plain-array) in imperative settings can't meet them all at the same time.

www.dbooks.org

We'll provide three solutions in this chapter. Firstly, we'll introduce a solution based on binary tree forest and numeric representation; Secondly, we'll show a concatenate-able list solution; Finally, we'll give the finger tree solution.

Most of the results are based on Chris, Okasaki's work in [6].

11.2 Binary random access list

11.2.1 Review of plain-array and list

Let's review the performance of plain-array and singly linked-list so that we know how they perform in different cases.

operation	Array	Linked-list
operation on head	$O(n)$	$O(1)$
operation on tail	$O(1)$	$O(n)$
access at random position	$O(1)$	average $O(n)$
remove at given position	average $O(n)$	$O(1)$
concatenate	$O(n_2)$	$O(n_1)$

Because we hold the head of linked list, operations on head such as insert and remove perform in constant time; while we need traverse to the end to perform removing or appending on tail; Given a position i, it need traverse i elements to access it. Once we are at that position, removing element from there is just bound to constant time by modifying some pointers. In order to concatenate two linked-lists, we need traverse to the end of the first one, and link it to the second one, which is bound to the length of the first linked-list;

On the other hand, for array, we must prepare free cell for inserting a new element to the head of it, and we need release the first cell after the first element being removed, all these two operations are achieved by shifting all the rest elements forward or backward, which costs linear time. While the operations on the tail of array are trivial constant time. Array also support accessing random position i by nature; However, removing the element at that position causes shifting all elements after it one position ahead. In order to concatenate two arrays, we need copy all elements from the second one to the end of the first one (ignore the memory re-allocation details), which is proportion to the length of the second array.

In the chapter about binomial heaps, we have explained the idea of using forest, which is a list of trees. It brings us the merit that, for any given number n, by representing it in binary number, we know how many binomial trees need to hold them. That each bit of 1 represents a binomial tree of that rank of bit. We can go one step ahead, if we have a n nodes binomial heap, for any given index $1 < i < n$, we can quickly know which binomial tree in the heap holds the i-th node.

11.2.2 Represent sequence by trees

One solution to realize a random-access sequence is to manage the sequence with a forest of complete binary trees. Figure 11.1 shows how we attach such trees to a sequence of numbers.

Here two trees t_1 and t_2 are used to represent sequence $\{x_1, x_2, x_3, x_4, x_5, x_6\}$. The size of binary tree t_1 is 2. The first two elements $\{x_1, x_2\}$ are leaves of t_1;

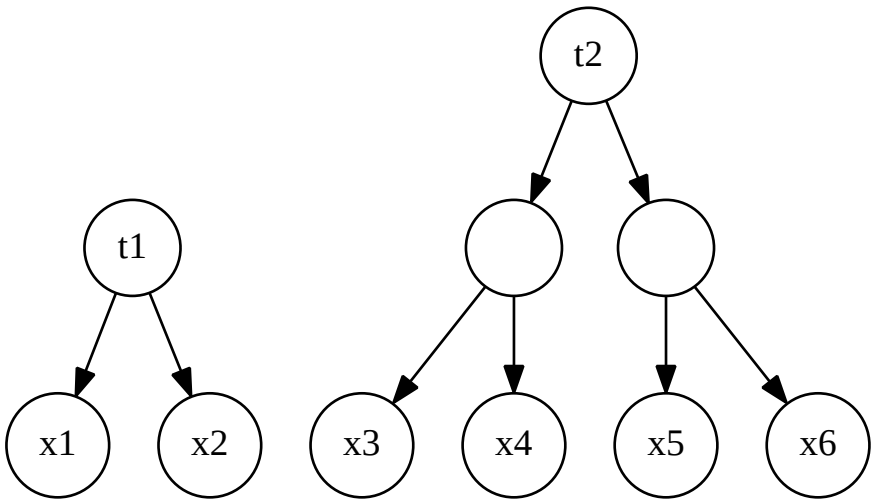


Figure 11.1: A sequence of 6 elements can be represented in a forest.

the size of binary tree t_2 is 4. The next four elements $\{x_3, x_4, x_5, x_6\}$ are leaves of t_2.

For a complete binary tree, we define the depth as 0 if the tree has only a leaf. The tree is denoted as as t_i if its depth is $i + 1$. It's obvious that there are 2^i leaves in t_i.

For any sequence contains n elements, it can be turned to a forest of complete binary trees in this manner. First we represent n in binary number like below.

$$n = 2^0 e_0 + 2^1 e_1 + ... + 2^m e_m \tag{11.1}$$

Where e_i is either 1 or 0, so $n = (e_m e_{m-1} ... e_1 e_0)_2$. If $e_i \neq 0$, we then need a complete binary tree with size 2^i. For example in figure 11.1, as the length of sequence is 6, which is $(110)_2$ in binary. The lowest bit is 0, so we needn't a tree of size 1; the second bit is 1, so we need a tree of size 2, which has depth of 2; the highest bit is also 1, thus we need a tree of size 4, which has depth of 3.

This method represents the sequence $\{x_1, x_2, ..., x_n\}$ to a list of trees $\{t_0, t_1, ..., t_m\}$ where t_i is either empty if $e_i = 0$ or a complete binary tree if $e_i = 1$. We call this representation as *Binary Random Access List* [6].

We can reused the definition of binary tree. For example, the following Haskell program defines the tree and the binary random access list.

```
data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)  -- size, left, right

type BRAList a = [Tree a]
```

The only difference from the typical binary tree is that we augment the size information to the tree. This enable us to get the size without calculation at every time. For instance.

```
size (Leaf _) = 1
size (Node sz _ _) = sz
```

www.dbooks.org

11.2.3 Insertion to the head of the sequence

The new forest representation of sequence enables many operation effectively. For example, the operation of inserting a new element y in front of sequence can be realized as the following.

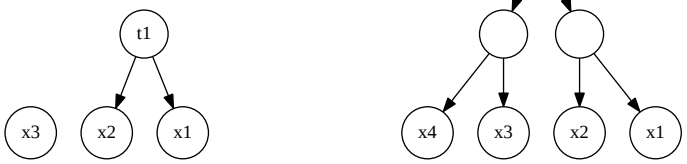
1. Create a tree t', with y as the only one leaf;
2. Examine the first tree in the forest, compare its size with t', if its size is greater than t', we just let t' be the new head of the forest, since the forest is a linked-list of tree, insert t' to its head is trivial operation, which is bound to constant $O(1)$ time;
3. Otherwise, if the size of first tree in the forest is equal to t', let's denote this tree in the forest as t_i, we can construct a new binary tree t'_{i+1} by linking t_i and t' as its left and right children. After that, we recursively try to insert t'_{i+1} to the forest.

Figure 11.2 and 11.3 illustrate the steps of inserting elements $x_1, x_2, ..., x_6$ to an empty forest.



(a) A singleton leaf of x_1

(b) Insert x_2. It causes linking, results a tree of height 1.



(c) Insert x_3. the result is two trees, t_1 and t_2

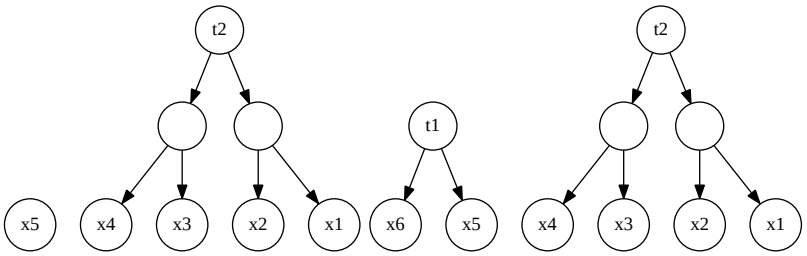
(d) Insert x_4. It first causes linking two leafs to a binary tree, then it performs linking again, which results a final tree of height 2.

Figure 11.2: Steps of inserting elements to an empty list, 1

As there are at most M trees in the forest, and m is bound to $O(\lg n)$, so the insertion to head algorithm is ensured to perform in $O(\lg n)$ even in worst case. We'll prove the amortized performance is $O(1)$ later.

Let's formalize the algorithm. we define the function of inserting an element in front of a sequence as $insert(S, x)$.

$$insert(S, x) = insertTree(S, leaf(x)) \tag{11.2}$$



(a) Insert x_5. The forest is a leaf (t_0) and t_2.

(b) Insert x_6. It links two leaf to t_1.

Figure 11.3: Steps of inserting elements to an empty list, 2

This function just wrap element x to a singleton tree with a leaf, and call $insertTree$ to insert this tree to the forest. Suppose the forest $F = \{t_1, t_2, ...\}$ if it's not empty, and $F' = \{t_2, t_3, ...\}$ is the rest of trees without the first one.

$$insertTree(F, t) = \begin{cases} \{t\} & : & F = \phi \\ \{t\} \cup F & : & size(t) < size(t_1) \\ insertTree(F', link(t, t_1)) & : & otherwise \end{cases} \tag{11.3}$$

Where function $link(t_1, t_2)$ create a new tree from two small trees with same size. Suppose function $tree(s, t_1, t_2)$ create a tree, set its size as s, makes t_1 as the left child, and t_2 as the right child, linking can be realized as below.

$$link(t_1, t_2) = tree(size(t_1) + size(t_2), t_1, t_2) \tag{11.4}$$

The relative Haskell programs can be given by translating these definitions.

```
cons :: a → BRAList a → BRAList a
cons x ts = insertTree ts (Leaf x)

insertTree :: BRAList a → Tree a → BRAList a
insertTree [] t = [t]
insertTree (t':ts) t = if size t < size t' then  t:t':ts
                       else insertTree ts (link t t')

-- Precondition: rank t1 = rank t2
link :: Tree a → Tree a → Tree a
link t1 t2 = Node (size t1 + size t2) t1 t2
```

Here we use the Lisp tradition to name the function that insert an element before a list as 'cons'.

Remove the element from the head of the sequence

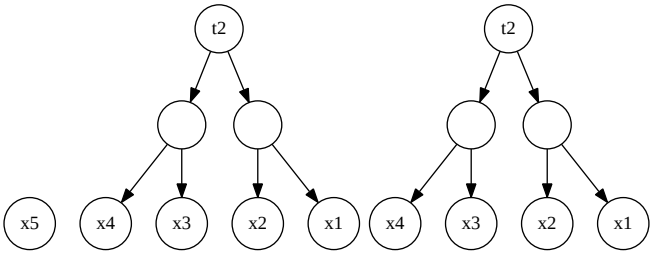
It's not complex to realize the inverse operation of 'cons', which can remove element from the head of the sequence.

- If the first tree in the forest is a singleton leaf, remove this tree from the forest;

www.dbooks.org

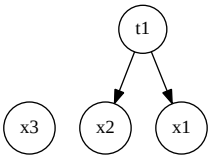
- otherwise, we can halve the first tree by unlinking its two children, so the first tree in the forest becomes two trees, we recursively halve the first tree until it turns to be a leaf.

Figure 11.4 illustrates the steps of removing elements from the head of the sequence.



(a) A sequence of 5 elements

(b) Result of removing x_5, the leaf is removed.



(c) Result of removing x_4, As there is not leaf tree, the tree is firstly divided into two sub trees of size 2. The first tree is next divided again into two leafs, after that, the first leaf, which contains x_4 is removed. What left in the forest is a leaf tree of x_3, and a tree of size 2 with elements x_2, x_1.

Figure 11.4: Steps of removing elements from head

If we assume the sequence isn't empty, so that we can skip the error handling such as trying to remove an element from an empty sequence, this can be expressed with the following definition. We denote the forest $F = \{t_1, t_2, ...\}$ and the trees without the first one as $F' = \{t_2, t_3, ...\}$

$$extractTree(F) = \begin{cases} (t_1, F') & : \quad t_1 \quad \text{is leaf} \\ extractTree(\{t_l, t_r\} \cup F') & : \quad otherwise \end{cases} \tag{11.5}$$

where $\{t_l, t_r\} = unlink(t_1)$ are the two children of t_1.

It can be translated to Haskell programs like below.

```
extractTree (t@(Leaf x):ts) = (t, ts)
extractTree (t@(Node _ t1 t2):ts) = extractTree (t1:t2:ts)
```

With this function defined, it's convenient to give *head* and *tail* functions, the former returns the first element in the sequence, the latter return the rest.

$$head(S) = key(first(extractTree(S))) \tag{11.6}$$

$$tail(S) = second(extractTree(S)) \tag{11.7}$$

Where function $first$ returns the first element in a paired-value (as known as tuple); $second$ returns the second element respectively. Function key is used to access the element inside a leaf. Below are Haskell programs corresponding to these two functions.

```
head' ts = x where (Leaf x, _) = extractTree ts
tail' = snd ∘ extractTree
```

Note that as `head` and `tail` functions have already been defined in Haskell standard library, we given them apostrophes to make them distinct. (another option is to hide the standard ones by importing. We skip the details as they are language specific).

Random access the element in binary random access list

As trees in the forest help managing the elements in blocks, giving an arbitrary index, it's easy to locate which tree this element is stored, after that performing a search in the tree yields the result. As all trees are binary (more accurate, complete binary tree), the search is essentially binary search, which is bound to the logarithm of the tree size. This brings us a faster random access capability than linear search in linked-list setting.

Given an index i, and a sequence S, which is actually a forest of trees, the algorithm is executed as the following [1].

1. Compare i with the size of the first tree T_1 in the forest, if i is less than or equal to the size, the element exists in T_1, perform looking up in T_1;
2. Otherwise, decrease i by the size of T_1, and repeat the previous step in the rest of the trees in the forest.

This algorithm can be represented as the below equation.

$$get(S, i) = \begin{cases} lookupTree(T_1, i) & : \quad i \leq |T_1| \\ get(S', i - |T_1|) & : \quad otherwise \end{cases} \tag{11.8}$$

Where $|T| = size(T)$, and $S' = \{T_2, T_3, ...\}$ is the rest of trees without the first one in the forest. Note that we don't handle out of bound error case, this is left as an exercise to the reader.

Function $lookupTree$ is a binary search algorithm. If the index i is 1, we just return the root of the tree, otherwise, we halve the tree by unlinking, if i is less than or equal to the size of the halved tree, we recursively look up the left tree, otherwise, we look up the right tree.

$$lookupTree(T, i) = \begin{cases} root(T) & : \quad i = 1 \\ lookupTree(left(T)) & : \quad i \leq \lfloor \frac{|T|}{2} \rfloor \\ lookupTree(right(T)) & : \quad otherwise \end{cases} \tag{11.9}$$

Where function $left$ returns the left tree T_l of T, while $right$ returns T_r.

The corresponding Haskell program is given as below.

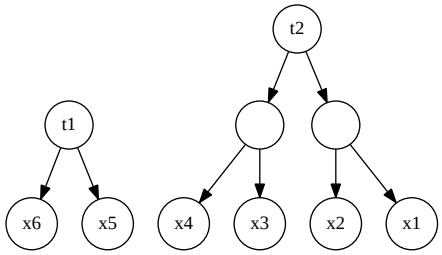
[1] We follow the tradition that the index i starts from 1 in algorithm description; while it starts from 0 in most programming languages

www.dbooks.org

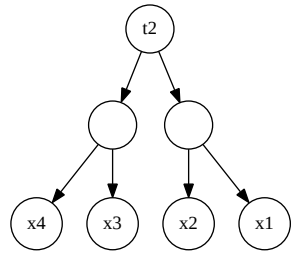
```
getAt (t:ts) i = if i < size t then lookupTree t i
                 else getAt ts (i - size t)

lookupTree (Leaf x) 0 = x
lookupTree (Node sz t1 t2) i = if i < sz `div` 2 then lookupTree t1 i
                               else lookupTree t2 (i - sz `div` 2)
```

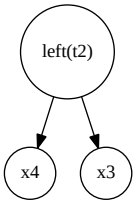
Figure 11.5 illustrates the steps of looking up the 4-th element in a sequence of size 6. It first examine the first tree, since the size is 2 which is smaller than 4, so it goes on looking up for the second tree with the updated index $i' = 4 - 2$, which is the 2nd element in the rest of the forest. As the size of the next tree is 4, which is greater than 2, so the element to be searched should be located in this tree. It then examines the left sub tree since the new index 2 is not greater than the half size 4/2=2; The process next visits the right grand-child, and the final result is returned.



(a) $getAt(S, 4))$, $4 > size(t_1) = 2$



(b) $getAt(S', 4-2) \Rightarrow lookupTree(t_2, 2)$



(c) $2 \leq \lfloor size(t_2)/2 \rfloor \Rightarrow lookupTree(left(t_2), 2)$



(d) $lookupTree(right(left(t_2)), 1)$, x_3 is returned.

Figure 11.5: Steps of locating the 4-th element in a sequence.

By using the similar idea, we can update element at any arbitrary position i. We first compare the size of the first tree T_1 in the forest with i, if it is less than i, it means the element to be updated doesn't exist in the first tree. We recursively examine the next tree in the forest, comparing it with $i - |T_1|$, where $|T_1|$ represents the size of the first tree. Otherwise if this size is greater than or equal to i, the element is in the tree, we halve the tree recursively until to get a leaf, at this stage, we can replace the element of this leaf with a new one.

$$set(S, i, x) = \begin{cases} \{updateTree(T_1, i, x)\} \cup S' & : \quad i < |T_1| \\ \{T_1\} \cup set(S', i - |T_1|, x) & : \quad otherwise \end{cases} \tag{11.10}$$

Where $S' = \{T_2, T_3, ...\}$ is the rest of the trees in the forest without the first one.

Function $setTree(T, i, x)$ performs a tree search and replace the i-th element with the given value x.

$$setTree(T, i, x) = \begin{cases} leaf(x) & : & i = 0 \wedge |T| = 1 \\ tree(|T|, setTree(T_l, i, x), T_r) & : & i < \lfloor \frac{|T|}{2} \rfloor \\ tree(|T|, T_l, setTree(T_r, i - \lfloor \frac{|T|}{2} \rfloor, x)) & : & otherwise \end{cases} \tag{11.11}$$

Where T_l and T_r are left and right sub tree of T respectively. The following Haskell program translates the equation accordingly.

```
setAt :: BRAList a → Int → a → BRAList a
setAt (t:ts) i x = if i < size t then (updateTree t i x):ts
                   else t:setAt ts (i-size t) x

updateTree :: Tree a → Int → a → Tree a
updateTree (Leaf _) 0 x = Leaf x
updateTree (Node sz t1 t2) i x =
    if i < sz `div` 2 then Node sz (updateTree t1 i x) t2
    else Node sz t1 (updateTree t2 (i - sz `div` 2) x)
```

As the nature of complete binary tree, for a sequence with n elements, which is represented by binary random access list, the number of trees in the forest is bound to $O(\lg n)$. Thus it takes $O(\lg n)$ time to locate the tree for arbitrary index i, that contains the element in the worst case. The followed tree search is bound to the heights of the tree, which is $O(\lg n)$ in the worst case as well. So the total performance of random access is $O(\lg n)$.

Exercise 11.1

1. The random access algorithm given in this section doesn't handle the error such as out of bound index at all. Modify the algorithm to handle this case, and implement it in your favorite programming language.

2. It's quite possible to realize the binary random access list in imperative settings, which is benefited with fast operation on the head of the sequence. the random access can be realized in two steps: firstly locate the tree, secondly use the capability of constant random access of array. Write a program to implement it in your favorite imperative programming language.

11.3 Numeric representation for binary random access list

In previous section, we mentioned that for any sequence with n elements, we can represent n in binary format so that $n = 2^0 e_0 + 2^1 e_1 + ... + 2^m e_m$. Where e_i is the i-th bit, which can be either 0 or 1. If $e_i \neq 0$ it means that there is a complete binary tree with size 2^i.

This fact indicates us that there is an explicit relationship between the binary form of n and the forest. Insertion a new element on the head can be simulated

www.dbooks.org

by increasing the binary number by one; while remove an element from the head mimics the decreasing of the corresponding binary number by one. This is as known as *numeric representation* [6].

In order to represent the binary random access list with binary number, we can define two states for a bit. That $Zero$ means there is no such a tree with size which is corresponding to the bit, while One, means such tree exists in the forest. And we can attach the tree with the state if it is One.

The following Haskell program for instance defines such states.

```
data Digit a = Zero
             | One (Tree a)

type RAList a = [Digit a]
```

Here we reuse the definition of complete binary tree and attach it to the state One. Note that we cache the size information in the tree as well.

With digit defined, forest can be treated as a list of digits. Let's see how inserting a new element can be realized as binary number increasing. Suppose function $one(t)$ creates a One state and attaches tree t to it. And function $getTree(s)$ get the tree which is attached to the One state s. The sequence S is a list of digits of states that $S = \{s_1, s_2, ...\}$, and S' is the rest of digits with the first one removed.

$$insertTree(S, t) = \begin{cases} \{one(t)\} & : & S = \phi \\ \{one(t)\} \cup S' & : & s_1 = Zero \\ \{Zero\} \cup insertTree(S', link(t, getTree(s_1))) & : & otherwise \end{cases} \tag{11.12}$$

When we insert a new tree t to a forest S of binary digits, If the forest is empty, we just create a One state, attach the tree to it, and make this state the only digit of the binary number. This is just like $0 + 1 = 1$;

Otherwise if the forest isn't empty, we need examine the first digit of the binary number. If the first digit is $Zero$, we just create a One state, attach the tree, and replace the $Zero$ state with the new created One state. This is just like $(...digits...0)_2 + 1 = (...digits...1)_2$. For example $6+1 = (110)_2+1 = (111)_2 = 7$.

The last case is that the first digit is One, here we make assumption that the tree t to be inserted has the same size with the tree attached to this One state at this stage. This can be ensured by calling this function from inserting a leaf, so that the size of the tree to be inserted grows in a series of $1, 2, 4, ..., 2^i, ...$ In such case, we need link these two trees (one is t, the other is the tree attached to the One state), and recursively insert the linked result to the rest of the digits. Note that the previous One state has to be replaced with a $Zero$ state. This is just like $(...digits...1)_2 + 1 = (...digits'...0)_2$, where $(...digits'...)_2 = (...digits...)_2 + 1$. For example $7 + 1 = (111)_2 + 1 = (1000)_2 = 8$

Translating this algorithm to Haskell yields the following program.

```
insertTree :: RAList a → Tree a → RAList a
insertTree [] t = [One t]
insertTree (Zero:ts) t = One t : ts
insertTree (One t' :ts) t = Zero : insertTree ts (link t t')
```

All the other functions, including $link(), cons()$ etc. are as same as before.

Next let's see how removing an element from a sequence can be represented as binary number deduction. If the sequence is a singleton *One* state attached with a leaf. After removal, it becomes empty. This is just like $1 - 1 = 0$;

Otherwise, we examine the first digit, if it is *One* state, it will be replaced with a *Zero* state to indicate that this tree will be no longer exist in the forest as it being removed. This is just like $(...digits...1)_2 - 1 = (...digits...0)_2$. For example $7 - 1 = (111)_2 - 1 = (110)_2 = 6$;

If the first digit in the sequence is a *Zero* state, we have to borrow from the further digits for removal. We recursively extract a tree from the rest digits, and halve the extracted tree to its two children. Then the *Zero* state will be replaced with a *One* state attached with the right children, and the left children is removed. This is something like $(...digits...0)_2 - 1 = (...digits'...1)_2$, where $(...digits'...)_2 = (...digits)_2 - 1$. For example $4 - 1 = (100)_2 - 1 = (11)_2 = 3$.The following equation illustrated this algorithm.

$$extractTree(S) = \begin{cases} (t, \phi) & : & S = \{one(t)\} \\ (t, S') & : & s_1 = one(t) \\ (t_l, \{one(t_r)\} \cup S'') & : & otherwise \end{cases} \tag{11.13}$$

Where $(t', S'') = extractTree(S')$, t_l and t_r are left and right sub-trees of t'. All other functions, including *head*, *tail* are as same as before.

Numeric representation doesn't change the performance of binary random access list, readers can refer to [2] for detailed discussion. Let's take for example, analyze the average performance (or amortized) of insertion on head algorithm by using aggregation analysis.

Considering the process of inserting $n = 2^m$ elements to an empty binary random access list. The numeric representation of the forest can be listed as the following.

i	forest (MSB ... LSB)
0	0, 0, ..., 0, 0
1	0, 0, ..., 0, 1
2	0, 0, ..., 1, 0
3	0, 0, ..., 1, 1
...	...
$2^m - 1$	1, 1, ..., 1, 1
2^m	1, 0, 0, ..., 0, 0
bits changed	1, 1, 2, ... 2^{m-1}. 2^m

The LSB of the forest changed every time when there is a new element inserted, it costs 2^m units of computation; The next bit changes every two times due to a linking operation, so it costs 2^{m-1} units; the bit next to MSB of the forest changed only one time which links all previous trees to a big tree as the only one in the forest. This happens at the half time of the total insertion process, and after the last element is inserted, the MSB flips to 1.

Sum these costs up yield to the total cost $T = 1+1+2+4+...+2^{m-1}+2^m = 2^{m+1}$ So the average cost for one insertion is

$$O(T/N) = O(\frac{2^{m+1}}{2^m}) = O(1) \tag{11.14}$$

Which proves that the insertion algorithm performs in amortized $O(1)$ constant time. The proof for deletion are left as an exercise to the reader.

www.dbooks.org

11.3.1 Imperative binary random access list

It's trivial to implement the imperative binary random access list by using binary trees, and the recursion can be eliminated by updating the focused tree in loops. This is left as an exercise to the reader. In this section, we'll show some different imperative implementation by using the properties of numeric representation.

Remind the chapter about binary heap. Binary heap can be represented by implicit array. We can use similar approach that use an array of 1 element to represent the leaf; use an array of 2 elements to represent a binary tree of height 1; and use an array of 2^m to represent a complete binary tree of height m.

This brings us the capability of accessing any element with index directly instead of divide and conquer tree search. However, the tree linking operation has to be implemented as array copying as the expense.

The following ANSI C code defines such a forest.

```
#define M sizeof(int) * 8
typedef int Key;

struct List {
  int n;
  Key* tree[M];
};
```

Where n is the number of the elements stored in this forest. Of course we can avoid limiting the max number of trees by using dynamic arrays, for example as the following ISO C++ code.

```
template<typename Key>
struct List {
  int n;
  vector<vector<key> > tree;
};
```

For illustration purpose only, we use ANSI C here[2].

Let's review the insertion process, if the first tree is empty (a *Zero* digit), we simply set the first tree as a leaf of the new element to be inserted; otherwise, the insertion will cause tree linking anyway, and such linking may be recursive until it reach a position (digit) that the corresponding tree is empty. The numeric representation reveals an important fact that if the first, second, ..., $(i-1)$-th trees all exist, and the i-th tree is empty, the result is creating a tree of size 2^i, and all the elements together with the new element to be inserted are stored in this new created tree. What's more, all trees after position i are kept as same as before.

Is there any good methods to locate this i position? As we can use binary number to represent the forest of n element, after a new element is inserted, n increases to $n+1$. Compare the binary form of n and $n+1$, we find that all bits before i change from 1 to 0, the i-th bit flip from 0 to 1, and all the bits after i keep unchanged. So we can use bit-wise exclusive or ($\oplus$) to detect this bit. Here is the algorithm.

function NUMBER-OF-BITS(n)
  $i \leftarrow 0$

[2]The complete ISO C++ example program is available with this book.

while $\lfloor \frac{n}{2} \rfloor \neq 0$ **do**
$n \leftarrow \lfloor \frac{n}{2} \rfloor$
$i \leftarrow i + 1$
return i

$i \leftarrow$ NUMBER-OF-BITS$(n \oplus (n + 1))$

The NUMBER-OF-BITS process can be easily implemented with bit shifting, for example the below ANSI C code.

```
int nbits(int n) {
  int i=0;
  while(n >>= 1)
    ++i;
  return i;
}
```

So the imperative insertion algorithm can be realized by first locating the bit which flip from 0 to 1, then creating a new array of size 2^i to represent a complete binary tree, and moving content of all trees before this bit to this array as well as the new element to be inserted.

function INSERT(L, x)
$i \leftarrow$ NUMBER-OF-BITS$(n \oplus (n + 1))$
TREE$(L)[i + 1] \leftarrow$ CREATE-ARRAY(2^i)
$l \leftarrow 1$
TREE$(L)[i + 1][l] \leftarrow x$
for $j \in [1, i]$ **do**
for $k \in [1, 2^j]$ **do**
$l \leftarrow l + 1$
TREE$(L)[i + 1][l] \leftarrow$ TREE$(L)[j][k]$
TREE$(L)[j] \leftarrow$ NIL
SIZE$(L) \leftarrow$ SIZE$(L) + 1$
return L

The corresponding ANSI C program is given as the following.

```
struct List insert(struct List a, Key x) {
  int i, j, sz;
  Key* xs;
  i = nbits((a.n + 1) ^ a.n );
  xs = a.tree[i] = (Key*)malloc(sizeof(Key)*(1 << i));
  for(j = 0, *xs++ = x, sz = 1; j < i; ++j, sz <<= 1) {
    memcpy((void*)xs, (void*)a.tree[j], sizeof(Key) * (sz));
    xs += sz;
    free(a.tree[j]);
    a.tree[j] = NULL;
  }
  ++a.n;
  return a;
}
```

However, the performance in theory isn't as good as before. This is because the linking operation downgrades from $O(1)$ constant time to linear array copying.

www.dbooks.org

We can again calculate the average (amortized) performance by using aggregation analysis. When insert $n = 2^m$ elements to an empty list which is represented by implicit binary trees in arrays, the numeric presentation of the forest of arrays are as same as before except for the cost of bit flipping.

i	forest (MSB ... LSB)
0	0, 0, ..., 0, 0
1	0, 0, ..., 0, 1
2	0, 0, ..., 1, 0
3	0, 0, ..., 1, 1
...	...
$2^m - 1$	1, 1, ..., 1, 1
2^m	1, 0, 0, ..., 0, 0
bit change cost	1×2^m, $1 \times 2^{m-1}$, $2 \times 2^{m-2}$, ... $2^{m-2} \times 2$, $2^{m-1} \times 1$

The LSB of the forest changed every time when there is a new element inserted, however, it creates leaf tree and performs copying only it changes from 0 to 1, so the cost is half of n unit, which is 2^{m-1}; The next bit flips as half as the LSB. Each time the bit gets flipped to 1, it copies the first tree as well as the new element to the second tree. the the cost of flipping a bit to 1 in this bit is 2 units, but not 1; For the MSB, it only flips to 1 at the last time, but the cost of flipping this bit, is copying all the previous trees to fill the array of size 2^m.

Summing all to cost and distributing them to the n times of insertion yields the amortized performance as below.

$$\begin{aligned} O(T/N) &= O(\frac{1 \times 2^m + 1 \times 2^{m-1} + 2 \times 2^{m-2} + ... + 2^{m-1} \times 1}{2^m}) \\ &= O(1 + \frac{m}{2}) \\ &= O(m) \end{aligned} \tag{11.15}$$

As $m = O(\lg n)$, so the amortized performance downgrade from constant time to logarithm, although it is still faster than the normal array insertion which is $O(n)$ in average.

The random accessing gets a bit faster because we can use array indexing instead of tree search.

function GET(L, i)
  for each $t \in$ TREES(L) **do**
    if $t \neq$ NIL **then**
      if $i \leq$ SIZE(t) **then**
        return t[i]
      else
        $i \leftarrow i-$ SIZE(t)

Here we skip the error handling such as out of bound indexing etc. The ANSI C program of this algorithm is like the following.

```
Key get(struct List a, int i) {
  int j, sz;
  for(j = 0, sz = 1; j < M; ++j, sz <<= 1)
    if(a.tree[j]) {
      if(i < sz)
        break;
```

```
      i -= sz;
    }
  return a.tree[j][i];
}
```

The imperative removal and random mutating algorithms are left as exercises to the reader.

Exercise 11.2

1. Please implement the random access algorithms, including looking up and updating, for binary random access list with numeric representation in your favorite programming language.
2. Prove that the amortized performance of deletion is $O(1)$ constant time by using aggregation analysis.
3. Design and implement the binary random access list by implicit array in your favorite imperative programming language.

11.4 Imperative paired-array list

11.4.1 Definition

In previous chapter about queue, a symmetric solution, naming paired-array is presented. It is capable to operate on both ends of the queue. Because the nature that array supports fast random access. It can be also used to realize a fast random access sequence in imperative setting.



Figure 11.6: A paired-array list, which is consist of 2 arrays linking in head-head manner.

Figure 11.6 shows the design of paired-array list. Two arrays are linked in head-head manner. To insert a new element on the head of the sequence, the element is appended at the end of front array; To append a new element on the tail of the sequence, the element is appended at the end of rear array;

Here is a ISO C++ code snippet to define the this data structure.

```
template<typename Key>
struct List {
  int n, m;
  vector<Key> front;
  vector<Key> rear;

  List() : n(0), m(0) {}
  int size() { return n + m; }
};
```

www.dbooks.org

Here we use vector provides in standard library to cover the dynamic memory management issues, so that we can concentrate on the algorithm design.

11.4.2 Insertion and appending

Suppose function FRONT(L) returns the front array, while REAR(L) returns the rear array. For illustration purpose, we assume the arrays are dynamic allocated. inserting and appending can be realized as the following.

function INSERT(L, x)
  $F \leftarrow$ FRONT(L)
  SIZE(F) $\leftarrow$ SIZE(F) + 1
  F[SIZE(F)] $\leftarrow x$

function APPEND(L, x)
  $R \leftarrow$ REAR(L)
  SIZE(R) $\leftarrow$ SIZE(R) + 1
  R[SIZE(R)] $\leftarrow x$

As all the above operations manipulate the front and rear array on tail, they are all constant $O(1)$ time. And the following are the corresponding ISO C++ programs.

```
template<typename Key>
void insert(List<Key>& xs, Key x) {
  ++xs.n;
  xs.front.push_back(x);
}

template<typename Key>
void append(List<Key>& xs, Key x) {
  ++xs.m;
  xs.rear.push_back(x);
}
```

11.4.3 random access

As the inner data structure is array (dynamic array as vector), which supports random access by nature, it's trivial to implement constant time indexing algorithm.

function GET(L, i)
  $F \leftarrow$ FRONT(L)
  $n \leftarrow$ SIZE(F)
  if $i \leq n$ **then**
    return $F[n - i + 1]$
  else
    return REAR(L)$[i - n]$

Here the index $i \in [1, |L|]$. If it is not greater than the size of front array, the element is stored in front. However, as front and rear arrays are connect head-to-head, so the elements in front array are in reverse order. We need locate the element by subtracting the size of front array by i; If the index i is greater than the size of front array, the element is stored in rear array. Since elements

are stored in normal order in rear, we just need subtract the index i by an offset which is the size of front array.

Here is the ISO C++ program implements this algorithm.

```
template<typename Key>
Key get(List<Key>& xs, int i) {
  if( i < xs.n )
    return xs.front[xs.n-i-1];
  else
    return xs.rear[i-xs.n];
}
```

The random mutating algorithm is left as an exercise to the reader.

11.4.4 removing and balancing

Removing isn't as simple as insertion and appending. This is because we must handle the condition that one array (either front or rear) becomes empty due to removal, while the other still contains elements. In extreme case, the list turns to be quite unbalanced. So we must fix it to resume the balance.

One idea is to trigger this fixing when either front or rear array becomes empty. We just cut the other array in half, and reverse the first half to form the new pair. The algorithm is described as the following.

function BALANCE(L)
  $F \leftarrow$ FRONT(L), $R \leftarrow$ REAR(L)
  $n \leftarrow$ SIZE(F), $m \leftarrow$ SIZE(R)
  if $F = \phi$ **then**
    $F \leftarrow$ REVERSE($R[1 \ldots \lfloor \frac{m}{2} \rfloor]$)
    $R \leftarrow R[\lfloor \frac{m}{2} \rfloor + 1 \ldots m]$
  else if $R = \phi$ **then**
    $R \leftarrow$ REVERSE($F[1 \ldots \lfloor \frac{n}{2} \rfloor]$)
    $F \leftarrow F[\lfloor \frac{n}{2} \rfloor + 1 \ldots n]$

Actually, the operations are symmetric for the case that front is empty and the case that rear is empty. Another approach is to swap the front and rear for one symmetric case and recursive resumes the balance, then swap the front and rear back. For example below ISO C++ program uses this method.

```
template<typename Key>
void balance(List<Key>& xs) {
  if(xs.n == 0) {
    back_insert_iterator<vector<Key> > i(xs.front);
    reverse_copy(xs.rear.begin(), xs.rear.begin() + xs.m/2, i);
    xs.rear.erase(xs.rear.begin(), xs.rear.begin() +xs.m/2);
    xs.n = xs.m/2;
    xs.m -= xs.n;
  } else if(xs.m == 0) {
    swap(xs.front, xs.rear);
    swap(xs.n, xs.m);
    balance(xs);
    swap(xs.front, xs.rear);
    swap(xs.n, xs.m);
  }
}
```

www.dbooks.org

With BALANCE algorithm defined, it's trivial to implement remove algorithm both on head and on tail.

function REMOVE-HEAD(L)
  BALANCE(L)
  $F \leftarrow$ FRONT(L)
  if $F = \phi$ **then**
    REMOVE-TAIL(L)
  else
    SIZE(F) $\leftarrow$ SIZE(F) - 1

function REMOVE-TAIL(L)
  BALANCE(L)
  $R \leftarrow$ REAR(L)
  if $R = \phi$ **then**
    REMOVE-HEAD(L)
  else
    SIZE(R) $\leftarrow$ SIZE(R) - 1

There is an edge case for each, that is even after balancing, the array targeted to perform removal is still empty. This happens that there is only one element stored in the paired-array list. The solution is just remove this singleton left element, and the overall list results empty. Below is the ISO C++ program implements this algorithm.

```
template<typename Key>
void remove_head(List<Key>& xs) {
  balance(xs);
  if(xs.front.empty())
    remove_tail(xs); //remove the singleton elem in rear
  else {
    xs.front.pop_back();
    --xs.n;
  }
}

template<typename Key>
void remove_tail(List<Key>& xs) {
  balance(xs);
  if(xs.rear.empty())
    remove_head(xs); //remove the singleton elem in front
  else {
    xs.rear.pop_back();
    --xs.m;
  }
}
```

It's obvious that the worst case performance is $O(n)$ where n is the number of elements stored in paired-array list. This happens when balancing is triggered, and both reverse and shifting are linear operation. However, the amortized performance of removal is still $O(1)$, the proof is left as exercise to the reader.

Exercise 11.3

1. Implement the random mutating algorithm in your favorite imperative programming language.

2. We utilized vector provided in standard library to manage memory dynamically, try to realize a version using plain array and manage the memory allocation manually. Compare this version and consider how does this affect the performance?

3. Prove that the amortized performance of removal is $O(1)$ for paired-array list.

11.5 Concatenate-able list

By using binary random access list, we realized sequence data structure which supports $O(\lg n)$ time insertion and removal on head, as well as random accessing element with a given index.

However, it's not so easy to concatenate two lists. As both lists are forests of complete binary trees, we can't merely merge them (Since forests are essentially list of trees, and for any size, there is at most one tree of that size. Even concatenate forests directly is not fast). One solution is to push the element from the first sequence one by one to a stack and then pop those elements and insert them to the head of the second one by using 'cons' function. Of course the stack can be implicitly used in recursion manner, for instance:

$$concat(s_1, s_2) = \begin{cases} s_2 & : & s_1 = \phi \\ cons(head(s_1), concat(tail(s_1), s_2)) & : & otherwise \end{cases} \tag{11.16}$$

Where function *cons*, *head* and *tail* are defined in previous section.

If the length of the two sequence is n, and m, this method takes $O(N \lg n)$ time repeatedly push all elements from the first sequence to stacks, and then takes $\Omega(n \lg(n+m))$ to insert the elements in front of the second sequence. Note that Ω means the upper limit, There is detailed definition for it in [2].

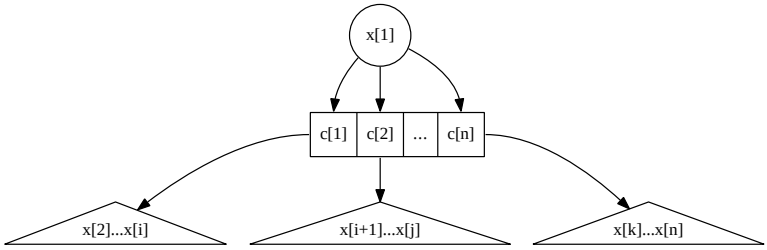
We have already implemented the real-time queue in previous chapter. It supports $O(1)$ time pop and push. If we can turn the sequence concatenation to a kind of pushing operation to queue, the performance will be improved to $O(1)$ as well. Okasaki gave such realization in [6], which can concatenate lists in constant time.

To represent a concatenate-able list, the data structure designed by Okasaki is essentially a K-ary tree. The root of the tree stores the first element in the list. So that we can access it in constant $O(1)$ time. The sub-trees or children are all small concatenate-able lists, which are managed by real-time queues. Concatenating another list to the end is just adding it as the last child, which is in turn a queue pushing operation. Appending a new element can be realized as that, first wrapping the element to a singleton tree, which is a leaf with no children. Then, concatenate this singleton to finalize appending.

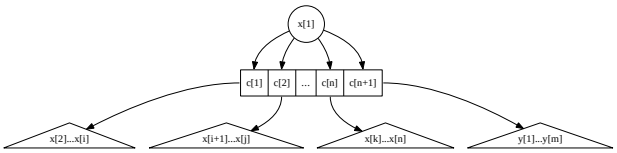
Figure 11.7 illustrates this data structure.

Such recursively designed data structure can be defined in the following Haskell code.

www.dbooks.org



(a) The data structure for list $\{x_1, x_2, ..., x_n\}$



(b) The result after concatenated with list $\{y_1, y_2, ..., y_m\}$

Figure 11.7: Data structure for concatenate-able list

```
data CList a = Empty | CList a (Queue (CList a))
```

It means that a concatenate-able list is either empty or a K-ary tree, which again consists of a queue of concatenate-able sub-lists and a root element. Here we reuse the realization of real-time queue mentioned in previous chapter.

Suppose function $clist(x, Q)$ constructs a concatenate-able list from an element x, and a queue of sub-lists Q. While function $root(s)$ returns the root element of such K-ary tree implemented list. and function $queue(s)$ returns the queue of sub-lists respectively. We can implement the algorithm to concatenate two lists like this.

$$concat(s_1, s_2) = \begin{cases} s_1 & : & s_2 = \phi \\ s_2 & : & s_1 = \phi \\ clist(x, push(Q, s_2)) & : & otherwise \end{cases} \tag{11.17}$$

Where $x = root(s_1)$ and $Q = queue(s_1)$. The idea of concatenation is that if either one of the list to be concatenated is empty, the result is just the other list; otherwise, we push the second list as the last child to the queue of the first list.

Since the push operation is $O(1)$ constant time for a well realized real-time queue, the performance of concatenation is bound to $O(1)$.

The *concat* function can be translated to the below Haskell program.

```
concat x Empty = x
concat Empty y = y
concat (CList x q) y = CList x (push q y)
```

Besides the good performance of concatenation, this design also brings satisfied features for adding element both on head and tail.

$$cons(x, s) = concat(clist(x, \phi), s) \tag{11.18}$$

$$append(s, x) = concat(s, clist(x, \phi)) \tag{11.19}$$

Getting the first element is just returning the root of the K-ary tree.

$$head(s) = root(s) \tag{11.20}$$

It's a bit complex to realize the algorithm that removes the first element from a concatenate-able list. This is because after the root, which is the first element in the sequence got removed, we have to re-construct the rest things, a queue of sub-lists, to a K-ary tree.

After the root being removed, there left all children of the K-ary tree. Note that all of them are also concatenate-able list, so that one natural solution is to concatenate them all together to a big list.

$$concatAll(Q) = \begin{cases} \phi & : & Q = \phi \\ concat(front(Q), concatAll(pop(Q))) & : & otherwise \end{cases} \tag{11.21}$$

Where function $front$ just returns the first element from a queue without removing it, while *pop* does the removing work.

If the queue is empty, it means that there is no children at all, so the result is also an empty list; Otherwise, we pop the first child, which is a concatenate-able list, from the queue, and recursively concatenate all the rest children to a list; finally, we concatenate this list behind the already popped first children.

With *concatAll* defined, we can then implement the algorithm of removing the first element from a list as below.

$$tail(s) = linkAll(queue(s)) \tag{11.22}$$

The corresponding Haskell program is given like the following.

```
head (CList x _) = x
tail (CList _ q) = linkAll q

linkAll q | isEmptyQ q = Empty
          | otherwise = link (front q) (linkAll (pop q))
```

Function `isEmptyQ` is used to test a queue is empty, it is trivial and we omit its definition. Readers can refer to the source code along with this book.

linkAll algorithm actually traverses the queue data structure, and reduces to a final result. This remind us of *folding* mentioned in the chapter of binary search tree. readers can refer to the appendix of this book for the detailed description of folding. It's quite possible to define a folding algorithm for queue instead of list[3] [8].

$$foldQ(f, e, Q) = \begin{cases} e & : & Q = \phi \\ f(front(Q), foldQ(f, e, pop(Q))) & : & otherwise \end{cases} \tag{11.23}$$

Function $foldQ$ takes three parameters, a function f, which is used for reducing, an initial value e, and the queue Q to be traversed.

[3]Some functional programming language, such as Haskell, defined type class, which is a concept of monoid so that it's easy to support folding on a customized data structure.

www.dbooks.org

Here are some examples to illustrate folding on queue. Suppose a queue Q contains elements $\{1, 2, 3, 4, 5\}$ from head to tail.

$$foldQ(+, 0, Q) = 1 + (2 + (3 + (4 + (5 + 0)))) = 15$$
$$foldQ(\times, 1, Q) = 1 \times (2 \times (3 \times (4 \times (5 \times 1)))) = 120$$
$$foldQ(\times, 0, Q) = 1 \times (2 \times (3 \times (4 \times (5 \times 0)))) = 0$$

Function $linkAll$ can be changed by using $foldQ$ accordingly.

$$linkAll(Q) = foldQ(link, \phi, Q) \tag{11.24}$$

The Haskell program can be modified as well.

```
linkAll = foldQ link Empty

foldQ :: (a → b → b) → b → Queue a → b
foldQ f z q | isEmptyQ q = z
            | otherwise = (front q) `f` foldQ f z (pop q)
```

However, the performance of removing can't be ensured in all cases. The worst case is that, user keeps appending n elements to a empty list, and then immediately performs removing. At this time, the K-ary tree has the first element stored in root. There are $n - 1$ children, all are leaves. So $linkAll()$ algorithm downgrades to $O(n)$ which is linear time.

Considering the add, append, concatenate and removing operations are randomly performed. The average case is amortized $O(1)$, The proof is left as en exercise to the reader.

Exercise 11.4

1. Can you figure out a solution to append an element to the end of a binary random access list?

2. Prove that the amortized performance of removal operation for concatenate-able list is $O(1)$. Hint: using the banker's method.

3. Implement the concatenate-able list in your favorite imperative language.

11.6 Finger tree

We haven't been able to meet all the performance targets listed at the beginning of this chapter.

Binary random access list enables to insert, remove element on the head of sequence, and random access elements fast. However, it performs poor when concatenates lists. There is no good way to append element at the end of binary random access list.

Concatenate-able list is capable to concatenates multiple lists in a fly, and it performs well for adding new element both on head and tail. However, it doesn't support randomly access element with a given index.

These two examples bring us some ideas:

- In order to support fast manipulation both on head and tail of the sequence, there must be some way to easily access the head and tail position;
- Tree like data structure helps to turn the random access into divide and conquer search, if the tree is well balance, the search can be ensured to be logarithm time.

11.6.1 Definition

Finger tree[6], which was first invented in 1977, can help to realize efficient sequence. And it is also well implemented in purely functional settings[5].

As we mentioned that the balance of the tree is critical to ensure the performance for search. One option is to use balanced tree as the under ground data structure for finger tree. For example the 2-3 tree, which is a special B-tree. (readers can refer to the chapter of B-tree of this book).

A 2-3 tree either contains 2 children or 3. It can be defined as below in Haskell.

```
data Node a = Br2 a a | Br3 a a a
```

In imperative settings, node can be defined with a list of sub nodes, which contains at most 3 children. For instance the following ANSI C code defines node.

```
union Node {
  Key* keys;
  union Node* children;
};
```

Note in this definition, a node can either contain $2 \sim 3$ keys, or $2 \sim 3$ sub nodes. Where key is the type of elements stored in leaf node.

We mark the left-most none-leaf node as the front finger (or left finger) and the right-most none-leaf node as the rear finger (or right finger). Since both fingers are essentially 2-3 trees with all leafs as children, they can be directly represented as list of 2 or 3 leafs. Of course a finger tree can be empty or contain only one element as leaf.

So the definition of a finger tree is specified like this.

- A finger tree is either empty;
- or a singleton leaf;
- or contains three parts: a left finger which is a list contains at most 3 elements; a sub finger tree; and a right finger which is also a list contains at most 3 elements.

Note that this definition is recursive, so it's quite possible to be translated to functional settings. The following Haskell definition summaries these cases for example.

```
data Tree a = Empty
            | Lf a
            | Tr [a] (Tree (Node a)) [a]
```

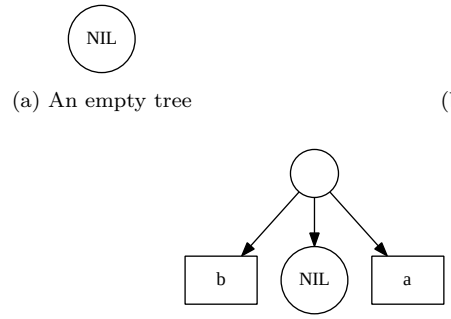
www.dbooks.org

In imperative settings, we can define the finger tree in a similar manner. What's more, we can add a parent field, so that it's possible to back-track to root from any tree node. Below ANSI C code defines finger tree accordingly.

```
struct Tree {
  union Node* front;
  union Node* rear;
  Tree* mid;
  Tree* parent;
};
```

We can use NIL pointer to represent an empty tree; and a leaf tree contains only one element in its front finger, both its rear finger and middle part are empty.

Figure 11.8 and 11.9 show some examples of figure tree.

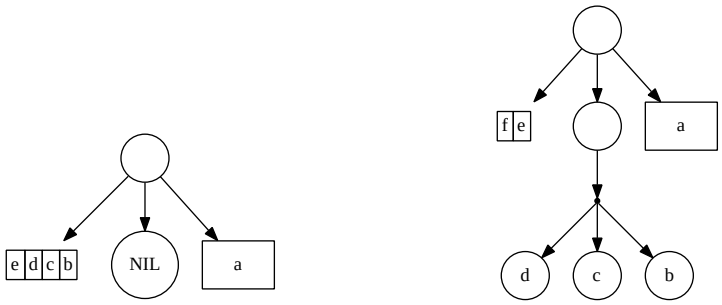


(a) An empty tree

(b) A singleton leaf

(c) Front finger and rear finger contain one element for each, the middle part is empty

Figure 11.8: Examples of finger tree, 1



(a) After inserting extra 3 elements to front finger, it exceeds the 2-3 tree constraint, which isn't balanced any more

(b) The tree resumes balancing. There are 2 elements in front finger; The middle part is a leaf, which contains a 3-branches 2-3 tree.

Figure 11.9: Examples of finger tree, 2

The first example is an empty finger tree; the second one shows the result after inserting one element to empty, it becomes a leaf of one node; the third example shows a finger tree contains 2 elements, one is in front finger, the other is in rear.

If we continuously insert new elements, to the tree, those elements will be put in the front finger one by one, until it exceeds the limit of 2-3 tree. The 4-th example shows such condition, that there are 4 elements in front finger, which isn't balanced any more.

The last example shows that the finger tree gets fixed so that it resumes balancing. There are two elements in the front finger. Note that the middle part is not empty any longer. It's a *leaf* of a 2-3 tree (why it's a leaf is explained later). The content of the leaf is a tree with 3 branches, each contains an element.

We can express these 5 examples as the following Haskell expression.

```
Empty
Lf a
[b] Empty [a]
[e, d, c, b] Empty [a]
[f, e] Lf (Br3 d c b) [a]
```

In the last example, why the middle part inner tree is a leaf? As we mentioned that the definition of finger tree is recursive. The middle part besides the front and rear finger is a deeper finger tree, which is defined as $Tree(Node(a))$. Every time we go deeper, the $Node$ is embedded one more level. if the element type of the first level tree is a, the element type for the second level tree is $Node(a)$, the third level is $Node(Node(a))$, ..., the n-th level is $Node(Node(Node(...(a))...)) = Node^n(a)$, where ^{n} indicates the $Node$ is applied n times.

11.6.2 Insert element to the head of sequence

The examples list above actually reveal the typical process that the elements are inserted one by one to a finger tree. It's possible to summarize these examples to some cases for insertion on head algorithm.

When we insert an element x to a finger tree T,

- If the tree is empty, the result is a leaf which contains the singleton element x;
- If the tree is a singleton leaf of element y, the result is a new finger tree. The front finger contains the new element x, the rear finger contains the previous element y; the middle part is a empty finger tree;
- If the number of elements stored in front finger isn't bigger than the upper limit of 2-3 tree, which is 3, the new element is just inserted to the head of front finger;
- otherwise, it means that the number of elements stored in front finger exceeds the upper limit of 2-3 tree. the last 3 elements in front finger is wrapped in a 2-3 tree and recursively inserted to the middle part. the new element x is inserted in front of the rest elements in front finger.

www.dbooks.org

Suppose that function $leaf(x)$ creates a leaf of element x, function $tree(F, T', R)$ creates a finger tree from three part: F is the front finger, which is a list contains several elements. Similarity, R is the rear finger, which is also a list. T' is the middle part which is a deeper finger tree. Function $tr3(a, b, c)$ creates a 2-3 tree from 3 elements a, b, c; while $tr2(a, b)$ creates a 2-3 tree from 2 elements a and b.

$$insertT(x, T) = \begin{cases} leaf(x) & : & T = \phi \\ tree(\{x\}, \phi, \{y\}) & : & T = leaf(y) \\ tree(\{x, x_1\}, insertT(tr3(x_2, x_3, x_4), T'), R) & : & T = tree(\{x_1, x_2, x_3, x_4\}, T', R) \\ tree(\{x\} \cup F, T', R) & : & otherwise \end{cases} \tag{11.25}$$

The performance of this algorithm is dominated by the recursive case. All the other cases are constant $O(1)$ time. The recursion depth is proportion to the height of the tree, so the algorithm is bound to $O(h)$ time, where h is the height. As we use 2-3 tree to ensure that the tree is well balanced, $h = O(\lg n)$, where n is the number of elements stored in the finger tree.

More analysis reveal that the amortized performance of $insertT$ is $O(1)$ because we can amortize the expensive recursion case to other trivial cases. Please refer to [6] and [5] for the detailed proof.

Translating the algorithm yields the below Haskell program.

```
cons :: a → Tree a → Tree a
cons a Empty = Lf a
cons a (Lf b) = Tr [a] Empty [b]
cons a (Tr [b, c, d, e] m r) = Tr [a, b] (cons (Br3 c d e) m) r
cons a (Tr f m r) = Tr (a:f) m r
```

Here we use the LISP naming convention to illustrate inserting a new element to a list.

The insertion algorithm can also be implemented in imperative approach. Suppose function TREE() creates an empty tree, that all fields, including front and rear finger, the middle part inner tree and parent are empty. Function NODE() creates an empty node.

function PREPEND-NODE(n, T)
  $r \leftarrow$ TREE()
  $p \leftarrow r$
  CONNECT-MID(p, T)
  while FULL?(FRONT(T)) **do**
    $F \leftarrow$ FRONT(T) ▷ $F = \{n_1, n_2, n_3, ...\}$
    FRONT(T) $\leftarrow \{n, F[1]\}$ ▷ $F[1] = n_1$
    $n \leftarrow$ NODE()
    CHILDREN(n) $\leftarrow F[2..]$ ▷ $F[2..] = \{n_2, n_3, ...\}$
    $p \leftarrow T$
    $T \leftarrow$ MID(T)
  if $T =$ NIL **then**
    $T \leftarrow$ TREE()
    FRONT(T)$\leftarrow \{n\}$
  else if $|$ FRONT(T) $| = 1 \wedge$ REAR(T) $= \phi$ **then**
    REAR(T) $\leftarrow$ FRONT(T)

```
        FRONT(T) ← {n}
    else
        FRONT(T) ← {n} ∪ FRONT(T)
    CONNECT-MID(p, T) ← T
    return FLAT(r)
```

Where the notation $L[i..]$ means a sub list of L with the first $i-1$ elements removed, that if $L = \{a_1, a_2, ..., a_n\}$, then $L[i..] = \{a_i, a_{i+1}, ..., a_n\}$.

Functions FRONT, REAR, MID, and PARENT are used to access the front finger, the rear finger, the middle part inner tree and the parent tree respectively; Function CHILDREN accesses the children of a node.

Function CONNECT-MID(T_1, T_2), connect T_2 as the inner middle part tree of T_1, and set the parent of T_2 as T_1 if T_2 isn't empty.

In this algorithm, we performs a one pass top-down traverse along the middle part inner tree if the front finger is full that it can't afford to store any more. The criteria for full for a 2-3 tree is that the finger contains 3 elements already. In such case, we extract all the elements except the first one off, wrap them to a new node (one level deeper node), and continuously insert this new node to its middle inner tree. The first element is left in the front finger, and the element to be inserted is put in front of it, so that this element becomes the new first one in the front finger.

After this traversal, the algorithm either reach an empty tree, or the tree still has room to hold more element in its front finger. We create a new leaf for the former case, and perform a trivial list insert to the front finger for the latter.

During the traversal, we use p to record the parent of the current tree we are processing. So any new created tree are connected as the middle part inner tree to p.

Finally, we return the root of the tree r. The last trick of this algorithm is the FLAT function. In order to simplify the logic, we create an empty 'ground' tree and set it as the parent of the root. We need eliminate this extra 'ground' level before return the root. This flatten algorithm is realized as the following.

```
function FLAT(T)
    while T ≠ NIL ∧ T is empty do
        T ← MID(T)
    if T ≠ NIL then
        PARENT(T) ← NIL
    return T
```

The while loop test if T is trivial empty, that it's not NIL$(= \phi)$, while both its front and rear fingers are empty.

Below Python code implements the insertion algorithm for finger tree.

```
def insert(x, t):
    return prepend_node(wrap(x), t)

def prepend_node(n, t):
    root = prev = Tree()
    prev.set_mid(t)
    while frontFull(t):
        f = t.front
        t.front = [n] + f[:1]
```

www.dbooks.org

```
            n = wraps(f[1:])
            prev = t
            t = t.mid
        if t is None:
            t = leaf(n)
        elif len(t.front)==1 and t.rear == []:
            t = Tree([n], None, t.front)
        else:
            t = Tree([n]+t.front, t.mid, t.rear)
        prev.set_mid(t)
        return flat(root)

def flat(t):
    while t is not None and t.empty():
        t = t.mid
    if t is not None:
        t.parent = None
    return t
```

The implementation of function `set_mid`, `frontFull`, `wrap`, `wraps`, `empty`, and tree constructor are trivial enough, that we skip the detail of them here. Readers can take these as exercises.

11.6.3 Remove element from the head of sequence

It's easy to implement the reverse operation that remove the first element from the list by reversing the $insertT()$ algorithm line by line.

Let's denote $F = \{f_1, f_2, ...\}$ is the front finger list, M is the middle part inner finger tree. $R = \{r_1, r_2, ...\}$ is the rear finger list of a finger tree, and $R' = \{r_2, r_3, ...\}$ is the rest of element with the first one removed from R.

$$extractT(T) = \begin{cases} (x, \phi) & : & T = leaf(x) \\ (x, leaf(y)) & : & T = tree(\{x\}, \phi, \{y\}) \\ (x, tree(\{r_1\}, \phi, R')) & : & T = tree(\{x\}, \phi, R) \\ (x, tree(toList(F'), M', R)) & : & T = tree(\{x\}, M, R), (F', M') = extractT(M) \\ (f_1, tree(\{f_2, f_3, ...\}, M, R)) & : & otherwise \end{cases} \tag{11.26}$$

Where function $toList(T)$ converts a 2-3 tree to plain list as the following.

$$toList(T) = \begin{cases} \{x, y\} & : & T = tr2(x, y) \\ \{x, y, z\} & : & T = tr3(x, y, z) \end{cases} \tag{11.27}$$

Here we skip the error handling such as trying to remove element from empty tree etc. If the finger tree is a leaf, the result after removal is an empty tree; If the finger tree contains two elements, one in the front finger, the other in rear, we return the element stored in front finger as the first element, and the resulted tree after removal is a leaf; If there is only one element in front finger, the middle part inner tree is empty, and the rear finger isn't empty, we return the only element in front finger, and borrow one element from the rear finger to front; If there is only one element in front finger, however, the middle part inner tree isn't empty, we can recursively remove a node from the inner tree, and flatten it to a plain list to replace the front finger, and remove the original

only element in front finger; The last case says that if the front finger contains more than one element, we can just remove the first element from front finger and keep all the other part unchanged.

Figure 11.10 shows the steps of removing two elements from the head of a sequence. There are 10 elements stored in the finger tree. When the first element is removed, there is still one element left in the front finger. However, when the next element is removed, the front finger is empty. So we 'borrow' one tree node from the middle part inner tree. This is a 2-3 tree. it is converted to a list of 3 elements, and the list is used as the new finger. the middle part inner tree change from three parts to a singleton leaf, which contains only one 2-3 tree node. There are three elements stored in this tree node.

Below is the corresponding Haskell program for 'uncons'.

```
uncons :: Tree a → (a, Tree a)
uncons (Lf a) = (a, Empty)
uncons (Tr [a] Empty [b]) = (a, Lf b)
uncons (Tr [a] Empty (r:rs)) = (a, Tr [r] Empty rs)
uncons (Tr [a] m r) = (a, Tr (nodeToList f) m' r) where (f, m') = uncons m
uncons (Tr f m r) = (head f, Tr (tail f) m r)
```

And the function `nodeToList` is defined like this.

```
nodeToList :: Node a → [a]
nodeToList (Br2 a b) = [a, b]
nodeToList (Br3 a b c) = [a, b, c]
```

Similar as above, we can define `head` and `tail` function from `uncons`.

```
head = fst ∘ uncons
tail = snd ∘ uncons
```

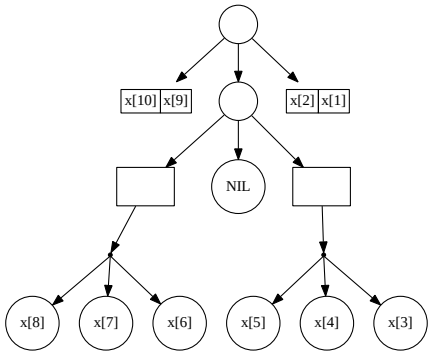
11.6.4 Handling the ill-formed finger tree when removing

The strategy used so far to remove element from finger tree is a kind of removing and borrowing. If the front finger becomes empty after removing, we borrows more nodes from the middle part inner tree. However there exists cases that the tree is ill-formed, for example, both the front fingers of the tree and its middle part inner tree are empty. Such ill-formed tree can result from imperatively splitting, which we'll introduce later.

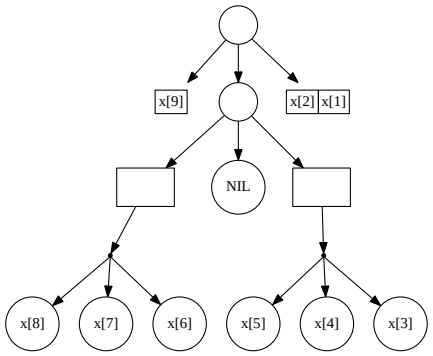
Here we developed an imperative algorithm which can remove the first element from finger tree even it is ill-formed. The idea is first perform a top-down traverse to find a sub tree which either has a non-empty front finger or both its front finger and middle part inner tree are empty. For the former case, we can safely extract the first element which is a node from the front finger; For the latter case, since only the rear finger isn't empty, we can swap it with the empty front finger, and change it to the former case.

After that, we need examine the node we extracted from the front finger is leaf node (How to do that? this is left as an exercise to the reader). If not, we need go on extracting the first sub node from the children of this node, and left the rest of other children as the new front finger to the parent of the current tree. We need repeatedly go up along with the parent field till the node we extracted is a leaf. At that time point, we arrive at the root of the tree. Figure 11.12 illustrates this process.

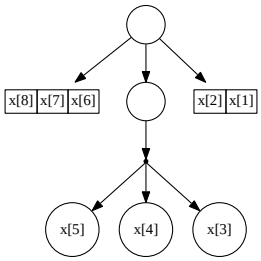
www.dbooks.org



(a) A sequence of 10 elements represented as a finger tree



(b) The first element is removed. There is one element left in front finger.



(c) Another element is remove from head. We borrowed one node from the middle part inner tree, change the node, which is a 2-3 tree to a list, and use it as the new front finger. the middle part inner tree becomes a leaf of one 2-3 tree node.

Figure 11.10: Examples of remove 2 elements from the head of a sequence

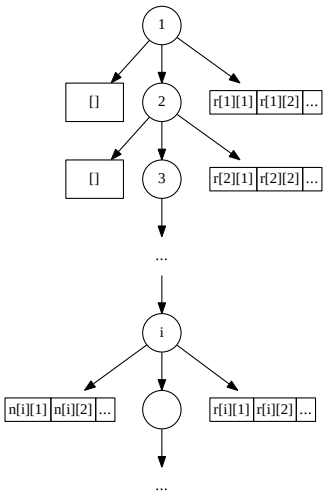
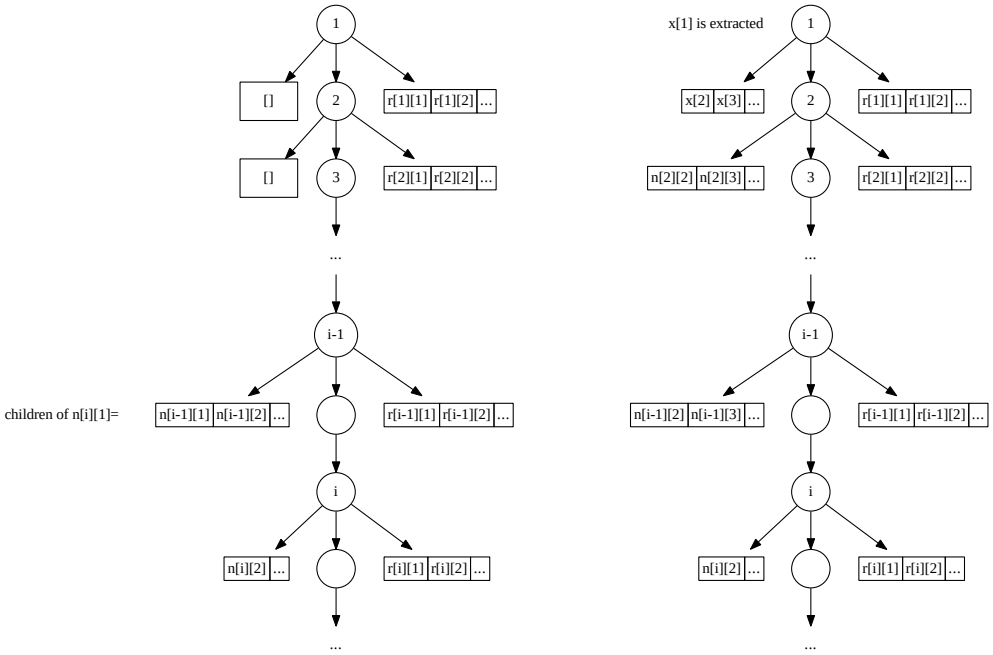


Figure 11.11: Example of an ill-formed tree. The front finger of the i-th level sub tree isn't empty.



(a) Extract the first element n[i][1] and put its children to the front finger of upper level tree.

(b) Repeat this process i times, and finally x[1] is extracted.

Figure 11.12: Traverse bottom-up till a leaf is extracted.

www.dbooks.org

Based on this idea, the following algorithm realizes the removal operation on head. The algorithm assumes that the tree passed in isn't empty.

function EXTRACT-HEAD(T)
  $r \leftarrow$ TREE()
  CONNECT-MID(r, T)
  while FRONT(T) $= \phi \wedge$ MID(T) $\neq$ NIL **do**
    $T \leftarrow$ MID(T)
  if FRONT(T) $= \phi \wedge$ REAR(T) $\neq \phi$ **then**
    EXCHANGE FRONT(T) $\leftrightarrow$ REAR(T)
  $n \leftarrow$ NODE()
  CHILDREN(n) $\leftarrow$ FRONT(T)
  repeat
    $L \leftarrow$ CHILDREN(n) ▷ $L = \{n_1, n_2, n_3, ...\}$
    $n \leftarrow L[1]$ ▷ $n \leftarrow n_1$
    FRONT(T) $\leftarrow L[2..]$ ▷ $L[2..] = \{n_2, n_3, ...\}$
    $T \leftarrow$ PARENT(T)
    if MID(T) becomes empty **then**
      MID(T) $\leftarrow$ NIL
  until n is a leaf
  return (ELEM(n), FLAT(r))

Note that function ELEM(n) returns the only element stored inside leaf node n. Similar as imperative insertion algorithm, a stub 'ground' tree is used as the parent of the root, which can simplify the logic a bit. That's why we need flatten the tree finally.

Below Python program translates the algorithm.

```
def extract_head(t):
    root = Tree()
    root.set_mid(t)
    while t.front == [] and t.mid is not None:
        t = t.mid
    if t.front == [] and t.rear != []:
        (t.front, t.rear) = (t.rear, t.front)
    n = wraps(t.front)
    while True: # a repeat-until loop
        ns = n.children
        n = ns[0]
        t.front = ns[1:]
        t = t.parent
        if t.mid.empty():
            t.mid.parent = None
            t.mid = None
        if n.leaf:
            break
    return (elem(n), flat(root))
```

Member function `Tree.empty()` returns true if both the front finger and the rear finger are empty. We put a flag `Node.leaf` to mark if a node is a leaf or compound node. The exercise of this section asks the reader to consider some alternatives.

As the ill-formed tree is allowed, the algorithms to access the first and last

element of the finger tree must be modified, so that they don't blindly return the first or last child of the finger as the finger can be empty if the tree is ill-formed.

The idea is quite similar to the EXTRACT-HEAD, that in case the finger is empty while the middle part inner tree isn't, we need traverse along with the inner tree till a point that either the finger becomes non-empty or all the nodes are stored in the other finger. For instance, the following algorithm can return the first leaf node even the tree is ill-formed.

function FIRST-LF(T)
  while FRONT(T) $= \phi \wedge$ MID(T) $\neq$ NIL **do**
    $T \leftarrow$ MID(T)
  if FRONT(T) $= \phi \wedge$ REAR(T) $\neq \phi$ **then**
    $n \leftarrow$ REAR(T)[1]
  else
    $n \leftarrow$ FRONT(T)[1]
  while n is NOT leaf **do**
    $n \leftarrow$ CHILDREN(n)[1]
  return n

Note the second loop in this algorithm that it keeps traversing on the first sub-node if current node isn't a leaf. So we always get a leaf node and it's trivial to get the element inside it.

function FIRST(T)
  return ELEM(FIRST-LF(T))

The following Python code translates the algorithm to real program.

```
def first(t):
    return elem(first_leaf(t))

def first_leaf(t):
    while t.front == [] and t.mid is not None:
        t = t.mid
    if t.front == [] and t.rear != []:
        n = t.rear[0]
    else:
        n = t.front[0]
    while not n.leaf:
        n = n.children[0]
    return n
```

To access the last element is quite similar, and we left it as an exercise to the reader.

www.dbooks.org

11.6.5 append element to the tail of the sequence

Because finger tree is symmetric, we can give the realization of appending element on tail by referencing to $insertT$ algorithm.

$$appendT(T, x) = \begin{cases} leaf(x) & : & T = \phi \\ tree(\{y\}, \phi, \{x\}) & : & T = leaf(y) \\ tree(F, appendT(M, tr3(x_1, x_2, x_3)), \{x_4, x\}) & : & T = tree(F, M, \{x_1, x_2, x_3, x_4\}) \\ tree(F, M, R \cup \{x\}) & : & otherwise \end{cases} \tag{11.28}$$

Generally speaking, if the rear finger is still valid 2-3 tree, that the number of elements is not greater than 4, the new elements is directly appended to rear finger. Otherwise, we break the rear finger, take the first 3 elements in rear finger to create a new 2-3 tree, and recursively append it to the middle part inner tree. If the finger tree is empty or a singleton leaf, it will be handled in the first two cases.

Translating the equation to Haskell yields the below program.

```
snoc :: Tree a → a → Tree a
snoc Empty a = Lf a
snoc (Lf a) b = Tr [a] Empty [b]
snoc (Tr f m [a, b, c, d]) e = Tr f (snoc m (Br3 a b c)) [d, e]
snoc (Tr f m r) a = Tr f m (r++[a])
```

Function name `snoc` is mirror of `cons`, which indicates the symmetric relationship.

Appending new element to the end imperatively is quite similar. The following algorithm realizes appending.

function APPEND-NODE(T, n)
  $r \leftarrow$ TREE()
  $p \leftarrow r$
  CONNECT-MID(p, T)
  while FULL?(REAR(T)) **do**
    $R \leftarrow$ REAR(T) ▷ $R = \{n_1, n_2, ..., , n_{m-1}, n_m\}$
    REAR(T) $\leftarrow \{n,$ LAST(R) $\}$ ▷ last element n_m
    $n \leftarrow$ NODE()
    CHILDREN(n) $\leftarrow R[1...m-1]$ ▷ $\{n1, n2, ..., n_{m-1}\}$
    $p \leftarrow T$
    $T \leftarrow$ MID(T)
  if $T =$ NIL **then**
    $T \leftarrow$ TREE()
    FRONT(T) $\leftarrow \{n\}$
  else if $|$ REAR(T) $| = 1 \wedge$ FRONT(T) $= \phi$ **then**
    FRONT(T) $\leftarrow$ REAR(T)
    REAR(T) $\leftarrow \{n\}$
  else
    REAR(T) $\leftarrow$ REAR(T) $\cup\{n\}$
  CONNECT-MID(p, T) $\leftarrow T$
  return FLAT(r)

And the corresponding Python programs is given as below.

```
def append_node(t, n):
    root = prev = Tree()
    prev.set_mid(t)
    while rearFull(t):
        r = t.rear
        t.rear = r[-1:] + [n]
        n = wraps(r[:-1])
        prev = t
        t = t.mid
    if t is None:
        t = leaf(n)
    elif len(t.rear) == 1 and t.front == []:
        t = Tree(t.rear, None, [n])
    else:
        t = Tree(t.front, t.mid, t.rear + [n])
    prev.set_mid(t)
    return flat(root)
```

11.6.6 remove element from the tail of the sequence

Similar to $appendT$, we can realize the algorithm which remove the last element from finger tree in symmetric manner of $extractT$.

We denote the non-empty, non-leaf finger tree as $tree(F, M, R)$, where F is the front finger, M is the middle part inner tree, and R is the rear finger.

$$removeT(T) = \begin{cases} (\phi, x) & : & T = leaf(x) \\ (leaf(y), x) & : & T = tree(\{y\}, \phi, \{x\}) \\ (tree(init(F), \phi, last(F)), x) & : & T = tree(F, \phi, \{x\}) \wedge F \neq \phi \\ (tree(F, M', toList(R')), x) & : & T = tree(F, M, \{x\}), (M', R') = removeT(M) \\ (tree(F, M, init(R)), last(R)) & : & otherwise \end{cases} \tag{11.29}$$

Function $toList(T)$ is used to flatten a 2-3 tree to plain list, which is defined previously. Function $init(L)$ returns all elements except for the last one in list L, that if $L = \{a_1, a_2, ..., a_{n-1}, a_n\}$, $init(L) = \{a_1, a_2, ..., a_{n-1}\}$. And Function $last(L)$ returns the last element, so that $last(L) = a_n$. Please refer to the appendix of this book for their implementation.

Algorithm $removeT()$ can be translated to the following Haskell program, we name it as `unsnoc` to indicate it's the reverse function of `snoc`.

```
unsnoc :: Tree a → (Tree a, a)
unsnoc (Lf a) = (Empty, a)
unsnoc (Tr [a] Empty [b]) = (Lf a, b)
unsnoc (Tr f@(_:_) Empty [a]) = (Tr (init f) Empty [last f], a)
unsnoc (Tr f m [a]) = (Tr f m' (nodeToList r), a) where (m', r) = unsnoc m
unsnoc (Tr f m r) = (Tr f m (init r), last r)
```

And we can define a special function `last` and `init` for finger tree which is similar to their counterpart for list.

```
last = snd ∘ unsnoc
init = fst ∘ unsnoc
```

www.dbooks.org

Imperatively removing the element from the end is almost as same as removing on the head. Although there seems to be a special case, that as we always store the only element (or sub node) in the front finger while the rear finger and middle part inner tree are empty (e.g. $Tree(\{n\}, NIL, \phi)$), it might get nothing if always try to fetch the last element from rear finger.

This can be solved by swapping the front and the rear finger if the rear is empty as in the following algorithm.

```
function EXTRACT-TAIL(T)
    r ← TREE()
    CONNECT-MID(r, T)
    while REAR(T) = φ∧ MID(T) ≠ NIL do
        T ← MID(T)
    if REAR(T) = φ∧ FRONT(T) ≠ φ then
        EXCHANGE FRONT(T) ↔ REAR(T)
    n ← NODE()
    CHILDREN(n) ← REAR(T)
    repeat
        L ← CHILDREN(n)                    ▷ L = {n_1, n_2, ..., n_{m-1}, n_m}
        n ← LAST(L)                        ▷ n ← n_m
        REAR(T) ← L[1...m − 1]             ▷ {n_1, n_2, ..., n_{m-1}}
        T ← PARENT(T)
        if MID(T) becomes empty then
            MID(T) ← NIL
    until n is a leaf
    return (ELEM(n), FLAT(r))
```

How to access the last element as well as implement this algorithm to working program are left as exercises.

11.6.7 concatenate

Consider the none-trivial case that concatenate two finger trees $T_1 = tree(F_1, M_1, R_1)$ and $T_2 = tree(F_2, M_2, R_2)$. One natural idea is to use F_1 as the new front finger for the concatenated result, and keep R_2 being the new rear finger. The rest of work is to merge M_1, R_1, F_2 and M_2 to a new middle part inner tree.

Note that both R_1 and F_2 are plain lists of node, so the sub-problem is to realize a algorithm like this.

$$merge(M_1, R_1 \cup F_2, M_2) =?$$

More observation reveals that both M_1 and M_2 are also finger trees, except that they are one level deeper than T_1 and T_2 in terms of $Node(a)$, where a is the type of element stored in the tree. We can recursively use the strategy that keep the front finger of M_1 and the rear finger of M_2, then merge the middle part inner tree of M_1, M_2, as well as the rear finger of M_1 and front finger of M_2.

If we denote function $front(T)$ returns the front finger, $rear(T)$ returns the rear finger, $mid(T)$ returns the middle part inner tree. the above *merge*

algorithm can be expressed for non-trivial case as the following.

$$\begin{aligned} &merge(M_1, R_1 \cup F_2, M_2) = tree(front(M_1), S, rear(M_2)) \\ &S = merge(mid(M_1), rear(M_1) \cup R_1 \cup F_2 \cup front(M_2), mid(M_2)) \end{aligned} \tag{11.30}$$

If we look back to the original concatenate solution, it can be expressed as below.

$$concat(T_1, T_2) = tree(F_1, merge(M_1, R_1 \cup F_2, M_2), R_2) \tag{11.31}$$

And compare it with equation 11.30, it's easy to note the fact that concatenating is essentially merging. So we have the final algorithm like this.

$$concat(T_1, T_2) = merge(T_1, \phi, T_2) \tag{11.32}$$

By adding edge cases, the $merge()$ algorithm can be completed as below.

$$merge(T_1, S, T_2) = \begin{cases} foldR(insertT, T_2, S) & : & T_1 = \phi \\ foldL(appendT, T_1, S) & : & T_2 = \phi \\ merge(\phi, \{x\} \cup S, T_2) & : & T_1 = leaf(x) \\ merge(T_1, S \cup \{x\}, \phi) & : & T_2 = leaf(x) \\ tree(F_1, merge(M_1, nodes(R_1 \cup S \cup F_2), M2), R_2) & : & otherwise \end{cases} \tag{11.33}$$

Most of these cases are straightforward. If any one of T_1 or T_2 is empty, the algorithm repeatedly insert/append all elements in S to the other tree; Function $foldL$ and $foldR$ are kinds of for-each process in imperative settings. The difference is that $foldL$ processes the list S from left to right while $foldR$ processes from right to left.

Here are their definition. Suppose list $L = \{a_1, a_2, ..., a_{n-1}, a_n\}$, $L' = \{a_2, a_3, ..., a_{n-1}, a_n\}$ is the rest of elements except for the first one.

$$foldL(f, e, L) = \begin{cases} e & : & L = \phi \\ foldL(f, f(e, a_1), L') & : & otherwise \end{cases} \tag{11.34}$$

$$foldR(f, e, L) = \begin{cases} e & : & L = \phi \\ f(a_1, foldR(f, e, L')) & : & otherwise \end{cases} \tag{11.35}$$

They are detailed explained in the appendix of this book.

If either one of the tree is a leaf, we can insert or append the element of this leaf to S, so that it becomes the trivial case of concatenating one empty tree with another.

Function *nodes* is used to wrap a list of elements to a list of 2-3 trees. This is because the contents of middle part inner tree, compare to the contents of finger, are one level deeper in terms of $Node$. Consider the time point that transforms from recursive case to edge case. Let's suppose M_1 is empty at that time, we then need repeatedly insert all elements from $R_1 \cup S \cup F_2$ to M_2. However, we can't directly do the insertion. If the element type is a, we can only insert $Node(a)$ which is 2-3 tree to M_2. This is just like what we did in the $insertT$

www.dbooks.org

algorithm, take out the last 3 elements, wrap them in a 2-3 tree, and recursive perform $insertT$. Here is the definition of $nodes$.

$$nodes(L) = \begin{cases} \{tr2(x_1, x_2)\} & : & L = \{x_1, x_2\} \\ \{tr3(x_1, x_2, x_3)\} & : & L = \{x_1, x_2, x_3\} \\ \{tr2(x_1, x_2), tr2(x_3, x_4)\} & : & L = \{x_1, x_2, x_3, x_4\} \\ \{tr3(x_1, x_2, x_3)\} \cup nodes(\{x_4, x_5, ...\}) & : & otherwise \end{cases} \tag{11.36}$$

Function $nodes$ follows the constraint of 2-3 tree, that if there are only 2 or 3 elements in the list, it just wrap them in singleton list contains a 2-3 tree; If there are 4 elements in the lists, it split them into two trees each is consist of 2 branches; Otherwise, if there are more elements than 4, it wraps the first three in to one tree with 3 branches, and recursively call $nodes$ to process the rest.

The performance of concatenation is determined by merging. Analyze the recursive case of merging reveals that the depth of recursion is proportion to the smaller height of the two trees. As the tree is ensured to be balanced by using 2-3 tree. it's height is bound to $O(\lg n')$ where n' is the number of elements. The edge case of merging performs as same as insertion, (It calls $insertT$ at most 8 times) which is amortized $O(1)$ time, and $O(\lg m)$ at worst case, where m is the difference in height of the two trees. So the overall performance is bound to $O(\lg n)$, where n is the total number of elements contains in two finger trees.

The following Haskell program implements the concatenation algorithm.

```
concat :: Tree a → Tree a → Tree a
concat t1 t2 = merge t1 [] t2
```

Note that there is `concat` function defined in prelude standard library, so we need distinct them either by hiding import or take a different name.

```
merge :: Tree a → [a] → Tree a → Tree a
merge Empty ts t2 = foldr cons t2 ts
merge t1 ts Empty = foldl snoc t1 ts
merge (Lf a) ts t2 = merge Empty (a:ts) t2
merge t1 ts (Lf a) = merge t1 (ts++[a]) Empty
merge (Tr f1 m1 r1) ts (Tr f2 m2 r2) = Tr f1 (merge m1 (nodes (r1 ++ ts ++ f2)) m2) r2
```

And the implementation of $nodes$ is as below.

```
nodes :: [a] → [Node a]
nodes [a, b] = [Br2 a b]
nodes [a, b, c] = [Br3 a b c]
nodes [a, b, c, d] = [Br2 a b, Br2 c d]
nodes (a:b:c:xs) = Br3 a b c:nodes xs
```

To concatenate two finger trees T_1 and T_2 in imperative approach, we can traverse the two trees along with the middle part inner tree till either tree turns to be empty. In every iteration, we create a new tree T, choose the front finger of T_1 as the front finger of T; and choose the rear finger of T_2 as the rear finger of T. The other two fingers (rear finger of T_1 and front finger of T_2) are put together as a list, and this list is then balanced grouped to several 2-3 tree nodes as N. Note that N grows along with traversing not only in terms of length, the depth of its elements increases by one in each iteration. We attach this new tree as the middle part inner tree of the upper level result tree to end this iteration.

Once either tree becomes empty, we stop traversing, and repeatedly insert the 2-3 tree nodes in N to the other non-empty tree, and set it as the new middle part inner tree of the upper level result.

Below algorithm describes this process in detail.

function CONCAT(T_1, T_2)
  return MERGE(T_1, ϕ, T_2)

function MERGE(T_1, N, T_2)
  $r \leftarrow$ TREE()
  $p \leftarrow r$
  while $T_1 \neq$ NIL $\wedge T_2 \neq$ NIL **do**
    $T \leftarrow$ TREE()
    FRONT(T) $\leftarrow$ FRONT(T_1)
    REAR(T) $\leftarrow$ REAR(T_2)
    CONNECT-MID(p, T)
    $p \leftarrow T$
    $N \leftarrow$ NODES(REAR(T_1) $\cup n \cup$ FRONT(T_2))
    $T_1 \leftarrow$ MID(T_1)
    $T_2 \leftarrow$ MID(T_2)
  if $T_1 =$ NIL **then**
    $T \leftarrow T_2$
    for each $n \in$ REVERSE(N) **do**
      $T \leftarrow$ PREPEND-NODE(n, T)
  else if $T_2 =$ NIL **then**
    $T \leftarrow T_1$
    for each $n \in N$ **do**
      $T \leftarrow$ APPEND-NODE(T, n)
  CONNECT-MID(p, T)
  return FLAT(r)

Note that the for-each loops in the algorithm can also be replaced by folding from left and right respectively. Translating this algorithm to Python program yields the below code.

```
def concat(t1, t2):
    return merge(t1, [], t2)

def merge(t1, ns, t2):
    root = prev = Tree() #sentinel dummy tree
    while t1 is not None and t2 is not None:
        t = Tree(t1.size + t2.size + sizeNs(ns), t1.front, None, t2.rear)
        prev.set_mid(t)
        prev = t
        ns = nodes(t1.rear + ns + t2.front)
        t1 = t1.mid
        t2 = t2.mid
    if t1 is None:
        prev.set_mid(foldR(prepend_node, ns, t2))
    elif t2 is None:
        prev.set_mid(reduce(append_node, ns, t1))
    return flat(root)
```

www.dbooks.org

Because Python only provides folding function from left as `reduce()`, a folding function from right is given like what we shown in the following code, that it repeatedly applies function in reverse order of the list.

```
def foldR(f, xs, z):
    for x in reversed(xs):
        z = f(x, z)
    return z
```

The only function in question is how to balanced-group nodes to bigger 2-3 trees. As a 2-3 tree can hold at most 3 sub trees, we can firstly take 3 nodes and wrap them to a ternary tree if there are more than 4 nodes in the list and continuously deal with the rest. If there are just 4 nodes, they can be wrapped to two binary trees. For other cases (there are 3 nodes, 2 nodes, 1 node), we simply wrap them all to a tree.

Denote node list $L = \{n_1, n_2, ...\}$, The following algorithm realizes this process.

function NODES(L)
  $N = \phi$
  while $|L| > 4$ **do**
    $n \leftarrow$ NODE()
    CHILDREN(n) $\leftarrow L[1..3]$ ▷ $\{n_1, n_2, n_3\}$
    $N \leftarrow N \cup \{n\}$
    $L \leftarrow L[4...]$ ▷ $\{n_4, n_5, ...\}$
  if $|L| = 4$ **then**
    $x \leftarrow$ NODE()
    CHILDREN(x) $\leftarrow \{L[1], L[2]\}$
    $y \leftarrow$ NODE()
    CHILDREN(y) $\leftarrow \{L[3], L[4]\}$
    $N \leftarrow N \cup \{x, y\}$
  else if $L \neq \phi$ **then**
    $n \leftarrow$ NODE()
    CHILDREN(n) $\leftarrow L$
    $N \leftarrow N \cup \{n\}$
  return N

It's straight forward to translate the algorithm to below Python program. Where function `wraps()` helps to create an empty node, then set a list as the children of this node.

```
def nodes(xs):
    res = []
    while len(xs) > 4:
        res.append(wraps(xs[:3]))
        xs = xs[3:]
    if len(xs) == 4:
        res.append(wraps(xs[:2]))
        res.append(wraps(xs[2:]))
    elif xs != []:
        res.append(wraps(xs))
    return res
```

Exercise 11.5

1. Implement the complete finger tree insertion program in your favorite imperative programming language. Don't check the example programs along with this chapter before having a try.

2. How to determine a node is a leaf? Does it contain only a raw element inside or a compound node, which contains sub nodes as children? Note that we can't distinguish it by testing the size, as there is case that node contains a singleton leaf, such as $node(1, \{node(1, \{x\}\})$. Try to solve this problem in both dynamic typed language (e.g. Python, lisp etc) and in strong static typed language (e.g. C++).

3. Implement the EXTRACT-TAIL algorithm in your favorite imperative programming language.

4. Realize algorithm to return the last element of a finger tree in both functional and imperative approach. The later one should be able to handle ill-formed tree.

5. Try to implement concatenation algorithm without using folding. You can either use recursive methods, or use imperative for-each method.

11.6.8 Random access of finger tree

size augmentation

The strategy to provide fast random access, is to turn the looking up into tree-search. In order to avoid calculating the size of tree many times, we augment an extra field to tree and node. The definition should be modified accordingly, for example the following Haskell definition adds size field in its constructor.

```
data Tree a = Empty
            | Lf a
            | Tr Int [a] (Tree (Node a)) [a]
```

And the previous ANSI C structure is augmented with size as well.

```
struct Tree {
  union Node* front;
  union Node* rear;
  Tree* mid;
  Tree* parent;
  int size;
};
```

Suppose the function $tree(s, F, M, R)$ creates a finger tree from size s, front finger F, rear finger R, and middle part inner tree M. When the size of the tree is needed, we can call a $size(T)$ function. It will be something like this.

$$size(T) = \begin{cases} 0 & : \quad T = \phi \\ ? & : \quad T = leaf(x) \\ s & : \quad T = tree(s, F, M, R) \end{cases}$$

If the tree is empty, the size is definitely zero; and if it can be expressed as $tree(s, F, M, R)$, the size is s; however, what if the tree is a singleton leaf? is it 1? No, it can be 1 only if $T = leaf(a)$ and a isn't a tree node, but a raw

www.dbooks.org

element stored in finger tree. In most cases, the size is not 1, because a can be again a tree node. That's why we put a '?' in above equation.

The correct way is to call some size function on the tree node as the following.

$$size(T) = \begin{cases} 0 & : \quad T = \phi \\ size'(x) & : \quad T = leaf(x) \\ s & : \quad T = tree(s, F, M, R) \end{cases} \tag{11.37}$$

Note that this isn't a recursive definition since $size \neq size'$, the argument to $size'$ is either a tree node, which is a 2-3 tree, or a plain element stored in the finger tree. To uniform these two cases, we can anyway wrap the single plain element to a tree node of only one element. So that we can express all the situation as a tree node augmented with a size field. The following Haskell program modifies the definition of tree node.

```
data Node a = Br Int [a]
```

The ANSI C node definition is modified accordingly.

```
struct Node {
  Key key;
  struct Node* children;
  int size;
};
```

We change it from union to structure. Although there is a overhead field 'key' if the node isn't a leaf.

Suppose function $tr(s, L)$, creates such a node (either one element being wrapped or a 2-3 tree) from a size information s, and a list L. Here are some example.

$tr(1, \{x\})$	a tree contains only one element
$tr(2, \{x, y\})$	a 2-3 tree contains two elements
$tr(3, \{x, y, z\})$	a 2-3 tree contains three elements

So the function $size'$ can be implemented as returning the size information of a tree node. We have $size'(tr(s, L)) = s$.

Wrapping an element x is just calling $tr(1, \{x\})$. We can define auxiliary functions $wrap$ and $unwrap$, for instance.

$$\begin{aligned} wrap(x) &= tr(1, \{x\}) \\ unwrap(n) &= x \quad : \quad n = tr(1, \{x\}) \end{aligned} \tag{11.38}$$

As both front finger and rear finger are lists of tree nodes, in order to calculate the total size of finger, we can provide a $size''(L)$ function, which sums up size of all nodes stored in the list. Denote $L = \{a_1, a_2, ...\}$ and $L' = \{a_2, a_3, ...\}$.

$$size''(L) = \begin{cases} 0 & : \quad L = \phi \\ size'(a_1) + size''(L') & : \quad otherwise \end{cases} \tag{11.39}$$

It's quite OK to define $size''(L)$ by using some high order functions. For example.

$$size''(L) = sum(map(size', L)) \tag{11.40}$$

And we can turn a list of tree nodes into one deeper 2-3 tree and vice-versa.

$$\begin{aligned} wraps(L) &= tr(size''(L), L) \\ unwraps(n) = L \quad &: \quad n = tr(s, L) \end{aligned} \tag{11.41}$$

These helper functions are translated to the following Haskell code.

```
size (Br s _) = s

sizeL = sum ∘ (map size)

sizeT Empty = 0
sizeT (Lf a) = size a
sizeT (Tr s _ _ _) = s
```

Here are the wrap and unwrap auxiliary functions.

```
wrap x = Br 1 [x]
unwrap (Br 1 [x]) = x
wraps xs = Br (sizeL xs) xs
unwraps (Br _ xs) = xs
```

We omitted their type definitions for illustration purpose.

In imperative settings, the size information for node and tree can be accessed through the size field. And the size of a list of nodes can be summed up for this field as the below algorithm.

function SIZE-NODES(L)
  $s \leftarrow 0$
  for $\forall n \in L$ **do**
    $s \leftarrow s +$ SIZE(n)
  return s

The following Python code, for example, translates this algorithm by using standard `sum()` and `map()` functions provided in library.

```
def sizeNs(xs):
    return sum(map(lambda x: x.size, xs))
```

As NIL is typically used to represent empty tree in imperative settings, it's convenient to provide a auxiliary size function to uniformed calculate the size of tree no matter it is NIL.

function SIZE-TR(T)
  if $T =$ NIL **then**
    return 0
  else
    return SIZE(T)

The algorithm is trivial and we skip its implementation example program.

Modification due to the augmented size

The algorithms have been presented so far need to be modified to accomplish with the augmented size. For example the $insertT$ function now inserts a tree node instead of a plain element.

$$insertT(x, T) = insertT'(wrap(x), T) \tag{11.42}$$

www.dbooks.org

The corresponding Haskell program is changed as below.

```
cons a t = cons' (wrap a) t
```

After being wrapped, x is augmented with size information of 1. In the implementation of previous insertion algorithm, function $tree(F, M, R)$ is used to create a finger tree from a front finger, a middle part inner tree and a rear finger. This function should also be modified to add size information of these three arguments.

$$tree'(F, M, R) = \begin{cases} fromL(F) & : & M = \phi \wedge R = \phi \\ fromL(R) & : & M = \phi \wedge F = \phi \\ tree'(unwraps(F'), M', R) & : & F = \phi, (F', M') = extractT'(M \\ tree'(F, M', unwraps(R')) & : & R = \phi, (M', R') = removeT'(M \\ tree(size''(F) + size(M) + size''(R), F, M, R) & : & otherwise \end{cases} \tag{11.43}$$

Where $fromL()$ helps to turn a list of nodes to a finger tree by repeatedly inserting all the element one by one to an empty tree.

$$fromL(L) = foldR(insertT', \phi, L)$$

Of course it can be implemented in pure recursive manner without using folding as well.

The last case is the most straightforward one. If none of F, M, and R is empty, it adds the size of these three part and construct the tree along with this size information by calling $tree(s, F, M, R)$ function. If both the middle part inner tree and one of the finger is empty, the algorithm repeatedly insert all elements stored in the other finger to an empty tree, so that the result is constructed from a list of tree nodes. If the middle part inner tree isn't empty, and one of the finger is empty, the algorithm 'borrows' one tree node from the middle part, either by extracting from head if front finger is empty or removing from tail if rear finger is empty. Then the algorithm unwraps the 'borrowed' tree node to a list, and recursively call $tree'()$ function to construct the result.

This algorithm can be translated to the following Haskell code for example.

```
tree f Empty [] = foldr cons' Empty f
tree [] Empty r = foldr cons' Empty r
tree [] m r = let (f, m') = uncons' m in tree (unwraps f) m' r
tree f m [] = let (m', r) = unsnoc' m in tree f m' (unwraps r)
tree f m r = Tr (sizeL f + sizeT m + sizeL r) f m r
```

Function $tree'()$ helps to minimize the modification. $insertT'()$ can be realized by using it like the following.

$$insertT'(x, T) = \begin{cases} leaf(x) & : & T = \phi \\ tree'(\{x\}, \phi, \{y\}) & : & T = leaf(x) \\ tree'(\{x, x_1\}, insertT'(wraps(\{x_2, x_3, x_4\}), M), R) & : & T = tree(s, \{x_1, x_2, x_3, x_4\} \\ tree'(\{x\} \cup F, M, R) & : & otherwise \end{cases} \tag{11.44}$$

And it's corresponding Haskell code is a line by line translation.

```
cons' a Empty = Lf a
cons' a (Lf b) = tree [a] Empty [b]
cons' a (Tr _ [b, c, d, e] m r) = tree [a, b] (cons' (wraps [c, d, e]) m) r
cons' a (Tr _ f m r) = tree (a:f) m r
```

The similar modification for augment size should also be tuned for imperative algorithms, for example, when a new node is prepend to the head of the finger tree, we should update size when traverse the tree.

function PREPEND-NODE(n, T)
  $r \leftarrow$ TREE()
  $p \leftarrow r$
  CONNECT-MID(p, T)
  while FULL?(FRONT(T)) **do**
    $F \leftarrow$ FRONT(T)
    FRONT(T) $\leftarrow \{n, F[1]\}$
    SIZE(T) $\leftarrow$ SIZE(T) + SIZE(n) ▷ update size
    $n \leftarrow$ NODE()
    CHILDREN(n) $\leftarrow F[2..]$
    $p \leftarrow T$
    $T \leftarrow$ MID(T)
  if $T =$ NIL **then**
    $T \leftarrow$ TREE()
    FRONT(T)$\leftarrow \{n\}$
  else if $\mid$ FRONT(T) $\mid = 1 \wedge$ REAR(T) $= \phi$ **then**
    REAR(T) $\leftarrow$ FRONT(T)
    FRONT(T) $\leftarrow \{n\}$
  else
    FRONT(T) $\leftarrow \{n\} \cup$ FRONT(T)
  SIZE(T) $\leftarrow$ SIZE(T) + SIZE(n) ▷ update size
  CONNECT-MID(p, T) $\leftarrow T$
  return FLAT(r)

The corresponding Python code are modified accordingly as below.

```
def prepend_node(n, t):
    root = prev = Tree()
    prev.set_mid(t)
    while frontFull(t):
        f = t.front
        t.front = [n] + f[:1]
        t.size = t.size + n.size
        n = wraps(f[1:])
        prev = t
        t = t.mid
    if t is None:
        t = leaf(n)
    elif len(t.front)==1 and t.rear == []:
        t = Tree(n.size + t.size, [n], None, t.front)
    else:
        t = Tree(n.size + t.size, [n]+t.front, t.mid, t.rear)
    prev.set_mid(t)
    return flat(root)
```

www.dbooks.org

Note that the tree constructor is also modified to take a size argument as the first parameter. And the `leaf` helper function does not only construct the tree from a node, but also set the size of the tree with the same size of the node inside it.

For simplification purpose, we skip the detailed description of what are modified in *extractT*, *appendT*, *removeT*, and *concat* algorithms. They are left as exercises to the reader.

Split a finger tree at a given position

With size information augmented, it's easy to locate a node at given position by performing a tree search. What's more, as the finger tree is constructed from three part F, M, and R; and it's nature of recursive, it's also possible to split it into three sub parts with a given position i: the left, the node at i, and the right part.

The idea is straight forward. Since we have the size information for F, M, and R. Denote these three sizes as S_f, S_m, and S_r. if the given position $i \leq S_f$, the node must be stored in F, we can go on seeking the node inside F; if $S_f < i \leq S_f + S_m$, the node must be stored in M, we need recursively perform search in M; otherwise, the node should be in R, we need search inside R.

If we skip the error handling of trying to split an empty tree, there is only one edge case as below.

$$splitAt(i, T) = \begin{cases} (\phi, x, \phi) & : \quad T = leaf(x) \\ ... & : \quad otherwise \end{cases}$$

Splitting a leaf results both the left and right parts empty, the node stored in leaf is the resulting node.

The recursive case handles the three sub cases by comparing i with the sizes. Suppose function $splitAtL(i, L)$ splits a list of nodes at given position i into three parts: $(A, x, B) = splitAtL(i, L)$, where x is the i-th node in L, A is a sub list contains all nodes before position i, and B is a sub list contains all rest nodes after i.

$$splitAt(i, T) = \begin{cases} (\phi, x, \phi) & : \quad T = leaf(x) \\ (fromL(A), x, tree'(B, M, R)) & : \quad i \leq S_f, (A, x, B) = splitAtL(i, F) \\ (tree'(F, M_l, A), x, tree'(B, M_r, R)) & : \quad S_f < i \leq S_f + S_m \\ (tree'(F, M, A), x, fromL(B)) & : \quad otherwise, (A, x, B) = splitAtL(i - S_f - S_m \end{cases} \tag{11.45}$$

Where M_l, x, M_r, A, B in the third case are calculated as the following.

$$\begin{aligned} &(M_l, t, M_r) = splitAt(i - S_f, M) \\ &(A, x, B) = splitAtL(i - S_f - size(M_l), unwraps(t)) \end{aligned}$$

And the function $splitAtL$ is just a linear traverse, since the length of list is limited not to exceed the constraint of 2-3 tree, the performance is still ensured to be constant $O(1)$ time. Denote $L = \{x_1, x_2, ...\}$ and $L' = \{x_2, x_3, ...\}$.

$$splitAtL(i, L) = \begin{cases} (\phi, x_1, \phi) & : \quad i = 0 \wedge L = \{x_1\} \\ (\phi, x_1, L') & : \quad i < size'(x_1) \\ (\{x_1\} \cup A, x, B) & : \quad otherwise \end{cases} \tag{11.46}$$

Where

$$(A, x, B) = splitAtL(i - size'(x_1), L')$$

The solution of splitting is a typical divide and conquer strategy. The performance of this algorithm is determined by the recursive case of searching in middle part inner tree. Other cases are all constant time as we've analyzed. The depth of recursion is proportion to the height of the tree h, so the algorithm is bound to $O(h)$. Because the tree is well balanced (by using 2-3 tree, and all the insertion/removal algorithms keep the tree balanced), so $h = O(\lg n)$ where n is the number of elements stored in finger tree. The overall performance of splitting is $O(\lg n)$.

Let's first give the Haskell program for *splitAtL* function

```
splitNodesAt 0 [x] = ([], x, [])
splitNodesAt i (x:xs) | i < size x = ([], x, xs)
                      | otherwise = let (xs', y, ys) = splitNodesAt (i-size x) xs
                                    in (x:xs', y, ys)
```

Then the program for *splitAt*, as there is already function defined in standard library with this name, we slightly change the name by adding a apostrophe.

```
splitAt' _ (Lf x) = (Empty, x, Empty)
splitAt' i (Tr _ f m r)
    | i < szf = let (xs, y, ys) = splitNodesAt i f
                in ((foldr cons' Empty xs), y, tree ys m r)
    | i < szf + szm = let (m1, t, m2) = splitAt' (i-szf) m
                          (xs, y, ys) = splitNodesAt (i-szf - sizeT m1) (unwraps t)
                      in (tree f m1 xs, y, tree ys m2 r)
    | otherwise = let (xs, y, ys) = splitNodesAt (i-szf -szm) r
                  in (tree f m xs, y, foldr cons' Empty ys)
    where
      szf = sizeL f
      szm = sizeT m
```

Random access

With the help of splitting at any arbitrary position, it's trivial to realize random access in $O(\lg n)$ time. Denote function $mid(x)$ returns the 2-nd element of a tuple, $left(x)$, and $right(x)$ return the first element and the 3-rd element of the tuple respectively.

$$getAt(S, i) = unwrap(mid(splitAt(i, S))) \tag{11.47}$$

It first splits the sequence at position i, then unwraps the node to get the element stored inside it. When mutate the i-th element of sequence S represented by finger tree, we first split it at i, then we replace the middle to what we want to mutate, and re-construct them to one finger tree by using concatenation.

$$setAt(S, i, x) = concat(L, insertT(x, R)) \tag{11.48}$$

where

$$(L, y, R) = splitAt(i, S)$$

www.dbooks.org

What's more, we can also realize a $removeAt(S, i)$ function, which can remove the i-th element from sequence S. The idea is first to split at i, unwrap and return the element of the i-th node; then concatenate the left and right to a new finger tree.

$$removeAt(S, i) = (unwrap(y), concat(L, R)) \tag{11.49}$$

These handy algorithms can be translated to the following Haskell program.

```
getAt t i = unwrap x where (_, x, _) = splitAt' i t
setAt t i x = let (l, _, r) = splitAt' i t in concat' l (cons x r)
removeAt t i = let (l, x, r) = splitAt' i t in (unwrap x, concat' l r)
```

Imperative random access

As we can directly mutate the tree in imperative settings, it's possible to realize GET-AT(T, i) and SET-AT(T, i, x) without using splitting. The idea is firstly implement a algorithm which can apply some operation to a given position. The following algorithm takes three arguments, a finger tree T, a position index at i which ranges from zero to the number of elements stored in the tree, and a function f, which will be applied to the element at i.

function APPLY-AT(T, i, f)
  while SIZE$(T) > 1$ **do**
    $S_f \leftarrow$ SIZE-NODES(FRONT(T))
    $S_m \leftarrow$ SIZE-TR(MID(T))
    if $i < S_f$ **then**
      return LOOKUP-NODES(FRONT(T), i, f)
    else if $i < S_f + S_m$ **then**
      $T \leftarrow$ MID(T)
      $i \leftarrow i - S_f$
    else
      return LOOKUP-NODES(REAR(T), $i - S_f - S_m$, f)
  $n \leftarrow$ FIRST-LF(T)
  $x \leftarrow$ ELEM(n)
  ELEM$(n) \leftarrow f(x)$
  return x

This algorithm is essentially a divide and conquer tree search. It repeatedly examine the current tree till reach a tree with size of 1 (can it be determined as a leaf? please consider the ill-formed case and refer to the exercise later). Every time, it checks the position to be located with the size information of front finger and middle part inner tree.

If the index i is less than the size of front finger, the location is at some node in it. The algorithm call a sub procedure to look-up among front finger; If the index is between the size of front finger and the total size till middle part inner tree, it means that the location is at some node inside the middle, the algorithm goes on traverse along the middle part inner tree with an updated index reduced by the size of front finger; Otherwise it means the location is at some node in rear finger, the similar looking up procedure is called accordingly.

After this loop, we've got a node, (can be a compound node) with what we are looking for at the first leaf inside this node. We can extract the element out, and apply the function f on it and store the new value back.

The algorithm returns the previous element before applying f as the final result.

What hasn't been factored is the algorithm LOOKUP-NODES(L, i, f). It takes a list of nodes, a position index, and a function to be applied. This algorithm can be implemented by checking every node in the list. If the node is a leaf, and the index is zero, we are at the right position to be looked up. The function can be applied on the element stored in this leaf, and the previous value is returned; Otherwise, we need compare the size of this node and the index to determine if the position is inside this node and search inside the children of the node if necessary.

function LOOKUP-NODES(L, i, f)
  loop
    for $\forall n \in L$ **do**
      if n is leaf $\wedge i = 0$ **then**
        $x \leftarrow$ ELEM(n)
        ELEM(n) $\leftarrow f(x)$
        return x
      if $i <$ SIZE(n) **then**
        $L \leftarrow$ CHILDREN(n)
        break
      $i \leftarrow i -$ SIZE(n)

The following are the corresponding Python code implements the algorithms.

```
def applyAt(t, i, f):
    while t.size > 1:
        szf = sizeNs(t.front)
        szm = sizeT(t.mid)
        if i < szf:
            return lookupNs(t.front, i, f)
        elif i < szf + szm:
            t = t.mid
            i = i - szf
        else:
            return lookupNs(t.rear, i - szf - szm, f)
    n = first_leaf(t)
    x = elem(n)
    n.children[0] = f(x)
    return x

def lookupNs(ns, i, f):
    while True:
        for n in ns:
            if n.leaf and i == 0:
                x = elem(n)
                n.children[0] = f(x)
                return x
            if i < n.size:
                ns = n.children
                break
            i = i - n.size
```

With auxiliary algorithm that can apply function at a given position, it's trivial to implement the GET-AT and SET-AT by passing special functions for

www.dbooks.org

applying.

```
function GET-AT(T, i)
    return APPLY-AT(T, i, λx.x)

function SET-AT(T, i, x)
    return APPLY-AT(T, i, λy.x)
```

That is we pass *id* function to implement getting element at a position, which doesn't change anything at all; and pass constant function to implement setting, which set the element to new value by ignoring its previous value.

Imperative splitting

It's not enough to just realizing APPLY-AT algorithm in imperative settings, this is because removing element at arbitrary position is also a typical case.

Almost all the imperative finger tree algorithms so far are kind of one-pass top-down manner. Although we sometimes need to book keeping the root. It means that we can even realize all of them without using the parent field.

Splitting operation, however, can be easily implemented by using parent field. We can first perform a top-down traverse along with the middle part inner tree as long as the splitting position doesn't located in front or rear finger. After that, we need a bottom-up traverse along with the parent field of the two split trees to fill out the necessary fields.

```
function SPLIT-AT(T, i)
    T_1 ← TREE()
    T_2 ← TREE()
    while S_f ≤ i < S_f + S_m do                          ▷ Top-down pass
        T'_1 ← TREE()
        T'_2 ← TREE()
        FRONT(T'_1) ← FRONT(T)
        REAR(T'_2) ← REAR(T)
        CONNECT-MID(T_1, T'_1)
        CONNECT-MID(T_2, T'_2)
        T_1 ← T'_1
        T_2 ← T'_2
        i ← i − S_f
        T ← MID(T)
    if i < S_f then
        (X, n, Y) ← SPLIT-NODES(FRONT(T), i)
        T'_1 ← FROM-NODES(X)
        T'_2 ← T
        SIZE(T'_2) ← SIZE(T) - SIZE-NODES(X) - SIZE(n)
        FRONT(T'_2) ← Y
    else if S_f + S_m ≤ i then
        (X, n, Y) ← SPLIT-NODES(REAR(T), i − S_f − S_m)
        T'_2 ← FROM-NODES(Y)
        T'_1 ← T
        SIZE(T'_1) ← SIZE(T) - SIZE-NODES(Y) - SIZE(n)
        REAR(T'_1) ← X
    CONNECT-MID(T_1, T'_1)
```

```
    CONNECT-MID(T_2, T'_2)
    i ← i− SIZE-TR(T'_1)
    while n is NOT leaf do                                  ▷ Bottom-up pass
        (X, n, Y) ← SPLIT-NODES(CHILDREN(n), i)
        i ← i− SIZE-NODES(X)
        REAR(T_1) ← X
        FRONT(T_2) ← Y
        SIZE(T_1) ← SUM-SIZES(T_1)
        SIZE(T_2) ← SUM-SIZES(T_2)
        T_1 ← PARENT(T_1)
        T_2 ← PARENT(T_2)
    return (FLAT(T_1), ELEM(n), FLAT(T_2))
```

The algorithm first creates two trees T_1 and T_2 to hold the split results. Note that they are created as 'ground' trees which are parents of the roots. The first pass is a top-down pass. Suppose S_f, and S_m retrieve the size of the front finger and the size of middle part inner tree respectively. If the position at which the tree to be split is located at middle part inner tree, we reuse the front finger of T for new created T'_1, and reuse rear finger of T for T'_2. At this time point, we can't fill the other fields for T'_1 and T'_2, they are left empty, and we'll finish filling them in the future. After that, we connect T_1 and T'_1 so the latter becomes the middle part inner tree of the former. The similar connection is done for T_2 and T'_2 as well. Finally, we update the position by deducing it by the size of front finger, and go on traversing along with the middle part inner tree.

When the first pass finishes, we are at a position that either the splitting should be performed in front finger, or in rear finger. Splitting the nodes in finger results a tuple, that the first part and the third part are lists before and after the splitting point, while the second part is a node contains the element at the original position to be split. As both fingers hold at most 3 nodes because they are actually 2-3 trees, the nodes splitting algorithm can be performed by a linear search.

```
function SPLIT-NODES(L, i)
    for j ∈ [1, LENGTH(L) ] do
        if i < SIZE(L[j]) then
            return (L[1...j − 1], L[j], L[j + 1... LENGTH(L) ])
        i ← i− SIZE(L[j])
```

We next create two new result trees T'_1 and T'_2 from this tuple, and connected them as the final middle part inner tree of T_1 and T_2.

Next we need perform a bottom-up traverse along with the result trees to fill out all the empty information we skipped in the first pass.

We loop on the second part of the tuple, the node, till it becomes a leaf. In each iteration, we repeatedly splitting the children of the node with an updated position i. The first list of nodes returned from splitting is used to fill the rear finger of T_1; and the other list of nodes is used to fill the front finger of T_2. After that, since all the three parts of a finger tree – the front and rear finger, and the middle part inner tree – are filled, we can then calculate the size of the tree by summing these three parts up.

```
function SUM-SIZES(T)
    return SIZE-NODES(FRONT(T)) + SIZE-TR(MID(T)) + SIZE-NODES(REAR(T))
```

www.dbooks.org

Next, the iteration goes on along with the parent fields of T_1 and T_2. The last 'black-box' algorithm is FROM-NODES(L), which can create a finger tree from a list of nodes. It can be easily realized by repeatedly perform insertion on an empty tree. The implementation is left as an exercise to the reader.

The example Python code for splitting is given as below.

```
def splitAt(t, i):
    (t1, t2) = (Tree(), Tree())
    while szf(t) ≤ i and i < szf(t) + szm(t):
        fst = Tree(0, t.front, None, [])
        snd = Tree(0, [], None, t.rear)
        t1.set_mid(fst)
        t2.set_mid(snd)
        (t1, t2) = (fst, snd)
        i = i - szf(t)
        t = t.mid

    if i < szf(t):
        (xs, n, ys) = splitNs(t.front, i)
        sz = t.size - sizeNs(xs) - n.size
        (fst, snd) = (fromNodes(xs), Tree(sz, ys, t.mid, t.rear))
    elif szf(t) + szm(t) ≤ i:
        (xs, n, ys) = splitNs(t.rear, i - szf(t) - szm(t))
        sz = t.size - sizeNs(ys) - n.size
        (fst, snd) = (Tree(sz, t.front, t.mid, xs), fromNodes(ys))
    t1.set_mid(fst)
    t2.set_mid(snd)

    i = i - sizeT(fst)
    while not n.leaf:
        (xs, n, ys) = splitNs(n.children, i)
        i = i - sizeNs(xs)
        (t1.rear, t2.front) = (xs, ys)
        t1.size = sizeNs(t1.front) + sizeT(t1.mid) + sizeNs(t1.rear)
        t2.size = sizeNs(t2.front) + sizeT(t2.mid) + sizeNs(t2.rear)
        (t1, t2) = (t1.parent, t2.parent)

    return (flat(t1), elem(n), flat(t2))
```

The program to split a list of nodes at a given position is listed like this.

```
def splitNs(ns, i):
    for j in range(len(ns)):
        if i < ns[j].size:
            return (ns[:j], ns[j], ns[j+1:])
        i = i - ns[j].size
```

With splitting defined, removing an element at arbitrary position can be realized trivially by first performing a splitting, then concatenating the two result tree to one big tree and return the element at that position.

function REMOVE-AT(T, i)
  (T_1, x, T_2) ← SPLIT-AT(T, i)
  return (x, CONCAT(T_1, T_2))

Exercise 11.6

1. Another way to realize $insertT'$ is to force increasing the size field by one, so that we needn't write function $tree'$. Try to realize the algorithm by using this idea.

2. Try to handle the augment size information as well as in $insertT'$ algorithm for the following algorithms (both functional and imperative): $extractT'$, $appendT'$, $removeT'$, and $concat'$. The *head*, *tail*, *init* and *last* functions should be kept unchanged. Don't refer to the downloadable programs along with this book before you take a try.

3. In the imperative Apply-At algorithm, it tests if the size of the current tree is greater than one. Why don't we test if the current tree is a leaf? Tell the difference between these two approaches.

4. Implement the From-Nodes(L) in your favorite imperative programming language. You can either use looping or create a folding-from-right sub algorithm.

11.7 Notes and short summary

Although we haven't been able to give a purely functional realization to match the $O(1)$ constant time random access as arrays in imperative settings. The result finger tree data structure achieves an overall well performed sequence. It manipulates fast in amortized $O(1)$ time both on head an on tail, it can also concatenates two sequence in logarithmic time as well as break one sequence into two sub sequences at any position. While neither arrays in imperative settings nor linked-list in functional settings satisfies all these goals. Some functional programming languages adopt this sequence realization in its standard library [7].

Just as the title of this chapter, we've presented the last corner stone of elementary data structures in both functional and imperative settings. We needn't concern about being lack of elementary data structures when solve problems with some typical algorithms.

For example, when writing a MTF (move-to-front) encoding algorithm[8], with the help of the sequence data structure explained in this chapter. We can implement it quite straightforward.

$$mtf(S, i) = \{x\} \cup S'$$

where $(x, S') = removeAt(S, i)$.

In the next following chapters, we'll first explains some typical divide and conquer sorting methods, including quick sort, merge sort and their variants; then some elementary searching algorithms, and string matching algorithms will be covered.

www.dbooks.org

Bibliography

[1] Chris Okasaki. “Purely Functional Data Structures”. Cambridge university press, (July 1, 1999), ISBN-13: 978-0521663502

[2] Chris Okasaki. “Purely Functional Random-Access Lists”. Functional Programming Languages and Computer Architecture, June 1995, pages 86-95.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. The MIT Press, 2001. ISBN: 0262032937.

[4] Miran Lipovaca. “Learn You a Haskell for Great Good! A Beginner’s Guide”. No Starch Press; 1 edition April 2011, 400 pp. ISBN: 978-1-59327-283-8

[5] Ralf Hinze and Ross Paterson. “Finger Trees: A Simple General-purpose Data Structure.” in Journal of Functional Programming16:2 (2006), pages 197-217. http://www.soi.city.ac.uk/ ross/papers/FingerTree.html

[6] Guibas, L. J., McCreight, E. M., Plass, M. F., Roberts, J. R. (1977), ”A new representation for linear lists”. Conference Record of the Ninth Annual ACM Symposium on Theory of Computing, pp. 49¨C60.

[7] Generic finger-tree structure. http://hackage.haskell.org/packages/archive/fingertree/0.0/doc/html/Data-FingerTree.html

[8] Wikipedia. Move-to-front transform. http://en.wikipedia.org/wiki/Move-to-front_transform

www.dbooks.org

Part V

Sorting and Searching

www.dbooks.org

Chapter 12

Divide and conquer, Quick sort vs. Merge sort

12.1 Introduction

It's proved that the best approximate performance of comparison based sorting is $O(n \lg n)$ [1]. In this chapter, two divide and conquer sorting algorithms are introduced. Both of them perform in $O(n \lg n)$ time. One is quick sort. It is the most popular sorting algorithm. Quick sort has been well studied, many programming libraries provide sorting tools based on quick sort.

In this chapter, we'll first introduce the idea of quick sort, which demonstrates the power of divide and conquer strategy well. Several variants will be explained, and we'll see when quick sort performs poor in some special cases. That the algorithm is not able to partition the sequence in balance.

In order to solve the unbalanced partition problem, we'll next introduce about merge sort, which ensure the sequence to be well partitioned in all the cases. Some variants of merge sort, including nature merge sort, bottom-up merge sort are shown as well.

Same as other chapters, all the algorithm will be realized in both imperative and functional approaches.

12.2 Quick sort

Consider a teacher arranges a group of kids in kindergarten to stand in a line for some game. The kids need stand in order of their heights, that the shortest one stands on the left most, while the tallest stands on the right most. How can the teacher instruct these kids, so that they can stand in a line by themselves?

There are many strategies, and the quick sort approach can be applied here:

1. The first kid raises his/her hand. The kids who are shorter than him/her stands to the left to this child; the kids who are taller than him/her stands to the right of this child;

2. All the kids move to the left, if there are, repeat the above step; all the kids move to the right repeat the same step as well.

www.dbooks.org



Figure 12.1: Instruct kids to stand in a line

Suppose a group of kids with their heights as $\{102, 100, 98, 95, 96, 99, 101, 97\}$ with [cm] as the unit. The following table illustrate how they stand in order of height by following this method.

102	100	98	95	96	99	101	97
100	98	95	96	99	101	97	*102*
98	95	96	99	97	*100*	101	*102*
95	96	97	*98*	99	*100*	*101*	*102*
95	**96**	97	*98*	*99*	*100*	*101*	*102*
95	*96*	97	*98*	*99*	*100*	*101*	*102*
95	*96*	*97*	*98*	*99*	*100*	*101*	*102*

At the beginning, the first child with height 102 cm raises his/her hand. We call this kid the pivot and mark this height in bold. It happens that this is the tallest kid. So all others stands to the left side, which is represented in the second row in the above table. Note that the child with height 102 cm is in the final ordered position, thus we mark it italic. Next the kid with height 100 cm raise hand, so the children of heights 98, 95, 96 and 99 cm stand to his/her left, and there is only 1 child of height 101 cm who is taller than this pivot kid. So he stands to the right hand. The 3rd row in the table shows this stage accordingly. After that, the child of 98 cm high is selected as pivot on left hand; while the child of 101 cm high on the right is selected as pivot. Since there are no other kids in the unsorted group with 101 cm as pivot, this small group is ordered already and the kid of height 101 cm is in the final proper position. The same method is applied to the group of kids which haven't been in correct order until all of them are stands in the final position.

12.2.1 Basic version

Summarize the above instruction leads to the recursive description of quick sort. In order to sort a sequence of elements L.

- If L is empty, the result is obviously empty; This is the trivial edge case;
- Otherwise, select an arbitrary element in L as a pivot, recursively sort all elements not greater than the pivot, put the result on the left hand of the pivot, *and* recursively sort all elements which are greater than the pivot, put the result on the right hand of the pivot.

Note that the emphasized word *and*, we don't use 'then' here, which indicates it's quite OK that the recursive sort on the left and right can be done in parallel. We'll return this parallelism topic soon.

Quick sort was first developed by C. A. R. Hoare in 1960 [1], [15]. What we describe here is a basic version. Note that it doesn't state how to select the pivot. We'll see soon that the pivot selection affects the performance of quick sort dramatically.

The most simple method to select the pivot is always choose the first one so that quick sort can be formalized as the following.

$$sort(L) = \begin{cases} \phi & : \quad L = \phi \\ sort(\{x|x \in L', x \leq l_1\}) \cup \{l_1\} \cup sort(\{x|x \in L', l_1 < x\}) & : \quad otherwise \end{cases} \tag{12.1}$$

Where l_1 is the first element of the non-empty list L, and L' contains the rest elements $\{l_2, l_3, ...\}$. Note that we use Zermelo Frankel expression (ZF expression for short)[1], which is also known as list comprehension. A ZF expression $\{a|a \in S, p_1(a), p_2(a), ...\}$ means taking all element in set S, if it satisfies all the predication $p_1, p_2, ...$ ZF expression is originally used for representing *set*, we extend it to express list for the sake of brevity. There can be duplicated elements, and different permutations represent for different list. Please refer to the appendix about list in this book for detail.

It's quite straightforward to translate this equation to real code if list comprehension is supported. The following Haskell code is given for example:

```
sort [] = []
sort (x:xs) = sort [y | y←xs, y ≤ x] ++ [x] ++ sort [y | y←xs, x < y]
```

This might be the shortest quick sort program in the world at the time when this book is written. Even a verbose version is still very expressive:

```
sort [] = []
sort (x:xs) = as ++ [x] ++ bs where
    as = sort [ a | a ← xs, a ≤ x]
    bs = sort [ b | b ← xs, x < b]
```

There are some variants of this basic quick sort program, such as using explicit filtering instead of list comprehension. The following Python program demonstrates this for example:

```
def sort(xs):
    if xs == []:
        return []
    pivot = xs[0]
    as = sort(filter(lambda x : x ≤ pivot, xs[1:]))
    bs = sort(filter(lambda x : pivot < x, xs[1:]))
    return as + [pivot] + bs
```

12.2.2 Strict weak ordering

We assume the elements are sorted in monotonic none decreasing order so far. It's quite possible to customize the algorithm, so that it can sort the elements

[1] Name for the two mathematics who found the modern set theory.

www.dbooks.org

in other ordering criteria. This is necessary in practice because users may sort numbers, strings, or other complex objects (even list of lists for example).

The typical generic solution is to abstract the comparison as a parameter as we mentioned in chapters about insertion sort and selection sort. Although it needn't the total ordering, the comparison must satisfy *strict weak ordering* at least [17] [16].

For the sake of brevity, we only considering sort the elements by using less than or equal (equivalent to not greater than) in the rest of the chapter.

12.2.3 Partition

Observing that the basic version actually takes two passes to find all elements which are greater than the pivot as well as to find the others which are not respectively. Such partition can be accomplished by only one pass. We explicitly define the partition as below.

$$partition(p, L) = \begin{cases} (\phi, \phi) & : & L = \phi \\ (\{l_1\} \cup A, B) & : & p(l_1), (A, B) = partition(p, L') \\ (A, \{l_1\} \cup B) & : & \neg p(l_1) \end{cases} \tag{12.2}$$

Note that the operation $\{x\} \cup L$ is just a 'cons' operation, which only takes constant time. The quick sort can be modified accordingly.

$$sort(L) = \begin{cases} \phi & : & L = \phi \\ sort(A) \cup \{l_1\} \cup sort(B) & : & otherwise, (A, B) = partition(\lambda_x x \leq l_1, L') \end{cases} \tag{12.3}$$

Translating this new algorithm into Haskell yields the below code.

```
sort [] = []
sort (x:xs) = sort as ++ [x] ++ sort bs where
    (as, bs) = partition (≤ x) xs

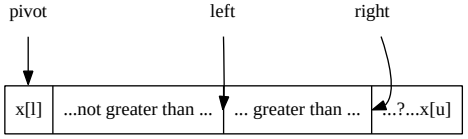
partition _ [] = ([], [])
partition p (x:xs) = let (as, bs) = partition p xs in
    if p x then (x:as, bs) else (as, x:bs)
```

The concept of partition is very critical to quick sort. Partition is also very important to many other sort algorithms. We'll explain how it generally affects the sorting methodology by the end of this chapter. Before further discussion about fine tuning of quick sort specific partition, let's see how to realize it in-place imperatively.

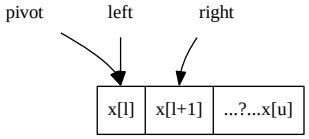
There are many partition methods. The one given by Nico Lomuto [4] [2] will be used here as it's easy to understand. We'll show other partition algorithms soon and see how partitioning affects the performance.

Figure 12.2 shows the idea of this one-pass partition method. The array is processed from left to right. At any time, the array consists of the following parts as shown in figure 12.2 (a):

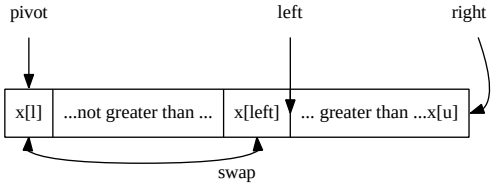
- The left most cell contains the pivot; By the end of the partition process, the pivot will be moved to the final proper position;



(a) Partition invariant



(b) Start



(c) Finish

Figure 12.2: Partition a range of array by using the left most element as pivot.

- A segment contains all elements which are not greater than the pivot. The right boundary of this segment is marked as 'left';
- A segment contains all elements which are greater than the pivot. The right boundary of this segment is marked as 'right'; It means that elements between 'left' and 'right' marks are greater than the pivot;
- The rest of elements after 'right' mark haven't been processed yet. They may be greater than the pivot or not.

At the beginning of partition, the 'left' mark points to the pivot and the 'right' mark points to the the second element next to the pivot in the array as in Figure 12.2 (b); Then the algorithm repeatedly advances the right mark one element after the other till passes the end of the array.

In every iteration, the element pointed by the 'right' mark is compared with the pivot. If it is greater than the pivot, it should be among the segment between the 'left' and 'right' marks, so that the algorithm goes on to advance the 'right' mark and examine the next element; Otherwise, since the element pointed by 'right' mark is less than or equal to the pivot (not greater than), it should be put before the 'left' mark. In order to achieve this, the 'left' mark needs be advanced by one, then exchange the elements pointed by the 'left' and 'right' marks.

Once the 'right' mark passes the last element, it means that all the elements have been processed. The elements which are greater than the pivot have been moved to the right hand of 'left' mark while the others are to the left hand of this mark. Note that the pivot should move between the two segments. An extra

www.dbooks.org

exchanging between the pivot and the element pointed by 'left' mark makes this final one to the correct location. This is shown by the swap bi-directional arrow in figure 12.2 (c).

The 'left' mark (which points the pivot finally) partitions the whole array into two parts, it is returned as the result. We typically increase the 'left' mark by one, so that it points to the first element greater than the pivot for convenient. Note that the array is modified in-place.

The partition algorithm can be described as the following. It takes three arguments, the array A, the lower and the upper bound to be partitioned [2].

```
1: function PARTITION(A, l, u)
2:     p ← A[l]                                          ▷ the pivot
3:     L ← l                                             ▷ the left mark
4:     for R ∈ [l + 1, u] do                             ▷ iterate on the right mark
5:         if ¬(p < A[R]) then    ▷ negate of < is enough for strict weak order
6:             L ← L + 1
7:             EXCHANGE A[L] ↔ A[R]
8:     EXCHANGE A[L] ↔ p
9:     return L + 1                                      ▷ The partition position
```

Below table shows the steps of partitioning the array $\{3, 2, 5, 4, 0, 1, 6, 7\}$.

(l) **3**	(r) 2	5	4	0	1	6	7	initialize, $pivot = 3, l = 1, r = 2$
3	(l)(r) 2	5	4	0	1	6	7	$2 < 3$, advance l, $(r = l)$
3	(l) 2	(r) 5	4	0	1	6	7	$5 > 3$, move on
3	(l) 2	5	(r) 4	0	1	6	7	$4 > 3$, move on
3	(l) 2	5	4	(r) 0	1	6	7	$0 < 3$
3	2	(l) 0	4	(r) 5	1	6	7	Advance l, then swap with r
3	2	(l) 0	4	5	(r) 1	6	7	$1 < 3$
3	2	0	(l) 1	5	(r) 4	6	7	Advance l, then swap with r
3	2	0	(l) 1	5	4	(r) 6	7	$6 > 3$, move on
3	2	0	(l) 1	5	4	6	(r) 7	$7 > 3$, move on
1	2	0	3	(l+1) 5	4	6	7	r passes the end, swap pivot and

This version of partition algorithm can be implemented in ANSI C as the following.

```
int partition(Key* xs, int l, int u) {
    int pivot, r;
    for (pivot = l, r = l + 1; r < u; ++r)
        if (!(xs[pivot] < xs[r])) {
            ++l;
            swap(xs[l], xs[r]);
        }
    swap(xs[pivot], xs[l]);
    return l + 1;
}
```

Where `swap(a, b)` can either be defined as function or a macro. In ISO C++, `swap(a, b)` is provided as a function template. the type of the elements can be defined somewhere or abstracted as a template parameter in ISO C++. We omit these language specific details here.

[2]The partition algorithm used here is slightly different from the one in [2]. The latter uses the last element in the slice as the pivot.

With this partition method realized, the imperative in-place quick sort can be accomplished as the following.

```
1: procedure QUICK-SORT(A, l, u)
2:     if l < u then
3:         m ← PARTITION(A, l, u)
4:         QUICK-SORT(A, l, m − 1)
5:         QUICK-SORT(A, m, u)
```

When sort an array, this procedure is called by passing the whole range as the lower and upper bounds. QUICK-SORT$(A, 1, |A|)$. Note that when $l \geq u$ it means the array slice is either empty, or just contains only one element, both can be treated as ordered, so the algorithm does nothing in such cases.

Below ANSI C example program completes the basic in-place quick sort.

```
void quicksort(Key* xs, int l, int u) {
    int m;
    if (l < u) {
        m = partition(xs, l, u);
        quicksort(xs, l, m - 1);
        quicksort(xs, m, u);
    }
}
```

12.2.4 Minor improvement in functional partition

Before exploring how to improve the partition for basic version quick sort, it's obviously that the one presented so far can be defined by using folding. Please refer to the appendix A of this book for definition of folding.

$$partition(p, L) = fold(f(p), (\phi, \phi), L) \tag{12.4}$$

Where function f compares the element to the pivot with predicate p (which is passed to f as a parameter, so that f is in curried form, see appendix A for detail. Alternatively, f can be a lexical closure which is in the scope of *partition*, so that it can access the predicate in this scope.), and update the result pair accordingly.

$$f(p, x, (A, B)) = \begin{cases} (\{x\} \cup A, B) & : \quad p(x) \\ (A, \{x\} \cup B) & : \quad otherwise(\neg p(x)) \end{cases} \tag{12.5}$$

Note we actually use pattern-matching style definition. In environment without pattern-matching support, the pair (A, B) should be represented by a variable, for example P, and use access functions to extract its first and second parts.

The example Haskell program needs to be modified accordingly.

```
sort [] = []
sort (x:xs) = sort small ++ [x] ++ sort big where
  (small, big) = foldr f ([], []) xs
  f a (as, bs) = if a ≤ x then (a:as, bs) else (as, a:bs)
```

www.dbooks.org

Accumulated partition

The partition algorithm by using folding actually accumulates to the result pair of lists (A, B). That if the element is not greater than the pivot, it's accumulated to A, otherwise to B. We can explicitly express it which save spaces and is friendly for tail-recursive call optimization (refer to the appendix A of this book for detail).

$$partition(p, L, A, B) = \begin{cases} (A, B) & : & L = \phi \\ partition(p, L', \{l_1\} \cup A, B) & : & p(l_1) \\ partition(p, L', A, \{l_1\} \cup B) & : & otherwise \end{cases} \tag{12.6}$$

Where l_1 is the first element in L if L isn't empty, and L' contains the rest elements except for l_1, that $L' = \{l_2, l_3, ...\}$ for example. The quick sort algorithm then uses this accumulated partition function by passing the $\lambda_x x \leq pivot$ as the partition predicate.

$$sort(L) = \begin{cases} \phi & : & L = \phi \\ sort(A) \cup \{l_1\} \cup sort(B) & : & otherwise \end{cases} \tag{12.7}$$

Where A, B are computed by the accumulated partition function defined above.

$$(A, B) = partition(\lambda_x x \leq l_1, L', \phi, \phi)$$

Accumulated quick sort

Observe the recursive case in the last quick sort definition. the list concatenation operations $sort(A) \cup \{l_1\} \cup sort(B)$ actually are proportion to the length of the list to be concatenated. Of course we can use some general solutions introduced in the appendix A of this book to improve it. Another way is to change the sort algorithm to accumulated manner. Something like below:

$$sort'(L, S) = \begin{cases} S & : & L = \phi \\ ... & : & otherwise \end{cases}$$

Where S is the accumulator, and we call this version by passing empty list as the accumulator to start sorting: $sort(L) = sort'(L, \phi)$. The key intuitive is that after the partition finishes, the two sub lists need to be recursively sorted. We can first recursively sort the list contains the elements which are greater than the pivot, then link the pivot in front of it and use it as an *accumulator* for next step sorting.

Based on this idea, the '...' part in above definition can be realized as the following.

$$sort'(L, S) = \begin{cases} S & : & L = \phi \\ sort(A, \{l_1\} \cup sort(B, ?)) & : & otherwise \end{cases}$$

The problem is what's the accumulator when sorting B. There is an important invariant actually, that at every time, the accumulator S holds the

elements have been sorted so far. So that we should sort B by accumulating to S.

$$sort'(L, S) = \begin{cases} S & : \quad L = \phi \\ sort(A, \{l_1\} \cup sort(B, S)) & : \quad otherwise \end{cases} \tag{12.8}$$

The following Haskell example program implements the accumulated quick sort algorithm.

```
asort xs = asort' xs []

asort' [] acc = acc
asort' (x:xs) acc = asort' as (x:asort' bs acc) where
  (as, bs) = part xs [] []
  part [] as bs = (as, bs)
  part (y:ys) as bs | y ≤ x = part ys (y:as) bs
                    | otherwise = part ys as (y:bs)
```

Exercise 12.1

- Implement the recursive basic quick sort algorithm in your favorite imperative programming language.
- Same as the imperative algorithm, one minor improvement is that besides the empty case, we needn't sort the singleton list, implement this idea in the functional algorithm as well.
- The accumulated quick sort algorithm developed in this section uses intermediate variable A, B. They can be eliminated by defining the partition function to mutually recursive call the sort function. Implement this idea in your favorite functional programming language. Please don't refer to the downloadable example program along with this book before you try it.

12.3 Performance analysis for quick sort

Quick sort performs well in practice, however, it's not easy to give theoretical analysis. It needs the tool of probability to prove the average case performance.

Nevertheless, it's intuitive to calculate the best case and worst case performance. It's obviously that the best case happens when every partition divides the sequence into two slices with equal size. Thus it takes $O(\lg n)$ recursive calls as shown in figure 12.3.

There are total $O(\lg n)$ levels of recursion. In the first level, it executes one partition, which processes n elements; In the second level, it executes partition two times, each processes $n/2$ elements, so the total time in the second level bounds to $2O(n/2) = O(n)$ as well. In the third level, it executes partition four times, each processes $n/4$ elements. The total time in the third level is also bound to $O(n)$; ... In the last level, there are n small slices each contains a single element, the time is bound to $O(n)$. Summing all the time in each level gives the total performance of quick sort in best case as $O(n \lg n)$.

www.dbooks.org

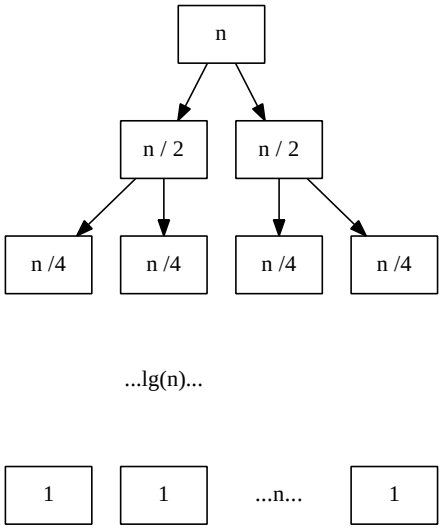


Figure 12.3: In the best case, quick sort divides the sequence into two slices with same length.

However, in the worst case, the partition process unluckily divides the sequence to two slices with unbalanced lengths in most time. That one slices with length $O(1)$, the other is $O(n)$. Thus the recursive time degrades to $O(n)$. If we draw a similar figure, unlike in the best case, which forms a balanced binary tree, the worst case degrades into a very unbalanced tree that every node has only one child, while the other is empty. The binary tree turns to be a linked list with $O(n)$ length. And in every level, all the elements are processed, so the total performance in worst case is $O(n^2)$, which is as same poor as insertion sort and selection sort.

Let's consider when the worst case will happen. One special case is that all the elements (or most of the elements) are same. Nico Lomuto's partition method deals with such sequence poor. We'll see how to solve this problem by introducing other partition algorithm in the next section.

The other two obvious cases which lead to worst case happen when the sequence has already in ascending or descending order. Partition the ascending sequence makes an empty sub list before the pivot, while the list after the pivot contains all the rest elements. Partition the descending sequence gives an opponent result.

There are other cases which lead quick sort performs poor. There is no completely satisfied solution which can avoid the worst case. We'll see some engineering practice in next section which can make it very seldom to meet the worst case.

12.3.1 Average case analysis ⋆

In average case, quick sort performs well. There is a vivid example that even the partition divides the list every time to two lists with length 1 to 9. The performance is still bound to $O(n \lg n)$ as shown in [2].

This subsection need some mathematic background, reader can safely skip to next part.

There are two methods to proof the average case performance, one uses an important fact that the performance is proportion to the total comparing operations during quick sort [2]. Different with the selections sort that every two elements have been compared. Quick sort avoid many unnecessary comparisons. For example suppose a partition operation on list $\{a_1, a_2, a_3, ..., a_n\}$. Select a_1 as the pivot, the partition builds two sub lists $A = \{x_1, x_2, ..., x_k\}$ and $B = \{y_1, y_2, ..., y_{n-k-1}\}$. In the rest time of quick sort, The element in A will never be compared with any elements in B.

Denote the final sorted result as $\{a_1, a_2, ..., a_n\}$, this indicates that if element $a_i < a_j$, they will not be compared any longer if and only if some element a_k where $a_i < a_k < a_j$ has ever been selected as pivot before a_i or a_j being selected as the pivot.

That is to say, the only chance that a_i and a_j being compared is either a_i is chosen as pivot or a_j is chosen as pivot before any other elements in ordered range $a_{i+1} < a_{i+2} < ... < a_{j-1}$ are selected.

Let $P(i, j)$ represent the probability that a_i and a_j being compared. We have:

$$P(i, j) = \frac{2}{j - i + 1} \tag{12.9}$$

Since the total number of compare operation can be given as:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(i, j) \tag{12.10}$$

Note the fact that if we compared a_i and a_j, we won't compare a_j and a_i again in the quick sort algorithm, and we never compare a_i onto itself. That's why we set the upper bound of i to $n - 1$; and lower bound of j to $i + 1$.

Substitute the probability, it yields:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} \end{aligned} \tag{12.11}$$

Using the harmonic series [18]

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... = \ln n + \gamma + \epsilon_n$$

$$C(n) = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n) \tag{12.12}$$

The other method to prove the average performance is to use the recursive fact that when sorting list of length n, the partition splits the list into two sub

www.dbooks.org

lists with length i and $n-i-1$. The partition process itself takes cn time because it examine every element with the pivot. So we have the following equation.

$$T(n) = T(i) + T(n-i-1) + cn \tag{12.13}$$

Where $T(n)$ is the total time when perform quick sort on list of length n. Since i is equally like to be any of $0, 1, ..., n-1$, taking math expectation to the equation gives:

$$\begin{aligned} T(n) &= E(T(i)) + E(T(n-i-1)) + cn \\ &= \frac{1}{n}\sum_{i=0}^{n-1} T(i) + \frac{1}{n}\sum_{i=0}^{n-1} T(n-i-1) + cn \\ &= \frac{1}{n}\sum_{i=0}^{n-1} T(i) + \frac{1}{n}\sum_{j=0}^{n-1} T(j) + cn \\ &= \frac{2}{n}\sum_{i=0}^{b-1} T(i) + cn \end{aligned} \tag{12.14}$$

Multiply by n to both sides, the equation changes to:

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + cn^2 \tag{12.15}$$

Substitute n to $n-1$ gives another equation:

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + c(n-1)^2 \tag{12.16}$$

Subtract equation (12.15) and (12.16) can eliminate all the $T(i)$ for $0 \leq i < n-1$.

$$nT(n) = (n+1)T(n-1) + 2cn - c \tag{12.17}$$

As we can drop the constant time c in computing performance. The equation can be one more step changed like below.

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} \tag{12.18}$$

Next we assign n to $n-1$, $n-2$, ..., which gives us $n-1$ equations.

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}$$

$$...$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

Sum all them up, and eliminate the same components in both sides, we can deduce to a function of n.

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{k=3}^{n+1}\frac{1}{k} \tag{12.19}$$

Using the harmonic series mentioned above, the final result is:

$$O(\frac{T(n)}{n+1}) = O(\frac{T(1)}{2} + 2c\ln n + \gamma + \epsilon_n) = O(\lg n) \tag{12.20}$$

Thus

$$O(T(n)) = O(n\lg n) \tag{12.21}$$

Exercise 12.2

- Why Lomuto's methods performs poor when there are many duplicated elements?

12.4 Engineering Improvement

Quick sort performs well in most cases as mentioned in previous section. However, there does exist the worst cases which downgrade the performance to quadratic. If the data is randomly prepared, such case is rare, however, there are some particular sequences which lead to the worst case and these kinds of sequences are very common in practice.

In this section, some engineering practices are introduces which either help to avoid poor performance in handling some special input data with improved partition algorithm, or try to uniform the possibilities among cases.

12.4.1 Engineering solution to duplicated elements

As presented in the exercise of above section, N. Lomuto's partition method isn't good at handling sequence with many duplicated elements. Consider a sequence with n equal elements like: $\{x, x, ..., x\}$. There are actually two methods to sort it.

1. The normal basic quick sort: That we select an arbitrary element, which is x as the pivot, partition it to two sub sequences, one is $\{x, x, ..., x\}$, which contains $n-1$ elements, the other is empty. then recursively sort the first one; this is obviously quadratic $O(n^2)$ solution.

2. The other way is to only pick those elements strictly smaller than x, and strictly greater than x. Such partition results two empty sub sequences, and n elements equal to the pivot. Next we recursively sort the sub sequences contains the smaller and the bigger elements, since both of them are empty, the recursive call returns immediately; The only thing left is to concatenate the sort results in front of and after the list of elements which are equal to the pivot.

www.dbooks.org

The latter one performs in $O(n)$ time if all elements are equal. This indicates an important improvement for partition. That instead of binary partition (split to two sub lists and a pivot), ternary partition (split to three sub lists) handles duplicated elements better.

We can define the ternary quick sort as the following.

$$sort(L) = \begin{cases} \phi & : & L = \phi \\ sort(S) \cup sort(E) \cup sort(G) & : & otherwise \end{cases} \tag{12.22}$$

Where S, E, G are sub lists contains all elements which are less than, equal to, and greater than the pivot respectively.

$$\begin{aligned} S &= \{x | x \in L, x < l_1\} \\ E &= \{x | x \in L, x = l_1\} \\ G &= \{x | x \in L, l_1 < x\} \end{aligned}$$

The basic ternary quick sort can be implemented in Haskell as the following example code.

```
sort [] = []
sort (x:xs) = sort [a | a←xs, a<x] ++
              x:[b | b←xs, b==x] ++ sort [c | c←xs, c>x]
```

Note that the comparison between elements must support abstract 'less-than' and 'equal-to' operations. The basic version of ternary sort takes linear $O(n)$ time to concatenate the three sub lists. It can be improved by using the standard techniques of accumulator.

Suppose function $sort'(L, A)$ is the accumulated ternary quick sort definition, that L is the sequence to be sorted, and the accumulator A contains the intermediate sorted result so far. We initialize the sorting with an empty accumulator: $sort(L) = sort'(L, \phi)$.

It's easy to give the trivial edge cases like below.

$$sort'(L, A) = \begin{cases} A & : & L = \phi \\ ... & : & otherwise \end{cases}$$

For the recursive case, as the ternary partition splits to three sub lists S, E, G, only S and G need recursive sort, E contains all elements equal to the pivot, which is in correct order thus needn't to be sorted any more. The idea is to sort G with accumulator A, then concatenate it behind E, then use this result as the new accumulator, and start to sort S:

$$sort'(L, A) = \begin{cases} A & : & L = \phi \\ sort(S, E \cup sort(G, A)) & : & otherwise \end{cases} \tag{12.23}$$

The partition can also be realized with accumulators. It is similar to what has been developed for the basic version of quick sort. Note that we can't just pass only one predication for pivot comparison. It actually needs two, one for

less-than, the other for equality testing. For the sake of brevity, we pass the pivot element instead.

$$partition(p, L, S, E, G) = \begin{cases} (S, E, G) & : & L = \phi \\ partition(p, L', \{l_1\} \cup S, E, G) & : & l_1 < p \\ partition(p, L', S, \{l_1\} \cup E, G) & : & l_1 = p \\ partition(p, L', S, E, \{l_1\} \cup G) & : & p < l_1 \end{cases} \tag{12.24}$$

Where l_1 is the first element in L if L isn't empty, and L' contains all rest elements except for l_1. Below Haskell program implements this algorithm. It starts the recursive sorting immediately in the edge case of parition.

```
sort xs = sort' xs []

sort' []     r = r
sort' (x:xs) r = part xs [] [x] [] r where
    part [] as bs cs r = sort' as (bs ++ sort' cs r)
    part (x':xs') as bs cs r | x' <  x = part xs' (x':as) bs cs r
                             | x' == x = part xs' as (x':bs) cs r
                             | x' >  x = part xs' as bs (x':cs) r
```

Richard Bird developed another version in [1], that instead of concatenating the recursively sorted results, it uses a list of sorted sub lists, and performs concatenation finally.

```
sort xs = concat $ pass xs []

pass [] xss = xss
pass (x:xs) xss = step xs [] [x] [] xss where
    step [] as bs cs xss = pass as (bs:pass cs xss)
    step (x':xs') as bs cs xss | x' <  x = step xs' (x':as) bs cs xss
                               | x' == x = step xs' as (x':bs) cs xss
                               | x' >  x = step xs' as bs (x':cs) xss
```

2-way partition

The cases with many duplicated elements can also be handled imperatively. Robert Sedgewick presented a partition method [3], [4] which holds two pointers. One moves from left to right, the other moves from right to left. The two pointers are initialized as the left and right boundaries of the array.

When start partition, the left most element is selected as the pivot. Then the left pointer i keeps advancing to right until it meets any element which is not less than the pivot; On the other hand[3], The right pointer j repeatedly scans to left until it meets any element which is not greater than the pivot.

At this time, all elements before the left pointer i are strictly less than the pivot, while all elements after the right pointer j are greater than the pivot. i points to an element which is either greater than or equal to the pivot; while j points to an element which is either less than or equal to the pivot, the situation at this stage is illustrated in figure 12.4 (a).

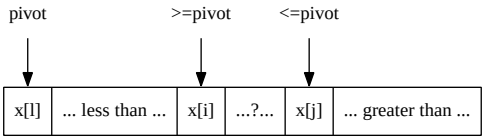
In order to partition all elements less than or equal to the pivot to the left, and the others to the right, we can exchange the two elements pointed by i, and

[3]We don't use 'then' because it's quite OK to perform the two scans in parallel.

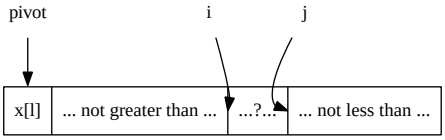
www.dbooks.org

j. After that the scan can be resumed. We repeat this process until either i meets j, or they overlap.

At any time point during partition. There is invariant that all elements before i (including the one pointed by i) are not greater than the pivot; while all elements after j (including the one pointed by j) are not less than the pivot. The elements between i and j haven't been examined yet. This invariant is shown in figure 12.4 (b).



(a) When pointer i, and j stop



(b) Partition invariant

Figure 12.4: Partition a range of array by using the left most element as the pivot.

After the left pointer i meets the right pointer j, or they overlap each other, we need one extra exchanging to move the pivot located at the first position to the correct place which is pointed by j. Next, the elements between the lower bound and j as well as the sub slice between i and the upper bound of the array are recursively sorted.

This algorithm can be described as the following.

```
1:  procedure SORT(A, l, u)                                  ▷ sort range [l, u)
2:      if u − l > 1 then          ▷ More than 1 element for non-trivial case
3:          i ← l, j ← u
4:          pivot ← A[l]
5:          loop
6:              repeat
7:                  i ← i + 1
8:              until A[i] ≥ pivot    ▷ Need handle error case that i ≥ u in fact.
9:              repeat
10:                 j ← j − 1
11:             until A[j] ≤ pivot    ▷ Need handle error case that j < l in fact.
12:             if j < i then
13:                 break
14:             EXCHANGE A[i] ↔ A[j]
15:         EXCHANGE A[l] ↔ A[j]                                 ▷ Move the pivot
16:         SORT(A, l, j)
17:         SORT(A, i, u)
```

Consider the extreme case that all elements are equal, this in-place quick sort

will partition the list to two equal length sub lists although it takes $\frac{n}{2}$ unnecessary swaps. As the partition is balanced, the overall performance is $O(n \lg n)$, which avoid downgrading to quadratic. The following ANSI C example program implements this algorithm.

```
void qsort(Key* xs, int l, int u) {
    int i, j, pivot;
    if (l < u - 1) {
        pivot = i = l; j = u;
        while (1) {
            while (i < u && xs[++i] < xs[pivot]);
            while (j >=l && xs[pivot] < xs[--j]);
            if (j < i) break;
            swap(xs[i], xs[j]);
        }
        swap(xs[pivot], xs[j]);
        qsort(xs, l, j);
        qsort(xs, i, u);
    }
}
```

Comparing this algorithm with the basic version based on N. Lumoto's partition method, we can find that it swaps fewer elements, because it skips those have already in proper sides of the pivot.

3-way partition

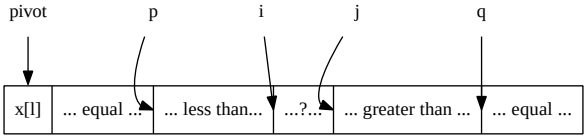
It's obviously that, we should avoid those unnecessary swapping for the duplicated elements. What's more, the algorithm can be developed with the idea of ternary sort (as known as 3-way partition in some materials), that all the elements which are strictly less than the pivot are put to the left sub slice, while those are greater than the pivot are put to the right. The middle part holds all the elements which are equal to the pivot. With such ternary partition, we need only recursively sort the ones which differ from the pivot. Thus in the above extreme case, there aren't any elements need further sorting. So the overall performance is linear $O(n)$.

The difficulty is how to do the 3-way partition. Jon Bentley and Douglas McIlroy developed a solution which keeps those elements equal to the pivot at the left most and right most sides as shown in figure 12.5 (a) [5] [6].

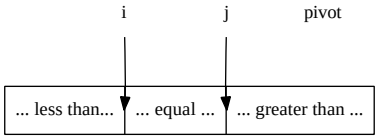
The majority part of scan process is as same as the one developed by Robert Sedgewick, that i and j keep advancing toward each other until they meet any element which is greater then or equal to the pivot for i, or less than or equal to the pivot for j respectively. At this time, if i and j don't meet each other or overlap, they are not only exchanged, but also examined if the elements pointed by them are identical to the pivot. Then necessary exchanging happens between i and p, as well as j and q.

By the end of the partition process, the elements equal to the pivot need to be swapped to the middle part from the left and right ends. The number of such extra exchanging operations are proportion to the number of duplicated elements. It's zero operation if elements are unique which there is no overhead in the case. The final partition result is shown in figure 12.5 (b). After that we only need recursively sort the 'less-than' and 'greater-than' sub slices.

www.dbooks.org



(a) Invariant of 3-way partition



(b) Swapping the equal parts to the middle

Figure 12.5: 3-way partition.

This algorithm can be given by modifying the 2-way partition as below.

```
1:  procedure SORT(A, l, u)
2:      if u − l > 1 then
3:          i ← l, j ← u
4:          p ← l, q ← u                ▷ points to the boundaries for equal elements
5:          pivot ← A[l]
6:          loop
7:              repeat
8:                  i ← i + 1
9:              until A[i] ≥ pivot              ▷ Skip the error handling for i ≥ u
10:             repeat
11:                 j ← j − 1
12:             until A[j] ≤ pivot              ▷ Skip the error handling for j < l
13:             if j ≤ i then
14:                 break          ▷ Note the difference form the above algorithm
15:             EXCHANGE A[i] ↔ A[j]
16:             if A[i] = pivot then                ▷ Handle the equal elements
17:                 p ← p + 1
18:                 EXCHANGE A[p] ↔ A[i]
19:             if A[j] = pivot then
20:                 q ← q − 1
21:                 EXCHANGE A[q] ↔ A[j]
22:         if i = j ∧ A[i] = pivot then                        ▷ A special case
23:             j ← j − 1, i ← i + 1
24:         for k from l to p do   ▷ Swap the equal elements to the middle part
25:             EXCHANGE A[k] ↔ A[j]
26:             j ← j − 1
27:         for k from u − 1 down-to q do
28:             EXCHANGE A[k] ↔ A[i]
29:             i ← i + 1
30:         SORT(A, l, j + 1)
```

31: SORT(A, i, u)

This algorithm can be translated to the following ANSI C example program.

```
void qsort2(Key* xs, int l, int u) {
    int i, j, k, p, q, pivot;
    if (l < u - 1) {
        i = p = l; j = q = u; pivot = xs[l];
        while (1) {
            while (i < u && xs[++i] < pivot);
            while (j >= l && pivot < xs[--j]);
            if (j <= i) break;
            swap(xs[i], xs[j]);
            if (xs[i] == pivot) { ++p; swap(xs[p], xs[i]); }
            if (xs[j] == pivot) { --q; swap(xs[q], xs[j]); }
        }
        if (i == j && xs[i] == pivot) { --j, ++i; }
        for (k = l; k <= p; ++k, --j) swap(xs[k], xs[j]);
        for (k = u-1; k >= q; --k, ++i) swap(xs[k], xs[i]);
        qsort2(xs, l, j + 1);
        qsort2(xs, i, u);
    }
}
```

It can be seen that the the algorithm turns to be a bit complex when it evolves to 3-way partition. There are some tricky edge cases should be handled with caution. Actually, we just need a ternary partition algorithm. This remind us the N. Lumoto's method, which is straightforward enough to be a start point.

The idea is to change the invariant a bit. We still select the first element as the pivot, as shown in figure 12.6, at any time, the left most section contains elements which are strictly less than the pivot; the next section contains the elements equal to the pivot; the right most section holds all the elements which are strictly greater than the pivot. The boundaries of three sections are marked as i, k, and j respectively. The rest part, which is between k and j are elements haven't been scanned yet.

At the beginning of this algorithm, the 'less-than' section is empty; the 'equal-to' section contains only one element, which is the pivot; so that i is initialized to the lower bound of the array, and k points to the element next to i. The 'greater-than' section is also initialized as empty, thus j is set to the upper bound.

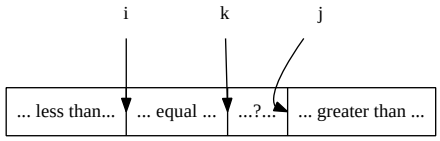


Figure 12.6: 3-way partition based on N. Lumoto's method.

When the partition process starts, the element pointed by k is examined. If it's equal to the pivot, k just advances to the next one; If it's greater than the pivot, we swap it with the last element in the unknown area, so that the length of 'greater-than' section increases by one. It's boundary j moves to the

www.dbooks.org

left. Since we don't know if the elements swapped to k is still greater than the pivot, it should be examined again repeatedly. Otherwise, if the element is less than the pivot, we can exchange it with the first one in the 'equal-to' section to resume the invariant. The partition algorithm stops when k meets j.

```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
3:         i ← l, j ← u, k ← l + 1
4:         pivot ← A[i]
5:         while k < j do
6:             while pivot < A[k] do
7:                 j ← j − 1
8:                 EXCHANGE A[k] ↔ A[j]
9:             if A[k] < pivot then
10:                EXCHANGE A[k] ↔ A[i]
11:                i ← i + 1
12:            k ← k + 1
13:        SORT(A, l, i)
14:        SORT(A, j, u)
```

Compare this one with the previous 3-way partition quick sort algorithm, it's more simple at the cost of more swapping operations. Below ANSI C program implements this algorithm.

```
void qsort(Key* xs, int l, int u) {
    int i, j, k; Key pivot;
    if (l < u - 1) {
        i = l; j = u; pivot = xs[l];
        for (k = l + 1; k < j; ++k) {
            while (pivot < xs[k]) { --j; swap(xs[j], xs[k]); }
            if (xs[k] < pivot) { swap(xs[i], xs[k]); ++i; }
        }
        qsort(xs, l, i);
        qsort(xs, j, u);
    }
}
```

Exercise 12.3

- All the quick sort imperative algorithms given in this section use the first element as the pivot, another method is to choose the last one as the pivot. Realize the quick sort algorithms, including the basic version, Sedgewick version, and ternary (3-way partition) version by using this approach.

12.5 Engineering solution to the worst case

Although the ternary quick sort (3-way partition) solves the issue for duplicated elements, it can't handle some typical worst cases. For example if many of the elements in the sequence are ordered, no matter it's in ascending or descending order, the partition result will be two unbalanced sub sequences, one with few elements, the other contains all the rest.

Consider the two extreme cases, $\{x_1 < x_2 < ... < x_n\}$ and $\{y_1 > y_2 > ... > y_n\}$. The partition results are shown in figure 12.7.

It's easy to give some more worst cases, for example, $\{x_m, x_{m-1}, ..., x_2, x_1, x_{m+1}, x_{m+2}, ...x_n\}$ where $\{x_1 < x_2 < ... < x_n\}$; Another one is $\{x_n, x_1, x_{n-1}, x_2, ...\}$. Their partition result trees are shown in figure 12.8.

Observing that the bad partition happens easily when blindly choose the first element as the pivot, there is a popular work around suggested by Robert Sedgwick in [3]. Instead of selecting the fixed position in the sequence, a small sampling helps to find a pivot which has lower possibility to cause a bad partition. One option is to examine the first element, the middle, and the last one, then choose the median of these three element. In the worst case, it can ensure that there is at least one element in the shorter partitioned sub list.

Note that there is one tricky in real-world implementation. Since the index is typically represented in limited length words, it may cause overflow when calculating the middle index by the naive expression `(l + u) / 2`. In order to avoid this issue, it can be accessed as `l + (u - l) / 2`. There are two methods to find the median, one needs at most three comparisons [5]; the other is to move the minimum value to the first location, the maximum value to the last location, and the median value to the meddle location by swapping. After that we can select the middle as the pivot. Below algorithm illustrated the second idea before calling the partition procedure.

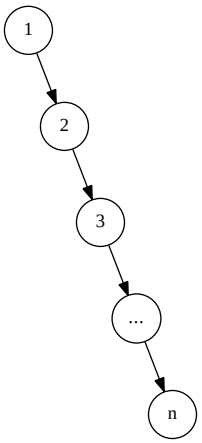
```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
3:         m ← ⌊(l+u)/2⌋                    ▷ Need handle overflow error in practice
4:         if A[m] < A[l] then                              ▷ Ensure A[l] ≤ A[m]
5:             EXCHANGE A[l] ↔ A[m]
6:         if A[u − 1] < A[l] then                          ▷ Ensure A[l] ≤ A[u − 1]
7:             EXCHANGE A[l] ↔ A[u − 1]
8:         if A[u − 1] < A[m] then                          ▷ Ensure A[m] ≤ A[u − 1]
9:             EXCHANGE A[m] ↔ A[u − 1]
10:        EXCHANGE A[l] ↔ A[m]
11:        (i, j) ← PARTITION(A, l, u)
12:        SORT(A, l, i)
13:        SORT(A, j, u)
```

It's obviously that this algorithm performs well in the 4 special worst cases given above. The imperative implementation of median-of-three is left as exercise to the reader.

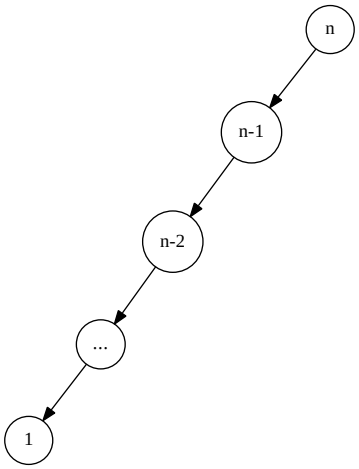
However, in purely functional settings, it's expensive to randomly access the middle and the last element. We can't directly translate the imperative median selection algorithm. The idea of taking a small sampling and then finding the median element as pivot can be realized alternatively by taking the first 3. For example, in the following Haskell program.

```
qsort [] = []
qsort [x] = [x]
qsort [x, y] = [min x y, max x y]
qsort (x:y:z:rest) = qsort (filter (< m) (s:rest)) ++ [m] ++ qsort (filter (≥ m) (l:rest)) where
    xs = [x, y, z]
    [s, m, l] = [minimum xs, median xs, maximum xs]
```

www.dbooks.org

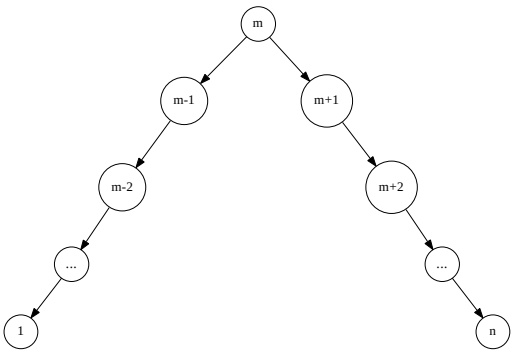


(a) The partition tree for $\{x_1 < x_2 < ... < x_n\}$, There aren't any elements less than or equal to the pivot (the first element) in every partition.

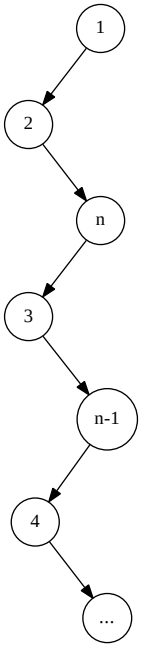


(b) The partition tree for $\{y_1 > y_2 > ... > y_n\}$, There aren't any elements greater than or equal to the pivot (the first element) in every partition.

Figure 12.7: The two worst cases.



(a) Except for the first partition, all the others are unbalanced.



(b) A zig-zag partition tree.

Figure 12.8: Another two worst cases.

www.dbooks.org

Unfortunately, none of the above 4 worst cases can be well handled by this program, this is because the sampling is not good. We need telescope, but not microscope to profile the whole list to be partitioned. We'll see the functional way to solve the partition problem later.

Except for the median-of-three, there is another popular engineering practice to get good partition result. instead of always taking the first element or the last one as the pivot. One alternative is to randomly select one. For example as the following modification.

```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
3:         EXCHANGE A[l] ↔ A[ RANDOM(l, u) ]
4:         (i, j) ← PARTITION(A, l, u)
5:         SORT(A, l, i)
6:         SORT(A, j, u)
```

The function $\text{RANDOM}(l, u)$ returns a random integer i between l and u, that $l \leq i < u$. The element at this position is exchanged with the first one, so that it is selected as the pivot for the further partition. This algorithm is called *random quick sort* [2].

Theoretically, neither median-of-three nor random quick sort can avoid the worst case completely. If the sequence to be sorted is randomly distributed, no matter choosing the first one as the pivot, or the any other arbitrary one are equally in effect. Considering the underlying data structure of the sequence is singly linked-list in functional setting, it's expensive to strictly apply the idea of random quick sort in purely functional approach.

Even with this bad news, the engineering improvement still makes sense in real world programming.

12.6 Other engineering practice

There is some other engineering practice which doesn't focus on solving the bad partition issue. Robert Sedgewick observed that when the list to be sorted is short, the overhead introduced by quick sort is relative expense, on the other hand, the insertion sort performs better in such case [4], [5]. Sedgewick, Bentley and McIlroy tried different threshold, as known as 'Cut-Off', that when there are lesson than 'Cut-Off' elements, the sort algorithm falls back to insertion sort.

```
1: procedure SORT(A, l, u)
2:     if u − l > CUT-OFF then
3:         QUICK-SORT(A, l, u)
4:     else
5:         INSERTION-SORT(A, l, u)
```

The implementation of this improvement is left as exercise to the reader.

Exercise 12.4

- Can you figure out more quick sort worst cases besides the four given in this section?

- Implement median-of-three method in your favorite imperative programming language.
- Implement random quick sort in your favorite imperative programming language.
- Implement the algorithm which falls back to insertion sort when the length of list is small in both imperative and functional approach.

12.7 Side words

It's sometimes called 'true quick sort' if the implementation equipped with most of the engineering practice we introduced, including insertion sort fall-back with cut-off, in-place exchanging, choose the pivot by median-of-three method, 3-way-partition.

The purely functional one, which express the idea of quick sort perfect can't take all of them. Thus someone think the functional quick sort is essentially tree sort.

Actually, quick sort does have close relationship with tree sort. Richard Bird shows how to derive quick sort from binary tree sort by deforestation [7].

Consider a binary search tree creation algorithm called $unfold$. Which turns a list of elements into a binary search tree.

$$unfold(L) = \begin{cases} \phi & : & L = \phi \\ tree(T_l, l_1, T_r) & : & otherwise \end{cases} \tag{12.25}$$

Where

$$\begin{aligned} T_l &= unfold(\{a | a \in L', a \leq l_1\}) \\ T_r &= unfold(\{a | a \in L', l_1 < a\}) \end{aligned} \tag{12.26}$$

The interesting point is that, this algorithm creates tree in a different way as we introduced in the chapter of binary search tree. If the list to be unfold is empty, the result is obviously an empty tree. This is the trivial edge case; Otherwise, the algorithm set the first element l_1 in the list as the key of the node, and recursively creates its left and right children. Where the elements used to form the left child are those which are less than or equal to the key in L', while the rest elements which are greater than the key are used to form the right child.

Remind the algorithm which turns a binary search tree to a list by in-order traversing:

$$toList(T) = \begin{cases} \phi & : & T = \phi \\ toList(left(T)) \cup \{key(T)\} \cup toList(right(T)) & : & otherwise \end{cases} \tag{12.27}$$

We can define quick sort algorithm by composing these two functions.

$$quickSort = toList \cdot unfold \tag{12.28}$$

The binary search tree built in the first step of applying $unfold$ is the intermediate result. This result is consumed by $toList$ and dropped after the second

www.dbooks.org

step. It's quite possible to eliminate this intermediate result, which leads to the basic version of quick sort.

The elimination of the intermediate binary search tree is called *deforestation*. This concept is based on Burstle-Darlington's work [9].

12.8 Merge sort

Although quick sort performs perfectly in average cases, it can't avoid the worst case no matter what engineering practice is applied. Merge sort, on the other kind, ensure the performance is bound to $O(n \lg n)$ in all the cases. It's particularly useful in theoretical algorithm design and analysis. Another feature is that merge sort is friendly for linked-space settings, which is suitable for sorting nonconsecutive stored sequences. Some functional programming and dynamic programming environments adopt merge sort as the standard library sorting solution, such as Haskel, Python and Java (later than Java 7).

In this section, we'll first brief the intuitive idea of merge sort, provide a basic version. After that, some variants of merge sort will be given including nature merge sort, and bottom-up merge sort.

12.8.1 Basic version

Same as quick sort, the essential idea behind merge sort is also divide and conquer. Different from quick sort, merge sort enforces the divide to be strictly balanced, that it always splits the sequence to be sorted at the middle point. After that, it recursively sort the sub sequences and merge the sorted two sequences to the final result. The algorithm can be described as the following.

In order to sort a sequence L,

- Trivial edge case: If the sequence to be sorted is empty, the result is obvious empty;
- Otherwise, split the sequence at the middle position, recursively sort the two sub sequences and merge the result.

The basic merge sort algorithm can be formalized with the following equation.

$$sort(L) = \begin{cases} \phi & : \quad L = \phi \\ merge(sort(L_1), sort(L_2)) & : \quad otherwise, (L_1, L_2) = splitAt(\lfloor \frac{|L|}{2} \rfloor, L) \end{cases} \tag{12.29}$$

Merge

There are two 'black-boxes' in the above merge sort definition, one is the *splitAt* function, which splits a list at a given position; the other is the *merge* function, which can merge two sorted lists into one.

As presented in the appendix of this book, it's trivial to realize *splitAt* in imperative settings by using random access. However, in functional settings,

it's typically realized as a linear algorithm:

$$splitAt(n, L) = \begin{cases} (\phi, L) & : \quad n = 0 \\ (\{l_1\} \cup A, B) & : \quad otherwise, (A, B) = splitAt(n-1, L') \end{cases} \tag{12.30}$$

Where l_1 is the first element of L, and L' represents the rest elements except of l_1 if L isn't empty.

The idea of merge can be illustrated as in figure 12.9. Consider two lines of kids. The kids have already stood in order of their heights. that the shortest one stands at the first, then a taller one, the tallest one stands at the end of the line.

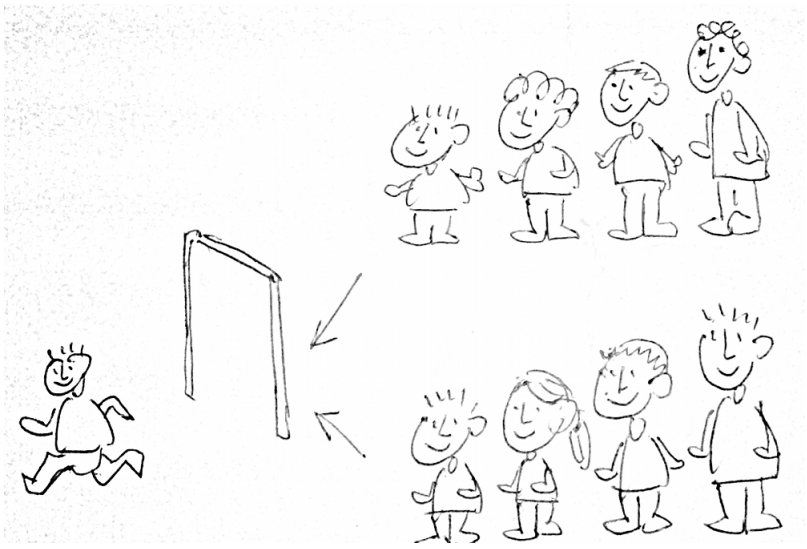


Figure 12.9: Two lines of kids pass a door.

Now let's ask the kids to pass a door one by one, every time there can be at most one kid pass the door. The kids must pass this door in the order of their height. The one can't pass the door before all the kids who are shorter than him/her.

Since the two lines of kids have already been 'sorted', the solution is to ask the first two kids, one from each line, compare their height, and let the shorter kid pass the door; Then they repeat this step until one line is empty, after that, all the rest kids can pass the door one by one.

This idea can be formalized in the following equation.

$$merge(A, B) = \begin{cases} A & : \quad B = \phi \\ B & : \quad A = \phi \\ \{a_1\} \cup merge(A', B) & : \quad a_1 \leq b_1 \\ \{b_1\} \cup merge(A, B') & : \quad otherwise \end{cases} \tag{12.31}$$

Where a_1 and b_1 are the first elements in list A and B; A' and B' are the rest elements except for the first ones respectively. The first two cases are trivial

www.dbooks.org

edge cases. That merge one sorted list with an empty list results the same sorted list; Otherwise, if both lists are non-empty, we take the first elements from the two lists, compare them, and use the minimum as the first one of the result, then recursively merge the rest.

With *merge* defined, the basic version of merge sort can be implemented like the following Haskell example code.

```
msort [] = []
msort [x] = [x]
msort xs = merge (msort as) (msort bs) where
  (as, bs) = splitAt (length xs `div` 2) xs

merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x ≤ y = x : merge xs (y:ys)
                    | x >  y = y : merge (x:xs) ys
```

Note that, the implementation differs from the algorithm definition that it treats the singleton list as trivial edge case as well.

Merge sort can also be realized imperatively. The basic version can be developed as the below algorithm.

```
1: procedure SORT(A)
2:     if |A| > 1 then
3:         m ← ⌊|A|/2⌋
4:         X ← COPY-ARRAY(A[1...m])
5:         Y ← COPY-ARRAY(A[m + 1...|A|])
6:         SORT(X)
7:         SORT(Y)
8:         MERGE(A, X, Y)
```

When the array to be sorted contains at least two elements, the non-trivial sorting process starts. It first copy the first half to a new created array X, and the second half to a second new array Y. Recursively sort them; and finally merge the sorted result back to A.

This version uses the same amount of extra spaces of A. This is because the MERGE algorithm isn't in-place at the moment. We'll introduce the imperative in-place merge sort in later section.

The merge process almost does the same thing as the functional definition. There is a verbose version and a simplified version by using sentinel.

The verbose merge algorithm continuously checks the element from the two input arrays, picks the smaller one and puts it back to the result array A, it then advances along the arrays respectively until either one input array is exhausted. After that, the algorithm appends the rest of the elements in the other input array to A.

```
1: procedure MERGE(A, X, Y)
2:     i ← 1, j ← 1, k ← 1
3:     m ← |X|, n ← |Y|
4:     while i ≤ m ∧ j ≤ n do
5:         if X[i] < Y[j] then
6:             A[k] ← X[i]
7:             i ← i + 1
8:         else
```

```
9:              A[k] ← Y[j]
10:             j ← j + 1
11:         k ← k + 1
12:     while i ≤ m do
13:         A[k] ← X[i]
14:         k ← k + 1
15:         i ← i + 1
16:     while j ≤ n do
17:         A[k] ← Y[j]
18:         k ← k + 1
19:         j ← j + 1
```

Although this algorithm is a bit verbose, it can be short in some programming environment with enough tools to manipulate array. The following Python program is an example.

```
def msort(xs):
    n = len(xs)
    if n > 1:
        ys = [x for x in xs[:n/2]]
        zs = [x for x in xs[n/2:]]
        ys = msort(ys)
        zs = msort(zs)
        xs = merge(xs, ys, zs)
    return xs

def merge(xs, ys, zs):
    i = 0
    while ys != [] and zs != []:
        xs[i] = ys.pop(0) if ys[0] < zs[0] else zs.pop(0)
        i = i + 1
    xs[i:] = ys if ys !=[] else zs
    return xs
```

Performance

Before dive into the improvement of this basic version, let's analyze the performance of merge sort. The algorithm contains two steps, divide step, and merge step. In divide step, the sequence to be sorted is always divided into two sub sequences with the same length. If we draw a similar partition tree as what we did for quick sort, it can be found this tree is a perfectly balanced binary tree as shown in figure 12.3. Thus the height of this tree is $O(\lg n)$. It means the recursion depth of merge sort is bound to $O(\lg n)$. Merge happens in every level. It's intuitive to analyze the merge algorithm, that it compare elements from two input sequences in pairs, after one sequence is fully examined the rest one is copied one by one to the result, thus it's a linear algorithm proportion to the length of the sequence. Based on this facts, denote $T(n)$ the time for sorting

www.dbooks.org

the sequence with length n, we can write the recursive time cost as below.

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + T(\frac{n}{2}) + cn \\ &= 2T(\frac{n}{2}) + cn \end{aligned} \tag{12.32}$$

It states that the cost consists of three parts: merge sort the first half takes $T(\frac{n}{2})$, merge sort the second half takes also $T(\frac{n}{2})$, merge the two results takes cn, where c is some constant. Solve this equation gives the result as $O(n \lg n)$.

Note that, this performance doesn't vary in all cases, as merge sort always uniformly divides the input.

Another significant performance indicator is space occupation. However, it varies a lot in different merge sort implementation. The detail space bounds analysis will be explained in every detailed variants later.

For the basic imperative merge sort, observe that it demands same amount of spaces as the input array in every recursion, copies the original elements to them for recursive sort, and these spaces can be released after this level of recursion. So the peak space requirement happens when the recursion enters to the deepest level, which is $O(n \lg n)$.

The functional merge sort consume much less than this amount, because the underlying data structure of the sequence is linked-list. Thus it needn't extra spaces for merge[4]. The only spaces requirement is for book-keeping the stack for recursive calls. This can be seen in the later explanation of even-odd split algorithm.

Minor improvement

We'll next improve the basic merge sort bit by bit for both the functional and imperative realizations. The first observation is that the imperative merge algorithm is a bit verbose. [2] presents an elegant simplification by using positive ∞ as the sentinel. That we append ∞ as the last element to the both ordered arrays for merging[5]. Thus we needn't test which array is not exhausted. Figure 12.10 illustrates this idea.

```
1:  procedure MERGE(A, X, Y)
2:      APPEND(X, ∞)
3:      APPEND(Y, ∞)
4:      i ← 1, j ← 1
5:      for k ← from 1 to |A| do
6:          if X[i] < Y[j] then
7:              A[k] ← X[i]
8:              i ← i + 1
9:          else
10:             A[k] ← Y[j]
11:             j ← j + 1
```

The following ANSI C program imlements this idea. It embeds the merge inside. `INF` is defined as a big constant number with the same type of `Key`. Where the type can either be defined elsewhere or we can abstract the type information

[4]The complex effects caused by lazy evaluation is ignored here, please refer to [7] for detail

[5]For sorting in monotonic non-increasing order, $-\infty$ can be used instead

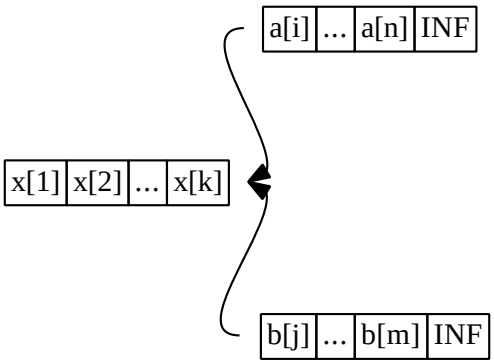


Figure 12.10: Merge with ∞ as sentinels.

by passing the comparator as parameter. We skip these implementation and language details here.

```
void msort(Key* xs, int l, int u) {
    int i, j, m;
    Key *as, *bs;
    if (u - l > 1) {
        m = l + (u - l) / 2;  /* avoid int overflow */
        msort(xs, l, m);
        msort(xs, m, u);
        as = (Key*) malloc(sizeof(Key) * (m - l + 1));
        bs = (Key*) malloc(sizeof(Key) * (u - m + 1));
        memcpy((void*)as, (void*)(xs + l), sizeof(Key) * (m - l));
        memcpy((void*)bs, (void*)(xs + m), sizeof(Key) * (u - m));
        as[m - l] = bs[u - m] = INF;
        for (i = j = 0; l < u; ++l)
            xs[l] = as[i] < bs[j] ? as[i++] : bs[j++];
        free(as);
        free(bs);
    }
}
```

Running this program takes much more time than the quick sort. Besides the major reason we'll explain later, one problem is that this version frequently allocates and releases memories for merging. While memory allocation is one of the well known bottle-neck in real world as mentioned by Bentley in [4]. One solution to address this issue is to allocate another array with the same size to the original one as the working area. The recursive sort for the first and second halves needn't allocate any more extra spaces, but use the working area when merging. Finally, the algorithm copies the merged result back.

This idea can be expressed as the following modified algorithm.

1: **procedure** SORT(A)
2: $B \leftarrow$ CREATE-ARRAY($|A|$)
3: SORT'$(A, B, 1, |A|)$

4: **procedure** SORT'(A, B, l, u)

www.dbooks.org

```
5:    if u − l > 0 then
6:        m ← ⌊(l+u)/2⌋
7:        SORT'(A, B, l, m)
8:        SORT'(A, B, m + 1, u)
9:        MERGE'(A, B, l, m, u)
```

This algorithm duplicates another array, and pass it along with the original array to be sorted to SORT' algorithm. In real implementation, this working area should be released either manually, or by some automatic tool such as GC (Garbage collection). The modified algorithm MERGE' also accepts a working area as parameter.

```
1: procedure MERGE'(A, B, l, m, u)
2:     i ← l, j ← m + 1, k ← l
3:     while i ≤ m ∧ j ≤ u do
4:         if A[i] < A[j] then
5:             B[k] ← A[i]
6:             i ← i + 1
7:         else
8:             B[k] ← A[j]
9:             j ← j + 1
10:        k ← k + 1
11:    while i ≤ m do
12:        B[k] ← A[i]
13:        k ← k + 1
14:        i ← i + 1
15:    while j ≤ u do
16:        B[k] ← A[j]
17:        k ← k + 1
18:        j ← j + 1
19:    for i ← from l to u do                    ▷ Copy back
20:        A[i] ← B[i]
```

By using this minor improvement, the space requirement reduced to $O(n)$ from $O(n \lg n)$. The following ANSI C program implements this minor improvement. For illustration purpose, we manually copy the merged result back to the original array in a loop. This can also be realized by using standard library provided tool, such as `memcpy`.

```
void merge(Key* xs, Key* ys, int l, int m, int u) {
    int i, j, k;
    i = k = l; j = m;
    while (i < m && j < u)
        ys[k++] = xs[i] < xs[j] ? xs[i++] : xs[j++];
    while (i < m)
        ys[k++] = xs[i++];
    while (j < u)
        ys[k++] = xs[j++];
    for(; l < u; ++l)
        xs[l] = ys[l];
}

void msort(Key* xs, Key* ys, int l, int u) {
```

```
    int m;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        msort(xs, ys, l, m);
        msort(xs, ys, m, u);
        merge(xs, ys, l, m, u);
    }
}

void sort(Key* xs, int l, int u) {
    Key* ys = (Key*) malloc(sizeof(Key) * (u - l));
    kmsort(xs, ys, l, u);
    free(ys);
}
```

This new version runs faster than the previous one. In my test machine, it speeds up about 20% to 25% when sorting 100,000 randomly generated numbers.

The basic functional merge sort can also be fine tuned. Observe that, it splits the list at the middle point. However, as the underlying data structure to represent list is singly linked-list, random access at a given position is a linear operation (refer to appendix A for detail). Alternatively, one can split the list in an even-odd manner. That all the elements in even position are collected in one sub list, while all the odd elements are collected in another. As for any list, there are either same amount of elements in even and odd positions, or they differ by one. So this divide strategy always leads to well splitting, thus the performance can be ensured to be $O(n \lg n)$ in all cases.

The even-odd splitting algorithm can be defined as below.

$$split(L) = \begin{cases} (\phi, \phi) & : & L = \phi \\ (\{l_1\}, \phi) & : & |L| = 1 \\ (\{l_1\} \cup A, \{l_2\} \cup B) & : & otherwise, (A, B) = split(L'') \end{cases} \tag{12.33}$$

When the list is empty, the split result are two empty lists; If there is only one element in the list, we put this single element, which is at position 1, to the odd sub list, the even sub list is empty; Otherwise, it means there are at least two elements in the list, We pick the first one to the odd sub list, the second one to the even sub list, and recursively split the rest elements.

All the other functions are kept same, the modified Haskell program is given as the following.

```
split [] = ([], [])
split [x] = ([x], [])
split (x:y:xs) = (x:xs', y:ys') where (xs', ys') = split xs
```

12.9 In-place merge sort

One drawback for the imperative merge sort is that it requires extra spaces for merging, the basic version without any optimization needs $O(n \lg n)$ in peak time, and the one by allocating a working area needs $O(n)$.

www.dbooks.org

It's nature for people to seek the in-place version merge sort, which can reuse the original array without allocating any extra spaces. In this section, we'll introduce some solutions to realize imperative in-place merge sort.

12.9.1 Naive in-place merge

The first idea is straightforward. As illustrated in figure 12.11, sub list A, and B are sorted, when performs in-place merge, the invariant ensures that all elements before i are merged, so that they are in non-decreasing order; every time we compare the i-th and the j-th elements. If the i-th is less than the j-th, the marker i just advances one step to the next. This is the easy case. Otherwise, it means that the j-th element is the next merge result, which should be put in front of i. In order to achieve this, all elements between i and j, including the i-th should be shift to the end by one cell. We repeat this process till all the elements in A and B are put to the correct positions.

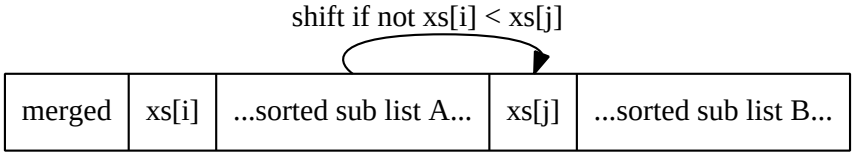


Figure 12.11: Naive in-place merge

```
1: procedure MERGE(A, l, m, u)
2:     while l ≤ m ∧ m ≤ u do
3:         if A[l] < A[m] then
4:             l ← l + 1
5:         else
6:             x ← A[m]
7:             for i ← m down-to l + 1 do                ▷ Shift
8:                 A[i] ← A[i − 1]
9:             A[l] ← x
```

However, this naive solution downgrades merge sort overall performance to quadratic $O(n^2)$! This is because that array shifting is a linear operation. It is proportion to the length of elements in the first sorted sub array which haven't been compared so far.

The following ANSI C program based on this algorithm runs very slow, that it takes about 12 times slower than the previous version when sorting 10,000 random numbers.

```
void naive_merge(Key* xs, int l, int m, int u) {
    int i; Key y;
    for(; l < m && m < u; ++l)
        if (!(xs[l] < xs[m])) {
            y = xs[m++];
            for (i = m - 1; i > l; --i) /* shift */
                xs[i] = xs[i-1];
            xs[l] = y;
        }
```

```
}

void msort3(Key* xs, int l, int u) {
    int m;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        msort3(xs, l, m);
        msort3(xs, m, u);
        naive_merge(xs, l, m, u);
    }
}
```

12.9.2 in-place working area

In order to implement the in-place merge sort in $O(n \lg n)$ time, when sorting a sub array, the rest part of the array must be reused as working area for merging. As the elements stored in the working area, will be sorted later, they can't be overwritten. We can modify the previous algorithm, which duplicates extra spaces for merging, a bit to achieve this. The idea is that, every time when we compare the first elements in the two sorted sub arrays, if we want to put the less element to the target position in the working area, we in-turn exchange what sored in the working area with this element. Thus after merging, the two sub arrays store what the working area previously contains. This idea can be illustrated in figure 12.12.

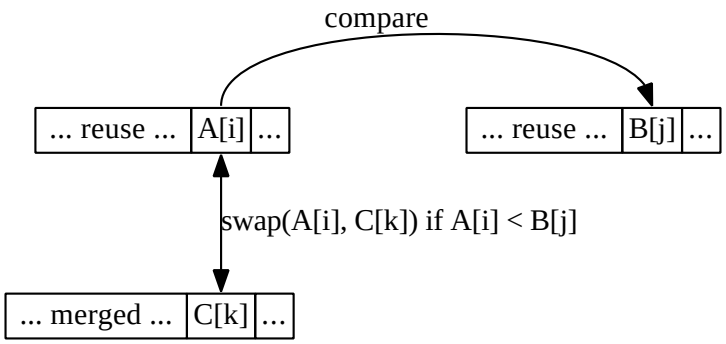


Figure 12.12: Merge without overwriting working area.

In our algorithm, both the two sorted sub arrays, and the working area for merging are parts of the original array to be sorted. we need supply the following arguments when merging: the start points and end points of the sorted sub arrays, which can be represented as ranges; and the start point of the working area. The following algorithm for example, uses $[a, b)$ to indicate the range include a, exclude b. It merges sorted range $[i, m)$ and range $[j, n)$ to the working area starts from k.

1: **procedure** MERGE($A, [i, m), [j, n), k$)
2: **while** $i < m \wedge j < n$ **do**
3: **if** $A[i] < A[j]$ **then**
4: EXCHANGE $A[k] \leftrightarrow A[i]$

www.dbooks.org

```
5:            i ← i + 1
6:        else
7:            EXCHANGE A[k] ↔ A[j]
8:            j ← j + 1
9:        k ← k + 1
10:   while i < m do
11:       EXCHANGE A[k] ↔ A[i]
12:       i ← i + 1
13:       k ← k + 1
14:   while j < m do
15:       EXCHANGE A[k] ↔ A[j]
16:       j ← j + 1
17:       k ← k + 1
```

Note that, the following two constraints must be satisfied when merging:

1. The working area should be within the bounds of the array. In other words, it should be big enough to hold elements exchanged in without causing any out-of-bound error;
2. The working area can be overlapped with either of the two sorted arrays, however, it should be ensured that there are not any unmerged elements being overwritten;

This algorithm can be implemented in ANSI C as the following example.

```
void wmerge(Key* xs, int i, int m, int j, int n, int w) {
    while (i < m && j < n)
        swap(xs, w++, xs[i] < xs[j] ? i++ : j++);
    while (i < m)
        swap(xs, w++, i++);
    while (j < n)
        swap(xs, w++, j++);
}
```

With this merging algorithm defined, it's easy to imagine a solution, which can sort half of the array; The next question is, how to deal with the rest of the unsorted part stored in the working area as shown in figure 12.13?

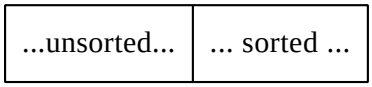


Figure 12.13: Half of the array is sorted.

One intuitive idea is to recursively sort another half of the working area, thus there are only $\frac{1}{4}$ elements haven't been sorted yet. Which is shown in figure 12.14. The key point at this stage is that we must merge the sorted $\frac{1}{4}$ elements B with the sorted $\frac{1}{2}$ elements A sooner or later.

Is the working area left, which only holds $\frac{1}{4}$ elements, big enough for merging A and B? Unfortunately, it isn't in the settings shown in figure 12.14.

However, the second constraint mentioned before gives us a hint, that we can exploit it by arranging the working area to overlap with either sub array

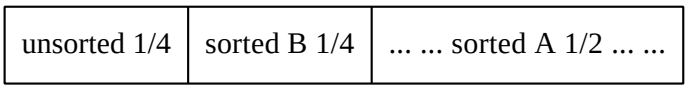
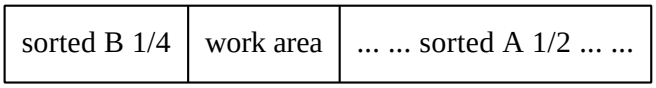


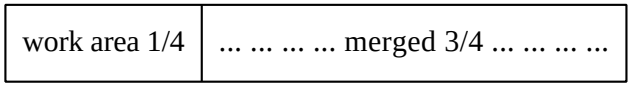
Figure 12.14: A and B must be merged at sometime.

if we can ensure the unmerged elements won't be overwritten under some well designed merging schema.

Actually, instead of making the second half of the working area be sorted, we can make the first half be sorted, and put the working area between the two sorted arrays as shown in figure 12.15 (a). This setup effects arranging the working area to overlap with the sub array A. This idea is proposed in [10].



(a)



(b)

Figure 12.15: Merge A and B with the working area.

Let's consider two extreme cases:

1. All the elements in B are less than any element in A. In this case, the merge algorithm finally moves the whole contents of B to the working area; the cells of B holds what previously stored in the working area; As the size of area is as same as B, it's OK to exchange their contents;

2. All the elements in A are less than any element in B. In this case, the merge algorithm continuously exchanges elements between A and the working area. After all the previous $\frac{1}{4}$ cells in the working area are filled with elements from A, the algorithm starts to overwrite the first half of A. Fortunately, the contents being overwritten are not those unmerged elements. The working area is in effect advances toward the end of the array, and finally moves to the right side; From this time point, the merge algorithm starts exchanging contents in B with the working area. The result is that the working area moves to the left most side which is shown in figure 12.15 (b).

We can repeat this step, that always sort the second half of the unsorted part, and exchange the sorted sub array to the first half as working area. Thus we keep reducing the working area from $\frac{1}{2}$ of the array, $\frac{1}{4}$ of the array, $\frac{1}{8}$ of the array, ... The scale of the merge problem keeps reducing. When there is only one element left in the working area, we needn't sort it any more since the singleton array is sorted by nature. Merging a singleton array to the other is equivalent to insert the element. In practice, the algorithm can finalize the last few elements by switching to insertion sort.

www.dbooks.org

The whole algorithm can be described as the following.

```
1: procedure SORT(A, l, u)
2:     if u − l > 0 then
3:         m ← ⌊(l+u)/2⌋
4:         w ← l + u − m
5:         SORT'(A, l, m, w)          ▷ The second half contains sorted elements
6:         while w − l > 1 do
7:             u' ← w
8:             w ← ⌈(l+u')/2⌉          ▷ Ensure the working area is big enough
9:             SORT'(A, w, u', l)      ▷ The first half holds the sorted elements
10:            MERGE(A, [l, l + u' − w], [u', u], w)
11:        for i ← w down-to l do                    ▷ Switch to insertion sort
12:            j ← i
13:            while j ≤ u ∧ A[j] < A[j − 1] do
14:                EXCHANGE A[j] ↔ A[j − 1]
15:                j ← j + 1
```

Note that in order to satisfy the first constraint, we must ensure the working area is big enough to hold all exchanged in elements, that's way we round it by ceiling when sort the second half of the working area. Note that we actually pass the ranges including the end points to the algorithm MERGE.

Next, we develop a Sort' algorithm, which mutually recursive call Sort and exchange the result to the working area.

```
1: procedure SORT'(A, l, u, w)
2:     if u − l > 0 then
3:         m ← ⌊(l+u)/2⌋
4:         SORT(A, l, m)
5:         SORT(A, m + 1, u)
6:         MERGE(A, [l, m], [m + 1, u], w)
7:     else                          ▷ Exchange all elements to the working area
8:         while l ≤ u do
9:             EXCHANGE A[l] ↔ A[w]
10:            l ← l + 1
11:            w ← w + 1
```

Different from the naive in-place sort, this algorithm doesn't shift the array during merging. The main algorithm reduces the unsorted part in sequence of $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, ...$, it takes $O(\lg n)$ steps to complete sorting. In every step, It recursively sorts half of the rest elements, and performs linear time merging.

Denote the time cost of sorting n elements as $T(n)$, we have the following equation.

$$T(n) = T(\frac{n}{2}) + c\frac{n}{2} + T(\frac{n}{4}) + c\frac{3n}{4} + T(\frac{n}{8}) + c\frac{7n}{8} + ... \tag{12.34}$$

Solving this equation by using telescope method, gets the result $O(n \lg n)$. The detailed process is left as exercise to the reader.

The following ANSI C code completes the implementation by using the example `wmerge` program given above.

```
void imsort(Key* xs, int l, int u);
```

```
void wsort(Key* xs, int l, int u, int w) {
    int m;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        imsort(xs, l, m);
        imsort(xs, m, u);
        wmerge(xs, l, m, m, u, w);
    }
    else
        while (l < u)
            swap(xs, l++, w++);
}

void imsort(Key* xs, int l, int u) {
    int m, n, w;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        w = l + u - m;
        wsort(xs, l, m, w); /* the last half contains sorted elements */
        while (w - l > 2) {
            n = w;
            w = l + (n - l + 1) / 2; /* ceiling */
            wsort(xs, w, n, l);  /* the first half contains sorted elements */
            wmerge(xs, l, l + n - w, n, u, w);
        }
        for (n = w; n > l; --n) /*switch to insertion sort*/
            for (m = n; m < u && xs[m] < xs[m-1]; ++m)
                swap(xs, m, m - 1);
    }
}
```

However, this program doesn't run faster than the version we developed in previous section, which doubles the array in advance as working area. In my machine, it is about 60% slower when sorting 100,000 random numbers due to many swap operations.

12.9.3 In-place merge sort vs. linked-list merge sort

The in-place merge sort is still a live area for research. In order to save the extra spaces for merging, some overhead has be introduced, which increases the complexity of the merge sort algorithm. However, if the underlying data structure isn't array, but linked-list, merge can be achieved without any extra spaces as shown in the even-odd functional merge sort algorithm presented in previous section.

In order to make it clearer, we can develop a purely imperative linked-list merge sort solution. The linked-list can be defined as a record type as shown in appendix A like below.

```
struct Node {
    Key key;
    struct Node* next;
};
```

www.dbooks.org

We can define an auxiliary function for node linking. Assume the list to be linked isn't empty, it can be implemented as the following.

```
struct Node* link(struct Node* x, struct Node* ys) {
    x→next = ys;
    return x;
}
```

One method to realize the imperative even-odd splitting, is to initialize two empty sub lists. Then iterate the list to be split. Every time, we link the current node in front of the first sub list, then exchange the two sub lists. So that, the second sub list will be linked at the next time iteration. This idea can be illustrated as below.

```
1: function SPLIT(L)
2:     (A, B) ← (φ, φ)
3:     while L ≠ φ do
4:         p ← L
5:         L ← NEXT(L)
6:         A ← LINK(p, A)
7:         EXCHANGE A ↔ B
8:     return (A, B)
```

The following example ANSI C program implements this splitting algorithm embedded.

```
struct Node* msort(struct Node* xs) {
    struct Node *p, *as, *bs;
    if (!xs || !xs→next) return xs;

    as = bs = NULL;
    while(xs) {
        p = xs;
        xs = xs→next;
        as = link(p, as);
        swap(as, bs);
    }
    as = msort(as);
    bs = msort(bs);
    return merge(as, bs);
}
```

The only thing left is to develop the imperative merging algorithm for linked-list. The idea is quite similar to the array merging version. As long as neither of the sub lists is exhausted, we pick the less one, and append it to the result list. After that, it just need link the non-empty one to the tail the result, but not a looping for copying. It needs some carefulness to initialize the result list, as its head node is the less one among the two sub lists. One simple method is to use a dummy sentinel head, and drop it before returning. This implementation detail can be given as the following.

```
struct Node* merge(struct Node* as, struct Node* bs) {
    struct Node s, *p;
    p = &s;
    while (as && bs) {
        if (as→key < bs→key) {
```

```
            link(p, as);
            as = as→next;
        }
        else {
            link(p, bs);
            bs = bs→next;
        }
        p = p→next;
    }
    if (as)
        link(p, as);
    if (bs)
        link(p, bs);
    return s.next;
}
```

Exercise 12.5

- Proof the performance of in-place merge sort is bound to $O(n \lg n)$.

12.10 Nature merge sort

Knuth gives another way to interpret the idea of divide and conquer merge sort. It just likes burn a candle in both ends [1]. This leads to the nature merge sort algorithm.

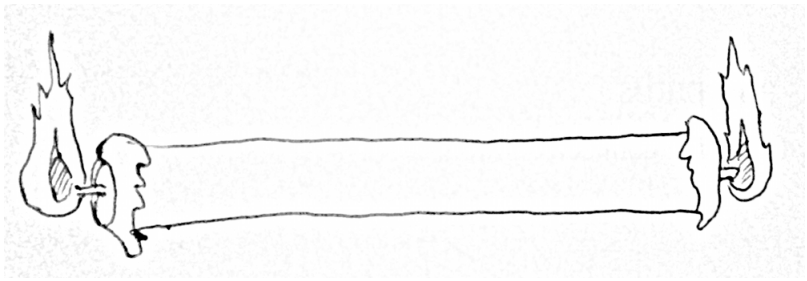


Figure 12.16: Burn a candle from both ends

For any given sequence, we can always find a non-decreasing sub sequence starts at any position. One particular case is that we can find such a sub sequence from the left-most position. The following table list some examples, the non-decreasing sub sequences are in bold font.

15 , 0, 4, 3, 5, 2, 7, 1, 12, 14, 13, 8, 9, 6, 10, 11
8, 12, 14 , 0, 1, 4, 11, 2, 3, 5, 9, 13, 10, 6, 15, 7
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

The first row in the table illustrates the worst case, that the second element is less than the first one, so the non-decreasing sub sequence is a singleton list, which only contains the first element; The last row shows the best case, the the sequence is ordered, and the non-decreasing list is the whole; The second row shows the average case.

www.dbooks.org

Symmetrically, we can always find a non-decreasing sub sequence from the end of the sequence to the left. This indicates us that we can merge the two non-decreasing sub sequences, one from the beginning, the other form the ending to a longer sorted sequence. The advantage of this idea is that, we utilize the nature ordered sub sequences, so that we needn't recursive sorting at all.

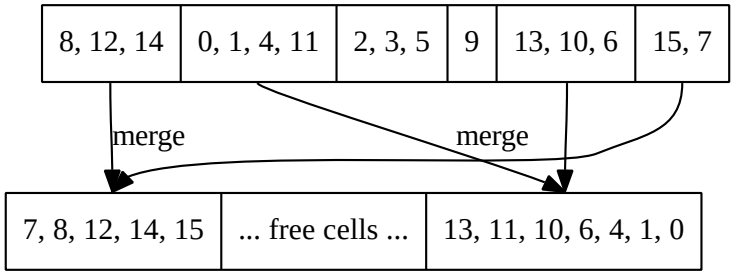


Figure 12.17: Nature merge sort

Figure 12.17 illustrates this idea. We starts the algorithm by scanning from both ends, finding the longest non-decreasing sub sequences respectively. After that, these two sub sequences are merged to the working area. The merged result starts from beginning. Next we repeat this step, which goes on scanning toward the center of the original sequence. This time we merge the two ordered sub sequences to the right hand of the working area toward the left. Such setup is easy for the next round of scanning. When all the elements in the original sequence have been scanned and merged to the target, we switch to use the elements stored in the working area for sorting, and use the previous sequence as new working area. Such switching happens repeatedly in each round. Finally, we copy all elements from the working area to the original array if necessary.

The only question left is when this algorithm stops. The answer is that when we start a new round of scanning, and find that the longest non-decreasing sub list spans to the end, which means the whole list is ordered, the sorting is done.

Because this kind of merge sort proceeds the target sequence in two ways, and uses the nature ordering of sub sequences, it's named *nature two-way merge sort*. In order to realize it, some carefulness must be paid. Figure 12.18 shows the invariant during the nature merge sort. At anytime, all elements before marker a and after marker d have been already scanned and merged. We are trying to span the non-decreasing sub sequence $[a, b)$ as long as possible, at the same time, we span the sub sequence from right to left to span $[c, d)$ as long as possible as well. The invariant for the working area is shown in the second row. All elements before f and after r have already been processed (Note that they may contain several ordered sub sequences). For the odd times (1, 3, 5, ...), we merge $[a, b)$ and $[c, d)$ from f toword right; while for the even times (2, 4, 6, ...), we merge the two sorted sub sequences after r toward left.

For imperative realization, the sequence is represented by array. Before sorting starts, we duplicate the array to create a working area. The pointers a, b are initialized to point the left most position, while c, d point to the right most position. Pointer f starts by pointing to the front of the working area, and r points to the rear position.

1: **function** SORT(A)

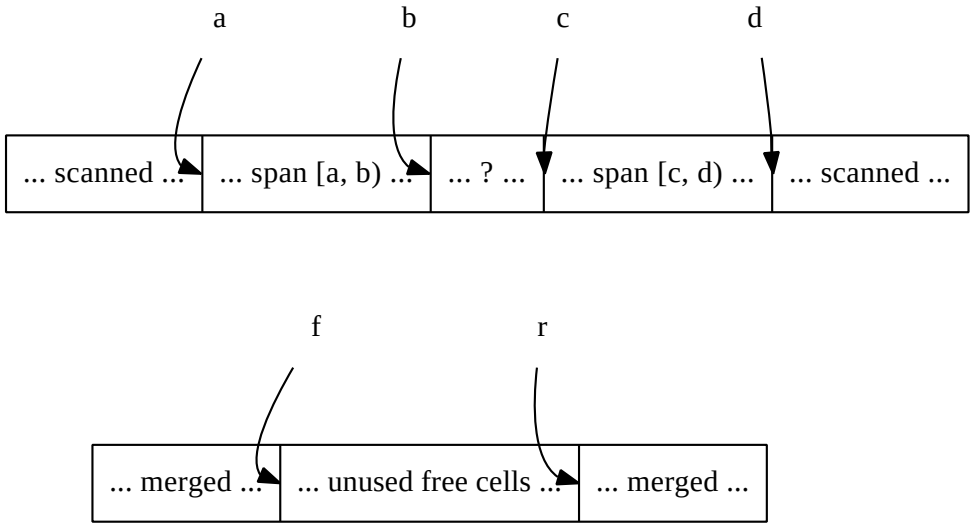


Figure 12.18: Invariant during nature merge sort

```
2:  if |A| > 1 then
3:      n ← |A|
4:      B ← CREATE-ARRAY(n)                          ▷ Create the working area
5:      loop
6:          [a, b) ← [1, 1)
7:          [c, d) ← [n + 1, n + 1)
8:          f ← 1, r ← n          ▷ front and rear pointers to the working area
9:          t ← False                                  ▷ merge to front or rear
10:         while b < c do                  ▷ There are still elements for scan
11:             repeat                                          ▷ Span [a, b)
12:                 b ← b + 1
13:             until b ≥ c ∨ A[b] < A[b − 1]
14:             repeat                                          ▷ Span [c, d)
15:                 c ← c − 1
16:             until c ≤ b ∨ A[c − 1] < A[c]
17:             if c < b then                                   ▷ Avoid overlap
18:                 c ← b
19:             if b − a ≥ n then     ▷ Done if [a, b) spans to the whole array
20:                 return A
21:             if t then                                       ▷ merge to front
22:                 f ← MERGE(A, [a, b), [c, d), B, f, 1)
23:             else                                            ▷ merge to rear
24:                 r ← MERGE(A, [a, b), [c, d), B, r, −1)
25:             a ← b, d ← c
26:             t ← ¬t                          ▷ Switch the merge direction
27:         EXCHANGE A ↔ B                                  ▷ Switch working area
28:     return A
```

The merge algorithm is almost as same as before except that we need pass

www.dbooks.org

a parameter to indicate the direction for merging.

```
1: function MERGE(A, [a, b), [c, d), B, w, Δ)
2:     while a < b ∧ c < d do
3:         if A[a] < A[d − 1] then
4:             B[w] ← A[a]
5:             a ← a + 1
6:         else
7:             B[w] ← A[d − 1]
8:             d ← d − 1
9:         w ← w + Δ
10:    while a < b do
11:        B[w] ← A[a]
12:        a ← a + 1
13:        w ← w + Δ
14:    while c < d do
15:        B[w] ← A[d − 1]
16:        d ← d − 1
17:        w ← w + Δ
18:    return w
```

The following ANSI C program implements this two-way nature merge sort algorithm. Note that it doesn't release the allocated working area explicitly.

```
int merge(Key* xs, int a, int b, int c, int d, Key* ys, int k, int delta) {
    for(; a < b && c < d; k += delta )
        ys[k] = xs[a] < xs[d-1] ? xs[a++] : xs[--d];
    for(; a < b; k += delta)
        ys[k] = xs[a++];
    for(; c < d; k += delta)
        ys[k] = xs[--d];
    return k;
}

Key* sort(Key* xs, Key* ys, int n) {
    int a, b, c, d, f, r, t;
    if(n < 2)
        return xs;
    for(;;) {
        a = b = 0;
        c = d = n;
        f = 0;
        r = n-1;
        t = 1;
        while(b < c) {
            do {      /* span [a, b) as much as possible */
                ++b;
            } while( b < c && xs[b-1] ≤ xs[b] );
            do{       /* span [c, d) as much as possible */
                --c;
            } while( b < c && xs[c] ≤ xs[c-1] );
            if( c < b )
                c = b;   /* eliminate overlap if any */
            if( b - a ≥ n)
```

```
            return xs;          /* sorted */
        if( t )
            f = merge(xs, a, b, c, d, ys, f, 1);
        else
            r = merge(xs, a, b, c, d, ys, r, -1);
        a = b;
        d = c;
        t = !t;
    }
    swap(&xs, &ys);
  }
  return xs; /*can't be here*/
}
```

The performance of nature merge sort depends on the actual ordering of the sub arrays. However, it in fact performs well even in the worst case. Suppose that we are unlucky when scanning the array, that the length of the non-decreasing sub arrays are always 1 during the first round scan. This leads to the result working area with merged ordered sub arrays of length 2. Suppose that we are unlucky again in the second round of scan, however, the previous results ensure that the non-decreasing sub arrays in this round are no shorter than 2, this time, the working area will be filled with merged ordered sub arrays of length 4, ... Repeat this we get the length of the non-decreasing sub arrays doubled in every round, so there are at most $O(\lg n)$ rounds, and in every round we scanned all the elements. The overall performance for this worst case is bound to $O(n \lg n)$. We'll go back to this interesting phenomena in the next section about bottom-up merge sort.

In purely functional settings however, it's not sensible to scan list from both ends since the underlying data structure is singly linked-list. The nature merge sort can be realized in another approach.

Observe that the list to be sorted is consist of several non-decreasing sub lists, that we can pick every two of such sub lists and merge them to a bigger one. We repeatedly pick and merge, so that the number of the non-decreasing sub lists halves continuously and finally there is only one such list, which is the sorted result. This idea can be formalized in the following equation.

$$sort(L) = sort'(group(L)) \tag{12.35}$$

Where function $group(L)$ groups the elements in the list into non-decreasing sub lists. This function can be described like below, the first two are trivial edge cases.

- If the list is empty, the result is a list contains an empty list;
- If there is only one element in the list, the result is a list contains a singleton list;
- Otherwise, The first two elements are compared, if the first one is less than or equal to the second, it is linked in front of the first sub list of the recursive grouping result; or a singleton list contains the first element is set as the first sub list before the recursive result.

www.dbooks.org

$$group(L) = \begin{cases} \{L\} & : & |L| \leq 1 \\ \{\{l_1\} \cup L_1, L_2, ...\} & : & l_1 \leq l_2, \{L_1, L_2, ...\} = group(L') \\ \{\{l_1\}, L_1, L_2, ...\} & : & otherwise \end{cases} \tag{12.36}$$

It's quite possible to abstract the grouping criteria as a parameter to develop a generic grouping function, for instance, as the following Haskell code [6].

```
groupBy' :: (a→a→Bool) →[a] →[[a]]
groupBy' _ [] = [[]]
groupBy' _ [x] = [[x]]
groupBy' f (x:xs@(x':_)) | f x x' = (x:ys):yss
                         | otherwise = [x]:r
  where
    r@(ys:yss) = groupBy' f xs
```

Different from the *sort* function, which sorts a list of elements, function $sort'$ accepts a list of sub lists which is the result of grouping.

$$sort'(\mathbb{L}) = \begin{cases} \phi & : & \mathbb{L} = \phi \\ L_1 & : & \mathbb{L} = \{L_1\} \\ sort'(mergePairs(\mathbb{L})) & : & otherwise \end{cases} \tag{12.37}$$

The first two are the trivial edge cases. If the list to be sorted is empty, the result is obviously empty; If it contains only one sub list, then we are done. We need just extract this single sub list as result; For the recursive case, we call a function $mergePairs$ to merge every two sub lists, then recursively call $sort'$.

The next undefined function is $mergePairs$, as the name indicates, it repeatedly merges pairs of non-decreasing sub lists into bigger ones.

$$mergePairs(L) = \begin{cases} L & : & |L| \leq 1 \\ \{merge(L_1, L_2)\} \cup mergePairs(L'') & : & otherwise \end{cases} \tag{12.38}$$

When there are less than two sub lists in the list, we are done; otherwise, we merge the first two sub lists L_1 and L_2, and recursively merge the rest of pairs in L''. The type of the result of $mergePairs$ is list of lists, however, it will be flattened by $sort'$ function finally.

The *merge* function is as same as before. The complete example Haskell program is given as below.

```
mergesort = sort' ∘ groupBy' (≤)

sort' [] = []
sort' [xs] = xs
sort' xss = sort' (mergePairs xss) where
  mergePairs (xs:ys:xss) = merge xs ys : mergePairs xss
  mergePairs xss = xss
```

[6] There is a 'groupBy' function provided in the Haskell standard library 'Data.List'. However, it doesn't fit here, because it accepts an equality testing function as parameter, which must satisfy the properties of reflexive, transitive, and symmetric. but what we use here, the less-than or equal to operation doesn't conform to symetric. Refer to appendix A of this book for detail.

Alternatively, observing that we can first pick two sub lists, merge them to an intermediate result, then repeatedly pick next sub list, and merge to this ordered result we've gotten so far until all the rest sub lists are merged. This is a typical folding algorithm as introduced in appendix A.

$$sort(L) = fold(merge, \phi, group(L)) \tag{12.39}$$

Translate this version to Haskell yields the folding version.

```
mergesort' = foldl merge [] ∘ groupBy' (≤)
```

Exercise 12.6

- Is the nature merge sort algorithm realized by folding is equivalent with the one by using $mergePairs$ in terms of performance? If yes, prove it; If not, which one is faster?

12.11 Bottom-up merge sort

The worst case analysis for nature merge sort raises an interesting topic, instead of realizing merge sort in top-down manner, we can develop a bottom-up version. The great advantage is that, we needn't do book keeping any more, so the algorithm is quite friendly for purely iterative implementation.

The idea of bottom-up merge sort is to turn the sequence to be sorted into n small sub sequences each contains only one element. Then we merge every two of such small sub sequences, so that we get $\frac{n}{2}$ ordered sub sequences each with length 2; If n is odd number, we left the last singleton sequence untouched. We repeatedly merge these pairs, and finally we get the sorted result. Knuth names this variant as 'straight two-way merge sort' [1]. The bottom-up merge sort is illustrated in figure 12.19

Different with the basic version and even-odd version, we needn't explicitly split the list to be sorted in every recursion. The whole list is split into n singletons at the very beginning, and we merge these sub lists in the rest of the algorithm.

$$sort(L) = sort'(wraps(L)) \tag{12.40}$$

$$wraps(L) = \begin{cases} \phi & : \quad L = \phi \\ \{\{l_1\}\} \cup wraps(L') & : \quad otherwise \end{cases} \tag{12.41}$$

Of course $wraps$ can be implemented by using mapping as introduced in appendix A.

$$sort(L) = sort'(map(\lambda_x \cdot \{x\}, L)) \tag{12.42}$$

We reuse the function $sort'$ and $mergePairs$ which are defined in section of nature merge sort. They repeatedly merge pairs of sub lists until there is only one.

Implement this version in Haskell gives the following example code.

www.dbooks.org

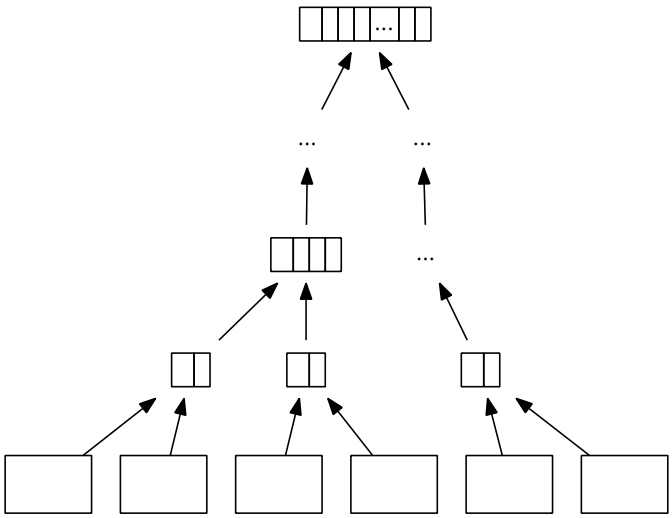


Figure 12.19: Bottom-up merge sort

```
sort = sort' ∘ map (λx→[x])
```

This version is based on what Okasaki presented in [6]. It is quite similar to the nature merge sort only differs in the way of grouping. Actually, it can be deduced as a special case (the worst case) of nature merge sort by the following equation.

$$sort(L) = sort'(groupBy(\lambda_{x,y} \cdot False, L)) \tag{12.43}$$

That instead of spanning the non-decreasing sub list as long as possible, the predicate always evaluates to false, so the sub list spans only one element.

Similar with nature merge sort, bottom-up merge sort can also be defined by folding. The detailed implementation is left as exercise to the reader.

Observing the bottom-up sort, we can find it's in tail-recursion call manner, thus it's quite easy to translate into purely iterative algorithm without any recursion.

```
1: function SORT(A)
2:     B ← φ
3:     for ∀a ∈ A do
4:         B ← APPEND({a})
5:     N ← |B|
6:     while N > 1 do
7:         for i ← from 1 to ⌊N/2⌋ do
8:             B[i] ← MERGE(B[2i − 1], B[2i])
9:         if ODD(N) then
10:            B[⌈N/2⌉] ← B[N]
11:        N ← ⌈N/2⌉
12:    if B = φ then
13:        return φ
```

14: **return** $B[1]$

The following example Python program implements the purely iterative bottom-up merge sort.

```
def mergesort(xs):
    ys = [[x] for x in xs]
    while len(ys) > 1:
        ys.append(merge(ys.pop(0), ys.pop(0)))
    return [] if ys == [] else ys.pop()

def merge(xs, ys):
    zs = []
    while xs != [] and ys !=[]:
        zs.append(xs.pop(0) if xs[0] < ys[0] else ys.pop(0))
    return zs + (xs if xs !=[] else ys)
```

The Python implementation combines multiple rounds of merging by consuming the pair of lists on the head, and appending the merged result to the tail. This greatly simply the logic of handling odd sub lists case as shown in the above pseudo code.

Exercise 12.7

- Implement the functional bottom-up merge sort by using folding.
- Implement the iterative bottom-up merge sort only with array indexing. Don't use any library supported tools, such as list, vector etc.

12.12 Parallelism

We mentioned in the basic version of quick sort, that the two sub sequences can be sorted in parallel after the divide phase finished. This strategy is also applicable for merge sort. Actually, the parallel version quick sort and morege sort, do not only distribute the recursive sub sequences sorting into two parallel processes, but divide the sequences into p sub sequences, where p is the number of processors. Idealy, if we can achieve sorting in T' time with parallelism, which satisifies $O(n \lg n) = pT'$. We say it is linear speed up, and the algorithm is parallel optimal.

However, a straightforward parallel extension to the sequential quick sort algorithm which samples several pivots, divides p sub sequences, and independently sorts them in parallel, isn't optimal. The bottleneck exists in the divide phase, which we can only achieve $O(n)$ time in average case.

The straightforward parallel extension to merge sort, on the other hand, block at the merge phase. Both parallel merge sort and quick sort in practice need good designs in order to achieve the optimal speed up. Actually, the divide and conquer nature makes merge sort and quick sort relative easy for parallelisim. Richard Cole found the $O(\lg n)$ parallel merge sort algorithm with n processors in 1986 in [13].

Parallelism is a big and complex topic which is out of the scope of this elementary book. Readers can refer to [13] and [14] for details.

www.dbooks.org

12.13 Short summary

In this chapter, two popular divide and conquer sorting methods, quick sort and merge sort are introduced. Both of them meet the upper performance limit of the comparison based sorting algorithms $O(n \lg n)$. Sedgewick said that quick sort is the greatest algorithm invented in the 20th century. Almost all programming environments adopt quick sort as the default sorting tool. As time goes on, some environments, especially those manipulate abstract sequence which is dynamic and not based on pure array switch to merge sort as the general purpose sorting tool[7].

The reason for this interesting phenomena can be partly explained by the treatment in this chapter. That quick sort performs perfectly in most cases, it needs fewer swapping than most other algorithms. However, the quick sort algorithm is based on swapping, in purely functional settings, swapping isn't the most efficient way due to the underlying data structure is singly linked-list, but not vectorized array. Merge sort, on the other hand, is friendly in such environment, as it costs constant spaces, and the performance can be ensured even in the worst case of quick sort, while the latter downgrade to quadratic time. However, merge sort doesn't performs as well as quick sort in purely imperative settings with arrays. It either needs extra spaces for merging, which is sometimes unreasonable, for example in embedded system with limited memory, or causes many overhead swaps by in-place workaround. In-place merging is till an active research area.

Although the title of this chapter is 'quick sort vs. merge sort', it's not the case that one algorithm has nothing to do with the other. Quick sort can be viewed as the optimized version of tree sort as explained in this chapter. Similarly, merge sort can also be deduced from tree sort as shown in [12].

There are many ways to categorize sorting algorithms, such as in [1]. One way is to from the point of view of easy/hard partition, and easy/hard merge [7].

Quick sort, for example, is quite easy for merging, because all the elements in the sub sequence before the pivot are no greater than any one after the pivot. The merging for quick sort is actually trivial sequence concatenation.

Merge sort, on the other hand, is more complex in merging than quick sort. However, it's quite easy to divide no matter what concrete divide method is taken: simple divide at the middle point, even-odd splitting, nature splitting, or bottom-up straight splitting. Compare to merge sort, it's more difficult for quick sort to achieve a perfect dividing. We show that in theory, the worst case can't be completely avoided, no matter what engineering practice is taken, median-of-three, random quick sort, 3-way partition etc.

We've shown some elementary sorting algorithms in this book till this chapter, including insertion sort, tree sort, selection sort, heap sort, quick sort and merge sort. Sorting is still a hot research area in computer science. At the time when this chapter is written, people are challenged by the buzz word 'big data', that the traditional convenient method can't handle more and more huge data within reasonable time and resources. Sorting a sequence of hundreds of Gigabytes becomes a routine in some fields.

[7] Actually, most of them are kind of hybrid sort, balanced with insertion sort to achieve good performance when the sequence is short

Exercise 12.8

- Design an algorithm to create binary search tree by using merge sort strategy.

www.dbooks.org

Bibliography

[1] Donald E. Knuth. “The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)”. Addison-Wesley Professional; 2 edition (May 4, 1998) ISBN-10: 0201896850 ISBN-13: 978-0201896855

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. ISBN:0262032937. The MIT Press. 2001

[3] Robert Sedgewick. “Implementing quick sort programs”. Communication of ACM. Volume 21, Number 10. 1978. pp.847 - 857.

[4] Jon Bentley. “Programming pearls, Second Edition”. Addison-Wesley Professional; 1999. ISBN-13: 978-0201657883

[5] Jon Bentley, Douglas McIlroy. “Engineering a sort function”. Software Practice and experience VOL. 23(11), 1249-1265 1993.

[6] Robert Sedgewick, Jon Bentley. “Quicksort is optimal”. http://www.cs.princeton.edu/ rs/talks/QuicksortIsOptimal.pdf

[7] Richard Bird. “Pearls of functional algorithm design”. Cambridge University Press. 2010. ISBN, 1139490605, 9781139490603

[8] Fethi Rabhi, Guy Lapalme. “Algorithms: a functional programming approach”. Second edition. Addison-Wesley, 1999. ISBN: 0201-59604-0

[9] Simon Peyton Jones. “The Implementation of functional programming languages”. Prentice-Hall International, 1987. ISBN: 0-13-453333-X

[10] Jyrki Katajainen, Tomi Pasanen, Jukka Teuhola. “Practical in-place mergesort”. Nordic Journal of Computing, 1996.

[11] Chris Okasaki. “Purely Functional Data Structures”. Cambridge university press, (July 1, 1999), ISBN-13: 978-0521663502

[12] Josè Bacelar Almeida and Jorge Sousa Pinto. “Deriving Sorting Algorithms”. Technical report, Data structures and Algorithms. 2008.

[13] Cole, Richard (August 1988). “Parallel merge sort”. SIAM J. Comput. 17 (4): 770¨C785. doi:10.1137/0217049. (August 1988)

[14] Powers, David M. W. “Parallelized Quicksort and Radixsort with Optimal Speedup”, Proceedings of International Conference on Parallel Computing Technologies. Novosibirsk. 1991.

www.dbooks.org

[15] Wikipedia. “Quicksort”. http://en.wikipedia.org/wiki/Quicksort

[16] Wikipedia. “Strict weak order”. http://en.wikipedia.org/wiki/Strict_weak_order

[17] Wikipedia. “Total order”. http://en.wokipedia.org/wiki/Total_order

[18] Wikipedia. “Harmonic series (mathematics)”. http://en.wikipedia.org/wiki/Harmonic_series_(mathematics)

Chapter 13

Searching

13.1 Introduction

Searching is quite a big and important area. Computer makes many hard searching problems realistic. They are almost impossible for human beings. A modern industry robot can even search and pick the correct gadget from the pipeline for assembly; A GPS car navigator can search among the map, for the best route to a specific place. The modern mobile phone is not only equipped with such map navigator, but it can also search for the best price for Internet shopping.

This chapter just scratches the surface of elementary searching. One good thing that computer offers is the brute-force scanning for a certain result in a large sequence. The divide and conquer search strategy will be briefed with two problems, one is to find the k-th big one among a list of unsorted elements; the other is the popular binary search among a list of sorted elements. We'll also introduce the extension of binary search for multiple-dimension data.

Text matching is also very important in our daily life, two well-known searching algorithms, Knuth-Morris-Pratt (KMP) and Boyer-Moore algorithms will be introduced. They set good examples for another searching strategy: information reusing.

Besides sequence search, some elementary methods for searching solution for some interesting problems will be introduced. They were mostly well studied in the early phase of AI (artificial intelligence), including the basic DFS (Depth first search), and BFS (Breadth first search).

Finally, Dynamic programming will be briefed for searching optimal solutions, and we'll also introduce about greedy algorithm which is applicable for some special cases.

All algorithms will be realized in both imperative and functional approaches.

13.2 Sequence search

Although modern computer offers fast speed for brute-force searching, and even if the Moore's law could be strictly followed, the grows of huge data is too fast to be handled well in this way. We've seen a vivid example in the introduction chapter of this book. It's why people study the computer search algorithms.

www.dbooks.org

13.2.1 Divide and conquer search

One solution is to use divide and conquer approach. That if we can repeatedly scale down the search domain, the data being dropped needn't be examined at all. This will definitely speed up the search.

k-selection problem

Consider a problem of finding the k-th smallest one among n elements. The most straightforward idea is to find the minimum first, then drop it and find the second minimum element among the rest. Repeat this minimum finding and dropping k steps will give the k-th smallest one. Finding the minimum among n elements costs linear $O(n)$ time. Thus this method performs $O(kn)$ time, if k is much smaller than n.

Another method is to use the 'heap' data structure we've introduced. No matter what concrete heap is used, e.g. binary heap with implicit array, Fibonacci heap or others, Accessing the top element followed by popping is typically bound $O(\lg n)$ time. Thus this method, as formalized in equation (13.1) and (13.2) performs in $O(k \lg n)$ time, if k is much smaller than n.

$$top(k, L) = find(k, heapify(L)) \tag{13.1}$$

$$find(k, H) = \begin{cases} top(H) & : \quad k = 0 \\ find(k-1, pop(H)) & : \quad otherwise \end{cases} \tag{13.2}$$

However, heap adds some complexity to the solution. Is there any simple, fast method to find the k-th element?

The divide and conquer strategy can help us. If we can divide all the elements into two sub lists A and B, and ensure all the elements in A is not greater than any elements in B, we can scale down the problem by following this method[1]:

1. Compare the length of sub list A and k;
2. If $k < |A|$, the k-th smallest one must be contained in A, we can drop B and *further search* in A;
3. If $|A| < k$, the k-th smallest one must be contained in B, we can drop A and *further search* the $(k - |A|)$-th smallest one in B.

Note that the *italic font* emphasizes the fact of recursion. The ideal case always divides the list into two equally big sub lists A and B, so that we can halve the problem each time. Such ideal case leads to a performance of $O(n)$ linear time.

Thus the key problem is how to realize dividing, which collects the first m smallest elements in one sub list, and put the rest in another.

This reminds us the partition algorithm in quick sort, which moves all the elements smaller than the pivot in front of it, and moves those greater than the pivot behind it. Based on this idea, we can develop a divide and conquer k-selection algorithm, which is called quick selection algorithm.

[1]This actually demands a more accurate definition of the k-th smallest in L: It's equal to the k-the element of L', where L' is a permutation of L, and L' is in monotonic non-decreasing order.

1. Randomly select an element (the first for instance) as the pivot;
2. Moves all elements which aren't greater than the pivot in a sub list A; and moves the rest to sub list B;
3. Compare the length of A with k, if $|A| = k - 1$, then the pivot is the k-th smallest one;
4. If $|A| > k - 1$, recursively find the k-th smallest one among A;
5. Otherwise, recursively find the $(k - 1 - |A|)$-th smallest one among B;

This algorithm can be formalized in below equation. Suppose $0 < k \leq |L|$, where L is a non-empty list of elements. Denote l_1 as the first element in L. It is chosen as the pivot; L' contains the rest elements except for l_1. $(A, B) = partition(\lambda_x \cdot x \leq l_1, L')$. It partitions L' by using the same algorithm defined in the chapter of quick sort.

$$top(k, L) = \begin{cases} l_1 & : & |A| = k - 1 \\ top(k - 1 - |A|, B) & : & |A| < k - 1 \\ top(k, A) & : & otherwise \end{cases} \tag{13.3}$$

$$partition(p, L) = \begin{cases} (\phi, \phi) & : & L = \phi \\ (\{l_1\} \cup A, B) & : & p(l_1), (A, B) = partition(p, L') \\ (A, \{l_1\} \cup B) & : & \neg p(l_1) \end{cases} \tag{13.4}$$

The following Haskell example program implements this algorithm.

```
top n (x:xs) | len == n - 1 = x
             | len < n - 1 = top (n - len - 1) bs
             | otherwise = top n as
    where
      (as, bs) = partition (≤x) xs
      len = length as
```

The partition function is provided in Haskell standard library, the detailed implementation can be referred to previous chapter about quick sort.

The lucky case is that, the k-th smallest element is selected as the pivot at the very beginning. The partition function examines the whole list, and finds that there are $k - 1$ elements not greater than the pivot, we are done in just $O(n)$ time. The worst case is that either the maximum or the minimum element is selected as the pivot every time. The partition always produces an empty sub list, that either A or B is empty. If we always pick the minimum as the pivot, the performance is bound to $O(kn)$. If we always pick the maximum as the pivot, the performance is $O((n - k)n)$.

The best case (not the lucky case), is that the pivot always partition the list perfectly. The length of A is nearly as same as the length of B. The list is halved every time. It needs about $O(\lg n)$ partitions, each partition takes linear time proportion to the length of the halved list. This can be expressed as $O(n + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{2^m})$, where m is the smallest number satisfies $\frac{n}{2^m} < k$. Summing the series leads to the result of $O(n)$.

www.dbooks.org

The average case analysis needs tool of mathematical expectation. It's quite similar to the proof given in previous chapter of quick sort. It's left as an exercise to the reader.

Similar as quick sort, this divide and conquer selection algorithm performs well most time in practice. We can take the same engineering practice such as media-of-three, or randomly select the pivot as we did for quick sort. Below is the imperative realization for example.

```
1: function TOP(k, A, l, u)
2:     EXCHANGE A[l] ↔ A[ RANDOM(l, u) ]          ▷ Randomly select in [l, u]
3:     p ← PARTITION(A, l, u)
4:     if p − l + 1 = k then
5:         return A[p]
6:     if k < p − l + 1 then
7:         return TOP(k, A, l, p − 1)
8:     return TOP(k − p + l − 1, A, p + 1, u)
```

This algorithm searches the k-th smallest element in range of $[l, u]$ for array A. The boundaries are included. It first randomly selects a position, and swaps it with the first one. Then this element is chosen as the pivot for partitioning. The partition algorithm in-place moves elements and returns the position where the pivot being moved. If the pivot is just located at position k, then we are done; if there are more than $k - 1$ elements not greater than the pivot, the algorithm recursively searches the k-th smallest one in range $[l, p-1]$; otherwise, k is deduced by the number of elements before the pivot, and recursively searches the range after the pivot $[p + 1, u]$.

There are many methods to realize the partition algorithm, below one is based on N. Lumoto's method. Other realizations are left as exercises to the reader.

```
1: function PARTITION(A, l, u)
2:     p ← A[l]
3:     L ← l
4:     for R ← l + 1 to u do
5:         if ¬(p < A[R]) then
6:             L ← L + 1
7:             EXCHANGE A[L] ↔ A[R]
8:     EXCHANGE A[L] ↔ p
9:     return L
```

Below ANSI C example program implements this algorithm. Note that it handles the special case that either the array is empty, or k is out of the boundaries of the array. It returns -1 to indicate the search failure.

```
int partition(Key* xs, int l, int u) {
    int r, p = l;
    for (r = l + 1; r < u; ++r)
        if (!(xs[p] < xs[r]))
            swap(xs, ++l, r);
    swap(xs, p, l);
    return l;
}
```

```
/* The result is stored in xs[k], returns k if u-l ≥k, otherwise -1 */
int top(int k, Key* xs, int l, int u) {
    int p;
    if (l < u) {
        swap(xs, l, rand() % (u - l) + l);
        p = partition(xs, l, u);
        if (p - l + 1 == k)
            return p;
        return (k < p - l + 1) ? top(k, xs, l, p) :
                                 top(k- p + l - 1, xs, p + 1, u);
    }
    return -1;
}
```

There is a method proposed by Blum, Floyd, Pratt, Rivest and Tarjan in 1973, which ensures the worst case performance being bound to $O(n)$ [2], [3]. It divides the list into small groups. Each group contains no more than 5 elements. The median of each group among these 5 elements are identified quickly. Then there are $\frac{n}{5}$ median elements selected. We repeat this step, and divide them again into groups of 5, and recursively select the *median of median*. It's obviously that the final 'true' median can be found in $O(\lg n)$ time. This is the best pivot for partitioning the list. Next, we halve the list by this pivot and recursively search for the k-th smallest one. The performance can be calculated as the following.

$$T(n) = c_1 lgn + c_2 n + T(\frac{n}{2}) \tag{13.5}$$

Where c_1 and c_2 are constant factors for the median of median and partition computation respectively. Solving this equation with telescope method or the master theory in [2] gives the linear $O(n)$ performance.

In case we just want to pick the top k smallest elements, but don't care about the order of them, the algorithm can be adjusted a little bit to fit.

$$tops(k, L) = \begin{cases} \phi & : & k = 0 \vee L = \phi \\ A & : & |A| = k \\ A \cup \{l_1\} \cup tops(k - |A| - 1, B) & : & |A| < k \\ tops(k, A) & : & otherwise \end{cases} \tag{13.6}$$

Where A, B have the same meaning as before that, $(A, B) = partition(\lambda_x \cdot x \leq l_1, L')$ if L isn't empty. The relative example program in Haskell is given as below.

```
tops _ [] = []
tops 0 _  = []
tops n (x:xs) | len == n = as
              | len < n  = as ++ [x] ++ tops (n-len-1) bs
              | otherwise = tops n as
    where
      (as, bs) = partition (≤ x) xs
      len = length as
```

www.dbooks.org

binary search

Another popular divide and conquer algorithm is binary search. We've shown it in the chapter about insertion sort. When I was in school, the teacher who taught math played a magic to me, He asked me to consider a natural number less than 1000. Then he asked me some questions, I only replied 'yes' or 'no', and finally he guessed my number. He typically asked questions like the following:

- Is it an even number?
- Is it a prime number?
- Are all digits same?
- Can it be divided by 3?
- ...

Most of the time he guessed the number within 10 questions. My classmates and I all thought it's unbelievable.

This game will not be so interesting if it downgrades to a popular TV program, that the price of a product is hidden, and you must figure out the exact price in 30 seconds. The host of the program tells you if your guess is higher or lower to the fact. If you win, the product is yours. The best strategy is to use similar divide and conquer approach to perform a binary search. So it's common to find such conversation between the player and the host:

- P: 1000;
- H: High;
- P: 500;
- H: Low;
- P: 750;
- H: Low;
- P: 890;
- H: Low;
- P: 990;
- H: Bingo.

My math teacher told us that, because the number we considered is within 1000, if he can halve the numbers every time by designing good questions, the number will be found in 10 questions. This is because $2^{10} = 1024 > 1000$. However, it would be boring to just ask it is higher than 500, is lower than 250, ... Actually, the question 'is it even' is very good, because it always halve the numbers[2].

[2]When the author revise this chapter, Microsoft released a game in social networks. User can consider a person's name, the AI robot asks 16 questions next. The user only answers with yes or no. The robot will tell you who is that person. Can you figure out how the robot works?

Come back to the binary search algorithm. It is only applicable to a sequence of ordered number. I've seen programmers tried to apply it to unsorted array, and took several hours to figure out why it doesn't work. The idea is quite straightforward, in order to find a number x in an ordered sequence A, we firstly check middle point number, compare it with x, if they are same, then we are done; If x is smaller, as A is ordered, we need only recursively search it among the first half; otherwise we search it among the second half. Once A gets empty and we haven't found x yet, it means x doesn't exist.

Before formalizing this algorithm, there is a surprising fact need to be noted. Donald Knuth stated that 'Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky¡'. Jon Bentley pointed out that most binary search implementation contains errors, and even the one given by him in the first version of 'Programming pearls' contains an error undetected over twenty years [4].

There are two kinds of realization, one is recursive, the other is iterative. The recursive solution is as same as what we described. Suppose the lower and upper boundaries of the array are l and u inclusive.

```
1: function BINARY-SEARCH(x, A, l, u)
2:     if u < l then
3:         Not found error
4:     else
5:         m ← l + ⌊(u − l)/2⌋                    ▷ avoid overflow of ⌊(l + u)/2⌋
6:         if A[m] = x then
7:             return m
8:         if x < A[m] then
9:             return BINARY-SEARCH(x, A, l, m - 1)
10:        else
11:            return BINARY-SEARCH(x, A, m + 1, u)
```

As the comment highlights, if the integer is represented with limited words, we can't merely use $\lfloor \frac{l+u}{2} \rfloor$ because it may cause overflow if l and u are big.

Binary search can also be realized in iterative manner, that we keep updating the boundaries according to the middle point comparison result.

```
1: function BINARY-SEARCH(x, A, l, u)
2:     while l < u do
3:         m ← l + ⌊(u − l)/2⌋
4:         if A[m] = x then
5:             return m
6:         if x < A[m] then
7:             u ← m − 1
8:         else
9:             l ← m + 1
       return NIL
```

The implementation is very good exercise, we left it to the reader. Please try all kinds of methods to verify your program.

Since the array is halved every time, the performance of binary search is bound to $O(\lg n)$ time.

In purely functional settings, the list is represented with singly linked-list. It's linear time to randomly access the element for a given position. Binary search doesn't make sense in such case. However, it good to analyze what the

www.dbooks.org

performance will downgrade to. Consider the following equation.

$$bsearch(x, L) = \begin{cases} Err & : & L = \phi \\ b_1 & : & x = b_1, (A, B) = splitAt(\lfloor \frac{|L|}{2} \rfloor, L) \\ bsearch(x, A) & : & B = \phi \vee x < b_1 \\ bsearch(x, B') & : & otherwise \end{cases}$$

Where b_1 is the first element if B isn't empty, and B' holds the rest except for b_1. The *splitAt* function takes $O(n)$ time to divide the list into two subs A and B (see the appendix A, and the chapter about merge sort for detail). If B isn't empty and x is equal to b_1, the search returns; Otherwise if it is less than b_1, as the list is sorted, we need recursively search in A, otherwise, we search in B. If the list is empty, we raise error to indicate search failure.

As we always split the list in the middle point, the number of elements halves in each recursion. In every recursive call, we takes linear time for splitting. The splitting function only traverses the first half of the linked-list, Thus the total time can be expressed as.

$$T(n) = c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} + ...$$

This results $O(n)$ time, which is as same as the brute force search from head to tail:

$$search(x, L) = \begin{cases} Err & : & L = \phi \\ l_1 & : & x = l_1 \\ search(x, L') & : & otherwise \end{cases}$$

As we mentioned in the chapter about insertion sort, the functional approach of binary search is through binary search tree. That the ordered sequence is represented in a tree (self balanced tree if necessary), which offers logarithm time searching [3].

Although it doesn't make sense to apply divide and conquer binary search on linked-list, binary search can still be very useful in purely functional settings. Consider solving an equation $a^x = y$, for given natural numbers a and y, where $a \leq y$. We want to find the integer solution for x if there is. Of course brute-force naive searching can solve it. We can examine all numbers one by one from 0 for $a^0, a^1, a^2, ...$, stops if $a^i = y$ or report that there is no solution if $a^i < y < a^{i+1}$ for some i. We initialize the solution domain as $X = \{0, 1, 2, ...\}$, and call the below exhausted searching function $solve(a, y, X)$.

$$solve(a, y, X) = \begin{cases} x_1 & : & a^{x_1} = y \\ solve(a, y, X') & : & a^{x_1} < y \\ Err & : & otherwise \end{cases}$$

This function examines the solution domain in monotonic increasing order. It takes the first candidate element x_1 from X, compare a^{x_1} and y, if they are equal, then x_1 is the solution and we are done; if it is less than y, then x_1 is dropped, and we search among the rest elements represented as X'; Otherwise,

[3] Some readers may argue that array should be used instead of linked-list, for example in Haskell. This book only deals with purely functional sequences in finger-tree. Different from the Haskell array, it can't support constant time random accessing

since $f(x) = a^x$ is non-decreasing function when a is natural number, so the rest elements will only make $f(x)$ bigger and bigger. There is no integer solution for this equation. The function returns error to indicate no solution.

The computation of a^x is expensive for big a and x if precession must be kept[4]. Can it be improved so that we can compute as less as possible? The divide and conquer binary search can help. Actually, we can estimate the upper limit of the solution domain. As $a^y \leq y$, We can search in range $\{0, 1, ..., y\}$. As the function $f(x) = a^x$ is non-decreasing against its argument x, we can firstly check the middle point candidate $x_m = \lfloor \frac{0+y}{2} \rfloor$, if $a^{x_m} = y$, the solution is found; if it is less than y, we can drop all candidate solutions before x_m; otherwise we drop all candidate solutions after it; Both halve the solution domain. We repeat this approach until either the solution is found or the solution domain becomes empty, which indicates there is no integer solution.

The binary search method can be formalized as the following equation. The non-decreasing function is abstracted as a parameter. To solve our problem, we can just call it as $bsearch(f, y, 0, y)$, where $f(x) = a^x$.

$$bsearch(f, y, l, u) = \begin{cases} Err & : & u < l \\ m & : & f(m) = y, m = \lfloor \frac{l+u}{2} \rfloor \\ bsearch(f, y, l, m-1) & : & f(m) > y \\ bsearch(f, y, m+1, u) & : & f(m) < y \end{cases} \tag{13.7}$$

As we halve the solution domain in every recursion, this method computes $f(x)$ in $O(\log y)$ times. It is much faster than the brute-force searching.

2 dimensions search

It's quite natural to think that the idea of binary search can be extended to 2 dimensions or even more general – multiple-dimensions domain. However, it is not so easy.

Consider the example of a $m \times n$ matrix M. The elements in each row and each column are in strict increasing order. Figure 13.1 illustrates such a matrix for example.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & ... \\ 2 & 4 & 5 & 6 & ... \\ 3 & 5 & 7 & 8 & ... \\ 4 & 6 & 8 & 9 & ... \\ ... & & & & \end{bmatrix}$$

Figure 13.1: A matrix in strict increasing order for each row and column.

Given a value x, how to locate all elements equal to x in the matrix quickly? We need develop an algorithm, which returns a list of locations (i, j) so that $M_{i,j} = x$.

[4]One alternative is to reuse the result of a^n when compute $a^{n+1} = aa^n$. Here we consider for general form monotonic function $f(n)$

www.dbooks.org

Richard Bird in [1] mentioned that he used this problem to interview candidates for entry to Oxford. The interesting story was that, those who had some computer background at school tended to use binary search. But it's easy to get stuck.

The usual way follows binary search idea is to examine element at $M_{\frac{m}{2},\frac{n}{2}}$. If it is less than x, we can only drop the elements in the top-left area; If it is greater than x, only the bottom-right area can be dropped. Both cases are illustrated in figure 13.2, the gray areas indicate elements can be dropped.

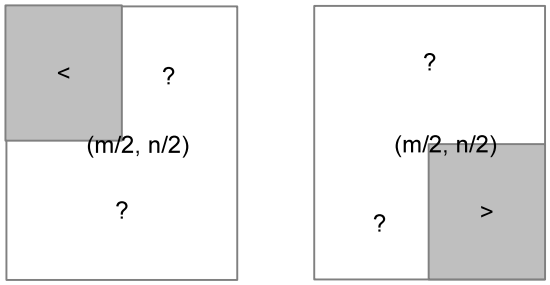


Figure 13.2: Left: the middle point element is smaller than x. All elements in the gray area are less than x; Right: the middle point element is greater than x. All elements in the gray area are greater than x.

The problem is that the solution domain changes from a rectangle to a 'L' shape in both cases. We can't just recursively apply search on it. In order to solve this problem systematically, we define the problem more generally, using brute-force search as a start point, and keep improving it bit by bit.

Consider a function $f(x, y)$, which is strict increasing for its arguments, for instance $f(x, y) = a^x + b^y$, where a and b are natural numbers. Given a value z, which is a natural number too, we want to solve the equation $f(x, y) = z$ by finding all none negative integral candidate pairs (x, y).

With this definition, the matrix search problem can be specialized by below function.

$$f(x, y) = \begin{cases} M_{x,y} & : \quad 1 \leq x \leq m, 1 \leq y \leq n \\ -1 & : \quad otherwise \end{cases}$$

Brute-force 2D search

As all solutions should be found for $f(x, y)$. One can immediately give the brute force solution by embedded looping.

```
1: function SOLVE(f, z)
2:     A ← φ
3:     for x ∈ {0, 1, 2, ..., z} do
4:         for y ∈ {0, 1, 2, ..., z} do
5:             if f(x, y) = z then
6:                 A ← A ∪ {(x, y)}
```

7: **return** A

This definitely calculates f for $(z+1)^2$ times. It can be formalized as in (13.8).

$$solve(f, z) = \{(x, y) | x \in \{0, 1, ..., z\}, y \in \{0, 1, ..., z\}, f(x, y) = z\} \quad (13.8)$$

Saddleback search

We haven't utilize the fact that $f(x, y)$ is strict increasing yet. Dijkstra pointed out in [6], instead of searching from bottom-left corner, starting from the top-left leads to one effective solution. As illustrated in figure 13.3, the search starts from $(0, z)$, for every point (p, q), we compare $f(p, q)$ with z:

- If $f(p, q) < z$, since f is strict increasing, for all $0 \leq y < q$, we have $f(p, y) < z$. We can drop all points in the vertical line section (in red color);
- If $f(p, q) > z$, then $f(x, q) > z$ for all $p < x \leq z$. We can drop all points in the horizontal line section (in blue color);
- Otherwise if $f(p, q) = z$, we mark (p, q) as one solution, then both line sections can be dropped.

This is a systematical way to scale down the solution domain rectangle. We keep dropping a row, or a column, or both.

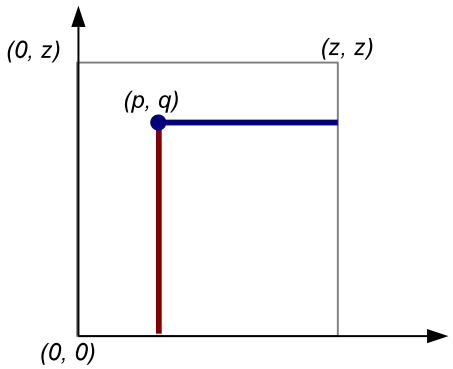


Figure 13.3: Search from top-left.

This method can be formalized as a function $search(f, z, p, q)$, which searches solutions for equation $f(x, y) = z$ in rectangle with top-left corner (p, q), and bottom-right corner $(z, 0)$. We start the searching by initializing $(p, q) = (0, z)$

www.dbooks.org

as $solve(f, z) = search(f, z, 0, z)$

$$search(f, z, p, q) = \begin{cases} \phi & : & p > z \vee q < 0 \\ search(f, z, p + 1, q) & : & f(p, q) < z \\ search(f, z, p, q - 1) & : & f(p, q) > z \\ \{(p, q)\} \cup search(f, z, p + 1, q - 1) & : & otherwise \end{cases} \tag{13.9}$$

The first clause is the edge case, there is no solution if (p, q) isn't top-left to $(z, 0)$. The following example Haskell program implements this algorithm.

```
solve f z = search 0 z where
  search p q | p > z || q < 0 = []
             | z' < z = search (p + 1) q
             | z' > z = search p (q - 1)
             | otherwise = (p, q) : search (p + 1) (q - 1)
    where z' = f p q
```

Considering the calculation of f may be expensive, this program stores the result of $f(p, q)$ to variable z'. This algorithm can also be implemented in iterative manner, that the boundaries of solution domain keeps being updated in a loop.

1: **function** SOLVE(f, z)
2: $p \leftarrow 0, q \leftarrow z$
3: $S \leftarrow \phi$
4: **while** $p \leq z \wedge q \geq 0$ **do**
5: $z' \leftarrow f(p, q)$
6: **if** $z' < z$ **then**
7: $p \leftarrow p + 1$
8: **else if** $z' > z$ **then**
9: $q \leftarrow q - 1$
10: **else**
11: $S \leftarrow S \cup \{(p, q)\}$
12: $p \leftarrow p + 1, q \leftarrow q - 1$
13: **return** S

It's intuitive to translate this imperative algorithm to real program, as the following example Python code.

```
def solve(f, z):
    (p, q) = (0, z)
    res = []
    while p <= z and q >= 0:
        z1 = f(p, q)
        if z1 < z:
            p = p + 1
        elif z1 > z:
            q = q - 1
        else:
            res.append((p, q))
            (p, q) = (p + 1, q - 1)
    return res
```

It is clear that in every iteration, At least one of p and q advances to the bottom-right corner by one. Thus it takes at most $2(z + 1)$ steps to complete

searching. This is the worst case. There are three best cases. The first one happens that in every iteration, both p and q advance by one, so that it needs only $z+1$ steps; The second case keeps advancing horizontally to right and ends when p exceeds z; The last case is similar, that it keeps moving down vertically to the bottom until q becomes negative.

Figure 13.4 illustrates the best cases and the worst cases respectively. Figure 13.4 (a) is the case that every point $(x, z-x)$ in diagonal satisfies $f(x, z-x) = z$, it uses $z+1$ steps to arrive at $(z, 0)$; (b) is the case that every point (x, z) along the top horizontal line gives the result $f(x, z) < z$, the algorithm takes $z+1$ steps to finish; (c) is the case that every point $(0, x)$ along the left vertical line gives the result $f(0, x) > z$, thus the algorithm takes $z+1$ steps to finish; (d) is the worst case. If we project all the horizontal sections along the search path to x axis, and all the vertical sections to y axis, it gives the total steps of $2(z+1)$.

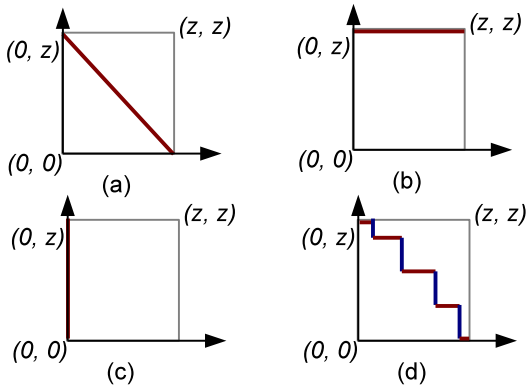


Figure 13.4: The best cases and the worst cases.

Compare to the quadratic brute-force method ($O(z^2)$), we improve to a linear algorithm bound to $O(z)$.

Bird imagined that the name 'saddleback' is because the 3D plot of f with the smallest bottom-left and the largest top-right and two wings looks like a saddle as shown in figure 13.5

Improved saddleback search

We haven't utilized the binary search tool so far, even the problem extends to 2-dimension domain. The basic saddleback search starts from the top-left corner $(0, z)$ to the bottom-right corner $(z, 0)$. This is actually over-general domain. we can constraint it a bit more accurate.

Since f is strict increasing, we can find the biggest number m, that $0 \leq m \leq z$, along the y axis which satisfies $f(0, m) \leq z$; Similarly, we can find the biggest n, that $0 \leq n \leq z$, along the x axis, which satisfies $f(n, 0) \leq z$; And the solution domain shrinks from $(0, z) - (z, 0)$ to $(0, m) - (n, 0)$ as shown in figure 13.6.

Of course m and n can be found by brute-force like below.

$$\begin{aligned} m &= max(\{y | 0 \leq y \leq z, f(0, y) \leq z\}) \\ n &= max(\{x | 0 \leq x \leq z, f(x, 0) \leq z\}) \end{aligned} \tag{13.10}$$

www.dbooks.org

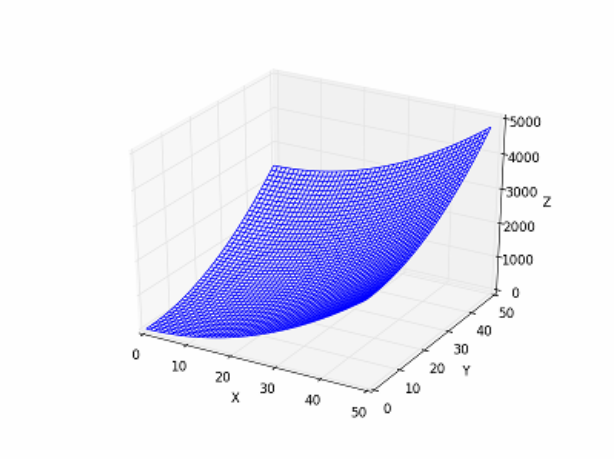


Figure 13.5: Plot of $f(x, y) = x^2 + y^2$.

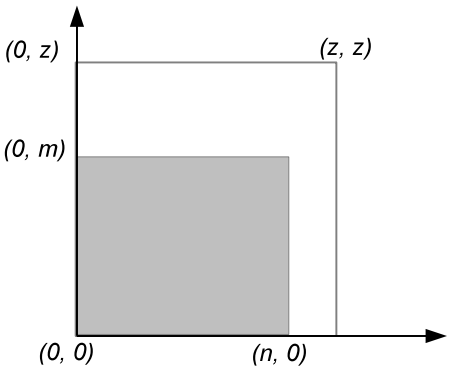


Figure 13.6: A more accurate search domain shown in gray color.

When searching m, the x variable of f is bound to 0. It turns to be one dimension search problem for a strict increasing function (or in functional term, a Curried function $f(0, y)$). Binary search works in such case. However, we need a bit modification for equation (13.7). Different from searching a solution $l \leq x \leq u$, so that $f(x) = y$ for a given y; we need search for a solution $l \leq x \leq u$ so that $f(x) \leq y < f(x+1)$.

$$bsearch(f, y, l, u) = \begin{cases} l & : & u \leq l \\ m & : & f(m) \leq y < f(m+1), m = \lfloor \frac{l+u}{2} \rfloor \\ bsearch(f, y, m+1, u) & : & f(m) \leq y \\ bsearch(f, y, l, m-1) & : & otherwise \end{cases} \tag{13.11}$$

The first clause handles the edge case of empty range. The lower boundary is returned in such case; If the middle point produces a value less than or equal to the target, while the next one evaluates to a bigger value, then the middle point is what we are looking for; Otherwise if the point next to the middle also evaluates to a value not greater than the target, the lower bound is set as the middle point plus one, and we perform recursively binary search; In the last case, the middle point evaluates to a value greater than the target, upper bound is updated as the point proceeds to the middle for further recursive searching. The following Haskell example code implements this modified binary search.

```
bsearch f y (l, u) | u ≤ l = l
                   | f m ≤ y = if f (m + 1) ≤ y
                               then bsearch f y (m + 1, u) else m
                   | otherwise = bsearch f y (l, m-1)
  where m = (l + u) `div` 2
```

Then m and n can be found with this binary search function.

$$\begin{aligned} m &= bsearch(\lambda_y \cdot f(0, y), z, 0, z) \\ n &= bsearch(\lambda_x \cdot f(x, 0), z, 0, z) \end{aligned} \tag{13.12}$$

And the improved saddleback search shrinks to this new search domain $solve(f, z) = search(f, z, 0, m)$:

$$search(f, z, p, q) = \begin{cases} \phi & : & p > n \vee q < 0 \\ search(f, z, p+1, q) & : & f(p, q) < z \\ search(f, z, p, q-1) & : & f(p, q) > z \\ \{(p, q)\} \cup search(f, z, p+1, q-1) & : & otherwise \end{cases} \tag{13.13}$$

It's almost as same as the basic saddleback version, except that it stops if p exceeds n, but not z. In real implementation, the result of $f(p, q)$ can be calculated once, and stored in a variable as shown in the following Haskell example.

```
solve' f z = search 0 m where
  search p q | p > n || q < 0 = []
             | z' < z = search (p + 1) q
             | z' > z = search p (q - 1)
             | otherwise = (p, q) : search (p + 1) (q - 1)
    where z' = f p q
```

www.dbooks.org

```
m = bsearch (f 0) z (0, z)
n = bsearch (λx→f x 0) z (0, z)
```

This improved saddleback search firstly performs binary search two rounds to find the proper m, and n. Each round is bound to $O(\lg z)$ times of calculation for f; After that, it takes $O(m + n)$ time in the worst case; and $O(min(m, n))$ time in the best case. The overall performance is given in the following table.

	times of evaluation f
worst case	$2 \log z + m + n$
best case	$2 \log z + min(m, n)$

For some function $f(x, y) = a^x + b^y$, for positive integers a and b, m and n will be relative small, that the performance is close to $O(\lg z)$.

This algorithm can also be realized in imperative approach. Firstly, the binary search should be modified.

```
1: function BINARY-SEARCH(f, y, (l, u))
2:     while l < u do
3:         m ← ⌊(l+u)/2⌋
4:         if f(m) ≤ y then
5:             if y < f(m + 1) then
6:                 return m
7:             l ← m + 1
8:         else
9:             u ← m
10:    return l
```

Utilize this algorithm, the boundaries m and n can be found before performing the saddleback search.

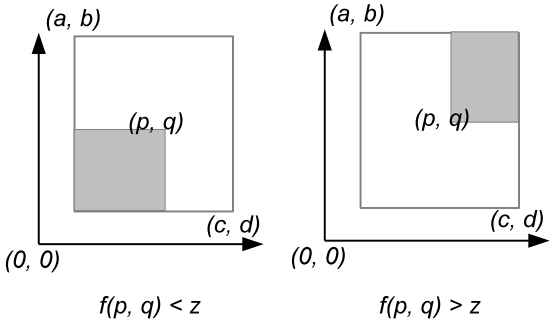
```
1: function SOLVE(f, z)
2:     m ← BINARY-SEARCH(λ_y · f(0, y), z, (0, z))
3:     n ← BINARY-SEARCH(λ_x · f(x, 0), z, (0, z))
4:     p ← 0, q ← m
5:     S ← φ
6:     while p ≤ n ∧ q ≥ 0 do
7:         z' ← f(p, q)
8:         if z' < z then
9:             p ← p + 1
10:        else if z' > z then
11:            q ← q − 1
12:        else
13:            S ← S ∪ {(p, q)}
14:            p ← p + 1, q ← q − 1
15:    return S
```

The implementation is left as exercise to the reader.

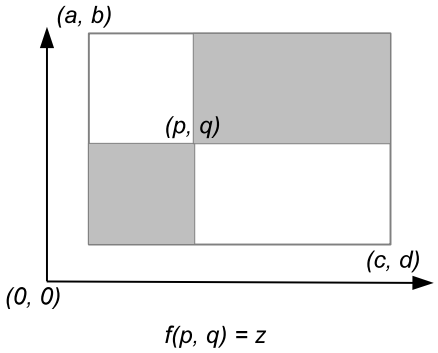
More improvement to saddleback search

In figure 13.2, two cases are shown for comparing the value of the middle point in a matrix with the given value. One case is the center value is smaller than the given value, the other is bigger. In both cases, we can only throw away $\frac{1}{4}$ candidates, and left a 'L' shape for further searching.

Actually, one important case is missing. We can extend the observation to any point inside the rectangle searching area. As shown in the figure 13.7.



(a) If $f(p, q) \neq z$, only lower-left or upper-right sub area (in gray color) can be thrown. Both left a 'L' shape.



(b) If $f(p, q) = z$, both sub areas can be thrown, the scale of the problem is halved.

Figure 13.7: The efficiency of scaling down the search domain.

Suppose we are searching in a rectangle from the upper-left corner (a, b) to the lower-right corner (c, d). If the (p, q) isn't the middle point, and $f(p, q) \neq z$. We can't ensure the area to be dropped is always $1/4$. However, if $f(p, q) = z$, as f is strict increasing, we are not only sure both the lower-left and the upper-right sub areas can be thrown, but also all the other points in the column p and row q. The problem can be scaled down fast, because only $1/2$ area is left.

This indicates us, instead of jumping to the middle point to start searching. A more efficient way is to find a point which evaluates to the target value. One straightforward way to find such a point, is to perform binary search along the center horizontal line or the center vertical line of the rectangle.

The performance of binary search along a line is logarithmic to the length of that line. A good idea is to always pick the shorter center line as shown in figure 13.8. That if the height of the rectangle is longer than the width, we perform binary search along the horizontal center line; otherwise we choose the

www.dbooks.org

vertical center line.

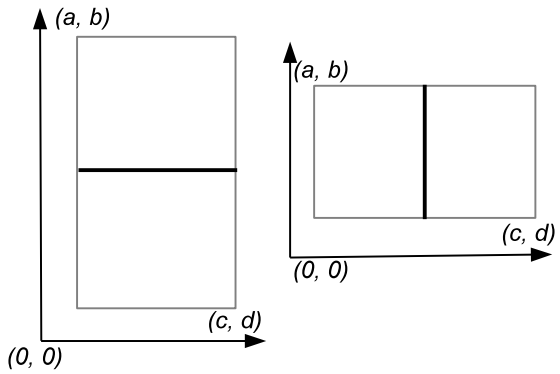


Figure 13.8: Binary search along the shorter center line.

However, what if we can't find a point (p, q) in the center line, that satisfies $f(p, q) = z$? Let's take the center horizontal line for example. even in such case, we can still find a point that $f(p, q) < z < f(p + 1, q)$. The only difference is that we can't drop the points in column p and row q completely.

Combine this conditions, the binary search along the horizontally line is to find a p, satisfies $f(p, q) \leq z < f(p + 1, q)$; While the vertical line search condition is $f(p, q) \leq z < f(p, q + 1)$.

The modified binary search ensures that, if all points in the line segment give $f(p, q) < z$, the upper bound will be found; and the lower bound will be found if they all greater than z. We can drop the whole area on one side of the center line in such case.

Sum up all the ideas, we can develop the efficient improved saddleback search as the following.

1. Perform binary search along the y axis and x axis to find the tight boundaries from $(0, m)$ to $(n, 0)$;

2. Denote the candidate rectangle as $(a, b) - (c, d)$, if the candidate rectangle is empty, the solution is empty;

3. If the height of the rectangle is longer than the width, perform binary search along the center horizontal line; otherwise, perform binary search along the center vertical line; denote the search result as (p, q);

4. If $f(p, q) = z$, record (p, q) as a solution, and recursively search two sub rectangles $(a, b) - (p - 1, q + 1)$ and $(p + 1, q - 1) - (c, d)$;

5. Otherwise, $f(p, q) \neq z$, recursively search the same two sub rectangles plus a line section. The line section is either $(p, q + 1) - (p, b)$ as shown in figure 13.9 (a); or $(p + 1, q) - (c, q)$ as shown in figure 13.9 (b).

This algorithm can be formalized as the following. The equation (13.11), and (13.12) are as same as before. A new *search* function should be defined.

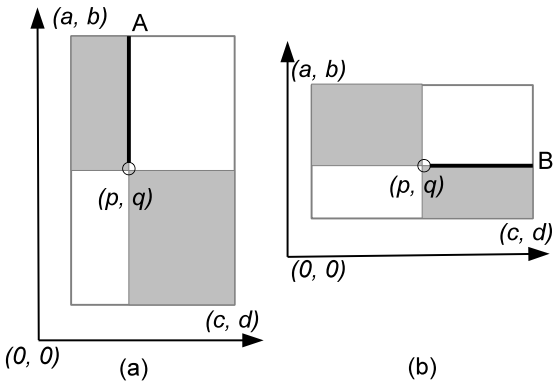


Figure 13.9: Recursively search the gray areas, the bold line should be included if $f(p, q) \neq z$.

Define $Search_{(a,b),(c,d)}$ as a function for searching rectangle with top-left corner (a, b), and bottom-right corner (c, d).

$$search_{(a,b),(c,d)} = \begin{cases} \phi & : & c < a \vee d < b \\ csearch & : & c - a < b - d \\ rsearch & : & otherwise \end{cases} \tag{13.14}$$

Function $csearch$ performs binary search in the center horizontal line to find a point (p, q) that $f(p, q) \leq z < f(p + 1, q)$. This is shown in figure 13.9 (a). There is a special edge case, that all points in the lines evaluate to values greater than z. The general binary search will return the lower bound as result, so that $(p, q) = (a, \lfloor \frac{b+d}{2} \rfloor)$. The whole upper side includes the center line can be dropped as shown in figure 13.10 (a).

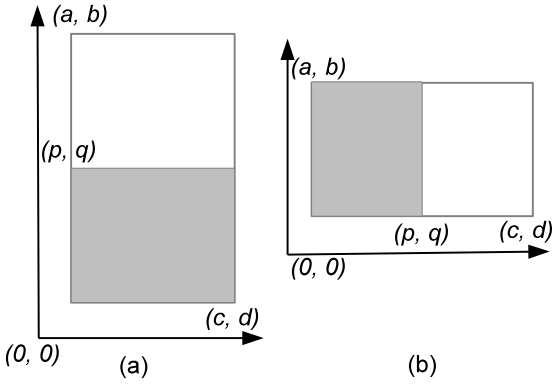


Figure 13.10: Edge cases when performing binary search in the center line.

www.dbooks.org

$$csearch = \begin{cases} search_{(p,q-1),(c,d)} & : & z < f(p,q) \\ search_{(a,b),(p-1,q+1)} \cup \{(p,q)\} \cup search_{(p+1,q-1),(c,d)} & : & f(p,q) = z \\ search_{(a,b),(p,q+1)} \cup search_{(p+1,q-1),(c,d)} & : & otherwise \end{cases} \tag{13.15}$$

Where

$$\begin{aligned} q &= \lfloor \tfrac{b+d}{2} \rfloor) \\ p &= bsearch(\lambda_x \cdot f(x,q), z, (a,c)) \end{aligned}$$

Function $rsearch$ is quite similar except that it searches in the center horizontal line.

$$rsearch = \begin{cases} search_{(a,b),(p-1,q)} & : & z < f(p,q) \\ search_{(a,b),(p-1,q+1)} \cup \{(p,q)\} \cup search_{(p+1,q-1),(c,d)} & : & f(p,q) = z \\ search_{(a,b),(p-1,q+1)} \cup search_{(p+1,q),(c,d)} & : & otherwise \end{cases} \tag{13.16}$$

Where

$$\begin{aligned} p &= \lfloor \tfrac{a+c}{2} \rfloor) \\ q &= bsearch(\lambda_y \cdot f(p,y), z, (d,b)) \end{aligned}$$

The following Haskell program implements this algorithm.

```
search f z (a, b) (c, d) | c < a || b < d = []
                         | c - a < b - d = let q = (b + d) `div` 2 in
                                           csearch (bsearch (λx → f x q) z (a, c), q)
                         | otherwise = let p = (a + c) `div` 2 in
                                       rsearch (p, bsearch (f p) z (d, b))
  where
    csearch (p, q) | z < f p q = search f z (p, q - 1) (c, d)
                   | f p q == z = search f z (a, b) (p - 1, q + 1) ++
                                  (p, q) : search f z (p + 1, q - 1) (c, d)
                   | otherwise = search f z (a, b) (p, q + 1) ++
                                 search f z (p + 1, q - 1) (c, d)
    rsearch (p, q) | z < f p q = search f z (a, b) (p - 1, q)
                   | f p q == z = search f z (a, b) (p - 1, q + 1) ++
                                  (p, q) : search f z (p + 1, q - 1) (c, d)
                   | otherwise = search f z (a, b) (p - 1, q + 1) ++
                                 search f z (p + 1, q) (c, d)
```

And the main program calls this function after performing binary search in X and Y axes.

```
solve f z = search f z (0, m) (n, 0) where
  m = bsearch (f 0) z (0, z)
  n = bsearch (λx → f x 0) z (0, z)
```

Since we drop half areas in every recursion, it takes $O(\log(mn))$ rounds of search. However, in order to locate the point (p, q), which halves the problem, we must perform binary search along the center line. which will call f about $O(\log(min(m, n)))$ times. Denote the time of searching a $m \times n$ rectangle as $T(m, n)$, the recursion relationship can be represented as the following.

$$T(m,n) = \log(min(m,n)) + 2T(\frac{m}{2}, \frac{n}{2}) \tag{13.17}$$

Suppose $m > n$, using telescope method, for $m = 2^i$, and $n = 2^j$. We have:

$$\begin{aligned} T(2^i, 2^j) &= j + 2T(2^{i-1}, 2^{j-1}) \\ &= \sum_{k=0}^{i-1} 2^k (j - k) \\ &= O(2^i (j - i)) \\ &= O(m \log(n/m)) \end{aligned} \tag{13.18}$$

Richard Bird proved that this is asymptotically optimal by a lower bound of searching a given value in $m \times n$ rectangle [1].

The imperative algorithm is almost as same as the functional version. We skip it for the sake of brevity.

Exercise 13.1

- Prove that the average case for the divide and conquer solution to k-selection problem is $O(n)$. Please refer to previous chapter about quick sort.
- Implement the imperative k-selection problem with 2-way partition, and median-of-three pivot selection.
- Implement the imperative k-selection problem to handle duplicated elements effectively.
- Realize the median-of-median k-selection algorithm and implement it in your favorite programming language.
- The $tops(k, L)$ algorithm uses list concatenation likes $A \cup \{l_1\} \cup tops(k - |A| - 1, B)$. It is linear operation which is proportion to the length of the list to be concatenated. Modify the algorithm so that the sub lists are concatenated by one pass.
- The author considered another divide and conquer solution for the k-selection problem. It finds the maximum of the first k elements and the minimum of the rest. Denote them as x, and y. If x is smaller than y, it means that all the first k elements are smaller than the rest, so that they are exactly the top k smallest; Otherwise, There are some elements in the first k should be swapped.

```
1: procedure TOPS(k, A)
2:     l ← 1
3:     u ← |A|
4:     loop
5:         i ← MAX-AT(A[l..k])
6:         j ← MIN-AT(A[k + 1..u])
7:         if A[i] < A[j] then
8:             break
9:         EXCHANGE A[l] ↔ A[j]
10:        EXCHANGE A[k + 1] ↔ A[i]
11:        l ← PARTITION(A, l, k)
12:        u ← PARTITION(A, k + 1, u)
```

www.dbooks.org

Explain why this algorithm works? What's the performance of it?

- Implement the binary search algorithm in both recursive and iterative manner, and try to verify your version automatically. You can either generate randomized data, test your program with the binary search invariant, or compare with the built-in binary search tool in your standard library.
- Find the solution to calculate the median of two sorted arrays A and B. The time should be bound to $O(\lg(|A| + |B|))$.
- Implement the improved saddleback search by firstly performing binary search to find a more accurate solution domain in your favorite imperative programming language.
- Realize the improved 2D search, by performing binary search along the shorter center line, in your favorite imperative programming language.
- Someone considers that the 2D search can be designed as the following. When search a rectangle, as the minimum value is at bottom-left, and the maximum at to-right. If the target value is less than the minimum or greater than the maximum, then there is no solution; otherwise, the rectangle is divided into 4 sub rectangles at the center point, then perform recursively searching.

```
1: procedure SEARCH(f, z, a, b, c, d)          ▷ (a, b): bottom-left (c, d): top-right
2:     if z ≤ f(a, b) ∨ f(c, d) ≥ z then
3:         if z = f(a, b) then
4:             record (a, b) as a solution
5:         if z = f(c, d) then
6:             record (c, d) as a solution
7:         return
8:     p ← ⌊(a+c)/2⌋
9:     q ← ⌊(b+d)/2⌋
10:    SEARCH(f, z, a, q, p, d)
11:    SEARCH(f, z, p, q, c, d)
12:    SEARCH(f, z, a, b, p, q)
13:    SEARCH(f, z, p, b, c, q)
```

What's the performance of this algorithm?

13.2.2 Information reuse

One interesting behavior is that people learning while searching. We do not only remember lessons which cause search fails, but also learn patterns which lead to success. This is a kind of information reusing, no matter the information is positive or negative. However, It's not easy to determine what information should be kept. Too little information isn't enough to help effective searching, while keeping too much is expensive in term of spaces.

In this section, we'll first introduce two interesting problems, Boyer-Moore majority number problem and the maximum sum of sub vector problem. Both

reuse information as little as possible. After that, two popular string matching algorithms, Knuth-Morris-Pratt algorithm and Boyer-Moore algorithm will be introduced.

Boyer-Moore majority number

Voting is quite critical to people. We use voting to choose the leader, make decision or reject a proposal. In the months when I was writing this chapter, there are three countries in the world voted their presidents. All of the three voting activities utilized computer to calculate the result.

Suppose there is a country in a small island wants a new president. According to the constitution, only if the candidate wins more than half of the votes can be selected as the president. Given a serious of votes, such as A, B, A, C, B, B, D, ..., can we develop a program tells who is the new president if there is, or indicate nobody wins more than half of the votes?

Of course this problem can be solved with brute-force by using a map. As what we did in the chapter of binary search tree[5].

```
template<typename T>
T majority(const T* xs, int n, T fail) {
    map<T, int> m;
    int i, max = 0;
    T r;
    for (i = 0; i < n; ++i)
        ++m[xs[i]];
    for (typename map<T, int>::iterator it = m.begin(); it != m.end(); ++it)
        if (it->second > max) {
            max = it->second;
            r = it->first;
        }
    return max * 2 > n ? r : fail;
}
```

This program first scan the votes, and accumulates the number of votes for each individual with a map. After that, it traverse the map to find the one with the most of votes. If the number is bigger than the half, the winner is found otherwise, it returns a special value to indicate fail.

The following pseudo code describes this algorithm.

```
1: function MAJORITY(A)
2:     M ← empty map
3:     for ∀a ∈ A do
4:         PUT(M, a, 1+ GET(M, a))
5:     max ← 0, m ← NIL
6:     for ∀(k, v) ∈ M do
7:         if max < v then
8:             max ← v, m ← k
9:     if max > |A|50% then
10:        return m
11:    else
```

[5]There is a probabilistic sub-linear space counting algorithm published in 2004, named as 'Count-min sketch'[8].

www.dbooks.org

```
12:         fail
```

For m individuals and n votes, this program firstly takes about $O(n \log m)$ time to build the map if the map is implemented in self balanced tree (red-black tree for instance); or about $O(n)$ time if the map is hash table based. However, the hash table needs more space. Next the program takes $O(m)$ time to traverse the map, and find the majority vote. The following table lists the time and space performance for different maps.

map	time	space
self-balanced tree	$O(n \log m)$	$O(m)$
hashing	$O(n)$	$O(m)$ at least

Boyer and Moore invented a cleaver algorithm in 1980, which can pick the majority element with only one scan if there is. Their algorithm only needs $O(1)$ space [7].

The idea is to record the first candidate as the winner so far, and mark him with 1 vote. During the scan process, if the winner being selected gets another vote, we just increase the vote counter; otherwise, it means somebody vote against this candidate, so the vote counter should be decreased by one. If the vote counter becomes zero, it means this candidate is voted out; We select the next candidate as the new winner and repeat the above scanning process.

Suppose there is a series of votes: A, B, C, B, B, C, A, B, A, B, B, D, B. Below table illustrates the steps of this processing.

winner	count	scan position
A	1	**A**, B, C, B, B, C, A, B, A, B, B, D, B
A	0	A, **B**, C, B, B, C, A, B, A, B, B, D, B
C	1	A, B, **C**, B, B, C, A, B, A, B, B, D, B
C	0	A, B, C, **B**, B, C, A, B, A, B, B, D, B
B	1	A, B, C, B, **B**, C, A, B, A, B, B, D, B
B	0	A, B, C, B, B, **C**, A, B, A, B, B, D, B
A	1	A, B, C, B, B, C, **A**, B, A, B, B, D, B
A	0	A, B, C, B, B, C, A, **B**, A, B, B, D, B
A	1	A, B, C, B, B, C, A, B, **A**, B, B, D, B
A	0	A, B, C, B, B, C, A, B, A, **B**, B, D, B
B	1	A, B, C, B, B, C, A, B, A, B, **B**, D, B
B	0	A, B, C, B, B, C, A, B, A, B, B, **D**, B
B	1	A, B, C, B, B, C, A, B, A, B, B, D, **B**

The key point is that, if there exits the majority greater than 50%, it can't be voted out by all the others. However, if there are not any candidates win more than half of the votes, the recorded 'winner' is invalid. Thus it is necessary to perform a second round scan for verification.

The following pseudo code illustrates this algorithm.

```
1: function MAJORITY(A)
2:     c ← 0
3:     for i ← 1 to |A| do
4:         if c = 0 then
5:             x ← A[i]
6:         if A[i] = x then
7:             c ← c + 1
8:         else
9:             c ← c − 1
```

```
10:     return x
```

If there is the majority element, this algorithm takes one pass to scan the votes. In every iteration, it either increases or decreases the counter according to the vote is support or against the current selection. If the counter becomes zero, it means the current selection is voted out. So the new one is selected as the updated candidate for further scan.

The process is linear $O(n)$ time, and the spaces needed are just two variables. One for recording the selected candidate so far, the other is for vote counting.

Although this algorithm can find the majority element if there is. it still picks an element even there isn't. The following modified algorithm verifies the final result with another round of scan.

```
1: function MAJORITY(A)
2:     c ← 0
3:     for i ← 1 to |A| do
4:         if c = 0 then
5:             x ← A[i]
6:         if A[i] = x then
7:             c ← c + 1
8:         else
9:             c ← c − 1
10:    c ← 0
11:    for i ← 1 to |A| do
12:        if A[i] = x then
13:            c ← c + 1
14:    if c > %50|A| then
15:        return x
16:    else
17:        fail
```

Even with this verification process, the algorithm is still bound to $O(n)$ time, and the space needed is constant. The following ISO C++ program implements this algorithm [6].

```
template<typename T>
T majority(const T* xs, int n, T fail) {
    T m;
    int i, c;
    for (i = 0, c = 0; i < n; ++i) {
        if (!c)
            m = xs[i];
        c += xs[i] == m ? 1 : -1;
    }
    for (i = 0, c = 0; i < n; ++i, c += xs[i] == m);
    return c * 2 > n ? m : fail;
}
```

Boyer-Moore majority algorithm can also be realized in purely functional approach. Different from the imperative settings, which use variables to record and update information, accumulators are used to define the core algorithm. Define function $maj(c, n, L)$, which takes a list of votes L, a selected candidate

[6] We actually uses the ANSI C style. The C++ template is only used to generalize the type of the element

www.dbooks.org

c so far, and a counter n. For non empty list L, we initialize c as the first vote l_1, and set the counter as 1 to start the algorithm: $maj(l_1, 1, L')$, where L' is the rest votes except for l_1. Below are the definition of this function.

$$maj(c, n, L) = \begin{cases} c & : & L = \phi \\ maj(c, n+1, L') & : & l_1 = c \\ maj(l_1, 1, L') & : & n = 0 \wedge l_1 \neq c \\ maj(c, n-1, L') & : & otherwise \end{cases} \tag{13.19}$$

We also need to define a function, which can verify the result. The idea is that, if the list of votes is empty, the final result is a failure; otherwise, we start the Boyer-Moore algorithm to find a candidate c, then we scan the list again to count the total votes c wins, and verify if this number is not less than the half.

$$majority(L) = \begin{cases} fail & : & L = \phi \\ c & : & c = maj(l_1, 1, L'), |\{x | x \in L, x = c\}| > \%50|L| \\ fail & : & otherwise \end{cases} \tag{13.20}$$

Below Haskell example code implements this algorithm.

```
majority :: (Eq a) ⇒ [a] → Maybe a
majority [] = Nothing
majority (x:xs) = let m = maj x 1 xs in verify m (x:xs)

maj c n [] = c
maj c n (x:xs) | c == x = maj c (n+1) xs
               | n == 0 = maj x 1 xs
               | otherwise = maj c (n-1) xs

verify m xs = if 2 * (length $ filter (==m) xs) > length xs
              then Just m else Nothing
```

Maximum sum of sub vector

Jon Bentley presents another interesting puzzle which can be solved by using quite similar idea in [4]. The problem is to find the maximum sum of sub vector. For example in the following array, The sub vector {19, -12, 1, 9, 18} yields the biggest sum 35.

3	-13	19	-12	1	9	18	-16	15	-15

Note that it is only required to output the value of the maximum sum. If all the numbers are positive, the answer is definitely the sum of all. Another special case is that all numbers are negative. We define the maximum sum is 0 for an empty sub vector.

Of course we can find the answer with brute-force, by calculating all sums of sub vectors and picking the maximum. Such naive method is typical quadratic.

```
1: function MAX-SUM(A)
2:     m ← 0
3:     for i ← 1 to |A| do
4:         s ← 0
5:         for j ← i to |A| do
6:             s ← s + A[j]
```

```
7:            m ← MAX(m, s)
8:    return m
```

The brute force algorithm does not reuse any information in previous search. Similar with Boyer-Moore majority vote algorithm, we can record the maximum sum end to the position where we are scanning. Of course we also need record the biggest sum found so far. The following figure illustrates this idea and the invariant during scan.

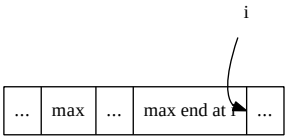


Figure 13.11: Invariant during scan.

At any time when we scan to the i-th position, the max sum found so far is recorded as A. At the same time, we also record the biggest sum end at i as B. Note that A and B may not be the same, in fact, we always maintain $B \leq A$. and when B becomes greater than A by adding with the next element, we update A with this new value. When B becomes negative, this happens when the next element is a negative number, we reset it to 0. The following tables illustrated the steps when we scan the example vector $\{3, -13, 19, -12, 1, 9, 18, -16, 15, -15\}$.

max sum	max end at i	list to be scan
0	0	$\{3, -13, 19, -12, 1, 9, 18, -16, 15, -15\}$
3	3	$\{-13, 19, -12, 1, 9, 18, -16, 15, -15\}$
3	0	$\{19, -12, 1, 9, 18, -16, 15, -15\}$
19	19	$\{-12, 1, 9, 18, -16, 15, -15\}$
19	7	$\{1, 9, 18, -16, 15, -15\}$
19	8	$\{9, 18, -16, 15, -15\}$
19	17	$\{18, -16, 15, -15\}$
35	35	$\{-16, 15, -15\}$
35	19	$\{15, -15\}$
35	34	$\{-15\}$
35	19	$\{\}$

This algorithm can be described as below.

```
1: function MAX-SUM(V)
2:     A ← 0, B ← 0
3:     for i ← 1 to |V| do
4:         B ← MAX(B + V[i], 0)
5:         A ← MAX(A, B)
```

It is trivial to implement this linear time algorithm, that we skip the details here.

This algorithm can also be defined in functional approach. Instead of mutating variables, we use accumulator to record A and B. In order to search the maximum sum of list L, we call the below function with $max_{sum}(0, 0, L)$.

$$max_{sum}(A, B, L) = \begin{cases} A & : \quad L = \phi \\ max_{sum}(A', B', L') & : \quad otherwise \end{cases} \tag{13.21}$$

www.dbooks.org

Where

$$B' = max(l_1 + B, 0)$$
$$A' = max(A, B')$$

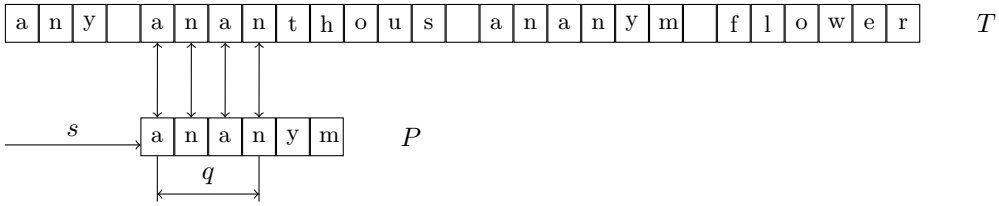
Below Haskell example code implements this algorithm.

```
maxsum = msum 0 0 where
  msum a _ [] = a
  msum a b (x:xs) = let b' = max (x+b) 0
                        a' = max a b'
                    in msum a' b' xs
```

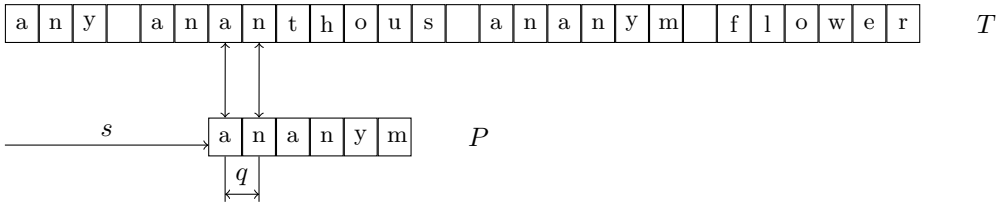
KMP

String matching is another important type of searching. Almost all the software editors are equipped with tools to find string in the text. In chapters about Trie, Patricia, and suffix tree, we have introduced some powerful data structures which can help to search string. In this section, we introduce another two string matching algorithms all based on information reusing.

Some programming environments provide built-in string search tools, however, most of them are brute-force solution including 'strstr' function in ANSI C standard library, 'find' in C++ standard template library, 'indexOf' in Java Development Kit etc. Figure 13.12 illustrate how such character-by-character comparison process works.



(a) The offset $s = 4$, after matching $q = 4$ characters, the 5th mismatch.



(b) Move $s = 4 + 2 = 6$, directly.

Figure 13.12: Match 'ananym' in 'any ananthous ananym flower'.

Suppose we search a pattern P in text T, as shown in figure 13.12 (a), at offset $s = 4$, the process examines every character in P and T to check if they are same. It successfully matches the first 4 characters 'anan'. However, the 5th character in the pattern string is 'y'. It doesn't match the corresponding character in the text, which is 't'.

At this stage, the brute-force solution terminates the attempt, increases s by one to 5, and restart the comparison between 'ananym' and 'nantho...'. Actually,

we can increase s not only by one. This is because we have already known that the first four characters 'anan' have been matched, and the failure happens at the 5th position. Observe the two letters prefix 'an' of the pattern string is also a suffix of 'anan' that we have matched so far. A more effective way is to shift s by two but not one, which is shown in figure 13.12 (b). By this means, we reused the information that 4 characters have been matched. This helps us to skip invalid positions as many as possible.

Knuth, Morris and Pratt presented this idea in [9] and developed a novel string matching algorithm. This algorithm is later called as 'KMP', which is consist of the three authors' initials.

For the sake of brevity, we denote the first k characters of text T as T_k. Which means T_k is the k-character prefix of T.

The key point to shift s effectively is to find a function of q, where q is the number of characters matched successfully. For instance, q is 4 in figure 13.12 (a), as the 5th character doesn't match.

Consider what situation we can shift s more than 1. As shown in figure 13.13, if we can shift the pattern P ahead, there must exist k, so that the first k characters are as same as the last k characters of P_q. In other words, the prefix P_k is suffix of P_q.

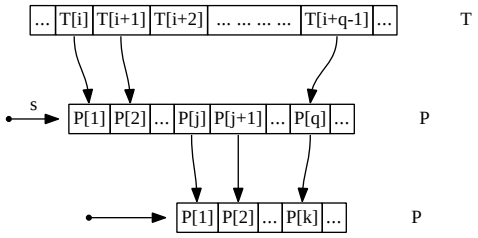


Figure 13.13: P_k is both prefix of P_q and suffix of P_q.

It's possible that there is no such a prefix that is the suffix at the same time. If we treat empty string as both the prefix and the suffix of any others, there must be at least one solution that $k = 0$. It's also quite possible that there are multiple k satisfy. To avoid missing any possible matching positions, we have to find the biggest k. We can define a *prefix function* $\pi(q)$ which tells us where we can fallback if the $(q+1)$-th character does not match [2].

$$\pi(q) = max\{k | k < q \wedge P_k \sqsupset P_q\} \tag{13.22}$$

Where $\sqsupset$ is read as 'is suffix of'. For instance, $A \sqsupset B$ means A is suffix of B. This function is used as the following. When we match pattern P against text T from offset s, If it fails after matching q characters, we next look up $\pi(q)$ to get a fallback q', and retry to compare $P[q']$ with the previous unmatched character. Based on this idea, the core algorithm of KMP can be described as the following.

```
1: function KMP(T, P)
2:     n ← |T|, m ← |P|
3:     build prefix function π from P
4:     q ← 0                    ▷ How many characters have been matched so far.
```

www.dbooks.org

```
5:    for i ← 1 to n do
6:        while q > 0 ∧ P[q + 1] ≠ T[i] do
7:            q ← π(q)
8:        if P[q + 1] = T[i] then
9:            q ← q + 1
10:       if q = m then
11:           found one solution at i − m
12:           q ← π(q)                          ▷ look for next solution
```

Although the definition of prefix function $\pi(q)$ is given in equation (13.22), realizing it blindly by finding the longest suffix isn't effective. Actually we can use the idea of information reusing again to build the prefix function.

The trivial edge case is that, the first character doesn't match. In this case the longest prefix, which is also the suffix is definitely empty, so $\pi(1) = k = 0$. We record the longest prefix as P_k. In this edge case $P_k = P_0$ is the empty string.

After that, when we scan at the q-th character in the pattern string P, we hold the invariant that the prefix function values $\pi(i)$ for i in $\{1, 2, ..., q-1\}$ have already been recorded, and P_k is the longest prefix which is also the suffix of P_{q-1}. As shown in figure 13.14, if $P[q] = P[k+1]$, A bigger k than before is found, we can increase the maximum of k by one; otherwise, if they are not same, we can use $\pi(k)$ to fallback to a shorter prefix $P_{k'}$ where $k' = \pi(k)$, and check if the next character after this new prefix is same as the q-th character. We need repeat this step until either k becomes zero (which means only empty string satisfies), or the q-th character matches.

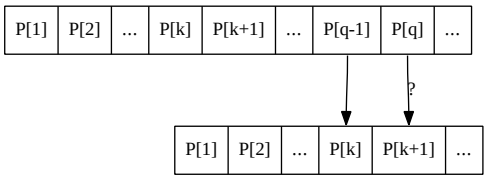


Figure 13.14: P_k is suffix of P_{q-1}, $P[q]$ and $P[k+1]$ are compared.

Realizing this idea gives the KMP prefix building algorithm.

```
1: function BUILD-PREFIX-FUNCTION(P)
2:     m ← |P|, k ← 0
3:     π(1) ← 0
4:     for q ← 2 to m do
5:         while k > 0 ∧ P[q] ≠ P[k + 1] do
6:             k ← π(k)
7:         if P[q] = P[k + 1] then
8:             k ← k + 1
9:         π(q) ← k
10:    return π
```

The following table lists the steps of building prefix function for pattern string 'ananym'. Note that the k in the table actually means the maximum k satisfies equation (13.22).

q	P_q	k	P_k
1	a	0	“”
2	an	0	“”
3	ana	1	a
4	anan	2	an
5	anany	0	“”
6	ananym	0	“”

Translating the KMP algorithm to Python gives the below example code.

```
def kmp_match(w, p):
    n = len(w)
    m = len(p)
    fallback = fprefix(p)
    k = 0 # how many elements have been matched so far.
    res = []
    for i in range(n):
        while k > 0 and p[k] != w[i]:
            k = fallback[k] #fall back
        if p[k] == w[i]:
            k = k + 1
        if k == m:
            res.append(i+1-m)
            k = fallback[k-1] # look for next
    return res

def fprefix(p):
    m = len(p)
    t = [0]*m  # fallback table
    k = 0
    for i in range(2, m):
        while k>0 and p[i-1] != p[k]:
            k = t[k-1] #fallback
        if p[i-1] == p[k]:
            k = k + 1
        t[i] = k
    return t
```

The KMP algorithm builds the prefix function for the pattern string as a kind of pre-processing before the search. Because of this, it can reuse as much information of the previous matching as possible.

The amortized performance of building the prefix function is $O(m)$. This can be proved by using potential method as in [2]. Using the similar method, it can be proved that the matching algorithm itself is also linear. Thus the total performance is $O(m+n)$ at the expense of the $O(m)$ space to record the prefix function table.

It seems that varies pattern string would affect the performance of KMP. Considering the case that we are finding pattern string ‘aaa...a’ of length m in a string ‘aaa...a’ of length n. All the characters are same, when the last character in the pattern is examined, we can only fallback by 1, and this 1 character fallback repeats until it falls back to zero. Even in this extreme case, KMP algorithm still holds its linear performance (why?). Please try to consider more cases such as $P = aaaa...b$, $T = aaaa...a$ and so on.

www.dbooks.org

Purely functional KMP algorithm

It is not easy to realize KMP matching algorithm in purely functional manner. The imperative algorithm represented so far intensely uses array to record prefix function values. Although it is possible to utilize sequence like structure in purely functional settings, it is typically implemented with finger tree. Unlike native arrays, finger tree needs logarithm time for random accessing[7].

Richard Bird presents a formal program deduction to KMP algorithm by using fold fusion law in chapter 17 of [1]. In this section, we show how to develop purely functional KMP algorithm step by step from a brute-force prefix function creation method.

Both text string and pattern are represented as singly linked-list in purely functional settings. During the scan process, these two lists are further partitioned, every one is broken into two parts. As shown in figure 13.15, The first j characters in the pattern string have been matched. $T[i+1]$ and $P[j+1]$ will be compared next. If they are same, we need append the character to the matched part. However, since strings are essentially singly linked list, such appending is proportion to j.

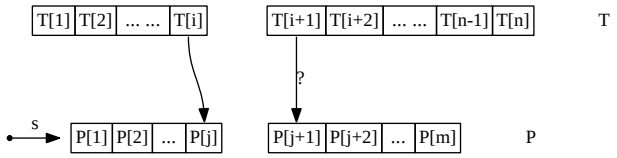


Figure 13.15: The first j characters in P are matched, next compare $P[j+1]$ with $T[i+1]$.

Denote the first i characters as T_p, which means the prefix of T, the rest characters as T_s for suffix; Similarly, the first j characters as P_p, and the rest as P_s; Denote the first character of T_s as t, the first character of P_s as p. We have the following 'cons' relationship.

$$T_s = cons(t, T'_s)$$
$$P_s = cons(p, P'_s)$$

If $t = p$, note the following updating process is bound to linear time.

$$T'_p = T_p \cup \{t\}$$
$$P'_p = P_p \cup \{p\}$$

We've introduced a method in the chapter about purely functional queue, which can solve this problem. By using a pair of front and rear list, we can turn the linear time appending to constant time linking. The key point is to represent the prefix part in reverse order.

$$\begin{aligned} T &= T_p \cup T_s = reverse(reverse(T_p)) \cup T_s = reverse(\overleftarrow{T_p}) \cup T_s \\ P &= P_p \cup P_s = reverse(reverse(P_p)) \cup P_s = reverse(\overleftarrow{P_p}) \cup P_s \end{aligned} \quad (13.23)$$

[7]Again, we don't use native array, even it is supported in some functional programming environments like Haskell.

The idea is to using pair $(\overleftarrow{T_p}, T_s)$ and $(\overleftarrow{P_p}, P_s)$ instead. With this change, the if $t = p$, we can update the prefix part fast in constant time.

$$\begin{aligned}\overleftarrow{T'_p} &= cons(t, \overleftarrow{T_p}) \\ \overleftarrow{P'_p} &= cons(p, \overleftarrow{P_p})\end{aligned} \tag{13.24}$$

The KMP matching algorithm starts by initializing the success prefix parts to empty strings as the following.

$$search(P, T) = kmp(\pi, (\phi, P)(\phi, T)) \tag{13.25}$$

Where π is the prefix function we explained before. The core part of KMP algorithm, except for the prefix function building, can be defined as below.

$$kmp(\pi, (\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) = \begin{cases} \{|\overleftarrow{T_p}|\} & : & P_s = \phi \wedge T_s = \phi \\ \phi & : & P_s \neq \phi \wedge T_s = \phi \\ \{|\overleftarrow{T_p}\} \cup kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) & : & P_s = \phi \wedge T_s \neq \phi \\ kmp(\pi, (\overleftarrow{P'_p}, P'_s), (\overleftarrow{T'_p}, T'_s)) & : & t = p \\ kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T'_p}, T'_s)) & : & t \neq p \wedge \overleftarrow{P_p} = \phi \\ kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) & : & t \neq p \wedge \overleftarrow{P_p} \neq \phi \end{cases} \tag{13.26}$$

The first clause states that, if the scan successfully ends to both the pattern and text strings, we get a solution, and the algorithm terminates. Note that we use the right position in the text string as the matching point. It's easy to use the left position by subtracting with the length of the pattern string. For sake of brevity, we switch to right position in functional solutions.

The second clause states that if the scan arrives at the end of text string, while there are still rest of characters in the pattern string haven't been matched, there is no solution. And the algorithm terminates.

The third clause states that, if all the characters in the pattern string have been successfully matched, while there are still characters in the text haven't been examined, we get a solution, and we fallback by calling prefix function π to go on searching other solutions.

The fourth clause deals with the case, that the next character in pattern string and text are same. In such case, the algorithm advances one character ahead, and recursively performs searching.

If the the next characters are not same and this is the first character in the pattern string, we just need advance to next character in the text, and try again. Otherwise if this isn't the first character in the pattern, we call prefix function π to fallback, and try again.

The brute-force way to build the prefix function is just to follow the definition equation (13.22).

$$\pi(\overleftarrow{P_p}, P_s) = (\overleftarrow{P'_p}, P'_s) \tag{13.27}$$

where

$$\begin{aligned}P'_p &= longest(\{s | s \in prefixes(P_p), s \sqsupset P_p\}) \\ P'_s &= P - P'_p\end{aligned}$$

www.dbooks.org

Every time when calculate the fallback position, the algorithm naively enumerates all prefixes of P_p, checks if it is also the suffix of P_p, and then pick the longest one as result. Note that we reuse the subtraction symbol here for list differ operation.

There is a tricky case which should be avoided. Because any string itself is both its prefix and suffix. Say $P_p \sqsubset P_p$ and $P_p \sqsupset P_p$. We shouldn't enumerate P_p as a candidate prefix. One solution of such prefix enumeration can be realized as the following.

$$prefixes(L) = \begin{cases} \{\phi\} & : \quad L = \phi \vee |L| = 1 \\ cons(\phi, map(\lambda_s \cdot cons(l_1, s), prefixes(L'))) & : \quad otherwise \end{cases} \tag{13.28}$$

Below Haskell example program implements this version of string matching algorithm.

```
kmpSearch1 ptn text = kmpSearch' next ([], ptn) ([], text)

kmpSearch' _ (sp, []) (sw, []) = [length sw]
kmpSearch' _ _ (_, []) = []
kmpSearch' f (sp, []) (sw, ws) = length sw : kmpSearch' f (f sp []) (sw, ws)
kmpSearch' f (sp, (p:ps)) (sw, (w:ws))
    | p == w = kmpSearch' f ((p:sp), ps) ((w:sw), ws)
    | otherwise = if sp ==[] then kmpSearch' f (sp, (p:ps)) ((w:sw), ws)
                  else kmpSearch' f (f sp (p:ps)) (sw, (w:ws))

next sp ps = (sp', ps') where
    prev = reverse sp
    prefix = longest [xs | xs ← inits prev, xs `isSuffixOf` prev]
    sp' = reverse prefix
    ps' = (prev ++ ps) \\ prefix
    longest = maximumBy (compare `on` length)

inits [] = [[]]
inits [_] = [[]]
inits (x:xs) = [] : (map (x:) $ inits xs)
```

This version does not only perform poorly, but it is also complex. We can simplify it a bit. Observing the KMP matching is a scan process from left to the right of the text, it can be represented with folding (refer to Appendix A for detail). Firstly, we can augment each character with an index for folding like below.

$$zip(T, \{1, 2, ...\}) \tag{13.29}$$

Zipping the text string with infinity natural numbers gives list of pairs. For example, text string 'The quick brown fox jumps over the lazy dog' turns into (T, 1), (h, 2), (e, 3), ... (o, 42), (g, 43).

The initial state for folding contains two parts, one is the pair of pattern (P_p, P_s), with prefix starts from empty, and the suffix is the whole pattern string (ϕ, P). For illustration purpose only, we revert back to normal pairs but not $(\overleftarrow{P_p}, P_s)$ notation. It can be easily replaced with reversed form in the finalized version. This is left as exercise to the reader. The other part is a

list of positions, where the successful matching are found. It starts from empty list. After the folding finishes, this list contains all solutions. What we need is to extract this list from the final state. The core KMP search algorithm is simplified like this.

$$kmp(P, T) = snd(fold(search, ((\phi, P), \phi), zip(T, \{1, 2, ...\}))) \tag{13.30}$$

The only 'black box' is the *search* function, which takes a state, and a pair of character and index, and it returns a new state as result. Denote the first character in P_s as p and the rest characters as P'_s ($P_s = cons(p, P'_s)$), we have the following definition.

$$search(((P_p, P_s), L), (c, i)) = \begin{cases} ((P_p \cup p, P'_s), L \cup \{i\}) & : \quad p = c \wedge P'_s = \phi \\ ((P_p \cup p, P'_s), L) & : \quad p = c \wedge P'_s \neq \phi \\ ((P_p, P_s), L) & : \quad P_p = \phi \\ search((\pi(P_p, P_s), L), (c, i)) & : \quad otherwise \end{cases} \tag{13.31}$$

If the first character in P_s matches the current character c during scan, we need further check if all the characters in the pattern have been examined, if so, we successfully find a solution, This position i in list L is recorded; Otherwise, we advance one character ahead and go on. If p does not match c, we need fallback for further retry. However, there is an edge case that we can't fallback any more. P_p is empty in this case, and we need do nothing but keep the current state.

The prefix-function π developed so far can also be improved a bit. Since we want to find the longest prefix of P_p, which is also suffix of it, we can scan from right to left instead. For any non empty list L, denote the first element as l_1, and all the rest except for the first one as L', define a function $init(L)$, which returns all the elements except for the last one as below.

$$init(L) = \begin{cases} \phi & : \quad |L| = 1 \\ cons(l_1, init(L')) & : \quad otherwise \end{cases} \tag{13.32}$$

Note that this function can not handle empty list. The idea of scan from right to left for P_p is first check if $init(P_p) \sqsupset P_p$, if yes, then we are done; otherwise, we examine if $init(init(P_p))$ is OK, and repeat this till the left most. Based on this idea, the prefix-function can be modified as the following.

$$\pi(P_p, P_s) = \begin{cases} (P_p, P_s) & : \quad P_p = \phi \\ fallback(init(P_p), cons(last(P_p), P_s)) & : \quad otherwise \end{cases} \tag{13.33}$$

Where

$$fallback(A, B) = \begin{cases} (A, B) & : \quad A \sqsupset P_p \\ (init(A), cons(last(A), B)) & : \quad otherwise \end{cases} \tag{13.34}$$

Note that fallback always terminates because empty string is suffix of any string. The $last(L)$ function returns the last element of a list, it is also a linear time operation (refer to Appendix A for detail). However, it's constant

www.dbooks.org

operation if we use $\overleftarrow{P_p}$ approach. This improved prefix-function is bound to linear time. It is still quite slower than the imperative algorithm which can look up prefix-function in constant $O(1)$ time. The following Haskell example program implements this minor improvement.

```
failure ([], ys) = ([], ys)
failure (xs, ys) = fallback (init xs) (last xs:ys) where
    fallback as bs | as `isSuffixOf` xs = (as, bs)
                   | otherwise = fallback (init as) (last as:bs)

kmpSearch ws txt = snd $ foldl f (([], ws), []) (zip txt [1..]) where
    f (p@(xs, (y:ys)), ns) (x, n) | x == y = if ys==[] then ((xs++[y], ys), ns++[n])
                                             else ((xs++[y], ys), ns)
                                  | xs == [] = (p, ns)
                                  | otherwise = f (failure p, ns) (x, n)
    f (p, ns) e = f (failure p, ns) e
```

The bottleneck is that we can not use native array to record prefix functions in purely functional settings. In fact the prefix function can be understood as a state transform function. It transfer from one state to the other according to the matching is success or fail. We can abstract such state changing as a tree. In environment supporting algebraic data type, Haskell for example, such state tree can be defined like below.

```
data State a = E | S a (State a) (State a)
```

A state is either empty, or contains three parts: the current state, the new state if match fails, and the new state if match succeeds. Such definition is quite similar to the binary tree. We can call it 'left-fail, right-success' tree. The state we are using here is (P_p, P_s).

Similar as imperative KMP algorithm, which builds the prefix function from the pattern string, the state transforming tree can also be built from the pattern. The idea is to build the tree from the very beginning state (ϕ, P), with both its children empty. We replace the left child with a new state by calling π function defined above, and replace the right child by advancing one character ahead. There is an edge case, that when the state transfers to (P, ϕ), we can not advance any more in success case, such node only contains child for failure case. The build function is defined as the following.

$$build((P_p, P_s), \phi, \phi) = \begin{cases} build(\pi(P_p, P_s), \phi, \phi) & : \quad P_s = \phi \\ build((P_p, P_s), L, R) & : \quad otherwise \end{cases} \tag{13.35}$$

Where

$$L = build(\pi(P_p, P_s), \phi, \phi)$$
$$R = build((P_s \cup \{p\}, P'_s), \phi, \phi))$$

The meaning of p and P'_s are as same as before, that p is the first character in P_s, and P'_s is the rest characters. The most interesting point is that the build function will never stop. It endless build a infinite tree. In strict programming environment, calling this function will freeze. However, in environments support lazy evaluation, only the nodes have to be used will be created. For example, both Haskell and Scheme/Lisp are capable to construct such infinite state tree. In imperative settings, it is typically realized by using pointers which links to ancestor of a node.

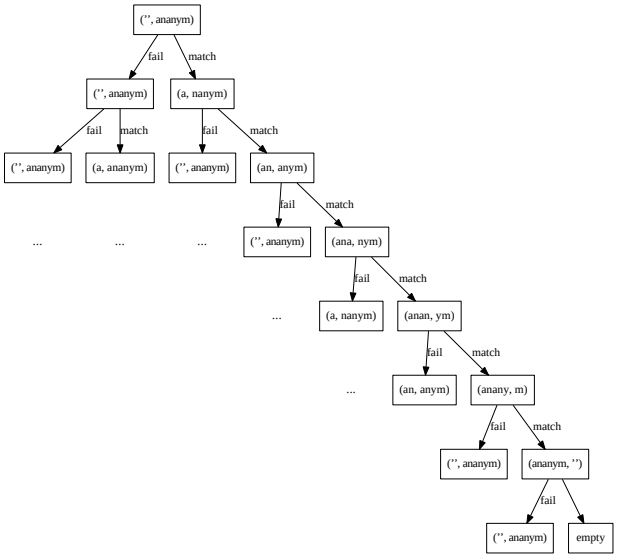


Figure 13.16: The infinite state tree for pattern 'ananym'.

Figure 13.16 illustrates such an infinite state tree for pattern string 'ananym'. Note that the right most edge represents the case that the matching continuously succeed for all characters. After that, since we can't match any more, so the right sub-tree is empty. Base on this fact, we can define a auxiliary function to test if a state indicates the whole pattern is successfully matched.

$$match((P_p, P_s), L, R) = \begin{cases} True & : & P_s = \phi \\ False & : & otherwise \end{cases} \tag{13.36}$$

With the help of state transform tree, we can realize KMP algorithm in an automaton manner.

$$kmp(P, T) = snd(fold(search, (Tr, []), zip(T, \{1, 2, ...\}))) \tag{13.37}$$

Where the tree $Tr = build((\phi, P), \phi, \phi)$ is the infinite state transform tree. Function *search* utilizes this tree to transform the state according to match or fail. Denote the first character in P_s as p, the rest characters as P'_s, and the matched positions found so far as A.

$$search((((P_p, P_s), L, R), A), (c, i)) = \begin{cases} (R, A \cup \{i\}) & : & p = c \wedge match(R) \\ (R, A) & : & p = c \wedge \neg match(R) \\ ((((P_p, P_s), L, R), A) & : & P_p = \phi \\ search((L, A), (c, i)) & : & otherwise \end{cases} \tag{13.38}$$

The following Haskell example program implements this algorithm.

```
data State a = E | S a (State a) (State a) -- state, ok-state, fail-state
               deriving (Eq, Show)
```

www.dbooks.org

```
build :: (Eq a)⇒State ([a], [a]) → State ([a], [a])
build (S s@(xs, []) E E) = S s (build (S (failure s) E E)) E
build (S s@(xs, (y:ys)) E E) = S s l r where
    l = build (S (failure s) E E) -- fail state
    r = build (S (xs++[y], ys) E E)

matched (S (_, []) _ _) = True
matched _ = False

kmpSearch3 :: (Eq a) ⇒ [a] → [a] → [Int]
kmpSearch3 ws txt = snd $ foldl f (auto, []) (zip txt [1..]) where
    auto = build (S ([], ws) E E)
    f (s@(S (xs, ys) l r), ns) (x, n)
        | [x] `isPrefixOf` ys = if matched r then (r, ns++[n])
                                else (r, ns)
        | xs == [] = (s, ns)
        | otherwise = f (l, ns) (x, n)
```

The bottle-neck is that the state tree building function calls π to fallback. While current definition of π isn't effective enough, because it enumerates all candidates from right to the left every time.

Since the state tree is infinite, we can adopt some common treatment for infinite structures. One good example is the Fibonacci series. The first two Fibonacci numbers are defined as 0 and 1; the rest Fibonacci numbers can be obtained by adding the previous two numbers.

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned} \tag{13.39}$$

Thus the Fibonacci numbers can be list one by one as the following

$$\begin{aligned} &F_0 = 0 \\ &F_1 = 1 \\ &F_2 = F_1 + F_0 \\ &F_3 = F_2 + F_1 \\ &... \end{aligned} \tag{13.40}$$

We can collect all numbers in both sides, and define $F = \{0, 1, F_1, F_2, ...\}$, Thus we have the following equation.

$$\begin{aligned} F = & \{0, 1, F_1 + F_0, F_2 + F_1, ...\} \\ = & \{0, 1\} \cup \{x + y | x \in \{F_0, F_1, F_2, ...\}, y \in \{F_1, F_2, F_3, ...\}\} \\ = & \{0, 1\} \cup \{x + y | x \in F, y \in F'\} \end{aligned} \tag{13.41}$$

Where $F' = tail(F)$ is all the Fibonacci numbers except for the first one. In environments support lazy evaluation, like Haskell for instance, this definition can be expressed like below.

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

The recursive definition for infinite Fibonacci series indicates an idea which can be used to get rid of the fallback function π. Denote the state transfer tree

as T, we can define the transfer function when matching a character on this tree as the following.

$$trans(T, c) = \begin{cases} root & : & T = \phi \\ R & : & T = ((P_p, P_s), L, R), c = p \\ trans(L, c) & : & otherwise \end{cases} \tag{13.42}$$

If we match a character against empty node, we transfer to the root of the tree. We'll define the root later soon. Otherwise, we compare if the character c is as same as the first character p in P_s. If they match, then we transfer to the right sub tree for this success case; otherwise, we transfer to the left sub tree for fail case.

With transfer function defined, we can modify the previous tree building function accordingly. This is quite similar to the previous Fibonacci series definition.

$$build(T, (P_p, P_s)) = ((P_p, P_s), T, build(trans(T, p), (P_p \cup \{p\}, P'_s)))$$

The right hand of this equation contains three parts. The first one is the state that we are matching (P_p, P_s); If the match fails, Since T itself can handle any fail case, we use it directly as the left sub tree; otherwise we recursive build the right sub tree for success case by advancing one character ahead, and calling transfer function we defined above.

However, there is an edge case which has to be handled specially, that if P_s is empty, which indicates a successful match. As defined above, there isn't right sub tree any more. Combining these cases gives the final building function.

$$build(T, (P_p, P_s)) = \begin{cases} ((P_p, P_s), T, \phi) & : & P_s = \phi \\ ((P_p, P_s), T, build(trans(T, p), (P_p \cup \{p\}, P'_s))) & : & otherwise \end{cases} \tag{13.43}$$

The last brick is to define the root of the infinite state transfer tree, which initializes the building.

$$root = build(\phi, (\phi, P)) \tag{13.44}$$

And the new KMP matching algorithm is modified with this root.

$$kmp(P, T) = snd(fold(trans, (root, []), zip(T, \{1, 2, ...\}))) \tag{13.45}$$

The following Haskell example program implements this final version.

```
kmpSearch ws txt = snd $ foldl tr (root, []) (zip txt [1..]) where
    root = build' E ([], ws)
    build' fails (xs, []) = S (xs, []) fails E
    build' fails s@(xs, (y:ys)) = S s fails succs where
        succs = build' (fst (tr (fails, []) (y, 0))) (xs++[y], ys)
    tr (E, ns) _ = (root, ns)
    tr ((S (xs, ys) fails succs), ns) (x, n)
        | [x] `isPrefixOf` ys = if matched succs then (succs, ns++[n]) else (succs, ns)
        | otherwise = tr (fails, ns) (x, n)
```

www.dbooks.org

Figure 13.17 shows the first 4 steps when search 'anaym' in text 'anal'. Since the first 3 steps all succeed, so the left sub trees of these 3 states are not actually constructed. They are marked as '?'. In the fourth step, the match fails, thus the right sub tree needn't be built. On the other hand, we must construct the left sub tree, which is on top of the result of $trans(right(right(right(T))), n)$, where function $right(T)$ returns the right sub tree of T. This can be further expanded according to the definition of building and state transforming functions till we get the concrete state $((a, nanym), L, R)$. The detailed deduce process is left as exercise to the reader.

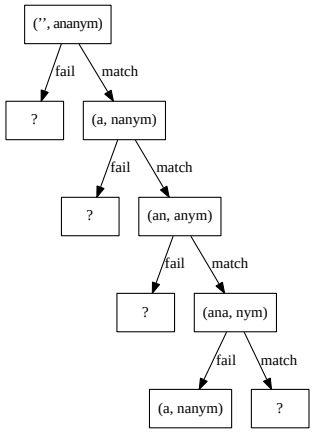


Figure 13.17: On demand construct the state transform tree when searching 'ananym' in text 'anal'.

This algorithm depends on the lazy evaluation critically. All the states to be transferred are built on demand. So that the building process is amortized $O(m)$, and the total performance is amortized $O(n + m)$. Readers can refer to [1] for detailed proof of it.

It's worth of comparing the final purely functional and the imperative algorithms. In many cases, we have expressive functional realization, however, for KMP matching algorithm, the imperative approach is much simpler and more intuitive. This is because we have to mimic the raw array by a infinite state transfer tree.

Boyer-Moore

Boyer-Moore string matching algorithm is another effective solution invited in 1977 [10]. The idea of Boyer-Moore algorithm comes from the following observation.

The bad character heuristics

When attempt to match the pattern, even if there are several characters from the left are same, it fails if the last one does not match, as shown in figure 13.18. What's more, we wouldn't find a match even if we slide the pattern down by 1, or 2. Actually, the length of the pattern 'ananym' is 6, the last character is

'm', however, the corresponding character in the text is 'h'. It does not appear in the pattern at all. We can directly slide the pattern down by 6.

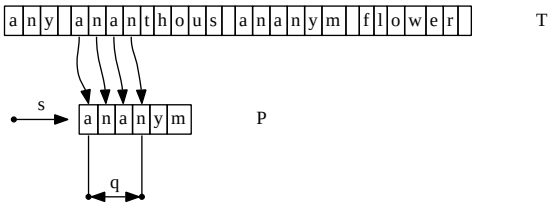
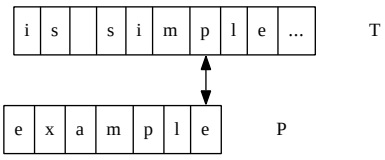
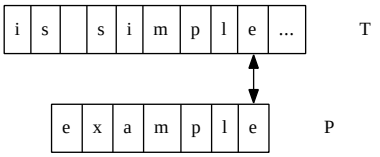


Figure 13.18: Since character 'h' doesn't appear in the pattern, we wouldn't find a match if we slide the pattern down less than the length of the pattern.

This leads to *the bad-character rule*. We can do a pre-processing for the pattern. If the character set of the text is already known, we can find all characters which don't appear in the pattern string. During the later scan process, as long as we find such a bad character, we can immediately slide the pattern down by its length. The question is what if the unmatched character does appear in the pattern? While, in order not to miss any potential matches, we have to slide down the pattern to check again. This is shown as in the figure 13.19



(a) The last character in the pattern 'e' doesn't match 'p'. However, 'p' appears in the pattern.



(b) We have to slide the pattern down by 2 to check again.

Figure 13.19: Slide the pattern if the unmatched character appears in the pattern.

It's quite possible that the unmatched character appears in the pattern more than one position. Denote the length of the pattern as $|P|$, the character appears in positions $p_1, p_2, ..., p_i$. In such case, we take the right most one to avoid missing any matches.

$$s = |P| - p_i \tag{13.46}$$

Note that the shifting length is 0 for the last position in the pattern according to the above equation. Thus we can skip it in realization. Another important

www.dbooks.org

point is that since the shifting length is calculated against the position aligned with the last character in the pattern string, (we deduce it from $|P|$), no matter where the mismatching happens when we scan from right to the left, we slide down the pattern string by looking up the bad character table with the one in the text aligned with the last character of the pattern. This is shown in figure 13.20.

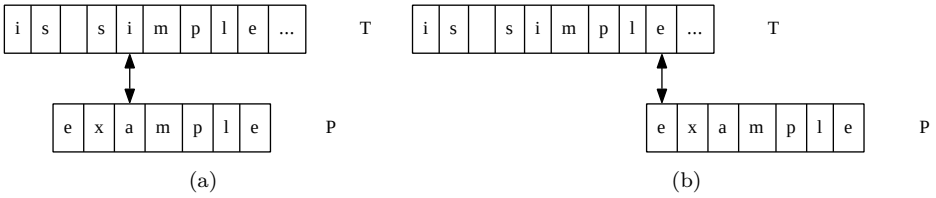


Figure 13.20: Even the mismatching happens in the middle, between char 'i' and 'a', we look up the shifting value with character 'e', which is 6 (calculated from the first 'e', the second 'e' is skipped to avoid zero shifting).

There is a good result in practice, that only using the bad-character rule leads to a simple and fast string matching algorithm, called Boyer-Moore-Horspool algorithm [11].

```
1: procedure BOYER-MOORE-HORSPOOL(T, P)
2:     for ∀c ∈ Σ do
3:         π[c] ← |P|
4:     for i ← 1 to |P| − 1 do                    ▷ Skip the last position
5:         π[P[i]] ← |P| − i
6:     s ← 0
7:     while s + |P| ≤ |T| do
8:         i ← |P|
9:         while i ≥ 1 ∧ P[i] = T[s + i] do          ▷ scan from right
10:            i ← i − 1
11:        if i < 1 then
12:            found one solution at s
13:            s ← s + 1                          ▷ go on finding the next
14:        else
15:            s ← s + π[T[s + |P|]]
```

The character set is denoted as Σ, we first initialize all the values of sliding table π as the length of the pattern string $|P|$. After that we process the pattern from left to right, update the sliding value. If a character appears multiple times in the pattern, the latter value, which is on the right hand, will overwrite the previous value. We start the matching scan process by aligning the pattern and the text string from the very left. However, for every alignment s, we scan from the right to the left until either there is unmatched character or all the characters in the pattern have been examined. The latter case indicates that we've found a match; while for the former case, we look up π to slide the pattern down to the right.

The following example Python code implements this algorithm accordingly.

```
def bmh_match(w, p):
```

```
n = len(w)
m = len(p)
tab = [m for _ in range(256)]  # table to hold the bad character rule.
for i in range(m-1):
    tab[ord(p[i])] = m - 1 - i
res = []
offset = 0
while offset + m <= n:
    i = m - 1
    while i >= 0 and p[i] == w[offset+i]:
        i = i - 1
    if i < 0:
        res.append(offset)
        offset = offset + 1
    else:
        offset = offset + tab[ord(w[offset + m - 1])]
return res
```

The algorithm firstly takes about $O(|\Sigma|+|P|)$ time to build the sliding table. If the character set size is small, the performance is dominated by the pattern and the text. There is definitely the worst case that all the characters in the pattern and text are same, e.g. searching 'aa...a' (m of 'a', denoted as a^m) in text 'aa.....a' (n of 'a', denoted as a^n). The performance in the worst case is $O(mn)$. This algorithm performs well if the pattern is long, and there are constant number of matching. The result is bound to linear time. This is as same as the best case of full Boyer-Moore algorithm which will be explained next.

The good suffix heuristics

Consider searching pattern 'abbabab' in text 'bbbababbabab...' like figure 13.21. By using the bad-character rule, the pattern will be slided by two.

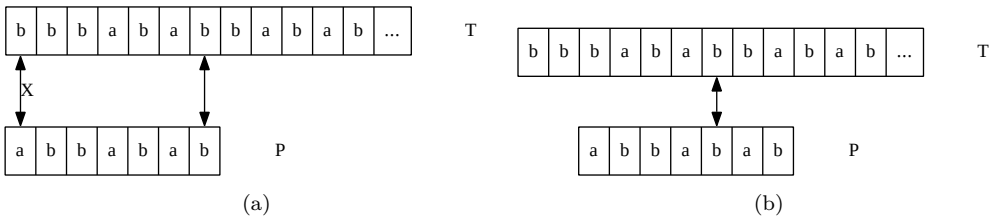


Figure 13.21: According to the bad-character rule, the pattern is slided by 2, so that the next 'b' is aligned.

Actually, we can do better than this. Observing that before the unmatched point, we have already successfully matched 6 characters 'bbabab' from right to the left. Since 'ab', which is the prefix of the pattern is also the suffix of what we matched so far, we can directly slide the pattern to align this suffix as shown in figure 13.22.

This is quite similar to the pre-processing of KMP algorithm, However, we can't always skip so many characters. Consider the following example as shown in figure 13.23. We have matched characters 'bab' when the unmatch happens.

www.dbooks.org

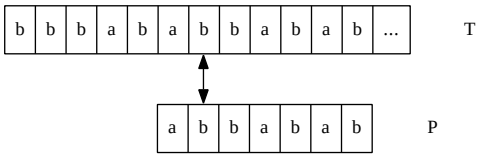


Figure 13.22: As the prefix 'ab' is also the suffix of what we've matched, we can slide down the pattern to a position so that 'ab' are aligned.

Although the prefix 'ab' of the pattern is also the suffix of 'bab', we can't slide the pattern so far. This is because 'bab' appears somewhere else, which starts from the 3rd character of the pattern. In order not to miss any potential matching, we can only slide the pattern by two.

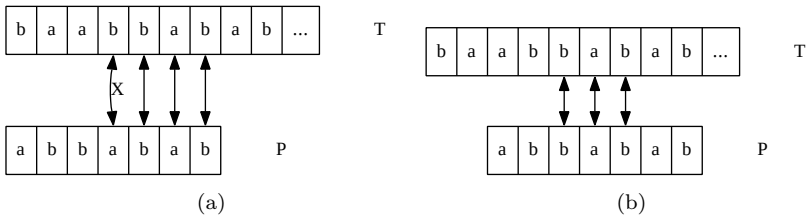


Figure 13.23: We've matched 'bab', which appears somewhere else in the pattern (from the 3rd to the 5th character). We can only slide down the pattern by 2 to avoid missing any potential matching.

The above situation forms the two cases of *the good-suffix rule*, as shown in figure 13.24.

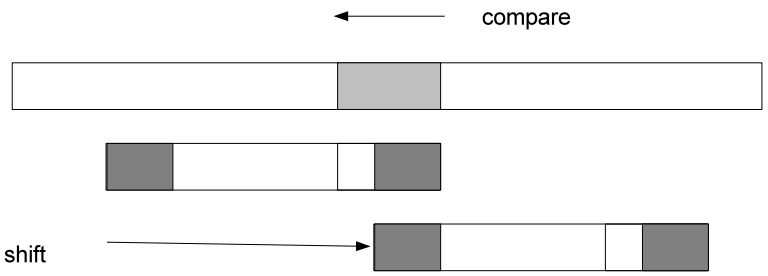
Both cases in good suffix rule handle the situation that there are multiple characters have been matched from right. We can slide the pattern to the right if any of the the following happens.

- Case 1 states that if a part of the matching suffix occurs as a prefix of the pattern, and the matching suffix doesn't appear in any other places in the pattern, we can slide the pattern to the right to make this prefix aligned;
- Case 2 states that if the matching suffix occurs some where else in the pattern, we can slide the pattern to make the right most occurrence aligned.

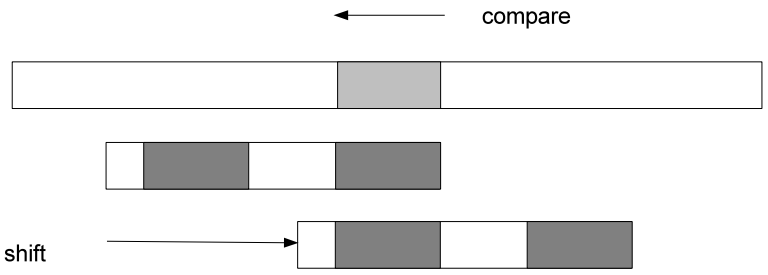
Note that in the scan process, we should apply case 2 first whenever it is possible, and then examine case 1 if the whole matched suffix does not appears in the pattern. Observe that both cases of the good-suffix rule only depend on the pattern string, a table can be built by pre-process the pattern for further looking up.

For the sake of brevity, we denote the suffix string from the i-th character of P as $\overline{P_i}$. That $\overline{P_i}$ is the sub-string $P[i]P[i+1]...P[m]$.

For case 1, we can check every suffix of P, which includes $\overline{P_m}$, $\overline{P_{m-1}}$, $\overline{P_{m-2}}$, ..., $\overline{P_2}$ to examine if it is the prefix of P. This can be achieved by a round of scan from right to the left.



(a) Case 1, Only a part of the matching suffix occurs as a prefix of the pattern.



(b) Case 2, The matching suffix occurs some where else in the pattern.

Figure 13.24: The light gray section in the text represents the characters have been matched; The dark gray parts indicate the same content in the pattern.

www.dbooks.org

For case 2, we can check every prefix of P includes P_1, P_2, ..., P_{m-1} to examine if the longest suffix is also a suffix of P. This can be achieved by another round of scan from left to the right.

```
1: function GOOD-SUFFIX(P)
2:     m ← |P|
3:     π_s ← {0, 0, ..., 0}                    ▷ Initialize the table of length m
4:     l ← 0                                   ▷ The last suffix which is also prefix of P
5:     for i ← m − 1 down-to 1 do              ▷ First loop for case 1
6:         if P̄_i ⊏ P then                     ▷ ⊏ means 'is prefix of'
7:             l ← i
8:         π_s[i] ← l
9:     for i ← 1 to m do                       ▷ Second loop for case 2
10:        s ← SUFFIX-LENGTH(P_i)
11:        if s ≠ 0 ∧ P[i − s] ≠ P[m − s] then
12:            π_s[m − s] ← m − i
13:    return π_s
```

This algorithm builds the good-suffix heuristics table π_s. It first checks every suffix of P from the shortest to the longest. If the suffix $\overline{P_i}$ is also the prefix of P, we record this suffix, and use it for all the entries until we find another suffix $\overline{P_j}$, $j < i$, and it is also the prefix of P.

After that, the algorithm checks every prefix of P from the shortest to the longest. It calls the function SUFFIX-LENGTH(P_i), to calculate the length of the longest suffix of P_i, which is also suffix of P. If this length s isn't zero, which means there exists a sub-string, that appears as the suffix of the pattern. It indicates that case 2 happens. The algorithm overwrites the s-th entry from the right of the table π_s. Note that to avoid finding the same occurrence of the matched suffix, we test if $P[i-s]$ and $P[m-s]$ are same.

Function SUFFIX-LENGTH is designed as the following.

```
1: function SUFFIX-LENGTH(P_i)
2:     m ← |P|
3:     j ← 0
4:     while P[m − j] = P[i − j] ∧ j < i do
5:         j ← j + 1
6:     return j
```

The following Python example program implements the good-suffix rule.

```
def good_suffix(p):
    m = len(p)
    tab = [0 for _ in range(m)]
    last = 0
    # first loop for case 1
    for i in range(m-1, 0, -1): # m-1, m-2, ∘ .., 1
        if is_prefix(p, i):
            last = i
        tab[i - 1] = last
    # second loop for case 2
    for i in range(m):
        slen = suffix_len(p, i)
        if slen != 0 and p[i - slen] != p[m - 1 - slen]:
            tab[m - 1 - slen] = m - 1 - i
```

```
    return tab

# test if p[i..m-1] `is prefix of` p
def is_prefix(p, i):
    for j in range(len(p) - i):
        if p[j] != p [i+j]:
            return False
    return True

# length of the longest suffix of p[..i], which is also a suffix of p
def suffix_len(p, i):
    m = len(p)
    j = 0
    while p[m - 1 - j] == p[i - j] and j < i:
        j = j + 1
    return j
```

It's quite possible that both the bad-character rule and the good-suffix rule can be applied when the unmatch happens. The Boyer-Moore algorithm compares and picks the bigger shift so that it can find the solution as quick as possible. The bad-character rule table can be explicitly built as below

1: **function** BAD-CHARACTER(P)
2: **for** $\forall c \in \Sigma$ **do**
3: $\pi_b[c] \leftarrow |P|$
4: **for** $i \leftarrow 1$ to $|P| - 1$ **do**
5: $\pi_b[P[i]] \leftarrow |P| - i$
6: **return** π_b

The following Python program implements the bad-character rule accordingly.

```
def bad_char(p):
    m = len(p)
    tab = [m for _ in range(256)]
    for i in range(m-1):
        tab[ord(p[i])] = m - 1 - i
    return tab
```

The final Boyer-Moore algorithm firstly builds the two rules from the pattern, then aligns the pattern to the beginning of the text and scans from right to the left for every alignment. If any unmatch happens, it tries both rules, and slides the pattern with the bigger shift.

1: **function** BOYER-MOORE(T, P)
2: $n \leftarrow |T|, m \leftarrow |P|$
3: $\pi_b \leftarrow$ BAD-CHARACTER(P)
4: $\pi_s \leftarrow$ GOOD-SUFFIX(P)
5: $s \leftarrow 0$
6: **while** $s + m \leq n$ **do**
7: $i \leftarrow m$
8: **while** $i \geq 1 \wedge P[i] = T[s + i]$ **do**
9: $i \leftarrow i - 1$
10: **if** $i < 1$ **then**
11: found one solution at s

www.dbooks.org

12: $s \leftarrow s + 1$ ▷ go on finding the next
13: **else**
14: $s \leftarrow s + max(\pi_b[T[s+m]], \pi_s[i])$

Here is the example implementation of Boyer-Moore algorithm in Python.

```
def bm_match(w, p):
    n = len(w)
    m = len(p)
    tab1 = bad_char(p)
    tab2 = good_suffix(p)
    res = []
    offset = 0
    while offset + m <= n:
        i = m - 1
        while i >= 0 and p[i] == w[offset + i]:
            i = i - 1
        if i < 0:
            res.append(offset)
            offset = offset + 1
        else:
            offset = offset + max(tab1[ord(w[offset + m - 1])], tab2[i])
    return res
```

The Boyer-Moore algorithm published in original paper is bound to $O(n+m)$ in worst case only if the pattern doesn't appear in the text [10]. Knuth, Morris, and Pratt proved this fact in 1977 [12]. However, when the pattern appears in the text, as we shown above, Boyer-Moore performs $O(nm)$ in the worst case.

Richard Birds shows a purely functional realization of Boyer-Moore algorithm in chapter 16 in [1]. We skipped it in this book.

Exercise 13.2

- Proof that Boyer-Moore majority vote algorithm is correct.
- Given a list, find the element occurs most. Are there any divide and conqueror solutions? Are there any divide and conqueror data structures, such as map can be used?
- How to find the elements occur more than 1/3 in a list? How to find the elements occur more than 1/m in the list?
- If we reject the empty array as valid sub-array, how to realize the maximum sum of sub-arrays puzzle?
- Bentley presents a divide and conquer algorithm to find the maximum sum in $O(n \log n)$ time in [4]. The idea is to split the list at the middle point. We can recursively find the maximum sum in the first half and second half; However, we also need to find maximum sum cross the middle point. The method is to scan from the middle point to both ends as the following.

  1: **function** MAX-SUM(A)
  2: **if** $A = \phi$ **then**
  3: **return** 0
  4: **else if** $|A| = 1$ **then**

```
5:          return MAX(0, A[1])
6:      else
7:          m ← ⌊|A|/2⌋
8:          a ← MAX-FROM(REVERSE(A[1...m]))
9:          b ← MAX-FROM(A[m + 1...|A|])
10:         c ← MAX-SUM(A[1...m])
11:         d ← MAX-SUM(A[m + 1...|A|)
12:         return MAX(a + b, c, d)

13: function MAX-FROM(A)
14:     sum ← 0, m ← 0
15:     for i ← 1 to |A| do
16:         sum ← sum + A[i]
17:         m ← MAX(m, sum)
18:     return m
```

It's easy to deduce the time performance is $T(n) = 2T(n/2) + O(n)$. Implement this algorithm in your favorite programming language.

- Given a $m \times n$ matrix contains positive and negative numbers, find the sub metrics with maximum sum of its elements.

- Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining. Figure 13.25 shows an example. For example, Given $\{0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1\}$,

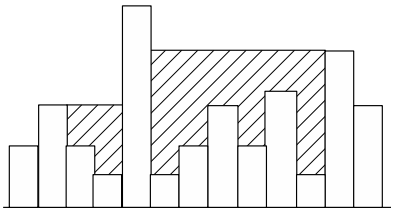


Figure 13.25: Shadowed areas are waters.

the result is 6.

- Explain why KMP algorithm perform in linear time even in the seemed 'worst' case.

- Implement the purely functional KMP algorithm by using reversed P_p to avoid the linear time appending operation.

- Deduce the state of the tree $left(right(right(right(T))))$ when searching 'ananym' in text 'anal'.

www.dbooks.org

13.3 Solution searching

One interesting thing that computer programming can offer is solving puzzles. In the early phase of classic artificial intelligent, people developed many methods to search for solutions. Different from the sequence searching and string matching, the solution doesn't obviously exist among a candidates set. It typically need construct the solution while trying varies of attempts. Some problems are solvable, while others are not. Among the solvable problems, not all of them just have one unique solution. For example, a maze may have multiple ways out. People sometimes need search for the best one.

13.3.1 DFS and BFS

DFS and BFS stand for deep-first search and breadth-first search. They are typically introduced as graph algorithms in textbooks. Graph is a comprehensive topic which is hard to be covered in this elementary book. In this section, we'll show how to use DFS and BFS to solve some real puzzles without formal introduction about the graph concept.

Maze

Maze is a classic and popular puzzle. Maze is amazing to both kids and adults. Figure 13.26 shows an example maze. There are also real maze gardens can be found in parks for fun. In the late 1990s, maze-solving games were quite often hold in robot mouse competition all over the world.

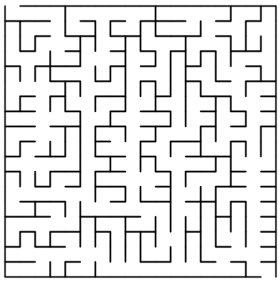


Figure 13.26: A maze

There are multiple methods to solve maze puzzle. We'll introduce an effective, yet not the best one in this section. There are some well known sayings about how to find the way out in maze, while not all of them are true.

For example, one method states that, wherever you have multiple ways, always turn right. This doesn't work as shown in figure 13.27. The obvious solution is first to go along the top horizontal line, then turn right, and keep going ahead at the 'T' section. However, if we always turn right, we'll endless loop around the inner big block.

This example tells us that the decision when there are multiple choices matters the solution. Like the fairy tale we read in our childhood, we can take some bread crumbs in a maze. When there are multiple ways, we can simply select one, left a piece of bread crumbs to mark this attempt. If we enter a died end,



Figure 13.27: It leads to loop way if always turns right.

we go back to the last place where we've made a decision by back-tracking the bread crumbs. Then we can alter to another way.

At any time, if we find there have been already bread crumbs left, it means we have entered a loop, we must go back and try different ways. Repeat these try-and-check steps, we can either find the way out, or give the 'no solution' fact. In the later case, we back-track to the start point.

One easy way to describe a maze, is by a $m \times n$ matrix, each element is either 0 or 1, which indicates if there is a way at this cell. The maze illustrated in figure 13.27 can be defined as the following matrix.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{matrix}$$

Given a start point $s = (i, j)$, and a goal $e = (p, q)$, we need find all solutions, that are the paths from s to e.

There is an obviously recursive exhaustive search method. That in order to find all paths from s to e, we can check all connected points to s, for every such point k, we recursively find all paths from k to e. This method can be illustrated as the following.

- Trivial case, if the start point s is as same as the target point e, we are done;
- Otherwise, for every connected point k to s, recursively find the paths from k to e; If e can be reached via k, put section s-k in front of each path between k and e.

However, we have to left 'bread crumbs' to avoid repeatedly trying the same attempts. This is because otherwise in the recursive case, we start from s, find a connected point k, then we further try to find paths from k to e. Since s is connected to k as well, so in the next recursion, we'll try to find paths from s to e again. It turns to be the very same origin problem, and we are trapped in infinite recursions.

www.dbooks.org

Our solution is to initialize an empty list, use it to record all the points we've visited so far. For every connected point, we look up the list to examine if it has already been visited. We skip all the visited candidates and only try those new ones. The corresponding algorithm can be defined like this.

$$solveMaze(m, s, e) = solve(s, \{\phi\}) \tag{13.47}$$

Where m is the matrix which defines a maze, s is the start point, and e is the end point. Function $solve$ is defined in the context of $solveMaze$, so that the maze and the end point can be accessed. It can be realized recursively like what we described above[8].

$$solve(s, P) = \begin{cases} \{\{s\} \cup p | p \in P\} & : \quad s = e \\ concat(\{ \quad solve(s', \{\{s\} \cup p | p \in P\}) | \\ \qquad s' \in adj(s), \neg visited(s')\}) & : \quad otherwise \end{cases} \tag{13.48}$$

Note that P also serves as an accumulator. Every connected point is recorded in all the possible paths to the current position. But they are stored in reversed order, that is the newly visited point is put to the head of all the lists, and the starting point is the last one. This is because the appending operation is linear ($O(n)$, where n is the number of elements stored in a list), while linking to the head is just constant time. We can output the result in correct order by reversing all possible solutions in equation (13.47)[9]:

$$solveMaze(m, s, e) = map(reverse, solve(s, \{\phi\})) \tag{13.49}$$

We need define functions $adj(p)$ and $visited(p)$, which finds all the connected points to p, and tests if point p has been visited respectively. Two points are connected if and only if they are next cells horizontally or vertically in the maze matrix, and both have zero value.

$$adj((x, y)) = \{(x', y') | \quad (x', y') \in \{(x-1, y), (x+1, y), (x, y-1), (x, y+1)\}, \\ 1 \leq x' \leq M, 1 \leq y' \leq N, m_{x'y'} = 0\} \tag{13.50}$$

Where M and N are the widths and heights of the maze.

Function $visited(p)$ examines if point p has been recorded in any lists in P.

$$visited(p) = \exists path \in P, p \in path \tag{13.51}$$

The following Haskell example code implements this algorithm.

```
solveMaze m from to = map reverse $ solve from [[]] where
    solve p paths | p == to = map (p:) paths
                  | otherwise = concat [solve p' (map (p:) paths) |
                                        p' ← adjacent p,
                                        not $ visited p' paths]
    adjacent (x, y) = [(x', y') |
                       (x', y') ← [(x-1, y), (x+1, y), (x, y-1), (x, y+1)],
```

[8] Function *concat* can flatten a list of lists. For example. $concat(\{\{a, b, c\}, \{x, y, z\}\}) = \{a, b, c, x, y, z\}$. Refer to appendix A for detail.

[9] the detailed definition of *reverse* can be found in the appendix A.

```
                inRange (bounds m) (x', y'),
                m ! (x', y') == 0]
    visited p paths = any (p `elem`) paths
```

For a maze defined as matrix like below example, all the solutions can be given by this program.

```
mz = [[0, 0, 1, 0, 1, 1],
      [1, 0, 1, 0, 1, 1],
      [1, 0, 0, 0, 0, 0],
      [1, 1, 0, 1, 1, 1],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 1, 1, 0]]

maze  = listArray ((1,1), (6, 6)) ∘ concat

solveMaze (maze mz) (1,1) (6, 6)
```

As we mentioned, this is a style of 'exhaustive search'. It recursively searches all the connected points as candidates. In a real maze solving game, a robot mouse competition for instance, it's enough to just find a route. We can adapt to a method close to what described at the beginning of this section. The robot mouse always tries the first connected point, and skip the others until it gets stuck. We need some data structure to store the 'bread crumbs', which help to remember the decisions being made. As we always attempt to find the way on top of the latest decision, it is the last-in, first-out manner. A stack can be used to realize it.

At the very beginning, only the starting point s is stored in the stack. we pop it out, find, for example, points a, and b, are connected to s. We push the two possible paths: $\{a, s\}$ and $\{b, s\}$ to the stack. Next we pop $\{a, s\}$ out, and examine connected points to a. Then all the paths with 3 steps will be pushed back. We repeat this process. At anytime, each element stored in the stack is a path, from the starting point to the farthest place can arrive in the reversed order. This can be illustrated in figure 13.28.

The stack can be realized with a list. The latest option is picked from the head, and the new candidates are also added to the head. The maze puzzle can be solved by using such a list of paths:

$$solveMaze'(m, s, e) = reverse(solve'(\{\{s\}\})) \tag{13.52}$$

As we are searching the first, but not all the solutions, *map* isn't used here. When the stack is empty, it means that we've tried all the options and failed to find a way out. There is no solution; otherwise, the top option is popped, expanded with all the adjacent points which haven't been visited before, and pushed back to the stack. Denote the stack as S, if it isn't empty, the top element is s_1, and the new stack after the top being popped as S'. s_1 is a list of points represents path P. Denote the first point in this path as p_1, and the

www.dbooks.org

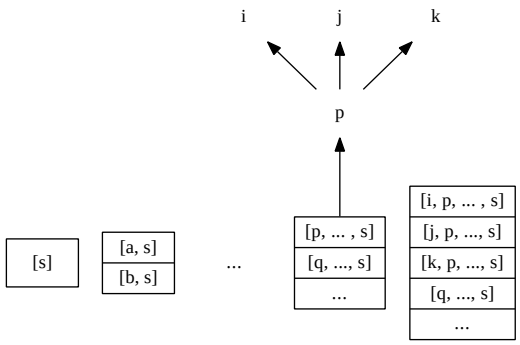


Figure 13.28: The stack is initialized with a singleton list of the starting point s. s is connected with point a and b. Paths $\{a, s\}$ and $\{b, s\}$ are pushed back. In some step, the path ended with point p is popped. p is connected with points i, j, and k. These 3 points are expanded as different options and pushed back to the stack. The candidate path ended with q won't be examined unless all the options above fail.

rest as P'. The solution can be formalized as the following.

$$solve'(S) = \begin{cases} \phi & : & S = \phi \\ s_1 & : & s_1 = e \\ solve'(S') & : & C = \{c | c \in adj(p_1), c \notin P'\} = \phi \\ solve'(\{\{p\} \cup P | p \in C\} \cup S) & : & otherwise, C \neq \phi \end{cases} \tag{13.53}$$

Where the *adj* function is defined above. This updated maze solution can be implemented with the below example Haskell program [10].

```
dfsSolve m from to = reverse $ solve [[from]] where
    solve [] = []
    solve (c@(p:path):cs)
        | p == to = c -- stop at the first solution
        | otherwise = let os = filter (`notElem` path) (adjacent p) in
                      if os == []
                      then solve cs
                      else solve ((map (:c) os) ++ cs)
```

It's quite easy to modify this algorithm to find all solutions. When we find a path in the second clause, instead of returning it immediately, we record it and go on checking the rest memorized options in the stack till until the stack becomes empty. We left it as an exercise to the reader.

The same idea can also be realized imperatively. We maintain a stack to store all possible paths from the starting point. In each iteration, the top option path is popped, if the farthest position is the end point, a solution is found; otherwise, all the adjacent, not visited yet points are appended as new paths and pushed back to the stack. This is repeated till all the candidate paths in the stacks are checked.

[10] The same code of *adjacent* function is skipped

We use the same notation to represent the stack S. But the paths will be stored as arrays instead of list in imperative settings as the former is more effective. Because of this the starting point is the first element in the path array, while the farthest reached place is the right most element. We use p_n to represent $\textsc{Last}(P)$ for path P. The imperative algorithm can be given as below.

```
1:  function SOLVE-MAZE(m, s, e)
2:      S ← φ
3:      PUSH(S, {s})
4:      L ← φ                                          ▷ the result list
5:      while S ≠ φ do
6:          P ← POP(S)
7:          if e = p_n then
8:              ADD(L, P)
9:          else
10:             for ∀p ∈ ADJACENT(m, p_n) do
11:                 if p ∉ P then
12:                     PUSH(S, P ∪ {p})
13:     return L
```

The following example Python program implements this maze solving algorithm.

```
def solve(m, src, dst):
    stack = [[src]]
    s = []
    while stack != []:
        path = stack.pop()
        if path[-1] == dst:
            s.append(path)
        else:
            for p in adjacent(m, path[-1]):
                if not p in path:
                    stack.append(path + [p])
    return s

def adjacent(m, p):
    (x, y) = p
    ds = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    ps = []
    for (dx, dy) in ds:
        x1 = x + dx
        y1 = y + dy
        if 0 <= x1 and x1 < len(m[0]) and
           0 <= y1 and y1 < len(m) and m[y][x] == 0:
            ps.append((x1, y1))
    return ps
```

And the same maze example given above can be solved by this program like the following.

```
mz = [[0, 0, 1, 0, 1, 1],
      [1, 0, 1, 0, 1, 1],
      [1, 0, 0, 0, 0, 0],
```

www.dbooks.org

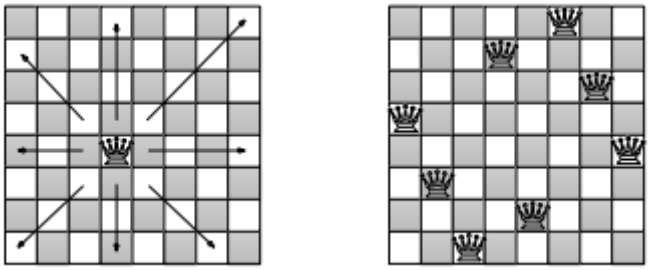
```
      [1, 1, 0, 1, 1, 1],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 1, 1, 0]]

solve(mz, (0, 0), (5,5))
```

It seems that in the worst case, there are 4 options (up, down, left, and right) at each step, each option is pushed to the stack and eventually examined during backtracking. Thus the complexity is bound to $O(4^n)$. The actual time won't be so large because we filtered out the places which have been visited before. In the worst case, all the reachable points are visited exactly once. So the time is bound to $O(n)$, where n is the number of points connected in total. As a stack is used to store candidate solutions, the space complexity is $O(n^2)$.

Eight queens puzzle

The eight queens puzzle is also a famous problem. Although cheese has very long history, this puzzle was first published in 1848 by Max Bezzel[13]. Queen in the cheese game is quite powerful. It can attack any other pieces in the same row, column and diagonal at any distance. The puzzle is to find a solution to put 8 queens in the board, so that none of them attack each other. Figure 13.29 (a) illustrates the places can be attacked by a queen and 13.29 (b) shows a solution of 8 queens puzzle.



(a) A queen piece. (b) An example solution

Figure 13.29: The eight queens puzzle.

It's obviously that the puzzle can be solved by brute-force, which takes P_{64}^8 times. This number is about 4×10^{10}. It can be easily improved by observing that, no two queens can be in the same row, and each queen must be put on one column between 1 to 8. Thus we can represent the arrangement as a permutation of $\{1, 2, 3, 4, 5, 6, 7, 8\}$. For instance, the arrangement $\{6, 2, 7, 1, 3, 5, 8, 4\}$ means, we put the first queen at row 1, column 6, the second queen at row 2 column 2, ..., and the last queen at row 8, column 4. By this means, we need only examine $8! = 40320$ possibilities.

We can find better solutions than this. Similar to the maze puzzle, we put queens one by one from the first row. For the first queen, there are 8 options, that we can put it at one of the eight columns. Then for the next queen, we again examine the 8 candidate columns. Some of them are not valid because those positions will be attacked by the first queen. We repeat this process, for

the i-th queen, we examine the 8 columns in row i, find which columns are safe. If none column is valid, it means all the columns in this row will be attacked by some queen we've previously arranged, we have to backtrack as what we did in the maze puzzle. When all the 8 queens are successfully put to the board, we find a solution. In order to find all the possible solutions, we need record it and go on to examine other candidate columns and perform back tracking if necessary. This process terminates when all the columns in the first row have been examined. The below equation starts the search.

$$solve(\{\phi\}, \phi) \tag{13.54}$$

In order to manage the candidate attempts, a stack S is used as same as in the maze puzzle. The stack is initialized with one empty element. And a list L is used to record all possible solutions. Denote the top element in the stack as s_1. It's actually an intermediate state of assignment, which is a partial permutation of 1 to 8. after pops s_1, the stack becomes S'. The $solve$ function can be defined as the following.

$$solve(S, L) = \begin{cases} L & : & S = \phi \\ solve(S', \{s_1\} \cup L) & : & |s_1| = 8 \\ solve(\left\{ \begin{array}{ll} \{i\} \cup s_1 | & i \in [1, 8], \\ & i \notin s_1, \\ & safe(i, s_1) \end{array} \right\} \cup S', L) & : & otherwise \end{cases} \tag{13.55}$$

If the stack is empty, all the possible candidates have been examined, it's not possible to backtrack any more. L has been accumulated all found solutions and returned as the result; Otherwise, if the length of the top element in the stack is 8, a valid solution is found. We add it to L, and go on finding other solutions; If the length is less than 8, we need try to put the next queen. Among all the columns from 1 to 8, we pick those not already occupied by previous queens (through the $i \notin s_1$ clause), and must not be attacked in diagonal direction (through the $safe$ predication). The valid assignments will be pushed to the stack for the further searching.

Function $safe(x, C)$ detects if the assignment of a queen in position x will be attacked by other queens in C in diagonal direction. There are 2 possible cases, 45° and 135° directions. Since the row of this new queen is $y = 1 + |C|$, where $|C|$ is the length of C, the $safe$ function can be defined as the following.

$$safe(x, C) = \forall (c, r) \in zip(reverse(C), \{1, 2, ...\}), |x - c| \neq |y - r| \tag{13.56}$$

Where zip takes two lists, and pairs every elements in them to a new list. Thus If $C = \{c_{i-1}, c_{i-2}, ..., c_2, c_1\}$ represents the column of the first $i - 1$ queens has been assigned, the above function will check list of pairs $\{(c_1, 1), (c_2, 2), ..., (c_{i-1}, i - 1)\}$ with position (x, y) forms any diagonal lines.

Translating this algorithm into Haskell gives the below example program.

```
solve = dfsSolve [[]] [] where
    dfsSolve [] s = s
    dfsSolve (c:cs) s
             | length c == 8 = dfsSolve cs (c:s)
```

www.dbooks.org

```
          | otherwise = dfsSolve ([(x:c) | x ← [1..8] \\ c,
                                  not $ attack x c] ++ cs) s
attack x cs = let y = 1 + length cs in
              any (λ(c, r) → abs(x - c) == abs(y - r)) $
                  zip (reverse cs) [1..]
```

Observing that the algorithm is tail recursive, it's easy to transform it into imperative realization. Instead of using list, we use array to represent queens assignment. Denote the stack as S, and the possible solutions as A. The imperative algorithm can be described as the following.

```
1: function SOLVE-QUEENS
2:     S ← {φ}
3:     L ← φ                                          ▷ The result list
4:     while S ≠ φ do
5:         A ← POP(S)                                 ▷ A is an intermediate assignment
6:         if |A| = 8 then
7:             ADD(L, A)
8:         else
9:             for i ← 1 to 8 do
10:                if VALID(i, A) then
11:                    PUSH(S, A ∪ {i})
12:    return L
```

The stack is initialized with the empty assignment. The main process repeatedly pops the top candidate from the stack. If there are still queens left, the algorithm examines possible columns in the next row from 1 to 8. If a column is safe, that it won't be attacked by any previous queens, this column will be appended to the assignment, and pushed back to the stack. Different from the functional approach, since array, but not list, is used, we needn't reverse the solution assignment any more.

Function VALID checks if column x is safe with previous queens put in A. It filters out the columns have already been occupied, and calculates if any diagonal lines are formed with existing queens.

```
1: function VALID(x, A)
2:     y ← 1 + |A|
3:     for i ← 1 to |A| do
4:         if x = A[i] ∨ |y − i| = |x − A[i]| then
5:             return False
6:     return True
```

The following Python example program implements this imperative algorithm.

```
def solve():
    stack = [[]]
    s = []
    while stack != []:
        a = stack.pop()
        if len(a) == 8:
            s.append(a)
        else:
            for i in range(1, 9):
                if valid(i, a):
```

```
                stack.append(a+[i])
    return s

def valid(x, a):
    y = len(a) + 1
    for i in range(1, y):
        if x == a[i-1] or abs(y - i) == abs(x - a[i-1]):
            return False
    return True
```

Although there are 8 optional columns for each queen, not all of them are valid and thus further expanded. Only those columns haven't been occupied by previous queens are tried. The algorithm only examines 15720, which is far less than $8^8 = 16777216$, possibilities [13].

It's quite easy to extend the algorithm, so that it can solve n queens puzzle, where $n \geq 4$. However, the time cost increases fast. The backtrack algorithm is just slightly better than the one permuting the sequence of 1 to 8 (which is bound to $o(n!)$). Another extension to the algorithm is based on the fact that the chess board is square, which is symmetric both vertically and horizontally. Thus a solution can generate other solutions by rotating and flipping. These aspects are left as exercises to the reader.

Peg puzzle

I once received a puzzle of the leap frogs. It said to be homework for 2nd grade student in China. As illustrated in figure 13.30, there are 6 frogs in 7 stones. Each frog can either hop to the next stone if it is not occupied, or leap over one frog to another empty stone. The frogs on the left side can only move to the right, while the ones on the right side can only move to the left. These rules are described in figure 13.31

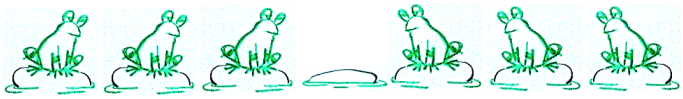


Figure 13.30: The leap frogs puzzle.

The goal of this puzzle is to arrange the frogs to jump according to the rules, so that the positions of the 3 frogs on the left are finally exchange with the ones on the right. If we denote the frog on the left as 'A', on the right as 'B', and the empty stone as 'O'. The puzzle is to find a solution to transform from 'AAAOBBB' to 'BBBOAAA'.

This puzzle is just a special form of the peg puzzles. The number of pegs is not limited to 6. it can be 8 or other bigger even numbers. Figure 13.32 shows some variants.

We can solve this puzzle by programing. The idea is similar to the 8 queens puzzle. Denote the positions from the left most stone as 1, 2, ..., 7. In ideal

www.dbooks.org



(a) Jump to the next stone (b) Jump over to the right (c) Jump over to the left

Figure 13.31: Moving rules.



(a) Solitaire (b) Hop over (c) Draught board

Figure 13.32: Variants of the peg puzzles from http://home.comcast.net/ stegmann/jumping.htm

cases, there are 4 options to arrange the move. For example when start, the frog on 3rd stone can hop right to the empty stone; symmetrically, the frog on the 5th stone can hop left; Alternatively, the frog on the 2nd stone can leap right, while the frog on the 6th stone can leap left.

We can record the state and try one of these 4 options at every step. Of course not all of them are possible at any time. If get stuck, we can backtrack and try other options.

As we restrict the left side frogs only moving to the right, and the right frogs only moving to the left, the moves are not reversible. There won't be any repetition cases as what we have to deal with in the maze puzzle. However, we still need record the steps in order to print them out finally.

In order to enforce these restriction, let A, O, B in representation 'AAAOBBB' be -1, 0, and 1 respectively. A state L is a list of elements, each element is one of these 3 values. It starts from $\{-1, -1, -1, 0, 1, 1, 1\}$. $L[i]$ access the i-th element, its value indicates if the i-th stone is empty, occupied by a frog from left side, or occupied by a frog from right side. Denote the position of the vacant stone as p. The 4 moving options can be stated as below.

- Leap left: $p < 6$ and $L[p+2] > 0$, swap $L[p] \leftrightarrow L[p+2]$;
- Hop left: $p < 7$ and $L[p+1] > 0$, swap $L[p] \leftrightarrow L[p+1]$;
- Leap right: $p > 2$ and $L[p-2] < 0$, swap $L[p-2] \leftrightarrow L[p]$;
- Hop right: $p > 1$ and $L[p-1] < 0$, swap $L[p-1] \leftrightarrow L[p]$.

Four functions $leap_l(L), hop_l(L), leap_r(L)$ and $hop_r(L)$ are defined accordingly. If the state L does not satisfy the move restriction, these function return L unchanged, otherwise, the changed state L' is returned accordingly.

We can also explicitly maintain a stack S to the attempts as well as the historic movements. The stack is initialized with a singleton list of starting state. The solution is accumulated to a list M, which is empty at the beginning:

$$solve(\{\{-1,-1,-1,0,1,1,1\}\},\phi) \tag{13.57}$$

As far as the stack isn't empty, we pop one intermediate attempt. If the latest state is equal to $\{1,1,1,0,-1,-1,-1\}$, a solution is found. We append the series of moves till this state to the result list M; otherwise, We expand to next possible state by trying all four possible moves, and push them back to the stack for further search. Denote the top element in the stack S as s_1, and the latest state in s_1 as L. The algorithm can be defined as the following.

$$solve(S,M) = \begin{cases} M & : & S = \phi \\ solve(S', \{reverse(s_1)\} \cup M) & : & L = \{1,1,1,0,-1,-1,-1\} \\ solve(P \cup S', M) & : & otherwise \end{cases} \tag{13.58}$$

Where P are possible moves from the latest state L:

$$P = \{L' | L' \in \{leap_l(L), hop_l(L), leap_r(L), hop_r(L)\}, L \neq L'\}$$

Note that the starting state is stored as the last element, while the final state is the first. That is the reason why we reverse it when adding to solution list.

Translating this algorithm to Haskell gives the following example program.

```
solve = dfsSolve [[[-1, -1, -1, 0, 1, 1, 1]]] [] where
    dfsSolve [] s = s
    dfsSolve (c:cs) s
             | head c == [1, 1, 1, 0, -1, -1, -1] = dfsSolve cs (reverse c:s)
             | otherwise = dfsSolve ((map (:c) $ moves $ head c) ++ cs) s

moves s = filter (/=s) [leapLeft s, hopLeft s, leapRight s, hopRight s] where
    leapLeft [] = []
    leapLeft (0:y:1:ys) = 1:y:0:ys
    leapLeft (y:ys) = y:leapLeft ys
    hopLeft [] = []
    hopLeft (0:1:ys) = 1:0:ys
    hopLeft (y:ys) = y:hopLeft ys
    leapRight [] = []
    leapRight (-1:y:0:ys) = 0:y:(-1):ys
    leapRight (y:ys) = y:leapRight ys
    hopRight [] = []
    hopRight (-1:0:ys) = 0:(-1):ys
    hopRight (y:ys) = y:hopRight ys
```

Running this program finds 2 symmetric solutions, each takes 15 steps. One solution is list in the below table.

www.dbooks.org

step	-1	-1	-1	0	1	1	1
1	-1	-1	0	-1	1	1	1
2	-1	-1	1	-1	0	1	1
3	-1	-1	1	-1	1	0	1
4	-1	-1	1	0	1	-1	1
5	-1	0	1	-1	1	-1	1
6	0	-1	1	-1	1	-1	1
7	1	-1	0	-1	1	-1	1
8	1	-1	1	-1	0	-1	1
9	1	-1	1	-1	1	-1	0
10	1	-1	1	-1	1	0	-1
11	1	-1	1	0	1	-1	-1
12	1	0	1	-1	1	-1	-1
13	1	1	0	-1	1	-1	-1
14	1	1	1	-1	0	-1	-1
15	1	1	1	0	-1	-1	-1

Observe that the algorithm is in tail recursive manner, it can also be realized imperatively. The algorithm can be more generalized, so that it solve the puzzles of n frogs on each side. We represent the start state {-1, -1, ..., -1, 0, 1, 1, ..., 1} as s, and the mirrored end state as e.

```
1: function SOLVE(s, e)
2:     S ← {{s}}
3:     M ← φ
4:     while S ≠ φ do
5:         s_1 ← POP(S)
6:         if s_1[1] = e then
7:             ADD(M, REVERSE(s_1))
8:         else
9:             for ∀m ∈ MOVES(s_1[1]) do
10:                PUSH(S, {m} ∪ s_1)
11:    return M
```

The possible moves can be also generalized with procedure MOVES to handle arbitrary number of frogs. The following Python program implements this solution.

```
def solve(start, end):
    stack = [[start]]
    s = []
    while stack != []:
        c = stack.pop()
        if c[0] == end:
            s.append(reversed(c))
        else:
            for m in moves(c[0]):
                stack.append([m]+c)
    return s

def moves(s):
    ms = []
    n = len(s)
    p = s.index(0)
```

```
    if p < n - 2 and s[p+2] > 0:
        ms.append(swap(s, p, p+2))
    if p < n - 1 and s[p+1] > 0:
        ms.append(swap(s, p, p+1))
    if p > 1 and s[p-2] < 0:
        ms.append(swap(s, p, p-2))
    if p > 0 and s[p-1] < 0:
        ms.append(swap(s, p, p-1))
    return ms

def swap(s, i, j):
    a = s[:]
    (a[i], a[j]) = (a[j], a[i])
    return a
```

For 3 frogs in each side, we know that it takes 15 steps to exchange them. It's interesting to examine the table that how many steps are needed along with the number of frogs in each side. Our program gives the following result.

number of frogs	1	2	3	4	5	...
number of steps	3	8	15	24	35	...

It seems that the number of steps are all square numbers minus one. It's natural to guess that the number of steps for n frogs in one side is $(n+1)^2 - 1$. Actually we can prove it is true.

Compare to the final state and the start state, each frog moves ahead $n+1$ stones in its opposite direction. Thus total $2n$ frogs move $2n(n+1)$ stones. Another important fact is that each frog on the left has to meet every one on the right one time. And leap will happen when meets. Since the frog moves two stones ahead by leap, and there are total n^2 meets happen, so that all these meets cause moving $2n^2$ stones ahead. The rest moves are not leap, but hop. The number of hops are $2n(n+1) - 2n^2 = 2n$. Sum up all n^2 leaps and $2n$ hops, the total number of steps are $n^2 + 2n = (n+1)^2 - 1$.

Summary of DFS

Observe the above three puzzles, although they vary in many aspects, their solutions show quite similar common structures. They all have some starting state. The maze starts from the entrance point; The 8 queens puzzle starts from the empty board; The leap frogs start from the state of 'AAAOBBB'. The solution is a kind of searching, at each attempt, there are several possible ways. For the maze puzzle, there are four different directions to try; For the 8 queens puzzle, there are eight columns to choose; For the leap frogs puzzle, there are four movements of leap or hop. We don't know how far we can go when make a decision, although the final state is clear. For the maze, it's the exit point; For the 8 queens puzzle, we are done when all the 8 queens being assigned on the board; For the leap frogs puzzle, the final state is that all frogs exchanged.

We use a common approach to solve them. We repeatedly select one possible candidate to try, record where we've achieved; If we get stuck, we backtrack and try other options. We are sure by using this strategy, we can either find a solution, or tell that the problem is unsolvable.

Of course there can be some variation, that we can stop when find one answer, or go on searching all the solutions.

www.dbooks.org

If we draw a tree rooted at the starting state, expand it so that every branch stands for a different attempt, our searching process is in a manner, that it searches deeper and deeper. We won't consider any other options in the same depth unless the searching fails so that we've to backtrack to upper level of the tree. Figure 13.33 illustrates the order we search a state tree. The arrow indicates how we go down and backtrack up. The number of the nodes shows the order we visit them.

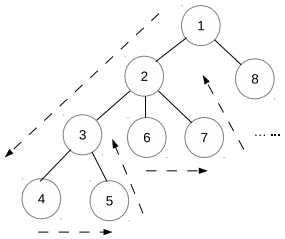


Figure 13.33: Example of DFS search order.

This kind of search strategy is called 'DFS' (Deep-first-search). We widely use it unintentionally. Some programming environments, Prolog for instance, adopt DFS as the default evaluation model. A maze is given by a set of rule base, such as:

```
c(a, b). c(a, e).
c(b, c). c(b, f).
c(e, d), c(e, f).
c(f, c).
c(g, d). c(g, h).
c(h, f).
```

Where predicate $c(X, Y)$ means place X is connected with Y. Note that this is a directed predicate, we can make Y to be connected with X as well by either adding a symmetric rule, or create a undirected predicate. Figure 13.34 shows such a directed graph. Given two places X and Y, Prolog can tell if they are connected by the following program.

```
go(X, X).
go(X, Y) :- c(X, Z), go(Z, Y)
```

This program says that, a place is connected with itself. Given two different places X and Y, if X is connected with Z, and Z is connected with Y, then X is connected with Y. Note that, there might be multiple choices for Z. Prolog selects a candidate, and go on further searching. It only tries other candidates if the recursive searching fails. In that case, Prolog backtracks and tries other alternatives. This is exactly what DFS does.

DFS is quite straightforward when we only need a solution, but don't care if the solution takes the fewest steps. For example, the solution it gives, may not be the shortest path for the maze. We'll see some more puzzles next. They demands to find the solution with the minimum attempts.

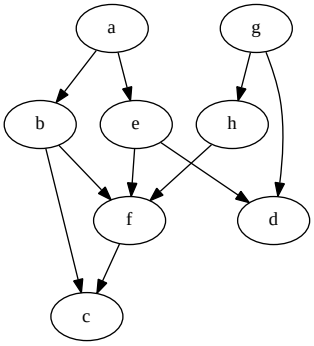


Figure 13.34: A directed graph.

The wolf, goat, and cabbage puzzle

This puzzle says that a farmer wants to cross a river with a wolf, a goat, and a bucket of cabbage. There is a boat. Only the farmer can drive it. But the boat is small. it can only hold one of the wolf, the goat, and the bucket of cabbage with the farmer at a time. The farmer has to pick them one by one to the other side of the river. However, the wolf would eat the goat, and the goat would eat the cabbage if the farmer is absent. The puzzle asks to find the fast solution so that they can all safely go cross the river.



Figure 13.35: The wolf, goat, cabbage puzzle

The key point to this puzzle is that the wolf does not eat the cabbage. The farmer can safely pick the goat to the other side. But next time, no matter if he pick the wolf or the cabbage to cross the river, he has to take one back to avoid the conflict. In order to find the fast the solution, at any time, if the farmer has multiple options, we can examine all of them in parallel, so that these different decisions compete. If we count the number of the times the farmer cross the river without considering the direction, that crossing the river back and forth means 2 times, we are actually checking the complete possibilities after 1 time,

www.dbooks.org

2 times, 3 times, ... When we find a situation, that they all arrive at the other bank, we are done. And this solution wins the competition, which is the fast solution.

The problem is that we can't examine all the possible solutions in parallel ideally. Even with a super computer equipped with many CPU cores, the setup is too expensive to solve such a simple puzzle.

Let's consider a lucky draw game. People blindly pick from a box with colored balls. There is only one black ball, all the others are white. The one who pick the black ball wins the game; Otherwise, he must return the ball to the box and wait for the next chance. In order to be fair enough, we can setup a rule that no one can try the second time before all others have tried. We can line people to a queue. Every time the first guy pick a ball, if he does not win, he then stands at the tail of the queue to wait for the second try. This queue helps to ensure our rule.

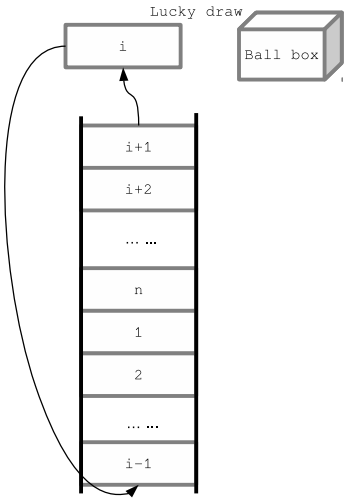


Figure 13.36: A lucky-draw game, the i-th person goes from the queue, pick a ball, then join the queue at tail if he fails to pick the black ball.

We can use the quite same idea to solve our puzzle. The two banks of the river can be represented as two sets A and B. A contains the wolf, the goat, the cabbage and the farmer; while B is empty. We take an element along with the farmer from one set to the other each time. The two sets can't hold conflict things if the farmer is absent. The goal is to exchange the contents of A and B with fewest steps.

We initialize a queue with state $A = \{w, g, c, p\}, B = \phi$ as the only element. As far as the queue isn't empty, we pick the first element from the head, expand it with all possible options, and put these new expanded candidates to the tail of the queue. If the first element on the head is the final goal, that $A = \phi, B = \{w, g, c, p\}$, we are done. Figure 13.37 illustrates the idea of this search order.

Note that as all possibilities in the same level are examined, there is no need for back-tracking.

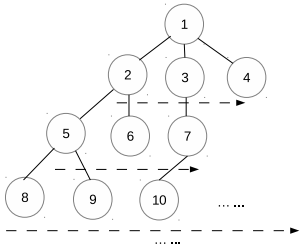


Figure 13.37: Start from state 1, check all possible options 2, 3, and 4 for next step; then all nodes in level 3, ...

There is a simple way to treat the set. A four bits binary number can be used, each bit stands for a thing, for example, the wolf $w = 1$, the goat $g = 2$, the cabbage $c = 4$, and the farmer $p = 8$. That 0 stands for the empty set, 15 stands for a full set. Value 3, solely means there are a wolf and a goat on the river bank. In this case, the wolf will eat the goat. Similarly, value 6 stands for another conflicting case. Every time, we move the highest bit (which is 8), or together with one of the other bits (4 or 2, or 1) from one number to the other. The possible moves can be defined as below.

$$mv(A, B) = \begin{cases} \{(A - 8 - i, B + 8 + i) | i \in \{0, 1, 2, 4\}, i = 0 \vee A \overline{\wedge} i \neq 0\} & : \quad B < 8 \\ \{(A + 8 + i, B - 8 - i) | i \in \{0, 1, 2, 4\}, i = 0 \vee B \overline{\wedge} i \neq 0\} & : \quad Otherwise \end{cases} \tag{13.59}$$

Where $\overline{\wedge}$ is the bitwise-and operation.

the solution can be given by reusing the queue defined in previous chapter. Denote the queue as Q, which is initialed with a singleton list $\{(15, 0)\}$. If Q is not empty, function $DeQ(Q)$ extracts the head element M, the updated queue becomes Q'. M is a list of pairs, stands for a series of movements between the river banks. The first element in $m_1 = (A', B')$ is the latest state. Function $EnQ'(Q, L)$ is a slightly different enqueue operation. It pushes all the possible moving sequences in L to the tail of the queue one by one and returns the updated queue. With these notations, the solution function is defined like below.

$$solve(Q) = \begin{cases} \phi & : \quad Q = \phi \\ reverse(M) & : \quad A' = 0 \\ solve(EnQ'(Q', \left\{ \{m\} \cup M | \begin{array}{l} m \in mv(m_1), \\ valid(m, M) \end{array} \right\})) & : \quad otherwise \end{cases} \tag{13.60}$$

Where function $valid(m, M)$ checks if the new moving candidate $m = (A'', B'')$ is valid. That neither A'' nor B'' is 3 or 6, and m hasn't been tried before in M to avoid any repeatedly attempts.

$$valid(m, M) = A'' \neq 3, A'' \neq 6, B'' \neq 3, B'' \neq 6, m \notin M \tag{13.61}$$

www.dbooks.org

The following example Haskell program implements this solution. Note that it uses a plain list to represent the queue for illustration purpose.

```
import Data.Bits

solve = bfsSolve [[(15, 0)]] where
    bfsSolve :: [[(Int, Int)]] → [(Int, Int)]
    bfsSolve [] = [] -- no solution
    bfsSolve (c:cs) | (fst $ head c) == 0 = reverse c
                    | otherwise = bfsSolve (cs ++ map (:c)
                                        (filter (`valid` c) $ moves $ head c))
    valid (a, b) r = not $ or [ a `elem` [3, 6], b `elem` [3, 6],
                                (a, b) `elem` r]

moves (a, b) = if b < 8 then trans a b else map swap (trans b a) where
    trans x y = [(x - 8 - i, y + 8 + i)
                    | i ←[0, 1, 2, 4], i == 0 || (x ∘ &. i) /= 0]
    swap (x, y) = (y, x)
```

This algorithm can be easily modified to find all the possible solutions, but not just stop after finding the first one. This is left as the exercise to the reader. The following shows the two best solutions to this puzzle.

Solution 1:

Left	river	Right
wolf, goat, cabbage, farmer		
wolf, cabbage		goat, farmer
wolf, cabbage, farmer		goat
cabbage		wolf, goat, farmer
goat, cabbage, farmer		wolf
goat		wolf, cabbage, farmer
goat, farmer		wolf, cabbage
		wolf, goat, cabbage, farmer

Solution 2:

Left	river	Right
wolf, goat, cabbage, farmer		
wolf, cabbage		goat, farmer
wolf, cabbage, farmer		goat
wolf		goat, cabbage, farmer
wolf, goat, farmer		cabbage
goat		wolf, cabbage, farmer
goat, farmer		wolf, cabbage
		wolf, goat, cabbage, farmer

This algorithm can also be realized imperatively. Observing that our solution is in tail recursive manner, we can translate it directly to a loop. We use a list S to hold all the solutions can be found. The singleton list $\{(15, 0)\}$ is pushed to queue when initializing. As long as the queue isn't empty, we extract the head C from the queue by calling DeQ procedure. Examine if it reaches the final goal, if not, we expand all the possible moves and push to the tail of the queue for further searching.

1: **function** SOLVE
2: $S \leftarrow \phi$
3: $Q \leftarrow \phi$

```
4:    EnQ(Q, {(15, 0)})
5:    while Q ≠ φ do
6:        C ← DeQ(Q)
7:        if c_1 = (0, 15) then
8:            Add(S, Reverse(C))
9:        else
10:           for ∀m ∈ Moves(C) do
11:               if Valid(m, C) then
12:                   EnQ(Q, {m} ∪ C)
13:   return S
```

Where Moves, and Valid procedures are as same as before. The following Python example program implements this imperative algorithm.

```
def solve():
    s = []
    queue = [[(0xf, 0)]]
    while queue != []:
        cur = queue.pop(0)
        if cur[0] == (0, 0xf):
            s.append(reverse(cur))
        else:
            for m in moves(cur):
                queue.append([m]+cur)
    return s

def moves(s):
    (a, b) = s[0]
    return valid(s, trans(a, b) if b < 8 else swaps(trans(b, a)))

def valid(s, mv):
    return [(a, b) for (a, b) in mv
        if a not in [3, 6] and b not in [3, 6] and (a, b) not in s]

def trans(a, b):
    masks = [ 8 | (1<<i) for i in range(4)]
    return [(a ^ mask, b | mask) for mask in masks if a & mask == mask]

def swaps(s):
    return [(b, a) for (a, b) in s]
```

There is a minor difference between the program and the pseudo code, that the function to generate candidate moving options filters the invalid cases inside it.

Every time, no matter the farmer drives the boat back and forth, there are m options for him to choose, where m is the number of objects on the river bank the farmer drives from. m is always less than 4, that the algorithm won't take more than n^4 times at step n. This estimation is far more than the actual time, because we avoid trying all invalid cases. Our solution examines all the possible moving in the worst case. Because we check recorded steps to avoid repeated attempt, the algorithm takes about $O(n^2)$ time to search for n possible steps.

www.dbooks.org

Water jugs puzzle

This is a popular puzzle in classic AI. The history of it should be very long. It says that there are two jugs, one is 9 quarts, the other is 4 quarts. How to use them to bring up from the river exactly 6 quarts of water?

There are varies versions of this puzzle, although the volume of the jugs, and the target volume of water differ. The solver is said to be young Blaise Pascal when he was a child, the French mathematician, scientist in one story, and Simèon Denis Poisson in another story. Later in the popular Hollywood movie 'Die-Hard 3', actor Bruce Willis and Samuel L. Jackson were also confronted with this puzzle.

Pòlya gave a nice way to solve this problem backwards in [14].

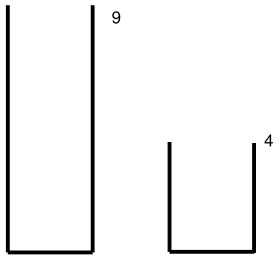


Figure 13.38: Two jugs with volume of 9 and 4.

Instead of thinking from the starting state as shown in figure 13.38. Pòlya pointed out that there will be 6 quarts of water in the bigger jugs at the final stage, which indicates the second last step, we can fill the 9 quarts jug, then pour out 3 quarts from it. In order to achieve this, there should be 1 quart of water left in the smaller jug as shown in figure 13.39.

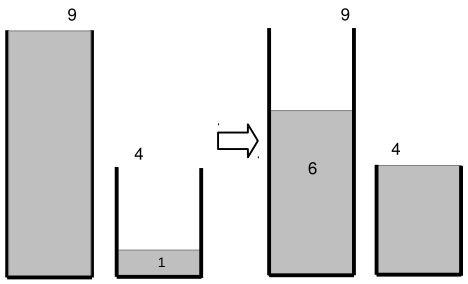


Figure 13.39: The last two steps.

It's easy to see that fill the 9 quarters jug, then pour to the 4 quarters jug twice can bring 1 quarters of water. As shown in figure 13.40. At this stage, we've found the solution. By reversing our findings, we can give the correct steps to bring exactly 6 quarters of water.

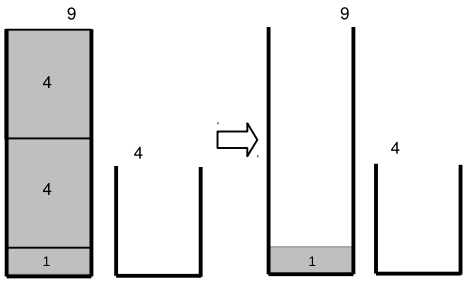


Figure 13.40: Fill the bigger jugs, and pour to the smaller one twice.

Pòlya's methodology is general. It's still hard to solve it without concrete algorithm. For instance, how to bring up 2 gallons of water from 899 and 1147 gallon jugs?

There are 6 ways to deal with 2 jugs in total. Denote the smaller jug as A, the bigger jug as B.

- Fill jug A from the river;
- Fill jug B from the river;
- Empty jug A;
- Empty jug B;
- Pour water from jug A to B;
- Pour water from jug B to A.

The following sequence shows an example. Note that in this example, we assume that $a < b < 2a$.

A	B	operation
0	0	start
a	0	fill A
0	a	pour A into B
a	a	fill A
2a - b	b	pour A into B
2a - b	0	empty B
0	2a - b	pour A into B
a	2a - b	fill A
3a - 2b	b	pour A into B
...	...	...

No matter what the above operations are taken, the amount of water in each jug can be expressed as $xa + yb$, where a and b are volumes of jugs, for some integers x and y. All the amounts of water we can get are linear combination of a and b. We can immediately tell given two jugs, if a goal g is solvable or not.

For instance, we can't bring 5 gallons of water with two jugs of volume 4 and 6 gallon. The number theory ensures that, the 2 water jugs puzzle can be

www.dbooks.org

solved if and only if g can be divided by the greatest common divisor of a and b. Written as:

$$gcd(a, b)|g \tag{13.62}$$

Where $m|n$ means n can be divided by m. What's more, if a and b are relatively prime, which means $gcd(a, b) = 1$, it's possible to bring up any quantity g of water.

Although $gcd(a, b)$ enables us to determine if the puzzle is solvable, it doesn't give us the detailed pour sequence. If we can find some integer x and y, so that $g = xa + yb$. We can arrange a sequence of operations (even it may not be the best solution) to solve it. The idea is that, without loss of generality, suppose $x > 0, y < 0$, we need fill jug A by x times, and empty jug B by y times in total.

Let's take $a = 3$, $b = 5$, and $g = 4$ for example, since $4 = 3 \times 3 - 5$, we can arrange a sequence like the following.

A	B	operation
0	0	start
3	0	fill A
0	3	pour A into B
3	3	fill A
1	5	pour A into B
1	0	empty B
0	1	pour A into B
3	1	fill A
0	4	pour A into B

In this sequence, we fill A by 3 times, and empty B by 1 time. The procedure can be described as the following:

Repeat x times:

1. Fill jug A;
2. Pour jug A into jug B, whenever B is full, empty it.

So the only problem left is to find the x and y. There is a powerful tool in number theory called, *Extended Euclid algorithm*, which can achieve this. Compare to the classic Euclid GCD algorithm, which can only give the greatest common divisor, The extended Euclid algorithm can give a pair of x, y as well, so that:

$$(d, x, y) = gcd_{ext}(a, b) \tag{13.63}$$

where $d = gcd(a, b)$ and $ax + by = d$. Without loss of generality, suppose $a < b$, there exits quotation q and remainder r that:

$$b = aq + r \tag{13.64}$$

Since d is the common divisor, it can divide both a and b, thus d can divide r as well. Because r is less than a, we can scale down the problem by finding GCD of a and r:

$$(d, x', y') = gcd_{ext}(r, a) \tag{13.65}$$

Where $d = x'r + y'a$ according to the definition of the extended Euclid algorithm. Transform $b = aq + r$ to $r = b - aq$, substitute r in above equation yields:

$$\begin{aligned} d &= x'(b - aq) + y'a \\ &= (y' - x'q)a + x'b \end{aligned} \tag{13.66}$$

This is the linear combination of a and b, so that we have:

$$\begin{cases} x = y' - x'\dfrac{b}{a} \\ y = x' \end{cases} \tag{13.67}$$

Note that this is a typical recursive relationship. The edge case happens when $a = 0$.

$$gcd(0, b) = b = 0a + 1b \tag{13.68}$$

Summarize the above result, the extended Euclid algorithm can be defined as the following:

$$gcd_{ext}(a, b) = \begin{cases} (b, 0, 1) & : \quad a = 0 \\ (d, y' - x'\dfrac{b}{a}, x') & : \quad otherwise \end{cases} \tag{13.69}$$

Where d, x', y' are defined in equation (13.65).

The 2 water jugs puzzle is almost solved, but there are still two detailed problems need to be tackled. First, extended Euclid algorithm gives the linear combination for the greatest common divisor d. While the target volume of water g isn't necessarily equal to d. This can be easily solved by multiplying x and y by m times, where $m = g/gcd(a, b)$; Second, we assume $x > 0$, to form a procedure to fill jug A with x times. However, the extended Euclid algorithm doesn't ensure x to be positive. For instance $gcd_{ext}(4, 9) = (1, -2, 1)$. Whenever we get a negative x, since $d = xa + yb$, we can continuously add b to x, and decrease y by a till x is greater than zero.

At this stage, we are able to give the complete solution to the 2 water jugs puzzle. Below is an example Haskell program.

```
extGcd 0 b = (b, 0, 1)
extGcd a b = let (d, x', y') = extGcd (b `mod` a) a in
               (d, y' - x' * (b `div` a), x')

solve a b g | g `mod` d /= 0 = [] -- no solution
            | otherwise = solve' (x * g `div` d)
    where
      (d, x, y) = extGcd a b
      solve' x | x < 0 = solve' (x + b)
               | otherwise = pour x [(0, 0)]
      pour 0 ps = reverse ((0, g):ps)
      pour x ps@((a', b'):_) | a' == 0 = pour (x - 1) ((a, b'):ps) -- fill a
                             | b' == b = pour x ((a', 0):ps) -- empty b
                             | otherwise = pour x ((max 0 (a' + b' - b),
                                                    min (a' + b') b):ps)
```

www.dbooks.org

Although we can solve the 2 water jugs puzzle with extended Euclid algorithm, the solution may not be the best. For instance, when we are going to bring 4 gallons of water from 3 and 5 gallons jugs. The extended Euclid algorithm produces the following sequence:

```
[(0,0),(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),
(0,4),(3,4),(2,5),(2,0),(0,2),(3,2),(0,5),(3,5),
(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),(0,4)]
```

It takes 23 steps to achieve the goal, while the best solution only need 6 steps:

```
[(0,0),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4)]
```

Observe the 23 steps, and we can find that jug B has already contained 4 gallons of water at the 8-th step. But the algorithm ignores this fact and goes on executing the left 15 steps. The reason is that the linear combination x and y we find with the extended Euclid algorithm are not the only numbers satisfying $g = xa + by$. For all these numbers, the smaller $|x| + |y|$, the less steps are needed. There is an exercise to addressing this problem in this section.

The interesting problem is how to find the best solution? We have two approaches, one is to find x and y to minimize $|x| + |y|$; the other is to adopt the quite similar idea as the wolf-goat-cabbage puzzle. We focus on the latter in this section. Since there are at most 6 possible options: fill A, fill B, pour A into B, pour B into A, empty A and empty B, we can try them in parallel, and check which decision can lead to the best solution. We need record all the states we've achieved to avoid any potential repetition. In order to realize this parallel approach with reasonable resources, a queue can be used to arrange our attempts. The elements stored in this queue are series of pairs (p, q), where p and q represent the volume of waters contained in each jug. These pairs record the sequence of our operations from the beginning to the latest. We initialize the queue with the singleton list contains the starting state $\{(0, 0)\}$.

$$solve(a, b, g) = solve'\{\{(0, 0)\}\} \tag{13.70}$$

Every time, when the queue isn't empty, we pick a sequence from the head of the queue. If this sequence ends with a pair contains the target volume g, we find a solution, we can print this sequence by reversing it; Otherwise, we expand the latest pair by trying all the possible 6 options, remove any duplicated states, and add them to the tail of the queue. Denote the queue as Q, the first sequence stored on the head of the queue as S, the latest pair in S as (p, q), and the rest of pairs as S'. After popping the head element, the queue becomes Q'. This algorithm can be defined like below:

$$solve'(Q) = \begin{cases} \phi & : & Q = \phi \\ reverse(S) & : & p = g \vee q = g \\ solve'(EnQ'(Q', \{\{s'\} \cup S' | s' \in try(S)\})) & : & otherwise \end{cases} \tag{13.71}$$

Where function EnQ' pushes a list of sequence to the queue one by one. Function $try(S)$ will try all possible 6 options to generate new pairs of water volumes:

$$try(S) = \{s' | s' \in \left\{ \begin{array}{l} fillA(p,q), fillB(p,q), \\ pourA(p,q), pourB(p,q), \\ emptyA(p,q), emptyB(p,q) \end{array} \right\}, s' \notin S'\} \tag{13.72}$$

It's intuitive to define the 6 options. For fill operations, the result is that the volume of the filled jug is full; for empty operation, the result volume is empty; for pour operation, we need test if the jug is big enough to hold all the water.

$$\begin{array}{ll} fillA(p,q) = (a,q) & fillB(p,q) = (p,b) \\ emptyA(p,q) = (0,q) & emptyB(p,q) = (p,0) \\ pourA(p,q) = (max(0, p+q-b), min(x+y, b)) & \\ pourB(p,q) = (min(x+y, a), max(0, x+y-a)) & \end{array} \tag{13.73}$$

The following example Haskell program implements this method:

```
solve' a b g = bfs [[(0, 0)]] where
  bfs [] = []
  bfs (c:cs) | fst (head c) == g || snd (head c) == g = reverse c
             | otherwise = bfs (cs ++ map (:c) (expand c))
  expand ((x, y):ps) = filter (`notElem` ps) $ map (λf → f x y)
                              [fillA, fillB, pourA, pourB, emptyA, emptyB]
  fillA _ y = (a, y)
  fillB x _ = (x, b)
  emptyA _ y = (0, y)
  emptyB x _ = (x, 0)
  pourA x y = (max 0 (x + y - b), min (x + y) b)
  pourB x y = (min (x + y) a, max 0 (x + y - a))
```

This method always returns the fast solution. It can also be realized in imperative approach. Instead of storing the complete sequence of operations in every element in the queue, we can store the unique state in a global history list, and use links to track the operation sequence, this can save spaces.

The idea is illustrated in figure 13.41. The initial state is (0, 0). Only 'fill A' and 'fill B' are possible. They are tried and added to the record list; Next we can try and record 'fill B' on top of (3, 0), which yields new state (3, 5). However, when try 'empty A' from state (3, 0), we would return to the start state (0, 0). As this previous state has been recorded, it is ignored. All the repeated states are in gray color in this figure.

With such settings, we needn't remember the operation sequence in each element in the queue explicitly. We can add a 'parent' link to each node in figure 13.41, and use it to back-traverse to the starting point from any state. The following example ANSI C code shows such a definition.

```
struct Step {
    int p, q;
    struct Step* parent;
};
```

www.dbooks.org

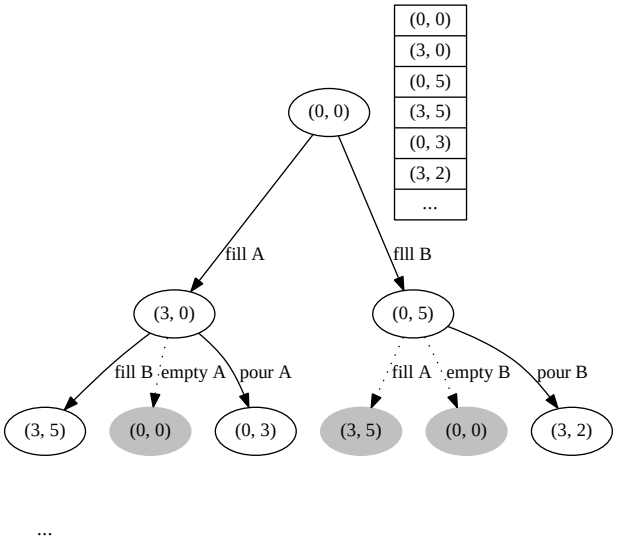


Figure 13.41: All attempted states are stored in a global list.

```
struct Step* make_step(int p, int q, struct Step* parent) {
    struct Step* s = (struct Step*) malloc(sizeof(struct Step));
    s→p = p;
    s→q = q;
    s→parent = parent;
    return s;
}
```

Where p, q are volumes of water in the 2 jugs. For any state s, define functions $p(s)$ and $q(s)$ return these 2 values, the imperative algorithm can be realized based on this idea as below.

```
1: function SOLVE(a, b, g)
2:     Q ← φ
3:     PUSH-AND-RECORD(Q, (0, 0))
4:     while Q ≠ φ do
5:         s ← POP(Q)
6:         if p(s) = g ∨ q(s) = g then
7:             return s
8:         else
9:             C ← EXPAND(s)
10:            for ∀c ∈ C do
11:                if c ≠ s ∧ ¬ VISITED(c) then
12:                    PUSH-AND-RECORD(Q, c)
13:    return NIL
```

Where PUSH-AND-RECORD does not only push an element to the queue, but also record this element as visited, so that we can check if an element has been visited before in the future. This can be implemented with a list. All push operations append the new elements to the tail. For pop operation, instead of removing the element pointed by head, the head pointer only advances to the

next one. This list contains historic data which has to be reset explicitly. The following ANSI C code illustrates this idea.

```
struct Step *steps[1000], **head, **tail = steps;

void push(struct Step* s) { *tail++ = s; }

struct Step* pop() { return *head++; }

int empty() { return head == tail; }

void reset() {
    struct Step **p;
    for (p = steps; p != tail; ++p)
        free(*p);
    head = tail = steps;
}
```

In order to test a state has been visited, we can traverse the list to compare p and q.

```
int eq(struct Step* a, struct Step* b) {
    return a->p == b->p && a->q == b->q;
}

int visited(struct Step* s) {
    struct Step **p;
    for (p = steps; p != tail; ++p)
        if (eq(*p, s)) return 1;
    return 0;
}
```

The main program can be implemented as below:

```
struct Step* solve(int a, int b, int g) {
    int i;
    struct Step *cur, *cs[6];
    reset();
    push(make_step(0, 0, NULL));
    while (!empty()) {
        cur = pop();
        if (cur->p == g || cur->q == g)
            return cur;
        else {
            expand(cur, a, b, cs);
            for (i = 0; i < 6; ++i)
                if(!eq(cur, cs[i]) && !visited(cs[i]))
                    push(cs[i]);
        }
    }
    return NULL;
}
```

Where function `expand` tries all the 6 possible options:

```
void expand(struct Step* s, int a, int b, struct Step** cs) {
    int p = s->p, q = s->q;
```

www.dbooks.org

```
    cs[0] = make_step(a, q, s); /*fill A*/
    cs[1] = make_step(p, b, s); /*fill B*/
    cs[2] = make_step(0, q, s); /*empty A*/
    cs[3] = make_step(p, 0, s); /*empty B*/
    cs[4] = make_step(max(0, p + q - b), min(p + q, b), s); /*pour A*/
    cs[5] = make_step(min(p + q, a), max(0, p + q - a), s); /*pour B*/
}
```

And the result steps is back tracked in reversed order, it can be output with a recursive function:

```
void print(struct Step* s) {
    if (s) {
        print(s->parent);
        printf("%d, %d\n", s->p, s->q);
    }
}
```

Kloski

Kloski is a block sliding puzzle. It appears in many countries. There are different sizes and layouts. Figure 13.42 illustrates a traditional Kloski game in China.



(a) Initial layout of blocks (b) Block layout after several movements

Figure 13.42: 'Huarong Dao', the traditional Kloski game in China.

In this puzzle, there are 10 blocks, each is labeled with text or icon. The smallest block has size of 1 unit square, the biggest one is 2×2 units size. Note there is a slot of 2 units wide at the middle-bottom of the board. The biggest block represents a king in ancient time, while the others are enemies. The goal is to move the biggest block to the slot, so that the king can escape. This game is named as 'Huarong Dao', or 'Huarong Escape' in China. Figure 13.43 shows the similar Kloski puzzle in Japan. The biggest block means daughter, while the others are her family members. This game is named as 'Daughter in the box' in Japan (Japanese name: *hakoiri musume*).

In this section, we want to find a solution, which can slide blocks from the initial state to the final state with the minimum movements.

The intuitive idea to model this puzzle is to use a 5×4 matrix representing the board. All pieces are labeled with a number. The following matrix M, for example, shows the initial state of the puzzle.



Figure 13.43: 'Daughter in the box', the Kloski game in Japan.

$$M = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix}$$

In this matrix, the cells of value i mean the i-th piece covers this cell. The special value 0 represents a free cell. By using sequence 1, 2, ... to identify pieces, a special layout can be further simplified as an array L. Each element is a list of cells covered by the piece indexed with this element. For example, $L[4] = \{(3,2),(3,3)\}$ means the 4-th piece covers cells at position $(3,2)$ and $(3,3)$, where (i,j) means the cell at row i and column j.

The starting layout can be written as the following Array.

$$\{\{(1,1),(2,1)\},\{(1,4),(2,4)\},\{(3,1),(4,1)\},\{(3,2),(3,3)\},\{(3,4),(4,4)\},$$
$$\{(5,1)\},\{(4,2)\},\{(4,3)\},\{(5,4)\},\{(1,2),(1,3),(2,2),(2,3)\}\}$$

When moving the Kloski blocks, we need examine all the 10 blocks, checking each block if it can move up, down, left and right. it seems that this approach would lead to a very huge amount of possibilities, because each step might have 10×4 options, there will be about 40^n cases in the n-th step.

Actually, there won't be so much options. For example, in the first step, there are only 4 valid moving: the 6-th piece moves right; the 7-th and 8-th move down; and the 9-th moves left.

All others are invalid moving. Figure 13.44 shows how to test if the moving is possible.

The left example illustrates sliding block labeled with 1 down. There are two cells covered by this block. The upper 1 moves to the cell previously occupied by this same block, which is also labeled with 1; The lower 1 moves to a free cell, which is labeled with 0;

The right example, on the other hand, illustrates invalid sliding. In this case, the upper cells could move to the cell occupied by the same block. However, the lower cell labeled with 1 can't move to the cell occupied by other block, which is labeled with 2.

In order to test the valid moving, we need examine all the cells a block will cover. If they are labeled with 0 or a number as same as this block, the moving is valid. Otherwise it conflicts with some other block. For a layout L,

www.dbooks.org

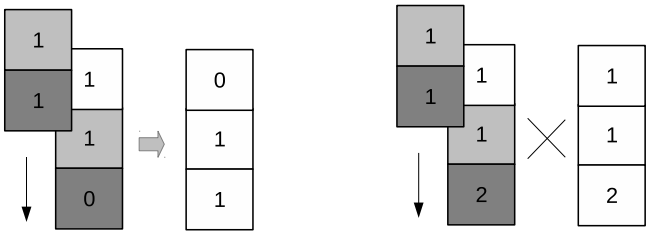


Figure 13.44: Left: both the upper and the lower 1 are OK; Right: the upper 1 is OK, the lower 1 conflicts with 2.

the corresponding matrix is M, suppose we want to move the k-th block with $(\Delta x, \Delta y)$, where $|\Delta x| \leq 1, |\Delta y| \leq 1$. The following equation tells if the moving is valid:

$$\begin{array}{rl} valid(L, k, \Delta x, \Delta y): & \\ \forall (i,j) \in L[k] \Rightarrow & i' = i + \Delta y, j' = j + \Delta x, \\ & (1,1) \leq (i', j') \leq (5,4), M_{i'j'} \in \{k, 0\} \end{array} \tag{13.74}$$

Another important point to solve Kloski puzzle, is about how to avoid repeated attempts. The obvious case is that after a series of sliding, we end up a matrix which have been transformed from. However, it is not enough to only avoid the same matrix. Consider the following two metrics. Although $M_1 \neq M_2$, we need drop options to M_2, because they are essentially the same.

$$M_1 = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix} \quad M_2 = \begin{bmatrix} 2 & 10 & 10 & 1 \\ 2 & 10 & 10 & 1 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 6 & 5 \\ 8 & 0 & 0 & 9 \end{bmatrix}$$

This fact tells us, that we should compare the layout, but not merely matrix to avoid repetition. Denote the corresponding layouts as L_1 and L_2 respectively, it's easy to verify that $||L_1|| = ||L_2||$, where $||L||$ is the normalized layout, which is defined as below:

$$||L|| = sort(\{sort(l_i) | \forall l_i \in L\}) \tag{13.75}$$

In other words, a normalized layout is ordered for all its elements, and every element is also ordered. The ordering can be defined as that $(a, b) \leq (c, d) \Leftrightarrow an + b \leq cn + d$, where n is the width of the matrix.

Observing that the Kloski board is symmetric, thus a layout can be mirrored from another one. Mirrored layout is also a kind of repeating, which should be avoided. The following M_1 and M_2 show such an example.

$$M_1 = \begin{bmatrix} 10 & 10 & 1 & 2 \\ 10 & 10 & 1 & 2 \\ 3 & 5 & 4 & 4 \\ 3 & 5 & 8 & 9 \\ 6 & 7 & 0 & 0 \end{bmatrix} \quad M_2 = \begin{bmatrix} 3 & 1 & 10 & 10 \\ 3 & 1 & 10 & 10 \\ 4 & 4 & 2 & 5 \\ 7 & 6 & 2 & 5 \\ 0 & 0 & 9 & 8 \end{bmatrix}$$

Note that, the normalized layouts are symmetric to each other. It's easy to get a mirrored layout like this:

$$mirror(L) = \{\{(i, n - j + 1)|\forall (i, j) \in l\}|\forall l \in L\} \tag{13.76}$$

We find that the matrix representation is useful in validating the moving, while the layout is handy to model the moving and avoid repeated attempt. We can use the similar approach to solve the Kloski puzzle. We need a queue, every element in the queue contains two parts: a series of moving and the latest layout led by the moving. Each moving is in form of $(k, (\Delta y, \Delta x))$, which means moving the k-th block, with Δy in row, and Δx in column in the board.

The queue contains the starting layout when initialized. Whenever this queue isn't empty, we pick the first one from the head, checking if the biggest block is on target, that $L[10] = \{(4, 2), (4, 3), (5, 2), (5, 3)\}$. If yes, then we are done; otherwise, we try to move every block with 4 options: left, right, up, and down, and store all the possible, unique new layout to the tail of the queue. During this searching, we need record all the normalized layouts we've ever found to avoid any duplication.

Denote the queue as Q, the historic layouts as H, the first layout on the head of the queue as L, its corresponding matrix as M. and the moving sequence to this layout as S. The algorithm can be defined as the following.

$$solve(Q, H) = \begin{cases} \phi & : \quad Q = \phi \\ reverse(S) & : \quad L[10] = \{(4, 2), (4, 3), (5, 2), (5, 3)\} \\ solve(Q', H') & : \quad otherwise \end{cases} \tag{13.77}$$

The first clause says that if the queue is empty, we've tried all the possibilities and can't find a solution; The second clause finds a solution, it returns the moving sequence in reversed order; These are two edge cases. Otherwise, the algorithm expands the current layout, puts all the valid new layouts to the tail of the queue to yield Q', and updates the normalized layouts to H'. Then it performs recursive searching.

In order to expand a layout to valid unique new layouts, we can define a function as below:

$$\begin{aligned} expand(L, H) = \{(k, (\Delta y, \Delta x)| \quad & \forall k \in \{1, 2, ..., 10\}, \\ & \forall (\Delta y, \Delta x) \in \{(0, -1), (0, 1), (-1, 0), (1, 0)\}, \\ & valid(L, k, \Delta x, \Delta y), unique(L', H)\} \end{aligned} \tag{13.78}$$

www.dbooks.org

Where L' is the the new layout by moving the k-th block with $(\Delta y, \Delta x)$ from L, M' is the corresponding matrix, and M'' is the matrix to the mirrored layout of L'. Function unique is defined like this:

$$unique(L', H) = M' \notin H \wedge M'' \notin H \tag{13.79}$$

We'll next show some example Haskell Kloski programs. As array isn't mutable in the purely functional settings, tree based map is used to represent layout [11]. Some type synonyms are defined as below:

```
import qualified Data.Map as M
import Data.Ix
import Data.List (sort)

type Point = (Integer, Integer)
type Layout = M.Map Integer [Point]
type Move = (Integer, Point)

data Ops = Op Layout [Move]
```

The main program is almost as same as the $solve(Q, H)$ function defined above.

```
solve :: [Ops] → [[[Point]]]→ [Move]
solve [] _ = [] -- no solution
solve (Op x seq : cs) visit
    | M.lookup 10 x == Just [(4, 2), (4, 3), (5, 2), (5, 3)] = reverse seq
    | otherwise = solve q visit'
  where
    ops = expand x visit
    visit' = map (layout ∘ move x) ops ++ visit
    q = cs ++ [Op (move x op) (op:seq) | op ← ops ]
```

Where function `layout` gives the normalized form by sorting. `move` returns the updated map by sliding the i-th block with $(\Delta y, \Delta x)$.

```
layout = sort ∘ map sort ∘ M.elems

move x (i, d) = M.update (Just ∘ map (flip shift d)) i x

shift (y, x) (dy, dx) = (y + dy, x + dx)
```

Function `expand` gives all the possible new options. It can be directly translated from $expand(L, H)$.

```
expand :: Layout → [[[Point]]] → [Move]
expand x visit = [(i, d) | i ←[1..10],
                           d ← [(0, -1), (0, 1), (-1, 0), (1, 0)],
                           valid i d, unique i d] where
  valid i d = all (λp → let p' = shift p d in
                    inRange (bounds board) p' &&
                    (M.keys $ M.filter (elem p') x) `elem` [[i], []])
              (maybe [] id $ M.lookup i x)
  unique i d = let mv = move x (i, d) in
               all (`notElem` visit) (map layout [mv, mirror mv])
```

[11] Alternatively, finger tree based sequence shown in previous chapter can be used

Note that we also filter out the mirrored layouts. The `mirror` function is given as the following.

```
mirror = M.map (map (λ (y, x) → (y, 5 - x)))
```

This program takes several minutes to produce the best solution, which takes 116 steps. The final 3 steps are shown as below:

```
...

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['A', 'A', '6', '0']
['A', 'A', '0', '8']

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['A', 'A', '0', '6']
['A', 'A', '0', '8']

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['0', 'A', 'A', '6']
['0', 'A', 'A', '8']

total 116 steps
```

The Kloski solution can also be realized imperatively. Note that the $solve(Q, H)$ is tail-recursive, it's easy to transform the algorithm with looping. We can also link one layout to its parent, so that the moving sequence can be recorded globally. This can save some spaces, as the queue needn't store the moving information in every element. When output the result, we only need back-tracking to the starting layout from the last one.

Suppose function $\textsc{Link}(L', L)$ links a new layout L' to its parent layout L. The following algorithm takes a starting layout, and searches for best moving sequence.

```
1:  function SOLVE(L_0)
2:      H ← ||L_0||
3:      Q ← φ
4:      PUSH(Q, LINK(L_0, NIL))
5:      while Q ≠ φ do
6:          L ← POP(Q)
7:          if L[10] = {(4,2),(4,3),(5,2),(5,3)} then
8:              return L
9:          else
10:             for each L' ∈ EXPAND(L, H) do
11:                 PUSH(Q, LINK(L', L))
12:                 APPEND(H, ||L'||)
```

www.dbooks.org

13: **return** NIL ▷ No solution

The following example Python program implements this algorithm:

```
class Node:
    def __init__(self, l, p = None):
        self.layout = l
        self.parent = p

def solve(start):
    visit = set([normalize(start)])
    queue = deque([Node(start)])
    while queue:
        cur = queue.popleft()
        layout = cur.layout
        if layout[-1] == [(4, 2), (4, 3), (5, 2), (5, 3)]:
            return cur
        else:
            for brd in expand(layout, visit):
                queue.append(Node(brd, cur))
                visit.add(normalize(brd))
    return None # no solution
```

Where `normalize` and `expand` are implemented as below:

```
def normalize(layout):
    return tuple(sorted([tuple(sorted(r)) for r in layout]))

def expand(layout, visit):
    def bound(y, x):
        return 1 ≤ y and y ≤ 5 and 1 ≤ x and x ≤ 4
    def valid(m, i, y, x):
        return m[y - 1][x - 1] in [0, i]
    def unique(brd):
        (m, n) = (normalize(brd), normalize(mirror(brd)))
        return m not in visit and n not in visit
    s = []
    d = [(0, -1), (0, 1), (-1, 0), (1, 0)]
    m = matrix(layout)
    for i in range(1, 11):
        for (dy, dx) in d:
            if all(bound(y + dy, x + dx) and valid(m, i, y + dy, x + dx)
                    for (y, x) in layout[i - 1]):
                brd = move(layout, (i, (dy, dx)))
                if unique(brd):
                    s.append(brd)
    return s
```

Like most programming languages, arrays are indexed from 0 but not 1 in Python. This has to be handled properly. The rest functions including `mirror`, `matrix`, and `move` are implemented as the following.

```
def mirror(layout):
    return [[(y, 5 - x) for (y, x) in r] for r in layout]

def matrix(layout):
    m = [[0]*4 for _ in range(5)]
```

```
    for (i, ps) in zip(range(1, 11), layout):
        for (y, x) in ps:
            m[y - 1][x - 1] = i
    return m

def move(layout, delta):
    (i, (dy, dx)) = delta
    m = dup(layout)
    m[i - 1] = [(y + dy, x + dx) for (y, x) in m[i - 1]]
    return m

def dup(layout):
    return [r[:] for r in layout]
```

It's possible to modify this Kloski algorithm, so that it does not only stop at the first solution, but also search all the solutions. In such case, the computation time is bound to the size of a space V, where V holds all the layouts can be transformed from the starting layout. If all these layouts are stored globally, with a parent field point to the predecessor, the space requirement of this algorithm is also bound to $O(V)$.

Summary of BFS

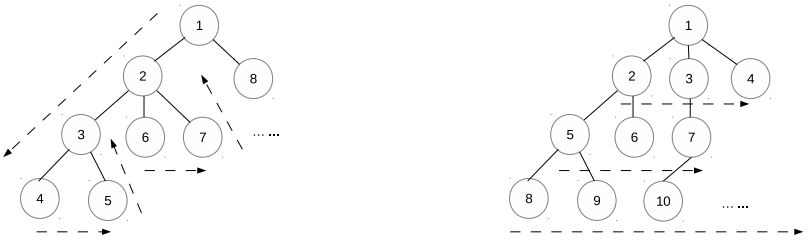
The above three puzzles, the wolf-goat-cabbage puzzle, the water jugs puzzle, and the Kloski puzzle show some common solution structure. Similar to the DFS problems, they all have the starting state and the end state. The wolf-goat-cabbage puzzle starts with the wolf, the goat, the cabbage and the farmer all in one side, while the other side is empty. It ends up in a state that they all moved to the other side. The water jugs puzzle starts with two empty jugs, and ends with either jug contains a certain volume of water. The Kloski puzzle starts from a layout and ends to another layout that the biggest block begging slided to a given position.

All problems specify a set of rules which can transfer from one state to another. Different form the DFS approach, we try all the possible options 'in parallel'. We won't search further until all the other alternatives in the same step have been examined. This method ensures that the solution with the minimum steps can be found before those with more steps. Review and compare the two figures we've drawn before shows the difference between these two approaches. For the later one, because we expand the searching horizontally, it is called as Breadth-first search (BFS for short).

As we can't perform search really in parallel, BFS realization typically utilizes a queue to store the search options. The candidate with less steps pops from the head, while the new candidate with more steps is pushed to the tail of the queue. Note that the queue should meet constant time enqueue and dequeue requirement, which we've explained in previous chapter of queue. Strictly speaking, the example functional programs shown above don't meet this criteria. They use list to mimic queue, which can only provide linear time pushing. Readers can replace them with the functional queue we explained before.

BFS provides a simple method to search for optimal solutions in terms of the number of steps. However, it can't search for more general optimal solution. Consider another directed graph as shown in figure 13.46, the length of each

www.dbooks.org



(a) Depth First Search (b) Breadth First Search

Figure 13.45: Search orders for DFS and BFS.

section varies. We can't use BFS to find the shortest route from one city to another.

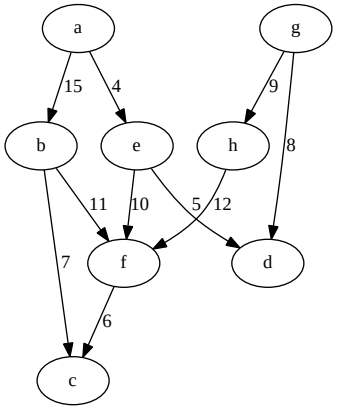


Figure 13.46: A weighted directed graph.

Note that the shortest route from city a to city c isn't the one with the fewest steps $a \to b \to c$. The total length of this route is 22; But the route with more steps $a \to e \to f \to c$ is the best. The length of it is 20. The coming sections introduce other algorithms to search for optimal solution.

13.3.2 Search the optimal solution

Searching for the optimal solution is quite important in many aspects. People need the 'best' solution to save time, space, cost, or energy. However, it's not easy to find the best solution with limited resources. There have been many optimal problems can only be solved by brute-force. Nevertheless, we've found that, for some of them, There exists special simplified ways to search the optimal solution.

Grady algorithm

Huffman coding

Huffman coding is a solution to encode information with the shortest length of code. Consider the popular ASCII code, which uses 7 bits to encode characters, digits, and symbols. ASCII code can represent $2^7 = 128$ different symbols. With 0, 1 bits, we need at least $\log_2 n$ bits to distinguish n different symbols. For text with only case insensitive English letters, we can define a code table like below.

char	code	char	code
A	00000	N	01101
B	00001	O	01110
C	00010	P	01111
D	00011	Q	10000
E	00100	R	10001
F	00101	S	10010
G	00110	T	10011
H	00111	U	10100
I	01000	V	10101
J	01001	W	10110
K	01010	X	10111
L	01011	Y	11000
M	01100	Z	11001

With this code table, text 'INTERNATIONAL' is encoded to 65 bits.

```
00010101101100100100100011011000000110010001001110101100000011010
```

Observe the above code table, which actually maps the letter 'A' to 'Z' from 0 to 25. There are 5 bits to represent every code. Code zero is forced as '00000' but not '0' for example. Such kind of coding method, is called fixed-length coding.

Another coding method is variable-length coding. That we can use just one bit '0' for 'A', two bits '10' for C, and 5 bits '11001' for 'Z'. Although this approach can shorten the total length of the code for 'INTERNATIONAL' from 65 bits dramatically, it causes problem when decoding. When processing a sequence of bits like '1101', we don't know if it means '1' followed by '101', which stands for 'BF'; or '110' followed by '1', which is 'GB', or '1101' which is 'N'.

The famous Morse code is variable-length coding system. That the most used letter 'E' is encoded as a dot, while 'Z' is encoded as two dashes and two dots. Morse code uses a special pause separator to indicate the termination of a code, so that the above problem won't happen. There is another solution to avoid ambiguity. Consider the following code table.

char	code	char	code
A	110	E	1110
I	101	L	1111
N	01	O	000
R	001	T	100

Text 'INTERNATIONAL' is encoded to 38 bits only:

```
10101100111000101110100101000011101111
```

www.dbooks.org

If decode the bits against the above code table, we won't meet any ambiguity symbols. This is because there is no code for any symbol is the prefix of another one. Such code is called *prefix-code*. (You may wonder why it isn't called as non-prefix code.) By using prefix-code, we needn't separators at all. So that the length of the code can be shorten.

This is a very interesting problem. Can we find a prefix-code table, which produce the shortest code for a given text? The very same problem was given to David A. Huffman in 1951, who was still a student in MIT[15]. His professor Robert M. Fano told the class that those who could solve this problem needn't take the final exam. Huffman almost gave up and started preparing the final exam when he found the most efficient answer.

The idea is to create the coding table according to the frequency of the symbol appeared in the text. The more used symbol is assigned with the shorter code.

It's not hard to process some text, and calculate the occurrence for each symbol. So that we have a symbol set, each one is augmented with a weight. The weight can be the number which indicates the frequency this symbol occurs. We can use the number of occurrence, or the probabilities for example.

Huffman discovered that a binary tree can be used to generate prefix-code. All symbols are stored in the leaf nodes. The codes are generated by traversing the tree from root. When go left, we add a zero; and when go right we add a one.

Figure 13.47 illustrates a binary tree. Taking symbol 'N' for example, starting from the root, we first go left, then right and arrive at 'N'. Thus the code for 'N' is '01'; While for symbol 'A', we can go right, right, then left. So 'A' is encode to '110'. Note that this approach ensures none code is the prefix of the other.

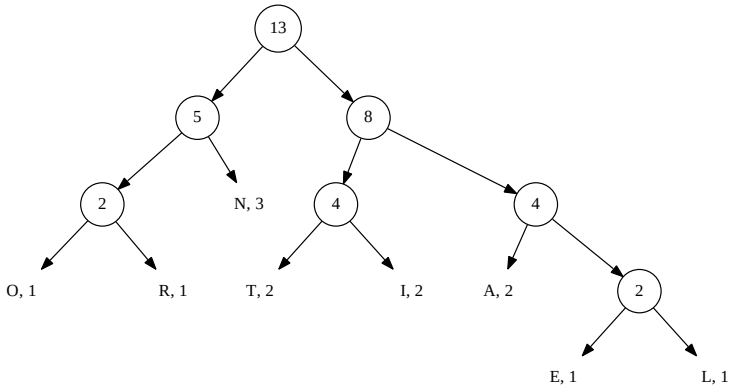
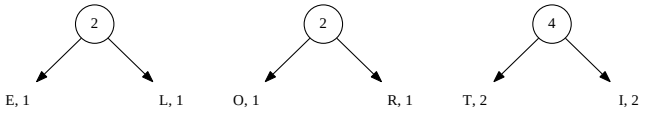


Figure 13.47: An encoding tree.

Note that this tree can also be used directly for decoding. When scan a series of bits, if the bit is zero, we go left; if the bit is one, we go right. When arrive at a leaf, we decode a symbol from that leaf. And we restart from the root of the tree for the coming bits.

Given a list of symbols with weights, we need build such a binary tree, so that the symbol with greater weight has shorter path from the root. Huffman

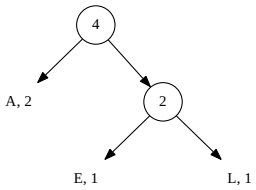
developed a bottom-up solution. When start, all symbols are put into a leaf node. Every time, we pick two nodes, which has the smallest weight, and merge them into a branch node. The weight of this branch is the sum of its two children. We repeatedly pick the two smallest weighted nodes and merge till there is only one tree left. Figure 13.48 illustrates such a building process.



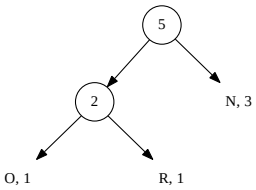
(a) 1.

(b) 2.

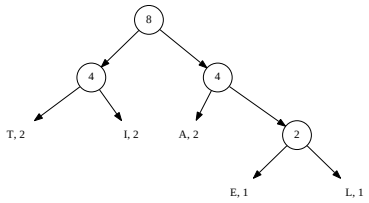
(c) 3.



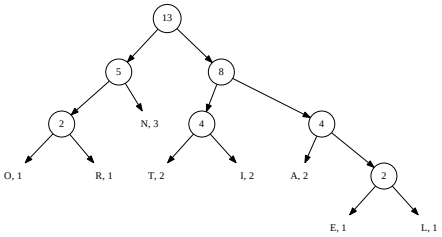
(d) 4.



(e) 5.



(f) 6.



(g) 7.

Figure 13.48: Steps to build a Huffman tree.

We can reuse the binary tree definition to formalize Huffman coding. We augment the weight information, and the symbols are only stored in leaf nodes. The following C like definition, shows an example.

```
struct Node {
    int w;
    char c;
    struct Node *left, *right;
};
```

www.dbooks.org

Some limitation can be added to the definition, as empty tree isn't allowed. A Huffman tree is either a leaf, which contains a symbol and its weight; or a branch, which only holds total weight of all leaves. The following Haskell code, for instance, explicitly specifies these two cases.

```
data HTr w a = Leaf w a | Branch w (HTr w a) (HTr w a)
```

When merge two Huffman trees T_1 and T_2 to a bigger one, These two trees are set as its children. We can select either one as the left, and the other as the right. the weight of the result tree T is the sum of its two children. so that $w = w_1 + w_2$. Define $T_1 < T_2$ if $w_1 < w_2$, One possible Huffman tree building algorithm can be realized as the following.

$$build(A) = \begin{cases} T_1 & : & A = \{T_1\} \\ build(\{merge(T_a, T_b)\} \cup A') & : & otherwise \end{cases} \tag{13.80}$$

A is a list of trees. It is initialized as leaves for all symbols and their weights. If there is only one tree in this list, we are done, the tree is the final Huffman tree. Otherwise, The two smallest tree T_a and T_b are extracted, and the rest trees are hold in list A'. T_a and T_b are merged to one bigger tree, and put back to the tree list for further recursive building.

$$(T_a, T_b, A') = extract(A) \tag{13.81}$$

We can scan the tree list to extract the 2 nodes with the smallest weight. Below equation shows that when the scan begins, the first 2 elements are compared and initialized as the two minimum ones. An empty accumulator is passed as the last argument.

$$extract(A) = extract'(min(T_1, T_2), max(T_1, T_2), \{T_3, T_4, ...\}, \phi) \tag{13.82}$$

For every tree, if its weight is less than the smallest two we've ever found, we update the result to contain this tree. For any given tree list A, denote the first tree in it as T_1, and the rest trees except T_1 as A'. The scan process can be defined as the following.

$$extract'(T_a, T_b, A, B) = \begin{cases} (T_a, T_b, B) & : & A = \phi \\ extract'(T'_a, T'_b, A', \{T_b\} \cup A) & : & T_1 < T_b \\ extract'(T_a, T_b, A', \{T_1\} \cup A) & : & otherwise \end{cases} \tag{13.83}$$

Where $T'_a = min(T_1, T_a)$, $T'_b = max(T_1, T_a)$ are the updated two trees with the smallest weights.

The following Haskell example program implements this Huffman tree building algorithm.

```
build [x] = x
build xs = build ((merge x y) : xs') where
  (x, y, xs') = extract xs

extract (x:y:xs) = min2 (min x y) (max x y) xs [] where
  min2 x y [] xs = (x, y, xs)
  min2 x y (z:zs) xs | z < y = min2 (min z x) (max z x) zs (y:xs)
                     | otherwise = min2 x y zs (z:xs)
```

This building solution can also be realized imperatively. Given an array of Huffman nodes, we can use the last two cells to hold the nodes with the smallest weights. Then we scan the rest of the array from right to left. Whenever there is a node with the smaller weight, this node will be exchanged with the bigger one of the last two. After all nodes have been examined, we merge the trees in the last two cells, and drop the last cell. This shrinks the array by one. We repeat this process till there is only one tree left.

```
1: function HUFFMAN(A)
2:     while |A| > 1 do
3:         n ← |A|
4:         for i ← n − 2 down to 1 do
5:             if A[i] < MAX(A[n], A[n − 1]) then
6:                 EXCHANGE A[i] ↔ MAX(A[n], A[n − 1])
7:         A[n − 1] ← MERGE(A[n], A[n − 1])
8:         DROP(A[n])
9:     return A[1]
```

The following C++ example program implements this algorithm. Note that this algorithm needn't the last two elements being ordered.

```
typedef vector<Node*> Nodes;

bool lessp(Node* a, Node* b) { return a->w < b->w; }

Node* max(Node* a, Node* b) { return lessp(a, b) ? b : a; }

void swap(Nodes& ts, int i, int j, int k) {
    swap(ts[i], ts[ts[j] < ts[k] ? k : j]);
}

Node* huffman(Nodes ts) {
    int n;
    while((n = ts.size()) > 1) {
        for (int i = n - 3; i >= 0; --i)
            if (lessp(ts[i], max(ts[n-1], ts[n-2])))
                swap(ts, i, n-1, n-2);
        ts[n-2] = merge(ts[n-1], ts[n-2]);
        ts.pop_back();
    }
    return ts.front();
}
```

The algorithm merges all the leaves, and it need scan the list in each iteration. Thus the performance is quadratic. This algorithm can be improved. Observe that each time, only the two trees with the smallest weights are merged. This reminds us the heap data structure. Heap ensures to access the smallest element fast. We can put all the leaves in a heap. For binary heap, this is typically a linear operation. Then we extract the minimum element twice, merge them, then put the bigger tree back to the heap. This is $O(\lg n)$ operation if binary heap is used. So the total performance is $O(n \lg n)$, which is better than the above algorithm. The next algorithm extracts the node from the heap, and

www.dbooks.org

starts Huffman tree building.

$$build(H) = reduce(top(H), pop(H)) \tag{13.84}$$

This algorithm stops when the heap is empty; Otherwise, it extracts another nodes from the heap for merging.

$$reduce(T, H) = \begin{cases} T & : \quad H = \phi \\ build(insert(merge(T, top(H)), pop(H))) & : \quad otherwise \end{cases} \tag{13.85}$$

Function *build* and *reduce* are mutually recursive. The following Haskell example program implements this algorithm by using heap defined in previous chapter.

```
huffman' :: (Num a, Ord a) ⇒ [(b, a)] → HTr a b
huffman' = build' ∘ Heap.fromList ∘ map (λ(c, w) → Leaf w c) where
  build' h = reduce (Heap.findMin h) (Heap.deleteMin h)
  reduce x Heap.E = x
  reduce x h = build' $ Heap.insert (Heap.deleteMin h) (merge x (Heap.findMin h))
```

The heap solution can also be realized imperatively. The leaves are firstly transformed to a heap, so that the one with the minimum weight is put on the top. As far as there are more than 1 elements in the heap, we extract the two smallest, merge them to a bigger one, and put back to the heap. The final tree left in the heap is the result Huffman tree.

```
1: function HUFFMAN'(A)
2:     BUILD-HEAP(A)
3:     while |A| > 1 do
4:         T_a ← HEAP-POP(A)
5:         T_b ← HEAP-POP(A)
6:         HEAP-PUSH(A, MERGE(T_a, T_b))
7:     return HEAP-POP(A)
```

The following example C++ code implements this heap solution. The heap used here is provided in the standard library. Because the max-heap, but not min-heap would be made by default, a greater predication is explicitly passed as argument.

```
bool greaterp(Node* a, Node* b) { return b→w < a→w; }

Node* pop(Nodes& h) {
    Node* m = h.front();
    pop_heap(h.begin(), h.end(), greaterp);
    h.pop_back();
    return m;
}

void push(Node* t, Nodes& h) {
    h.push_back(t);
    push_heap(h.begin(), h.end(), greaterp);
}

Node* huffman1(Nodes ts) {
```

```
    make_heap(ts.begin(), ts.end(), greaterp);
    while (ts.size() > 1) {
        Node* t1 = pop(ts);
        Node* t2 = pop(ts);
        push(merge(t1, t2), ts);
    }
    return ts.front();
}
```

When the symbol-weight list has been already sorted, there exists a linear time method to build the Huffman tree. Observe that during the Huffman tree building, it produces a series of merged trees with weight in ascending order. We can use a queue to manage the merged trees. Every time, we pick the two trees with the smallest weight from both the queue and the list, merge them and push the result to the queue. All the trees in the list will be processed, and there will be only one tree left in the queue. This tree is the result Huffman tree. This process starts by passing an empty queue as below.

$$build'(A) = reduce'(extract''(\phi, A)) \tag{13.86}$$

Suppose A is in ascending order by weight, At any time, the tree with the smallest weight is either the header of the queue, or the first element of the list. Denote the header of the queue is T_a, after pops it, the queue is Q'; The first element in A is T_b, the rest elements are hold in A'. Function $extract''$ can be defined like the following.

$$extract''(Q, A) = \begin{cases} (T_b, (Q, A')) & : & Q = \phi \\ (T_a, (Q', A)) & : & A = \phi \vee T_a < T_b \\ (T_b, (Q, A')) & : & otherwise \end{cases} \tag{13.87}$$

Actually, the pair of queue and tree list can be viewed as a special heap. The tree with the minimum weight is continuously extracted and merged.

$$reduce'(T, (Q, A)) = \begin{cases} T & : & Q = \phi \wedge A = \phi \\ reduce'(extract''(push(Q'', merge(T, T')), A'')) & : & otherwise \end{cases} \tag{13.88}$$

Where $(T', (Q'', A'')) = extract''(Q, A)$, which means extracting another tree. The following Haskell example program shows the implementation of this method. Note that this program explicitly sort the leaves, which isn't necessary if the leaves are ordered. Again, the list, but not a real queue is used here for illustration purpose. List isn't good at pushing new element, please refer to the chapter of queue for details about it.

```
huffman'' :: (Num a, Ord a) ⇒ [(b, a)] → HTr a b
huffman'' = reduce ∘ wrap ∘ sort ∘ map (λ(c, w) → Leaf w c) where
  wrap xs = delMin ([], xs)
  reduce (x, ([], [])) = x
  reduce (x, h) = let (y, (q, xs)) = delMin h in
                  reduce $ delMin (q ++ [merge x y], xs)
  delMin ([], (x:xs)) = (x, ([], xs))
  delMin ((q:qs), []) = (q, (qs, []))
```

www.dbooks.org

```
delMin ((q:qs), (x:xs)) | q < x = (q, (qs, (x:xs)))
                        | otherwise = (x, ((q:qs), xs))
```

This algorithm can also be realized imperatively.

```
1: function HUFFMAN"(A)                                  ▷ A is ordered by weight
2:     Q ← φ
3:     T ← EXTRACT(Q, A)
4:     while Q ≠ φ ∨ A ≠ φ do
5:         PUSH(Q, MERGE(T, EXTRACT(Q, A)))
6:         T ← EXTRACT(Q, A)
7:     return T
```

Where function EXTRACT(Q, A) extracts the tree with the smallest weight from the queue and the array of trees. It mutates the queue and array if necessary. Denote the head of the queue is T_a, and the first element of the array as T_b.

```
1: function EXTRACT(Q, A)
2:     if Q ≠ φ ∧ (A = φ ∨ T_a < T_b) then
3:         return POP(Q)
4:     else
5:         return DETACH(A)
```

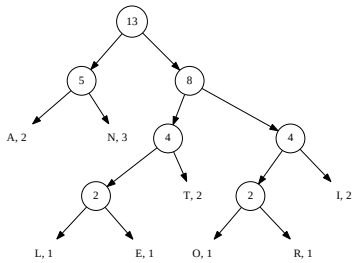
Where procedure DETACH(A), removes the first element from A, and returns this element as result. In most imperative settings, as detaching the first element is slow linear operation for array, we can store the trees in descending order by weight, and remove the last element. This is a fast constant time operation. The below C++ example code shows this idea.

```
Node* extract(queue<Node*>& q, Nodes& ts) {
    Node* t;
    if (!q.empty() && (ts.empty() || lessp(q.front(), ts.back()))) {
        t = q.front();
        q.pop();
    } else {
        t = ts.back();
        ts.pop_back();
    }
    return t;
}

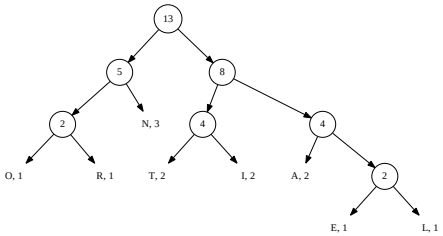
Node* huffman2(Nodes ts) {
    queue<Node*> q;
    sort(ts.begin(), ts.end(), greaterp);
    Node* t = extract(q, ts);
    while (!q.empty() || !ts.empty()) {
        q.push(merge(t, extract(q, ts)));
        t = extract(q, ts);
    }
    return t;
}
```

Note that the sorting isn't necessary if the trees have already been ordered. It can be a linear time reversing in case the trees are in ascending order by weight.

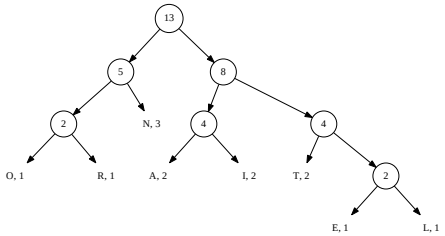
There are three different Huffman man tree building methods explained. Although they follow the same approach developed by Huffman, the result trees varies. Figure 13.49 shows the three different Huffman trees built with these methods.



(a) Created by scan method.



(b) Created by heap method.



(c) Linear time building for sorted list.

Figure 13.49: Variation of Huffman trees for the same symbol list.

Although these three trees are not identical. They are all able to generate the most efficient code. The formal proof is skipped here. The detailed information can be referred to [15] and Section 16.3 of [2].

The Huffman tree building is the core idea of Huffman coding. Many things can be easily achieved with the Huffman tree. For example, the code table can be generated by traversing the tree. We start from the root with the empty prefix p. For any branches, we append a zero to the prefix if turn left, and append a one if turn right. When a leaf node is arrived, the symbol represented by this node and the prefix are put to the code table. Denote the symbol of a leaf node as c, the children for tree T as T_l and T_r respectively. The code table association list can be built with $code(T, \phi)$, which is defined as below.

$$code(T, p) = \begin{cases} \{(c, p)\} & : \quad leaf(T) \\ code(T_l, p \cup \{0\}) \cup code(T_r, p \cup \{1\}) & : \quad otherwise \end{cases} \tag{13.89}$$

Where function $leaf(T)$ tests if tree T is a leaf or a branch node. The following Haskell example program generates a map as the code table according to this algorithm.

```
code tr = Map.fromList $ traverse [] tr where
  traverse bits (Leaf _ c) = [(c, bits)]
  traverse bits (Branch _ l r) = (traverse (bits ++ [0]) l) ++
                                 (traverse (bits ++ [1]) r)
```

www.dbooks.org

The imperative code table generating algorithm is left as exercise. The encoding process can scan the text, and look up the code table to output the bit sequence. The realization is skipped here.

The decoding process is realized by looking up the Huffman tree according to the bit sequence. We start from the root, whenever a zero is received, we turn left, otherwise if a one is received, we turn right. If a leaf node is arrived, the symbol represented by this leaf is output, and we start another looking up from the root. The decoding process ends when all the bits are consumed. Denote the bit sequence as $B = \{b_1, b_2, ...\}$, all bits except the first one are hold in B', below definition realizes the decoding algorithm.

$$decode(T, B) = \begin{cases} \{c\} & : & B = \phi \wedge leaf(T) \\ \{c\} \cup decode(root(T), B) & : & leaf(T) \\ decode(T_l, B') & : & b_1 = 0 \\ decode(T_r, B') & : & otherwise \end{cases} \tag{13.90}$$

Where $root(T)$ returns the root of the Huffman tree. The following Haskell example code implements this algorithm.

```
decode tr cs = find tr cs where
  find (Leaf _ c) [] = [c]
  find (Leaf _ c) bs = c : find tr bs
  find (Branch _ l r) (b:bs) = find (if b == 0 then l else r) bs
```

Note that this is an on-line decoding algorithm with linear time performance. It consumes one bit per time. This can be clearly noted from the below imperative realization, where the index keeps increasing by one.

```
1: function DECODE(T, B)
2:     W ← φ
3:     n ← |B|, i ← 1
4:     while i < n do
5:         R ← T
6:         while ¬ LEAF(R) do
7:             if B[i] = 0 then
8:                 R ← LEFT(R)
9:             else
10:                R ← RIGHT(R)
11:            i ← i + 1
12:        W ← W ∪ SYMBOL(R)
13:    return W
```

This imperative algorithm can be implemented as the following example C++ program.

```
string decode(Node* root, const char* bits) {
    string w;
    while (*bits) {
        Node* t = root;
        while (!isleaf(t))
            t = '0' == *bits++ ? t->left : t->right;
        w += t->c;
    }
```

```
    return w;
}
```

Huffman coding, especially the Huffman tree building shows an interesting strategy. Each time, there are multiple options for merging. Among the trees in the list, Huffman method always selects two trees with the smallest weight. This is the best choice at that merge stage. However, these series of *local* best options generate a global optimal prefix code.

It's not always the case that the local optimal choice also leads to the global optimal solution. In most cases, it doesn't. Huffman coding is a special one. We call the strategy that always choosing the local best option as *greedy* strategy.

Greedy method works for many problems. However, it's not easy to tell if the greedy method can be applied to get the global optimal solution. The generic formal proof is still an active research area. Section 16.4 in [2] provides a good treatment for Matroid tool, which covers many problems that greedy algorithm can be applied.

Change-making problem

We often change money when visiting other countries. People tend to use credit card more often nowadays than before, because it's quite convenient to buy things without considering much about changes. If we changed some money in the bank, there are often some foreign money left by the end of the trip. Some people like to change them to coins for collection. Can we find a solution, which can change the given amount of money with the least number of coins?

Let's use USA coin system for example. There are 5 different coins: 1 cent, 5 cent, 25 cent, 50 cent, and 1 dollar. A dollar is equal to 100 cents. Using the greedy method introduced above, we can always pick the largest coin which is not greater than the remaining amount of money to be changed. Denote list $C = \{1, 5, 25, 50, 100\}$, which stands for the value of coins. For any given money X, the change coins can be generated as below.

$$change(X, C) = \begin{cases} \phi & : \quad X = 0 \\ \{c_m\} \cup change(X - c_m, C) & : \quad otherwise, \\ & \quad c_m = max(\{c \in C, c \leq X\}) \end{cases} \tag{13.91}$$

If C is in descending order, c_m can be found as the first one not greater than X. If we want to change 1.42 dollar, This function produces a coin list of $\{100, 25, 5, 5, 5, 1, 1\}$. The output coins list can be easily transformed to contain pairs $\{(100, 1), (25, 1), (5, 3), (1, 2)\}$. That we need one dollar, a quarter, three coins of 5 cent, and 2 coins of 1 cent to make the change. The following Haskell example program outputs result as such.

```
solve x = assoc ∘ change x where
  change 0 _ = []
  change x cs = let c = head $ filter (≤ x) cs in c : change (x - c) cs

assoc = (map (λcs → (head cs, length cs))) ∘ group
```

As mentioned above, this program assumes the coins are in descending order, for instance like below.

www.dbooks.org

```
solve 142 [100, 50, 25, 5, 1]
```

This algorithm is tail recursive, it can be transformed to a imperative looping.

1: **function** CHANGE(X, C)
2: $R \leftarrow \phi$
3: **while** $X \neq 0$ **do**
4: $c_m = max(\{c \in C, c \leq X\})$
5: $R \leftarrow \{c_m\} \cup R$
6: $X \leftarrow X - c_m$
7: **return** R

The following example Python program implements this imperative version and manages the result with a dictionary.

```
def change(x, coins):
    cs = {}
    while x != 0:
        m = max([c for c in coins if c <= x])
        cs[m] = 1 + cs.setdefault(m, 0)
        x = x - m
    return cs
```

For a coin system like USA, the greedy approach can find the optimal solution. The amount of coins is the minimum. Fortunately, our greedy method works in most countries. But it is not always true. For example, suppose a country have coins of value 1, 3, and 4 units. The best change for value 6, is to use two coins of 3 units, however, the greedy method gives a result of three coins: one coin of 4, two coins of 1. Which isn't the optimal result.

Summary of greedy method

As shown in the change making problem, greedy method doesn't always give the best result. In order to find the optimal solution, we need dynamic programming which will be introduced in the next section.

However, the result is often good enough in practice. Let's take the word-wrap problem for example. In modern software editors and browsers, text spans to multiple lines if the length of the content is too long to be hold. With word-wrap supported, user needn't hard line breaking. Although dynamic programming can wrap with the minimum number of lines, it's overkill. On the contrary, greedy algorithm can wrap with lines approximate to the optimal result with quite effective realization as below. Here it wraps text T, not to exceeds line width W, with space s between each word.

1: $L \leftarrow W$
2: **for** $w \in T$ **do**
3: **if** $|w| + s > L$ **then**
4: Insert line break
5: $L \leftarrow W - |w|$
6: **else**
7: $L \leftarrow L - |w| - s$

For each word w in the text, it uses a greedy strategy to put as many words in a line as possible unless it exceeds the line width. Many word processors use

a similar algorithm to do word-wrapping.

There are many cases, the strict optimal result, but not the approximate one is necessary. Dynamic programming can help to solve such problems.

Dynamic programming

In the change-making problem, we mentioned the greedy method can't always give the optimal solution. For any coin system, are there any way to find the best changes?

Suppose we have find the best solution which makes X value of money. The coins needed are contained in C_m. We can partition these coins into two collections, C_1 and C_2. They make money of X_1, and X_2 respectively. We'll prove that C_1 is the optimal solution for X_1, and C_2 is the optimal solution for X_2.

Proof. For X_1, Suppose there exists another solution C_1', which uses less coins than C_1. Then changing solution $C_1' \cup C_2$ uses less coins to make X than C_m. This is conflict with the fact that C_m is the optimal solution to X. Similarity, we can prove C_2 is the optimal solution to X_2. □

Note that it is not true in the reverse situation. If we arbitrary select a value $Y < X$, divide the original problem to find the optimal solutions for sub problems Y and $X - Y$. Combine the two optimal solutions doesn't necessarily yield optimal solution for X. Consider this example. There are coins with value 1, 2, and 4. The optimal solution for making value 6, is to use 2 coins of value 2, and 4; However, if we divide $6 = 3+3$, since each 3 can be made with optimal solution $3 = 1 + 2$, the combined solution contains 4 coins $(1 + 1 + 2 + 2)$.

If an optimal problem can be divided into several sub optimal problems, we call it has optimal substructure. We see that the change-making problem has optimal substructure. But the dividing has to be done based on the coins, but not with an arbitrary value.

The optimal substructure can be expressed recursively as the following.

$$change(X) = \begin{cases} \phi & : \quad X = 0 \\ least(\{c \cup change(X - c) | c \in C, c \leq X\}) & : \quad otherwise \end{cases} \tag{13.92}$$

For any coin system C, the changing result for zero is empty; otherwise, we check every candidate coin c, which is not greater then value X, and recursively find the best solution for $X - c$; We pick the coin collection which contains the least coins as the result.

Below Haskell example program implements this top-down recursive solution.

```
change _ 0 = []
change cs x = minimumBy (compare `on` length)
                [c:change cs (x - c) | c ← cs, c ≤ x]
```

Although this program outputs correct answer `[2, 4]` when evaluates `change [1, 2, 4] 6`, it performs very bad when changing 1.42 dollar with USA coins system. It failed to find the answer within 15 minutes in a computer with 2.7GHz CPU and 8G memory.

www.dbooks.org

The reason why it's slow is because there are a lot of duplicated computing in the top-down recursive solution. When it computes $change(142)$, it needs to examine $change(141), change(137), change(117), change(92)$, and $change(42)$. While $change(141)$ next computes to smaller values by deducing with 1, 2, 25, 50 and 100 cents. it will eventually meets value 137, 117, 92, and 42 again. The search space explodes with power of 5.

This is quite similar to compute Fibonacci numbers in a top-down recursive way.

$$F_n = \begin{cases} 1 & : \quad n = 1 \vee n = 2 \\ F_{n-1} + F_{n-2} & : \quad otherwise \end{cases} \tag{13.93}$$

When we calculate F_8 for example, we recursively calculate F_7 and F_6. While when we calculate F_7, we need calculate F_6 again, and F_5, ... As shown in the below expand forms, the calculation is doubled every time, and the same value is calculate again and again.

$$\begin{aligned} F_8 &= F_7 + F_6 \\ &= F_6 + F_5 + F_5 + F_4 \\ &= F_5 + F_4 + F_4 + F_3 + F_4 + F_3 + F_3 + F_2 \\ &= ... \end{aligned}$$

In order to avoid duplicated computation, a table F can be maintained when calculating the Fibonacci numbers. The first two elements are filled as 1, all others are left blank. During the top-down recursive calculation, If need F_k, we first look up this table for the k-th cell, if it isn't blank, we use that value directly. Otherwise we need further calculation. Whenever a value is calculated, we store it in the corresponding cell for looking up in the future.

```
1: F ← {1, 1, NIL, NIL, ...}
2: function FIBONACCI(n)
3:     if n > 2 ∧ F[n] = NIL then
4:         F[n] ← FIBONACCI(n − 1) + FIBONACCI(n − 2)
5:     return F[n]
```

By using the similar idea, we can develop a new top-down change-making solution. We use a table T to maintain the best changes, it is initialized to all empty coin list. During the top-down recursive computation, we look up this table for smaller changing values. Whenever a intermediate value is calculated, it is stored in the table.

```
1: T ← {φ, φ, ...}
2: function CHANGE(X)
3:     if X > 0 ∧ T[X] = φ then
4:         for c ∈ C do
5:             if c ≤ X then
6:                 C_m ← {c} ∪ CHANGE(X − c)
7:                 if T[X] = φ ∨ |C_m| < |T[X]| then
8:                     T[X] ← C_m
9:     return T[X]
```

The solution to change 0 money is definitely empty ϕ, otherwise, we look up $T[X]$ to retrieve the solution to change X money. If it is empty, we need recursively calculate it. We examine all coins in the coin system C which is not

greater than X. This is the sub problem of making changes for money $X - c$. The minimum amount of coins plus one coin of c is stored in $T[X]$ finally as the result.

The following example Python program implements this algorithm just takes 8000 ms to give the answer of changing 1.42 dollar in US coin system.

```
tab = [[] for _ in range(1000)]

def change(x, cs):
    if x > 0 and tab[x] == []:
        for s in [[c] + change(x - c, cs) for c in cs if c <= x]:
            if tab[x] == [] or len(s) < len(tab[x]):
                tab[x] = s
    return tab[x]
```

Another solution to calculate Fibonacci number, is to compute them in order of $F_1, F_2, F_3, ..., F_n$. This is quite natural when people write down Fibonacci series.

1: **function** FIBO(n)
2: $F = \{1, 1, NIL, NIL, ...\}$
3: **for** $i \leftarrow 3$ to n **do**
4: $F[i] \leftarrow F[i-1] + F[i-2]$
5: **return** $F[n]$

We can use the quite similar idea to solve the change making problem. Starts from zero money, which can be changed by an empty list of coins, we next try to figure out how to change money of value 1. In US coin system for example, A cent can be used; The next values of 2, 3, and 4, can be changed by two coins of 1 cent, three coins of 1 cent, and 4 coins of 1 cent. At this stage, the solution table looks like below

0	1	2	3	4
ϕ	$\{1\}$	$\{1,1\}$	$\{1,1,1\}$	$\{1,1,1,1\}$

The interesting case happens for changing value 5. There are two options, use another coin of 1 cent, which need 5 coins in total; The other way is to use 1 coin of 5 cent, which uses less coins than the former. So the solution table can be extended to this.

0	1	2	3	4	5
ϕ	$\{1\}$	$\{1,1\}$	$\{1,1,1\}$	$\{1,1,1,1\}$	$\{5\}$

For the next change value 6, since there are two types of coin, 1 cent and 5 cent, are less than this value, we need examine both of them.

- If we choose the 1 cent coin, we need next make changes for 5; Since we've already known that the best solution to change 5 is $\{5\}$, which only needs a coin of 5 cents, by looking up the solution table, we have one candidate solution to change 6 as $\{5, 1\}$;
- The other option is to choose the 5 cent coin, we need next make changes for 1; By looking up the solution table we've filled so far, the sub optimal solution to change 1 is $\{1\}$. Thus we get another candidate solution to change 6 as $\{1, 5\}$;

It happens that, both options yield a solution of two coins, we can select either of them as the best solution. Generally speaking, the candidate with fewest number of coins is selected as the solution, and filled into the table.

www.dbooks.org

At any iteration, when we are trying to change the $i < X$ value of money, we examine all the types of coin. For any coin c not greater than i, we look up the solution table to fetch the sub solution $T[i-c]$. The number of coins in this sub solution plus the one coin of c are the total coins needed in this candidate solution. The fewest candidate is then selected and updated to the solution table.

The following algorithm realizes this bottom-up idea.

```
1: function CHANGE(X)
2:     T ← {φ, φ, ...}
3:     for i ← 1 to X do
4:         for c ∈ C, c ≤ i do
5:             if T[i] = φ ∨ 1 + |T[i − c]| < |T[i]| then
6:                 T[i] ← {c} ∪ T[i − c]
7:     return T[X]
```

This algorithm can be directly translated to imperative programs, like Python for example.

```
def changemk(x, cs):
    s = [[] for _ in range(x+1)]
    for i in range(1, x+1):
        for c in cs:
            if c ≤ i and (s[i] == [] or 1 + len(s[i-c]) < len(s[i])):
                s[i] = [c] + s[i-c]
    return s[x]
```

Observe the solution table, it's easy to find that, there are many duplicated contents being stored.

6	7	8	9	10	...
$\{1,5\}$	$\{1,1,5\}$	$\{1,1,1,5\}$	$\{1,1,1,1,5\}$	$\{5,5\}$	...

This is because the optimal sub solutions are completely copied and saved in parent solution. In order to use less space, we can only record the 'delta' part from the sub optimal solution. In change-making problem, it means that we only need to record the coin being selected for value i.

```
1: function CHANGE'(X)
2:     T ← {0, ∞, ∞, ...}
3:     S ← {NIL, NIL, ...}
4:     for i ← 1 to X do
5:         for c ∈ C, c ≤ i do
6:             if 1 + T[i − c] < T[i] then
7:                 T[i] ← 1 + T[i − c]
8:                 S[i] ← c
9:     while X > 0 do
10:        PRINT(S[X])
11:        X ← X − S[X]
```

Instead of recording the complete solution list of coins, this new algorithm uses two tables T and S. T holds the minimum number of coins needed for changing value 0, 1, 2, ...; while S holds the first coin being selected for the optimal solution. For the complete coin list to change money X, the first coin is thus $S[X]$, the sub optimal solution is to change money $X' = X - S[X]$. We can look up table $S[X']$ for the next coin. The coins for sub optimal solutions

are repeatedly looked up like this till the beginning of the table. Below Python example program implements this algorithm.

```
def chgmk(x, cs):
    cnt = [0] + [x+1] * x
    s = [0]
    for i in range(1, x+1):
        coin = 0
        for c in cs:
            if c ≤ i and 1 + cnt[i-c] < cnt[i]:
                cnt[i] = 1 + cnt[i-c]
                coin = c
        s.append(coin)
    r = []
    while x > 0:
        r.append(s[x])
        x = x - s[x]
    return r
```

This change-making solution loops n times for given money n. It examines at most the full coin system in each iteration. The time is bound to $\Theta(nk)$ where k is the number of coins for a certain coin system. The last algorithm adds $O(n)$ spaces to record sub optimal solutions with table T and S.

In purely functional settings, There is no means to mutate the solution table and look up in constant time. One alternative is to use finger tree as we mentioned in previous chapter [12]. We can store the minimum number of coins, and the coin leads to the sub optimal solution in pairs.

The solution table, which is a finger tree, is initialized as $T = \{(0, 0)\}$. It means change 0 money need no coin. We can fold on list $\{1, 2, ..., X\}$, start from this table, with a binary function $change(T, i)$. The folding will build the solution table, and we can construct the coin list from this table by function $make(X, T)$.

$$makeChange(X) = make(X, fold(change, \{(0, 0)\}, \{1, 2, ..., X\})) \tag{13.94}$$

In function $change(T, i)$, all the coins not greater than i are examined to select the one lead to the best result. The fewest number of coins, and the coin being selected are formed to a pair. This pair is inserted to the finger tree, so that a new solution table is returned.

$$change(T, i) = insert(T, fold(sel, (\infty, 0), \{c | c \in C, c \leq i\})) \tag{13.95}$$

Again, folding is used to select the candidate with the minimum number of coins. This folding starts with initial value $(\infty, 0)$, on all valid coins. function $sel((n, c), c')$ accepts two arguments, one is a pair of length and coin, which is the best solution so far; the other is a candidate coin, it examines if this candidate can make better solution.

$$sel((n, c), c') = \begin{cases} (1 + n', c') & : \quad 1 + n' < n, (n', c') = T[i - c'] \\ (n, c) & : \quad otherwise \end{cases} \tag{13.96}$$

[12] Some purely functional programming environments, Haskell for instance, provide built-in array; while other almost pure ones, such as ML, provide mutable array

www.dbooks.org

After the solution table is built, the coins needed can be generated from it.

$$make(X, T) = \begin{cases} \phi & : \quad X = 0 \\ \{c\} \cup make(X - c, T) & : \quad otherwise, (n, c) = T[X] \end{cases} \tag{13.97}$$

The following example Haskell program uses `Data.Sequence`, which is the library of finger tree, to implement change making solution.

```
import Data.Sequence (Seq, singleton, index, (|>))

changemk x cs = makeChange x $ foldl change (singleton (0, 0)) [1..x] where
  change tab i = let sel c = min (1 + fst (index tab (i - c)), c)
                 in tab |> (foldr sel ((x + 1), 0) $ filter (≤i) cs)
  makeChange 0 _ = []
  makeChange x tab = let c = snd $ index tab x in c : makeChange (x - c) tab
```

It's necessary to memorize the optimal solution to sub problems no matter using the top-down or the bottom-up approach. This is because a sub problem is used many times when computing the overall optimal solution. Such properties are called overlapping sub problems.

Properties of dynamic programming

Dynamic programming was originally named by Richard Bellman in 1940s. It is a powerful tool to search for optimal solution for problems with two properties.

- Optimal sub structure. The problem can be broken down into smaller problems, and the optimal solution can be constructed efficiently from solutions of these sub problems;
- Overlapping sub problems. The problem can be broken down into sub problems which are reused several times in finding the overall solution.

The change-making problem, as we've explained, has both optimal sub structures, and overlapping sub problems.

Longest common subsequence problem

The longest common subsequence problem, is different with the longest common substring problem. We've show how to solve the later in the chapter of suffix tree. The longest common subsequence needn't be consecutive part of the original sequence.

For example, The longest common substring for text "Mississippi", and "Misunderstanding" is "Miss", while the longest common subsequence for them are "Misssi". This is shown in figure 13.50.

If we rotate the figure vertically, and consider the two texts as two pieces of source code, it turns to be a 'diff' result between them. Most modern version control tools need calculate the difference content among the versions. The longest common subsequence problem plays a very important role.

If either one of the two strings X and Y is empty, the longest common subsequence $LCS(X, Y)$ is definitely empty; Otherwise, denote $X = \{x_1, x_2, ..., x_n\}$, $Y = \{y_1, y_2, ..., y_m\}$, if the first elements x_1 and y_1 are same, we can recursively

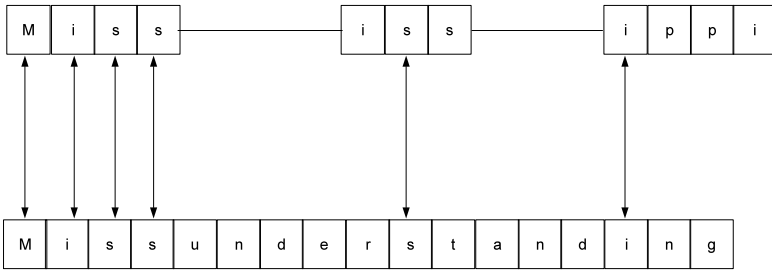


Figure 13.50: The longest common subsequence

find the longest subsequence of $X' = \{x_2, x_3, ..., x_n\}$ and $Y' = \{y_2, y_3, ..., y_m\}$. And the final result $LCS(X, Y)$ can be constructed by concatenating x_1 with $LCS(X', Y')$; Otherwise if $x_1 \neq y_1$, we need recursively find the longest common subsequences of $LCS(X, Y')$ and $LCS(X', Y)$, and pick the longer one as the final result. Summarize these cases gives the below definition.

$$LCS(X, Y) = \begin{cases} \phi & : \quad X = \phi \vee Y = \phi \\ \{x_1\} \cup LCS(X', Y') & : \quad x_1 = y_1 \\ longer(LCS(X, Y'), LCS(X', Y)) & : \quad otherwise \end{cases} \tag{13.98}$$

Note that this algorithm shows clearly the optimal substructure, that the longest common subsequence problem can be broken to smaller problems. The sub problem is ensured to be at least one element shorter than the original one.

It's also clear that, there are overlapping sub-problems. The longest common subsequences to the sub strings are used multiple times in finding the overall optimal solution.

The existence of these two properties, the optimal substructure and the overlapping sub-problem, indicates the dynamic programming can be used to solve this problem.

A 2-dimension table can be used to record the solutions to the sub-problems. The rows and columns represent the substrings of X and Y respectively.

www.dbooks.org

		a	n	t	e	n	n	a
		1	2	3	4	5	6	7
b	1							
a	2							
n	3							
a	4							
n	5							
a	6							

This table shows an example of finding the longest common subsequence for strings "antenna" and "banana". Their lengths are 7, and 6. The right bottom corner of this table is looked up first, Since it's empty we need compare the 7th element in "antenna" and the 6th in "banana", they are both 'a', Thus we need next recursively look up the cell at row 5, column 6; It's still empty, and we repeated this till either get a trivial case that one substring becomes empty, or some cell we are looking up has been filled before. Similar to the change-making problem, whenever the optimal solution for a sub-problem is found, it is recorded in the cell for further reusing. Note that this process is in the reversed order comparing to the recursive equation given above, that we start from the right most element of each string.

Considering that the longest common subsequence for any empty string is still empty, we can extended the solution table so that the first row and column hold the empty strings.

		a	n	t	e	n	n	a
		ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
b	ϕ							
a	ϕ							
n	ϕ							
a	ϕ							
n	ϕ							
a	ϕ							

Below algorithm realizes the top-down recursive dynamic programming solution with such a table.

```
1: T ← NIL
2: function LCS(X, Y)
3:     m ← |X|, n ← |Y|
4:     m' ← m + 1, n' ← n + 1
5:     if T = NIL then
6:         T ← {{φ, φ, ..., φ}, {φ, NIL, NIL, ...}, ...}          ▷ m' × n'
7:     if X ≠ φ ∧ Y ≠ φ ∧ T[m'][n'] = NIL then
8:         if X[m] = Y[n] then
9:             T[m'][n'] ← APPEND(LCS(X[1..m − 1], Y[1..n − 1]), X[m])
10:        else
11:            T[m'][n'] ← LONGER(LCS(X, Y[1..n − 1]), LCS(X[1..m − 1], Y))
12:    return T[m'][n']
```

The table is firstly initialized with the first row and column filled with empty strings; the rest are all NIL values. Unless either string is empty, or the cell content isn't NIL, the last two elements of the strings are compared, and recursively computes the longest common subsequence with substrings. The following

Python example program implements this algorithm.

```
def lcs(xs, ys):
    m = len(xs)
    n = len(ys)
    global tab
    if tab is None:
        tab = [[""]*(n+1)] + [[""] + [None]*n for _ in xrange(m)]
    if m != 0 and n !=0 and tab[m][n] is None:
        if xs[-1] == ys[-1]:
            tab[m][n] = lcs(xs[:-1], ys[:-1]) + xs[-1]
        else:
            (a, b) = (lcs(xs, ys[:-1]), lcs(xs[:-1], ys))
            tab[m][n] = a if len(b) < len(a) else b
    return tab[m][n]
```

The longest common subsequence can also be found in a bottom-up manner as what we've done with the change-making problem. Besides that, instead of recording the whole sequences in the table, we can just store the lengths of the longest subsequences, and later construct the subsubsequence with this table and the two strings. This time, the table is initialized with all values set as 0.

```
1: function LCS(X, Y)
2:     m ← |X|, n ← |Y|
3:     T ← {{0, 0, ...}, {0, 0, ...}, ...}                ▷ (m + 1) × (n + 1)
4:     for i ← 1 to m do
5:         for j ← 1 to n do
6:             if X[i] = Y[j] then
7:                 T[i + 1][j + 1] ← T[i][j] + 1
8:             else
9:                 T[i + 1][j + 1] ← MAX(T[i][j + 1], T[i + 1][j])
10:    return GET(T, X, Y, m, n)

11: function GET(T, X, Y, i, j)
12:    if i = 0 ∨ j = 0 then
13:        return φ
14:    else if X[i] = Y[j] then
15:        return APPEND(GET(T, X, Y, i − 1, j − 1), X[i])
16:    else if T[i − 1][j] > T[i][j − 1] then
17:        return GET(T, X, Y, i − 1, j)
18:    else
19:        return GET(T, X, Y, i, j − 1)
```

In the bottom-up approach, we start from the cell at the second row and the second column. The cell is corresponding to the first element in both X, and Y. If they are same, the length of the longest common subsequence so far is 1. This can be yielded by increasing the length of empty sequence, which is stored in the top-left cell, by one; Otherwise, we pick the maximum value from the upper cell and left cell. The table is repeatedly filled in this manner.

After that, a back-track is performed to construct the longest common subsequence. This time we start from the bottom-right corner of the table. If the last elements in X and Y are same, we put this element as the last one of the result, and go on looking up the cell along the diagonal line; Otherwise, we

www.dbooks.org

compare the values in the left cell and the above cell, and go on looking up the cell with the bigger value.

The following example Python program implements this algorithm.

```
def lcs(xs, ys):
    m = len(xs)
    n = len(ys)
    c = [[0]*(n+1) for _ in xrange(m+1)]
    for i in xrange(1, m+1):
        for j in xrange(1, n+1):
            if xs[i-1] == ys[j-1]:
                c[i][j] = c[i-1][j-1] + 1
            else:
                c[i][j] = max(c[i-1][j], c[i][j-1])

    return get(c, xs, ys, m, n)

def get(c, xs, ys, i, j):
    if i==0 or j==0:
        return []
    elif xs[i-1] == ys[j-1]:
        return get(c, xs, ys, i-1, j-1) + [xs[i-1]]
    elif c[i-1][j] > c[i][j-1]:
        return get(c, xs, ys, i-1, j)
    else:
        return get(c, xs, ys, i, j-1)
```

The bottom-up dynamic programming solution can also be defined in purely functional way. The finger tree can be used as a table. The first row is filled with $n+1$ zero values. This table can be built by folding on sequence X. Then the longest common subsequence is constructed from the table.

$$LCS(X, Y) = construct(fold(f, \{\{0, 0, ..., 0\}\}, zip(\{1, 2, ...\}, X))) \quad (13.99)$$

Note that, since the table need be looked up by index, X is zipped with natural numbers. Function f creates a new row of this table by folding on sequence Y, and records the lengths of the longest common sequence for all possible cases so far.

$$f(T, (i, x)) = insert(T, fold(longest, \{0\}, zip(\{1, 2, ...\}, Y))) \quad (13.100)$$

Function *longest* takes the intermediate filled row result, and a pair of index and element in Y, it compares if this element is the same as the one in X. Then fills the new cell with the length of the longest one.

$$longest(R, (j, y)) = \begin{cases} insert(R, 1 + T[i-1][j-1]) & : \quad x = y \\ insert(R, max(T[i-1][j], T[i][j-1])) & : \quad otherwise \end{cases} \quad (13.101)$$

After the table is built. The longest common sub sequence can be constructed recursively by looking up this table. We can pass the reversed sequences $\overleftarrow{X}$, and $\overleftarrow{Y}$ together with their lengths m and n for efficient building.

$$construct(T) = get((\overleftarrow{X}, m), (\overleftarrow{Y}, n)) \quad (13.102)$$

If the sequences are not empty, denote the first elements as x and y. The rest elements are hold in $\overleftarrow{X}'$ and $\overleftarrow{Y}'$ respectively. The function get can be defined as the following.

$$get((\overleftarrow{X}, i), (\overleftarrow{Y}, j)) = \begin{cases} \phi & : \overleftarrow{X} = \phi \wedge \overleftarrow{Y} = \phi \\ get((\overleftarrow{X}', i-1), (\overleftarrow{Y}', j-1)) \cup \{x\} & : x = y \\ get((\overleftarrow{X}', i-1), (\overleftarrow{Y}, j)) & : T[i-1][j] > T[i][j-1] \\ get((\overleftarrow{X}, i), (\overleftarrow{Y}', j-1)) & : otherwise \end{cases} \tag{13.103}$$

Below Haskell example program implements this solution.

```
lcs' xs ys = construct $ foldl f (singleton $ fromList $ replicate (n+1) 0)
                                 (zip [1..] xs) where
  (m, n) = (length xs, length ys)
  f tab (i, x) = tab |> (foldl longer (singleton 0) (zip [1..] ys)) where
    longer r (j, y) = r |> if x == y
                      then 1 + (tab `index` (i-1) `index` (j-1))
                      else max (tab `index` (i-1) `index` j) (r `index` (j-1))
  construct tab = get (reverse xs, m) (reverse ys, n) where
    get ([], 0) ([], 0) = []
    get ((x:xs), i) ((y:ys), j)
      | x == y = get (xs, i-1) (ys, j-1) ++ [x]
      | (tab `index` (i-1) `index` j) > (tab `index` i `index` (j-1)) =
                get (xs, i-1) ((y:ys), j)
      | otherwise = get ((x:xs), i) (ys, j-1)
```

Subset sum problem

Dynamic programming does not limit to solve the optimization problem, but can also solve some more general searching problems. Subset sum problem is such an example. Given a set of integers, is there a non-empty subset sums to zero? for example, there are two subsets of $\{11, 64, -82, -68, 86, 55, -88, -21, 51\}$ both sum to zero. One is $\{64, -82, 55, -88, 51\}$, the other is $\{64, -82, -68, 86\}$.

Of course summing to zero is a special case, because sometimes, people want to find a subset, whose sum is a given value s. Here we are going to develop a method to find all the candidate subsets.

There is obvious a brute-force exhausting search solution. For every element, we can either pick it or not. So there are total 2^n options for set with n elements. Because for every selection, we need check if it sums to s. This is a linear operation. The overall complexity is bound to $O(n2^n)$. This is the exponential algorithm, which takes very huge time if the set is big.

There is a recursive solution to subset sum problem. If the set is empty, there is no solution definitely; Otherwise, let the set is $X = \{x_1, x_2, ...\}$. If $x_1 = s$, then subset $\{x_1\}$ is a solution, we need next search for subsets $X' = \{x_2, x_3, ...\}$ for those sum to s; Otherwise if $x_1 \neq s$, there are two different kinds of possibilities. We need search X' for both sum s, and sum $s - x_1$. For any subset sum to $s - x_1$, we can add x_1 to it to form a new set as a solution. The following

www.dbooks.org

equation defines this algorithm.

$$solve(X, s) = \begin{cases} \phi & : & X = \phi \\ \{\{x_1\}\} \cup solve(X', s) & : & x_1 = s \\ solve(X', s) \cup \{\{x_1\} \cup S | S \in solve(X', s - x_1)\} & : & otherwise \end{cases} \tag{13.104}$$

There are clear substructures in this definition, although they are not in a sense of optimal. And there are also overlapping sub-problems. This indicates the problem can be solved with dynamic programming with a table to memorize the solutions to sub-problems.

Instead of developing a solution to output all the subsets directly, let's consider how to give the existence answer firstly. That output 'yes' if there exists some subset sum to s, and 'no' otherwise.

One fact is that, the upper and lower limit for all possible answer can be calculated in one scan. If the given sum s doesn't belong to this range, there is no solution obviously.

$$\begin{cases} s_l = \sum\{x \in X, x < 0\} \\ s_u = \sum\{x \in X, x > 0\} \end{cases} \tag{13.105}$$

Otherwise, if $s_l \leq s \leq s_u$, since the values are all integers, we can use a table, with $s_u - s_l + 1$ columns, each column represents a possible value in this range, from s_l to s_u. The value of the cell is either true or false to represents if there exists subset sum to this value. All cells are initialized as false. Starts from the first element x_1 in X, definitely, set $\{x_1\}$ can sum to x_1, so that the cell represents this value in the first row can be filled as true.

	s_l	$s_l + 1$	...	x_1	...	s_u
x_1	F	F	...	T	...	F

With the next element x_2, There are three possible sums. Similar as the first row, $\{x_2\}$ sums to x_2; For all possible sums in previous row, they can also been achieved without x_2. So the cell below to x_1 should also be filled as true; By adding x_2 to all possible sums so far, we can also get some new values. That the cell represents $x_1 + x_2$ should be true.

	s_l	$s_l + 1$	...	x_1	...	x_2	...	$x_1 + x_2$	...	s_u
x_1	F	F	...	T	...	F	...	F	...	F
x_2	F	F	...	T	...	T	...	T	...	F

Generally speaking, when fill the i-th row, all the possible sums constructed with $\{x_1, x_2, ..., x_{i-1}\}$ so far can also be achieved with x_i. So the cells previously are true should also be true in this new row. The cell represents value x_i should also be true since the singleton set $\{x_i\}$ sums to it. And we can also adds x_i to all previously constructed sums to get the new results. Cells represent these new sums should also be filled as true.

When all the elements are processed like this, a table with $|X|$ rows is built. Looking up the cell represents s in the last row tells if there exists subset can sum to this value. As mentioned above, there is no solution if $s < s_l$ or $s_u < s$. We skip handling this case for the sake of brevity.

```
1: function SUBSET-SUM(X, s)
2:     s_l ← Σ{x ∈ X, x < 0}
3:     s_u ← Σ{x ∈ X, x > 0}
4:     n ← |X|
```

```
5:    T ← {{False, False, ...}, {False, False, ...}, ...}        ▷ n × (s_u − s_l + 1)
6:    for i ← 1 to n do
7:        for j ← s_l to s_u do
8:            if X[i] = j then
9:                T[i][j] ← True
10:           if i > 1 then
11:               T[i][j] ← T[i][j] ∨ T[i − 1][j]
12:               j' ← j − X[i]
13:               if s_l ≤ j' ≤ s_u then
14:                   T[i][j] ← T[i][j] ∨ T[i − 1][j']
15:   return T[n][s]
```

Note that the index to the columns of the table, doesn't range from 1 to $s_u - s_l + 1$, but maps directly from s_l to s_u. Because most programming environments don't support negative index, this can be dealt with $T[i][j - s_l]$. The following example Python program utilizes the property of negative indexing.

```
def solve(xs, s):
    low = sum([x for x in xs if x < 0])
    up  = sum([x for x in xs if x > 0])
    tab = [[False]*(up-low+1) for _ in xs]
    for i in xrange(0, len(xs)):
        for j in xrange(low, up+1):
            tab[i][j] = (xs[i] == j)
            j1 = j - xs[i];
            tab[i][j] = tab[i][j] or tab[i-1][j] or
                        (low <= j1 and j1 <= up and tab[i-1][j1])
    return tab[-1][s]
```

Note that this program doesn't use different branches for $i = 0$ and $i = 1, 2, ..., n - 1$. This is because when $i = 0$, the row index to $i - 1 = -1$ refers to the last row in the table, which are all false. This simplifies the logic one more step.

With this table built, it's easy to construct all subsets sum to s. The method is to look up the last row for cell represents s. If the last element $x_n = s$, then $\{x_n\}$ definitely is a candidate. We next look up the previous row for s, and recursively construct all the possible subsets sum to s with $\{x_1, x_2, x_3, ..., x_{n-1}\}$. Finally, we look up the second last row for cell represents $s - x_n$. And for every subset sums to this value, we add element x_n to construct a new subset, which sums to s.

```
1: function GET(X, s, T, n)
2:     S ← φ
3:     if X[n] = s then
4:         S ← S ∪ {X[n]}
5:     if n > 1 then
6:         if T[n − 1][s] then
7:             S ← S ∪ GET(X, s, T, n − 1)
8:         if T[n − 1][s − X[n]] then
9:             S ← S ∪ {{X[n]} ∪ S' | S' ∈ GET(X, s − X[n], T, n − 1) }
10:    return S
```

The following Python example program translates this algorithm.

www.dbooks.org

```
def get(xs, s, tab, n):
    r = []
    if xs[n] == s:
        r.append([xs[n]])
    if n > 0:
        if tab[n-1][s]:
            r = r + get(xs, s, tab, n-1)
        if tab[n-1][s - xs[n]]:
            r = r + [[xs[n]] + ys for ys in get(xs, s - xs[n], tab, n-1)]
    return r
```

This dynamic programming solution to subset sum problem loops $O(n(s_u - s_l + 1))$ times to build the table, and recursively uses $O(n)$ time to construct the final solution from this table. The space it used is also bound to $O(n(s_u - s_l + 1))$.

Instead of using table with n rows, a vector can be used alternatively. For every cell represents a possible sum, the list of subsets are stored. This vector is initialized to contain all empty sets. For every element in X, we update the vector, so that it records all the possible sums which can be built so far. When all the elements are considered, the cell corresponding to s contains the final result.

```
1: function SUBSET-SUM(X, s)
2:     s_l ← Σ{x ∈ X, x < 0}
3:     s_u ← Σ{x ∈ X, x > 0}
4:     T ← {φ, φ, ...}                                  ▷ s_u − s_l + 1
5:     for x ∈ X do
6:         T' ← DUPLICATE(T)
7:         for j ← s_l to s_u do
8:             j' ← j − x
9:             if x = j then
10:                T'[j] ← T'[j] ∪ {x}
11:            if s_l ≤ j' ≤ s_u ∧ T[j'] ≠ φ then
12:                T'[j] ← T'[j] ∪ {{x} ∪ S|S ∈ T[j']}
13:        T ← T'
14:    return T[s]
```

The corresponding Python example program is given as below.

```
def subsetsum(xs, s):
    low = sum([x for x in xs if x < 0])
    up  = sum([x for x in xs if x > 0])
    tab = [[] for _ in xrange(low, up+1)]
    for x in xs:
        tab1 = tab[:]
        for j in xrange(low, up+1):
            if x == j:
                tab1[j].append([x])
            j1 = j - x
            if low <= j1 and j1 <= up and tab[j1] != []:
                tab1[j] = tab1[j] + [[x] + ys for ys in tab[j1]]
        tab = tab1
    return tab[s]
```

This imperative algorithm shows a clear structure, that the solution table is

built by looping every element. This can be realized in purely functional way by folding. A finger tree can be used to represents the vector spans from s_l to s_u. It is initialized with all empty values as in the following equation.

$$subsetsum(X, s) = fold(build, \{\phi, \phi, ..., \}, X)[s] \tag{13.106}$$

After folding, the solution table is built, the answer is looked up at cell s[13].

For every element $x \in X$, function *build* folds the list $\{s_l, s_l + 1, ..., s_u\}$, with every value j, it checks if it equals to x and appends the singleton set $\{x\}$ to the j-th cell. Not that here the cell is indexed from s_l, but not 0. If the cell corresponding to $j - x$ is not empty, the candidate solutions stored in that place are also duplicated and add element x is added to every solution.

$$build(T, x) = fold(f, T, \{s_l, s_l + 1, ..., s_u\}) \tag{13.107}$$

$$f(T, j) = \begin{cases} T'[j] \cup \{\{x\} \cup Y | Y \in T[j']\} & : \quad s_l \leq j' \leq s_u \wedge T[j'] \neq \phi, j' = j - x \\ T' & : \quad otherwise \end{cases} \tag{13.108}$$

Here the adjustment is applied on T', which is another adjustment to T as shown as below.

$$T' = \begin{cases} \{x\} \cup T[j] & : \quad x = j \\ T & : \quad otherwise \end{cases} \tag{13.109}$$

Note that the first clause in both equation (13.108) and (13.109) return a new table with certain cell being updated with the given value.

The following Haskell example program implements this algorithm.

```
subsetsum xs s = foldl build (fromList [[] | _ ← [l..u]]) xs `idx` s where
  l = sum $ filter (< 0) xs
  u = sum $ filter (> 0) xs
  idx t i = index t (i - l)
  build tab x = foldl (λt j → let j' = j - x in
                  adjustIf (l ≤ j' && j' ≤ u && tab `idx` j' /= [])
                           (++ [(x:ys) | ys ← tab `idx` j']) j
                           (adjustIf (x == j) ([x]:) j t)) tab [l..u]
  adjustIf pred f i seq = if pred then adjust f (i - l) seq else seq
```

Some materials like [16] provide common structures to abstract dynamic programming. So that problems can be solved with a generic solution by customizing the precondition, the comparison of candidate solutions for better choice, and the merge method for sub solutions. However, the variety of problems makes things complex in practice. It's important to study the properties of the problem carefully.

Exercise 13.3

- Realize a maze solver by using the stack approach, which can find all the possible paths.

[13] Again, here we skip the error handling to the case that $s < s_l$ or $s > s_u$. There is no solution if s is out of range.

www.dbooks.org

- There are 92 distinct solutions for the 8 queens puzzle. For any one solution, rotating it $90°, 180°, 270°$ gives solutions too. Also flipping it vertically and horizontally also generate solutions. Some solutions are symmetric, so that rotation or flip gives the same one. There are 12 unique solutions in this sense. Modify the program to find the 12 unique solutions. Improve the program, so that the 92 distinct solutions can be found with fewer search.
- Make the 8 queens puzzle solution generic so that it can solve n queens puzzle.
- Make the functional solution to the leap frogs puzzle generic, so that it can solve n frogs case.
- Modify the wolf, goat, and cabbage puzzle algorithm, so that it can find all possible solutions.
- Give the complete algorithm definition to solve the 2 water jugs puzzle with extended Euclid algorithm.
- We needn't the exact linear combination information x and y in fact. After we know the puzzle is solvable by testing with GCD, we can blindly execute the process that: fill A, pour A into B, whenever B is full, empty it till there is expected volume in one jug. Realize this solution. Can this one find faster solution than the original version?
- Compare to the extended Euclid method, the BFS approach is a kind of brute-force searching. Improve the extended Euclid approach by finding the best linear combination which minimize $|x| + |y|$.
- John Horton Conway introduced the sliding tile puzzle. Figure 13.51 shows a simplified verson. There are 8 cells, 7 of them are occupied by pieces labeled from 1 to 7. Each piece can slide to the free cell if they are connected. The line between cells means there is a connectoin. The goal is to reverse the pieces from 1, 2, 3, 4, 5, 6, 7 to 7, 6, 5, 4, 3, 2, 1 by sliding. Develop a program to solve this puzzle.

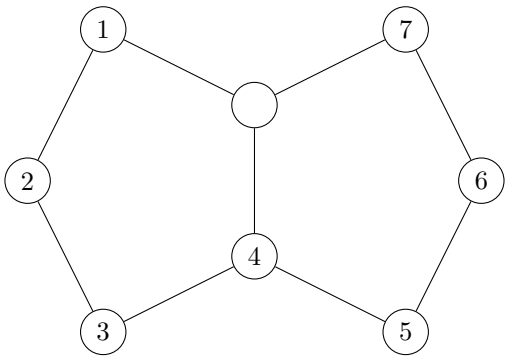


Figure 13.51: Conway sliding puzzle

- Realize the imperative Huffman code table generating algorithm.

- One option to realize the bottom-up solution for the longest common subsequence problem is to record the direction in the table. Thus, instead of storing the length information, three values like 'N', for north, 'W' for west, and 'NW' for northwest are used to indicate how to construct the final result. We start from the bottom-right corner of the table, if the cell value is 'NW', we go along the diagonal by moving to the cell in the upper-left; if it's 'N', we move vertically to the upper row; and move horizontally if it's 'W'. Implement this approach in your favorite programming language.

- Given a list of non-negative integers, find the maximum sum composed by numbers that none of them are adjacent.

- Levenshtein edit distance is defined as the cost of converting from one string s to another string t. It is widely used in spell-checking, OCR correction etc. There are three operations allowed in Levenshtein edit distance. Insert a character; delete a character; and substitute a character. Each operation mutate one character a time. The following exaple shows how to convert string "kitten" to "sitting". The Levenshtein edit distance is 3 in this case.

  1. kitten → sitten (substitution of 's' for 'k');
  2. sitten → sittin (substitution of 'i' for 'e');
  3. sitten → sitting (insertion of 'g' at the end).

  Develop a program to calculate Levenshtein edit distance for two strings with Dynamic Programming.

13.4 Short summary

This chapter introduces the elementary methods about searching. Some of them instruct the computer to scan for interesting information among the data. They often have some structure, that can be updated during the scan. This can be considered as a special case for the information reusing approach. The other commonly used strategy is divide and conquer, that the scale of the search domain is kept decreasing till some obvious result. This chapter also explains methods to search for solutions among domains. The solutions typically are not the elements being searched. They can be a series of decisions or some operation arrangement. If there are multiple solutions, sometimes, people want to find the optimized one. For some spacial cases, there exist simplified approach such as the greedy methods. And dynamic programming can be used for more wide range of problems when they shows optimal substructures.

www.dbooks.org

Bibliography

[1] Donald E. Knuth. “The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)”. Addison-Wesley Professional; 2 edition (May 4, 1998) ISBN-10: 0201896850 ISBN-13: 978-0201896855

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. ISBN:0262032937. The MIT Press. 2001

[3] M. Blum, R.W. Floyd, V. Pratt, R. Rivest and R. Tarjan, ”Time bounds for selection,” J. Comput. System Sci. 7 (1973) 448-461.

[4] Jon Bentley. “Programming pearls, Second Edition”. Addison-Wesley Professional; 1999. ISBN-13: 978-0201657883

[5] Richard Bird. “Pearls of functional algorithm design”. Chapter 3. Cambridge University Press. 2010. ISBN, 1139490605, 9781139490603

[6] Edsger W. Dijkstra. “The saddleback search”. EWD-934. 1985. http://www.cs.utexas.edu/users/EWD/index09xx.html.

[7] Robert Boyer, and Strother Moore. “MJRTY - A Fast Majority Vote Algorithm”. Automated Reasoning: Essays in Honor of Woody Bledsoe, Automated Reasoning Series, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991, pp. 105-117.

[8] Cormode, Graham; S. Muthukrishnan (2004). “An Improved Data Stream Summary: The Count-Min Sketch and its Applications”. J. Algorithms 55: 29¨C38.

[9] Knuth Donald, Morris James H., jr, Pratt Vaughan. “Fast pattern matching in strings”. SIAM Journal on Computing 6 (2): 323¨C350. 1977.

[10] Robert Boyer, Strother Moore. “A Fast String Searching Algorithm”. Comm. ACM (New York, NY, USA: Association for Computing Machinery) 20 (10): 762¨C772. 1977

[11] R. N. Horspool. “Practical fast searching in strings”. Software - Practice & Experience 10 (6): 501¨C506. 1980.

[12] Wikipedia. “Boyer-Moore string search algorithm”. http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm

[13] Wikipedia. “Eight queens puzzle”. http://en.wikipedia.org/wiki/Eight_queens_puzzle

www.dbooks.org

[14] George Pólya. “How to solve it: A new aspect of mathematical method”. Princeton University Press(April 25, 2004). ISBN-13: 978-0691119663

[15] Wikipedia. “David A. Huffman”. http://en.wikipedia.org/wiki/David_A._Huffman

[16] Fethi Rabhi, Guy Lapalme “Algorithms: a functional programming approach”. Second edition. Addison-Wesley.

Part VI

Appendix

www.dbooks.org

Appendix A

Lists

A.1 Introduction

This book intensely uses recursive list manipulations in purely functional settings. List can be treated as a counterpart to array in imperative settings, which are the bricks to many algorithms and data structures.

For the readers who are not familiar with functional list manipulation, this appendix provides a quick reference. All operations listed in this appendix are not only described in equations, but also implemented in both functional programming languages as well as imperative languages as examples.

Besides the elementary list operations, this appendix also contains explanation of some high order function concepts such as mapping, folding etc.

A.2 List Definition

Like arrays in imperative settings, lists play a critical role in functional setting[1]. Lists are built-in support in some programming languages like Lisp families and ML families so it needn't explicitly define list in those environment.

List, or more precisely, singly linked-list is a data structure that can be described below.

- A *list* is either empty;
- Or contains an element and a *list*.

Note that this definition is recursive. Figure A.1 illustrates a list with n nodes. Each node contains two part, a key element and a sub list. The sub list contained in the last node is empty, which is denoted as 'NIL'.

This data structure can be explicitly defined in programming languages support record (or compound type) concept. The following ISO C++ code defines list[2].

[1] Some reader may argue that 'lambda calculus plays the most critical role'. Lambda calculus is somewhat as assembly languages to the computation world, which is worthy studying from the essence of computation model to the practical programs. However, we don't dive into the topic in this book. Users can refer to [4] for detail.

[2] We only use template to parameterize the type of the element in this chapter. Except this point, all imperative source code are in ANSI C style to avoid language specific features.

www.dbooks.org

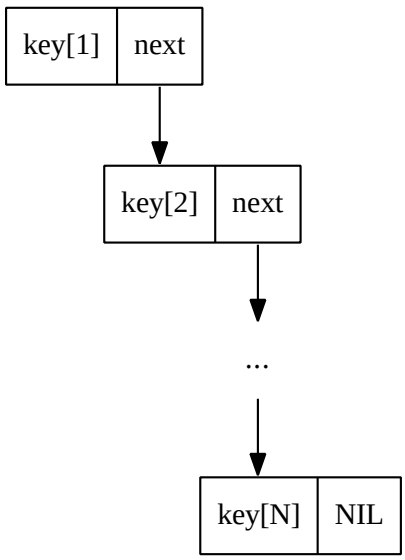


Figure A.1: A list contains n nodes

```
template<typename T>
struct List {
  T key;
  List* next;
};
```

A.2.1 Empty list

It is worth to mention about 'empty' list a bit more in detail. In environment supporting the nil concept, for example, C or java like programming languages, empty list can have two different representations. One is the trivial 'NIL' (or null, or 0, which varies from languages); the other is an non-NIL empty list as {}, the latter is typically allocated with memory but filled with nothing. In Lisp dialects, the empty is commonly written as `'()`. In ML families, it's written as `[]`. We use ϕ to denote empty list in equations and use 'NIL' in pseudo code sometimes to describe algorithms in this book.

A.2.2 Access the element and the sub list

Given a list L, two functions can be defined to access the element stored in it and the sub list respectively. They are typically denoted as $first(L)$, and $rest(L)$ or $head(L)$ and $tail(L)$ for the same meaning. These two functions are named as `car` and `cdr` in Lisp for historic reason about the design of machine registers [5]. In languages support Pattern matching (e.g. ML families, Prolog and Erlang etc.) These two functions are commonly realized by matching the `cons` which we'll introduced later. for example the following Haskell program:

```
head (x:xs) = x
tail (x:xs) = xs
```

If the list is defined in record syntax like what we did above, these two functions can be realized by accessing the record fields [3].

```
template<typename T>
T first(List<T> *xs) { return xs->key; }

template<typename T>
List<T>* rest(List<T>* xs) { return xs->next; }
```

In this book, L' is used to denote the $rest(L)$ sometimes, also we uses l_1 to represent $first(L)$ in the context that the list is literately given in form $L = \{l_1, l_2, ..., l_n\}$.

More interesting, as far as in an environment support recursion, we can define List. The following example define a list of integers in C++ compile time.

```
struct Empty;

template<int x, typename T> struct List {
  static const int first = x;
  typedef T rest;
};
```

This line constructs a list of $\{1, 2, 3, 4, 5\}$ in compile time.

```
typedef List<1, List<2, List<3, List<4 List<5, Empty> > > > > A;
```

A.3 Basic list manipulation

A.3.1 Construction

The last C++ template meta programming example actually shows literate construction of a list. A list can be constructed from an element with a sub list, where the sub list can be empty. We denote function $cons(x, L)$ as the constructor. This name is used in most Lisp dialects. In ML families, there are 'cons' operator defined as `::`, (in Haskell it's `:`).

We can define `cons` to create a record as we defined above in ISO C++, for example[4].

```
template<typename T>
List<T>* cons(T x, List<T>* xs) {
  List<T>* lst = new List<T>;
  lst->key = x;
  lst->next = xs;
  return lst;
}
```

[3] They can be also named as 'key' and 'next' or be defined as class methods.

[4] It is often defined as a constructor method for the class template, However, we define it as a standalone function for illustration purpose.

www.dbooks.org

A.3.2 Empty testing and length calculating

It is trivial to test if a list is empty. If the environment contains nil concept, the testing should also handle nil case. Both Lisp dialects and ML families provide `null` testing functions. Empty testing can also be realized by pattern-matching with empty list if possible. The following Haskell program shows such example.

```
null [] = True
null _ = False
```

In this book we will either use $empty(L)$ or $L = \phi$ where empty testing happens.

With empty testing defined, it's possible to calculate length for a list. In imperative settings, LENGTH is often implemented like the following.

```
1: function LENGTH(L)
2:     n ← 0
3:     while L ≠ NIL do
4:         n ← n + 1
5:         L ← NEXT(L)
6:     return n
```

This ISO C++ code translates the algorithm to real program.

```
template<typename T>
int length(List<T>* xs) {
  int n = 0;
  for (; xs; ++n, xs = xs->next);
  return n
}
```

However, in purely functional setting, we can't mutate a counter variable. the idea is that, if the list is empty, then its size is zero; otherwise, we can recursively calculate the length of the sub list, then add it by one to get the length of this list.

$$length(L) = \begin{cases} 0 & : \quad L = \phi \\ 1 + length(L') & : \quad otherwise \end{cases} \tag{A.1}$$

Here $L' = rest(L)$ as mentioned above, it's $\{l_2, l_3, ..., l_n\}$ for list contains n elements. Note that both L and L' can be empty ϕ. In this equation, we also use '=' to test if list L is empty. In order to know the length of a list, we need traverse all the elements from the head to the end, so that this algorithm is proportion to the number of elements stored in the list. It is a linear algorithm bound to $O(n)$ time.

Below are two programs in Haskell and in Scheme/Lisp realize this recursive algorithm.

```
length [] = 0
length (x:xs) = 1 + length xs
```

```
(define (length lst)
  (if (null? lst) 0 (+ 1 (length (cdr lst)))))
```

How to testing if two lists are identical is left as exercise to the reader.

A.3.3 indexing

One big difference between array and list (singly-linked list accurately) is that array supports random access. Many programming languages support using `x[i]` to access the i-th element stored in array in constant $O(1)$ time. The index typically starts from 0, but it's not the all case. Some programming languages using 1 as the first index. In this appendix, we treat index starting from 0. However, we must traverse the list with i steps to reach the target element. The traversing is quite similar to the length calculation. Thus it's commonly expressed as below in imperative settings.

```
1: function GET-AT(L, i)
2:     while i ≠ 0 do
3:         L ← NEXT(L)
4:         i ← i − 1
5:     return FIRST(L)
```

Note that this algorithm doesn't handle the error case such that the index isn't within the bound of the list. We assume that $0 \leq i < |L|$, where $|L| = length(L)$. The error handling is left as exercise to the reader. The following ISO C++ code is a line-by-line translation of this algorithm.

```
template<typename T>
T getAt(List<T>* lst, int n) {
  while(n--)
    lst = lst->next;
  return lst->key;
}
```

However, in purely functional settings, we turn to recursive traversing instead of while-loop.

$$getAt(L, i) = \begin{cases} First(L) & : \quad i = 0 \\ getAt(Rest(L), i - 1) & : \quad otherwise \end{cases} \tag{A.2}$$

In order to *get the i-th element*, the algorithm does the following:

- if i is 0, then we are done, the result is the first element in the list;
- Otherwise, the result is to *get the $(i-1)$-th element* from the sub-list.

This algorithm can be translated to the following Haskell code.

```
getAt i (x:xs) = if i == 0 then x else getAt i-1 xs
```

Note that we are using pattern matching to ensure the list isn't empty, which actually handles all out-of-bound cases with un-matched pattern error. Thus if $i > |L|$, we finally arrive at a edge case that the index is $i - |L|$, while the list is empty; On the other hand, if $i < 0$, minus it by one makes it even farther away from 0. We finally end at the same error that the index is some negative, while the list is empty;

The indexing algorithm takes time proportion to the value of index, which is bound to $O(i)$ linear time. This section only address the read semantics. How to mutate the element at a given position is explained in later section.

www.dbooks.org

A.3.4 Access the last element

Although accessing the first element and the rest list L' is trivial, the opposite operations, that retrieving the last element and the initial sub list need linear time without using a tail pointer. If the list isn't empty, we need traverse it till the tail to get these two components. Below are their imperative descriptions.

```
1: function LAST(L)
2:     x ← NIL
3:     while L ≠ NIL do
4:         x ← FIRST(L)
5:         L ← REST(L)
6:     return x

7: function INIT(L)
8:     L' ← NIL
9:     while REST(L) ≠ NIL do
10:        L' ← APPEND(L', FIRST(L))
11:        L ← REST(L)
12:    return L'
```

The algorithm assumes that the input list isn't empty, so the error handling is skipped. Note that the INIT() algorithm uses the appending algorithm which will be defined later.

Below are the corresponding ISO C++ implementation. The optimized version by utilizing tail pointer is left as exercise.

```
template<typename T>
T last(List<T>* xs) {
  T x; /* Can be set to a special value to indicate empty list err. */
  for (; xs; xs = xs->next)
    x = xs->key;
  return x;
}

template<typename T>
List<T>* init(List<T>* xs) {
  List<T>* ys = NULL;
  for (; xs->next; xs = xs->next)
    ys = append(ys, xs->key);
  return ys;
}
```

While these two algorithms can be implemented in purely recursive manner as well. When we want to access *the last element*.

- If the list contains only one element (the rest sub-list is empty), the result is this very element;
- Otherwise, the result is *the last element* of the rest sub-list.

$$last(L) = \begin{cases} First(L) & : & Rest(L) = \phi \\ last(Rest(L)) & : & otherwise \end{cases} \tag{A.3}$$

The similar approach can be used to *get a list contains all elements except for the last one.*

- The edge case: If the list contains only one element, then the result is an empty list;
- Otherwise, we can first *get a list contains all elements except for the last one* from the rest sub-list, then construct the final result from the first element and this intermediate result.

$$init(L) = \begin{cases} \phi & : \quad L' = \phi \\ cons(l_1, init(L')) & : \quad otherwise \end{cases} \tag{A.4}$$

Here we denote l_1 as the first element of L, and L' is the rest sub-list. This recursive algorithm needn't use appending, It actually construct the final result list from right to left. We'll introduce a high-level concept of such kind of computation later in this appendix.

Below are Haskell programs implement $last()$ and $init()$ algorithms by using pattern matching.

```
last [x] = x
last (_:xs) = last xs

init [x] = []
init (x:xs) = x : init xs
```

Where `[x]` matches the singleton list contains only one element, while `(_:xs)` matches any non-empty list, and the underscore (`_`) is used to indicate that we don't care about the element. For the detail of pattern matching, readers can refer to any Haskell tutorial materials, such as [8].

A.3.5 Reverse indexing

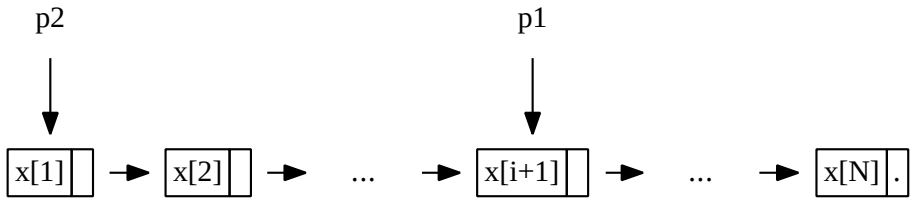
Reverse indexing is a general case for $last()$, finding the last i-th element in a singly-linked list with the minimized memory spaces is interesting, and this problem is often used in technical interview in some companies. A naive implementation takes 2 rounds of traversing, the first round is to determine the length of the list n, then, calculate the left-hand index by $n - i - 1$. Finally a second round of traverse is used to access the element with the left-hand index. This idea can be give as the following equation.

$$getAtR(L, i) = getAt(L, length(L) - i - 1)$$

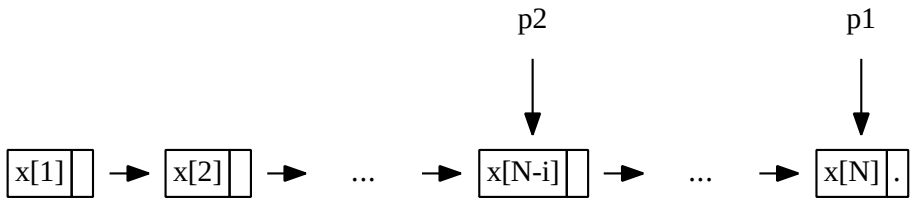
There exists better imperative solution. For illustration purpose, we omit the error cases such as index is out-of-bound etc. The idea is to keep two pointers p_1, p_2, with the distance of i between them, that $rest^i(p_2) = p_1$, where $rest^i(p_2)$ means repleatedly apply $rest()$ function i times. It says that succeeds i steps from p_2 gets p_1. We can start p_2 from the head of the list and advance the two pointers in parallel till one of them (p_1) arrives at the end of the list. At that time point, pointer p_2 exactly arrived at the i-th element from right. Figure A.2 illustrates this idea.

It is straightforward to realize the imperative algorithm based on this 'double pointers' solution.

www.dbooks.org



(a) p_2 starts from the head, which is behind p_1 in i steps.



(b) When p_1 reaches the end, p_2 points to the i-th element from right.

Figure A.2: Double pointers solution to reverse indexing.

```
1: function GET-AT-R(L, i)
2:     p ← L
3:     while i ≠ 0 do
4:         L ← REST(L)
5:         i ← i − 1
6:     while REST(L) ≠ NIL do
7:         L ← REST(L)
8:         p ← REST(p)
9:     return FIRST(p)
```

The following ISO C++ code implements the 'double pointers' right indexing algorithm.

```
template<typename T>
T getAtR(List<T>* xs, int i) {
  List<T>* p = xs;
  while(i--)
    xs = xs->next;
  for(; xs->next; xs = xs->next, p = p->next);
  return p->key;
}
```

The same idea can be realized recursively as well. If we want to access the i-th element of list L from right, we can examine the two lists L and $S = \{l_i, l_{i+1}, ..., l_n\}$ simultaneously, where S is a sub-list of L without the first $i-1$ elements.

- The edge case: If S is a singleton list, then the i-th element from right is the first element in L;

- Otherwise, we drop the first element from L and S, and recursively examine L' and S'.

This algorithm description can be formalized as the following equations.

$$getAtR(L, i) = examine(L, drop(i, L)) \tag{A.5}$$

Where function $examine(L, S)$ is defined as below.

$$examine(L, S) = \begin{cases} first(L) & : & |S| = 1 \\ examine(rest(L), rest(S)) & : & otherwise \end{cases} \tag{A.6}$$

We'll explain the detail of *drop*() function in later section about list mutating operations. Here it can be implemented as repeatedly call *rest*() with specified times.

$$drop(n, L) = \begin{cases} L & : & n = 0 \\ drop(n-1, rest(L)) & : & otherwise \end{cases}$$

Translating the equations to Haskell yields this example program.

```
atR :: [a] → Int → a
atR xs i = get xs (drop i xs) where
  get (x:_) [_] = x
  get (_:xs) (_:ys) = get xs ys
  drop n as@(_:as') = if n == 0 then as else drop (n-1) as'
```

Here we use dummy variable `_` as the placeholders for components we don't care.

A.3.6 Mutating

Strictly speaking, we can't mutate the list at all in purely functional settings. Unlike in imperative settings, mutate is actually realized by creating new list. Almost all functional environments support garbage collection, the original list may either be persisted for reusing, or released (dropped) at sometime (Chapter 2 in [6]).

Appending

Function *cons* can be viewed as building list by insertion element always on head. If we chains multiple *cons* operations, it can repeatedly construct a list from right to the left. Appending on the other hand, is an operation adding element to the tail. Compare to *cons* which is trivial constant time $O(1)$ operation, We must traverse the whole list to locate the appending position. It means that appending is bound to $O(n)$, where n is the length of the list. In order to speed up the appending, imperative implementation typically uses a field (variable) to record the tail position of a list, so that the traversing can be avoided. However, in purely functional settings we can't use such 'tail' pointer. The appending has to be realized in recursive manner.

$$append(L, x) = \begin{cases} \{x\} & : & L = \phi \\ cons(first(L), append(rest(L), x)) & : & otherwise \end{cases} \tag{A.7}$$

www.dbooks.org

That the algorithm handles two different appending cases:

- If the list is empty, the result is a singleton list contains x, which is the element to be appended. The singleton list notion $\{x\} = cons(x, \phi)$, is a simplified form of *cons* the element with an empty list ϕ;
- Otherwise, for the none-empty list, the result can be achieved by first appending the element x to the rest sub-list, then construct the first element of L with the recursive appending result.

For the none-trivial case, if we denote $L = \{l_1, l_2, ...\}$, and $L' = \{l_2, l_3, ...\}$ the equation can be written as.

$$append(L, x) = \begin{cases} \{x\} & : & L = \phi \\ cons(l_1, append(L', x)) & : & otherwise \end{cases} \tag{A.8}$$

We'll use both forms in the rest of this appendix.

The following Scheme/Lisp program implements this algorithm.

```
(define (append lst x)
  (if (null? lst)
      (list x)
      (cons (car lst) (append (cdr lst) x))))
```

Even without the tail pointer, it's possible to traverse the list imperatively and append the element at the end.

```
1: function APPEND(L, x)
2:     if L = NIL then
3:         return CONS(x, NIL)
4:     H ← L
5:     while REST(L) ≠ NIL do
6:         L ← REST(L)
7:     REST(L) ← CONS(x, NIL)
8:     return H
```

The following ISO C++ programs implements this algorithm. How to utilize a tail field to speed up the appending is left as exercise to the reader for interesting.

```
template<typename T>
List<T>* append(List<T>* xs, T x) {
  List<T> *tail, *head;
  for (head = tail = xs; xs; xs = xs→next)
    tail = xs;
  if (!head)
    head = cons<T>(x, NULL);
  else
    tail→next = cons<T>(x, NULL);
  return head;
}
```

Mutate element at a given position

Although we have defined random access algorithm $getAt(L, i)$, we can't just mutate the element returned by this function in a sense of purely functional settings. It is quite common to provide reference semantics in imperative programming languages and in some 'almost' functional environment. Readers can refer to [4] for detail. For example, the following ISO C++ example returns a reference instead of a value in indexing program.

```
template<typename T>
T& getAt(List<T>* xs, int n) {
  while (n--)
    xs = xs->next;
  return xs->key;
}
```

So that we can use this function to mutate the 2nd element as below.

```
List<int>* xs = cons(1, cons(2, cons<int>(3, NULL)));
getAt(xs, 1) = 4;
```

In an impure functional environment, such as Scheme/Lisp, to set the i-th element to a given value can be implemented by mutate the referenced cell directly as well.

```
(define (set-at! lst i x)
  (if (= i 0)
      (set-car! lst x)
      (set-at! (cdr lst) (- i 1) x)))
```

This program first checks if the index i is zero, if so, it mutate the first element of the list to given value x; otherwise, it deduces the index i by one, and tries to mutate the rest of the list at this new index with value x. This function doesn't return meaningful value. It is for use of side-effect. For instance, the following code mutates the 2nd element in a list.

```
(define lst '(1 2 3 4 5))
(set-at! lst 1 4)
(display lst)
```

```
(1 4 3 4 5)
```

In order to realize a purely functional $setAt(L, i, x)$ algorithm, we need avoid directly mutating the cell, but creating a new one:

- Edge case: If we want to set the value of the first element ($i = 0$), we construct a new list, with the new value and the sub-list of the previous one;
- Otherwise, we construct a new list, with the previous first element, and a new sub-list, which has the $(i-1)$-th element set with the new value.

This recursive description can be formalized by the following equation.

$$setAt(L, i, x) = \begin{cases} cons(x, L') & : & i = 0 \\ cons(l_1, setAt(L', i-1, x)) & : & otherwise \end{cases} \tag{A.9}$$

www.dbooks.org

Comparing the below Scheme/Lisp implementation to the previous one reveals the difference from imperative mutating.

```
(define (set-at lst i x)
  (if (= i 0)
      (cons x (cdr lst))
      (cons (car lst) (set-at (cdr lst) (- i 1) x))))
```

Here we skip the error handling for out-of-bound error etc. Again, similar to the random access algorithm, the performance is bound to linear time, as traverse is need to locate the position to set the value.

insertion

There are two semantics about list insertion. One is to insert an element at a given position, which can be denoted as $insert(L, i, x)$. The algorithm is close to $setAt(L, i, x)$; The other is to insert an element to a sorted list, so that the the result list is still sorted.

Let's first consider how to insert an element x at a given position i. The obvious thing is that we need firstly traverse i elements to get to the position, the rest of work is to construct a new sub-list with x being the head of this sub-list. Finally, we construct the whole result by attaching this new sub-list to the end of the first i elements.

The algorithm can be described accordingly to this idea. If we want to insert an element x to a list L at i.

- Edge case: If i is zero, then the insertion turns to be a trivial 'cons' operation – $cons(x, L)$;
- Otherwise, we recursively *insert* x to the sub-list L' at position $i - 1$; then construct the first element with this result.

Below equation formalizes the insertion algorithm.

$$insert(L, i, x) = \begin{cases} cons(x, L) & : & i = 0 \\ cons(l_1, insert(L', i - 1, x)) & : & otherwise \end{cases} \tag{A.10}$$

The following Haskell program implements this algorithm.

```
insert xs 0 y = y:xs
insert (x:xs) i y = x : insert xs (i-1) y
```

This algorithm doesn't handle the out-of-bound error. However, we can interpret the case, that the position i exceeds the length of the list as appending. Readers can considering about it in the exercise of this section.

The algorithm can also be designed imperatively: If the position is zero, just construct the new list with the element to be inserted as the first one; Otherwise, we record the head of the list, then start traversing the list i steps. We also need an extra variable to memorize the previous position for the later list insertion operation. Below is the pseudo code.

```
1: function INSERT(L, i, x)
2:     if i = 0 then
3:         return CONS(x, L)
```

```
4:    H ← L
5:    p ← L
6:    while i ≠ 0 do
7:        p ← L
8:        L ← Rest(L)
9:        i ← i − 1
10:   Rest(p) ← Cons(x, L)
11:   return H
```

And the ISO C++ example program is given by translating this algorithm.

```
template<typename T>
List<T>* insert(List<T>* xs, int i , int x) {
  List<T> *head, *prev;
  if (i == 0)
    return cons(x, xs);
  for (head = xs; i; --i, xs = xs->next)
    prev = xs;
  prev->next = cons(x, xs);
  return head;
}
```

If the list L is sorted, that is for any position $1 \leq i \leq j \leq n$, we have $l_i \leq l_j$. We can design an algorithm which inserts a new element x to the list, so that the result list is still sorted.

$$insert(x, L) = \begin{cases} cons(x, \phi) & : & L = \phi \\ cons(x, L) & : & x < l_1 \\ cons(l_1, insert(x, L')) & : & otherwise \end{cases} \tag{A.11}$$

The idea is that, to insert an element x to a sorted list L:

- If either L is empty or x is less than the first element in L, we just put x in front of L to construct the result;
- Otherwise, we recursively insert x to the sub-list L'.

The following Haskell program implements this algorithm. Note that we use $\leq$, to determine the ordering. Actually this constraint can be loosened to the strict less ($<$), that if elements can be compared in terms of $<$, we can design a program to insert element so that the result list is still sorted. Readers can refer to the chapters of sorting in this book for details about ordering.

```
insert y [] = [y]
insert y xs@(x:xs') = if y ≤ x then y : xs else x : insert y xs'
```

Since the algorithm need compare the elements one by one, it's also a linear time algorithm. Note that here we use the 'as' notion for pattern matching in Haskell. Readers can refer to [8] and [7] for details.

This ordered insertion algorithm can be designed in imperative manner, for example like the following pseudo code[5].

```
1: function Insert(x, L)
2:     if L = φ ∨ x < First(L) then
```

[5]Reader can refer to the chapter 'The evolution of insertion sort' in this book for a minor different one

www.dbooks.org

```
3:        return CONS(x, L)
4:    H ← L
5:    while REST(L) ≠ φ∧ FIRST(REST(L)) < x do
6:        L ← REST(L)
7:    REST(L) ← CONS(x, REST(L))
8:    return H
```

If either the list is empty, or the new element to be inserted is less than the first element in the list, we can just put this element as the new first one; Otherwise, we record the head, then traverse the list till a position, where x is less than the rest of the sub-list, and put x in that position. Compare this one to the 'insert at' algorithm shown previously, the variable *prev* uses to point to the previous position during traversing is omitted by examine the sub-list instead of current list. The following ISO C++ program implements this algorithm.

```
template<typename T>
List<T>* insert(T x, List<T>* xs) {
  List<T> *head;
  if (!xs || x < xs→key)
    return cons(x, xs);
  for (head = xs; xs→next && xs→next→key < x; xs = xs→next);
  xs→next = cons(x, xs→next);
  return head;
}
```

With this linear time ordered insertion defined, it's possible to implement quadratic time insertion-sort by repeatedly inserting elements to an empty list as formalized in this equation.

$$sort(L) = \begin{cases} \phi & : \quad L = \phi \\ insert(l_1, sort(L')) & : \quad otherwise \end{cases} \tag{A.12}$$

This equation says that if the list to be sorted is empty, the result is also empty, otherwise, we can firstly recursively sort all elements except for the first one, then ordered insert the first element to this intermediate result. The corresponding Haskell program is given as below.

```
isort [] = []
isort (x:xs) = insert x (isort xs)
```

And the imperative linked-list base insertion sort is described in the following. That we initialize the result list as empty, then take the element one by one from the list to be sorted, and ordered insert them to the result list.

```
1: function SORT(L)
2:     L' ← φ
3:     while L ≠ φ do
4:         L' ← INSERT(FIRST(L), L')
5:         L ← REST(L)
6:     return L'
```

Note that, at any time during the loop, the result list is kept sorted. There is a major difference between the recursive algorithm (formalized by the equation) and the procedural one (described by the pseudo code), that the former process the list from right, while the latter from left. We'll see in later section about 'tail-recursion' how to eliminate this difference.

The ISO C++ version of linked-list insertion sort is list like this.

```
template<typename T>
List<T>* isort(List<T>* xs) {
  List<T>* ys = NULL;
  for(; xs; xs = xs→next)
    ys = insert(xs→key, ys);
  return ys;
}
```

There is also a dedicated chapter discusses insertion sort in this book. Please refer to that chapter for more details including performance analysis and fine-tuning.

deletion

In purely functional settings, there is no deletion at all in terms of mutating, the data is persist, what the semantic deletion means is actually to create a 'new' list with all the elements in previous one except for the element being 'deleted'.

Similar to the insertion, there are also two deletion semantics. One is to delete the element at a given position; the other is to find and delete elements of a given value. The first can be expressed as $delete(L, i)$, while the second is $delete(L, x)$.

In order to design the algorithm $delete(L, i)$ (or 'delete at'), we can use the idea which is quite similar to random access and insertion, that we first traverse the list to the specified position, then construct the result list with the elements we have traversed, and all the others except for the next one we haven't traversed yet.

The strategy can be realized in a recursive manner that in order to *delete the i-th element from list L*,

- If i is zero, that we are going to delete the first element of a list, the result is obviously the rest of the list;
- If the list to be removed element is empty, the result is anyway empty;
- Otherwise, we can recursively *delete the $(i-1)$-th element from the sub-list L'*, then construct the final result from the first element of L and this intermediate result.

Note there are two edge cases, and the second case is major used for error handling. This algorithm can be formalized with the following equation.

$$delete(L, i) = \begin{cases} L' & : \quad i = 0 \\ \phi & : \quad L = \phi \\ cons(l_1, delete(L', i-1)) & : \end{cases} \tag{A.13}$$

Where $L' = rest(L), l_1 = first(L)$. The corresponding Haskell example program is given below.

```
del (_:xs) 0 = xs
del [] _ = []
del (x:xs) i = x : del xs (i-1)
```

www.dbooks.org

This is a linear time algorithm as well, and there are also alternatives for implementation, for example, we can first split the list at position $i-1$, to get 2 sub-lists L_1 and L_2, then we can concatenate L_1 and L_2'.

The 'delete at' algorithm can also be realized imperatively, that we traverse to the position by looping:

```
1: function DELETE(L, i)
2:     if i = 0 then
3:         return REST(L)
4:     H ← L
5:     p ← L
6:     while i ≠ 0 do
7:         i ← i − 1
8:         p ← L
9:         L ← REST(L)
10:    REST(p) ← REST(L)
11:    return H
```

Different from the recursive approach, The error handling for out-of-bound is skipped. Besides that the algorithm also skips the handling of resource releasing which is necessary in environments without GC (Garbage collection). Below ISO C++ code for example, explicitly releases the node to be deleted.

```
template<typename T>
List<T>* del(List<T>* xs, int i) {
  List<T> *head, *prev;
  if (i == 0)
    head = xs->next;
  else {
    for (head = xs; i; --i, xs = xs->next)
      prev = xs;
    prev->next = xs->next;
  }
  xs->next = NULL;
  delete xs;
  return head;
}
```

Note that the statement `xs->next = NULL` is necessary if the destructor is designed to release the whole linked-list recursively.

The 'find and delete' semantic can further be represented in two ways, one is just find the first occurrence of a given value, and delete this element from the list; The other is to find *ALL* occurrence of this value, and delete these elements. The later is more general case, and it can be achieved by a minor modification of the former. We left the 'find all and delete' algorithm as an exercise to the reader.

The algorithm can be designed exactly as the term 'find and delete' but not 'find then delete', that the finding and deleting are processed in one pass traversing.

- If the list to be dealt with is empty, the result is obviously empty;
- If the list isn't empty, we examine the first element of the list, if it is identical to the given value, the result is the sub list;

- Otherwise, we keep the first element, and recursively find and delete the element in the sub list with the given value. The final result is a list constructed with the kept first element, and the recursive deleting result.

This algorithm can be formalized by the following equation.

$$delete(L, x) = \begin{cases} \phi & : & L = \phi \\ L' & : & l_1 = x \\ cons(l_1, delete(L', x)) & : & otherwise \end{cases} \tag{A.14}$$

This algorithm is bound to linear time as it traverses the list to find and delete element. Translating this equation to Haskell program yields the below code, note that, the first edge case is handled by pattern-matching the empty list; while the other two cases are further processed by if-else expression.

```
del [] _ = []
del (x:xs) y = if x == y then xs else x : del xs y
```

Different from the above imperative algorithms, which skip the error handling in most cases, the imperative 'find and delete' realization must deal with the problem that the given value doesn't exist.

```
1:  function DELETE(L, x)
2:      if L = φ then                                   ▷ Empty list
3:          return φ
4:      if FIRST(L) = x then
5:          H ← REST(L)
6:      else
7:          H ← L
8:          while L ≠ φ∧ FIRST(L) ≠ x do                ▷ List isn't empty
9:              p ← L
10:             L ← REST(L)
11:         if L ≠ φ then                               ▷ Found
12:             REST(p) ← REST(L)
13:     return H
```

If the list is empty, the result is anyway empty; otherwise, the algorithm traverses the list till either finds an element identical to the given value or to the end of the list. If the element is found, it is removed from the list. The following ISO C++ program implements the algorithm. Note that there are codes release the memory explicitly.

```
template<typename T>
List<T>* del(List<T>* xs, T x) {
  List<T> *head, *prev;
  if (!xs)
    return xs;
  if (xs->key == x)
    head = xs->next;
  else {
    for (head = xs; xs && xs->key != x; xs = xs->next)
      prev = xs;
    if (xs)
      prev->next = xs->next;
  }
```

www.dbooks.org

```
  if (xs) {
    xs→next = NULL;
    delete xs;
  }
  return head;
}
```

concatenate

Concatenation can be considered as a general case for appending, that appending only adds one more extra element to the end of the list, while concatenation adds multiple elements.

However, It will lead to quadratic algorithm if implement concatenation naively by appending, which performs poor. Consider the following equation.

$$concat(L_1, L_2) = \begin{cases} L_1 & : & L_2 = \phi \\ concat(append(L_1, first(L_2)), rest(L_2)) & : & otherwise \end{cases}$$

Note that each appending algorithm need traverse to the end of the list, which is proportion to the length of L_1, and we need do this linear time appending work $|L_2|$ times, so the total performance is $O(|L_1| + (|L_1| + 1) + ... + (|L_1| + |L_2|)) = O(|L_1||L_2| + |L_2|^2)$.

The key point is that the linking operation of linked-list is fast (constant $O(1)$ time), we can traverse to the end of L_1 only one time, and link the second list to the tail of L_1.

$$concat(L_1, L_2) = \begin{cases} L_2 & : & L_1 = \phi \\ cons(first(L_1), concat(rest(L_1), L_2)) & : & otherwise \end{cases} \tag{A.15}$$

This algorithm only traverses the first list one time to get the tail of L_1, and then linking the second list with this tail. So the algorithm is bound to linear $O(|L_1|)$ time.

This algorithm is described as the following.

- If the first list is empty, the concatenate result is the second list;
- Otherwise, we concatenate the second list to the sub-list of the first one, and construct the final result with the first element and this intermediate result.

Most functional languages provide built-in functions or operators for list concatenation, for example in ML families `++` is used for this purpose.

```
[] ++ ys = ys
xs ++ [] = xs
(x:xs) ++ ys = x : xs ++ ys
```

Note we add another edge case that if the second list is empty, we needn't traverse to the end of the first one and perform linking, the result is merely the first list.

In imperative settings, concatenation can be realized in constant $O(1)$ time with the augmented tail record. We skip the detailed implementation for this

method, reader can refer to the source code which can be download along with this appendix.

The imperative algorithm without using augmented tail record can be described as below.

```
1: function CONCAT(L_1, L_2)
2:     if L_1 = φ then
3:         return L_2
4:     if L_2 = φ then
5:         return L_1
6:     H ← L_1
7:     while REST(L_1) ≠ φ do
8:         L_1 ← REST(L_1)
9:     REST(L_1) ← L_2
10:    return H
```

And the corresponding ISO C++ example code is given like this.

```
template<typename T>
List<T>* concat(List<T>* xs, List<T>* ys) {
  List<T>* head;
  if (!xs)
    return ys;
  if (!ys)
    return xs;
  for (head = xs; xs->next; xs = xs->next);
  xs->next = ys;
  return head;
}
```

A.3.7 sum and product

Recursive sum and product

It is common to calculate the sum or product of a list of numbers. They are quite similar in terms of algorithm structure. We'll see how to abstract such structure in later section.

In order to calculate the *sum of a list*:

- If the list is empty, the result is zero;
- Otherwise, the result is the first element plus the *sum of the rest of the list*.

Formalize this description gives the following equation.

$$sum(L) = \begin{cases} 0 & : \quad L = \phi \\ l_1 + sum(L') & : \quad otherwise \end{cases} \tag{A.16}$$

However, we can't merely replace plus to times in this equation to achieve product algorithm, because it always returns zero. We can define the product of empty list as 1 to solve this problem.

$$product(L) = \begin{cases} 1 & : \quad L = \phi \\ l_1 \times product(L') & : \quad otherwise \end{cases} \tag{A.17}$$

www.dbooks.org

The following Haskell program implements sum and product.

```
sum [] = 0
sum (x:xs) = x + sum xs

product [] = 1
product (x:xs) = x * product xs
```

Both algorithms traverse the whole list during calculation, so they are bound to $O(n)$ linear time.

Tail call recursion

Note that both sum and product algorithms actually compute the result from right to left. We can change them to the normal way, that calculate the *accumulated* result from left to right. For example with sum, the result is actually accumulated from 0, and adds element one by one to this accumulated result till all the list is consumed. Such approach can be described as the following.

When accumulate result of a list by summing:

- If the list is empty, we are done and return the accumulated result;
- Otherwise, we take the first element from the list, accumulate it to the result by summing, and go on processing the rest of the list.

Formalize this idea to equation yields another version of sum algorithm.

$$sum'(A, L) = \begin{cases} A & : \quad L = \phi \\ sum'(A + l_1, L') & : \quad otherwise \end{cases} \tag{A.18}$$

And sum can be implemented by calling this function by passing start value 0 and the list as arguments.

$$sum(L) = sum'(0, L) \tag{A.19}$$

The interesting point of this approach is that, besides it calculates the result in a normal order from left to right; by observing the equation of $sum'(A, L)$, we found it needn't remember any intermediate results or states when perform recursion. All such states are either passed as arguments (A for example) or can be dropped (previous elements of the list for example). So in a practical implementation, such kind of recursive function can be optimized by eliminating the recursion at all.

We call such kind of function as 'tail recursion' (or 'tail call'), and the optimization of removing recursion in this case as 'tail recursion optimization'[10] because the recursion happens as the final action in such function. The advantage of tail recursion optimization is that the performance can be greatly improved, so that we can avoid the issue of stack overflow in deep recursion algorithms such as sum and product.

Changing the sum and product Haskell programs to tail-recursion manner gives the following modified programs.

```
sum = sum' 0 where
    sum' acc [] = acc
    sum' acc (x:xs) = sum' (acc + x) xs
```

```
product = product' 1 where
    product' acc [] = acc
    product' acc (x:xs) = product' (acc * x) xs
```

In previous section about insertion sort, we mentioned that the functional version sorts the elements form right, this can also be modified to tail recursive realization.

$$sort'(A, L) = \begin{cases} A & : & L = \phi \\ sort'(insert(l_1, A), L') & : & otherwise \end{cases} \tag{A.20}$$

The the sorting algorithm is just calling this function by passing empty list as the accumulator argument.

$$sort(L) = sort'(\phi, L) \tag{A.21}$$

Implementing this tail recursive algorithm to real program is left as exercise to the reader.

As the end of this sub-section, let's consider an interesting problem, that how to design an algorithm to compute b^n effectively? (refer to problem 1.16 in [5].)

A naive brute-force solution is to repeatedly multiply b for n times from 1, which leads to a linear $O(n)$ algorithm.

```
1: function POW(b, n)
2:     x ← 1
3:     loop n times
4:         x ← x × b
5:     return x
```

Actually, the solution can be greatly improved. Consider we are trying to calculate b^8. By the first 2 iterations in above naive algorithm, we got $x = b^2$. At this stage, we needn't multiply x with b to get b^3, we can directly calculate x^2, which leads to b^4. And if we do this again, we get $(b^4)^2 = b^8$. Thus we only need looping 3 times but not 8 times.

An algorithm based on this idea to compute b^n if $n = 2^m$ for some non-negative integer m can be shown in the following equation.

$$pow(b, n) = \begin{cases} b & : & n = 1 \\ pow(b, \frac{n}{2})^2 & : & otherwise \end{cases}$$

It is easy to extend this divide and conquer algorithm so that n can be any non-negative integer.

- For the trivial case, that n is zero, the result is 1;
- If n is even number, we can halve n, and compute $b^{\frac{n}{2}}$ first. Then calculate the square number of this result.
- Otherwise, n is odd. Since $n - 1$ is even, we can first recursively compute b^{n-1}, the multiply b one more time to this result.

Below equation formalizes this description.

$$pow(b, n) = \begin{cases} 1 & : & n = 0 \\ pow(b, \frac{n}{2})^2 & : & 2|n \\ b \times pow(b, n - 1) & : & otherwise \end{cases} \tag{A.22}$$

www.dbooks.org

However, it's hard to turn this algorithm to be tail-recursive mainly because of the 2nd clause. In fact, the 2nd clause can be alternatively realized by squaring the base number, and halve the exponent.

$$pow(b, n) = \begin{cases} 1 & : & n = 0 \\ pow(b^2, \frac{n}{2}) & : & 2|n \\ b \times pow(b, n-1) & : & otherwise \end{cases} \tag{A.23}$$

With this change, it's easy to get a tail-recursive algorithm as the following, so that $b^n = pow'(b, n, 1)$.

$$pow'(b, n, A) = \begin{cases} A & : & n = 0 \\ pow'(b^2, \frac{n}{2}, A) & : & 2|n \\ pow'(b, n-1, A \times b) & : & otherwise \end{cases} \tag{A.24}$$

Compare to the naive brute-force algorithm, we improved the performance to $O(\lg n)$. Actually, this algorithm can be improved even one more step.

Observe that if we represent n in binary format $n = (a_m a_{m-1} ... a_1 a_0)_2$, we clear know that the computation for b^{2^i} is necessary if $a_i = 1$. This is quite similar to the idea of Binomial heap (reader can refer to the chapter of binomial heap in this book). Thus we can calculate the final result by multiplying all of them for bits with value 1.

For instance, when we compute b^{11}, as $11 = (1011)_2 = 2^3 + 2 + 1$, thus $b^{11} = b^{2^3} \times b^2 \times b$. We can get the result by the following steps.

1. calculate b^1, which is b;
2. Get b^2 from previous result;
3. Get b^{2^2} from step 2;
4. Get b^{2^3} from step 3.

Finally, we multiply the results of step 1, 2, and 4 which yields b^{11}.

Summarize this idea, we can improve the algorithm as below.

$$pow'(b, n, A) = \begin{cases} A & : & n = 0 \\ pow'(b^2, \frac{n}{2}, A) & : & 2|n \\ pow'(b^2, \lfloor \frac{n}{2} \rfloor, A \times b) & : & otherwise \end{cases} \tag{A.25}$$

This algorithm essentially shift n to right for 1 bit each time (by dividing n by 2). If the LSB (Least Significant Bit, which is the lowest bit) is 0, it means n is even. It goes on computing the square of the base, without accumulating the final product (Just like the 3rd step in above example); If the LSB is 1, it means n is odd. It squares the base and accumulates it to the product A; The edge case is when n is zero, which means we exhaust all the bits in n, thus the final result is the accumulator A. At any time, the updated base number b', the shifted exponent number n', and the accumulator A satisfy the invariant that $b^n = b'^{n'} A$.

This algorithm can be implemented in Haskell like the following.

```
pow b n = pow' b n 1 where
  pow' b n acc | n == 0 = acc
               | even n = pow' (b*b) (n `div` 2) acc
               | otherwise = pow' (b*b) (n `div` 2) (acc*b)
```

Compare to previous algorithm, which minus n by one to change it to even when n is odd, this one halves n every time. It exactly runs m rounds, where m is the number of bits of n. However, the performance is still bound to $O(\lg n)$. How to implement this algorithm imperatively is left as exercise to the reader.

Imperative sum and product

The imperative sum and product are just applying plus and times while traversing the list.

```
1: function SUM(L)
2:     s ← 0
3:     while L ≠ φ do
4:         s ← s+ FIRST(L)
5:         L ← REST(L)
6:     return s

7: function PRODUCT(L)
8:     p ← 1
9:     while L ≠ φ do
10:        p ← p× FIRST(L)
11:        L ← REST(L)
12:    return p
```

The corresponding ISO C++ example programs are list as the following.

```
template<typename T>
T sum(List<T>* xs) {
  T s;
  for (s = 0; xs; xs = xs->next)
    s += xs->key;
  return s;
}

template<typename T>
T product(List<T>* xs) {
  T p;
  for (p = 1; xs; xs = xs->next)
    p *= xs->key;
  return p;
}
```

One interesting usage of product algorithm is that we can calculate factorial of n by calculating the product of $\{1, 2, ..., n\}$ that $n! = product([1..n])$.

A.3.8 maximum and minimum

Another very useful use case is to get the minimum or maximum element of a list. We'll see that their algorithm structures are quite similar again. We'll generalize this kind of feature and introduce about higher level abstraction in later section. For both maximum and minimum algorithms, we assume that the given list isn't empty.

In order to find the minimum element in a list.

www.dbooks.org

- If the list contains only one element, (a singleton list), the minimum element is this one;
- Otherwise, we can firstly find the minimum element of the rest list, then compare the first element with this intermediate result to determine the final minimum value.

This algorithm can be formalized by the following equation.

$$min(L) = \begin{cases} l_1 & : & L = \{l_1\} \\ l_1 & : & l_1 \leq min(L') \\ min(L') & : & otherwise \end{cases} \tag{A.26}$$

In order to get the maximum element instead of the minimum one, we can simply replace the $\leq$ comparison to $\geq$ in the above equation.

$$max(L) = \begin{cases} l_1 & : & L = \{l_1\} \\ l_1 & : & l_1 \geq max(L') \\ max(L') & : & otherwise \end{cases} \tag{A.27}$$

Note that both maximum and minimum actually process the list from right to left. It remind us about tail recursion. We can modify them so that the list is processed from left to right. What's more, the tail recursion version brings us 'on-line' algorithm, that at any time, we hold the minimum or maximum result of the list we examined so far.

$$min'(L, a) = \begin{cases} a & : & L = \phi \\ min(L', l_1) & : & l_1 < a \\ min(L', a) & : & otherwise \end{cases} \tag{A.28}$$

$$max'(L, a) = \begin{cases} a & : & L = \phi \\ max(L', l_1) & : & a < l_1 \\ max(L', a) & : & otherwise \end{cases} \tag{A.29}$$

Different from the tail recursion sum and product, we can't pass constant value to min', or max' in practice, this is because we have to pass infinity ($min(L, \infty)$) or negative infinity ($max(L, -\infty)$) in theory, but in a real machine neither of them can be represented since the length of word is limited.

Actually, there is workaround, we can instead pass the first element of the list, so that the algorithms become applicable.

$$\begin{aligned} min(L) &= min(L', l_1) \\ max(L) &= max(L', l_1) \end{aligned} \tag{A.30}$$

The corresponding real programs are given as the following. We skip the none tail recursion programs, as they are intuitive enough. Reader can take them as exercises for interesting.

```
min (x:xs) = min' xs x where
    min' [] a = a
    min' (x:xs) a = if x < a then min' xs x else min' xs a

max (x:xs) = max' xs x where
    max' [] a = a
    max' (x:xs) a = if a < x then max' xs x else max' xs a
```

The tail call version can be easily translated to imperative min/max algorithms.

```
1: function MIN(L)
2:     m ← FIRST(L)
3:     L ← REST(L)
4:     while L ≠ φ do
5:         if FIRST(L) < m then
6:             m ← FIRST(L)
7:         L ← REST(L)
8:     return m

9: function MAX(L)
10:    m ← FIRST(L)
11:    L ← REST(L)
12:    while L ≠ φ do
13:        if m < FIRST(L) then
14:            m ← FIRST(L)
15:        L ← REST(L)
16:    return m
```

The corresponding ISO C++ programs are given as below.

```
template<typename T>
T min(List<T>* xs) {
  T x;
  for (x = xs->key; xs; xs = xs->next)
    if (xs->key < x)
      x = xs->key;
  return x;
}

template<typename T>
T max(List<T>* xs) {
  T x;
  for (x = xs->key; xs; xs = xs->next)
    if (x < xs->key)
      x = xs->key;
  return x;
}
```

Another method to achieve tail-call maximum(and minimum) algorithm is by discarding the smaller element each time. The edge case is as same as before; for recursion case, since there are at least two elements in the list, we can take the first two for comparing, then drop one and go on process the rest. For a list with more than two elements, denote L'' as $rest(rest(L)) = \{l_3, l_4, ...\}$, we have the following equation.

$$max(L) = \begin{cases} l_1 & : & |L| = 1 \\ max(cons(l_1, L'')) & : & l_2 < l_1 \\ max(L') & : & otherwise \end{cases} \tag{A.31}$$

www.dbooks.org

$$min(L) = \begin{cases} l_1 & : & |L| = 1 \\ min(cons(l_1, L'')) & : & l_1 < l_2 \\ min(L') & : & otherwise \end{cases} \tag{A.32}$$

The relative example Haskell programs are given as below.

```
min [x] = x
min (x:y:xs) = if x < y then min (x:xs) else min (y:xs)

max [x] = x
max (x:y:xs) = if x < y then max (y:ys) else max (x:xs)
```

Exercise A.1

- Given two lists L_1 and L_2, design a algorithm $eq(L_1, L_2)$ to test if they are equal to each other. Here equality means the lengths are same, and at the same time, every elements in both lists are identical.
- Consider varies of options to handle the out-of-bound error case when randomly access the element in list. Realize them in both imperative and functional programming languages. Compare the solutions based on exception and error code.
- Augment the list with a 'tail' field, so that the appending algorithm can be realized in constant $O(1)$ time but not linear $O(n)$ time. Feel free to choose your favorite imperative programming language. Please don't refer to the example source code along with this book before you try it.
- With 'tail' field augmented to list, for which list operations this field must be updated? How it affects the performance?
- Handle the out-of-bound case in insertion algorithm by treating it as appending.
- Write the insertion sort algorithm by only using less than ($<$).
- Design and implement the algorithm that find all the occurrence of a given value and delete them from the list.
- Reimplenent the algorithm to calculate the length of a list in tail-call recursion manner.
- Implement the insertion sort in tail recursive manner.
- Implement the $O(\lg n)$ algorithm to calculate b^n in your favorite imperative programming language. Note that we only need accumulate the intermediate result when the bit is not zero.

A.4 Transformation

In previous section, we list some basic operations for linked-list. In this section, we focus on the transformation algorithms for list. Some of them are corner stones of abstraction for functional programming. We'll show how to use list transformation to solve some interesting problems.

A.4.1 mapping and for-each

It is every-day programming routine that, we need output something as readable string. If we have a list of numbers, and we want to print the list to console like '3 1 2 5 4'. One option is to convert the numbers to strings, so that we can feed them to the printing function. One such trivial conversion program may like this.

$$toStr(L) = \begin{cases} \phi & : & L = \phi \\ cons(str(l_1), toStr(L')) & : & otherwise \end{cases} \tag{A.33}$$

The other example is that we have a dictionary which is actually a list of words grouped in their initial letters, for example: [[a, an, another, ...], [bat, bath, bool, bus, ...], ..., [zero, zoo, ...]]. We want to know the frequency of them in English, so that we process some English text, for example, 'Hamlet' or the 'Bible' and augment each of the word with a number of occurrence in these texts. Now we have a list like this:

```
[[(a, 1041), (an, 432), (another, 802), ... ],
 [(bat, 5), (bath, 34), (bool, 11), (bus, 0), ...],
 ...,
 [(zero 12), (zoo, 0), ...]]
```

If we want to find which word in each initial is used most, how to write a program to work this problem out? The output is a list of words that every one has the most occurrence in the group, which is categorized by initial, something like '[a, but, can, ...]'. We actually need a program which can transfer a list of group of augmented words into a list of words.

Let's work it out step by step. First, we need define a function, which takes a list of word - number pairs, and find the word has the biggest number augmented. Sorting is overkill. What we need is just a special $max'()$ function, Note that the $max()$ function developed in previous section can't be used directly. Suppose for a pair of values $p = (a, b)$, function $fst(p) = a$, and $snd(p) = b$ are accessors to extract the values, $max'()$ can be defined as the following.

$$max'(L) = \begin{cases} l_1 & : & |L| = 1 \\ l_1 & : & snd(max'(L')) < snd(l_1) \\ max'(L') & : & otherwise \end{cases} \tag{A.34}$$

Alternatively, we can define a dedicated function to compare word-number of occurrence pair, and generalize the $max()$ function by passing a compare function.

$$less(p_1, p_2) = snd(p_1) < snd(p_2) \tag{A.35}$$

$$maxBy(cmp, L) = \begin{cases} l_1 & : & |L| = 1 \\ l_1 & : & cmp(l_1, maxBy(cmp, L')) \\ maxBy(cmp, L') & : & otherwise \end{cases} \tag{A.36}$$

www.dbooks.org

Then $max'()$ is just a special case of $maxBy()$ with the compare function comparing on the second value in a pair.

$$max'(L) = maxBy(\neg less, L) \tag{A.37}$$

Here we write all functions in purely recursive way, they can be modified in tail call manner. This is left as exercise to the reader.

With $max'()$ function defined, it's possible to complete the solution by processing the whole list.

$$solve(L) = \begin{cases} \phi & : \quad L = \phi \\ cons(fst(max'(l_1)), solve(L')) & : \quad otherwise \end{cases} \tag{A.38}$$

Map

Compare the $solve()$ function in (A.38) and $toStr()$ function in (A.33), it reveals very similar algorithm structure. although they targets on very different problems, and one is trivial while the other is a bit complex.

The structure of $toStr()$ applies the function $str()$ which can turn a number into string on every element in the list; while $solve()$ first applies $max'()$ function to every element (which is actually a list of pairs), then applies $fst()$ function, which essentially turns a list of pairs into a string. It is not hard to abstract such common structure like the following equation, which is called as *mapping*.

$$map(f, L) = \begin{cases} \phi & : \quad L = \phi \\ cons(f(l_1)), map(f, L')) & : \quad otherwise \end{cases} \tag{A.39}$$

Because map takes a 'converter' function f as argument, it's called a kind of high-order function. In functional programming environment such as Haskell, mapping can be implemented just like the above equation.

```
map :: (a→b)→[a]→[b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

The two concrete cases we discussed above can all be represented in high order mapping.

$$\begin{aligned} toStr &= map \quad str \\ solve &= map \quad (fst \cdot max') \end{aligned}$$

Where $f \cdot g$ means function composing, that we first apply g then apply f. For instance function $h(x) = f(g(x))$ can be represented as $h = f \cdot g$, reading like function h is composed by f and g. Note that we use Curried form to omit the argument L for brevity. Informally speaking, If we feed a function which needs 2 arguments, for instance $f(x, y) = z$ with only 1 argument, the result turns to be a function which need 1 argument. For instance, if we feed f with only argument x, it turns to be a new function take one argument y, defined as $g(y) = f(x, y)$, or $g = fx$. Note that x isn't a free variable any more, as it is bound to a value. Reader can refer to any book about functional programming for details about function composing and Currying.

Mapping can also be understood from the domain theory point of view. Consider function $y = f(x)$, it actually defines a mapping from domain of

variable x to the domain of value y. (x and y can have different types). If the domains can be represented as set X, and Y, we have the following relation.

$$Y = \{f(x)|x \in X\} \tag{A.40}$$

This type of set definition is called Zermelo Frankel set abstraction (as known as ZF expression) [7]. The different is that here the mapping is from a list to another list, so there can be duplicated elements. In languages support list comprehension, for example Haskell and Python etc (Note that the Python list is a built-in type, but not the linked-list we discussed in this appendix), mapping can be implemented as a special case of list comprehension.

```
map f xs = [ f x | x ← xs]
```

List comprehension is a powerful tool. Here is another example that realizes the permutation algorithm in list comprehension. Many textbooks introduce how to implement all-permutation for a list, such as [7], and [9]. It is possible to design a more general version $perm(L, r)$, that if the length of the list L is n, this algorithm permutes r elements from the total n elements. We know that there are $P_n^r = \frac{n!}{(n-r)!}$ solutions.

$$perm(L, r) = \begin{cases} \{\phi\} & : \quad r = 0 \vee |L| < r \\ \{\{l\} \cup P | l \in L, P \in perm(L - \{l\}, r - 1)\} & : \quad otherwise \end{cases} \tag{A.41}$$

In this equation, $\{l\} \cup P$ means $cons(l, P)$, and $L - \{l\}$ denotes $delete(L, l)$, which is defined in previous section. If we take zero element for permutation, or there are too few elements (less than r), the result is a list contains a empty list; Otherwise for non-trivial case, the algorithm picks one element l from the list, and recursively permutes the rest $n - 1$ elements by picking up $r - 1$ ones; then it puts all the possible l in front of all the possible $r - 1$ permutations. Here is the Haskell implementation of this algorithm.

```
perm _ 0 = [[]]
perm xs r | length xs < r = [[]]
          | otherwise = [ x:ys | x ←xs, ys ← perm (delete x xs) (r-1)]
```

We'll go back to the list comprehension later in section about filtering.

Mapping can also be realized imperatively. We can apply the function while traversing the list, and construct the new list from left to right. Since that the new element is appended to the result list, we can track the tail position to achieve constant time appending, so the mapping algorithms is linear in terms of the passed in function.

```
1: function MAP(f, L)
2:     L' ← φ
3:     p ← φ
4:     while L ≠ φ do
5:         if p = φ then
6:             p ← CONS(f( FIRST(L) ), φ)
7:             L' ← p
8:         else
9:             NEXT(p) ← CONS(f( FIRST(L) ), φ)
10:            p ← NEXT(p)
```

www.dbooks.org

```
11:         L ← NEXT(L)
12:     return L′
```

In some static typed programming luangaes without type inference feature, like C++[6], It is a bit complex to annotate the type of the passed-in function. See [11] for detail. In fact some C++ environment provides the very same mapping concept as in `std::transform`. However, it needs the reader to know some language specific features, which are out of the scope of this book.

For brevity purpose, we switch to Python programming language for example code. So that the type inference can be avoid in compile time. The definition of a simple singly linked-list in Python is give as the following.

```
class List:
    def __init__(self, x = None, xs = None):
        self.key = x
        self.next = xs

def cons(x, xs):
    return List(x, xs)
```

The mapping program, takes a function and a linked-list, and maps the functions to every element as described in above algorithm.

```
def mapL(f, xs):
    ys = prev = List()
    while xs is not None:
        prev.next = List(f(xs.key))
        prev = prev.next
        xs = xs.next
    return ys.next
```

Different from the pseudo code, this program uses a dummy node as the head of the resulting list. So it needn't test if the variable stores the last appending position is NIL. This small trick makes the program compact. We only need drop the dummy node before returning the result.

For each

For the trivial task such as printing a list of elements out, it's quite OK to just print each element without converting the whole list to a list of strings. We can actually simplify the program.

```
1: function PRINT(L)
2:     while L ≠ φ do
3:         print FIRST(L)
4:         L ← REST(L)
```

More generally, we can pass a procedure such as printing, to this list traverse, so the procedure is performed *for each* element.

```
1: function FOR-EACH(L, P)
2:     while L ≠ φ do
3:         P(FIRST(L))
4:         L ← REST(L)
```

[6] At least in ISO C++ 1998 standard.

For-each algorithm can be formalized in recursive approach as well.

$$foreach(L, p) = \begin{cases} u & : & L = \phi \\ do(p(l_1), foreach(L', p)) & : & otherwise \end{cases} \tag{A.42}$$

Here u means unit, it's can be understood as doing nothing. The type of unit is similar to the 'void' concept in C or java like programming languages. The $do()$ function evaluates all its arguments, discards all the results except for the last one, and returns the last result as the final value of $do()$. It is equivalent to `(begin ...)` in Lisp families, and `do` block in Haskell in some sense. For the details about unit type, please refer to [4].

Note that the for-each algorithm is just a simplified mapping, there are only two minor difference points:

- It needn't form a result list, we care the 'side effect' rather than the returned value;
- For each focus more on traversing, while mapping focus more on applying function, thus the order of arguments are typically arranged as $map(f, L)$ and $foreach(L, p)$.

Some Functional programming facilities provide options for both returning the result list or discarding it. For example Haskell Monad library provides both `mapM`, `mapM_` and `forM`, `forM_`. Readers can refer to language specific materials for detail.

Examples for mapping

We'll show how to use mapping by an example, which is a problem of ACM/ICPC[12]. For sake of brevity, we modified the problem description a bit. Suppose there are n lights in a room, all of them are off. We execute the following process n times:

1. We switch all the lights in the room, so that they are all on;
2. We switch the 2, 4, 6, ... lights, that every other light is switched, if the light is on, it will be off, and it will be on if the previous state is off;
3. We switch every third lights, that the 3, 6, 9, ... are switched;
4. ...

And at the last round, only the last light (the n-th light) is switched.

The question is how many lights are on finally?

Before we show the best answer to this puzzle, let's first work out a naive brute-force solution. Suppose there are n lights, which can be represented as a list of 0, 1 numbers, where 0 means the light is off, and 1 means on. The initial state is a list of n zeros: $\{0, 0, ..., 0\}$.

We can label the light from 1 to n. A mapping can help us to turn the above list into a labeled list[7].

[7]Readers who are familiar with functional programming, may use zipping to achieve this. We'll explain zipping in later section.

www.dbooks.org

$$map(\lambda_i \cdot (i, 0), \{1, 2, 3, ...n\})$$

This mapping augments each natural number with zero, the result is a list of pairs: $L = \{(1, 0), (2, 0), ..., (n, 0)\}$.

Next we operate this list of pairs n times from 1 to n. For every time i, we switch the second value in this pair if the first label can be divided by i. Consider the fact that $1 - 0 = 1$, and $1 - 1 = 0$, we can realize switching of 0, 1 value x by $1 - x$. At the i-th operation, for light (j, x), if $i|j$, (or $j \mod i = 0$), we then perform switching, otherwise, we leave the light untouched.

$$switch(i, (j, x)) = \begin{cases} (j, 1 - x) & : \quad j \mod i = 0 \\ (j, x) & : \quad otherwise \end{cases} \tag{A.43}$$

The i-th operation on all lights can be realized as mapping again:

$$map(switch(i), L) \tag{A.44}$$

Note that, here we use Curried form of $switch()$ function, which is equivalent to

$$map(\lambda_{(j,x)} \cdot switch(i, (j, x)), L)$$

Here we need define a function $proc()$, which can perform the above mapping on L over and over by n times. One option is to realize it in purely recursive way as the following, so that we can call it like $proc(\{1, 2, ..., n\}, L)$[8].

$$proc(I, L) = \begin{cases} L & : \quad I = \phi \\ operate(I', map(switch(i_1), L)) & : \quad otherwise \end{cases} \tag{A.45}$$

Where $I = cons(i_1, I')$ if I isn't empty.

At this stage, we can sum the second value of each pair in list L to get the answer. The sum function has been defined in previous section, so the only thing left is mapping.

$$solve(n) = sum(map(snd, proc(\{1, 2, ..., n\}, L))) \tag{A.46}$$

Translating this naive brute-force solution to Haskell yields below program.

```
solve' = sum ∘ (map snd) ∘ proc  where
    proc n = operate [1..n] $ map (λi → (i, 0)) [1..n]
    operate [] xs = xs
    operate (i:is) xs = operate is (map (switch i) xs)

switch i (j, x) = if j `mod` i == 0 then (j, 1 - x) else (j, x)
```

Let's see what's the answer for there are 1, 2, ..., 100 lights.

```
[1,1,1,2,2,2,2,2,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,5,5,5,
6,6,6,6,6,6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8,
8,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,10]
```

This result is interesting:

[8] This can also be realized by folding, which will be explained in later section.

- the first 3 answers are 1;
- the 4-th to the 8-th answers are 2;
- the 9-th to the 15-th answers are 3;
- ...

It seems that the i^2-th to the $((i+1)^2-1)$-th answers are i. Actually, we can prove this fact as the following.

Proof. Given n lights, labeled from 1 to n, consider which lights are on finally. Since the initial states for all lights are off, we can say that, the lights which are manipulated odd times are on. For every light i, it will be switched at the j round if i can be divided by j (denote as $j|i$). So only the lights which have odd number of factors are on at the end.

So the key point to solve this puzzle, is to find all numbers which have odd number of factors. For any positive integer n, denote S the set of all factors of n. S is initialized to ϕ. if p is a factor of n, there must exist a positive integer q that $n = pq$, which means q is also a factor of n. So we add 2 different factors to the set S if and only if $p \neq q$, which keeps $|S|$ even all the time unless $p = q$. In such case, n is a perfect square number, and we can only add 1 factor to the set S, which leads to an odd number of factors. □

At this stage, we can design a fast solution by finding the number of perfect square numbers under n.

$$solve(n) = \lfloor \sqrt{n} \rfloor \tag{A.47}$$

The next Haskell command verifies that the answer for 1, 2, ..., 100 lights are as same as above.

```
map (floor.sqrt) [1..100]
[1,1,1,2,2,2,2,2,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,5,5,
6,6,6,6,6,6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8,
8,8,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,10]
```

Mapping is generic concept that it doesn't only limit in linked-list, but also can be applied to many complex data structures. The chapter about binary search tree in this book explains how to map on trees. As long as we can traverse a data structure in some order, and the empty data structure can be identified, we can use the same mapping idea. We'll return to this kind of high-order concept in the section of folding later.

A.4.2 reverse

How to reverse a singly linked-list with minimum space is a popular technical interview problem in some companies. The pointer manipulation must be arranged carefully in imperative programming languages such as ANSI C. However, we'll show that, there exists an easy way to write this program:

1. Firstly, write a pure recursive straightforward solution;
2. Then, transform the pure recursive solution to tail-call manner;

www.dbooks.org

3. Finally, translate the tail-call solution to pure imperative pointer operations.

The pure recursive solution is simple enough that we can write it out immediately. In order to *reverse a list* L.

- If L is empty, the reversed result is empty. This is the trivial edge case;
- Otherwise, we can first reverse the rest of the sub-list, then append the first element to the end.

This idea can be formalized to the below equation.

$$reverse(L) = \begin{cases} \phi & : & L = \phi \\ append(reverse(L'), l_1) & : & otherwise \end{cases} \tag{A.48}$$

Translating it to Haskell yields below program.

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

However, this solution doesn't perform well, as appending has to traverse to the end of list, which leads to a quadratic time algorithm. It is not hard to improve this program by changing it to tail-call manner. That we can use a accumulator to store the intermediate reversed result, and initialize the accumulated result as empty. So the algorithm is formalized as $reverse(L) = reverse'(L, \phi)$.

$$reverse'(L, A) = \begin{cases} A & : & L = \phi \\ reverse'(L', \{l_1\} \cup A) & : & otherwise \end{cases} \tag{A.49}$$

Where $\{l_1\} \cup A$ means $cons(l_1, A)$. Different from appending, it's a constant $O(1)$ time operation. The core idea is that we repeatedly take the element one by one from the head of the original list, and put them in front the accumulated result. This is just like we store all the elements in a stack, them pop them out. This is a linear time algorithm.

Below Haskell program implements this tail-call version.

```
reverse' [] acc = acc
reverse' (x:xs) acc = reverse' xs (x:acc)
```

Since the nature of tail-recursion call needn't book-keep any context (typically by stack), most modern compilers are able to optimize it to a pure imperative loop, and reuse the current context and stack etc. Let's manually do this optimization so that we can get a imperative algorithm.

```
1: function REVERSE(L)
2:     A ← φ
3:     while L ≠ φ do
4:         A ← CONS(FIRST(L), A)
5:         L ← REST(L)
```

However, because we translate it directly from a functional solution, this algorithm actually produces a new reversed list, but does not mutate the original one. It is not hard to change it to an in-place solution by reusing L. For example, the following ISO C++ program implements the in-place algorithm. It takes $O(1)$ memory space, and reverses the list in $O(n)$ time.

```
template<typename T>
List<T>* reverse(List<T>* xs) {
  List<T> *p, *ys = NULL;
  while (xs) {
    p = xs;
    xs = xs→next;
    p→next = ys;
    ys = p;
  }
  return ys;
}
```

Exercise A.2

- Implement the algorithm to find the maximum element in a list of pair in tail call approach in your favorite programming language.

A.5 Extract sub-lists

Different from arrays which are capable to slice a continuous segment fast and easily, It needs more work to extract sub lists from singly linked list. Such operations are typically linear algorithms.

A.5.1 take, drop, and split-at

Taking first n elements from a list is semantically similar to extract sub list from the very left like $sublist(L, 1, n)$, where the second and the third arguments to $sublist$ are the positions the sub-list starts and ends. For the trivial edge case, that either n is zero or the list is empty, the sub list is empty; Otherwise, we can recursively take the first $n-1$ elements from the rest of the list, and put the first element in front of it.

$$take(n, L) = \begin{cases} \phi & : \quad L = \phi \vee n = 0 \\ cons(l_1, take(n-1, L')) & : \quad otherwise \end{cases} \tag{A.50}$$

Note that the edge cases actually handle the out-of-bound error. The following Haskell program implements this algorithm.

```
take _ [] = []
take 0 _ = []
take n (x:xs) = x : take (n-1) xs
```

Dropping on the other hand, drops the first n elements and returns the left as result. It is equivalent to get the sub list from right like $sublist(L, n+1, |L|)$, where $|L|$ is the length of the list. Dropping can be designed quite similar to taking by discarding the first element in the recursive case.

$$drop(n, L) = \begin{cases} \phi & : \quad L = \phi \\ L & : \quad n = 0 \\ drop(n-1, L')) & : \quad otherwise \end{cases} \tag{A.51}$$

Translating the algorithm to Haskell gives the below example program.

www.dbooks.org

```
drop _ [] = []
drop 0 L = L
drop n (x:xs) = drop (n-1) xs
```

The imperative taking and dropping are quite straight-forward, that they are left as exercises to the reader.

With taking and dropping defined, extracting sub list at arbitrary position for arbitrary length can be realized by calling them.

$$sublist(L, from, count) = take(count, drop(from - 1, L)) \tag{A.52}$$

or in another semantics by providing left and right boundaries:

$$sublist(L, from, to) = drop(from - 1, take(to, L)) \tag{A.53}$$

Note that the elements in range $[from, to]$ is returned by this function, with both ends included. All the above algorithms perform in linear time.

take-while and drop-while

Compare to taking and dropping, there is another type of operation, that we either keep taking or dropping elements as far as a certain condition is met. The taking and dropping algorithms can be viewed as special cases for take-while and drop-while.

Take-while examines elements one by one as far as the condition is satisfied, and ignore all the rest of elements even some of them satisfy the condition. This is the different point from filtering which we'll explained in later section. Take-while stops once the condition tests fail; while filtering traverses the whole list.

$$takeWhile(p, L) = \begin{cases} \phi & : & L = \phi \\ \phi & : & \neg p(l_1) \\ cons(l_1, takeWhile(p, L')) & : & otherwise \end{cases} \tag{A.54}$$

Take-while accepts two arguments, one is the predicate function p, which can be applied to element in the list and returns Boolean value as result; the other argument is the list to be processed.

It is easy to define the drop-while symmetrically.

$$dropWhile(p, L) = \begin{cases} \phi & : & L = \phi \\ L & : & \neg p(l_1) \\ dropWhile(p, L') & : & otherwise \end{cases} \tag{A.55}$$

The corresponding Haskell example programs are given as below.

```
takeWhile _ [] = []
takeWhile p (x:xs) = if p x then x : takeWhile p xs else []

dropWhile _ [] = []
dropWhile p xs@(x:xs') = if p x then dropWhile p xs' else xs
```

split-at

With taking and dropping defined, splitting-at can be realized trivially by calling them.

$$splitAt(i, L) = (take(i, L), drop(i, L)) \tag{A.56}$$

A.5.2 breaking and grouping

breaking

Breaking can be considered as a general form of splitting, instead of splitting at a given position, breaking examines every element for a certain predicate, and finds the longest prefix of the list for that condition. The result is a pair of sub-lists, one is that longest prefix, the other is the rest.

There are two different breaking semantics, one is to pick elements satisfying the predicate as long as possible; the other is to pick those don't satisfy. The former is typically defined as *span*, while the later as *break*.

Span can be described, for example, in such recursive manner: In order to span a list L for predicate p:

- If the list is empty, the result for this edge trivial case is a pair of empty lists (ϕ, ϕ);
- Otherwise, we test the predicate against the first element l_1, if l_1 satisfies the predicate, we denote the intermediate result for spanning the rest of list as $(A, B) = span(p, L')$, then we put l_1 in front of A to get pair $(\{l_1\} \cup A, B)$, otherwise, we just return (ϕ, L) as the result.

For breaking, we just test the negate of predicate and all the others are as same as spanning. Alternatively, one can define breaking by using span as in the later example program.

$$span(p, L) = \begin{cases} (\phi, \phi) & : \quad L = \phi \\ (\{l_1\} \cup A, B) & : \quad p(l_1) = True, (A, B) = span(p, L') \\ (\phi, L) & : \quad otherwise \end{cases} \tag{A.57}$$

$$break(p, L) = \begin{cases} (\phi, \phi) & : \quad L = \phi \\ (\{l_1\} \cup A, B) & : \quad \neg p(l_1), (A, B) = break(p, L') \\ (\phi, L) & : \quad otherwise \end{cases} \tag{A.58}$$

Note that both functions only find the longest *prefix*, they stop immediately when the condition is fail even if there are elements in the rest of the list meet the predicate (or not). Translating them to Haskell gives the following example program.

```
span _ [] = ([], [])
span p xs@(x:xs') = if p x then let (as, bs) = span p xs' in (x:as, bs) else ([], xs)

break p = span (not ∘ p)
```

www.dbooks.org

Span and break can also be realized imperatively as the following.

```
1: function SPAN(p, L)
2:     A ← φ
3:     while L ≠ φ ∧ p(l_1) do
4:         A ← CONS(l_1, A)
5:         L ← REST(L)
6:     return (A, L)

7: function BREAK(p, L)
8:     return SPAN(¬p, L)
```

This algorithm creates a new list to hold the longest prefix, another option is to turn it into in-place algorithm to reuse the spaces as in the following Python example.

```
def span(p, xs):
    ys = xs
    last = None
    while xs is not None and p(xs.key):
        last = xs
        xs = xs.next
    if last is None:
        return (None, xs)
    last.next = None
    return (ys, xs)
```

Note that both span and break need traverse the list to test the predicate, thus they are linear algorithms bound to $O(n)$.

grouping

Grouping is a commonly used operation to solve the problems that we need divide the list into some small groups. For example, Suppose we want to group the string 'Mississippi', which is actual a list of char { 'M', 's', 's', 'i', 's', 's', 'i', 'p', 'p', 'i'}. into several small lists in sequence, that each one contains consecutive identical characters. The grouping operation is expected to be:

```
group(`Mississippi') = { `M'', `i', `ss', `i', `ss', `i', `pp', `i'}
```

Another example, is that we have a list of numbers:

$$L = \{15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2\}$$

We want to divide it into several small lists, that each sub-list is ordered descending. The grouping operation is expected to be :

$$group(L) = \{\{15, 9, 0\}, \{12, 11, 7\}, \{10, 5\}, \{6\}, \{13, 1\}, \{4\}, \{8, 3\}, \{14, 2\}\}$$

Both cases play very important role in real algorithms. The string grouping is used in creating Trie/Patricia data structure, which is a powerful tool in string searching area; The ordered sub-list grouping can be used in nature merge sort. There are dedicated chapters in this book explain the detail of these algorithms.

It is obvious that we need abstract the grouping condition so that we know where to break the original list into small ones. This predicate can be passed to the algorithm as an argument like $group(p, L)$, where predicate p accepts two consecutive elements and test if the condition matches.

The first idea to solve the grouping problem is traversing – takes two elements at each time, if the predicate test succeeds, put both elements into a small group; otherwise, only put the first one into the group, and use the second one to initialize another new group. Denote the first two elements (if there are) are l_1, l_2, and the sub-list without the first element as L'. The result is a list of list $G = \{g_1, g_2, ...\}$, denoted as $G = group(p, L)$.

$$group(p, L) = \begin{cases} \{\phi\} & : & L = \phi \\ \{\{l_1\}\} & : & |L| = 1 \\ \{\{l_1\} \cup g_1', g_2', ...\} & : & p(l_1, l_2), G' = group(p, L') = \{g_1', g_2', ...\} \\ \{\{l_1\}, g_1', g_2', ...\} & : & otherwise \end{cases} \tag{A.59}$$

Note that $\{l_1\} \cup g_1'$ actually means $cons(l_1, g_1')$, which performs in constant time. This is a linear algorithm performs proportion to the length of the list, it traverses the list in one pass which is bound to $O(n)$. Translating this program to Haskell gives the below example code.

```
group _ [] = [[]]
group _ [x] = [[x]]
group p (x:xs@(x':_)) | p x x' = (x:ys):yss
                      | otherwise = [x]:r
  where
    r@(ys:yss) = group p xs
```

It is possible to implement this algorithm in imperative approach, that we initialize the result groups as $\{l_1\}$ if L isn't empty, then we traverse the list from the second one, and append to the last group if the two consecutive elements satisfy the predicate; otherwise we start a new group.

```
1:  function GROUP(p, L)
2:      if L = φ then
3:          return {φ}
4:      x ← FIRST(L)
5:      L ← REST(L)
6:      g ← {x}
7:      G ← {g}
8:      while L ≠ φ do
9:          y ← FIRST(L)
10:         if p(x, y) then
11:             g ← APPEND(g, y)
12:         else
13:             g ← {y}
14:             G ← APPEND(G, g)
15:         x ← y
16:         L ← NEXT(L)
17:     return G
```

However, different from the recursive algorithm, this program performs in

www.dbooks.org

quadratic time if the appending function isn't optimized by storing the tail position. The corresponding Python program is given as below.

```
def group(p, xs):
    if xs is None:
        return List(None)
    (x, xs) = (xs.key, xs.next)
    g = List(x)
    G = List(g)
    while xs is not None:
        y = xs.key
        if p(x, y):
            g = append(g, y)
        else:
            g = List(y)
            G = append(G, g)
        x = y
        xs = xs.next
    return G
```

With the grouping function defined, the two example cases mentioned at the beginning of this section can be realized by passing different predictions.

$$group(=, \{m, i, s, s, i, s, s, i, p, p, i\}) = \{\{M\}, \{i\}, \{ss\}, \{i\}, \{ss\}, \{i\}, \{pp\}, \{i\}\}$$

$$\begin{aligned} &group(\geq, \{15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2\}) \\ &= \{\{15, 9, 0\}, \{12, 11, 7\}, \{10, 5\}, \{6\}, \{13, 1\}, \{4\}, \{8, 3\}, \{14, 2\}\} \end{aligned}$$

Another solution is to use the *span* function we have defined to realize grouping. We pass a predicate to span, which will break the list into two parts: The first part is the longest sub-list satisfying the condition. We can repeatedly apply the span with the same predication to the second part, till it becomes empty.

However, the predicate function we passed to span is *an unary function*, that it takes an element as argument, and test if it satisfies the condition. While in grouping algorithm, the predicate function is *a binary function*. It takes two adjacent elements for testing. The solution is that, we can use currying and pass the first element to the binary predicate, and use it to test the rest of elements.

$$group(p, L) = \begin{cases} \{\phi\} & : & L = \phi \\ \{\{l_1\} \cup A\} \cup group(p, B) & : & otherwise \end{cases} \tag{A.60}$$

Where $(A, B) = span(\lambda_x \cdot p(l_1, x), L')$ is the result of spanning on the rest sub-list of L.

Although this new defined grouping function can generate correct result for the first case as in the following Haskell code snippet.

```
groupBy (==) "Mississippi"
["m","i","ss","i","ss","i","pp","i"]
```

However, it seems that this algorithm can't group the list of numbers into ordered sub lists.

```
groupBy (≥) [15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2]
[[15,9,0,12,11,7,10,5,6,13,1,4,8,3,14,2]]
```

The reason is because that, as the first element 15 is used as the left parameter to $\geq$ operator for span, while 15 is the maximum value in this list, so the span function ends with putting all elements to A, and B is left empty. This might seem a defect, but it is actually the correct behavior if the semantic is to group equal elements together.

Strictly speaking, the equality predicate must satisfy three properties: reflexive, transitive, and symmetric. They are specified as the following.

- Reflexive. $x = x$, which says that any element is equal to itself;
- Transitive. $x = y, y = z \Rightarrow x = z$, which says that if two elements are equal, and one of them is equal to another, then all the tree are equal;
- Symmetric. $x = y \Leftrightarrow y = x$, which says that the order of comparing two equal elements doesn't affect the result.

When we group character list "Mississippi", the equal (=) operator is used, which obviously conforms these three properties. So that it generates correct grouping result. However, when passing ($\geq$) as equality predicate, to group a list of numbers, it violets both reflexive and symmetric properties, that is reason why we get wrong grouping result.

This fact means that the second algorithm we designed by using span, limits the semantic to strictly equality, while the first one does not. It just tests the condition for every two adjacent elements, which is much weaker than equality.

Exercise A.3

1. Implement the in-place imperative taking and dropping algorithms in your favorite programming language, note that the out of bound cases should be handled. Please try both languages with and without GC (Garbage Collection) support.

2. Implement take-while and drop-while in your favorite imperative programming language. Please try both dynamic type language and static type language (with and without type inference). How to specify the type of predicate function as generic as possible in static type system?

3. Consider the following definition of span.

$$span(p, L) = \begin{cases} (\phi, \phi) & : & L = \phi \\ (\{l_1\} \cup A, B) & : & p(l_1) = True, (A, B) = span(p, L') \\ (A, \{l_1\} \cup B) & : & otherwise \end{cases}$$

   What's the difference between this algorithm and the the one we've shown in this section?

4. Implement the grouping algorithm by using span in imperative way in your favorite programming language.

www.dbooks.org

A.6 Folding

We are ready to introduce one of the most critical concept in high order programming, folding. It is so powerful tool that almost all the algorithms so far in this appendix can be realized by folding. Folding is sometimes be named as reducing (the abstracted concept is identical to the buzz term 'map-reduce' in cloud computing in some sense). For example, both STL and Python provide reduce function which realizes partial form of folding.

A.6.1 folding from right

Remind the sum and product definition in previous section, they are quite similar actually.

$$sum(L) = \begin{cases} 0 & : \quad L = \phi \\ l_1 + sum(L') & : \quad otherwise \end{cases}$$

$$product(L) = \begin{cases} 1 & : \quad L = \phi \\ l_1 \times product(L') & : \quad otherwise \end{cases}$$

It is obvious that they have same structure. What's more, if we list the insertion sort definition, we can find that it also shares this structure.

$$sort(L) = \begin{cases} \phi & : \quad L = \phi \\ insert(l_1, sort(L')) & : \quad otherwise \end{cases}$$

This hint us that we can abstract this essential common structure, so that we needn't repeat it again and again. Observing *sum*, *product*, and *sort*, there are two different points which we can parameterize.

- The result of the trivial edge case varies. It is zero for sum, 1 for product, and empty list for sorting.
- The function applied to the first element and the intermediate result varies. It is plus for sum, multiply for product, and ordered-insertion for sorting.

If we parameterize the result of trivial edge case as initial value z (stands for abstract zero concept), the function applied in recursive case as f (which takes two parameters, one is the first element in the list, the other is the recursive result for the rest of the list), this common structure can be defined as something like the following.

$$proc(f, z, L) = \begin{cases} z & : \quad L = \phi \\ f(l_1, proc(f, z, L')) & : \quad otherwise \end{cases}$$

That's it, and we should name this common structure a better name instead of the meaningless 'proc'. Let's see the characteristic of this common structure. For list $L = \{x_1, x_2, ..., x_n\}$, we can expand the computation like the following.

$$\begin{aligned} proc(f, z, L) &= f(x_1, proc(f, z, L') \\ &= f(x_1, f(x_2, proc(f, z, L'')) \\ &... \\ &= f(x_1, f(x_2, f(..., f(x_n, f(f, z, \phi))...) \\ &= f(x_1, f(x_2, f(..., f(x_n, z))...) \end{aligned}$$

Since f takes two parameters, it's a binary function, thus we can write it in infix form. The infix form is defined as below.

$$x \oplus_f y = f(x, y) \tag{A.61}$$

The above expanded result is equivalent to the following by using infix notion.

$$proc(f, z, L) = x_1 \oplus_f (x_2 \oplus_f (...(x_n \oplus_f z))...)$$

Note that the parentheses are necessary, because the computation starts from the right-most $(x_n \oplus_f z)$, and repeatedly fold to left towards x_1. This is quite similar to folding a Chinese hand-fan as illustrated in the following photos. A Chinese hand-fan is made of bamboo and paper. Multiple bamboo frames are stuck together with an axis at one end. The arc shape paper is fully expanded by these frames as shown in Figure A.3 (a); The fan can be closed by folding the paper. Figure A.3 (b) shows that some part of the fan is folded from right. After these folding finished, the fan results a stick, as shown in Figure A.3 (c).



(a) A folding fan fully opened.



(b) The fan is partly folded on right.



(c) The fan is fully folded, closed to a stick.

Figure A.3: Folding a Chinese hand-fan

We can considered that each bamboo frame along with the paper on it as an element, so these frames forms a list. A unit process to close the fan is to rotate

www.dbooks.org

a frame for a certain angle, so that it lays on top of the collapsed part. When we start closing the fan, the initial collapsed result is the first bamboo frame. The close process is folding from one end, and repeatedly apply the unit close steps, till all the frames is rotated, and the folding result is a stick closed form.

Actually, the sum and product algorithms exactly do the same thing as closing the fan.

$$\begin{aligned} sum(\{1,2,3,4,5\}) &= 1+(2+(3+(4+5))) \\ &= 1+(2+(3+9)) \\ &= 1+(2+12) \\ &= 1+14 \\ &= 15 \end{aligned}$$

$$\begin{aligned} product(\{1,2,3,4,5\}) &= 1\times(2\times(3\times(4\times 5))) \\ &= 1\times(2\times(3\times 20)) \\ &= 1\times(2\times 60) \\ &= 1\times 120 \\ &= 120 \end{aligned}$$

In functional programming, we name this process *folding*, and particularly, since we execute from the most inner structure, which starts from the right-most one. This type of folding is named *folding right*.

$$foldr(f,z,L) = \begin{cases} z & : \quad L=\phi \\ f(l_1, foldr(f,z,L')) & : \quad otherwise \end{cases} \tag{A.62}$$

Let's see how to use fold-right to realize sum and product.

$$\begin{aligned} \sum_{i=1}^{N} x_i &= x_1+(x_2+(x_3+...+(x_{n-1}+x_n))...) \\ &= foldr(+,0,\{x_1,x_2,...,x_n\}) \end{aligned} \tag{A.63}$$

$$\begin{aligned} \prod_{i=1}^{N} x_i &= x_1\times(x_2\times(x_3\times...+(x_{n-1}\times x_n))...) \\ &= foldr(\times,1,\{x_1,x_2,...,x_n\}) \end{aligned} \tag{A.64}$$

The insertion-sort algorithm can also be defined by using folding right.

$$sort(L) = foldr(insert,\phi,L) \tag{A.65}$$

A.6.2 folding from left

As mentioned in section of 'tail recursive' call, both pure recursive sum and product compute from right to left and they must book keep all the intermediate results and contexts. As we abstract fold-right from the very same structure, folding from right does the book keeping as well. This will be expensive if the list is very long.

Since we can change the realization of sum and product to tail-recursive call manner, it quite possible that we can provide another folding algorithm, which processes the list from left to right in normal order, and enable the tail-call optimization by reusing the same context.

Instead of induction from sum, product and insertion sort, we can directly change the folding right to tail call. Observe that the initial value z, actually represents the intermediate result. We can use it as the accumulator.

$$foldl(f, z, L) = \begin{cases} z & : \quad L = \phi \\ foldl(f, f(z, l_1), L') & : \quad otherwise \end{cases} \tag{A.66}$$

Every time when the list isn't empty, we take the first element, apply function f on the accumulator z and it to get a new accumulator $z' = f(z, l_1)$. After that we can repeatedly folding with the very same function f, the updated accumulator z', and list L'.

Let's verify that this tail-call algorithm actually folding from left.

$$\begin{aligned} \sum_{i=1}^{5} i &= foldl(+, 0, \{1, 2, 3, 4, 5\}) \\ &= foldl(+, 0 + 1, \{2, 3, 4, 5\}) \\ &= foldl(+, (0 + 1) + 2, \{3, 4, 5\} \\ &= foldl(+, ((0 + 1) + 2) + 3, \{4, 5\}) \\ &= foldl(+, (((0 + 1) + 2) + 3) + 4, \{5\}) \\ &= foldl(+, ((((0 + 1) + 2 + 3) + 4 + 5, \phi) \\ &= 0 + 1 + 2 + 3 + 4 + 5 \end{aligned}$$

Note that, we actually delayed the evaluation of $f(z, l_1)$ in every step. (This is the exact behavior in system support lazy-evaluation, for instance, Haskell. However, in strict system such as standard ML, it's not the case.) Actually, they will be evaluated in sequence of $\{1, 3, 6, 10, 15\}$ in each call.

Generally, folding-left can be expanded in form of

$$foldl(f, z, L) = f(f(...(f(f(z, l_1), l_2), ..., l_n) \tag{A.67}$$

Or in infix manner as

$$foldl(f, z, L) = ((...(z \oplus_f l_1) \oplus_f l_2) \oplus_f ...) \oplus l_n \tag{A.68}$$

With folding from left defined, sum, product, and insertion-sort can be transparently implemented by calling $foldl$ as $sum(L) = foldl(+, 0, L)$, $product(L) = foldl(+, 1, L)$, and $sort(L) = foldl(insert, \phi, L)$. Compare with the folding-right version, they are almost same at first glares, however, the internal implementation differs.

Imperative folding and generic folding concept

The tail-call nature of folding-left algorithm is quite friendly for imperative settings, that even the compiler isn't equipped with tail-call recursive optimization, we can anyway implement the folding in while-loop manually.

```
1: function FOLD(f, z, L)
2:     while L ≠ φ do
3:         z ← f(z, FIRST(L) )
4:         L ← REST(L)
5:     return z
```

Translating this algorithm to Python yields the following example program.

www.dbooks.org

```
def fold(f, z, xs):
    for x in xs:
        z = f(z, x)
    return z
```

Actually, Python provides built-in function 'reduce' which does the very same thing. (in ISO C++, this is provided as reduce algorithm in STL.) Almost no imperative environment provides folding-right function because it will cause stack overflow problem if the list is too long. However, there still exist cases that the folding from right semantics is necessary. For example, one defines a container, which only provides insertion function to the head of the container, but there is no any appending method, so that we want such a $fromList$ tool.

$$fromList(L) = foldr(insertHead, empty, L)$$

Calling $fromList$ with the insertion function as well as an empty initialized container, can turn a list into the special container. Actually the singly linked-list is such a container, which performs well on insertion to the head, but poor to linear time if appending on the tail. Folding from right is quite nature when duplicate a linked-list while keeps the elements ordering. While folding from left will generate a reversed list.

In such cases, there exists an alternative way to implement imperative folding right by first reverse the list, and then folding the reversed one from left.

1: **function** FOLD-RIGHT(f, z, L)
2: **return** FOLD(f, z, REVERSE(L))

Note that, here we must use the tail-call version of reversing, or the stack overflow issue still exists.

One may think that folding-left should be chosen in most cases over folding-right because it's friendly for tail-recursion call optimization, suitable for both functional and imperative settings, and it's an online algorithm. However, folding-right plays a critical role when the input list is infinity and the binary function f is lazy. For example, below Haskell program wraps every element in an infinity list to a singleton, and returns the first 10 result.

```
take 10 $ foldr (λx xs →[x]:xs) [] [1..]
[[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]
```

This can't be achieved by using folding left because the outer most evaluation can't be finished until all the list being processed. The details is specific to lazy evaluation feature, which is out of the scope of this book. Readers can refer to [13] for details.

Although the main topic of this appendix is about singly linked-list related algorithms, the folding concept itself is generic which doesn't only limit to list, but also can be applied to other data structures.

We can fold a tree, a queue, or even more complicated data structures as long as we have the following:

- The empty data structure can be identified for trivial edge case; (e.g. empty tree)
- We can traverse the data structure (e.g. traverse the tree in pre-order).

Some languages provide this high-level concept support, for example, Haskell achieve this via *monoid*, readers can refer to [8] for detail.

There are many chapters in this book use the widen concept of folding.

A.6.3 folding in practice

We have seen that *sum*, *product*, and insertion sort all can be realized in folding. The brute-force solution for the puzzle shown in mapping section can also be designed by mixed use of mapping and folding.

Remind that we create a list of pairs, each pair contains the number of the light, and the on-off state. After that we process from 1 to n, switch the light if the number can be divided. The whole process can be viewed as folding.

$$fold(step, \{(1,0),(2,0),...,(n,0)\}, \{1,2,...,n\})$$

The initial value is the very first state, that all the lights are off. The list to be folding is the operations from 1 to n. Function *step* takes two arguments, one is the light states pair list, the other is the operation time i. It then maps on all lights and performs switching. We can then substitute the *step* with mapping.

$$fold(\lambda_{L,i} \cdot map(switch(i), L), \{(1,0),(2,0),...,(n,0)\}, \{1,2,...,n\})$$

We'll simplify the λ notation, and directly write $map(switch(i), L)$ for brevity purpose. The result of this folding is the final states pairs, we need take the second one of the pair for each element via mapping, then calculate the summation.

$$sum(map(snd, fold(map(switch(i), L), \{(1,0),(2,0),...,(n,0)\}, \{1,2,...,n\}))) \tag{A.69}$$

There are materials provides plenty of good examples of using folding, especially in [1], folding together with fusion law are well explained.

concatenate a list of list

In previous section A.3.6 about concatenation, we explained how to concatenate two lists. Actually, concatenation of lists can be considered equivalent to summation of numbers. Thus we can design a general algorithm, which can concatenate multiple lists into one big list.

What's more, we can realize this general concatenation by using folding. As sum can be represented as $sum(L) = foldr(+, 0, L)$, it's straightforward to write the following equation.

$$concats(L) = foldr(concat, \phi, L) \tag{A.70}$$

Where L is a list of list, for example $\{\{1,2,3\},\{4,5,6\},\{7,8,9\},...\}$. Function $concat(L_1, L_2)$ is what we defined in section A.3.6.

In some environments which support lazy-evaluation, such as Haskell, this algorithm is capable to concatenate infinite list of list, as the binary function `++` is lazy.

www.dbooks.org

Exercise A.4

- What's the performance of *concats* algorithm? is it linear or quadratic?
- Design another linear time *concats* algorithm without using folding.
- Realize mapping algorithm by using folding.

A.7 Searching and matching

Searching and matching are very important algorithms. They are not only limited to linked list, but also applicable to a wide range of data structures. We just scratch the surface of searching and matching in this appendix. There are dedicated chapters explain about them in this book.

A.7.1 Existence testing

The simplest searching case is to test if a given element exists in a list. A linear time traverse can solve this problem. In order to determine element x exists in list L:

- If the list is empty, it's obvious that the element doesn't exist in L;
- If the first element in the list equals to x, we know that x exists;
- Otherwise, we need recursively test if x exists in the rest sub-list L'.

This simple description can be directly formalized to equation as the following.

$$x \in L = \begin{cases} False & : \quad L = \phi \\ True & : \quad l_1 = x \\ x \in L' & : \quad otherwise \end{cases} \tag{A.71}$$

This is definitely a linear algorithm which is bound to $O(n)$ time. The best case happens in the two trivial clauses that either the list is empty or the first element is what we are finding; The worst case happens when the element doesn't exist at all or it is the last element. In both cases, we need traverse the whole list. If the element exists and the probability is equal for all the positions, the average case takes about $\frac{n+1}{2}$ steps for traversing.

This algorithm is so trivial that we left the implementation as exercise to the reader. If the list is ordered, one may expect to improve the algorithm to logarithm time but not linear. However, as we discussed, since list doesn't support constant time random accessing, binary search can't be applied here. There is a dedicated chapter in this book discusses how to evolve the linked list to binary tree to achieve quick searching.

A.7.2 Looking up

One extra step from existence testing is to find the interesting information stored in the list. There are two typical methods to augment extra data to the element. Since the linked list is chain of nodes, we can store satellite data in the node, then provide $key(n)$ to access the key of the node, $rest(n)$ for the rest sub-list, and $value(n)$ for the augmented data. The other method, is to pair the key and data, for example $\{(1, hello), (2, world), (3, foo), ...\}$. We'll introduce how to form such pairing list in later section.

The algorithm is almost as same as the existence testing, that it traverses the list, examines the key one by one. Whenever it finds a node which has the same key as what we are looking up, it stops, and returns the augmented data. It is obvious that this is linear strategy. If the satellite data is augmented to the node directly, the algorithm can be defined as the following.

$$lookup(x, L) = \begin{cases} \phi & : & L = \phi \\ value(l_1) & : & key(l_1) = x \\ lookup(x, L') & : & otherwise \end{cases} \tag{A.72}$$

In this algorithm, L is a list of nodes which are augmented with satellite data. Note that the first case actually means looking up failure, so that the result is empty. Some functional programming languages, such as Haskell, provide `Maybe` type to handle the possibility of fail. This algorithm can be slightly modified to handle the key-value pair list as well.

$$lookup(x, L) = \begin{cases} \phi & : & L = \phi \\ snd(l_1) & : & fst(l_1) = x \\ lookup(x, L') & : & otherwise \end{cases} \tag{A.73}$$

Here L is a list of pairs, functions $fst(p)$ and $snd(p)$ access the first part and second part of the pair respectively.

Both algorithms are in tail-call manner, they can be transformed to imperative looping easily. We left this as exercise to the reader.

A.7.3 finding and filtering

Let's take one more step ahead, looking up algorithm performs linear search by comparing the key of an element is equal to the given value. A more general case is to find an element matching a certain predicate. We can abstract this matching condition as a parameter for this generic linear finding algorithm.

$$find(p, L) = \begin{cases} \phi & : & L = \phi \\ l_1 & : & p(l_1) \\ find(p, L') & : & otherwise \end{cases} \tag{A.74}$$

The algorithm traverses the list by examining if the element satisfies the predicate p. It fails if the list is empty while there is still nothing found. This is handled in the first trivial edge case; If the first element in the list satisfies the condition, the algorithm returns the whole element (node), and user can further handle it as he like (either extract the satellite data or do whatever); otherwise, the algorithm recursively perform finding on the rest of the sub-list. Below is the corresponding Haskell example program.

www.dbooks.org

```
find _ [] = Nothing
find p (x:xs) = if p x then Just x else find p xs
```

Translating this to imperative algorithm is straightforward. Here we use 'NIL' to represent the fail case.

1: **function** FIND(p, L)
2: **while** $L \neq \phi$ **do**
3: **if** p(FIRST(L)) **then**
4: **return** FIRST(L)
5: $L \leftarrow$ REST(L)
6: **return** NIL

And here is the Python example of finding.

```
def find(p, xs):
    while xs is not None:
        if p(xs.key):
            return xs
        xs = xs.next
    return None
```

It is quite possible that there are multiple elements in the list which satisfy the precondition. The finding algorithm designed so far just picks the first one it meets, and stops immediately. It can be considered as a special case of finding all elements under a certain condition.

Another viewpoint of finding all elements with a given predicate is to treat the finding algorithm as a black box, the input to this box is a list, while the output is another list contains all elements satisfying the predicate. This can be called as filtering as shown in the below figure.



Figure A.4: The input is the original list $\{x_1, x_2, ..., x_n\}$, the output is a list $\{x'_1, x'_2, ..., x'_m\}$, that for $\forall x'_i$, predicate $p(x'_i)$ is satisfied.

This figure can be formalized in ZF expression form. However, we actually enumerate among list instead of a set.

$$filter(p, L) = \{x | x \in L \wedge p(x)\} \tag{A.75}$$

Some environment such as Haskell (and Python for any iterable), supports this form as list comprehension.

```
filter p xs = [ x | x ← xs, p x]
```

And in Python for built-in list as

```
def filter(p, xs):
    return [x for x in xs if p(x)]
```

Note that the Python built-in list isn't singly-linked list as we mentioned in this appendix.

In order to modify the finding algorithm to realize filtering, the found elements are appended to a result list. And instead of stopping the traverse, all the rest of elements should be examined with the predicate.

$$filter(p, L) = \begin{cases} \phi & : & L = \phi \\ cons(l_1, filter(p, L')) & : & p(l_1) \\ filter(p, L') & : & otherwise \end{cases} \tag{A.76}$$

This algorithm returns empty result if the list is empty for trivial edge case; For non-empty list, suppose the recursive result of filtering the rest of the sublist is A, the algorithm examine if the first element satisfies the predicate, it is put in front of A by a 'cons' operation ($O(1)$ time).

The corresponding Haskell program is given as below.

```
filter _ [] = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

Although we mentioned that the next found element is 'appended' to the result list, this algorithm actually constructs the result list from the right most to the left, so that appending is avoided, which ensure the linear $O(n)$ performance. Compare this algorithm with the following imperative quadratic realization reveals the difference.

```
1: function FILTER(p, L)
2:     L' ← φ
3:     while L ≠ φ do
4:         if p(FIRST(L)) then
5:             L' ← APPEND(L', FIRST(L))          ▷ Linear operation
6:         L ← REST(L)
```

As the comment of appending statement, it's typically proportion to the length of the result list if the tail position isn't memorized. This fact indicates that directly transforming the recursive filter algorithm into tail-call form will downgrade the performance from $O(n)$ to $O(n^2)$. As shown in the below equation, that $filter(p, L) = filter'(p, L, \phi)$ performs as poorly as the imperative one.

$$filter'(p, L, A) = \begin{cases} A & : & L = \phi \\ filter'(p, L', A \cup \{l_1\}) & : & p(l_1) \\ filter'(p, L', A) & : & otherwise \end{cases} \tag{A.77}$$

One solution to achieve linear time performance imperatively is to construct the result list in reverse order, and perform the $O(n)$ reversion again (refer to the above section) to get the final result. This is left as exercise to the reader.

The fact of construction the result list from right to left indicates the possibility of realizing filtering with folding-right concept. We need design some combinator function f, so that $filter(p, L) = foldr(f, \phi, L)$. It requires that function f takes two arguments, one is the element iterated among the list; the other is the intermediate result constructed from right. $f(x, A)$ can be defined as that it tests the predicate against x, if succeed, the result is updated to $cons(x, A)$, otherwise, A is kept same.

$$f(x, A) = \begin{cases} cons(x, A) & : & p(x) \\ A & : & otherwise \end{cases} \tag{A.78}$$

www.dbooks.org

However, the predicate must be passed to function f as well. This can be achieved by using currying, so f actually has the prototype $f(p, x, A)$, and filtering is defined as following.

$$filter(p, L) = foldr(\lambda_{x,A} \cdot f(p, x, A), \phi, L) \tag{A.79}$$

Which can be simplified by η-conversion. For detailed definition of η-conversion, readers can refer to [2].

$$filter(p, L) = foldr(f(p), \phi, L) \tag{A.80}$$

The following Haskell example program implements this equation.

```
filter p = foldr f [] where
    f x xs = if p x then x : xs else xs
```

Similar to mapping and folding, filtering is actually a generic concept, that we can apply a predicate on any traversable data structures to get what we are interesting. readers can refer to the topic about monoid in [8] for further reading.

A.7.4 Matching

Matching generally means to find a given pattern among some data structures. In this section, we limit the topic within list. Even this limitation will leads to a very wide and deep topic, that there are dedicated chapters in this book introduce matching algorithms. So we only select the algorithm to test if a given list exists in another (typically longer) list.

Before dive into the algorithm of finding the sub-list at any position, two special edge cases are used for warm up. They are algorithms to test if a given list is either prefix or suffix of another.

In the section about span, we have seen how to find a prefix under a certain condition. prefix matching can be considered as a special case in some sense. That it compares each of the elements between the two lists from the beginning until meets any different elements or pass the end of one list. Define $P \subseteq L$ if P is prefix of L.

$$P \subseteq L = \begin{cases} True & : & P = \phi \\ False & : & p_1 \neq l_1 \\ P' \subseteq L' & : & otherwise \end{cases} \tag{A.81}$$

This is obviously a linear algorithm. However, We can't use the very same approach to test if a list is suffix of another because it isn't cheap to start from the end of the list and keep iterating backwards. Arrays, on the other hand which support random access can be easily traversed backwards.

As we only need the yes-no result, one solution to realize a linear suffix testing algorithm is to reverse both lists, (which is linear time), and use prefix testing instead. Define $L \supseteq P$ if P is suffix of L.

$$L \supseteq P = reverse(P) \subseteq reverse(L) \tag{A.82}$$

With $\subseteq$ defined, it enables to test if a list is infix of another. The idea is to traverse the target list, and repeatedly applying the prefix testing till any success or arrives at the end.

```
1: function Is-Infix(P, L)
2:     while L ≠ φ do
3:         if P ⊆ L then
4:             return TRUE
5:         L ← Rest(L)
6:     return FALSE
```

Formalize this algorithm to recursive equation leads to the below definition.

$$infix?(P, L) = \begin{cases} True & : & P \subseteq L \\ False & : & L = \phi \\ infix?(P, L') & : & otherwise \end{cases} \tag{A.83}$$

Note that there is a tricky implicit constraint in this equation. If the pattern P is empty, it is definitely the infix of any target list. This case is actually covered by the first condition in the above equation because empty list is also the prefix of any list. In most programming languages support pattern matching, we can't arrange the second clause as the first edge case, or it will return false for $infix?(\phi, \phi)$. (One exception is Prolog, but this is a language specific feature, which we won't covered in this book.)

Since prefix testing is linear, and it is called while traversing the list, this algorithm is quadratic $O(nm)$. where n and m are the length of the pattern and target lists respectively. There is no trivial way to improve this 'position by position' scanning algorithm to linear even if the data structure changes from linked-list to randomly accessible array.

There are chapters in this book introduce several approaches for fast matching, including suffix tree with Ukkonen algorithm, Knuth-Morris-Pratt algorithm and Boyer-Moore algorithm.

Alternatively, we can enumerate all suffixes of the target list, and check if the pattern is prefix of any these suffixes. Which can be represented as the following.

$$infix?(P, L) = \exists S \in suffixes(L) \wedge P \subseteq S \tag{A.84}$$

This can be represented as list comprehension, for example the below Haskell program.

```
isInfixOf x y = (not ∘ null) [ s | s ← tails(y), x `isPrefixOf` s]
```

Where function `isPrefixOf` is the prefixing testing function defined according to our previous design. function `tails` generate all suffixes of a list. The implementation of `tails` is left as an exercise to the reader.

Exercise A.5

- Implement the linear existence testing in both functional and imperative approaches in your favorite programming languages.
- Implement the looking up algorithm in your favorite imperative programming language.
- Realize the linear time filtering algorithm by firstly building the result list in reverse order, and finally reverse it to resume the normal result.

www.dbooks.org

Implement this algorithm in both imperative looping and functional tail-recursion call.

- Implement the imperative algorithm of prefix testing in your favorite programming language.
- Implement the algorithm to enumerate all suffixes of a list.

A.8 zipping and unzipping

It is quite common to construct a list of paired elements. For example, in the naive brute-force solution for the 'light switching' puzzle which is shown in section of mapping, we need to represent the state of all lights. It is initialized as $\{(1,0),(2,0),...,(n,0)\}$. Another example is to build a key-value list, such as $\{(1,a),(2,an),(3,another),...\}$.

In 'light switching' example, the list of pairs is built like the following.

$$map(\lambda_i \cdot (i,0), \{1,2,...,n\})$$

The more general case is that, There have been already two lists prepared, what we need is a handy 'zipper' method.

$$zip(A,B) = \begin{cases} \phi & : \quad A = \phi \vee B = \phi \\ cons((a_1,b_1), zip(A',B')) & : \quad otherwise \end{cases} \tag{A.85}$$

Note that this algorithm is capable to handle the case that the two lists being zipped have different lengths. The result list of pairs aligns with the shorter one. And it's even possible to zip an infinite list with another one with limited length in environment support lazy evaluation. For example with this auxiliary function defined, we can initialize the lights state as

$$zip(\{0,0,...\}, \{1,2,...,n\}$$

In some languages support list enumeration, such as Haskell (Python provides similar `range` function, but it manipulates built-in list[9], which isn't linked-list actually), this can be expressed as `zip (repeat 0) [1..n]`. Given a list of words, we can also index them with consecutive numbers as

$$zip(\{1,2,...\}, \{a,an,another,...\})$$

Note that the zipping algorithm is linear, as it uses constant time 'cons' operation in each recursive call. However, directly translating *zip* into imperative manner would down-grade the performance to quadratic unless the linked-list is optimized with tail position cache or we in-place modify one of the passed-in list.

1: **function** ZIP(A, B)
2: $C \leftarrow \phi$
3: **while** $A \neq \phi \wedge B \neq \phi$ **do**

[9]To be precise, the `range` function in Python 2 will return a built-in list, while in Python 3, it will return a iterable.

```
        C ← APPEND(C, (FIRST(A), FIRST(B)))
        A ← REST(A)
        B ← REST(B)
    return C
```

Note that, the appending operation is proportion to the length of the result list C, so it will get more and more slowly along with traversing. There are three solutions to improve this algorithm to linear time. The first method is to use a similar approach as we did in infix-testing, that we construct the result list of pairs in reverse order by always insert the paired elements on head; then perform a linear reverse operation before return the final result; The second method is to modify one passed-in list, for example A, in-place while traversing. Translate it from list of elements to list of pairs; The third method is to remember the last appending position. Please try these solutions as exercise.

The key point of linear time zipping is that the result list is actually built from right to left, it's quite possible to provide a folding-right realization. This is left as exercise to the reader.

It is natural to extend the zipper algorithm so that multiple lists can be zipped to one list of multiple-elements. For example, Haskell standard library provides, `zip`, `zip3`, `zip4`, ..., till `zip7`. Another typical extension to zipper is that, sometimes, we don't want to list of pairs (or tuples more generally), instead, we want to apply some combinator function to each pair of elements.

For example, consider the case that we have a list of unit prices for every fruit: apple, orange, banana, ..., as $\{1.00, 0.80, 10.05, ...\}$, with same unit of Dollar; And the cart of customer holds a list of purchased quantity, for instance $\{3, 1, 0, ...\}$, means this customer, put 3 apples, an orange in the cart. He doesn't take any banana, so the quantity of banana is zero. We want to generate a list of cost for the customer, contains how much should pay for apple, orange, banana,... respectively.

The program can be written from scratch as below.

$$paylist(U, Q) = \begin{cases} \phi & : \quad U = \phi \vee Q = \phi \\ cons(u_1 \times q_1, paylist(U', Q')) & : \quad otherwise \end{cases}$$

Compare this equation with the zipper algorithm. It is easy to find the common structure of the two, and we can parameterize the combinator function as f, so that the 'generic' zipper algorithm can be defined as the following.

$$zipWith(f, A, B) = \begin{cases} \phi & : \quad A = \phi \vee B = \phi \\ cons(f(a_1, b_1), zipWith(f, A', B')) & : \quad otherwise \end{cases} \tag{A.86}$$

Here is an example that defines the inner-product (or dot-product)[14] by using $zipWith$.

$$A \cdot B = sum(zipWith(\times, A, B)) \tag{A.87}$$

It is necessary to realize the inverse operation of zipping, that converts a list of pairs, to different lists of elements. Back to the purchasing example, It is quite possible that the unit price information is stored in a association list like $U = \{(apple, 1.00), (orange, 0.80), (banana, 10.05), ...\}$, so that it's convenient

www.dbooks.org

to look up the price with a given product name, for instance, $lookup(melon, U)$. Similarly, the cart can also be represented clearly in such manner, for example, $Q = \{(apple, 3), (orange, 1), (banana, 0), ...\}$.

Given such a 'product - unit price' list and a 'product - quantity' list, how to calculate the total payment?

One straight forward idea derived from the previous solution is to extract the unit price list and the purchased quantity list, then calculate the inner-product of them.

$$pay = sum(zipWith(\times, snd(unzip(P)), snd(unzip(Q)))) \tag{A.88}$$

Although the definition of *unzip* can be directly written as the inverse of *zip*, here we give a realization based on folding-right.

$$unzip(L) = foldr(\lambda_{(a,b),(A,B)} \cdot (cons(a, A), cons(b, B)), (\phi, \phi), L) \tag{A.89}$$

The initial result is a pair of empty list. During the folding process, the head of the list, which is a pair of elements, as well as the intermediate result are passed to the combinator function. This combinator function is given as a lambda expression, that it extracts the paired elements, and put them in front of the two intermediate lists respectively. Note that we use implicit pattern matching to extract the elements from pairs. Alternatively this can be done by using fst, and snd functions explicitly as

$$\lambda_{p,P} \cdot (cons(fst(p), fst(P)), cons(snd(p), snd(P)))$$

The following Haskell example code implements *unzip* algorithm.

```
unzip = foldr λ(a, b) (as, bs) → (a:as, b:bs) ([], [])
```

Zip and unzip concepts can be extended more generally rather than only limiting within linked-list. It is quite useful to zip two lists to a tree, where the data stored in the tree are paired elements from both lists. General zip and unzip can also be used to track the traverse path of a collection to mimic the 'parent' pointer in imperative implementations. Please refer to the last chapter of [8] for a good treatment.

Exercise A.6

- Design and implement iota (I) algorithm, which can enumerate a list with some given parameters. For example:
  - $iota(..., n) = \{1, 2, 3, ..., n\}$;
  - $iota(m, n) = \{m, m + 1, m + 2, ..., n\}$, Where $m \leq n$;
  - $iota(m, m + a, ..., n) = \{m, m + a, m + 2a, ..., n\}$;
  - $iota(m, m, ...) = repeat(m) = \{m, m, m, ...\}$;
  - $iota(m, ...) = \{m, m + 1, m + 2, ...\}$.

  Note that the last two cases demand generate infinite list essentially. Consider how to represents infinite list? You may refer to the streaming and lazy evaluation materials such as [5] and [8].

- Design and implement a linear time imperative zipper algorithm.
- Realize the zipper algorithm with folding-right approach.
- For the purchase payment example, suppose the quantity association list only contains those items with the quantity isn't zero, that instead of a list of $Q = \{(apple, 3), (banana, 0), (orange, 1), ...\}$, it hold a list like $Q = \{(apple, 3), (orange, 1), ...\}$. The 'banana' information is filtered because the customer doesn't pick any bananas. Write a program, taking the unit-price association list, and this kind of quantity list, to calculate the total payment.

A.9 Notes and short summary

In this appendix, a quick introduction about how to build, manipulate, transfer, and searching singly linked list is briefed in both purely functional and imperative approaches. Most of the modern programming environments have been equipped with tools to handle such elementary data structures. However, such tools are designed for general purpose cases, Serious programming shouldn't take them as black-boxes.

Since linked-list is so critical that it builds the corner stones for almost all functional programming environments, just like the importance of array to imperative settings. We take this topic as an appendix to the book. It is quite OK that the reader starts with the first chapter about binary search tree, which is a kind of 'hello world' topic, and refers to this appendix when meets any unfamiliar list operations.

Exercise A.7

- Develop a program to remove the duplicated elements in a linked-list. In imperative settings, the duplicated elements should be removed in-place. In purely functional settings, construct a new list contains the unique elements. The order of the elements should be kept as their origianl appearance. What is the complexity of the program? Try to simplify the solution if auxiliary data structures are allowed.
- A decimal non-negative integer can be represented in linked-list. For example 1024 can be represented as '$4 \rightarrow 2 \rightarrow 0 \rightarrow 1$'. Generally, $n = d_m...d_2d_1$ can be represented as '$d_1 \rightarrow d_2 \rightarrow ... \rightarrow d_m$'. Given two numbers a, b in linked-list form. Realize basic arithmetic operations such as plus and minus.
- In imperative settings, a linked-list may be corrupted, that it is circular. In such list, some node points back to previous one. Figure A.5 shows such situation. The normal iteration ends up infinite looping.
  1. Write a program to detect if a linked-list is circular;
  2. Write a program to find the node where the loop starts (the node being pointed by two precedents).

www.dbooks.org

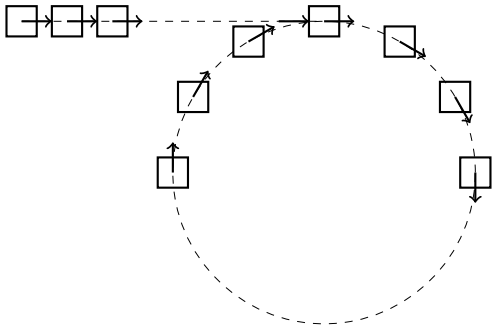


Figure A.5: A circular linked-list

Bibliography

[1] Richard Bird. “Pearls of Functional Algorithm Design”. Cambridge University Press; 1 edition (November 1, 2010). ISBN: 978-0521513388

[2] Simon L. Peyton Jones. “The Implementation of Functional Programming Languages”. Prentice-Hall International Series in Computer Since. Prentice Hall (May 1987). ISBN: 978-0134533339

[3] Andrei Alexandrescu. “Modern C++ design: Generic Programming and Design Patterns Applied”. Addison Wesley February 01, 2001, ISBN 0-201-70431-5

[4] Benjamin C. Pierce. “Types and Programming Languages”. The MIT Press, 2002. ISBN:0262162091

[5] Harold Abelson, Gerald Jay Sussman, Julie Sussman. “Structure and Interpretation of Computer Programs, 2nd Edition”. MIT Press, 1996, ISBN 0-262-51087-1

[6] Chris Okasaki. “Purely Functional Data Structures”. Cambridge university press, (July 1, 1999), ISBN-13: 978-0521663502

[7] Fethi Rabhi, Guy Lapalme. “Algorithms: a functional programming approach”. Second edition. Addison-Wesley, 1999. ISBN: 0201-59604-0

[8] Miran Lipovaca. “Learn You a Haskell for Great Good! A Beginner’s Guide”. No Starch Press; 1 edition April 2011, 400 pp. ISBN: 978-1-59327-283-8

[9] Joe Armstrong. “Programming Erlang: Software for a Concurrent World”. Pragmatic Bookshelf; 1 edition (July 18, 2007). ISBN-13: 978-1934356005

[10] Wikipedia. “Tail call”. https://en.wikipedia.org/wiki/Tail_call

[11] SGI. “transform”. http://www.sgi.com/tech/stl/transform.html

[12] ACM/ICPC. “The drunk jailer.” Peking University judge online for ACM/ICPC. http://poj.org/problem?id=1218.

[13] Haskell wiki. “Haskell programming tips”. 4.4 Choose the appropriate fold. http://www.haskell.org/haskellwiki/Haskell_programming_tips

[14] Wikipedia. “Dot product”. http://en.wikipedia.org/wiki/Dot_product

www.dbooks.org

Appendix B

The imperative red-black tree deletion algorithm

This appendix introduces the imperative red-black tree deletion algorithm. The idea is to perform the tree rotations and re-coloring on top of the binary search tree deletion result to adjust the balance of the tree. As mentioned in the chapter of red-black tree, the 5th red-black tree property will be violated when deleting a black node. This is because the number of black nodes in the paths through that node will reduce by one. To keep the number of black nodes unchanged, we introduce a concept of 'doubly blackness'.

B.1 Doubly Black node

In order to realize 'doubly black' node, we need add a new color definition. As the following C++ example.

```
enum class Color { RED, BLACK, DOUBLY_BLACK };
```

When deleting a node, we re-use the binary search tree algorithm and record the parent of the deleted node. We need further processing if the deleted node is black before performing the fixing.

```
1:  function DELETE(T, x)
2:      p ← PARENT(x)
3:      q ← NIL
4:      if LEFT(x) = NIL then
5:          q ← RIGHT(x)
6:          replace x with RIGHT(x)
7:      else if RIGHT(x) = NIL then
8:          q ← LEFT(x)
9:          replace x with LEFT(x)
10:     else
11:         y ← MIN(RIGHT(x))
12:         p ← PARENT(y)
13:         q ← RIGHT(y)
14:         KEY(x) ← KEY(y)
15:         copy satellite data from y to x
```


www.dbooks.org

```
16:         replace y with RIGHT(y)
17:         x ← y
18:     if COLOR(x) = BLACK then
19:         T ← DELETE-FIX(T, MAKE-BLACK(p, q), q = NIL?)
20:     release x
21:     return T
```

The deletion algorithm takes the root T and the node x to be deleted. If x has an empty sub-tree, we can cut off x, then replace it with the other sub-tree q. Otherwise, we locate the minimum node y in the right child of x, then replace x with y. We next cut off y after that. If the deleted node x is black, we call MAKE-BLACK(p, q) to keep the blackness and start further processing.

```
1: function MAKE-BLACK(p, q)
2:     if p = NIL ∧ q = NIL then
3:         return NIL                    ▷ Returns empty if delete a singleton leaf.
4:     else if q = NIL then
5:         n ← Doubly Black NIL
6:         PARENT(n) ← p
7:         return n
8:     else
9:         return BLACKEN(q)
```

If both p and q passed to MAKE-BLACK are empty, we are deleting the only leaf node from a singleton tree. It becames empty as the result. If the parent p isn't empty, but q is, it means we deleted a black leaf node, and used the NIL node to replace that black node. According to the red-black tree property 3, NIL node is black. We can change the NIL node to 'doubly black' NIL to retain the number of black nodes for all the paths through that node. Otherwise, if both p and q are not empty, we call BLACKEN procedure, if q is red, then we color it to black; if it is already black, we color it to doubly black.

In order to resume all the red-black tree properties, we next get rid of the doubly blackness by tree rotations and re-coloring. There are three different cases (pp292 in [2]). The doubly black node can either be normal node or NIL node in all the cases.

B.1.1 The doubly black node has a black sibling, and one of its nephew is red.

We can rotate the tree to fix the doubly black. There are 4 sub-cases, all can be transformed to a uniformed structure as shown in figure **??**.

The following algorithm realized this fixing.

```
1: function DELETE-FIX(T, x, f)
2:     n ← NIL
3:     if f = True then                          ▷ x is a doubly black NIL
4:         n ← x
5:     if x = NIL then                           ▷ Delete the singleton leaf
6:         return NIL
7:     while x ≠ T∧ COLOR(x) = B² do    ▷ x isn't root and is doubly black
8:         if SIBLING(x) ≠ NIL then              ▷ The sibling isn't empty
9:             s ← SIBLING(x)
```

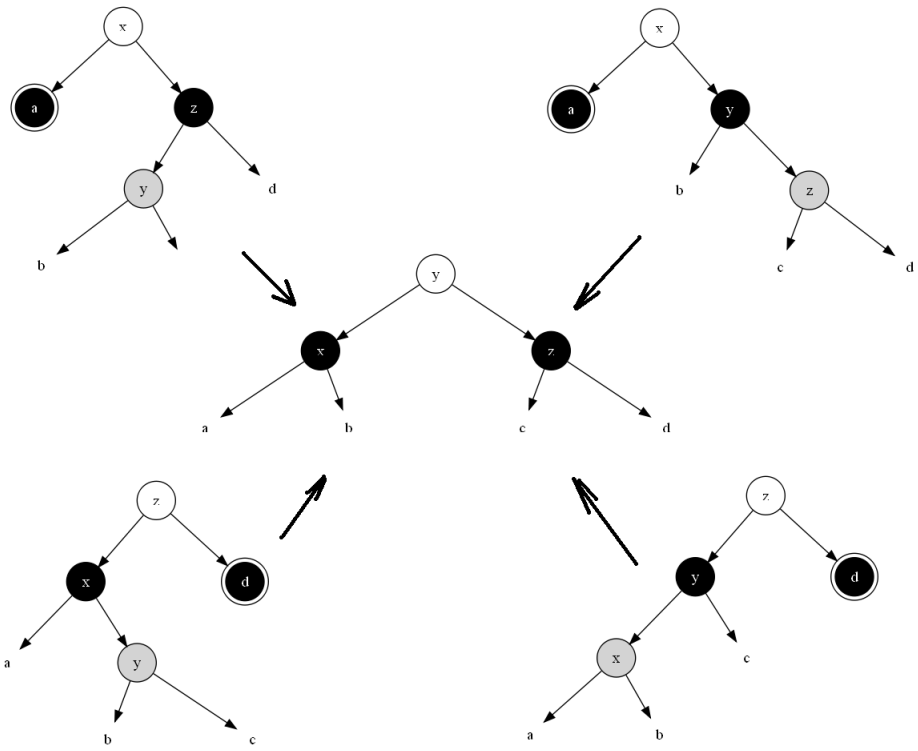


Figure B.1: The doubly black node has a black sibling, and one of its nephew is red. It can be fixed with a rotation.

www.dbooks.org

```
10:             ...
11:             if s is black ∧ LEFT(s) is red then        ▷ The sibling is black, a
    nephew is red
12:                 if x = LEFT(PARENT(x)) then                ▷ x is the left
13:                     set x, PARENT(x), and LEFT(s) all black
14:                     T ← ROTATE-RIGHT(T, s)
15:                     T ← ROTATE-LEFT(T, PARENT(x))
16:                 else                                       ▷ x is the right
17:                     set x, PARENT(x), s, and LEFT(s) all black
18:                     T ← ROTATE-RIGHT(T, PARENT(x))
19:             else if s is black ∧ RIGHT(s) is red then ▷ The sibling is black,
    a nephew is red
20:                 if x = LEFT(PARENT(x)) then                ▷ x is the left
21:                     set x, PARENT(x), s, and RIGHT(s) all black
22:                     T ← ROTATE-LEFT(T, PARENT(x))
23:                 else                                       ▷ x is the right
24:                     set x, PARENT(x), and RIGHT(s) all black
25:                     T ← ROTATE-LEFT(T, s)
26:                     T ← ROTATE-RIGHT(T, PARENT(x))
27:                 ...
```

B.1.2 The sibling of the doubly black node is red.

We can rotate the tree to change the doubly black node to normal black node. As shown in figure B.2, after transform from the left to the right structure, we can re-color node a or c to black.

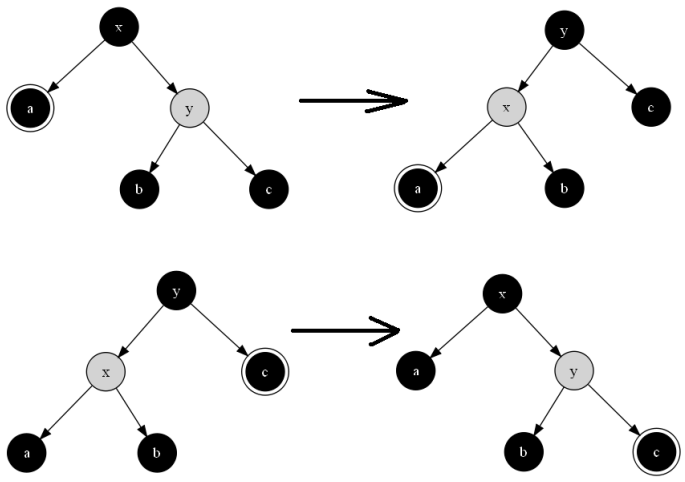


Figure B.2: The sibling of the doubly black node is red

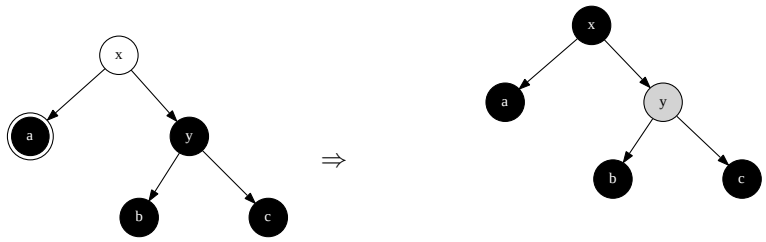
We can add this fixing on top of the previous algorithm.

```
1: function DELETE-FIX(T, x, f)
2:     n ← NIL
3:     if f = True then                                ▷ x is a doubly black NIL
```

```
4:        n ← x
5:    if x = NIL then                                  ▷ Delete the singleton leaf
6:        return NIL
7:    while x ≠ T∧ Color(x) = B² do    ▷ x isn't root and is doubly black
8:        if Sibling(x) ≠ NIL then                     ▷ The sibling isn't empty
9:            s ← Sibling(x)
10:           if s is red then                         ▷ The sibling is red
11:               set Parent(x) red
12:               set s black
13:               if x = Left(Parent(x)) then          ▷ x is the left
14:                   T ← Rotate-LeftT, Parent(x)
15:               else                                 ▷ x is the right
16:                   T ← Rotate-RightT, Parent(x)
17:           else if s is black ∧ Left(s) is red then ▷ THe sibling is black, a
    nephew is red
18:               ...
```

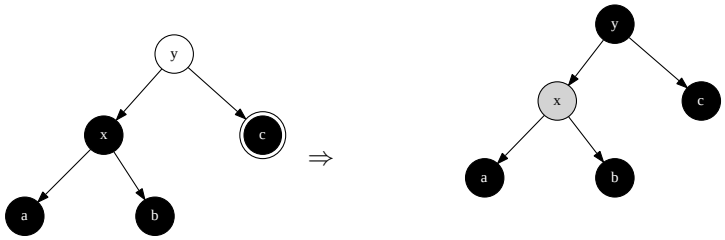
B.1.3 The sibling of the doubly black node, and both nephews are black.

In this case, we can re-color the sibing to red, change the doubly black node back to black, then propagate the doubly blackness to the parent in upper level. As shown in figure B.3, there are two symetric sub-cases.



(a) x is red or black.

(b) Change x to black if it's red, otherwise change it to doubly black.



(c) y is red or black.

(d) Change y to black if it's red, otherwsie change it to doubly black.

Figure B.3: Propagate the doubly blackness upward

www.dbooks.org

The sibling of the doubly black node isn't empty in all the 3 cases. If the sibling is empty, we can change the doubly black node back to back, and propagate the blackness upward. If the doubly blackness reach the root, we can change the root to black to terminate the fixing process. It can also terminate if the doubly black node is re-colored in the midway. At last of the algorithm, if the doubly black node passed in is a doubly black empty node, we turn it back to normal NIL node. The complete altorithm is given as below.

```
1: function DELETE-FIX(T, x, f)
2:     n ← NIL
3:     if f = True then                                  ▷ x is a doubly black NIL
4:         n ← x
5:     if x = NIL then                                   ▷ Delete the singleton leaf
6:         return NIL
7:     while x ≠ T ∧ COLOR(x) = B² do                    ▷ x isn't root and is doubly black
8:         if SIBLING(x) ≠ NIL then                      ▷ The sibling isn't empty
9:             s ← SIBLING(x)
10:            if s is red then                          ▷ The sibling is red
11:                set PARENT(x) red
12:                set s black
13:                if x = LEFT(PARENT(x)) then           ▷ x is the left
14:                    T ← ROTATE-LEFTT, PARENT(x)
15:                else                                  ▷ x is the right
16:                    T ← ROTATE-RIGHTT, PARENT(x)
17:            else if s is black ∧ LEFT(s) is red then ▷ The sibling is black, a nephew is red
18:                if x = LEFT(PARENT(x)) then           ▷ x is the left
19:                    set x, PARENT(x), and LEFT(s) all black
20:                    T ← ROTATE-RIGHT(T, s)
21:                    T ← ROTATE-LEFT(T, PARENT(x))
22:                else                                  ▷ x is the right
23:                    set x, PARENT(x), s, and LEFT(s) all black
24:                    T ← ROTATE-RIGHT(T, PARENT(x))
25:            else if s is black ∧ RIGHT(s) is red then ▷ The sibling is black, a nephew is red
26:                if x = LEFT(PARENT(x)) then           ▷ x is the left
27:                    set x, PARENT(x), s, and RIGHT(s) all black
28:                    T ← ROTATE-LEFT(T, PARENT(x))
29:                else                                  ▷ x is the right
30:                    set x, PARENT(x), and RIGHT(s) all black
31:                    T ← ROTATE-LEFT(T, s)
32:                    T ← ROTATE-RIGHT(T, PARENT(x))
33:            else if s, LEFT(s), and RIGHT(s) are all black then   ▷ The sibling and nephews are all black
34:                set x black
35:                set s red
36:                BLACKEN(PARENT(x))
37:                x ← PARENT(x)
38:        else                                          ▷ No sibling, propagate the blackness upward
```

39: set x black
40: BLACKEN(PARENT(x))
41: $x \leftarrow$ PARENT(x)
42: set T black
43: **if** $n \neq$ NIL **then** ▷ Replace the doubly black NIL with normal NIL
44: replace n with NIL
45: **return** T

When fixing the deltion result, we passed in the root T; the node (can be doubly black) to be fixed as x; and a flag f. The flag is true if the node to be fixed x is doubly black NIL. We record the doubly black NIL with n, and replace n with the normal NIL after fixing.

The following is the example deletion program in C++.

```
Node* del(Node* t, Node* x) {
    if (!x) return t;
    Node* parent = x→parent;
    Node* db = nullptr;         //doubly black
    Node* y;

    if (x→left == nullptr) {
        db = x→right;
        x→replaceWith(db);
    } else if (x→right == nullptr) {
        db = x→left;
        x→replaceWith(db);
    } else {
        y = min(x→right);
        parent = y→parent;
        db = y→right;
        x→key = y→key;
        y→replaceWith(db);
        x = y;
    }
    if (x→color == Color::BLACK)
        t = deleteFix(t, makeBlack(parent, db), db == nullptr);
    remove(x);
    return t;
}
```

Where `makeBlack` check if the node change to doubly black after deletion, and handle the special case of doubly black NIL.

```
Node* makeBlack(Node* parent, Node* x) {
    if (!parent && ! x)
        return nullptr;
    if (!x)
        return Node::replace(parent, x, new Node(0, Color::DOUBLY_BLACK));
    return blacken(x);
}
```

The function `replace(parent, x, y)` replaces the child of the `parent`, which is `x`, with the node `y`.

```
static Node* replace(Node* parent, Node* x, Node* y) {
    if (parent == nullptr) {
```

www.dbooks.org

```
        if (y) y→parent = nullptr;
    } else if (parent→left == x) {
        parent→setLeft(y);
    } else {
        parent→setRight(y);
    }
    if (x) x→parent = nullptr;
    return y;
}
```

The function `blacken(node)` re-colors the red node to black, and re-colors the black node to doubly black.

```
Node* blacken(Node* x) {
    x→color = isRed(x) ? Color::BLACK : Color::DOUBLY_BLACK;
    return x;
}
```

Below is the final fixing process.

```
Node* deleteFix(Node* t, Node* db, bool isDBEmpty) {
    Node* dbEmpty = isDBEmpty ? db : nullptr;
    if (!db) return nullptr;    // remove the root from a leaf tree;
    while (db != t && db→color == Color::DOUBLY_BLACK) {
        if (db→sibling() != nullptr) {
            if (isRed(db→sibling())) {
                // the sibling is red, (transform to make the sibling black)
                setColors(db→parent, Color::RED,
                          db→sibling(), Color::BLACK);
                if (db == db→parent→left)
                    t = leftRotate(t, db→parent);
                else
                    t = rightRotate(t, db→parent);
            } else if (isBlack(db→sibling()) && isRed(db→sibling()→left)) {
                // the sibling is black, and one nephew is red
                if (db == db→parent→left) {
                    setColors(db, Color::BLACK,
                              db→parent, Color::BLACK,
                              db→sibling()→left, db→parent→color);
                    t = rightRotate(t, db→sibling());
                    t = leftRotate(t, db→parent);
                } else {
                    setColors(db, Color::BLACK,
                              db→parent, Color::BLACK,
                              db→sibling(), db→parent→color,
                              db→sibling()→left, Color::BLACK);
                    t = rightRotate(t, db→parent);
                }
            } else if (isBlack(db→sibling()) && isRed(db→sibling()→right)) {
                if (db == db→parent→left) {
                    setColors(db, Color::BLACK,
                              db→parent, Color::BLACK,
                              db→sibling(), db→parent→color,
                              db→sibling()→right, Color::BLACK);
                    t = leftRotate(t, db→parent);
                } else {
```

```
                setColors(db, Color::BLACK,
                          db→parent, Color::BLACK,
                          db→sibling()→right, db→parent→color);
                t = leftRotate(t, db→sibling());
                t = rightRotate(t, db→parent);
            }
        } else if (isBlack(db→sibling()) &&
                   isBlack(db→sibling()→left) &&
                   isBlack(db→sibling()→right)) {
            // the sibling and both nephews are black.
            // move the blackness up
            setColors(db, Color::BLACK,
                      db→sibling(), Color::RED);
            blacken(db→parent);
            db = db→parent;
        }
    } else { // no sibling, move the blackness up
        db→color = Color::BLACK;
        blacken(db→parent);
        db = db→parent;
    }
}
t→color = Color::BLACK;
if (dbEmpty) { // change the doubly black nil to nil
    dbEmpty→replaceWith(nullptr);
    delete dbEmpty;
}
return t;
}
```

Where `isBlack(node)` tests if a node is black, the NIL node is also black according to the red-black tree property.

```
bool isBlack(Node* x) {
    return x == nullptr || x→color == Color::BLACK;
}

bool isRed(Node* x) {
    return x != nullptr && x→color == Color::RED;
}
```

The `setColors` is a set of auxiliary functions to re-color nodes.

```
void setColors(Node* x, Color a, Node* y, Color b) { x→color = a; y→color =
b; }

void setColors(Node* x, Color a, Node* y, Color b, Node* z, Color c) {
    setColors(x, a, y, b);
    z→color = c;
}

void setColors(Node* x, Color a, Node* y, Color b,
               Node* z, Color c, Node* q, Color d) {
    setColors(x, a, y, b);
    setColors(z, c, q, d);
}
```

www.dbooks.org

Before returning the final result, the algorithm check the doubly black NIL, it calls the `replaceWith` method of the node, which is implemented with the `replace` function defined previously.

```
void replaceWith(Node* y) {
    replace(parent, this, y);
}
```

The algorithm terminates when reach the root or the doubly blackness is eliminated. Consider the red-black tree is balanced, the deletion algorithm is bound to $O(\lg n)$ time for the tree contains n nodes.

Exercise B.1

- Implement the red-black tree deletion algorithm in an imperative programming langauge.
- Write a program in an imperative programming language to test if a given tree satisfies the 5 red-black tree properties. Use this program to verify the red-black tree deletion algorithm.

Appendix C

AVL tree - proofs and deletion algorithm

This appendix provides proofs to result we uses in the AVL tree pattern matching insert algorithm.

C.1 Height increment after insertion

When insert a key to the AVL tree, the increment to the height can be deduced into four cases given as below.

$$\begin{aligned}\Delta H &= |T'| - |T| \\ &= \begin{cases} \Delta H_r & : \Delta \geq 0 \wedge \Delta' \geq 0 \\ \Delta + \Delta H_r & : \Delta \leq 0 \wedge \Delta' \geq 0 \\ \Delta H_l - \Delta & : \Delta \geq 0 \wedge \Delta' \leq 0 \\ \Delta H_l & : otherwise \end{cases}\end{aligned}$$

To prove this equation, note the fact that the height can't increase both in left and right with only one insertion.

These 4 cases can be explained from the balance factor definition that it equals to the difference of the right and left sub trees.

- If $\Delta \geq 0$ and $\Delta' \geq 0$, it means that the height of right sub tree isn't less than the left sub tree before and after insertion. In this case, the increment in height of the tree only is 'contributed' from the right sub tree, which is ΔH_r.
- If $\Delta \leq 0$, it means the height of left sub tree isn't less than the right before. Since $\Delta' \geq 0$ after insertion, we know the height of right sub tree increases, and the left side keeps same ($|T_l'| = |T_l|$). The increment in height is as the following.

$$\begin{aligned}\Delta H &= max(|T_r'|, |T_l'|) - max(|T_r|, |T_l|) && \{\Delta \leq 0 \wedge \Delta' \geq 0\} \\ &= |T_r'| - |T_l| && \{|T_l| = |T_l'|\} \\ &= |T_r| + \Delta H_r - |T_l| \\ &= \Delta + \Delta H_r\end{aligned}$$

www.dbooks.org

- For the case $\Delta \geq 0$ and $\Delta' \leq 0$, similar as the above one, we can get the below result.

$$\begin{aligned} \Delta H &= max(|T'_r|, |T'_l|) - max(|T_r|, |T_l|) \quad \{\Delta \geq 0 \wedge \Delta' \leq 0\} \\ &= |T'_l| - |T_r| \\ &= |T_l| + \Delta H_l - |T_r| \\ &= \Delta H_l - \Delta \end{aligned}$$

- For the last case, both Δ and Δ' are no bigger than zero. It means the height of left sub tree is always greater than or equal to the right. The increment in height is only 'contributed' from the left sub tree, which is ΔH_l.

C.2 Proof to the balance adjustment after insertion

Given the 4 cases need fix as shown in figure 4.3, The balance factors are either -2, or +2. After adjustment, the factor resumes to 0. The left and right sub trees have the same height.

These four cases are left-left lean, right-right lean, right-left lean, and left-right lean cases. We denote the balancing factor before fixing as $\delta(x), \delta(y)$, and $\delta(z)$, while after fixing, they changes to $\delta'(x), \delta'(y)$, and $\delta'(z)$ respectively.

We'll next prove that, after fixing, we have $\delta(y) = 0$ for all four cases, and we'll provide the result values of $\delta'(x)$ and $\delta'(z)$.

Left-left lean case

As the structure of sub tree x doesn't change due to fixing, we immediately get $\delta'(x) = \delta(x)$.

Since $\delta(y) = -1$ and $\delta(z) = -2$, we have

$$\begin{aligned} \delta(y) = |C| - |x| = -1 \Rightarrow |C| = |x| - 1 \\ \delta(z) = |D| - |y| = -2 \Rightarrow |D| = |y| - 2 \end{aligned} \tag{C.1}$$

After fixing.

$$\begin{aligned} \delta'(z) &= |D| - |C| && \{From(C.1)\} \\ &= |y| - 2 - (|x| - 1) \\ &= |y| - |x| - 1 && \{x \text{ is child of } y \Rightarrow |y| - |x| = 1\} \\ &= 0 \end{aligned} \tag{C.2}$$

For $\delta'(y)$, we have the following fact after fixing.

$$\begin{aligned} \delta'(y) &= |z| - |x| \\ &= 1 + max(|C|, |D|) - |x| && \{\text{By (C.2), we have} |C| = |D|\} \\ &= 1 + |C| - |x| && \{\text{By (C.1)}\} \\ &= 1 + |x| - 1 - |x| \\ &= 0 \end{aligned} \tag{C.3}$$

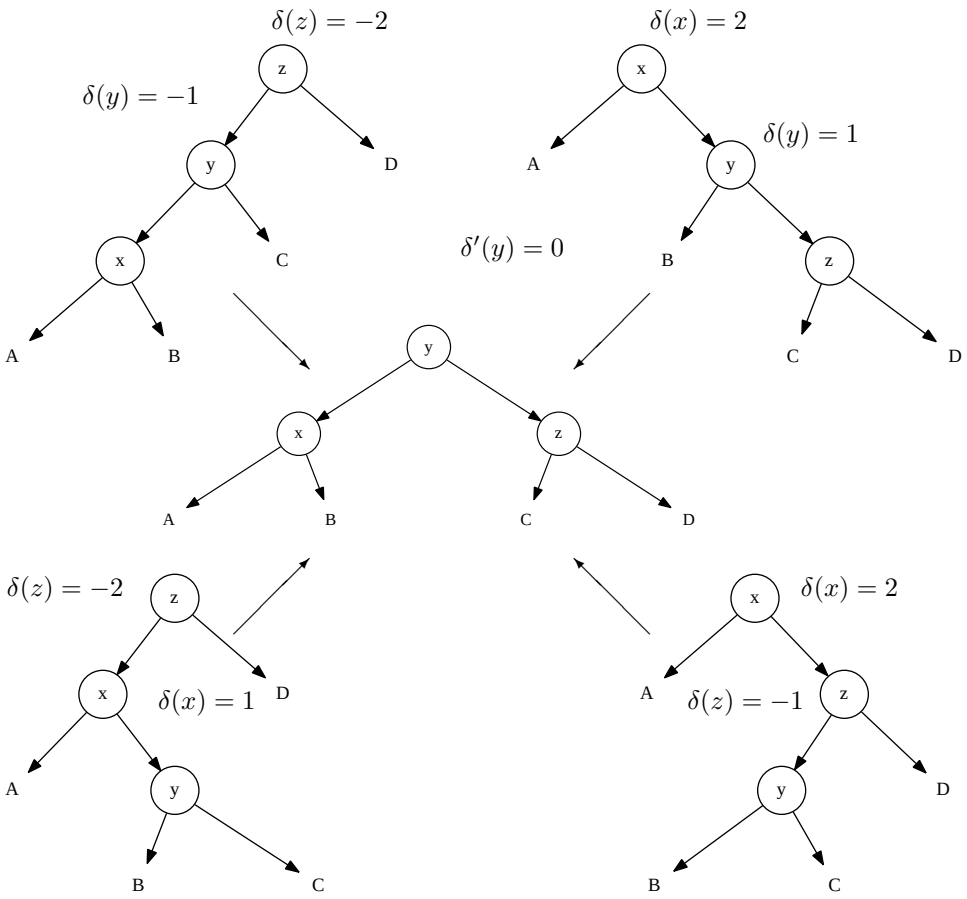


Figure C.1: 4 cases for balancing a AVL tree after insertion

www.dbooks.org

Summarize the above results, the left-left lean case adjust the balancing factors as the following.

$$\begin{aligned} \delta'(x) &= \delta(x) \\ \delta'(y) &= 0 \\ \delta'(z) &= 0 \end{aligned} \tag{C.4}$$

Right-right lean case

Since right-right case is symmetric to left-left case, we can easily achieve the result balancing factors as

$$\begin{aligned} \delta'(x) &= 0 \\ \delta'(y) &= 0 \\ \delta'(z) &= \delta(z) \end{aligned} \tag{C.5}$$

Right-left lean case

First let's consider $\delta'(x)$. After balance fixing, we have.

$$\delta'(x) = |B| - |A| \tag{C.6}$$

Before fixing, if we calculate the height of z, we can get.

$$\begin{aligned} |z| &= 1 + max(|y|, |D|) && \{\delta(z) = -1 \Rightarrow |y| > |D|\} \\ &= 1 + |y| \\ &= 2 + max(|B|, |C|) \end{aligned} \tag{C.7}$$

While since $\delta(x) = 2$, we can deduce that.

$$\begin{aligned} \delta(x) = 2 \quad &\Rightarrow |z| - |A| = 2 && \{\text{By (C.7)}\} \\ &\Rightarrow 2 + max(|B|, |C|) - |A| = 2 \\ &\Rightarrow max(|B|, |C|) - |A| = 0 \end{aligned} \tag{C.8}$$

If $\delta(y) = 1$, which means $|C| - |B| = 1$, it means

$$max(|B|, |C|) = |C| = |B| + 1 \tag{C.9}$$

Take this into (C.8) yields

$$\begin{aligned} &|B| + 1 - |A| = 0 \Rightarrow |B| - |A| = -1 \quad \{\text{By (C.6) }\} \\ &\Rightarrow \delta'(x) = -1 \end{aligned} \tag{C.10}$$

If $\delta(y) \neq 1$, it means $max(|B|, |C|) = |B|$, taking this into (C.8), yields.

$$\begin{aligned} &|B| - |A| = 0 \quad \{\text{By (C.6)}\} \\ &\Rightarrow \delta'(x) = 0 \end{aligned} \tag{C.11}$$

Summarize these 2 cases, we get relationship of $\delta'(x)$ and $\delta(y)$ as the following.

$$\delta'(x) = \begin{cases} -1 & : \quad \delta(y) = 1 \\ 0 & : \quad otherwise \end{cases} \tag{C.12}$$

For $\delta'(z)$ according to definition, it is equal to.

$$\begin{array}{lll} \delta'(z) & = |D| - |C| & \{\delta(z) = -1 = |D| - |y|\} \\ & = |y| - |C| - 1 & \{|y| = 1 + max(|B|, |C|)\} \\ & = max(|B|, |C|) - |C| & \end{array} \tag{C.13}$$

If $\delta(y) = -1$, then we have $|C| - |B| = -1$, so $max(|B|, |C|) = |B| = |C| + 1$. Takes this into (C.13), we get $\delta'(z) = 1$.

If $\delta(y) \neq -1$, then $max(|B|, |C|) = |C|$, we get $\delta'(z) = 0$.

Combined these two cases, the relationship between $\delta'(z)$ and $\delta(y)$ is as below.

$$\delta'(z) = \begin{cases} 1 & : \quad \delta(y) = -1 \\ 0 & : \quad otherwise \end{cases} \tag{C.14}$$

Finally, for $\delta'(y)$, we deduce it like below.

$$\begin{array}{ll} \delta'(y) & = |z| - |x| \\ & = max(|C|, |D|) - max(|A|, |B|) \end{array} \tag{C.15}$$

There are three cases.

- If $\delta(y) = 0$, it means $|B| = |C|$, and according to (C.12) and (C.14), we have $\delta'(x) = 0 \Rightarrow |A| = |B|$, and $\delta'(z) = 0 \Rightarrow |C| = |D|$, these lead to $\delta'(y) = 0$.
- If $\delta(y) = 1$, From (C.14), we have $\delta'(z) = 0 \Rightarrow |C| = |D|$.

$$\begin{array}{lll} \delta'(y) & = max(|C|, |D|) - max(|A|, |B|) & \{|C| = |D|\} \\ & = |C| - max(|A|, |B|) & \{\text{From (C.12): } \delta'(x) = -1 \Rightarrow |B| - |A| = -1\} \\ & = |C| - (|B| + 1) & \{\delta(y) = 1 \Rightarrow |C| - |B| = 1\} \\ & = 0 & \end{array}$$

- If $\delta(y) = -1$, From (C.12), we have $\delta'(x) = 0 \Rightarrow |A| = |B|$.

$$\begin{array}{lll} \delta'(y) & = max(|C|, |D|) - max(|A|, |B|) & \{|A| = |B|\} \\ & = max(|C|, |D|) - |B| & \{\text{From (C.14): } |D| - |C| = 1\} \\ & = |C| + 1 - |B| & \{\delta(y) = -1 \Rightarrow |C| - |B| = -1\} \\ & = 0 & \end{array}$$

All three cases lead to the same result that $\delta'(y) = 0$.

Collect all the above results, we get the new balancing factors after fixing as the following.

$$\begin{array}{l} \delta'(x) = \begin{cases} -1 & : \quad \delta(y) = 1 \\ 0 & : \quad otherwise \end{cases} \\ \delta'(y) = 0 \\ \delta'(z) = \begin{cases} 1 & : \quad \delta(y) = -1 \\ 0 & : \quad otherwise \end{cases} \end{array} \tag{C.16}$$

www.dbooks.org

Left-right lean case

Left-right lean case is symmetric to the Right-left lean case. By using the similar deduction, we can find the new balancing factors are identical to the result in (C.16).

C.3 Deletion algorithm

Deletion can reduce the height of the sub tree where the node is removed. If the balance factor exceeds the range of $[-1, 1]$, then we need fix to resume the AVL properties.

C.3.1 Functional deletion

We can reuse most of the deletion algorithm for BST, then check the balance factor and perform fixing. Similiar to the insertion algorithm, the result after deletion is a pair $(T', \Delta H)$, where T' is the new tree and ΔH is the decrement of the height. Let function $first(pair)$ return the first component in a pair. We define the deletion algorithm as below.

$$delete(T, k) = first(del(T, k)) \tag{C.17}$$

where

$$del(T, k) = \begin{cases} (\phi, 0) & : & T = \phi \\ tree(del(T_l, k), k', (T_r, 0), \Delta) & : & k < k' \\ tree((T_l, 0), k', del(T_r, k), \Delta) & : & k > k' \\ (T_r, -1) & : & k = k', T_l = \phi \\ (T_l, -1) & : & k = k', T_r = \phi \\ tree((T_l, 0), k'', del(T_r, k''), \Delta) & : & otherwise, k'' = min(T_r) \end{cases} \tag{C.18}$$

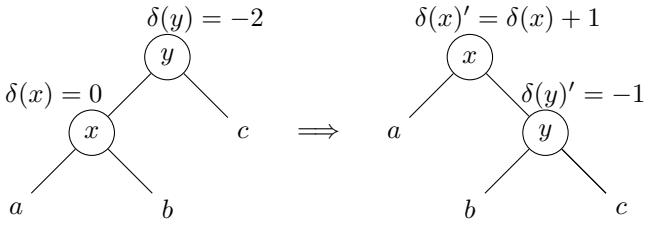
If the tree is empty, the result is empty as well; Otherwise, we compare the key with the value to be delete and recursively lookup and perform delete along the sub trees. If the node to be delete has only one child, we cut that node off. Otherwise, we use the minimum value in the right sub tree to replace the key, and cut the minimum node off.

We can reuse the $tree()$ function, as well as the result for ΔH. But different from insertion, there are extra two cases can violate AVL properties, thus need fixing.

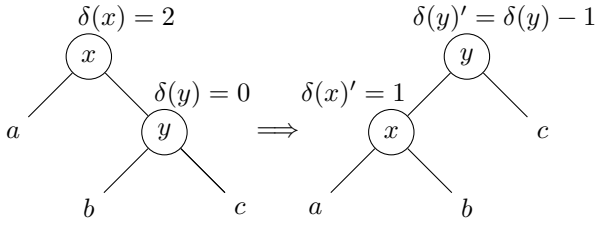
As shown in figure C.2, both cases can be fixed by a tree rotation. It can be essentailly catpured by pattern matching to the tree structure.

$$balance(T, \Delta H) = \begin{cases} ... & : & \\ (A, x, (B, y, C, -1), \delta(x) + 1, \Delta H) & : & T = ((A, x, B\delta(x)), y, C, -2, \Delta H) \\ ((A, x, B1), y, C, \delta(y) - 1, \Delta H) & : & T = (A, x, (B, y, C\delta(y)), 2, \Delta H) \\ ... & : & \end{cases} \tag{C.19}$$

Below is the example program in Haskell



(a) Fix AVL violation after deletion. case A



(b) Fix AVL violation after deletion. case B

Figure C.2: AVL tree fixing for delete

```
delete::(Ord a) ⇒ AVLTree a → a → AVLTree a
delete t x = fst $ del t x where
  -- result is a pair (t, d), t: tree, d: decrement in height
  del Empty _ = (Empty, 0)
  del (Br l k r d) x
    | x < k = node (del l x) k (r, 0) d
    | x > k = node (l, 0) k (del r x) d
    -- x == k, delete this node
    | isEmpty l = (r, -1)
    | isEmpty r = (l, -1)
    | otherwise = node (l, 0) k' (del r k') d where k' = min r
```

Where the auxiliary function `min` are implemented similiar to the binary search tree, it traverses along the left sub tree till the end. Function `isEmpty` test if a tree is empty.

```
isEmpty Empty = True
isEmpty _ = False

min :: AVLTree a → a
min (Br Empty x _ _) = x
min (Br l _ _ _) = min l
```

After adding the extra two cases for fixing, there are total 7 cases in the `balance` function.

```
balance :: (AVLTree a, Int) → (AVLTree a, Int)
balance (Br (Br (Br a x b dx) y c (-1)) z d (-2), dH) =
        (Br (Br a x b dx) y (Br c z d 0) 0, dH-1)
balance (Br a x (Br b y (Br c z d dz)    1)    2, dH) =
        (Br (Br a x b 0) y (Br c z d dz) 0, dH-1)
balance (Br (Br a x (Br b y c dy)    1) z d (-2), dH) =
        (Br (Br a x b dx') y (Br c z d dz') 0, dH-1) where
```

www.dbooks.org

```
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
balance (Br a x (Br (Br b y c dy) z d (-1))    2, dH) =
        (Br (Br a x b dx') y (Br c z d dz') 0, dH-1) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
-- Delete specific fixing
balance (Br (Br a x b dx) y c (-2), dH) = (Br a x (Br b y c (-1)) (dx+1), dH)
balance (Br a x (Br b y c dy)    2, dH) = (Br (Br a x b    1) y c (dy-1), dH)
balance (t, d) = (t, d)
```

C.3.2 Imperative deletion

Imperative deletion algorithm uses tree rotation to resume the balance of the tree. There are more cases need to handle. We firstly apply the binary search tree deletion process, then fix the balance if it violates the AVL properties due to the sub tree height shrink. The main deletion algorithm is given as the following.

```
1: function DELETE(T, x)
2:     if x = NIL then
3:         return T
4:     p ← PARENT(x)
5:     if LEFT(x) = NIL then
6:         y ← RIGHT(x)
7:         replace x with y
8:     else if RIGHT(x) = NIL then
9:         y ← LEFT(x)
10:        replace x with y
11:    else
12:        z ← MIN(RIGHT(x))
13:        copy key and satellite date from z to x
14:        p ← PARENT(z)
15:        y ← RIGHT(z)
16:        replace z with y
17:    return AVL-DELETE-FIX(T, p, y)
```

If the node subject to delete is empty, the tree doesn't change. In normal case, we record the parent of that node as p. If either of sub tree is empty, we cut off the node, and replace it with the other sub tree. Otherwise, we locate the node contains the minimum key from the right sub tree as z, copy the key and the satellite data to x, then cut z off. Finally, we call the fixing function and pass the root, the parent, and the node replaces the removed one as parameters.

Denote the balance factor of the parent node as $\delta(p)$, by checking the new value $\delta(p)'$, there are three different cases.

- Case 1, $|\delta(p)| = 0$, $|\delta(p)'| = 1$. It means, after deletion, although one sub tree height shrinks, the parent node still satisfies the AVL property. The algorithm terminates as the tree is still balanced;
- Case 2, $|\delta(p)| = 1$, $|\delta(p)'| = 0$. It means before the deletion, the height difference between the two sub trees are 1; while after deletion, the sub

tree with greater height shrinks by 1, thus the heights are same now. As the result, the height of the parent tree also decrease by 1. We need go on the bottom-up updating along the parent link towards the root;

- Case 3, $|\delta(p)| = 1$, $|\delta(p)'| = 2$. It means the tree after deletion violates the AVL property, we need fix it by tree rotation.

For case 3, most of fixing operations are similar to to insertion fixing. However, we need add two extra sub cases to cover the situations illustrated in figure C.2. The fixing algorithm is given as the following.

```
1:  function AVL-Delete-Fix(T, p, x)
2:      while p ≠ NIL do
3:          l ← Left(p), r ← Right(p)
4:          δ ← δ(p), δ' ← δ
5:          if x = l then
6:              δ' ← δ' + 1
7:          else
8:              δ' ← δ' − 1
9:          if p is leaf then                                  ▷ l = r = NIL
10:             δ' ← 0
11:         if |δ| = 1 ∧ |δ'| = 0 then
12:             x ← p
13:             p ← Parent(x)
14:         else if |δ| = 0 ∧ |δ'| = 1 then
15:             return T
16:         else if |δ| = 1 ∧ |δ'| = 2 then
17:             if δ' = 2 then
18:                 if δ(r) = 1 then                           ▷ Right-right case
19:                     δ(p) ← 0
20:                     δ(r) ← 0
21:                     p ← r
22:                     T ← Left-Rotate(T, p)
23:                 else if δ(r) = −1 then                     ▷ Right-left case
24:                     δ_y ← δ( Left(r) )
25:                     if δ_y = 1 then
26:                         δ(p) ← −1
27:                     else
28:                         δ(p) ← 0
29:                     δ( Left(r) ) ← 0
30:                     if δ_y = −1 then
31:                         δ(r) ← 1
32:                     else
33:                         δ(r) ← 0
34:                 else                                       ▷ Delete specific right-right case
35:                     δ(p) ← 1
36:                     δ(r) ← δ(r) − 1
37:                     T ← Left-Rotate(T, p)
38:                     break                                  ▷ No furthur height change
39:             else if δ' = −2 then
40:                 if δ(l) = −1 then                          ▷ Left-left case
```

www.dbooks.org

41: $\delta(p) \leftarrow 0$
42: $\delta(l) \leftarrow 0$
43: $p \leftarrow l$
44: $T \leftarrow$ RIGHT-ROTATE(T, p)
45: **else if** $\delta(l) = 1$ **then** ▷ Left-right case
46: $\delta_y \leftarrow \delta($ RIGHT(l) $)$
47: **if** $\delta_y = -1$ **then**
48: $\delta(p) \leftarrow 1$
49: **else**
50: $\delta(p) \leftarrow 0$
51: $\delta($ RIGHT(l) $) \leftarrow 0$
52: **if** $\delta_y = 1$ **then**
53: $\delta(l) \leftarrow -1$
54: **else**
55: $\delta(l) \leftarrow 0$
56: **else** ▷ Delete specific left-left case
57: $\delta(p) \leftarrow -1$
58: $\delta(l) \leftarrow \delta(l) + 1$
59: $T \leftarrow$ RIGHT-ROTATE(T, p)
60: break ▷ No furthur height change
▷ Height decrease, go on bottom-up updating
61: $x \leftarrow p$
62: $p \leftarrow$ PARENT(x)
63: **if** $p =$ NIL **then** ▷ Delete the root
64: **return** x
65: **return** T

The follow C++ example program implements this AVL tree delete algorithm.

```
Node* del(Node* t, Node* x) {
    if (!x) return t;
    Node *y, *parent = x→parent;
    if (!x→left) {
        y = x→replaceWith(x→right);
    } else if (!x→right) {
        y = x→replaceWith(x→left);
    } else {
        y = min(x→right);
        x→key = y→key;
        parent = y→parent;
        x = y;
        y = y→replaceWith(y→right);
    }
    t = deleteFix(t, parent, y);
    remove(x);
    return t;
}
```

Where method `replaceWith(tree)` replaces the current node with the parameter passed in, and returns the new node as the result.

```
Node* replaceWith(Node* y) {
```

```
    return replace(parent, this, y);
}
```

And the function `replace(parent, x, y)` replaces child `x` with `y` to its parent.

```
// change from: parent -> x to parent -> y
Node* replace(Node* parent, Node* x, Node* y) {
    if (!parent) {
        if (y) y->parent = nullptr;
    } else if (parent->left == x) {
        parent->setLeft(y);
    } else {
        parent->setRight(y);
    }
    if (x) x->parent = nullptr;
    return y;
}
```

Function `min(t)` recursively find the minimum node for a given tree. Function `remove(tree)` helps to release the memory for a given node.

```
Node* min(Node* t) {
    while (t && t->left) t = t->left;
    return t;
}

void remove(Node* x) {
    if (x) {
        x->parent = x->left = x->right = nullptr;
        delete x;
    }
}
```

The fixing method is implemented as below.

```
Node* deleteFix(Node* t, Node* parent, Node* x) {
    int d1, d2, dy;
    Node *p, *l, *r;
    while (parent) {
        d2 = d1 = parent->delta;
        d2 += (x == parent->left ? 1 : -1);
        if (isLeaf(parent)) d2 = 0;
        parent->delta = d2;
        p = parent;
        l = parent->left;
        r = parent->right;
        if (abs(d1) == 1 && abs(d2) == 0) {
            x = parent;
            parent = x->parent;
        } else if (abs(d1) == 0 && abs(d2) == 1) {
            return t;
        } else if (abs(d1) == 1 && abs(d2) == 2) {
            if (d2 == 2) {
                if (r->delta == 1) {  // right-right case
                    p->delta = r->delta = 0;
                    parent = r;
```

www.dbooks.org

```
                t = leftRotate(t, p);
            } else if (r→delta == -1) { // right-left case
                dy = r→left→delta;
                p→delta = dy == 1 ? -1 : 0;
                r→left→delta = 0;
                r→delta = dy == -1 ? 1 : 0;
                parent = r→left;
                t = rightRotate(t, r);
                t = leftRotate(t, p);
            } else { // delete specific right-right case
                p→delta = 1;
                r→delta--;
                t = leftRotate(t, p);
                break; // no further height change
            }
        } else if (d2 == -2) {
            if (l→delta == -1) { // left-left case
                p→delta = l→delta = 0;
                parent = l;
                t = rightRotate(t, p);
            } else if (l→delta == 1) { // left-right case
                dy = l→right→delta;
                l→delta = dy == 1 ? -1 : 0;
                l→right→delta = 0;
                p→delta = dy == -1 ? 1 : 0;
                parent = l→right;
                t = leftRotate(t, l);
                t = rightRotate(t, p);
            } else { // delete specific left-left case
                p→delta = -1;
                l→delta++;
                t = rightRotate(t, p);
                break; // no further height change
            }
        }
        // the 4 rebalance cases cause height decrease, go on bottom-up update
        x = parent;
        parent = x→parent;
    } else {
        printf("shouldn't be here, d1 = %d, d2 = %d", d1, d2);
        assert(false);
    }
}
if (!parent) return x; // delete the root
return t;
}
```

Exercise C.1

Compare the imperative AVL tree fixing algorithms for insertion and deletion, there are similarities. Develop a common AVL tree fixing algorith can serve for both insertion and deletion.

Appendix D

Suffix Tree

D.1 Introduction

Suffix Tree is an important data structure. It can be used to realize many important string operations particularly fast[3]. It is also widely used in bio-information area such as DNA pattern matching[4]. Weiner introduced suffix tree in 1973[2]. The latest on-line construction algorithm was found in 1995[1].

The suffix tree for a string S is a special Patricia. Each edge is labeled with some sub-string of S. Each suffix of S corresponds to exactly one path from the root to a leaf. Figure D.1 shows the suffix tree for English word 'banana'.

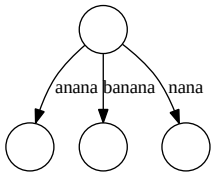


Figure D.1: The suffix tree for 'banana'

All suffixes, 'banana', 'anana', 'nana', 'ana', 'na', 'a', '' can be found in the above tree. Among them the first 3 suffixes are explicitly shown; others are implicitly represented. The reason for why 'ana, 'na', 'a', and '' are not shown is because they are prefixes of the others. In order to show all suffixes explicitly, we can append a special pad terminal symbol, which doesn't occur in other places in the string. Such terminator is typically denoted as '$'. By this means, there is no suffix being the prefix of the others.

Although the suffix tree for 'banana' is simple, the suffix tree for 'bananas', as shown in figure D.2, is quite different.

To create suffix suffix tree for a given string, we can utilize the insertion algorithm explained in previous chapter for Patricia.

```
1: function SUFFIX-TREE(S)
2:     T ← NIL
3:     for i ← 1 to |S| do
4:         T ← PATRICIA-INSERT(T, RIGHT(S, i))
5:     return T
```


www.dbooks.org

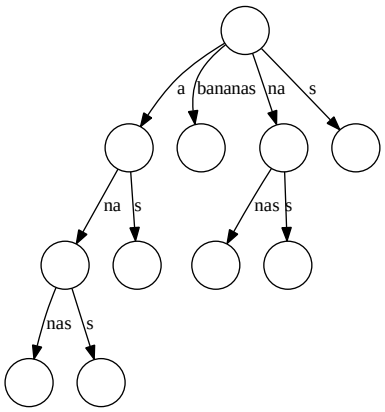


Figure D.2: The suffix tree for 'bananas'

For non-empty string $S = s_1 s_2 ... s_i ... s_n$ of length $n = |S|$, function RIGHT(S, i) $= s_i s_{i+1} ... s_n$. It extracts the sub-string of S from the i-th character to the last one. This straightforward algorithm can also be defined as below.

$$suffix_T(S) = fold(insert_{Patricia}, \phi, suffixes(S)) \tag{D.1}$$

Where function $suffixes(S)$ gives all the suffixes for string S. If the string is empty, the result is one empty string; otherwise, S itself is one suffix, the others can be given by recursively call $suffixes(S')$, where S' is given by drop the first character from S.

$$suffixes(S) = \begin{cases} \{\phi\} & : \quad S = \phi \\ \{S\} \cup suffixes(S') & : \quad otherwise \end{cases} \tag{D.2}$$

This solution constructs suffix tree in $O(n^2)$ time, for string of length n. It totally inserts n suffixes to the tree, and each insertion takes linear time proportion to the length of the suffix. The efficiency isn't good enough.

In this chapter, we firstly explain a fast on-line suffix trie construction solution by using suffix link concept. Because trie isn't space efficient, we next introduce a linear time on-line suffix tree construction algorithm found by Ukkonen. and show how to solve some interesting string manipulation problems with suffix tree.

D.2 Suffix trie

Just likes the relationship between trie and Patricia, Suffix trie has much simpler structure than suffix tree. Figure D.3 shows the suffix trie for 'banana'.

Compare with figure D.1, we can find the difference between suffix tree and suffix trie. Instead of representing a word, every edge in suffix trie only represents a character. Thus suffix trie needs much more spaces. If we pack all nodes which have only one child, the suffix trie is turned into a suffix tree.

We can reuse the trie definition for suffix trie. Each node is bound to a character, and contains multiple sub trees as children. A child can be referred from the bounded character.

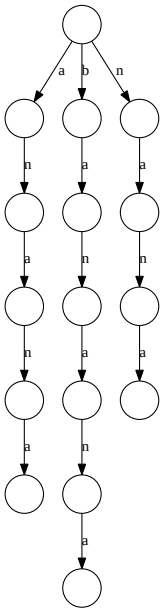


Figure D.3: Suffix trie for 'banana'

D.2.1 Node transfer and suffix link

For string S of length n, define $S_i = s_1 s_2 ... s_i$. It is the prefix contains the first i characters.

In suffix trie, each node represents a suffix string. for example in figure D.4, node X represents suffix 'a', by adding character 'c', node X transfers to Y which represents suffix 'ac'. We say node X transfer to Y with the edge of character 'c'[1].

$Y \leftarrow \textsc{Children}(X)[c]$

We also say that node X has a 'c'-child Y. Below Python expression reflects this concept.

```
y = x.children[c]
```

If node A in a suffix trie represents suffix $s_i s_{i+1} ... s_n$, and node B represents suffix $s_{i+1} s_{i+2} ... s_n$, we say node B represents *the suffix* of node A. We can create a link from A to B. This link is defined as *the suffix link* of node A[1]. Suffix link is drawn in dotted style. In figure D.4, the suffix link of node A points to node B, and the suffix link of node B points to node C.

Suffix link is valid for all nodes except the root. We can add a suffix link field to the trie definition. Below Python example code shows this update.

```
class STrie:
    def __init__(self, suffix=None):
        self.children = {}
        self.suffix = suffix
```

www.dbooks.org

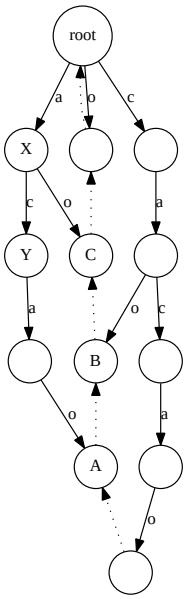


Figure D.4: Suffix trie for string "cacao". Node $X \leftarrow$ "a", node $Y \leftarrow$ "ac", X transfers to Y with character 'c'

suffix string
$s_1 s_2 s_3 ... s_i$
$s_2 s_3 ... s_i$
...
$s_{i-1} s_i$
s_i
""

Table D.1: suffixes for S_i

D.2.2 On-line construction

For string S, Suppose we have constructed suffix trie for the i-th prefix $S_i = s_1 s_2 ... s_i$. Denote it as $SuffixTrie(S_i)$. Let's consider how to obtain $SuffixTrie(S_{i+1})$ from $SuffixTrie(S_i)$.

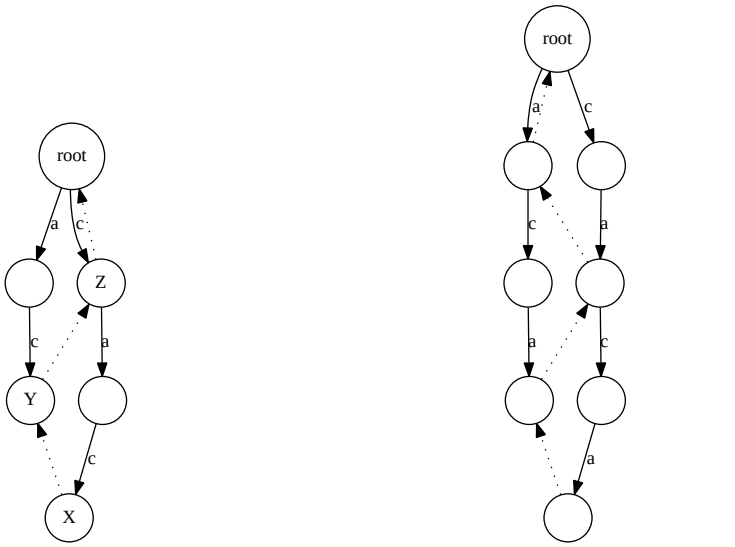
If list all suffixes corresponding to $SuffixTrie(S_i)$, from the longest (which is S_i) to the shortest (which is empty), we can get table D.1. There are total $i + 1$ suffixes.

One solution is to append the character s_{i+1} to every suffix in this table, then add another empty string. This idea can be realized by adding a new child for every node in the trie, and binding all these new child with edge of character s_{i+1}.

Algorithm 5 Update $SuffixTrie(S_i)$ to $SuffixTrie(S_{i+1})$, initial version.

1: **for** $\forall T \in SuffixTrie(S_i)$ **do**
2: CHILDREN$(T)[s_{i+1}] \leftarrow$ CREATE-EMPTY-NODE

However, some nodes in $SuffixTrie(S_i)$ may have the s_{i+1}-child already. For example, in figure D.5, node X and Y are corresponding for suffix 'cac' and 'ac' respectively. They don't have the 'a'-child. But node Z, which represents suffix 'c' has the 'a'-child already.



(a) Suffix trie for string "cac". (b) Suffix trie for string "caca".

Figure D.5: Suffix Trie of "cac" and "caca"

When append s_{i+1} to $SuffixTrie(S_i)$. In this example s_{i+1} is character 'a'. We need create new nodes for X and Y, but we needn't do this for Z.

If check the nodes one by one according to table D.1, we can stop immediately when meet a node which has the s_{i+1}-child. This is because if node X in $SuffixTrie(S_i)$ has the s_{i+1}-child, according to the definition of suffix link, any suffix nodes X' of X in $SuffixTrie(S_i)$ must also have the s_{i+1}-child. In other words, let $c = s_{i+1}$, if wc is a sub-string of S_i, then every suffix of wc is also a sub-string of S_i [1]. The only exception is the root, which represents for empty string "".

According to this fact, we can refine the algorithm 5 to the following.

Algorithm 6 Update $SuffixTrie(S_i)$ to $SuffixTrie(S_{i+1})$, second version.

```
1: for each T ∈ SuffixTrie(S_i) in descending order of suffix length do
2:     if CHILDREN(T)[s_{i+1}] = NIL then
3:         CHILDREN(T)[s_{i+1}] ← CREATE-EMPTY-NODE
4:     else
5:         break
```

The next question is how to iterate all nodes in descending order of the suffix length? Define the *top* of a suffix trie as the deepest leaf node. This definition ensures the top represents the longest suffix. Along the suffix link from the top to the next node, the length of the suffix decrease by one. This fact tells us that We can traverse the suffix tree from the top to the root by using the suffix

www.dbooks.org

links. And the order of such traversing is exactly what we want. Finally, there is a special suffix trie for empty string $SuffixTrie(NIL)$, We define the top equals to the root in this case.

```
function INSERT(top, c)
    if top = NIL then                                   ▷ The trie is empty
        top ← CREATE-EMPTY-NODE
    T ← top
    T' ← CREATE-EMPTY-NODE                              ▷ dummy init value
    while T ≠ NIL ∧ CHILDREN(T)[c] = NIL do
        CHILDREN(T)[c] ← CREATE-EMPTY-NODE
        SUFFIX-LINK(T') ← CHILDREN(T)[c]
        T' ← CHILDREN(T)[c]
        T ← SUFFIX-LINK(T)
    if T ≠ NIL then
        SUFFIX-LINK(T') ← CHILDREN(T)[c]
    return CHILDREN(top)[c]                             ▷ returns the new top
```

Function INSERT, updates $SuffixTrie(S_i)$ to $SuffixTrie(S_{i+1})$. It takes two arguments, one is the top of $SuffixTrie(S_i)$, the other is s_{i+1} character. If the top is NIL, it means the tree is empty, so there is no root. The algorithm creates a root node in this case. A sentinel empty node T' is created. It keeps tracking the previous created new node. In the main loop, the algorithm checks every node one by one along the suffix link. If the node hasn't the s_{i+1}-child, it then creates a new node, and binds the edge to character s_{i+1}. The algorithm repeatedly goes up along the suffix link until either arrives at the root, or find a node which has the s_{i+1}-child already. After the loop, if the node isn't empty, it means we stop at a node which has the s_{i+1}-child. The last suffix link then points to that child. Finally, the new top position is returned, so that we can further insert other characters to the suffix trie.

For a given string S, the suffix trie can be built by repeatedly calling INSERT function.

```
1: function SUFFIX-TRIE(S)
2:     t ← NIL
3:     for i ← 1 to |S| do
4:         t ← INSERT(t, s_i)
5:     return t
```

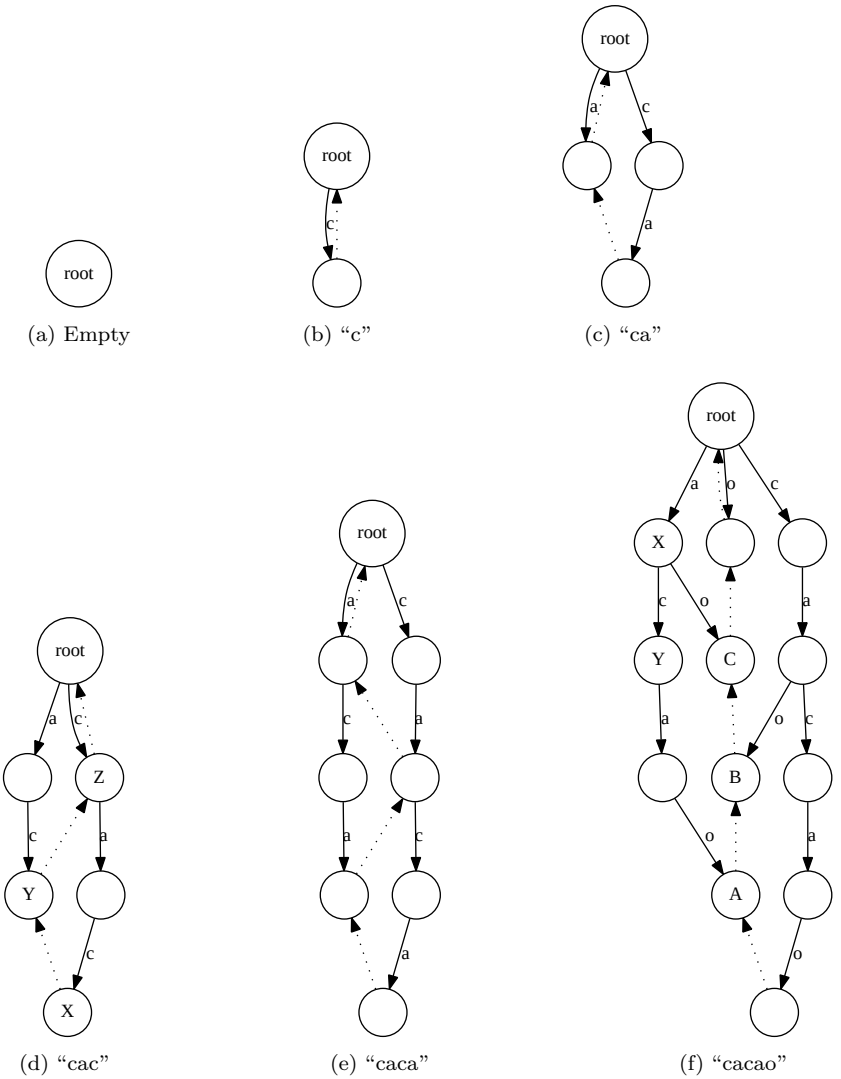
This algorithm returns the top of the suffix trie, but not the root. In order to access the root, we can traverse along the suffix link.

```
1: function ROOT(T)
2:     while SUFFIX-LINK(T) ≠ NIL do
3:         T ← SUFFIX-LINK(T)
4:     return T
```

Figure D.6 shows the steps when construct suffix trie for "cacao". Only the last layer of suffix links are shown.

For INSERT algorithm, the computation time is proportion to the size of suffix trie. In the worse case, the suffix trie is built in $O(n^2)$ time, where $n = |S|$. One example is $S = a^n b^n$, that there are n characters of a and n characters of b.

The following example Python program implements the suffix trie construc-



(a) Empty (b) “c” (c) “ca”

(d) “cac” (e) “caca” (f) “cacao”

Figure D.6: Construct suffix trie for “cacao”. There are 6 steps. Only the last layer of suffix links are shown in dotted arrow.

www.dbooks.org

tion algorithm.

```
def suffix_trie(str):
    t = None
    for c in str:
        t = insert(t, c)
    return root(t)

def insert(top, c):
    if top is None:
        top=STrie()
    node = top
    new_node = STrie() #dummy init value
    while (node is not None) and (c not in node.children):
        new_node.suffix = node.children[c] = STrie(node)
        new_node = node.children[c]
        node = node.suffix
    if node is not None:
        new_node.suffix = node.children[c]
    return top.children[c] #update top

def root(node):
    while node.suffix is not None:
        node = node.suffix
    return node
```

D.3 Suffix Tree

Suffix trie isn't space efficient, and the construction time is quadratic. If don't care about the speed, we can compress the suffix trie to suffix tree[6]. Ukkonen found a linear time on-line suffix tree construction algorithm in 1995.

D.3.1 On-line construction

Active point and end point

The suffix trie construction algorithm shows very important fact about what happens when $SuffixTrie(S_i)$ updates to $SuffixTrie(S_{i+1})$. Let's review the last two steps in figure D.6.

There are two different updates.

1. All leaves are appended with a new node for s_{i+1};
2. Some non-leaf nodes are branched out with a new node for s_{i+1}.

The first type of update is trivial, because for all new coming characters, we need do this work anyway. Ukkonen defines leaf as the 'open' node.

The second type of update is important. We need figure out which internal nodes need branch out. We only focus on these nodes and apply the update.

Ukkonen defines the path along the suffix links from the top to the end as 'boundary path'. Denote the nodes in boundary path as, $n_1, n_2, ..., n_j, ..., n_k$. These nodes start from the leaf node (the first one is the top position), suppose

that after the j-th node, they are not leaves any longer, we need repeatedly branch out from this time point till the k-th node.

Ukkonen defines the first none-leaf node n_j as 'active point' and the last node n_k as 'end point'. The end point can be the root.

Reference pair

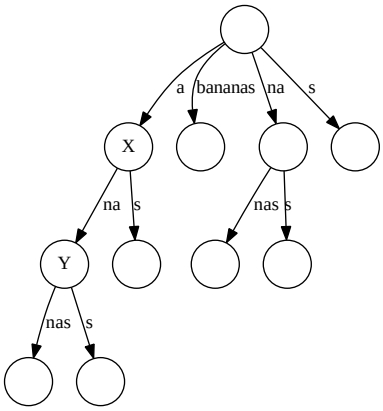


Figure D.7: Suffix tree of "bananas". X transfer to Y with sub-string "na".

Figure D.7 shows the suffix tree of English word "bananas". Node X represents suffix "a". By adding sub-string "na", node X transfers to node Y, which represents suffix "ana". In other words, we can represent Y with a pair of node and sub-string, like (X, w), where $w =$ "na". Ukkonen defines such kind of pair as *reference pair*. Not only the explicit node, but also the implicit position in suffix tree can be represented with reference pair. For example, (X, "n") represents to a position which is not an explicit node. By using reference pair, we can represent every position in a suffix tree.

In order to save spaces, for string S, all sub-strings can be represented as a pair of index (l, r), where l is the left index and r is the right index of the character for the sub-string. For instance, if $S =$ "bananas", and the index starts from 1, sub-string "na" can be represented with pair $(3, 4)$. As the result, there is only one copy of the complete string, and all positions in the suffix tree is defined as $(node, (l, r))$. This is the final form of reference pair.

With reference pair, node transfer for suffix tree can be defined as the following.

$\textsc{Children}(X)[s_l] \leftarrow ((l, r), Y) \iff Y \leftarrow (X, (l, r))$

If character $s_l = c$, we say that node X has a c-child, This child is Y. Y can be transferred from X with sub string (l, r) Each node can have at most one c-child.

canonical reference pair

It's obvious that the one position in a suffix tree may have multiple reference pairs. For example, node Y in Figure D.7 can be either denoted as $(X, (3, 4))$ or $(root, (2, 4))$. If we define empty string $\epsilon = (i, i - 1)$, Y can also be represented as (Y, ϵ).

www.dbooks.org

The canonical reference pair is the one which has the closest node to the position. Specially, in case the position is an explicit node, the canonical reference pair is $(node, \epsilon)$, so (Y, ϵ) is the canonical reference pair of node Y.

Below algorithm converts a reference pair $(node, (l, r))$ to the canonical reference pair $(node', (l', r))$. Note that since r doesn't change, the algorithm can only return $(node', l')$ as the result.

Algorithm 7 Convert reference pair to canonical reference pair

```
1:  function CANONIZE(node, (l, r))
2:      if node = NIL then
3:          if (l, r) = ϵ then
4:              return ( NIL, l)
5:          else
6:              return CANONIZE(root, (l + 1, r))
7:      while l ≤ r do                                  ▷ (l, r) isn't empty
8:          ((l', r'), node') ← CHILDREN(node)[s_l]
9:          if r − l ≥ r' − l' then
10:             l ← l + r' − l' + 1                 ▷ Remove |(l', r')| chars from (l, r)
11:             node ← node'
12:         else
13:             break
14:     return (node, l)
```

If the passed in node parameter is NIL, it means a very special case. The function is called like the following.

CANONIZE(SUFFIX-LINK($root$), (l, r))

Because the suffix link of root points to NIL, the result should be $(root, (l + 1, r))$ if (l, r) is not ϵ. Otherwise, (NIL, ϵ) is returned to indicate a terminal position.

We explain this special case in detail in later sections.

The algorithm

In D.3.1, we mentioned, all updating to leaves is trivial, because we only need append the new coming character to the leaf. With reference pair, it means, when update $SuffixTree(S_i)$ to $SuffixTree(S_{i+1})$, all reference pairs in form $(node, (l, i))$, are leaves. They will change to $(node, (l, i+1))$ next time. Ukkonen defines leaf in form $(node, (l, \infty))$, here ∞ means "open to grow". We can skip all leaves until the suffix tree is completely constructed. After that, we can change all ∞ to the length of the string.

So the main algorithm only cares about *positions* from the active point to the end point. However, how to find the active point and the end point?

When start suffix tree construction, there is only a root node. There aren't any branches or leaves. The active point should be $(root, \epsilon)$, or $(root, (1, 0))$ (the string index starts from 1).

For the end point, it is a position where we can finish updating $SuffixTree(S_i)$. According to the suffix trie algorithm, we know it should be a *position* which has the s_{i+1}-child already. Because a position in suffix trie may not be an explicit node in suffix tree, if $(node, (l, r))$ is the end point, there are two cases.

1. $(l, r) = \epsilon$. It means the node itself is the end point. This node has the s_{i+1}-child, which means CHILDREN$(node)[s_{i+1}] \neq$ NIL;

2. Otherwise, $l \leq r$, the end point is an implicit position. It must satisfy $s_{i+1} = s_{l'+|(l,r)|}$, where CHILDREN$(node)[s_l] = ((l', r'), node')$, $|(l, r)|$ means the length of sub-string (l, r). It equals to $r - l + 1$. This is illustrated in figure D.8. We can also say that $(node, (l, r))$ has a s_{i+1}-child implicitly.

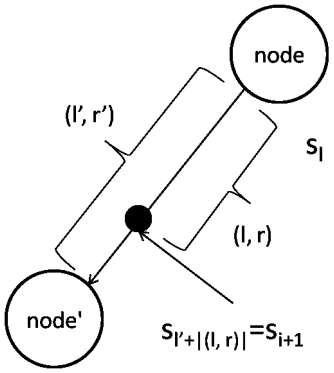


Figure D.8: Implicit end point

Ukkonen finds a very important fact that if $(node, (l, i))$ is the end point of $SuffixTree(S_i)$, then $(node, (l, i+1))$ is the active point of $SuffixTree(S_{i+1})$.

This is because if $(node, (l, i))$ is the end point of $SuffixTree(S_i)$, it must have a s_{i+1}-child (either explicitly or implicitly). If this end point represents suffix $s_k s_{k+1} ... s_i$, it is the longest suffix in $SuffixTree(S_i)$ which satisfies $s_k s_{k+1} ... s_i s_{i+1}$ is a sub-string of S_i. Consider S_{i+1}, $s_k s_{k+1} ... s_i s_{i+1}$ must occur at least twice in S_{i+1}, so position $(node, (l, i + 1))$ is the active point of $SuffixTree(S_{i+1})$. Figure D.9 shows about this truth.

Summarize the above facts, the algorithm of Ukkonen's on-line construction can be given as the following.

```
1:  function UPDATE(node, (l, i))
2:      prev ← CREATE-EMPTY-NODE                    ▷ Initialized as sentinel
3:      loop                                        ▷ Traverse along the suffix links
4:          (finish, node') ← END-POINT-BRANCH?(node, (l, i − 1), s_i)
5:          if finish then
6:              break
7:          CHILDREN(node')[s_i] ← ((i, ∞), CREATE-EMPTY-NODE)
8:          SUFFIX-LINK(prev) ← node'
9:          prev ← node'
10:         (node, l) ← CANONIZE(SUFFIX-LINK(node), (l, i − 1))
11:     SUFFIX-LINK(prev) ← node
12:     return (node, l)                            ▷ The end point
```

This algorithm takes reference pair $(node, (l, i))$ as arguments, note that position $(node, (l, i - 1)$ is the active point for $SuffixTree(S_{i-1})$. Then we start a loop, this loop goes along the suffix links until the current position

www.dbooks.org

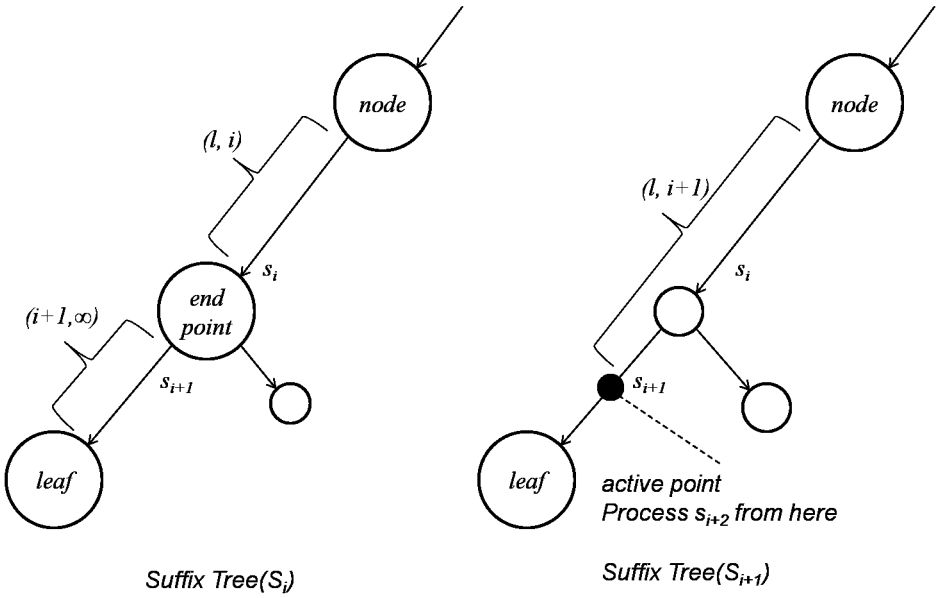


Figure D.9: End point in $SuffixTree(S_i)$ and active point in $SuffixTree(S_{i+1})$.

$(node, (l, i-1))$ is the end point. Otherwise, function END-POINT-BRANCH? returns a position, from where the new leaf branch out.

The END-POINT-BRANCH? algorithm is realized as below.

```
function END-POINT-BRANCH?(node, (l, r), c)
    if (l, r) = ε then
        if node = NIL then
            return (TRUE, root)
        else
            return (CHILDREN(node)[c] = NIL, node)
    else
        ((l', r'), node') ← CHILDREN(node)[s_l]
        pos ← l' + |(l, r)|
        if s_pos = c then
            return (TRUE, node)
        else
            p ← CREATE-EMPTY-NODE
            CHILDREN(node)[s_l'] ← ((l', pos − 1), p)
            CHILDREN(p)[s_pos] ← ((pos, r'), node')
            return (FALSE, p)
```

If the position is $(root, \epsilon)$, it means we have arrived at the root. It's definitely the end point, so that we can finish this round of updating. If the position is in form of $(node, \epsilon)$, it means the reference pair represents an explicit node, we can examine if this node has already the c-child, where $c = s_i$. If not, we need branch out a leaf.

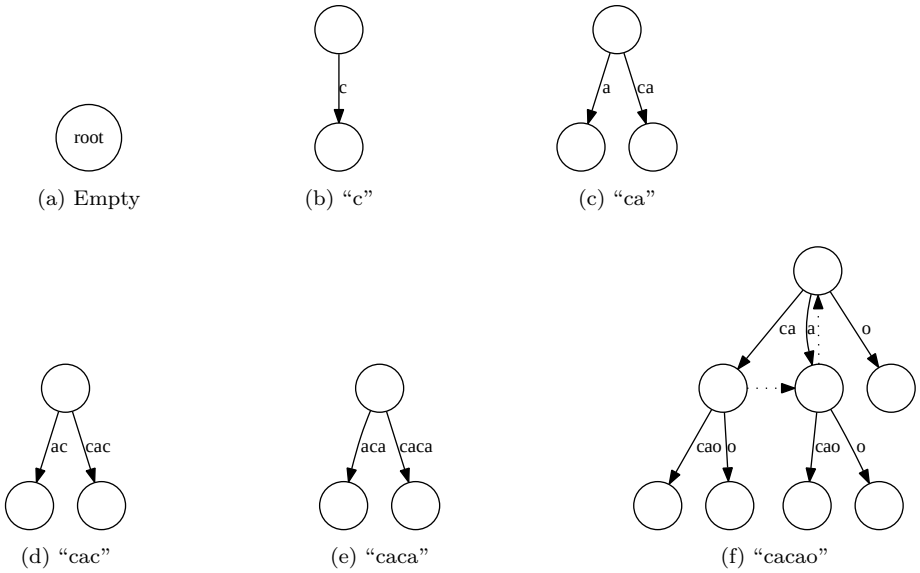
Otherwise, the position $(node, (l, r))$ points to an implicit node. We need find the exact position next to it to see if there is a c-child. If yes, we meet an

end point, the updating loop can be finished; else, we turn the position to an explicit node, and return it for further branching.

We can finalize the Ukkonen's algorithm as below.

1: **function** SUFFIX-TREE(S)
2: $root \leftarrow$ CREATE-EMPTY-NODE
3: $node \leftarrow root, l \leftarrow 0$
4: **for** $i \leftarrow 1$ to $|S|$ **do**
5: $(node, l) \leftarrow$ UPDATE$(node, (l, i))$
6: $(node, l) \leftarrow$ CANONIZE$(node, (l, i))$
7: **return** $root$

Figure D.10 shows the steps when constructing the suffix tree for string "cacao".



(a) Empty (b) "c" (c) "ca"

(d) "cac" (e) "caca" (f) "cacao"

Figure D.10: Construct suffix tree for "cacao". There are 6 steps. Only the last layer of suffix links are shown in dotted arrow.

Note that we needn't set suffix link for leaf nodes, only branch nodes need suffix links.

The following example Python program implements Ukkonen's algorithm. First is the node definition.

```
class Node:
    def __init__(self, suffix=None):
        self.children = {} # 'c':(word, Node), where word = (l, r)
        self.suffix = suffix
```

Because there is only one copy of the complete string, all sub-strings are represent in $(left, right)$ pairs, and the leaf are open pairs as $(left, \infty)$. The suffix tree is defined like below.

```
class STree:
    def __init__(self, s):
```

www.dbooks.org

```
        self.str = s
        self.infinity = len(s)+1000
        self.root = Node()
```

The infinity is defined as the length of the string plus a big number. Some auxiliary functions are defined.

```
def substr(str, str_ref):
    (l, r)=str_ref
    return str[l:r+1]

def length(str_ref):
    (l, r)=str_ref
    return r-l+1
```

The main entry for Ukkonen's algorithm is implemented as the following.

```
def suffix_tree(str):
    t = STree(str)
    node = t.root # init active point is (root, Empty)
    l = 0
    for i in range(len(str)):
        (node, l) = update(t, node, (l, i))
        (node, l) = canonize(t, node, (l, i))
    return t

def update(t, node, str_ref):
    (l, i) = str_ref
    c = t.str[i] # current char
    prev = Node() # dummy init
    while True:
        (finish, p) = branch(t, node, (l, i-1), c)
        if finish:
            break
        p.children[c]=((i, t.infinity), Node())
        prev.suffix = p
        prev = p
        (node, l) = canonize(t, node.suffix, (l, i-1))
    prev.suffix = node
    return (node, l)

def branch(t, node, str_ref, c):
    (l, r) = str_ref
    if length(str_ref)≤0: # (node, empty)
        if node is None: #_|_
            return (True, t.root)
        else:
            return ((c in node.children), node)
    else:
        ((l1, r1), node1) = node.children[t.str[l]]
        pos = l1+length(str_ref)
        if t.str[pos]==c:
            return (True, node)
        else:
            branch_node = Node()
            node.children[t.str[l1]]=((l1, pos-1), branch_node)
```

```
            branch_node.children[t.str[pos]] = ((pos, r1), node1)
            return (False, branch_node)

def canonize(t, node, str_ref):
    (l, r) = str_ref
    if node is None:
        if length(str_ref)≤0:
            return (None, l)
        else:
            return canonize(t, t.root, (l+1, r))
    while l≤r: # str_ref is not empty
        ((l1, r1), child) = node.children[t.str[l]]
        if r-l ≥ r1-l1:
            l += r1-l1+1
            node = child
        else:
            break
    return (node, l)
```

Functional suffix tree construction

Giegerich and Kurtz found Ukkonen's algorithm can be transformed to McCreight's algorithm[7]. The three suffix tree construction algorithms found by Weiner, McCreight, and Ukkonen are all bound to $O(n)$ time. Giegerich and Kurtz conjectured any sequential suffix tree construction method doesn't base on suffix links, active suffixes, etc., fails to meet the $O(n)$-criterion.

There is implementation in PLT/Scheme[10] based on Ukkonen's algorithm, However, it updates suffix links during the processing, which is not purely functional.

A lazy suffix tree construction method is discussed in [8]. And this method is contributed to Haskell Hackage by Bryan O'Sullivan. [9]. The method depends on the lazy evaluation property. The tree won't be constructed until it is traversed. However, it can't ensure the $O(n)$ performance if the programming environments or languages don't support lazy evaluation.

The following Haskell program defines the suffix tree. A suffix tree is either a leaf, or a branch containing multiple sub trees. Each sub tree is bound to a string.

```
data Tr = Lf | Br [(String, Tr)] deriving (Eq)
type EdgeFunc = [String]→(String, [String])
```

The edge function extracts a common prefix from a list of strings. The prefix returned by edge function may not be the longest one, empty string is also allowed. The exact behavior can be customized with different edge functions.

$$build(edge, X)$$

This defines a generic radix tree building function. It takes an edge function, and a set of strings. X can be all suffixes of a string, so that we get suffix trie or suffix tree. We'll also explain later that X can be all prefixes, which lead to normal prefix trie or Patricia.

Suppose all the strings are built from a character set Σ. When build the tree, if the string is empty, X only contains one empty string as well. The result

www.dbooks.org

tree is an empty leaf; Otherwise, we examine every character in Σ, group the strings in X with their initial characters, the apply the edge function to these groups.

$$build(edge, X) = \begin{cases} leaf & : X = \{\phi\} \\ branch(\{(\{c\} \cup p, & build(edge, X'))| \\ & c \in \Sigma, \\ & G \in \{group(X, c)\}, \\ & (p, X') \in \{edge(G)\}\}) \end{cases} \quad : otherwise \tag{D.3}$$

The algorithm categorizes all suffixes by the first letter in several groups. It removes the first letter for each element in every group. For example, the suffixes {“acac”, “cac”, “ac”, “c”} are categorized to groups {(’a’, [“cac”, “c”]), (’c’, [“ac”, “”])}.

$$group(X, c) = \{C' | \{c_1\} \cup C' \in X, c_1 = c\} \tag{D.4}$$

Function *group* enumerates all suffixes in X, for each one, denote the first character as c_1, the rest characters as C'. If c_1 is same as the given character c, then C' is collected.

Below example Haskell program implements the generic radix tree building algorithm.

```
alpha = ['a'..'z']++['A'..'Z']

lazyTree::EdgeFunc → [String] → Tr
lazyTree edge = build where
    build [[]] = Lf
    build ss = Br [(a:prefix, build ss') |
                        a←alpha,
                        xs@(x:_) ←[[cs | c:cs←ss, c==a]],
                        (prefix, ss')←[edge xs]]
```

Different edge functions produce different radix trees. Since edge function extracts common prefix from a set of strings. The simplest one constantly uses the empty string as the common prefix. This edge function builds a trie.

$$edgeTrie(X) = (\phi, X) \tag{D.5}$$

We can also realize an edge function, that extracts the longest common prefix. Such edge function builds a Patricia. Denote the strings as $X = \{x_1, x_2, ..., x_n\}$, for the each string x_i, let the initial character be c_i, and the rest characters in x_i as W_i. If there is only one string in X, the longest common prefix is definitely this string; If there are two strings start with different initial characters, the longest common prefix is empty; Otherwise,it means all the strings share the same initial character. This character definitely belongs to the longest common prefix. We can remove it from all strings, and recursively call the edge function.

$$edgeTree(X) = \begin{cases} (x_1, \{\phi\}) & : X = \{x_1\} \\ (\phi, X) & : |X| > 1, \exists x_i \in X, c_i \neq c_1 \\ (\{c_1\} \cup p, Y) & : (p, Y) = edgeTree(\{W_i | x_i \in X\}) \end{cases} \tag{D.6}$$

Here are some examples for $edgeTree$ function.

$$edgeTree(\{\text{“an”, “another”, “and”}\}) = (\text{“an”}, \{\text{“”, “other”, “d”}\})$$
$$edgeTree(\{\text{“bool”, “foo, “bar”}\}) = (\text{“”}, \{\text{“bool”, “fool”, “bar”}\})$$

The following example Haskell program implements this edge function.

```
edgeTree::EdgeFunc
edgeTree [s] = (s, [[]])
edgeTree awss@((a:w):ss) | null [c|c:_←ss, a/=c] = (a:prefix, ss')
                         | otherwise             = ("", awss)
                         where (prefix, ss') = edgeTree (w:[u| _:u←ss])
edgeTree ss = ("", ss)
```

For any given string, we can build suffix trie and suffix tree by feeding suffixes to these two edge functions.

$$suffixTrie(S) = build(edgeTrie, suffixes(S)) \tag{D.7}$$

$$suffixTree(S) = build(edgeTree, suffixes(S)) \tag{D.8}$$

Because the $build(edge, X)$ is generic, it can be used to build other radix trees, such as the normal prefix trie and Patricia.

$$trie(S) = build(edgeTrie, prefixes(S)) \tag{D.9}$$

$$tree(S) = build(edgeTree, prefixes(S)) \tag{D.10}$$

D.4 Suffix tree applications

Suffix tree can help to solve many string and DNA pattern manipulation problems particularly fast.

D.4.1 String/Pattern searching

There a plenty of string searching algorithms, such as the famous KMP(Knuth-Morris-Pratt algorithm is introduced in the chapter of search) algorithm. Suffix tree can perform at the same level as KMP[11]. the string searching in bound to $O(m)$ time, where m is the length of the sub-string to be search. However, $O(n)$ time is required to build the suffix tree in advance, where n is the length of the text[12].

Not only sub-string searching, but also pattern matching, including regular expression matching can be solved with suffix tree. Ukkonen summarizes this kind of problems as sub-string motifs: *For a string S, $SuffixTree(S)$ gives complete occurrence counts of all sub-string motifs of S in $O(n)$ time, although S may have $O(n^2)$ sub-strings.*

www.dbooks.org

Find the number of sub-string occurrence

Every internal node in $SuffixTree(S)$ is corresponding to a sub-string occurs multiple times in S. If this sub-string occurs k times in S, then there are total k sub-trees under this node[13].

```
1: function LOOKUP-PATTERN(T, s)
2:     loop
3:         match ← FALSE
4:         for ∀(s_i, T_i) ∈ VALUES(CHILDREN(T)) do
5:             if s ⊏ s_i then
6:                 return MAX(|CHILDREN(T_i)|, 1)
7:             else if s_i ⊏ s then
8:                 match ← TRUE
9:                 T ← T_i
10:                s ← s − s_i
11:                break
12:        if ¬match then
13:            return 0
```

When look up a sub-string pattern s in text w, we build the suffix tree T from the text. Start from the root, we iterate all children. For every string reference pair s_i and sub-tree T_i, we check if the s is prefix of s_i. If yes, the number of sub-trees in T_i is returned as the result. There is a special case that T_i is a leaf without any children. We need return 1 but not zero. This is why we use the maximum function. Otherwise, if s_i is prefix of s, then we remove s_i part from s, and recursively look up in T_i.

The following Python program implements this algorithm.

```
def lookup_pattern(t, s):
    node = t.root
    while True:
        match = False
        for _, (str_ref, tr) in node.children.items():
            edge = substr(t, str_ref)
            if string.find(edge, s)==0: #s `isPrefixOf` edge
                return max(len(tr.children), 1)
            elif string.find(s, edge)==0: #edge `isPrefixOf` s
                match = True
                node = tr
                s = s[len(edge):]
                break
        if not match:
            return 0
    return 0 # not found
```

This algorithm can also be realized in recursive way. For the non-leaf suffix tree T, denote the children as $C = \{(s_1, T_1), (s_2, T_2), ...\}$. We search the sub string among the children.

$$lookup_{pattern}(T, s) = find(C, s) \tag{D.11}$$

If children C is empty, it means the sub string doesn't occurs at all. Otherwise, we examine the first pair (s_1, T_1), if s is prefix of s_1, then the number of sub-trees in T_1 is the result. If s_1 is prefix of s, we remove s_1 from s, and

recursively look up it in T_1; otherwise, we go on to examine the rest children denoted as C'.

$$find(C, s) = \begin{cases} 0 & : \quad C = \phi \\ max(1, |C_1|) & : \quad s \sqsubset s_1 \\ lookup_{pattern}(T_1, s - s_1) & : \quad s_1 \sqsubset s \\ find(C', s) & : \quad otherwise \end{cases} \tag{D.12}$$

The following Haskell example code implements this algorithm.

```
lookupPattern (Br lst) ptn = find lst where
    find [] = 0
    find ((s, t):xs)
        | ptn `isPrefixOf` s = numberOfBranch t
        | s `isPrefixOf` ptn = lookupPattern t (drop (length s) ptn)
        | otherwise = find xs
    numberOfBranch (Br ys) = length ys
    numberOfBranch _ = 1

findPattern s ptn = lookupPattern (suffixTree $ s++"$") ptn
```

We always append special terminator to the string (the '$' in above program), so that there won't be any suffix becomes the prefix of the other[3].

Suffix tree also supports searching pattern like "a**n", we skip it here. Readers can refer to [13] and [14] for details.

D.4.2 Find the longest repeated sub-string

After adding a special terminator character to string S, The longest repeated sub-string can be found by searching the deepest branches in suffix tree.

Consider the example suffix tree shown in figure D.11

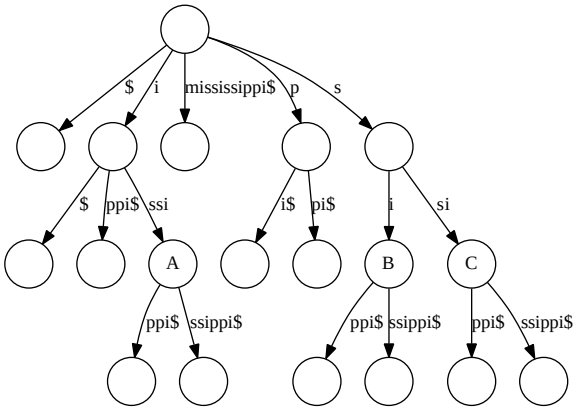


Figure D.11: The suffix tree for 'mississippi$'

There are three branch nodes, A, B, and C with depth 3. However, A represents the longest repeated sub-string "issi". B and C represent for "si", "ssi", they are shorter than A.

www.dbooks.org

This example tells us that the "depth" of the branch node should be measured by the number of characters traversed from the root. But not the number of explicit branch nodes.

To find the longest repeated sub-string, we can perform BFS in the suffix tree.

```
1: function LONGEST-REPEATED-SUBSTRING(T)
2:     Q ← (NIL, ROOT(T))
3:     R ← NIL
4:     while Q is not empty do
5:         (s, T) ← POP(Q)
6:         for each ((l, r), T') ∈ CHILDREN(T) do
7:             if T' is not leaf then
8:                 s' ← CONCATENATE(s, (l, r))
9:                 PUSH(Q, (s', T'))
10:                R ← UPDATE(R, s')
11:    return R
```

This algorithm initializes a queue with a pair of an empty string and the root. Then it repeatedly examine the candidate in the queue.

For each node, the algorithm examines each children one by one. If it is a branch node, the child is pushed back to the queue for further search. And the sub-string represented by this child will be treated as a candidate of the longest repeated sub-string.

Function $\textsc{Update}(R, s')$ updates the longest repeated sub-string candidates. If multiple candidates have the same length, they are all kept in a result list.

```
1: function UPDATE(L, s)
2:     if L = NIL ∨|l_1| < |s| then
3:         return l ← {s}
4:     if |l_1| = |s| then
5:         return APPEND(L, s)
6:     return L
```

The above algorithm can be implemented in Python as the following example program.

```
def lrs(t):
    queue = [("", t.root)]
    res = []
    while len(queue)>0:
        (s, node) = queue.pop(0)
        for _, (str_ref, tr) in node.children.items():
            if len(tr.children)>0:
                s1 = s+t.substr(str_ref)
                queue.append((s1, tr))
                res = update_max(res, s1)
    return res

def update_max(lst, x):
    if lst ==[] or len(lst[0]) < len(x):
        return [x]
    if len(lst[0]) == len(x):
        return lst + [x]
```

```
    return lst
```

Searching the deepest branch can also be realized recursively. If the tree is just a leaf node, empty string is returned, else the algorithm tries to find the longest repeated sub-string from the children.

$$LRS(T) = \begin{cases} \phi & : \quad leaf(T) \\ longest(\{s_i \cup LRS(T_i) | (s_i, T_i) \in C, \neg leaf(T_i)\}) & : \quad otherwise \end{cases} \tag{D.13}$$

The following Haskell example program implements the longest repeated sub-string algorithm.

```
isLeaf Lf = True
isLeaf _ = False

lrs' Lf = ""
lrs' (Br lst) = find $ filter (not ∘ isLeaf ∘ snd) lst where
    find [] = ""
    find ((s, t):xs) = maximumBy (compare `on` length) [s++(lrs' t), find xs]
```

D.4.3 Find the longest common sub-string

The longest common sub-string, can also be quickly found with suffix tree. The solution is to build a generalized suffix tree. If the two strings are denoted as txt_1 and txt_2, a generalized suffix tree is $SuffixTree(txt_1\$_1 txt_2\$_2)$. Where $\$_1$ is a special terminator character for txt_1, and $\$_2 \neq \$_1$ is another special terminator character for txt_2.

The longest common sub-string is indicated by the deepest branch node, with two forks corresponding to both "...$\$_1$..." and "...$\$_2$"(no $\$_1$). The definition of the *deepest* node is as same as the one for the longest repeated sub-string, it is the number of characters traversed from root.

If a node has "...$\$_1$..." under it, the node must represent a sub-string of txt_1, as $\$_1$ is the terminator of txt_1. On the other hand, since it also has "...$\$_2$" (without $\$_1$), this node must represent a sub-string of txt_2 too. Because it's the deepest one satisfied this criteria, so the node represents the longest common sub-string.

Again, we can use BFS (bread first search) to find the longest common sub-string.

```
1:  function LONGEST-COMMON-SUBSTRING(T)
2:      Q ← (NIL, ROOT(T))
3:      R ← NIL
4:      while Q is not empty do
5:          (s, T) ← POP(Q)
6:          if MATCH-FORK(T) then
7:              R ← UPDATE(R, s)
8:          for each ((l, r), T') ∈ CHILDREN(T) do
9:              if T' is not leaf then
10:                 s' ← CONCATENATE(s, (l, r))
11:                 PUSH(Q, (s', T'))
12:     return R
```

www.dbooks.org

Most part is as same as the the longest repeated sub-sting searching algorithm. The function MATCH-FORK checks if the children satisfy the common sub-string criteria.

```
1: function MATCH-FORK(T)
2:     if | CHILDREN(T) | = 2 then
3:         {(s_1, T_1), (s_2, T_2)} ← CHILDREN(T)
4:         return T_1 is leaf ∧T_2 is leaf ∧ XOR($_1 ∈ s_1, $_1 ∈ s_2))
5:     return FALSE
```

In this function, it checks if the two children are both leaf. One contains $\$_2$, while the other doesn't. This is because if one child is a leaf, it always contains $\$_1$ according to the definition of suffix tree.

The following Python program implement the longest common sub-string program.

```
def lcs(t):
    queue = [("", t.root)]
    res = []
    while len(queue)>0:
        (s, node) = queue.pop(0)
        if match_fork(t, node):
            res = update_max(res, s)
        for _, (str_ref, tr) in node.children.items():
            if len(tr.children)>0:
                s1 = s + t.substr(str_ref)
                queue.append((s1, tr))
    return res

def is_leaf(node):
    return node.children=={}

def match_fork(t, node):
    if len(node.children)==2:
        [(_, (str_ref1, tr1)), (_, (str_ref2, tr2))]=node.children.items()
        return is_leaf(tr1) and is_leaf(tr2) and
            (t.substr(str_ref1).find('#')!=-1) !=
            (t.substr(str_ref2).find('#')!=-1)
    return False
```

The longest common sub-string finding algorithm can also be realized recursively. If the suffix tree T is a leaf, the result is empty; Otherwise, we examine all children in T. For those satisfy the matching criteria, the sub-string are collected as candidates; for those don't matching, we recursively search the common sub-string among the children. The longest candidate is selected as the final result.

$$LCS(T) = \begin{cases} \phi & : leaf(T) \\ longest(\begin{array}{l} \{s_i | (s_i, T_i) \in C, match(T_i)\} \cup \\ \{s_i \cup LCS(T_i) | (s_i, T_i) \in C, \neg match(T_i)\}) \end{array} & : otherwise \end{cases} \tag{D.14}$$

The following Haskell example program implements the longest common sub-string algorithm.

```
lcs Lf = []
lcs (Br lst) = find $ filter (not ∘ isLeaf ∘ snd) lst where
    find [] = []
    find ((s, t):xs) = maxBy (compare `on` length)
                       (if match t
                        then s:(find xs)
                        else  (map (s++) (lcs t)) ++ (find xs))

match (Br [(s1, Lf), (s2, Lf)]) = ("#" `isInfixOf` s1) /= ("#" `isInfixOf` s2)
match _ = False
```

D.4.4 Find the longest palindrome

A palindrome is a string, S, such that $S = reverse(S)$. For example, "level", "rotator", "civic" are all palindrome.

The longest palindrome in a string $s_1 s_2 ... s_n$ can be found in $O(n)$ time with suffix tree. The solution can be benefit from the longest common sub-string algorithm.

For string S, if sub-string w is a palindrome, then it must be sub-string of $reverse(S)$ too. for instance, "issi" is a palindrome, it is a sub-string of "mississippi". When reverse to "ippississim", "issi" is also a sub-string.

Based on this fact, we can find the longest palindrome by searching the longest common sub-string for S and $reverse(S)$.

$$palindrome_m(S) = LCS(suffixTree(S \cup reverse(S))) \tag{D.15}$$

The following Haskell example program finds the longest palindrome.

```
longestPalindromes s = lcs $ suffixTree (s++"#"++(reverse s)++"$")
```

D.4.5 Others

Suffix tree can also be used for data compression, such as Burrows-Wheeler transform, LZW compression (LZSS) etc. [3]

D.5 Notes and short summary

Suffix Tree was first introduced by Weiner in 1973 [2]. In 1976, McCreight greatly simplified the construction algorithm. McCreight constructs the suffix tree from right to left. In 1995, Ukkonen gave the first on-line construction algorithms from left to right. All the three algorithms are linear time ($O(n)$). And some research shows the relationship among these 3 algorithms. [7]

www.dbooks.org

Bibliography

[1] Esko Ukkonen. "On-line construction of suffix trees". Algorithmica 14 (3): 249–260. doi:10.1007/BF01206331. http://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf

[2] Weiner, P. "Linear pattern matching algorithms", 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1-11, doi:10.1109/SWAT.1973.13

[3] Suffix Tree, Wikipedia. http://en.wikipedia.org/wiki/Suffix_tree

[4] Esko Ukkonen. "Suffix tree and suffix array techniques for pattern analysis in strings". http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

[5] Trie, Wikipedia. http://en.wikipedia.org/wiki/Trie

[6] Suffix Tree (Java). http://en.literateprograms.org/Suffix_tree_(Java)

[7] Robert Giegerich and Stefan Kurtz. "From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction". Science of Computer Programming 25(2-3):187-218, 1995. http://citeseer.ist.psu.edu/giegerich95comparison.html

[8] Robert Giegerich and Stefan Kurtz. "A Comparison of Imperative and Purely Functional Suffix Tree Constructions". Algorithmica 19 (3): 331–353. doi:10.1007/PL00009177. www.zbh.uni-hamburg.de/pubs/pdf/GieKur1997.pdf

[9] Bryan O'Sullivan. "suffixtree: Efficient, lazy suffix tree implementation". http://hackage.haskell.org/package/suffixtree

[10] Danny. http://hkn.eecs.berkeley.edu/ dyoo/plt/suffixtree/

[11] Zhang Shaojie. "Lecture of Suffix Trees". http://www.cs.ucf.edu/ shzhang/-Combio09/lec3.pdf

[12] Lloyd Allison. "Suffix Trees". http://www.allisons.org/ll/AlgDS/Tree/Suffix/

[13] Esko Ukkonen. "Suffix tree and suffix array techniques for pattern analysis in strings". http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

[14] Esko Ukkonen "Approximate string-matching over suffix trees". Proc. CPM 93. Lecture Notes in Computer Science 684, pp. 228-242, Springer 1993. http://www.cs.helsinki.fi/u/ukkonen/cpm931.ps

www.dbooks.org

GNU Free Documentation License

Version 1.3, 3 November 2008
Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

www.dbooks.org

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section

when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

www.dbooks.org

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses,

www.dbooks.org

the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

www.dbooks.org

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

8 queens puzzle, 452

Auto completion, 116
AVL tree, 75
    balancing, 80
    definition, 75
    deletion, 83
    imperative insertion, 83
    insertion, 78
    verification, 82

B-tree, 127
    delete, 136
    insert, 129
    search, 150
    split, 129
BFS, 481
Binary heap, 157
    build heap, 160
    decrease key, 167
    heap push, 169
    Heapify, 159
    insertion, 169
    merge, 174
    pop, 162
    top, 162
    top-k, 166
Binary Random Access List
    Definition, 286
    Insertion, 288
    Random access, 291
    Remove from head, 289
Binary search, 402
binary search tree, 23
    data layout, 24
    delete, 34
    insertion, 27
    looking up, 31
    min/max, 32
    randomly build, 38
    search, 31
    succ/pred, 32
    traverse, 28
binary tree, 24
Binomial Heap
    Linking, 218
Binomial heap, 213
    definition, 216
    insertion, 220
    pop, 225
Binomial tree, 213
    merge, 222
Boyer-Moor majority number, 419
Boyer-Moore algorithm, 436
Breadth-first search, 481

Change-making problem, 493
Cock-tail sort, 197

Deep-first search, 446
DFS, 446
Dynamic programming, 495

Fibonacci Heap, 228
    decrease key, 241
    delete min, 232
    insert, 230
    merge, 231
    pop, 232
Finger Tree
    Imperative splitting, 334
Finger tree
    Append to tail, 318
    Concatenate, 320
    Definition, 307
    Ill-formed tree, 313
    Imperative random access, 332
    Insert to head, 309
    Random access, 325, 331
    Remove from head, 312
    Remove from tail, 319
    Size augmentation, 325

www.dbooks.org

splitting, 330
folding, 558

Grady algorithm, 483

Heap sort, 169
Huffman coding, 483

Implicit binary heap, 157
in-order traverse, 28
Insertion sort
binary search, 46
binary search tree, 49
linked-list setting, 47
insertion sort, 43
insertion, 44
Integer Patricia, 96
Integer prefix tree, 96
Integer tree
insert, 97
look up, 103
Integer trie, 92
insert, 93
look up, 95

Kloski puzzle, 474
KMP, 424
Knuth-Morris-Pratt algorithm, 424

LCS, 501
left child, right sibling, 217
Leftist heap, 172
heap sort, 175
insertion, 174
merge, 173
pop, 174
rank, 172
S-value, 172
top, 174
List
append, 525
break, 553
concat, 534
concats, 563
cons, 519
Construction, 519
definition, 517
delete, 531
delete at, 531
drop, 551
drop while, 552
elem, 564
empty, 518
empty testing, 520
existence testing, 564
Extract sub-list, 551
filter, 565
find, 565
fold from left, 560
fold from right, 558
foldl, 560
foldr, 558
for each, 546
get at, 521
group, 554
head, 518
index, 521
infix, 568
init, 522
insert, 528
insert at, 528
last, 522
length, 520
lookup, 565
map, 543, 544
matching, 568
maximum, 539
minimum, 539
mutate, 525
prefix, 568
product, 535
reverse, 549
Reverse index, 523
rindex, 523
set at, 527
span, 553
split at, 551, 553
suffix, 568
sum, 535
tail, 518
take, 551
take while, 552
Transformation, 542
unzip, 570
zip, 570
Longest common sub-string, 619
Longest common subsequence problem, 501
Longest palindrome, 621
Longest repeated sub-string, 617

Maximum sum problem, 422
Maze problem, 446
Merge Sort, 368
  Basic version, 368
  Bottom-up merge sort, 389
  In-place merge sort, 375
  In-place working area, 377
  Linked-list merge sort, 381
  Merge, 368
  Naive in-place merge, 376
  Nature merge sort, 383
  Performance analysis, 371
  Work area allocation, 372
minimum free number, 7
MTF, 337

Paired-array list
  Definition, 299
  Insertion and appending, 300
  Random access, 300
  Removing and balancing, 301
Pairing heap, 246
  decrease key, 248
  definition, 246
  delete, 252
  delete min, 249
  find min, 246
  insert, 246
  pop, 249
  top, 246
Parallel merge sort, 391
Parallel quick sort, 391
Patricia, 109
Peg puzzle, 455
post-order traverse, 28
pre-order traverse, 28
Prefix tree, 109
  insert, 110
  look up, 115

Queue
  Balance Queue, 270
  Circular buffer, 263
  Incremental concatenate, 274
  Incremental reverse, 272
  Lazy real-time queue, 279
  Paired-array queue, 269
  Paired-list queue, 266
  Real-time Queue, 272
  Singly linked-list, 260
Quick Sort
  2-way partition, 357
  3-way partition, 359
  Accmulated partition, 350
  Accumulated quick sort, 350
  Average case analysis, 352
  Basic version, 344
  Engineering improvement, 355
  Handle duplicated elements, 355
  Insertion sort fall-back, 366
  One pass functional partition, 349
  Performance analysis, 351
  Strict weak ordering, 345
Quick sort, 343
  partition, 346

Radix tree, 91
range traverse, 34
Reb-black tree
  doubly black, 577
Red-black tree
  Imperative deletion, 577
red-black tree, 53, 58
  deletion, 63
  imperative insertion, 69
  insertion, 60
  red-black properties, 58

Saddelback search, 407
Selection algorithm, 398
selection sort, 189
  minimum finding, 191
  parameterize the comparator, 195
  tail-recursive call minimum finding, 193
Sequence
  Binary random access list, 286
  Concatenate-able list, 303
  finger tree, 306
  Imperative binary random access list, 296
  numeric representation for binary random access list, 293
  Paired-array list, 299
Skew heap, 176
  insertion, 176
  merge, 176
  pop, 176
  top, 176
Splay heap, 178

www.dbooks.org

insertion, 183
merge, 184
pop, 183
splaying, 178
top, 183
Subset sum problem, 506
Suffix link, 601
Suffix tree, 599, 606
active point, 606
Canonical reference pair, 607
end point, 606
functional construction, 613
node transfer, 601
on-line construction, 608
reference pair, 607
string searching, 615
sub-string occurrence, 616
Suffix trie, 600
on-line construction, 602

T9, 121
Tail call, 536
Tail recursion, 536
Tail recursive call, 536
Textonym input method, 121
The wolf, goat, and cabbage puzzle, 461
Tounament knock out
explicit infinity, 206
tree reconstruction, 30
tree rotation, 56
Trie, 104
insert, 106
look up, 108
Trounament knock out, 201

Ukkonen's algorithm, 608

Water jugs puzzle, 466
word counter, 23