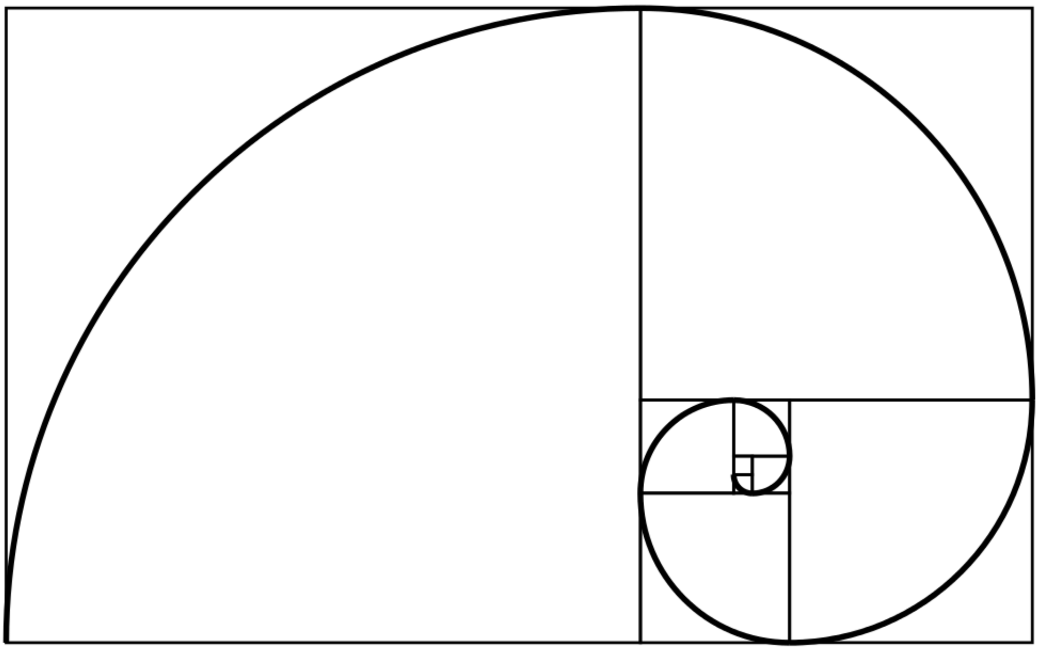
Elementary Algorithms



Xinyu LIU [1]

March 6, 2023

[1] **Xinyu LIU**

Version: $e = \sum_{n=0}^{\infty} \frac{1}{n!} = 1 + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \cdots = 2.718283$

Email: liuxinyu95@gmail.com

Contents

Preface **9**

1 List **17**

1.1 Introduction . . . 17
1.2 Definition . . . 17
1.2.1 Access . . . 18
1.3 Basic operations . . . 18
1.3.1 index . . . 18
1.3.2 Last . . . 19
1.3.3 Right index . . . 19
1.3.4 Mutate . . . 21
insert . . . 21
delete . . . 23
concatenate . . . 24
1.3.5 sum and product . . . 24
1.3.6 maximum and minimum . . . 26
1.4 Transform . . . 27
1.4.1 map and for-each . . . 28
For each . . . 29
1.4.2 reverse . . . 31
1.5 Sub-list . . . 32
1.5.1 break and group . . . 33
1.6 Fold . . . 35
1.7 Search and filter . . . 38
1.8 zip and unzip . . . 40

2 Binary Search Tree **43**

2.1 Introduction . . . 43
2.2 Data Layout . . . 45
2.3 Insertion . . . 45
2.4 Traverse . . . 47
2.5 Query . . . 49
2.5.1 Look up . . . 49
2.5.2 Minimum and maximum . . . 50
2.5.3 Successor and predecessor . . . 50
2.6 Deletion . . . 51
2.7 Random build . . . 54
2.8 Map . . . 55
2.9 Appendix: Example programs . . . 55

3 Insertion sort **57**
3.1 Introduction . . . 57
3.2 Insertion . . . 58
3.3 Binary search . . . 59
3.4 List . . . 59
3.5 Binary search tree . . . 60
3.6 Summary . . . 61

4 Red-black tree **63**
4.1 Introduction . . . 63
4.1.1 Balance . . . 64
4.1.2 Tree rotation . . . 64
4.2 Definition . . . 66
4.3 Insert . . . 68
4.4 Delete . . . 70
4.5 Imperative red-black tree algorithm ⋆ . . . 74
4.6 Summary . . . 75
4.7 Appendix: Example programs . . . 76

5 AVL tree **79**
5.1 Introduction . . . 79
5.2 Definition . . . 79
5.3 Insert . . . 81
5.3.1 Balance . . . 82
Verification . . . 83
5.4 Imperative AVL tree algorithm ★ . . . 84
5.5 Summary . . . 86
5.6 Appendix: Example programs . . . 86

6 Radix tree **89**
6.1 Integer trie . . . 89
6.1.1 Definition . . . 90
6.1.2 Insert . . . 90
6.1.3 Look up . . . 92
6.2 Integer prefix tree . . . 92
6.2.1 Definition . . . 93
6.2.2 Insert . . . 93
6.2.3 Lookup . . . 97
6.3 Trie . . . 98
6.3.1 Definition . . . 98
6.3.2 Insert . . . 98
6.3.3 Look up . . . 100
6.4 Prefix tree . . . 101
6.4.1 Definition . . . 101
6.4.2 Insert . . . 101
6.4.3 Look up . . . 104
6.5 Applications of trie and prefix tree . . . 105
6.5.1 Dictionary and input completion . . . 105
6.5.2 Predictive text input . . . 107
6.6 Summary . . . 109
6.7 Appendix: Example programs . . . 110

7 B-Tree **115**
- 7.1 Introduction . . . 115
- 7.2 Insert . . . 117
  - 7.2.1 Insert then split . . . 117
  - 7.2.2 Split before insert . . . 120
  - 7.2.3 Paired lists . . . 122
- 7.3 Look up . . . 124
- 7.4 Delete . . . 126
  - 7.4.1 Delete and fix . . . 126
  - 7.4.2 Merge before delete . . . 129
- 7.5 Summary . . . 132
- 7.6 Appendix: Example programs . . . 133

8 Binary Heaps **137**
- 8.1 Definition . . . 137
- 8.2 Binary heap by array . . . 137
  - 8.2.1 Heapify . . . 138
  - 8.2.2 Build . . . 139
  - 8.2.3 Heap operations . . . 140
    - Pop . . . 140
    - Top-k . . . 140
    - Increase priority . . . 142
    - Insertion . . . 142
  - 8.2.4 Heap sort . . . 142
- 8.3 Leftist heap and skew heap . . . 143
  - 8.3.1 Leftist heap . . . 144
    - Merge . . . 145
    - Top and pop . . . 145
    - Insert . . . 146
    - Heap sort . . . 146
  - 8.3.2 Skew heap . . . 146
    - Merge . . . 147
- 8.4 Splay heap . . . 147
  - 8.4.1 Splay . . . 148
  - 8.4.2 Pop . . . 151
  - 8.4.3 Merge . . . 152
- 8.5 Summary . . . 152
- 8.6 Appendix - example programs . . . 152

9 Selection sort **157**
- 9.1 Introduction . . . 157
- 9.2 Find the minimum . . . 158
  - 9.2.1 Performance . . . 159
- 9.3 Improvement . . . 159
  - 9.3.1 Cock-tail sort . . . 160
- 9.4 Further improvement . . . 162
  - 9.4.1 Tournament knock out . . . 162
  - 9.4.2 Heap sort . . . 165
- 9.5 Appendix - example programs . . . 166

10 Binomial heap, Fibonacci heap, and pairing heap **169**
10.1 Introduction . . . 169
10.2 Binomial Heaps . . . 169
Binomial tree . . . 170
10.2.1 Link . . . 172
Insert . . . 173
10.2.2 Merge . . . 174
Pop . . . 175
10.3 Fibonacci heap . . . 176
10.3.1 Insert . . . 177
Merge . . . 177
Pop . . . 178
10.3.2 Increase priority . . . 182
10.3.3 The name of Fibonacci heap . . . 183
10.4 Pairing Heaps . . . 184
10.4.1 Definition . . . 185
10.4.2 Merge, insert, and top . . . 185
10.4.3 Increase priority . . . 185
10.4.4 Pop . . . 186
Delete . . . 187
10.5 Summary . . . 188
10.6 Appendix - example programs . . . 188

11 Queue **193**
11.1 Introduction . . . 193
11.2 Linked-list queue . . . 193
11.3 Circular buffer . . . 194
11.4 Paired-list queue . . . 196
11.5 Balance Queue . . . 197
11.6 Real-time queue . . . 197
11.7 Lazy real-time queue . . . 200
11.8 Appendix - example programs . . . 201

12 Sequence **203**
12.1 Introduction . . . 203
12.2 Binary random access list . . . 203
12.3 Numeric representation . . . 206
12.4 paired-array sequence . . . 209
12.5 Concatenate-able list . . . 210
12.6 Finger tree . . . 211
12.6.1 Insert . . . 212
12.6.2 Extract . . . 213
12.6.3 Append and remove . . . 215
12.6.4 concatenate . . . 215
12.6.5 Random access . . . 216
12.7 Appendix - example programs . . . 218

13 Quick sort and merge sort **223**
13.1 Introduction . . . 223
13.2 Quick sort . . . 223
13.2.1 Partition . . . 224
13.2.2 In-place sort . . . 225

13.2.3 Performance . . . 227
Average case★ . . . 227
13.2.4 Improvement . . . 229
Worst cases . . . 233
13.2.5 quick sort and tree sort . . . 236
13.3 Merge sort . . . 236
13.3.1 Merge . . . 237
13.3.2 Performance . . . 238
Improvement . . . 239
13.3.3 In-place merge sort . . . 240
13.3.4 Nature merge sort . . . 243
13.3.5 Bottom-up merge sort . . . 246
13.4 Parallelism . . . 247
13.5 Summary . . . 247
13.6 Appendix: Example programs . . . 248

14 Solution search **251**
14.1 k selection problem . . . 251
14.2 Binary search . . . 252
14.2.1 2D search . . . 253
14.3 The majority number . . . 261
14.4 Maximum sum of sub-vector . . . 262
14.5 String matching . . . 264
14.6 Solution search . . . 266
14.6.1 DFS and BFS . . . 266
Maze . . . 266
Eight queens puzzle . . . 268
Peg puzzle . . . 270
The wolf, goat, and cabbage puzzle . . . 274
Water jugs puzzle . . . 277
Kloski . . . 281
14.6.2 Greedy algorithm . . . 285
Huffman coding . . . 285
Change making problem . . . 289
14.6.3 Dynamic programming . . . 290
Longest common sub-sequence . . . 292
Subset sum . . . 294
14.7 Appendix - example programs . . . 297

Appendices

A Imperative delete for red-black tree **307**

B AVL tree - proofs and the delete algorithm **315**
I Height increment . . . 315
II Balance adjustment after insert . . . 316
III Delete algorithm . . . 318
* Functional delete . . . 319
† Imperative delete . . . 320
IV Example program . . . 322

C Answers **325**

GNU Free Documentation License **343**
1. APPLICABILITY AND DEFINITIONS . . . 343
2. VERBATIM COPYING . . . 344
3. COPYING IN QUANTITY . . . 345
4. MODIFICATIONS . . . 345
5. COMBINING DOCUMENTS . . . 347
6. COLLECTIONS OF DOCUMENTS . . . 347
7. AGGREGATION WITH INDEPENDENT WORKS . . . 347
8. TRANSLATION . . . 348
9. TERMINATION . . . 348
10. FUTURE REVISIONS OF THIS LICENSE . . . 348
11. RELICENSING . . . 349
ADDENDUM: How to use this License for your documents . . . 349

Preface

Programmers learn elementary algorithms at school. Except for programming contest or code interview, they seldom use algorithms in commercial software development. When talking about algorithms in AI and machine learning, people actually mean scientific modeling, but not about data structure or elementary algorithm. Even when programmers need them, they have already been provided in libraries. It seems quite enough to know about how to use the library as a tool but not 're-invent the wheel'. Elementary algorithms are fundamental things, Let's start with two problems.

The smallest free number

Richard Bird gives an interesting problem to find the minimum number that not appears in a given list (Chapter 1,[?]). People often use number to index entities. A number is either occupied or free. When acquires, we want to always allocate the smallest available one. Suppose numbers are non-negative integers and those being occupied are recorded in a list, for example:

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

How can we find the smallest free number, 10 from the list? It seems quite easy with exhaustive search:

```
function Min-Free(A)
    x ← 0
    loop
        if x ∉ A then
            return x
        else
            x ← x + 1
```

Where the $\notin$ is realized like below.

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

Where $|X|$ is the length of X. Some environments have built-in implementation to test existence of an element. When there are millions of numbers, this solution performs poor. The time spent is quadratic to the length of the list. In a computer with 2 cores of 2.10 GHz CPU, and 2G RAM, the C implementation takes 5.4s to search the minimum free number among 100,000 numbers, and takes more than 8 minutes to handle a million numbers.

Improvement

For n numbers $x_1, x_2, ..., x_n$, if there exists free number, some x_i must be out of the range $[0, n)$; otherwise the list is exactly some permutation of $0, 1, ..., n-1$ hence n should be returned as the minimum free number.

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

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

```
function Min-Free(A)
    F ←[False, False, ..., False]        ▷ n + 1
    for x in A do
        if x < n then
            F[x] ← True
    for i ← 0 to n do
        if F[i] = False then
            return i
```

Initializes F with all False values. For every number x in A, mark the flag $F[x]$ true if $x < n$. Finally, scan F to find the first false flag. This program takes time proportion to n. It uses $n+1$ flags to cover the special case that $sort(A) = [0, 1, 2, ..., n-1]$. It needs $O(n)$ space to store the flags F, then release it when finish. To avoid repeated allocation and release, we can allocate a sufficient big one in advance for reusing, and change to bit-wise flags instead of array. The C implementation handles 1 million numbers in 0.023s in the same computer.

Divide and Conquer

The divide and conquer strategy breaks the problem into smaller ones, then solve them separately. Collect the numbers $x_i \leq \lfloor n/2 \rfloor$ into a sub-list A' and the rest into another sub-list A''. According to (1), if the length of A' equals to $\lfloor n/2 \rfloor$, it means A' is 'full'. The minimum free number must be in A'', otherwise in A'. Both cases lead to a smaller problem. When search in A'', the boundaries change. We do not start from 0, but from $\lfloor n/2 \rfloor + 1$. We define the algorithm as $search(A, l, u)$, where l is the lower bound and u is the upper bound. When start $l = 0$, $u = |A| - 1$, i.e., $minfree(A) = search(A, 0, |A| - 1)$

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

where:

$$\begin{cases} m = \lfloor \dfrac{l+u}{2} \rfloor \\ A' = [x \in A, x \leq m], A'' = [x \in A, x > m] \end{cases}$$

This algorithm doesn't need additional space[1]. Each recursive call performs $O(|A|)$ comparisons to partition A' and A'', hence halves the problem as $T(n) = T(n/2) + O(n)$. We can reduce it to $O(n)$ according to the master theorem. Alternatively, the first call takes $O(n)$ time to partition A' and A'', the second call takes $O(n/2)$ time, the third call takes $O(n/4)$ time ... The total time is $O(n + n/2 + n/4 + ...) = O(2n) = O(n)$. Below example Haskell program implements this algorithm.

[1]The recursion takes $O(\lg n)$ stack spaces, but it can be eliminated through tail recursion optimization

```haskell
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
```

There are $O(\lg n)$ recursive calls. We can eliminate the recursion with loops:

function MIN-FREE(A)
  $l \leftarrow 0, u \leftarrow |A|$
  while $u - l > 0$ **do**
    $m \leftarrow l + \frac{u-l}{2}$
    $left \leftarrow l$
    for $right \leftarrow l$ to $u - 1$ **do**
      if $A[right] \leq m$ **then**
        Exchange $A[left] \leftrightarrow A[right]$
        $left \leftarrow left + 1$
    if $left < m + 1$ **then**
      $u \leftarrow left$
    else
      $l \leftarrow left$

As shown in figure 1, this program re-arranges the array such that all elements before $left$ are less than or equal to m; while those between $left$ and $right$ are greater than m.

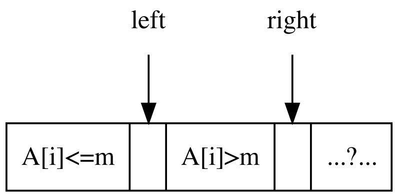


Figure 1: All $A[i] \leq m$ where $0 \leq i < left$; while $A[i] > m$ where $left \leq i < right$. The rest elements are yet to be scanned.

Regular number

The second problem is to find the 1,500-th number, which only contains factor 2, 3 or 5. Such numbers are called the regular number, also known as 5-smooth in number theory, and Hamming numbers named after Richard Hamming. 2, 3, and 5 are definitely regular numbers. $60 = 2^2 3^1 5^1$ is the 25-th regular number. $21 = 2^0 3^1 7^1$ is not because it has a factor of 7. Define $1 = 2^0 3^0 5^0$ as the 0-th regular number. The first 10 numbers are:

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, ...

The brute-force solution

We can check numbers one by one from 1, extract all factors of 2, 3 and 5 to see if the remaining is 1:

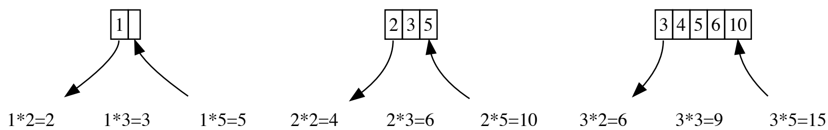
function REGULAR-NUMBER(n)

```
    x ← 1
    while n > 0 do
        x ← x + 1
        if Valid?(x) then
            n ← n − 1
    return x

function Valid?(x)
    while x mod 2 = 0 do
        x ← x/2
    while x mod 3 = 0 do
        x ← x/3
    while x mod 5 = 0 do
        x ← x/5
    return x = 1 ?
```

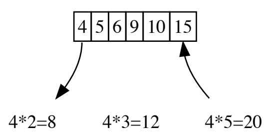
This 'brute-force' algorithm performs poor when n increases. The C implementation takes 40.39s in above computer to find the 1500-th number (860934420).

Improvement

Modular and divide are expensive[2] operations. Instead of checking every number, we can generate regular numbers with 2, 3, 5 in ascending order from 1. We can use the queue data structure to solve this problem. A queue allows to add element to one end (enqueue), and delete from the other end (dequeue). The element enqueued first will be dequeued first (First In First Out). Initialize the queue with the 0th regular number 1, we repeatedly dequeue a number, multiply it by 2, 3, 5 to generate 3 numbers; then add them to the queue in ascending order. If the generated number already exists in the queue, we drop it to avoid duplication, as shown in figure 2.



(a) Start from 1 (b) Add 2, 3, 5 (c) Add 4, 6, 10



(d) Add 9, 15, drop 6

Figure 2: First 4 steps

We can design the algorithm based on this idea:

```
function Regular-Number(n)
    Q ← [1]
    while n > 0 do
        x ← Dequeue(Q)
        Unique-Enqueue(Q, 2x)
        Unique-Enqueue(Q, 3x)
```

```
        Unique-Enqueue(Q, 5x)
        n ← n − 1
    return x

function Unique-Enqueue(Q, x)
    i ← 0, m ← |Q|
    while i < m and Q[i] < x do
        i ← i + 1
    if i ≥ m or x ≠ Q[i] then
        Insert(Q, i, x)
```

The Unique-Enqueue function takes $O(m)$ time to insert an unique element in ascending order, where $m = |Q|$ is the length of the queue. m increases proportion to n (Each time, we dequeue an element, and enqueue 3 new at most. The increase ratio ≤ 2), the total time is $O(1 + 2 + 3 + ... + n) = O(n^2)$. Figure3 shows the number of queue access against n. It is a quadratic curve, which reflects the $O(n^2)$ performance.

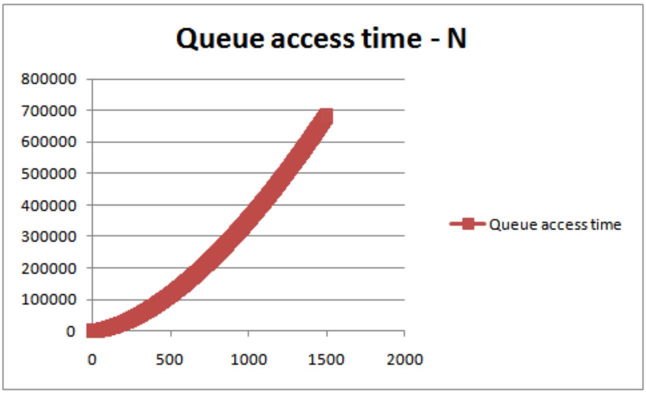


Figure 3: Queue access count - n.

The corresponding C implementation takes 0.016s to output 860934420, about 2500 times faster than the brute-force solution. Let xs be the infinite list of all regular numbers $[x_1, x_2, x_3, ...]$. Multiply every number by 2, the result is again infinite many regular numbers: $[2x_1, 2x_2, 2x_3, ...]$. So as multiple by 3 and 5. If we merge the three infinite series together, filter out the duplicated numbers, and prepend 1 as the first, then we get xs again:

$$xs = 1 : [2x|x \leftarrow xs] \cup [3x|x \leftarrow xs] \cup [5x|x \leftarrow xs] \tag{2}$$

Where symbol $x{:}xs$ links x before list xs. It is called 'cons' in Lisp. 1 is linked as the head of the 0th regular number. $\cup$ implements the infinite lists merge:

$$(a{:}as) \cup (b{:}bs) = \begin{cases} a < b: & a : as \cup (b{:}bs) \\ a = b: & a : as \cup bs \\ a > b: & b : (a{:}as) \cup bs \end{cases}$$

Below is the example program in Haskell:

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

merge (a:as) (b:bs) | a < b = a : merge as (b:bs)
                    | a == b = a : merge as bs
                    | otherwise = b : merge (a:as) bs
```

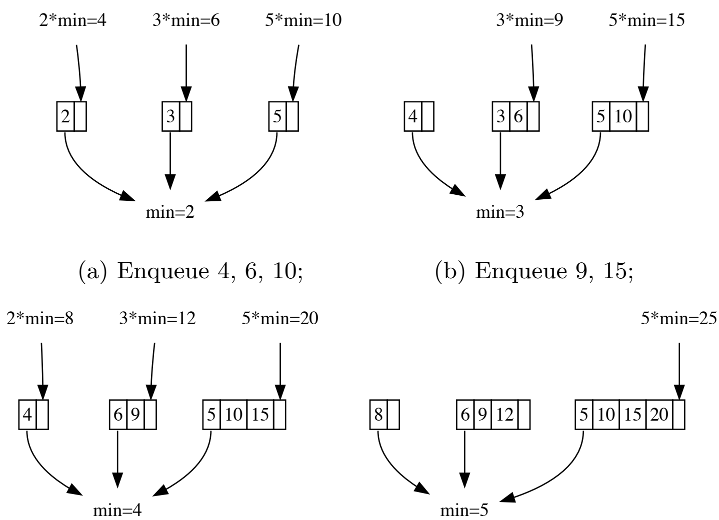
The 1500th number 860934420 is given by `ns !! 1500`. It takes 0.03s to output the answer in the same computer.

Queues

The above solution generates and filters out duplicated numbers. It need scan the queue to keep the ascending order. We category all regular numbers into 3 disjoint buckets: $Q_2 = \{2^i | i > 0\}$, $Q_{23} = \{2^i 3^j | i \geq 0, j > 0\}$, and $Q_{235} = \{2^i 3^j 5^k | i, j \geq 0, k > 0\}$. The constraints that $j \neq 0$ in Q_{23}, and $k \neq 0$ in Q_{235} ensure there is no overlap. Realize the buckets as 3 queues starting from $Q_2 = \{2\}$, $Q_{23} = \{3\}$, and $Q_{235} = \{5\}$. Each time extract the smallest number x from the three queues, then do the following:

- If x comes from Q_2, enqueue $2x$ to Q_2, $3x$ to Q_{23}, and $5x$ to Q_{235};
- If x comes from Q_{23}, enqueue $3x$ to Q_{23}, and $5x$ to Q_{235}. We do not add $2x$ to Q_2, because Q_2 does not hold any numbers divisible by 3.
- If x comes from Q_{235}, enqueue $5x$ to Q_{235}. We do not add $2x$ to Q_2, or $3x$ to Q_{23} because they don't hold numbers divisible by 5.

We reach to the answer after dequeue n smallest numbers from the three queues. Figure 4 gives the first 4 steps.



(a) Enqueue 4, 6, 10; (b) Enqueue 9, 15;

(c) Enqueue 8, 12, 20; (d) Enqueue 25.

Figure 4: First 4 steps with Q_2, Q_{23}, Q_{235}.

function REGULAR-NUMBER(n)
  $x \leftarrow 1$
  $Q_2 \leftarrow \{2\}$, $Q_{23} \leftarrow \{3\}$, $Q_{235} \leftarrow \{5\}$
  while $n > 0$ **do**
    $x \leftarrow min(\text{HEAD}(Q_2), \text{HEAD}(Q_{23}), \text{HEAD}(Q_{235}))$
    if $x = \text{HEAD}(Q_2)$ **then**
      DEQUEUE(Q_2)
      ENQUEUE($Q_2, 2x$)
      ENQUEUE($Q_{23}, 3x$)

```
            Enqueue(Q_235, 5x)
        else if x = Head(Q_23) then
            Dequeue(Q_23)
            Enqueue(Q_23, 3x)
            Enqueue(Q_235, 5x)
        else
            Dequeue(Q_235)
            Enqueue(Q_235, 5x)
        n ← n − 1
    return x
```

This algorithm loops n times. Each time extracts the minimum number from three queues in constant time. Then adds at most 3 numbers to the queues in constant time. The overall performance is $O(n)$.

Summary

Although the brute-force solution solve both puzzles, they can't scale up. This book aims to provide both functional and imperative elementary algorithms and data structures. We referenced many results from Okasaki's work[3] and classic text books[4]. We try to avoid relying on any specific programming language, because the reader may or may not be familiar with it, and programming languages keep changing. Instead, we use pseudo code or mathematics notation to make the algorithm definition generic. When give code examples, the functional ones look more like Haskell, and the imperative ones look like a mix of several languages.

I wrote the first edition from 2009 to 2017, then rewrote the second edition from 2020 to 2023. The pdf version is available in github.

Exercise 1

1. For the free number puzzle, since all numbers are not negative, we can leverage the sign as a flag to indicate a number exists. We can scan the number list, for every number $|x| < n$ (where n is the length), negate the number at position $|x|$. Then we run another round of scan to find out the first positive number. It's position is the answer. Write a program to realize this method.
2. There are n numbers 1, 2, ..., n. After some processing, they are shuffled, and a number x is altered to y. Suppose $1 \leq y \leq n$, design a solution to find x and y in linear time with constant space.
3. Below example program is a solution for the regular number puzzle. Is it equivalent to the queue based solution?

```
Int regularNum(Int m) {
    [Int] nums(m + 1)
    Int n = 0, i = 0, j = 0, k = 0
    nums[0] = 1
    Int x2 = 2 * nums[i]
    Int x3 = 3 * nums[j]
    Int x5 = 5 * nums[k]
    while n < m {
        n = n + 1
        nums[n] = min(x2, x3, x5)
        if x2 == nums[n] {
            i = i + 1
            x2 = 2 * nums[i]
        }
```

```
        if x3 == nums[n] {
            j = j + 1
            x3 = 3 * nums[j]
        }
        if x5 == nums[n] {
            k = k + 1
            x5 = 5 * nums[k]
        }
    }
    return nums[m]
}
```

Chapter 1

List

1.1 Introduction

List and array are build blocks for other complex data structure. Both hold multiple elements as a container. Array is a range of consecutive cells indexed by a number (address). It is typically bounded with fixed size. While list increases on-demand. One can traverse a list one by one from head to tail. Particularly in functional settings, list plays critical role to control the computation and logic flow[1]. Readers already be familiar with map, filter, fold are safe to skip this chapter, and directly start from chapter 2.

1.2 Definition

List, or singly linked-list is a data structure recursively defined as: A *list* is either empty, denoted as [ ] or NIL; or contains an element and liked with a *list*. Figure 1.1 shows a list of nodes. Each contains two parts, an element (key), and a reference to the sub-list (next). The next to the last node is empty (NIL).

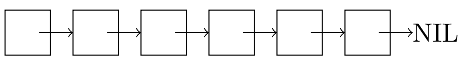


Figure 1.1: A list of nodes

Every node links to the next or NIL. We often define list with the compound structure[2], for example:

```
data List<A> {
    A key
    List<A> next
}
```

Many traditional environments support the NIL concept. There are two ways to represent the empty list: one is to use NIL (or null, or ∅) directly; the other is to create a list, but put nothing as [ ]. From implementation perspective, NIL need not allocate any memories, while [ ] does.

[1]In low level, lambda calculus plays the most critical role as one of the computation model equivalent to Turing machine[93], [99].

[2]In most cases, the data stored in list have the same type. However, there is also heterogeneous list, like the list in Lisp for example.

1.2.1 Access

Given a none empty list X, define two functions[3] to access the first element, and the rest sub-list. They are often called as *first* X and *rest* X, or *head* X and *tail* X[4]. Conversely, we can construct a list from an element x and another list xs (can be empty), as $x{:}xs$. It is called the `cons` operation. We have the following equations:

$$\begin{cases} head\ (x{:}xs) & = x \\ tail\ (x{:}xs) & = xs \end{cases} \tag{1.1}$$

For a none empty list X, we also denote the first element as x_1, and the rest sub-list as X'. For example, when $X = [x_1, x_2, x_3, ...]$, then $X' = [x_2, x_3, ...]$.

Exercise 1.2

1. For list of type A, suppose we can test if any two elements $x, y \in A$ are equal, define an algorithm to test if two lists are identical.

1.3 Basic operations

From the definition, we can count the length recursively. the length of the empty list is 0, or it is 1 plus the length of the sub-list.

$$\begin{array}{rcl} length\ [\,] & = & 0 \\ length\ (x{:}xs) & = & 1 + length\ xs \end{array} \tag{1.2}$$

We traverse the list to count the length, the performance is bound to $O(n)$, where n is the number of elements. We use $|X|$ as the length of X when the context is clear. To avoid repeatedly counting, we can persist the length in a variable, and update it when mutate (add or delete). Below is the iterative length counting:

```
function Length(X)
    n ← 0
    while X ≠ NIL do
        n ← n + 1
        X ← Next(X)
    return n
```

1.3.1 index

Array supports random access at position i in constant time, while we need traverse the list i steps to access the target element.

$$getAt\ i\ (x{:}xs) = \begin{cases} i = 0: & x \\ i \neq 0: & getAt\ (i-1)\ xs \end{cases} \tag{1.3}$$

We leave the empty list not handled. The behavior when $[\,]$ is undefined. As such, the out of bound case also leads to the undefined behavior. If $i > |X|$, we end up the edge case to access the $(i - |X|)$ position of the empty list. On the other hand, if $i < 0$, after minus it by one, it's even farther away from 0, and finally ends up with some negative position of the empty list. *getAt* is bound to $O(i)$ time as it advances the list i steps. Below is the imperative implementation:

[3] We often write function $f(x)$ as $f\ x$, and $f(x, y, ..., z)$ as $f\ x\ y\ ...\ z$.

[4] They are named as `car` and `cdr` in Lisp due to the design of machine registers[63].

```
function Get-At(i, X)
    while i ≠ 0 do
        X ← Next(X)                          ▷ error when X = NIL
        i ← i − 1
    return First(X)
```

Exercise 1.3

1. In the iterative Get-At(i, X) implementation, what is the behavior when X is empty? what if i is out of the bound or negative?

1.3.2 Last

There is a pair of symmetric operations to 'first/rest', namely 'last/init'. For a none empty list $X = [x_1, x_2, ..., x_n]$, function *last* returns the tail element x_n, while *init* returns the sub-list of $[x_1, x_2, ..., x_{n-1}]$. Although they are symmetric pairs left to right, 'last/init' need traverse the list, hence are linear time.

$$\begin{array}{rcl} last\ [x] & = & x \\ last\ (x{:}xs) & = & last\ xs \end{array} \qquad \begin{array}{rcl} init\ [x] & = & [\,] \\ init\ (x : xs) & = & x : init\ xs \end{array} \tag{1.4}$$

Both do not handle the empty list. The behavior is undefined with [ ]. Below are the iterative implementation:

```
function Last(X)
    x ← NIL
    while X ≠ NIL do
        x ← First(X)
        X ← Rest(X)
    return x

function Init(X)
    X' ← NIL
    while Rest(X) ≠ NIL do                   ▷ Error when X is NIL
        X' ← Cons(First(X), X')
        X ← Rest(X)
    return Reverse(X')
```

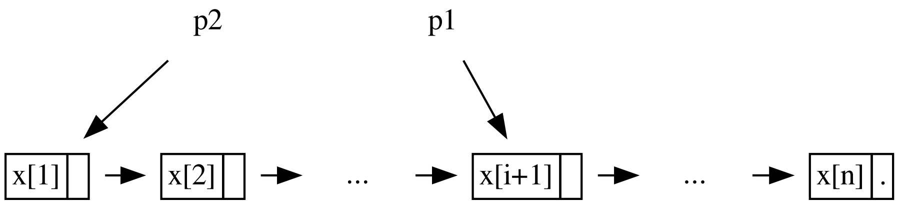
Init accumulates the result through Cons. However, the order is reversed. We need reverse (section 1.4.2) it back.

1.3.3 Right index

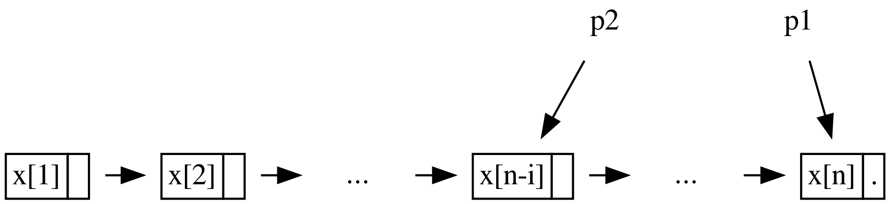
last is a special case of right index. The generic case is to find the last i-th element (from right). The naive implementation traverses two rounds: count the length n first, then access the $(n - i - 1)$-th element from left:

$$lastAt\ i\ X = getAt\ (|X| - i - 1)\ L$$

The better solution uses two pointers p_1, p_2 with the distance if i, i.e., $rest^i(p_2) = p_1$, where $rest^i(p_2)$ means repeatedly apply *rest* for i times. When advance p_2 by i steps, it meets p_1. p_2 starts from the head. Advance both pointers in parallel till p_1 arrives at tail. At this time point, p_2 exactly points to the i-th element from right. as shown in figure 1.2. p_1 and p_2 form a sliding window of width i.



(a) p_2 starts from the head, behind p_1 in i steps.



(b) When p_1 reaches the tail, p_2 points to the i-th element from right.

Figure 1.2: Sliding window

function LAST-AT(i, X)
  $p \leftarrow X$
  while $i > 0$ **do**
    $X \leftarrow$ REST(X) ▷ Error if out of bound
    $i \leftarrow i - 1$
  while REST(X) $\neq$ NIL **do**
    $X \leftarrow$ REST(X)
    $p \leftarrow$ REST(p)
  return FIRST(p)

We can't alter the pointers in purely functional settings. Instead, we advance two lists $X = [x_1, x_2, ..., x_n]$ and $Y = [x_i, x_{i+1}, ..., x_n]$ simultaneously, where Y is the sub-list without the first $i - 1$ elements.

$$lastAt\ i\ X\ = slide\ X\ (drop\ i\ X) \tag{1.5}$$

Where:

$$\begin{array}{rcl} slide\ (x{:}xs)\ [y] & = & x \\ slide\ (x{:}xs)\ (y{:}ys) & = & slide\ xs\ ys \end{array} \tag{1.6}$$

Function $drop\ m\ X$ discards the first m elements.

$$\begin{array}{rcl} drop\ 0\ xs & = & xs \\ drop\ m\ [\,] & = & [\,] \\ drop\ m\ (x{:}xs) & = & drop\ (m-1)\ xs \end{array} \tag{1.7}$$

Exercise 1.4

1. In the INIT algorithm, can we use APPEND(X', FIRST(X)) instead of CONS?
2. How to handle empty list or out of bound error in LAST-AT?

1.3.4 Mutate

Mutate includes append, insert, update, and delete. The functional environment actually implements mutate by creating a new list for the changed part, while keeps (persists) the original one for reuse, or release at sometime (chapter 2 in[3]).

Append is the symmetric operation of *cons*, it appends element to the tail, while *cons* add from head. It is also known as 'snoc' (reverse of 'cons'). As it need traverse the list to the tail, the performance is $O(n)$, where n is the length. To avoid repeatedly traverse, we can persist the tail reference, and update it for changes.

$$
\begin{array}{rcl}
append\ [\ ]\ x & = & [x] \\
append\ (y{:}ys)\ x & = & y : append\ ys\ x
\end{array}
\tag{1.8}
$$

Below is the corresponding iterative implementation[5]:

```
function Append(X, x)
    if X = NIL then
        return Cons(x, NIL)
    H ← X                                   ▷ Copy of the head
    while Rest(X) ≠ NIL do
        X ← Rest(X)
    Rest(X) ← Cons(x, NIL)
    return H
```

To update the Rest, it is typically implemented by updating the `next` reference, for example:

```
List<A> append(List<A> xs, A x) {
    if xs == null then return cons(x, null)
    var head = xs
    while xs.next ≠ null {
        xs = xs.next
    }
    xs.next = cons(x, null)
    return head
}
```

Similar to *getAt*, we need advance to the target position and change the element.

$$
\begin{array}{rcl}
setAt\ 0\ x\ (y{:}ys) & = & x : ys \\
setAt\ i\ x\ (y{:}ys) & = & y : setAt\ (i-1)\ x\ ys
\end{array}
\tag{1.9}
$$

The *setAt* is bound to $O(i)$ time, where i is the position for update.

Exercise 1.5

1. Add the 'tail' reference, optimize the *append* to constant time.
2. When need update the tail reference? How does it affect the performance?
3. Handle the empty list and out of bound error for *setAt*.

insert

There are two different cases about insertion: (1) insert an element at a given position: *insert* i x X, similar to *setAt*; (2) insert an element to a sorted list, and maintain the ordering.

$$
\begin{array}{rcl}
insert\ 0\ x\ ys & = & x : ys \\
insert\ i\ x\ (y{:}ys) & = & y : insert\ (i-1)\ x\ ys
\end{array}
\tag{1.10}
$$

[5]The parameter orders are also symmetric: *cons x xs* and *append xs x*

When i exceeds the length, treat it as append (the exercise of this section). Below is the iterative implementation:

```
function Insert(i, x, X)
    if i = 0 then
        return Cons(x, X)
    H ← X
    p ← X
    while i > 0 and X ≠ NIL do
        p ← X
        X ← Rest(X)
        i ← i − 1
    Rest(p) ← Cons(x, X)
    return H
```

Let the list $L = [x_1, x_2, ..., x_n]$ be sorted, i.e., for any position $1 \leq i \leq j \leq n$, then $x_i \leq x_j$. Where $\leq$ is abstract ordering. It can be $\geq$, subset between sets, and etc. We define *insert* to maintain the ordering.

$$
\begin{array}{rcl}
insert\ x\ [\ ] & = & [x] \\
insert\ x\ (y{:}ys) & = & \begin{cases} x \leq y : & x{:}y{:}ys \\ \text{否则} : & y : insert\ x\ ys \end{cases}
\end{array}
\tag{1.11}
$$

Since it need compare elements one by one, the performance is bound to $O(n)$ time, where n is the length. Below is the iterative implementation:

```
function Insert(x, X)
    if X = NIL or x < First(X) then
        return Cons(x, X)
    H ← X
    while Rest(X) ≠ NIL and First(Rest(X)) < x do
        X ← Rest(X)
    Rest(X) ← Cons(x, Rest(X))
    return H
```

With *insert*, we can further define the insertion sort: repeatedly insert elements to the empty list. Since each insert takes liner time, the overall time is bound to $O(n^2)$.

$$
\begin{array}{rcl}
sort\ [\ ] & = & [\ ] \\
sort\ (x{:}xs) & = & insert\ x\ (sort\ xs)
\end{array}
\tag{1.12}
$$

We can eliminate the recursion to implement the iterative implementation. Scan the list, and insert elements one by one:

```
function Sort(X)
    S ← NIL
    while X ≠ NIL do
        S ← Insert(First(X), S)
        X ← Rest(X)
    return S
```

At any time during loop, the S is sorted. The recursive implementation processes the list from right, while the iterative one is from left. We'll use 'tail-recursion' in section 1.3.5 to eliminate this difference. Chapter 3 is about insertion sort in detail, including performance analysis and optimization.

Exercise 1.6

1. Handle the out-of-bound case when insert, treat it as append.
2. Implement insert for array. When insert at position i, all elements after i need shift to the end.

delete

Symmetric to insert, there are two cases for deletion: (1). delete the element at a position *delAt i X*; (2) look up then delete the element of a given value *delete x X*. To delete the element at position i, we advance i steps to the target position, then by pass the element, and link the rest sub-list.

$$
\begin{array}{rcl}
delAt\ i\ [\,] & = & [\,] \\
delAt\ 0\ (x{:}xs) & = & xs \\
delAt\ i\ (x{:}xs) & = & x : delAt\ (i-1)\ xs
\end{array}
\tag{1.13}
$$

It is bound to $O(i)$ time as we need advance i steps to delete. Below is the iterative implementation.

```
function Del-At(i, X)
    S ← Cons(⊥, X)                          ▷ Sentinel node
    p ← S
    while i > 0 and X ≠ NIL do
        i ← i − 1
        p ← X
        X ← Rest(X)
    if X ≠ NIL then
        Rest(p) ← Rest(X)
    return Rest(S)
```

To simplify the implementation, we introduce a sentinel node S, it contains a special value $\perp$, and points to X. With S, we are save to cut-off any node in X even for the head. Finally, we return the list after S as the result, and discard S. For 'find and delete', there are two sub-cases: (1) find and delete the first occurrence of a value; (2) remove all the occurrences. The later is more generic (see the exercise).

$$
\begin{array}{rcl}
delete\ x\ [\,] & = & [\,] \\
delete\ x\ (y{:}ys) & = & \begin{cases} x = y : & ys \\ x \neq y : & y : delete\ x\ ys \end{cases}
\end{array}
\tag{1.14}
$$

Because we scan the lit to find the target element, the time is bound to $O(n)$, where n is the length. We use a sentinel node to simplify the iterative implementation too:

```
function Delete(x, X)
    S ← Cons(⊥, X)
    p ← X
    while X ≠ NIL and First(X) ≠ x do
        p ← X
        X ← Rest(X)
    if X ≠ NIL then
        Rest(p) ← Rest(X)
    return Rest(S)
```

Exercise 1.7

1. Implement the algorithm to find and delete all occurrences of a given value.

2. Design the delete algorithm for array, all elements after the delete position need shift to front.

concatenate

Append is a special case for concatenation. It adds only one element, while concatenation adds multiple. However, the performance would be quadratic if repeatedly append. Let $|xs| = n$, $|ys| = m$ be the lengths, we need advance to the tail of xs for m times, the performance is $O(n + (n + 1) + ... + (n + m)) = O(nm + m^2)$.

$$\begin{array}{rcl} xs \mathbin{+\!\!+} [\,] & = & xs \\ xs \mathbin{+\!\!+} (y{:}ys) & = & \mathit{append}\ xs\ y \mathbin{+\!\!+} ys \end{array}$$

While the 'cons' is fast (constant time), we can traverse to the tail of xs only once, then link to ys.

$$\begin{array}{rcl} [\,] \mathbin{+\!\!+} ys & = & ys \\ xs \mathbin{+\!\!+} [\,] & = & xs \\ (x{:}xs) \mathbin{+\!\!+} ys & = & x : (xs \mathbin{+\!\!+} ys) \end{array} \tag{1.15}$$

This improvement has the performance of $O(n)$. In imperative settings, we can implement concatenation in constant time with the tail reference variable (see exercise).

```
function Concat(X, Y)
    if X = NIL then
        return Y
    if Y = NIL then
        return X
    H ← X
    while Rest(X) ≠ NIL do
        X ← Rest(X)
    Rest(X) ← Y
    return H
```

1.3.5 sum and product

We often need to calculate the sum or product of a list. They have the same structure. We will introduce how to abstract them to higher order computation in section 1.6. For empty list, define the sum as 0, the product as 1.

$$\begin{array}{rcl} \mathit{sum}\ [\,] & = & 0 \\ \mathit{sum}\ (x{:}xs) & = & x + \mathit{sum}\ xs \end{array} \qquad \begin{array}{rcl} \mathit{product}\ [\,] & = & 1 \\ \mathit{product}(x{:}xs) & = & x \cdot \mathit{product}\ xs \end{array} \tag{1.16}$$

Both need traverse the list, hence the performance is $O(n)$, where n is the length. They compute from right to left. We can change to *accumulate* the result from left. For sum, accumulate from 0; while for product, accumulate from 1.

$$\begin{array}{rcl} \mathit{sum}'\ a\ [\,] & = & a \\ \mathit{sum}'\ a\ (x{:}xs) & = & \mathit{sum}\ (x + a)\ xs \end{array} \qquad \begin{array}{rcl} \mathit{prod}'\ a\ [\,] & = & a \\ \mathit{prod}'\ a\ (x{:}xs) & = & \mathit{prod}'\ (x \cdot a)\ xs \end{array} \tag{1.17}$$

Given a list, we call sum' with 0, and prod' with 1 as the accumulators:

$$\mathit{sum}\ xs = \mathit{sum}'\ 0\ xs \qquad \mathit{product}\ xs = \mathit{prod}'\ 1\ xs \tag{1.18}$$

Or in Curried form:

$$sum = sum' \ 0 \qquad product = prod' \ 1$$

Curried form was introduced by Schönfinkel (1889 - 1942) in 1924, then widely used by Haskell Curry from 1958. It is known as *Currying*[73]. For a function taking 2 parameters $f(x, y)$, when fix x with a value, it becomes a function of y: $g(y) = f(x, y)$ or $g = f \ x$. For multiple variables of $f(x, y, ..., z)$, we convert it to a series of Curried functions: $f, f \ x, f \ x \ y, ...$, each takes one parameter: $f(x, y, ..., z) = f(x)(y)...(z) = f \ x \ y \ ... \ z$.

The accumulated implementation computes from left to right, needn't book keeping any context, state, or intermediate result for recursion. All states are either passed as argument (for example a), or dropped (for example the previous element). We can further optimize such recursive calls to loops. Because the recursion happens at the tail of the function, we call them *tail recursion* (or 'tail call'), and the process to eliminate recursion as 'tail recursion optimization'[61]. It greatly improves the performance and avoid stack overflow due to deep recursions. In section 1.3.4 about insertion sort, the recursive implementation sorts elements form right. We also optimize it to tail call:

$$\begin{array}{rcl} sort' \ a \ [\,] & = & a \\ sort' \ a \ (x{:}xs) & = & sort' \ (insert \ x \ a) \ xs \end{array} \tag{1.19}$$

We pass $[\,]$ to start sorting (Curried form): $sort = sort' \ [\,]$. As a typical tail call example, consider how to compute b^n effectively? (problem 1.16 in[63].) A direct implementation repeatedly multiplies b for n times from 1, which is bound to $O(n)$ time:

```
function Pow(b, n)
    x ← 1
    loop n times
        x ← x · b
    return x
```

When compute b^8, after the first 2 loops, we get $x = b^2$. At this stage, we needn't multiply x with b to get b^3, but directly compute x^2, which gives b^4. If do this again, we get $(b^4)^2 = b^8$. We only need loop 3 times, but not 8 times. If $n = 2^m$ for some none negative integer m, we can compute b^n fast as below:

$$\begin{array}{rcl} b^1 & = & b \\ b^n & = & (b^{\frac{n}{2}})^2 \end{array}$$

We next extend this divide and conquer method to any none negative integer n: if $n = 0$, define $b^0 = 1$; if n is even, we halve n, to compute $b^{\frac{n}{2}}$. Then square it; if n is odd, since $n - 1$ is even, we recursively compute b^{n-1}, then multiply b atop it.

$$\begin{array}{rcl} b^0 & = & 1 \\ b^n & = & \begin{cases} 2|n : & (b^{\frac{n}{2}})^2 \\ \text{otherwise} : & b \cdot b^{n-1} \end{cases} \end{array} \tag{1.20}$$

However, the 2nd clause blocks us from turning it to tail recursive. Alternatively, we square the base number, and halve the exponent.

$$\begin{array}{rcl} b^0 & = & 1 \\ b^n & = & \begin{cases} 2|n : & (b^2)^{\frac{n}{2}} \\ \text{otherwise} : & b \cdot b^{n-1} \end{cases} \end{array} \tag{1.21}$$

With this change, we get a tail recursive function to compute $b^n = pow(b, n, 1)$.

$$\begin{array}{rcl} pow(b, 0, a) & = & a \\ pow(b, n, a) & = & \begin{cases} 2|n : & pow(b^2, \dfrac{n}{2}, a) \\ \text{otherwise} : & pow(b, n - 1, ab) \end{cases} \end{array} \tag{1.22}$$

This implementation is bound to $O(\lg n)$ time. We can improve it further. Represent n in binary format $n = (a_m a_{m-1} ... a_1 a_0)_2$. We need compute b^{2^i} if $a_i = 1$, similar to the Binomial heap (section 10.2, chapter 10) algorithm. Finally, we multiplying them together. For example, when compute b^{11}, as $11 = (1011)_2 = 2^3 + 2 + 1$, gives $b^{11} = b^{2^3} \times b^2 \times b$. We follow these steps:

1. compute b^1, which is b;
2. Square to b^2;
3. Square to b^{2^2};
4. Square to b^{2^3}.

Finally, multiply the result of step 1, 2, and 4 to get b^{11}.

$$
\begin{array}{rcl}
pow(b, 0, a) & = & a \\
pow(b, n, a) & = & \begin{cases} 2|n : & pow(b^2, \frac{n}{2}, a) \\ \text{otherwise} : & pow(b^2, \lfloor \frac{n}{2} \rfloor, ab) \end{cases}
\end{array}
\tag{1.23}
$$

This algorithm essentially shifts n to right 1 bit a time (divide n by 2). If the LSB (the least significant bit) is 0, n is even, squares the base and keeps the accumulator a unchanged. If the LSB is 1, n is odd, squares the base and accumulates it to a. When n is zero, we exhaust all bits, a is the final result. At any time, the updated base b', the shifted exponent n', and the accumulator a satisfy the invariant $b^n = a(b')^{n'}$.The previous implementation minus one for odd n, the improvement halves n every time. It exactly runs m rounds, where m is the number of bits. We leave the imperative implementation as exercise.

Back to the sum and product, the iterative implementation applies plus and multiply while traversing:

```
function Sum(X)
    s ← 0
    while X ≠ NIL do
        s ← s+ First(X)
        X ← Rest(X)
    return s

function Product(X)
    p ← 1
    while X ≠ NIL do
        p ← p · First(X)
        X ← Rest(X)
    return p
```

With product, we can define factorial of n as: $n! = product\ [1..n]$.

1.3.6 maximum and minimum

For a list of comparable elements (we can define order for any two elements), there is the maximum and minimum. max/min share the same structure:

$$
\begin{array}{rcl}
\min\ [x] & = & x \\
\min\ (x{:}xs) & = & \begin{cases} x < \min\ xs : & x \\ \text{otherwise} : & \min\ xs \end{cases}
\end{array}
\qquad
\begin{array}{rcl}
\max\ [x] & = & x \\
\max\ (x{:}xs) & = & \begin{cases} x > \max\ xs : & x \\ \text{otherwise} : & \max\ xs \end{cases}
\end{array}
\tag{1.24}
$$

Both process the list from right. We can change them to tail recursive. It also makes the computation 'on-line', that at any time, the accumulator is the min/max so far. Use *min* for example:

$$
\begin{array}{rcl}
\min' \ a \ [\,] & = & a \\
\min' \ a \ (x{:}xs) & = & \begin{cases} x < a : & \min' \ x \ xs \\ \text{否则} : & \min' \ a \ xs \end{cases}
\end{array} \tag{1.25}
$$

Different from $sum'/prod'$, we can't pass a fixed starting value to min'/max', unless $\pm\infty$ (Curried form):

$$
\min = \min' \ \infty \qquad \max = \max' \ -\infty
$$

We can pass the first element given min/max only takes none empty list:

$$
\min \ (x{:}xs) = \min' \ x \ xs \qquad \max \ (x{:}xs) = \max' \ x \ xs \tag{1.26}
$$

We can optimize the tail recursive implementation with loops. Use the Min for example.

```
function Min(X)
    m ← First(X)
    X ← Rest(X)
    while X ≠ NIL do
        if First(X) < m then
            m ← First(X)
        X ← Rest(X)
    return m
```

Alternatively, we can re-use the first element as the accumulator. Every time, we compare the first two elements, and drop one. Below is the example for *min*. *max* is symmetric.

$$
\begin{array}{rcl}
\min \ [x] & = & x \\
\min \ (x_1{:}x_2{:}xs) & = & \begin{cases} x_1 < x_2 : & \min \ (x_1{:}xs) \\ \text{otherwise} : & \min \ (x_2{:}xs) \end{cases}
\end{array} \tag{1.27}
$$

Exercise 1.8

1. Change *length* to tail recursive.
2. Change the insertion sort to tail recursive.
3. Compute b^n through the binary format of n.

1.4 Transform

In algebra, there are two types of transformation: one keeps the list structure, but only transforms the elements; the other alter the list structure, hence the result is not isomorphic. Particularly, we call the former *map*.

1.4.1 map and for-each

The first example converts a list of numbers to strings. Transform [3, 1, 2, 4, 5] to ["three", "one", "two", "four", "five"]

$$
\begin{array}{rcl}
toStr\ [\,] & = & [\,] \\
toStr\ (x{:}xs) & = & (str\ x) : toStr\ xs
\end{array}
\tag{1.28}
$$

For the second example, given a dictionary, which is a list of words grouped by their initials:

```
[[a, an, another, ... ],
 [bat, bath, bool, bus, ...],
 ...,
 [zero, zoo, ...]]
```

Next process a text (*Hamlet* for example), augment each word with the number of occurrence, like:

```
[[(a, 1041), (an, 432), (another, 802), ... ],
 [(bat, 5), (bath, 34), (bool, 11), (bus, 0), ...],
 ...,
 [(zero 12), (zoo, 0), ...]]
```

Now for every initial letter, which word does occur most? The answer is a list of words, that every one has the most occurrences in the group, like `[a, but, can, ...]`. We need a program that transforms **a list of groups of word-number pairs** into **a list of words**. First, define a function, which takes a list of word-number pairs, finds the word paired with the biggest number. Sort is overkill. We need a special max function *maxBy cmp xs*, where *cmp* is the generic compare function.

$$
\begin{array}{rcl}
maxBy\ cmp\ [x] & = & x \\
maxBy\ cmp\ (x_1{:}x_2{:}xs) & = & \begin{cases} cmp\ x_1\ x_2 : & maxBy\ cmp\ (x_2{:}xs) \\ \text{otherwise} : & maxBy\ cmp\ (x_1{:}xs) \end{cases}
\end{array}
\tag{1.29}
$$

For a pair $p = (a, b)$ we define two functions:

$$
\begin{cases} fst\ (a, b) = & a \\ snd\ (a, b) = & b \end{cases}
\tag{1.30}
$$

Then define a special compare function for word-count pairs:

$$
less\ p_1\ p_2 = snd\ p_1 < snd\ p_2
\tag{1.31}
$$

Then pass *less* to *maxBy* (in Curried form): $\max'' = maxBy\ less$. Finally, call max'' to process the list:

$$
\begin{array}{rcl}
solve\ [\,] & = & [\,] \\
solve\ (x{:}xs) & = & (fst\ (\max''\ \ x)) : solve\ xs
\end{array}
\tag{1.32}
$$

solve and *toStr* share the same structure for different problems. We abstract this common structure as *map*:

$$
\begin{array}{rcl}
map\ f\ [\,] & = & [\,] \\
map\ f\ (x{:}xs) & = & (f\ x) : map\ f\ xs
\end{array}
\tag{1.33}
$$

map takes a function f, applies it to every element to form a new list. A function that computes with other functions is called *high-order* function. Let the type of f is $A \to B$. It sends an element of A to the result of B, the type of map is:

$$map :: (A \to B) \to [A] \to [B] \tag{1.34}$$

Read as: map takes a function of $A \to B$, converts a list $[A]$ to another list $[B]$. We can define the above two examples with map as below (in Curried form):

$$toStr = map\ str \qquad solve = map\ (fst \circ \max'')$$

Where $f \circ g$ is function composition, i.e. first apply g then apply f. $(f \circ g)\ x = f(g(x))$, read as f after g. From the set theory point of view. Function $y = f(x)$ defines the map from x in set X to y in set Y:

$$Y = \{f(x) | x \in X\} \tag{1.35}$$

This type of set definition is called Zermelo-Frankel set abstraction (known as ZF expression)[72]. The difference is that the mapping is from a list (but not set) to another: $Y = [f(x) | x \leftarrow Y]$. There can be duplicated elements. For list, such ZF style expression is called *list comprehension*. It is a powerful tool. let us see how to realize the permutation algorithm for example. Extend from full-permutations[72] [94], we define a generic *perm* X r, that permutes r out of the total n elements in list X. There are total $P_n^r = \dfrac{n!}{(n-r)!}$ permutations.

$$perm\ X\ r = \begin{cases} |X| < r \text{ or } r = 0: & [[\ ]] \\ \text{otherwise}: & [x{:}ys \mid x \leftarrow X, ys \leftarrow perm\ (delete\ x\ X)\ (r-1)] \end{cases} \tag{1.36}$$

If pick zero element, or there are too few (less than r), the result is a list of empty[[ ]]; otherwise, for every x in X, we recursively pick $r-1$ out of the rest $n-1$ elements; then prepend x for each.

We use a sentinel node in the iterative MAP implementation.

```
function MAP(f, X)
    X' ← CONS(⊥, NIL)                          ▷ the sentinel
    p ← X'
    while X ≠ NIL do
        x ← FIRST(X)
        X ← REST(X)
        REST(p) ← CONS(f(x), NIL)
        p ← REST(p)
    return REST(X')                            ▷ discard the sentinel
```

For each

Sometimes we only need process the elements one by one without building the new list, for example, print every element:

```
function PRINT(X)
    while X ≠ NIL do
        print FIRST(X)
        X ← REST(X)
```

More generally, we pass a procedure P, then apply P to each element.

```
function FOR-EACH(P, X)
    while X ≠ NIL do
        P(FIRST(X))
        X ← REST(X)
```

For example, let us solve the "n-lights puzzle"[96] with map. There are n lights in a room, all are off. We execute the following n rounds:

1. Switch all lights on;
2. Switch lights of number 2, 4, 6, ... , that every other light is switched;
3. Switch every third lights, number 3, 6, 9, ... ;
4. ...

At the last round, only the n-th light is switched. How many lights are on in the end? We start with a brute-force solution. Represent the n lights as a list of 0/1 numbers (0: off, 1: on). Start from all zeros: [0, 0, ..., 0]. Label the light from 1 to n, then map them to (i, on/off) pairs:

$$lights = map\ (i \mapsto (i, 0))\ [1, 2, ..., n]$$

It binds each number to zero, i.e., a list of pairs: L = [(1, 0), (2, 0), ..., (n, 0)]. We operate this list of pairs n rounds. In the i-th round, for every pair (j, x), if $i|j$ (meaning $j \bmod i = 0$), then switch it on/off. As $1 - 0 = 1$ and $1 - 1 = 0$, we switch x to $1 - x$.

$$switch\ i\ (j, x)) = \begin{cases} j \bmod i = 0 : & (j, 1 - x) \\ \text{否则} : & (j, x) \end{cases} \tag{1.37}$$

Realize the i-th round of operation as $map\ (switch\ i)\ L$ (we use the Curried form of $switch$). Next, define a function $op()$, which performs mapping on L over and over for n rounds: $op\ [1, 2, ..., n]\ L$.

$$\begin{array}{rcl} op\ [\,]\ L & = & L \\ op\ (i{:}is)\ L & = & op\ is\ (map\ (switch\ i)\ L) \end{array} \tag{1.38}$$

Finally, sum the second value of each pair to get the answer.

$$solve\ n = sum\ (map\ snd\ (op\ [1, 2, ..., n]\ L)) \tag{1.39}$$

Below is the example Haskell implementation:

```haskell
solve = sum ∘ (map snd) ∘ proc  where
    lights = map (λi → (i, 0)) [1..n]
    proc n = operate [1..n] lights
    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)
```

Run this program from 1 to 100 lights, below are the answers (added line breaks):

```
[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,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,10]
```

They form a pattern: 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. Let's prove it:

Proof. Given n lights labeled from 1 to n, all light are off when start. The lights which are switched odd times are on finally. For every light i, we switch it at round j if j divides i ($j|i$). Only the lights which have odd number of factors are on in the end. The key point to solve this puzzle, is to find all the numbers that have odd number of factors. For any natural number n, let S be the set of all factors of n. Initialize S as $\varnothing$. If p is a factor of n, there must exist a natural number q such that $n = pq$. It means q is also a factor of n. We add 2 different factors to 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 square number. We can only add 1 factor to set S, which leads to odd number of factors. □

We have a fast solution by counting the square numbers under n.

$$solve(n) = \lfloor \sqrt{n} \rfloor \tag{1.40}$$

Below Haskell example program outputs the answer for 1, 2, ..., 100 lights:

```
map (floor ∘ sqrt) [1..100]
```

Map is abstract, does not limit to list, but applies to many complex algebraic structures. The next chapter explains how to map trees. We can apply mapping as long as we can traverse the structure, and the empty is defined.

1.4.2 reverse

It's a good exercise to reverse a singly linked-list with constant space. One must carefully manipulate the node reference, while there exists an easy method: (1) Write a purely recursive solution; (2) Change it to tail recursive; (3) Convert to imperative implementation. The purely recursive solution is direct:

$$\begin{array}{rcl} reverse[\,] & = & [\,] \\ reverse(x{:}xs) & = & append\ (reverse\ xs)\ x \end{array}$$

Next convert it to tail recursive. Use an accumulator to store the reversed part, start from an empty list: $reverse = reverse'\ [\,]$

$$\begin{array}{rcl} reverse'\ a\ [\,] & = & a \\ reverse'\ a\ (x{:}xs) & = & reverse'\ (x{:}a)\ xs \end{array} \tag{1.41}$$

Different from appending, cons (:) takes constant time. We repeatedly extract the head element, and prepend to the accumulator. It likes to store the elements in a stack, then pop them out. The overall performance is $O(n)$, where n is the length. Since tail call need not keep the context, we next convert it to iterative loops:

```
function REVERSE(X)
    A ← NIL
    while X ≠ NIL do
        A ← CONS(FIRST(X), A)
        X ← REST(X)
    return A
```

However, this implementation creates a new reversed list, but not reverses in-place. We change it further:

```
List<T> reverse(List<T> xs) {
  List<T> p, ys = null
  while xs ≠ null {
    p = xs
    xs = xs.next
    p.next = ys
    ys = p
  }
  return ys
}
```

Exercise 1.9

1. Given a number from 0 to 1 billion, write a program to 'read' it out. for example, output string 'one hundred and twenty three' for 123. What if there is decimal part?
2. Find the maximum v in a list of pairs $[(k, v)]$ in tail recursive way.

1.5 Sub-list

One can slice an array fast, but need linear time to traverse and extract sub-list. *take* extracts the first n elements, it is equivalent get a sub-list from 1 to n: $sublist\ 1\ n\ X$. *drop* discards the first n elements. It is equivalent to get a sub-list from right: $sublist\ (n+1)\ |X|\ X$, which is symmetric to *take*:

$$
\begin{array}{rcl}
take\ 0\ xs & = & [\ ] \\
take\ n\ [\ ] & = & [\ ] \\
take\ n\ (x{:}xs) & = & x : take\ (n-1)\ xs
\end{array}
\quad
\begin{array}{rcl}
drop\ 0\ xs & = & xs \\
drop\ n\ [\ ] & = & [\ ] \\
drop\ n\ (x{:}xs) & = & drop\ (n-1)\ xs
\end{array}
\tag{1.42}
$$

When $n > |X|$ or $n < 0$, it ends up with the empty list case. We leave the imperative implementation as exercise. We can extract the sub-list at any position for a given length:

$$sublist\ from\ cnt\ X = take\ cnt\ (drop\ (from - 1)\ X) \tag{1.43}$$

Or slice the list with left and right boundaries:

$$slice\ from\ to\ X = drop\ (from - 1)\ (take\ to\ X) \tag{1.44}$$

The range $[from, to]$ includes both ends. We can split the list at a position:

$$splitAt\ i\ X = (take\ i\ X, drop\ i\ X) \tag{1.45}$$

We can extend *take*/*drop* to keep taking or dropping as far as some condition is satisfied, Define *takeWhile*/*dropWhile*, that scan every element with a predicate p, stop when any element doesn't satisfy. They ignore the rest even if some elements satisfy p. We'll see this difference in the section of filtering.

$$
\begin{array}{rcl}
takeWhile\ p\ [\ ] & = & [\ ] \\
takeWhile\ p\ (x{:}xs) & = & \begin{cases} (p\ x) : & x : takeWhile\ p\ xs \\ \text{otherwise} : & [\ ] \end{cases}
\end{array}
\quad
\begin{array}{rcl}
dropWhile\ p\ [\ ] & = & [\ ] \\
dropWhile\ p\ (x{:}xs) & = & \begin{cases} (p\ x) : & dropWhile\ p\ xs \\ \text{otherwise} : & x{:}xs \end{cases}
\end{array}
\tag{1.46}
$$

Exercise 1.10

1. Define *sublist* and *slice* in Curried Form without X as parameter.

1.5.1 break and group

Break and group re-arrange a list into multiple sub-lists. They typically collect the sub-lists while traversing to achieve linear performance. We can consider *break*/*span* generic splitting. Not at a given position, *break*/*span* scans the list, extracts the longest prefix with a prediction p. There are two cases for p: pick the elements satisfied; or pick those not satisfied. The former is *span*, the later is *break*.

$$
\begin{array}{rcl}
span\ p\ [\,] & = & ([\,], [\,]) \\
span\ p\ (x{:}xs) & = & \begin{cases} (p\ x) : & (x{:}as, bs) \text{ where} : (as, bs) = span\ p\ xs \\ \text{otherwise} : & ([\,], x{:}xs) \end{cases}
\end{array} \tag{1.47}
$$

We define *break* by negating the predication: $break\ p = span\ (\neg p)$. *span* and *break* find the longest *prefix*. They stop immediately when the condition is broken and ignore the rest. Below is the iterative implementation of span:

function Span(p, X)
  $A \leftarrow X$
  $tail \leftarrow$ NIL
  while $X \neq$ NIL and p(First(X)) **do**
    $tail \leftarrow X$
    $X \leftarrow$ Rest(X)
  if $tail =$ NIL **then**
    return (NIL, X)
  Rest($tail$) $\leftarrow$ NIL
  return (A, X)

span and *break* cut the list into two parts, *group* divides list into multiple sub-lists. For example, group a long string into small units, each contains consecutive same character:

group “Mississippi” = [“M”, “i”, “ss”, “i”, “ss”,“i”, “pp”, “i”]

For another example, given a list of numbers: X = [15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2], divide it into small descending sub-lists:

$$group\ X = [[15, 9, 0], [12, 11, 7], [10, 5], [6], [13, 1], [4], [8, 3], [14, 2]]$$

Both are useful. We can build a Radix tree from string groups, support fast text search (chapter 6). We can implement the nature merge sort algorithm from number groups (chapter 13). Abstract the group condition as a relation $\sim$. It tests whether two consecutive elements x, y are ‘equivalent’: $x \sim y$. We scan the list, compare two elements each time. If they are equivalent, then we add both to a group; otherwise put them to two different ones.

$$
\begin{array}{rcl}
group\ \sim\ [\,] & = & [[\,]] \\
group\ \sim\ [x] & = & [[x]] \\
group\ \sim\ (x{:}y{:}xs) & = & \begin{cases} x \sim y : & (x{:}ys){:}yss, \text{where} : (ys{:}yss) = group\ \sim\ (y{:}xs) \\ \text{otherwise} : & [x]{:}ys{:}yss \end{cases}
\end{array} \tag{1.48}
$$

It is bound to $O(n)$ time, where n is the length. For the iterative implementation, if the list X isn’t empty, initialize the result groups as $[[x_1]]$. Scan from the second element, append it to the last group if the two consecutive elements are ‘equivalent’; otherwise we start a new group.

function Group($\sim, X$)

```
    if X = NIL then
        return [[ ]]
    x ← First(X)
    X ← Rest(X)
    g ← [x]
    G ← [g]
    while X ≠ NIL do
        y ← First(X)
        if x ~ y then
            g ← Append(g, y)
        else
            g ← [y]
            G ← Append(G, g)
        x ← y
        X ← Next(X)
    return G
```

However, the performance will downgrade to quadratic without the tail reference optimization for Append. We can change to Cons if don't care the order. We can define the above 2 examples with *group* as *group* (=) "Mississippi" and *group* ($\geq$) X. Alternatively, we can realize grouping with *span*. Given a predication p, *span* cuts the list into two parts: the longest sub-list satisfies p and the rest. We can repeatedly apply span to the rest till it becomes empty. However, span takes an unary function as the predication, while the group predication is a binary function. We solve it with Currying: pass and fix the first argument of the binary predication.

$$
\begin{array}{rcl}
group\ \sim\ [\ ] & = & [[\ ]] \\
group\ \sim\ (x{:}xs) & = & (x{:}as) : group\ \sim\ bs, \text{where} : (as, bs) = span\ (y \mapsto x \sim y)\ xs
\end{array}
\tag{1.49}
$$

Although the new function groups string correctly, it can't group numbers descending lists: *group* ($\geq$) X = [[15,9,0,12,11,7,10,5,6,13,1,4,8,3,14,2]]. When put the first number 15 as the left hand of $\geq$, it is the maximum , hence *span* ends with putting all numbers to *as* and leaves *bs* empty. It is not a defect, but the correct behavior. Because group is defined to put equivalent elements together. The equivalent relation ($\sim$) must satisfy three axioms: reflexive, transitive, and symmetric.

1. **Reflexive**. $x \sim x$;
2. **Transitive**. $x \sim y, y \sim z \Rightarrow x \sim z$;
3. **Symmetric**. $x \sim y \Leftrightarrow y \sim x$.

When group "Mississippi", the equal (=) operator satisfies the three axioms, and generates the correct result. However, the Curried ($\geq$) as an equivalent relationship, violets both reflexive and symmetric axioms, hence generates unexpected result. The second implementation via *span*, limits its use case to strict equivalence; while the first one does not. It only tests the predication for every two elements matches, which is weaker than equivalence.

Exercise 1.11

1. Change the *take*/*drop* implementation. When n is negative, returns [ ] for *take*, and the entire list for *drop*.
2. Implement the in-place imperative *take*/*drop*.

3. Implement the iterative 'take while' and 'drop while'.
4. Consider the below *span* implementation:

$$
\begin{array}{rcl}
span\ p\ [\,] & = & ([\,], [\,]) \\
span\ p\ (x{:}xs) & = & \begin{cases} (p\ x): & (x:as, bs), \text{where}: (as, bs) = span(p, xs) \\ \text{otherwise}: & (as, x:bs) \end{cases}
\end{array}
$$

What is the difference here?

1.6 Fold

Almost all list algorithms share the common structure. It is not by chance. The commonality is rooted from the recursive nature of list. We can abstract the list algorithm to a high level concept, fold[6], which is essentially the initial algebra of all list computations[99]. Observe *sum*, *product*, and *sort* for the common structure: the result for empty list is 0 for sum, 1 for product, and [ ] for sort; the binary operation that applies to the head and the recursive result. It's plus for sum, multiply for product, and ordered insertion for sort. We abstract the result for empty list as the *initial value* z (generic zero), the binary operation as $\oplus$. define:

$$
\begin{array}{rcl}
h\ \oplus\ z\ [\,] & = & z \\
h\ \oplus\ z\ (x{:}xs) & = & x \oplus (h\ \oplus\ z\ xs)
\end{array}
\tag{1.50}
$$

Feed a list $X = [x_1, x_2, ..., x_n]$ and expand:

$$
\begin{array}{rl}
 & h\ \oplus\ z\ [x_1, x_2, ..., x_n] \\
= & x_1 \oplus (h\ \oplus\ z\ [x_2, x_3, ..., x_n]) \\
= & x_1 \oplus (x_2 \oplus (h\ \oplus\ z\ [x_3, ..., x_n])) \\
 & ... \\
= & x_1 \oplus (x_2 \oplus (...(x_n \oplus (h\ \oplus\ z\ [\,]))...)) \\
= & x_1 \oplus (x_2 \oplus (...(x_n \oplus z)...))
\end{array}
$$

The parentheses are necessary, because the computation starts from the right-most $(x_n \oplus z)$, repeatedly folds left towards x_1. This is quite similar to a fold-fan in figure 1.3. Fold-fan is made of bamboo and paper. Multiple frames stack together with an axis at one end. The arc shape paper is fully expanded by these frames; We can close the fan by folding the paper. It ends up as a stick.

Figure 1.3: Fold fan

[6]also known as reduce

Consider the fold-fan as a list of bamboo frames. The binary operation is to fold a frame to the top of the stack (initialized empty). To fold the fan, start from one end, repeatedly apply the binary operation, till all the frames are stacked. The sum and product algorithms do the same thing essentially.

$$
\begin{array}{ll}
\mathit{sum}\ [1,2,3,4,5] & = 1 + (2 + (3 + (4 + 5))) \\
 & = 1 + (2 + (3 + 9)) \\
 & = 1 + (2 + 12) \\
 & = 1 + 14 \\
 & = 15
\end{array}
\qquad
\begin{array}{ll}
\mathit{product}\ [1,2,3,4,5] & = 1 \times (2 \times (3 \times (4 \times 5))) \\
 & = 1 \times (2 \times (3 \times 20)) \\
 & = 1 \times (2 \times 60) \\
 & = 1 \times 120 \\
 & = 120
\end{array}
$$

We name this kind of processes *fold*. Particularly, since the computation is from right, we denote it as $foldr$:

$$
\begin{array}{rcl}
foldr\ f\ z\ [\,] & = & z \\
foldr\ f\ z\ (x{:}xs) & = & f\ x\ (foldr\ f\ z\ xs)
\end{array}
\tag{1.51}
$$

Define sum and product with $foldr$ as below:

$$
\begin{array}{ll}
\sum_{i=1}^{n} x_i & = x_1 + (x_2 + (x_3 + ... + (x_{n-1} + x_n))...) \\
 & = foldr\ (+)\ 0\ [x_1, x_2, ..., x_n]
\end{array}
\tag{1.52}
$$

$$
\begin{array}{ll}
\prod_{i=1}^{n} x_i & = x_1 \times (x_2 \times (x_3 \times ... + (x_{n-1} \times x_n))...) \\
 & = foldr\ (\times)\ 1\ [x_1, x_2, ..., x_n]
\end{array}
\tag{1.53}
$$

Or in Curried form: $\mathit{sum} = foldr\ (+)\ 0$, $\mathit{product} = foldr\ (\times)\ 1$, for insertion-sort, it is: $\mathit{sort} = foldr\ \mathit{insert}\ [\,]$.

Convert $foldr$ to tail recursive. It generates the result from left. denote it as $foldl$:

$$
\begin{array}{rcl}
foldl\ f\ z\ [\,] & = & z \\
foldl\ f\ z\ (x{:}xs) & = & foldl\ f\ (f\ zx)\ xs
\end{array}
\tag{1.54}
$$

Use *sum* for example, we can see how the computation is expanded from left to right:

$$
\begin{array}{rl}
 & foldl\ (+)\ 0\ [1,2,3,4,5] \\
= & foldl\ (+)\ (0+1)\ [2,3,4,5] \\
= & foldl\ (+)\ (0+1+2)\ [3,4,5] \\
= & foldl\ (+)\ (0+1+2+3)\ [4,5] \\
= & foldl\ (+)\ (0+1+2+3+4)\ [5] \\
= & foldl\ (+)\ (0+1+2+3+4+5)\ [\,] \\
= & 0+1+2+3+4+5
\end{array}
$$

The evaluation of $f(z,x)$ is delayed in every step (the lazy evaluation). Otherwise, they will be evaluated in sequence of $[1,3,6,10,15]$ in each call. Generally, we can expand $foldl$ as (infix notation):

$$
foldl\ (\oplus)\ z\ [x_1, x_2, ..., x_n] = z \oplus x_1 \oplus x_2 \oplus ... \oplus x_n
\tag{1.55}
$$

$foldl$ is tail recursive. We can convert it to loops, called Reduce.

```
function Reduce(f, z, X)
    while X ≠ NIL do
        z ← f(z, First(X) )
        X ← Rest(X)
    return z
```

Both $foldr$ and $foldl$ have their own suitable use cases. They are not necessarily exchangeable. For example, some container only allows to add element to one end (like stack). We can define a function *fromList* to build such a container from a list (in Curried form):

$$fromList = foldr\ add\ \varnothing$$

Where $\varnothing$ is the empty container. The singly linked-list is such a container. It performs well (constant time) when add element to the head, but need linear time when append to tail. $foldr$ is a natural choice when duplicate a list while keeping the order. But $foldl$ will generate a reversed list. As a workaround, we first reverse the list, then reduce it:

```
function Reduce-Right(f, z, X)
    return Reduce(f, z, Reverse(X))
```

One may prefer $foldl$ as it is tail recursive, fits for both functional and imperative settings as an online algorithm. However, $foldr$ plays a critical role when handling infinite list (modeled as stream) with lazy evaluation. For example, below program wraps every natural number to a singleton list, and returns the first 10:

$$\begin{array}{l} take\ 10\ (foldr\ (x\ xs \mapsto [x]{:}xs)\ [\ ]\ [1, 2, ...]) \\ \Rightarrow [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]] \end{array}$$

It does not work with $foldl$ or the evaluation never ends. We use a unified notation $fold$ when both fold left and right work. We also use $fold_l$ and $fold_r$ to indicate direction doesn't matter. Although this chapter is about list, the fold concept is generic, can apply to other algebraic structures. We can fold a tree (2.6 in[99]), a queue, and many other things as long as the following 2 things are defined: (1) empty (for example the empty tree); (2) decomposed recursive structure (like to decompose tree into sub-trees and key). People abstract them further with concepts like foldable, monoid, and traversable.

For example, we implement the n-lights puzzle with $fold$ and map. In the brute-force solution, we create a list of pairs. Each pair (i, s) has a light number i, and on/off state s. Every round j, we switch the i-th light when the $j|i$. Define this process with $fold$:

$$foldr\ step\ [(1, 0), (2, 0), ..., (n, 0)]\ [1, 2, ..., n]$$

All lights are off at the beginning. We fold the list of rounds 1 to n. Function *step* takes two parameters: the round number i, and the list of pairs: $step\ i\ L\ = map\ (switch\ i)\ L$. The result of $foldr$ is the pairs of light number and final on/off state, we next extract the state out through *map*, and count the number with *sum*:

$$sum\ (map\ snd\ (foldr\ step\ [(1, 0), (2, 0), ..., (n, 0)]\ [1, 2, ..., n])) \tag{1.56}$$

What if we $fold$ a list of lists with "$+\!\!+$" (section 1.3.4)? It concatenates them to a list, just like *sum* to numbers.

$$concat = fold_r\ (+\!\!+)\ [\ ] \tag{1.57}$$

For example: $concat\ [[1], [2, 3, 4], [5, 6, 7, 8, 9]] \Rightarrow [1, 2, 3, 4, 5, 6, 7, 8, 9]$.

Exercise 1.12

1. To define insertion-sort with $foldr$, we design the insert function as $insert\ x\ X$, and sort as $sort = foldr\ insert\ [\ ]$. The type for $foldr$ is:

   $$foldr :: (A \to B \to B) \to B \to [A] \to B$$

   Where its first parameter f has the type of $A \to B \to B$, the initial value z has the type B. It folds a list of A, and builds the result of B. How to define the insertion-sort with $foldl$? What is the type of $foldl$?

2. What's the performance of *concat*?
3. Design a linear time *concat* algorithm
4. Define *map* in *foldr*

1.7 Search and filter

Search and filter are generic concepts for a wide range of things. For list, it often takes linear time to scan and find the result. First consider how to test if x is in list X? We compare every element with x, until either they are equal, or reach to the end:

$$
\begin{array}{rcl}
a \in [\,] & = & False \\
a \in (b : bs) & = & \begin{cases} b = a : & True \\ b \neq a : & a \in bs \end{cases}
\end{array}
\tag{1.58}
$$

The existence check is also called *elem*. The performance is $O(n)$ where n is the length. We can not improve it to $O(\lg n)$ with binary search directly even for ordered list. This is because list does not support constant time random access (chapter 3).

Let's extend *elem*. In the n-lights puzzle, we use a list of pairs $[(k, v)]$. Every pair contains a key and a value. Such list is called 'associate list' (abbrev. assoc list). We can lookup the value with a key.

$$
\begin{array}{rcl}
lookup\ x\ [\,] & = & Nothing \\
lookup\ x\ ((k, v){:}kvs) & = & \begin{cases} k = x : & Just\ (k, v) \\ k \neq x : & lookup\ x\ kvs \end{cases}
\end{array}
\tag{1.59}
$$

Different from *elem*, we want to find the corresponding value besides the existence of key x. However, it is not guaranteed the value always exists. We use the algebraic type class 'Maybe'. A type of **Maybe** A has two kinds of value. It may be some a in A or nothing. Denoted as $Just\ a$ and $Nothing$ respectively. This is a way to deal with null reference[7] (4.2.2 in[99]).

We can make *lookup* generic, to find the element that satisfies a given predicate:

$$
\begin{array}{rcl}
find\ p\ [\,] & = & Nothing \\
find\ p\ (x{:}xs) & = & \begin{cases} (p\ x) : & Just\ x \\ \text{otherwise} : & find\ p\ xs \end{cases}
\end{array}
\tag{1.60}
$$

Although there can be multiple elements satisfy p, the *find* function picks the first. We can expand it to find all elements. It is called *filter* as shown in figure 1.4. Define (ZF expression): $filter\ p\ X = [x | x \leftarrow X, p\ x]$.

Figure 1.4: Input: $[x_1, x_2, ..., x_n]$, Output: $[x'_1, x'_2, ..., x'_m]$. and $\forall x'_i \Rightarrow p(x'_i)$.

Different from *find*, *filter* returns empty list instead of *Nothing* when no element satisfies the predicate.

$$
\begin{array}{rcl}
filter\ p\ [\,] & = & [\,] \\
filter\ p\ (x{:}xs) & = & \begin{cases} (p\ x) : & x : filter\ p\ xs \\ \text{otherwise} : & filter\ p\ xs \end{cases}
\end{array}
\tag{1.61}
$$

[7]Similar to `Optional<A>` in some environments.

This definition builds the result from right. For iterative implementation, the performance will drop to $O(n^2)$ if build the result with APPEND. If change to CONS, then the order is reversed. We can further reverse it back in linear time (see the exercise).

```
function FILTER(p, X)
    X' ← NIL
    while X ≠ NIL do
        if p(FIRST(X)) then
            X' ← APPEND(X', FIRST(X))                ▷ Linear time
        L ← REST(X)
```

The nature to build result from right reminds us $foldr$. Define f to test an element against the predicate, and prepend it to the result: $f\ p\ x\ as\ = \textbf{if}\ p\ x\ \textbf{then}\ x{:}as\ \textbf{else}\ as$. Use its Curried form to define $filter$:

$$filter\ p = foldr\ (x\ as \mapsto f\ p\ x\ as)\ [\,] \tag{1.62}$$

We can further simplify it (called η-conversion[73]) as:

$$filter\ p = foldr\ (f\ p)\ [\,] \tag{1.63}$$

Filter is a generic concept not only limit to list. We can apply a predicate to any traversable structure to extract things.

Match is to find a pattern from some structure. Even if limit to list and string, there are still too many things to cover (chapter 14). The very basic problem is to test whether list as exits in bs. There are two special cases: to test if as is prefix or suffix of bs. The *span* function actually finds the longest prefix under a given predicate. Similarly, we can compare each element between as and bs. Define $as \subseteq bs$ if as is prefix of bs:

$$\begin{array}{rcl}
[\,] \subseteq bs & = & True \\
(a{:}as) \subseteq [\,] & = & False \\
(a{:}as) \subseteq (b{:}bs) & = & \begin{cases} a \neq b: & False \\ a = b: & as \subseteq bs \end{cases}
\end{array} \tag{1.64}$$

Prefix testing takes linear time to scan the two lists. However, we can not do suffix testing in this way because it is expensive to align the right ends and scan backwards. This is different from array. Alternatively, we can reverse both lists in linear time, convert the problem to prefix testing:

$$A \supseteq B = reverse(A) \subseteq reverse(B) \tag{1.65}$$

With $\subseteq$, we can test if a list is the sub-list of another one (infix testing). Define empty is infix of any list, we repeatedly apply prefix testing while traverse B:

$$\begin{array}{rcl}
\textit{infix?}\ (a{:}as)\ [\,] & = & False \\
\textit{infix?}\ A\ B & = & \begin{cases} A \subseteq B: & True \\ \text{otherwise}: & \textit{infix?}\ A\ B' \end{cases}
\end{array} \tag{1.66}$$

Below is the iterative implementation:

```
function IS-INFIX(A, B)
    if A = NIL then
        return TRUE
    n ← |A|
    while B ≠ NIL and n ≤ |B| do
```

    if $A \subseteq B$ **then**
        return TRUE
    $B \leftarrow$ REST(B)
return FALSE

Because prefix testing runs in linear time, and is called in every loop. This implementation is bound to $O(nm)$ time, where m, n are the length of the two lists. Symmetrically, we can enumerate all suffixes of B, and test if A is prefix of any:

$$infix?\ A\ B = \exists S \in suffixes\ B, A \subseteq S \tag{1.67}$$

Below example Haskell program implement infix testing with list comprehension:

```
isInfixOf a b = (not ∘ null) [s | s ← tails b, a `isPrefixOf` s]
```

Where `isPrefixOf` does the prefixing testing, `tails` generates all suffixes of a given list (exercise of this section).

Exercise 1.13

1. Implement the linear time existence testing algorithm.
2. Implement the iterative lookup algorithm.
3. Implement the linear time filter algorithm through *reverse*.
4. Implement the iterative prefix testing algorithm.
5. Enumerate all suffixes of a list.

1.8 zip and unzip

The assoc list is a light weighted dictionary (map) for small data. It is easier than tree or heap based dictionary with the overhead of linear time lookup performance. In the 'n-lights' puzzle, we build the assoc list as: $map\ (i \mapsto (i, 0))\ [1, 2, ..., n]$. We define a *zip* function:

$$\begin{array}{rcl} zip\ as\ [\,] & = & [\,] \\ zip\ [\,]\ bs & = & [\,] \\ zip\ (a{:}as)\ (b{:}bs) & = & (a, b) : zip\ as\ bs \end{array} \tag{1.68}$$

This implementation works when the two lists have different lengths. The result has the same length as the shorter one. We can even zip infinite lists (under lazy evaluation), for example[8]: $zip\ [0, 0, ...]\ [1, 2, ..., n]$. For a list of words, we can index it as: *zip* [1, 2, ...] [a, an, another, ...]. *zip* builds the result from right. We can define it with $foldr$. It is bound to $O(m)$ time, where m is the length of the shorter list. When implement the iterative *zip*, the performance will drop to quadratic if using APPEND, we can use CONS then reverse the result. However, this method can't handle two infinite lists. In imperative settings, we can reuse A to hold the zip result (treat as transform every element to a pair).

function ZIP(A, B)
    $C \leftarrow$ NIL
    while $A \neq$ NIL and $B \neq$ NIL **do**
        $C \leftarrow$ APPEND(C, (FIRST(A), FIRST(B)))   ▷ Linear time
        $A \leftarrow$ REST(A)
        $B \leftarrow$ REST(B)
    return C

[8]Or $zip\ (repeat\ 0)\ [1..n]$, where $repeat\ x = x : repeat\ x$.

We can extend to *zip* multiple lists. Some programming environments provide, `zip`, `zip3`, `zip4`, ... Sometimes, we want to apply a binary function to combine elements, but not just form a pair. For example, given a list of unit prices $[1.00, 0.80, 10.05, ...]$ for fruits: apple, orange, banana, ... and a list of quantities, like $[3, 1, 0, ...]$, meaning, buy 3 apples, 1 orange, 0 banana, ... Below program generates the payment list:

$$
\begin{array}{rcl}
pays\ us\ [\,] & = & [\,] \\
pays\ [\,]\ qs & = & [\,] \\
pays\ (u{:}us)\ (q{:}qs) & = & uq : pays\ us\ qs
\end{array}
$$

It has the same structure as *zip* except using multiply but not 'cons'. We can abstract the binary function as f:

$$
\begin{array}{rcl}
zipWith\ f\ as\ [\,] & = & [\,] \\
zipWith\ f\ [\,]\ bs & = & [\,] \\
zipWith\ f\ (a{:}as)\ (b{:}bs) & = & (f\ a\ b) : zipWith\ f\ as\ bs
\end{array}
\tag{1.69}
$$

For example, we can define the inner-product (or dot-product)[98] as: $A{\cdot}B = sum\ (zipWith\ (\cdot)\ A\ B)$ or define the infinite Fibonacci sequence with lazy evaluation:

$$
F = 0 : 1 : zipWith\ (+)\ F\ F' \tag{1.70}
$$

Let F be the infinite Fibonacci numbers, starts from 0 and 1. F' drops the head. From the third number, every Fibonacci number is the sum of the corresponding numbers from F and F' at the same position. Below example program takes the first 15 Fibonacci numbers:

```
fib = 0 : 1 : zipWith (+) fib (tail fib)

take 15 fib
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]
```

unzip is the inverse of *zip*. It converts a list of pairs to two separated lists. Define it with *foldr* in Curried form:

$$
unzip = foldr\ ((a, b)\ (as, bs) \mapsto (a{:}as, b{:}bs))\ ([\,], [\,]) \tag{1.71}
$$

For the fruits example, given the unit price as an assoc list: U = [(apple, 1.00), (orange, 0.80), (banana, 10.05), ...], the purchased quantity is also an assoc list: Q = [(apple, 3), (orange, 1), (banana, 0), ...]. We extract the unit prices and the quantities, then compute their inner-product:

$$
pay = sum\ (zipWith\ (\cdot)\ snd(unzip\ U)\ snd(unzip\ Q)) \tag{1.72}
$$

zip and *unzip* are generic. We can expand to *zip* two trees, where the nodes contain paired elements from both. When traverse a collection of elements, we can also use the generic *zip* and *unzip* to track the path. This is a method to mimic the 'parent' reference in imperative implementation (last chapter of[10]).

List is fundamental to build more complex data structures and algorithms particularly in functional settings. We introduced elementary algorithms to construct, access, update, and transform list; how to search, filter data, and compute with list. Although most programming environments provide pre-defined tools and libraries to support list, we should not simply treat them as black-boxes. Rabhi and Lapalme introduce many functional algorithms about list[72]. Haskell library provides detailed documentation about basic list algorithms. Bird gives good examples of folding[1], and introduces about the *fold fusion law*.

Exercise 1.14

1. Design the iota (I) operator for list, below are the use cases:
   - $iota(..., n) = [1, 2, 3, ..., n]$;
   - $iota(m, n) = [m, m + 1, m + 2, ..., n]$, where $m \leq n$;
   - $iota(m, m + a, ..., n) = [m, m + a, m + 2a, ..., n]$;
   - $iota(m, m, ...) = repeat(m) = [m, m, m, ...]$;
   - $iota(m, ...) = [m, m + 1, m + 2, ...]$.
2. Implement the linear time imperative zip.
3. Define zip with $foldr$.
4. For the fruits example, suppose the quantity assoc list only contains the items without zero quantity. i.e. instead of

   $$Q = [(apple, 3), (banana, 0), (orange, 1), ...]$$

   but

   $$Q = [(apple, 3), (orange, 1), ...]$$

   Write a program to calculate the total payment.
5. Implement $lastAt$ with zip.
6. Write a program to remove the duplicated elements in a list while maintain the original order. For imperative implementation, the elements should be removed in-place. What is the complexity? How to simplify it with additional data structure?
7. List can represent decimal non-negative integer. For example 1024 as list is $4 \to 2 \to 0 \to 1$. Generally, $n = d_m ... d_2 d_1$ can be represented as $d_1 \to d_2 \to ... \to d_m$. Given two numbers a, b in list form. Realize arithmetic operations such as add and subtraction.
8. In imperative settings, a circular linked-list is corrupted, that some node points back to previous one, as shown in figure 1.5. When traverse, it falls into infinite loops. Design an algorithm to detect if a list is circular. On top of that, improve it to find the node where loop starts (the node being pointed by two precedents).

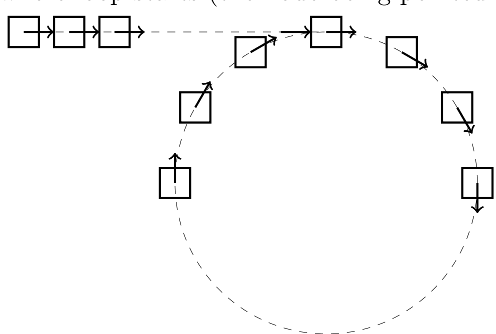


Figure 1.5: A circular linked-list

Chapter 2

Binary Search Tree

2.1 Introduction

Array and list are typically considered the basic data structures. However, we'll see they are not necessarily easy to implement in chapter 12. Upon imperative settings, array is the most elementary data structures. It is possible to implement linked-list using arrays (Equation 3.4). While in functional settings, linked-list acts as the building blocks to create array and other data structures.

We start from Binary Search Trees as the first data structure. Let us see an interesting programming problem given by Bentley in *Programming Pearls*[2]. It is about to count the number of words in text. Here is an example solution:

```
void wordcount(Input in) {
    bst<string, int> map;
    while string w = read(in) {
        map[w] = if map[w] == null then 1 else map[w] + 1
    }
    for var (w, c) in map {
        print(w, ":", c)
    }
}
```

We can run it to count the words in a text file:

```
$ cat bbe.txt | wordcount > wc.txt
```

The map is a binary search tree. Here we use the word as the key, and its occurrence number as the value. This program runs fast, which reflects the power of binary search tree. Before dive into it, let us first see the more generic tree, the binary tree. A binary tree can be defined recursively. It is

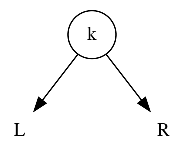
- either empty;
- or contains 3 parts: the element, and two sub-trees called left and right children.

Figure 2.1 shows an example of binary tree.

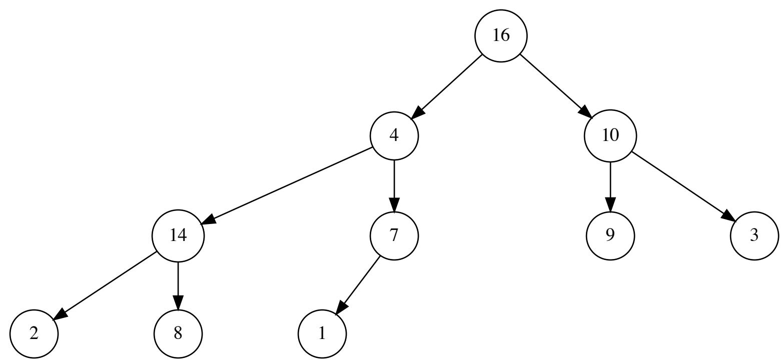
A binary search tree is a special binary tree that its elements are comparable[1], and satisfies the following constraints:

- For any node, all the keys in its left sub-tree are less than the key in this node;

[1] It is abstract ordering, not limit to magnitude, but like precedence, subset of etc. the 'less than' ($<$) is abstract in this chapter.



(a) Binary tree structure



(b) A binary tree

Figure 2.1: Binary tree concept and an example.

- the key in this node is less than any key in its right sub-tree.

Figure 2.2 shows an example of binary search tree. Comparing with Figure 2.1, we can see the differences in keys ordering. To highlight the elements in binary search tree is comparable, we call it as *key*, and name the augmented satellite data as *value*.

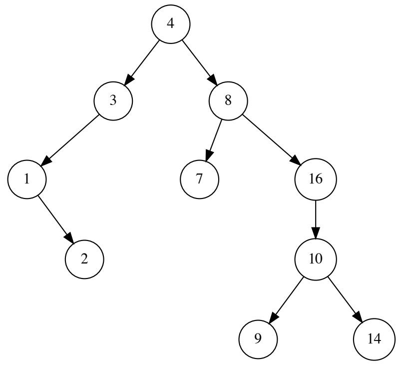


Figure 2.2: A Binary Search Tree

2.2 Data Layout

Based on the recursive definition of binary search tree, we can design the data layout as shown in figure 2.3. A node stores the key as a field, it can also store augmented data (known as satellite data). The next two fields are pointers to the left and right sub-trees. To make it easy for backtracking, it can also store a parent field pointed to its ancestor node.

For illustration purpose, we'll skip the augmented data. The appendix of this chapter includes an example definition. In functional settings, it is seldom to use pointers for backtracking. Typically, there is no such need, because the algorithm is usually top-down recursive. Below is the example functional definition:

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

2.3 Insertion

When insert a key k (or along with a value) to binary search tree T, we need ensure the key ordering property is always hold:

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

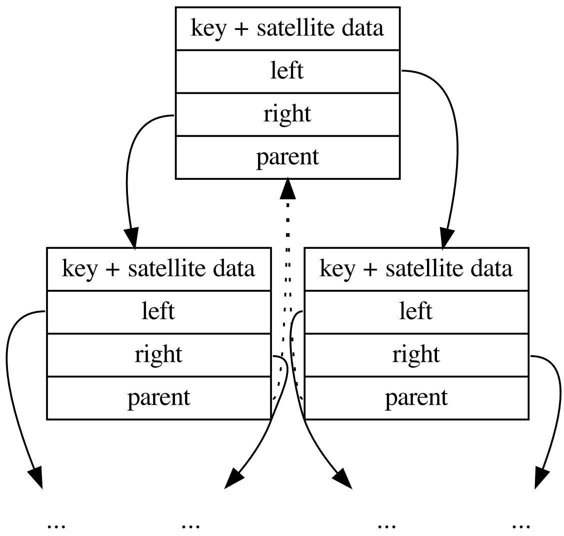


Figure 2.3: Node layout with parent field.

There is an exceptional case that k is equal to the key of root. It means k already exists in the tree. We can overwrite it, or append data, or do nothing. We'll skip such case handling. This algorithm is simple and straightforward. We can define it as a recursive function:

$$
\begin{array}{rcl}
insert(\varnothing, k) & = & Node(\varnothing, k, \varnothing) \\
insert(Node(T_l, k', T_r), k) & = & \begin{cases} k < k' : & Node(insert(T_l, k), k', T_r) \\ otherwise : & Node(T_l, k', insert(T_r, k)) \end{cases}
\end{array} \tag{2.1}
$$

For the none empty node, T_l denotes the left sub-tree, T_r denotes the right sub-tree, and k' is the key. The function $Node(l, k, r)$ creates a node from two sub-trees and a key. $\varnothing$ means empty (also known as NIL. This symbol was invented by mathematician André Weil for null set. It came from the Norwegian alphabet). Below is the corresponding example program in Haskell for insertion.

```
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 example program utilizes the *pattern matching* features. The appendix of this chapter provides another example without using this feature. Insertion can also be implemented without recursion. Here is a pure iterative algorithm:

```
function INSERT(T, k)
    root ← T
    x ← CREATE-LEAF(k)
    parent ← NIL
    while T ≠ NIL do
        parent ← T
        if k < KEY(T) then
            T ← LEFT(T)
        else
            T ← RIGHT(T)
```

```
    PARENT(x) ← parent
    if parent = NIL then                         ▷ tree T is empty
        return x
    else if k < KEY(parent) then
        LEFT(parent) ← x
    else
        RIGHT(parent) ← x
    return root

function CREATE-LEAF(k)
    x ← EMPTY-NODE
    KEY(x) ← k
    LEFT(x) ← NIL
    RIGHT(x) ← NIL
    PARENT(x) ← NIL
    return x
```

While a bit more complex than the functional one, the iterative implementation runs faster, and it is capable to process very deep tree.

2.4 Traverse

Traverse is to visit every element one by one. There are 3 different ways to walk through a binary tree: (1) pre-order tree walk, (2) in-order tree walk, (3) and post-order tree walk. They are named to highlight the order of visiting key before/after sub-trees.

- pre-order: **key** - left - right;
- in-order: left - **key** - right;
- post-order: left - right - **key**.

Each 'visit' operation is recursive, for example in pre-order traverse, when visit the left sub-tree, we recursively traverse it if it is not empty. For the tree shown in figure 2.2, the corresponding visiting orders are as below:

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

It is not by accident that the in-order traverse lists the elements one by one increasingly. The definition of the binary search tree ensures it is always true. We leave the proof as an exercise. Specifically, the in-order traverse algorithm is defined as:

- If the tree is empty, stop and return;
- Otherwise, in-order traverse the left sub-tree; then visit the key; finally in-order traverse the right sub-tree.

We can further define a generic *map* to apply any given function f to every element in the tree along the in-order traverse. The result is a new tree mapped by f.

$$\begin{array}{rcl} map(f, \varnothing) & = & \varnothing \\ map(f, Node(T_l, k, T_r)) & = & Node(map(f, T_l), f(k), map(f, T_r)) \end{array} \tag{2.2}$$

If we only need manipulate keys but not to transform the tree, we can implement this algorithm imperatively.

```
function In-Order-Traverse(T, f)
    if T ≠ NIL then
        In-Order-Traverse(Left(T, f))
        f(Key(T))
        In-Order-Traverse(Right(T, f))
```

Leverage in-order traverse, we can change the *map* function to convert a binary search tree to a sorted list. Instead building the tree in recursive case, we concatenate the result to a list:

$$\begin{array}{rcl} toList(\varnothing) & = & [\,] \\ toList(Node(T_l, k, T_r)) & = & toList(T_l) \mathbin{+\!\!+} [k] \mathbin{+\!\!+} toList(T_r) \end{array} \tag{2.3}$$

We can develop a method to sort a list of elements: first build a binary search tree from the list, then turn it back to list through in-order traversing. This method is called as 'tree sort'. For a given list $X = [x_1, x_2, x_3, ..., x_n]$.

$$sort(X) = toList(fromList(X)) \tag{2.4}$$

And we can write it in point-free form[8].

$$sort = toList \circ fromList$$

Where function $fromList$ repeatedly inserts elements from a list to a tree. It can be defined to recursively process the list.

$$\begin{array}{rcl} fromList([\,]) & = & \varnothing \\ fromList(X) & = & insert(fromList(X'), x_1) \end{array}$$

When the list is empty, the result is an empty tree; otherwise, it inserts the first element x_1 to the tree, then recursively inserts the rests $X' = [x_2, x_3, ..., x_n]$. By using list folding[7] (see appendix A.6), we can also define $fromList$ as the following:

$$fromList(X) = \text{fold}_l(insert, \varnothing, X) \tag{2.5}$$

We can also rewrite it in Curried form[9] (also known as partial application) such as to omit parameter X:

$$fromList = \text{fold}_l\ \ insert\ \varnothing$$

Exercise 2.1

1. Given the in-order and pre-order traverse results, re-construct the tree, and output the post-order traverse result. For example:
   - Pre-order: 1, 2, 4, 3, 5, 6;
   - In-order: 4, 2, 1, 5, 3, 6;
   - Post-order: ?
2. Write a program to re-construct the binary tree from the pre-order and in-order traverse lists.
3. For binary search tree, prove that the in-order traverse always visits elements in increase order
4. Consider the performance of tree sort algorithm, what is its complexity for n elements?

2.5 Query

Because the elements stored in binary search tree is well ordered and organized recursively. It supports varies of search effectively. This is one of the reasons people name it as binary search tree. There are mainly three types of querying: (1) look up a key; (2) find the minimum or maximum element; (3) given any node, find its predecessor or successor.

2.5.1 Look up

Because binary search tree is recursive and all elements satisfy the ordering property, we can look up a key k top-down from the root as the following:

- If the tree is empty, terminate. The key does not exist;
- Compare k with the key of root, if equal, we are done. The key is stored in the root;
- If k is less than the key of root, then recursively look up the left sub-tree;
- Otherwise, look up the right sub-tree.

We can define the recursive *lookup* function for this algorithm as below.

$$
\begin{array}{rcl}
lookup(\varnothing, x) & = & \varnothing \\
lookup(Node(T_l, k, T_r), x) & = & \begin{cases} k = x : & T \\ x < k : & lookup(T_l, x) \\ otherwise : & lookup(T_r, x) \end{cases}
\end{array}
\tag{2.6}
$$

This function returns the tree node being located or empty if not found. One may instead return the value that bound to the key. However, in search implementation, we need consider using $Maybe$ type (also known as `Optional<T>`) to handle the not found case, for example:

```
lookup Empty _ = Nothing
lookup t@(Node l k r) x | k == x = Just k
                        | x < k = lookup l x
                        | otherwise = lookup r x
```

If the binary search tree is well balanced, which means almost all branch nodes have both none empty sub-trees except for leaves. This is not the formal definition of balance. We'll define it in chapter 4. For a balanced tree of n elements, the algorithm takes $O(\lg n)$ time to look up a key. If the tree is poor balanced, the worst case is bound to $O(n)$ time. If denote the height of the tree as h, we can represent the performance of look up as $O(h)$.

We can also implement looking up purely iterative without recursion:

```
function SEARCH(T, x)
    while T ≠ NIL and KEY(T) ≠ x do
        if x < KEY(T) then
            T ← LEFT(T)
        else
            T ← RIGHT(T)
    return T
```

2.5.2 Minimum and maximum

From the definition, we know that less keys are always on the left. To locate the minimum element, we can keep traversing along the left sub-trees till reach to a node, where its left sub-tree is empty. In a symmetric way, keep traversing along the right sub-trees gives the maximum.

$$\begin{array}{rcl} min(Node(\varnothing, k, T_r)) & = & k \\ min(Node(T_l, k, T_r)) & = & min(T_l) \end{array} \tag{2.7}$$

$$\begin{array}{rcl} max(Node(T_l, k, \varnothing)) & = & k \\ max(Node(T_l, k, T_r)) & = & max(T_r) \end{array} \tag{2.8}$$

Both functions are bound to $O(h)$ time, where h is the height of the tree.

2.5.3 Successor and predecessor

When treat binary search tree as a generic container (a collection of elements), it is common to traverse it with a bi-directional iterator. Start from the minimum element, one can keep moving forward with the iterator towards the maximum, or go back and forth. Below example program prints elements in sorted order.

```
void printTree (Node<T> t) {
    for (var it = Iterator(t), it.hasNext(); it = it.next()) {
        print(it.get(), ", ");
    }
}
```

Such use cases demand us to design algorithm to find the successor or predecessor of any node. The successor of element x is defined as the smallest element y that satisfies $x < y$. If the node of x has none empty right sub-tree, then minimum element of the right sub-tree is the successor. As shown in figure 2.4, to find the successor of 8, we search the minimum element in its right sub-tree, which is 9. If the right sub-tree of node x is empty, we need back-track along the parent field till the closest ancestor whose left sub-tree is also an ancestor of x. In figure 2.4, since node 2 does not have right sub-tree, we go up to its parent of node 1. However, node 1 does not have left sub-tree, we need go up again, hence reach to node 3. As the left sub-tree of node 3 is also an ancestor of node 2, node 3 is the successor of node 2.

If we finally reach to the root when back-track along the parent, but still can not find an ancestor on the right, then the node does not have a successor. Below algorithm finds the successor of a given node x:

function SUCC(x)
    if RIGHT(x) $\neq$ NIL **then**
        return MIN(RIGHT(x))
    else
        $p \leftarrow$ PARENT(x)
        while $p \neq$ NIL and $x =$ RIGHT(p) **do**
            $x \leftarrow p$
            $p \leftarrow$ PARENT(p)
        return p

This algorithm returns NIL when x does not has successor. The predecessor finding algorithm is symmetric:

function PRED(x)
    if LEFT(x) $\neq$ NIL **then**

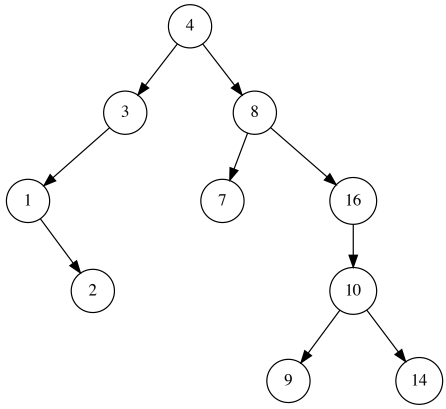


Figure 2.4: The successor of 8, is the minimum one in its right sub-tree, 9; In order to find the successor of 2, we go up to its parent 1, then 3.

        return MAX(LEFT(x))
    else
        $p \leftarrow$ PARENT(x)
        while $p \neq$ NIL and $x =$ LEFT(p) **do**
            $x \leftarrow p$
            $p \leftarrow$ PARENT(p)
        return p

It seems hard to find the purely functional solution, because there is no pointer like field linking to the parent node[2]. 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[?].

Our original purpose to develop *succ* and *pred* functions is 'to traverse all the elements' as a generic container. However, in functional settings, we typically traverse the tree in-order through *map*. We'll meet similar situations in the rest of this book. A problems valid in imperative settings may not be necessarily meaningful in functional settings. For example, to delete an element from red-black tree[5].

Exercise 2.2

1. Use PRED and SUCC to write an iterator to traverse the binary search tree as a generic container. What's the time complexity to traverse a tree of n elements?
2. One can traverse elements inside a range $[a, b]$ for example:
   `for_each (m.lower_bound(12), m.upper_bound(26), f);`
   Write an equivalent functional program for binary search tree.

2.6 Deletion

We need special consideration when delete an element from the binary search tree. This is because we must keep the ordering property, that for any node, all keys in left sub-tree

[2]There is `ref` in ML and OCaml, we limit to the purely functional settings.

are less than the key of this node, and they are all less than any keys in right sub tree. Blindly deleting a node may break this constraint.

To delete a node x from a binary search tree[6].

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

The simplicity comes from the fact that, for the node to be deleted, if the right sub-tree is not empty, then the minimum element is some node in it. It can't have two none empty children, and end up in the trivial case. Therefore, the node can be directly splice out from the tree.

Figure 2.5, 2.6, and 2.7 illustrate different cases for deletion.

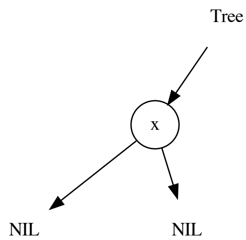
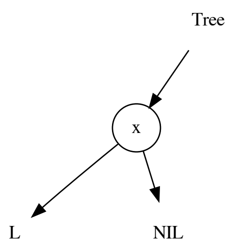
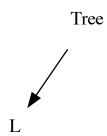


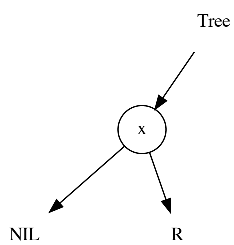
Figure 2.5: x can be spliced out.



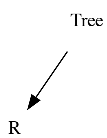
(a) Before delete x.



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

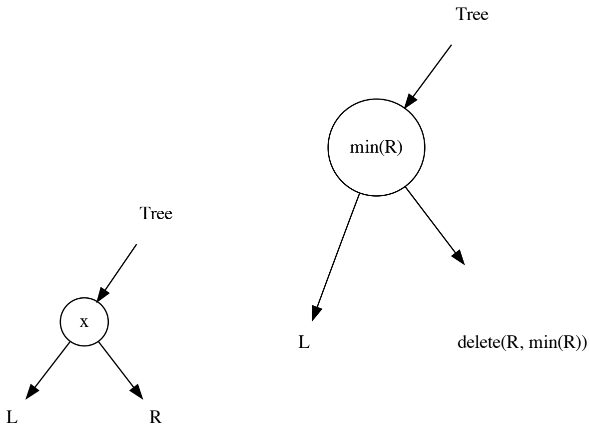


(c) Before delete x.



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

Figure 2.6: Delete a node with only one none empty sub-tree.



(a) Before delete x.

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

Figure 2.7: Delete a node with two none empty sub-trees.

Based on this idea, we can define the *delete* algorithm as below:

$$
\begin{array}{rcl}
delete(\varnothing, x) & = & \varnothing \\
delete(Node(T_l, k, T_r), x) & = & \begin{cases} x < k : & Node(delete(T_l, x), k, T_r) \\ x > k : & Node(T_l, k, delete(T_r, x)) \\ x = k : & del(T_l, T_r) \end{cases}
\end{array}
\tag{2.9}
$$

Function *del* performs slicing, and mutually call *delete* recursively to cut off the minimum from the right sub-tree.

$$
\begin{array}{rcl}
del(\varnothing, T_r) & = & T_r \\
del(T_l, \varnothing) & = & T_l \\
del(T_l, T_r) & = & Node(T_l, y, delete(T_r, y))
\end{array}
\tag{2.10}
$$

Where $y = min(T_r)$ is the minimum element in the right sub-tree. Here is the corresponding example 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)
                      | otherwise = del l r
  where
    del Empty r = r
    del l Empty = l
    del l r = let k' = min r in Node l k' (delete r k')
```

This algorithm firstly looks up the node to be deleted, then executes the deletion. It takes $O(h)$ time where h is the height of the tree.

The imperative deletion algorithm needs set the parent properly in addition. The following one returns the root of the result tree.

function DELETE(T, x)
  $r \leftarrow T$
  $x' \leftarrow x$ ▷ save x

```
    p ← Parent(x)
    if Left(x) = NIL then
        x ← Right(x)
    else if Right(x) = NIL then
        x ← Left(x)
    else                                          ▷ neither children is empty
        y ← Min(Right(x))
        Key(x) ← Key(y)
        Copy other satellite data from y to x
        if Parent(y) ≠ x then                     ▷ y does not have left sub-tree
            Left(Parent(y)) ← Right(y)
        else                                      ▷ y is the root of the right sub-tree
            Right(x) ← Right(y)
        if Right(y) ≠ NIL then
            Parent(Right(y)) ← Parent(y)
        Remove y
        return r
    if x ≠ NIL then
        Parent(x) ← p
    if p = NIL then                               ▷ remove the root
        r ← x
    else
        if Left(p) = x' then
            Left(p) ← x
        else
            Right(p) ← x
    Remove x'
    return r
```

We assume the node to be deleted is not empty. This algorithm first records the root, creates copy reference to x, and its parent. If either sub-tree is empty, then we splice x out. Otherwise, the node has two none empty sub-trees. We first located the minimum node y in its right sub-tree, then replace the key of x with the one in y, copy the satellite data, and finally, splice y out. We also need handle the special case, that y is the root of the right sub-tree.

At last, we need reset the stored parent if x has only one none empty sub-tree. If the parent pointer we copied is empty, it means that we are deleting the root. In this case, we need return the new root. After the parent is set properly, we can safely remove x. The deletion algorithm is bound to $O(h)$ time, where h is the height of the tree.

Exercise 2.3

1. There is a symmetric deletion algorithm. When neither sub-tree is empty, we can replace the key by splicing the maximum node off the left sub-tree. Write a program to implement this solution.

2.7 Random build

All binary search tree algorithms we give so far are bound bound to $O(h)$ time. The height h of the tree impacts the performance. For a poor unbalanced tree, $O(h)$ tends to be $O(n)$. It leads to the worst case. While for well balanced tree, $O(h)$ close to $O(\lg n)$. We can gain good performance.

We'll see two well designed solutions to keep the tree balanced in chapter 4 and 5. There exists a simple method, to build the binary search tree randomly [4]. It decreases the possibility of giving a unbalanced binary tree. The idea is to randomly shuffle the elements before building the tree.

Exercise 2.4

1. Write a randomly building algorithm for binary search tree.

2.8 Map

We can use binary search tree to realize the map data structure (also known as associative data structure or dictionary). A finite map is a collection of key-value pairs. The keys are unique, that every key is mapped to a value. For keys of type K, values of type V, the map is $Map\ K\ V$ or `Map<K, V>`. For none empty map, it contains n mappings of $k_1 \mapsto v_1, k_2 \mapsto v_2, ..., k_n \mapsto v_n$. When use the binary search tree to implement map, we constrain K to be ordered set. Every node stores both key and value. We use the tree insert/update operation to bind a key to a value. Given a key k, we use the tree lookup to find the mapped value, or returns nothing when k does not exist. The red-black tree and AVL tree introduced in later chapters can also be used to implement map.

2.9 Appendix: Example programs

Definition of binary search tree node with parent field.

```
data Node<T> {
    T key
    Node<T> left
    Node<T> right
    Node<T> parent

    Node(T k) = Node(null, k, null)

    Node(Node<T> l, T k, Node<T> r) {
        left = l, key = k, right = r
        if (left ≠ null) then left.parent = this
        if (right ≠ null) then right.parent = this
    }
}
```

Example program of recursive insertion. It does not use pattern matching.

```
Node<T> insert (Node<T> t, T x) {
    if (t == null) {
        return Node(null, x, null)
    } else if (t.key < x) {
        return Node(insert(t.left, x), t.key, t.right)
    } else {
        return Node(t.left, t.key, insert(t.right, x))
    }
}
```

Example program to look up a key. Purely iterative without recursion.

```
Optional<Node<T>> lookup (Node<T> t, T x) {
    while (t ≠ null and t.key ≠ x) {
        if (x < t.key) {
            t = t.left
```

```
        } else {
            t = t.right
        }
    }
    return Optional.of(t);
}
```

Example iterative program to find the minimum of a tree.

```
Optional<Node<T>> min (Node<T> t) {
    while (t ≠ null and t.left ≠ null) {
        t = t.left
    }
    return Optional.of(t);
}
```

Example program to find the successor of a node.

```
Optional<Node<T>> succ (Node<T> x) {
    if (x == null) {
        return Optional.Nothing
    } else if (x.right ≠ null) {
        return min(x.right)
    } else {
        p = x.parent
        while (p ≠ null and x == p.right) {
            x = p
            p = p.parent
        }
        return Optional.of(p);
    }
}
```

Chapter 3

Insertion sort

3.1 Introduction

Insertion sort is a straightforward sort algorithm[1]. We give its preliminary definition for list in chapter 1. For a collection of comparable elements, we repeatedly pick one, insert them to a list and maintain the ordering. As every insertion takes linear time, its performance is bound to $O(n^2)$ where n is the number of elements. This performance is not as good as the divide and conqueror sort algorithms, like quick sort and merge sort. However, we can still find its application today. For example, a well tuned quick sort implementation falls back to insertion sort for small data set. The idea of insertion sort is similar to sort a deck of a poker cards([4] pp.15). The cards are shuffled. A player takes card one by one. At any time, all cards on hand are sorted. When draws a new card, the player inserts it in proper position according to the order of points as shown in figure 3.1.

Figure 3.1: Insert card 8 to a deck.

Based on this idea, we can implement insertion sort as below:

```
function Sort(A)
    S ← NIL
    for each a ∈ A do
        Insert(a, S)
    return S
```

We store the sorted result in a new array, alternatively, we can change it to in-place:

```
function Sort(A)
```

[1]We skip the 'Bubble sort' method

for $i \leftarrow 2$ to $|A|$ **do**
    ordered insert $A[i]$ to $A[1...(i-1)]$

Where the index i ranges from 1 to $n = |A|$. We start from 2, because the singleton sub-array $[A[1]]$ is ordered. When process the i-th element, all elements before i are sorted. We continuously insert elements till consuming all the unsorted ones, as shown in figure 3.2.

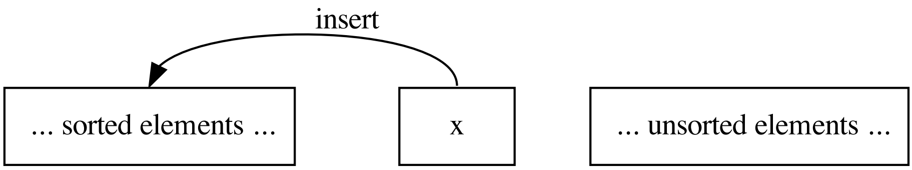


Figure 3.2: Continuously insert elements to the sorted part.

3.2 Insertion

In chapter 1, we give the ordered insertion algorithm for list. For array, we also scan it to locate the insert position either from left or right. Below algorithm is from right:

function SORT(A)
    for $i \leftarrow 2$ to $|A|$ **do** ▷ Insert $A[i]$ to $A[1...(i-1)]$
        $x \leftarrow A[i]$ ▷ Save $A[i]$ to x
        $j \leftarrow i - 1$
        while $j > 0$ and $x < A[j]$ **do**
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j - 1$
        $A[j+1] \leftarrow x$

It's expensive to insert at arbitrary position, as array stores elements continuously. When insert x at position i, we need shift all elements after i (i.e. $A[i+1], A[i+2], ...$) one cell to right. After free up the cell at i, we put x in, as shown in figure 3.3.

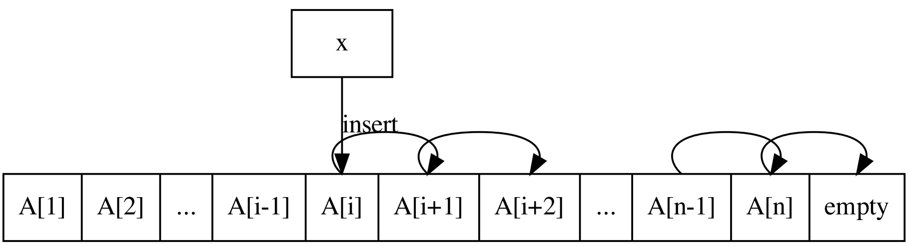


Figure 3.3: Insert x to A at i.

For the array of length n, suppose after comparing x to the first i elements, we located the position to insert. Then we shift the rest $n - i + 1$ elements, and put x in the i-th cell. Overall, we need traverse the whole array if scan from left. On the other hand, if scan from right to left, we examine $n - i + 1$ elements, and perform the same amount of shifts. We can also define a separated INSERT() function, and call it inside the loop. The insertion takes linear time no matter scans from left or right, hence the sort algorithm is bound to $O(n^2)$, where n is the number of elements.

Exercise 3.1

1. Implement the insert to scan from left to right.
2. Define the insert function, and call it from the sort algorithm.

3.3 Binary search

When insert a poker card, human does not scan, but takes a quick glance at the deck to locate the position. We can do this because the deck is sorted. Binary search is such a method that applies to ordered sequence.

```
function Sort(A)
    for i ← 2 to |A| do
        x ← A[i]
        p ← Binary-Search(x, A[1...(i − 1)])
        for j ← i down to p do
            A[j] ← A[j − 1]
        A[p] ← x
```

Binary search utilize the fact that the slice $A[1...(i-1)]$ is ordered. Suppose it is ascending without loss of generality (as we can define $\leq$ abstract). To find the position j that satisfies $A[j-1] \leq x \leq A[j]$, we compare x to the middle element $A[m]$, where $m = \lfloor \frac{i}{2} \rfloor$. If $x < A[m]$, we then recursively apply binary search to the first half; otherwise, we search the second half. As every time, we halve the elements, binary search takes $O(\lg i)$ time to locate the insert position.

```
function Binary-Search(x, A)
    l ← 1, u ← 1 + |A|
    while l < u do
        m ← ⌊(l + u)/2⌋
        if A[m] = x then
            return m                          ▷ Duplicated element
        else if A[m] < x then
            l ← m + 1
        else
            u ← m
    return l
```

The improved sort algorithm is still bound to $O(n^2)$. The one with scan takes $O(n^2)$ comparisons and $O(n^2)$ shifts; with binary search, it overall takes $O(n \lg n)$ comparisons and $O(n^2)$ shifts.

Exercise 3.2

1. Implement the recursive binary search.

3.4 List

With binary search, the search time improved to $O(n \lg n)$. However, as we need shift array cells when insert, the overall time is still bound to $O(n^2)$. On the other hand, when use list, the insert operation is constant time at a given node reference. In chapter 1, we define the insertion sort algorithm for list as below:

$$
\begin{array}{rcl}
sort(\varnothing) & = & \varnothing \\
sort(x : xs) & = & insert(x, sort(xs))
\end{array}
\tag{3.1}
$$

Or with $fold_l$ in Curried form:

$$
sort = fold_l(insert, \varnothing) \tag{3.2}
$$

However, the list *insert* algorithm still takes linear time, because we need scan to locate the insert position:

$$
\begin{array}{rcl}
insert(x,\ \varnothing) & = & [x] \\
insert(x,\ y : ys) & = & \begin{cases} x \leq y : & x : y : ys \\ otherwise : & y : insert(x, ys) \end{cases}
\end{array}
\tag{3.3}
$$

Instead of using node reference, we can also realize list through an additional index array. For every element $A[i]$, $Next[i]$ stores the index to the next element follows $A[i]$, i.e. $A[Next[i]]$ is the next element of $A[i]$. There are two special indexes: for the tail node $A[m]$, we define $Next[m] = -1$, indicating it points to NIL; we also define $Next[0]$ to index the head element. With the index array, we can implement the insertion algorithm as below:

```
function Insert(A, Next, i)
    j ← 0                                              ▷ Next[0] for head
    while Next[j] ≠ −1 and A[Next[j]] < A[i] do
        j ← Next[j]
    Next[i] ← Next[j]
    Next[j] ← i

function Sort(A)
    n ← |A|
    Next = [1, 2, ..., n, −1]                          ▷ n + 1 indexes
    for i ← 1 to n do
        Insert(A, Next, i)
    return Next
```

With list, although the insert operation changes to constant time, we need traverse the list to locate the position. It is still bound to $O(n^2)$ times comparison. Unlike array, list does not support random access, hence we can not use binary search to speed up.

Exercise 3.3

1. For the index array based list, we return the re-arranged index as result. Design an algorithm to re-order the original array A from the index $Next$.

3.5 Binary search tree

We drive into a corner. We want to improve both comparison and insertion at the same time, or will end up with $O(n^2)$ performance. For comparison, we need binary search to achieve $O(\lg n)$ time; on the other hand, we need change the data structure, because array can not support constant time insertion at a position. We introduce a powerful data structure in chapter 2, the binary search tree. It supports binary search from its definition by nature. At the same time, we can insert a new node in binary search tree fast at the given location.

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

Where INSERT-TREE() and TO-LIST() are defined in chapter 2. In average case, the performance of tree sort is bound to $O(n \lg n)$, where n is the number of elements. This is the lower limit of comparison based sort([?] pp.180-193). However, in the worst case, if the tree is poor balanced the performance drops to $O(n^2)$.

3.6 Summary

Insertion sort is often used as the first example of sorting. It is straightforward and easy to implement. However its performance is quadratic. Insertion sort does not only appear in textbooks, it has practical use case in the quick sort implementation. It is an engineering practice to fallback to insertion sort when the number of elements is small.

Chapter 4

Red-black tree

4.1 Introduction

As the example in chapter 2, we use the binary search tree as a dictionary to count the word occurrence in text. One may want to feed a address book to a binary search tree, and use it to look up the contact as below example program:

```
void addrBook(Input in) {
    bst<string, string> dict
    while (string name, string addr) = read(in) {
        dict[name] = addr
    }
    loop {
        string name = read(console)
        var addr = dict[name]
        if (addr == null) {
            print("not found")
        } else {
            print("address: ", addr)
        }
    }
}
```

Unlike the word counter program, this one performs poorly, especially when search names like Zara, Zed, Zulu, etc. This is because the address entries are typically listed in lexicographic order, i.e. the names are input in ascending order. If insert numbers 1, 2, 3, ..., n to a binary search tree, it ends up like in figure 4.1. It is an extremely unbalanced binary search tree. The *lookup*() is bound to $O(h)$ time for a tree with height h. When the tree is well balanced, the performance is $O(\lg n)$, where n is the number of elements in the tree. But in this extreme case, the performance downgrades to $O(n)$. It is equivalent to list scan.

Exercise 4.1

1. For a big address entry list in lexicographic order, one may want to speed up building the address book with two concurrent tasks: one reads from the head; while the other reads from the tail, till they meet at some middle point. What does the binary search tree look like? What if split the list into multiple sections to scale the concurrency?
2. Find more cases to exploit a binary search tree, for example in figure 4.2.

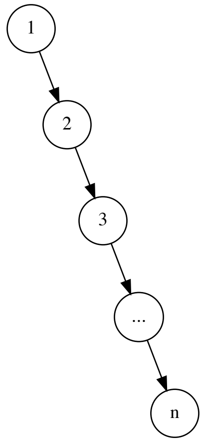


Figure 4.1: unbalanced tree

4.1.1 Balance

To avoid extremely unbalanced case, we can shuffle the input(12.4 in[4]), however, when the input is entered by user interactively, we can not randomize the sequence. People developed solutions to make the tree balanced. They mostly rely on the rotation operation. Rotation changes the tree structure while maintain the elements ordering. This chapter introduces the red-black tree, the widely used self-adjusting balanced binary search tree. Next chapter is about AVL tree, another self-balanced tree. Chapter 8 introduce the splay tree. It adjusts the tree in steps to make it balanced.

4.1.2 Tree rotation

Tree rotation transforms the tree structure while keeping the in-order traverse result unchanged. There are multiple binary search trees generate the same ordered element sequence. Figure 4.3 shows the tree rotation.

Tree rotation can be defined with pattern matching:

$$
\begin{array}{rcl}
rotate_l\ (a, x, (b, y, c)) & = & ((a, x, b), y, c)) \\
rotate_l\ T & = & T
\end{array}
\tag{4.1}
$$

and

$$
\begin{array}{rcl}
rotate_r\ ((a, x, b), y, c) & = & (a, x, (b, y, c)) \\
rotate_r\ T & = & T
\end{array}
\tag{4.2}
$$

The second row in each equation keeps the tree unchanged if the pattern does not match (for example, both sub-trees are empty). We can also implement tree rotation imperatively. We need re-assign sub-trees and parent node reference. When rotate, we pass both the root T, and the node x as parameters:

function LEFT-ROTATE(T, x)
  $p \leftarrow$ PARENT(x)
  $y \leftarrow$ RIGHT(x) ▷ assume $y \neq$ NIL
  $a \leftarrow$ LEFT(x)
  $b \leftarrow$ LEFT(y)
  $c \leftarrow$ RIGHT(y)
  REPLACE(x, y) ▷ replace node x with y

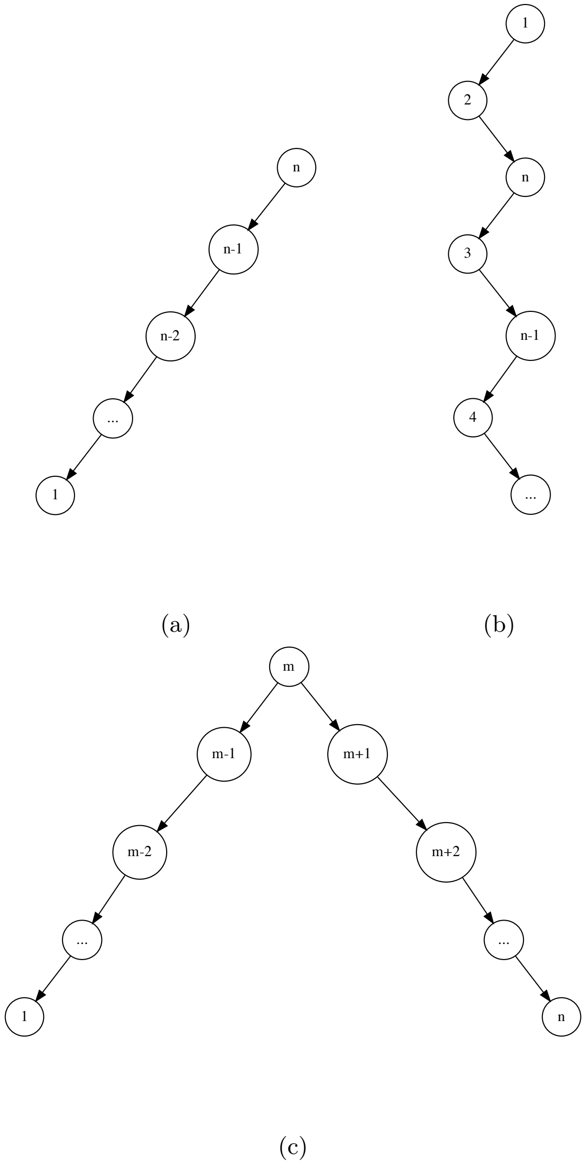


Figure 4.2: Unbalanced trees

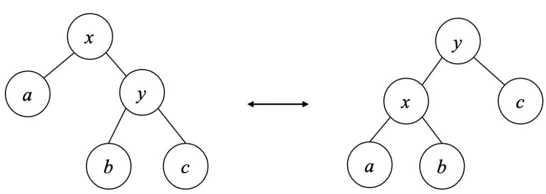


Figure 4.3: 'left rotate' and 'right rotate'.

```
    Set-Subtrees(x, a, b)          ▷ Set a, b as the sub-trees of x
    Set-Subtrees(y, x, c)          ▷ Set x, c as the sub-trees of y
    if p = NIL then                ▷ x was the root
        T ← y
    return T
```

The Right-Rotate is symmetric, we leave it as exercise. The Replace(x, y) uses node y to replace x:

```
function Replace(x, y)
    p ← Parent(x)
    if p = NIL then                ▷ x is the root
        if y ≠ NIL then Parent(y) ← NIL
    else if Left(p) = x then
        Set-Left(p, y)
    else
        Set-Right(p, y)
    Parent(x) ← NIL
```

Procedure Set-Subtrees(x, L, R) assigns L as the left, and R as the right sub-trees of x:

```
function Set-Subtrees(x, L, R)
    Set-Left(x, L)
    Set-Right(x, R)
```

It further calls Set-Left and Set-Right to set the two sub-trees:

```
function Set-Left(x, y)
    Left(x) ← y
    if y ≠ NIL then Parent(y) ← x

function Set-Right(x, y)
    Right(x) ← y
    if y ≠ NIL then Parent(y) ← x
```

We can see how pattern matching simplifies the tree rotation. Based on this idea, Okasaki developed the purely functional algorithm for red-black tree in 1995[13].

Exercise 4.2

1. Implement the Right-Rotate.

4.2 Definition

A red-black tree is a self-balancing binary search tree[14]. It is essentially equivalent to 2-3-4 tree[1]. By coloring the node red or black, and performing rotation, red-black tree provides an efficient way to keep the tree balanced. On top of the binary search tree definition, we label the node with a color. We say it is a red-black tree if the coloring satisfies the following 5 rules([4] pp273):

1. Every node is either red or black.

2. The root is black.

[1] Chapter 7, B-tree. For any 2-3-4 tree, there is at least one red-black tree has the same ordered data.

3. Every leaf (NIL) is black.

4. If a node is red, then both sub-trees are black.

5. For every node, all paths from it to descendant leaves contain the same number of black nodes.

Why do they keep the red-black tree balanced? The key point is that, the longest path from the root to leaf can not be as 2 times longer than the shortest path. Consider rule 4, there can not be any two adjacent red nodes. Therefore, the shortest path only contains black nodes. Any longer path must have red ones. In addition, rule 5 ensures all paths have the same number of black nodes. So as to the root. It eventually ensures any path is not 2 times longer than others[14]. Figure 4.4 gives an example of red-black tree.

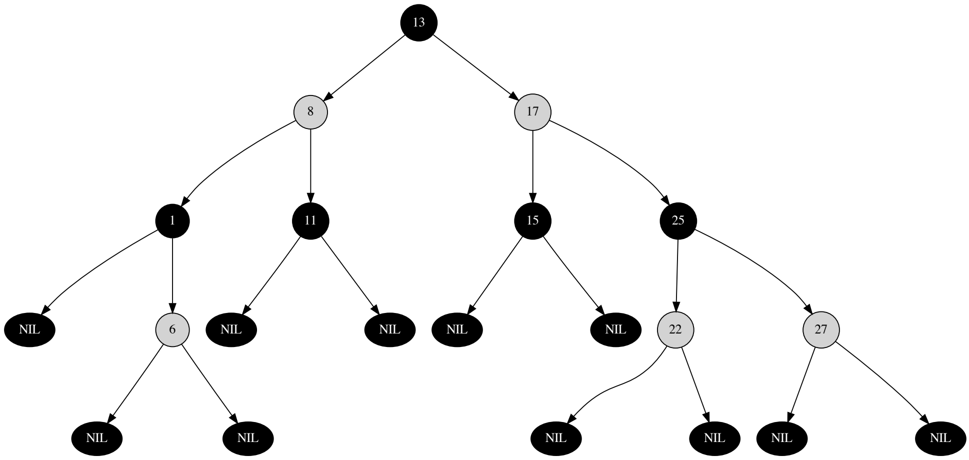


Figure 4.4: A red-black tree

As all NIL nodes are black, we can hide them as shown in figure 4.5. All operations including *lookup*, *min*/*max*, are same as the binary search tree. However, the *insert* and *delete* are special, as we need maintain the coloring rules.

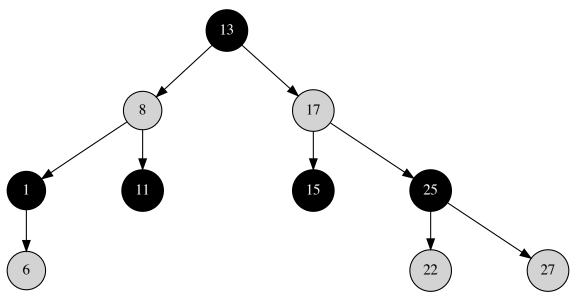


Figure 4.5: Hide the NIL nodes

Below example program adds the color field atop binary search tree definition:

```
data Color = R | B
data RBTree a = Empty
```

```
              | Node Color (RBTree a) a (RBTree a)
```

Exercise 4.3

1. Prove the height h of a red-black tree of n nodes is at most $2\lg(n+1)$

4.3 Insert

The *insert* algorithm for red-black tree has two steps. The first step is as same as the binary search tree. The tree may become unbalanced after that, we need fix it to resume the red-black tree coloring in the second step. When insert a new element, we always make it red. Unless the new node is the root, we won't break any coloring rules except for the 4-th. Because it may bring two adjacent red nodes. Okasaki finds there are 4 cases violate rule 4. All have two adjacent red nodes. They share a uniformed structure after fixing[13] as shown in figure 4.6.

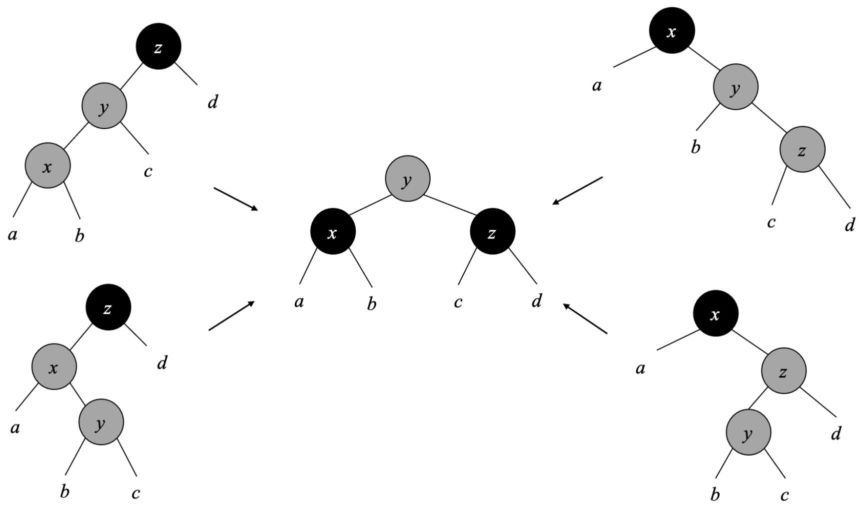


Figure 4.6: Fix 4 cases to the same structure.

All 4 transformations move the redness one level up. When perform bottom-up recursive fixing, it may color the root red. While rule 2 requires the root always be black. We need revert the root back to black finally. With pattern matching, we can define a *balance* function to fix the tree. Denote the color as $\mathcal{C}$ with values black $\mathcal{B}$, and red $\mathcal{R}$. A none empty node is in the form of $T = (\mathcal{C}, l, k, r)$, where l, r are the left and right sub-trees, k is the key.

$$
\begin{array}{rcl}
balance\ \mathcal{B}\ (\mathcal{R}, (\mathcal{R}, a, x, b), y, c)\ z\ d & = & (\mathcal{R}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, d)) \\
balance\ \mathcal{B}, (\mathcal{R}, a, x, (\mathcal{R}, b, y, c))\ z\ d & = & (\mathcal{R}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, d)) \\
balance\ \mathcal{B}\ a\ x\ (\mathcal{R}, b, y, (\mathcal{R}, c, z, d)) & = & (\mathcal{R}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, d)) \\
balance\ \mathcal{B}\ a\ x\ (\mathcal{R}, (\mathcal{R}, b, y, c), z, d) & = & (\mathcal{R}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, d)) \\
balance\ T & = & T
\end{array}
\tag{4.3}
$$

The last row says if the tree is not in any 4 patterns, then we leave it unchanged. We define the *insert* algorithm for red-black tree as below:

$$
insert\ T\ k = makeBlack\ (ins\ T\ k) \tag{4.4}
$$

where

$$
\begin{array}{rcl}
ins\ \varnothing\ k & = & (\mathcal{R}, \varnothing, k, \varnothing) \\
ins\ (\mathcal{C}, l, k', r)\ k & = & \begin{cases} k < k' : & balance\ \mathcal{C}\ (ins\ l\ k)\ k'\ r \\ k > k' : & balance\ \mathcal{C}\ l\ k'\ (ins\ r\ k) \end{cases}
\end{array}
\tag{4.5}
$$

If the tree is empty, we create a red node of k with two empty sub-trees; otherwise, let the sub-trees and the key be l, r, k', we compare k and k', then recursively insert k to a sub-tree. After that, we call *balance* to fix the coloring, then force the root to be black finally.

$$
makeBlack\ (\mathcal{C}, l, k, r) = (\mathcal{B}, l, k, r) \tag{4.6}
$$

Below is the corresponding example 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)
    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
```

We skip to handle the duplicated keys. If the key already exists, we can overwrite, drop, or store the values in a list ([4], pp269). Figure 4.7 shows two red-black trees built from sequence 11, 2, 14, 1, 7, 15, 5, 8, 4 and 1, 2, ..., 8. The second example demonstrates the tree is well balanced even for ordered input.

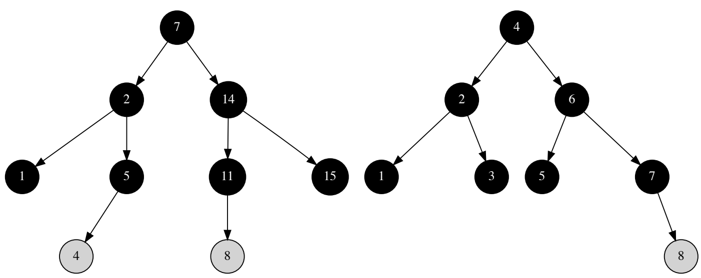


Figure 4.7: Red-black tree examples

The algorithm performs top-down recursive insertion and fixing. It is bound to $O(h)$ time, where h is the height of the tree. As the red-black tree coloring rules are maintained, h is the logarithm to the number of nodes n. The overall performance is $O(\lg n)$.

Exercise 4.4

1. Implement the *insert* algorithm without using pattern matching, but test the 4 cases separately.

4.4 Delete

Delete is more complex than insert. We can also use pattern matching and recursion to simplify the *delete* algorithm for red-black tree[2]. There are alternatives to mimic delete. Sometimes, we build the read-only tree, then use it for frequently looking up[5]. When delete, we mark the deleted node with a flag, and later rebuild the tree if such nodes exceeds 50%. Delete may also violate the red-black tree coloring rules. We use the same idea to apply fixing after delete. The coloring violation only happens when delete a black node according to rule 5. The black nodes along the path decreases by one, hence not all paths contain the same number of black nodes.

To resume the blackness, we introduce a special 'doubly-black' node([4], pp290). One such node is counted as 2 black nodes. When delete a black node x, we can move the blackness either up to its parent or down to one sub-tree. Let this node be y that accepts the blackness. If y was red, we turn it black; if y was already black, we make it 'doubly-black', denoted as $\mathcal{B}^2$. Below example program adds the 'doubly-black' support:

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

Because all empty leaves are black, when push the blackness down to a leaf, it becomes 'doubly-black' empty (`BBEmpty`, or bold $\boldsymbol{\varnothing}$). The first step is to perform the normal binary search tree delete; then if the cut off node is black, we shift the blackness, and fix the tree coloring.

$$delete = makeBlack \circ del \tag{4.7}$$

This definition is in Curried form. When delete the only element, the tree becomes empty. To cover this case, we modify *makeBlack* as below:

$$\begin{array}{rcl} makeBlack\ \varnothing & = & \varnothing \\ makeBlack\ (\mathcal{C}, l, k, r) & = & (\mathcal{B}, l, k, r) \end{array} \tag{4.8}$$

Where *del* accepts the tree and k to be deleted:

$$\begin{array}{rcl} del\ \varnothing\ k & = & \varnothing \\ del\ (\mathcal{C}, l, k', r)\ k & = & \begin{cases} k < k' : & fixB^2(\mathcal{C}, (del\ l\ k), k', r) \\ k > k' : & fixB^2(\mathcal{C}, l, k', (del\ r\ k)) \\ k = k' : & \begin{cases} l = \varnothing : & (\mathcal{C} = \mathcal{B} \mapsto shiftB\ r, r) \\ r = \varnothing : & (\mathcal{C} = \mathcal{B} \mapsto shiftB\ l, l) \\ else : & fixB^2(\mathcal{C}, l, k'', (del\ r\ k'')) \\ & \text{where } k'' = min(r) \end{cases} \end{cases} \end{array} \tag{4.9}$$

When the tree is empty, the result is $\varnothing$; otherwise, we compare the key k' in the tree with k. If $k < k'$, we recursively delete k from the left sub-tree; if $k > k'$ then delete from the right. Because the recursive result may contain doubly-black node, we need apply $fixB^2$ to fix it. When $k = k'$, we need splice it out. If either sub-tree is empty, we replace it with the other, then shift the blackness if the spliced node is black. This is represented with McCarthy form $(p \mapsto a, b)$, which is equivalent to '(if p then a else b)'. If neither sub-tree is empty, we cut the minimum element $k'' = min(r)$, and use k'' to replace k.

[2]Actually, the tree is rebuilt in purely functional setting, although the common part is reused. This feature is called 'persist'

To reserve the blackness, $shiftB$ makes a black node doubly-black, and forces it black for other cases. It flips doubly-black to normal black when applied twice.

$$
\begin{array}{rcl}
shiftB\ (\mathcal{B}, l, k, r) & = & (\mathcal{B}^2, l, k, r) \\
shiftB\ (\mathcal{C}, l, k, r) & = & (\mathcal{B}, l, k, r) \\
shiftB\ \varnothing & = & \boldsymbol{\varnothing} \\
shiftB\ \boldsymbol{\varnothing} & = & \varnothing
\end{array}
\tag{4.10}
$$

Below is the example program (except the doubly-black fixing part).

```
delete :: (Ord a) ⇒ RBTree a → a → RBTree a
delete t k = makeBlack $ del t k where
    del Empty _ = Empty
    del (Node color l k' r) k
        | k < k' = fixDB color (del l k) k' r
        | k > k' = fixDB color l k' (del r k)
        | isEmpty l = if color == B then shiftBlack r else r
        | isEmpty r = if color == B then shiftBlack l else l
        | otherwise = fixDB color l k'' (del r k'') where k''= min r
    makeBlack (Node _ l k r) = Node B l k r
    makeBlack _ = Empty

shiftBlack (Node B l k r) = Node BB l k r
shiftBlack (Node _ l k r) = Node B  l k r
shiftBlack Empty = BBEmpty
shiftBlack BBEmpty = Empty
```

The $fixB^2$ function eliminates the doubly-black node by rotation and re-coloring. The doubly-black node can be branch node or empty $\boldsymbol{\varnothing}$. There are three cases:

Case 1. *The sibling of the doubly-black node is black, and it has a red sub-tree.* We can fix this case with a rotation. There are 4 sub-cases, all can be transformed to a uniformed pattern, as shown in figure A.1.

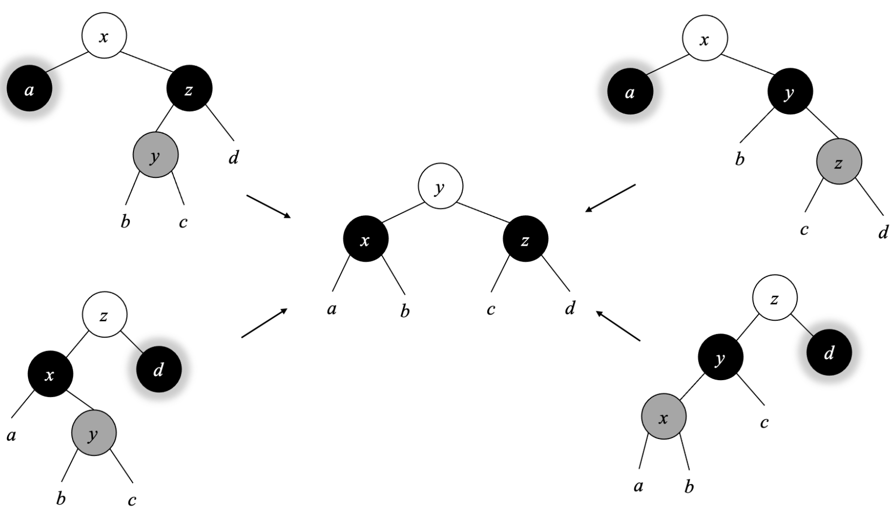


Figure 4.8: 4 sub-cases share the uniformed fixing pattern

The fixing for these 4 sub-cases can be realized with pattern matching.

$$
\begin{array}{rcl}
fixB^2\ \mathcal{C}\ a_{\mathcal{B}^2}\ x\ (\mathcal{B}, (\mathcal{R}, b, y, c), z, d) & = & (\mathcal{C}, (\mathcal{B}, shiftB(a), x, b), y, (\mathcal{B}, c, z, d)) \\
fixB^2\ \mathcal{C}\ a_{\mathcal{B}^2}\ x\ (\mathcal{B}, b, y, (\mathcal{R}, c, z, d)) & = & (\mathcal{C}, (\mathcal{B}, shiftB(a), x, b), y, (\mathcal{B}, c, z, d)) \\
fixB^2\ \mathcal{C}\ (\mathcal{B}, a, x, (\mathcal{R}, b, y, c))\ z\ d_{\mathcal{B}^2} & = & (\mathcal{C}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, shiftB(d))) \\
fixB^2\ \mathcal{C}\ (\mathcal{B}, (\mathcal{R}, a, x, b), y, c)\ z\ d_{\mathcal{B}^2} & = & (\mathcal{C}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, shiftB(d)))
\end{array}
\tag{4.11}
$$

Where $a_{\mathcal{B}^2}$ means node a is doubly-black, it can be branch or $\varnothing$.

Case 2. *The sibling of the doubly-black is red.* We can rotate the tree to turn it into case 1 or 3, as shown in figure A.2.

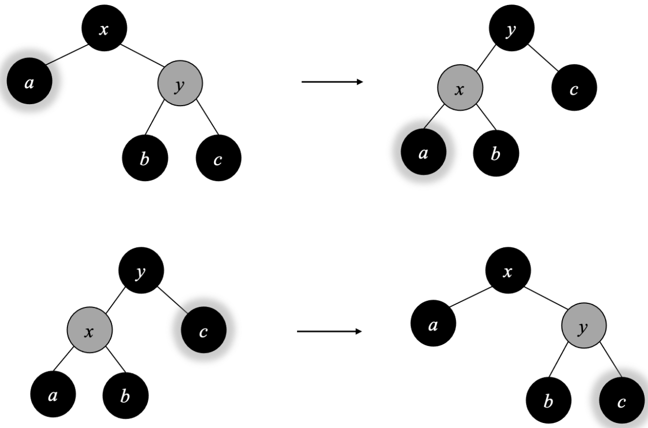


Figure 4.9: The sibling of the doubly-black is red.

We add this fixing as additional 2 rows in equation (4.11):

$$
\begin{array}{rcl}
 & \ldots & \\
fixB^2\ \mathcal{B}\ a_{\mathcal{B}^2}\ x\ (\mathcal{R}, b, y, c) & = & fixB^2\ \mathcal{B}\ (fixB^2\ \mathcal{R}\ a\ x\ b)\ y\ c \\
fixB^2\ \mathcal{B}\ (\mathcal{R}, a, x, b)\ y\ c_{\mathcal{B}^2} & = & fixB^2\ \mathcal{B}\ a\ x\ (fixB^2\ \mathcal{R}\ b\ y\ c)
\end{array}
\tag{4.12}
$$

Case 3. *The sibling of the doubly-black node, and its two sub-trees are all black.* In this case, we change the sibling to red, flip the doubly-black node to black, and propagate the doubly-blackness a level up to parent as shown in figure A.3.

There are two symmetric sub-cases. For the upper case, x was either red or black. x changes to black if it was red, otherwise changes to doubly-black; Same coloring changes to y in the lower case. We add this fixing to equation (4.12):

$$
\begin{array}{rcl}
 & \ldots & \\
fixB^2\ \mathcal{C}\ a_{\mathcal{B}^2}\ x\ (\mathcal{B}, b, y, c) & = & shiftB\ (\mathcal{C}, (shiftB\ a), x, (\mathcal{R}, b, y, c)) \\
fixB^2\ \mathcal{C}\ (\mathcal{B}, a, x, b)\ y\ c_{\mathcal{B}^2} & = & shiftB\ (\mathcal{C}, (\mathcal{R}, a, x, b), y, (shiftB\ c)) \\
fixB^2\ \mathcal{C}\ l\ k\ r & = & (\mathcal{C}, l, k, r)
\end{array}
\tag{4.13}
$$

If none of the patterns match, the last row keeps the node unchanged. The doubly-black fixing is recursive. It terminates in two ways: One is **Case 1**, the doubly-black node is eliminated. Otherwise the blackness may move up till the root. Finally the we force the root be black. Below example program puts all three cases together:

```
-- the sibling is black, and has a red sub-tree
fixDB color a@(Node BB _ _ _) x (Node B (Node R b y c) z d)
      = Node color (Node B (shiftBlack a) x b) y (Node B c z d)
```

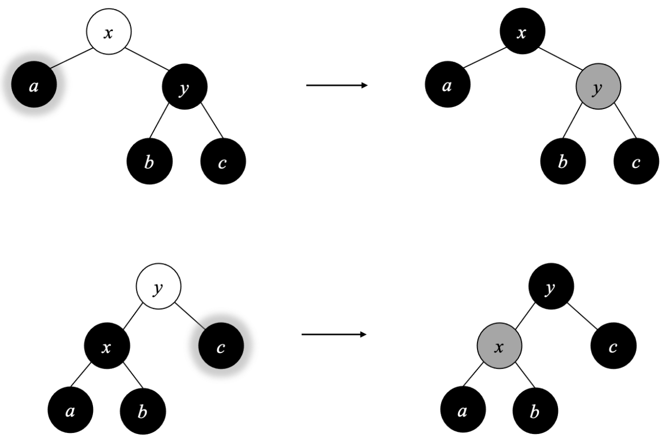


Figure 4.10: move the blackness up.

```
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 (shiftBlack 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 (shiftBlack 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 (shiftBlack 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, move the blackness up
fixDB color a@(Node BB _ _ _) x (Node B b y c)
      = shiftBlack (Node color (shiftBlack a) x (Node R b y c))
fixDB color BBEmpty x (Node B b y c)
      = shiftBlack (Node color Empty x (Node R b y c))
fixDB color (Node B a x b) y c@(Node BB _ _ _)
      = shiftBlack (Node color (Node R a x b) y (shiftBlack c))
fixDB color (Node B a x b) y BBEmpty
      = shiftBlack (Node color (Node R a x b) y Empty)
-- otherwise
fixDB color l k r = Node color l k r
```

The delete algorithm is bound to $O(h)$ time, where h is the height of the tree. As red-black tree maintains the balance, $h = O(\lg n)$ for n nodes.

Exercise 4.5

1. Implement the alternative delete algorithm: mark the node as deleted without actually removing it. When the marked nodes exceed 50%, re-build the tree.

4.5 Imperative red-black tree algorithm ⋆

We simplify the red-black tree implementation with pattern matching. In this section, we give the imperative algorithm for completeness. When insert, the first step is as same as the binary search tree, then as the second step, we fix the balance through tree rotations.

```
function Insert(T, k)
    root ← T
    x ← Create-Leaf(k)
    Color(x) ← RED
    p ← NIL
    while T ≠ NIL do
        p ← T
        if k < Key(T) then
            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)
```

We make the new node red, and then perform fixing before return. There are 3 basic cases, each one has a symmetric case, hence there are total 6 cases. Among them, we can merge two cases, because both have a red 'uncle' node. We change the parent and uncle to black, and set grand parent to red:

```
function Insert-Fix(T, x)
    while Parent(x) ≠ NIL and 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 on the right
                    x ← Parent(x)
                    T ← Left-Rotate(T, x)
                                                      ▷ Case 3, x is on the left
                Color(Parent(x)) ← BLACK
                Color(Grand-Parent(x)) ← RED
                T ← Right-Rotate(T, Grand-Parent(x))
            else
                if x = Left(Parent(x)) then           ▷ Case 2, Symmetric
                    x ← Parent(x)
                    T ← Right-Rotate(T, x)
                                                      ▷ Case 3, Symmetric
                Color(Parent(x)) ← BLACK
                Color(Grand-Parent(x)) ← RED
```

$T \leftarrow$ LEFT-ROTATE(T, GRAND-PARENT(x))
COLOR(T) $\leftarrow$ BLACK
return T

This algorithm takes $O(\lg n)$ time to insert a key, where n is the number of nodes. Compare to the *balance* function defined previously, they have different logic. Even input the same sequence of keys, they build different red-black trees. Figure 4.11 shows the result when input the same sequence of keys to the imperative algorithm. We can see the difference from figure 4.7. There is a bit performance overhead in the pattern matching algorithm. Okasaki discussed the difference in detail in[13].

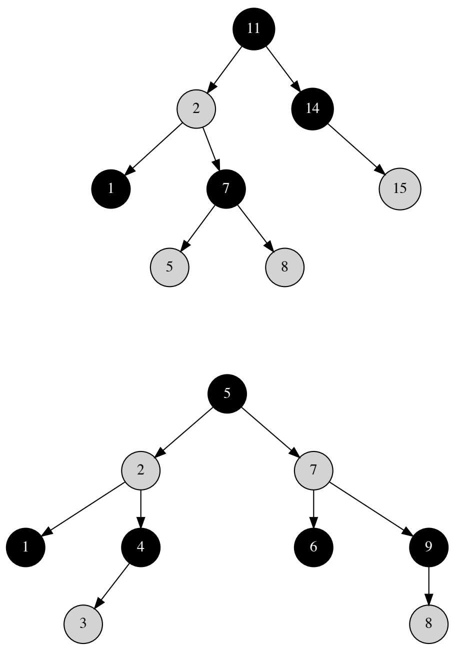


Figure 4.11: Red-black trees created by imperative algorithm.

We provide the imperative delete algorithm in Appendix A of this book.

4.6 Summary

Red-black tree is a popular implementation of balanced binary search tree. We introduce another one, called AVL tree in the next chapter. Red-black tree is a good start for more data structures. If extend the number of children from 2 to k, and maintain the balance, it leads to B-tree; If store the data along with the edge but not inside node, it leads to Radix tree. To maintain the balance, we need handle multiple cases. Okasaki's developed a method that makes the red-black tree easy to implement. There are many implementations based on this idea[16]. We also provide AVL tree and Splay tree implementation based on pattern matching in this book.

4.7 Appendix: Example programs

Definition of red-black tree node with parent field. When not explicitly defined, the color of the new node is red by default.

```
data Node<T> {
    T key
    Color color
    Node<T> left
    Node<T> right
    Node<T> parent

    Node(T x) = Node(null, x, null, Color.RED)

    Node(Node<T> l, T k, Node<T> r, Color c) {
        left = l, key = k, right = r, color = c
        if left ≠ null then left.parent = this
        if right ≠ null then right.parent = this
    }

    Self setLeft(l) {
        left = l
        if l ≠ null then l.parent = this
    }

    Self setRight(r) {
        right = r
        if r ≠ null then r.parent = this
    }

    Node<T> sibling() = if parent.left == this then parent.right
                        else parent.left

    Node<T> uncle() = parent.sibling()

    Node<T> grandparent() = parent.parent
}
```

Insert a key to red-black tree:

```
Node<T> insert(Node<T> t, T key) {
    root = t
    x = Node(key)
    parent = null
    while (t ≠ null) {
        parent = t
        t = if (key < t.key) then t.left else t.right
    }
    if (parent == null) {    //tree is empty
        root = x
    } else if (key < parent.key) {
        parent.setLeft(x)
    } else {
        parent.setRight(x)
    }
    return insertFix(root, x)
}
```

Fix the balance:

```
// Fix the red→red violation
Node<T> insertFix(Node<T> t, Node<T> x) {
    while (x.parent ≠ null and x.parent.color == Color.RED) {
        if (x.uncle().color == Color.RED) {
            // case 1: ((a:R x:R b) y:B c:R) ⟹ ((a:R x:B b) y:R c:B)
```

```
            x.parent.color = Color.BLACK
            x.grandparent().color = Color.RED
            x.uncle().color = Color.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 = leftRotate(t, x)
                }
                // case 3: ((a:R x:R b) y:B c) ==> (a:R x:B (b y:R c))
                x.parent.color = Color.BLACK
                x.grandparent().color = Color.RED
                t = rightRotate(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 = rightRotate(t, x)
                }
                // case 3': (a x:B (b y:R c:R)) ==> ((a x:R b) y:B c:R)
                x.parent.color = Color.BLACK
                x.grandparent().color = Color.RED
                t = leftRotate(t, x.grandparent())
            }
        }
    }
    t.color = Color.BLACK
    return t
}
```

Chapter 5

AVL tree

5.1 Introduction

The idea of red-black tree is to limit the number nodes along a path within a range. AVL tree takes a direct approach: quantify the difference between branches. For a node T, define:

$$\delta(T) = |r| - |l| \tag{5.1}$$

Where $|T|$ is the height of tree T, l and r are the left and right sub-trees. Define $\delta(\varnothing) = 0$ for the empty tree. If $\delta(T) = 0$ for every node T, the tree is definitely balanced. For example, a complete binary tree has $n = 2^h - 1$ nodes for height h. There are not any empty branches unless the leaves. The less absolute value of $\delta(T)$, the more balanced between the sub-trees. We call $\delta(T)$ the *balance factor* of a binary tree.

5.2 Definition

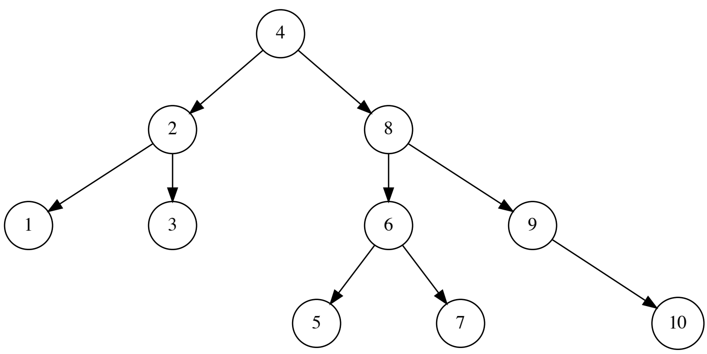


Figure 5.1: an AVL tree

A binary search tree is an AVL tree if every sub-tree T satisfies:

$$|\delta(T)| \leq 1 \tag{5.2}$$

There are three valid values for $\delta(T)$: ± 1, and 0. Figure 5.1 shows an AVL tree. This definition ensures the tree height $h = O(\lg n)$, where n is the number of nodes in the tree. Let's prove it. For an AVL tree of height h, the number of nodes varies. There are at most $2^h - 1$ nodes for a complete binary tree case. We are interesting in how many nodes at least. Let the minimum number be $N(h)$. We have the following result:

- Empty tree $\varnothing$: $h = 0$, $N(0) = 0$;
- Singleton tree: $h = 1$, $N(1) = 1$;

Figure 5.2 shows an AVL tree T of height h. It contains three parts, the key k, and two sub-trees l, r. We have the following equation:

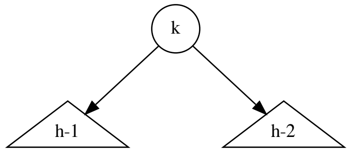


Figure 5.2: An AVL tree of height h. The height of one sub-tree is $h-1$, the other is no less than $h-2$.

$$h = max(|l|, |r|) + 1 \tag{5.3}$$

There must be a sub-tree of height $h-1$. From the definition. we have $||l| - |r|| \leq 1$ holds. Hence the height of the other tree can not be lower than $h-2$. The total number of the nodes in T is the sum of both sub-trees plus 1 (for the root):

$$N(h) = N(h-1) + N(h-2) + 1 \tag{5.4}$$

This recursive equation is similar to Fibonacci numbers. Actually we can transform it to Fibonacci numbers through $N'(h) = N(h) + 1$. Equation (5.4) then changes to:

$$N'(h) = N'(h-1) + N'(h-2) \tag{5.5}$$

Lemma 5.2.1. *Let $N(h)$ be the minimum number of nodes for an AVL tree of height h, and $N'(h) = N(h) + 1$, then*

$$N'(h) \geq \phi^h \tag{5.6}$$

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

Proof. When $h = 0$ or 1, 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, assume $N'(h) \geq \phi^h$.

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

□

From Lemma 5.2.1, we immediately obtain:

$$h \leq log_{\phi}(n+1) = log_{\phi}2 \cdot \lg(n+1) \approx 1.44 \lg(n+1) \tag{5.7}$$

We prove the height of AVL tree is proportion to $O(\lg n)$, indicating AVL tree is balanced.

When insert or delete, the balance factor may exceed the valid value range, we need fix to resume $|\delta| < 1$. Traditionally, the fixing is through tree rotations. We give the simplified implementation based on pattern matching. The idea is similar to the functional red-black tree(Okasaki,[13]). Because of this 'modify-fix' approach, AVL tree is also self-balanced binary search tree. We can re-use the binary search tree definition. Although the balance factor δ can be computed recursively, we record it inside each node as $T = (l, k, r, \delta)$, and update it when mutate the tree[1]. Below example program adds δ as an `Int`:

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

For AVL tree, *lookup*, *max*, *min* are as same as the binary search tree. We focus on *insert* and *delete* algorithms.

5.3 Insert

When insert a new element, $|\delta(T)|$ may exceed 1. We can use pattern matching similar to red-black tree to develop a simplified solution. After insert element x, for those sub-trees which are the ancestors of x, the height may increase at most by 1. We need recursively update the balance factor along the path of insertion. Define the insert result as a pair $(T', \Delta H)$, where T' is the updated tree and ΔH is the increment of height. We modify the binary search tree *insert* function as below:

$$insert = fst \circ ins \tag{5.8}$$

Where $fst\ (a, b) = a$ returns the first element in a pair. $ins(T, k)$ does the actual work to insert element k into tree T:

$$\begin{array}{rcl} ins\ \varnothing\ k &=& ((\varnothing, k, \varnothing, 0), 1) \\ ins\ (l, k', r, \delta)\ k &=& \begin{cases} k < k' : tree\ (ins\ l\ k)\ k'\ (r, 0)\ \delta \\ k > k' : tree\ (l, 0)\ k'\ (ins\ r, k)\ \delta \end{cases} \end{array} \tag{5.9}$$

If the tree is empty $\varnothing$, the result is a leaf of k with balance factor 0. The height increases to 1. Otherwise let $T = (l, k', r, \delta)$. We compare the new element k with k'. If $k < k'$, we recursively insert k it to the left sub-tree l, otherwise insert to r. As the recursive insert result is a pair of $(l', \Delta l)$ or $(r', \Delta r)$, we need adjust the balance factor and update tree height through function *tree*, it takes 4 parameters: $(l', \Delta l)$, k', $(r', \Delta r)$, and δ. The result is $(T', \Delta H)$, where T' is the new tree, and ΔH is defined as:

$$\Delta H = |T'| - |T| \tag{5.10}$$

We can further break it down into 4 cases:

$$\begin{array}{rcl} \Delta H &=& |T'| - |T| \\ &=& 1 + max(|r'|, |l'|) - (1 + max(|r|, |l|)) \\ &=& max(|r'|, |l'|) - max(|r|, |l|) \\ &=& \begin{cases} \delta \geq 0, \delta' \geq 0 : & \Delta r \\ \delta \leq 0, \delta' \geq 0 : & \delta + \Delta r \\ \delta \geq 0, \delta' \leq 0 : & \Delta l - \delta \\ otherwise : & \Delta l \end{cases} \end{array} \tag{5.11}$$

[1]Alternatively, we can record the height instead of δ[20].

Where $\delta' = \delta(T') = |r'| - |l'|$, is the updated balance factor. Appendix B provides the proof for it. We need determine δ' before balance adjustment.

$$
\begin{array}{rcl}
\delta' & = & |r'| - |l'| \\
& = & |r| + \Delta r - (|l| + \Delta l) \\
& = & |r| - |l| + \Delta r - \Delta l \\
& = & \delta + \Delta r - \Delta l
\end{array}
\tag{5.12}
$$

With the changes in height and balance factor, we can define the *tree* function in (5.9):

$$
tree\ (l', \Delta l)\ k\ (r', \Delta r)\ \delta = balance\ (l', k, r', \delta')\ \Delta H \tag{5.13}
$$

Below example programs implements what we deduced so far:

```
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 = tree (l,  0) k (ins r) d

tree (l, dl) k (r, dr) d = balance (Br l k r d') deltaH where
    d' = d + dr - dl
    deltaH | d ≥ 0 && d' ≥ 0 = dr
           | d ≤ 0 && d' ≥ 0 = d+dr
           | d ≥ 0 && d' ≤ 0 = dl - d
           | otherwise = dl
```

5.3.1 Balance

There are 4 cases need fix as shown in figure 5.3. The balance factor is ± 2, exceeds the range of $[-1, 1]$. We adjust them to a uniformed structure in the center, with the $\delta(y) = 0$.

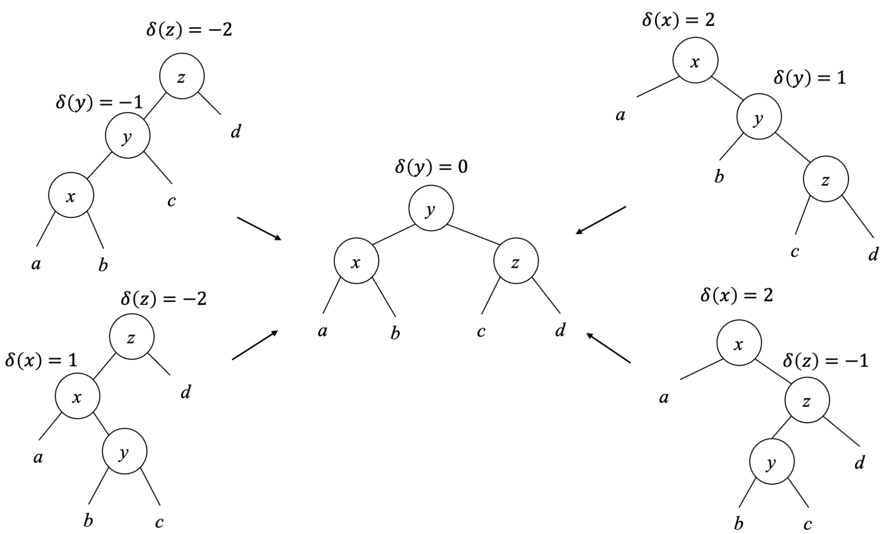


Figure 5.3: Fix 4 cases to the same structure

We call the 4 cases: left-left, right-right, right-left, and left-right. Denote the balance factors before fixing as $\delta(x), \delta(y)$, and $\delta(z)$; after fixing, they change to $\delta'(x), \delta'(y) = 0$, and $\delta'(z)$ respectively. The values of $\delta'(x)$ and $\delta'(z)$ can be given as below. Appendix B gives the proof.

Left-left:

$$\begin{array}{l}\delta'(x) = \delta(x) \\ \delta'(y) = 0 \\ \delta'(z) = 0\end{array} \tag{5.14}$$

Right-right:

$$\begin{array}{l}\delta'(x) = 0 \\ \delta'(y) = 0 \\ \delta'(z) = \delta(z)\end{array} \tag{5.15}$$

Right-left and Left-right:

$$\begin{array}{l}\delta'(x) = \begin{cases}\delta(y) = 1 : & -1 \\ otherwise : & 0\end{cases} \\ \delta'(y) = 0 \\ \delta'(z) = \begin{cases}\delta(y) = -1 : & 1 \\ otherwise : & 0\end{cases}\end{array} \tag{5.16}$$

Based on this, we can implement the pattern matching fix as below:

$$\begin{array}{rcl}
balance\ (((a, x, b, \delta(x)), y, c, -1), z, d, -2)\ \Delta H & = & ((a, x, b, \delta(x)), y, (c, z, d, 0), 0, \Delta H - 1) \\
balance\ (a, x, (b, y, (c, z, d, \delta(z)), 1), 2)\ \Delta H & = & ((a, x, b, 0), y, (c, z, d, \delta(z)), 0, \Delta H - 1) \\
balance\ ((a, x, (b, y, c, \delta(y)), 1), z, d, -2)\ \Delta H & = & ((a, x, b, \delta'(x)), y, (c, z, d, \delta'(z)), 0, \Delta H - 1) \\
balance\ (a, x, ((b, y, c, \delta(y)), z, d, -1), 2)\ \Delta H & = & ((a, x, b, \delta'(x)), y, (c, z, d, \delta'(z)), 0, \Delta H - 1) \\
balance\ T\ \Delta H & = & (T, \Delta H)
\end{array} \tag{5.17}$$

Where $\delta'(x)$ and $\delta'(z)$ are defined in (B.17). If none of the pattern matches, the last row keeps the tree unchanged. Below is the example program implements *balance*:

```
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
    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
balance t d = (t, d)
```

The performance of *insert* is proportion to the height of the tree. From (5.7), it is bound to is $O(\lg n)$ where n is the number of elements in the tree.

Verification

To test an AVL tree, we need verify two things: It is a binary search tree; and for every sub-tree T, equation (5.2): $\delta(T) \leq 1$ holds. Below function examines the height difference between the two sub-trees recursively:

$$\begin{array}{rcl}avl?\ \varnothing & = & True \\ avl?\ T & = & avl?\ l\ \wedge avl?\ r\ \wedge ||r| - |l|| \leq 1\end{array} \tag{5.18}$$

Where l, r are the left and right sub-trees. The height is calculated recursively:

$$
\begin{array}{rcl}
|\varnothing| & = & 0 \\
|T| & = & 1 + max(|r|, |l|)
\end{array}
\tag{5.19}
$$

Below example program implements AVL tree height verification:

```
isAVL Empty = True
isAVL (Br l _ r _) = isAVL l && isAVL r && abs (height r - height l) ≤ 1

height Empty = 0
height (Br l _ r _) = 1 + max (height l) (height r)
```

Exercise 5.1

1. We only give the algorithm to test AVL height. Complete the program to test if a binary tree is AVL tree.

5.4 Imperative AVL tree algorithm ★

This section gives the imperative algorithm for completeness. Similar to the red-black tree algorithm, we first re-use the binary search tree insert, then fix the balance through tree rotations.

```
function Insert(T, k)
    root ← T
    x ← Create-Leaf(k)
    δ(x) ← 0
    parent ← NIL
    while T ≠ NIL do
        parent ← T
        if k < Key(T) then
            T ← Left(T)
        else
            T ← Right(T)
    Parent(x) ← parent
    if parent = NIL then                    ▷ tree T is empty
        return x
    else if k < Key(parent) then
        Left(parent) ← x
    else
        Right(parent) ← x
    return AVL-Insert-Fix(root, x)
```

After insert, the balance factor δ may change because of the tree growth. Insert to the right may increase δ by 1, while insert to the left may decrease it. We perform bottom-up fixing from x to root. Denote the new balance factor as δ', there are 3 cases:

- $|\delta| = 1$, $|\delta'| = 0$. The new node makes the tree well balanced. The height of the parent keeps unchanged.
- $|\delta| = 0$, $|\delta'| = 1$. Either the left or the right sub-tree increases its height. We need go on checking the upper level.
- $|\delta| = 1$, $|\delta'| = 2$. We need rotate the tree to fix the balance factor.

```
function AVL-Insert-Fix(T, x)
    while Parent(x) ≠ NIL do
        P ← Parent(x)
        L ← Left(x)
        R ← Right(x)
        δ ← δ(P)
        if x = Left(P) then
            δ' ← δ − 1
        else
            δ' ← δ + 1
        δ(P) ← δ'
        if |δ| = 1 and |δ'| = 0 then                    ▷ Height unchanged
            return T
        else if |δ| = 0 and |δ'| = 1 then               ▷ Go on bottom-up update
            x ← P
        else if |δ| = 1 and |δ'| = 2 then
            if δ' = 2 then
                if δ(R) = 1 then                        ▷ Right-right
                    δ(P) ← 0                            ▷ By (B.6)
                    δ(R) ← 0
                    T ← Left-Rotate(T, P)
                if δ(R) = −1 then                       ▷ Right-left
                    δ_y ← δ(Left(R))                    ▷ By (B.17)
                    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
                    δ(P) ← 0
                    δ(L) ← 0
                    Right-Rotate(T, P)
                else                                    ▷ Left-Right
                    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

Besides rotation, we also need update δ for the impacted nodes. The right-right and left-left cases need one rotation, while the right-left and left-right case need two rotations. We skip the AVL tree delete algorithm in this chapter. Appendix B provides the delete implementation.

5.5 Summary

AVL tree was developed in 1962 by Adelson-Velskii and Landis[18],[19]. It is named after the two authors. AVL tree was developed earlier than the red-black tree. Both are self-balance binary search trees. Most tree operations are bound $O(\lg n)$ time. From (5.7), AVL tree is more rigidly balanced, and performs faster than red-black tree in looking up intensive applications[18]. However, red-black tree performs better in frequently insertion and removal cases. Many popular self-balance binary search tree libraries are implemented on top of red-black tree. AVL tree also provides the intuitive and effective solution to the balance problem.

5.6 Appendix: Example programs

Definition of AVL tree node.

```
data Node<T> {
    int delta
    T key
    Node<T> left
    Node<T> right
    Node<T> parent
}
```

Fix the balance:

```
Node<T> insertFix(Node<T> t, Node<T> x) {
    while (x.parent != null ) {
        var (p, l, r) = (x.parent, x.parent.left, x.parent.right)
        var d1 = p.delta
        var d2 = if x == parent.left then d1 - 1 else d1 + 1
        p.delta = d2

        if abs(d1) == 1 and abs(d2) == 0 {
            return t
        } else if abs(d1) == 0 and abs(d2) == 1 {
            x = p
        } else if abs(d1) == 1 and abs(d2) == 2 {
            if d2 == 2 {
                if r.delta == 1 {    //Right-right
                    p.delta = 0
                    r.delta = 0
                    t = rotateLeft(t, p)
                } else if r.delta == -1 {    //Right-Left
                    var dy = r.left.delta
                    p.delta = if dy == 1 then -1 else 0
                    r.left.delta = 0
                    r.delta = if dy == -1 then 1 else 0
                    t = rotateRight(t, r)
                    t = rotateLeft(t, p)
                }
```

```
            } else if d2 == -2 {
                if l.delta == -1 {    //Left-left
                    p.delta = 0
                    l.delta = 0
                    t = rotateRight(t, p)
                } else if l.delta == 1 {    //Left-right
                    var dy = l.right.delta
                    l.delta = if dy == 1 then -1 else 0
                    l.right.delta = 0
                    p.delta = if dy == -1 then 1 else 0
                    t = rotateLeft(t, l)
                    t = rotateRight(t, p)
                }
            }
            break
        }
    }
    return t
}
```

Chapter 6

Radix tree

Binary search tree stores data in nodes. Can we use the edges to carry information? Radix trees, including trie, prefix tree, and suffix tree are the data structures developed based on this idea in 1960s. They are widely used in compiler design[21], and bio-information processing, like DNA pattern matching[23].

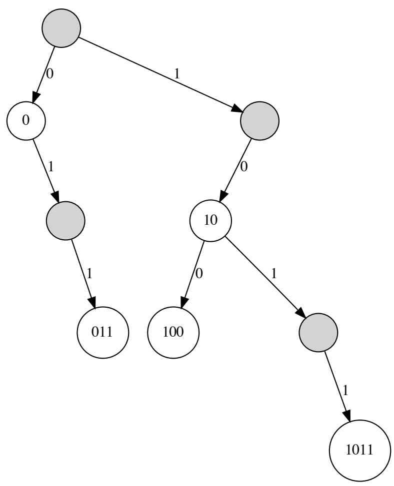


Figure 6.1: Radix tree.

Figure 6.1 shows a Radix tree. It contains bits 1011, 10, 011, 100 and 0. When lookup a key $k = (b_0 b_1 ... b_n)_2$, we take the first bit b_0 (MSB from left), check whether it is 0 or 1. For 0, turn left, else turn right. Then take the second bit and repeat looking up till either reach a leaf node or consume all the n bits. We needn't store keys in Radix tree node. The information is represented by edges. The nodes labelled with key in figure 6.1 are for illustration purpose. If the keys are integers, we can represent them in binary format, and implement lookup with bit-wise manipulations.

6.1 Integer trie

We call the data structure in figure 6.1 *binary trie*. Trie was developed by Edward Fredkin in 1960. It comes from "re**trie**val", pronounce as /'tri:/ by Freddkin, while others pronounce it as /'trai/ "try"[24]. Although it's also called prefix tree in some context, we treat trie and prefix tree different in this chapter. 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'[21]. Consider the binary trie in figure 6.2. The three keys are different bit strings of "11", "011", and "0011" although they are all equal to 3.

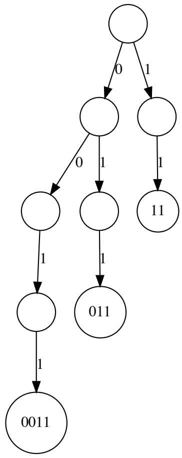


Figure 6.2: A big-endian trie.

It is inefficient to treat the prefix zeros as valid bits. For 32 bits integers, we need a tree of 32 levels to insert number 1. Okasaki suggested to use little-endian integers instead[21]. 1 is represented as bits $(1)_2$, 2 as $(01)_2$, and 3 is $(11)_2$, ...

6.1.1 Definition

We can re-use binary tree structure to define the little-endian binary trie. A node is either empty, or a branch containing the left, right sub-trees, and an optional value. The left sub-tree is encoded as 0 and the right sub-tree is encoded as 1.

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

Given a node in the binary trie, the integer key bound to it is uniquely determined through its position. That is the reason we need not save the key, but only the value in the node. The type of the key is always integer, we call the tree $IntTrie\ A$ if the value is of type A.

6.1.2 Insert

When insert an integer key k and a value v, we convert k into binary form. If k is even, the lowest bit is 0, we recursively insert to the left sub-tree; otherwise if k is odd, the lowest bit is 1, we recursively insert to the right. We next divide k by 2 to remove the lowest bit. For none empty trie $T = (l, v', r)$, where l, r are the left and right sub-trees, and v' is the optional value, function $insert$ can be defined as below:

$$
\begin{array}{rcl}
insert\ \varnothing\ k\ v & = & insert(\varnothing, Nothing, \varnothing)\ k\ v \\
insert\ (l, v', r)\ 0\ v & = & (l, Just\ v, r) \\
insert\ (l, v', r)\ k\ v & = & \begin{cases} even(k): & (insert\ l\ \frac{k}{2}\ v, v', r) \\ odd(k): & (l, v', insert\ r\ \lfloor \frac{k}{2} \rfloor\ v) \end{cases}
\end{array}
\tag{6.1}
$$

If $k = 0$, we put v in the node. When $T = \varnothing$, it becomes $(\varnothing, Just\ v, \varnothing)$. As far as $k \neq 0$, we goes down along the tree based on the parity of k. We create empty leaf $(\varnothing, Nothing, \varnothing)$ whenever meet $\varnothing$ node. This algorithm overrides the value if k already exists. Alternatively, we can store a list, and append v to it. Figure 6.3 shows an example trie, generated by inserting the key-value pairs of $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$. Below is the example program implements *insert*:

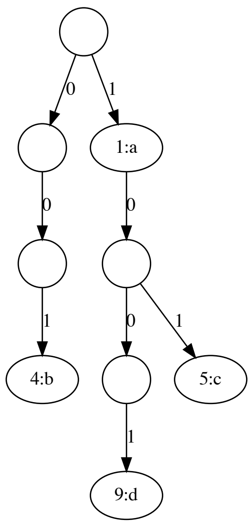


Figure 6.3: A little-endian integer binary trie of $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$.

```
insert Empty k x = insert (Branch Empty Nothing Empty) k x
insert (Branch l v r) 0 x = Branch l (Just x) r
insert (Branch l v r) k x | even k    = Branch (insert l (k `div` 2) x) v r
                          | otherwise = Branch l v (insert r (k `div` 2) x)
```

We can define the even/odd testing by modular 2, and check if the remainder is 0 or not: $even(k) = (k \bmod 2 = 0)$. Or use bit-wise operation in some environment, like `(k & 0x1) == 0`. We can eliminate the recursion through loops to realize an iterative implementation as below:

```
function INSERT(T, k, v)
    if T = NIL then
        T ← EMPTY-NODE                                   ▷ (NIL, Nothing, NIL)
    p ← T
    while k ≠ 0 do
        if EVEN?(k) then
            if LEFT(p) = NIL then
                LEFT(p) ← EMPTY-NODE
            p ← LEFT(p)
        else
            if RIGHT(p) = NIL then
                RIGHT(p) ← EMPTY-NODE
            p ← RIGHT(p)
        k ← ⌊k/2⌋
    VALUE(p) ← v
    return T
```

INSERT takes, a trie T, a key k, and a value v. For integer k with m bits in binary, it goes into m levels of the trie. The performance is bound to $O(m)$.

6.1.3 Look up

When look up key k in a none empty integer trie, if $k = 0$, then the root node is the target. Otherwise, we check the lowest bit, then recursively look up the left or right sub-tree accordingly.

$$
\begin{array}{rcl}
lookup\ \varnothing\ k & = & Nothing \\
lookup\ (l, v, r)\ 0 & = & v \\
lookup\ (l, v, r)\ k & = & \begin{cases} even(k): & lookup\ l\ \dfrac{k}{2} \\ odd(k): & lookup\ r\ \lfloor \dfrac{k}{2} \rfloor \end{cases}
\end{array}
\tag{6.2}
$$

Below example program implements the *lookup* function:

```
lookup Empty _ = Nothing
lookup (Branch _ v _) 0 = v
lookup (Branch l _ r) k | even k     = lookup l (k `div` 2)
                        | otherwise = lookup r (k `div` 2)
```

We can eliminate the recursion to implement the iterative *lookup* as the following:

function LOOKUP(T, k)
    while $k \neq 0$ and $T \neq$NIL **do**
        if EVEN?(k) **then**
            $T \leftarrow$ LEFT(T)
        else
            $T \leftarrow$ RIGHT(T)
        $k \leftarrow \lfloor k/2 \rfloor$
    if $T \neq$ NIL **then**
        return VALUE(T)
    else
        return NIL

The *lookup* function is bound to $O(m)$ time, where m is the number of bits of k.

Exercise 6.1

1. Can we change the definition from `Branch (IntTrie a) (Maybe a) (IntTrie a)` to `Branch (IntTrie a) a (IntTrie a)`, and return *Nothing* if the value does not exist, and *Just v* otherwise?

6.2 Integer prefix tree

Trie is not space efficient. As shown in figure 6.3, there are only 4 nodes with value, while the rest 5 are empty. The space usage is less than 50%. To improve the efficiency, we can consolidate the chained nodes to one. Integer prefix tree is such a data structure developed by Donald R. Morrison in 1968. He named it as 'Patricia', standing for **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric[22]. When the keys are integer, we call it integer prefix tree or simply integer tree when the context is clear. Okasaki provided the implementation in[21]. Consolidate the chained nodes in figure 6.3, we obtained an integer tree as shown in figure 6.4.

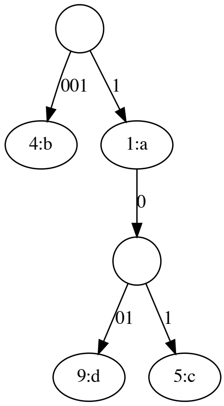


Figure 6.4: Little endian integer tree for the map $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$.

The key to the branch node is the longest common prefix for its descendant trees. In other words, the sibling sub-trees branch out at the bit where ends at their longest prefix. As the result, integer tree eliminates the redundant spaces in trie.

6.2.1 Definition

Integer prefix tree is a special binary tree. It is either empty or a node of:

- A leaf contains an integer key k and a value v;
- Or a branch with the left and right sub-trees, that share the **longest common prefix** bits for their keys. For the left sub-tree, the next bit is 0, for the right, it is 1.

Below example program defines the integer prefix tree. The branch node contains 4 components: The longest prefix, a mask integer indicating from which bit the sub-trees branch out, the left and right sub-trees. The mask is $m = 2^n$ for some integer $n \geq 0$. All bits that are lower than n do not belong to the common prefix.

```
data IntTree a = Empty
               | Leaf Int a
               | Branch Int Int (IntTree a) (IntTree a)
```

6.2.2 Insert

When insert integer y to tree T, if T is empty, we create a leaf of y; If T is a singleton leaf of x, besides the new leaf of y, we need create a branch node, set x and y as the two sub-trees. To determine whether y is on the left or right, we need find the longest common prefix p of x and y. For example if $x = 12 = (1100)_2$, $y = 15 = (1111)_2$, then $p = (11oo)_2$, where o denotes the bits we don't care. We can use another integer m to mask those bits. In this example, $m = 4 = (100)_2$. The next bit after p presents 2^1. It is 0 in x, 1 in y. Hence, we set x as the left sub-tree and y as the right, as shown in figure 6.5.

If T is neither empty nor a leaf, we firstly check if y matches the longest common prefix p in the root, then recursively insert it to the sub-tree according to the next bit after p. For example, when insert $y = 14 = (1110)_2$ to the tree shown in figure 6.5, since $p = (11oo)_2$ and the next bit (the bit of 2^1) is 1, we recursively insert y to the right

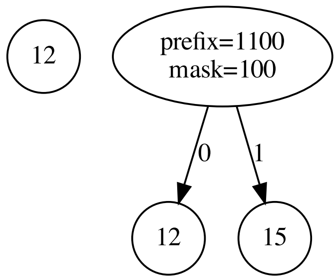
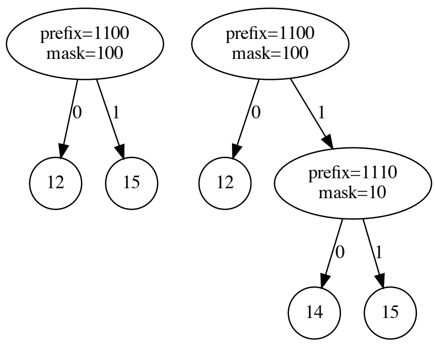
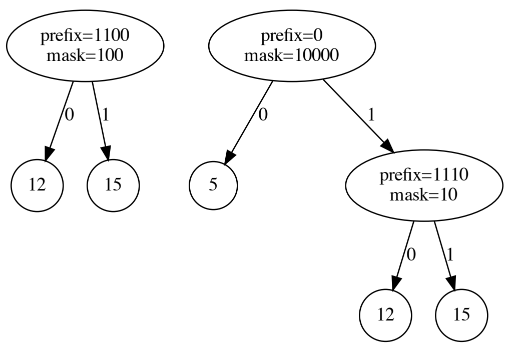


Figure 6.5: Left: T is a leaf of 12; Right: After insert 15.

sub-tree. If y does not match p in the root, we need branch a new leaf as shown in figure 6.6.



(a) Insert $14 = (1110)_2$, which matches $p = (1100)_2$. It is inserted to the right.



(b) Insert $5 = (101)_2$, which does not match $p = (1100)_2$. Branch out a new leaf.

Figure 6.6: The tree is a branch node.

For integer key k and value v, let (k, v) be the leaf. For branch node, denote it as (p, m, l, r), where p is the longest common prefix, m is the mask, l and r are the left and right sub-trees. Below *insert* function defines the above 3 cases:

$$
\begin{array}{rcl}
insert\ \varnothing\ k\ v & = & (k, v) \\
insert\ (k, v')\ k\ v & = & (k, v) \\
insert\ (k', v')\ k\ v & = & join\ k\ (k, v)\ k'\ (k', v') \\
insert\ (p, m, l, r)\ k\ v & = & \begin{cases} match(k, p, m) : & \begin{cases} zero(k, m) : & (p, m, insert\ l\ k\ v) \\ otherwise : & (p, m, insert\ r\ k\ v) \end{cases} \\ otherwise : & join\ k\ (k, v)\ p\ (p, m, l, r) \end{cases}
\end{array}
\tag{6.3}
$$

The first clause creates a leaf when $T = \varnothing$; the second clause overrides the value for the same key. Function $match(k, p, m)$ tests if integer k and prefix p have the same bits after masked with m through: $mask(k, m) = p$, where $mask(k, m) = \overline{m - 1} \& k$. It applies bit-wise not to $m - 1$, then does bit-wise and with k. $zero(k, m)$ tests the next bit in k masked with m is 0 or not. We shift m one bit to right, then do bit-wise and with k:

$$
zero(k, m) = x \& (m >> 1) \tag{6.4}
$$

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 as $(p, m) = LCP(p_1, p_2)$, creates a new branch node, then set T_1 and T_2 as the two sub-trees:

$$
join(p_1, T_1, p_2, T_2) = \begin{cases} zero(p1, m) : & (p, m, T_1, T_2) \\ otherwise : & (p, m, T_2, T_1) \end{cases} \tag{6.5}
$$

To calculate the longest common prefix, we can firstly compute bit-wise exclusive-or for $p1$ and $p2$, then count the highest bit $highest(xor(p_1, p_2))$ as:

$$
\begin{array}{rcl}
highest(0) & = & 0 \\
highest(n) & = & 1 + highest(n >> 1)
\end{array}
$$

Then generate a mask $m = 2^{highest(xor(p_1, p_2))}$. The longest common prefix p can be given by masking the bits with m for either p_1 or p_2, like $p = mask(p_1, m)$. The following example program implements the *insert* function:

```
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
        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

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 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)
```

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

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

We can also implement *insert* imperatively:

```
function Insert(T, k, v)
    if T = NIL then
        return Create-Leaf(k, v)
    y ← T
    p ← NIL
    while y is not leaf, and Match(k, Prefix(y), Mask(y)) do
        p ← y
        if Zero?(k, Mask(y)) then
            y ← Left(y)
        else
            y ← Right(y)
    if y is leaf, and k = Key(y) then
        Value(y) ← v
    else
        z ← Branch(y, Create-Leaf(k, v))
        if p = NIL then
            T ← z
        else
            if Left(p) = y then
                Left(p) ← z
            else
                Right(p) ← z
    return T
```

Where Branch(T_1, T_2) creates a new branch node, extracts the longest common prefix, then sets T_1 and T_2 as the two sub-trees.

```
function Branch(T1, T2)
    T ← Empty-Node
    (Prefix(T), Mask(T)) ← LCP(Prefix(T1), Prefix(T2))
    if Zero?(Prefix(T1), Mask(T)) then
        Left(T) ← T1
        Right(T) ← T2
    else
        Left(T) ← T2
        Right(T) ← T1
    return T

function Zero?(x, m)
    return (x & ⌊m/2⌋) = 0
```

Function LCP find the longest bit prefix from two integers:

```
function LCP(a, b)
    d ← xor(a, b)
    m ← 1
    while d ≠ 0 do
        d ← ⌊d/2⌋
```

    $m \leftarrow 2m$
  return (MaskBit(a, m), m)

function MaskBit(x, m)
  return $x \& \overline{m - 1}$

Figure 6.7 gives an example integer tree created from the *insert* algorithm. Although integer prefix tree consolidates the chained nodes, the operation to extract the longest common prefix need linear scan the bits. For integer of m bits, the insert is bound to $O(m)$.

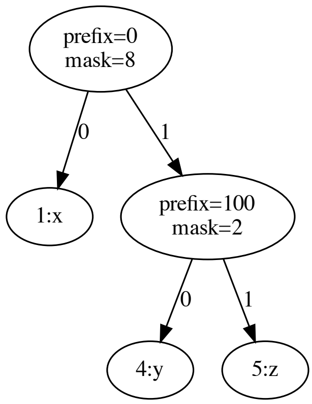


Figure 6.7: Insert $\{1 \rightarrow x, 4 \rightarrow y, 5 \rightarrow z\}$ to the big-endian integer tree.

6.2.3 Lookup

When lookup key k, if the integer tree $T = \varnothing$ or it is a leaf of $T = (k', v)$ with different key, then k does not exist; if $k = k'$, then v is the result; if $T = (p, m, l, r)$ is a branch node, we need check if the common prefix p matches k under the mask m, then recursively lookup the sub-tree l or r upon next bit. If fails to match the common prefix p, then k does not exist.

$$
\begin{array}{rcl}
lookup\ \varnothing\ k & = & Nothing \\
lookup\ (k', v)\ k & = & \begin{cases} k = k' : & Just\ v \\ otherwise : & Nothing \end{cases} \\
lookup\ (p, m, l, r)\ k & = & \begin{cases} match(k, p, m) : & \begin{cases} zero(k, m) : & lookup\ l\ k \\ otherwise : & lookup\ r\ k \end{cases} \\ otherwise : & Nothing \end{cases}
\end{array}
\tag{6.6}
$$

We can also eliminate the recursion to implement the iterative lookup algorithm.

function Look-Up(T, k)
  if $T =$ NIL **then**
    return NIL
  while T is not leaf, and Match(k, Prefix(T), Mask(T)) **do**
    if Zero?(k, Mask(T)) **then**
      $T \leftarrow$ Left(T)
    else
      $T \leftarrow$ Right(T)

```
    if T is leaf, and KEY(T) = k then
        return VALUE(T)
    else
        return NIL
```

The *lookup* algorithm is bound to $O(m)$, where m is the number of bits in the key.

Exercise 6.2

1. Write a program to implement the *lookup* function.
2. Implement the pre-order traverse for both integer trie and integer tree. Only output the keys when the nodes store values. What pattern does the result follow?

6.3 Trie

From integer trie and tree, we can extend the key to a list of elements. Particularly the trie and tree with key in alphabetic string are powerful tools for text manipulation.

6.3.1 Definition

When extend the key type from 0/1 bits to generic list, the tree structure changes from binary tree to multiple sub-trees. Taking English characters for example, there are up to 26 sub-trees when ignore the case as shown in figure 6.8.

Not all the 26 sub-trees contain data. In figure 6.8, there are only three none empty sub-trees bound to 'a', 'b', and 'z'. Other sub-trees, such as for 'c', are empty. We can hide them in the figure. When it is case sensitive, or extent the key from alphabetic string to generic list, we can adopt collection types, like map to define trie.

A trie is either empty or a node of 2 kinds:

1. A leaf of value v without any sub-trees;
2. A branch, containing a value v and multiple sub-trees. Each sub-tree is bound to an element k of type K.

Let the type of value be V, we denote the trie as $Trie\ K\ V$. Below example program defines trie.

```
data Trie k v = Trie { value :: Maybe v
                     , subTrees :: [(k, Trie k v)]}
```

The empty trie is in form of $(Nothing, \varnothing)$.

6.3.2 Insert

When insert a pair of key and value to the trie, where the key is a list of elements. Let the trie be $T = (v, ts)$, where v is the value stored in the trie, and $ts = \{c_1 \mapsto T_1, c_2 \mapsto T_2, ..., c_m \mapsto T_m\}$ contains mappings between elements and sub-trees. Element c_i is mapped to sub-tree T_i. We can either implement the mapping through associated list: $[(c_1, T_1), (c_2, T_2), ..., (c_m, T_m)]$, or through self-balanced tree map (Chapter 4 or 5).

$$
\begin{array}{rcl}
insert\ (v, ts)\ \varnothing\ v' & = & (v', ts) \\
insert\ (v, ts)\ (k : ks)\ v' & = & (v, ins\ ts)
\end{array}
\tag{6.7}
$$

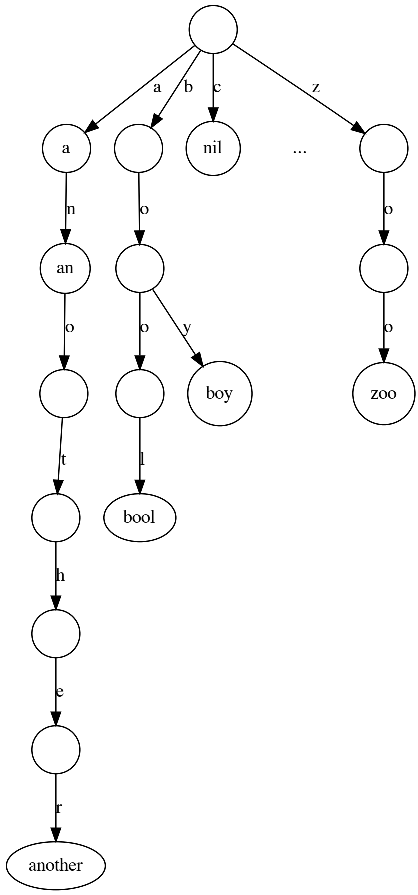


Figure 6.8: A trie of 26 branches, containing key 'a', 'an', 'another', 'bool', 'boy', and 'zoo'.

When the key is empty, we override the value; otherwise, we extract the first element k, check if there is a map among the sub-trees for k, and recursively insert ks and v':

$$
\begin{array}{rcl}
ins\ \varnothing & = & [k \mapsto insert\ (Nothing, \varnothing)\ ks\ v'] \\
ins\ ((c \mapsto t) : ts) & = & \begin{cases} c = k : & (k \mapsto insert\ t\ ks\ v') : ts \\ otherwise : & (c \mapsto t) : (ins\ ts) \end{cases}
\end{array} \tag{6.8}
$$

If there is no sub-tree in the node, we create a mapping from k to an empty trie node $t = (Nothing, \varnothing)$; otherwise, we located the sub-tree t mapped to k, then recursively insert ks and v' to t. Below is the example program implement *insert*, it's based on associated list to manage sub-tree mappings.

```
insert (Trie _ ts) [] x = Trie (Just x) ts
insert (Trie v ts) (k:ks) x = Trie v (ins ts) where
    ins [] = [(k, insert empty ks x)]
    ins ((c, t) : ts) = if c == k then (k, insert t ks x) : ts
                        else (c, t) : (ins ts)

empty = Trie Nothing []
```

We can also eliminate the recursion to implement *insert* iteratively.

1: **function** INSERT(T, k, v)
2: **if** $T =$ NIL **then**
3: $T \leftarrow$ EMPTY-NODE
4: $p \leftarrow T$
5: **for** each c in k **do**
6: **if** SUB-TREES(p)[c] = NIL **then**
7: SUB-TREES(p)[c] $\leftarrow$ EMPTY-NODE
8: $p \leftarrow$ SUB-TREES(p)[c]
9: VALUE(p) $\leftarrow v$
10: **return** T

For the key type $[K]$ (list of K), if K is finite set of m elements, and the length of the key is n, then the insert algorithm is bound to $O(mn)$. When the key is lower case English strings, then $m = 26$, the insert operation is proportion to the length of key string.

6.3.3 Look up

When look up a none empty key $(k : ks)$ from trie $T = (v, ts)$, starting from the first element k, if there exists sub-tree T' mapped to k, we then recursively lookup ks in T'. When the key is empty, then return the value as result:

$$
\begin{array}{rcl}
lookup\ \varnothing\ (v, ts) & = & v \\
lookup\ (k : ks)\ (v, ts) & = & \begin{cases} lookup_l\ k\ ts = Nothing : & Nothing \\ lookup_l\ k\ ts = Just\ t : & lookup\ ks\ t \end{cases}
\end{array} \tag{6.9}
$$

Where function $lookup_l$ is defined in chapter 1. It looks up if a key exits in an assoc list. Below is the corresponding iterative implementation:

1: **function** LOOK-UP(T, key)
2: **if** $T =$ NIL **then**
3: **return** *Nothing*
4: **for** each c in key **do**
5: **if** SUB-TREES(T)[c] = NIL **then**
6: **return** *Nothing*

$T \leftarrow$ SUB-TREES$(T)[c]$
return VALUE(T)

The lookup algorithm is bound to $O(mn)$, where n is the length of the key, and m is the size of the element set.

Exercise 6.3

1. Use the self-balance binary tree, like red-black tree or AVL tree to implement a *map* data structure, and manage the sub-trees with *map*. We call such implementation $MapTrie$ and $MapTree$ respectively. What are the performance of *insert* and *lookup* for map based tree and trie?

6.4 Prefix tree

Trie is not space efficient. We can consolidate the chained nodes to obtain the prefix tree.

6.4.1 Definition

A prefix tree node t contains two parts: an optional value v; zero or multiple sub prefix trees, each t_i is bound to a list s_i. The sub-trees and their mappings are denoted as $[s_i \mapsto t_i]$. These lists share the longest common prefix s bound to the node t. i.e. s is the longest common prefix of $s \mathbin{+\!\!+} s_1$, $s \mathbin{+\!\!+} s_2$, ... For any $i \neq j$, list s_i and s_j don't have none empty common prefix. Consolidate the chained nodes in figure 6.8, we obtain the corresponding prefix tree in figure 6.9.

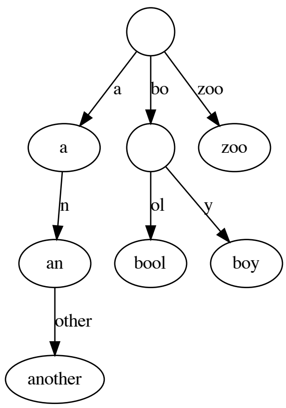


Figure 6.9: A prefix tree with keys: 'a', 'an', 'another', 'bool', 'boy', 'zoo'.

Below example program defines the prefix tree:

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

We denote prefix tree $t = (v, ts)$. Particularly, $(Nothing, \varnothing)$ is the empty node, and $(Just\ v, \varnothing)$ is a leaf node of value v.

6.4.2 Insert

When insert key s, if the prefix tree is empty, we create a leaf node of s as figure 6.10 (a); otherwise, if there exits common prefix between s and s_i, where s_i is bound to some sub-tree t_i, we branch out a new leaf t_j, extract the common prefix, and map it to a new internal branch node t', then put t_i and t_j as two sub-trees of t'. Figure 6.10 (b) shows this case. There are two special cases: s is the prefix of s_i as shown in figure 6.10 (c) → (e); or s_i is the prefix of s as shown in figure 6.10 (d) → (e).

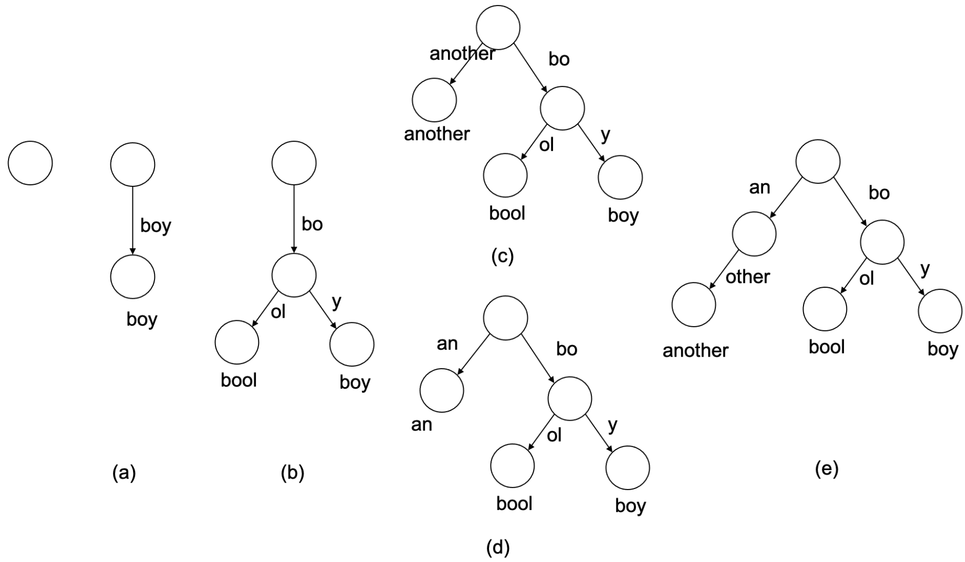


Figure 6.10: (a) insert 'boy' to empty tree; (b) insert 'bool', branch a new node out; (c) insert 'another' to (b); (d) insert 'an' to (b); (e) insert 'an' to (c), same result as insert 'another' to (d)

Below function inserts key s and value v to the prefix tree $t = (v', ts)$:

$$\begin{array}{rcl} insert\ (v', ts)\ \varnothing\ v & = & (Just\ v, ts) \\ insert\ (v', ts)\ s\ v & = & (v', ins\ ts) \end{array} \tag{6.10}$$

If the key s is empty, we overwrite the value to v; otherwise, we call ins to examine the sub-trees and their prefixes.

$$\begin{array}{rcl} ins\ \varnothing & = & [s \mapsto (Just\ v, \varnothing)] \\ ins\ (s' \mapsto t) : ts' & = & \begin{cases} match\ s\ s' : & (branch\ s\ v\ s'\ t) : ts' \\ otherwise : & (s' \mapsto t) : ins\ ts' \end{cases} \end{array} \tag{6.11}$$

If there is no sub-tree in the node, then we create a leaf of v as the single sub-tree, and map s to it; otherwise, for each sub-tree mapping $s' \mapsto t$, we compare s' with s. If they have common prefix (tested by the *match* function), then we *branch* out new sub-tree. We define two lists matching if they have common prefix:

$$\begin{array}{rcl} match\ \varnothing\ B & = & True \\ match\ A\ \varnothing & = & True \\ match\ (a : as)\ (b : bs) & = & a = b \end{array} \tag{6.12}$$

To extract the longest common prefix of two lists A and B, we define a function $(C, A', B') = lcp\ A\ B$, where $C \mathbin{+\!\!+} A' = A$ and $C \mathbin{+\!\!+} B' = B$ hold. If either A or B is empty, or their first elements are different, then the common prefix $C = \varnothing$; otherwise, we recursively extract the longest common prefix from the rest lists, and preprend the head element:

$$
\begin{array}{rcl}
lcp\ \varnothing\ B & = & (\varnothing, \varnothing, B) \\
lcp\ A\ \varnothing & = & (\varnothing, A, \varnothing) \\
lcp\ (a:as)\ (b:bs) & = & \begin{cases} a \neq b : & (\varnothing, a:as, b:bs) \\ otherwise : & (a:cs, as', bs') \end{cases}
\end{array}
\tag{6.13}
$$

where $(cs, as', bs') = lcp\ as\ bs$ in the recursive case. Function $branch\ A\ v\ B\ t$ takes two keys A, B, a value v, and a tree t. It extracts the longest common prefix C from A and B, maps it to a new branch node, and assign sub-trees:

$$
\begin{array}{l}
branch\ A\ v\ B\ t = \\
lcp\ A\ B = \begin{cases} (C, \varnothing, B') : & (C, (Just\ v, [B' \mapsto t])) \\ (C, A', \varnothing) : & (C, insert\ t\ A'\ v) \\ (C, A', B') : & (C, (Nothing, [A' \mapsto (Just\ v, \varnothing), B' \mapsto t])) \end{cases}
\end{array}
\tag{6.14}
$$

If A is the prefix of B, then A is mapped to the node of v, and the remaining list is re-mapped to t, which is the single sub-tree in the branch; if B is the prefix of A, then we recursively insert the remaining list and the value to t; otherwise, we create a leaf node of v put it together with t as the two sub-trees of the branch. The following example program implements the *insert* algorithm:

```haskell
insert (PrefixTree _ ts) [] v = PrefixTree (Just v) ts
insert (PrefixTree v' ts) k v = PrefixTree v' (ins ts) where
    ins [] = [(k, leaf v)]
    ins ((k', t) : ts) | match k k' = (branch k v k' t) : ts
                       | otherwise  = (k', t) : ins ts

leaf v = PrefixTree (Just v) []

match [] _ = True
match _ [] = True
match (a:_) (b:_) = a == b

branch a v b t = case lcp a b of
  (c, [], b') -> (c, PrefixTree (Just v) [(b', t)])
  (c, a', []) -> (c, insert t a' v)
  (c, a', b') -> (c, PrefixTree Nothing [(a', leaf v), (b', t)])

lcp [] bs = ([], [], bs)
lcp as [] = ([], as, [])
lcp (a:as) (b:bs) | a /= b = ([], a:as, b:bs)
                  | otherwise = (a:cs, as', bs') where
                      (cs, as', bs') = lcp as bs
```

We can eliminate the recursion to implement the *insert* algorithm in loops.

```
function INSERT(T, k, v)
    if T = NIL then
        T ← EMPTY-NODE
    p ← T
    loop
        match ← FALSE
        for each s_i ↦ T_i in SUB-TREES(p) do
            if k = s_i then
                VALUE(T_i) ← v                ▷ Overwrite
                return T
```

$c \leftarrow$ LCP(k, s_i)
$k_1 \leftarrow k - c$, $k_2 \leftarrow s_i - c$
if $c \neq$ NIL **then**
$match \leftarrow$ TRUE
if $k_2 =$ NIL **then** ▷ s_i is prefix of k
$p \leftarrow T_i$, $k \leftarrow k_1$
break
else ▷ Branch out a new leaf
ADD(SUB-TREES(p), $c \mapsto$ BRANCH(k_1, LEAF(v), k_2, T_i))
DELETE(SUB-TREES(p), $s_i \mapsto T_i$)
return T
if not $match$ **then** ▷ Add a new leaf
ADD(SUB-TREES(p), $k \mapsto$ LEAF(v))
break
return T

Function LCP extracts the longest common prefix from two lists.

function LCP(A, B)
$i \leftarrow 1$
while $i \leq |A|$ and $i \leq |B|$ and $A[i] = B[i]$ **do**
$i \leftarrow i + 1$
return $A[1...i-1]$

There is a special case in BRANCH(s_1, T_1, s_2, T_2). If s_1 is empty, the key to be insert is some prefix. We set T_2 as the sub-tree of T_1. Otherwise, we create a new branch node and set T_1 and T_2 as the two sub-trees.

function BRANCH(s_1, T_1, s_2, T_2)
if $s_1 =$ NIL **then**
ADD(SUB-TREES(T_1), $s_2 \mapsto T_2$)
return T_1
$T \leftarrow$ EMPTY-NODE
SUB-TREES$(T) \leftarrow \{s_1 \mapsto T_1, s_2 \mapsto T_2\}$
return T

Although the prefix tree improves the space efficiency of trie, it is still bound to $O(mn)$, where n is the length of the key, and m is the size of the element set.

6.4.3 Look up

When look up a key k, we start from the root. If $k = \varnothing$ is empty, then return the root value as the result; otherwise, we examine the sub-tree mappings, locate the one $s_i \mapsto t_i$, such that s_i is some prefix of k, then recursively look up $k - s_i$ in sub-tree t_i. If there does not exist s_i as the prefix of k, then there is no such key in the prefix tree.

$$\begin{array}{rcl} lookup\ \varnothing\ (v, ts) & = & v \\ lookup\ k\ (v, ts) & = & find\ ((s, t) \mapsto s \sqsubseteq k)\ ts\ = \\ & & \begin{cases} Nothing: & Nothing \\ Just\ (s, t): & lookup\ (k - s)\ t \end{cases} \end{array} \tag{6.15}$$

Where $A \sqsubseteq B$ means list A is prefix of B. Function $find$ is defined in chapter 1, which searches element in a list with a given predication. Below example program implements the look up algorithm.

```
lookup [] (PrefixTree v _) = v
lookup ks (PrefixTree v ts) =
```

```
case find (λ(s, t) → s `isPrefixOf` ks) ts of
  Nothing → Nothing
  Just (s, t) → lookup (drop (length s) ks) t
```

The prefix testing is linear to the length of the list, the *lookup* algorithm is bound to $O(mn)$ time, where m is the size of the element set, and n is the length of the key. We skip the imperative implementation, and leave it as the exercise.

Exercise 6.4

1. Eliminate the recursion to implement the prefix tree *lookup* purely with loops

6.5 Applications of trie and prefix tree

We can use trie and prefix tree to solve many interesting problems, like implement a dictionary, populate candidate inputs, and realize the textonym input method. Different from the industry implementation, we give the examples to illustrate the ideas of trie and prefix tree.

6.5.1 Dictionary and input completion

As shown in figure 6.11, when user enters some characters, the dictionary application searches the library, populates a list of candidate words or phrases that start from what input.

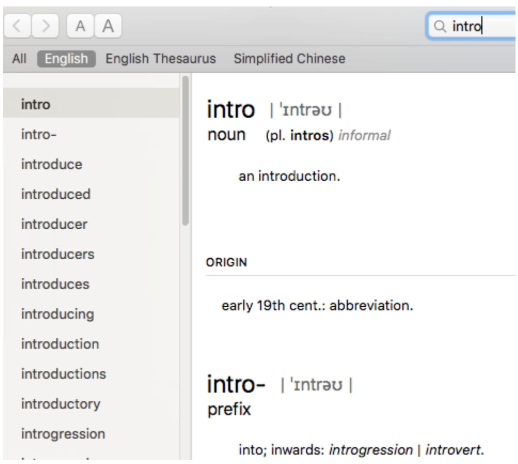


Figure 6.11: A dictionary application

A dictionary can contain hundreds of thousands words. It's expensive to perform a complete search. Commercial dictionaries adopt varies engineering approach, like caching, indexing to speed up search. Similarly, figure 6.12 shows a smart text input component. When type some characters, it populates a candidate lists, with all items starting with the input string.

Both examples give the 'auto-completion' functionality. We can implement it with prefix tree. For illustration purpose, we limit to English characters, and set a upper bound n for the number of candidates. A dictionary stores key-value pairs, where the key is English word or phrase, the value is the corresponding meaning and explanation. When user input string s, we look up the prefix tree for all keys start with s. If s is empty

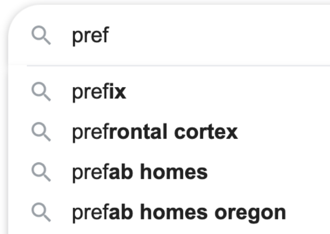


Figure 6.12: A smart text input component

we expand all sub-trees till reach to n candidates; otherwise, we locate the sub-tree from the mapped key, and look up recursively. In the environment supports lazy evaluation, we can expand all candidates, and take the first n on demand: $take\ n\ (startsWith\ s\ t)$, where t is the prefix tree.

$$
\begin{array}{rcl}
startsWith\ \varnothing\ (Nothing, ts) & = & enum\ ts \\
startsWith\ \varnothing\ (Just\ x, ts) & = & (\varnothing, x) : enum\ ts \\
startsWith\ s\ (v, ts) & = & find\ ((k, t) \mapsto s \sqsubseteq k \text{ or } k \sqsubseteq s)\ ts = \\
\multicolumn{3}{l}{\begin{cases} Nothing: & \varnothing \\ Just\ (k, t): & [(k \mathbin{+\!\!+} a, b) | (a, b) \in startsWith\ (s - k)\ t] \end{cases}}
\end{array}
\tag{6.16}
$$

Given a prefix s, function $startsWith$ searches all candidates in the prefix tree starts with s. If s is empty, it enumerates all sub-trees, and prepend $(\varnothing, x)$ for none empty value x in the root. Function $enum\ ts$ is defined as:

$$
enum = concatMap\ (k, t) \mapsto [(k \mathbin{+\!\!+} a, b) | (a, b) \in startsWith\ \varnothing\ t]
\tag{6.17}
$$

Where *concatMap* (also known as *flatMap*) is an important concept for list computation. Literally, it results like firstly map on each element, then concatenate the result together. It's typically realized with 'build-foldr' fusion law to eliminate the intermediate list overhead. (see chapter 5 in my book *Isomorphism – mathematics of programming*) If the input prefix s is not empty, we examine the sub-tree mappings, for each list and sub-tree pair (k, t), if either s is prefix of k or vice versa, we recursively expand t and prepend k to each result key; otherwise, s does not match any sub-trees, hence the result is empty. Below example program implements this algorithm.

```
startsWith [] (PrefixTree Nothing ts) = enum ts
startsWith [] (PrefixTree (Just v) ts) = ([], v) : enum ts
startsWith k (PrefixTree _ ts) =
  case find (λ(s, t) → s `isPrefixOf` k || k `isPrefixOf` s) ts of
    Nothing → []
    Just (s, t) → [(s ++ a, b) |
                          (a, b) ← startsWith (drop (length s) k) t]

enum = concatMap (λ(k, t) → [(k ++ a, b) | (a, b) ← startsWith [] t])
```

We can also realize the algorithm STARTS-WITH(T, k, n) imperatively. From the root, we loop on every sub-tree mapping $k_i \mapsto T_i$. If k is the prefix for any sub-tree T_i, we expand all things in it up to n items; if k_i is the prefix of k, we then drop that prefix, update the key to $k - k_i$, then search T_i for this new key.

```
function STARTS-WITH(T, k, n)
    if T = NIL then
        return NIL
    s ← NIL
```

```
    repeat
        match ← FALSE
        for k_i ↦ T_i in SUB-TREES(T) do
            if k is prefix of k_i then
                return EXPAND(s ++ k_i, T_i, n)
            if k_i is prefix of k then
                match ← TRUE
                k ← k − k_i                          ▷ drop the prefix
                T ← T_i
                s ← s ++ k_i
                break
    until not match
    return NIL
```

Where function EXPAND(s, T, n) populates n results from T and prepand s to each key. We implement it with 'breadth first search' method (see section 14.3):

```
function EXPAND(s, T, n)
    R ← NIL
    Q ← [(s, T)]
    while |R| < n and Q ≠ NIL do
        (k, T) ← POP(Q)
        v ← VALUE(T)
        if v ≠ NIL then
            INSERT(R, (k, v))
        for k_i ↦ T_i in SUB-TREES(T) do
            PUSH(Q, (k ++ k_i, T_i))
```

6.5.2 Predictive text input

Before 2010, most mobile phones had a small keypad as shown in 6.13, called ITU-T keypad. It maps a digit to 3 - 4 characters. For example, when input word 'home', one can press keys in below sequence:

Figure 6.13: The mobile phone ITU-T keypad.

1. Press key '4' twice to enter 'h';
2. Press key '6' three times to enter 'o';
3. Press key '6' to enter 'm';
4. Press key '3' twice to enter 'e';

A smarter input method allows to press less keys:

1. Press key sequence '4', '6', '6', '3', the word 'home' appears as a candidate;
2. Press key '*' to change to next candidate, word 'good' appears;
3. Press key '*' again for another candidate, word 'gone' appears;
4. ...

This is called predictive input, or abbreviated as 'T9'[25],[26]. We can realize it by storing the word dictionary in a prefix tree. The commercial implementations uses multiple layers of caches/index in both memory and file system. We simplify it as an example of prefix tree application. First, we need define the digit key mappings:

$$M_{T9} = \{ \quad \begin{array}{l} 2 \mapsto \texttt{"abc"}, 3 \mapsto \texttt{"def"}, 4 \mapsto \texttt{"ghi"}, \\ 5 \mapsto \texttt{"jkl"}, 6 \mapsto \texttt{"mno"}, 7 \mapsto \texttt{"pqrs"}, \\ 8 \mapsto \texttt{"tuv"}, 9 \mapsto \texttt{"wxyz"} \quad \} \end{array} \tag{6.18}$$

$M_{T9}[i]$ gives the corresponding characters for digit i. We can also define the reversed mapping from a character back to digit.

$$M_{T9}^{-1} = concatMap\ ((d, s) \mapsto [(c, d) | c \in s])\ M_{T9} \tag{6.19}$$

Given a string, we can convert it to a sequence of digits by looking up M_{T9}^{-1}.

$$digits(s) = [M_{T9}^{-1}[c] | c \in s] \tag{6.20}$$

For any character does not belong [`a..z`], we map it to a special key `'#'` as fallback. Below example program defines the above two mappings.

```
mapT9 = Map.fromList [('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)
```

Suppose we already build the prefix tree (v, ts) from all words in a dictionary. We need change the above auto completion algorithm to process digit string ds. For every sub-tree mappings $(s \mapsto t) \in ts$, we convert the prefix s to $digits(s)$, check if it matches to ds (either one is the prefix of the other). There can be multiple sub-trees match ds as:

$$pfx = [(s, t) | (s \mapsto t) \in ts, digits(s) \sqsubseteq ds\ or\ ds\ \sqsubseteq digits(s)]$$

$$\begin{array}{rcl} find_{T9}\ t\ \varnothing & = & [\varnothing] \\ find_{T9}\ (v, ts)\ ds & = & concatMap\ find\ pfx \end{array} \tag{6.21}$$

For each mapping (s, t) in pfx, function $find$ recursively lookup the remaining digits ds' in t, where $ds' = drop\ |s|\ ds$, then prepend s to every candidate. However, the length may exceeds the number of digits, we need cut and only take $n = |ds|$ characters:

$$find\ (s, t) = [take\ n\ (s \mathbin{+\!\!+} s_i) | s_i \in find_{T9}\ t\ ds'] \tag{6.22}$$

The following example program implements the predictive input look up algorithm:

```
findT9 _ [] = [[]]
findT9 (PrefixTree _ ts) k = concatMap find pfx where
  find (s, t) = map (take (length k) ∘ (s++)) $ findT9 t (drop (length s) k)
  pfx = [(s, t) | (s, t) ← ts, let ds = digits s in
              ds `isPrefixOf` k || k `isPrefixOf` ds]
```

To realize the predictive text input imperatively, we can perform breadth first search with a queue Q of tuples (prefix, D, t). Every tuple records the possible *prefix* searched so far; the remaining digits D to be searched; and the sub-tree t we are going to search. Q is initialized with the empty prefix, the whole digits sequence, and the root. We repeatedly pop the tuple from the queue, and examine the sub-tree mappings. for every mapping $(s \mapsto T')$, we convert s to $digits(s)$. If D is prefix of it, then we find a candidate. We append s to *prefix*, and record it in the result. If $digits(s)$ is prefix of D, we need further search the sub-tree T'. We create a new tuple of $(\mathit{prefix} \mathbin{+\!\!+} s, D', T')$, where D' is the remaining digits to be searched. Then push this new tuple back to the queue.

function LOOK-UP-T9(T, D)
    $R \leftarrow$ NIL
    if $T =$ NIL or $D =$ NIL **then**
        return R
    $n \leftarrow |D|$
    $Q \leftarrow \{(\text{NIL}, D, T)\}$
    while $Q \neq$ NIL **do**
        $(\mathit{prefix}, D, T) \leftarrow$ POP(Q)
        for $(s \mapsto T') \in$ SUB-TREES(T) **do**
            $D' \leftarrow$ DIGITS(s)
            if $D' \sqsubset D$ **then** ▷ D' is prefix of D
                APPEND$(R, (\mathit{prefix} \mathbin{+\!\!+} s)[1..n])$ ▷ limit the length to n
            else if $D \sqsubset D'$ **then**
                PUSH$(Q, (\mathit{prefix} \mathbin{+\!\!+} s, D - D', T'))$
    return R

Exercise 6.5

1. Implement the auto-completion and predictive text input with trie.
2. How to ensure the candidates in lexicographic order in the auto-completion and predictive text input program? What's the performance change accordingly?
3. In the environment without lazy evaluation support, how to return the first n candidates on-demand?

6.6 Summary

We start from integer trie and prefix tree. By turning the integer key to binary format, we re-used binary tree to realize the integer based map data structure. Then extend the key from integer to generic list, and limit the list element to finite set. Particularly for alphabetic strings, the generic trie and prefix tree can be used as tools to manipulate the text information. We give example applications about auto-completion and predictive text input. as another instance of radix tree, the suffix tree is closely related to trie and prefix tree used in text, and DNA processing.

6.7 Appendix: Example programs

Definition of integer binary trie:

```
data IntTrie<T> {
    IntTrie<T> left = null
    IntTrie<T> right = null
    Optional<T> value = Optional.Nothing
}
```

The following example *insert* program uses bit-wise operation to test even/odd, and shift the bit to right:

```
IntTrie<T> insert(IntTrie<T> t, Int key,
                  Optional<T> value = Optional.Nothing) {
    if t == null then t = IntTrie<T>()
    p = t
    while key ≠ 0 {
        if key & 1 == 0 {
            p = if p.left == null then IntTrie<T>() else p.left
        } else {
            p = if p.right == null then IntTrie<T>() else p.right
        }
        key = key >> 1
    }
    p.value = Optional.of(value)
    return t
}
```

Definition of integer prefix tree:

```
data IntTree<T> {
    Int key
    T value
    Int prefix
    Int mask = 1
    IntTree<T> left = null
    IntTree<T> right = null

    IntTree(Int k, T v) {
        key = k, value = v, prefix = k
    }

    bool isLeaf = (left == null and right == null)

    Self replace(IntTree<T> x, IntTree<T> y) {
        if left == x then left = y else right = y
    }

    bool match(Int k) = maskbit(k, mask) == prefix
}

Int maskbit(Int x, Int mask) = x & (~(mask - 1))
```

Insert key-value to integer prefix tree.

```
IntTree<T> insert(IntTree<T> t, Int key, T value) {
    if t == null then return IntTree(key, value)
    node = t
    Node<T> parent = null
    while (not node.isLeaf()) and node.match(key) {
        parent = node
        node = if zero(key, node.mask) then node.left else node.right
    }
    if node.isleaf() and key == node.key {
```

```
        node.value = value
    } else {
        p = branch(node, IntTree(key, value))
        if parent == null then return p
        parent.replace(node, p)
    }
    return t
}

IntTree<T> branch(IntTree<T> t1, IntTree<T> t2) {
    var t = IntTree<T>()
    (t.prefix, t.mask) = lcp(t1.prefix, t2.prefix)
    (t.left, t.right) = if zero(t1.prefix, t.mask) then (t1, t2)
                        else (t2, t1)
    return t
}

bool zero(int x, int mask) = (x & (mask >> 1) == 0)

Int lcp(Int p1, Int p2) {
    Int diff = p1 ^ p2
    Int mask = 1
    while diff ≠ 0 {
        diff = diff >> 1
        mask = mask << 1
    }
    return (maskbit(p1, mask), mask)
}
```

Definition of trie and the insert program:

```
data Trie<K, V> {
  Optional<V> value = Optional.Nothing
  Map<K, Trie<K, V>> subTrees = Map.empty()
}

Trie<K, V> insert(Trie<K, V> t, [K] key, V value) {
    if t == null then t = Trie<K, V>()
    var p = t
    for c in key {
        if p.subTrees[c] == null then p.subTrees[c] = Trie<K, V>()
        p = p.subTrees[c]
    }
    p.value = Optional.of(value)
    return t
}
```

Definition of Prefix Tree and insert program:

```
data PrefixTree<K, V> {
    Optional<V> value = Optional.Nothing
    Map<[K], PrefixTree<K, V>> subTrees = Map.empty()

    Self PrefixTree(V v) {
        value = Optional.of(v)
    }
}

PrefixTree<K, V> insert(PrefixTree<K, V> t, [K] key, V value) {
    if t == null then t = PrefixTree()
    var node = t
    loop {
        bool match = false
        for var (k, tr) in node.subtrees {
            if key == k {
                tr.value = value
```

```
                return t
            }
            prefix, k1, k2 = lcp(key, k)
            if prefix ≠ [] {
                match = true
                if k2 == [] {
                    node = tr
                    key = k1
                    break
                } else {
                    node.subtrees[prefix] = branch(k1, PrefixTree(value),
                                                   k2, tr)
                    node.subtrees.delete(k)
                    return t
                }
            }
        }
        if !match {
            node.subtrees[key] = PrefixTree(value)
            break
        }
    }
    return t
}
```

The longest common prefix `lcp` and `branch` example programs.

```
([K], [K], [K]) lcp([K] s1, [K] s2) {
    j = 0
    while j < length(s1) and j < length(s2) and s1[j] == s2[j] {
        j = j + 1
    }
    return (s1[0..j-1], s1[j..], s2[j..])
}

PrefixTree<K, V> branch([K] key1, PrefixTree<K, V> tree1,
                        [K] key2, PrefixTree<K, V> tree2) {
    if key1 == []:
        tree1.subtrees[key2] = tree2
        return tree1
    t = PrefixTree()
    t.subtrees[key1] = tree1
    t.subtrees[key2] = tree2
    return t
}
```

Populate multiple candidates, they share the common prefix

```
[([K], V)] startsWith(PrefixTree<K, V> t, [K] key, Int n) {
    if t == null then return []
    [T] s = []
    repeat {
        bool match = false
        for var (k, tr) in t.subtrees {
            if key.isPrefixOf(k) {
                return expand(s ++ k, tr, n)
            } else if k.isPrefixOf(key) {
                match = true
                key = key[length(k)..]
                t = tr
                s = s ++ k
                break
            }
        }
    } until not match
    return []
```

```
}

[([K], V)] expand([K] s, PrefixTree<K, V> t, Int n) {
    [([K], V)] r = []
    var q = Queue([(s, t)])
    while length(r) < n and !q.isEmpty() {
        var (s, t) = q.pop()
        v = t.value
        if v.isPresent() then r.append((s, v.get()))
        for k, tr in t.subtrees {
            q.push((s ++ k, tr))
        }
    }
    return r
}
```

Predictive text input lookup

```
var T9MAP={'2':"abc", '3':"def", '4':"ghi", '5':"jkl", λ
       '6':"mno", '7':"pqrs", '8':"tuv", '9':"wxyz"}

var T9RMAP = { c : d for var (d, cs) in T9MAP for var c in cs }

string digits(string w) = ''.join([T9RMAP[c] for c in w])

[string] lookupT9(PrefixTree<char, V> t, string key) {
    if t == null or key == "" then return []
    res = []
    n = length(key)
    q = Queue(("", key, t))
    while not q.isEmpty() {
        (prefix, key, t) = q.pop()
        for var (k, tr) in t.subtrees {
            ds = digits(k)
            if key.isPrefixOf(ds) {
                res.append((prefix ++ k)[:n])
            } else if ds.isPrefixOf(key) {
                q.append((prefix ++ k, key[length(k)..], tr))
            }
        }
    }
    return res
}
```

Chapter 7

B-Tree

7.1 Introduction

The integer prefix tree in previous chapter gives a way to encode the information in the edge of the binary tree. Another way to extend the binary search tree is to increase the sub-trees from 2 to k. B-tree is such a data structure, that can be considered as a generic form of k-ary search tree. It is also developed to be self-balanced[39]. B-tree is widely used in computer file system (some are based on B+ tree, an extension of B-tree) and database system. Figure 7.1 gives an example B-tree, we can find the difference and similarity between B-tree and binary search tree.

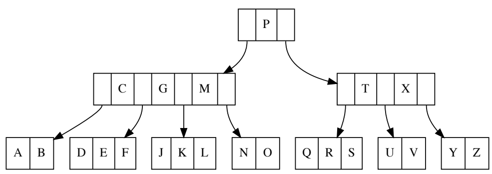


Figure 7.1: A B-Tree

A binary search tree is either empty or contains a key k and two sub-trees l and r. Every key in l is less than k, while k is less than every key in r:

$$\forall\ x \in l, y \in r \Rightarrow x < k < y \tag{7.1}$$

Extend to multiple keys and sub-trees, we obtain the B-tree. A B-tree is either empty or contains n keys and $n+1$ sub-trees, each sub-tree is also a B-Tree. We denote these keys and sub-trees as $k_1, k_2, ..., k_n$ and $t_1, t_2, ..., t_n, t_{n+1}$, as shown in figure 7.2.

C[1]	K[1]	C[2]	K[2]	...	C[n]	K[n]	C[n+1]

Figure 7.2: A B-Tree node

For every node, the keys and sub-trees satisfy the following rules:

- Keys are in ascending order: $k_1 < k_2 < ... < k_n$;
- For every key k_i, all keys in sub-tree t_i are less than it, while k_i is less than every key in sub-tree t_{i+1}:

$$\forall\, x_i \in t_i, i = 0, 1, ..., n \;\Rightarrow x_1 < k_1 < x_2 < k_2 < ... < x_n < k_n < x_{n+1} \tag{7.2}$$

Leaf node has no sub-tree. There can be optional values bound to the keys in B-tree node. We skip the values for simplicity. Let the type of keys be K, the type of the B-tree is $BTree\ K$, or denoted as `BTree<K>`. On top of it, we also need define a set of self-balance rules:

1. All leaves have the same depth;

2. Let d be the *minimum degree* number of a B-tree, such that each node:

   - has at most $2d - 1$ keys;
   - has at least $d - 1$ keys, except for the root;

In summary:

$$d - 1 \leq |keys(t)| \leq 2d - 1 \tag{7.3}$$

We next prove that a B-tree satisfying these rules is always balanced.

Proof. Consider a B-tree of n keys. The minimum degree $d \geq 2$. Let the height be h. All the nodes have at least $d - 1$ keys except for the root. The root contains at least 1 key. There are at least 2 nodes at depth 1, at least $2d$ nodes at depth 2, at least $2d^2$ nodes at depth 3, ..., at least $2d^{h-1}$ nodes at depth h. Multiply all nodes with $d - 1$ except for the root, the total number of keys satisfies the following:

$$\begin{aligned} n &\geq 1 + (d-1)(2 + 2d + 2d^2 + ... + 2d^{h-1}) \\ &= 1 + 2(d-1)\sum_{k=0}^{h-1} d^k \\ &= 1 + 2(d-1)\frac{d^h - 1}{d - 1} \\ &= 2d^h - 1 \end{aligned} \tag{7.4}$$

It limits the tree height with logarithm of the number of keys.

$$h \leq \log_d \frac{n+1}{2} \tag{7.5}$$

□

Hence B-tree is balanced. The simplest B-tree is called 2-3-4 tree, where $d = 2$. Every node except for the root contains 2, 3, or 4 sub-trees. Essentially, a red-black tree can be mapped to a 2-3-4 tree. For a none empty B-tree of degree d, we denote it as $(d, (ks, ts))$, where ks are the keys, ts are the sub-trees. Below example program defines the B-tree.

```
data BTree a = BTree [a] [BTree a]
```

The empty node is in the form of $(\varnothing, \varnothing)$ or `BTree [] []`. Instead of storing d in every node, we pass it together with B-tree t as a pair (d, t).

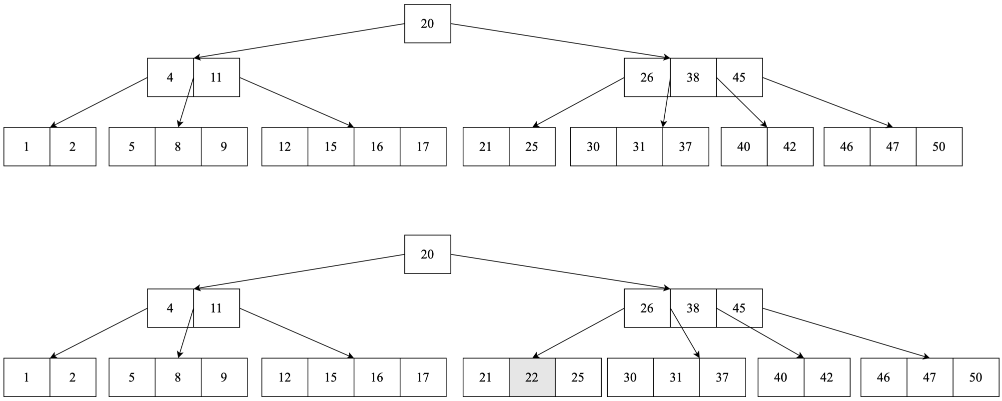


Figure 7.3: Insert 22 to a 2-3-4 tree. $22 > 20$, go to the right sub-tree; next as $22 < 26$, go to the first sub-tree; finally, $21 < 22 < 25$, and the leaf is not full.

7.2 Insert

The idea is similar to the binary search tree. While we need deal with multiple keys and sub-trees. When insert key x to B-tree t, starting from the root, we examine the keys in the node to locate a position[1] where all keys on the left are less than x, while the rest keys on the right are greater than x. If the node is a leaf, and it is not full ($|keys(t)| < 2d-1$), we insert x at this position. Otherwise, this position points to a sub-tree t', we recursively insert x to t'.

As an example, consider the 2-3-4 tree in figure 7.3. when insert $x = 22$, because $20 < 22$, we next examine the sub-tree on the right, which contains 26, 38, 45. Since $22 < 26$, we next go to the first sub-tree containing 21 and 25. This is a leaf, and it is not full. Hence we insert 22 to this node.

However, if there are $2d-1$ keys in the leaf, we will break the B-tree rules after insert x, as the node will be too 'full'. For the same B-tree in figure 7.3, we'll meet this issue when insert 18. There are two solutions: insert then split, and split before insert.

7.2.1 Insert then split

We can adopt the similar 'insert then fix' method for the red-black tree. First, we insert the key to the proper ordering position without considering the B-tree balance rules. As the next step, if the new tree violates the balance rules, we perform a recursive bottom-up fixing by splitting the overly full node. We need define the function to test whether a given node satisfies the minimum degree constraint or not.

$$\begin{cases} full\ d\ (ks, ts) & = |ks| > 2d-1 \\ low\ d\ (ks, ts) & = |ks| < d-1 \end{cases} \tag{7.6}$$

When the node contains too many keys and sub-trees, we define *split* function to break it into 3 parts at a given position m as shown in figure 7.4:

$$split\ m\ (ks, ts) = ((ks_l, ts_l), k, (ks_r, ts_r)) \tag{7.7}$$

We reuse the list *splitAt* function defined in chapter 1 (Equation 1.45) to realize it.

[1] In fact, it is sufficient to only support less-than and equality. See exercise 1.

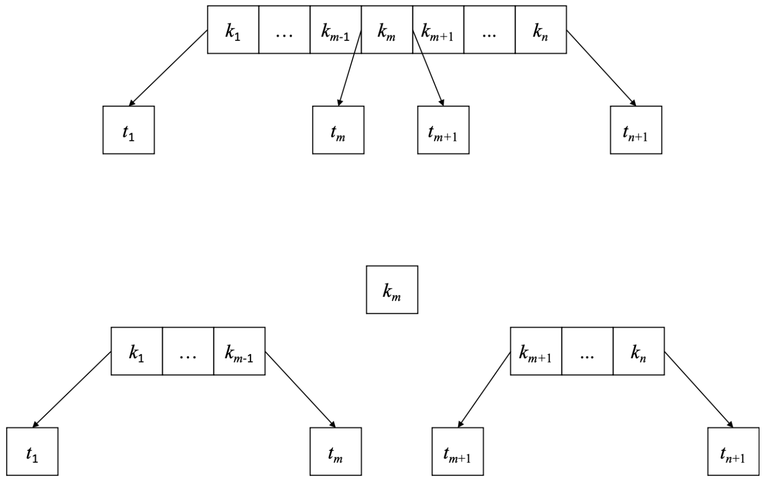


Figure 7.4: Split the node into 3 parts at m

$$\begin{cases} (ks_l, (k : ks_r)) & = splitAt\ (m - 1)\ ks \\ (ts_l, ts_r) & = splitAt\ m\ ts \end{cases}$$

We can define the reversed operation *unsplit* to combine the 3 parts back into a B-tree node.

$$unsplit\ (ks_l, ts_l)\ k\ (ks_r, ts_r) = (ks_l \mathbin{+\!\!+} [k] \mathbin{+\!\!+} ks_r, ts_l \mathbin{+\!\!+} ts_r) \tag{7.8}$$

Below function first inserts x to the tree t, then calls fix to resume the B-tree balance rules with the given degree d.

$$insert\ x\ (d, t) = fix\ (d, ins\ t) \tag{7.9}$$

After ins, if the root contains too many keys, function fix calls *split* to break it and build a new root.

$$fix\ (d, t) = \begin{cases} full\ d\ t : & (d, ([k], [l, r])), \text{where } (l, k, r) = split\ d\ t \\ otherwise : & (d, t) \end{cases} \tag{7.10}$$

ins need handle two cases: for leaf node, we can reuse the list ordered *insert* function defined in chapter 1 (Equation 1.11); otherwise, we need find the position to recursively insert to sub-tree. To do that, we define a *partition* function as:

$$partition\ x\ (ks, ts) = (l, t', r) \tag{7.11}$$

Where $l = (ks_l, ts_l)$ and $r = (ks_r, ts_r)$. It further uses the list partition function *span* defined in chapter 1 (Equation 1.47):

$$\begin{cases} (ks_l, ks_r) & = span\ (< x)\ ks \\ (ts_l, (t' : ts_r)) & = splitAt\ |ks_l|\ ts \end{cases}$$

As such, we separate all the keys and sub-trees less than x on the left as l, and those greater than x on the right as r. The last sub-tree that less than x is extracted as t'. We then recursively insert x to t', as shown in figure 7.5.

$$\begin{array}{lcll} ins\ (ks, \varnothing) & = & (insert_L\ x\ ks, \varnothing) & \text{list insert for leaf} \\ ins\ (ks, ts) & = & balance\ d\ l\ (ins\ t')\ r & \text{where } (l, t', r) = partition\ x\ t \end{array} \tag{7.12}$$

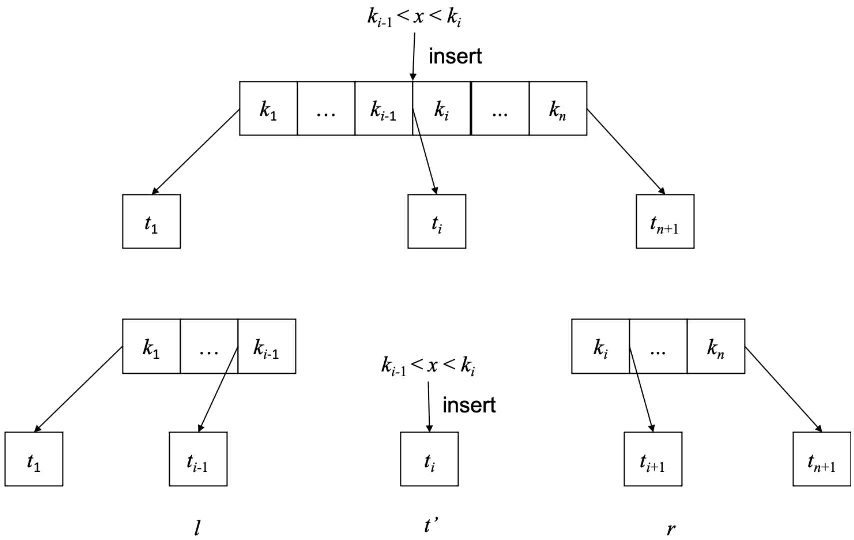


Figure 7.5: partition a node with x

After insert x to t', it may contains too many keys that violates B-tree rules. We define function *balance* to recursively recover B-tree rules by splitting sub-tree.

$$balance\ d\ (ks_l, ts_l)\ t\ (ks_r, ts_r) = \begin{cases} full\ d\ t : fix_f \\ otherwise : (ks_l \mathbin{+\!\!+} ks_r, ts_l \mathbin{+\!\!+} [t] \mathbin{+\!\!+} ts_r) \end{cases} \tag{7.13}$$

where fix_f splits sub-tree t with degree d as $(t_1, k, t_2) = split\ d\ t$, then combine them to a new B-tree node:

$$fix_f = (ks_l \mathbin{+\!\!+} [k] \mathbin{+\!\!+} ks_r, ts_l \mathbin{+\!\!+} [t_1, t_2] \mathbin{+\!\!+} ts_r) \tag{7.14}$$

The following example program implements *insert* for B-tree.

```
partition x (BTree ks ts) = (l, t, r) where
  l = (ks1, ts1)
  r = (ks2, ts2)
  (ks1, ks2) = span (< x) ks
  (ts1, (t:ts2)) = splitAt (length ks1) ts

split d (BTree ks ts) = (BTree ks1 ts1, k, BTree ks2 ts2) where
  (ks1, k:ks2) = splitAt (d - 1) ks
  (ts1, ts2) = splitAt d ts

insert x (d, t) = fixRoot (d, ins t) where
    ins (BTree ks []) = BTree (List.insert x ks) []
    ins t = balance d l (ins t') r where (l, t', r) = partition x t

fixRoot (d, t) | full d t  = let (t1, k, t2) = split d t in
                               (d, BTree [k] [t1, t2])
               | otherwise = (d, t)

balance d (ks1, ts1) t (ks2, ts2)
    | full d t  = fixFull
    | otherwise = BTree (ks1 ++ ks2) (ts1 ++ [t] ++ ts2)
  where
    fixFull = let (t1, k, t2) = split d t in
               BTree (ks1 ++ [k] ++ ks2) (ts1 ++ [t1, t2] ++ ts2)
```

Figure 7.6 shows the example B-trees built by repeatedly insert elements from list "GMPXACDEJKNORSTUVYZ".

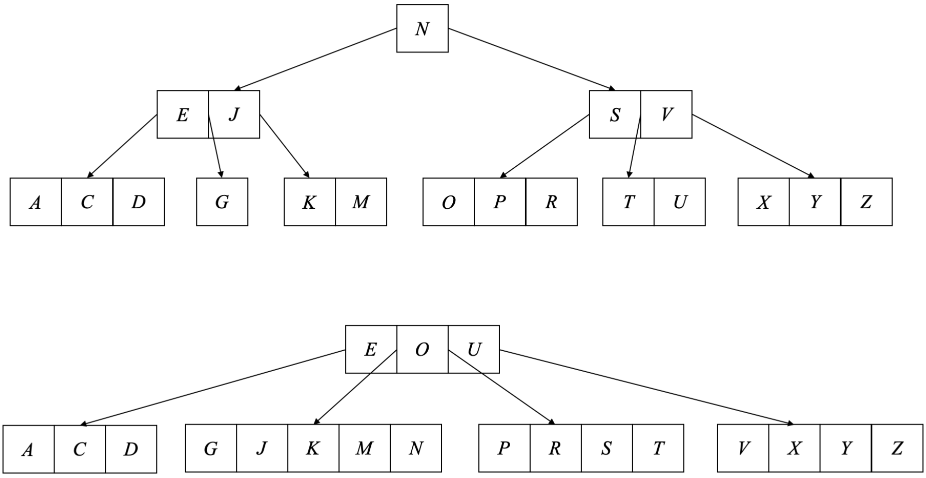


Figure 7.6: Repeatedly insert elements from "GMPXACDEJKNORSTUVYZ". above: $d = 2$ (2-3-4 tree), below: $d = 3$

7.2.2 Split before insert

The second method is to split a node before insertion to prevent it becoming overly full. We often see this method in imperative implementations. When perform top-down recursive insert, if we reach to a node with $2d - 1$ keys, we divide it into 3 parts as shown in figure 7.4, such that each new node has $d - 1$ keys. They will be valid B-tree node after insertion. For node x, let $K(x)$ be the keys, $T(x)$ be the sub-trees. Denote the i-th key of x as $k_i(x)$, the j-th sub-tree as $t_j(x)$. Below algorithm splits the i-th sub-tree of node z:

procedure SPLIT(z, i)
    $d \leftarrow$ DEG(z)
    $x \leftarrow t_i(z)$
    $y \leftarrow$ CREATE-NODE
    $K(y) \leftarrow [k_{d+1}(x), k_{d+2}(x), ..., k_{2d-1}(x)]$
    $K(x) \leftarrow [k_1(x), k_2(x), ..., k_{d-1}(x)]$
    if x is not leaf **then**
        $T(y) \leftarrow [t_{d+1}(x), t_{d+2}(x), ..., t_{2d}(x)]$
        $T(x) \leftarrow [t_1(x), t_2(x), ..., t_d(x)]$
    INSERT-AT$(K(z), i, k_d(x))$
    INSERT-AT$(T(z), i + 1, y)$

When split the node $x = t_i(z)$, we push the d-th key $k_d(x)$ up to the parent node z. If z is already full, the pushing will break B-tree rules. To solve this problem, we need do the top-down check from the root along the path when insert. Split any node with $2d - 1$ keys. Since all parent nodes are processed to be not full, they can accept the additional key pushed up. This method needs one single pass down the tree without any back-tracking. If the root is full, we create a new node, and put the root as it singleton sub-tree. Below is the insert algorithm:

function INSERT(t, k)
    $r \leftarrow t$

```
    if r is full then                                  ▷ root is full
        s ← CREATE-NODE
        T(s) ← [r]
        SPLIT(s, 1)
        r ← s
    return INSERT-NONFULL(r, k)
```

Where INSERT-NONFULL assumes the node r passed in is not full. If r is a leaf, we insert k to the keys based on order (Exercise 3 asks to realize the ordered insert with binary search); otherwise, we locate the position, where $k_i(r) < k < k_{i+1}(r)$. Split the sub-tree $t_i(r)$ if it is full, and go on insert to this sub-tree.

```
function INSERT-NONFULL(r, k)
    n ← |K(r)|
    if r is leaf then
        i ← 1
        while i ≤ n and k > k_i(r) do
            i ← i + 1
        INSERT-AT(K(r), i, k)
    else
        i ← n
        while i > 1 and k < k_i(r) do
            i ← i − 1
        if t_i(r) is full then
            SPLIT(r, i)
            if k > k_i(r) then
                i ← i + 1
        INSERT-NONFULL(t_i(r), k)
    return r
```

This algorithm is recursive. Exercise 2 asks to eliminate the recursion with pure loops. Figure 7.7 gives the result with the same input of "GMPXACDEJKNORSTUVYZ".

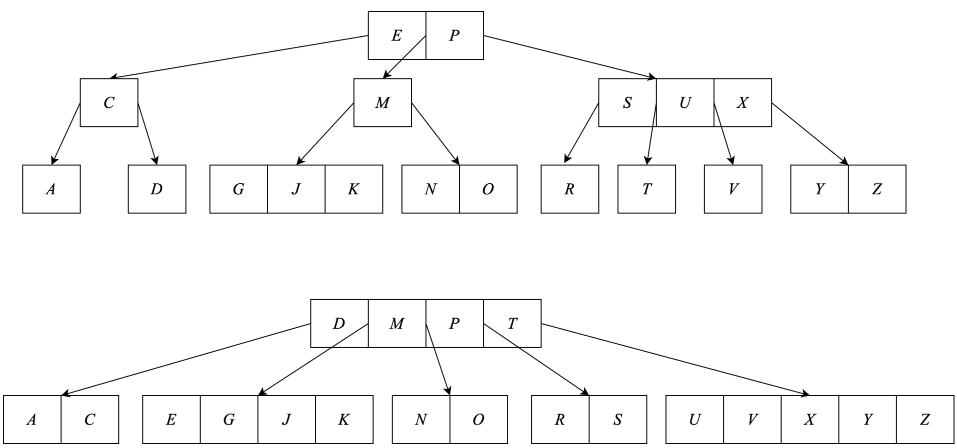


Figure 7.7: Insert from "GMPXACDEJKNORSTUVYZ". up: $d = 2$, 2-3-4 tree; bottom: $d = 3$.

7.2.3 Paired lists

When use list to store ordered keys, we always start from the first key, and scan the list to find the insert position. If the keys are stored in array, we can improve it with binary search. Can we start somewhere in the node, go left or right depending on the order of keys? One idea is to separate the B-tree node into three parts: left l, a sub-tree t', and right r. Where left and right are lists of pairs, each pair contains a key and a sub-tree: (k_i, t_i). However, l is reversed. In other words, l and r are head-to-head connected by t' as a U-shape shown in figure 7.8. We can move forward and backward both in constant time.

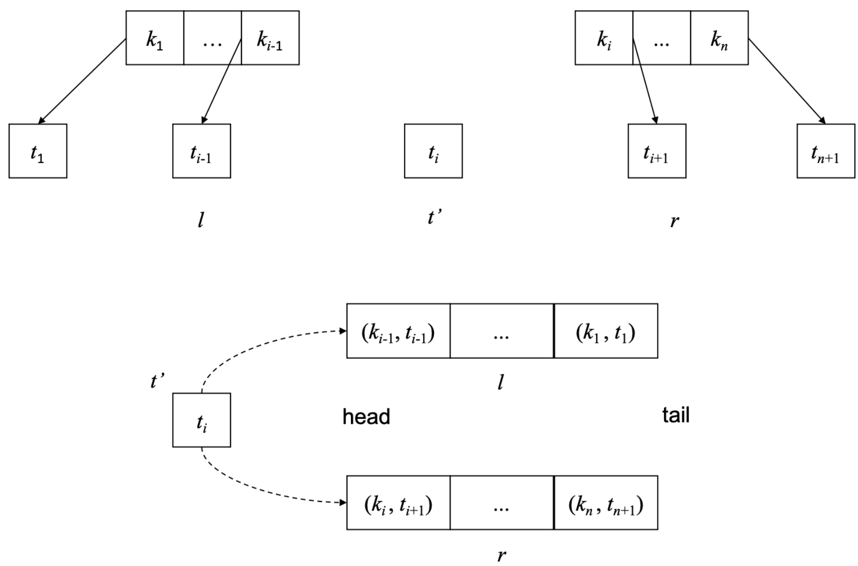


Figure 7.8: Define the B-tree node with a sub-tree and paired lists

Below example program defines B-tree node. It's either empty, or contains 3 parts: the left (key, sub-tree) list in reversed order, a sub-tree, and the right (key, sub-tree) list. We denoted the none empty node as (l, t', r).

```
data BTree a = Empty
             | BTree [(a, BTree a)] (BTree a) [(a, BTree a)]
```

When move to right by a step, we take the first pair (k, t) from r, then form another pair (k, t') in front of l, and replace t' with t. When move to left a step, it is symmetric. Both operations take constant time.

$$
\begin{array}{rcl}
step_l \ ((k, t) : l, t', r) & = & (l, t, (k, t') : r) \\
step_r \ (l, t', (k, t) : r) & = & ((k, t') : l, t, r)
\end{array}
\tag{7.15}
$$

With the left/right moves, we can implement a generic partition function *partition* $p \ t$, that separates the tree t with a given predicate p into 3 parts: left, middle, right: (l, m, r), such that all sub-trees in l and m satisfy p, while the sub-trees in r do not. Let the function

$hd = fst \circ head$, which picks the first pair (a, b) from a list, then extracts a out.

$$
\begin{array}{rcl}
partition\ p\ (\varnothing, m, r) & = & \begin{cases} p(hd(r)) : & partition\ p\ (step_r\ t) \\ otherwise : & (\varnothing, m, r) \end{cases} \\
partition\ p\ (l, m, \varnothing) & = & \begin{cases} (not \circ p)(hd(l)) : & partition\ p\ (step_l\ t) \\ otherwise : & (l, m, \varnothing) \end{cases} \\
partition\ p\ (l, m, r) & = & \begin{cases} p(hd(l)) \text{ and } (not \circ p)(hd(r)) : & (l, m, r) \\ p(hd(r)) : & partition\ p\ (step_r\ t) \\ (not \circ p)(hd(l)) : & partition\ p\ (step_l\ t) \end{cases}
\end{array}
\tag{7.16}
$$

For example, $partition\ (< k)\ t$ moves all keys and sub-trees in t less than k out of the right part. Below example program implements the $partition$ function:

```
partition p t@(BTree [] m r)
  | p (hd r) = partition p (stepR t)
  | otherwise = ([], m, r)
partition p t@(BTree l m [])
  | (not ∘ p) (hd l) = partition p (stepL t)
  | otherwise = (l, m, [])
partition p t@(BTree l m r)
  | p (hd l) && (not ∘ p) (hd r) = (l, m, r)
  | p (hd r) = partition p (stepR t)
  | (not ∘ p) (hd l) = partition p (stepL t)
```

We can also use $step_l/step_r$ to split a B-tree at position d when it becomes overly full. Let $n = |l|$ be the number of keys/sub-trees of the left part. $f^n(x)$ means repeatedly apply function f to x for n times.

$$
split\ d\ t = \begin{cases} n < d : & sp(step_r^{d-n}(t)) \\ n > d : & sp(step_r^{n-d}(t)) \\ otherwise : & sp(t) \end{cases}
\tag{7.17}
$$

Where sp does the separation work as below:

$$
sp\ (l, t, (k, t') : r) = ((l, t, \varnothing), k, (\varnothing, t', r))
\tag{7.18}
$$

With $partition$ and $split$ defined, we can define B-tree insert algorithm for the paired lists implementation. Firstly, we need modify the low/full testing to count both left and right parts:

$$
\begin{array}{ll}
full\ d\ \varnothing & = False \\
full\ d\ (l, t', r) & = |l| + |r| > 2d - 1
\end{array}
\tag{7.19}
$$

and

$$
\begin{array}{ll}
low\ d\ \varnothing & = False \\
low\ d\ (l, t', r) & = |l| + |r| < d - 1
\end{array}
\tag{7.20}
$$

When insert key x to B-tree t of degree d, we do the recursive insertion, then fix the root if it gets overly full:

$$
insert\ x\ (d, t) = fix\ (d, ins\ t)
\tag{7.21}
$$

Where fix splits the root at d if needed:

$$
fix\ (d, t) = \begin{cases} full\ d\ t : & (d, (\varnothing, t_1, [(k, t_2)] \text{ where } (t_1, k, t_2) = split\ d\ t \\ otherwise : & (d, t) \end{cases}
\tag{7.22}
$$

Function *ins* need handle both $t = \varnothing$, and $t \neq \varnothing$ cases. For empty case, we create a singleton leaf; otherwise, we call $(l, t', r) = partition\ (< x)\ t$ to locate the position for recursive insert:

$$
\begin{array}{rcl}
ins\ \varnothing & = & (\varnothing, \varnothing, [(x, \varnothing)]) \\
ins\ t & = & \begin{cases}
t' = \varnothing : & balance\ d\ l\ \varnothing\ ((x, \varnothing) : r) \\
t' \neq \varnothing : & balance\ d\ l\ (ins\ t')\ r
\end{cases}
\end{array}
\tag{7.23}
$$

Function *balance* examines if the sub-tree t contains too many keys, and splits it.

$$
balance\ d\ l\ t\ r = \begin{cases}
full\ d\ t : & fixFull \\
otherwise : & (l, t, r)
\end{cases}
\tag{7.24}
$$

Where $fixFull = (l, t_1, ((k, t_2) : r)$, and $(t_1, k, t_2) = split\ d\ t$. Below example program implements the insert algorithm:

```
insert x (d, t) = fixRoot (d, ins t) where
  ins Empty = BTree [] Empty [(x, Empty)]
  ins t = let (l, t', r) = partition (< x) t in
    case t' of
      Empty → balance d l Empty ((x, Empty):r)
      _     → balance d l (ins t') r

fixRoot (d, t) | full d t = let (t1, k, t2) = split d t in
                    (d, BTree [] t1 [(k, t2)])
               | otherwise = (d, t)

balance d l t r | full d t = fixFull
                | otherwise = BTree l t r
  where
    fixFull = let (t1, k, t2) = split d t in BTree l t1 ((k, t2):r)

split d t@(BTree l _ _) | n < d = sp $ iterate stepR t !! (d - n)
                        | n > d = sp $ iterate stepL t !! (n - d)
                        | otherwise = sp t
  where
    n = length l
    sp (BTree l t ((k, t'):r)) = (BTree l t [], k, BTree [] t' r)
```

Exercise 7.1

1. Can we use $\leq$ to support duplicated keys in B-Tree?
2. For the 'split then insert' algorithm, eliminate the recursion with loops.
3. We use linear search among keys to find the proper insert position. Improve the imperative implementation with binary search. Is the big-O performance improved?

7.3 Look up

For look up, we can extend from the binary search tree to multiple branches, and obtain the generic B-tree look up solution. There are only two directions when look up the binary search tree: left and right, while, there are multiple ways in B-tree. Consider look up k in B-tree $t = (ks, ts)$, if t is a leaf (ts is empty), then the problem becomes list look up; otherwise, we partition the t with k into three parts: $l = (ks_l, ts_l), t', r = (ks_r, ts_r)$, where all keys in l and sub-tree t' are less then k, and the remaining ($\geq k$) is in r. If the first key in ks_r equals k, then we find the answer; otherwise, we recursive look up in sub-tree t'.

$$
\begin{array}{rcl}
lookup\ k\ (ks, \varnothing) & = & \begin{cases} k \in ks : & Just\ (ks, \varnothing) \\ otherwise : & Nothing \end{cases} \\
lookup\ k\ (ks, ts) & = & \begin{cases} Just\ k = safeHd\ ks_r : & Just\ (ks, ts) \\ otherwise : & lookup\ k\ t' \end{cases}
\end{array}
\tag{7.25}
$$

Where $((ks_l, ts_l), t', (ks_r, ts_r)) = partition\ k\ t$, and

$$
\begin{array}{rcl}
safeHd\ [] & = & Nothing \\
safeHd\ (x:xs) & = & Just\ x
\end{array}
$$

Below example program[2] implements *lookup*.

```
lookup k t@(BTree ks []) = if k `elem` ks then Just t else Nothing
lookup k t = if (Just k) == safeHd ks then Just t
             else lookup k t'  where
  (_, t', (ks, _)) = partition k t
```

For the paired list implementation, the idea is similar. If the tree is not empty, we partition it with the predicate '$< k$'. Then check if the first key in the right part equals to k, or recursively look up the partitioned sub-tree:

$$
\begin{array}{rcl}
lookup\ k\ \varnothing & = & Nothing \\
lookup\ k\ t & = & \begin{cases} Just\ k = safeFst\ (safeHd\ r) : & Just\ (l, t', r) \\ otherwise : & lookup\ k\ t' \end{cases}
\end{array}
\tag{7.26}
$$

Where $(l, t', r) = partition\ (< k)\ t$ for the none empty tree case. *safeFst* applies *fst* function to a 'Maybe' value. Below example program utilizes *fmap* to do this:

```
lookup x Empty = Nothing
lookup x t = let (l, t', r) = partition (< x) t in
  if (Just x) == fmap fst (safeHd r) then Just (BTree l t' r)
  else lookup x t'
```

For the imperative implementation, we start from the root r, find a position i among the keys, such that $k_i(r) \leq k < k_{i+1}(r)$. If $k_i(r) = k$ then return the node r and i as a pair; otherwise, move to sub-tree $t_i(r)$ to go on looking up. If r is a leaf and k is not in the keys, then return nothing. It means k does not exist in the tree.

```
function LOOK-UP(r, k)
    loop
        i ← 1, n ← |K(r)|
        while i ≤ n and k > k_i(r) do
            i ← i + 1
        if i ≤ n and k = k_i(r) then
            return (r, i)
        if r is leaf then
            return Nothing                      ▷ k does not exist
        else
            r ← t_i(r)                          ▷ go to the i-th sub-tree
```

Exercise 7.2

1. Improve the imperative look up with binary search among keys.

[2] `safeHd` is provided as `listToMaybe` in some library.

7.4 Delete

After delete a key, the number of keys may be too few to be a valid B-tree node. Except the root, the number of keys should not be less than $d - 1$, where d is the minimum degree. There are two methods symmetric to insert: we can either delete then fix, or merge before delete.

7.4.1 Delete and fix

We first extend the *delete* algorithm for binary search tree to multiple branches, then fix the B-tree balance rules. The main delete program is defined with these two steps:

$$delete\ x\ (d, t) = fix(d, del\ x\ t) \tag{7.27}$$

Where function *del* is the one we extend to support multiple branches. If t is a leaf, we merely delete x from the keys; otherwise, we partition the tree with x into 3 parts: (l, t', r). Where all the keys in l and sub-tree t' are less than x, and the rest in r are great than or equal ($\geq$) to x. When r isn't empty, we pick the first key k_i from it. If the key equals to x, ($k_i = x$), we next replace it with the maximum key k' of sub-tree t' ($k' = max(t')$), and recursively delete k' from t' as shown in figure 7.9. Otherwise (either r is empty, or $k_i \neq x$), we recursively delete x from sub-tree t'.

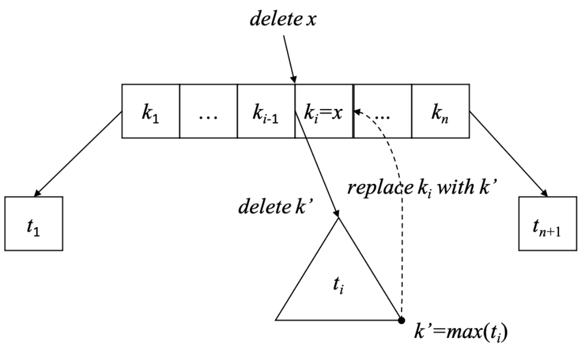


Figure 7.9: Replace k_i with $k' = max(t')$, then recursively delete k' from t'.

$$\begin{array}{rcl} del\ x\ (ks, \varnothing) & = & (delete_l\ x\ ks, \varnothing) \\ del\ x\ t & = & \begin{cases} Just\ x = safeHd\ ks' : & balance\ d\ l\ (del\ k'\ t')\ (k' : (tail\ ks'), ts') \\ otherwise : & balance\ d\ l\ (del\ x\ t')\ (ks', ts') \end{cases} \end{array} \tag{7.28}$$

Where $(l, t', (ks', ts')) = partition\ x\ t$, are the 3 parts partitioned by x. On top of it, we extract the maximum key k' from t'. The *max* function is defined as:

$$\begin{array}{rcl} max\ (ks, \varnothing) & = & last\ ks \\ max\ (ks, ts) & = & max\ (last\ ts) \end{array} \tag{7.29}$$

Function *last* returns the last element from a list (Equation 1.4 in chapter 1). $delete_l$ is the list delete algorithm (Equation 1.14 in chapter 1). *tail* drops the first element from a list and returns the rest (Equation 1.1). We need modify the *balance* function, which we defined for *insert* before, with the additional logic to merge the node if it contains too less keys.

$$balance\ d\ (ks_l, ts_l)\ t\ (ks_r, ts_r) = \begin{cases} full\ d\ t : fix_f \\ low\ d\ t : fix_l \\ otherwise : (ks_l \mathbin{+\!\!+} ks_r, ts_l \mathbin{+\!\!+} [t] \mathbin{+\!\!+} ts_r) \end{cases} \tag{7.30}$$

If t is overly low ($< d - 1$ keys), we call fix_l to merge it with the left part (ks_l, ts_l) or right part (ks_r, ts_r) depends on which side of keys is not empty. Use the left part for example: we extract the last element from ks_l and ts_l respectively, say k_m and t_m. Then call *unsplit* (defined in Equation 7.8) to merge them with t as $unsplit\ t_m\ k_m\ t$. It forms a new sub-tree with more keys. Finally we call balance again to build the result B-tree.

$$fix_l = \begin{cases} ks_l \neq \varnothing : & balance\ d\ (init\ ks_l, init\ ts_l)\ (unsplit\ t_m\ k_m\ t)\ (ks_r, ts_r) \\ ks_r \neq \varnothing : & balance\ d\ (ks_l, ts_l)\ (unsplit\ t\ k_1\ t_1)\ (tail\ ks_r, tail\ ts_r) \\ otherwise : & t \end{cases} \tag{7.31}$$

The last case (otherwise) means $ks_l = ks_r = \varnothing$, both sides are empty. The tree is a singleton leaf hence need not fixing. k_1 and t_1 are the first element in ks_r and ts_r respectively. Finally, we need modify the fix function defined for *insert*, add new logic for *delete*:

$$\begin{array}{lcl} fix\ (d, (\varnothing, [t])) & = & (d, t) \\ fix\ (d, t) & = & \begin{cases} full\ d\ t : & (d, ([k], [l, r])), \text{where } (l, k, r) = split\ d\ t \\ otherwise : & (d, t) \end{cases} \end{array} \tag{7.32}$$

What we add is the first case. After delete, if the root contains nothing but a sub-tree, we can shrink the height, pull the single sub-tree as the new root. The following example program implements the *delete* algorithm.

```
delete x (d, t) = fixRoot (d, del x t) where
    del x (BTree ks []) = BTree (List.delete x ks) []
    del x t = if (Just x) == safeHd ks' then
                let k' = max t' in
                   balance d l (del k' t') (k':(tail ks'), ts')
              else balance d l (del x t') r
      where
        (l, t', r@(ks', ts')) = partition x t

fixRoot (d, BTree [] [t]) = (d, t)
fixRoot (d, t) | full d t  = let (t1, k, t2) = split d t in
                                (d, BTree [k] [t1, t2])
               | otherwise = (d, t)

balance d (ks1, ts1) t (ks2, ts2)
    | full d t  = fixFull
    | low  d t  = fixLow
    | otherwise = BTree (ks1 ++ ks2) (ts1 ++ [t] ++ ts2)
  where
    fixFull = let (t1, k, t2) = split d t in
                BTree (ks1 ++ [k] ++ ks2) (ts1 ++ [t1, t2] ++ ts2)
    fixLow | not $ null ks1 = balance d (init ks1, init ts1)
                                        (unsplit (last ts1) (last ks1) t)
                                        (ks2, ts2)
           | not $ null ks2 = balance d (ks1, ts1)
                                        (unsplit t (head ks2) (head ts2))
                                        (tail ks2, tail ts2)
           | otherwise = t
```

We leave the *delete* function for the 'paired list' implementation as an exercise. Figure 7.10, 7.11, and 7.12 give examples of delete.

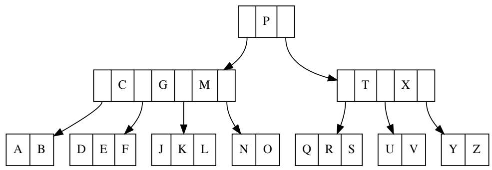


Figure 7.10: Before delete

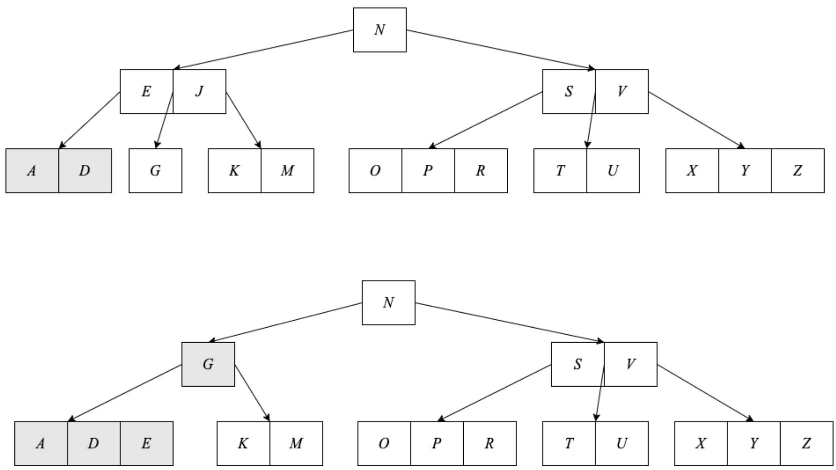


Figure 7.11: Delete 'C', then delete 'J'

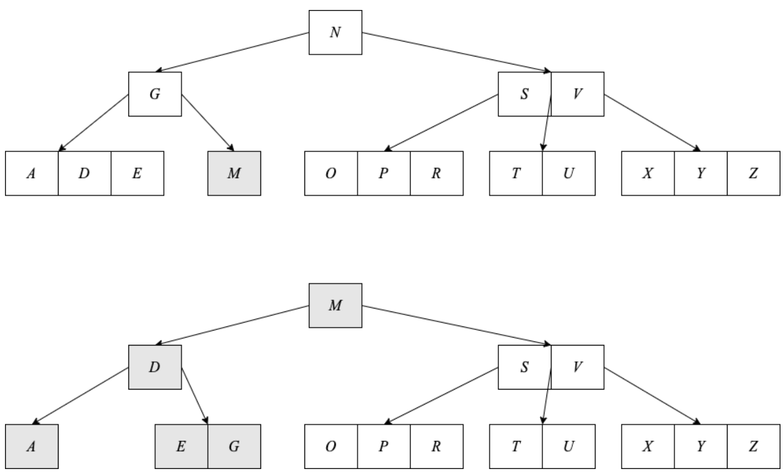


Figure 7.12: Delete 'K', then delete 'N'

7.4.2 Merge before delete

The other way is to merge the nodes before delete if there are too few keys. Consider delete key x from the tree t, let us start from the easy case.

Case 1. If x exists in node t, and t is a leaf, we can directly remove x from t. If t is the singleton node in the tree (root), we needn't worry about too few keys.

Case 2. If x exists in node t, but t is not a leaf. There are three sub-cases:

Case 2a. As shown in figure 7.9, let the predecessor of $k_i = x$ be k', where $k' = max(t_i)$. If t_i has sufficient keys ($\geq d$), we replace k_i with k', then recursively delete k' from t_i.

Case 2b. If t_i does not have enough keys, but the sub-tree t_{i+1} does ($\geq d$). Symmetrically, we replace k_i with its successor k'', where $k'' = min(t_{i+1})$, then recursively delete k'' from t_{i+1}, as shown in figure 7.13.

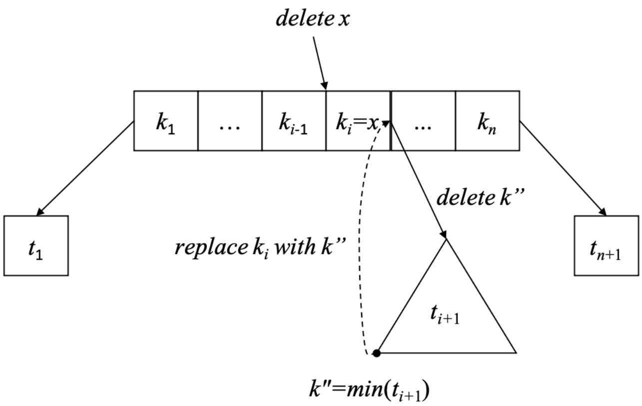


Figure 7.13: Replace k_i with $k'' = min(t_{i+1})$, then delete k'' from t_{i+1}.

Case 2c. If neither t_i nor t_{i+1} contains sufficient keys ($|t_i| = |t_{i+1}| = d - 1$), we merge t_i, x, t_{i+1} to a new node. This new node has $2d - 1$ keys, we can safely perform delete on it as shown in figure 7.14.

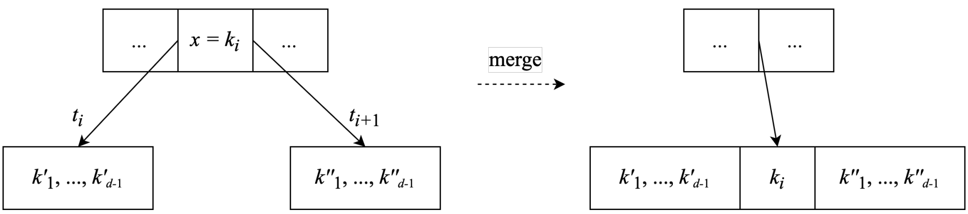


Figure 7.14: Merge before delete

Merge pushes a key k_i to the sub-tree. After that, if node t becomes empty, it means k_i is the only key in t, and t_i, t_{i+1} are the only two sub-trees. We need shrink the tree height as shown in figure 7.15.

Case 3. If node t does not contain x, we need recursively delete x from a sub-tree t_i. There are two sub-cases if there are too few keys in t_i:

Case 3a. Among the two siblings t_{i-1}, t_{i+1}, if either one has enough keys ($\geq d$), we move a key from t to t_i, then move a key from the sibling up to t, and move the corresponding sub-tree from the sibling to t_i. As shown in figure 7.16, t_i received one more key. We next recursively delete x from t_i.

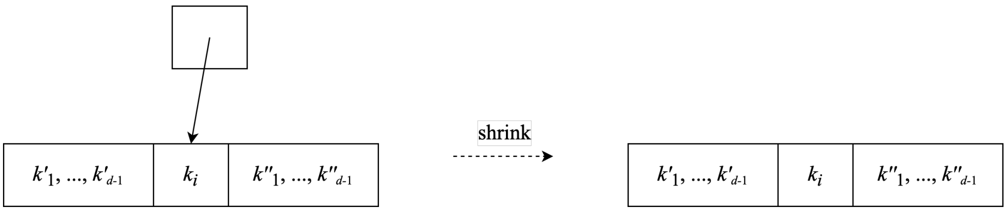


Figure 7.15: Shrink

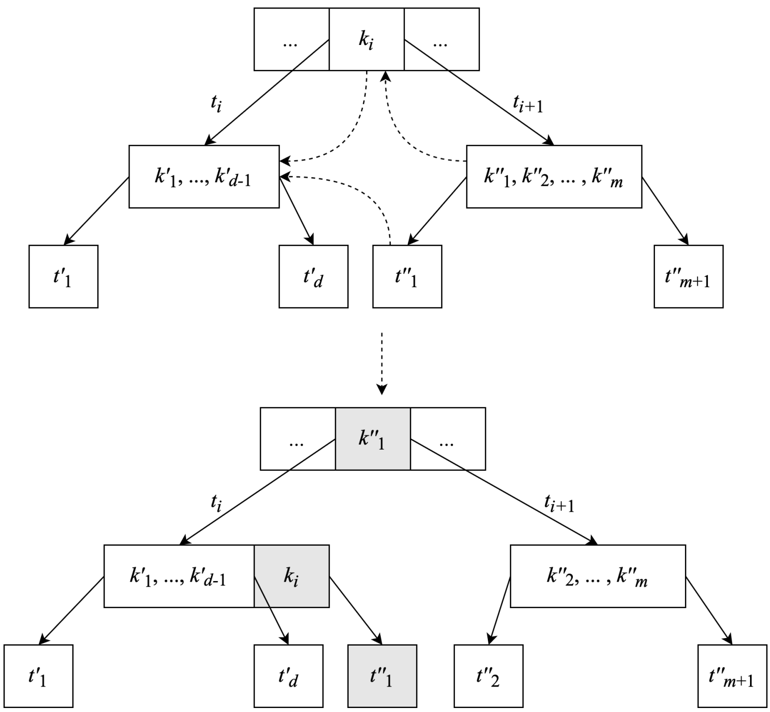


Figure 7.16: Borrow from the right sibling.

Case 3b. If neither sibling has sufficient keys ($|t_{i-1}| = |t_{i+1}| = d - 1$), we merge t_i, a key from t, and either sibling into a new node, as shown in figure 7.17. Then recursively delete x from it.

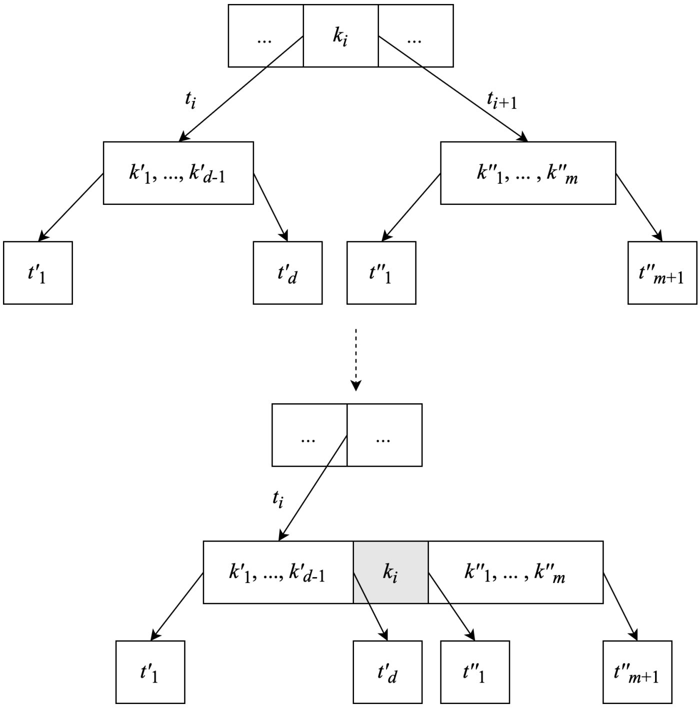


Figure 7.17: Merge t_i, k, t_{i+1}

Below DELETE algorithm implements the 'merge then delete' method:

```
function Delete(t, k)
    if t is empty then
        return t
    i ← 1, n ← |K(t)|
    while i ≤ n and k > k_i(t) do
        i ← i + 1
    if k = k_i(t) then
        if t is leaf then                                   ▷ case 1
            Remove(K(t), k)
        else                                                ▷ case 2
            if |K(t_i(t))| ≥ d then                         ▷ case 2a
                k_i(t) ← Max(t_i(t))
                Delete(t_i(t), k_i(t))
            else if |K(t_{i+1}(t))| ≥ d then                ▷ case 2b
                k_i(t) ← Min(t_{i+1}(t))
                Delete(t_{i+1}(t), k_i(t))
            else                                            ▷ case 2c
                Merge-At(t, i)
                Delete(t_i(t), k)
                if K(T) is empty then
                    t ← t_i(t)                              ▷ Shrinks height
        return t
```

```
    if t is not leaf then
        if k > k_n(t) then
            i ← i + 1
        if |K(t_i(t))| < d then                              ▷ case 3
            if i > 1 and |K(t_{i-1}(t))| ≥ d then            ▷ case 3a: left
                INSERT(K(t_i(t)), k_{i-1}(t))
                k_{i-1}(t) ← POP-LAST(K(t_{i-1}(t)))
                if t_i(t) is not leaf then
                    INSERT(T(t_i(t)), POP-BACK(T(t_{i-1}(t))))
            else if i ≤ n and |K(t_{i+1}(t))| ≥ d then       ▷ case 3a: right
                APPEND(K(t_i(t)), k_i(t))
                k_i(t) ← POP-FIRST(K(t_{i+1}(t)))
                if t_i(t) is not leaf then
                    APPEND(T(t_i(t)), POP-FIRST(T(t_{i+1}(t))))
            else                                             ▷ case 3b
                if i = n + 1 then
                    i ← i - 1
                MERGE-AT(t, i)
        DELETE(t_i(t), k)
        if K(t) is empty then                                ▷ Shrinks height
            t ← t_1(t)
    return t
```

Where MERGE-AT(t, i) merges sub-tree $t_i(t)$, key $k_i(t)$, and $t_{i+1}(t)$ into one sub-tree.

```
procedure MERGE-AT(t, i)
    x ← t_i(t)
    y ← t_{i+1}(t)
    K(x) ← K(x) ++ [k_i(t)] ++ K(y)
    T(x) ← T(x) ++ T(y)
    REMOVE-AT(K(t), i)
    REMOVE-AT(T(t), i + 1)
```

Exercise 7.3

1. When delete a key k from the branch node, we use the maximum key from the predecessor sub-tree $k' = max(t')$ to replace k, then recursively delete k' from t'. There is a symmetric method, to replace k with the minimum key from the successor sub-tree. Implement this solution.

2. Define the *delete* function for the 'paired list' implementation.

7.5 Summary

We extend the binary search tree to multiple branches, then constrain the branches within a range to develop the B-tree. B-tree is used as a tool to control the magnetic disk access (chapter 18, [4]). Because all B-tree nodes store keys in a range, not too few or too many. B-tree is balanced. Most of the tree operations are proportion to the height. The performance is bound to $O(\lg n)$ time, where n is the number of keys in B-tree.

7.6 Appendix: Example programs

Definition of B-tree:

```
data BTree<K, Int deg> {
    [K] keys
    [BTree<K>] subStrees;
}
```

Split node

```
void split(BTree<K, deg> z, Int i) {
    var d = deg
    var x = z.subTrees[i]
    var y = BTree<K, deg>()
    y.keys = x.keys[d ...]
    x.keys = x.keys[ ... d - 1]
    if not isLeaf(x) {
      y.subTrees = x.subTrees[d ... ]
      x.subTrees = x.subTrees[... d]
    }
    z.keys.insert(i, x.keys[d - 1])
    z.subTrees.insert(i + 1, y)
}

Bool isLeaf(BTree<K, deg> t) = t.subTrees == []
```

Insert a key to B-tree:

```
BTree<K, deg> insert(BTree<K, deg> tr, K key) {
    var root = tr
    if isFull(root) {
        var s = BTree<K, deg>()
        s.subTrees.insert(0, root)
        split(s, 0)
        root = s
    }
    return insertNonfull(root, key)
}
```

Insert a key to a non-full node.

```
BTree<K, deg> insertNonfull(BTree<K, deg> tr, K key) {
    if isLeaf(tr) {
        orderedInsert(tr.keys, key)
    } else {
        Int i = length(tr.keys)
        while i > 0 and key < tr.keys[i - 1] {
            i = i - 1
        }
        if isFull(tr.subTrees[i]) {
            split(tr, i)
            if key > tr.keys[i] then i = i + 1
        }
        insertNonfull(tr.subTree[i], key)
    }
    return tr
}
```

Where `orderedInsert` inserts an element to an ordered list.

```
void orderedInsert([K] lst, K x) {
    Int i = length(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
    }
}

Bool isFull(BTree<K, deg> x) = length(x.keys) ≥ 2 ∗ deg - 1
Bool isLow(BTree<K, deg> x) = length(x.keys) ≤ deg - 1
```

Iterative look up:

```
Optional<(BTree<K, deg>, Int)> lookup(BTree<K, deg> tr, K key) {
    loop {
        Int i = 0, n = length(tr.keys)
        while i < n and key > tr.keys[i] {
            i = i + 1
        }
        if i < n and key == tr.keys[i] then return Optional.of((tr, i))
        if isLeaf(tr) {
            return Optional.Nothing
        } else {
            tr = tr.subTrees[i]
        }
    }
}
```

Imperative merge before delete:

```
BTree<K, deg> delete(BTree<K, deg> t, K x) {
    if empty(t.keys) then return t
    Int i = 0, n = length(t.keys)
    while i < n and x > t.keys[i] { i = i + 1 }
    if x == t.keys[i] {
        if isLeaf(t) {         // case 1
            removeAt(t.keys, i)
        } else {
            var tl = t.subtrees[i]
            var tr = t.subtrees[i + 1]
            if not low(tl) {         // case 2a
                t.keys[i] = max(tl)
                delete(tl, t.keys[i])
            } else if not low(tr) {  // case 2b
                t.keys[i] = min(tr)
                delete(tr, t.keys[i])
            } else {                 // case 2c
                mergeSubtrees(t, i)
                delete(d, tl, x)
                if empty(t.keys) then t = tl  // shrink height
            }
        return t
    }
    if not isLeaf(t) {
        if x > t.keys[n - 1] then i = i + 1
        if low(t.subtrees[i]) {
            var tl = if i == 0 then null else t.subtrees[i - 1]
            var tr = if i == n then null else t.subtrees[i + 1]
            if tl ≠ null and (not low(tl)) {   // case 3a, left
                insert(t.subtrees[i].keys, 0, t.keys[i - 1])
                t.keys[i - 1] = popLast(tl.keys)
                if not isLeaf(tl) {
                    insert(t.subtrees[i].subtrees, 0, popLast(tl.subtrees))
                }
            } else if tr ≠ null and (not low(tr)) {  // case 3a, right
                append(t.subtrees[i].keys, t.keys[i])
                t.keys[i] = popFirst(tr.keys)
                if not isLeaf(tr) {
                    append(t.subtrees[i].subtrees, popFirst(tr.subtrees))
```

```
                }
            } else {       // case 3b
                mergeSubtrees(t, if i < n then i else (i - 1))
                if i == n then i = i - 1
            }
        delete(t.subtrees[i], x)
        if empty(t.keys) then t = t.subtrees[0]    // shrink height
        }
    }
    return t
}
```

merge sub-trees, find the min/max key from a B-tree.

```
void mergeSubtrees(BTree<K, deg>, Int i) {
    t.subtrees[i].keys += [t.keys[i]] + t.subtrees[i + 1].keys
    t.subtrees[i].subtrees += t.subtrees[i + 1].subtrees
    removeAt(t.keys, i)
    removeAt(t.subtrees, i + 1)
}

K max(BTree<K, deg> t) {
    while not empty(t.subtrees) {
        t = last(t.subtrees)
    }
    return last(t.keys)
}

K min(BTree<K, deg> t) {
    while not empty(t.subtrees) {
        t = t.subtrees[0]
    }
    return t.keys[0]
}
```

Chapter 8

Binary Heaps

8.1 Definition

Heaps are widely used for sorting, priority scheduling and graph algorithms, and etc.[40]. The most popular implementation models the heap as a complete binary tree in array[4]. The most efficient heap sort algorithm developed by R.W. Floyd is also based on this method[41] [42]. For the generic heap definition, we can implement with varies data structures but not limit to array. In this chapter, we focus on the heaps implemented with binary trees, including leftist heap, skew heap, and splay heap[3]. A heap is either empty, or stores comparable elements that satisfies a property and three operations:

1. **The heap property**: the top element is always the minimum;
2. **Pop**: removes the top element from the heap and maintain the heap property: the new top is still the minimum in the rest;
3. **Insert**: add a new element to the heap and maintain the heap property;
4. **Other**: operations like merge also maintain the heap property.

Because elements are comparable, we can also define the heap always keeps the maximum on top. We call the heap with the minimum on top as *min-heap*, the maximum on top as *max-heap*. When implement heap with a tree, we can put the minimum (or the maximum) in the root. After pop, we remove the root, and rebuild the tree from the sub-trees. We call the heap implemented with binary tree as *binary heap*. This chapter gives three types of binary heap.

8.2 Binary heap by array

The first implementation is to represent the a complete binary tree with an array. The complete binary tree is 'almost' full. The full binary tree of depth k contains $2^k - 1$ nodes. We can number every node top-down, from left to right as 1, 2, ..., $2^k - 1$. The node number i in the complete binary tree is located at the same position in the full binary tree. The leaves only appear in the bottom layer, or the second last layer. Figure 8.1 shows a complete binary tree and the array. As the complete binary tree, the i-th cell in array corresponds to a node, its parent node maps to the $\lfloor i/2 \rfloor$-th cell; the left sub-tree maps to the $2i$-th cell, and the right sub-tree maps to the $2i+1$-th cell. If any sub-tree maps to an index out of the array bound, then the sub-tree does not exist (i.e. leaf node). We can define the map as below (index starts from 1):

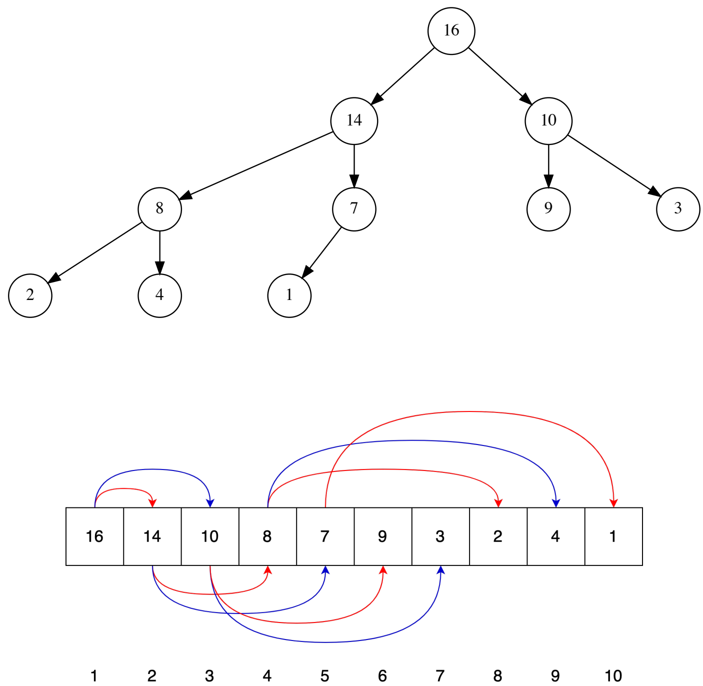


Figure 8.1: Map between a complete binary tree and an array.

$$\begin{cases} parent(i) & = \lfloor \frac{i}{2} \rfloor \\ left(i) & = 2i \\ right(i) & = 2i + 1 \end{cases} \tag{8.1}$$

8.2.1 Heapify

Heapify is the process maintain heap property, keep the minimum element on the top. For binary heap, we can obtain a stronger property as the binary tree is recursive: every sub-tree stores its minimum element in the root. In other words, every sub-tree is also a binary heap. Consider the min-heap represented with array, for any cell index i, we examine if all the elements in sub-trees are greater then or equal to it ($\geq$). Exchange when not satisfies. Repeat this for all sub-trees rooted at i.

function HEAPIFY(A, i)
  $n \leftarrow |A|$
  loop
    $s \leftarrow i$ ▷ s is the smallest
    $l \leftarrow$ LEFT(i), $r \leftarrow$ RIGHT(i)
    if $l \leq n$ and $A[l] < A[i]$ **then**
      $s \leftarrow l$
    if $r \leq n$ and $A[r] < A[s]$ **then**
      $s \leftarrow r$
    if $s \neq i$ **then**
      EXCHANGE $A[i] \leftrightarrow A[s]$
      $i \leftarrow s$
    else
      return

For index i in array A, any sub-tree node should not be less than $A[i]$. Otherwise, we exchange $A[i]$ with the smallest one, and recursively check the sub-trees. As the process time is proportion to the height of the tree, HEAPIFY is bound to $O(\lg n)$, where n is the length of the array. Figure 8.2 gives the steps when apply HEAPIFY from 2 to array [1, 13, 7, 3, 10, 12, 14, 15, 9, 16]. The result is [1, 3, 7, 9, 10, 12, 14, 15, 13, 16].

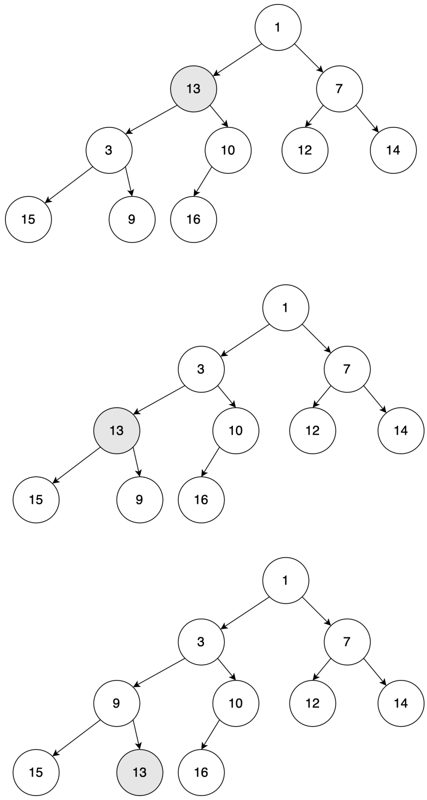


Figure 8.2: Heapify. Step 1: the minimum of 13, 3, 10 is 3, exchange 3 ↔ 13; Step 2: the minimum of 13, 15, 9 is 9, exchange 13 ↔ 9; Step 3: 13 is leaf, terminate.

8.2.2 Build

We can build heap from arbitrary array with HEAPIFY. List how many nodes in each level of a complete binary tree: $1, 2, 4, 8, ...$. They are all power of 2 except for the last level. Because the tree is not necessarily full, there are at most 2^{p-1} nodes, where p is the smallest integer satisfying $2^p - 1 \geq n$, and n is the length of the array. Skip all leaves because HEAPIFY takes no effect on them, we start applying HEAPIFY to the last branch node (which index $\leq \lfloor n/2 \rfloor$) bottom-up. The build function is defined as below:

```
function BUILD-HEAP(A)
    n ← |A|
    for i ← ⌊n/2⌋ down to 1 do
        HEAPIFY(A, i)
```

Although HEAPIFY is bound $O(\lg n)$ time, BUILD-HEAP is bound to $O(n)$, but not $O(n \lg n)$. We skip all leaves, check and move down a level at most for 1/4 nodes; check and move down two levels at most for 1/8 nodes; check and move down three levels at most for 1/16 nodes... the total comparison and move times is up to:

$$S = n(\frac{1}{4} + 2\frac{1}{8} + 3\frac{1}{16} + ...) \tag{8.2}$$

Multiply by 2 for both sides:

$$2S = n(\frac{1}{2} + 2\frac{1}{4} + 3\frac{1}{8} + ...) \tag{8.3}$$

Subtract (8.2) from (8.3):

$$\begin{array}{rcll} 2S - S & = & n[\frac{1}{2} + (2\frac{1}{4} - \frac{1}{4}) + (3\frac{1}{8} - 2\frac{1}{8}) + ...] & \text{shift by one and subtract} \\ S & = & n[\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...] & \text{geometric series} \\ & = & n & \end{array}$$

Figure 8.3 shows the steps to build a min-heap from array $[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$. The black node is where HEAPIFY is applied. The grey nodes are swapped to maintain the heap property.

8.2.3 Heap operations

Heap operations include access the top, pop, look up the top k elements, decrease an element in min-heap (or increase an element in max-heap), and insert a new element. For binary heap, the root stores the minimum element, corresponding to the first cell in array:

1: **function** TOP(A)
2:     **return** $A[1]$

Pop

After pop, the remaining elements in array shift ahead by one. However, after removed the root of the binary tree, the rest is not a binary tree any more. To avoid such situation, we swap the first and the last element in array, then reduce the array length by one. It equivalent to remove a leaf but not the root. We then apply HEAPIFY to recover the heap property:

1: **function** POP(A)
2:     $x \leftarrow A[1], n \leftarrow |A|$
3:     EXCHANGE $A[1] \leftrightarrow A[n]$
4:     REMOVE(A, n)
5:     **if** A is not empty **then**
6:         HEAPIFY(A, 1)
7:     **return** x

It takes constant time to remove the last element from array, hence pop is also bound to $O(\lg n)$ time as same as HEAPIFY.

Top-k

We can obtain top k elements by repeatedly applying pop.

1: **function** TOP-K(A, k)
2:     $R \leftarrow [\,]$
3:     BUILD-HEAP(A)

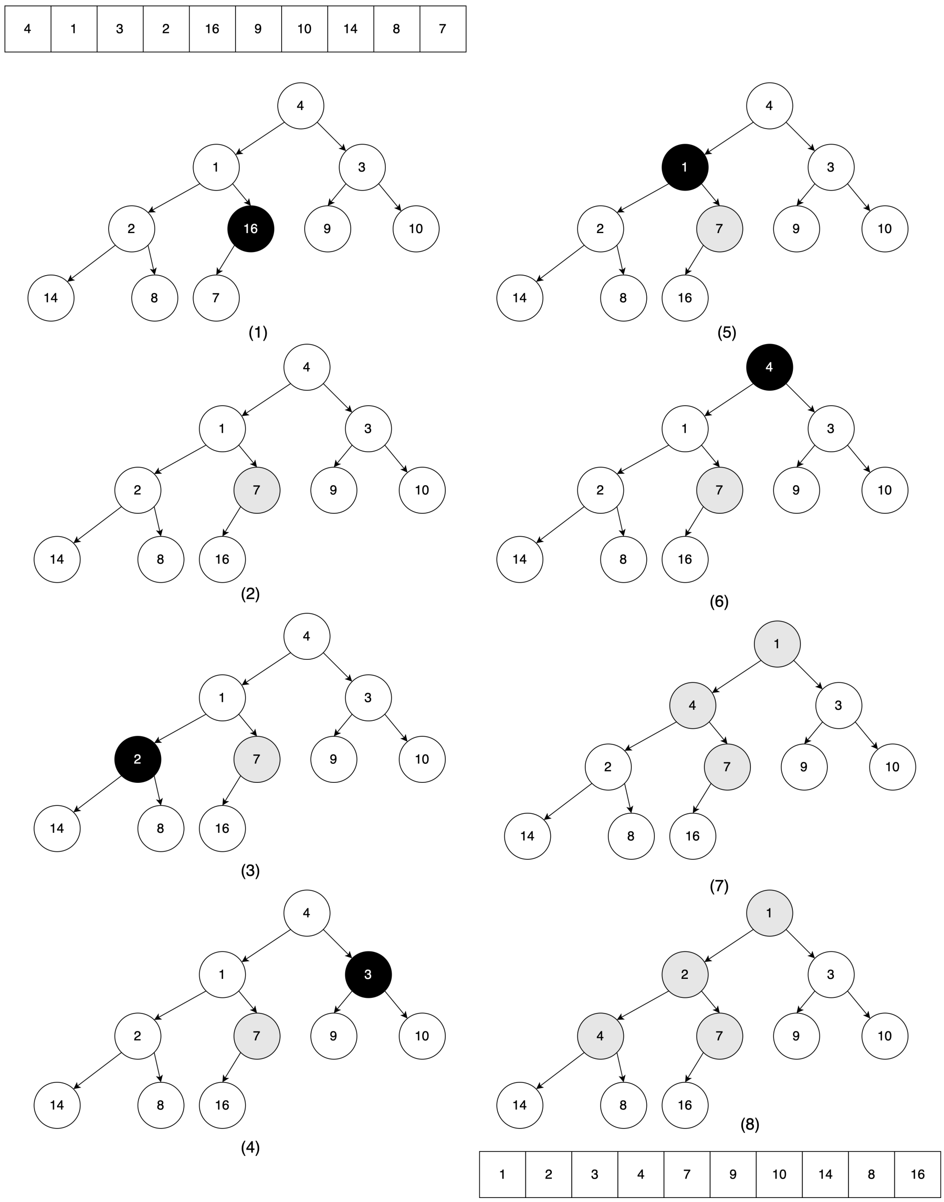


Figure 8.3: Build heap. (1) $16 > 7$; (2) exchange $16 \leftrightarrow 7$; (3) $2 < 14$ and $2 < 8$; (4) $3 < 9$ and $3 < 10$; (5) $1 < 2$ and $1 < 7$; (6) $1 < 4$ and $1 < 3$; (7) exchange $4 \leftrightarrow 1$; (8) exchange $4 \leftrightarrow 2$, end.

    loop MIN(k, $|A|$) times ▷ cut off when k out of array bound
        APPEND(R, POP(A))
    return R

Increase priority

We can implement a priority queue with heap, to schedule tasks with priorities. Every time, we peek the high priority task to execute. To make an urgent task run earlier, we can increase its priority. It corresponds to decrease an element in a min-heap, as shown in 8.4.

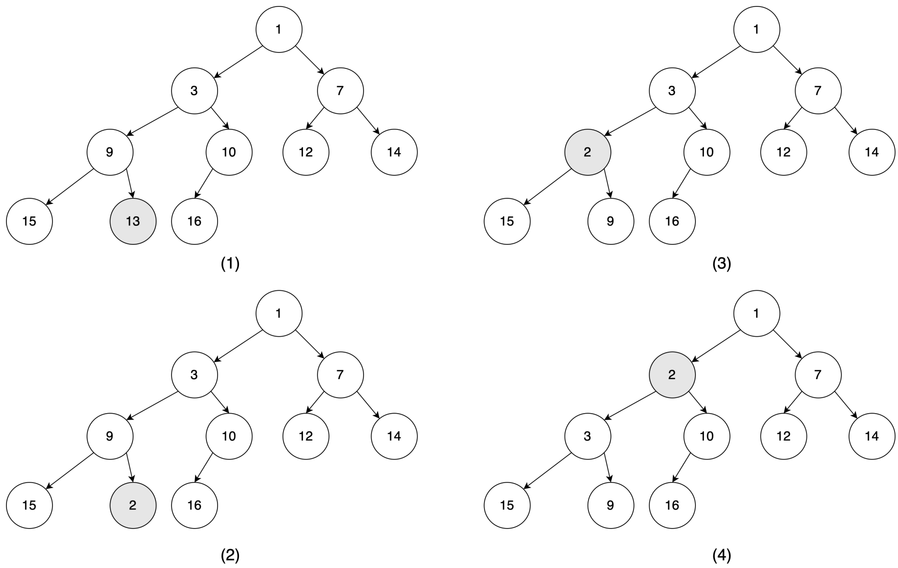


Figure 8.4: Decrease 13 to 2. Exchange 2 and 9, then exchange with 3.

The heap property may not be satisfied when decrease some element in a min-heap. Let the decreased element indexed at i in the array, below function resumes the heap property bottom-up. It is bound to $O(\lg n)$ time.

function HEAP-FIX(A, i)
    while $i > 1$ and $A[i] < A[$ PARENT(i) $]$ **do**
        EXCHANGE $A[i] \leftrightarrow A[$ PARENT(i) $]$
        $i \leftarrow$ PARENT(i)

Insertion

We can realize push with HEAP-FIX[4]. Use min-heap for example, we append the new element k to the tail of the array, then apply HEAP-FIX to recover the heap property:

function PUSH(A, k)
    APPEND(A, k)
    HEAP-FIX($A, |A|$)

8.2.4 Heap sort

We can sort elements with heap. Build a min-heap from a collection of n elements, the repeatedly pop the top element to obtain the ascending result. It takes $O(n)$ time to build the heap. The pop is bound to $O(\lg n)$ time, and runs for n times. Therefore, the total time is bound to $O(n \lg n)$. The space is bound to $O(n)$ as we need another list to hold the result.

```
function Heap-Sort(A)
    R ← [ ]
    Build-Heap(A)
    while A ≠ [ ] do
        Append(R, Pop(A))
    return R
```

Robert. W. Floyd gave a fast implementation with max-heap. The top stores the maximum one. Every time, swap the head and the tail elements in the array. After that the maximum is stored to the expected position, and the previous tail becomes the new top. We next decrease the heap size by one, and apply Heapify to maintain the heap property. Repeat this till the heap size decrease to one. This algorithm needn't the additional space to store the result.

```
function Heap-Sort(A)
    Build-Max-Heap(A)
    n ← |A|
    while n > 1 do
        Exchange A[1] ↔ A[n]
        n ← n − 1
        Heapify(A[1...n], 1)
```

Exercise 8.1

1. Consider another idea about in-place heap sort: Build a min-heap from the array A, the first element a_1 is in the right position. Treat the rest $[a_2, a_3, ..., a_n]$ as the new heap, and apply Heapify from a_2. Repeat this till the last element. Is this method correct?

   ```
   function Heap-Sort(A)
       Build-Heap(A)
       for i = 1 to n − 1 do
           Heapify(A[i...n], 1)
   ```

2. Similarly, can we apply Heapify k times from left to right to get the top-k elements?

   ```
   function Top-K(A, k)
       Build-Heap(A)
       n ← |A|
       for i ← 1 to min(k, n) do
           Heapify(A[i...n], 1)
   ```

8.3 Leftist heap and skew heap

When implement the heap with a explicit binary tree, after pop the rot, there remain two sub-trees. Both are heaps as shown in figure 8.5. How can we merge them to a new heap? To maintain the heap property, the new root must be the minimum for the remaining. We can give the first edge cases easily:

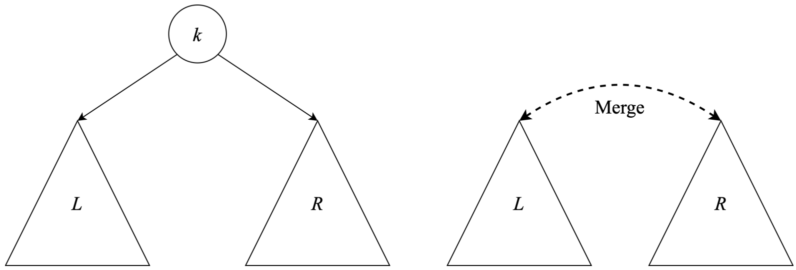


Figure 8.5: Merge left and right sub-trees after pop.

$$
\begin{array}{rcl}
merge(\varnothing, R) & = & R \\
merge(L, \varnothing) & = & L \\
merge(L, R) & = & ?
\end{array}
$$

Both left and right sub-trees are heaps. When they are not empty, each root stores the minimum respectively. We can compare the two roots, and peek the smaller as the new root. Let $L = (A, x, B)$, $R = (A', y, B')$, where A, A', B, B' are sub-trees. If $x < y$, then x is the new root. We keep A, and merge B and R recursively; alternatively, we can keep B, and merge A and R. The new heap can be $(merge(A, R), x, B)$ or $(A, x, merge(B, R))$. Both are right. To simplify, we always merge the right sub-tree. This method generates *leftist heap*.

8.3.1 Leftist heap

The leftist heap is implemented with leftist tree. C. A. Crane in 1972[43] developed leftist tree. He defined a rank for every node (also known as S-value) as the distance to the nearest NIL. The rank of NIL is 0. As shown in 8.6, The nearest leaf node to 4 is 8, the rank of 4 is 2; Both 4 and 8 are leaves, their ranks are 1. Although the left sub-tree of 5 is not empty, its right sub-tree is NIL, hence the rank is 1. We can define the merge method with rank as below. Let the ranks for left and right sub-trees be r_l, r_r respectively:

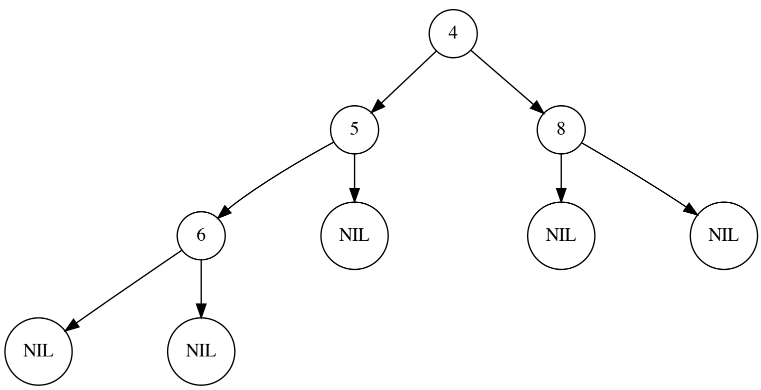


Figure 8.6: $rank(4) = 2$, $rank(6) = rank(8) = rank(5) = 1$.

1. Always merge the right sub-tree;

2. When $r_l < r_r$, exchange the left and right sub-trees.

We call above merge rules 'leftist property'. Basically, a leftist tree always has the shortest path to some NIL on the right. It tends to be unbalanced, while maintain a critical constraint:

Theorem 8.3.1. *For a leftist tree T of n nodes, the path from root to the rightmost NIL has at most $\lfloor \log(n+1) \rfloor$ nodes.*

We skip the proof[44] [51]. With this theorem, algorithms process along this path are ensured bound to $O(\lg n)$ time. We can define the leftist tree by reusing binary tree plus an additional rank. Let the none empty leftist tree be (r, L, k, R):

```
data LHeap a = E -- Empty
             | Node Int (LHeap a) a (LHeap a)
```

Function *rank* returns the rank value:

$$
\begin{array}{rcl}
rank\ \varnothing & = & 0 \\
rank\ (r, L, k, R) & = & r
\end{array}
\tag{8.4}
$$

Merge

To merge two leftist heaps, we define a *make* function. It compares the ranks of the sub-trees and swap them if necessary.

$$
make(A, k, B) = \begin{cases}
rank(A) < rank(B) : & (rank(A) + 1, B, k, A) \\
否则 : & (rank(B) + 1, A, k, B)
\end{cases}
\tag{8.5}
$$

It takes two sub-trees A and B. If rank of A is smaller, we let B be the left sub-tree, and A be the right. The rank of the new node is $rank(A) + 1$; otherwise if rank of B is smaller, we let A be the left sub-tree, and B be the right. The rank of the new node is $rank(B) + 1$. Given two leftist heaps H_1 and H_2, if they are not empty, let them be (r_1, L_1, K_1, R_1) and (r_2, L_2, k_2, R_2) respectively. Below function defines merge:

$$
\begin{array}{rcl}
merge\ \varnothing\ H_2 & = & H_2 \\
merge\ H_1\ \varnothing & = & H_1 \\
merge\ H_1\ H_2 & = & \begin{cases}
k_1 < k_2 : & make(L_1, k_1, merge\ R_1\ H_2) \\
否则 : & make(L_2, k_2, merge\ H_1\ R_2)
\end{cases}
\end{array}
\tag{8.6}
$$

We always apply *merge* to the right sub-tree recursively, hence the leftist property is maintained, and it is bound to $O(\lg n)$ time.The binary heap implemented by array performs well in most cases, and it suitable for the modern cache technology. However, it takes $O(n)$ time for merge. We need concatenate two arrays, and rebuild the heap[50].

```
1: function Merge-Heap(A, B)
2:     C ← Concat(A, B)
3:     Build-Heap(C)
```

We can define most heap operations with *merge*.

Top and pop

We can access the top element in $O(1)$ time, assume the heap is not empty:

$$
top\ (r, L, k, R) = k
\tag{8.7}
$$

After pop the root, we merge the left and right sub-trees as a new heap. Same as *merge*, pop is also bound to $O(\lg n)$ time.

$$
pop\ (r, L, k, R) = merge\ L\ R
\tag{8.8}
$$

Insert

To insert a new element k, we build a singleton leaf of k, then merge it with the heap:

$$insert\ k\ H = merge\ (1, \varnothing, k, \varnothing)\ H \tag{8.9}$$

Or write it in Curried form as $build = fold_r\ insert\ \varnothing$.

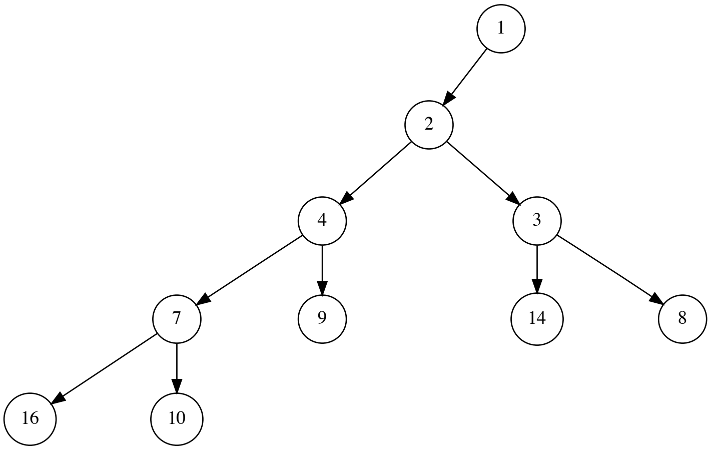


Figure 8.7: Build the leftist heap from $[9, 4, 16, 7, 10, 2, 14, 3, 8, 1]$.

Heap sort

Given a list, we build a leftist heap from it, then repeatedly pop the minimum element from top to obtain the sorted result.

$$sort = heapSort \circ build \tag{8.10}$$

Where

$$\begin{array}{rcl} heapSort\ [\,] & = & [\,] \\ heapSort\ H & = & (top\ H) : (heapSort\ (pop\ H)) \end{array} \tag{8.11}$$

We call pop n times, each takes $O(\lg n)$ time. The total time is bound to $O(n \lg n)$.

8.3.2 Skew heap

Leftist heap may lead to unbalanced tree in some cases as shown in figure 8.8. Skew heap is a self-adjusting heap. It simplifies the leftist heap and improves balance[46] [47]. When build the leftist heap, we swap the left and right sub-trees when the rank on left is smaller than the right. However, this method can't handle the case when either sub-tree has a NIL node. The rank is always 1 no matter how big the sub-tree is. Skew heap simplified the merge, it always swap the left and right sub-trees.

Skew heap is implemented with skew tree. Skew tree is a binary tree. The root stores the minimum element, every sub-tree is also a skew tree. Skew tree needn't the rank. We can directly re-use the binary tree definition. Let the none empty tree be (L, k, R).

```
data SHeap a = E -- Emtpy
             | Node (SHeap a) a (SHeap a)
```

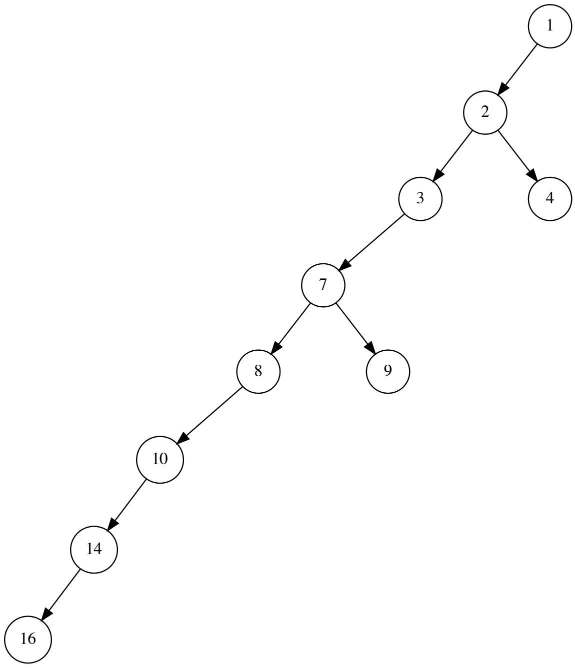


Figure 8.8: Leftist heap built from $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$.

Merge

When merge two none empty skew trees, we choose the smaller root as the new root. Then merge the greater tree with a sub-tree, and swap the left and right sub-trees. Let the two trees be $H_1 = (L_1, k_1, R_1)$ and $H_2 = (L_2, k_2, R_2)$. If $k_1 < k_2$, then choose k_1 as the new root. We can either merge H_2 with L_1, or merge H_2 with R_1. We choose R_1, and swap the left and right sub-trees. The result is $(merge(R_1, H_2), k_1, L_1)$.

$$
\begin{array}{rcl}
merge\ \varnothing\ H_2 & = & H_2 \\
merge\ H_1\ \varnothing & = & H_1 \\
merge\ H_1\ H_2 & = & \begin{cases} k_1 < k_2 : & (merge(R_1, H_2), k_1, L_1) \\ \text{otherwise} : & (merge(H_1, R_2), k_2, L_2) \end{cases}
\end{array}
\tag{8.12}
$$

Similar with leftist tree, the other operations, including insert, top, and pop are implemented with *merge*. Skew heap outputs a balanced tree even for ordered list as shown in figure 8.9.

8.4 Splay heap

The leftist heap and skew heap are implemented with binary tree. If change to binary search tree, then the minimum element will not be in root. We need $O(\lg n)$ time to locate the minimum. The performance will downgrade if the tree is not balanced. Although we can use the red-black tree to secure balancing, the splay tree provides a light weight implementation. It dynamically make the tree balanced. Splay tree takes cache-like approach. It rotates the node currently being accessed to the root, reduces the access

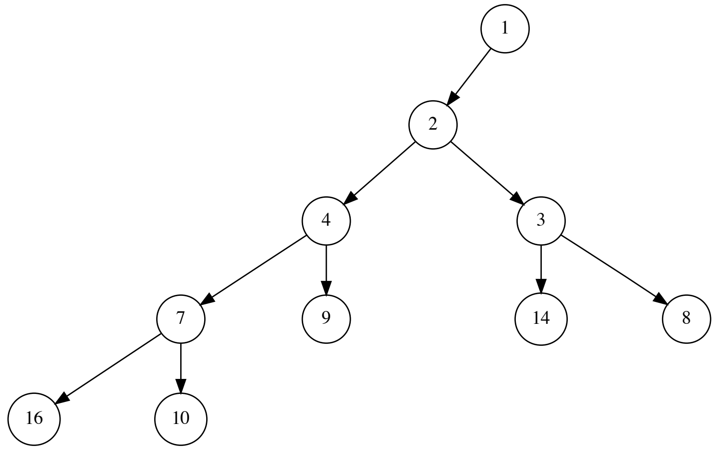


Figure 8.9: Skew tree built from $[1, 2, ..., 10]$.

time for next visit. We define such operation as 'splay'. The tree tends to be more balanced after several splay operations. Most splay tree operations perform in amortized $O(\lg n)$ time. Daniel Dominic Sleator and Robert Endre Tarjan developed splay tree in 1985[48] [49].

8.4.1 Splay

We introduce two methods to implement splay. The first is pattern matching, it need match multiple cases; the second has the uniformed form, but the implementation is complex. Let the node to be accessed be x, the parent node be p. If it has grand parent node, then denote it as g. There are 3 cases, each has two symmetric sub-cases. We explain one of them as shown in 8.10:

1. *Zig-zig*: Both x and p are on the left; or on the right. We rotate twice to make x as root.

2. *Zig-zag*: x is on the left, while p is on the right; or x is on the right, while P is on the left. After rotation, x becomes the root, p and g are siblings.

3. *Zig*: p is the root, we rotate to make x as root.

There are total 6 cases. Let the none empty tree be $T = (L, k, R)$, define splay as

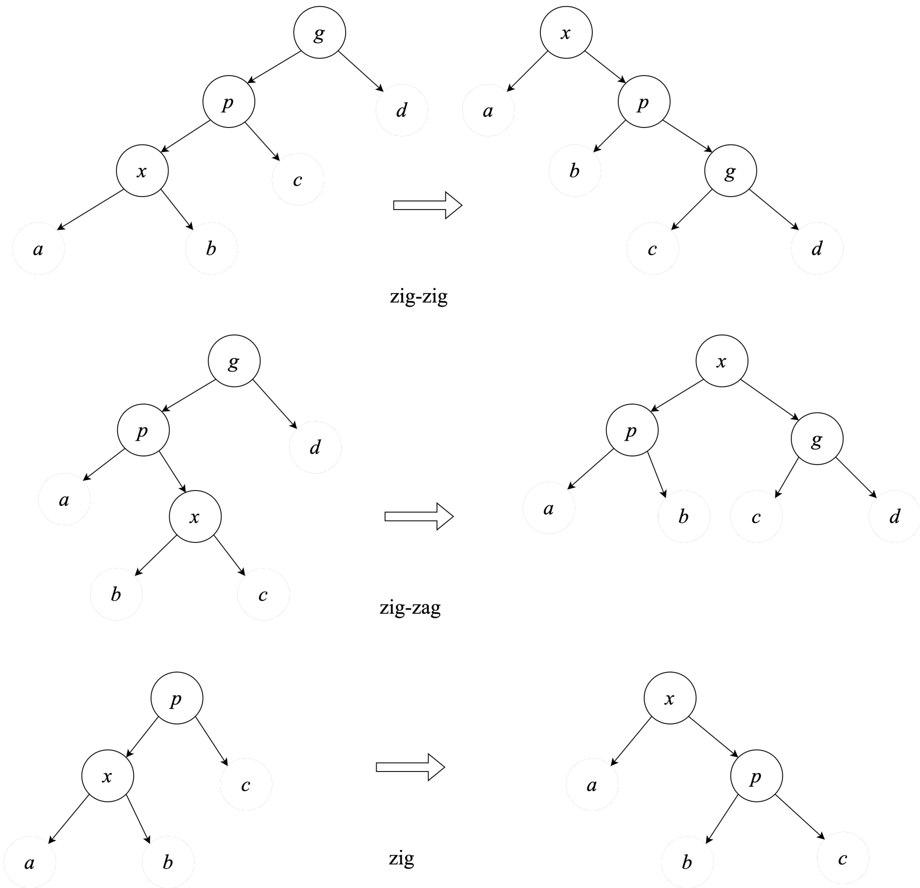


Figure 8.10: zig-zig: x and p are both on left or right, x becomes new root. zig-zag: x and p are on different sides, x becomes new root, p and g are siblings. zig: p is root, rotate to make x as root.

below when access element y:

$$
\begin{array}{rcll}
splay\ (((a, x, b), p, c), g, d)\ y & = & \begin{cases} x = y : & (a, x, (b, p, (c, g, d))) \\ \text{otherwise} : & T \end{cases} & \text{zig-zig} \\
splay\ (a, g, (b, p, (c, x, d)))\ y & = & \begin{cases} x = y : & (((a, g, b), p, c), x, d) \\ \text{otherwise} : & T \end{cases} & \text{zig-zig symmetric} \\
splay\ (a, p, (b, x, c), g, d)\ y & = & \begin{cases} x = y : & ((a, p, b), x, (c, g, d)) \\ \text{otherwise} : & T \end{cases} & \text{zig-zag} \\
splay\ (a, g, ((b, x, c), p, d))\ y & = & \begin{cases} x = y : & ((a, g, b), x, (c, p, d)) \\ \text{otherwise} : & T \end{cases} & \text{zig-zag symmetric} \\
splay\ ((a, x, b), p, c)\ y & = & \begin{cases} x = y : & (a, x, (b, p, c)) \\ \text{otherwise} : & T \end{cases} & \text{zig} \\
splay\ (a, p, (b, x, c))\ y & = & \begin{cases} x = y : & ((a, p, b), x, c) \\ \text{otherwise} : & T \end{cases} & \text{zig symmetric} \\
splay\ T\ y & = & T & \text{others}
\end{array}
\tag{8.13}
$$

The first two are 'zig-zig' cases; then two 'zig-zag' cases; then two zig cases. The tree keeps changed for all other cases. Every time when insert a new element, we trigger splay to adjust the balance. IF the tree is empty, the result is a singleton leaf; otherwise, we compare the new element and the root, then recursively insert to left (less than) or right (greater than) sub-tree and apply splay.

$$
\begin{array}{rcl}
insert\ \varnothing\ y & = & (\varnothing, y, \varnothing) \\
insert\ (L, x, R)\ y & = & \begin{cases} y < x : & splay\ ((insert\ L\ y), x, R)\ y \\ \text{otherwise} : & splay\ (L, x, (insert\ R\ y))\ y \end{cases}
\end{array}
\tag{8.14}
$$

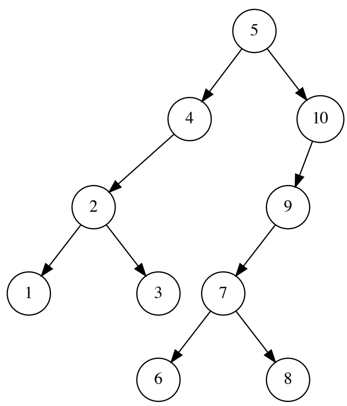


Figure 8.11: Splay tree built from $[1, 2, ..., 10]$.

Figure 8.11 gives the splay tree built from $[1, 2, ..., 10]$. It generates a well balanced tree. Okasaki found a simple rule for splaying[3]. Whenever we follow two left branches, or two right branches continuously, we rotate the two nodes. When access node of x, if move to left or right twice, then partition T as L and R, where L contains all the elements less than x, while R contains the remaining. Then we create a new tree with x as the root, and L, R as the left and right sub-trees. The partition process is recursively applied to sub-trees.

$$
\begin{array}{rcl}
partition\ \varnothing\ y & = & (\varnothing, \varnothing) \\
partition\ (L, x, R)\ y & = &
\end{array}
$$
$$
\begin{cases}
x < y & \begin{cases}
R = \varnothing & (T, \varnothing) \\
R = (L', x', R') & \begin{cases}
x' < y & (((L, x, L'), x', A), B) \\
 & \text{where: } (A, B) = partition\ R'\ y \\
\text{otherwise} & ((L, x, A), (B, x', R')) \\
 & \text{where: } (A, B) = partition\ L'\ y
\end{cases}
\end{cases} \\
\text{otherwise} & \begin{cases}
L = \varnothing & (\varnothing, T) \\
L = (L', x', R') & \begin{cases}
y < x' & (A, (L', x', (R', x, R))) \\
 & \text{where: } (A, B) = partition\ L'\ y \\
\text{otherwise} & ((L', x', A), (B, x, R)) \\
 & \text{where: } (A, B) = partition\ R'\ y
\end{cases}
\end{cases}
\end{cases}
\tag{8.15}
$$

Function *partition* takes a tree T, and a pivot y. For empty tree, the result is a pair of empty trees; otherwise let the tree be (L, x, R). We compare the pivot y and the root x. If $x < y$, there are two sub-cases: (1) R is empty. All elements in the binary search tree are less then y, hence the result is $(T, \varnothing)$; (2) Let $R = (L', x', R')$, if $x' < y$, we recursively partition R' with the pivot y. Put all the elements less than y in A, and the rest in B. The result is a pair of trees: $((L, x, L'), x', A)$ and B. If $x' > y$, then recursively partition L' with y to obtain (A, B). The result is also a pair of (L, x, A) and (B, x', R'). When $y < x$, the result is symmetric.

Alternatively, we can define insert with *partition*. When insert a new element k to splay heap T, we first partition the heap to two sub-trees of L and R. Where L contains all elements smaller than k, while R contains the rest. Then construct a new tree with k as the root, and L, R as the sub-trees.

$$
insert\ T\ k = (L, k, R), \text{其中：} (L, R) = partition\ T\ k \tag{8.16}
$$

8.4.2 Pop

Since splay tree is essentially a binary search tree, the minimum is at the left most. We need keep traversing the left sub-tree to access the heap 'top'. Let the none empty tree be $T = (L, k, R)$, we define the *top* function as below:

$$
\begin{array}{rcl}
top\ (\varnothing, k, R) & = & k \\
top\ (L, k, R) & = & top\ L
\end{array}
\tag{8.17}
$$

This is equivalent to *min* for the binary search tree. When pop, we need remove the minimum. We apply splay when move left twice.

$$
\begin{array}{rcl}
pop\ (\varnothing, k, R) & = & R \\
pop\ ((\varnothing, k', R'), k, R) & = & (R', k, R) \\
pop\ ((L', k', R'), k, R) & = & (pop\ L', k', (R', k, R))
\end{array}
\tag{8.18}
$$

The third row performs splaying without calling *partition*. It uses the binary search tree property. Top and pop are bound to $O(\lg n)$ time when the splay tree is balanced.

8.4.3 Merge

We can implement *merge* with *partition* to obtain the $O(\lg n)$ time bound. When merge two none-empty splay trees, we choose a root as the pivot to partition the other tree, then recursively merge the sub-trees:

$$
\begin{array}{rcl}
merge\ \varnothing\ T & = & T \\
merge\ (L, x, R)\ T & = & ((merge\ L\ L')\ x\ (merge\ R\ R'))
\end{array}
\tag{8.19}
$$

where

$$(L', R') = partition\ T\ x$$

If a heap is empty, then the result is the other heap; otherwise, let a heap be (L, x, R). We use x to partition T to (L', R'), where L contains all elements less than x in T, while R' contains the rest. Then we recursively merge L and L' to the left sub-tree, and merge R and R' to the right sub-tree.

8.5 Summary

We give the generic definition of binary heap in this chapter. There are several implementations. The array based representation is suitable for imperative implementation. It maps a complete binary tree to array, supports random access any element. We also directly use the binary tree to implement the heap in functional way. Most operations are bound to $O(\lg n)$ time, some are $O(1)$ amortized time. Okasaki gave detailed analysis[3]. When extend from binary tree to k-ary tree, we obtain binomial heap, Fibonacci heap, and pairing heap. We introduce these heaps in chapter 10.

Exercise 8.2

1. Realize leftist heap, skew heap, and splay heap in imperative approach.
2. Define fold for heap.

8.6 Appendix - example programs

For the complete binary tree represented by array, access parent, and sub-trees with bit-wise operation (index from 0):

```
Int parent(Int i) = ((i + 1) >> 1) - 1

Int left(Int i) = (i << 1) + 1

Int right(Int i) = (i + 1) << 1
```

Heapify, parameterized the comparison:

```
void heapify([K] a, Int i, Less<K> lt) {
    Int l, r, m
    Int n = length(a)
    loop {
        m = i
        l = left(i)
        r = right(i)
        if l < n and lt(a[l], a[i]) then m = l
        if r < n and lt(a[r], a[m]) then m = r
        if m ≠ i {
```

```
            swap(a, i, m);
            i = m
        } else {
            break
        }
    }
}
```

Build the binary heap from array:

```
void buildHeap([K] a, Less<K> lt) {
    Int n = length(a)
    for Int i = (n-1) / 2 downto 0 {
        heapify(a, i, lt)
    }
}
```

Pop:

```
K pop([K] a, Less<K> lt) {
    var n = length(a)
    t = a[n]
    swap(a, 0, n - 1)
    remove(a, n - 1)
    if a ≠ [] then heapify(a, 0, lt)
    return t
}
```

Obtain the top-k elements:

```
[K] topk([K] a, Int k, Less<K> lt) {
    buildHeap(a, lt)
    [K] r = []
    loop min(k, length(a)) {
        append(r, pop(a, lt))
    }
    return r
}
```

Decrease the key in min-heap:

```
void decreaseKey([K] a, Int i, K k, Less<K> lt) {
    if lt(k, a[i]) {
        a[i] = k
        heapFix(a, i, lt)
    }
}

void heapFix([K] a, Int i, Less<K> lt) {
    while i > 0 and lt(a[i], a[parent(i)]) {
        swap(a, i, parent(i))
        i = parent(i)
    }
}
```

Push new element:

```
void push([K] a, K k, less<K> lt) {
    append(a, k)
    heapFix(a, length(a) - 1, lt)
}
```

Heap sort:

```
void heapSort([K] a, less<K> lt) {
    buildHeap(a, not ∘ lt)
```

```
    n = length(a)
    while n > 1 {
        swap(a, 0, n - 1)
        n = n - 1
        heapify(a[0 .. (n - 1)], 0, not ∘ lt)
    }
}
```

Merge two leftist heaps:

```
merge E h = h
merge h E = h
merge h1@(Node _ x l r) h2@(Node _ y l' r') =
    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 two skew heaps:

```
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'
```

Splay operation:

```
-- 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
-- others
splay t _ = t
```

Insert new element to the splay heap:

```
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
```

Partition the splay tree:

```
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)
```

Merge two splay trees:

```
merge E t = t
merge (Node l x r) t = Node (merge l l') x (merge r r')
    where (l', r') = partition t x
```

Chapter 9

Selection sort

9.1 Introduction

Selection sort is a straightforward sorting algorithm. It repeatedly selects the minimum (or maximum) from a collection of elements. It performs below the divide and conqueror sort algorithms, like quick sort and merge sort. We'll give different ways to improve it, and finally evolve it to heap sort, achieving $O(n \lg n)$, the upper limit of comparison based sort algorithm time bound. When facing a bunch of grapes, there are two types of kids. One pick the biggest grape to eat every time, the other always eat the smallest one. The first type eats the grape in ascending order of size, the other eats in descending order. In either case, the kid essentially applies selection sort method. It can be defined as:

1. If the collection is empty, the sorted result is empty;
2. Otherwise, select the minimum element, and append it to the sorted result.

It sorts elements in ascending order. We can obtain descending order by selecting the maximum. The compare operation can be abstract.

$$
\begin{array}{rcl}
sort\ [\,] & = & [\,] \\
sort\ A & = & m : sort\ (A - [m]) \quad \text{where } m = min\ A
\end{array}
\tag{9.1}
$$

Where $A - [m]$ is the remaining elements in A except m. The corresponding imperative implementation is as below:

function $\textsc{Sort}(A)$
    $X \leftarrow [\,]$
    while $A \neq [\,]$ **do**
        $x \leftarrow \textsc{Min}(A)$
        $\textsc{Del}(A, x)$
        $\textsc{Append}(X, x)$
    return X

Figure 9.1 shows the process of selection sort. We can improve it to in-place sort. The idea is to reuse A. Place the minimum element in $A[1]$, the second smallest one in $A[2]$, ...When find the i-th smallest element, swap it with $A[i]$.

function $\textsc{Sort}(A)$
    for $i \leftarrow 1$ to $|A|$ **do**
        $m \leftarrow \textsc{Min-At}(A, i)$
        $\textsc{Exchange}\ A[i] \leftrightarrow A[m]$

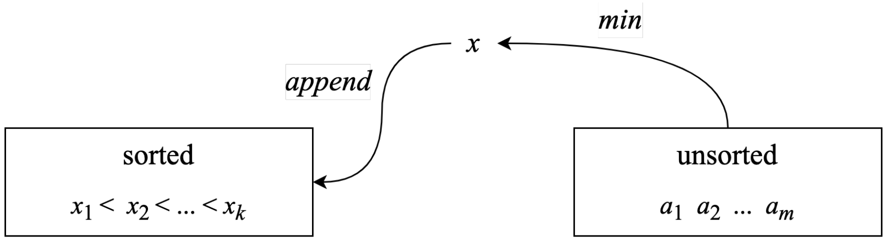


Figure 9.1: The left is sorted, repeatedly select the minimum of the rest and append.

Let $A = [a_1, a_2, ..., a_n]$, when select the i-th smallest element, $[a_1, a_2, ..., a_{i-1}]$ are sorted. We find the minimum of $[a_i, a_{i+1}, ..., a_n]$, and swap it with a_i. Repeat this to process all elements as shown in figure 9.2.

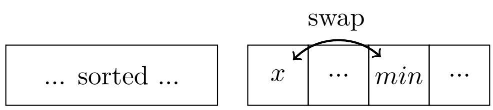


Figure 9.2: The left is sorted, repeatedly find the minimum and swap to the right position.

9.2 Find the minimum

We can use the 'compare and swap' method to find the minimum element. Label the elements with $1, 2, ..., n$. Compare the elements of number 1 and 2, pick the smaller and compare it with number 3, ... repeat till the last element of number n.

```
function MIN-AT(A, i)
    m ← i
    for i ← m + 1 to |A| do
        if A[i] < A[m] then
            m ← i
    return m
```

The MIN-AT find the minimum m from slice $A[i...]$. Let m start pointing to $A[i]$, then scan $A[i+1], A[i+2], ...$.

We can also find the minimum from list of elements L recursively. When L is a singleton, the only element is the minimum; otherwise pick an element x from L, then recursively find the minimum y from the remaining, the smaller one between x and y is the minimum of L.

$$\begin{array}{rcl} min\ [x] & = & (x, [\ ]) \\ min\ (x : xs) & = & \begin{cases} x < y : & (x, xs),\ \text{where } (y, ys) = min\ xs \\ \text{otherwise} : & (y,\ x : ys) \end{cases} \end{array} \tag{9.2}$$

We can further improve it tail recursively. Divide the elements with two groups A and B. A is initialized empty ($[\ ]$), B contains all elements. We pick two elements from B, compare and put the greater one to A, leave the smaller one as m. Then repeatedly pick element from B, compare with m till B becomes empty. Finally, m is the minimum element. At any time, we have the invariant: $L = A \mathbin{+\!\!+} [m] \mathbin{+\!\!+} B$, where $a \leq m \leq b, a \in A, b \in B$.

$$min\ (x : xs) = min'\ [\ ]\ x\ xs \tag{9.3}$$

Where:

$$
\begin{array}{rcl}
min' \; as \; m \; [\,] & = & (m, A) \\
min' \; as \; m \; (b : bs) & = & \begin{cases} b < m : & min' \; (m : as) \; b \; bs \\ \text{otherwise} : & min' \; (b : as) \; m \; bs \end{cases}
\end{array}
\tag{9.4}
$$

Function *min* return a pair: the minimum and the remaining elements. We can define selection sort as below:

$$
\begin{array}{rcl}
sort \; [\,] & = & [\,] \\
sort \; xs & = & m : (sort \; xs'), \; \text{where} \; (m, xs') = min \; xs
\end{array}
\tag{9.5}
$$

9.2.1 Performance

Selection sort need scan the unsorted elements to find the minimum for n times. It compares $n + (n-1) + (n-2) + ... + 1$ times. The time bound is $O(\frac{n(n+1)}{2}) = O(n^2)$. Compare to the insertion sort, selection sort performs same in the best, worst, and average cases. While insertion sort performs best at $O(n)$ (the linked-list is in reversed ordered), and worst at $O(n^2)$.

Exercise 9.1

1. What is the problem with below implementation of *min*?

$$
\begin{array}{rcl}
min' \; as \; m \; [\,] & = & (m, A) \\
min' \; as \; m \; (b : bs) & = & \begin{cases} b < m : & min' \; (as \mathbin{+\!\!+} [m]) \; b \; bs \\ \text{否则} : & min' \; (as \mathbin{+\!\!+} [b]) \; m \; bs \end{cases}
\end{array}
$$

2. Implement the selection sort for both in-placed and not.

9.3 Improvement

To sort in ascending, descending, and varies of ordering, we abstract the comparison as $\lhd$.

$$
\begin{array}{rcl}
sortBy \lhd \; [\,] & = & [] \\
sortBy \lhd \; xs & = & m : sortBy \; \lhd \; xs', \; \text{where} \; (m, xs') = minBy \; \lhd \; xs
\end{array}
\tag{9.6}
$$

We also use $\lhd$ to find the 'minimum':

$$
\begin{array}{rcl}
minBy \; \lhd \; [x] & = & (x, [\,]) \\
minBy \; \lhd \; (x : xs) & = & \begin{cases} x \lhd y : & (x, xs), \; \text{where} \; (y, ys) = minBy \; xs \\ \text{otherwise} : & (y, \; x : ys) \end{cases}
\end{array}
\tag{9.7}
$$

For example, we pass the $<$ to sort a collection of numbers in ascending order: $sortBy \; (<) \; [3, 1, 4, ...]$. As the constraint, we need the comparison $\lhd$ satisfy the *strict weak order*[52].

- Irreflexivity: for all x, $x < x$ is false;
- Asymmetry: for all x and y, if $x < y$, then $y < x$ is false;
- Transitivity, for all x, y, and z, if $x < y$, and $y < z$, then $x < z$.

The in-place selection sort traverses all elements, we can find the minimum as an inner loop to make the implementation compact:

```
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]
```

After sort the first $n - 1$ elements, the last one must be the maximum. We can save the last loop. Besides, we needn't swap if the i-th smallest is exactly $A[i]$.

```
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]
```

9.3.1 Cock-tail sort

Knuth gives another selection sort implementation[51]. Select the maximum, but not the minimum, and move it to the tail, as shown in figure ??. At any time, the right most part is sorted. We scan the unsorted part, find the maximum and swap to the right.

```
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]
```

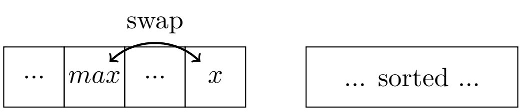


Figure 9.3: Select the maximum and swap to tail

We obtain the ascending order as well. Further, we can pick both the minimum and maximum in one pass, swap the minimum to the head, and the maximum to the tail. We can halve the inner loop times. The method is called 'cock-tail sort'.

```
procedure Sort(A)
    for i ← 1 to ⌊|A|/2⌋ do
        min ← i
        max ← |A| + 1 − i
        if A[max] < A[min] then
            Exchange A[min] ↔ A[max]
        for j ← i + 1 to |A| − i do
            if A[j] < A[min] then
```

```
            min ← j
        if A[max] < A[j] then
            max ← j
    EXCHANGE A[i] ↔ A[min]
    EXCHANGE A[|A| + 1 − i] ↔ A[max]
```

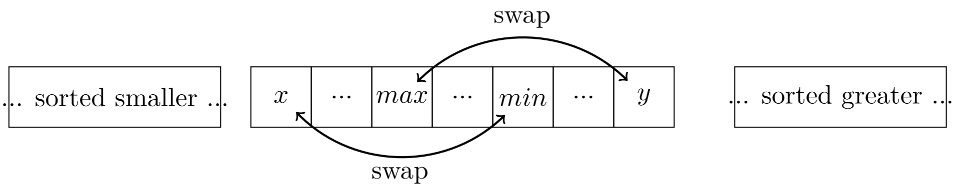


Figure 9.4: Find the minimum and maximum, swap both to the right positions.

It's necessary to swap if the right most element less than the right most one before the inner loop. This is because the scan excludes them. We can also implement the cock-tail sort recursively:

1. If the list is empty or singleton, it's sorted;
2. Otherwise, we select the minimum and the maximum, move them to the head and tail, then recursively sort the rest elements.

$$
\begin{array}{rcl}
sort\ [\ ] & = & [\ ] \\
sort\ [x] & = & [x] \\
sort\ xs & = & a : (sort\ xs') \mathbin{+\!\!+} [b], \text{where } (a, b, xs') = minMax\ xs
\end{array} \tag{9.8}
$$

Where function *minMax* extracts the minimum and maximum from a list:

$$
minMax\ (x : y : xs) = foldr\ sel(min\ x\ y, max\ x\ y, [\ ])\ xs \tag{9.9}
$$

We initialize the minimum as the first element x_0, and the maximum as the second element x_1, and process the list with $foldr$. Function *sel* is defined as:

$$
sel\ x\ (x_0, x_1, xs) = \begin{cases}
x < x_0 : & (x, x_1, x_0 : xs) \\
x_1 < x : & (x_0, x, x_1 : xs) \\
\text{otherwsie} : & (x_0, x_1, x : xs)
\end{cases}
$$

Although *minMax* is bound to $O(n)$ time, $\mathbin{+\!\!+}[b]$ is expensive. As shown in figure 9.4, let the left sorted part be A, the right sorted part be B. We can turn the cock-tail sort to tail recursive with A and B as the accumulators.

$$
\begin{array}{rcl}
sort'\ A\ B\ [\ ] & = & A \mathbin{+\!\!+} B \\
sort'\ A\ B\ [x] & = & A \mathbin{+\!\!+} (x : B) \\
sort'\ A\ B\ (x : xs) & = & sort'\ (A \mathbin{+\!\!+} [x_0])\ xs'\ (x_1 : B)
\end{array} \tag{9.10}
$$

Where $(x_0, x_1, xs') = minMax\ xs$. We pass empty A and B to initialize sorting: $sort = sort'\ [\ ]\ [\ ]$. The append only happens to $A \mathbin{+\!\!+} [x_0]$, while x_1 is linked before B. Every recursion performs an append operation. To eliminate it, we can maintain A in reversed order: $\overleftarrow{A}$, hence x_0 is linked ahead but appended. We have the following equations:

$$
\begin{array}{rcl}
A' & = & A \mathbin{+\!\!+} [x] \\
 & = & reverse\ (x : reverse\ A) \\
 & = & reverse\ (x : \overleftarrow{A}) \\
 & = & \overleftarrow{x : \overleftarrow{A}}
\end{array} \tag{9.11}
$$

Finally, we reverse $\overleftarrow{A'}$ back to A'. We can improve the algorithm as below:

$$
\begin{array}{rcl}
sort'\ A\ B\ [\,] & = & (reverse\ A) +\!\!+ B \\
sort'\ A\ B\ [x] & = & (reverse\ x : A) +\!\!+ B \\
sort'\ A\ B\ (x : xs) & = & sort'\ (x_0 : A)\ xs'\ (x_1 : B)
\end{array}
\tag{9.12}
$$

9.4 Further improvement

Although cock-tail sort halves the loops, it's still bound to $O(n^2)$ time. To sort by comparison, we need the outer loop to examine all the elements for ordering. Do we need scan all the elements to select the minimum every time? After find the first smallest one, we've traversed the whole collection, obtain some information, like which are greater, which are smaller. However, we discard such information for further selection, but restart a fresh scan. The idea is information reusing. Let's see one inspired from football match.

9.4.1 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 in the finals. Let's go back to 1978 and imagine a special way to determine the champion: In the first round, the teams are grouped into 8 pairs to play. There will be 8 winners, and 8 teams will be out. Then in the second round, 8 teams are grouped into 4 pairs. There will be 4 winners. Then the top 4 teams are grouped into 2 pairs, there will be two teams left for the final. The champion is determined after 4 rounds of games. There are total $8+4+2+1 = 15$ games. Besides the champion, we also want to know which is the silver medal team. In the real world cup, the team lost the final is the runner-up. However, it isn't fair in some sense. We often hear about the 'group of death'. Suppose Brazil is grouped with Germam in round one. Although both teams are strong, one team is knocked out. It's quite possible that team would beat other teams except for the champion, as shown in figure 9.5.

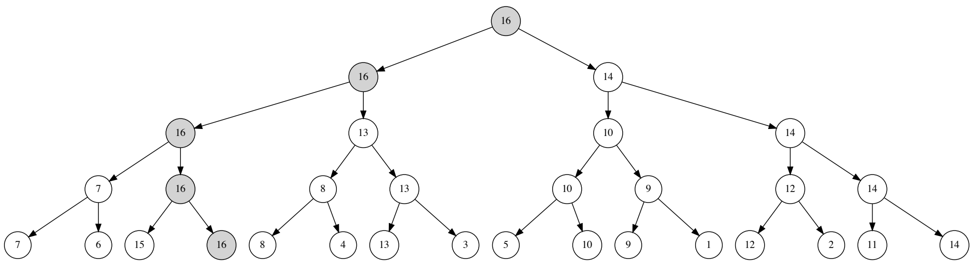
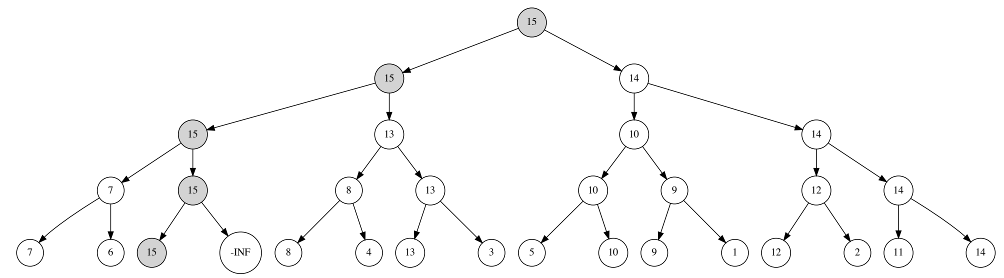


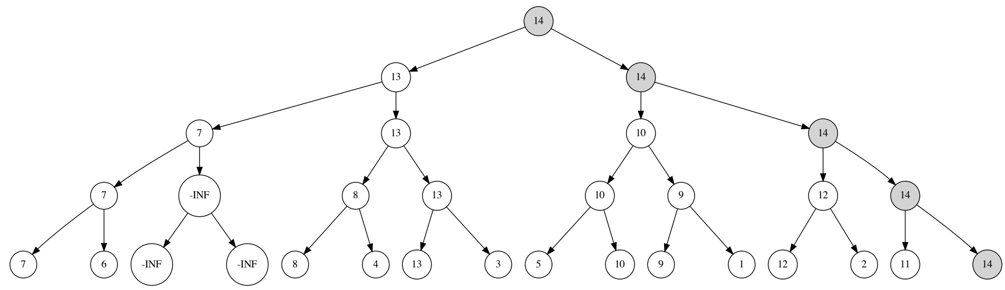
Figure 9.5: The element 15 is knocked out in the first round.

Assign every team a number to measure its strength. Suppose the team with greater number always beats the smaller one (this is obviously not true in real world). The champion number is 16. the runner-up is not 14, but 15, which is out in the first round. We need figure out a way to quickly identify the second greater number in the tournament tree. The apply it to select the 3rd, the 4th, ... to sort. We can mutate the champion to a very small number, i.e. $-\infty$, hence it won't be selected next time, and the previous runner-up will become the new champion. For 2^m teams, where m is some natural number, it takes $2^{m-1} + 2^{m-2} + ... + 2 + 1 = 2^m - 1$ comparisons to determine the new champion. This is same as before. Actually, we needn't perform bottom-up comparisons because the tournament tree stores sufficient ordering information. The champion must beat the runner-up at sometime. We can locate the runner-up along the path from the root to the leaf of the champion. We grey the path in figure 9.5 of $[14, 13, 7, 15]$. This method is defined as below:

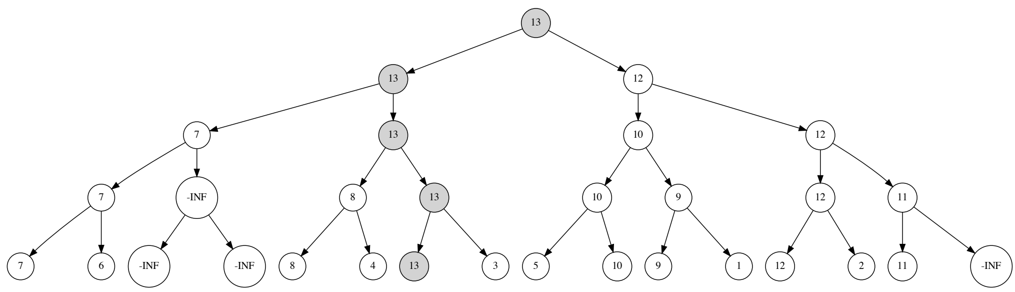
1. Build a tournament tree with the maximum (the champion) at the root;
2. Take the root, replace it with $-\infty$ along the path to leaf;
3. Perform a bottom-up back-track along the path, find the new champion and store it in the root;
4. Repeat step 2 to process all elements.



Take 16, replace with $-\infty$, 15 becomes the new root.



Take 15, replace with $-\infty$, 14 becomes the new root.



Take 14, replace with $-\infty$, 13 becomes the new root.

Figure 9.6: The first 3 steps of tournament tree sort.

To sort a collection of elements, we build a tournament tree from them, repeatedly select the champion from it. Figure 9.6 gives the first 3 steps. We can re-use the binary tree definition. To make back-track easy, we need the parent field in each node. When n is not 2^m form some natural number m, there is remaining element without "player", and directly enters the next round of games. To build the tournament tree, we build n singleton trees from every element. Then pick every two t_1, t_2 to create a bigger binary tree t. Where the root of t is $max(key(t_1), key(t_2))$, the left and right sub-trees are t_1,

t_2. Repeat to obtain a collection of new trees, each height increases by one. If there is remaining, then enters the next round. After this round, trees halve to $\lfloor \frac{n}{2} \rfloor$. Repeat this to obtain the final tournament tree. The process is bound to $O(n + \frac{n}{2} + \frac{n}{4} + ...) = O(2n) = O(n)$ time.

```
function Build-Tree(A)
    T ← [ ]
    for each x ∈ A do
        Append(T, Node(NIL, x, NIL))
    while |T| > 1 do
        T' ← [ ]
        for every t1, t2 ∈ T do
            k ← Max(Key(t1), Key(t2))
            Append(T', Node(t1, k, t2))
        if |T| is odd then
            Append(T', Last(T))
        T ← T'
    return T[1]
```

We replace the root with $-\infty$ top-down, then back-track through the parent field to find the new maximum.

```
function Pop(T)
    m ← Key(T)
    Key(T) ← −∞
    while T is not leaf do                ▷ top-down replace m with −∞.
        if Key(Left(T)) = m then
            T ← Left(T)
        else
            T ← Right(T)
        Key(T) ← −∞
    while Parent(T) ≠ NIL do              ▷ bottom-up to find the new maximum.
        T ← Parent(T)
        Key(T) ← Max(Key(Left(T)), Key(Right(T)))
    return (m, T)                         ▷ the maximum and the new tree.
```

Pop process the tree in two passes, top-down, then bottom-up along the path of the champion. Because the tournament tree is balanced, the length of this path, i.e. height of the tree, is bound to $O(\lg n)$, where n is the number of the elements. Below is the tournament tree sort. We first build the tree in $O(n)$ time, then pop the maximum for n times, each pop takes $O(\lg n)$ time. The total time is bound to $O(n \lg n)$.

```
procedure Sort(A)
    T ← Build-Tree(A)
    for i ← |A| down to 1 do
        A[i] ← Extract-Max(T)
```

We can also implement tournament tree sort recursively. Reuse the binary search tree definition, let an none empty tree be (l, k, r), where k is the element, l, r are the left and right sub-trees. Define $wrap\ x = (\varnothing, x, \varnothing)$ to create a leaf node. We can convert the n elements to a list of n single trees: $ts = map\ wrap\ xs$. For every pair of trees t_1, t_2, we merge them to a bigger tree, pick the greater element as the new root, and t_1, t_2 become the left and right sub-trees.

$$merge\ t_1\ t_2 = (t_1, max\ k_1\ k_2, t_2) \tag{9.13}$$

Where $k_1 = key\ t_1, k_2 = key\ t_2$ are the elements at root respectively. Define a function *build ts* to repeatedly merge two trees, and build the final tournament tree.

$$
\begin{array}{rcl}
build\ [\ ] & = & \varnothing \\
build\ [t] & = & t \\
build\ ts & = & build\ (pairs\ ts)
\end{array}
\tag{9.14}
$$

Where:

$$
\begin{array}{rcl}
pairs\ (t_1 : t_2 : ts) & = & (merge\ t_1\ t_2) : pairs\ ts \\
pairs\ ts & = & ts
\end{array}
\tag{9.15}
$$

When pop the champion, we examine the sub-trees to see which one holds the same element as the root. Then recursively pop the champion from the sub-tree till the leaf node. Then replace it with $-\infty$.

$$
\begin{array}{rcl}
pop\ (\varnothing, k, \varnothing) & = & (\varnothing, -\infty, \varnothing) \\
pop\ (l, k, r) & = & \begin{cases} k = key\ l : & (l', max\ (key\ l')\ (key\ r), r), \text{where} l' = pop\ l \\ k = key\ r : & (l, max\ (key\ l)\ (key\ r'), r'), \text{where} r' = pop\ r \end{cases}
\end{array}
\tag{9.16}
$$

Then repeatedly pop from the tournament tree to sort (in descending order):

$$
\begin{array}{rcl}
sort\ \varnothing & = & [\ ] \\
sort\ (l, -\infty, r) & = & [\ ] \\
sort\ t & = & (key\ t) : sort\ (pop\ t)
\end{array}
\tag{9.17}
$$

Exercise 9.2

1. Implement the recursive tournament tree sort in ascending order.
2. When there are duplicated elements, how to sort it with tournament tree?
3. Compare the tournament tree sort and binary search tree sort in terms of space and time performance.
4. Compare heap sort and tournament tree sort in terms of space and time performance.

9.4.2 Heap sort

We improve the selection based sort to $O(n \lg n)$ time through tournament tree. It is the upper limit of the comparison based sort[51]. However, there are still rooms for improvement. After sort, The binary holds all $-\infty$, occupying $2n$ nodes for n elements. It's there a way to release node after pop? Can we halve $2n$ nodes to n? Treat the tree as empty when the root element is $-\infty$, and rename *key* to *top*, we can write (9.17) in a generic way:

$$
\begin{array}{rcl}
sort\ \varnothing & = & [\ ] \\
sort\ t & = & (top\ t) : sort\ (pop\ t)
\end{array}
\tag{9.18}
$$

This is exactly as same as the definition of heap sort. Heap always stores the minimum (or the maximum) on the top, and provides fast pop operation. The array implementation encodes the binary tree structure as indices, uses exactly n cells to represent the heap. The functional heaps, like the leftist heap and splay heap use n nodes as well. We'll give more well performed heaps in next chapter.

9.5 Appendix - example programs

Tail recursive selection sort:

```
sort [] = []
sort xs = x : sort xs'
  where
    (x, xs') = extractMin xs

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
```

Cock-tail sort:

```
[A] cocktailSort([A] xs) {
    Int n = length(xs)
    for Int i = 0 to n / 2 {
        var (mi, ma) = (i, n - 1 -i)
        if xs[ma] < xs[mi] then swap(xs[mi], xs[ma])
        for Int j = i + 1 to n - 1 - i {
            if xs[j] < xs[mi] then mi = j
            if xs[ma] < xs[j] then ma = j
        }
        swap(xs[i], xs[mi])
        swap(xs[n - 1 - i], xs[ma])
    }
    return xs
}
```

Tail recursive cock-tail sort:

```
csort xs = cocktail [] [] xs
  where
    cocktail as bs []  = reverse as ++ bs
    cocktail as bs [x] = reverse (x:as) ++ bs
    cocktail as bs xs  = let (mi, ma, xs') = minMax xs
                         in cocktail (mi:as) (ma:bs) xs'

minMax (x:y:xs) = foldr sel (min x y, max x y, []) xs
  where
    sel x (mi, ma, ys) | x < mi = (x, ma, mi:ys)
                       | ma < x = (mi, x, ma:ys)
                       | otherwise = (mi, ma, x:ys)
```

Build the tournament tree (reuse the binary tree structure):

```
Node<T> build([T] xs) {
    [T] ts = []
    for x in xs {
        append(ts, Node(null, x, null))
    }
    while length(ts) >  1 {
        [T] ts' = []
        for l, r in ts {
            append(ts', Node(l, max(l.key, r.key), r))
        }
        if odd(length(ts)) then append(ts', last(ts))
        ts = ts'
    }
    return ts[0];
}
```

Pop from the tournament tree:

```
T pop(Node<T> t) {
    T m = t.key
    t.key = -INF
    while not isLeaf(t) {
        t = if t.left.key == m then t→left else t→right
        t.key = -INF
    }
    while (t.parent ≠ null) {
        t = t.parent
        t.key = max(t.left.key, t.right.key)
    }
    return (m, t);
}
```

Tournament tree sort:

```
void sort([A] xs) {
    Node<T> t = build(xs)
    for Int n = length(xs) - 1 downto 0 {
        (xs[n], t) = pop(t)
    }
}
```

Recursive tournament tree sort (descending order):

```
data Tr a = Empty | Br (Tr a) a (Tr a)

data Infinite a = NegInf | Only a | Inf deriving (Eq, Ord)

key (Br _ k _ ) = k

wrap x = Br Empty (Only x) Empty

merge t1@(Br _ k1 _) t2@(Br _ k2 _) = Br t1 (max k1 k2) t2

fromList = build ∘ (map wrap) where
  build [] = Empty
  build [t] = t
  build ts = build (pairs ts)
  pairs (t1:t2:ts) = (merge t1 t2) : pair ts
  pairs ts = ts

pop (Br Empty _ Empty) = Br Empty NegInf Empty
pop (Br l k r) | k == key l = let l' = pop l in Br l' (max (key l') (key r)) r
               | k == key r = let r' = pop r in Br l (max (key l) (key r')) r'

toList Empty = []
toList (Br _ Inf _) = []
toList t@(Br _ Only k _) = k : toList (pop t)

sort = toList ∘ fromList
```

Chapter 10

Binomial heap, Fibonacci heap, and pairing heap

10.1 Introduction

Binary heap stores elements in a binary tree, we can extend it to k-ary tree[54] ($k > 2$ multi-ways tree), or multiple trees. This chapter introduces binomial heap, which consists of forest of k-ary trees. When delay some operations to a Binomial heap, we obtained Fibonacci heap. It improves the heap merge performance from $O(\lg n)$ time bound to amortized constant time. This is critical for graph algorithm design. We give pairing heap as a simplified heap implementation with good overall performance.

10.2 Binomial Heaps

Binomial heap is named after Newton's binomial theorem. It consists of a set of k-ary trees (also called a forest). Every tree has the size equal to a binomial coefficient. Newton proved that $(a + b)^n$ expands to:

$$(a+b)^n = a^n + \binom{n}{1} a^{n-1} b + ... + \binom{n}{n-1} a b^{n-1} + b \tag{10.1}$$

When n is a natural number, the coefficients is some row in Pascal's triangle[1] [55].

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
...
```

The first row is 1, all the first and last numbers are 1 for every row. Any other number is the sum of the top-left and top-right numbers in the previous row. There are many methods to generate pascal triangles, like recursion.

[1]Also know as the *Jia Xian*'s triangle named after ancient Chinese mathematician Jia Xian (1010-1070). Newton generalized n to rational numbers, later Euler expand it to real exponents.

Binomial tree

A binomial tree is a multi-ways tree with an integer rank. Denoted as B_0 if the rank is 0, and B_n for rank n.

1. B_0 has only one node;
2. B_n is formed by two B_{n-1} trees, the one with the greater root element is the left most sub-tree of the other, as shown in figure 10.1.

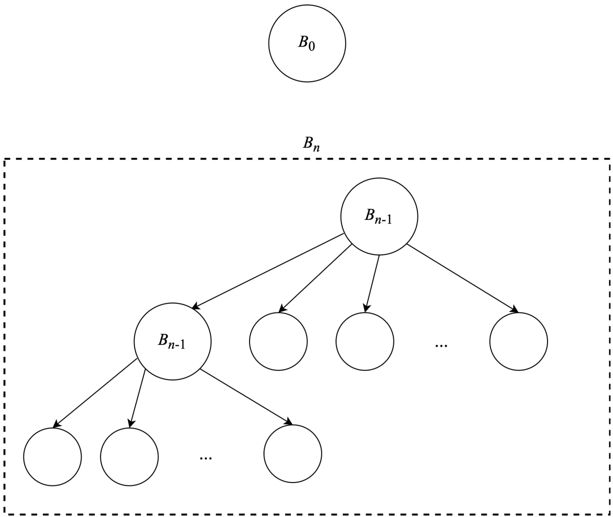


Figure 10.1: Binomial tree

Figure 10.2 gives examples of B_0 to B_4.

We can find the number of nodes in every row in B_n is a binomial coefficient. For example in B_4, there is a node (root) in level 0, 4 nodes in level 1, 6 nodes in level 2, 4 nodes in level 3, and a node in level 4. They are exactly same as the 4th row (start from 0) of Pascal's triangle: 1, 4, 6, 4, 1. This is the reason why we name it binomial tree. We can further know there are 2^n elements in a B_n tree.

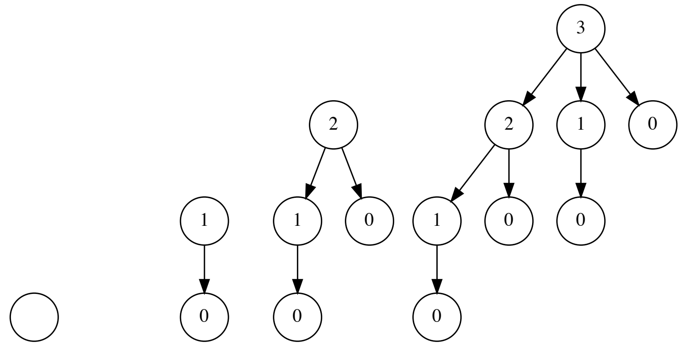
A binomial heap is a set of binomial trees (a forest) that satisfies the following two rules:

1. Every tree satisfies the *heap property*, i.e. for min heap, the element in every node is not less than ($\geq$) its parent;
2. Every tree has unique rank. i.e. any two trees have different ranks.

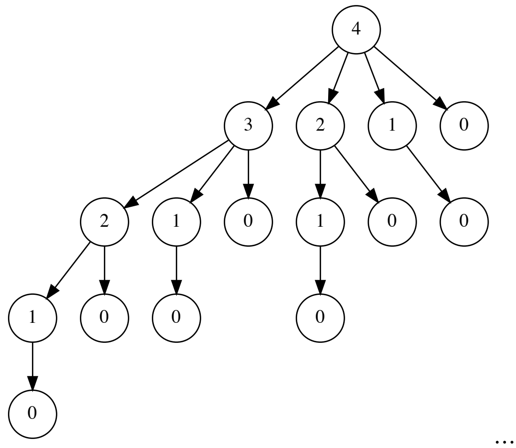
From the 2nd rule, for a binomial heap of n elements, convert n to its binary format $(a_m ... a_1, a_0)_2$, where a_0 is the least significant bit (LSB) and a_m is the most significant bit (MSB). If if $a_i = 0$, there is no tree of rank i; if $a_i = 1$, there is a tree of rank i. For example, consider a binomial heap of 5 elements. As 5 is 101 in binary, there are 2 binomial trees, one is B_0, the other is B_2. The binomial heap in figure 10.3 has 19 elements, 19 is $(10011)_2$. There is a B_0, a B_1, and a B_4.

We define the binomial tree as (r, k, ts), where r is the rank, k is the element in the root, and ts is the list of sub-trees ordered by rank.

```
data BiTree a = Node Int a [BiTree a]

type BiHeap a = [BiTree a]
```



(a) B_0 (b) B_1 (c) B_2 (d) B_3



(e) B_4

Figure 10.2: Binomial trees of rank 0, 1, 2, 3, 4, ...

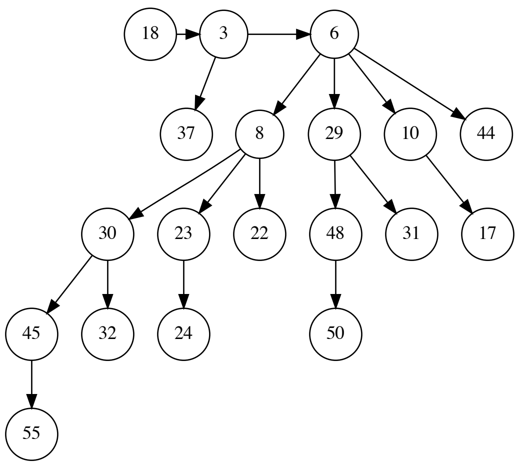


Figure 10.3: A binomial heap with 19 elements

There is a method called 'left-child, right-sibling'[4], that can reuse the binary tree data structure to define multi-ways tree. Every node has the left and right part. the left references to the first sub-tree; the right references to its sibling. All siblings form a list as shown in figure 10.4. Alternatively, we can use an array or a list to represent the sub-trees.

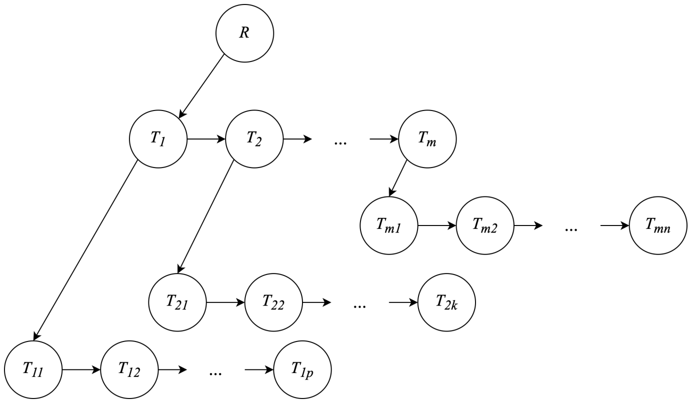


Figure 10.4: R is the root, $T_1, T_2, ..., T_m$ are sub-trees of R. The left of R is T_1, the right is NIL. $T_{11}, ..., T_{1p}$ are sub-trees of T_1. The left of T_1 is T_{11}, the right is its sibling T_2. The left of T_2 is T_{21}, the left is sibling.

10.2.1 Link

To link two B_n trees to a B_{n+1} tree, we compare the two root elements, choose the smaller one as the root, and put the other tree ahead of other sub-trees as shown in figure 10.5.

$$link\ (r, x, ts)\ (r, y, ts') = \begin{cases} x < y : & (r+1, x, (r, t, ts') : ts) \\ \text{otherwise} : & (r+1, y, (r, x, ts) : ts') \end{cases} \tag{10.2}$$

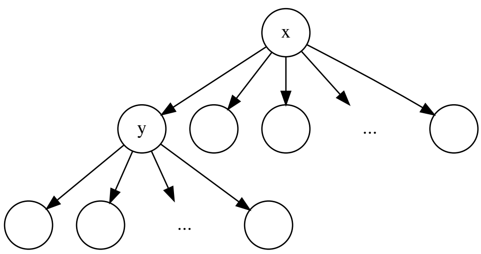


Figure 10.5: If $x < y$, link y as the first sub-tree of x.

We can implement link with 'left child, right sibling' method as below. Link operation is bound to constant time.

```
function Link(x, y)
    if Key(y) < Key(x) then
        Exchange x ↔ y
    Sibling(y) ← Sub-Trees(T1)
    Sub-Trees(x) ← y
    Parent(y) ← x
```

```
    RANK(x) ← RANK(y) + 1
    return x
```

Exercise 10.1

1. Write a program to generate Pascal's triangle.
2. Prove that the i-th row in tree B_n has $\binom{n}{i}$ nodes.
3. Prove there are 2^n elements in B_n tree.
4. Use a container to store sub-trees, how to implement link? How to secure the operation is in constant time?

Insert

When insert a new tree, we keep the trees in binomial heap ordered by rank (ascending):

$$\begin{array}{rcl} ins\ t\ [\,] & = & [t] \\ ins\ t\ (t' : ts) & = & \begin{cases} rank\ t < rank\ t' : & t : t' : ts \\ rank\ t' < rank\ t : & t' : ins\ t\ ts \\ \text{otherwise} : & ins\ (link\ t\ t')\ ts \end{cases} \end{array} \tag{10.3}$$

Where $rank\ (r, k, ts) = r$ gives the rank of a tree. For empty heap $[\,]$, it becomes a single list of the new tree t; otherwise, we compare the rank of t with the first tree t', if t has less rank, then it becomes the new first one; if t' has less rank, we recursively insert t to the rest trees; if they have the same rank, then link t and t' to a bigger tree, and recursively insert to the rest. For n elements, there are at most $O(\lg n)$ binomial trees in the heap. ins links $O(\lg n)$ time at most, as linking is bound to constant time, the overall performance is bound to $O(\lg n)$[2]. We can define insert for binomial heap with ins. First wrap the new element x in a singleton tree, then insert the tree to the heap:

$$insert\ x = ins\ (0, x, [\,]) \tag{10.4}$$

This is a Curried definition, we can further insert a list of elements to the heap by using fold:

$$fromList = foldr\ insert\ [\,] \tag{10.5}$$

Below is the implementation with 'left child, right sibling' method:

```
function INSERT-TREE(T, H)
    ⊥← p ← NODE(0, NIL, NIL)
    while H ≠ NIL 且 RANK(H) ≤ RANK(T) do
        T1 ← H
        H ← SIBLING(H)
        if RANK(T) = RANK(T1) then
            T ← LINK(T, T1)
        else
            SIBLING(p) ← T1
            p ← T1
    SIBLING(p) ← T
    SIBLING(T) ← H
    return REMOVE-FIRST(⊥)
```

[2] It's similar to adding two binary numbers. A more generic topic is *numeric representation*[3].

```
function Remove-First(H)
    n ← Sibling(H)
    Sibling(H) ← NIL
    return n
```

10.2.2 Merge

When merge two binomial heaps, we actually merge two lists of binomial trees. Every tree has unique rank in merged result, and the ranks are in ascending order. The tree merge process is similar to merge sort. Every time, we pick the first tree from each heap, compare their ranks, put the smaller one to the result. If the two trees have the same rank, we link them to a bigger one, and recursively insert to the merge result.

$$
\begin{array}{rcl}
merge\ ts_1\ [\,] & = & ts_1 \\
merge\ [\,]\ ts_2 & = & ts_2 \\
merge\ (t_1 : ts_1)\ (t_2 : ts_2) & = & \begin{cases} rank\ t_1 < rank\ t_2 : & t_1 : (merge\ ts_1\ (t_2 : ts_2)) \\ rank\ t_2 < rank\ t_1 : & t_2 : (merge\ (t_1 : ts_1)\ ts_2) \\ \text{otherwise} : & ins\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2) \end{cases}
\end{array}
\tag{10.6}
$$

Alternatively, when t_1 and t_2 have the same rank, we can insert the linked tree back to either heap, and recursively merge:

$$merge\ (ins\ (link\ t_1\ t_2)\ ts_1)\ ts_2$$

We can also eliminate recursion, and implement iterative merge:

```
function Merge(H1, H2)
    H ← p ← Node(0, NIL, NIL)
    while H1 ≠ NIL and H2 ≠ NIL do
        if Rank(H1) < Rank(H2) then
            Sibling(p) ← H1
            p ← Sibling(p)
            H1 ← Sibling(H1)
        else if Rank(H2) < Rank(H1) then
            Sibling(p) ← H2
            p ← Sibling(p)
            H2 ← Sibling(H2)
        else                                        ▷ same rank
            T1 ← H1, T2 ← H2
            H1 ← Sibling(H1), H2 ← Sibling(H2)
            H1 ← Insert-Tree(Link(T1, T2), H1)
    if H1 ≠ NIL then
        Sibling(p) ← H1
    if H2 ≠ NIL then
        Sibling(p) ← H2
    return Remove-First(H)
```

If there are m_1 trees in H_1, m_2 trees in H_2. There are at most $m_1 + m_2$ trees after merge. The merge is bound to $O(m_1 + m_2)$ time if all trees have different ranks. If there exist trees of the same rank, we call *ins* up to $O(m_1 + m_2)$ times. Consider $m_1 = 1 + \lfloor \lg n_1 \rfloor$ and $m_2 = 1 + \lfloor \lg n_2 \rfloor$, where n_1, n_2 are the numbers of elements in each heap, and $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor \leq 2 \lfloor \lg n \rfloor$, where $n = n_1 + n_2$. The final performance of merge is $O(\lg n)$.

Pop

Although every tree has the minimal element in its root, we don't know which tree holds the overall minimum in the heap. We need locate it from all trees. As there are $O(\lg n)$ trees, it takes $O(\lg n)$ time to find the top element. For pop, we need further remove the top element and maintain heap property. Let the trees be $B_i, B_j, ..., B_p, ..., B_m$ in the heap, and the minimum is in the root of B_p. After remove the top, there leave p sub binomial trees with ranks of $p-1, p-2, ..., 0$. We can reverse them to form a new binomial heap H_p. The other trees without B_p also form a binomial heap $H' = H - [B_p]$. We merge H_p and H' to get the final result as shown in figure 10.6. Below is the definition to access the minimal element in the heap.

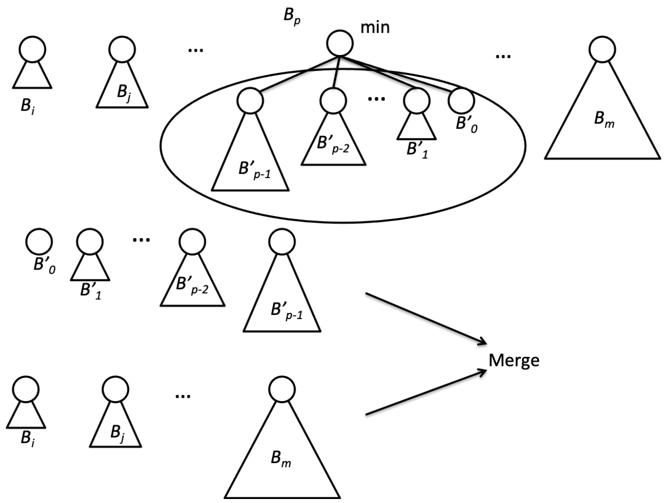


Figure 10.6: Binomial heap pop.

$$top\ (t:ts) = foldr\ f\ (key\ t)\ ts \tag{10.7}$$
$$f\ (r,x,ts)\ y = min\ x\ y$$

It's means to traverse all trees and find the which root has the minimum.

1: **function** TOP(H)
2: $m \leftarrow \infty$
3: **while** $H \neq$ NIL **do**
4: $m \leftarrow$ MIN(m, KEY(H))
5: $H \leftarrow$ SIBLING(H)
6: **return** m

To support pop, we need extract the tree containing the minimum out:

$$\begin{array}{rcl} min'\ [t] & = & (t, [\,]) \\ min'\ (t:ts) & = & \begin{cases} key\ t < key\ t' : & (t, ts), 其中 : (t', ts') = min'\ ts \\ 否则 : & (t', t:ts') \end{cases} \end{array} \tag{10.8}$$

Where $key\ (r,k,ts) = k$ accesses the root element, the result of min' is a pair: the tree containing the minimum and the remaining trees. We next define pop with it:

$$pop\ H = (k, merge\ (reverse\ ts)\ H'), 其中 : ((r,k,ts), H') = min'\ H \tag{10.9}$$

The iterative implementation is as below:

```
function Pop(H)
    (T_m, H) ← Extract-Min(H)
    H ← Merge(H, Reverse(Sub-Trees(T_m)))
    Sub-Trees(T_m)
    return (Key(T_m), H)
```

Where the list reverse is defined in chapter 1, Extract-Min is implemented as below:

```
function Extract-Min(H)
    H' ← H, p ← NIL
    T_m ← T_p ← NIL
    while H ≠ NIL do
        if T_m = NIL or Key(H) < Key(T_m) then
            T_m ← H
            T_p ← p
        p ← H
        H ← Sibling(H)
    if T_p ≠ NIL then
        Sibling(T_p) ← Sibling(T_m)
    else
        H' ← Sibling(T_m)
    Sibling(T_m) ← NIL
    return (T_m, H')
```

We can implement heap sort with *pop*. First build a binomial heap from a list of elements, then repeatedly pop the smallest element.

$$sort = heapSort \circ fromList \tag{10.10}$$

Where *heapSort* is defined as:

$$\begin{array}{rcl} heapSort\ [\ ] &=& [\ ] \\ heapSort\ H &=& k : (heapSort\ H'), \text{where} : (k, H') = pop\ H \end{array} \tag{10.11}$$

Binomial heap insert and merge are bound to $O(\lg n)$ time in worst case, their amortized performance are constant time, we skip the proof.

10.3 Fibonacci heap

Binomial heap is named from binomial theorem, Fibonacci heap is named after Fibonacci numbers[3]. Fibonacci heap is essentially a 'lazy' binomial heap. It delays some operation. However, it does not mean the binomial heap turns to be Fibonacci heap automatically in lazy evaluation environment. Such environment only makes the implementation easy[56]. All operations except for pop are bound to amortized constant time[57].

When insert new element x to a binomial heap, we wrap x to a single tree, then insert to the forest. We keep the rank ordering, if two ranks are same, we link them, and recursively insert. The performance is bound to $O(\lg n)$ time. Taking lazy strategy, we delay the ordered (by rank) insert and link later. Put the single tree of x directly to the forest. To access the top element in constant time, we need record which tree has the minimum in its root. A Fibonacci heap is either empty $\varnothing$, or a forest of trees denoted as (n, t_m, ts). Where t_m is the tree holds the minimal element, n is the number of elements

[3] Michael L. Fredman and Robert E. Tarjan, used Fibonacci numbers to prove the performance time bound, they decided to use Fibonacci to name this data structure.[4]

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 $O(\lg n)$

Table 10.1: Performance of Fibonacci heap and binomial heap

in the heap, and ts is the rest trees. Below example program defines Fibonacci heap (reused the definition of binomial tree).

```
data FibHeap a = E | FH { size :: Int
                        , minTree :: BiTree a
                        , trees :: [BiTree a]}
```

We can access the top element in constant time: $top\ H = key\ minTree\ H$.

10.3.1 Insert

We define insert as a special case of merge: one heap contains a singleton tree:

$$insert\ x\ H = merge\ (singleton\ x)\ H$$

Or simplified in Curried form:

$$insert = merge \circ singleton \tag{10.12}$$

$$singleton\ x = (1, (1, x, [\,]), [\,])$$

We can also implement insert as add a tree to the forest, then update the reference to the tree holds the minimum.

1: **function** INSERT(k, H)
2: $x \leftarrow$ SINGLETON(k) ▷ wrap k to a tree
3: ADD(x, TREES(H))
4: $T_m \leftarrow$ MIN-TREE(H)
5: **if** T_m = NIL or $k <$ KEY(T_m) **then**
6: MIN-TREE(H) $\leftarrow x$
7: SIZE(H) $\leftarrow$ SIZE(H) + 1

Where TREES(H) access the list of trees in H, MIN-TREE(H) returns the tree that holds the minimal element.

Merge

Different from binomial heap, we delay the link operation, but only put the trees from two heaps together, and pick the new top element.

$$\begin{array}{rcl}
merge\ h\ \varnothing &=& h \\
merge\ \varnothing\ h &=& h \\
merge\ (n, t_m, ts)\ (n', t'_m, ts') &=& \begin{cases} key\ t_m < key\ t'_m : & (n + n', t_m, t'_m : ts \mathbin{+\!\!+} ts') \\ \text{otherwise} : & (n + n', t'_m, t_m : ts \mathbin{+\!\!+} ts') \end{cases}
\end{array} \tag{10.13}$$

When neither tree is empty, the $\mathbin{+\!\!+}$ takes time that is proportion to the number of trees in one heap. We can improve it to constant time with doubly linked-list to store trees as shown in below example program.

```
data Node<K> {
    K key
    Int rank
    Node<k> next, prev, parent, subTrees
}

data FibHeap<K> {
    Int size
    Node<K> minTree, trees
}
```

```
function Merge(H1, H2)
    H ← Fib-Heap
    Trees(H) ← Concat(Trees(H1), Trees(H2))
    if Key(Min-Tree(H1)) < Key(Min-Tree(H2)) then
        Min-Tree(H) ← Min-Tree(H1)
    else
        Min-Tree(H) ← Min-Tree(H2)
    Size(H) = Size(H1) + Size(H2)
    return H

function Concat(s1, s2)
    e1 ← Prev(s1)
    e2 ← Prev(s2)
    Next(e1) ← s2
    Prev(s2) ← e1
    Next(e2) ← s1
    Prev(s1) ← e2
    return s1
```

Pop

As the link operation is delayed to future during merge, we need 'compensate' it during pop. We define it as tree consolidation. Consider another problem: given a list of numbers of 2^m (m is natural numbers), for e.g., $L = [2, 1, 1, 4, 8, 1, 1, 2, 4]$, we repeatedly sum the two equal numbers until all numbers are unique. The result is $[8, 16]$. This process is shown in table 10.2. The first column gives the number we are 'scanning'; the second is the middle step, i.e. compare current number and the first number in result list, add them when equal; the last column is the merge result, which inputs to the next step. The consolidation process can be defined with fold:

number	compare, add	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

Table 10.2: Consolidation steps.

$$consolidate = foldr\ melt\ [\,] \tag{10.14}$$

Where *melt* is defined as below:

$$\begin{array}{rcl} melt\ x\ [\,] & = & x \\ melt\ x\ (x' : xs) & = & \begin{cases} x = x' : & melt\ 2x\ xs \\ x < x' : & x : x' : xs \\ x > x' : & x' : melt\ x\ xs \end{cases} \end{array} \tag{10.15}$$

Let $n = sum\ L$, the sum of all numbers. *consolidate* actually represent n in binary format. If the i-th bit is 1, then the result contains 2^i (i starts from 0). For e.g., $sum[2, 1, 1, 4, 8, 1, 1, 2, 4] = 24$. It's 11000 in binary, the 3rd and 4th bit are 1, hence the result contains $2^3 = 8, 2^4 = 16$. We can consolidate trees in similar way: compare the rank, and link the trees:

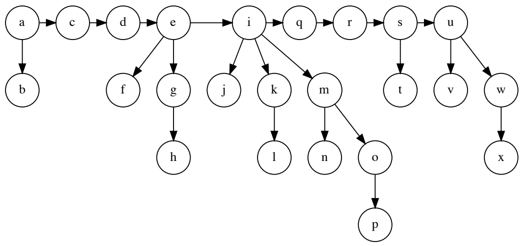
$$\begin{array}{rcl} melt\ t\ [\,] & = & [t] \\ melt\ t\ (t' : ts) & = & \begin{cases} rank\ t = rank\ t' : & melt\ (link\ t\ t')\ ts \\ rank\ t < rank\ t' : & t : t' : ts \\ rank\ t > rank\ t' : & t' : melt\ t\ ts \end{cases} \end{array} \tag{10.16}$$

Figure 10.7 gives the consolidation steps. It is similar to number consolidation when compare with table 10.2. We can use an auxiliary array A to do the consolidation. $A[i]$ stores the tree of rank i. We traverse the trees in the heap. If meet another tree of rank i, we link them together to obtain a bigger tree of rank $i + 1$, clean $A[i]$, and next check whether $A[i + 1]$ is empty or not. If there is a tree of rank $i + 1$, then link them together again. Array A stores the final consolidation result after traverse.

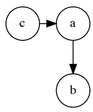
```
function Consolidate(H)
    R ← Max-Rank(Size(H))
    A ← [NIL, NIL, ..., NIL]                    ▷ total R cells
    for each T in Trees(H) do
        r ← Rank(T)
        while A[r] ≠ NIL do
            T' ← A[r]
            T ← Link(T, T')
            A[r] ← NIL
            r ← r + 1
        A[r] ← T
    T_m ← NIL
    Trees(H) ← NIL
    for each T in A do
        if T ≠ NIL then
            append T to Trees(H)
            if T_m = NIL or Key(T) < Key(T_m) then
                T_m ← T
    Min-Tree(H) ← T_m
```

It becomes a binomial heap after consolidation. There are $O(\lg n)$ trees. Max-Rank(n) returns the upper limit of rank R in a heap of n elements. From the binomial tree result, the biggest tree B_R has 2^R elements. We have $2^R \leq n < 2^{R+1}$, we estimate the rough upper limit is $R \leq \log_2 n$. We'll give more accurate estimation of R in later section. We need additionally scan all trees, find the minimal root element. We can reuse min' defined in (10.8) to extract the min-tree.

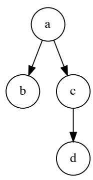
$$\begin{array}{rcl} pop\ (1, (0, x, [\,]), [\,]) & = & (x, [\,]) \\ pop\ (n, (r, x, ts_m), ts) & = & (x, (n - 1, t_m, ts')) \end{array} \tag{10.17}$$



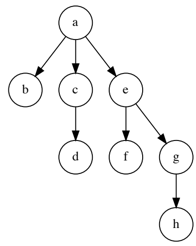
Before



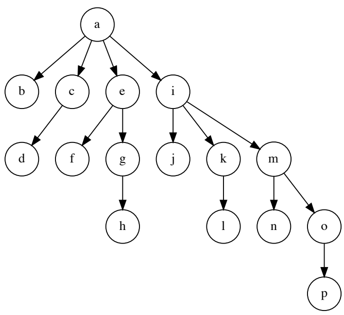
Step 1, 2



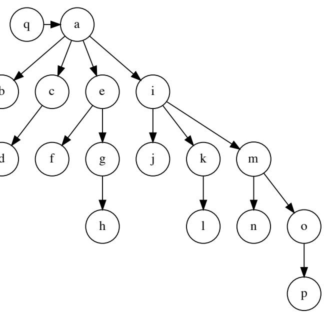
Step 3, link d and c, then link a.



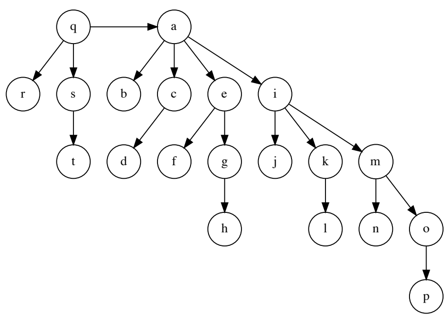
Step 4



Step 5



Step 6



Step 7, 8, link r and q, then link s and q.

Figure 10.7: Consolidation

Where $(t_m, ts') = min'\ consolidate\ (ts_m \mathbin{+\!\!+} ts)$. It takes $O(|ts_m|)$ time for $\mathbin{+\!\!+}$ to concatenate trees. The corresponding iterative implementation is as below:

```
function Pop(H)
    T_m ← Min-Tree(H)
    for each T in Sub-Trees(T_m) do
        append T to Trees(H)
        Parent(T) ← NIL
    remove T_m from Trees(H)
    Size(H) ← Size(H) - 1
    Consolidate(H)
    return (Key(T_m), H)
```

We use the 'potential' method to evaluate the amortized performance. The gravity potential energy in physics is defined as:

$$E = mgh$$

As shown in figure 10.8, consider some process, that moves an object of mass m up and down, and finally stops at height h'. Let the friction resistance be W_f, the process works the following power:

$$W = mg(h' - h) + W_f$$

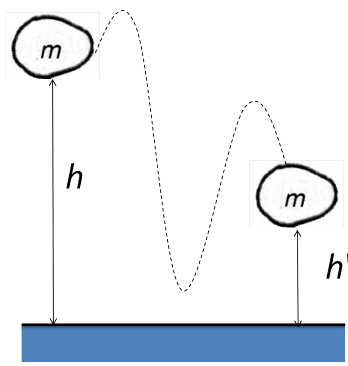


Figure 10.8: Gravity potential energy.

Consider heap pop. To evaluate the cost, let the potential be $\Phi(H)$ before pop. It is the result accumulated by a series of insert and merge operations. The heap becomes H' after tree consolidation. The new potential is $\Phi(H')$. The difference between $\Phi(H')$ and $\Phi(H)$, plus the cost of tree consolidation give the amortized performance. Define the potential as:

$$\Phi(H) = t(H) \tag{10.18}$$

Where $t(H)$ is the number of trees in the heap. Let the upper bound of rank for all trees as $R(n)$, where n is the number of elements in the heap. After tree consolidation, there are at most $t(H') = R(n) + 1$ trees. Before consolidation, there is another operation contributes to running time. we removed the root of min-tree, then add all sub-trees to the heap. We consolidate at most $R(n) + t(H) - 1$ trees. Let the pop performance bound to T, the consolidation bound to T_c, the amortized time is given as below:

$$\begin{array}{rcl} T & = & T_c + \Phi(H') - \Phi(H) \\ & = & O(R(n) + t(H) - 1) + (R(n) + 1) - t(H) \\ & = & O(R(n)) \end{array} \tag{10.19}$$

Insert, merge, and pop ensure all trees are binomial trees, therefore, the upper bound of $R(n)$ is $O(\lg n)$.

10.3.2 Increase priority

We can use heap to manage tasks with priority. When need prioritize a task, we decrease the corresponding element, making it close to the heap top. Some graph algorithms, like the minimum spanning tree and Dijkstra's algorithm rely on this heap operation[4] meet amortized constant time. Let x be a node in the heap H, we need decrease its value to k. As shown in figure 10.9, if the element in x is less than the one in its parent y, we cut x off y, the add it the heap (forest). Although it ensures the parent still holds the minimum in the tree, it is not binomial tree any more. The performance drops when loss too many sub-trees. We add another rule to address this problem: *If a node losses its second sub-tree, it is immediately cut from parent, and added to the heap (forest).*

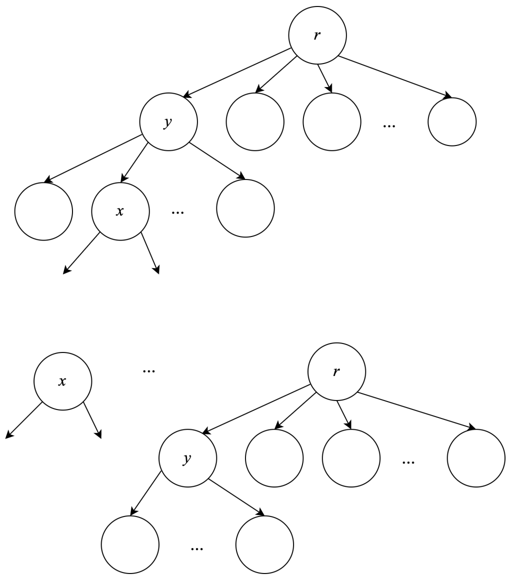


Figure 10.9: If $key\ x < key\ y$, cut x off and add to the heap.

function DECREASE(H, x, k)
    KEY$(x) \leftarrow k$
    $p \leftarrow$ PARENT(x)
    if $p \neq$ NIL and $k <$ KEY(p) **then**
        CUT(H, x)
        CASCADE-CUT(H, p)
    if $k <$ TOP(H) **then**
        MIN-TREE$(H) \leftarrow x$

Where function CASCADE-CUT uses a mark to record whether a node lost sub-tree before. The mark is cleared later in CUT function.

function CUT(H, x)
    $p \leftarrow$ PARENT(x)
    remove x from p
    RANK$(p) \leftarrow$ RANK(p) - 1
    add x to TREES(H)
    PARENT$(x) \leftarrow$ NIL

```
    MARK(x) ← False
```

During cascade cut, if node x is marked, it has lost some sub-tree before. We need recursively cut along the parent till root.

```
function CASCADE-CUT(H, x)
    p ← PARENT(x)
    if p ≠ NIL then
        if MARK(x) = False then
            MARK(x) ← True
        else
            CUT(H, x)
            CASCADE-CUT(H, p)
```

Exercise 10.2

Prove DECREASE is bound to amortized $O(1)$ time.

10.3.3 The name of Fibonacci heap

We are yet to implement MAX-RANK(n). It defines the upper bound of tree rank for a Fibonacci heap of n elements.

Lemma 10.3.1. *For any tree x in a Fibonacci Heap, let $k = rank(x)$, and $|x| = size(x)$, then*

$$|x| \geq F_{k+2} \tag{10.20}$$

Where F_k is the k-th Fibonacci number:

$$\begin{array}{rcl} F_0 & = & 0 \\ F_1 & = & 1 \\ F_k & = & F_{k-1} + F_{k-2} \end{array}$$

Proof. For tree x, let its k sub-trees be $y_1, y_2, ..., y_k$, ordered by the time when they are linked to x. Where y_1 is the first, and y_k is the latest. Obviously, $|y_i| \geq 0$. When link y_i to x, there have already been sub-trees of $y_1, y_2, ..., y_{i-1}$. Because we only link nodes of the same rank, by that time we have:

$$rank(y_i) = rank(x) = i - 1$$

After that, y_i can lost additional sub-tree at most, (through the DECREASE). Once loss the second sub-tree, it will be cut off then add to the forest. For any $i = 2, 3, ..., k$, we have:

$$rank(y_i) \geq i - 2$$

Let s_k be the *minimum possible size* of tree x, where $k = rank(x)$. It starts from $s_0 = 1$, $s_1 = 2$. i.e. there is at least a node in tree of rank 0, at least two nodes in tree of rank 1, at least k nodes in tree of rank k.

$$\begin{array}{rcl} |x| & \geq & s_k \\ & = & 2 + s_{rank(y_2)} + s_{rank(y_3)} + ... + s_{rank(y_k)} \\ & \geq & 2 + s_0 + s_1 + ... + s_{k-2} \end{array}$$

The last row holds because $rank(y_i) \geq i - 2$, and s_k is monotonic, hence $s_{rank(y_i)} \geq s_{i-2}$. We next show that $s_k > F_{k+2}$. Apply induction. For edge case, $s_0 = 1 \geq F_2 = 1$, and $s_1 = 2 \geq F_3 = 2$; For induction case $k \geq 2$.

$$\begin{array}{rcll} |x| & \geq & s_k & \\ & \geq & 2 + s_0 + s_1 + ... + s_{k-2} & \\ & \geq & 2 + F_2 + F_3 + ... + F_k & \text{induction hypothesis} \\ & = & 1 + F_0 + F_1 + F_2 + ... + F_k & \text{from} F_0 = 0, F_1 = 1 \end{array}$$

Next, we prove:

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i \tag{10.21}$$

Use induction again:

- Edge case, $F_2 = 1 + F_0 = 2$
- Induction case, suppose it's true for $k + 1$.

$$\begin{array}{rcll} F_{k+2} & = & F_{k+1} + F_k & \\ & = & (1 + \sum_{i=0}^{k-1} F_i) + F_k & \text{induction hypothesis} \\ & = & 1 + \sum_{i=0}^{k} F_i & \end{array}$$

Wrap up to the final result:

$$n \geq |x| \geq F_{k+2} \tag{10.22}$$

□

For Fibonacci sequence, $F_k \geq \phi^k$, where $\phi = \frac{1 + \sqrt{5}}{2}$ is the golden ratio. We prove that pop is amortized $O(\lg n)$ algorithm. We can define *maxRank* as:

$$MaxRank(n) = 1 + \lfloor \log_\phi n \rfloor \tag{10.23}$$

We can also implement MAX-DEGREE from Fibonacci numbers:

```
function MAX-RANK(n)
    F_0 ← 0, F_1 ← 1
    k ← 2
    repeat
        F_k ← F_{k_1} + F_{k_2}
        k ← k + 1
    until F_k < n
    return k − 2
```

10.4 Pairing Heaps

It's complex to implement Fibonacci heap. Pairing heap provides another option. It's easy to implement, and the performance is good. Most operations, like insert, top, merge are bound to constant time. the pop is conjectured to be amortized $O(\lg n)$ time[58] [3].

10.4.1 Definition

A pairing heap is a multi-way tree. The root holds the minimum. A pairing heap is either empty $\varnothing$, or a k-ary tree, consists of a root and multiple sub-trees, denoted as (x, ts). We can also use 'left child, right sibling' way to define the tree.

```
data PHeap a = E | Node a [PHeap a]
```

10.4.2 Merge, insert, and top

There are two cases when merge two heaps:

1. Either heap is $\varnothing$, the result is the other heap;
2. Otherwise, compare the two roots, turn the greater one as the new sub-tree of the other.

$$\begin{array}{rcl} merge\ \varnothing\ h_2 & = & h_2 \\ merge\ h_1\ \varnothing & = & h_1 \\ merge\ (x, ts_1)\ (y, ts_2) & = & \begin{cases} x < y : & (x, (y, ts_2) : ts_1) \\ \text{otherwise} : & (y, (x, ts1) : ts_2) \end{cases} \end{array} \tag{10.24}$$

merge is bound to constant time. With the 'left-child, right sibling' method, we link the heap with greater root as the first sub-tree of the other.

```
function MERGE(H1, H2)
    if H1 = NIL then
        return H2
    if H2 = NIL then
        return H1
    if KEY(H2) < KEY(H1) then
        EXCHANGE(H1 ↔ H2)
    SUB-TREES(H1) ← LINK(H2, SUB-TREES(H1))
    PARENT(H2) ← H1
    return H1
```

Similar to Fibonacci heap, we implement insert with merge as (10.12). We access the top element from the root: $top\ (x, ts) = x$. Both operations are bound to constant time.

10.4.3 Increase priority

When decrease the value in a node, we cut the sub-tree rooted with this node, then merge it back to the heap. If the node is the root, we can directly decrease its value.

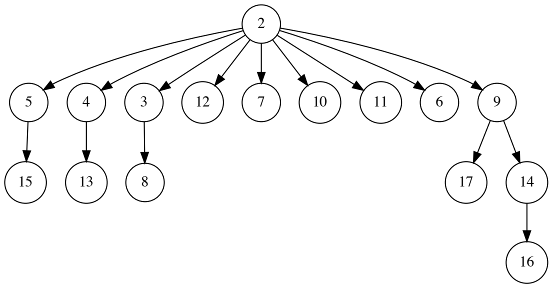
```
function DECREASE(H, x, k)
    KEY(x) ← k
    p ← PARENT(x)
    if p ≠ NIL then
        Remove x from SUB-TREES(p)
        PARENT(x) ← NIL
        return MERGE(H, x)
    return H
```

10.4.4 Pop

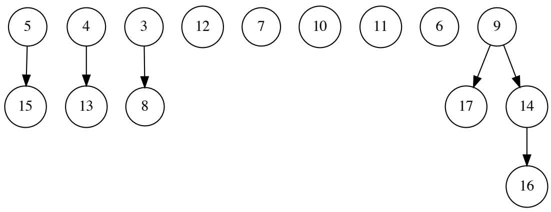
After pop the root, we consolidate the remaining sub-trees to a tree:

$$pop\ (x, ts) = consolidate\ ts \tag{10.25}$$

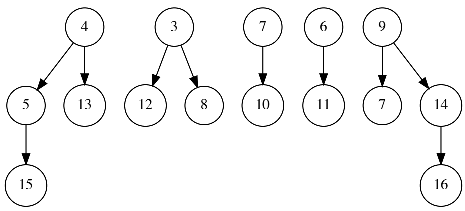
We firstly merge every two sub-trees from left to right, then merge these paired results from right to left to a tree. This explains the why we name it 'paring heap'. Figure 10.10 and 10.11 show the paired merge.



(a) Before pop.



(b) Pop 2, there are 9 sub-trees.



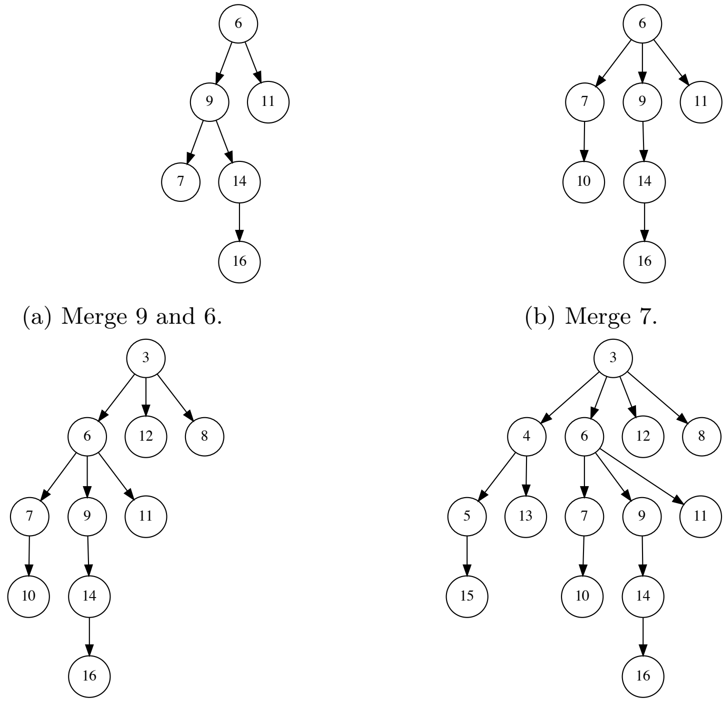
(c) Merge with pairs, leave the last tree.

Figure 10.10: Pop the root, merge sub-trees in pairs.

$$\begin{array}{rcl} consolidate\ [\,] & = & \varnothing \\ consolidate\ [t] & = & t \\ consolidate\ (t_1 : t_2 : ts) & = & merge\ (merge\ t_1\ t_2)\ (consolidate\ ts) \end{array} \tag{10.26}$$

The corresponding 'left child, right sibling' implementation is as below:

function POP(H)
    $L \leftarrow$ NIL
    for every T_x, T_y in SUB-TREES(H) **do**
        $T \leftarrow$ MERGE(T_x, T_y)
        $L \leftarrow$ LINK(T, L)
    $H \leftarrow$ NIL
    for T in L **do**



(a) Merge 9 and 6. (b) Merge 7.

(c) Merge 3. (d) Merge 4.

Figure 10.11: Merge from right to left.

        $H \leftarrow$ MERGE(H, T)
    return H

We iterate to merge T_x, T_y to T, and link ahead of L. When loop on L the second time, we actually traversed from right to left. When there are odd number of sub-trees, $T_y =$ NIL at last, hence $T = T_x$ in this case.

Delete

To delete a node x, we can first decrease the value in x to $-\infty$, then followed with a pop. There is an alternative method. If x is the root, we pop it; otherwise, we cut x off H, then apply pop to x, and merge x back to H:

function DELETE(H, x)
    if $H = x$ **then**
        POP(H)
    else
        $H \leftarrow$ CUT(H, x)
        $x \leftarrow$ POP(x)
        MERGE(H, x)

As delete is implemented with pop, the performance is conjectured to be amortized $O(\lg n)$ time.

Exercise 10.3

Implement delete for paring heap.

10.5 Summary

In this chapter, we extend the heap from binary tree based implementation to more data structures. Binomial heap and Fibonacci heap use forest of multi-way trees, pairing heap use a single multi-way tree.It's a common practice to post pone some expensive operation, and obtain better amortized performance.

10.6 Appendix - example programs

Definition of multi-way tree (left child, right sibling):

```
data Node<K> {
    Int rank
    K key
    Node<K> parent, subTrees, sibling,
    Bool mark

    Node(K x) {
        key = x
        rank = 0
        parent = subTrees = sibling = null
        mark = false
    }
}
```

Link binomial trees:

```
Node<K> link(Node<K> t1, Node<K> t2) {
    if t2.key < t1.key then (t1, t2) = (t2, t1)
    t2.sibling = t1.subTrees
    t1.subTrees = t2
    t2.parent = t1
    t1.rank = t1.rank + 1
    return t1
}
```

Binomial heap insert:

```
Node<K> insert(K x, Node<K> h) = insertTree(Node(x), h)

Node<K> insertTree(Node<K> t, Node<K> h) {
    var h1 = Node()
    var prev = h1
    while h ≠ null and h.rank ≤ t.rank {
        var t1 = h
        h = h.sibling
        if t.rank == t1.rank {
            t = link(t, t1)
        } else {
            prev.sibling = t1
            prev = t1
        }
    }
    prev.sibling = t
    t.sibling = h
    return removeFirst(h1)
}

Node<K> removeFirst(Node<K> h) {
    var next = h.sibling
    h.sibling = null
    return next
}
```

Binomial heap recursive insert:

```
data BiTree a = Node { rank :: Int
                     , key :: a
                     , subTrees :: [BiTree a]}

type BiHeap a = [BiTree a]

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)

insertTree t [] = [t]
insertTree t ts@(t':ts') | rank t < rank t' = t:ts
                         | rank t > rank t' = t' : insertTree t ts'
                         | otherwise = insertTree (link t t') ts'

insert x = insertTree (Node 0 x [])
```

Binomial heap merge:

```
Node<K> merge(h1, h2) {
    var h = Node()
    var prev = h
    while h1 ≠ null and h2 ≠ null {
        if h1.rank < h2.rank {
            prev.sibling = h1
            prev = prev.sibling
            h1 = h1.sibling
        } else if h2.rank < h1.rank {
            prev.sibling = h2
            prev = prev.sibling
            h2 = h2.sibling
        } else {
            var (t1, t2) = (h1, h2)
            (h1, h2) = (h1.sibling, h2.sibling)
            h1 = insertTree(link(t1, t2), h1)
    }
    if h1 ≠ null then prev.sibling = h1
    if h2 ≠ null then prev.sibling = h2
    return removeFirst(h)
}
```

Binomial heap recursive merge:

```
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 (link t1 t2) (merge ts1' ts2')
```

Binomial tree pop:

```
Node<K> reverse(Node<K> h) {
    Node<K> prev = null
    while h ≠ null {
        var x = h
        h = h.sibling
        x.sibling = prev
        prev = x
    }
    return prev
}

(Node<K>, Node<K>) extractMin(Node<K> h) {
```

```
    var head = h
    Node<K> tp = null
    Node<K> tm = null
    Node<K> prev = null
    while h ≠ null {
        if tm == null or h.key < tm.key {
            tm = h
            tp = prev
        }
        prev = h
        h = h.sibling
    }
    if tp ≠ null {
        tp.sibling = tm.sibling
    } else {
        head = tm.sibling
    }
    tm.sibling = null
    return (tm, head)
}

(K, Node<K>) pop(Node<K> h) {
    var (tm, h) = extractMin(h)
    h = merge(h, reverse(tm.subtrees))
    tm.subtrees = null
    return (tm.key, h)
}
```

Binomial heap recursive pop:

```
pop h = merge (reverse $ subTrees t) ts where
    (t, ts) = extractMin h

extractMin [t] = (t, [])
extractMin (t:ts) = if key t < key t' then (t, ts)
                    else (t', t:ts') where
    (t', ts') = extractMin ts
```

Merge Fibonacci heaps with bidirectional linked list:

```
FibHeap<K> merge(FibHeap<K> h1, FibHeap<K> h2) {
    if isEmpty(h1) then return h2
    if isEmpty(h2) then return h1
    FibHeap<K> h = FibHeap<K>()
    h.trees = concat(h1.trees, h2.trees)
    h.minTree = if h1.minTree.key < h2.minTree.key
                then h1.minTree else h2.minTree
    h.size = h1.size + h2.size
    return h
}

bool isEmpty(FibHeap<K> h) = (h == null or h.trees == null)

Node<K> concat(Node<K> first1, Node<K> first2) {
  var last1 = first1.prev
  var last2 = first2.prev
  last1.next = first2
  first2.prev = last1
  last2.next = first1
  first1.prev = last2
  return first1
}
```

Consolidate trees in Fibonacci heap:

```
consolidate = foldr melt [] where
```

```
    melt t [] = [t]
    meld t (t':ts) | rank t == rank t' = meld (link t t') ts
                   | rank t <  rank t' = t : t' : ts
                   | otherwise = t' : meld t ts
```

Consolidate trees with auxiliary array:

```
void consolidate(FibHeap<K> h) {
    Int R = maxRank(h.size) + 1
    Node<K>[R] a = [null, ...]
    while h.trees ≠ null {
        var x = h.trees
        h.trees = remove(h.trees, x)
        Int r = x.rank
        while a[r] ≠ null {
            var y = a[r]
            x = link(x, y)
            a[r] = null
            r = r + 1
        }
        a[r] = x
    }
    h.minTr = null
    h.trees = null
    for var t in a if t ≠ null {
        h.trees = append(h.trees, t)
        if h.minTr == null or t.key < h.minTr.key then h.minTr = t
    }
}
```

Fibonacci heap pop:

```
pop (FH _ (Node _ x []) []) = (x, E)
pop (FH sz (Node _ x tsm) ts) = (x, FH (sz - 1) tm ts') where
    (tm, ts') = extractMin $ consolidate (tsm ++ ts)
```

Decrease value in Fibonacci heap:

```
void decrease(FibHeap<K> h, Node<K> x, K k) {
    var p = x.parent
    x.key = k
    if p ≠ null and k < p.key {
        cut(h, x)
        cascadeCut(h, p)
    }
    if k < h.minTr.key then h.minTr = x
}

void cut(FibHeap<K> h, Node<K> x) {
    var p = x.parent
    p.subTrees = remove(p.subTrees, x)
    p.rank = p.rank - 1
    h.trees = append(h.trees, x)
    x.parent = null
    x.mark = false
}

void cascadeCut(FibHeap<K> h, Node<K> x) {
    var p = x.parent
    if p == null then return
    if x.mark {
        cut(h, x)
        cascadeCut(h, p)
    } else {
        x.mark = true
    }
```

```
}
```

Chapter 11

Queue

11.1 Introduction

Queue supports first-in, first-out (FIFO). There are many ways to implement queue, e.g., through linked list, doubly liked list, circular buffer, etc. Okasaki gave 16 different implementations in [3]. A queue satisfies the following two requirements:

1. Add a new element to the tail in constant time;

2. Access or remove an element from head in constant time.

It's easy to realize queue with doubly linked list. We skip this implementation, and focus on using other basic data structures, like (singly) linked list or array.

11.2 Linked-list queue

We can insert or remove element from the head of a linked list. However, to support FIFO, we have to do one operation in head, and the other in tail. We need $O(n)$ time traverse to reach the tail, where n is the length. To achieve the constant time performance goal, we use an extra variable to record the tail position, and apply a sentinel node S to simplify the empty queue case handling, as shown in figure 11.1.

```
data Node<K> {
  Key key
  Node next
}

data Queue {
  Node head, tail
}
```

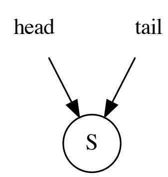


Figure 11.1: Both head and tail point to S for empty queue.

The two important queue operations are 'enqueue' (also called push, snoc, append, or push back) and 'dequeue' (also called pop, or pop front). When implement queue with list, we push on head, and pop from tail.

function ENQUEUE(Q, x)
    $p \leftarrow$ NODE(x)
    NEXT(p) $\leftarrow$ NIL
    NEXT(TAIL(Q)) $\leftarrow p$
    TAIL(Q) $\leftarrow p$

As there is at least a S node even for empty queue, we need not check if the tail is NIL.

function DEQUEUE(Q)
    $x \leftarrow$ HEAD(Q)
    NEXT(HEAD(Q)) $\leftarrow$ NEXT(x)
    if $x =$ TAIL(Q) **then** ▷ Q is empty
        TAIL(Q) $\leftarrow$ HEAD(Q)
    return KEY(x)

As the S node is ahead of all other nodes, HEAD actually returns the next node to S, as shown in figure 11.2. It's easy to expand this implementation to concurrent environment with two locks on the head and tail respectively. S node helps to prevent dead-lock when the queue is empty[59] [60].

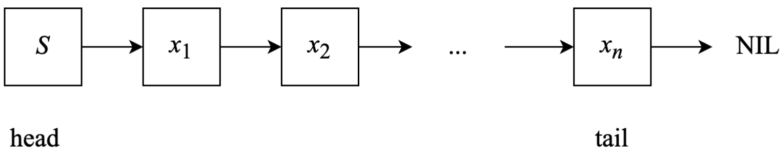


Figure 11.2: List with S node.

11.3 Circular buffer

Symmetrically, we can append element to the tail of array, but it takes linear time $O(n)$ to remove element from head. This is because we need shift all 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, as shown figure 11.4, and 11.3. We can use the head index, the length count, and the size of the array to define a queue. It's empty when the count is 0, it's full when count = size, we can also simplify the enqueue/dequeue implementation with modular operation.

function ENQUEUE(Q, x)
    if not FULL(Q) **then**
        COUNT(Q) $\leftarrow$ COUNT(Q) + 1
        tail $\leftarrow$ (HEAD(Q) + COUNT(Q)) mod SIZE(Q)
        BUF(Q)[tail] $\leftarrow x$

function DEQUEUE(Q)
    $x \leftarrow$ NIL
    if not EMPTY(Q) **then**
        $h \leftarrow$ HEAD(Q)
        $x \leftarrow$ BUF(Q)[h]
        HEAD(Q) $\leftarrow$ (h + 1) mod SIZE(Q)

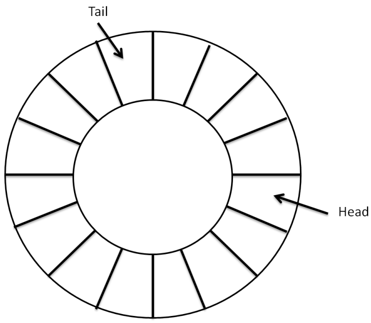
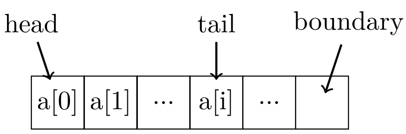
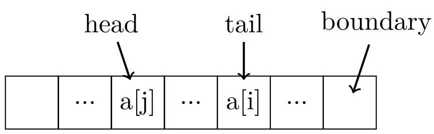


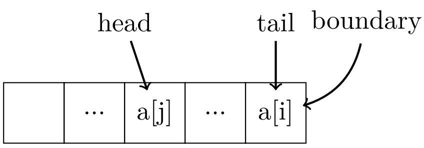
Figure 11.3: Circular buffer.



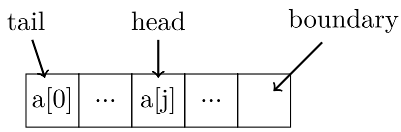
(a) Enqueue some elements.



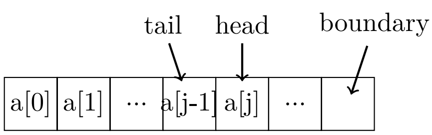
(b) Free cells after dequeue.



(c) Enqueue more elements to the boundary



(d) Enqueue the next element to the first cell.



(e) All cells are occupied, full.

Figure 11.4: Circular buffer queue

        COUNT(Q) $\leftarrow$ COUNT(Q) - 1
    return x

Exercise 11.1

The circular buffer is allocated with a predefined size. We can use two references head and tail instead of count. How to determine if a circular buffer queue is full or empty? (the head can be either ahead of tail or behind it.)

11.4 Paired-list queue

We can access list head in constant time, but need linear time to access the tail. We can connect two lists 'tail to tail' to implement queue, as shown in figure 11.5. We define such queue as (f, r), where f is the front list, and r is the rear list. The empty list is $([\,], [\,])$. We push new element to the head of r, and pop from the tail of f. Both are constant time.

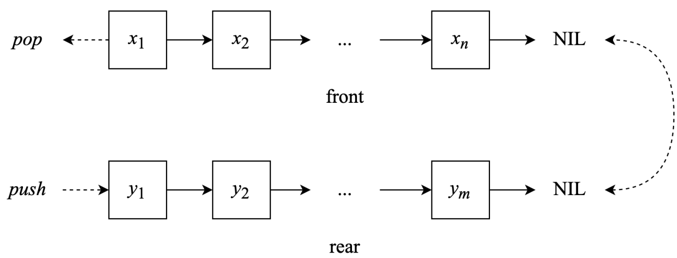


Figure 11.5: paired-list queue.

$$\begin{cases} push\ x\ (f, r) & = (f, x{:}r) \\ pop\ (x{:}f, r) & = (f, r) \end{cases} \tag{11.1}$$

f may become empty after a series of pops, while r still contains elements. To continue pop, we reverse r to replace f, i.e., $([\,], r) \mapsto (reverse\ r, [\,])$. We need check and adjust balance after every push/pop:

$$\begin{array}{rcl} balance\ [\,]\ r & = & (reverse\ r, [\,]) \\ balance\ f\ r & = & (f, r) \end{array} \tag{11.2}$$

Although the time is bound to linear time when reverse r, the amortised performance is constant time. We adjust the push/pop as below:

$$\begin{cases} push\ x\ (f, r) & = balance\ f\ (x{:}r) \\ pop\ (x{:}f, r) & = balance\ f\ r \end{cases} \tag{11.3}$$

There is a symmetric implementation with a pair of arrays. Table 11.1 shows the symmetric between list and array. We connect two arrays head to head to form a queue, as shown in figure 11.6. When array R becomes empty, we reverse array F to replace R.

Exercise 11.2

1. Why need balance check and adjustment after push?
2. Prove the amortized performance of paired-list queue is constant time.
3. Implement the paired-array queue.

operation	array	list
insert to head	$O(n)$	$O(1)$
append to tail	$O(1)$	$O(n)$
remove from head	$O(n)$	$O(1)$
remove from tail	$O(1)$	$O(n)$

Table 11.1: array and list

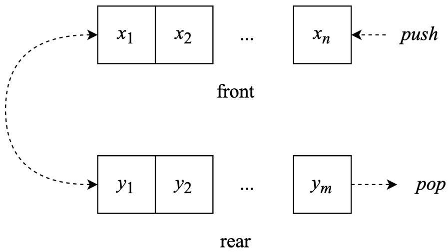


Figure 11.6: paired-array queue.

11.5 Balance Queue

Although paired-list queue performs in amortized constant time, it is linear time in worse case. For e.g., there is an element in f, then repeat pushing n elements. Now it takes $O(n)$ time to pop. The lengths of f and r are unbalance in this case. To solve it, we add another rule: keep the length of r is not greater than f, otherwise we reverse.

$$|r| \leq |f| \tag{11.4}$$

We check the lengths in every push/pop, however, it takes linear time to compute length. We can record the length in a variable, and update it during push/pop. Denote the paired-list queue as (f, n, r, m), where $n = |f|$, $m = |r|$. From the balance rule (11.4), we can check the length of f to test if a queue is empty:

$$Q = \phi \iff n = 0 \tag{11.5}$$

The definition of push/pop change to:

$$\begin{cases} push\ x\ (f, n, r, m) & = balance\ (f, n, x{:}r, m+1) \\ pop\ (x{:}f, n, r, m) & = balance\ (f, n-1, r, m) \end{cases} \tag{11.6}$$

Where *balance* is defined as:

$$balance\ (f, n, r, m) = \begin{cases} m \leq n: & (f, n, r, m) \\ \text{otherwise}: & (f \mathbin{+\!\!+} reverse\ r, m+n, [\ ], 0) \end{cases} \tag{11.7}$$

11.6 Real-time queue

It still takes linear time to reverse, concatenate lists in balanced queue. A real-time queue need guarantee constant time in every push/pop operation. The performance bottleneck happens in $f \mathbin{+\!\!+} reverse\ r$. At this time, $m >$, breaks the balance rule. Since m, n are integers, we know $m = n + 1$. $\mathbin{+\!\!+}$ takes $O(n)$ time, and reverse takes $O(m)$ time. The total time is bound to $O(n + m)$, which is proportion to the number of elements. Our solution is to distribute this computation to multiple push and pop operations. Let's revisit the tail recursive[61] [62] reverse:

$$reverse = reverse'\ [\ ] \tag{11.8}$$

This is in Curried form, where:

$$\begin{array}{rcl} reverse'\ a\ [\ ] &=& a \\ reverse'\ a\ (x{:}xs) &=& reverse'\ (x{:}a)\ xs \end{array} \tag{11.9}$$

We can turn the tail recursive implementation to stepped computation. We model it as a series of state transformation. Define a state machine with two states: reverse state S_r, and complete state S_f. We *slow-down* the reverse computation as below:

$$\begin{array}{rcl} step\ S_r\ a\ [\ ] &=& (S_f, a) \\ step\ S_r\ a\ (x{:}xs) &=& (S_r, (x{:}a), xs) \end{array} \tag{11.10}$$

Each step, we check and transform the state. S_r means the reverse is on going. If there is no remaining element to reverse, we change the state to S_f (done); otherwise, we pick the head element x, link it ahead of a. This step terminates, but not continues to recursion. The new state with the intermediate reverse result will be input to the next *step*. For example:

$$\begin{array}{rcl} step\ S_r\ \text{"hello"}\ [\ ] &=& (S_r, \text{"ello"}, \text{"h"}) \\ step\ S_r\ \text{"ello"}\ \text{"h"} &=& (S_r, \text{"llo"}, \text{"eh"}) \\ & ... & \\ step\ S_r\ \text{"o"}\ \text{"lleh"} &=& (S_r, [\ ], \text{"olleh"}) \\ step\ S_r\ [\ ]\ \text{"olleh"} &=& (S_f, \text{"olleh"}) \end{array}$$

We can next distribute the reverse steps to push/pop operations. However, it only solves half problem. We next need slow-down $+\!\!+$ computation, which is more complex. We use state machine too. To concatenate $xs +\!\!+ ys$, we first reverse xs to $\overleftarrow{xs}$, then pick elements form $\overleftarrow{xs}$ one by one, and link each head of ys. The idea is similar to *reverse′*:

$$\begin{array}{rcl} xs +\!\!+ ys &=& (reverse\ reverse\ xs) +\!\!+ ys \\ &=& (reverse'\ [\ ]\ (reverse\ xs)) +\!\!+ ys \\ &=& reverse'\ ys\ (reverse\ xs) \\ &=& reverse'\ ys\ \overleftarrow{xs} \end{array} \tag{11.11}$$

We need add another state. After reverse r, we step by step concatenate from $\overleftarrow{f}$. The three states are: S_r of reverse, S_c of concatenate, S_f of completion. The two phases are:

1. Reverse f and r in parallel to: $\overleftarrow{f}$ and $\overleftarrow{r}$ step by step;
2. Stepped taking elements from $\overleftarrow{f}$, and link each ahead of $\overleftarrow{r}$.

$$\begin{array}{rcll} next\ (S_r, f', x{:}f, r', y{:}r) &=& (S_r, x{:}f', f, y{:}r', r) & \text{reverse } f, r \\ next\ (S_r, f', [\ ], r', [y]) &=& next\ (S_c, f', y{:}r') & \text{reverse done, start concatenation} \\ next\ (S_c, a, [\ ]) &=& (S_f, a) & \text{done} \\ next\ (S_c, a, x{:}f') &=& (S_c, x{:}a, f') & \text{concatenation} \end{array} \tag{11.12}$$

We need arrange these steps to each push/pop next. From the balance rule, when $m = n + 1$, we kick off $f +\!\!+ reverse\ r$. it takes $n + 1$ steps to reverse r, within these steps, we reverse f in parallel. After that, we use another $n + 1$ steps to concatenate.

$2n + 2$ steps in total. The critical question is: Before we complete the $2n + 2$ steps, will the queue become unbalanced due to a series of push/pop operations?

Luckily, repeat pushing won't break the balance rule again before we complete $f \mathbin{+\!\!+} reverse\ r$ in $2n+2$ steps. We will obtain a new front list $f' = f \mathbin{+\!\!+} reverse\ r$ after $2n+2$ steps, while the time to break the balance rule again is:

$$\begin{array}{rcl} |r'| & = & |f'| + 1 \\ & = & |f| + |r| + 1 \\ & = & 2n + 2 \end{array} \tag{11.13}$$

Thanks to the balance rule. It means even repeat pushing as many elements as possible, from the previous to the next time when the queue is unbalanced, the $2n + 2$ steps are guaranteed to be completed, hence the new f is ready. We can next safely start to compute $f' \mathbin{+\!\!+} reverse\ r'$.

However, pop may happen before the completion of $2n + 2$ steps. We are facing the situation that needs extract element from f, while the new front list $f' = f \mathbin{+\!\!+} reverse\ r$ hasn't been ready yet. To solve this issue, we duplicate a copy of f when doing $reverse\ f$. We are save even repeat pop for n times. Table 11.2 shows the queue during phase 1 (reverse f and r in parallel)[1].

f copy	on-going part	new r
$\{f_i, f_{i+1}, ..., f_n\}$	$(S_r, \tilde{f}, ..., \tilde{r}, ...)$	$\{...\}$
first $i - 1$ elements out	intermediate $\overleftarrow{f}$, $\overleftarrow{r}$	newly pushed

Table 11.2: Before completion of the first n steps.

The copy of f is exhausted after repeated n pops. We are about to stepped concatenation. What if pop happens at this time? Since f is exhausted, it becomes [ ]. We needn't concatenate anymore. This is because $f \mathbin{+\!\!+} \overleftarrow{r} = [\,] \mathbin{+\!\!+} \overleftarrow{r} = \overleftarrow{r}$. In fact, we only need to concatenate the elements in f that haven't been popped. Because we pop elements from the head of f, we use a counter to record the remaining elements in f. It's initialized as 0. We apply +1 every time when reverse an element in f. It means we need concatenate this element in the future; Whenever pop happens, we apply -1, means we needn't concatenate this one any more. We also decrease it during concatenation process, and cancel the process when it is 0. Below is the updated state transformation:

$$\begin{array}{rcll} next\ (S_r, n, f', x{:}f, r', y{:}r) & = & (S_r, n+1, x{:}f', f, y{:}r', r) & \text{reverse } f, r \\ next\ (S_r, n, f', [\,], r', [y]) & = & next\ (S_c, n, f', y{:}r') & \text{reverse done, start concatenation} \\ next\ (S_c, 0, a, f) & = & (S_f, a) & \text{done} \\ next\ (S_c, n, a, x{:}f') & = & (S_c, n-1, x{:}a, f') & \text{concatenation} \\ next\ S_0 & = & S_0 & \text{idle} \end{array} \tag{11.14}$$

We define addition idle state S_0 to simplify the transition logic. The queue contains 3 parts: the front list f with its length n, the state S of on going $f \mathbin{+\!\!+} reverse\ r$, and the rear list r with its length m. Denoted as (f, n, S, r, m). The empty queue is $([\,], 0, S_0, [\,], 0)$. We can tell a queue is empty when $n = 0$ according to the balance rule. The push/pop are updated as:

$$\begin{cases} push\ x\ (f, n, S, r, m) & = balance\ f\ n\ S\ (x{:}r)\ (m+1) \\ pop\ (x{:}f, n, S, r, m) & = balance\ f\ (n-1)\ (abort\ S)\ r\ m \end{cases} \tag{11.15}$$

[1]Although it takes linear time to duplicate a list, however, the one time copying won't happen at all. We actually duplicate the reference to the front list, and delay the element level copying to each step

Where *abort* decrease the counter in pop to cancel an element for concatenation. We'll define it later. *balance* triggers stepped $f \mathbin{+\!\!+} reverse\ r$ if the queue is unbalanced, else runs a step:

$$balance\ f\ n\ S\ r\ m = \begin{cases} m \leq n : & step\ f\ n\ S\ r\ m \\ \text{otherwise} : & step\ f\ (n+m)\ (next\ (S_r, 0, [\,], f, [\,], r))\ [\,]\ 0 \end{cases} \tag{11.16}$$

Where *step* transforms the state machine to next state. It ends with the idle state S_0 when completes.

$$step\ f\ n\ S\ r\ m = queue\ (next\ S) \tag{11.17}$$

Where:

$$\begin{array}{rcll} queue\ (S_f, f') & = & (f', n, S_0, r, m) & \text{replace } f \text{ with } f' \\ queue\ S' & = & (f, n, S', r, m) & \end{array} \tag{11.18}$$

We define *abort* to cancel an element:

$$\begin{array}{rcl} abort\ (S_c, 0, (x{:}a), f') & = & (S_f, a) \\ abort\ (S_c, n, a, f') & = & (S_c, n-1, a, f') \\ abort\ (S_r, n, f'f, r'r) & = & (S_r, n-1, f', f, r', r) \\ abort\ S & = & S \end{array} \tag{11.19}$$

Exercise 11.3

1. Why need rollback an element (we cancelled the previous 'cons', removed x and return a as the result) when $n = 0$ in *abort*?

11.7 Lazy real-time queue

The key to realize real-time queue is to break down the expensive $f \mathbin{+\!\!+} reverse\ r$. We can simplify it with lazy evaluation. Assume function *rotate* compute $f \mathbin{+\!\!+} reverse\ r$ in steps, i.e., below two functions are equivalent with an accumulator a.

$$rotate\ xs\ ys\ a = xs \mathbin{+\!\!+} (reverse\ ys) \mathbin{+\!\!+} a \tag{11.20}$$

We initialize xs as the front list f, ys as the rear list r, the accumulator a empty $[\,]$. We implement *rotate* from the edge case:

$$rotate\ [\,]\ [y]\ a = y{:}a \tag{11.21}$$

The recursive case is:

$$\begin{array}{rll} & rotate\ (x{:}xs)\ (y{:}ys)\ a & \\ = & (x{:}xs) \mathbin{+\!\!+} (reverse\ (y{:}ys)) \mathbin{+\!\!+} a & \text{from (11.20)} \\ = & x : (xs \mathbin{+\!\!+} reverse\ (y{:}ys)) \mathbin{+\!\!+} a) & \text{concatenation is associative} \\ = & x : (xs \mathbin{+\!\!+} reverse\ ys \mathbin{+\!\!+} (y{:}a)) & \text{reverse property, and associative} \\ = & x : rotate\ xs\ ys\ (y{:}a) & \text{reverse of (11.20)} \end{array} \tag{11.22}$$

Summarize them together:

$$\begin{array}{rcl} rotate\ [\,]\ [y]\ a & = & y{:}a \\ rotate\ (x{:}xs)\ (y{:}ys)\ a & = & x : rotate\ xs\ ys\ (y{:}a) \end{array} \tag{11.23}$$

In lazy evaluation settings, (:) is delayed to push/pop, hence the *rotate* is broken down. We change the paired-list queue definition to (f, r, rot), where rot is the on going $f \mathbin{+\!\!+} reverse\ r$ computation. It is initialized empty [ ].

$$\begin{cases} push\ x\ (f, r, rot) & = balance\ f\ (x{:}r)\ rot \\ pop\ (x{:}f, r, rot) & = balance\ f\ r\ rot \end{cases} \tag{11.24}$$

Every time, *balance* advances the rotation one step, and starts another round when completes.

$$\begin{array}{rcll} balance\ f\ r\ [\,] & = & (f', [\,], f') & 其中 : f' = rotate\ f\ r\ [\,] \\ balance\ f\ r\ (x{:}rot) & = & (f, r, rot) & 推进轮转 \end{array} \tag{11.25}$$

Exercise 11.4

Implement bidirectional queue, support add/remove elements on both head and tail in constant time.

11.8 Appendix - example programs

List implemented queue:

```
Queue<K> enQ(Queue<K> q, K x) {
    var p = Node(x)
    p.next = null
    q.tail.next = p
    q.tail = p
    return q
}

K deQ(Queue<K> q) {
    var p = q.head.next   //the next of S
    q.head.next = p.next
    if q.tail == p then q.tail = q.head //empty
    return p.key
}
```

Circular buffer queue:

```
data Queue<K> {
    [K] buf
    int head, cnt, size

    Queue(int max) {
        buf = Array<K>(max)
        size = max
        head = cnt = 0
    }
}
```

Enqueue, dequeue implementation for circular buffer queue:

```
N offset(N i, N size) = if i < size then i else i - size

void enQ(Queue<K> q, K x) {
    if q.cnt < q.size {
        q.buf[offset(q.head + q.cnt, q.size)] = x;
        q.cnt = q.cnt + 1
    }
}
```

```
K head(Queue<K> q) = if q.cnt == 0 then null else q.buf[q.head]

K deQ(Queue<K> q) {
    K x = null
    if q.cnt > 0 {
        x = head(q)
        q.head = offset(q→head + 1, q→size);
        q.cnt = q.cnt -1
    }
    return x
}
```

Real-time queue:

```
data State a = Empty
             | Reverse Int [a] [a] [a] [a] -- n, acc f, f, acc r, r
             | Concat Int [a] [a]          -- n, acc, reversed f
             | Done [a]  -- f' = f ++ reverse r

-- f, n = length f, state, r, m = length r
data RealtimeQueue a = RTQ [a] Int (State a) [a] Int

push x (RTQ f n s r m) = balance f n s (x:r) (m + 1)

pop (RTQ (_:f) n s r m) = balance f (n - 1) (abort s) r m

top (RTQ (x:_) _ _ _ _) = x

balance f n s r m
    | m ≤ n =  step f n s r m
    | otherwise = step f (m + n) (next (Reverse 0 [] f [] r)) [] 0

step f n s r m = queue (next s) where
  queue (Done f') = RTQ f' n Empty r m
  queue s' = RTQ f n s' r m

next (Reverse n f' (x:f) r' (y:r)) = Reverse (n + 1) (x:f') f (y:r') r
next (Reverse n f' [] r' [y]) = next $ Concat n (y:r') f'
next (Concat 0 acc _) = Done acc
next (Concat n acc (x:f')) = Concat (n-1) (x:acc) f'
next s = s

abort (Concat 0 (_:acc) _) = Done acc -- rollback 1 elem
abort (Concat n acc f') = Concat (n - 1) acc f'
abort (Reverse n f' f r' r) = Reverse (n - 1) f' f r' r
abort s = s
```

Lazy real-time queue:

```
data LazyRTQueue a = LQ [a] [a] [a] -- front, rear, f ++ reverse r

empty = LQ [] [] []

push (LQ f r rot) x = balance f (x:r) rot

pop (LQ (_:f) r rot) = balance f r rot

top (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)
```

Chapter 12

Sequence

12.1 Introduction

Sequence is a combination of array and list. We set the following goals for the ideal sequence:

1. Add, remove element on head and tail in constant time;
2. Fast (no slower than linear time) concatenate two sequences;
3. Fast access, update element at any position;
4. Fast split at any position;

Array and list only satisfy these goals partially as shown in below table. Where n is the length for the sequence. If there are two sequences, then we use n_1, n_2 for their lengths respectively.

operation	array	list
add/remove on head	$O(n)$	$O(1)$
add/remove on tail	$O(1)$	$O(n)$
concatenate	$O(n_2)$	$O(n_1)$
random access at i	$O(1)$	$O(i)$
remove at i	$O(n-i)$	$O(i)$

We give three implementations: binary random access list, concatenate-able list, and finger tree.

12.2 Binary random access list

The binary random access list is a set of full binary trees (forest). The elements are stored in leaves. For any integer $n \geq 0$, we know how many trees need to hold n elements from its binary format. Every bit of 1 represents a binary tree, the tree size is determined by the magnitude of the bit. For any index $1 \leq i \leq n$, we can locate the binary tree that stores the i-th element. As shown in figure 12.1, tree t_1, t_2 represent sequence $[x_1, x_2, x_3, x_4, x_5, x_6]$.

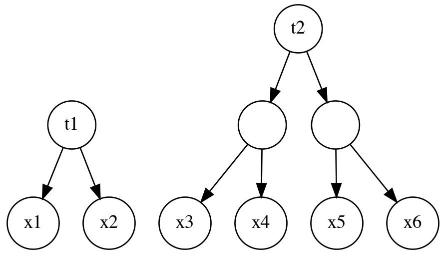


Figure 12.1: A sequence of 6 elements.

Denote the full binary tree of depth $i + 1$ as t_i. t_0 only has a leaf node. There are 2^i leaves in t_i. For sequence of n elements, represent n in binary as $n = (e_m e_{m-1} ... e_1 e_0)_2$, where e_i is either 1 or 0.

$$n = 2^0 e_0 + 2^1 e_1 + ... + 2^m e_m \tag{12.1}$$

If $e_i \neq 0$, there is a full binary tree t_i of size 2^i. For example in figure 12.1, the length of the sequence is $6 = (110)_2$. The lowest bit is 0, there's no tree of size 1; the 2nd bit is 1, there is t_1 of size 2; the highest bit is 1, there is t_2 of size 4. In this way, we represent sequence $[x_1, x_2, ..., x_n]$ as a list of trees. Each tree has unique size, in ascending order. We call it *binary random access list*[3]. We can customize the binary tree definition: (1) only store the element in leaf node as (x); (2) augment the size in each branch node as (s, l, r), where s is the size of the tree, l, r are left and right tree respectively. We get the size information as below:

$$\begin{array}{rcl} size\ (x) & = & 1 \\ size\ (s, l, r) & = & s \end{array} \tag{12.2}$$

To add a new element y before sequence S, we create a singleton t_0 tree $t' = (y)$, then insert it to the forest. $insert\ y\ S = insert_T\ (y)\ S$, or define it in Curried form:

$$insert\ y = insert_T\ (y) \tag{12.3}$$

We compare t' with the first tree t_i in the forest, if t_i is bigger, then put t' ahead of the forest (in constant time); if they have the same size, then link them to a bigger tree (in constant time): $t'_{i+1} = (2s, t_i, t')$, then recursively insert t'_{i+1} to the forest, as shown in figure 12.2.

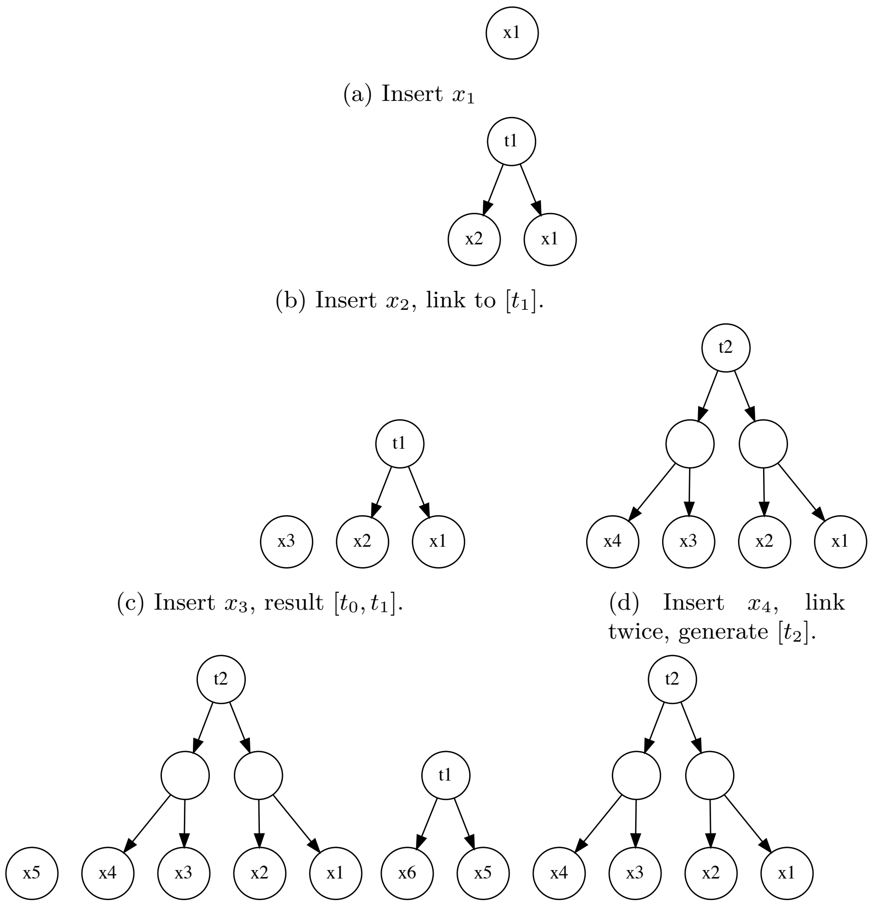
$$\begin{array}{rcl} insert_T\ t\ [\,] & = & [t] \\ insert_T\ t\ (t_1 \colon ts) & = & \begin{cases} size\ t < size\ t_1 : & t : t_1 : ts \\ \text{otherwise} : & insert_T\ (link\ t\ t_1)\ ts \end{cases} \end{array} \tag{12.4}$$

Where $link$ links two trees of the same size: $link\ t_1\ t_2 = (size\ t_1 + size\ t_2, t_1, t_2)$.

For n elements, there are $m = O(\lg n)$ trees in the forest. The performance is bound to $O(\lg n)$ time. We'll prove the amortized performance is constant time.

Symmetrically, we can reverse the insert process to define remove. If the first tree is t_0 (singleton leaf), we remove t_0; otherwise, we repeat splitting the first tree to obtain a t_0 and remove it, as shown in figure 12.3.

$$\begin{array}{rcl} extract\ ((x) \colon ts) & = & (x, ts) \\ extract\ ((s, t_1, t_2) \colon ts) & = & extract\ (t_1 \colon t_2 \colon ts) \end{array} \tag{12.5}$$



(a) Insert x_1

(b) Insert x_2, link to $[t_1]$.

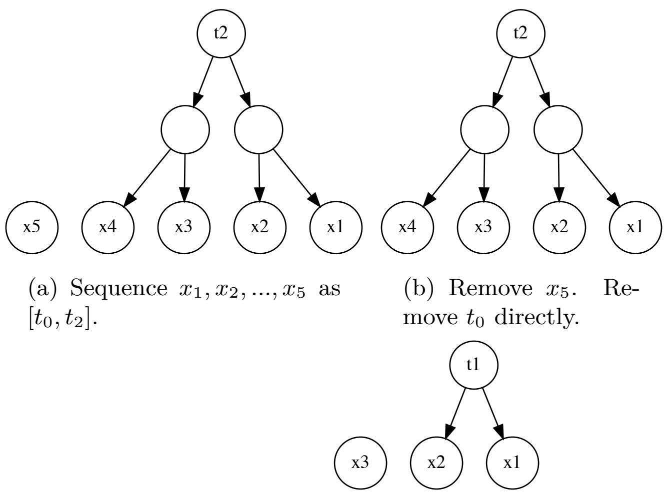
(c) Insert x_3, result $[t_0, t_1]$.

(d) Insert x_4, link twice, generate $[t_2]$.

(e) Insert x_5, result $[t_0, t_2]$.

(f) Insert x_6, result $[t_1, t_2]$.

Figure 12.2: Insert $x_1, x_2, ..., x_6$.



(a) Sequence $x_1, x_2, ..., x_5$ as $[t_0, t_2]$.

(b) Remove x_5. Remove t_0 directly.

(c) Remove x_4. Split twice to get $[t_0, t_0, t_1]$, then remove the head to get $[t_0, t_1]$.

Figure 12.3: Remove

We call *extract* to remove element from head:

$$\begin{cases} head & = fst \circ extract \\ tail & = snd \circ extract \end{cases} \tag{12.6}$$

Where $fst\ (a, b) = a$, $snd\ (a, b) = b$ access the component in a pair.

The trees divides elements into chunks. For a given index $1 \leq i \leq n$, we first locate the corresponding tree, then lookup the tree to access the element.

1. For the first tree t in the forest, if $i \leq size(t)$, then the element is in t, we next lookup t for the target element;

2. Otherwise, let $i' = i - size(t)$, then recursively lookup the i'-th element in the rest trees.

$$(t{:}ts)[i] = \begin{cases} i \leq size\ t: & lookup_T\ i\ t \\ \text{otherwise}: & ts[i - size\ t] \end{cases} \tag{12.7}$$

Where $lookup_T$ applies binary search. If $i = 1$, returns the root, else divides the tree and recursively lookup:

$$\begin{array}{rcl} lookup_T\ 1\ (x) & = & x \\ lookup_T\ i\ (s, t_1, t_2) & = & \begin{cases} i \leq \lfloor \frac{s}{2} \rfloor: & lookup_T\ i\ t_1 \\ \text{otherwise}: & lookup_T\ (i - \lfloor \frac{s}{2} \rfloor)\ t_2 \end{cases} \end{array} \tag{12.8}$$

Figure 12.4 gives the steps to lookup the 4-th element in a sequence of length 6. The size of the first tree is $2 < 4$, move to the next tree and update the index to $i' = 4 - 2$. The size of the second tree is $4 > i' = 2$, we need lookup it. Because the index 2 is less than the half size $4/2 = 2$, we lookup the left, then the right, and finally locate the element. Similarly, we can alter an element at a given position.

There are $O(\lg n)$ full binary trees to hold n elements. For index i, we need at most $O(\lg n)$ time to locate the tree, the next lookup time is proportion to the height, which is $O(\lg n)$ at most. The overall random access time is bound to $O(\lg n)$.

Exercise 12.1

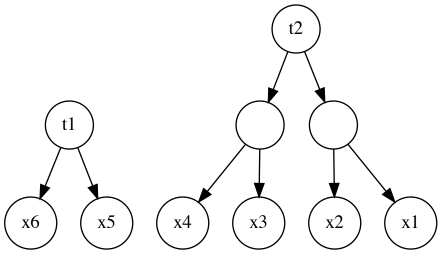
How to handle the out of bound exception?

12.3 Numeric representation

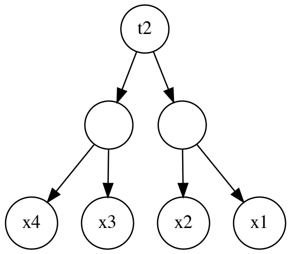
The binary form of $n = 2^0 e_0 + 2^1 e_1 + ... + 2^m e_m$ maps to the forest. The e_i is the i-th bit. If $e_i = 1$, there is a full binary tree of size 2^i. Adding an element corresponds to +1 to a binary number; while deleting corresponds to -1. We call such correspondence *numeric representation*[3]. To explicitly express this correspondence, we define two states: *Zero* means none existence of the binary tree, while *One t* means there exits tree t. As such, we represent the forest as a list of binary states, and implement insert as binary add.

$$\begin{array}{rcl} add\ t\ [\,] & = & [One\ t] \\ add\ t\ (Zero{:}ds) & = & (One\ t) : ds \\ add\ t\ (One\ t'{:}ds) & = & Zero : add\ (link\ t\ t')\ ds \end{array} \tag{12.9}$$

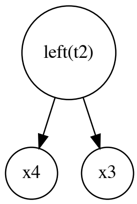
When add tree t, if the forest is empty, we create a state of *One t*, it's the only bit, corresponding to $0 + 1 = 1$. If the forest isn't empty, and the first bit is *Zero*, we



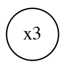
(a) $S[4], 4 > size(t_1) = 2$



(b) $S'[4-2] \Rightarrow lookup_T\ 2\ t_2$



(c) $2 \leq \lfloor \frac{size(t_2)}{2} \rfloor \Rightarrow lookup_T\ 2\ left(t_2)$



(d) $lookup_T\ 1\ right(left(t_2))$, return x_3

Figure 12.4: Steps to access $S[4]$

use the state *One t* to replace *Zero*, corresponding to binary add $(...digits...0)_2 + 1 = (...digits...1)_2$. For e.g. $6+1 = (110)_2+1 = (111)_2 = 7$. If the first bit is *One t′*, we assume t and t' have the same size because we always start to insert from a singleton leaf $t_0 = (x)$. The tree size increase as a sequence of $1, 2, 4, ..., 2^i, ...$. We link t and t', recursively insert to the rest bits. The original *One t′* is replaced by *Zero*. It corresponds to binary add $(...digits...1)_2 + 1 = (...digits'...0)_2$. For e.g. $7 + 1 = (111)_2 + 1 = (1000)_2 = 8$.

Symmetrically, we can implement remove as binary subtraction. If the sequence is a singleton bit *One t*, it becomes empty after remove, corresponding to $1 - 1 = 0$. If there are multiple bits and the first one is *One t*, we replace it by *Zero*. This corresponds to $(...digits...1)_2 - 1 = (...digits...0)_2$. For e.g., $7 - 1 = (111)_2 - 1 = (110)_2 = 6$. If the first bit is *Zero*, we need borrow. We cursively extract tree from the rest bits, split into two t_1, t_2, replace *Zero* to *One* t_2, and remove t_1. It corresponds to $(...digits...0)_2 - 1 = (...digits'...1)_2$. For e.g., $4 - 1 = (100)_2 - 1 = (11)_2 = 3$.

$$\begin{array}{rcl} minus\ [One\ t] & = & (t, [\,]) \\ minus\ ((One\ t){:}ts) & = & (t, Zero{:}ts) \\ minus\ (Zero{:}ts) & = & (t_1, (One\ t_2){:}ts'), \text{where} : (s, t_1, t_2) = minus\ ts \end{array} \tag{12.10}$$

Numeric representation doesn't change the performance. We next evaluate the amortized time by aggregation. The steps to insert $n = 2^m$ elements to empty is given as table 12.1:

i	binary (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

Table 12.1: Insert 2^m elements.

The LSB changes every time when insert, total 2^m times. The second bit changes every other time (link trees), total 2^{m-1} times. The second highest bit only changes 1 time, links all trees to a final one. The highest bit changes to 1 after insert the last element. Sum all times: $T = 1+1+2+4+...+2^{m-1}+2^m = 2^{m+1}$. Hence the amortized performance is:

$$O(T/n) = O(\frac{2^{m+1}}{2^m}) = O(1) \tag{12.11}$$

Proved the amortized constant time performance.

Exercise 12.2

1. Implement the random access for numeric representation $S[i], 1 \leq i \leq n$, where n is the length of the sequence.
2. Prove the amortized performance of delete is constant time. (hint: use aggregation method).
3. We can represent the full binary tree with array of length 2^m, where m is none negative integer. Implement the binary tree forest, insert, and random access. What are the performance?

12.4 paired-array sequence

We give paired-array queue in chapter 11. We can expand it to paired-array sequence as array supports random access. As shown in figure 12.5, we link two arrays head to head. When add an element from left, we append to the tail of f; when add from right, we append to the tail of r. We denote the sequence as a pair $S = (f, r)$, Front$(S) = f$, Rear$(S) = r$ access them respectively. We implement insert/append as below:

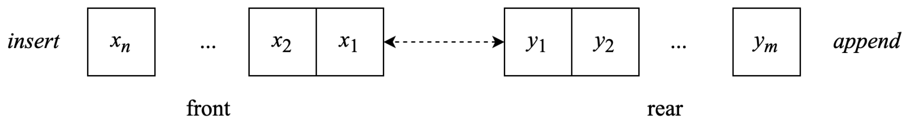


Figure 12.5: Paired-array sequence.

function Insert(x, S)
    Append$(x$, Front$(S))$
function Append(x, S)
    Append$(x$, Rear$(S))$

When access the i-th element, we first determine i index to f or r, then locate the position. If $i \leq |f|$, the element is in f. Because f and r are connected head to head, we need index from right of f at position $|f| - i + 1$; if $i > |f|$, the element is in r. We index from left at position $i - |f|$.

function Get(i, S)
    $f, r \leftarrow$ Front(S), Rear(S)
    $n \leftarrow$ Size(f)
    if $i \leq n$ **then**
        return $f[n - i + 1]$ ▷ reversed
    else
        return $r[i - n]$

Removing can makes f or r empty ([ ]), while the other is not. To re-balance, we halve the none empty one, and reverse either half to form a new pair. As f and r are symmetric, we can swap them, call Balance, then swap back.

function Balance(S)
    $f \leftarrow$ Front(S), $r \leftarrow$ Rear(S)
    $n \leftarrow$ Size(f), $m \leftarrow$ Size(r)
    if $F = [\ ]$ **then**
        $k \leftarrow \lfloor \frac{m}{2} \rfloor$
        return (Reverse$(r[1...k]), r[(k+1)...m])$
    if $R = [\ ]$ **then**
        $k \leftarrow \lfloor \frac{n}{2} \rfloor$
        return $(f[(k+1)...n]$, Reverse$(f[1...k]))$
    return (f, r)

Every time when delete, we check f, r and balance them:

function Remove-Head(S)
    Balance(S)
    $f, r \leftarrow$ Front(S), Rear(S)
    if $f = [\ ]$ **then** ▷ $S = ([], [x])$
        $r \leftarrow [\ ]$

```
    else
        REMOVE-LAST(f)

function REMOVE-TAIL(S)
    BALANCE(S)
    f, r ← FRONT(S), REAR(S)
    if r = [ ] then                              ▷ S = ([x], [])
        f ← [ ]
    else
        REMOVE-LAST(r)
```

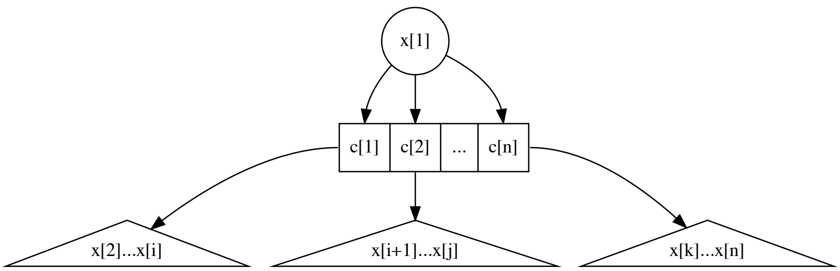
Due to reverse, the performance is $O(n)$ in the worst case, where n is the number of elements, while it is amortized constant time.

Exercise 12.3

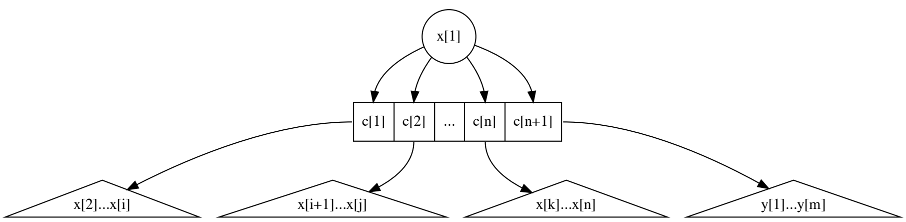
1. For paired-array delete, prove the amortized performance is constant time.

12.5 Concatenate-able list

We achieve $O(\lg n)$ time insert, delete, random index with binary tree forest. However, it's not easy to concatenate two sequences. We can't merely merge trees, but need link trees with the same size. Figure 12.6 shows an implementation of concatenate-able list. The first element x_1 is in root, the rest is organized with smaller sequences, each one is a sub-tree. These sub-trees are put in a real-time queue (see chapter 11). We denote the sequence as $(x_1, Q_x) = [x_1, x_2, ..., x_n]$. When concatenate with another sequence of $(y_1, Q_y) = [y_1, y_2, ..., y_m]$, we append it to Q_x. The real-time queue guarantees the en-queue in constant time, hence the concatenate performance is in constant time.



(a) $(x_1, Q_x) = [x_1, x_2, ..., x_n]$



(b) Concatenate with $(y_1, Q_y) = [y_1, y_2, ..., y_m]$, add$c_{n+1}$ to Q_x

Figure 12.6: Concatenate-able list

$$
\begin{array}{rcl}
s \doubleplus \varnothing & = & s \\
\varnothing \doubleplus s & = & s \\
(x, Q) \doubleplus s & = & (x,\ push\ s\ Q)
\end{array}
\tag{12.12}
$$

When insert new element z, we create a singleton of $(z, \varnothing)$, then concatenate it to the sequence:

$$
\begin{cases}
insert\ x\ s & = (x, \varnothing) \doubleplus s \\
append\ x\ s & = s \doubleplus (x, \varnothing)
\end{cases}
\tag{12.13}
$$

When delete x_1 from head, we lose the root. The rest sub-trees are all concatenate-able lists. We concatenate them all to a new sequence.

$$
\begin{array}{rcl}
concat\ \varnothing & = & \varnothing \\
concat\ Q & = & (top\ Q) \doubleplus concat\ (pop\ Q)
\end{array}
\tag{12.14}
$$

The real-time queue hold sub-trees, we pop the first c_1, and recursively concatenate the rest to s, then concatenate c_1 and s. We define delete from head with *concat*.

$$
tail\ (x, Q) = concat\ Q \tag{12.15}
$$

Function *concat* traverses the queue, and reduces to a result, it essentially folds on Q[10].

$$
\begin{array}{rcl}
fold\ f\ z\ \varnothing & = & z \\
fold\ f\ z\ Q & = & f\ (top\ Q)\ (fold\ f\ z\ (pop\ Q))
\end{array}
\tag{12.16}
$$

Where f is a binary function, z is zero unit. Here are examples of folding on queue $Q = [1, 2, ..., 5]$:

$$
\begin{array}{rcl}
fold\ (+)\ 0\ Q & = & 1 + (2 + (3 + (4 + (5 + 0)))) = 15 \\
fold\ (\times)\ 1\ Q & = & 1 \times (2 \times (3 \times (4 \times (5 \times 1)))) = 120 \\
fold\ (\times)\ 0\ Q & = & 1 \times (2 \times (3 \times (4 \times (5 \times 0)))) = 0
\end{array}
$$

We can define *concat* with fold (Curried form):

$$
concat = fold\ (\doubleplus)\ \varnothing \tag{12.17}
$$

The performance is bound to linear time in worst case: delete after repeatedly add n elements. All $n - 1$ sub-trees are singleton. *concat* takes $O(n)$ time to consolidate. The amortized performance is constant time if add, append, delete randomly happen.

Exercise 12.4

1. Prove the amortized performance for delete is constant time

12.6 Finger tree

Binary random access list supports to insert, remove from head in amortized constant time, and index in logarithm time. But we can't easily append element to tail, or fast concatenate. With concatenate-able list, we can concatenate, insert, and append in amortized constant time, but can't easily index element. From these two examples, we need: 1, access head, tail fast to insert or delete; 2, the recursive structure, e.g., tree, realizes random access as divide and conquer search. Finger tree[66] implements sequence with these two ideas[65]. It's critical to maintain the tree balanced to guarantee search performance. Finger tree leverages 2-3 tree (a type of B-tree). A 2-3 tree is consist of 2 or 3 sub-trees, as (t_1, t_2) or (t_1, t_2, t_3).

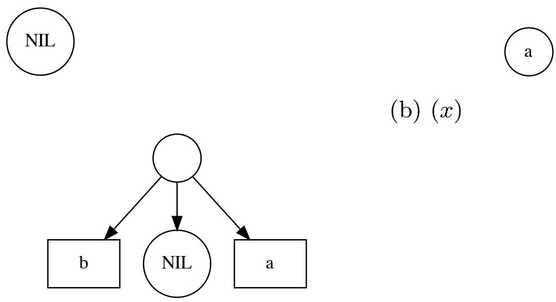
```
data Node a = Br2 a a | Br3 a a a
```

We define a finger tree as one of below three:

1. empty $\varnothing$;
2. a singleton leaf (x);
3. a tree with three parts: a sub-tree, left and right finger, denoted as (f, t, r). Each finger is a list up to 3 elements[1].

```
data Tree a = Empty
            | Lf a
            | Tr [a] (Tree (Node a)) [a]
```

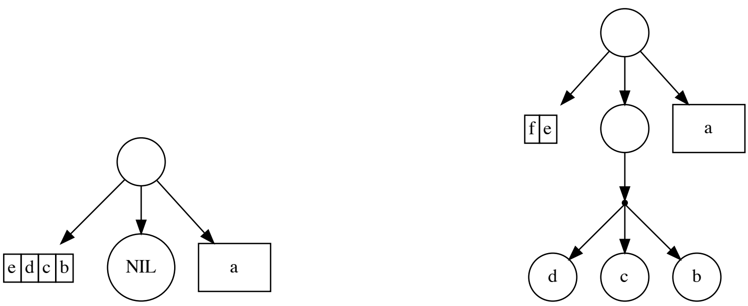
12.6.1 Insert



(a) $\varnothing$ (b) (x)

(c) $([b], \varnothing, [a])$

Figure 12.7: Finger tree, example 1



(a) Insert 3 elements to f, not a valid, balanced 2-3 tree.

(b) Re-balance. there are two elements in f; the middle is a singleton of a 2-3 tree.

Figure 12.8: Finger tree, example 2

[1]f: front, r: rear

As shown in figure 12.7 and 12.8. Example 1 (a) is $\varnothing$, (b) is a singleton, (c) has two element in f and r for each. When add more, f will exceeds 2-3 tree, as in example 2 (a). We need re-balance as in (b). There are two elements in f, the middle is singleton of a 2-3 tree. These examples are list as below:

$\varnothing$	`Empty`
(a)	`Lf a`
$([b], \varnothing, [a])$	`Tr [b] Empty [a]`
$([e, d, c, b], \varnothing, [a])$	`Tr [e, d, c, b] Empty [a]`
$([f, e], (d, c, b), [a])$	`Tr [f, e] Lf (Br3 d c b) [a]`

In example 2 (b), the middle component is a singleton leaf. Finger tree is recursive, apart from f and r, the middle is a deeper finger tree of type $Tree\ (Node\ a)$. One more wrap, one level deeper. Summarize above examples, we define insert a to tree T as below:

1. If $T = \varnothing$, the result is a singleton (a);
2. If $T = (b)$ is a leaf, the result is $([a], \varnothing, [b])$;
3. For $T = (f, t, r)$, if there are ≤ 3 elements in f, we insert a to f, otherwise (> 3), extract the last 3 elements from f to a 2-3 tree t', recursively insert t' to t, then insert a to f.

$$
\begin{array}{rcl}
insert\ a\ \varnothing & = & (x) \\
insert\ a\ (b) & = & ([a], \varnothing, [b]) \\
insert\ a\ ([b, c, d, e], t, r) & = & ([a, b], insert\ (c, d, e)\ t, r) \\
insert\ a\ (f, t, r) & = & (a{:}f, t, r)
\end{array}
\tag{12.18}
$$

The insert performance is constant time except for the recursive case. The recursion time is proportion to the height of the tree h. Because of 2-3 trees, it's balanced, hence $h = O(\lg n)$, where n is the number of elements. When distribute the recursion to other cases, the amortized performance is constant time[3] [65]. We can repeatedly insert a list of elements by folding:

$$
xs \gg t = foldr\ insert\ t\ xs \tag{12.19}
$$

Exercise 12.5

1. Eliminate recursion, implement insert with loop.

12.6.2 Extract

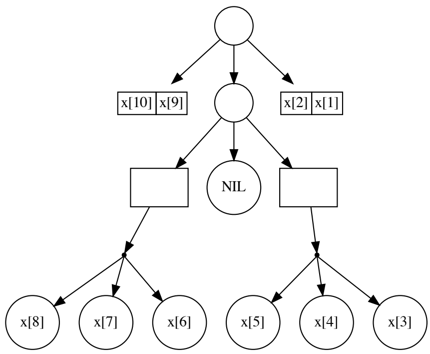
We implement extract as the reverse of *insert*.

$$
\begin{array}{rcl}
extract\ (a) & = & (a, \varnothing) \\
extract\ ([a], \varnothing, [b]) & = & (a, (b)) \\
extract\ ([a], \varnothing, b{:}bs) & = & (a, ([b], \varnothing, bs)) \\
extract\ ([a], t, r) & = & (a, (toList\ f, t', r)), \text{where} : (f, t') = extract\ t \\
extract\ (a{:}as, t, r) & = & (a, (as, t, r))
\end{array}
\tag{12.20}
$$

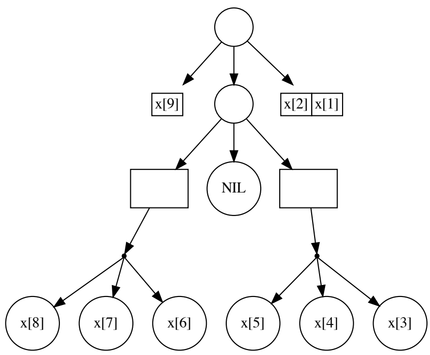
Where *toList* flatten a 2-3 tree to list:

$$
\begin{array}{rcl}
toList\ (a, b) & = & [a, b] \\
toList\ (a, b, c) & = & [a, b, c]
\end{array}
\tag{12.21}
$$

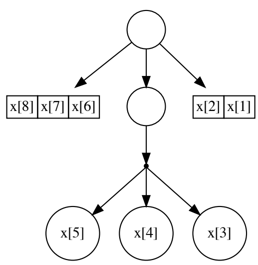
We skip error handling (e.g., extract from empty tree). If the tree is a singleton leaf, the result is empty; if there are two elements, the result is a singleton; if f is a singleton list, the middle is empty, while r isn't empty, we extract the only one in f, then borrow one from r to f; if the middle isn't empty, we recursively extract a node from the middle, flatten that node to list to replace f (the original one is extracted). If f has more than one element, we extract the first. Figure 12.9 gives examples that extract 2 elements.



(a) A sequence of 10 elements.



(b) Extract one, f becomes a singleton list.



(c) Extract another, borrow an element from the middle, flatten the 2-3 tree to a list as the new f.

Figure 12.9: Extract

We can define *head*, *tail* with *extract*.

$$\begin{cases} head &= fst \circ extract \\ tail &= snd \circ extract \end{cases} \tag{12.22}$$

Exercise 12.6

1. Eliminate recursion, implement extract in loops.

12.6.3 Append and remove

We implement append, remove on right symmetrically.

$$
\begin{array}{rcl}
append\ \varnothing\ a &=& (a) \\
append\ (a)\ b &=& ([a], \varnothing, [b]) \\
append\ (f, t, [a, b, c, d])\ e &=& (f, append\ t\ (a, b, c), [d, e]) \\
append\ (f, t, r)\ a &=& (f, t, r \mathbin{+\!\!+} [a])
\end{array}
\tag{12.23}
$$

If there are no more 4 elements in r, we append the new element to tail of r. Otherwise, we extract the first 3 from r, form a new 2-3 tree, and recursively append it to the middle. We can repeatedly append a list of elements by folding from left:

$$
t \ll xs = foldl\ append\ t\ xs \tag{12.24}
$$

The remove is reversed operation of append:

$$
\begin{array}{rcl}
remove\ (a) &=& (\varnothing, a) \\
remove\ ([a], \varnothing, [b]) &=& ((a), b) \\
remove\ (f, \varnothing, [a]) &=& ((init f, \varnothing, [last f]), a) \\
remove\ (f, t, [a]) &=& ((f, t', toList\ r), a), \text{其中} : (t', r) = remove\ t \\
remove\ (f, t, r) &=& ((f, t, init\ r), last\ r)
\end{array}
\tag{12.25}
$$

Where *last* accesses the last element of a list, *init* returns the rest (see chapter 1).

12.6.4 concatenate

When concatenate two none empty finger trees $T_1 = (f_1, t_1, r_1)$, $T_2 = (f_2, t_2, r_2)$, we use f_1 as the result front f, r_2 as the result rear r. Then merge t_1, r_1, f_2, t_2 as the middle tree. Because both r_1 and f_2 are list of nodes, it equivalent to the below problem:

$$
merge\ t_1\ (r_1 \mathbin{+\!\!+} f_2)\ t_2 = ?
$$

Both t_1 and t_2 are finger trees deeper than T_1 and T_2 a level. If the type of element in T_1 is a, then the type of element in t_1 $Node\ a$. We recursively merge, keep the front of t_1 and rear of t_2, then further merge the middle of t_1, t_2, and the rear of t_1, the front of t_2.

$$
\begin{array}{rcl}
merge\ \varnothing\ ts\ t_2 &=& ts \gg t_2 \\
merge\ t_1\ ts\ \varnothing &=& t_1 \ll ts \\
merge\ (a)\ ts\ t_2 &=& merge\ \varnothing\ (a{:}ts)\ t_2 \\
merge\ t_1\ ts\ (a) &=& merge\ t_1\ (ts \mathbin{+\!\!+} [a])\ \varnothing \\
merge\ (f_1, t_1, r_1)\ ts\ (f_2, t_2, r_2) &=& (f_1, merge\ t_1\ (nodes\ (r_1 \mathbin{+\!\!+} ts \mathbin{+\!\!+} f_2))\ t_2, r_2)
\end{array}
\tag{12.26}
$$

Where *nodes* collects elements to a list of 2-3 trees. This is because type of the element in the middle is deeper than the finger.

$$
\begin{array}{rcl}
nodes\ [a, b] &=& [(a, b)] \\
nodes\ [a, b, c] &=& [(a, b, c)] \\
nodes\ [a, b, c, d] &=& [(a, b), (c, d)] \\
nodes\ (a{:}b{:}c{:}ts) &=& (a, b, c){:}nodes\ ts
\end{array}
\tag{12.27}
$$

We then define finger tree concatenation with *merge*:

$$(f_1, t_1, r_1) \doubleplus (f_2, t_2, r_2) = (f_1, merge\ t_1\ (r_1 \doubleplus f_2)\ t_2, r_2) \tag{12.28}$$

Compare with (12.26), concatenation is essentially merge, we can define them in a unified way:

$$T_1 \doubleplus T_2 = merge\ T_1\ [\,]\ T_2 \tag{12.29}$$

The performance is proportion to the number of recursions, which is the smaller height of the two trees. The 2-3 trees are balanced, the height is $O(\lg n)$, where n is the number of elements. In edge cases, merge performs as same as insert (call *insert* at most 8 times) in amortized constant time; In worst case, the performance is $O(m)$, where m is the height difference between the two trees. The overall performance is bound $O(\lg n)$, where n is the total elements of the two trees.

12.6.5 Random access

The idea is to turn random access into tree search. To avoid repeatedly compute tree size, we augment a size variable s to each branch node as (s, f, t, r).

```
data Tree a = Empty
            | Lf a
            | Tr Int [a] (Tree (Node a)) [a]
```

$$\begin{array}{rcl} size\ \varnothing & = & 0 \\ size\ (x) & = & size\ x \\ size\ (s, f, t, r) & = & s \end{array} \tag{12.30}$$

Here $size\ (x)$ is not necessarily 1. x can be a deeper node, like $Node\ a$. It is only 1 at level one. For termination, we wrap x as an element cell $(x)_e$, and define $size\ (x)_e = 1$ (see the example in appendix).

$$\begin{cases} x \lhd t = insert\ (x)_e\ t \\ t \rhd x = append\ t\ (x)_e \end{cases} \tag{12.31}$$

and:

$$\begin{cases} xs \ll t = foldr\ (\lhd)\ t\ xs \\ t \gg xs = foldl\ (\rhd)\ t\ xs \end{cases} \tag{12.32}$$

We also need calculate the size of a 2-3 tree:

$$\begin{array}{rcl} size\ (t_1, t_2) & = & size\ t_1 + size\ t_2 \\ size\ (t_1, t_2, t_3) & = & size\ t_1 + size\ t_2 + size\ t_3 \end{array} \tag{12.33}$$

Given a list of nodes (e.g., finger at deeper level), we calculate size from $sum \circ (map\ size)$. We need update the size when insert or delete element. With size augmented, we can lookup the tree for any position i. The finger tree (s, f, t, r) has recursive structure. Let the size of these components be s_f, s_t, s_r, and $s = s_f + s_t + s_r$. If $i \leq s_f$, the location is in f, we further lookup f; if $s_f < i \leq s_f + s_t$, then the location is in t, we need recursively lookup t; otherwise, we lookup r. We also need handle leaf case of

(x). We use a pair (i, t) to define the position i at data structure t, and define $lookup_T$ as below:

$$
\begin{array}{rcl}
lookup_T\ i\ (x) & = & (i, x) \\
lookup_T\ i\ (s, f, t, r) & = & \begin{cases} i < s_f: & lookup_s\ i\ f \\ s_f \leq i < s_f + s_t: & lookup_N\ (lookup_T\ (i - s_f)\ t) \\ \text{otherwise}: & lookup_s\ (i - s_f - s_t)\ r \end{cases}
\end{array}
\tag{12.34}
$$

Where $s_f = sum\ (map\ size\ f), s_t = size\ t$, are the sizes of the first two components. When lookup location i, if the tree is a leaf (x), the result is (i, x); otherwise we need figure out which component among (s, f, t, r) that i points to. If it either in f or r, then we lookup the figure:

$$
lookup_s\ i\ (x{:}xs) = \begin{cases} i < size\ x: & (i, x) \\ \text{otherwise}: & lookup_s\ (i - size\ x)\ xs \end{cases}
\tag{12.35}
$$

If i is in some element x $(i < size\ x)$, we return (i, x); otherwise, we continue looking up the rest elements. If i points to the middle t, we recursively lookup to obtain a place (i', m), where m is a 2-3 tree. We next lookup m:

$$
\begin{array}{rcl}
lookup_N\ i\ (t_1, t_2) & = & \begin{cases} i < size\ t_1: & (i, t_1) \\ \text{otherwise}: & (i - size\ t_1, t_2) \end{cases} \\
lookup_N\ i\ (t_1, t_2, t_3) & = & \begin{cases} i < size\ t_1: & (i, t_1) \\ size\ t_1 \leq i < size\ t_1 + size\ t_2: & (i - size\ t_1, t_2) \\ \text{otherwise}: & (i - size\ t_1 - size\ t_2, t_3) \end{cases}
\end{array}
\tag{12.36}
$$

Because we previously wrapped x inside $(x)_e$, we need extract x out finally:

$$
T[i] = \begin{cases} \text{if } lookup_T\ i\ T = (i', (x)_e): & Just\ x \\ \text{otherwise}: & Nothing \end{cases}
\tag{12.37}
$$

We return the result of type $Maybe\ a = Nothing | Just\ a$, means either found, or lookup failed[2]. The random access looks up the finger tree recursively, proportion to the tree depth. Because finger tree is balanced, the performs is bound to $O(\lg n)$, where n is the number of elements.

We achieved balanced performance with finger tree implementation. The operations at head and tail are bound to amortized constant time, concatenation, split, and random access are in logarithm time[67]. By the end of this chapter, we've seen many elementary data structures. They are useful to solve some classic problems. For example, we can use sequence to implement MTF (move-to-front[3]) encoding algorithm[68]. MTF move any element at position i to the front of the sequence:

$$
mtf\ i\ S = x \lhd S', \text{where}(x, S') = extractAt\ i\ S
$$

In the next chapters, we'll go through the classic divide and conquer sorting algorithms, including quick sort, merge sort and their variants; then give the string matching algorithms and elementary search algorithms.

Exercise 12.7

1. For random access, how to handle empty tree $\varnothing$ and out of bound cases?
2. Implement $cut\ i\ S$, split sequence S at position i.

[2]Many programming environments provide equivalent tool, like the `Optional<T>` in Java/C++.

[3]Used in Burrows-Wheeler transform (BWT) data compression algorithm.

12.7 Appendix - example programs

Binary random access list (forest):

```
data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)

type BRAList a = [Tree a]

size (Leaf _) = 1
size (Node sz _ _) = sz

link t1 t2 = Node (size t1 + size t2) t1 t2

insert x = insertTree (Leaf x) where
    insertTree t [] = [t]
    insertTree t (t':ts) = if size t < size t' then  t:t':ts
                           else insertTree (link t t') ts

extract ((Leaf x):ts) = (x, ts)
extract ((Node _ t1 t2):ts) = extract (t1:t2:ts)

head' = fst ∘ extract
tail' = snd ∘ extract

getAt i (t:ts) | i < size t = lookupTree i t
               | otherwise = getAt (i - size t) ts
  where
    lookupTree 0 (Leaf x) = x
    lookupTree i (Node sz t1 t2)
        | i < sz `div` 2 = lookupTree i t1
        | otherwise = lookupTree (i - sz `div` 2) t2
```

Numeric representation of binary random access list:

```
data Digit a = Zero | One (Tree a)

type RAList a = [Digit a]

insert x = add (Leaf x) where
  add t [] = [One t]
  add t (Zero:ts) = One t : ts
  add t (One t' :ts) = Zero : add (link t t') ts

minus [One t] = (t, [])
minus (One t:ts) = (t, Zero:ts)
minus (Zero:ts) = (t1, One t2:ts') where
    (Node _ t1 t2, ts') = minus ts

head' ts = x where (Leaf x, _) = minus ts
tail' = snd ∘ minus
```

Paired-array sequence:

```
Data Seq<K> {
    [K] front = [], rear = []
}

Int length(S<K> s) = length(s.front) + length(s.rear)

void insert(K x, Seq<K> s) = append(x, s.front)

void append(K x, Seq<K> s) = append(x, s.rear)

K get(Int i, Seq<K> s) {
```

```
    Int n = length(s.front)
    return if i < n then s.front[n - i - 1] else s.rear[i - n]
}
```

Concatenate-able list:

```
data CList a = Empty | CList a (Queue (CList a))

wrap x = CList x emptyQ

x ++ Empty = x
Empty ++ y = y
(CList x q) ++ y = CList x (push q y)

fold f z q | isEmpty q = z
           | otherwise = (top q) `f` fold f z (pop q)

concat = fold (++) Empty

insert x xs = (wrap x) ++ xs
append xs x = xs ++ wrap x

head (CList x _) = x
tail (CList _ q) = concat q
```

Finger tree:

```
-- 2-3 tree
data Node a = Tr2 Int a a
            | Tr3 Int a a a

-- finger tree
data Tree a = Empty
            | Lf a
            | Br Int [a] (Tree (Node a)) [a] -- size, front, mid, rear

newtype Elem a = Elem { getElem :: a } -- wrap element

newtype Seq a = Seq (Tree (Elem a)) -- sequence

class Sized a where  -- support size measurement
  size :: a -> Int

instance Sized (Elem a) where
  size _ = 1  -- 1 for any element

instance Sized (Node a) where
  size (Tr2 s _ _) = s
  size (Tr3 s _ _ _) = s

instance Sized a => Sized (Tree a) where
  size Empty = 0
  size (Lf a) = size a
  size (Br s _ _ _) = s

instance Sized (Seq a) where
  size (Seq xs) = size xs

tr2 a b = Tr2 (size a + size b) a b
tr3 a b c = Tr3 (size a + size b + size c) a b c

nodesOf (Tr2 _ a b) = [a, b]
nodesOf (Tr3 _ a b c) = [a, b, c]

-- left
x <| Seq xs = Seq (Elem x `cons` xs)
```

```
cons :: (Sized a) ⇒ a → Tree a → Tree a
cons a Empty = Lf a
cons a (Lf b) = Br (size a + size b) [a] Empty [b]
cons a (Br s [b, c, d, e] m r) = Br (s + size a) [a, b] ((tr3 c d e) `cons` m) r
cons a (Br s f m r) = Br (s + size a) (a:f) m r

head' (Seq xs) = getElem $ fst $ uncons xs
tail' (Seq xs) = Seq $ snd $ uncons xs

uncons :: (Sized a) ⇒ Tree a → (a, Tree a)
uncons (Lf a) = (a, Empty)
uncons (Br _ [a] Empty [b]) = (a, Lf b)
uncons (Br s [a] Empty (r:rs)) = (a, Br (s - size a) [r] Empty rs)
uncons (Br s [a] m r) = (a, Br (s - size a) (nodesOf f) m' r)
    where (f, m') = uncons m
uncons (Br s (a:f) m r) = (a, Br (s - size a) f m r)

-- right
Seq xs |> x  = Seq (xs `snoc` Elem x)

snoc :: (Sized a) ⇒ Tree a → a → Tree a
snoc Empty a = Lf a
snoc (Lf a) b = Br (size a + size b) [a] Empty [b]
snoc (Br s f m [a, b, c, d]) e = Br (s + size e) f (m `snoc` (tr3 a b c)) [d, e]
snoc (Br s f m r) a = Br (s + size a) f m (r ++ [a])

last' (Seq xs) = getElem $ snd $ unsnoc xs
init' (Seq xs) = Seq $ fst $ unsnoc xs

unsnoc :: (Sized a) ⇒ Tree a → (Tree a, a)
unsnoc (Lf a) = (Empty, a)
unsnoc (Br _ [a] Empty [b]) = (Lf a, b)
unsnoc (Br s f@(_:_:_) Empty [a]) = (Br (s - size a) (init f) Empty [last f], a)
unsnoc (Br s f m [a]) = (Br (s - size a) f m' (nodesOf r), a)
    where (m', r) = unsnoc m
unsnoc (Br s f m r) = (Br (s - size a) f m (init r), a) where a = last r

-- concatenate
Seq xs ++ Seq ys = Seq (xs >+< ys)

xs >+< ys = merge xs [] ys

t <<< xs = foldl snoc t xs
xs >>> t = foldr cons t xs

merge :: (Sized a) ⇒ Tree a → [a] → Tree a → Tree a
merge Empty es t2 = es >>> t2
merge t1 es Empty = t1 <<< es
merge (Lf a) es t2 = merge Empty (a:es) t2
merge t1 es (Lf a) = merge t1 (es ++ [a]) Empty
merge (Br s1 f1 m1 r1) es (Br s2 f2 m2 r2) =
    Br (s1 + s2 + (sum $ map size es)) f1 (merge m1 (trees (r1 ++ es ++ f2)) m2) r2

trees [a, b] = [tr2 a b]
trees [a, b, c] = [tr3 a b c]
trees [a, b, c, d] = [tr2 a b, tr2 c d]
trees (a:b:c:es) = (tr3 a b c):trees es

-- index
data Place a = Place Int a

getAt :: Seq a → Int → Maybe a
getAt (Seq xs) i | i < size xs = case lookupTree i xs of
                     Place _ (Elem x) → Just x
```

```
                  | otherwise = Nothing

lookupTree :: (Sized a) ⇒ Int → Tree a → Place a
lookupTree n (Lf a) = Place n a
lookupTree n (Br s f m r) | n < sf = lookups n f
                          | n < sm = case lookupTree (n - sf) m of
                                        Place n' xs → lookupNode n' xs
                          | n < s = lookups (n - sm) r
  where sf = sum $ map size f
        sm = sf + size m

lookupNode :: (Sized a) ⇒ Int → Node a → Place a
lookupNode n (Tr2 _ a b) | n < sa = Place n a
                         | otherwise = Place (n - sa) b
  where sa = size a
lookupNode n (Tr3 _ a b c) | n < sa = Place n a
                           | n < sab = Place (n - sa) b
                           | otherwise = Place (n - sab) c
  where sa = size a
        sab = sa + size b

lookups :: (Sized a) ⇒ Int → [a] → Place a
lookups n (x:xs) = if n < sx then Place n x
                   else lookups (n - sx) xs
  where sx = size x
```

Chapter 13

Quick sort and merge sort

13.1 Introduction

People proved the performance upper limit be $O(n \lg n)$ for comparison based sort[51]. This chapter gives two divide and conquer sort algorithms: quick sort and merge sort, both achieve $O(n \lg n)$ time bound. We also give their variants, like natural merge sort, in-place merge sort, and etc.

13.2 Quick sort

Consider arrange kids in a line ordered by height.

1. The first kid raises hand, all shorter one move to left, and the others move to right;
2. All kids on the left and right repeat.

For example, the heights (in cm) are $[102, 100, 98, 95, 96, 99, 101, 97]$. Table 13.1 gives the steps. (1) The kid of 102 cm raises hand as the pivot (underlined in the first row). It happens the tallest, hence all others move to the left as shown in the second row in the table. (2) The kid of 100 cm is the pivot. Kids of height 98, 95, 96, and 99 cm move to the left, and the kid of 101 cm move to the right, as shown in the third row. (3) The kid of 98 cm is the left pivot, while 101 cm is the right pivot. Because there is only one kid on the right, it's sorted. Repeat this to sort all kids.

<u>102</u>	100	98	95	96	99	101	97
<u>100</u>	98	95	96	99	101	97	'102'
<u>98</u>	95	96	99	97	'100'	101	'102'
<u>95</u>	96	97	'98'	99	'100'	'101'	'102'
'95'	<u>96</u>	97	'98'	'99'	'100'	'101'	'102'
'95'	'96'	97	'98'	'99'	'100'	'101'	'102'
'95'	'96'	'97'	'98'	'99'	'100'	'101'	'102'

Table 13.1: Sort steps

We can summarize the quick sort definition, when sort list L:

- If L is empty[ ], the result is [ ];
- Otherwise, select an element as the pivot p, recursively sort elements $\leq p$ to the left; *and* sort other elements $> p$ to the right.

We say *and*, but not 'then', indicate we can parallel sort left and right. C. A. R. Hoare developed quick sort in 1960[51] [78]. There are varies of ways to pick the pivot, for example, always choose the first element.

$$\begin{array}{rcl} sort\ [\,] & = & [\,] \\ sort\ (x{:}xs) & = & sort\ [y|y \in xs, y \leq x] \doubleplus [x] \doubleplus sort\ [y|y \in xs, x < y] \end{array} \tag{13.1}$$

We use the Zermelo Frankel expression (ZF expression)[1]. $\{a|a \in S, p_1(a), p_2(a), ...\}$ selects elements in set S, that satisfy every the predication $p_1, p_2, ...$ (see chapter 1). Below is example code:

```
sort [] = []
sort (x:xs) = sort [y | y←xs, y ≤ x] ++ [x] ++ sort [y | y←xs, x < y]
```

We assume to sort in ascending order. We can abstract the comparison to sort different things like numbers, strings, and etc. (see chapter 3) We needn't total ordering, but at least need *strict weak ordering*[79] [52](see chapter 9). We use $\leq$ as the abstract comparison.

13.2.1 Partition

We traverse elements in two passes: first filter all elements $\leq x$; next filter all $> x$. We can combine them into one pass:

$$\begin{array}{rcl} part\ p\ [\,] & = & ([\,],[\,]) \\ part\ p\ (x{:}xs) & = & \begin{cases} p(x): & (x{:}as, bs), \text{where}: (as, bs) = part\ p\ xs \\ \text{otherwise}: & (as, x{:}bs) \end{cases} \end{array} \tag{13.2}$$

And change the quick sort definition to:

$$\begin{array}{rcl} sort\ [\,] & = & [\,] \\ sort\ (x{:}xs) & = & sort\ as \doubleplus [x] \doubleplus sort\ bs, \text{where}: (as, bs) = part\ (\leq x)\ xs \end{array} \tag{13.3}$$

We can also define partition with fold:

$$part\ p\ = foldr\ f\ ([\,],[\,]) \tag{13.4}$$

[1]Name after two mathematicians found the modern set theory.

Where f is defined as:

$$f\ (as, bs)\ x = \begin{cases} p(x): & (x{:}as, bs) \\ \text{otherwise}: & (as, x{:}bs) \end{cases} \tag{13.5}$$

It's essentially to accumulate to (as, bs). If $p(x)$ holds, then add x to as, otherwise to bs. We can implement a tail recursive partition:

$$\begin{array}{rcl} part\ p\ [\ ]\ as\ bs & = & (as, bs) \\ part\ p\ (x{:}xs)\ as\ bs & = & \begin{cases} p(x): & part\ p\ xs\ (x{:}as)\ bs \\ \text{otherwise}: & part\ p\ xs\ as\ (x{:}bs) \end{cases} \end{array} \tag{13.6}$$

To partition $x{:}xs$, we call:

$$(as, bs) = part\ (\leq x)\ xs\ [\ ]\ [\ ]$$

We change concatenation $sort\ as \mathbin{+\!\!+} [x] \mathbin{+\!\!+} sort\ bs$ with accumulator as:

$$\begin{array}{rcl} sort\ s\ [\ ] & = & s \\ sort\ s\ (x{:}xs) & = & sort\ (x : sort\ s\ bs)\ as \end{array} \tag{13.7}$$

Where s is the accumulator, we initialize sort with an empty list: $qsort = sort\ [\ ]$. After partition, we need recursively sort as, bs. We can first sort bs, prepend x, then pass it as the new accumulator to sort as:

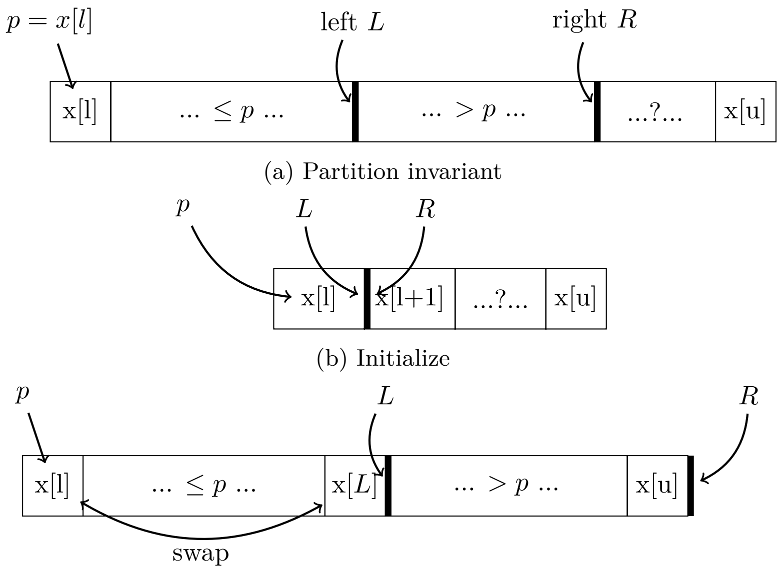
```
sort = sort' []

sort' acc [] = acc
sort' acc (x:xs) = sort' (x : sort' acc bs) as 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)
```

13.2.2 In-place sort

Figure 13.1 gives a way to partition in-place[2] [4]. We scan from left to right. At any time, the array is consist of three parts as shown in figure 13.1 (a):

- The pivot is the left element $p = x[l]$. It moves to the final position after partition;
- A section of elements $\leq p$, extend right to L;
- A section of elements $> p$, extend right to R. The elements between L and $R > p$;
- Elements after R haven't been partitioned (may $>, =, <$ p).

When partition starts, L points to p, R points to the next, as shown in figure 13.1 (b). We advance R to right till reach to the array boundary. Every time, we compare $x[R]$ and p. If $x[R] > p$, it should be between L and R, we move R forward; otherwise if $X[R] \leq p$, it should be on the left of L. We advance L a step, then swap $x[L] \leftrightarrow x[R]$. When R passes the last element, the partition ends. Elements $> p$ move to the right of L, while others on the left side. We need move p to the position between the two parts. To do that, we swap $p \leftrightarrow x[L]$, as shown in 13.1 (c). L finally points to p, partitioned the array in two parts. We return $L + 1$ as the result, that points to the first element $> p$. Let the array be A, the lower, upper boundary be l, u. The in-place partition is defined below:



(a) Partition invariant

(b) Initialize

(c) Terminate

Figure 13.1: In-place partition, pivot $p = x[l]$

function PARTITION(A, l, u)
    $p \leftarrow A[l]$ ▷ pivot
    $L \leftarrow l$ ▷ left
    for R in $[l + 1, u]$ **do** ▷ iterate right
        if $p \geq A[R]$ **then**
            $L \leftarrow L + 1$
            EXCHANGE $A[L] \leftrightarrow A[R]$
    EXCHANGE $A[L] \leftrightarrow p$
    return $L + 1$ ▷ partition position

Table 13.2 lists the steps to partition $[3, 2, 5, 4, 0, 1, 6, 7]$.

3(l)	2(r)	5	4	0	1	6	7	start, $p = 3$、$l = 1$、$r = 2$
3	2(l)(r)	5	4	0	1	6	7	$2 < 3$, advance l $(r = l)$
3	2(l)	5(r)	4	0	1	6	7	$5 > 3$, move on
3	2(l)	5	4(r)	0	1	6	7	$4 > 3$, move on
3	2(l)	5	4	0(r)	1	6	7	$0 < 3$
3	2	0(l)	4	5(r)	1	6	7	advance l, swap with r
3	2	0(l)	4	5	1(r)	6	7	$1 < 3$
3	2	0	1(l)	5	4(r)	6	7	advance l, swap with r
3	2	0	1(l)	5	4	6(r)	7	$6 > 3$, move on
3	2	0	1(l)	5	4	6	7(r)	$7 > 3$, move on
1	2	0	3	5(l+1)	4	6	7	terminate, swap p and l

Table 13.2: Partition array

With PARTITION defined, we implement quick sort as below:

procedure QUICK-SORT(A, l, u)
    if $l < u$ **then**
        $m \leftarrow$ PARTITION(A, l, u)

```
        Quick-Sort(A, l, m - 1)
        Quick-Sort(A, m, u)
```

We pass the array and its boundaries, as Quick-Sort$(A, 1, |A|)$ to sort. When the array is empty or singleton, sort returns immediately.

Exercise 13.1

1. Improve the basic quick sort definition when the list is singleton.

13.2.3 Performance

Quick sort performs well in most cases. We start from the best/worst cases. For the best case, we always halve the elements into two equal sized parts. As shown in figure 13.2, there are total $O(\lg n)$ levels of recursions. At level one, we processes n elements with one partition; at level two, we partition twice, each processes $n/2$ elements, taking total $2O(n/2) = O(n)$ time; at level three, we partition four times, each process $n/4$ elements, taking total $O(n)$ time too, ..., at the last level, there are n singleton segments, taking total $O(n)$ time. Sum all levels, the time is bound to $O(n \lg n)$.

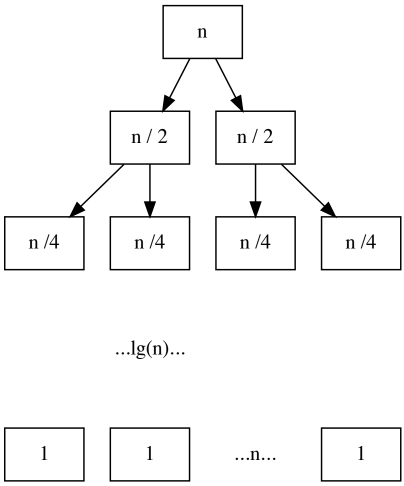


Figure 13.2: The best case, halve every time.

For the worst case, the partition is totally unbalanced, one part is of $O(1)$ length, the other is $O(n)$. The level of recursions decays to $O(n)$. Model the partition as a tree. It's balanced binary tree in the best case, while it becomes a linked-list of $O(n)$ length in the worst case. Every branch node has an empty sub-tree. At each level, we process all elements, hence the total time is bound to $O(n^2)$. This is same as insertion sort, and selection sort. We can list several worst cases, for example, there are many duplicated elements, or the sequence is largely ordered, and so on. There isn't a method can avoid the worst case completely.

Average case★

Quick sort performs well in average. For example, even if every partition gives two parts of 1:9, the performance still achieves $O(n \lg n)$[4]. We give two method to evaluate the performance. The first one is based on the fact, that the performance is proportion to the number of comparisons. In selection sort, every two elements are compared, while in quick sort, we save many comparisons. When partition sequence $[a_1, a_2, a_3, ..., a_n]$ with a_1 as the pivot, we obtain two sub sequences $A = [x_1, x_2, ..., x_k]$ and $B = [y_1, y_2, ..., y_{n-k-1}]$. After that, none element in A will compare with any one in B. Let the sorted result be $[a_1, a_2, ..., a_n]$, if $a_i < a_j$, we do not compare them if and only if there is some element a_k, where $a_i < a_k < a_j$, is picked as the pivot before either a_i or a_j being the pivot. In other word, the only chance that we compare a_i and a_j is either a_i or a_j is chosen as the pivot before any other elements in $a_{i+1} < a_{i+2} < ... < a_{j-1}$ being the pivot. Let $P(i, j)$ be the probability that we compare a_i and a_j. We have:

$$P(i,j) = \frac{2}{j-i+1} \tag{13.8}$$

The total number of comparisons is:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(i,j) \tag{13.9}$$

If we compare a_i and a_j, we won't compare a_j and a_i again, and we never compare a_i with itself. The upper bound of i is $n-1$, and the lower bound of j is $i+1$. Substitute the probability:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \end{aligned} \tag{13.10}$$

Use the result of harmonic series[80].

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... = \ln n + \gamma + \epsilon_n$$

$$C(n) = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n) \tag{13.11}$$

The other method uses the recursion. Let the length of the sequence be n, we partition it into two parts of length i and $n-i-1$. The partition takes cn time because it compares every element with the pivot. The total time is:

$$T(n) = T(i) + T(n-i-1) + cn \tag{13.12}$$

Where $T(n)$ is the time to sort n elements. i equally distributes across $0, 1, ..., n-1$. Taking math expectation:

$$\begin{aligned} T(n) &= E(T(i)) + E(T(n-i-1)) + cn \\ &= \frac{1}{n} \sum_{i=0}^{n-1} T(i) + \frac{1}{n} \sum_{i=0}^{n-1} T(n-i-1) + cn \\ &= \frac{1}{n} \sum_{i=0}^{n-1} T(i) + \frac{1}{n} \sum_{j=0}^{n-1} T(j) + cn \\ &= \frac{2}{n} \sum_{i=0}^{b-1} T(i) + cn \end{aligned} \tag{13.13}$$

Multiply n to both sides:

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + cn^2 \tag{13.14}$$

Substitute n to $n-1$:

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + c(n-1)^2 \tag{13.15}$$

Take (13.14) - (13.15), cancel all $T(i)$ for $0 \leq i < n-1$.

$$nT(n) = (n+1)T(n-1) + 2cn - c \tag{13.16}$$

Drop the constant c, we obtain:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} \tag{13.17}$$

Assign n to $n-1$, $n-2$, ..., to give $n-1$ equations.

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}$$

$$\ldots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

Sum up and cancel the same components on both sides, we get a function of n.

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{k=3}^{n+1} \frac{1}{k} \tag{13.18}$$

Use the result of the harmonic series:

$$O(\frac{T(n)}{n+1}) = O(\frac{T(1)}{2} + 2c\ln n + \gamma + \epsilon_n) = O(\lg n) \tag{13.19}$$

Therefore:

$$O(T(n)) = O(n \lg n) \tag{13.20}$$

13.2.4 Improvement

The PARTITION procedure doesn't perform well when there are many duplicated elements. Consider the extreme case that all n elements are equal $[x, x, ..., x]$:

1. From the quick sort definition: pick any element as the pivot, hence $p = x$, partition into two sub-sequences. One is $[x, x, ..., x]$ of length $n-1$, the other is empty. Next recursively sort the $n-1$ elements, the total time decays to $O(n^2)$.

2. Modify the partition with $< x$ and $> x$. The result are two empty sub-sequences, and n elements equal to x. The recursion on empty sequence terminates immediately. The result is $[\,] \mathbin{+\!\!+} [x, x, ..., x] \mathbin{+\!\!+} [\,]$. The performance is $O(n)$.

We improve from *binary* partition to *ternary* partition to handle duplicated elements:

$$\begin{array}{rcl} sort\ [\,] &=& [\,] \\ sort\ (x{:}xs) &=& sort\ S \mathbin{+\!\!+} sort\ E \mathbin{+\!\!+} sort\ G \end{array} \tag{13.21}$$

Where:

$$\begin{cases} S = [y | y \in xs, y < x] \\ E = [y | y \in xs, y = x] \\ G = [y | y \in xs, y > x] \end{cases}$$

To concatenate three lists in linear time, we can use an accumulator: $qsort = sort\ [\,]$, where:

$$\begin{array}{rcl} sort\ A\ [\,] &=& A \\ sort\ A\ (x{:}xs) &=& sort\ (E \mathbin{+\!\!+} sort\ A\ G)\ S \end{array} \tag{13.22}$$

We partition the list in three parts: S, E, G, where E contains elements of same value, hence sorted. We first sort G with accumulator A, append the result to E as the new accumulator, and use it to sort S. We also improve the partition with accumulator:

$$\begin{array}{rcl} part\ S\ E\ G\ x\ [\,] &=& (S, E, G) \\ part\ S\ E\ G\ x\ (y{:}ys) &=& \begin{cases} y < x: & (y{:}S, E, G) \\ y = x: & (S, y{:}E, G) \\ y > x: & (S, E, y{:}G) \end{cases} \end{array} \tag{13.23}$$

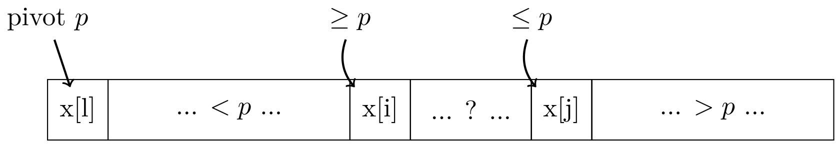
Richard Bird developed another improvement[1], instead concatenate the recursive sort results, put them in a list and concatenate finally:

```
sort :: (Ord a) ⇒ [a] → [a]
sort = concat ∘ (pass [])

pass xss [] = xss
pass xss (x:xs) = step xs [] [x] [] xss where
    step [] as bs cs xss = pass (bs : pass xss cs) as
    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
```

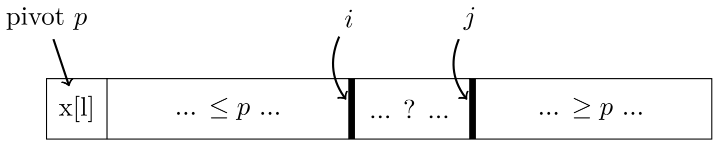
Robert Sedgewick developed two-way partition method[69] [2]. Use two pointers i, j from left and right boundaries. Pick the first element as the pivot p. Advance i to right till an element $\geq p$; while (in parallel) move j to left till an element $\leq p$. At this time, all elements left to i are less than the pivot ($< p$), while those right to j are greater than the pivot ($> p$). i points to one that $\geq p$, and j points to one that $\leq p$, as shown in figure 13.3 (a). To move all elements $\leq p$ to left, and the remaining to right, we exchange $x[i] \leftrightarrow x[j]$, then continue scan. We repeat this till i and j meet. At any time, we keep the invariant: All elements left to i (include i) are $\leq p$; while all right to j (include j) are $\geq p$. The elements between i and j are yet to scan, as shown in figure 13.3 (b).

When i meets j, we need an extra exchange, swap the pivot p to position j. Then recursive sort sub-array $A[l...j)$ and $A[i...u)$.

```
1: procedure Sort(A, l, u)                          ▷ sort range [l, u)
2:     if u − l > 1 then                            ▷ At least 2 elements
3:         i ← l, j ← u
4:         pivot ← A[l]
5:         loop
```



(a) When i and j stop

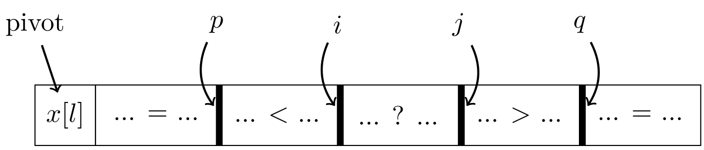


(b) Partition invariant

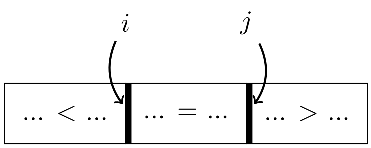
Figure 13.3: 2-way scan

repeat
    $i \leftarrow i + 1$
until $A[i] \geq pivot$ ▷ Ignore $i \geq u$
repeat
    $j \leftarrow j - 1$
until $A[j] \leq pivot$ ▷ Ignore $j < l$
if $j < i$ **then**
    break
EXCHANGE $A[i] \leftrightarrow A[j]$
EXCHANGE $A[l] \leftrightarrow A[j]$ ▷ Move the pivot
SORT(A, l, j)
SORT(A, i, u)

Consider the special case that all elements are equal, the array is partitioned into two same parts with $\frac{n}{2}$ swaps. Because of the balanced partition, the performance is $O(n \lg n)$. It takes less swaps than the one pass scan method, since it skips the elements on the right side of the pivot. We can combine 2-way scan and ternary partition. Only recursively sort the elements different with the pivot. Jon Bentley and Douglas McIlroy developed a method as shown in figure 13.4 (a), that store the elements equal to the pivot on both sides[70] [71].



(a) Ternary partition invariant.



(b) Swap the elements $= p$ to the middle.

Figure 13.4: Ternary partition

We scan from two sides, pause when i reach an element $\geq$ the pivot, and j reach one $\leq$

the pivot. If i doesn't meet or pass j, we exchange $A[i] \leftrightarrow A[j]$, then check if $A[i]$ or $A[j]$ equals to the pivot. If yes, we exchange $A[i] \leftrightarrow A[p]$ or $A[j] \leftrightarrow A[q]$ respectively. Finally, we swap all the elements equal to the pivot to the middle. This step do nothing if all elements are unique. The partition result is shown as 13.4 (b). We next only recursively sort the elements not equal to the pivot.

```
procedure Sort(A, l, u)
    if u − l > 1 then
        i ← l, j ← u
        p ← l, q ← u              ▷ point to the boundaries of duplicated elements
        pivot ← A[l]
        loop
            repeat
                i ← i + 1
            until A[i] ≥ pivot                          ▷ Ignore i ≥ u case
            repeat
                j ← j − 1
            until A[j] ≤ pivot                          ▷ Ignore j < l case
            if j ≤ i then
                break
            Exchange A[i] ↔ A[j]
            if A[i] = pivot then                        ▷ duplicated element
                p ← p + 1
                Exchange A[p] ↔ A[i]
            if A[j] = pivot then
                q ← q − 1
                Exchange A[q] ↔ A[j]
        if i = j and A[i] = pivot then
            j ← j − 1, i ← i + 1
        for k from l to p do        ▷ Swap the duplicated elements to the middle
            Exchange A[k] ↔ A[j]
            j ← j − 1
        for k from u − 1 down-to q do
            Exchange A[k] ↔ A[i]
            i ← i + 1
        Sort(A, l, j + 1)
        Sort(A, i, u)
```

It becomes complex when combine 2-way scan and ternary partition. We can change the one pass scan to ternary partition directly. Pick the first element as the pivot, as shown in figure **??**. At any time, the left part contains elements $< p$; the next part contains those $= p$; and the right part contains those $> p$. The boundaries are i, k, j. Elements between $[k, j)$ are yet to be partitioned. We scan from left to right. When start, the part $< p$ is empty; the part $= p$ has an element; i points to the lower boundary, k points to the next. The part $> p$ is empty too, j points to the upper boundary.

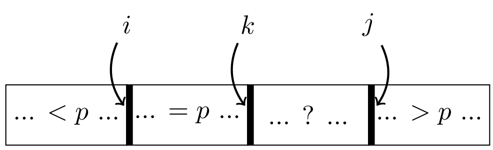


Figure 13.5: 1 way scan ternary partition

We iterate on k, if $A[k] = p$, then move k to the next; if $A[k] > p$, then exchange $A[k] \leftrightarrow A[j-1]$, the range of elements that $> p$ increases by one. Its boundary j moves to left a step. Because we don't know if the element moved to k is still $> p$, we compare again and repeat. Otherwise if $A[k] < p$, we exchange $A[k] \leftrightarrow A[i]$, where $A[i]$ is the first element that $= p$. The partition terminates when k meets j.

```
procedure Sort(A, l, u)
    if u − l > 1 then
        i ← l, j ← u, k ← l + 1
        pivot ← A[i]
        while k < j do
            while pivot < A[k] do
                j ← j − 1
                Exchange A[k] ↔ A[j]
            if A[k] < pivot then
                Exchange A[k] ↔ A[i]
                i ← i + 1
            k ← k + 1
        Sort(A, l, i)
        Sort(A, j, u)
```

Compare with the ternary partition through 2-way scan, this implementation is less complex but need more swaps.

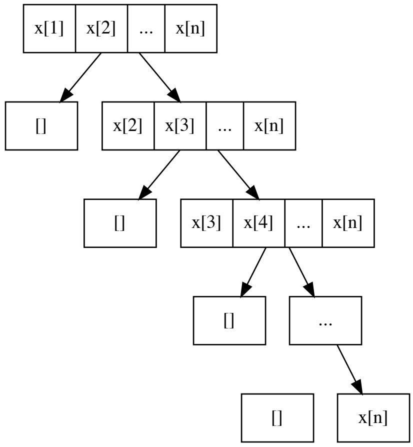
Worst cases

Although ternary partition handles duplicated elements well, there are the worst cases. For example, when most elements are ordered (ascending or descending), the partition is unbalanced. Figure 13.6 gives two of the worst cases: $[x_1 < x_2 < ... < x_n]$ and $[y_1 > y_2 > ... > y_n]$. It's easy to give more, for example: $[x_m, x_{m-1}, ..., x_2, x_1, x_{m+1}, x_{m+2}, ...x_n]$, where $[x_1 < x_2 < ... < x_n]$, and $[x_n, x_1, x_{n-1}, x_2, ...]$ as shown in figure 13.7.

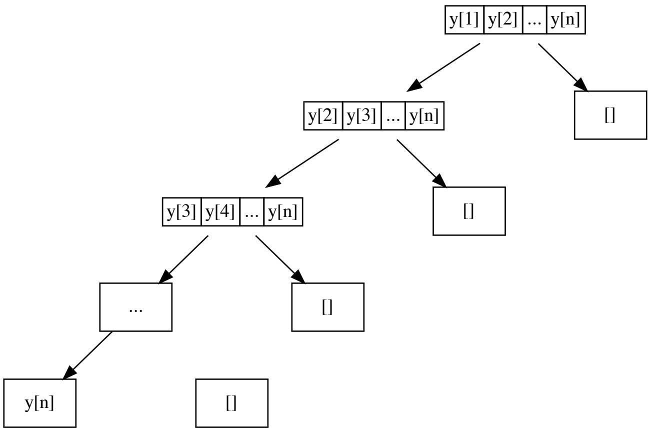
In these worst cases, the partition is unbalanced when choose the first element as the pivot. Robert Sedgwick improved the pivot selection[69]: Instead pick a fixed position, sample several elements to avoid bad pivot. We sample the first, the middle, and the last, pick the median as the pivot. We can either compare every two (total 3 times)[70], or swap the least one to head, swap the greatest one end, and move the median to the middle.

```
procedure Sort(A, l, u)
    if u − l > 1 then
        m ← ⌊(l + u)/2⌋                  ▷ or l + (u − l)/2 to void overflow
        if A[m] < A[l] then              ▷ Ensure A[l] ≤ A[m]
            Exchange A[l] ↔ A[m]
        if A[u − 1] < A[l] then          ▷ Ensure A[l] ≤ A[u − 1]
            Exchange A[l] ↔ A[u − 1]
        if A[u − 1] < A[m] then          ▷ Ensure A[m] ≤ A[u − 1]
            Exchange A[m] ↔ A[u − 1]
        Exchange A[l] ↔ A[m]
        (i, j) ← Partition(A, l, u)
        Sort(A, l, i)
        Sort(A, j, u)
```

This implementation handles the above four worst cases well. We call it 'median of three'. Alternatively, we can randomly pick pivot:

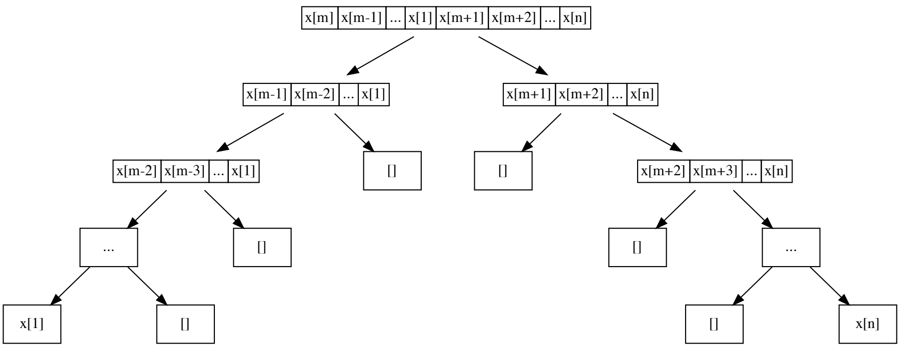


(a) Partition tree of $[x_1 < x_2 < ... < x_n]$, the sub-trees of $\leq p$ are empty.

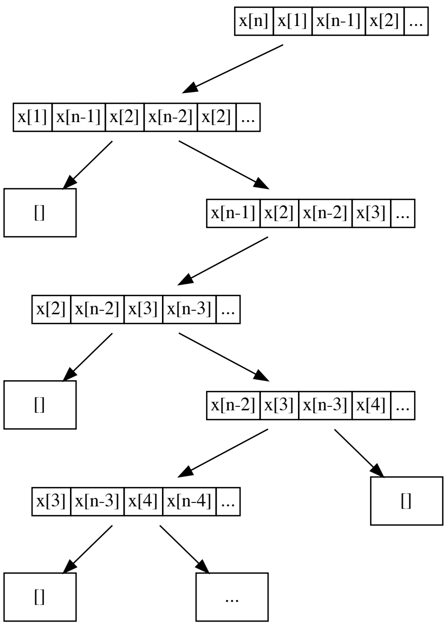


(b) Partition tree of $[y_1 > y_2 > ... > y_n]$, the sub-trees of $\geq p$ are empty.

Figure 13.6: The worst cases - 1.



(a) Unbalanced partitions except for the first time.



(b) A zig-zag partition tree.

Figure 13.7: The worst cases - 2.

procedure SORT(A, l, u)
  if $u - l > 1$ **then**
    EXCHANGE $A[l] \leftrightarrow A[$ RANDOM(l, u) $]$
    $(i, j) \leftarrow$ PARTITION(A, l, u)
    SORT(A, l, i)
    SORT(A, j, u)

Where RANDOM(l, u) returns integer $l \leq i < u$ randomly. We swap $A[i]$ with the first element as the pivot. This method is called *random quick sort*[4]. Theoretically, neither 'median of three' nor random quick sort can avoid the worst case completely. If the sequence is random, it's same to choose any one as the pivot. Nonetheless, these improvements are widely used in engineering practice.

There are other improvements besides partition. Sedgewick found quick sort had overhead when the list is short, while insert sort performed better[2][70]. Sedgewick, Bentley and McIlroy evaluated varies thresholds, as 'cut-off'. When the elements are less than the 'cut-off', then switch to insert sort.

procedure SORT(A, l, u)
  if $u - l >$ CUT-OFF **then**
    QUICK-SORT(A, l, u)
  else
    INSERTION-SORT(A, l, u)

13.2.5 quick sort and tree sort

The 'true quick sort' is the combination of multiple engineering improvements, falls back to insert sort for small sequence, in-place swaps, choose the pivot as the 'median of three', 2-way scan, and ternary partition. Some people think the basic recursive definition is essentially tree sort. Richard Bird derived quick sort from binary tree sort by deforestation[72]. Define *unfold* that converts a list to binary search tree:

$$\begin{array}{rcl} unfold\ [\ ] & = & \varnothing \\ unfold\ (x{:}xs) & = & (unfold\ [a|a \in xs, a \leq x], x, unfold\ [a|a \in xs, a > x]) \end{array} \tag{13.24}$$

Compare with the binary tree insert (see chapter 2), *unfold* creates the tree differently. If the list is empty, the tree is empty; otherwise, use the first element x as the key, then recursively build the left, right sub-trees. Where the left sub-tree has the elements $\leq x$; and the right tree has elements that $> x$. While to convert a binary search tree to ordered list, we define in-order traverse as:

$$\begin{array}{rcl} toList\ \varnothing & = & [\ ] \\ toList\ (l, k, r) & = & toList\ l \mathbin{+\!\!+} [k] \mathbin{+\!\!+} toList\ r \end{array} \tag{13.25}$$

We define quick sort by composing the two functions:

$$sort = toList \circ unfold \tag{13.26}$$

We first build the binary search tree through *unfold*, then pass it to *toList* to generate the list, and discard the tree. When eliminate the intermediate tree (through *deforestation* by Burstle-Darlington's work[?]), we obtain the quick sort.

13.3 Merge sort

Quick sort performs well in most cases. However, there are the worst cases can't be completely avoided. Merge sort guarantees $O(n \lg n)$ performance in all cases. It supports both arrays and lists. Many programming environments provide merge sort as the standard sort tool[2]. Merge sort takes divide and conquer approach. It always splits the sequence in half and half, recursively sort them and merge.

$$\begin{array}{rcl} sort\ [\ ] & = & [\ ] \\ sort\ [x] & = & [x] \\ sort\ xs & = & merge\ (sort\ as)\ (sort\ bs), \text{where} : (as, bs) = halve\ xs \end{array} \tag{13.27}$$

Where *halve* splits the sequence, for array, we can cut at the middle: $splitAt \lfloor \frac{|xs|}{2} \rfloor\ xs$. However, it takes linear time to move to the middle point of a list (see chapter 1):

$$splitAt\ n\ xs = shift\ n\ [\ ]\ xs \tag{13.28}$$

Where:

$$\begin{array}{rcl} shift\ 0\ as\ bs & = & (as, bs) \\ shift\ n\ as\ (b{:}bs) & = & shift\ (n-1)\ (b{:}as)\ bs \end{array} \tag{13.29}$$

Because *halve* needn't keep the relative order among elements, we can simplify the implementation with odd-even split. There are same number of elements in odd and even positions, or they only differ by one. $halve = split\ [\ ]\ [\ ]$, where:

$$\begin{array}{rcl} split\ as\ bs\ [\ ] & = & (as, bs) \\ split\ as\ bs\ [x] & = & (x{:}as, bs) \\ split\ as\ bs\ (x{:}y{:}xs) & = & split\ (x{:}as)\ (y{:}bs)\ xs \end{array} \tag{13.30}$$

We can further simplify it with folding, as in below example, we add x to a every time, then swap $as \leftrightarrow bs$:

```
halve = foldr f ([], []) where
  f x (as, bs) = (bs, x : as)
```

13.3.1 Merge

Merge is demonstrated as figure 13.8. Consider two groups of kids, already ordered from short to tall. They need pass a gate, one kid per time. We arrange the first kid from each group to compare, the shorter one pass the gate. Repeat this till a group pass the gate, then the remaining kids pass the gate one by one.

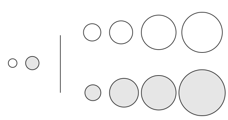


Figure 13.8: Merge

$$\begin{array}{rcl} merge\ [\ ]\ bs & = & bs \\ merge\ as\ [\ ] & = & as \\ merge\ (a{:}as)\ (b{:}bs) & = & \begin{cases} a < b : & a : merge\ as\ (b{:}bs) \\ \text{otherwise} : & b : merge\ (a{:}as)\ bs \end{cases} \end{array} \tag{13.31}$$

For array, we directly cut at the middle position, recursively sort two halves, then merge:

[2]For example in the standard library of Haskell, Python, and Java.

```
procedure Sort(A)
    n ← |A|
    if n > 1 then
        m ← ⌊n/2⌋
        X ← Copy-Array(A[1...m])
        Y ← Copy-Array(A[m + 1...n])
        Sort(X)
        Sort(Y)
        Merge(A, X, Y)
```

We allocated additional space of the same size of A because Merge is not in-pace. We repeatedly compare elements from X and Y, pick the less one to A. When either sub-array finish, we add all the remaining to A.

```
procedure Merge(A, X, Y)
    i ← 1, j ← 1, k ← 1
    m ← |X|, n ← |Y|
    while i ≤ m and j ≤ n do
        if X[i] < Y[j] then
            A[k] ← X[i]
            i ← i + 1
        else
            A[k] ← Y[j]
            j ← j + 1
        k ← k + 1
    while i ≤ m do
        A[k] ← X[i]
        k ← k + 1
        i ← i + 1
    while j ≤ n do
        A[k] ← Y[j]
        k ← k + 1
        j ← j + 1
```

13.3.2 Performance

Merge sort has two steps: partition and merge. We always halve the sequence. The partition tree is a balanced binary tree as shown in figure 13.2. The height is $O(\lg n)$, so as the recursion depth. The merge happens at every level, compares elements one by one from each sorted sub-sequence. Hence merge takes linear time. For sequence of length n, let $T(n)$ be the merge sort time, we have below recursive breakdown:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + cn = 2T(\frac{n}{2}) + cn \tag{13.32}$$

The time consists of three parts: sort the first half, sort the second half, each takes $T(\frac{n}{2})$ time; and merge in cn time, where c is a constant. Solving this equation gives $O(n \lg n)$ result. The other performance factor is space. Varies implementation differ a lot. The basic merge sort allocates the space of the same size as the array in each recursion, copies elements and sorts, then release the space. When reach to the deepest recursion, consume the largest space of $O(n \lg n)$.

Improvement

To simplify merge, we append ∞ to X and Y[3].

```
procedure Merge(A, X, Y)
    Append(X, ∞)
    Append(Y, ∞)
    i ← 1, j ← 1, n ← |A|
    for k ← from 1 to n do
        if X[i] < Y[j] then
            A[k] ← X[i]
            i ← i + 1
        else
            A[k] ← Y[j]
            j ← j + 1
```

It's expensive to allocate/release space repeatedly[2]. We can pre-allocate a work area of the same size as A. Reuse it during recursive merge, and finally release it.

```
procedure Sort(A)
    n ← |A|
    Sort'(A, Create-Array(n), 1, n)

procedure Sort'(A, B, l, u)
    if u − l > 0 then
        m ← ⌊(l + u)/2⌋
        Sort'(A, B, l, m)
        Sort'(A, B, m + 1, u)
        Merge'(A, B, l, m, u)
```

We need update Merge′ with the passed in work area:

```
procedure Merge'(A, B, l, m, u)
    i ← l, j ← m + 1, k ← l
    while i ≤ m and j ≤ u do
        if A[i] < A[j] then
            B[k] ← A[i]
            i ← i + 1
        else
            B[k] ← A[j]
            j ← j + 1
        k ← k + 1
    while i ≤ m do
        B[k] ← A[i]
        k ← k + 1
        i ← i + 1
    while j ≤ u do
        B[k] ← A[j]
        k ← k + 1
        j ← j + 1
    for i ← from l to u do                ▷ copy back
        A[i] ← B[i]
```

This implementation reduces the space from $O(n \lg n)$ to $O(n)$, improve performance

[3] $-\infty$ for descending order

20% to 25% for 100,000 numeric elements.

13.3.3 In-place merge sort

To avoid additional space, we consider how to reuse the array as the work area. As shown in figure 13.9, sub-array A and B are sorted, when merge in-place, the part before l are merged and ordered. If $A[l] < A[m]$, move l to right a step; otherwise if $A[l] \geq A[m]$, we need move $A[m]$ to merge result before l. We need shift all elements between l and m (including l) to right a step.

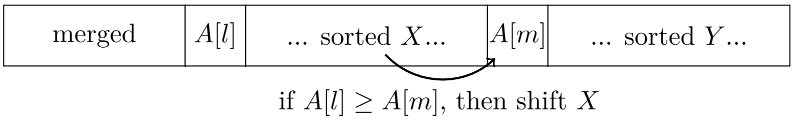


Figure 13.9: In-place shift and merge

```
procedure Merge(A, l, m, u)
    while l ≤ m ∧ m ≤ u do
        if A[l] < A[m] then
            l ← l + 1
        else
            x ← A[m]
            for i ← m down-to l + 1 do        ▷ Shift
                A[i] ← A[i − 1]
            A[l] ← x
```

However, the in-place shift and merge downgrades the performance to $O(n^2)$ time. Array shift takes linear time, proportion to the length of X. When sort a sub-array, our idea is to reuse the remaining part as the work area, and avoid overwriting the elements in it. When compare elements from sorted sub-array A and B, we chose the less one and store it in the work area, but we need exchange the element out to free up the cell. After merge, A and B together store the content of the original work area, as shown in figure 13.10.

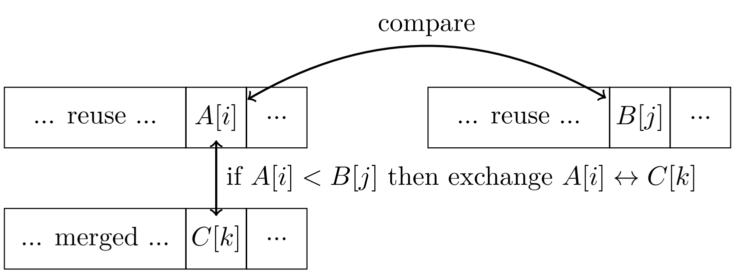


Figure 13.10: Merge and swap

The sorted array A, B, and work area C are all part of the array. We pass the start, end positions of A and B as ranges $[i, m)$, $[j, n)$[4]. The work area starts from k.

```
procedure Merge(A, [i, m), [j, n), k)
    while i < m and j < n do
        if A[i] < A[j] then
```

[4] range $[a, b)$ includes a, but excludes b.

```
            Exchange A[k] ↔ A[i]
            i ← i + 1
        else
            Exchange A[k] ↔ A[j]
            j ← j + 1
        k ← k + 1
    while i < m do
        Exchange A[k] ↔ A[i]
        i ← i + 1
        k ← k + 1
    while j < m do
        Exchange A[k] ↔ A[j]
        j ← j + 1
        k ← k + 1
```

The work area satisfies below two rules:

1. The work area has sufficient size to hold elements swapped in;
2. The work area can overlap with either sorted sub-arrays, but not overwrite any unmerged elements.

We can use half array as the work area to sort the other half, as shown in figure 13.11.

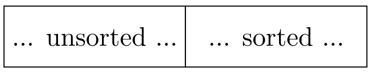


Figure 13.11: Merge and sort half array

We next sort further half of the work area (remaining $\frac{1}{4}$), as shown in figure 13.12. We must merge A ($\frac{1}{2}$ array) and B ($\frac{1}{4}$ array) later sometime. However, the work area can only hold $\frac{1}{4}$ array, insufficient for size of $A + B$.

Figure 13.12: Work area can't support merge A and B.

The second rule gives us an opportunity: arrange the work area overlapped with either sub-array, and only override the merged part. We first sort the second 1/2 of the work area, as the result, swap B to the first 1/2, the new work area is between A and B, as shown in the upper of figure 13.13. The work area is overlapped with A[74]. Consider two extremes:

1. $x < y$, for all x in B, y in A. After merge, contents of B and the work area are swapped (the size of B equals to the work area);
2. $y < x$, for all x in B, y in A. During merge, we repeatedly swap content of A and the work area. After half of A is swapped, we start overriding A. Fortunately, we only override the merged content. The right boundary of work area keep moving to the 3/4 of the array. After that, we start swap the content of B and the work area. Finally, the work area moves to the left side of the array, as shown in the bottom of figure 13.13 (b).

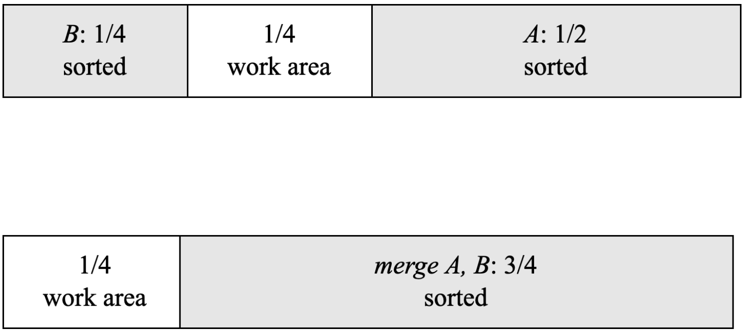


Figure 13.13: Merge A and B with the work area.

The other cases are between the above two extremes. The work area finally moves to the first 1/4 of the array. Repeat this, we always sort the second 1/2 of the work area, swap the result to the first 1/2, and keep the work area in the middle. We halve the work area every time $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, ...$ of the array, terminate when there is only one element left. We cal also switch to insert sort for the last few elements.

procedure SORT(A, l, u)
    if $u - l > 0$ **then**
        $m \leftarrow \lfloor \frac{l+u}{2} \rfloor$
        $w \leftarrow l + u - m$
        SORT'(A, l, m, w) ▷ sort half
        while $w - l > 1$ **do**
            $u' \leftarrow w$
            $w \leftarrow \lceil \frac{l+u'}{2} \rceil$ ▷ halve the work area
            SORT'(A, w, u', l) ▷ sort the remaining half
            MERGE$(A, [l, l + u' - w], [u', u], w)$
        for $i \leftarrow w$ down-to l **do** ▷ Switch to insert sort
            $j \leftarrow i$
            while $j \leq u$ and $A[j] < A[j-1]$ **do**
                EXCHANGE $A[j] \leftrightarrow A[j-1]$
                $j \leftarrow j + 1$

We round up the work area to ensure sufficient size, then pass the range and work area to MERGE. We next update SORT', which calls SORT to swap the work area and merged part.

procedure SORT'(A, l, u, w)
    if $u - l > 0$ **then**
        $m \leftarrow \lfloor \frac{l+u}{2} \rfloor$
        SORT(A, l, m)
        SORT$(A, m + 1, u)$
        MERGE$(A, [l, m), [m + 1, u), w)$
    else ▷ Swap elements to the work area
        while $l \leq u$ **do**
            EXCHANGE $A[l] \leftrightarrow A[w]$
            $l \leftarrow l + 1$
            $w \leftarrow w + 1$

This implementation needn't shift sub-array, it keeps reducing the unordered part:

$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, ...$, completes in $O(\lg n)$ steps. Every step sorts half of the remaining, then merge in linear time. Let the time to sort n elements be $T(n)$, we have the following recursive result:

$$T(n) = T(\frac{n}{2}) + c\frac{n}{2} + T(\frac{n}{4}) + c\frac{3n}{4} + T(\frac{n}{8}) + c\frac{7n}{8} + ... \tag{13.33}$$

For half elements, the time is:

$$T(\frac{n}{2}) = T(\frac{n}{4}) + c\frac{n}{4} + T(\frac{n}{8}) + c\frac{3n}{8} + T(\frac{n}{16}) + c\frac{7n}{16} + ... \tag{13.34}$$

Subtract (13.33) and (13.34):

$$T(n) - T(\frac{n}{2}) = T(\frac{n}{2}) + cn(\frac{1}{2} + \frac{1}{2} + ...)$$

Add $\frac{1}{2}$ total $\lg n$ times, hence:

$$T(n) = 2T(\frac{1}{2}) + \frac{c}{2} n \lg n$$

Apply telescope method, obtain the result $O(n \lg^2 n)$.

13.3.4 Nature merge sort

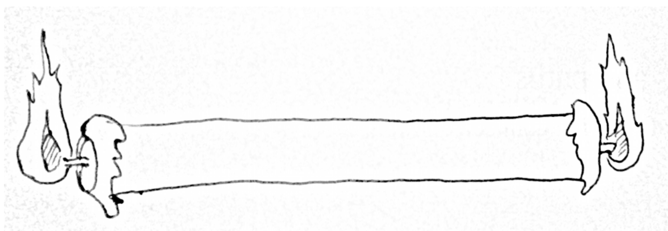


Figure 13.14: Burn from both ends

Knuth gives another implementation, called *nature merge sort*. It likes burning a candle from both ends[51]. For any sequence, one can always find a ordered segment from any position. Particularly, we can find such a segment from left end as shown in below table.

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 is the extreme case of a singleton segment, the second is less than the first; the third row is the other extreme that the segment extends to the right end, the whole sequence is ordered. Symmetrically, we can always find the ordered segment from right end. We can merge the two sorted segments, one from left, another from right. The advantage is to re-use the nature ordered sub-sequences for partition.

As shown in figure 13.15, we scan from both ends, find the two longest ordered segments respectively. Then merge them to the left of the work area. Next, we repeat to

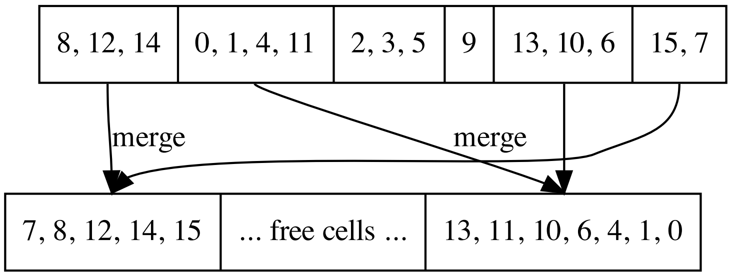


Figure 13.15: Nature merge sort

scan from left and right to center. This time, we merge the two segments for the right to left of the work area. We switch the merge direction right/left in-turns. After scan all elements and merge them to the work area, we swap the original array and the work area, then start a new round of bi-directional scan and merge, terminates when the ordered segment extends to cover the whole array. This implementation process the array from both directions based on nature ordering. We called it *nature two-way merge sort*. As shown in figure 13.16, elements before a and after d are scanned. We span the ordered segment $[a, b)$ to right, meanwhile, span $[c, d)$ to left. For the work area, elements before f and after f are merged (consist of multiple sub-sequences). In odd rounds, we merge $[a, b)$ and $[c, d)$ from f to right; in even rounds, we merge from r to left.

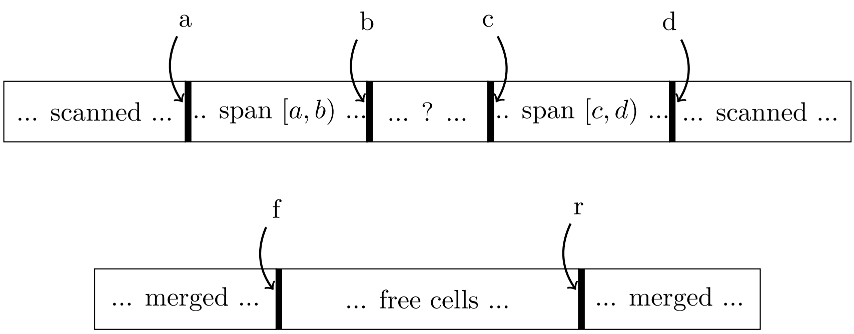


Figure 13.16: A status of nature merge sort

When sort starts, we allocate a work area with the same size of the array. a and b point to the left side, c and d point to the right side. f and r point to the two sides of the work area respectively.

```
function Sort(A)
    if |A| > 1 then
        n ← |A|
        B ← Create-Array(n)                     ▷ the work area
        loop
            [a, b) ← [1, 1)
            [c, d) ← [n + 1, n + 1)
            f ← 1, r ← n                         ▷ front, rear of the work area
            t ← 1                                ▷ even/odd round
            while b < c do                       ▷ elements yet to scan
                repeat                           ▷ Span [a, b)
```

```
                b ← b + 1
            until b ≥ c ∨ A[b] < A[b − 1]
            repeat                                    ▷ Span [c, d)
                c ← c − 1
            until c ≤ b ∨ A[c − 1] < A[c]
            if c < b then                             ▷ Avoid overlap
                c ← b
            if b − a ≥ n then     ▷ Terminates if [a, b) spans the whole array
                return A
            if t is odd then                          ▷ merge to front
                f ← Merge(A, [a, b), [c, d), B, f, 1)
            else                                      ▷ merge to rear
                r ← Merge(A, [a, b), [c, d), B, r, −1)
            a ← b, d ← c
            t ← t + 1
        Exchange A ↔ B                                ▷ Switch work area
    return A
```

We need pass the merge direction in:

```
function Merge(A, [a, b), [c, d), B, w, Δ)
    while a < b and c < d do
        if A[a] < A[d − 1] then
            B[w] ← A[a]
            a ← a + 1
        else
            B[w] ← A[d − 1]
            d ← d − 1
        w ← w + Δ
    while a < b do
        B[w] ← A[a]
        a ← a + 1
        w ← w + Δ
    while c < d do
        B[w] ← A[d − 1]
        d ← d − 1
        w ← w + Δ
    return w
```

The performance does not depend on how ordered the elements are. In the 'worst' case, the ordered sub-sequences are all singleton. After merge, the length of the new ordered sub-sequences are at least 2. Suppose we still encounter the 'worst' case in the second round, the merged sub-sequences have length at least 4, ... every round double the sub-sequence length, hence we need at most $O(\lg n)$ rounds. Because we can all elements every round, the total time is bound to $O(n \lg n)$. For list, we can't scan from tail back easily as array. A list consists multiple ordered sub-lists, we can merge them in pairs. It halves the sub-lists every round, and finally build the sorted result. We can define this as below (Curried form):

$$sort = sort' \circ group \tag{13.35}$$

Where *group* breaks the list into ordered sub-lists:

$$\begin{array}{rcl} group\ [\,] &=& [[\,]] \\ group\ [x] &=& [[x]] \\ group\ (x{:}y{:}xs) &=& \begin{cases} x < y: & (x{:}g){:}gs, \text{where}: (g{:}gs) = group\ (y{:}xs) \\ \text{otherwise}: & [x]{:}g{:}gs \end{cases} \end{array} \tag{13.36}$$

$$\begin{array}{rcl} sort'\ [\,] &=& [\,] \\ sort'\ [g] &=& g \\ sort'\ gs &=& sort'\ (mergePairs\ gs) \end{array} \tag{13.37}$$

Where *mergePairs* is defined as:

$$\begin{array}{rcl} mergePairs\ (g_1{:}g_2{:}gs) &=& merge\ g_1\ g_2 : mergePairs\ gs \\ mergePairs\ gs &=& gs \end{array} \tag{13.38}$$

Alternatively, we can define *sort′* as fold:

$$sort' = foldr\ merge\ [\,] \tag{13.39}$$

Exercise 13.2

1. Is the performance of *mergePairs* and folded merge same? If yes, prove it, if not, which one is faster?

13.3.5 Bottom-up merge sort

We can develop the bottom-up merge sort from the above performance analysis for nature merge sort. First wrap all elements as n singleton sub-lists. Then merge them in pairs to obtain $\frac{n}{2}$ ordered sub-lists of length 2; If n is odd, there remains a single list. Repeat this paired merge to the sort all the list. Knuth called it 'straight two-way merge sort'[51], as shown in figure 13.17.

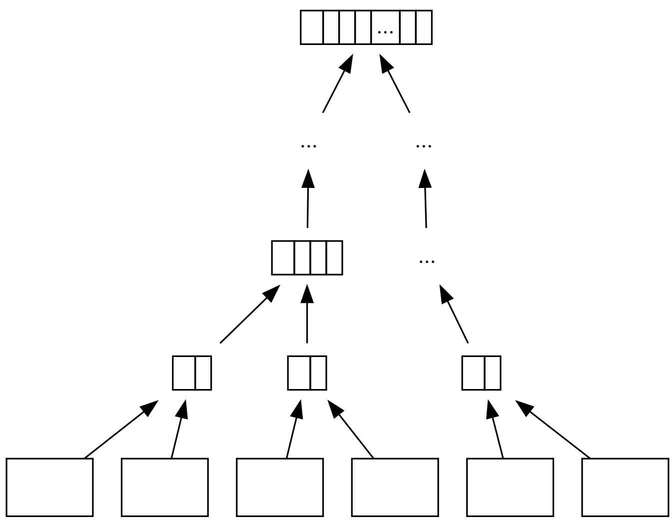


Figure 13.17: Bottom-up merge sort

We needn't partition the list. When start, convert $[x_1, x_2, ..., x_n]$ to $[[x_1], [x_2], ..., [x_n]]$, then apply paired merge:

$$sort = sort' \circ map(x \mapsto [x]) \tag{13.40}$$

We reuse the *mergePairs* defined for nature merge sort, terminates when consolidate to one list[3]. The bottom up sort is similar to the nature merge sort, different only in partition method. It can be deduced from nature merge sort as a special case (the 'worst' case). Nature merge sort always span the ordered sub-sequence as long as possible; while the bottom up merge sort only span the length to 1. From the tail recursive implementation, we can eliminate the recursion and convert it to iterative way.

```
function Sort(A)
    n ← |A|
    B ← Create-Array(n)
    for i from 1 to n do
        B[i] = [A[i]]
    while n > 1 do
        for i ← from 1 to ⌊n/2⌋ do
            B[i] ← Merge(B[2i − 1], B[2i])
        if Odd(n) then
            B[⌈n/2⌉] ← B[n]
        n ← ⌈n/2⌉
    if B = [ ] then
        return [ ]
    return B[1]
```

Exercise 13.3

1. Implement the bottom-up merge sort with fold

13.4 Parallelism

In quick sort implementation, we can parallel sorting the two sub-sequences after partition. Similarly, to parallel merge sort. Actually, we don't limit by two concurrent tasks, but divide into p sub-sequences, where p is the number of processors. Ideally, if we can achieve sorting in T' time with parallelism, where $O(n \lg n) = pT'$, we say it's *linear speed up*, and the algorithm is parallel optimal. However, it is not parallel optimal by choosing $p - 1$ pivots, and partition the sequence into p parts for quick sort. The bottleneck happens in the divide phase, that can only achieve in $O(n)$ time. While, the bottleneck is the merge phase for parallel merge sort. Both need specific design to speed up. Basically, the divide and conquer nature makes merge sort and quick sort relative easy for parallelism. Richard Cole developed parallel merge sort achieved $O(\lg n)$ performance with n processors in 1986[76]. Parallelism is a big and complex topic out of the elementary scope[76] [77].

13.5 Summary

This chapter gives two popular divide and conquer sort algorithms: quick sort and merge sort. Both achieved the best performance of $O(n \lg n)$ for comparison based sort. Sedgewick quoted quick sort as the greatest algorithm developed in the 20th century.

Many programming environments provide sort tool based on it. Merge sort is a powerful tool when handling sequence of complex entities, or not persisted in array[5]. Quick sort performs well in most cases with fewer swaps than other methods. However, swap is not suitable for linked-list, while merge sort is. It costs constant spaces and the performance is guaranteed for all cases. Quick sort has advantage for vector storage like arrays, because it needn't extra work area and can sort in-place. This is a valuable feature particularly in embedded system where memory is limited. In-place merging is till an active research area.

We can considered quick sort as the optimized tree sort. Similarly, we can also deduce merge sort from tree sort[75]. We can categorize sort algorithms in different ways[51], for example, the implementations of partition and merge[72]. Quick sort is easy to merge, because all the elements in one sub-sequence are not greater than the other. Merge is equivalent to concatenation. On the other hand, in merge sort, it's more complex than quick sort, but it is easy to partition no matter we cut at the middle, even-odd split, nature split, or bottom up split. While it's more difficult to achieve perfect partition in quick sort or completely avoid the worst case no matter with median-of-three pivot, random quick sort, or ternary quick sort.

As of this chapter, we've seen the elementary sort algorithms, including insert sort, tree sort, selection sort, heap sort, quick sort, and merge sort. Sort is an important domain in computer algorithm design. People are facing the 'big data' challenge when I wrote this chapter. It becomes routine to sort hundreds of Gigabytes with limited resources and time.

Exercise 13.4

1. Build a binary search tree from a sequence using the idea of merge sort.

13.6 Appendix: Example programs

In-place partition:

```
Int partition([K] xs, Int l, Int u) {
    for (Int pivot = l, Int r = l + 1; r < u; r = r + 1) {
        if xs[pivot] ≥ xs[r] {
            l = l + 1
            swap(xs[l], xs[r])
        }
    }
    swap(xs[pivot], xs[l])
    return l + 1
}

Void sort([K] xs, Int l, Int u) {
    if l < u {
        Int m = partition(xs, l, u)
        sort(xs, l, m - 1)
        sort(xs, m, u)
    }
}
```

Bi-directional scan:

```
Void sort([K] xs, Int l, Int u) {
    if l < u - 1 {
        Int pivot = l, Int i = l, Int j = u
        loop {
```

[5]In practice, most are kind of hybrid sort, for example, fallback to insert sort for small sequence.

```
            while i < u and xs[i] < xs[pivot] {
                i = i + 1
            }
            while j ≥ l and xs[pivot] < xs[j] {
                j = j - 1
            }
            if j < i then break
            swap(xs[i], xs[j])
        }
        swap(xs[pivot], xs[j])
        sort(xs, l, j)
        sort(xs, i, u)
    }
}
```

Merge sort:

```
[K] sort([K] xs) {
    Int n = length(xs)
    if n > 1 {
        var ys = sort(xs[0 ... n/2 - 1])
        var zs = sort(xs[n/2 ...])
        xs = merge(xs, ys, zs)
    }
    return xs
}

[K] merge([K] xs, [K] ys, [K] zs) {
    Int i = 0
    while ys ≠ [] and zs ≠ [] {
        xs[i] = if ys[0] < zs[0] then pop(ys) else pop(zs)
        i = i + 1
    }
    xs[i...] = if ys ≠[] then ys else zs
    return xs
}
```

Merge sort with work area:

```
Void sort([K] xs) = msort(xs, copy(xs), 0, length(xs))

Void msort([K] xs, [K] ys, Int l, Int u) {
    if (u - l > 1) {
        Int m = l + (u - l) / 2
        msort(xs, ys, l, m)
        msort(xs, ys, m, u)
        merge(xs, ys, l, m, u)
    }
}

Void merge([K] xs, [K] ys, Int l, Int m, Int u) {
    Int i = l, Int k = l; Int j = m
    while i < m and j < u {
        ys[k++] = if xs[i] < xs[j] then xs[i++] else xs[j++]
    }
    while i < m {
        ys[k++] = xs[i++]
    }
    while j < u {
        ys[k++] = xs[j++]
    }
    while l < u {
        xs[l] = ys[l]
        l++
    }
}
```

In-place merge sort:

```
Void merge([K] xs, (Int i, Int m), (Int j, Int n), Int w) {
    while i < m and j < n {
        swap(xs, w++, if xs[i] < xs[j] then i++ else j++)
    }
    while i < m {
        swap(xs, w++, i++)
    }
    while j < n {
        swap(xs, w++, j++)
    }
}

Void wsort([K] xs, (Int l, Int u), Int w) {
    if u - l > 1 {
        Int m = l + (u - l) / 2
        imsort(xs, l, m)
        imsort(xs, m, u)
        merge(xs, (l, m), (m, u), w)
    }
    else {
        while l < u { swap(xs, l++, w++) }
    }
}

Void imsort([K] xs, Int l, Int u) {
    if u - l > 1 {
        Int m = l + (u - l) / 2
        Int w = l + u - m
        wsort(xs, l, m, w)
        while w - l > 2 {
            Int n = w
            w = l + (n - l + 1) / 2;
            wsort(xs, w, n, l);
            merge(xs, (l, l + n - w), (n, u), w);
        }
        for Int n = w; n > l; --n {
            for Int m = n; m < u and xs[m] < xs[m-1]; m++ {
                swap(xs, m, m - 1)
            }
        }
    }
}
```

Iterative bottom up merge sort:

```
[K] sort([K] xs) {
    var ys = [[x] | x in xs]
    while length(ys) > 1 {
        ys += merge(pop(ys), pop(ys))
    }
    return if ys == [] then [] else pop(ys)
}

[K] merge([K] xs, [K] ys) {
    [K] zs = []
    while xs ≠ [] and ys ≠[] {
        zs += if xs[0] < ys[0] then pop(xs) else pop(ys)
    }
    return zs ++ (if xs ≠[] then xs else ys)
}
```

Chapter 14

Solution search

We can search the solution for many problem with computer. For example, we build robot to search and pick the right gadget in assembly lane; we develop car navigator to search the map for the best route. We make smart phone application to search the best shopping plan. This chapter surveys the elementary lookup, matching, and solution search algorithms.

14.1 k selection problem

A selection algorithm is to find the k-th smallest (or largest) element in a list or array. The ordering is abstract, we use $\leq$ for example. The simplest method is to repeatedly find the minimum for k times. It takes $O(n)$ time to find the minimum, the total performance is bound to $O(kn)$. We can also use heap to update, access the top element in $O(\lg n)$ time, hence find the k-th element in $O(k \lg n)$ time.

$$top\ k\ xs = find\ k\ (heapify\ xs) \tag{14.1}$$

Or in Curried form:

$$top\ k\ = (find\ k) \circ heapify \tag{14.2}$$

Where:

$$\begin{array}{rcl} find\ 0 & = & top \\ find\ k & = & (find\ (k-1)) \circ pop \end{array} \tag{14.3}$$

We can do it even better. Apply the divide and conquer approach, split the elements into A and B, where all elements in A isn't greater ($\leq$) than any one B. Let $m = |A|$ be the size of A, compare m and k:

1. If $k < m$, the k-th element is in A, drop B and recursively search in A;

2. If $m < k$, the k-th element is in B, drop A and recursively search the $(k-m)$-th element in B.

In ideal case, the split is balanced (the sizes of A and B almost same). We halve the size every time, the performance is $O(n + n/2 + n/4 + ...) = O(n)$. Reuse the *part* function in quick sort (chapter 13), select an element (for example the first) as the pivot p. Collect all elements $\leq p$ in A, the rest in B. If $m = k - 1$, then p is the k-th element; otherwise, we recursively find in A or B.

$$
top\ k\ (x{:}xs) = \begin{cases} m = k - 1: & x, \text{where}: m = |A|, (A, B) = part\ (\leq x)\ xs \\ m < k - 1: & top\ (k - m - 1)\ B \\ \text{otherwise}: & top\ k\ A \end{cases} \tag{14.4}
$$

Same as the quick sort algorithm, the worst case happens when the partition is always unbalanced. The performance downgrades to $O(kn)$ or $O((n-k)n)$. In average case, we can find the k-th element in linear time. All engineering practices in quick sort are applicable too, like the 'media of three'[1], and randomly select the pivot:

function TOP(k, A, l, u)
  EXCHANGE $A[l] \leftrightarrow A[$ RANDOM(l, u) $]$ ▷ Randomly select in $[l, u]$
  $p \leftarrow$ PARTITION(A, l, u)
  if $p - l + 1 = k$ **then**
    return $A[p]$
  if $k < p - l + 1$ **then**
    return TOP$(k, A, l, p - 1)$
  return TOP$(k - p + l - 1, A, p + 1, u)$

We can change to return all the top k elements (in arbitrary order), as below example program:

```
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
```

14.2 Binary search

My high school teacher once played a 'math magic'. He asked a student pick a number from 0 to 1000 in mind. He asked 10 questions, then figured out that number from the yes or no answers from the student. For example: is it even? is it prime? can it be divided by 3? and etc. If halves the numbers with every question, one can find any number within 1000 because $2^{10} = 1024 > 1000$. The question whether it is even, perfectly halves numbers[2]. This kind of games becomes not so interesting when a player guess the price in TV programs: 1000, high, 50, low, 750, low, 890, low, 990, correct!. The player applied the *binary search* method. To find x in an ordered sequence A, one firstly tries the middle point y. Done if $x = y$; if $x < y$, then drop the second half of A as it's ordered; otherwise drop the first half. When it becomes $A = [\ ]$, then x doesn't exist. A need be ordered, I often see people are struggled with unordered data, confusing why the binary search does not work. 'Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky' said by Donald Knuth. Jon Bentley said most binary search implementations had error, including the one he given in '*Programming pearls*'. He

[1] Blum, Floyd, Pratt, Rivest, and Tarjan developed a linear timer algorithm in 1973 [4] [81]. Split the elements into groups, each has 5 elements at most. It gives $n/5$ medians. Repeat this to pick the median of median.

[2] There's a 'mind reading' game in social network. One thinks about a person in mind. The AI robot asks 16 questions, and tells who that person is from the yes or no answers

corrected the error after two decades[2]. Below is the binary search definition, where the lower and upper bounds of A is l and u (exclude u).

$$
bsearch\ x\ A\ (l, u) = \begin{cases}
u < l : & Nothing \\
x = A[m] : & m, \text{where} : m = l + \lfloor \frac{u - l}{2} \rfloor \\
x < A[m] : & bsearch\ x\ A\ (l, m - 1) \\
\text{otherwise} : & bsearch\ x\ A\ (m + 1, u)
\end{cases} \tag{14.5}
$$

We can eliminate the recursion, implement with a loop.

```
function Binary-Search(x, A, l, u)
    while l < u do
        m ← l + ⌊(u − l)/2⌋                    ▷ avoid ⌊(l + u)/2⌋ overflow
        if A[m] = x then
            return m
        if x < A[m] then
            u ← m − 1
        else
            l ← m + 1
    Not found
```

The performance of binary search is bound to $O(\lg n)$ because it halves A every time. We can extend it to solve equation of monotone functions, for example $a^x = y$, where $a \leq y$, a and y are nature numbers. To find the integral x, we can exhaust $a^0, a^1, a^2, ...$, till $a^i = y$ or $a^i < y < a^{i+1}$ (no solution). If a and x are big numbers, it's expensive to compute a^x[3]. Let's apply binary search. As $a^y \geq y$, we search in range $[0, 1, ..., y]$. Function $f(x) = a^x$ is monotone, fix x, we examine the middle point: $x_m = \lfloor \frac{0 + y}{2} \rfloor$. If $a^{x_m} = y$, then x_m is the solution; if $a^m < y$, we discard the range before x_m; otherwise discard the range after x_m. Both halve the search range. When the range becomes empty, it means no solution. Below are the binary search implementation. We denote the monotone function as f, call $bsearch\ f\ y\ (0, y)$, where $f(x) = a^x$. We only need compute $f(x)$ for $O(\lg y)$ times, better than the exhaustive search.

$$
bsearch\ f\ y\ (l, u) = \begin{cases}
u < l : & Nothing \\
f(m) = y : & m, \text{where} : m = \lfloor \frac{l + u}{2} \rfloor \\
f(m) < y : & bsearch\ f\ y\ (m + 1, u) \\
f(m) > y : & bsearch\ f\ y\ (l, m - 1)
\end{cases} \tag{14.6}
$$

14.2.1 2D search

We can extend binary search to 2D or even higher dimension. Let the matrix M of size $m \times n$. Its elements in each row, column are ascending nature numbers as shown in figure 14.1. How to locate all elements equal to z? i.e. find all locations of (i, j), such that $M_{i,j} = z$.

$$
[(x, y) | x \leftarrow [1, 2, ..., m], y \leftarrow [1, 2, ..., n], M_{x,y} = z] \tag{14.7}
$$

Richard Bird used to interview students with this question[1]. Those who had programming experience at school tended to apply binary search. But it was easy to get stuck. One often checks the middle point $M_{\frac{m}{2}, \frac{n}{2}}$. If it is less than z, then drop the

[3] One can reuse the result of a^n to compute $a^{n+1} = aa^n$. We consider generic monotone $f(n)$.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & ... \\ 2 & 4 & 5 & 6 & ... \\ 3 & 5 & 7 & 8 & ... \\ 4 & 6 & 8 & 9 & ... \\ ... \end{bmatrix}$$

Figure 14.1: Each row, column is ascending.

top-left rectangle; if greater than z then drop the bottom-right rectangle, as shown in figure 14.2, discard the shaded rectangle. Both cases lead to a L-shape search area, where we can't apply recursive search directly. We define the 2D search as: given $f(x, y)$, search integer solution (x, y), such that $f(x, y) = z$ in an area. The matrix search can be specialized as below:

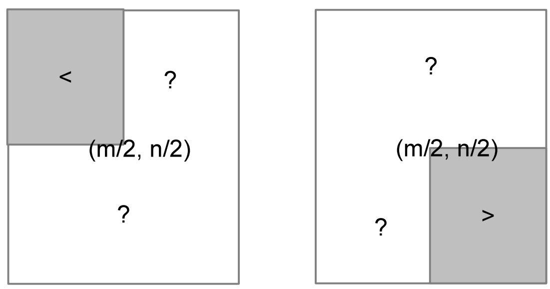


Figure 14.2: Left: the middle point $< z$, all shaded rectangle $< z$; Right: the middle point $> z$, all shaded rectangle $> z$.

$$f(x, y) = \begin{cases} 1 \leq x \leq m, 1 \leq y \leq n : & M_{x,y} \\ \text{otherwise} : & -1 \end{cases}$$

For monotone function $f(x, y)$, e.g., $f(x, y) = x^a = y^b$, where a, b are nature numbers, the effective solution is search from the top-left, but not bottom-left[82]. As shown in figure 14.3, start from $(0, z)$, for each location (p, q), compare $f(p, q)$ and z:

1. If $f(p, q) < z$, since f is monotone increasing, $f(p, y) < z$ for all $0 \leq y < q$. We drop all points in the vertical line segment (red);

2. If $f(p, q) > z$, then $f(x, q) > z$ for all $p < x \leq z$. We drop all points in the horizontal line segment (blue);

3. If $f(p, q) = z$, then (p, q) is a solution. We drop both line segments.

We line by line reduce the search rectangle. Every time drop a row, or a column, or both.

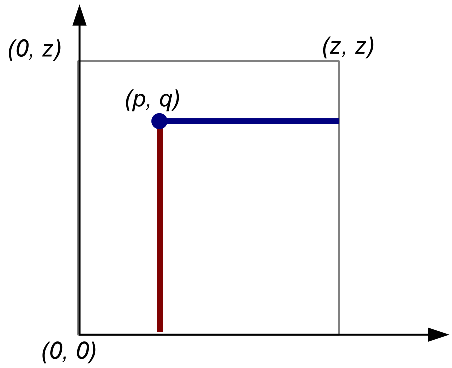


Figure 14.3: Search from top-left.

Define *search* function, and pass the top-left corner: $search(f, z, 0, z)$

$$search\ f\ z\ p\ q\ = \begin{cases} p > z 或 q < 0 : & [\,] \\ f(p,q) < z : & search\ f\ z\ (p+1)\ q \\ f(p,q) > z : & search\ f\ z\ p\ (q-1) \\ f(p,q) = z : & (p,q) : search\ f\ z\ (p+1)\ (q-1) \end{cases} \tag{14.8}$$

Every time, at least one of p and q advance towards the bottom or right by one step. It needs at most $2(z+1)$ steps to complete. There are three best cases: (1) both p and q advance a step a time, in total $z+1$ steps; (2) move to the right horizontally till p exceeds z; (3) move down vertically till q becomes negative. Figure 14.4 gives these cases. In 14.4 (a), all points in the diagonal line $(x, z-x)$ satisfy $f(x, z-x) = z$. It takes total $z+1$ steps to reach $(z, 0)$; in (b), all points in the top horizontal line (x, z) satisfy $f(x, z) < z$. It terminates after $z+1$ steps; in (c), all points in the left vertical line $(0, x)$ satisfy $f(0, x) > z$. It terminates after $z+1$ steps; (d) is the worst case. If project all the horizontal sections in the search path to x axis, all the vertical sections to y axis, it gives the total steps of $2(z+1)$. This method improved the performance of exhaustive search from $O(z^2)$ to $O(z)$.

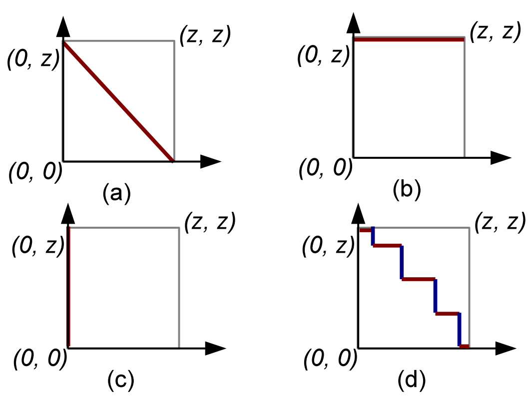


Figure 14.4: The best and worst cases.

This algorithm is called ‘saddle back’ search. The plot image of f has the smallest bottom-left and the largest top-right. It looks like a saddle with two wings as shown in figure 14.5. The search rectangle is $(0, z) - (z, 0)$, we can further reduce it. f is monotone increasing, we can find the maximum m along y axis, satisfying $f(0, m) \leq z$; find the maximum n along x axis, satisfying $f(n, 0) \leq z$. Reduce the search rectangle to $(0, m) - (n, 0)$, as shown in figure 14.6.

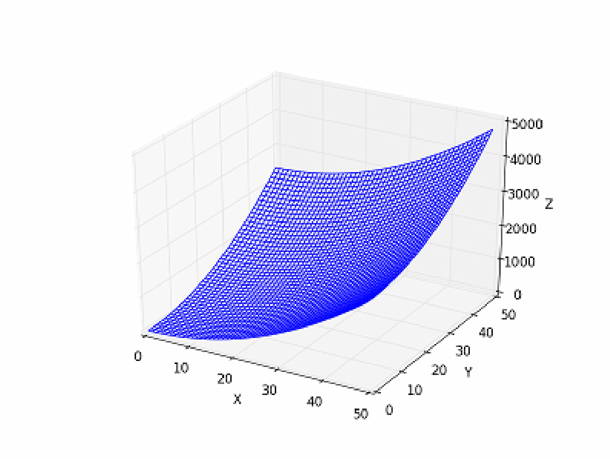


Figure 14.5: Plot of $f(x, y) = x^2 + y^2$.

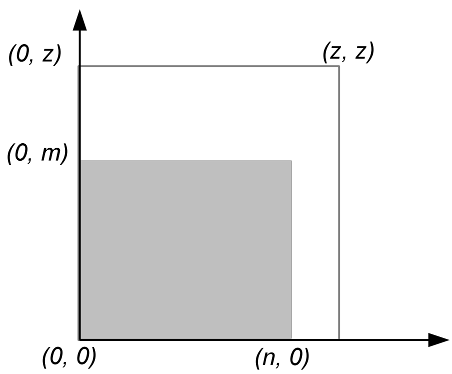


Figure 14.6: Reduced search rectangle.

$$
\begin{array}{rl}
m & = max(\{y | 0 \leq y \leq z, f(0, y) \leq z\}) \\
n & = max(\{x | 0 \leq x \leq z, f(x, 0) \leq z\})
\end{array}
\tag{14.9}
$$

We can apply binary search to find m, n (fix $x = 0$ to search m, fix $y = 0$ to search n). Modify (14.6), search $l \leq x \leq u$ satisfying $f(x) \leq y < f(x + 1)$.

$$
bsearch\ f\ y\ (l, u) = \begin{cases}
u \leq l : & l \\
f(m) \leq y < f(m+1) : & m, \text{where} : m = \lfloor \frac{l+u}{2} \rfloor \\
f(m) \leq y : & bsearch\ f\ y\ (m+1, u) \\
f(m) > y : & bsearch\ f\ y\ (l, m-1)
\end{cases}
\tag{14.10}
$$

Then determine m, n with binary search:

$$\begin{cases} m & = bsearch\ (y \mapsto f(0, y))\ z\ (0, z) \\ n & = bsearch\ (x \mapsto f(x, 0))\ z\ (0, z) \end{cases} \tag{14.11}$$

Finally, apply saddle back search in this smaller rectangle: $solve(f, z) = search(f, z, 0, \boldsymbol{m})$

$$search\ f\ z\ p\ q\ = \begin{cases} p > \boldsymbol{n} \text{or} q < 0 : & [\,] \\ f(p, q) < z : & search\ f\ z\ (p+1)\ q \\ f(p, q) > z : & search\ f\ z\ p\ (q-1) \\ f(p, q) = z : & (p, q) : search\ f\ z\ (p+1)\ (q-1) \end{cases} \tag{14.12}$$

We apply two rounds of binary search to find m, n, each round compute f for $O(\lg z)$ times; The saddle back search compute f for $O(m + n)$ times in the worst case; it's $O(\min(m, n))$ in the best case. Below table gives the total performance. For functions like $f(x, y) = x^a + y^b$, where a, b are nature numbers, the boundary m, n are very small. The total performance is close to $O(\lg z)$.

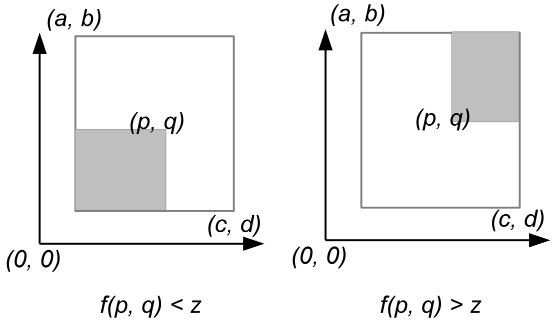
	compute f
worst	$2 \log z + m + n$
best	$2 \log z + \min(m, n)$

As shown in figure 14.7, for a point (p, q) in rectangle $(a, b) - (c, d)$, if $f(p, q) \neq z$, we can only discard the shaded part ($\leq 1/4$). If $f(p, q) = z$, we can discard the bottom-left, top-right parts, and all points in row p and column q since f is monotone. Hence reduced the search rectangle by 1/2. To find the point satisfying $f(p, q) = z$, we apply binary search along the horizontal or vertical central line. Because the performance is bound to $O(\lg |L|)$ for line L, we chose the shorter central line as shown in figure 14.8.

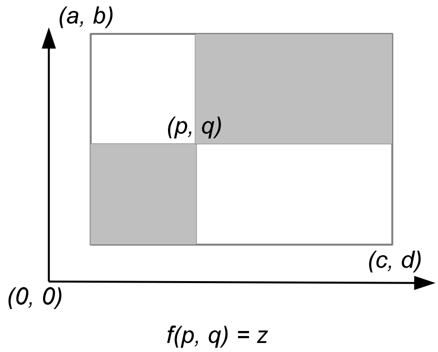
If there is no point satisfying $f(p, q) = z$, we find a point, such that $f(p, q) < z < f(p+1, q)$ in the horizontal central line ($f(p, q) < z < f(p, q+1)$ for vertical central line). We can't discard all points in row p and column q. In summary, we apply binary search along horizontal central line for the point: $f(p, q) \leq z < f(p+1, q)$; or search the vertical central line for the point: $f(p, q) \leq z < f(p, q+1)$. If all points in the line segment are $f(p, q) < z$, then return the upper bound; if all are $f(p, q) > z$, then return the lower bound. We can discard half side in this case. Below is the improved saddle back search:

1. Apply binary search along the x, y axes for the search rectangle $(0, m) - (n, 0)$;
2. For rectangle $(a, b) - (c, d)$, if the height $>$ width, apply binary search along the horizontal central line; otherwise search along the vertical central line for the point (p, q);
3. If $f(p, q) = z$, it is a solution. Recursively search rectangles $(a, b) - (p-1, q+1)$ and $(p+1, q-1) - (c, d)$;
4. If $f(p, q) \neq z$, recursively search the two rectangles and a line section, either $(p, q+1) - (p, b)$ in figure 14.9 (a); or $(p+1, q) - (c, q)$ in figure 14.9 (b).

$$search\ (a, b)\ (c, d) = \begin{cases} c < a \text{or} d < b : & [\,] \\ c - a < b - d : & csearch \\ \text{otherwise} : & rsearch \end{cases} \tag{14.13}$$



(a) If $f(p, q) \neq z$, we can only drop the shaded area, the remaining is a 'L' shape.



(b) If $f(p, q) = z$, we can drop 1/2 rectangle.

Figure 14.7: Reduce the search rectangle.

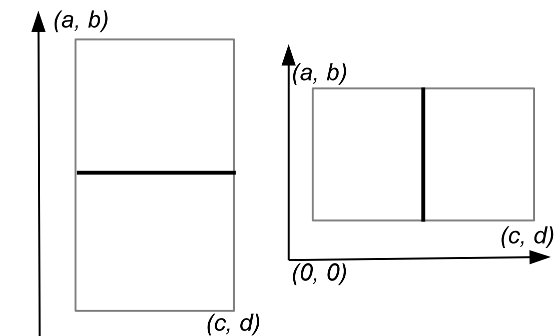


Figure 14.8: Chose the shorter center line.

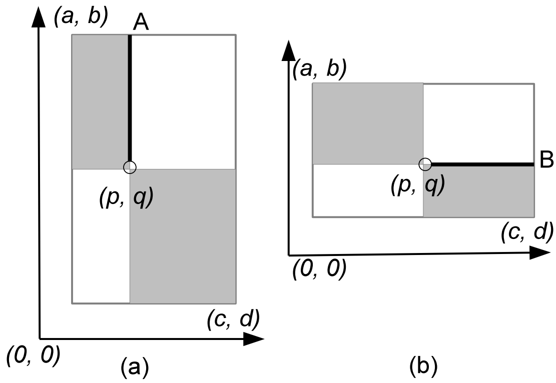


Figure 14.9: Recursively search the shaded parts, include the bold line if $f(p, q) \neq z$.

Where *csearch* apply binary search to the horizontal central line for point (p, q), such that $f(p, q) \leq z < f(p+1, q)$, as shown in figure 14.9 (a). If all function values are greater than z, then return the lower bound $(a, \lfloor \frac{b+d}{2} \rfloor)$. Drop the above side (include the central line) as shown in figure 14.10 (a).

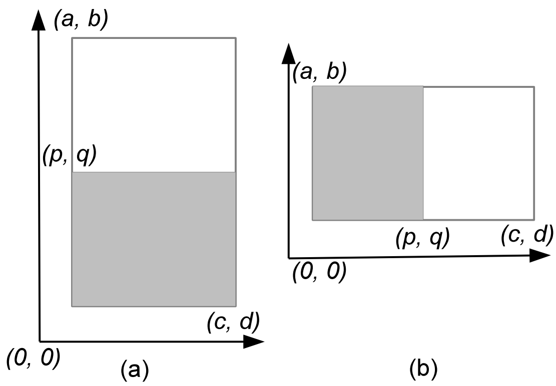


Figure 14.10: Special case.

Let

$$\begin{cases} q = \lfloor \dfrac{b+d}{2} \rfloor \\ p = bsearch\ (x \mapsto f(x, q))\ z\ (a, c) \end{cases}$$

$$csearch = \begin{cases} f(p, q) > z: & search\ (p, q-1)\ (c, d) \\ f(p, q) = z: & search\ (a, b)\ (p-1, q+1) \mathbin{+\!\!+} [(p, q)] \mathbin{+\!\!+} search\ (p+1, q-1)\ (c, d) \\ f(p, q) < z: & search\ (a, b)\ (p, q+1) \mathbin{+\!\!+} search\ (p+1, q-1)\ (c, d) \end{cases} \tag{14.14}$$

Function *rsearch* is symmetric along the vertical central line. Below example program implements the improved saddle back search:

```
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)
```

```
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)
```

As we halve the rectangle every time, we search $O(\lg(mn))$ rounds. We apply binary search along the central line for (p, q), compute f for $O(\lg(\min(m, n)))$ times. Let the time be $T(m, n)$ when search $m \times n$ rectangle. We have the following recursive equation:

$$T(m, n) = \lg(min(m, n)) + 2T(\frac{m}{2}, \frac{n}{2}) \tag{14.15}$$

Suppose $m = 2^i > n = 2^j$, use telescope method:

$$\begin{array}{rl} T(2^i, 2^j) & = j + 2T(2^{i-1}, 2^{j-1}) \\ & = \sum_{k=0}^{i-1} 2^k (j - k) \\ & = O(2^i (j - i)) \\ & = O(m \lg(n/m)) \end{array} \tag{14.16}$$

Richard Bird proved this is asymptotically optimal by a lower bound of searching a given value in $m \times n$ rectangle[1].

Exercise 14.1

1. Prove the performance of k-selection problem is $O(n)$ in average (refer to the quick sort performance analysis).
2. To find the top k element in A, we can search $x = \max(take\ k\ A), y = \min(drop\ k\ A)$. If $x < y$, then the first k elements in A is the answer; otherwise, we partition the first k elements with x, partition the rest with y, then recursively find in sub-sequence $[a|a \leftarrow A, x < a < y]$ for the top k' elements, where $k' = k - |[a|a \leftarrow A, a \leq x]|$. Implement this solution, and evaluate its performance.
3. Find the median of two sorted arrays A and B in $O(\lg(m + n))$ time, where $m = |A|, n = |B|$. The median x is defined as $||\{a \leq x : a \in A\}| + |\{b \leq x : b \in B\}| - |\{a > x : a \in A\}| - |\{b > x : b \in B\}|| \leq 1$.
4. For the saddle back search, eliminate recursion, implement it in loops to update the boundary.
5. For 2D search, let the bottom-left be the minimum, the top-right be the maximum. if z is less than the minimum or greater than the maximum, then no solution; otherwise cut the rectangle into 4 parts with a horizontal line and a vertical line crossed at the center. then recursive search in these 4 small rectangles. Implement this solution and evaluate its performance.

14.3 The majority number

People often vote and use computer to count the result. Suppose the candidate wins if and only if get more than half votes. From the votes sequence A, B, A, C, B, B, D, ..., how to find the winner efficiently? We can use a map to count the result (see chapter 2)[4].

```
Optional<T> majority([T] xs) {
    Map<T, Int> m
    for var x in xs {
        if x in m then m[x]++ else mx[x] = 0
    }
    var (r, v) = (Optional<T>.Nothing, length(xs) / 2 - 1)
    for var (x, c) in m {
      if c > v then (r, v) = (Optional.of(x), c)
    }
    return r
}
```

We can implement the map with the red-black tree or hash table. For m candidates, n votes, below table gives the performance:

map	time	space
tree	$O(n \lg m)$	$O(m)$
hash	$O(n)$	at least $O(m)$

Define the element occurs over 50% as 'majority'. Boyer and Moore developed an algorithm in 1980, which picks the majority element in one scan if there is. The algorithm needs constant space[83]. There is at most 1 majority, if repeat dropping two different elements till all remaining ones are same. If the majority exists, then it is the remaining. Start from the first vote, let the candidate be the winner so far with point 1. If the next one votes the same candidate, then add the winner point by 1, otherwise deduce by 1. The candidate won't be the winner when the point reduces to 0. We pick the candidate of the next vote as the new winner and go on. As shown in below table, if there exists majority m, then other candidate can't beat m. Otherwise if the majority doesn't exist (invalid vote result, no winner), then we need discard the recorded 'winner'. We need another scan to valid the winner.

winner	count	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**

[4]There is a probabilistic sub-linear space counting algorithm published in 2004, named as 'Count-min sketch'[84].

$$\begin{array}{rcl} maj\ [\,] & = & \varnothing \\ maj\ (x{:}xs) & = & scan\ (x, 1)\ xs \end{array} \tag{14.17}$$

Where *scan* is defined as:

$$\begin{array}{rcl} scan\ (m, v)\ [\,] & = & m \\ scan\ (m, v)\ (x{:}xs) & = & \begin{cases} m = x : & scan\ (m, v+1)\ xs \\ v = 0 : & scan\ (x, 1)\ xs \\ \text{otherwise} : & scan\ (m, v-1)\ xs \end{cases} \end{array} \tag{14.18}$$

We can also implement with fold:$maj = foldr\ f\ (\varnothing, 0)$, where:

$$f\ x\ (m, v) = \begin{cases} x = m : & (m, v+1) \\ v = 0 : & (x, 1) \\ \text{otherwise} : & (m, v-1) \end{cases} \tag{14.19}$$

Finally, we need verify the winner is the true majority element:

$$verify\ m = \text{if } 2|filter\ (= m)\ xs| > |xs| \text{ then } Just\ m \text{ else } \varnothing \tag{14.20}$$

Below is the corresponding iterative implementation:

```
function MAJORITY(A)
    c ← 0, m ← ∅
    for each a in A do
        if c = 0 then
            m ← a
        if a = m then
            c ← c + 1
        else
            c ← c − 1
    c ← 0
    for each a in A do
        if a = m then
            c ← c + 1
    if c > %50|A| then
        return x
    else
        return ∅
```

Exercise 14.2

1. Extend to find k majorities that occurs over $\lfloor n/k \rfloor$ in collection A, where $n = |A|$. Hint: Drop k different elements every time, till the remaining is less than k unique candidates. Any k-majority (the one over $\lfloor n/k \rfloor$) must remain in the end.

14.4 Maximum sum of sub-vector

For vector V, define a range $V[i...j]$ as sub-vector, the sum of sub-vector is $S = V[i] + V[i+1] + ... + V[j]$. Empty vector [ ] is sub-vector of any vector with sum 0. How to find the maximum sum of a given vector V [2]? For example, in vector [3, -13, 19, -12, 1, 9, 18, -16, 15, -15], the sub-vector [19, -12, 1, 9, 18] gives the maximum sum of 35. If all elements are positive, then the max is the total sum. If all are negative, then the empty vector gives the max sum of 0. Below is the exhaustive search implementation:

```
function Max-Sum(V)
    m ← 0, n ← |V|
    for i ← 1 to n do
        s ← 0
        for j ← i to n do
            s ← s + V[j]
            m ← Max(m, s)
    return m
```

The performance of exhaustive search is $O(n^2)$, where n is the vector length. Similar to majority number algorithm, we scan the vector. For every position i, record the sum of sub-vector ends with i as A, and the maximum sum so far as B. As shown in figure 14.11. A is not necessarily equal to B. We maintain $B \leq A$ always hold. When $B + V[i] > A$, we replace A with this greater value. When $B + V[i] < 0$, we reset B to 0. Below table gives the steps when scan $[3, -13, 19, -12, 1, 9, 18, -16, 15, -15]$.

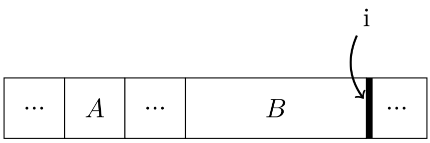


Figure 14.11: A: max sum so far; B: sum of the sub-vector ends with i.

max sum	max end at i	yet to 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	$[\ ]$

```
function Max-Sum(V)
    A ← 0, B ← 0, n ← |V|
    for i ← 1 to n do
        B ← Max(B + V[i], 0)
        A ← Max(A, B)
    return A
```

We can also find the maximum sum with fold: $S_{max} = fst \circ foldr\ f\ (0, 0)$, where f update the maximum sum so far:

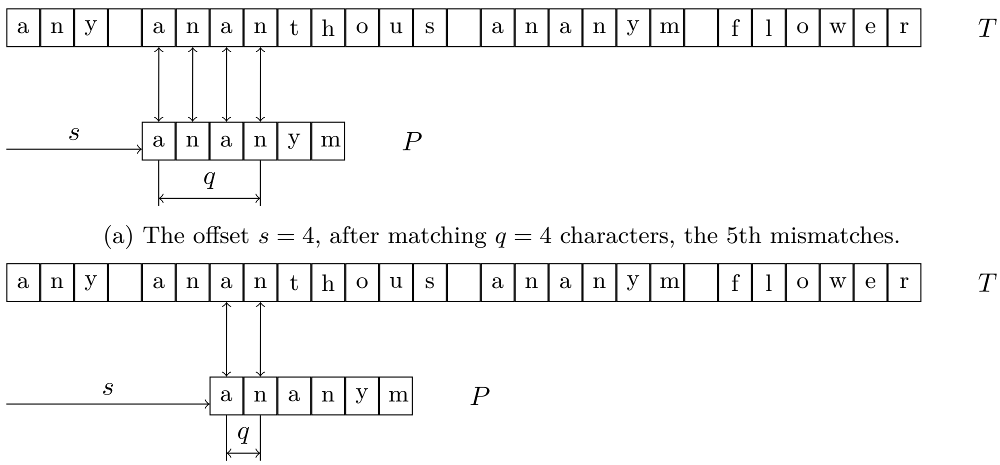
$$f\ x\ (S_m, S) = (S'_m = \max(S_m, S'), S' = \max(0, x + S)) \tag{14.23}$$

Exercise 14.3

1. Modify the solution that finds the max sum of sub-vector, returns the sub-vector of the maximum sum.
2. Bentley gives a divide and conquer algorithm to find the max sum in $O(n \lg n)$ time[2]. Split the vector at middle, recursively find the max sum in two halves, and the max sum that crosses the middle. Then pick the greatest. Implement this solution.
3. Find the sub-metrics in a $m \times n$ metrics that gives the maximum sum.

14.5 String matching

String matching is widely used in editor applications. We can use data structures like radix tree, prefix tree (chapter 6) to search string. We can also directly match the string, as shown in figure 14.12[5].



(a) The offset $s = 4$, after matching $q = 4$ characters, the 5th mismatches.

(b) Move $s = 4 + 2 = 6$.

Figure 14.12: Match 'ananym' in 'any ananthous ananym flower'.

We match a pattern P in text T. As shown in figure 14.12 (a), at offset $s = 4$, we one by one match characters in P and T. The first 4 are same, the 5th is 'y' in P, but 't' in T. We terminate matching, add s by 1 (move P to right by 1). Then restart matching 'ananym' and 'nantho...'. Actually, we can increase s more than 1. The first two characters 'an' happen to be the suffix of 'anan'. We can add s by 2 (move P to right 2 steps) as shown in figure 14.12 (b). We reuse the information from the 4 matched characters, skip some positions. Knuth, Morris and Pratt developed an efficient matching algorithm from this idea[85], known as 'KMP'. the initials of the three authors.

Denote the first k characters of text T as T_k (the k-character prefix of T). To shift P to the right s steps as many as possible, we need reuse the information of the matched q characters. As shown in figure 14.13, if we can shift P ahead, there exists some k, such that the first k characters are same as the last k characters of P_q, i.e., the prefix P_k is suffix of P_q. Define empty string "" is both the prefix and suffix of any string, hence the minimum $k = 0$ always exists. We need find the maximum k for the string that is both the prefix and suffix. Define the *prefix function* $\pi(q)$, that gives where to fallback when the $(q + 1)$-th character doesn't match[4].

[5]Some programming environment provide match tool, like `strstr` in C library, `find` in C++ library, `indexOf` in Java library.

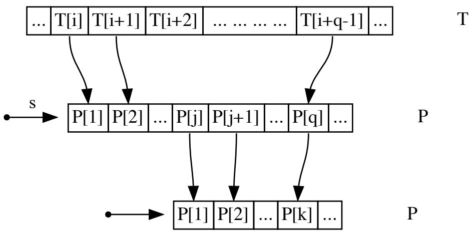


Figure 14.13: P_k is both the prefix and suffix of P_q.

$$\pi(q) = \max\{k | 0 \leq k < q, \text{and } P_k \text{ is suffix of } P_q\} \tag{14.24}$$

When match pattern P against text T from offset s, if fails after matching q characters, we next look up $q' = \pi(q)$ to get a fallback position q'. Then retry to compare $P[q']$ with the text:

```
function KMP(T, P)
    π ← Build-Prefixes(P)
    n ← |T|, m ← |P|, q ← 0
    for i ← 1 to n do
        while q > 0 and P[q + 1] ≠ T[i] do
            q ← π(q)
        if P[q + 1] = T[i] then
            q ← q + 1
        if q = m then
            position i − m is a solution
            q ← π(q)                                ▷ search further
```

The definition (14.24) is not practical to build $\pi(q)$. We can pre-process P as the following. If the first character doesn't match, then the longest prefix and suffix is empty: $\pi(1) = 0$, i.e., $P_k = P_0 = [\ ]$. When scan the q-th character in P, the prefix function values $\pi(i)$, $i = 1, 2, ..., q - 1$, are already known, and so far, the longest prefix P_k is also the suffix of P_{q-1}. As shown in figure 14.14, if $P[q] = P[k + 1]$, we find a greater k. We increase k by 1; otherwise, if $P[q] \neq P[k + 1]$, we lookup $\pi(k)$, and fallback to a shorter prefix $P_{k'}$, where $k' = \pi(k)$. Then compare the next character of this new prefix with the q-th character. Repeat this till k becomes zero (empty string), or the q-th character matches. Below table gives the prefix function values of 'ananym'. The k column is the maximum satisfying (14.24).

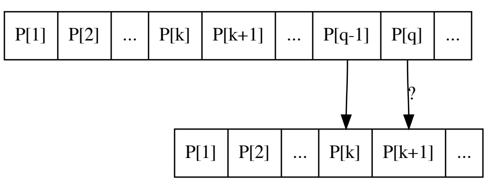


Figure 14.14: P_k is suffix of P_{q-1}, compare $P[q]$ and $P[k + 1]$.

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	“”

```
function Build-Prefixes(P)
    m ← |P|, k ← 0
    π(1) ← 0
    for q ← 2 to m do
        while k > 0 and P[q] ≠ P[k + 1] do
            k ← π(k)
        if P[q] = P[k + 1] then
            k ← k + 1
        π(q) ← k
    return π
```

The KMP algorithm pre-process the pattern string to build the prefix function in amortized $O(m)$ time[4]. The string matching is amortized $O(n)$ time. The total performance is $O(m + n)$, with additional $O(m)$ space to store the prefix function. Varies of pattern string P don't impact the performance. Consider match pattern 'aaa...ab' (length of m) in string 'aaa...a' (length of n). The m-th character doesn't match, we can only fallback by 1 repeatedly. The algorithm is still bound to linear time in this case.

14.6 Solution search

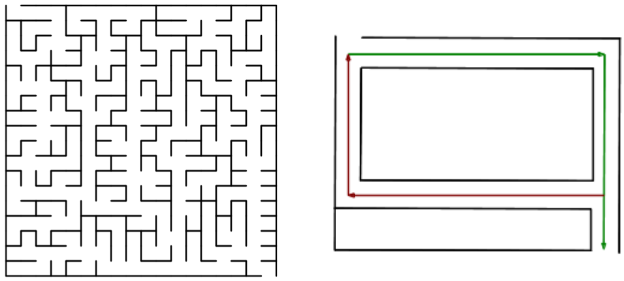
In early years of artificial intelligent, people developed methods to search for solutions. Different from the sequence searching and string matching, the solution may not directly exist among a set of candidates. We need construct the solution while try varies of options. Some problems are not solvable. Among the solvable ones, there can be multiple solutions. For example, a maze may have multiple ways out. We need find the optimal solution sometimes.

14.6.1 DFS and BFS

DFS stands for deep-first search, and BFS stands for breadth-first search. They are typical graph search algorithms. We give examples of them and skip the concept of graph.

Maze

Maze is a classic puzzle. There is saying: always turn right. However, it ends into loop as shown in figure 14.15. The decision matters when there are multiple ways. In fairy tales, one take some bread crumbs in a maze. Select a way, leave a piece of bread. If later enter a died end, then go back to the last place through the bread crumbs. Then go to another way. Whenever see bread crumbs left, one knows he visited it before. Then go back and try a different way. Repeat the 'try and check' step, one will either find the way out, or go back to the starting point (no solution). We use $m \times n$ matrix to define a maze, each element is 0, 1, means there is a way or not. Below matrix defines the maze in figure 14.15:



(a) Maze (b) Loop when keep turning right.

Figure 14.15: Maze

$$
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0
\end{array}
$$

Given a start point $s = (i, j)$, a destination $e = (p, q)$, we need find all paths from s to e. We first examine all points connected with s. For every such point k, recursively find all paths from k to e. Then prepend path s-k to every path from k to e. We need leave some 'bread crumbs' to avoid looping. We use a list P to record all visited points. Look it up and only try new ways.

$$solveMaze\ M\ \ s\ e = solve\ s\ [[\ ]] \tag{14.25}$$

Where:

$$solve\ s\ P = \begin{cases} s = e: & map\ (reverse \circ (s\ :))\ P \\ \text{otherwise}: & concat\ [solve\ k\ \ (map\ (s\ :)\ \ P) | k \leftarrow adj\ \ s, k \notin P] \end{cases} \tag{14.26}$$

The paths in P are reversed, we *reverse* the result back finally. Function *adj* p enumerates adjacent points to p:

$$adj\ \ (x, y) = [(x', y')| \quad \begin{array}{l} (x', y') \leftarrow [(x-1, y), (x+1, y), (x, y-1), (x, y+1)], \\ 1 \leq x' \leq m, 1 \leq y' \leq n, M_{x'y'} = 0\} \end{array} \tag{14.27}$$

This is essentially 'exhaustive search' all possible paths. We only need one way out. We need some data structure serves for the 'bread crumbs', recording the previous decisions. We always search on top of the latest decision. We can use stack to realize this is the last-in, first-out order. The stack starts from $[s]$. Pop s out and find all connected points of $a, b, ...$, push the new paths $[a, s]$, $[b, s]$ to the stack. Next pop $[a, s]$ out, examine all points connected to a. Then push new paths consist of 3 steps to the stack. Repeat this. The stack records paths in reversed order: from the farthest place back to the starting point, as shown in figure 14.16. If the stack becomes empty, we've tried all ways, and terminate the search; otherwise, we pop a path, expand to new adjacent points, and push the new paths back.

$$solveMaze\ M\ \ s\ e = solve\ [[s]] \tag{14.28}$$

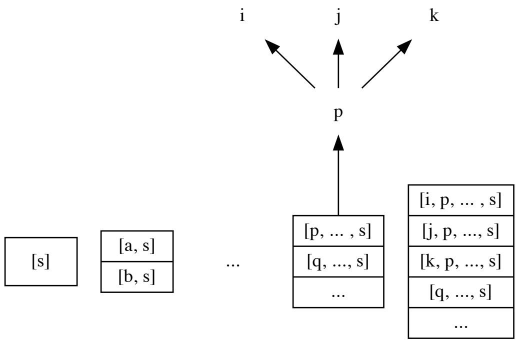


Figure 14.16: Search with a stack

Where:

$$
\begin{array}{rcl}
solve\ [\,] & = & [\,] \\
solve\ ((p{:}ps){:}cs) & = & \begin{cases} c = e: & reverse\ (p{:}ps) \\ ks = [\,]: & solve\ cs, \text{where } ks = filter\ (\notin ps)\ (adj\ p) \\ ks \neq [\,]: & solve\ ((map\ (:p{:}ps)\ ks) \mathbin{+\!\!+} cs) \end{cases}
\end{array}
\tag{14.29}
$$

Below is the iterative implementation:

```
function SOLVE-MAZE(M, s, e)
    S ← [s], L = [ ]
    while S ≠ [ ] do
        P ← POP(S)
        p ← LAST(P)
        if e = p then
            ADD(L, P)                              ▷ find a solution
        else
            for each k in ADJACENT(M, p) do
                if k ∉ P then
                    PUSH(S, P ++ [k])
    return L
```

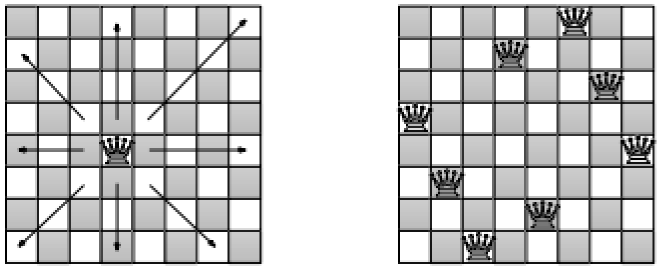
Each step tries 4 options (up, down, left, and right) through the backtrack. It seems the performance is $O(4^n)$, where n is the length of the path. The actual time won't be so large because we skip the visited places. In the worst case, we traverse all the reachable points *exactly once*. Hence the time is bound to $O(n)$, where n is the number of connected points. We need additional $O(n^2)$ space for the stack.

Exercise 14.4

1. Modify the implementation with stack, find all ways to the maze.

Eight queens puzzle

Although cheese has very long history, it was late in 1848, that Max Bezzel gave the 8 queens puzzle[89]. The queue is a powerful piece, it can attack any other pieces in the same row, column or diagonal at any distance, as shown in figure 14.17 (a). How to put 8 queens in the cheese board, such that none of them attack each other. Figure 14.17 (b) gives a solution.



(a) Queen (b) A solution

Figure 14.17: The eight queens puzzle.

To put 8 queens in 64 cells, there are total P_{64}^{8} permutations, about 4×10^{10}. Since no two queens can be in the same row or column. A solution must be a permutation of $[1, 2, 3, 4, 5, 6, 7, 8]$. For example, the permutation $[6, 2, 7, 1, 3, 5, 8, 4]$ means the first queen is at row 1, column 6, the second queen is at row 2 column 2, ..., and the 8th queen is at row 8, column 4. As such, we reduced the solution domain to $8! = 40320$ permutations. We arrange queues from the first row, there are 8 options (columns). For the next queue, we need skip some columns to avoid attacking the first queue. For the i-th queue, we need find the columns at row i, that not being attached by the first $i - 1$ queues. If all 8 columns are invalid, we go back to adjust the previous $i - 1$ queues. We find a solution after arrange all 8 queues. We record it and further search/backtrack to find all solutions. We start the search with a stack and a list: *solve* [[ ]] [ ]

$$
\begin{array}{rcl}
solve\ [\ ]\ s & = & s \\
solve\ (c{:}cs)\ s & = & \begin{cases} |c| = 8 : & solve\ cs\ (c{:}s) \\ \text{otherwise} : & solve\ ([x{:}c | x \leftarrow [1..8], x \notin c, safe\ x\ c] \mathbin{+\!\!+} cs)\ s \end{cases}
\end{array}
\tag{14.30}
$$

We've exhausted all options when the stack becomes empty, s records all the solutions; If the top arrangement c has length of 8, we add this newly find solution to s, then continue search; if $|c| < 8$, we find the columns that are not occupied ($x \notin c$), and attached by other queues in diagonal (through *safe* x c). Then push the new valid arrangement to the stack.

$$
safe\ x\ c = \forall (i, j) \leftarrow zip\ (reverse\ c)\ [1, 2, ...] \ \Rightarrow |x - i| \neq |y - j|, \text{where} : y = 1 + |c| \tag{14.31}
$$

safe checks if the queue at $y = 1 + |c|$ row, x column is in the diagonal with any other queue. Let $c = [i_{y-1}, i_{y-2}, ..., i_1]$ be the columns of the first $y - 1$ queues. We reverse c, zip with 1, 2, ... to form coordinates: $[(i_1, 1), (i_2, 2), ..., (i_{y-1}, y - 1)]$. Then check every (i, j) forms a diagonal with (x, y): $|x - i| \neq |y - j|$. This implementation is tail recursive, we can eliminate recursion with loops:

function SOLVE-QUEENS
    $S \leftarrow [[\ ]]$
    $L \leftarrow [\ ]$ ▷ Stores the solution
    while $S \neq [\ ]$ **do**
        $A \leftarrow$ POP(S) ▷ A: arrangement
        if $|A| = 8$ **then**
            ADD(L, A)
        else
            for $i \leftarrow 1$ to 8 **do**

```
            if Valid(i, A) then
                Push(S, A ++ [i])
    return L

function Valid(x, A)
    y ← 1 + |A|
    for i ← 1 to |A| do
        if x = A[i] or |y − i| = |x − A[i]| then
            return False
    return True
```

We only try the unoccupied columns among the 8, in total 15720 arrangements. It is far less than $8^8 = 16777216$[89]. Because the square board is horizontal and vertical symmetric, when find a solution, we can rotate, flip to obtain other symmetric solutions. We can expand to solve n queues puzzle, where $n \geq 4$. However, the time increase fast along with n. The backtrack algorithm is slightly better than the exhaustive permutations of 8 (bound to $o(n!)$).

Exercise 14.5

1. There are 92 solutions to the 8 queens puzzle. For any solution, we can rotate 90°, 180°, 270°, and flip it to obtain symmetric solutions. We can generate all solutions from 12 unique ones. Implement this method.
2. Extend the 8 queens to n queens.

Peg puzzle

As shown in figure 14.18, 6 frogs stay in 7 stones. Each frog can hop to the next stone if not occupied, or leap over to another empty one. The frogs can only move forward or stop, but not go back. Figure 14.19 give the rules. How to arrange the frogs to hop, leap, such that the left and right swap? Mark the left frogs as -1, the right as 1, the empty stone as 0. We are seeking the solution from $s = [-1, -1, -1, 0, 1, 1, 1]$ to $e = [1, 1, 1, 0, -1, -1, -1]$.

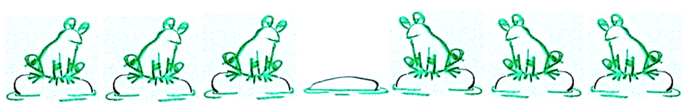
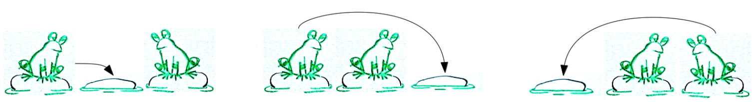


Figure 14.18: The leap frogs puzzle.



(a) Hop to the next stone

(b) Leap over to the right

(c) Leap over to the left

Figure 14.19: Moving rules.

This is a special form of the peg puzzle. The number of pegs can be 8 or other even numbers. Figure 14.20 shows some variants[6].

(a) Solitaire (b) Hop over (c) Draught board

Figure 14.20: Variants of the peg puzzle

Label the stones from left as 1, 2, ..., 7. There are at most 4 options for every move. When start for example, the frog on the 3rd stone can hop right to the empty stone; the frog on the 5th stone can hop left; the frog on the 2nd stone can leap right, the frog on the 6th stone can leap left. We record the stone status and try the 4 options at every step. backtrack and try other options when get stuck. Because every frog can only moves forward, the movement is not revertible. We needn't worry about repetition. We record the steps only for the final output. State L is some permutation of s. $L[i]$ is $\pm 1, 0$, indicates there is a frog on the i-th stone heading left, right, or the stone is empty. Let the empty stone be p, the 4 movements are:

1. Leap left: $p < 6$ and $L[p+2] > 0$, swap $L[p] \leftrightarrow L[p+2]$;
2. Hop left: $p < 7$ and $L[p+1] > 0$, swap $L[p] \leftrightarrow L[p+1]$;
3. Leap right: $p > 2$ and $L[p-2] < 0$, swap $L[p-2] \leftrightarrow L[p]$;
4. Hop right: $p > 1$ and $L[p-1] < 0$, swap $L[p-1] \leftrightarrow L[p]$.

Define four functions: $leap_l$, hop_l, $leap_r$, and hop_r, transition the status $L \mapsto L'$. If can't move, then returns L unchanged. We use a stack S to record the attempts. The stack starts from a singleton list, containing the initial status. List M records all solutions. We repeat pop the stack. If state $L = e$, then we add this new solution to M; otherwise, we try 4 moves on top of L, and push the new status back.

$$solve\ [[-1,-1,-1,0,1,1,1]]\ [\ ] \tag{14.32}$$

Where:

$$\begin{array}{rcl} solve\ [\ ]\ s &=& s \\ solve\ (c{:}cs)\ s &=& \begin{cases} L = e: & solve\ cs(reverse\ c : s), \text{where} : L = head\ c \\ \text{otherwise}: & solve\ ((map\ (: c)\ (moves\ L)) \mathbin{+\!\!+} cs)\ s \end{cases} \end{array} \tag{14.33}$$

function $moves$ tries 4 movements atop L:

$$moves\ L = filter(\neq L)\ [leap_l\ L, hop_l\ L, leap_r\ L, hop_r\ L] \tag{14.34}$$

The corresponding iterative implementation is as below:

```
function SOLVE(s, e)
    S ← [[s]]
```

[6] from http://home.comcast.net/~stegmann/jumping.htm

```
    M ← [ ]
    while S ≠ [ ] do
        s ← Pop(S)
        if s[1] = e then
            Add(M, Reverse(s))
        else
            for each m in Moves(s[1]) do
                Push(S, m:s)
    return M
```

This method gives two symmetric solution (15 steps for each). Below table lists one:

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

For 3 frogs in each side, it takes 15 steps. Extend this solution, we obtain a table of number of steps against the number of frogs in each side:

n frogs on each side	1	2	3	4	5	...
number of steps	3	8	15	24	35	...

The number of steps are all square numbers minus one: $(n+1)^2 - 1$. Let us prove it:

Proof. Compare the start and end states, every frog moves ahead $n+1$ stones. The $2n$ frogs in total move $2n(n+1)$ stones. Every frog on the left must meet every one from right once. The frog must leap over another one when meet. Because there are total n^2 meets, they cause all frogs move ahead $2n^2$ stones. The remaining moves are not leaps, but hops. There are total $2n(n+1) - 2n^2 = 2n$ hops. Sum up all n^2 leaps and $2n$ hops, the total steps are $n^2 + 2n = (n+1)^2 - 1$. □

The three puzzles share a common solution structure: start from some state. For example, the entrance to the maze; the empty chess board; pegs of [-1, -1, -1, 0, 1, 1, 1]. Search the solution, try multiple options every step. For example, 4 directions of up, down, left, and right in maze; 8 columns at each row; leap and hop, right and left. Although we don't know how far a decision leads to, we clearly know the final state. For example, the exit of the maze; complete arranging 8 queens; all pegs are swapped.

We apply the same strategy: repeatedly try an option; record it with the new state we obtain; backtrack when stuck and try another option. We either find a solution or exhaust all options and know the problem is unsolvable. There are variants, like to stop when find a solution, or continue search all solutions. If build a tree rooted at the starting state, every branch is an option, the search grows the tree deeper and deeper. We don't try alternatives at the same depth until fail and backtrack. Figure 14.21 shows the search order with arrows that go down then backtrack.

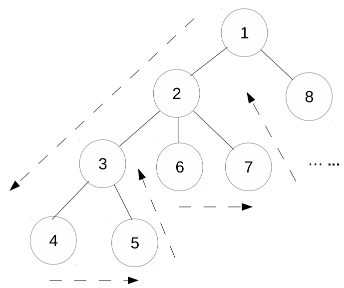


Figure 14.21: DFS search order.

We call it deep first search (DFS), and widely use it in practice. Some programming environments, like Prolog, use DFS as the default evaluation model. Prolog define a maze with rules:

```
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 X is connected with Y. This is a directed predicate, we can add a symmetric rule $c(Y, X)$ or create a undirected predicate. Figure 14.22 shows a directed graph. Given two places X and Y, Prolog tells if they are connected with the following program:

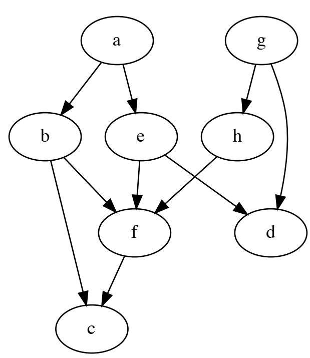


Figure 14.22: A directed graph.

```
go(X, X).
go(X, Y) :- c(X, Z), go(Z, Y)
```

This program says: a place X is connected with itself. Given two places X and Y, if X is connected with Z, and Z is connected with Y, then X is connected with Y. For multiple choices of Z, Prolog chooses one, and go on searching. It only tries another Z if the recursive search fails and backtrack. This is exactly the DFS. We can apply DFS when only need a solution, but don't care the number of steps. For example, we need a way out of the maze, although it may not be the shortest.

Exercise 14.6

1. Extend the pegs puzzle solution for n pegs on each side.

The wolf, goat, and cabbage puzzle

This traditional puzzle says that a farmer need cross the river with a wolf, a goat, and a bucket of cabbage. There is a boat. Only the farmer can drive it. The boat can only carry one thing a time. The wolf would kill the goat; the goat would bite the cabbage if they stay alone without the farmer. The puzzle asks to find the best solution to cross the river.

Since the wolf doesn't bite the cabbage, the farmer can safely carry the goal to the other side and go back. No matter carry the wolf or the cabbage next, the farmer need carry one back to avoid conflict. To find the best the solution, we parallel try all options and compare. Despite the direction, count back and forth 2 steps. We check all possible status after 1 step, 2 steps, 3 steps, ... till the farmer and all things move to the other side at n steps. This is the best solution.

But how to parallel try all options? Consider a lucky draw. People pick one from a box of colored balls. There is a black ball, and the rest are white. The one pick the black wins, or need return the white ball back to the box and wait for the next draw. We can define the rule that nobody try a second draw before all others pick. We line people in a queue. Every time the first person picks a ball, move to the tail if doesn't win. The queue ensures the fairness.

We apply the same method for the cross river puzzle. Let set A, B contains the things on each side. When start, $A = \{w, g, c, p\}$ includes the wolf, the goat, the cabbage, and the farmer; $B = \varnothing$. We move the farmer with or without another element between A and B. If a set doesn't contain the farmer, then it should has conflict elements. The goal is to swap elements in A and B with the fewest steps. We initiate a queue Q with the start status: $A = \{w, g, c, p\}$, $B = \ni$. As far as Q isn't empty, we de-queue the head, try all options, then en-queue the new status back to the tail. We find the solution when the head becomes $A = \varnothing$、$B = \{w, g, c, p\}$. Figure 14.24 shows the search order. As all options at the same level are tried, we needn't backtrack.

We can represent the set with a four bits binary number, each bit stands for an element, e.g., the wolf $w = 1$, the goat $g = 2$, the cabbage $c = 4$, and the farmer $p = 8$. 0 is the empty set, 15 is the full set. $3 = 1 + 2$, means the set {wolf, goat}. It's invalid because the wolf will kill the goat; $6 = 2 + 4$, is another conflict {goat, cabbage}. Every time, we move the highest bit (8), with or without another bit (4, 2, 1) from one number to the other. The options are:

$$mv\ A\ B = \begin{cases} B < 8 : & [(A - 8 - i, B + 8 + i) | i \leftarrow [0, 1, 2, 4], i = 0 \text{ or } A \barwedge i \neq 0] \\ \text{otherwise} : & [(A + 8 + i, B - 8 - i) | i \leftarrow [0, 1, 2, 4], i = 0 \text{ or } B \barwedge i \neq 0] \end{cases} \tag{14.35}$$

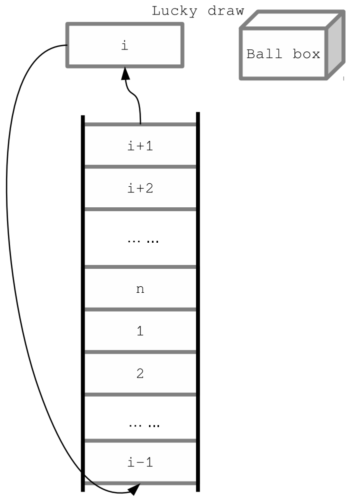


Figure 14.23: The i-th person de-queues, draw, then in-queue if doesn't win.

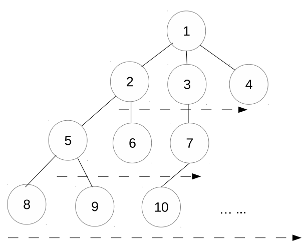


Figure 14.24: Start from 1, check all options 2, 3, 4 for the next step; then all option for the 3rd step, ...

Where $\overline{\wedge}$ is bitwise-and. We start searching from $Q = \{[(15, 0)]\}$, as: *solve Q*

$$
\begin{array}{rcl}
solve\ \varnothing & = & \varnothing \\
solve\ Q & = & \begin{cases} A = 0 : & reverse\ c, \text{where} : (A, B) = c, (c, Q') = pop\ Q \\ \text{where} : & solve\ (pushAll\ (map\ (: c)\ (filter\ (valid\ c)\ (mv\ A\ B)))\ Q') \end{cases}
\end{array}
\tag{14.36}
$$

Where *valid c* checks if the move (A, B) is valid, neither is 3 or 6, and is new (not in c):

$$
A, B \neq 3 \text{ or } 6, (A, B) \notin c \tag{14.37}
$$

Below is the iterative implementation:

```
function Solve
    S ← []
    Q ← {[(15, 0)]}
    while Q ≠ ∅ do
        C ← DeQ(Q)
        if C[1] = (0, 15) then
            Add(S, Reverse(C))
        else
            for each m in Moves(C) do
                if Valid(m, C) then
                    EnQ(Q, m:C)
    return S
```

It outputs two best solutions:

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

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

Water jugs puzzle

Given two water jugs, 9 litres and 4 litres. How to get 6 litres from river? This puzzle has history back to ancient Greece. A story said the French mathematician Simèon Denis Poisson solved this puzzle when he was a child. It also appears in Hollywood movie 'Die-Hard 3'. Pòlya uses this puzzle as an example of backwards induction[90].

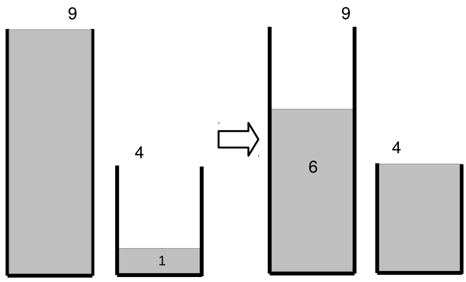


Figure 14.25: The last two steps.

After fill the 9 litres jug, then pour to the 4 litres jug twice, then we obtain 1 litre of water, as shown in figure 14.26. Backwards induction is a strategy, but not detailed algorithm. It can't directly answer how to get 2 litres of water from two jugs of 899 litres and 1147 litres for example.

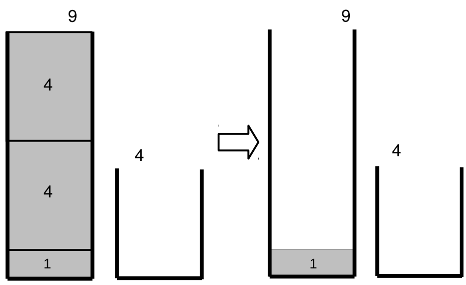


Figure 14.26: Fill the bigger jug, then pour to the smaller one twice.

Let the small jug be A, the big jug be B. There are 6 operations each time: (1) Fill jug A; (2) Fill jug B; (3) Empty jug A; (4) Empty jug B; (5) Pour from jug A to B; (6) Pour water from jug B to A. Below lists a series of operations (assume $a < b < 2a$).

A	B	operation
0	0	start
a	0	fill A
0	a	pour A to B
a	a	fill A
2a - b	b	pour A to B
2a - b	0	empty B
0	2a - b	pour A to B
a	2a - b	fill A
3a - 2b	b	pour A to B
...	...	...

Whatever operations, the water in each jug must be $xa + yb$, from some integers x and y, where a and b are jug volumes. From the number theory, we can get g litres of water if and only if g is dividable by the greatest common divisor of a and b, i.e., $gcd(a, b)|g$. If $gcd(a, b) = 1$ (a and b are coprime), then we can get any nature number g litres of water. Although we know the existence of the solution, we don't know the detailed steps. We can solve the Diophantine equation $g = xa + yb$, design the operations from x and y. Assume $x > 0, y < 0$, we fill jug A total x times, empty jug B total y times. For example, the small jug $a = 3$ litres, the big jug $b = 5$ litres, and the goal is to get $g = 4$ litres of water. Because $4 = 3 \times 3 - 5$, we design below operations:

A	B	operation
0	0	start
3	0	fill A
0	3	pour A to B
3	3	fill A
1	5	pour A to B
1	0	empty B
0	1	pour A to B
3	1	fill A
0	4	pour A to B

We fill jug A 3 times, empty jug B 1 time. We can apply the *Extended Euclid algorithm* in number theory to find x and y:

$$(d, x, y) = gcd_{ext}(a, b) \tag{14.38}$$

Where $d = gcd(a, b)$, $ax + by = d$. Assume $a < b$, the quotient q and remainder r satisfy $b = aq + r$. The common divisor d divides both a and b, hence d divides r too. Because $r < a$, we can scale down the problem to find $gcd(a, r)$:

$$(d, x', y') = gcd_{ext}(r, a) \tag{14.39}$$

Where $d = x'r + y'a$. Substitute $r = b - aq$ in:

$$\begin{array}{rcl} d & = & x'(b - aq) + y'a \\ & = & (y' - x'q)a + x'b \end{array} \tag{14.40}$$

Compare with $d = ax + by$, we have the following recursion:

$$\begin{cases} x & = y' - x' \dfrac{b}{a} \\ y & = x' \end{cases} \tag{14.41}$$

The edge case happens when $a = 0$: $gcd(0, b) = b = 0a + 1b$. Hence the extended Euclid algorithm can be defined as:

$$
\begin{array}{rcl}
gcd_{ext}(0, b) & = & (b, 0, 1) \\
gcd_{ext}(a, b) & = & (d, y' - x'\dfrac{b}{a}, x')
\end{array}
\tag{14.42}
$$

Where d, x', y' are defined in (14.39). If $g = md$, then mx and my is a solution; if $x < 0$, for example: $gcd_{ext}(4, 9) = (1, -2, 1)$. Since $d = xa + yb$, we repeatedly add x by b, and decrease y by a till $x > 0$. Such solution may not be the best one. For example, to get 4 litres of water from two jugs of 3 and 5 liters, the extended Euclid algorithm gives 23 steps:

```
[(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)]
```

While the best solution only need 6 steps:

```
[(0,0),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4)]
```

There are infinite many solutions for the Diophantine equation $g = xa + by$. The smaller $|x| + |y|$, the fewer steps. We can apply the same method as the 'cross river' puzzle. Try the 6 operations (fill A, fill B, pour A into B, pour B into A, empty A and empty B) in parallel to find the best solution. We use a queue to arrange the attempts. The element in the queue are series of pairs (p, q), where p and q are waters in each jug, as operations from the beginning. The queue starts from $\{[(0, 0)]\}$.

$$
solve\ a\ b\ g = bfs \{[(0, 0)]\}
\tag{14.43}
$$

As far as the queue isn't empty, we pop a sequence from the head. If the last pair of the sequence contains g litres, we find a solution. We reverse and output the sequence; otherwise, we try 6 operations atop the latest pair, filter out the duplicated ones and add back to the queue.

$$
\begin{array}{rcl}
bfs\ \varnothing & = & [\,] \\
bfs\ Q & = & \begin{cases} p \text{ or } q = g : & reverse\ s, \text{where} : (p, q) = head\ s, (s, Q') = pop\ Q \\ \text{otherwise} : & bfs\ (pushAll\ (map\ (: s)\ (try\ s))\ Q') \end{cases}
\end{array}
\tag{14.44}
$$

$$
try\ s = filter\ (\notin s)\ [f\ (p, q) | f \leftarrow \{fl_A, fl_B, pr_A, pr_B, em_A, em_B\}]
\tag{14.45}
$$

Where:

$$
\begin{cases}
fl_A\ (p, q) = (a, q) \\
fl_B\ (p, q) = (p, b) \\
em_A\ (p, q) = (0, q) \\
em_B\ (p, q) = (p, 0) \\
pr_A\ (p, q) = (\max(0, p + q - b), \min(x + y, b)) \\
pr_B\ (p, q) = (\min(x + y, a), \max(0, x + y - a))
\end{cases}
\tag{14.46}
$$

This method returns the solution with the fewest steps. To avoid storing the complete operation sequence in the queue, we can use a global history list, and link every operation back to its predecessor. As shown in figure 14.27, the start state is (0, 0), only 'fill A' and 'fill B' are applicable. We next try 'fill B' atop (3, 0), record the new state (3, 5). If apply 'empty A' to (3, 0), we'll go back to the starting point (0, 0). We skip it (shaded state). We add a 'parent' reference to each node in 14.27, and backtrack along it to the beginning.

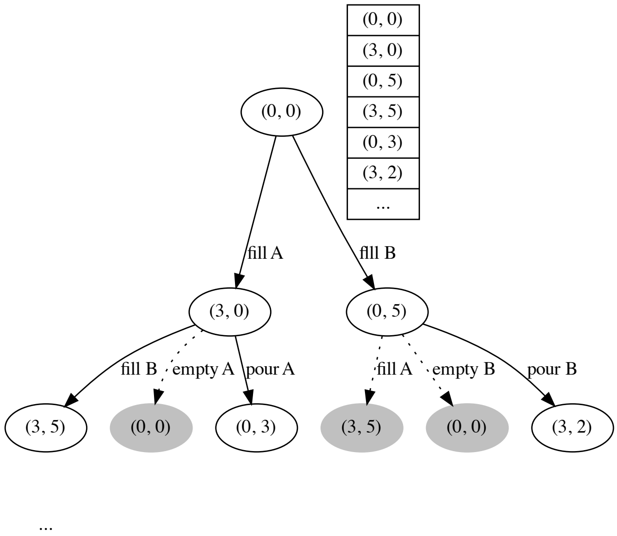


Figure 14.27: Store all states with a global list.

```
function Solve(a, b, g)
    Q ← {(0, 0, NIL)}                                   ▷ Queue
    V ← {(0, 0, NIL)}                                   ▷ Visited set
    while Q ≠ ∅ do
        s ← Pop(Q)
        if p(s) = g or q(s) = g then
            return Back-track(s)
        else
            for each c in Expand(s, a, b) do
                if c ≠ s and c ∉ V then
                    Push(Q, c)
                    Add(V, c)
    return NIL

function Expand(s, a, b)
    p ← p(s), q ← q(s)
    return [(a, q, s), (p, b, s), (0, q, s), (p, 0, s), (max(0, p+q−b), min(p+q, b), s), (min(p+
q, a), max(0, p + q − a), s)]

function Back-track(s)
    r ← [ ]
    while s ≠ NIL do
        (p, q, s′) = s
        r ← (p, q):r
        s ← s′
    return r
```

Exercise 14.7

1. Implement the solution to the water jugs puzzle with the extended Euclid algorithm.

2. Improve the extended Euclid algorithm, find the x and y that minimize $|x| + |y|$ for the optimal solution.

Kloski

Kloski is a block slide puzzle, as shown in figure 14.28. There are 10 blocks of 3 sizes: 4 pieces of 1×1; 4 pieces of 1×2, 1 piece of 2×1, 1 piece of 2×2. The goal is to slide the big block to the bottom slot. Figure 14.29 shows variants of this puzzle in Japan.

(a) Initial layout

(b) Layout after several movements

Figure 14.28: 'Huarong Escape', the traditional Chinese Kloski puzzle.

Figure 14.29: 'Daughter in the box', the Japanese Kloski puzzle.

We define the board as a 5×4 matrix, the row and column start from 0. Label the pieces from 1 to 10. 0 means empty cell. The matrix M gives the initial layout. The cells with value i is occupied by piece i. We use a map L to represent the layout, where $L[i]$ is the set of cells occupied by piece i. For example, $L[4] = \{(2, 1), (2, 2)\}$ means the 4th piece occupies cells $(2, 1)$ and $(2, 2)$. Label all 20 cells from 0 to 19, we can convert a pair of row, col to label: $c = 4y + x$. The 4th piece occupies cells $L[4] = \{9, 10\}$.

$$M = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix} \quad L = \left\{ \begin{array}{l} 1 \mapsto \{0, 4\}, 2 \mapsto \{3, 7\}, 3 \mapsto \{8, 12\}, \\ 4 \mapsto \{9, 10\}, 5 \mapsto \{11, 15\}, \\ 6 \mapsto \{16\}, 7 \mapsto \{13\}, 8 \mapsto \{14\}, \\ 9 \mapsto \{19\}, 10 \mapsto \{1, 2, 5, 6\} \end{array} \right\}$$

Define map $\varphi(M) \mapsto L$ and its reverse $\varphi^{-1}(L) \mapsto M$ to convert board and layout:

function $\varphi(M)$

```
    L ← {}
    for y ← 0 ~ 4 do
        for x ← 0 ~ 3 do
            k ← M[y][x]
            L[k] ← ADD(L[k], 4y + x)
    return L

function φ^{-1}(L)
    M ← [[0] × 4] × 5
    for each (k ↦ S) in L do
        for each c in S do
            x ← c mod 4, y ← ⌊c/4⌋
            M[y][x] ← k
    return M
```

We try all the 10 blocks in 4 directions: up, down, left, and right. For board matrix, the movement means: $(\Delta y, \Delta x) = (0, \pm 1), (\pm 1, 0)$; for layout of cell labels, it means: $d = \pm 1, \pm 4$. For example, move piece $L[i] = \{c_1, c_2\}$ to left, it becomes: $\{c_1 - 1, c_2 - 1\}$. We need avoid invalid movement in two edge cases: $d = 1, c \bmod 4 = 3$ and $d = -1, c \bmod 4 = 0$, they are invalid because the piece jump from one side to the other. Consider the two free cells, there are at most 8 movements. For example, the first step only have 4 options: move piece 6 right, move piece 7 or 8 down, move piece 9 left. Figure 14.30 shows how to verify the movement is valid.

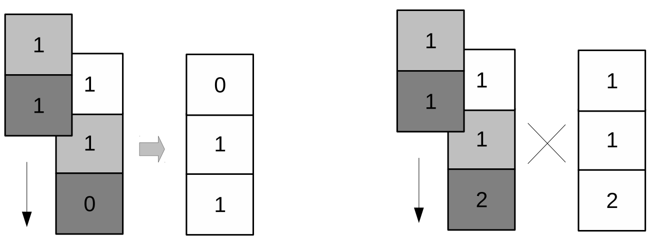


Figure 14.30: Left: two cells of 1 can move; Right: the lower cell of 1 conflicts with the cell of 2.

For the movement of piece k, it is valid if the target cells have value of 0 or k:

$$
\begin{array}{ll}
valid\ L[k]\ d: & \\
\forall c \in L[k] \Rightarrow & y = \lfloor c/4 \rfloor + \lfloor d/4 \rfloor, x = (c \bmod 4) + (d \bmod 4), \\
& (0,0) \le (y, x) \le (4, 3), M[y][x] \in \{k, 0\}
\end{array}
\tag{14.47}
$$

We may return to some layout after a series of slides. It's insufficient to only avoid duplicated matrix. Although $M_1 \neq M_2$, they are essentially the same layout.

$$M_1 = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix} \quad M_2 = \begin{bmatrix} 2 & 10 & 10 & 1 \\ 2 & 10 & 10 & 1 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 6 & 5 \\ 8 & 0 & 0 & 9 \end{bmatrix}$$

We need avoid duplicated layout. Treat all pieces of the same size same, we define normalized layout as: $\|L\| = \{p | (k \mapsto p) \in L\}$, the set of all cell labels in L. Both matrix above have the same normalized layout as {{1, 2, 5, 6}, {0, 4}, {3, 7}, {8, 12}, {9, 10}, {11, 15}, {16}, {13}, {14}, {19}}. We also need avoid mirrored layout, for example:

$$M_1 = \begin{bmatrix} 10 & 10 & 1 & 2 \\ 10 & 10 & 1 & 2 \\ 3 & 5 & 4 & 4 \\ 3 & 5 & 8 & 9 \\ 6 & 7 & 0 & 0 \end{bmatrix} \quad M_2 = \begin{bmatrix} 3 & 1 & 10 & 10 \\ 3 & 1 & 10 & 10 \\ 4 & 4 & 2 & 5 \\ 7 & 6 & 2 & 5 \\ 0 & 0 & 9 & 8 \end{bmatrix}$$

Both have the same normalized layout. Define the mirror function:

$$mirror(\|L\|) = \{\{f(c) | c \in s\} | s \in \|L\|\} \tag{14.48}$$

Where $f(c) = 4y' + x', y' = \lfloor c/4 \rfloor, x' = 3 - (c \bmod 4)$. We use a queue to arrange the search. The element in the queue has two parts: a series of movements, and the resulted layout. The movement is a pair (k, d), means move piece k by d $(\pm 1, \pm 4)$. Initialize the queue $Q = \{(s, [\,])\}$, where s is the start layout. As far as the queue isn't empty $Q \neq \varnothing$, we get its head, examine whether the big block (piece 10) arrives at $t = \{13, 14, 17, 18\}$, i.e., $L[10] = t$. Terminates if yes; otherwise, we try up, down, left, right for every piece, add every valid (k, d), that leads to unique layout to the queue. We use a set H to records all visited normalized layouts to avoid repetition.

$$\begin{array}{rcl} solve\ \varnothing\ H & = & [\,] \\ solve\ Q\ H & = & \begin{cases} L[10] = t : & reverse\ ms, \text{where} : ((L, ms), Q') = pop\ Q \\ \text{otherwise} : & solve\ (pushAll\ cs\ Q')\ H' \end{cases} \end{array} \tag{14.49}$$

Where $cs = [(move\ L\ e, e{:}ms) | e \leftarrow expand\ L]$ are the new movements expanded.

$$expand\ L = \{(k, d) | \quad \begin{array}{l} k \leftarrow [1, 2, ..., 10], d \leftarrow [\pm 1, \pm 4], \\ valid\ k\ d, unique\ k\ d\} \end{array} \tag{14.50}$$

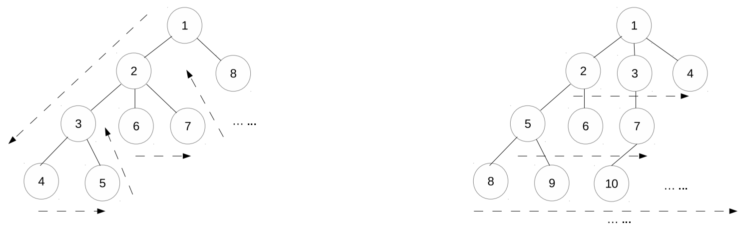
Function $move$ slides piece $L[k]$ by d to: $move\ L\ (k, d) = map\ (+d)\ L[k]$. $unique$ checks if the normalized layout $\|L'\| \notin H$ and its mirror $mirror(\|L'\|) \notin H$. Add them to H' if new. Below are the iterative implementation. The solution has 116 steps (1 cell a step). The last 3 are:

function SOLVE(s, e)
  $H \leftarrow \{\|s\|\}$
  $Q \leftarrow \{(s, \varnothing)\}$
  while $Q \neq \varnothing$ **do**
    $(L, p) \leftarrow$ POP(Q)
    if $L[10] = e$ **then**
      return (L, p)
    else
      for each L' in EXPAND(L, H) **do**
        PUSH($Q, (L', L)$)
        ADD($H, \|L'\|$)

return ∅

```
['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']
```

The cross river puzzle, water jugs puzzle, and the Kloski puzzle share the common solution structure. Similar to the DFS, they have start and end states. For example, the cross river puzzle starts with all things on one side, the other side is empty; it ends with all things on the other side. The water jugs puzzle starts with tow empty jugs; it ends with either jug has g litres of water. The Klotski puzzle starts with some layout, it ends with some layout that the big block arrives at the bottom slot. Every puzzle have a set of rules, transfer from a state to another. We 'parallel' try all options. We don't search further until complete trying all options of the same step. This search strategy ensure we find the solution with the fewest step before others. Because we expand horizontally, it's called Breadth-first search. Figure 14.31 compares DFS and BFS.



(a) DFS (b) BFS

Figure 14.31: DFS and BFS.

Because we can't really search in parallel, we realize BFS with a queue. Repeat de-queue the candidate with fewer steps from head, and en-queue new candidate with more steps to tail. BFS provides a simple method to search the solution with the fewest steps. However, it can't directly search for generic optimal solution. Consider the directed graph in figure 14.32, the length of each section varies. We can't use BFS to find the shortest path between two cities. For example, the shortest path from a to c is not the one with the fewest steps: $a \to b \to c$. The total length is 22, but the path with more steps $a \to e \to f \to c$ has the length of 20.

Exercise 14.8

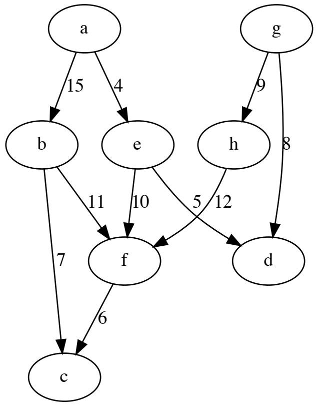


Figure 14.32: A weighted directed graph.

1. John Conway gives a slide tile puzzle. Figure 14.33 is a simplified example. There are 8 cells, 7 are occupied. Label the pieces from 1 to 7. Each piece can slide to the connected free cell. (two cells are connected if there is a line between them.) How to reverse the pieces from 1, 2, 3, 4, 5, 6, 7 to 7, 6, 5, 4, 3, 2, 1 by sliding? Write a program to solve this puzzle.

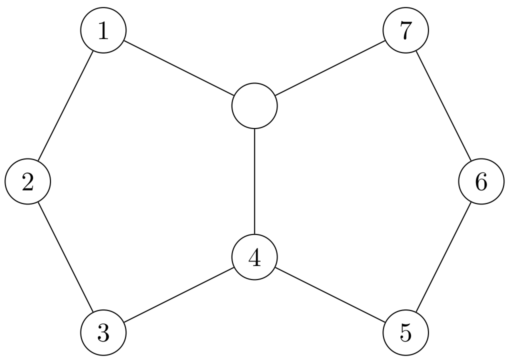


Figure 14.33: Conway slide puzzle

14.6.2 Greedy algorithm

People need find the 'best' solution to minimize time, space, cost, energy, and etc. It's not easy to find the optimal solution within limited resource. Many problem don't have solution in polynomial time, however, there exist simple solution for a small portion of special problems.

Huffman coding

Huffman coding encodes information with the shortest length. The ASCII code needs 7 bits to encode characters, digits, and symbols. It can represent $2^7 = 128$ symbols. We need at least $\log_2 n$ 0/1 bits to distinguish n symbols. Below table encodes upper case English letters, maps A to Z from 0 to 25, each with 5 bits. Zero is padded as 00000 but not 0. Such scheme is called fixed-length coding.

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

It encodes 'INTERNATIONAL' to a binary number of 65 bits:

```
00010101101100100100100011011000000110010001001110101100000011010
```

Another scheme is variable-length coding. Encode A as single bit 0, encode C as 10 of two bits, encode Z as 11001 of 5 bits. Although the code length is shorter, it has ambiguity when decode. For example, the binary number 1101 can stand for 1 followed with 101 (decoded as 'BF') or 110 followed with 1 (decoded as 'GB'), or 1101 (decoded as N). The Morse code is variable-length. It encodes the most used letter 'E' as '.', encodes 'Z' as '- -..'. Particularly, it uses a special pause separator to indicate the termination of a code, eliminates the ambiguity. Below code table is ambiguity free:

char	code	char	code
A	110	E	1110
I	101	L	1111
N	01	O	000
R	001	T	100

It encodes 'INTERNATIONAL' with 38 bits only:

```
10101100111000101110100101000011101111
```

The reason why it's ambiguity free is because there is no code is the prefix of the other. Such code is called *prefix-code*. (but not the 'non-prefix code'.) Since the prefix-code needn't separator, we can further shorten the code length. Given a text, can we find a prefix-code scheme, that produces the shortest code? In 1951, Robert M. Fano told the class that those who could solve this problem needn't take the final exam. Huffman was still a student in MIT[91]. He almost gave up and started preparing the final exam when found the answer. Huffman created the coding table according to the frequency of the symbol appeared in the text. The more used one is assigned with the shorter code. Process the text, and calculate the occurrence for each symbol. Define the weight as the frequency. Huffman uses a binary tree to generate the prefix-code. The symbols are stored in the leaf nodes. Traverse from the root to generate the code, add 0 when go left, 1 when go right, as shown in figure 14.34. For example, starting from the root, go left, then right, we arrive at 'N'. Therefore, 'N' is encoded as '01'; While the paths of 'A' is right, right, left, encoded as '110'.

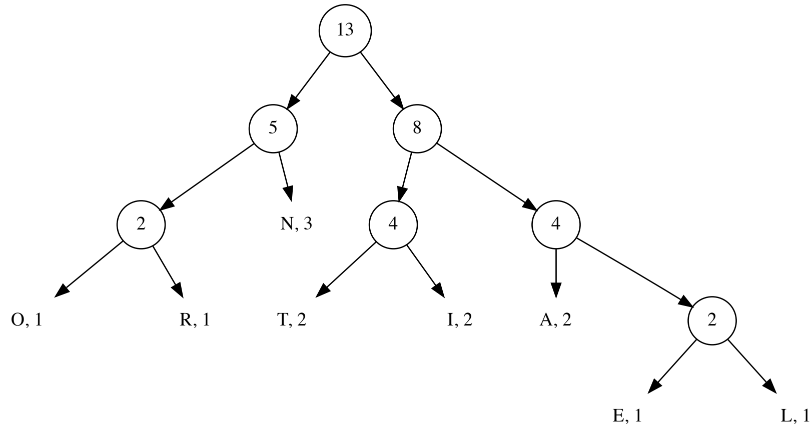


Figure 14.34: Huffman tree

We can use the tree to decode as well. Scan the binary bits, go left for 0, and right for 1. When arrive at a leaf, we decode the symbol from it. Then restart from the root to continue scan. Huffman build the tree in bottom-up way. When start, wrap all symbols in leaves. Every time, pick two nodes with the minimum weights, merge them to a branch node of weight w. where $w = w_1 + w_2$ is the sum of the two weights. Repeat pick and merge the two smallest weighted trees till we get the final tree, as shown in figure 14.35.

We reuse the binary tree definition for Huffman tree. We augment the weight and only hold the symbol in leaf node. Let the branch node be (w, l, r), where w is the weight, l and r are the left and right sub-trees. Let the leaf be (w, c), where c is the symbol. When merge trees, we sum the weight: $merge\ a\ b\ = (weight\ a + weight\ b, a, b)$, where:

$$\begin{array}{rcl} weight\ (w, a) & = & w \\ weight\ (w, l, r) & = & w \end{array} \tag{14.51}$$

Below function repeatedly pick and merge the minimum weighted trees:

$$\begin{array}{rcl} build\ [t] & = & t \\ build\ ts & = & build\ (merge\ t_1 t_2)\ ts', \text{where} : (t_1, t_2, ts') = extract\ ts \end{array} \tag{14.52}$$

Function $extract$ picks two trees with minimal weight. Define $t_1 < t_2$ if $weight\ t_1 < weight\ t_2$.

$$extract(t_1{:}t_2{:}ts) = foldr\ min_2\ (\min t_1\ t_2, \max t_1\ t_2, [\,])\ ts \tag{14.53}$$

Where:

$$min_2\ t\ (t_1, t_2, ts) = \begin{cases} t < t_2 : & (\min\ t\ t_1, \max\ t\ t_1, t_2{:}ts) \\ \text{otherwise} : & (t_1, t_2, t{:}ts) \end{cases} \tag{14.54}$$

To iterate building Huffman tree, we store n sub-trees in array A. Scan A from right to left, if the weight of $A[i]$ is less than $A[n-1]$ or $A[n]$, we swap $A[i]$ and $\text{MAX}(A[n-1], A[n])$. Merge $A[n]$ and $A[n-1]$ after scan, and shrink the array by one. Repeat this to build the Huffman tree:

```
function Huffman(A)
    while |A| > 1 do
        n ← |A|
```

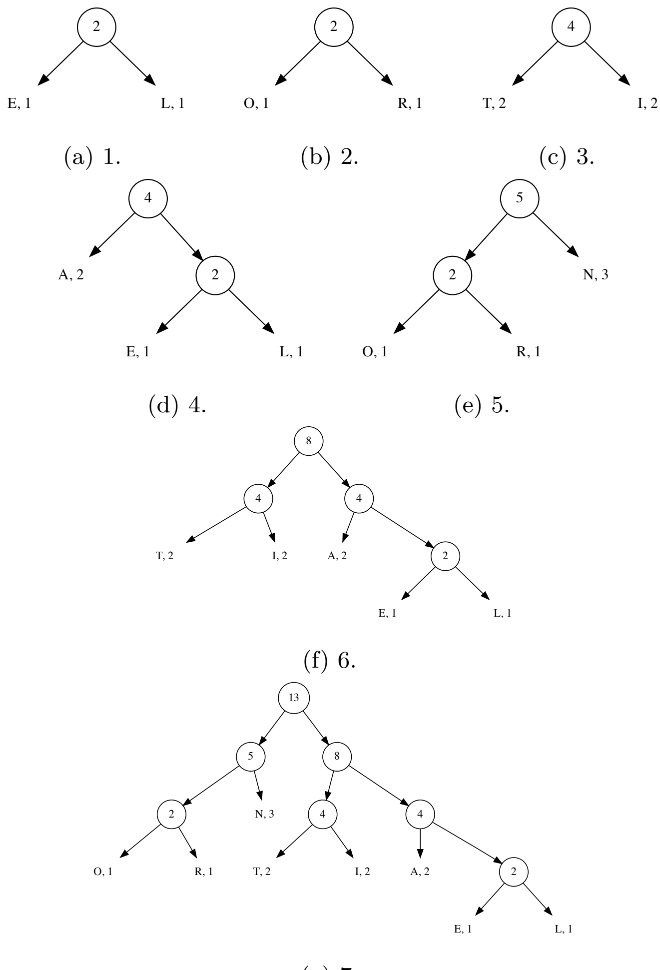


Figure 14.35: Build a Huffman tree.

```
        for i ← n − 2 down to 1 do
            T ← Max(A[n], A[n − 1])
            if A[i] < T then
                Exchange A[i] ↔ T
        A[n − 1] ← Merge(A[n], A[n − 1])
        Drop(A[n])
    return A[1]
```

We can build the code table from the Huffman tree. Let $p = [\ ]$. Traverse from the root, update $p \leftarrow 0{:}p$ when go left; $p \leftarrow 1{:}p$ when go right. When arrive at leaf of symbol c, record $c \mapsto reverse\ p$ to the code table. Define (Curried form): $code\ = traverse\ [\ ]$, where:

$$
\begin{array}{rcl}
traverse\ p\ (w, c) & = & [c \mapsto reverse\ p] \\
traverse\ p\ (w, l, r) & = & traverse\ (0{:}p)\ l \mathbin{+\!\!+} traverse\ (1{:}p)\ l
\end{array}
\tag{14.55}
$$

When encoding, we scan the text w while looking up the code table $dict$ to generate binary bits:

$$
encode\ dict\ w = concatMap\ (c \mapsto dict[c])\ w, \text{where} : dict = code\ T \tag{14.56}
$$

Conversely, when decoding, we scan the binary bits bs while looking up the tree. Start from the root, go left for 0, right for 1; output symbol c when arrive at leaf; then reset to the root to continue. $decode\ T\ bs = lookup\ T\ bs$, where:

$$
\begin{array}{rcl}
lookup\ (w, c)\ [\ ] & = & [c] \\
lookup\ (w, c)\ bs & = & c : lookup\ T\ bs \\
lookup\ (w, l, r)\ (b{:}bs) & = & lookup\ (\text{if } b = 0 \text{ then } l \text{ else } r)\ bs
\end{array}
\tag{14.57}
$$

Huffman tree building reflects a special strategy: always pick the two trees with the minimal weight for merge every time. The series of *local* optimal options generate a *global* optimal prefix-code. Local optimal sub-solutions are not necessary lead to global optimal solution usually. Huffman coding is an exception. We call the strategy that always choose the local optimal option as the *greedy* strategy. Greedy method simplifies and works for many problems. However, it's not easy to tell whether the greedy method generates the global optimal solution. The generic formal proof is still an active research area[4].

Exercise 14.9

1. Implement the imperative Huffman code table algorithm.

Change making problem

How to change money with as few coins as possible? Suppose there are 5 values of coins: 1, 5, 25, 50, and 100. We define it as a set $C = \{1, 5, 25, 50, 100\}$. To change x money, we can apply the greedy method, always choose the coin values most:

$$
\begin{array}{rcl}
change\ 0 & = & [\ ] \\
change\ x & = & c_m : change\ (x - c_m), \text{where} : c_m = \max\ \{c \in C, c \le x\}
\end{array}
\tag{14.58}
$$

For example, to change 142 money, this function output a coin list: [100, 25, 5, 5, 5, 1, 1]. We can convert it to [(100, 1), (25, 1), (5, 3), (1, 2)], meaning 1 coin of 100, 1 coin of 25, 3 coins of 5, 2 coins of 1. For the coin system of C, the greedy method can find the optimal solution. Actually, it is applicable for most coin systems in the world. There are exceptions for example: $C = \{1, 3, 4\}$. To change money $x = 6$, the optimal solution is 2 coins of 3, however, the greedy method gives $6 = 4 + 1 + 1$, total 3 coins.

Although it's not the optimal solution, the greedy method often gives a simplified sub-optimal implementation. The result is often good enough in practice. For example, the word-wrap is a common functionality in editors. If the length of the text T exceeds the page width w, we need break it into lines. Let the space between words be s, below greedy implementation gives the approximate optimal solution: put as many words as possible in a line.

$L \leftarrow W$
for $w \in T$ **do**
    if $|w| + s > L$ **then**
        Insert line break
        $L \leftarrow W - |w|$
    else
        $L \leftarrow L - |w| - s$

Exercise 14.10

1. Use heap to build the Huffman tree: take two trees from the top, merge then add back to the heap.
2. If we sort the symbols by their weight as A, there is a linear time algorithm to build the Huffman tree: use a tree Q to store the merge result, repeat take the minimal weighted tree from Q and the head of A, merge then add to the queue. After process all trees in A, there is a single tree in the Q, which is the Huffman tree. Implement this algorithm.
3. Given a Huffman tree T, implement the decode algorithm with fold left.

14.6.3 Dynamic programming

Consider how to find the best solution to change money for any coin system. Let the best solution to change x money is C_m (the list of coins). We can partition C_m into two groups: C_1 and C_2, with values x_1 and x_2 respectively, i.e., $C_m = C_1 \mathbin{+\!\!+} C_2$ and $x = x_1 + x_2$. We'll prove that C_1 is the optimal solution to change x_1, and C_2 is the optimal solution to change x_2.

Proof. For x_1, suppose there exists another solution C'_1 with less coins than C_1. Then the solution $C'_1 \mathbin{+\!\!+} C_2$ changes x with less coins than C_m. This conflicts with the fact that C_m is the optimal solution to change x. We can prove C_2 is the optimal solution to change X_2 in the same way. □

The reverse predication is not true. for any integer $y < x$, divide the original problem to two sub-problems: change y and $x - y$. It's not necessary the overall optimal solution when combine the two optimal solutions. For example, use 3 values $C = \{1, 2, 4\}$ to change $x = 6$. The optimal solution needs two coins: $2 + 4$. As $6 = 3 + 3$, divide it to two same sub-problems of changing 3. Each sub-problem has the optimal solution: $3 = 1 +$, however, the combination $(1 + 2) + (1 + 2)$ needs 4 coins. If an optimal problem can be divided into several sub optimal problems, we call it has optimal substructure. The change money problem has optimal substructure, but we need divide based on the coin value, but not an arbitrary integer.

$$\begin{array}{rcl} \textit{change}\ 0 & = & [\,] \\ \textit{change}\ x & = & \min\ [c : \textit{change}\ (x - c) | c \in C, c < x] \end{array} \tag{14.59}$$

Where min picks the shortest list, However, this definition is impractical. There are too much duplicated computation. For example $C = \{1, 2, 25, 50, 100\}$, when computes *change*(142), it needs further compute *change*(141), *change*(137), *change*(117), *change*(92), *change*(42). For *change*(141), minus it by 1, 2, 25, 50, 100, we go back to 137, 117, 92, 42. The search domain expands at 5^n. Reuse the idea to generate Fibonacci numbers, we can use a table T to records the optimal solution to the sub-problems. T starts from empty. When change money y, we lookup $T[y]$ first. If $T[y] = \varnothing$, then recursively compute the sub-problem, then save the sub-solution in $T[y]$.

$T \leftarrow [[\,], \varnothing, \varnothing, ...]$ ▷ $T[0] = [\,]$
function CHANGE(x)
  if $x > 0$ and $T[x] = \varnothing$ **then**
    for each c in C and $c \leq x$ **do**
      $C_m \leftarrow c : \text{CHANGE}(x - c)$
      if $T[x] = \varnothing$ or $|C_m| < |T[x]|$ **then**
        $T[x] \leftarrow C_m$
  return $T[x]$

We can bottom-up generate optimal solutions for each sub-problem. From $T[0] = [\,]$, generate $T[1] = [1]$, $T[2] = [1, 1]$, $T[3] = [1, 1, 1]$, $T[4] = [1, 1, 1, 1]$, as shown in table 14.1(a). There are two options for $T[5]$: 5 coins of 1, or a coin of 5. The latter need fewer coins. We update the optimal table to 14.1(b), $T[5] = [5]$. Next change money $x = 6$. Both 1 and 5 are less than 6, there are two options: (1) $1 + T[5]$ gives $[1, 5]$; (2) $5 + T[1]$ gives $[5, 1]$. They are equivalent, we pick either $T[6] = [1, 5]$. For $T[i]$, where $i \leq x$, we check every coin value $c \leq i$. Lookup $T[i - c]$ for the sub-problem, then plus c to get a new solution. We pick the fewest one as $T[i]$.

x	0	1	2	3	4
optimal solution	[ ]	[1]	[1, 1]	[1, 1, 1]	[1, 1, 1, 1]

(a) Optimal solution for $x \leq 4$

x	0	1	2	3	4	5
optimal solution	[ ]	[1]	[1, 1]	[1, 1, 1]	[1, 1, 1, 1]	[5]

(b) Optimal solution for $x \leq 5$

Table 14.1: Optimal solution table

function CHANGE(x)
  $T \leftarrow [[\,], \varnothing, ...]$
  for $i \leftarrow 1$ to x **do**
    for each c in C and $c \leq i$ **do**
      if $T[i] = \varnothing$ or $1 + |T[i - c]| < |T[i]|$ **then**
        $T[i] \leftarrow c : T[i - c]$
  return $T[x]$

There are many duplicated content in the optimal solution table as below. A solution contains the sub-solutions. We can only record the changed part: the coin c we chosen for $T[i]$ and the number n of coins, i.e., $T[i] = (n, c)$. To generate the list of coins for x, we lookup $T[x]$ to get c, then lookup $T[x - c]$ to get c', ... repeat this to $T[0]$.

value	6	7	8	9	10	...
optimal solution	[1, 5]	[1, 1, 5]	[1, 1, 1, 5]	[1, 1, 1, 1, 5]	[5, 5]	...

```
function CHANGE(x)
    T ← [(0, ∅), (∞, ∅), (∞, ∅), ...]
    for i ← 1 to x do
        for each c in C and c ≤ i do
            (n, _) ← T[i − c], (m, _) ← T[i]
            if 1 + n < m then
                T[i] ← (1 + n, c)
    s ← [ ]
    while x > 0 do
        (_, c) ← T[x]
        s ← c : s
        x ← x − c
    return s
```

We can build the optimal solution table T with left fold: *foldl fill* $[(0, 0)]$ $[1, 2, ...]$, where:

$$fill\ T\ x = T \triangleright \min \{(fst\ T[x - c], c) | c \in C, c \le x\} \tag{14.60}$$

Where $s \triangleright a$ append a to the right of s (see finger tree in chapter 12). Then rebuild the optimal solution backwards from T:

$$\begin{array}{rcl} change\ 0\ T & = & [\,] \\ change\ x\ T & = & c : change\ (x - c)\ T, \text{where} : c = snd\ T[x] \end{array} \tag{14.61}$$

For $x = n$, we loop n times, check at most $k = |C|$ coins. The performance is bound to $\Theta(nk)$[7], and need $O(n)$ space to persist T both in the top-down and bottom-up implementations. The solution to the sub-problem is used many times to compute the global optimal solution. We call it overlapping sub-problems. Richard Bellman developed dynamic programming in 1940s. It has two properties.

1. Optimal sub-structure. The problem can be broken down into small problems. The optimal solution can be constructed from the solutions of these sub-problems;

2. Overlapping sub-problems. The solution of the sub-problem is reused multiple times to find the overall solution.

Longest common sub-sequence

Different with sub-string, the sub-sequence needn't be consecutive. For example: the longest common sub-string of 'Mississippi' and 'Missunderstanding' is 'Miss', while the longest common sub-sequence is 'Misssi' as shown in figure 14.36. If rotate the figure by 90°, it turns to be a 'diff' result between them. This is a common function in version control tools. The longest common sub-sequence of x and y are defined as below:

$$\begin{array}{rcl} LCS([\,], ys) & = & [\,] \\ LCS(xs, [\,]) & = & [\,] \\ LCS(x{:}xs, y{:}ys) & = & \begin{cases} x = y : & x : LCS(xs, ys) \\ \text{otherwise} : & \max\ LCS(x{:}xs, ys)\ LCS(xs, y{:}ys) \end{cases} \end{array} \tag{14.62}$$

Where max picks the longer sequence. There is optimal sub-structure in the definition of LCS. It can be broken down into sub-problems. The sequence length reduced at least by 1 every time. There are overlapping sub-problems. The longest common sub-sequence

[7]upper bound

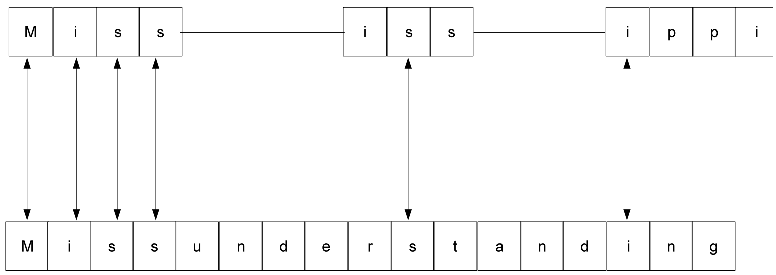


Figure 14.36: The longest common sub-sequence

of the sub-strings are reused multiple times to find the global optimal solution. We use a 2D table T to record the optimal solution of the sub-problems. The row and column represent xs and ys respectively. Index the sequence from 0. Row 0, column 0 represents the empty sequence. $T[i][j]$ is the length of $LCS(xs[0..j], ys[0..i])$. We finally build the longest common sub-sequences from T. Because $LCS([\ ], ys) = LCS(xs, [\ ]) = [\ ]$, row 0 and column 0 are all 0s. Consider 'antenna' and 'banana' for example, we fill row 1 from $T[1][1]$. 'b' is different from any one in 'antenna', hence row 1 are all 0s. For $T[2][1]$, the row and column are corresponding to 'a', $T[2][1] = T[1][0] + 1 = 1$, i.e., $LCS(\text{a}, \text{ba}) = \text{a}$. Next move to $T[2][2]$, 'a' neq 'n', we choose the greater one between the above ($LCS(\text{an}, \text{b})$) and the left ($LCS(\text{a}, \text{ba})$) as $T[2][2]$, which equals to 1, i.e., $LCS(\text{ba}, \text{an}) = \text{a}$. In this way, we step by step fill the table out. The rule is: for $T[i][j]$, if $xs[i-1] = ys[i-1]$, then $T[i][j] = T[i-1][j-1] + 1$; otherwise, pick the greater one from above $T[i-1][j]$ and the left $T[i][j-1]$.

		0	1	2	3	4	5	6	7
		[ ]	a	n	t	e	n	n	a
0	[ ]	0	0	0	0	0	0	0	0
1	b	0	0	0	0	0	0	0	0
2	a	0	1	1	1	1	1	1	1
3	n	0	1	2	2	2	2	2	2
4	a	0	1	2	2	2	2	2	3
5	n	0	1	2	2	2	3	3	3
6	a	0	1	2	2	2	3	3	4

function LCS(xs, ys)
    $m \leftarrow |xs|, n \leftarrow |ys|$
    $T \leftarrow [[0, 0, ...], [0, 0, ...], ...]$ ▷ $(m+1) \times (n+1)$
    for $i \leftarrow 1$ to m **do**
        for $j \leftarrow 1$ to n **do**

```
            if xs[i] = ys[j] then
                T[i + 1][j + 1] ← T[i][j] + 1
            else
                T[i + 1][j + 1] ← MAX(T[i][j + 1], T[i + 1][j])
    return FETCH(T, xs, ys)                                ▷ build the LCS
```

We next build the longest common sub-sequence from T. Start from the bottom-right, if $xs[m] = ys[n]$, then $xs[m]$ is the tail of the LCS, we next compare $xs[m-1]$ and $ys[n-1]$; otherwise, we pick the greater one from $T[m-1][n]$ and $T[m][n-1]$ and go on.

```
function FETCH(T, xs, ys)
    m ← |xs|, n ← |ys|
    r ← [ ]
    while m > 0 and n > 0 do
        if xs[m − 1] = ys[n − 1] then
            r ← xs[m − 1] : r
            m ← m − 1
            n ← n − 1
        else if T[m − 1][n] > T[m][n − 1] then
            m ← m − 1
        else
            n ← n − 1
    return r
```

Exercise 14.11

1. For the longest common sub-sequence, build the optimal solution table with fold.

Subset sum

Given a set X of integers, how to find all the subsets $S \subseteq X$, that the sum of elements in S is s, i.e., $\sum S = \sum_{i \in S} i = s$? For example, $X = \{11, 64, -82, -68, 86, 55, -88, -21, 51\}$, there are three subsets with sum $s = 0$: $S = \varnothing$, $\{64, -82, 55, -88, 51\}$, $\{64, -82, -68, 86\}$. We need exhaust 2^n subset sums, where $n = |X|$, the performance is $O(n2^n)$.

$$
\begin{array}{rcl}
sets\ s\ \varnothing & = & [\varnothing] \\
sets\ s\ (x{:}xs) & = & \begin{cases} s = x: & \{x\} : sets\ s\ xs \\ \text{otherwise}: & (sets\ s\ xs) \doubleplus [x{:}S | S \in sets\ (s - x)\ xs] \end{cases}
\end{array}
\tag{14.63}
$$

There is sub-structure and overlapping sub-problems in above exhaustive search definition, we can apply dynamic programming method. We bottom-up build solution table T, and generate the final subset. First consider the existence of some subset S, satisfying $\sum S = s$. We scan the elements to determine the bottom/up bound of the subset sum $l \leq s \leq u$. if $s < l$ or $s > u$, then there's no solution.

$$
l = \sum\{x \in X, x < 0\}, u = \sum\{x \in X, x > 0\} \tag{14.64}
$$

As the elements are integers, there are $m = u - l + 1$ columns in table T, each corresponds to a value: $l \leq j \leq u$. There are $n = |X| + 1$ rows, each corresponds to some element x_i. $T[i][j]$ indicates whether exists some subset $S \subseteq \{x_1, x_2, ..., x_i\}$, satisfying $\sum S = j$. Row 0 is special, represents the sum of empty set $\varnothing$. All entries in T start from false F except $T[0][0] =$ T, meaning $\sum \varnothing = 0$. Start from x_1 to build row 1. Besides $\sum \varnothing = 0$, $\sum\{x_1\} = x_1$, hence $T[1][0] =$T, $T[1][x_1] =$T.

	l	$l+1$	...	0	...	x_1	...	u
$\varnothing$	F	F	...	T	...	F	...	F
x_1	F	F	...	T	...	T	...	F
...	F	F	...	T	...	T	...	F

Add x_2, we get 4 possible subset sums: $\sum\varnothing = 0$, $\sum\{x_1\} = x_1$, $\sum\{x_2\} = x_2$, $\sum\{x_1, x_2\} = x_1 + x_2$.

	l	$l+1$	...	0	...	x_1	...	x_2	...	x_1+x_2	...	u
$\varnothing$	F	F	...	T	...	T	...	F	...	F	...	F
x_1	F	F	...	T	...	T	...	F	...	F	...	F
x_2	F	F	...	T	...	T	...	T	...	T	...	F
...	F	F	...	T	...	T	...	T	...	T	...	F

We add element x_i to fill row i. We can obtain all subset sums from previous elements: $\{x_1, x_2, ..., x_{i-1}\}$, hence all the entries of true in previous row are still true. Because $\sum\{x_i\} = x_i$, hence $T[i][x_i] = $ T. We add x_i to each previous sum, generate some new sums, the corresponding entries of them are all true. After add all n elements, the Boolean value of $T[n][s]$ gives whether the subset sum s exists.

```
function Subset-Sum(X, s)
    l ← ∑{x ∈ X, x < 0}, u ← ∑{x ∈ X, x > 0}
    n ← |X|
    T ← {{F, F, ...}, {F, F, ...}, ...}            ▷ (n + 1) × (u − l + 1)
    T[0][0] ← T                                    ▷ ∑∅ = 0
    for i ← 1 to n do
        T[i][X[i]] ←T
        for j ← l to u do
            T[i][j] ← T[i][j] ∨ T[i − 1][j]
            j′ ← j − X[i]
            if l ≤ j′ ≤ u then
                T[i][j] ← T[i][j] ∨ T[i − 1][j′]
    return T[n][s]
```

The column index j does not start from 0, but from l to u. We can convert it by $j - l$ in programming environment. We next generate all subsets S satisfying $\sum S = s$ from table T. If $T[n][s]$ =F then there's no solution; otherwise, there are two cases: (1) if $x_n = s$, then the singleton set $\{x_n\}$ is a solution. We next lookup $T[n-1][s]$, if it's true T, then recursively generate all subsets from $\{x_1, x_2, x_3, ..., x_{n-1}\}$ that the sum is s. (2) let $s' = s - x_n$, if $l \leq s' \leq u$ and $T[n-1][s']$ is true, we recursively generate subsets from $\{x_1, x_2, x_3, ..., x_{n-1}\}$ that the sum is s', then add x_n to each subset.

```
function Get(X, s, T, n)
    r ← [ ]
    if X[n] = s then
        r ← {X[n]} : r
    if n > 1 then
        if T[n − 1][s] then
            r ← r ++ Get(X, s, T, n − 1)
        s′ ← s − X[n]
        if l ≤ s′ ≤ u and T[n − 1][s′] then
```

$\qquad\qquad r \leftarrow r \mathbin{+\!\!+} [(X[n]{:}r') | r' \leftarrow \text{GET}(X, s', T, n-1)\ ]$

$\qquad$ **return** r

The dynamic programming method loops $O(n(u-l+1))$ times to build table T, then recursively generate the solution in $O(n)$ levels. The 2D table need $O(n(u-l+1))$ space. We can replace it with a 1D vector V of $u-l+1$ entries. each $V[j] = \{S_1, S_2, ...\}$ stores the subsets that $\sum S_1 = \sum S_2 = ... = j$. V start from all empty entries. For each x_i, we update V a round, add the new obtained sums with x_i. The final solution is in $V[s]$.

function SUBSET-SUM(X, s)
$\quad l \leftarrow \sum\{x \in X, x < 0\}, u \leftarrow \sum\{x \in X, x > 0\}$
$\quad V \leftarrow [\varnothing, \varnothing, ...]$ $\qquad \triangleright\ u - l + 1$
$\quad$ **for** each x in X **do**
$\qquad U \leftarrow \text{COPY}(V)$
$\qquad$ **for** $j \leftarrow l$ to u **do**
$\qquad\quad$ **if** $x = j$ **then**
$\qquad\qquad U[j] \leftarrow \{\{x\}\} \cup U[j]$
$\qquad\quad j' \leftarrow j - x$
$\qquad\quad$ **if** $l \leq j' \leq u$ and $V[j'] \neq \varnothing$ **then**
$\qquad\qquad U[j] \leftarrow U[j] \cup \{(\{x\} \cup S) | S \in V[j']\}$
$\qquad V \leftarrow U$
$\quad$ **return** $V[s]$

We can build the solution vector with left fold: $V = foldl\ bld\ (replicate(u-l+1)\ \varnothing)\ X$, where *replicate* n a generates list $[a, a, ..., a]$ of length n. *bld* updates V with each elements in X.

$$bld\ V\ x = foldl\ f\ V\ [l, l+1..., u] \tag{14.65}$$

Where:

$$f\ V\ j = \begin{cases} j = x : & V[j] \cup \{\{x\}\} \\ l \leq j' \leq u \text{ and } T[j'] \neq \varnothing : & V[j] \cup \{\{x\} S | S \in T[j']\}, \text{where} : j' = j - x \\ \text{otherwise} : & V \end{cases} \tag{14.66}$$

Exercise 14.12

1. For the longest common sub-sequence problem, an alternative solution is to record the direction in the table. There are three directions: 'N' for north, 'W' for west, and 'NW'. Given such a table, we can build the longest common sub-sequence from the bottom-right entry. If the entry is 'NW', next go to the upper-left entry; if it's 'N', go to the above row; and go to the previous entry if it's 'W'. Implement this solution.
2. For the subset sum upper/lower bound, does $l \leq 0 \leq u$ always hold? can we reduce the range between the bounds?
3. Given a list of non-negative integers, find the maximum sum composed by numbers that none of them are adjacent.
4. Edit distance (also known as 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 symbol changes: insert, delete, and replace. Each operation mutate a character a time. For example the edit distance is 3 for 'kitten' $\mapsto$ 'sitting':

    1. **k**itten $\rightarrow$ **s**itten (k $\mapsto$ s);

2. sitt**e**n → sitt**i**n (e ↦ i);
3. sittin → sittin**g** (+ g).

Compute the edit distance with dynamic programming.

14.7 Appendix - example programs

Find the top-k element:

```
Optional<K> top(Int k, [K] xs, Int l, Int u) {
    if l < u {
        swap(xs, l, rand(l, u))
        var p = partition(xs, l, u)
        if p - l + 1 == k
            return Optional.of(xs[p])
        return if k < p - l + 1 then top(k, xs, l, p)
               else top(k- p + l - 1, xs, p + 1, u)
    }
    return Optional.Nothing
}

Int partition([K] xs, Int l, Int u) {
    var p = l
    for var r = l + 1 to u {
        if not xs[p] < xs[r] {
            l = l + 1
            swap(xs, l, r)
        }
    }
    swap(xs, p, l)
    return l
}
```

Saddle back search:

```
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
  m = bsearch (f 0) z (0, z)
  n = bsearch (λx→f x 0) z (0, z)

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
```

Boyer-Moore majority:

```
Optional<T> majority([T] xs) {
    var (m, c) = (Optional<T>.Nothing, 0)
    for var x in xs {
        if c == 0 then (m, c) = (Optional.of(x), 0)
        if x == m then c++ else c--
    }
    c = 0
    for var x in xs {
        if x == m then c++
    }
    return if c > length(xs)/2 then m else Optional<T>.Nothing
}
```

Find the majority with fold:

```
majority xs = verify $ foldr maj (Nothing, 0) xs where
  maj x (Nothing, 0) = (Just x, 1)
  maj x (Just y, v) | x == y = (Just y, v + 1)
                    | v == 0 = (Just x, 1)
                    | otherwise = (Just y, v - 1)
  verify (Nothing, _) = Nothing
  verify (Just m, _)  = if 2 * (length $ filter (==m) xs) > length xs
                        then Just m else Nothing
```

The maximum sum of sub-vector:

```
maxSum :: (Ord a, Num a) => [a] -> a
maxSum = fst ∘ foldr f (0, 0) where
  f x (m, mSofar) = (m', mSofar') where
    mSofar' = max 0 (mSofar + x)
    m' = max mSofar' m
```

KMP string matching:

```
[Int] match([T] w, [T]p) {
    n = length(w), m = length(p)
    [Int] fallback = prefixes(p)
    [Int] r = []
    Int k = 0
    for i = 0 to n {
        while k > 0 and p[k] ≠ w[i] {
            k = fallback[k]
        }
        if p[k] == w[i] then k = k + 1
        if k == m {
            add(r, i + 1 - m)
            k = fallback[k - 1]
        }
    }
    return r
}

[Int] prefixes([T] p) {
    m = length(p)
    [Int] t = [0] * m  //fallback table
    Int k = 0
    for i = 2 to m {
        while k > 0 and p[i-1] ≠ p[k] {
            k = t[k-1] #fallback
        }
        if p[i-1] == p[k] then k = k + 1
        t[i] = k
    }
    return t
}
```

The maze puzzle:

```
dfsSolve m from to = solve [[from]] where
  solve [] = []
  solve (c@(p:path):cs)
      | p == to = reverse c
      | otherwise = let os = filter (`notElem` path) (adj p) in
                    if os == [] then solve cs
                    else solve ((map (:c) os) ++ cs)
  adj (x, y) = [(x', y') | (x', y') <- [(x-1, y), (x+1, y), (x, y-1), (x, y+1)],
                         inRange (bounds m) (x', y'), m ! (x', y') == 0]
```

The eight queens puzzle:

```
solve = dfsSolve [[]] [] where
  dfsSolve [] s = s
  dfsSolve (c:cs) s
           | length c == 8 = dfsSolve cs (c:s)
           | otherwise = dfsSolve ([(x:c) | x <- [1..8] \\ c,
                                   not $ attack x c] ++ cs) s
  attack x c = let y = 1 + length c in
                 any (λ(i, j) -> abs(x - i) == abs(y - j)) $
                     zip (reverse c) [1..]
```

The peg puzzle:

```
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
```

Iterative solution to the peg puzzle:

```
[Int] solve([Int] start, [Int] end) {
    stack = [[start]]
    s = []
    while stack ≠ [] {
        c = pop(stack)
        if c[0] == end {
            s += reverse(c)
        } else {
            for [Int] m in moves(c[0]) {
                stack += (m:c)
            }
        }
    }
    return s
}

[[Int]] moves([Int] s) {
    [[Int]] ms = []
    n = length(s)
    p = find(s, 0)
    if p < n - 2 and s[p+2] > 0 then ms += swap(s, p, p+2)
    if p < n - 1 and s[p+1] > 0 then ms += swap(s, p, p+1)
    if p > 1 and s[p-2] < 0 then ms += swap(s, p, p-2)
    if p > 0 and s[p-1] < 0 then ms += swap(s, p, p-1)
    return ms
}

[Int] swap([Int] s, Int i, Int j) {
    a = copy(s)
    (a[i], a[j]) = (a[j], a[i])
    return a
```

```
}
```

The wolf, goat, cabbage cross river puzzle:

```
import Data.Bits
import qualified Data.Sequence as Queue
import Data.Sequence (Seq((:<|)), (><))

solve = bfsSolve $ Queue.singleton [(15, 0)] where
  bfsSolve Queue.Empty = [] -- no solution
  bfsSolve (c@(p:_) :<| cs)
    | fst p == 0 = reverse c
    | otherwise = bfsSolve (cs >< (Queue.fromList $ map (:c)
                                   (filter (`valid` c) $ moves p)))

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)
```

The extended Euclid algorithm to solve the water jugs puzzle:

```
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 = []
            | 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)
                             | b' == b = pour x ((a', 0):ps)
                             | otherwise = pour x ((max 0 (a' + b' - b),
                                                    min (a' + b') b):ps)
```

BFS solution to the water jugs puzzle:

```
import qualified Data.Sequence as Queue
import Data.Sequence (Seq((:<|)), (><))

solve' a b g = bfs $ Queue.singleton [(0, 0)] where
  bfs Queue.Empty = []
  bfs (c@(p:_) :<| cs)
    | fst p == g || snd p == g = reverse c
    | otherwise = bfs (cs >< (Queue.fromList $ map (:c) $ expand c))
  expand ((x, y):ps) = filter (`notElem` ps) $ map (\f -> 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))
```

Iterative BFS for water jugs puzzle:

```
data Step {
    Pair<Int> (p, q)
    Step parent
    Step(Pair<Int>(x, y), Step p = null) {
      (p, q) = (x, y), parent = p
```

```
    }
}

Bool (==) (Step a, Step b) = {a.(p, q) == b.(p, q)}
Bool (≠) (Step a, Step b) = not ∘ (==)

[Step] expand(Step s, Int a, Int b) {
    var (p, q) = s.(p, q)
    return [Step(a, q, s), /*fill A*/
            Step(p, b, s), /*fill B*/
            Step(0, q, s), /*empty A*/
            Step(p, 0, s), /*empty B*/
            Step(max(0, p + q - b), min(p + q, b), s), /*pour A into B*/
            Step(min(p + q, a), max(0, p + q - a), s)] /*pour B into A*/
}

Optional<[Step]> solve(Int a, Int b, Int g) {
    q = Queue<Step>(Step(0, 0))
    Set<Step> visited = {head(q)}
    while not empty(q) {
        var cur = pop(q)
        if cur.p == g || cur.q == g {
            return Optional.of(backtrack(cur))
        } else {
            for s in expand(cur, a, b) {
                if cur ≠ s and s not in visited {
                    push(q, s)
                    visited += s
                }
            }
        }
    }
    return Optional.Nothing
}

[Step] backtrack(Step s) {
    [Step] seq
    while s ≠ null {
        seq = s : seq
        s = s.parent
    }
    return seq
}
```

Klotski puzzle:

```
import qualified Data.Map as Map
import qualified Data.Set as Set
import qualified Data.Sequence as Queue
import Data.Sequence (Seq((:<|)), (><))

cellOf (y, x) = y * 4 + x
posOf c = (c `div` 4, c `mod` 4)

cellSet = Set.fromList ∘ (map cellOf)

type Layout = Map.Map Integer (Set.Set Integer)
type NormLayout = Set.Set (Set.Set Integer)
type Move = (Integer, Integer)

start = Map.map cellSet $ Map.fromList
        [(1, [(0, 0), (1, 0)]),
         (2, [(0, 3), (1, 3)]),
         (3, [(2, 0), (3, 0)]),
         (4, [(2, 1), (2, 2)]),
         (5, [(2, 3), (3, 3)]),
```

```
        (6, [(4, 0)]), (7, [(3, 1)]), (8, [(3, 2)]), (9, [(4, 3)]),
        (10, [(0, 1), (0, 2), (1, 1), (1, 2)])]

end = cellSet [(3, 1), (3, 2), (4, 1), (4, 2)]

normalize = Set.fromList ∘ Map.elems

mirror = Map.map (Set.map f) where
  f c = let (y, x) = posOf c in cellOf (y, 3 - x)

klotski = solve q visited where
  q = Queue.singleton (start, [])
  visited = Set.singleton (normalize start)

solve Queue.Empty _ = []
solve ((x, ms) :<| cs) visited | Map.lookup 10 x == Just end = reverse ms
                               | otherwise = solve q visited'
  where
    q = cs >< (Queue.fromList [(move x op, op:ms) | op ← ops ])
    visited' = foldr Set.insert visited (map (normalize ∘ move x) ops)
    ops = expand x visited

expand x visited = [(i, d) | i ←[1..10], d ← [-1, 1, -4, 4],
                             valid i d, unique i d]
  where
    valid i d = let p = trans d (maybe Set.empty id $ Map.lookup i x) in
                (not $ any (outside d) p) &&
                (Map.keysSet $ Map.filter (overlapped p) x)
                    `Set.isSubsetOf` Set.singleton i
    outside d c = c < 0 || c ≥ 20 ||
                (d == 1 && c `mod` 4 == 0) || (d == -1 && c `mod` 4 == 3)
    unique i d = let ly = move x (i, d) in all (`Set.notMember` visited)
                 [normalize ly, normalize (mirror ly)]

move x (i, d) = Map.update (Just ∘ trans d) i x

trans d = Set.map (d+)

overlapped :: (Set.Set Integer) → (Set.Set Integer) → Bool
overlapped a b = (not ∘ Set.null) $ Set.intersection a b
```

Iterative solution to the Klotski puzzle:

```
type Layout = [Set<Int>]

Layout START = [{0, 4}, {3, 7}, {8, 12}, {9, 10},
    {11, 15},{16},{13}, {14}, {19}, {1, 2, 5, 6}]

Set<Int> END = {13, 14, 17, 18}

(Int, Int) pos(Int c) = (y = c / 4, x = c mod 4)

[[Int]] matrix(Layout layout) {
    [[Int]] m = replicate(replicate(0, 4), 5)
    for Int i, var p in (zip([1, 2, ...], layout)) {
        for var c in p {
            y, x = pos(c)
            m[y][x] = i
        }
    }
    return m
}

data Node {
    Node parent
    Layout layout
```

```
    Node(Layout l, Node p = null) {
        layout = l, parent = p
    }
}

//usage: solve(START, END)
Optional<Node> solve(Layout start, Set<Int> end) {
    var visit = {Set(start)}
    var queue = Queue.of(Node(start))
    while not empty(queue) {
        cur = pop(queue)
        if last(cur.layout) == end {
            return Optional.of(cur)
        } else {
            for ly in expand(cur.layout, visit) {
                push(queue, Node(ly, cur))
                add(visit, Set(ly))
            }
        }
    }
    return Optional.None
}

[Layout] expand(Layout layout, Set<Set<Layout>> visit):
    Bool bound(Set<Int> piece, Int d) {
        for c in piece {
            if c + d < 0 or c + d ≥ 20 then return False
            if d == 1 and c mod 4 == 3 then return False
            if d == -1 and c mod 4 == 0 then return False
        }
        return True
    }

    var m = matrix(layout)
    Bool valid(Set<Int> piece, Int d, Int i) {
        for c in piece {
            y, x = pos(c + d)
            if m[y][x] not in [0, i] then return False
        }
        return True
    }

    Bool unique(Layout ly) {
        n = Set(ly)
        Set<Set<Int>> m = map(map(c → 4 ∗ (c / 4) + 3 - (c mod 4), p), n)
        return (n not in visit) and (m not in visit)
    }

    [Layout] s = []
    for i, p in zip([1, 2, ...], layout) {
        for d in [-1, 1, -4, 4] {
            if bound(p, d) and valid(p, d, i) {
                ly = move(layout, i - 1, d)
                if unique(ly) then s.append(ly)
            }
        }
    }
    return
}

Layout move(Layout layout, Int i, Int d) {
    ly = clone(layout)
    ly[i] = map((d+), layout[i])
    return ly
```

```
}
```

Code, decode with a Huffman tree:

```
code = Map.fromList ∘ (traverse []) where
  traverse bits (Leaf _ c) = [(c, reverse bits)]
  traverse bits (Branch _ l r) = traverse (0:bits) l ++ traverse (1:bits) r

encode dict = concatMap (dict !)

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
```

Greedy change-making:

```
import qualified Data.Set as Set
import Data.List (group)

solve x = assoc ∘ change x where
  change 0 _ = []
  change x cs = let c = Set.findMax $ Set.filter (≤ x) cs in c : change (x - c) cs
  assoc = (map (λcs → (head cs, length cs))) ∘ group

example = solve 142 $ Set.fromList [1, 5, 25, 50, 100]
```

Dynamic programming change-making:

```
[Int] change(Int x, Set<Int> cs) {
    t = [(0, None)] ++ [(x + 1, None)] * x
    for i = 1 to x {
        for c in cs {
            if c ≤ i {
                (n, _) = t[i - c]
                (m, _) = t[i]
                if 1 + n < m then t[i] = (1 + n, c)
    s = []
    while x > 0:
        (_, c) = t[x]
        s += c
        x = x - c
    return s
}
```

Dynamic programming with fold to solve the change-making problem:

```
import qualified Data.Set as Set
import Data.Sequence ((|>), singleton, index)

changemk x cs = makeChange x $ foldl fill (singleton (0, 0)) [1..x] where
  fill tab i = tab |> (n, c) where
    (n, c) = minimum $ Set.map lookup $ Set.filter (≤ i) cs
    lookup c  = (1 + fst (tab `index` (i - c)), c)
  makeChange 0 _ = []
  makeChange x tab = let c = snd $ tab `index` x in c : makeChange (x - c) tab
```

The longest common sub-sequence:

```
[K] lcs([K] xs, [K] ys) {
    Int m = length(xs), n = length(ys)
    [[Int]] c = [[0]*(n + 1)]*(m + 1)
    for i = 1 to m {
        for j = 1 to n {
            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 fetch(c, xs, ys)
}

[K] fetch([[Int]] c, [K] xs, [K] ys) {
    [K] r = []
    var m = length(xs), n = length(ys)
    while m > 0 and n > 0 {
        if xs[m - 1] == ys[n - 1] {
            r += xs[m - 1]
            m = m - 1
            n = n - 1
        } else if c[m - 1][n] > c[m][n - 1] {
            m = m - 1
        } else {
            n = n - 1
        }
    }
    return reverse(r)
}
```

Existence of the subset sum:

```
Bool subsetsum([Int] xs, Int s) {
    Int l = 0, u = 0, n = length(xs)
    for x in xs {
      if x > 0 then u++ else l++
    }
    tab = [[False]*(u - l + 1)] * (n + 1)
    tab = [0][0 - l] = True
    for i, x in zip([1, 2, ..., n], xs) {
        tab[i][x - l] = True
        for j = l to u {
            tab[i][j - l] or = tab[i-1][j - l]
            j1 = j - x
            if l ≤ j1 ≤ u then tab[i][j - l] or = tab[i-1][j1 - l]
        }
    }
    return tab[n][s - l]
}
```

Solve the subset sum with a vector:

```
{{Int}} subsetsum(xs, s) {
    Int l = 0, u = 0, n = length(xs)
    for x in xs {
      if x > 0 then u++ else l++
    }
    tab = {} * (u - l + 1)
    for x in xs {
        tab1 = copy(tab)
        for j = low to up {
            if x == j then add(tab1[j], {x})
            j1 = j - x
            if low ≤ j1 ≤ up and tab[j1] {
                tab1[j] |= {add(ys, x) for ys in tab[j1]}
            }
        }
        tab = tab1
    }
    return tab[s]
}
```

Appendix A

Imperative delete for red-black tree

We need handle more cases for imperative *delete* than *insert*. To resume balance after cutting off a node fro the red-black tree, we perform rotations and re-coloring. When delete a black node, rule 5 will be violated because the number of black nodes along the path through that node reduces by one. We introduce 'doubly-black' to maintain the number of black nodes unchanged. Below example program adds 'doubly black' to the color definition:

```
data Color {RED, BLACK, DOUBLY_BLACK}
```

When delete a node, we re-use the binary search tree *delete* in the first step, then further fix the balance if the node is black.

```
function DELETE(T, x)
    p ← PARENT(x)
    q ← NIL
    if LEFT(x) = NIL then
        q ← RIGHT(x)
        REPLACE(x, RIGHT(x))                 ▷ replace x with its right sub-tree
    else if RIGHT(x) = NIL then
        q ← LEFT(x)
        REPLACE(x, LEFTx())                  ▷ replace x with its left sub-tree
    else
        y ← MIN(RIGHT(x))
        p ← PARENT(y)
        q ← RIGHT(y)
        KEY(x) ← KEY(y)
        copy data from y to x
        REPLACE(y, RIGHT(y))                 ▷ replace y with its right sub-tree
        x ← y
    if COLOR(x) = BLACK then
        T ← DELETE-FIX(T, MAKE-BLACK(p, q), q = NIL?)
    release x
    return T
```

DELETE takes the root T and the node x to be deleted as the parameters. x can be located through *lookup*. If x has an empty sub-tree, we cut off x, then replace it with the other sub-tree q. Otherwise, we locate the minimum node y in the right sub-tree of

x, then replace x with y. We cut off y after that. If x is black, we call MAKE-BLACK(p, q) to maintain the blackness before further fixing.

function MAKE-BLACK(p, q)
  if p = NIL and q = NIL **then**
    return NIL ▷ The tree was singleton
  else if q = NIL **then**
    $n \leftarrow$ Doubly Black NIL
    PARENT(n) $\leftarrow p$
    return n
  else
    return BLACKEN(q)

If both p and q are empty, we are deleting the only leaf from a singleton tree. The result is empty. If the parent p is not empty, but q is, we are deleting a black leaf. We use NIL to replace that black leaf. As NIL is already black, we change it to 'doubly black' NIL to maintain the blackness. Otherwise, if neither p nor q is empty, we call BLACKEN(q). If q is red, it changes to black; if q is already black, it changes to doubly black. As the next step, we need eliminate the doubly blackness through tree rotations and re-coloring. There are three different cases ([4], pp292). The doubly black node can be NIL or not in all the cases.

Case 1. *The sibling of the doubly black node is black, and it has a red sub-tree.* 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 A.1.

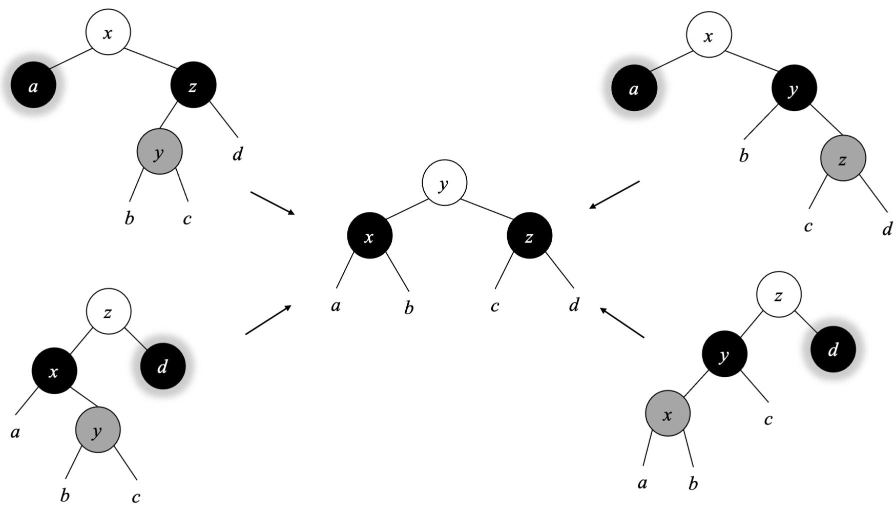


Figure A.1: The doubly black node has a black sibling, and a red nephew. It can be fixed with a rotation.

function DELETE-FIX(T, x, f)
  $n \leftarrow$ NIL
  if f = True **then** ▷ x is doubly black NIL
    $n \leftarrow x$
  if x = NIL **then** ▷ Delete the singleton leaf
    return NIL
  while $x \neq T$ and COLOR(x) = $\mathcal{B}^2$ **do** ▷ x is doubly black, but not the root
    if SIBLING(x) $\neq$ NIL **then** ▷ The sibling is not empty
      $s \leftarrow$ SIBLING(x)

```
            ...
        if s is black and LEFT(s) is red then
            if x = LEFT(PARENT(x)) then                          ▷ x is the left
                set x, PARENT(x), and LEFT(s) all black
                T ← ROTATE-RIGHT(T, s)
                T ← ROTATE-LEFT(T, PARENT(x))
            else                                                 ▷ x is the right
                set x, PARENT(x), s, and LEFT(s) all black
                T ← ROTATE-RIGHT(T, PARENT(x))
        else if s is black and RIGHT(s) is red then
            if x = LEFT(PARENT(x)) then                          ▷ x is the left
                set x, PARENT(x), s, and RIGHT(s) all black
                T ← ROTATE-LEFT(T, PARENT(x))
            else                                                 ▷ x is the right
                set x, PARENT(x), and RIGHT(s) all black
                T ← ROTATE-LEFT(T, s)
                T ← ROTATE-RIGHT(T, PARENT(x))
            ...
```

Case 2. *The sibling of the doubly black is red.* We can rotate the tree to change the doubly black node to black. As shown in figure A.2, change a or c to black. We can add this fixing to the previous implementation.

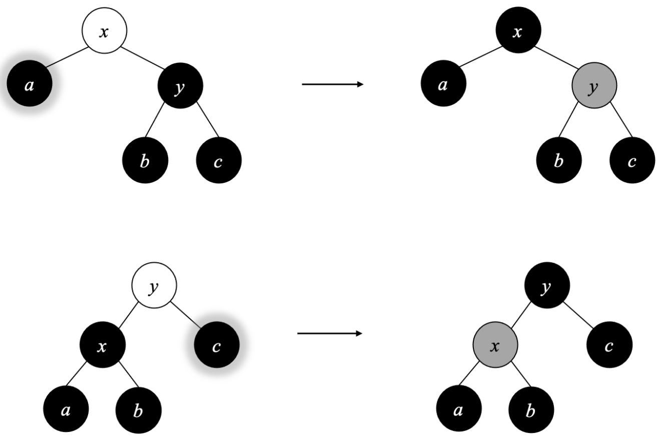


Figure A.2: The sibling of the doubly black is red

```
function DELETE-FIX(T, x, f)
    n ← NIL
    if f = True then                                   ▷ x is doubly black NIL
        n ← x
    if x = NIL then                                    ▷ Delete the singleton leaf
        return NIL
    while x ≠ T and COLOR(x) = B² do
        if SIBLING(x) ≠ NIL then
            s ← SIBLING(x)
            if s is red then                           ▷ The sibling is red
                set PARENT(x) red
                set s black
                if x = LEFT(PARENT(x)) then            ▷ x is the left
```

```
            T ← Rotate-LeftT, Parent(x)
        else                                        ▷ x is the right
            T ← Rotate-RightT, Parent(x)
    else if s is black and Left(s) is red then
        ...
```

Case 3. *The sibling of the doubly black node, and its two sub-trees are all black.* In this case, we re-color the sibling to red, change the doubly black node back to black, then move the doubly blackness up to the parent. As shown in figure A.3, there are two symetric sub-cases.

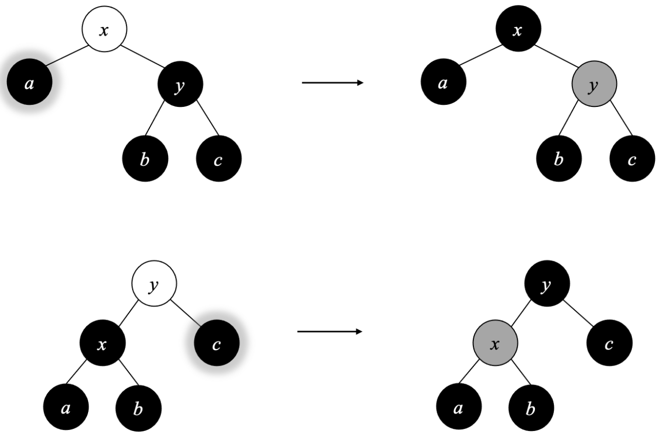


Figure A.3: move the blackness up

The sibling of the doubly black isn't empty in all above 3 cases. Otherwise, we change the doubly black node back to black, and move the blackness up. When reach the root, we force the root to be black to complete fixing. It also terminates if the doubly black node is eliminated after re-color in the midway. At last, if the doubly black node passed in is empty, we turn it back to normal NIL.

```
function Delete-Fix(T, x, f)
    n ← NIL
    if f = True then                                ▷ x is a doubly black NIL
        n ← x
    if x = NIL then                                 ▷ Delete the singleton leaf
        return NIL
    while x ≠ T and Color(x) = B² do
        if Sibling(x) ≠ NIL then                    ▷ The sibling is not empty
            s ← Sibling(x)
            if s is red then                        ▷ The sibling is red
                set Parent(x) red
                set s black
                if x = Left(Parent(x)) then         ▷ x is the left
                    T ← Rotate-LeftT, Parent(x)
                else                                ▷ x is the right
                    T ← Rotate-RightT, Parent(x)
            else if s is black and Left(s) is red then
                if x = Left(Parent(x)) then         ▷ x is the left
                    set x, Parent(x), and Left(s) all black
                    T ← Rotate-Right(T, s)
```

```
                T ← Rotate-Left(T, Parent(x))
            else                                              ▷ x is the right
                set x, Parent(x), s, and Left(s) all black
                T ← Rotate-Right(T, Parent(x))
        else if s is black and Right(s) is red then
            if x = Left(Parent(x)) then                       ▷ x is the left
                set x, Parent(x), s, and Right(s) all black
                T ← Rotate-Left(T, Parent(x))
            else                                              ▷ x is the right
                set x, Parent(x), and Right(s) all black
                T ← Rotate-Left(T, s)
                T ← Rotate-Right(T, Parent(x))
        else if s, Left(s), and Right(s) are all black then
            set x black
            set s red
            Blacken(Parent(x))
            x ← Parent(x)
    else                                                      ▷ move the blackness up
        set x black
        Blacken(Parent(x))
        x ← Parent(x)
set T black
if n ≠ NIL then
    replace n with NIL
return T
```

When fixing, we pass in the root T, the node x (can be doubly black), and a flag f. The flag is true if x is doubly black NIL. We record it with n, and replace n with the normal NIL after fixing.

Below is the example program implements delete:

```
Node del(Node t, Node x) {
    if x == null then return t
    var parent = x.parent;
    Node db = null;        //doubly black

    if x.left == null {
        db = x.right
        x.replaceWith(db)
    } else if x.right == null {
        db = x.left
        x.replaceWith(db)
    } else {
        var 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 == null);
    }
    remove(x)
    return t
}
```

Where `makeBlack` checks if the node changes to doubly black, and handles the special case of doubly black NIL.

```
Node makeBlack(Node parent, Node x) {
    if parent == null and x == null then return null
    return if x == null
        then replace(parent, x, Node(0, Color.DOUBLY_BLACK))
        else blacken(x)
}
```

The function `replace(parent, x, y)` replaces the child of the `parent`, which is `x`, with `y`.

```
Node replace(Node parent, Node x, Node y) {
    if parent == null {
        if y ≠ null then y.parent = null
    } else if parent.left == x {
        parent.setLeft(y)
    } else {
        parent.setRight(y)
    }
    if x ≠ null then x.parent = null
    return y
}
```

The function `blacken(node)` changes the red node to black, and the black node to doubly black:

```
Node blacken(Node x) {
    x.color = if isRed(x) then Color.BLACK else Color.DOUBLY_BLACK
    return x
}
```

Below example program implements the fixing:

```
Node deleteFix(Node t, Node db, Bool isDBEmpty) {
    var dbEmpty = if isDBEmpty then db else null
    if db == null then return null    // delete the root
    while (db ≠ t and db.color == Color.DOUBLY_BLACK) {
        var s = db.sibling()
        var p = db.parent
        if (s ≠ null) {
            if isRed(s) {
                // the sibling is red
                p.color = Color.RED
                s.color = Color.BLACK
                t = if db == p.left then leftRotate(t, p)
                                    else rightRotate(t, p)
            } else if isBlack(s) and isRed(s.left) {
                // the sibling is black, and one sub-tree is red
                if db == p.left {
                    db.color = Color.BLACK
                    p.color  = Color.BLACK
                    s.left.color = p.color
                    t = rightRotate(t, s)
                    t = leftRotate(t, p)
                } else {
                    db.color = Color.BLACK
                    p.color  = Color.BLACK
                    s.color = p.color
                    s.left.color = Color.BLACK
                    t = rightRotate(t, p)
                }
            } else if isBlack(s) and isRed(s.right) {
                if (db == p.left) {
```

```
                    db.color = Color.BLACK
                    p.color  = Color.BLACK
                    s.color  = p.color
                    s.right.color = Color.BLACK
                    t = leftRotate(t, p)
                } else {
                    db.color = Color.BLACK
                    p.color  = Color.BLACK
                    s.right.color = p.color
                    t = leftRotate(t, s)
                    t = rightRotate(t, p)
                }
            } else if isBlack(s) and isBlack(s.left) and
                      isBlack(s.right) {
                // the sibling and both sub-trees are black.
                // move blackness up
                db.color = Color.BLACK
                s.color  = Color.RED
                blacken(p)
                db = p
            }
        } else { // no sibling, move blackness up
            db.color = Color.BLACK
            blacken(p)
            db = p
        }
    }
    t.color = Color.BLACK
    if (dbEmpty ≠ null) { // change the doubly black nil to nil
        dbEmpty.replaceWith(null)
        delete dbEmpty
    }
    return t
}
```

Where `isBlack(x)` tests if a node is black, the NIL node is also black.

```
Bool isBlack(Node x) = (x == null or x.color == Color.BLACK)

Bool isRed(Node x) = (x ≠ null and x.color == Color.RED)
```

Before returning the final result, we check the doubly black NIL, and call the `replaceWith` function defined in `Node`.

```
data Node<T> {
    //...
    void replaceWith(Node y) = replace(parent, this, y)
}
```

The program terminates when reach the root or the doubly blackness is eliminated. As we maintain the red-black tree balanced, the delete algorithm is bound to $O(\lg n)$ time for the tree of n nodes.

Exercise A.1

1. Write a program to test if a tree satisfies the 5 red-black tree rules. Use this program to verify the red-black tree delete implementation.

Appendix B

AVL tree - proofs and the delete algorithm

I Height increment

When insert an element, the increment of the height can be deduced into 4 cases:

$$
\begin{array}{rcl}
\Delta H & = & |T'| - |T| \\
 & = & 1 + max(|r'|, |l'|) - (1 + max(|r|, |l|)) \\
 & = & max(|r'|, |l'|) - max(|r|, |l|) \\
 & = & \begin{cases} \delta \geq 0, \delta' \geq 0: & \Delta r \\ \delta \leq 0, \delta' \geq 0: & \delta + \Delta r \\ \delta \geq 0, \delta' \leq 0: & \Delta l - \delta \\ otherwise: & \Delta l \end{cases}
\end{array}
\tag{B.1}
$$

Proof. When insert, the height can not increase both on left and right. We can explain the 4 cases from the balance factor definition, which is the difference of the right and left sub-trees:

1. If $\delta \geq 0$ and $\delta' \geq 0$, it means the height of the right sub-tree is not less than the left sub-tree before and after insertion. In this case, the height increment is only 'contributed' from the right, which is Δr.

2. If $\delta \leq 0$, it means the height of left sub-tree is not less than the right before. Since $\delta' \geq 0$ after insert, we know the height of right sub-tree increases, and the left side keeps same ($|l'| = |l|$). The height increment is:

$$
\begin{array}{rll}
\Delta H & = max(|r'|, |l'|) - max(|r|, |l|) & \{\delta \leq 0 \text{ and } \delta' \geq 0\} \\
 & = |r'| - |l| & \{|l| = |l'|\} \\
 & = |r| + \Delta r - |l| & \\
 & = \delta + \Delta r &
\end{array}
$$

3. If $\delta \geq 0$ and $\delta' \leq 0$, similar to the above case, we have the following:

$$
\begin{array}{rll}
\Delta H & = max(|r'|, |l'|) - max(|r|, |l|) & \{\delta \geq 0 \text{ and } \delta' \leq 0\} \\
 & = |l'| - |r| & \\
 & = |l| + \Delta l - |r| & \\
 & = \Delta l - \delta &
\end{array}
$$

4. Otherwise, δ and δ' are not bigger than zero. It means the height of the left sub-tree is not less than the right. The height increment is only 'contributed' from the left, which is Δl.

□

II Balance adjustment after insert

The balance factors are ± 2 in the 4 cases shown in figure B.1. After fixing, $\delta(y)$ resumes to 0. The height of left and right sub-trees are equal.

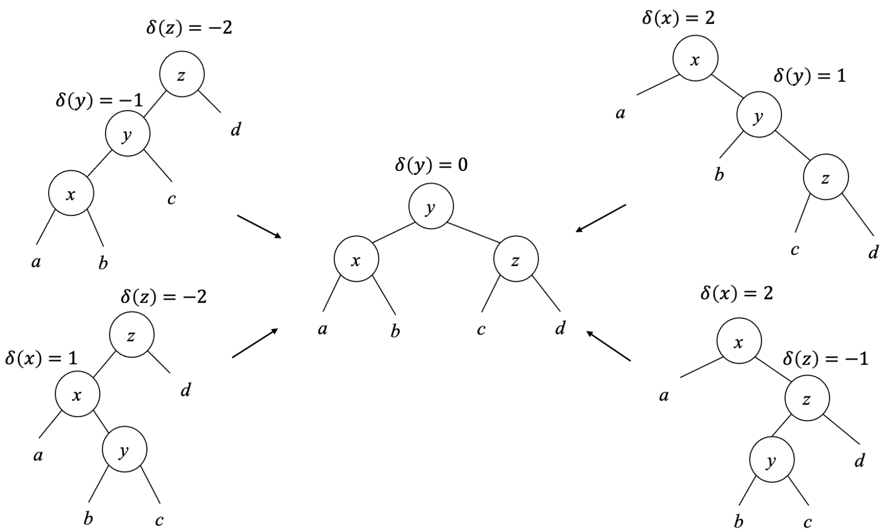


Figure B.1: Fix 4 cases to the same structure

The four cases are left-left, right-right, right-left, and left-right. Let the balance factors before fixing be $\delta(x), \delta(y)$, and $\delta(z)$, after fixing, they change to $\delta'(x), \delta'(y)$, and $\delta'(z)$ respectively. We next prove that, $\delta(y) = 0$ for all 4 cases after fixing, and give the result of $\delta'(x)$ and $\delta'(z)$.

Proof. We break into 4 cases:

Left-left

The sub-tree x keeps unchanged, hence $\delta'(x) = \delta(x)$. As $\delta(y) = -1$ and $\delta(z) = -2$, we have:

$$\begin{array}{lll} \delta(y) = |c| - |x| = -1 & \Rightarrow & |c| = |x| - 1 \\ \delta(z) = |d| - |y| = -2 & \Rightarrow & |d| = |y| - 2 \end{array} \tag{B.2}$$

After fixing:

$$\begin{array}{rcll} \delta'(z) & = & |d| - |c| & \{from(B.2)\} \\ & = & |y| - 2 - (|x| - 1) & \\ & = & |y| - |x| - 1 & \{x \text{ is sub-tree of } y \Rightarrow |y| - |x| = 1\} \\ & = & 0 & \end{array} \tag{B.3}$$

For $\delta'(y)$, we have the following:

$$\begin{array}{rcll} \delta'(y) & = & |z| - |x| & \\ & = & 1 + max(|c|, |d|) - |x| & \{\text{by (B.3)}, |c| = |d|\} \\ & = & 1 + |c| - |x| & \{\text{by (B.2)}\} \\ & = & 1 + |x| - 1 - |x| & \\ & = & 0 & \end{array} \tag{B.4}$$

Summarize the above, the balance factors change to the following in left-left case:

$$\begin{array}{l} \delta'(x) = \delta(x) \\ \delta'(y) = 0 \\ \delta'(z) = 0 \end{array} \tag{B.5}$$

Right-right

The right-right case is symmetric to left-left:

$$\begin{array}{l} \delta'(x) = 0 \\ \delta'(y) = 0 \\ \delta'(z) = \delta(z) \end{array} \tag{B.6}$$

Right-left

Consider $\delta'(x)$, after fixing, it is:

$$\delta'(x) = |b| - |a| \tag{B.7}$$

Before fixing, the height of z can be obtained as:

$$\begin{array}{rll} |z| & = 1 + max(|y|, |d|) & \{\delta(z) = -1 \Rightarrow |y| > |d|\} \\ & = 1 + |y| & \\ & = 2 + max(|b|, |c|) & \end{array} \tag{B.8}$$

Since $\delta(x) = 2$, we have:

$$\begin{array}{rll} \delta(x) = 2 & \Rightarrow |z| - |a| = 2 & \{\text{by (B.8)}\} \\ & \Rightarrow 2 + max(|b|, |c|) - |a| = 2 & \\ & \Rightarrow max(|b|, |c|) - |a| = 0 & \end{array} \tag{B.9}$$

If $\delta(y) = |c| - |b| = 1$, then:

$$max(|b|, |c|) = |c| = |b| + 1 \tag{B.10}$$

Take this into (B.9) gives:

$$\begin{array}{l} |b| + 1 - |a| = 0 \Rightarrow |b| - |a| = -1 \quad \{\text{by (B.7)} \} \\ \Rightarrow \delta'(x) = -1 \end{array} \tag{B.11}$$

If $\delta(y) \neq 1$, then $max(|b|, |c|) = |b|$. Take this into (B.9) gives:

$$\begin{array}{l} |b| - |a| = 0 \quad \{\text{by (B.7)}\} \\ \Rightarrow \delta'(x) = 0 \end{array} \tag{B.12}$$

Summarize the 2 cases, we obtain the result of $\delta'(x)$ in $\delta(y)$ as the following:

$$\delta'(x) = \begin{cases} \delta(y) = 1 : & -1 \\ otherwise : & 0 \end{cases} \tag{B.13}$$

For $\delta'(z)$, from the definition, it equals to:

$$\begin{array}{rcll} \delta'(z) & = & |d| - |c| & \{\delta(z) = -1 = |d| - |y|\} \\ & = & |y| - |c| - 1 & \{|y| = 1 + max(|b|, |c|)\} \\ & = & max(|b|, |c|) - |c| & \end{array} \tag{B.14}$$

If $\delta(y) = |c| - |b| = -1$, then $max(|b|, |c|) = |b| = |c| + 1$. Take this into (B.14), we have $\delta'(z) = 1$. If $\delta(y) \neq -1$, then $max(|b|, |c|) = |c|$. We have $\delta'(z) = 0$. Combined these two cases, we obtain the result of $\delta'(z)$ in $\delta(y)$ as below:

$$\delta'(z) = \begin{cases} \delta(y) = -1 : & 1 \\ otherwise : & 0 \end{cases} \tag{B.15}$$

Finally, for $\delta'(y)$, we deduce it like below:

$$\begin{array}{rl} \delta'(y) & = |z| - |x| \\ & = max(|c|, |d|) - max(|a|, |b|) \end{array} \tag{B.16}$$

There are three cases:

1. If $\delta(y) = 0$, then $|b| = |c|$. According to (B.13) and (B.15), we have $\delta'(x) = 0 \Rightarrow |a| = |b|$, and $\delta'(z) = 0 \Rightarrow |c| = |d|$. These lead to $\delta'(y) = 0$.

2. If $\delta(y) = 1$, from (B.15), we have $\delta'(z) = 0 \Rightarrow |c| = |d|$.

$$\begin{array}{rcll} \delta'(y) & = & max(|c|, |d|) - max(|a|, |b|) & \{|c| = |d|\} \\ & = & |c| - max(|a|, |b|) & \{\text{from (B.13)}: \delta'(x) = -1 \Rightarrow |b| - |a| = -1\} \\ & = & |c| - (|b| + 1) & \{\delta(y) = 1 \Rightarrow |c| - |b| = 1\} \\ & = & 0 & \end{array}$$

3. If $\delta(y) = -1$, from (B.13), we have $\delta'(x) = 0 \Rightarrow |a| = |b|$.

$$\begin{array}{rcll} \delta'(y) & = & max(|c|, |d|) - max(|a|, |b|) & \{|a| = |b|\} \\ & = & max(|c|, |d|) - |b| & \{\text{from (B.15)}: |d| - |c| = 1\} \\ & = & |c| + 1 - |b| & \{\delta(y) = -1 \Rightarrow |c| - |b| = -1\} \\ & = & 0 & \end{array}$$

All three cases lead to the same result $\delta'(y) = 0$. Summarize all above, we get the updated balance factors after fixing as below:

$$\begin{array}{l} \delta'(x) = \begin{cases} \delta(y) = 1 : & -1 \\ otherwise : & 0 \end{cases} \\ \delta'(y) = 0 \\ \delta'(z) = \begin{cases} \delta(y) = -1 : & 1 \\ otherwise : & 0 \end{cases} \end{array} \tag{B.17}$$

Left-right

Left-right is symmetric to the right-left case. With similar method, we can obtain the new balance factors that is identical to (B.17).

□

III Delete algorithm

Deletion may reduce the height of the sub-tree. If the balance factor exceeds the range of $[-1, 1]$, then we need fixing.

* Functional delete

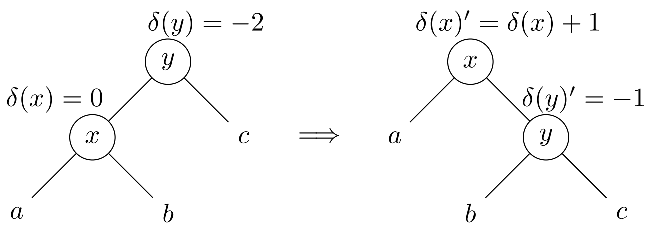
When delete, we re-use the binary search tree *delete* in the first step, then check the balance factors and perform fixing. The result is a pair $(T', \Delta H)$, where T' is the new tree and ΔH is the height decrement. We define *delete* as below:

$$delete = fst \circ del \tag{B.18}$$

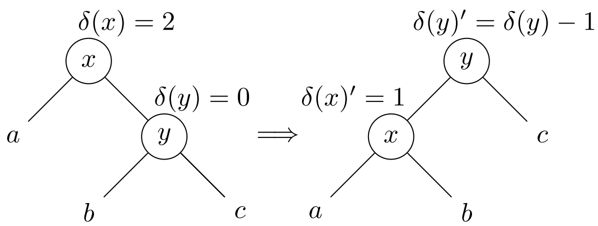
where $del(T, k)$ does the actual work to delete element k from T:

$$\begin{array}{rcl} del\ \varnothing\ k & = & (\varnothing, 0) \\ del\ (l, k', r, \delta) & = & \begin{cases} k < k' : & tree\ (del\ l\ k)\ k'\ (r, 0)\ \delta \\ k > k' : & tree\ (l, 0)\ k'\ (del\ r\ k)\ \delta \\ k = k' : & \begin{cases} l = \varnothing : & (r, -1) \\ r = \varnothing : & (l, -1) \\ else : & tree\ (l, 0)\ k''\ (del\ r\ k'')\ \delta \\ & \text{where } k'' = min(r) \end{cases} \end{cases} \end{array} \tag{B.19}$$

If the tree is empty, the result is $(\varnothing, 0)$; otherwise, let the tree be $T = (l, k', r, \delta)$. We compare the k and k', lookup and delete recursively. When $k = k'$, we locate the node to be deleted. If it has either empty sub-tree, we cut the node off, and replace it with the other sub-tree; otherwise, we use the minimum k'' in the right sub-tree to replace k', and cut k'' off. We re-use the *tree* function and ΔH result. Additional to the *insert* cases, there are two cases violate AVL rule, and need fixing. As shown in figure B.2, both cases can be fixed by a tree rotation. We define them as pattern matching:



(a) Fix case A



(b) Fix case B

Figure B.2: delete fix

$$\begin{array}{rcl} & \ldots & \\ balance\ ((a, x, b, \delta(x)), y, c, -2)\ \Delta H & = & (a, x, (b, y, c, -1), \delta(x) + 1, \Delta H) \\ balance\ (a, x, (b, y, c, \delta(y)), 2)\ \Delta H & = & ((a, x, b, 1), y, c, \delta(y) - 1, \Delta H) \\ & \ldots & \end{array} \tag{B.20}$$

Below is the example program:

```
delete t x = fst $ del t x where
  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
    | isEmpty l = (r, -1)
    | isEmpty r = (l, -1)
    | otherwise = node (l, 0) k' (del r k') d where k' = min r
```

Where `min` and `isEmpty` are defined as below:

```
min (Br Empty x _ _) = x
min (Br l _ _ _) = min l

isEmpty Empty = True
isEmpty _ = False
```

With the additional two, there are total 7 cases in `balance` implementation:

```
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
    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
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)
```

† Imperative delete

The imperative *delete* uses tree rotations for fixing. In the first step, we re-use the binary search tree algorithm to delete the node x from tree T; then in the second step, check the balance factor and perform rotation.

function DELETE(T, x)
  if $x =$ NIL **then**
    return T
  $p \leftarrow$ PARENT(x)
  if LEFT(x) = NIL **then**
    $y \leftarrow$ RIGHT(x)
    replace x with y
  else if RIGHT(x) = NIL **then**
    $y \leftarrow$ LEFT(x)
    replace x with y
  else
    $z \leftarrow$ MIN(RIGHT(x))
    copy data from z to x
    $p \leftarrow$ PARENT(z)
    $y \leftarrow$ RIGHT(z)

```
        replace z with y
    return AVL-DELETE-FIX(T, p, y)
```

When delete node x, we record its parent in p. If either sub-tree is empty, we cut off x, and replace it with the other sub-tree. Otherwise if neither sub-tree is empty, we locate the minimum element z of the right sub-tree, copy data from z to x, then cut z off. Finally, we call AVL-Delete-Fix with the root T, the parent p, and the replacement node y. Let the balance factor of p be $\delta(p)$, and it changes to $\delta(p)'$ after delete. There are three cases:

1. $|\delta(p)| = 0$, $|\delta(p)'| = 1$. After delete, although a sub-tree height decreases, the parent still satisfies the AVL rule. The algorithm terminates as the tree is still balanced;

2. $|\delta(p)| = 1$, $|\delta(p)'| = 0$. Before the delete, the height difference between the two sub-trees is 1; while after delete, the higher sub-tree shrinks by 1. Both sub-trees have the same height now. As the result, the height of the parent also decrease by 1. We need continue the bottom-up update along the parent reference to the root;

3. $|\delta(p)| = 1$, $|\delta(p)'| = 2$. After delete, the tree violates the AVL height rule, we need rotate the tree to fix it.

For case 3, the implementation is similar to the insert fixing. We need add two additional sub-cases as shown in figure B.2.

```
function AVL-DELETE-FIX(T, p, x)
    while p ≠ NIL do
        l ← LEFT(p), r ← RIGHT(p)
        δ ← δ(p), δ' ← δ
        if x = l then
            δ' ← δ' + 1
        else
            δ' ← δ' − 1
        if p is leaf then                                  ▷ l = r = NIL
            δ' ← 0
        if |δ| = 1 ∧ |δ'| = 0 then
            x ← p
            p ← PARENT(x)
        else if |δ| = 0 ∧ |δ'| = 1 then
            return T
        else if |δ| = 1 ∧ |δ'| = 2 then
            if δ' = 2 then
                if δ(r) = 1 then                           ▷ Right-right
                    δ(p) ← 0
                    δ(r) ← 0
                    p ← r
                    T ← LEFT-ROTATE(T, p)
                else if δ(r) = −1 then                     ▷ Right-left
                    δ_y ← δ( LEFT(r) )
                    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
        else                                        ▷ Delete specific right-right
            δ(p) ← 1
            δ(r) ← δ(r) − 1
            T ← LEFT-ROTATE(T, p)
            break                                   ▷ No furthur height change
    else if δ′ = −2 then
        if δ(l) = −1 then                           ▷ Left-left
            δ(p) ← 0
            δ(l) ← 0
            p ← l
            T ← RIGHT-ROTATE(T, p)
        else if δ(l) = 1 then                       ▷ Left-right
            δ_y ← δ( RIGHT(l) )
            if δ_y = −1 then
                δ(p) ← 1
            else
                δ(p) ← 0
            δ( RIGHT(l) ) ← 0
            if δ_y = 1 then
                δ(l) ← −1
            else
                δ(l) ← 0
        else                                        ▷ Delete specific left-left
            δ(p) ← −1
            δ(l) ← δ(l) + 1
            T ← RIGHT-ROTATE(T, p)
            break                                   ▷ No furthur height change
                                ▷ Height decreases, go on bottom-up updating
    x ← p
    p ← PARENT(x)
if p = NIL then                                     ▷ Delete the root
    return x
return T
```

Exercise B.1

1. Compare the imperative tree fixing for *insert* and *delete*, there are similarities. Develop a common fix function for both *insert* and *delete*.

IV Example program

The main *delete* program:

```
Node del(Node t, Node x) {
    if x == null then return t
    Node y
    var parent = x.parent
    if x.left == null {
        y = x.replaceWith(x.right)
```

```
    } else if x.right == null {
        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)
    release(x)
    return t
}
```

Where `replaceWith` is defined in the chapter of red-black tree. `release(x)` releases the memory of a node. Function `deleteFix` is implemented as below:

```
Node deleteFix(Node t, Node parent, Node x) {
    int d1, d2, dy
    Node p, l, r
    while parent ≠ null {
        d2 = d1 = parent.delta
        d2 = d2 + if x == parent.left then 1 else -1
        if isLeaf(parent) then d2 = 0
        parent.delta = d2
        p = parent
        l = parent.left
        r = parent.right
        if abs(d1) == 1 and abs(d2) == 0 {
            x = parent
            parent = x.parent
        } else if abs(d1) == 0 and abs(d2) == 1 {
            return t
        } else if abs(d1) == 1 and abs(d2) == 2 {
            if d2 == 2 {
                if r.delta == 1 {  // right-right
                    p.delta = 0
                    r.delta = 0
                    parent = r
                    t = leftRotate(t, p)
                } else if r.delta == -1 { // right-left
                    dy = r.left.delta
                    p.delta = if dy == 1 then -1 else 0
                    r.left.delta = 0
                    r.delta = if dy == -1 then 1 else 0
                    parent = r.left
                    t = rightRotate(t, r)
                    t = leftRotate(t, p)
                } else { // delete specific right-right
                    p.delta = 1
                    r.delta = r.delta - 1
                    t = leftRotate(t, p)
                    break // no further height change
                }
            } else if d2 == -2 {
                if (l.delta == -1) { // left-left
                    p.delta = 0
                    l.delta = 0
                    parent = l
                    t = rightRotate(t, p)
                } else if l.delta == 1 { // left-right
                    dy = l.right.delta
                    l.delta = if dy == 1 then -1 else 0
                    l.right.delta = 0
                    p.delta = if dy == -1 then 1 else 0
                    parent = l.right;
```

```
                    t = leftRotate(t, l)
                    t = rightRotate(t, p)
                } else { // delete specific left-left
                    p.delta = -1
                    l.delta = l.delta + 1
                    t = rightRotate(t, p)
                    break // no further height change
                }
            }
            // height decreases, go on bottom-up update
            x = parent
            parent = x.parent
        }
    }
    if parent == null then return x // delete the root
    return t
}
```

Appendix C

Answers

Answer of exercise 2.1

1. Given the in-order and pre-order traverse results, re-construct the tree, and output the post-order traverse result. For example:

   - Pre-order: 1, 2, 4, 3, 5, 6;
   - In-order: 4, 2, 1, 5, 3, 6;
   - Post-order: ?

   [4, 2, 5, 6, 3, 1]

2. Write a program to re-construct the binary tree from the pre-order and in-order traverse lists.

   Let P be the pre-order traverse result, I be the in-order result. If$P = I = [\ ]$, then the binary tree is empty $\varnothing$. Otherwise, the pre-order is recursive 'key - left - right', hence the first element m in P is the key of the root. The in-order is recursive 'left - key - right', we can find m in I, which splits I into three parts: $[a_1, a_2, ..., a_{i-1}, m, a_{i+1}, a_{a+2}, ..., a_n]$. Let $I_l = I[1, i), I_r = I[i+1, n]$, where $[l, r)$ includes l, but excludes r. Either can be empty $[\ ]$. In these three parts I_l, m, I_r, I_l is the in-order traverse result of the left sub-tree, I_r is the in-order result of the right sub-tree. Let $k = |I_l|$ be the size of the left sub-tree, we can split $P[2, n]$ at k to two parts: P_l, P_r, where P_l contains the first k elements. We next recursively rebuild the left sub-tree from (P_l, I_l), rebuild the right sub-tree from (P_r, I_r):

   $$\begin{array}{rcl} rebuild\ [\ ]\ [\ ] & = & \varnothing \\ rebuild\ (m{:}ps)\ I & = & (rebuild\ P_l\ I_l, m, rebuild\ P_r\ I_r) \end{array}$$

   Where:

   $$\begin{cases} (I_l, I_r) & = splitWith\ m\ I \\ (P_l, P_r) & = splitAt\ |I_l|\ ps \end{cases}$$

   Below is the example program:

```
rebuild [] _ = Empty
rebuild [c] _ = leaf c
rebuild (x:xs) ins = Node (rebuild prl inl) x (rebuild prr inr) where
  (inl, _:inr) = (takeWhile (≠ x) ins, dropWhile (≠x) ins)
  (prl, prr) = splitAt (length inl) xs
```

We can also update the left and right boundary to implement:

```
Node<T> rebuild([T] pre, [T] ins, Int l = 0, Int r = length(ins)) {
    if l ≥ r then return null
    T c = popFront(pre)
    Int m = find(c, ins)
    var left = rebuild(pre, ins, l, m)
    var right = rebuild(pre, ins, m + 1, r)
    return Node(left, c, right)
}
```

3. For binary search tree, prove that the in-order traverse always visits elements in increase order
4. Consider the performance of tree sort algorithm, what is its complexity for n elements?

Answer of exercise 8.1

1. No, it is not correct. The sub-array $[a_2, a_3, ..., a_n]$ can't map back to binary heap. It's insufficient to only apply HEAPIFY from a_2, we need run BUILD-HEAP to rebuild the heap.
2. For the same reason, it does not work.

Answer of exercise 8.2

1. Realize leftist heap, skew heap, and splay heap in imperative approach.
2. Define fold for heap.

$$
\begin{array}{rcl}
fold\ f\ z\ \varnothing & = & z \\
fold\ f\ z\ H & = & fold\ f\ (f\ (top\ H)\ z)\ (pop\ H)
\end{array}
$$

Answer of exercise 9.1

1. We should use link but not append. Appending is linear to the length of the list, while linking is constant time.
2. Implement the selection sort for both in-placed and not. TO-DO

Answer of exercise 10.1

1. Write a program to generate Pascal's triangle.

```
pascal = gen [1] where
  gen cs (x:y:xs) = gen ((x + y) : cs) (y:xs)
  gen cs _ = 1 : cs
```

2. Prove that the i-th row in tree B_n has $\binom{n}{i}$ nodes.

   Proof. Use induction. There is only a root node for B_0. Assume every row in B_n is binomial number. Tree B_{n+1} is composed from two B_n trees. The 0-th row contains root: $1 = \binom{n+1}{0}$. The i-th row has two parts: one from the $(i-1)$-th row of the left most sub-tree B_n, the other from the i-th row of the other B_n tree. In total:

$$\begin{array}{rcl}
\binom{n}{i-1} + \binom{n}{i} & = & \frac{n!}{(i-1)!(n-i+1)!} + \frac{n!}{i!(n-i)!} \\
& = & \frac{n!}{(i-1)!(n-i)!}(\frac{1}{i} - \frac{1}{n-i+1}) \\
& = & \frac{n!}{(i-1)!(n-i)!} \frac{n+1}{i(n-i+1)} \\
& = & \frac{(n+1)!}{i!(n-i+1)!} \\
& = & \binom{n+1}{i}
\end{array}$$

□

3. Prove there are 2^n elements in B_n tree.

   Proof. From previous exercise, sum all rows of B_n tree:

$$\begin{array}{rll}
& \binom{n}{0} + \binom{n}{1} + ... + \binom{n}{n} & \text{Sum rows} \\
= & (1+1)^n & \text{Let } a = b = 1 \text{ in } (a+b)^n \\
= & 2^n &
\end{array}$$

□

4. Use a container to store sub-trees, how to implement link? How to secure the operation is in constant time? If store all sub-trees in an array, we need linear time to insert a new tree ahead of all sub-trees:

```
function LINK'(T1, T2)
    if KEY(T2) < KEY(T1) then
        Exchange T1 ↔ T2
    PARENT(T2) ← T1
    INSERT(SUB-TREES(T1), 1, T2)
    RANK(T1) ← RANK(T2) + 1
    return T1
```

   We can store the sub-trees in reversed order, it's need constant time to append the new tree on tail.

Answer of exercise 11.2

1. Why need balance check and adjustment after push?
   Consider the case, first *push a* ([ ], [ ]), then *pop*.
2. Prove the amortized performance of paired-list queue is constant time.
3. Implement the paired-array queue.

```
function PUSH(Q, x)
    APPEND(FRONT(Q), x)

function POP(Q)
    if REAR(Q) = [ ] then
        REAR(Q) ← REVERSE(FRONT(Q))
        FRONT(Q) ← [ ]
    n ← LENGTH(REAR(Q))
    x ← REAR(Q)[n]
    LENGTH(REAR(Q)) ← n − 1
    return x
```

Answer of exercise 12.5

1. Eliminate recursion, implement insert with loop.
   Let Mid$(T) = t$ access the middle part of tree $T = (f, t, r)$.

```
function Insert(x, T)
    n = (x)
    ⊥← p ← ([ ], T, [ ])
    while |Front(T)| ≥ 3 do
        f ← Front(T)
        n ← (f[2], f[3], ...)
        Front(T) ← [n, f[1]]
        p ← T
        T ← Mid(T)
    if T = NIL then
        T ← ([n], NIL, [ ])
    else if |Front(T)| = 1 and Rear(T) = [ ] then
        Rear(T) ← Front(T)
        Front(T) ← [n]
    else
        Insert(Front(T), n)
    Mid(p) ← T
    T ← Mid(⊥), Mid(⊥) ← NIL
    return T
```

   We wrap x in a leaf (x). If there are more than 3 elements in f, we go top-down along the middle part. We extract the elements except the first one in f out, wrap them in a node n (depth + 1), then insert n to the middle. We form n and the remaining in f as the new f finger. At the end of traverse, we either reach an empty tree, or a tree can hold more elements in f. For empty tree case, we create a new leaf node; otherwise, we insert n to the head of f. We return the root T. To simplify implementation, we create a special $\perp$ node as the parent of the root.

Answer of exercise 12.6

1. Eliminate recursion, implement extract in loops.
   We borrow node from the middle when f is empty. However, the tree may not well formed, e.g., both f and the middle are empty. It is caused by splitting.

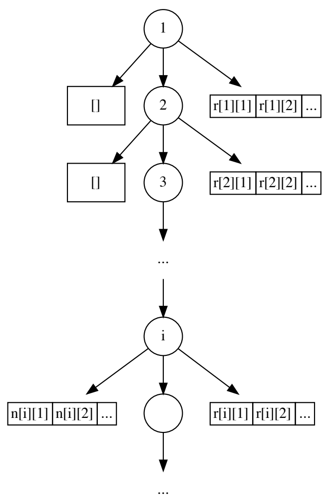
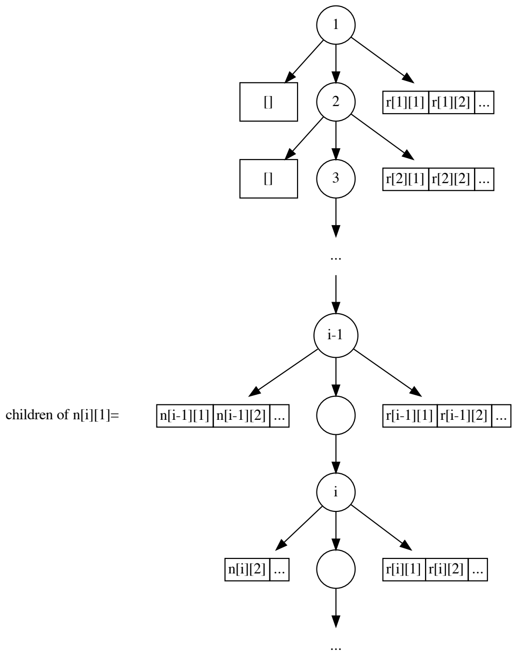
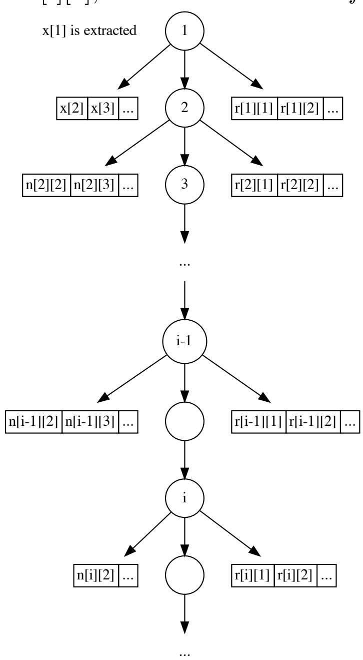


Figure 12.10: The f isn't empty at level i.

To extract the first element, we need a top-bottom pass, locate a sub-tree, either f isn't empty, or both f and the middle are empty as shown in figure 12.10. For the former, we extract the first node from f; for the latter, we swap f and r, convert it to the former case. If the node extracted from f isn't a leaf, we need go on extracting. We back track along the parent, till extract a leaf and reach to the root, as shown in figure 12.11.



Extract the first $n[i][1]$, move its sub-tree to f in upper level.



Repeat i times, extract $x[1]$.

Figure 12.11: Bottom up back track to extract a leaf.

Assume the tree isn't empty, we implement extract as below:

```
function Extract(T)
    ⊥ ← ([], T, [])
    while Front(T) = [ ] and Mid(T) ≠ NIL do
        T ← Mid(T)
    if Front(T) = [ ] and Rear(T) ≠ [ ] then
        Exchange Front(T) ↔ Rear(T)
    f ← Front(T), r ← Rear(T)
    n ← (f[1], f[2], ...)                          ▷ n is 2-3 tree
    repeat
        Front(T) ← [n_2, n_3, ..]
        n ← n_1
        T ← Parent(T)
        if Mid(T) becomes empty then
            Mid(T) ← NIL
    until n is leaf
    return (Elem(n), Mid(⊥))
```

Where function Elem(n) access the element in sub-tree n. We need change the way to access the first/last element of finger tree. If the finger is empty, and the middle isn't empty, we need search along the middle.

```
function First-Leaf(T)
    while Front(T) = [ ] and Mid(T) ≠ NIL do
        T ← Mid(T)
    if Front(T) = [ ] and Rear(T) ≠ [ ] then
        n ← Rear(T)[1]
    else
        n ← Front(T)[1]
    while n is NOT leaf do
        n ← n_1
    return n

function First(T)
    return Elem(First-Leaf(T))
```

In the second loop, if the node is not a leaf, we need traverse along the first sub-tree. The method to access the last element is symmetric.

Answer of exercise 12.7

1. For random access, how to handle empty tree ∅ and out of bound cases?
   We check boundaries during random access, for example:

$$
\begin{array}{rcl}
\varnothing[i] & = & \mathit{Nothing} \\
T[i] & = & \begin{cases} i < 0 \text{ or } i \geq \mathit{size}\ T : & \mathit{Nothing} \\ \text{otherwise} : & ... \end{cases}
\end{array}
$$

2. Implement *cut* i S, split sequence S at position i.
   We give an implementation based on the tree definition in appendix. We first do boundary check, if $0 \leq i < \mathit{size}\ s$, we next call *catTree* i S to split the tree:

```
cut :: Int → Seq a → (Seq a, Maybe a, Seq a)
cut i (Seq xs) | i < 0 = (Seq Empty, Nothing, Seq xs)
               | i < size xs = case cutTree i xs of
                 (a, Just (Place _ (Elem x)), b) → (Seq a, Just x, Seq b)
                 (a, Nothing, b) → (Seq a, Nothing, Seq b)
               | otherwise = (Seq xs, Nothing, Seq Empty)
```

cutTree splits the tree in three parts: left, middle, and right. We wrap the middle in *Maybe* type to handle the not found case; when found, the result is a pair of position i' and node a, wrapped in the type of *Place*. if i points to finger f or r, we call *cutList* to further split, then build the result; if i points to middle, we recursively cut the middle to obtain a place *Place* i' a, then cut the 2-3 tree a at position i':

```
cutTree :: (Sized a) ⇒ Int → Tree a → (Tree a, Maybe (Place a), Tree a)
cutTree _ Empty = (Empty, Nothing, Empty)
cutTree i (Lf a) | i < size a = (Empty, Just (Place i a), Empty)
                 | otherwise = (Lf a, Nothing, Empty)
cutTree i (Br s f m r)
  | i < sf = case cutList i f of
               (xs, x, ys) → (Empty <<< xs, x, tree ys m r)
  | i < sm = case cutTree (i - sf) m of
               (t1, Just (Place i' a), t2) → let (xs, x, ys) = cutNode i' a
                 in (tree f t1 xs, x, tree ys t2 r)
  | i < s  = case cutList (i - sm) r of
               (xs, x, ys) → (tree f m xs, x, ys >>> Empty)
  where
    sf = sum $ map size f
    sm = sf + size m
```

Where *tree* f m r builds a finger tree, and simplify the result:

```
tree as Empty [] = as >>> Empty
tree [] Empty bs = Empty <<< bs
tree [] m r = Br (size m + sum (map size r)) (nodesOf f) m' r
    where (f, m') = uncons m
tree f m [] = Br (size m + sum (map size f)) f m' (nodesOf r)
    where (m', r) = unsnoc m
tree f m r = Br (size m + sum (map size f) + sum (map size r)) f m r
```

We implement the finger cut and 2-3 tree cut as below:

```
cutList :: (Sized a) ⇒ Int → [a] → ([a], Maybe (Place a), [a])
cutList _ [] = ([], Nothing, [])
cutList i (x:xs) | i < sx = ([], Just (Place i x), xs)
                 | otherwise = let (xs', y, ys) = cutList (i - sx) xs
                                in (x:xs', y, ys)
  where sx = size x

cutNode :: (Sized a) ⇒ Int → Node a → ([a], Maybe (Place a), [a])
cutNode i (Tr2 _ a b) | i < sa = ([], Just (Place i a), [b])
                      | otherwise = ([a], Just (Place (i - sa) b), [])
  where sa = size a
cutNode i (Tr3 _ a b c) | i < sa = ([], Just (Place i a), [b, c])
                        | i < sab = ([a], Just (Place (i - sa) b), [c])
                        | otherwise = ([a, b], Just (Place (i - sab) c), [])
  where sa = size a
        sab = sa + size b
```

With *cut* defined, we can update or delete any element at given position, move to front (MTF), they all bound to $O(\lg n)$ time.

```
setAt s i x = case cut i s of
  (_, Nothing, _) → s
```

```
  (xs, Just y, ys) → xs ++ (x <| ys)

extractAt s i = case cut i s of (xs, Just y, ys) → (y, xs ++ ys)

moveToFront i s = if i < 0 || i ≥ size s then s
                  else let (a, s') = extractAt s i in a <| s'
```

Answer of exercise 14.1

1. Prove the performance of k-selection problem is $O(n)$ in average (refer to the quick sort performance analysis).
2. To find the top k element in A, we can search $x = \max\ (take\ k\ A), y = \min\ (drop\ k\ A)$. If $x < y$, then the first k elements in A is the answer; otherwise, we partition the first k elements with x, partition the rest with y, then recursively find in subsequence $[a|a \leftarrow A, x < a < y]$ for the top k' elements, where $k' = k - |[a|a \leftarrow A, a \leq x]|$. Implement this solution, and evaluate its performance.

   procedure TOPS(k, A)
   $\quad l \leftarrow 1$
   $\quad u \leftarrow |A|$
   $\quad$ **loop**
   $\quad\quad i \leftarrow$ MAX-AT$(A[l..k])$
   $\quad\quad j \leftarrow$ MIN-AT$(A[k+1..u])$
   $\quad\quad$ **if** $A[i] < A[j]$ **then**
   $\quad\quad\quad$ break
   $\quad\quad$ EXCHANGE $A[l] \leftrightarrow A[j]$
   $\quad\quad$ EXCHANGE $A[k+1] \leftrightarrow A[i]$
   $\quad\quad l \leftarrow$ PARTITION(A, l, k)
   $\quad\quad u \leftarrow$ PARTITION$(A, k+1, u)$

   The performance is $O(n)$ in average. Every loop, it takes linear time to locate the min i, max j. Then partition two rounds in linear time. If the partition is balanced, we discard half elements in average, hence the total time is bound to: $O(n + n/2 + n/4...) = O(n)$.
3. Find the median of two sorted arrays A and B in $O(\lg(m+n))$ time, where $m = |A|, n = |B|$. The median x is defined as $||\{a \leq x : a \in A\}| + |\{b \leq x : b \in B\}| - |\{a > x : a \in A\}| - |\{b > x : b \in B\}|| \leq 1$.
4. For the saddle back search, eliminate recursion, implement it in loops to update the boundary.

   function SOLVE(f, z)
   $\quad p \leftarrow 0, q \leftarrow z$
   $\quad S \leftarrow \phi$
   $\quad$ **while** $p \leq z$ 且 $q \geq 0$ **do**
   $\quad\quad z' \leftarrow f(p, q)$
   $\quad\quad$ **if** $z' < z$ **then**
   $\quad\quad\quad p \leftarrow p + 1$
   $\quad\quad$ **else if** $z' > z$ **then**
   $\quad\quad\quad q \leftarrow q - 1$
   $\quad\quad$ **else**
   $\quad\quad\quad S \leftarrow S \cup \{(p, q)\}$
   $\quad\quad\quad p \leftarrow p + 1, q \leftarrow q - 1$
   $\quad$ **return** S

5. For 2D search, let the bottom-left be the minimum, the top-right be the maximum. if z is less than the minimum or greater than the maximum, then no solution; otherwise cut the rectangle into 4 parts with a horizontal line and a vertical line crossed at the center. then recursive search in these 4 small rectangles. Implement this solution and evaluate its performance.

   procedure SEARCH(f, z, a, b, c, d) ▷ (a, b): bottom-left (c, d): top-right
       if $z \le f(a, b)$ or $f(c, d) \ge z$ **then**
           if $z = f(a, b)$ **then**
               record (a, b) as a solution
           if $z = f(c, d)$ **then**
               record (c, d) as a solution
           return
       $p \leftarrow \lfloor \frac{a+c}{2} \rfloor$
       $q \leftarrow \lfloor \frac{b+d}{2} \rfloor$
       SEARCH(f, z, a, q, p, d)
       SEARCH(f, z, p, q, c, d)
       SEARCH(f, z, a, b, p, q)
       SEARCH(f, z, p, b, c, q)

   Performance:

Answer of exercise 14.2

1. Extend to find k majorities that occurs over $\lfloor n/k \rfloor$ in collection A, where $n = |A|$.

   We use a dictionary of $Map : T \mapsto Int$, where T is the element type if A. It records the net-wins for candidate a. Start the dictionary from empty $\varnothing$. We scan A while update the dictionary: $foldr\ maj\ \varnothing\ A$, where maj is defined as:

$$maj\ a\ m = \begin{cases} a \in m: & m[a] \leftarrow m[a] + 1 \\ |m| < k: & m[a] \leftarrow 1 \\ \text{otherwise}: & filter\ (b \mapsto m[b] \neq 0)\ \{b \mapsto m[b] - 1 | b \in m\} \end{cases} \tag{14.21}$$

   For every a in A, if $a \notin m$ (new to the dictionary), and the candidates in m is less than k, we add a to m with one net-win vote: $m[a] \leftarrow 1$; if $a \in m$, add the vote by 1: $m[a] \leftarrow m[a] + 1$; otherwise, if there are already k candidates, we reduce the vote by 1 for every one, and remove the candidate when the vote becomes 0.
   We need verify the remaining candidates at last, whether the votes $> n/k$, let $m' = \{(a, 0) | a \in m\}$. Scan A again: $foldr\ cnt\ m'\ A$, where cnt is defined as:

$$cnt\ a\ m' = \text{if } a \in m' \text{ then } m'[a] \leftarrow m'[a] + 1 \text{ else } m' \tag{14.22}$$

   After scan, m' records the votes for each candidate, we filter the true winners in: $keys\ (filter\ (> n/k)\ m')$.

```
majorities k xs = verify $ foldr maj Map.empty xs where
  maj :: (Eq a, Ord a) ⇒ a → Map.Map a Int → Map.Map a Int
  maj x m | x `Map.member` m = Map.adjust (1+) x m
          | Map.size m < k = Map.insert x 1 m
          | otherwise = Map.filter (≠0) $ Map.map (-1+) m
  verify m = Map.keys $ Map.filter (> th) $ foldr cnt m' xs where
    m' = Map.map (const 0) m
    cnt :: (Eq a, Ord a) ⇒ a → Map.Map a Int → Map.Map a Int
    cnt x m = if x `Map.member` m then Map.adjust (1+) x m else m
    th = (length xs) `div` k
```

Below is the corresponding iterative implementation:

```
function Maj(k, A)
    m ← {}
    for each a in A do
        if a ∈ m then
            m[a] ← m[a] + 1
        else if |m| < k then
            m[a] ← 1
        else
            for each c in m do
                m[c] ← m[c] − 1
                if m[c] = 0 then
                    Remove(c, m)
    for each c in m do
        m[c] ← 0
    for each a in A do                                  ▷ verify
        if a ∈ m then
            m[a] ← m[a] + 1
    r = [ ], n ← |A|
    for each c in m do
        if m[c] > n/k then
            Add(c, r)
    return r
```

Answer of exercise 14.3

1. Modify the solution that finds the max sum of sub-vector, returns the sub-vector of the maximum sum.
   If want to return the sub-list together with the maximum sum, we can maintain two pairs P_m and P during folding, each pair contains the sum and the sub-list (S, L).

$$
\begin{array}{lcl}
max_s & = & 1st \circ foldr\ f\ ((0, [\,]), (0, [\,])) \\
\text{where}: & & f\ x\ (P_m, (S, L)) = (P'_m = \max(P_m, P'), P' = \max((0, [\,]), (x + S, x{:}L)))
\end{array}
$$

2. Bentley gives a divide and conquer algorithm to find the max sum in $O(n \lg n)$ time[2]. Split the vector at middle, recursively find the max sum in two halves, and the max sum that crosses the middle. Then pick the greatest. Implement this solution.

```
function Max-Sum(A)
    if A = φ then
        return 0
    else if |A| = 1 then
        return Max(0, A[1])
    else
        m ← ⌊|A|/2⌋
        a ← Max-From(Reverse(A[1...m]))
        b ← Max-From(A[m + 1...|A|])
        c ← Max-Sum(A[1...m])
        d ← Max-Sum(A[m + 1...|A|)
        return Max(a + b, c, d)
```

function MAX-FROM(A)
    $sum \leftarrow 0, m \leftarrow 0$
    for $i \leftarrow 1$ to $|A|$ **do**
        $sum \leftarrow sum + A[i]$
        $m \leftarrow$ MAX(m, sum)
    return m

Consider the recursive equation: $T(n) = 2T(n/2) + O(n)$, from the master theorem, the performance is $O(n)$.

Answer of exercise 14.10

1. Use heap to build the Huffman tree: take two trees from the top, merge then add back to the heap.

$$Huffman\ H = \begin{cases} H = \varnothing : & \varnothing \\ |H| = 1 : & pop\ H \\ \text{otherwise} : & Huffman\ (push\ (merge\ t_a\ t_b)\ H'') \end{cases}$$

Where: $(t_a, H') = pop\ H, (t_b, H'') = pop\ H'$

function HUFFMAN(H)
    while $|H| > 1$ **do**
        $t_a \leftarrow$ POP(H)
        $t_b \leftarrow$ POP(H)
        PUSH(H, MERGE(t_a, t_b))
    return POP(H)

2. If we sort the symbols by their weight as A, there is a linear time algorithm to build the Huffman tree: use a tree Q to store the merge result, repeat take the minimal weighted tree from Q and the head of A, merge then add to the queue. After process all trees in A, there is a single tree in the Q, which is the Huffman tree. Implement this algorithm.
$Huffman\ (t{:}ts) = build\ (t, (ts, \varnothing))$, where:

$$\begin{array}{rcl} build\ (t, ([\ ], \varnothing)) & = & t \\ build\ (t, h) & = & build\ (extract\ (ts, push\ (merge\ t\ t')\ q)) \end{array}$$

其中：$(t', (ts, q)) = extract\ h$

$$\begin{array}{rcl} extract\ (t{:}ts, \varnothing)) & = & (t, (ts, \varnothing)) \\ extract\ ([\ ], q) & = & (t, ([\ ], q')), \text{where} : (t, q') = pop\ q \\ extract\ (t{:}ts, q) & = & \begin{cases} t' < t : & (t', (t{:}ts, q')), \text{where} : (t', q') = pop\ q \\ t < t' : & (t, (ts, q)) \end{cases} \end{array}$$

3. Given a Huffman tree T, implement the decode algorithm with fold left.
$decode = snd \circ (foldl\ lookup\ (T, [\ ]))$, where:

$$\begin{array}{rcl} lookup\ ((w, c), cs)\ b & = & (T, c{:}cs) \\ lookup\ ((w, l, r), cs)\ b & = & \text{if } b = 0 \text{ then } (l, cs) \text{ else } (r, cs) \end{array}$$

Answer of exercise 14.11

1. For the longest common sub-sequence, build the optimal solution table with fold.

```
import Data.Sequence (Seq, singleton, fromList, index, (|>))

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)
```

Bibliography

[1] Richard Bird. “Pearls of functional algorithm design”. Cambridge University Press; 1 edition (November 1, 2010). ISBN-10: 0521513383. pp1 - pp6.

[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.

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

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

[7] Wikipedia. “Fold(high-order function)”. https://en.wikipedia.org/wiki/Fold_(higher-order_function)

[8] Wikipedia. “Function Composition”. https://en.wikipedia.org/wiki/Function_composition

[9] Wikipedia. “Partial application”. https://en.wikipedia.org/wiki/Partial_application

[10] 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

[11] Wikipedia. “Bubble sort”. https://en.wikipedia.org/wiki/Bubble_sort

[12] 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

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

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

[15] Lyn Turbak. “Red-Black Trees”. http://cs.wellesley.edu/~cs231/fall01/red-black.pdf Nov. 2, 2001.

[16] Rosetta Code. “Pattern matching”. http://rosettacode.org/wiki/Pattern_matching

[17] Hackage. “Data.Tree.AVL”. http://hackage.haskell.org/packages/archive/AvlTree/4.2/doc/html/Data-Tree-AVL.html

[18] Wikipedia. "AVL tree". https://en.wikipedia.org/wiki/AVL_tree

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

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

[21] Chris Okasaki and Andrew Gill. "Fast Mergeable Integer Maps". Workshop on ML, September 1998, pages 77-86.

[22] D.R. Morrison, "PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric", Journal of the ACM, 15(4), October 1968, pages 514-534.

[23] Wikipedia. "Suffix Tree". https://en.wikipedia.org/wiki/Suffix_tree

[24] Wikipedia. "Trie". https://en.wikipedia.org/wiki/Trie

[25] Wikipedia. "T9 (predictive text)". https://en.wikipedia.org/wiki/T9_(predictive_text)

[26] Wikipedia. "Predictive text". https://en.wikipedia.org/wiki/Predictive_text

[27] 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

[28] Weiner, P. "Linear pattern matching algorithms", 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1-11, doi:10.1109/SWAT.1973.13

[29] Esko Ukkonen. "Suffix tree and suffix array techniques for pattern analysis in strings". http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

[30] Suffix Tree (Java). http://en.literateprograms.org/Suffix_tree_(Java)

[31] 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

[32] Robert Giegerich and Stefan Kurtz. "A Comparison of Imperative and Purely Functional Suffix Tree Constructions". Algorithmica 19 (3): 331–353. doi:10.1007/PL00009177. http://www.zbh.uni-hamburg.de/pubs/pdf/GieKur1997.pdf

[33] Bryan O'Sullivan. "suffixtree: Efficient, lazy suffix tree implementation". http://hackage.haskell.org/package/suffixtree

[34] Danny. http://hkn.eecs.berkeley.edu/~dyoo/plt/suffixtree/

[35] Dan Gusfield. "Algorithms on Strings, Trees and Sequences Computer Science and Computational Biology". Cambridge University Press; 1 edition (May 28, 1997) ISBN: 9780521585194

[36] Lloyd Allison. "Suffix Trees". http://www.allisons.org/ll/AlgDS/Tree/Suffix/

[37] Esko Ukkonen. "Suffix tree and suffix array techniques for pattern analysis in strings". http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

[38] 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

[39] Wikipeida. "B-tree". https://en.wikipedia.org/wiki/B-tree

[40] Wikipedia. "Heap (data structure)". https://en.wikipedia.org/wiki/Heap_(data_structure)

[41] Wikipedia. "Heapsort". https://en.wikipedia.org/wiki/Heapsort

[42] Rosetta Code. "Sorting algorithms/Heapsort". http://rosettacode.org/wiki/Sorting_algorithms/Heapsort

[43] Wikipedia. "Leftist Tree". https://en.wikipedia.org/wiki/Leftist_tree

[44] Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in Java. http://www.brpreiss.com/books/opus5/index.html

[45] 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

[46] Wikipedia. "Skew heap". https://en.wikipedia.org/wiki/Skew_heap

[47] Sleator, Daniel Dominic; Jarjan, Robert Endre. "Self-adjusting heaps" SIAM Journal on Computing 15(1):52-69. doi:10.1137/0215004 ISSN 00975397 (1986)

[48] Wikipedia. "Splay tree". https://en.wikipedia.org/wiki/Splay_tree

[49] 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

[50] NIST, "binary heap". http://xw2k.nist.gov/dads//HTML/binaryheap.html

[51] 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

[52] Wikipedia. "Strict weak order". https://en.wikipedia.org/wiki/Strict_weak_order

[53] Wikipedia. "FIFA world cup". https://en.wikipedia.org/wiki/FIFA_World_Cup

[54] Wikipedia. "K-ary tree". https://en.wikipedia.org/wiki/K-ary_tree

[55] Wikipedia, "Pascal's triangle". https://en.wikipedia.org/wiki/Pascal's_triangle

[56] 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

[57] Chris Okasaki. "Fibonacci Heaps." http://darcs.haskell.org/nofib/gc/fibheaps/orig

[58] 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.

[59] 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

[60] Herb Sutter. "Writing a Generalized Concurrent Queue". Dr. Dobb's Oct 29, 2008. http://drdobbs.com/cpp/211601363?pgno=1

[61] Wikipedia. "Tail-call". https://en.wikipedia.org/wiki/Tail_call

[62] Wikipedia. "Recursion (computer science)". https://en.wikipedia.org/wiki/Recursion_(computer_science)#Tail-recursive_functions

[63] Harold Abelson, Gerald Jay Sussman, Julie Sussman. "Structure and Interpretation of Computer Programs, 2nd Edition". MIT Press, 1996, ISBN 0-262-51087-1.

[64] Chris Okasaki. "Purely Functional Random-Access Lists". Functional Programming Languages and Computer Architecture, June 1995, pages 86-95.

[65] Ralf Hinze and Ross Paterson. "Finger Trees: A Simple General-purpose Data Structure," in Journal of Functional Programming 16:2 (2006), pages 197-217. http://www.soi.city.ac.uk/~ross/papers/FingerTree.html

[66] 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. 49-60.

[67] Generic finger-tree structure. http://hackage.haskell.org/packages/archive/fingertree/0.0/doc/html/Data-FingerTree.html

[68] Wikipedia. "Move-to-front transform". https://en.wikipedia.org/wiki/Move-to-front_transform

[69] Robert Sedgewick. "Implementing quick sort programs". Communication of ACM. Volume 21, Number 10. 1978. pp.847 - 857.

[70] Jon Bentley, Douglas McIlroy. "Engineering a sort function". Software Practice and experience VOL. 23(11), 1249-1265 1993.

[71] Robert Sedgewick, Jon Bentley. "Quicksort is optimal". http://www.cs.princeton.edu/~rs/talks/QuicksortIsOptimal.pdf

[72] Fethi Rabhi, Guy Lapalme. "Algorithms: a functional programming approach". Second edition. Addison-Wesley, 1999. ISBN: 0201-59604-0

[73] Simon Peyton Jones. "The Implementation of functional programming languages". Prentice-Hall International, 1987. ISBN: 0-13-453333-X

[74] Jyrki Katajainen, Tomi Pasanen, Jukka Teuhola. "Practical in-place mergesort". Nordic Journal of Computing, 1996.

[75] Josè Bacelar Almeida and Jorge Sousa Pinto. "Deriving Sorting Algorithms". Technical report, Data structures and Algorithms. 2008.

[76] Cole, Richard (August 1988). "Parallel merge sort". SIAM J. Comput. 17 (4): 770-785. doi:10.1137/0217049. (August 1988)

[77] Powers, David M. W. "Parallelized Quicksort and Radixsort with Optimal Speedup", Proceedings of International Conference on Parallel Computing Technologies. Novosibirsk. 1991.

[78] Wikipedia. "Quicksort". https://en.wikipedia.org/wiki/Quicksort

[79] Wikipedia. "Total order". http://en.wokipedia.org/wiki/Total_order

[80] Wikipedia. “Harmonic series (mathematics)”. https://en.wikipedia.org/wiki/Harmonic_series_(mathematics)

[81] M. Blum, R.W. Floyd, V. Pratt, R. Rivest and R. Tarjan, ”Time bounds for selection,” J. Comput. System Sci. 7 (1973) 448-461.

[82] Edsger W. Dijkstra. “The saddleback search”. EWD-934. 1985. http://www.cs.utexas.edu/users/EWD/index09xx.html.

[83] 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.

[84] Cormode, Graham; S. Muthukrishnan (2004). “An Improved Data Stream Summary: The Count-Min Sketch and its Applications”. J. Algorithms 55: 29-38.

[85] Knuth Donald, Morris James H., jr, Pratt Vaughan. “Fast pattern matching in strings”. SIAM Journal on Computing 6 (2): 323-350. 1977.

[86] Robert Boyer, Strother Moore. “A Fast String Searching Algorithm”. Comm. ACM (New York, NY, USA: Association for Computing Machinery) 20 (10): 762-772. 1977

[87] R. N. Horspool. “Practical fast searching in strings”. Software - Practice & Experience 10 (6): 501-506. 1980.

[88] Wikipedia. “Boyer-Moore string search algorithm”. https://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm

[89] Wikipedia. “Eight queens puzzle”. https://en.wikipedia.org/wiki/Eight_queens_puzzle

[90] George Pólya. “How to solve it: A new aspect of mathematical method”. Princeton University Press(April 25, 2004). ISBN-13: 978-0691119663

[91] Wikipedia. “David A. Huffman”. https://en.wikipedia.org/wiki/David_A._Huffman

[92] Andrei Alexandrescu. “Modern C++ design: Generic Programming and Design Patterns Applied”. Addison Wesley February 01, 2001, ISBN 0-201-70431-5

[93] Benjamin C. Pierce. “Types and Programming Languages”. The MIT Press, 2002. ISBN:0262162091

[94] Joe Armstrong. “Programming Erlang: Software for a Concurrent World”. Pragmatic Bookshelf; 1 edition (July 18, 2007). ISBN-13: 978-1934356005

[95] SGI. “transform”. http://www.sgi.com/tech/stl/transform.html

[96] ACM/ICPC. “The drunk jailer.” Peking University judge online for ACM/ICPC. http://poj.org/problem?id=1218.

[97] Haskell wiki. “Haskell programming tips”. 4.4 Choose the appropriate fold. http://www.haskell.org/haskellwiki/Haskell_programming_tips

[98] Wikipedia. “Dot product”. https://en.wikipedia.org/wiki/Dot_product

[99] Xinyu LIU. “Isomorphism - mathematics of programming”. https://github.com/liuxinyu95/unplugged

GNU Free Documentation License

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.

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.

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, 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.

"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, 268

Auto completion, 105
AVL tree, 79
  balance, 82
  definition, 79
  imperative insert, 84
  insert, 81
  verification, 83

B-tree, 115
  delete, 126
  insert, 117
  look up, 124
BFS, 284
Binary heap, 137
  build, 139
  decrease key, 142
  Heapify, 138
  insertion, 142
  pop, 140
  push, 142
  top, 140
  top-k, 140
binary heap by array, 137
Binary Random Access List
  insert, 204
  random access, 206
  remove, 204
Binary search, 252
binary search tree, 43
  data layout, 45
  delete, 51
  insertion, 45
  looking up, 49
  min/max, 50
  random build, 54
  search, 49
  succ/pred, 50
  traverse, 47
binary tree, 43
Binomial Heap
  Link, 172
Binomial heap, 169
  definition, 170
  insert, 173
  pop, 175
  push, 173
Binomial tree, 170
  merge, 174
Boyer-Moor majority number, 261
Breadth-first search, 284

Change making problem, 289
Cock-tail sort, 160
complete binary tree, 137
Curried Form, 25
Currying, 25

Deep-first search, 266
DFS, 266
Dynamic programming, 290

equivalent, 34

Fibonacci Heap
  decrease key, 182
  delete min, 178
  insert, 177
  merge, 177
  pop, 178
Fibonacci heap, 176
Finger tree
  Append to right, 215
  Concatenate, 215
  insert to left, 213
  Random access, 216
  Remove from left, 213
  Remove from right, 215
fold, 35

Greedy algorithm, 285

Heap sort, 142
Huffman coding, 285

in-order traverse, 47

Insertion sort
- binary search, 59
- binary search tree, 60
- linked-list setting, 59

insertion sort, 57
- insertion, 58

Integer Patricia, 92
Integer prefix tree, 92
Integer tree
- insert, 93
- lookup, 97

Integer trie, 89
- insert, 90
- look up, 92

Kloski puzzle, 281
KMP, 264
Knuth-Morris-Pratt algorithm, 264

LCS, 292
left child, right sibling, 172
Leftist heap, 144
- heap sort, 146
- insert, 146
- merge, 145
- pop, 145
- rank, 144
- S-value, 144
- top, 145

List
- append, 21
- break, 33
- concat, 24
- concats, 37
- cons, 18
- Construction, 18
- definition, 17
- delete, 23
- delete at, 23
- drop, 32
- drop while, 32
- elem, 38
- empty, 17
- empty testing, 17
- existence testing, 38
- Extract sub-list, 32
- filter, 38
- find, 38
- fold from left, 36
- fold from right, 35
- foldl, 36
- foldr, 35
- for each, 29
- get at, 18
- group, 33
- head, 18
- index, 18
- infix, 39
- init, 19
- insert, 21
- insert at, 21
- last, 19
- length, 18
- list comprehension, 29
- lookup, 38
- map, 28
- matching, 39
- maximum, 26
- minimum, 26
- mutate, 21
- prefix, 39
- product, 24
- reverse, 31
- Right index, 19
- rindex, 19
- set at, 21
- span, 33
- split at, 32
- suffix, 39
- sum, 24
- tail, 18
- take, 32
- take while, 32
- Transform, 27
- unzip, 40
- ZF expression, 29
- zip, 40

Longest common sub-sequence, 292

Maximum sum problem, 262
Maze problem, 266
Merge Sort, 236
- Bottom-up merge sort, 246
- In-place merge sort, 240
- Merge, 237
- Nature merge sort, 243
- Performance, 238
- Work area, 239, 240

minimum free number, 9
MTF, 217

Paired-array sequence
- random access, 209
- remove and balance, 209

Pairing heap, 184
- decrease key, 185
- definition, 185
- delete, 187
- insert, 185
- pop, 186
- top, 185

Parallel merge sort, 247
Parallel quick sort, 247
Patricia, 101
Peg puzzle, 270
post-order traverse, 47
pre-order traverse, 47
Prefix tree, 101
- insert, 101
- look up, 104

Queue
- Balance Queue, 197
- Circular buffer, 194
- Lazy real-time queue, 200
- linked-list, 193
- Paired-array queue, 196
- Paired-list queue, 196
- Real-time queue, 197

Quick Sort
- 2-way partition, 230
- 3-way partition, 231
- Average case, 227

Quick sort, 223
- Improvement, 229
- partition, 224
- Performance, 227
- Ternary partition, 229

Radix tree, 89
range traverse, 51
Red-black tree
- Imperative delete, 307

red-black tree, 63, 66
- delete, 70
- imperative insertion, 74
- insert, 68
- red-black properties, 66

reduce, 36

Saddle back search, 254
Selection algorithm, 251
selection sort, 157
- min, 158
- tail-recursive min, 158

Sequence
- Binary random access list, 203
- Concatenate-able list, 210
- finger tree, 211
- numeric representation, 206
- Paired-array sequence, 209

Skew heap, 146
- insertion, 147
- merge, 147
- pop, 147
- top, 147

Splay heap, 147
- insert, 151
- merge, 152
- pop, 151
- splay, 148
- top, 151

strict weak order, 159
Subset sum, 294

T9, 107
Tail call, 24
Tail recursion, 24
Tail recursive call, 24
The wolf, goat, and cabbage puzzle, 274
Tournament knock out, 162
tree reconstruction, 48
tree rotation, 64
Trie, 98
- insert, 98
- look up, 100

Water jugs puzzle, 277
word counter, 43