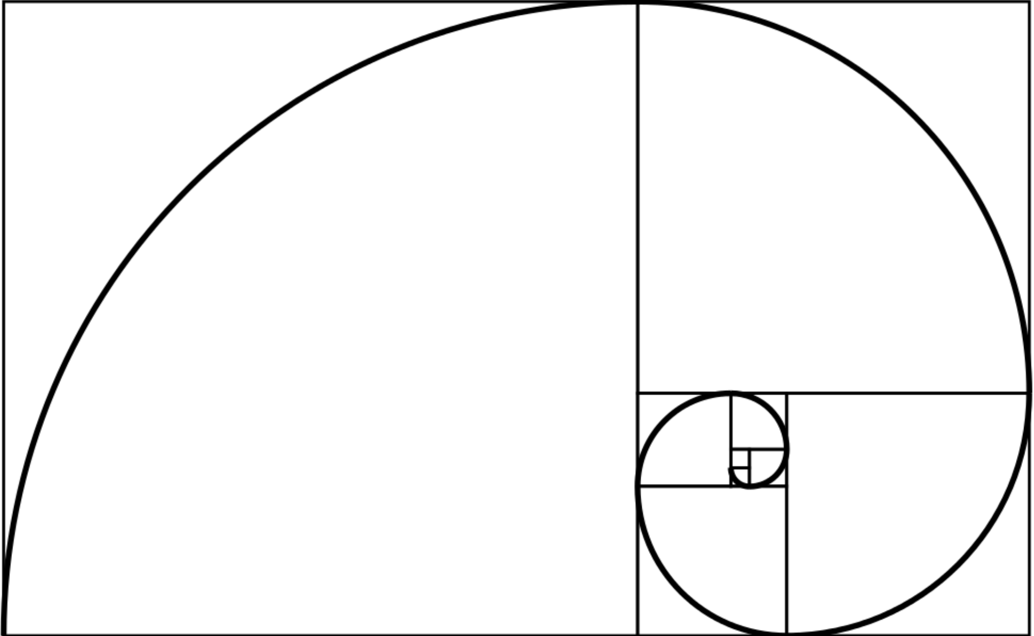
# Elementary Algorithms



Xinyu LIU [1]

January 15, 2023

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

# Contents


0.1 The smallest free number . . . 10
0.1.1 Improvement . . . 11
0.1.2 Divide and Conquer . . . 12
0.1.3 Expressiveness and performance . . . 13
0.2 Regular number . . . 13
0.2.1 The brute-force solution . . . 13
0.2.2 Improvement . . . 14
0.2.3 Queues . . . 16
0.3 Summary . . . 17

**1 List** **19**
1.1 Introduction . . . 19
1.2 Definition . . . 19
1.2.1 Access . . . 20
1.3 Basic operations . . . 20
1.3.1 index . . . 20
1.3.2 Last . . . 21
1.3.3 Reverse index . . . 22
1.3.4 Mutate . . . 23
Append . . . 23
Set value . . . 24
insert . . . 25
delete . . . 26
concatenate . . . 27
1.3.5 sum and product . . . 28
Recursive sum and product . . . 28
Tail call recursion . . . 29
1.3.6 maximum and minimum . . . 31
1.4 Transform . . . 32
1.4.1 map and for-each . . . 33
Map . . . 34
For each . . . 35
Examples . . . 35
1.4.2 reverse . . . 37
1.5 Sub-list . . . 38
1.5.1 take, drop, and split-at . . . 38
conditional take and drop . . . 39
1.5.2 break and group . . . 39
break and span . . . 39
group . . . 40
1.6 Fold . . . 42

1.6.1 fold right . . . 42
1.6.2 fold left . . . 44
1.6.3 example . . . 45
concatenate . . . 46
1.7 Search and filter . . . 46
1.7.1 Exist . . . 46
1.7.2 Look up . . . 47
1.7.3 find and filter . . . 47
1.7.4 Match . . . 48
1.8 zip and unzip . . . 49
1.9 Further reading . . . 52

**2 Binary Search Tree** **53**
2.1 Introduction . . . 53
2.2 Data Layout . . . 55
2.3 Insertion . . . 55
2.4 Traverse . . . 57
2.5 Query . . . 59
2.5.1 Look up . . . 59
2.5.2 Minimum and maximum . . . 60
2.5.3 Successor and predecessor . . . 60
2.6 Deletion . . . 61
2.7 Random build . . . 64
2.8 Map . . . 65
2.9 Appendix: Example programs . . . 65

**3 Insertion sort** **67**
3.1 Introduction . . . 67
3.2 Insertion . . . 68
3.3 Binary search . . . 69
3.4 List . . . 69
3.5 Binary search tree . . . 70
3.6 Summary . . . 71

**4 Red-black tree** **73**
4.1 Introduction . . . 73
4.1.1 Balance . . . 74
4.1.2 Tree rotation . . . 74
4.2 Definition . . . 76
4.3 Insert . . . 78
4.4 Delete . . . 80
4.5 Imperative red-black tree algorithm ⋆ . . . 84
4.6 Summary . . . 85
4.7 Appendix: Example programs . . . 86

**5 AVL tree** **89**
5.1 Introduction . . . 89
5.2 Definition . . . 89
5.3 Insert . . . 91
5.3.1 Balance . . . 92
Verification . . . 93
5.4 Imperative AVL tree algorithm ★ . . . 94

5.5 Summary . . . 96
5.6 Appendix: Example programs . . . 96

6 Radix tree 99
6.1 Integer trie . . . 99
6.1.1 Definition . . . 100
6.1.2 Insert . . . 100
6.1.3 Look up . . . 102
6.2 Integer prefix tree . . . 102
6.2.1 Definition . . . 103
6.2.2 Insert . . . 103
6.2.3 Lookup . . . 107
6.3 Trie . . . 108
6.3.1 Definition . . . 108
6.3.2 Insert . . . 108
6.3.3 Look up . . . 110
6.4 Prefix tree . . . 111
6.4.1 Definition . . . 111
6.4.2 Insert . . . 111
6.4.3 Look up . . . 114
6.5 Applications of trie and prefix tree . . . 115
6.5.1 Dictionary and input completion . . . 115
6.5.2 Predictive text input . . . 117
6.6 Summary . . . 119
6.7 Appendix: Example programs . . . 120

7 B-Tree 125
7.1 Introduction . . . 125
7.2 Insert . . . 127
7.2.1 Insert then split . . . 127
7.2.2 Split before insert . . . 130
7.2.3 Paired lists . . . 132
7.3 Look up . . . 134
7.4 Delete . . . 136
7.4.1 Delete and fix . . . 136
7.4.2 Merge before delete . . . 139
7.5 Summary . . . 142
7.6 Appendix: Example programs . . . 143

8 Binary Heaps 147
8.1 Definition . . . 147
8.2 Binary heap by array . . . 147
8.2.1 Heapify . . . 148
8.2.2 Build . . . 149
8.2.3 Heap operations . . . 150
Pop . . . 150
Top-k . . . 150
Increase priority . . . 152
Insertion . . . 152
8.2.4 Heap sort . . . 152
8.3 Leftist heap and skew heap . . . 153
8.3.1 Leftist heap . . . 154

Merge . . . 155
Top and pop . . . 155
Insert . . . 156
Heap sort . . . 156
8.3.2 Skew heap . . . 156
Merge . . . 157
8.4 Splay heap . . . 157
8.4.1 Splay . . . 158
8.4.2 Pop . . . 161
8.4.3 Merge . . . 162
8.5 Summary . . . 162
8.6 Appendix - example programs . . . 162

**9 Selection sort** **167**
9.1 Introduction . . . 167
9.2 Find the minimum . . . 168
9.2.1 Performance . . . 169
9.3 Improvement . . . 169
9.3.1 Cock-tail sort . . . 170
9.4 Further improvement . . . 172
9.4.1 Tournament knock out . . . 172
9.4.2 Heap sort . . . 175
9.5 Appendix - example programs . . . 176

**10 Binomial heap, Fibonacci heap, and pairing heap** **179**
10.1 Introduction . . . 179
10.2 Binomial Heaps . . . 179
Binomial tree . . . 180
10.2.1 Link . . . 182
Insert . . . 183
10.2.2 Merge . . . 184
Pop . . . 185
10.3 Fibonacci heap . . . 186
10.3.1 Insert . . . 187
Merge . . . 187
Pop . . . 188
10.3.2 Increase priority . . . 192
10.3.3 The name of Fibonacci heap . . . 193
10.4 Pairing Heaps . . . 194
10.4.1 Definition . . . 195
10.4.2 Merge, insert, and top . . . 195
10.4.3 Increase priority . . . 195
10.4.4 Pop . . . 196
Delete . . . 197
10.5 Summary . . . 198
10.6 Appendix - example programs . . . 198

**11 Queue** **203**
11.1 Introduction . . . 203
11.2 Linked-list queue . . . 203
11.3 Circular buffer . . . 204
11.4 Paired-list queue . . . 206

11.5 Balance Queue . . . . . 207
11.6 Real-time queue . . . . . 207
11.7 Lazy real-time queue . . . . . 210
11.8 Appendix - example programs . . . . . 211

**12 Sequence** **213**
12.1 Introduction . . . . . 213
12.2 Binary random access list . . . . . 213
12.3 Numeric representation . . . . . 216
12.4 paired-array sequence . . . . . 219
12.5 Concatenate-able list . . . . . 220
12.6 Finger tree . . . . . 221
12.6.1 Insert . . . . . 222
12.6.2 Extract . . . . . 223
12.6.3 Append and remove . . . . . 225
12.6.4 concatenate . . . . . 225
12.6.5 Random access . . . . . 226
12.7 Appendix - example programs . . . . . 228

**13 Quick sort and merge sort** **233**
13.1 Introduction . . . . . 233
13.2 Quick sort . . . . . 233
13.2.1 Partition . . . . . 234
13.2.2 In-place sort . . . . . 235
13.2.3 Performance . . . . . 237
Average case★ . . . . . 237
13.2.4 Improvement . . . . . 239
Worst cases . . . . . 243
13.2.5 quick sort and tree sort . . . . . 246
13.3 Merge sort . . . . . 246
13.3.1 Merge . . . . . 247
13.3.2 Performance . . . . . 248
Improvement . . . . . 249
13.3.3 In-place merge sort . . . . . 250
13.3.4 Nature merge sort . . . . . 253
13.3.5 Bottom-up merge sort . . . . . 256
13.4 Parallelism . . . . . 257
13.5 Summary . . . . . 257
13.6 Appendix: Example programs . . . . . 258

**14 Searching** **261**
14.1 Introduction . . . . . 261
14.2 Sequence search . . . . . 261
14.2.1 Divide and conquer search . . . . . 261
$k$-selection problem . . . . . 262
binary search . . . . . 265
2 dimensions search . . . . . 268
Brute-force 2D search . . . . . 269
Saddleback search . . . . . 269
Improved saddleback search . . . . . 271
More improvement to saddleback search . . . . . 275
14.2.2 Information reuse . . . . . 280

Boyer-Moore majority number . . . 280
Maximum sum of sub vector . . . 284
KMP . . . 285
Purely functional KMP algorithm . . . 288
Boyer-Moore . . . 296
The bad character heuristics . . . 296
The good suffix heuristics . . . 299
14.3 Solution searching . . . 305
14.3.1 DFS and BFS . . . 305
Maze . . . 305
Eight queens puzzle . . . 311
Peg puzzle . . . 313
Summary of DFS . . . 317
The wolf, goat, and cabbage puzzle . . . 319
Water jugs puzzle . . . 323
Kloski . . . 330
Summary of BFS . . . 336
14.3.2 Search the optimal solution . . . 338
Grady algorithm . . . 338
Huffman coding . . . 338
Change-making problem . . . 347
Summary of greedy method . . . 348
Dynamic programming . . . 348
Properties of dynamic programming . . . 353
Longest common subsequence problem . . . 353
Subset sum problem . . . 358
14.4 Short summary . . . 363

**Appendices**

**A Imperative delete for red-black tree** **367**

**B AVL tree - proofs and the delete algorithm** **375**
B.1 Height increment . . . 375
B.2 Balance adjustment after insert . . . 376
B.3 Delete algorithm . . . 378
B.3.1 Functional delete . . . 379
B.3.2 Imperative delete . . . 380
B.4 Example program . . . 382

**GNU Free Documentation License** **391**
1. APPLICABILITY AND DEFINITIONS . . . 391
2. VERBATIM COPYING . . . 392
3. COPYING IN QUANTITY . . . 393
4. MODIFICATIONS . . . 393
5. COMBINING DOCUMENTS . . . 395
6. COLLECTIONS OF DOCUMENTS . . . 395
7. AGGREGATION WITH INDEPENDENT WORKS . . . 395
8. TRANSLATION . . . 396
9. TERMINATION . . . 396
10. FUTURE REVISIONS OF THIS LICENSE . . . 396
11. RELICENSING . . . 397

ADDENDUM: How to use this License for your documents . . . . . . . . . . . . 397

Programmers learn elementary algorithms at school. Except for programming contest, code interview, they seldom use algorithms in commercial software development. When talking about algorithms in AI and machine learning, it actually means 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'.

I would say elementary algorithms are critical in solving 'interesting problems', the usefulness of the problem set aside. Let's start with two problems.

## 0.1 The smallest free number

Richard Bird gives an interesting programming problem to find the minimum number that not appears in a given list(Chapter 1,[? ]). It's common to use a number as the identifier (Id) to index entities. At any time, a number is either occupied or free. When client tries to acquire a new number as index, 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, which is 10, from the list? It seems quite easy to figure out the solution.

1: **function** MIN-FREE($A$)
2: $\quad x \leftarrow 0$
3: $\quad$ **loop**
4: $\quad\quad$ **if** $x \notin A$ **then**
5: $\quad\quad\quad$ **return** $x$
6: $\quad\quad$ **else**
7: $\quad\quad\quad x \leftarrow x + 1$

Where the $\notin$ is realized like below.

1: **function** '$\notin$'$(x, X)$
2: $\quad$ **for** $i \leftarrow 1$ to $|X|$ **do**
3: $\quad\quad$ **if** $x = X[i]$ **then**
4: $\quad\quad\quad$ **return** False
5: $\quad$ **return** True

Some environments have built-in implementation to test if an element is in a list. Below is an example program.

```
def minfree(lst):
    i = 0
    while True:
        if i not in lst:
            return i
        i = i + 1
```

However, when there are millions of numbers being used, 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.

### 0.1.1 Improvement

The key idea to improve the solution is based on the fact that, for $n$ numbers $x_1, x_2, ..., x_n$, if there exists free number, some $x_i$ must be outside 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. In summary:

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

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

1: **function** MIN-FREE($A$)
2: $F \leftarrow$[False, False, ..., False] where $|F| = n + 1$
3: **for** $\forall x \in A$ **do**
4: **if** $x < n$ **then**
5: $F[x] \leftarrow$ True
6: **for** $i \leftarrow [0, n]$ **do**
7: **if** $F[i] =$ False **then**
8: **return** $i$

Line 2 initializes a flag array all of False values. Then we scan all numbers in $A$ and mark the corresponding flag to True if the value is less than $n$. Finally, we iterate to find the first False flag. This program takes time proportion to $n$. It uses $n + 1$ flags to cover the special case that $sorted(A) = [0, 1, 2, ..., n-1]$. This solution is much faster than the brute force one. In the same computer, the Python implementation takes 0.02s when dealing with 100,000 numbers.

Although this solution only takes $O(n)$ time, it needs additional $O(n)$ space to store the flags. We haven't tuned it yet. Each time the program allocates memory to create an array of $n + 1$ flags, then releases it when finish. Such memory allocation and release is expensive and cost a lot of processing time.

To improve it, we can allocate the memory in advance for later reusing, and change to bit-wise flags instead of array. For example as the following C program:

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

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

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

unsigned int bits[N / WORD_LENGTH + 1];

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

```
      }
    }
}
```

This program can handle 1 million numbers in 0.023s in the same computer.

### 0.1.2 Divide and Conquer

The above improvement costs $O(n)$ additional space for flags, can we eliminate it? The divide and conquer strategy is to break the problem into smaller ones, then solve them separately to get the answer.

We can put numbers $x_i \leq \lfloor n/2 \rfloor$ into a sub-list $A'$ and put 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 $A''$. We can recursively search in $A''$ which is shorter the original list. Otherwise, it means the minimum free number is in $A'$, which again leads to a smaller problem.

When search in $A''$, the conditions change a bit. We do not start from 0, but from $\lfloor n/2 \rfloor + 1$ as the new lower bound. We define the algorithm as $search(A, l, u)$, where $l$ is the lower bound and $u$ is the upper bound index. For the empty list as a special case, we return the $l$ as the result.

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

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

where

$$\begin{array}{rcl} m & = & \lfloor \frac{l + u}{2} \rfloor \\ A' & = & [x | x \in A, x \leq m] \\ A'' & = & [x | x \in A, x > m] \end{array}$$

This algorithm doesn't need additional space[1]. Each recursive call performs $O(|A|)$ comparisons to build $A'$ and $A''$. After that the problem scale halves. Therefore, the time is bound to $T(n) = T(n/2) + O(n)$, which reduce to $O(n)$ according to master theorem. Alternatively, observe that the first call takes $O(n)$ to build $A'$ and $A''$ and the second call takes $O(n/2)$, and $O(n/4)$ for the third... The total time is $O(n + n/2 + n/4 + ...) = O(2n) = O(n)$. We use $[a | a \in A, p(a)]$ for list. It is different with $\{a | a \in A, p(a)\}$, which is a set.

Below example Haskell program implements this algorithm.

```
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
```

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

### 0.1.3 Expressiveness and performance

One may concern the performance of this divide and conquer algorithm. There are $O(\lg n)$ recursive calls, which need additional stack space. If wanted, we can eliminate the recursion:

```
1: function MIN-FREE(A)
2:     l ← 0, u ← |A|
3:     while u − l > 0 do
4:         m ← l + (u − l)/2
5:         left ← l
6:         for right ← l to u − 1 do
7:             if A[right] ≤ m then
8:                 A[left] ↔ A[right]
9:                 left ← left + 1
10:        if left < m + 1 then
11:            u ← left
12:        else
13:            l ← 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 $m$.

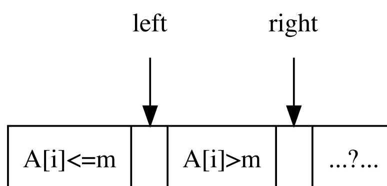



Figure 1: Divide the array, all $A[i] \leq m$ where $0 \leq i < left$; while all $A[i] > m$ where $left \leq i < right$. The rest elements haven't been processed yet.

This solution is fast and needn't extra stack space. However, compare to the previous recursive one, there is some expressiveness drops. Depends on individual taste, one may prefer one over the other.

## 0.2 Regular number

The second puzzle is to find the 1,500-th number, which only contains factor 2, 3 or 5. Such numbers are called regular number, also known as 5-smooth indicating the greatest prime factor is at most 5, or Hamming numbers named after Richard Hamming in computer science. 2, 3, and 5 are of course regular numbers. $60 = 2^2 3^1 5^1$ is the 25-th number. $21 = 2^0 3^1 7^1$ is not valid because it has a factor 7. We consider $1 = 2^0 3^0 5^0$ be the 0-th regular number. The first 10 regular numbers are:

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

### 0.2.1 The brute-force solution

The straightforward way is to check numbers one by one from 1, extract all factors of 2, 3 and 5 to see if the left part is 1:

```
1: function REGULAR-NUMBER(n)
```

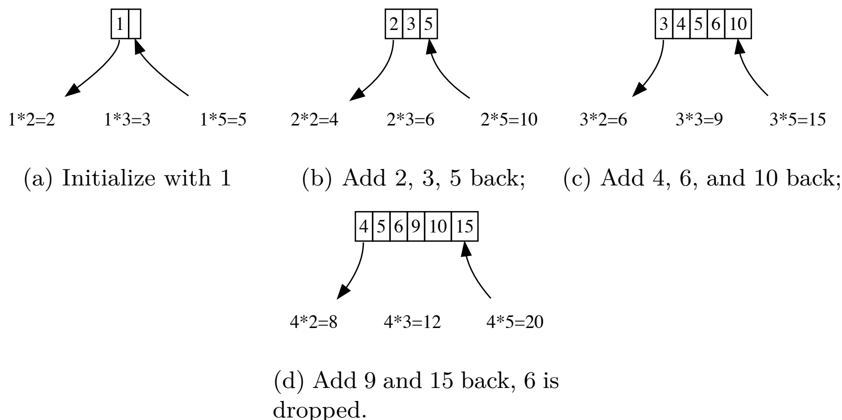
```
2:     x ← 1
3:     while n > 0 do
4:         x ← x + 1
5:         if VALID?(x) then
6:             n ← n − 1
7:     return x

8: function VALID?(x)
9:     while x mod 2 = 0 do
10:        x ← ⌊x/2⌋
11:    while x mod 3 = 0 do
12:        x ← ⌊x/3⌋
13:    while x mod 5 = 0 do
14:        x ← ⌊x/5⌋
15:    return x = 1 ?
```

This 'brute-force' algorithm works for small $n$. However, to find the 1500-th regular number (which is 860934420), its C implementation takes 40.39s in above computer. When $n$ increases to 15,000, it can't terminate after 10 minutes.

### 0.2.2 Improvement

Modular and divide calculations are very expensive[2]. And they are executed a lot in loops. Instead of checking if a number only contains 2, 3, or 5 as factors, we can construct regular number from these three factors. We can start from 1, multiply it with 2, 3, or 5 to generate the rest numbers. The problem turns to be how to generate regular numbers in order? One method is to utilize the queue data structure.

A queue allows to add element to one end (called enqueue), and delete from the other (called dequeue). The element enqueued first will be dequeued first. This nature is called FIFO (First In First Out). The idea is to add 1 as the first number to the queue. We repeatedly dequeue a number, multiply it with 2, 3, and 5, to generate 3 new numbers; then add them back to the queue in order. A new generated number may already exist in the queue. In such case, we drop the duplicated number. Because the new number may be smaller than the others in the queue, we must put them at the correct position. Figure 2 shows this idea.




(a) Initialize with 1 (b) Add 2, 3, 5 back; (c) Add 4, 6, and 10 back;

(d) Add 9 and 15 back, 6 is dropped.

Figure 2: First 4 steps to generate regular numbers.

We can design the algorithm based on this idea:

```
1: function Regular-Number(n)
2:     Q ← ∅
3:     x ← 1
4:     Enqueue(Q, x)
5:     while n > 0 do
6:         x ← Dequeue(Q)
7:         Unique-Enqueue(Q, 2x)
8:         Unique-Enqueue(Q, 3x)
9:         Unique-Enqueue(Q, 5x)
10:        n ← n − 1
11:    return x

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

The INSERT function takes $O(m)$ time to insert a number at proper position, where $m = |Q|$ is the length of the queue. It skips insertion if the number already exists. The length of the queue increases proportion to $n$ (Each time, we dequeue an element, and enqueue 3 new at most. The increase ratio $\leq 2$), the total time is $O(1 + 2 + 3 + ... + n) = O(n^2)$.

Figure3 shows the number of access to the queue against $n$. It is a quadratic curve, which reflects the $O(n^2)$ performance.

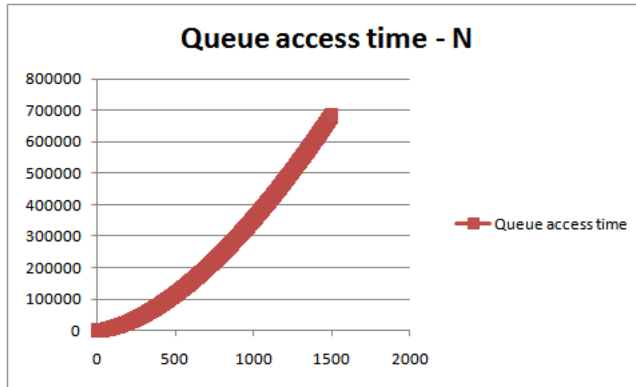



Figure 3: Queue access count - $n$.

The corresponding C implementation takes 0.016s to output 860934420. It is about 2500 times faster than the naive search solution.

We can also realize this improvement recursively. Suppose $X$ is an infinite list of all regular numbers $[x_1, x_2, x_3, ...]$. For every number, we multiply it by 2, the result is still a list of regular numbers: $[2x_1, 2x_2, 2x_3, ...]$. We can also multiply numbers in $X$ by 3 and 5, to generate two new infinite lists. If we merge them together, remove the duplicated numbers, and prepend 1 as the first, then we get $X$ again. In other words, the following equation holds:

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

Where symbol $x : X$ means to link $x$ before list $X$, such that $x$ becomes the first element. It is called 'cons' in Lisp. We link 1 before the rest, as it is the first regular number. To implement infinite lists merge, we define $\cup$ to recursively compare elements in two sorted lists. Let $X = [x_1, x_2, x_3 ...]$, $Y = [y_1, y_2, y_3, ...]$ be two such lists, $X' = [x_2, x_3, ...]$ and $Y' = [y_2, y_3, ...]$ contain the rest elements except without their heads $x_1$ and $y_1$. We define merge as below:

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

We need not concern about either $X$ or $Y$ is empty, because they are both infinite lists. In functional settings that support lazy evaluation, this algorithm can be implemented as the following example program:

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

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

The 1500th number 860934420 is given by `ns !! 1500`. In the same computer, it takes about 0.03s to output the answer.

### 0.2.3 Queues

Although the improved solution is much faster than the original brute-force one, it generates duplicated numbers, and they are eventually dropped. In order to keep numbers ordered, it needs linear time scan and insertion, which degrades the enqueue operation from constant time to $O(|Q|)$. To avoid duplication, we can separate 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. The bucket is realized as a queue. They are initialized as $Q_2 = \{2\}$, $Q_{23} = \{3\}$, and $Q_{235} = \{5\}$. Starting from 1, each time we extract the smallest number $x$ from the three queues as the next regular number. Then do the following:

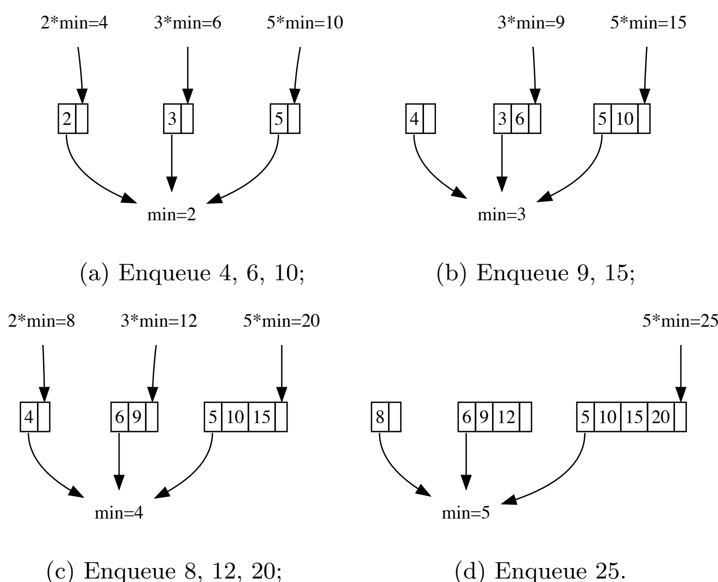
- If $x$ comes from $Q_2$, we enqueue $2x$, $3x$, and $5x$ back to $Q_2$, $Q_{23}$, and $Q_{235}$ respectively;

- If $x$ comes from $Q_{23}$, we only enqueue $3x$ to $Q_{23}$, and $5x$ to $Q_{235}$. We should not add $2x$ to $Q_2$, because $Q_2$ cannot hold any numbers divided by 3.

- If $x$ comes from $Q_{235}$, we only need enqueue $5x$ to $Q_{235}$. We should not add $2x$ to $Q_2$, or $3x$ to $Q_{23}$ because they can't hold numbers divided by 5.

We reach to the answer after repeatedly enqueue the smallest number $n$ times. The following algorithm implements this idea:

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




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

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

Figure 4: First 4 steps with $Q_2$, $Q_{23}$, and $Q_{235}$. They were initialize with 2, 3, 5.

9: ENQUEUE($Q_{23}, 3x$)
10: ENQUEUE($Q_{235}, 5x$)
11: **else if** $x =$ HEAD($Q_{23}$) **then**
12: DEQUEUE($Q_{23}$)
13: ENQUEUE($Q_{23}, 3x$)
14: ENQUEUE($Q_{235}, 5x$)
15: **else**
16: DEQUEUE($Q_{235}$)
17: ENQUEUE($Q_{235}, 5x$)
18: $n \leftarrow n - 1$
19: **return** $x$

This algorithm loops on $n$. Each time it extracts the minimum number from the head of three queues. This takes constant time. Then it add at most 3 numbers to each queue respectively. This takes constant time too. Therefore the algorithm is bound to $O(n)$.

## 0.3 Summary

One might think the brute-force solution was sufficient to solve both programming puzzles. However, as the problem scales up, we have to seek for better solutions. There are many interesting problems, which were hard before, but through computer programming, we are able to solve them nowadays. This book aims to provide both functional and imperative definition for the commonly used elementary algorithms and data structures. We referenced many results from Okasaki's work[3] and classic text books(for example[4]). We try to avoid relying on a specific programming language, because it may or may not be familiar with the reader, 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 C, Java, and Python. They are only for illustration purpose, but not guaranteed following any language specification strictly.

## 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) {
    nums = Int[m + 1]
    n = 0, i = 0, j = 0, k = 0
    nums[0] = 1
    x2 = 2 * nums[i]
    x3 = 3 * nums[j]
    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 the preliminary build blocks to create complex data structure. Both can hold multiple elements as a container. Array is trivially implemented as a range of consecutive cells indexed by a number. The number is called address or position. Array is typically bounded. Its size need be determined before using. While list increases on-demand to hold additional elements. One can traverse a list one by one from head to tail. Particularly in functional settings, the list related algorithms play critical roles to control the computation and logic structure[1]. Readers already familiar with map, filter, fold algorithms are safe to skip this chapter, and directly start from chapter 2.

## 1.2 Definition

List, also known as singly linked-list is a data structure recursively defined as below:

- 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 node contains two part, an element called key, and a reference to the sub-list called next. The sub-list reference in the last node is empty, marked as 'NIL'.

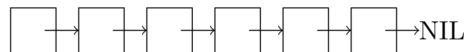



Figure 1.1: A list of nodes

Every node links to the next one or NIL. Linked-list is often defined through compound structure[2], for example:

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

[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.

It needs more clarification for the empty list. Many traditional environments support *null* concept. There are two different ways to represent empty list. One is to use null (or NIL) directly; the other is to construct a list, but put nothing as `[]`. From implementation perspective, null need not allocate any memory, while `[]` does. In this book, we use $\varnothing$ to represent generic empty list, set, or container.

### 1.2.1 Access

Given a none empty list $L$, we need define two functions to access its first element, and the rest sub-list. They are often called $first(L), rest(L)$ or $head(L), tail(L)$[3]. On the other hand, we can construct a list from an element $x$ and another list $xs$ (can be empty), denoted as $x : xs$. It is also called the `cons` operation. We have the following equations hold:

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

For a none empty list $X$, we will also use $x_1$ for the first element, and use $X'$ for the rest sub-list. 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: for empty list, the length is zero, otherwise, it is the length of the sub-list plus one.

$$\begin{array}{rcl} length(\varnothing) & = & 0 \\ length(L) & = & 1 + length(L') \end{array} \tag{1.2}$$

In order to count the length, this algorithm traverses all the elements from head to end, hence it is bound to $O(n)$ time, where $n$ is the number of elements. To avoid repeatedly counting, we can also persist the length in a variable, and update it when mutate (add or delete) the list. Below is the iterative way to count length:

```
1: function LENGTH(L)
2:     n ← 0
3:     while L ≠ NIL do
4:         n ← n + 1
5:         L ← NEXT(L)
6:     return n
```

We will also use notion $|L|$ for the length of list $L$ when the context is clear.

### 1.3.1 index

Different from array, which supports random access an element at position $i$ in constant time, we need traverse the list $i$ steps to access the target element.

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

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

In order to get the $i$-th element from a none empty list:

- if $i$ is 0, the result is the first element;
- Otherwise, the result is the $(i-1)$-th element in the sub-list.

We intend to leave the empty list not handled. The behavior when pass $\varnothing$ is undefined. As such, the out of bound case also leads to undefined behavior. If $i > |L|$ exceeds the length, we end up the edge case to access the $(i - |L|)$-th element of the empty list. On the other hand, if $i < 0$, minus it by one makes it even farther away from 0. We finally end up with the same situation that the index is negative, while the list is empty.

This algorithm is bound to $O(i)$ time as it advances the list $i$ steps. Below is the corresponding imperative implementation:

```
1: function Get-At(i, L)
2:     while i ≠ 0 do
3:         L ← Next(L)                    ▷ Raise error when L = NIL
4:         i ← i − 1
5:     return First(L)
```

## Exercise 1.3

1. In the iterative Get-At$(i, L)$ algorithm, what is the behavior when $L$ is empty? what is the behavior when $i$ is out of the bound or negative?

### 1.3.2 Last

There is a pair of symmetric operations to 'first/rest'. They are called 'last/init'. For a none empty list $X = [x_1, x_2, ..., x_n]$, function *last* returns the last 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 linear time, because we need traverse the whole list to tail.

When access the last element of list $X$:

- If the $X$ contains only one element as $[x_1]$, then $x_1$ is the last one;
- Otherwise, the result is the last element of the sub-list $X'$.

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

Similarly, when extract the sub-list of $X$ contains all elements without the last one:

- If $X$ is a singleton $[x_1]$, the result is empty $[\,]$;
- Otherwise, we recursively get the initial sub-list for $X'$, then prepend $x_1$ to it as the result.

$$
\begin{array}{rcl}
init([x]) & = & [\,] \\
init(x : xs) & = & x : init(xs)
\end{array}
\tag{1.5}
$$

We leave the empty list not handled for both operations. The behavior is undefined if pass $\varnothing$ in. Below are the iterative implementation:

```
1: function Last(L)
2:     x ← NIL
3:     while L ≠ NIL do
4:         x ← First(L)
```

5: $L \leftarrow$ REST($L$)
6: **return** $x$

7: **function** INIT($L$)
8: $L' \leftarrow$ NIL
9: **while** REST($L$) $\neq$ NIL **do** ▷ Raise error when $L$ is NIL
10: $L' \leftarrow$ CONS(FIRST($L$), $L'$)
11: $L \leftarrow$ REST($L$)
12: **return** REVERSE($L'$)

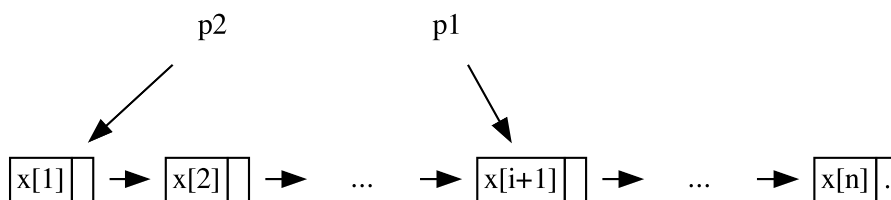
As advancing towards the tail, this algorithm accumulates the 'init' result through 'cons'. However, such result is in the reversed order. We need apply reverse (defined in section 1.4.2) again to return the correct result. There is a question to ask if we can use 'append' instead of 'cons' in the exercise.

### 1.3.3 Reverse index

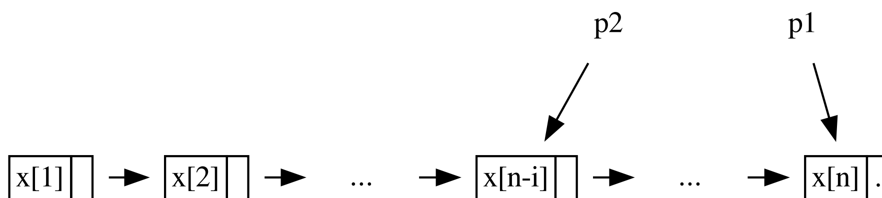
*last* is a special case of reverse index. The generic case is to find the last $i$-th element of a given list. The naive implementation takes two rounds of traverse: Determine the length $n$ through the first round; then access the $(n-i-1)$-th element through the second round:

$$lastAt(i, L) = getAt(|L| - i - 1, L) \tag{1.6}$$

There actually exists better solution. The idea is to keep two pointers $p_1, p_2$ with the distance $i$ between them. The equation $rest^i(p_2) = p_1$ holds, where $rest^i(p_2)$ means repleatedly apply $rest()$ function $i$ times. When succeed $p_2$ by $i$ steps gets $p_1$. We start by pointing $p_2$ to the list head, and advance both pointers in parallel till $p_1$ arrives at tail. At that time point, $p_2$ exactly points to the $i$-th element from right. Figure 1.2 shows this idea. As $p_1, p_2$ form a window, this method is also called 'sliding window' solution.




(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 formed by two pointers

1: **function** LAST-AT($i, L$)

```
2:    p ← L
3:    while i > 0 do
4:        L ← REST(L)                          ▷ Raise error if out of bound
5:        i ← i − 1
6:    while REST(L) ≠ NIL do
7:        L ← REST(L)
8:        p ← REST(p)
9:    return FIRST(p)
```

The functional implementation need special consideration as we cannot update pointers directly. 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.

- If $Y$ is a singleton list, i.e. $[x_n]$, then the last $i$-th element is the head of $X$;
- Otherwise, we drop the first element from both $X$ and $Y$, then recursively check $X'$ and $Y'$.

$$lastAt(i, X) = slide(X, drop(i, X)) \tag{1.7}$$

where function $slide(X, Y)$ drops the heads for both lists:

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

Function $drop(m, X)$ discards the first $m$ elements from list $X$. It can be implemented by advancing $X$ by $m$ steps:

$$\begin{array}{rcl} drop(0,\ X) & = & X \\ drop(m,\ \varnothing) & = & \varnothing \\ drop(m,\ x : xs) & = & drop(m - 1, xs) \end{array} \tag{1.9}$$

## Exercise 1.4

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

### 1.3.4 Mutate

Mutate operations include append, insert, update, and delete. Some functional environments actually implement mutate by creating a new list, while the original one is persisted for later reuse, or released at sometime (chapter 2 in[3]).

#### Append

Append is the symmetric operation of *cons*, it adds element on the tail instead of head. Because of this, it is also called 'snoc'. For linked-list, it means we need traverse to the tail, hence it takes $O(n)$ time, where $n$ is the length. To avoid repeatedly traverse, we can record the tail reference as a variable, and keep updating it upon changes.

$$\begin{array}{rcl} append(\varnothing, x) & = & [x] \\ append(y : ys, x) & = & y : append(ys, x) \end{array} \tag{1.10}$$

- If append $x$ to the empty list, the result is $[x]$;

- Otherwise, we firstly recursive append $x$ to the rest sub-list, then prepend the original head to form the result.

The corresponding iterative implementation is as the following:

1: **function** APPEND($L, x$)
2: **if** $L$ = NIL **then**
3: **return** CONS($x$, NIL)
4: $H \leftarrow L$ ▷ save the head
5: **while** REST($L$) $\neq$ NIL **do**
6: $L \leftarrow$ REST($L$)
7: REST($L$) $\leftarrow$ CONS($x$, NIL)
8: **return** $H$

Update the REST is typically implemented by setting the `next` reference field as shown in below example program.

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

## Exercise 1.5

1. Add a 'tail' field in list definition, optimize the append algorithm to constant time.

2. With the additional 'tail' field, when need we update the tail variable? How does it affect the performance?

### Set value

Similar to *getAt*, we need advance to the target position, then change the element there. To define function $setAt(i, x, L)$:

- If $i = 0$, it means we are changing the first element, the result is $x : L'$;

- Otherwise, we need recursively set the value at position $i - 1$ for the sub-list $L'$.

$$\begin{array}{rcl} setAt(0, x,\ y : ys) & = & x : ys \\ setAt(i, x,\ y : ys) & = & y : setAt(i-1, x, ys) \end{array} \tag{1.11}$$

This algorithm is bound to $O(i)$ time, where $i$ is the position to update.

## Exercise 1.6

1. Handle the empty list and out of bound error for *setAt*.

### insert

There are two different cases about insertion. One is to insert an element at a given position: $insert(i, x, L)$. The algorithm is similar to $setAt$; The other is to insert an element to a sorted list, and keep the order still sorted.

To insert $x$ at position $i$, we need firstly advance $i$ steps, then construct a new sub-list with $x$ as the head, then concatenate it to the first $i$ elements[4].

- If $i = 0$, it then turns to be a 'cons' operation: $x : L$;
- Otherwise, we recursively insert $x$ to $L'$ at position $i - 1$; then prepend the original head.

$$\begin{array}{rcl} insert(0, x,\ L) & = & x : L \\ insert(i, x,\ y : ys) & = & x : insert(i - 1, x, ys) \end{array} \tag{1.12}$$

When $i$ exceeds the list length, we can treat it as to append $x$. We leave this as an exercise. The following is the corresponding iterative implementation:

```
1: function Insert(i, x, L)
2:     if i = 0 then
3:         return Cons(x, L)
4:     H ← L
5:     p ← L
6:     while i > 0 and L ≠ NIL do
7:         p ← L
8:         L ← Rest(L)
9:         i ← i − 1
10:    Rest(p) ← Cons(x, L)
11:    return H
```

If the list $L = [x_1, x_2, ..., x_n]$ is sorted, i.e. for any position $1 \leq i \leq j \leq n$, then $x_i \leq x_j$ holds. Here $\leq$ is abstract ordering. It can actually mean $\geq$ for descending order, or subset relationship etc. We can design the insert algorithm to maintain the sorted order. To insert element $x$ to a sorted list $L$:

- If either $L$ is empty or $x$ is not greater than the first element in $L$, we prepend $x$ to $L$ and returns $x : L$;
- Otherwise, we recursively insert $x$ to the sub-list $L'$.

$$\begin{array}{rcl} insert(x,\ \varnothing) & = & [x] \\ insert(x,\ y : ys) & = & \begin{cases} x \leq y : & x : y : ys \\ otherwise : & y : insert(x, ys) \end{cases} \end{array} \tag{1.13}$$

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

```
1: function Insert(x, L)
2:     if L = NIL or x < First(L) then
3:         return Cons(x, L)
4:     H ← L
5:     while Rest(L) ≠ NIL and First(Rest(L)) < x do
6:         L ← Rest(L)
```

[4] $i$ starts from 0.

7: $\text{REST}(L) \leftarrow \text{CONS}(x, \text{REST}(L))$
8: **return** $H$

With this linear time ordered insertion defined, we can further develop the insertion-sort algorithm. The idea is to repeatedly insert elements to the empty list. Since each insert takes liner time, the overall sort is bound to $O(n^2)$.

$$
\begin{array}{rcl}
sort(\varnothing) & = & \varnothing \\
sort(x : xs) & = & insert(x, sort(xs))
\end{array}
\tag{1.14}
$$

This is a recursive algorithm. It firstly sorts the sub-list, then inserts the first element in it. We can eliminate the recursion to develop a iterative implementation. The idea is to scan the list, and one by one insert them:

1: **function** $\text{SORT}(L)$
2: $\quad S \leftarrow \text{NIL}$
3: $\quad$ **while** $L \neq \text{NIL}$ **do**
4: $\quad\quad S \leftarrow \text{INSERT}(\text{FIRST}(L), S)$
5: $\quad\quad L \leftarrow \text{REST}(L)$
6: $\quad$ **return** $S$

At any time during the loop, the result is sorted. There is a major difference between the recursive and the iterative implementations. The recursive one processes the list from right, while the iterative one is from left. We'll introduce 'tail-recursion' in section 1.3.5 to eliminate this difference. Chapter 3 introduces insertion sort in detail, including performance analysis and optimization.

## Exercise 1.7

1. Handle the out-of-bound case in insertion, and treat it as append.
2. Design the insertion algorithm for array. When insert at position $i$, all elements after $i$ need shift to the end by one.
3. Implement the insertion sort only with less than ($<$) defined.

### delete

Symmetric to insert, delete also has two cases. One is to delete the element at a position; the other is to look up, then delete the element of a given value. The first case is defined as $delAt(i, L)$, the second case is defined as $delete(x, L)$.

To delete the element at position $i$, we need advance $i$ steps to the target position, then by pass the element, and link the rest sub-list.

- If $L$ is empty, then the result is empty too;
- If $i = 0$, we are deleting the head, the result is $L'$;
- Otherwise, recursively delete the $(i-1)$-th element from $L'$, then prepend the original head as the result.

$$
\begin{array}{rcl}
delAt(i,\ \varnothing) & = & \varnothing \\
delAt(0,\ x : xs) & = & xs \\
delAt(i,\ x : xs) & = & x : delAt(i-1, xs)
\end{array}
\tag{1.15}
$$

This algorithm is bound to $O(i)$ as we need advance $i$ steps to perform deleting. Below is the iterative implementation:

1: **function** $\text{DEL-AT}(i, L)$

```
2:     S ← Cons(⊥, L)                                    ▷ A sentinel node
3:     p ← S
4:     while i > 0 and L ≠ NIL do
5:         i ← i − 1
6:         p ← L
7:         L ← Rest(L)
8:     if L ≠ NIL then
9:         Rest(p) ← Rest(L)
10:    return Rest(S)
```

To simplify the implementation, we introduce a sentinel node $S$, it contains a special value $\perp$, and its next reference points to $L$. With $S$, we are save to cut-off any node in $L$ even for the first one. Finally, we return the list after $S$ as the result, and $S$ itself can be discarded.

For the 'find and delete' case, there are two options. We can either find and delete the first occurrence of a value; or remove all the occurrences. The later is more generic, we leave it as an exercise. When delete $x$ from list $L$:

- If the list is empty, the result is $\varnothing$;
- Otherwise, we compare the head and $x$, if they are equal, then the result is $L'$;
- If the head does not equal to $x$, we keep the head, and recursively delete $x$ in $L'$.

$$
\begin{array}{rcl}
delete(x,\ \varnothing) & = & \varnothing \\
delete(x,\ y:ys) & = & \begin{cases} x = y: & ys \\ x \neq y: & y : delete(x, ys) \end{cases}
\end{array}
\tag{1.16}
$$

This algorithm is bound to $O(n)$ time, where $n$ is the length, as it need scan the list to find the target element. For the iterative implementation, we also introduce a sentinel node to simplify the logic:

```
1: function Delete(x, L)
2:     S ← Cons(⊥, L)
3:     p ← L
4:     while L ≠ NIL and First(L) ≠ x do
5:         p ← L
6:         L ← Rest(L)
7:     if L ≠ NIL then
8:         Rest(p) ← Rest(L)
9:     return Rest(S)
```

## Exercise 1.8

1. Design 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 by one.

### concatenate

Append is a special case for concatenation. Append only adds one element, while concatenation adds multiple ones. However, the performance would be quadratic if repeatedly appending as below:

$$
\begin{array}{rcl}
X \mathbin{+\!\!+} \varnothing & = & X \\
X \mathbin{+\!\!+} (y:ys) & = & append(X, y) \mathbin{+\!\!+} ys
\end{array}
\tag{1.17}
$$

In this implementation when concatenate $X$ and $Y$, each append operation traverses to the tail, and we do this for $|Y|$ times. the total time is bound to $O(|X| + (|X| + 1) + ... + (|X| + |Y|)) = O(|X||Y| + |Y|^2)$. Consider the link (cons) operation is fast (constant time), we can traverse to the tail of $X$ only once, then link $Y$ to the tail.

- If $X$ is empty, the result is $Y$;
- Otherwise, we concatenate the sub-list $X'$ with $Y$, then prepend the head as the result.

We can further improve it a bit: when $Y$ is empty, we needn't traverse, but directly return $X$:

$$\begin{array}{rcl} \varnothing \mathbin{+\!\!+} Y &=& Y \\ X \mathbin{+\!\!+} \varnothing &=& X \\ (x : xs) \mathbin{+\!\!+} Y &=& x : (xs \mathbin{+\!\!+} Y) \end{array} \tag{1.18}$$

The modified algorithm only traverse list $X$, then link its tail to $Y$, hence it is bound $O(|X|)$ time. In imperative settings, concatenation can be realized in constant time with the additional tail variable. We leave its implementation as exercise. Below is the iterative implementation without using the tail variable:

```
1: function Concat(X, Y)
2:     if X = NIL then
3:         return Y
4:     if Y = NIL then
5:         return X
6:     H ← X
7:     while Rest(X) ≠ NIL do
8:         X ← Rest(X)
9:     Rest(X) ← Y
10:    return H
```

## 1.3.5 sum and product

It is common to calculate the sum or product of a list of numbers. They have almost same structure. We will introduce how to abstract them to higher order computation in section 1.6.

### Recursive sum and product

To calculate the sum of a list:

- If the list is empty, the result is zero;
- Otherwise, the result is the first element plus the sum of the rest.

$$\begin{array}{rcl} sum(\varnothing) &=& 0 \\ sum(x : xs) &=& x + sum(xs) \end{array} \tag{1.19}$$

We can't merely replace $+$ to $\times$ to obtain product algorithm, because it always returns zero. We need define the product of the empty list as 1.

$$\begin{array}{rcl} product(\varnothing) &=& 1 \\ product(x : xs) &=& x \cdot product(xs) \end{array} \tag{1.20}$$

Both algorithms traverse the list, hence are bound to $O(n)$ time, where $n$ is the length.

### Tail call recursion

Both sum and product algorithms calculate from right to left. We can change them to calculate the *accumulated* result from left to right. For sum, it accumulates from 0, then adds element one by one; while for product, it starts from 1, then repeatedly multiplying elements. The accumulate process can be defined as:

- If the list is empty, return the accumulated result;
- Otherwise, accumulate the first element to the result, then go on accumulating.

Below are the accumulated sum and product:

$$\begin{array}{rlrl} sum'(A,\ \varnothing) = & A & prod'(A,\ \varnothing) = & A \\ sum'(A,\ x:xs) = & sum(x+A, xs) & prod'(A,\ x:xs) = & prod'(x \cdot A, xs) \end{array} \tag{1.21}$$

Given a list, we can call $sum'$ with 0, and $prod'$ with 1:

$$sum(X) = sum'(0, X) \qquad product(X) = prod'(1, X) \tag{1.22}$$

Or merely simplify it to 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 pass one argument $x$, it ends up to another function of $y$: $g(y) = f(x, y)$ or $g = f\ x$. We can further extend it to multiple variables, that $f(x, y, ..., z)$ can be Curried to a series of functions: $f, f\ x, f\ x\ y, ...$. No matter how many variables, we can treat them as a series of Curried function, each has only one parameter: $f(x, y, ..., z) = f(x)(y)...(z) = f\ x\ y\ ...\ z$.

The accumulated sum does not only calculate the result from left to right, it needn't book keeping any context, state, or intermediate result for recursion. All such states are either passed as argument (i.e. $A$), or can be dropped (the previous element in the list). Such recursive calls are often optimized as pure loops in practice. We call this kind of function as *tail recursion* (or 'tail call'), and the optimization to eliminate recursion is called 'tail recursion optimization'[61], because the recursion happens at the tail place in the function. The performance of tail call can be greatly improved after optimization, and we can avoid the issue of stack overflow in deep recursions.

In section 1.3.4 about insertion sort, we mentioned the recursive algorithm sorts elements form right. We can also optimize it to tail call:

$$\begin{array}{rcl} sort'(A,\ \varnothing) & = & A \\ sort'(A,\ x:xs) & = & sort'(insert(x, A), xs) \end{array} \tag{1.23}$$

And the sort is defined in Curried form with $\varnothing$ as the start value:

$$sort = sort'(\varnothing) \tag{1.24}$$

As a typical tail call problem, let's consider how to compute $b^n$ effectively? (refer to problem 1.16 in[63].) A brute-force solution is to repeatedly multiplying $b$ for $n$ times from 1. This algorithm is bound to $O(n)$:

1: **function** Pow($b, n$)
2: $\quad x \leftarrow 1$
3: $\quad$ **loop** $n$ times

```
4:        x ← x · b
5:    return x
```

Actually, the solution can be greatly improved. 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 $(b^4)^2 = b^8$. Thus we only need loop 3 times, but not 8 times.

Based on this idea, if $n = 2^m$ for some none negative integer $m$, we can design below algorithm to compute $b^n$:

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

We next extend this divide and conquer method for 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;
- Otherwise $n$ is odd. Since $n - 1$ is even, we recursively compute $b^{n-1}$, the multiply $b$ atop it.

$$\begin{array}{rcl} b^0 & = & 1 \\ b^n & = & \begin{cases} 2|n: & (b^{\frac{n}{2}})^2 \\ otherwise: & b \cdot b^{n-1} \end{cases} \end{array} \tag{1.25}$$

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

$$\begin{array}{rcl} b^0 & = & 1 \\ b^n & = & \begin{cases} 2|n: & (b^2)^{\frac{n}{2}} \\ otherwise: & b \cdot b^{n-1} \end{cases} \end{array} \tag{1.26}$$

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

$$\begin{array}{rcl} pow(b, 0, A) & = & A \\ pow(b, n, A) & = & \begin{cases} 2|n: & pow(b^2, \frac{n}{2}, A) \\ otherwise: & pow(b, n-1, b \cdot A) \end{cases} \end{array} \tag{1.27}$$

Compare to the brute-force implementation, this one improves to $O(\lg n)$ time. Actually, we can improve it further. If represent $n$ in binary format $n = (a_m a_{m-1} ... a_1 a_0)_2$, we clearly know that the computation for $b^{2^i}$ is necessary if $a_i = 1$. This is quite similar to the idea of Binomial heap (section 10.2). We can multiplying all of them for bits of 1.

For example, when compute $b^{11}$, as $11 = (1011)_2 = 2^3 + 2 + 1$, thus $b^{11} = b^{2^3} \times b^2 \times b$. We get the result by these steps:

1. calculate $b^1$, which is $b$;
2. Square to $b^2$ from the previous result;
3. Square again to $b^{2^2}$ from step 2;
4. Square to $b^{2^3}$ from step 3.

Finally, we multiply the result of step 1, 2, and 4 to get $b^{11}$. Summarize this idea, we improve the algorithm as below.

$$
\begin{array}{rcl}
pow(b, 0, A) & = & A \\
pow(b, n, A) & = & \begin{cases} 2|n : & pow(b^2, \frac{n}{2}, A) \\ otherwise : & pow(b^2, \lfloor \frac{n}{2} \rfloor, b \cdot A) \end{cases}
\end{array}
\tag{1.28}
$$

This algorithm essentially shifts $n$ to right 1 bit each time (divide $n$ by 2). If the LSB (Least Significant Bit, the lowest) is 0, $n$ is even. It squares the base and keeps the accumulator $A$ unchanged; If the LSB is 1, $n$ is odd. It 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 number $b'$, the shifted exponent number $n'$, and the accumulator $A$ satisfy the invariant $b^n = A \cdot (b')^{n'}$.

Compare to previous implementation, which minus by one for odd $n$, this algorithm 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:

```
1: function Sum(L)
2:     s ← 0
3:     while L ≠ NIL do
4:         s ← s+ First(L)
5:         L ← Rest(L)
6:     return s

7: function Product(L)
8:     p ← 1
9:     while L ≠ NIL do
10:        p ← p · First(L)
11:        L ← Rest(L)
12:    return p
```

One interesting usage of product is to calculate 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. The algorithm structure of $max/min$ is same. For a none empty list:

- If there is only one element (a singleton) $[x_1]$, the result is $x_1$;
- Otherwise, we recursively find the min/max of the sub-list, then compare it with the first element to determine the result.

$$
\begin{array}{rcl}
min([x]) & = & x \\
min(x:xs) & = & \begin{cases} x < min(xs) : & x \\ otherwise : & min(xs) \end{cases}
\end{array}
\tag{1.29}
$$

and

$$
\begin{array}{rcl}
max([x]) & = & x \\
max(x:xs) & = & \begin{cases} x > max(xs) : & x \\ otherwise : & max(xs) \end{cases}
\end{array}
\tag{1.30}
$$

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

$$
\begin{array}{rcl}
min'(a,\ \varnothing) & = & a \\
min'(a,\ x:xs) & = & \begin{cases} x < a: & min'(x, xs) \\ otherwise: & min'(a, xs) \end{cases}
\end{array}
\tag{1.31}
$$

Different from $sum'/prod'$, we can't pass a fixed starting value to the tail recursive $min'/max'$, unless we use $\pm\infty$ in below Curried form:

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

Alternatively, we can pass the first element as the accumulator given min/max only takes none empty list:

$$
min(x:xs) = min'(x,\ xs) \qquad max(x:xs) = max'(x,\ xs) \tag{1.32}
$$

The optimized tail recursive algorithm can be further changed to purely iterative implementation. We give the MIN example, and skip MAX.

```
1: function MIN(L)
2:     m ← FIRST(L)
3:     L ← REST(L)
4:     while L ≠ NIL do
5:         if FIRST(L) < m then
6:             m ← FIRST(L)
7:         L ← REST(L)
8:     return m
```

There is a way to realize the tail recursive algorithm without using accumulator explicitly. The idea is to re-use the first element as the accumulator. Every time, we compare the head with the next element; then drop the greater one for *min*, and drop the less one for *max*.

$$
\begin{array}{rcl}
min([x]) & = & x \\
min(x_1 : x_2 : xs) & = & \begin{cases} x_1 < x_2: & min(x_1 : xs) \\ otherwise: & min(x_2 : xs) \end{cases}
\end{array}
\tag{1.33}
$$

We skip the definition for *max* as it is symmetric.

### Exercise 1.9

1. Change the *length* to tail call.
2. Change the insertion sort to tail call.
3. Implement the $O(\lg n)$ algorithm to calculate $b^n$ by represent $n$ in binary.

## 1.4 Transform

From algebraic perspective, there are two types of transform: one keeps the list structure, but only change the elements; the other alter the list structure, hence the result is not isomorphic to the original list. Particularly, we call the former *map*.

### 1.4.1 map and for-each

The first example is to convert a list of numbers to their represented strings, like to change [3, 1, 2, 4, 5] to ["three", "one", "two", "four", "five"]

$$
\begin{array}{rcl}
toStr(\varnothing) & = & \varnothing \\
toStr(x : xs) & = & str(x) : toStr(xs)
\end{array}
\tag{1.34}
$$

For the second example, consider a dictionary, which is a list of words grouped by initial letter. Like:

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

Next we process a text (*Hamlet* for example), and augment each word with their 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, we want to figure out which word occurs most. How to write a program to do this work? The output is a list of words, that every one has the most occurrences in the group, something like `[a, but, can, ...]`. We need develop a program that transform **a list of groups of word-number pairs** into **a list of words**.

First, we need define a function. It takes a list of word-number pairs, finds the word paired with the biggest number. Sort is overkill. What we need is a special max function $maxBy(cmp, L)$, where $cmp$ compares two elements abstractly.

$$
\begin{array}{rcl}
maxBy(cmp,\ [x]) & = & x \\
maxBy(cmp,\ x_1 : x_2 : xs) & = & \begin{cases} cmp(x_1, x_2) : & maxBy(cmp, x_2 : xs) \\ otherwise : & maxBy(cmp, x_1 : xs) \end{cases}
\end{array}
\tag{1.35}
$$

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

$$
\begin{cases} fst\ (a, b) = & a \\ snd\ (a, b) = & b \end{cases}
\tag{1.36}
$$

Instead of embedded parenthesis $fst((a, b)) = a$, we omit one layer, and use a space. Generally, we treat $f\ x = f(x)$ when the context is clear. Then we can define a special compare function for word-count pairs:

$$
less(p_1, p_2) = snd(p_1) < snd(p_2)
\tag{1.37}
$$

Then pass $less$ to $maxBy$ to finalize our definition (in Curried form):

$$
max'' = maxBy(less)
\tag{1.38}
$$

With $max''()$ defined, we can develop the solution to process the whole list.

$$
\begin{array}{rcl}
solve(\varnothing) & = & \varnothing \\
solve(x : xs) & = & fst(max''(x)) : solve(xs)
\end{array}
\tag{1.39}
$$

### Map

The $solve()$ and $toStr()$ functions reveal the same structure, although they are developed for different problems. We can abstract this common structure as *map*:

$$
\begin{array}{rcl}
map(f,\ \varnothing) & = & \varnothing \\
map(f,\ x:xs) & = & f(x) : map(f, xs)
\end{array}
\tag{1.40}
$$

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

$$
map :: (A \to B) \to [A] \to [B] \tag{1.41}
$$

We read it as: map takes a function of $A \to B$, then convert a list $[A]$ to another list $[B]$. The two examples in previous section can be defined with map as (in Curried form):

$$
\begin{array}{l}
toStr = map\ str \\
solve = map\ (fst \circ max'')
\end{array}
$$

Where $f \circ g$ means function composite, i.e. first apply $g$ then apply $f$. $(f \circ g)\ x = f(g(x))$, Read as $f$ after $g$. Map can also be defined from the domain theory point of view. Function $y = f(x)$ defines the map from $x$ in set $X$ to $y$ in set $Y$:

$$
Y = \{f(x) | x \in X\} \tag{1.42}
$$

This type of set definition is called Zermelo-Frankel set abstraction (known as ZF expression)[72]. The different is that the mapping is from a list (but not set) to another: $Y = [f(x) | x \in Y]$. There can be duplicated elements. For list, such ZF style expression is called *list comprehension*.

List comprehension is a powerful tool. As an example, let us see how to realize the permutation algorithm. Extend from generating all-permutations as[72] and[94], we define a generic $perm(L, r)$, that permutes $r$ out of the total $n$ elements in the list $L$. There are total $P_n^r = \dfrac{n!}{(n-r)!}$ solutions.

$$
perm(L, r) = \begin{cases}
|L| < r \text{ or } r = 0 : & [[\ ]] \\
otherwise : & [x : ys \mid x \in L, ys \in perm(delete(x, L), r-1)]
\end{cases}
\tag{1.43}
$$

If pick zero element for permutation, or there are too few (less than $r$), the result is a list of empty list; otherwise, we recursively pick $r-1$ out of the rest $n-1$ elements; then prepend $x$ before each. Below Haskell example program utilizes the list comprehension feature:

```
perm xs r | r == 0 || length xs < r = [[]]
          | otherwise = [ x:ys | x <-xs,
                                 ys <- perm (delete x xs) (r-1)]
```

For the iterative Map implementation, below algorithm uses a sentinel node to simplify the logic to handle head reference.

1: **function** MAP($f, L$)
2: $\quad L' \leftarrow$ CONS($\perp$, NIL) ▷ Sentinel node
3: $\quad p \leftarrow L'$
4: $\quad$**while** $L \neq$ NIL **do**

```
5:         x ← FIRST(L)
6:         L ← REST(L)
7:         REST(p) ← CONS(f(x), NIL)
8:         p ← REST(p)
9:     return REST(L')                              ▷ Drop the sentinel
```

### For each

Sometimes we only need to traverse the list, repeatedly process the elements one by one without building the new list. Here is an example that print every element out:

```
1: function PRINT(L)
2:     while L ≠ NIL do
3:         print FIRST(L)
4:         L ← REST(L)
```

More generally, we can pass a procedure $P$, then traverse the list and apply $P$ to each element.

```
1: function FOR-EACH(P, L)
2:     while L ≠ NIL do
3:         P(FIRST(L))
4:         L ← REST(L)
```

### Examples

As an example, let's see a "$n$-lights puzzle"[96]. There are $n$ lights in a room, all of them are off. We execute the following $n$ rounds:

1. Switch all the lights in the room (all on);
2. Switch lights with number 2, 4, 6, ... , that every other light is switched, if the light is on, it will be off;
3. Switch every third lights, number 3, 6, 9, ... ;
4. ...

And at the last round, only the last light (the $n$-th light) is switched. The question is how many lights are on in the end?

Let's start with a brute-force solution, then improve it step by step. We represent the state of $n$ lights as a list of 0/1 numbers. 0 is off, 1 is on. The initial state are all zeros: [0, 0, ..., 0]. We label the light from 1 to $n$, then map them to ($i$, on/off) pairs:

$$lights = map(i \mapsto (i, 0),\ [1, 2, 3, ...n])$$

It binds each number to zero, the result is a list of pairs: $L = [(1, 0), (2, 0), ..., (n, 0)]$. Next we operate this list of pairs for $n$ rounds. In the $i$-th round, switch the second value in this pair if its label is divided by $i$. Consider $1 - 0 = 1$, and $1 - 1 = 0$, we can switch 0/1 value of $x$ by $1 - x$. For light $(j, x)$, if $i|j$, (i.e. $j \bmod i = 0$), then switch, otherwise leave the light untouched.

$$switch(i, (j, x)) = \begin{cases} j \bmod i = 0 : & (j, 1 - x) \\ otherwise : & (j, x) \end{cases} \tag{1.44}$$

The $i$-th round for all lights can be realized as map:

$$map(switch(i), L) \tag{1.45}$$

Here we use the Curried form of *switch*, which is equivalent to:

$$map((j, x) \mapsto switch(i, (j, x)), L)$$

Next, we define a function $op()$, which performs above mapping on $L$ over and over by $n$ rounds. We call this function with $op([1, 2, ..., n], L)$.

$$\begin{array}{rcl} op(\varnothing,\ L) & = & L \\ op(i:is,\ L) & = & op(is, map(switch(i), L)) \end{array} \tag{1.46}$$

At this stage, we can sum the second value of each pair in list $L$ to get the answer.

$$solve(n) = sum(map(snd, op([1, 2, ..., n], lights))) \tag{1.47}$$

Below is the example Haskell implementation of this brute-force solution:

```
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 light to 100 lights, let's see what the answers are (we 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]
```

This result is interesting:

- the first 3 answers are 1;
- the 4-th to the 8-th answers are 2;
- the 9-th to the 15-th answers are 3;
- ...

It seems that the $i^2$-th to the $((i + 1)^2 - 1)$-th answers are $i$. Actually, we can prove it:

*Proof.* Given $n$ lights labeled from 1 to $n$, consider which lights are on finally. Since the initial states for all lights are off, we can say that, the lights which are manipulated odd times are on. For every light $i$, it will be switched at the $j$ round if $i$ can be divided by $j$ (denote as $j|i$). 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 numbers which have odd number of factors. For any positive integer $n$, let $S$ be the set of all factors of $n$. $S$ is initialized to $\varnothing$. If $p$ is a factor of $n$, there must exist a positive integer $q$ such that $n = pq$ holds. 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. $\square$

At this stage, we can design a fast solution by finding the number of square numbers under $n$.

$$solve(n) = \lfloor \sqrt{n} \rfloor \tag{1.48}$$

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

```
map (floor ∘ sqrt) [1..100]
```

Map is a generic concept does not limit to list. It can be applied to many complex algebraic structures. The next chapter about binary search tree explains how to map on trees. As long as we can traverse the structure, and the empty is defined, we can use the same mapping idea.

### 1.4.2 reverse

It's a classic exercise to reverse a singly linked-list with minimum space. One must carefully manipulate the node reference, however, there exists easy method to implement reverse:

1. Write a purely recursive solution;
2. Change it to tail-call;
3. Translate the tail-call solution to imperative operations.

The purely recursive solution is straightforward. To reverse a list $L$.

- If $L$ is empty, the reversed result is empty;
- Otherwise, recursively reverse sub-list $L'$, then append the first element to the end.

$$\begin{array}{rcl} reverse(\varnothing) & = & \varnothing \\ reverse(x:xs) & = & append(reverse(xs), x) \end{array} \tag{1.49}$$

However, the performance is poor. As it need traverse to the end to append, this algorithm is bound to quadratic time. We can optimize it with tail call, use an accumulator to store the reversed part so far. We initialize the accumulator as empty: $reverse = reverse'(\varnothing)$.

$$\begin{array}{rcl} reverse'(A,\ \varnothing) & = & A \\ reverse'(A,\ x:xs) & = & reverse'(x:A,\ xs) \end{array} \tag{1.50}$$

Different from appending, cons (:) is a constant time operation. The idea is to repeatedly take the elements from the head, and prepend them to the accumulator. It essentially likes to store 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 can optimize it to purely iterative loops:

1: **function** REVERSE($L$)
2: $\quad A \leftarrow$ NIL
3: $\quad$ **while** $L \neq$ NIL **do**
4: $\quad\quad A \leftarrow$ CONS(FIRST($L$), $A$)
5: $\quad\quad L \leftarrow$ REST($L$)
6: $\quad$ **return** $A$

However, this algorithm creates a new reversed list, but not mutate the original one. We need change it to in-place mutate $L$ as the below example program:

```
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 1.10

1. Given a number from 0 to 1 billion, write a program to give its English representation. For e.g. 123 is 'one hundred and twenty three'. What if there is decimal part?
2. Implement the algorithm to find the maximum value in a list of pairs $[(k, v)]$ in tail call.

## 1.5 Sub-list

Different from array which is capable to slice a continuous segment fast, it typically need linear time to traverse and extract sub-list.

### 1.5.1 take, drop, and split-at

Taking the first $n$ elements is essentially to slice the list from 1 to $n$: $sublist(1, n, L)$. If either $n = 0$ or $L = \varnothing$, the sub-list is empty; otherwise, we recursively take the first $n-1$ elements from the $L'$, then prepend the first element.

$$
\begin{array}{rcl}
take(0, L) & = & \varnothing \\
take(n, \varnothing) & = & \varnothing \\
take(n, x : xs) & = & x : take(n-1, xs)
\end{array}
\tag{1.51}
$$

This algorithm handles the out of bound case like this: if $n > |L|$ or $n$ is negative, it ends up to the edge case that $L$ becomes empty, hence returns the whole list as the result.

Drop, on the other hand, discards the first $n$ elements and returns the rest. It is equivalent to slice the sub-list from right: $sublist(n+1, |L|, L)$, where $|L|$ is the length. Its implementation is symmetric:

$$
\begin{array}{rcl}
drop(0, L) & = & L \\
drop(n, \varnothing) & = & \varnothing \\
drop(n, x : xs) & = & drop(n-1, xs)
\end{array}
\tag{1.52}
$$

We leave the imperative implementation for *take*/*drop* as exercise. As the next step, we can develop a algorithm to extract sub-list at any position for a given length:

$$
sublist(from, cnt, L) = take(cnt, drop(from - 1, L)) \tag{1.53}
$$

Or slice the list with left and right boundaries:

$$
slice(from, to, L) = drop(from - 1, take(to, L)) \tag{1.54}
$$

The boundary is defined as $[from, to]$. It includes both ends. We can also split a list at a given position:

$$splitAt(i, L) = (take(i, L), drop(i, L)) \tag{1.55}$$

## Exercise 1.11

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

#### conditional take and drop

Instead of specifying number of elements for *take/drop*, one may want to provide a predication. We keep taking or dropping as far as the condition meets. We define such algorithm as *takeWhile/dropWhile*.

*takeWhile/dropWhile* examine elements one by one against the prediction. They ignore the rest even if some elements satisfy the condition. We'll see this different in the section of filtering.

$$\begin{array}{rcl} takeWhile(p,\ \varnothing) & = & \varnothing \\ takeWhile(p,\ x:xs) & = & \begin{cases} p(x): & x: takeWhile(p, xs) \\ otherwise: & \varnothing \end{cases} \end{array} \tag{1.56}$$

Where $p$ is the prediction. When applied to an element, $p$ returns true or false to indicate the condition is satisfied. *dropWhile* is symmetric:

$$\begin{array}{rcl} dropWhile(p,\ \varnothing) & = & \varnothing \\ dropWhile(p,\ x:xs) & = & \begin{cases} p(x): & dropWhile(p, xs) \\ otherwise: & x:xs \end{cases} \end{array} \tag{1.57}$$

### 1.5.2 break and group

Break and group are operations to re-arrange a list into multiple sub-lists. They typically perform the re-arrangement while traverse the list to keep the performance linear.

#### break and span

*break/span* can be considered as a general form of splitting. Instead of splitting at a given position, *break/span* scans elements with a prediction. It extracts the longest prefix of the list against the condition, and returns it together with the rest as a pair.

There are two different cases. For a given predication, one is to pick the elements satisfied; the other is to pick the elements not satisfied. The former is called *span*, the later is called *break*.

$$\begin{array}{rcl} span(p,\ \varnothing) & = & (\varnothing, \varnothing) \\ span(p,\ x:xs) & = & \begin{cases} p(x): & (x:A,\ B) \text{ where } (A, B) = span(p, xs) \\ otherwise: & (\varnothing,\ x:xs) \end{cases} \end{array} \tag{1.58}$$

and we can define *break* with span by negating the predication in Curried form:

$$break(p) = span(\neg p) \tag{1.59}$$

Both *span* and *break* find the longest *prefix*. They stop immediately when the condition does not meet and ignores the rest. Below is the iterative implementation for span:

```
1: function SPAN(p, L)
2:     A ← NIL
3:     while L ≠ NIL and p(FIRST(L)) do
4:         A ← CONS(FIRST(L), A)
5:         L ← REST(L)
6:     return (A, L)
```

This algorithm creates a new list to hold the longest prefix, another option is to reuse the original list and break it in-place:

```
1: function SPAN(p, L)
2:     A ← L
3:     tail ← NIL
4:     while L ≠ NIL and p(FIRST(L)) do
5:         tail ← L
6:         L ← REST(L)
7:     if tail = NIL then
8:         return (NIL, L)
9:     REST(tail) ← NIL
10:    return (A, L)
```

#### group

span breaks list into two parts, group divides list into multiple sub-lists. For example, we can use group to change a long word into small units, each contains consecutive same characters:

```
group ``Mississippi'' = [``M'', ``i'', ``ss'', ``i'',
                         ``ss'',``i'', ``pp'', ``i'']
```

For another example, given a list of numbers:

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

We can divide it into small lists, each one is in descending order:

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

These are useful operations. The string groups can be used to build Radix tree, a data structure support fast text search. The number groups can be used to implement nature merge sort algorithm. We'll introduce them in later chapters.

We can abstract the group condition as a relation $\sim$. It tests whether two consecutive elements $x$, $y$ are generic 'equivalent': $x \sim y$. We scan and list and compare two elements each time. If they match, we add both to a group; otherwise, only add $x$ to the group, and use $y$ to initialize another group.

$$\begin{array}{rcl}
group(\sim,\ \varnothing) & = & [\varnothing] \\
group(\sim,\ [x]) & = & [[x]] \\
group(\sim,\ x : y : xs) & = & \begin{cases} x \sim y : & (x : ys) : yss \\ otherwise : & [x] : ys : yss \end{cases}
\end{array} \tag{1.60}$$

where $(ys : yss) = group(\sim, xs)$. This algorithm is bound to $O(n)$ time, where $n$ is the length. We can also implement the iterative group algorithm. For the none empty list $L$, we initialize the result groups as $[[x_1]]$, where $x_1$ is the first element. We scan the list from the second one, append it to the last group if the two consecutive elements are 'equivalent'; otherwise we start a new group.

1: **function** GROUP($\sim, L$)
2: **if** $L =$ NIL **then**
3: **return** [NIL]
4: $x \leftarrow$ FIRST($L$)
5: $L \leftarrow$ REST($L$)
6: $g \leftarrow [x]$
7: $G \leftarrow [g]$
8: **while** $L \neq$ NIL **do**
9: $y \leftarrow$ FIRST($L$)
10: **if** $x \sim y$ **then**
11: $g \leftarrow$ APPEND($g, y$)
12: **else**
13: $g \leftarrow [y]$
14: $G \leftarrow$ APPEND($G, g$)
15: $x \leftarrow y$
16: $L \leftarrow$ NEXT($L$)
17: **return** $G$

However, this program performs in quadratic time if the append isn't optimized with the tail reference. If don't care the order, we can alternatively change append to cons. With the group algorithm defined, we can realize the above 2 cases as below:

$$group(=, [m, i, s, s, i, s, s, i, p, p, i]) = [[M], [i], [ss], [i], [ss], [i], [pp], [i]]$$

and

$$\begin{array}{l} group(\geq, [15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2]) \\ = [[15, 9, 0], [12, 11, 7], [10, 5], [6], [13, 1], [4], [8, 3], [14, 2]] \end{array}$$

Another method to implement group is to use the *span* function. Given a predication, span breaks the list into two parts: the longest sub-list satisfies the condition, and the rest. We can repeatedly apply span to the rest part till it becomes empty. However, the predication passed to span is an unary function. It takes an element and tests it. While in group, the predication is a binary function. It takes two elements and compares. We can use Currying: to pass and fix the first element in the binary predication, then use the Curried function to test the other.

$$\begin{array}{rcl} group(\sim, \ \varnothing) & = & [\varnothing] \\ group(\sim, \ x : xs) & = & (x : A) : group(\sim, B) \end{array} \tag{1.61}$$

Where $(A, B) = span(y \mapsto x \sim y, xs)$ is the span result applied to the rest sub-list. Although this new group function generates the correct result for string case:

```
group (==) ``Mississippi''
[``m'', ``i'', ``ss'', ``i'', ``ss'', ``i'', ``pp'', ``i'']
```

However, it can't group the list of numbers correctly with $\leq$ relation:

```
group (≥) [15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2]
[[15,9,0,12,11,7,10,5,6,13,1,4,8,3,14,2]]
```

When the first number 15 is used as the left hand of $\geq$, it is the maximum value, hence *span* ends with putting all elements to $A$, and leaves $B$ empty. It is not a defect, but the correct behavior, because group is defined to put equivalent elements together. To be accurate, the equivalent relation ($\sim$) needs satisfy three things: reflexive, transitive, and symmetric.

1. **Reflexive**. $x \sim x$, any element equals to itself;

2. **Transitive**. $x \sim y, y \sim z \Rightarrow x \sim z$, if two elements equal, and one of them equals to another, then all three equal;

3. **Symmetric**. $x \sim y \Leftrightarrow y \sim x$, the order of comparing two equal elements doesn't affect the result.

When group "Mississippi", we use the equal ($=$) operator. It conforms the three rules, and generates the correct result. However, when pass Curried ($\geq$) predication for numbers, it violets both reflexive and symmetric rules, hence generates unexpected result. The second algorithm using span, limits its use case to strictly equality; while the first algorithm does not. It only tests the predication for every two elements matches, which is weaker than equality relation.

**Exercise 1.12**

1. Change the *take/drop* algorithm, such that when $n$ is negative, returns $\varnothing$ for take, and the whole list for drop.
2. Implement the in-place imperative *take/drop* algorithms.
3. Implement the iterative 'take while' and 'drop while' algorithms.
4. Consider the below *span* implementation:

$$
\begin{array}{rcl}
span(p,\ \varnothing) & = & (\varnothing, \varnothing) \\
span(p,\ x:xs) & = & \begin{cases} p(x): & (x:A, B) \\ otherwise: & (A, x:B) \end{cases}
\end{array}
$$

   where $(A, B) = span(p, xs)$. What is the difference between this one and the algorithm we defined previously?

## 1.6 Fold

We've seen most list algorithms share some common structure. This is not by chance. Such commonality is rooted from the recursive nature of list. We can abstract the list algorithms to a higher level concept, fold[5], which is essentially the initial algebra of all list related computation[99].

### 1.6.1 fold right

Compare *sum*, *product* and *sort*, we can find the common structure.

$$
\begin{array}{rcl}
h(\varnothing) & = & z \\
h(x:xs) & = & x \oplus h(xs)
\end{array}
\tag{1.62}
$$

There are two things we can abstract as parameters:

- The result for empty list. It is 0 for sum, 1 for product, and $\varnothing$ for sort.
- The binary operation applies to the head and the recursive result. It is plus for sum, multiply for product, and ordered-insertion for sort.

[5] also known as reduce

We abstract the result for empty list as the *initial value*, denoted as $z$ to mimic the generic zero concept. The binary operation as $\oplus$. The above definition can be then parameterized as:

$$\begin{array}{rcl} h(\oplus, z, \ \varnothing) & = & z \\ h(\oplus, z, \ x : xs) & = & x \oplus h(\oplus, z, xs) \end{array} \tag{1.63}$$

Let's feed it a list $L = [x_1, x_2, ..., x_n]$, and expand to see how it behaves like:

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

We need add the parentheses, because the computation starts from the right-most $(x_n \oplus z)$. It repeatedly folds to 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

We can consider the fold-fan as a list of bamboo frames. The binary operation is to fold a frame to the top of the stack. The initial stack is empty. To fold the fan, we start from one end, repeatedly apply the binary operation, till all the frames are stacked. The sum and product algorithms do the same thing like folding fan.

$$\begin{array}{rl} sum([1, 2, 3, 4, 5]) & = 1 + (2 + (3 + (4 + 5))) \\ & = 1 + (2 + (3 + 9)) \\ & = 1 + (2 + 12) \\ & = 1 + 14 \\ & = 15 \end{array}$$

$$\begin{array}{rl} product([1, 2, 3, 4, 5]) & = 1 \times (2 \times (3 \times (4 \times 5))) \\ & = 1 \times (2 \times (3 \times 20)) \\ & = 1 \times (2 \times 60) \\ & = 1 \times 120 \\ & = 120 \end{array}$$

We name this kind of process fold. Particularly, since the computation starts from the right end, we denote it $foldr$:

$$\begin{array}{rcl} foldr(f, z, \ \varnothing) & = & z \\ foldr(f, z, \ x : xs) & = & f(x, foldr(f, z, xs)) \end{array} \tag{1.64}$$

We can define sum and product with $foldr$ as below:

$$\begin{array}{rl} \sum_{i=1}^{n} x_i & = x_1 + (x_2 + (x_3 + ... + (x_{n-1} + x_n))...) \\ & = foldr(+, 0, [x_1, x_2, ..., x_n]) \end{array} \tag{1.65}$$

$$\begin{array}{rl} \prod_{i=1}^{n} x_i & = x_1 \times (x_2 \times (x_3 \times ... + (x_{n-1} \times x_n))...) \\ & = foldr(\times, 1, [x_1, x_2, ..., x_n]) \end{array} \tag{1.66}$$

Or in Curried form: $sum = foldr(+, 0)$, $product = foldr(\times, 1)$. We can also define the insertion sort with $foldr$ as:

$$sort = foldr(insert, \varnothing) \tag{1.67}$$

### 1.6.2 fold left

We can convert $foldr$ to tail call. It generates the same result, but computes from left to right. For this reason, we define it as $foldl$:

$$\begin{array}{rcl} foldl(f, z, \ \varnothing) & = & z \\ foldl(f, z, \ x : xs) & = & foldl(f, f(z, x), xs) \end{array} \tag{1.68}$$

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

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

Here we delay the evaluation of $f(z, x)$ in every step. This is the behavior for lazy-evaluation. Otherwise, they will be evaluated in sequence of $[1, 3, 6, 10, 15]$ in each call. Generally, we can expand $foldl$ as:

$$foldl(f, z, [x_1, x_2, ..., x_n]) = f(f(...(f(f(z, x_1), x_2), ..., x_n) \tag{1.69}$$

Or express as infix:

$$foldl(\oplus, z, [x_1, x_2, ..., x_n]) = z \oplus x_1 \oplus x_2 \oplus ... \oplus x_n \tag{1.70}$$

$foldl$ is tail recursive. We can implement it with loops. We initialize the result as $z$, then apply the binary operation on top of it with every element. It is typically called REDUCE in most imperative environment.

```
1: function REDUCE(f, z, L)
2:     while L ≠ NIL do
3:         z ← f(z, FIRST(L) )
4:         L ← REST(L)
5:     return z
```

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

$$fromList = foldr(add, empty)$$

Where *empty* is the empty container. The singly linked-list is such a container. It performs well when add element to the head, but poorly when append to tail. $foldr$ is a natural choice when duplicate a list while keep the order. But $foldl$ will generate a reversed list. As a workaround, to implement the iterative reducing from right, we can first reverse the list, then reduce it:

1: **function** REDUCE-RIGHT($f, z, L$)
2: **return** REDUCE($f, z$, REVERSE($L$))

One may think $foldl$ should be the preferred one as it is optimized with tail call, hence fits for both functional and imperative settings. It is also the online algorithm that always holds the result so far. 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:

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

It does not work with $foldl$ because the outer most evaluation never ends. We use a unified notation $fold$ when either left or right works. In this book, we also use $fold_l$ and $fold_r$ to emphasis folding over the direction. Although this chapter is about list, the fold concept is generic. It can be applied to other algebraic structures. We can fold a tree (2.6 in[99]), a queue, and many other things as long as they satisfy the following 2 criteria:

- The empty is defined (like the empty tree);
- We can decompose the recursive structure (like decompose tree into sub-trees and key).

People abstract them further with concepts like foldable, monoid, and traversable.

### Exercise 1.13

1. To define insertion-sort with $foldr$, we designe the insert function as $insert(x, L)$, such that it can be expressed as $sort = foldr(insert, \varnothing)$. 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 $A \to B \to B$, the initial value $z$ has the type $B$. It folds on a list of $A$, and builds the result of $B$. How to define the insertion-sort with $foldl$? What is the type signature of $foldl$?

## 1.6.3 example

As an example, let's see how to 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 number $i$, and on/off state $s$. Every round $j$, we scan the lights, toggle the $i$-th switch when the $j$ divides the $i$. We can define this process with $fold$:

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

As the initial state, all lights are off. We fold on the list of round numbers from 1 to $n$. Function *step* takes two parameters: the round number $i$, and the list of pairs. It performs switching through *map*:

$$fold_r((i, L) \mapsto map(switch(i), L), [(1, 0), (2, 0), ..., (n, 0)], [1, 2, ..., n])$$

The $fold_r$ result is the pairs of final on/off state, we next extract the state from each through $map$, and count the number with $sum$:

$$
\begin{array}{rl}
sum(map(snd, & fold_r((i, L) \mapsto \\
& map(switch(i), L), [(1, 0), (2, 0), ..., (n, 0)], [1, 2, ..., n])))
\end{array}
\tag{1.71}
$$

**concatenate**

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

$$
concat = fold_r(+\!\!+, \varnothing) \tag{1.72}
$$

This is in Curried form. Its usage is as:

$$
concat([[1], [2, 3, 4], [5, 6, 7, 8, 9]]) \Rightarrow [1, 2, 3, 4, 5, 6, 7, 8, 9]
$$

### Exercise 1.14

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

## 1.7 Search and filter

Search and filter are generic concepts apply to a wide range of things. For list, it often takes linear time to find the result, as we need traverse in most cases.

### 1.7.1 Exist

Given some $a$ of type $A$, and a list of $A$, how to test if $x$ is in the list? The idea is to compare every element in the list with $a$, until either they are equal, or reach to the end:

- If the list is empty, then $a$ does not exist;
- If the first element equals to $a$, then it exists;
- Otherwise, recursively test if $a$ exists in the rest sub-list.

$$
\begin{array}{rcl}
a \in \varnothing & = & False \\
a \in (b : bs) & = & \begin{cases} b = a : & True \\ b \neq a : & a \in bs \end{cases}
\end{array}
\tag{1.73}
$$

This algorithm is also called $elem$. It bounds to $O(n)$ where $n$ is the length. If the list is ordered (ascending for example), one may want to improve the algorithm to logarithm time with the idea of divide and conquer. However, list does not support random access, we can't apply binary search. See chapter 3 for details.

### 1.7.2 Look up

Let's extend *elem* a bit. In the $n$-lights puzzle, we use a list of pairs $[(k, v)]$. Every pair contains a key and a value. This kind of list is called 'associate list' (or assoc list). If want to look up a given value in such list, we need extract some part (the value) for comparison.

$$
\begin{array}{rcl}
lookup(x, \varnothing) & = & Nothing \\
lookup(x, (k, v) : kvs) & = & \begin{cases} v = x : & Just\ (k, v) \\ v \neq x : & lookup(x, kvs) \end{cases}
\end{array} \tag{1.74}
$$

Different from *elem*, we do not return true/false. Instead, we want to return the pair of key-value when find. However, it is not guaranteed the value always exists. We use an algebraic type called 'Maybe'. A type of **Maybe** $A$ has two different kinds of value. It maybe some $a$ in $A$ of nothing. Denoted as $Just\ a$ or $Nothing$. This is the way to deal with null reference issues(4.2.2 in[99]).

### 1.7.3 find and filter

We can make 'look up' more generic. Instead of only comparing if the element equals to the given value, we can abstract to find the element that satisfies a specific predicate:

$$
\begin{array}{rcl}
find(p,\ \varnothing) & = & Nothing \\
find(p,\ (x : xs)) & = & \begin{cases} p(x) : & Just\ x \\ otherwise : & find(p, xs) \end{cases}
\end{array} \tag{1.75}
$$

Although there can be multiple elements match, the *find* algorithm picks the first. We can expand it to find all elements. It is often called *filter* as demonstrated in figure 1.4.

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

We can define it in ZF expression:

$$
filter(p, X) = [x_i | x_i \in X, p(x_i)] \tag{1.76}
$$

Different from *find*, when there is no element satisfies the predicate, *filter* returns the empty list. It scans to examine every element one by one:

$$
\begin{array}{rcl}
filter(p,\ \varnothing) & = & \varnothing \\
filter(p,\ x : xs) & = & \begin{cases} p(x) : & x : filter(p, xs) \\ otherwise : & filter(p, xs) \end{cases}
\end{array} \tag{1.77}
$$

This definition builds the result from right to left. For iterative implementation, if build the result with *append*, it will degrade to $O(n^2)$.

```
1: function FILTER(p, L)
2:     L' ← NIL
3:     while L ≠ NIL do
4:         if p(FIRST(L)) then
5:             L' ← APPEND(L', FIRST(L))                    ▷ Linear time
```

6: $L \leftarrow \text{REST}(L)$

The right way is to use *cons* instead, however, it builds the result in the reversed order. We can further reverse it within linear time (see the exercise). The nature to build result from right indicates that we can define filter in $foldr$. We need define a function $f$ to test an element against the predicate, if OK, prepend to the result:

$$f(x, A) = \begin{cases} p(x) : & x : A \\ otherwise : & A \end{cases} \tag{1.78}$$

We also need pass the predicate $p$ to $f$. There are actually 3 parameters as $f(p, x, A)$. Filter is defined in $foldr$ with a Curried form of $f$:

$$filter(p) = foldr((x, A) \mapsto f(p, x, A), \varnothing) \tag{1.79}$$

We can further simplify it (called $\eta$-conversion[73]) as:

$$filter(p) = foldr(f(p), \varnothing) \tag{1.80}$$

Filter is also a generic concept not only limit to list. We can apply a predicate on any traversable structures to extract the result.

### 1.7.4 Match

Match is to find a pattern among some structure. Even if we limit to list and string, there are still too many things to cover. We have dedicated chapters about string matching. This section deals with the problem, that given a list $A$, and test if it exits in another list $B$. There are two special cases: to test if $A$ is prefix or suffix of $B$. The *span* algorithm in (1.58) actually finds a prefix under a certain condition. We can do similar things: to compare each element between $A$ and $B$ from left till meet any different one or reach the end of either list. Define $A \subseteq B$ if $A$ is prefix of $B$:

$$\begin{array}{rcl} \varnothing \subseteq B & = & True \\ (a : as) \subseteq \varnothing & = & False \\ (a : as) \subseteq (b : bs) & = & \begin{cases} a \neq b : & False \\ a = b : & as \subseteq bs \end{cases} \end{array} \tag{1.81}$$

Prefix testing takes linear time as it scans the lists. However, we can not do suffix testing in this way because it is hard to start from the aligned right ends, and scan backwards for lists. This is different from array. Alternatively, we can reverse both lists in linear time, hence change the problem to prefix testing:

$$A \supseteq B = reverse(A) \subseteq reverse(B) \tag{1.82}$$

With $\subseteq$ defined, we can test if a list is the sub-list of another one. We call it infix testing. The idea is to scan the target list, and repeatedly applying the prefix testing:

$$\begin{array}{rcl} infix?(a : as,\ \varnothing) & = & False \\ infix?(A,\ B) & = & \begin{cases} A \subseteq B : & True \\ otherwise : & infix?(A, B') \end{cases} \end{array} \tag{1.83}$$

For the edge case that $A$ is empty, we define empty is infix of any list. Because $\varnothing \subseteq B$ is always true, it gives the right result. It also evaluates $infix?(\varnothing, \varnothing)$ correctly. Below is the corresponding iterative implementation:

1: **function** IS-INFIX($A, B$)
2: **if** $A =$ NIL **then**
3: **return** TRUE
4: $n \leftarrow |A|$
5: **while** $B \neq$ NIL and $n \leq |B|$ **do**
6: **if** $A \subseteq B$ **then**
7: **return** TRUE
8: $B \leftarrow$ REST($B$)
9: **return** FALSE

Because prefix testing runs in linear time, and it is called in the loop of scan. This algorithm is bound to $O(nm)$, where $m, n$ are the length of the two lists respectively. It is an interesting problem to improve this 'position by position' scan algorithm to linear time, even when we apply it to arrays. Chapter 13 introduces some smart methods, like the Knuth-Morris-Pratt (KMP) algorithm and Boyer-Moore algorithm. Appendix C introduces another method called suffix-tree.

In a symmetric way, we can enumerate all suffixes of $B$, and check if $A$ is prefix of any of them:

$$infix?(A, B) = \exists S \in \mathit{suffixes}(B), A \subseteq S \tag{1.84}$$

This can be implemented with list comprehension as below example Haskell program:

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

Where function `isPrefixOf` does the prefixing testing, `tails` generates all suffixes of a given list. We left its implementation as an exercise.

### Exercise 1.15

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

## 1.8 zip and unzip

The assoc list of paired values is often used as a light weighted dictionary for small set of data. It is easier to build assoc list than tree or heap based dictionary, although the look up performance of assoc list is linear instead of logarithm. In the '$n$-lights' puzzle, we build the assoc list as below:

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

More often, we need 'zip' two lists to one. We can define a *zip* function to do that:

$$\begin{array}{rcl} zip(A,\ \varnothing) & = & \varnothing \\ zip(\varnothing,\ B) & = & \varnothing \\ zip(a : as,\ b : bs) & = & (a, b) : zip(as, bs) \end{array} \tag{1.85}$$

This algorithm works even the two lists have different length. The result length equals to the shorter one. We can even use it to zip infinite lists (under lazy evaluation if both are infinite), for example[6]:

[6] In Haskell: `zip (repeat 0) [1..n]`

$$zip([0, 0, ...], [1, 2, ..., n])$$

For a list of words, we can index them with numbers as:

$$zip([1, 2, ...], [a, an, another, ...])$$

*zip* build the result from right. We can also 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*, unless with the reference to the tail position.

1: **function** ZIP($A, B$)
2: $\quad C \leftarrow$ NIL
3: $\quad$ **while** $A \neq$ NIL and $B \neq$ NIL **do**
4: $\qquad C \leftarrow$ APPEND(C, (FIRST($A$), FIRST($B$))) $\quad \triangleright$ Linear time
5: $\qquad A \leftarrow$ REST($A$)
6: $\qquad B \leftarrow$ REST($B$)
7: $\quad$ **return** $C$

To avoid *append*, we can use 'cons' then reverse the result. However, it can not deal with two infinite lists. In imperative settings, we can also re-use $A$ to store the result (treat it as transform a list of elements to a list of pairs).

We can extend to *zip* multiple lists to one. Some programming libraries provide, `zip`, `zip3`, `zip4`, ..., till `zip7`. Sometimes, we don't want to build a list of pairs, but apply a combinator function. For example, given a list of unit prices $[1.00, 0.80, 10.05, ...]$ for fruits: apple, orange, banana, ... When customer has a list of quantities, like $[3, 1, 0, ...]$, means this customer, buys 3 apples, 1 orange, 0 banana, ... Below program generates a payment list:

$$\begin{array}{rcl}
pays(U,\ \varnothing) & = & \varnothing \\
pays(\varnothing,\ Q) & = & \varnothing \\
pays(u : us,\ q : qs) & = & (u \cdot q) : pays(us, qs)
\end{array}$$

It is same as the *zip* function except uses multiply but not 'cons' to combine elements. We can abstract the combinator as a function $f$, and pass it to *zip* to build a generic algorithm:

$$\begin{array}{rcl}
zipWith(f, A,\ \varnothing) & = & \varnothing \\
zipWith(f, \varnothing,\ B) & = & \varnothing \\
zipWith(f, a : as,\ b : bs) & = & f(a, b) : zipWith(f, as, bs)
\end{array} \tag{1.86}$$

Here is an example that defines the inner-product (or dot-product) [98] through $zipWith$:

$$A \cdot B = sum(zipWith(\cdot, A, B)) \tag{1.87}$$

*unzip* is the inverse operation of *zip*. It converts a list of pairs to two separated lists. Below is its definition with $foldr$ in Curried form:

$$unzip = foldr((a, b), (A, B) \mapsto (a : A, b : B), (\varnothing, \varnothing)) \tag{1.88}$$

We fold from a pair of empty lists, break $a, b$ from the pairs and prepend them to the two intermediate lists respectively. We can also use $fst$ and $snd$ explicitly as:

$$(p, P) \mapsto (fst(p) : fst(P), snd(p) : snd(P))$$

For the fruits example, suppose the unit price is stored in a assoc list: $U = [(apple, 1.00), (orange, 0.8$ for lookup, for example $lookup(melon, U)$. The purchase quantity is a assoc list: $Q =$

$[(apple, 3), (orange, 1), (banana, 0), ...]$. How to calculate the total payment? The straight forward way is to extract the unit price and the quantity lists, then compute their inner-product:

$$pay = sum(zipWith(\cdot, snd(unzip(U)), snd(unzip(Q)))) \tag{1.89}$$

As an example, let's see how to use $zipWith$ to define infinite Fibonacci numbers with lazy evaluation:

$$F = 0 : 1 : zipWith(+, F, F') \tag{1.90}$$

Where $F$ is the infinite list of Fibonacci numbers, starts from 0 and 1. $F'$ is the rest Fibonacci numbers without the first one. From the third, every Fibonacci number is the sum of numbers from $F$ and $F'$ at the same position. Below example program list 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]
```

$zip$ 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]).

## Exercise 1.16

1. Design the iota ($I$) algorithm for below usages:

   - $iota(..., n) = [1, 2, 3, ..., n]$;
   - $iota(m, n) = [m, m + 1, m + 2, ..., n]$, where $m \leq n$;
   - $iota(m, m + a, ..., n) = [m, m + a, m + 2a, ..., n]$;
   - $iota(m, m, ...) = repeat(m) = [m, m, m, ...]$;
   - $iota(m, ...) = [m, m + 1, m + 2, ...]$.

   The last two cases are about infinite list. One possible implementation is through streaming and lazy evaluation ([63] and[10]).
2. Implement the linear time imperative $zip$ algorithm
3. Define $zip$ with $foldr$.
4. For the fruits example, suppose the quantity assoc list only contains the items with none-zero quantity. i.e. instead of

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

   but

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

   because customer does not buy banana. Design a program to calculate the total payment.
5. Implement $lastAt$ with $zip$.

## 1.9 Further reading

List is the fundamental thing 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 atop 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 in[72]. Haskell library provides detailed documentation about basic list algorithms. There are materials provide good examples of folding, especially in[1]. It also introduces about the *fold fusion law*.

### Exercise 1.17

1. Design algorithm to remove the duplicated elements in a list. For imperative implementation, the elements should be removed in-place. The original element order should be maintained. What is the complexity of this algorithm? How to simplify it with additional data structure?
2. 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.
3. 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).

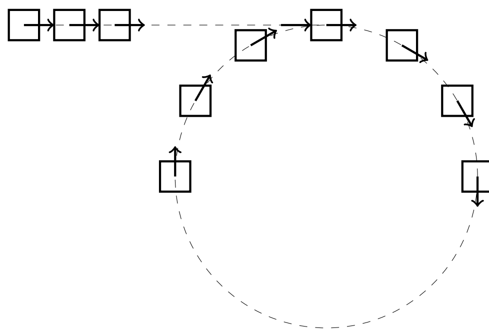


Figure 1.5: A circular linked-list

# 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

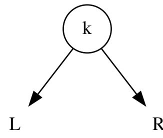
- either empty;
- or contains 3 parts: the element, and two sub-trees called left and right children.

Figure 2.1 shows an example of binary tree.

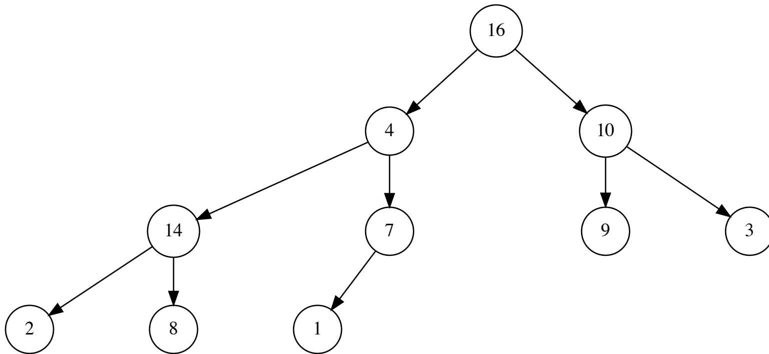
A binary search tree is a special binary tree that its elements are comparable[1], and satisfies the following constraints:

- 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*.

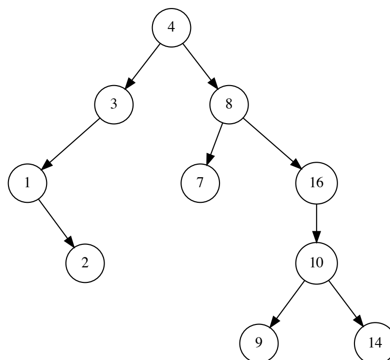



Figure 2.2: A Binary Search Tree

## 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.

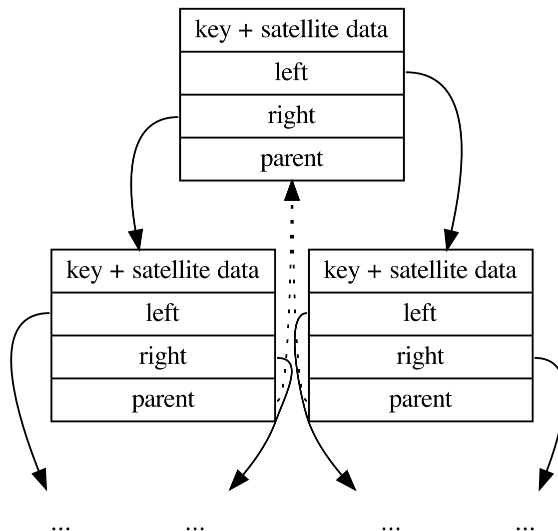



Figure 2.3: Node layout with parent field.

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:

1: **function** INSERT($T, k$)
2: $\quad root \leftarrow T$
3: $\quad x \leftarrow$ CREATE-LEAF($k$)
4: $\quad parent \leftarrow$ NIL
5: $\quad$ **while** $T \neq$ NIL **do**
6: $\quad\quad parent \leftarrow T$
7: $\quad\quad$ **if** $k <$ KEY($T$) **then**
8: $\quad\quad\quad T \leftarrow$ LEFT($T$)
9: $\quad\quad$ **else**
10: $\quad\quad\quad T \leftarrow$ RIGHT($T$)

```
11:     PARENT(x) ← parent
12:     if parent = NIL then                                   ▷ tree T is empty
13:         return x
14:     else if k < KEY(parent) then
15:         LEFT(parent) ← x
16:     else
17:         RIGHT(parent) ← x
18:     return root

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

While 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.

1: **function** IN-ORDER-TRAVERSE($T, f$)
2:     **if** $T \neq$ NIL **then**
3:         IN-ORDER-TRAVERSE(LEFT($T, f$))
4:         $f$(KEY($T$))
5:         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:

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

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:

1: **function** SEARCH($T, x$)
2:     **while** $T \neq$ NIL and KEY($T$) $\neq x$ **do**
3:         **if** $x <$ KEY($T$) **then**
4:             $T \leftarrow$ LEFT($T$)
5:         **else**
6:             $T \leftarrow$ RIGHT($T$)
7:     **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 $x$:

1: **function** SUCC($x$)
2: **if** RIGHT($x$) $\neq$ NIL **then**
3: **return** MIN(RIGHT($x$))
4: **else**
5: $p \leftarrow$ PARENT($x$)
6: **while** $p \neq$ NIL and $x =$ RIGHT($p$) **do**
7: $x \leftarrow p$
8: $p \leftarrow$ PARENT($p$)
9: **return** $p$

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

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

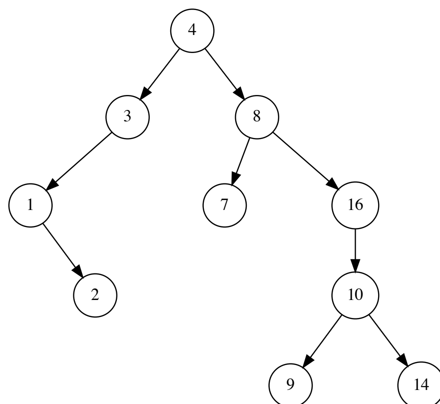



Figure 2.4: The successor of 8, is the minimum one in its right sub-tree, 9; In order to find the successor of 2, we go up to its parent 1, then 3.

3: **return** MAX(LEFT($x$))
4: **else**
5: $p \leftarrow$ PARENT($x$)
6: **while** $p \neq$ NIL and $x =$ LEFT($p$) **do**
7: $x \leftarrow p$
8: $p \leftarrow$ PARENT($p$)
9: **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.

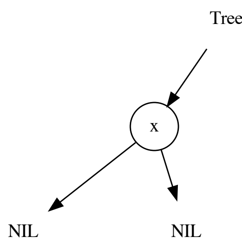
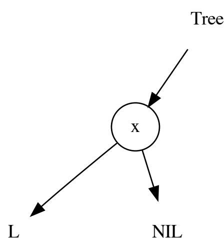
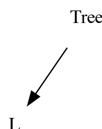


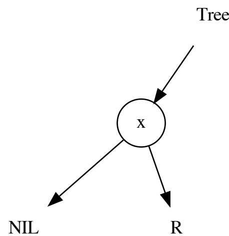

Figure 2.5: $x$ can be spliced out.



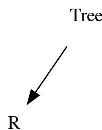

(a) Before delete $x$.




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

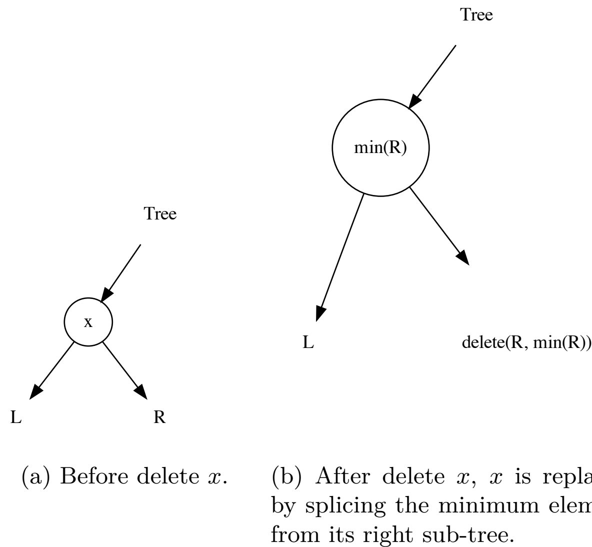



(c) Before delete $x$.




(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.

1: **function** DELETE($T, x$)
2: $\quad r \leftarrow T$
3: $\quad x' \leftarrow x$ $\triangleright$ save $x$

```
4:      p ← Parent(x)
5:      if Left(x) = NIL then
6:          x ← Right(x)
7:      else if Right(x) = NIL then
8:          x ← Left(x)
9:      else                                          ▷ neither children is empty
10:         y ← Min(Right(x))
11:         Key(x) ← Key(y)
12:         Copy other satellite data from y to x
13:         if Parent(y) ≠ x then                     ▷ y does not have left sub-tree
14:             Left(Parent(y)) ← Right(y)
15:         else                                      ▷ y is the root of the right sub-tree
16:             Right(x) ← Right(y)
17:         if Right(y) ≠ NIL then
18:             Parent(Right(y)) ← Parent(y)
19:         Remove y
20:         return r
21:     if x ≠ NIL then
22:         Parent(x) ← p
23:     if p = NIL then                               ▷ remove the root
24:         r ← x
25:     else
26:         if Left(p) = x' then
27:             Left(p) ← x
28:         else
29:             Right(p) ← x
30:     Remove x'
31:     return r
```

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(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(t);
}
```

Example program to find the successor of a node.

```
Optional<Node<T>> succ (Node<T> x) {
    if (x == null) {
        return Optional.None
    } 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(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:

1: **function** SORT($A$)
2: $S \leftarrow$ NIL
3: **for** each $a \in A$ **do**
4: INSERT($a, S$)
5: **return** $S$

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

1: **function** SORT($A$)

[1] We skip the 'Bubble sort' method

2: **for** $i \leftarrow 2$ to $|A|$ **do**
3: 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.

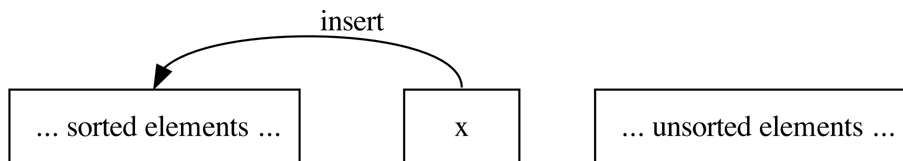



Figure 3.2: Continuously insert elements to the sorted part.

## 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:

1: **function** Sort($A$)
2: **for** $i \leftarrow 2$ to $|A|$ **do** ▷ Insert $A[i]$ to $A[1...(i-1)]$
3: $x \leftarrow A[i]$ ▷ Save $A[i]$ to $x$
4: $j \leftarrow i - 1$
5: **while** $j > 0$ and $x < A[j]$ **do**
6: $A[j+1] \leftarrow A[j]$
7: $j \leftarrow j - 1$
8: $A[j+1] \leftarrow x$

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.

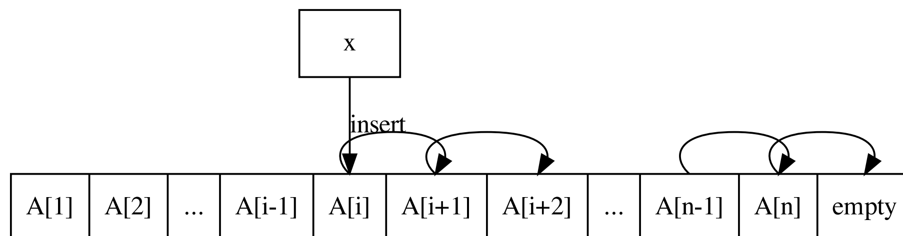



Figure 3.3: Insert $x$ to $A$ at $i$.

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.

```
1: function SORT(A)
2:     for i ← 2 to |A| do
3:         x ← A[i]
4:         p ← BINARY-SEARCH(x, A[1...(i − 1)])
5:         for j ← i down to p do
6:             A[j] ← A[j − 1]
7:         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.

```
1: function BINARY-SEARCH(x, A)
2:     l ← 1, u ← 1 + |A|
3:     while l < u do
4:         m ← ⌊(l + u)/2⌋
5:         if A[m] = x then
6:             return m                      ▷ Duplicated element
7:         else if A[m] < x then
8:             l ← m + 1
9:         else
10:            u ← m
11:    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:

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

7: function Sort(A)
8:     n ← |A|
9:     Next = [1, 2, ..., n, −1]                      ▷ n + 1 indexes
10:    for i ← 1 to n do
11:        Insert(A, Next, i)
12:    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.

1: **function** SORT($A$)
2: $T \leftarrow \varnothing$
3: **for** each $x \in A$ **do**
4: $T \leftarrow$ INSERT-TREE($T, x$)
5: **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.

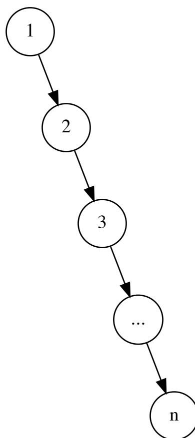



Figure 4.1: unbalanced tree

### 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:

1: **function** Left-Rotate($T, x$)
2: $p \leftarrow$ Parent($x$)
3: $y \leftarrow$ Right($x$) ▷ assume $y \neq$ NIL
4: $a \leftarrow$ Left($x$)
5: $b \leftarrow$ Left($y$)
6: $c \leftarrow$ Right($y$)
7: Replace($x, y$) ▷ replace node $x$ with $y$

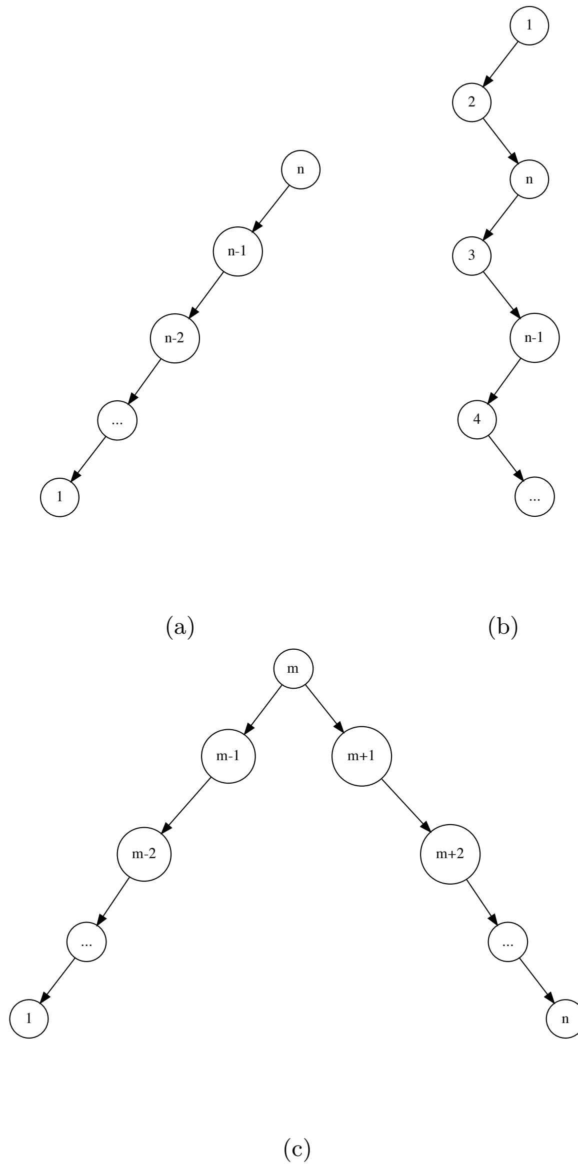



Figure 4.2: Unbalanced trees

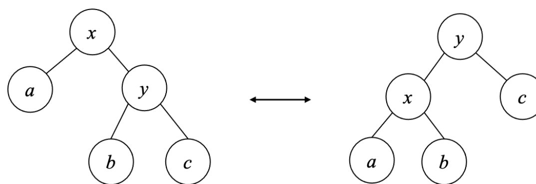



Figure 4.3: 'left rotate' and 'right rotate'.

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

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

```
1: function Replace(x, y)
2:     p ← Parent(x)
3:     if p = NIL then                       ▷ x is the root
4:         if y ≠ NIL then Parent(y) ← NIL
5:     else if Left(p) = x then
6:         Set-Left(p, y)
7:     else
8:         Set-Right(p, y)
9:     Parent(x) ← NIL
```

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

```
1: function Set-Subtrees(x, L, R)
2:     Set-Left(x, L)
3:     Set-Right(x, R)
```

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

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

4: function Set-Right(x, y)
5:     Right(x) ← y
6:     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.

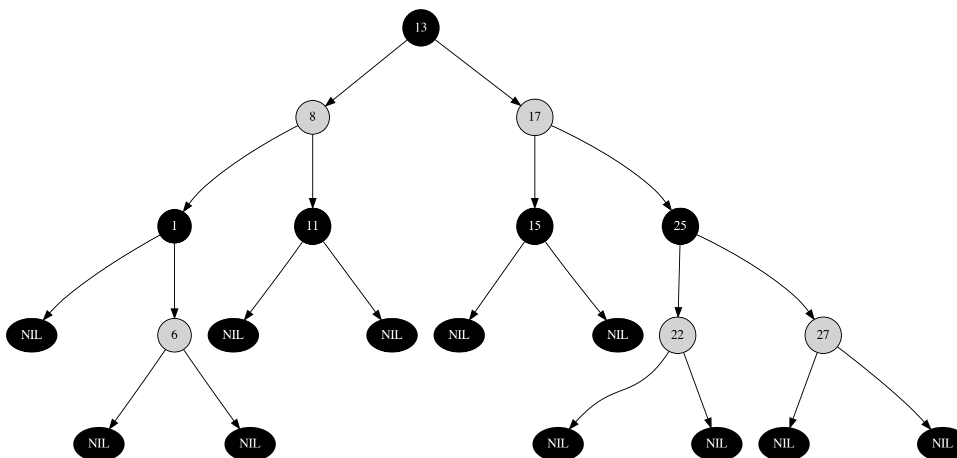



Figure 4.4: A red-black tree

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.

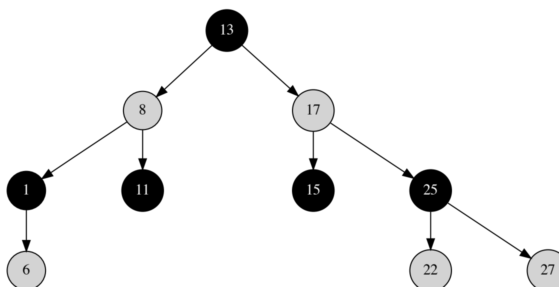



Figure 4.5: Hide the NIL nodes

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.

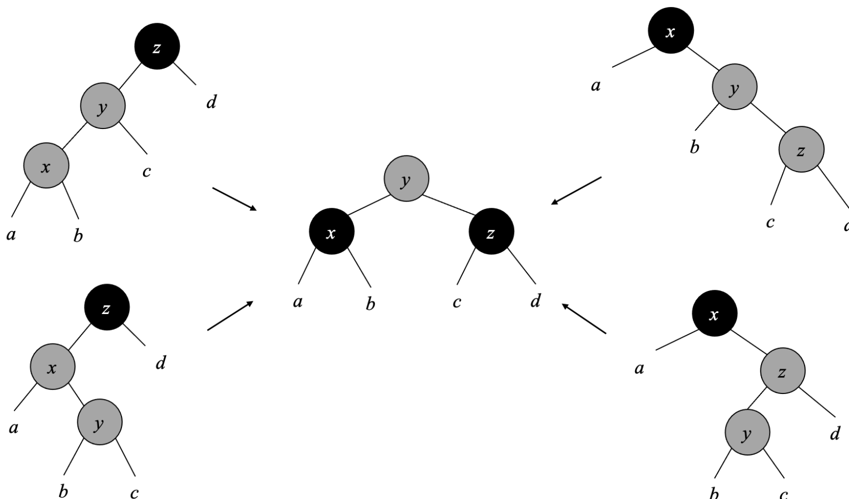



Figure 4.6: Fix 4 cases to the same structure.

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.

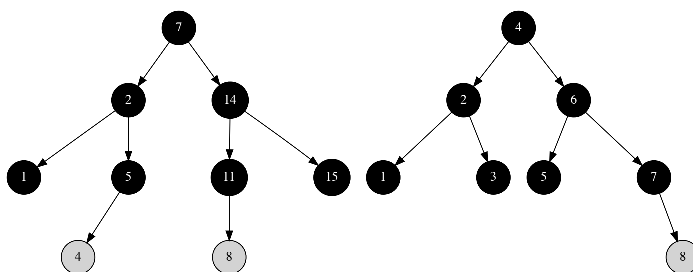



Figure 4.7: Red-black tree examples

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.

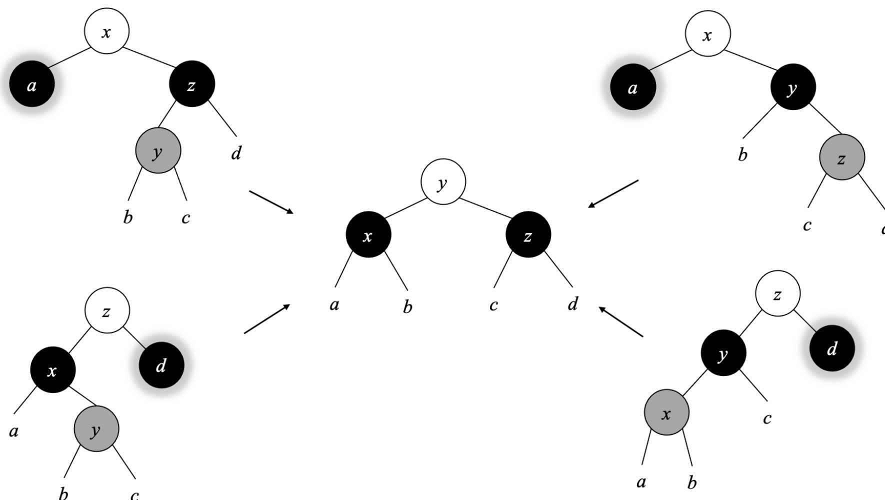



Figure 4.8: 4 sub-cases share the uniformed fixing pattern

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.

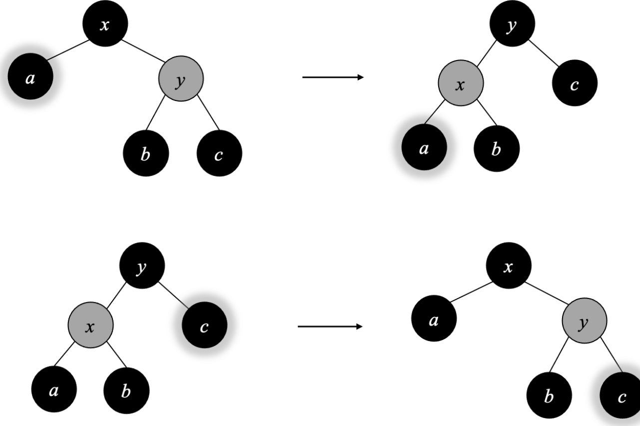



Figure 4.9: The sibling of the doubly-black is red.

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)
```

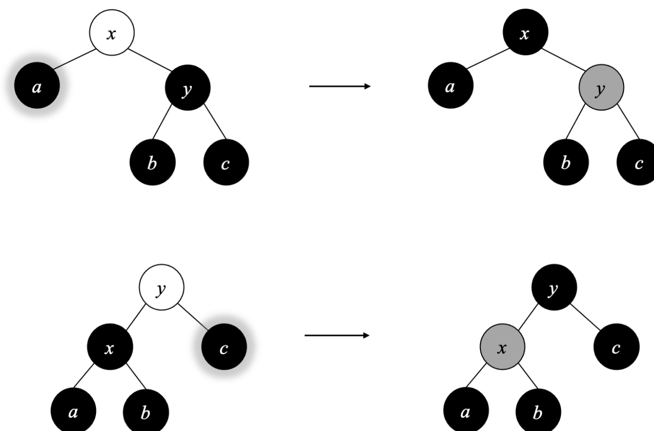



Figure 4.10: move the blackness up.

```
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.

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

```
1: function INSERT-FIX(T, x)
2:     while PARENT(x) ≠ NIL and COLOR(PARENT(x)) = RED do
3:         if COLOR(UNCLE(x)) = RED then            ▷ Case 1, x's uncle is red
4:             COLOR(PARENT(x)) ← BLACK
5:             COLOR(GRAND-PARENT(x)) ← RED
6:             COLOR(UNCLE(x)) ← BLACK
7:             x ← GRAND-PARENT(x)
8:         else                                      ▷ x's uncle is black
9:             if PARENT(x) = LEFT(GRAND-PARENT(x)) then
10:                if x = RIGHT(PARENT(x)) then      ▷ Case 2, x is on the right
11:                    x ← PARENT(x)
12:                    T ← LEFT-ROTATE(T, x)
                                                     ▷ Case 3, x is on the left
13:                COLOR(PARENT(x)) ← BLACK
14:                COLOR(GRAND-PARENT(x)) ← RED
15:                T ← RIGHT-ROTATE(T, GRAND-PARENT(x))
16:            else
17:                if x = LEFT(PARENT(x)) then       ▷ Case 2, Symmetric
18:                    x ← PARENT(x)
19:                    T ← RIGHT-ROTATE(T, x)
                                                     ▷ Case 3, Symmetric
20:                COLOR(PARENT(x)) ← BLACK
21:                COLOR(GRAND-PARENT(x)) ← RED
```

22: $T \leftarrow$ LEFT-ROTATE($T$, GRAND-PARENT($x$))
23: COLOR($T$) $\leftarrow$ BLACK
24: **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].

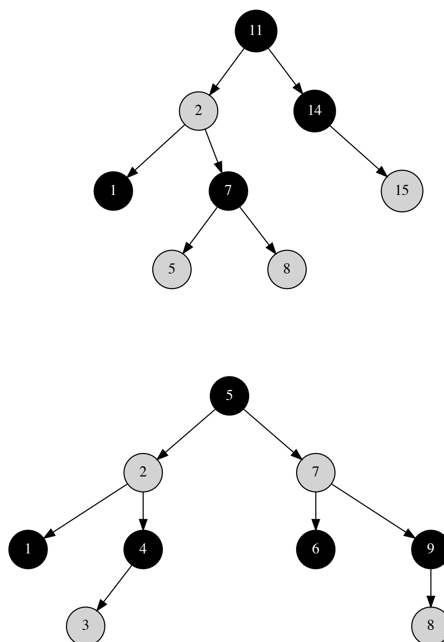



Figure 4.11: Red-black trees created by imperative algorithm.

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

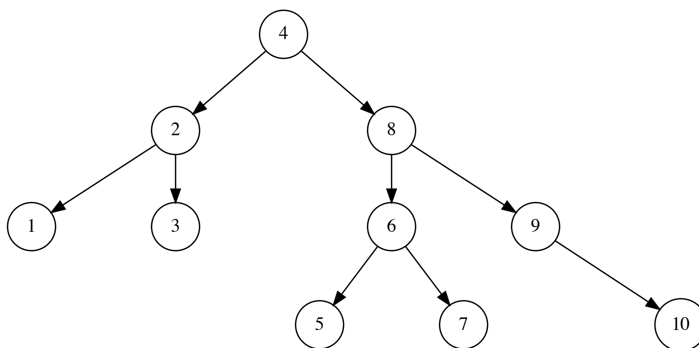



Figure 5.1: an AVL tree

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 $\delta(T)$: $\pm 1$, 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:

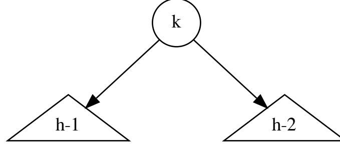



Figure 5.2: An AVL tree of height $h$. The height of one sub-tree is $h-1$, the other is no less than $h-2$.

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

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 $\delta$ 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 $\delta$ 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 $\Delta H$ 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 $\delta$. The result is $(T', \Delta H)$, where $T'$ is the new tree, and $\Delta H$ 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 $\delta$ [20].

Where $\delta' = \delta(T') = |r'| - |l'|$, is the updated balance factor. Appendix B provides the proof for it. We need determine $\delta'$ 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 $\pm 2$, exceeds the range of $[-1, 1]$. We adjust them to a uniformed structure in the center, with the $\delta(y) = 0$.

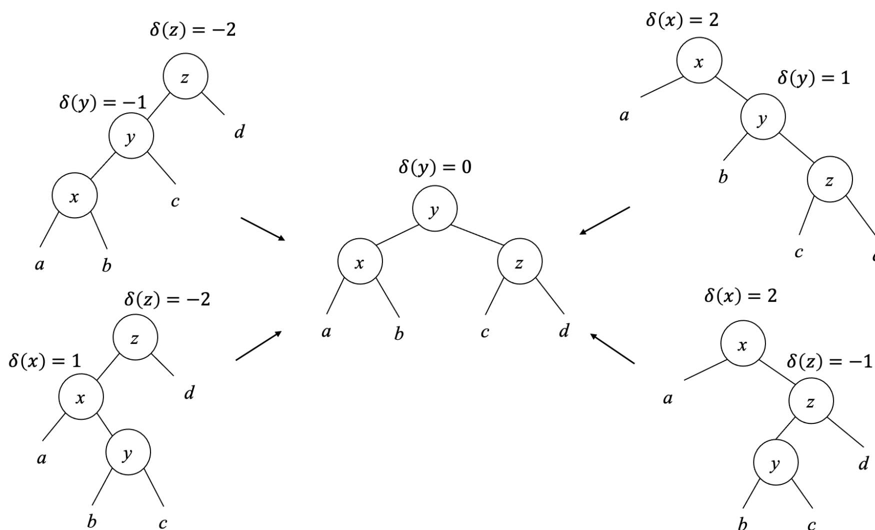



Figure 5.3: Fix 4 cases to the same structure

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:

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

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.

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

After insert, the balance factor $\delta$ may change because of the tree growth. Insert to the right may increase $\delta$ 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 $\delta'$, 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.

```
1: function AVL-Insert-Fix(T, x)
2:     while Parent(x) ≠ NIL do
3:         P ← Parent(x)
4:         L ← Left(x)
5:         R ← Right(x)
6:         δ ← δ(P)
7:         if x = Left(P) then
8:             δ′ ← δ − 1
9:         else
10:            δ′ ← δ + 1
11:        δ(P) ← δ′
12:        if |δ| = 1 and |δ′| = 0 then                  ▷ Height unchanged
13:            return T
14:        else if |δ| = 0 and |δ′| = 1 then             ▷ Go on bottom-up update
15:            x ← P
16:        else if |δ| = 1 and |δ′| = 2 then
17:            if δ′ = 2 then
18:                if δ(R) = 1 then                      ▷ Right-right
19:                    δ(P) ← 0                          ▷ By (B.6)
20:                    δ(R) ← 0
21:                    T ← Left-Rotate(T, P)
22:                if δ(R) = −1 then                     ▷ Right-left
23:                    δ_y ← δ(Left(R))                  ▷ By (B.17)
24:                    if δ_y = 1 then
25:                        δ(P) ← −1
26:                    else
27:                        δ(P) ← 0
28:                    δ(Left(R)) ← 0
29:                    if δ_y = −1 then
30:                        δ(R) ← 1
31:                    else
32:                        δ(R) ← 0
33:                    T ← Right-Rotate(T, R)
34:                    T ← Left-Rotate(T, P)
35:            if δ′ = −2 then
36:                if δ(L) = −1 then                     ▷ Left-left
37:                    δ(P) ← 0
38:                    δ(L) ← 0
39:                    Right-Rotate(T, P)
40:                else                                  ▷ Left-Right
41:                    δ_y ← δ(Right(L))
42:                    if δ_y = 1 then
43:                        δ(L) ← −1
44:                    else
45:                        δ(L) ← 0
46:                    δ(Right(L)) ← 0
47:                    if δ_y = −1 then
48:                        δ(P) ← 1
49:                    else
50:                        δ(P) ← 0
51:                    Left-Rotate(T, L)
```

52: RIGHT-ROTATE($T, P$)
53: break
54: **return** $T$

Besides rotation, we also need update $\delta$ 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].

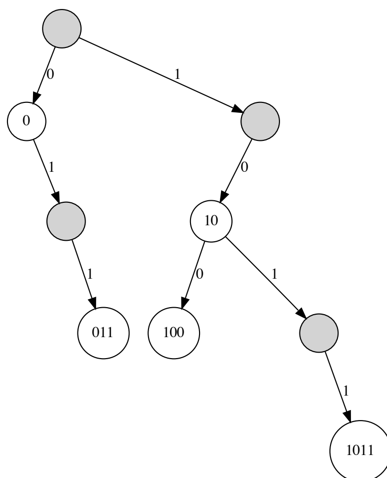



Figure 6.1: Radix tree.

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*:

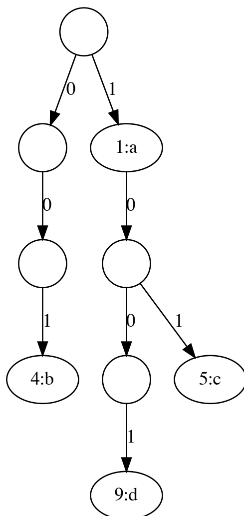



Figure 6.3: A little-endian integer binary trie of $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$.

```
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:

1: **function** INSERT($T, k, v$)
2:     **if** $T =$ NIL **then**
3:         $T \leftarrow$ EMPTY-NODE     ▷ (NIL, Nothing, NIL)
4:     $p \leftarrow T$
5:     **while** $k \neq 0$ **do**
6:         **if** EVEN?($k$) **then**
7:             **if** LEFT($p$) = NIL **then**
8:                 LEFT($p$) $\leftarrow$ EMPTY-NODE
9:             $p \leftarrow$ LEFT($p$)
10:         **else**
11:             **if** RIGHT($p$) = NIL **then**
12:                 RIGHT($p$) $\leftarrow$ EMPTY-NODE
13:             $p \leftarrow$ RIGHT($p$)
14:         $k \leftarrow \lfloor k/2 \rfloor$
15:     VALUE($p$) $\leftarrow v$
16:     **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:

1: **function** LOOKUP($T, k$)
2: **while** $k \neq 0$ and $T \neq$NIL **do**
3: **if** EVEN?($k$) **then**
4: $T \leftarrow$ LEFT($T$)
5: **else**
6: $T \leftarrow$ RIGHT($T$)
7: $k \leftarrow \lfloor k/2 \rfloor$
8: **if** $T \neq$ NIL **then**
9: **return** VALUE($T$)
10: **else**
11: **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.

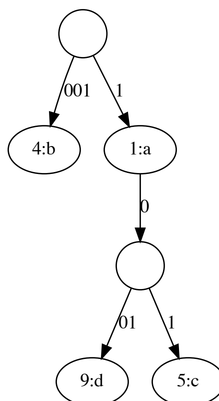



Figure 6.4: Little endian integer tree for the map $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$.

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

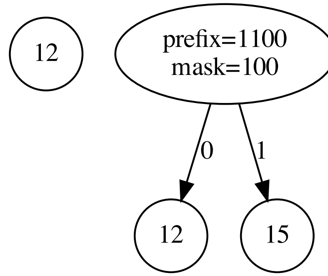
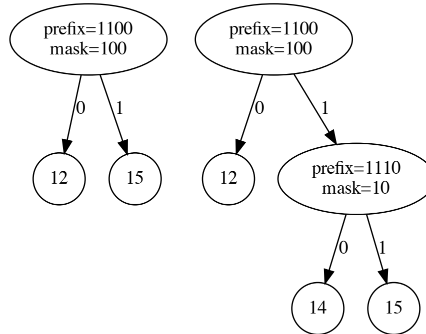
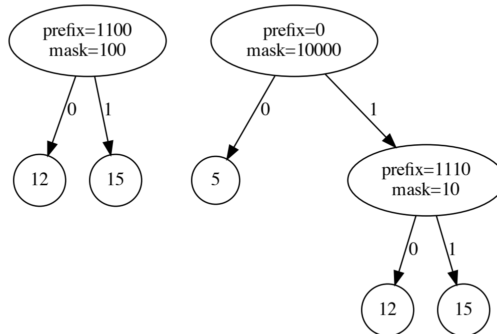



Figure 6.5: Left: $T$ is a leaf of 12; Right: After insert 15.

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:

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

1: **function** BRANCH($T_1, T_2$)
2: $T \leftarrow$ EMPTY-NODE
3: (PREFIX($T$), MASK($T$)) $\leftarrow$ LCP(PREFIX($T_1$), PREFIX($T_2$))
4: **if** ZERO?(PREFIX($T_1$), MASK($T$)) **then**
5: LEFT($T$) $\leftarrow T_1$
6: RIGHT($T$) $\leftarrow T_2$
7: **else**
8: LEFT($T$) $\leftarrow T_2$
9: RIGHT($T$) $\leftarrow T_1$
10: **return** $T$

11: **function** ZERO?($x, m$)
12: **return** $(x \& \lfloor \frac{m}{2} \rfloor) = 0$

Function LCP find the longest bit prefix from two integers:

1: **function** LCP($a, b$)
2: $d \leftarrow xor(a, b)$
3: $m \leftarrow 1$
4: **while** $d \neq 0$ **do**
5: $d \leftarrow \lfloor \frac{d}{2} \rfloor$

6: $m \leftarrow 2m$
7: **return** (MASKBIT$(a, m)$, $m$)

8: **function** MASKBIT$(x, m)$
9: **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 $O(m)$.

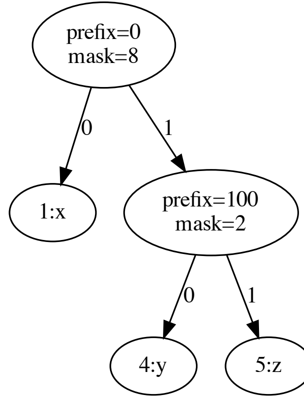



Figure 6.7: Insert $\{1 \rightarrow x, 4 \rightarrow y, 5 \rightarrow z\}$ to the big-endian integer tree.

### 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.

1: **function** LOOK-UP$(T, k)$
2: **if** $T =$ NIL **then**
3: **return** NIL
4: **while** $T$ is not leaf, and MATCH($k$, PREFIX($T$), MASK($T$)) **do**
5: **if** ZERO?($k$, MASK($T$)) **then**
6: $T \leftarrow$ LEFT($T$)
7: **else**
8: $T \leftarrow$ RIGHT($T$)

9: **if** $T$ is leaf, and KEY$(T) = k$ **then**
10: **return** VALUE$(T)$
11: **else**
12: **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).

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

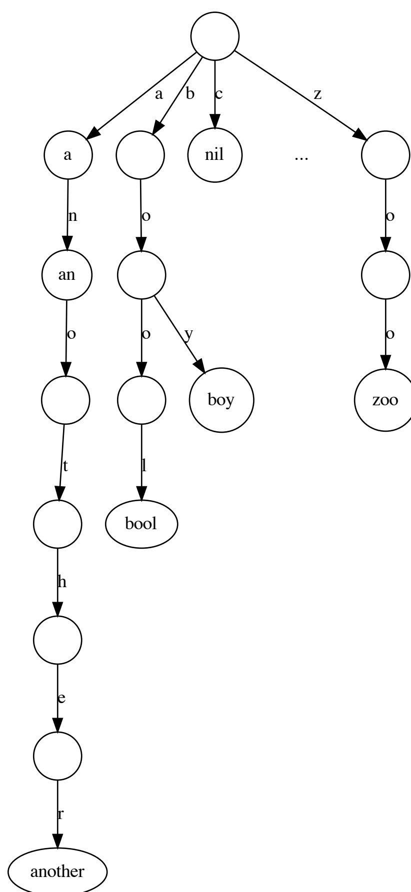



Figure 6.8: A trie of 26 branches, containing key 'a', 'an', 'another', 'bool', 'boy', and 'zoo'.

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*

7: $\quad\quad T \leftarrow$ SUB-TREES$(T)[c]$
8: $\quad$ **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 $s \doubleplus s_1$, $s \doubleplus s_2$, ... 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.

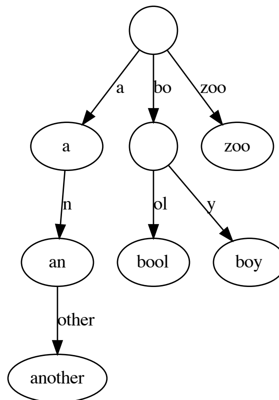



Figure 6.9: A prefix tree with keys: 'a', 'an', 'another', 'bool', 'boy', 'zoo'.

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) $\to$ (e); or $s_i$ is the prefix of $s$ as shown in figure 6.10 (d) $\to$ (e).

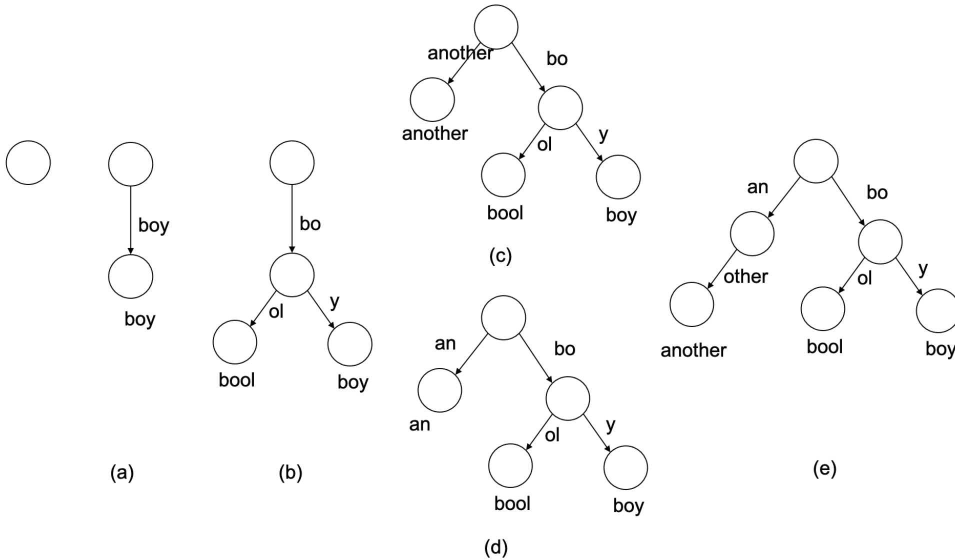



Figure 6.10: (a) insert 'boy' to empty tree; (b) insert 'bool', branch a new node out; (c) insert 'another' to (b); (d) insert 'an' to (b); (e) insert 'an' to (c), same result as insert 'another' to (d)

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:

```
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.

1: **function** INSERT($T, k, v$)
2:     **if** $T =$ NIL **then**
3:         $T \leftarrow$ EMPTY-NODE
4:     $p \leftarrow T$
5:     **loop**
6:         $match \leftarrow$ FALSE
7:         **for** each $s_i \mapsto T_i$ in SUB-TREES($p$) **do**
8:             **if** $k = s_i$ **then**
9:                 VALUE($T_i$) $\leftarrow v$ ▷ Overwrite
10:                 **return** $T$

11: $c \leftarrow$ LCP$(k, s_i)$
12: $k_1 \leftarrow k - c$, $k_2 \leftarrow s_i - c$
13: **if** $c \neq$ NIL **then**
14: *match* $\leftarrow$ TRUE
15: **if** $k_2 =$ NIL **then** ▷ $s_i$ is prefix of $k$
16: $p \leftarrow T_i$, $k \leftarrow k_1$
17: break
18: **else** ▷ Branch out a new leaf
19: ADD(SUB-TREES($p$), $c \mapsto$ BRANCH($k_1$, LEAF($v$), $k_2$, $T_i$))
20: DELETE(SUB-TREES($p$), $s_i \mapsto T_i$)
21: **return** $T$
22: **if** not *match* **then** ▷ Add a new leaf
23: ADD(SUB-TREES($p$), $k \mapsto$ LEAF($v$))
24: break
25: **return** $T$

Function LCP extracts the longest common prefix from two lists.

1: **function** LCP$(A, B)$
2: $i \leftarrow 1$
3: **while** $i \leq |A|$ and $i \leq |B|$ and $A[i] = B[i]$ **do**
4: $i \leftarrow i + 1$
5: **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.

1: **function** BRANCH$(s_1, T_1, s_2, T_2)$
2: **if** $s_1 =$ NIL **then**
3: ADD(SUB-TREES($T_1$), $s_2 \mapsto T_2$)
4: **return** $T_1$
5: $T \leftarrow$ EMPTY-NODE
6: SUB-TREES($T$) $\leftarrow \{s_1 \mapsto T_1, s_2 \mapsto T_2\}$
7: **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.

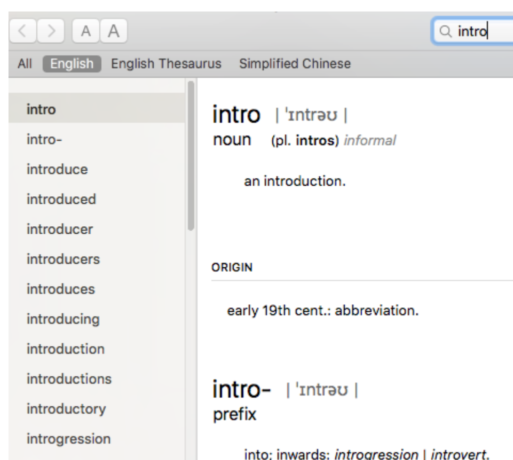



Figure 6.11: A dictionary application

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

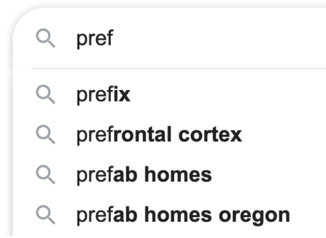



Figure 6.12: A smart text input component

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.

$$
\begin{array}{rcl}
startsWith\ \varnothing\ (Nothing, ts) & = & enum\ ts \\
startsWith\ \varnothing\ (Just\ x, ts) & = & (\varnothing, x) : enum\ ts \\
startsWith\ s\ (v, ts) & = & find\ ((k, t) \mapsto s \sqsubseteq k \text{ or } k \sqsubseteq s)\ ts = \\
\end{array}
\begin{cases} Nothing: & \varnothing \\ Just\ (k, t): & [(k \mathbin{+\!\!+} a, b) | (a, b) \in startsWith\ (s - k)\ t] \end{cases}
\tag{6.16}
$$

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 prepand $(\varnothing, x)$ 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 prepand $k$ 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.

1: **function** STARTS-WITH$(T, k, n)$
2:     **if** $T =$ NIL **then**
3:         **return** NIL
4:     $s \leftarrow$ NIL

```
5:     repeat
6:         match ← FALSE
7:         for k_i ↦ T_i in SUB-TREES(T) do
8:             if k is prefix of k_i then
9:                 return EXPAND(s ++ k_i, T_i, n)
10:            if k_i is prefix of k then
11:                match ← TRUE
12:                k ← k − k_i                              ▷ drop the prefix
13:                T ← T_i
14:                s ← s ++ k_i
15:                break
16:    until not match
17:    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):

```
1: function EXPAND(s, T, n)
2:     R ← NIL
3:     Q ← [(s, T)]
4:     while |R| < n and Q ≠ NIL do
5:         (k, T) ← POP(Q)
6:         v ← VALUE(T)
7:         if v ≠ NIL then
8:             INSERT(R, (k, v))
9:         for k_i ↦ T_i in SUB-TREES(T) do
10:            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:

$$pfx = [(s, t) | (s \mapsto t) \in ts, digits(s) \sqsubseteq ds \text{ or } ds \sqsubseteq digits(s)]$$

$$\begin{array}{rcl} find_{T9}\ t\ \varnothing & = & [\varnothing] \\ find_{T9}\ (v, ts)\ ds & = & concatMap\ find\ pfx \end{array} \tag{6.21}$$

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 $(prefix \mathbin{+\!\!+} s, D', T')$, where $D'$ is the remaining digits to be searched. Then push this new tuple back to the queue.

1: **function** LOOK-UP-T9($T, D$)
2: $R \leftarrow$ NIL
3: **if** $T =$ NIL or $D =$ NIL **then**
4: **return** $R$
5: $n \leftarrow |D|$
6: $Q \leftarrow \{(\text{NIL}, D, T)\}$
7: **while** $Q \neq$ NIL **do**
8: $(prefix, D, T) \leftarrow$ POP($Q$)
9: **for** $(s \mapsto T') \in$ SUB-TREES($T$) **do**
10: $D' \leftarrow$ DIGITS($s$)
11: **if** $D' \sqsubset D$ **then** ▷ $D'$ is prefix of $D$
12: APPEND($R, (prefix \mathbin{+\!\!+} s)[1..n]$) ▷ limit the length to $n$
13: **else if** $D \sqsubset D'$ **then**
14: PUSH($Q, (prefix \mathbin{+\!\!+} s, D - D', T')$)
15: **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.None
}
```

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.None) {
    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.None
  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.None
    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.

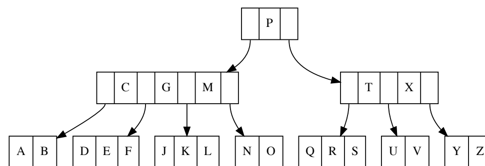



Figure 7.1: A B-Tree

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.

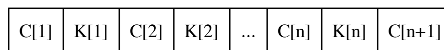



Figure 7.2: A B-Tree node

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 $(d, t)$.

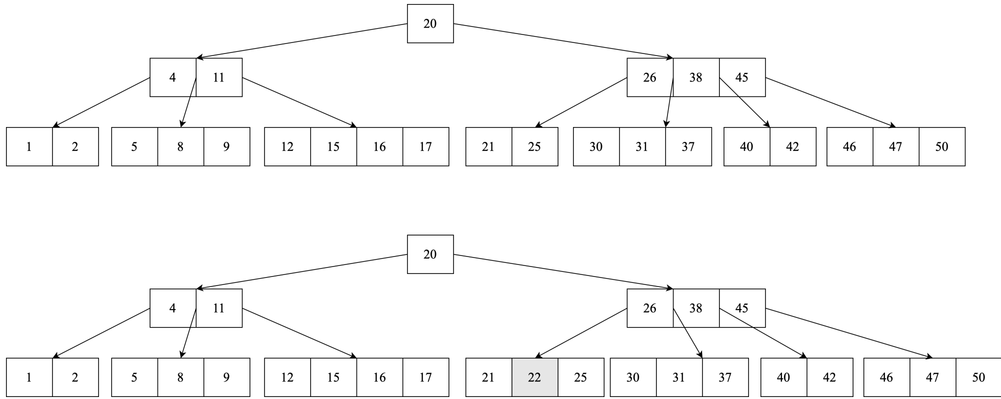



Figure 7.3: Insert 22 to a 2-3-4 tree. $22 > 20$, go to the right sub-tree; next as $22 < 26$, go to the first sub-tree; finally, $21 < 22 < 25$, and the leaf is not full.

## 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 1.55) to realize it.

[1]In fact, it is sufficient to only support less-than and equality. See exercise 1.

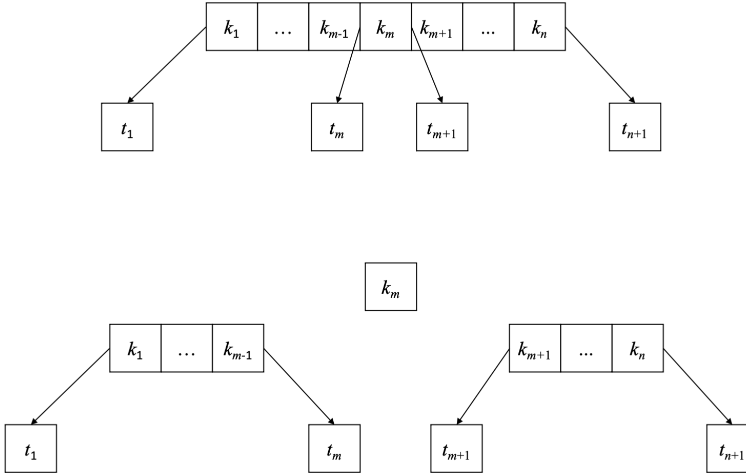



Figure 7.4: Split the node into 3 parts at $m$

$$
\begin{cases}
(ks_l, (k : ks_r)) & = splitAt\ (m-1)\ ks \\
(ts_l, ts_r) & = splitAt\ m\ ts
\end{cases}
$$

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 1.13); 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 1.58):

$$
\begin{cases}
(ks_l, ks_r) & = span\ (< x)\ ks \\
(ts_l, (t' : ts_r)) & = splitAt\ |ks_l|\ ts
\end{cases}
$$

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.

$$
\begin{array}{rclr}
ins\ (ks, \varnothing) & = & (insert_L\ x\ ks, \varnothing) & \text{list insert for leaf} \\
ins\ (ks, ts) & = & balance\ d\ l\ (ins\ t')\ r & \text{where } (l, t', r) = partition\ x\ t
\end{array} \tag{7.12}
$$

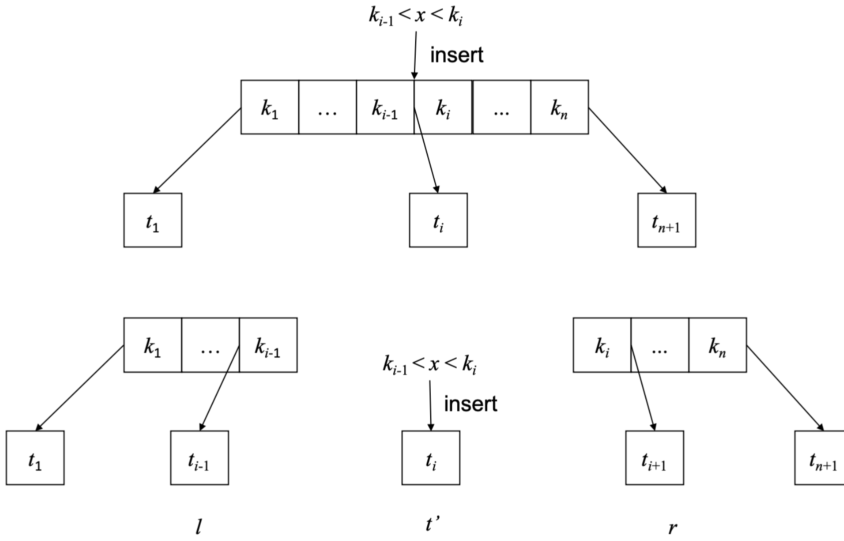



Figure 7.5: partition a node with $x$

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".

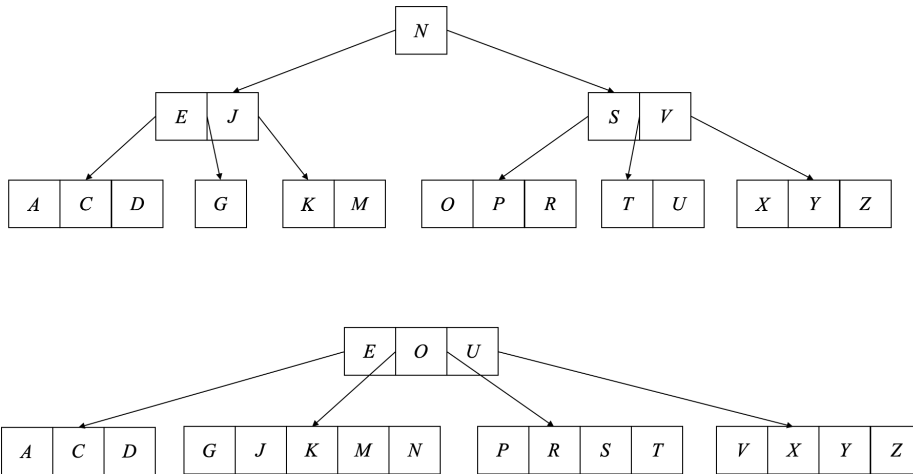



Figure 7.6: Repeatedly insert elements from "GMPXACDEJKNORSTUVYZ". above: $d = 2$ (2-3-4 tree), below: $d = 3$

### 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 $z$:

1: **procedure** SPLIT($z, i$)
2: $\quad d \leftarrow$ DEG($z$)
3: $\quad x \leftarrow t_i(z)$
4: $\quad y \leftarrow$ CREATE-NODE
5: $\quad K(y) \leftarrow [k_{d+1}(x), k_{d+2}(x), ..., k_{2d-1}(x)]$
6: $\quad K(x) \leftarrow [k_1(x), k_2(x), ..., k_{d-1}(x)]$
7: $\quad$ **if** $x$ is not leaf **then**
8: $\quad\quad T(y) \leftarrow [t_{d+1}(x), t_{d+2}(x), ..., t_{2d}(x)]$
9: $\quad\quad T(x) \leftarrow [t_1(x), t_2(x), ..., t_d(x)]$
10: $\quad$ INSERT-AT($K(z), i, k_d(x)$)
11: $\quad$ 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:

1: **function** INSERT($t, k$)
2: $\quad r \leftarrow t$

```
3:      if r is full then                                    ▷ root is full
4:          s ← CREATE-NODE
5:          T(s) ← [r]
6:          SPLIT(s, 1)
7:          r ← s
8:      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.

```
1: function INSERT-NONFULL(r, k)
2:      n ← |K(r)|
3:      if r is leaf then
4:          i ← 1
5:          while i ≤ n and k > k_i(r) do
6:              i ← i + 1
7:          INSERT-AT(K(r), i, k)
8:      else
9:          i ← n
10:         while i > 1 and k < k_i(r) do
11:             i ← i − 1
12:         if t_i(r) is full then
13:             SPLIT(r, i)
14:             if k > k_i(r) then
15:                 i ← i + 1
16:         INSERT-NONFULL(t_i(r), k)
17:     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”.

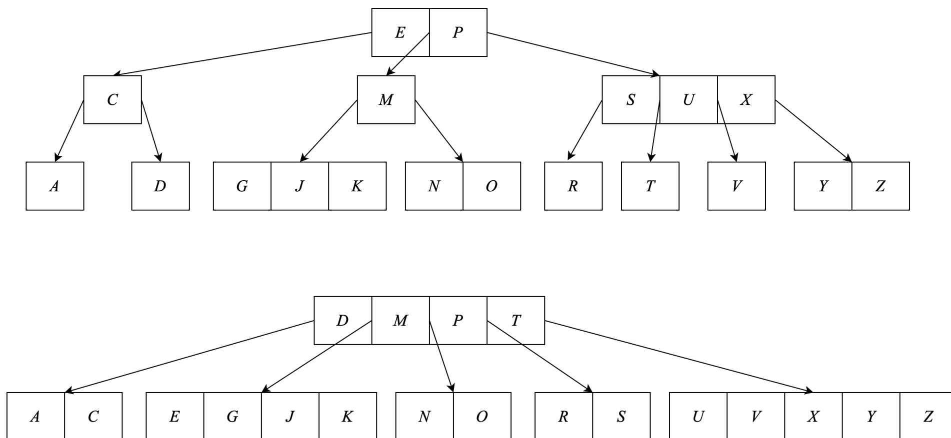



Figure 7.7: Insert from “GMPXACDEJKNORSTUVYZ”. up: $d = 2$, 2-3-4 tree; bottom: $d = 3$.

### 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.

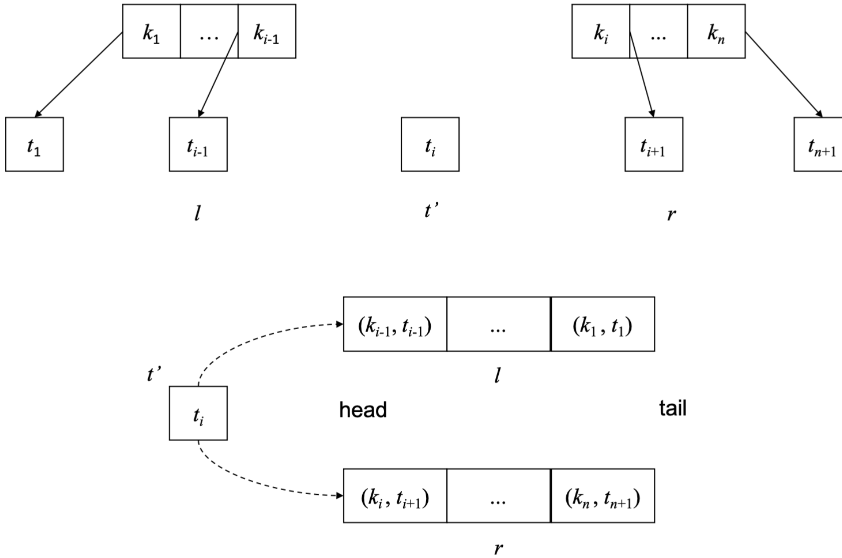



Figure 7.8: Define the B-tree node with a sub-tree and paired lists

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.

```
1:  function LOOK-UP(r, k)
2:      loop
3:          i ← 1, n ← |K(r)|
4:          while i ≤ n and k > k_i(r) do
5:              i ← i + 1
6:          if i ≤ n and k = k_i(r) then
7:              return (r, i)
8:          if r is leaf then
9:              return Nothing                    ▷ k does not exist
10:         else
11:             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 $t'$.

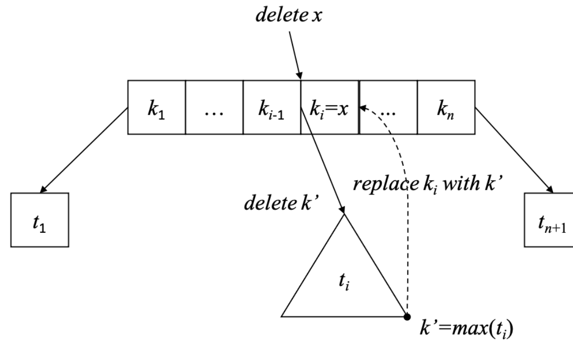



Figure 7.9: Replace $k_i$ with $k' = max(t')$, then recursively delete $k'$ from $t'$.

$$\begin{array}{rcl}
del\ x\ (ks, \varnothing) & = & (delete_l\ x\ ks, \varnothing) \\
del\ x\ t & = & \begin{cases} Just\ x = safeHd\ ks' : & balance\ d\ l\ (del\ k'\ t')\ (k' : (tail\ ks'), ts') \\ otherwise : & balance\ d\ l\ (del\ x\ t')\ (ks', ts') \end{cases}
\end{array} \tag{7.28}$$

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 1.16 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.

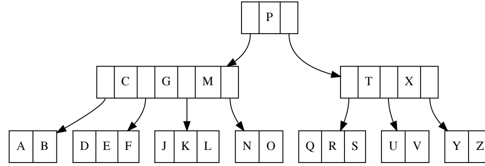
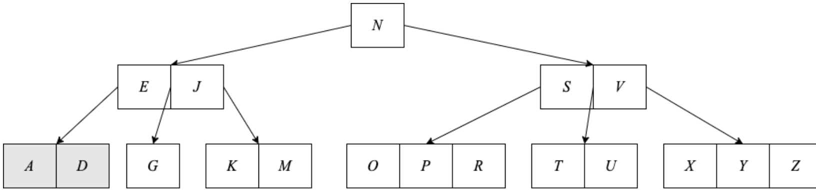



Figure 7.10: Before delete



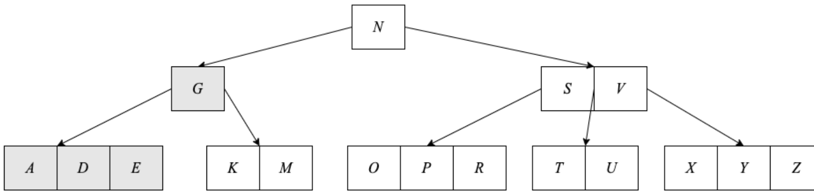
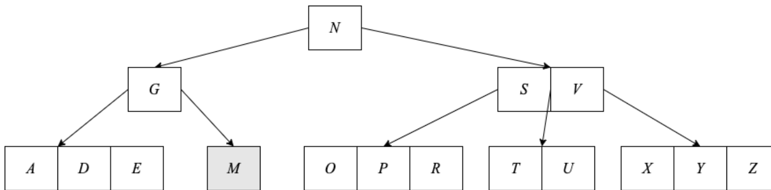




Figure 7.11: Delete 'C', then delete 'J'



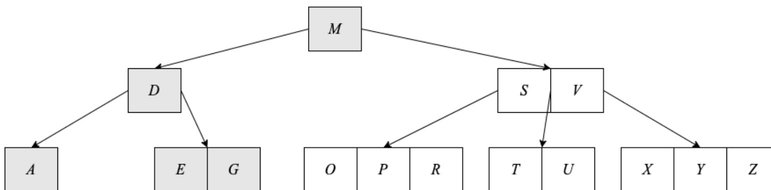




Figure 7.12: Delete 'K', then delete 'N'

### 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.

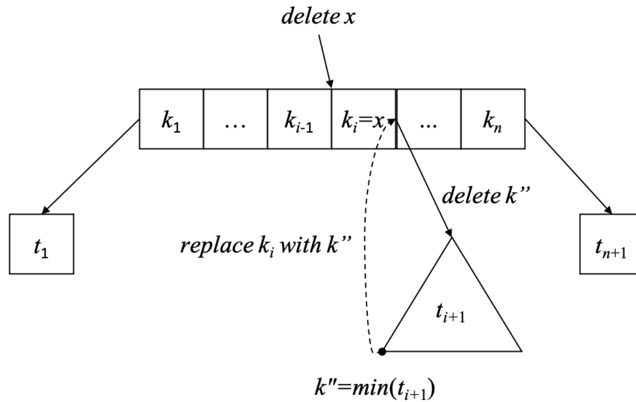



Figure 7.13: Replace $k_i$ with $k'' = min(t_{i+1})$, then delete $k''$ from $t_{i+1}$.

**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.

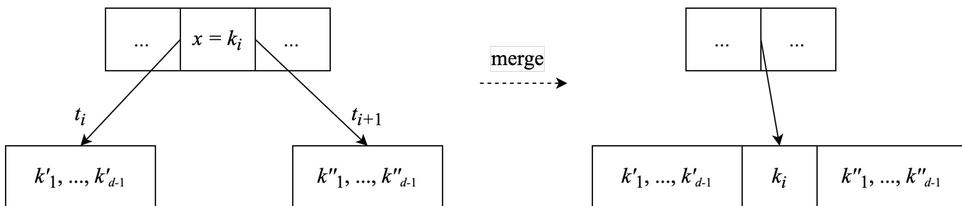



Figure 7.14: Merge before delete

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 $t_i$.

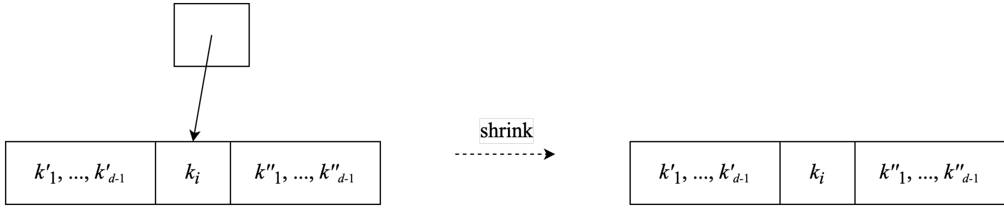



Figure 7.15: Shrink

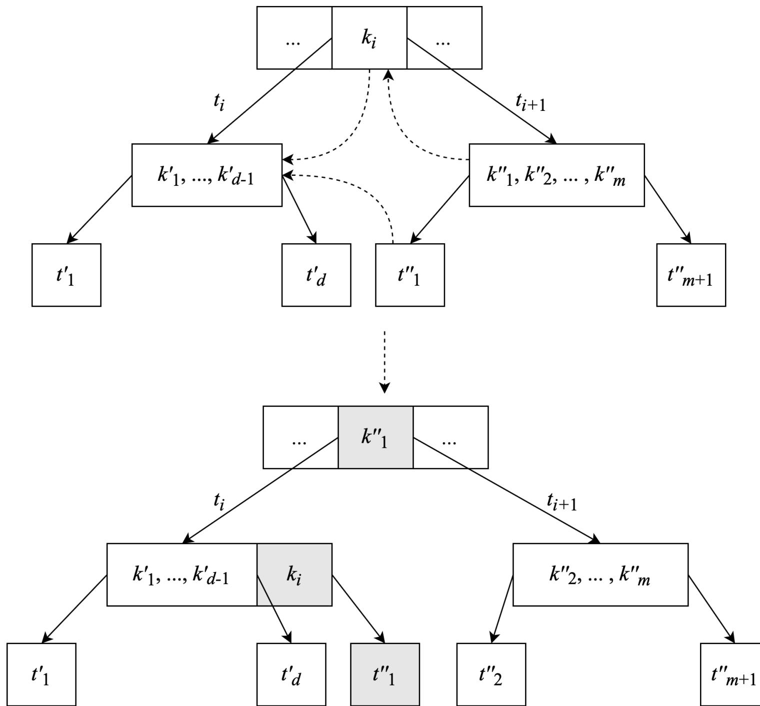



Figure 7.16: Borrow from the right sibling.

**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.

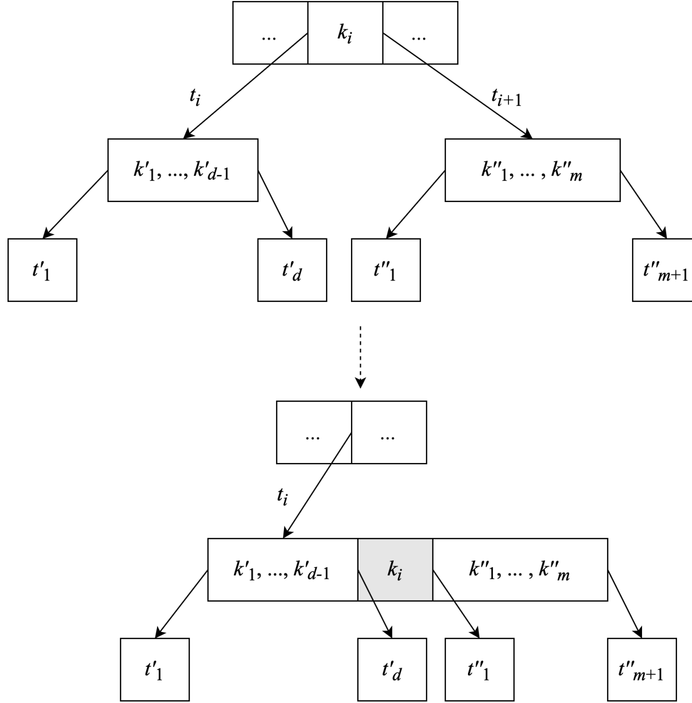



Figure 7.17: Merge $t_i$, $k$, $t_{i+1}$

Below DELETE algorithm implements the 'merge then delete' method:

```
1:  function DELETE(t, k)
2:      if t is empty then
3:          return t
4:      i ← 1, n ← |K(t)|
5:      while i ≤ n and k > k_i(t) do
6:          i ← i + 1
7:      if k = k_i(t) then
8:          if t is leaf then                                    ▷ case 1
9:              REMOVE(K(t), k)
10:         else                                                 ▷ case 2
11:             if |K(t_i(t))| ≥ d then                          ▷ case 2a
12:                 k_i(t) ← MAX(t_i(t))
13:                 DELETE(t_i(t), k_i(t))
14:             else if |K(t_{i+1}(t))| ≥ d then                 ▷ case 2b
15:                 k_i(t) ← MIN(t_{i+1}(t))
16:                 DELETE(t_{i+1}(t), k_i(t))
17:             else                                             ▷ case 2c
18:                 MERGE-AT(t, i)
19:                 DELETE(t_i(t), k)
20:                 if K(T) is empty then
21:                     t ← t_i(t)                               ▷ Shrinks height
22:         return t
```

23: **if** $t$ is not leaf **then**
24: **if** $k > k_n(t)$ **then**
25: $i \leftarrow i + 1$
26: **if** $|K(t_i(t))| < d$ **then** ▷ case 3
27: **if** $i > 1$ and $|K(t_{i-1}(t))| \geq d$ **then** ▷ case 3a: left
28: INSERT($K(t_i(t)), k_{i-1}(t)$)
29: $k_{i-1}(t) \leftarrow$ POP-LAST($K(t_{i-1}(t))$)
30: **if** $t_i(t)$ is not leaf **then**
31: INSERT($T(t_i(t))$, POP-BACK($T(t_{i-1}(t))$))
32: **else if** $i \leq n$ and $|K(t_{i+1}(t))| \geq d$ **then** ▷ case 3a: right
33: APPEND($K(t_i(t)), k_i(t)$)
34: $k_i(t) \leftarrow$ POP-FIRST($K(t_{i+1}(t))$)
35: **if** $t_i(t)$ is not leaf **then**
36: APPEND($T(t_i(t))$, POP-FIRST($T(t_{i+1}(t))$))
37: **else** ▷ case 3b
38: **if** $i = n + 1$ **then**
39: $i \leftarrow i - 1$
40: MERGE-AT($t, i$)
41: DELETE($t_i(t), k$)
42: **if** $K(t)$ is empty **then** ▷ Shrinks height
43: $t \leftarrow t_1(t)$
44: **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.

1: **procedure** MERGE-AT($t, i$)
2: $x \leftarrow t_i(t)$
3: $y \leftarrow t_{i+1}(t)$
4: $K(x) \leftarrow K(x) \mathbin{+\!\!+} [k_i(t)] \mathbin{+\!\!+} K(y)$
5: $T(x) \leftarrow T(x) \mathbin{+\!\!+} T(y)$
6: REMOVE-AT($K(t), i$)
7: 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((tr, i))
        if isLeaf(tr) {
            return Optional.None
        } 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):

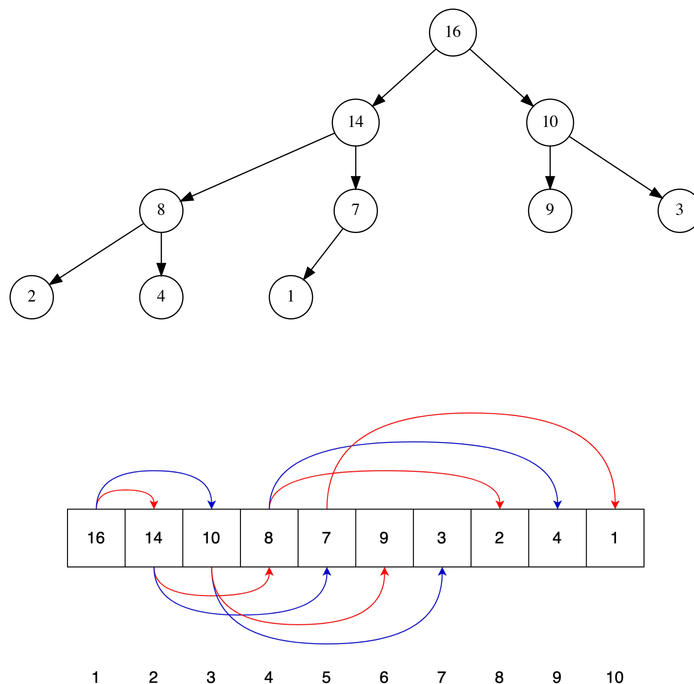



Figure 8.1: Map between a complete binary tree and an array.

$$
\begin{cases}
parent(i) & = \lfloor \frac{i}{2} \rfloor \\
left(i) & = 2i \\
right(i) & = 2i + 1
\end{cases}
\tag{8.1}
$$

### 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 $i$.

1: **function** HEAPIFY($A, i$)
2: $n \leftarrow |A|$
3: **loop**
4: $s \leftarrow i$ ▷ $s$ is the smallest
5: $l \leftarrow$ LEFT($i$), $r \leftarrow$ RIGHT($i$)
6: **if** $l \leq n$ and $A[l] < A[i]$ **then**
7: $s \leftarrow l$
8: **if** $r \leq n$ and $A[r] < A[s]$ **then**
9: $s \leftarrow r$
10: **if** $s \neq i$ **then**
11: EXCHANGE $A[i] \leftrightarrow A[s]$
12: $i \leftarrow s$
13: **else**
14: **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].

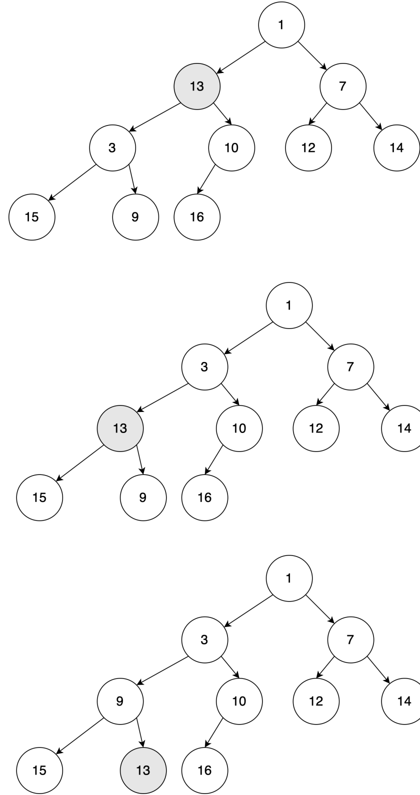


Figure 8.2: Heapify. Step 1: the minimum of 13, 3, 10 is 3, exchange 3 $\leftrightarrow$ 13; Step 2: the minimum of 13, 15, 9 is 9, exchange 13 $\leftrightarrow$ 9; Step 3: 13 is leaf, terminate.

### 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:

1: **function** BUILD-HEAP($A$)
2: $\quad n \leftarrow |A|$
3: $\quad$ **for** $i \leftarrow \lfloor n/2 \rfloor$ down to 1 **do**
4: $\quad\quad$ 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 ← A[1], n ← |A|
3:     Exchange A[1] ↔ 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 ← [ ]
3:     Build-Heap(A)
```

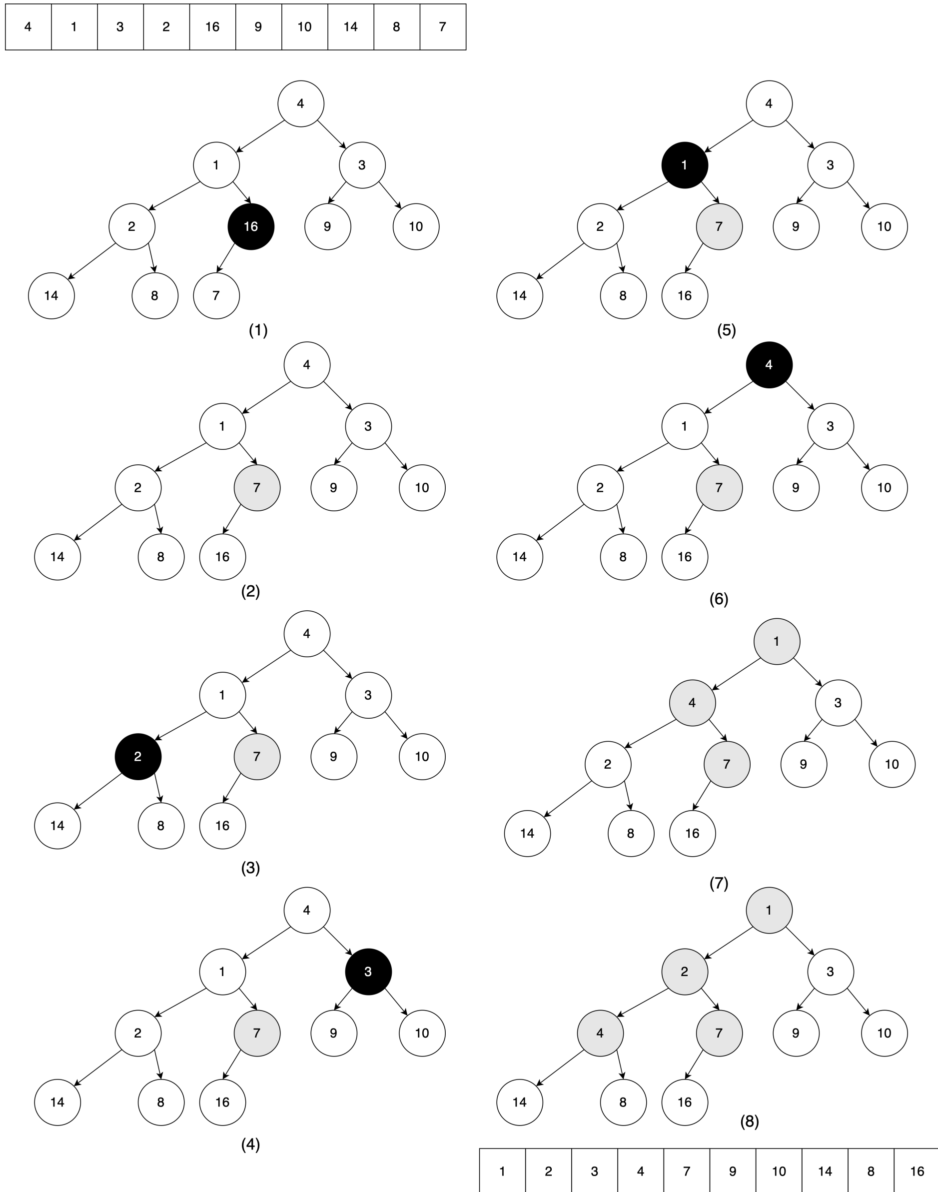



Figure 8.3: Build heap. (1) $16 > 7$; (2) exchange $16 \leftrightarrow 7$; (3) $2 < 14$ and $2 < 8$; (4) $3 < 9$ and $3 < 10$; (5) $1 < 2$ and $1 < 7$; (6) $1 < 4$ and $1 < 3$; (7) exchange $4 \leftrightarrow 1$; (8) exchange $4 \leftrightarrow 2$, end.

```
4:     loop MIN(k, |A|) times                    ▷ cut off when k out of array bound
5:         APPEND(R, POP(A))
6:     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.

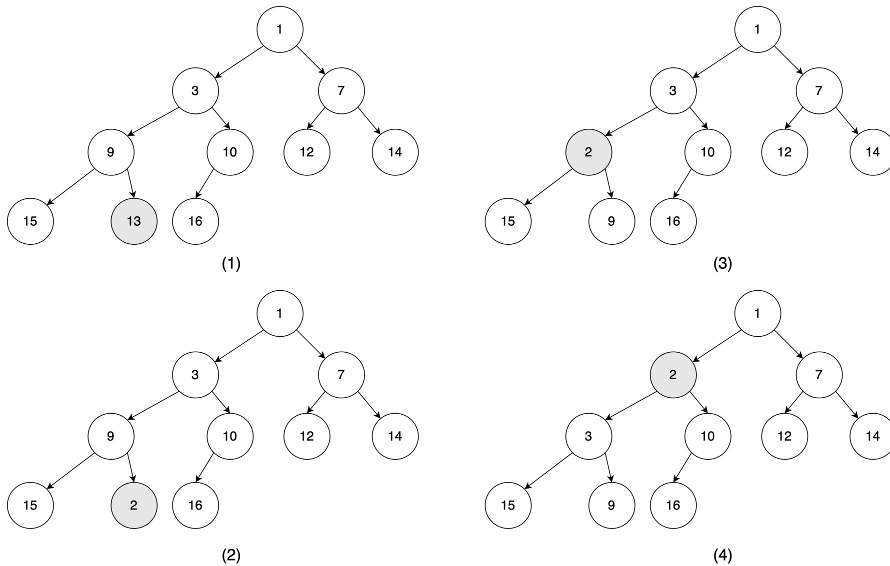



Figure 8.4: Decrease 13 to 2. Exchange 2 and 9, then exchange with 3.

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.

```
1: function HEAP-FIX(A, i)
2:     while i > 1 and A[i] < A[ PARENT(i) ] do
3:         EXCHANGE A[i] ↔ A[ PARENT(i) ]
4:         i ← 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:

```
1: function PUSH(A, k)
2:     APPEND(A, k)
3:     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.

```
1: function Heap-Sort(A)
2:     R ← [ ]
3:     Build-Heap(A)
4:     while A ≠ [ ] do
5:         Append(R, Pop(A))
6:     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.

```
1: function Heap-Sort(A)
2:     Build-Max-Heap(A)
3:     n ← |A|
4:     while n > 1 do
5:         Exchange A[1] ↔ A[n]
6:         n ← n − 1
7:         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?

```
1: function Heap-Sort(A)
2:     Build-Heap(A)
3:     for i = 1 to n − 1 do
4:         Heapify(A[i...n], 1)
```

2. Similarly, can we apply Heapify $k$ times from left to right to get the top-$k$ elements?

```
1: function Top-K(A, k)
2:     Build-Heap(A)
3:     n ← |A|
4:     for i ← 1 to min(k, n) do
5:         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:

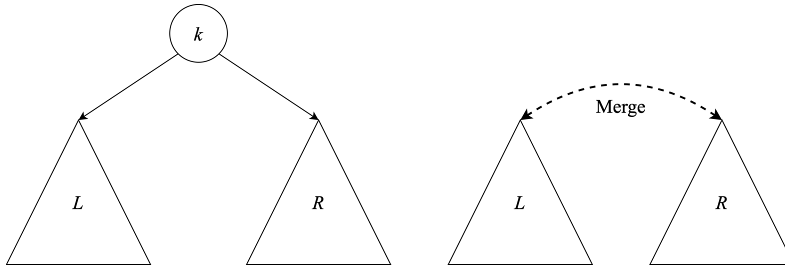



Figure 8.5: Merge left and right sub-trees after pop.

$$
\begin{array}{rcl}
merge(\varnothing, R) & = & R \\
merge(L, \varnothing) & = & L \\
merge(L, R) & = & ?
\end{array}
$$

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:

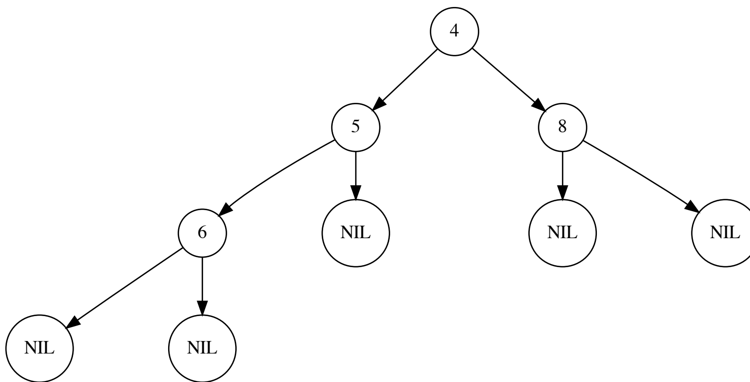



Figure 8.6: $rank(4) = 2$, $rank(6) = rank(8) = rank(5) = 1$.

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: $\quad C \leftarrow$ CONCAT$(A, B)$
3: $\quad$ 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 $build = fold_r\ insert\ \varnothing$.

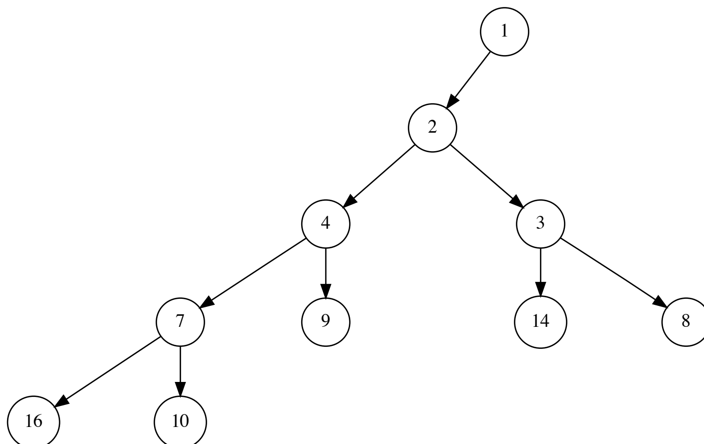



Figure 8.7: Build the leftist heap from $[9, 4, 16, 7, 10, 2, 14, 3, 8, 1]$.

### 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)
```

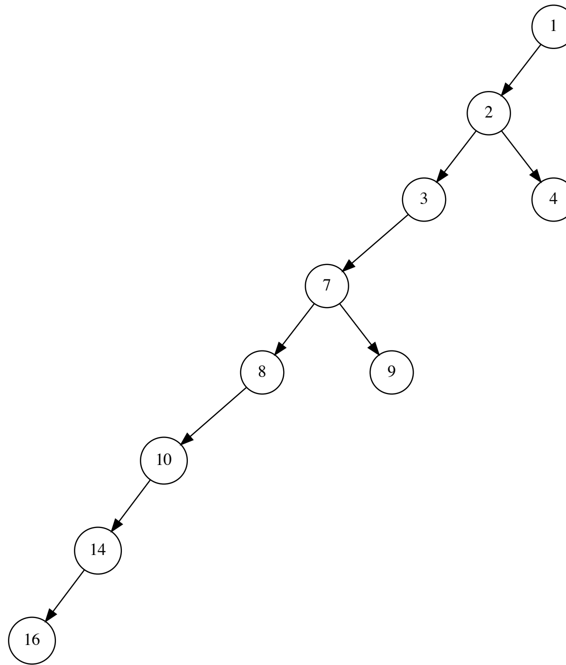



Figure 8.8: Leftist heap built from $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$.

**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

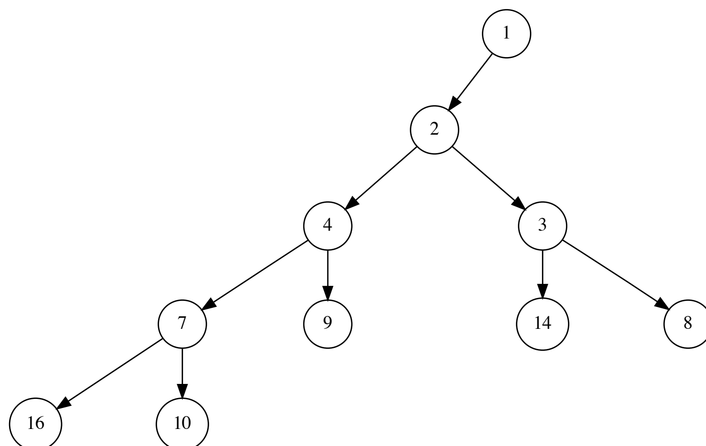



Figure 8.9: Skew tree built from $[1, 2, ..., 10]$.

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

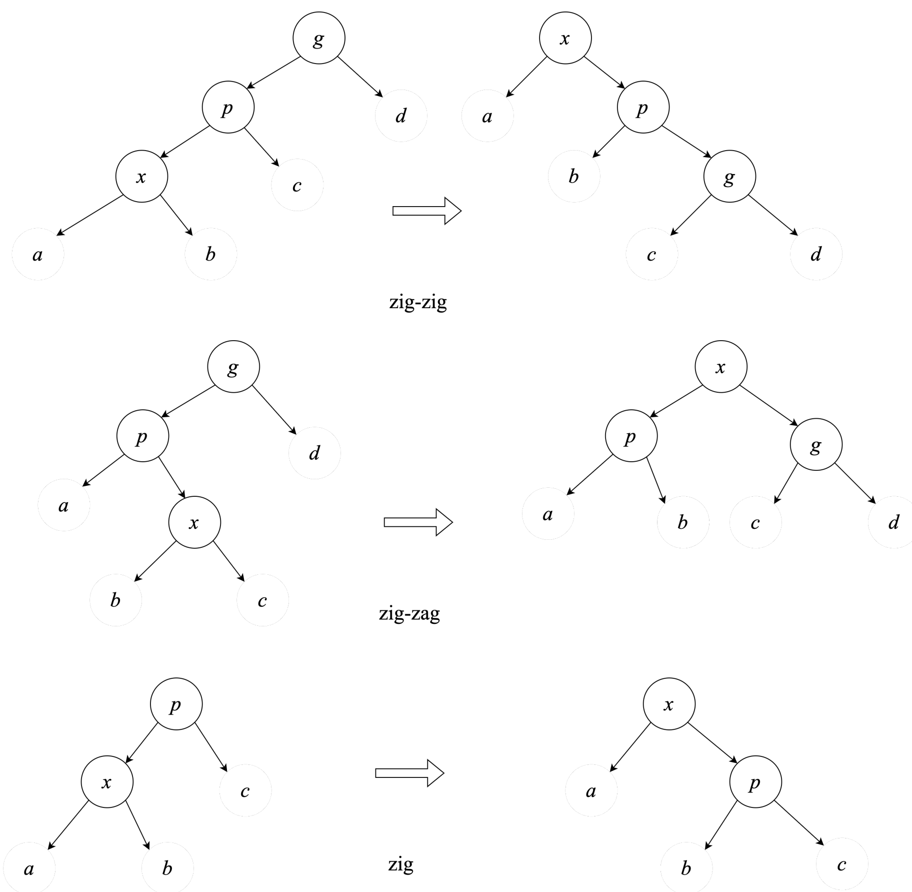



Figure 8.10: zig-zig: $x$ and $p$ are both on left or right, $x$ becomes new root. zig-zag: $x$ and $p$ are on different sides, $x$ becomes new root, $p$ and $g$ are siblings. zig: $p$ is root, rotate to make $x$ as root.

below when access element $y$:

$$
\begin{array}{rcll}
splay\ (((a, x, b), p, c), g, d)\ y & = & \begin{cases} x = y : & (a, x, (b, p, (c, g, d))) \\ otherwise : & T \end{cases} & \text{zig-zig} \\
splay\ (a, g, (b, p, (c, x, d)))\ y & = & \begin{cases} x = y : & (((a, g, b), p, c), x, d) \\ otherwise : & T \end{cases} & \text{zig-zig symmetric} \\
splay\ (a, p, (b, x, c), g, d)\ y & = & \begin{cases} x = y : & ((a, p, b), x, (c, g, d)) \\ otherwise : & T \end{cases} & \text{zig-zag} \\
splay\ (a, g, ((b, x, c), p, d))\ y & = & \begin{cases} x = y : & ((a, g, b), x, (c, p, d)) \\ otherwise : & T \end{cases} & \text{zig-zag symmetric} \\
splay\ ((a, x, b), p, c)\ y & = & \begin{cases} x = y : & (a, x, (b, p, c)) \\ otherwise : & T \end{cases} & \text{zig} \\
splay\ (a, p, (b, x, c))\ y & = & \begin{cases} x = y : & ((a, p, b), x, c) \\ otherwise : & T \end{cases} & \text{zig symmetric} \\
splay\ T\ y & = & T & \text{others}
\end{array}
\tag{8.13}
$$

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.

$$
\begin{array}{rcl}
insert\ \varnothing\ y & = & (\varnothing, y, \varnothing) \\
insert\ (L, x, R)\ y & = & \begin{cases} y < x : & splay\ ((insert\ L\ y), x, R)\ y \\ otherwise : & splay\ (L, x, (insert\ R\ y))\ y \end{cases}
\end{array}
\tag{8.14}
$$

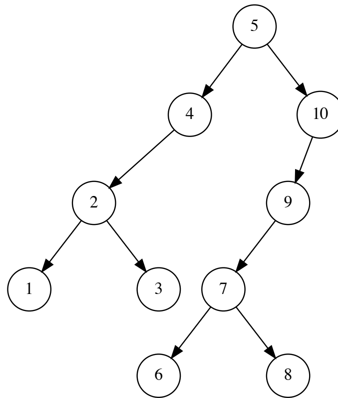



Figure 8.11: Splay tree built from $[1, 2, ..., 10]$.

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.

$$
\begin{array}{rcl}
partition\ \varnothing\ y & = & (\varnothing, \varnothing) \\
partition\ (L, x, R)\ y & = & \\
\multicolumn{3}{l}{
\begin{cases}
x < y & \begin{cases}
R = \varnothing & (T, \varnothing) \\
R = (L', x', R') & \begin{cases}
x' < y & (((L, x, L'), x', A), B) \\
 & \text{where: } (A, B) = partition\ R'\ y \\
\text{otherwise} & ((L, x, A), (B, x', R')) \\
 & \text{where: } (A, B) = partition\ L'\ y
\end{cases}
\end{cases} \\
\text{otherwise} & \begin{cases}
L = \varnothing & (\varnothing, T) \\
L = (L', x', R') & \begin{cases}
y < x' & (A, (L', x', (R', x, R))) \\
 & \text{where: } (A, B) = partition\ L'\ y \\
\text{otherwise} & ((L', x', A), (B, x, R)) \\
 & \text{where: } (A, B) = partition\ R'\ y
\end{cases}
\end{cases}
\end{cases}
}
\end{array}
\tag{8.15}
$$

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.

## 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:

```
1: function Sort(A)
2:     X ← [ ]
3:     while A ≠ [ ] do
4:         x ← Min(A)
5:         Del(A, x)
6:         Append(X, x)
7:     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 $A[i]$.

```
1: function Sort(A)
2:     for i ← 1 to |A| do
3:         m ← Min-At(A, i)
4:         Exchange A[i] ↔ A[m]
```

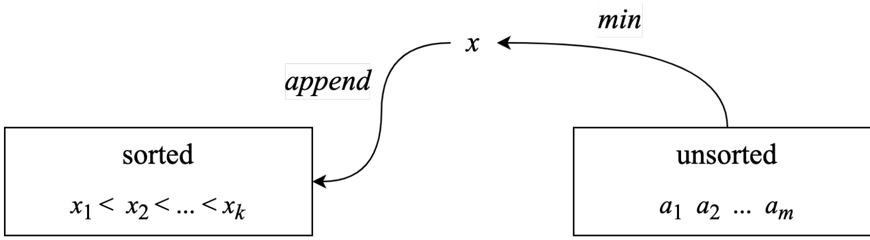



Figure 9.1: The left is sorted, repeatedly select the minimum of the rest and append.

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.

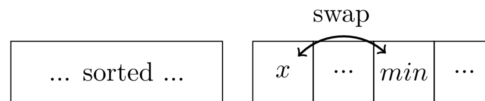



Figure 9.2: The left is sorted, repeatedly find the minimum and swap to the right position.

## 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 $n$.

1: **function** MIN-AT($A, i$)
2: $\quad m \leftarrow i$
3: $\quad$ **for** $i \leftarrow m + 1$ to $|A|$ **do**
4: $\quad\quad$ **if** $A[i] < A[m]$ **then**
5: $\quad\quad\quad m \leftarrow i$
6: $\quad$ **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:

```
1: procedure Sort(A)
2:     for i ← 1 to |A| do
3:         m ← i
4:         for j ← i + 1 to |A| do
5:             if A[i] < A[m] then
6:                 m ← i
7:         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 $A[i]$.

```
1: procedure Sort(A)
2:     for i ← 1 to |A| − 1 do
3:         m ← i
4:         for j ← i + 1 to |A| do
5:             if A[i] < A[m] then
6:                 m ← i
7:         if m ≠ i then
8:             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.

```
1: procedure Sort'(A)
2:     for i ← |A| down-to 2 do
3:         m ← i
4:         for j ← 1 to i − 1 do
5:             if A[m] < A[i] then
6:                 m ← i
7:         Exchange A[i] ↔ A[m]
```

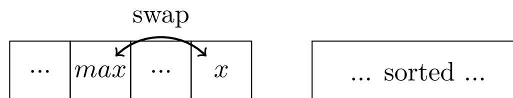



Figure 9.3: Select the maximum and swap to tail

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'.

```
1: procedure Sort(A)
2:     for i ← 1 to ⌊|A|/2⌋ do
3:         min ← i
4:         max ← |A| + 1 − i
5:         if A[max] < A[min] then
6:             Exchange A[min] ↔ A[max]
7:         for j ← i + 1 to |A| − i do
8:             if A[j] < A[min] then
```

9: $min \leftarrow j$
10: **if** $A[max] < A[j]$ **then**
11: $max \leftarrow j$
12: EXCHANGE $A[i] \leftrightarrow A[min]$
13: EXCHANGE $A[|A| + 1 - i] \leftrightarrow A[max]$

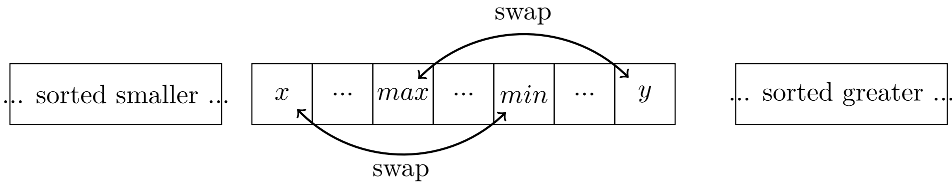



Figure 9.4: Find the minimum and maximum, swap both to the right positions.

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:

$$
\begin{array}{rcl}
sort'\ A\ B\ [\,] & = & (reverse\ A) \doubleplus B \\
sort'\ A\ B\ [x] & = & (reverse\ x : A) \doubleplus B \\
sort'\ A\ B\ (x : xs) & = & sort'\ (x_0 : A)\ xs'\ (x_1 : B)
\end{array}
\tag{9.12}
$$

## 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.

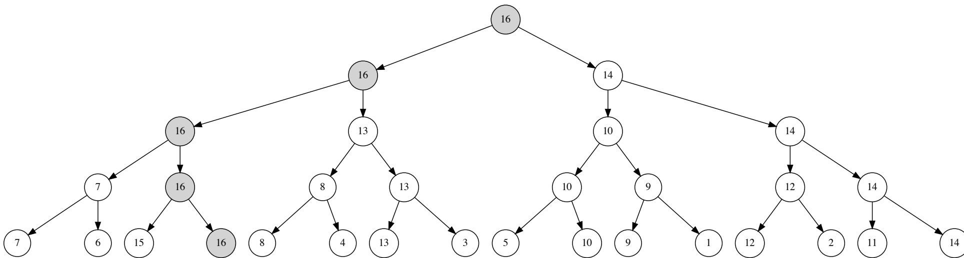
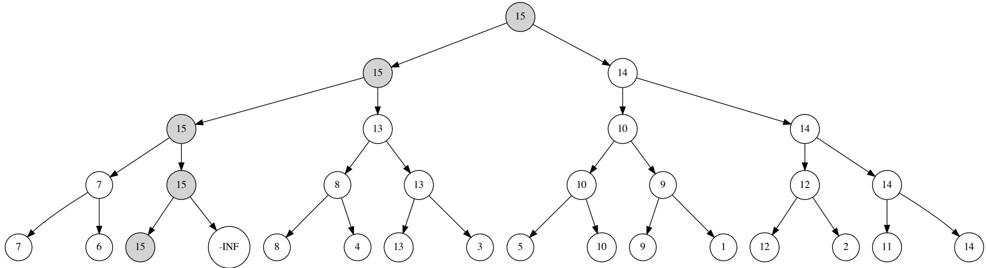



Figure 9.5: The element 15 is knocked out in the first round.

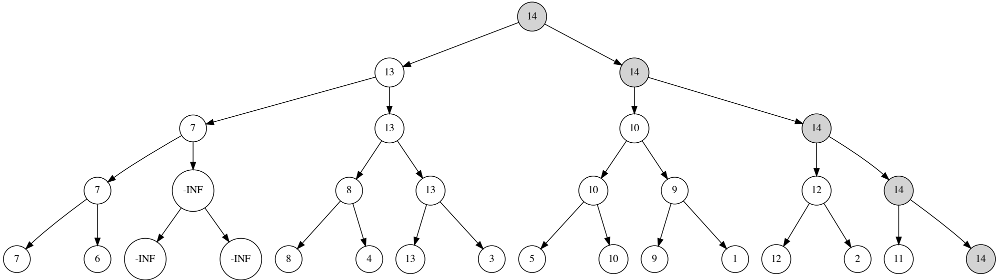
Assign every team a number to measure its strength. Suppose the team with greater number always beats the smaller one (this is obviously not true in real world). The champion number is 16. the runner-up is not 14, but 15, which is out in the first round. We need figure out a way to quickly identify the 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:

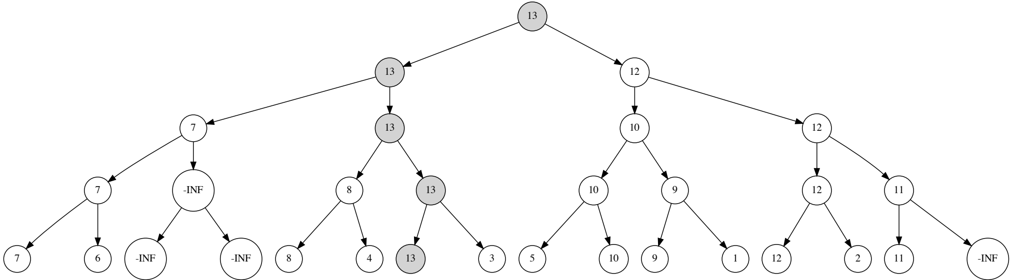
1. Build a tournament tree with the maximum (the champion) at the root;
2. Take the root, replace it with $-\infty$ along the path to leaf;
3. Perform a bottom-up back-track along the path, find the new champion and store it in the root;
4. Repeat step 2 to process all elements.




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.

```
1: function Build-Tree(A)
2:     T ← [ ]
3:     for each x ∈ A do
4:         Append(T, Node(NIL, x, NIL))
5:     while |T| > 1 do
6:         T′ ← [ ]
7:         for every t1, t2 ∈ T do
8:             k ← Max(Key(t1), Key(t2))
9:             Append(T′, Node(t1, k, t2))
10:        if |T| is odd then
11:            Append(T′, Last(T))
12:        T ← T′
13:    return T[1]
```

We replace the root with $-\infty$ top-down, then back-track through the parent field to find the new maximum.

```
1: function Pop(T)
2:     m ← Key(T)
3:     Key(T) ← −∞
4:     while T is not leaf do                ▷ top-down replace m with −∞.
5:         if Key(Left(T)) = m then
6:             T ← Left(T)
7:         else
8:             T ← Right(T)
9:         Key(T) ← −∞
10:    while Parent(T) ≠ NIL do              ▷ bottom-up to find the new maximum.
11:        T ← Parent(T)
12:        Key(T) ← Max(Key(Left(T)), Key(Right(T)))
13:    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:

$$(a+b)^n = a^n + \binom{n}{1} a^{n-1}b + ... + \binom{n}{n-1} ab^{n-1} + b \tag{10.1}$$

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.

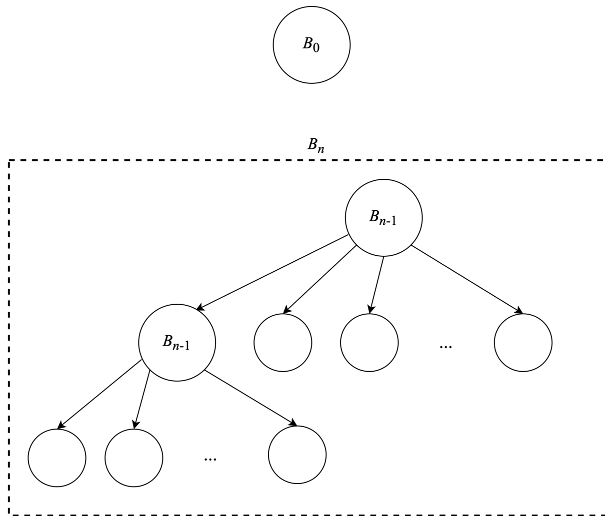



Figure 10.1: Binomial tree

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.

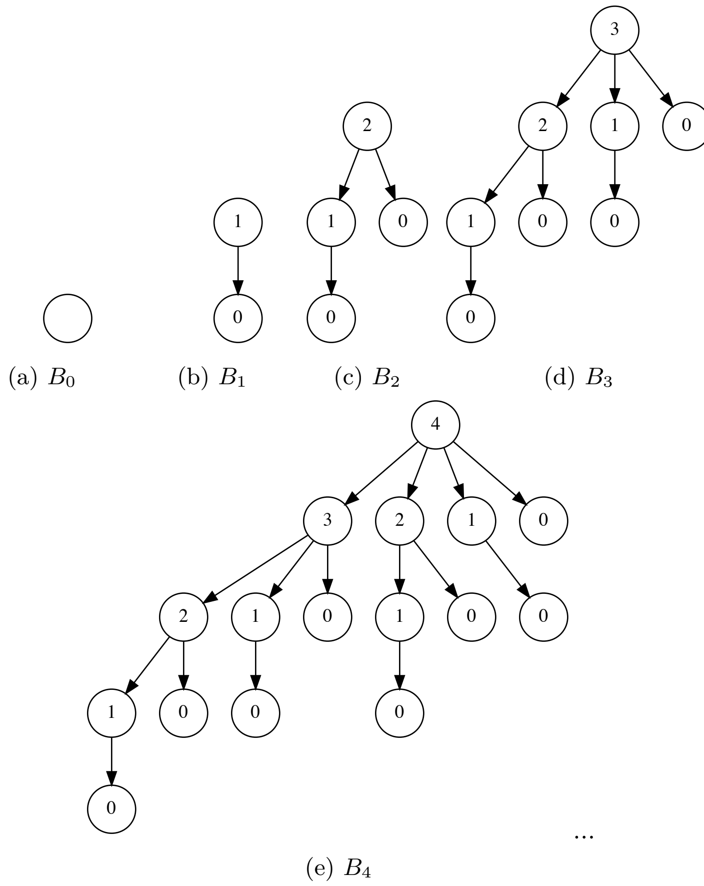
A binomial heap is a set of binomial trees (a forest) that satisfies the following two rules:

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, ...

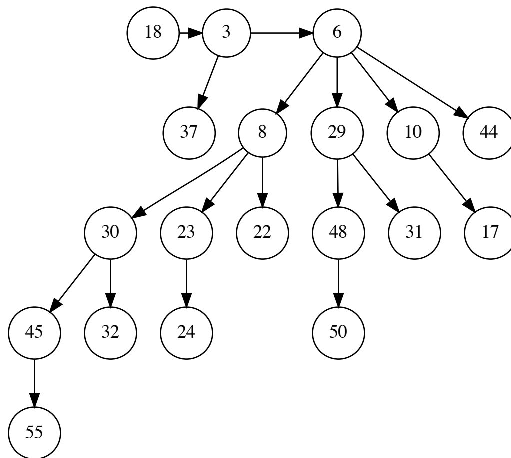



Figure 10.3: A binomial heap with 19 elements

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.

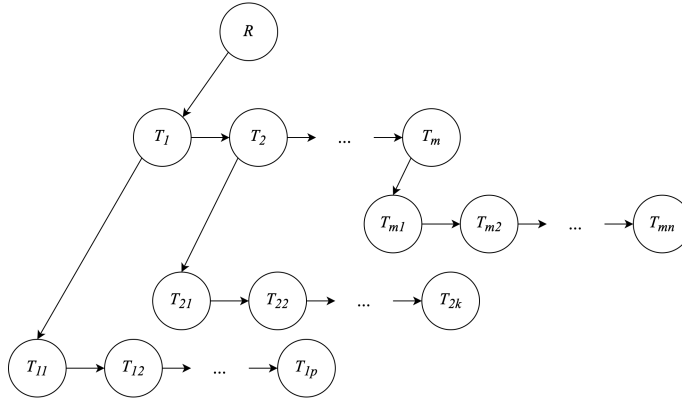



Figure 10.4: $R$ is the root, $T_1, T_2, ..., T_m$ are sub-trees of $R$. The left of $R$ is $T_1$, the right is NIL. $T_{11}, ..., T_{1p}$ are sub-trees of $T_1$. The left of $T_1$ is $T_{11}$, the right is its sibling $T_2$. The left of $T_2$ is $T_{21}$, the left is sibling.

### 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.

$$link\ (r, x, ts)\ (r, y, ts') = \begin{cases} x < y : & (r+1, x, (r, t, ts') : ts) \\ \text{otherwise} : & (r+1, y, (r, x, ts) : ts') \end{cases} \tag{10.2}$$

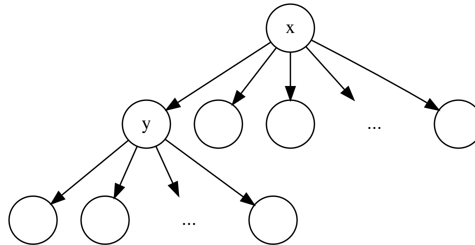



Figure 10.5: If $x < y$, link $y$ as the first sub-tree of $x$.

We can implement link with 'left child, right sibling' method as below. Link operation is bound to constant time.

1: **function** LINK$(x, y)$
2:     **if** KEY$(y) <$ KEY$(x)$ **then**
3:         Exchange $x \leftrightarrow y$
4:     SIBLING$(y) \leftarrow$ SUB-TREES$(T_1)$
5:     SUB-TREES$(x) \leftarrow y$
6:     PARENT$(y) \leftarrow x$

```
7:     RANK(x) ← RANK(y) + 1
8:     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:

```
1: function INSERT-TREE(T, H)
2:     ⊥← p ← NODE(0, NIL, NIL)
3:     while H ≠ NIL 且 RANK(H) ≤ RANK(T) do
4:         T1 ← H
5:         H ← SIBLING(H)
6:         if RANK(T) = RANK(T1) then
7:             T ← LINK(T, T1)
8:         else
9:             SIBLING(p) ← T1
10:            p ← T1
11:    SIBLING(p) ← T
12:    SIBLING(T) ← H
13:    return REMOVE-FIRST(⊥)
```

[2] It's similar to adding two binary numbers. A more generic topic is *numeric representation*[3].

```
14: function REMOVE-FIRST(H)
15:     n ← SIBLING(H)
16:     SIBLING(H) ← NIL
17:     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:

```
1: function MERGE(H1, H2)
2:     H ← p ← NODE(0, NIL, NIL)
3:     while H1 ≠ NIL and H2 ≠ NIL do
4:         if RANK(H1) < RANK(H2) then
5:             SIBLING(p) ← H1
6:             p ← SIBLING(p)
7:             H1 ← SIBLING(H1)
8:         else if RANK(H2) < RANK(H1) then
9:             SIBLING(p) ← H2
10:            p ← SIBLING(p)
11:            H2 ← SIBLING(H2)
12:        else                                          ▷ same rank
13:            T1 ← H1, T2 ← H2
14:            H1 ← SIBLING(H1), H2 ← SIBLING(H2)
15:            H1 ← INSERT-TREE(LINK(T1, T2), H1)
16:    if H1 ≠ NIL then
17:        SIBLING(p) ← H1
18:    if H2 ≠ NIL then
19:        SIBLING(p) ← H2
20:    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.

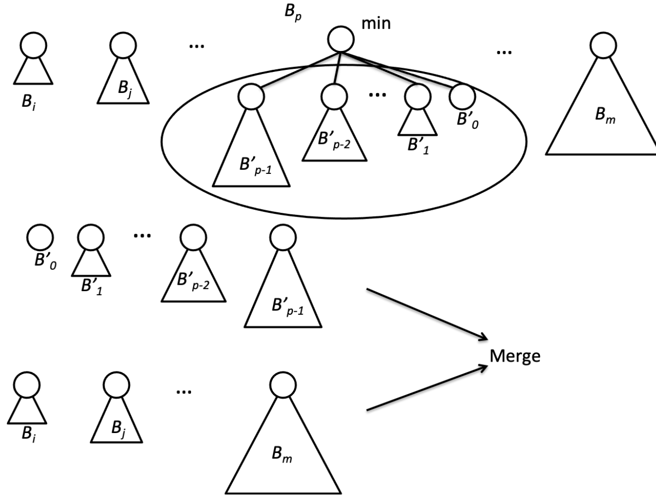



Figure 10.6: Binomial heap pop.

$$
\begin{aligned}
top\ (t:ts) &= foldr\ f\ (key\ t)\ ts \\
f\ (r, x, ts)\ y &= min\ x\ y
\end{aligned} \tag{10.7}
$$

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:

$$
\begin{array}{rcl}
min'\ [t] & = & (t, [\,]) \\
min'\ (t:ts) & = & \begin{cases} key\ t < key\ t' : & (t, ts), \text{其中}: (t', ts') = min'\ ts \\ \text{否则}: & (t', t:ts') \end{cases}
\end{array} \tag{10.8}
$$

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:

$$
pop\ H = (k, merge\ (reverse\ ts)\ H'), \text{其中}: ((r, k, ts), H') = min'\ H \tag{10.9}
$$

The iterative implementation is as below:

```
1: function Pop(H)
2:     (T_m, H) ← Extract-Min(H)
3:     H ← Merge(H, Reverse(Sub-Trees(T_m)))
4:     Sub-Trees(T_m)
5:     return (Key(T_m), H)
```

Where the list reverse is defined in chapter 1, EXTRACT-MIN is implemented as below:

```
1: function Extract-Min(H)
2:     H' ← H, p ← NIL
3:     T_m ← T_p ← NIL
4:     while H ≠ NIL do
5:         if T_m = NIL or Key(H) < Key(T_m) then
6:             T_m ← H
7:             T_p ← p
8:         p ← H
9:         H ← Sibling(H)
10:    if T_p ≠ NIL then
11:        Sibling(T_p) ← Sibling(T_m)
12:    else
13:        H' ← Sibling(T_m)
14:    Sibling(T_m) ← NIL
15:    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: $\quad x \leftarrow$ SINGLETON$(k)$ ▷ wrap $k$ to a tree
3: $\quad$ADD$(x$, TREES$(H))$
4: $\quad T_m \leftarrow$ MIN-TREE$(H)$
5: $\quad$**if** $T_m =$ NIL or $k <$ KEY$(T_m)$ **then**
6: $\quad\quad$MIN-TREE$(H) \leftarrow x$
7: $\quad$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
}
```

1: **function** MERGE($H_1, H_2$)
2: $H \leftarrow$ FIB-HEAP
3: TREES($H$) $\leftarrow$ CONCAT(TREES($H_1$), TREES($H_2$))
4: **if** KEY(MIN-TREE($H_1$)) $<$ KEY(MIN-TREE($H_2$)) **then**
5: MIN-TREE($H$) $\leftarrow$ MIN-TREE($H_1$)
6: **else**
7: MIN-TREE($H$) $\leftarrow$ MIN-TREE($H_2$)
SIZE($H$) = SIZE($H_1$) + SIZE($H_2$)
8: **return** $H$

9: **function** CONCAT($s_1, s_2$)
10: $e_1 \leftarrow$ PREV($s_1$)
11: $e_2 \leftarrow$ PREV($s_2$)
12: NEXT($e_1$) $\leftarrow s_2$
13: PREV($s_2$) $\leftarrow e_1$
14: NEXT($e_2$) $\leftarrow s_1$
15: PREV($s_1$) $\leftarrow e_2$
16: **return** $s_1$

### 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.

```
1: function CONSOLIDATE(H)
2:     R ← MAX-RANK(SIZE(H))
3:     A ← [NIL, NIL, ..., NIL]                    ▷ total R cells
4:     for each T in TREES(H) do
5:         r ← RANK(T)
6:         while A[r] ≠ NIL do
7:             T' ← A[r]
8:             T ← LINK(T, T')
9:             A[r] ← NIL
10:            r ← r + 1
11:        A[r] ← T
12:    T_m ← NIL
13:    TREES(H) ← NIL
14:    for each T in A do
15:        if T ≠ NIL then
16:            append T to TREES(H)
17:            if T_m = NIL or KEY(T) < KEY(T_m) then
18:                T_m ← T
19:    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.

$$\begin{array}{rcl}
pop\ (1, (0, x, [\,]), [\,]) & = & (x, [\,]) \\
pop\ (n, (r, x, ts_m), ts) & = & (x, (n - 1, t_m, ts'))
\end{array} \tag{10.17}$$

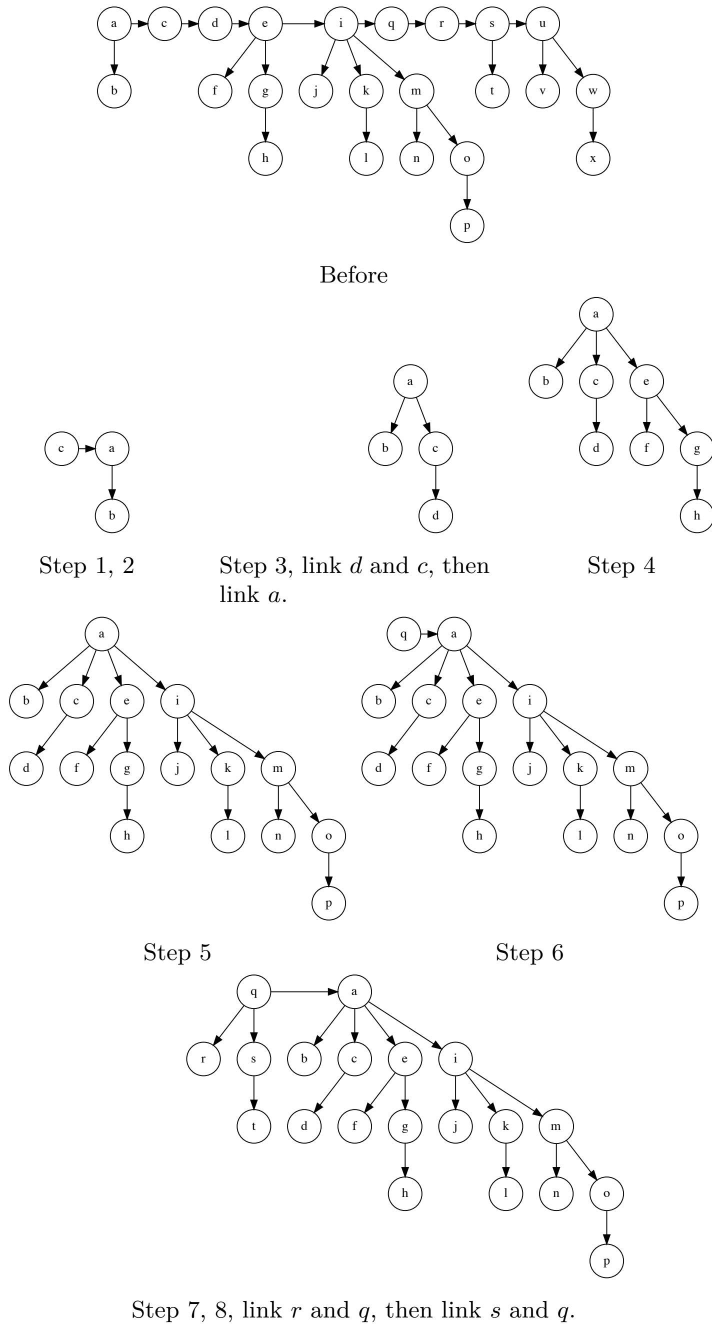



Figure 10.7: Consolidation

Where $(t_m, ts') = min'\ consolidate\ (ts_m + ts)$. It takes $O(|ts_m|)$ time for $+\!\!+$ to concatenate trees. The corresponding iterative implementation is as below:

1: **function** Pop($H$)
2: $T_m \leftarrow$ Min-Tree($H$)
3: **for** each $T$ in Sub-Trees($T_m$) **do**
4: append $T$ to Trees($H$)
5: Parent($T$) $\leftarrow$ NIL
6: remove $T_m$ from Trees($H$)
7: Size($H$) $\leftarrow$ Size($H$) - 1
8: Consolidate($H$)
9: **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:

$$W = mg(h' - h) + W_f$$

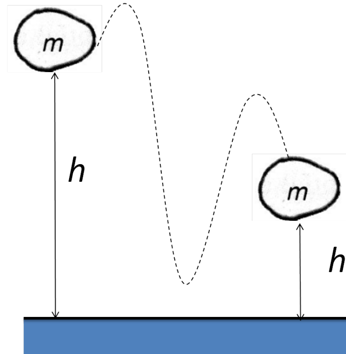



Figure 10.8: Gravity potential energy.

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).*

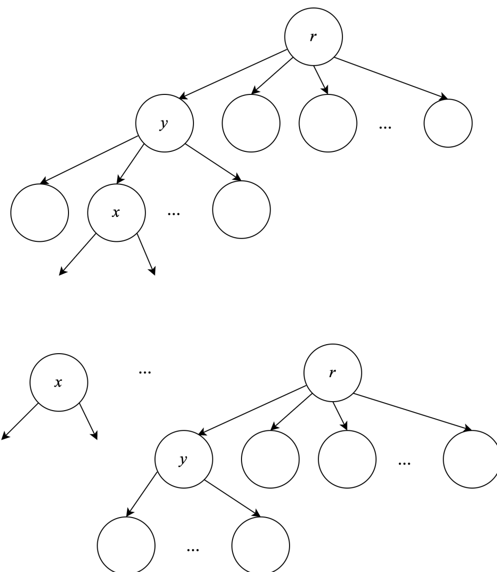



Figure 10.9: If $key\ x < key\ y$, cut $x$ off and add to the heap.

1: **function** DECREASE$(H, x, k)$
2: KEY$(x) \leftarrow k$
3: $p \leftarrow$ PARENT$(x)$
4: **if** $p \neq$ NIL and $k <$ KEY$(p)$ **then**
5: CUT$(H, x)$
6: CASCADE-CUT$(H, p)$
7: **if** $k <$ TOP$(H)$ **then**
8: 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.

1: **function** CUT$(H, x)$
2: $p \leftarrow$ PARENT$(x)$
3: remove $x$ from $p$
4: RANK$(p) \leftarrow$ RANK$(p)$ - 1
5: add $x$ to TREES$(H)$
6: PARENT$(x) \leftarrow$ NIL

7: 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.

```
1: function CASCADE-CUT(H, x)
2:     p ← PARENT(x)
3:     if p ≠ NIL then
4:         if MARK(x) = False then
5:             MARK(x) ← True
6:         else
7:             CUT(H, x)
8:             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 $\phi = \dfrac{1+\sqrt{5}}{2}$ 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:

```
1: function MAX-RANK(n)
2:     F_0 ← 0, F_1 ← 1
3:     k ← 2
4:     repeat
5:         F_k ← F_{k_1} + F_{k_2}
6:         k ← k + 1
7:     until F_k < n
8:     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.

1: **function** MERGE($H_1, H_2$)
2:     **if** $H_1$ = NIL **then**
3:         **return** $H_2$
4:     **if** $H_2$ = NIL **then**
5:         **return** $H_1$
6:     **if** KEY($H_2$) $<$ KEY($H_1$) **then**
7:         EXCHANGE($H_1 \leftrightarrow H_2$)
8:     SUB-TREES($H_1$) $\leftarrow$ LINK($H_2$, SUB-TREES($H_1$))
9:     PARENT($H_2$) $\leftarrow H_1$
10:     **return** $H_1$

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.

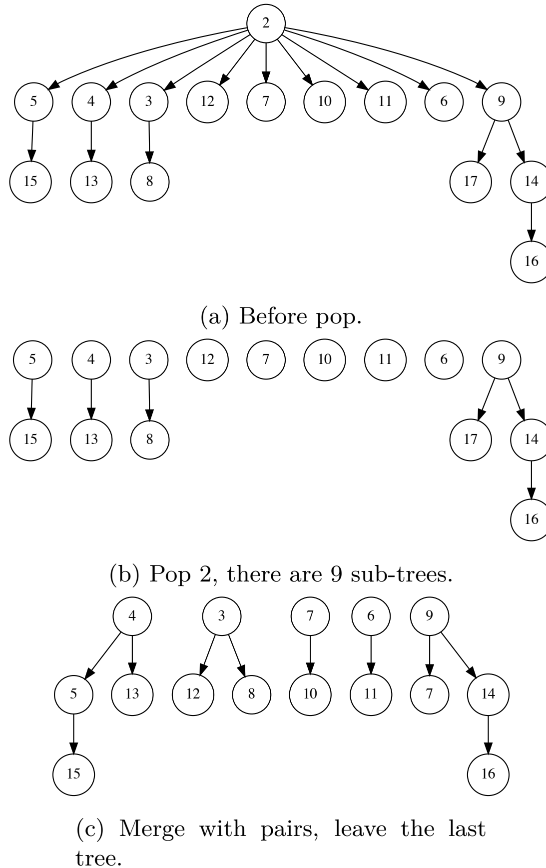
1: **function** DECREASE($H, x, k$)
2:     KEY($x$) $\leftarrow k$
3:     $p \leftarrow$ PARENT($x$)
4:     **if** $p \neq$ NIL **then**
5:         Remove $x$ from SUB-TREES($p$)
6:         PARENT($x$) $\leftarrow$ NIL
7:         **return** MERGE($H, x$)
8:     **return** $H$

### 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.

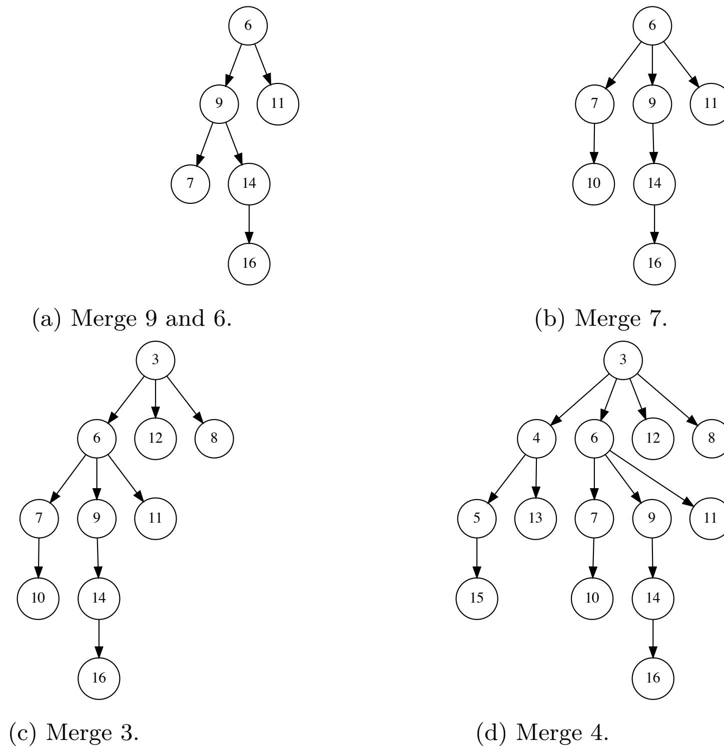
(c) Merge with pairs, leave the last tree.

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:

1: **function** POP($H$)
2: $\quad L \leftarrow$ NIL
3: $\quad$ **for** every $T_x$, $T_y$ in SUB-TREES($H$) **do**
4: $\quad\quad T \leftarrow$ MERGE($T_x, T_y$)
5: $\quad\quad L \leftarrow$ LINK($T, L$)
6: $\quad H \leftarrow$ NIL
7: $\quad$ **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.

8: $H \leftarrow$ MERGE$(H, T)$
9: **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 $H$:

1: **function** DELETE$(H, x)$
2: **if** $H = x$ **then**
3: POP$(H)$
4: **else**
5: $H \leftarrow$ CUT$(H, x)$
6: $x \leftarrow$ POP$(x)$
7: 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
}
```

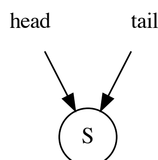



Figure 11.1: Both head and tail point to $S$ for empty queue.

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.

1: **function** ENQUEUE($Q, x$)
2: $p \leftarrow$ NODE($x$)
3: NEXT($p$) $\leftarrow$ NIL
4: NEXT(TAIL($Q$)) $\leftarrow p$
5: 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.

1: **function** DEQUEUE($Q$)
2: $x \leftarrow$ HEAD($Q$)
3: NEXT(HEAD($Q$)) $\leftarrow$ NEXT($x$)
4: **if** $x =$ TAIL($Q$) **then** ▷ $Q$ is empty
5: TAIL($Q$) $\leftarrow$ HEAD($Q$)
6: **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].

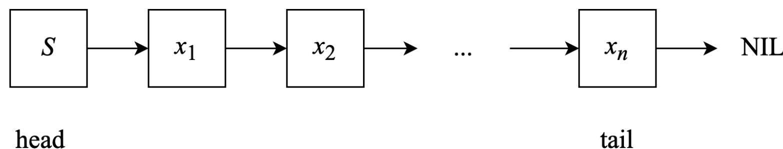



Figure 11.2: List with $S$ node.

## 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.

1: **function** ENQUEUE($Q, x$)
2: **if** not FULL($Q$) **then**
3: COUNT($Q$) $\leftarrow$ COUNT($Q$) + 1
4: tail $\leftarrow$ (HEAD($Q$) + COUNT($Q$)) mod SIZE($Q$)
5: BUF($Q$)[tail] $\leftarrow x$

1: **function** DEQUEUE($Q$)
2: $x \leftarrow$ NIL
3: **if** not EMPTY($Q$) **then**
4: $h \leftarrow$ HEAD($Q$)
5: $x \leftarrow$ BUF($Q$)[$h$]
6: HEAD($Q$) $\leftarrow$ (h + 1) mod SIZE($Q$)

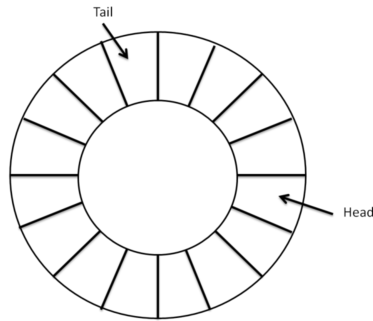
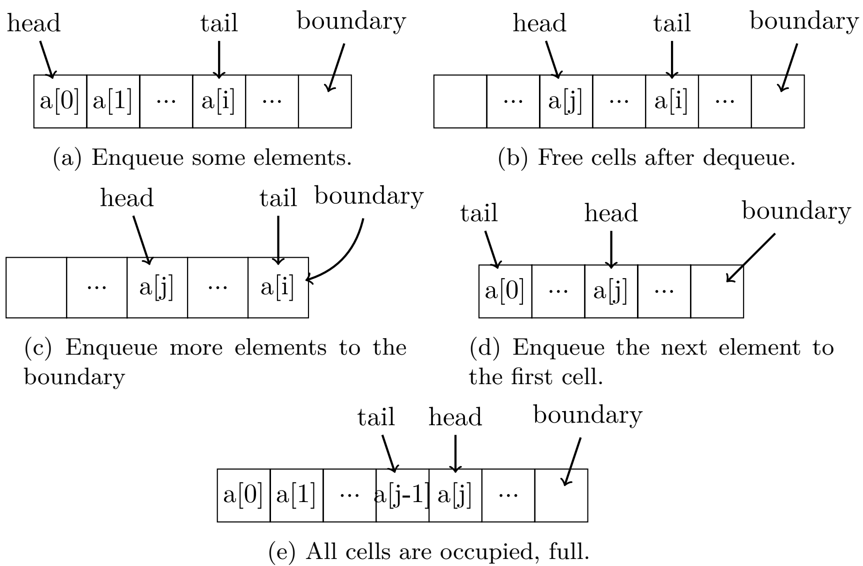



Figure 11.3: Circular buffer.




(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

7: $\text{COUNT}(Q) \leftarrow \text{COUNT}(Q)$ - 1
8: **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.

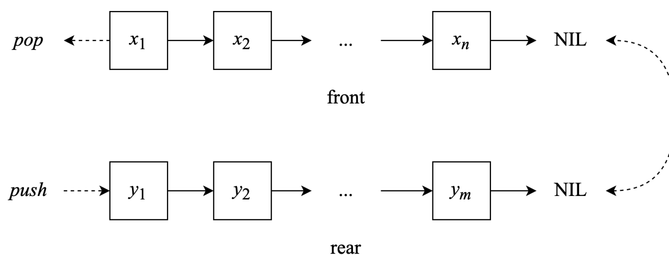



Figure 11.5: paired-list queue.

$$\begin{cases} push\ x\ (f, r) & = (f, x{:}r) \\ pop\ (x{:}f, r) & = (f, r) \end{cases} \tag{11.1}$$

$f$ 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

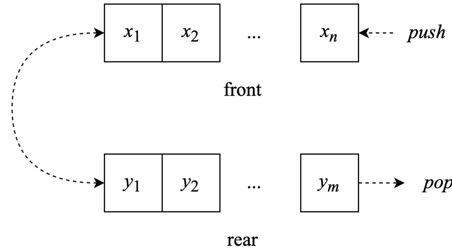



Figure 11.6: paired-array queue.

## 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:

$$\begin{array}{rcl} step \; S_r \; \text{“hello”} \; [\,] &=& (S_r, \text{“ello”}, \text{“h”}) \\ step \; S_r \; \text{“ello”} \; \text{“h”} &=& (S_r, \text{“llo”}, \text{“eh”}) \\ & \ldots & \\ step \; S_r \; \text{“o”} \; \text{“lleh”} &=& (S_r, [\,], \text{“olleh”}) \\ step \; S_r \; [\,] \; \text{“olleh”} &=& (S_f, \text{“olleh”}) \end{array}$$

We can next distribute the reverse steps to push/pop operations. However, it only solves half problem. We next need slow-down $\mathbin{+\!\!+}$ computation, which is more complex. We use state machine too. To concatenate $xs \mathbin{+\!\!+} ys$, 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 $reverse'$:

$$\begin{array}{rcl} xs \mathbin{+\!\!+} ys &=& (reverse \; reverse \; xs) \mathbin{+\!\!+} ys \\ &=& (reverse' \; [\,] \; (reverse \; xs)) \mathbin{+\!\!+} ys \\ &=& reverse' \; ys \; (reverse \; xs) \\ &=& reverse' \; ys \; \overleftarrow{xs} \end{array} \tag{11.11}$$

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 $f \mathbin{+\!\!+} reverse \; r$. 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:

$$\begin{array}{cll} & rotate\ (x{:}xs)\ (y{:}ys)\ a & \\ = & (x{:}xs) \mathbin{+\!\!+} (reverse\ (y{:}ys)) \mathbin{+\!\!+} a & \text{from (11.20)} \\ = & x : (xs \mathbin{+\!\!+} reverse\ (y{:}ys)) \mathbin{+\!\!+} a) & \text{concatenation is associative} \\ = & x : (xs \mathbin{+\!\!+} reverse\ ys \mathbin{+\!\!+} (y{:}a)) & \text{reverse property, and associative} \\ = & x : rotate\ xs\ ys\ (y{:}a) & \text{reverse of (11.20)} \end{array} \tag{11.22}$$

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.

$$
\begin{array}{rcll}
balance\ f\ r\ [\,] & = & (f', [\,], f') & \text{其中}: f' = rotate\ f\ r\ [\,] \\
balance\ f\ r\ (x{:}rot) & = & (f, r, rot) & \text{推进轮转}
\end{array}
\tag{11.25}
$$

### 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 $[x_1, x_2, x_3, x_4, x_5, x_6]$.

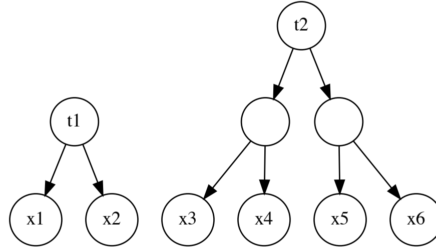



Figure 12.1: A sequence of 6 elements.

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.

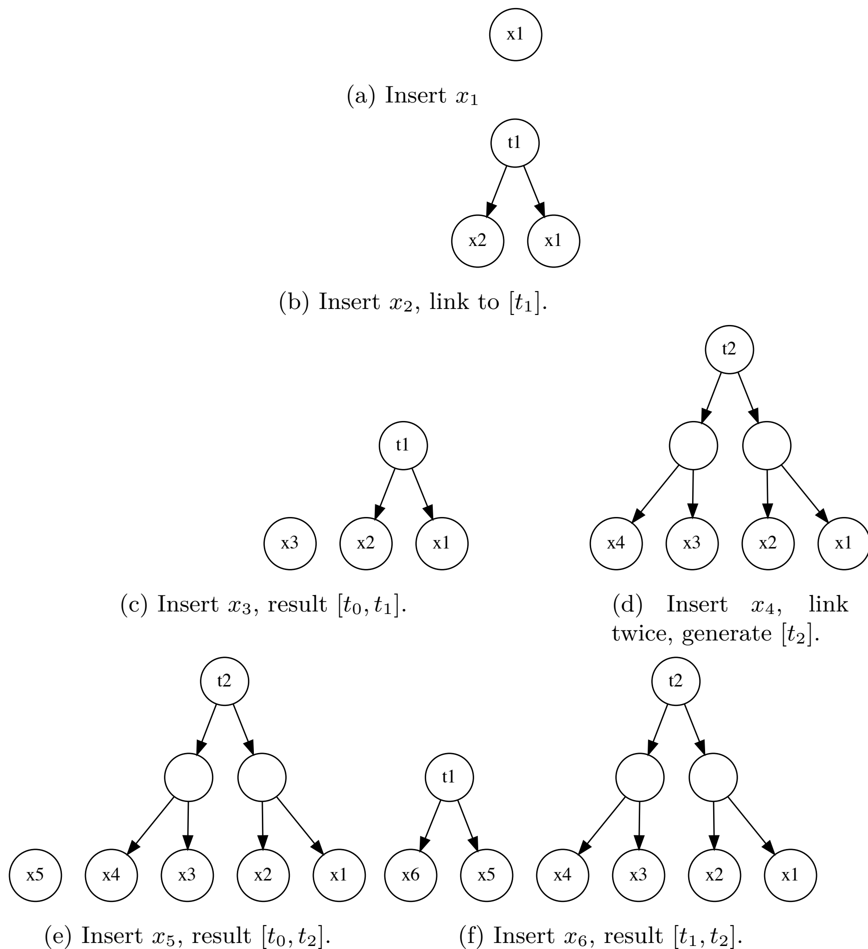
$$\begin{array}{rcl} insert_T\ t\ [\,] & = & [t] \\ insert_T\ t\ (t_1 {:} ts) & = & \begin{cases} size\ t < size\ t_1 : & t : t_1 : ts \\ \text{otherwise} : & insert_T\ (link\ t\ t_1)\ ts \end{cases} \end{array} \tag{12.4}$$

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.

$$\begin{array}{rcl} extract\ ((x){:}ts) & = & (x, ts) \\ extract\ ((s, t_1, t_2){:}ts) & = & extract\ (t_1{:}t_2{:}ts) \end{array} \tag{12.5}$$




(a) Insert $x_1$

(b) Insert $x_2$, link to $[t_1]$.

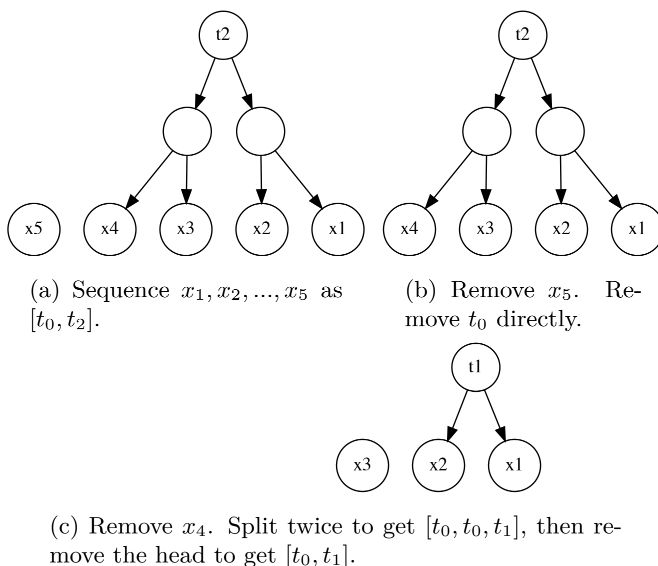
(c) Insert $x_3$, result $[t_0, t_1]$.

(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**

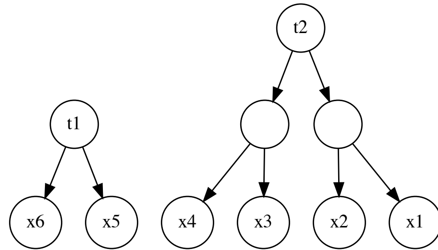
How to handle the out of bound exception?

## 12.3 Numeric representation

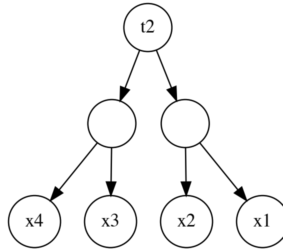
The binary form of $n = 2^0 e_0 + 2^1 e_1 + ... + 2^m e_m$ maps to the forest. The $e_i$ is the $i$-th bit. If $e_i = 1$, there is a full binary tree of size $2^i$. Adding an element corresponds to +1 to a binary number; while deleting corresponds to -1. We call such correspondence *numeric representation*[3]. 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.

$$
\begin{array}{rcl}
add\ t\ [\,] & = & [One\ t] \\
add\ t\ (Zero{:}ds) & = & (One\ t) : ds \\
add\ t\ (One\ t'{:}ds) & = & Zero : add\ (link\ t\ t')\ ds
\end{array}
\tag{12.9}
$$

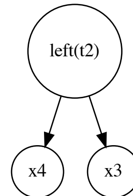
When add tree $t$, if the forest is empty, we create a state of *One t*, it's the only bit, corresponding to $0 + 1 = 1$. If the forest isn't empty, and the first bit is *Zero*, we



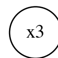

(a) $S[4], 4 > size(t_1) = 2$




(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:

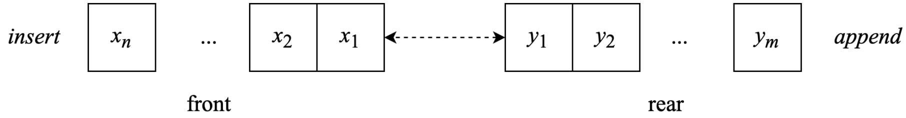



Figure 12.5: Paired-array sequence.

```
1: function INSERT(x, S)
2:     APPEND(x, FRONT(S))
3: function APPEND(x, S)
4:     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 $i - |f|$.

```
1: function GET(i, S)
2:     f, r ← FRONT(S), REAR(S)
3:     n ← SIZE(f)
4:     if i ≤ n then
5:         return f[n − i + 1]                          ▷ reversed
6:     else
7:         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.

```
1: function BALANCE(S)
2:     f ← FRONT(S), r ← REAR(S)
3:     n ← SIZE(f), m ← SIZE(r)
4:     if F = [ ] then
5:         k ← ⌊m/2⌋
6:         return (REVERSE(r[1...k]), r[(k + 1)...m])
7:     if R = [ ] then
8:         k ← ⌊n/2⌋
9:         return (f[(k + 1)...n], REVERSE(f[1...k]))
10:    return (f, r)
```

Every time when delete, we check $f$, $r$ and balance them:

```
1: function REMOVE-HEAD(S)
2:     BALANCE(S)
3:     f, r ← FRONT(S), REAR(S)
4:     if f = [ ] then                                  ▷ S = ([], [x])
5:         r ← [ ]
```

```
6:      else
7:          REMOVE-LAST(f)

8:  function REMOVE-TAIL(S)
9:      BALANCE(S)
10:     f, r ← FRONT(S), REAR(S)
11:     if r = [ ] then                          ▷ S = ([x], [])
12:         f ← [ ]
13:     else
14:         REMOVE-LAST(r)
```

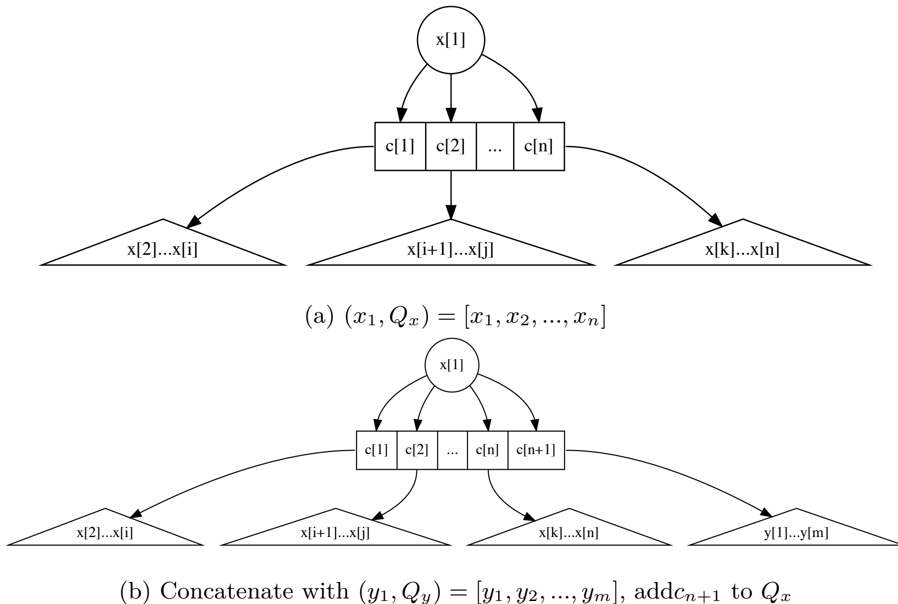
Due to reverse, the performance is $O(n)$ in the worst case, where $n$ is the number of elements, while it is amortized constant time.

### 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 $(t_1, t_2, t_3)$.

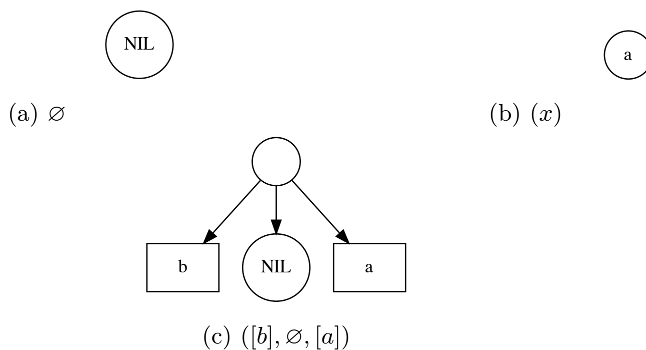
```
data Node a = Br2 a a | Br3 a a a
```

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

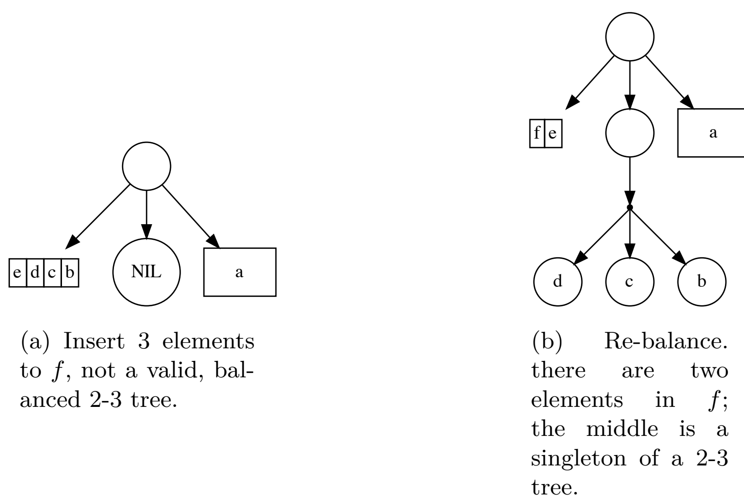



(a) $\varnothing$

(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 $\leq 3$ 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

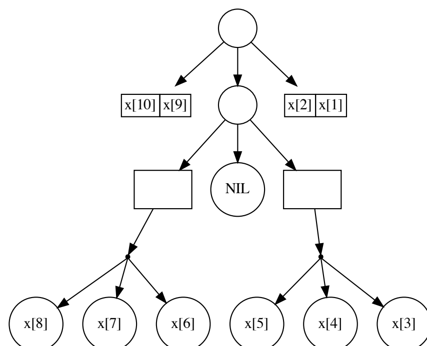
We implement extract as the reverse of *insert*.

$$
\begin{array}{rcl}
extract\ (a) & = & (a, \varnothing) \\
extract\ ([a], \varnothing, [b]) & = & (a, (b)) \\
extract\ ([a], \varnothing, b{:}bs) & = & (a, ([b], \varnothing, bs)) \\
extract\ ([a], t, r) & = & (a, (toList\ f, t', r)), \text{where} : (f, t') = extract\ t \\
extract\ (a{:}as, t, r) & = & (a, (as, t, r))
\end{array}
\tag{12.20}
$$

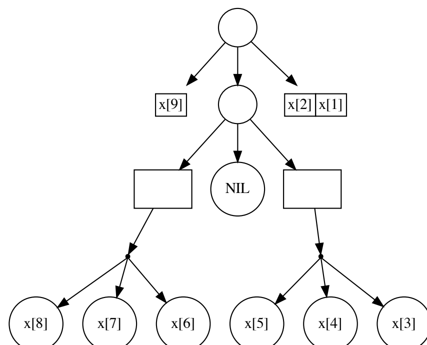
Where *toList* flatten a 2-3 tree to list:

$$
\begin{array}{rcl}
toList\ (a, b) & = & [a, b] \\
toList\ (a, b, c) & = & [a, b, c]
\end{array}
\tag{12.21}
$$

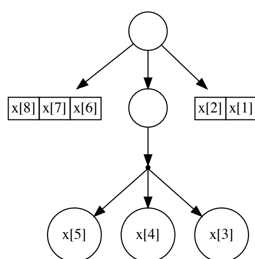
We skip error handling (e.g., extract from empty tree). If the tree is a singleton leaf, the result is empty; if there are two elements, the result is a singleton; if $f$ is a singleton list, the middle is empty, while $r$ isn't empty, we extract the only one in $f$, then borrow one from $r$ to $f$; 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*:

$$(f_1, t_1, r_1) \mathbin{+\!\!+} (f_2, t_2, r_2) = (f_1, merge\ t_1\ (r_1 \mathbin{+\!\!+} f_2)\ t_2, r_2) \tag{12.28}$$

Compare with (12.26), concatenation is essentially merge, we can define them in a unified way:

$$T_1 \mathbin{+\!\!+} T_2 = merge\ T_1\ [\,]\ T_2 \tag{12.29}$$

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:

$$\begin{array}{rcl}
lookup_T\ i\ (x) & = & (i, x) \\
lookup_T\ i\ (s, f, t, r) & = & \begin{cases} i < s_f : & lookup_s\ i\ f \\ s_f \le i < s_f + s_t : & lookup_N\ (lookup_T\ (i - s_f)\ t) \\ \text{otherwise} : & lookup_s\ (i - s_f - s_t)\ r \end{cases}
\end{array} \tag{12.34}$$

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 $m$:

$$\begin{array}{rcl}
lookup_N\ i\ (t_1, t_2) & = & \begin{cases} i < size\ t_1 : & (i, t_1) \\ \text{otherwise} : & (i - size\ t_1, t_2) \end{cases} \\
lookup_N\ i\ (t_1, t_2, t_3) & = & \begin{cases} i < size\ t_1 : & (i, t_1) \\ size\ t_1 \le i < size\ t_1 + size\ t_2 : & (i - size\ t_1, t_2) \\ \text{otherwise} : & (i - size\ t_1 - size\ t_2, t_3) \end{cases}
\end{array} \tag{12.36}$$

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
```

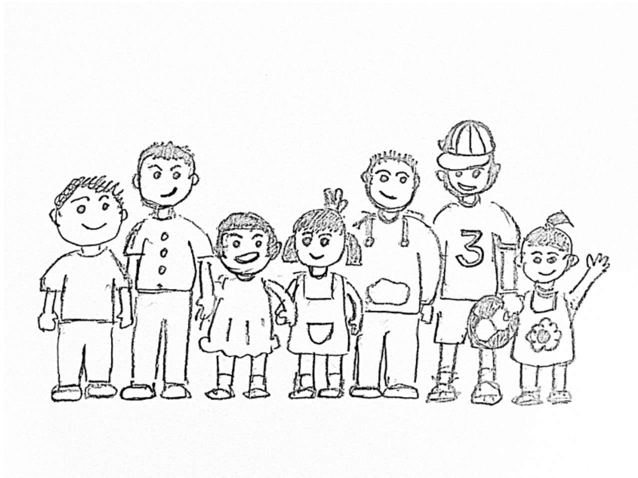
# Chapter 13

# 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.

$$
\begin{array}{rcl}
sort\ [\,] & = & [\,] \\
sort\ (x{:}xs) & = & sort\ [y | y \in xs, y \leq x] \mathbin{+\!\!+} [x] \mathbin{+\!\!+} sort\ [y | y \in xs, x < y]
\end{array}
\tag{13.1}
$$

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:

$$
\begin{array}{rcl}
sort\ [\,] & = & [\,] \\
sort\ (x{:}xs) & = & sort\ as \mathbin{+\!\!+} [x] \mathbin{+\!\!+} sort\ bs, \text{where} : (as, bs) = part\ (\leq x)\ xs
\end{array}
\tag{13.3}
$$

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:

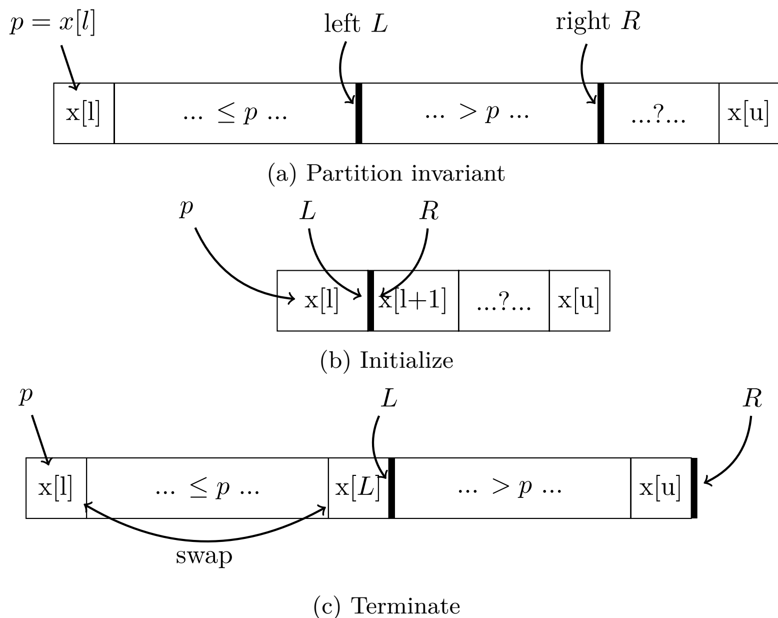



Figure 13.1: In-place partition, pivot $p = x[l]$

1: **function** PARTITION(A, l, u)
2: $p \leftarrow A[l]$ ▷ pivot
3: $L \leftarrow l$ ▷ left
4: **for** $R$ in $[l+1, u]$ **do** ▷ iterate right
5: **if** $p \geq A[R]$ **then**
6: $L \leftarrow L + 1$
7: EXCHANGE $A[L] \leftrightarrow A[R]$
8: EXCHANGE $A[L] \leftrightarrow p$
9: **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:

1: **procedure** QUICK-SORT$(A, l, u)$
2: **if** $l < u$ **then**
3: $m \leftarrow$ PARTITION$(A, l, u)$

4: $\quad$ QUICK-SORT$(A, l, m-1)$
5: $\quad$ 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 $O(n \lg n)$.

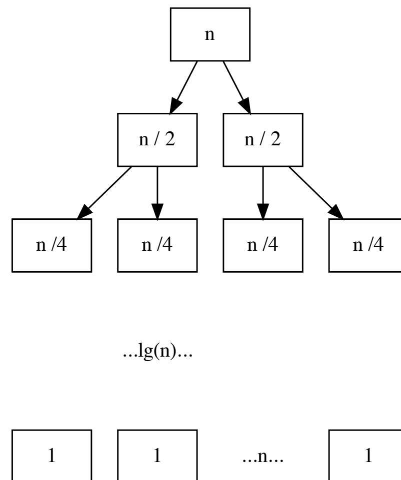



Figure 13.2: The best case, halve every time.

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:

$$
\begin{array}{rcl}
part\ S\ E\ G\ x\ [\,] & = & (S, E, G) \\
part\ S\ E\ G\ x\ (y{:}ys) & = & \begin{cases}
y < x : & (y{:}S, E, G) \\
y = x : & (S, y{:}E, G) \\
y > x : & (S, E, y{:}G)
\end{cases}
\end{array}
\tag{13.23}
$$

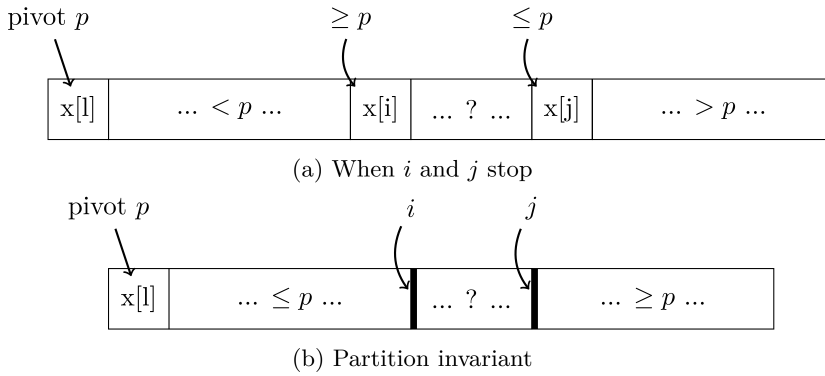
Richard Bird developed another improvement[1], instead concatenate the recursive sort results, put them in a list and concatenate finally:

```
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:         $i \leftarrow l$, $j \leftarrow u$
4:         $pivot \leftarrow A[l]$
5:         **loop**



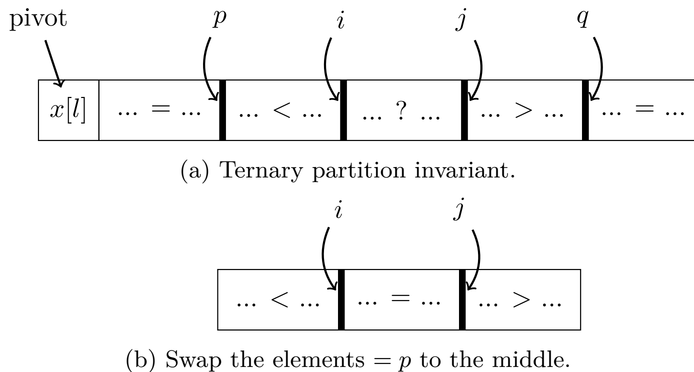

(a) When $i$ and $j$ stop

(b) Partition invariant

Figure 13.3: 2-way scan

```
6:          repeat
7:              i ← i + 1
8:          until A[i] ≥ pivot                          ▷ Ignore i ≥ u
9:          repeat
10:             j ← j − 1
11:         until A[j] ≤ pivot                          ▷ Ignore j < l
12:         if j < i then
13:             break
14:         Exchange A[i] ↔ A[j]
15:     Exchange A[l] ↔ A[j]                            ▷ Move the pivot
16:     Sort(A, l, j)
17:     Sort(A, i, u)
```

Consider the 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.

```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
3:         i ← l, j ← u
4:         p ← l, q ← u          ▷ point to the boundaries of duplicated elements
5:         pivot ← A[l]
6:         loop
7:             repeat
8:                 i ← i + 1
9:             until A[i] ≥ pivot                          ▷ Ignore i ≥ u case
10:            repeat
11:                j ← j − 1
12:            until A[j] ≤ pivot                          ▷ Ignore j < l case
13:            if j ≤ i then
14:                break
15:            EXCHANGE A[i] ↔ A[j]
16:            if A[i] = pivot then                        ▷ duplicated element
17:                p ← p + 1
18:                EXCHANGE A[p] ↔ A[i]
19:            if A[j] = pivot then
20:                q ← q − 1
21:                EXCHANGE A[q] ↔ A[j]
22:        if i = j and A[i] = pivot then
23:            j ← j − 1, i ← i + 1
24:        for k from l to p do       ▷ Swap the duplicated elements to the middle
25:            EXCHANGE A[k] ↔ A[j]
26:            j ← j − 1
27:        for k from u − 1 down-to q do
28:            EXCHANGE A[k] ↔ A[i]
29:            i ← i + 1
30:        SORT(A, l, j + 1)
31:        SORT(A, i, u)
```

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.

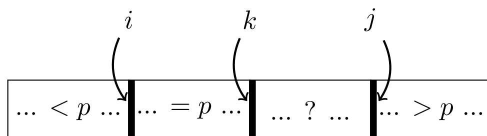



Figure 13.5: 1 way scan ternary partition

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 $j$.

```
 1: procedure Sort(A, l, u)
 2:     if u − l > 1 then
 3:         i ← l, j ← u, k ← l + 1
 4:         pivot ← A[i]
 5:         while k < j do
 6:             while pivot < A[k] do
 7:                 j ← j − 1
 8:                 Exchange A[k] ↔ A[j]
 9:             if A[k] < pivot then
10:                 Exchange A[k] ↔ A[i]
11:                 i ← i + 1
12:             k ← k + 1
13:         Sort(A, l, i)
14:         Sort(A, j, u)
```

Compare with the ternary partition through 2-way scan, this implementation is less complex but need more swaps.

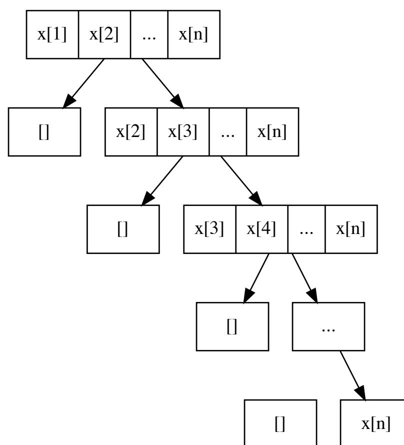
### Worst cases

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.

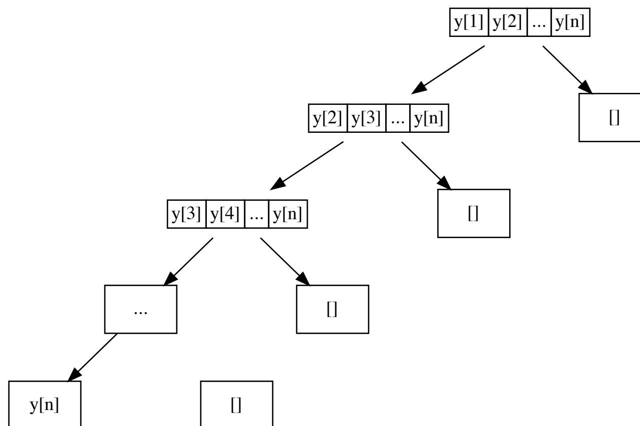
In these worst cases, the partition is unbalanced when choose the first element as the pivot. Robert Sedgwick improved the pivot selection[69]: Instead pick a fixed position, sample several elements to avoid bad pivot. We sample the first, the middle, and the last, pick the median as the pivot. We can 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.

```
 1: procedure Sort(A, l, u)
 2:     if u − l > 1 then
 3:         m ← ⌊(l + u)/2⌋                      ▷ or l + (u − l)/2 to void overflow
 4:         if A[m] < A[l] then                  ▷ Ensure A[l] ≤ A[m]
 5:             Exchange A[l] ↔ A[m]
 6:         if A[u − 1] < A[l] then              ▷ Ensure A[l] ≤ A[u − 1]
 7:             Exchange A[l] ↔ A[u − 1]
 8:         if A[u − 1] < A[m] then              ▷ Ensure A[m] ≤ A[u − 1]
 9:             Exchange A[m] ↔ A[u − 1]
10:         Exchange A[l] ↔ A[m]
11:         (i, j) ← Partition(A, l, u)
12:         Sort(A, l, i)
13:         Sort(A, j, u)
```

This implementation handles the above four worst cases well. We call it 'median of three'. Alternatively, we can randomly pick pivot:

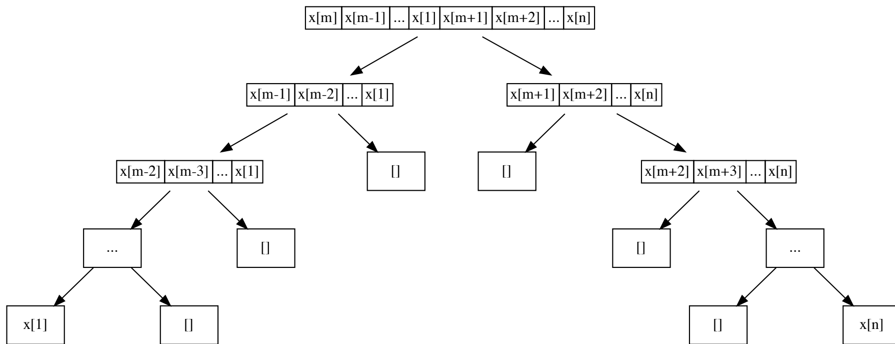



(a) Partition tree of $[x_1 < x_2 < ... < x_n]$, the sub-trees of $\leq p$ are empty.

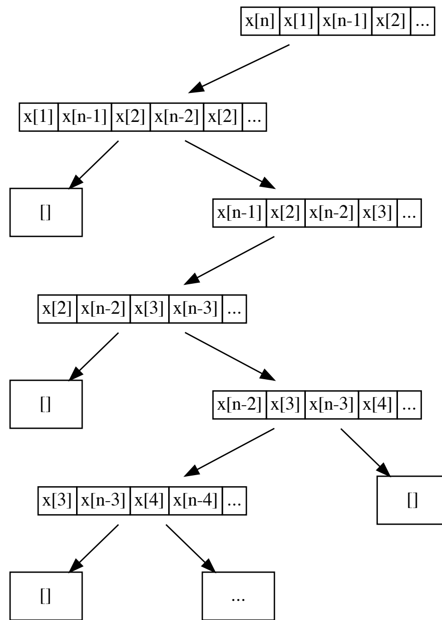



(b) Partition tree of $[y_1 > y_2 > ... > y_n]$, the sub-trees of $\geq p$ are empty.

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.

1: **procedure** Sort$(A, l, u)$
2: **if** $u - l > 1$ **then**
3: Exchange $A[l] \leftrightarrow A[$ Random$(l, u)$ $]$
4: $(i, j) \leftarrow$ Partition$(A, l, u)$
5: Sort$(A, l, i)$
6: 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.

1: **procedure** Sort$(A, l, u)$
2: **if** $u - l >$ Cut-Off **then**
3: Quick-Sort$(A, l, u)$
4: **else**
5: Insertion-Sort$(A, l, u)$

### 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.

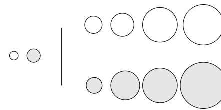


Figure 13.8: Merge

$$
\begin{array}{rcl}
merge\ [\ ]\ bs & = & bs \\
merge\ as\ [\ ] & = & as \\
merge\ (a{:}as)\ (b{:}bs) & = & \begin{cases} a < b: & a : merge\ as\ (b{:}bs) \\ \text{otherwise}: & b : merge\ (a{:}as)\ bs \end{cases}
\end{array}
\tag{13.31}
$$

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.

```
1: procedure Sort(A)
2:     n ← |A|
3:     if n > 1 then
4:         m ← ⌊n/2⌋
5:         X ← Copy-Array(A[1...m])
6:         Y ← Copy-Array(A[m + 1...n])
7:         Sort(X)
8:         Sort(Y)
9:         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 $A$.

```
1: procedure Merge(A, X, Y)
2:     i ← 1, j ← 1, k ← 1
3:     m ← |X|, n ← |Y|
4:     while i ≤ m and j ≤ n do
5:         if X[i] < Y[j] then
6:             A[k] ← X[i]
7:             i ← i + 1
8:         else
9:             A[k] ← Y[j]
10:            j ← j + 1
11:        k ← k + 1
12:    while i ≤ m do
13:        A[k] ← X[i]
14:        k ← k + 1
15:        i ← i + 1
16:    while j ≤ n do
17:        A[k] ← Y[j]
18:        k ← k + 1
19:        j ← j + 1
```

### 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 $\infty$ to $X$ and $Y$[3].

```
1: procedure Merge(A, X, Y)
2:     Append(X, ∞)
3:     Append(Y, ∞)
4:     i ← 1, j ← 1, n ← |A|
5:     for k ← from 1 to n do
6:         if X[i] < Y[j] then
7:             A[k] ← X[i]
8:             i ← i + 1
9:         else
10:            A[k] ← Y[j]
11:            j ← j + 1
```

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.

```
1: procedure Sort(A)
2:     n ← |A|
3:     Sort'(A, Create-Array(n), 1, n)

4: procedure Sort'(A, B, l, u)
5:     if u − l > 0 then
6:         m ← ⌊(l + u)/2⌋
7:         Sort'(A, B, l, m)
8:         Sort'(A, B, m + 1, u)
9:         Merge'(A, B, l, m, u)
```

We need update Merge′ with the passed in work area:

```
1: procedure Merge'(A, B, l, m, u)
2:     i ← l, j ← m + 1, k ← l
3:     while i ≤ m and j ≤ u do
4:         if A[i] < A[j] then
5:             B[k] ← A[i]
6:             i ← i + 1
7:         else
8:             B[k] ← A[j]
9:             j ← j + 1
10:        k ← k + 1
11:    while i ≤ m do
12:        B[k] ← A[i]
13:        k ← k + 1
14:        i ← i + 1
15:    while j ≤ u do
16:        B[k] ← A[j]
17:        k ← k + 1
18:        j ← j + 1
19:    for i ← from l to u do                ▷ copy back
20:        A[i] ← B[i]
```

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.

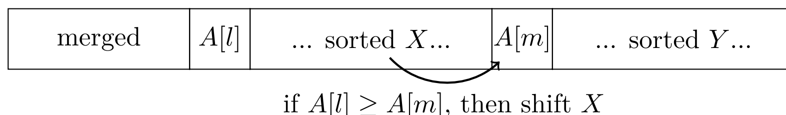



Figure 13.9: In-place shift and merge

```
1: procedure MERGE(A, l, m, u)
2:     while l ≤ m ∧ m ≤ u do
3:         if A[l] < A[m] then
4:             l ← l + 1
5:         else
6:             x ← A[m]
7:             for i ← m down-to l + 1 do                ▷ Shift
8:                 A[i] ← A[i − 1]
9:             A[l] ← x
```

However, 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.

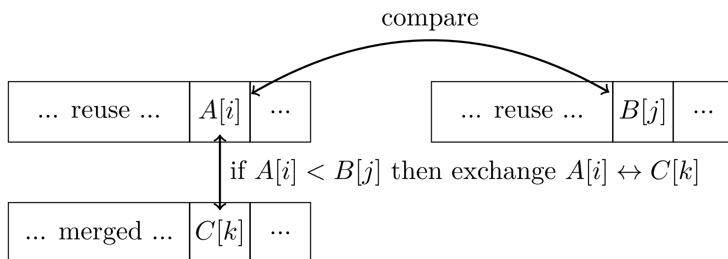



Figure 13.10: Merge and swap

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 $k$.

```
1: procedure MERGE(A, [i, m), [j, n), k)
2:     while i < m and j < n do
3:         if A[i] < A[j] then
```

[4]range $[a, b)$ includes $a$, but excludes $b$.

```
4:          EXCHANGE A[k] ↔ A[i]
5:          i ← i + 1
6:      else
7:          EXCHANGE A[k] ↔ A[j]
8:          j ← j + 1
9:      k ← k + 1
10:  while i < m do
11:      EXCHANGE A[k] ↔ A[i]
12:      i ← i + 1
13:      k ← k + 1
14:  while j < m do
15:      EXCHANGE A[k] ↔ A[j]
16:      j ← j + 1
17:      k ← k + 1
```

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.

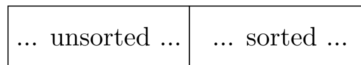



Figure 13.11: Merge and sort half array

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 $A + B$.

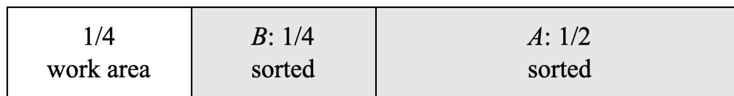



Figure 13.12: Work area can't support merge $A$ and $B$.

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).

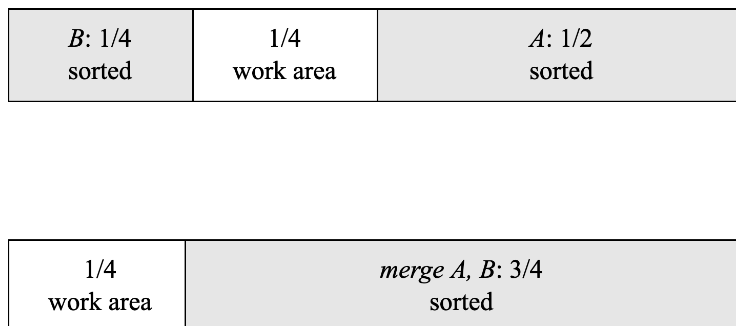



Figure 13.13: Merge $A$ and $B$ with the work area.

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.

1: **procedure** Sort($A, l, u$)
2: **if** $u - l > 0$ **then**
3: $m \leftarrow \lfloor \frac{l + u}{2} \rfloor$
4: $w \leftarrow l + u - m$
5: Sort'($A, l, m, w$) ▷ sort half
6: **while** $w - l > 1$ **do**
7: $u' \leftarrow w$
8: $w \leftarrow \lceil \frac{l + u'}{2} \rceil$ ▷ halve the work area
9: Sort'($A, w, u', l$) ▷ sort the remaining half
10: Merge($A, [l, l + u' - w], [u', u], w$)
11: **for** $i \leftarrow w$ down-to $l$ **do** ▷ Switch to insert sort
12: $j \leftarrow i$
13: **while** $j \leq u$ and $A[j] < A[j-1]$ **do**
14: Exchange $A[j] \leftrightarrow A[j-1]$
15: $j \leftarrow j + 1$

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.

1: **procedure** Sort'($A, l, u, w$)
2: **if** $u - l > 0$ **then**
3: $m \leftarrow \lfloor \frac{l + u}{2} \rfloor$
4: Sort($A, l, m$)
5: Sort($A, m + 1, u$)
6: Merge($A, [l, m), [m + 1, u), w$)
7: **else** ▷ Swap elements to the work area
8: **while** $l \leq u$ **do**
9: Exchange $A[l] \leftrightarrow A[w]$
10: $l \leftarrow l + 1$
11: $w \leftarrow w + 1$

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

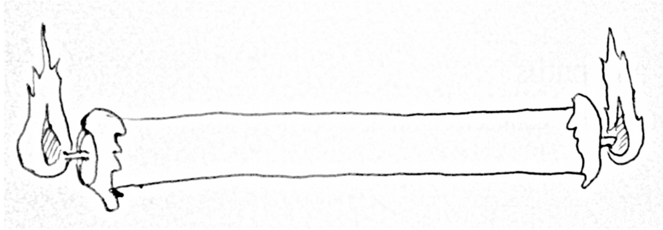


Figure 13.14: Burn from both ends

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.

$\underline{15}$, 0, 4, 3, 5, 2, 7, 1, 12, 14, 13, 8, 9, 6, 10, 11
$\underline{8, 12, 14}$, 0, 1, 4, 11, 2, 3, 5, 9, 13, 10, 6, 15, 7
$\underline{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}$

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

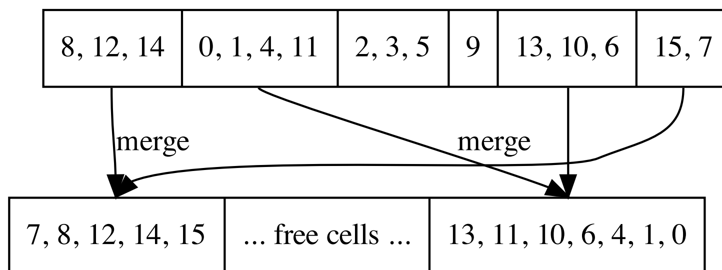



Figure 13.15: Nature merge sort

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.

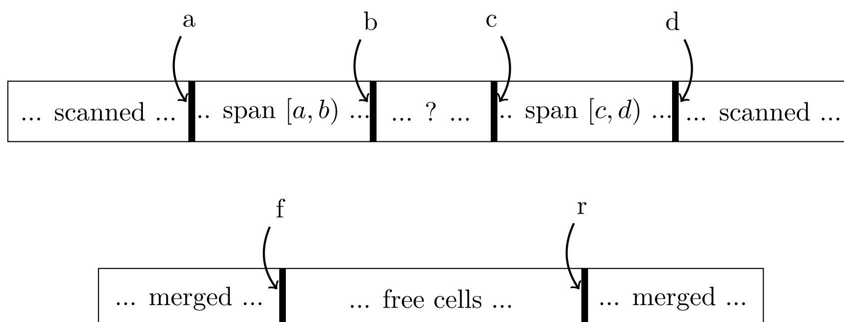



Figure 13.16: A status of nature merge sort

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.

1: **function** SORT($A$)
2:     **if** $|A| > 1$ **then**
3:         $n \leftarrow |A|$
4:         $B \leftarrow$ CREATE-ARRAY($n$) ▷ the work area
5:         **loop**
6:             $[a, b) \leftarrow [1, 1)$
7:             $[c, d) \leftarrow [n + 1, n + 1)$
8:             $f \leftarrow 1, r \leftarrow n$ ▷ front, rear of the work area
9:             $t \leftarrow 1$ ▷ even/odd round
10:             **while** $b < c$ **do** ▷ elements yet to scan
11:                 **repeat** ▷ Span $[a, b)$

```
12:                 b ← b + 1
13:             until b ≥ c ∨ A[b] < A[b − 1]
14:             repeat                                          ▷ Span [c, d)
15:                 c ← c − 1
16:             until c ≤ b ∨ A[c − 1] < A[c]
17:             if c < b then                                   ▷ Avoid overlap
18:                 c ← b
19:             if b − a ≥ n then         ▷ Terminates if [a, b) spans the whole array
20:                 return A
21:             if t is odd then                                ▷ merge to front
22:                 f ← Merge(A, [a, b), [c, d), B, f, 1)
23:             else                                            ▷ merge to rear
24:                 r ← Merge(A, [a, b), [c, d), B, r, −1)
25:             a ← b, d ← c
26:             t ← t + 1
27:         Exchange A ↔ B                                      ▷ Switch work area
28:     return A
```

We need pass the merge direction in:

```
1: function Merge(A, [a, b), [c, d), B, w, Δ)
2:     while a < b and c < d do
3:         if A[a] < A[d − 1] then
4:             B[w] ← A[a]
5:             a ← a + 1
6:         else
7:             B[w] ← A[d − 1]
8:             d ← d − 1
9:         w ← w + Δ
10:    while a < b do
11:        B[w] ← A[a]
12:        a ← a + 1
13:        w ← w + Δ
14:    while c < d do
15:        B[w] ← A[d − 1]
16:        d ← d − 1
17:        w ← w + Δ
18:    return w
```

The 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.

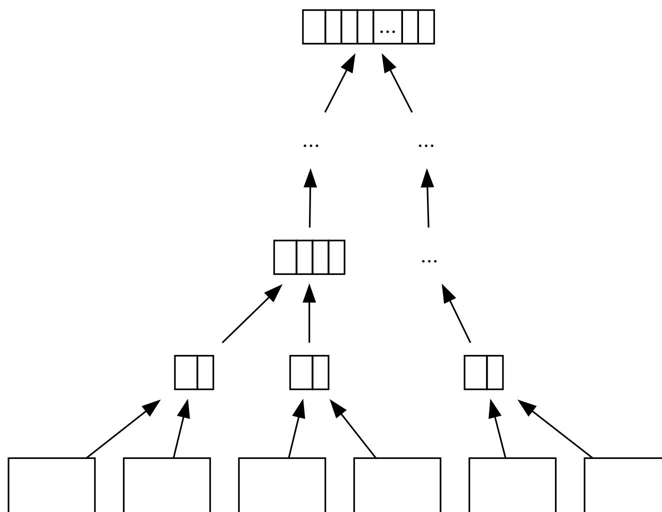



Figure 13.17: Bottom-up merge sort

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.

1: **function** SORT($A$)
2: $n \leftarrow |A|$
3: $B \leftarrow$ CREATE-ARRAY($n$)
4: **for** $i$ from 1 to $n$ **do**
5: $B[i] = [A[i]]$
6: **while** $n > 1$ **do**
7: **for** $i \leftarrow$ from 1 to $\lfloor \frac{n}{2} \rfloor$ **do**
8: $B[i] \leftarrow$ MERGE($B[2i-1], B[2i]$)
9: **if** ODD($n$) **then**
10: $B[\lceil \frac{n}{2} \rceil] \leftarrow B[n]$
11: $n \leftarrow \lceil \frac{n}{2} \rceil$
12: **if** $B = [\ ]$ **then**
13: **return** $[\ ]$
14: **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, Range<Int> (i, m), Range<Int> (j, 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, Range<Int> (l, 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

# Searching

## 14.1 Introduction

Searching is quite a big and important area. Computer makes many hard searching problems realistic. They are almost impossible for human beings. A modern industry robot can even search and pick the correct gadget from the pipeline for assembly; A GPS car navigator can search among the map, for the best route to a specific place. The modern mobile phone is not only equipped with such map navigator, but it can also search for the best price for Internet shopping.

This chapter just scratches the surface of elementary searching. One good thing that computer offers is the brute-force scanning for a certain result in a large sequence. The divide and conquer search strategy will be briefed with two problems, one is to find the $k$-th big one among a list of unsorted elements; the other is the popular binary search among a list of sorted elements. We'll also introduce the extension of binary search for multiple-dimension data.

Text matching is also very important in our daily life, two well-known searching algorithms, Knuth-Morris-Pratt (KMP) and Boyer-Moore algorithms will be introduced. They set good examples for another searching strategy: information reusing.

Besides sequence search, some elementary methods for searching solution for some interesting problems will be introduced. They were mostly well studied in the early phase of AI (artificial intelligence), including the basic DFS (Depth first search), and BFS (Breadth first search).

Finally, Dynamic programming will be briefed for searching optimal solutions, and we'll also introduce about greedy algorithm which is applicable for some special cases.

All algorithms will be realized in both imperative and functional approaches.

## 14.2 Sequence search

Although modern computer offers fast speed for brute-force searching, and even if the Moore's law could be strictly followed, the grows of huge data is too fast to be handled well in this way. We've seen a vivid example in the introduction chapter of this book. It's why people study the computer search algorithms.

### 14.2.1 Divide and conquer search

One solution is to use divide and conquer approach. That if we can repeatedly scale down the search domain, the data being dropped needn't be examined at all. This will

definitely speed up the search.

### $k$-selection problem

Consider a problem of finding the $k$-th smallest one among $n$ elements. The most straightforward idea is to find the minimum first, then drop it and find the second minimum element among the rest. Repeat this minimum finding and dropping $k$ steps will give the $k$-th smallest one. Finding the minimum among $n$ elements costs linear $O(n)$ time. Thus this method performs $O(kn)$ time, if $k$ is much smaller than $n$.

Another method is to use the 'heap' data structure we've introduced. No matter what concrete heap is used, e.g. binary heap with implicit array, Fibonacci heap or others, Accessing the top element followed by popping is typically bound $O(\lg n)$ time. Thus this method, as formalized in equation (14.1) and (14.2) performs in $O(k \lg n)$ time, if $k$ is much smaller than $n$.

$$top(k, L) = find(k, heapify(L)) \tag{14.1}$$

$$find(k, H) = \begin{cases} top(H) & : \quad k = 0 \\ find(k-1, pop(H)) & : \quad otherwise \end{cases} \tag{14.2}$$

However, heap adds some complexity to the solution. Is there any simple, fast method to find the $k$-th element?

The divide and conquer strategy can help us. If we can divide all the elements into two sub lists $A$ and $B$, and ensure all the elements in $A$ is not greater than any elements in $B$, we can scale down the problem by following this method[1]:

1. Compare the length of sub list $A$ and $k$;
2. If $k < |A|$, the $k$-th smallest one must be contained in $A$, we can drop $B$ and *further search* in $A$;
3. If $|A| < k$, the $k$-th smallest one must be contained in $B$, we can drop $A$ and *further search* the $(k - |A|)$-th smallest one in $B$.

Note that the *italic font* emphasizes the fact of recursion. The ideal case always divides the list into two equally big sub lists $A$ and $B$, so that we can halve the problem each time. Such ideal case leads to a performance of $O(n)$ linear time.

Thus the key problem is how to realize dividing, which collects the first $m$ smallest elements in one sub list, and put the rest in another.

This reminds us the partition algorithm in quick sort, which moves all the elements smaller than the pivot in front of it, and moves those greater than the pivot behind it. Based on this idea, we can develop a divide and conquer $k$-selection algorithm, which is called quick selection algorithm.

1. Randomly select an element (the first for instance) as the pivot;
2. Moves all elements which aren't greater than the pivot in a sub list $A$; and moves the rest to sub list $B$;
3. Compare the length of $A$ with $k$, if $|A| = k - 1$, then the pivot is the $k$-th smallest one;
4. If $|A| > k - 1$, recursively find the $k$-th smallest one among $A$;

[1]This actually demands a more accurate definition of the $k$-th smallest in $L$: It's equal to the $k$-the element of $L'$, where $L'$ is a permutation of $L$, and $L'$ is in monotonic non-decreasing order.

5. Otherwise, recursively find the $(k - 1 - |A|)$-th smallest one among $B$;

This algorithm can be formalized in below equation. Suppose $0 < k \leq |L|$ , where $L$ is a non-empty list of elements. Denote $l_1$ as the first element in $L$. It is chosen as the pivot; $L'$ contains the rest elements except for $l_1$. $(A, B) = partition(\lambda_x \cdot x \leq l_1, L')$. It partitions $L'$ by using the same algorithm defined in the chapter of quick sort.

$$top(k, L) = \begin{cases} l_1 & : |A| = k - 1 \\ top(k - 1 - |A|, B) & : |A| < k - 1 \\ top(k, A) & : otherwise \end{cases} \tag{14.3}$$

$$partition(p, L) = \begin{cases} (\phi, \phi) & : L = \phi \\ (\{l_1\} \cup A, B) & : p(l_1), (A, B) = partition(p, L') \\ (A, \{l_1\} \cup B) & : \neg p(l_1) \end{cases} \tag{14.4}$$

The following Haskell example program implements this algorithm.

```
top n (x:xs) | len == n - 1 = x
             | len < n - 1 = top (n - len - 1) bs
             | otherwise = top n as
    where
      (as, bs) = partition (<=x) xs
      len = length as
```

The partition function is provided in Haskell standard library, the detailed implementation can be referred to previous chapter about quick sort.

The lucky case is that, the $k$-th smallest element is selected as the pivot at the very beginning. The partition function examines the whole list, and finds that there are $k - 1$ elements not greater than the pivot, we are done in just $O(n)$ time. The worst case is that either the maximum or the minimum element is selected as the pivot every time. The partition always produces an empty sub list, that either $A$ or $B$ is empty. If we always pick the minimum as the pivot, the performance is bound to $O(kn)$. If we always pick the maximum as the pivot, the performance is $O((n - k)n)$.

The best case (not the lucky case), is that the pivot always partition the list perfectly. The length of $A$ is nearly as same as the length of $B$. The list is halved every time. It needs about $O(\lg n)$ partitions, each partition takes linear time proportion to the length of the halved list. This can be expressed as $O(n + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{2^m})$, where $m$ is the smallest number satisfies $\frac{n}{2^m} < k$. Summing the series leads to the result of $O(n)$.

The average case analysis needs tool of mathematical expectation. It's quite similar to the proof given in previous chapter of quick sort. It's left as an exercise to the reader.

Similar as quick sort, this divide and conquer selection algorithm performs well most time in practice. We can take the same engineering practice such as media-of-three, or randomly select the pivot as we did for quick sort. Below is the imperative realization for example.

1: **function** TOP$(k, A, l, u)$
2: EXCHANGE $A[l] \leftrightarrow A[$ RANDOM$(l, u)$ $]$ ▷ Randomly select in $[l, u]$
3: $p \leftarrow$ PARTITION$(A, l, u)$
4: **if** $p - l + 1 = k$ **then**
5: **return** $A[p]$
6: **if** $k < p - l + 1$ **then**
7: **return** TOP$(k, A, l, p - 1)$
8: **return** TOP$(k - p + l - 1, A, p + 1, u)$

This algorithm searches the $k$-th smallest element in range of $[l, u]$ for array $A$. The boundaries are included. It first randomly selects a position, and swaps it with the first

one. Then this element is chosen as the pivot for partitioning. The partition algorithm in-place moves elements and returns the position where the pivot being moved. If the pivot is just located at position $k$, then we are done; if there are more than $k-1$ elements not greater than the pivot, the algorithm recursively searches the $k$-th smallest one in range $[l, p-1]$; otherwise, $k$ is deduced by the number of elements before the pivot, and recursively searches the range after the pivot $[p+1, u]$.

There are many methods to realize the partition algorithm, below one is based on N. Lumoto's method. Other realizations are left as exercises to the reader.

1: **function** PARTITION(A, l, u)
2: $p \leftarrow A[l]$
3: $L \leftarrow l$
4: **for** $R \leftarrow l+1$ to $u$ **do**
5: **if** $\neg(p < A[R])$ **then**
6: $L \leftarrow L+1$
7: EXCHANGE $A[L] \leftrightarrow A[R]$
8: EXCHANGE $A[L] \leftrightarrow p$
9: **return** $L$

Below ANSI C example program implements this algorithm. Note that it handles the special case that either the array is empty, or $k$ is out of the boundaries of the array. It returns -1 to indicate the search failure.

```
int partition(Key* xs, int l, int u) {
    int r, p = l;
    for (r = l + 1; r < u; ++r)
        if (!(xs[p] < xs[r]))
            swap(xs, ++l, r);
    swap(xs, p, l);
    return l;
}

/* The result is stored in xs[k], returns k if u-l ≥k, otherwise -1 */
int top(int k, Key* xs, int l, int u) {
    int p;
    if (l < u) {
        swap(xs, l, rand() % (u - l) + l);
        p = partition(xs, l, u);
        if (p - l + 1 == k)
            return p;
        return (k < p - l + 1) ? top(k, xs, l, p) :
                                 top(k- p + l - 1, xs, p + 1, u);
    }
    return -1;
}
```

There is a method proposed by Blum, Floyd, Pratt, Rivest and Tarjan in 1973, which ensures the worst case performance being bound to $O(n)$ [4], [81]. It divides the list into small groups. Each group contains no more than 5 elements. The median of each group among these 5 elements are identified quickly. Then there are $\frac{n}{5}$ median elements selected. We repeat this step, and divide them again into groups of 5, and recursively select the *median of median*. It's obviously that the final 'true' median can be found in $O(\lg n)$ time. This is the best pivot for partitioning the list. Next, we halve the list by this pivot and recursively search for the $k$-th smallest one. The performance can be calculated as the following.

$$T(n) = c_1 lgn + c_2 n + T(\frac{n}{2}) \tag{14.5}$$

Where $c_1$ and $c_2$ are constant factors for the median of median and partition computation respectively. Solving this equation with telescope method or the master theory

in[4] gives the linear $O(n)$ performance.

In case we just want to pick the top $k$ smallest elements, but don't care about the order of them, the algorithm can be adjusted a little bit to fit.

$$
tops(k, L) = \begin{cases}
\phi & : k = 0 \lor L = \phi \\
A & : |A| = k \\
A \cup \{l_1\} \cup tops(k - |A| - 1, B) & : |A| < k \\
tops(k, A) & : otherwise
\end{cases} \tag{14.6}
$$

Where $A$, $B$ have the same meaning as before that, $(A, B) = partition(\lambda_x \cdot x \leq l_1, L')$ if $L$ isn't empty. The relative example program in Haskell is given as below.

```
tops _ [] = []
tops 0 _  = []
tops n (x:xs) | len ==n = as
              | len < n  = as ++ [x] ++ tops (n-len-1) bs
              | otherwise = tops n as
    where
      (as, bs) = partition (≤ x) xs
      len = length as
```

### binary search

Another popular divide and conquer algorithm is binary search. We've shown it in the chapter about insertion sort. When I was in school, the teacher who taught math played a magic to me, He asked me to consider a natural number less than 1000. Then he asked me some questions, I only replied 'yes' or 'no', and finally he guessed my number. He typically asked questions like the following:

- Is it an even number?
- Is it a prime number?
- Are all digits same?
- Can it be divided by 3?
- ...

Most of the time he guessed the number within 10 questions. My classmates and I all thought it's unbelievable.

This game will not be so interesting if it downgrades to a popular TV program, that the price of a product is hidden, and you must figure out the exact price in 30 seconds. The host of the program tells you if your guess is higher or lower to the fact. If you win, the product is yours. The best strategy is to use similar divide and conquer approach to perform a binary search. So it's common to find such conversation between the player and the host:

- P: 1000;
- H: High;
- P: 500;
- H: Low;
- P: 750;

- H: Low;
- P: 890;
- H: Low;
- P: 990;
- H: Bingo.

My math teacher told us that, because the number we considered is within 1000, if he can halve the numbers every time by designing good questions, the number will be found in 10 questions. This is because $2^{10} = 1024 > 1000$. However, it would be boring to just ask it is higher than 500, is lower than 250, ... Actually, the question 'is it even' is very good, because it always halve the numbers[2].

Come back to the binary search algorithm. It is only applicable to a sequence of ordered number. I've seen programmers tried to apply it to unsorted array, and took several hours to figure out why it doesn't work. The idea is quite straightforward, in order to find a number $x$ in an ordered sequence $A$, we firstly check middle point number, compare it with $x$, if they are same, then we are done; If $x$ is smaller, as $A$ is ordered, we need only recursively search it among the first half; otherwise we search it among the second half. Once $A$ gets empty and we haven't found $x$ yet, it means $x$ doesn't exist.

Before formalizing this algorithm, there is a surprising fact need to be noted. Donald Knuth stated that 'Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky¡'. Jon Bentley pointed out that most binary search implementation contains errors, and even the one given by him in the first version of 'Programming pearls' contains an error undetected over twenty years[4].

There are two kinds of realization, one is recursive, the other is iterative. The recursive solution is as same as what we described. Suppose the lower and upper boundaries of the array are $l$ and $u$ inclusive.

1: **function** BINARY-SEARCH($x, A, l, u$)
2:     **if** $u < l$ **then**
3:         Not found error
4:     **else**
5:         $m \leftarrow l + \lfloor \frac{u-l}{2} \rfloor$     ▷ avoid overflow of $\lfloor \frac{l+u}{2} \rfloor$
6:         **if** $A[m] = x$ **then**
7:             **return** $m$
8:         **if** $x < A[m]$ **then**
9:             **return** BINARY-SEARCH(x, A, l, m - 1)
10:        **else**
11:            **return** BINARY-SEARCH(x, A, m + 1, u)

As the comment highlights, if the integer is represented with limited words, we can't merely use $\lfloor \frac{l+u}{2} \rfloor$ because it may cause overflow if $l$ and $u$ are big.

Binary search can also be realized in iterative manner, that we keep updating the boundaries according to the middle point comparison result.

1: **function** BINARY-SEARCH($x, A, l, u$)
2:     **while** $l < u$ **do**
3:         $m \leftarrow l + \lfloor \frac{u-l}{2} \rfloor$
4:         **if** $A[m] = x$ **then**
5:             **return** $m$

[2] When the author revise this chapter, Microsoft released a game in social networks. User can consider a person's name, the AI robot asks 16 questions next. The user only answers with yes or no. The robot will tell you who is that person. Can you figure out how the robot works?

6: **if** $x < A[m]$ **then**
7: $u \leftarrow m - 1$
8: **else**
9: $l \leftarrow m + 1$
**return** NIL

The implementation is very good exercise, we left it to the reader. Please try all kinds of methods to verify your program.

Since the array is halved every time, the performance of binary search is bound to $O(\lg n)$ time.

In purely functional settings, the list is represented with singly linked-list. It's linear time to randomly access the element for a given position. Binary search doesn't make sense in such case. However, it good to analyze what the performance will downgrade to. Consider the following equation.

$$
bsearch(x, L) = \begin{cases} Err & : L = \phi \\ b_1 & : x = b_1, (A, B) = splitAt(\lfloor \frac{|L|}{2} \rfloor, L) \\ bsearch(x, A) & : B = \phi \lor x < b_1 \\ bsearch(x, B') & : otherwise \end{cases}
$$

Where $b_1$ is the first element if $B$ isn't empty, and $B'$ holds the rest except for $b_1$. The *splitAt* function takes $O(n)$ time to divide the list into two subs $A$ and $B$ (see the appendix A, and the chapter about merge sort for detail). If $B$ isn't empty and $x$ is equal to $b_1$, the search returns; Otherwise if it is less than $b_1$, as the list is sorted, we need recursively search in $A$, otherwise, we search in $B$. If the list is empty, we raise error to indicate search failure.

As we always split the list in the middle point, the number of elements halves in each recursion. In every recursive call, we takes linear time for splitting. The splitting function only traverses the first half of the linked-list, Thus the total time can be expressed as.

$$
T(n) = c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} + ...
$$

This results $O(n)$ time, which is as same as the brute force search from head to tail:

$$
search(x, L) = \begin{cases} Err & : L = \phi \\ l_1 & : x = l_1 \\ search(x, L') & : otherwise \end{cases}
$$

As we mentioned in the chapter about insertion sort, the functional approach of binary search is through binary search tree. That the ordered sequence is represented in a tree ( self balanced tree if necessary), which offers logarithm time searching [3].

Although it doesn't make sense to apply divide and conquer binary search on linked-list, binary search can still be very useful in purely functional settings. Consider solving an equation $a^x = y$, for given natural numbers $a$ and $y$, where $a \leq y$. We want to find the integer solution for $x$ if there is. Of course brute-force naive searching can solve it. We can examine all numbers one by one from 0 for $a^0, a^1, a^2, ...$, stops if $a^i = y$ or report that there is no solution if $a^i < y < a^{i+1}$ for some $i$. We initialize the solution domain as $X = \{0, 1, 2, ...\}$, and call the below exhausted searching function $solve(a, y, X)$.

$$
solve(a, y, X) = \begin{cases} x_1 & : a^{x_1} = y \\ solve(a, y, X') & : a^{x_1} < y \\ Err & : otherwise \end{cases}
$$

[3] Some readers may argue that array should be used instead of linked-list, for example in Haskell. This book only deals with purely functional sequences in finger-tree. Different from the Haskell array, it can't support constant time random accessing

This function examines the solution domain in monotonic increasing order. It takes the first candidate element $x_1$ from $X$, compare $a^{x_1}$ and $y$, if they are equal, then $x_1$ is the solution and we are done; if it is less than $y$, then $x_1$ is dropped, and we search among the rest elements represented as $X'$; Otherwise, since $f(x) = a^x$ is non-decreasing function when $a$ is natural number, so the rest elements will only make $f(x)$ bigger and bigger. There is no integer solution for this equation. The function returns error to indicate no solution.

The computation of $a^x$ is expensive for big $a$ and $x$ if precession must be kept[4]. Can it be improved so that we can compute as less as possible? The divide and conquer binary search can help. Actually, we can estimate the upper limit of the solution domain. As $a^y \leq y$, We can search in range $\{0, 1, ..., y\}$. As the function $f(x) = a^x$ is non-decreasing against its argument $x$, we can firstly check the middle point candidate $x_m = \lfloor \frac{0+y}{2} \rfloor$, if $a^{x_m} = y$, the solution is found; if it is less than $y$, we can drop all candidate solutions before $x_m$; otherwise we drop all candidate solutions after it; Both halve the solution domain. We repeat this approach until either the solution is found or the solution domain becomes empty, which indicates there is no integer solution.

The binary search method can be formalized as the following equation. The non-decreasing function is abstracted as a parameter. To solve our problem, we can just call it as $bsearch(f, y, 0, y)$, where $f(x) = a^x$.

$$
bsearch(f, y, l, u) = \begin{cases} Err & : u < l \\ m & : f(m) = y, m = \lfloor \frac{l+u}{2} \rfloor \\ bsearch(f, y, l, m-1) & : f(m) > y \\ bsearch(f, y, m+1, u) & : f(m) < y \end{cases} \tag{14.7}
$$

As we halve the solution domain in every recursion, this method computes $f(x)$ in $O(\log y)$ times. It is much faster than the brute-force searching.

### 2 dimensions search

It's quite natural to think that the idea of binary search can be extended to 2 dimensions or even more general – multiple-dimensions domain. However, it is not so easy.

Consider the example of a $m \times n$ matrix $M$. The elements in each row and each column are in strict increasing order. Figure 14.1 illustrates such a matrix for example.

$$
\begin{bmatrix} 1 & 2 & 3 & 4 & ... \\ 2 & 4 & 5 & 6 & ... \\ 3 & 5 & 7 & 8 & ... \\ 4 & 6 & 8 & 9 & ... \\ ... \end{bmatrix}
$$

Figure 14.1: A matrix in strict increasing order for each row and column.

Given a value $x$, how to locate all elements equal to $x$ in the matrix quickly? We need develop an algorithm, which returns a list of locations $(i, j)$ so that $M_{i,j} = x$.

Richard Bird in[1] mentioned that he used this problem to interview candidates for entry to Oxford. The interesting story was that, those who had some computer background at school tended to use binary search. But it's easy to get stuck.

The usual way follows binary search idea is to examine element at $M_{\frac{m}{2}, \frac{n}{2}}$. If it is less than $x$, we can only drop the elements in the top-left area; If it is greater than $x$, only

[4] One alternative is to reuse the result of $a^n$ when compute $a^{n+1} = aa^n$. Here we consider for general form monotonic function $f(n)$

the bottom-right area can be dropped. Both cases are illustrated in figure 14.2, the gray areas indicate elements can be dropped.

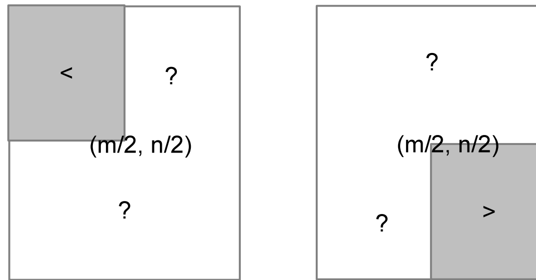



Figure 14.2: Left: the middle point element is smaller than $x$. All elements in the gray area are less than $x$; Right: the middle point element is greater than $x$. All elements in the gray area are greater than $x$.

The problem is that the solution domain changes from a rectangle to a 'L' shape in both cases. We can't just recursively apply search on it. In order to solve this problem systematically, we define the problem more generally, using brute-force search as a start point, and keep improving it bit by bit.

Consider a function $f(x, y)$, which is strict increasing for its arguments, for instance $f(x, y) = a^x + b^y$, where $a$ and $b$ are natural numbers. Given a value $z$, which is a natural number too, we want to solve the equation $f(x, y) = z$ by finding all none negative integral candidate pairs $(x, y)$.

With this definition, the matrix search problem can be specialized by below function.

$$f(x, y) = \begin{cases} M_{x,y} & : 1 \leq x \leq m, 1 \leq y \leq n \\ -1 & : otherwise \end{cases}$$

**Brute-force 2D search**

As all solutions should be found for $f(x, y)$. One can immediately give the brute force solution by embedded looping.

```
1: function SOLVE(f, z)
2:     A ← φ
3:     for x ∈ {0, 1, 2, ..., z} do
4:         for y ∈ {0, 1, 2, ..., z} do
5:             if f(x, y) = z then
6:                 A ← A ∪ {(x, y)}
7:     return A
```

This definitely calculates $f$ for $(z + 1)^2$ times. It can be formalized as in (14.8).

$$solve(f, z) = \{(x, y) | x \in \{0, 1, ..., z\}, y \in \{0, 1, ..., z\}, f(x, y) = z\} \tag{14.8}$$

**Saddleback search**

We haven't utilize the fact that $f(x, y)$ is strict increasing yet. Dijkstra pointed out in[82], instead of searching from bottom-left corner, starting from the top-left leads to

one effective solution. As illustrated in figure 14.3, the search starts from $(0, z)$, for every point $(p, q)$, we compare $f(p, q)$ with $z$:

- If $f(p, q) < z$, since $f$ is strict increasing, for all $0 \leq y < q$, we have $f(p, y) < z$. We can drop all points in the vertical line section (in red color);
- If $f(p, q) > z$, then $f(x, q) > z$ for all $p < x \leq z$. We can drop all points in the horizontal line section (in blue color);
- Otherwise if $f(p, q) = z$, we mark $(p, q)$ as one solution, then both line sections can be dropped.

This is a systematical way to scale down the solution domain rectangle. We keep dropping a row, or a column, or both.

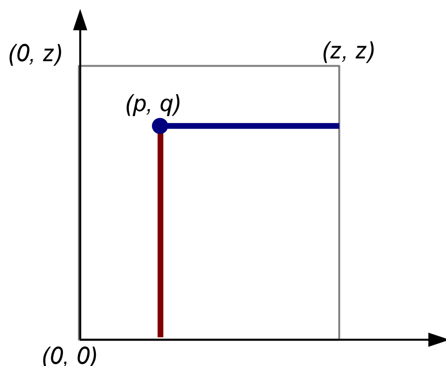



Figure 14.3: Search from top-left.

This method can be formalized as a function $search(f, z, p, q)$, which searches solutions for equation $f(x, y) = z$ in rectangle with top-left corner $(p, q)$, and bottom-right corner $(z, 0)$. We start the searching by initializing $(p, q) = (0, z)$ as $solve(f, z) = search(f, z, 0, z)$

$$
search(f, z, p, q) = \begin{cases} \phi & : p > z \lor q < 0 \\ search(f, z, p + 1, q) & : f(p, q) < z \\ search(f, z, p, q - 1) & : f(p, q) > z \\ \{(p, q)\} \cup search(f, z, p + 1, q - 1) & : otherwise \end{cases} \tag{14.9}
$$

The first clause is the edge case, there is no solution if $(p, q)$ isn't top-left to $(z, 0)$. The following example Haskell program implements this algorithm.

```
solve f z = search 0 z where
  search p q | p > z || q < 0 = []
             | z' < z = search (p + 1) q
             | z' > z = search p (q - 1)
             | otherwise = (p, q) : search (p + 1) (q - 1)
    where z' = f p q
```

Considering the calculation of $f$ may be expensive, this program stores the result of $f(p, q)$ to variable $z'$. This algorithm can also be implemented in iterative manner, that the boundaries of solution domain keeps being updated in a loop.

1: **function** SOLVE$(f, z)$
2: $\quad p \leftarrow 0, q \leftarrow z$

3: $S \leftarrow \phi$
4: **while** $p \leq z \wedge q \geq 0$ **do**
5: $z' \leftarrow f(p, q)$
6: **if** $z' < z$ **then**
7: $p \leftarrow p + 1$
8: **else if** $z' > z$ **then**
9: $q \leftarrow q - 1$
10: **else**
11: $S \leftarrow S \cup \{(p, q)\}$
12: $p \leftarrow p + 1, q \leftarrow q - 1$
13: **return** $S$

It's intuitive to translate this imperative algorithm to real program, as the following example Python code.

```
def solve(f, z):
    (p, q) = (0, z)
    res = []
    while p ≤ z and q ≥ 0:
        z1 = f(p, q)
        if z1 < z:
            p = p + 1
        elif z1 > z:
            q = q - 1
        else:
            res.append((p, q))
            (p, q) = (p + 1, q - 1)
    return res
```

It is clear that in every iteration, At least one of $p$ and $q$ advances to the bottom-right corner by one. Thus it takes at most $2(z + 1)$ steps to complete searching. This is the worst case. There are three best cases. The first one happens that in every iteration, both $p$ and $q$ advance by one, so that it needs only $z + 1$ steps; The second case keeps advancing horizontally to right and ends when $p$ exceeds $z$; The last case is similar, that it keeps moving down vertically to the bottom until $q$ becomes negative.

Figure 14.4 illustrates the best cases and the worst cases respectively. Figure 14.4 (a) is the case that every point $(x, z - x)$ in diagonal satisfies $f(x, z - x) = z$, it uses $z + 1$ steps to arrive at $(z, 0)$; (b) is the case that every point $(x, z)$ along the top horizontal line gives the result $f(x, z) < z$, the algorithm takes $z + 1$ steps to finish; (c) is the case that every point $(0, x)$ along the left vertical line gives the result $f(0, x) > z$, thus the algorithm takes $z + 1$ steps to finish; (d) is the worst case. If we project all the horizontal sections along the search path to $x$ axis, and all the vertical sections to $y$ axis, it gives the total steps of $2(z + 1)$.

Compare to the quadratic brute-force method ($O(z^2)$), we improve to a linear algorithm bound to $O(z)$.

Bird imagined that the name 'saddleback' is because the 3D plot of $f$ with the smallest bottom-left and the largest top-right and two wings looks like a saddle as shown in figure 14.5

### Improved saddleback search

We haven't utilized the binary search tool so far, even the problem extends to 2-dimension domain. The basic saddleback search starts from the top-left corner $(0, z)$ to the bottom-right corner $(z, 0)$. This is actually over-general domain. we can constraint it a bit more accurate.

Since $f$ is strict increasing, we can find the biggest number $m$, that $0 \leq m \leq z$, along the $y$ axis which satisfies $f(0, m) \leq z$; Similarly, we can find the biggest $n$, that

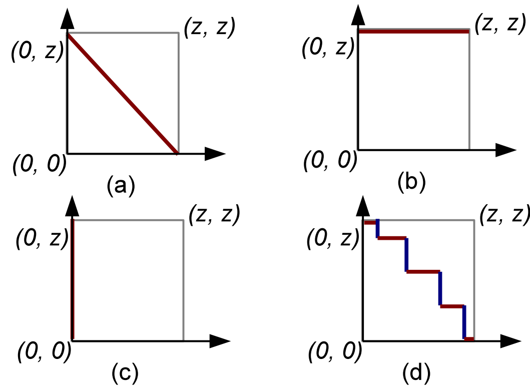



Figure 14.4: The best cases and the worst cases.

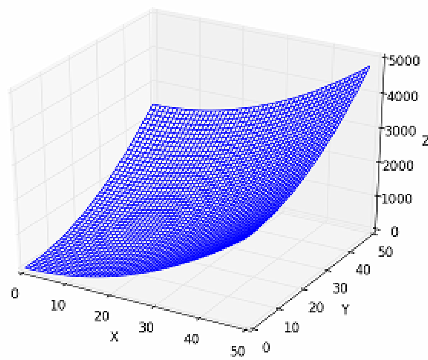



Figure 14.5: Plot of $f(x, y) = x^2 + y^2$.

$0 \leq n \leq z$, along the $x$ axis, which satisfies $f(n, 0) \leq z$; And the solution domain shrinks from $(0, z) - (z, 0)$ to $(0, m) - (n, 0)$ as shown in figure 14.6.

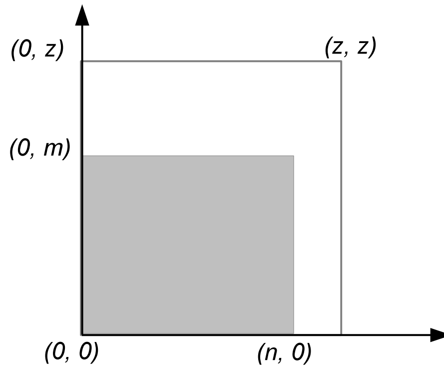



Figure 14.6: A more accurate search domain shown in gray color.

Of course $m$ and $n$ can be found by brute-force like below.

$$\begin{array}{rl} m & = max(\{y | 0 \leq y \leq z, f(0, y) \leq z\}) \\ n & = max(\{x | 0 \leq x \leq z, f(x, 0) \leq z\}) \end{array} \tag{14.10}$$

When searching $m$, the $x$ variable of $f$ is bound to 0. It turns to be one dimension search problem for a strict increasing function (or in functional term, a Curried function $f(0, y)$). Binary search works in such case. However, we need a bit modification for equation (14.7). Different from searching a solution $l \leq x \leq u$, so that $f(x) = y$ for a given $y$; we need search for a solution $l \leq x \leq u$ so that $f(x) \leq y < f(x+1)$.

$$bsearch(f, y, l, u) = \begin{cases} l & : u \leq l \\ m & : f(m) \leq y < f(m+1), m = \lfloor \frac{l+u}{2} \rfloor \\ bsearch(f, y, m+1, u) & : f(m) \leq y \\ bsearch(f, y, l, m-1) & : otherwise \end{cases} \tag{14.11}$$

The first clause handles the edge case of empty range. The lower boundary is returned in such case; If the middle point produces a value less than or equal to the target, while the next one evaluates to a bigger value, then the middle point is what we are looking for; Otherwise if the point next to the middle also evaluates to a value not greater than the target, the lower bound is set as the middle point plus one, and we perform recursively binary search; In the last case, the middle point evaluates to a value greater than the target, upper bound is updated as the point proceeds to the middle for further recursive searching. The following Haskell example code implements this modified binary search.

```
bsearch f y (l, u) | u ≤ l = l
                   | f m ≤ y = if f (m + 1) ≤ y
                               then bsearch f y (m + 1, u) else m
                   | otherwise = bsearch f y (l, m-1)
  where m = (l + u) `div` 2
```

Then $m$ and $n$ can be found with this binary search function.

$$\begin{array}{rl} m & = bsearch(\lambda_y \cdot f(0, y), z, 0, z) \\ n & = bsearch(\lambda_x \cdot f(x, 0), z, 0, z) \end{array} \tag{14.12}$$

And the improved saddleback search shrinks to this new search domain $solve(f, z) = search(f, z, 0, m)$:

$$search(f, z, p, q) = \begin{cases} \phi & : p > n \lor q < 0 \\ search(f, z, p + 1, q) & : f(p, q) < z \\ search(f, z, p, q - 1) & : f(p, q) > z \\ \{(p, q)\} \cup search(f, z, p + 1, q - 1) & : otherwise \end{cases} \tag{14.13}$$

It's almost as same as the basic saddleback version, except that it stops if $p$ exceeds $n$, but not $z$. In real implementation, the result of $f(p, q)$ can be calculated once, and stored in a variable as shown in the following Haskell example.

```
solve' f z = search 0 m where
  search p q | p > n || q < 0 = []
             | z' < z = search (p + 1) q
             | z' > z = search p (q - 1)
             | otherwise = (p, q) : search (p + 1) (q - 1)
    where z' = f p q
  m = bsearch (f 0) z (0, z)
  n = bsearch (λx→f x 0) z (0, z)
```

This improved saddleback search firstly performs binary search two rounds to find the proper $m$, and $n$. Each round is bound to $O(\lg z)$ times of calculation for $f$; After that, it takes $O(m + n)$ time in the worst case; and $O(min(m, n))$ time in the best case. The overall performance is given in the following table.

| | times of evaluation $f$ |
|---|---|
| worst case | $2 \log z + m + n$ |
| best case | $2 \log z + min(m, n)$ |

For some function $f(x, y) = a^x + b^y$, for positive integers $a$ and $b$, $m$ and $n$ will be relative small, that the performance is close to $O(\lg z)$.

This algorithm can also be realized in imperative approach. Firstly, the binary search should be modified.

1: **function** BINARY-SEARCH($f, y, (l, u)$)
2:     **while** $l < u$ **do**
3:         $m \leftarrow \lfloor \frac{l+u}{2} \rfloor$
4:         **if** $f(m) \leq y$ **then**
5:             **if** $y < f(m + 1)$ **then**
6:                 **return** m
7:             $l \leftarrow m + 1$
8:         **else**
9:             $u \leftarrow m$
10:     **return** $l$

Utilize this algorithm, the boundaries $m$ and $n$ can be found before performing the saddleback search.

1: **function** SOLVE($f, z$)
2:     $m \leftarrow$ BINARY-SEARCH($\lambda_y \cdot f(0, y), z, (0, z)$)
3:     $n \leftarrow$ BINARY-SEARCH($\lambda_x \cdot f(x, 0), z, (0, z)$)
4:     $p \leftarrow 0, q \leftarrow m$
5:     $S \leftarrow \phi$
6:     **while** $p \leq n \land q \geq 0$ **do**
7:         $z' \leftarrow f(p, q)$
8:         **if** $z' < z$ **then**
9:             $p \leftarrow p + 1$
10:         **else if** $z' > z$ **then**

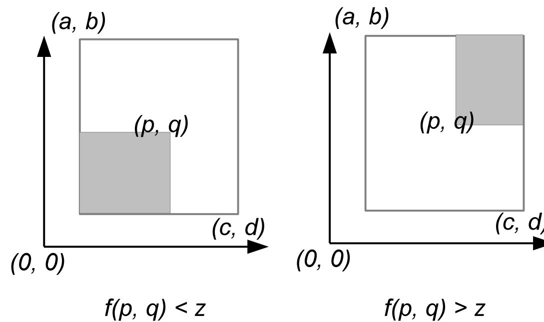
11: $q \leftarrow q - 1$
12: **else**
13: $S \leftarrow S \cup \{(p, q)\}$
14: $p \leftarrow p + 1, q \leftarrow q - 1$
15: **return** $S$

The implementation is left as exercise to the reader.

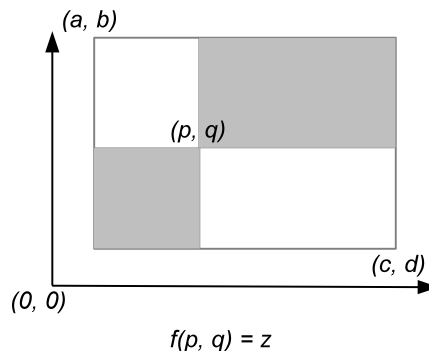
### More improvement to saddleback search

In figure 14.2, two cases are shown for comparing the value of the middle point in a matrix with the given value. One case is the center value is smaller than the given value, the other is bigger. In both cases, we can only throw away $\frac{1}{4}$ candidates, and left a 'L' shape for further searching.

Actually, one important case is missing. We can extend the observation to any point inside the rectangle searching area. As shown in the figure 14.7.




(a) If $f(p, q) \neq z$, only lower-left or upper-right sub area (in gray color) can be thrown. Both left a 'L' shape.




(b) If $f(p, q) = z$, both sub areas can be thrown, the scale of the problem is halved.

Figure 14.7: The efficiency of scaling down the search domain.

Suppose we are searching in a rectangle from the upper-left corner $(a, b)$ to the lower-right corner $(c, d)$. If the $(p, q)$ isn't the middle point, and $f(p, q) \neq z$. We can't ensure the area to be dropped is always 1/4. However, if $f(p, q) = z$, as $f$ is strict increasing, we

are not only sure both the lower-left and the upper-right sub areas can be thrown, but also all the other points in the column $p$ and row $q$. The problem can be scaled down fast, because only 1/2 area is left.

This indicates us, instead of jumping to the middle point to start searching. A more efficient way is to find a point which evaluates to the target value. One straightforward way to find such a point, is to perform binary search along the center horizontal line or the center vertical line of the rectangle.

The performance of binary search along a line is logarithmic to the length of that line. A good idea is to always pick the shorter center line as shown in figure 14.8. That if the height of the rectangle is longer than the width, we perform binary search along the horizontal center line; otherwise we choose the vertical center line.

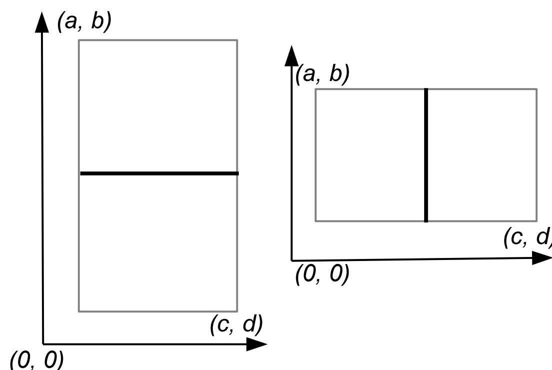



Figure 14.8: Binary search along the shorter center line.

However, what if we can't find a point $(p, q)$ in the center line, that satisfies $f(p, q) = z$? Let's take the center horizontal line for example. even in such case, we can still find a point that $f(p, q) < z < f(p + 1, q)$. The only difference is that we can't drop the points in column $p$ and row $q$ completely.

Combine this conditions, the binary search along the horizontally line is to find a $p$, satisfies $f(p, q) \leq z < f(p+1, q)$; While the vertical line search condition is $f(p, q) \leq z < f(p, q + 1)$.

The modified binary search ensures that, if all points in the line segment give $f(p, q) < z$, the upper bound will be found; and the lower bound will be found if they all greater than $z$. We can drop the whole area on one side of the center line in such case.

Sum up all the ideas, we can develop the efficient improved saddleback search as the following.

1. Perform binary search along the $y$ axis and $x$ axis to find the tight boundaries from $(0, m)$ to $(n, 0)$;

2. Denote the candidate rectangle as $(a, b) - (c, d)$, if the candidate rectangle is empty, the solution is empty;

3. If the height of the rectangle is longer than the width, perform binary search along the center horizontal line; otherwise, perform binary search along the center vertical line; denote the search result as $(p, q)$;

4. If $f(p, q) = z$, record $(p, q)$ as a solution, and recursively search two sub rectangles $(a, b) - (p - 1, q + 1)$ and $(p + 1, q - 1) - (c, d)$;

5. Otherwise, $f(p, q) \neq z$, recursively search the same two sub rectangles plus a line section. The line section is either $(p, q+1) - (p, b)$ as shown in figure 14.9 (a); or $(p+1, q) - (c, q)$ as shown in figure 14.9 (b).

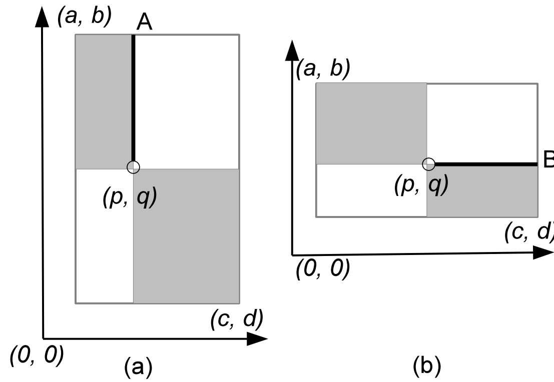



Figure 14.9: Recursively search the gray areas, the bold line should be included if $f(p, q) \neq z$.

This algorithm can be formalized as the following. The equation (14.11), and (14.12) are as same as before. A new *search* function should be defined.

Define $Search_{(a,b),(c,d)}$ as a function for searching rectangle with top-left corner $(a, b)$, and bottom-right corner $(c, d)$.

$$search_{(a,b),(c,d)} = \begin{cases} \phi & : c < a \lor d < b \\ csearch & : c - a < b - d \\ rsearch & : otherwise \end{cases} \tag{14.14}$$

Function *csearch* performs binary search in the center horizontal line to find a point $(p, q)$ that $f(p, q) \leq z < f(p+1, q)$. This is shown in figure 14.9 (a). There is a special edge case, that all points in the lines evaluate to values greater than $z$. The general binary search will return the lower bound as result, so that $(p, q) = (a, \lfloor \frac{b+d}{2} \rfloor)$. The whole upper side includes the center line can be dropped as shown in figure 14.10 (a).

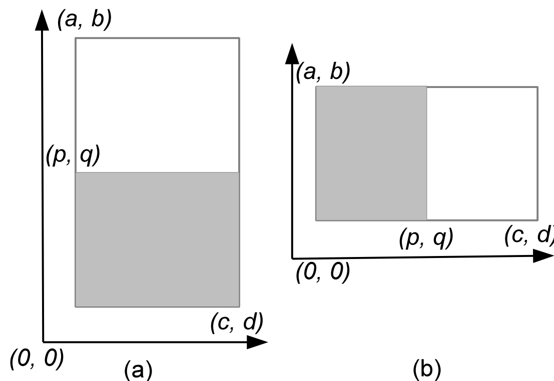



Figure 14.10: Edge cases when performing binary search in the center line.

$$csearch = \begin{cases} search_{(p,q-1),(c,d)} & : z < f(p,q) \\ search_{(a,b),(p-1,q+1)} \cup \{(p,q)\} \cup search_{(p+1,q-1),(c,d)} & : f(p,q) = z \\ search_{(a,b),(p,q+1)} \cup search_{(p+1,q-1),(c,d)} & : otherwise \end{cases} \tag{14.15}$$

Where

$$q = \lfloor \frac{b+d}{2} \rfloor) \\ p = bsearch(\lambda_x \cdot f(x,q), z, (a,c))$$

Function *rsearch* is quite similar except that it searches in the center horizontal line.

$$rsearch = \begin{cases} search_{(a,b),(p-1,q)} & : z < f(p,q) \\ search_{(a,b),(p-1,q+1)} \cup \{(p,q)\} \cup search_{(p+1,q-1),(c,d)} & : f(p,q) = z \\ search_{(a,b),(p-1,q+1)} \cup search_{(p+1,q),(c,d)} & : otherwise \end{cases} \tag{14.16}$$

Where

$$p = \lfloor \frac{a+c}{2} \rfloor) \\ q = bsearch(\lambda_y \cdot f(p,y), z, (d,b))$$

The following Haskell program implements this algorithm.

```
search f z (a, b) (c, d) | c < a || b < d = []
                         | c - a < b - d = let q = (b + d) `div` 2 in
                                    csearch (bsearch (λx → f x q) z (a, c), q)
                         | otherwise = let p = (a + c) `div` 2 in
                                    rsearch (p, bsearch (f p) z (d, b))
  where
    csearch (p, q) | z < f p q = search f z (p, q - 1) (c, d)
                   | f p q == z = search f z (a, b) (p - 1, q + 1) ++
                                  (p, q) : search f z (p + 1, q - 1) (c, d)
                   | otherwise = search f z (a, b) (p, q + 1) ++
                                 search f z (p + 1, q - 1) (c, d)
    rsearch (p, q) | z < f p q = search f z (a, b) (p - 1, q)
                   | f p q == z = search f z (a, b) (p - 1, q + 1) ++
                                  (p, q) : search f z (p + 1, q - 1) (c, d)
                   | otherwise = search f z (a, b) (p - 1, q + 1) ++
                                  search f z (p + 1, q) (c, d)
```

And the main program calls this function after performing binary search in $X$ and $Y$ axes.

```
solve f z = search f z (0, m) (n, 0) where
  m = bsearch (f 0) z (0, z)
  n = bsearch (λx → f x 0) z (0, z)
```

Since we drop half areas in every recursion, it takes $O(\log(mn))$ rounds of search. However, in order to locate the point $(p,q)$, which halves the problem, we must perform binary search along the center line. which will call $f$ about $O(\log(min(m,n)))$ times. Denote the time of searching a $m \times n$ rectangle as $T(m,n)$, the recursion relationship can be represented as the following.

$$T(m,n) = \log(min(m,n)) + 2T(\frac{m}{2}, \frac{n}{2}) \tag{14.17}$$

Suppose $m > n$, using telescope method, for $m = 2^i$, and $n = 2^j$. We have:

$$\begin{array}{rl} T(2^i, 2^j) & = j + 2T(2^{i-1}, 2^{j-1}) \\ & = \sum_{k=0}^{i-1} 2^k(j-k) \\ & = O(2^i(j-i)) \\ & = O(m \log(n/m)) \end{array} \tag{14.18}$$

Richard Bird proved that this is asymptotically optimal by a lower bound of searching a given value in $m \times n$ rectangle[1].

The imperative algorithm is almost as same as the functional version. We skip it for the sake of brevity.

## Exercise 14.1

- Prove that the average case for the divide and conquer solution to $k$-selection problem is $O(n)$. Please refer to previous chapter about quick sort.
- Implement the imperative $k$-selection problem with 2-way partition, and median-of-three pivot selection.
- Implement the imperative $k$-selection problem to handle duplicated elements effectively.
- Realize the median-of-median $k$-selection algorithm and implement it in your favorite programming language.
- The $tops(k, L)$ algorithm uses list concatenation likes $A \cup \{l_1\} \cup tops(k - |A| - 1, B)$. It is linear operation which is proportion to the length of the list to be concatenated. Modify the algorithm so that the sub lists are concatenated by one pass.
- The author considered another divide and conquer solution for the $k$-selection problem. It finds the maximum of the first $k$ elements and the minimum of the rest. Denote them as $x$, and $y$. If $x$ is smaller than $y$, it means that all the first $k$ elements are smaller than the rest, so that they are exactly the top $k$ smallest; Otherwise, There are some elements in the first $k$ should be swapped.

```
1: procedure Tops(k, A)
2:     l ← 1
3:     u ← |A|
4:     loop
5:         i ← Max-At(A[l..k])
6:         j ← Min-At(A[k + 1..u])
7:         if A[i] < A[j] then
8:             break
9:         Exchange A[l] ↔ A[j]
10:        Exchange A[k + 1] ↔ A[i]
11:        l ← Partition(A, l, k)
12:        u ← Partition(A, k + 1, u)
```

  Explain why this algorithm works? What's the performance of it?

- Implement the binary search algorithm in both recursive and iterative manner, and try to verify your version automatically. You can either generate randomized data, test your program with the binary search invariant, or compare with the built-in binary search tool in your standard library.
- Find the solution to calculate the median of two sorted arrays $A$ and $B$. The time should be bound to $O(\lg(|A| + |B|))$.
- Implement the improved saddleback search by firstly performing binary search to find a more accurate solution domain in your favorite imperative programming language.

- Realize the improved 2D search, by performing binary search along the shorter center line, in your favorite imperative programming language.

- Someone considers that the 2D search can be designed as the following. When search a rectangle, as the minimum value is at bottom-left, and the maximum at to-right. If the target value is less than the minimum or greater than the maximum, then there is no solution; otherwise, the rectangle is divided into 4 sub rectangles at the center point, then perform recursively searching.

```
1: procedure SEARCH(f, z, a, b, c, d)      ▷ (a, b): bottom-left (c, d): top-right
2:     if z ≤ f(a, b) ∨ f(c, d) ≥ z then
3:         if z = f(a, b) then
4:             record (a, b) as a solution
5:         if z = f(c, d) then
6:             record (c, d) as a solution
7:         return
8:     p ← ⌊(a+c)/2⌋
9:     q ← ⌊(b+d)/2⌋
10:    SEARCH(f, z, a, q, p, d)
11:    SEARCH(f, z, p, q, c, d)
12:    SEARCH(f, z, a, b, p, q)
13:    SEARCH(f, z, p, b, c, q)
```

  What's the performance of this algorithm?

### 14.2.2 Information reuse

One interesting behavior is that people learning while searching. We do not only remember lessons which cause search fails, but also learn patterns which lead to success. This is a kind of information reusing, no matter the information is positive or negative. However, It's not easy to determine what information should be kept. Too little information isn't enough to help effective searching, while keeping too much is expensive in term of spaces.

In this section, we'll first introduce two interesting problems, Boyer-Moore majority number problem and the maximum sum of sub vector problem. Both reuse information as little as possible. After that, two popular string matching algorithms, Knuth-Morris-Pratt algorithm and Boyer-Moore algorithm will be introduced.

#### Boyer-Moore majority number

Voting is quite critical to people. We use voting to choose the leader, make decision or reject a proposal. In the months when I was writing this chapter, there are three countries in the world voted their presidents. All of the three voting activities utilized computer to calculate the result.

Suppose there is a country in a small island wants a new president. According to the constitution, only if the candidate wins more than half of the votes can be selected as the president. Given a serious of votes, such as A, B, A, C, B, B, D, ..., can we develop a program tells who is the new president if there is, or indicate nobody wins more than half of the votes?

Of course this problem can be solved with brute-force by using a map. As what we did in the chapter of binary search tree[5].

[5]There is a probabilistic sub-linear space counting algorithm published in 2004, named as 'Count-min sketch'[84].

```
template<typename T>
T majority(const T* xs, int n, T fail) {
    map<T, int> m;
    int i, max = 0;
    T r;
    for (i = 0; i < n; ++i)
        ++m[xs[i]];
    for (typename map<T, int>::iterator it = m.begin(); it != m.end(); ++it)
        if (it->second > max) {
            max = it->second;
            r = it->first;
        }
    return max * 2 > n ? r : fail;
}
```

This program first scan the votes, and accumulates the number of votes for each individual with a map. After that, it traverse the map to find the one with the most of votes. If the number is bigger than the half, the winner is found otherwise, it returns a special value to indicate fail.

The following pseudo code describes this algorithm.

1: **function** MAJORITY($A$)
2: $M \leftarrow$ empty map
3: **for** $\forall a \in A$ **do**
4: PUT($M$, $a$, 1+ GET($M, a$))
5: $max \leftarrow 0$, $m \leftarrow NIL$
6: **for** $\forall (k, v) \in M$ **do**
7: **if** $max < v$ **then**
8: $max \leftarrow v$, $m \leftarrow k$
9: **if** $max > |A|50\%$ **then**
10: **return** $m$
11: **else**
12: fail

For $m$ individuals and $n$ votes, this program firstly takes about $O(n \log m)$ time to build the map if the map is implemented in self balanced tree (red-black tree for instance); or about $O(n)$ time if the map is hash table based. However, the hash table needs more space. Next the program takes $O(m)$ time to traverse the map, and find the majority vote. The following table lists the time and space performance for different maps.

| map | time | space |
|---|---|---|
| self-balanced tree | $O(n \log m)$ | $O(m)$ |
| hashing | $O(n)$ | $O(m)$ at least |

Boyer and Moore invented a cleaver algorithm in 1980, which can pick the majority element with only one scan if there is. Their algorithm only needs $O(1)$ space[83].

The idea is to record the first candidate as the winner so far, and mark him with 1 vote. During the scan process, if the winner being selected gets another vote, we just increase the vote counter; otherwise, it means somebody vote against this candidate, so the vote counter should be decreased by one. If the vote counter becomes zero, it means this candidate is voted out; We select the next candidate as the new winner and repeat the above scanning process.

Suppose there is a series of votes: A, B, C, B, B, C, A, B, A, B, B, D, B. Below table illustrates the steps of this processing.

| winner | count | scan position |
|---|---|---|
| A | 1 | **A**, B, C, B, B, C, A, B, A, B, B, D, B |
| A | 0 | A, **B**, C, B, B, C, A, B, A, B, B, D, B |
| C | 1 | A, B, **C**, B, B, C, A, B, A, B, B, D, B |
| C | 0 | A, B, C, **B**, B, C, A, B, A, B, B, D, B |
| B | 1 | A, B, C, B, **B**, C, A, B, A, B, B, D, B |
| B | 0 | A, B, C, B, B, **C**, A, B, A, B, B, D, B |
| A | 1 | A, B, C, B, B, C, **A**, B, A, B, B, D, B |
| A | 0 | A, B, C, B, B, C, A, **B**, A, B, B, D, B |
| A | 1 | A, B, C, B, B, C, A, B, **A**, B, B, D, B |
| A | 0 | A, B, C, B, B, C, A, B, A, **B**, B, D, B |
| B | 1 | A, B, C, B, B, C, A, B, A, B, **B**, D, B |
| B | 0 | A, B, C, B, B, C, A, B, A, B, B, **D**, B |
| B | 1 | A, B, C, B, B, C, A, B, A, B, B, D, **B** |

The key point is that, if there exits the majority greater than 50%, it can't be voted out by all the others. However, if there are not any candidates win more than half of the votes, the recorded 'winner' is invalid. Thus it is necessary to perform a second round scan for verification.

The following pseudo code illustrates this algorithm.

1: **function** MAJORITY($A$)
2: $c \leftarrow 0$
3: **for** $i \leftarrow 1$ to $|A|$ **do**
4: **if** $c = 0$ **then**
5: $x \leftarrow A[i]$
6: **if** $A[i] = x$ **then**
7: $c \leftarrow c + 1$
8: **else**
9: $c \leftarrow c - 1$
10: **return** $x$

If there is the majority element, this algorithm takes one pass to scan the votes. In every iteration, it either increases or decreases the counter according to the vote is support or against the current selection. If the counter becomes zero, it means the current selection is voted out. So the new one is selected as the updated candidate for further scan.

The process is linear $O(n)$ time, and the spaces needed are just two variables. One for recording the selected candidate so far, the other is for vote counting.

Although this algorithm can find the majority element if there is. it still picks an element even there isn't. The following modified algorithm verifies the final result with another round of scan.

1: **function** MAJORITY($A$)
2: $c \leftarrow 0$
3: **for** $i \leftarrow 1$ to $|A|$ **do**
4: **if** $c = 0$ **then**
5: $x \leftarrow A[i]$
6: **if** $A[i] = x$ **then**
7: $c \leftarrow c + 1$
8: **else**
9: $c \leftarrow c - 1$
10: $c \leftarrow 0$
11: **for** $i \leftarrow 1$ to $|A|$ **do**
12: **if** $A[i] = x$ **then**
13: $c \leftarrow c + 1$

14: **if** $c > \%50|A|$ **then**
15: **return** $x$
16: **else**
17: fail

Even with this verification process, the algorithm is still bound to $O(n)$ time, and the space needed is constant. The following ISO C++ program implements this algorithm [6].

```
template<typename T>
T majority(const T* xs, int n, T fail) {
    T m;
    int i, c;
    for (i = 0, c = 0; i < n; ++i) {
        if (!c)
            m = xs[i];
        c += xs[i] == m ? 1 : -1;
    }
    for (i = 0, c = 0; i < n; ++i, c += xs[i] == m);
    return c * 2 > n ? m : fail;
}
```

Boyer-Moore majority algorithm can also be realized in purely functional approach. Different from the imperative settings, which use variables to record and update information, accumulators are used to define the core algorithm. Define function $maj(c, n, L)$, which takes a list of votes $L$, a selected candidate $c$ so far, and a counter $n$. For non empty list $L$, we initialize $c$ as the first vote $l_1$, and set the counter as 1 to start the algorithm: $maj(l_1, 1, L')$, where $L'$ is the rest votes except for $l_1$. Below are the definition of this function.

$$maj(c,n,L) = \begin{cases} c & : L = \phi \\ maj(c, n+1, L') & : l_1 = c \\ maj(l_1, 1, L') & : n = 0 \land l_1 \neq c \\ maj(c, n-1, L') & : otherwise \end{cases} \tag{14.19}$$

We also need to define a function, which can verify the result. The idea is that, if the list of votes is empty, the final result is a failure; otherwise, we start the Boyer-Moore algorithm to find a candidate $c$, then we scan the list again to count the total votes $c$ wins, and verify if this number is not less than the half.

$$majority(L) = \begin{cases} fail & : L = \phi \\ c & : c = maj(l_1, 1, L'), |\{x | x \in L, x = c\}| > \%50|L| \\ fail & : otherwise \end{cases} \tag{14.20}$$

Below Haskell example code implements this algorithm.

```
majority :: (Eq a) => [a] -> Maybe a
majority [] = Nothing
majority (x:xs) = let m = maj x 1 xs in verify m (x:xs)

maj c n [] = c
maj c n (x:xs) | c == x = maj c (n+1) xs
               | n == 0 = maj x 1 xs
               | otherwise = maj c (n-1) xs

verify m xs = if 2 * (length $ filter (==m) xs) > length xs
              then Just m else Nothing
```

[6]We actually uses the ANSI C style. The C++ template is only used to generalize the type of the element

### Maximum sum of sub vector

Jon Bentley presents another interesting puzzle which can be solved by using quite similar idea in[4]. The problem is to find the maximum sum of sub vector. For example in the following array, The sub vector {19, -12, 1, 9, 18} yields the biggest sum 35.

| 3 | -13 | 19 | -12 | 1 | 9 | 18 | -16 | 15 | -15 |
|---|---|---|---|---|---|---|---|---|---|

Note that it is only required to output the value of the maximum sum. If all the numbers are positive, the answer is definitely the sum of all. Another special case is that all numbers are negative. We define the maximum sum is 0 for an empty sub vector.

Of course we can find the answer with brute-force, by calculating all sums of sub vectors and picking the maximum. Such naive method is typical quadratic.

1: **function** MAX-SUM($A$)
2: $m \leftarrow 0$
3: **for** $i \leftarrow 1$ to $|A|$ **do**
4: $s \leftarrow 0$
5: **for** $j \leftarrow i$ to $|A|$ **do**
6: $s \leftarrow s + A[j]$
7: $m \leftarrow$ MAX$(m, s)$
8: **return** $m$

The brute force algorithm does not reuse any information in previous search. Similar with Boyer-Moore majority vote algorithm, we can record the maximum sum end to the position where we are scanning. Of course we also need record the biggest sum found so far. The following figure illustrates this idea and the invariant during scan.

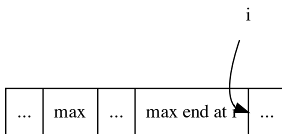



Figure 14.11: Invariant during scan.

At any time when we scan to the $i$-th position, the max sum found so far is recorded as $A$. At the same time, we also record the biggest sum end at $i$ as $B$. Note that $A$ and $B$ may not be the same, in fact, we always maintain $B \leq A$. and when $B$ becomes greater than $A$ by adding with the next element, we update $A$ with this new value. When $B$ becomes negative, this happens when the next element is a negative number, we reset it to 0. The following tables illustrated the steps when we scan the example vector $\{3, -13, 19, -12, 1, 9, 18, -16, 15, -15\}$.

| max sum | max end at $i$ | list to be scan |
|---|---|---|
| 0 | 0 | $\{3, -13, 19, -12, 1, 9, 18, -16, 15, -15\}$ |
| 3 | 3 | $\{-13, 19, -12, 1, 9, 18, -16, 15, -15\}$ |
| 3 | 0 | $\{19, -12, 1, 9, 18, -16, 15, -15\}$ |
| 19 | 19 | $\{-12, 1, 9, 18, -16, 15, -15\}$ |
| 19 | 7 | $\{1, 9, 18, -16, 15, -15\}$ |
| 19 | 8 | $\{9, 18, -16, 15, -15\}$ |
| 19 | 17 | $\{18, -16, 15, -15\}$ |
| 35 | 35 | $\{-16, 15, -15\}$ |
| 35 | 19 | $\{15, -15\}$ |
| 35 | 34 | $\{-15\}$ |
| 35 | 19 | $\{\}$ |

This algorithm can be described as below.

1: **function** MAX-SUM($V$)
2: $A \leftarrow 0, B \leftarrow 0$
3: **for** $i \leftarrow 1$ to $|V|$ **do**
4: $B \leftarrow$ MAX($B + V[i], 0$)
5: $A \leftarrow$ MAX($A, B$)

It is trivial to implement this linear time algorithm, that we skip the details here.

This algorithm can also be defined in functional approach. Instead of mutating variables, we use accumulator to record $A$ and $B$. In order to search the maximum sum of list $L$, we call the below function with $max_{sum}(0, 0, L)$.

$$max_{sum}(A, B, L) = \begin{cases} A & : L = \phi \\ max_{sum}(A', B', L') & : otherwise \end{cases} \tag{14.21}$$

Where

$$B' = max(l_1 + B, 0)$$
$$A' = max(A, B')$$

Below Haskell example code implements this algorithm.

```
maxsum = msum 0 0 where
  msum a _ [] = a
  msum a b (x:xs) = let b' = max (x+b) 0
                        a' = max a b'
                    in msum a' b' xs
```

### KMP

String matching is another important type of searching. Almost all the software editors are equipped with tools to find string in the text. In chapters about Trie, Patricia, and suffix tree, we have introduced some powerful data structures which can help to search string. In this section, we introduce another two string matching algorithms all based on information reusing.

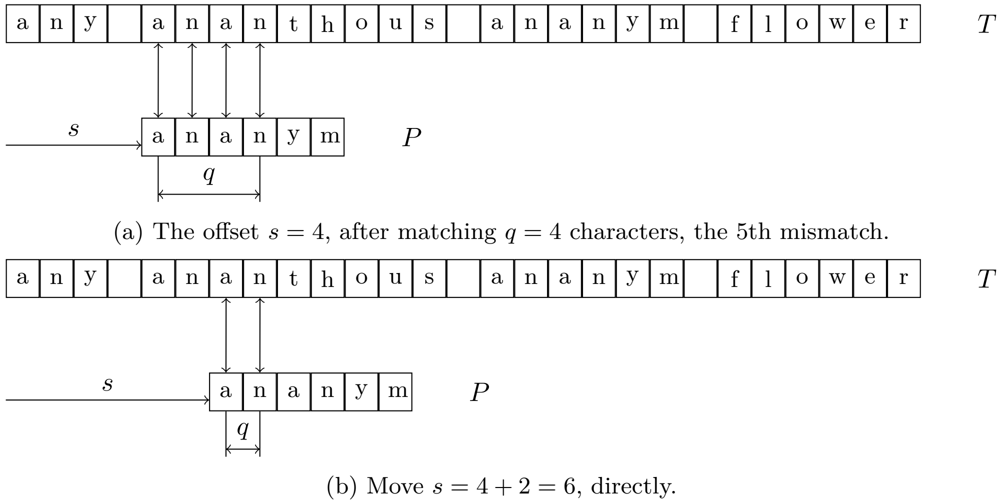
Some programming environments provide built-in string search tools, however, most of them are brute-force solution including 'strstr' function in ANSI C standard library, 'find' in C++ standard template library, 'indexOf' in Java Development Kit etc. Figure 14.12 illustrate how such character-by-character comparison process works.

Suppose we search a pattern $P$ in text $T$, as shown in figure 14.12 (a), at offset $s = 4$, the process examines every character in $P$ and $T$ to check if they are same. It successfully matches the first 4 characters 'anan'. However, the 5th character in the pattern string is 'y'. It doesn't match the corresponding character in the text, which is 't'.

At this stage, the brute-force solution terminates the attempt, increases $s$ by one to 5, and restart the comparison between 'ananym' and 'nantho...'. Actually, we can increase $s$ not only by one. This is because we have already known that the first four characters 'anan' have been matched, and the failure happens at the 5th position. Observe the two letters prefix 'an' of the pattern string is also a suffix of 'anan' that we have matched so far. A more effective way is to shift $s$ by two but not one, which is shown in figure 14.12 (b). By this means, we reused the information that 4 characters have been matched. This helps us to skip invalid positions as many as possible.

Knuth, Morris and Pratt presented this idea in[85] and developed a novel string matching algorithm. This algorithm is later called as 'KMP', which is consist of the three authors' initials.

For the sake of brevity, we denote the first $k$ characters of text $T$ as $T_k$. Which means $T_k$ is the $k$-character prefix of $T$.




(a) The offset $s = 4$, after matching $q = 4$ characters, the 5th mismatch.

(b) Move $s = 4 + 2 = 6$, directly.

Figure 14.12: Match 'ananym' in 'any ananthous ananym flower'.

The key point to shift $s$ effectively is to find a function of $q$, where $q$ is the number of characters matched successfully. For instance, $q$ is 4 in figure 14.12 (a), as the 5th character doesn't match.

Consider what situation we can shift $s$ more than 1. As shown in figure 14.13, if we can shift the pattern $P$ ahead, there must exist $k$, so that the first $k$ characters are as same as the last $k$ characters of $P_q$. In other words, the prefix $P_k$ is suffix of $P_q$.

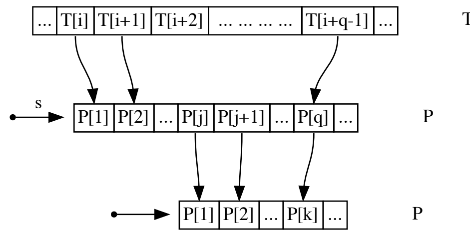



Figure 14.13: $P_k$ is both prefix of $P_q$ and suffix of $P_q$.

It's possible that there is no such a prefix that is the suffix at the same time. If we treat empty string as both the prefix and the suffix of any others, there must be at least one solution that $k = 0$. It's also quite possible that there are multiple $k$ satisfy. To avoid missing any possible matching positions, we have to find the biggest $k$. We can define a *prefix function* $\pi(q)$ which tells us where we can fallback if the $(q + 1)$-th character does not match[4].

$$\pi(q) = max\{k | k < q \wedge P_k \sqsupset P_q\} \tag{14.22}$$

Where $\sqsupset$ is read as 'is suffix of'. For instance, $A \sqsupset B$ means $A$ is suffix of $B$. This function is used as the following. When we match pattern $P$ against text $T$ from offset $s$, If it fails after matching $q$ characters, we next look up $\pi(q)$ to get a fallback $q'$, and retry to compare $P[q']$ with the previous unmatched character. Based on this idea, the core algorithm of KMP can be described as the following.

1: **function** KMP$(T, P)$

2: $n \leftarrow |T|, m \leftarrow |P|$
3: build prefix function $\pi$ from $P$
4: $q \leftarrow 0$ ▷ How many characters have been matched so far.
5: **for** $i \leftarrow 1$ to $n$ **do**
6: **while** $q > 0 \land P[q+1] \neq T[i]$ **do**
7: $q \leftarrow \pi(q)$
8: **if** $P[q+1] = T[i]$ **then**
9: $q \leftarrow q + 1$
10: **if** $q = m$ **then**
11: found one solution at $i - m$
12: $q \leftarrow \pi(q)$ ▷ look for next solution

Although the definition of prefix function $\pi(q)$ is given in equation (14.22), realizing it blindly by finding the longest suffix isn't effective. Actually we can use the idea of information reusing again to build the prefix function.

The trivial edge case is that, the first character doesn't match. In this case the longest prefix, which is also the suffix is definitely empty, so $\pi(1) = k = 0$. We record the longest prefix as $P_k$. In this edge case $P_k = P_0$ is the empty string.

After that, when we scan at the $q$-th character in the pattern string $P$, we hold the invariant that the prefix function values $\pi(i)$ for $i$ in $\{1, 2, ..., q-1\}$ have already been recorded, and $P_k$ is the longest prefix which is also the suffix of $P_{q-1}$. As shown in figure 14.14, if $P[q] = P[k+1]$, A bigger $k$ than before is found, we can increase the maximum of $k$ by one; otherwise, if they are not same, we can use $\pi(k)$ to fallback to a shorter prefix $P_{k'}$ where $k' = \pi(k)$, and check if the next character after this new prefix is same as the $q$-th character. We need repeat this step until either $k$ becomes zero (which means only empty string satisfies), or the $q$-th character matches.

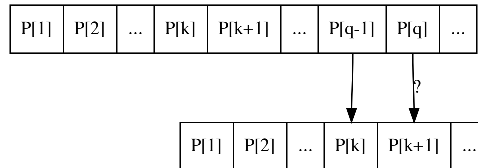



Figure 14.14: $P_k$ is suffix of $P_{q-1}$, $P[q]$ and $P[k+1]$ are compared.

Realizing this idea gives the KMP prefix building algorithm.

1: **function** BUILD-PREFIX-FUNCTION($P$)
2: $m \leftarrow |P|, k \leftarrow 0$
3: $\pi(1) \leftarrow 0$
4: **for** $q \leftarrow 2$ to $m$ **do**
5: **while** $k > 0 \land P[q] \neq P[k+1]$ **do**
6: $k \leftarrow \pi(k)$
7: **if** $P[q] = P[k+1]$ **then**
8: $k \leftarrow k + 1$
9: $\pi(q) \leftarrow k$
10: **return** $\pi$

The following table lists the steps of building prefix function for pattern string 'ananym'. Note that the $k$ in the table actually means the maximum $k$ satisfies equation (14.22).

| $q$ | $P_q$ | $k$ | $P_k$ |
|---|---|---|---|
| 1 | a | 0 | “” |
| 2 | an | 0 | “” |
| 3 | ana | 1 | a |
| 4 | anan | 2 | an |
| 5 | anany | 0 | “” |
| 6 | ananym | 0 | “” |

Translating the KMP algorithm to Python gives the below example code.

```
def kmp_match(w, p):
    n = len(w)
    m = len(p)
    fallback = fprefix(p)
    k = 0 # how many elements have been matched so far.
    res = []
    for i in range(n):
        while k > 0 and p[k] != w[i]:
            k = fallback[k] #fall back
        if p[k] == w[i]:
            k = k + 1
        if k == m:
            res.append(i+1-m)
            k = fallback[k-1] # look for next
    return res

def fprefix(p):
    m = len(p)
    t = [0]*m  # fallback table
    k = 0
    for i in range(2, m):
        while k>0 and p[i-1] != p[k]:
            k = t[k-1] #fallback
        if p[i-1] == p[k]:
            k = k + 1
        t[i] = k
    return t
```

The KMP algorithm builds the prefix function for the pattern string as a kind of pre-processing before the search. Because of this, it can reuse as much information of the previous matching as possible.

The amortized performance of building the prefix function is $O(m)$. This can be proved by using potential method as in[4]. Using the similar method, it can be proved that the matching algorithm itself is also linear. Thus the total performance is $O(m+n)$ at the expense of the $O(m)$ space to record the prefix function table.

It seems that varies pattern string would affect the performance of KMP. Considering the case that we are finding pattern string ‘aaa...a’ of length $m$ in a string ‘aaa...a’ of length $n$. All the characters are same, when the last character in the pattern is examined, we can only fallback by 1, and this 1 character fallback repeats until it falls back to zero. Even in this extreme case, KMP algorithm still holds its linear performance (why?). Please try to consider more cases such as $P = aaaa...b$, $T = aaaa...a$ and so on.

#### Purely functional KMP algorithm

It is not easy to realize KMP matching algorithm in purely functional manner. The imperative algorithm represented so far intensely uses array to record prefix function values. Although it is possible to utilize sequence like structure in purely functional settings, it is typically implemented with finger tree. Unlike native arrays, finger tree

needs logarithm time for random accessing[7].

Richard Bird presents a formal program deduction to KMP algorithm by using fold fusion law in chapter 17 of[1]. In this section, we show how to develop purely functional KMP algorithm step by step from a brute-force prefix function creation method.

Both text string and pattern are represented as singly linked-list in purely functional settings. During the scan process, these two lists are further partitioned, every one is broken into two parts. As shown in figure 14.15, The first $j$ characters in the pattern string have been matched. $T[i+1]$ and $P[j+1]$ will be compared next. If they are same, we need append the character to the matched part. However, since strings are essentially singly linked list, such appending is proportion to $j$.

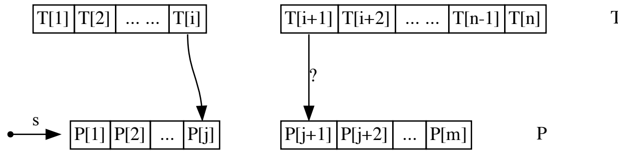



Figure 14.15: The first $j$ characters in $P$ are matched, next compare $P[j+1]$ with $T[i+1]$.

Denote the first $i$ characters as $T_p$, which means the prefix of $T$, the rest characters as $T_s$ for suffix; Similarly, the first $j$ characters as $P_p$, and the rest as $P_s$; Denote the first character of $T_s$ as $t$, the first character of $P_s$ as $p$. We have the following 'cons' relationship.

$$\begin{array}{l} T_s = cons(t, T'_s) \\ P_s = cons(p, P'_s) \end{array}$$

If $t = p$, note the following updating process is bound to linear time.

$$\begin{array}{l} T'_p = T_p \cup \{t\} \\ P'_p = P_p \cup \{p\} \end{array}$$

We've introduced a method in the chapter about purely functional queue, which can solve this problem. By using a pair of front and rear list, we can turn the linear time appending to constant time linking. The key point is to represent the prefix part in reverse order.

$$\begin{array}{l} T = T_p \cup T_s = reverse(reverse(T_p)) \cup T_s = reverse(\overleftarrow{T_p}) \cup T_s \\ P = P_p \cup P_s = reverse(reverse(P_p)) \cup P_s = reverse(\overleftarrow{P_p}) \cup P_s \end{array} \tag{14.23}$$

The idea is to using pair $(\overleftarrow{T_p}, T_s)$ and $(\overleftarrow{P_p}, P_s)$ instead. With this change, the if $t = p$, we can update the prefix part fast in constant time.

$$\begin{array}{l} \overleftarrow{T'_p} = cons(t, \overleftarrow{T_p}) \\ \overleftarrow{P'_p} = cons(p, \overleftarrow{P_p}) \end{array} \tag{14.24}$$

The KMP matching algorithm starts by initializing the success prefix parts to empty strings as the following.

$$search(P, T) = kmp(\pi, (\phi, P)(\phi, T)) \tag{14.25}$$

[7]Again, we don't use native array, even it is supported in some functional programming environments like Haskell.

Where $\pi$ is the prefix function we explained before. The core part of KMP algorithm, except for the prefix function building, can be defined as below.

$$kmp(\pi, (\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) = \begin{cases} \{|\overleftarrow{T_p}|\} & : P_s = \phi \land T_s = \phi \\ \phi & : P_s \neq \phi \land T_s = \phi \\ \{|\overleftarrow{T_p}\} \cup kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) & : P_s = \phi \land T_s \neq \phi \\ kmp(\pi, (\overleftarrow{P'_p}, P'_s), (\overleftarrow{T'_p}, T'_s)) & : t = p \\ kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T'_p}, T'_s)) & : t \neq p \land \overleftarrow{P_p} = \phi \\ kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) & : t \neq p \land \overleftarrow{P_p} \neq \phi \end{cases} \tag{14.26}$$

The first clause states that, if the scan successfully ends to both the pattern and text strings, we get a solution, and the algorithm terminates. Note that we use the right position in the text string as the matching point. It's easy to use the left position by subtracting with the length of the pattern string. For sake of brevity, we switch to right position in functional solutions.

The second clause states that if the scan arrives at the end of text string, while there are still rest of characters in the pattern string haven't been matched, there is no solution. And the algorithm terminates.

The third clause states that, if all the characters in the pattern string have been successfully matched, while there are still characters in the text haven't been examined, we get a solution, and we fallback by calling prefix function $\pi$ to go on searching other solutions.

The fourth clause deals with the case, that the next character in pattern string and text are same. In such case, the algorithm advances one character ahead, and recursively performs searching.

If the the next characters are not same and this is the first character in the pattern string, we just need advance to next character in the text, and try again. Otherwise if this isn't the first character in the pattern, we call prefix function $\pi$ to fallback, and try again.

The brute-force way to build the prefix function is just to follow the definition equation (14.22).

$$\pi(\overleftarrow{P_p}, P_s) = (\overleftarrow{P'_p}, P'_s) \tag{14.27}$$

where

$$\begin{array}{l} P'_p = longest(\{s | s \in prefixes(P_p), s \sqsupset P_p\}) \\ P'_s = P - P'_p \end{array}$$

Every time when calculate the fallback position, the algorithm naively enumerates all prefixes of $P_p$, checks if it is also the suffix of $P_p$, and then pick the longest one as result. Note that we reuse the subtraction symbol here for list differ operation.

There is a tricky case which should be avoided. Because any string itself is both its prefix and suffix. Say $P_p \sqsubset P_p$ and $P_p \sqsupset P_p$. We shouldn't enumerate $P_p$ as a candidate prefix. One solution of such prefix enumeration can be realized as the following.

$$prefixes(L) = \begin{cases} \{\phi\} & : L = \phi \lor |L| = 1 \\ cons(\phi, map(\lambda_s \cdot cons(l_1, s), prefixes(L'))) & : otherwise \end{cases} \tag{14.28}$$

Below Haskell example program implements this version of string matching algorithm.

```
kmpSearch1 ptn text = kmpSearch' next ([], ptn) ([], text)
```

```
kmpSearch' _ (sp, []) (sw, []) = [length sw]
kmpSearch' _ _ (_, []) = []
kmpSearch' f (sp, []) (sw, ws) = length sw : kmpSearch' f (f sp []) (sw, ws)
kmpSearch' f (sp, (p:ps)) (sw, (w:ws))
    | p == w = kmpSearch' f ((p:sp), ps) ((w:sw), ws)
    | otherwise = if sp ==[] then kmpSearch' f (sp, (p:ps)) ((w:sw), ws)
                  else kmpSearch' f (f sp (p:ps)) (sw, (w:ws))

next sp ps = (sp', ps') where
    prev = reverse sp
    prefix = longest [xs | xs ← inits prev, xs `isSuffixOf` prev]
    sp' = reverse prefix
    ps' = (prev ++ ps) \\ prefix
    longest = maximumBy (compare `on` length)

inits [] = [[]]
inits [_] = [[]]
inits (x:xs) = [] : (map (x:) $ inits xs)
```

This version does not only perform poorly, but it is also complex. We can simplify it a bit. Observing the KMP matching is a scan process from left to the right of the text, it can be represented with folding (refer to Appendix A for detail). Firstly, we can augment each character with an index for folding like below.

$$zip(T, \{1, 2, ...\}) \tag{14.29}$$

Zipping the text string with infinity natural numbers gives list of pairs. For example, text string 'The quick brown fox jumps over the lazy dog' turns into (T, 1), (h, 2), (e, 3), ... (o, 42), (g, 43).

The initial state for folding contains two parts, one is the pair of pattern $(P_p, P_s)$, with prefix starts from empty, and the suffix is the whole pattern string $(\phi, P)$. For illustration purpose only, we revert back to normal pairs but not $(\overleftarrow{P_p}, P_s)$ notation. It can be easily replaced with reversed form in the finalized version. This is left as exercise to the reader. The other part is a list of positions, where the successful matching are found. It starts from empty list. After the folding finishes, this list contains all solutions. What we need is to extract this list from the final state. The core KMP search algorithm is simplified like this.

$$kmp(P, T) = snd(fold(search, ((\phi, P), \phi), zip(T, \{1, 2, ...\}))) \tag{14.30}$$

The only 'black box' is the *search* function, which takes a state, and a pair of character and index, and it returns a new state as result. Denote the first character in $P_s$ as $p$ and the rest characters as $P'_s$ ($P_s = cons(p, P'_s)$), we have the following definition.

$$search(((P_p, P_s), L), (c, i)) = \begin{cases} ((P_p \cup p, P'_s), L \cup \{i\}) & : \quad p = c \land P'_s = \phi \\ ((P_p \cup p, P'_s), L) & : \quad p = c \land P'_s \neq \phi \\ ((P_p, P_s), L) & : \quad P_p = \phi \\ search((\pi(P_p, P_s), L), (c, i)) & : \quad otherwise \end{cases} \tag{14.31}$$

If the first character in $P_s$ matches the current character $c$ during scan, we need further check if all the characters in the pattern have been examined, if so, we successfully find a solution, This position $i$ in list $L$ is recorded; Otherwise, we advance one character ahead and go on. If $p$ does not match $c$, we need fallback for further retry. However, there is an edge case that we can't fallback any more. $P_p$ is empty in this case, and we need do nothing but keep the current state.

The prefix-function $\pi$ developed so far can also be improved a bit. Since we want to find the longest prefix of $P_p$, which is also suffix of it, we can scan from right to left

instead. For any non empty list $L$, denote the first element as $l_1$, and all the rest except for the first one as $L'$, define a function $init(L)$, which returns all the elements except for the last one as below.

$$init(L) = \begin{cases} \phi & : |L| = 1 \\ cons(l_1, init(L')) & : otherwise \end{cases} \tag{14.32}$$

Note that this function can not handle empty list. The idea of scan from right to left for $P_p$ is first check if $init(P_p) \sqsupset P_p$, if yes, then we are done; otherwise, we examine if $init(init(P_p))$ is OK, and repeat this till the left most. Based on this idea, the prefix-function can be modified as the following.

$$\pi(P_p, P_s) = \begin{cases} (P_p, P_s) & : P_p = \phi \\ fallback(init(P_p), cons(last(P_p), P_s)) & : otherwise \end{cases} \tag{14.33}$$

Where

$$fallback(A, B) = \begin{cases} (A, B) & : A \sqsupset P_p \\ (init(A), cons(last(A), B)) & : otherwise \end{cases} \tag{14.34}$$

Note that fallback always terminates because empty string is suffix of any string. The $last(L)$ function returns the last element of a list, it is also a linear time operation (refer to Appendix A for detail). However, it's constant operation if we use $\overleftarrow{P_p}$ approach. This improved prefix-function is bound to linear time. It is still quite slower than the imperative algorithm which can look up prefix-function in constant $O(1)$ time. The following Haskell example program implements this minor improvement.

```
failure ([], ys) = ([], ys)
failure (xs, ys) = fallback (init xs) (last xs:ys) where
    fallback as bs | as `isSuffixOf` xs = (as, bs)
                   | otherwise = fallback (init as) (last as:bs)

kmpSearch ws txt = snd $ foldl f (([], ws), []) (zip txt [1..]) where
    f (p@(xs, (y:ys)), ns) (x, n) | x == y = if ys==[] then ((xs++[y], ys), ns++[n])
                                             else ((xs++[y], ys), ns)
                                  | xs == [] = (p, ns)
                                  | otherwise = f (failure p, ns) (x, n)
    f (p, ns) e = f (failure p, ns) e
```

The bottleneck is that we can not use native array to record prefix functions in purely functional settings. In fact the prefix function can be understood as a state transform function. It transfer from one state to the other according to the matching is success or fail. We can abstract such state changing as a tree. In environment supporting algebraic data type, Haskell for example, such state tree can be defined like below.

```
data State a = E | S a (State a) (State a)
```

A state is either empty, or contains three parts: the current state, the new state if match fails, and the new state if match succeeds. Such definition is quite similar to the binary tree. We can call it 'left-fail, right-success' tree. The state we are using here is $(P_p, P_s)$.

Similar as imperative KMP algorithm, which builds the prefix function from the pattern string, the state transforming tree can also be built from the pattern. The idea is to build the tree from the very beginning state $(\phi, P)$, with both its children empty. We replace the left child with a new state by calling $\pi$ function defined above, and replace the right child by advancing one character ahead. There is an edge case, that when the

state transfers to $(P, \phi)$, we can not advance any more in success case, such node only contains child for failure case. The build function is defined as the following.

$$build((P_p, P_s), \phi, \phi) = \begin{cases} build(\pi(P_p, P_s), \phi, \phi) & : & P_s = \phi \\ build((P_p, P_s), L, R) & : & otherwise \end{cases} \tag{14.35}$$

Where

$$\begin{array}{l} L = build(\pi(P_p, P_s), \phi, \phi) \\ R = build((P_s \cup \{p\}, P_s'), \phi, \phi)) \end{array}$$

The meaning of $p$ and $P_s'$ are as same as before, that $p$ is the first character in $P_s$, and $P_s'$ is the rest characters. The most interesting point is that the build function will never stop. It endless build a infinite tree. In strict programming environment, calling this function will freeze. However, in environments support lazy evaluation, only the nodes have to be used will be created. For example, both Haskell and Scheme/Lisp are capable to construct such infinite state tree. In imperative settings, it is typically realized by using pointers which links to ancestor of a node.

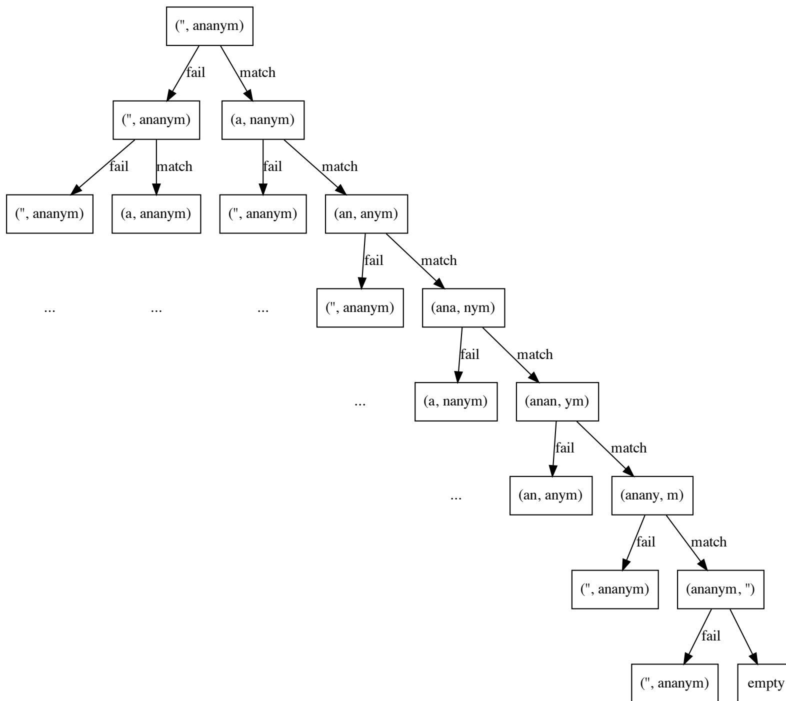



Figure 14.16: The infinite state tree for pattern 'ananym'.

Figure 14.16 illustrates such an infinite state tree for pattern string 'ananym'. Note that the right most edge represents the case that the matching continuously succeed for all characters. After that, since we can't match any more, so the right sub-tree is empty. Base on this fact, we can define a auxiliary function to test if a state indicates the whole pattern is successfully matched.

$$match((P_p, P_s), L, R) = \begin{cases} True & : & P_s = \phi \\ False & : & otherwise \end{cases} \tag{14.36}$$

With the help of state transform tree, we can realize KMP algorithm in an automaton manner.

$$kmp(P, T) = snd(fold(search, (Tr, []), zip(T, \{1, 2, ...\}))) \tag{14.37}$$

Where the tree $Tr = build((\phi, P), \phi, \phi)$ is the infinite state transform tree. Function *search* utilizes this tree to transform the state according to match or fail. Denote the first character in $P_s$ as $p$, the rest characters as $P'_s$, and the matched positions found so far as $A$.

$$search((((P_p, P_s), L, R), A), (c, i)) = \begin{cases} (R, A \cup \{i\}) & : p = c \land match(R) \\ (R, A) & : p = c \land \neg match(R) \\ ((((P_p, P_s), L, R), A) & : P_p = \phi \\ search((L, A), (c, i)) & : otherwise \end{cases} \tag{14.38}$$

The following Haskell example program implements this algorithm.

```
data State a = E | S a (State a) (State a) -- state, ok-state, fail-state
               deriving (Eq, Show)

build :: (Eq a)⇒State ([a], [a]) → State ([a], [a])
build (S s@(xs, []) E E) = S s (build (S (failure s) E E)) E
build (S s@(xs, (y:ys)) E E) = S s l r where
    l = build (S (failure s) E E) -- fail state
    r = build (S (xs++[y], ys) E E)

matched (S (_, []) _ _) = True
matched _ = False

kmpSearch3 :: (Eq a) ⇒ [a] → [a] → [Int]
kmpSearch3 ws txt = snd $ foldl f (auto, []) (zip txt [1..]) where
    auto = build (S ([], ws) E E)
    f (s@(S (xs, ys) l r), ns) (x, n)
        | [x] `isPrefixOf` ys = if matched r then (r, ns++[n])
                                else (r, ns)
        | xs == [] = (s, ns)
        | otherwise = f (l, ns) (x, n)
```

The bottle-neck is that the state tree building function calls $\pi$ to fallback. While current definition of $\pi$ isn't effective enough, because it enumerates all candidates from right to the left every time.

Since the state tree is infinite, we can adopt some common treatment for infinite structures. One good example is the Fibonacci series. The first two Fibonacci numbers are defined as 0 and 1; the rest Fibonacci numbers can be obtained by adding the previous two numbers.

$$\begin{array}{l} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{array} \tag{14.39}$$

Thus the Fibonacci numbers can be list one by one as the following

$$\begin{array}{l} F_0 = 0 \\ F_1 = 1 \\ F_2 = F_1 + F_0 \\ F_3 = F_2 + F_1 \\ ... \end{array} \tag{14.40}$$

We can collect all numbers in both sides, and define $F = \{0, 1, F_1, F_2, ...\}$, Thus we have the following equation.

$$\begin{array}{rl} F = & \{0, 1, F_1 + F_0, F_2 + F_1, ...\} \\ = & \{0, 1\} \cup \{x + y | x \in \{F_0, F_1, F_2, ...\}, y \in \{F_1, F_2, F_3, ...\}\} \\ = & \{0, 1\} \cup \{x + y | x \in F, y \in F'\} \end{array} \tag{14.41}$$

Where $F' = tail(F)$ is all the Fibonacci numbers except for the first one. In environments support lazy evaluation, like Haskell for instance, this definition can be expressed like below.

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

The recursive definition for infinite Fibonacci series indicates an idea which can be used to get rid of the fallback function $\pi$. Denote the state transfer tree as $T$, we can define the transfer function when matching a character on this tree as the following.

$$trans(T, c) = \begin{cases} root & : T = \phi \\ R & : T = ((P_p, P_s), L, R), c = p \\ trans(L, c) & : otherwise \end{cases} \tag{14.42}$$

If we match a character against empty node, we transfer to the root of the tree. We'll define the root later soon. Otherwise, we compare if the character $c$ is as same as the first character $p$ in $P_s$. If they match, then we transfer to the right sub tree for this success case; otherwise, we transfer to the left sub tree for fail case.

With transfer function defined, we can modify the previous tree building function accordingly. This is quite similar to the previous Fibonacci series definition.

$$build(T, (P_p, P_s)) = ((P_p, P_s), T, build(trans(T, p), (P_p \cup \{p\}, P_s')))$$

The right hand of this equation contains three parts. The first one is the state that we are matching $(P_p, P_s)$; If the match fails, Since $T$ itself can handle any fail case, we use it directly as the left sub tree; otherwise we recursive build the right sub tree for success case by advancing one character ahead, and calling transfer function we defined above.

However, there is an edge case which has to be handled specially, that if $P_s$ is empty, which indicates a successful match. As defined above, there isn't right sub tree any more. Combining these cases gives the final building function.

$$build(T, (P_p, P_s)) = \begin{cases} ((P_p, P_s), T, \phi) & : P_s = \phi \\ ((P_p, P_s), T, build(trans(T, p), (P_p \cup \{p\}, P_s'))) & : otherwise \end{cases} \tag{14.43}$$

The last brick is to define the root of the infinite state transfer tree, which initializes the building.

$$root = build(\phi, (\phi, P)) \tag{14.44}$$

And the new KMP matching algorithm is modified with this root.

$$kmp(P, T) = snd(fold(trans, (root, []), zip(T, \{1, 2, ...\}))) \tag{14.45}$$

The following Haskell example program implements this final version.

```
kmpSearch ws txt = snd $ foldl tr (root, []) (zip txt [1..]) where
    root = build' E ([], ws)
    build' fails (xs, []) = S (xs, []) fails E
    build' fails s@(xs, (y:ys)) = S s fails succs where
        succs = build' (fst (tr (fails, []) (y, 0))) (xs++[y], ys)
    tr (E, ns) _ = (root, ns)
    tr ((S (xs, ys) fails succs), ns) (x, n)
        | [x] `isPrefixOf` ys = if matched succs then (succs, ns++[n]) else (succs, ns)
        | otherwise = tr (fails, ns) (x, n)
```

Figure 14.17 shows the first 4 steps when search 'anaym' in text 'anal'. Since the first 3 steps all succeed, so the left sub trees of these 3 states are not actually constructed. They are marked as '?'. In the fourth step, the match fails, thus the right sub tree needn't

be built. On the other hand, we must construct the left sub tree, which is on top of the result of $trans(right(right(right(T))), n)$, where function $right(T)$ returns the right sub tree of $T$. This can be further expanded according to the definition of building and state transforming functions till we get the concrete state $((a, nanym), L, R)$. The detailed deduce process is left as exercise to the reader.

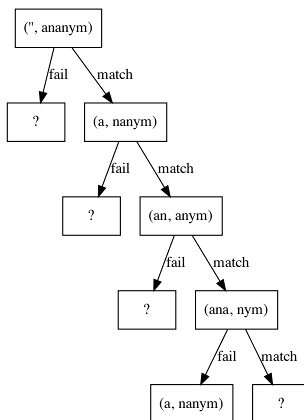



Figure 14.17: On demand construct the state transform tree when searching 'ananym' in text 'anal'.

This algorithm depends on the lazy evaluation critically. All the states to be transferred are built on demand. So that the building process is amortized $O(m)$, and the total performance is amortized $O(n+m)$. Readers can refer to[1] for detailed proof of it.

It's worth of comparing the final purely functional and the imperative algorithms. In many cases, we have expressive functional realization, however, for KMP matching algorithm, the imperative approach is much simpler and more intuitive. This is because we have to mimic the raw array by a infinite state transfer tree.

### Boyer-Moore

Boyer-Moore string matching algorithm is another effective solution invited in 1977[86]. The idea of Boyer-Moore algorithm comes from the following observation.

### The bad character heuristics

When attempt to match the pattern, even if there are several characters from the left are same, it fails if the last one does not match, as shown in figure 14.18. What's more, we wouldn't find a match even if we slide the pattern down by 1, or 2. Actually, the length of the pattern 'ananym' is 6, the last character is 'm', however, the corresponding character in the text is 'h'. It does not appear in the pattern at all. We can directly slide the pattern down by 6.

This leads to *the bad-character rule*. We can do a pre-processing for the pattern. If the character set of the text is already known, we can find all characters which don't appear in the pattern string. During the later scan process, as long as we find such a bad character, we can immediately slide the pattern down by its length. The question is what if the unmatched character does appear in the pattern? While, in order not to miss any potential matches, we have to slide down the pattern to check again. This is shown as in the figure 14.19

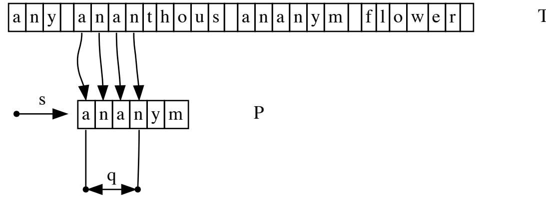
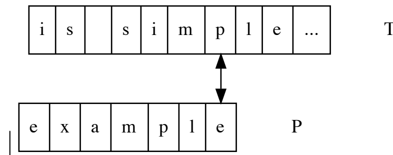
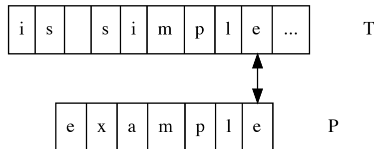



Figure 14.18: Since character ‘h’ doesn’t appear in the pattern, we wouldn’t find a match if we slide the pattern down less than the length of the pattern.




(a) The last character in the pattern ‘e’ doesn’t match ‘p’. However, ‘p’ appears in the pattern.




(b) We have to slide the pattern down by 2 to check again.

Figure 14.19: Slide the pattern if the unmatched character appears in the pattern.

It's quite possible that the unmatched character appears in the pattern more than one position. Denote the length of the pattern as $|P|$, the character appears in positions $p_1, p_2, ..., p_i$. In such case, we take the right most one to avoid missing any matches.

$$s = |P| - p_i \tag{14.46}$$

Note that the shifting length is 0 for the last position in the pattern according to the above equation. Thus we can skip it in realization. Another important point is that since the shifting length is calculated against the position aligned with the last character in the pattern string, (we deduce it from $|P|$), no matter where the mismatching happens when we scan from right to the left, we slide down the pattern string by looking up the bad character table with the one in the text aligned with the last character of the pattern. This is shown in figure 14.20.

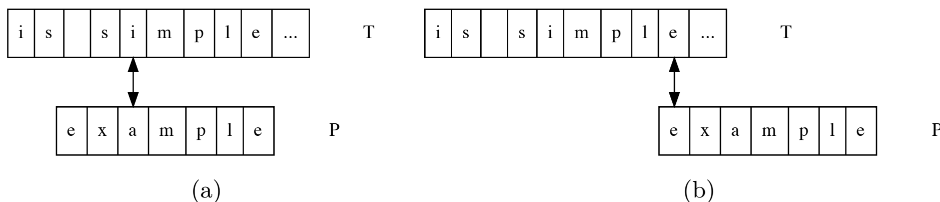



Figure 14.20: Even the mismatching happens in the middle, between char 'i' and 'a', we look up the shifting value with character 'e', which is 6 (calculated from the first 'e', the second 'e' is skipped to avoid zero shifting).

There is a good result in practice, that only using the bad-character rule leads to a simple and fast string matching algorithm, called Boyer-Moore-Horspool algorithm[87].

```
1: procedure BOYER-MOORE-HORSPOOL(T, P)
2:     for ∀c ∈ Σ do
3:         π[c] ← |P|
4:     for i ← 1 to |P| − 1 do                    ▷ Skip the last position
5:         π[P[i]] ← |P| − i
6:     s ← 0
7:     while s + |P| ≤ |T| do
8:         i ← |P|
9:         while i ≥ 1 ∧ P[i] = T[s + i] do         ▷ scan from right
10:            i ← i − 1
11:        if i < 1 then
12:            found one solution at s
13:            s ← s + 1                             ▷ go on finding the next
14:        else
15:            s ← s + π[T[s + |P|]]
```

The character set is denoted as $\Sigma$, we first initialize all the values of sliding table $\pi$ as the length of the pattern string $|P|$. After that we process the pattern from left to right, update the sliding value. If a character appears multiple times in the pattern, the latter value, which is on the right hand, will overwrite the previous value. We start the matching scan process by aligning the pattern and the text string from the very left. However, for every alignment $s$, we scan from the right to the left until either there is unmatched character or all the characters in the pattern have been examined. The latter case indicates that we've found a match; while for the former case, we look up $\pi$ to slide the pattern down to the right.

The following example Python code implements this algorithm accordingly.

```
def bmh_match(w, p):
    n = len(w)
    m = len(p)
    tab = [m for _ in range(256)]  # table to hold the bad character rule.
    for i in range(m-1):
        tab[ord(p[i])] = m - 1 - i
    res = []
    offset = 0
    while offset + m ≤ n:
        i = m - 1
        while i ≥ 0 and p[i] == w[offset+i]:
            i = i - 1
        if i < 0:
            res.append(offset)
            offset = offset + 1
        else:
            offset = offset + tab[ord(w[offset + m - 1])]
    return res
```

The algorithm firstly takes about $O(|\Sigma| + |P|)$ time to build the sliding table. If the character set size is small, the performance is dominated by the pattern and the text. There is definitely the worst case that all the characters in the pattern and text are same, e.g. searching 'aa...a' ($m$ of 'a', denoted as $a^m$) in text 'aa......a' ($n$ of 'a', denoted as $a^n$). The performance in the worst case is $O(mn)$. This algorithm performs well if the pattern is long, and there are constant number of matching. The result is bound to linear time. This is as same as the best case of full Boyer-Moore algorithm which will be explained next.

### The good suffix heuristics

Consider searching pattern 'abbabab' in text 'bbbababbabab...' like figure 14.21. By using the bad-character rule, the pattern will be slided by two.

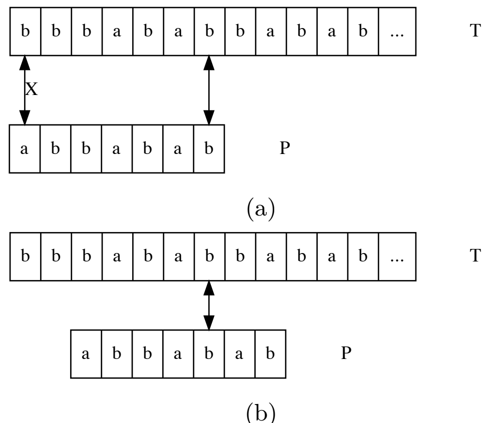



Figure 14.21: According to the bad-character rule, the pattern is slided by 2, so that the next 'b' is aligned.

Actually, we can do better than this. Observing that before the unmatched point, we have already successfully matched 6 characters 'bbabab' from right to the left. Since 'ab', which is the prefix of the pattern is also the suffix of what we matched so far, we can directly slide the pattern to align this suffix as shown in figure 14.22.

This is quite similar to the pre-processing of KMP algorithm, However, we can't always skip so many characters. Consider the following example as shown in figure 14.23. We

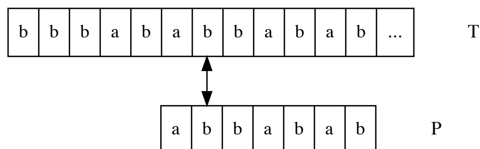



Figure 14.22: As the prefix 'ab' is also the suffix of what we've matched, we can slide down the pattern to a position so that 'ab' are aligned.

have matched characters 'bab' when the unmatch happens. Although the prefix 'ab' of the pattern is also the suffix of 'bab', we can't slide the pattern so far. This is because 'bab' appears somewhere else, which starts from the 3rd character of the pattern. In order not to miss any potential matching, we can only slide the pattern by two.

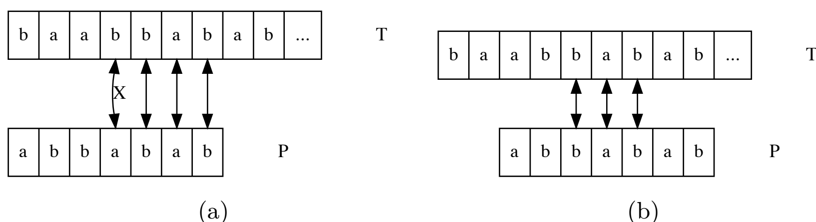



Figure 14.23: We've matched 'bab', which appears somewhere else in the pattern (from the 3rd to the 5th character). We can only slide down the pattern by 2 to avoid missing any potential matching.

The above situation forms the two cases of *the good-suffix rule*, as shown in figure 14.24.

Both cases in good suffix rule handle the situation that there are multiple characters have been matched from right. We can slide the pattern to the right if any of the the following happens.

- Case 1 states that if a part of the matching suffix occurs as a prefix of the pattern, and the matching suffix doesn't appear in any other places in the pattern, we can slide the pattern to the right to make this prefix aligned;
- Case 2 states that if the matching suffix occurs some where else in the pattern, we can slide the pattern to make the right most occurrence aligned.

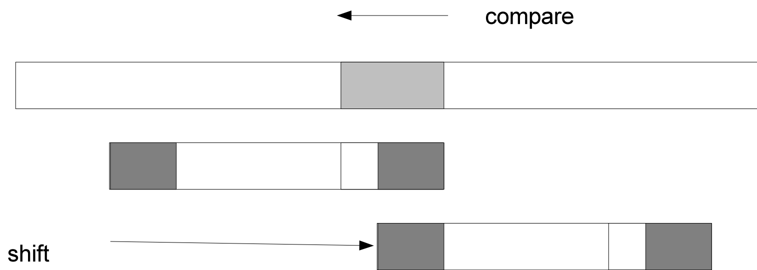
Note that in the scan process, we should apply case 2 first whenever it is possible, and then examine case 1 if the whole matched suffix does not appears in the pattern. Observe that both cases of the good-suffix rule only depend on the pattern string, a table can be built by pre-process the pattern for further looking up.

For the sake of brevity, we denote the suffix string from the $i$-th character of $P$ as $\overline{P_i}$. That $\overline{P_i}$ is the sub-string $P[i]P[i+1]...P[m]$.

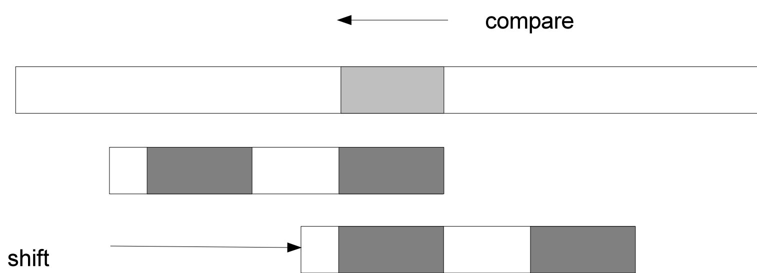
For case 1, we can check every suffix of $P$, which includes $\overline{P_m}$, $\overline{P_{m-1}}$, $\overline{P_{m-2}}$, ..., $\overline{P_2}$ to examine if it is the prefix of $P$. This can be achieved by a round of scan from right to the left.

For case 2, we can check every prefix of $P$ includes $P_1$, $P_2$, ..., $P_{m-1}$ to examine if the longest suffix is also a suffix of $P$. This can be achieved by another round of scan from left to the right.

1: **function** GOOD-SUFFIX($P$)
2: $\quad m \leftarrow |P|$




(a) Case 1, Only a part of the matching suffix occurs as a prefix of the pattern.




(b) Case 2, The matching suffix occurs some where else in the pattern.

Figure 14.24: The light gray section in the text represents the characters have been matched; The dark gray parts indicate the same content in the pattern.

3: $\pi_s \leftarrow \{0, 0, ..., 0\}$ ▷ Initialize the table of length $m$
4: $l \leftarrow 0$ ▷ The last suffix which is also prefix of $P$
5: **for** $i \leftarrow m - 1$ down-to 1 **do** ▷ First loop for case 1
6: **if** $\overline{P_i} \sqsubset P$ **then** ▷ $\sqsubset$ means 'is prefix of'
7: $l \leftarrow i$
8: $\pi_s[i] \leftarrow l$
9: **for** $i \leftarrow 1$ to $m$ **do** ▷ Second loop for case 2
10: $s \leftarrow$ Suffix-Length($P_i$)
11: **if** $s \neq 0 \land P[i - s] \neq P[m - s]$ **then**
12: $\pi_s[m - s] \leftarrow m - i$
13: **return** $\pi_s$

This algorithm builds the good-suffix heuristics table $\pi_s$. It first checks every suffix of $P$ from the shortest to the longest. If the suffix $\overline{P_i}$ is also the prefix of $P$, we record this suffix, and use it for all the entries until we find another suffix $\overline{P_j}$, $j < i$, and it is also the prefix of $P$.

After that, the algorithm checks every prefix of $P$ from the shortest to the longest. It calls the function Suffix-Length($P_i$), to calculate the length of the longest suffix of $P_i$, which is also suffix of $P$. If this length $s$ isn't zero, which means there exists a sub-string, that appears as the suffix of the pattern. It indicates that case 2 happens. The algorithm overwrites the $s$-th entry from the right of the table $\pi_s$. Note that to avoid finding the same occurrence of the matched suffix, we test if $P[i - s]$ and $P[m - s]$ are same.

Function Suffix-Length is designed as the following.

1: **function** Suffix-Length($P_i$)
2: $m \leftarrow |P|$
3: $j \leftarrow 0$
4: **while** $P[m - j] = P[i - j] \land j < i$ **do**
5: $j \leftarrow j + 1$
6: **return** $j$

The following Python example program implements the good-suffix rule.

```
def good_suffix(p):
    m = len(p)
    tab = [0 for _ in range(m)]
    last = 0
    # first loop for case 1
    for i in range(m-1, 0, -1): # m-1, m-2, ..., 1
        if is_prefix(p, i):
            last = i
        tab[i - 1] = last
    # second loop for case 2
    for i in range(m):
        slen = suffix_len(p, i)
        if slen != 0 and p[i - slen] != p[m - 1 - slen]:
            tab[m - 1 - slen] = m - 1 - i
    return tab

# test if p[i..m-1] `is prefix of` p
def is_prefix(p, i):
    for j in range(len(p) - i):
        if p[j] != p [i+j]:
            return False
    return True

# length of the longest suffix of p[..i], which is also a suffix of p
def suffix_len(p, i):
    m = len(p)
```

```
    j = 0
    while p[m - 1 - j] == p[i - j] and j < i:
        j = j + 1
    return j
```

It's quite possible that both the bad-character rule and the good-suffix rule can be applied when the unmatch happens. The Boyer-Moore algorithm compares and picks the bigger shift so that it can find the solution as quick as possible. The bad-character rule table can be explicitly built as below

1: **function** BAD-CHARACTER($P$)
2: **for** $\forall c \in \Sigma$ **do**
3: $\pi_b[c] \leftarrow |P|$
4: **for** $i \leftarrow 1$ to $|P| - 1$ **do**
5: $\pi_b[P[i]] \leftarrow |P| - i$
6: **return** $\pi_b$

The following Python program implements the bad-character rule accordingly.

```
def bad_char(p):
    m = len(p)
    tab = [m for _ in range(256)]
    for i in range(m-1):
        tab[ord(p[i])] = m - 1 - i
    return tab
```

The final Boyer-Moore algorithm firstly builds the two rules from the pattern, then aligns the pattern to the beginning of the text and scans from right to the left for every alignment. If any unmatch happens, it tries both rules, and slides the pattern with the bigger shift.

1: **function** BOYER-MOORE($T, P$)
2: $n \leftarrow |T|, m \leftarrow |P|$
3: $\pi_b \leftarrow$ BAD-CHARACTER($P$)
4: $\pi_s \leftarrow$ GOOD-SUFFIX($P$)
5: $s \leftarrow 0$
6: **while** $s + m \leq n$ **do**
7: $i \leftarrow m$
8: **while** $i \geq 1 \land P[i] = T[s + i]$ **do**
9: $i \leftarrow i - 1$
10: **if** $i < 1$ **then**
11: found one solution at $s$
12: $s \leftarrow s + 1$ ▷ go on finding the next
13: **else**
14: $s \leftarrow s + max(\pi_b[T[s + m]], \pi_s[i])$

Here is the example implementation of Boyer-Moore algorithm in Python.

```
def bm_match(w, p):
    n = len(w)
    m = len(p)
    tab1 = bad_char(p)
    tab2 = good_suffix(p)
    res = []
    offset = 0
    while offset + m ≤ n:
        i = m - 1
        while i ≥ 0 and p[i] == w[offset + i]:
            i = i - 1
        if i < 0:
            res.append(offset)
```

```
            offset = offset + 1
        else:
            offset = offset + max(tab1[ord(w[offset + m - 1])], tab2[i])
    return res
```

The Boyer-Moore algorithm published in original paper is bound to $O(n+m)$ in worst case only if the pattern doesn't appear in the text[86]. Knuth, Morris, and Pratt proved this fact in 1977[88]. However, when the pattern appears in the text, as we shown above, Boyer-Moore performs $O(nm)$ in the worst case.

Richard Birds shows a purely functional realization of Boyer-Moore algorithm in chapter 16 in[1]. We skipped it in this book.

## Exercise 14.2

- Proof that Boyer-Moore majority vote algorithm is correct.
- Given a list, find the element occurs most. Are there any divide and conqueror solutions? Are there any divide and conqueror data structures, such as map can be used?
- How to find the elements occur more than 1/3 in a list? How to find the elements occur more than 1/m in the list?
- If we reject the empty array as valid sub-array, how to realize the maximum sum of sub-arrays puzzle?
- Bentley presents a divide and conquer algorithm to find the maximum sum in $O(n \log n)$ time in[4]. The idea is to split the list at the middle point. We can recursively find the maximum sum in the first half and second half; However, we also need to find maximum sum cross the middle point. The method is to scan from the middle point to both ends as the following.

  1: **function** MAX-SUM($A$)
  2:     **if** $A = \phi$ **then**
  3:         **return** 0
  4:     **else if** $|A| = 1$ **then**
  5:         **return** MAX($0, A[1]$)
  6:     **else**
  7:         $m \leftarrow \lfloor \frac{|A|}{2} \rfloor$
  8:         $a \leftarrow$ MAX-FROM(REVERSE($A[1...m]$))
  9:         $b \leftarrow$ MAX-FROM($A[m+1...|A|]$)
  10:         $c \leftarrow$ MAX-SUM($A[1...m]$)
  11:         $d \leftarrow$ MAX-SUM($A[m+1...|A|$)
  12:         **return** MAX($a+b, c, d$)

  13: **function** MAX-FROM($A$)
  14:     $sum \leftarrow 0, m \leftarrow 0$
  15:     **for** $i \leftarrow 1$ to $|A|$ **do**
  16:         $sum \leftarrow sum + A[i]$
  17:         $m \leftarrow$ MAX($m, sum$)
  18:     **return** $m$

  It's easy to deduce the time performance is $T(n) = 2T(n/2) + O(n)$. Implement this algorithm in your favorite programming language.

- Given a $m \times n$ matrix contains positive and negative numbers, find the sub metrics with maximum sum of its elements.

- Given $n$ non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining. Figure 14.25 shows an example. For example, Given $\{0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1\}$, the result is 6.

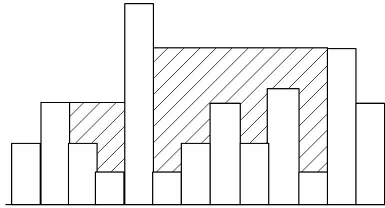


Figure 14.25: Shadowed areas are waters.

- Explain why KMP algorithm perform in linear time even in the seemed 'worst' case.

- Implement the purely functional KMP algorithm by using reversed $P_p$ to avoid the linear time appending operation.

- Deduce the state of the tree $left(right(right(right(T))))$ when searching 'ananym' in text 'anal'.

## 14.3 Solution searching

One interesting thing that computer programming can offer is solving puzzles. In the early phase of classic artificial intelligent, people developed many methods to search for solutions. Different from the sequence searching and string matching, the solution doesn't obviously exist among a candidates set. It typically need construct the solution while trying varies of attempts. Some problems are solvable, while others are not. Among the solvable problems, not all of them just have one unique solution. For example, a maze may have multiple ways out. People sometimes need search for the best one.

### 14.3.1 DFS and BFS

DFS and BFS stand for deep-first search and breadth-first search. They are typically introduced as graph algorithms in textbooks. Graph is a comprehensive topic which is hard to be covered in this elementary book. In this section, we'll show how to use DFS and BFS to solve some real puzzles without formal introduction about the graph concept.

#### Maze

Maze is a classic and popular puzzle. Maze is amazing to both kids and adults. Figure 14.26 shows an example maze. There are also real maze gardens can be found in parks for fun. In the late 1990s, maze-solving games were quite often hold in robot mouse competition all over the world.

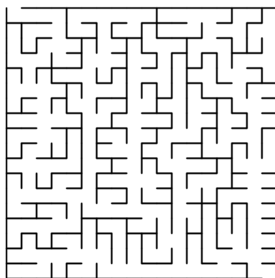


Figure 14.26: A maze

There are multiple methods to solve maze puzzle. We'll introduce an effective, yet not the best one in this section. There are some well known sayings about how to find the way out in maze, while not all of them are true.

For example, one method states that, wherever you have multiple ways, always turn right. This doesn't work as shown in figure 14.27. The obvious solution is first to go along the top horizontal line, then turn right, and keep going ahead at the 'T' section. However, if we always turn right, we'll endless loop around the inner big block.

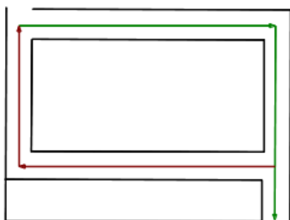


Figure 14.27: It leads to loop way if always turns right.

This example tells us that the decision when there are multiple choices matters the solution. Like the fairy tale we read in our childhood, we can take some bread crumbs in a maze. When there are multiple ways, we can simply select one, left a piece of bread crumbs to mark this attempt. If we enter a died end, we go back to the last place where we've made a decision by back-tracking the bread crumbs. Then we can alter to another way.

At any time, if we find there have been already bread crumbs left, it means we have entered a loop, we must go back and try different ways. Repeat these try-and-check steps, we can either find the way out, or give the 'no solution' fact. In the later case, we back-track to the start point.

One easy way to describe a maze, is by a $m \times n$ matrix, each element is either 0 or 1, which indicates if there is a way at this cell. The maze illustrated in figure 14.27 can be defined as the following matrix.

$$\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0
\end{array}$$

Given a start point $s = (i, j)$, and a goal $e = (p, q)$, we need find all solutions, that are the paths from $s$ to $e$.

There is an obviously recursive exhaustive search method. That in order to find all paths from $s$ to $e$, we can check all connected points to $s$, for every such point $k$, we recursively find all paths from $k$ to $e$. This method can be illustrated as the following.

- Trivial case, if the start point $s$ is as same as the target point $e$, we are done;
- Otherwise, for every connected point $k$ to $s$, recursively find the paths from $k$ to $e$; If $e$ can be reached via $k$, put section $s$-$k$ in front of each path between $k$ and $e$.

However, we have to left 'bread crumbs' to avoid repeatedly trying the same attempts. This is because otherwise in the recursive case, we start from $s$, find a connected point $k$, then we further try to find paths from $k$ to $e$. Since $s$ is connected to $k$ as well, so in the next recursion, we'll try to find paths from $s$ to $e$ again. It turns to be the very same origin problem, and we are trapped in infinite recursions.

Our solution is to initialize an empty list, use it to record all the points we've visited so far. For every connected point, we look up the list to examine if it has already been visited. We skip all the visited candidates and only try those new ones. The corresponding algorithm can be defined like this.

$$solveMaze(m, s, e) = solve(s, \{\phi\}) \tag{14.47}$$

Where $m$ is the matrix which defines a maze, $s$ is the start point, and $e$ is the end point. Function $solve$ is defined in the context of $solveMaze$, so that the maze and the end point can be accessed. It can be realized recursively like what we described above[8].

$$solve(s, P) = \begin{cases} \{\{s\} \cup p | p \in P\} & : s = e \\ concat(\{ \begin{array}{l} solve(s', \{\{s\} \cup p | p \in P\}) | \\ s' \in adj(s), \neg visited(s')\}) \end{array} & : otherwise \end{cases} \tag{14.48}$$

Note that $P$ also serves as an accumulator. Every connected point is recorded in all the possible paths to the current position. But they are stored in reversed order, that is the newly visited point is put to the head of all the lists, and the starting point is the last one. This is because the appending operation is linear ($O(n)$, where $n$ is the number of elements stored in a list), while linking to the head is just constant time. We can output the result in correct order by reversing all possible solutions in equation (14.47)[9]:

$$solveMaze(m, s, e) = map(reverse, solve(s, \{\phi\})) \tag{14.49}$$

We need define functions $adj(p)$ and $visited(p)$, which finds all the connected points to $p$, and tests if point $p$ has been visited respectively. Two points are connected if and

[8] Function $concat$ can flatten a list of lists. For example. $concat(\{\{a, b, c\}, \{x, y, z\}\}) = \{a, b, c, x, y, z\}$. Refer to appendix A for detail.

[9] the detailed definition of $reverse$ can be found in the appendix A.

only if they are next cells horizontally or vertically in the maze matrix, and both have zero value.

$$adj((x, y)) = \{(x', y') | \quad (x', y') \in \{(x-1, y), (x+1, y), (x, y-1), (x, y+1)\}, \quad 1 \leq x' \leq M, 1 \leq y' \leq N, m_{x'y'} = 0\} \tag{14.50}$$

Where $M$ and $N$ are the widths and heights of the maze.

Function $visited(p)$ examines if point $p$ has been recorded in any lists in $P$.

$$visited(p) = \exists path \in P, p \in path \tag{14.51}$$

The following Haskell example code implements this algorithm.

```
solveMaze m from to = map reverse $ solve from [[]] where
    solve p paths | p == to = map (p:) paths
                  | otherwise = concat [solve p' (map (p:) paths) |
                                        p' <- adjacent p,
                                        not $ visited p' paths]
    adjacent (x, y) = [(x', y') |
                       (x', y') <- [(x-1, y), (x+1, y), (x, y-1), (x, y+1)],
                       inRange (bounds m) (x', y'),
                       m ! (x', y') == 0]
    visited p paths = any (p `elem`) paths
```

For a maze defined as matrix like below example, all the solutions can be given by this program.

```
mz = [[0, 0, 1, 0, 1, 1],
      [1, 0, 1, 0, 1, 1],
      [1, 0, 0, 0, 0, 0],
      [1, 1, 0, 1, 1, 1],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 1, 1, 0]]

maze  = listArray ((1,1), (6, 6)) ∘ concat

solveMaze (maze mz) (1,1) (6, 6)
```

As we mentioned, this is a style of 'exhaustive search'. It recursively searches all the connected points as candidates. In a real maze solving game, a robot mouse competition for instance, it's enough to just find a route. We can adapt to a method close to what described at the beginning of this section. The robot mouse always tries the first connected point, and skip the others until it gets stuck. We need some data structure to store the 'bread crumbs', which help to remember the decisions being made. As we always attempt to find the way on top of the latest decision, it is the last-in, first-out manner. A stack can be used to realize it.

At the very beginning, only the starting point $s$ is stored in the stack. we pop it out, find, for example, points $a$, and $b$, are connected to $s$. We push the two possible paths: $\{a, s\}$ and $\{b, s\}$ to the stack. Next we pop $\{a, s\}$ out, and examine connected points to $a$. Then all the paths with 3 steps will be pushed back. We repeat this process. At anytime, each element stored in the stack is a path, from the starting point to the farthest place can arrive in the reversed order. This can be illustrated in figure 14.28.

The stack can be realized with a list. The latest option is picked from the head, and the new candidates are also added to the head. The maze puzzle can be solved by using such a list of paths:

$$solveMaze'(m, s, e) = reverse(solve'(\{\{s\}\})) \tag{14.52}$$

As we are searching the first, but not all the solutions, $map$ isn't used here. When the stack is empty, it means that we've tried all the options and failed to find a way out.

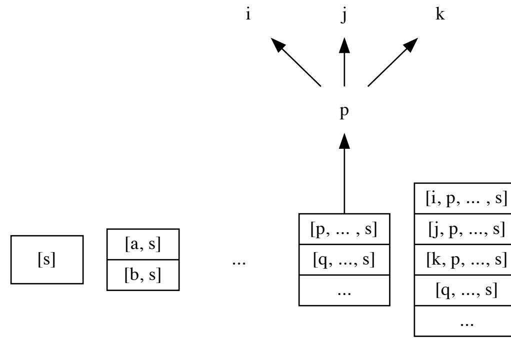



Figure 14.28: The stack is initialized with a singleton list of the starting point $s$. $s$ is connected with point $a$ and $b$. Paths $\{a, s\}$ and $\{b, s\}$ are pushed back. In some step, the path ended with point $p$ is popped. $p$ is connected with points $i$, $j$, and $k$. These 3 points are expanded as different options and pushed back to the stack. The candidate path ended with $q$ won't be examined unless all the options above fail.

There is no solution; otherwise, the top option is popped, expanded with all the adjacent points which haven't been visited before, and pushed back to the stack. Denote the stack as $S$, if it isn't empty, the top element is $s_1$, and the new stack after the top being popped as $S'$. $s_1$ is a list of points represents path $P$. Denote the first point in this path as $p_1$, and the rest as $P'$. The solution can be formalized as the following.

$$solve'(S) = \begin{cases} \phi & : S = \phi \\ s_1 & : s_1 = e \\ solve'(S') & : C = \{c | c \in adj(p_1), c \notin P'\} = \phi \\ solve'(\{\{p\} \cup P | p \in C\} \cup S) & : otherwise, C \neq \phi \end{cases} \tag{14.53}$$

Where the *adj* function is defined above. This updated maze solution can be implemented with the below example Haskell program [10].

```
dfsSolve m from to = reverse $ solve [[from]] where
    solve [] = []
    solve (c@(p:path):cs)
        | p == to = c -- stop at the first solution
        | otherwise = let os = filter (`notElem` path) (adjacent p) in
                          if os == []
                          then solve cs
                          else solve ((map (:c) os) ++ cs)
```

It's quite easy to modify this algorithm to find all solutions. When we find a path in the second clause, instead of returning it immediately, we record it and go on checking the rest memorized options in the stack till until the stack becomes empty. We left it as an exercise to the reader.

The same idea can also be realized imperatively. We maintain a stack to store all possible paths from the starting point. In each iteration, the top option path is popped, if the farthest position is the end point, a solution is found; otherwise, all the adjacent, not visited yet points are appended as new paths and pushed back to the stack. This is repeated till all the candidate paths in the stacks are checked.

We use the same notation to represent the stack $S$. But the paths will be stored as arrays instead of list in imperative settings as the former is more effective. Because of this

[10]The same code of *adjacent* function is skipped

the starting point is the first element in the path array, while the farthest reached place is the right most element. We use $p_n$ to represent LAST($P$) for path $P$. The imperative algorithm can be given as below.

```
1: function SOLVE-MAZE(m, s, e)
2:     S ← φ
3:     PUSH(S, {s})
4:     L ← φ                                  ▷ the result list
5:     while S ≠ φ do
6:         P ← POP(S)
7:         if e = p_n then
8:             ADD(L, P)
9:         else
10:            for ∀p ∈ ADJACENT(m, p_n) do
11:                if p ∉ P then
12:                    PUSH(S, P ∪ {p})
13:    return L
```

The following example Python program implements this maze solving algorithm.

```
def solve(m, src, dst):
    stack = [[src]]
    s = []
    while stack ≠ []:
        path = stack.pop()
        if path[-1] == dst:
            s.append(path)
        else:
            for p in adjacent(m, path[-1]):
                if not p in path:
                    stack.append(path + [p])
    return s

def adjacent(m, p):
    (x, y) = p
    ds = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    ps = []
    for (dx, dy) in ds:
        x1 = x + dx
        y1 = y + dy
        if 0 ≤ x1 and x1 < len(m[0]) and
           0 ≤ y1 and y1 < len(m) and m[y][x] == 0:
            ps.append((x1, y1))
    return ps
```

And the same maze example given above can be solved by this program like the following.

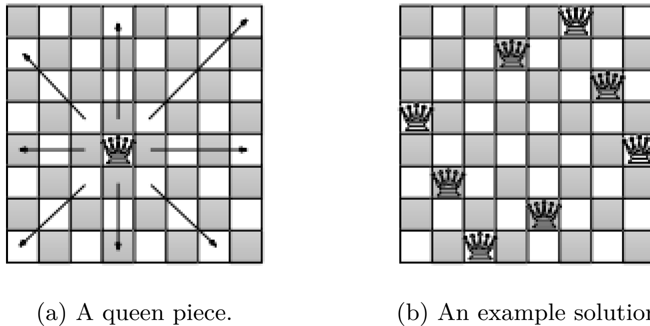
```
mz = [[0, 0, 1, 0, 1, 1],
      [1, 0, 1, 0, 1, 1],
      [1, 0, 0, 0, 0, 0],
      [1, 1, 0, 1, 1, 1],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 1, 1, 0]]

solve(mz, (0, 0), (5,5))
```

It seems that in the worst case, there are 4 options (up, down, left, and right) at each step, each option is pushed to the stack and eventually examined during backtracking. Thus the complexity is bound to $O(4^n)$. The actual time won't be so large because we filtered out the places which have been visited before. In the worst case, all the reachable points are visited exactly once. So the time is bound to $O(n)$, where $n$ is the number

of points connected in total. As a stack is used to store candidate solutions, the space complexity is $O(n^2)$.

### Eight queens puzzle

The eight queens puzzle is also a famous problem. Although cheese has very long history, this puzzle was first published in 1848 by Max Bezzel[89]. Queen in the cheese game is quite powerful. It can attack any other pieces in the same row, column and diagonal at any distance. The puzzle is to find a solution to put 8 queens in the board, so that none of them attack each other. Figure 14.29 (a) illustrates the places can be attacked by a queen and 14.29 (b) shows a solution of 8 queens puzzle.



(a) A queen piece. (b) An example solution

Figure 14.29: The eight queens puzzle.

It's obviously that the puzzle can be solved by brute-force, which takes $P_{64}^8$ times. This number is about $4 \times 10^{10}$. It can be easily improved by observing that, no two queens can be in the same row, and each queen must be put on one column between 1 to 8. Thus we can represent the arrangement as a permutation of $\{1, 2, 3, 4, 5, 6, 7, 8\}$. For instance, the arrangement $\{6, 2, 7, 1, 3, 5, 8, 4\}$ means, we put the first queen at row 1, column 6, the second queen at row 2 column 2, ..., and the last queen at row 8, column 4. By this means, we need only examine $8! = 40320$ possibilities.

We can find better solutions than this. Similar to the maze puzzle, we put queens one by one from the first row. For the first queen, there are 8 options, that we can put it at one of the eight columns. Then for the next queen, we again examine the 8 candidate columns. Some of them are not valid because those positions will be attacked by the first queen. We repeat this process, for the $i$-th queen, we examine the 8 columns in row $i$, find which columns are safe. If none column is valid, it means all the columns in this row will be attacked by some queen we've previously arranged, we have to backtrack as what we did in the maze puzzle. When all the 8 queens are successfully put to the board, we find a solution. In order to find all the possible solutions, we need record it and go on to examine other candidate columns and perform back tracking if necessary. This process terminates when all the columns in the first row have been examined. The below equation starts the search.

$$solve(\{\phi\}, \phi) \tag{14.54}$$

In order to manage the candidate attempts, a stack $S$ is used as same as in the maze puzzle. The stack is initialized with one empty element. And a list $L$ is used to record all possible solutions. Denote the top element in the stack as $s_1$. It's actually an intermediate state of assignment, which is a partial permutation of 1 to 8. after pops $s_1$, the stack

becomes $S'$. The *solve* function can be defined as the following.

$$solve(S, L) = \begin{cases} L & : S = \phi \\ solve(S', \{s_1\} \cup L) & : |s_1| = 8 \\ solve(\left\{ \begin{array}{ll} \{i\} \cup s_1 | & i \in [1, 8], \\ & i \notin s_1, \\ & safe(i, s_1) \end{array} \right\} \cup S', L) & : otherwise \end{cases} \tag{14.55}$$

If the stack is empty, all the possible candidates have been examined, it's not possible to backtrack any more. $L$ has been accumulated all found solutions and returned as the result; Otherwise, if the length of the top element in the stack is 8, a valid solution is found. We add it to $L$, and go on finding other solutions; If the length is less than 8, we need try to put the next queen. Among all the columns from 1 to 8, we pick those not already occupied by previous queens (through the $i \notin s_1$ clause), and must not be attacked in diagonal direction (through the *safe* predication). The valid assignments will be pushed to the stack for the further searching.

Function $safe(x, C)$ detects if the assignment of a queen in position $x$ will be attacked by other queens in $C$ in diagonal direction. There are 2 possible cases, 45° and 135° directions. Since the row of this new queen is $y = 1 + |C|$, where $|C|$ is the length of $C$, the *safe* function can be defined as the following.

$$safe(x, C) = \forall (c, r) \in zip(reverse(C), \{1, 2, ...\}), |x - c| \neq |y - r| \tag{14.56}$$

Where *zip* takes two lists, and pairs every elements in them to a new list. Thus If $C = \{c_{i-1}, c_{i-2}, ..., c_2, c_1\}$ represents the column of the first $i-1$ queens has been assigned, the above function will check list of pairs $\{(c_1, 1), (c_2, 2), ..., (c_{i-1}, i-1)\}$ with position $(x, y)$ forms any diagonal lines.

Translating this algorithm into Haskell gives the below example program.

```
solve = dfsSolve [[]] [] where
    dfsSolve [] s = s
    dfsSolve (c:cs) s
             | length c == 8 = dfsSolve cs (c:s)
             | otherwise = dfsSolve ([(x:c) | x <- [1..8] \\ c,
                                 not $ attack x c] ++ cs) s
    attack x cs = let y = 1 + length cs in
                  any (λ(c, r) -> abs(x - c) == abs(y - r)) $
                      zip (reverse cs) [1..]
```

Observing that the algorithm is tail recursive, it's easy to transform it into imperative realization. Instead of using list, we use array to represent queens assignment. Denote the stack as $S$, and the possible solutions as $A$. The imperative algorithm can be described as the following.

1: **function** Solve-Queens
2: $\quad S \leftarrow \{\phi\}$
3: $\quad L \leftarrow \phi$ ▷ The result list
4: $\quad$ **while** $S \neq \phi$ **do**
5: $\quad\quad A \leftarrow$ Pop($S$) ▷ $A$ is an intermediate assignment
6: $\quad\quad$ **if** $|A| = 8$ **then**
7: $\quad\quad\quad$ Add($L, A$)
8: $\quad\quad$ **else**
9: $\quad\quad\quad$ **for** $i \leftarrow 1$ to 8 **do**
10: $\quad\quad\quad\quad$ **if** Valid($i, A$) **then**
11: $\quad\quad\quad\quad\quad$ Push($S, A \cup \{i\}$)
12: $\quad$ **return** $L$

The stack is initialized with the empty assignment. The main process repeatedly pops the top candidate from the stack. If there are still queens left, the algorithm examines possible columns in the next row from 1 to 8. If a column is safe, that it won't be attacked by any previous queens, this column will be appended to the assignment, and pushed back to the stack. Different from the functional approach, since array, but not list, is used, we needn't reverse the solution assignment any more.

Function VALID checks if column $x$ is safe with previous queens put in $A$. It filters out the columns have already been occupied, and calculates if any diagonal lines are formed with existing queens.

1: **function** VALID($x, A$)
2: $y \leftarrow 1 + |A|$
3: **for** $i \leftarrow 1$ to $|A|$ **do**
4: **if** $x = A[i] \lor |y - i| = |x - A[i]|$ **then**
5: **return** False
6: **return** True

The following Python example program implements this imperative algorithm.

```
def solve():
    stack = [[]]
    s = []
    while stack ≠ []:
        a = stack.pop()
        if len(a) == 8:
            s.append(a)
        else:
            for i in range(1, 9):
                if valid(i, a):
                    stack.append(a+[i])
    return s

def valid(x, a):
    y = len(a) + 1
    for i in range(1, y):
        if x == a[i-1] or abs(y - i) == abs(x - a[i-1]):
            return False
    return True
```

Although there are 8 optional columns for each queen, not all of them are valid and thus further expanded. Only those columns haven't been occupied by previous queens are tried. The algorithm only examines 15720, which is far less than $8^8 = 16777216$, possibilities[89].

It's quite easy to extend the algorithm, so that it can solve $n$ queens puzzle, where $n \geq 4$. However, the time cost increases fast. The backtrack algorithm is just slightly better than the one permuting the sequence of 1 to 8 (which is bound to $o(n!)$). Another extension to the algorithm is based on the fact that the chess board is square, which is symmetric both vertically and horizontally. Thus a solution can generate other solutions by rotating and flipping. These aspects are left as exercises to the reader.

### Peg puzzle

I once received a puzzle of the leap frogs. It said to be homework for 2nd grade student in China. As illustrated in figure 14.30, there are 6 frogs in 7 stones. Each frog can either hop to the next stone if it is not occupied, or leap over one frog to another empty stone. The frogs on the left side can only move to the right, while the ones on the right side can only move to the left. These rules are described in figure 14.31

The goal of this puzzle is to arrange the frogs to jump according to the rules, so that the positions of the 3 frogs on the left are finally exchange with the ones on the right. If

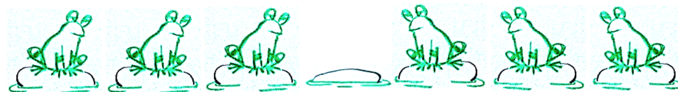
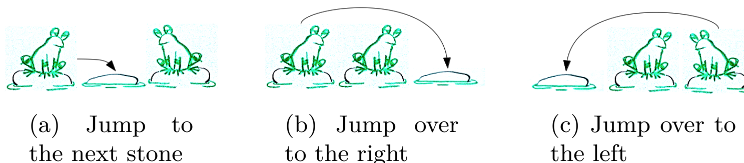


Figure 14.30: The leap frogs puzzle.

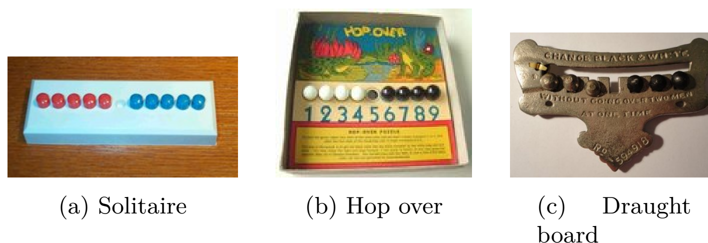
we denote the frog on the left as 'A', on the right as 'B', and the empty stone as 'O'. The puzzle is to find a solution to transform from 'AAAOBBB' to 'BBBOAAA'.



(a) Jump to the next stone (b) Jump over to the right (c) Jump over to the left

Figure 14.31: Moving rules.

This puzzle is just a special form of the peg puzzles. The number of pegs is not limited to 6. it can be 8 or other bigger even numbers. Figure 14.32 shows some variants.




(a) Solitaire (b) Hop over (c) Draught board

Figure 14.32: Variants of the peg puzzles from http://home.comcast.net/~stegmann/jumping.htm

We can solve this puzzle by programing. The idea is similar to the 8 queens puzzle. Denote the positions from the left most stone as 1, 2, ..., 7. In ideal cases, there are 4 options to arrange the move. For example when start, the frog on 3rd stone can hop right to the empty stone; symmetrically, the frog on the 5th stone can hop left; Alternatively, the frog on the 2nd stone can leap right, while the frog on the 6th stone can leap left.

We can record the state and try one of these 4 options at every step. Of course not all of them are possible at any time. If get stuck, we can backtrack and try other options.

As we restrict the left side frogs only moving to the right, and the right frogs only moving to the left, the moves are not reversible. There won't be any repetition cases as what we have to deal with in the maze puzzle. However, we still need record the steps in order to print them out finally.

In order to enforce these restriction, let A, O, B in representation 'AAAOBBB' be -1, 0, and 1 respectively. A state $L$ is a list of elements, each element is one of these 3 values. It starts from $\{-1, -1, -1, 0, 1, 1, 1\}$. $L[i]$ access the $i$-th element, its value indicates if the $i$-th stone is empty, occupied by a frog from left side, or occupied by a frog from right

side. Denote the position of the vacant stone as $p$. The 4 moving options can be stated as below.

- Leap left: $p < 6$ and $L[p+2] > 0$, swap $L[p] \leftrightarrow L[p+2]$;
- Hop left: $p < 7$ and $L[p+1] > 0$, swap $L[p] \leftrightarrow L[p+1]$;
- Leap right: $p > 2$ and $L[p-2] < 0$, swap $L[p-2] \leftrightarrow L[p]$;
- Hop right: $p > 1$ and $L[p-1] < 0$, swap $L[p-1] \leftrightarrow L[p]$.

Four functions $leap_l(L), hop_l(L), leap_r(L)$ and $hop_r(L)$ are defined accordingly. If the state $L$ does not satisfy the move restriction, these function return $L$ unchanged, otherwise, the changed state $L'$ is returned accordingly.

We can also explicitly maintain a stack $S$ to the attempts as well as the historic movements. The stack is initialized with a singleton list of starting state. The solution is accumulated to a list $M$, which is empty at the beginning:

$$solve(\{\{-1,-1,-1,0,1,1,1\}\}, \phi) \tag{14.57}$$

As far as the stack isn't empty, we pop one intermediate attempt. If the latest state is equal to $\{1,1,1,0,-1,-1,-1\}$, a solution is found. We append the series of moves till this state to the result list $M$; otherwise, We expand to next possible state by trying all four possible moves, and push them back to the stack for further search. Denote the top element in the stack $S$ as $s_1$, and the latest state in $s_1$ as $L$. The algorithm can be defined as the following.

$$solve(S, M) = \begin{cases} M & : S = \phi \\ solve(S', \{reverse(s_1)\} \cup M) & : L = \{1,1,1,0,-1,-1,-1\} \\ solve(P \cup S', M) & : otherwise \end{cases} \tag{14.58}$$

Where $P$ are possible moves from the latest state $L$:

$$P = \{L' | L' \in \{leap_l(L), hop_l(L), leap_r(L), hop_r(L)\}, L \neq L'\}$$

Note that the starting state is stored as the last element, while the final state is the first. That is the reason why we reverse it when adding to solution list.

Translating this algorithm to Haskell gives the following example program.

```
solve = dfsSolve [[[-1, -1, -1, 0, 1, 1, 1]]] [] where
    dfsSolve [] s = s
    dfsSolve (c:cs) s
             | head c == [1, 1, 1, 0, -1, -1, -1] = dfsSolve cs (reverse c:s)
             | otherwise = dfsSolve ((map (:c) $ moves $ head c) ++ cs) s

moves s = filter (/=s) [leapLeft s, hopLeft s, leapRight s, hopRight s] where
    leapLeft [] = []
    leapLeft (0:y:1:ys) = 1:y:0:ys
    leapLeft (y:ys) = y:leapLeft ys
    hopLeft [] = []
    hopLeft (0:1:ys) = 1:0:ys
    hopLeft (y:ys) = y:hopLeft ys
    leapRight [] = []
    leapRight (-1:y:0:ys) = 0:y:(-1):ys
    leapRight (y:ys) = y:leapRight ys
    hopRight [] = []
    hopRight (-1:0:ys) = 0:(-1):ys
    hopRight (y:ys) = y:hopRight ys
```

Running this program finds 2 symmetric solutions, each takes 15 steps. One solution is list in the below table.

| step | -1 | -1 | -1 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | -1 | -1 | 0 | -1 | 1 | 1 | 1 |
| 2 | -1 | -1 | 1 | -1 | 0 | 1 | 1 |
| 3 | -1 | -1 | 1 | -1 | 1 | 0 | 1 |
| 4 | -1 | -1 | 1 | 0 | 1 | -1 | 1 |
| 5 | -1 | 0 | 1 | -1 | 1 | -1 | 1 |
| 6 | 0 | -1 | 1 | -1 | 1 | -1 | 1 |
| 7 | 1 | -1 | 0 | -1 | 1 | -1 | 1 |
| 8 | 1 | -1 | 1 | -1 | 0 | -1 | 1 |
| 9 | 1 | -1 | 1 | -1 | 1 | -1 | 0 |
| 10 | 1 | -1 | 1 | -1 | 1 | 0 | -1 |
| 11 | 1 | -1 | 1 | 0 | 1 | -1 | -1 |
| 12 | 1 | 0 | 1 | -1 | 1 | -1 | -1 |
| 13 | 1 | 1 | 0 | -1 | 1 | -1 | -1 |
| 14 | 1 | 1 | 1 | -1 | 0 | -1 | -1 |
| 15 | 1 | 1 | 1 | 0 | -1 | -1 | -1 |

Observe that the algorithm is in tail recursive manner, it can also be realized imperatively. The algorithm can be more generalized, so that it solve the puzzles of $n$ frogs on each side. We represent the start state {-1, -1, ..., -1, 0, 1, 1, ..., 1} as $s$, and the mirrored end state as $e$.

1: **function** SOLVE($s, e$)
2: $\quad S \leftarrow \{\{s\}\}$
3: $\quad M \leftarrow \phi$
4: $\quad$ **while** $S \neq \phi$ **do**
5: $\quad\quad s_1 \leftarrow$ POP($S$)
6: $\quad\quad$ **if** $s_1[1] = e$ **then**
7: $\quad\quad\quad$ ADD($M$, REVERSE($s_1$))
8: $\quad\quad$ **else**
9: $\quad\quad\quad$ **for** $\forall m \in$ MOVES($s_1[1]$) **do**
10: $\quad\quad\quad\quad$ PUSH($S$, $\{m\} \cup s_1$)
11: $\quad$ **return** $M$

The possible moves can be also generalized with procedure MOVES to handle arbitrary number of frogs. The following Python program implements this solution.

```
def solve(start, end):
    stack = [[start]]
    s = []
    while stack ≠ []:
        c = stack.pop()
        if c[0] == end:
            s.append(reversed(c))
        else:
            for m in moves(c[0]):
                stack.append([m]+c)
    return s

def moves(s):
    ms = []
    n = len(s)
    p = s.index(0)
    if p < n - 2 and s[p+2] > 0:
        ms.append(swap(s, p, p+2))
    if p < n - 1 and s[p+1] > 0:
        ms.append(swap(s, p, p+1))
```

```
    if p > 1 and s[p-2] < 0:
        ms.append(swap(s, p, p-2))
    if p > 0 and s[p-1] < 0:
        ms.append(swap(s, p, p-1))
    return ms

def swap(s, i, j):
    a = s[:]
    (a[i], a[j]) = (a[j], a[i])
    return a
```

For 3 frogs in each side, we know that it takes 15 steps to exchange them. It's interesting to examine the table that how many steps are needed along with the number of frogs in each side. Our program gives the following result.

| number of frogs | 1 | 2 | 3 | 4 | 5 | ... |
|---|---|---|---|---|---|---|
| number of steps | 3 | 8 | 15 | 24 | 35 | ... |

It seems that the number of steps are all square numbers minus one. It's natural to guess that the number of steps for $n$ frogs in one side is $(n+1)^2 - 1$. Actually we can prove it is true.

Compare to the final state and the start state, each frog moves ahead $n+1$ stones in its opposite direction. Thus total $2n$ frogs move $2n(n+1)$ stones. Another important fact is that each frog on the left has to meet every one on the right one time. And leap will happen when meets. Since the frog moves two stones ahead by leap, and there are total $n^2$ meets happen, so that all these meets cause moving $2n^2$ stones ahead. The rest moves are not leap, but hop. The number of hops are $2n(n+1) - 2n^2 = 2n$. Sum up all $n^2$ leaps and $2n$ hops, the total number of steps are $n^2 + 2n = (n+1)^2 - 1$.

### Summary of DFS

Observe the above three puzzles, although they vary in many aspects, their solutions show quite similar common structures. They all have some starting state. The maze starts from the entrance point; The 8 queens puzzle starts from the empty board; The leap frogs start from the state of 'AAAOBBB'. The solution is a kind of searching, at each attempt, there are several possible ways. For the maze puzzle, there are four different directions to try; For the 8 queens puzzle, there are eight columns to choose; For the leap frogs puzzle, there are four movements of leap or hop. We don't know how far we can go when make a decision, although the final state is clear. For the maze, it's the exit point; For the 8 queens puzzle, we are done when all the 8 queens being assigned on the board; For the leap frogs puzzle, the final state is that all frogs exchanged.

We use a common approach to solve them. We repeatedly select one possible candidate to try, record where we've achieved; If we get stuck, we backtrack and try other options. We are sure by using this strategy, we can either find a solution, or tell that the problem is unsolvable.

Of course there can be some variation, that we can stop when find one answer, or go on searching all the solutions.

If we draw a tree rooted at the starting state, expand it so that every branch stands for a different attempt, our searching process is in a manner, that it searches deeper and deeper. We won't consider any other options in the same depth unless the searching fails so that we've to backtrack to upper level of the tree. Figure 14.33 illustrates the order we search a state tree. The arrow indicates how we go down and backtrack up. The number of the nodes shows the order we visit them.

This kind of search strategy is called 'DFS' (Deep-first-search). We widely use it unintentionally. Some programming environments, Prolog for instance, adopt DFS as the default evaluation model. A maze is given by a set of rule base, such as:

Figure 14.33: Example of DFS search order.

```
c(a, b). c(a, e).
c(b, c). c(b, f).
c(e, d), c(e, f).
c(f, c).
c(g, d). c(g, h).
c(h, f).
```

Where predicate $c(X, Y)$ means place $X$ is connected with $Y$. Note that this is a directed predicate, we can make $Y$ to be connected with $X$ as well by either adding a symmetric rule, or create a undirected predicate. Figure 14.34 shows such a directed graph. Given two places $X$ and $Y$, Prolog can tell if they are connected by the following program.

Figure 14.34: A directed graph.

```
go(X, X).
go(X, Y) :- c(X, Z), go(Z, Y)
```

This program says that, a place is connected with itself. Given two different places $X$ and $Y$, if $X$ is connected with $Z$, and $Z$ is connected with $Y$, then $X$ is connected with $Y$. Note that, there might be multiple choices for $Z$. Prolog selects a candidate, and go on further searching. It only tries other candidates if the recursive searching fails. In that case, Prolog backtracks and tries other alternatives. This is exactly what DFS does.

DFS is quite straightforward when we only need a solution, but don't care if the solution takes the fewest steps. For example, the solution it gives, may not be the shortest path for the maze. We'll see some more puzzles next. They demands to find the solution with the minimum attempts.

### The wolf, goat, and cabbage puzzle

This puzzle says that a farmer wants to cross a river with a wolf, a goat, and a bucket of cabbage. There is a boat. Only the farmer can drive it. But the boat is small. it can only hold one of the wolf, the goat, and the bucket of cabbage with the farmer at a time. The farmer has to pick them one by one to the other side of the river. However, the wolf would eat the goat, and the goat would eat the cabbage if the farmer is absent. The puzzle asks to find the fast solution so that they can all safely go cross the river.

Figure 14.35: The wolf, goat, cabbage puzzle

The key point to this puzzle is that the wolf does not eat the cabbage. The farmer can safely pick the goat to the other side. But next time, no matter if he pick the wolf or the cabbage to cross the river, he has to take one back to avoid the conflict. In order to find the fast the solution, at any time, if the farmer has multiple options, we can examine all of them in parallel, so that these different decisions compete. If we count the number of the times the farmer cross the river without considering the direction, that crossing the river back and forth means 2 times, we are actually checking the complete possibilities after 1 time, 2 times, 3 times, ... When we find a situation, that they all arrive at the other bank, we are done. And this solution wins the competition, which is the fast solution.

The problem is that we can't examine all the possible solutions in parallel ideally. Even with a super computer equipped with many CPU cores, the setup is too expensive to solve such a simple puzzle.

Let's consider a lucky draw game. People blindly pick from a box with colored balls. There is only one black ball, all the others are white. The one who pick the black ball wins the game; Otherwise, he must return the ball to the box and wait for the next chance. In order to be fair enough, we can setup a rule that no one can try the second time before all others have tried. We can line people to a queue. Every time the first guy pick a ball, if he does not win, he then stands at the tail of the queue to wait for the second try. This queue helps to ensure our rule.

We can use the quite same idea to solve our puzzle. The two banks of the river can be represented as two sets $A$ and $B$. $A$ contains the wolf, the goat, the cabbage and the farmer; while $B$ is empty. We take an element along with the farmer from one set to the other each time. The two sets can't hold conflict things if the farmer is absent. The goal is to exchange the contents of $A$ and $B$ with fewest steps.

We initialize a queue with state $A = \{w, g, c, p\}, B = \phi$ as the only element. As far as the queue isn't empty, we pick the first element from the head, expand it with all possible

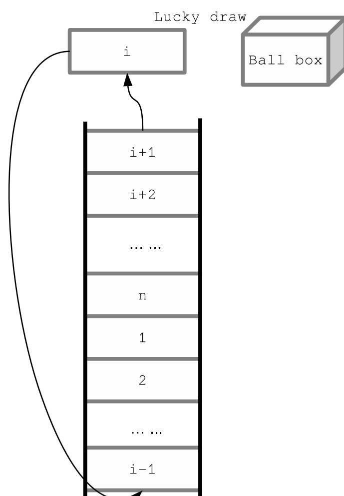



Figure 14.36: A lucky-draw game, the $i$-th person goes from the queue, pick a ball, then join the queue at tail if he fails to pick the black ball.

options, and put these new expanded candidates to the tail of the queue. If the first element on the head is the final goal, that $A = \phi, B = \{w, g, c, p\}$, we are done. Figure 14.37 illustrates the idea of this search order. Note that as all possibilities in the same level are examined, there is no need for back-tracking.

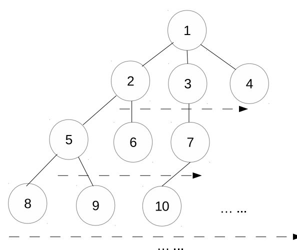



Figure 14.37: Start from state 1, check all possible options 2, 3, and 4 for next step; then all nodes in level 3, ...

There is a simple way to treat the set. A four bits binary number can be used, each bit stands for a thing, for example, the wolf $w = 1$, the goat $g = 2$, the cabbage $c = 4$, and the farmer $p = 8$. That 0 stands for the empty set, 15 stands for a full set. Value 3, solely means there are a wolf and a goat on the river bank. In this case, the wolf will eat the goat. Similarly, value 6 stands for another conflicting case. Every time, we move the highest bit (which is 8), or together with one of the other bits (4 or 2, or 1) from one

number to the other. The possible moves can be defined as below.

$$mv(A, B) = \begin{cases} \{(A - 8 - i, B + 8 + i) | i \in \{0, 1, 2, 4\}, i = 0 \lor A \overline{\land} i \neq 0\} & : B < 8 \\ \{(A + 8 + i, B - 8 - i) | i \in \{0, 1, 2, 4\}, i = 0 \lor B \overline{\land} i \neq 0\} & : Otherwise \end{cases} \tag{14.59}$$

Where $\overline{\land}$ is the bitwise-and operation.

the solution can be given by reusing the queue defined in previous chapter. Denote the queue as $Q$, which is initialed with a singleton list $\{(15,\ 0)\}$. If $Q$ is not empty, function $DeQ(Q)$ extracts the head element $M$, the updated queue becomes $Q'$. $M$ is a list of pairs, stands for a series of movements between the river banks. The first element in $m_1 = (A', B')$ is the latest state. Function $EnQ'(Q, L)$ is a slightly different enqueue operation. It pushes all the possible moving sequences in $L$ to the tail of the queue one by one and returns the updated queue. With these notations, the solution function is defined like below.

$$solve(Q) = \begin{cases} \phi & : Q = \phi \\ reverse(M) & : A' = 0 \\ solve(EnQ'(Q', \left\{ \begin{array}{ll} \{m\} \cup M | & m \in mv(m_1), \\ & valid(m, M) \end{array} \right\})) & : otherwise \end{cases} \tag{14.60}$$

Where function $valid(m, M)$ checks if the new moving candidate $m = (A'', B'')$ is valid. That neither $A''$ nor $B''$ is 3 or 6, and $m$ hasn't been tried before in $M$ to avoid any repeatedly attempts.

$$valid(m, M) = A'' \neq 3, A'' \neq 6, B'' \neq 3, B'' \neq 6, m \notin M \tag{14.61}$$

The following example Haskell program implements this solution. Note that it uses a plain list to represent the queue for illustration purpose.

```
import Data.Bits

solve = bfsSolve [[(15, 0)]] where
    bfsSolve :: [[(Int, Int)]] → [(Int, Int)]
    bfsSolve [] = [] -- no solution
    bfsSolve (c:cs) | (fst $ head c) == 0 = reverse c
                    | otherwise = bfsSolve (cs ++ map (:c)
                                    (filter (`valid` c) $ moves $ head c))
    valid (a, b) r = not $ or [ a `elem` [3, 6], b `elem` [3, 6],
                              (a, b) `elem` r]

moves (a, b) = if b < 8 then trans a b else map swap (trans b a) where
    trans x y = [(x - 8 - i, y + 8 + i)
                    | i ←[0, 1, 2, 4], i == 0 || (x .&. i) /= 0]
    swap (x, y) = (y, x)
```

This algorithm can be easily modified to find all the possible solutions, but not just stop after finding the first one. This is left as the exercise to the reader. The following shows the two best solutions to this puzzle.

Solution 1:

| Left | river | Right |
|---|---|---|
| wolf, goat, cabbage, farmer | | |
| wolf, cabbage | | goat, farmer |
| wolf, cabbage, farmer | | goat |
| cabbage | | wolf, goat, farmer |
| goat, cabbage, farmer | | wolf |
| goat | | wolf, cabbage, farmer |
| goat, farmer | | wolf, cabbage |
| | | wolf, goat, cabbage, farmer |

Solution 2:

| Left | river | Right |
|---|---|---|
| wolf, goat, cabbage, farmer | | |
| wolf, cabbage | | goat, farmer |
| wolf, cabbage, farmer | | goat |
| wolf | | goat, cabbage, farmer |
| wolf, goat, farmer | | cabbage |
| goat | | wolf, cabbage, farmer |
| goat, farmer | | wolf, cabbage |
| | | wolf, goat, cabbage, farmer |

This algorithm can also be realized imperatively. Observing that our solution is in tail recursive manner, we can translate it directly to a loop. We use a list $S$ to hold all the solutions can be found. The singleton list $\{(15, 0)\}$ is pushed to queue when initializing. As long as the queue isn't empty, we extract the head $C$ from the queue by calling DeQ procedure. Examine if it reaches the final goal, if not, we expand all the possible moves and push to the tail of the queue for further searching.

1: **function** Solve
2: $S \leftarrow \phi$
3: $Q \leftarrow \phi$
4: EnQ$(Q, \{(15, 0)\})$
5: **while** $Q \neq \phi$ **do**
6: $C \leftarrow$ DeQ$(Q)$
7: **if** $c_1 = (0, 15)$ **then**
8: Add$(S$, Reverse$(C))$
9: **else**
10: **for** $\forall m \in$ Moves$(C)$ **do**
11: **if** Valid$(m, C)$ **then**
12: EnQ$(Q, \{m\} \cup C)$
13: **return** $S$

Where Moves, and Valid procedures are as same as before. The following Python example program implements this imperative algorithm.

```
def solve():
    s = []
    queue = [[(0xf, 0)]]
    while queue ≠ []:
        cur = queue.pop(0)
        if cur[0] == (0, 0xf):
            s.append(reverse(cur))
        else:
            for m in moves(cur):
                queue.append([m]+cur)
    return s

def moves(s):
    (a, b) = s[0]
    return valid(s, trans(a, b) if b < 8 else swaps(trans(b, a)))

def valid(s, mv):
    return [(a, b) for (a, b) in mv
        if a not in [3, 6] and b not in [3, 6] and (a, b) not in s]

def trans(a, b):
    masks = [ 8 | (1<<i) for i in range(4)]
    return [(a ^ mask, b | mask) for mask in masks if a & mask == mask]

def swaps(s):
```

```
    return [(b, a) for (a, b) in s]
```

There is a minor difference between the program and the pseudo code, that the function to generate candidate moving options filters the invalid cases inside it.

Every time, no matter the farmer drives the boat back and forth, there are $m$ options for him to choose, where $m$ is the number of objects on the river bank the farmer drives from. $m$ is always less than 4, that the algorithm won't take more than $n^4$ times at step $n$. This estimation is far more than the actual time, because we avoid trying all invalid cases. Our solution examines all the possible moving in the worst case. Because we check recorded steps to avoid repeated attempt, the algorithm takes about $O(n^2)$ time to search for $n$ possible steps.

### Water jugs puzzle

This is a popular puzzle in classic AI. The history of it should be very long. It says that there are two jugs, one is 9 quarts, the other is 4 quarts. How to use them to bring up from the river exactly 6 quarts of water?

There are varies versions of this puzzle, although the volume of the jugs, and the target volume of water differ. The solver is said to be young Blaise Pascal when he was a child, the French mathematician, scientist in one story, and Simèon Denis Poisson in another story. Later in the popular Hollywood movie 'Die-Hard 3', actor Bruce Willis and Samuel L. Jackson were also confronted with this puzzle.

Pòlya gave a nice way to solve this problem backwards in[90].

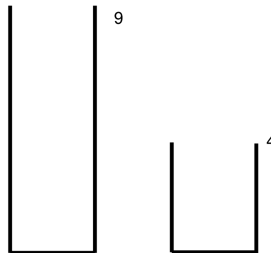



Figure 14.38: Two jugs with volume of 9 and 4.

Instead of thinking from the starting state as shown in figure 14.38. Pòlya pointed out that there will be 6 quarts of water in the bigger jugs at the final stage, which indicates the second last step, we can fill the 9 quarts jug, then pour out 3 quarts from it. In order to achieve this, there should be 1 quart of water left in the smaller jug as shown in figure 14.39.

It's easy to see that fill the 9 quarters jug, then pour to the 4 quarters jug twice can bring 1 quarters of water. As shown in figure 14.40. At this stage, we've found the solution. By reversing our findings, we can give the correct steps to bring exactly 6 quarters of water.

Pòlya's methodology is general. It's still hard to solve it without concrete algorithm. For instance, how to bring up 2 gallons of water from 899 and 1147 gallon jugs?

There are 6 ways to deal with 2 jugs in total. Denote the smaller jug as $A$, the bigger jug as $B$.

- Fill jug $A$ from the river;

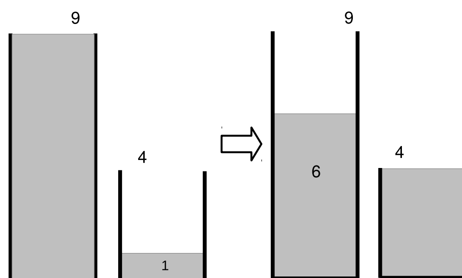



Figure 14.39: The last two steps.

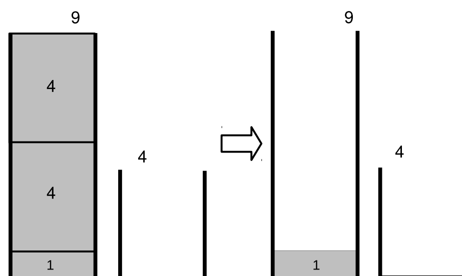



Figure 14.40: Fill the bigger jugs, and pour to the smaller one twice.

- Fill jug $B$ from the river;
- Empty jug $A$;
- Empty jug $B$;
- Pour water from jug $A$ to $B$;
- Pour water from jug $B$ to $A$.

The following sequence shows an example. Note that in this example, we assume that $a < b < 2a$.

| $A$ | $B$ | operation |
|---|---|---|
| 0 | 0 | start |
| a | 0 | fill $A$ |
| 0 | a | pour $A$ into $B$ |
| a | a | fill $A$ |
| 2a - b | b | pour $A$ into $B$ |
| 2a - b | 0 | empty $B$ |
| 0 | 2a - b | pour $A$ into $B$ |
| a | 2a - b | fill $A$ |
| 3a - 2b | b | pour $A$ into $B$ |
| ... | ... | ... |

No matter what the above operations are taken, the amount of water in each jug can be expressed as $xa + yb$, where $a$ and $b$ are volumes of jugs, for some integers $x$ and $y$. All the amounts of water we can get are linear combination of $a$ and $b$. We can immediately tell given two jugs, if a goal $g$ is solvable or not.

For instance, we can't bring 5 gallons of water with two jugs of volume 4 and 6 gallon. The number theory ensures that, the 2 water jugs puzzle can be solved if and only if $g$ can be divided by the greatest common divisor of $a$ and $b$. Written as:

$$gcd(a, b) | g \tag{14.62}$$

Where $m|n$ means $n$ can be divided by $m$. What's more, if $a$ and $b$ are relatively prime, which means $gcd(a, b) = 1$, it's possible to bring up any quantity $g$ of water.

Although $gcd(a, b)$ enables us to determine if the puzzle is solvable, it doesn't give us the detailed pour sequence. If we can find some integer $x$ and $y$, so that $g = xa + yb$. We can arrange a sequence of operations (even it may not be the best solution) to solve it. The idea is that, without loss of generality, suppose $x > 0, y < 0$, we need fill jug $A$ by $x$ times, and empty jug $B$ by $y$ times in total.

Let's take $a = 3$, $b = 5$, and $g = 4$ for example, since $4 = 3 \times 3 - 5$, we can arrange a sequence like the following.

| $A$ | $B$ | operation |
|---|---|---|
| 0 | 0 | start |
| 3 | 0 | fill $A$ |
| 0 | 3 | pour $A$ into $B$ |
| 3 | 3 | fill $A$ |
| 1 | 5 | pour $A$ into $B$ |
| 1 | 0 | empty $B$ |
| 0 | 1 | pour $A$ into $B$ |
| 3 | 1 | fill $A$ |
| 0 | 4 | pour $A$ into $B$ |

In this sequence, we fill $A$ by 3 times, and empty $B$ by 1 time. The procedure can be described as the following:

Repeat $x$ times:

1. Fill jug $A$;
2. Pour jug $A$ into jug $B$, whenever $B$ is full, empty it.

So the only problem left is to find the $x$ and $y$. There is a powerful tool in number theory called, *Extended Euclid algorithm*, which can achieve this. Compare to the classic Euclid GCD algorithm, which can only give the greatest common divisor, The extended Euclid algorithm can give a pair of $x, y$ as well, so that:

$$(d, x, y) = gcd_{ext}(a, b) \tag{14.63}$$

where $d = gcd(a, b)$ and $ax + by = d$. Without loss of generality, suppose $a < b$, there exits quotation $q$ and remainder $r$ that:

$$b = aq + r \tag{14.64}$$

Since $d$ is the common divisor, it can divide both $a$ and $b$, thus $d$ can divide $r$ as well. Because $r$ is less than $a$, we can scale down the problem by finding GCD of $a$ and $r$:

$$(d, x', y') = gcd_{ext}(r, a) \tag{14.65}$$

Where $d = x'r + y'a$ according to the definition of the extended Euclid algorithm. Transform $b = aq + r$ to $r = b - aq$, substitute $r$ in above equation yields:

$$\begin{array}{rcl} d & = & x'(b - aq) + y'a \\ & = & (y' - x'q)a + x'b \end{array} \tag{14.66}$$

This is the linear combination of $a$ and $b$, so that we have:

$$\begin{cases} x = y' - x' \dfrac{b}{a} \\ y = x' \end{cases} \tag{14.67}$$

Note that this is a typical recursive relationship. The edge case happens when $a = 0$.

$$gcd(0, b) = b = 0a + 1b \tag{14.68}$$

Summarize the above result, the extended Euclid algorithm can be defined as the following:

$$gcd_{ext}(a, b) = \begin{cases} (b, 0, 1) & : \quad a = 0 \\ (d, y' - x' \dfrac{b}{a}, x') & : \quad otherwise \end{cases} \tag{14.69}$$

Where $d$, $x'$, $y'$ are defined in equation (14.65).

The 2 water jugs puzzle is almost solved, but there are still two detailed problems need to be tackled. First, extended Euclid algorithm gives the linear combination for the greatest common divisor $d$. While the target volume of water $g$ isn't necessarily equal to $d$. This can be easily solved by multiplying $x$ and $y$ by $m$ times, where $m = g/gcd(a, b)$; Second, we assume $x > 0$, to form a procedure to fill jug $A$ with $x$ times. However, the extended Euclid algorithm doesn't ensure $x$ to be positive. For instance $gcd_{ext}(4, 9) = (1, -2, 1)$. Whenever we get a negative $x$, since $d = xa + yb$, we can continuously add $b$ to $x$, and decrease $y$ by $a$ till $x$ is greater than zero.

At this stage, we are able to give the complete solution to the 2 water jugs puzzle. Below is an example Haskell program.

```
extGcd 0 b = (b, 0, 1)
extGcd a b = let (d, x', y') = extGcd (b `mod` a) a in
               (d, y' - x' * (b `div` a), x')

solve a b g | g `mod` d /= 0 = [] -- no solution
            | otherwise = solve' (x * g `div` d)
    where
      (d, x, y) = extGcd a b
      solve' x | x < 0 = solve' (x + b)
               | otherwise = pour x [(0, 0)]
      pour 0 ps = reverse ((0, g):ps)
      pour x ps@((a', b'):_) | a' == 0 = pour (x - 1) ((a, b'):ps) -- fill a
                             | b' == b = pour x ((a', 0):ps) -- empty b
                             | otherwise = pour x ((max 0 (a' + b' - b),
                                                   min (a' + b') b):ps)
```

Although we can solve the 2 water jugs puzzle with extended Euclid algorithm, the solution may not be the best. For instance, when we are going to bring 4 gallons of water from 3 and 5 gallons jugs. The extended Euclid algorithm produces the following sequence:

```
[(0,0),(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),
(0,4),(3,4),(2,5),(2,0),(0,2),(3,2),(0,5),(3,5),
(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),(0,4)]
```

It takes 23 steps to achieve the goal, while the best solution only need 6 steps:

```
[(0,0),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4)]
```

Observe the 23 steps, and we can find that jug $B$ has already contained 4 gallons of water at the 8-th step. But the algorithm ignores this fact and goes on executing the left 15 steps. The reason is that the linear combination $x$ and $y$ we find with the extended Euclid algorithm are not the only numbers satisfying $g = xa + by$. For all these numbers, the smaller $|x| + |y|$, the less steps are needed. There is an exercise to addressing this problem in this section.

The interesting problem is how to find the best solution? We have two approaches, one is to find $x$ and $y$ to minimize $|x| + |y|$; the other is to adopt the quite similar idea as the wolf-goat-cabbage puzzle. We focus on the latter in this section. Since there are at most 6 possible options: fill $A$, fill $B$, pour $A$ into $B$, pour $B$ into $A$, empty $A$ and empty $B$, we can try them in parallel, and check which decision can lead to the best solution. We need record all the states we've achieved to avoid any potential repetition. In order to realize this parallel approach with reasonable resources, a queue can be used to arrange our attempts. The elements stored in this queue are series of pairs $(p, q)$, where $p$ and $q$ represent the volume of waters contained in each jug. These pairs record the sequence of our operations from the beginning to the latest. We initialize the queue with the singleton list contains the starting state $\{(0, 0)\}$.

$$solve(a, b, g) = solve'\{\{(0, 0)\}\} \tag{14.70}$$

Every time, when the queue isn't empty, we pick a sequence from the head of the queue. If this sequence ends with a pair contains the target volume $g$, we find a solution, we can print this sequence by reversing it; Otherwise, we expand the latest pair by trying all the possible 6 options, remove any duplicated states, and add them to the tail of the queue. Denote the queue as $Q$, the first sequence stored on the head of the queue as $S$, the latest pair in $S$ as $(p, q)$, and the rest of pairs as $S'$. After popping the head element, the queue becomes $Q'$. This algorithm can be defined like below:

$$solve'(Q) = \begin{cases} \phi & : Q = \phi \\ reverse(S) & : p = g \lor q = g \\ solve'(EnQ'(Q', \{\{s'\} \cup S' | s' \in try(S)\})) & : otherwise \end{cases} \tag{14.71}$$

Where function $EnQ'$ pushes a list of sequence to the queue one by one. Function $try(S)$ will try all possible 6 options to generate new pairs of water volumes:

$$try(S) = \{s' | s' \in \left\{ \begin{array}{l} fillA(p, q), fillB(p, q), \\ pourA(p, q), pourB(p, q), \\ emptyA(p, q), emptyB(p, q) \end{array} \right\}, s' \notin S'\} \tag{14.72}$$

It's intuitive to define the 6 options. For fill operations, the result is that the volume of the filled jug is full; for empty operation, the result volume is empty; for pour operation, we need test if the jug is big enough to hold all the water.

$$\begin{array}{ll} fillA(p, q) = (a, q) & fillB(p, q) = (p, b) \\ emptyA(p, q) = (0, q) & emptyB(p, q) = (p, 0) \\ pourA(p, q) = (max(0, p + q - b), min(x + y, b)) & \\ pourB(p, q) = (min(x + y, a), max(0, x + y - a)) & \end{array} \tag{14.73}$$

The following example Haskell program implements this method:

```
solve' a b g = bfs [[(0, 0)]] where
  bfs [] = []
  bfs (c:cs) | fst (head c) == g || snd (head c) == g = reverse c
             | otherwise = bfs (cs ++ map (:c) (expand c))
  expand ((x, y):ps) = filter (`notElem` ps) $ map (\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))
```

This method always returns the fast solution. It can also be realized in imperative approach. Instead of storing the complete sequence of operations in every element in the queue, we can store the unique state in a global history list, and use links to track the operation sequence, this can save spaces.

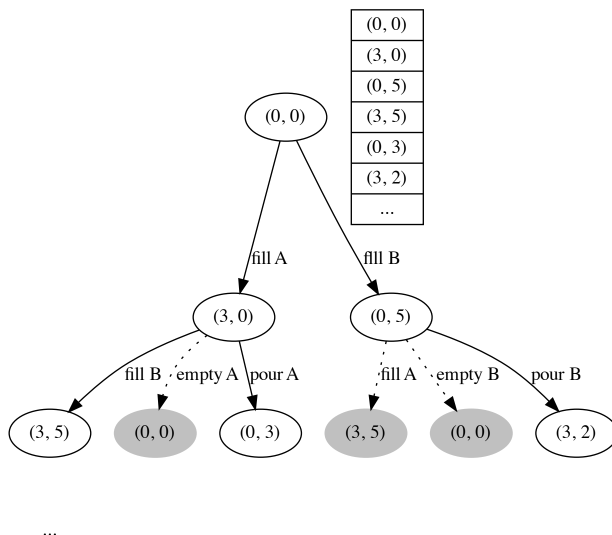



Figure 14.41: All attempted states are stored in a global list.

The idea is illustrated in figure 14.41. The initial state is (0, 0). Only 'fill A' and 'fill B' are possible. They are tried and added to the record list; Next we can try and record 'fill B' on top of (3, 0), which yields new state (3, 5). However, when try 'empty A' from state (3, 0), we would return to the start state (0, 0). As this previous state has been recorded, it is ignored. All the repeated states are in gray color in this figure.

With such settings, we needn't remember the operation sequence in each element in the queue explicitly. We can add a 'parent' link to each node in figure 14.41, and use it to back-traverse to the starting point from any state. The following example ANSI C code shows such a definition.

```
struct Step {
    int p, q;
    struct Step* parent;
};

struct Step* make_step(int p, int q, struct Step* parent) {
    struct Step* s = (struct Step*) malloc(sizeof(struct Step));
    s->p = p;
    s->q = q;
    s->parent = parent;
    return s;
}
```

Where $p, q$ are volumes of water in the 2 jugs. For any state $s$, define functions $p(s)$

and $q(s)$ return these 2 values, the imperative algorithm can be realized based on this idea as below.

1: **function** SOLVE$(a, b, g)$
2: $Q \leftarrow \phi$
3: PUSH-AND-RECORD$(Q,\ (0,\ 0))$
4: **while** $Q \neq \phi$ **do**
5: $s \leftarrow$ POP$(Q)$
6: **if** $p(s) = g \vee q(s) = g$ **then**
7: **return** $s$
8: **else**
9: $C \leftarrow$ EXPAND$(s)$
10: **for** $\forall c \in C$ **do**
11: **if** $c \neq s \wedge \neg$ VISITED$(c)$ **then**
12: PUSH-AND-RECORD$(Q, c)$
13: **return** NIL

Where PUSH-AND-RECORD does not only push an element to the queue, but also record this element as visited, so that we can check if an element has been visited before in the future. This can be implemented with a list. All push operations append the new elements to the tail. For pop operation, instead of removing the element pointed by head, the head pointer only advances to the next one. This list contains historic data which has to be reset explicitly. The following ANSI C code illustrates this idea.

```
struct Step *steps[1000], **head, **tail = steps;

void push(struct Step* s) { *tail++ = s; }

struct Step* pop() { return *head++; }

int empty() { return head == tail; }

void reset() {
    struct Step **p;
    for (p = steps; p != tail; ++p)
        free(*p);
    head = tail = steps;
}
```

In order to test a state has been visited, we can traverse the list to compare $p$ and $q$.

```
int eq(struct Step* a, struct Step* b) {
    return a->p == b->p && a->q == b->q;
}

int visited(struct Step* s) {
    struct Step **p;
    for (p = steps; p != tail; ++p)
        if (eq(*p, s)) return 1;
    return 0;
}
```

The main program can be implemented as below:

```
struct Step* solve(int a, int b, int g) {
    int i;
    struct Step *cur, *cs[6];
    reset();
    push(make_step(0, 0, NULL));
    while (!empty()) {
        cur = pop();
        if (cur->p == g || cur->q == g)
```

```
            return cur;
        else {
            expand(cur, a, b, cs);
            for (i = 0; i < 6; ++i)
                if(!eq(cur, cs[i]) && !visited(cs[i]))
                    push(cs[i]);
        }
    }
    return NULL;
}
```

Where function `expand` tries all the 6 possible options:

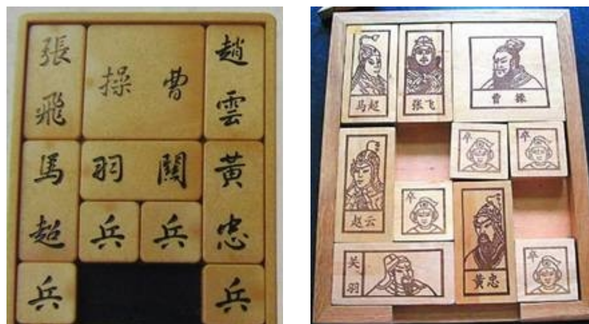
```
void expand(struct Step* s, int a, int b, struct Step** cs) {
    int p = s->p, q = s->q;
    cs[0] = make_step(a, q, s); /*fill A*/
    cs[1] = make_step(p, b, s); /*fill B*/
    cs[2] = make_step(0, q, s); /*empty A*/
    cs[3] = make_step(p, 0, s); /*empty B*/
    cs[4] = make_step(max(0, p + q - b), min(p + q, b), s); /*pour A*/
    cs[5] = make_step(min(p + q, a), max(0, p + q - a), s); /*pour B*/
}
```

And the result steps is back tracked in reversed order, it can be output with a recursive function:

```
void print(struct Step* s) {
    if (s) {
        print(s->parent);
        printf("%d, %d\n", s->p, s->q);
    }
}
```

### Kloski

Kloski is a block sliding puzzle. It appears in many countries. There are different sizes and layouts. Figure 14.42 illustrates a traditional Kloski game in China.




(a) Initial layout of blocks

(b) Block layout after several movements

Figure 14.42: 'Huarong Dao', the traditional Kloski game in China.

In this puzzle, there are 10 blocks, each is labeled with text or icon. The smallest block has size of 1 unit square, the biggest one is $2 \times 2$ units size. Note there is a slot of 2 units wide at the middle-bottom of the board. The biggest block represents a king in ancient time, while the others are enemies. The goal is to move the biggest block to the slot, so that the king can escape. This game is named as 'Huarong Dao', or 'Huarong

Escape' in China. Figure 14.43 shows the similar Kloski puzzle in Japan. The biggest block means daughter, while the others are her family members. This game is named as 'Daughter in the box' in Japan (Japanese name: *hakoiri musume*).

Figure 14.43: 'Daughter in the box', the Kloski game in Japan.

In this section, we want to find a solution, which can slide blocks from the initial state to the final state with the minimum movements.

The intuitive idea to model this puzzle is to use a $5 \times 4$ matrix representing the board. All pieces are labeled with a number. The following matrix $M$, for example, shows the initial state of the puzzle.

$$M = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix}$$

In this matrix, the cells of value $i$ mean the $i$-th piece covers this cell. The special value 0 represents a free cell. By using sequence 1, 2, ... to identify pieces, a special layout can be further simplified as an array $L$. Each element is a list of cells covered by the piece indexed with this element. For example, $L[4] = \{(3,2), (3,3)\}$ means the 4-th piece covers cells at position $(3,2)$ and $(3,3)$, where $(i,j)$ means the cell at row $i$ and column $j$.

The starting layout can be written as the following Array.

$$\{\{(1,1),(2,1)\},\{(1,4),(2,4)\},\{(3,1),(4,1)\},\{(3,2),(3,3)\},\{(3,4),(4,4)\},$$
$$\{(5,1)\},\{(4,2)\},\{(4,3)\},\{(5,4)\},\{(1,2),(1,3),(2,2),(2,3)\}\}$$

When moving the Kloski blocks, we need examine all the 10 blocks, checking each block if it can move up, down, left and right. it seems that this approach would lead to a very huge amount of possibilities, because each step might have $10 \times 4$ options, there will be about $40^n$ cases in the $n$-th step.

Actually, there won't be so much options. For example, in the first step, there are only 4 valid moving: the 6-th piece moves right; the 7-th and 8-th move down; and the 9-th moves left.

All others are invalid moving. Figure 14.44 shows how to test if the moving is possible.

The left example illustrates sliding block labeled with 1 down. There are two cells covered by this block. The upper 1 moves to the cell previously occupied by this same block, which is also labeled with 1; The lower 1 moves to a free cell, which is labeled with 0;

The right example, on the other hand, illustrates invalid sliding. In this case, the upper cells could move to the cell occupied by the same block. However, the lower cell labeled with 1 can't move to the cell occupied by other block, which is labeled with 2.

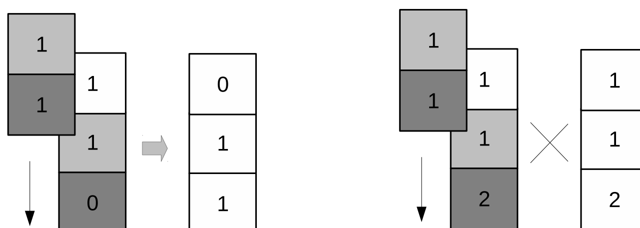



Figure 14.44: Left: both the upper and the lower 1 are OK; Right: the upper 1 is OK, the lower 1 conflicts with 2.

In order to test the valid moving, we need examine all the cells a block will cover. If they are labeled with 0 or a number as same as this block, the moving is valid. Otherwise it conflicts with some other block. For a layout $L$, the corresponding matrix is $M$, suppose we want to move the $k$-th block with $(\Delta x, \Delta y)$, where $|\Delta x| \leq 1, |\Delta y| \leq 1$. The following equation tells if the moving is valid:

$$
\begin{array}{rl}
valid(L, k, \Delta x, \Delta y) : & \\
\forall (i, j) \in L[k] \Rightarrow & i' = i + \Delta y, j' = j + \Delta x, \\
& (1, 1) \leq (i', j') \leq (5, 4), M_{i'j'} \in \{k, 0\}
\end{array}
\tag{14.74}
$$

Another important point to solve Kloski puzzle, is about how to avoid repeated attempts. The obvious case is that after a series of sliding, we end up a matrix which have been transformed from. However, it is not enough to only avoid the same matrix. Consider the following two metrics. Although $M_1 \neq M_2$, we need drop options to $M_2$, because they are essentially the same.

$$
M_1 = \left[ \begin{array}{cccc}
1 & 10 & 10 & 2 \\
1 & 10 & 10 & 2 \\
3 & 4 & 4 & 5 \\
3 & 7 & 8 & 5 \\
6 & 0 & 0 & 9
\end{array} \right]
\quad
M_2 = \left[ \begin{array}{cccc}
2 & 10 & 10 & 1 \\
2 & 10 & 10 & 1 \\
3 & 4 & 4 & 5 \\
3 & 7 & 6 & 5 \\
8 & 0 & 0 & 9
\end{array} \right]
$$

This fact tells us, that we should compare the layout, but not merely matrix to avoid repetition. Denote the corresponding layouts as $L_1$ and $L_2$ respectively, it's easy to verify that $||L_1|| = ||L_2||$, where $||L||$ is the normalized layout, which is defined as below:

$$
||L|| = sort(\{sort(l_i) | \forall l_i \in L\}) \tag{14.75}
$$

In other words, a normalized layout is ordered for all its elements, and every element is also ordered. The ordering can be defined as that $(a, b) \leq (c, d) \Leftrightarrow an + b \leq cn + d$, where $n$ is the width of the matrix.

Observing that the Kloski board is symmetric, thus a layout can be mirrored from another one. Mirrored layout is also a kind of repeating, which should be avoided. The following $M_1$ and $M_2$ show such an example.

$$M_1 = \begin{bmatrix} 10 & 10 & 1 & 2 \\ 10 & 10 & 1 & 2 \\ 3 & 5 & 4 & 4 \\ 3 & 5 & 8 & 9 \\ 6 & 7 & 0 & 0 \end{bmatrix} \quad M_2 = \begin{bmatrix} 3 & 1 & 10 & 10 \\ 3 & 1 & 10 & 10 \\ 4 & 4 & 2 & 5 \\ 7 & 6 & 2 & 5 \\ 0 & 0 & 9 & 8 \end{bmatrix}$$

Note that, the normalized layouts are symmetric to each other. It's easy to get a mirrored layout like this:

$$mirror(L) = \{\{(i, n-j+1) | \forall (i,j) \in l\} | \forall l \in L\} \tag{14.76}$$

We find that the matrix representation is useful in validating the moving, while the layout is handy to model the moving and avoid repeated attempt. We can use the similar approach to solve the Kloski puzzle. We need a queue, every element in the queue contains two parts: a series of moving and the latest layout led by the moving. Each moving is in form of $(k, (\Delta y, \Delta x))$, which means moving the $k$-th block, with $\Delta y$ in row, and $\Delta x$ in column in the board.

The queue contains the starting layout when initialized. Whenever this queue isn't empty, we pick the first one from the head, checking if the biggest block is on target, that $L[10] = \{(4,2),(4,3),(5,2),(5,3)\}$. If yes, then we are done; otherwise, we try to move every block with 4 options: left, right, up, and down, and store all the possible, unique new layout to the tail of the queue. During this searching, we need record all the normalized layouts we've ever found to avoid any duplication.

Denote the queue as $Q$, the historic layouts as $H$, the first layout on the head of the queue as $L$, its corresponding matrix as $M$. and the moving sequence to this layout as $S$. The algorithm can be defined as the following.

$$solve(Q, H) = \begin{cases} \phi & : Q = \phi \\ reverse(S) & : L[10] = \{(4,2),(4,3),(5,2),(5,3)\} \\ solve(Q', H') & : otherwise \end{cases} \tag{14.77}$$

The first clause says that if the queue is empty, we've tried all the possibilities and can't find a solution; The second clause finds a solution, it returns the moving sequence in reversed order; These are two edge cases. Otherwise, the algorithm expands the current layout, puts all the valid new layouts to the tail of the queue to yield $Q'$, and updates the normalized layouts to $H'$. Then it performs recursive searching.

In order to expand a layout to valid unique new layouts, we can define a function as below:

$$\begin{array}{rl} expand(L, H) = \{(k, (\Delta y, \Delta x) | & \forall k \in \{1, 2, ..., 10\}, \\ & \forall (\Delta y, \Delta x) \in \{(0,-1),(0,1),(-1,0),(1,0)\}, \\ & valid(L, k, \Delta x, \Delta y), unique(L', H)\} \end{array} \tag{14.78}$$

Where $L'$ is the the new layout by moving the $k$-th block with $(\Delta y, \Delta x)$ from $L$, $M'$ is the corresponding matrix, and $M''$ is the matrix to the mirrored layout of $L'$. Function unique is defined like this:

$$unique(L', H) = M' \notin H \wedge M'' \notin H \tag{14.79}$$

We'll next show some example Haskell Kloski programs. As array isn't mutable in the purely functional settings, tree based map is used to represent layout [11]. Some type synonyms are defined as below:

[11] Alternatively, finger tree based sequence shown in previous chapter can be used

```
import qualified Data.Map as M
import Data.Ix
import Data.List (sort)

type Point = (Integer, Integer)
type Layout = M.Map Integer [Point]
type Move = (Integer, Point)

data Ops = Op Layout [Move]
```

The main program is almost as same as the $solve(Q, H)$ function defined above.

```
solve :: [Ops] → [[[Point]]]→ [Move]
solve [] _ = [] -- no solution
solve (Op x seq : cs) visit
    | M.lookup 10 x == Just [(4, 2), (4, 3), (5, 2), (5, 3)] = reverse seq
    | otherwise = solve q visit'
  where
    ops = expand x visit
    visit' = map (layout ∘ move x) ops ++ visit
    q = cs ++ [Op (move x op) (op:seq) | op ← ops ]
```

Where function `layout` gives the normalized form by sorting. `move` returns the updated map by sliding the $i$-th block with $(\Delta y, \Delta x)$.

```
layout = sort ∘ map sort ∘ M.elems

move x (i, d) = M.update (Just ∘ map (flip shift d)) i x

shift (y, x) (dy, dx) = (y + dy, x + dx)
```

Function `expand` gives all the possible new options. It can be directly translated from $expand(L, H)$.

```
expand :: Layout → [[[Point]]] → [Move]
expand x visit = [(i, d) | i ←[1..10],
                           d ← [(0, -1), (0, 1), (-1, 0), (1, 0)],
                           valid i d, unique i d] where
  valid i d = all (λp → let p' = shift p d in
                    inRange (bounds board) p' &&
                    (M.keys $ M.filter (elem p') x) `elem` [[i], []])
              (maybe [] id $ M.lookup i x)
  unique i d = let mv = move x (i, d) in
               all (`notElem` visit) (map layout [mv, mirror mv])
```

Note that we also filter out the mirrored layouts. The `mirror` function is given as the following.

```
mirror = M.map (map (λ (y, x) → (y, 5 - x)))
```

This program takes several minutes to produce the best solution, which takes 116 steps. The final 3 steps are shown as below:

```
...

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['A', 'A', '6', '0']
['A', 'A', '0', '8']

['5', '3', '2', '1']
```

```
['5', '3', '2', '1']
['7', '9', '4', '4']
['A', 'A', '0', '6']
['A', 'A', '0', '8']

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['0', 'A', 'A', '6']
['0', 'A', 'A', '8']

total 116 steps
```

The Kloski solution can also be realized imperatively. Note that the $solve(Q, H)$ is tail-recursive, it's easy to transform the algorithm with looping. We can also link one layout to its parent, so that the moving sequence can be recorded globally. This can save some spaces, as the queue needn't store the moving information in every element. When output the result, we only need back-tracking to the starting layout from the last one.

Suppose function LINK$(L', L)$ links a new layout $L'$ to its parent layout $L$. The following algorithm takes a starting layout, and searches for best moving sequence.

1: **function** SOLVE$(L_0)$
2: $H \leftarrow ||L_0||$
3: $Q \leftarrow \phi$
4: PUSH($Q$, LINK($L_0$, NIL))
5: **while** $Q \neq \phi$ **do**
6: $L \leftarrow$ POP$(Q)$
7: **if** $L[10] = \{(4, 2), (4, 3), (5, 2), (5, 3)\}$ **then**
8: **return** $L$
9: **else**
10: **for** each $L' \in$ EXPAND$(L, H)$ **do**
11: PUSH($Q$, LINK$(L', L)$)
12: APPEND$(H, ||L'||)$
13: **return** NIL ▷ No solution

The following example Python program implements this algorithm:

```
class Node:
    def __init__(self, l, p = None):
        self.layout = l
        self.parent = p

def solve(start):
    visit = set([normalize(start)])
    queue = deque([Node(start)])
    while queue:
        cur = queue.popleft()
        layout = cur.layout
        if layout[-1] == [(4, 2), (4, 3), (5, 2), (5, 3)]:
            return cur
        else:
            for brd in expand(layout, visit):
                queue.append(Node(brd, cur))
                visit.add(normalize(brd))
    return None # no solution
```

Where `normalize` and `expand` are implemented as below:

```
def normalize(layout):
```

```
    return tuple(sorted([tuple(sorted(r)) for r in layout]))

def expand(layout, visit):
    def bound(y, x):
        return 1 ≤ y and y ≤ 5 and 1 ≤ x and x ≤ 4
    def valid(m, i, y, x):
        return m[y - 1][x - 1] in [0, i]
    def unique(brd):
        (m, n) = (normalize(brd), normalize(mirror(brd)))
        return m not in visit and n not in visit
    s = []
    d = [(0, -1), (0, 1), (-1, 0), (1, 0)]
    m = matrix(layout)
    for i in range(1, 11):
        for (dy, dx) in d:
            if all(bound(y + dy, x + dx) and valid(m, i, y + dy, x + dx)
                    for (y, x) in layout[i - 1]):
                brd = move(layout, (i, (dy, dx)))
                if unique(brd):
                    s.append(brd)
    return s
```

Like most programming languages, arrays are indexed from 0 but not 1 in Python. This has to be handled properly. The rest functions including `mirror`, `matrix`, and `move` are implemented as the following.

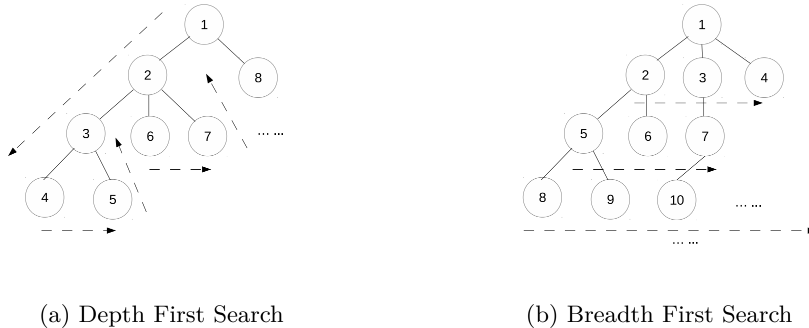
```
def mirror(layout):
    return [[(y, 5 - x) for (y, x) in r] for r in layout]

def matrix(layout):
    m = [[0]*4 for _ in range(5)]
    for (i, ps) in zip(range(1, 11), layout):
        for (y, x) in ps:
            m[y - 1][x - 1] = i
    return m

def move(layout, delta):
    (i, (dy, dx)) = delta
    m = dup(layout)
    m[i - 1] = [(y + dy, x + dx) for (y, x) in m[i - 1]]
    return m

def dup(layout):
    return [r[:] for r in layout]
```

It's possible to modify this Kloski algorithm, so that it does not only stop at the first solution, but also search all the solutions. In such case, the computation time is bound to the size of a space $V$, where $V$ holds all the layouts can be transformed from the starting layout. If all these layouts are stored globally, with a parent field point to the predecessor, the space requirement of this algorithm is also bound to $O(V)$.

### Summary of BFS

The above three puzzles, the wolf-goat-cabbage puzzle, the water jugs puzzle, and the Kloski puzzle show some common solution structure. Similar to the DFS problems, they all have the starting state and the end state. The wolf-goat-cabbage puzzle starts with the wolf, the goat, the cabbage and the farmer all in one side, while the other side is empty. It ends up in a state that they all moved to the other side. The water jugs puzzle starts with two empty jugs, and ends with either jug contains a certain volume of water. The Kloski puzzle starts from a layout and ends to another layout that the biggest block begging slided to a given position.

All problems specify a set of rules which can transfer from one state to another. Different form the DFS approach, we try all the possible options 'in parallel'. We won't search further until all the other alternatives in the same step have been examined. This method ensures that the solution with the minimum steps can be found before those with more steps. Review and compare the two figures we've drawn before shows the difference between these two approaches. For the later one, because we expand the searching horizontally, it is called as Breadth-first search (BFS for short).




(a) Depth First Search (b) Breadth First Search

Figure 14.45: Search orders for DFS and BFS.

As we can't perform search really in parallel, BFS realization typically utilizes a queue to store the search options. The candidate with less steps pops from the head, while the new candidate with more steps is pushed to the tail of the queue. Note that the queue should meet constant time enqueue and dequeue requirement, which we've explained in previous chapter of queue. Strictly speaking, the example functional programs shown above don't meet this criteria. They use list to mimic queue, which can only provide linear time pushing. Readers can replace them with the functional queue we explained before.

BFS provides a simple method to search for optimal solutions in terms of the number of steps. However, it can't search for more general optimal solution. Consider another directed graph as shown in figure 14.46, the length of each section varies. We can't use BFS to find the shortest route from one city to another.

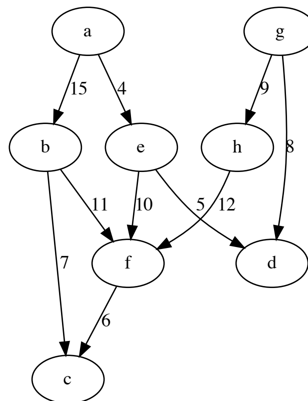



Figure 14.46: A weighted directed graph.

Note that the shortest route from city $a$ to city $c$ isn't the one with the fewest steps

$a \to b \to c$. The total length of this route is 22; But the route with more steps $a \to e \to f \to c$ is the best. The length of it is 20. The coming sections introduce other algorithms to search for optimal solution.

### 14.3.2 Search the optimal solution

Searching for the optimal solution is quite important in many aspects. People need the 'best' solution to save time, space, cost, or energy. However, it's not easy to find the best solution with limited resources. There have been many optimal problems can only be solved by brute-force. Nevertheless, we've found that, for some of them, There exists special simplified ways to search the optimal solution.

#### Grady algorithm

#### Huffman coding

Huffman coding is a solution to encode information with the shortest length of code. Consider the popular ASCII code, which uses 7 bits to encode characters, digits, and symbols. ASCII code can represent $2^7 = 128$ different symbols. With 0, 1 bits, we need at least $\log_2 n$ bits to distinguish $n$ different symbols. For text with only case insensitive English letters, we can define a code table like below.

| char | code | char | code |
|---|---|---|---|
| A | 00000 | N | 01101 |
| B | 00001 | O | 01110 |
| C | 00010 | P | 01111 |
| D | 00011 | Q | 10000 |
| E | 00100 | R | 10001 |
| F | 00101 | S | 10010 |
| G | 00110 | T | 10011 |
| H | 00111 | U | 10100 |
| I | 01000 | V | 10101 |
| J | 01001 | W | 10110 |
| K | 01010 | X | 10111 |
| L | 01011 | Y | 11000 |
| M | 01100 | Z | 11001 |

With this code table, text 'INTERNATIONAL' is encoded to 65 bits.

```
00010101101100100100100011011000000110010001001110101100000011010
```

Observe the above code table, which actually maps the letter 'A' to 'Z' from 0 to 25. There are 5 bits to represent every code. Code zero is forced as '00000' but not '0' for example. Such kind of coding method, is called fixed-length coding.

Another coding method is variable-length coding. That we can use just one bit '0' for 'A', two bits '10' for C, and 5 bits '11001' for 'Z'. Although this approach can shorten the total length of the code for 'INTERNATIONAL' from 65 bits dramatically, it causes problem when decoding. When processing a sequence of bits like '1101', we don't know if it means '1' followed by '101', which stands for 'BF'; or '110' followed by '1', which is 'GB', or '1101' which is 'N'.

The famous Morse code is variable-length coding system. That the most used letter 'E' is encoded as a dot, while 'Z' is encoded as two dashes and two dots. Morse code uses a special pause separator to indicate the termination of a code, so that the above problem won't happen. There is another solution to avoid ambiguity. Consider the following code table.

| char | code | char | code |
|---|---|---|---|
| A | 110 | E | 1110 |
| I | 101 | L | 1111 |
| N | 01 | O | 000 |
| R | 001 | T | 100 |

Text 'INTERNATIONAL' is encoded to 38 bits only:

```
10101100111000101110100101000011101111
```

If decode the bits against the above code table, we won't meet any ambiguity symbols. This is because there is no code for any symbol is the prefix of another one. Such code is called *prefix-code*. (You may wonder why it isn't called as non-prefix code.) By using prefix-code, we needn't separators at all. So that the length of the code can be shorten.

This is a very interesting problem. Can we find a prefix-code table, which produce the shortest code for a given text? The very same problem was given to David A. Huffman in 1951, who was still a student in MIT [91]. His professor Robert M. Fano told the class that those who could solve this problem needn't take the final exam. Huffman almost gave up and started preparing the final exam when he found the most efficient answer.

The idea is to create the coding table according to the frequency of the symbol appeared in the text. The more used symbol is assigned with the shorter code.

It's not hard to process some text, and calculate the occurrence for each symbol. So that we have a symbol set, each one is augmented with a weight. The weight can be the number which indicates the frequency this symbol occurs. We can use the number of occurrence, or the probabilities for example.

Huffman discovered that a binary tree can be used to generate prefix-code. All symbols are stored in the leaf nodes. The codes are generated by traversing the tree from root. When go left, we add a zero; and when go right we add a one.

Figure 14.47 illustrates a binary tree. Taking symbol 'N' for example, starting from the root, we first go left, then right and arrive at 'N'. Thus the code for 'N' is '01'; While for symbol 'A', we can go right, right, then left. So 'A' is encode to '110'. Note that this approach ensures none code is the prefix of the other.

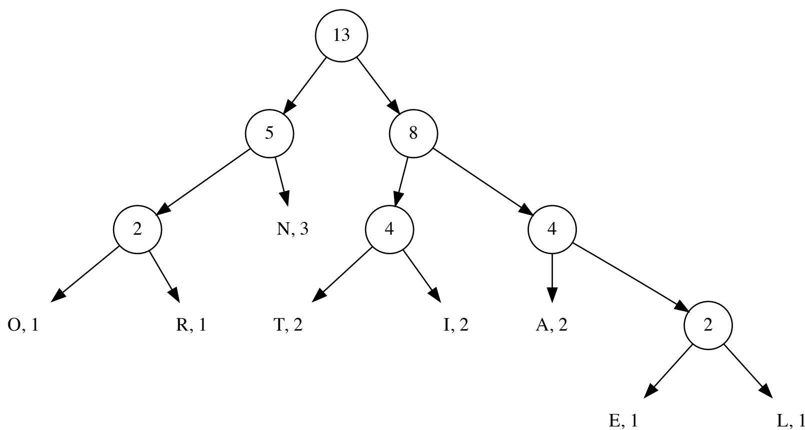
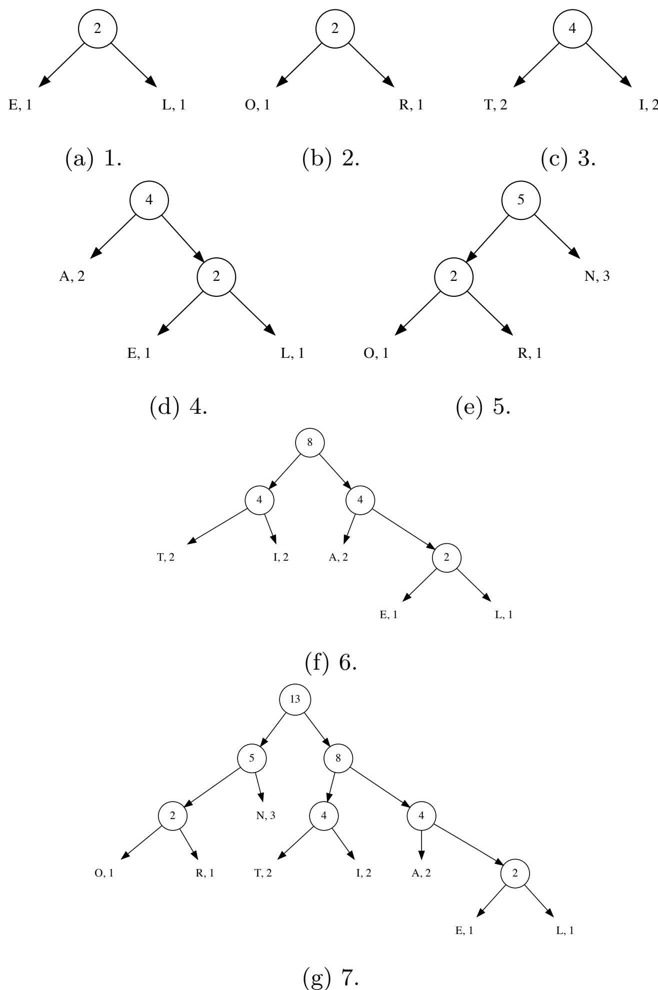



Figure 14.47: An encoding tree.

Note that this tree can also be used directly for decoding. When scan a series of bits, if the bit is zero, we go left; if the bit is one, we go right. When arrive at a leaf, we decode a symbol from that leaf. And we restart from the root of the tree for the coming bits.

Given a list of symbols with weights, we need build such a binary tree, so that the symbol with greater weight has shorter path from the root. Huffman developed a bottom-up solution. When start, all symbols are put into a leaf node. Every time, we pick two nodes, which has the smallest weight, and merge them into a branch node. The weight of this branch is the sum of its two children. We repeatedly pick the two smallest weighted nodes and merge till there is only one tree left. Figure 14.48 illustrates such a building process.




(a) 1. (b) 2. (c) 3.

(d) 4. (e) 5.

(f) 6.

(g) 7.

Figure 14.48: Steps to build a Huffman tree.

We can reuse the binary tree definition to formalize Huffman coding. We augment the weight information, and the symbols are only stored in leaf nodes. The following C like definition, shows an example.

```
struct Node {
    int w;
    char c;
    struct Node *left, *right;
};
```

Some limitation can be added to the definition, as empty tree isn't allowed. A Huffman tree is either a leaf, which contains a symbol and its weight; or a branch, which only holds

total weight of all leaves. The following Haskell code, for instance, explicitly specifies these two cases.

```
data HTr w a = Leaf w a | Branch w (HTr w a) (HTr w a)
```

When merge two Huffman trees $T_1$ and $T_2$ to a bigger one, These two trees are set as its children. We can select either one as the left, and the other as the right. the weight of the result tree $T$ is the sum of its two children. so that $w = w_1 + w_2$. Define $T_1 < T_2$ if $w_1 < w_2$, One possible Huffman tree building algorithm can be realized as the following.

$$build(A) = \begin{cases} T_1 & : A = \{T_1\} \\ build(\{merge(T_a, T_b)\} \cup A') & : otherwise \end{cases} \tag{14.80}$$

$A$ is a list of trees. It is initialized as leaves for all symbols and their weights. If there is only one tree in this list, we are done, the tree is the final Huffman tree. Otherwise, The two smallest tree $T_a$ and $T_b$ are extracted, and the rest trees are hold in list $A'$. $T_a$ and $T_b$ are merged to one bigger tree, and put back to the tree list for further recursive building.

$$(T_a, T_b, A') = extract(A) \tag{14.81}$$

We can scan the tree list to extract the 2 nodes with the smallest weight. Below equation shows that when the scan begins, the first 2 elements are compared and initialized as the two minimum ones. An empty accumulator is passed as the last argument.

$$extract(A) = extract'(min(T_1, T_2), max(T_1, T_2), \{T_3, T_4, ...\}, \phi) \tag{14.82}$$

For every tree, if its weight is less than the smallest two we've ever found, we update the result to contain this tree. For any given tree list $A$, denote the first tree in it as $T_1$, and the rest trees except $T_1$ as $A'$. The scan process can be defined as the following.

$$extract'(T_a, T_b, A, B) = \begin{cases} (T_a, T_b, B) & : A = \phi \\ extract'(T'_a, T'_b, A', \{T_b\} \cup A) & : T_1 < T_b \\ extract'(T_a, T_b, A', \{T_1\} \cup A) & : otherwise \end{cases} \tag{14.83}$$

Where $T'_a = min(T_1, T_a)$, $T'_b = max(T_1, T_a)$ are the updated two trees with the smallest weights.

The following Haskell example program implements this Huffman tree building algorithm.

```
build [x] = x
build xs = build ((merge x y) : xs') where
  (x, y, xs') = extract xs

extract (x:y:xs) = min2 (min x y) (max x y) xs [] where
  min2 x y [] xs = (x, y, xs)
  min2 x y (z:zs) xs | z < y = min2 (min z x) (max z x) zs (y:xs)
                     | otherwise = min2 x y zs (z:xs)
```

This building solution can also be realized imperatively. Given an array of Huffman nodes, we can use the last two cells to hold the nodes with the smallest weights. Then we scan the rest of the array from right to left. Whenever there is a node with the smaller weight, this node will be exchanged with the bigger one of the last two. After all nodes have been examined, we merge the trees in the last two cells, and drop the last cell. This shrinks the array by one. We repeat this process till there is only one tree left.

1: **function** HUFFMAN($A$)
2:     **while** $|A| > 1$ **do**

3: $n \leftarrow |A|$
4: **for** $i \leftarrow n - 2$ down to 1 **do**
5: **if** $A[i] < \text{MAX}(A[n], A[n-1])$ **then**
6: EXCHANGE $A[i] \leftrightarrow \text{MAX}(A[n], A[n-1])$
7: $A[n-1] \leftarrow \text{MERGE}(A[n], A[n-1])$
8: DROP($A[n]$)
9: **return** $A[1]$

The following C++ example program implements this algorithm. Note that this algorithm needn't the last two elements being ordered.

```
typedef vector<Node*> Nodes;

bool lessp(Node* a, Node* b) { return a→w < b→w; }

Node* max(Node* a, Node* b) { return lessp(a, b) ? b : a; }

void swap(Nodes& ts, int i, int j, int k) {
    swap(ts[i], ts[ts[j] < ts[k] ? k : j]);
}

Node* huffman(Nodes ts) {
    int n;
    while((n = ts.size()) > 1) {
        for (int i = n - 3; i ≥ 0; --i)
            if (lessp(ts[i], max(ts[n-1], ts[n-2])))
                swap(ts, i, n-1, n-2);
        ts[n-2] = merge(ts[n-1], ts[n-2]);
        ts.pop_back();
    }
    return ts.front();
}
```

The algorithm merges all the leaves, and it need scan the list in each iteration. Thus the performance is quadratic. This algorithm can be improved. Observe that each time, only the two trees with the smallest weights are merged. This reminds us the heap data structure. Heap ensures to access the smallest element fast. We can put all the leaves in a heap. For binary heap, this is typically a linear operation. Then we extract the minimum element twice, merge them, then put the bigger tree back to the heap. This is $O(\lg n)$ operation if binary heap is used. So the total performance is $O(n \lg n)$, which is better than the above algorithm. The next algorithm extracts the node from the heap, and starts Huffman tree building.

$$build(H) = reduce(top(H), pop(H)) \tag{14.84}$$

This algorithm stops when the heap is empty; Otherwise, it extracts another nodes from the heap for merging.

$$reduce(T, H) = \begin{cases} T & : H = \phi \\ build(insert(merge(T, top(H)), pop(H))) & : otherwise \end{cases} \tag{14.85}$$

Function *build* and *reduce* are mutually recursive. The following Haskell example program implements this algorithm by using heap defined in previous chapter.

```
huffman' :: (Num a, Ord a) ⇒ [(b, a)] → HTr a b
huffman' = build' ∘ Heap.fromList ∘ map (λ(c, w) → Leaf w c) where
  build' h = reduce (Heap.findMin h) (Heap.deleteMin h)
  reduce x Heap.E = x
  reduce x h = build' $ Heap.insert (Heap.deleteMin h) (merge x (Heap.findMin h))
```

The heap solution can also be realized imperatively. The leaves are firstly transformed to a heap, so that the one with the minimum weight is put on the top. As far as there are more than 1 elements in the heap, we extract the two smallest, merge them to a bigger one, and put back to the heap. The final tree left in the heap is the result Huffman tree.

1: **function** HUFFMAN'($A$)
2: BUILD-HEAP($A$)
3: **while** $|A| > 1$ **do**
4: $T_a \leftarrow$ HEAP-POP($A$)
5: $T_b \leftarrow$ HEAP-POP($A$)
6: HEAP-PUSH($A$, MERGE($T_a, T_b$))
7: **return** HEAP-POP($A$)

The following example C++ code implements this heap solution. The heap used here is provided in the standard library. Because the max-heap, but not min-heap would be made by default, a greater predication is explicitly passed as argument.

```
bool greaterp(Node* a, Node* b) { return b->w < a->w; }

Node* pop(Nodes& h) {
    Node* m = h.front();
    pop_heap(h.begin(), h.end(), greaterp);
    h.pop_back();
    return m;
}

void push(Node* t, Nodes& h) {
    h.push_back(t);
    push_heap(h.begin(), h.end(), greaterp);
}

Node* huffman1(Nodes ts) {
    make_heap(ts.begin(), ts.end(), greaterp);
    while (ts.size() > 1) {
        Node* t1 = pop(ts);
        Node* t2 = pop(ts);
        push(merge(t1, t2), ts);
    }
    return ts.front();
}
```

When the symbol-weight list has been already sorted, there exists a linear time method to build the Huffman tree. Observe that during the Huffman tree building, it produces a series of merged trees with weight in ascending order. We can use a queue to manage the merged trees. Every time, we pick the two trees with the smallest weight from both the queue and the list, merge them and push the result to the queue. All the trees in the list will be processed, and there will be only one tree left in the queue. This tree is the result Huffman tree. This process starts by passing an empty queue as below.

$$build'(A) = reduce'(extract''(\phi, A)) \tag{14.86}$$

Suppose $A$ is in ascending order by weight, At any time, the tree with the smallest weight is either the header of the queue, or the first element of the list. Denote the header of the queue is $T_a$, after pops it, the queue is $Q'$; The first element in $A$ is $T_b$, the rest elements are hold in $A'$. Function $extract''$ can be defined like the following.

$$extract''(Q, A) = \begin{cases} (T_b, (Q, A')) & : Q = \phi \\ (T_a, (Q', A)) & : A = \phi \lor T_a < T_b \\ (T_b, (Q, A')) & : otherwise \end{cases} \tag{14.87}$$

Actually, the pair of queue and tree list can be viewed as a special heap. The tree with the minimum weight is continuously extracted and merged.

$$
reduce'(T, (Q, A)) = \begin{cases} T & : Q = \phi \land A = \phi \\ reduce'(extract''(push(Q'', merge(T, T')), A'')) & : otherwise \end{cases} \tag{14.88}
$$

Where $(T', (Q'', A'')) = extract''(Q, A)$, which means extracting another tree. The following Haskell example program shows the implementation of this method. Note that this program explicitly sort the leaves, which isn't necessary if the leaves are ordered. Again, the list, but not a real queue is used here for illustration purpose. List isn't good at pushing new element, please refer to the chapter of queue for details about it.

```
huffman'' :: (Num a, Ord a) ⇒ [(b, a)] → HTr a b
huffman'' = reduce ∘ wrap ∘ sort ∘ map (λ(c, w) → Leaf w c) where
  wrap xs = delMin ([], xs)
  reduce (x, ([], [])) = x
  reduce (x, h) = let (y, (q, xs)) = delMin h in
                  reduce $ delMin (q ++ [merge x y], xs)
  delMin ([], (x:xs)) = (x, ([], xs))
  delMin ((q:qs), []) = (q, (qs, []))
  delMin ((q:qs), (x:xs)) | q < x = (q, (qs, (x:xs)))
                          | otherwise = (x, ((q:qs), xs))
```

This algorithm can also be realized imperatively.

1: **function** HUFFMAN”($A$) ▷ $A$ is ordered by weight
2: $Q \leftarrow \phi$
3: $T \leftarrow$ EXTRACT($Q, A$)
4: **while** $Q \neq \phi \lor A \neq \phi$ **do**
5: PUSH($Q$, MERGE($T$, EXTRACT($Q, A$)))
6: $T \leftarrow$ EXTRACT($Q, A$)
7: **return** $T$

Where function EXTRACT($Q, A$) extracts the tree with the smallest weight from the queue and the array of trees. It mutates the queue and array if necessary. Denote the head of the queue is $T_a$, and the first element of the array as $T_b$.

1: **function** EXTRACT($Q, A$)
2: **if** $Q \neq \phi \land (A = \phi \lor T_a < T_b)$ **then**
3: **return** POP($Q$)
4: **else**
5: **return** DETACH($A$)

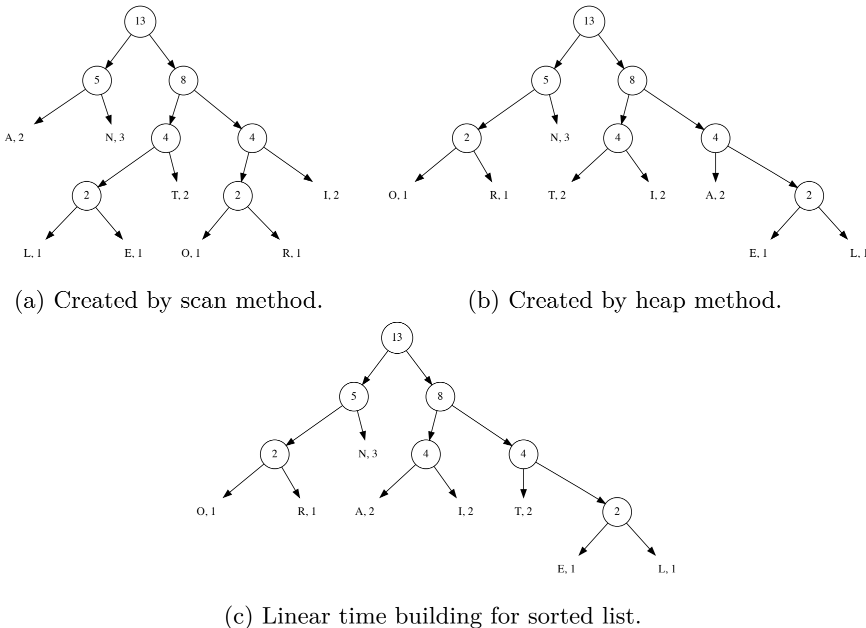
Where procedure DETACH($A$), removes the first element from $A$, and returns this element as result. In most imperative settings, as detaching the first element is slow linear operation for array, we can store the trees in descending order by weight, and remove the last element. This is a fast constant time operation. The below C++ example code shows this idea.

```
Node* extract(queue<Node*>& q, Nodes& ts) {
    Node* t;
    if (!q.empty() && (ts.empty() || lessp(q.front(), ts.back()))) {
        t = q.front();
        q.pop();
    } else {
        t = ts.back();
        ts.pop_back();
    }
    return t;
}
```

```
Node* huffman2(Nodes ts) {
    queue<Node*> q;
    sort(ts.begin(), ts.end(), greaterp);
    Node* t = extract(q, ts);
    while (!q.empty() || !ts.empty()) {
        q.push(merge(t, extract(q, ts)));
        t = extract(q, ts);
    }
    return t;
}
```

Note that the sorting isn't necessary if the trees have already been ordered. It can be a linear time reversing in case the trees are in ascending order by weight.

There are three different Huffman man tree building methods explained. Although they follow the same approach developed by Huffman, the result trees varies. Figure 14.49 shows the three different Huffman trees built with these methods.




(a) Created by scan method.

(b) Created by heap method.

(c) Linear time building for sorted list.

Figure 14.49: Variation of Huffman trees for the same symbol list.

Although these three trees are not identical. They are all able to generate the most efficient code. The formal proof is skipped here. The detailed information can be referred to [91] and Section 16.3 of [4].

The Huffman tree building is the core idea of Huffman coding. Many things can be easily achieved with the Huffman tree. For example, the code table can be generated by traversing the tree. We start from the root with the empty prefix $p$. For any branches, we append a zero to the prefix if turn left, and append a one if turn right. When a leaf node is arrived, the symbol represented by this node and the prefix are put to the code table. Denote the symbol of a leaf node as $c$, the children for tree $T$ as $T_l$ and $T_r$ respectively. The code table association list can be built with $code(T, \phi)$, which is defined as below.

$$code(T, p) = \begin{cases} \{(c, p)\} & : \ leaf(T) \\ code(T_l, p \cup \{0\}) \cup code(T_r, p \cup \{1\}) & : \ otherwise \end{cases} \tag{14.89}$$

Where function $leaf(T)$ tests if tree $T$ is a leaf or a branch node. The following Haskell example program generates a map as the code table according to this algorithm.

```
code tr = Map.fromList $ traverse [] tr where
  traverse bits (Leaf _ c) = [(c, bits)]
  traverse bits (Branch _ l r) = (traverse (bits ++ [0]) l) ++
                                 (traverse (bits ++ [1]) r)
```

The imperative code table generating algorithm is left as exercise. The encoding process can scan the text, and look up the code table to output the bit sequence. The realization is skipped here.

The decoding process is realized by looking up the Huffman tree according to the bit sequence. We start from the root, whenever a zero is received, we turn left, otherwise if a one is received, we turn right. If a leaf node is arrived, the symbol represented by this leaf is output, and we start another looking up from the root. The decoding process ends when all the bits are consumed. Denote the bit sequence as $B = \{b_1, b_2, ...\}$, all bits except the first one are hold in $B'$, below definition realizes the decoding algorithm.

$$decode(T, B) = \begin{cases} \{c\} & : B = \phi \land leaf(T) \\ \{c\} \cup decode(root(T), B) & : leaf(T) \\ decode(T_l, B') & : b_1 = 0 \\ decode(T_r, B') & : otherwise \end{cases} \tag{14.90}$$

Where $root(T)$ returns the root of the Huffman tree. The following Haskell example code implements this algorithm.

```
decode tr cs = find tr cs where
  find (Leaf _ c) [] = [c]
  find (Leaf _ c) bs = c : find tr bs
  find (Branch _ l r) (b:bs) = find (if b == 0 then l else r) bs
```

Note that this is an on-line decoding algorithm with linear time performance. It consumes one bit per time. This can be clearly noted from the below imperative realization, where the index keeps increasing by one.

```
1: function DECODE(T, B)
2:     W ← φ
3:     n ← |B|, i ← 1
4:     while i < n do
5:         R ← T
6:         while ¬ LEAF(R) do
7:             if B[i] = 0 then
8:                 R ← LEFT(R)
9:             else
10:                R ← RIGHT(R)
11:            i ← i + 1
12:        W ← W∪ SYMBOL(R)
13:    return W
```

This imperative algorithm can be implemented as the following example C++ program.

```
string decode(Node* root, const char* bits) {
    string w;
    while (*bits) {
        Node* t = root;
        while (!isleaf(t))
            t = '0' == *bits++ ? t->left : t->right;
        w += t->c;
```

```
    }
    return w;
}
```

Huffman coding, especially the Huffman tree building shows an interesting strategy. Each time, there are multiple options for merging. Among the trees in the list, Huffman method always selects two trees with the smallest weight. This is the best choice at that merge stage. However, these series of *local* best options generate a global optimal prefix code.

It's not always the case that the local optimal choice also leads to the global optimal solution. In most cases, it doesn't. Huffman coding is a special one. We call the strategy that always choosing the local best option as *greedy* strategy.

Greedy method works for many problems. However, it's not easy to tell if the greedy method can be applied to get the global optimal solution. The generic formal proof is still an active research area. Section 16.4 in[4] provides a good treatment for Matroid tool, which covers many problems that greedy algorithm can be applied.

### Change-making problem

We often change money when visiting other countries. People tend to use credit card more often nowadays than before, because it's quite convenient to buy things without considering much about changes. If we changed some money in the bank, there are often some foreign money left by the end of the trip. Some people like to change them to coins for collection. Can we find a solution, which can change the given amount of money with the least number of coins?

Let's use USA coin system for example. There are 5 different coins: 1 cent, 5 cent, 25 cent, 50 cent, and 1 dollar. A dollar is equal to 100 cents. Using the greedy method introduced above, we can always pick the largest coin which is not greater than the remaining amount of money to be changed. Denote list $C = \{1, 5, 25, 50, 100\}$, which stands for the value of coins. For any given money $X$, the change coins can be generated as below.

$$change(X, C) = \begin{cases} \phi & : X = 0 \\ \{c_m\} \cup change(X - c_m, C) & : \begin{array}{l} otherwise, \\ c_m = max(\{c \in C, c \leq X\}) \end{array} \end{cases} \tag{14.91}$$

If $C$ is in descending order, $c_m$ can be found as the first one not greater than $X$. If we want to change 1.42 dollar, This function produces a coin list of $\{100, 25, 5, 5, 5, 1, 1\}$. The output coins list can be easily transformed to contain pairs $\{(100, 1), (25, 1), (5, 3), (1, 2)\}$. That we need one dollar, a quarter, three coins of 5 cent, and 2 coins of 1 cent to make the change. The following Haskell example program outputs result as such.

```
solve x = assoc ∘ change x where
  change 0 _ = []
  change x cs = let c = head $ filter (≤ x) cs in c : change (x - c) cs

assoc = (map (λcs → (head cs, length cs))) ∘ group
```

As mentioned above, this program assumes the coins are in descending order, for instance like below.

```
solve 142 [100, 50, 25, 5, 1]
```

This algorithm is tail recursive, it can be transformed to a imperative looping.

1: **function** CHANGE($X, C$)
2: $\quad R \leftarrow \phi$

3: **while** $X \neq 0$ **do**
4: $c_m = max(\{c \in C, c \leq X\})$
5: $R \leftarrow \{c_m\} \cup R$
6: $X \leftarrow X - c_m$
7: **return** $R$

The following example Python program implements this imperative version and manages the result with a dictionary.

```
def change(x, coins):
    cs = {}
    while x ≠ 0:
        m = max([c for c in coins if c ≤ x])
        cs[m] = 1 + cs.setdefault(m, 0)
        x = x - m
    return cs
```

For a coin system like USA, the greedy approach can find the optimal solution. The amount of coins is the minimum. Fortunately, our greedy method works in most countries. But it is not always true. For example, suppose a country have coins of value 1, 3, and 4 units. The best change for value 6, is to use two coins of 3 units, however, the greedy method gives a result of three coins: one coin of 4, two coins of 1. Which isn't the optimal result.

### Summary of greedy method

As shown in the change making problem, greedy method doesn't always give the best result. In order to find the optimal solution, we need dynamic programming which will be introduced in the next section.

However, the result is often good enough in practice. Let's take the word-wrap problem for example. In modern software editors and browsers, text spans to multiple lines if the length of the content is too long to be hold. With word-wrap supported, user needn't hard line breaking. Although dynamic programming can wrap with the minimum number of lines, it's overkill. On the contrary, greedy algorithm can wrap with lines approximate to the optimal result with quite effective realization as below. Here it wraps text $T$, not to exceeds line width $W$, with space $s$ between each word.

1: $L \leftarrow W$
2: **for** $w \in T$ **do**
3: **if** $|w| + s > L$ **then**
4: Insert line break
5: $L \leftarrow W - |w|$
6: **else**
7: $L \leftarrow L - |w| - s$

For each word $w$ in the text, it uses a greedy strategy to put as many words in a line as possible unless it exceeds the line width. Many word processors use a similar algorithm to do word-wrapping.

There are many cases, the strict optimal result, but not the approximate one is necessary. Dynamic programming can help to solve such problems.

### Dynamic programming

In the change-making problem, we mentioned the greedy method can't always give the optimal solution. For any coin system, are there any way to find the best changes?

Suppose we have find the best solution which makes $X$ value of money. The coins needed are contained in $C_m$. We can partition these coins into two collections, $C_1$ and

$C_2$. They make money of $X_1$, and $X_2$ respectively. We'll prove that $C_1$ is the optimal solution for $X_1$, and $C_2$ is the optimal solution for $X_2$.

*Proof.* For $X_1$, Suppose there exists another solution $C_1'$, which uses less coins than $C_1$. Then changing solution $C_1' \cup C_2$ uses less coins to make $X$ than $C_m$. This is conflict with the fact that $C_m$ is the optimal solution to $X$. Similarity, we can prove $C_2$ is the optimal solution to $X_2$. □

Note that it is not true in the reverse situation. If we arbitrary select a value $Y < X$, divide the original problem to find the optimal solutions for sub problems $Y$ and $X - Y$. Combine the two optimal solutions doesn't necessarily yield optimal solution for $X$. Consider this example. There are coins with value 1, 2, and 4. The optimal solution for making value 6, is to use 2 coins of value 2, and 4; However, if we divide $6 = 3 + 3$, since each 3 can be made with optimal solution $3 = 1 + 2$, the combined solution contains 4 coins (1 + 1 + 2 + 2).

If an optimal problem can be divided into several sub optimal problems, we call it has optimal substructure. We see that the change-making problem has optimal substructure. But the dividing has to be done based on the coins, but not with an arbitrary value.

The optimal substructure can be expressed recursively as the following.

$$change(X) = \begin{cases} \phi & : X = 0 \\ least(\{c \cup change(X - c) | c \in C, c \leq X\}) & : otherwise \end{cases} \tag{14.92}$$

For any coin system $C$, the changing result for zero is empty; otherwise, we check every candidate coin $c$, which is not greater then value $X$, and recursively find the best solution for $X - c$; We pick the coin collection which contains the least coins as the result.

Below Haskell example program implements this top-down recursive solution.

```
change _ 0 = []
change cs x = minimumBy (compare `on` length)
                  [c:change cs (x - c) | c ← cs, c ≤ x]
```

Although this program outputs correct answer `[2, 4]` when evaluates `change [1, 2, 4] 6`, it performs very bad when changing 1.42 dollar with USA coins system. It failed to find the answer within 15 minutes in a computer with 2.7GHz CPU and 8G memory.

The reason why it's slow is because there are a lot of duplicated computing in the top-down recursive solution. When it computes $change(142)$, it needs to examine $change(141)$, $change(137)$ and $change(42)$. While $change(141)$ next computes to smaller values by deducing with 1, 2, 25, 50 and 100 cents. it will eventually meets value 137, 117, 92, and 42 again. The search space explodes with power of 5.

This is quite similar to compute Fibonacci numbers in a top-down recursive way.

$$F_n = \begin{cases} 1 & : n = 1 \vee n = 2 \\ F_{n-1} + F_{n-2} & : otherwise \end{cases} \tag{14.93}$$

When we calculate $F_8$ for example, we recursively calculate $F_7$ and $F_6$. While when we calculate $F_7$, we need calculate $F_6$ again, and $F_5$, ... As shown in the below expand forms, the calculation is doubled every time, and the same value is calculate again and again.

$$\begin{array}{rl} F_8 = & F_7 + F_6 \\ = & F_6 + F_5 + F_5 + F_4 \\ = & F_5 + F_4 + F_4 + F_3 + F_4 + F_3 + F_3 + F_2 \\ = & ... \end{array}$$

In order to avoid duplicated computation, a table $F$ can be maintained when calculating the Fibonacci numbers. The first two elements are filled as 1, all others are left blank. During the top-down recursive calculation, If need $F_k$, we first look up this table for the $k$-th cell, if it isn't blank, we use that value directly. Otherwise we need further calculation. Whenever a value is calculated, we store it in the corresponding cell for looking up in the future.

1: $F \leftarrow \{1, 1, NIL, NIL, ...\}$
2: **function** FIBONACCI($n$)
3:     **if** $n > 2 \land F[n] = NIL$ **then**
4:         $F[n] \leftarrow$ FIBONACCI($n-1$) + FIBONACCI($n-2$)
5:     **return** $F[n]$

By using the similar idea, we can develop a new top-down change-making solution. We use a table $T$ to maintain the best changes, it is initialized to all empty coin list. During the top-down recursive computation, we look up this table for smaller changing values. Whenever a intermediate value is calculated, it is stored in the table.

1: $T \leftarrow \{\phi, \phi, ...\}$
2: **function** CHANGE($X$)
3:     **if** $X > 0 \land T[X] = \phi$ **then**
4:         **for** $c \in C$ **do**
5:             **if** $c \leq X$ **then**
6:                 $C_m \leftarrow \{c\} \cup$ CHANGE($X - c$)
7:                 **if** $T[X] = \phi \lor |C_m| < |T[X]|$ **then**
8:                     $T[X] \leftarrow C_m$
9:     **return** $T[X]$

The solution to change 0 money is definitely empty $\phi$, otherwise, we look up $T[X]$ to retrieve the solution to change $X$ money. If it is empty, we need recursively calculate it. We examine all coins in the coin system $C$ which is not greater than $X$. This is the sub problem of making changes for money $X - c$. The minimum amount of coins plus one coin of $c$ is stored in $T[X]$ finally as the result.

The following example Python program implements this algorithm just takes 8000 ms to give the answer of changing 1.42 dollar in US coin system.

```
tab = [[] for _ in range(1000)]

def change(x, cs):
    if x > 0 and tab[x] == []:
        for s in [[c] + change(x - c, cs) for c in cs if c <= x]:
            if tab[x] == [] or len(s) < len(tab[x]):
                tab[x] = s
    return tab[x]
```

Another solution to calculate Fibonacci number, is to compute them in order of $F_1, F_2, F_3, ..., F_n$. This is quite natural when people write down Fibonacci series.

1: **function** FIBO($n$)
2:     $F = \{1, 1, NIL, NIL, ...\}$
3:     **for** $i \leftarrow 3$ to $n$ **do**
4:         $F[i] \leftarrow F[i-1] + F[i-2]$
5:     **return** $F[n]$

We can use the quite similar idea to solve the change making problem. Starts from zero money, which can be changed by an empty list of coins, we next try to figure out how to change money of value 1. In US coin system for example, A cent can be used; The next values of 2, 3, and 4, can be changed by two coins of 1 cent, three coins of 1 cent,

and 4 coins of 1 cent. At this stage, the solution table looks like below

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $\phi$ | $\{1\}$ | $\{1,1\}$ | $\{1,1,1\}$ | $\{1,1,1,1\}$ |

The interesting case happens for changing value 5. There are two options, use another coin of 1 cent, which need 5 coins in total; The other way is to use 1 coin of 5 cent, which uses less coins than the former. So the solution table can be extended to this.

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $\phi$ | $\{1\}$ | $\{1,1\}$ | $\{1,1,1\}$ | $\{1,1,1,1\}$ | $\{5\}$ |

For the next change value 6, since there are two types of coin, 1 cent and 5 cent, are less than this value, we need examine both of them.

- If we choose the 1 cent coin, we need next make changes for 5; Since we've already known that the best solution to change 5 is $\{5\}$, which only needs a coin of 5 cents, by looking up the solution table, we have one candidate solution to change 6 as $\{5,1\}$;
- The other option is to choose the 5 cent coin, we need next make changes for 1; By looking up the solution table we've filled so far, the sub optimal solution to change 1 is $\{1\}$. Thus we get another candidate solution to change 6 as $\{1,5\}$;

It happens that, both options yield a solution of two coins, we can select either of them as the best solution. Generally speaking, the candidate with fewest number of coins is selected as the solution, and filled into the table.

At any iteration, when we are trying to change the $i < X$ value of money, we examine all the types of coin. For any coin $c$ not greater than $i$, we look up the solution table to fetch the sub solution $T[i-c]$. The number of coins in this sub solution plus the one coin of $c$ are the total coins needed in this candidate solution. The fewest candidate is then selected and updated to the solution table.

The following algorithm realizes this bottom-up idea.

1: **function** CHANGE($X$)
2: $\quad T \leftarrow \{\phi, \phi, ...\}$
3: $\quad$ **for** $i \leftarrow 1$ to $X$ **do**
4: $\quad\quad$ **for** $c \in C, c \leq i$ **do**
5: $\quad\quad\quad$ **if** $T[i] = \phi \lor 1 + |T[i-c]| < |T[i]|$ **then**
6: $\quad\quad\quad\quad T[i] \leftarrow \{c\} \cup T[i-c]$
7: $\quad$ **return** $T[X]$

This algorithm can be directly translated to imperative programs, like Python for example.

```
def changemk(x, cs):
    s = [[] for _ in range(x+1)]
    for i in range(1, x+1):
        for c in cs:
            if c <= i and (s[i] == [] or 1 + len(s[i-c]) < len(s[i])):
                s[i] = [c] + s[i-c]
    return s[x]
```

Observe the solution table, it's easy to find that, there are many duplicated contents being stored.

| 6 | 7 | 8 | 9 | 10 | ... |
|---|---|---|---|---|---|
| $\{1,5\}$ | $\{1,1,5\}$ | $\{1,1,1,5\}$ | $\{1,1,1,1,5\}$ | $\{5,5\}$ | ... |

This is because the optimal sub solutions are completely copied and saved in parent solution. In order to use less space, we can only record the 'delta' part from the sub optimal solution. In change-making problem, it means that we only need to record the coin being selected for value $i$.

1: **function** CHANGE'($X$)
2: $T \leftarrow \{0, \infty, \infty, ...\}$
3: $S \leftarrow \{NIL, NIL, ...\}$
4: **for** $i \leftarrow 1$ to $X$ **do**
5: **for** $c \in C, c \leq i$ **do**
6: **if** $1 + T[i - c] < T[i]$ **then**
7: $T[i] \leftarrow 1 + T[i - c]$
8: $S[i] \leftarrow c$
9: **while** $X > 0$ **do**
10: PRINT($S[X]$)
11: $X \leftarrow X - S[X]$

Instead of recording the complete solution list of coins, this new algorithm uses two tables $T$ and $S$. $T$ holds the minimum number of coins needed for changing value 0, 1, 2, ...; while $S$ holds the first coin being selected for the optimal solution. For the complete coin list to change money $X$, the first coin is thus $S[X]$, the sub optimal solution is to change money $X' = X - S[X]$. We can look up table $S[X']$ for the next coin. The coins for sub optimal solutions are repeatedly looked up like this till the beginning of the table. Below Python example program implements this algorithm.

```
def chgmk(x, cs):
    cnt = [0] + [x+1] * x
    s = [0]
    for i in range(1, x+1):
        coin = 0
        for c in cs:
            if c <= i and 1 + cnt[i-c] < cnt[i]:
                cnt[i] = 1 + cnt[i-c]
                coin = c
        s.append(coin)
    r = []
    while x > 0:
        r.append(s[x])
        x = x - s[x]
    return r
```

This change-making solution loops $n$ times for given money $n$. It examines at most the full coin system in each iteration. The time is bound to $\Theta(nk)$ where $k$ is the number of coins for a certain coin system. The last algorithm adds $O(n)$ spaces to record sub optimal solutions with table $T$ and $S$.

In purely functional settings, There is no means to mutate the solution table and look up in constant time. One alternative is to use finger tree as we mentioned in previous chapter [12]. We can store the minimum number of coins, and the coin leads to the sub optimal solution in pairs.

The solution table, which is a finger tree, is initialized as $T = \{(0, 0)\}$. It means change 0 money need no coin. We can fold on list $\{1, 2, ..., X\}$, start from this table, with a binary function $change(T, i)$. The folding will build the solution table, and we can construct the coin list from this table by function $make(X, T)$.

$$makeChange(X) = make(X, fold(change, \{(0, 0)\}, \{1, 2, ..., X\})) \tag{14.94}$$

In function $change(T, i)$, all the coins not greater than $i$ are examined to select the one lead to the best result. The fewest number of coins, and the coin being selected are

[12]Some purely functional programming environments, Haskell for instance, provide built-in array; while other almost pure ones, such as ML, provide mutable array

formed to a pair. This pair is inserted to the finger tree, so that a new solution table is returned.

$$change(T, i) = insert(T, fold(sel, (\infty, 0), \{c | c \in C, c \leq i\})) \tag{14.95}$$

Again, folding is used to select the candidate with the minimum number of coins. This folding starts with initial value $(\infty, 0)$, on all valid coins. function $sel((n, c), c')$ accepts two arguments, one is a pair of length and coin, which is the best solution so far; the other is a candidate coin, it examines if this candidate can make better solution.

$$sel((n, c), c') = \begin{cases} (1 + n', c') & : \quad 1 + n' < n, (n', c') = T[i - c'] \\ (n, c) & : \quad otherwise \end{cases} \tag{14.96}$$

After the solution table is built, the coins needed can be generated from it.

$$make(X, T) = \begin{cases} \phi & : \quad X = 0 \\ \{c\} \cup make(X - c, T) & : \quad otherwise, (n, c) = T[X] \end{cases} \tag{14.97}$$

The following example Haskell program uses `Data.Sequence`, which is the library of finger tree, to implement change making solution.

```
import Data.Sequence (Seq, singleton, index, (|>))

changemk x cs = makeChange x $ foldl change (singleton (0, 0)) [1..x] where
  change tab i = let sel c = min (1 + fst (index tab (i - c)), c)
                 in tab |> (foldr sel ((x + 1), 0) $ filter (≤i) cs)
  makeChange 0 _ = []
  makeChange x tab = let c = snd $ index tab x in c : makeChange (x - c) tab
```

It's necessary to memorize the optimal solution to sub problems no matter using the top-down or the bottom-up approach. This is because a sub problem is used many times when computing the overall optimal solution. Such properties are called overlapping sub problems.

### Properties of dynamic programming

Dynamic programming was originally named by Richard Bellman in 1940s. It is a powerful tool to search for optimal solution for problems with two properties.

- Optimal sub structure. The problem can be broken down into smaller problems, and the optimal solution can be constructed efficiently from solutions of these sub problems;
- Overlapping sub problems. The problem can be broken down into sub problems which are reused several times in finding the overall solution.

The change-making problem, as we've explained, has both optimal sub structures, and overlapping sub problems.

### Longest common subsequence problem

The longest common subsequence problem, is different with the longest common substring problem. We've show how to solve the later in the chapter of suffix tree. The longest common subsequence needn't be consecutive part of the original sequence.

For example, The longest common substring for text "Mississippi", and "Missunderstanding" is "Miss", while the longest common subsequence for them are "Misssi". This is shown in figure 14.50.

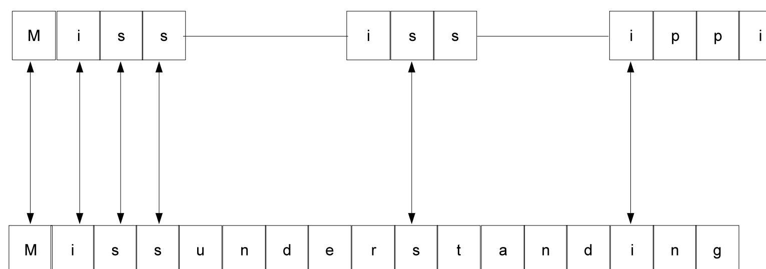



Figure 14.50: The longest common subsequence

If we rotate the figure vertically, and consider the two texts as two pieces of source code, it turns to be a 'diff' result between them. Most modern version control tools need calculate the difference content among the versions. The longest common subsequence problem plays a very important role.

If either one of the two strings $X$ and $Y$ is empty, the longest common subsequence $LCS(X, Y)$ is definitely empty; Otherwise, denote $X = \{x_1, x_2, ..., x_n\}$, $Y = \{y_1, y_2, ..., y_m\}$, if the first elements $x_1$ and $y_1$ are same, we can recursively find the longest subsequence of $X' = \{x_2, x_3, ..., x_n\}$ and $Y' = \{y_2, y_3, ..., y_m\}$. And the final result $LCS(X, Y)$ can be constructed by concatenating $x_1$ with $LCS(X', Y')$; Otherwise if $x_1 \neq y_1$, we need recursively find the longest common subsequences of $LCS(X, Y')$ and $LCS(X', Y)$, and pick the longer one as the final result. Summarize these cases gives the below definition.

$$LCS(X, Y) = \begin{cases} \phi & : X = \phi \lor Y = \phi \\ \{x_1\} \cup LCS(X', Y') & : x_1 = y_1 \\ longer(LCS(X, Y'), LCS(X', Y)) & : otherwise \end{cases} \quad (14.98)$$

Note that this algorithm shows clearly the optimal substructure, that the longest common subsequence problem can be broken to smaller problems. The sub problem is ensured to be at least one element shorter than the original one.

It's also clear that, there are overlapping sub-problems. The longest common subsequences to the sub strings are used multiple times in finding the overall optimal solution.

The existence of these two properties, the optimal substructure and the overlapping sub-problem, indicates the dynamic programming can be used to solve this problem.

A 2-dimension table can be used to record the solutions to the sub-problems. The rows and columns represent the substrings of $X$ and $Y$ respectively.

| | | a | n | t | e | n | n | a |
|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| b | 1 | | | | | | | |
| a | 2 | | | | | | | |
| n | 3 | | | | | | | |
| a | 4 | | | | | | | |
| n | 5 | | | | | | | |
| a | 6 | | | | | | | |

This table shows an example of finding the longest common subsequence for strings "antenna" and "banana". Their lengths are 7, and 6. The right bottom corner of this table is looked up first, Since it's empty we need compare the 7th element in "antenna" and the 6th in "banana", they are both 'a', Thus we need next recursively look up the cell at row 5, column 6; It's still empty, and we repeated this till either get a trivial case that one substring becomes empty, or some cell we are looking up has been filled before. Similar to the change-making problem, whenever the optimal solution for a sub-problem is found, it is recorded in the cell for further reusing. Note that this process is in the reversed order comparing to the recursive equation given above, that we start from the right most element of each string.

Considering that the longest common subsequence for any empty string is still empty, we can extended the solution table so that the first row and column hold the empty strings.

| | | a | n | t | e | n | n | a |
|---|---|---|---|---|---|---|---|---|
| | | $\phi$ | $\phi$ | $\phi$ | $\phi$ | $\phi$ | $\phi$ | $\phi$ |
| b | $\phi$ | | | | | | | |
| a | $\phi$ | | | | | | | |
| n | $\phi$ | | | | | | | |
| a | $\phi$ | | | | | | | |
| n | $\phi$ | | | | | | | |
| a | $\phi$ | | | | | | | |

Below algorithm realizes the top-down recursive dynamic programming solution with such a table.

1: $T \leftarrow$ NIL
2: **function** LCS$(X, Y)$
3: $\quad m \leftarrow |X|, n \leftarrow |Y|$
4: $\quad m' \leftarrow m + 1, n' \leftarrow n + 1$
5: $\quad$ **if** $T =$ NIL **then**
6: $\quad\quad T \leftarrow \{\{\phi, \phi, ..., \phi\}, \{\phi, NIL, NIL, ...\}, ...\}$ $\quad\triangleright\ m' \times n'$
7: $\quad$ **if** $X \neq \phi \wedge Y \neq \phi \wedge T[m'][n'] =$ NIL **then**
8: $\quad\quad$ **if** $X[m] = Y[n]$ **then**
9: $\quad\quad\quad T[m'][n'] \leftarrow$ APPEND(LCS$(X[1..m-1], Y[1..n-1])$, $X[m]$)
10: $\quad\quad$ **else**
11: $\quad\quad\quad T[m'][n'] \leftarrow$ LONGER(LCS$(X, Y[1..n-1])$, LCS$(X[1..m-1], Y)$)
12: $\quad$ **return** $T[m'][n']$

The table is firstly initialized with the first row and column filled with empty strings; the rest are all NIL values. Unless either string is empty, or the cell content isn't NIL, the last two elements of the strings are compared, and recursively computes the longest common subsequence with substrings. The following Python example program implements this algorithm.

```
def lcs(xs, ys):
    m = len(xs)
    n = len(ys)
```

```
    global tab
    if tab is None:
        tab = [[""]*(n+1)] + [[""] + [None]*n for _ in xrange(m)]
    if m != 0 and n !=0 and tab[m][n] is None:
        if xs[-1] == ys[-1]:
            tab[m][n] = lcs(xs[:-1], ys[:-1]) + xs[-1]
        else:
            (a, b) = (lcs(xs, ys[:-1]), lcs(xs[:-1], ys))
            tab[m][n] = a if len(b) < len(a) else b
    return tab[m][n]
```

The longest common subsequence can also be found in a bottom-up manner as what we've done with the change-making problem. Besides that, instead of recording the whole sequences in the table, we can just store the lengths of the longest subsequences, and later construct the subsubsequence with this table and the two strings. This time, the table is initialized with all values set as 0.

1: **function** LCS($X, Y$)
2: $m \leftarrow |X|, n \leftarrow |Y|$
3: $T \leftarrow \{\{0, 0, ...\}, \{0, 0, ...\}, ...\}$ ▷ $(m + 1) \times (n + 1)$
4: **for** $i \leftarrow 1$ to $m$ **do**
5: **for** $j \leftarrow 1$ to $n$ **do**
6: **if** $X[i] = Y[j]$ **then**
7: $T[i + 1][j + 1] \leftarrow T[i][j] + 1$
8: **else**
9: $T[i + 1][j + 1] \leftarrow \text{MAX}(T[i][j + 1], T[i + 1][j])$
10: **return** GET($T, X, Y, m, n$)

11: **function** GET($T, X, Y, i, j$)
12: **if** $i = 0 \lor j = 0$ **then**
13: **return** $\phi$
14: **else if** $X[i] = Y[j]$ **then**
15: **return** APPEND(GET($T, X, Y, i - 1, j - 1$), $X[i]$)
16: **else if** $T[i - 1][j] > T[i][j - 1]$ **then**
17: **return** GET($T, X, Y, i - 1, j$)
18: **else**
19: **return** GET($T, X, Y, i, j - 1$)

In the bottom-up approach, we start from the cell at the second row and the second column. The cell is corresponding to the first element in both $X$, and $Y$. If they are same, the length of the longest common subsequence so far is 1. This can be yielded by increasing the length of empty sequence, which is stored in the top-left cell, by one; Otherwise, we pick the maximum value from the upper cell and left cell. The table is repeatedly filled in this manner.

After that, a back-track is performed to construct the longest common subsequence. This time we start from the bottom-right corner of the table. If the last elements in $X$ and $Y$ are same, we put this element as the last one of the result, and go on looking up the cell along the diagonal line; Otherwise, we compare the values in the left cell and the above cell, and go on looking up the cell with the bigger value.

The following example Python program implements this algorithm.

```
def lcs(xs, ys):
    m = len(xs)
    n = len(ys)
    c = [[0]*(n+1) for _ in xrange(m+1)]
    for i in xrange(1, m+1):
        for j in xrange(1, n+1):
```

```
            if xs[i-1] == ys[j-1]:
                c[i][j] = c[i-1][j-1] + 1
            else:
                c[i][j] = max(c[i-1][j], c[i][j-1])

    return get(c, xs, ys, m, n)

def get(c, xs, ys, i, j):
    if i==0 or j==0:
        return []
    elif xs[i-1] == ys[j-1]:
        return get(c, xs, ys, i-1, j-1) + [xs[i-1]]
    elif c[i-1][j] > c[i][j-1]:
        return get(c, xs, ys, i-1, j)
    else:
        return get(c, xs, ys, i, j-1)
```

The bottom-up dynamic programming solution can also be defined in purely functional way. The finger tree can be used as a table. The first row is filled with $n+1$ zero values. This table can be built by folding on sequence $X$. Then the longest common subsequence is constructed from the table.

$$LCS(X, Y) = construct(fold(f, \{\{0, 0, ..., 0\}\}, zip(\{1, 2, ...\}, X))) \tag{14.99}$$

Note that, since the table need be looked up by index, $X$ is zipped with natural numbers. Function $f$ creates a new row of this table by folding on sequence $Y$, and records the lengths of the longest common sequence for all possible cases so far.

$$f(T, (i, x)) = insert(T, fold(longest, \{0\}, zip(\{1, 2, ...\}, Y))) \tag{14.100}$$

Function *longest* takes the intermediate filled row result, and a pair of index and element in $Y$, it compares if this element is the same as the one in $X$. Then fills the new cell with the length of the longest one.

$$longest(R, (j, y)) = \begin{cases} insert(R, 1 + T[i-1][j-1]) & : x = y \\ insert(R, max(T[i-1][j], T[i][j-1])) & : otherwise \end{cases} \tag{14.101}$$

After the table is built. The longest common sub sequence can be constructed recursively by looking up this table. We can pass the reversed sequences $\overleftarrow{X}$, and $\overleftarrow{Y}$ together with their lengths $m$ and $n$ for efficient building.

$$construct(T) = get((\overleftarrow{X}, m), (\overleftarrow{Y}, n)) \tag{14.102}$$

If the sequences are not empty, denote the first elements as $x$ and $y$. The rest elements are hold in $\overleftarrow{X}'$ and $\overleftarrow{Y}'$ respectively. The function *get* can be defined as the following.

$$get((\overleftarrow{X}, i), (\overleftarrow{Y}, j)) = \begin{cases} \phi & : \overleftarrow{X} = \phi \land \overleftarrow{Y} = \phi \\ get((\overleftarrow{X}', i-1), (\overleftarrow{Y}', j-1)) \cup \{x\} & : x = y \\ get((\overleftarrow{X}', i-1), (\overleftarrow{Y}, j)) & : T[i-1][j] > T[i][j-1] \\ get((\overleftarrow{X}, i), (\overleftarrow{Y}', j-1)) & : otherwise \end{cases} \tag{14.103}$$

Below Haskell example program implements this solution.

```
lcs' xs ys = construct $ foldl f (singleton $ fromList $ replicate (n+1) 0)
                                (zip [1..] xs) where
  (m, n) = (length xs, length ys)
  f tab (i, x) = tab |> (foldl longer (singleton 0) (zip [1..] ys)) where
    longer r (j, y) = r |> if x == y
```

```
                    then 1 + (tab `index` (i-1) `index` (j-1))
                    else max (tab `index` (i-1) `index` j) (r `index` (j-1))
  construct tab = get (reverse xs, m) (reverse ys, n) where
    get ([], 0) ([], 0) = []
    get ((x:xs), i) ((y:ys), j)
      | x == y = get (xs, i-1) (ys, j-1) ++ [x]
      | (tab `index` (i-1) `index` j) > (tab `index` i `index` (j-1)) =
                  get (xs, i-1) ((y:ys), j)
      | otherwise = get ((x:xs), i) (ys, j-1)
```

### Subset sum problem

Dynamic programming does not limit to solve the optimization problem, but can also solve some more general searching problems. Subset sum problem is such an example. Given a set of integers, is there a non-empty subset sums to zero? for example, there are two subsets of $\{11, 64, -82, -68, 86, 55, -88, -21, 51\}$ both sum to zero. One is $\{64, -82, 55, -88, 51\}$, the other is $\{64, -82, -68, 86\}$.

Of course summing to zero is a special case, because sometimes, people want to find a subset, whose sum is a given value $s$. Here we are going to develop a method to find all the candidate subsets.

There is obvious a brute-force exhausting search solution. For every element, we can either pick it or not. So there are total $2^n$ options for set with $n$ elements. Because for every selection, we need check if it sums to $s$. This is a linear operation. The overall complexity is bound to $O(n2^n)$. This is the exponential algorithm, which takes very huge time if the set is big.

There is a recursive solution to subset sum problem. If the set is empty, there is no solution definitely; Otherwise, let the set is $X = \{x_1, x_2, ...\}$. If $x_1 = s$, then subset $\{x_1\}$ is a solution, we need next search for subsets $X' = \{x_2, x_3, ...\}$ for those sum to $s$; Otherwise if $x_1 \neq s$, there are two different kinds of possibilities. We need search $X'$ for both sum $s$, and sum $s - x_1$. For any subset sum to $s - x_1$, we can add $x_1$ to it to form a new set as a solution. The following equation defines this algorithm.

$$solve(X, s) = \begin{cases} \phi & : X = \phi \\ \{\{x_1\}\} \cup solve(X', s) & : x_1 = s \\ solve(X', s) \cup \{\{x_1\} \cup S | S \in solve(X', s - x_1)\} & : otherwise \end{cases} \tag{14.104}$$

There are clear substructures in this definition, although they are not in a sense of optimal. And there are also overlapping sub-problems. This indicates the problem can be solved with dynamic programming with a table to memorize the solutions to sub-problems.

Instead of developing a solution to output all the subsets directly, let's consider how to give the existence answer firstly. That output 'yes' if there exists some subset sum to $s$, and 'no' otherwise.

One fact is that, the upper and lower limit for all possible answer can be calculated in one scan. If the given sum $s$ doesn't belong to this range, there is no solution obviously.

$$\begin{cases} s_l = \sum\{x \in X, x < 0\} \\ s_u = \sum\{x \in X, x > 0\} \end{cases} \tag{14.105}$$

Otherwise, if $s_l \leq s \leq s_u$, since the values are all integers, we can use a table, with $s_u - s_l + 1$ columns, each column represents a possible value in this range, from $s_l$ to $s_u$. The value of the cell is either true or false to represents if there exists subset sum to this value. All cells are initialized as false. Starts from the first element $x_1$ in $X$, definitely, set $\{x_1\}$ can sum to $x_1$, so that the cell represents this value in the first row can be filled as true.

| | $s_l$ | $s_l+1$ | ... | $x_1$ | ... | $s_u$ |
|---|---|---|---|---|---|---|
| $x_1$ | F | F | ... | T | ... | F |

With the next element $x_2$, There are three possible sums. Similar as the first row, $\{x_2\}$ sums to $x_2$; For all possible sums in previous row, they can also been achieved without $x_2$. So the cell below to $x_1$ should also be filled as true; By adding $x_2$ to all possible sums so far, we can also get some new values. That the cell represents $x_1 + x_2$ should be true.

| | $s_l$ | $s_l+1$ | ... | $x_1$ | ... | $x_2$ | ... | $x_1+x_2$ | ... | $s_u$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $x_1$ | F | F | ... | T | ... | F | ... | F | ... | F |
| $x_2$ | F | F | ... | T | ... | T | ... | T | ... | F |

Generally speaking, when fill the $i$-th row, all the possible sums constructed with $\{x_1, x_2, ..., x_{i-1}\}$ so far can also be achieved with $x_i$. So the cells previously are true should also be true in this new row. The cell represents value $x_i$ should also be true since the singleton set $\{x_i\}$ sums to it. And we can also adds $x_i$ to all previously constructed sums to get the new results. Cells represent these new sums should also be filled as true.

When all the elements are processed like this, a table with $|X|$ rows is built. Looking up the cell represents $s$ in the last row tells if there exists subset can sum to this value. As mentioned above, there is no solution if $s < s_l$ or $s_u < s$. We skip handling this case for the sake of brevity.

1: **function** SUBSET-SUM$(X, s)$
2: $s_l \leftarrow \sum\{x \in X, x < 0\}$
3: $s_u \leftarrow \sum\{x \in X, x > 0\}$
4: $n \leftarrow |X|$
5: $T \leftarrow \{\{False, False, ...\}, \{False, False, ...\}, ...\}$ ▷ $n \times (s_u - s_l + 1)$
6: **for** $i \leftarrow 1$ to $n$ **do**
7: **for** $j \leftarrow s_l$ to $s_u$ **do**
8: **if** $X[i] = j$ **then**
9: $T[i][j] \leftarrow True$
10: **if** $i > 1$ **then**
11: $T[i][j] \leftarrow T[i][j] \vee T[i-1][j]$
12: $j' \leftarrow j - X[i]$
13: **if** $s_l \leq j' \leq s_u$ **then**
14: $T[i][j] \leftarrow T[i][j] \vee T[i-1][j']$
15: **return** $T[n][s]$

Note that the index to the columns of the table, doesn't range from 1 to $s_u - s_l + 1$, but maps directly from $s_l$ to $s_u$. Because most programming environments don't support negative index, this can be dealt with $T[i][j - s_l]$. The following example Python program utilizes the property of negative indexing.

```
def solve(xs, s):
    low = sum([x for x in xs if x < 0])
    up  = sum([x for x in xs if x > 0])
    tab = [[False]*(up-low+1) for _ in xs]
    for i in xrange(0, len(xs)):
        for j in xrange(low, up+1):
            tab[i][j] = (xs[i] == j)
            j1 = j - xs[i];
            tab[i][j] = tab[i][j] or tab[i-1][j] or
                        (low ≤ j1 and j1 ≤ up and tab[i-1][j1])
    return tab[-1][s]
```

Note that this program doesn't use different branches for $i = 0$ and $i = 1, 2, ..., n-1$. This is because when $i = 0$, the row index to $i - 1 = -1$ refers to the last row in the table, which are all false. This simplifies the logic one more step.

With this table built, it's easy to construct all subsets sum to $s$. The method is to

look up the last row for cell represents $s$. If the last element $x_n = s$, then $\{x_n\}$ definitely is a candidate. We next look up the previous row for $s$, and recursively construct all the possible subsets sum to $s$ with $\{x_1, x_2, x_3, ..., x_{n-1}\}$. Finally, we look up the second last row for cell represents $s - x_n$. And for every subset sums to this value, we add element $x_n$ to construct a new subset, which sums to $s$.

1: **function** GET($X, s, T, n$)
2: $S \leftarrow \phi$
3: **if** $X[n] = s$ **then**
4: $S \leftarrow S \cup \{X[n]\}$
5: **if** $n > 1$ **then**
6: **if** $T[n-1][s]$ **then**
7: $S \leftarrow S\cup$ GET($X, s, T, n-1$)
8: **if** $T[n-1][s - X[n]]$ **then**
9: $S \leftarrow S \cup \{\{X[n]\} \cup S' | S' \in$ GET$(X, s - X[n], T, n-1)$ $\}$
10: **return** $S$

The following Python example program translates this algorithm.

```
def get(xs, s, tab, n):
    r = []
    if xs[n] == s:
        r.append([xs[n]])
    if n > 0:
        if tab[n-1][s]:
            r = r + get(xs, s, tab, n-1)
        if tab[n-1][s - xs[n]]:
            r = r + [[xs[n]] + ys for ys in get(xs, s - xs[n], tab, n-1)]
    return r
```

This dynamic programming solution to subset sum problem loops $O(n(s_u - s_l + 1))$ times to build the table, and recursively uses $O(n)$ time to construct the final solution from this table. The space it used is also bound to $O(n(s_u - s_l + 1))$.

Instead of using table with $n$ rows, a vector can be used alternatively. For every cell represents a possible sum, the list of subsets are stored. This vector is initialized to contain all empty sets. For every element in $X$, we update the vector, so that it records all the possible sums which can be built so far. When all the elements are considered, the cell corresponding to $s$ contains the final result.

1: **function** SUBSET-SUM($X, s$)
2: $s_l \leftarrow \sum\{x \in X, x < 0\}$
3: $s_u \leftarrow \sum\{x \in X, x > 0\}$
4: $T \leftarrow \{\phi, \phi, ...\}$ ▷ $s_u - s_l + 1$
5: **for** $x \in X$ **do**
6: $T' \leftarrow$ DUPLICATE($T$)
7: **for** $j \leftarrow s_l$ to $s_u$ **do**
8: $j' \leftarrow j - x$
9: **if** $x = j$ **then**
10: $T'[j] \leftarrow T'[j] \cup \{x\}$
11: **if** $s_l \leq j' \leq s_u \wedge T[j'] \neq \phi$ **then**
12: $T'[j] \leftarrow T'[j] \cup \{\{x\} \cup S | S \in T[j']\}$
13: $T \leftarrow T'$
14: **return** $T[s]$

The corresponding Python example program is given as below.

```
def subsetsum(xs, s):
    low = sum([x for x in xs if x < 0])
```

```
    up  = sum([x for x in xs if x > 0])
    tab = [[] for _ in xrange(low, up+1)]
    for x in xs:
        tab1 = tab[:]
        for j in xrange(low, up+1):
            if x == j:
                tab1[j].append([x])
            j1 = j - x
            if low ≤ j1 and j1 ≤ up and tab[j1] ≠ []:
                tab1[j] = tab1[j] + [[x] + ys for ys in tab[j1]]
        tab = tab1
    return tab[s]
```

This imperative algorithm shows a clear structure, that the solution table is built by looping every element. This can be realized in purely functional way by folding. A finger tree can be used to represents the vector spans from $s_l$ to $s_u$. It is initialized with all empty values as in the following equation.

$$subsetsum(X, s) = fold(build, \{\phi, \phi, ..., \}, X)[s] \tag{14.106}$$

After folding, the solution table is built, the answer is looked up at cell $s$[13].

For every element $x \in X$, function *build* folds the list $\{s_l, s_l + 1, ..., s_u\}$, with every value $j$, it checks if it equals to $x$ and appends the singleton set $\{x\}$ to the $j$-th cell. Not that here the cell is indexed from $s_l$, but not 0. If the cell corresponding to $j - x$ is not empty, the candidate solutions stored in that place are also duplicated and add element $x$ is added to every solution.

$$build(T, x) = fold(f, T, \{s_l, s_l + 1, ..., s_u\}) \tag{14.107}$$

$$f(T, j) = \left\{ \begin{array}{lcl} T'[j] \cup \{\{x\} \cup Y | Y \in T[j']\} & : & s_l \leq j' \leq s_u \land T[j'] \neq \phi, j' = j - x \\ T' & : & otherwise \end{array} \right. \tag{14.108}$$

Here the adjustment is applied on $T'$, which is another adjustment to $T$ as shown as below.

$$T' = \left\{ \begin{array}{lcl} \{x\} \cup T[j] & : & x = j \\ T & : & otherwise \end{array} \right. \tag{14.109}$$

Note that the first clause in both equation (14.108) and (14.109) return a new table with certain cell being updated with the given value.

The following Haskell example program implements this algorithm.

```
subsetsum xs s = foldl build (fromList [[] | _ ← [l..u]]) xs `idx` s where
  l = sum $ filter (< 0) xs
  u = sum $ filter (> 0) xs
  idx t i = index t (i - l)
  build tab x = foldl (λt j → let j' = j - x in
                  adjustIf (l ≤ j' && j' ≤ u && tab `idx` j' /= [])
                           (++ [(x:ys) | ys ← tab `idx` j']) j
                           (adjustIf (x == j) ([x]:) j t)) tab [l..u]
  adjustIf pred f i seq = if pred then adjust f (i - l) seq else seq
```

Some materials like[16] provide common structures to abstract dynamic programming. So that problems can be solved with a generic solution by customizing the precondition, the comparison of candidate solutions for better choice, and the merge method for sub solutions. However, the variety of problems makes things complex in practice. It's important to study the properties of the problem carefully.

[13] Again, here we skip the error handling to the case that $s < s_l$ or $s > s_u$. There is no solution if $s$ is out of range.

## Exercise 14.3

- Realize a maze solver by using the stack approach, which can find all the possible paths.
- There are 92 distinct solutions for the 8 queens puzzle. For any one solution, rotating it $90^\circ, 180^\circ, 270^\circ$ gives solutions too. Also flipping it vertically and horizontally also generate solutions. Some solutions are symmetric, so that rotation or flip gives the same one. There are 12 unique solutions in this sense. Modify the program to find the 12 unique solutions. Improve the program, so that the 92 distinct solutions can be found with fewer search.
- Make the 8 queens puzzle solution generic so that it can solve $n$ queens puzzle.
- Make the functional solution to the leap frogs puzzle generic, so that it can solve $n$ frogs case.
- Modify the wolf, goat, and cabbage puzzle algorithm, so that it can find all possible solutions.
- Give the complete algorithm definition to solve the 2 water jugs puzzle with extended Euclid algorithm.
- We needn't the exact linear combination information $x$ and $y$ in fact. After we know the puzzle is solvable by testing with GCD, we can blindly execute the process that: fill $A$, pour $A$ into $B$, whenever $B$ is full, empty it till there is expected volume in one jug. Realize this solution. Can this one find faster solution than the original version?
- Compare to the extended Euclid method, the BFS approach is a kind of brute-force searching. Improve the extended Euclid approach by finding the best linear combination which minimize $|x| + |y|$.
- John Horton Conway introduced the sliding tile puzzle. Figure 14.51 shows a simplified verson. There are 8 cells, 7 of them are occupied by pieces labeled from 1 to 7. Each piece can slide to the free cell if they are connected. The line between cells means there is a connectoin. The goal is to reverse the pieces from 1, 2, 3, 4, 5, 6, 7 to 7, 6, 5, 4, 3, 2, 1 by sliding. Develop a program to solve this puzzle.

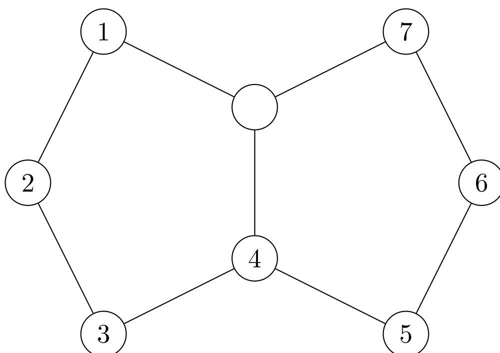



Figure 14.51: Conway sliding puzzle

- Realize the imperative Huffman code table generating algorithm.

- One option to realize the bottom-up solution for the longest common subsequence problem is to record the direction in the table. Thus, instead of storing the length information, three values like 'N', for north, 'W' for west, and 'NW' for northwest are used to indicate how to construct the final result. We start from the bottom-right corner of the table, if the cell value is 'NW', we go along the diagonal by moving to the cell in the upper-left; if it's 'N', we move vertically to the upper row; and move horizontally if it's 'W'. Implement this approach in your favorite programming language.

- Given a list of non-negative integers, find the maximum sum composed by numbers that none of them are adjacent.

- Levenshtein edit distance is defined as the cost of converting from one string $s$ to another string $t$. It is widely used in spell-checking, OCR correction etc. There are three operations allowed in Levenshtein edit distance. Insert a character; delete a character; and substitute a character. Each operation mutate one character a time. The following exaple shows how to convert string "kitten" to "sitting". The Levenshtein edit distance is 3 in this case.

  1. $\underline{\text{k}}$itten → $\underline{\text{s}}$itten (substitution of 's' for 'k');
  2. sitt$\underline{\text{e}}$n → sitt$\underline{\text{i}}$n (substitution of 'i' for 'e');
  3. sitten → sittin$\underline{\text{g}}$ (insertion of 'g' at the end).

  Develop a program to calculate Levenshtein edit distance for two strings with Dynamic Programming.

## 14.4 Short summary

This chapter introduces the elementary methods about searching. Some of them instruct the computer to scan for interesting information among the data. They often have some structure, that can be updated during the scan. This can be considered as a special case for the information reusing approach. The other commonly used strategy is divide and conquer, that the scale of the search domain is kept decreasing till some obvious result. This chapter also explains methods to search for solutions among domains. The solutions typically are not the elements being searched. They can be a series of decisions or some operation arrangement. If there are multiple solutions, sometimes, people want to find the optimized one. For some spacial cases, there exist simplified approach such as the greedy methods. And dynamic programming can be used for more wide range of problems when they shows optimal substructures.

# Bibliography


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

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

[3] M. Blum, R.W. Floyd, V. Pratt, R. Rivest and R. Tarjan, ”Time bounds for selection,” J. Comput. System Sci. 7 (1973) 448-461.

[4] Jon Bentley. “Programming pearls, Second Edition”. Addison-Wesley Professional; 1999. ISBN-13: 978-0201657883

[5] Richard Bird. “Pearls of functional algorithm design”. Chapter 3. Cambridge University Press. 2010. ISBN, 1139490605, 9781139490603

[6] Edsger W. Dijkstra. “The saddleback search”. EWD-934. 1985. http://www.cs.utexas.edu/users/EWD/index09xx.html.

[7] Robert Boyer, and Strother Moore. “MJRTY - A Fast Majority Vote Algorithm”. Automated Reasoning: Essays in Honor of Woody Bledsoe, Automated Reasoning Series, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991, pp. 105-117.

[8] Cormode, Graham; S. Muthukrishnan (2004). “An Improved Data Stream Summary: The Count-Min Sketch and its Applications”. J. Algorithms 55: 29¨C38.

[9] Knuth Donald, Morris James H., jr, Pratt Vaughan. “Fast pattern matching in strings”. SIAM Journal on Computing 6 (2): 323¨C350. 1977.

[10] Robert Boyer, Strother Moore. “A Fast String Searching Algorithm”. Comm. ACM (New York, NY, USA: Association for Computing Machinery) 20 (10): 762¨C772. 1977

[11] R. N. Horspool. “Practical fast searching in strings”. Software - Practice & Experience 10 (6): 501¨C506. 1980.

[12] Wikipedia. “Boyer-Moore string search algorithm”. https://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm

[13] Wikipedia. “Eight queens puzzle”. https://en.wikipedia.org/wiki/Eight_queens_puzzle

[14] George Pólya. “How to solve it: A new aspect of mathematical method”. Princeton University Press(April 25, 2004). ISBN-13: 978-0691119663

[15] Wikipedia. “David A. Huffman”. https://en.wikipedia.org/wiki/David_A._Huffman

[16] Fethi Rabhi, Guy Lapalme "Algorithms: a functional programming approach". Second edition. Addison-Wesley.

# 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.

```
 1: function Delete(T, x)
 2:     p ← Parent(x)
 3:     q ← NIL
 4:     if Left(x) = NIL then
 5:         q ← Right(x)
 6:         Replace(x, Right(x))                    ▷ replace x with its right sub-tree
 7:     else if Right(x) = NIL then
 8:         q ← Left(x)
 9:         Replace(x, Leftx())                      ▷ replace x with its left sub-tree
10:     else
11:         y ← Min(Right(x))
12:         p ← Parent(y)
13:         q ← Right(y)
14:         Key(x) ← Key(y)
15:         copy data from y to x
16:         Replace(y, Right(y))                    ▷ replace y with its right sub-tree
17:         x ← y
18:     if Color(x) = BLACK then
19:         T ← Delete-Fix(T, Make-Black(p, q), q = NIL?)
20:     release x
21:     return T
```

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.

```
1: function MAKE-BLACK(p, q)
2:     if p = NIL and q = NIL then
3:         return NIL                              ▷ The tree was singleton
4:     else if q = NIL then
5:         n ← Doubly Black NIL
6:         PARENT(n) ← p
7:         return n
8:     else
9:         return BLACKEN(q)
```

If both $p$ and $q$ 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.

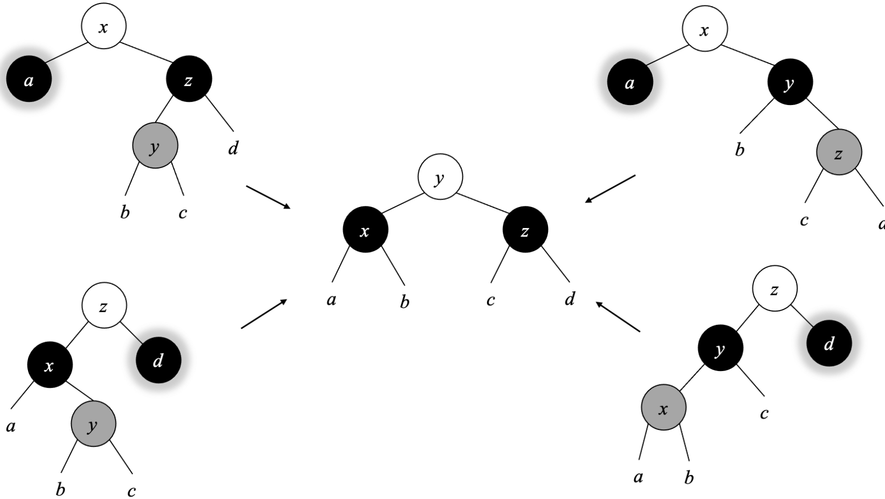



Figure A.1: The doubly black node has a black sibling, and a red nephew. It can be fixed with a rotation.

```
1: function DELETE-FIX(T, x, f)
2:     n ← NIL
3:     if f = True then                            ▷ x is doubly black NIL
4:         n ← x
5:     if x = NIL then                             ▷ Delete the singleton leaf
6:         return NIL
7:     while x ≠ T and COLOR(x) = B² do     ▷ x is doubly black, but not the root
8:         if SIBLING(x) ≠ NIL then                ▷ The sibling is not empty
9:             s ← SIBLING(x)
```

```
10:         ...
11:         if s is black and LEFT(s) is red then
12:             if x = LEFT(PARENT(x)) then                       ▷ x is the left
13:                 set x, PARENT(x), and LEFT(s) all black
14:                 T ← ROTATE-RIGHT(T, s)
15:                 T ← ROTATE-LEFT(T, PARENT(x))
16:             else                                              ▷ x is the right
17:                 set x, PARENT(x), s, and LEFT(s) all black
18:                 T ← ROTATE-RIGHT(T, PARENT(x))
19:         else if s is black and RIGHT(s) is red then
20:             if x = LEFT(PARENT(x)) then                       ▷ x is the left
21:                 set x, PARENT(x), s, and RIGHT(s) all black
22:                 T ← ROTATE-LEFT(T, PARENT(x))
23:             else                                              ▷ x is the right
24:                 set x, PARENT(x), and RIGHT(s) all black
25:                 T ← ROTATE-LEFT(T, s)
26:                 T ← ROTATE-RIGHT(T, PARENT(x))
27:             ...
```

**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.

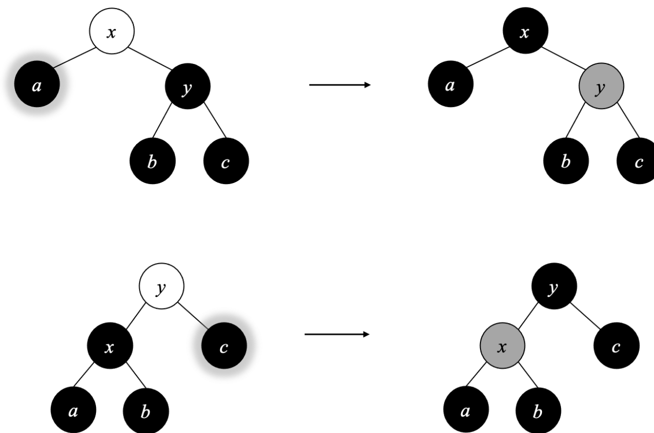



Figure A.2: The sibling of the doubly black is red

```
 1: function DELETE-FIX(T, x, f)
 2:     n ← NIL
 3:     if f = True then                                  ▷ x is doubly black NIL
 4:         n ← x
 5:     if x = NIL then                                   ▷ Delete the singleton leaf
 6:         return NIL
 7:     while x ≠ T and COLOR(x) = B² do
 8:         if SIBLING(x) ≠ NIL then
 9:             s ← SIBLING(x)
10:             if s is red then                          ▷ The sibling is red
11:                 set PARENT(x) red
12:                 set s black
13:                 if x = LEFT(PARENT(x)) then           ▷ x is the left
```

14: $T \leftarrow$ ROTATE-LEFT$T$, PARENT$(x)$
15: **else** ▷ $x$ is the right
16: $T \leftarrow$ ROTATE-RIGHT$T$, PARENT$(x)$
17: **else if** $s$ is black and LEFT$(s)$ is red **then**
18: ...

**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.

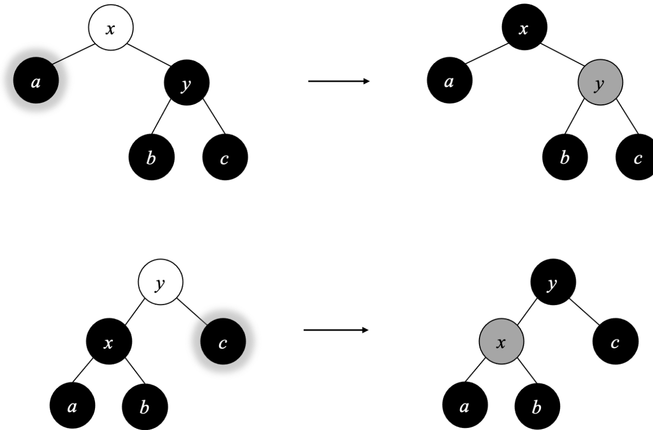



Figure A.3: move the blackness up

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.

1: **function** DELETE-FIX$(T, x, f)$
2: $n \leftarrow$ NIL
3: **if** $f =$ True **then** ▷ $x$ is a doubly black NIL
4: $n \leftarrow x$
5: **if** $x =$ NIL **then** ▷ Delete the singleton leaf
6: **return** NIL
7: **while** $x \neq T$ and COLOR$(x) = \mathcal{B}^2$ **do**
8: **if** SIBLING$(x) \neq$ NIL **then** ▷ The sibling is not empty
9: $s \leftarrow$ SIBLING$(x)$
10: **if** $s$ is red **then** ▷ The sibling is red
11: set PARENT$(x)$ red
12: set $s$ black
13: **if** $x =$ LEFT(PARENT$(x)$) **then** ▷ $x$ is the left
14: $T \leftarrow$ ROTATE-LEFT$T$, PARENT$(x)$
15: **else** ▷ $x$ is the right
16: $T \leftarrow$ ROTATE-RIGHT$T$, PARENT$(x)$
17: **else if** $s$ is black and LEFT$(s)$ is red **then**
18: **if** $x =$ LEFT(PARENT$(x)$) **then** ▷ $x$ is the left
19: set $x$, PARENT$(x)$, and LEFT$(s)$ all black
20: $T \leftarrow$ ROTATE-RIGHT$(T, s)$

```
21:                 T ← Rotate-Left(T, Parent(x))
22:             else                                              ▷ x is the right
23:                 set x, Parent(x), s, and Left(s) all black
24:                 T ← Rotate-Right(T, Parent(x))
25:         else if s is black and Right(s) is red then
26:             if x = Left(Parent(x)) then                        ▷ x is the left
27:                 set x, Parent(x), s, and Right(s) all black
28:                 T ← Rotate-Left(T, Parent(x))
29:             else                                              ▷ x is the right
30:                 set x, Parent(x), and Right(s) all black
31:                 T ← Rotate-Left(T, s)
32:                 T ← Rotate-Right(T, Parent(x))
33:         else if s, Left(s), and Right(s) are all black then
34:             set x black
35:             set s red
36:             Blacken(Parent(x))
37:             x ← Parent(x)
38:     else                                              ▷ move the blackness up
39:         set x black
40:         Blacken(Parent(x))
41:         x ← Parent(x)
42:     set T black
43:     if n ≠ NIL then
44:         replace n with NIL
45:     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

## B.1 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 $\Delta r$.

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, $\delta$ and $\delta'$ 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 $\Delta l$.

□

## B.2 Balance adjustment after insert

The balance factors are $\pm 2$ 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.

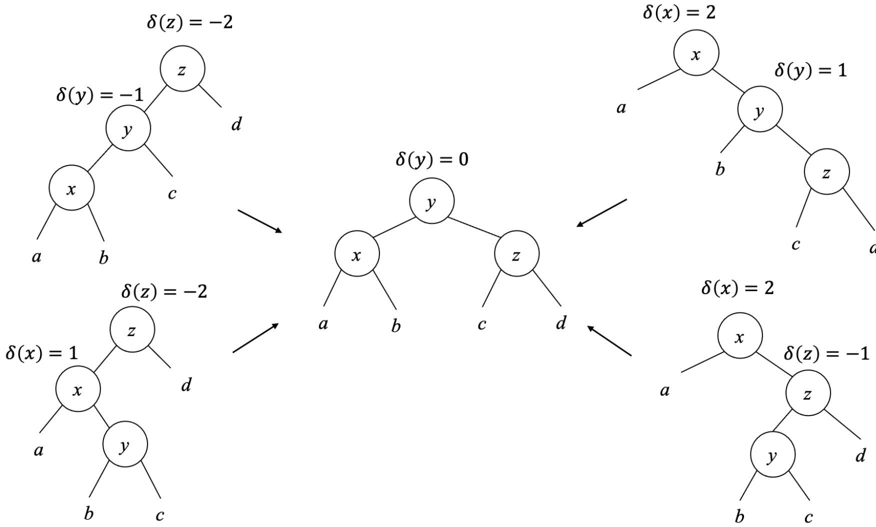



Figure B.1: Fix 4 cases to the same structure

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:

$$\begin{array}{lcl} \delta(y) = |c| - |x| = -1 & \Rightarrow & |c| = |x| - 1 \\ \delta(z) = |d| - |y| = -2 & \Rightarrow & |d| = |y| - 2 \end{array} \tag{B.2}$$

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).

□

## B.3 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.

### B.3.1 Functional delete

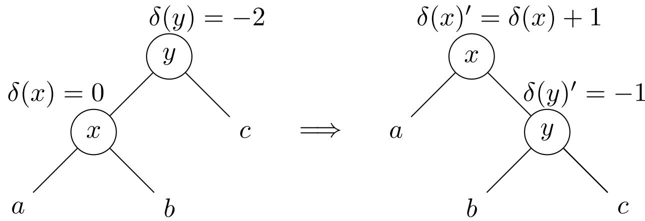
When delete, we re-use the binary search tree *delete* in the first step, then check the balance factors and perform fixing. The result is a pair $(T', \Delta H)$, where $T'$ is the new tree and $\Delta H$ is the height decrement. We define *delete* as below:

$$delete = fst \circ del \tag{B.18}$$

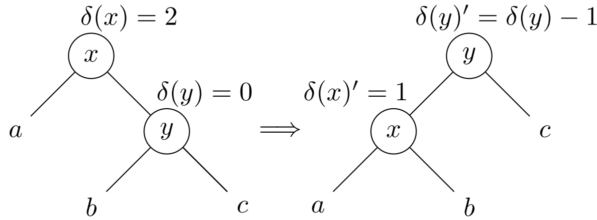
where $del(T, k)$ does the actual work to delete element $k$ from $T$:

$$\begin{array}{rcl}
del\ \varnothing\ k & = & (\varnothing, 0) \\
del\ (l, k', r, \delta) & = & \begin{cases}
k < k' : & tree\ (del\ l\ k)\ k'\ (r, 0)\ \delta \\
k > k' : & tree\ (l, 0)\ k'\ (del\ r\ k)\ \delta \\
k = k' : & \begin{cases}
l = \varnothing : & (r, -1) \\
r = \varnothing : & (l, -1) \\
else : & tree\ (l, 0)\ k''\ (del\ r\ k'')\ \delta \\
 & \text{where } k'' = min(r)
\end{cases}
\end{cases}
\end{array} \tag{B.19}$$

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 $\Delta H$ 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)
```

### B.3.2 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.

1: **function** DELETE($T, x$)
2:     **if** $x$ = NIL **then**
3:         **return** $T$
4:     $p \leftarrow$ PARENT($x$)
5:     **if** LEFT($x$) = NIL **then**
6:         $y \leftarrow$ RIGHT($x$)
7:         replace $x$ with $y$
8:     **else if** RIGHT($x$) = NIL **then**
9:         $y \leftarrow$ LEFT($x$)
10:         replace $x$ with $y$
11:     **else**
12:         $z \leftarrow$ MIN(RIGHT($x$))
13:         copy data from $z$ to $x$
14:         $p \leftarrow$ PARENT($z$)
15:         $y \leftarrow$ RIGHT($z$)

```
16:         replace z with y
17:     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.

```
1: function AVL-Delete-Fix(T, p, x)
2:     while p ≠ NIL do
3:         l ← Left(p), r ← Right(p)
4:         δ ← δ(p), δ' ← δ
5:         if x = l then
6:             δ' ← δ' + 1
7:         else
8:             δ' ← δ' − 1
9:         if p is leaf then                                ▷ l = r = NIL
10:            δ' ← 0
11:        if |δ| = 1 ∧ |δ'| = 0 then
12:            x ← p
13:            p ← Parent(x)
14:        else if |δ| = 0 ∧ |δ'| = 1 then
15:            return T
16:        else if |δ| = 1 ∧ |δ'| = 2 then
17:            if δ' = 2 then
18:                if δ(r) = 1 then                         ▷ Right-right
19:                    δ(p) ← 0
20:                    δ(r) ← 0
21:                    p ← r
22:                    T ← Left-Rotate(T, p)
23:                else if δ(r) = −1 then                   ▷ Right-left
24:                    δ_y ← δ( Left(r) )
25:                    if δ_y = 1 then
26:                        δ(p) ← −1
27:                    else
28:                        δ(p) ← 0
29:                    δ( Left(r) ) ← 0
30:                    if δ_y = −1 then
```

31: $\delta(r) \leftarrow 1$
32: **else**
33: $\delta(r) \leftarrow 0$
34: **else** ▷ Delete specific right-right
35: $\delta(p) \leftarrow 1$
36: $\delta(r) \leftarrow \delta(r) - 1$
37: $T \leftarrow$ LEFT-ROTATE$(T, p)$
38: break ▷ No furthur height change
39: **else if** $\delta' = -2$ **then**
40: **if** $\delta(l) = -1$ **then** ▷ Left-left
41: $\delta(p) \leftarrow 0$
42: $\delta(l) \leftarrow 0$
43: $p \leftarrow l$
44: $T \leftarrow$ RIGHT-ROTATE$(T, p)$
45: **else if** $\delta(l) = 1$ **then** ▷ Left-right
46: $\delta_y \leftarrow \delta($ RIGHT$(l)$ $)$
47: **if** $\delta_y = -1$ **then**
48: $\delta(p) \leftarrow 1$
49: **else**
50: $\delta(p) \leftarrow 0$
51: $\delta($ RIGHT$(l)$ $) \leftarrow 0$
52: **if** $\delta_y = 1$ **then**
53: $\delta(l) \leftarrow -1$
54: **else**
55: $\delta(l) \leftarrow 0$
56: **else** ▷ Delete specific left-left
57: $\delta(p) \leftarrow -1$
58: $\delta(l) \leftarrow \delta(l) + 1$
59: $T \leftarrow$ RIGHT-ROTATE$(T, p)$
60: break ▷ No furthur height change
▷ Height decreases, go on bottom-up updating
61: $x \leftarrow p$
62: $p \leftarrow$ PARENT$(x)$
63: **if** $p =$ NIL **then** ▷ Delete the root
64: **return** $x$
65: **return** $T$

### 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*.

## B.4 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
}
```

# 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

Version 1.3, 3 November 2008
Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

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, 311

Auto completion, 115
AVL tree, 89
- balance, 92
- definition, 89
- imperative insert, 94
- insert, 91
- verification, 93

B-tree, 125
- delete, 136
- insert, 127
- look up, 134

BFS, 336
Binary heap, 147
- build, 149
- decrease key, 152
- Heapify, 148
- insertion, 152
- pop, 150
- push, 152
- top, 150
- top-k, 150

binary heap by array, 147
Binary Random Access List
- insert, 214
- random access, 216
- remove, 214

Binary search, 265
binary search tree, 53
- data layout, 55
- delete, 61
- insertion, 55
- looking up, 59
- min/max, 60
- random build, 64
- search, 59
- succ/pred, 60
- traverse, 57

binary tree, 53
Binomial Heap
- Link, 182

Binomial heap, 179
- definition, 180
- insert, 183
- pop, 185
- push, 183

Binomial tree, 180
- merge, 184

Boyer-Moor majority number, 280
Boyer-Moore algorithm, 296
Breadth-first search, 336

Change-making problem, 347
Cock-tail sort, 170
complete binary tree, 147
Curried Form, 29
Currying, 29

Deep-first search, 305
DFS, 305
Dynamic programming, 348

equivalent, 41

Fibonacci Heap
- decrease key, 192
- delete min, 188
- insert, 187
- merge, 187
- pop, 188

Fibonacci heap, 186
Finger tree
- Append to right, 225
- Concatenate, 225
- insert to left, 223
- Random access, 226
- Remove from left, 223
- Remove from right, 225

fold, 42

Grady algorithm, 338

Heap sort, 152
Huffman coding, 338

in-order traverse, 57

Insertion sort
- binary search, 69
- binary search tree, 70
- linked-list setting, 69

insertion sort, 67
- insertion, 68

Integer Patricia, 102
Integer prefix tree, 102
Integer tree
- insert, 103
- lookup, 107

Integer trie, 99
- insert, 100
- look up, 102

Kloski puzzle, 330
KMP, 285
Knuth-Morris-Pratt algorithm, 285

LCS, 353
left child, right sibling, 182
Leftist heap, 154
- heap sort, 156
- insert, 156
- merge, 155
- pop, 155
- rank, 154
- S-value, 154
- top, 155

List
- append, 23
- break, 39
- concat, 27
- concats, 46
- cons, 20
- Construction, 20
- definition, 19
- delete, 26
- delete at, 26
- drop, 38
- drop while, 39
- elem, 46
- empty, 20
- empty testing, 20
- existence testing, 46
- Extract sub-list, 38
- filter, 47
- find, 47
- fold from left, 44
- fold from right, 42
- foldl, 44
- foldr, 42
- for each, 35
- get at, 20
- group, 40
- head, 20
- index, 20
- infix, 48
- init, 21
- insert, 25
- insert at, 25
- last, 21
- length, 20
- lookup, 47
- map, 33, 34
- matching, 48
- maximum, 31
- minimum, 31
- mutate, 23
- prefix, 48
- product, 28
- reverse, 37
- Reverse index, 22
- rindex, 22
- set at, 24
- span, 39
- split at, 38
- suffix, 48
- sum, 28
- tail, 20
- take, 38
- take while, 39
- Transform, 32
- unzip, 49
- zip, 49

Longest common subsequence problem, 353

Maximum sum problem, 284
Maze problem, 305
Merge Sort, 246
- Bottom-up merge sort, 256
- In-place merge sort, 250
- Merge, 247
- Nature merge sort, 253
- Performance, 248
- Work area, 249, 250

minimum free number, 10
MTF, 227

Paired-array sequence
- random access, 219
- remove and balance, 219

Pairing heap, 194
- decrease key, 195

- definition, 195
- delete, 197
- insert, 195
- pop, 196
- top, 195

Parallel merge sort, 257
Parallel quick sort, 257
Patricia, 111
Peg puzzle, 313
post-order traverse, 57
pre-order traverse, 57
Prefix tree, 111
- insert, 111
- look up, 114

Queue
- Balance Queue, 207
- Circular buffer, 204
- Lazy real-time queue, 210
- linked-list, 203
- Paired-array queue, 206
- Paired-list queue, 206
- Real-time queue, 207

Quick Sort
- 2-way partition, 240
- 3-way partition, 241
- Average case, 237

Quick sort, 233
- Improvement, 239
- partition, 234
- Performance, 237
- Ternary partition, 239

Radix tree, 99
range traverse, 61
Red-black tree
- Imperative delete, 367

red-black tree, 73, 76
- delete, 80
- imperative insertion, 84
- insert, 78
- red-black properties, 76

reduce, 44

Saddelback search, 269
Selection algorithm, 262
selection sort, 167
- min, 168
- tail-recursive min, 168

Sequence
- Binary random access list, 213
- Concatenate-able list, 220
- finger tree, 221
- numeric representation, 216
- Paired-array sequence, 219

Skew heap, 156
- insertion, 157
- merge, 157
- pop, 157
- top, 157

Splay heap, 157
- insert, 161
- merge, 162
- pop, 161
- splay, 158
- top, 161

strict weak order, 169
Subset sum problem, 358

T9, 117
Tail call, 29
Tail recursion, 29
Tail recursive call, 29
The wolf, goat, and cabbage puzzle, 319
Tournament knock out, 172
tree reconstruction, 58
tree rotation, 74
Trie, 108
- insert, 108
- look up, 110

Water jugs puzzle, 323
word counter, 53