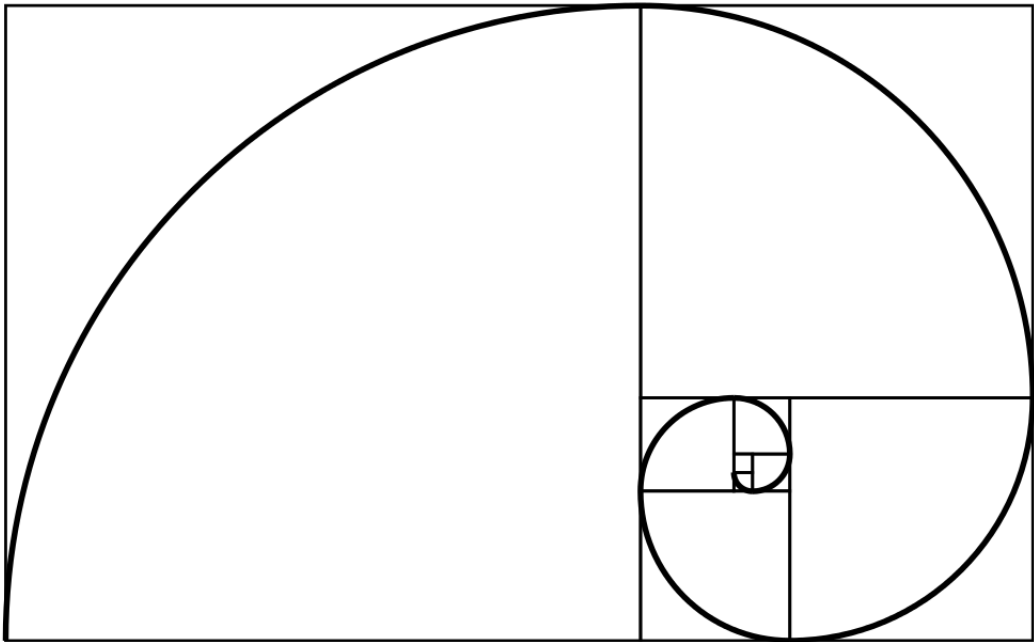# Elementary Functional Algorithms

Xinyu LIU [1]

October 31, 2023

[1]**Xinyu LIU**

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

https://github.com/liuxinyu95

Preface to the second edition

Some people did not see their works published, like Fermat and Galois; Some people refused to publish any 'imperfect' works despite they had already been great and important (at least from our view point), like Gauss; Some people lost their works in fires or wars, like Riemann. We owe Samuel Fermat, who collected his father's notes and mails, published a special edition of *Arithmetica, augmented with Fermat's comments*; We owe Liouville, who re-discovered and published Galois' papers in *Journal de Mathématiques Pures et Appliquées*; We owe Dedekind, who collected the burnt notebook from Mrs Riemann after her husband's early death. We appreciate those publishers who save, protect, and spread human knowledge no matter it's profitable or not (in terms of money). The development of technology and internet greatly change the way that people write, log, edit, and publish things. I was able to write and share about elementary functional algorithms and data structures online from 2009. It was accumulated to a draft book by 2015, and was published in 2017 in Chinese.

I always think I am so lucky compare to the people in the past. It's impossible to make this happen before, while there are plenty of wonderful works in this domain yet to be published. Thanks to the editors, the publisher, and every reader. I collected the feedback, questions, and comments, started to re-write the book from the end of 2020, and completed a new edition by May 2023. The main changes are:

1. Rewrite all chapters, use simple notations instead of long text descriptions.

2. Reorganize the book structure. (a) Move the list to the first chapter, make it friendly to the new readers to the functional programming; (b) Add a section of paired-list B-tree in chapter 7. Add the remove algorithms to the red-black tree and AVL tree to the appendix; (c) Remove the chapter of the suffix tree. For imperative string matching, focus on KMP algorithm, and remove the Boyer-Moore algorithm; (d) Add the answers to all the 120 exercises.

3. Unify the programming language of the example programs. This is one of the most feedback to the first edition. Use Haskell for functional examples, and consolidate imperative ones in a language named 'Bourbaki' (Nicolas Bourbaki is the pseudonym of a group of (mainly) French mathematicians, including André Weil, Henri Cartan, and etc. The virtual figure indicates the programming language does not exist, while it reflects some popular languages behind, like Python, Java, Kotlin and etc.). Collect the examples as an appendix after each chapter.

4. Correct all known errors in the first edition.

Chris Okasaki and Richard Bird established the main content of purely functional data structures and algorithms. I essentially tailored and re-organized the materials under their framework. Although the new functional applications and technologies evolve fast, the

elementary algorithms are relative stable. This slow changing part allows me to spend over 10 years, iteratively write and improve this book[1].

Oct 2023, Beijing, China

---

[1]The pdf can be downloaded from github.

Preface to the first edition

Programmers learn elementary algorithms at school. Except for programming contest or code interview, they seldom develop algorithms at work. Most time, the needed algorithms or data structures are provided in libraries. We needn't bother to 're-invent the wheel'. When talking about algorithms in AI and machine learning, people actually mean scientific modeling. Elementary algorithms are fundamental things, Let's start with two puzzles.

The smallest free number

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

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

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

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

Where the $\notin$ is realized as below.

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

Where $|X| = n$ is the length of X. Some environments have built-in implementation to test existence of an element. However, this solution performs poor with millions of numbers. The time spent is quadratic to n. In a computer with 2 cores of 2.10 GHz CPU, and 2G RAM, the C implementation takes 5.4s to find the answer among 100K numbers, and exceeds 8 minutes to handle 1 million numbers.

Improvement

For n numbers $x_1, x_2, ..., x_n$, if there is a free number, some x_i must be out of the range $[0, n)$[2]; otherwise the list is exactly some permutation of $0, 1, ..., n - 1$ hence n is the minimum free number.

$$minfree(x_1, x_2, ..., x_n) \leq n \qquad (1)$$

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

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

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

Divide and Conquer

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

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

where:

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

This algorithm needn't additional space[3]. Each recursive call performs $O(|A|)$ comparisons to partition A' and A'', hence halves the problem as $T(n) = T(n/2) + O(n)$. We can reduce it to $O(n)$ according to the master theorem[4]. Below example program implements this algorithm.

---

[2]Range $[a, b)$ starts from a to b, but excludes b.

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

[4]Alternatively, the first call takes $O(n)$ time to partition A' and A'', the second call takes $O(n/2)$ time, the third call takes $O(n/4)$ time ... The total time is $O(n + n/2 + n/4 + ...) = O(2n) = O(n)$.

```
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 = [x | x ← xs, x ≤ m]
       bs = [x | x ← xs, x > m]
```

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

1: **function** MIN-FREE(A)
2:     $l \leftarrow 0, u \leftarrow |A|$
3:     **while** $u - l > 0$ **do**
4:         $m \leftarrow \lfloor \dfrac{u + l}{2} \rfloor$
5:         $left \leftarrow l$
6:         **for** $right \leftarrow l$ to $u - 1$ **do**
7:             **if** $A[right] \leq m$ **then**
8:                 Exchange $A[left] \leftrightarrow A[right]$
9:                 $left \leftarrow left + 1$
10:         **if** $left < m + 1$ **then**
11:             $u \leftarrow left$
12:         **else**
13:             $l \leftarrow left$

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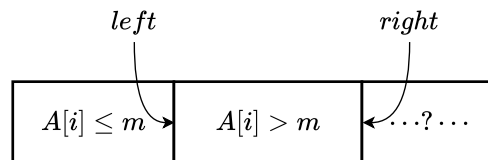

Figure 1: All $A[i] \leq m$ where $0 \leq i < left$; while $A[i] > m$ where $left \leq i < right$. The rest are yet to be scanned.

Regular number

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

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

The brute-force solution

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

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

---

[5]Also known as 5-smooth numbers in number theory, and Hamming numbers named after Richard Hamming.

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

Improvement

Modular and divide are expensive operations[2]. Instead of checking every number, we can generate regular numbers from 1 in ascending order. We use a queue, which allows to add number to one end (enqueue), and remove from the other end (dequeue). The number enqueued first will be dequeued first (First In First Out). Initialize the queue with 1, the 0th regular number. We repeatedly dequeue a number, multiply it by 2, 3, 5 to generate 3 new numbers; then add them to the queue in ascending order. We drop the number if it is already in the queue, as shown in fig. 2.
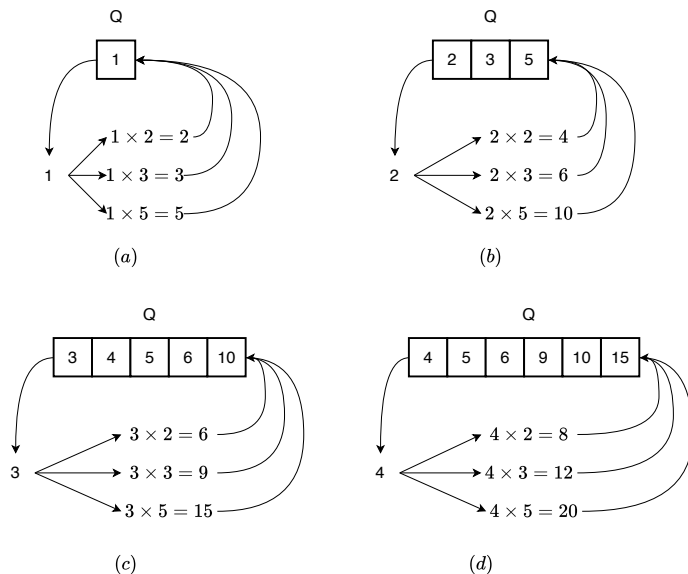

Figure 2: First 4 steps.

```
1:  function REGULAR-NUMBER(n)
2:      Q ← [1]
```

3:      **while** $n > 0$ **do**
4:          $x \leftarrow \text{DEQUEUE}(Q)$
5:          $\text{UNIQUE-ENQUEUE}(Q, 2x)$
6:          $\text{UNIQUE-ENQUEUE}(Q, 3x)$
7:          $\text{UNIQUE-ENQUEUE}(Q, 5x)$
8:          $n \leftarrow n - 1$
9:      **return** x

10: **function** $\text{UNIQUE-ENQUEUE}(Q, x)$
11:     $i \leftarrow 0, m \leftarrow |Q|$
12:     **while** $i < m$ and $Q[i] < x$ **do**
13:        $i \leftarrow i + 1$
14:     **if** $i \geq m$ or $x \neq Q[i]$ **then**
15:        $\text{INSERT}(Q, i, x)$

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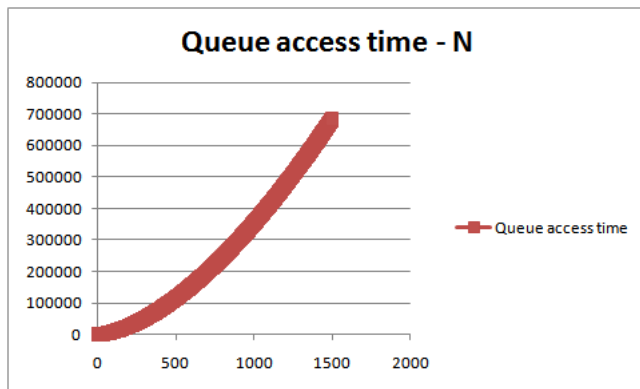

Figure 3: Queue access count - n.

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

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

Where $x{:}xs$ links x before list xs. It is called 'cons' in Lisp. 1 is linked as the head (the 0th regular number). $\cup$ merges two lists:

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

Below is the example program:

```
xs = 1 : [2*x | x ← xs] `merge` [3*x | x ← xs] `merge` [5*x | x ← xs]

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

This example program gives the 1500th number 860934420 by `xs !! 1500` in 0.03s in the same computer.

Queues

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

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

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

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

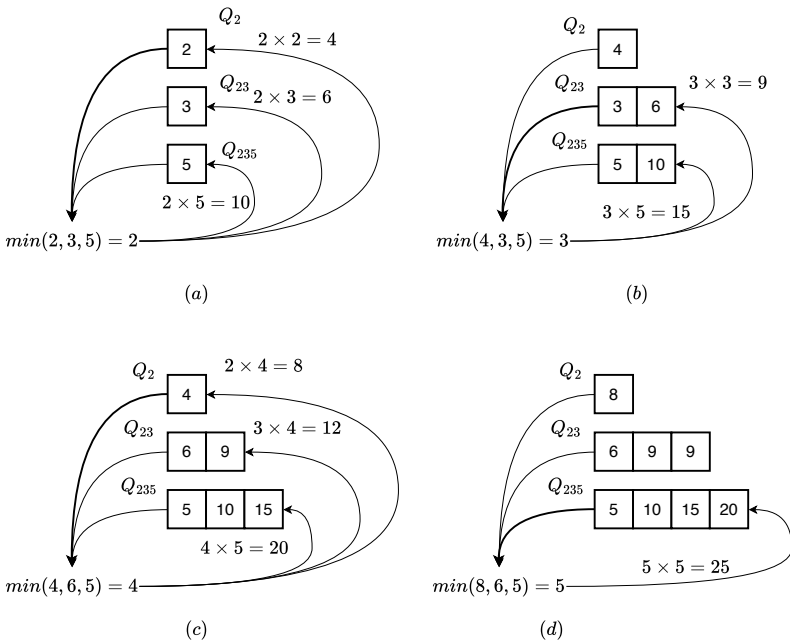We reach to the answer after dequeue n numbers. Figure 4 gives the first 4 steps.

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

1: **function** REGULAR-NUMBER(n)
2:     $x \leftarrow 1$
3:     $Q_2 \leftarrow \{2\}$, $Q_{23} \leftarrow \{3\}$, $Q_{235} \leftarrow \{5\}$

```
4:      while n > 0 do
5:          x ← min(HEAD(Q₂), HEAD(Q₂₃), HEAD(Q₂₃₅))
6:          if x = HEAD(Q₂) then
7:              DEQUEUE(Q₂)
8:              ENQUEUE(Q₂, 2x)
9:              ENQUEUE(Q₂₃, 3x)
10:             ENQUEUE(Q₂₃₅, 5x)
11:         else if x = HEAD(Q₂₃) then
12:             DEQUEUE(Q₂₃)
13:             ENQUEUE(Q₂₃, 3x)
14:             ENQUEUE(Q₂₃₅, 5x)
15:         else
16:             DEQUEUE(Q₂₃₅)
17:             ENQUEUE(Q₂₃₅, 5x)
18:         n ← n - 1
19:     return x
```

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

Summary

Both brute-force solutions can't scale up. This book is *not* about coding contest or code interview, but aims to provide both purely functional algorithms and their counterpart imperative implementations. We referenced many results from Okasaki's work[3] and classic text books[4]. We avoid relying on a specific programming language, because the reader may or may not be familiar with it, and programming languages keep changing. Instead, we use pseudo code or mathematics notation to make the algorithm definition generic. When give code examples, the functional ones look more like Haskell, and the imperative ones look like a mix of several languages.

Exercise 1

1.1. For the free number puzzle, since all numbers are not negative, we can reuse the sign as a flag. For every $|x| < n$ (where n is the length), negate the number at position $|x|$. Then scan to find the first positive number. Its position is the answer. Write a program to realize this solution.

1.2. There are n numbers 1, 2, ..., n. After some processing, they are shuffled, and a number x is altered to y. Suppose $1 \le y \le n$, design a solution to find x and y in linear time with constant space.

1.3. Below example program is a solution for the regular number puzzle. Is it equivalent to the queue based solution?

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

```
            i = i + 1
            x2 = 2 * nums[i]
        }
        if x3 == nums[n] {
            j = j + 1
            x3 = 3 * nums[j]
        }
        if x5 == nums[n] {
            k = k + 1
            x5 = 5 * nums[k]
        }
    }
    return nums[m]
}
```

Contents

Chapter 1

List

1.1 Introduction

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

1.2 Definition

List, or singly linked-list is a data structure recursively defined as: A *list* is either empty, denoted as [ ] or NIL; or contains an element (also called key) and liked with a sub-*list* (called next). Figure 1.1 shows a list of nodes. Every node links to the next or NIL (the last). We often define list with the compound structure[2], for example:
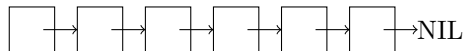
Figure 1.1: A list of nodes

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

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

---

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

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

1.2.1   Access

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

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

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

Exercise 1.2

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

1.3   Basic operations

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

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

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

1: **function** LENGTH(X)
2:     $n \leftarrow 0$
3:     **while** $X \neq$ NIL **do**
4:         $n \leftarrow n + 1$
5:         $X \leftarrow$ NEXT(X)
6:     **return** n

1.3.1   index

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

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

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

---

[3]We often write function $f(x)$ as $f\ x$, and $f(x, y, ..., z)$ as $f\ x\ y\ ...\ z$.
[4]They are named as `car` and `cdr` in Lisp due to the design of machine registers[63].

```
1: function GET-AT(i, X)
2:     while i ≠ 0 do
3:         X ← NEXT(X)                    ▷ error when X = NIL
4:         i ← i − 1
5:     return FIRST(X)
```

Exercise 1.3

1.3.1. For the iterative GET-AT(i, X), what is the behavior when X is empty? what if i is out of bound?

1.3.2 Last

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

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

Neither handles the empty list. The behavior is undefined with [ ]. Below are the iterative implementation:

```
1: function LAST(X)
2:     x ← NIL
3:     while X ≠ NIL do
4:         x ← FIRST(X)
5:         X ← REST(X)
6:     return x

7: function INIT(X)
8:     X' ← NIL
9:     while REST(X) ≠ NIL do            ▷ Error when X is NIL
10:        X' ← CONS(FIRST(X), X')
11:        X ← REST(X)
12:    return REVERSE(X')
```

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

1.3.3 Right index

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

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

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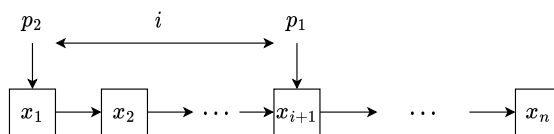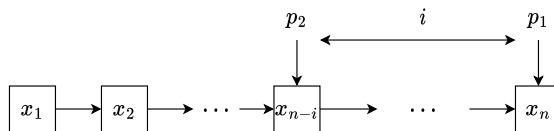
Figure 1.2: Sliding window. (a) p_2 starts from the head, behind p_1 in i steps; (b) When p_1 reaches the tail, p_2 points to the i-th element from right.

```
1: function LAST-AT(i, X)
2:      p ← X
3:      while i > 0 do
4:          X ← REST(X)                                    ▷ Error if out of bound
5:          i ← i − 1
6:      while REST(X) ≠ NIL do
7:          X ← REST(X)
8:          p ← REST(p)
9:      return FIRST(p)
```

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

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

Where:

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

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

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

Exercise 1.4

1.4.1. In the INIT algorithm, can we use APPEND(X', FIRST(X)) instead of CONS?

1.4.2. How to handle empty list or out of bound error in LAST-AT?

1.3.4 Mutate

Mutate includes append, insert, update, and delete. The functional environment actually implements mutate by creating a new list for the changed part, while keeps (persists) the original one for reuse, or release at sometime (chapter 2 in [3]). Append is the symmetric operation of *cons*, it appends element to the tail, while *cons* adds from head. It is also known as 'snoc' (reverse of 'cons'). As it need traverse the list to the tail, the performance is $O(n)$, where n is the length. To avoid repeatedly traverse, we can persist the tail reference, and update it for changes.

$$\begin{aligned} append \ [\ ] \ x &= [x] \\ append \ (y{:}ys) \ x &= y : append \ ys \ x \end{aligned} \qquad (1.8)$$

Below is the corresponding iterative implementation[5]:

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

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

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

Similar to *getAt*, we need advance to the target position i in $O(i)$ time to change the element.

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

Exercise 1.5

1.5.1. Add the 'tail' reference, optimize the *append* to constant time.

1.5.2. When need update the tail reference? How does it affect the performance?

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

insert

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

---

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

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

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

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

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

$$\begin{aligned}
insert\ x\ [\ ] &= [x] \\
insert\ x\ (y{:}ys) &= \begin{cases} x \le y: & x{:}y{:}ys \\ \text{otherwise}: & y : insert\ x\ ys \end{cases}
\end{aligned} \qquad (1.11)$$

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

```
1: function INSERT(x, X)
2:     if X = NIL or x < FIRST(X) then
3:         return CONS(x, X)
4:     H ← X
5:     while REST(X) ≠ NIL and FIRST(REST(X)) < x do
6:         X ← REST(X)
7:     REST(X) ← CONS(x, REST(X))
8:     return H
```

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

$$\begin{aligned}
sort\ [\ ] &= [\ ] \\
sort\ (x{:}xs) &= insert\ x\ (sort\ xs)
\end{aligned} \qquad (1.12)$$

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

```
1: function SORT(X)
2:     S ← NIL
3:     while X ≠ NIL do
4:         S ← INSERT(FIRST(X), S)
5:         X ← REST(X)
6:     return S
```

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

Exercise 1.6

1.6.1. Handle the out-of-bound case when insert, treat it as append.

1.6.2. Implement insert for array. When insert at position i, all elements after i need shift to the end.

delete

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

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

It is bound to $O(i)$ time. Below is the iterative implementation.

1: **function** DEL-AT(i, X)
2:    $S \leftarrow$ CONS$(\perp, X)$ ▷ Sentinel node
3:    $p \leftarrow S$
4:    **while** $i > 0$ and $X \neq$ NIL **do**
5:       $i \leftarrow i - 1$
6:       $p \leftarrow X$
7:       $X \leftarrow$ REST(X)
8:    **if** $X \neq$ NIL **then**
9:       REST$(p) \leftarrow$ REST(X)
10:   **return** REST(S)

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

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

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

1: **function** DELETE(x, X)
2:    $S \leftarrow$ CONS$(\perp, X)$
3:    $p \leftarrow X$
4:    **while** $X \neq$ NIL and FIRST$(X) \neq x$ **do**
5:       $p \leftarrow X$
6:       $X \leftarrow$ REST(X)
7:    **if** $X \neq$ NIL **then**
8:       REST$(p) \leftarrow$ REST(X)
9:    **return** REST(S)

Exercise 1.7

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

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

concatenate

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

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

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

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

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

```
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

Sum and product have the same structure. We will introduce how to abstract them to higher order computation in section 1.6. For empty list, define the sum as 0, the product as 1.

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

Both need traverse the list, hence the performance is $O(n)$. They compute from right to left. We can change to *accumulate* the result from left to right.

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

Given a list, we call sum' with 0, and $prod'$ with 1 to start accumulating:

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

Or in Curried form:

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

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

The accumulated implementation computes from left to right, needn't book keeping any context, state, or intermediate result for recursion. All states are either passed as argument (for example a), or dropped (for example the previous element). We can further optimize such recursive calls to loops. Because the recursion happens at the tail of the function, we call them *tail recursion* (or 'tail call'), and the process to eliminate recursion as 'tail recursion optimization'[61]. It greatly improves the performance and avoid stack overflow due to deep recursions. In eq. (1.12) about insertion sort, the recursive implementation sorts elements form right. We can also optimize it to tail call:

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

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

1: **function** POW(b, n)
2:     $x \leftarrow 1$
3:     **loop** n times
4:         $x \leftarrow x \cdot b$
5:     **return** x

When compute b^8, after the first 2 loops, we get $x = b^2$. At this stage, we needn't multiply x with b to get b^3, but directly compute x^2, which gives b^4. If do this again, we get $(b^4)^2 = b^8$. We only need loop 4 times, but not 8 times. If $n = 2^m$ for some integer $m \geq 0$, we can compute b^n fast as below:

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

We next extend this divide and conquer method to any integer $n \geq 0$: if $n = 0$, define $b^0 = 1$; if n is even, we halve n to compute $b^{\frac{n}{2}}$, then square it; if n is odd, since $n - 1$ is even, we recursively compute b^{n-1}, then multiply b:

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

However, $(b^{\frac{n}{2}})^2$ is not tail recursive. Alternatively, we square the base number, and halve the exponent.

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

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

$$pow(b, 0, a) = a$$

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

This implementation is bound to $O(\lg n)$ time. Write n in binary format $n = (a_m a_{m-1}...a_1 a_0)_2$, we need compute b^{2^i} if $a_i = 1$, similar to the Binomial heap (section 10.1) algorithm. Multiply them together finally. For example, when compute b^{11}, as $11 = (1011)_2 = 2^3 + 2 + 1$, gives $b^{11} = b^{2^3} \times b^2 \times b$. We follow these steps:

1. compute b^1, which is b;

2. Square to b^2;

3. Square to b^{2^2};

4. Square to b^{2^3}.

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

$$pow(b, 0, a) = a$$

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

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

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

```
1: function SUM(X)
2:     s ← 0
3:     while X ≠ NIL do
4:         s ← s+ FIRST(X)
5:         X ← REST(X)
6:     return s

7: function PRODUCT(X)
8:     p ← 1
9:     while X ≠ NIL do
10:        p ← p · FIRST(X)
11:        X ← REST(X)
12:    return p
```

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

1.3.6   maximum and minimum

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

$$\min \; [x] \quad = \quad x \qquad\qquad\qquad \max \; [x] \quad = \quad x$$

$$\min \; (x\!:\!xs) \quad = \quad \begin{cases} x < \min \; xs : & x \\ \text{otherwise} : & \min \; xs \end{cases} \qquad \max \; (x\!:\!xs) \quad = \quad \begin{cases} x > \max \; xs : & x \\ \text{otherwise} : & \max \; xs \end{cases}$$

$$(1.24)$$

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

$$\min' \; a \; [\,] \quad = \quad a$$

$$\min' \; a \; (x\!:\!xs) \quad = \quad \begin{cases} x < a : & \min' \; x \; xs \\ \text{otherwise} : & \min' \; a \; xs \end{cases} \qquad (1.25)$$

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

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

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

$$\min \; (x\!:\!xs) = \min' \; x \; xs \qquad\qquad \max \; (x\!:\!xs) = \max' \; x \; xs \qquad (1.26)$$

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

1: **function** MIN(X)
2:     $m \leftarrow$ FIRST(X)
3:     $X \leftarrow$ REST(X)
4:     **while** $X \neq$ NIL **do**
5:         **if** FIRST(X) $< m$ **then**
6:             $m \leftarrow$ FIRST(X)
7:         $X \leftarrow$ REST(X)
8:     **return** m

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

$$\min \; [x] \quad = \quad x$$

$$\min \; (x_1\!:\!x_2\!:\!xs) \quad = \quad \begin{cases} x_1 < x_2 : & \min \; (x_1\!:\!xs) \\ \text{otherwise} : & \min \; (x_2\!:\!xs) \end{cases} \qquad (1.27)$$

Exercise 1.8

1.8.1. Change *length* to tail recursive.

1.8.2. Compute b^n through the binary format of n.

1.4   Transform

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

1.4.1   map and for-each

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$solve$ and $toStr$ share the same structure. We abstract it as map:

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

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

$$map :: (A \to B) \to [A] \to [B] \qquad (1.34)$$

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

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

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

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

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

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

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

We use a sentinel node in the iterative MAP implementation.

```
1: function MAP(f, X)
2:     X' ← CONS(⊥, NIL)                        ▷ the sentinel
3:     p ← X'
4:     while X ≠ NIL do
5:         x ← FIRST(X)
6:         X ← REST(X)
7:         REST(p) ← CONS(f(x), NIL)
8:         p ← REST(p)
9:     return REST(X')                          ▷ discard the sentinel
```

For each

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

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

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

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

For example, consider the "n-lights puzzle"[96]. There are n lights in a room, all are off. We execute the following for n rounds:

1. Switch all lights on;

2. Switch lights of number 2, 4, 6, ... , that every other light is switched;

3. Switch every third lights, number 3, 6, 9, ... ;

4. ...

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

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

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

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

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

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

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

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

Below is the example program:

```
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 from 1 to 100 lights to give below answers (added line breaks):

```
[1,1,1,
 2,2,2,2,2,
 3,3,3,3,3,3,3,
 4,4,4,4,4,4,4,4,4,
 5,5,5,5,5,5,5,5,5,5,5,
 6,6,6,6,6,6,6,6,6,6,6,6,6,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,10]
```

They form a pattern: the first 3 answers are 1; the 4-th to the 8-th answers are 2; the 9-th to the 15-th answers are 3; ... It seems that from the i^2-th to the $((i+1)^2 - 1)$-th answers are i. Let's prove it:

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

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

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

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

```
map (floor ∘ sqrt) [1..100]
```

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

1.4.2   reverse

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

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

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

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

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

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

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

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

Exercise 1.9

1.9.1. Find the maximum v in a list of pairs $[(k, v)]$ in tail recursive way.

1.5   Sub-list

One can slice an array fast, but need linear time to traverse and extract sub-list. *take* extracts the first n elements. It is equivalent to: *sublist* $1\ n\ X$. *drop* discards the first n elements. It is equivalent to: *sublist* $(n + 1)\ |X|\ X$, which is symmetric to *take*[6]:

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

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

$$
sublist\ s\ n\ X = take\ n\ (drop\ (s-1)\ X) \tag{1.43}
$$

Or slice the list with left and right boundaries:

$$
slice\ s\ e\ X = drop\ (s-1)\ (take\ e\ X) \tag{1.44}
$$

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

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

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

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

---

[6]Some programming languages provide built-in implementation, for example in Python: `xs[:m]` and `xs[m:]` correspond to *take* and *drop*.

1.5.1 break and group

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

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

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

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

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

$$group\ \text{``Mississippi''} = [\text{``M''}, \text{``i''}, \text{``ss''}, \text{``i''}, \text{``ss''}, \text{``i''}, \text{``pp''}, \text{``i''}]$$

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

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

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

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

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

```
1: function GROUP(∼, X)
```

2:     **if** $X = $ NIL **then**
3:         **return** [[ ]]
4:     $x \leftarrow$ FIRST(X)
5:     $X \leftarrow$ REST(X)
6:     $g \leftarrow [x]$
7:     $G \leftarrow [g]$
8:     **while** $X \neq$ NIL **do**
9:         $y \leftarrow$ FIRST(X)
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:        $X \leftarrow$ NEXT(X)
17:    **return** G

However, the performance downgrades to quadratic without the tail reference optimization for APPEND. We can change to CONS if don't care the order. We can define the above 2 examples as *group* $(=)$ "Mississippi" and *group* $(\geq)$ X. Alternatively, we can realize grouping with *span*: repeatedly apply span to the rest till it becomes empty. However, span takes an unary function as the predication, while the group needs a binary. We solve it with Currying: pass and fix the first argument.

$$
\begin{aligned}
group \;\sim\; [\,] &= [[\,]] \\
group \;\sim\; (x{:}xs) &= (x{:}as) : group \;\sim\; bs, \text{where } (as, bs) = span\ (x \sim)\ xs
\end{aligned}
$$
(1.49)

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

1. **Reflexive**. $x \sim x$;

2. **Symmetric**. $x \sim y \Leftrightarrow y \sim x$.

3. **Transitive**. $x \sim y, y \sim z \Rightarrow x \sim z$;

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

Exercise 1.10

1.10.1. Change the *take/drop* implementation. When n is negative, returns [ ] for *take*, and the entire list for *drop*.

1.10.2. Implement the in-place imperative *take/drop*.

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

1.10.4. Consider the below *span* implementation:

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

What is the difference here?

1.6   Fold

Almost all list algorithms share the common structure. It is not by chance. The commonality is rooted from the recursive nature of list. We can abstract the list algorithm to a high level concept, fold[7], which is essentially the initial algebra of all list computations[99]. Observe *sum*, *product*, and *sort* for the common structure: the result for empty list is 0 for sum, 1 for product, and [ ] for sort; the binary operation that applies to the head and the recursive result. It's plus for sum, multiply for product, and ordered insertion for sort. We abstract the result for empty list as the *initial value z* (generic zero), the binary operation as $\oplus$. define:

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

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

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

The parentheses are necessary, because the computation starts from the right-most $(x_n \oplus z)$, repeatedly folds left towards x_1. This is quite similar to a fold-fan in fig. 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

Consider the fold-fan as a list of bamboo frames. The binary operation is to fold a frame to the top of the stack (initialized empty). To fold the fan, start from one end,

---

[7]also known as reduce

repeatedly apply the binary operation, till all the frames are stacked. The sum and product algorithms do the same thing essentially.

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

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

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

Define sum and product with $foldr$ as below:

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

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

Or in Curried form: $sum = foldr\ (+)\ 0$, $product = foldr\ (\times)\ 1$, for insertion-sort, it is: $sort = foldr\ insert\ [\,]$. Convert $foldr$ to tail recursive. It generates the result from left. denote it as $foldl$:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

What if we $fold$ a list of lists with "$+\!\!\!+$" (eq. (1.15))? It concatenates them to a list, just like sum to numbers.

$$concat = fold_r\ (+\!\!\!+)\ [\ ] \qquad (1.57)$$

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

Exercise 1.11

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

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

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

1.11.2. What's the performance of $concat$? Design a linear time $concat$ algorithm.

1.11.3. Define map in $foldr$.

1.7   Search and filter

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

$$a \in [\,] = False$$
$$a \in (b{:}bs) = \begin{cases} b = a : & True \\ b \neq a : & a \in bs \end{cases} \tag{1.60}$$

The existence check is also called *elem*. The performance is $O(n)$. We can not improve it to $O(\lg n)$ with binary search directly even for ordered list. This is because list does not support constant time random access (chapter 3). Let's extend *elem*. In the n-lights puzzle, we use a list of pairs $[(k, v)]$. Every pair contains a key and a value. Such list is called 'associate list' (abbrev. assoc list). We can lookup the value with a key.

$$lookup\ x\ [\,] = Nothing$$
$$lookup\ x\ ((k, v){:}kvs) = \begin{cases} k = x : & Just\ (k, v) \\ k \neq x : & lookup\ x\ kvs \end{cases} \tag{1.61}$$

Different from *elem*, we want to find the corresponding value besides the existence of key x. However, it is not guaranteed the value always exists. We use the algebraic type class 'Maybe'. A type of **Maybe** A has two kinds of value. It may be some a in A or nothing. Denoted as *Just a* and *Nothing* respectively. This is a way to deal with null reference[8] (4.2.2 in [99]). We can make *lookup* generic, to find the element that satisfies a given predicate:

$$find\ p\ [\,] = Nothing$$
$$find\ p\ (x{:}xs) = \begin{cases} p(x) : & Just\ x \\ otherwise : & find\ p\ xs \end{cases} \tag{1.62}$$

Although there can be multiple elements satisfying p, the *find* function picks the first. We can expand to find all, which is called *filter* as shown in fig. 1.4. Define (ZF expression): $filter\ p\ X = [x | x \leftarrow X, p(x)]$. We can use logic and to chain multiple filters, as: $filter\ (p_1 \wedge p_2 \wedge \cdots)\ X = [x | x \leftarrow X,\ p_1(x), p_2(x), \cdots]$[9].

$$\text{Input} \longrightarrow \boxed{\text{filter } p} \longrightarrow \text{Output}$$

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

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

$$filter\ p\ [\,] = [\,]$$
$$filter\ p\ (x{:}xs) = \begin{cases} p(x) : & x : filter\ p\ xs \\ otherwise : & filter\ p\ xs \end{cases} \tag{1.63}$$

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

---

[8]Similar to `Optional<A>` in some environments.
[9]We may use the simplified notation: $[x \leftarrow X, p_1(x), p_2(x), ...]$, or even $[p_1(x), p_2(x), ...]$ when context is clear

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

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

$$filter\ p = foldr\ (x\ as \mapsto f\ p\ x\ as)\ [\ ] \tag{1.64}$$

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

$$filter\ p = foldr\ (f\ p)\ [\ ] \tag{1.65}$$

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

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

$$
\begin{array}{rcl}
[\ ] \subseteq bs & = & True \\
(a{:}as) \subseteq [\ ] & = & False \\
(a{:}as) \subseteq (b{:}bs) & = & \begin{cases} a \neq b: & False \\ a = b: & as \subseteq bs \end{cases}
\end{array}
\tag{1.66}
$$

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

$$as \supseteq bs = reverse(as) \subseteq reverse(bs) \tag{1.67}$$

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

$$
\begin{array}{rcl}
infix?\ (a{:}as)\ [\ ] & = & False \\
infix?\ as\ (b{:}bs) & = & \begin{cases} as \subseteq (b{:}bs): & True \\ \text{otherwise}: & infix?\ as\ bs \end{cases}
\end{array}
\tag{1.68}
$$

Below is the iterative implementation:

```
1: function IS-INFIX(A, B)
2:     if A = NIL then
3:         return TRUE
4:     n ← |A|
5:     while B ≠ NIL and n ≤ |B| do
6:         if A ⊆ B then
7:             return TRUE
8:         B ← REST(B)
```

9:       **return** FALSE

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

$$infix?\ A\ B = \exists S \in suffixes\ B, A \subseteq S \tag{1.69}$$

Below example program implements infix testing with list comprehension:

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

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

Exercise 1.12

1.12.1. Implement the linear time filter algorithm through *reverse*.

1.12.2. Enumerate all suffixes of a list.

1.8   zip and unzip

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

$$
\begin{aligned}
zip\ as\ [\ ] &= [\ ] \\
zip\ [\ ]\ bs &= [\ ] \\
zip\ (a{:}as)\ (b{:}bs) &= (a, b) : zip\ as\ bs
\end{aligned}
\tag{1.70}
$$

This implementation works even the two lists have different lengths. The result has the same length as the shorter one. We can even zip infinite lists (under lazy evaluation), for example[10]: $zip\ [0, 0, ...]\ [1, 2, ..., n]$. For a list of words, we can index it as: $zip\ [1, 2, ...]\ [a, an, another, ...]$. zip builds the result from right. We can define it with $foldr$. It is bound to $O(m)$ time, where m is the length of the shorter list. When implement the iterative zip, the performance will drop to quadratic if using APPEND, we can use CONS then reverse the result. However, this method can't handle two infinite lists. In imperative settings, we can reuse A to hold the zip result (treat as transform every element to a pair).

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

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

---

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

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

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

$$
\begin{array}{rcl}
zipWith\ f\ as\ [\,] & = & [\,] \\
zipWith\ f\ [\,]\ bs & = & [\,] \\
zipWith\ f\ (a{:}as)\ (b{:}bs) & = & (f\ a\ b) : zipWith\ f\ as\ bs
\end{array}
\tag{1.71}
$$

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

$$
F = 0 : 1 : zipWith\ (+)\ F\ F'
\tag{1.72}
$$

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

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

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

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

$$
unzip = foldr\ ((a,b)\ (as,bs) \mapsto (a{:}as, b{:}bs))\ ([\,],[\,])
\tag{1.73}
$$

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

$$
pay = sum\ (zipWith\ (\cdot)\ snd(unzip\ U)\ snd(unzip\ Q))
\tag{1.74}
$$

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

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

Exercise 1.13

1.13.1. Design the iota (the Greek letter I) operator for list, below are the use cases:

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

- $iota(m, m+a, ..., n) = [m, m+a, m+2a, ..., m+ka]$, where k is the maximum integer satisfying $m + ka \leq n$;

- $iota(m, m, ...) = repeat(m) = [m, m, m, ...]$;

- $iota(m, ...) = [m, m+1, m+2, ...]$.

1.13.2. Implement the linear time imperative *zip*.

1.13.3. Define *zip* with fold (hint: define fold for two lists $foldr2\ f\ z\ xs\ ys$).

1.13.4. Implement *lastAt* with *zip*.

1.13.5. Write a program to remove the duplicated elements in a list while maintain the original order. For imperative implementation, the elements should be removed in-place. What is the complexity? How to simplify it with additional data structure?

1.13.6. List can represent decimal non-negative integer. For example 1024 as list is $4 \rightarrow 2 \rightarrow 0 \rightarrow 1$. Generally, $n = d_m...d_2d_1$ can be represented as $d_1 \rightarrow d_2 \rightarrow ... \rightarrow d_m$. Given two numbers a, b in list form. Realize arithmetic operations such as add and subtraction.

1.13.7. In imperative settings, a circular linked-list is corrupted, that some node points back to previous one, as shown in fig. 1.6. 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).
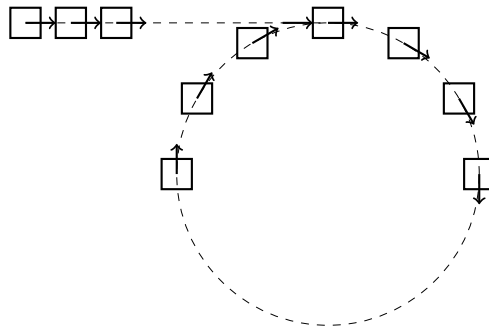
Figure 1.5: A circular linked-list

Chapter 2

Binary Search Tree

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 (section 3.4). While in functional settings, linked-list acts as the building blocks to create array and other data structures. The binary search trees is another basic data structure. Jon Bentley gives a problem in *Programming Pearls*[2]: how to count the number of word occurrences in a text. Here is a solution:

```
void wordCount(Input in) {
    Map<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)
    }
}
```

2.1 Definition

The `map` is a binary search tree. Here we use the word as the key, and its occurrence number as the value. This program is a typical application of binary search tree. Let us firstly define the *binary tree*. A binary tree is either empty $(\varnothing)$[1]; or contains 3 parts: an element k, and two sub-trees called left(l) and right(r) children, denoted as (l, k, r). A none empty binary tree consists of multiple nodes, each is either empty or stores the element of type K. We define the type of the binary tree as $Tree\ K$. We say a node is a leaf if both sub-trees are empty, otherwise it's a branch node.

A binary search tree is a special binary tree that its elements are comparable[2], and satisfies: for any non empty node (l, k, r), all the keys in the left sub-tree $< k$; $k <$ any key in the right sub-tree. Figure 2.2 shows an example of binary search tree. Comparing with fig. 2.1, we can see the different ordering. For this reason, we call the comparable element as *key*, and the augmented data as *value*. The type is $Tree\ (K, V)$. A node contains a key, a value (optional), left, right sub-tree references, and a parent reference for easy backtracking. When the context is clear, we skip the value. The appendix

---

[1]The great mathematician André Weil invented this symbol for null set. It comes from the Norwegian alphabet.

[2]It is abstract ordering, not limit to magnitude, but like precedence, subset of etc. the 'less than' $(<)$ is abstract.
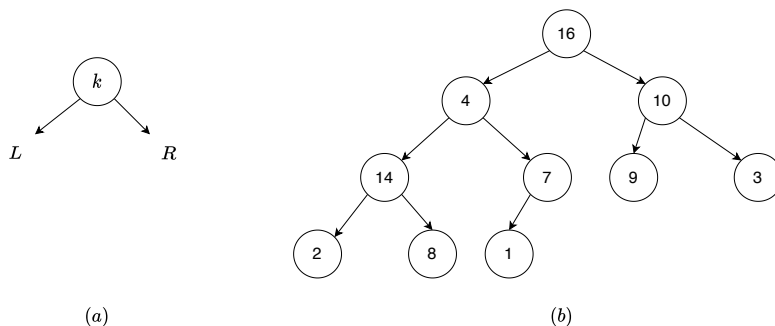
Figure 2.1: Binary tree

of this chapter includes an example definition. We needn't reference for backtracking in functional settings, but use top-down recursive computation. Below is the example functional definition (known as algebraic data type, abbrev: ADT):
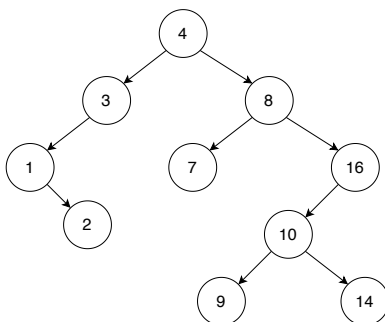
Figure 2.2: A binary search tree

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

2.2   Insert

When insert a key k (or with the value) to the binary search tree T, we need maintain the ordering. If the tree is empty, create a leaf of k. Otherwise, let the tree be (l, x, r). If $k < x$, insert it to the left sub-tree l; otherwise, insert to the right r. If $k = x$, it already exists in the tree. We overwrite the value (update). Alternatively, we can append the data or do nothing. We skip this case.

$$
\begin{aligned}
insert\ k\ \varnothing &= (\varnothing, k, \varnothing) \\
insert\ k\ (l, x, r) &= \begin{cases} k < x: & (insert\ k\ l, x, r) \\ \text{otherwise}: & (l, x, insert\ k\ r) \end{cases}
\end{aligned}
\tag{2.1}
$$

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

This implementation uses the *pattern matching* feature. There is an example without pattern matching in the appendix. We can eliminate the recursion with iterative loops:

1: **function** INSERT(T, k)
2:      $root \leftarrow T$
3:      $x \leftarrow$ CREATE-LEAF(k)
4:      $parent \leftarrow$ NIL
5:      **while** $T \neq$ NIL **do**
6:          $parent \leftarrow T$
7:          **if** $k <$ KEY(T) **then**
8:              $T \leftarrow$ LEFT(T)
9:          **else**
10:             $T \leftarrow$ RIGHT(T)
11:     PARENT(x) $\leftarrow parent$
12:     **if** $parent =$ NIL **then** $\triangleright$ T in empty
13:         **return** x
14:     **else if** $k <$ KEY($parent$) **then**
15:         LEFT($parent$) $\leftarrow x$
16:     **else**
17:         RIGHT($parent$) $\leftarrow x$
18:     **return** $root$

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

Where KEY(T) accesses the key of the node:

$$
\begin{aligned}
key\ \varnothing &= \quad Nothing \\
key\ (l, k, r) &= \quad Just\ k
\end{aligned}
\tag{2.2}
$$

We can repeat insert every element from a list, convert the list to a binary search tree:

$$
\begin{aligned}
fromList\ [\,] &= \quad \varnothing \\
fromList\ (x{:}xs) &= \quad insert\ x\ (fromList\ xs)
\end{aligned}
$$

Or define it with fold (chapter 1) in Curried form: $fromList = foldr\ insert\ \varnothing$. We arrange the arguments in symmetric order: $insert\ k\ t$ and INSERT(T, k), apply $foldr$ for the former, and $foldl$ (or for-loop) for the latter:

1: **function** FROM-LIST(X)
2:      $T \leftarrow$ NIL
3:      **for** each x in X **do**
4:          $T \leftarrow$ INSERT(T, x)
5:      **return** T

2.3   Traverse

There are 3 ways to visit the elements in a binary tree: pre-order, in-order, and post-order. They are named to highlight the order of visiting key between/before/after sub-trees.

- pre-order: **key** - left - right;

- in-order: left - **key** - right;

- post-order: left - right - **key**.

The 'visit' is recursive. For the tree in fig. 2.2, the corresponding orders are:

- 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 gives an ascending list, but is guaranteed (Exercise 2.1.3). Define *map* that in-order traverses and applies function f to every element. It transforms a tree to another tree of the same structure (isomorphic).

$$\begin{array}{rcl} map\ f\ \varnothing & = & \varnothing \\ map\ f\ (l,k,r) & = & (map\ f\ l, f\ k, map\ f\ r) \end{array} \tag{2.3}$$

If we only need process keys but not transform the tree, we can implement in-order traverse as below:

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

We can change the *map* function, and convert a binary search tree to a sorted list (also called flatten).

$$\begin{array}{rcl} toList\ \varnothing & = & [\ ] \\ toList\ (l,k,r) & = & toList\ l + [k] + toList\ r \end{array} \tag{2.4}$$

We can develop a sort algorithm: convert a list to a binary search tree, then convert the tree back to ordered list, namely 'tree sort': $sort\ X = toList\ (fromList\ X)$, or write as function composition [8]:

$$sort = toList \circ fromList \tag{2.5}$$

Define the generic fold for binary trees (see chapter 1 for fold):

$$\begin{array}{rcl} foldt\ f\ g\ z\ \varnothing & = & z \\ foldt\ f\ g\ z\ (l,k,r) & = & g\ (foldt\ f\ g\ z\ l)\ (f\ k)\ (foldt\ f\ g\ z\ r) \end{array} \tag{2.6}$$

Where $f : A \to B$, sends the key k of type A to $m = f(k)$ of type B. It recursively folds the left and right sub-trees (from z) to get x and y respectively, then combines the three things together as $g\ x\ m\ y$. We can define *map* with $foldt$:

$$map\ f = foldt\ f\ (x\ m\ y \mapsto (x, m, y))\ \varnothing \tag{2.7}$$

$foldt$ preserves the tree structure with the ternary function g. If don't care about the tree structure, we can use a binary function $f : A \times B \to B$ to simplify and fold a tree of type $Tree\ A$ to a value of type B:

$$\begin{array}{rcl} fold\ f\ z\ \varnothing & = & z \\ fold\ f\ z\ (l,k,r) & = & fold\ f\ (f\ k\ (fold\ f\ z\ r))\ l \end{array} \tag{2.8}$$

For example: $sum = fold\ (+)\ 0$ sums all elements of the tree; $length = fold\ (x\ n \mapsto n+1)\ 0$ counts the number of elements. However, $fold$ can not define *map*, as the binary function f losses the tree structure.

Exercise 2.1

2.1.1. Given the in-order and pre-order traverse results, rebuild 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: ?

2.1.2. Write a program to rebuild the binary tree from the pre-order and in-order traverse lists.

2.1.3. For binary search tree, prove that the in-order traverse always gives ordered list.

2.1.4. What is the complexity of tree sort?

2.1.5. Define *toList* with fold.

2.1.6. Define *depth t* with fold, which calculates the height of a binary tree.

2.4 Query

Because the binary search tree organises ordered elements recursively, it supports varies of query efficiently. This is the reason we name it 'search' tree. There are three types of query: (1) lookup a key; (2) find the minimum or maximum; (3) given a node, find its predecessor or successor. When lookup the value of some key x in a tree of type *Tree* (K, V):

- If the tree is empty, x does not exist;

- For tree $(l, (k, v), r)$, if $k = x$, then v is the result;

- If $x < k$, then recursively lookup l, otherwise, lookup r.

$$
\begin{aligned}
lookup\ x\ \varnothing &= Nothing \\
lookup\ x\ (l, (k, v), r), x) &= \begin{cases} k = x: & Just\ v \\ x < k: & lookup\ x\ l \\ \text{otherwise}: & lookup\ x\ r \end{cases}
\end{aligned}
\tag{2.9}
$$

We use the *Maybe* type[3] to handle the 'not found' case. Let the height of the tree be h, the performance of *lookup* is $O(h)$. If the tree is balanced (see chapter 4), the performance is $O(\lg n)$, where n is the number of elements. It decreases to $O(n)$ time in the worse case for extremely unbalanced tree. Below implementation eliminates the recursion with loops:

```
1: function LOOKUP(T, x)
2:     while T ≠ NIL and KEY(T) ≠ x do
3:         if x < KEY(T) then
4:             T ← LEFT(T)
5:         else
6:             T ← RIGHT(T)
7:     return VALUE(T)                                    ▷ returns ∅ if T =NIL
```

---

[3]Also known as `Optional<T>` type, see chapter 1.

In binary search tree, the less keys are on the left, while the greater keys are on the right. To locate the minimum, we keep going to the left till the left sub-tree is empty. Symmetrically, we keep going to the right to find the maximum. Both min / max are bound to $O(h)$ time, where h is the height of the tree.

$$\begin{array}{rclcrcl} \min (\varnothing, k, r) & = & k & \quad & \max (l, k, \varnothing) & = & k \\ \min (l, k, r) & = & \min l & \quad & \max (l, k, r) & = & \max r \end{array} \qquad (2.10)$$

We sometimes traverse a binary search tree as a container. Start from the minimum, keep moving forward step by step 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 case need to find the successor or predecessor of a node. Define the successor of x as the smallest y that $x < y$. If x has none empty right sub-tree r, the minimum of r is the successor. As shown in fig. 2.3, to find the successor of 8, we search the minimum in its right, which is 9. If the right sub-tree of x is empty, we need back-track along the parent till the closest ancestor whose left sub-tree is also an ancestor of x. In fig. 2.3, 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.
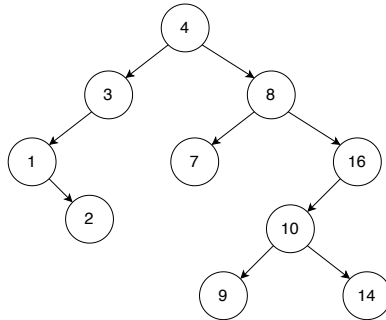
Figure 2.3: The successor of 8 is 9, the minimum of its right; for the successor of 2, we go up to its parent 1, then 3.

If we finally reach to the root along the parent path, but still can not find an ancestor on the right, then the node does not have the successor (the last element). Below algorithm finds the successor of 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 hasn't the successor. The predecessor algorithm is symmetric:

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

The purely functional settings don't use parent reference[4]. Some implementation records the visited paths for back-track or tree rebuilding, called zipper(last chapter of[10]). The original purpose for SUCC and PRED is 'to traverse the tree' as a container. However, we typically in-order traverse the tree through *map* in functional settings. It's only meaningful to find the successor and predecessor in imperative settings.

Exercise 2.2

2.2.1. How to test whether an element k exists in the tree t?

2.2.2. 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?

2.2.3. One can traverse the elements inside 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.

2.5   Delete

We need maintain the ordering while delete: for any node (l, k, r), all left are still less than k, all right are still greater than k after delete. To delete x[6]: (1) if x is a leaf or only has a none empty sub-tree, cut x off; (2) if x has two none empty sub-trees, use the minimum y of its right sub-tree to replace x, then cut the original y off. Because the minimum of the right sub-tree can not have two none empty sub-trees, we eventually convert case 2 to 1, directly cut the minimum node off, as shown in figs. 2.4 to 2.6.
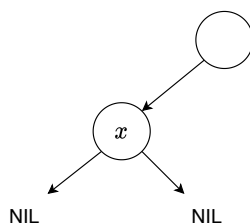
Figure 2.4: Cut the leaf x off.

---

[4]There is `ref` in ML and OCaml, we limit to the purely functional settings.
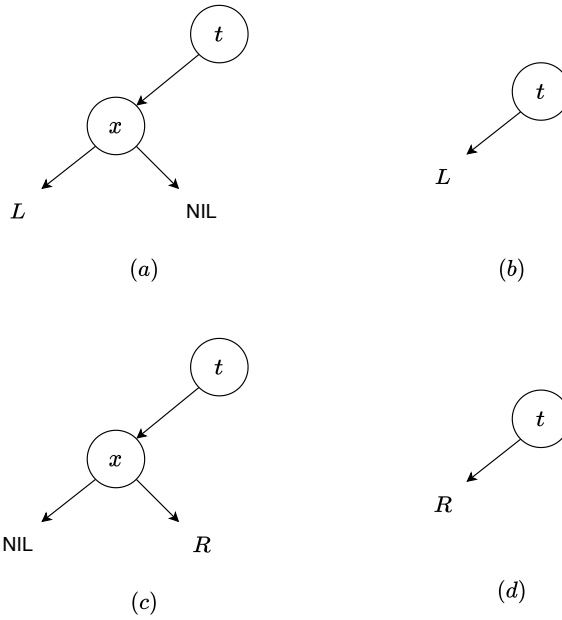
Figure 2.5: Delete a node with only a none empty sub-tree.
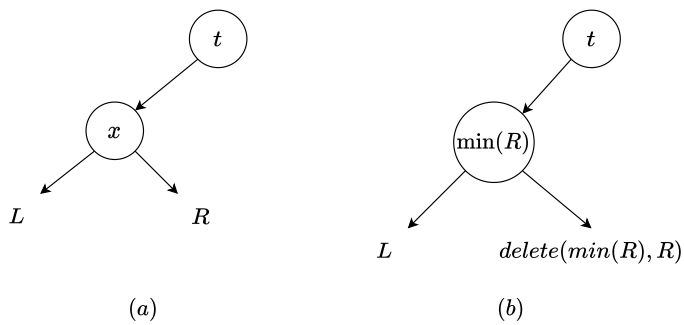
Figure 2.6: Delete a branch with two none empty sub-trees.

$$delete \; x \; \varnothing \;\; = \;\; \varnothing$$

$$delete \; x \; (l, k, r) \;\; = \;\; \begin{cases} x < k : & (delete \; x \; l, k, r) \\ x > k : & (l, k, delete \; x \; r) \\ x = k : & del \; l \; r \end{cases} \tag{2.11}$$

Where:

$$\begin{aligned} del \; \varnothing \; r \;\; &= \;\; r \\ del \; l \; \varnothing \;\; &= \;\; l \\ del \; l \; r \;\; &= \;\; (l, y, delete \; y \; r), \text{where } y = \min \; r \end{aligned} \tag{2.12}$$

The performance of delete is $O(h)$, where h is the height. The imperative implementation needs set the parent reference in addition.

```
1: function DELETE(T, x)
2:     r ← T
3:     x' ← x                                              ▷ save x
4:     p ← PARENT(x)
5:     if LEFT(x) = NIL then
6:         x ← RIGHT(x)
7:     else if RIGHT(x) = NIL then
8:         x ← LEFT(x)
9:     else                                        ▷ neither sub-tree is empty
10:        y ← MIN(RIGHT(x))
11:        KEY(x) ← KEY(y)
12:        VALUE(x) ← VALUE(y)
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
```

Assume x is not empty, first record the root, copy reference to x and its parent. When delete, we also need handle the special case, that y is the root of the right sub-tree. Finally, we need reset the stored parent if x has only one none empty sub-tree. If the copied parent is empty, we are deleting the root. We return the new root in this case. After setting the parent, we can safely remove x.

The performance of the binary search tree algorithms depend on the height h. When unbalanced, $O(h)$ is close to $O(n)$, while for well balanced tree, $O(h)$ is close to $O(\lg n)$.

Chapter 4 and 5 introduce self-balanced solution, there is another simple balance method: shuffle the elements, then build the tree[4]. It decreases the possibility of poorly balanced.

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. Each key is unique, and mapped to some value. For keys of type K, values of type V, the type of 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 an ordered set. Every node stores a pair of key and value. The type of the tree is $Tree\ (K, V)$. We use the tree insert/update operation to associate a key with a value. Given a key k, we use *lookup* to find the mapped value v, or returns nothing or $\varnothing$ when k does not exist. The red-black tree and AVL tree in chapter 4 and 5 can also implement map.

Exercise 2.3

2.3.1. There is a symmetric deletion algorithm. When neither sub-tree is empty, we replace with the maximum of the left sub-tree, then cut it off. Write a program to implement this solution.

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

2.3.3. How to find the two nodes with the greatest distance in a binary tree?

2.6   Appendix: Example programs

Definition of binary search tree node with parent reference.

```
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
    }
}
```

Recursive insert without using 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))
    }
}
```

Map and fold:

```
mapt _ Empty = Empty
mapt f (Node l x r)= Node (mapt f l) (f x) (mapt f r)

foldt _ _ z Empty = z
foldt f g z (Node l k r) = g (foldt f g z l) (f k) (foldt f g z r)
```

```
maptr :: (a → b) → Tree a → Tree b
maptr f = foldt f Node Empty

fold _ z Empty = z
fold f z (Node l k r) = fold f (k `f` (fold f z r)) l
```

Iterative lookup without recursion:

```
Optional<Node<T>> lookup (Node<T> t, T x) {
    while (t ≠ null and t.key ≠ x) {
        if (x < t.key) {
            t = t.left
        } else {
            t = t.right
        }
    }
    return Optional.of(t);
}
```

Example iterative program to find the minimum of a tree.

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

Iterative find the successor.

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

delete:

```
delete _ Empty = Empty
delete x (Node l k r) | x < k = Node (delete x l) k r
                      | x > k = Node l k (delete x r)
                      | 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 k' r)
```

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 fig. 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:     $S \leftarrow [\ ]$
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 of $A[1]$ is ordered. When process the i-th element, all elements before i are sorted. We continuously insert elements till n, as shown in fig. 3.2.
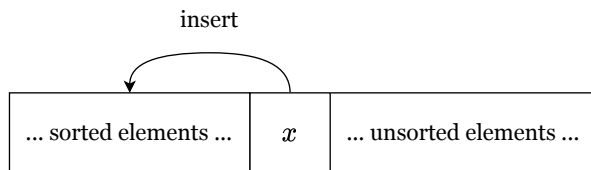
Figure 3.2: Continuously insert elements to the sorted part.

3.2 Insertion

In chapter 1, we define the ordered insertion for list. For array, we scan 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. $A[i+1], A[i+2], ...$) right. then put x in the freed cell, as shown in fig. 3.3.
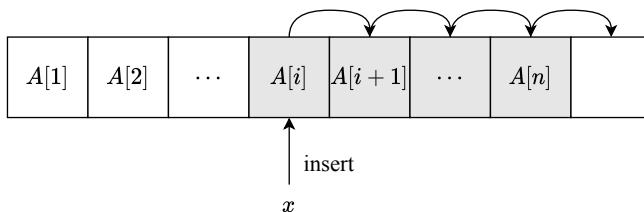
Figure 3.3: Insert x to A at i.

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 traverse the entire array if scan from left. While, if scan from right, we examine $n - i + 1$ elements, and do the same amount of shifts. The insertion takes linear time no matter scans from left or right, hence the sort algorithm is bound to $O(n^2)$. We can also define a separated INSERT() function, and call it inside the loop.

Exercise 3.1

3.1.1. Implement the insert to scan from left to right.

3.1.2. 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** $i \leftarrow 2$ to $|A|$ **do**
3:         $x \leftarrow A[i]$
4:         $p \leftarrow$ BINARY-SEARCH(x, $A[1...(i-1)]$)
5:         **for** $j \leftarrow i$ down to p **do**
6:             $A[j] \leftarrow A[j-1]$
7:         $A[p] \leftarrow x$

Because the slice $A[1...(i-1)]$ is already ordered, to find the position j such that $A[j-1] \leq x \leq A[j]$, 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 we halve the elements every time, binary search takes $O(\lg i)$ time to locate the insert position.

1: **function** BINARY-SEARCH(x, A)
2:     $l \leftarrow 1, u \leftarrow 1 + |A|$
3:     **while** $l < u$ **do**
4:         $m \leftarrow \lfloor \dfrac{l+u}{2} \rfloor$
5:         **if** $A[m] = x$ **then**
6:             **return** m ▷ Duplicated element
7:         **else if** $A[m] < x$ **then**
8:             $l \leftarrow m + 1$
9:         **else**
10:             $u \leftarrow m$
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 takes $O(n \lg n)$ comparisons and $O(n^2)$ shifts.

3.4 List

With binary search, the total number of comparisons reduced 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:

$$\begin{aligned} sort\ [\,] &= [\,] \\ sort\ (x\!:\!xs) &= insert\ x\ (sort\ xs) \end{aligned} \tag{3.1}$$

Or define with $foldr$ in Curried form: $sort = foldr\ insert\ [\,]$. However, the list $insert$ algorithm still takes linear time, because we need scan to locate the insert position:

$$\begin{aligned} insert\ x\ [\,] &= [x] \\ insert\ x\ (y\!:\!ys) &= \begin{cases} x \leq y: & x : y : ys \\ \text{otherwise}: & y : insert\ x\ ys \end{cases} \end{aligned} \tag{3.2}$$

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:     $j \leftarrow 0$ ▷ $Next[0]$ for head
3:     **while** $Next[j] \neq -1$ and $A[Next[j]] < A[i]$ **do**
4:         $j \leftarrow Next[j]$
5:     $Next[i] \leftarrow Next[j]$
6:     $Next[j] \leftarrow i$

7: **function** SORT(A)
8:     $n \leftarrow |A|$
9:     $Next = [1, 2, ..., n, -1]$ ▷ $n + 1$ indexes
10:     **for** $i \leftarrow 1$ 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 3.2

3.2.1. 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)

Or $sort = toList \circ fromList$ for list, where INSERT-TREE(), TO-LIST(), and $fromList$ are defined in chapter 2. In average case, the performance of tree sort is bound to $O(n \lg n)$. This is the lower limit of comparison based sort([12] pp.180-193). However, in the worst case that the tree is poor balanced the performance drops to $O(n^2)$.

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

As the example in chapter 2, we use the binary search tree as a dictionary to count the word occurrence. One may want to feed a address book, and use the binary search tree to lookup the contact, for example:

```
void addrBook(Input in) {
    Map<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 in lexicographic order. If insert numbers 1, 2, 3, ..., n to a binary search tree, it ends up like in fig. 4.1. It is an extremely unbalanced binary search tree. The *lookup* is bound to $O(h)$ time for a tree of height h. When the tree is well balanced, the performance is $O(\lg n)$, where n is the number of elements. But in this extreme case, the performance downgrades to $O(n)$, same as list scan.
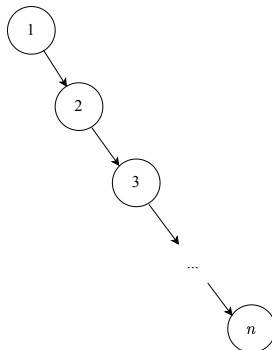
Figure 4.1: unbalanced tree

Exercise 4.1

4.1.1. For a big address book in lexicographic order, one may want to speed up with two concurrent tasks: one reads from the head; the other from the tail. They meet and stop at some middle point. What does the binary search tree look like? What if split the list into multiple sections to scale up the concurrency?
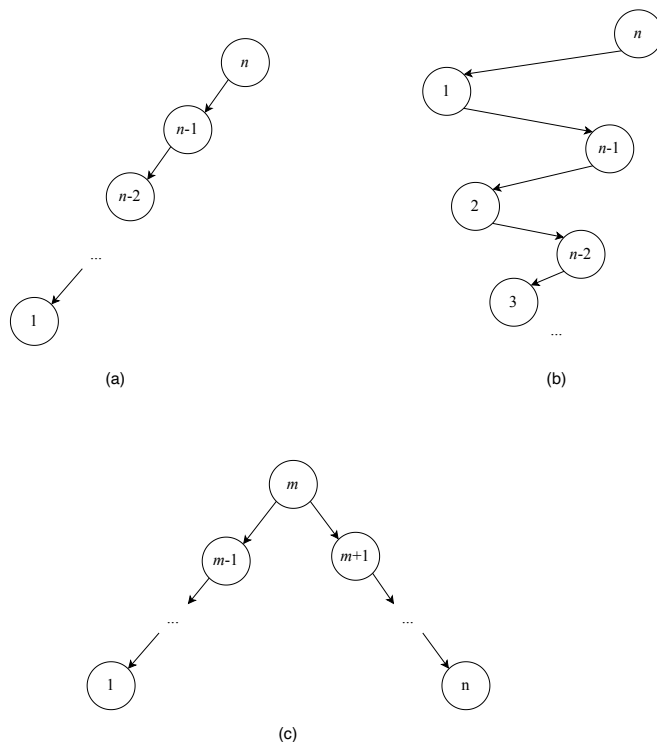
Figure 4.2: Unbalanced trees

4.1 Balance

To avoid extremely unbalanced tree, we can shuffle the input(section 2.5), however, we can not randomize interactive input (e.g. entered by user). Most re-balancing solutions rely on the tree rotation. It changes the tree structure while maintain the elements ordering. This chapter introduces the red-black tree, a popular self-balancing binary search tree. Next chapter is about AVL tree, another self-balancing tree. Chapter 8 introduces the splay tree. It adjusts the tree in steps. Multiple binary trees can have the same in-order traverse result. Figure 4.3 shows the tree rotation. We can define them with pattern matching:

$$\begin{aligned} rotate_l\ (a, x, (b, y, c)) &= ((a, x, b), y, c)) & rotate_r\ ((a, x, b), y, c) &= (a, x, (b, y, c)) \\ rotate_l\ T &= T & rotate_r\ T &= T \end{aligned}$$
$$(4.1)$$

Each second clause keeps the tree unchanged if the pattern does not match (e.g. both sub-trees are empty). We can also implement tree rotation imperatively. We need re-assign sub-trees and parent reference. When rotate, we pass both the root T, and the node x as parameters:
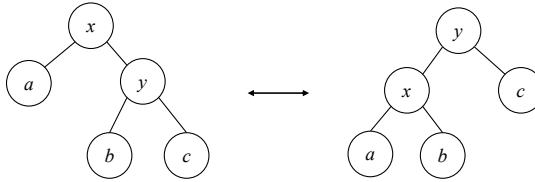
Figure 4.3: 'left rotate' and 'right rotate'.

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
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:        $T \leftarrow y$
12:    **return** T

The RIGHT-ROTATE is symmetric (as Exercise 4.2). The REPLACE(x, y) uses node y to replace x:

1: **function** REPLACE(x, y)
2:     $p \leftarrow$ PARENT(x)
3:     **if** $p =$ NIL **then**   ▷ x is the root
4:         **if** $y \neq$ NIL **then** PARENT$(y) \leftarrow$ NIL
5:     **else if** LEFT$(p) = x$ **then**
6:         SET-LEFT(p, y)
7:     **else**
8:         SET-RIGHT(p, y)
9:     PARENT$(x) \leftarrow$ 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$(x) \leftarrow y$
3:     **if** $y \neq$ NIL **then** PARENT$(y) \leftarrow x$

4: **function** SET-RIGHT(x, y)
5:     RIGHT$(x) \leftarrow y$
6:     **if** $y \neq$ NIL **then** PARENT$(y) \leftarrow x$

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

4.2.1. Implement the RIGHT-ROTATE.

4.2   Definition

A red-black tree is a self-balancing binary search tree[14]. It is 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.

3. Every NIL node 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 exceed 2 times of the shortest path. Consider rule 4,
there can not be any two adjacent red nodes. Hence 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 can't
exceed 2 times of the others[14]. Figure 4.4 gives an example of red-black tree.
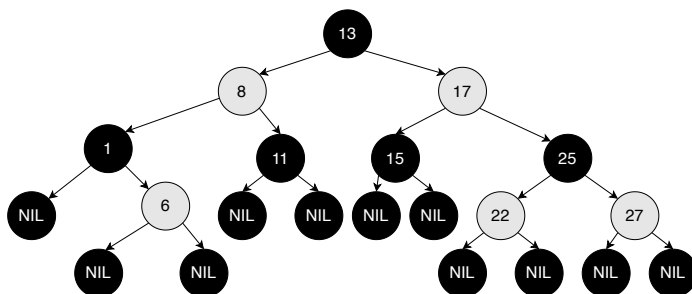
Figure 4.4: A red-black tree

As all NIL nodes are black, we can hide them as shown in fig. 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. Below example program adds
the color variable atop binary search tree definition. Denote the empty tree as $\varnothing$, the
none empty tree as (c, l, k, r), where c is the color (red/black), k is the element, l and r
are left and right sub-trees.

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

Exercise 4.3

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

---

[1]Chapter 7, B-tree. For any 2-3-4 tree, there is at least one red-black tree has the same ordered data.
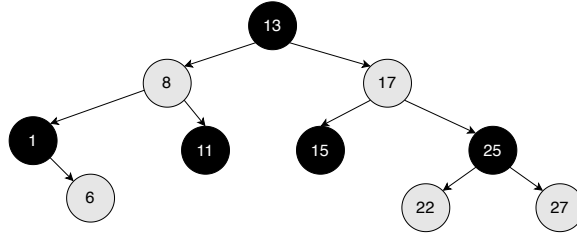
Figure 4.5: Hide the NIL nodes

4.3 Insert

The *insert* operation takes two steps. The first is as same as the binary search tree. The second is to restore the coloring if it becomes unbalanced. We always color the new element red unless it is the root. Hence don't break any coloring rules except the 4-th (As it may bring two adjacent red nodes). There are 4 cases violate rule 4. They share the same structure after fixing[13] as shown in fig. 4.6.
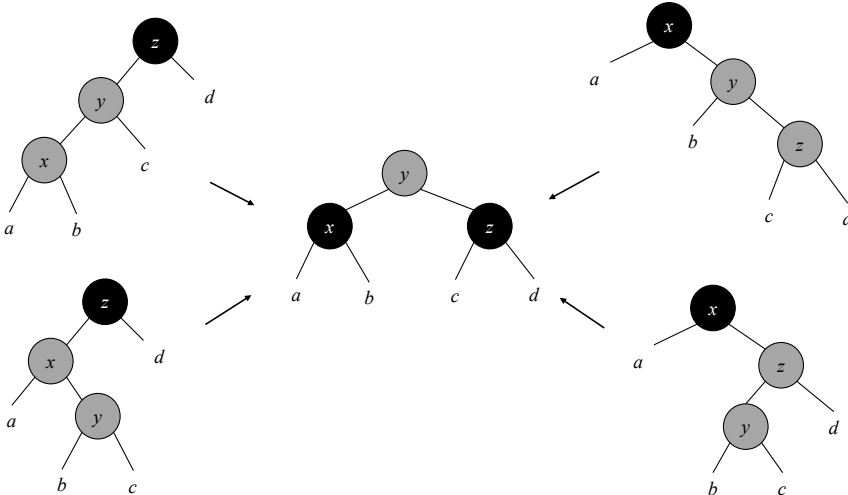
Figure 4.6: Fix 4 cases to the same structure.

All 4 transformations move the redness one level up. When fix recursively bottom-up, it may color the root red, hence violates rule 2. We need revert the root black finally. Define a *balance* function to fix the coloring with pattern matching. Denote the color as $\mathcal{C}$ with values black $\mathcal{B}$, and red $\mathcal{R}$.

$$
\begin{array}{rcl}
balance\ \mathcal{B}\ (\mathcal{R}, (\mathcal{R}, a, x, b), y, c)\ z\ d & = & (\mathcal{R}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, d)) \\
balance\ \mathcal{B}, (\mathcal{R}, a, x, (\mathcal{R}, b, y, c))\ z\ d & = & (\mathcal{R}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, d)) \\
balance\ \mathcal{B}\ a\ x\ (\mathcal{R}, b, y, (\mathcal{R}, c, z, d)) & = & (\mathcal{R}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, d)) \\
balance\ \mathcal{B}\ a\ x\ (\mathcal{R}, (\mathcal{R}, b, y, c), z, d) & = & (\mathcal{R}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, d)) \\
balance\ T & = & T
\end{array}
\tag{4.2}
$$

If none of the 4 patterns matches, we leave the tree unchanged. Define *insert x T = makeBlack (ins x T)*, or in Curried form:

$$
insert\ x = makeBlack \circ ins\ x
\tag{4.3}
$$

Where:

$$ins\ x\ \varnothing\quad =\quad (\mathcal{R}, \varnothing, x, \varnothing)$$
$$ins\ x\ (\mathcal{C}, l, k, r)\quad =\quad \begin{cases} x < k: & balance\ \mathcal{C}\ (ins\ x\ l)\ k\ r \\ x > k: & balance\ \mathcal{C}\ l\ k\ (ins\ x\ r) \end{cases} \tag{4.4}$$

If the tree is empty, we create a red leaf of x; otherwise, compare x and k, recursively insert x to a sub-tree. After that, call *balance* to fix the coloring, finally force the root to be black.

$$makeBlack\ (\mathcal{C}, l, k, r) = (\mathcal{B}, l, k, r) \tag{4.5}$$

Below is the example program:

```
insert x = makeBlack ∘ (ins x) where
    ins x Empty = Node R Empty x Empty
    ins x (Node color l k r)
        | x < k      = balance color (ins x l) k r
        | otherwise = balance color l k (ins x 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 is well balanced even for ordered input.
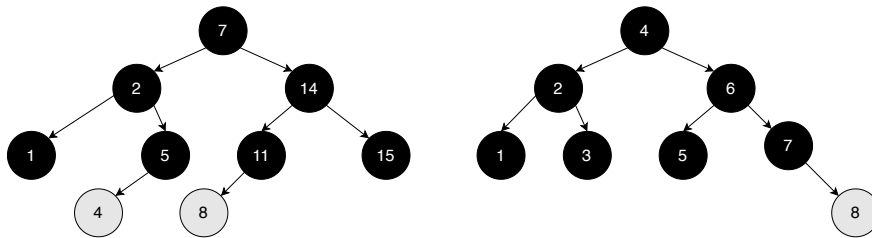
Figure 4.7: Red-black tree examples

The *insert* performs top-down recursive fixing. It is bound to $O(h)$ time, where h is the height. As the coloring rules are maintained, h is logarithm to n, the number of elements. The overall performance is $O(\lg n)$.

Exercise 4.4

4.4.1. Implement the *insert* without pattern matching, handle the 4 cases separately.

4.4 Delete

Delete is more complex than insert. We can simplify the recursive implementation with pattern matching[2]. There are alternative implementation to mimic delete. Build a read-only tree for frequently looking up[5]. When delete a node, mark it with a flag, and trigger

---

[2]Actually, we reuse the unchanged part to rebuild the tree in purely functional settings, known as the 'persist' feature

tree rebuilding if such nodes exceed 50%. Delete may also violate the coloring rules, hence need fixing. The violation only happens when delete a black node according to rule 5. The black nodes along the path decrease by one, causing not all paths contain the same number of black nodes. To resume the blackness, we introduce a special 'doubly-black' node([4], pp290). Such a node is counted as 2 black nodes. When delete a black node x, move the blackness up to parent or down to a sub-tree. Let node y accept the blackness. If y was red, turn it black; if y was already black, turn it 'doubly-black' as $\mathcal{B}^2$. Below example program adds the 'doubly-black' to color definition.

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

Because NIL is black, when push the blackness down to NIL, it becomes 'doubly-black' empty (`BBEmpty`, or bold $\varnothing$). The first step is normal binary search tree delete; then, if we cut a black node off, shift the blackness, and fix the coloring (Curried form):

$$delete \ x = makeBlack \circ del \ x \qquad (4.6)$$

When delete a singleton tree, it becomes empty. To cover this case, we modify $makeBlack$ as:

$$\begin{aligned} makeBlack \ \varnothing &= \varnothing \\ makeBlack \ (\mathcal{C}, l, k, r) &= (\mathcal{B}, l, k, r) \end{aligned} \qquad (4.7)$$

Where del accepts x and the tree:

$$del \ x \ \varnothing = \varnothing$$
$$del \ x \ (\mathcal{C}, l, k, r) = \begin{cases} x < k: & fixB^2(\mathcal{C}, (del \ x \ l), k, r) \\ x > k: & fixB^2(\mathcal{C}, l, k, (del \ x \ r)) \\ x = k: & \begin{cases} l = \varnothing: & \text{if } \mathcal{C} = \mathcal{B} \text{ then } shiftB \ r \text{ else } r \\ r = \varnothing: & \text{if } \mathcal{C} = \mathcal{B} \text{ then } shiftB \ l \text{ else } l \\ \text{otherwise}: & fixB^2(\mathcal{C}, l, m, (del \ m \ r)), \text{where}: m = min(r) \end{cases} \end{cases}$$
$$(4.8)$$

When the tree is empty, the result is $\varnothing$; otherwise, we compare x and k. If $x < k$, we recursively delete from left; otherwise right. Because the result may contain doubly-black node, we apply $fixB^2$. When $x = k$, we found the node to cut. If either sub-tree is empty, replace with the none empty one, then shift the blackness if the node was black. If neither sub-tree is empty, cut the minimum $m = \min \ r$ off, and use m to replace k. To reserve the blackness, $shiftB$ makes a black node doubly-black, and forces it black otherwise. It flips doubly-black to normal black when applied twice.

$$\begin{aligned} shiftB \ (\mathcal{B}, l, k, r) &= (\mathcal{B}^2, l, k, r) \\ shiftB \ (\mathcal{C}, l, k, r) &= (\mathcal{B}, l, k, r) \\ shiftB \ \varnothing &= \boldsymbol{\varnothing} \\ shiftB \ \boldsymbol{\varnothing} &= \varnothing \end{aligned} \qquad (4.9)$$

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

```
delete x = makeBlack ∘ (del x) where
    del x Empty = Empty
    del x (Node color l k r)
        | x < k = fixDB color (del x l) k r
        | x > k = fixDB color l k (del x r)
        | isEmpty l = if color == B then shiftBlack r else r
        | isEmpty r = if color == B then shiftBlack l else l
        | otherwise = fixDB color l m (del m r) where m = min r
    makeBlack (Node _ l k r) = Node B l k r
```

```
    makeBlack _ = Empty

isEmpty Empty = True
isEmpty _ = False

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 by rotation and re-coloring. The doubly-black node can be a branch or empty $\varnothing$. 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 transform to the same pattern, as shown in fig. A.1.
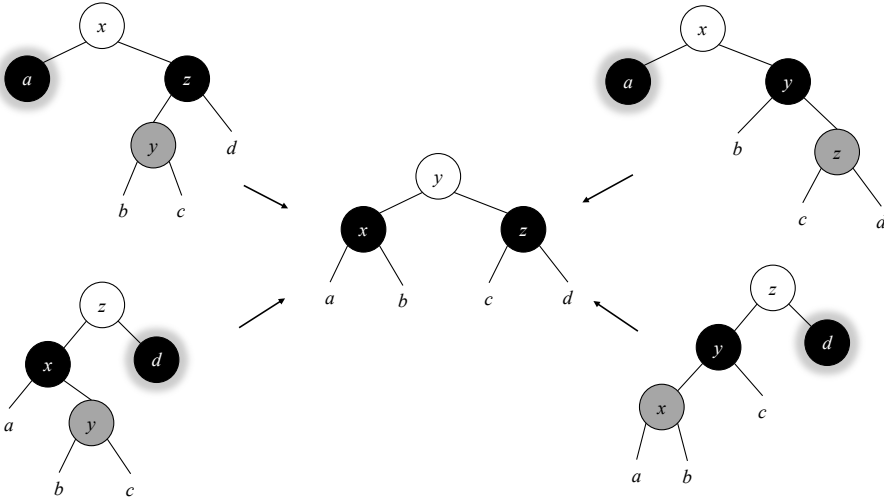
Figure 4.8: Transform 4 sub-cases to the same pattern

$$
\begin{aligned}
fixB^2\ \mathcal{C}\ a_{\mathcal{B}^2}\ x\ (\mathcal{B}, (\mathcal{R}, b, y, c), z, d) &= (\mathcal{C}, (\mathcal{B}, shiftB(a), x, b), y, (\mathcal{B}, c, z, d)) \\
fixB^2\ \mathcal{C}\ a_{\mathcal{B}^2}\ x\ (\mathcal{B}, b, y, (\mathcal{R}, c, z, d)) &= (\mathcal{C}, (\mathcal{B}, shiftB(a), x, b), y, (\mathcal{B}, c, z, d)) \\
fixB^2\ \mathcal{C}\ (\mathcal{B}, a, x, (\mathcal{R}, b, y, c))\ z\ d_{\mathcal{B}^2} &= (\mathcal{C}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, shiftB(d))) \\
fixB^2\ \mathcal{C}\ (\mathcal{B}, (\mathcal{R}, a, x, b), y, c)\ z\ d_{\mathcal{B}^2} &= (\mathcal{C}, (\mathcal{B}, a, x, b), y, (\mathcal{B}, c, z, shiftB(d)))
\end{aligned}
$$
$$(4.10)$$

Where $a_{\mathcal{B}^2}$ means node a is doubly-black.

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 fig. A.2. We add this fixing as additional 2 rows in eq. (4.10):

$$
\begin{aligned}
&\cdots \\
fixB^2\ \mathcal{B}\ a_{\mathcal{B}^2}\ x\ (\mathcal{R}, b, y, c) &= fixB^2\ \mathcal{B}\ (fixB^2\ \mathcal{R}\ a\ x\ b)\ y\ c \\
fixB^2\ \mathcal{B}\ (\mathcal{R}, a, x, b)\ y\ c_{\mathcal{B}^2} &= fixB^2\ \mathcal{B}\ a\ x\ (fixB^2\ \mathcal{R}\ b\ y\ c)
\end{aligned}
$$
$$(4.11)$$

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 fig. 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 eq. (4.11):
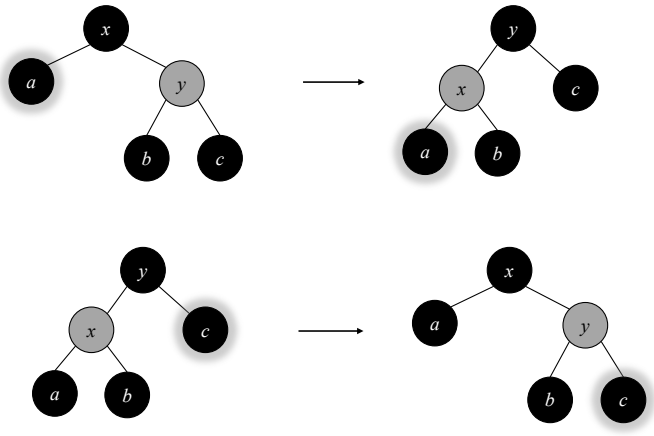
Figure 4.9: The sibling of the doubly-black is red.
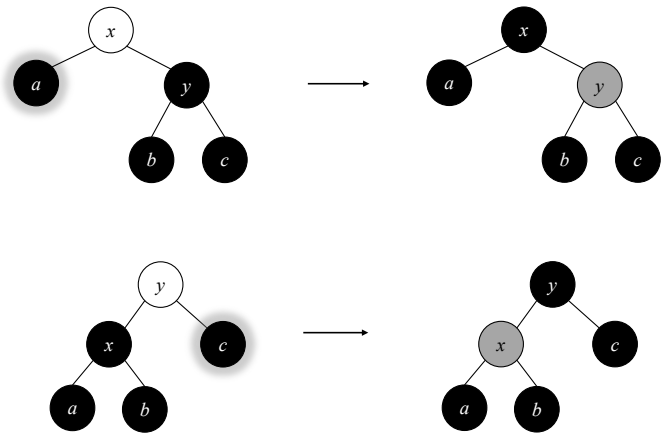
Figure 4.10: move the blackness up.

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

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:

```
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)
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)
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)
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)
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

4.5.1. Implement the 'mark-rebuild' delete algorithm: mark the node as deleted without actually removing it. When the marked nodes exceed 50%, rebuild the tree.

4.5   Imperative red-black tree⋆

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:       $root \leftarrow T$

```
 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 ← LEFT-ROTATE(T, GRAND-PARENT(x))
23:     COLOR(T) ← 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, as shown in Figure 4.11 and fig. 4.7. There is a bit performance overhead in the pattern matching algorithm.

Okasaki discussed the difference in detail in[13].
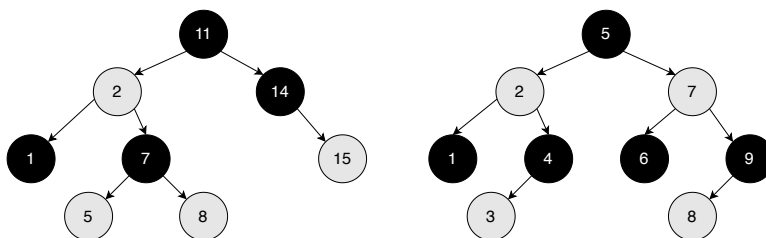
Figure 4.11: Red-black trees created by imperative algorithm.

We provide the imperative delete algorithm in Appendix A of the book. Red-black tree is a popular self-balancing binary search tree. It is a good start for more complex data structures. If extend from 2 to k sub-trees and maintain the balance, we obtain B-tree; If store the data along with the edge but not in node, we obtain the 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 implement AVL tree and Splay tree based on pattern matching in this book.

4.6   Appendix: Example programs

Definition of red-black tree node with parent reference. Set the color red by default.

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

The idea of red-black tree is to limit the number of 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 of height h has $n = 2^h - 1$ nodes. There are not any empty branches except for 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.

5.1 Definition
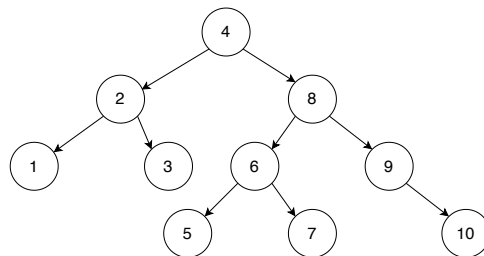
Figure 5.1: an AVL tree

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 of $\delta(T)$: ± 1, and 0. Figure 5.1 shows an AVL tree. This definition ensures the tree height $h = O(\lg n)$, where n is the number of nodes. 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:
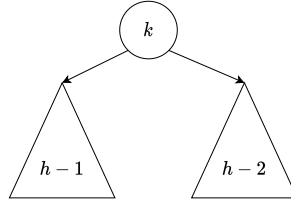
Figure 5.2: An AVL tree of height h. The height of one sub-tree is $h - 1$, the other is no less than $h - 2$.

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

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 5.1.1. *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 $N'(h) \geq \phi^h$.

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

$\square$

From lemma 5.1.1, we immediately obtain:

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

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 simplify the

implementation with pattern matching. The idea is similar to the functional red-black tree[13]. Because of this 'modify-fix' approach, AVL tree is also self-balancing. We can re-use the binary search tree definition. Although the balance factor δ can be computed recursively, we record it inside each node as $T = (l, k, r, \delta)$, and update it when mutate the tree[1]. Below example program adds δ as an `Int`:

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

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

5.2 Insert

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

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

Where $fst\ (a, b) = a$ returns the first component of a pair. $ins\ x\ T$ inserts element x into tree T:

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

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

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

We can further break it down into 4 cases:

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

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

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

---

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

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

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

Below example programs implements what we deduced so far:

```
insert x  = fst ∘ ins x where
    ins x Empty = (Br Empty x Empty 0, 1)
    ins x (Br l k r d)
        | x < k = tree (ins x l) k (r, 0) d
        | x > k = tree (l, 0) k (ins x 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
```

5.2.1   Balance

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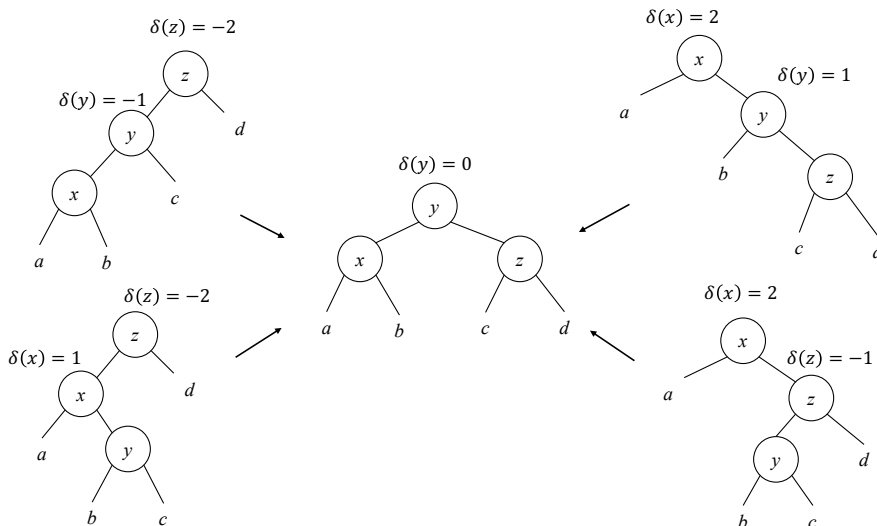
Figure 5.3: Fix 4 cases to the same structure

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

$$
\begin{array}{cc}
\text{Left-left} & \text{Right-right} \\
\begin{cases} \delta'(x) &= \delta(x) \\ \delta'(y) &= 0 \\ \delta'(z) &= 0 \end{cases} &
\begin{cases} \delta'(x) &= 0 \\ \delta'(y) &= 0 \\ \delta'(z) &= \delta(z) \end{cases}
\end{array} \qquad (5.14)
$$

Right-left and Left-right are same:

$$\delta'(x) = \begin{cases} \delta(y) = 1 : & -1 \\ otherwise : & 0 \end{cases}$$
$$\delta'(y) = 0 \qquad\qquad (5.15)$$
$$\delta'(z) = \begin{cases} \delta(y) = -1 : & 1 \\ otherwise : & 0 \end{cases}$$

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

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

$$(5.16)$$

Where $\delta'(x)$ and $\delta'(z)$ are defined in eq. (B.17). If none of the pattern matches, keep 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 eq. (5.7), it is bound to is $O(\lg n)$.

5.2.2 Verification

To validate an AVL tree, we need verify two things: (1) It is a binary search tree; (2) For every sub-tree T, eq. (5.2): $\delta(T) \leq 1$ holds. Below function examines the height difference between the two sub-trees recursively:

$$
\begin{array}{rcl}
avl?\ \varnothing & = & True \\
avl?\ T & = & avl?\ l\ \text{and}\ avl?\ r\ \text{and}\ \delta = |r| - |l|\ \text{and}\ |\delta| \leq 1
\end{array}
$$

$$(5.17)$$

Where the height is calculated recursively:

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

$$(5.18)$$

Below example program implements AVL tree height verification:

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

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

Exercise 5.1

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

5.3   Imperative 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:     $root \leftarrow T$
3:     $x \leftarrow$ CREATE-LEAF(k)
4:     $\delta(x) \leftarrow 0$
5:     $parent \leftarrow$ NIL
6:     **while** $T \neq$ NIL **do**
7:         $parent \leftarrow T$
8:         **if** $k <$ KEY(T) **then**
9:             $T \leftarrow$ LEFT(T)
10:         **else**
11:             $T \leftarrow$ RIGHT(T)
12:     PARENT(x) $\leftarrow parent$
13:     **if** $parent =$ NIL **then**   ▷ tree T is empty
14:         **return** x
15:     **else if** $k <$ KEY($parent$) **then**
16:         LEFT($parent$) $\leftarrow x$
17:     **else**
18:         RIGHT($parent$) $\leftarrow x$
19:     **return** AVL-INSERT-FIX($root, x$)

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

- $|\delta| = 1$, $|\delta'| = 0$. The new node makes the tree well balanced. The height of the parent keeps unchanged.

- $|\delta| = 0$, $|\delta'| = 1$. Either the left or the right sub-tree increases its height. We need go on checking the upper level.

- $|\delta| = 1$, $|\delta'| = 2$. We need rotate the tree to fix the balance factor.

1: **function** AVL-INSERT-FIX(T, x)
2:     **while** PARENT(x) $\neq$ NIL **do**
3:         $P \leftarrow$ PARENT(x)
4:         $L \leftarrow$ LEFT(x)
5:         $R \leftarrow$ RIGHT(x)
6:         $\delta \leftarrow \delta(P)$
7:         **if** $x =$ LEFT(P) **then**
8:             $\delta' \leftarrow \delta - 1$
9:         **else**
10:             $\delta' \leftarrow \delta + 1$
11:         $\delta(P) \leftarrow \delta'$

```
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 eq. (B.6)
20:                     δ(R) ← 0
21:                     T ← LEFT-ROTATE(T, P)
22:                 if δ(R) = −1 then                             ▷ Right-left
23:                     δy ← δ(LEFT(R))                           ▷ By eq. (B.17)
24:                     if δy = 1 then
25:                         δ(P) ← −1
26:                     else
27:                         δ(P) ← 0
28:                     δ(LEFT(R)) ← 0
29:                     if δy = −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:                     δy ← δ(RIGHT(L))
42:                     if δy = 1 then
43:                         δ(L) ← −1
44:                     else
45:                         δ(L) ← 0
46:                     δ(RIGHT(L)) ← 0
47:                     if δy = −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 δ for the impacted nodes. The right-right and left-left cases need one rotation, while the right-left and left-right case need two rotations. Appendix B provides the delete implementation.

AVL tree was developed in 1962 (earlier than the red-black tree) by Adelson-Velskii and Landis[18],[19]. It is named after the two authors. Most tree operations are bound $O(\lg n)$ time. From eq. (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.

5.4   Appendix: Example programs

Definition of AVL tree node.

```
data Node<T> {
    int delta
    T key
    Node<T> left, right, 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].
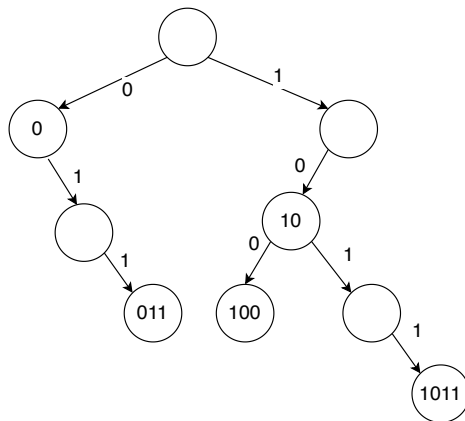
Figure 6.1: Radix tree.

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 finish all the n bits. We needn't store keys in node because they are represented by edges. The nodes labelled with key in fig. 6.1 are for illustration purpose. For the integer keys, we represent them in binary format, and implement lookup with bit-wise manipulations.

6.1 Integer trie

We call the data structure in fig. 6.1 *binary trie*, which was developed by Edward Fredkin in 1960. The name 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, we treat trie and prefix tree differently in this book. 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 fig. 6.2. The three keys of "11", "011", and "0011" all equal to 3.
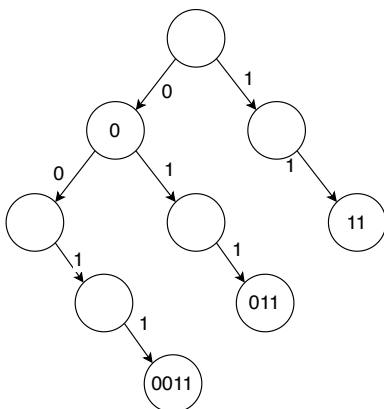
Figure 6.2: A big-endian trie.

It is inefficient to treat the prefix zeros as valid bits. We need a tree of 32 levels to insert 1 as an integer of 32 bits. Okasaki suggests 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

Re-use binary tree definition, a node is either empty, or a branch containing the left, right sub-trees, and an optional value v. The left sub-tree l is encoded as 0 and the right r is encoded as 1.

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

Given a node, the corresponding integer key 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 tree is $IntTrie\ A$ where A is the type of the value.

6.1.2 Insert

When insert a key k and a value x, we convert the integer k into binary. If k is even, the lowest bit is 0, we recursively insert to the left sub-tree; otherwise k is odd, the lowest bit is 1, we insert to the right. Next divide k by 2 to remove the lowest bit. For none empty trie $T = (l, v, r)$, function $insert$ is defined as below:

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

If $k = 0$, we put x in the node. This algorithm overrides the value if k already exists. Alternatively, we can store a list, and append x to it. 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 $\varnothing$. Figure 6.3 shows an example trie, generated with key-value pairs of $\{1 \rightarrow a, 4 \rightarrow b, 5 \rightarrow c, 9 \rightarrow d\}$. Below example program implements $insert$:
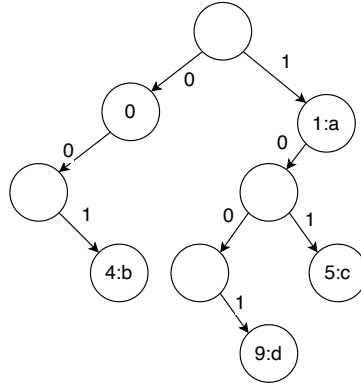
Figure 6.3: A little-endian integer binary trie of $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$.

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

We can test even/odd with the remainder modulo 2: $even(k) = (k \bmod 2 = 0)$m or use bit-wise operation, like (k & 0x1) == 0. We can eliminate the recursion through loops to realize an iterative implementation:

1: **function** INSERT(T, k, x)
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) $\leftarrow x$
16:    **return** T

INSERT takes a trie T, a key k, and a value x. For integer k of m bits, it goes m levels. The performance is bound to $O(m)$. We design *insert k x T* and INSERT(T, k, x) symmetric, apply *foldr* to the former, and *foldl* (or for-loops) to the latter to convert a list of key-value pairs to tree. For example:

$$fromList = foldr \ (uncurry \ insert) \ \varnothing \qquad\qquad (6.2)$$

The usage is *fromList* [(1, a), (4, b), (5, c), (9, d)], where *uncurry* is the revert of Currying, it unpack a pair and feed to *insert*:

$$uncurry \ f \ (a, b) = f \ a \ b \qquad\qquad (6.3)$$

6.1.3   Lookup

When lookup key k, if $k = 0$, then the root is the target. Otherwise, we check the lowest bit, then recursively look up the left or right sub-tree accordingly.

$$
\begin{aligned}
lookup\ k\ \varnothing &= Nothing \\
lookup\ 0\ (l, v, r) &= v \\
lookup\ k\ (l, v, r) &= \begin{cases} even(k): & lookup\ \dfrac{k}{2}\ l \\ odd(k): & lookup\ \lfloor\dfrac{k}{2}\rfloor\ r \end{cases}
\end{aligned} \tag{6.4}
$$

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

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

6.1.1. Can we change the definition from `Branch (IntTrie a) (Maybe a) (Int-Trie 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 fig. 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. 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. Consolidate the chained nodes in fig. 6.3, gives the integer tree in fig. 6.4. The key of 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.

6.2.1   Definition

Integer prefix tree is a special binary tree. It is empty $\varnothing$, or a leaf of (k, v), that 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. Denoted as (p, m, l, r). Below example program defines the integer prefix tree. The branch node contains 4 components: The longest prefix p, a mask integer m indicating from which bit the sub-trees branch out, the left and right sub-trees l and r.
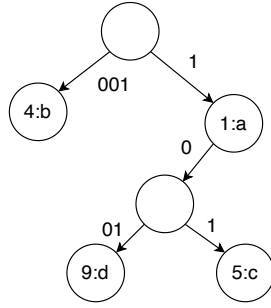
Figure 6.4: Little endian integer tree of the map $\{1 \rightarrow a, 4 \rightarrow b, 5 \rightarrow c, 9 \rightarrow d\}$.

The mask $m = 2^n$ for some integer $n \geq 0$. All bits below 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, we create a new leaf of y, and a branch, set x and y as the two sub-trees. To determine which one is on the left between x and y, we 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 in fig. 6.5.
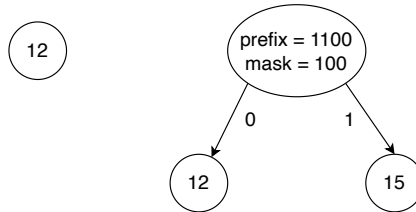
Figure 6.5: Left: T is a leaf of 12; Right: After insert 15.

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 to p. For example, when insert $y = 14 = (1110)_2$ to the tree in fig. 6.5, since $p = (11oo)_2$ and the next bit (the bit of 2^1) is 1, we recursively insert y to the right sub-tree. If y does not match p in the root, we need branch a new leaf as shown in fig. 6.6.

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

$$(6.5)$$
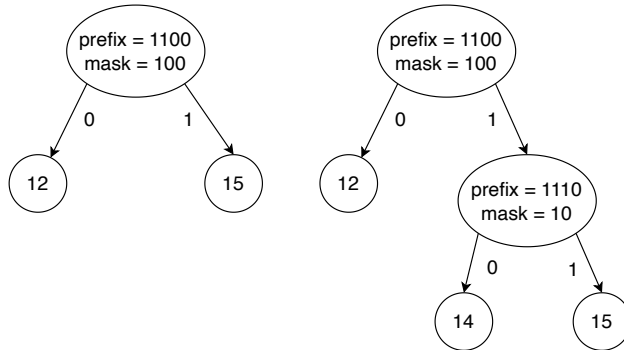
(a) Insert $14 = (1110)_2$, which matches $p = (1100)_2$, to right.
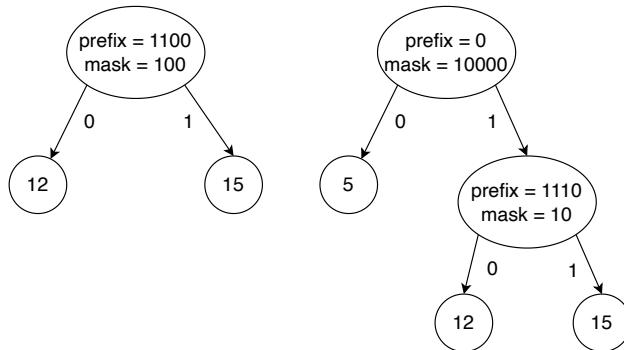
(b) Insert $5 = (101)_2$, which does not match $p = (1100)_2$. Branch out a new leaf.

Figure 6.6: Insert to a branch.

We create a leaf of (k, v) when $T = \varnothing$, override the value for the same key. Function $match(k, p, m)$ tests if k and 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:

$$zero(k, m) = x\&(m \gg 1) \tag{6.6}$$

Function $join(p_1, T_1, p_2, T_2)$ takes two 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 sub-trees:

$$join(p_1, T_1, p_2, T_2) = \begin{cases} zero(p_1, m): & (p, m, T_1, T_2) \\ otherwise: & (p, m, T_2, T_1) \end{cases} \tag{6.7}$$

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

$$\begin{aligned} highest(0) &= 0 \\ highest(n) &= 1 + highest(n \gg 1) \end{aligned}$$

Then generate a mask $m = 2^h$. 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 k x t
  = 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 k x l) r
                        else Branch p m l (insert k x r)
        | 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 tree. Although the integer prefix tree consolidates the chained nodes, the operation to extract the longest common prefix need scan the bits. For integer of m bits, it is bound to $O(m)$.
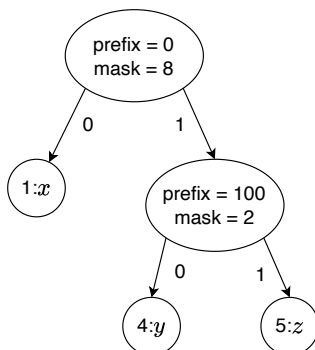
Figure 6.7: Insert $\{1 \to x, 4 \to y, 5 \to z\}$ to the big-endian integer tree.

6.2.3 Lookup

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

$$
\begin{aligned}
lookup\ k\ \varnothing &= \quad Nothing \\
lookup\ k\ (k', v) &= \quad \begin{cases} k = k' : & Just\ v \\ otherwise : & Nothing \end{cases} \\
lookup\ k\ (p, m, l, r) &= \quad \begin{cases} match(k, p, m) : & \begin{cases} zero(k, m) : & lookup\ k\ l \\ otherwise : & lookup\ k\ r \end{cases} \\ otherwise : & Nothing \end{cases}
\end{aligned}
\tag{6.8}
$$

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

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

6.2.1. Implement the *lookup* function for integer tree.

6.2.2. 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

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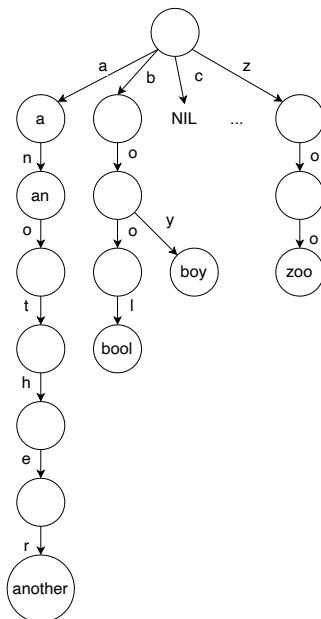
Figure 6.8: A trie of 26 branches, containing key 'a', 'an', 'another', 'bool', 'boy', and 'zoo'.

Not all the 26 sub-trees contain data. In fig. 6.8, there are only three none empty sub-trees bound to 'a', 'b', and 'z'. Other sub-trees, such as 'c', are empty. We will hide them later. When it is case sensitive, or extent the key from alphabetic string to generic list, we adopt collection types, like map to define trie. A trie of type $Trie\ K\ V$ is either empty $\varnothing$ or a node of 2 cases:

1. A leaf of value v without any sub-trees as $(v, \varnothing)$, where the type of v is V;

2. A branch, containing a value v and multiple sub-trees. Each sub-tree is bound to an element k of type K. Denoted as (v, ts), where $ts = \{k_1 \mapsto T_1, k_2 \mapsto T_2, ..., k_m \mapsto T_m\}$, contains the mapping from k_i to sub-tree T_i. The mapping can be assoc list or self-balancing trees (see chapter 4, 5).

Let the empty content be $(Nothing, \varnothing)$, Below example program defines trie.

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

6.3.1   Insert

When insert a pair of key and value, where the key is a list of elements. Let the trie be $T = (v, ts)$, $ts[k]$ looks up k in map ts. It returns empty when k doesn't exist; $ts[k] \leftarrow t$ inserts a mapping from k to tree t, and returns the updated map.

$$
\begin{array}{rcl}
insert \ [\ ] \ v \ (v', ts) & = & (Just \ v, ts) \\
insert \ (k{:}ks) \ v \ (v', ts) & = & (v', ts[k] \leftarrow insert \ ks \ v \ ts[k])
\end{array} \tag{6.9}
$$

Below is the example program:

```
insert [] x (Trie _ ts) = Trie (Just x) ts
insert (k:ks) x (Trie v ts) = Trie v (Map.insert k (insert ks x t) ts) where
    t = case Map.lookup k ts of
          Nothing → Trie Nothing Map.empty
          (Just t) → t
```

We can eliminate the recursion with loops:

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 $O(n \lg m)$. For lower case English strings, $m = 26$, the insert operation is proportion to the length of the key string.

6.3.2 Lookup

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:

$$
\begin{array}{rcl}
lookup \ [\ ] \ (v, ts) & = & v \\
lookup \ (k{:}ks) \ (v, ts) & = & \begin{cases} ts[k] = Nothing: & Nothing \\ ts[k] = Just \ t: & lookup \ ks \ t \end{cases}
\end{array} \tag{6.10}
$$

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

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

6.4 Prefix tree

Trie is not space efficient. We consolidate the chained nodes to obtain the prefix tree. 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, 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 $s + s_1$, $s + s_2$, ... For any $i \neq j$, list s_i and s_j don't have none empty common prefix. Consolidate the chained nodes in fig. 6.8, we get the prefix tree in fig. 6.9.
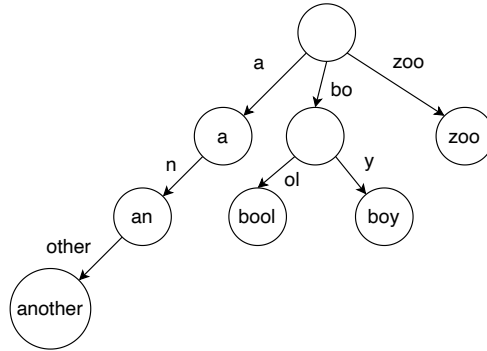
Figure 6.9: A prefix tree with keys: 'a', 'an', 'another', 'bool', 'boy', 'zoo'.

Below example program defines the prefix tree[1]:

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

Denote prefix tree $t = (v, ts)$. Particularly, $(Nothing, [\,])$ is the empty node, and $(Just\ v, [\,])$ is the leaf of v.

6.4.1   Insert

When insert key s, if the tree is empty, we create a leaf of s as fig. 6.10 (a); otherwise, if there is none empty 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 fig. 6.10 (c) $\to$ (e); or s_i is the prefix of s as shown in fig. 6.10 (d) $\to$ (e).

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

$$
\begin{array}{rcl}
insert\ [\,]\ v\ (v', ts) & = & (Just\ v, ts) \\
insert\ s\ v\ (v', ts) & = & (v', ins\ ts)
\end{array}
\tag{6.11}
$$

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

$$
\begin{array}{rcl}
ins\ [\,] & = & [s \mapsto (Just\ v, [\,])] \\
ins\ ((s' \mapsto t){:}ts') & = & \begin{cases} match\ s\ s' : & (branch\ s\ v\ s'\ t) : ts' \\ otherwise : & (s' \mapsto t) : ins\ ts' \end{cases}
\end{array}
\tag{6.12}
$$

If the node hasn't any sub-trees, we create a leaf of v as the only sub-tree, and map s to it; otherwise, for each 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:

---

[1]Alternatively, we can use `Map [k] PrefixTree k v` to manage the sub-trees.
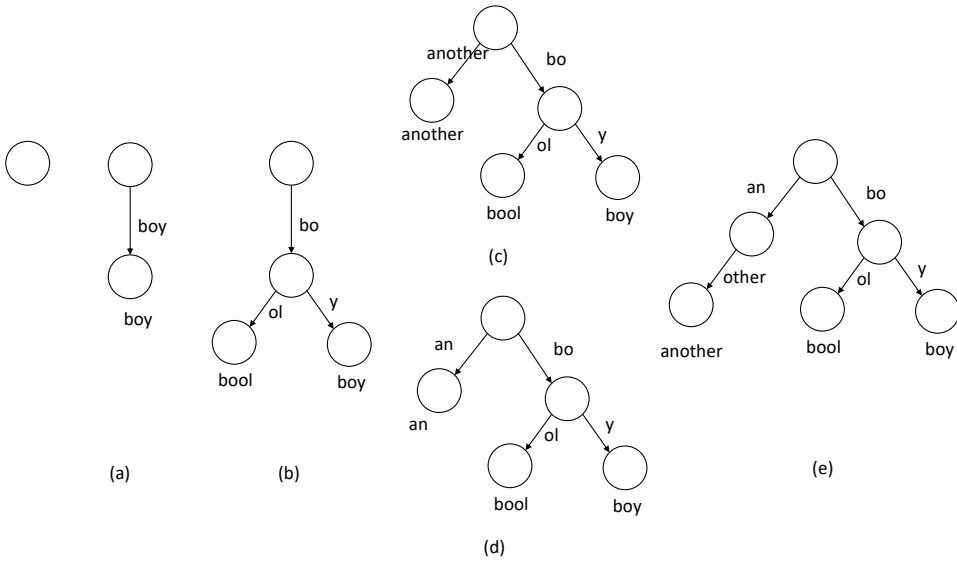
Figure 6.10: (a) insert 'boy' to empty tree; (b) insert 'bool', branch a new node out; (c) insert 'another' to (b); (d) insert 'an' to (b); (e) insert 'an' to (c), same result as insert 'another' to (d)

$$
\begin{array}{rcl}
match\ [\ ]\ B & = & True \\
match\ A\ [\ ] & = & True \\
match\ (a{:}as)\ (b{:}bs) & = & a = b
\end{array}
\tag{6.13}
$$

To extract the longest common prefix of A and B, define $(C, A', B') = lcp\ A\ B$, where $C + A' = A$ and $C + B' = B$. If either A or B is empty, or their first elements are different, then the common prefix $C = [\ ]$; otherwise, we recursively extract the longest common prefix from the rest, and preprend the head:

$$
\begin{array}{rcl}
lcp\ [\ ]\ B & = & ([\ ], [\ ], B) \\
lcp\ A\ [\ ] & = & ([\ ], A, [\ ]) \\
lcp\ (a{:}as)\ (b{:}bs) & = & \begin{cases} a \neq b: & ([\ ], a{:}as, b{:}bs) \\ \text{otherwise}: & (a{:}cs, as', bs'), \text{where } (cs, as', bs') = lcp\ as\ bs \end{cases}
\end{array}
\tag{6.14}
$$

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:

$$
\begin{array}{l}
branch\ A\ v\ B\ t = \\
lcp\ A\ B = \begin{cases} (C, [\ ], B'): & (C, (Just\ v, [B' \mapsto t])) \\ (C, A', [\ ]): & (C, insert\ A'\ v\ t) \\ (C, A', B'): & (C, (Nothing, [A' \mapsto (Just\ v, [\ ]), B' \mapsto t])) \end{cases}
\end{array}
\tag{6.15}
$$

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 only 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 of v, put it together with t as the two sub-trees. The following example program implements the *insert* algorithm:

```
insert [] v (PrefixTree _ ts) = PrefixTree (Just v) ts
insert k v (PrefixTree v' ts) = 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 a' v t)
  (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, it is still bound to $O(mn)$, where n is the length of the key, and m is the size of the element set.

6.4.2 Lookup

When look up a key k, we start from the root. If $k = [\ ]$, then return the root value; otherwise, 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 does not exist s_i as the prefix of k, then there is no such key in the tree.

$$\begin{aligned} lookup\ [\ ]\ (v, ts) &= v \\ lookup\ k\ (v, ts) &= find\ ((s, t) \mapsto s \sqsubseteq k)\ ts = \\ &\quad \begin{cases} Nothing : & Nothing \\ Just\ (s, t) : & lookup\ (k - s)\ t \end{cases} \end{aligned} \tag{6.16}$$

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 lookup.

```
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 leave the imperative implementation as the exercise.

Exercise 6.3

6.3.1. 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 to develop a smart dictionary that can populate candidate words, and realize the textonym input method. Different from the commercial software, we give the examples to illustrate the ideas of trie and prefix tree.

6.5.1   Dictionary and input completion

As shown in fig. 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.
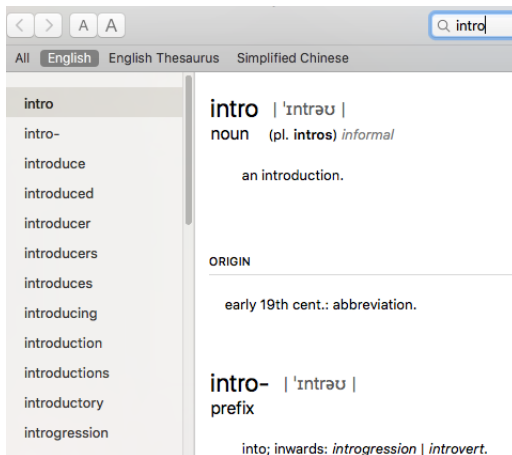
Figure 6.11: A dictionary application

A dictionary contains 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. 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.
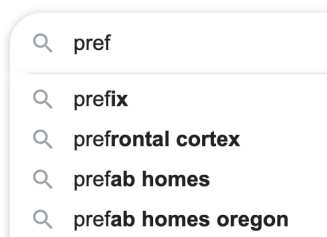
Figure 6.12: A smart text input component

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 we expand all sub-trees till reach to n candidates; otherwise, we locate the sub-tree from the mapped key, and look up recursively. If 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.

$$\begin{aligned}
startsWith\ [\ ]\ (Nothing, ts) &= enum\ ts \\
startsWith\ [\ ]\ (Just\ x, ts) &= ([\ ], x) : enum\ ts \\
startsWith\ s\ (v, ts) &= find\ ((k, t) \mapsto s \sqsubseteq k\ or\ k \sqsubseteq s)\ ts = \\
&\begin{cases} Nothing: & [\ ] \\ Just\ (k, t): & [(k + a, b) | (a, b) \in startsWith\ (s - k)\ t] \end{cases}
\end{aligned} \qquad (6.17)$$

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 $([\ ], x)$ for none empty value x in the root. Function $enum\ ts$ is defined as:

$$enum = concatMap\ (k, t) \mapsto [(k + a, b) | (a, b) \in startsWith\ [\ ]\ t] \qquad (6.18)$$

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 the 'build-foldr' fusion law to eliminate the intermediate list overhead. (chapter 5 of[99]) 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:         $match \leftarrow$ FALSE
7:         **for** $k_i \mapsto T_i$ 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:                 $match \leftarrow$ TRUE
12:                 $k \leftarrow k - k_i$ ▷ drop the prefix
13:                 $T \leftarrow T_i$
14:                 $s \leftarrow s + k_i$
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 14.6.1):

```
 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 kᵢ ↦ Tᵢ in SUB-TREES(T) do
10:             PUSH(Q, (k + kᵢ, Tᵢ))
```

6.5.2 Predictive text input

Before 2010, most mobile phones had a small keypad as shown in fig. 6.13, called ITU-T keypad. It maps a digit to 3 - 4 characters. For example, when input the word 'home', one presses 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]. The commercial implementations use 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:

$$M_{T9} = \{ \quad 2 \mapsto \texttt{"abc"}, 3 \mapsto \texttt{"def"}, 4 \mapsto \texttt{"ghi"},$$
$$5 \mapsto \texttt{"jkl"}, 6 \mapsto \texttt{"mno"}, 7 \mapsto \texttt{"pqrs"}, \qquad (6.19)$$
$$8 \mapsto \texttt{"tuv"}, 9 \mapsto \texttt{"wxyz"} \quad \}$$

$M_{T9}[i]$ gives the corresponding characters for digit i. We can also define the reversed mapping from a character back to digit.

$$M_{T9}^{-1} = concatMap \ ((d, s) \mapsto [(c, d)|c \leftarrow s]) \ M_{T9} \qquad (6.20)$$

Given a string, we can convert it to a sequence of digits by looking up M_{T9}^{-1}.

$$digits(s) = [M_{T9}^{-1}[c]|c \leftarrow s] \qquad (6.21)$$

For any character not in $[\texttt{a..z}]$, we map it to a special key '#'. Below example program defines the 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 builded the prefix tree (v, ts) from the words. We need change the above auto completion algorithm to process digit string ds. For every sub-tree mappings $(s \mapsto t) \in ts$, we convert the prefix s to $digits(s)$, check if it matches to ds (either one is the prefix of the other). There can be multiple sub-trees match ds as:

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

$$\begin{aligned} find_{T9} \ t \ [\ ] &= \ [[\ ]] \\ find_{T9} \ (v, ts) \ ds &= \ concatMap \ find \ pfx \end{aligned} \qquad (6.22)$$

For each mapping (s, t) in pfx, function $find$ recursively looks up the remaining digits ds' in t, where $ds' = drop \ |s| \ ds$, then prepend s to every candidate. However, the length may exceed the number of input digits, we need cut and only take $n = |ds|$ characters:

$$find \ (s, t) = [take \ n \ (s + s_i)|s_i \in find_{T9} \ t \ ds'] \qquad (6.23)$$

The following example program implements the predictive input 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 $(prefix + s, D', T')$, where D' is the remaining digits to be searched. Then add 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** $Q \neq \text{NIL}$ **do**
8:          $(\textit{prefix}, D, T) \leftarrow \text{POP}(Q)$
9:          **for** $(s \mapsto T') \in \text{SUB-TREES}(T)$ **do**
10:             $D' \leftarrow \text{DIGITS}(s)$
11:             **if** $D' \sqsubset D$ **then**   ▷ D' is prefix of D
12:                 $\text{APPEND}(R, (\textit{prefix} + s)[1..n])$ ▷ limit the length to n
13:             **else if** $D \sqsubset D'$ **then**
14:                 $\text{PUSH}(Q, (\textit{prefix} + s, D - D', T'))$
15:     **return** R

We start from integer trie and prefix tree. With binary format, we re-use binary tree to realize the integer based map data structure. Then extend the key from integer to generic list of finite set. Particularly for alphabetic strings, the generic trie and prefix tree are powerful tools to manipulate the text. We give examples about auto-completion and predictive text input. As another instance of radix tree, the *suffix tree*, that is widely used in text/DNA processing, is closely related to trie and prefix tree.

Exercise 6.4

6.4.1. Implement the auto-completion and predictive text input with trie.

6.4.2. How to ensure the candidates in lexicographic order in the auto-completion and predictive text input program? What's the performance change accordingly?

6.6   Appendix: Example programs

Definition of integer binary trie:

```
data IntTrie<T> {
    IntTrie<T> left = null, right = null
    Optional<T> value = Optional.Nothing
}
```

Imperative *insert* with bit-wise operations:

```
IntTrie<T> insert(IntTrie<T> t, Int key,
                  Optional<T> value = Optional.Nothing) {
    if t == null then t = IntTrie<T>()
    p = t
    while key ≠ 0 {
        if key & 1 == 0 {
            p = if p.left == null then IntTrie<T>() else p.left
        } else {
            p = if p.right == null then IntTrie<T>() else p.right
        }
        key = key >> 1
    }
    p.value = Optional.of(value)
    return t
}
```

Definition of integer prefix tree:

```
data IntTree<T> {
    Int key
    T value
    Int prefix = 0, mask = 1
    IntTree<T> left = null, 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:

```
data Trie<K, V> {
  Optional<V> value = Optional.Nothing
  Map<K, Trie<K, V>> subTrees = Map.empty()
}

Trie<K, V> insert(Trie<K, V> t, [K] key, V value) {
    if t == null then t = Trie<K, V>()
    var p = t
```

```
    for c in key {
        if p.subTrees[c] == null then p.subTrees[c] = Trie<K, V>()
        p = p.subTrees[c]
    }
    p.value = Optional.of(value)
    return t
}
```

Definition of Prefix Tree and insert:

```
data PrefixTree<K, V> {
    Optional<V> value = Optional.Nothing
    Map<[K], PrefixTree<K, V>> subTrees = Map.empty()

    Self PrefixTree(V v) {
        value = Optional.of(v)
    }
}

PrefixTree<K, V> insert(PrefixTree<K, V> t, [K] key, V value) {
    if t == null then t = PrefixTree()
    var node = t
    loop {
        bool match = false
        for var (k, tr) in node.subtrees {
            if key == k {
                tr.value = value
                return t
            }
            prefix, k1, k2 = lcp(key, k)
            if prefix ≠ [] {
                match = true
                if k2 == [] {
                    node = tr
                    key = k1
                    break
                } else {
                    node.subtrees[prefix] = branch(k1, PrefixTree(value),
                                                   k2, tr)
                    node.subtrees.delete(k)
                    return t
                }
            }
        }
        if !match {
            node.subtrees[key] = PrefixTree(value)
            break
        }
    }
    return t
}
```

The longest common prefix `lcp` and `branch`.

```
([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",
           '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-balancing[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.
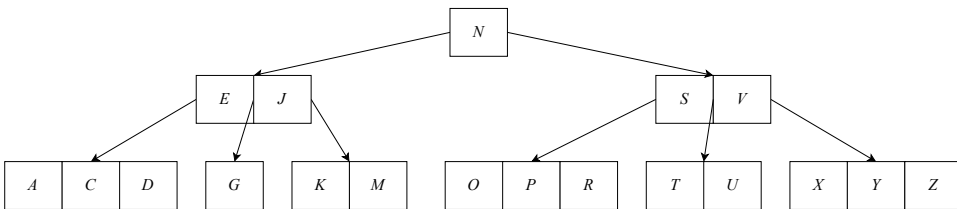
Figure 7.1: A B-Tree

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: a B-tree is either empty or contains n keys and $n+1$ sub-trees, each sub-tree is also a B-tree, denoted as $k_1, k_2, ..., k_n$ and $t_1, t_2, ..., t_n, t_{n+1}$, as shown in fig. 7.2.
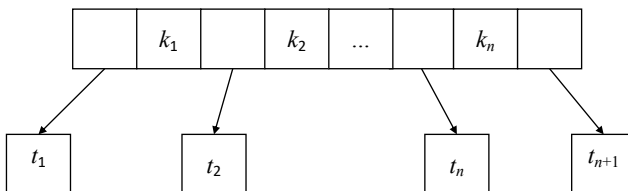
Figure 7.2: A B-tree node

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}:

$$\forall \ x_i \in t_i, i = 0, 1, ..., n \ \Rightarrow x_1 < k_1 < x_2 < k_2 < ... < x_n < k_n < x_{n+1} \qquad (7.2)$$

Leaf node has no sub-tree (accurately, all sub-trees are empty). There can be optional values bound to the keys. We skip the values for simplicity. The type of B-tree is $BTree \ K$ (or `BTree<K>`), where K is the type of keys. On top of it, we also need define a set of self-balancing 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:

$$d - 1 \leq |keys(t)| \leq 2d - 1 \qquad (7.3)$$

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:

$$
\begin{aligned}
n \ &\geq 1 + (d - 1)(2 + 2d + 2d^2 + ... + 2d^{h-1}) \\
&= 1 + 2(d - 1) \sum_{k=0}^{h-1} d^k \\
&= 1 + 2(d - 1) \frac{d^h - 1}{d - 1} \\
&= 2d^h - 1
\end{aligned}
\qquad (7.4)
$$

It limits the tree height with logarithm of the number of keys.

$$h \leq \log_d \frac{n + 1}{2} \qquad (7.5)$$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

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]
```

Let the empty node be $(\varnothing, \varnothing)$ (or `BTree [] []`). Instead of storing d in every node, we pass it together with B-tree t as a pair (d, t).

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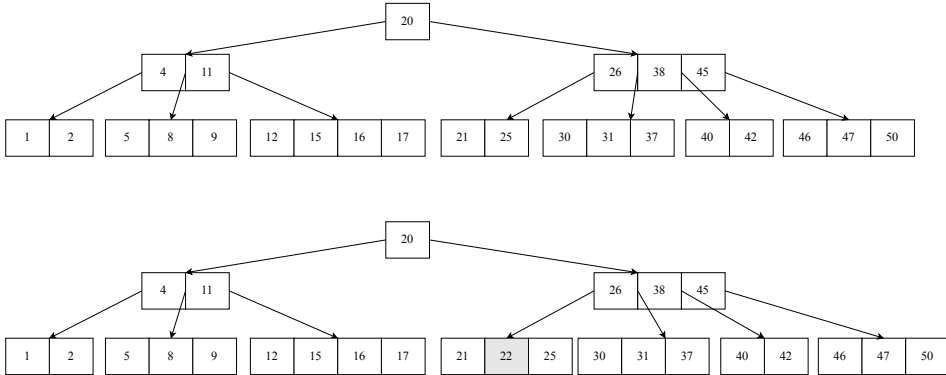
Figure 7.3: Insert 22 to a 2-3-4 tree. $22 > 20$, go to the right sub-tree; next as $22 < 26$, go to the first sub-tree; finally, $21 < 22 < 25$, and the leaf is not full.

For example, consider the 2-3-4 tree in fig. 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 already $2d - 1$ keys in the leaf, we will break the B-tree rules after insert (too 'full'). For the same B-tree in fig. 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 balancing rules. Next, if the new tree violates any rule, we recursively bottom-up split and fix the overly full node. Define a function to test whether a node satisfies the minimum degree constraint:

$$\begin{cases} full\ d\ (ks, ts) & = |ks| > 2d - 1 \\ low\ d\ (ks, ts) & = |ks| < d - 1 \end{cases} \tag{7.6}$$

When the node contains too many keys and sub-trees, define *split* function to break it into 3 parts at position m as shown in fig. 7.4:

$$split\ m\ (ks, ts) = ((ks_l, ts_l), k, (ks_r, ts_r)) \tag{7.7}$$

We reuse the list *splitAt* function in eq. (1.45).

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

---

[1] In fact, it is sufficient to only support less-than and equality. See Exercise 7.1.1.
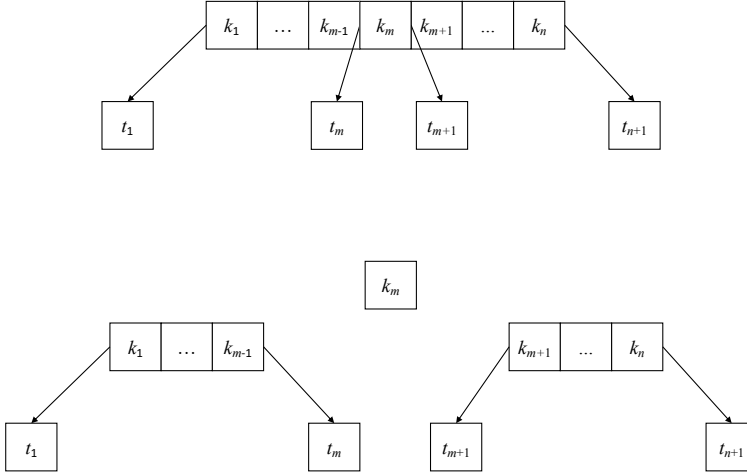
Figure 7.4: Split the node into 3 parts at m

Conversely, *unsplit* combines the 3 parts back into a B-tree node:

$$unsplit\ (ks_l, ts_l)\ k\ (ks_r, ts_r) = (ks_l \mathbin{+\!\!+} [k] \mathbin{+\!\!+} ks_r, ts_l \mathbin{+\!\!+} ts_r) \tag{7.8}$$

We first insert x to the tree t, then call fix to resume the B-tree balancing rules with 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, reuse the list ordered *insert* defined in eq. (1.11); otherwise, we need find the position of the sub-tree where recursively insert to. Define a *partition* 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 calls the list partition function *span* defined in eq. (1.47):

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

As such, we separate all the keys and sub-trees less than x on the left as l, and those greater than x on the right as r. The last sub-tree that less than x is extracted as t'. We then recursively insert x to t', as shown in fig. 7.5.

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

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.
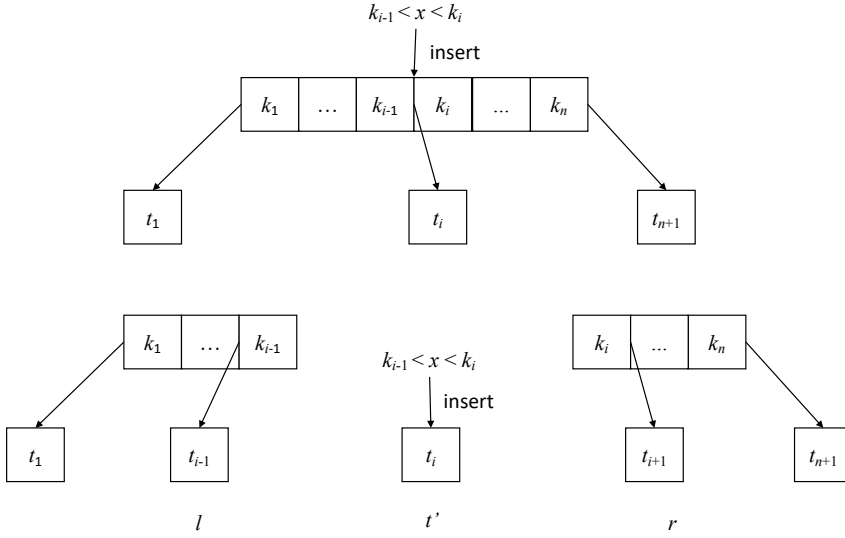
Figure 7.5: partition a node with x

$$balance\ d\ (ks_l, ts_l)\ t\ (ks_r, ts_r) = \begin{cases} full\ d\ t : fix_f \\ otherwise : (ks_l \mathbin{+\!\!+} ks_r, ts_l \mathbin{+\!\!+} [t] \mathbin{+\!\!+} ts_r) \end{cases} \tag{7.13}$$

where fix_f splits sub-tree t as $(t_1, k, t_2) = split\ d\ t$, then combine them to a new 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 from "GMPXACDEJKNORSTUVYZ".
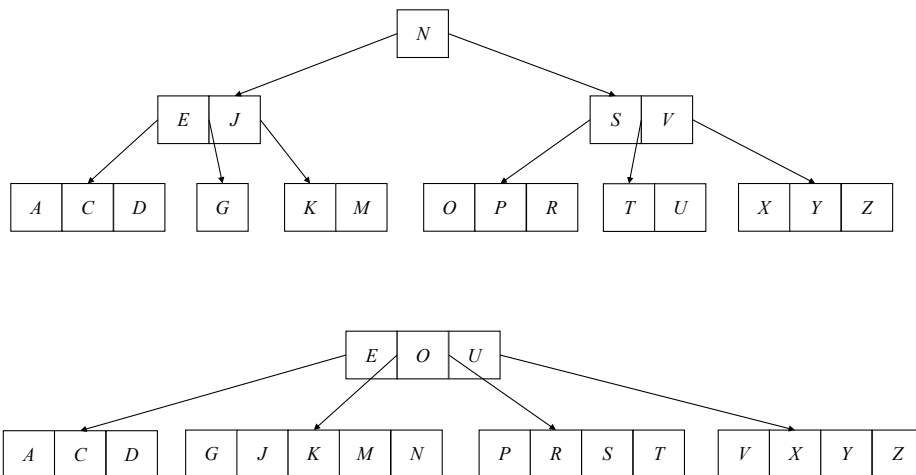
Figure 7.6: Repeatedly insert elements from "GMPXACDEJKNORSTUVYZ". above: $d = 2$ (2-3-4 tree), below: $d = 3$

7.2.2 Split before insert

The second method is to split a node before insertion to prevent it overly full. We often see this method in imperative implementation. 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 fig. 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:      d ← DEG(z)
3:      x ← t_i(z)
4:      y ← CREATE-NODE
5:      K(y) ← [k_{d+1}(x), k_{d+2}(x), ..., k_{2d-1}(x)]
6:      K(x) ← [k_1(x), k_2(x), ..., k_{d-1}(x)]
7:      if x is not leaf then
8:          T(y) ← [t_{d+1}(x), t_{d+2}(x), ..., t_{2d}(x)]
9:          T(x) ← [t_1(x), t_2(x), ..., t_d(x)]
10:     INSERT-AT(K(z), i, k_d(x))
11:     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:      r ← t
3:      if r is full then                                           ▷ root is full
4:          s ← CREATE-NODE
5:          T(s) ← [r]
6:          SPLIT(s, 1)
```

7:              $r \leftarrow s$
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 7.1.3 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:        $n \leftarrow |K(r)|$
3:        **if** r is leaf **then**
4:              $i \leftarrow 1$
5:              **while** $i \leq n$ and $k > k_i(r)$ **do**
6:                    $i \leftarrow i + 1$
7:              INSERT-AT$(K(r), i, k)$
8:        **else**
9:              $i \leftarrow n$
10:            **while** $i > 1$ and $k < k_i(r)$ **do**
11:                  $i \leftarrow i - 1$
12:            **if** $t_i(r)$ is full **then**
13:                  SPLIT(r, i)
14:                  **if** $k > k_i(r)$ **then**
15:                        $i \leftarrow i + 1$
16:            INSERT-NONFULL$(t_i(r), k)$
17:      **return** r

This algorithm is recursive. Exercise 7.1.2 asks to eliminate the recursion with pure loops. Figure 7.7 gives the result with the same input of "GMPXACDEJKNORSTUVYZ".
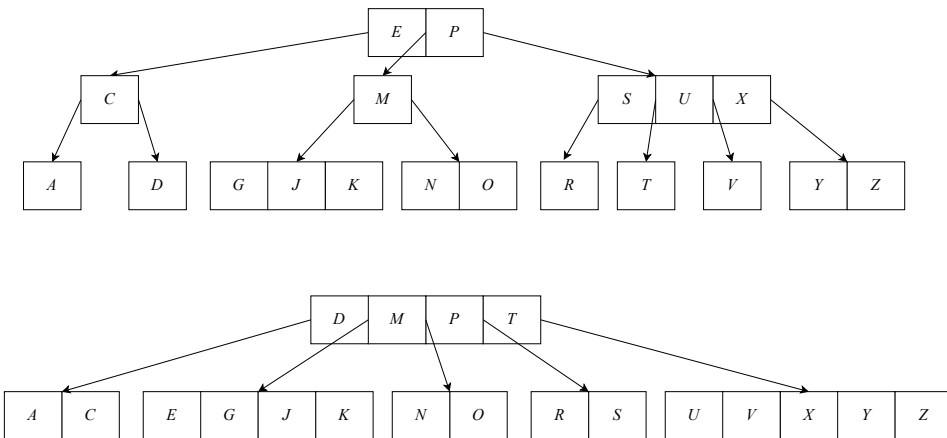
Figure 7.7: Insert from "GMPXACDEJKNORSTUVYZ". up: $d = 2$, 2-3-4 tree; bottom: $d = 3$.

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 fig. 7.8. We can move forward and backward both in constant time.
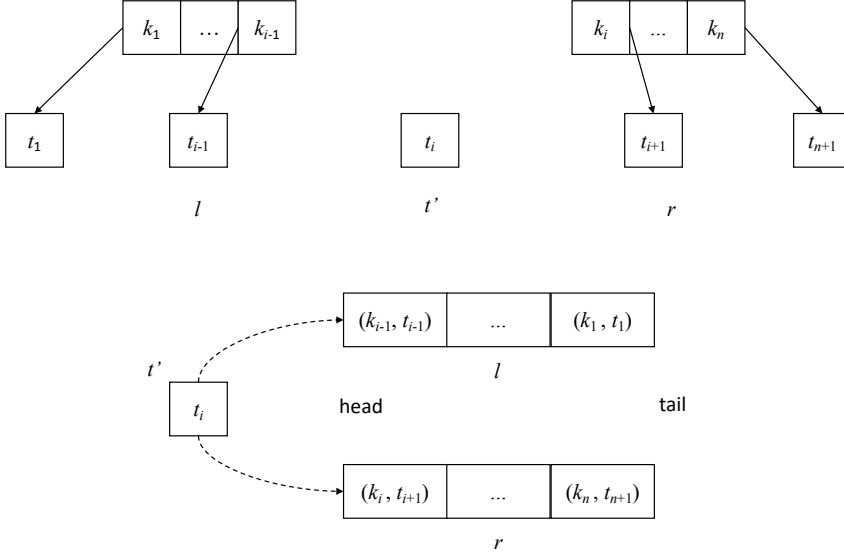
Figure 7.8: Define the B-tree node with a sub-tree and paired lists

Below example program defines B-tree node. It's either empty, or contains 3 parts: the left list of (key, sub-tree) pairs in reversed order, a sub-tree, and the right list of (key, sub-tree) pairs. 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 p t*, that separates the tree t with a 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 $hd = fst \circ head$, picks the first pair (a, b) from a list, then extracts a out.

$$
\begin{array}{rcl}
partition \ p \ (\varnothing, m, r) & = & \begin{cases} p(hd(r)): & partition \ p \ (step_r \ t) \\ otherwise: & (\varnothing, m, r) \end{cases} \\[2ex]
partition \ p \ (l, m, \varnothing) & = & \begin{cases} (not \circ p)(hd(l)): & partition \ p \ (step_l \ t) \\ otherwise: & (l, m, \varnothing) \end{cases} \\[2ex]
partition \ p \ (l, m, r) & = & \begin{cases} p(hd(l)) \text{ and } (not \circ p)(hd(r)): & (l, m, r) \\ p(hd(r)): & partition \ p \ (step_r \ t) \\ (not \circ p)(hd(l)): & partition \ p \ (step_l \ t) \end{cases}
\end{array}
\tag{7.16}
$$

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 use $step_l/step_r$ to split a B-tree at position d when it's 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:

$$\begin{aligned} full\ d\ \varnothing \quad &= False \\ full\ d\ (l, t', r) \quad &= |l| + |r| > 2d - 1 \end{aligned} \tag{7.19}$$

and

$$\begin{aligned} low\ d\ \varnothing \quad &= False \\ low\ d\ (l, t', r) \quad &= |l| + |r| < d - 1 \end{aligned} \tag{7.20}$$

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:

$$\begin{aligned} ins\ \varnothing \quad &= \quad (\varnothing, \varnothing, [(x, \varnothing)]) \\ ins\ t \quad &= \quad \begin{cases} t' = \varnothing: & balance\ d\ l\ \varnothing\ ((x, \varnothing){:}r) \\ t' \neq \varnothing: & balance\ d\ l\ (ins\ t')\ r \end{cases} \end{aligned} \tag{7.23}$$

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

7.1.1. Can we use $\leq$ to support duplicated keys in B-Tree?

7.1.2. For the 'split then insert' algorithm, eliminate the recursion with loops.

7.1.3. 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   Lookup

For lookup, we extend from the binary search tree to multiple branches, and obtain the generic B-tree lookup solution. There are only two directions when lookup the binary search tree: left and right, while, there are multiple ways in B-tree. Consider lookup k in B-tree $t = (ks, ts)$, if t is a leaf (ts is empty), then the problem becomes list lookup; otherwise, we partition 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'.

$$
\begin{aligned}
lookup\ k\ (ks, \varnothing) &= \begin{cases} k \in ks: & Just\ (ks, \varnothing) \\ otherwise: & Nothing \end{cases} \\
lookup\ k\ (ks, ts) &= \begin{cases} Just\ k = safeHd\ ks_r: & Just\ (ks, ts) \\ otherwise: & lookup\ k\ t' \end{cases}
\end{aligned} \tag{7.25}
$$

Where $((ks_l, ts_l), t', (ks_r, ts_r)) = partition\ k\ t$, and

$$
\begin{aligned}
safeHd\ [] &= Nothing \\
safeHd\ (x{:}xs) &= Just\ x
\end{aligned}
$$

Below example program[2] implements *lookup*.

---

[2] safeHd is provided as listToMaybe in some library.

```
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:

$$lookup\ k\ \varnothing\ =\ Nothing$$
$$lookup\ k\ t\ =\ \begin{cases} Just\ k = safeFst\ (safeHd\ r): & Just\ (l, t', r) \\ otherwise: & lookup\ k\ t' \end{cases} \quad (7.26)$$

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 $t_i(r)$, and go on looking up. If r is a leaf and k is not in the keys, then return nothing as k does not exist in the tree.

1: **function** LOOKUP(r, k)
2:     **loop**
3:         $i \leftarrow 1, n \leftarrow |K(r)|$
4:         **while** $i \leq n$ and $k > k_i(r)$ **do**
5:             $i \leftarrow i + 1$
6:         **if** $i \leq n$ 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:           $r \leftarrow t_i(r)$ ▷ go to the i-th sub-tree

Exercise 7.2

7.2.1. Improve the imperative lookup with binary search among keys.

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 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 from binary search tree to multiple branches, then fix the B-tree balancing rules. The main program is defined with two steps:

$$delete\ x\ (d, t) = fix\ (d, del\ x\ t) \quad (7.27)$$

Where *del* is the function 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 $(\geq x)$. When r isn't empty, we pick the first key k_i from it. If the key $k_i = x$, we next replace it with the maximum key $k' = max(t')$, and recursively delete k' from t' as shown in fig. 7.9. Otherwise (either r is empty, or $k_i \neq x$), we recursively delete x from sub-tree t'.
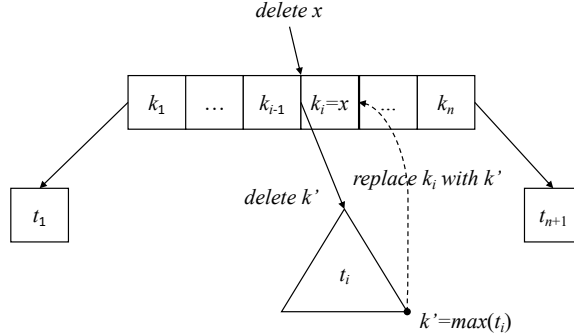
Figure 7.9: Replace k_i with $k' = max(t')$, then recursively delete k' from t'.

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

$$
\begin{aligned}
max \ (ks, \varnothing) \ & = \ last \ ks \\
max \ (ks, ts) \ & = \ max \ (last \ ts)
\end{aligned}
$$
(7.29)

Function *last* returns the last element from a list (eq. (1.4)). $delete_l$ is the list delete algorithm (eq. (1.14)). *tail* drops the first element from a list and returns the rest (eq. (1.1)). We need modify the *balance* function, which defined for *insert* before, with the additional logic to merge the node if it contains too few keys.

$$
balance \ d \ (ks_l, ts_l) \ t \ (ks_r, ts_r) = \begin{cases} full \ d \ t : fix_f \\ low \ d \ t : fix_l \\ otherwise : (ks_l \mathbin{+\!\!+} ks_r, ts_l \mathbin{+\!\!+} [t] \mathbin{+\!\!+} ts_r) \end{cases}
$$
(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* (eq. (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.

$$
fix_l = \begin{cases} ks_l \neq \varnothing : & balance \ d \ (init \ ks_l, init \ ts_l) \ (unsplit \ t_m \ k_m \ t) \ (ks_r, ts_r) \\ ks_r \neq \varnothing : & balance \ d \ (ks_l, ts_l) \ (unsplit \ t \ k_1 \ t_1) \ (tail \ ks_r, tail \ ts_r) \\ otherwise : & t \end{cases}
$$
(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 $delete$:

$$fix\ (d, (\varnothing, [t])) \ = \ (d, t)$$

$$fix\ (d, t) \quad = \quad \begin{cases} full\ d\ t: & (d, ([k], [l, r])), \text{where } (l, k, r) = split\ d\ t \\ otherwise: & (d, t) \end{cases} \quad (7.32)$$

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. Fig-
ures 7.10 to 7.12 give examples of delete.
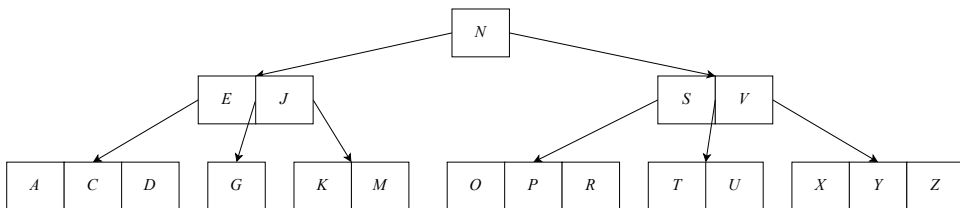
Figure 7.10: Before delete

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.
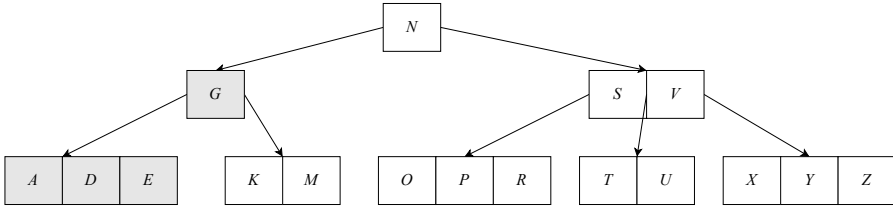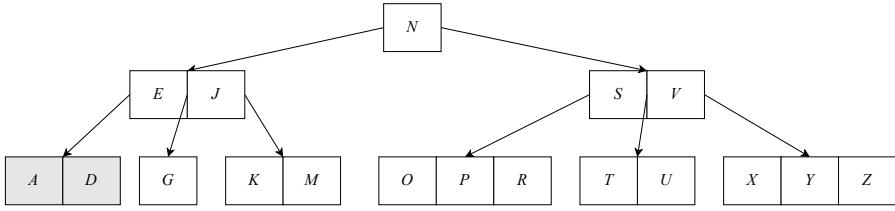
Figure 7.11: Delete 'C', then delete 'J'
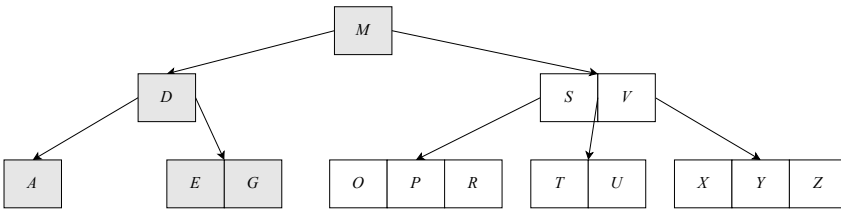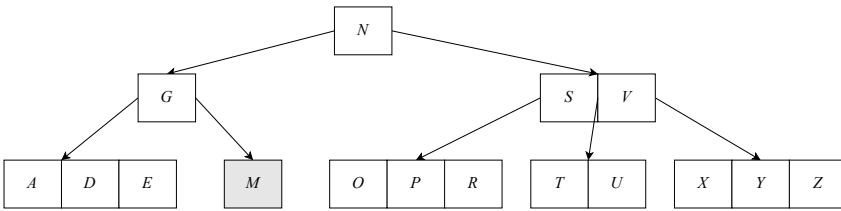
Figure 7.12: Delete 'K', then delete 'N'

Case 2. If x exists in node t, but t is not a leaf. There are three sub-cases:

Case 2a. As shown in fig. 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 fig. 7.13.
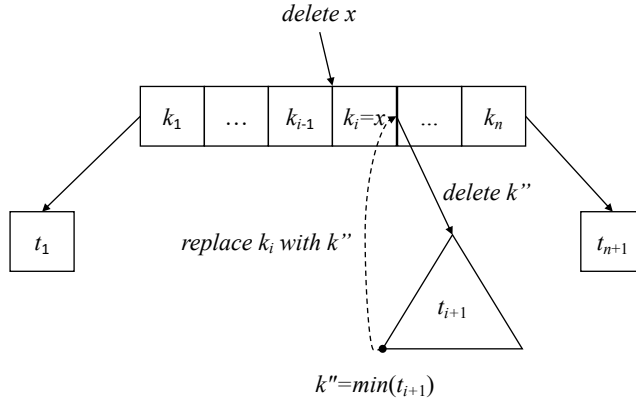
Figure 7.13: Replace k_i with $k'' = min(t_{i+1})$, then delete k'' from t_{i+1}.

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 fig. 7.14.
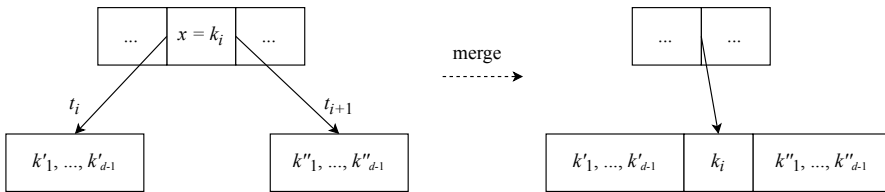
Figure 7.14: Merge before delete

Merge pushes a key k_i down 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 fig. 7.15.
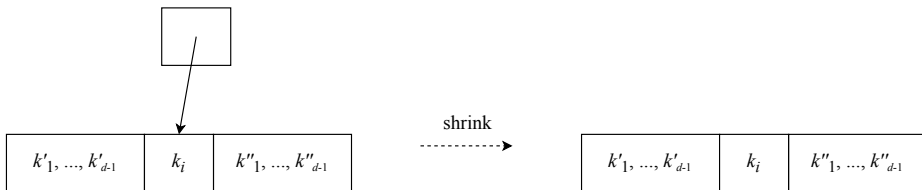
Figure 7.15: Shrink

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 fig. 7.16, t_i received one more key. We next recursively delete x from t_i.
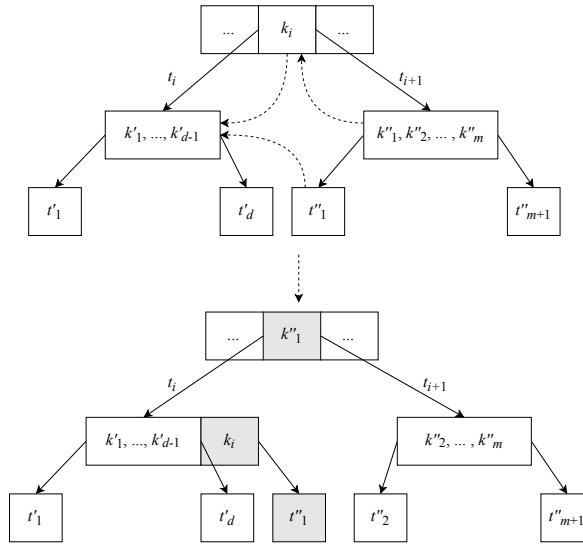
Figure 7.16: Borrow from the right sibling.

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 fig. 7.17. Then recursively delete x from it.
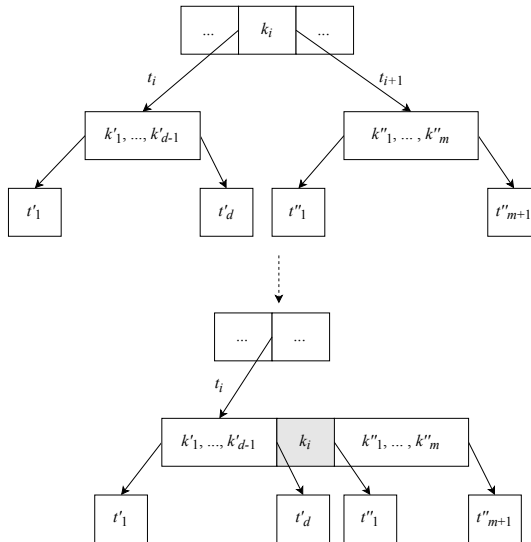
Figure 7.17: Merge t_i, k, t_{i+1}

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 = kᵢ(t) then
 8:            if t is leaf then                                              ▷ case 1
 9:                REMOVE(K(t), k)
10:            else                                                          ▷ case 2
11:                if |K(tᵢ(t))| ≥ d then                                    ▷ case 2a
12:                    kᵢ(t) ← MAX(tᵢ(t))
13:                    DELETE(tᵢ(t), kᵢ(t))
14:                else if |K(tᵢ₊₁(t))| ≥ d then                             ▷ case 2b
15:                    kᵢ(t) ← MIN(tᵢ₊₁(t))
16:                    DELETE(tᵢ₊₁(t), kᵢ(t))
17:                else                                                      ▷ case 2c
18:                    MERGE-AT(t, i)
19:                    DELETE(tᵢ(t), k)
20:                    if K(T) is empty then
21:                        t ← tᵢ(t)                                         ▷ Shrinks height
22:            return t
23:        if t is not leaf then
24:            if k > kₙ(t) then
25:                i ← i + 1
26:            if |K(tᵢ(t))| < d then                                        ▷ case 3
27:                if i > 1 and |K(tᵢ₋₁(t))| ≥ d then                        ▷ case 3a: left
28:                    INSERT(K(tᵢ(t)), kᵢ₋₁(t))
29:                    kᵢ₋₁(t) ← POP-LAST(K(tᵢ₋₁(t)))
30:                    if tᵢ(t) is not leaf then
31:                        INSERT(T(tᵢ(t)), POP-BACK(T(tᵢ₋₁(t))))
32:                else if i ≤ n and |K(tᵢ₊₁(t))| ≥ d then                   ▷ case 3a: right
33:                    APPEND(K(tᵢ(t)), kᵢ(t))
34:                    kᵢ(t) ← POP-FIRST(K(tᵢ₊₁(t)))
35:                    if tᵢ(t) is not leaf then
36:                        APPEND(T(tᵢ(t)), POP-FIRST(T(tᵢ₊₁(t))))
37:                else                                                      ▷ case 3b
38:                    if i = n + 1 then
39:                        i ← i − 1
40:                    MERGE-AT(t, i)
41:            DELETE(tᵢ(t), k)
42:            if K(t) is empty then                                         ▷ Shrinks height
43:                t ← t₁(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 ← tᵢ(t)
 3:     y ← tᵢ₊₁(t)
 4:     K(x) ← K(x) ⧺ [kᵢ(t)] ⧺ K(y)
 5:     T(x) ← T(x) ⧺ T(y)
 6:     REMOVE-AT(K(t), i)
 7:     REMOVE-AT(T(t), i + 1)
```

Exercise 7.3

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.

7.3.2. 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.

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 lookup:

```
Optional<(BTree<K, deg>, Int)> lookup(BTree<K, deg> tr, K key) {
    loop {
        Int i = 0, n = length(tr.keys)
        while i < n and key > tr.keys[i] {
            i = i + 1
        }
        if i < n and key == tr.keys[i] then return Optional.of((tr, i))
        if isLeaf(tr) {
            return Optional.Nothing
        } else {
            tr = tr.subTrees[i]
        }
    }
}
```

Imperative merge before delete:

```
BTree<K, deg> delete(BTree<K, deg> t, K x) {
    if empty(t.keys) then return t
    Int i = 0, n = length(t.keys)
    while i < n and x > t.keys[i] { i = i + 1 }
    if x == t.keys[i] {
        if isLeaf(t) {            // case 1
            removeAt(t.keys, i)
        } else {
            var tl = t.subtrees[i]
            var tr = t.subtrees[i + 1]
            if not low(tl) {          // case 2a
                t.keys[i] = max(tl)
                delete(tl, t.keys[i])
            } else if not low(tr) {  // case 2b
                t.keys[i] = min(tr)
                delete(tr, t.keys[i])
            } else {                    // case 2c
                mergeSubtrees(t, i)
                delete(d, tl, x)
                if empty(t.keys) then t = tl  // shrink height
            }
```

```
            return t
    }
    if not isLeaf(t) {
        if x > t.keys[n - 1] then i = i + 1
        if low(t.subtrees[i]) {
            var tl = if i == 0 then null else t.subtrees[i - 1]
            var tr = if i == n then null else t.subtrees[i + 1]
            if tl ≠ null and (not low(tl)) {    // case 3a, left
                insert(t.subtrees[i].keys, 0, t.keys[i - 1])
                t.keys[i - 1] = popLast(tl.keys)
                if not isLeaf(tl) {
                    insert(t.subtrees[i].subtrees, 0, popLast(tl.subtrees))
                }
            } else if tr ≠ null and (not low(tr)) {  // case 3a, right
                append(t.subtrees[i].keys, t.keys[i])
                t.keys[i] = popFirst(tr.keys)
                if not isLeaf(tr) {
                    append(t.subtrees[i].subtrees, popFirst(tr.subtrees))
                }
            } else {          // case 3b
                mergeSubtrees(t, if i < n then i else (i - 1))
                if i == n then i = i - 1
            }
        delete(t.subtrees[i], x)
        if empty(t.keys) then t = t.subtrees[0]     // shrink height
        }
    }
    return t
}
```

merge sub-trees, find the min/max key from a B-tree.

```
void mergeSubtrees(BTree<K, deg>, Int i) {
    t.subtrees[i].keys += [t.keys[i]] + t.subtrees[i + 1].keys
    t.subtrees[i].subtrees += t.subtrees[i + 1].subtrees
    removeAt(t.keys, i)
    removeAt(t.subtrees, i + 1)
}

K max(BTree<K, deg> t) {
    while not empty(t.subtrees) {
        t = last(t.subtrees)
    }
    return last(t.keys)
}

K min(BTree<K, deg> t) {
    while not empty(t.subtrees) {
        t = t.subtrees[0]
    }
    return t.keys[0]
}
```

Chapter 8

Binary Heaps

8.1 Definition

Heaps are widely used for sorting, priority scheduling, graph algorithms, and etc.[40]. The most popular implementation uses array to represent the heap as a complete binary tree[4]. Robert W. Floyd developed an efficient heap sort algorithm based on this idea[41][42]. We can implement the heap with varies data structures but not limit to array. This chapter focuses 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 satisfy a property, and define the following 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 of 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.

Alternatively, we can define the heap that always keeps the maximum on the top. We call the heap with the minimum on the top as *min-heap*, the maximum on the top as *max-heap*. When implement heap with a tree, we can put the minimum (or the maximum) in the root. After pop, remove the root, and rebuild the tree from the sub-trees. We call the heap implemented with binary tree as *binary heap*.

8.2 Binary heap by array

The first implementation is to represent the complete binary tree with 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 a 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. The i-th cell in array corresponds to a node, its parent 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):
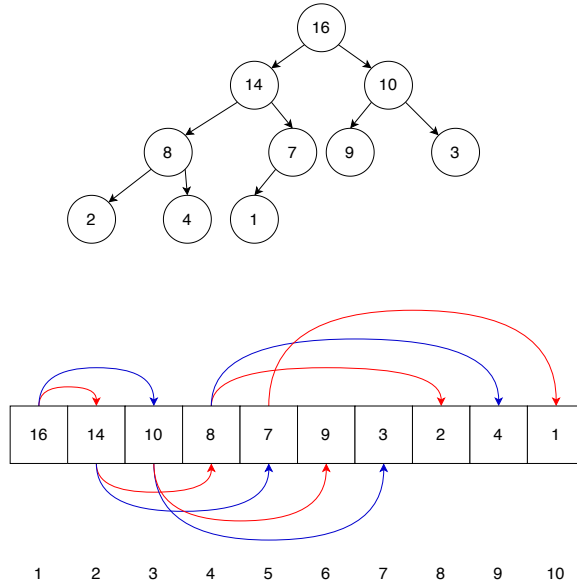
Figure 8.1: Map between a complete binary tree and an array.

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

8.2.1 Heapify

Heapify is the process that maintains heap property, keeping the minimum element on the top. For binary heap, we 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. For the cell indexed i in array representation, we examine whether all the sub-tree elements are greater than or equal to it ($\geq$), exchange when do not satisfy.

```
 1: function HEAPIFY(A, i)
 2:     n ← |A|
 3:     loop
 4:         s ← i                                   ▷ s is the smallest
 5:         l ← LEFT(i), r ← RIGHT(i)
 6:         if l ≤ n and A[l] < A[i] then
 7:             s ← l
 8:         if r ≤ n and A[r] < A[s] then
 9:             s ← r
10:         if s ≠ i then
11:             EXCHANGE A[i] ↔ A[s]
12:             i ← s
13:         else
14:             return
```

Because we recursive check sub-trees, the process time is proportion to the height of the tree. HEAPIFY is bound to $O(\lg n)$, where n is the length of the array. Figure 8.2 gives the steps when apply HEAPIFY from 2 to array [1, 13, 7, 3, 10, 12, 14, 15, 9, 16]. The result is [1, 3, 7, 9, 10, 12, 14, 15, 13, 16].
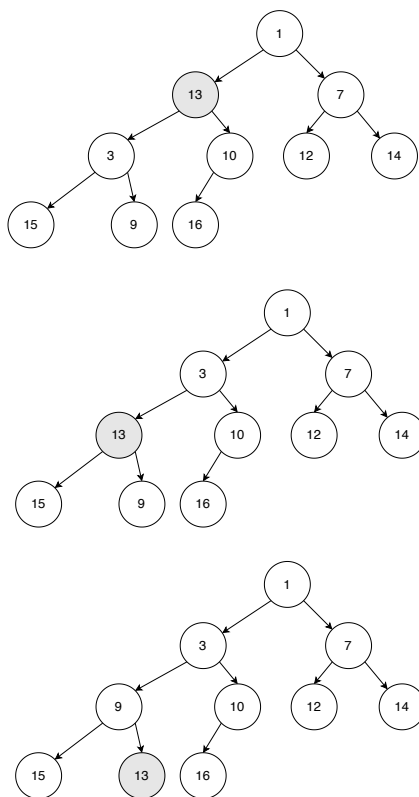
Figure 8.2: Heapify. Step 1: the minimum of 13, 3, 10 is 3, exchange 3 ↔ 13; Step 2: the minimum of 13, 15, 9 is 9, exchange 13 ↔ 9; Step 3: 13 is leaf, terminate.

8.2.2   Build

We can build heap from an array with HEAPIFY. List the numberf of nodes in each level
of a complete binary tree: $1, 2, 4, 8, ...$. They are all powers 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 (indexed at $\leq \lfloor n/2 \rfloor$) bottom-up. The build function is defined as below:

1: **function** BUILD-HEAP(A)
2:     $n \leftarrow |A|$
3:     **for** $i \leftarrow \lfloor n/2 \rfloor$ down to 1 **do**
4:         HEAPIFY(A, i)

Although 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 number of comparisons and moves 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 eq. (8.2) from eq. (8.3):

$$
\begin{aligned}
2S - S &= n[\frac{1}{2} + (2\frac{1}{4} - \frac{1}{4}) + (3\frac{1}{8} - 2\frac{1}{8}) + ...] \quad \text{shift by one and subtract} \\
S &= n[\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...] \quad\quad\quad\quad\quad\quad \text{geometric series} \\
&= n
\end{aligned}
$$

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, lookup the top k elements, decrease an
element in min-heap (or increase an element in max-heap), and insert. The root of the
binary heap, which is the first array cell stores the minimum: $Top(A) = A[1]$.

After pop, the remaining elements all shift ahead a cell of the array. However, without
the root, the rest is not a binary tree any more. Alternatively, we swap the head and the
tail of the array, then reduce the array length by one. It's equivalent to remove a leaf but
not the root. We then apply HEAPIFY to the root to recover the heap property:

1: **function** POP(A)
2:     $x \leftarrow A[1], n \leftarrow |A|$
3:     EXCHANGE $A[1] \leftrightarrow A[n]$
4:     REMOVE(A, n)
5:     **if** A is not empty **then**
6:         HEAPIFY($A, 1$)
7:     **return** x
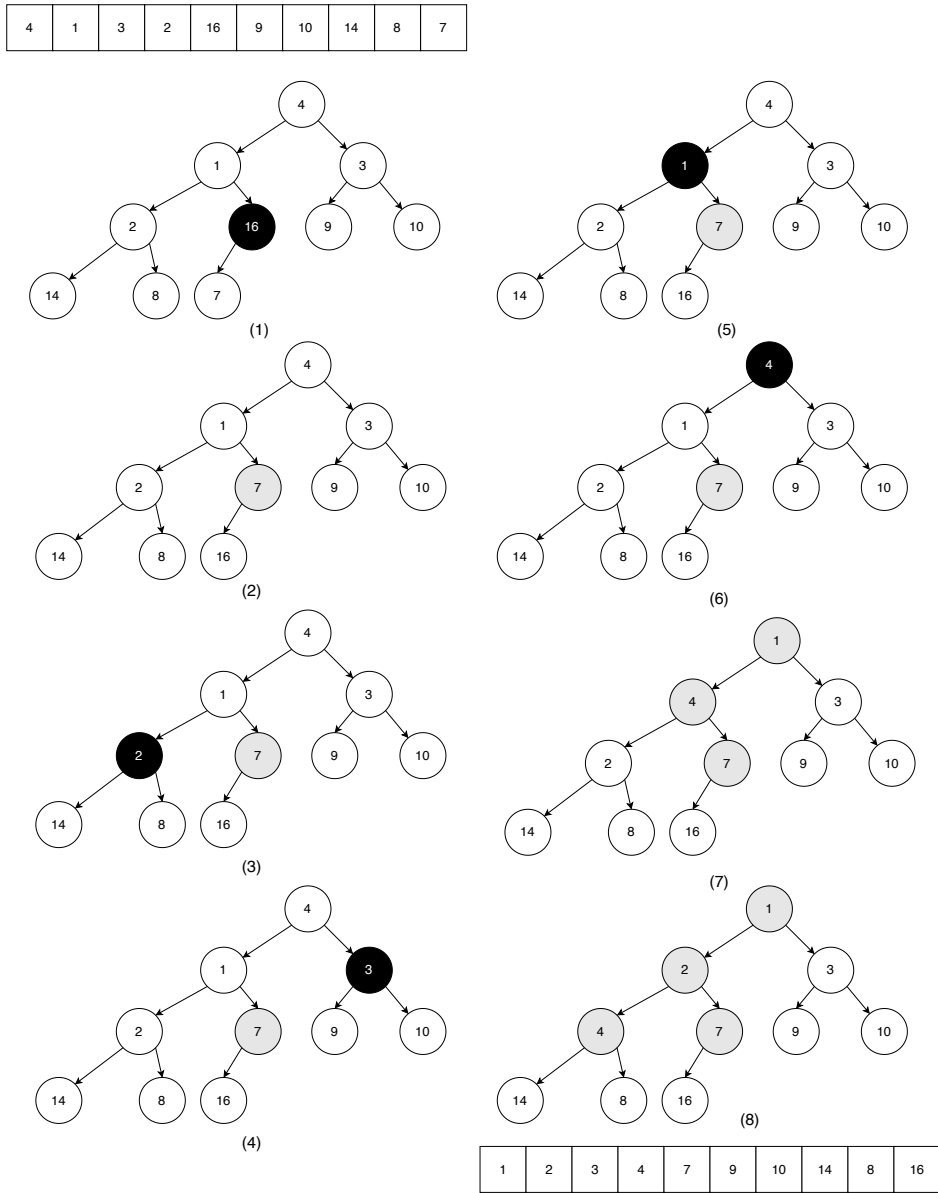
Figure 8.3: Build heap. (1) $16 > 7$; (2) exchange $16 \leftrightarrow 7$; (3) $2 < 14$ and $2 < 8$; (4) $3 < 9$ and $3 < 10$; (5) $1 < 2$ and $1 < 7$; (6) $1 < 4$ and $1 < 3$; (7) exchange $4 \leftrightarrow 1$; (8) exchange $4 \leftrightarrow 2$, end.

It takes constant time to remove the last element from array, pop is bound to $O(\lg n)$ time as it calls HEAPIFY. We can develop a solution to find the top k elements from an array. First build a heap from the array, then repeatedly pop k times:

1: **function** TOP-K(A, k)
2:     $R \leftarrow [\,]$
3:     BUILD-HEAP(A)
4:     **loop** MIN(k, $|A|$) times   ▷ cut off when $k > |A|$
5:         APPEND(R, POP(A))
6:     **return** R

Further, we can implement a *priority queue* to schedule tasks with priorities. Every time, we peek the highest priority task to run. To run an urgent task earlier, we can increase its priority, meaning to decrease an element in a min-heap, as shown in fig. 8.4.
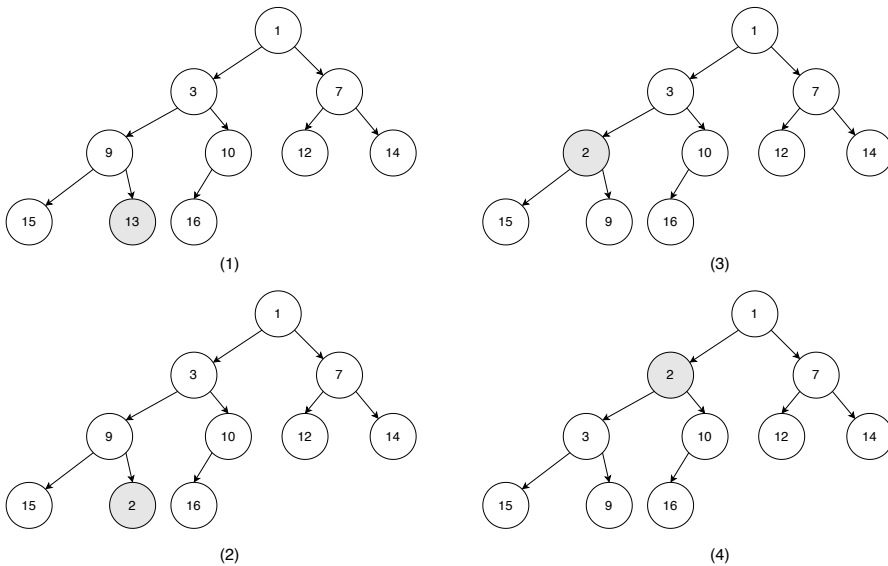
Figure 8.4: Decrease 13 to 2. Exchange 2 and 9, then exchange with 3.

The heap property may be broken when decrease some element in a min-heap. Let the decreased element be $A[i]$, 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 $A[i] \leftrightarrow A[\,$PARENT($i$)$\,]$
4:         $i \leftarrow$ PARENT(i)

We can realize push with HEAP-FIX[4]. Append the new element k to the tail, 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 n elements (with (n) time), then repeatedly pop the top to build the ascending result. Each pop is bound to $O(\lg n)$ time,

the total time is bound to $O(n \lg n)$. Besides, we need another list of length n to hold the result.

1: **function** HEAP-SORT(A)
2:     $R \leftarrow [\,]$
3:     BUILD-HEAP(A)
4:     **while** $A \neq [\,]$ **do**
5:         APPEND(R, POP(A))
6:     **return** R

Floyd developed a fast implementation with max-heap. Every time, swap the head and the tail of the array. The maximum is swapped to the expected position, the tail; and the previous tail becomes the new top. Next decrease the heap size by one, and apply HEAPIFY to restore the heap property. Repeat this till the heap size decrease to one. This algorithm needn't the additional space.

1: **function** HEAP-SORT(A)
2:     BUILD-MAX-HEAP(A)
3:     $n \leftarrow |A|$
4:     **while** $n > 1$ **do**
5:         EXCHANGE $A[1] \leftrightarrow A[n]$
6:         $n \leftarrow n - 1$
7:         HEAPIFY($A[1...n], 1$)

Exercise 8.1

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($A[i...n], 1$)

8.1.2. 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:     $n \leftarrow |A|$
4:     **for** $i \leftarrow 1$ to $min(k, n)$ **do**
5:         HEAPIFY($A[i...n], 1$)

8.3   Leftist heap and skew heap

With an explicit binary tree, after pop the root, there remain two sub-trees. Both are heaps as shown in fig. 8.5. How can we merge them to a new heap? To maintain the heap property, the new root must be the minimum of the remaining. We can give the two edge cases easily:

$$
\begin{aligned}
merge \; \varnothing \; R &= R \\
merge \; L \; \varnothing &= L \\
merge \; L \; R &= ?
\end{aligned}
$$
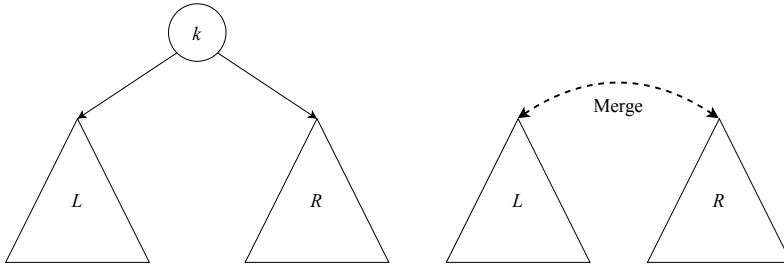
Figure 8.5: Merge left and right sub-trees after pop.

Both left and right sub-trees are heaps. If neither is empty, each root stores the minimum respectively. We compare the two roots, and peek the smaller one 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 $(A, x, merge\ B\ R)$. We always merge the right sub-tree for simplicity. This method generates the *leftist heap.*

8.3.1   Leftist heap

The leftist heap is implemented with leftist tree. C. A. Crane in $1972^{[43]}$ developed the leftist tree. He defines 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 fig. 8.6, The nearest leaf node to 4 is 8, the rank of 4 is 2; Both 6 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. Define the merge method with rank as below. Denote the rank of the left and right sub-trees be r_l, r_r respectively:
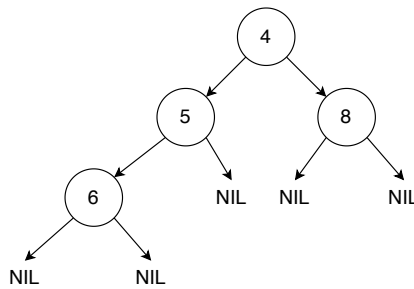
Figure 8.6: $rank(4) = 2$, $rank(6) = rank(8) = rank(5) = 1$.

1. Always merge the right sub-tree;

2. When $r_l < r_r$, swap the left and right sub-trees.

We call this merge method the 'leftist property'. Basically, a leftist tree always has the shortest path to some NIL on the right. Although it seems unbalanced, there is critical fact:

Theorem 8.3.1. *For a leftist tree T of n nodes, the path from the root to the rightmost NIL has at most $\lfloor \log(n+1) \rfloor$ nodes.*

We skip the proof$^{[44]\ [51]}$. This theorem ensures any algorithm processes along this path is bound to $O(\lg n)$ time. We define the leftist tree as a binary tree plus an additional rank. Let the none empty leftist tree be (r, L, k, R):

```
data LHeap a = Empty | Node Int (LHeap a) a (LHeap a)
```

Function *rank* returns the rank value:

$$
\begin{aligned}
rank \ \varnothing &= 0 \\
rank \ (r, L, k, R) &= r
\end{aligned} \tag{8.4}
$$

Define an auxiliary *make* function. It compares the ranks of the sub-trees and swap them if necessary.

$$
make(A, k, B) = \begin{cases} rank(A) < rank(B) : & (1 + rank(A), B, k, A) \\ \text{otherwise} : & (1 + rank(B), A, k, B) \end{cases} \tag{8.5}
$$

For the two trees A and B, if $rank(A)$ is smaller, set B as the left sub-tree, and A as the right. The rank of the new node is $1 + rank(A)$; otherwise if $rank(B)$ is smaller, set A as the left sub-tree, and B as the right. The rank of the new node is $1 + rank(B)$. Given two leftist heaps, and denote them as $H_1 = (r_1, L_1, K_1, R_1)$ and $H_2 = (r_2, L_2, k_2, R_2)$ if not empty, define merge:

$$
\begin{aligned}
merge \ \varnothing \ H_2 &= H_2 \\
merge \ H_1 \ \varnothing &= H_1 \\
merge \ H_1 \ H_2 &= \begin{cases} k_1 < k_2 : & make(L_1, k_1, merge \ R_1 \ H_2) \\ \text{otherwise} : & make(L_2, k_2, merge \ H_1 \ R_2) \end{cases}
\end{aligned} \tag{8.6}
$$

We always *merge* to the right sub-tree recursively to maintain the leftist property. It is bound to $O(\lg n)$ time. The binary heap implemented with array performs well in most cases, suitable for the modern cache technology. However, it takes $O(n)$ time to merge, because it needs concatenate two arrays, and rebuild the heap[50]:

1: **function** MERGE-HEAP(A, B)
2:     $C \leftarrow$ CONCAT(A, B)
3:     BUILD-HEAP(C)

We can access the top of the leftist heap in constant time(assume it is not empty):

$$
top \ (r, L, k, R) = k \tag{8.7}
$$

After pop the root, we merge the two sub-trees in $O(\lg n)$ time.

$$
pop \ (r, L, k, R) = merge \ L \ R \tag{8.8}
$$

To insert a new element k, create a leaf of k, then merge it to the heap:

$$
insert \ k \ H = merge \ (1, \varnothing, k, \varnothing) \ H \tag{8.9}
$$

We can build a leftist heap from a list as (Curried form): $build = fold_r \ insert \ \varnothing$. Then repeatedly pop the minimum to output the sorted result:

$$
sort = heapSort \circ build \tag{8.10}
$$

Where

$$
\begin{aligned}
heapSort \ \varnothing &= [\ ] \\
heapSort \ H &= (top \ H) : heapSort \ (pop \ H)
\end{aligned} \tag{8.11}
$$

It pops n times, each takes $O(\lg n)$ time. The total time is bound to $O(n \lg n)$.
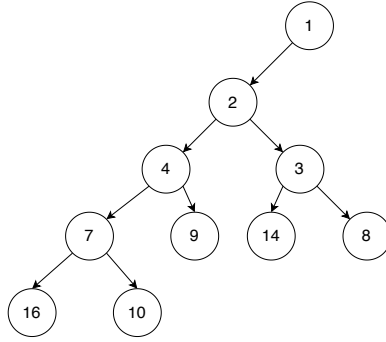
Figure 8.7: Build the leftist heap from $[9, 4, 16, 7, 10, 2, 14, 3, 8, 1]$.

8.3.2 Skew heap

Leftist heap may lead to unbalanced tree in some cases as shown in fig. 8.8. Skew heap is a self-adjusting heap. It simplifies the leftist heap and improves balance[46] [47]. To 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 that either sub-tree has a NIL node. The rank is always 1 no matter how big the sub-tree is. Skew heap always swaps the sub-trees for simplification.
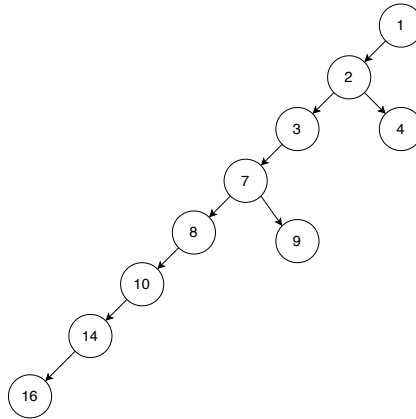
Figure 8.8: Leftist heap built from $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$.

Skew heap is implemented with skew tree. A 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. Denote the none empty tree as (L, k, R).

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

When merge two none empty skew trees $H_1 = (L_1, k_1, R_1)$ and $H_2 = (L_2, k_2, R_2)$, if $k_1 < k_2$, then choose k_1 (otherwise k_2) as the new root. Then merge the greater one with a sub-tree. 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 ($merge\ R_1\ H_2, k_1, L_1$).

$$merge \ \varnothing \ H_2 \quad = \quad H_2$$
$$merge \ H_1 \ \varnothing \quad = \quad H_1$$
$$merge \ H_1 \ H_2 \quad = \quad \begin{cases} k_1 < k_2 : & (merge \ R_1 \ H_2, k_1, L_1) \\ \text{otherwise} : & (merge \ H_1 \ R_2, k_2, L_2) \end{cases} \tag{8.12}$$

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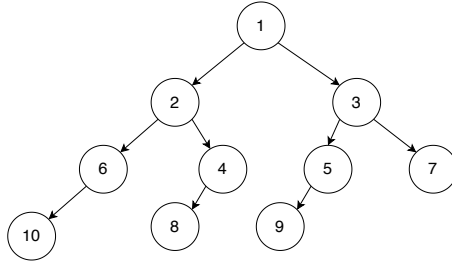
Figure 8.9: Skew tree built from $[1, 2, ..., 10]$.

8.4 Splay heap

Both leftist heap and skew heap are implemented with binary trees. If change to binary search tree, then the minimum will not be in root. We need $O(\lg n)$ time to locate the minimum. The performance downgrades if the tree is not balanced. Although we can use the red-black tree to secure balancing, the splay tree provides a simple solution. It dynamically 'splays' to balance the tree. Splay tree takes cache-like approach. It rotates the node currently being accessed to the root, reduces the access time for the next visit. We define such operation as 'splay'. The tree tends to be more balanced after multiple splays. Most splay tree operations perform in amortized $O(\lg n)$ time. Sleator and Tarjan developed splay tree in 1985 [48] [49].

We give two implementations. The first is pattern matching, it needs match multiple cases; the second has the uniformed form, but the implementation is complex. When access node x, denote the parent node as p, and grand parent node (if there is) as g. There are 3 cases, each has two symmetric sub-cases. Figure 8.10 shows one:

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 below when access element y:
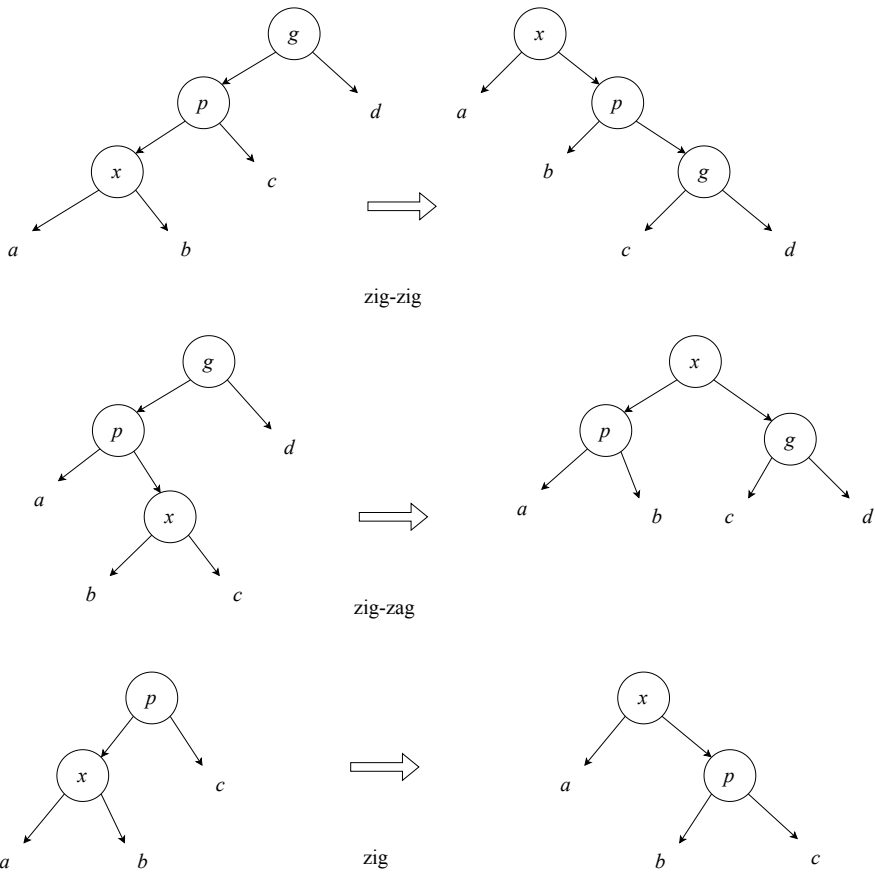
Figure 8.10: **zig-zig**: x and p are both on left or right, x becomes the new root. **zig-zag**: x and p are on different sides, x becomes the new root, p and g are siblings. **zig**: p is the root, rotate to make x as the root.

$$
\begin{aligned}
splay\ y\ (((a, x, b), p, c), g, d) &= \begin{cases} x = y: & (a, x, (b, p, (c, g, d))) \\ \text{otherwise}: & T \end{cases} & \text{zig-zig} \\[2ex]
splay\ y\ (a, g, (b, p, (c, x, d))) &= \begin{cases} x = y: & (((a, g, b), p, c), x, d) \\ \text{otherwise}: & T \end{cases} & \text{zig-zig symmetric} \\[2ex]
splay\ y\ (a, p, (b, x, c), g, d) &= \begin{cases} x = y: & ((a, p, b), x, (c, g, d)) \\ \text{otherwise}: & T \end{cases} & \text{zig-zag} \\[2ex]
splay\ y\ (a, g, ((b, x, c), p, d)) &= \begin{cases} x = y: & ((a, g, b), x, (c, p, d)) \\ \text{otherwise}: & T \end{cases} & \text{zig-zag symmetric} \\[2ex]
splay\ y\ ((a, x, b), p, c) &= \begin{cases} x = y: & (a, x, (b, p, c)) \\ \text{otherwise}: & T \end{cases} & \text{zig} \\[2ex]
splay\ y\ (a, p, (b, x, c)) &= \begin{cases} x = y: & ((a, p, b), x, c) \\ \text{otherwise}: & T \end{cases} & \text{zig symmetric} \\[2ex]
splay\ y\ T &= T & \text{others}
\end{aligned}
$$

$$\text{(8.13)}$$

The tree is unchanged for all other cases. Every time when insert, we trigger splay to adjust the balance. If the tree is empty, the result is a leaf; otherwise, compare the new element and the root, then recursively insert to left (less than) or right (greater than) sub-tree and splay.

$$
\begin{aligned}
insert\ y\ \varnothing &= (\varnothing, y, \varnothing) \\
insert\ y\ (L, x, R) &= \begin{cases} y < x: & splay\ y\ (insert\ y\ L, x, R) \\ \text{otherwise}: & splay\ y\ (L, x, insert\ y\ R) \end{cases} & \text{(8.14)}
\end{aligned}
$$
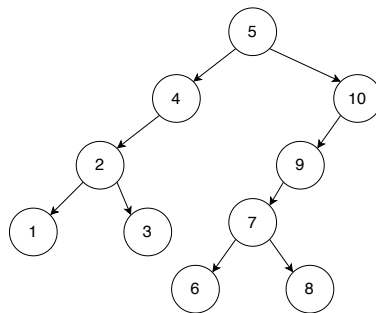
Figure 8.11: Splay tree built from $[1, 2, ..., 10]$.

Figure 8.11 gives the splay tree built from $[1, 2, ..., 10]$. It generates a relative balanced tree. Okasaki gives a simple rule for splaying[3]. Whenever follow two left or two right branches continuously, rotate the two nodes. When access x, if have moved left or right twice, then partition T as L and R recursively, where L contains all elements less than x, while R contains the remaining. Then create a new tree with x as the root, and L, R as the sub-trees.

$$
\begin{aligned}
partition\ y\ \varnothing\ &=\ (\varnothing, \varnothing) \\
partition\ y\ (L, x, R)\ &=
\end{aligned}
$$

$$
\begin{cases}
x < y & \begin{cases} R = \varnothing & (T, \varnothing) \\[4pt] R = (L', x', R') & \begin{cases} x' < y & (((L, x, L'), x', A), B) \\ & \text{where: } (A, B) = partition\ y\ R' \\[4pt] \text{otherwise} & ((L, x, A), (B, x', R')) \\ & \text{where: } (A, B) = partition\ y\ L' \end{cases} \end{cases} \\[12pt]
\text{otherwise} & \begin{cases} L = \varnothing & (\varnothing, T) \\[4pt] L = (L', x', R') & \begin{cases} y < x' & (A, (L', x', (R', x, R))) \\ & \text{where: } (A, B) = partition\ y\ L' \\[4pt] \text{otherwise} & ((L', x', A), (B, x, R)) \\ & \text{where: } (A, B) = partition\ y\ R' \end{cases} \end{cases}
\end{cases}
$$

$$(8.15)$$

We *partition*s the tree T with a pivot y. For empty tree, the result is $(\varnothing, \varnothing)$; otherwise for tree (L, x, R), if $x < y$, there are two sub-cases: (1) R is empty. All elements in the tree are less then y, the result is $(T, \varnothing)$; (2) $R = (L', x', R')$. If $x' < y$, then recursively partition R' with y. Put all 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 into (A, B). The result is also a pair of (L, x, A) and (B, x', R'). The result is symmetric when $y < x$.

Alternatively, we can define insert with *partition*. When insert element k to T, first partition the heap to two sub-trees of L and R, satisfying $L < k < R$ (that L contains all elements smaller than k, and R contains the rest). Then create a new tree of (L, k, R) (rooted at k, with L, R as sub-trees).

$$insert\ k\ T = (L, k, R), \text{where } (L, R) = partition\ k\ T \tag{8.16}$$

Since splay tree is essentially a binary search tree, the minimum is at the left most. We keep traversing the left sub-tree to access the 'top' of the heap:

$$
\begin{aligned}
top\ (\varnothing, k, R) &= k \\
top\ (L, k, R) &= top\ L
\end{aligned}
\tag{8.17}
$$

This is equivalent to min for the binary search tree (alternatively, we can define: $top = \min$). For pop, we remove the minimum and splay when move left twice.

$$
\begin{aligned}
pop\ (\varnothing, k, R) &= R \\
pop\ ((\varnothing, k', R'), k, R) &= (R', k, R) \\
pop\ ((L', k', R'), k, R) &= (pop\ L', k', (R', k, R))
\end{aligned}
\tag{8.18}
$$

The third row performs splaying based on the binary search tree property without calling *partition*. Top and pop both are bound to $O(\lg n)$ time when the splay tree is balanced.

We can implement *merge* in $O(\lg n)$ time with *partition*. When merge two none-empty trees, we choose either root as the pivot to partition the other, then recursively merge the sub-trees:

$$
\begin{aligned}
merge\ T\ \varnothing &= T \\
merge\ \varnothing\ T &= T \\
merge\ (L, x, R)\ T &= ((merge\ L\ L')\ x\ (merge\ R\ R'))
\end{aligned}
\tag{8.19}
$$

where

$$(L', R') = partition\ x\ T$$

8.5   Summary

We give the generic definition of binary heap in this chapter. There are several imple-
mentations. The array based representation is suitable for imperative implementation. It
maps a complete binary tree to array, supporting random access in constant time. We
directly use the binary tree to implement the heap in functional way. Most operations
are bound to $O(\lg n)$ time, some are amortized $O(1)$ time[3]. When extend from the bi-
nary tree to k-ary tree, we obtain binomial heap, Fibonacci heap, and pairing heap (See
chapter 10).

Exercise 8.2

8.2.1. Implement leftist heap and skew heap imperatively.

8.2.2. Define fold for heap.

8.6   Appendix - example programs

Access parent, and sub-trees in a complete binary tree with bit-wise operation (index
from 0) for the array representation:

```
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
}
```

Find 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:

```
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 Empty h = h
merge h Empty = 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 Empty h = h
merge h Empty = 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
```

Splay heap insert:

```
insert Empty y = Node Empty y Empty
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 Empty _ = (Empty, Empty)
partition t@(Node l x r) y
    | x < y =
        case r of
          Empty → (t, Empty)
          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
          Empty → (Empty, 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 t Empty = t
merge Empty 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

Selection sort is a straightforward sorting algorithm. It repeatedly selects the minimum (or maximum) from a collection of elements. The performance is below the divide and conqueror algorithms, like quick sort and merge sort. We'll seek varies of improvement and finally evolve it to heap sort, achieving $O(n \lg n)$ time bound, the upper limit of comparison based sort algorithms.

When facing a bunch of grapes, there are two types of people. One love to pick the biggest grape every time, the other always pick the smallest one. The former enjoy the grape in ascending order of size, while the latter in descending order. In either case, one essentially applies selection sort method, 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 as shown in fig. 9.1. If select the maximum, then it sorts in descending order. The compare operation can be abstract.

$$
\begin{aligned}
sort\ [\,] &= [\,] \\
sort\ A &= m : sort\ (A - [m]) \quad \text{where } m = \min\ A
\end{aligned}
\tag{9.1}
$$

$A - [m]$ means the remaining elements in A except m. The corresponding imperative implementation is as below:

1: **function** SORT(A)
2:     $X \leftarrow [\,]$
3:     **while** $A \neq [\,]$ **do**
4:         $x \leftarrow$ MIN(A)
5:         DEL(A, x)
6:         APPEND(X, x)
7:     **return** X

We can improve it to in-place sort by reusing A. Place the minimum in $A[1]$, the second smallest in $A[2]$, ... When find the i-th smallest element, swap it with $A[i]$.

1: **function** SORT(A)
2:     **for** $i \leftarrow 1$ to $|A|$ **do**
3:         $m \leftarrow$ MIN-AT(A, i)
4:         EXCHANGE $A[i] \leftrightarrow A[m]$

Let $A = [a_1, a_2, ..., a_n]$, when select the i-th smallest element, $[a_1, a_2, ..., a_{i-1}]$ are sorted. Call MIN-AT(A, i) to find the minimum of $[a_i, a_{i+1}, ..., a_n]$, then swap with a_i. Repeat this to process all elements as shown in fig. 9.2.
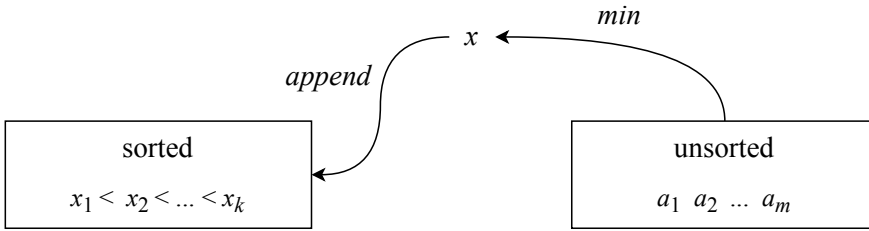
Figure 9.1: The left is sorted, repeatedly select the minimum of the rest and append.
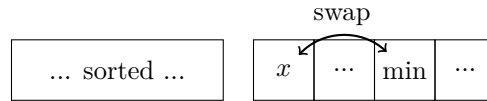
Figure 9.2: The left is sorted, repeatedly find the minimum and swap to the right position.

9.1 Find the minimum

We use the 'compare and swap' method to find the minimum. 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:     $m \leftarrow i$
3:     **for** $i \leftarrow m + 1$ to $|A|$ **do**
4:         **if** $A[i] < A[m]$ **then**
5:             $m \leftarrow i$
6:     **return** m

MIN-AT locates the minimum at m from slice $A[i...]$. m starts from $A[i]$, then scan $A[i + 1], A[i + 2], ...$ To find the minimum of a list L, if L is a singleton $[x]$, then x 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.

$$
\begin{aligned}
\min \ [x] \quad &= \quad (x, [\,]) \\
\min \ (x{:}xs) \quad &= \quad \begin{cases} x < y : & (x, xs), \text{ where } (y, ys) = \min \ xs \\ \text{otherwise} : & (y, \ x{:}ys) \end{cases}
\end{aligned}
\tag{9.2}
$$

We can further improve it to tail recursive. Divide the elements into 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. m holds the minimum finally. At any time, we have the invariant: $L = A + [m] + B$, where $a \leq m \leq b$ for every $a \in A, b \in B$.

$$
\min \ (x{:}xs) = min' \ [\,] \ x \ xs
\tag{9.3}
$$

Where:

$$
\begin{aligned}
min' \ as \ m \ [\,] \quad &= \quad (m, as) \\
min' \ as \ m \ (b{:}bs) \quad &= \quad \begin{cases} b < m : & min' \ (m{:}as) \ b \ bs \\ \text{otherwise} : & min' \ (b{:}as) \ m \ bs \end{cases}
\end{aligned}
\tag{9.4}
$$

Function min returns a pair: the minimum and the list of the remaining. We define selection sort as below:

$$
\begin{aligned}
sort\ [\ ] &= \ [\ ] \\
sort\ xs &= \ m : (sort\ xs'),\ \text{where}\ (m, xs') = \min\ xs
\end{aligned} \tag{9.5}
$$

9.1.1 Performance

Selection sort need scan and find the minimum for n times. It compares $n + (n - 1) + (n - 2) + ... + 1$ times, which is $O(\dfrac{n(n + 1)}{2}) = O(n^2)$. 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 list is in reversed ordered), and worst at $O(n^2)$.

Exercise 9.1

9.1.1. What is the problem with below implementation of min?

$$
\begin{aligned}
min'\ as\ m\ [\ ] &= \ (m, as) \\
min'\ as\ m\ (b{:}bs) &= \ \begin{cases} b < m : & min'\ (as \mathbin{+\!\!+} [m])\ b\ bs \\ \text{otherwise} : & min'\ (as \mathbin{+\!\!+} [b])\ m\ bs \end{cases}
\end{aligned}
$$

9.1.2. Implement the in-place selection sort.

9.2 Improvement

To sort in ascending, descending order flexibly, we abstract the comparison as '$\lhd$'.

$$
\begin{aligned}
sortBy \lhd\ [\ ] &= \ [\ ] \\
sortBy \lhd\ xs &= \ m : sortBy\ \lhd\ xs',\ \text{where}\ (m, xs') = minBy\ \lhd\ xs
\end{aligned} \tag{9.6}
$$

And use $\lhd$ to find the 'minimum':

$$
\begin{aligned}
minBy\ \lhd\ [x] &= \ (x, [\ ]) \\
minBy\ \lhd\ (x{:}xs) &= \ \begin{cases} x \lhd y : & (x, xs),\ \text{where}\ (y, ys) = minBy\ \lhd\ xs \\ \text{otherwise} : & (y,\ x{:}ys) \end{cases}
\end{aligned} \tag{9.7}
$$

For example, we pass $(<)$ to sort a collection of numbers in ascending order: $sortBy\ (<)\ [3, 1, 4, ...]$. As the constraint, the comparison $\lhd$ need satisfy the *strict weak ordering*[52].

- Irreflexivity: for all x, $x \not< x$ (x is not less than itself);

- Asymmetry: for all x and y, if $x < y$, then $y \not< x$;

- Transitivity, for all x, y, and z, if $x < y$, and $y < z$, then $x < z$.

As an impact in-place implementation, we traverse elements, and find the minimum in an inner loop:

```
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:                    $m \leftarrow i$
7:             EXCHANGE $A[i] \leftrightarrow A[m]$

We only need sort $n-1$ elements, and leave the last one to save the last loop. Besides, we needn't swap if $A[i]$ is exact the i-th smallest.

1: **procedure** SORT(A)
2:      **for** $i \leftarrow 1$ to $|A| - 1$ **do**
3:          $m \leftarrow i$
4:          **for** $j \leftarrow i + 1$ to $|A|$ **do**
5:              **if** $A[i] < A[m]$ **then**
6:                  $m \leftarrow i$
7:          **if** $m \neq i$ **then**
8:              EXCHANGE $A[i] \leftrightarrow A[m]$

9.2.1   Cock-tail sort

Knuth gives another implementation [51] called 'cock-tail sort'. Select the maximum, but not the minimum, and move it to the tail as shown in fig. 9.3. 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** $i \leftarrow |A|$ down-to 2 **do**
3:          $m \leftarrow i$
4:          **for** $j \leftarrow 1$ to $i - 1$ **do**
5:              **if** $A[m] < A[i]$ **then**
6:                  $m \leftarrow i$
7:          EXCHANGE $A[i] \leftrightarrow A[m]$
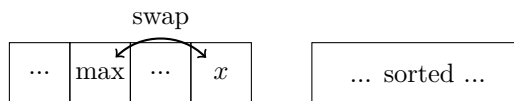
Figure 9.3: Select the maximum and swap to tail

Further, we can pick both the minimum and maximum in one pass, swap the minimum to the head, and the maximum to the tail, hence halve the inner loops.

1: **procedure** SORT(A)
2:      **for** $i \leftarrow 1$ to $\lfloor \frac{|A|}{2} \rfloor$ **do**
3:          $min \leftarrow i$
4:          $max \leftarrow |A| + 1 - i$
5:          **if** $A[max] < A[min]$ **then**
6:              EXCHANGE $A[min] \leftrightarrow A[max]$
7:          **for** $j \leftarrow i + 1$ 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 $A[|A| + 1 - i] \leftrightarrow A[max]$

It's necessary to swap if the right most element less than the left most one before the inner loop starts. This is because the scan excludes them. We can also implement the
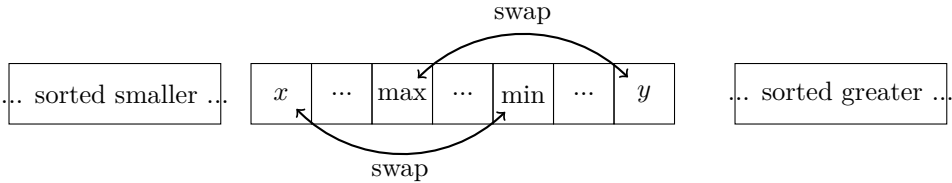
Figure 9.4: Find the minimum and maximum, swap both to the right positions.

cock-tail sort recursively:

1. If the list is empty or singleton, it's sorted;

2. Otherwise, select the minimum and the maximum, move them to the head and tail, then recursively sort the rest elements.

$$
\begin{array}{rcl}
sort\ [\,] & = & [\,] \\
sort\ [x] & = & [x] \\
sort\ xs & = & a : (sort\ xs') + [b], \text{where } (a, b, xs') = min\text{-}max\ xs
\end{array}
\tag{9.8}
$$

Where function *min-max* extracts the minimum and maximum from a list:

$$
min\text{-}max\ (x{:}y{:}xs) = foldr\ sel\ (\min\ x\ y, \max\ x\ y, [\,])\ xs
\tag{9.9}
$$

We initialize the minimum as the first element x_0, the maximum as the second element x_1, and process the list with $foldr$. Define sel 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 *min-max* is bound to $O(n)$ time, $+[b]$ is expensive. As shown in fig. 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.

$$
\begin{array}{rcl}
sort'\ A\ B\ [\,] & = & A + B \\
sort'\ A\ B\ [x] & = & A + (x{:}B) \\
sort'\ A\ B\ (x{:}xs) & = & sort'\ (A + [x_0])\ xs'\ (x_1{:}B), \text{where } (x_0, x_1, xs') = min\text{-}max\ xs
\end{array}
\tag{9.10}
$$

Start sorting with empty accumulators: $sort = sort'\ [\,]\ [\,]$. The append only happens to $A + [x_0]$, while x_1 is linked before B. To further eliminate the $+[x_0]$, we maintain A in reversed order: $\overleftarrow{A}$, hence x_0 is preprended but not appended. We have the following equations:

$$
\begin{array}{rcl}
A' & = & A + [x] \\
& = & reverse\ (x : reverse\ A) \\
& = & reverse\ (x : \overleftarrow{A}) \\
& = & x : \overleftarrow{\overleftarrow{A}}
\end{array}
\tag{9.11}
$$

Finally, reverse $\overleftarrow{A'}$ back to A'. We improve the algorithm as:

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

9.3   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.

9.3.1   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 the team would beat others except for the champion, as shown in fig. 9.5.
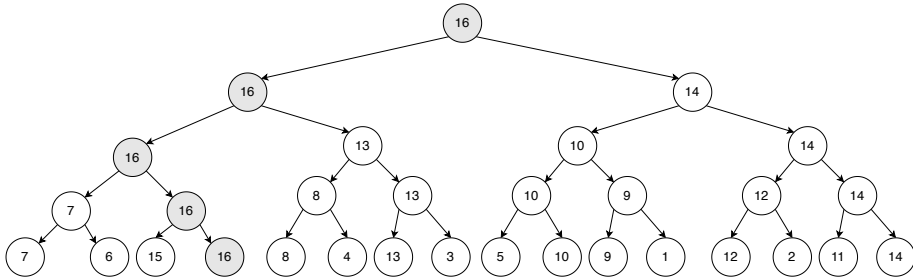
Figure 9.5: The element 15 is knocked out in the first round.

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. Then apply it to select the 3rd, the 4th, ... to sort. We can overwrite 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 fig. 9.5, compare with $[14, 13, 7, 15]$. This method is defined as below:

1. Build a tournament tree with the maximum (the champion) at the root;

2. Extract 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.
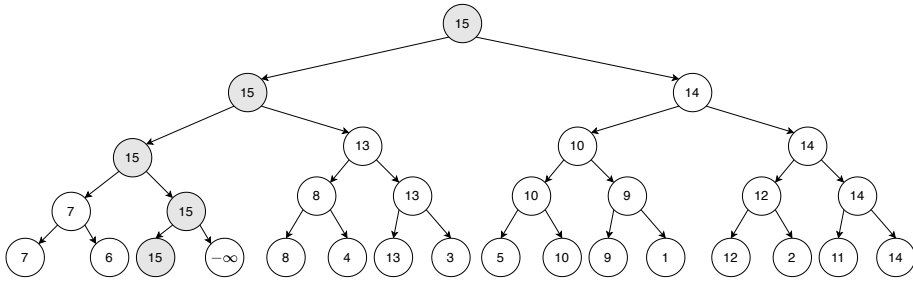
Take 16, replace with $-\infty$, 15 becomes the new root.
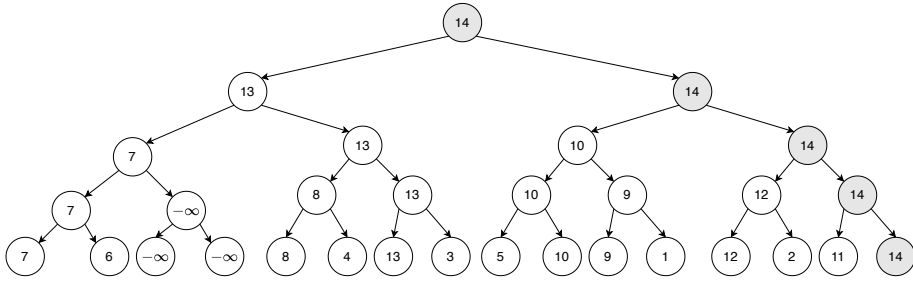
Take 15, replace with $-\infty$, 14 becomes the new root.
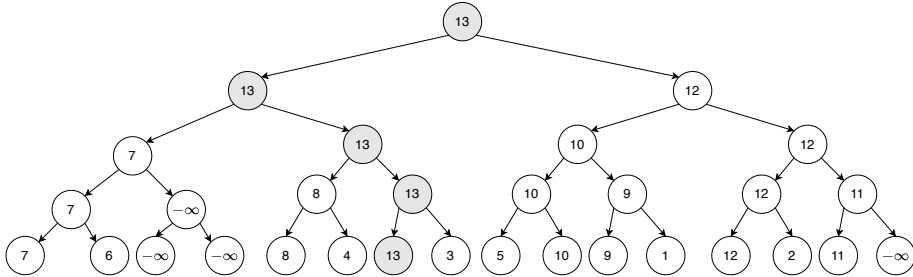
Take 14, replace with $-\infty$, 13 becomes the new root.

Figure 9.6: The first 3 steps of tournament tree sort.

To sort a collection of elements, we build a tournament tree, repeatedly select the champion out. Figure 9.6 gives the first 3 steps. Re-use the binary tree definition. To make back-track easy, we add a parent field in each node. When n is not 2^m for some natural number m, there is remaining element without "player", and directly enters the next round. 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. Enter the next round if there is a tree left (odd number of trees). The trees halve in every round $\lfloor \frac{n}{2} \rfloor$, $\lfloor \frac{n}{4} \rfloor$, ... till the final tournament tree. This process is bound to $O(n + \frac{n}{2} + \frac{n}{4} + ...) = O(2n) = O(n)$ time.

1: **function** BUILD-TREE(A)
2:     $T \leftarrow [\ ]$

```
 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]
```

When pop, 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 processes 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. the 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], T) ← POP(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. Convert the n elements to a list of 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 set t_1, t_2 as the sub-trees.

$$merge\ t_1\ t_2 = (t_1, \max\ k_1\ k_2, t_2) \tag{9.13}$$

Where $k_1 = key\ t_1, k_2 = key\ t_2$ respectively. Define $build\ ts$ to repeatedly merge trees, and build the final tournament tree.

$$\begin{aligned}
build\ [\,] &= \varnothing \\
build\ [t] &= t \\
build\ ts &= build\ (pairs\ ts)
\end{aligned} \tag{9.14}$$

Where:

$$\begin{aligned}
pairs\ (t_1\!:\!t_2\!:\!ts) &= (merge\ t_1\ t_2) : pairs\ ts \\
pairs\ ts &= ts
\end{aligned} \tag{9.15}$$

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, and replace it with $-\infty$.

$$\begin{aligned}
pop\ (\varnothing, k, \varnothing) &= (\varnothing, -\infty, \varnothing) \\
pop\ (l, k, r) &= \begin{cases}
k = key\ l: & (l', \max\ (key\ l')\ (key\ r), r), \text{where } l' = pop\ l \\
k = key\ r: & (l, \max\ (key\ l)\ (key\ r'), r'), \text{where } r' = pop\ r
\end{cases}
\end{aligned} \tag{9.16}$$

Then repeatedly pop the tournament tree to sort (in descending order):

$$\begin{aligned}
sort\ \varnothing &= [\,] \\
sort\ (l, -\infty, r) &= [\,] \\
sort\ t &= (key\ t) : sort\ (pop\ t)
\end{aligned} \tag{9.17}$$

Exercise 9.2

9.2.1. Implement the recursive tournament tree sort in ascending order.

9.2.2. How to handle duplicated elements with the tournament tree? is tournament tree sort stable?

9.2.3. Compare the tournament tree sort and binary search tree sort in terms of space and time performance.

9.2.4. Compare heap sort and tournament tree sort in terms of space and time performance.

9.3.2   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 is still room for improvement. After sort, the binary tree holds all $-\infty$, occupying $2n$ nodes for n elements. It's there a way to release nodes 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 eq. (9.17) in a generic way:

$$\begin{aligned}
sort\ \varnothing &= [\,] \\
sort\ t &= (top\ t) : sort\ (pop\ t)
\end{aligned} \tag{9.18}$$

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.

9.4   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

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. The binomial heap is a forest of k-ary trees. When delay some operations of the Binomial heap, we obtain the Fibonacci heap. It improves the heap merge performance from $O(\lg n)$ to amortized constant time, which is critical for graph algorithm design. The pairing heap gives a simplified implementation with good overall performance.

10.1 Binomial Heaps

Binomial heap is named after Newton's binomial theorem. It consists of a set of k-ary trees (a forest). Every tree has the size equal to a binomial coefficient. Newton proves that $(a + b)^n$ expands to:

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

When n is a natural number, the list of 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 ways to generate pascal triangle, like recursion.

10.1.1 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]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 expanded it to real exponents.

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 fig. 10.1.
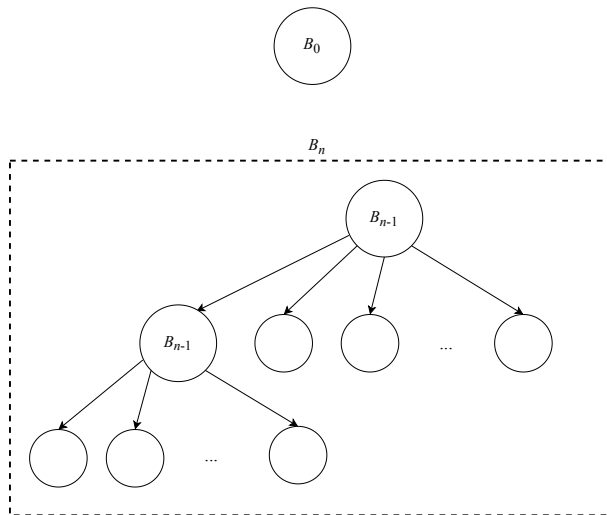
Figure 10.1: Binomial tree

Figure 10.2 gives examples of B_0 to B_4. The number of nodes in every row in B_n is a binomial coefficient. For example in B_4, there is a node (the 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 further know there are 2^n elements in a B_n tree.

A binomial heap is a set of binomial trees (a forest) satisfying 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 the 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 $n = (a_m...a_1, a_0)_2$, where a_0 is the least significant bit (LSB) and a_m is the most significant bit (MSB). There is a tree of rank i if and only if $a_i = 1$. For example, consider a binomial heap of 5 elements. As $5 = (101)_2$ in binary, there are 2 binomial trees: B_0 and B_2. The binomial heap in fig. 10.3 has 19 elements, $19 = (10011)_2$. There are three trees: B_0, B_1, and B_4.

We define the binomial tree as (r, k, ts), where r is the rank, k is the root element, and ts is the list of sub-trees ordered by rank.

```
data BiTree a = Node Int a [BiTree a]
type BiHeap a = [BiTree a]
```

There is a method called 'left-child, right-sibling'[4], that reuses the binary tree data structure to define multi-ways tree. Every node has the left and right parts. the left references to the first sub-tree; the right references to its sibling. All siblings form a list as shown in fig. 10.4. Alternatively, we can use an array or a list to hold the sub-trees.
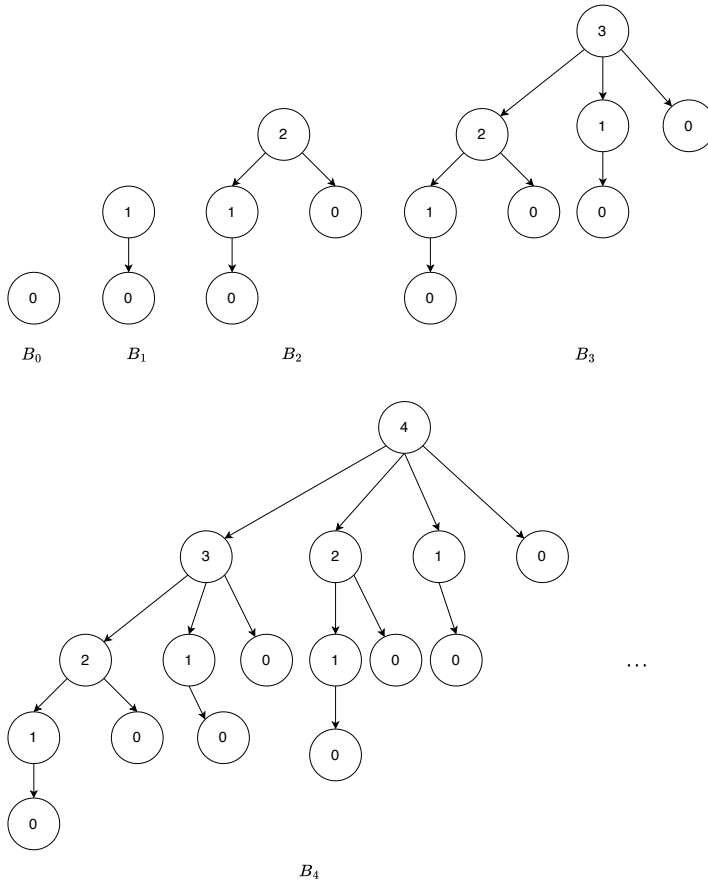
Figure 10.2: Binomial trees of rank 0, 1, 2, 3, 4, ...
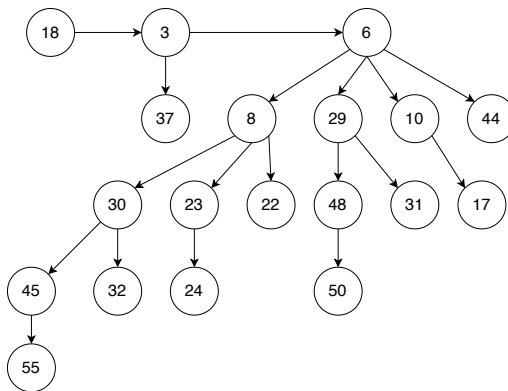
Figure 10.3: A binomial heap with 19 elements
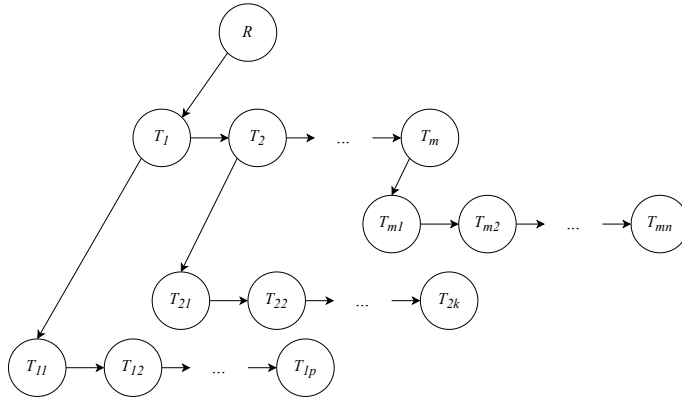
Figure 10.4: R is the root, $T_1, T_2, ..., T_m$ are sub-trees of R. The left of R is T_1, the right is NIL. $T_{11}, ..., T_{1p}$ are sub-trees of T_1. The left of T_1 is T_{11}, the right is its sibling T_2. The left of T_2 is T_{21}, the left is sibling.

10.1.2   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 fig. 10.5.

$$link \ (r, x, ts) \ (r, y, ts') = \begin{cases} x < y : & (r + 1, x, (r, t, ts'){:}ts) \\ \text{otherwise} : & (r + 1, y, (r, x, ts){:}ts') \end{cases} \tag{10.2}$$
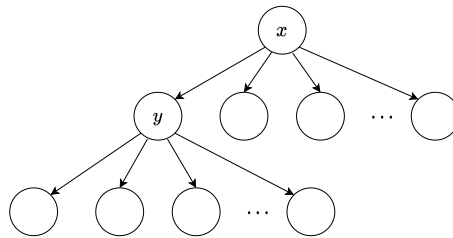
Figure 10.5: If $x < y$, link y as the first sub-tree of x.

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 ↔ y
4:     SIBLING(y) ← SUB-TREES(T₁)
5:     SUB-TREES(x) ← y
6:     PARENT(y) ← x
7:     RANK(x) ← RANK(y) + 1
8:     return x
```

Exercise 10.1

10.1.1. Write a program to generate Pascal's triangle.

10.1.2. Prove that the i-th row in tree B_n has $\binom{n}{i}$ nodes.

10.1.3. Prove there are 2^n elements in a B_n tree.

10.1.4. Use a container to store sub-trees, how to implement link? How to ensure it is in constant time?

10.1.3 Insert

When insert a tree, we keep the forest ordered by rank (ascending):

$$
\begin{aligned}
ins\ t\ [\ ] \quad &= \quad [t] \\
ins\ t\ (t':ts) \quad &= \quad
\begin{cases}
rank\ t < rank\ t' : & t:t':ts \\
rank\ t' < rank\ t : & t' : ins\ t\ ts \\
\text{otherwise} : & ins\ (link\ t\ t')\ ts
\end{cases}
\end{aligned}
\tag{10.3}
$$

Where $rank\ (r, k, ts) = r$ returns the rank of a tree. For empty heap $[\ ]$, it becomes a singleton of $[t]$; otherwise, compare the rank of t with the first tree t', if t has the less rank, then it becomes the new first tree; if t' has the 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. 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 constant time, the overall performance is bound to $O(\lg n)^2$. We define $insert$ for binomial heap with ins. First wrap the new element x in a singleton tree, then insert to the heap:

$$
insert\ x = ins\ (0, x, [\ ])
\tag{10.4}
$$

This is in Curried form. We can further insert a list of elements with *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 and 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(⊥)

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

---

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

10.1.4   Merge

When merge two binomial heaps, we actually merge two lists of binomial trees. Every tree has the unique rank in the merged result in ascending order. The tree merge process is similar to merge sort (see chapter 13). 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.

$$
\begin{aligned}
merge\ ts_1\ [\ ] &= ts_1 \\
merge\ [\ ]\ ts_2 &= ts_2 \\
merge\ (t_1{:}ts_1)\ (t_2{:}ts_2) &=
\begin{cases}
rank\ t_1 < rank\ t_2: & t_1 : (merge\ ts_1\ (t_2{:}ts_2)) \\
rank\ t_2 < rank\ t_1: & t_2 : (merge\ (t_1{:}ts_1)\ ts_2) \\
otherwise: & ins\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2)
\end{cases}
\end{aligned}
\tag{10.6}
$$

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 eliminate recursion, and implement the 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 are 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 $O(\lg n)$.

10.1.5   Pop

Although every tree has the minimal element in its root, we don't know which tree holds the overall minimum of 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.

$$top\ (t{:}ts) = foldr\ f\ (key\ t)\ ts \quad \text{where}\ f\ (r, x, ts)\ y = \min\ x\ y \tag{10.7}$$

```
1: function TOP(H)
2:     m ← ∞
3:     while H ≠ NIL do
4:         m ← MIN(m, KEY(H))
5:         H ← SIBLING(H)
6:     return m
```

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 fig. 10.6. To support pop, we need extract the tree containing the minimum out:
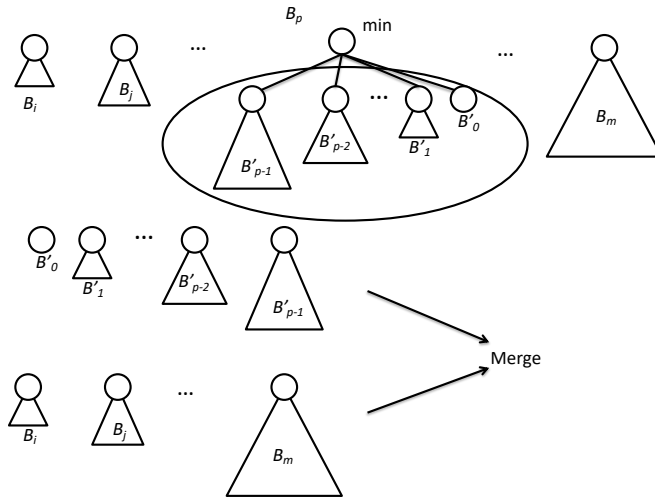
Figure 10.6: Binomial heap pop.

$$min' \ [t] \ = \ (t, [\ ])$$
$$min' \ (t{:}ts) \ = \ \begin{cases} key \ t < key \ t' : & (t, ts), \text{where } (t', ts') = min' \ ts \\ \text{otherwise} : & (t', t{:}ts') \end{cases} \qquad (10.8)$$

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:

$$pop \ H = merge \ (reverse \ ts) \ H', \text{where } ((r, k, ts), H') = min' \ H \qquad (10.9)$$

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:    $H' \leftarrow H, p \leftarrow \text{NIL}$
3:    $T_m \leftarrow T_p \leftarrow \text{NIL}$
4:    **while** $H \neq \text{NIL}$ **do**
5:        **if** $T_m = \text{NIL}$ or $\text{KEY}(H) < \text{KEY}(T_m)$ **then**
6:            $T_m \leftarrow H$
7:            $T_p \leftarrow p$
8:        $p \leftarrow H$
9:        $H \leftarrow \text{SIBLING}(H)$
10:    **if** $T_p \neq \text{NIL}$ **then**
11:        $\text{SIBLING}(T_p) \leftarrow \text{SIBLING}(T_m)$
12:    **else**
13:        $H' \leftarrow \text{SIBLING}(T_m)$
14:    $\text{SIBLING}(T_m) \leftarrow \text{NIL}$
15:    **return** (T_m, H')

We can implement heap sort with *pop*. First build a binomial heap from a list, then repeatedly pop the minimum.

$$sort = heapSort \circ fromList \tag{10.10}$$

Where *heapSort* is defined as:

$$\begin{aligned} heapSort\ [\ ] &= [\ ] \\ heapSort\ H &= (top\ H) : heapSort\ (pop\ H) \end{aligned} \tag{10.11}$$

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.

10.2   Fibonacci heap

Binomial heap is named after the 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].

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

When insert x to a binomial heap, we wrap it to a single tree, then insert to the forest. We maintain the rank ordering. If two ranks are same, 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 singleton tree of x directly to the forest. To access the top element in constant time, we need record which tree has the overall minimum. A

---

[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]

Fibonacci heap is either empty $\varnothing$, or a forest of trees denoted as (n, t_m, ts). Where n is the number of elements in the heap, t_m is the tree holds the top element, and ts is the rest trees. Below example defines Fibonacci heap (reuse the definition of binomial tree).

```
data FibHeap a = Empty | FibHeap { size :: Int
                                 , minTree :: BiTree a
                                 , trees :: [BiTree a]}
```

We can access the top in constant time (Curried form): $top = key \circ minTree$.

10.2.1 Insert

We define insert as a special case of merge: one heap is a singleton tree: $insert\ x\ H = merge\ (singleton\ x)\ H$. Or simplified in Curried form:

$$insert\ x = merge \circ (singleton\ x) \tag{10.12}$$

Where:

$$singleton\ x = (1, (1, x, [\ ]), [\ ])$$

Below is the imperative implementation:

1: **function** INSERT(k, H)
2: $\quad x \leftarrow$ SINGLETON(k) $\qquad\qquad\qquad\qquad\qquad\qquad$ ▷ 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: $\qquad$ 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.

10.2.2 Merge

When merge two heaps, we delay the link, but only put the trees together, then pick the new top.

$$
\begin{array}{rcl}
merge\ h\ \varnothing & = & h \\
merge\ \varnothing\ h & = & h \\
merge\ (n, t_m, ts)\ (n', t'_m, ts') & = & \begin{cases} key\ t_m < key\ t'_m : & (n + n', t_m, t'_m : ts + ts') \\ otherwise : & (n + n', t'_m, t_m : ts + ts') \end{cases}
\end{array}
\tag{10.13}
$$

When neither tree is empty, the $+$ takes time that is proportion to the number of trees in one heap. We can improve it to constant time with doubly linked-list as 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(H₁, H₂)
 2:     H ← FIB-HEAP
 3:     TREES(H) ← CONCAT(TREES(H₁), TREES(H₂))
 4:     if KEY(MIN-TREE(H₁)) < KEY(MIN-TREE(H₂)) then
 5:         MIN-TREE(H) ← MIN-TREE(H₁)
 6:     else
 7:         MIN-TREE(H) ← MIN-TREE(H₂)
        SIZE(H) = SIZE(H₁) + SIZE(H₂)
 8:     return H

 9: function CONCAT(s₁, s₂)
10:     e₁ ← PREV(s₁)
11:     e₂ ← PREV(s₂)
12:     NEXT(e₁) ← s₂
13:     PREV(s₂) ← e₁
14:     NEXT(e₂) ← s₁
15:     PREV(s₁) ← e₂
16:     return s₁
```

10.2.3 Pop

As the *merge* function delays the link, we need 'compensate' it during pop. We define it as tree consolidation. Consider another problem: given a list of numbers of 2^m (for some integer $m \geq 0$), for e.g., $L = [2, 1, 1, 4, 8, 1, 1, 2, 4]$, we repeatedly sum two equal numbers until every one is unique. The result is $[8, 16]$, as 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 to the next step. We can define the consolidation 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:

$$
\begin{aligned}
melt \; x \; [\,] &= x \\
melt \; x \; (x':xs) &= \begin{cases} x = x' : & melt \; 2x \; xs \\ x < x' : & x:x':xs \\ x > x' : & x' : melt \; x \; xs \end{cases}
\end{aligned} \tag{10.15}
$$

Let $n = sum \; L$. *consolidate* actually represents n in binary format. The result contains 2^i (i starts from 0) if and only if the i-th bit is 1. For e.g., $sum[2, 1, 1, 4, 8, 1, 1, 2, 4] = 24$.

It's $(11000)_2$ 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 trees:

$$
\begin{aligned}
melt\ t\ [\,] &= [t] \\
melt\ t\ (t':ts) &= \begin{cases} rank\ t = rank\ t' : & melt\ (link\ t\ t')\ ts \\ rank\ t < rank\ t' : & t:t':ts \\ rank\ t > rank\ t' : & t' : melt\ t\ ts \end{cases}
\end{aligned}
\tag{10.16}
$$

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.
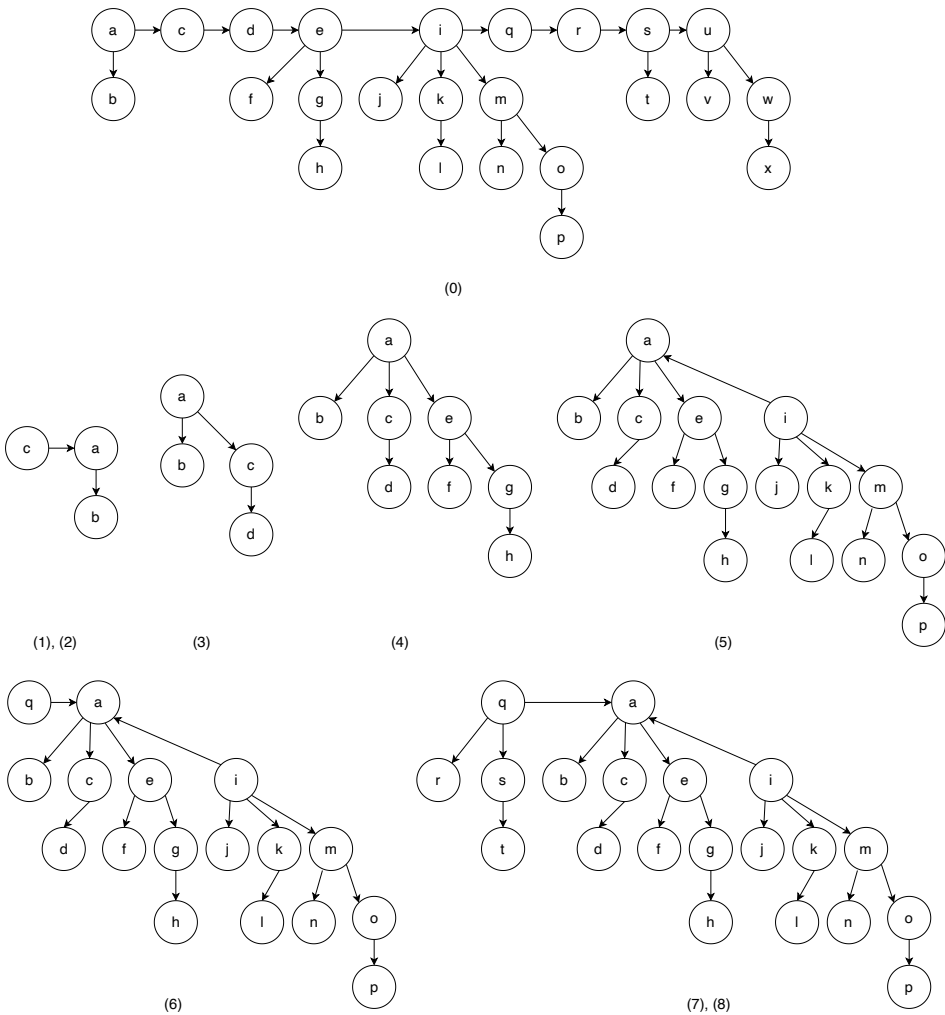
Figure 10.7: Consolidation. Step 3, link d and c, then link a; Step 7, 8, link r and q, then link s and q.

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:      Tₘ ← 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 Tₘ = NIL or KEY(T) < KEY(Tₘ) then
18:                  Tₘ ← T
19:      MIN-TREE(H) ← Tₘ
```

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 eq. (10.8) to extract the min-tree.

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

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

```
1: function POP(H)
2:      Tₘ ← MIN-TREE(H)
3:      for each T in SUB-TREES(Tₘ) do
4:          append T to TREES(H)
5:          PARENT(T) ← NIL
6:      remove Tₘ from TREES(H)
7:      SIZE(H) ← SIZE(H) - 1
8:      CONSOLIDATE(H)
9:      return (KEY(Tₘ), 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 fig. 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:

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

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
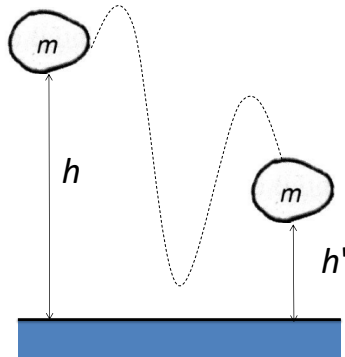
Figure 10.8: Gravity potential energy.

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:

$$\begin{aligned} T &= T_c + \Phi(H') - \Phi(H) \\ &= O(R(n) + t(H) - 1) + (R(n) + 1) - t(H) \\ &= O(R(n)) \end{aligned} \tag{10.19}$$

Insert, merge, and pop ensure all trees are binomial trees, therefore, the upper bound of $R(n)$ is $O(\lg n)$.

10.2.4 Increase priority

We can use heap to manage tasks with priority. When need prioritize a task, we decrease (for the min heap) the value, making it close to the heap top. Some graph algorithms, like the minimum spanning tree and Dijkstra's algorithm require this heap operation[4] meet amortized constant time. Let x be a node, we need decrease its value to k. As shown in fig. 10.9, if x is less than its parent y (in terms of key), we cut x off y, then add it to the heap (forest). Although the parent still holds the minimum of the tree, it is not a 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).*

1: **function** DECREASE(H, x, k)
2:     KEY(x) ← k
3:     p ← PARENT(x)
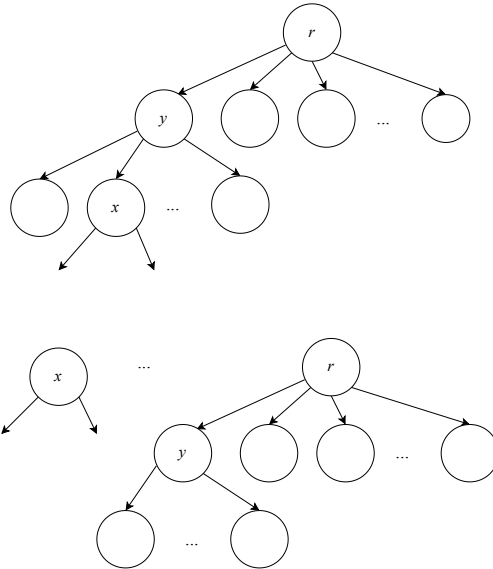
Figure 10.9: If *key x < key y*, cut *x* off and add to the heap.

```
4:     if p ≠ 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) ← 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 ← PARENT(x)
3:     remove x from p
4:     RANK(p) ← RANK(p) - 1
5:     add x to TREES(H)
6:     PARENT(x) ← 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

Why is DECREASE bound to amortized $O(1)$ time?

10.2.5   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 10.2.1. *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:

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_k &= F_{k-1} + F_{k-2}
\end{aligned}
$$

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 earliest, 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 an additional sub-tree at most, (through the DECREASE). Once loss the second sub-tree, it will be cut off then added 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.

$$
\begin{aligned}
|x| &\geq s_k \\
&= 2 + s_{rank(y_2)} + s_{rank(y_3)} + ... + s_{rank(y_k)} \\
&\geq 2 + s_0 + s_1 + ... + s_{k-2}
\end{aligned}
$$

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 $k \geq 2$.

$$
\begin{aligned}
|x| &\geq s_k \\
&\geq 2 + s_0 + s_1 + ... + s_{k-2} \\
&\geq 2 + F_2 + F_3 + ... + F_k & \text{induction hypothesis} \\
&= 1 + F_0 + F_1 + F_2 + ... + F_k & F_0 = 0, F_1 = 1
\end{aligned}
$$

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 $k + 1$.

$$
\begin{aligned}
F_{k+2} &= F_{k+1} + F_k \\
&= (1 + \sum_{i=0}^{k-1} F_i) + F_k \quad \text{induction hypothesis} \\
&= 1 + \sum_{i=0}^{k} F_i
\end{aligned}
$$

Wrap up to the final result:

$$n \geq |x| \geq F_{k+2} \tag{10.22}$$

$\square$

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-RANK from Fibonacci numbers:

1: **function** MAX-RANK(n)
2:     $F_0 \leftarrow 0, F_1 \leftarrow 1$
3:     $k \leftarrow 2$
4:     **repeat**
5:         $F_k \leftarrow F_{k_1} + F_{k_2}$
6:         $k \leftarrow k + 1$
7:     **until** $F_k < n$
8:     **return** $k - 2$

10.3 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].

10.3.1 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 = Empty | Node a [PHeap a]
```

10.3.2 Merge, insert, and top

There are two cases for merge:

1. Either heap is $\varnothing$, the result is the other heap;

2. Otherwise, compare the two roots, set the greater one as the new sub-tree of the other.

$$\begin{aligned}
merge\ \varnothing\ h_2 &= h_2 \\
merge\ h_1\ \varnothing &= h_1 \\
merge\ (x, ts_1)\ (y, ts_2) &= \begin{cases} x < y: & (x, (y, ts_2){:}ts_1) \\ \text{otherwise}: & (y, (x, ts1){:}ts_2) \end{cases}
\end{aligned} \qquad (10.24)$$

merge is in constant time. Below is the imperative implementation with the 'left-child, right sibling' method:

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) ← LINK(H_2, SUB-TREES(H_1))
9:     PARENT(H_2) ← H_1
10:     **return** H_1

Similar to Fibonacci heap, we implement insert with merge as eq. (10.12), access the top from the root: $top\ (x, ts) = x$. Both operations are in constant time.

10.3.3   decrease key

When decrease the value of a node, if it is the root, then directly alter the value; otherwise, cut the sub-tree rooted at this node, then merge it back to the heap.

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

10.3.4   Pop

After pop the root, we consolidate the sub-trees to a tree:

$$pop\ (x, ts) = consolidate\ ts \qquad (10.25)$$

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 name of 'paring heap'. Figures 10.10 and 10.11 show the paired merge.

$$\begin{aligned}
consolidate\ [\,] &= \varnothing \\
consolidate\ [t] &= t \\
consolidate\ (t_1{:}t_2{:}ts) &= merge\ (merge\ t_1\ t_2)\ (consolidate\ ts)
\end{aligned} \qquad (10.26)$$

The corresponding 'left child, right sibling' implementation is as below:

1: **function** POP(H)
2:     L ← NIL
3:     **for** every T_x, T_y in SUB-TREES(H) **do**
4:         T ← MERGE(T_x, T_y)
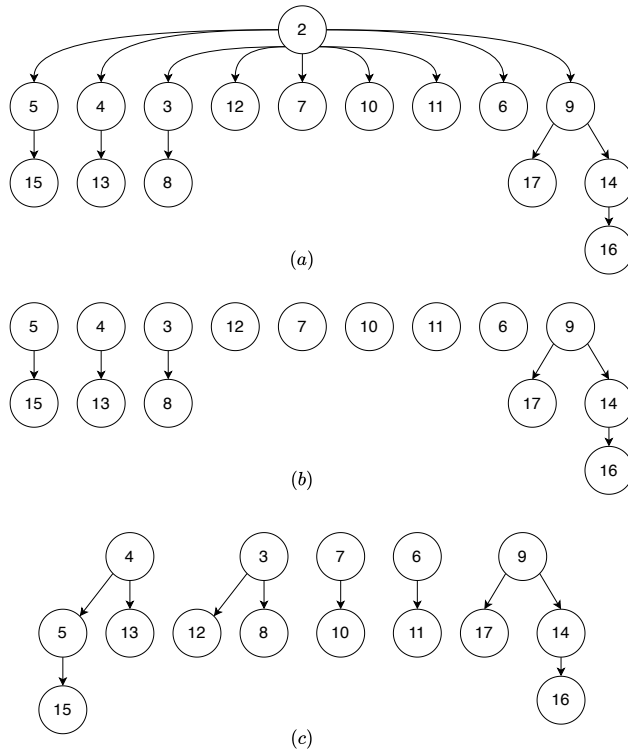
Figure 10.10: Pop the root, merge the 9 sub-trees in pairs, leave the last tree.

5:       $L \leftarrow \text{LINK}(T, L)$
6:    $H \leftarrow \text{NIL}$
7:    **for** T **in** L **do**
8:        $H \leftarrow \text{MERGE}(H, T)$
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 traverse from right to left. If there are odd number of sub-trees, $T_y = $ NIL at last, hence $T = T_x$ in this case.

10.3.5   Delete

To delete a node x, first decrease the value of x to $-\infty$, then followed with a pop. Alternatively, if x is the root, pop it; otherwise, cut x off H, then $pop(x)$, and merge it back to H:

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

As delete is implemented with pop, the performance is conjectured to be amortized $O(\lg n)$ time.
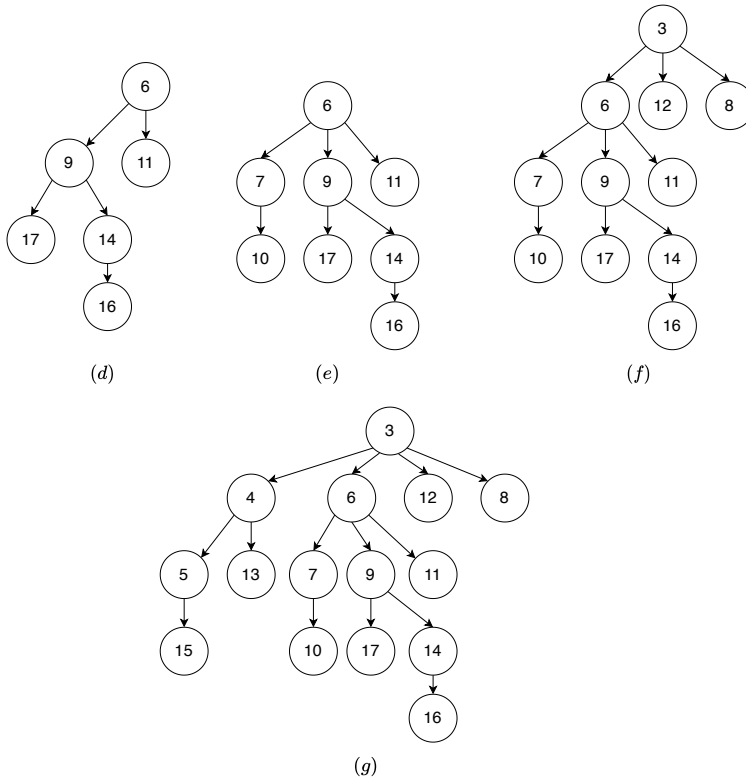
Figure 10.11: Merge from right to left. (d) merge 9, 6; (e) merge 7; (f) merge 3; (g) merge 4.

Exercise 10.3

10.3.1. If continuously insert n elements then followed with a pop. The performance overhead is big when n is a large number (although the amortized performance is $O(\lg n)$). How to mitigate such worst case?

10.3.2. Implement *delete* for the pairing heap.

10.3.3. Implement DECREASE-KEY for the pairing heap.

10.4   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:

```haskell
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:

```haskell
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 (FibHeap _ (Node _ x []) []) = (x, Empty)
pop (FibHeap sz (Node _ x tsm) ts) = (x, FibHeap (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

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 gives 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.

11.1   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 to traverse and reach the tail, where n is the length. To achieve the constant time performance goal, we need an additional variable to record the tail position, and use a sentinel node S to simplify the empty queue handling, as shown in fig. 11.1.

```
data Node<K> {
  K key
  Node<K> next
}

data Queue<K> {
  Node<K> head, tail
}
```
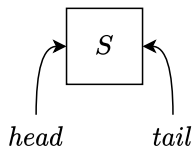

Figure 11.1: Both head and tail point to S for empty queue.

Define 'enqueue' (also called push, snoc, append, or push back) and 'dequeue' (also called pop, or pop front) to add and remove element respectively. 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 fig. 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].
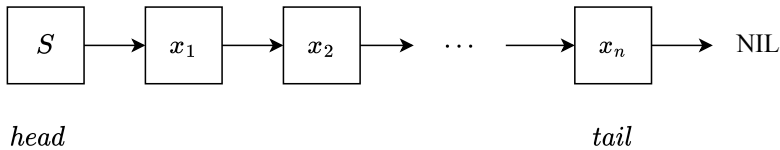
Figure 11.2: List with S node.

11.2   Circular buffer

Symmetrically, we can append element to the tail of array in constant time, but it takes $O(n)$ time to remove 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 remove from head, as shown figs. 11.3 and 11.4. 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.
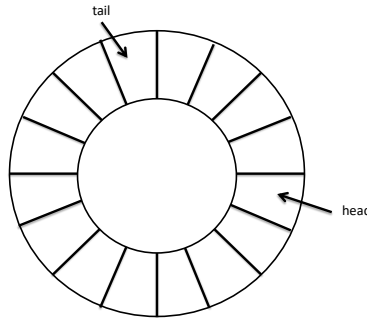
Figure 11.3: Circular buffer.

1: **function** ENQUEUE(Q, x)

(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.
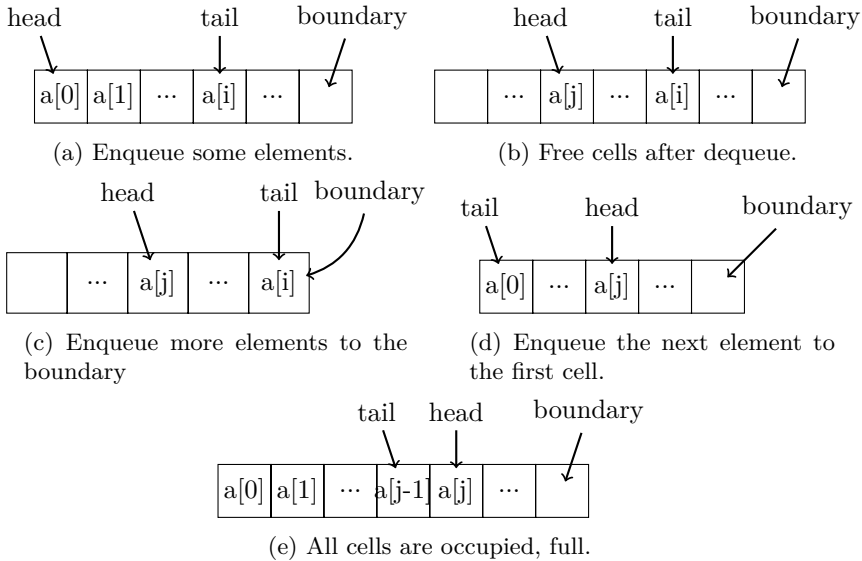
Figure 11.4: Circular buffer queue

```
2:      if not FULL(Q) then
3:          COUNT(Q) ← COUNT(Q) + 1
4:          tail ← (HEAD(Q) + COUNT(Q)) mod SIZE(Q)
5:          BUF(Q)[tail] ← x
```

```
1: function DEQUEUE(Q)
2:      x ← NIL
3:      if not EMPTY(Q) then
4:          h ← HEAD(Q)
5:          x ← BUF(Q)[h]
6:          HEAD(Q) ← (h + 1) mod SIZE(Q)
7:          COUNT(Q) ← COUNT(Q) - 1
8:      return x
```

Exercise 11.1

11.1.1. The circular buffer is allocated with a predefined size. We can use two references, head and tail instead. How to determine if a circular buffer queue is full or empty? (the head can be either ahead of tail or behind it.)

11.3 Paired-list queue

We can access the list head in constant time, but need linear time to access the tail. We connect two lists 'tail to tail' to implement queue, as shown in fig. 11.5. Define the 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.

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

f may become empty after a series of pops, while r still contains elements. To continue
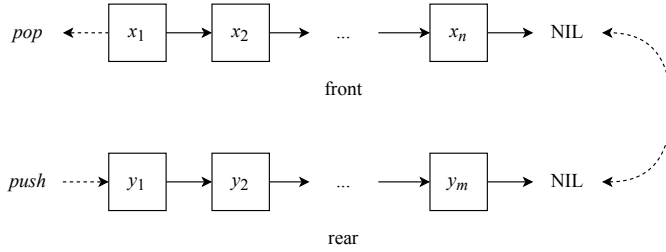
Figure 11.5: paired-list queue.

pop, we reverse r to replace f, i.e., $([\,],r) \mapsto (reverser,[\,])$. We need check and balance in every push/pop:

$$\begin{array}{rcl} balance\ [\,]\ r & = & (reverse\ r,[\,]) \\ balance\ f\ r & = & (f,r) \end{array} \qquad (11.2)$$

Although the time is bound to linear time when reverse r, the amortised performance is constant time. Update push/pop as below:

$$\begin{cases} push\ x\ (f,r) & = balance\ f\ (x{:}r) \\ pop\ (x{:}f,r) & = balance\ f\ r \end{cases} \qquad (11.3)$$

There is a symmetric implementation with a pair of arrays (table 11.1). Connect two arrays head to head, as shown in fig. 11.6. When R becomes empty, reverse array F to replace R.

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
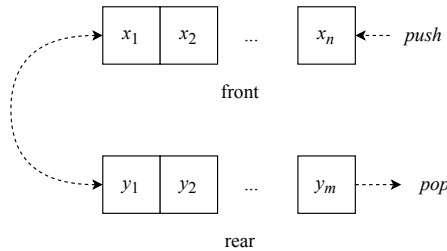
Figure 11.6: paired-array queue.

Exercise 11.2

11.2.1. Why need balance check and adjustment after push?

11.2.2. Do the amortized analysis for the paired-list queue.

11.2.3. Implement the paired-array queue.

11.4   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. f and r are unbalance in this case. To solve it, we add another rule: the length of r is not greater than f, otherwise 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 eq. (11.4), we only need check the length of f to test if a queue is empty:

$$Q = \phi \iff n = 0 \tag{11.5}$$

Change the definition of push/pop 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:

$$balance \ (f, n, r, m) = \begin{cases} m \leq n : & (f, n, r, m) \\ \text{otherwise} : & (f \mathbin{+\!\!+} reverse \ r, m + n, [\,], 0) \end{cases} \tag{11.7}$$

11.5   Real-time queue

It still takes linear time to reverse and concatenate lists in balanced queue. The real-time queue need guarantee constant time in every push/pop operation. The performance bottleneck happens in $f \mathbin{+\!\!+} reverse \ r$. At this time, $m > n$, 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 the computation to multiple push and pop operations. Revisit the tail recursive [61] [62] reverse (in Curried form):

$$reverse \ = \ reverse' \ [\,] \tag{11.8}$$

where:

$$\begin{array}{rcl} reverse' \ a \ [\,] & = & a \\ reverse' \ a \ (x{:}xs) & = & reverse' \ (x{:}a) \ xs \end{array} \tag{11.9}$$

We turn the tail recursive implementation to stepped computation. Model it as a series of state transformations. 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, link the head x ahead of a. Every step terminates, but not continues running recursively. The new state with the intermediate reverse result is input to the next *step*. For example:

$$
\begin{aligned}
step\ S_r\ \text{``hello''}\ [\,] &= (S_r, \text{``ello''}, \text{``h''}) \\
step\ S_r\ \text{``ello''}\ \text{``h''} &= (S_r, \text{``llo''}, \text{``eh''}) \\
&\cdots \\
step\ S_r\ \text{``o''}\ \text{``lleh''} &= (S_r, [\,], \text{``olleh''}) \\
step\ S_r\ [\,]\ \text{``olleh''} &= (S_f, \text{``olleh''})
\end{aligned}
$$

However, it only solves half problem. We also need slow-down $+\!\!+$ computation, which is more complex. Use state machine again. To concatenate $xs +\!\!+ ys$, we first reverse xs to $\overleftarrow{xs}$, then pick elements from $\overleftarrow{xs}$ one by one, and link each ahead of ys. The idea is similar to $reverse'$:

$$
\begin{aligned}
xs +\!\!+ ys &= (reverse\ reverse\ xs) +\!\!+ ys \\
&= (reverse'\ [\,]\ \overleftarrow{xs}) +\!\!+ ys \qquad\qquad (11.11) \\
&= reverse'\ ys\ \overleftarrow{xs}
\end{aligned}
$$

We need add another state: after reverse r, 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. Stepped reverse f to $\overleftarrow{f}$, and r to $\overleftarrow{r}$ in parallel;

2. Stepped take elements from $\overleftarrow{f}$, and link each ahead of $\overleftarrow{r}$.

$$
\begin{aligned}
next\ (S_r, f', x{:}f, r', y{:}r) &= (S_r, x{:}f', f, y{:}r', r) & \text{reverse } f, r \\
next\ (S_r, f', [\,], r', [y]) &= next\ (S_c, f', y{:}r') & \text{reverse done, start concatenation} \\
next\ (S_c, a, [\,]) &= (S_f, a) & \text{done} \\
next\ (S_c, a, x{:}f') &= (S_c, x{:}a, f') & \text{concatenation}
\end{aligned}
$$

$$(11.12)$$

We need arrange these steps to each push/pop. From the balance rule, when $m = n{+}1$, triggers $f +\!\!+ reverse\ r$. it takes $n + 1$ steps to reverse r. Within these steps, we reverse f in parallel. After that, it takes another $n + 1$ steps to concatenate, in total $2n + 2$ steps. The critical question is: Before complete the $2n + 2$ steps, will the queue become unbalanced again due to a series of push/pop operations?

Luckily, repeat pushing won't break the balance rule again before we complete $f +\!\!+ reverse\ r$. We will obtain a new front list $f' = f +\!\!+ reverse\ r$ after $2n + 2$ steps, while the time to break the balance rule again is:

$$
\begin{aligned}
|r'| &= |f'| + 1 \\
&= |f| + |r| + 1 \qquad\qquad (11.13) \\
&= 2n + 2
\end{aligned}
$$

Thanks to the balance rule. Even repeat pushing as many as possible, the $2n + 2$ steps are guaranteed to be completed before the next time when the queue is unbalanced, hence the new f will be ready. We can safely start to compute $f' +\!\!+ reverse\ r'$.

However, pop may happen before the completion of $2n + 2$ steps. We are facing the situation that needs to extract element from f, while the new front list $f' = f +\!\!+ reverse\ r$ hasn't been ready yet. To solve this issue, we duplicate a copy of f when doing $reverse\ f$. We are safe even repeat pop for n times. Table 11.2 shows the queue during phase 1 (reverse f and r in parallel)[1].

The copy of f is exhausted after repeating n pops. We are about to start stepped concatenation. What if pop happens at this time? Since f is exhausted, it becomes $[\,]$. We needn't concatenate anymore. This is because $f +\!\!+ \overleftarrow{r} = [\,] +\!\!+ \overleftarrow{r} = \overleftarrow{r}$. In fact,

---

[1] Although it takes linear time to duplicate a list, however, the one-shot copying won't happen at all. We actually duplicate the reference to the front list, and delay the element level copying to each step

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.

we only need to concatenate the elements in f that haven't been popped. Because we pop from the head of f, let us use a counter to record the remaining elements in f. It's initialized as 0. We apply +1 every time when reverse an element. 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, and cancel the process when it is 0. Below is the updated state transformation:

$$
\begin{aligned}
next\ (S_r, n, f', x{:}f, r', y{:}r) &= (S_r, n+1, x{:}f', f, y{:}r', r) & \text{reverse } f, r \\
next\ (S_r, n, f', [\ ], r', [y]) &= next\ (S_c, n, f', y{:}r') & \text{reverse done, start concatenation} \\
next\ (S_c, 0, a, f) &= (S_f, a) & \text{done} \\
next\ (S_c, n, a, x{:}f') &= (S_c, n-1, x{:}a, f') & \text{concatenation} \\
next\ S_0 &= S_0 & \text{idle}
\end{aligned}
$$

$$(11.14)$$

We define an addition idle state S_0 to simplify the logic. The queue contains 3 parts: the front list f with its length n, the state S of on-going f ++ $reverse\ r$, and the rear list r with its length m. Denoted as (f, n, S, r, m). The empty queue is $([\ ], 0, S_0, [\ ], 0)$. The queue is empty when $n = 0$ according to the balance rule. Update the push/pop as:

$$
\begin{cases}
push\ x\ (f, n, S, r, m) &= balance\ f\ n\ S\ (x{:}r)\ (m+1) \\
pop\ (x{:}f, n, S, r, m) &= balance\ f\ (n-1)\ (abort\ S)\ r\ m
\end{cases}
$$

$$(11.15)$$

Where *abort* decreases the counter to cancel an element for concatenation. We'll define it later. *balance* triggers stepped f ++ $reverse\ r$ if the queue is unbalanced, otherwise runs a step:

$$
balance\ f\ n\ S\ r\ m = \begin{cases}
m \le n: & step\ f\ n\ S\ r\ m \\
\text{otherwise}: & step\ f\ (n+m)\ (next\ (S_r, 0, [\ ], f, [\ ], r))\ [\ ]\ 0
\end{cases}
$$

$$(11.16)$$

Where *step* transforms the state machine, ends with the idle state S_0 when completes.

$$step\ f\ n\ S\ r\ m = queue\ (next\ S) \qquad (11.17)$$

Where:

$$
\begin{aligned}
queue\ (S_f, f') &= (f', n, S_0, r, m) & \text{replace } f \text{ with } f' \\
queue\ S' &= (f, n, S', r, m)
\end{aligned}
$$

$$(11.18)$$

Finally, define *abort* to cancel an element:

$$
\begin{aligned}
abort\ (S_c, 0, (x{:}a), f') &= (S_f, a) \\
abort\ (S_c, n, a, f') &= (S_c, n-1, a, f') \\
abort\ (S_r, n, f'f, r'r) &= (S_r, n-1, f', f, r', r) \\
abort\ S &= S
\end{aligned}
$$

$$(11.19)$$

Exercise 11.3

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*?

11.3.2. Implement the real-time queue with paired arrays. We can't copy the array when start rotation, or the performance will downgrade to linear time. Please implement 'lazy' copy, i.e., copy an element per step.

11.6   Lazy real-time queue

The key to realize real-time queue is to break down the expensive $f \mathbin{+\mkern-10mu+} reverse\ r$. We can simplify it with lazy evaluation. Assume function *rotate* computes $f \mathbin{+\mkern-10mu+} reverse\ r$ in steps, i.e., below two functions are equivalent with an accumulator a.

$$rotate\ xs\ ys\ a = xs \mathbin{+\mkern-10mu+} (reverse\ ys) \mathbin{+\mkern-10mu+} a \tag{11.20}$$

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:

$$
\begin{aligned}
&rotate\ (x{:}xs)\ (y{:}ys)\ a \\
={}& (x{:}xs) \mathbin{+\mkern-10mu+} (reverse\ (y{:}ys)) \mathbin{+\mkern-10mu+} a & \text{from eq. (11.20)} \\
={}& x : (xs \mathbin{+\mkern-10mu+} reverse\ (y{:}ys)) \mathbin{+\mkern-10mu+} a) & \text{concatenation is associative} \\
={}& x : (xs \mathbin{+\mkern-10mu+} reverse\ ys \mathbin{+\mkern-10mu+} (y{:}a)) & \text{reverse property, and associative} \\
={}& x : rotate\ xs\ ys\ (y{:}a) & \text{reverse of eq. (11.20)}
\end{aligned}
\tag{11.22}
$$

Summarize together:

$$
\begin{aligned}
rotate\ [\ ]\ [y]\ a &= y{:}a \\
rotate\ (x{:}xs)\ (y{:}ys)\ a &= x : rotate\ xs\ ys\ (y{:}a)
\end{aligned}
\tag{11.23}
$$

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 $f \mathbin{+\mkern-10mu+} reverse\ r$ 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.

$$
\begin{aligned}
balance\ f\ r\ [\ ] &= (f', [\ ], f') & \text{where} : f' = rotate\ f\ r\ [\ ] \\
balance\ f\ r\ (x{:}rot) &= (f, r, rot) & \text{advance the rotation}
\end{aligned}
\tag{11.25}
$$

Exercise 11.4

Implement bidirectional queue, support add/remove elements on both head and tail in constant time.

11.7   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

Sequence is the 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, and use n_1, n_2 for lengths if there are two.

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.

12.1 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 fig. 12.1, tree t_1, t_2 represent sequence $[x_1, x_2, x_3, x_4, x_5, x_6]$.

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.

$$n = 2^0 e_0 + 2^1 e_1 + ... + 2^m e_m \tag{12.1}$$
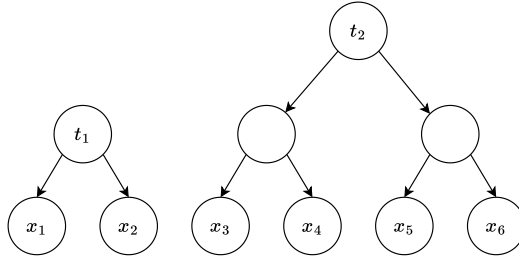
Figure 12.1: A sequence of 6 elements.

If $e_i \neq 0$, there is a full binary tree t_i of size 2^i. For example in fig. 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]. 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 sub-trees respectively. We get the size 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 in Curried form:

$$insert\ y = insert_T\ (y) \tag{12.3}$$

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 fig. 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: $link\ t_1\ t_2 = (size\ t_1 + size\ t_2, t_1, t_2)$.

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.

We define remove reversely. If the first tree is t_0 (singleton leaf), then remove it; otherwise, recursively split the first tree to obtain a t_0 and remove, as shown in fig. 12.3.

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

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 divide elements into chunks. For a given index $1 \leq i \leq n$, first locate the corresponding tree, then lookup the tree to access the element.
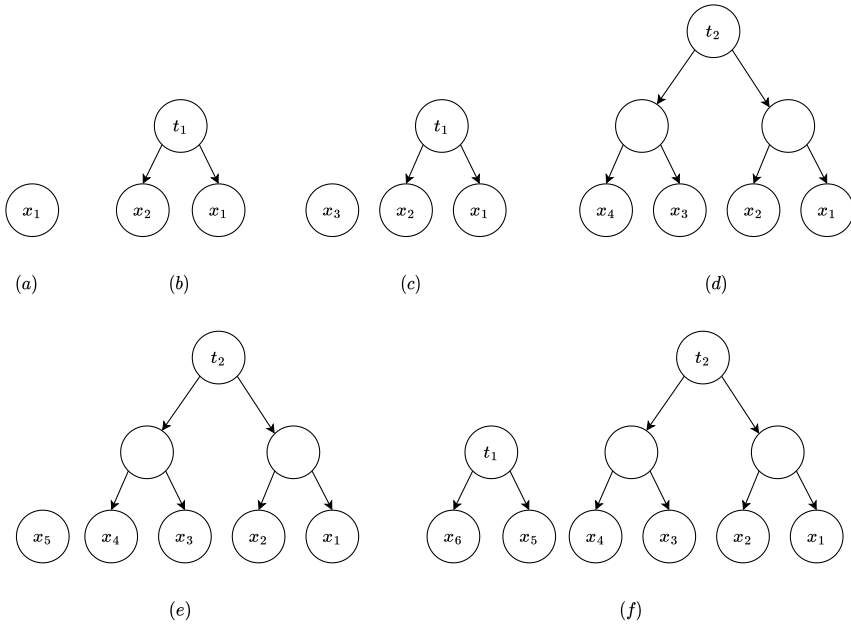
Figure 12.2: Insert $x_1, x_2, ..., x_6$. (a) Insert x_1, (b) Insert x_2, link to $[t_1]$. (c) Insert x_3, result $[t_0, t_1]$. (d) Insert x_4, link twice, generate $[t_2]$. (e) Insert x_5, result $[t_0, t_2]$. (f) Insert x_6, result $[t_1, t_2]$.
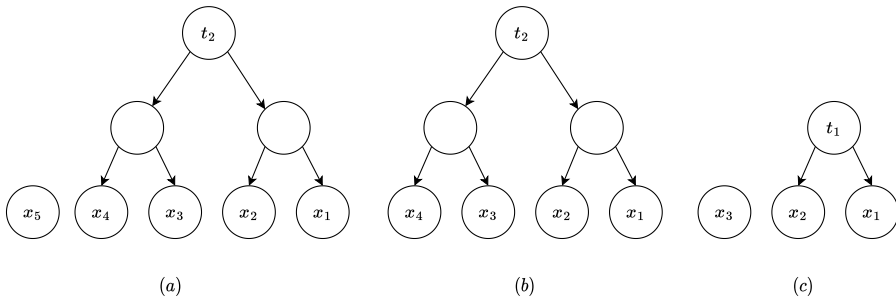
Figure 12.3: Remove: (a) $x_1, x_2, ..., x_5$ as $[t_0, t_2]$. (b) Remove x_5 (t_0) directly. (c) Remove x_4. Split twice to get $[t_0, t_0, t_1]$, then remove the head to get $[t_0, t_1]$.

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 looks up:

$$\begin{aligned} lookup_T\ 1\ (x) &= x \\ lookup_T\ i\ (s, t_1, t_2) &= \begin{cases} i \leq \lfloor \frac{s}{2} \rfloor : & lookup_T\ i\ t_1 \\ \text{otherwise}: & lookup_T\ (i - \lfloor \frac{s}{2} \rfloor)\ t_2 \end{cases} \end{aligned} \tag{12.8}$$

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.

(a)

(b)

(c)

(d)
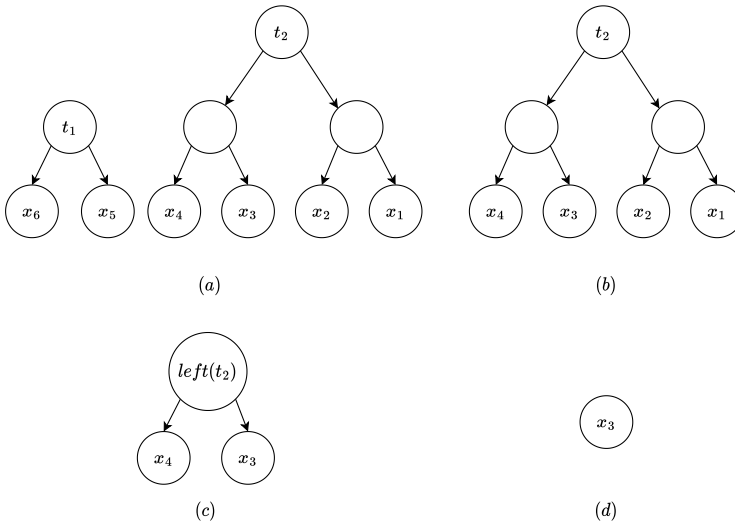
Figure 12.4: Steps to access $S[4]$: (a) $S[4], 4 > size(t_1) = 2$, (b) $S'[4-2] \Rightarrow lookup_T\ 2\ t_2$, (c) $2 \leq \lfloor \frac{size(t_2)}{2} \rfloor \Rightarrow lookup_T\ 2\ left(t_2)$, (d) $lookup_T\ 1\ right(left(t_2))$, return x_3.

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

12.1.1. How to handle the out of bound exception?

12.2 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.

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

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 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 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., $4 - 1 = (100)_2 - 1 = (11)_2 = 3$.

$$
\begin{aligned}
minus\ [One\ t] &= (t, [\ ]) \\
minus\ ((One\ t){:}ts) &= (t, Zero{:}ts) \\
minus\ (Zero{:}ts) &= (t_1, (One\ t_2){:}ts'), \text{where} : (s, t_1, t_2) = minus\ ts
\end{aligned}
\tag{12.10}
$$

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 the final one. The highest bit changes to 1 after insert the last element. Sum as: $T = 1 + 1 + 2 + 4 + ... + 2^{m-1} + 2^m = 2^{m+1}$. Hence the amortized performance is:

$$O(T/n) = O(\frac{2^{m+1}}{2^m}) = O(1) \qquad\qquad (12.11)$$

Proved the amortized constant time performance.

Exercise 12.2

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.

12.2.2. Analyze the amortized performance of delete.

12.2.3. 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.

12.3   paired-array sequence

Expand the paired-array queue (section 11.3 in chapter 11) to paired-array sequence. As shown in fig. 12.5, 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. Denote the sequence as a pair $S = (f, r)$, FRONT$(S) = f$, REAR$(S) = r$ access them respectively. Implement insert/append as below:
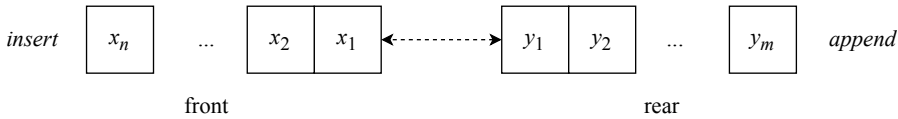
Figure 12.5: Paired-array sequence.

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 which array i indexes to, f or r? 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:     $f, r \leftarrow$ FRONT(S), REAR(S)
3:     $n \leftarrow$ SIZE(f)
4:     **if** $i \leq n$ **then**
5:         **return** $f[n - i + 1]$ ▷ reversed
6:     **else**
7:         **return** $r[i - n]$

Removing can make 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:     $f \leftarrow$ FRONT(S), $r \leftarrow$ REAR(S)
3:     $n \leftarrow$ SIZE(f), $m \leftarrow$ SIZE(r)
4:     **if** $F = [\ ]$ **then**

5:         $k \leftarrow \lfloor \dfrac{m}{2} \rfloor$

6:         **return** $(\text{REVERSE}(r[1...k]), r[(k+1)...m])$

7:     **if** $R = [\,]$ **then**

8:         $k \leftarrow \lfloor \dfrac{n}{2} \rfloor$

9:         **return** $(f[(k+1)...n], \text{REVERSE}(f[1...k]))$

10:     **return** (f, r)

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

1: **function** REMOVE-HEAD(S)

2:     BALANCE(S)

3:     $f, r \leftarrow \text{FRONT}(S), \text{REAR}(S)$

4:     **if** $f = [\,]$ **then** $\triangleright S = ([\,], [x])$

5:         $r \leftarrow [\,]$

6:     **else**

7:         REMOVE-LAST(f)

8: **function** REMOVE-TAIL(S)

9:     BALANCE(S)

10:     $f, r \leftarrow \text{FRONT}(S), \text{REAR}(S)$

11:     **if** $r = [\,]$ **then** $\triangleright S = ([x], [\,])$

12:         $f \leftarrow [\,]$

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

12.3.1. Analyze the amortized performance for paired-array delete.

12.4 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 (section 11.5 in 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.

$$\begin{aligned} s \uplus \varnothing &= s \\ \varnothing \uplus s &= s \\ (x, Q) \uplus s &= (x, \ push \ s \ Q) \end{aligned} \tag{12.12}$$

When insert new element z, we create a singleton of $(z, \varnothing)$, then concatenate it to the sequence:

$$\begin{cases} insert \ z \ s &= (z, \varnothing) \uplus s \\ append \ z \ s &= s \uplus (z, \varnothing) \end{cases} \tag{12.13}$$
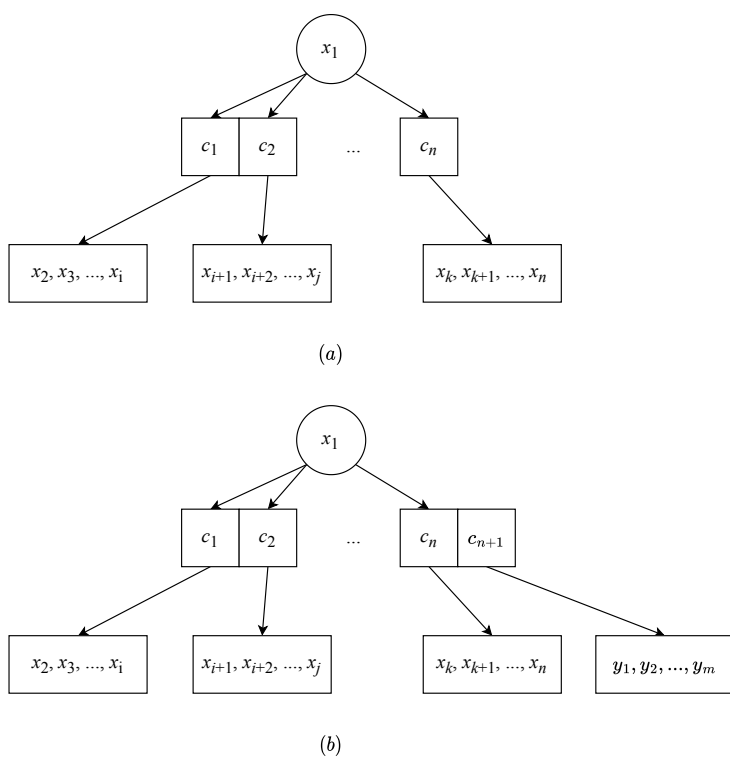
Figure 12.6: Concatenate-able list: (a) $(x_1, Q_x) = [x_1, x_2, ..., x_n]$, (b) Concatenate with $(y_1, Q_y) = [y_1, y_2, ..., y_m]$, add$c_{n+1}$ to Q_x.

When delete x_1 from head, we lose the root. The rest sub-trees are all concatenate-able lists. We concatenate them together to a new sequence.

$$\begin{aligned} concat\ \varnothing &= \varnothing \\ concat\ Q &= (top\ Q) + concat\ (pop\ Q) \end{aligned} \tag{12.14}$$

The real-time queue holds 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].

$$\begin{aligned} fold\ f\ z\ \varnothing &= z \\ fold\ f\ z\ Q &= f\ (top\ Q)\ (fold\ f\ z\ (pop\ Q)) \end{aligned} \tag{12.16}$$

Where f is a binary function, z is zero unit. Here are examples of folding on queue $Q = \{1, 2, ..., 5\}$:

$$\begin{aligned} fold\ (+)\ 0\ Q &= 1 + (2 + (3 + (4 + (5 + 0)))) = 15 \\ fold\ (\times)\ 1\ Q &= 1 \times (2 \times (3 \times (4 \times (5 \times 1)))) = 120 \\ fold\ (\times)\ 0\ Q &= 1 \times (2 \times (3 \times (4 \times (5 \times 0)))) = 0 \end{aligned}$$

We can define *concat* with fold (Curried form):

$$concat = fold\ (+)\ \varnothing \tag{12.17}$$

Normally, the add, append, delete randomly happen. 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. Nevertheless, the amortized performance is constant time.

12.5 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, to realize 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].

---

[1]f: front, r: rear

```
data Tree a = Empty
           | Lf a
           | Tr [a] (Tree (Node a)) [a]
```

12.5.1 Insert
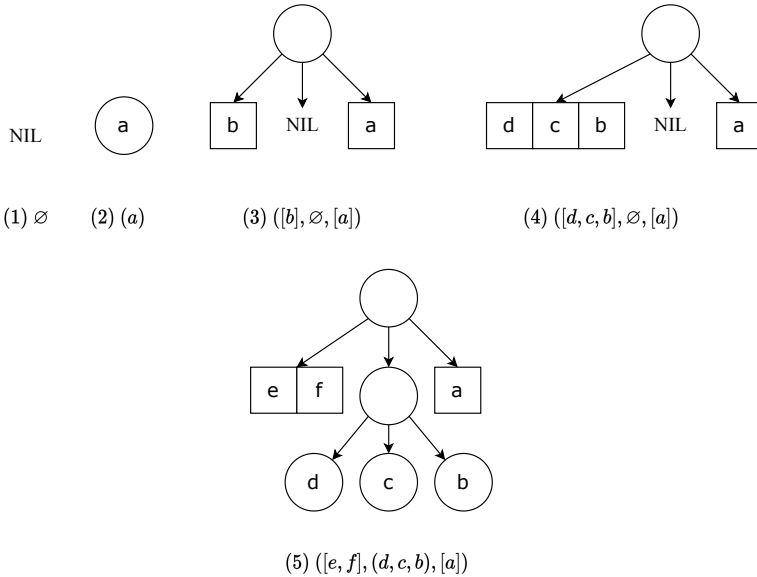
Figure 12.7: Finger tree example

As shown in fig. 12.7. (1) is $\varnothing$; (2) is a singleton; (3) has two element in f and r for each; (4) f will exceed 2-3 tree when add more. We need re-balance as in (5). There are two elements in f, the middle is singleton of a 2-3 tree. These examples are list as below:

$$
\begin{array}{ll}
\varnothing & \texttt{Empty} \\
(a) & \texttt{Lf a} \\
([b], \varnothing, [a]) & \texttt{Tr [b] Empty [a]} \\
([d, c, b], \varnothing, [a]) & \texttt{Tr [d, c, b] Empty [a]} \\
([f, e], (d, c, b), [a]) & \texttt{Tr [f, e] Lf (Br3 d c b) [a]}
\end{array}
$$

In (5), the middle component is a singleton leaf. Finger tree is recursive, apart from f and r, the middle is a deeper finger tree of type $Tree \ (Node \ a)$. One more wrap, one level deeper. Summarize above examples, we define insert a to tree T as below:

1. If $T = \varnothing$, the result is a singleton (a);

2. If $T = (b)$ is a leaf, the result is $([a], \varnothing, [b])$;

3. For $T = (f, t, r)$, if there are < 3 elements in f, we insert a to f, otherwise (≥ 3), extract the last 3 elements from f to a 2-3 tree t', recursively insert t' to t, then insert a to f.

$$
\begin{array}{rcl}
insert\ a\ \varnothing & = & (x) \\
insert\ a\ (b) & = & ([a], \varnothing, [b]) \\
insert\ a\ ([b,c,d], t, r) & = & ([a], insert\ (b,c,d)\ t, r) \\
insert\ a\ (f,t,r) & = & (a{:}f,t,r)
\end{array}
\tag{12.18}
$$

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 12.4

12.4.1. Eliminate recursion, implement insert with loop.

12.5.2 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* flattens 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 12.8 gives examples that extract 2 elements. We define *head, tail* with *extract*.

$$
\begin{cases}
head & = fst \circ extract \\
tail & = snd \circ extract
\end{cases}
\tag{12.22}
$$

Exercise 12.5

12.5.1. Eliminate recursion, implement extract in loops.

12.5.3 Append and remove

We implement append, remove on right symmetrically.

$$
\begin{array}{rcl}
append\ \varnothing\ a & = & (a) \\
append\ (a)\ b & = & ([a], \varnothing, [b]) \\
append\ (f, t, [a, b, c])\ d & = & (f, append\ t\ (a, b, c), [d]) \\
append\ (f, t, r)\ a & = & (f, t, r \mathbin{+\!\!+} [a])
\end{array}
\tag{12.23}
$$
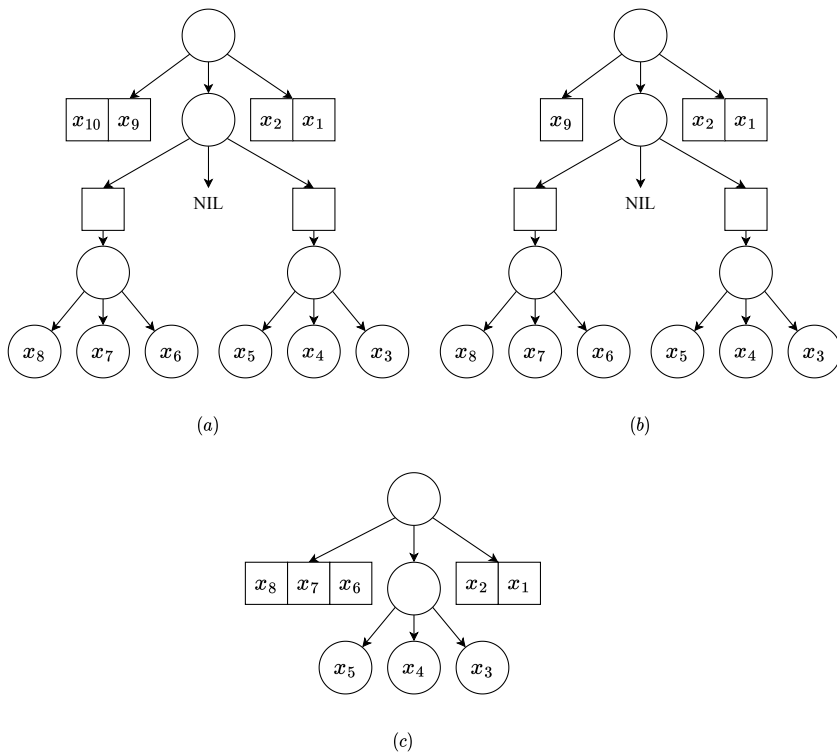
Figure 12.8: Extract: (a) A sequence of 10 elements. (b) Extract one, f becomes a singleton list. (c) Extract another, borrow an element from the middle, flatten the 2-3 tree to a list as the new f.

If there are less than 3 elements in r, we append the new element to the tail of r. Otherwise, we extract 3 to form a 2-3 tree, and recursively append the tree to the middle. We can repeatedly append a list of elements by folding from left:

$$t \lll xs = foldl \; append \; t \; xs \tag{12.24}$$

remove is reverse of *append*:

$$
\begin{aligned}
remove \; (a) &= (\varnothing, a) \\
remove \; ([a], \varnothing, [b]) &= ((a), b) \\
remove \; (f, \varnothing, [a]) &= ((init f, \varnothing, [last f]), a) \\
remove \; (f, t, [a]) &= ((f, t', toList \; r), a), \text{where} : (t', r) = remove \; t \\
remove \; (f, t, r) &= ((f, t, init \; r), last \; r)
\end{aligned}
\tag{12.25}
$$

Where *last* accesses the tail element of a list, *init* returns the rest (eq. (1.4) in chapter 1).

12.5.4 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 is equivalent to the below problem:

$$merge \; t_1 \; (r_1 +\!\!\!+ f_2) \; t_2 =?$$

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 is *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.

$$
\begin{aligned}
merge \; \varnothing \; ts \; t_2 &= ts \ggg t_2 \\
merge \; t_1 \; ts \; \varnothing &= t_1 \lll ts \\
merge \; (a) \; ts \; t_2 &= merge \; \varnothing \; (a{:}ts) \; t_2 \\
merge \; t_1 \; ts \; (a) &= merge \; t_1 \; (ts +\!\!\!+ [a]) \; \varnothing \\
merge \; (f_1, t_1, r_1) \; ts \; (f_2, t_2, r_2) &= (f_1, merge \; t_1 \; (nodes \; (r_1 +\!\!\!+ ts +\!\!\!+ f_2)) \; t_2, r_2)
\end{aligned}
\tag{12.26}
$$

Where function *nodes* collects elements to a list of 2-3 trees. This is because the type of the element in the middle is deeper than the finger.

$$
\begin{aligned}
nodes \; [a, b] &= [(a, b)] \\
nodes \; [a, b, c] &= [(a, b, c)] \\
nodes \; [a, b, c, d] &= [(a, b), (c, d)] \\
nodes \; (a{:}b{:}c{:}ts) &= (a, b, c){:}nodes \; ts
\end{aligned}
\tag{12.27}
$$

We define finger tree concatenation with *merge*:

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

Compare with eq. (12.26), concatenation is essentially merge, let us define them in a unified way:

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

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.

12.5.5   Random access

The idea is to turn random access into tree search. To avoid repeatedly computing 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]
```

$$
\begin{aligned}
size \; \varnothing &= 0 \\
size \; (x) &= size \; x \\
size \; (s, f, t, r) &= s
\end{aligned}
\tag{12.30}
$$

Here $size \; (x)$ is not necessarily 1. x can be a deeper node, like $Node \; a$. It is 1 only at level one. For termination, we wrap the level one 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:

$$
\begin{aligned}
size \; (t_1, t_2) &= size \; t_1 + size \; t_2 \\
size \; (t_1, t_2, t_3) &= size \; t_1 + size \; t_2 + size \; t_3
\end{aligned}
\tag{12.33}
$$

Given a list of nodes (e.g., a finger at deeper level), we calculate the size by: $sum \circ (map \; size)$. We need update the size when insert or delete element. With size augmented, we can lookup the tree at 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, where $s = s_f + s_t + s_r$. 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, lookup r. We also need to handle the leaf case of (x). Use a pair (i, t) to define the position i at data structure t, and define $lookup_T$ as below:

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

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:

$$lookup_N \ i \ (t_1, t_2) \quad = \quad \begin{cases} i < size \ t_1 : & (i, t_1) \\ \text{otherwise} : & (i - size \ t_1, t_2) \end{cases}$$

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

$$(12.36)$$

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 achieve balanced performance with finger tree implementation. The operations at head and tail are bound to amortized constant time, the 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 moves any element at position i to the front of the sequence (see Exercise 12.6.2):

$$mtf \ i \ S = x \lhd S', \text{where}(x, S') = extractAt \ i \ S$$

In the next two 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 12.6

12.6.1. For random access, how to handle the empty tree $\varnothing$ and out of bound cases?

12.6.2. Implement *cut i S*, split sequence S at position i.

12.6   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
```

---

[2]Some programming environments provide equivalent tool, like the `Optional<T>` in Java/C++.

[3]Used in Burrows-Wheeler transform (BWT) data compression algorithm.

```
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] m r) = Br (s + size a) [a] ((tr3 b c d) `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 = Br (s + size d) f (m `snoc` (tr3 a b c)) [d]
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

People prove the upper limit of performance is $O(n \lg n)$ 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.

13.1 Quick sort
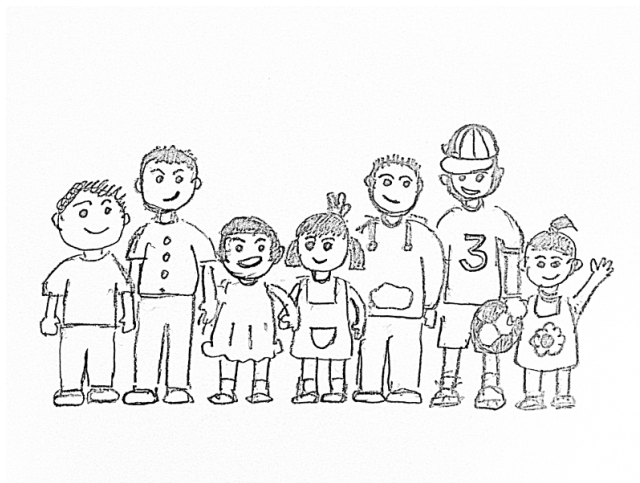
Consider arrange kids in a line ordered by height.

1. The first kid raises hand, all shorter ones 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). He happens to be 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. Those of height 98, 95, 96, and 99 cm move to the left, and the one of 101 cm moves 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.

Summarize the quick sort. Let the list be L:

102	100	98	95	96	99	101	97
100	98	95	96	99	101	97	'102'
98	95	96	99	97	'100'	101	'102'
95	96	97	'98'	99	'100'	'101'	'102'
'95'	96	97	'98'	'99'	'100'	'101'	'102'
'95'	'96'	97	'98'	'99'	'100'	'101'	'102'
'95'	'96'	'97'	'98'	'99'	'100'	'101'	'102'

Table 13.1: Sort steps

- 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 sort left and right in parallel. 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.

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

We use ZF expression (see sections 1.4.1 and 1.7) to filter the list. Below is the example program:

```
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 $(\leq)$ as generic comparison to sort different things like numbers, strings, and etc. We needn't total ordering, but at least need *strict weak ordering*[79] [52] (see section 9.2).

13.1.1   Partition

We traverse the elements in two passes: first filter all elements $\leq x$; next filter all $> x$. Let us combine them into one pass:

$$
\begin{aligned}
part\ p\ [\,] &= ([\,],[\,]) \\
part\ p\ (x{:}xs) &= \begin{cases} p(x): & (x{:}as, bs),\text{where } (as, bs) = part\ p\ xs \\ \text{otherwise}: & (as, x{:}bs) \end{cases}
\end{aligned}
\tag{13.2}
$$

And change the quick sort to:

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

Alternatively, we can define partition with fold (in Curried form):

$$
part\ p\ = foldr\ f\ ([\,],[\,])
\tag{13.4}
$$

Where:

$$
f\ (as, bs)\ x = \begin{cases} p(x): & (x{:}as, bs) \\ \text{otherwise}: & (as, x{:}bs) \end{cases}
\tag{13.5}
$$

It's essentially to accumulate to (as, bs). If $p(x)$ holds, then add x to as, otherwise to bs. Change the partition to tail recursive:

$$
\begin{aligned}
part\ p\ [\ ]\ as\ bs &= (as, bs) \\
part\ p\ (x{:}xs)\ as\ bs &= \begin{cases} p(x): & part\ p\ xs\ (x{:}as)\ bs \\ \text{otherwise}: & part\ p\ xs\ as\ (x{:}bs) \end{cases}
\end{aligned}
\tag{13.6}
$$

To partition $(x{:}xs)$, call $(as, bs) = part\ (\leq x)\ xs\ [\ ]\ [\ ]$. We eliminate the concatenation ($+\!\!+$) in '$sort\ as + [x] + sort\ bs$' with an accumulators s as:

$$
\begin{aligned}
sort'\ s\ [\ ] &= s \\
sort'\ s\ (x{:}xs) &= sort'\ (x : sort'\ s\ bs)\ as
\end{aligned}
\tag{13.7}
$$

Start it with an empty list: $sort = sort'\ [\ ]$. 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' s [] = s
sort' s (x:xs) = sort' (x : sort' s 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)
```

13.1.2 In-place sort

Figure 13.1 gives a way to partition in-place[2] [4]. Scan from left to right. At any time, the array is consist of four parts (fig. 13.1 (a)):

(a) Partition invariant

(b) Initialize

(c) Terminate
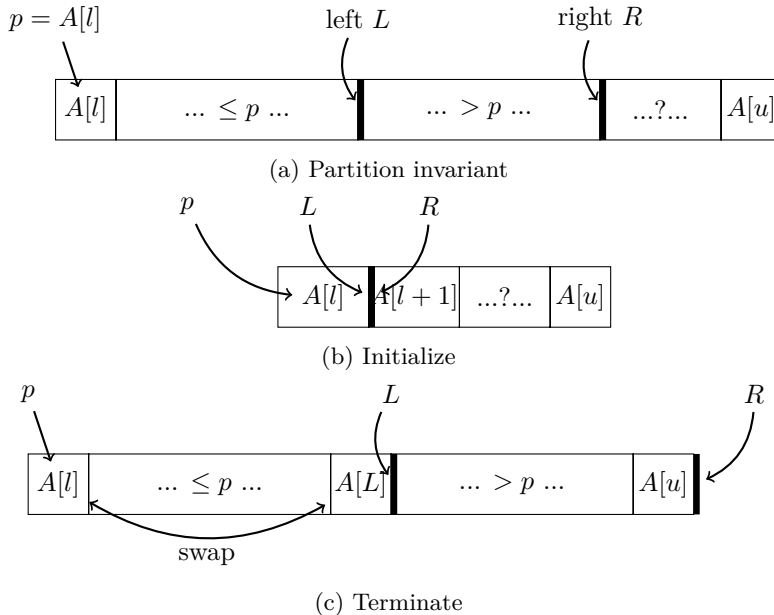
Figure 13.1: In-place partition, pivot $p = A[l]$

1. The pivot is the left element: $p = A[l]$. It moves to the final position after partition;

2. A section of elements $\leq p$, extends right to L;

3. A section of elements $> p$, extends right to R. The elements between L and R are $> p$;

4. Elements after R haven't been partitioned (maybe $>$ or $=$ or $< p$).

When partition starts, L points to p, R points to the next (fig. 13.1 (b)). We advance R to the right boundary. Every time, compare $A[R]$ and p. If $A[R] > p$, it should be between L and R, we move R forward; otherwise if $A[R] \leq p$, it should be on the left of L. We advance L a step, then swap $A[L] \leftrightarrow A[R]$. The partition ends when R passes the last element. All elements $> p$ are moved to the right of L, while others are on the left. We need move p to the position between the two parts. To do that, swap $p \leftrightarrow A[L]$, as shown in fig. 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 boundaries be l, u. The in-place partition is defined below:

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 on 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 $[3, 2, 5, 4, 0, 1, 6, 7]$.

<u>3</u>(l)	2(r)	5	4	0	1	6	7	start, $p = 3$, $l = 1$, $r = 2$
<u>3</u>	2(l)(r)	5	4	0	1	6	7	$2 < 3$, advance l ($r = l$)
<u>3</u>	2(l)	5(r)	4	0	1	6	7	$5 > 3$, move on
<u>3</u>	2(l)	5	4(r)	0	1	6	7	$4 > 3$, move on
<u>3</u>	2(l)	5	4	0(r)	1	6	7	$0 < 3$
<u>3</u>	2	0(l)	4	5(r)	1	6	7	advance l, swap with r
<u>3</u>	2	0(l)	4	5	1(r)	6	7	$1 < 3$
<u>3</u>	2	0	1(l)	5	4(r)	6	7	advance l, swap with r
<u>3</u>	2	0	1(l)	5	4	6(r)	7	$6 > 3$, move on
<u>3</u>	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

We implement quick sort with PARTITION as bellow:

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

We pass the array and its boundaries as QUICK-SORT($A, 1, |A|$) to sort. It returns immediately if the array is empty or a singleton.

Exercise 13.1

13.1.1. Optimize the basic quick sort definition for the singleton list case.

13.1.3 Performance

Quick sort performs well in most cases. Consider the best/worst cases. For the best case, it always halves the elements into two equal sized parts. As shown in figure fig. 13.2, there are total $O(\lg n)$ levels of recursions. At level one, it processes n elements with one partition; at level two, partitions twice, each processes $n/2$ elements, taking total $2O(n/2) = O(n)$ time; at level three, partitions four times, each processes $n/4$ 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 $O(n \lg n)$.
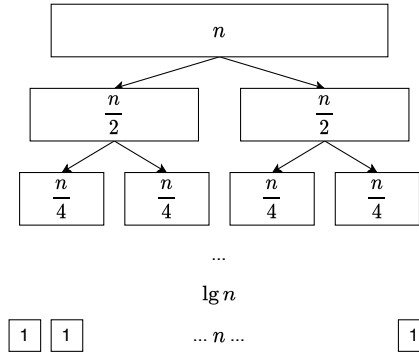
Figure 13.2: The best case, halve every time.

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. There are several challenging cases, for example, the sequence has many duplicated elements, or is largely ordered, and etc. 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 methods 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 x_i in A will be compared with any y_j 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].

$$
H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots = \ln n + \gamma + \epsilon_n
$$

$$
C(n) = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n)
\tag{13.11}
$$

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:

$$
\begin{aligned}
T(n) &= E(T(i)) + E(T(n-i-1)) + cn \\
&= \frac{1}{n} \sum_{i=0}^{n-1} T(i) + \frac{1}{n} \sum_{i=0}^{n-1} T(n-i-1) + cn \\
&= \frac{1}{n} \sum_{i=0}^{n-1} T(i) + \frac{1}{n} \sum_{j=0}^{n-1} T(j) + cn \\
&= \frac{2}{n} \sum_{i=0}^{n-1} T(i) + cn
\end{aligned}
\tag{13.13}
$$

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 (eq. (13.14)) - (eq. (13.15)), cancel all $T(i)$ for $0 \le i < n-1$.

$$
nT(n) = (n+1)T(n-1) + 2cn - c
\tag{13.16}
$$

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.

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}$$

$$...$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

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:

$$O(T(n)) = O(n \lg n) \tag{13.20}$$

13.1.4 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 the duplicated elements:

$$
\begin{array}{rcl}
sort\ [\ ] & = & [\ ] \\
sort\ (x{:}xs) & = & sort\ S \mathbin{+\!\!+} sort\ E \mathbin{+\!\!+} sort\ G
\end{array}
\tag{13.21}
$$

Where:

$$
\begin{cases}
S = [y \leftarrow xs, y < x] \\
E = [y \leftarrow xs, y = x] \\
G = [y \leftarrow xs, y > x]
\end{cases}
$$

we use an accumulator to improve the list concatenation: $qsort = sort\ [\ ]$, where:

$$
\begin{array}{rcl}
sort\ A\ [\ ] & = & A \\
sort\ A\ (x{:}xs) & = & sort\ (E \mathbin{+\!\!+} sort\ A\ G)\ S
\end{array}
\tag{13.22}
$$

The sub-list E contains elements of the same value, hence sorted. We first sort G with the accumulator A, then prepend E as the new accumulator, and use it to sort S. We also improve the partition with accumulator:

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

Richard Bird gives another improvement[1]: collect the recursive sort results with a list and concatenate it 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 develops the two-way partition method[69] [2]. Use two pointers i, j to the left and right boundaries. Pick the first element as the pivot p. Advance i to right till meet an element $\geq p$; while (in parallel) move j to left till meet an element $\leq p$. At this time, all elements on the left to i are less than the pivot $(< p)$, while those on the right to j are greater than the pivot $(> p)$. i points to an element that is $\geq p$, and j points to an element that is $\leq p$, as shown in fig. 13.3 (a). To move all elements $\leq p$ to the left, and the remaining to the right, we exchange $x[i] \leftrightarrow x[j]$, then continue scan. Repeat this till i meets j. At any time, we keep the invariant: all elements one the left to i (include i) are $\leq p$; while all on the right to j (include j) are $\geq p$. The elements between i and j are yet to scan, as shown in fig. 13.3 (b).

(a) When i and j stop

(b) Partition invariant
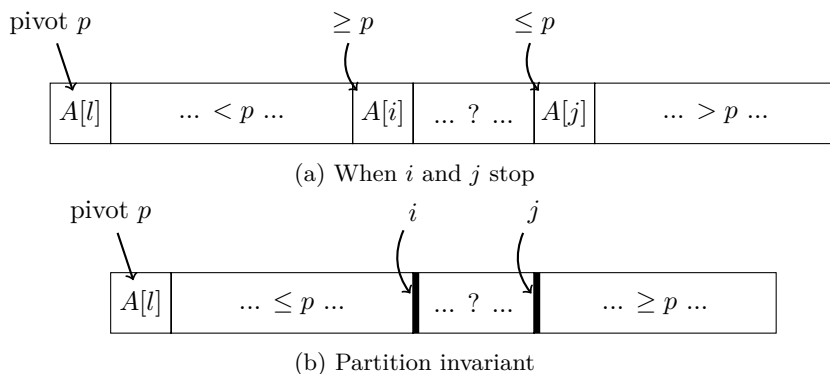
Figure 13.3: 2-way scan

When i meets j, we need an additional exchange: swap the pivot p to position j. Then recursive sort the sub-arrays $A[l...j)$ 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**

```
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)
```

For the extreme 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 the 2-way scan and the ternary partition. Only recursively sort the elements different with the pivot. Jon Bentley and Douglas McIlroy develop a method as shown in fig. 13.4 (a), that store the elements equal to the pivot on both sides [70] [71].
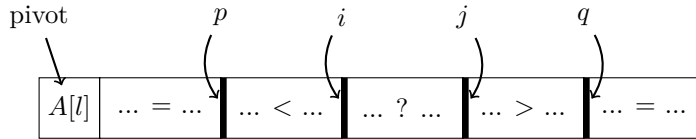
(a) Ternary partition invariant.
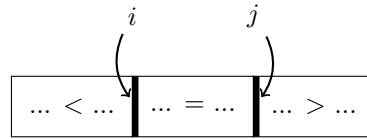
(b) Swap the elements $= p$ to the middle.

Figure 13.4: Ternary partition

We scan from two sides, pause when i reaches an element $\geq$ the pivot, and j reaches 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 does nothing if all elements are distinct. The partition result is shown as fig. 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 the 2-way scan and the ternary partition. Alternatively, we change the one pass scan to the ternary partition directly. Pick the first element as the pivot, as shown in fig. 13.5. 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.
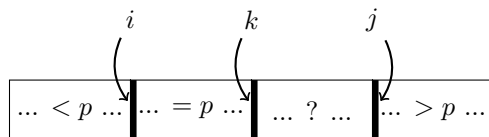
Figure 13.5: 1 way scan ternary partition

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 need 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)
```

This implementation is less complex but need more swaps than the version of ternary partition through 2-way scan.

Challenging cases

Although ternary partition handles duplicated elements well, there are other challenging cases. For example, when most elements are ordered (ascending or descending), the partition is unbalanced. Figure 13.6 gives two 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 fig. 13.7.
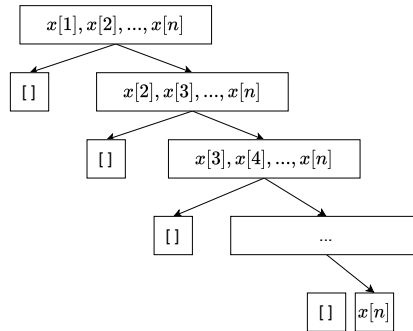
(a) Partition tree of $[x_1 < x_2 < ... < x_n]$, the sub-trees of $\leq p$ are empty.
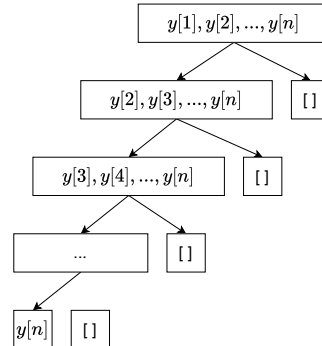
(b) Partition tree of $[y_1 > y_2 > ... > y_n]$, the sub-trees of $\geq p$ are empty.

Figure 13.6: The challenging cases - 1.

In these challenging cases, the partition is unbalanced when choose the first element as the pivot. Robert Sedgwick improves 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 to end, and move the median to the middle.

```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
```
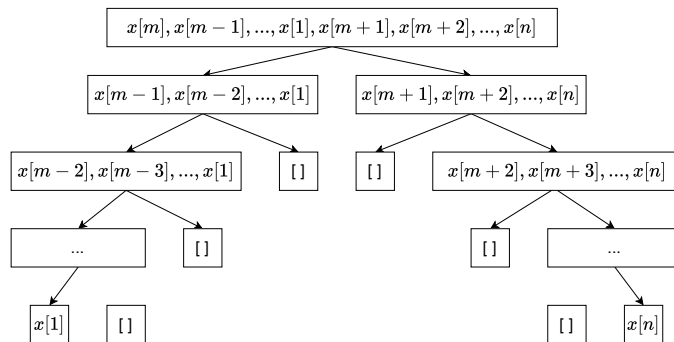
(a) Unbalanced partitions except for the first time.
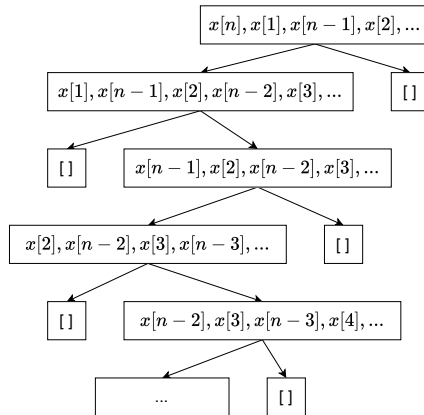
(b) A zig-zag partition tree.

Figure 13.7: The challenging cases - 2.

3:	$m \leftarrow \lfloor \dfrac{l+u}{2} \rfloor$	▷ or $l + \dfrac{u-l}{2}$ to void overflow
4:	**if** $A[m] < A[l]$ **then**	▷ Ensure $A[l] \leq A[m]$
5:	EXCHANGE $A[l] \leftrightarrow A[m]$	
6:	**if** $A[u-1] < A[l]$ **then**	▷ Ensure $A[l] \leq A[u-1]$
7:	EXCHANGE $A[l] \leftrightarrow A[u-1]$	
8:	**if** $A[u-1] < A[m]$ **then**	▷ Ensure $A[m] \leq A[u-1]$
9:	EXCHANGE $A[m] \leftrightarrow A[u-1]$	
10:	EXCHANGE $A[l] \leftrightarrow A[m]$	
11:	$(i, j) \leftarrow$ PARTITION(A, l, u)	
12:	SORT(A, l, i)	
13:	SORT(A, j, u)	

This implementation handles the above four challenging cases well, known as the 'median of three'. Alternatively, we can randomly pick the pivot:

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 find quick sort has overhead when the list is short, while insert sort performs better[2] [70]. Sedgewick, Bentley and McIlroy evaluate 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)

13.1.5   quick sort and tree sort

The 'true quick sort' is the combination of multiple engineering improvements, e.g., 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 derives quick sort from binary tree sort by deforestation[72]. Define *unfold* that converts a list to binary search tree:

$$
\begin{array}{lcl}
unfold\,[\,] & = & \varnothing \\
unfold\,(x{:}xs) & = & (unfold\,[a \leftarrow xs, a \leq x], x, unfold\,[a \leftarrow xs, a > x])
\end{array}
\tag{13.24}
$$

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, and right sub-trees. Where the left sub-tree has the elements $\leq x$; and the right has the elements $> x$. Define in-order traverse to convert a binary search tree to ordered list:

$$toList \ \varnothing \quad = \quad [\,] $$
$$toList \ (l, k, r) \quad = \quad toList \ l +\!\!+ [k] +\!\!+ toList \ r \tag{13.25}$$

Then define quick sort as:

$$sort = toList \circ unfold \tag{13.26}$$

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[73]), we obtain the quick sort.

13.2   Merge sort

Quick sort performs well in most cases. However, it can't avoid the worst cases completely. Merge sort guarantees $O(n \lg n)$ performance in all cases, supports both array and list. Many programming environments provide merge sort as the standard tool[1]. Merge sort takes divide and conquer approach. It always splits the sequence in half and half, recursively sorts and merges them.

$$sort \ [\,] \quad = \quad [\,] $$
$$sort \ [x] \quad = \quad [x]$$
$$sort \ xs \quad = \quad merge \ (sort \ as) \ (sort \ bs), \text{where} : (as, bs) = halve \ xs \tag{13.27}$$

Where *halve* splits the sequence. For array, we can cut at the middle: $splitAt \ \lfloor \dfrac{|xs|}{2} \rfloor \ xs$. However, it takes linear time to move to the middle point of a list (see eq. (1.45)):

$$splitAt \ n \ xs = shift \ n \ [\,] \ xs \tag{13.28}$$

Where:

$$shift \ 0 \ as \ bs \quad = \quad (as, bs)$$
$$shift \ n \ as \ (b{:}bs) \quad = \quad shift \ (n-1) \ (b{:}as) \ bs \tag{13.29}$$

Because *halve* needn't keep the relative order among elements, we can simplify it with odd-even split. There are same number of elements in odd and even positions, or they only differ by one. Define $halve = split \ [\,] \ [\,]$, where:

$$split \ as \ bs \ [\,] \quad = \quad (as, bs)$$
$$split \ as \ bs \ [x] \quad = \quad (x{:}as, bs)$$
$$split \ as \ bs \ (x{:}y{:}xs) \quad = \quad split \ (x{:}as) \ (y{:}bs) \ xs \tag{13.30}$$

We can further simplify it with folding. As in below example, we add x to as every time, then swap $as \leftrightarrow bs$:

```
halve = foldr f ([], []) where
  f x (as, bs) = (bs, x:as)
```

As in fig. 13.8, consider two groups of kids, each is 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. Repeat this till either group complete pass the gate, then the remaining kids go one by one.

---

[1]For example in the standard library of Haskell, Python, and Java.
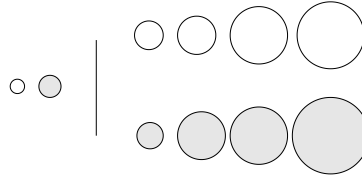
Figure 13.8: Merge

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

For array, we directly cut at the middle, recursively sort two halves, then merge:

1: **procedure** SORT(A)
2:     $n \leftarrow |A|$
3:     **if** $n > 1$ **then**
4:         $m \leftarrow \lfloor \frac{n}{2} \rfloor$
5:         $X \leftarrow$ COPY-ARRAY($A[1...m]$)
6:         $Y \leftarrow$ 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 as 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 finishes, then add all the remaining to A.

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

To simplify merge, we adjoin ∞ to X and Y[2].

1: **procedure** MERGE(A, X, Y)
2:     APPEND(X, ∞)

---

[2] $-\infty$ for descending order

```
 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
```

13.2.1 Performance

Merge sort takes two steps: partition and merge. It always halves the sequence. The binary partition tree is balanced as shown in fig. 13.2. Its height is $O(\lg n)$, so as the recursion depth. The merge happens at all levels, it compares elements one by one from each sorted sub-sequence, hence 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 and second halves, each takes $T(\frac{n}{2})$ time; and merge in cn time, where c is a constant. Solving this equation gives $T(n) = O(n \lg n)$. For 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. It consumes the largest space of $O(n \lg n)$ when reaches to the deepest recursion.

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 recursion, and release it finally.

```
 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:          $k \leftarrow k + 1$
14:          $i \leftarrow i + 1$
15:      **while** $j \leq u$ **do**
16:          $B[k] \leftarrow A[j]$
17:          $k \leftarrow k + 1$
18:          $j \leftarrow j + 1$
19:      **for** $i \leftarrow$ from l to u **do**   ▷ copy back
20:          $A[i] \leftarrow B[i]$

This implementation reduces the space from $O(n \lg n)$ to $O(n)$, improves performance 20% to 25% for 100K numeric elements.

13.2.2 In-place merge sort

To avoid additional space, consider how to reuse the array as the work area. As shown in fig. 13.9, sub-array X and Y 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 ($A[l] \geq A[m]$), move $A[m]$ to the merged part before l. We need shift all elements in range $[l, m)$[3] to right a step.

merged	$A[l]$	... sorted X ...	$A[m]$	... sorted Y ...
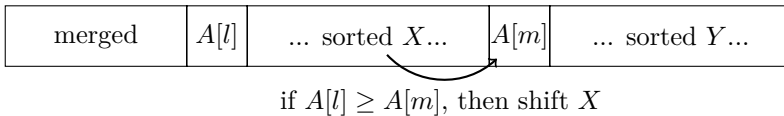
if $A[l] \geq A[m]$, then shift X

Figure 13.9: In-place shift and merge

1: **procedure** MERGE(A, l, m, u)
2:      **while** $l \leq m \wedge m \leq u$ **do**
3:          **if** $A[l] < A[m]$ **then**
4:              $l \leftarrow l + 1$
5:          **else**
6:              $x \leftarrow A[m]$
7:              **for** $i \leftarrow m$ down-to $l + 1$ **do**   ▷ Shift
8:                  $A[i] \leftarrow A[i - 1]$
9:              $A[l] \leftarrow x$

However, it downgrades to $O(n^2)$ time because array shift takes linear time ($O(|X|)$). When sort sub-array, we want to reuse the remaining part as the work area, and must avoid overwriting any elements. We compare elements from sorted sub-array X and Y, pick the less one and store it in the work area. However, we need exchange the element out to free up the cell. After merge, X and Y together store the content of the original work area, as shown in fig. 13.10.

The sorted X, Y, and the work area Z are all sub-arrays. We pass the start, end positions of X and Y as ranges $[i, m)$, $[j, n)$. 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:              EXCHANGE $A[k] \leftrightarrow A[i]$
5:              $i \leftarrow i + 1$
6:          **else**
7:              EXCHANGE $A[k] \leftrightarrow A[j]$

---

[3]range $[a, b)$ includes a, but excludes b.
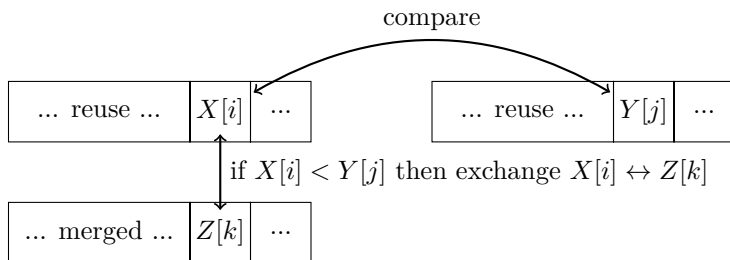
Figure 13.10: Merge and swap

8:	$j \leftarrow j + 1$
9:	$k \leftarrow k + 1$
10:	**while** $i < m$ **do**
11:	EXCHANGE $A[k] \leftrightarrow A[i]$
12:	$i \leftarrow i + 1$
13:	$k \leftarrow k + 1$
14:	**while** $j < m$ **do**
15:	EXCHANGE $A[k] \leftrightarrow A[j]$
16:	$j \leftarrow j + 1$
17:	$k \leftarrow k + 1$

The work area has two properties: 1) It has sufficient size to hold the elements swapped in; 2) It can overlap with either sorted sub-arrays, but must not overwrite any unmerged elements. One idea is to use half array as the work area to sort the other half, as shown in fig. 13.11.
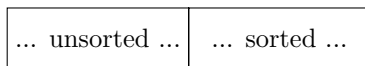
Figure 13.11: Merge and sort half array

We next sort further half of the work area (remaining $\frac{1}{4}$), as shown in fig. 13.12. We must merge X ($\frac{1}{2}$ array) and Y ($\frac{1}{4}$ array) later sometime. However, the work area can only hold $\frac{1}{4}$ array, insufficient for the size of $X + Y$.
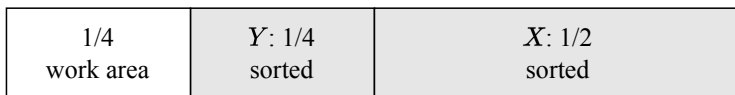
Figure 13.12: Work area can't support merge X and Y.

The second property gives a way out: 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 Y to the first $1/2$, the new work area is between X and Y, as shown in the upper of fig. 13.13. The work area is overlapped with X[74]. Consider two extremes:

1. $y < x$, for all y in Y, x in X. After merge, contents of Y and the work area are swapped (the size of Y equals to the work area);

Y: 1/4 sorted	1/4 work area	X: 1/2 sorted

1/4 work area	merge X, Y: 3/4 sorted
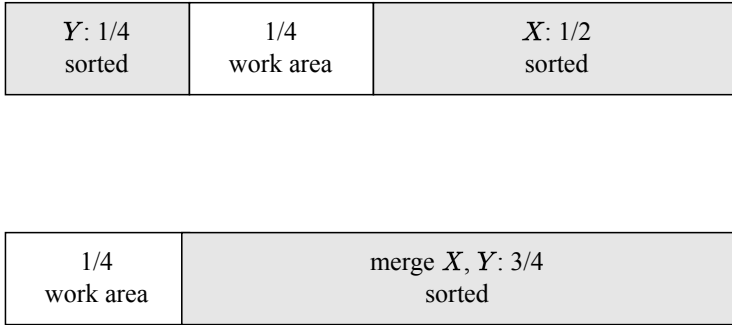
Figure 13.13: Merge X and Y with the work area.

2. $x < y$, for all y in Y, x in X. During merge, we repeatedly swap content of X and the work area. After half of X is swapped, we start overriding X. Fortunately, we only override the merged content. The right boundary of work area keep extending to the 3/4 of the array. After that, we start swapping the content of Y and the work area. Finally, the work area moves to the left side of the array, as shown in the bottom of fig. 13.13.

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 $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$ of the array, terminate when there is only one element left. Alternatively, we can fallback to insert sort for the last few elements.

```
1: procedure SORT(A, l, u)
2:     if u - l > 0 then
3:         m ← ⌊(l + u)/2⌋
4:         w ← l + u - m
5:         SORT'(A, l, m, w)                          ▷ sort half
6:         while w - l > 1 do
7:             u' ← w
8:             w ← ⌈(l + u')/2⌉                        ▷ 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 ← w down-to l do                      ▷ Switch to insert sort
12:            j ← i
13:            while j ≤ u and A[j] < A[j - 1] do
14:                EXCHANGE A[j] ↔ A[j - 1]
15:                j ← 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 the merged part.

```
1: procedure SORT'(A, l, u, w)
2:     if u - l > 0 then
3:         m ← ⌊(l + u)/2⌋
4:         SORT(A, l, m)
5:         SORT(A, m + 1, u)
```

6:                MERGE($A, [l, m), [m + 1, u), w$)
7:        **else**   ▷ 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 eq. (13.33) and eq. (13.34):

$$T(n) - T(\frac{n}{2}) = T(\frac{n}{2}) + cn(\frac{1}{2} + \frac{1}{2} + ...)$$

It adds $\frac{1}{2}$ with total $\lg n$ times, hence:

$$T(n) = 2T(\frac{n}{2}) + \frac{c}{2}n \lg n$$

Apply telescope method, (or master theorem) gives the result $O(n \lg^2 n)$.

13.2.3   Nature merge sort
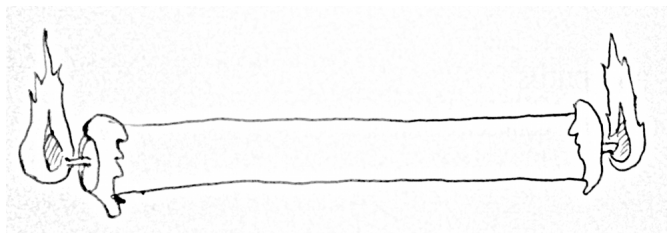
Figure 13.14: Burn from both ends

Knuth gives another implementation, called *nature merge sort*. It likes burning a candle from both ends[51]. For any sequence, one can always find an 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 third row is the other extreme that the segment extends to the right end, the whole sequence is ordered. Symmetrically, we can always find ordered segment from right end, then merge the two sorted segments, one from left, another from right. The advantage is to re-use the nature ordered sub-sequences for partition.
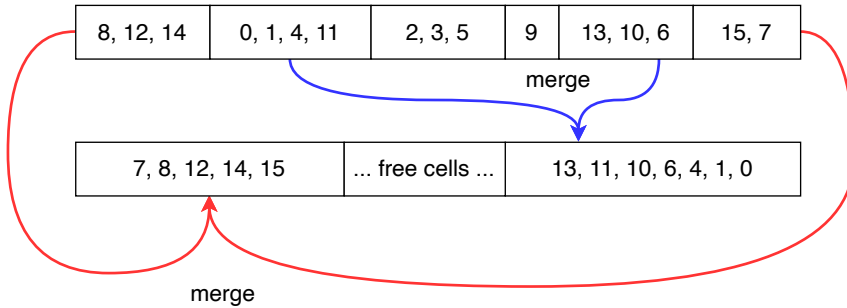
Figure 13.15: Nature merge sort

As shown in fig. 13.15, scan from both ends, find the two longest ordered segments respectively. Then merge them to the left of the work area. Next, restart to scan from left and right to center. This time, merge the two segments from the right to left of the work area. We switch the merge direction right/left in-turns. After scanning all elements and merge them to the work area, swap the original array and the work area, then start a new round of bi-directional scan and merge. Terminate when the ordered segment extends to cover the whole array. This implementation processes the array from both directions based on nature ordering, called *nature two-way merge sort*. As shown in fig. 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, merge from r to left.
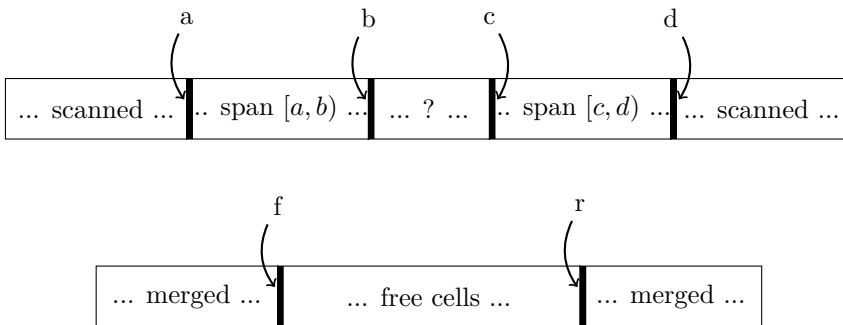
Figure 13.16: A status of nature merge sort

When start, we allocate a work area of the same size as the array. a, b point to the left side, c, d point to the right side. f, r point to the two sides of the work area respectively.

```
1: function SORT(A)
2:     if |A| > 1 then
3:         n ← |A|
4:         B ← CREATE-ARRAY(n)                    ▷ the work area
```

```
 5:          loop
 6:              [a, b) ← [1, 1)
 7:              [c, d) ← [n + 1, n + 1)
 8:              f ← 1, r ← n                              ▷ front, rear of the work area
 9:              t ← 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            ▷ Terminate 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 doubles the sub-sequence length, hence we need at most $O(\lg n)$ rounds. Because we scan 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 merge them in pairs. It halves the sub-lists every round, and finally builds the sorted result. Define this

as (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:

$$
\begin{array}{rcl}
mergePairs \; (g_1{:}g_2{:}gs) & = & merge \; g_1 \; g_2 : mergePairs \; gs \\
mergePairs \; gs & = & gs
\end{array}
\tag{13.38}
$$

Exercise 13.2

13.2.1. One defines $sort' = foldr \; merge \; [\,]$. Is the performance same as the pairwise merge ($mergePairs$)? If yes, prove it; if not, which one is faster?

13.2.4 Bottom-up merge sort

We can develop the bottom-up merge sort from the above performance analysis. First wrap all elements as n singletons. 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 sort all. Knuth calls it 'straight two-way merge sort'[51], as shown in fig. 13.17.
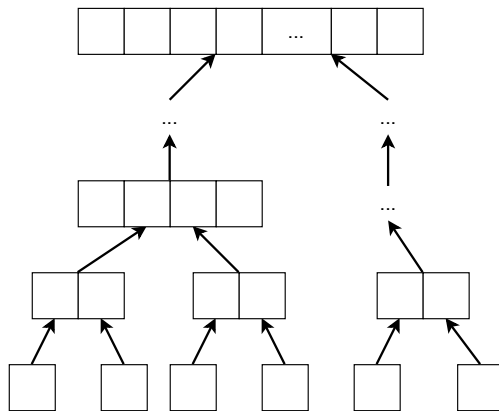
Figure 13.17: Bottom-up merge sort

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:

$$sort = sort' \circ map(x \mapsto [x]) \tag{13.39}$$

We reuse the *mergePairs* defined for nature merge sort, terminates when consolidates 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 spans the ordered sub-sequence as long as possible; while the bottom up merge sort only spans the length to 1. From the tail recursive implementation, we can eliminate the recursion and convert it to iterative loops.

```
1: function SORT(A)
2:     n ← |A|
3:     B ← CREATE-ARRAY(n)
4:     for i from 1 to n do
5:         B[i] = [A[i]]
6:     while n > 1 do
7:         for i ← from 1 to ⌊n/2⌋ do
8:             B[i] ← MERGE(B[2i − 1], B[2i])
9:         if ODD(n) then
10:            B[⌈n/2⌉] ← B[n]
11:        n ← ⌈n/2⌉
12:    if B = [ ] then
13:        return [ ]
14:    return B[1]
```

Exercise 13.3

13.3.1. Define the generic pairwise fold $foldp$, and use it to implement the bottom-up merge sort.

13.3   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].

13.4   Summary

This chapter gives two popular divide and conquer sort algorithms: quick sort and merge sort. Both achieve the best performance of $O(n \lg n)$ for comparison based sort. Sedgewick quotes 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

---

[5]In practice, most are kind of hybrid sort, for example, fallback to insert sort for small sequence.

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 space 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 consider quick sort as the optimized tree sort. Similarly, we can also deduce merge sort from tree sort[75]. People categorize sort algorithms in different ways[51], for example, from the perspective 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. While merge sort is easy to partition no matter cut at the middle, even-odd split, nature split, or bottom up split. It's difficult to achieve perfect partition in quick sort, we can't 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 write this chapter. It becomes routine to sort hundreds of Gigabytes with limited resources and time.

Exercise 13.4

13.4.1. Build a binary search tree from a sequence using the idea of merge sort.

13.5 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 {
            while i < u and xs[i] < xs[pivot] {
                i = i + 1
            }
            while j ≥ l and xs[pivot] < xs[j] {
                j = j - 1
            }
            if j < i then break
```

```
            swap(xs[i], xs[j])
        }
        swap(xs[pivot], xs[j])
        sort(xs, l, j)
        sort(xs, i, u)
    }
}
```

Merge sort:

```
[K] sort([K] xs) {
    Int n = length(xs)
    if n > 1 {
        var ys = sort(xs[0 ... n/2 - 1])
        var zs = sort(xs[n/2 ...])
        xs = merge(xs, ys, zs)
    }
    return xs
}

[K] merge([K] xs, [K] ys, [K] zs) {
    Int i = 0
    while ys ≠ [] and zs ≠ [] {
        xs[i] = if ys[0] < zs[0] then pop(ys) else pop(zs)
        i = i + 1
    }
    xs[i...] = if ys ≠ [] then ys else zs
    return xs
}
```

Merge sort with work area:

```
Void sort([K] xs) = msort(xs, copy(xs), 0, length(xs))

Void msort([K] xs, [K] ys, Int l, Int u) {
    if (u - l > 1) {
        Int m = l + (u - l) / 2
        msort(xs, ys, l, m)
        msort(xs, ys, m, u)
        merge(xs, ys, l, m, u)
    }
}

Void merge([K] xs, [K] ys, Int l, Int m, Int u) {
    Int i = l, Int k = l; Int j = m
    while i < m and j < u {
        ys[k++] = if xs[i] < xs[j] then xs[i++] else xs[j++]
    }
    while i < m {
        ys[k++] = xs[i++]
    }
    while j < u {
        ys[k++] = xs[j++]
    }
    while l < u {
        xs[l] = ys[l]
        l++
    }
}
```

In-place merge sort:

```
Void merge([K] xs, (Int i, Int m), (Int j, Int n), Int w) {
    while i < m and j < n {
        swap(xs, w++, if xs[i] < xs[j] then i++ else j++)
    }
```

```
        while i < m {
            swap(xs, w++, i++)
        }
        while j < n {
            swap(xs, w++, j++)
        }
}

Void wsort([K] xs, (Int l, Int u), Int w) {
    if u - l > 1 {
        Int m = l + (u - l) / 2
        imsort(xs, l, m)
        imsort(xs, m, u)
        merge(xs, (l, m), (m, u), w)
    }
    else {
        while l < u { swap(xs, l++, w++) }
    }
}

Void imsort([K] xs, Int l, Int u) {
    if u - l > 1 {
        Int m = l + (u - l) / 2
        Int w = l + u - m
        wsort(xs, l, m, w)
        while w - l > 2 {
            Int n = w
            w = l + (n - l + 1) / 2;
            wsort(xs, w, n, l);
            merge(xs, (l, l + n - w), (n, u), w);
        }
        for Int n = w; n > l; --n {
            for Int m = n; m < u and xs[m] < xs[m-1]; m++ {
                swap(xs, m, m - 1)
            }
        }
    }
}
```

Iterative bottom up merge sort:

```
[K] sort([K] xs) {
    var ys = [[x] | x in xs]
    while length(ys) > 1 {
        ys += merge(pop(ys), pop(ys))
    }
    return if ys == [] then [] else pop(ys)
}

[K] merge([K] xs, [K] ys) {
    [K] zs = []
    while xs ≠ [] and ys ≠ [] {
        zs += if xs[0] < ys[0] then pop(xs) else pop(ys)
    }
    return zs ++ (if xs ≠ [] then xs else ys)
}
```

Chapter 14

Solution search

Computers enables people to search the solution for many problems: we build robot to search and pick the gadget in assembly lane; we develop car navigator to search the map for the best route; we make smart phone application to search the best shopping plan. This chapter is about the elementary lookup, matching, and solution search algorithms.

14.1 k selection problem

A selection problem is to find the k-th smallest (or largest, $k > 0$) element in a collection xs (list or array). The ordering is abstract, denoted as $\leq$. The intuitive method repeatedly finds the minimum for k times. It takes $O(n)$ time to find the minimum, where $n = |xs|$ is the size. The total performance is bound to $O(kn)$. Alternatively, we can use heap to update, access the top element in $O(\lg n)$ time, hence find the k-th element in $O(k \lg n)$ time.

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

Or in Curried form:

$$top\ k\ = (find\ k) \circ heapify \tag{14.2}$$

Where:

$$
\begin{aligned}
find\ 1 &= top \\
find\ k &= (find\ (k-1)) \circ pop
\end{aligned}
\tag{14.3}
$$

We can do it even better with the divide and conquer method. Pick an arbitrary element p in xs, split xs into as and bs with p ($X = as + [p] + bs$), where $as = [x \leftarrow xs, x \leq p]$ and $B = [x \leftarrow xs, x > p]$. Let $m = |as|$ be the size of as, compare m and k:

1. If $m = k - 1$, then p is the k-th element;

2. If $m < k - 1$, the k-th element is in as, drop bs and recursively search in as;

3. If $m > k - 1$, the k-th element is in bs, drop as and recursively search the $(k-m)$-th element in bs.

Reuse the *part* function in quick sort (see eq. (13.2)):

$$
top\ k\ (x{:}xs) = \begin{cases}
m = k - 1: & x, \text{where } m = |as|, (as, bs) = part\ (\leq x)\ xs \\
m < k - 1: & top\ (k - m)\ bs \\
\text{otherwise}: & top\ k\ as
\end{cases}
\tag{14.4}
$$

In ideal case, the split is balanced (the sizes of *as* and *bs* are almost same), halves the size every time. The performance is $O(n + n/2 + n/4 + ...) = O(n)$. Same as the quick sort algorithm, the worst case happens when the partition is always unbalanced. The performance downgrades to $O(kn)$ or $O((n - k)n)$. In average case, we can find the k-th element in linear time. Most engineering practices in quick sort are applicable too, like the 'media of three'[1], and randomly pivot:

```
1: function TOP(k, xs, l, u)
2:     EXCHANGE xs[l] ↔ xs[ RANDOM(l, u) ]                    ▷ Randomly select in [l, u]
3:     p ← PARTITION(xs, l, u)
4:     if p − l + 1 = k then
5:         return xs[p]
6:     if k < p − l + 1 then
7:         return TOP(k, xs, l, p − 1)
8:     return TOP(k − p + l − 1, xs, p + 1, u)
```

We can change to return all the top k elements (in arbitrary order), as below example program:

```
tops _ [] = []
tops 0 _  = []
tops n (x:xs) | len == k = as
              | len <  n = as ++ [x] ++ tops (n - len - 1) bs
              | otherwise = tops n as
    where
       (as, bs) = partition (≤ x) xs
       len = length as
```

14.2 Binary search

My high school teacher once played a 'math magic'. He asked a student to pick a number from 0 to 1000 in mind. He asked 10 questions, then figured out that number based on the yes/no answers from the student. For example: is it even? is it prime? can it be divided by 3? and etc. If halves the numbers with every question, one can find any number within 1000 because $2^{10} = 1024 > 1000$. The question of whether it is even, perfectly halves numbers[2]. The game becomes not so interesting when the player guess like: 1000, high; 50, low; 750, low; 890, low; 990, correct! This is the *binary search* method. To find x in an ordered sequence A, one firstly tries the middle point y. Done if $x = y$; if $x < y$, then drop the second half of A as it's ordered; otherwise drop the first half. When $A = [\ ]$, then x doesn't exist. A need be ordered, I often see people are struggled with unordered data, confusing why the binary search does not work. *'Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky'* said Donald Knuth. Jon Bentley said most binary search implementations had error, including the one he gave in '*Programming pearls*'. He corrected the error after two decades[2]. Below is the binary search definition, where the lower and upper bounds of A is l and u (exclude u).

---

[1]Blum, Floyd, Pratt, Rivest, and Tarjan developed a linear time algorithm in 1973[4] [81]. Split the elements into groups, each has 5 elements at most. It gives $n/5$ medians. Repeat this to pick the median of median.

[2]There's a 'mind reading' game in social network. One thinks about a person in mind. The AI robot asks 16 questions, and tells who that person is from the yes/no answers

$$bsearch\ x\ A\ (l, u) = \begin{cases} u < l : & Nothing \\ x = A[m] : & Just\ m,\ \text{where}\ m = l + \lfloor \frac{u - l}{2} \rfloor \\ x < A[m] : & bsearch\ x\ A\ (l, m - 1) \\ \text{otherwise} : & bsearch\ x\ A\ (m + 1, u) \end{cases} \tag{14.5}$$

Or implement with loops:

```
1: function BINARY-SEARCH(x, A, l, u)
2:     while l < u do
3:         m ← l + ⌊(u − l)/2⌋              ▷ avoid ⌊(l + u)/2⌋ overflow
4:         if A[m] = x then
5:             return m
6:         if x < A[m] then
7:             u ← m − 1
8:         else
9:             l ← m + 1
10:    Not found
```

The performance of binary search is bound to $O(\lg n)$ because it halves A every time. We can extend it to solve equation of monotone functions, for example $a^x = y$, where $a \leq y$, a and y are nature numbers. To find the integral x, exhaust $a^0, a^1, a^2, ...$, till $a^i = y$ or $a^i < y < a^{i+1}$ (no solution). If a and x are big numbers, it's expensive to compute a^x in loops[3]. Let's apply binary search. As $a^y \geq y$, we search in $[0, 1, ..., y]$. Function $f(x) = a^x$ is monotone, fix x, we examine the middle point: $x_m = \lfloor \frac{0 + y}{2} \rfloor$. If $a^{x_m} = y$, then x_m is the solution; if $a^m < y$, we discard the range before x_m; otherwise discard the range after x_m. Both halve the search range. When the range becomes empty, it means no solution. Below is the implementation. Denote the monotone function as f, call *bsearch* f y $(0, y)$, where $f(x) = a^x$ to start. This method computes $f(x)$ for $O(\lg y)$ times, better than the exhaustive search.

$$bsearch\ f\ y\ (l, u) = \begin{cases} u < l : & Nothing \\ f(m) = y : & Just\ m,\ \text{where}\ m = \lfloor \frac{l + u}{2} \rfloor \\ f(m) < y : & bsearch\ f\ y\ (m + 1, u) \\ f(m) > y : & bsearch\ f\ y\ (l, m - 1) \end{cases} \tag{14.6}$$

14.2.1 2D search

Extend binary search to 2D or even higher dimension. Consider matrix M of size $m \times n$. The elements in each row, column are ascending nature numbers as shown in fig. 14.1. How to locate all elements equal to z? i.e. find all locations of (i, j), such that $M_{i,j} = z$.

$$[(x, y) | x \leftarrow [1, 2, ..., m], y \leftarrow [1, 2, ..., n], M_{x,y} = z] \tag{14.7}$$

Richard Bird used to interview students with this question[1]. Those who had programming experience at school tended to apply binary search. But it was easy to get stuck. One often checks the middle point $M_{\frac{m}{2}, \frac{n}{2}}$. If it is less than z, then drop the top-left rectangle; if greater than z then drop the bottom-right rectangle, as shown in fig. 14.2, discard the shaded rectangle. Both cases lead to a L-shape search area, where one can't apply recursive search directly any more. Define the 2D search as: given $f(x, y)$, search integer solution (x, y), such that $f(x, y) = z$. The matrix search is just a special case as:

---

[3]One can reuse the result of a^n to compute $a^{n+1} = aa^n$. We consider generic monotone $f(n)$.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & ... \\ 2 & 4 & 5 & 6 & ... \\ 3 & 5 & 7 & 8 & ... \\ 4 & 6 & 8 & 9 & ... \\ ... \end{bmatrix}$$

Figure 14.1: Each row, column is ascending.
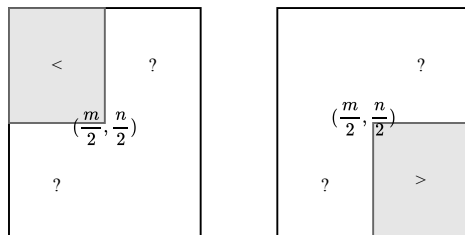
Figure 14.2: Left: the middle point $< z$, all shaded rectangle $< z$; Right: the middle point $> z$, all shaded rectangle $> z$.

$$f(x, y) = \begin{cases} 1 \le x \le m, 1 \le y \le n : & M_{x,y} \\ \text{otherwise} : & -1 \end{cases}$$

For monotone function $f(x, y)$, e.g., $f(x, y) = x^a + y^b$, where a, b are nature numbers, the effective method is to search from the top-left, but not bottom-left[82]. As shown in fig. 14.3, start from $(0, z)$, for each location (p, q), compare $f(p, q)$ and z:

1. If $f(p, q) < z$, since f is monotone increasing, $f(p, y) < z$ for all $0 \le y < q$. Drop all points in the vertical line segment (red);

2. If $f(p, q) > z$, then $f(x, q) > z$ for all $p < x \le z$. Drop all points in the horizontal line segment (blue);

3. If $f(p, q) = z$, then (p, q) is a solution. Drop both line segments.

Reduce the search rectangle line by line, every time drop a row, or a column, or both.
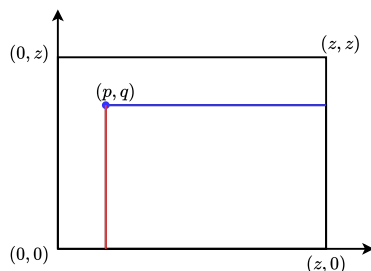
Figure 14.3: Search from top-left.

Define *search* function as below, and pass the top-left corner: *search f z 0 z*

$$
search\ f\ z\ p\ q\ =\ \begin{cases} p > z \text{ or } q < 0: & [\ ] \\ f(p,q) < z: & search\ f\ z\ (p+1)\ q \\ f(p,q) > z: & search\ f\ z\ p\ (q-1) \\ f(p,q) = z: & (p,q): search\ f\ z\ (p+1)\ (q-1) \end{cases} \tag{14.8}
$$

Every time, at least one of p and q advances towards the bottom or right by one. It needs at most $2(z+1)$ steps. There are three best cases: (1) both p and q advance one a time, there are total $z+1$ steps. As in fig. 14.4 (a), all points in the diagonal line $(x, z-x)$ satisfy $f(x, z-x) = z$. It reaches to $(z, 0)$ in $z+1$ steps; (2) move to the right horizontally till p exceeds z. As in fig. 14.4 (b), all points in the top horizontal line (x, z) satisfy $f(x, z) < z$. It terminates after $z+1$ steps; (3) move down vertically till q becomes negative. As in fig. 14.4 (c), all points in the left vertical line $(0, x)$ satisfy $f(0, x) > z$. It terminates after $z+1$ steps; (d) is the worst case. If project all the horizontal sections in the search path to x axis, all the vertical sections to y axis, it gives the total steps of $2(z+1)$. This method improves the performance of exhaustive search from $O(z^2)$ to $O(z)$.
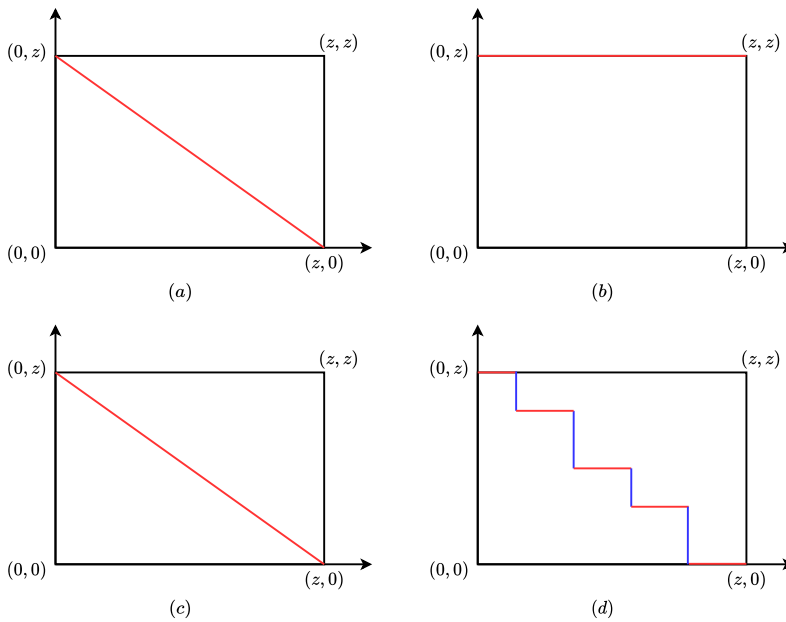
Figure 14.4: The best and worst cases.

This algorithm is called the *'saddle back'* search. The plot image of f has the smallest bottom-left and the largest top-right. It looks like a saddle with two wings as shown in fig. 14.5. We can further reduce the search rectangle of $(0, z) - (z, 0)$. Since f is monotone increasing, find the maximum m along y axis satisfying $f(0, m) \le z$; find the maximum n along x axis satisfying $f(n, 0) \le z$. Reduce the search rectangle to $(0, m) - (n, 0)$, as shown in fig. 14.6.

$$
\begin{cases} m & = \max\ \{0 \le y \le z, f(0, y) \le z\} \\ n & = \max\ \{0 \le x \le z, f(x, 0) \le z\} \end{cases} \tag{14.9}
$$

We can apply binary search to find m, n (fix $x = 0$ to search m, fix $y = 0$ to search n). Modify eq. (14.6), search $l \le x \le u$ satisfying $f(x) \le y < f(x+1)$.
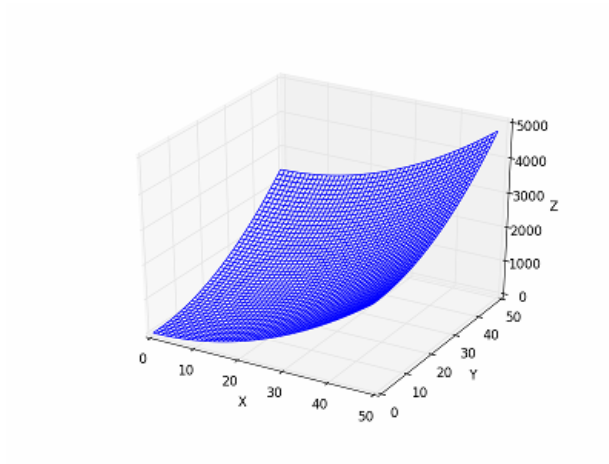
Figure 14.5: Plot of $f(x, y) = x^2 + y^2$.
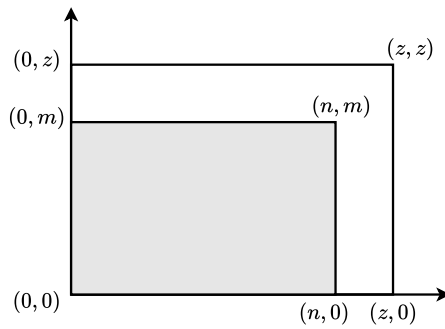
Figure 14.6: Reduced search rectangle.

$$
bsearch\ f\ y\ (l, u) = \begin{cases} u \le l : & l \\ f(m) \le y < f(m+1) : & m, \text{where } m = \lfloor \dfrac{l+u}{2} \rfloor \\ f(m) \le y : & bsearch\ f\ y\ (m+1, u) \\ f(m) > y : & bsearch\ f\ y\ (l, m-1) \end{cases} \tag{14.10}
$$

Then determine m, n with binary search:

$$
\begin{cases} m &= bsearch\ (y \mapsto f(0, y))\ z\ (0, z) \\ n &= bsearch\ (x \mapsto f(x, 0))\ z\ (0, z) \end{cases} \tag{14.11}
$$

Finally, apply saddle back search in this smaller rectangle: $solve(f, z) = search\ f\ z\ 0\ \boldsymbol{m}$

$$
search\ f\ z\ p\ q = \begin{cases} p > \boldsymbol{n} \text{ or } q < 0 : & [\ ] \\ f(p, q) < z : & search\ f\ z\ (p+1)\ q \\ f(p, q) > z : & search\ f\ z\ p\ (q-1) \\ f(p, q) = z : & (p, q) : search\ f\ z\ (p+1)\ (q-1) \end{cases} \tag{14.12}
$$

We apply two rounds of binary search to find m, n, each round computes f for $O(\lg z)$ times; The saddle back search computes f for $O(m + n)$ times in the worst case; it's $O(\min(m, n))$ in the best case as below table. For functions like $f(x, y) = x^a + y^b$, $a, b \in \mathbb{N}$, the boundary m, n are very small. The total performance is close to $O(\lg z)$.

	steps to compute f
worst	$2 \log z + m + n$
best	$2 \log z + \min(m, n)$

As shown in fig. 14.7, for a point (p, q) in rectangle $(a, b) - (c, d)$, if $f(p, q) \ne z$, we can only discard the shaded part ($\le 1/4$). If $f(p, q) = z$, we can discard the bottom-left, top-right parts, and all points in row p and column q since f is monotone. Hence reduce the search rectangle by $1/2$. To find the point satisfying $f(p, q) = z$, we apply binary search along the horizontal or vertical central line. Because the performance is bound to $O(\lg |L|)$ for line L, we chose the shorter central line as shown in fig. 14.8.

If there is no point satisfying $f(p, q) = z$, find a point, such that $f(p, q) < z < f(p + 1, q)$ in the horizontal central line (or $f(p, q) < z < f(p, q + 1)$ for vertical central line). We can't discard all points in row p and column q. In summary, we apply binary search along horizontal central line for the point: $f(p, q) \le z < f(p + 1, q)$; or search the vertical central line for the point: $f(p, q) \le z < f(p, q + 1)$. If all points in the line segment are $f(p, q) < z$, then return the upper bound; if all are $f(p, q) > z$, then return the lower bound. We discard half side in this case. Below is the improved saddle back search:

1. Apply binary search along the x, y axes for the search rectangle $(0, m) - (n, 0)$;

2. For rectangle $(a, b) - (c, d)$, if the height > width, apply binary search along the horizontal central line; otherwise search along the vertical central line for the point (p, q);

3. If $f(p, q) = z$, it is a solution. Recursively search rectangles $(a, b) - (p - 1, q + 1)$ and $(p + 1, q - 1) - (c, d)$;
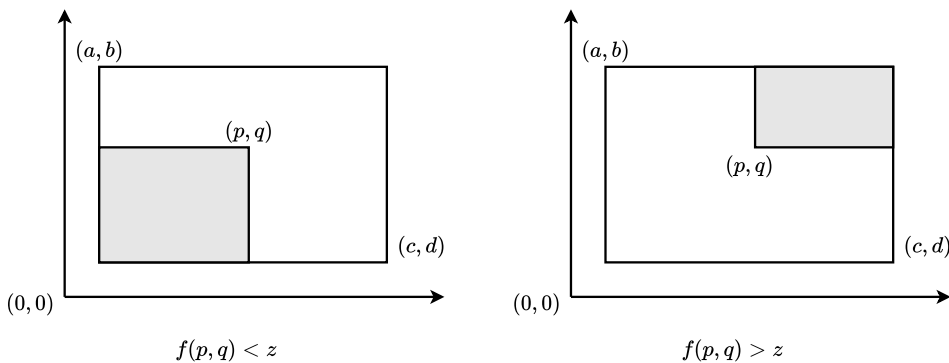
(a) If $f(p, q) \neq z$, we can only drop the shaded area, the remaining is a 'L' shape.
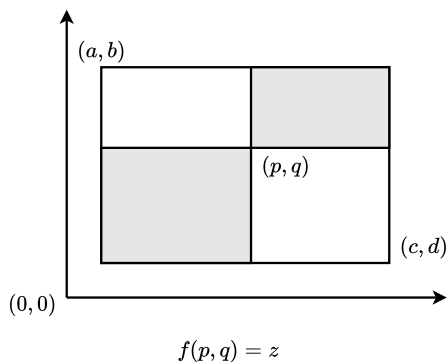
(b) If $f(p, q) = z$, we can drop $1/2$ rect-angle.

Figure 14.7: Reduce the search rectangle.
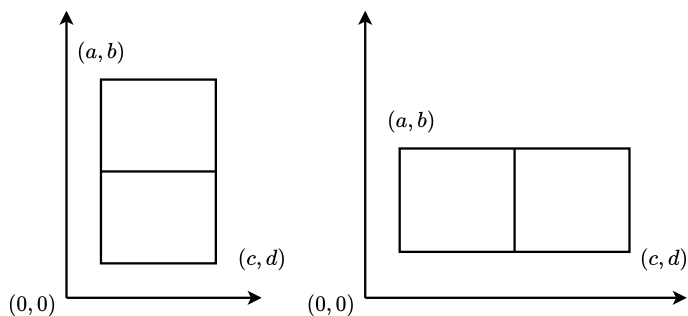
Figure 14.8: Chose the shorter center line.
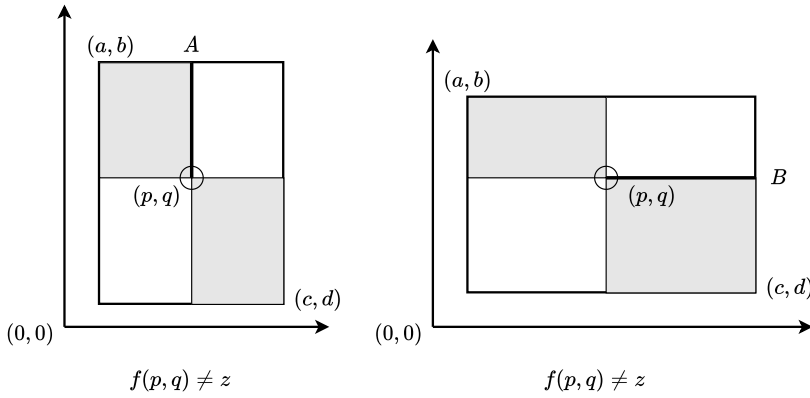
Figure 14.9: Recursively search the shaded parts, include the bold line if $f(p, q) \neq z$.

4. If $f(p, q) \neq z$, recursively search the two rectangles and a line section, either $(p, q + 1) - (p, b)$ in fig. 14.9 (a); or $(p + 1, q) - (c, q)$ in fig. 14.9 (b).

$$ search\ (a, b)\ (c, d) = \begin{cases} c < a \text{ or } d < b: & [\ ] \\ c - a < b - d: & csearch \\ \text{otherwise}: & rsearch \end{cases} \tag{14.13} $$

Where *csearch* apply binary search to the horizontal central line for point (p, q), such that $f(p, q) \leq z < f(p + 1, q)$, as shown in fig. 14.9 (a). If all function values are greater than z, then return the lower bound $(a, \lfloor \frac{b+d}{2} \rfloor)$. Drop the above side (include the central line) as shown in fig. 14.10 (a).
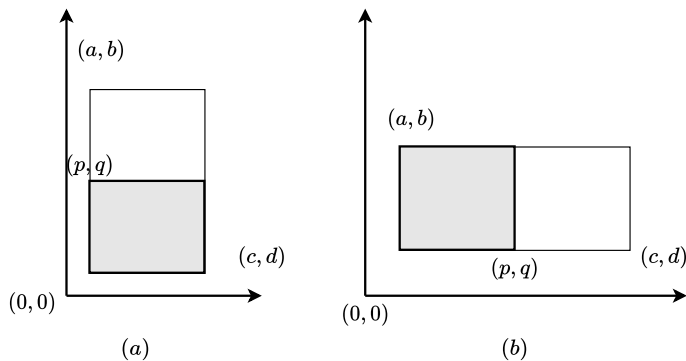
Figure 14.10: Special case.

Let

$$ \begin{cases} q = \lfloor \dfrac{b+d}{2} \rfloor \\ p = bsearch\ (x \mapsto f(x, q))\ z\ (a, c) \end{cases} $$

$$
csearch = \begin{cases} f(p,q) > z : & search\ (p, q-1)\ (c,d) \\ f(p,q) = z : & search\ (a,b)\ (p-1, q+1) +\!\!+ [(p,q)] +\!\!+ search\ (p+1, q-1)\ (c,d) \\ f(p,q) < z : & search\ (a,b)\ (p, q+1) +\!\!+ search\ (p+1, q-1)\ (c,d) \end{cases}
$$

$$(14.14)$$

Function $rsearch$ is symmetric along the vertical central line. Below example program implements the improved saddle back search:

```
solve f z = search f z (0, m) (n, 0) where
  m = bsearch (f 0) z (0, z)
  n = bsearch (λx → f x 0) z (0, z)

search f z (a, b) (c, d)
      | c < a || b < d = []
      | c - a < b - d = let q = (b + d) `div` 2 in
          csearch (bsearch (λx → f x q) z (a, c), q)
      | otherwise = let p = (a + c) `div` 2 in
          rsearch (p, bsearch (f p) z (d, b))
  where
    csearch (p, q)
        | z < f p q = search f z (p, q - 1) (c, d)
        | f p q == z = search f z (a, b) (p - 1, q + 1) +
                (p, q) : search f z (p + 1, q - 1) (c, d)
        | otherwise = search f z (a, b) (p, q + 1) +
                search f z (p + 1, q - 1) (c, d)
    rsearch (p, q)
        | z < f p q = search f z (a, b) (p - 1, q)
        | f p q == z = search f z (a, b) (p - 1, q + 1) +
                (p, q) : search f z (p + 1, q - 1) (c, d)
        | otherwise = search f z (a, b) (p - 1, q + 1) +
                search f z (p + 1, q) (c, d)
```

As we halve the rectangle every time, we search $O(\lg(mn))$ rounds. We apply binary search along the central line for (p,q), compute f for $O(\lg(\min(m,n)))$ times. Let the time be $T(m,n)$ when search $m \times n$ rectangle. We have the following recursive equation:

$$T(m,n) = \lg(min(m,n)) + 2T(\frac{m}{2}, \frac{n}{2}) \tag{14.15}$$

Suppose $m = 2^i > n = 2^j$, use telescope method:

$$
\begin{aligned}
T(2^i, 2^j) &= j + 2T(2^{i-1}, 2^{j-1}) \\
&= \sum_{k=0}^{i-1} 2^k (j - k) \\
&= O(2^i (j - i)) \\
&= O(m \lg(n/m))
\end{aligned}
\tag{14.16}
$$

Richard Bird proves this is asymptotically optimal by a lower bound of searching a given value in $m \times n$ rectangle[1].

Exercise 14.1

14.1.1. Prove the performance of k-selection problem is $O(n)$ in average.

14.1.2. To find the top k element in A, we can search $x = \max\ (take\ k\ A), y = \min\ (drop\ k\ A)$. If $x < y$, then the first k elements in A is the answer; otherwise, we partition the first k elements with x, partition the rest with y, then recursively find in sub-sequence $[a \leftarrow A, x < a < y]$ for the top k' elements, where $k' = k - \|[a \leftarrow A, a \leq x]\|$. Implement this solution, and evaluate its performance.

14.1.3. Find the 'simplified' median of two sorted arrays A and B in $O(\lg(m+n))$ time, where $m = |A|, n = |B|$. The array index starts from 0. The simplified median is defined as $median(A, B) = C[\lfloor \dfrac{m+n}{2} \rfloor]$, where $C = merge(A, B)$ is the merged sorted array[4].

14.1.4. For the saddle back search, eliminate recursion, implement it in loops to update the boundaries.

14.1.5. For 2D search, let the bottom-left be the minimum, the top-right be the maximum. if z is less than the minimum or greater than the maximum, then no solution; otherwise cut the rectangle into 4 parts with a horizontal line and a vertical line crossed at the center. then recursive search in these 4 small rectangles. Implement this solution and evaluate its performance.

14.3   The majority number

People often vote and use computer to count the result. Suppose a candidate wins if and only if gets more than half votes. From the votes sequence A, B, A, C, B, B, D, ..., how to find the winner efficiently? We can use a map to count the result (see chapter 2)[6].

```
Optional<T> majority([T] xs) {
    Map<T, Int> m
    for var x in xs {
        if x in m then m[x]++ else mx[x] = 0
    }
    var (r, v) = (Optional<T>.Nothing, length(xs) / 2 - 1)
    for var (x, c) in m {
      if c > v then (r, v) = (Optional.of(x), c)
    }
    return r
}
```

The map is often implemented as a red-black tree or Hash table. For m candidates, n votes, below table gives the performance:

map	time	space
self-balancing tree	$O(n \lg m)$	$O(m)$
Hash table	$O(n)$	at least $O(m)$

Define the element occurs over 50% as the 'majority'. Boyer and Moore developed an algorithm in 1980, which picks the majority element in one scan if there is with constant space[83]. There is at most 1 majority. Repeat dropping two different elements till all remaining are same. If the majority exists, then it is the remaining. Start from the first vote, let the candidate be the winner so far with point 1. If the next one votes the same candidate, then add the winner point by 1, otherwise -1. The candidate won't be the winner when the point reduces to 0. We pick the candidate of the next vote as the

---

[4]In statistics, the median of an ascending data set x with n elements is defined as:

$$median(x) = \begin{cases} odd(n): & x[\frac{n+1}{2}] \\ even(n): & \frac{1}{2}(x[\frac{n}{2}] + x[\frac{n}{2} + 1]) \end{cases}$$

[6]There is a probabilistic sub-linear space counting algorithm published in 2004, named as 'Count-min sketch'[84].

new winner and go on. As shown in below table, if there exists majority m, then other candidates can't beat m. Otherwise if the majority doesn't exist (invalid vote result, no winner), then discard the recorded 'winner'. We need another scan to valid the winner.

winner	count	position
A	1	**A**, B, C, B, B, C, A, B, A, B, B, D, B
A	0	A, **B**, C, B, B, C, A, B, A, B, B, D, B
C	1	A, B, **C**, B, B, C, A, B, A, B, B, D, B
C	0	A, B, C, **B**, B, C, A, B, A, B, B, D, B
B	1	A, B, C, B, **B**, C, A, B, A, B, B, D, B
B	0	A, B, C, B, B, **C**, A, B, A, B, B, D, B
A	1	A, B, C, B, B, C, **A**, B, A, B, B, D, B
A	0	A, B, C, B, B, C, A, **B**, A, B, B, D, B
A	1	A, B, C, B, B, C, A, B, **A**, B, B, D, B
A	0	A, B, C, B, B, C, A, B, A, **B**, B, D, B
B	1	A, B, C, B, B, C, A, B, A, B, **B**, D, B
B	0	A, B, C, B, B, C, A, B, A, B, B, **D**, B
B	1	A, B, C, B, B, C, A, B, A, B, B, D, **B**

$$
\begin{aligned}
maj\ [\ ] &= \varnothing \\
maj\ (x{:}xs) &= scan\ (x, 1)\ xs
\end{aligned}
\tag{14.17}
$$

Where $scan$ is defined as:

$$
\begin{aligned}
scan\ (m, v)\ [\ ] &= m \\
scan\ (m, v)\ (x{:}xs) &=
\begin{cases}
m = x: & scan\ (m, v+1)\ xs \\
v = 0: & scan\ (x, 1)\ xs \\
\text{otherwise}: & scan\ (m, v-1)\ xs
\end{cases}
\end{aligned}
\tag{14.18}
$$

Or implement with fold (Curried form): $maj = foldr\ f\ (\varnothing, 0)$, where:

$$
f\ x\ (m, v) =
\begin{cases}
x = m: & (m, v+1) \\
v = 0: & (x, 1) \\
\text{otherwise}: & (m, v-1)
\end{cases}
\tag{14.19}
$$

Finally, verify the winner is the true majority:

$$
verify\ m = \text{if}\ 2|filter\ (= m)\ xs| > |xs|\ \text{then}\ Just\ m\ \text{else}\ Nothing
\tag{14.20}
$$

Below is the corresponding iterative implementation:

```
1: function MAJORITY(A)
2:     c ← 0, m ← ∅
3:     for each a in A do
4:         if c = 0 then
5:             m ← a
6:         if a = m then
7:             c ← c + 1
8:         else
9:             c ← c − 1
```

```
10:      c ← 0
11:      for each a in A do
12:          if a = m then
13:              c ← c + 1
14:      if c > %50|A| then
15:          return Just m
16:      else
17:          return Nothing
```

Exercise 14.2

14.2.1. Extend to find k majorities that occurs over $\lfloor n/k \rfloor$ times in collection A, where $n = |A|$. Hint: Drop k different elements every time, till the remaining is less than k distinct candidates. Any k-majority (the one over $\lfloor n/k \rfloor$) must remain in the end.

14.4   Maximum sum of sub-vector

For vector V, define a range $V[i...j]$ as sub-vector, the sum of sub-vector is $S = V[i] + V[i+1] + ... + V[j]$. Empty $[\ ]$ is sub-vector of any vector with sum 0. How to find the maximum sum of a given vector V [2]? For example, in vector [3, -13, 19, -12, 1, 9, 18, -16, 15, -15], the sub-vector [19, -12, 1, 9, 18] gives the maximum sum of 35. If all elements are positive, then the max is the total sum. If all are negative, then the empty vector gives the max sum of 0. Below is the exhaustive search implementation:

```
1: function MAX-SUM(V)
2:      m ← 0, n ← |V|
3:      for i ← 1 to n do
4:          s ← 0
5:          for j ← i to n do
6:              s ← s + V[j]
7:              m ← MAX(m, s)
8:      return m
```

The performance of exhaustive search is $O(n^2)$, where n is the vector length. Similar to majority number algorithm, we scan the vector. For every position i, record the sum of sub-vector ends with i as A, and the maximum sum so far as B. As shown in fig. 14.11. A is not necessarily equal to B. We maintain $B \leq A$ always hold. When $B + V[i] > A$, we replace A with this greater value. When $B + V[i] < 0$, we reset B to 0. Below table gives the steps when scan $[3, -13, 19, -12, 1, 9, 18, -16, 15, -15]$.
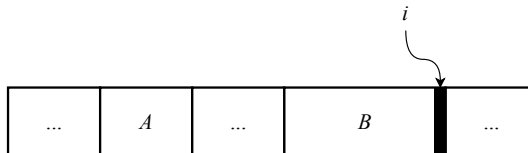
Figure 14.11: A: max sum so far; B: sum of the sub-vector ends with i.

max sum	max end at i	yet to scan
0	0	$[3, -13, 19, -12, 1, 9, 18, -16, 15, -15]$
3	3	$[-13, 19, -12, 1, 9, 18, -16, 15, -15]$
3	0	$[19, -12, 1, 9, 18, -16, 15, -15]$
19	19	$[-12, 1, 9, 18, -16, 15, -15]$
19	7	$[1, 9, 18, -16, 15, -15]$
19	8	$[9, 18, -16, 15, -15]$
19	17	$[18, -16, 15, -15]$
35	35	$[-16, 15, -15]$
35	19	$[15, -15]$
35	34	$[-15]$
35	19	$[\,]$

1: **function** MAX-SUM(V)
2:     $A \leftarrow 0, B \leftarrow 0, n \leftarrow |V|$
3:     **for** $i \leftarrow 1$ to n **do**
4:         $B \leftarrow$ MAX($B + V[i], 0$)
5:         $A \leftarrow$ MAX(A, B)
6:     **return** A

Or implement with fold (Curried form): $S_{max} = fst \circ foldr \ f \ (0, 0)$, where f update the maximum sum so far:

$$f \ x \ (S_m, S) = (S'_m = \max(S_m, S'), S' = \max(0, x + S)) \tag{14.23}$$

Exercise 14.3

14.3.1. Modify the solution that finds the max sum of sub-vector, returns the sub-vector of the maximum sum.

14.3.2. Bentley gives a divide and conquer algorithm to find the max sum in $O(n \lg n)$ time[2]. Split the vector at middle, recursively find the max sum of the two halves, and the max sum that crosses the middle. Then pick the greatest. Implement this solution.

14.3.3. Find the sub-metrics in a $m \times n$ metrics that gives the maximum sum.

14.5   String matching

String matching is widely used in editor applications. We can use data structures like radix tree, prefix tree (see chapter 6) to search, or directly match the string as shown in fig. 14.12[7]. We match a pattern P in text T character by character. At offset $s = 4$, the first 4 are same as shown in fig. 14.12 (a), the 5th is 'y' in P, but 't' in T. We stop here, add s by 1 (move P to right by 1), then restart matching 'ananym' and 'nantho...'. Actually, we can increase s more than 1. The first two characters 'an' happen to be the suffix of 'anan'. We can add s by 2 as shown in fig. 14.12 (b). We reuse the information from the 4 matched characters, skip some positions. Knuth, Morris and Pratt develope an efficient matching algorithm from this idea[85], known as 'KMP'. the initials of the three authors.

---

[7]Some programming environment provide match tool, like `strstr` in C library, `find` in C++ library, `indexOf` in Java library.

(a) The offset $s = 4$, after matching $q = 4$ characters, the 5th mismatches.

(b) Move $s = 4 + 2 = 6$.
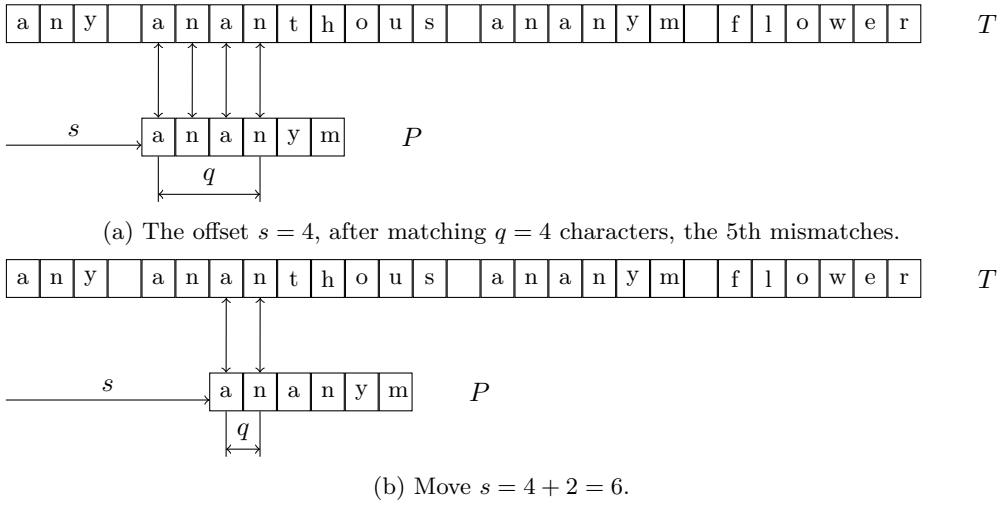
Figure 14.12: Match 'ananym' in 'any ananthous ananym flower'.

Denote the first k characters of text T as T_k (the k-character prefix of T). To shift P to the right s steps as far as possible, we need reuse the information of the matched q characters. As shown in fig. 14.13, if we can shift P ahead, there exists some k, such that the first k characters are same as the last k characters of P_q, i.e., the prefix P_k is suffix of P_q. Define empty "" be both the prefix and suffix of any string, hence the minimum $k = 0$ always exists. We need find the maximum k for the string that is both the prefix and suffix. Define the *prefix function* $\pi(q)$, that gives where to fallback when the $(q + 1)$-th character doesn't match[4].
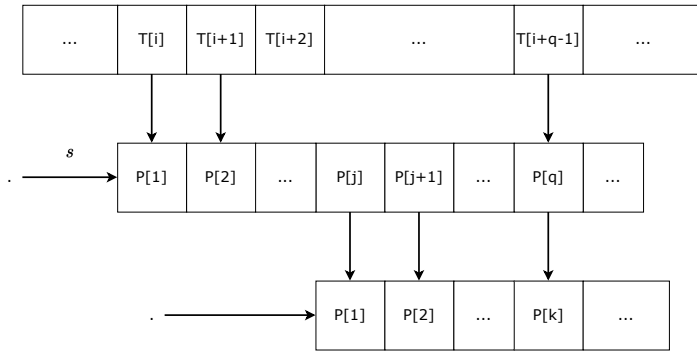
Figure 14.13: P_k is both the prefix and suffix of P_q.

$$\pi(q) = \max\{k | 0 \le k < q, \text{and } P_k \text{ is suffix of } P_q\} \qquad (14.24)$$

When match pattern P against text T from offset s, if fails after matching q characters, we next look up $q' = \pi(q)$ to get a fallback position q'. Then retry to compare $P[q']$ with the text:

1: **function** KMP(T, P)
2:     $\pi \leftarrow$ BUILD-PREFIXES(P)
3:     $n \leftarrow |T|, m \leftarrow |P|, q \leftarrow 0$
4:     **for** $i \leftarrow 1$ to n **do**

5:          **while** $q > 0$ and $P[q + 1] \neq T[i]$ **do**
6:              $q \leftarrow \pi(q)$
7:          **if** $P[q + 1] = T[i]$ **then**
8:              $q \leftarrow q + 1$
9:          **if** $q = m$ **then**
10:             position $i - m$ is a solution
11:             $q \leftarrow \pi(q)$ ▷ search further

  The definition eq. (14.24) is not practical to build $\pi(q)$. Let us pre-process P. If the first character doesn't match, then the longest prefix and suffix is empty: $\pi(1) = 0$, i.e., $P_k = P_0 = $ "". When scan the q-th character in P, the prefix function values $\pi(i)$, $i = 1, 2, ..., q - 1$, are already known, and so far, the longest prefix P_k is also the suffix of P_{q-1}. As shown in fig. 14.14, if $P[q] = P[k + 1]$, we find a greater k, and increase k by 1; otherwise, if $P[q] \neq P[k + 1]$, then lookup $\pi(k)$ and fallback to a shorter prefix $P_{k'}$, where $k' = \pi(k)$. Then compare the next character of this new prefix with the q-th character. Repeat this till k becomes zero (empty string), or the q-th character matches. Below table gives the prefix function values of 'ananym'. The k column is the maximum satisfying eq. (14.24).
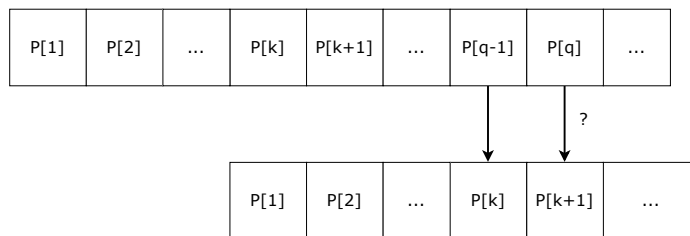
Figure 14.14: P_k is suffix of P_{q-1}, compare $P[q]$ and $P[k + 1]$.

q	P_q	k	P_k
1	a	0	""
2	an	0	""
3	ana	1	a
4	anan	2	an
5	anany	0	""
6	ananym	0	""

1: **function** BUILD-PREFIXES(P)
2:     $m \leftarrow |P|, k \leftarrow 0$
3:     $\pi(1) \leftarrow 0$
4:     **for** $q \leftarrow 2$ to m **do**
5:         **while** $k > 0$ and $P[q] \neq P[k + 1]$ **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** π

  The KMP algorithm builds the prefix function in amortized $O(m)$ time [4], and matches the string in amortized $O(n)$ time, where $m = |P|$, $n = |T|$ are the lengths. The total

amortized performance is $O(m + n)$, with additional $O(m)$ space to store the prefix function. Varies of pattern string P don't impact the performance. Consider matching pattern 'aaa...ab' (length of m) in string 'aaa...a' (length of n). The m-th character doesn't match, we can only fallback by 1 repeatedly. The algorithm is still bound to linear time in this case.

14.6 Solution search

In early years of artificial intelligent, people developed methods to search solutions. Different from sequence and string matching, the solution may not directly exist among a set of candidates. We need construct the solution while try varies of options. Some problems are not solvable. Among the solvable ones, there can be multiple solutions. For example, a maze may have multiple ways out. We need find the optimal solution sometimes.

14.6.1 DFS and BFS

DFS stands for deep-first search, and BFS stands for breadth-first search. They are typical graph search algorithms. We give some examples and skip the formal definition of graph.

Maze

Maze is a classic puzzle. There is saying: always turn right. However, it ends up loops as shown in fig. 14.15 (b). The decision matters when there are multiple ways. In fairy tales, one takes some bread crumbs in a maze. Select a way, leave a piece of bread. If later enters a died end, then goes back to the last place through the bread crumbs. then goes to another way. Whenever sees bread crumbs left, one knows he visited it before. Then goes back and tries a different way. Repeats the 'try and check' step, one will either find the way out, or go back to the starting point (no solution). We use $m \times n$ matrix to define a maze, each element is 0 or 1, means there is a way or not. Below matrix defines the maze in fig. 14.15 (b):
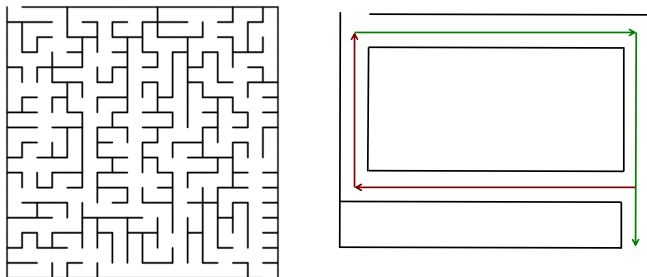
(a) Maze   (b) Loop when keep turning right.

Figure 14.15: Maze

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Given a start point $s = (i, j)$, a destination $e = (p, q)$, we need find all paths from s to e. We first examine all points connected with s. For every such point k, recursively find all paths from k to e. Then prepend path s-k to every path from k to e. We need leave some 'bread crumbs' to avoid looping. We use a list P to record all visited points. Look it up and only try new ways.

$$solveMaze\ M\ s\ e = solve\ s\ [[\ ]] \tag{14.25}$$

Where:

$$solve\ s\ P = \begin{cases} s = e: & map\ (reverse \circ (s :))\ P \\ otherwise: & concat\ [solve\ k\ (map\ (s :)\ P)|k \leftarrow adj\ s, k \notin P] \end{cases} \tag{14.26}$$

The paths in P are reversed, we *reverse* the result back finally. Function $adj\ p$ enumerates adjacent points to p:

$$adj\ (x, y) = \begin{array}{l} [\leftarrow [(x-1, y), (x+1, y), (x, y-1), (x, y+1)], \\ 1 \leq x' \leq m, 1 \leq y' \leq n, M_{x'y'} = 0] \end{array} \tag{14.27}$$

This essentially 'exhaustive searches' all possible paths. We only need one way out. We need some data structure serves for the 'bread crumbs', recording the previous decisions. We always search on top of the latest decision. We can use stack to realize the last-in, first-out order. The stack starts from $[s]$. Pop s out and find all connected points of $a, b, ...,$ push the new paths $[a, s], [b, s]$ to the stack. Next pop $[a, s]$ out, examine all points connected to a. Then push new paths consist of 3 steps to the stack. Repeat this. The stack records paths in reversed order: from the farthest place back to the starting point, as shown in fig. 14.16. If the stack becomes empty, we've tried all ways, and terminate the search; otherwise, we pop a path, expand to new adjacent points, and push the new paths back.
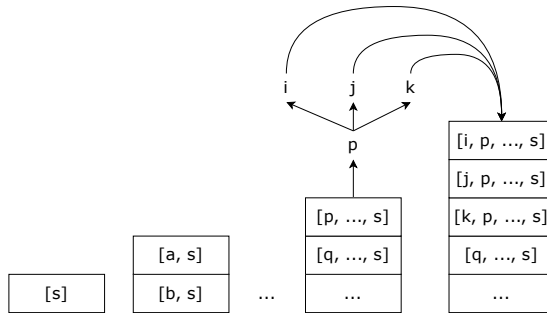
Figure 14.16: Search with a stack

$$solveMaze\ M\ s\ e = solve\ [[s]] \tag{14.28}$$

Where:

$$solve \: [\,] \quad = \quad [\,]$$

$$solve \: ((p{:}ps){:}cs) \quad = \quad \begin{cases} c = e: & reverse \: (p{:}ps) \\ ks = [\,]: & solve \: cs, \text{where } ks = filter \: (\notin ps) \: (adj \: p) \\ ks \neq [\,]: & solve \: ((map \: (:p{:}ps) \: ks) \mathbin{+\!\!+} cs) \end{cases}$$

$$(14.29)$$

Below is the iterative implementation:

1: **function** SOLVE-MAZE(M, s, e)
2:     $S \leftarrow [s], L = [\,]$
3:     **while** $S \neq [\,]$ **do**
4:         $P \leftarrow$ POP(S)
5:         $p \leftarrow$ LAST(P)
6:         **if** $e = p$ **then**
7:             ADD(L, P)   ▷ find a solution
8:         **else**
9:             **for** each k in ADJACENT(M, p) **do**
10:                 **if** $k \notin P$ **then**
11:                     PUSH($S, P \mathbin{+\!\!+} [k]$)
12:     **return** L

Each step tries 4 options (up, down, left, and right) through the backtrack. It seems the performance is $O(4^n)$, where n is the length of the path. The actual time won't be so large because we skip the visited places. In the worst case, we traverse all the reachable points *exactly once*. Hence the time is bound to $O(n)$, where n is the number of connected points. We need additional $O(n^2)$ space for the stack.

Exercise 14.4

14.4.1. Modify the implementation with stack, find all ways to the maze.

The eight queens puzzle

Although cheese has very long history, it was late in 1848, that Max Bezzel gave the 8 queens puzzle[89]. The queue is a powerful piece. It attacks any other pieces in the same row, column or diagonal at any distance, as shown in fig. 14.17 (a). How to arrange 8 queens in the cheese board, such that none of them attacks each other? Figure 14.17 (b) gives a solution.
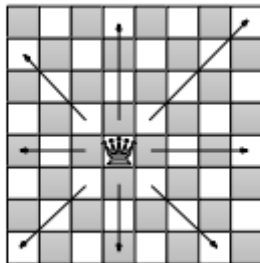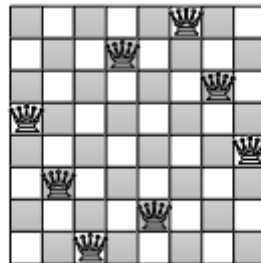
(a) Queen   (b) A solution

Figure 14.17: The eight queens puzzle.

There are total P^8_{64} permutations to arrange 8 queens in 64 cells, about 4×10^{10}. Since no two queens can be in the same row or column, a solution must be a permutation of

$[1, 2, 3, 4, 5, 6, 7, 8]$. For example, the permutation $[6, 2, 7, 1, 3, 5, 8, 4]$ means the first queen is at row 1, column 6, the second queen is at row 2 column 2, ..., and the 8th queen is at row 8, column 4. As such, we reduce the solution domain to $8! = 40320$ permutations. Arrange queues from the first row, there are 8 options (columns). For the next queue, we need skip some columns to avoid attacking the first queue. For the i-th queue, find the columns at row i, that not being attacked by the first $i - 1$ queues. If all 8 columns are invalid, then go back to adjust the previous $i - 1$ queues. We find a solution after arranged all 8 queues. Record it and search/backtrack further to find all solutions. Start searching with an empty stack and list: $solve\ [[\ ]]\ [\ ]$

$$solve\ [\ ]\ s\ =\ s$$
$$solve\ (c{:}cs)\ s\ =\ \begin{cases} |c| = 8: & solve\ cs\ (c{:}s) \\ \text{otherwise}: & solve\ ([x{:}c|x \leftarrow [1..8], x \notin c, safe\ x\ c] + cs)\ s \end{cases}$$
$$(14.30)$$

We've exhausted all options when the stack becomes empty, s records all the solutions; If the arrangement c of the stack top has length of 8, we add this newly find solution to s, then continue search; if $|c| < 8$, find the columns that are not occupied ($x \notin c$), or attacked by other queues in diagonal (through $safe\ x\ c$). Then push the new valid arrangements to the stack.

$$safe\ x\ c = \forall (i, j) \leftarrow zip\ (reverse\ c)\ [1, 2, ...] \Rightarrow |x - i| \neq |y - j|, \text{where } y = 1 + |c| \quad (14.31)$$

Function $safe$ checks if the queue at row $y = 1 + |c|$, column x is in the diagonal with any other queue. Let $c = [i_{y-1}, i_{y-2}, ..., i_1]$ be the columns of the first $y - 1$ queues. Reverse c, zip with 1, 2, ... to form coordinates: $[(i_1, 1), (i_2, 2), ..., (i_{y-1}, y - 1)]$. Then check every (i, j) forms a diagonal with (x, y): $|x - i| \neq |y - j|$. This implementation is tail recursive, we can turn it into loops:

```
 1: function SOLVE-QUEENS
 2:     S ← [[ ]]
 3:     L ← [ ]                                            ▷ Stores the solution
 4:     while S ≠ [ ] do
 5:         A ← POP(S)                                     ▷ A: arrangement
 6:         if |A| = 8 then
 7:             ADD(L, A)
 8:         else
 9:             for i ← 1 to 8 do
10:                 if VALID(i, A) then
11:                     PUSH(S, A ++ [i])
12:     return L

13: function VALID(x, A)
14:     y ← 1 + |A|
15:     for i ← 1 to |A| do
16:         if x = A[i] or |y − i| = |x − A[i]| then
17:             return False
18:     return True
```

We only try the unoccupied ones out of the 8 columns, in total 15,720 arrangements. It is less than $8! = 40,320$[89]. Because the square board is horizontal and vertical symmetric, when find a solution, we can rotate and flip to obtain other symmetric solutions (see Exercise 14.5.2). We can expand to solve n queues puzzle ($n \geq 4$). However, the time increases fast along with n. The backtrack algorithm is slightly better than the exhaustive permutations (bound to $o(n!)$).

Exercise 14.5

14.5.1. Extend the 8 queens to n queens.

14.5.2. There are 92 solutions to the 8 queens puzzle. For any solution, it's also a solution if rotates 90°. We can flip to get another solution. There are essentially 12 distinct solutions. Write a program to find them.

The peg puzzle

As shown in fig. 14.18, there are 6 frogs in 7 stones, grouped on two sides. Each frog can hop to the next stone if not occupied, or leap over to another empty one. The frogs can only move forward or stay, but not go back. Figure 14.19 give the rules. How to arrange the frogs to hop, leap, such that the left and right sides swap? Mark the left frogs as -1, the right as 1, the empty stone as 0. We are seeking the solution that transitions from $s = $ `[-1, -1, -1, 0, 1, 1, 1]` to $e = $ `[1, 1, 1, 0, -1, -1, -1]`.
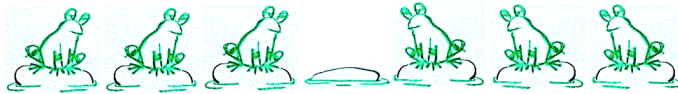
Figure 14.18: The leap frogs puzzle.

(a) Hop to the next stone

(b) Leap over to the right

(c) Leap over to the left
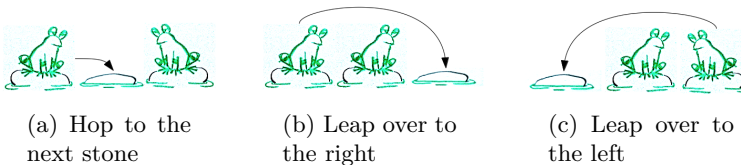
Figure 14.19: Moving rules.

This is a special form of the peg puzzle. The number of pegs can be 8 or other even numbers (as shown in Figure 14.20[8]).

(a) Solitaire

(b) Hop over

(c) Draught board
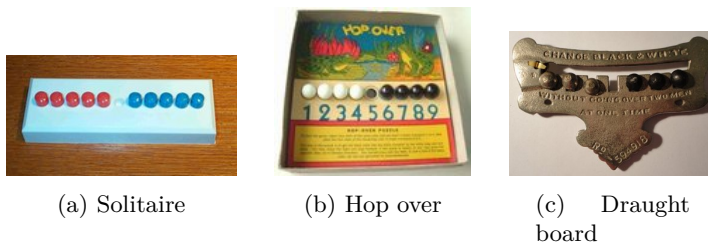
Figure 14.20: Variants of the peg puzzle

Label the stones from left as 1, 2, ..., 7. There are at most 4 options for every move. When start for example, the frog on the 3rd stone can hop right to the empty stone; the

---

[8]from http://home.comcast.net/~stegmann/jumping.htm

frog on the 5th stone can hop left; the frog on the 2nd stone can leap right, the frog on the 6th stone can leap left. We record the stone status and try the 4 options at every step. Backtrack and try other options when get stuck. Because every frog can only move forward, the movement is not revertible. Hence, we needn't worry about repetition. We record the steps only for the final output. State L is some permutation of s. $L[i]$ is $\pm 1, 0$, indicates there is a frog on the i-th stone heading left, right, or the stone is empty. Let the empty stone be p, the 4 movements are:

1. Leap left: $p < 6$ and $L[p+2] > 0$, swap $L[p] \leftrightarrow L[p+2]$;

2. Hop left: $p < 7$ and $L[p+1] > 0$, swap $L[p] \leftrightarrow L[p+1]$;

3. Leap right: $p > 2$ and $L[p-2] < 0$, swap $L[p-2] \leftrightarrow L[p]$;

4. Hop right: $p > 1$ and $L[p-1] < 0$, swap $L[p-1] \leftrightarrow L[p]$.

Define four functions: $leap_l$, hop_l, $leap_r$, and hop_r, transition the status $L \mapsto L'$. If can't move, then returns L unchanged. We use a stack S to record the attempts. The stack starts from a singleton list, containing the initial status. List M records all solutions. We repeatedly pop the stack. If the state $L = e$, then we add this new solution to M; otherwise, we try 4 moves on top of L, and push the new status back.

$$solve\ [[-1, -1, -1, 0, 1, 1, 1]]\ [\ ] \tag{14.32}$$

Where:

$$\begin{aligned} solve\ [\ ]\ s\ &=\ s \\ solve\ (c{:}cs)\ s\ &=\ \begin{cases} L = e: & solve\ cs(reverse\ c : s), \text{where} : L = head\ c \\ \text{otherwise} : & solve\ ((map\ (:c)\ (moves\ L)) + cs)\ s \end{cases} \end{aligned} \tag{14.33}$$

function $moves$ tries 4 movements from L:

$$moves\ L = filter(\neq L)\ [leap_l\ L, hop_l\ L, leap_r\ L, hop_r\ L] \tag{14.34}$$

Below is the corresponding iterative implementation:

```
1: function SOLVE(s, e)
2:     S ← [[s]]
3:     M ← [ ]
4:     while S ≠ [ ] do
5:         s ← POP(S)
6:         if s[1] = e then
7:             ADD(M, REVERSE(s))
8:         else
9:             for each m in MOVES(s[1]) do
10:                 PUSH(S, m:s)
11:     return M
```

This method gives two symmetric solutions (15 steps for each). Below table lists one:

step	-1	-1	-1	0	1	1	1
1	-1	-1	0	-1	1	1	1
2	-1	-1	1	-1	0	1	1
3	-1	-1	1	-1	1	0	1
4	-1	-1	1	0	1	-1	1
5	-1	0	1	-1	1	-1	1
6	0	-1	1	-1	1	-1	1
7	1	-1	0	-1	1	-1	1
8	1	-1	1	-1	0	-1	1
9	1	-1	1	-1	1	-1	0
10	1	-1	1	-1	1	0	-1
11	1	-1	1	0	1	-1	-1
12	1	0	1	-1	1	-1	-1
13	1	1	0	-1	1	-1	-1
14	1	1	1	-1	0	-1	-1
15	1	1	1	0	-1	-1	-1

For 3 frogs on each side, it takes 15 steps. We extend 3 to n, and list the steps against n:

n frogs on each side	1	2	3	4	5	...
number of steps	3	8	15	24	35	...

The steps are all square numbers minus one: $(n+1)^2 - 1$. Let us prove it:

Proof. Compare the start and end states, every frog moves ahead $n+1$ stones. The $2n$ frogs in total move $2n(n+1)$ stones. Every frog on the left must meet every one from right once. The frog must leap over another one when meet. Because there are total n^2 meets, they cause all frogs move ahead $2n^2$ stones. The remaining moves are not leaps, but hops. There are total $2n(n+1) - 2n^2 = 2n$ hops. Sum up all n^2 leaps and $2n$ hops, the total steps are $n^2 + 2n = (n+1)^2 - 1$. □

The three puzzles share a common solution structure: start from some state, e.g. the entrance to the maze; the empty chess board; pegs of [-1, -1, -1, 0, 1, 1, 1]. Search the solution, try multiple options every step, e.g. 4 directions of up, down, left, and right in maze; 8 columns in each row; leap and hop, right and left. Although we don't know how far a decision leads to, we clearly know the final state, e.g. the exit of the maze; complete arranging 8 queens; all pegs are swapped.

We apply the same strategy: repeatedly try an option; record it with the newly obtained state; backtrack when stuck and try another option. We either find a solution or exhaust all options, hence know the problem is unsolvable. There are variants, like to stop when find a solution, or continue search all solutions. If build a tree rooted at the starting state, every branch represents an option, the search grows the tree deeper and deeper. We don't try alternatives at the same depth until fail and backtrack. Figure 14.21 shows the search order with arrows that go down then backtrack.

We call it deep first search (DFS), which is widely used in practice. Some programming environments, like Prolog, use DFS as the default evaluation model. For example define a maze with rules in Prolog:

```
c(a, b). c(a, e).
c(b, c). c(b, f).
```
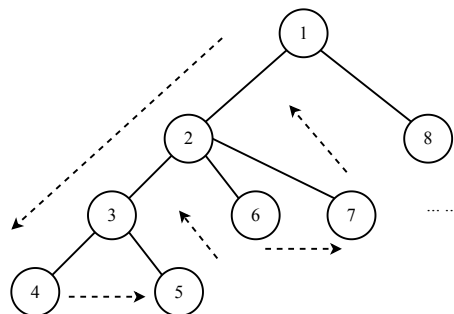
Figure 14.21: DFS search order.

```
c(e, d), c(e, f).
c(f, c).
c(g, d). c(g, h).
c(h, f).
```

Where predicate $c(X, Y)$ means X is connected with Y. This is a directed predicate. We can add a reverse symmetric rule $c(Y, X)$ or create a undirected predicate. Figure 14.22 shows a directed graph. Given two places X and Y, Prolog tells if they are connected with the following program:
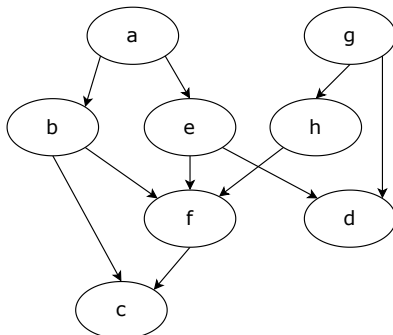
Figure 14.22: A directed graph.

```
go(X, X).
go(X, Y) :- c(X, Z), go(Z, Y)
```

This program says: a place X is connected with itself. Given two places X and Y, if X is connected with Z, and Z is connected with Y, then X is connected with Y. For multiple choices of Z, Prolog chooses one, then goes on searching. It only tries another Z if the recursive search completes (fails or succeeds) and backtracks. This is exactly the DFS. We can apply DFS when only need a solution, but don't care the number of steps. For example, we need a way out of the maze, although it may not be the shortest.

Exercise 14.6

14.6.1. Extend the pegs puzzle solution for n pegs on each side.

The wolf, goat, and cabbage puzzle

This traditional puzzle says that a farmer is going to cross the river with a wolf, a goat, and a bucket of cabbage. There is a small boat. Only the farmer can drive it. The boat

can only carry one thing a time. The wolf would kill the goat; the goat would bite the cabbage if they stay alone without the farmer. The puzzle asks to find the best (fast) solution to cross the river.

Since the wolf doesn't bite the cabbage, the farmer can safely carry the goal to the other side and go back. No matter carry the wolf or the cabbage next, the farmer need carry one back to avoid conflict. To find the best the solution, we parallel try all options and compare. Despite the direction, count back and forth 2 steps. We check all possible status after 1, 2, 3, ... steps, till the farmer and all things move to the other side at step n. This is the best solution.

But how to parallel try all options? Consider a lucky draw. People pick a ball from a box of colored balls. There is a black ball, and the rest are white. The one picks the black wins, or returns the white ball back to the box and waits for the next draw. We define the rule that nobody can try a second draw before all others pick. We line people in a queue. Every time the first person picks a ball, moves to the tail if doesn't win. The queue ensures the fairness.
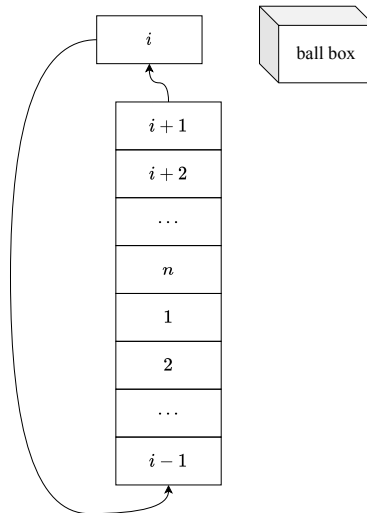
Figure 14.23: The i-th person de-queues, draws, then in-queues if doesn't win.

We apply the same method for the cross river puzzle. Let set A, B contain the things on each side. When start, $A = \{w, g, c, p\}$ includes the wolf, the goat, the cabbage, and the farmer; $B = \varnothing$. We move the farmer with or without another element between A and B. If a set doesn't contain the farmer, then it should not have conflict elements. The goal is to swap elements in A and B with the fewest steps. We initiate a queue Q with the start status: $A = \{w, g, c, p\}$, $B = \varnothing$. As far as Q isn't empty, we de-queue the head, try all options, then en-queue the new status back to the tail. We find the solution when the head becomes $A = \varnothing$, $B = \{w, g, c, p\}$. Figure 14.24 shows the search order. As all options at the same level are tried, we needn't backtrack.

We can represent the set with a binary number of four bits, each bit stands for an element, e.g., the wolf $w = 1$, the goat $g = 2$, the cabbage $c = 4$, and the farmer $p = 8$. 0 is the empty set, 15 is the full set. $3 = 1 + 2$, means the set $\{$wolf, goat$\}$. It's invalid because the wolf will kill the goat; $6 = 2 + 4$, is another conflict $\{$goat, cabbage$\}$. Every time, we move the highest bit (8), with or without another bit (4, 2, 1) from one number to the other. The options are:
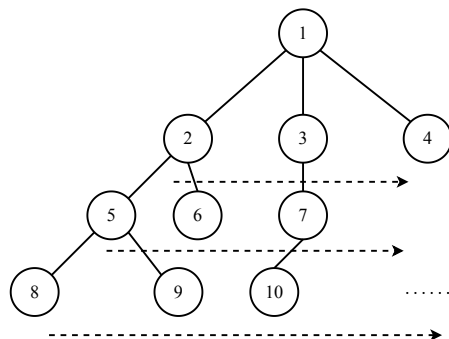
Figure 14.24: Start from 1, check all options 2, 3, 4 for the next step; then all option for the 3rd step, ...

$$mv\ A\ B = \begin{cases} B < 8: & [(A - 8 - i, B + 8 + i)|i \leftarrow [0, 1, 2, 4], i = 0 \text{ or } A\overline{\wedge}i \neq 0] \\ \text{otherwise}: & [(A + 8 + i, B - 8 - i)|i \leftarrow [0, 1, 2, 4], i = 0 \text{ or } B\overline{\wedge}i \neq 0] \end{cases}$$
$$(14.35)$$

Where $\overline{\wedge}$ is bitwise-and. We start searching from $Q = \{[(15, 0)]\}$, as: *solve Q*

$$solve\ \varnothing\ =\ \varnothing$$
$$solve\ Q\ =\ \begin{cases} A = 0: & reverse\ c, \text{where: } (A, B) = c, (c, Q') = pop\ Q \\ \text{otherwise}: & solve\ (pushAll\ (map\ (: c)\ (filter\ (valid\ c)\ (mv\ A\ B)))\ Q') \end{cases}$$
$$(14.36)$$

Where *valid c* checks if the move (A, B) is valid, neither 3 nor 6, and is new ($\notin c$):

$$A, B \neq 3 \text{ or } 6, (A, B) \notin c \tag{14.37}$$

Below is the iterative implementation:

```
1: function SOLVE
2:     S ← []
3:     Q ← {[(15, 0)]}
4:     while Q ≠ ∅ do
5:         C ← DEQ(Q)
6:         if C[1] = (0, 15) then
7:             ADD(S, REVERSE(C))
8:         else
9:             for each m in MOVES(C) do
10:                if VALID(m, C) then
11:                    ENQ(Q, m:C)
12:     return S
```

It outputs two best solutions:

Left	river	Right
wolf, goat, cabbage, farmer		
wolf, cabbage		goat, farmer
wolf, cabbage, farmer		goat
cabbage		wolf, goat, farmer
goat, cabbage, farmer		wolf
goat		wolf, cabbage, farmer
goat, farmer		wolf, cabbage
		wolf, goat, cabbage, farmer

Left	river	Right
wolf, goat, cabbage, farmer		
wolf, cabbage		goat, farmer
wolf, cabbage, farmer		goat
wolf		goat, cabbage, farmer
wolf, goat, farmer		cabbage
goat		wolf, cabbage, farmer
goat, farmer		wolf, cabbage
		wolf, goat, cabbage, farmer

Water jugs puzzle

Given two water jugs, 9 and 4 litres. How to get 6 litres of water from river? This puzzle has history back to ancient Greece. A story said the French mathematician Poisson solved this puzzle when he was a child. It also appears in Hollywood movie 'Die-Hard 3'. Pòlya uses this puzzle as an example of backwards induction[90].
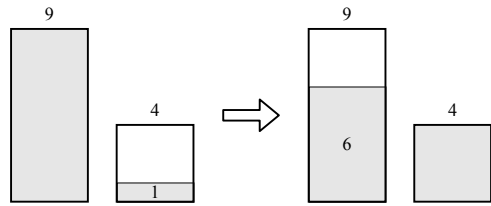
Figure 14.25: The last two steps.

After fill the 9 litres jug, then pour to the 4 litres jug twice, one obtains 1 litre of water, as shown in fig. 14.26. Backwards induction is a strategy, but not detailed algorithm. It can't directly answer how to get 2 litres of water from two jugs of 899 litres and 1147 litres for example.
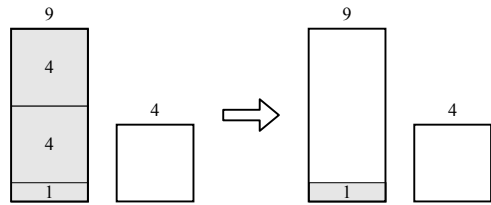
Figure 14.26: Fill the bigger jug, then pour to the smaller one twice.

Let the small jug be A, the big jug be B. There are 6 operations each time: (1) Fill jug A; (2) Fill jug B; (3) Empty jug A; (4) Empty jug B; (5) Pour from jug A to B; (6) Pour water from jug B to A. Below lists a series of operations (assume $a < b < 2a$).

A	B	operation
0	0	start
a	0	fill A
0	a	pour A to B
a	a	fill A
2a - b	b	pour A to B
2a - b	0	empty B
0	2a - b	pour A to B
a	2a - b	fill A
3a - 2b	b	pour A to B
...	...	...

Whatever operations, the water in each jug must be $xa + yb$, from some integers x and y, where a and b are jug volumes. From the number theory, we can get g litres of water if and only if g is some multiple of the greatest common divisor of a and b, i.e., $gcd(a,b)|g$. If $gcd(a,b) = 1$ (a and b are coprime), then we can get any nature number g litres of water. Although we know the existence of the solution, we don't know the detailed steps. We can solve the Diophantine equation $g = xa + yb$, then design the operations from x and y. Assume $x > 0, y < 0$, we fill jug A total x times, empty jug B total y times. For example, the small jug $a = 3$ litres, the big jug $b = 5$ litres, and the goal is to get $g = 4$ litres of water. Because $4 = 3 \times 3 - 5$, we design below operations:

A	B	operation
0	0	start
3	0	fill A
0	3	pour A to B
3	3	fill A
1	5	pour A to B
1	0	empty B
0	1	pour A to B
3	1	fill A
0	4	pour A to B

We fill jug A 3 times, empty jug B 1 time. We can apply the *Extended Euclid algorithm* in number theory to find x and y:

$$(d, x, y) = gcd_{ext}(a, b) \tag{14.38}$$

Where $d = gcd(a, b)$, $ax + by = d$. Assume $a < b$, the quotient q and remainder r satisfy $b = aq + r$. The common divisor d divides both a and b, hence d divides r too. Because $r < a$, we can scale down the problem to find $gcd(a, r)$:

$$(d, x', y') = gcd_{ext}(r, a) \tag{14.39}$$

Where $d = x'r + y'a$. Substitute $r = b - aq$ in:

$$
\begin{aligned}
d &= x'(b - aq) + y'a \\
&= (y' - x'q)a + x'b
\end{aligned}
\tag{14.40}
$$

Compare with $d = ax + by$, we have the following recursion:

$$\begin{cases} x & = y' - x'\dfrac{b}{a} \\ y & = x' \end{cases} \tag{14.41}$$

The edge case happens when $a = 0$: $gcd(0, b) = b = 0a + 1b$. Hence the extended Euclid algorithm can be defined as:

$$\begin{aligned} gcd_{ext}(0, b) & = & (b, 0, 1) \\ gcd_{ext}(a, b) & = & (d, y' - x'\dfrac{b}{a}, x') \end{aligned} \tag{14.42}$$

Where d, x', y' are defined in eq. (14.39). If $g = md$, then mx and my is a solution; if $x < 0$, for example: $gcd_{ext}(4, 9) = (1, -2, 1)$. Since $d = xa + yb$, we repeatedly add x by b, and decrease y by a till $x > 0$. Such solution may not be the best one. For example, to get 4 litres of water from two jugs of 3 and 5 liters, the extended Euclid algorithm gives 23 steps:

```
[(0,0),(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),
(0,4),(3,4),(2,5),(2,0),(0,2),(3,2),(0,5),(3,5),
(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),(0,4)]
```

While the best solution only need 6 steps:

```
[(0,0),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4)]
```

There are infinite many solutions for the Diophantine equation $g = xa + by$. The smaller $|x| + |y|$, the fewer steps. We can apply the same method as the 'cross river' puzzle. Try the 6 operations (fill A, fill B, pour A into B, pour B into A, empty A, and empty B) in parallel to find the best solution. We use a queue to arrange the attempts. The elements in the queue are series of pairs (p, q), where p and q are water in each jug, as operations from the beginning. The queue starts from $\{[(0,0)]\}$.

$$solve\ a\ b\ g = bfs\ \{[(0,0)]\} \tag{14.43}$$

As far as the queue isn't empty, we pop a sequence from the head. If the last pair of the sequence contains g litres, we find a solution. We reverse and output the sequence; otherwise, we try 6 operations atop the latest pair, filter out the duplicated ones and add back to the queue.

$$\begin{aligned} bfs\ \varnothing & = & [\,] \\ bfs\ Q & = & \begin{cases} p\ or\ q = g: & reverse\ s, \text{where: } (p, q) = head\ s, (s, Q') = pop\ Q \\ \text{otherwise}: & bfs\ (pushAll\ (map\ (:\ s)\ (try\ s))\ Q') \end{cases} \end{aligned} \tag{14.44}$$

$$try\ s = filter\ (\not\in s)\ [f\ (p, q)|f \leftarrow \{fl_A, fl_B, pr_A, pr_B, em_A, em_B\}] \tag{14.45}$$

Where:

$$\begin{cases} fl_A\ (p, q) = (a, q) \\ fl_B\ (p, q) = (p, b) \\ em_A\ (p, q) = (0, q) \\ em_B\ (p, q) = (p, 0) \\ pr_A\ (p, q) = (\max(0, p + q - b), \min(x + y, b)) \\ pr_B\ (p, q) = (\min(x + y, a), \max(0, x + y - a)) \end{cases} \tag{14.46}$$

This method returns the solution with the fewest steps. To avoid storing the complete operation sequence in the queue, we can use a global history list, and link every operation

back to its predecessor. As shown in fig. 14.27, the start state is (0, 0), only 'fill A' and
'fill B' are applicable. We next try 'fill B' atop (3, 0), record the new state (3, 5). If
apply 'empty A' to (3, 0), we'll go back to the starting point (0, 0). We skip it (shaded
state). We add a 'parent' reference to each node in fig. 14.27, and backtrack along it to
the beginning.
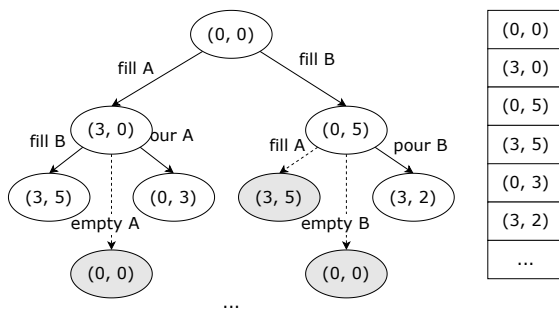
Figure 14.27: Store all states with a global list.

```
 1: function SOLVE(a, b, g)
 2:     Q ← {(0, 0, NIL)}                                              ▷ Queue
 3:     V ← {(0, 0, NIL)}                                              ▷ Visited set
 4:     while Q ≠ ∅ do
 5:         s ← POP(Q)
 6:         if p(s) = g or q(s) = g then
 7:             return BACK-TRACK(s)
 8:         else
 9:             for each c in EXPAND(s, a, b) do
10:                 if c ≠ s and c ∉ V then
11:                     PUSH(Q, c)
12:                     ADD(V, c)
13:     return NIL

14: function EXPAND(s, a, b)
15:     p ← p(s), q ← q(s)
16:     return [(a, q, s), (p, b, s), (0, q, s), (p, 0, s), (max(0, p+q−b), min(p+q, b), s), (min(p+
    q, a), max(0, p + q − a), s)]

17: function BACK-TRACK(s)
18:     r ← [ ]
19:     while s ≠ NIL do
20:         (p, q, s') = s
21:         r ← (p, q):r
22:         s ← s'
23:     return r
```

Exercise 14.7

14.7.1. Improve the extended Euclid algorithm, find the x and y that minimize $|x| + |y|$
   for the optimal solution for the two jugs puzzle.

Kloski

Kloski is a block sliding puzzle, as shown in fig. 14.28. There are 10 blocks of 3 sizes: 4 pieces of 1×1; 4 pieces of 1×2, 1 piece of 2×1, 1 piece of 2×2. The goal is to slide the big block to the bottom slot. Figure 14.29 shows variants of this puzzle in Japan.

(a) Initial layout   (b) Layout after several movements
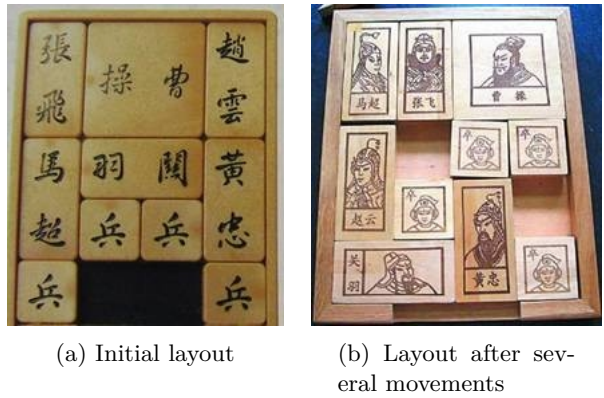
Figure 14.28: 'Huarong Escape', the traditional Chinese Kloski puzzle.

Figure 14.29: 'Daughter in the box', the Japanese Kloski puzzle.

We define the board as a 5×4 matrix, the row and column start from 0. Label the pieces from 1 to 10. 0 means empty cell. The matrix M gives the initial layout. The cells with value i is occupied by piece i. We use a map L to represent the layout, where $L[i]$ is the set of cells occupied by piece i. For example, $L[4] = \{(2, 1), (2, 2)\}$ means the 4th piece occupies cells $(2, 1)$ and $(2, 2)$. Label all 20 cells from 0 to 19, we can convert a pair of row, col to label by: $c = 4y + x$. The 4th piece occupies cells $L[4] = \{9, 10\}$.

$$
M = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix} \quad L = \left\{ \begin{array}{l} 1 \mapsto \{0, 4\}, 2 \mapsto \{3, 7\}, 3 \mapsto \{8, 12\}, \\ 4 \mapsto \{9, 10\}, 5 \mapsto \{11, 15\}, \\ 6 \mapsto \{16\}, 7 \mapsto \{13\}, 8 \mapsto \{14\}, \\ 9 \mapsto \{19\}, 10 \mapsto \{1, 2, 5, 6\} \end{array} \right\}
$$

Define map $\varphi(M) \mapsto L$ and its reverse $\varphi^{-1}(L) \mapsto M$ to convert between board and layout:

1: **function** $\varphi(M)$
2:      $L \leftarrow \{\}$
3:      **for** $y \leftarrow 0 \sim 4$ **do**
4:          **for** $x \leftarrow 0 \sim 3$ **do**

5:              $k \leftarrow M[y][x]$
6:              $L[k] \leftarrow \text{ADD}(L[k], 4y + x)$
7:       **return** L

8: **function** $\varphi^{-1}(L)$
9:       $M \leftarrow [[0] \times 4] \times 5$
10:       **for** each $(k \mapsto S)$ in L **do**
11:            **for** each c in S **do**
12:                 $x \leftarrow c \bmod 4, y \leftarrow \lfloor c/4 \rfloor$
13:                 $M[y][x] \leftarrow k$
14:       **return** M

We try all the 10 blocks in 4 directions: up, down, left, and right. For board matrix, the movement means: $(\Delta y, \Delta x) = (0, \pm 1), (\pm 1, 0)$; for layout of cell labels, it means: $d = \pm 1, \pm 4$. For example, move piece $L[i] = \{c_1, c_2\}$ to left, it becomes: $\{c_1 - 1, c_2 - 1\}$. We need avoid invalid movement in two edge cases: $d = 1, c \bmod 4 = 3$ and $d = -1, c \bmod 4 = 0$. They are invalid because the piece jumps from one side to the other. Consider the two free cells, there are at most 8 movements. For example, the first step only have 4 options: move piece 6 right, move piece 7 or 8 down, move piece 9 left. Figure 14.30 shows how to verify the movement is valid.
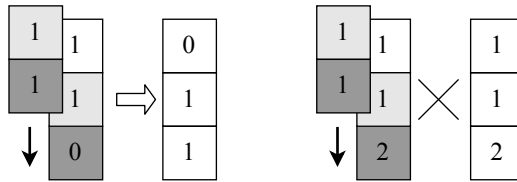
Figure 14.30: Left: two cells of 1 can move; Right: the lower cell of 1 conflicts with the cell of 2.

For the movement of piece k, it is valid if the target cells have value of 0 or k:

$$
\begin{aligned}
&valid\ L[k]\ d: \\
&\forall c \in L[k] \Rightarrow \quad y = \lfloor c/4 \rfloor + \lfloor d/4 \rfloor, x = (c \bmod 4) + (d \bmod 4), \qquad (14.47)\\
&\qquad\qquad\qquad (0,0) \leq (y, x) \leq (4, 3), M[y][x] \in \{k, 0\}
\end{aligned}
$$

We may return to some layout after a series of slides. It's insufficient to only avoid duplicated matrix. Although $M_1 \neq M_2$, they are essentially the same layout.

$$
M_1 = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix} \quad
M_2 = \begin{bmatrix} 2 & 10 & 10 & 1 \\ 2 & 10 & 10 & 1 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 6 & 5 \\ 8 & 0 & 0 & 9 \end{bmatrix}
$$

We need avoid duplicated layout. Treat all pieces of the same size same, we define normalized layout as: $\|L\| = \{p | (k \mapsto p) \in L\}$, the set of all cell labels in L. Both matrix above have the same normalized layout as $\{\{1, 2, 5, 6\}, \{0, 4\}, \{3, 7\}, \{8, 12\}, \{9, 10\}, \{11, 15\}, \{16\}, \{13\}, \{14\}, \{19\}\}$. We also need avoid mirrored layout, for example:

$$
M_1 = \begin{bmatrix} 10 & 10 & 1 & 2 \\ 10 & 10 & 1 & 2 \\ 3 & 5 & 4 & 4 \\ 3 & 5 & 8 & 9 \\ 6 & 7 & 0 & 0 \end{bmatrix} \quad
M_2 = \begin{bmatrix} 3 & 1 & 10 & 10 \\ 3 & 1 & 10 & 10 \\ 4 & 4 & 2 & 5 \\ 7 & 6 & 2 & 5 \\ 0 & 0 & 9 & 8 \end{bmatrix}
$$

They have symmetric normalized layout. Define the mirror function:

$$mirror(\|L\|) = \{\{f(c)|c \in s\}|s \in \|L\|\} \tag{14.48}$$

Where $f(c) = 4y' + x'$, $y' = \lfloor c/4 \rfloor$, $x' = 3 - (c \bmod 4)$. We use a queue to arrange the search. The element in the queue has two parts: a series of movements, and the resulted layout. The movement is a pair (k, d), means move piece k by d (some value of ± 1, ± 4). Initialize the queue $Q = \{(s, [\ ])\}$, where s is the starting layout. As far as the queue isn't empty $Q \neq \varnothing$, we get its head, examine whether the big block (piece 10) arrives at $t = \{13, 14, 17, 18\}$, i.e., $L[10] = t$. Terminate if yes; otherwise, we try up, down, left, right for every piece, add every valid (k, d), that leads to distinct layout to the queue. We use a set H to record all visited normalized layouts to avoid repetition.

$$
\begin{aligned}
solve\ \varnothing\ H &= [\ ] \\
solve\ Q\ H &= \begin{cases} L[10] = t : & reverse\ ms, \text{where: } ((L, ms), Q') = pop\ Q \\ \text{otherwise} : & solve\ (pushAll\ cs\ Q')\ H' \end{cases}
\end{aligned} \tag{14.49}
$$

Where $cs = [(move\ L\ e, e{:}ms)|e \leftarrow expand\ L]$ are the new movements expanded.

$$
\begin{aligned}
expand\ L = \{(k, d)| \quad &k \leftarrow [1, 2, ..., 10], d \leftarrow [\pm 1, \pm 4], \\ &valid\ k\ d, unique\ k\ d\}
\end{aligned} \tag{14.50}
$$

Function $move$ slides piece $L[k]$ by d to: $move\ L\ (k, d) = map\ (+d)\ L[k]$. $unique$ checks if the normalized layout $\|L'\| \notin H$ and its $mirror(\|L'\|) \notin H$. Add them to H' if new. Below are the iterative implementation. The solution has 116 steps (1 cell a step). The last 3 are:

```
 1: function SOLVE(s, e)
 2:     H ← {‖s‖}
 3:     Q ← {(s, ∅)}
 4:     while Q ≠ ∅ do
 5:         (L, p) ← POP(Q)
 6:         if L[10] = e then
 7:             return (L, p)
 8:         else
 9:             for each L' in EXPAND(L, H) do
10:                 PUSH(Q, (L', L))
11:                 ADD(H, ‖L'‖)
12:     return ∅
```

```
['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['A', 'A', '6', '0']
['A', 'A', '0', '8']

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['A', 'A', '0', '6']
['A', 'A', '0', '8']

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['0', 'A', 'A', '6']
['0', 'A', 'A', '8']
```

The cross river puzzle, the water jugs puzzle, and the Kloski puzzle share another common solution structure. Similar to the DFS, they have start and end states. For example, the cross river puzzle starts with all things on one side, the other side is empty; it ends with all things on the other side. The water jugs puzzle starts with two empty jugs; it ends with either jug has g litres of water. The Klotski puzzle starts with a given layout, it ends with some layout that the big block arrives at the bottom slot. Every puzzle has a set of rules, transfers from a state to another. We 'parallel' try all options. We don't search further until complete trying all options of the same step. This search strategy ensures we find the solution with the fewest steps before others. Because we expand horizontally (as in fig. 14.31), it's called breadth-first search (BFS).

(a) DFS   (b) BFS
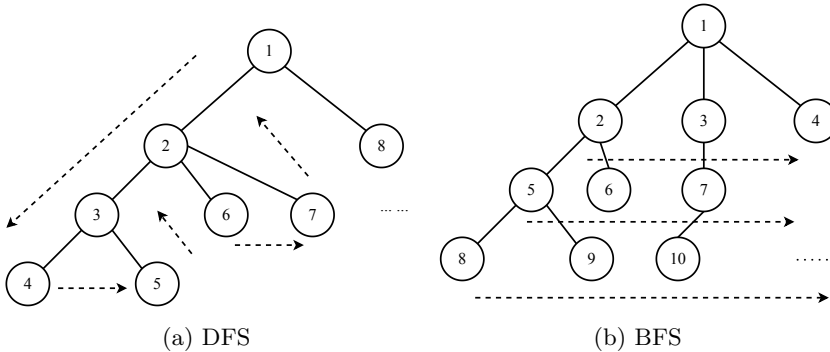
Figure 14.31: DFS and BFS.

Because we can't really search in parallel, we realize BFS with a queue. Repeatedly de-queue the candidate with fewer steps from head, and en-queue new candidate with more steps to tail. BFS provides a simple method to search the solution with the fewest steps. However, it can't directly search for generic optimal solution. Consider the directed graph in fig. 14.32, the length of each section varies. We can't use BFS to find the shortest path between two nodes. For example, the shortest path from a to c is not the one with the fewest steps: $a \rightarrow b \rightarrow c$ (the total length of 22), but the path with more steps $a \rightarrow e \rightarrow f \rightarrow c$ (the total length of 20).
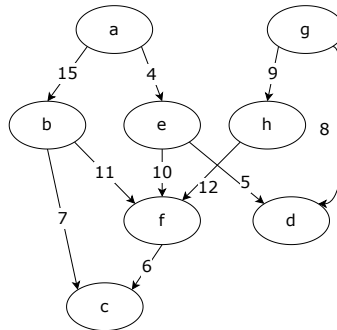
Figure 14.32: A weighted directed graph.

Exercise 14.8

14.8.1. Traverse a binary tree level by level, i.e., first visit the root, then the nodes at level 1, then level 2, ...

14.8.2. John Conway[9] gives a slide tile puzzle. Figure 14.33 is a simplified example. There are 8 cells, 7 are occupied. Label the pieces from 1 to 7. Each piece can slide to the connected free cell. (two cells are connected if there is a line between them.) How to reverse the pieces from 1, 2, 3, 4, 5, 6, 7 to 7, 6, 5, 4, 3, 2, 1 by sliding? Write a program to solve this puzzle.
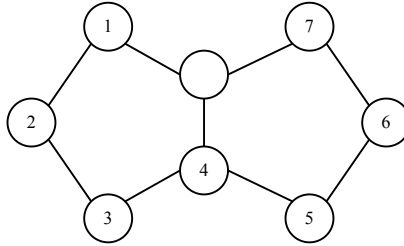
Figure 14.33: Conway slide puzzle

14.6.2 Greedy algorithm

People need find the 'best' solution to minimize time, space, cost, energy, and etc. It's not easy to find the optimal solution within limited resource. Many problems don't have solution in polynomial time, however, there exist simple solutions for a small portion of the special problems.

Huffman coding

Huffman coding encodes information with the shortest length. The ASCII code needs 7 bits to encode characters, digits, and symbols. It can represent $2^7 = 128$ symbols. We need at least $\log_2 n$ bits of 0/1 to distinguish n symbols. Below table encodes upper case English letters, maps A to Z from 0 to 25, each with 5 bits. Zero is padded as 00000 but not 0. Such scheme is called fixed-length coding. For example, it encodes 'INTERNATIONAL' to a binary number of 65 bits:

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

```
00010101101100100100100011011000000110010001001110101100000011010
```

---

[9]John Conway (1937 - 2020), British mathematician.

Another scheme is the variable-length coding. Encode A as single bit 0, encode C as 10 of two bits, encode Z as 11001 of 5 bits. Although the code length is shorter, there is ambiguity when decode. For example, the binary number 1101 may stand for 1 followed with 101 (decoded as 'BF') or 110 followed with 1 (decoded as 'GB'), or 1101 (decoded as N). The Morse code is variable-length. It encodes the most used letter 'E' as '.', encodes 'Z' as '- -..'. Particularly, it uses a special pause separator to indicate the termination of a code, hence, eliminates the ambiguity. Below code table is ambiguity free, it encodes 'INTERNATIONAL' with only 38 bits.

char	code	char	code
A	110	E	1110
I	101	L	1111
N	01	O	000
R	001	T	100

```
10101100111000101110100101000011101111
```

The reason why it's ambiguity free is because there is no code is the prefix of the other. Such code is called *prefix-code*. (but not the 'non-prefix code'.) Since the prefix-code needn't separator, we can further shorten the code length. Given a text, can we find a prefix-code scheme, that produces the shortest code? In 1951, Robert M. Fano told the class that who could solve this problem needn't take the final exam. Huffman was still a student in MIT[91]. He almost gave up and started preparing the final exam when found the answer. Huffman created the coding table according to the frequency of the symbol appeared in the text. The more used one is assigned with the shorter code. He could process the text, and calculate the occurrence for each symbol, then define the weight as the frequency. Huffman uses a binary tree to generate the prefix-code. The symbols are stored in the leaf nodes. Traverse from the root to generate the code, add 0 when go left, 1 when go right, as shown in fig. 14.34. For example, starting from the root, go left, then right, we arrive at 'N'. Therefore, 'N' is encoded as '01'; While the paths of 'A' is right → right → left, encoded as '110'.
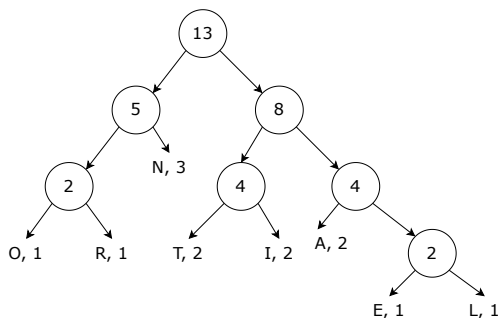
Figure 14.34: Huffman tree

We can use the tree to decode as well. Scan the binary bits, go left for 0, and right for 1. When arrive at a leaf, we decode the symbol from it. Then restart from the root to continue scan. Huffman builds the tree in bottom-up way. When start, wrap all symbols in leaves. Every time, pick two nodes with the minimum weights, merge them to a branch node of weight w. where $w = w_1 + w_2$ is the sum of the two weights. Repeatedly pick and merge the two smallest weighted trees till we get the final tree, as shown in fig. 14.35.
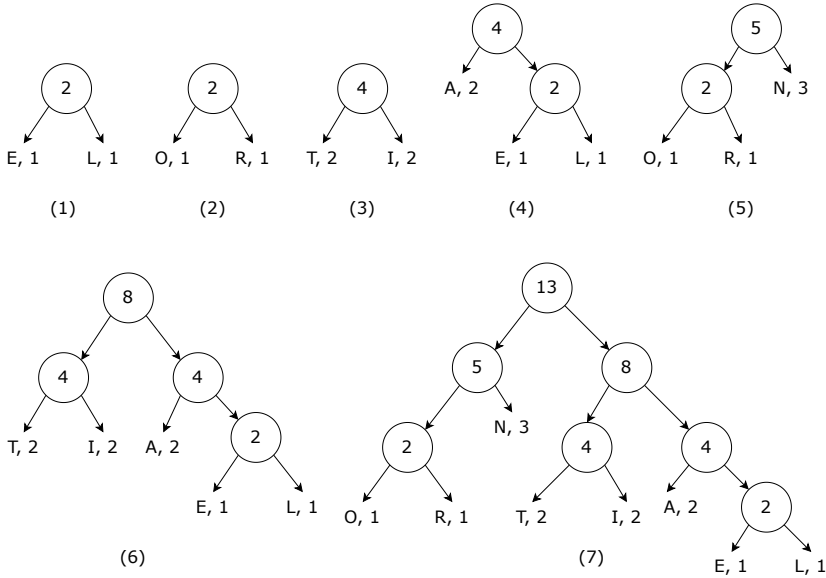
Figure 14.35: Build a Huffman tree.

We reuse the binary tree definition for Huffman tree. We augment the weight and only hold the symbol in leaf node. Let the branch node be (w, l, r), where w is the weight, l and r are the left and right sub-trees. Let the leaf be (w, c), where c is the symbol. When merge trees, we sum the weight: $merge\ a\ b\ = (weight\ a + weight\ b, a, b)$, where:

$$\begin{aligned} weight\ (w, a) &= w \\ weight\ (w, l, r) &= w \end{aligned} \tag{14.51}$$

Below function repeatedly pick and merge the minimum weighted trees:

$$\begin{aligned} build\ [t] &= t \\ build\ ts &= build\ (merge\ t_1\ t_2)\ ts', \text{where: } (t_1, t_2, ts') = extract\ ts \end{aligned} \tag{14.52}$$

Function $extract$ picks two trees with the minimal weight. Define $t_1 < t_2$ if $weight\ t_1 < weight\ t_2$.

$$extract(t_1{:}t_2{:}ts) = foldr\ min_2\ (\min t_1\ t_2, \max t_1\ t_2, [\ ])\ ts \tag{14.53}$$

Where:

$$min_2\ t\ (t_1, t_2, ts) = \begin{cases} t < t_2: & (\min\ t\ t_1, \max\ t\ t_1, t_2{:}ts) \\ \text{otherwise}: & (t_1, t_2, t{:}ts) \end{cases} \tag{14.54}$$

To iterate building Huffman tree, we store n sub-trees in array A. Scan A from right to left. If the weight of $A[i]$ is less than $A[n-1]$ or $A[n]$, we swap $A[i]$ and $\text{MAX}(A[n-1], A[n])$. Merge $A[n]$ and $A[n-1]$ after scan, and shrink the array by one. Repeat this to build the Huffman tree:

```
1: function HUFFMAN(A)
2:     while |A| > 1 do
3:         n ← |A|
4:         for i ← n − 2 down to 1 do
5:             T ← MAX(A[n], A[n − 1])
```

6:              **if** $A[i] < T$ **then**
7:                  EXCHANGE $A[i] \leftrightarrow T$
8:          $A[n-1] \leftarrow$ MERGE$(A[n], A[n-1])$
9:          DROP$(A[n])$
10:     **return** $A[1]$

We can build the code table from the Huffman tree. Let $p = [\ ]$. Traverse from the root, update $p \leftarrow 0{:}p$ when go left; $p \leftarrow 1{:}p$ when go right. When arrive at leaf of symbol c, record the map $c \mapsto reverse\ p$ to the code table. Define (Curried form): $code\ =\ traverse\ [\ ]$, where:

$$\begin{aligned} traverse\ p\ (w, c) &= [c \mapsto reverse\ p] \\ traverse\ p\ (w, l, r) &= traverse\ (0{:}p)\ l \mathbin{+\!\!+} traverse\ (1{:}p)\ l \end{aligned} \qquad (14.55)$$

When encoding, we scan the text w while looking up the code table $dict$ to generate binary bits:

$$encode\ dict\ w = concatMap\ (c \mapsto dict[c])\ w, \text{where: } dict = code\ T \qquad (14.56)$$

Conversely, when decoding, we scan the binary bits bs while looking up the tree. Start from the root, go left for 0, right for 1; output symbol c when arrive at leaf; then reset to the root to continue. $decode\ T\ bs = lookup\ T\ bs$, where:

$$\begin{aligned} lookup\ (w, c)\ [\ ] &= [c] \\ lookup\ (w, c)\ bs &= c : lookup\ T\ bs \\ lookup\ (w, l, r)\ (b{:}bs) &= lookup\ (\text{if } b = 0 \text{ then } l \text{ else } r)\ bs \end{aligned} \qquad (14.57)$$

Huffman tree building reflects a special strategy: always pick and merge the two trees with the minimal weight every time. The series of *local* optimal options generate a *global* optimal prefix-code. Local optimal sub-solutions do not necessarily lead to global optimal solution usually. Huffman coding is an exception. We call the strategy that always choose the local optimal option as the *greedy* strategy. Greedy method simplifies and works for many problems. However, it's not easy to tell whether the greedy method generates the global optimal solution. The generic formal proof is still an active research area[4].

Exercise 14.9

14.9.1. Implement the imperative Huffman code table algorithm.

Change making problem

How to change money with coins as few as possible? Suppose there are 5 coin values: 1, 5, 25, 50, and 100 as a set $C = \{1, 5, 25, 50, 100\}$. To change money of value x, we can apply the greedy method, always choose the most valued coin:

$$\begin{aligned} change\ 0 &= [\ ] \\ change\ x &= c_m : change\ (x - c_m), \text{where} : c_m = \max\ \{c \in C, c \le x\} \end{aligned} \qquad (14.58)$$

For example, to change money of value 142, this function outputs a coin list: $[100, 25, 5, 5, 5, 1, 1]$. We can convert it to $[(100, 1), (25, 1), (5, 3), (1, 2)]$, meaning 1 coin of 100, 1 coin of 25, 3 coins of 5, 2 coins of 1. For the coin system of C, the greedy method can find the optimal solution. Actually, it is applicable for the most coin systems in the world. There are exceptions, for example: $C = \{1, 3, 4\}$. To change money $x = 6$, the optimal solution is 2 coins of 3, however, the greedy method gives $6 = 4 + 1 + 1$, total 3 coins.

Although it's not the optimal solution, the greedy method often gives a simplified sub-optimal implementation. The result is often good enough in practice. For example, the word-wrap is a common functionality in editors. If the length of the text T exceeds the page width w, we need break it into lines. Let the space between words be s, below greedy implementation gives the approximate optimal solution: put as many words as possible in a line.

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:          $L \leftarrow L - |w| - s$

Exercise 14.10

14.10.1. Use heap to build the Huffman tree: take two trees from the top, merge then add back to the heap.

14.10.2. If we sort the symbols by their weight as A, there is a linear time algorithm to build the Huffman tree: use a tree Q to store the merge result, repeat take the minimal weighted tree from Q and the head of A, merge then add to the queue. After process all trees in A, there is a single tree in the Q, which is the Huffman tree. Implement this algorithm.

14.10.3. Given a Huffman tree T, implement the decode algorithm with fold left.

14.6.3 Dynamic programming

Consider how to find the best solution C_m (the list of coins) to change money x for any coin system. We can partition C_m into two groups: C_1 and C_2, with values x_1 and x_2 respectively, i.e., $C_m = C_1 + C_2$ and $x = x_1 + x_2$. We'll prove that C_1 is the optimal solution to change x_1, and C_2 is the optimal solution to change x_2.

Proof. For x_1, suppose there exists another solution C_1' with less coins than C_1. Then the solution $C_1' + C_2$ changes x with less coins than C_m. This conflicts with the fact that C_m is the optimal solution to change x. We can prove C_2 is the optimal solution to change X_2 in the same way. □

However, the converse is not true. for any integer $y < x$, divide the original problem to two sub-problems: change y and $x - y$. It's not necessary the overall optimal solution when combine the two optimal sub-solutions. For example, use 3 values $C = \{1, 2, 4\}$ to change $x = 6$. The optimal solution needs two coins: $2 + 4$. As $6 = 3 + 3$, divide it to two same sub-problems of changing 3. Each sub-problem has the optimal solution: $3 = 1 + 2$, however, the combination $(1 + 2) + (1 + 2)$ needs 4 coins. If an optimal problem can be divided into several optimal sub-problems, we call it has optimal substructure. The change money problem has optimal substructure, but we need divide based on the coin value, but not an arbitrary integer.

$$
\begin{aligned}
change\ 0 &= [\ ] \\
change\ x &= \min\ [c : change\ (x - c)|c \in C, c < x]
\end{aligned}
\tag{14.59}
$$

Where min picks the shortest list. However, this definition is impractical. There are too many duplicated computations. For example $C = \{1, 2, 25, 50, 100\}$, when computes $change(142)$, it needs further compute $change(141)$, $change(137)$, $change(117)$,

change(92), and *change*(42). For *change*(141), minus it by 1, 2, 25, 50, 100, we go back to 137, 117, 92, 42. The search domain expands at 5^n. Remind the idea to generate Fibonacci numbers, we can use a table T to records the optimal solution to the sub-problems. T starts from empty. When change money y, we lookup $T[y]$ first. If $T[y] = \varnothing$, then recursively compute the sub-problem, and save the sub-solution in $T[y]$.

```
1: T ← [[ ], ∅, ∅, ...]                                            ▷ T[0] = [ ]
2: function CHANGE(x)
3:     if x > 0 and T[x] = ∅ then
4:         for each c in C and c ≤ x do
5:             Cm ← c : CHANGE(x − c)
6:             if T[x] = ∅ or |Cm| < |T[x]| then
7:                 T[x] ← Cm
8:     return T[x]
```

We can bottom-up generate the optimal solutions for each sub-problem. From $T[0] = [\ ]$, generate $T[1] = [1]$, $T[2] = [1,1]$, $T[3] = [1,1,1]$, $T[4] = [1,1,1,1]$, as shown in table 14.1(a). There are two options for $T[5]$: 5 coins of 1, or a coin of 5. The latter need fewer coins. We update the optimal table to table 14.1(b), $T[5] = [5]$. Next change money $x = 6$. Both 1 and 5 are less than 6, there are two options: (1) $1 + T[5]$ gives $[1,5]$; (2) $5 + T[1]$ gives $[5,1]$. They are equivalent, we pick either $T[6] = [1,5]$. For $T[i]$, where $i \leq x$, we check every coin value $c \leq i$. Lookup $T[i-c]$ for the sub-problem, then plus c to get a new solution. We choose the fewest one as $T[i]$.

x	0	1	2	3	4
optimal solution	[ ]	[1]	[1,1]	[1,1,1]	[1,1,1,1]

(a) Optimal solution for $x \leq 4$

x	0	1	2	3	4	5
optimal solution	[ ]	[1]	[1,1]	[1,1,1]	[1,1,1,1]	[5]

(b) Optimal solution for $x \leq 5$

Table 14.1: Optimal solution table

```
1: function CHANGE(x)
2:     T ← [[ ], ∅, ...]
3:     for i ← 1 to x do
4:         for each c in C and c ≤ i do
5:             if T[i] = ∅ or 1 + |T[i − c]| < |T[i]| then
6:                 T[i] ← c : T[i − c]
7:     return T[x]
```

There are many duplicated contents in the below optimal solution table. A solution contains the sub-solutions. We can only record the incremental part: the coin c chosen for $T[i]$ and the number n of coins, i.e., $T[i] = (n, c)$. To generate the list of coins for x, we lookup $T[x]$ to get c, then lookup $T[x-c]$ to get c', ... repeat this to $T[0]$.

value	6	7	8	9	10	...
optimal solution	[1,5]	[1,1,5]	[1,1,1,5]	[1,1,1,1,5]	[5,5]	...

```
1: function CHANGE(x)
2:     T ← [(0, ∅), (∞, ∅), (∞, ∅), ...]
```

3:      **for** $i \leftarrow 1$ to x **do**
4:          **for** each c in C and $c \leq i$ **do**
5:              $(n, \_) \leftarrow T[i - c], (m, \_) \leftarrow T[i]$
6:              **if** $1 + n < m$ **then**
7:                  $T[i] \leftarrow (1 + n, c)$
8:      $s \leftarrow [\,]$
9:      **while** $x > 0$ **do**
10:          $(\_, c) \leftarrow T[x]$
11:          $s \leftarrow c : s$
12:          $x \leftarrow x - c$
13:      **return** s

We can build the optimal solution table T with left fold: *foldl fill* $[(0, 0)]$ $[1, 2, ...]$, where:

$$fill\ T\ x = T \rhd \min\ \{(fst\ T[x - c], c) | c \in C, c \leq x\} \qquad (14.60)$$

Where $s \rhd a$ append a to the right of s (see finger tree in section 12.5). Then rebuild the optimal solution backwards from T:

$$\begin{aligned} change\ 0\ T &= [\,] \\ change\ x\ T &= c : change\ (x - c)\ T, \text{where: } c = snd\ T[x] \end{aligned} \qquad (14.61)$$

For $x = n$, we loop n times, check at most $k = |C|$ coins. The performance is bound to $\Theta(nk)$[10], and need $O(n)$ space to persist T both in the top-down and bottom-up implementations. The solution to the sub-problem is used many times to compute the global optimal solution. We call it overlapping sub-problems. Richard Bellman developed dynamic programming in 1940s. It has two properties.

1. Optimal sub-structure. The problem can be broken down into small problems. The optimal solution can be constructed from the solutions of these sub-problems;

2. Overlapping sub-problems. The solution of the sub-problem is reused multiple times to find the overall solution.

Longest common sub-sequence

Different with sub-string, the sub-sequence needn't be consecutive. For example: the longest common sub-string of 'Mississippi' and 'Missunderstanding' is 'Miss', while the longest common sub-sequence is 'Misssi' as shown in fig. 14.36. If rotate the figure by 90°, it turns to be a 'diff' result between them. This is a common function in version control tools. The longest common sub-sequence of x and y are defined as below:
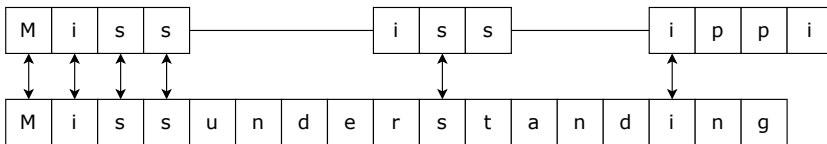
Figure 14.36: The longest common sub-sequence.

$$\begin{aligned} LCS([\,], ys) &= [\,] \\ LCS(xs, [\,]) &= [\,] \\ LCS(x : xs, y : ys) &= \begin{cases} x = y : & x : LCS(xs, ys) \\ \text{otherwise} : & \max\ LCS(x : xs, ys)\ LCS(xs, y : ys) \end{cases} \end{aligned} \qquad (14.62)$$

_____
[10]upper bound

Where max picks the longer sequence. There is optimal sub-structure in the definition of LCS. It can be broken down into sub-problems. The sequence length reduces at least by 1 every time. There are overlapping sub-problems. The longest common sub-sequence of the sub-strings are reused multiple times to find the global optimal solution. We use a 2D table T to record the optimal solution of the sub-problems. The row and column represent xs and ys respectively. Index the sequence from 0. Row 0, column 0 represents the empty sequence. $T[i][j]$ is the length of $LCS(xs[0..j], ys[0..i])$. We finally build the longest common sub-sequences from T. Because $LCS([\,], ys) = LCS(xs, [\,]) = [\,]$, row 0 and column 0 are all 0s. Consider 'antenna' and 'banana' for example, we fill row 1 from $T[1][1]$. 'b' is different from any one in 'antenna', hence row 1 are all 0s. For $T[2][1]$, the row and column are corresponding to 'a', $T[2][1] = T[1][0] + 1 = 1$, i.e., $LCS(a, ba) = a$. Next move to $T[2][2]$, 'a' $\neq$ 'n', we choose the greater one between the above ($LCS(an, b)$) and the left ($LCS(a, ba)$) as $T[2][2]$, which equals to 1, i.e., $LCS(ba, an) = a$. In this way, we step by step fill the table out. The rule is: for $T[i][j]$, if $xs[i-1] = ys[i-1]$, then $T[i][j] = T[i-1][j-1] + 1$; otherwise, pick the greater one from above $T[i-1][j]$ and the left $T[i][j-1]$.

		0	1	2	3	4	5	6	7
		[ ]	a	n	t	e	n	n	a
0	[ ]	0	0	0	0	0	0	0	0
1	b	0	0	0	0	0	0	0	0
2	a	0	1	1	1	1	1	1	1
3	n	0	1	2	2	2	2	2	2
4	a	0	1	2	2	2	2	2	3
5	n	0	1	2	2	2	3	3	3
6	a	0	1	2	2	2	3	3	4

1: **function** LCS(xs, ys)
2:     $m \leftarrow |xs|, n \leftarrow |ys|$
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** $xs[i] = ys[j]$ **then**
7:                 $T[i+1][j+1] \leftarrow T[i][j] + 1$
8:             **else**
9:                 $T[i+1][j+1] \leftarrow$ Max($T[i][j+1], T[i+1][j]$)
10:    **return** Fetch(T, xs, ys)   ▷ build the LCS

We next build the longest common sub-sequence from T. Start from the bottom-right, if $xs[m] = ys[n]$, then $xs[m]$ is the tail of the LCS, we next compare $xs[m-1]$ and $ys[n-1]$; otherwise, we pick the greater one from $T[m-1][n]$ and $T[m][n-1]$ and go on.

1: **function** Fetch(T, xs, ys)
2:     $m \leftarrow |xs|, n \leftarrow |ys|$
3:     $r \leftarrow [\,]$
4:     **while** $m > 0$ and $n > 0$ **do**
5:         **if** $xs[m-1] = ys[n-1]$ **then**
6:             $r \leftarrow xs[m-1] : r$
7:             $m \leftarrow m - 1$
8:             $n \leftarrow n - 1$
9:         **else if** $T[m-1][n] > T[m][n-1]$ **then**

10:              $m \leftarrow m - 1$
11:          **else**
12:              $n \leftarrow n - 1$
13:      **return** r

Exercise 14.11

14.11.1. For the longest common sub-sequence, build the optimal solution table with fold.

Subset sum

Given a set X of integers, how to find all the subsets $S \subseteq X$, that the sum of elements in S is s, i.e., $\sum S = \sum_{i \in S} i = s$? For example, $X = \{11, 64, -82, -68, 86, 55, -88, -21, 51\}$, there are three subsets with sum $s = 0$: $S = \varnothing$, $\{64, -82, 55, -88, 51\}$, $\{64, -82, -68, 86\}$. We need exhaust 2^n subset sums, where $n = |X|$, the performance is $O(n2^n)$.

$$
\begin{aligned}
sets\ s\ \varnothing &= [\varnothing] \\
sets\ s\ (x{:}xs) &= \begin{cases} s = x : & \{x\} : sets\ s\ xs \\ \text{otherwise} : & (sets\ s\ xs) + [x{:}S | S \in sets\ (s - x)\ xs] \end{cases}
\end{aligned}
\tag{14.63}
$$

There is sub-structure and overlapping sub-problems in above exhaustive search definition, we can apply dynamic programming method. We bottom-up build solution table T, and generate the final subset. First consider the existence of some subset S, satisfying $\sum S = s$. We scan the elements to determine the bottom/up bound of the subset sum $l \leq s \leq u$. if $s < l$ or $s > u$, then there's no solution.

$$
l = \sum_X \{x < 0\}, u = \sum_X \{x > 0\}
\tag{14.64}
$$

As the elements are integers, there are $m = u - l + 1$ columns in table T, each corresponds to a value: $l \leq j \leq u$. There are $n = |X| + 1$ rows, each corresponds to some element x_i. $T[i][j]$ indicates whether exists some subset $S \subseteq \{x_1, x_2, ..., x_i\}$, satisfying $\sum S = j$. Row 0 is special, represents the sum of empty set $\varnothing$. All entries in T start from false F except $T[0][0] = T$, meaning $\sum \varnothing = 0$. Start from x_1 to build row 1. As $\sum \varnothing = 0$, $\sum \{x_1\} = x_1$, hence $T[1][0] = T$, $T[1][x_1] = T$, the rest are all F.

	l	$l+1$	...	0	...	x_1	...	u
$\varnothing$	F	F	...	T	...	F	...	F
x_1	F	F	...	T	...	T	...	F
...	F	F	...	T	...	T	...	F

Add x_2, we get 4 possible subset sums: $\sum \varnothing = 0$, $\sum \{x_1\} = x_1$, $\sum \{x_2\} = x_2$, $\sum \{x_1, x_2\} = x_1 + x_2$.

	l	$l+1$	...	0	...	x_1	...	x_2	...	$x_1 + x_2$	...	u
$\varnothing$	F	F	...	T	...	T	...	F	...	F	...	F
x_1	F	F	...	T	...	T	...	F	...	F	...	F
x_2	F	F	...	T	...	T	...	T	...	T	...	F
...	F	F	...	T	...	T	...	T	...	T	...	F

When add element x_i to fill row i, we can obtain all subset sums from previous elements: $\{x_1, x_2, ..., x_{i-1}\}$, hence all the entries of true in previous row are still true. Because $\sum\{x_i\} = x_i$, hence $T[i][x_i] = T$. We add x_i to each previous sum, generate some new sums, the corresponding entries of them are all true. After adding all n elements, the Boolean value of $T[n][s]$ gives whether the subset sum s exists.

```
1: function SUBSET-SUM(X, s)
2:     l ← ∑{x ∈ X, x < 0}, u ← ∑{x ∈ X, x > 0}
3:     n ← |X|
4:     T ← {{F, F, ...}, {F, F, ...}, ...}                       ▷ (n + 1) × (u − l + 1)
5:     T[0][0] ← T                                               ▷ ∑∅ = 0
6:     for i ← 1 to n do
7:         T[i][X[i]] ← T
8:         for j ← l to u do
9:             T[i][j] ← T[i][j] ∨ T[i − 1][j]
10:            j′ ← j − X[i]
11:            if l ≤ j′ ≤ u then
12:                T[i][j] ← T[i][j] ∨ T[i − 1][j′]
13:    return T[n][s]
```

The column index j does not start from 0, but from l to u. We can convert it by $j - l$ in programming environment. We next generate all subsets S satisfying $\sum S = s$ from table T. If $T[n][s] = F$ then there's no solution; otherwise, there are two cases: (1) if $x_n = s$, then the singleton set $\{x_n\}$ is a solution. We next lookup $T[n-1][s]$, if it's true T, then recursively generate all subsets from $\{x_1, x_2, x_3, ..., x_{n-1}\}$ with the sum s; (2) let $s' = s - x_n$, if $l \leq s' \leq u$ and $T[n-1][s']$ is true, we recursively generate subsets from $\{x_1, x_2, x_3, ..., x_{n-1}\}$ with the sum s', then add x_n to each subset.

```
1: function GET(X, s, T, n)
2:     r ← [ ]
3:     if X[n] = s then
4:         r ← {X[n]} : r
5:     if n > 1 then
6:         if T[n − 1][s] then
7:             r ← r ++ GET(X, s, T, n − 1)
8:         s′ ← s − X[n]
9:         if l ≤ s′ ≤ u and T[n − 1][s′] then
10:            r ← r ++ [(X[n]:r′)|r′ ← GET(X, s′, T, n − 1) ]
11:    return r
```

The dynamic programming method loops $O(n(u-l+1))$ times to build table T, then recursively generates the solution in $O(n)$ levels. The 2D table need $O(n(u-l+1))$ space. We can replace it with a 1D vector V of $u - l + 1$ entries. each $V[j] = \{S_1, S_2, ...\}$ stores the subsets that $\sum S_1 = \sum S_2 = ... = j$. V starts from all empty entries. For each x_i, we update V a round, add the new obtained sums with x_i. The final solution is in $V[s]$.

```
1: function SUBSET-SUM(X, s)
2:     l ← ∑{x ∈ X, x < 0}, u ← ∑{x ∈ X, x > 0}
3:     V ← [∅, ∅, ...]                                           ▷ u − l + 1
4:     for each x in X do
5:         U ← COPY(V)
6:         for j ← l to u do
7:             if x = j then
8:                 U[j] ← {{x}} ∪ U[j]
9:             j′ ← j − x
```

```
10:            if l ≤ j′ ≤ u and V[j′] ≠ ∅ then
11:                U[j] ← U[j] ∪ {({x} ∪ S)|S ∈ V[j′]}
12:        V ← U
13:    return V[s]
```

We can build the solution vector with left fold: $V = foldl\ bld\ (replicate\ (u-l+1)\ \varnothing)\ X$, where *replicate n a* generates list $[a, a, ..., a]$ of length n. *bld* updates V with each element in X.

$$bld\ V\ x = foldl\ f\ V\ [l, l+1..., u] \tag{14.65}$$

Where:

$$f\ V\ j = \begin{cases} j = x: & V[j] \cup \{\{x\}\} \\ l \le j' \le u \text{ and } T[j'] \ne \varnothing: & V[j] \cup \{\{x\}S|S \in T[j']\}, \text{where}: j' = j - x \\ \text{otherwise}: & V \end{cases} \tag{14.66}$$

Exercise 14.12

14.12.1. For the longest common sub-sequence problem, an alternative solution is to record the length and the direction in the table. There are three directions: 'N' for north, 'W' for west, and 'NW'. Given such a table, we can build the longest common sub-sequence from the bottom-right entry. If the entry is 'NW', next go to the upper-left entry; if it's 'N', go to the above row; and go to the previous entry if it's 'W'. Implement this solution.

14.12.2. For the subset sum upper/lower bound, does $l \le 0 \le u$ always hold? can we reduce the range between the bounds?

14.12.3. Given a list of non-negative integers, find the maximum sum composed by numbers that none of them are adjacent.

14.12.4. Edit distance (also known as Levenshtein edit distance) is defined as the cost of converting from one string s to another string t. It is widely used in spell-checking, OCR correction etc. There are three symbol changes: insert, delete, and replace. Each operation mutate a character a time. For example the edit distance is 3 for 'kitten' $\mapsto$ 'sitting':

1. **k**itten → **s**itten (k $\mapsto$ s);

2. sitt**e**n → sitt**i**n (e $\mapsto$ i);

3. sittin → sitting (+ g).

Compute the edit distance with dynamic programming.

14.7 Appendix - example programs

Find the top-k element:

```
Optional<K> top(Int k, [K] xs, Int l, Int u) {
    if l < u {
        swap(xs, l, rand(l, u))
        var p = partition(xs, l, u)
        if p - l + 1 == k
            return Optional.of(xs[p])
        return if k < p - l + 1 then top(k, xs, l, p)
                else top(k- p + l - 1, xs, p + 1, u)
    }
```

```
         return Optional.Nothing
}

Int partition([K] xs, Int l, Int u) {
    var p = l
    for var r = l + 1 to u {
        if not xs[p] < xs[r] {
            l = l + 1
            swap(xs, l, r)
        }
    }
    swap(xs, p, l)
    return l
}
```

Saddle back search:

```
solve f z = search 0 m where
  search p q | p > n || q < 0 = []
             | z' < z = search (p + 1) q
             | z' > z = search p (q - 1)
             | otherwise = (p, q) : search (p + 1) (q - 1)
    where z' = f p q
  m = bsearch (f 0) z (0, z)
  n = bsearch (λx→f x 0) z (0, z)

bsearch f y (l, u) | u ≤ l = l
                   | f m ≤ y = if f (m + 1) ≤ y then bsearch f y (m + 1, u) else m
                   | otherwise = bsearch f y (l, m-1)
  where m = (l + u) `div` 2
```

Boyer-Moore majority:

```
Optional<T> majority([T] xs) {
    var (m, c) = (Optional<T>.Nothing, 0)
    for var x in xs {
        if c == 0 then (m, c) = (Optional.of(x), 0)
        if x == m then c++ else c--
    }
    c = 0
    for var x in xs {
        if x == m then c++
    }
    return if c > length(xs)/2 then m else Optional<T>.Nothing
}
```

Find the majority with fold:

```
majority xs = verify $ foldr maj (Nothing, 0) xs where
  maj x (Nothing, 0) = (Just x, 1)
  maj x (Just y, v) | x == y = (Just y, v + 1)
                    | v == 0 = (Just x, 1)
                    | otherwise = (Just y, v - 1)
  verify (Nothing, _) = Nothing
  verify (Just m, _)  = if 2 * (length $ filter (==m) xs) > length xs
                        then Just m else Nothing
```

The maximum sum of sub-vector:

```
maxSum :: (Ord a, Num a) ⇒ [a] → a
maxSum = fst ∘ foldr f (0, 0) where
  f x (m, mSofar) = (m', mSofar') where
    mSofar' = max 0 (mSofar + x)
    m' = max mSofar' m
```

KMP string matching:

```
[Int] match([T] w, [T]p) {
    n = length(w), m = length(p)
    [Int] fallback = prefixes(p)
    [Int] r = []
    Int k = 0
    for i = 0 to n {
        while k > 0 and p[k] ≠ w[i] {
            k = fallback[k]
        }
        if p[k] == w[i] then k = k + 1
        if k == m {
            add(r, i + 1 - m)
            k = fallback[k - 1]
        }
    }
    return r
}

[Int] prefixes([T] p) {
    m = length(p)
    [Int] t = [0] * m  //fallback table
    Int k = 0
    for i = 2 to m {
        while k > 0 and p[i-1] ≠ p[k] {
            k = t[k-1] #fallback
        }
        if p[i-1] == p[k] then k = k + 1
        t[i] = k
    }
    return t
}
```

The maze puzzle:

```
dfsSolve m from to = solve [[from]] where
  solve [] = []
  solve (c@(p:path):cs)
      | p == to = reverse c
      | otherwise = let os = filter (`notElem` path) (adj p) in
                      if os == [] then solve cs
                      else solve ((map (:c) os) ++ cs)
  adj (x, y) = [(x', y') | (x', y') ← [(x-1, y), (x+1, y), (x, y-1), (x, y+1)],
                          inRange (bounds m) (x', y'), m ! (x', y') == 0]
```

The eight queens puzzle:

```
solve = dfsSolve [[]] [] where
  dfsSolve [] s = s
  dfsSolve (c:cs) s
          | length c == 8 = dfsSolve cs (c:s)
          | otherwise = dfsSolve (([(x:c) | x ← [1..8] \\ c,
                            not $ attack x c] ++ cs) s
  attack x c = let y = 1 + length c in
                any (λ(i, j) → abs(x - i) == abs(y - j)) $
                    zip (reverse c) [1..]
```

The peg puzzle:

```
solve = dfsSolve [[[-1, -1, -1, 0, 1, 1, 1]]] [] where
    dfsSolve [] s = s
    dfsSolve (c:cs) s
            | head c == [1, 1, 1, 0, -1, -1, -1] = dfsSolve cs (reverse c:s)
            | otherwise = dfsSolve ((map (:c) $ moves $ head c) ++ cs) s

moves s = filter (≠s) [leapLeft s, hopLeft s, leapRight s, hopRight s] where
```

```
    leapLeft [] = []
    leapLeft (0:y:1:ys) = 1:y:0:ys
    leapLeft (y:ys) = y:leapLeft ys
    hopLeft [] = []
    hopLeft (0:1:ys) = 1:0:ys
    hopLeft (y:ys) = y:hopLeft ys
    leapRight [] = []
    leapRight (-1:y:0:ys) = 0:y:(-1):ys
    leapRight (y:ys) = y:leapRight ys
    hopRight [] = []
    hopRight (-1:0:ys) = 0:(-1):ys
    hopRight (y:ys) = y:hopRight ys
```

Iterative solution to the peg puzzle:

```
[Int] solve([Int] start, [Int] end) {
    stack = [[start]]
    s = []
    while stack ≠ [] {
        c = pop(stack)
        if c[0] == end {
            s += reverse(c)
        } else {
            for [Int] m in moves(c[0]) {
                stack += (m:c)
            }
        }
    }
    return s
}

[[Int]] moves([Int] s) {
    [[Int]] ms = []
    n = length(s)
    p = find(s, 0)
    if p < n - 2 and s[p+2] > 0 then ms += swap(s, p, p+2)
    if p < n - 1 and s[p+1] > 0 then ms += swap(s, p, p+1)
    if p > 1 and s[p-2] < 0 then ms += swap(s, p, p-2)
    if p > 0 and s[p-1] < 0 then ms += swap(s, p, p-1)
    return ms
}

[Int] swap([Int] s, Int i, Int j) {
    a = copy(s)
    (a[i], a[j]) = (a[j], a[i])
    return a
}
```

The wolf, goat, cabbage cross river puzzle:

```
import Data.Bits
import qualified Data.Sequence as Queue
import Data.Sequence (Seq((:<|)), (><))

solve = bfsSolve $ Queue.singleton [(15, 0)] where
  bfsSolve Queue.Empty = []  -- no solution
  bfsSolve (c@(p:_) :<| cs)
    | fst p == 0 = reverse c
    | otherwise = bfsSolve (cs >< (Queue.fromList $ map (:c)
                                  (filter (`valid` c) $ moves p)))

valid (a, b) r = not $ or [ a `elem` [3, 6], b `elem` [3, 6], (a, b) `elem` r]

moves (a, b) = if b < 8 then trans a b else map swap (trans b a) where
    trans x y = [(x - 8 - i, y + 8 + i)
                    | i ←[0, 1, 2, 4], i == 0 || (x .&. i) ≠ 0]
```

```
        swap (x, y) = (y, x)
```

The extended Euclid algorithm to solve the water jugs puzzle:

```
extGcd 0 b = (b, 0, 1)
extGcd a b = let (d, x', y') = extGcd (b `mod` a) a in
                 (d, y' - x' * (b `div` a), x')

solve a b g | g `mod` d ≠ 0 = []
            | otherwise = solve' (x * g `div` d)
    where
      (d, x, y) = extGcd a b
      solve' x | x < 0 = solve' (x + b)
               | otherwise = pour x [(0, 0)]
      pour 0 ps = reverse ((0, g):ps)
      pour x ps@((a', b'):_) | a' == 0 = pour (x - 1) ((a, b'):ps)
                             | b' == b = pour x ((a', 0):ps)
                             | otherwise = pour x ((max 0 (a' + b' - b),
                                                    min (a' + b') b):ps)
```

BFS solution to the water jugs puzzle:

```
import qualified Data.Sequence as Queue
import Data.Sequence (Seq((:<|)), (><))

solve' a b g = bfs $ Queue.singleton [(0, 0)] where
  bfs Queue.Empty = []
  bfs (c@(p:_) :<| cs)
      | fst p == g || snd p == g = reverse c
      | otherwise = bfs (cs >< (Queue.fromList $ map (:c) $ expand c))
  expand ((x, y):ps) = filter (`notElem` ps) $ map (λf → f x y)
                                 [fillA, fillB, pourA, pourB, emptyA, emptyB]
  fillA _ y = (a, y)
  fillB x _ = (x, b)
  emptyA _ y = (0, y)
  emptyB x _ = (x, 0)
  pourA x y = (max 0 (x + y - b), min (x + y) b)
  pourB x y = (min (x + y) a, max 0 (x + y - a))
```

Iterative BFS for water jugs puzzle:

```
data Step {
    Pair<Int> (p, q)
    Step parent
    Step(Pair<Int>(x, y), Step p = null) {
      (p, q) = (x, y), parent = p
    }
}

Bool (==) (Step a, Step b) = {a.(p, q) == b.(p, q)}
Bool (≠) (Step a, Step b) = not ∘ (==)

[Step] expand(Step s, Int a, Int b) {
    var (p, q) = s.(p, q)
    return [Step(a, q, s), /*fill A*/
            Step(p, b, s), /*fill B*/
            Step(0, q, s), /*empty A*/
            Step(p, 0, s), /*empty B*/
            Step(max(0, p + q - b), min(p + q, b), s), /*pour A into B*/
            Step(min(p + q, a), max(0, p + q - a), s)] /*pour B into A*/
}

Optional<[Step]> solve(Int a, Int b, Int g) {
    q = Queue<Step>(Step(0, 0))
    Set<Step> visited = {head(q)}
    while not empty(q) {
```

```
            var cur = pop(q)
            if cur.p == g || cur.q == g {
                return Optional.of(backtrack(cur))
            } else {
                for s in expand(cur, a, b) {
                    if cur ≠ s and s not in visited {
                        push(q, s)
                        visited += s
                    }
                }
            }
        }
    }
    return Optional.Nothing
}

[Step] backtrack(Step s) {
    [Step] seq
    while s ≠ null {
        seq = s : seq
        s = s.parent
    }
    return seq
}
```

Klotski puzzle:

```
import qualified Data.Map as Map
import qualified Data.Set as Set
import qualified Data.Sequence as Queue
import Data.Sequence (Seq((:<|)), (><))

cellOf (y, x) = y * 4 + x
posOf c = (c `div` 4, c `mod` 4)

cellSet = Set.fromList ∘ (map cellOf)

type Layout = Map.Map Integer (Set.Set Integer)
type NormLayout = Set.Set (Set.Set Integer)
type Move = (Integer, Integer)

start = Map.map cellSet $ Map.fromList
        [(1, [(0, 0), (1, 0)]),
         (2, [(0, 3), (1, 3)]),
         (3, [(2, 0), (3, 0)]),
         (4, [(2, 1), (2, 2)]),
         (5, [(2, 3), (3, 3)]),
         (6, [(4, 0)]), (7, [(3, 1)]), (8, [(3, 2)]), (9, [(4, 3)]),
         (10, [(0, 1), (0, 2), (1, 1), (1, 2)])]

end = cellSet [(3, 1), (3, 2), (4, 1), (4, 2)]

normalize = Set.fromList ∘ Map.elems

mirror = Map.map (Set.map f) where
  f c = let (y, x) = posOf c in cellOf (y, 3 - x)

klotski = solve q visited where
  q = Queue.singleton (start, [])
  visited = Set.singleton (normalize start)

solve Queue.Empty _ = []
solve ((x, ms) :<| cs) visited | Map.lookup 10 x == Just end = reverse ms
                               | otherwise = solve q visited'
  where
    q = cs >< (Queue.fromList [(move x op, op:ms) | op ← ops ])
    visited' = foldr Set.insert visited (map (normalize ∘ move x) ops)
```

```
      ops = expand x visited

expand x visited = [(i, d) | i ←[1..10], d ← [-1, 1, -4, 4],
                             valid i d, unique i d]
  where
    valid i d = let p = trans d (maybe Set.empty id $ Map.lookup i x) in
                   (not $ any (outside d) p) &&
                   (Map.keysSet $ Map.filter (overlapped p) x)
                      `Set.isSubsetOf` Set.singleton i
    outside d c = c < 0 || c ≥ 20 ||
                  (d == 1 && c `mod` 4 == 0) || (d == -1 && c `mod` 4 == 3)
    unique i d = let ly = move x (i, d) in all (`Set.notMember` visited)
                    [normalize ly, normalize (mirror ly)]

move x (i, d) = Map.update (Just ∘ trans d) i x

trans d = Set.map (d+)

overlapped :: (Set.Set Integer) → (Set.Set Integer) → Bool
overlapped a b = (not ∘ Set.null) $ Set.intersection a b
```

Iterative solution to the Klotski puzzle:

```
type Layout = [Set<Int>]

Layout START = [{0, 4}, {3, 7}, {8, 12}, {9, 10},
    {11, 15},{16},{13}, {14}, {19}, {1, 2, 5, 6}]

Set<Int> END = {13, 14, 17, 18}

(Int, Int) pos(Int c) = (y = c / 4, x = c mod 4)

[[Int]] matrix(Layout layout) {
    [[Int]] m = replicate(replicate(0, 4), 5)
    for Int i, var p in (zip([1, 2, ...], layout)) {
        for var c in p {
            y, x = pos(c)
            m[y][x] = i
        }
    }
    return m
}

data Node {
    Node parent
    Layout layout

    Node(Layout l, Node p = null) {
        layout = l, parent = p
    }
}

//usage: solve(START, END)
Optional<Node> solve(Layout start, Set<Int> end) {
    var visit = {Set(start)}
    var queue = Queue.of(Node(start))
    while not empty(queue) {
        cur = pop(queue)
        if last(cur.layout) == end {
            return Optional.of(cur)
        } else {
            for ly in expand(cur.layout, visit) {
                push(queue, Node(ly, cur))
                add(visit, Set(ly))
            }
        }
```

```
    }
    return Optional.None
}

[Layout] expand(Layout layout, Set<Set<Layout>> visit):
    Bool bound(Set<Int> piece, Int d) {
        for c in piece {
            if c + d < 0 or c + d ≥ 20 then return False
            if d == 1 and c mod 4 == 3 then return False
            if d == -1 and c mod 4 == 0 then return False
        }
        return True
    }

    var m = matrix(layout)
    Bool valid(Set<Int> piece, Int d, Int i) {
        for c in piece {
            y, x = pos(c + d)
            if m[y][x] not in [0, i] then return False
        }
        return True
    }

    Bool unique(Layout ly) {
        n = Set(ly)
        Set<Set<Int>> m = map(map(c → 4 * (c / 4) + 3 - (c mod 4), p), n)
        return (n not in visit) and (m not in visit)
    }

    [Layout] s = []
    for i, p in zip([1, 2, ...], layout) {
        for d in [-1, 1, -4, 4] {
            if bound(p, d) and valid(p, d, i) {
                ly = move(layout, i - 1, d)
                if unique(ly) then s.append(ly)
            }
        }
    }
    return
}

Layout move(Layout layout, Int i, Int d) {
    ly = clone(layout)
    ly[i] = map((d+), layout[i])
    return ly
}
```

Code, decode with a Huffman tree:

```
code = Map.fromList ∘ (traverse []) where
  traverse bits (Leaf _ c) = [(c, reverse bits)]
  traverse bits (Branch _ l r) = traverse (0:bits) l ++ traverse (1:bits) r

encode dict = concatMap (dict !)

decode tr cs = find tr cs where
  find (Leaf _ c) [] = [c]
  find (Leaf _ c) bs = c : find tr bs
  find (Branch _ l r) (b:bs) = find (if b == 0 then l else r) bs
```

Greedy change-making:

```
import qualified Data.Set as Set
import Data.List (group)

solve x = assoc ∘ change x where
```

```
    change 0 _ = []
    change x cs = let c = Set.findMax $ Set.filter (≤ x) cs in c : change (x - c) cs
    assoc = (map (λcs → (head cs, length cs))) ∘ group

example = solve 142 $ Set.fromList [1, 5, 25, 50, 100]
```

Dynamic programming change-making:

```
[Int] change(Int x, Set<Int> cs) {
    t = [(0, None)] ++ [(x + 1, None)] * x
    for i = 1 to x {
        for c in cs {
            if c ≤ i {
                (n, _) = t[i - c]
                (m, _) = t[i]
                if 1 + n < m then t[i] = (1 + n, c)
    s = []
    while x > 0:
        (_, c) = t[x]
        s += c
        x = x - c
    return s
}
```

Dynamic programming with fold to solve the change-making problem:

```
import qualified Data.Set as Set
import Data.Sequence ((|>), singleton, index)

changemk x cs = makeChange x $ foldl fill (singleton (0, 0)) [1..x] where
  fill tab i = tab |> (n, c) where
    (n, c) = minimum $ Set.map lookup $ Set.filter (≤ i) cs
    lookup c  = (1 + fst (tab `index` (i - c)), c)
  makeChange 0 _ = []
  makeChange x tab = let c = snd $ tab `index` x in c : makeChange (x - c) tab
```

The longest common sub-sequence:

```
[K] lcs([K] xs, [K] ys) {
    Int m = length(xs), n = length(ys)
    [[Int]] c = [[0]*(n + 1)]*(m + 1)
    for i = 1 to m {
        for j = 1 to n {
            if xs[i-1] == ys[j-1] {
                c[i][j] = c[i-1][j-1] + 1
            } else {
                c[i][j] = max(c[i-1][j], c[i][j-1])
            }
        }
    }
    return fetch(c, xs, ys)
}

[K] fetch([[Int]] c, [K] xs, [K] ys) {
    [K] r = []
    var m = length(xs), n = length(ys)
    while m > 0 and n > 0 {
        if xs[m - 1] == ys[n - 1] {
            r += xs[m - 1]
            m = m - 1
            n = n - 1
        } else if c[m - 1][n] > c[m][n - 1] {
            m = m - 1
        } else {
            n = n - 1
        }
```

```
    }
    return reverse(r)
}
```

Existence of the subset sum:

```
Bool subsetsum([Int] xs, Int s) {
    Int l = 0, u = 0, n = length(xs)
    for x in xs {
      if x > 0 then u++ else l++
    }
    tab = [[False]*(u - l + 1)] * (n + 1)
    tab = [0][0 - l] = True
    for i, x in zip([1, 2, ..., n], xs) {
        tab[i][x - l] = True
        for j = l to u {
            tab[i][j - l] or = tab[i-1][j - l]
            j1 = j - x
            if l ≤ j1 ≤ u then tab[i][j - l] or = tab[i-1][j1 - l]
        }
    }
    return tab[n][s - l]
}
```

Solve the subset sum with a vector:

```
{{Int}} subsetsum(xs, s) {
    Int l = 0, u = 0, n = length(xs)
    for x in xs {
      if x > 0 then u++ else l++
    }
    tab = {} * (u - l + 1)
    for x in xs {
        tab1 = copy(tab)
        for j = low to up {
            if x == j then add(tab1[j], {x})
            j1 = j - x
            if low ≤ j1 ≤ up and tab[j1] {
                tab1[j] |= {add(ys, x) for ys in tab[j1]}
            }
        }
        tab = tab1
    }
    return tab[s]
}
```

Appendix A

Imperative delete for red-black tree

We need handle more cases for imperative *delete* than *insert*. To resume balance after cutting off a node from 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:     $p \leftarrow$ PARENT(x)
3:     $q \leftarrow$ NIL
4:     **if** LEFT(x) = NIL **then**
5:         $q \leftarrow$ RIGHT(x)
6:         REPLACE(x, RIGHT(x))   ▷ replace x with its right sub-tree
7:     **else if** RIGHT(x) = NIL **then**
8:         $q \leftarrow$ LEFT(x)
9:         REPLACE(x, LEFTx())   ▷ replace x with its left sub-tree
10:     **else**
11:         $y \leftarrow$ MIN(RIGHT(x))
12:         $p \leftarrow$ PARENT(y)
13:         $q \leftarrow$ RIGHT(y)
14:         KEY(x) $\leftarrow$ KEY(y)
15:         copy data from y to x
16:         REPLACE(y, RIGHT(y))   ▷ replace y with its right sub-tree
17:         $x \leftarrow y$
18:     **if** COLOR(x) = BLACK **then**
19:         $T \leftarrow$ 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 it. 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 fig. A.1.
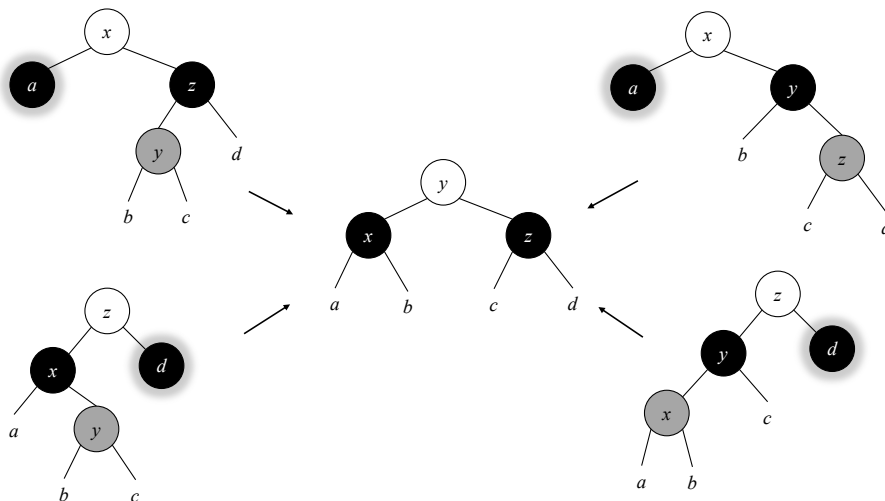
Figure A.1: The doubly black node has a black sibling, and a red nephew. It can be fixed with a rotation.

```
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 fig. A.2, change a or c to black. We can add this fixing to the previous implementation.
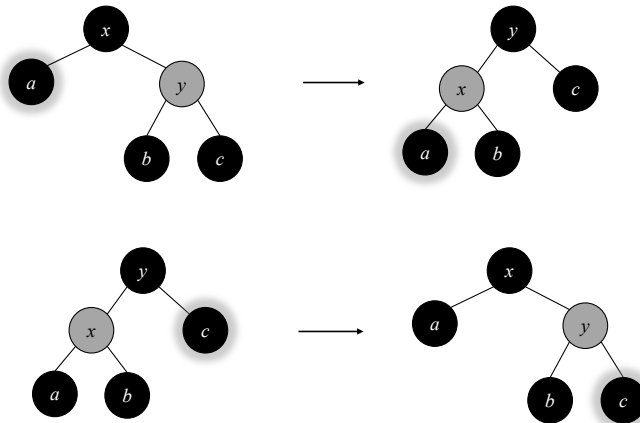
Figure A.2: The sibling of the doubly black is red

```
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-LEFTT, PARENT(x)
15:          **else**   ▷ x is the right
16:              $T \leftarrow$ ROTATE-RIGHTT, 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 fig. A.3, there are two symetric sub-cases.
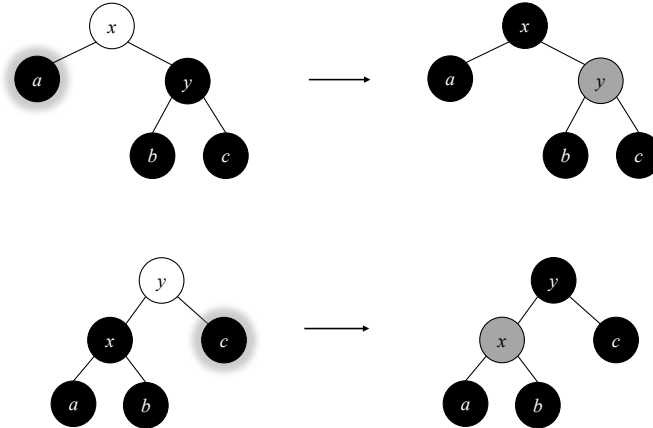
Figure A.3: move the blackness up

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-LEFTT, PARENT(x)
15:                 **else**   ▷ x is the right
16:                     $T \leftarrow$ ROTATE-RIGHTT, 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 re-placeWith 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.

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:

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

Proof. When insert, the height can not increase both on left and right. We can explain the 4 cases from the balance factor definition, which is the difference of the right and left sub-trees:

1. If $\delta \geq 0$ and $\delta' \geq 0$, it means the height of the right sub-tree is not less than the left sub-tree before and after insertion. In this case, the height increment is only 'contributed' from the right, which is Δr.

2. If $\delta \leq 0$, it means the height of left sub-tree is not less than the right before. Since $\delta' \geq 0$ after insert, we know the height of right sub-tree increases, and the left side keeps same ($|l'| = |l|$). The height increment is:

$$
\begin{aligned}
\Delta H &= max(|r'|, |l'|) - max(|r|, |l|) \quad &\{\delta \leq 0 \text{ and } \delta' \geq 0\} \\
&= |r'| - |l| \quad &\{|l| = |l'|\} \\
&= |r| + \Delta r - |l| \\
&= \delta + \Delta r
\end{aligned}
$$

3. If $\delta \geq 0$ and $\delta' \leq 0$, similar to the above case, we have the following:

$$
\begin{aligned}
\Delta H &= max(|r'|, |l'|) - max(|r|, |l|) \quad &\{\delta \geq 0 \text{ and } \delta' \leq 0\} \\
&= |l'| - |r| \\
&= |l| + \Delta l - |r| \\
&= \Delta l - \delta
\end{aligned}
$$

4. Otherwise, δ and δ' are not bigger than zero. It means the height of the left sub-tree is not less than the right. The height increment is only 'contributed' from the left, which is Δl.

$\square$

B.2   Balance adjustment after insert

The balance factors are ± 2 in the 4 cases shown in figure fig. B.1. After fixing, $\delta(y)$ resumes to 0. The height of left and right sub-trees are equal.
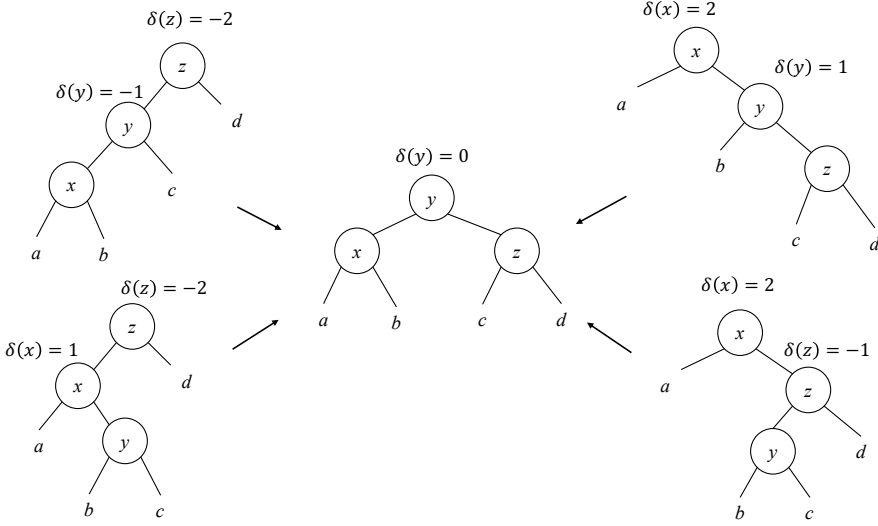
Figure B.1: Fix 4 cases to the same structure

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:

$$\begin{aligned} \delta(y) = |c| - |x| = -1 &\Rightarrow |c| = |x| - 1 \\ \delta(z) = |d| - |y| = -2 &\Rightarrow |d| = |y| - 2 \end{aligned} \tag{B.2}$$

After fixing:

$$\begin{aligned} \delta'(z) &= |d| - |c| & \{from(eq. \ (\text{B.2}))\} \\ &= |y| - 2 - (|x| - 1) \\ &= |y| - |x| - 1 & \{x \text{ is sub-tree of } y \Rightarrow |y| - |x| = 1\} \\ &= 0 \end{aligned} \tag{B.3}$$

For $\delta'(y)$, we have the following:

$$
\begin{aligned}
\delta'(y) &= |z| - |x| \\
&= 1 + max(|c|, |d|) - |x| \quad \{\text{by (eq. (B.3)), } |c| = |d|\} \\
&= 1 + |c| - |x| \quad\quad\quad\quad \{\text{by (eq. (B.2))}\} \\
&= 1 + |x| - 1 - |x| \\
&= 0
\end{aligned}
\tag{B.4}
$$

Summarize the above, the balance factors change to the following in left-left case:

$$
\begin{aligned}
\delta'(x) &= \delta(x) \\
\delta'(y) &= 0 \\
\delta'(z) &= 0
\end{aligned}
\tag{B.5}
$$

Right-right

The right-right case is symmetric to left-left:

$$
\begin{aligned}
\delta'(x) &= 0 \\
\delta'(y) &= 0 \\
\delta'(z) &= \delta(z)
\end{aligned}
\tag{B.6}
$$

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:

$$
\begin{aligned}
|z| &= 1 + max(|y|, |d|) \quad \{\delta(z) = -1 \Rightarrow |y| > |d|\} \\
&= 1 + |y| \\
&= 2 + max(|b|, |c|)
\end{aligned}
\tag{B.8}
$$

Since $\delta(x) = 2$, we have:

$$
\begin{aligned}
\delta(x) = 2 \quad &\Rightarrow |z| - |a| = 2 \quad\quad\quad\quad\quad \{\text{by (eq. (B.8))}\} \\
&\Rightarrow 2 + max(|b|, |c|) - |a| = 2 \\
&\Rightarrow max(|b|, |c|) - |a| = 0
\end{aligned}
\tag{B.9}
$$

If $\delta(y) = |c| - |b| = 1$, then:

$$
max(|b|, |c|) = |c| = |b| + 1
\tag{B.10}
$$

Take this into (eq. (B.9)) gives:

$$
\begin{aligned}
|b| + 1 - |a| = 0 &\Rightarrow |b| - |a| = -1 \quad \{\text{by (eq. (B.7)) }\} \\
&\Rightarrow \delta'(x) = -1
\end{aligned}
\tag{B.11}
$$

If $\delta(y) \neq 1$, then $max(|b|, |c|) = |b|$. Take this into (eq. (B.9)) gives:

$$
\begin{aligned}
|b| - |a| = 0 &\quad \{\text{by (eq. (B.7))}\} \\
&\Rightarrow \delta'(x) = 0
\end{aligned}
\tag{B.12}
$$

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:

$$\begin{aligned} \delta'(z) &= |d| - |c| & \{\delta(z) = -1 = |d| - |y|\} \\ &= |y| - |c| - 1 & \{|y| = 1 + max(|b|, |c|)\} \\ &= max(|b|, |c|) - |c| \end{aligned}$$
(B.14)

If $\delta(y) = |c| - |b| = -1$, then $max(|b|, |c|) = |b| = |c| + 1$. Take this into (eq. (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:

$$\delta'(z) = \begin{cases} \delta(y) = -1 : & 1 \\ otherwise : & 0 \end{cases}$$
(B.15)

Finally, for $\delta'(y)$, we deduce it like below:

$$\begin{aligned} \delta'(y) &= |z| - |x| \\ &= max(|c|, |d|) - max(|a|, |b|) \end{aligned}$$
(B.16)

There are three cases:

1. If $\delta(y) = 0$, then $|b| = |c|$. According to (eq. (B.13)) and (eq. (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 (eq. (B.15)), we have $\delta'(z) = 0 \Rightarrow |c| = |d|$.

$$\begin{aligned} \delta'(y) &= max(|c|, |d|) - max(|a|, |b|) & \{|c| = |d|\} \\ &= |c| - max(|a|, |b|) & \{\text{from (eq. (B.13)):} \ \delta'(x) = -1 \Rightarrow |b| - |a| = -1\} \\ &= |c| - (|b| + 1) & \{\delta(y) = 1 \Rightarrow |c| - |b| = 1\} \\ &= 0 \end{aligned}$$

3. If $\delta(y) = -1$, from (eq. (B.13)), we have $\delta'(x) = 0 \Rightarrow |a| = |b|$.

$$\begin{aligned} \delta'(y) &= max(|c|, |d|) - max(|a|, |b|) & \{|a| = |b|\} \\ &= max(|c|, |d|) - |b| & \{\text{from (eq. (B.15)):} \ |d| - |c| = 1\} \\ &= |c| + 1 - |b| & \{\delta(y) = -1 \Rightarrow |c| - |b| = -1\} \\ &= 0 \end{aligned}$$

All three cases lead to the same result $\delta'(y) = 0$. Summarize all above, we get the updated balance factors after fixing as below:

$$\begin{aligned} \delta'(x) &= \begin{cases} \delta(y) = 1 : & -1 \\ otherwise : & 0 \end{cases} \\ \delta'(y) &= 0 \\ \delta'(z) &= \begin{cases} \delta(y) = -1 : & 1 \\ otherwise : & 0 \end{cases} \end{aligned}$$
(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 (eq. (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 ΔH is the height decrement. We define *delete* as below:

$$delete = fst \circ del \tag{B.18}$$

where $del(T, k)$ does the actual work to delete element k from T:

$$
\begin{aligned}
del \; \varnothing \; k \;\; &= \;\; (\varnothing, 0) \\
del \; (l, k', r, \delta) \;\; &= \;\;
\begin{cases}
k < k' : & tree \; (del \; l \; k) \; k' \; (r, 0) \; \delta \\
k > k' : & tree \; (l, 0) \; k' \; (del \; r \; k) \; \delta \\
k = k' : &
\begin{cases}
l = \varnothing : & (r, -1) \\
r = \varnothing : & (l, -1) \\
else : & tree \; (l, 0) \; k'' \; (del \; r \; k'') \; \delta \\
& where \; k'' = min(r)
\end{cases}
\end{cases}
\end{aligned}
\tag{B.19}
$$

If the tree is empty, the result is $(\varnothing, 0)$; otherwise, let the tree be $T = (l, k', r, \delta)$. We compare the k and k', lookup and delete recursively. When $k = k'$, we locate the node to be deleted. If it has either empty sub-tree, we cut the node off, and replace it with the other sub-tree; otherwise, we use the minimum k'' in the right sub-tree to replace k', and cut k'' off. We re-use the *tree* function and ΔH result. Additional to the *insert* cases, there are two cases violate AVL rule, and need fixing. As shown in figure fig. B.2, both cases can be fixed by a tree rotation. We define them as pattern matching:
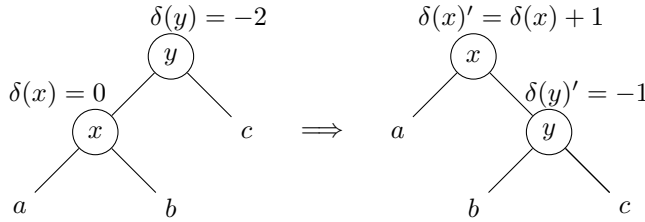
(a) Fix case A
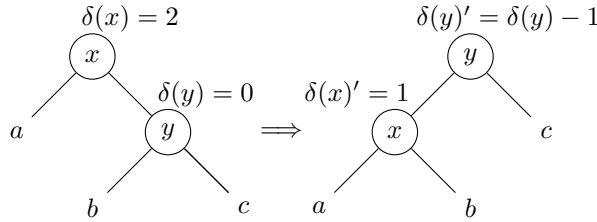
(b) Fix case B

Figure B.2: delete fix

$$
\begin{aligned}
&\cdots \\
balance \; ((a, x, b, \delta(x)), y, c, -2) \; \Delta H \;\; &= \;\; (a, x, (b, y, c, -1), \delta(x) + 1, \Delta H) \\
balance \; (a, x, (b, y, c, \delta(y)), 2) \; \Delta H \;\; &= \;\; ((a, x, b, 1), y, c, \delta(y) - 1, \Delta H) \\
&\cdots
\end{aligned}
\tag{B.20}
$$

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 ← PARENT(x)
 5:     if LEFT(x) = NIL then
 6:         y ← RIGHT(x)
 7:         replace x with y
 8:     else if RIGHT(x) = NIL then
 9:         y ← LEFT(x)
10:         replace x with y
11:     else
12:         z ← MIN(RIGHT(x))
13:         copy data from z to x
14:         p ← PARENT(z)
15:         y ← RIGHT(z)
```

16:          replace z with y

17:    **return** AVL-DELETE-FIX(T, p, y)

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 fig. B.2.

1: **function** AVL-DELETE-FIX(T, p, x)
2:    **while** $p \neq$ NIL **do**
3:        $l \leftarrow$ LEFT(p), $r \leftarrow$ RIGHT(p)
4:        $\delta \leftarrow \delta(p)$, $\delta' \leftarrow \delta$
5:        **if** $x = l$ **then**
6:            $\delta' \leftarrow \delta' + 1$
7:        **else**
8:            $\delta' \leftarrow \delta' - 1$
9:        **if** p is leaf **then**   ▷ $l = r =$ NIL
10:            $\delta' \leftarrow 0$
11:        **if** $|\delta| = 1 \wedge |\delta'| = 0$ **then**
12:            $x \leftarrow p$
13:            $p \leftarrow$ PARENT(x)
14:        **else if** $|\delta| = 0 \wedge |\delta'| = 1$ **then**
15:            **return** T
16:        **else if** $|\delta| = 1 \wedge |\delta'| = 2$ **then**
17:            **if** $\delta' = 2$ **then**
18:                **if** $\delta(r) = 1$ **then**   ▷ Right-right
19:                    $\delta(p) \leftarrow 0$
20:                    $\delta(r) \leftarrow 0$
21:                    $p \leftarrow r$
22:                    $T \leftarrow$ LEFT-ROTATE(T, p)
23:                **else if** $\delta(r) = -1$ **then**   ▷ Right-left
24:                    $\delta_y \leftarrow \delta($ LEFT(r) $)$
25:                    **if** $\delta_y = 1$ **then**
26:                        $\delta(p) \leftarrow -1$
27:                    **else**
28:                        $\delta(p) \leftarrow 0$
29:                    $\delta($ LEFT(r) $) \leftarrow 0$
30:                    **if** $\delta_y = -1$ **then**

```
31:                    δ(r) ← 1
32:                else
33:                    δ(r) ← 0
34:            else                                        ▷ Delete specific right-right
35:                δ(p) ← 1
36:                δ(r) ← δ(r) − 1
37:                T ← LEFT-ROTATE(T, p)
38:                break                                   ▷ No furthur height change
39:        else if δ' = −2 then
40:            if δ(l) = −1 then                           ▷ Left-left
41:                δ(p) ← 0
42:                δ(l) ← 0
43:                p ← l
44:                T ← RIGHT-ROTATE(T, p)
45:            else if δ(l) = 1 then                       ▷ Left-right
46:                δ_y ← δ( RIGHT(l) )
47:                if δ_y = −1 then
48:                    δ(p) ← 1
49:                else
50:                    δ(p) ← 0
51:                δ( RIGHT(l) ) ← 0
52:                if δ_y = 1 then
53:                    δ(l) ← −1
54:                else
55:                    δ(l) ← 0
56:            else                                        ▷ Delete specific left-left
57:                δ(p) ← −1
58:                δ(l) ← δ(l) + 1
59:                T ← RIGHT-ROTATE(T, p)
60:                break                                   ▷ No furthur height change
                                                           ▷ Height decreases, go on bottom-up updating
61:            x ← p
62:            p ← PARENT(x)
63:        if p = NIL then                                 ▷ Delete the root
64:            return x
65:    return T
```

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
}
```

Appendix C

Answers

Answer of exercise 1

1.1. For the free number puzzle, since all numbers are not negative, we can reuse the sign as a flag. For every $|x| < n$ (where n is the length), negate the number at position $|x|$. Then scan to find the first positive number. Its position is the answer. Write a program to realize this solution.

```
Int minFree([Int] nums) {
    var n = length(nums)
    for Int i = 0 to n - 1 {
        var k = abs(nums[i])
        if k < n then nums[k] = -abs(nums[k])
    }
    for Int i = 0 to n - 1 {
        if nums[i] > 0 then return i
    }
    return n
}
```

1.2. 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.

For example $X = [3, 1, 3, 5, 4]$, the missing number $x = 2$, the duplicated one $y = 3$. We give 4 methods: (1) divide and conquer; (2) pigeon hole sort; (3) sign encoding; and (4) equations.

Divide and conquer: Partition the numbers with the middle point $m = \lfloor \frac{1+n}{2} \rfloor$: the left $as = [a \leq m, a \leftarrow X]$, and the right $bs = [b > m, b \leftarrow X]$. If the length of $|as| < m$, then the missing number is on the left, let $s = 1 + 2 + ... + m = \frac{m(m+1)}{2}$, then $x = s - sum(as)$. We can also calculate the missing one on the right. Let $s' = (m + 1) + (m + 2) + ... + n = \frac{(n+m+1)(n-m)}{2}$, then $y = sum(bs) - s'$. If the length of $|as| > m$, then the duplicated number is on the left. Use the similar method, we calculate the missing number $x = s' - sum(bs)$, and the duplicated number $y = sum(as) - s$. Otherwise if the length $|as| = m$, then there are m numbers not greater than m. But we don't know whether they are some permutation of 1, 2, ..., m. We can calculate and compare $sum(as)$ and s. If equal, then we can drop all numbers on the left, then recursively find x and y on the right; otherwise, we drop the right, recursively find on the left. In recursive

293

finding, we need replace the lower bound of 1 with l. Because we halve the list every time, the overall performance is $O(n)$ according to the master theorem.

```
missDup xs = solve xs 1 (length xs) where
  solve xs@(_:_:_) l u | k < m - l + 1 = (sl - sl', sr' - sr)
                       | k > m - l + 1 = (sr - sr', sl' - sl)
                       | sl == sl' = solve bs (m + 1) u
                       | otherwise = solve as l m
      where
          m = (l + u) `div` 2
          (as, bs) = partition (≤m) xs
          k = length as
          sl = (l + m) * (m - l + 1) `div` 2
          sr = (m + 1 + u) * (u - m) `div` 2
          (sl', sr') = (sum as, sum bs)
```

Pigeon hole sort. Since all numbers are within the range from 1 to n, we can do pigeon hole sort. Scan from left to right, for every number x at position i, if $x \neq i$, we swap it with number y at position x. We find the duplicated number if $x = y$, besides, we find the missing number i. Repeat this till $x = i$ or meet the duplicated number. Because every number is swapped to its right position a time, the total performance is $O(n)$.

```
(Int, Int) missDup([Int] xs) {
    Int miss = -1, dup = -1
    for Int i = 0 to length(xs) - 1 {
        while xs[i] ≠ i {
            Int j = xs[i]
            if xs[j] == xs[i] {
                dup = xs[j]
                miss = i
                break
            } else {
                j = xs[i]
                (xs[i], xs[j]) = (xs[j], xs[i])
            }
        }
    }
    return (miss, dup)
```

Sign encoding. Setup an array of n flags. For every number x, mark the x-th flag in the array true. When meet the duplicated number, the corresponding flag was marked before. Let the duplicated number be d, we know $s = 1 + 2 + ... + n = \dfrac{n(n+1)}{2}$, and the sum s' of all numbers. We can calculate the missing number $m = d + s - s'$. However, this method need additional n flags. The existence of a number is a type of binary information (yes/no), we can encode it as the positive/negative sign, hence re-use the space. For every x, flip the number at position $|x|$ to negative, where $|x|$ is the absolute value. If a number of some position is already negative, it's the duplicated one, and we can next calculate the missing one.

```
(Int, Int) missDup([Int] xs) {
    Int miss = -1, dup = -1
    Int n = length(xs)
    Int s = sum(xs)
    for i = 0 to n - 1 {
        Int j = abs(xs[i]) - 1
        if xs[j] < 0 {
            dup = j
            miss = dup + n * (n + 1) / 2 - s
            break
```

```
        }
        xs[j] = -abs(xs[j])
    }
    return (miss, dup)
```

Equation. Consider a simplified problem: random drop a number after shuffle 1 to n, how to find it? We sum all the numbers, then subtract it from $\dfrac{n(n+1)}{2}$:

$$m = s - s'$$

Where m is the missing number, s is the sum from 1 to n, s' is the sum of all numbers. However, for a missing number and a duplicated number, we can't solve with only one equation:

$$\sum(x[i] - i) = d - m \tag{3}$$

Where the left hand is the sum of the i-th number minus i. Can we figure out a second equation? We can use square: sum the difference between the square of the i-th number and the square of i:

$$\sum(x[i]^2 - i^2) = d^2 - m^2 = (d + m)(d - m) \tag{4}$$

Since $d - m \neq 0$, we can divide eq. (3) by eq. (4) on both sides to get another equation:

$$\sum(x[i]^2 - i^2)/\sum(x[i] - i) = d + m \tag{5}$$

Compare equation eq. (3) and eq. (5), there are two equations with two unknowns. We can solve them:

$$
\begin{cases}
m = \dfrac{1}{2}\left(\dfrac{\sum(x[i]^2 - i^2)}{\sum(x[i] - i)} - \sum(x[i] - i)\right) \\
d = \dfrac{1}{2}\left(\dfrac{\sum(x[i]^2 - i^2)}{\sum(x[i] - i)} + \sum(x[i] - i)\right)
\end{cases}
$$

```
missDup xs = ((b `div` a - a) `div` 2, (b `div` a + a) `div` 2)
  where
    ys = zip xs [1..]
    a = sum [x - y | (x, y) ← ys]
    b = sum [x^2 - y^2 | (x, y) ← ys]
```

1.3. Yes, it's essentially a solution based on queue.

Answer of exercise 1.2

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

We use type constraint to ensure compare lists of the same type.

```
(==) :: [a] → [a] → Bool
[] == [] = True
(x:xs) == (y:ys) = x == y && (xs == ys)
xs == ys = False
```

<div align="center">**Answer of exercise 1.3**</div>

1.3.1. For the iterative GET-AT(i, X), what is the behavior when X is empty? what if i
is out of bound?
The behavior is undefined. We can handle it with `Optional<T>`:

```
Optional<T> getAt(List<T> xs, Int i) {
    while i ≠ 0 and xs ≠ null {
        xs = xs.next
        i--
    }
    return if xs ≠ null then Optional.of(xs.key) else Optional.Nothing
}
```

<div align="center">**Answer of exercise 1.4**</div>

1.4.1. In the INIT algorithm, can we use APPEND$(X', \text{FIRST}(X))$ instead of CONS?
APPEND need traverse to the tail every time, hence degrades the performance from
$O(n)$ to $O(n^2)$, where n is the length.

1.4.2. How to handle empty list or out of bound error in LAST-AT?
Use `Optional<T>`:

```
Optional<T> lastAt(List<T> xs, Int i) {
    List<T> p = xs
    while i ≠ 0 and xs ≠ null {
        xs = xs.next
        i--
    }
    if xs == null then return Optional.Nothing
    while xs.next ≠ null {
        xs = xs.next
        p = p.next
    }
    return Optional.of(p.key)
}
```

<div align="center">**Answer of exercise 1.5**</div>

1.5.1. Add the 'tail' reference, optimize the *append* to constant time.
We need wrap the linked-list with the head and tail variables:

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

    Node<A> head = null
    Node<A> tail = null
    Int length = 0
}

List<A> append(List<A> xs, A x) {
    List.Node<A> tl = xs.tail
    xs.tail = List.Node<A>(x, null)
    if tl == null {
        xs.head = xs.tail
    } else {
        tl.next = xs.tail
    }
```

```
        xs.length++
        return xs
}
```

1.5.2. When need update the tail reference? How does it affect the performance?
We need update the tail reference when append to, delete from the tail, add to empty list, delete the element from a singleton list, and split the list. All operations are in constant time except for the splitting $as = bs + cs$. We need linear time to reach the tail of bs.

1.5.3. Handle the empty list and out of bound error for $setAt$.
$setAt\ 0\ x\ [\ ]$ means $x : [\ ]$. Let $|xs| = n$, we treat $setAt\ n\ x\ xs$ same as $xs + [x]$. Other out of bound cases will raise exception.

Answer of exercise 1.6

1.6.1. Handle the out-of-bound case when insert, treat it as append.

```
insert n x [] = [x]
insert n x (y:ys) | n ≤ 0 = x : y : ys
                  | otherwise y : insert (n - 1) x ys
```

1.6.2. Implement insert for array. When insert at position i, all elements after i need shift to the end.

```
[K] insert([K] xs, Int i, K x) {
    append(xs, x)
    for Int j = length(xs) - 1, j > i, j-- {
        swap(xs[j], xs[j-1])
    }
    return xs
}
```

Answer of exercise 1.7

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

```
delAll x [] = []
delAll x (y:ys) | x == y = delAll x ys
                | otherwise = y : delAll x ys
```

Or use $foldr$:

```
delAll x = foldr f [] where
  f y ys = if x == y then ys else y : ys
```

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

```
[K] delAt(Int i, [K] xs) {
    Int n = length(xs)
    if 0 ≤ i < n {
        while i < n - 1 {
            xs[i] = xs[i + 1]
            i++
        }
        popLast(xs)
    }
    return xs
}
```

Answer of exercise 1.8

1.8.1. Change *length* to tail recursive.

```
length  = len 0 where
  len n [] = n
  len n (x:xs) = len (n + 1) xs
```

1.8.2. Compute b^n through the binary format of n.
b needn't necessarily be integer, it can be real, or any element in a set that defines the unit e (e.g. 1) and binary operation (e.g. multiply). We call such set monoid in mathematics.

```
Monoid pow(Monoid b, Int n) {
    Monoid a = Monoid.e
    while n ≠ 0 {
        if n & 1 == 1 then a = a * b
        b = b * b
        n = n >> 1
    return a
}
```

Answer of exercise 1.9

1.9.1. Find the maximum v in a list of pairs $[(k, v)]$ in tail recursive way.

```
maxValue ((k, v):kvs) = maxV v kvs where
  maxV m [] = m
  maxV m ((_, v):kvs) = maxV (max m v) kvs
```

Answer of exercise 1.10

1.10.1. Change the *take/drop* implementation. When n is negative, returns $[\,]$ for *take*, and the entire list for *drop*.

```
safeTake n [] = []
safeTake n (x:xs) | n ≤ 0 = []
                  | otherwise = x : safeTake (n - 1) xs

safeDrop n [] = []
safeDrop n (x:xs) | n ≤ 0 = x:xs
                  | otherwise = safeDrop (n - 1) xs
```

1.10.2. Implement the in-place imperative *take/drop*.
We need append to/delete from the tail of array, otherwise, need shift elements. Let the length of array xs be n, we implement $take(m, xs)$ by deleting $n - m$ elements from tail. For $drop(m, xs)$, we start from $xs[m]$, move every element $xs[i]$ ahead to $xs[i - m]$. Finally remove m elements from the tail.

```
[K] take(Int m, [K] xs) {
    Int d = length(xs) - m
    while d > 0 and xs ≠ [] {
        pop(xs)
        d--
    }
    return xs
}

[K] drop(Int m, xs) {
    if m ≤ 0 then return xs
```

```
        for Int i = m to length(xs) { xs[i - n] = xs[i] }
        while m > 0 and xs ≠ [] {
            pop(xs)
            m--
        }
        return xs
}
```

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

```
sublist from cnt = take cnt ∘ drop (from - 1)

slice from to = drop (from - 1) ∘ take to
```

1.10.4. Consider the below *span* implementation:

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

What is the difference here?
It separates xs into two: all elements satisfying p and the others: $as = [p(a), a \leftarrow xs]$ and $bs = [not\ p(b), b \leftarrow xs]$. as is not necessarily the longest prefix that satisfies p, but the longest sub-sequence satisfying p.

Answer of exercise 1.11

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

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

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

$foldl$ has the type of $foldl :: (B \rightarrow A \rightarrow B) \rightarrow B \rightarrow [A] \rightarrow B$. We need swap the arguments of insert for $foldl$:

$$sort = foldl\ (xs\ x \mapsto insert\ x\ xs)\ [\,]$$

Or define a $flip$ function:

$$flip\ f\ x\ y = f\ y\ x \tag{1.58}$$

Then define the insertion sort as: $sort = foldl\ (flip\ insert)\ [\,]$

1.11.2. What's the performance of *concat*? Design a linear time *concat* algorithm.
The performance varies when define *concat* with $foldl$ and $foldr$. $as \mathbin{+\!\!+} bs$ takes $O(m)$ time, where $m = |as|$. When using $foldl$ for $xs_1 \mathbin{+\!\!+} xs_2 \mathbin{+\!\!+} ... \mathbin{+\!\!+} xs_n$, as gets longer and longer, the performance is $O(m_1 + (m_1 + m_2) + ... + \sum_1^{n-1} m_i)$. While using $foldr$ for $xs_1 \mathbin{+\!\!+} (xs_2 \mathbin{+\!\!+} (... \mathbin{+\!\!+} xs_n)...)$, the length of as doesn't keep increasing, but is the length of every xs_i. The performance is $O(m_{n-1} + ... + m_2 + m_1)$, bound to linear time. We can expand its definition:

```
concat [] = []
concat ([]:xss) = concat xss
concat ((x:xs):xss) = x : concat (xs:xss)
```

1.11.3. Define *map* in *foldr*.

$$map\ f = foldr\ (x\ xs \mapsto (f\ x){:}xs)\ [\ ] \qquad (1.59)$$

Answer of exercise 1.12

1.12.1. Implement the linear time filter algorithm through *reverse*.

```
List<K> filterL((K → Bool) p, List<K> xs) {
    List<K> ys = null
    while xs ≠ null {
        if p(xs.key) then ys = cons(xs.key, ys)
        xs = xs.next
    }
    return reverse(ys)
}
```

1.12.2. Enumerate all suffixes of a list.
Method 1: repeatedly drop the head of xs:

$$[xs, tail\ xs, tail\ (tail\ xs), ..., [\ ]]$$

Method 2: use pattern matching to drop the head:

```
tails [] = [[]]
tails xs@(_:ys) =  xs : tails ys
```

Method 3: use $foldr$, start from $[[\ ]]$, add element to right to build the suffixes:

```
tails = foldr f [[]] where
  f x xss@(xs:_) = (x:xs) : xss
```

Answer of exercise 1.13

1.13.1. Design the iota (the Greek letter I) operator for list, below are the use cases:

- $iota(..., n) = [1, 2, 3, ..., n]$;
- $iota(m, n) = [m, m + 1, m + 2, ..., n]$, where $m \leq n$;
- $iota(m, m+a, ..., n) = [m, m+a, m+2a, ..., m+ka]$, where k is the maximum integer satisfying $m + ka \leq n$;
- $iota(m, m, ...) = repeat(m) = [m, m, m, ...]$;
- $iota(m, ...) = [m, m + 1, m + 2, ...]$.

1. Tail recursive way to generate $[1, 2, ..., n]$:

```
iota = iota' [] where
  iota' ns n | n < 1 = ns
             | otherwise = iota' (n : ns) (n - 1)
```

2. With start value: $[m, m + 1, ..., n]$. Change the lower limit from 1 to m:

```
iota m n = iota' [] n where
  iota' ns n | n < m = ns
             | otherwise = iota' (n : ns) (n - 1)
```

3. With step: $[m, m + a, m + 2a, ..., m + ka]$, where k is the maximum integer satisfying $m + ka \leq n$.

```
iota m n a | m ≤ n = m : iota (m + a) n a
           | otherwise = []
```

4. Remove the termination condition to generate open sequence of $n = \infty$: $[m, m+1, ...]$.

```
iota m = m : iota (m + 1)
```

For example, generate the first 10 natural numbers: `take 10 (iota 1)`.

5. Remove $+1$, to implement *repeat*:

```
repeat m = m : repeat m
```

Summarize above, we define the *iterate* function that covers all *iota* use cases:

```
iterate f x = x : iterate f (f x)

iota1 n = take n $ iterate (+1) 1
iota2 m n = takeWhile (≤n) $ iterate (+1) m
iota3 m n a = takeWhile (≤n) $ iterate (+a) m
repeat m = iterate id m
iota5 m = iterate (+1) m
```

1.13.2. Implement the linear time imperative *zip*.

```
[(A, B)] zip([A] xs, [B] ys) {
    [(A, B)] zs = null
    while xs ≠ null and ys ≠ null {
        zs = cons((xs.key, ys.key), zs)
        xs = xs.next
        ys = ys.next
    }
    return reverse(zs)
}
```

1.13.3. Define *zip* with fold (hint: define fold for two lists $foldr2\ f\ z\ xs\ ys$).
Define fold for two lists:

$$
\begin{aligned}
foldr2\ f\ z\ [\ ]\ ys &= z \\
foldr2\ f\ z\ xs\ [\ ] &= z \\
foldr2\ f\ z\ (x{:}xs)\ (y{:}ys) &= foldr2\ f\ (f\ x\ y\ z)\ xs\ ys
\end{aligned}
\tag{1.75}
$$

Then define *zip* with $foldr2$ (Curried form):

$$zip = foldr2\ f\ [\ ], \text{where } f\ x\ y\ xys = (x, y){:}xys$$

1.13.4. Implement $lastAt$ with *zip*.
To index the right k-th element, we drop the first k elements of $xs = [x_0, x_1, ..., x_{n-1}]$ to get $ys = drop\ k\ xs$. Then $zip\ xs\ ys$, the last pair is (x_{n-k-1}, x_{n-1}).

$$lastAt\ k\ xs = (fst \circ last)\ (zip\ xs\ (drop\ k\ xs))$$

1.13.5. Write a program to remove the duplicated elements in a list while maintain the original order. For imperative implementation, the elements should be removed in-place. What is the complexity? How to simplify it with additional data structure?

```
dedup [] = []
dedup (x : xs) = x : dedup (filter (x ≠) xs)
```

We can define *dedup* with $foldr$:

```
dedup  = foldr f [] where
  f x xs = x : filter (x ≠) xs
```

Because we *filter* for every element, the time is bound to $O(n^2)$. We can use set (see chapter 3, 4) to implement *dedup* in $O(n \lg n)$ time:

```
dedup = Set.toList ∘ Set.fromList
```

The corresponding iterative in-place implementation:

```
[K] dedup([K] xs) {
    Int n = length(xs)
    for Int i = 0, i < n, i++ {
        Int j = i + 1
        while j < n {
            if xs[i] == xs[j] {
                swap(xs[j], xs[n - 1])
                n--
            } else {
                j++
            }
        }
    }
    Int m = length(xs) - n
    loop m { pop(xs) }
    return xs
}
```

1.13.6. List can represent decimal non-negative integer. For example 1024 as list is $4 \rightarrow 2 \rightarrow 0 \rightarrow 1$. Generally, $n = d_m...d_2d_1$ can be represented as $d_1 \rightarrow d_2 \rightarrow ... \rightarrow d_m$. Given two numbers a, b in list form. Realize arithmetic operations such as add and subtraction.

Wrap every digit of a decimal natural number into a list, the higher digit is on the right. Convert $n = (d_m...d_2d_1)_{10}$ to list of $[d_1, d_2, ..., d_m]$. For example, 1024 is represented as $[4, 2, 0, 1]$. We can convert such a list back to number, by: $foldr\ (c\ d \mapsto 10d + c)\ 0$. Conversely, below function convert a natural number to list:

```
toList n | n < 10 = [n]
         | otherwise = (n `mod` 10) : toList (n `div` 10)
```

1. Add. 0 is the unit element, i.e., $0 + as = as + 0 = as$, where 0 is represented as empty list $[\ ]$ or list of all zeros $[0]$.

$$
\begin{aligned}
[\ ] + bs &= bs \\
[0] + bs &= bs \\
as + [\ ] &= as \\
as + [0] &= as
\end{aligned}
$$

Add every digit together $d = (a + b) \bmod 10$, and add the carry $c = \lfloor \dfrac{a+b}{10} \rfloor$ to the next digit:

$$(a{:}as) + (b{:}bs) = d{:}(as + bs + [c])$$

Below is the example program:

```
add [] bs = bs
add [0] bs = bs
```

```
add as [] = as
add as [0] = as
add (a:as) (b:bs) = ((a + b) `mod` 10) : add as (add bs [(a + b) `div` 10])
```

2. Minus. $as - 0 = as$, otherwise subtract each digit of as and bs. If the digit $a < b$, we need carry out: $d = 10 + a - b$, and the remaining $as' = as - [1]$.

```
minus as [] = as
minus as [0] = as
minus (a:as) (b:bs) | a < b = (10 + a - b) : minus (minus as [1]) bs
                    | otherwise = (a - b) : minus as bs
```

3. Multiply. For $as \times bs$, we multiply every digit b in bs to as, then times the by 10 and accumulate the result: $cs' = 10 \times cs + (b \times as)$. When compute $b \times as$, if $b = 0$, then it's 0; otherwise, multiply b with the first digit a to get $d = ab \bmod 10$, add the carry $c = \lfloor \frac{ab}{10} \rfloor$ to the further result:

$$b \times (a{:}as) = d : ([c] + (b \times as))$$

Below is the corresponding example program:

```
mul as = foldr (λ b cs → add (mul1 b as) (0:cs)) []

mul1 0 _ = []
mul1 b [] = []
mul1 b (a:as) = (b * a `mod` 10) : add [b * a `div` 10] (mul1 b as)
```

4. Divide (with remainder). First, define how to test zero, how to compare two numbers. A number is zero if it's an empty list or all digits are 0:

```
isZero = all (== 0)
```

To compare as and bs, 0 is less than any none zero number; otherwise, compare from the highest bit to the lowest (EQ: equal, LT: less than, GT: greater than):

```
cmp [] [] = EQ
cmp [] bs = if isZero bs then EQ else LT
cmp as [] = if isZero as then EQ else GT
cmp (a:as) (b:bs) = case cmp as bs of EQ → compare a b
                                      r → r
```

Then we can define equal, and less than test with cmp:

```
eq as bs = EQ == cmp as bs
lt as bs = LT == cmp as bs
```

The brute-force way to implement divide is to repeat minus $bs \neq 0$ from as:

$$\lfloor as/bs \rfloor = \begin{cases} as < bs : & 0 \\ \text{otherwise} : & 1 + \lfloor (as - bs)/bs \rfloor \end{cases}$$

To improve it, consider $as = q \cdot bs + r$, where q is the quotient, $0 \leq r < bs$ is the remainder. Let $as = (a_m...a_2a_1)_{10} = [a_1, a_2, ..., a_m]$, we first use the brute force way to divide a_m by bs to get the quotient q_m and the remainder $r_m = a_m - q_m \cdot bs$. Then add the next digit a_{m-1} to divide $10r_m + a_{m-1}$ by bs to get the quotient q_{m-1} and the remainder r_{m-1}. Finally, put all digits together. This is exactly the $foldr$ process:

```
ldiv as bs | isZero bs = error "divide by 0"
           | otherwise = foldr f ([], []) as where
  f a (qs, rs) = (q:qs, (a:rs) `minus` (q `mul1` bs)) where
    q =  ndec (a:rs)
  ndec as = if as `lt` bs then 0 else 1 + ndec (as `minus` bs)
```

1.13.7. In imperative settings, a circular linked-list is corrupted, that some node points back to previous one, as shown in fig. 1.6. 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).
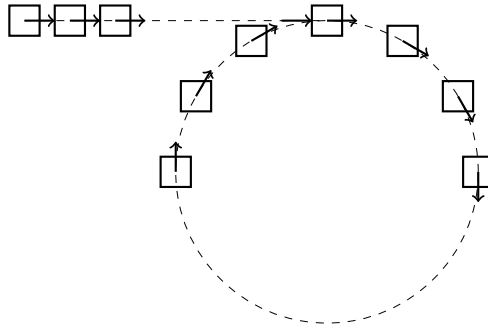
Figure 1.6: List with cycle

If a list has cycle, the traverse will not stop. Consider two runners in a circle, the faster runner will catch, meet, surpass the slower one again and again[11]. We use two pointers to traverse the list: p moves 1 node a time, q moves 2 nodes a time. If they meet at sometime, then there is a circle.

```
Bool hasCycle(List<K> h) {
    var p = h, q = h
    while p ≠ null and q ≠ null {
        p = p.next
        q = q.next
        if q == null then return False
        q = q.next
        if p == q return True
    }
    return False
}
```

Robert W. Folyd calls this method 'tortoise and hare' algorithm. We can further find the node where the circle starts. As shown in fig. 1.7, let OA has k nodes. The loop starts from A. The circle contains n nodes. When p arrives at A, the faster pointer (doubled speed) q will arrives at B. From this time point, the two pointers loop. p is behind q by k nodes. From circle perspective, it equivalent to q is going to catch up p, which is ahead of $n - k$ nodes. It takes time of:

$$t = \frac{n - k}{2v - v} = \frac{n - k}{v}$$

---

[11]Strictly speaking, it's limit to continuous situation. For discrete case like linked-list, they meet if the two speed integers co-prime.
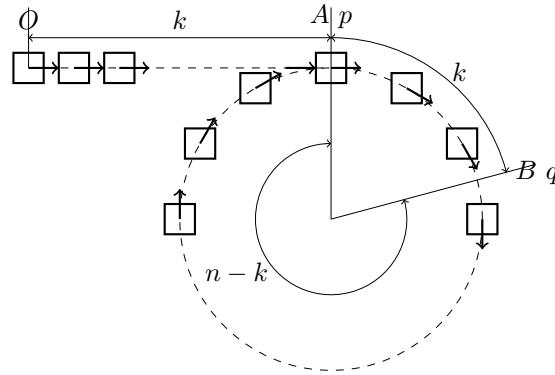
Figure 1.7: Tortoise and hare

When meet, p moves from A by distance of $v\dfrac{n-k}{v} = n - k$. p will arrive at A again after moving forward k nodes. At this time, if reset q to the head O, let it move 1 node a time. Then q will arrive at A after k nodes too, i.e., p and q meet at A. Although we assumed $k < n$, it's true also for $k \geq n$.

Proof. p and q start from the head. When p arrives at A, q will arrive at a place in the circle pass A by $k \bmod n$ nodes. Convert to q will catch up p ahead of $n - (k \bmod n)$ nodes. It takes time:

$$t = \frac{n - (k \bmod n)}{2v - v} = \frac{n - (k \bmod n)}{v}$$

When meet, p moves from A by distance of $v\dfrac{n - (k \bmod n)}{v} = n - (k \bmod n)$. If p further moves forward k, it will arrives at the place of $(k \bmod n) + n - (k \bmod n) = n$, it's exactly at A again. At this time after p and q meet, if reset q to the head O, let it move 1 node a time. Then a will arrive at A after k nodes too, i.e., p and q meet at A again. □

We can implement a program to find A:

```
Optional<List<K>> findCycle(List<K> h) {
    var p = h, q = h
    while p ≠ null and q ≠ null {
        p = p.next
        q = q.next
        if q == null then return Optional.Nothing
        q = q.next
        if p == q {
            q = h
            while p ≠ q {
                p = p.next
                q = q.next
            }
            return Optional.of(p)
        }
    }
    return Optional.Nothing
}
```

Answer of exercise 2.1

2.1.1. Given the in-order and pre-order traverse results, rebuild the tree, and output the post-order traverse result. For example:

- Pre-order: 1, 2, 4, 3, 5, 6;
- In-order: 4, 2, 1, 5, 3, 6;
- Post-order: ?

[4, 2, 5, 6, 3, 1]

2.1.2. Write a program to rebuild the binary tree from the pre-order and in-order traverse lists.

Let P be the pre-order traverse result, I be the in-order result. If $P = I = [\ ]$, then the binary tree is empty $\varnothing$. Otherwise, the pre-order is recursive 'key - left - right', hence the first element m in P is the key of the root. The in-order is recursive 'left - key - right', we can find m in I, which splits I into three parts: $[a_1, a_2, ..., a_{i-1}, m, a_{i+1}, a_{a+2}, ..., a_n]$. Let $I_l = I[1, i), I_r = I[i + 1, n]$, where $[l, r)$ includes l, but excludes r. Either can be empty $[\ ]$. In these three parts I_l, m, I_r, I_l is the in-order traverse result of the left sub-tree, I_r is the in-order result of the right sub-tree. Let $k = |I_l|$ be the size of the left sub-tree, we can split $P[2, n]$ at k to two parts: P_l, P_r, where P_l contains the first k elements. We next recursively rebuild the left sub-tree from (P_l, I_l), rebuild the right sub-tree from (P_r, I_r):

$$\begin{aligned} rebuild\ [\ ]\ [\ ] &= \varnothing \\ rebuild\ (m{:}ps)\ I &= (rebuild\ P_l\ I_l, m, rebuild\ P_r\ I_r) \end{aligned}$$

Where:

$$\begin{cases} (I_l, I_r) &= splitWith\ m\ I \\ (P_l, P_r) &= splitAt\ |I_l|\ ps \end{cases}$$

Below is the example program:

```
rebuild [] _ = Empty
rebuild [c] _ = Node Empty c Empty
rebuild (x:xs) ins = Node (rebuild prl inl) x (rebuild prr inr) where
  (inl, _:inr) = (takeWhile (≠ x) ins, dropWhile (≠x) ins)
  (prl, prr) = splitAt (length inl) xs
```

We can also update the left and right boundary imperatively:

```
Node<T> rebuild([T] pre, [T] ins, Int l = 0, Int r = length(ins)) {
    if l ≥ r then return null
    T c = popFront(pre)
    Int m = find(c, ins)
    var left = rebuild(pre, ins, l, m)
    var right = rebuild(pre, ins, m + 1, r)
    return Node(left, c, right)
}
```

2.1.3. For binary search tree, prove that the in-order traverse always gives ordered list.

Proof. Use proof with absurdity. Suppose there exits (finite sized) binary search tree, the in-order result is not ordered. Among all such trees, select the smallest T. First T can't be $\varnothing$, as the in-order result is $[\ ]$, which is ordered. Second, T can't be singleton $(\varnothing, k, \varnothing)$, as the in-order result is $[k]$, which is ordered. Hence T must be a branch node of (l, k, r). The in-order result is $toList\ l \mathbin{+\!\!+} [x] \mathbin{+\!\!+} toList\ r$. Because T is the smallest tree that the in-order result is not ordered, while l and r are smaller than T, hence both $toList\ l$ and $toList\ r$ are ordered. According to

the binary search tree definition, for every $x \in l, x < k$, and every $y \in r, y > k$. Hence the in-order result $toList \ l + [x] + toList \ r$ is ordered, which conflicts with the assumption, that the in-order result of T is not ordered.

Therefore, for any binary search tree, the in-order result is ordered. $\qquad \square$

2.1.4. What is the complexity of tree sort?

$(n \lg n)$, where n is the number of elements.

2.1.5. Define *toList* with fold.

$$
\begin{aligned}
toList \ &= \ foldt \ id \ (as \ b \ bs \mapsto as + b : bs) \ [\ ] \\
&= \ fold \ (:) \ [\ ]
\end{aligned}
$$

2.1.6. Define *depth t* with fold, to calculate the height of a binary tree.

$$
depth = foldt \ (x \mapsto 1) \ (x \ d \ y \mapsto d + \max \ x \ y) \ 0
$$

Answer of exercise 2.2

2.2.1. How to test whether an element k exists in the tree t?

```
member x (Node l k r) | x == k = True
                      | x < k = member x l
                      | otherwise = member x r
```

2.2.2. 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?

```
data TreeIterator<T> {
    Node<T> node = null

    TreeIterator(Node<T> root) { node = min(root) }

    T get() = node.key

    Bool hasNext() = node ≠ null

    Self next() { if hasNext() then node = succ(node) }
}
```

Although we need find the min/max of the sub-tree or back-track along the parent reference, we take linear time to iterate the tree container. During the traverse process, we visit every node a time (arrive and leave), for example:

```
for var it = TreeIterator(root), it.hasNext(), it = it.next() {
    print(it.get())
}
```

The traverse performance is $O(n)$.

2.2.3. One can traverse the elements inside 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.

```
mapR f a b t = map' t where
    map' Empty = Empty
    map' (Node l k r) | k < a = map' r
                      | a ≤ k && k ≤ b = Node (map' l) (f k) (map' r)
                      | k > b = map' l
```

<div align="center">**Answer of exercise 2.3**</div>

2.3.1. There is a symmetric deletion algorithm. When neither sub-tree is empty, we replace with the maximum of the left sub-tree, then cut it off. Write a program to implement this solution.

```
delete _ Empty = Empty
delete x (Node l k r) | x < k = Node (delete x l) k r
                      | x > k = Node l k (delete x r)
                      | otherwise = del l r
  where
    del Empty r = r
    del l Empty = l
    del l r = let m = max l in Node (delete m l) m r
```

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

```
fromList = (foldr insert Empty) ∘ shuffle
```

2.3.3. How to find the two nodes with the greatest distance in a binary tree?

We first find the maximum distance m, then give the longest path $[s, a, b, ..., e]$. The two ends s and e are the two nodes in question. To define the distance between two nodes, let the connected path (without direction) be $s \to n_1 \to n_2 \to ... \to n_m \to e$, every edge has the length of 1, then the total length from s to e is the distance between them, which is $m + 1$. Define the maximum distance of the empty tree as 0. For singleton leaf $(\varnothing, k, \varnothing)$, as the longest path is $[k]$, the maximum distance is also 0 (from k to k). Consider the branch node (l, k, r), the maximum distance must be one of the three: (1) from the deepest node on the left to the root, then to the deepest node on the right: *depth l + depth r*; (2) the maximum distance of the left sub-tree l; (3) the maximum distance of the right sub-tree r.

```
maxDistance Empty = 0
maxDistance (Node Empty _ Empty) = 0
maxDistance (Node l _ r) = maximum [depth l + depth r,
                             maxDistance l, maxDistance r]
```

Where the definition of *depth* is in Exercise 2.1.6. We can adjust it to find the longest path. For the empty tree, the longest path is [ ], for singleton leaf, the longest path is $[k]$, for branch node (l, k, r), the longest path is the maximum of the three: (1) The reverse of the path from the root to the deepest node on the left, and k, and the path from the root the deepest node on the right; (2) the longest path of the left; (3) the longest path of the right.

```
maxPath Empty = []
maxPath (Node Empty k Empty) = [k]
maxPath (Node l k r) = longest [(reverse depthPath l) ++ k : depthPath r,
                         maxPath l, maxPath r] where
  longest = maximumBy (compare `on` length)
  depthPath = foldt id (λ xs k ys → k : longest [xs, ys]) []
```

This implementation traverses the tree when calculate the depth, then traverses another two rounds for the left and right sub-trees. To avoid duplication, we can bottom-up fill the depth d and the maximum distance m in each node. This can be done through tree map: $Tree\ A \mapsto Tree\ (Int, Int)$ in one traverse:

```
maxDist = extract ∘ mapTr where
  extract Empty = 0
  extract (Node _ (_, m) _) = m
  mapTr Empty = Empty
  mapTr (Node l _ r) = f (mapTr l) (mapTr r)
```

```
  f l r = Node l (1 + max d1 d2, maximum [d1 + d2, m1, m2]) r where
    (d1, m1) = pairof l
    (d2, m2) = pairof r
    pairof Empty = (0, 0)
    pairof (Node _ k _) = k
```

We can further simplify it with fold:

```
maxDist = snd ∘ pair ∘ foldt id g Empty where
  g l _ r = Node l (1 + max d1 d2, maximum [d1 + d2, m1, m2]) r where
    (d1, m1) = pair l
    (d2, m2) = pair r
  pair = (maybe (0, 0) id) ∘ key
```

Answer of exercise 3.1

3.1.1. Implement the insert to scan from left to right.

```
Void insert([T] xs, T x) {
    Int i = 0, n = length(xs)
    append(xs, x)
    while i < n and xs[i] < x { i++ }
    while i < n {
        xs[n] = xs[n - 1]
        n = n - 1
    }
    xs[i] = x
}
```

3.1.2. Define the insert function, and call it from the sort algorithm.

```
Void insert([T] xs, T x) {
    append(xs, x)
    Int i = length(xs) - 1
    while i > 0 and xs[i] < xs[i-1] {
        swap(xs[i], xs[i-1])
        i--
    }
}

[T] sort([T] xs) {
    [T] ys = []
    for x in xs {
        insert(ys, x)
    }
    return ys
}
```

Answer of exercise 3.2

3.2.1. 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 $Next$.

```
[K] reorder([K] xs, [Int] next) {
    Int i = -1
    [Int] ys = []
    while next[i] ≠ -1 {
        append(ys, xs[next[i]])
        i = next[i]
    }
    return ys
}
```

<div align="center">

Answer of exercise 4.1

</div>

4.1.1. For a big address book in lexicographic order, one may want to speed up with two concurrent tasks: one reads from the head; the other from the tail. They meet and stop at some middle point. What does the binary search tree look like? What if split the list into multiple sections to scale up the concurrency?

Because the address entries are in lexicographic order, the two tasks generate two unbalanced tree, head:
$T_h = ((...(\varnothing, k_1, \varnothing), ...), k_m, \varnothing),$
and tail:
$T_t = (\varnothing, k_{m+1}, (\varnothing, k_{m+2}, ...(\varnothing, k_n, \varnothing))...),$
as shown in fig. 4.2(c). With multiple slices, each one generates a tree likes T_h, then combine to a big unbalanced tree. Figure 4.2(b) builds a zig-zag tree from the elements with interleaved order. Each node has an empty sub-tree.

<div align="center">

Answer of exercise 4.2

</div>

4.2.1. Implement the RIGHT-ROTATE.

```
 1: function RIGHT-ROTATE(T, y)
 2:     p ← PARENT(y)
 3:     x ← LEFT(y)                              ▷ assert x ≠ NIL
 4:     a ← LEFT(x)
 5:     b ← RIGHT(x)
 6:     c ← RIGHT(y)
 7:     REPLACE(y, x)                            ▷ replace y with x
 8:     SET-SUBTREES(y, b, c)                    ▷ assign b, c sub-trees of y
 9:     SET-SUBTREES(x, a, y)                    ▷ assign a, y sub-trees of x
10:     if p = NIL then                          ▷ y was the root
11:         T ← x
12:     return T
```

<div align="center">

Answer of exercise 4.3

</div>

4.3.1. Prove the height h of a red-black tree of n nodes is at most $2\lg(n+1)$
We first define the number of black nodes on any path from, but not including, a node x down to a leaf the **black-height** of the node, denoted $bh(x)$. By property 5, all descending paths from the node have the same number of black nodes, hence the black-height is well defined. Particularly, we define the black-height of a red-black tree to be the black-height of its root.

Proof. We first prove that any sub-tree rooted at x contains at least $2^{bh(x)} - 1$ nodes. Apply mathematical induction on the height of x. if $h(x) = 0$, then x is NIL. It contains at least $2^0 - 1 = 0$ nodes. Consider a branch node x, The black-height of its two sub-trees is either $bh(x)$ (the root of the sub-tree is black) or $bh(x) - 1$ (the root of the sub-tree is red). Since the height of the sub-tree of x is less than $h(x)$, from the induction hypothesis, each sub-tree contains at least $2^{bh(x)-1} - 1$ nodes. Hence the sub-tree rooted at x contains at least: $2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nodes.
Let the height of the tree be h, according to property 4, there are not adjacent red nodes. There are more than half black nodes on any path from the root to the leaf. Hence the black-height of the tree is at least $h/2$. We have:

$$n \geq 2^{h/2} - 1 \Rightarrow 2^{h/2} \leq n + 1$$

Take logarithm on both sides: $h/2 \le \lg(n+1)$, i.e., $h \le 2\lg(n+1)$. $\square$

Answer of exercise 4.4

4.4.1. Implement the *insert* without pattern matching, handle the 4 cases separately.

```
Node<T> insert(Node<T> t, T x) = makeBlack(ins(t, x))

Node<T> makeBlack(Node<T> t) {
    t.color = Color.BLACK
    return t
}

Node<T> ins(Node<T> t, T x) {
    if t == null then return Node(null, x, null, Color.BLACK)
    return if x < t.key
            then balance(t.color, ins(t.left, x), t.key, t.right)
            else balance(t.color, t.left, t.key, ins(t.right, x))
}

Node<T> balance(Color c, Node<T> l, T k, Node<T> r) {
    return if c == Color.BLACK {
        if isRed(l) and isRed(l.left) {
            Node(Node(l.left.left, l.left.key, l.left.right, Color.BLACK),
                l.key,
                Node(l.right, k, r, Color.BLACK),
                Color.RED)
        } else if isRed(l) and isRed(l.right) {
            Node(Node(l.left, l.key, l.right.left, Color.BLACK),
                l.right.key,
                Node(l.right.right, k, r, Color.BLACK),
                Color.RED)
        } else if isRed(r) and isRed(r.right) {
            Node(Node(l, k, r.left, Color.BLACK),
                r.key,
                Node(r.left.right, r.right.key, r.right.right, Color.BLACK),
                Color.RED)
        } else if isRed(r) and isRed(r.left) {
            Node(Node(l, k, r.left.left, Color.BLACK),
                r.left.key,
                Node(r.left.right, r.key, r.right, Color.BLACK),
                Color.RED)
        } else {
            Node(l, k, r, c)
        }
    } else {
        Node(l, k, r, c)
    }
}

Bool isRed(Node<T> t) = (t != null and t.color == Color.RED)
```

Answer of exercise 4.5

4.5.1. Implement the 'mark-rebuild' delete algorithm: mark the node as deleted without actually removing it. When the marked nodes exceed 50%, rebuild the tree. We augment each key x with a flag a in the node. The type of the tree is *Tree* $(K, Bool)$. When insert x, call *insert* $(x, True)$ to mark the node active. When delete, flip a to False, then count the active node number and trigger rebuild if exceeds half.

$$delete\ x\ = rebuild \circ del\ x$$

Where:

$$del\ x\ \varnothing\ =\ \varnothing$$

$$del\ x\ (c, l, (k, a), r)\ =\ \begin{cases} x < k : & (c, del\ x\ l, (k, a), r) \\ x > k : & (c, l, (k, a), del\ x\ r) \\ x = k : & (c, l, (k, False), r) \end{cases}$$

If there are more than half nodes are inactive, we convert the tree to a list, filter the active nodes and rebuild the tree.

$$rebuild\ t = \begin{cases} size\ t < \dfrac{1}{2}(cap\ t) : & (fromList \circ toList)\ t \\ otherwise : & t \end{cases}$$

Where *toList* traverse the tree and skip the deleted nodes:

$$toList\ \varnothing\ =\ [\ ]$$

$$toList\ (c, l, (k, a), r)\ =\ \begin{cases} a : & toList\ l + [k] + toList\ r \\ otherwise : & toList\ l + toList\ r \end{cases}$$

To avoid traverse the entire tree every time when count the nodes, we save the tree capacity and size in each node. Extend the type of the tree to $Tree\ (K, Bool, Int, Int)$, and define *node* as:

$$node\ \varnothing\ =\ \varnothing$$

$$node\ c\ l\ (k, a, \_, \_)\ r\ =\ (c, l, (k, a, sz, ca), r)$$

Where:

$$\begin{cases} sz = size\ l + size\ r + (\text{if } a \text{ then } 1 \text{ else } 0) \\ ca = 1 + cap\ l + cap\ r \end{cases}$$

Function *size* and *cap* access the stored size and capacity:

$$size\ \varnothing\ =\ 0 \qquad\qquad cap\ \varnothing\ =\ 0$$

$$size\ (\_, (\_, \_, sz, \_), \_)\ =\ sz \qquad cap\ (\_, (\_, \_, \_, ca), \_)\ =\ ca$$

we replace the node constructor (c, l, k, r) in insert/delete implementation with the *node* function. Below is the example program.

```
data Elem a = Elem a Bool Int Int deriving (Eq)

active (Elem _ a _ _) = a
getElem (Elem x _ _ _) = x

instance Ord a ⇒ Ord (Elem a) where
  compare = compare `on` getElem

insert x = makeBlack ∘ ins (Elem x True 1 1) where
    ins e Empty = Node R Empty e Empty
    ins e (Node color l k r)
        | e < k     = balance color (ins e l) k r
        | otherwise = balance color l k (ins e 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

node c l (Elem k a _ _) r = Node c l (Elem k a sz ca) r where
  sz = size l + size r + if a then 1 else 0
  ca = cap l + cap r + 1

size Empty = 0
size (Node _ _ (Elem _ _ sz _) _) = sz

cap Empty = 0
cap (Node _ _ (Elem _ _ _ ca) _) = ca

delete x = rebuild ∘ del x where
  del _ Empty = Empty
  del x (Node c l e@(Elem k a sz ca) r)
    | x < k = node c (del x l) e r
    | x > k = node c l e (del x r)
    | x == k = node c l (Elem k False 0 0) r

rebuild t | 2 * size t < cap t = (fromList ∘ toList) t
          | otherwise = t

fromList :: (Ord a) ⇒ [a] → RBTree (Elem a)
fromList = foldr insert Empty

toList Empty = []
toList (Node _ l e r) | active e = toList l ++ [getElem e] ++ toList r
                      | otherwise = toList l ++ toList r
```

<div align="center">

Answer of exercise 5.1

</div>

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

Beside verify the height property, we reuse the *toList* function for binary tree, verify if the in-order traverse result is sorted:

```
verifyAVL t = isAVL t && sorted (toList t) where
  sorted [] = True
  sorted xs = and (zipWith (≤) xs (tail xs))
```

<div align="center">

Answer of exercise 6.1

</div>

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

Besides *lookup*, we need handle branch node without value (the blank circle nodes in fig. 6.3) when *insert*. Alternatively, we can add an additional constructor in the algebraic data type (ADT) to replace the Maybe type:

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

<div align="center">

Answer of exercise 6.2

</div>

6.2.1. Implement the *lookup* function for integer tree.

```haskell
import Data.Bits

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

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

lookup :: Key → IntTree a → Maybe a
lookup _ Empty = Nothing
lookup k (Leaf k' v) = if k == k' then Just v else Nothing
lookup k (Branch p m l r) | match k p m = if zero k m then lookup k l
                                                      else lookup k r
                          | otherwise = Nothing

match :: Key → Prefix → Mask → Bool
match k p m = (mask k m) == p

mask :: Int → Mask → Int
mask x m = (x .&. complement (m - 1))

zero :: Int → Mask → Bool
zero x m = x .&. (shiftR m 1) == 0
```

6.2.2. 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?

We first convert an integer trie to assoc-list. The pre-order is the recursive 'middle-
left-right' order. When traverse the empty tree, the result is [ ], For branch node
(l, m, r), let the recursive pre-order lists of the left and right sub-trees be as and bs
respectively. For the middle Maybe value m, if it's Nothing, the result is $as \mathbin{+\mkern-8mu+} bs$;
if it's Just v, then the result is (k, v):$as \mathbin{+\mkern-8mu+} bs$, where k is the corresponding binary
integer (in little endian, 0 for left, 1 for right).

```haskell
toList = go 0 1 where
  go _ _ Empty = []
  go k n (Branch l m r) = case m of
    Nothing → as ++ bs
    (Just v) → (k, v) : as ++ bs
    where
      as = go k       (2 * n) l
      bs = go (n + k) (2 * n) r
```

We start from the root. Let $k = 0$, the depth $d = 0$. If go left, then $k' = 0$, if go
right, then $k' = 1 = 2^d + k = 1 + 0$; For the next level $d = 1$, the corresponding
k for the four nodes are $(00)_2 = 0$, $(10)_2 = 2^1 + 0$, $(01)_2 = 1$, and $(11)_2 = 2^1 + 1$.
Basically, for the node in level d, let $k = (a_d...a_2a_1a_0)_2$, when go left, then $k' = k$,
when go right, then $k' = a_d 2^d + k$. In above implementation, we start with $k = 0$,
$n = 2^0 = 1$ to call *go k n*. Call *go k 2n* when go left, *go $(n + k)$ 2n* when go right.
We get the keys in pre-order through $keys = fst \circ unzip \circ toList$. We can use tail
recursive call to optimize $as \mathbin{+\mkern-8mu+} bs$:

```haskell
toList = go 0 1 [] where
  go _ _ z Empty = z
  go k n z (Branch l m r) = case m of
    Nothing → xs
    (Just v) → (k, v) : xs
    where xs = go k (2 * n) (go (n + k) (2 * n) z r) l
```

Further, we can define generic pre-order fold for integer trie. Different from the *fold* in chapter 2, the keys are computed while folding.

```
foldpre f z = go 0 1 z where
  go _ _ z Empty = z
  go k n z (Branch l m r) = f k m (go k (2 * n) (go (n + k) (2 * n) z r) l)
```

We redefine the *toList* with fold:

```
toList = foldpre f [] where
  f _ Nothing xs = xs
  f k (Just v) xs = (k, v) : xs
```

It's more straightforward to implement the pre-order traverse for integer prefix tree than trie. We needn't compute keys. Below is the generic fold in pre-order:

```
foldpre _ z Empty = z
foldpre f z (Leaf k v) = f k v z
foldpre f z (Branch p m l r) = foldpre f (foldpre f z r) l
```

We can convert a integer tree to assoc-list and get keys with fold:

```
toList = foldpre (λk v xs → (k, v):xs) []
keys = fst ∘ unzip ∘ toList
```

When populate keys of a tree, their binary bits are in ascending order for both little endian binary trie and big endian integer prefix tree. To verify it, we define a function *bitsLE* converting an integer to a list of bits in little endian. Then use it to verify the key ordering of trie:

```
verify kvs = sorted $ map bitsLE $ keys $ fromList kvs where
  sorted [] = True
  sorted xs = and $ zipWith (≤) xs (tail xs)
  bitsLE 0 = []
  bitsLE n = (n `mod` 2) : bitsLE (n `div` 2)
```

Where *kvs* is a list of random key-value pairs. The corresponding verification for big endian integer prefix tree is as below:

```
verify kvs = sorted $ keys $ fromList kvs
```

Answer of exercise 6.3

6.3.1. Eliminate the recursion to implement the prefix tree *lookup* purely with loops.

```
Optional<V> lookup(PrefixTree<K, V> t, K key) {
    if t == null then return Optional.Nothing
    Bool match
    repeat {
        match = False
        for k, tr in t.subtrees {
            if k == key then return Optional.of(tr.value)
            (K prefix, K k1, K k2) = lcp(key, k)
            if prefix ≠ [] and k2 == [] {
                match = True
                key = k1
                t = tr
                break
            }
        }
    } until not match
    return Optional.Nothing
}
```

Answer of exercise 6.4

6.4.1. Implement the auto-completion and predictive text input with trie.
For input prefix ks, we advance to node t in the trie, expand all sub-trees, the take the first n items:

```
import Data.Map (Map)
import qualified Data.Map as Map

startsWith :: Ord k ⇒ [k] → Trie k v → [([k], v)]
startsWith [] (Trie Nothing  ts) = enum ts
startsWith [] (Trie (Just v) ts) = ([], v) : enum ts
startsWith (k:ks) (Trie _ ts) = case Map.lookup k ts of
  Nothing → []
  Just t → map (first (k:)) (startsWith ks t)

enum :: Ord k ⇒ Map k (Trie k v) → [([k], v)]
enum = (concatMap (λ(k, t) →
                    map (first (k:)) (startsWith [] t))) ∘ Map.assocs

get n k t = take n $ startsWith k t
```

Where $first\ f\ (a, b) = (f\ a, b)$, it applies the function f to the first one of a pair. When implement predictive input with trie, we lookup M_{T9} for all characters mapped to a digit, then lookup the trie for candidate words.

```
findT9 [] _ = [[]]
findT9 (d:ds) (Trie _ ts) = concatMap find cts where
  cts = case Map.lookup d mapT9 of
    Nothing → []
    Just cs → Map.assocs $ Map.filterWithKey (λc _ → c `elem` cs) ts
  find (c, t) = map (c:) (findT9 ds t)
```

6.4.2. How to ensure the candidates in lexicographic order in the auto-completion and predictive text input program? What's the performance change accordingly?
From Exercise 6.2.2, if traverse a binary prefix tree in pre-order, the result is in lexicographic order. For multi-way prefix tree, we need traverse sub-trees in lexicographic order. If the sub-trees are managed with self-balancing tree (like the red-black tree or AVL tree), we can do this in linear time (Exercise 2.2.2). If the sub-trees are stored in hash table or assoc-list, then we need $O(n \lg n)$ time to sort them.

Answer of exercise 7.1

7.1.1. Can we use $\leq$ to support duplicated keys in B-Tree?
We can use $\leq$ to allow duplicated keys in B-tree: all left ones are $\leq x$, while $x \leq$ all right ones. However, we need additional logic to handle duplicated keys when lookup and delete a key. Typically, we constraint the key to be unique, while a key can map to multiple values: $k \mapsto [v_1, v_2, ...]$, called *multi-map*.

7.1.2. For the 'split then insert' algorithm, eliminate the recursion with loops.

```
BTree<K, deg> insertNonfull(BTree<K, deg> tr, K key) {
    var root = t
    while not is_leaf(t) {
        Int i = length(t.keys)
        while i > 0 and key < t.keys[i-1] {
            i = i - 1
        }
        if full(d, t.subTrees[i]) {
            split(d, t, i)
            if key > t.keys[i] then i = i + 1
```

```
        }
        t = t.subTrees[i]
    }
    orderedInsert(t.keys, key)
    return root
```

7.1.3. 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?

```
void orderedInsert([K] xs, K x) {
    append(xs, x)
    Int p = binarySearch(xs, x)
    for Int i = length(lst), i > p, i = i - 1 {
        xs[i] = xs[i-1]
    }
    xs[p] = x
}

Int binarySearch([K] xs, K x) {
    Int l = 0, u = length(xs)
    while l < u {
        Int m = (l + u) / 2
        if xs[m] == x {
            return m
        } else if xs[m] < x {
            l = m + 1
        } else {
            u = m
        }
    }
    return l
}
```

The performance is still linear. Although the binary search speed up to $O(\lg n)$, the insertion takes $O(n)$ time to shift elements.

Answer of exercise 7.2

7.2.1. Improve the imperative lookup with binary search among keys.

```
Optional<(BTree<K, deg>, Int)> lookup(BTree<K, deg> tr, K key) {
    loop {
        Int l = 0, u = length(tr.keys)
        while l < u {
            Int m = (l + u) / 2
            if key == tr.keys[m] {
                return Optional.of((tr, m))
            } else if t.keys[m] < key {
                l = m + 1
            } else {
                u = m
            }
        }
        if isLeaf(tr) {
            return Optional.Nothing
        } else {
            tr = tr.subTrees[l]
        }
    }
}
```

Answer of exercise 7.3

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.
We first define the *min* function to get the minimum key:

```
min' (BTree ks []) = head ks
min' (BTree _  ts) = min' $ head ts
```

When partition the tree with x, we need use $\leq$, but not $<$, because x may be the last one on the left. We abstract the partition predicate as a parameter:

```
partitionWith p (BTree ks ts) = (l, t, r) where
  l = (ks1, ts1)
  r = (ks2, ts2)
  (ks1, ks2) = L.span p ks
  (ts1, (t:ts2)) = L.splitAt (length ks1) ts
```

We can next implement the delete with *min*:

```
delete' x (d, t) = fixRoot (d, del x t) where
    del x (BTree ks []) = BTree (L.delete x ks) []
    del x t = if (Just x) == (listToMaybe $ reverse ks') then
                  let k' = min' t' in
                      balance d ((init ks') ++ [k'], ts') (del k' t') r
              else balance d l (del x t') r
      where
        (l@(ks', ts'), t', r) = partitionWith (≤ x) t
```

7.3.2. Define the *delete* function for the 'paired list' implementation.

```
delete x (d, t) = fixRoot (d, del x t) where
  del _ Empty = Empty
  del x t = if (Just x) == fmap fst (listToMaybe r) then
                case t' of
                  Empty → balance d l Empty (tail r)
                  _     → let k' = max' t' in
                      balance d l (del k' t') ((k', snd $ head r):(tail r))
            else balance d l (del x t') r
    where
      (l, t', r) = partition (< x) t

max' t@(BTree _ _ []) = max' (stepL t)
max' (BTree _ _ [(k, Empty)]) = k
max' (BTree _ _ [(k, t)]) = max' t
max' t = max' (stepR t)
```

We need add additional logic in *balance* to fix the too low cases after delete:

```
balance :: Int → [(a, BTree a)] → BTree a → [(a, BTree a)] → BTree a
balance d l t r | full d t = fixFull
                | low  d t = fixLow l t r
                | otherwise = BTree l t r
  where
    fixFull = let (t1, k, t2) = split d t in BTree l t1 ((k, t2):r)
    fixLow ((k', t'):l) t r = balance d l (unsplit t' k' t) r
    fixLow l t ((k', t'):r) = balance d l (unsplit t k' t') r
    fixLow l t r = t        —— l == r == []
```

Where *unsplit* is the reverse of *split*:

```
unsplit t1 k t2@(BTree (_:_) _ _) = unsplit t1 k (stepL t2)
unsplit t1@(BTree _ _ (_:_)) k t2 = unsplit (stepR t1) k t2
```

<div align="center">

Answer of exercise 8.1

</div>

8.1.1. No, it is not correct. The sub-array $[a_2, a_3, ..., a_n]$ can't map back to binary heap. It's insufficient to only apply HEAPIFY from a_2, we need run BUILD-HEAP to rebuild the heap.

8.1.2. For the same reason, it does not work.

<div align="center">

Answer of exercise 8.2

</div>

8.2.1. Implement leftist heap and skew heap imperatively.
We add a parent reference in the node definition for easy back-tracking.

```
data Node<T> {
    T value
    Int rank = 1
    Node<T> left = null, right = null, parent = null
}
```

When merge two leftist heaps, we firstly top-down merge along the right sub-tree, then bottom-up update the rank along the parent reference. Swap the sub-trees if the left has the smaller rank. To simplify the empty tree handling, we use a sentinel node.

```
Node<T> merge(Node<T> a, Node<T> b) {
    var h = Node(null)  // the sentinel node
    while a ≠ null and b ≠ null {
        if b.value < a.value then swap(a, b)
        var c = Node(a.value, parent = h, left = a.left)
        h.right = c
        h = c
        a = a.right
    }
    h.right = if a ≠ null then a else b
    while h.parent ≠ null {
        if rank(h.left) < rank(h.right) then swap(h.left, h.right)
        h.rank = 1 + rank(h.right)
        h = h.parent
    }
    h = h.right
    if h ≠ null then h.parent = null
    return h
}

Int rank(Node<T> x) = if x ≠ null then x.rank else 0

Node<T> insert(Node<T> h, T x) = merge(Node(x), h)

T top(Node<T> h) = h.value

Node<T> pop(Node<T> h) = merge(h.left, h.right)
```

It's simpler to merge skew heaps, as we needn't update the rank, the parent, or back-track.

```
data Node<T> {
    T value
    Node<T> left = null, right = null
}
```

Below is the merge function, the others are same as the leftist heap.

```
Node<T> merge(Node<T> a, Node<T> b) {
    var h = Node(None)
```

```
    var root = h
    while a ≠ null and b ≠ null {
        if b.value < a.value then swap(a, b)
        var c = Node(a.value, left = null, right = a.left)
        h.left = c
        h = c
        a = a.right
    }
    h.left = if a ≠ null then a else b
    root = root.left
    return root
}
```

8.2.2. Define fold for heap.

$$
\begin{array}{rcl}
fold\ f\ z\ \varnothing &=& z \\
fold\ f\ z\ H &=& fold\ f\ (f\ (top\ H)\ z)\ (pop\ H)
\end{array}
$$

Answer of exercise 9.1

9.1.1. We should use link but not append. Appending is linear to the length of the list, while linking is constant time.

9.1.2. Implement the in-place selection sort.

```
Void sort([K] xs) {
    var n = length(xs)
    for var i = 0 to n - 1 {
        var m = i
        for Int j = i + 1 to n - 1 {
            if xs[j] < xs[m] then m = j
        }
        swap(xs[i], xs[m])
    }
}
```

Answer of exercise 9.2

9.2.1. Implement the recursive tournament tree sort in ascending order.
We can realize the ascending sort by replacing the max and $-\infty$ with the min and ∞. Further, we can abstract them as two parameters:

```
minBy p a b = if p a b then a else b

merge p t1 t2 = Br t1 (minBy p (key t1) (key t2)) t2

fromListWith p xs = build $ map wrap xs where
  build [] = Empty
  build [t] = t
  build ts = build $ pair ts
  pair (t1:t2:ts) = (merge p t1 t2) : pair ts
  pair ts = ts

popWith p inf = delMin where
  delMin (Br Empty _ Empty) = Br Empty inf Empty
  delMin (Br l k r) | k == key l = let l' = delMin l in
                        Br l' (minBy p (key l') (key r)) r
                    | k == key r = let r' = delMin r in
                        Br l (minBy p (key l) (key r')) r'

toListWith p inf = flat where
```

```
    flat Empty = []
    flat t | inf == key t = []
           | otherwise = (top t) : (flat $ popWith p inf t)

sortBy p inf xs = toListWith p inf $ fromListWith p xs where
```

sortBy $(<)\ \infty$ defines the ascending sort, and *sortBy* $(>)\ -\infty$ defines the descending sort.

9.2.2. How to handle duplicated elements with the tournament tree? is tournament tree sort stable?

From Exercise 9.2.1, we can handle the duplicated elements with *sortBy* $(\leq)\ \infty$ (ascending sort) for example. The tournament tree sort is not stable.

9.2.3. Compare the tournament tree sort and binary search tree sort in terms of space and time performance.

They are both bound to $O(n \lg n)$ time, and $O(n)$ space. The difference is, the binary search tree does not change after build (unless insert, delete), while the tournament tree changes to a tree with n nodes of infinity.

9.2.4. Compare heap sort and tournament tree sort in terms of space and time performance.

They are both bound to $O(n \lg n)$ time, and $O(n)$ space. The difference is, the heap becomes empty after sort complete, while the tournament tree still occupies $O(n)$ space.

Answer of exercise 10.1

10.1.1. Write a program to generate Pascal's triangle.

```
pascal = gen [1] where
  gen cs (x:y:xs) = gen ((x + y) : cs) (y:xs)
  gen cs _ = 1 : cs
```

Given any row in Pascal's triangle, this function generates the next row. We can build the first n rows of pascal triangle as: `take n (iterate pascal [1])`.

10.1.2. Prove that the i-th row in tree B_n has $\binom{n}{i}$ nodes.

Proof. Use induction. There is only a node (the root) in B_0. Assume every row in B_n is a list of binomial numbers. Tree B_{n+1} is composed from two B_n trees. The 0-th row contains the root: $1 = \binom{n+1}{0}$. The i-th row has two parts: one from the $(i-1)$-th row of the left most sub-tree B_n, the other from the i-th row of the other B_n tree. In total:

$$
\begin{aligned}
\binom{n}{i-1} + \binom{n}{i} &= \frac{n!}{(i-1)!(n-i+1)!} + \frac{n!}{i!(n-i)!} \\
&= \frac{n![i + (n-i+1)]}{i!(n-i+1)!} \\
&= \frac{(n+1)!}{i!(n-i+1)!} \\
&= \binom{n+1}{i}
\end{aligned}
$$

$\square$

10.1.3. Prove there are 2^n elements in a B_n tree.

Proof. From the previous exercise, sum all rows of B_n tree:

$$\begin{aligned}
& \binom{n}{0} + \binom{n}{1} + ... + \binom{n}{n} && \text{Sum rows} \\
=\ & (1 + 1)^n && \text{Let } a = b = 1 \text{ in } (a + b)^n \\
=\ & 2^n
\end{aligned}$$

$\square$

10.1.4. Use a container to store sub-trees, how to implement link? How to ensure it is in constant time?

If store all sub-trees in an array, we need linear time to insert a new tree ahead:

```
1: function LINK'(T₁, T₂)
2:     if KEY(T₂) < KEY(T₁) then
3:         Exchange T₁ ↔ T₂
4:     PARENT(T₂) ← T₁
5:     INSERT(SUB-TREES(T₁), 1, T₂)
6:     RANK(T₁) ← RANK(T₂) + 1
7:     return T₁
```

We can store the sub-trees in reversed order, it needs constant time to append the new tree to the tail.

Answer of exercise 10.2

Why is DECREASE bound to amortized $O(1)$ time?

Define the potential function as:

$$\Phi(H) = t(H) + 2m(H)$$

Where $t(H)$ is the number of trees in the heap, and $m(H)$ is the number of the nodes being marked. As we mark, then later cut and clear the flag, its coefficient is 2. DECREASE takes $O(1)$ time to cut x off, then recursively call CASCADE-CUT. Assume it's recursively called c times. Each time takes $O(1)$ time to call CUT, then continue recursion. Hence the total cost of DECREASE is $O(c)$.

For the potential change, let H be the heap before we call DECREASE, every recursive CASCADE-CUT cuts a marked node off, then clear the flag (except for the last call). After that, there are $t(H) + c$ trees, including the original $t(H)$ trees, the cut and added back $c - 1$ trees, and the tree with x as the root. There are at most $m(H) - c + 2$ marked nodes, including the original $m(H)$ nodes, minus the $c - 1$ nodes being cleared in the CASCADE-CUT call. The last call may mark another node. The potential changes at most:

$$t(H) + c + 2(m(H) - c + 2) - [t(H) + 2m(H)] = 4 - c$$

Hence the amortized cost is at most $O(c) + 4 - c = O(1)$.

Answer of exercise 10.3

10.3.1. If continuously insert n elements then followed with a pop. The performance overhead is big when n is a large number (although the amortized performance is $O(\lg n)$). How to mitigate such worst case?

Set a threshold m to the number of sub-trees. When insert, check whether it exceeds m, if yes, then perform a pop then add the element back.

```
MAX_SUBTREES = 16

Node<K> insert(Node<K> h, K x) {
    if h ≠ null and length(h.subTrees) > MAX_SUBTREES {
        h = insert(pop(h), top(h))
    }
    return merge(h, Node(x))
}
```

10.3.2. Implement *delete* for the pairing heap.
Add a parent reference to the node definition:

```
data Node<K> {
    K key
    Node<K> parent = null
    [Node<K>] subTrees = []

    Node<K>(K k) { key = k }
}
```

When delete x, first lookup the heap h to find the sub-tree t rooted at x. If t is the root of h, then merely do a *pop*; otherwise, get the parent p of t, remove t from its sub-trees. Then apply *pop*, and finally merge $pop(t)$ back with h.

```
Node<K> delete(Node<K> h, K x) {
    var tr = lookuptr(h, x)
    if tr == null then return h
    if tr == h then return pop(h)
    tr.parent.subtrees.remove(tr)
    tr.parent = null
    return merge(pop(tr), h)
}

Node<K> lookuptr(Node<K> h, K x) {
    if h.key == x then return h
    for var t in h.subtrees {
        var tr = lookuptr(t, x)
        if tr ≠ null then return tr
    }
    return null
}
```

The recursive lookup takes $O(n)$ time, where n is the number of elements in the heap. Then it takes $O(m)$ time to remove t from m sub-trees. The total performance is $O(n)$.

10.3.3. Implement DECREASE-KEY for the pairing heap.
If decrease the key of the root of h, we directly update it to x; otherwise, get the parent of tr, then cut tr off from the sub-trees; update the key of tr to x, then merge tr back to h:

```
Node<K> decreaseKey(Node<K> h, Node<K> tr, K x) {
    if tr == null or tr.key < x then return h
    tr.key = x
    if tr == h then return h
    tr.parent.subtrees.remove(tr)   // O(m), where m = length(subtrees)
    tr.parent = null
    return merge(tr, h)
}
```

When use, we first call `lookuptr(h, y)` to find the node, then update from y to x, i.e., `deceaseKey(h, lookuptr(h, y), x)`. The performance is as same as delete, which is $O(n)$.

<div align="center">

Answer of exercise 11.1

</div>

11.1.1. The circular buffer is allocated with a predefined size. We can use two references, head and tail instead. How to determine if a circular buffer queue is full or empty? (the head can be either ahead of tail or behind it.)

Although we can deal with different cases that the head is before/behind the tail, as shown in fig. 11.4, let us seek for some simple and unified solution. Consider an array with both sides open and are extending infinitely. Let the head index be h, the tail be t, the size of the buffer be s. The range $[h, t)$ (left close, right open) is occupied with elements. The empty/full testing is as below:

$$\begin{cases} empty(h, t): & h = t \\ full(h, t): & t - h = s \end{cases}$$

The circular buffer is essentially modular arithmetic on top of this. Write $[n]_s = n \bmod s$. Apply to above full test: $[t]_s - [h]_s = [s]_s = 0$, which gives $[t]_s = [h]_s$. It is exactly the empty test condition. It means we can't differentiate empty with full only with the modular result. We need either add a flag (indicate the order between h and t), or use the original index (of the infinite long segment) without applying mod for the empty/full test. Consider the integer number has limited byte size, we can mod the index with a big number p ($p > s$) which is co-prime with s:

$$\begin{cases} empty(h, t): & [h]_p = [t]_p \\ full(h, t): & [t - h]_p = s \end{cases}$$

Below is the example program:

```
Int P_LIMIT = 104743 // the 10000th prime
Bool empty(Queue<K> q) = (q.h == q.t)
Bool full(Queue<K> q) = (q.s == (q.t - q.h) mod P_LIMIT)

void enqueue(Queue<K> q, K x) {
    if not full(q) {
        q.t = (q.t + 1) mod P_LIMIT
        q.buf[q.t mod q.s] = x
    }
}

Optional<K> dequeue(Queue<K> q) {
    Optional<K> x = Optional.Nothing
    if not empty(q) {
        x = Optional.of(q.buf[q.h mod q.s])
        q.h = (q.h + 1) mod P_LIMIT
    }
    return x
}
```

<div align="center">

Answer of exercise 11.2

</div>

11.2.1. Why need balance check and adjustment after push?
Consider the case, first *push a* ($[\,],[\,]$), then *pop*.

11.2.2. Do the amortized analysis for the paired-list queue.
We use the banker's accounting method. Each element in the rear list r has 1 credit. When push to rear, we take a step to add, and increase 1 credit. The amortized cost is $O(2)$. When pop, if it doesn't cause list reverse, we take a step

to remove an element without decreasing the credit. The amortized cost is $O(1)$. If causes list reverse, we take m steps to reverse, and take a step to remove an element, where m is the length of r. We also use m credits in r. The amortized cost is $O(m + 1 - m) = O(1)$.

11.2.3. Implement the paired-array queue.

```
1: function PUSH(Q, x)
2:     APPEND(FRONT(Q), x)

3: function POP(Q)
4:     if REAR(Q) = [ ] then
5:         REAR(Q) ← REVERSE(FRONT(Q))
6:         FRONT(Q) ← [ ]
7:     n ← LENGTH(REAR(Q))
8:     x ← REAR(Q)[n]
9:     LENGTH(REAR(Q)) ← n − 1
10:    return x
```

Answer of exercise 11.3

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*?
The *abort* function is only called during pop. When $n = 0$, the rotate just finished, it is about to transform the state from $(S_c, 0, (x{:}a), f')$ to (S_f, a). While x being linked previously is the one to be popped, hence, we need remove x and return a as the result.

11.3.2. Implement the real-time queue with paired arrays. We can't copy the array when start rotation, or the performance will downgrade to linear time. Please implement 'lazy' copy, i.e., copy an element per step.
Assume we can get the length of an array in constant time. We push element to the tail of array f, and pop from the tail of array r. When the two arrays are not balanced, start a state machine to compute $acc = reverse(f) + r$ step by step. If $f \neq [\ ]$, we extract the tail element and append to the tail of acc. After reversed f, we copy the elements from the left of r, and append to the tail of acc one by one, i.e., $append(acc, r[i])$, where $i = 0, 1, .., |r| - 1$. While rotating in steps, one can still pop from the tail of r, the rotation completes when i exceeds $|r|$.

```
data State<K> {
    [K] acc, front, rear
    Int idx

    State([K] f, [K] r) {
        acc = [], front = f, rear = r
        idx = 0
    }

    // compute reverse(f) ++ r step by step
    Self step() {
        if front ≠ [] then acc.append(front.popLast())  // reversing
        if s.front == [] and idx < length(rear) {    // concatenating
            acc.append(rear[idx])
            idx = idx + 1
        }
    }

    Bool done() = (front == [] and length(rear) < idx)
}
```

```
data RealtimeQueue<K> {
    [K] front = []
    [K] rear = []
    State<K> state = null

    Bool isEmpty() = (front == [] and rear == [])

    Self push(K x) {
        front.append(x)
        balance()
    }

    K pop() {
        x = rear.popLast()
        balance()
        return x
    }

    Void balance() {
        if state == null and length(rear) < length(front) {
            state = State(front, rear).step()
            front = []
        }
        if state ≠ null and state.step().done() {
            rear = state.acc
            state = null
        }
    }
}
```

Answer of exercise 12.1

12.1.1. How to handle the out of bound exception?

We can use *Maybe* type to handle the out of bound cases. If the index $i < 0$, return *Nothing*; if the index exceeds, then it eventually converts to the empty forest after recursion. We return *Nothing* in this case too.

```
getAt [] _ = Nothing
getAt (t:ts) i | i < 0 = Nothing
               | i < size t = lookupTree i t
               | otherwise = getAt ts (i - size t)
  where
    lookupTree 0 (Leaf x) = Just x
    lookupTree i (Node sz t1 t2) = if i < sz `div` 2 then lookupTree i t1
                                   else lookupTree (i - sz `div` 2) t2
```

Answer of exercise 12.2

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.
We skip the index out of bound exception:

```
getAt (Zero:ts) i = getAt ts i
getAt (One t:ts) i = if i < size t then lookupTree i t
                     else getAt ts (i - size t)
  where
    lookupTree 0 (Leaf x) = x
    lookupTree i (Node sz t1 t2) = if i < sz `div` 2 then lookupTree i t1
                                   else lookupTree (i - sz `div` 2) t2
```

12.2.2. Analyze the amortized performance of delete.

Consider the reverse process of insert, delete from a sequence with $n = 2^m$ elements repeatedly till it becomes empty. The analysis is symmetric to insert, and it gives the amortized constant time.

12.2.3. 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.

```
data List<K> {
    Int size = 0
    [[K]] trees = [[]]
}

Int nbits(Int n) {
    Int i = 0
    while n ≠ 0 {
        i = i + 1
        n = n / 2
    }
    return i
}

List<K> insert(List<K> xs, K x) {
    var i = nbits(xs.size xor (1 + xs.size))    // locate the bit flip to 1
    if i ≥ length(xs.trees) then xs.trees.append([])
    xs.trees[i] = [x]
    for Int j = 0, j < i, j++ {
        xs.trees[i] = xs.trees[i] ++ xs.trees[j]
        xs.trees[j] = []
    }
    xs.size = xs.size + 1
    return xs
}

Optional<K> get(List<K> xs, Int i) {
    for t in xs.trees {
        Int size = length(t)
        if i < size then return Optional.of(t[i])
        i = i - size
    }
    return Optional.Nothing
}
```

Answer of exercise 12.3

12.3.1. Analyze the amortized performance for paired-array delete.

Define the potential of the paired-array sequence as the difference of the array lengths: $\Phi(s) = |r| - |f| = n - m$, where $m = |f|$ and $n = |r|$. When delete from the head, if $f \neq [\,]$, then takes $O(1)$ time to remove the last element of f. If $f = [\,]$, then takes $O(n)$ time to halve r, reverse and replace as f', and then uses another $O(1)$ time to remove the last element of f'. The amortized cost is:

$$
\begin{aligned}
c &= n + 1 + \Phi(s') - \Phi(s) \\
&= n + 1 + (|r'| - |f'|) - (|r| - |f|) \\
&= n + 1 + (n - \lceil \frac{n}{2} \rceil) - (\lceil \frac{n}{2} \rceil - 1) - (n - 0) \\
&= 1
\end{aligned}
$$

Hence the amortized time is $O(1)$ when delete from head, symmetrically, the amortized time is $O(1)$ when delete from tail too.

<div align="center">**Answer of exercise 12.4**</div>

12.4.1. Eliminate recursion, implement insert with loop.
Let $\text{MID}(T) = t$ access the middle part of tree $T = (f, t, r)$.

```
 1: function INSERT(x, T)
 2:     n = (x)
 3:     ⊥ ← p ← ([ ], T, [ ])
 4:     while |FRONT(T)| ≥ 3 do
 5:         f ← FRONT(T)
 6:         n ← (f[2], f[3], ...)
 7:         FRONT(T) ← [n, f[1]]
 8:         p ← T
 9:         T ← MID(T)
10:     if T = NIL then
11:         T ← ([n], NIL, [ ])
12:     else if |FRONT(T)| = 1 and REAR(T) = [ ] then
13:         REAR(T) ← FRONT(T)
14:         FRONT(T) ← [n]
15:     else
16:         INSERT(FRONT(T), n)
17:     MID(p) ← T
18:     T ← MID(⊥), MID(⊥) ← NIL
19:     return T
```

We wrap x in a leaf (x). If there are more than 3 elements in f, we go top-down along the middle part. We extract the elements except the first one in f out, wrap them in a node n (depth $+ 1$), then insert n to the middle. We form n and the remaining in f as the new f finger. At the end of traverse, we either reach an empty tree, or a tree can hold more elements in f. For empty tree case, we create a new leaf node; otherwise, we insert n to the head of f. We return the root T. To simplify implementation, we create a special $\perp$ node as the parent of the root.

<div align="center">**Answer of exercise 12.5**</div>

12.5.1. Eliminate recursion, implement extract in loops.
We borrow node from the middle when f is empty. However, the tree may not well formed, e.g., both f and the middle are empty. It is caused by splitting.
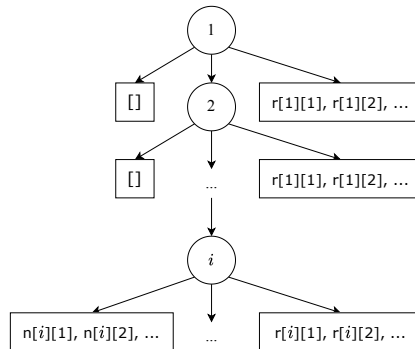

Figure 12.9: The f isn't empty at level i.

To extract the first element, we need a top-bottom pass, locate a sub-tree, either f isn't empty, or both f and the middle are empty as shown in fig. 12.9. For the former, we extract the first node from f; for the latter, we swap f and r, convert

it to the former case. If the node extracted from f isn't a leaf, we need go on extracting. We back track along the parent, till extract a leaf and reach to the root, as shown in fig. 12.10.
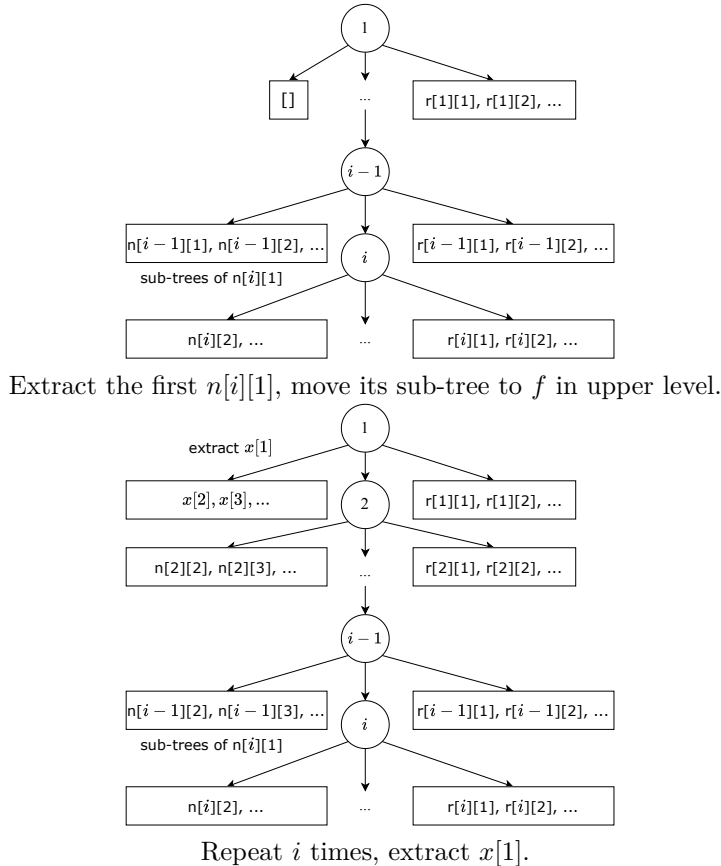

Extract the first $n[i][1]$, move its sub-tree to f in upper level.

Repeat i times, extract $x[1]$.

Figure 12.10: Bottom up back track to extract a leaf.

Assume the tree isn't empty, we implement extract as below:

1: **function** EXTRACT(T)
2:     $\perp \leftarrow ([], T, [])$
3:     **while** FRONT(T) = [ ] and MID(T) $\neq$ NIL **do**
4:         $T \leftarrow$ MID(T)
5:     **if** FRONT(T) = [ ] and REAR(T) $\neq$ [ ] **then**
6:         EXCHANGE FRONT(T) $\leftrightarrow$ REAR(T)
7:     $f \leftarrow$ FRONT(T), $r \leftarrow$ REAR(T)
8:     $n \leftarrow (f[1], f[2], ...)$ $\triangleright$ n is 2-3 tree
9:     **repeat**
10:         FRONT(T) $\leftarrow [n_2, n_3, ..]$
11:         $n \leftarrow n_1$
12:         $T \leftarrow$ PARENT(T)
13:         **if** MID(T) becomes empty **then**
14:             MID(T) $\leftarrow$ NIL
15:     **until** n is leaf
16:     **return** (ELEM(n), MID($\perp$))

Where function ELEM(n) accesses the element in sub-tree n. We need change the

way to access the first/last element of finger tree. If the finger is empty, and the
middle isn't empty, we need search along the middle.

1: **function** First-Leaf(T)
2:     **while** Front(T) = [ ] and Mid(T) ≠ NIL **do**
3:         $T \leftarrow$ Mid(T)
4:     **if** Front(T) = [ ] and Rear(T) ≠ [ ] **then**
5:         $n \leftarrow$ Rear(T)[1]
6:     **else**
7:         $n \leftarrow$ Front(T)[1]
8:     **while** n is NOT leaf **do**
9:         $n \leftarrow n_1$
10:     **return** n

11: **function** First(T)
12:     **return** Elem(First-Leaf(T))

In the second loop, if the node is not a leaf, we need traverse along the first sub-tree.
The method to access the last element is symmetric.

Answer of exercise 12.6

12.6.1. For random access, how to handle the empty tree $\varnothing$ and out of bound cases?
We check boundaries during random access, for example:

$$\varnothing[i] = \textit{Nothing}$$
$$T[i] = \begin{cases} i < 0 \text{ or } i \geq \textit{size } T : & \textit{Nothing} \\ \text{otherwise} : & \text{...} \end{cases}$$

12.6.2. Implement *cut i S*, split sequence S at position i.
We give an implementation based on the tree definition in appendix. First do
boundary check, if $0 \leq i < \textit{size } s$, we next call *cutTree i S* to split the tree:

```
cut :: Int → Seq a → (Seq a, Maybe a, Seq a)
cut i (Seq xs) | i < 0 = (Seq Empty, Nothing, Seq xs)
               | i < size xs = case cutTree i xs of
                 (a, Just (Place _ (Elem x)), b) → (Seq a, Just x, Seq b)
                 (a, Nothing, b) → (Seq a, Nothing, Seq b)
               | otherwise = (Seq xs, Nothing, Seq Empty)
```

cutTree splits the tree in three parts: left, middle, and right. We wrap the middle
in *Maybe* type to handle the not found case; when found, the result is a pair of
position i' and node a, wrapped in the type of *Place*. if i points to finger f or r,
we call *cutList* to further split, then build the result; if i points to the middle, we
recursively cut the middle to obtain a place *Place i' a*, then cut the 2-3 tree a at
position i':

```
cutTree :: (Sized a) ⇒ Int → Tree a → (Tree a, Maybe (Place a), Tree a)
cutTree _ Empty = (Empty, Nothing, Empty)
cutTree i (Lf a) | i < size a = (Empty, Just (Place i a), Empty)
                 | otherwise = (Lf a, Nothing, Empty)
cutTree i (Br s f m r)
  | i < sf = case cutList i f of
             (xs, x, ys) → (Empty <<< xs, x, tree ys m r)
  | i < sm = case cutTree (i - sf) m of
             (t1, Just (Place i' a), t2) →
               let (xs, x, ys) = cutNode i' a
```

```
                        in (tree f t1 xs, x, tree ys t2 r)
  | i < s  = case cutList (i - sm) r of
                (xs, x, ys) → (tree f m xs, x, ys >>> Empty)
  where
    sf = sum $ map size f
    sm = sf + size m
```

Where *tree f m r* builds a finger tree, and simplifies the result:

```
tree as Empty [] = as >>> Empty
tree [] Empty bs = Empty <<< bs
tree [] m r = Br (size m + sum (map size r)) (nodesOf f) m' r
    where (f, m') = uncons m
tree f m [] = Br (size m + sum (map size f)) f m' (nodesOf r)
    where (m', r) = unsnoc m
tree f m r = Br (size m + sum (map size f) + sum (map size r)) f m r
```

We implement the finger cut and 2-3 tree cut as below:

```
cutList :: (Sized a) ⇒ Int → [a] → ([a], Maybe (Place a), [a])
cutList _ [] = ([], Nothing, [])
cutList i (x:xs) | i < sx = ([], Just (Place i x), xs)
                 | otherwise = let (xs', y, ys) = cutList (i - sx) xs
                               in (x:xs', y, ys)
  where sx = size x

cutNode :: (Sized a) ⇒ Int → Node a → ([a], Maybe (Place a), [a])
cutNode i (Tr2 _ a b) | i < sa = ([], Just (Place i a), [b])
                      | otherwise = ([a], Just (Place (i - sa) b), [])
  where sa = size a
cutNode i (Tr3 _ a b c) | i < sa = ([], Just (Place i a), [b, c])
                        | i < sab = ([a], Just (Place (i - sa) b), [c])
                        | otherwise = ([a, b], Just (Place (i - sab) c), [])
  where sa = size a
        sab = sa + size b
```

With *cut* defined, we can update or delete any element at a given position, move to front (MTF), they all bound to $O(\lg n)$ time.

```
setAt s i x = case cut i s of
  (_, Nothing, _) → s
  (xs, Just y, ys) → xs ++ (x <| ys)

extractAt s i = case cut i s of (xs, Just y, ys) → (y, xs ++ ys)

moveToFront i s = if i < 0 || i ≥ size s then s
                  else let (a, s') = extractAt s i in a <| s'
```

Answer of exercise 13.1

13.1.1. Optimize the basic quick sort definition for the singleton list case.
Add a case:

```
sort [x] = [x]
```

Answer of exercise 13.2

13.2.1. The pairwise merge is faster. This problem is essentially to merge k ordered sequences. Assume the average length of the k sequences is n for simplification. When merge with fold, we first merge s_1 and the empty sequence $merge([\,], s_1) = [\,] \oplus s_1$, then merge s_2 to get $[\,] \oplus s_1 \oplus s_2$, then merge s_3 to get $[\,] \oplus s_1 \oplus s_2 \oplus s_3$, ..., the

total complexity is $O(n+2n+3n+4n+...+kn) = O(n\dfrac{k(k+1)}{2}) = O(nk^2)$. While for pairwise merging, the first round is to merge $s_1 \oplus s_2$, $s_3 \oplus s_4$, ..., takes $O(kn)$ time in total; the next round is to merge $(s_1 \oplus s_2) \oplus (s_3 \oplus s_4)$, ..., taking $O(kn)$ time too. There are total lgk rounds, hence the total complexity is $O(nk\lg k)$. Therefore the pairwise merge performs better.

We can also merge with a min-heap of size k. Store the minimum elements from each sequence in the heap, keep popping the overall minimum one, and replace it with the next element in that sequence. The complexity is $O(nk\lg k)$ too.

Answer of exercise 13.3

13.3.1. Define the generic pairwise fold $foldp$, and use it to implement the bottom-up merge sort.

It's sufficient the binary combination function f be associative, i.e., $f(f(x,y),z) = f(x, f(y,z))$. Let the unit of f be z, define the pairwise fold as:

```
foldp f z [] = z
foldp f z [x] = f x z
foldp f z xs = foldp f z (pairs xs) where
  pairs (x:y:ys) = (f x y) : pairs ys
  pairs ys = ys
```

For example, we can define $sum = foldp\ (+)\ 0$, and the bottom-up merge sort is defined as[4]:

```
sort = foldp merge [] ∘ map (:[])
```

Answer of exercise 13.4

13.4.1. Build a binary search tree from a sequence using the idea of merge sort.

We can build the binary search tree either top-down or bottom-up. For top-down, halve the list to two, recursively build two trees respectively then merge; for bottom-up, wrap each element to a singleton leaf, then pair-wise merge trees repeatedly to the final result. Both approaches depend on the tree merge, let us define it[6]. If either tree is empty, the merge result is the other; otherwise, let the two trees be (A, x, B) and (C, y, D), if $x < y$, use y to partition B to two trees B_y and B^y, where B_y has all elements $\{b < y\}$, and B_y has the rest $\{b \geq y\}$; use x to partition C to C_x and C^x. We then form the merge result as:

$$(A, x, B) \oplus (C, y, D) = ((A \oplus C_x, x, B_y \oplus C^x), y, D \oplus B^y)$$

Where '$\oplus$' means tree merge. Symmetrically, when $x \geq y$, we partition A and D. Below example program implements tree *partition* and *merge*. It uses *foldp* defined in Exercise 13.3.

```
toTree :: (Ord a) ⇒ [a] → Tree a
toTree = foldp merge Empty ∘ map leaf

partition x Empty = (Empty, Empty)
partition x (Node a y b)
```

---

[4] (:[]) is equivalent to $x\ xs \mapsto [x]:xs$.

[6] For the standalone problem of binary search tree merge, we can flatten the trees to two arrays (or lists through in-order traverse), then apply *merge* (see eq. (13.31)), and rebuild the tree with the middle element as the root.

```
   |  x < y  = let (a1, a2) = partition x a in (a1, Node a2 y b)
   |  otherwise = let (b1, b2) = partition x b in (Node a y b1, b2)

merge Empty t = t
merge t Empty = t
merge (Node a x b) (Node c y d)
   |  x < y  = let
       (b1, b2) = partition y b
       (c1, c2) = partition x c
       in Node (Node (merge a c1) x (merge b1 c2)) y (merge d b2)
   |  otherwise = let
       (a1, a2) = partition y a
       (d1, d2) = partition x d
       in Node (merge a1 c) y (Node (merge a2 d1) x (merge b d2))
```

Answer of exercise 14.1

14.1.1. Prove the performance of k-selection problem is $O(n)$ in average.
Refer to the performance analysis of quick sort in section 13.1.3.

14.1.2. To find the top k element in A, we can search $x = \max{(take\ k\ A)}, y = \min{(drop\ k\ A)}$.
If $x < y$, then the first k elements in A is the answer; otherwise, we partition the
first k elements with x, partition the rest with y, then recursively find in sub-
sequence $[a \leftarrow A, x < a < y]$ for the top k' elements, where $k' = k - |[a \leftarrow A, a \leq x]|$. Implement this solution, and evaluate its performance.

1: **procedure** TOPS(k, A)
2:     $l \leftarrow 1$
3:     $u \leftarrow |A|$
4:     **loop**
5:         $i \leftarrow$ MAX-AT$(A[l..k])$
6:         $j \leftarrow$ MIN-AT$(A[k + 1..u])$
7:         **if** $A[i] < A[j]$ **then**
8:             break
9:         EXCHANGE $A[l] \leftrightarrow A[j]$
10:        EXCHANGE $A[k + 1] \leftrightarrow A[i]$
11:        $l \leftarrow$ PARTITION(A, l, k)
12:        $u \leftarrow$ PARTITION$(A, k + 1, u)$

The performance is $O(n)$ in average. Every loop takes linear time to locate the
min i, max j. Then partitions two rounds in linear time. If the partition is
balanced, we discard half elements in average, hence the total time is bound to:
$O(n + n/2 + n/4...) = O(n)$.

14.1.3. Find the 'simplified' median of two sorted arrays A and B in $O(\lg(m + n))$ time,
where $m = |A|, n = |B|$. The array index starts from 0. The simplified median
is defined as $median(A, B) = C[\lfloor \dfrac{m + n}{2} \rfloor]$, where $C = merge(A, B)$ is the merged
sorted array[5].
We give two solutions. The first applies binary search to each array. Let $l = 0$
and $u = m$ be the lower and upper bounds respectively. We guess the median in
A is at index $i = \lfloor \dfrac{l + u}{2} \rfloor$. According to the definition of the simplified median,

---

[5]In statistics, the median of an ascending data set x with n elements is defined as:

$$median(x) = \begin{cases} odd(n): & x[\frac{n+1}{2}] \\ even(n): & \frac{1}{2}(x[\frac{n}{2}] + x[\frac{n}{2} + 1]) \end{cases}$$

there are total $h = \lfloor \dfrac{m+n}{2} \rfloor$ elements before it. Where there are i elements before $A[i]$ in A. If we guess right, then there are $j = h - i$ elements before $A[i]$ in B. If $B[j] \leq A[i] \leq B[j+1]$ holds, then the guessed $A[i]$ is the median. Otherwise, we need update the boundaries l or u if the guess is too big or small. Below example program implements this solution:

```
K median([K] a, [K] b) {
    if a == [] then return b[length(b) / 2]
    if b == [] then return a[length(a) / 2]
    Int i = medianOf(a, b)
    return if i == -1 then return median(b, a) else a[i]
}

Int medianOf([K] a, [K] b) {
    Int l = 0, u = length(a)
    while l < u {
        var i = (l + u) / 2
        var j = (length(a) + length(b)) / 2  - i
        if j < 1 or j ≥ len(b) {
            if (j == 0 and a[i] ≤ b[0]) or
                (j == len(b) and b[j - 1] ≤ a[i]) then return i
            if j ≥ len(b) then l = i + 1 else u = i
        } else {
            if b[j - 1] ≤ a[i] and a[i] ≤ b[j] then return i
            if a[i] < b[j - 1] then l = i + 1 else u = i
        }
    }
    return -1
}
```

The second solution is to develop a generic function that looks for the k-th element. Assume $m \geq n$ (otherwise swap A and B), If either array is empty, then return the k-th element of the other array. If $k = 1$, then return the smaller one between $A[0]$ and $B[0]$. Otherwise guess $j = min(k/2, n)$ and $i = k - j$, then check $A[i]$ and $B[j]$. If $A[i] < B[j]$, we drop all elements before $A[i]$ and after $B[j]$, then recursively find the $(k - i)$-th element of the remaining; otherwise, drop all before $B[j]$ and after $A[i]$ for recursively finding.

```
K median([K] xs, [K] ys) {
    Int n = length(xs), m = length(ys)
    return kth(xs, 0, n, ys, 0, m, (m + n) / 2 + 1)
}

K kth([K] xs, Int x0, Int x1, [K] ys, Int y0, Int y1, Int k) {
    if x1 - x0 < y1 - y0 then return kth(ys, y0, y1, xs, x0, x1, k)
    if x1 ≤ x0 then return ys[y0 + k - 1]
    if y1 ≤ y0 then return xs[x0 + k - 1]
    if k == 1 then return min(xs[x0], ys[y0])
    var j = min(k / 2, y1 - y0), i = k - j
    i = x0 + i, j = y0 + j
    if xs[i - 1] < ys[j - 1] then
        return kth(xs, i, x1, ys, y0, j, k - i + x0)
    else
        return kth(xs, x0, i, ys, j, y1, k - j + y0)
}
```

We can't define the simplified median as the $m \in A + B$, satisfying:

$$|[y \leftarrow A + B, y < m]| - |[y \leftarrow A + B, y > m]| = 0, \pm 1$$

Meaning there are same amount of elements that are greater than m and less than m, or they only differ by 1. Consider the counter example: $[0, 1, 2, 3, 3, 3, 3, 3, 5]$. It doesn't work even if we use $\leq$ and $\geq$ instead.

14.1.4. For the saddle back search, eliminate recursion, implement it in loops to update the boundary.

```
1: function SOLVE(f, z)
2:     p ← 0, q ← z
3:     S ← φ
4:     while p ≤ z and q ≥ 0 do
5:         z' ← f(p, q)
6:         if z' < z then
7:             p ← p + 1
8:         else if z' > z then
9:             q ← q - 1
10:        else
11:            S ← S ∪ {(p, q)}
12:            p ← p + 1, q ← q - 1
13:    return S
```

14.1.5. For 2D search, let the bottom-left be the minimum, the top-right be the maximum. if z is less than the minimum or greater than the maximum, then no solution; otherwise cut the rectangle into 4 parts with a horizontal line and a vertical line crossed at the center. then recursive search in these 4 small rectangles. Implement this solution and evaluate its performance.

```
1: procedure SEARCH(f, z, a, b, c, d)          ▷ (a, b): bottom-left (c, d): top-right
2:     if z ≤ f(a, b) or 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)
```

Let the time to search in rectangle of area A be $T(A)$. We take $O(1)$ time to check whether $z \leq f(a, b)$ or $f(c, d) \geq z$, then divide into 4 smaller areas, i.e., $T(A) = 4T(A/4) + O(c)$. We apply the master theorem, the complexity is $O(A) = O(mn)$, which is proportion to the area. It's essentially same as exhaustive search in the rectangle.

Answer of exercise 14.2

14.2.1. Extend to find k majorities that occurs over $\lfloor n/k \rfloor$ times in collection A, where $n = |A|$.

We use a dictionary of $Map : T \mapsto Int$, where T is the element type in A. It records the net-wins for candidate a. Start the dictionary from empty $\varnothing$. We scan A while updating the dictionary: $foldr\ maj\ \varnothing\ A$, where maj is defined as:

$$maj \ a \ m = \begin{cases} a \in m : & m[a] \leftarrow m[a] + 1 \\ |m| < k : & m[a] \leftarrow 1 \\ \text{otherwise} : & filter \ (b \mapsto m[b] \neq 0) \ \{b \mapsto m[b] - 1 | b \in m\} \end{cases} \tag{14.21}$$

For every a in A, if $a \notin m$ (new to the dictionary), and the candidates in m is less than k, we add a to m with one net-win vote: $m[a] \leftarrow 1$; if $a \in m$, add the vote by 1: $m[a] \leftarrow m[a] + 1$; otherwise, if there are already k candidates, we reduce the vote by 1 for every one, and remove the candidate when the vote becomes 0.

We need verify the remaining candidates at last, whether the votes $> n/k$, let $m' = \{(a, 0) | a \in m\}$. Scan A again: $foldr \ cnt \ m' \ A$, where cnt is defined as:

$$cnt \ a \ m' = \text{if } a \in m' \text{ then } m'[a] \leftarrow m'[a] + 1 \text{ else } m' \tag{14.22}$$

After scan, m' records the votes for each candidate, we filter the true winners in: $keys \ (filter \ (> n/k) \ m')$.

```
majorities k xs = verify $ foldr maj Map.empty xs where
  maj x m | x `Map.member` m = Map.adjust (1+) x m
          | Map.size m < k = Map.insert x 1 m
          | otherwise = Map.filter (≠0) $ Map.map (-1 +) m
  verify m = Map.keys $ Map.filter (> th) $ foldr cnt m' xs where
    m' = Map.map (const 0) m
    cnt x m = if x `Map.member` m then Map.adjust (1+) x m else m
    th = (length xs) `div` k
```

Below is the corresponding iterative implementation:

1: **function** MAJ(k, A)
2:     $m \leftarrow \{\}$
3:     **for** each a in A **do**
4:         **if** $a \in m$ **then**
5:             $m[a] \leftarrow m[a] + 1$
6:         **else if** $|m| < k$ **then**
7:             $m[a] \leftarrow 1$
8:         **else**
9:             **for** each c in m **do**
10:                $m[c] \leftarrow m[c] - 1$
11:                **if** $m[c] = 0$ **then**
12:                    REMOVE(c, m)
13:     **for** each c in m **do**
14:         $m[c] \leftarrow 0$
15:     **for** each a in A **do**   ▷ verify
16:         **if** $a \in m$ **then**
17:             $m[a] \leftarrow m[a] + 1$
18:     $r = [\ ], n \leftarrow |A|$
19:     **for** each c in m **do**
20:         **if** $m[c] > \dfrac{n}{k}$ **then**
21:             ADD(c, r)
22:     **return** r

Answer of exercise 14.3

14.3.1. Modify the solution that finds the max sum of sub-vector, returns the sub-vector of the maximum sum.

If want to return the sub-vector together with the maximum sum, we can maintain two pairs P_m and P during folding, each pair contains the sum and the sub-vector (S, L).

$$max_s \quad = \quad 1st \circ foldr \ f \ ((0, [\ ]), (0, [\ ]))$$
$$\text{where}: \quad f \ x \ (P_m, (S, L)) = (P'_m = \max(P_m, P'), P' = \max((0, [\ ]), (x + S, x{:}L)))$$

14.3.2. Bentley gives a divide and conquer algorithm to find the max sum in $O(n \lg n)$ time [2]. Split the vector at middle, recursively find the max sum of the two halves, and the max sum that crosses the middle. Then pick the greatest. Implement this solution.

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

Consider the recursive equation: $T(n) = 2T(n/2) + O(n)$, from the master theorem, the performance is $O(n)$.

14.3.3. Find the sub-metrics in a $m \times n$ metrics that gives the maximum sum. We start from the first row of the metrics, add a row per round $[M[1, *], M[2, *], ..., M[i, *]]$. Then sum the numbers in each column and convert it to a vector:

$$V = \left[\sum_{j=1}^{i} M[1, j], \sum_{j=1}^{i} M[2, j], ..., \sum_{j=1}^{i} M[n, j] \right]$$

Next use the *maxsum* to find the max sum in vector V, and record the global maximum sum.

```
maxSum = maximum ∘ (map maxS) ∘ acc ∘ rows where
    rows = init ∘ tails          — exclude the empty row
    acc = concatMap (scanl1 (zipWith (+)))  — accumulated sum along columns
    maxS = snd ∘ (foldl f (0, 0))      — max sum in a vector
    f (m, s) x = let m' = max (m + x) 0
                     s' = max m' s in (m', s')
```

Where `tails` is defined in Exercise 1.12.2, `zipWith` is defined in section 1.8, `concatMap` is defined in section 6.5.1. The *scanl* is similar to *foldl*, it records the result every time in a list. *scanl1* is a special case of *scanl*, that the initial value is the first element.

$$scanl1\ f\ [\,] \quad = \quad [\,]$$
$$scanl1\ f\ (x{:}xs) \quad = \quad scanl\ f\ x\ xs$$

where

$$scanl\ f\ q\ [\,] \quad = \quad [q]$$
$$scanl\ f\ (x{:}xs) \quad = \quad q{:}scanl\ f\ (f\ q\ x)\ xs$$

Below is the corresponding imperative program:

```
K maxsum2([[K]] m) {
    Int n = length(m), k = length(m[0]) // number of row, col
    K maxs = 0                // max so far
    for i = 0 to n - 1 {
        xs = [0] * k
        for j = i to n - 1 {
            xs = [x + y for (x, y) in zip(xs, m[j])]
            maxs = max(maxs, maxsum1(xs))
        }
    }
    return maxs
}

K maxsum1([K] xs) {
    K s = 0 // max so far
    K m = 0 // max end here
    for x in xs {
        m = max(m + x, 0)
        s = max(m, s)
    }
    return s
}
```

Answer of exercise 14.4

14.4.1. Modify the implementation with stack, find all ways to the maze.

```
dfsSolveAll m from to = map reverse $ solve [[from]] [] where
    solve [] ss = ss
    solve (c@(p:path):cs) ss
        | p == to = solve cs (c:ss) -- find a solution, go on search
        | otherwise = let os = filter (`notElem` path) (adjacent p) in
                        if os == [] then solve cs ss
                        else solve ((map (:c) os) ++ cs) ss
    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]
```

The corresponding imperative program:

```
[[(Int, Int)]] solve([[Int]] m, (Int, Int) src, (Int, Int) dst) {
    [[(Int, Int)]] stack = [[src]]
    [[(Int, Int)]] s = []
    while stack ≠ [] {
        path = stack.pop()
        if last(path) == dst {
            s.append(path)
        } else {
            for p in adjacent(m, last(path)) {
                if not p in path then stack.append(path + [p])
            }
        }
    }
```

```
        return s
}

[(Int, Int)] adjacent([[(Int, Int)]] m, (Int x, Int y)) {
    [(Int, Int)] ps = []
    for (dx, dy) in [(0, 1), (0, -1), (1, 0), (-1, 0)] {
        Int x1 = x + dx, y1 = y + dy
        if 0 ≤ x1 < len(m[0]) and 0 ≤ y1 < len(m)
            and m[y][x] == 0 then ps.append((x1, y1))
    }
    return ps
}
```

Answer of exercise 14.5

14.5.1. Extend the 8 queens to n queens.

We are about to put n queens on the $n \times n$ board. We replace 8 with the passed in parameter n. Below table lists the solutions and the number of queens till 5.

n	solution
1	[1]
2	no solution
3	no solution
4	[2,4,1,3],[3,1,4,2]
5	[2,4,1,3,5],[3,1,4,2,5],[1,3,5,2,4], ... total 10 solutions

14.5.2. There are 92 solutions to the 8 queens puzzle. For any solution, it's also a solution if rotates 90°. We can flip to get another solution. There are essentially 12 distinct solutions. Write a program to find them.

The solutions are symmetric because the square board is symmetric. The dihedral group D_8 defines the square symmetric, including 8 permutations: identity id, counter clockwise rotate around the center by 90°, 180°, and 270°. Reflect horizontally, vertically, along the two diagonals. They send the queen at location (i, j) to:

permutation	position
id	(i, j)
reflect along Y and X	$(9 - i, j)$, $(i, 9 - j)$
reflect along two diagonals	(j, i), $(9 - j, 9 - i)$
rotate 90°, 180°, 270°	$(9 - j, i)$, $(9 - i, 9 - j)$, $(j, 9 - i)$

We apply the 8 permutations for every solution to generate 8 solutions. We treat them same, and use a set to store the 12 distinct solutions.

```
import Data.List ((\\), sortOn)
import Data.Set (Set, empty, insert, notMember, size)
import Data.Tuple (swap)

d8 = [id,
      reverse, map (9 - ),                         -- reflect Y, X
      trans swap, trans (λ(i, j) → (9 - j, 9 - i)), -- reflect AC, BD
      trans (λ(i, j) → (9 - j, i)),                -- 90
      trans (λ(i, j) → (9 - i, 9 - j)),            -- 180
      trans (λ(i, j) → (j, 9 - i))]                -- 270
  where
```

```
      trans f xs = snd $ unzip $ sortOn fst $ map f $ zip [1..8] xs

uniqueSolve = dfs [[]] (empty :: Set [Int]) where
  dfs [] s = s
  dfs (c:cs) s
    | length c == 8 = dfs cs (uniqueAdd c s)
    | otherwise = dfs ([(x:c) | x ← [1..8] \\ c,
                          not (attack x c)] ++ cs) s
  uniqueAdd c s = if all (`notMember` s) [f c | f ← d8]
                  then insert c s else s
  attack x cs = let y = 1 + length cs in
      any (λ(c, r) → abs(x - c) == abs(y - r)) $ zip (reverse cs) [1..]
```

The 12 solutions are:

```
[3,6,4,1,8,5,7,2],[3,6,8,1,4,7,5,2],[4,1,5,8,6,3,7,2],[4,2,7,3,6,8,5,1]
[4,6,8,3,1,7,5,2],[4,7,1,8,5,2,6,3],[5,2,4,7,3,8,6,1],[5,3,8,4,7,1,6,2]
[5,7,1,3,8,6,4,2],[5,7,4,1,3,8,6,2],[6,2,7,1,4,8,5,3],[6,4,7,1,8,2,5,3]
```

Answer of exercise 14.6

14.6.1. Extend the pegs puzzle solution for n pegs on each side.
We use n to build the start/end states:

```
solve n = dfs [[start]] [] where
    dfs [] s = s
    dfs (c:cs) s
        | head c == end = dfs cs (reverse c:s)
        | otherwise = dfs ((map (:c) $ moves $ head c) ++ cs) s
    start = replicate n (-1) ++ [0] ++ replicate n 1
    end = reverse start
```

Answer of exercise 14.7

14.7.1. Improve the extended Euclid algorithm, find the x and y that minimize $|x| + |y|$
for the optimal solution for the two jugs puzzle.

```
import Data.List
import Data.Function (on)

—— Extended Euclidean Algorithm
gcmex a 0 = (a, 1, 0)
gcmex a b = (g, y', x' - y' * (a `div` b)) where
  (g, x', y') = gcmex b (a `mod` b)

—— Solve the linear Diophantine equation ax + by = c
solve a b c | c `mod` g ≠ 0 = (0, 0, 0, 0) —— no solution
            | otherwise = (x1, u, y1, v)
  where
    (g, x0, y0) = gcmex a b
    (x1, y1) = (x0 * c `div` g, y0 * c `div` g)
    (u, v) = (b `div` g, a `div` g)

—— Minimize |x| + |y|
jars a b c = (x, y) where
  (x1, u, y1, v) = solve a b c
  x = x1 - k * u
  y = y1 + k * v
  k = minimumBy (compare `on` (λi → abs (x1 - i * u) +
                                    abs (y1 + i * v))) [-m..m]
  m = max (abs x1 `div` u) (abs y1 `div` v)

—— Populate the steps
```

```
water a b c = if x > 0 then pour a x b y
                    else map swap $ pour b y a x
  where
    (x, y) = jars a b c

—— Pour from a to b, fill a for x times, and empty b for y times.
pour a x b y = steps x y [(0, 0)]
  where
    steps 0 0 ps = reverse ps
    steps x y ps@((a', b'):_)
      | a' == 0 = steps (x - 1) y ((a, b'):ps)   —— fill a
      | b' == b = steps x (y + 1) ((a', 0):ps)   —— empty b
      | otherwise = steps x y ((max (a' + b' - b) 0,
                                min (a' + b') b):ps) —— a to b
```

See section 2.2.3, chapter 2 in '*Isomorphism - mathematics of programming*' for more details.

<div align="center">

Answer of exercise 14.8

</div>

14.8.1. Traverse a binary tree level by level, i.e., first visit the root, then the nodes at level 1, then level 2, ...
This is to BFS visit the tree. Below example uses a sequence based queue (see section 12.5).

```
data Tr a = E | Br (Tr a) a (Tr a)

toList t = bfs (Q.singleton t) where
  bfs Q.Empty = []
  bfs (E :<| q) = bfs q
  bfs ((Br l k r) :<| q) = k : bfs (q |> l |> r)
```

14.8.2. Conway slide puzzle.
We use 8 numbers from 0 to 7 to represent pieces, where 0 is the free cell. The pieces next to 0 can slide in, we represent it as the number 0 moves forward/backward in a permutation. When moves forward beyond tail, it goes back to the head; when moves backward beyond head, it goes to the tail. Particularly, when 0 is the first, it can swap with the 5th piece and vice versa. The solution is very long with 12948 steps. The last several steps is as below.

```
start = [0..7]
end = 0:[7,6..1]

solve1 = dfs [[start]] where
  dfs [] = []
  dfs (c:cs)
    | head c == end = reverse c
    | otherwise = dfs ((map (:c) $ moves c) ++ cs)

moves (s:visited) = filter (`notElem` visited) [fwd s, bk s, cut s]
  where
    fwd xs = case break (0 ==) xs of
      (as, 0:b:bs) → as ++ (b:0:bs)
      (a:as, [0]) → 0:as ++ [a]
    bk xs = case break (0 ==) xs of
      ([], 0:bs) → bs ++ [0]
      (as, 0:bs) → (init as) ++ (0 : last as : bs)
    cut xs = case splitAt 4 xs of
      ((0:as), (x:bs)) → (x:as) ++ (0:bs)
      ((x:as), (0:bs)) → (0:as) ++ (x:bs)
      _ → xs
```

```
...
[1,0,7,6,5,4,3,2],[1,7,0,6,5,4,3,2],[1,7,6,0,5,4,3,2],[1,7,6,5,0,4,3,2],
[1,7,6,5,4,0,3,2],[1,7,6,5,4,3,0,2],[1,7,6,5,4,3,2,0],[0,7,6,5,4,3,2,1]
```

Answer of exercise 14.9

14.9.1. Implement the imperative Huffman code table algorithm.

```
data Node<T> {
    Optional<T> c = Nothing
    Int w
    Node<T> left = null, right = null

    Bool isLeaf() = (left == null and right == null)
}

Node<T> merge(Node<T> a, Node<T> b) = Node(Nothing, a.w + b.w, a, b)

Bool (<)(Node<T> a, Node<T> b) = (a.w < b.w)

Node<T> huffman([Node<T>] ts) {
    while length(ts) > 1 {
        Int n = length(ts)
        for Int i = n - 3 down to i 0 {
            if ts[i] < max(ts[n-1], ts[n-2]) {
                Int j = if ts[n-1] < ts[n-2] then n - 2 else n - 1
                swap(ts[i], ts[j])
            }
        }
        ts[n-2] = merge(ts[n-1], ts[n-2])
        ts.popLast()
    }
    return ts[0]
}

Map<T, [T]> codeTab(Node<T> t, [T] bits = [], Map<T, [T]> codes = {}) {
    if t.isLeaf() {
        codes[t.c] = bits
    } else {
        codeTab(t.left,  bits + [0], codes)
        codeTab(t.right, bits + [1], codes)
    }
    return codes
}
```

Answer of exercise 14.10

14.10.1. Use heap to build the Huffman tree: take two trees from the top, merge then add back to the heap.

$$
Huffman\ H = \begin{cases} H = \varnothing: & \varnothing \\ |H| = 1: & pop\ H \\ \text{otherwise}: & Huffman\ (push\ (merge\ t_a\ t_b)\ H'') \end{cases}
$$

Where: $(t_a, H') = pop\ H, (t_b, H'') = pop\ H'$

1: **function** HUFFMAN(H)
2:     **while** $|H| > 1$ **do**
3:         $t_a \leftarrow$ POP(H)
4:         $t_b \leftarrow$ POP(H)

5:          $\text{PUSH}(H, \text{MERGE}(t_a, t_b))$

6:      **return** $\text{POP}(H)$

14.10.2. If we sort the symbols by their weight as A, there is a linear time algorithm to build the Huffman tree: use a tree Q to store the merge result, repeat take the minimal weighted tree from Q and the head of A, merge then add to the queue. After process all trees in A, there is a single tree in the Q, which is the Huffman tree. Implement this algorithm.

$Huffman\ (t{:}ts) = build\ (t, (ts, \varnothing))$, where:

$$
\begin{aligned}
build\ (t, ([\,], \varnothing)) &= t \\
build\ (t, h) &= build\ (extract\ (ts, push\ (merge\ t\ t')\ q))
\end{aligned}
$$

where $(t', (ts, q)) = extract\ h$

$$
\begin{aligned}
extract\ (t{:}ts, \varnothing) &= (t, (ts, \varnothing)) \\
extract\ ([\,], q) &= (t, ([\,], q')), \text{where}: (t, q') = pop\ q \\
extract\ (t{:}ts, q) &= \begin{cases} t' < t : & (t', (t{:}ts, q')), \text{where}: (t', q') = pop\ q \\ t < t' : & (t, (ts, q)) \end{cases}
\end{aligned}
$$

14.10.3. Given a Huffman tree T, implement the decode algorithm with fold left.

$decode = snd \circ (foldl\ lookup\ (T, [\,]))$, where:

$$
\begin{aligned}
lookup\ ((w, c), cs)\ b &= (T, c{:}cs) \\
lookup\ ((w, l, r), cs)\ b &= \text{if } b = 0 \text{ then } (l, cs) \text{ else } (r, cs)
\end{aligned}
$$

Answer of exercise 14.11

14.11.1. For the longest common sub-sequence, build the optimal solution table with fold.

```haskell
import Data.Sequence (Seq, singleton, fromList, index, (|>))

lcs xs ys = construct $ foldl f (singleton $ fromList $ replicate (n+1) 0)
                                (zip [1..] xs) where
  (m, n) = (length xs, length ys)
  f tab (i, x) = tab |> (foldl longer (singleton 0) (zip [1..] ys)) where
    longer r (j, y) = r |> if x == y
                    then 1 + (tab `index` (i-1) `index` (j-1))
                    else max (tab `index` (i-1) `index` j) (r `index` (j-1))
  construct tab = get (reverse xs, m) (reverse ys, n) where
    get ([], 0) ([], 0) = []
    get ((x:xs), i) ((y:ys), j)
      | x == y = get (xs, i-1) (ys, j-1) ++ [x]
      | (tab `index` (i-1) `index` j) > (tab `index` i `index` (j-1)) =
          get (xs, i-1) ((y:ys), j)
      | otherwise = get ((x:xs), i) (ys, j-1)
```

Answer of exercise 14.12

14.12.1. For the longest common sub-sequence problem, an alternative solution is to record the length and the direction in the table. There are three directions: 'N' for north, 'W' for west, and 'NW'. Given such a table, we can build the longest common sub-sequence from the bottom-right entry. If the entry is 'NW', next go to the upper-left entry; if it's 'N', go to the above row; and go to the previous entry if it's 'W'. Implement this solution.

```
data DIR = N | W | NW

[K] lcs([K] xs, [K] ys) {
    Int m = length(x), n = length(ys)
    [[(Int, DIR)]] c = [[(0, null)] * (n + 1)] * (m + 1)
    for i = 1 to m {
        for j = 1 to n {
            if xs[i-1] == ys[j-1] {
                c[i][j] = (fst(c[i-1][j-1]) + 1, DIR.NW)
            } else {
                c[i][j] = if fst(c[i-1][j]) > fst(c[i][j-1])
                    then (fst(c[i-1][j]), DIR.N)
                    else (fst(c[i][j-1]), DIR.W)
            }
        }
    }
    return rebuild(c, xs, ys)
}

[K] rebuild([[(Int, DIR)]] c, [K] xs, [K] ys) {
    [K] r = []
    Int m = length(xs), n = length(ys)
    while m > 0 and n > 0 {
        DIR d = snd(c[m][n])
        if d == DIR.NW {
            r.append(xs[m - 1]) // or ys[n - 1]
            m = m - 1, n = n - 1
        } else if d == DIR.N {
            m = m - 1
        } else if d == DIR.W {
            n = n - 1
        }
    }
    return reverse(r)
}
```

14.12.2. For the subset sum upper/lower bound, does $l \leq 0 \leq u$ always hold? can we reduce
the range between the bounds?
For non-empty subset (the sum of empty is 0), $l \leq 0 \leq u$ does not hold necessarily.
Consider set X of only positive numbers, the lower bound can't be less than 0, and
$l = \min(X)$. For set of only negative numbers, the upper bound can't be greater
than 0, and $u = \max(X)$.

14.12.3. Compute the edit distance between two strings.

```
Int lev([K] s, [K] t) {
    [[Int]] d = [[0]*n]*m   //d[i][j]: distance between s[:i] and t[:j]
    for Int i = 0 to length(s) {
        d[i][0] = i  //drop all chars of source prefix gives []
    }
    for Int j = 0 to length(t) {
        d[0][j] = j  //insert all chars of target prefix to []
    }
    for Int j = 1 to length(t) {
        for i = 1 to length(m) {
            c = if s[i-1] == t[j-1] then 0 else 1
            d[i][j] = min([d[i-1][j] + 1,     //deletion
                           d[i][j-1] + 1,     //insertion
                           d[i-1][j-1] + c]) //substitution
        }
    }
    return d[length(s)][length(t)]
}
```

References

[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. https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD934.PDF.

[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