AlgoXY elementary algorithms

Liu Xinyu [1]

February 10, 2013

[1] **Liu Xinyu**
Draft version 2012, Beijing
Email: liuxinyu95@gmail.com

Contents

Preface

Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

0.1 Why?

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

Instead of answering this question directly, I would like to say algorithms and data structures are critical in solving 'interesting problems', while if the problem is useful is another thing.

Let's start with two problems. It looks like both of them can be solved in brute-force way even by a fresh programmer.

0.2 The smallest free ID problem, the power of algorithm

This problem is discussed in Chapter 1 of Richard Bird's book [1]. It's common that many applications and systems use ID (identifier) to manage the objects and entities. At any time, some IDs are used, and some of them are available for using. When some client tries to acquire a new ID, we want to always allocate it the smallest available one. Suppose ID is no-negative integers and all IDs in using are maintained in a list (or an array) which is not ordered. For example:

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

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

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

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

Where the $\notin$ is realized like below. Here we use notation $[a, b)$ in Math to define a range from a to b with b excluded.

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

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

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

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

0.2.1 Improvement 1

The key idea to improve the solution is based on a fact that for a series of N numbers $x_1, x_2, ..., x_N$, if there are free numbers, not all of the x_i are in range $[0, N)$; otherwise the list is exactly one permutation of $0, 1, ..., N-1$ and N should be returned as the minimum free number respectively. It means that $max(x_i) \geq N - 1$. And we have the following fact.

$$minfree(x_1, x_2, ..., x_N) \leq N \tag{1}$$

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

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

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

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

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

We haven't fine tune this algorithm yet. Observe that each time we have to allocate memory to create a $N-$element array of flags, and release the memory

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

when finish. The memory allocation and release is very expensive that they cost a lot of processing time.

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

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

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

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

unsigned int bits[N/WORD_LENGTH+1];

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

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

The last for-loop can be further improved as below. This is just a minor fine-tuning.

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

0.2.2 Improvement 2, Divide and Conquer

Although the above improvement looks perfect, it costs $O(N)$ extra spaces to keep a check list. if N is huge number, which means the huge amount of spaces are need.

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

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

When we search the minimum free number in A'', the condition changes a little bit, we are not searching the smallest free number from 0, but actually from $\lfloor N/2 \rfloor + 1$ as the lower bound. So the algorithm is something like $minfree(A, l, u)$, where l is the lower bound and u is the upper bound index of the element.

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

The divide and conquer solution can be formally expressed as a function rather than the pseudo code.

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

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

where

$$\begin{aligned} m &= \lfloor \tfrac{l+u}{2} \rfloor \\ A' &= \{\forall x \in A \wedge x \leq m\} \\ A'' &= \{\forall x \in A \wedge x > m\} \end{aligned}$$

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

In functional programming language such as Haskell, partition list has already been provided in library. This algorithm can be translated as the following.

```
import Data.List

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

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

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

```
    m = (l + u) `div` 2
    (as, bs) = partition (≤m) xs
```

0.2.3 Expressiveness V.S. Performance

Imperative language programmers may concern about the performance of this kind of implementation. For instance in this minimum free number problem, The function recursively called proportion to $O(\lg N)$, which means the stack size consumed is $O(\lg N)$. It's not free in terms of space.

If we go one step ahead, we can eliminate the recursion by iteration which yields the following C program.

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

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

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

0.3 The number puzzle, power of data structure

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

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

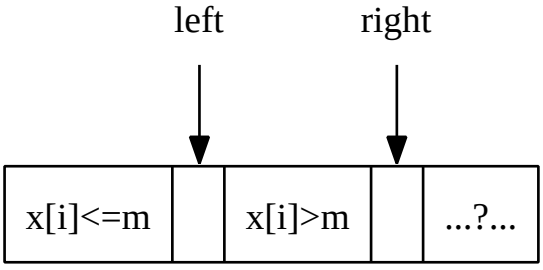


Figure 1: Divide the array, all $x[i] \leq m$ where $0 \leq i < left$; while all $x[i] > m$ where $left \leq i < right$. The left elements are unknown.

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

0.3.1 The brute-force solution

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

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

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

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

0.3.2 Improvement 1

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

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

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

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

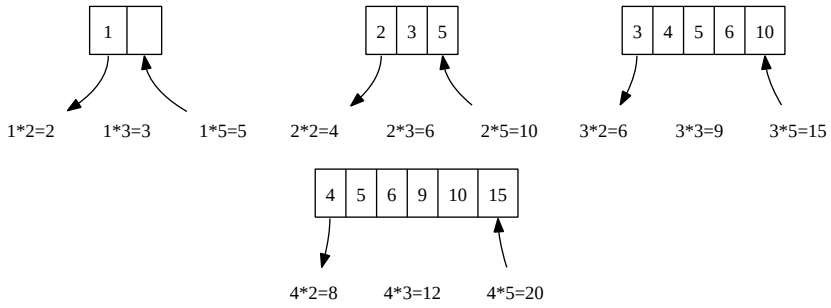


Figure 2: First 4 steps of constructing numbers with a queue.
1. Queue is initialized with 1 as the only element;
2. New elements 2, 3, and 5 are pushed back;
3. New elements 4, 6, and 10, are pushed back in order;
4. New elements 9 and 15 are pushed back, element 6 already exists.

This algorithm is shown as the following.

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

11: function UNIQUE-ENQUEUE(Q, x)
```

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

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

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

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

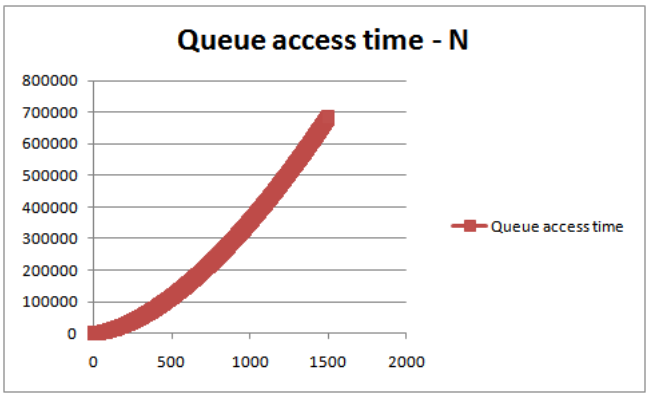


Figure 3: Queue access count v.s. n.

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

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

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

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

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

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

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

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

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

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

0.3.3 Improvement 2

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

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

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

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

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

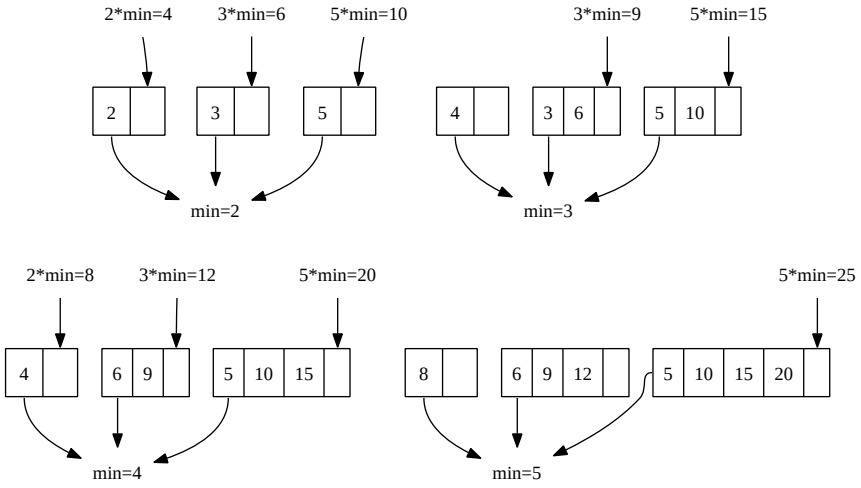


Figure 4: First 4 steps of constructing numbers with Q_2, Q_3, and Q_5.
1. Queues are initialized with 2, 3, 5 as the only element;
2. New elements 4, 6, and 10 are pushed back;
3. New elements 9, and 15, are pushed back;
4. New elements 8, 12, and 20 are pushed back;
5. New element 25 is pushed back.

```
16:            DEQUEUE(Q_3)
17:            ENQUEUE(Q_3, 3x)
18:            ENQUEUE(Q_5, 5x)
19:        else
20:            DEQUEUE(Q_5)
21:            ENQUEUE(Q_5, 5x)
22:        n ← n − 1
23:    return x
```

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

```
typedef unsigned long Integer;

Integer get_number(int n){
  if(n==1)
    return 1;
  queue<Integer> Q2, Q3, Q5;
  Q2.push(2);
  Q3.push(3);
  Q5.push(5);
  Integer x;
```

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

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

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

Where

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

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

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

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

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

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

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

0.4 Notes and short summary

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

The first problem shows the power of algorithms, while the second problem tells why data structure is important. There are plenty of interesting problems, which are hard to solve before computer was invented. With the aid of computer and programming, we are able to find the answer in a quite different way. Compare to what we learned in mathematics course in school, we haven't been taught the method like this.

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

This series of post focus on providing both imperative and functional algorithms and data structures. Many functional data structures can be referenced from Okasaki's book[6]. While the imperative ones can be founded in classic text books [3] or even in WIKIpedia. Multiple programming languages, including, C, C++, Python, Haskell, and Scheme/Lisp will be used. In order to make it easy to read by programmers with different background, pseudo code and mathematical function are the regular descriptions of each post.

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

0.5 Structure of the contents

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

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

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

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

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

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

0.6 Appendix

All programs provided along with this article are free for downloading. download position: http://sites.google.com/site/algoxy/introduction

Bibliography

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

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

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

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

Part I

Trees

Binary search tree, the 'hello world' data structure
Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

Chapter 1

Binary search tree, the 'hello world' data structure

1.1 Introduction

It's typically considered that Arrays or Lists are the 'hello world' data structure. However, we'll see they are not so easy to implement actually. In some procedural settings, Arrays are the elementary representation, and it is possible to realize linked list by array (section 10.3 in [3]); While in some functional settings, Linked list are the elementary bricks to build arrays and other data structures.

Considering these factors, we start with Binary Search Tree (or BST) as the 'hello world' data structure. Jon Bentley mentioned an interesting problem in 'programming pearls' [2]. The problem is about to count the number of times each word occurs in a big text. And the solution is something like the below C++ code.

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

And we can run it to produce the word counting result as the following [1].

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

The map provided in standard template library is a kind of balanced binary search tree with augmented data. Here we use the word in the text as the key and the number of occurrence as the augmented data. This program is fast, and

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

it reflects the power of binary search tree. We'll introduce how to implement BST in this post and show how to solve the balancing problem in later post.

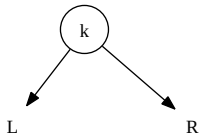
Before we dive into binary search tree. Let's first introduce about the more general binary tree.

The concept of Binary tree is a recursive definition. Binary search tree is just a special type of binary tree. The Binary tree is typically defined as the following.

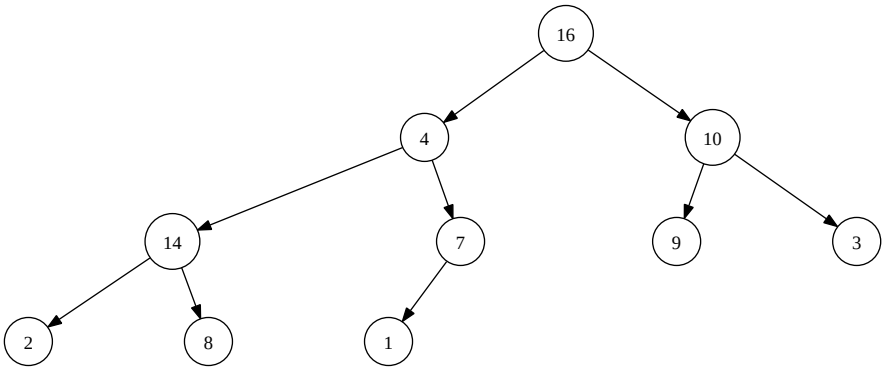
A binary tree is

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

Figure 1.1 shows this concept and an example binary tree.



(a) Concept of binary tree



(b) An example binary tree

Figure 1.1: Binary tree concept and an example.

A binary search tree is a binary tree which satisfies the below criteria. for each node in binary search tree,

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

Figure 1.2 shows an example of binary search tree. Compare with Figure 1.1 we can see the difference about the key ordering between them.

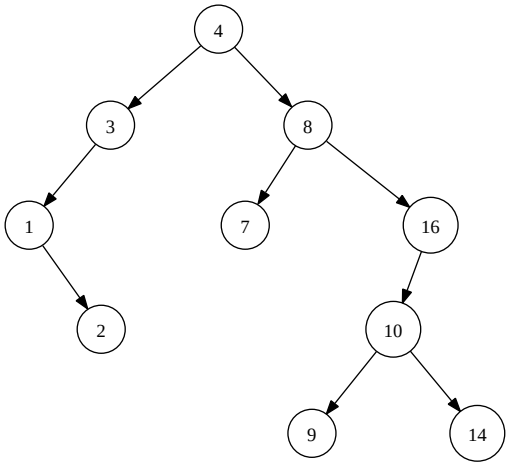


Figure 1.2: A Binary search tree example.

1.2 Data Layout

Based on the recursive definition of binary search tree, we can draw the data layout in procedural setting with pointer supported as in figure 1.3.

The node contains a field of key, which can be augmented with satellite data; a field contains a pointer to the left child and a field point to the right child. In order to back-track an ancestor easily, a parent field can be provided as well.

In this post, we'll ignore the satellite data for simple illustration purpose. Based on this layout, the node of binary search tree can be defined in a procedural language, such as C++ as the following.

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

  node* left;
  node* right;
  node* parent; //parent is optional, it's␣helpful␣for␣succ/pred
␣␣T␣key;
};
```

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

Because in pure functional setting, It's hard to use pointer for back tracking the ancestors, (and typically, there is no need to do back tracking, since we can

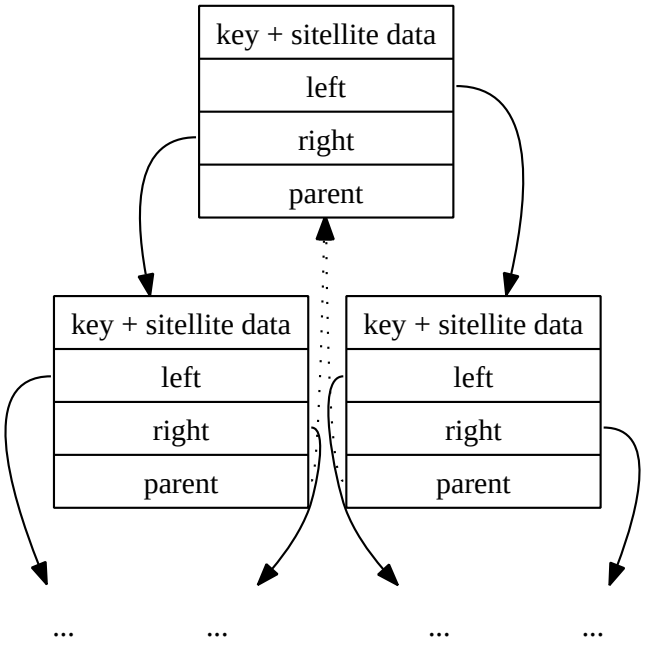


Figure 1.3: Layout of nodes with parent field.

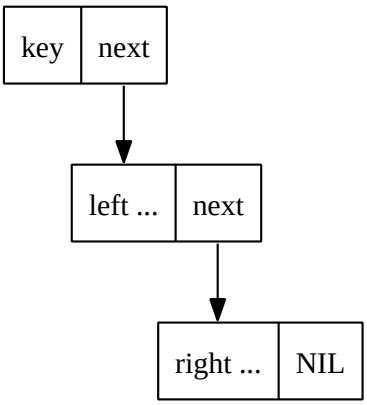


Figure 1.4: Binary search tree node layout on top of linked list. Where 'left...' and 'right ...' are either empty or binary search tree node composed in the same way.

provide top-down solution in recursive way) there is not 'parent' field in such layout.

For simplified reason, we'll skip the detailed layout in the future, and only focus on the logic layout of data structures. For example, below is the definition of binary search tree node in Haskell.

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

1.3 Insertion

To insert a key k (may be along with a value in practice) to a binary search tree T, we can follow a quite straight forward way.

- If the tree is empty, then construct a leave node with key=k;
- If k is less than the key of root node, insert it to the left child;
- If k is greater than the key of root, insert it to the right child;

There is an exceptional case that if k is equal to the key of root, it means it has already existed, we can either overwrite the data, or just do nothing. For simple reason, this case is skipped in this post.

This algorithm is described recursively. It is so simple that is why we consider binary search tree is 'hello world' data structure. Formally, the algorithm can be represented with a recursive function.

$$insert(T, k) = \begin{cases} node(\phi, k, \phi) & : & T = \phi \\ node(insert(L, k), Key, R) & : & k < Key \\ node(L, Key, insert(R, k)) & : & otherwise \end{cases} \tag{1.1}$$

Where

$$\begin{aligned} L &= left(T) \\ R &= right(T) \\ Key &= key(T) \end{aligned}$$

The node function creates a new node with given left sub-tree, a key and a right sub-tree as parameters. ϕ means NIL or Empty. function $left$, $right$ and key are access functions which can get the left sub-tree, right sub-tree and the key of a node.

Translate the above functions directly to Haskell yields the following program.

```
insert::(Ord a) ⇒ Tree a → a → Tree a
insert Empty k = Node Empty k Empty
insert (Node l x r) k | k < x = Node (insert l k) x r
                      | otherwise = Node l x (insert r k)
```

This program utilized the pattern matching features provided by the language. However, even in functional settings without this feature, for instance, Scheme/Lisp, the program is still expressive.

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

It is possible to turn the algorithm completely into imperative way without recursion.

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

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

Compare with the functional algorithm, it is obviously that this one is more complex although it is fast and can handle very deep tree. A complete C++ program and a python program are available along with this post for reference.

1.4 Traversing

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

Since there are three parts in a tree, as left child, the root, which contains the key and satellite data, and the right child. If we denote them as $(left, current, right)$, the three traversing methods are defined as the following.

- pre-order traverse, visit $current$, then $left$, finally $right$;
- in-order traverse, visit $left$, then $current$, finally $right$;
- post-order traverse, visit $left$, then $right$, finally $current$.

Note that each visiting operation is recursive. And we see the order of visiting $current$ determines the name of the traversing method.

For the binary search tree shown in figure 1.2, below are the three different traversing results.

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

It can be found that the in-order walk of a binary search tree outputs the elements in increase order, which is particularly helpful. The definition of binary search tree ensures this interesting property, while the proof of this fact is left as an exercise of this post.

In-order tree walk algorithm can be described as the following:

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

Translate the above description yields a generic map function

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

where

$$\begin{aligned} l' &= map(f, left(T)) \\ r' &= map(f, right(T)) \\ k' &= f(key(T)) \end{aligned}$$

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

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

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

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

$$toList(T) = \begin{cases} \phi & : \quad T = \phi \\ toList(left(T)) \cup \{key(T)\} \cup toList(right(T)) & : \quad otherwise \end{cases} \tag{1.3}$$

Below is the Haskell program based on this definition.

```
toList::(Ord a)⇒Tree a → [a]
toList Empty = []
toList (Node l x r) = toList l ++ [x] ++ toList r
```

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

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

And we can write it in function composition form.

$$sort = toList.fromList$$

Where function $fromList$ repeatedly insert every element to a binary search tree.

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

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

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

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

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

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

Exercise 1.1

- Given the in-order traverse result and pre-order traverse result, can you re-construct the tree from these result and figure out the post-order traversing result?

  Pre-order result: 1, 2, 4, 3, 5, 6; In-order result: 4, 2, 1, 5, 3, 6; Post-order result: ?

- Write a program in your favorite language to re-construct the binary tree from pre-order result and in-order result.

- Prove why in-order walk output the elements stored in a binary search tree in increase order?
- Can you analyze the performance of tree sort with big-O notation?

1.5 Querying a binary search tree

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

1.5.1 Looking up

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

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

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

$$lookup(T, x) = \begin{cases} \phi & : & T = \phi \\ T & : & key(T) = x \\ lookup(left(T), x) & : & x < key(T) \\ lookup(right(T), x) & : & otherwise \end{cases} \tag{1.6}$$

In the real application, we may return the satellite data instead of the node as the search result. This algorithm is simple and straightforward. Here is a translation of Haskell program.

```
lookup::(Ord a)⇒ Tree a → a → Tree a
lookup Empty _ = Empty
lookup t@(Node l k r) x | k == x = t
                        | x < k = lookup l x
                        | otherwise = lookup r x
```

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

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

1: **function** SEARCH(T, x)

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

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

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

1.5.2 Minimum and maximum

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

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

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

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

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

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

1.5.3 Successor and predecessor

The last kind of querying, to find the successor or predecessor of an element is useful when a tree is treated as a generic container and traversed by using iterator. It will be relative easier to implement if parent of a node can be accessed directly.

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

However, If we consider the original purpose of providing *succ/pred* function, 'to traverse all the binary search tree elements one by one' as a generic container, we realize that they don't make significant sense in functional settings

because we can traverse the tree in increase order by *mapT* function we defined previously.

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

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

When finding the successor of element x, which is the smallest one y that satisfies $y > x$, there are two cases. If the node with value x has non-NIL right child, the minimum element in right child is the answer; For example, in Figure 1.2, in order to find the successor of 8, we search it's right sub tree for the minimum one, which yields 9 as the result. While if node x don't have right child, we need back-track to find the closest ancestors whose left child is also ancestor of x. In Figure 1.2, since 2 don't have right sub tree, we go back to its parent 1. However, node 1 don't have left child, so we go back again and reach to node 3, the left child of 3, is also ancestor of 2, thus, 3 is the successor of node 2.

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

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

The predecessor case is quite similar to the successor algorithm, they are symmetrical to each other.

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

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

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

def pred(x):
```

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

Exercise 1.2

- Can you figure out how to iterate a tree as a generic container by using $pred()/succ()$? What's the performance of such traversing process in terms of big-O?

- A reader discussed about traversing all elements inside a range $[a, b]$. In C++, the algorithm looks like the below code:

  for_each$(m.lower\_bound(12), m.upper\_bound(26), f)$;

  Can you provide the purely function solution for this problem?

1.6 Deletion

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

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

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

In this post, different with the algorithm described in [3], A simpler one from SGI STL implementation is used.[6]

To delete a node x from a tree.

- If x has no child or only one child, splice x out;

- Otherwise (x has two children), use minimum element of its right sub tree to replace x, and splice the original minimum element out.

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

Figure 1.5, 1.6, and 1.7 illustrate these different cases when deleting a node from the tree.

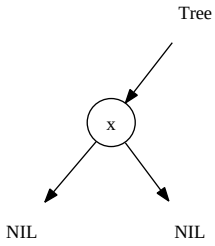
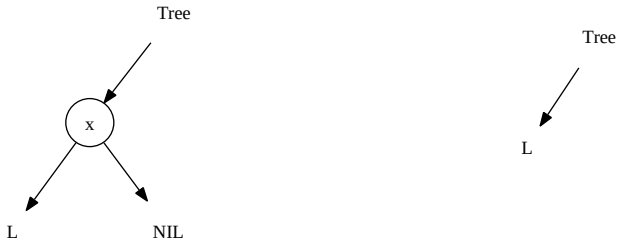
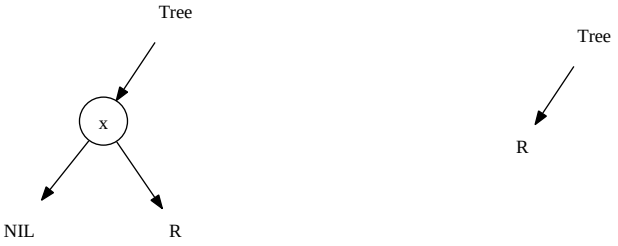


Figure 1.5: x can be spliced out.



(a) Before delete x (b) After delete x

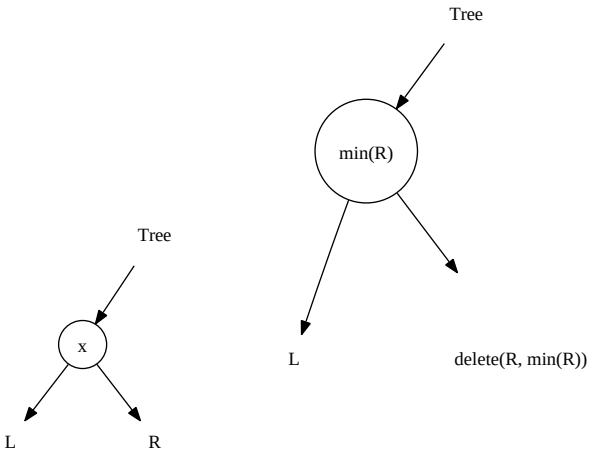
x is spliced out, and replaced by its left child.



(c) Before delete x (d) Before delete x

x is spliced out, and replaced by its right child.

Figure 1.6: Delete a node which has only one non-NIL child.



(a) Before delete x (b) After delete x

x is replaced by splicing the minimum element from its right child.

Figure 1.7: Delete a node which has both children.

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

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

Where

$$\begin{aligned} L &= left(T) \\ R &= right(T) \\ K &= key(T) \\ y &= min(R) \end{aligned}$$

Translating the function to Haskell yields the below program.

```
delete::(Ord a)⇒ Tree a → a → Tree a
delete Empty _ = Empty
delete (Node l k r) x | x < k = (Node (delete l x) k r)
                      | x > k = (Node l k (delete r x))
                      -- x == k
                      | isEmpty l = r
                      | isEmpty r = l
                      | otherwise = (Node l k' (delete r k'))
                        where k' = min r
```

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

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

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

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

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

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

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

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

```
def tree_delete(t, x):
```

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

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

Exercise 1.3

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

1.7 Randomly build binary search tree

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

How to make the binary search tree balanced will be discussed in next post. However, there exists a simple way. Binary search tree can be randomly built as described in [3]. Randomly building can help to avoid (decrease the possibility) unbalanced binary trees. The idea is that before building the tree, we can call a random process, to shuffle the elements.

Exercise 1.4

- Write a randomly building process for binary search tree.

1.8 Appendix

All programs are provided along with this post. They are free for downloading. We provided C, C++, Python, Haskell, and Scheme/Lisp programs as example.

Bibliography

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

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

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

[4] SGI. "Standard Template Library Programmer's Guide". http://www.sgi.com/tech/stl/

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

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

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

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

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

The evolution of insertion sort
Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

Chapter 2

The evolution of insertion sort

2.1 Introduction

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

The idea of insertion sort can be vivid illustrated by a real life poker game[3]. Suppose the cards are shuffled, and a player starts taking card one by one.

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

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

```
function SORT(A)
    X ← Φ
    for each x ∈ A do
        INSERT(X, x)
    return X
```

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

$$insert = foldL \quad insert \quad \Phi \tag{2.1}$$

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

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

Note that in above algorithm, we store the sorted result in X, so this isn't in-place sorting. It's easy to change it to in-place algorithm.

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

At any time, when we process the i-th element, all elements before i have already been sorted. we continuously insert the current elements until consume all the unsorted data. This idea is illustrated as in figure 9.3.

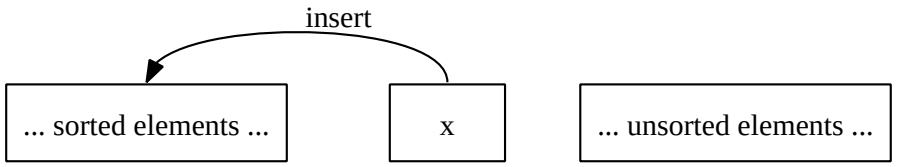


Figure 2.2: The left part is sorted data, continuously insert elements to sorted part.

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

$$sort(A) = \begin{cases} \Phi & : \quad A = \Phi \\ insert(sort(\{A_2, A_3, ...\}), A_1) & : \quad otherwise \end{cases} \tag{2.2}$$

2.2 Insertion

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

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

function SORT(A)
  for $i \leftarrow 2$ to LENGTH(A) **do** ▷ Insert $A[i]$ to sorted sequence $A[1...i-1]$
    $x \leftarrow A[i]$

$j \leftarrow i - 1$
while $j > 0 \wedge x < A[j]$ **do**
$A[j + 1] \leftarrow A[j]$
$j \leftarrow j - 1$
$A[j + 1] \leftarrow x$

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

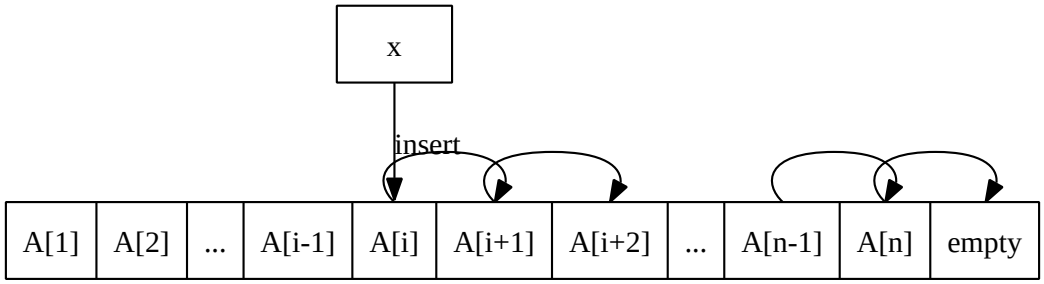


Figure 2.3: Insert x to array A at position i.

If the length of array is N, this indicates we need examine the first i elements, then perform $N - i + 1$ moves, and then insert x to the i-th cell. So insertion from left to right need traverse the whole array anyway. While if we scan from right to left, we totally examine the last $j = N - i + 1$ elements, and perform the same amount of moves. If j is small (e.g. less than $N/2$), there is possibility to perform less operations than scan from left to right.

Translate the above algorithm to Python yields the following code.

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

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

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

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

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

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

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

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

Exercise 2.1

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

2.3 Improvement 1

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

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

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

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

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

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

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

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

The Python program regarding to this algorithm is given below.

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

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

Exercise 2.2

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

2.4 Improvement 2

Although we improve the search time to $O(N \lg N)$ in previous section, the number of moves is still $O(N^2)$. The reason of why movement takes so long time, is because the sequence is stored in plain array. The nature of array is continuously layout data structure, so the insertion operation is expensive. This hints us that we can use linked-list setting to represent the sequence. It

can improve the insertion operation from $O(N)$ to constant time $O(1)$.

$$insert(A, x) = \begin{cases} \{x\} & : & A = \Phi \\ \{x\} \cup A & : & x < A_1 \\ \{A_1\} \cup insert(\{A_2, A_3, ...A_n\}, x) & : & otherwise \end{cases} \quad (2.3)$$

Translating the algorithm to Haskell yields the below program.

```
insert :: (Ord a) ⇒ [a] → a → [a]
insert [] x = [x]
insert (y:ys) x = if x < y then x:y:ys else y:insert ys x
```

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

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

Or we can represent the recursion with folding.

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

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

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

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

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

Thus the insert function can be given as below.

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

```
  return head;
}
```

Instead of using explicit linked-list such as by pointer or reference based structure. Linked-list can also be realized by another index array. For any array element A_i, $Next_i$ stores the index of next element follows A_i. It means A_{Next_i} is the next element after A_i.

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

function INSERT($A, Next, i$)
  $j \leftarrow \perp$
  while $Next_j \neq NIL \wedge A_{Next_j} < A_i$ **do**
    $j \leftarrow Next_j$
  $Next_i \leftarrow Next_j$
  $Next_j \leftarrow i$

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

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

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

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

Exercise 2.3

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

2.5 Final improvement by binary search tree

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

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

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

So the algorithm changes to this.

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

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

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

2.6 Short summary

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

Bibliography

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

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

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

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

Liu Xinyu **Liu Xinyu**

Email: liuxinyu95@gmail.com

Chapter 3

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

3.1 Introduction

3.1.1 Exploit the binary search tree

We have shown the power of binary search tree by using it to count the occurrence of every word in Bible. The idea is to use binary search tree as a dictionary for counting.

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

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

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

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

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

[1] A name-phone number contact list book

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

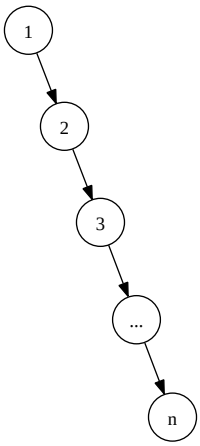


Figure 3.1: unbalanced tree

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

Exercise 3.1

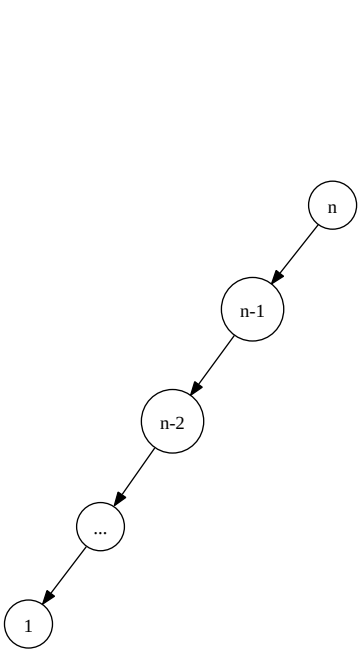
- For a very big yellow page list, one may want to speed up the dictionary building process by two concurrent tasks (threads or processes). One task reads the name-phone pair from the head of the list, while the other one reads from the tail. The building terminates when these two tasks meet at the middle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

- Can you find any more cases to exploit a binary search tree? Please consider the unbalanced trees shown in figure 3.2.

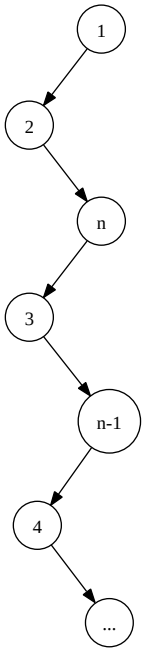
3.1.2 How to ensure the balance of the tree

In order to avoid such case, we can shuffle the input sequence by randomized algorithm, such as described in Section 12.4 in [3]. However, this method doesn't always work, for example the input is fed from user interactively, and the tree need to be built/updated after each input.

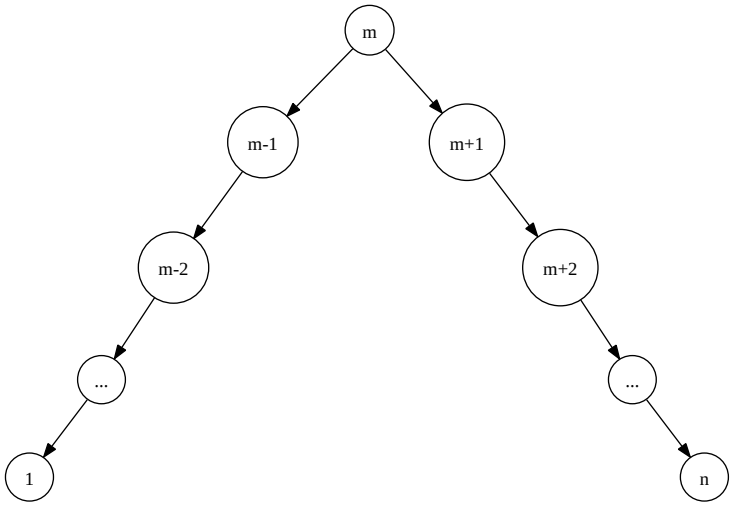
There are many solutions people have ever found to make binary search tree balanced. Many of them rely on the rotation operations to binary search tree. Rotation operations change the tree structure while maintain the ordering of the elements. Thus it either improve or keep the balance property of the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



(a)



(b)

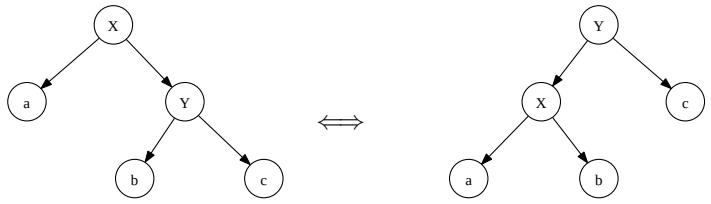


(c)

Figure 3.2: Some unbalanced trees

In this chapter, we'll first introduce about red-black tree, which is one of the most popular and widely used self-adjusting balanced binary search tree. In next chapter, we'll introduce about AVL tree, which is another intuitive solution; In later chapter about binary heaps, we'll show another interesting tree called splay tree, which can gradually adjust the the tree to make it more and more balanced.

3.1.3 Tree rotation



(a) (b)

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

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

Although tree rotation can be realized in procedural way, there exists quite simple function description if using pattern matching.

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

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

However, the pseudo code dealing imperatively has to set all fields accordingly.

```
1: function LEFT-ROTATE(T, x)
2:     p ← PARENT(x)
3:     y ← RIGHT(x)                          ▷ Assume y ≠ NIL
4:     a ← LEFT(x)
5:     b ← LEFT(y)
6:     c ← RIGHT(y)
```

```
7:     Replace(x, y)
8:     Set-Children(x, a, b)
9:     Set-Children(y, x, c)
10:    if p = NIL then
11:        T ← y
12:    return T

13: function Right-Rotate(T, y)
14:    p ← Parent(y)
15:    x ← Left(y)                                   ▷ Assume x ≠ NIL
16:    a ← Left(x)
17:    b ← Right(x)
18:    c ← Right(y)
19:    Replace(y, x)
20:    Set-Children(y, b, c)
21:    Set-Children(x, a, y)
22:    if p = NIL then
23:        T ← x
24:    return T

25: function Set-Left(x, y)
26:    Left(x) ← y
27:    if y ≠ NIL then Parent(y) ← x

28: function Set-Right(x, y)
29:    Right(x) ← y
30:    if y ≠ NIL then Parent(y) ← x

31: function Set-Children(x, L, R)
32:    Set-Left(x, L)
33:    Set-Right(x, R)

34: function Replace(x, y)
35:    if Parent(x) = NIL then
36:        if y ≠ NIL then Parent(y) ← NIL
37:    else if Left(Parent(x)) = x then Set-Left(Parent(x), y)
38:    elseSet-Right(Parent(x), y)
39:    Parent(x) ← NIL
```

Compare these pseudo codes with the pattern matching functions, the former focus on the structure states changing, while the latter focus on the rotation process. As the title of this chapter indicated, red-black tree needn't be so complex as it was thought. Most traditional algorithm text books use the classic procedural way to teach red-black tree, there are several cases need to deal and all need carefulness to manipulate the node fields. However, by changing the mind to functional settings, things get intuitive and simple. Although there is some performance overhead.

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

3.2 Definition of red-black tree

Red-black tree is a type of self-balancing binary search tree[3]. [2] By using color changing and rotation, red-black tree provides a very simple and straightforward way to keep the tree balanced.

For a binary search tree, we can augment the nodes with a color field, a node can be colored either red or black. We call a binary search tree red-black tree if it satisfies the following 5 properties[3].

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

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

Please note the 4-th property, which means there won't be two adjacent red nodes. so the shortest path only contains black nodes, any paths longer than the shortest one has interval red nodes. According to property 5, all paths have the same number of black nodes, this finally ensure there won't be any path is 2 times longer than others[3]. Figure 3.4 shows an example red-black tree.

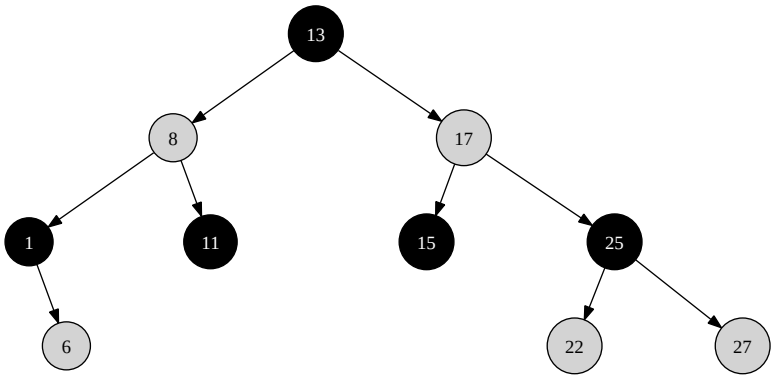


Figure 3.4: An example red-black tree

All read only operations such as search, min/max are as same as in binary search tree. While only the insertion and deletion are special.

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

As we have shown in word occurrence example, many implementation of set or map container are based on red-black tree. One example is the C++ Standard library (STL)[6].

As mentioned previously, the only change in data layout is that there is color information augmented to binary search tree. This can be represented as a data field in imperative languages such as C++ like below.

```
enum Color {Red, Black};

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

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

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

Exercise 3.2

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

3.3 Insertion

Inserting a new node as what has been mentioned in binary search tree may cause the tree unbalanced. The red-black properties has to be maintained, so we need do some fixing by transform the tree after insertion.

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

Functional and procedural implementation have different fixing methods. One is intuitive but has some overhead, the other is a bit complex but has higher performance. Most text books about algorithm introduce the later one. In this chapter, we focus on the former to show how easily a red-black tree insertion algorithm can be realized. The traditional procedural method will be given only for comparison purpose.

As described by Chris Okasaki, there are total 4 cases which violate property 4. All of them has 2 adjacent red nodes. However, they have a uniformed form after fixing[2] as shown in figure 4.3.

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

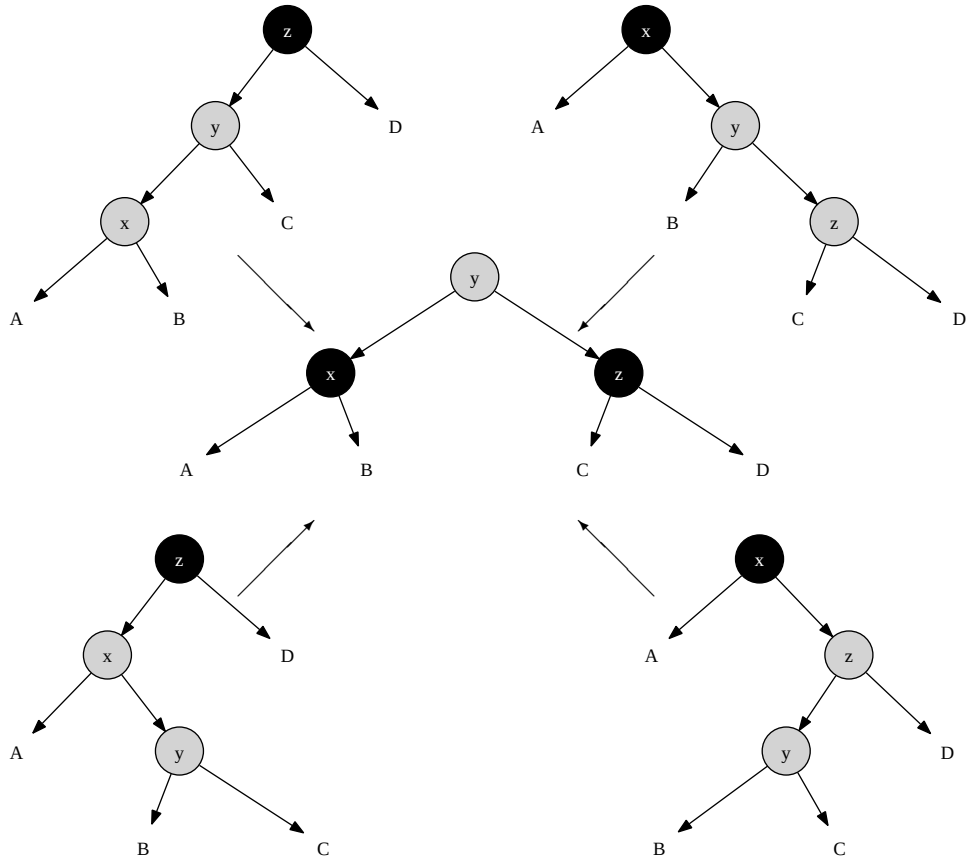


Figure 3.5: 4 cases for balancing a red-black tree after insertion

Observing that the 4 cases and the fixed result have strong pattern features, the fixing function can be defined by using the similar method we mentioned in tree rotation. To avoid too long formula, we abbreviate *Color* as $\mathcal{C}$, *Black* as $\mathcal{B}$, and *Red* as $\mathcal{R}$.

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

where function $node()$ can construct a red-black tree node with 4 parameters as color, the left child, the key and the right child. Function $match()$ can test if a tree is one of the 4 possible patterns as the following.

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

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

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

where

$$ins(T, k) = \begin{cases} node(\mathcal{R}, \phi, k, \phi) & : & T = \phi \\ balance(node(ins(L, k), Key, R)) & : & k < Key \\ balance(node(L, Key, ins(R, k))) & : & otherwise \end{cases} \tag{3.5}$$

L, R, Key represent the left child, right child and the key of a tree.

$$\begin{array}{l} L = left(T) \\ R = right(T) \\ Key = key(T) \end{array}$$

Function $makeBlack()$ is defined as the following, it forces the color of a non-empty tree to be black.

$$makeBlack(T) = node(\mathcal{B}, L, Key, R) \tag{3.6}$$

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

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

balance::Color → RBTree a → a → RBTree a → RBTree a
balance B (Node R (Node R a x b) y c) z d =
```

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

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

This program doesn't handle the case of inserting a duplicated key. However, it is possible to handle it either by overwriting, or skipping. Another option is to augment the data with a linked list[3].

Figure 3.6 shows two red-black trees built from feeding list 11, 2, 14, 1, 7, 15, 5, 8, 4 and 1, 2, ..., 8.

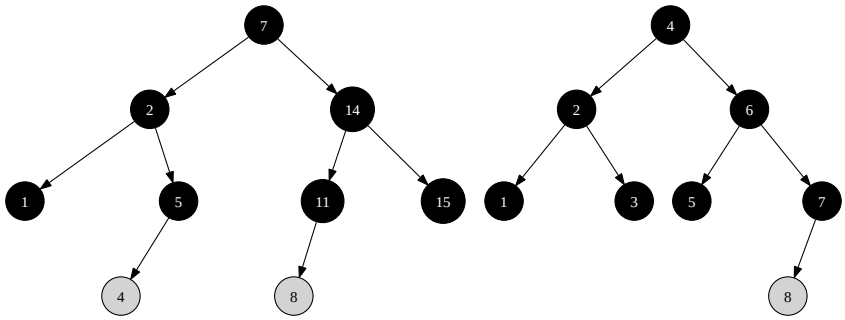


Figure 3.6: insert results generated by the Haskell algorithm

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

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

Exercise 3.3

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

3.4 Deletion

Remind the deletion section in binary search tree. Deletion is 'imperative only' for red-black tree as well. In typically practice, it often builds the tree just one time, and performs looking up frequently after that. Okasaki explained why he didn't provide red-black tree deletion in his work in [3]. One reason is that deletions are much messier than insertions.

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

Not only in functional settings, even in imperative settings, deletion is more complex than insertion. We face more cases to fix. Deletion may also violate the red black tree properties, so we need fix it after the normal deletion as described in binary search tree.

The deletion algorithm in this book are based on top of a handout in [5]. The problem only happens if you try to delete a black node, because it will violate the 4-th property of red-black tree, which means the number of black node in the path may decreased so that it is not uniformed black-height any more.

When delete a black node, we can resume red-black property number 4 by introducing a 'doubly-black' concept[3]. It means that the although the node is deleted, the blackness is kept by storing it in the parent node. If the parent node is red, it turns to black, However, if it has been already black, it turns to 'doubly-black'.

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

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

When deleting a node, we first perform the same deleting algorithm in binary search tree mentioned in previous chapter. After that, if the node to be sliced out is black, we need fix the tree to keep the red-black properties. Let's denote the empty tree as ϕ, and for non-empty tree, it can be decomposed to $node(Color, L, Key, R)$ for its color, left sub-tree, key and the right sub-tree. The delete function is defined as the following.

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

where

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

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

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

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

$$blackenRoot(T) = \begin{cases} \phi & : & T = \phi \\ node(\mathcal{B}, L, Key, R) & : & otherwise \end{cases} \tag{3.9}$$

Compare with the *makeBlack* function, which is defined in red-black tree insertion section, they are almost same, except for the case of empty tree. This is only valid in deletion, because insertion can't result an empty tree, while deletion may.

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

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

where Φ means doubly-black empty node and $\mathcal{B}^2$ is the doubly-black color.

Summarizing the above functions yields the following Haskell program.

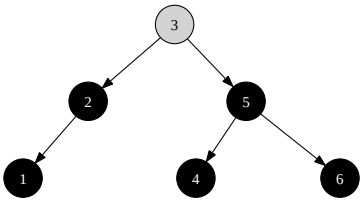
```
delete::(Ord a)⇒RBTree a → a → RBTree a
delete t x = blackenRoot(del t x) where
    del Empty _ = Empty
    del (Node color l k r) x
        | x < k = fixDB color (del l x) k r
        | x > k = fixDB color l k (del r x)
        -- x == k, delete this node
        | isEmpty l = if color==B then makeBlack r else r
```

```
        | isEmpty r = if color==B then makeBlack l else l
        | otherwise = fixDB color l k' (del r k') where k'= min r
    blackenRoot (Node _ l k r) = Node B l k r
    blackenRoot _ = Empty

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

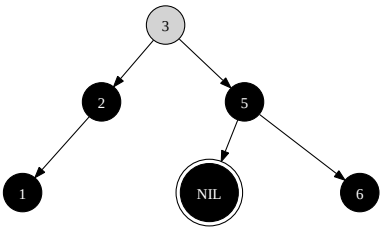
The final attack to the red-black tree deletion algorithm is to realize the $fixBlack^2$ function. The purpose of this function is to eliminate the 'doubly-black' colored node by rotation and color changing.

Let's solve the doubly-black empty node first. For any node, If one of its child is doubly-black empty, and the other child is non-empty, we can safely replace the doubly-black empty with a normal empty node.

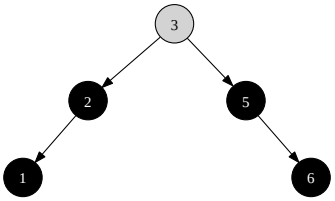
Like figure 3.7, if we are going to delete the node 4 from the tree (Instead show the whole tree, only part of the tree is shown), the program will use a doubly-black empty node to replace node 4. In the figure, the doubly-black node is shown in black circle with 2 edges. It means that for node 5, it has a doubly-black empty left child and has a right non-empty child (a leaf node with key 6). In such case we can safely change the doubly-black empty to normal empty node. which won't violate any red-black properties.



(a) Delete 4 from the tree.



(b) After 4 is sliced off, it is doubly-black empty.

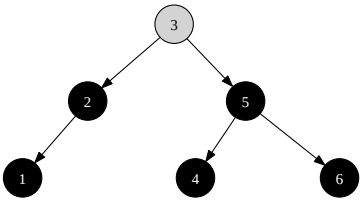


(c) We can safely change it to normal NIL.

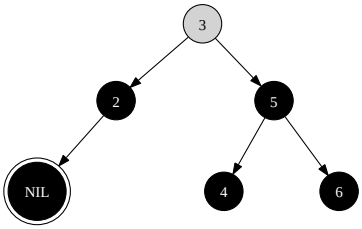
Figure 3.7: One child is doubly-black empty node, the other child is non-empty

On the other hand, if a node has a doubly-black empty node and the other child is empty, we have to push the doubly-blackness up one level. For example

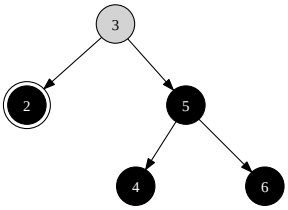
in figure 3.8, if we want to delete node 1 from the tree, the program will use a doubly-black empty node to replace 1. Then node 2 has a doubly-black empty node and has an empty right node. In such case we must mark node 2 as doubly-black after change its left child back to empty.



(a) Delete 1 from the tree.



(b) After 1 is sliced off, it is doubly-black empty.



(c) We must push the doubly-blackness up to node 2.

Figure 3.8: One child is doubly-black empty node, the other child is empty.

Based on above analysis, in order to fix the doubly-black empty node, we define the function partially like the following.

$$fixBlack^2(T) = \begin{cases} node(\mathcal{B}^2, \phi, Key, \phi) & : & (L = \phi \wedge R = \Phi) \vee (L = \Phi \wedge R = \phi) \\ node(\mathcal{C}, \phi, Key, R) & : & L = \Phi \wedge R \neq \phi \\ node(\mathcal{C}, L, Key, \phi) & : & R = \Phi \wedge L \neq \phi \\ \ldots & : & \ldots \end{cases} \tag{3.11}$$

After dealing with doubly-black empty node, we need to fix the case that the sibling of the doubly-black node is black and it has one red child. In this situation, we can fix the doubly-blackness with one rotation. Actually there are 4 different sub-cases, all of them can be transformed to one uniformed pattern. They are shown in the figure 3.9. These cases are described in [3] as case 3 and case 4.

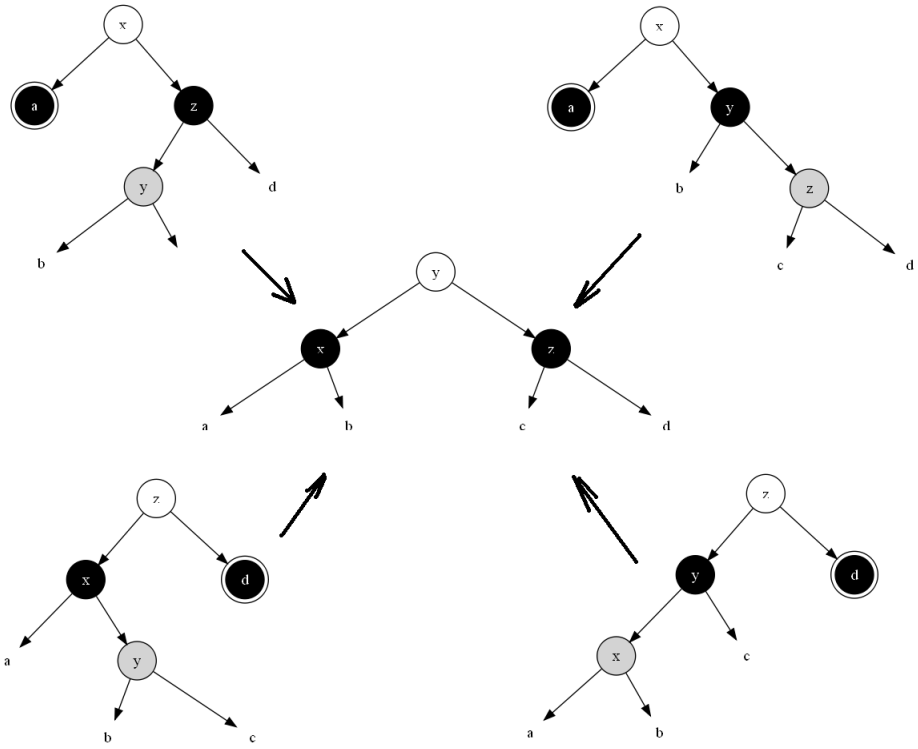


Figure 3.9: Fix the doubly black by rotation, the sibling of the doubly-black node is black, and it has one red child.

The handling of these 4 sub-cases can be defined on top of formula 3.11.

$$fixBlack^2(T) = \begin{cases} & \ldots & : & \ldots \\ node(\mathcal{C}, node(\mathcal{B}, mkBlk(A), x, B), y, node(\mathcal{B}, C, z, D)) & : & p1.1 \\ node(\mathcal{C}, node(\mathcal{B}, A, x, B), y, node(\mathcal{B}, C, z, mkBlk(D))) & : & p1.2 \\ & \ldots & : & \ldots \end{cases} \tag{3.12}$$

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

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

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

Besides the above cases, there is another one that not only the sibling of the doubly-black node is black, but also its two children are black. We can change the color of the sibling node to red; resume the doubly-black node to black and propagate the doubly-blackness one level up to the parent node as shown in figure 3.10. Note that there are two symmetric sub-cases. This case is described in [3] as case 2.

We go on adding this fixing after formula 3.12.

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

where $p2.1$ and $p2.2$ are two patterns as below.

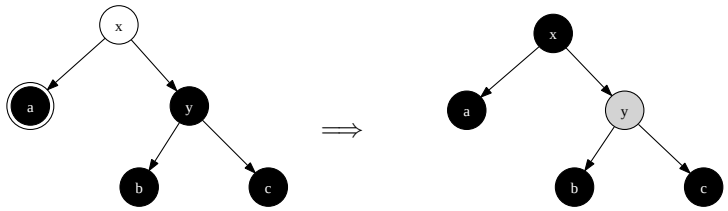
$$p2.1 = \left\{ \begin{array}{l} node(\mathcal{C}, A, x, node(\mathcal{B}, B, y, C)) \wedge \\ Color(A) = \mathcal{B}^2 \wedge Color(B) = Color(C) = \mathcal{B} \end{array} \right\}$$

$$p2.2 = \left\{ \begin{array}{l} node(\mathcal{C}, node(\mathcal{B}, A, x, B), y, C) \wedge \\ Color(C) = \mathcal{B}^2 \wedge Color(A) = Color(B) = \mathcal{B} \end{array} \right\}$$

There is a final case left, that the sibling of the doubly-black node is red. We can do a rotation to change this case to pattern $p1.1$ or $p1.2$. Figure 3.11 shows about it.

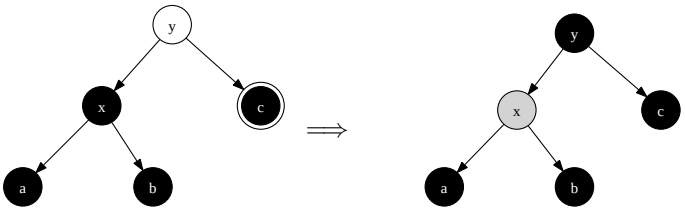
We can finish formula 3.13 with 3.14.

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



(a) Color of x can be either black or red.

(b) If x was red, then it becomes black, otherwise, it becomes doubly-black.



(c) Color of y can be either black or red.

(d) If y was red, then it becomes black, otherwise, it becomes doubly-black.

Figure 3.10: propagate the blackness up.

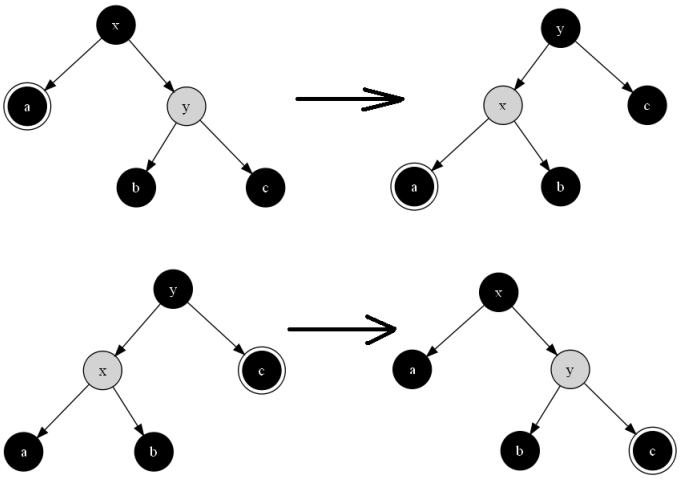


Figure 3.11: The sibling of the doubly-black node is red.

where $p3.1$ and $p3.2$ are two patterns as the following.

$$p3.1 = \{Color(T) = \mathcal{B} \wedge Color(L) = \mathcal{B}^2 \wedge Color(R) = \mathcal{R}\}$$

$$p3.2 = \{Color(T) = \mathcal{B} \wedge Color(L) = \mathcal{R} \wedge Color(R) = \mathcal{B}^2\}$$

This two cases are described in [3] as case 1.

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

Put formula 3.11, 3.12, 3.13, and 3.14 together, we can write the final Haskell program.

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

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

Exercise 3.4

- As we mentioned in this section, deletion can be implemented by just marking the node as deleted without actually removing it. Once the number of marked nodes exceeds 50% of the total node number, a tree re-build is performed. Try to implement this method in your favorite programming language.

3.5 Imperative red-black tree algorithm ⋆

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

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

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

The only difference from the binary search tree insertion algorithm is that we set the color of the new node as red, and perform fixing before return. It is easy to translate the pseudo code to real imperative programming language, for instance Python [3].

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

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

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

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

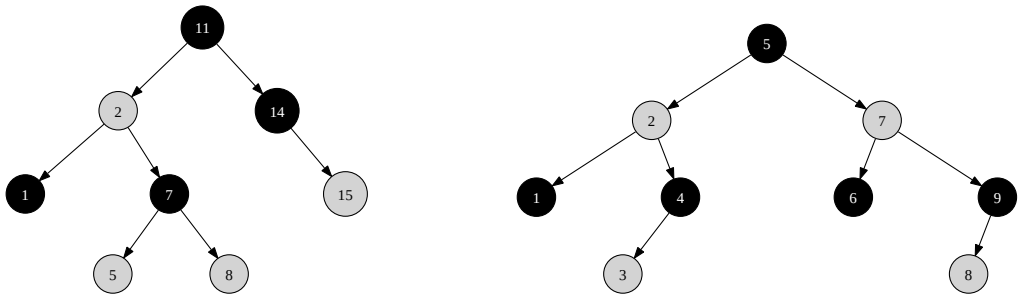
This program takes $O(\lg N)$ time to insert a new key to the red-black tree. Compare this pseudo code and the *balance* function we defined in previous section, we can see the difference. They differ not only in terms of simplicity, but also in logic. Even if we feed the same series of keys to the two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

Translate the above algorithm to Python yields the below program.

```
# Fix the red→red violation
def rb_insert_fix(t, x):
    while(x.parent and x.parent.color==RED):
        if x.uncle().color == RED:
            #case 1: ((a:R x:R b) y:B c:R) ⟹ ((a:R x:B b) y:R c:B)
            set_color([x.parent, x.grandparent(), x.uncle()],
                      [BLACK, RED, BLACK])
            x = x.grandparent()
        else:
            if x.parent == x.grandparent().left:
                if x == x.parent.right:
```

```
                #case 2: ((a x:R b:R) y:B c) ⟹ case 3
                x = x.parent
                t=left_rotate(t, x)
            # case 3: ((a:R x:R b) y:B c) ⟹ (a:R x:B (b y:R c))
            set_color([x.parent, x.grandparent()], [BLACK, RED])
            t=right_rotate(t, x.grandparent())
        else:
            if x == x.parent.left:
                #case 2': (a x:B (b:R y:R c)) ⟹ case 3'
                x = x.parent
                t = right_rotate(t, x)
            # case 3': (a x:B (b y:R c:R)) ⟹ ((a x:R b) y:B c:R)
            set_color([x.parent, x.grandparent()], [BLACK, RED])
            t=left_rotate(t, x.grandparent())
    t.color = BLACK
    return t
```

Figure 3.12 shows the results of feeding same series of keys to the above python insertion program. Compare them with figure 3.6, one can tell the difference clearly.



(a) (b)

Figure 3.12: Red-black trees created by imperative algorithm.

We skip the red-black tree delete algorithm in imperative settings, because it is even more complex than the insertion. The implementation of deleting is left as an exercise of this chapter.

Exercise 3.5

- Implement the red-black tree deleting algorithm in your favorite imperative programming language. you can refer to [3] for algorithm details.

3.6 More words

Red-black tree is the most popular implementation of balanced binary search tree. Another one is the AVL tree, which we'll introduce in next chapter. Red-black tree can be a good start point for more data structures. If we extend the

number of children from 2 to K, and keep the balance as well, it leads to B-tree, If we store the data along with edge but not inside node, it leads to Tries. However, the multiple cases handling and the long program tends to make new comers think red-black tree is complex.

Okasaki's work helps making the red-black tree much easily understand. There are many implementation in other programming languages in that manner [7]. It's also inspired me to find the pattern matching solution for Splay tree and AVL tree etc.

Bibliography

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

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

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

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

[5] Lyn Turbak. "Red-Black Trees". cs.wellesley.edu/ cs231/fall01/red-black.pdf Nov. 2, 2001.

[6] SGI STL. http://www.sgi.com/tech/stl/

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

AVL tree

Liu Xinyu **Liu Xinyu**

Email: liuxinyu95@gmail.com

Chapter 4

AVL tree

4.1 Introduction

4.1.1 How to measure the balance of a tree?

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

$$\delta(T) = |L| - |R| \tag{4.1}$$

Where $|T|$ means the height of tree T, and L, R denotes the left sub-tree and right sub-tree.

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

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

4.2 Definition of AVL tree

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

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

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

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

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

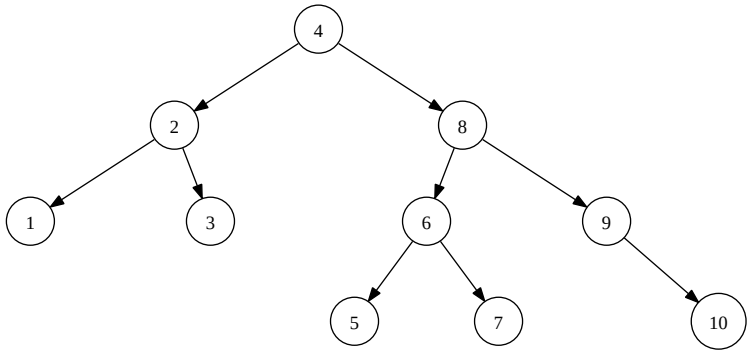


Figure 4.1: An example AVL tree

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

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

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

$$h = max(height(L), height(R)) + 1 \tag{4.3}$$

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

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

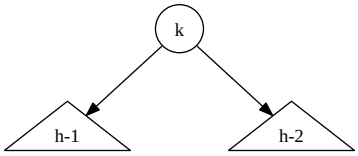


Figure 4.2: An AVL tree with height h, one of the sub-tree with height $h - 1$, the other is $h - 2$

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

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

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

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

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

Proof. For the trivial case, we have

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

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

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

□

From Lemma 4.2.1, we immediately get

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

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

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

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

Based on this idea, we can add one data field δ to the original binary search tree as the following C++ code example [1].

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

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

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

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

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

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

4.3 Insertion

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

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

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

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

where

$$ins(T, k) = \begin{cases} (node(\phi, k, \phi, 0), 1) & : \quad T = \phi \\ (tree(ins(L, k), Key, (R, 0)), \Delta) & : \quad k < Key \\ (tree((L, 0), Key, ins(R, k)), \Delta) & : \quad otherwise \end{cases} \tag{4.9}$$

L, R, Key, Δ represent the left child, right child, the key and the balance factor of a tree.

$$\begin{aligned} L &= left(T) \\ R &= right(T) \\ Key &= key(T) \\ \Delta &= \delta(T) \end{aligned}$$

When we insert a new key k to a AVL tree T, if the tree is empty, we just need create a leaf node with k, set the balance factor as 0, and the height is increased by one. This is the trivial case. Function $node()$ is defined to build a tree by taking a left sub-tree, a right sub-tree, a key and a balance factor.

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

As we defined above, the result of the recursive insertion is a pair like $(L', \Delta H_l)$, we need do balancing adjustment as well as updating the increment of height. Function $tree()$ is defined to dealing with this task. It takes 4 parameters as $(L', \Delta H_l)$, Key, $(R', \Delta H_r)$, and Δ. The result of this function is defined as $(T', \Delta H)$, where T' is the new tree after adjustment, and ΔH is the new increment of height which is defined as

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

This can be further detailed deduced in 4 cases.

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

To prove this equation, note the fact that the height can't increase both in left and right with only one insertion.

These 4 cases can be explained from the definition of balance factor definition that it equal to the difference from the right sub tree and left sub tree.

- If $\Delta \geq 0$ and $\Delta' \geq 0$, it means that the height of right sub tree isn't less than the height of left sub tree both before insertion and after insertion. In this case, the increment in height of the tree is only 'contributed' from the right sub tree, which is ΔH_r.

- If $\Delta \leq 0$, which means the height of left sub tree isn't less than the height of right sub tree before, and it becomes $\Delta' \geq 0$, which means that the height of right sub tree increases due to insertion, and the left side keeps same ($|L'| = |L|$). So the increment in height is

$$\begin{aligned}
\Delta H &= max(|R'|, |L'|) - max(|R|, |L|) && \{\Delta \leq 0 \wedge \Delta' \geq 0\} \\
&= |R'| - |L'| && \{|L| = |L'|\} \\
&= |R| + \Delta H_r - |L| \\
&= \Delta + \Delta H_r
\end{aligned}$$

- For the case $\Delta \geq 0 \wedge \Delta' \leq 0$, Similar as the second one, we can get.

$$\begin{aligned}
\Delta H &= max(|R'|, |L'|) - max(|R|, |L|) && \{\Delta \geq 0 \wedge \Delta' \leq 0\} \\
&= |L'| - |R| \\
&= |L| + \Delta H_l - |R| \\
&= \Delta H_l - \Delta
\end{aligned}$$

- For the last case, the both Δ and Δ' is no bigger than zero, which means the height left sub tree is always greater than or equal to the right sub tree, so the increment in height is only 'contributed' from the right sub tree, which is ΔH_l.

The next problem in front of us is how to determine the new balancing factor value Δ' before performing balancing adjustment. According to the definition of AVL tree, the balancing factor is the height of right sub tree minus the height of right sub tree. We have the following facts.

$$\begin{aligned}\Delta' &= |R'| - |L'| \\ &= |R| + \Delta H_r - (|L| + \Delta H_l) \\ &= |R| - |L| + \Delta H_r - \Delta H_l \\ &= \Delta + \Delta H_r - \Delta H_l\end{aligned} \tag{4.12}$$

With all these changes in height and balancing factor get clear, it's possible to define the $tree()$ function mentioned in (4.9).

$$tree((L', \Delta H_l), Key, (R', \Delta H_r), \Delta) = balance(node(L', Key, R', \Delta'), \Delta H) \tag{4.13}$$

Before we moving into details of balancing adjustment, let's translate the above equations to real programs in Haskell.

First is the insert function.

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

Here we also handle the case that inserting a duplicated key (which means the key has already existed.) as just overwriting.

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

And the definition of height increment is as below.

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

4.3.1 Balancing adjustment

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

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

We call these four cases left-left lean, right-right lean, right-left lean, and left-right lean cases in clock-wise direction from top-left. We denote the balancing

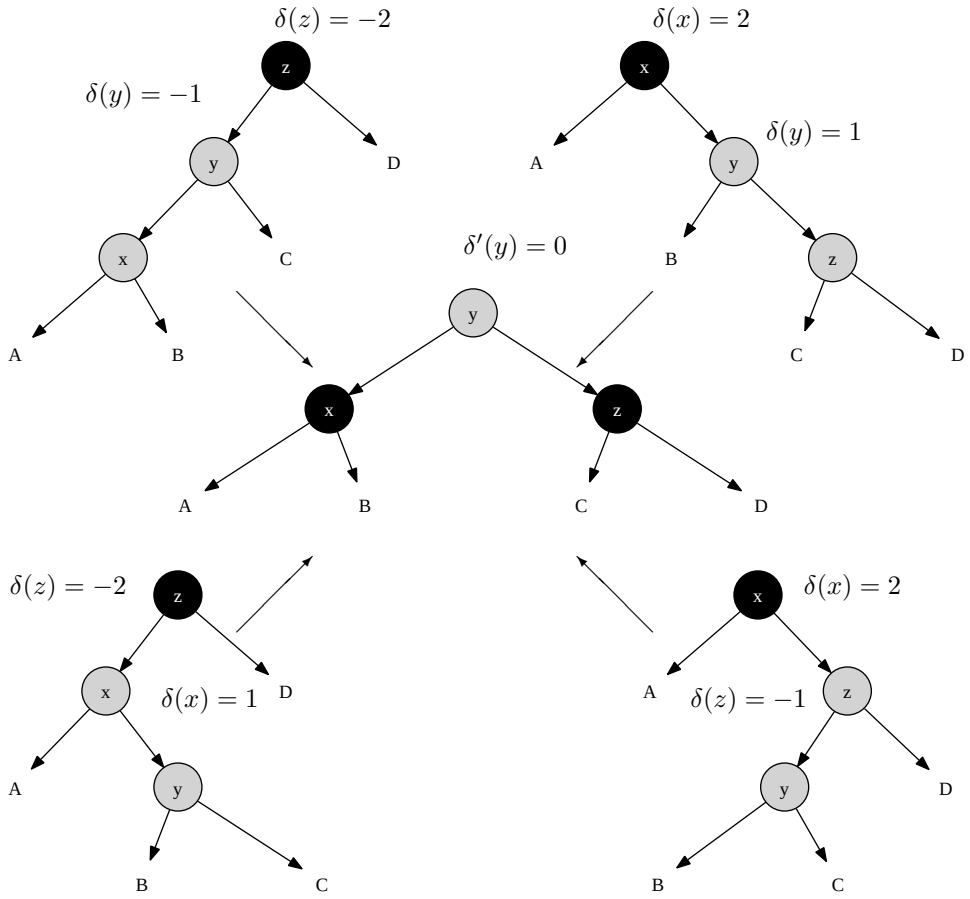


Figure 4.3: 4 cases for balancing a AVL tree after insertion

factor before fixing as $\delta(x), \delta(y)$, and $\delta(z)$, while after fixing, they changes to $\delta'(x), \delta'(y)$, and $\delta'(z)$ respectively.

We'll next prove that, after fixing, we have $\delta(y) = 0$ for all four cases, and we'll provide the result values of $\delta'(x)$ and $\delta'(z)$.

Left-left lean case

As the structure of sub tree x doesn't change due to fixing, we immediately get $\delta'(x) = \delta(x)$.

Since $\delta(y) = -1$ and $\delta(z) = -2$, we have

$$\begin{aligned} \delta(y) &= |C| - |x| = -1 \Rightarrow |C| = |x| - 1 \\ \delta(z) &= |D| - |y| = -2 \Rightarrow |D| = |y| - 2 \end{aligned} \tag{4.14}$$

After fixing.

$$\begin{aligned} \delta'(z) &= |D| - |C| && \{From(4.14)\} \\ &= |y| - 2 - (|x| - 1) \\ &= |y| - |x| - 1 && \{x \text{ is child of } y \Rightarrow |y| - |x| = 1\} \\ &= 0 \end{aligned} \tag{4.15}$$

For $\delta'(y)$, we have the following fact after fixing.

$$\begin{aligned} \delta'(y) &= |z| - |x| \\ &= 1 + max(|C|, |D|) - |x| && \{\text{By (4.15), we have} |C| = |D|\} \\ &= 1 + |C| - |x| && \{\text{By (4.14)}\} \\ &= 1 + |x| - 1 - |x| \\ &= 0 \end{aligned} \tag{4.16}$$

Summarize the above results, the left-left lean case adjust the balancing factors as the following.

$$\begin{aligned} \delta'(x) &= \delta(x) \\ \delta'(y) &= 0 \\ \delta'(z) &= 0 \end{aligned} \tag{4.17}$$

Right-right lean case

Since right-right case is symmetric to left-left case, we can easily achieve the result balancing factors as

$$\begin{aligned} \delta'(x) &= 0 \\ \delta'(y) &= 0 \\ \delta'(z) &= \delta(z) \end{aligned} \tag{4.18}$$

Right-left lean case

First let's consider $\delta'(x)$. After balance fixing, we have.

$$\delta'(x) = |B| - |A| \tag{4.19}$$

Before fixing, if we calculate the height of z, we can get.

$$\begin{array}{ll} |z| & = 1 + max(|y|, |D|) \quad \{\delta(z) = -1 \Rightarrow |y| > |D|\} \\ & = 1 + |y| \\ & = 2 + max(|B|, |C|) \end{array} \tag{4.20}$$

While since $\delta(x) = 2$, we can deduce that.

$$\begin{array}{lll} \delta(x) = 2 & \Rightarrow |z| - |A| = 2 & \{\text{By (4.20)}\} \\ & \Rightarrow 2 + max(|B|, |C|) - |A| = 2 & \\ & \Rightarrow max(|B|, |C|) - |A| = 0 & \end{array} \tag{4.21}$$

If $\delta(y) = 1$, which means $|C| - |B| = 1$, it means

$$max(|B|, |C|) = |C| = |B| + 1 \tag{4.22}$$

Take this into (4.21) yields

$$\begin{array}{l} |B| + 1 - |A| = 0 \Rightarrow |B| - |A| = -1 \quad \{\text{By (4.19) }\} \\ \Rightarrow \delta'(x) = -1 \end{array} \tag{4.23}$$

If $\delta(y) \neq 1$, it means $max(|B|, |C|) = |B|$, taking this into (4.21), yields.

$$\begin{array}{l} |B| - |A| = 0 \quad \{\text{By (4.19)}\} \\ \Rightarrow \delta'(x) = 0 \end{array} \tag{4.24}$$

Summarize these 2 cases, we get relationship of $\delta'(x)$ and $\delta(y)$ as the following.

$$\delta'(x) = \begin{cases} -1 & : \quad \delta(y) = 1 \\ 0 & : \quad otherwise \end{cases} \tag{4.25}$$

For $\delta'(z)$ according to definition, it is equal to.

$$\begin{array}{lll} \delta'(z) & = |D| - |C| & \{\delta(z) = -1 = |D| - |y|\} \\ & = |y| - |C| - 1 & \{|y| = 1 + max(|B|, |C|)\} \\ & = max(|B|, |C|) - |C| & \end{array} \tag{4.26}$$

If $\delta(y) = -1$, then we have $|C| - |B| = -1$, so $max(|B|, |C|) = |B| = |C| + 1$. Takes this into (4.26), we get $\delta'(z) = 1$.

If $\delta(y) \neq -1$, then $max(|B|, |C|) = |C|$, we get $\delta'(z) = 0$.

Combined these two cases, the relationship between $\delta'(z)$ and $\delta(y)$ is as below.

$$\delta'(z) = \begin{cases} 1 & : \quad \delta(y) = -1 \\ 0 & : \quad otherwise \end{cases} \tag{4.27}$$

Finally, for $\delta'(y)$, we deduce it like below.

$$\begin{array}{ll} \delta'(y) & = |z| - |x| \\ & = max(|C|, |D|) - max(|A|, |B|) \end{array} \tag{4.28}$$

There are three cases.

- If $\delta(y) = 0$, it means $|B| = |C|$, and according to (4.25) and (4.27), we have $\delta'(x) = 0 \Rightarrow |A| = |B|$, and $\delta'(z) = 0 \Rightarrow |C| = |D|$, these lead to $\delta'(y) = 0$.
- If $\delta(y) = 1$, From (4.27), we have $\delta'(z) = 0 \Rightarrow |C| = |D|$.

$$\begin{aligned} \delta'(y) &= max(|C|, |D|) - max(|A|, |B|) && \{|C| = |D|\} \\ &= |C| - max(|A|, |B|) && \{\text{From (4.25): } \delta'(x) = -1 \Rightarrow |B| - |A| = -1\} \\ &= |C| - (|B| + 1) && \{\delta(y) = 1 \Rightarrow |C| - |B| = 1\} \\ &= 0 \end{aligned}$$

- If $\delta(y) = -1$, From (4.25), we have $\delta'(x) = 0 \Rightarrow |A| = |B|$.

$$\begin{aligned} \delta'(y) &= max(|C|, |D|) - max(|A|, |B|) && \{|A| = |B|\} \\ &= max(|C|, |D|) - |B| && \{\text{From (4.27): } |D| - |C| = 1\} \\ &= |C| + 1 - |B| && \{\delta(y) = -1 \Rightarrow |C| - |B| = -1\} \\ &= 0 \end{aligned}$$

Both three cases lead to the same result that $\delta'(y) = 0$.

Collect all the above results, we get the new balancing factors after fixing as the following.

$$\begin{aligned} \delta'(x) &= \begin{cases} -1 & : \quad \delta(y) = 1 \\ 0 & : \quad otherwise \end{cases} \\ \delta'(y) &= 0 \\ \delta'(z) &= \begin{cases} 1 & : \quad \delta(y) = -1 \\ 0 & : \quad otherwise \end{cases} \end{aligned} \tag{4.29}$$

Left-right lean case

Left-right lean case is symmetric to the Right-left lean case. By using the similar deduction, we can find the new balancing factors are identical to the result in (4.29).

4.3.2 Pattern Matching

All the problems have been solved and it's time to define the final pattern matching fixing function.

$$balance(T, \Delta H) = \begin{cases} (node(node(A, x, B, \delta(x)), y, node(C, z, D, 0), 0), 0) & : \quad P_{ll}(T) \\ (node(node(A, x, B, 0), y, node(C, z, D, \delta(z)), 0), 0) & : \quad P_{rr}(T) \\ (node(node(A, x, B, \delta'(x)), y, node(C, z, D, \delta'(z)), 0), 0) & : \quad P_{rl}(T) \vee P_{lr}(T) \\ (T, \Delta H) & : \quad otherwise \end{cases} \tag{4.30}$$

Where $P_{ll}(T)$ means the pattern of tree T is left-left lean respectively. $\delta'(x)$ and $delta'(z)$ are defined in (4.29). The four patterns are tested as below.

$$\begin{aligned} P_{ll}(T) &= node(node(node(A, x, B, \delta(x)), y, C, -1), z, D, -2) \\ P_{rr}(T) &= node(A, x, node(B, y, node(C, z, D, \delta(z)), 1), 2) \\ P_{rl}(T) &= node(node(A, x, node(B, y, C, \delta(y)), 1), z, D, -2) \\ P_{lr}(T) &= node(A, x, node(node(B, y, C, \delta(y)), z, D, -1), 2) \end{aligned} \tag{4.31}$$

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

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

The insertion algorithm takes time proportion to the height of the tree, and according to the result we proved above, its performance is $O(\lg N)$ where N is the number of elements stored in the AVL tree.

Verification

One can easily create a function to verify a tree is AVL tree. Actually we need verify two things, first, it's a binary search tree; second, it satisfies AVL tree property.

We left the first verification problem as an exercise to the reader.

In order to test if a binary tree satisfies AVL tree property, we can test the difference in height between its two children, and recursively test that both children conform to AVL property until we arrive at an empty leaf.

$$avl?(T) = \begin{cases} True & : \quad T = \Phi \\ avl?(L) \wedge avl?(R) \wedge ||R| - |L|| \leq 1 & : \quad otherwise \end{cases} \tag{4.32}$$

And the height of a AVL tree can also be calculate from the definition.

$$|T| = \begin{cases} 0 & : \quad T = \Phi \\ 1 + max(|R|, |L|) & : \quad otherwise \end{cases} \tag{4.33}$$

The corresponding Haskell program is given as the following.

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

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

Exercise 4.1

Write a program to verify a binary tree is a binary search tree in your favorite programming language. If you choose to use an imperative language, please consider realize this program without recursion.

4.4 Deletion

As we mentioned before, deletion doesn't make significant sense in purely functional settings. As the tree is read only, it's typically performs frequently looking up after build.

Even if we implement deletion, it's actually re-building the tree as we presented in chapter of red-black tree. We left the deletion of AVL tree as an exercise to the reader.

Exercise 4.2

- Take red-black tree deletion algorithm as an example, write the AVL tree deletion program in purely functional approach in your favorite programming language.
- Write the deletion algorithm in imperative approach in your favorite programming language.

4.5 Imperative AVL tree algorithm ⋆

We almost finished all the content in this chapter about AVL tree. However, it necessary to show the traditional insert-and-rotate approach as the comparator to pattern matching algorithm.

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

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

Note that after insertion, the height of the tree may increase, so that the balancing factor δ may also change, insert on right side will increase δ by 1,

while insert on left side will decrease it. By the end of this algorithm, we need perform bottom-up fixing from node x towards root.

We can translate the pseudo code to real programming language, such as Python [2].

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

This is a top-down algorithm search the tree from root down to the proper position and insert the new key as a leaf. By the end of this algorithm, it calls fixing procedure, by passing the root and the new node inserted.

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

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

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

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

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

```
10:         L ← LEFT(x)
11:         R ← RIGHT(x)
12:         if |δ| = 1 and |δ'| = 0 then          ▷ Height doesn't change, terminates.
13:             return T
14:         else if |δ| = 0 and |δ'| = 1 then          ▷ Go on bottom-up updating.
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 case
19:                     δ(P) ← 0                                          ▷ By (4.18)
20:                     δ(R) ← 0
21:                     T ← LEFT-ROTATE(T, P)
22:                 if δ(R) = −1 then                          ▷ Right-left case
23:                     δ_y ← δ(LEFT(R))                                  ▷ By (4.29)
24:                     if δ_y = 1 then
25:                         δ(P) ← −1
26:                     else
27:                         δ(P) ← 0
28:                     δ(LEFT(R)) ← 0
29:                     if δ_y = −1 then
30:                         δ(R) ← 1
31:                     else
32:                         δ(R) ← 0
33:                     T ← RIGHT-ROTATE(T, R)
34:                     T ← LEFT-ROTATE(T, P)
35:             if δ' = −2 then
36:                 if δ(L) = −1 then                          ▷ Left-left case
37:                     δ(P) ← 0
38:                     δ(L) ← 0
39:                     RIGHT-ROTATE(T, P)
40:                 else                                       ▷ Left-Right case
41:                     δ_y ← δ(RIGHT(L))
42:                     if δ_y = 1 then
43:                         δ(L) ← −1
44:                     else
45:                         δ(L) ← 0
46:                     δ(RIGHT(L)) ← 0
47:                     if δ_y = −1 then
48:                         δ(P) ← 1
49:                     else
50:                         δ(P) ← 0
51:                     LEFT-ROTATE(T, L)
52:                     RIGHT-ROTATE(T, P)
53:             break
54:     return T
```

Here we reuse the rotation algorithms mentioned in red-black tree chapter. Rotation operation doesn't update balancing factor δ at all, However, since rotation changes (actually improves) the balance situation we should update

these factors. Here we refer the results from above section. Among the four cases, right-right case and left-left case only need one rotation, while right-left case and left-right case need two rotations.

The relative python program is shown as the following.

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

```
                t = right_rotate(t, p)
            break
    return t
```

We skip the AVL tree deletion algorithm and left this as an exercise to the reader.

4.6 Chapter note

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

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

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

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

Bibliography

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

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

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

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

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

Trie and Patricia with Functional and imperative implementation

Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

Chapter 5

Trie and Patricia with Functional and imperative implementation

5.1 abstract

Trie and Patricia are important data structures in information retrieving and manipulating. None of these data structures are new. They were invented in 1960s. This post collects some existing knowledge about them. Some functional and imperative implementation are given in order to show the basic idea of these data structures. There are multiple programming languages used, including, C++, Haskell, python and scheme/lisp. C++ and python are mostly used to show the imperative implementation, while Haskell and Scheme are used for functional purpose.

There may be mistakes in the post, please feel free to point out.

Keywords: Trie, Patricia, Radix tree

5.2 Introduction

There isn't a separate chapter about Trie or Patricia in CLRS book. While these data structure are very basic, especially in information retrieving. Some of them are also widely used in compiler design[2], and bio-information area, such as DNA pattern matching [2].

In CLRS book index, Trie is redirected to Radix tree, while Radix tree is described in Problem 12-1 [3].

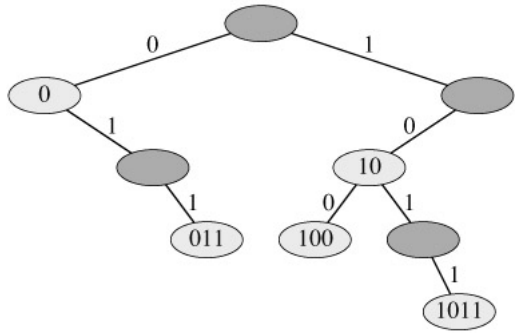


Figure 5.1: an Radix tree example in CLRS

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

Note that radix tree needn't store key in node at all. The information are represented by edges in fact. The node with key string in the above figure are only for illustration.

It is very natural to come to the idea 'is it possible to represent keys in integers instead of string, because integer can be denoted in binary format?'. Such approach can save spaces and it is fast if we can use bit-wise manipulation.

I'll first show the integer based Trie data structure and implementation in section 5.3. Then we can point out the drawbacks and go to the improved data structure of integer based Patricia in section 5.4. After that, I'll show alphabetic Trie and Patricia and list some typical use of them in textual manipulation engineering problems.

This article provides example implementation of Trie and Patricia in C, C++, Haskell, Python, and Scheme/Lisp languages. Some functional implementation can be referenced from current Haskell packages **?? ??**.

All source code can be downloaded in appendix 8.7, please refer to appendix for detailed information about build and run.

5.3 Integer Trie

Let's give a definition of the data structure in figure 5.1. To be more accurate, it should be called as *binary trie*. a binary trie is a binary tree in which the placement of each key is controlled by its bits, each 0 means "go left at the next node" and each 1 means "go right at the next node"[2].

Because integers can be represented in binary format in computer, it is possible to store integer keys rather than 0,1 strings. When we insert an integer as a new key to the trie, we take first bit, if it is 0, we recursively insert the other bits to the left sub tree; if it is 1, we insert into right sub tree.

However, there is a problem if we treat the key as integer. Consider a binary trie shows in figure 5.2. If the keys are represented in string based on '0' and '1', all the three keys are different. While if they are turned into integers, they are identical. So if we want to insert a data with integer key 3, where should we put it into the trie?

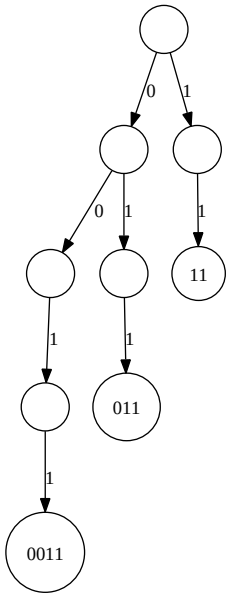


Figure 5.2: a big-endian trie

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

Chris Okasaki shows a method to solve this problem in [2]. Instead of normal big-endian integer, we can use little-endian integer as key. By using little-endian, decimal integer 1 is represent as binary 1, if we insert it to an empty binary trie, we get a trie with a root and a right leaf. There is only 1 level. For integer 3, it is 11 in binary, we needn't add any prefix 0, the position in the trie is unique.

5.3.1 Definition of Integer Trie

Trie is invented by Edward Fredkin. It comes from "retrieval", pronounce as /'tri:/ by the inventor, while it is pronounced /'trai/ "try" by other authors [4]. The definition of little-endian binary trie is simple, we can reuse the structure of binary tree, with its left sub tree to store 0 part and right tree to store 1 part. The augment data can be stored as value.

Definition of little-endian integer Trie in C++

We can utilize C++ template to abstract the value stored in Trie. The type of key is integer. Each node contains a value, a left child and a right child.

```
template<class T>
struct IntTrie{
  IntTrie():value(), left(0), right(0){}
  ~IntTrie(){
    delete left;
    delete right;
  }
  T value;
  IntTrie* left;
  IntTrie* right;
};
```

In order to simplify the release, recursive destruction is added.

Definition of little-endian integer Trie in Haskell

In trie, since a node may not contains value, so we use Haskell Maybe data to represent this situation. A IntTrie node is either an empty node, or a branch node. The branch node contains a left child a 'Maybe' value and a right child.

```
data IntTrie a = Empty
               | Branch (IntTrie a) (Maybe a) (IntTrie a) -- left, value, right

type Key = Int

-- helpers
left :: IntTrie a → IntTrie a
left (Branch l _ _) = l
left Empty = Empty

right :: IntTrie a → IntTrie a
right (Branch _ _ r) = r
right Empty = Empty

value :: IntTrie a → Maybe a
value (Branch _ v _) = v
value Empty = Nothing
```

In order to access the children and value some helper functions are given.

Definition of little-endian integer Trie in Python

The definition of integer trie in Python is shown as below. All fields are initialized as empty values.

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

left child and right child represent sub Trie branches, and value is used to store actual data.

Definition of little-endian integer Trie in Scheme/Lisp

In Scheme/Lisp, we provides some helper functions to create and access Trie data. The underground data structure is still list.

```
;; Definition

(define (make-int-trie l v r) ;; left, value, right
  (list l v r))

;; Helpers
(define (left trie)
  (if (null? trie) '() (car trie)))

(define (value trie)
  (if (null? trie) '() (cadr trie)))

(define (right trie)
  (if (null? trie) '() (caddr trie)))
```

Some helper functions are provided for easy access to children and value.

5.3.2 Insertion of integer trie

Iterative insertion algorithm

Since the key is little-endian, when we insert a key into trie, we take the bit from right most (LSB). If it is 0, we go to the left child, and go to right for 1. if the child is empty, we need create new node, and repeat this until meet the last bit (MSB) of the integer. Below is the iterative algorithm of insertion.

1: **procedure** INT-TRIE-INSERT($T, x, data$)
2: **if** $T = NIL$ **then**
3: $T \leftarrow EmptyNode$
4: $p \leftarrow T$
5: **while** $x \neq 0$ **do**
6: **if** $EVEN(x) = TRUE$ **then**
7: $p \leftarrow LEFT(p)$
8: **else**
9: $p \leftarrow RIGHT(p)$
10: **if** $p = NIL$ **then**
11: $p \leftarrow EmptyNode$
12: $x \leftarrow x/2$
13: $DATA(p) \leftarrow data$

Insertion of integer Trie in C++

With C++ language we can speed up the above even/odd test and key update with bit-wise operation.

```
template<class T>
IntTrie<T>* insert(IntTrie<T>* t, int key, T value=T()){
  if(!t)
    t = new IntTrie<T>();

  IntTrie<T>* p = t;
  while(key){
    if( (key&0x1) == 0){
```

```
      if(!p→left) p→left = new IntTrie<T>();
      p = p→left;
    }
    else{
      if(!p→right) p→right = new IntTrie<T>();
      p = p→right;
    }
    key>>=1;
  }
  p→value = value;
  return t;
}
```

In order to verify this program, some helper functions are provided to simplify the repeat insertions. And we also provide a function to convert the Trie to readable string.

```
template<class T, class Iterator>
IntTrie<T>* list_to_trie(Iterator first, Iterator last){
  IntTrie<T>* t(0);
  for(;first!=last; ++first)
    t = insert(t, *first);
  return t;
}

template<class T, class Iterator>
IntTrie<T>* map_to_trie(Iterator first, Iterator last){
  IntTrie<T>* t(0);
  for(;first!=last; ++first)
    t = insert(t, first→first, first→second);
  return t;
}

template<class T>
std::string trie_to_str(IntTrie<T>* t, int prefix=0, int depth=0){
  std::stringstream s;
  s<<"("<<prefix;
  if(t→value!=T())
    s<<":"<<t→value;
  if(t→left)
    s<<",␣"<<trie_to_str(t→left, prefix, depth+1);
  if(t→right)
    s<<",␣"<<trie_to_str(t→right, (1<<depth)+prefix, depth+1);
  s<<")";
  return s.str();
}
```

Function list_to_trie just inserts keys, all values are treated as default value as type T, while map_to_trie inserts both keys and values repeatedly. Function trie_to_str helps to convert a Trie to literal string in a modified pre-order traverse.

The verification cases are as the following.

```
const int lst[] = {1, 4, 5};
std::list<int> l(lst, lst+sizeof(lst)/sizeof(int));

IntTrie<int>* ti = list_to_trie<int, std::list<int>::iterator>(l.begin(), l.end());
```

```
std::copy(l.begin(), l.end(),
          std::ostream_iterator<int>(std::cout, ",␣"));
std::cout<<"⟹"<<trie_to_str(ti)<<"λn";

IntTrie<char>* tc;
typedef std::list<std::pair<int, char> > Dict;
const int  keys[] = {4, 1, 5, 9};
const char vals[] = "bacd";
Dict m;
for(int i=0; i<sizeof(keys)/sizeof(int); ++i)
  m.push_back(std::make_pair(keys[i], vals[i]));
tc = map_to_trie<char, Dict::iterator>(m.begin(), m.end());
std::copy(keys, keys+sizeof(keys)/sizeof(int),
          std::ostream_iterator<int>(std::cout, ",␣"));
std::cout<<"⟹"<<trie_to_str(tc);
```

The above code lines will output results in console like this.

```
1, 4, 5, ==>(0, (0, (0, (4))), (1, (1, (5))))
4, 1, 5, 9, ==>(0, (0, (0, (4:b))), (1:a, (1, (1, (9:d)), (5:c))))
```

Insertion of integer trie in Python

Imperative implementation of insertion in Python can be easily given by translating the pseudo code of the algorithm.

```
def trie_insert(t, key, value = None):
    if t is None:
        t = IntTrie()

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

In order to test this insertion program, some test helpers are provided:

```
def trie_to_str(t, prefix=0, depth=0):
    to_str = lambda x: "%s" %x
    str="("+to_str(prefix)
    if t.value is not None:
        str += ":"+t.value
    if t.left is not None:
        str += ",␣"+trie_to_str(t.left, prefix, depth+1)
    if t.right is not None:
        str += ",␣"+trie_to_str(t.right, (1<<depth)+prefix, depth+1)
    str+=")"
```

```
    return str

def list_to_trie(l):
    t = None
    for x in l:
        t = trie_insert(t, x)
    return t

def map_to_trie(m):
    t = None
    for k, v in m.items():
        t = trie_insert(t, k, v)
    return t
```

Function trie_to_str can print the contents of trie in pre-order, It doesn't only print the value of the node, but also print the edge information.

Function list_to_trie can repeatedly insert a list of keys into a trie, since the default parameter of value is None, so all data relative to keys are empty. If the data isn't empty, function map_to_trie can insert a list of key-value pairs into the trie.

Then a test class is given to encapsulate test cases:

```
class IntTrieTest:
    def run(self):
        self.test_insert()

    def test_insert(self):
        t = None
        t = trie_insert(t, 0);
        t = trie_insert(t, 1);
        t = trie_insert(t, 4);
        t = trie_insert(t, 5);
        print trie_to_str(t)
        t1 = list_to_trie([1, 4, 5])
        print trie_to_str(t1)
        t2 = map_to_trie({4:'b', 1:'a', 5:'c', 9:'d'})
        print trie_to_str(t2)

if __name__ == "__main__":
    IntTrieTest().run()
```

Running this program will print the following result.

```
(0, (0, (0, (4))), (1, (1, (5))))
(0, (0, (0, (4))), (1, (1, (5))))
(0, (0, (0, (4:b))), (1:a, (1, (1, (9:d)), (5:c))))
```

Please note that by pre-order traverse trie, we can get a 'lexical' order result of the keys. For instance, the last result print the little-endian format of key 1, 4, 5, 9 as below.

```
001
1
1001
101
```

They are in lexical order. We'll go back to this feature of trie later in alphabetic trie section.

Recursive insertion algorithm

Insertion can also be implemented in a recursive manner. If the LSB is 0, it means that the key to be inserted is an even number, we recursively insert the date to left child, we can divide the number by 2 and round to integer to get rid of the LSB. If the LSB is 1, the key is then an odd number, the recursive insertion will be happened to right child. This algorithm is described as below.

1: **function** INT-TRIE-INSERT'$(T, x, data)$
2: **if** $T = NIL$ **then**
3: $T \leftarrow CREATE - EMPTY - NODE()$
4: **if** $x = 0$ **then**
5: $VALUE(T) \leftarrow data$
6: **else**
7: **if** $EVEN(x)$ **then**
8: $LEFT(T) \leftarrow INT - TRIE - INSERT'(LEFT(T), x/2, data)$
9: **else**
10: $RIGHT(T) \leftarrow INT{-}TRIE{-}INSERT'(RIGHT(T), x/2, data)$
11: **return** T

Insertion of integer Trie in Haskell

To simplify the problem, If user insert a data with key already exists, we just overwrite the previous stored data. This approach can be easily replaced with other methods, such as storing data as a linked list etc.

Insertion integer key into a trie can be implemented with Haskell as below.

```
insert :: IntTrie a → Key → a → IntTrie a
insert t 0 x = Branch (left t) (Just x) (right t)
insert t k x =
  if even k
       then Branch (insert (left t) (k `div` 2) x) (value t) (right t)
       else Branch (left t) (value t) (insert (right t) (k `div` 2) x)
```

If the key is zero, we just insert the data to current node, in other cases, the program go down along the trie according to the last bit of the key.

To test this program, some test helper functions are provided.

```
fromList :: [(Key, a)] → IntTrie a
fromList xs = foldl ins Empty xs where
    ins t (k, v) = insert t k v

-- k =○ .. a2, a1, a0 ⟹ k' = ai ∗ m + k, where m=2^i
toString :: (Show a)⇒IntTrie a → String
toString t = toStr t 0 1 where
    toStr Empty k m = "."
    toStr tr k m = "(" ++ (toStr (left tr) k (2∗m)) ++
                     " " ++ (show k) ++ (valueStr (value tr)) ++
                     " " ++ (toStr (right tr) (m+k) (2∗m)) ++ ")"
    valueStr (Just x) = ":" ++ (show x)
    valueStr _ = ""
```

fromList function can create a trie from a list of integer-data pairs. toString function can turn a trie data structure to readable string for printing. This is a modified in-order tree traverse, since the number stored is in little-endian, the program store the 2^m to calculate the keys. The following code shows a test.

```
main = do
  putStrLn (toString (fromList [(1, 'a'), (4, 'b'), (5, 'c'), (9, 'd')]))
```

This will output:

```
(((. 0 (. 4:'b' .)) 0 .) 0 (((. 1 (. 9:'d' .)) 1 (. 5:'c' .)) 1:'a' .))
```

Figure 5.3 shows this result.

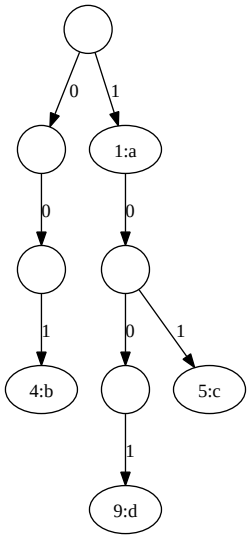


Figure 5.3: An little-endian integer binary trie for the map $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$.

Insertion of integer Trie in Scheme/Lisp

Insertion program implemented with Scheme/Lisp is quite similar, since Scheme has a complex numeric system, it will use fraction instead of round to integer, we minus the key by 1 before divide it with 2.

```
;; Insertion
;;   if user insert an existed value, just overwrite the old value
;;   usage: (insert t k x) t: trie, k: key, x: value
(define (insert t k x)
  (if (= k 0)
      (make-int-trie (left t) x (right t))
      (if (even? k)
          (make-int-trie (insert (left t) (/ k 2) x) (value t) (right t))
          (make-int-trie (left t) (value t) (insert (right t) (/ (- k 1) 2) x)))))
```

In order to make creation of Trie from a list of key-value pairs easy, here is a helper function.

```
(define (list→trie lst) ;;lst is list of pairs
  (define (insert-pair t p)
    (insert t (car p) (cadr p)))
  (fold-left insert-pair '() lst))
```

In order to convert the Trie to a readable string, a converter function is given as the following.

```
(define (trie→string trie)
  (define (value→string x)
    (cond ((null? x) ".")
          ((number? x) (number→string x))
          ((string? x) x)
          (else "unknown␣value")))
  (define (trie→str t k m)
    (if (null? t)
        "."
        (string-append "(" (trie→str (left t) k (* m 2)) "␣"
                       (number→string k) (value→string (value t)) "␣"
                       (trie→str (right t) (+ m k) (* m 2)) ")")))
  (trie→str trie 0 1))
```

To verify the program, we can test it with a easy test case.

```
(define (test-int-trie)
  (define t (list→trie (list '(1 "a") '(4 "b") '(5 "c") '(9 "d"))))
  (display (trie→string t)) (newline))
```

Evaluate this test function will generate below output.

```
(test-int-trie)
(((. 0. (. 4b ○ )) 0. ○ ) 0. (((. 1. (. 9d ○ )) 1. (. 5c ○ )) 1a ○ ))
```

Which is identical to the Haskell output.

5.3.3 Look up in integer binary trie

Iterative looking up algorithm

To look up a key in a little-endian integer binary trie. We take each bit of the key from left (LSB), and go left or right according to if the bit is 0, until we consumes all bits. this algorithm can be described as below pseudo code.

1: **function** INT-TRIE-LOOKUP(T, x)
2: **while** $x \neq 0$ and $T \neq NIL$ **do**
3: **if** $EVEN(x) = TRUE$ **then**
4: $T \leftarrow LEFT(T)$
5: **else**
6: $T \leftarrow RIGHT(T)$
7: $x \leftarrow x/2$
8: **if** $T \neq NIL$ **then**
9: **return** $DATA(T)$
10: **else**
11: **return** NIL

Look up implemented in C++

In C++, we can test the LSB by using bit-wise operation. The following code snippet searches a key in an integer Trie. If the target node is found, the value of that node is returned, else it will return the default value of the value type[1].

```
template<class T>
T lookup(IntTrie<T>* t, int key){
  while(key && t){
    if( (key & 0x1) == 0)
      t = t->left;
    else
      t = t->right;
    key>>=1;
  }
  if(t)
    return t->value;
  else
    return T();
}
```

To verify this program, some simple test cases are provided.

```
std::cout<<"\nlook up 4: "<<lookup(tc, 4)
         <<"\nlook up 9: "<<lookup(tc, 9)
         <<"\nlook up 0: "<<lookup(tc, 0);
```

Where 'tc' is the Trie we created in insertion section. the output is like below.

```
look up 4: b
look up 9: d
look up 0:
```

Look up implemented in Python

By translating the pseudo code algorithm, we can easily get a python implementation.

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

In this implementation, instead of using even-odd property, bit-wise manipulation is used to test if a bit is 0 or 1.

Here is the smoke test of the lookup function.

[1] One good alternative is to raise exception if not found

```
class IntTrieTest:
    #...
    def test_lookup(self):
        t = map_to_trie({4:'y', 1:'x', 5:'z'})
        print "look␣up␣4:␣", lookup(t, 4)
        print "look␣up␣5:␣", lookup(t, 5)
        print "look␣up␣0:␣", lookup(t, 0)
```

The output of the test cases is as below.

```
look up 4:  y
look up 5:  z
look up 0:  None
```

Recursive looking up algorithm

Looking up in integer Trie can also be implemented in recursive manner. We take the LSB of the key to be found, if it is 0, we recursively look it up in left child, else in right child.

1: **function** INT-TRIE-LOOKUP'(T, x)
2: **if** $T = NIL$ **then**
3: **return** NIL
4: **else if** $x = 0$ **then**
5: **return** $VALUE(T)$
6: **else if** $EVEN(x)$ **then**
7: **return** $INT - TRIE - LOOKUP'(LEFT(T), x/2)$
8: **else**
9: **return** $INT - TRIE - LOOKUP'(RIGHT(T), x/2)$

Look up implemented in Haskell

In Haskell, we can use pattern matching to realize the above long if-then-else statements. The program is as the following.

```
search :: IntTrie a → Key → Maybe a
search Empty k = Nothing
search t 0 = value t
search t k = if even k then search (left t) (k `div` 2)
             else search (right t) (k `div` 2)
```

If trie is empty, we simply returns nothing; if key is zero we return the value of current node; in other case we recursively search either left child or right child according to the LSB is 0 or not.

To test this program, we can write a smoke test case as following.

```
testIntTrie = "t=" ++ (toString t) ++
              "λnsearch t 4: " ++ (show $ search t 4) ++
              "λnsearch t 0: " ++ (show $ search t 0)
    where
      t = fromList [(1, 'a'), (4, 'b'), (5, 'c'), (9, 'd')]

main = do
  putStrLn testIntTrie
```

This program will output these result.

```
t=(((. 0 (. 4:'b' .)) 0 .) 0 (((. 1 (. 9:'d' .)) 1 (. 5:'c' .)) 1:'a' .))
search t 4: Just 'b'
search t 0: Nothing
```

Look up implemented in Scheme/Lisp

Scheme/Lisp implementation is quite similar. Note that we decrees key by 1 before divide it with 2.

```
(define (lookup t k)
  (if (null? t) '()
      (if (= k 0) (value t)
          (if (even? k)
              (lookup (left t) (/ k 2))
              (lookup (right t) (/ (- k 1) 2))))))
```

Test cases use the same Trie which is created in insertion section.

```
(define (test-int-trie)
  (define t (list→trie (list '(1 "a") '(4 "b") '(5 "c") '(9 "d"))))
  (display (trie→string t)) (newline)
  (display "lookup␣4:␣") (display (lookup t 4)) (newline)
  (display "lookup␣0:␣") (display (lookup t 0)) (newline))
```

the result is as same as the one output by Haskell program.

```
(test-int-trie)
(((. 0. (. 4b ○)) 0. ○) 0. (((. 1. (. 9d ○)) 1. (. 5c ○)) 1a ○))
lookup 4: b
lookup 0: ()
```

5.4 Integer Patricia Tree

It's very easy to find the drawbacks of integer binary trie. Trie wasts a lot of spaces. Note in figure 5.3, all nodes except leafs store the real data. Typically, an integer binary trie contains many nodes only have one child. It is very easy to come to the idea for improvement, to compress the chained nodes which have only one child. Patricia is such a data structure invented by Donald R. Morrison in 1968. Patricia means practical algorithm to retrieve information coded in alphanumeric[3]. Wikipedia redirect Patricia as Radix tree.

Chris Okasaki gave his implementation of Integer Patricia tree in paper [2]. If we merge the chained nodes which have only one child together in figure 5.3, We can get a patricia as shown in figure 5.4.

From this figure, we can found that the keys of sibling nodes have the longest common prefix. They only branches out at certain bit. It means that we can save a lot of data by storing the common prefix.

Different from integer Trie, using big-endian integer in Patricia doesn't cause the problem mentioned in section 5.3. Because all zero bits before MSB can be just omitted to save space. Big-endian integer is more natural than little-endian integer. Chris Okasaki list some significant advantages of big-endian Patricia trees [2].

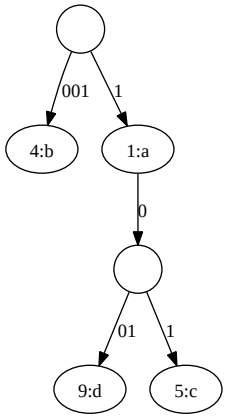


Figure 5.4: Little endian patricia for the map $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$.

5.4.1 Definition of Integer Patricia tree

Integer Patricia tree is a special kind of binary tree, it is

- either a leaf node contains an integer key and a value
- or a branch node, contains a left child and a right child. The integer keys of two children shares the longest common prefix bits, the next bit of the left child's key is zero while it is one for right child's key.

Definition of big-endian integer Patricia tree in Haskell

If we translate the above recursive definition to Haskell, we can get below Integer Patrica Tree code.

```
data IntTree a = Empty
               | Leaf Key a
               | Branch Prefix Mask (IntTree a) (IntTree a) -- prefix, mask, left, right

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

In order to tell from which bit the left and right children differ, a mask is recorded by the branch node. Typically, a mask is 2^n, all lower bits than n doesn't belong to common prefix

Definition of big-endian integer Patricia tree in Python

Such definition can be represent in Python similarly. Some helper functions are provided for easy operation later on.

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

```
        self.left = self.right = None

    def set_children(self, l, r):
        self.left = l
        self.right = r

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

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

    def get_prefix(self):
        if self.prefix is None:
            return self.key
        else:
            return self.prefix
```

Some helper member functions are provided in this definition. When Initialized, prefix, mask and children are all set to invalid value. Note the get_prefix() function, in case the prefix hasn't been initialized, which means it is a leaf node, the key itself is returned.

Definition of big-endian integer Patricia tree in C++

With ISO C++, the type of the data stored in Patricia can be abstracted as a template parameter. The definition is similar to the python version.

```
template<class T>
struct IntPatricia{
  IntPatricia(int k=0, T v=T()):
    key(k), value(v), prefix(k), mask(1), left(0), right(0){}

  ~IntPatricia(){
    delete left;
    delete right;
  }

  bool is_leaf(){
    return left == 0 && right == 0;
  }

  bool match(int x){
    return (!is_leaf()) && (maskbit(x, mask) == prefix);
  }

  void replace_child(IntPatricia<T>* x, IntPatricia<T>* y){
    if(left == x)
      left = y;
    else
      right = y;
  }
```

```
  void set_children(IntPatricia<T>* l, IntPatricia<T>* r){
    left = l;
    right = r;
  }

  int key;
  T value;
  int prefix;
  int mask;
  IntPatricia* left;
  IntPatricia* right;
};
```

In order to release the memory easily, the program just recursively deletes the children in destructor. The default value of type T is used for initialization. The prefix is initialized to be the same value as key.

For the member function match(), I'll explain it in later part.

Definition of big-endian integer Patricia tree in Scheme/Lisp

In Scheme/Lisp program, the data structure behind is still list, we provide creator functions and accessors to create Patricia and access the children, key, value, prefix and mask.

```
(define (make-leaf k v) ;; key and value
  (list k v))

(define (make-branch p m l r) ;; prefix, mask, left and right
  (list p m l r))

;; Helpers
(define (leaf? t)
  (= (length t) 2))

(define (branch? t)
  (= (length t) 4))

(define (key t)
  (if (leaf? t) (car t) '()))

(define (value t)
  (if (leaf? t) (cadr t) '()))

(define (prefix t)
  (if (branch? t) (car t) '()))

(define (mask t)
  (if (branch? t) (cadr t) '()))

(define (left t)
  (if (branch? t) (caddr t) '()))

(define (right t)
  (if (branch? t) (cadddr t) '()))
```

Function key and value are only applicable to leaf node while prefix, mask, children accessors are only applicable to branch node. So we test the node type in these functions.

5.4.2 Insertion of Integer Patricia tree

When insert a key into a integer Patricia tree, if the tree is empty, we can just create a leaf node with the given key and data. (as shown in figure 5.5).

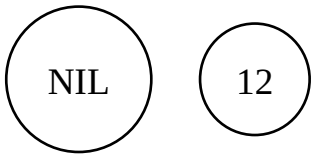


Figure 5.5: (a). Insert key 12 to an empty patricia tree.

If the tree only contains a leaf node x, we can create a branch, put the new key and data as a leaf y of the branch. To determine if the new leaf y should be left node or right node. We need find the longest common prefix of x and y, for example if key(x) is 12 (1100 in binary), key(y) is 15 (1111 in binary), then the longest common prefix is 11*oo*. The *o* denotes the bits we don't care about. we can use an integer to mask the those bits. In this case, the mask number is 4 (100 in binary). The next bit after the prefix presents 2^1. It's 0 in key(x), while it is 1 in key(y). So we put x as left child and y as right child. Figure 5.6 shows this case.

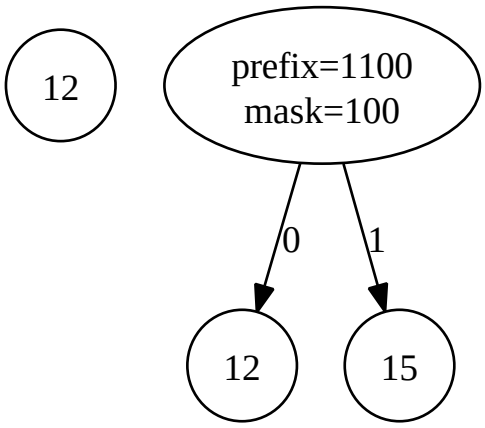


Figure 5.6: (b). Insert key 15 to the result tree in (a).

If the tree is neither empty, nor a leaf node, we need firstly check if the key to be inserted matches common prefix with root node. If it does, then we can recursively insert the key to the left child or right child according to the next bit. For instance, if we want to insert key 14 (1110 in binary) to the result tree in figure 5.6, since it has common prefix 11*oo*, and the next bit (the bit of 2^1) is

1, so we tried to insert 14 to the right child. Otherwise, if the key to be inserted doesn't match the common prefix with the root node, we need branch a new leaf node. Figure 5.7 shows these 2 different cases.

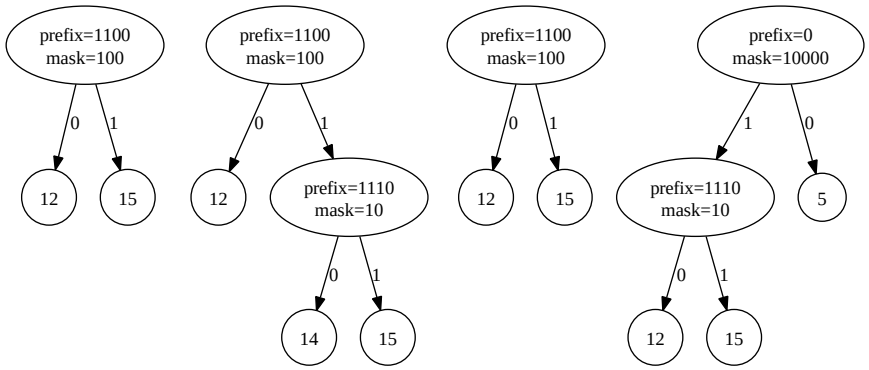


Figure 5.7: (c). Insert key 14 to the result tree in (b); (d). Insert key 5 to the result tree in (b).

Iterative insertion algorithm for integer Patricia

Summarize the above cases, the insertion of integer patricia can be described with the following algorithm.

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

In the above algorithm, MATCH procedure test if an integer key x, has the same prefix of node y above the mask bit. For instance, Suppose the prefix of node y can be denoted as $p(n), p(n-1), ..., p(i), ..., p(0)$ in binary, key x is $k(n), k(n-1), ..., k(i), ..., k(0)$, and mask of node y is $100...0 = 2^i$, if and only if $p(j) = k(j)$ for all $i \leq j \leq n$, we say the key matches.

Insertion of big-endian integer Patricia tree in Python

Based on the above algorithm, the main insertion program can be realized as the following.

```
def insert(t, key, value = None):
    if t is None:
        t = IntTree(key, value)
        return t

    node = t
    parent = None
    while(True):
        if match(key, node):
            parent = node
            if zero(key, node.mask):
                node = node.left
            else:
                node = node.right
        else:
            if node.is_leaf() and key == node.key:
                node.value = value
            else:
                new_node = branch(node, IntTree(key, value))
                if parent is None:
                    t = new_node
                else:
                    parent.replace_child(node, new_node)
            break
    return t
```

The sub procedure of match, branch, lcp etc. are given as below.

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

def match(key, tree):
    if tree.is_leaf():
        return False
    return maskbit(key, tree.mask) == tree.prefix

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

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

```
        mask<<=1
    return (maskbit(p1, mask), mask)

def branch(t1, t2):
    t = IntTree()
    (t.prefix, t.mask) = lcp(t1.get_prefix(), t2.get_prefix())
    if zero(t1.get_prefix(), t.mask):
        t.set_children(t1, t2)
    else:
        t.set_children(t2, t1)
    return t
```

Function maskbit() can clear all bits covered by a mask to 0. For instance, $x = 101101(b)$, and $mask = 2^3 = 100(b)$, the lowest 2 bits will be cleared to 0, which means $maskbit(x, mask) = 101100(b)$. This can be easily done by bit-wise operation.

Function zero() is used to check if the bit next to mask bit is 0. For instance, if $x = 101101(b), y = 101111(b)$, and $mask = 2^3 = 100(b)$, zero will check if the 2nd lowest bit is 0. So $zero(x, mask) = true, zero(y, mask) = false$.

Function lcp can extract 'the Longest Common Prefix' of two integer. For the x and y in above example, because only the last 2 bits are different, so $lcp(x, y) = 101100(b)$. And we set a mask to $2^3 = 100(b)$ to indicate that the last 2 bits are not effective for the prefix value.

To convert a list or a map into a Patricia tree, we can repeatedly insert the elements one by one. Since the program is same, except for the insert function, we can abstract list_to_xxx and map_to_xxx to utility functions

```
# in trieutil.py
def from_list(l, insert_func):
    t = None
    for x in l:
        t = insert_func(t, x)
    return t

def from_map(m, insert_func):
    t = None
    for k, v in m.items():
        t = insert_func(t, k, v)
    return t
```

With this high level functions, we can provide list_to_patricia and map_to_patricia as below.

```
def list_to_patricia(l):
    return from_list(l, insert)

def map_to_patricia(m):
    return from_map(m, insert)
```

In order to have smoke test of the above insertion program, some test cases and output helper are given.

```
def to_string(t):
    to_str = lambda x: "%s" %x
    if t is None:
```

```
        return ""
    if t.is_leaf():
        str = to_str(t.key)
        if t.value is not None:
            str += ":"+to_str(t.value)
        return str
    str ="["+to_str(t.prefix)+"@"+to_str(t.mask)+"]"
    str+="("+to_string(t.left)+","+to_string(t.right)+")"
    return str

class IntTreeTest:
    def run(self):
        self.test_insert()

    def test_insert(self):
        print "test insert"
        t = list_to_patricia([6])
        print to_string(t)
        t = list_to_patricia([6, 7])
        print to_string(t)
        t = map_to_patricia({1:'x', 4:'y', 5:'z'})
        print to_string(t)

if __name__ == "__main__":
    IntTreeTest().run()
```

The program will output a result as the following.

```
test insert
6
[6@2](6,7)
[0@8](1:x,[4@2](4:y,5:z))
```

This result means the program creates a Patrica tree shown in Figure 5.8.

Insertion of big-endian integer Patricia tree in C++

In the below C++ program, the default value of data type is used if user doesn't provide data. It is nearly strict translation of the pseudo code.

```
template<class T>
IntPatricia<T>* insert(IntPatricia<T>* t, int key, T value=T()){
  if(!t)
    return new IntPatricia<T>(key, value);

  IntPatricia<T>* node = t;
  IntPatricia<T>* parent(0);

  while( node->is_leaf()==false && node->match(key) ){
    parent = node;
    if(zero(key, node->mask))
      node = node->left;
    else
      node = node->right;
  }
```

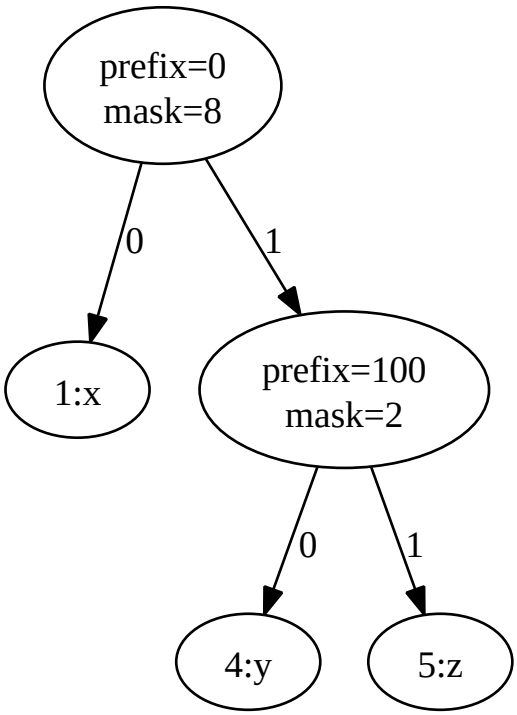


Figure 5.8: Insert map $1 \to x, 4 \to y, 5 \to z$ into a big-endian integer Patricia tree.

```
  if(node→is_leaf() && key == node→key)
    node→value = value;
  else{
    IntPatricia<T>* p = branch(node, new IntPatricia<T>(key, value));
    if(!parent)
      return p;
    parent→replace_child(node, p);
  }
  return t;
}
```

Let's review the implementation of member function match()

```
bool match(int x){
  return (!is_leaf()) && (maskbit(x, mask) == prefix);
}
```

if a node is not a leaf, and it has common prefix (in bit-wise) as the key to be inserted, we say the node match the key. It is realized by a maskbit() function as below.

```
int maskbit(int x, int mask){
  return x & (~(mask-1));
}
```

Since mask is always 2^n, minus 1 will flip it to $111...1(b)$, then we reverse the it by bit-wise not, and clear all the lowest $n-1$ bits of x by bit-wise and.

The branch() function in above program is as the following.

```
template<class T>
IntPatricia<T>* branch(IntPatricia<T>* t1, IntPatricia<T>* t2){
  IntPatricia<T>* t = new IntPatricia<T>();
  t→mask = lcp(t→prefix, t1→prefix, t2→prefix);
  if(zero(t1→prefix, t→mask))
    t→set_children(t1, t2);
  else
    t→set_children(t2, t1);
  return t;
}
```

It will extract the 'Longest Common Prefix', and create a new node, put the 2 nodes to be merged as its children. Function lcp() is implemented as below.

```
int lcp(int& p, int p1, int p2){
  int diff = p1^p2;
  int mask = 1;
  while(diff){
    diff>>=1;
    mask<<=1;
  }
  p = maskbit(p1, mask);
  return mask;
}
```

Because we can only return one value in C++, we set the reference of parameter p as the common prefix result and returns the mask value.

To decide which child is left and which one is right when branching, we need test if the bit next to mask bit is zero.

```
bool zero(int x, int mask){
  return (x & (mask>>1)) == 0;
}
```

To verify the C++ program, some simple test cases are provided.

```
IntPatricia<int>* ti(0);
const int lst[] = {6, 7};
ti = std::accumulate(lst, lst+sizeof(lst)/sizeof(int), ti,
                     std::ptr_fun(insert_key<int>));
std::copy(lst, lst+sizeof(lst)/sizeof(int),
std::ostream_iterator<int>(std::cout, ",␣"));
std::cout<<"⟹"<<patricia_to_str(ti)<<"λn";

const int keys[] = {1, 4, 5};
const char vals[] = "xyz";
IntPatricia<char>* tc(0);
for(unsigned int i=0; i<sizeof(keys)/sizeof(int); ++i)
  tc = insert(tc, keys[i], vals[i]);
std::copy(keys, keys+sizeof(keys)/sizeof(int),
   std::ostream_iterator<int>(std::cout, ",␣"));
std::cout<<"⟹"<<patricia_to_str(tc);
```

To avoid repeating ourselves, we provide a different way instead of write a list_to_patrica(), which is very similar to list_to_trie in previous section.

In C++ STL, std::accumulate() plays a similar role of fold-left. But the functor we provide to accumulate must take 2 parameters, so we provide a wrapper function as below.

```
template<class T>
IntPatricia<T>* insert_key(IntPatricia<T>* t, int key){
  return insert(t, key);
}
```

With all these code line, we can get the following result.

```
6, 7, ==>[6@2](6,7)
1, 4, 5, ==>[0@8](1:x,[4@2](4:y,5:z))
```

Recursive insertion algorithm for integer Patricia

To implement insertion in recursive way, we treat the different cases separately. If the tree is empty, we just create a leaf node and return; if the tree is a leaf node, we need check the key of the node is as same as the key to be inserted, we overwrite the data in case they are same, else we need branch a new node and extract the longest common prefix and mask bit; In other case, we need examine if the key as common prefix with the branch node, and recursively perform insertion either to left child or to right child according to the next different bit is 0 or 1; Below recursive algorithm describes this approach.

1: **function** INT-PATRICIA-INSERT'$(T, x, data)$
2: **if** $T = NIL$ or (T is a leaf and $x = KEY(T)$) **then**
3: **return** $CREATE - LEAF(x, data)$

4: **else if** $MATCH(x, PREFIX(T), MASK(T))$ **then**
5: **if** $ZERO(x, MASK(T))$ **then**
6: $LEFT(T) \leftarrow INT-PATRICIA-INSERT'(LEFT(T), x, data)$
7: **else**
8: $RIGHT(T) \leftarrow INT-PATRICIA-INSERT'(RIGHT(T), x, data)$
9: **return** T
10: **else**
11: **return** $BRANCH(T, CREATE - LEAF(x, data))$

Insertion of big-endian integer Patricia tree in Haskell

Insertion of big-endian integer Patricia tree can be implemented in Haskell by Change the above algorithm to recursive approach.

```
-- usage: insert tree key x
insert :: IntTree a → Key → a → IntTree a
insert t k x
  = case t of
      Empty → Leaf k x
      Leaf k' x' → if k==k' then Leaf k x
                   else join k (Leaf k x) k' t -- t@(Leaf k' x')
      Branch p m l r
        | match k p m → if zero k m
                        then Branch p m (insert l k x) r
                        else Branch p m l (insert r k x)
        | otherwise → join k (Leaf k x) p t -- t@(Branch p m l r)
```

The match, zero and join functions in this program are defined as below.

```
-- join 2 nodes together.
-- (prefix1, tree1) ++ (prefix2, tree2)
--  1. find the longest common prefix == lcp(prefix1, prefix2), where
--        prefix1 = a(n),a(n-1),...a(i+1),a(i),x...
--        prefix2 = a(n),a(n-1),...a(i+1),a(i),y...
--        prefix  = a(n),a(n-1),...a(i+1),a(i),00...0
--  2. mask bit = 100...0b (=2^i)
--        so mask is something like, 1,2,4,...,128,256,...
--  3. if      x=='0', y=='1' then (tree1⇒left, tree2⇒right),
--     else if x=='1', y=='0' then (tree2⇒left, tree1⇒right).
join :: Prefix → IntTree a → Prefix → IntTree a → IntTree a
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' means 'longest common prefix'
lcp :: Prefix → Prefix → (Prefix, Mask)
lcp p1 p2 = (p, m) where
    m = bit (highestBit (p1 `xor` p2))
    p = mask p1 m

-- get the order of highest bit of 1.
-- For a number x = 00...0,1,a(i-1)...a(1)
-- the result is i
```

```
highestBit :: Int → Int
highestBit x = if x==0 then 0 else 1+highestBit (shiftR x 1)

-- For a number x = a(n),a(n-1)...a(i),a(i-1),...,a(0)
-- and a mask m = 100..0 (=2^i)
-- the result of mask x m is a(n),a(n-1)...a(i),00..0
mask :: Int → Mask → Int
mask x m = (x ∘ &. complement (m-1)) -- complement means bit-wise not.

-- Test if the next bit after mask bit is zero
-- For a number x = a(n),a(n-1)...a(i),1,...a(0)
-- and a mask   m = 100..0 (=2^i)
-- because the bit next to a(i) is 1, so the result is False
-- For a number y = a(n),a(n-1)...a(i),0,...a(0) the result is True.
zero :: Int → Mask → Bool
zero x m = x ∘ &. (shiftR m 1) == 0

-- Test if a key matches a prefix above of the mask bit
-- For a prefix: p(n),p(n-1)...p(i)...p(0)
--     a key:    k(n),k(n-1)...k(i)...k(0)
-- and a mask:                 100..0 = (2^i)
-- If and only if p(j)==k(j), i≤j≤n the result is True
match :: Key → Prefix → Mask → Bool
match k p m = (mask k m) == p
```

In order to test the above insertion program, some test helper functions are provided.

```
-- Generate a Int Patricia tree from a list
-- Usage: fromList [(k1, x1), (k2, x2),..., (kn, xn)]
fromList :: [(Key, a)] → IntTree a
fromList xs = foldl ins Empty xs where
    ins t (k, v) = insert t k v

toString :: (Show a)⇒IntTree a → String
toString t =
    case t of
      Empty → "."
      Leaf k x → (show k) ++ ":" ++ (show x)
      Branch p m l r → "[" ++ (show p) ++ "@" ++ (show m) ++ "]" ++
                       "(" ++ (toString l) ++ ", " ++ (toString r) ++ ")"
```

With these helpers, insertion can be test as the following.

```
testIntTree = "t=" ++ (toString t)
    where
      t = fromList [(1, 'x'), (4, 'y'), (5, 'z')]

main = do
  putStrLn testIntTree
```

This test will output:

```
t=[0@8](1:'x', [4@2](4:'y', 5:'z'))
```

This result means the program creates a Patrica tree shown in Figure 5.8.

Insertion of big-endian integer Patricia tree in Scheme/Lisp

In Scheme/Lisp, we use switch-case like condition to test if the node is empty, or a leaf or a branch.

```
(define (insert t k x) ;; t: patrica, k: key, x: value
  (cond ((null? t) (make-leaf k x))
        ((leaf? t) (if (= (key t) k)
                       (make-leaf k x) ;; overwrite
                       (branch k (make-leaf k x) (key t) t)))
        ((branch? t) (if (match? k (prefix t) (mask t))
                         (if (zero-bit? k (mask t))
                             (make-branch (prefix t)
                                          (mask t)
                                          (insert (left t) k x)
                                          (right t))
                             (make-branch (prefix t)
                                          (mask t)
                                          (left t)
                                          (insert (right t) k x)))
                         (branch k (make-leaf k x) (prefix t) t)))))
```

Where the function match?, zero-bit?, and branch are given as the following. We use the scheme fix number bit-wise operations to mask the number and test bit.

```
(define (mask-bit x m)
  (fix:and x (fix:not (- m 1))))

(define (zero-bit? x m)
  (= (fix:and x (fix:lsh m -1)) 0))

(define (lcp x y) ;; get the longest common prefix
  (define (count-mask z)
    (if (= z 0) 1 (* 2 (count-mask (fix:lsh z -1)))))
  (let* ((m (count-mask (fix:xor x y)))
         (p (mask-bit x m)))
    (cons p m)))

(define (match? k p m)
  (= (mask-bit k m) p))

(define (branch p1 t1 p2 t2) ;; pi: prefix i, ti: Patricia i
  (let* ((pm (lcp p1 p2))
         (p (car pm))
         (m (cdr pm)))
    (if (zero-bit? p1 m)
        (make-branch p m t1 t2)
        (make-branch p m t2 t1))))
```

We can use the very same list-¿trie function which is defined in integer trie. Below is an example to create a integer Patricia tree.

```
(define (test-int-patricia)
  (define t (list→trie (list '(1 "x") '(4 "y") '(5 "z"))))
  (display t) (newline))
```

Evaluate it will generate a Patricia tree like below.

```
(test-int-patricia)
(0 8 (1 x) (4 2 (4 y) (5 z)))
```

It is identical to the insert result output by Hasekll insertion program.

5.4.3 Look up in Integer Patricia tree

Consider the property of integer Patricia tree, to look up a key, we test if the key has common prefix with the root, if yes, we then check the next bit differs from common prefix is zero or one. If it is zero, we then do look up in the left child, else we turn to right.

Iterative looking up in integer Patricia tree

In case we reach a leaf node, we can directly check if the key of the leaf is equal to what we are looking up. This algorithm can be described with the following pseudo code.

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

Look up in big-endian integer Patricia tree in Python

With Python, we can directly translate the pseudo code into valid program.

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

We can verify this program by some simple smoke test cases.

```
print "test␣look␣up"
t = map_to_patricia({1:'x', 4:'y', 5:'z'})
print "look␣up␣4:␣", lookup(t, 4)
print "look␣up␣0:␣", lookup(t, 0)
```

We can get similar output as below.

```
test look up
look up 4:  y
look up 0:  None
```

Look up in big-endian integer Patricia tree in C++

With C++ language, if the program doesn't find the key, we can either raise exception to indicate a search failure or return a special value.

```
template<class T>
T lookup(IntPatricia<T>* t, int key){
  if(!t)
    return T(); //or throw exception

  while( (!t→is_leaf()) && t→match(key)){
    if(zero(key, t→mask))
      t = t→left;
    else
      t = t→right;
  }
  if(t→is_leaf() && t→key == key)
    return t→value;
  else
    return T(); //or throw exception
}
```

We can try some test cases to search keys in a Patricia tree we created when test insertion.

```
std::cout<<"\nlook␣up␣4:␣"<<lookup(tc, 4)
         <<"\nlook␣up␣0:␣"<<lookup(tc, 0)<<"\n";
```

The output result is as the following.

```
look up 4: y
look up 0:
```

Recursive looking up in integer Patricia tree

We can easily change the while-loop in above iterative algorithm into recursive calls, so that we can have a functional approach.

1: **function** INT-PATRICIA-LOOK-UP'(T, x)
2: **if** $T = NIL$ **then**
3: **return** NIL
4: **else if** T is a leaf and $x = KEY(T)$ **then**
5: **return** $VALUE(T)$
6: **else if** $MATCH(x, PREFIX(T), MASK(T))$ **then**
7: **if** $ZERO(x, MASK(T))$ **then**
8: **return** $INT - PATRICIA - LOOK - UP'(LEFT(T), x)$
9: **else**
10: **return** $INT - PATRICIA - LOOK - UP'(RIGHT(T), x)$
11: **else**
12: **return** NIL

Look up in big-endian integer Patricia tree in Haskell

By changing the above if-then-else into pattern matching, we can get Haskell version of looking up program.

```
-- look up a key
search :: IntTree a → Key → Maybe a
search t k
  = case t of
      Empty → Nothing
      Leaf k' x → if k==k' then Just x else Nothing
      Branch p m l r
             | match k p m → if zero k m then search l k
                             else search r k
             | otherwise → Nothing
```

And we can test this program with looking up some keys in the previously created Patricia tree.

```
testIntTree = "t=" ++ (toString t) ++ "λnsearch t 4: " ++ (show $ search t 4) ++
              "λnsearch t 0: " ++ (show $ search t 0)
    where
      t = fromList [(1, 'x'), (4, 'y'), (5, 'z')]

main = do
  putStrLn testIntTree
```

The output result is as the following.

```
t=[0@8](1:'x', [4@2](4:'y', 5:'z'))
search t 4: Just 'y'
search t 0: Nothing
```

Look up in big-endian integer Patricia tree in Scheme/Lisp

Scheme/Lisp program for looking up is similar in case the tree is empty, we just returns nothing; If it is a leaf node and the key is equal to the number we are looking for, we find the result; If it is branch, we need test if binary format of the prefix matches the number, then we recursively search either in left child or in right child according to the next bit after mask is zero or not.

```
(define (lookup t k)
  (cond ((null? t) '())
        ((leaf? t) (if (= (key t) k) (value t) '()))
        ((branch? t) (if (match? k (prefix t) (mask t))
                         (if (zero-bit? k (mask t))
                             (lookup (left t) k)
                             (lookup (right t) k))
                         '()))))
```

We can test it with the Patricia tree we create in the insertion program.

```
(define (test-int-patricia)
  (define t (list→trie (list '(1 "x") '(4 "y") '(5 "z"))))
  (display t) (newline)
  (display "lookup␣4:␣") (display (lookup t 4)) (newline)
  (display "lookup␣0:␣") (display (lookup t 0)) (newline))
```

The result is like below.

```
(test-int-patricia)
(0 8 (1 x) (4 2 (4 y) (5 z)))
lookup 4: y
lookup 0: ()
```

5.5 Alphabetic Trie

Integer based Trie and Patricia Tree can be a good start point. Such technical plays important role in Compiler implementation. Okasaki pointed that the widely used Haskell Compiler GHC (Glasgow Haskell Compiler), utilizes a similar implementation for several years before 1998 [2].

While if we extend the type of the key from integer to alphabetic value, Trie and Patricia tree can be very useful in textual manipulation engineering problems.

5.5.1 Definition of alphabetic Trie

If the key is alphabetic value, just left and right children can't represent all values. For English, there are 26 letters and each can be lower case or upper case. If we don't care about case, one solution is to limit the number of branches (children) to 26. Some simplified ANSI C implementation of Trie are defined by using an array of 26 letters. This can be illustrated as in Figure 5.9.

In each node, not all branches may contain data. for instance, in the above figure, the root node only has its branches represent letter a, b, and z have sub trees. Other branches such as for letter c, is empty. For other nodes, empty branches (point to nil) are not shown.

I'll give such simplified implementation in ANSI C in later section, however, before we go to the detailed source code, let's consider some alternatives.

In case of language other than English, there may be more letters than 26, and if we need solve case sensitive problem. we face a problem of dynamic size of sub branches. There are 2 typical method to represent children, one is by using Hash table, the other is by using map. We'll show these two types of method in Python and C++.

Definition of alphabetic Trie in ANSI C

ANSI C implementation is to illustrate a simplified approach limited only to case-insensitive English language. The program can't deal with letters other than lower case 'a' to 'z' such as digits, space, tab etc.

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

In order to initialize/destroy the children and data, I also provide 2 helper functions.

```
struct Trie* create_node(){
  struct Trie* t = (struct Trie*)malloc(sizeof(struct Trie));
```

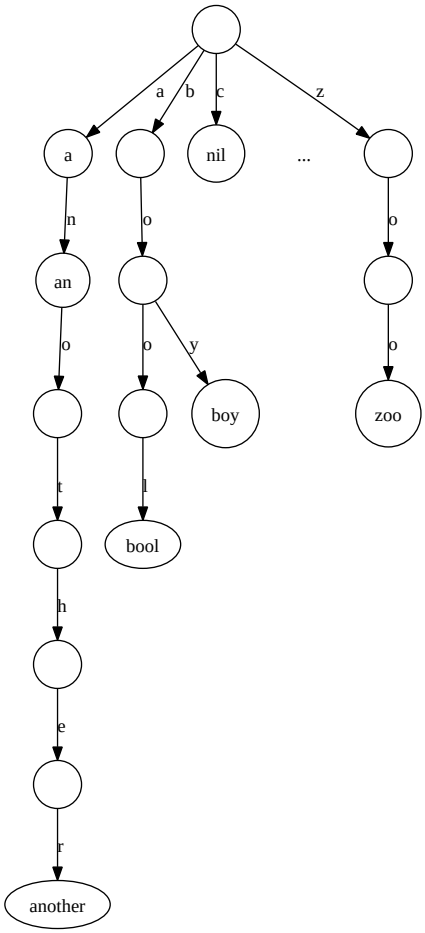


Figure 5.9: A Trie with 26 branches, with key a, an, another, bool, boy and zoo inserted.

```
  int i;
  for(i=0; i<26; ++i)
    t->children[i]=0;
  t->data=0;
  return t;
}

void destroy(struct Trie* t){
  if(!t)
    return;

  int i;
  for(i=0; i<26; ++i)
    destroy(t->children[i]);

  if(t->data)
    free(t->data);
  free(t);
}
```

Note that, the destroy function uses recursive approach to free all children nodes.

Definition of alphabetic Trie in C++

With C++ and STL, we can abstract the language and characters as type parameter. Since the number of characters of the undetermined language varies, we can use std::map to store children of a node.

```
template<class Char, class Value>
struct Trie{
  typedef Trie<Char, Value> Self;
  typedef std::map<Char, Self*> Children;
  typedef Value ValueType;

  Trie():value(Value()){}

  virtual ~Trie(){
    for(typename Children::iterator it=children.begin();
        it!=children.end(); ++it)
      delete it->second;
  }

  Value value;
  Children children;
};
```

For simple illustration purpose, recursive destructor is used to release the memory.

Definition of alphabetic Trie in Haskell

We can use Haskell record syntax to get some "free" accessor functions[4].

```
data Trie a = Trie { value :: Maybe a
```

```
                  , children :: [(Char, Trie a)]}

empty = Trie Nothing []
```

Neither Map nor Hash table is used, just a list of pairs can realize the same purpose. Function empty can help to create an empty Trie node. This implementation doesn't constrain the key values to lower case English letters, it can actually contains any values of 'Char' type.

Definition of alphabetic Trie in Python

In Python version, we can use Hash table as the data structure to represent children nodes.

```
class Trie:
    def __init__(self):
        self.value = None
        self.children = {}
```

Definition of alphabetic Trie in Scheme/Lisp

The definition of alphabetic Trie in Scheme/Lisp is a list of two elements, one is the value of the node, the other is a children list. The children list is a list of pairs, one is the character binding to the child, the other is a Trie node.

```
(define (make-trie v lop) ;; v: value, lop: children, list of char-trie pairs
  (cons v lop))

(define (value t)
  (if (null? t) '() (car t)))

(define (children t)
  (if (null? t) '() (cdr t)))
```

In order to create the child and access it easily, we also provide functions for such purpose.

```
(define (make-child k t)
  (cons k t))

(define (key child)
  (if (null? child) '() (car child)))

(define (tree child)
  (if (null? child) '() (cdr child)))
```

5.5.2 Insertion of alphabetic trie

To insert a key with type of string into a Trie, we pick the first letter from the key string. Then check from the root node, we examine which branch among the children represents this letter. If the branch is null, we then create an empty node. After that, we pick the next letter from the key string and pick the proper branch from the grand children of the root.

We repeat the above process till finishing all the letters of the key. At this time point, we can finally set the data to be inserted as the value of the node.

Note that the value of root node of Trie is always empty.

Iterative algorithm of trie insertion

The below pseudo code describes the above insertion algorithm.

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

Simplified insertion of alphabetic trie in ANSI C

Go on with the above ANSI C definition, because only lower case English letter is supported, we can use plan array manipulation to do the insertion.

```
struct Trie* insert(struct Trie* t, const char* key, void* value){
  if(!t)
    t=create_node();

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

In order to test the above program, some helper functions to print content of the Trie is provided as the following.

```
void print_trie(struct Trie* t, const char* prefix){
  printf("(%s", prefix);
  if(t->data)
    printf(":%s", (char*)(t->data));
  int i;
  for(i=0; i<26; ++i){
    if(t->children[i]){
      printf(", ");
      char* new_prefix=(char*)malloc(strlen(prefix+1)*sizeof(char));
      sprintf(new_prefix, "%s%c", prefix, i+'a');
      print_trie(t->children[i], new_prefix);
```

```
    }
  }
  printf(")");
}
```

After that, we can test the insertion program with such test cases.

```
struct Trie* test_insert(){
  struct Trie* t=0;
  t = insert(t, "a", 0);
  t = insert(t, "an", 0);
  t = insert(t, "another", 0);
  t = insert(t, "boy", 0);
  t = insert(t, "bool", 0);
  t = insert(t, "zoo", 0);
  print_trie(t, "");
  return t;
}

int main(int argc, char** argv){
  struct Trie* t = test_insert();
  destroy(t);
  return 0;
}
```

This program will output a Trie like this.

```
(, (a, (an, (ano, (anot, (anoth, (anothe, (another)))))),
(b, (bo, (boo, (bool)), (boy))), (z, (zo, (zoo))))
```

It is exactly the Trie as shown in figure 5.9.

Insertion of alphabetic Trie in C++

With above C++ definition, we can utilize STL provided search function in std::map to locate a child quickly, the program is implemented as the following, note that if user only provides key for insert, we also insert a default value of that type.

```
template<class Char, class Value, class Key>
Trie<Char, Value>* insert(Trie<Char, Value>* t, Key key, Value value=Value()){
  if(!t)
    t = new Trie<Char, Value>();

  Trie<Char, Value>* p(t);
  for(typename Key::iterator it=key.begin(); it!=key.end(); ++it){
    if(p->children.find(*it) == p->children.end())
      p->children[*it] = new Trie<Char, Value>();
    p = p->children[*it];
  }
  p->value = value;
  return t;
}

template<class T, class K>
T* insert_key(T* t, K key){
```

```
  return insert(t, key);
}
```

Where insert_key() acts as a adapter, we'll use similar accumulation method to create trie from list later.

To test this program, we provide the helper functions to print the trie on console.

```
template<class T>
std::string trie_to_str(T* t, std::string prefix=""){
  std::ostringstream s;
  s<<"("<<prefix;
  if(t->value != typename T::ValueType())
    s<<":"<<t->value;
  for(typename T::Children::iterator it=t->children.begin();
      it!=t->children.end(); ++it)
    s<<",␣"<<trie_to_str(it->second, prefix+it->first);
  s<<")";
  return s.str();
}
```

After that, we can test our program with some simple test cases.

```
typedef Trie<char, std::string> TrieType;
TrieType* t(0);
const char* lst[] = {"a", "an", "another", "b", "bob", "bool", "home"};
t = std::accumulate(lst, lst+sizeof(lst)/sizeof(char*), t,
                    std::ptr_fun(insert_key<TrieType, std::string>));
std::copy(lst, lst+sizeof(lst)/sizeof(char*),
          std::ostream_iterator<std::string>(std::cout, ",␣"));
std::cout<<"\n⟹"<<trie_to_str(t)<<"\n";
delete t;

t=0;
const char* keys[] = {"001", "100", "101"};
const char* vals[] = {"y", "x", "z"};
for(unsigned int i=0; i<sizeof(keys)/sizeof(char*); ++i)
  t = insert(t, std::string(keys[i]), std::string(vals[i]));
std::copy(keys, keys+sizeof(keys)/sizeof(char*),
          std::ostream_iterator<std::string>(std::cout, ",␣"));
std::cout<<"\n⟹"<<trie_to_str(t)<<"\n";
delete t;
```

It will output result like this.

```
a, an, another, b, bob, bool, home,
==>(, (a, (an, (ano, (anot, (anoth, (anothe, (another)))))))), (b, (bo,
(bob), (boo, (bool)))), (h, (ho, (hom, (home)))))
001, 100, 101, ==>(, (0, (00, (001:y))), (1, (10, (100:x), (101:z))))
```

Insertion of alphabetic trie in Python

In python the implementation is very similar to the pseudo code.

```
def trie_insert(t, key, value = None):
    if t is None:
```

```
        t = Trie()

    p = t
    for c in key:
        if not c in p.children:
            p.children[c] = Trie()
        p = p.children[c]
    p.value = value
    return t
```

And we define the helper functions as the following.

```
def trie_to_str(t, prefix=""):
    str="("+prefix
    if t.value is not None:
        str += ":"+t.value
    for k,v in sorted(t.children.items()):
        str += ",␣"+trie_to_str(v, prefix+k)
    str+=")"
    return str

def list_to_trie(l):
    return from_list(l, trie_insert)

def map_to_trie(m):
    return from_map(m, trie_insert)
```

With these helpers, we can test the insert program as below.

```
class TrieTest:
    #...
    def test_insert(self):
        t = None
        t = trie_insert(t, "a")
        t = trie_insert(t, "an")
        t = trie_insert(t, "another")
        t = trie_insert(t, "b")
        t = trie_insert(t, "bob")
        t = trie_insert(t, "bool")
        t = trie_insert(t, "home")
        print trie_to_str(t)
```

It will print a trie in console.

```
(, (a, (an, (ano, (anot, (anoth, (anothe, (another)))))))),
(b, (bo, (bob), (boo, (bool)))), (h, (ho, (hom, (home)))))
```

Recursive algorithm of Trie insertion

The iterative algorithms can transform to recursive algorithm by such approach. We take one character from the key, and locate the child branch, then recursively insert the left characters of the key to that branch. If the branch is empty, we create a new node and add it to children before doing the recursively insertion.

1: **function** TRIE-INSERT'($T, key, data$)
2: **if** $T = NIL$ **then**
3: $T \leftarrow EmptyNode$

```
4:    if key = NIL then
5:        VALUE(T) ← data
6:    else
7:        p ← FIND(CHILDREN(T), FIRST(key))
8:        if p = NIL then
9:            p ← APPEND(CHILDREN(T), FIRST(key), EmptyNode)
10:       TRIE − INSERT'(p, REST(key), data)
11:   return T
```

Insertion of alphabetic trie in Haskell

To realize the insertion in Haskell, The only thing we need do is to translate the for-each loop into recursive call.

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

If the key is empty, the program reaches the trivial terminator case. It just set the value. In other case, it examine the children recursively. Each element of the children is a pair, contains a character and a branch.

Some helper functions are provided as the following.

```
fromList :: [(String, a)] → Trie a
fromList xs = foldl ins empty xs where
    ins t (k, v) = insert t k v

toString :: (Show a)⇒ Trie a → String
toString t = toStr t "" where
    toStr t prefix = "(" ++ prefix ++ showMaybe (value t) ++
                     (concat (map (λ(k, v)→ ", " ++ toStr v (prefix++[k])))
                                 (sort (children t)))
                     ++ ")"
    showMaybe Nothing = ""
    showMaybe (Just x)  = ":" ++ show x

sort :: (Ord a)⇒[(a, b)] → [(a, b)]
sort [] = []
sort (p:ps) = sort xs ++ [p] ++ sort ys where
    xs = [x | x←ps, fst x ≤ fst p ]
    ys = [y | y←ps, fst y > fst p ]
```

The fromList function provide an easy way to repeatedly extract key-value pairs from a list and insert them into a Trie.

Function toString can print the Trie in a modified pre-order way. Because the children stored in a unsorted list, a sort function is provided to sort the branches. The quick-sort algorithm is used.

We can test the above program with the below test cases.

```
testTrie = "t=" ++ (toString t)
    where
      t = fromList [("a", 1), ("an", 2), ("another", 7), ("boy", 3),
                    ("bool", 4), ("zoo", 3)]

main = do
  putStrLn testTrie
```

The program outputs:

```
t=(, (a:1, (an:2, (ano, (anot, (anoth, (anothe, (another:7)))))),
(b, (bo, (boy:3), (boo, (bool:4)))), (z, (zo, (zoo:3))))
```

It is identical to the ANSI C result except the values we inserted.

Insertion of alphabetic trie in Scheme/Lisp

In order to manipulate string like list, we provide two helper function to provide car, cdr like operations for string.

```
(define (string-car s)
  (string-head s 1))

(define (string-cdr s)
  (string-tail s 1))
```

After that, we can implement insert program as the following.

```
(define (insert t k x)
  (define (ins lst k ks x) ;; return list of child
    (if (null? lst)
        (list (make-child k (insert '() ks x)))
        (if (string=? (key (car lst)) k)
            (cons (make-child k (insert (tree (car lst)) ks x)) (cdr lst))
            (cons (car lst) (ins (cdr lst) k ks x)))))
  (if (string-null? k)
      (make-trie x (children t))
      (make-trie (value t)
                 (ins (children t) (string-car k) (string-cdr k) x))))
```

In order to print readable string for a Trie, we provide a pre-order manner of Trie traverse function. It can convert a Trie to string.

```
(define (trie→string t)
  (define (value→string x)
    (cond ((null? x) ".")
          ((number? x) (number→string x))
          ((string? x) x)
          (else "unknown␣value")))
  (define (trie→str t prefix)
    (define (child→str c)
      (string-append ",␣" (trie→str (tree c) (string-append prefix (key c)))))
    (let ((lst (map child→str (sort-children (children t)))))
      (string-append "(" prefix (value→string (value t))
                     (fold-left string-append "" lst) ")")))
  (trie→str t ""))
```

Where sort-children is a quick sort algorithm to sort all children of a node based on keys.

```
(define (sort-children lst)
  (if (null? lst) '()
      (let ((xs (filter (lambda (c) (string≤? (key c) (key (car lst))))
                        (cdr lst)))
            (ys (filter (lambda (c) (string>?  (key c) (key (car lst))))
                        (cdr lst))))
        (append (sort-children xs)
                (list (car lst))
                (sort-children ys)))))
```

Function filter is only available after R^6RS, for R^5RS, we define the filter function manually.

```
(define (filter pred lst)
  (keep-matching-items lst pred))
```

With all of these definition, we can test our insert program with some simple test cases.

```
(define (test-trie)
  (define t (list→trie (list '("a" 1) '("an" 2) '("another" 7)
                             '("boy" 3) '("bool" 4) '("zoo" 3))))
  (define t2 (list→trie (list '("zoo" 3) '("bool" 4) '("boy" 3)
                              '("another" 7) '("an" 2) '("a" 1))))
  (display (trie→string t)) (newline)
  (display (trie→string t2)) (newline))
```

In the above test program, function trie-¿string is reused, it is previous defined for integer trie.

Evaluate test-trie function will output the following result.

```
(test-trie)
(., (a1, (an2, (ano., (anot., (anoth., (anothe., (another7)))))),
    (b., (bo., (boo., (bool4)), (boy3))), (z., (zo., (zoo3))))
(., (a1, (an2, (ano., (anot., (anoth., (anothe., (another7)))))),
    (b., (bo., (boo., (bool4)), (boy3))), (z., (zo., (zoo3))))
```

5.5.3 Look up in alphabetic trie

To look up a key in a Trie, we also extract the character from the key string one by one. For each character, we search among the children branches to see if there is a branch represented by this character. In case there is no such child, the look up process terminates immediately to indicate a fail result. If we reach the last character, The data stored in the current node is the result we are looking up.

Iterative look up algorithm for alphabetic Trie

This process can be described in pseudo code as below.

1: **function** TRIE-LOOK-UP(T, key)
2: **if** $T = NIL$ **then**
3: **return** NIL

```
4:    p = T
5:    for each c in key do
6:        if CHILDREN(p)[c] = NIL then
7:            return NIL
8:        p ← CHILDREN(p)[c]
9:    return DATA(p)
```

Look up in alphabetic Trie in C++

We can easily translate the iterative algorithm to C++. If the key specified can't be found in the Trie, our program returns a default value of the data type. As alternative, it is also a choice to raise exception.

```
template<class T, class Key>
typename T::ValueType lookup(T* t, Key key){
  if(!t)
    return typename T::ValueType(); //or throw exception

  T* p(t);
  for(typename Key::iterator it=key.begin(); it!=key.end(); ++it){
    if(p→children.find(*it) == p→children.end())
      return typename T::ValueType(); //or throw exception
    p = p→children[*it];
  }
  return p→value;
}
```

To verify the look up program, we can test it with some simple test cases.

```
Trie<char, int>* t(0);
const char* keys[] = {"a", "an", "another", "b", "bool", "bob", "home"};
const int vals[] = {1, 2, 7, 1, 4, 3, 4};
for(unsigned int i=0; i<sizeof(keys)/sizeof(char*); ++i)
  t = insert(t, std::string(keys[i]), vals[i]);
std::cout<<"\nlookup␣another:␣"<<lookup(t, std::string("another"))
         <<"\nlookup␣home:␣"<<lookup(t, std::string("home"))
         <<"\nlookup␣the:␣"<<lookup(t, std::string("the"))<<"\n";
delete t;
```

We can get result as below.

```
lookup another: 7
lookup home: 4
lookup the: 0
```

We see that key word "the" isn't contained in Trie, in our program, the default value of integer, 0 is returned.

Look up in alphabetic trie in Python

By translating the algorithm in Python language, we can get a imperative program.

```
def lookup(t, key):
    if t is None:
```

```
        return None

    p = t
    for c in key:
        if not c in p.children:
            return None
        p = p.children[c]
    return p.value
```

We can use the similiar test cases to test looking up function.

```
class TrieTest:
    #...
    def test_lookup(self):
        t = map_to_trie({"a":1, "an":2, "another":7, "b":1,
                         "bool":4, "bob":3, "home":4})
        print "find another: ", lookup(t, "another")
        print "find home: ", lookup(t, "home")
        print "find the: ", lookup(t, "the")
```

The result of these test cases are.

```
find another:  7
find home:  4
find the:  None
```

Recursive look up algorithm for alphabetic Trie

In recursive algorithm, we take first character from the key to be looked up. If it can be found in a child for the current node, we then recursively search the rest characters of the key from that child branch. else it means the key can't be found.

1: **function** TRIE-LOOK-UP'(T, key)
2: **if** $key = NIL$ **then**
3: **return** $VALUE(T)$
4: $p \leftarrow FIND(CHILDREN(T), FIRST(key))$
5: **if** $p = NIL$ **then**
6: **return** NIL
7: **else**
8: **return** $TRIE - LOOK - UP'(p, REST(key))$

Look up in alphabetic trie in Haskell

To express this algorithm in Haskell, we can utilize 'lookup' function in Haskell standard library[4].

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

We can append some search test cases right after insert.

```
testTrie = "t=" ++ (toString t) ++
           "\nsearch t an: " ++ (show (find t "an")) ++
           "\nsearch t boy: " ++ (show (find t "boy")) ++
           "\nsearch t the: " ++ (show (find t "the"))
...
```

Here is the search result.

```
search t an: Just 2
search t boy: Just 3
search t the: Nothing
```

Look up in alphabetic trie in Scheme/Lisp

In Scheme/Lisp program, if the key is empty, we just return the value of the current node, else we recursively find in children of the node to see if there is a child binding to a character, which match the first character of the key. We repeat this process till examine all characters of the key.

```
(define (lookup t k)
  (define (find k lst)
    (if (null? lst) '()
        (if (string=? k (key (car lst)))
            (tree (car lst))
            (find k (cdr lst)))))
  (if (string-null? k) (value t)
      (let ((child (find (string-car k) (children t))))
        (if (null? child) '()
            (lookup child (string-cdr k))))))
```

we can test this look up with similar test cases as in Haskell program.

```
(define (test-trie)
  (define t (list→trie (list '("a" 1) '("an" 2) '("another" 7)
                             '("boy" 3) '("bool" 4) '("zoo" 3))))
  (display (trie→string t)) (newline)
  (display "lookup␣an:␣") (display (lookup t "an")) (newline)
  (display "lookup␣boy:␣") (display (lookup t "boy")) (newline)
  (display "lookup␣the:␣") (display (lookup t "the")) (newline))
```

This program will output the following result.

```
(test-trie)
(., (a1, (an2, (ano., (anot., (anoth., (anothe., (another7)))))),
    (b., (bo., (boo., (bool4)), (boy3))), (z., (zo., (zoo3))))
lookup an: 2
lookup boy: 3
lookup the: ()
```

5.6 Alphabetic Patricia Tree

Alphabetic Trie has the same problem as integer Trie. It is not memory efficient. We can use the same method to compress alphabetic Trie into Patricia.

5.6.1 Definition of alphabetic Patricia Tree

Alphabetic patricia tree is a special tree, each node contains multiple branches. All children of a node share the longest common prefix string. There is no node has only one children, because it is conflict with the longest common prefix property.

If we turn the Trie shown in figure 5.9 into Patricia tree by compressing those nodes which has only one child. we can get a Patricia tree like in figure 5.10.

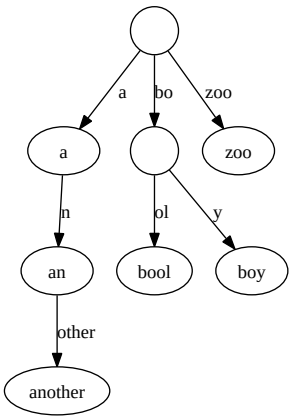


Figure 5.10: A Patricia tree, with key a, an, another, bool, boy and zoo inserted.

Note that the root node always contains empty value.

Definition of alphabetic Patricia Tree in Haskell

We can use a similar definition as Trie in Haskell, we need change the type of the first element of children from single character to string.

```
type Key = String

data Patricia a = Patricia { value :: Maybe a
                           , children :: [(Key, Patricia a)]}

empty = Patricia Nothing []

leaf :: a → Patricia a
leaf x = Patricia (Just x) []
```

Besides the definition, helper functions to create a empty Patricia node and to create a leaf node are provided.

Definition of alphabetic Patricia tree in Python

The definition of Patricia tree is same as Trie in Python.

```
class Patricia:
    def __init__(self, value = None):
        self.value = value
```

```
        self.children = {}
```

Definition of alphabetic Patricia tree in C++

With ISO C++, we abstract the key type of value type as type parameters, and utilize STL provide map container to represent children of a node.

```
template<class Key, class Value>
struct Patricia{
  typedef Patricia<Key, Value> Self;
  typedef std::map<Key, Self*> Children;
  typedef Key   KeyType;
  typedef Value ValueType;

  Patricia(Value v=Value()):value(v){}

  virtual ~Patricia(){
    for(typename Children::iterator it=children.begin();
        it!=children.end(); ++it)
      delete it->second;
  }

  Value value;
  Children children;
};
```

For illustration purpose, we simply release the memory in a recursive way.

Definition of alphabetic Patricia tree in Scheme/Lisp

We can fully reuse the definition of alphabetic Trie in Scheme/Lisp. In order to provide a easy way to create a leaf node, we define an extra helper function.

```
(define (make-leaf x)
  (make-trie x '()))
```

5.6.2 Insertion of alphabetic Patricia Tree

When insert a key, s, into the Patricia tree, if the tree is empty, we can just create an leaf node. Otherwise, we need check each child of the Patricia tree. Every branch of the children is binding to a key, we denote them as, $s_1, s_2, ..., s_n$, which means there are n branches. if s and s_i have common prefix, we then need branch out 2 new sub branches. Branch itself is represent with the common prefix, each new branches is represent with the different parts. Note there are 2 special cases. One is that s is the substring of s_i, the other is that s_i is the substring of s. Figure 5.11 shows these different cases.

Iterative insertion algorithm for alphabetic Patricia

The insertion algorithm can be described as below pseudo code.

1: **function** PATRICIA-INSERT($T, key, value$)
2: **if** $T = NIL$ **then**
3: $T \leftarrow NewNode$

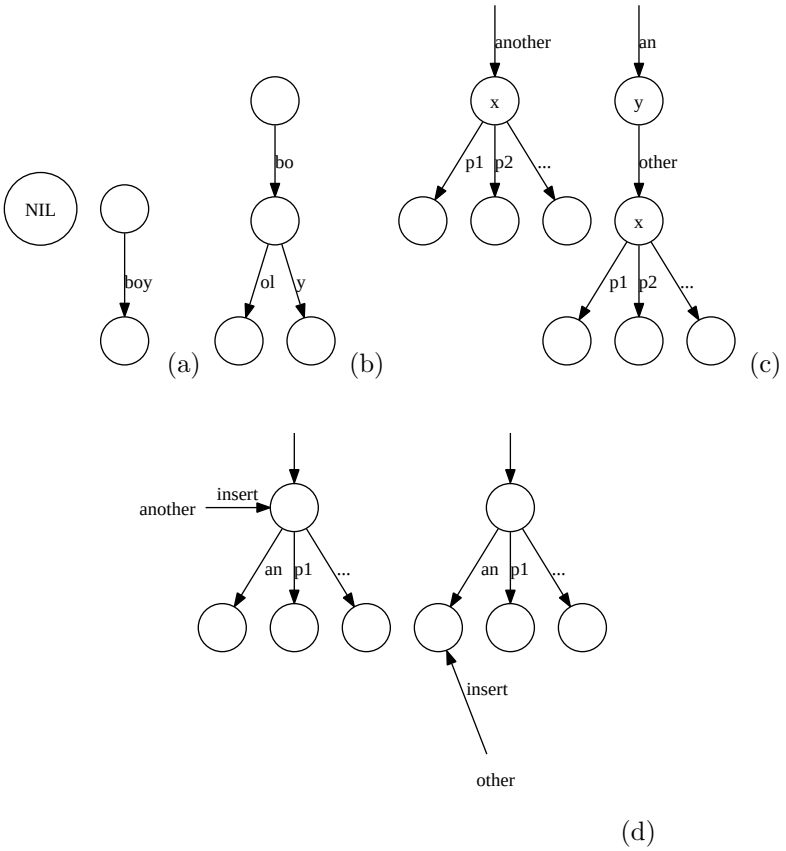


Figure 5.11: (a). Insert key, "boy" into an empty Patricia tree, the result is a leaf node;
(b). Insert key, "bool" into (a), result is a branch with common prefix "bo".
(c). Insert "an", with value y into node x with prefix "another".
(d). Insert "another", into a node with prefix "an", the key to be inserted update to "other", and do further insertion.

```
4:    p = T
5:    loop
6:        match ← FALSE
7:        for each i in CHILDREN(p) do
8:            if key = KEY(i) then
9:                VALUE(p) ← value
10:               return T
11:           prefix ← LONGEST − COMMON − PREFIX(key, KEY(i))
12:           key1 ← key subtract prefix
13:           key2 ← KEY(i) subtract prefix
14:           if prefix ≠ NIL then
15:               match ← TRUE
16:               if key2 = NIL then
17:                   p ← TREE(i)
18:                   key ← key substract prefix
19:                   break
20:               else
21:                   CHILDREN(p)[prefix] ← BRANCH(key1, value, key2, TREE(i))
22:                   DELETE CHILDREN(p)[KEY(i)]
23:                   return T
24:           if match = FALSE then
25:               CHILDREN(p)[key] ← CREATE − LEAF(value)
26:   return T
```

In the above algorithm, LONGEST-COMMON-PREFIX function will find the longest common prefix of two given string, for example, string "bool" and "boy" has longest common prefix "bo". BRANCH function will create a branch node and update keys accordingly.

Insertion of alphabetic Patricia in C++

in C++, to support implicit type conversion we utilize the KeyType and ValueType as parameter types. If we define Patricia¡std::string, std::string¿, we can directly provide char* parameters. the algorithm is implemented as the following.

```
template<class K, class V>
Patricia<K, V>* insert(Patricia<K, V>* t,
                       typename Patricia<K, V>::KeyType key,
                       typename Patricia<K, V>::ValueType value=V()){
  if(!t)
    t = new Patricia<K, V>();

  Patricia<K, V>* p = t;
  typedef typename Patricia<K, V>::Children::iterator Iterator;
  for(;;){
    bool match(false);
    for(Iterator it = p->children.begin(); it!=p->children.end(); ++it){
      K k=it->first;
      if(key == k){
        p->value = value; //overwrite
        return t;
      }
```

```
      K prefix = lcp(key, k);
      if(!prefix.empty()){
        match=true;
        if(k.empty()){ //e.g. insert "another" into "an"
          p = it→second;
          break;
        }
        else{
          p→children[prefix] = branch(key, new Patricia<K, V>(value),
                                      k, it→second);
          p→children.erase(it);
          return t;
        }
      }
    }
    if(!match){
      p→children[key] = new Patricia<K, V>(value);
      break;
    }
  }
  return t;
}
```

Where the lcp and branch functions are defined like this.

```
template<class K>
K lcp(K& s1, K& s2){
  typename K::iterator it1(s1.begin()), it2(s2.begin());
  for(; it1!=s1.end() && it2!=s2.end() && *it1 == *it2; ++it1, ++it2);
  K res(s1.begin(), it1);
  s1 = K(it1, s1.end());
  s2 = K(it2, s2.end());
  return res;
}

template<class T>
T* branch(typename T::KeyType k1, T* t1,
          typename T::KeyType k2, T* t2){
  if(k1.empty()){ //e.g. insert "an" into "another"
    t1→children[k2] = t2;
    return t1;
  }
  T* t = new T();
  t→children[k1] = t1;
  t→children[k2] = t2;
  return t;
}
```

Function lcp() will extract the longest common prefix and modify its parameters. Function branch() will create a new node and set the 2 nodes to be merged as the children. There is a special case, if the key of one node is sub-string of the other, it will chain them together.

We find the implementation of patricia_to_str() will be very same as trie_to_str(), so we can reuse it. Also the convert from a list of keys to trie can be reused

```
// list_to_trie
```

```
template<class Iterator, class T>
T* list_to_trie(Iterator first, Iterator last, T* t){
  typedef typename T::ValueType ValueType;
  return std::accumulate(first, last, t,
                         std::ptr_fun(insert_key<T, ValueType>));
}
```

We put all of the helper function templates to a utility header file, and we can test patricia insertion program as below.

```
template<class Iterator>
void test_list_to_patricia(Iterator first, Iterator last){
  typedef Patricia<std::string, std::string> PatriciaType;
  PatriciaType* t(0);
  t = list_to_trie(first, last, t);
  std::copy(first, last,
            std::ostream_iterator<std::string>(std::cout, ",␣"));
  std::cout<<"\n⟹"<<trie_to_str(t)<<"\n";
  delete t;
}

void test_insert(){
  const char* lst1[] = {"a", "an", "another", "b", "bob", "bool", "home"};
  test_list_to_patricia(lst1, lst1+sizeof(lst1)/sizeof(char*));

  const char* lst2[] = {"home", "bool", "bob", "b", "another", "an", "a"};
  test_list_to_patricia(lst2, lst2+sizeof(lst2)/sizeof(char*));

  const char* lst3[] = {"romane", "romanus", "romulus"};
  test_list_to_patricia(lst3, lst3+sizeof(lst3)/sizeof(char*));

  typedef Patricia<std::string, std::string> PatriciaType;
  PatriciaType* t(0);
  const char* keys[] = {"001", "100", "101"};
  const char* vals[] = {"y", "x", "z"};
  for(unsigned int i=0; i<sizeof(keys)/sizeof(char*); ++i)
    t = insert(t, std::string(keys[i]), std::string(vals[i]));
  std::copy(keys, keys+sizeof(keys)/sizeof(char*),
            std::ostream_iterator<std::string>(std::cout, ",␣"));
  std::cout<<"\n⟹"<<trie_to_str(t)<<"\n";
  delete t;
}
```

Running test_insert() function will generate the following output.

```
a, an, another, b, bob, bool, home,
==>(, (a, (an, (another))), (b, (bo, (bob), (bool))), (home))
home, bool, bob, b, another, an, a,
==>(, (a, (an, (another))), (b, (bo, (bob), (bool))), (home))
romane, romanus, romulus,
==>(, (rom, (roman, (romane), (romanus)), (romulus)))
001, 100, 101, ==>(, (001:y), (10, (100:x), (101:z)))
```

Insertion of alphabetic Patrica Tree in Python

By translate the insertion algorithm into Python language, we can get a program as below.

```
def insert(t, key, value = None):
    if t is None:
        t = Patricia()

    node = t
    while(True):
        match = False
        for k, tr in node.children.items():
            if key == k: # just overwrite
                node.value = value
                return t
            (prefix, k1, k2)=lcp(key, k)
            if prefix != "":
                match = True
                if k2 == "":
                    # example: insert "another" into "an", go on traversing
                    node = tr
                    key = k1
                    break
                else: #branch out a new leaf
                    node.children[prefix] = branch(k1, Patricia(value), k2, tr)
                    del node.children[k]
                    return t
        if not match: # add a new leaf
            node.children[key] = Patricia(value)
            break
    return t
```

Where the longest common prefix finding and branching functions are implemented as the following.

```
# longest common prefix
# returns (p, s1', s2'), where p is lcp, s1'=s1-p, s2'=s2-p
def lcp(s1, s2):
    j=0
    while j<len(s1) and j<len(s2) and s1[0:j]==s2[0:j]:
        j+=1
    j-=1
    return (s1[0:j], s1[j:], s2[j:])

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

Function lcp check every characters of two strings are same one by one till it met a different one or either of the string finished.

In order to test the insertion program, some helper functions are provided.

```
def to_string(t):
    return trie_to_str(t)

def list_to_patricia(l):
    return from_list(l, insert)

def map_to_patricia(m):
    return from_map(m, insert)
```

We can reuse the trie_to_str since their implementation are same. to_string function can turn a Patricia tree into string by traversing it in pre-order. list_to_patricia helps to convert a list of object into a Patricia tree by repeatedly insert every elements into the tree. While map_to_string does similar thing except it can convert a list of key-value pairs into a Patricia tree.

Then we can test the insertion program with below test cases.

```
class PatriciaTest:
    #...
    def test_insert(self):
        print "test␣insert"
        t = list_to_patricia(["a", "an", "another", "b", "bob", "bool", "home"])
        print to_string(t)
        t = list_to_patricia(["romane", "romanus", "romulus"])
        print to_string(t)
        t = map_to_patricia({"001":'y', "100":'x', "101":'z'})
        print to_string(t)
        t = list_to_patricia(["home", "bool", "bob", "b", "another", "an", "a"]);
        print to_string(t)
```

These test cases will output a series of result like this.

```
(, (a, (an, (another))), (b, (bo, (bob), (bool))), (home))
(, (rom, (roman, (romane), (romanus)), (romulus)))
(, (001:y), (10, (100:x), (101:z)))
(, (a, (an, (another))), (b, (bo, (bob), (bool))), (home))
```

Recursive insertion algorithm for alphabetic Patricia

The insertion can also be implemented recursively. When doing insertion, the program check all the children of the Patricia node, to see if there is a node can match the key. Match means they have common prefix. One special case is that the keys are same, the program just overwrite the value of that child. If there is no child can match the key, the program create a new leaf, and add it as a new child.

1: **function** PATRICIA-INSERT'($T, key, value$)
2: **if** $T = NIL$ **then**
3: $T \leftarrow EmptyNode$
4: $p \leftarrow FIND - MATCH(CHILDREN(T), key)$
5: **if** $p = NIL$ **then**
6: $ADD(CHILDREN(T), CREATE - LEAF(key, value))$

```
7:     else if KEY(p) = key then
8:         VALUE(p) ← value
9:     else
10:        q ← BRANCH(CREATE − LEAF(key, value), p)
11:        ADD(CHILDREN(T), q)
12:        DELETE(CHILDREN(T), p)
13:    return T
```

The recursion happens inside call to BRANCH. The longest common prefix of 2 nodes are extracted. If the key to be inserted is the sub-string of the node, we just chain them together; If the prefix of the node is the sub-string of the key, we recursively insert the rest of the key to the node. In other case, we create a new node with the common prefix and set its two children.

```
1: function BRANCH(T1, T2)
2:     prefix ← LONGEST − COMMON − PREFIX(T1, T2)
3:     p ← EmptyNode
4:     if prefix = KEY(T1) then
5:         KEY(T2) ← KEY(T2) subtract prefix
6:         p ← CREATE − LEAF(prefix, VALUE(T1))
7:         ADD(CHILDREN(p), T2)
8:     else if prefix = KEY(T2) then
9:         KEY(T1) ← KEY(T1) subtract prefix
10:        p ← PATRICIA − INSERT′(T2, KEY(T1), VALUE(T1))
11:        KEY(p) ← prefix
12:    else
13:        KEY(T2) ← KEY(T2) subtract prefix
14:        KEY(T1) ← KEY(T1) subtract prefix
15:        ADD(CHILDREN(p), T1, T2)
16:        KEY(p) ← prefix
17:    return p
```

Insertion of alphabetic Patrica Tree in Haskell

By implementing the above algorithm in Recursive way, we can get a Haskell program of Patricia insertion.

```
insert :: Patricia a → Key → a → Patricia a
insert t k x = Patricia (value t) (ins (children t) k x) where
    ins []     k x = [(k, Patricia (Just x) [])]
    ins (p:ps) k x
        | (fst p) == k
            = (k, Patricia (Just x) (children (snd p))):ps --overwrite
        | match (fst p) k
            = (branch k x (fst p) (snd p)):ps
        | otherwise
            = p:(ins ps k x)
```

Function insert takes a Patricia tree, a key and a value. It will call an internal function ins to insert the data into the children of the tree. If the tree has no children, it simply create a leaf node and put it as the single child of the tree. In other case it will check each child to see if any one has common prefix with

the key. There is a special case, the child has very same key, we can overwrite the data. If the child has common prefix is located, we branch out a new node.

A function match is provided to determine if two keys have common prefix as below.

```
match :: Key → Key → Bool
match [] _ = False
match _ [] = False
match x y = head x == head y
```

This function is straightforward, only if the first character of the two keys are identical, we say they have common prefix.

Branch out function and the longest common prefix function are implemented like the following.

```
branch :: Key → a → Key → Patricia a → (Key, Patricia a)
branch k1 x k2 t2
    | k1 == k
        -- ex: insert "an" into "another"
        = (k, Patricia (Just x) [(k2', t2)])
    | k2 == k
        -- ex: insert "another" into "an"
        = (k, insert t2 k1' x)
    | otherwise = (k, Patricia Nothing [(k1', leaf x), (k2', t2)])
   where
      k = lcp k1 k2
      k1' = drop (length k) k1
      k2' = drop (length k) k2

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

Function take a key, k1, a value, another key, k2, and a Patricia tree t2. It will first call lcp to get the longest common prefix, k, and the different part of the original key. If k is just k1, which means k1 is a sub-string of k2, we create a new Patricia node, put the new value in it, then set the left part of k2 and t2 as the single child of this new node. If k is just k2, which means k2 is a sub-string of k1, we recursively insert the update key and value to t2. Otherwise, we create a new node, along with the the longest common prefix. the new node has 2 children, one is t2, the other is a leaf node of the data to be inserted. Each of them are binding to updated keys.

In order to test the above program, we provided some helper functions.

```
fromList :: [(Key, a)] → Patricia a
fromList xs = foldl ins empty xs where
    ins t (k, v) = insert t k v

sort :: (Ord a)⇒[(a, b)] → [(a, b)]
sort [] = []
sort (p:ps) = sort xs ++ [p] ++ sort ys where
    xs = [x | x←ps, fst x ≤ fst p ]
    ys = [y | y←ps, fst y > fst p ]
```

```
toString :: (Show a)⇒Patricia a → String
toString t = toStr t "" where
    toStr t prefix = "(" ++ prefix ++ showMaybe (value t) ++
                     (concat $ map (λ(k, v)→", " ++ toStr v (prefix++k))
                             (sort $children t))
                     ++ ")"
    showMaybe Nothing = ""
    showMaybe (Just x) = ":" ++ show x
```

Function fromList can recursively insert each key-value pair into a Patricia node. Function sort helps to sort a list of key-value pairs based on keys by using quick sort algorithm. toString turns a Patricia tree into a string by using modified pre-order.

After that, we can test our insert program with the following test cases.

```
testPatricia = "t1=" ++ (toString t1) ++ "λn" ++
               "t2=" ++ (toString t2)
    where
      t1 = fromList [("a", 1), ("an", 2), ("another", 7),
                     ("boy", 3), ("bool", 4), ("zoo", 3)]
      t2 = fromList [("zoo", 3), ("bool", 4), ("boy", 3),
                     ("another", 7), ("an", 2), ("a", 1)]

main = do
  putStrLn testPatricia
```

No matter what's the insert order, the 2 test cases output an identical result.

```
t1=(, (a:1, (an:2, (another:7))), (bo, (bool:4), (boy:3)), (zoo:3))
t2=(, (a:1, (an:2, (another:7))), (bo, (bool:4), (boy:3)), (zoo:3))
```

Insertion of alphabetic Patrica Tree in Scheme/Lisp

In Scheme/Lisp, If the root doesn't has child, we create a leaf node with the value, and bind the key with this node; if the key binding to one child is equal to the string we want to insert, we just overwrite the current value; If the key has common prefix of the string to be inserted, we branch out a new node.

```
(define (insert t k x)
  (define (ins lst k x) ;; lst: [(key patrica)]
    (if (null? lst) (list (make-child k (make-leaf x)))
        (cond ((string=? (key (car lst)) k)
               (cons (make-child k (make-trie x (children (tree (car lst)))))
                     (cdr lst)))
              ((match? (key (car lst)) k)
               (cons (branch k x (key (car lst)) (tree (car lst)))
                     (cdr lst)))
              (else (cons (car lst) (ins (cdr lst) k x))))))
  (make-trie (value t) (ins (children t) k x)))
```

The match? function just test if two strings have common prefix.

```
(define (match? x y)
  (and (not (or (string-null? x) (string-null? y)))
       (string=? (string-car x) (string-car y))))
```

Function branch takes 4 parameters, the first key, the value to be inserted, the second key, and the Patricia tree need to be branch out. If will first find the longest common prefix of the two keys, If it is equal to the first key, it means that the first key is the prefix of the second key, we just create a new node with the value and chained the Patricia tree by setting it as the only one child of this new node; If the longest common prefix is equal to the second key, it means that the second key is the prefix of the first key, we just recursively insert the different part (the remove the prefix part) into this Patricia tree; In other case, we just create a branch node and set its two children, one is the leaf node with the value to be inserted, the other is the Patricia tree passed as the fourth parameter.

```
(define (branch k1 x k2 t2) ;; returns (key tree)
  (let* ((k (lcp k1 k2))
         (k1-new (string-tail k1 (string-length k)))
         (k2-new (string-tail k2 (string-length k))))
    (cond ((string=? k1 k) ;; e.g. insert "an" into "another"
           (make-child k (make-trie x (list (make-child k2-new t2)))))
          ((string=? k2 k) ;; e.g. insert "another" into "an"
           (make-child k (insert t2 k1-new x)))
          (else (make-child k (make-trie
                               '()
                               (list (make-child k1-new (make-leaf x))
                                     (make-child k2-new t2))))))))
```

Where the longest common prefix is extracted as the following.

```
(define (lcp x y)
  (let ((len (string-match-forward x y)))
    (string-head x len)))
```

We can reuse the list-¿trie and trie-¿string functions to test our program.

```
(define (test-patricia)
  (define t (list→trie (list '("a" 1) '("an" 2) '("another" 7)
                        '("boy" 3) '("bool" 4) '("zoo" 3))))
  (define t2 (list→trie (list '("zoo" 3) '("bool" 4) '("boy" 3)
                         '("another" 7) '("an" 2) '("a" 1))))
  (display (trie→string t)) (newline)
  (display (trie→string t2)) (newline))
```

Evaluate this function will print t and t2 as below.

```
(test-patricia)
(., (a1, (an2, (another7))), (bo., (bool4), (boy3)), (zoo3))
(., (a1, (an2, (another7))), (bo., (bool4), (boy3)), (zoo3))
```

5.6.3 Look up in alphabetic Patricia tree

Different with Trie, we can't take each character from the key to look up. We need check each child to see if it is a prefix of the key to be found. If there is such a child, we then remove the prefix from the key, and search this updated key in that child. if we can't find any children as a prefix of the key, it means the looking up failed.

Iterative look up algorithm for alphabetic Patricia tree

This algorithm can be described in pseudo code as below.

```
1: function PATRICIA-LOOK-UP(T, key)
2:     if T = NIL then
3:         return NIL
4:     repeat
5:         match ← FALSE
6:         for each i in CHILDREN(T) do
7:             if key = KEY(i) then
8:                 return VALUE(TREE(i))
9:             if KEY(i) IS-PREFIX-OF key then
10:                match ← TRUE
11:                key ← key subtract KEY(i)
12:                T ← TREE(i)
13:                break
14:    until match = FALSE
15:    return NIL
```

Look up in alphabetic Patrica Tree in C++

In C++, we abstract the key type as a template parameter. By refer to the KeyType defined in Patricia, we can get the support of implicitly type conversion. If we can't find the key in a Patricia tree, the below program returns default value of data type. One alternative is to throw exception.

```
template<class K, class V>
V lookup(Patricia<K, V>* t, typename Patricia<K, V>::KeyType key){
  typedef typename Patricia<K, V>::Children::iterator Iterator;
  if(!t)
    return V(); //or throw exception
  for(;;){
    bool match(false);
    for(Iterator it=t->children.begin(); it!=t->children.end(); ++it){
      K k = it->first;
      if(key == k)
        return it->second->value;
      K prefix = lcp(key, k);
      if((!prefix.empty()) && k.empty()){
        match = true;
        t = it->second;
        break;
      }
    }
    if(!match)
      return V(); //or throw exception
  }
}
```

To verify the look up program, we test it with the following simple test cases.

```
Patricia<std::string, int>* t(0);
const char* keys[] = {"a", "an", "another", "boy", "bool", "home"};
const int vals[] = {1, 2, 7, 3, 4, 4};
```

```
for(unsigned int i=0; i<sizeof(keys)/sizeof(char*); ++i)
  t = insert(t, keys[i], vals[i]);
std::cout<<"\nlookup another: "<<lookup(t, "another")
         <<"\nlookup boo: "<<lookup(t, "boo")
         <<"\nlookup boy: "<<lookup(t, "boy")
         <<"\nlookup by: "<<lookup(t, "by")
         <<"\nlookup boolean: "<<lookup(t, "boolean")<<"\n";
delete t;
```

This program will output the the result like below.

```
lookup another: 7
lookup boo: 0
lookup boy: 3
lookup by: 0
lookup boolean: 0
```

Look up in alphabetic Patricia Tree in Python

The implementation of looking up in Python is similar to the pseudo code. Because Python don't support repeat-until loop directly, a while loop is used instead.

```
def lookup(t, key):
    if t is None:
        return None
    while(True):
        match = False
        for k, tr in t.children.items():
            if k == key:
                return tr.value
            (prefix, k1, k2) = lcp(key, k)
            if prefix != "" and k2 == "":
                match = True
                key = k1
                t = tr
                break
        if not match:
            return None
```

We can verify the looking up program as below.

```
class PatriciaTest:
    #○..
    def test_lookup(self):
        t = map_to_patricia({"a":1, "an":2, "another":7, "b":1, "bob":3, λ
                             "bool":4, "home":4})
        print "search t another", lookup(t, "another")
        print "search t boo", lookup(t, "boo")
        print "search t bob", lookup(t, "bob")
        print "search t boolean", lookup(t, "boolean")
```

The test result output in console is like the following.

```
search t another 7
search t boo None
```

```
search t bob 3
search t boolean None
```

Recursive look up algorithm for alphabetic Patricia tree

To implement the look up recursively, we just look up among the children of the Patricia tree.

1: **function** PATRICIA-LOOK-UP'(T, key)
2: **if** $T = NIL$ **then**
3: **return** NIL
4: **else**
5: **return** $FIND - IN - CHILDREN(CHILDREN(T), key)$

The real recursion happens in FIND-IN-CHILDREN call, we pass the children list as an argument. If it is not empty, we take first child, and check if the prefix of this child is equal to the key; the value of this child will be returned if they are same; if the prefix of the child is just a prefix of the key, we recursively find in this child with updated key.

1: **function** FIND-IN-CHILDREN(l, key)
2: **if** $l = NIL$ **then**
3: **return** NIL
4: **else if** $KEY(FIRST(l)) = key$ **then**
5: **return** $VALUE(FIRST(l))$
6: **else if** $KEY(FIRST(l))$ is prefix of key **then**
7: $key \leftarrow key$ subtract $KEY(FIRST(l))$
8: **return** $PATRICIA - LOOK - UP'(FIRST(l), key)$
9: **else**
10: **return** $FIND - IN - CHILDREN(REST(l), key)$

Look up in alphabetic Patrica Tree in Haskell

In Haskell implementation, the above algorithm should be turned into recursive way.

```
-- lookup
import qualified Data.List

find :: Patricia a → Key → Maybe a
find t k = find' (children t) k where
    find' [] _ = Nothing
    find' (p:ps) k
          | (fst p) == k = value (snd p)
          | (fst p) `Data.List.isPrefixOf` k = find (snd p) (diff (fst p) k)
          | otherwise = find' ps k
    diff k1 k2 = drop (length (lcp k1 k2)) k2
```

When we search a given key in a Patricia tree, we recursively check each of the child. If there are no children at all, we stop the recursion and indicate a look up failure. In other case, we pick the prefix-node pair one by one. If the prefix is as same as the given key, it means the target node is found and the value of the node is returned. If the key has common prefix with the child, the key will be updated by removing the longest common prefix and we performs looking up recursively.

We can verify the above Haskell program with the following simple cases.

```
testPatricia = "t1=" ++ (toString t1) ++ "\n" ++
               "find t1 another =" ++ (show (find t1 "another")) ++ "\n" ++
               "find t1 bo = " ++ (show (find t1 "bo")) ++ "\n" ++
               "find t1 boy = " ++ (show (find t1 "boy")) ++ "\n" ++
               "find t1 boolean = " ++ (show (find t1 "boolean"))
    where
      t1 = fromList [("a", 1), ("an", 2), ("another", 7), ("boy", 3),
           ("bool", 4), ("zoo", 3)]

main = do
  putStrLn testPatricia
```

The output is as below.

```
t1=(, (a:1, (an:2, (another:7))), (bo, (bool:4), (boy:3)), (zoo:3))
find t1 another =Just 7
find t1 bo = Nothing
find t1 boy = Just 3
find t1 boolean = Nothing
```

Look up in alphabetic Patricia Tree in Scheme/Lisp

The Scheme/Lisp program is given as the following. The function delegate the looking up to an inner function which will check each child to see if the key binding to the child match the string we are looking for.

```
(define (lookup t k)
  (define (find lst k) ;; lst, [(k patricia)]
    (if (null? lst) '()
        (cond ((string=? (key (car lst)) k) (value (tree (car lst))))
              ((string-prefix? (key (car lst)) k)
               (lookup (tree (car lst))
                       (string-tail k (string-length (key (car lst))))))
              (else (find (cdr lst) k)))))
  (find (children t) k))
```

In order to verify this program, some simple test cases are given to search in the Patricia we created in previous section.

```
(define (test-patricia)
  (define t (list→trie (list '("a" 1) '("an" 2) '("another" 7)
                        '("boy" 3) '("bool" 4) '("zoo" 3))))
  (display (trie→string t)) (newline)
  (display "lookup␣another:␣") (display (lookup t "another")) (newline)
  (display "lookup␣bo:␣") (display (lookup t "bo")) (newline)
  (display "lookup␣boy:␣") (display (lookup t "boy")) (newline)
  (display "lookup␣by:␣") (display (lookup t "by")) (newline)
  (display "lookup␣boolean:␣") (display (lookup t "boolean")) (newline))
```

This program will output the same result as the Haskell one.

```
(test-patricia)
(., (a1, (an2, (another7))), (bo., (bool4), (boy3)), (zoo3))
lookup another: 7
lookup bo: ()
```

```
lookup boy: 3
lookup by: ()
lookup boolean: ()
```

5.7 Trie and Patricia used in Industry

Trie and Patricia are widely used in software industry. Integer based Patricia tree is widely used in compiler. Some daily used software has very interesting features can be realized with Trie and Patricia. In the following sections, I'll list some of them, including, e-dictionary, word auto-completion, t9 input method etc. The commercial implementation typically doesn't adopt Trie or Patricia directly. However, Trie and Patricia can be shown as a kind of example realization.

5.7.1 e-dictionary and word auto-completion

Figure 5.12 shows a screen shot of an English-Chinese dictionary. In order to provide good user experience, when user input something, the dictionary will search its word library, and list all candidate words and phrases similar to what user have entered.

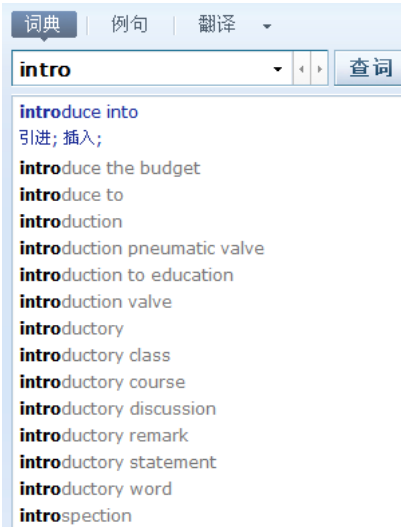


Figure 5.12: e-dictionary. All candidates starting with what user input are listed.

Typically such dictionary contains hundreds of thousands words, performs a whole word search is expensive. Commercial software adopts complex approach, including caching, indexing etc to speed up this process.

Similar with e-dictionary, figure 5.13 shows a popular Internet search engine, when user input something, it will provide a candidate lists, with all items start with what user has entered. And these candidates are shown in an order of

popularity. The more people search for a word, the upper position it will be shown in the list.

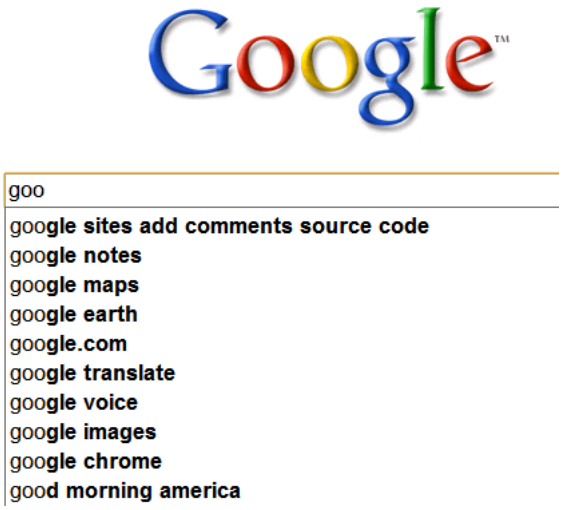


Figure 5.13: Search engine. All candidates key words starting with what user input are listed.

In both case, we say the software provide a kind of word auto-completion support. In some modern IDE, the editor can even helps user to auto-complete programmings.

In this section, I'll show a very simple implementation of e-dictionary with Trie and Patricia. To simplify the problem, let us assume the dictionary only support English - English information.

Typically, a dictionary contains a lot of key-value pairs, the keys are English words or phrases, and the relative values are the meaning of the words.

We can store all words and their meanings to a Trie, the drawback for this approach is that it isn't space effective. We'll use Patricia as alternative later on.

As an example, when user want to look up 'a', the dictionary does not only return the meaning of then English word 'a', but also provide a list of candidate words, which are all start with 'a', including 'abandon', 'about', 'accent', 'adam', ... Of course all these words are stored in Trie.

If there are too many candidates, one solution is only display the top 10 words for the user, and if he like, he can browse more.

Below pseudo code reuse the looking up program in previous sections and Expand all potential top N candidates.

1: **function** TRIE-LOOK-UP-TOP-N(T, key, N)
2: $p \leftarrow TRIE - LOOK - UP'(T, key)$
3: **return** $EXPAND - TOP - N(p, key, N)$

Note that we should modify the TRIE-LOOK-UP a bit, instead of return the value of the node, TRIE-LOOK-UP' returns the node itself.

Another alternative is to use Patricia instead of Trie. It can save much

spaces.

Iterative algorithm of search top N candidate in Patricia

The algorithm is similar to the Patricia look up one, but when we found a node which key start from the string we are looking for, we expand all its children until we get N candidates.

```
1: function PATRICIA-LOOK-UP-TOP-N(T, key, N)
2:     if T = NIL then
3:         return NIL
4:     prefix ← NIL
5:     repeat
6:         match ← FALSE
7:         for each i in CHILDREN(T) do
8:             if key is prefix of KEY(i) then
9:                 return EXPAND − TOP − N(TREE(i), prefix, N)
10:            if KEY(i) is prefix of key then
11:                match ← TRUE
12:                key ← key subtract KEY(i)
13:                T ← TREE(i)
14:                prefix ← prefix + KEY(i)
15:                break
16:    until match = FALSE
17:    return NIL
```

An e-dictionary in Python

In Python implementation, a function trie_lookup is provided to perform search all top N candidate started with a given string.

```
def trie_lookup(t, key, n):
    if t is None:
        return None

    p = t
    for c in key:
        if not c in p.children:
            return None
        p=p.children[c]
    return expand(key, p, n)

def expand(prefix, t, n):
    res = []
    q = [(prefix, t)]
    while len(res)<n and len(q)>0:
        (s, p) = q.pop(0)
        if p.value is not None:
            res.append((s, p.value))
        for k, tr in p.children.items():
            q.append((s+k, tr))
    return res
```

Compare with the Trie look up function, the first part of this program is almost same. The difference part is after we successfully located the node which matches the key, all sub trees are expanded from this node in a bread-first search manner, and the top n candidates are returned.

This program can be verified by below simple test cases.

```
class LookupTest:
    def __init__(self):
        dict = {"a":"the␣first␣letter␣of␣English", λ
           "an":"...same␣dict␣as␣in␣Haskell␣example"}
        self.tt = trie.map_to_trie(dict)

    def run(self):
        self.test_trie_lookup()

    def test_trie_lookup(self):
        print "test␣lookup␣top␣5"
        print "search␣a␣", trie_lookup(self.tt, "a", 5)
        print "search␣ab␣", trie_lookup(self.tt, "ab", 5)
```

The test will output the following result.

```
test lookup to 5
search a  [('a', 'the first letter of English'), ('an', "used instead of 'a'
when the following word begins with a vowel sound"), ('adam', 'a character in
the Bible who was the first man made by God'), ('about', 'on the subject of;
connected with'), ('abandon', 'to leave a place, thing or person forever')]
search ab  [('about', 'on the subject of; connected with'), ('abandon', 'to
leave a place, thing or person forever')]
```

To save the spaces, we can also implement such a dictionary search by using Patricia.

```
def patricia_lookup(t, key, n):
    if t is None:
        return None
    prefix = ""
    while(True):
        match = False
        for k, tr in t.children.items():
            if string.find(k, key) == 0: #is prefix of
                return expand(prefix+k, tr, n)
            if string.find(key, k) ==0:
                match = True
                key = key[len(k):]
                t = tr
                prefix += k
                break
        if not match:
            return None
```

In this program, we called Python string class to test if a string x is prefix of a string y. In case we locate a node with the key we are looking up is either equal of as prefix of the this sub tree, we expand it till we find n candidates. Function expand() can be reused here.

We can test this program with the very same test cases and the results are identical to the previous one.

An e-dictionary in C++

In C++ implementation, we overload the look up function by providing an extra integer n to indicate we want to search top n candidates. the result is a list of key-value pairs,

```
//lookup top n candidate with prefix key in Trie
template<class K, class V>
std::list<std::pair<K, V> > lookup(Trie<K, V>* t,
                                   typename Trie<K, V>::KeyType key,
                                   unsigned int n)
{
  typedef std::list<std::pair<K, V> > Result;
  if(!t)
    return Result();

  Trie<K, V>* p(t);
  for(typename K::iterator it=key.begin(); it!=key.end(); ++it){
    if(p->children.find(*it) == p->children.end())
      return Result();
    p = p->children[*it];
  }
  return expand(key, p, n);
}
```

The program is almost same as the Trie looking up one, except it will call expand function when it located the node with the key. Function expand is as the following.

```
template<class T>
std::list<std::pair<typename T::KeyType, typename T::ValueType> >
expand(typename T::KeyType prefix, T* t, unsigned int n)
{
  typedef typename T::KeyType KeyType;
  typedef typename T::ValueType ValueType;
  typedef std::list<std::pair<KeyType, ValueType> > Result;

  Result res;
  std::queue<std::pair<KeyType, T*> > q;
  q.push(std::make_pair(prefix, t));
  while(res.size()<n && (!q.empty())){
    std::pair<KeyType, T*> i = q.front();
    KeyType s = i.first;
    T* p = i.second;
    q.pop();
    if(p->value != ValueType()){
      res.push_back(std::make_pair(s, p->value));
    }
    for(typename T::Children::iterator it = p->children.begin();
        it!=p->children.end(); ++it)
      q.push(std::make_pair(s+it->first, it->second));
  }
```

```
  return res;
}
```

This function use a bread-first search approach to expand top N candidates, it maintain a queue to store the node it is currently dealing with. Each time the program picks a candidate node from the queue, expands all its children and put them to the queue. the program will terminate when the queue is empty or we have already found N candidates.

Function expand is generic we'll use it in later sections.

Then we can provide a helper function to convert the candidate list to readable string. Note that this list is actually a list of pairs so we can provide a generic function.

```
//list of pairs to string
template<class Container>
std::string lop_to_str(Container coll){
  typedef typename Container::iterator Iterator;
  std::ostringstream s;
  s<<"[";
  for(Iterator it=coll.begin(); it!=coll.end(); ++it)
    s<<"("<<it→first<<",␣"<<it→second<<"),␣";
  s<<"]";
  return s.str();
}
```

After that, we can test the program with some simple test cases.

```
Trie<std::string, std::string>* t(0);
const char* dict[] = {
  "a", "the␣first␣letter␣of␣English", λ
  "an", "used␣instead␣of␣'a'␣when␣the␣following␣word␣begins␣with␣a␣vowel␣sound", λ
  "another", "one␣more␣person␣or␣thing␣or␣an␣extra␣amount", λ
  "abandon", "to␣leave␣a␣place,␣thing␣or␣person␣forever", λ
  "about", "on␣the␣subject␣of;␣connected␣with", λ
  "adam", "a␣character␣in␣the␣Bible␣who␣was␣the␣first␣man␣made␣by␣God", λ
  "boy", "a␣male␣child␣or,␣more␣generally,␣a␣male␣of␣any␣age", λ
  "body", "the␣whole␣physical␣structure␣that␣forms␣a␣person␣or␣animal", λ
  "zoo", "an␣area␣in␣which␣animals,␣especially␣wild␣animals,␣are␣kept" λ
         "so␣that␣people␣can␣go␣and␣look␣at␣them,␣or␣study␣them"};

const char** first=dict;
const char** last =dict + sizeof(dict)/sizeof(char*);
for(;first!=last; ++first, ++first)
  t = insert(t, *first, *(first+1));
}

std::cout<<"test␣lookup␣top␣5␣in␣Trieλn"
         <<"search␣a␣"<<lop_to_str(lookup(t, "a", 5))<<"λn"
         <<"search␣ab␣"<<lop_to_str(lookup(t, "ab", 5))<<"λn";
delete t;
```

The result print to the console is something like this:

```
test lookup top 5 in Trie
search a [(a, the first letter of English), (an, used instead of 'a'
when the following word begins with a vowel sound), (adam, a character
```

```
in the Bible who was the first man made by God), (about, on the
subject of; connected with), (abandon, to leave a place, thing or
person forever), ]
search ab [(about, on the subject of; connected with), (abandon, to
leave a place, thing or person forever), ]
```

To save the the space with Patricia, we provide a C++ program to search top N candidate as below.

```
template<class K, class V>
std::list<std::pair<K, V> > lookup(Patricia<K, V>* t,
                                   typename Patricia<K, V>::KeyType key,
                                   unsigned int n)
{
  typedef typename std::list<std::pair<K, V> > Result;
  typedef typename Patricia<K, V>::Children::iterator Iterator;
  if(!t)
    return Result();
  K prefix;
  for(;;){
    bool match(false);
    for(Iterator it=t->children.begin(); it!=t->children.end(); ++it){
      K k(it->first);
      if(is_prefix_of(key, k))
        return expand(prefix+k, it->second, n);
      if(is_prefix_of(k, key)){
        match = true;
        prefix += k;
        lcp<K>(key, k); //update key
        t = it->second;
        break;
      }
    }
    if(!match)
      return Result();
  }
}
```

The program iterate all children if the string we are looked up is prefix of one child, we expand this child to find top N candidates; If the in the opposite case, we update the string and go on examine into this child Patricia tree.

Where the function is_prefix_of() is defined as below.

```
// x `is prefix of` y?
template<class T>
bool is_prefix_of(T x, T y){
  if(x.size()≤y.size())
    return std::equal(x.begin(), x.end(), y.begin());
  return false;
}
λend{lstlisitng}

We use STL equal function to check if x is prefix of y.

The test case is nearly same as the one in Trie.
```

```
λbegin{lstlisting}
Patricia<std::string, std::string>* t(0);
const char* dict[] = {
  "a", "the␣first␣letter␣of␣English", λ
  "an", "used␣instead␣of␣'a'␣when␣the␣following␣word␣begins␣with␣a␣vowel␣sound", λ
  "another", "one␣more␣person␣or␣thing␣or␣an␣extra␣amount", λ
  "abandon", "to␣leave␣a␣place,␣thing␣or␣person␣forever", λ
  "about", "on␣the␣subject␣of;␣connected␣with", λ
  "adam", "a␣character␣in␣the␣Bible␣who␣was␣the␣first␣man␣made␣by␣God", λ
  "boy", "a␣male␣child␣or,␣more␣generally,␣a␣male␣of␣any␣age", λ
  "body", "the␣whole␣physical␣structure␣that␣forms␣a␣person␣or␣animal", λ
  "zoo", "an␣area␣in␣which␣animals,␣especially␣wild␣animals,␣are␣kept" λ
         "so␣that␣people␣can␣go␣and␣look␣at␣them,␣or␣study␣them"};

const char** first=dict;
const char** last =dict + sizeof(dict)/sizeof(char*);
for(;first!=last; ++first, ++first)
  t = insert(t, *first, *(first+1));
}

std::cout<<"test␣lookup␣top␣5␣in␣Trieλn"
         <<"search␣a␣"<<lop_to_str(lookup(t, "a", 5))<<"λn"
         <<"search␣ab␣"<<lop_to_str(lookup(t, "ab", 5))<<"λn";
delete t;
```

This test case will output a very same result in console.

Recursive algorithm of search top N candidate in Patricia

This algorithm can also be implemented recursively, if the string we are looking for is empty, we expand all children until we get N candidates. else we recursively examine the children of the node to see if we can find one has prefix as this string.

1: **function** PATRICIA-LOOK-UP-TOP-N(T, key, N)
2: **if** $T = NIL$ **then**
3: **return** NIL
4: **if** $KEY = NIL$ **then**
5: **return** $EXPAND - TOP - N(T, NIL, N)$
6: **else**
7: **return** $FIND{-}IN{-}CHILDREN{-}TOP{-}N(CHILDREN(T), key, N)$
8: **function** FIND-IN-CHILDREN-TOP-N(l, key, N)
9: **if** $l = NIL$ **then**
10: **return** NIL
11: **else if** $KEY(FIRST(l)) = key$ **then**
12: **return** $EXPAND - TOP - N(FIRST(l), key, N)$
13: **else if** $KEY(FIRST(l)$ is prefix of key **then**
14: **return** $PATRICIA - LOOK - UP - TOP - N(FIRST(l), key$ subtract $KEY(FIRST(l)))$
15: **else if** key is prefix of $KEY(FIRST(l)$ **then**
16: **return** $PATRICIA{-}LOOK{-}UP{-}TOP{-}N(FIRST(l), NIL, N)$
17: **else if**
18: **thenreturn** $FIND{-}IN{-}CHILDREN{-}TOP{-}N(REST(l), key, N)$

An e-dictionary in Haskell

In Haskell implementation, we provide a function named as findAll. Thanks for the lazy evaluation support, findAll won't produce all candidates words until we need them. we can use something like 'take 10 findAll' to get the top 10 words easily.

findAll is given as the following.

```
findAll:: Trie a → String → [(String, a)]
findAll t [] =
    case value t of
      Nothing → enum (children t)
      Just x  → ("", x):(enum (children t))
    where
      enum [] = []
      enum (p:ps) = (mapAppend (fst p) (findAll (snd p) [])) ++ (enum ps)
findAll t (k:ks) =
    case lookup k (children t) of
      Nothing → []
      Just t' → mapAppend k (findAll t' ks)

mapAppend x lst = map (λp→(x:(fst p), (snd p))) lst
```

function findAll take a Trie, a word to be looked up, it will output a list of pairs, the first element of the pair is the candidate word, the second element of the pair is the meaning of the word.

Compare with the find function of Trie, the none-trivial case is very similar. We take a letter form the words to be looked up, if there is no child starting with this letter, the program returns empty list. If there is such a child starting with this letter, this child should be a candidate. We use function mapAppend to add this letter in front of all elements of recursively founded candidate words.

In case we consumed all letters, we next returns all potential words, which means the program will traverse all children of the current node.

Note that only the node with value field not equal to 'None' is a meaningful word in our dictionary. We need append the list with the right meaning.

With this function, we can construct a very simple dictionary and return top 5 candidate to user. Here is the test program.

```
testFindAll = "λnlook up a: " ++ (show $ take 5 $findAll t "a") ++
              "λnlook up ab: " ++ (show $ take 5 $findAll t "ab")
    where
      t = fromList [
        ("a", "the first letter of English"),
        ("an", "used instead of 'a' when the following word begins with"
               "a vowel sound"),
        ("another", "one more person or thing or an extra amount"),
        ("abandon", "to leave a place, thing or person forever"),
        ("about", "on the subject of; connected with"),
        ("adam", "a character in the Bible who was the first man made by God"),
        ("boy", "a male child or, more generally, a male of any age"),
        ("body", "the whole physical structure that forms a person or animal"),
        ("zoo", "an area in which animals, especially wild animals, are kept"
                " so that people can go and look at them, or study them")]
```

```
main = do
    putStrLn testFindAll
```

This program will out put a result like this:

```
look up a: [("a","the first letter of English"),("an","used instead of 'a'
when the following word begins with a vowel sound"),("another","one more
person or thing or an extra amount"),("abandon","to leave a place, thing
or person forever"),("about","on the subject of; connected with")]
look up ab: [("abandon","to leave a place, thing or person forever"),
("about","on the subject of; connected with")]
```

The Trie solution wasts a lot of spaces. It is very easy to improve the above program with Patricia. Below source code shows the Patricia approach.

```
findAll' :: Patricia a → Key → [(Key, a)]
findAll' t [] =
    case value t of
      Nothing → enum $ children t
      Just x  → ("", x):(enum $ children t)
    where
      enum [] = []
      enum (p:ps) = (mapAppend' (fst p) (findAll' (snd p) [])) ++ (enum ps)
findAll' t k = find' (children t) k where
    find' [] _ = []
    find' (p:ps) k
          | (fst p) == k
              = mapAppend' k (findAll' (snd p) [])
          | (fst p) `Data.List.isPrefixOf` k
              = mapAppend' (fst p) (findAll' (snd p) (k `diff` (fst p)))
          | k `Data.List.isPrefixOf` (fst p)
              = findAll' (snd p) []
          | otherwise = find' ps k
    diff x y = drop (length y) x

mapAppend' s lst = map (λp→(s++(fst p), snd p)) lst
```

If compare this program with the one implemented by Trie, we can find they are very similar to each other. In none-trivial case, we just examine each child to see if any one match the key to be looked up. If one child is exactly equal to the key, we then expand all its sub branches and put them to the candidate list. If the child correspond to a prefix of the key, the program goes on find the the rest part of the key along this child and concatenate this prefix to all later results. If the current key is prefix to a child, the program will traverse this child and return all its sub branches as candidate list.

This program can be tested with the very same case as above, and it will output the same result.

An e-dictionary in Scheme/Lisp

In Scheme/Lisp implementation with Trie, a function named find is used to search all candidates start with a given string. If the string is empty, the program will enumerate all sub trees as result; else the program calls an inner function find-child to search a child which matches the first character of the given string.

Then the program recursively apply the find function to this child with the rest characters of the string to be searched.

```
(define (find t k)
  (define (find-child lst k)
    (tree (find-matching-item lst (lambda (c) (string=? (key c) k)))))
  (if (string-null? k)
      (enumerate t)
      (let ((t-new (find-child (children t) (string-car k))))
        (if (null? t-new) '()
          (map-string-append (string-car k) (find t-new (string-cdr k)))))))
```

Note that the map-string-append will insert the first character to all the elements (more accurately, each element is a pair with a key and a value, map-string-append insert the character in front of each key) in the result returned by recursive call. It is defined like this.

```
(define (map-string-append x lst) ;; lst: [(key value)]
  (map (lambda (p) (cons (string-append x (car p)) (cdr p))) lst))
```

The enumerate function which can expend all sub trees is implemented as the following.

```
(define (enumerate t) ;; enumerate all sub trees
  (if (null? t) '()
      (let ((res (append-map
                  (lambda (p)(map-string-append (key p)(enumerate (tree p))))
                  (children t))))
        (if (null? (value t)) res
            (cons (cons "" (value t)) res)))))
```

The test case is a very simple list of word-meaning pairs.

```
(define dict
  (list '("a" "the␣first␣letter␣of␣English")
        '("an" "used␣instead␣of␣'a'␣when␣the␣following␣word␣begins␣with␣a␣vowel␣sound")
        '("another" "one␣more␣person␣or␣thing␣or␣an␣extra␣amount")
        '("abandon" "to␣leave␣a␣place,␣thing␣or␣person␣forever")
        '("about" "on␣the␣subject␣of;␣connected␣with")
        '("adam" "a␣character␣in␣the␣Bible␣who␣was␣the␣first␣man␣made␣by␣God")
        '("boy" "a␣male␣child␣or,␣more␣generally,␣a␣male␣of␣any␣age")
        '("body" "the␣whole␣physical␣structure␣that␣forms␣a␣person␣or␣animal")
        '("zoo" "an␣area␣in␣which␣animals,␣especially␣wild␣animals,
␣␣are␣kept␣so␣that␣people␣can␣go␣and␣look␣at␣them,␣or␣study␣them")))
```

After feed this dict to a Trie, if user tries to find 'a*' or 'ab*' like below.

```
(define (test-trie-find-all)
  (define t (list→trie dict))
  (display "find␣a*:␣") (display (find t "a")) (newline)
  (display "find␣ab*:␣") (display (find t "ab")) (newline))
```

The result is a list with all candidates start with the given string.

```
(test-trie-find-all)
find a*: ((a . the first letter of English) (an . used instead of 'a'
when the following word begins with a vowel sound) (another . one more
person or thing or an extra amount) (abandon . to leave a place, thing
```

```
or person forever) (about . on the subject of; connected with) (adam
. a character in the Bible who was the first man made by God))
find ab*: ((abandon . to leave a place, thing or person forever)
(about . on the subject of; connected with))
```

Trie approach isn't space effective. Patricia can be one alternative to improve in terms of space.

We can fully reuse the function enumerate, map-string-append which are defined for trie. the find function for Patricia is implemented as the following.

```
(define (find t k)
  (define (find-child lst k)
    (if (null? lst) '()
        (cond ((string=? (key (car lst)) k)
               (map-string-append k (enumerate (tree (car lst)))))
              ((string-prefix? (key (car lst)) k)
               (let ((k-new (string-tail k (string-length (key (car lst))))))
                 (map-string-append (key (car lst)) (find (tree (car lst)) k-new))))
              ((string-prefix? k (key (car lst))) (enumerate (tree (car lst))))
              (else (find-child (cdr lst) k)))))
  (if (string-null? k)
      (enumerate t)
      (find-child (children t) k)))
```

If the same test cases of search all candidates of 'a*' and 'ab*' are fed we can get a very same result.

5.7.2 T9 input method

Most mobile phones around year 2000 has a key pad. To edit a short message/email with such key-pad, users typically have quite different experience from PC. Because a mobile-phone key pad, or so called ITU-T key pad has few keys. Figure fig:itut-keypad shows an example.

Figure 5.14: an ITU-T keypad for mobile phone.

There are typical 2 methods to input an English word/phrase with ITU-T key pad. For instance, if user wants to enter a word "home", He can press the key in below sequence.

- Press key '4' twice to enter the letter 'h';
- Press key '6' three times to enter the letter 'o';
- Press key '6' twice to enter the letter 'm';
- Press key '3' twice to enter the letter 'e';

Another high efficient way is to simplify the key press sequence like the following.

- Press key '4', '6', '6', '3', word "home" appears on top of the candidate list;
- Press key '*' to change a candidate word, so word "good" appears;
- Press key '*' again to change another candidate word, next word "gone" appears;
- ...

Compare these 2 method, we can see method 2 is much easier for the end user, and it is operation efficient. The only overhead is to store a candidate words dictionary.

Method 2 is called as T9 input method, or predictive input method [6], [7]. The abbreviation 'T9' stands for 'textonym'. In this section, I'll show an example implementation of T9 by using Trie and Patricia.

In order to provide candidate words to user, a dictionary must be prepared in advance. Trie or Patricia can be used to store the Dictionary. In the real commercial software, complex indexing dictionary is used. We show the very simple Trie and Patricia only for illustration purpose.

Iterative algorithm of T9 looking up

Below pseudo code shows how to realize T9 with Trie.

```
1:  function TRIE-LOOK-UP-T9(T, key)
2:      PUSH − BACK(Q, NIL, key, T)
3:      r ← NIL
4:      while Q is not empty do
5:          p, k, t ← POP − FRONT(Q)
6:          i ← FIRST − LETTER(k)
7:          for each c in T9 − MAPPING(i) do
8:              if c is in CHILDREN(t) then
9:                  k′ ← k subtract i
10:                 if k′ is empty then
11:                     APPEND(r, p + c)
12:                 else
13:                     PUSH − BACK(Q, p + c, k′, CHILDREN(t)[c])
14:     return r
```

This is actually a bread-first search program. It utilizes a queue to store the current node and key string we are examining. The algorithm takes the first digit from the key, looks up it in T9 mapping to get all English letters corresponding to this digit. For each letter, if it can be found in the children of current node, the node along with the English string found so far are push back to the queue. In case all digits are examined, a candidate is found. We'll append this candidate to the result list. The loop will terminate when the queue is empty.

Since Trie is not space effective, minor modification of the above program can work with Patricia, which can help to save extra spaces.

```
1: function PATRICIA-LOOK-UP-T9(T, key)
2:     PUSH − BACK(Q, NIL, key, T)
3:     r ← NIL
4:     while Q is not empty do
5:         p, k, t ← POP − FRONT(Q)
6:         for each child in CHILDREN(t) do
7:             k′ ← CONVERT − T9(KEY(child))
8:             if k′ IS-PREFIX-OF k then
9:                 if k′ = k then
10:                    APPEND(r, p + KEY(child))
11:                else
12:                    PUSH − BACK(Q, p + KEY(child), k − k′, child)
13:    return r
```

T9 implementation in Python

In Python implementation, T9 looking up is realized in a typical bread-first search algorithm as the following.

```
T9MAP={'2':"abc", '3':"def", '4':"ghi", '5':"jkl", λ
       '6':"mno", '7':"pqrs", '8':"tuv", '9':"wxyz"}

def trie_lookup_t9(t, key):
    if t is None or key == "":
        return None
    q = [("", key, t)]
    res = []
    while len(q)>0:
        (prefix, k, t) = q.pop(0)
        i=k[0]
        if not i in T9MAP:
            return None #invalid input
        for c in T9MAP[i]:
            if c in t.children:
                if k[1:]=="":
                    res.append((prefix+c, t.children[c].value))
                else:
                    q.append((prefix+c, k[1:], t.children[c]))
    return res
```

Function trie_lookup_t9 check if the parameters are valid first. Then it push the initial data into a queue. The program repeatedly pop the item from the

queue, including what node it will examine next, the number sequence string, and the alphabetic string it has been searched.

For each popped item, the program takes the next digit from the number sequence, and looks up in T9 map to find the corresponding English letters. With all these letters, if they can be found in the children of the current node, we'll push this child along with the updated number sequence string and updated alphabetic string into the queue. In case we process all numbers, we find a candidate result.

We can verify the above program with the following test cases.

```
class LookupTest:
    def __init__(self):
        t9dict = ["home", "good", "gone", "hood", "a", "another", "an"]
        self.t9t = trie.list_to_trie(t9dict)

    def test_trie_t9(self):
        print "search␣4␣", trie_lookup_t9(self.t9t, "4")
        print "search␣46␣", trie_lookup_t9(self.t9t, "46")
        print "search␣4663␣", trie_lookup_t9(self.t9t, "4663")
        print "search␣2␣", trie_lookup_t9(self.t9t, "2")
        print "search␣22␣", trie_lookup_t9(self.t9t, "22")
```

If we run the test, it will output a very same result as the above Haskell program.

```
search 4  [('g', None), ('h', None)]
search 46  [('go', None), ('ho', None)]
search 4663  [('gone', None), ('good', None), ('home', None), ('hood', None)]
search 2  [('a', None)]
search 22  []
```

To save the spaces, Patricia can be used instead of Trie.

```
def patricia_lookup_t9(t, key):
    if t is None or key == "":
        return None
    q = [("", key, t)]
    res = []
    while len(q)>0:
        (prefix, key, t) = q.pop(0)
        for k, tr in t.children.items():
            digits = toT9(k)
            if string.find(key, digits)==0: #is prefix of
                if key == digits:
                    res.append((prefix+k, tr.value))
                else:
                    q.append((prefix+k, key[len(k):], tr))
    return res
```

Compare to the implementation with Trie, they are very similar. We also used a bread-first search approach. The different part is that we convert the string of each child to number sequence string according to T9 mapping. if it is prefix of the key we are looking for, we push this child along with updated key and prefix. In case we examined all digits, we find a candidate result.

The convert function is a reverse mapping process as below.

```
def toT9(s):
    res=""
    for c in s:
        for k, v in T9MAP.items():
            if string.find(v, c)>=0:
                res+=k
                break
        #error handling skipped.
    return res
```

For illustration purpose, the error handling for invalid letters is skipped. If we feed the program with the same test cases, we can get a result as the following.

```
search 4  []
search 46  [('go', None), ('ho', None)]
search 466  []
search 4663  [('good', None), ('gone', None), ('home', None), ('hood', None)]
search 2  [('a', None)]
search 22  []
```

The result is slightly different from the one output by Trie. The reason is as same as what we analyzed in Haskell implementation. It is easily to modify the program to output a similar result.

T9 implemented in C++

First we define T9 mapping as a Singleton object, this is because we want to it can be used both in Trie look up and Patricia look up programs.

```
struct t9map{
  typedef std::map<char, std::string> Map;
  Map map;

  t9map(){
    map['2']="abc";
    map['3']="def";
    map['4']="ghi";
    map['5']="jkl";
    map['6']="mno";
    map['7']="pqrs";
    map['8']="tuv";
    map['9']="wxyz";
  }

  static t9map& inst(){
    static t9map i;
    return i;
  }
};
```

Note in other languages or keypad layout, we can define different mappings and pass them as an argument to the looking up function.

With this mapping, the looking up in Trie can be given as below. Although we want to keep the genericity of the program, for illustration purpose, we just simply use the t9 mapping directly.

In order to keep the code as short as possible, a boost library tool, boost::tuple is used. For more about boost::tuple, please refer to [8].

```
template<class K, class V>
std::list<std::pair<K, V> > lookup_t9(Trie<K, V>* t,
                                      typename Trie<K, V>::KeyType key)
{
  typedef std::list<std::pair<K, V> > Result;
  typedef typename Trie<K, V>::KeyType Key;
  typedef typename Trie<K, V>::Char Char;

  if((!t) || key.empty())
    return Result();

  Key prefix;
  std::map<Char, Key> m = t9map::inst().map;
  std::queue<boost::tuple<Key, Key, Trie<K, V>*> > q;
  q.push(boost::make_tuple(prefix, key, t));
  Result res;
  while(!q.empty()){
    boost::tie(prefix, key, t) = q.front();
    q.pop();
    Char c = *key.begin();
    key = Key(key.begin()+1, key.end());
    if(m.find(c) == m.end())
      return Result();
    Key cs = m[c];
    for(typename Key::iterator it=cs.begin(); it!=cs.end(); ++it)
      if(t->children.find(*it)!=t->children.end()){
        if(key.empty())
          res.push_back(std::make_pair(prefix+*it, t->children[*it]->value));
        else
          q.push(boost::make_tuple(prefix+*it, key, t->children[*it]));
      }
  }
  return res;
}
```

This program will first check if the Patricia tree or the key are empty to deal with with trivial case. It next initialize a queue, and push one tuple to it. the tuple contains 3 elements, a prefix to represent a string the program has been searched, current key it need look up, and a node it will examine.

Then the program repeatedly pops the tuple from the queue, takes the first character from the key, and looks up in T9 map to get a candidate English letter list. With each letter in this list, the program examine if it exists in the children of current node. In case it find such a child, if there is no left letter to look up, it means we found a candidate result, we push it to the result list. Else, we create a new tuple with updated prefix, key and this child; the push it to the queue for later process.

Below are some simple test cases for verification.

```
Trie<std::string, std::string>* t9trie(0);
const char* t9dict[] = {"home", "good", "gone", "hood", "a", "another", "an"};
t9trie = list_to_trie(t9dict, t9dict+sizeof(t9dict)/sizeof(char*), t9trie);
std::cout<<"test␣t9␣lookup␣in␣Trie\n"
         <<"search␣4␣"<<lop_to_str(lookup_t9(t9trie, "4"))<<"\n"
         <<"serach␣46␣"<<lop_to_str(lookup_t9(t9trie, "46"))<<"\n"
         <<"serach␣4663␣"<<lop_to_str(lookup_t9(t9trie, "4663"))<<"\n"
         <<"serach␣2␣"<<lop_to_str(lookup_t9(t9trie, "2"))<<"\n"
         <<"serach␣22␣"<<lop_to_str(lookup_t9(t9trie, "22"))<<"\n\n";
delete t9trie;
```

It will output the same result as the Python program.

```
test t9 lookup in Trie
search 4 [(g, ), (h, ), ]
serach 46 [(go, ), (ho, ), ]
serach 4663 [(gone, ), (good, ), (home, ), (hood, ), ]
serach 2 [(a, ), ]
serach 22 []
```

In order to save space, a looking up program for Patricia is also provided.

```
template<class K, class V>
std::list<std::pair<K, V> > lookup_t9(Patricia<K, V>* t,
                                      typename Patricia<K, V>::KeyType key)
{
  typedef std::list<std::pair<K, V> > Result;
  typedef typename Patricia<K, V>::KeyType Key;
  typedef typename Key::value_type Char;
  typedef typename Patricia<K, V>::Children::iterator Iterator;

  if((!t) || key.empty())
    return Result();

  Key prefix;
  std::map<Char, Key> m = t9map::inst().map;
  std::queue<boost::tuple<Key, Key, Patricia<K, V>*> > q;
  q.push(boost::make_tuple(prefix, key, t));
  Result res;
  while(!q.empty()){
    boost::tie(prefix, key, t) = q.front();
    q.pop();
    for(Iterator it=t->children.begin(); it!=t->children.end(); ++it){
      Key digits = t9map::inst().toT9(it->first);
      if(is_prefix_of(digits, key)){
        if(digits == key)
          res.push_back(std::make_pair(prefix+it->first, it->second->value));
        else{
          key =Key(key.begin()+it->first.size(), key.end());
          q.push(boost::make_tuple(prefix+it->first, key, it->second));
        }
      }
    }
  }
  return res;
```

```
}
```

The program is similar to the one with Trie very much. This is a typical bread-first search approach. Note that we added a member function to_t9() to convert a English word/phrase back to digit number string. This member function is implemented as the following.

```
struct t9map{
  //...
  std::string to_t9(std::string s){
    std::string res;
    for(std::string::iterator c=s.begin(); c!=s.end(); ++c){
      for(Map::iterator m=map.begin(); m!=map.end(); ++m){
        std::string val = m->second;
        if(std::find(val.begin(), val.end(), *c)!=val.end()){
          res.push_back(m->first);
          break;
        }
      }
    } // skip error handling.
    return res;
  }
```

The error handling for invalid letters is omitted in order to keep the code short for easy understanding. We can use the very similar test cases as above except we need change the Trie to Patrica. It will output as below.

```
test t9 lookup in Patricia
search 4 []
serach 46 [(go, ), (ho, ), ]
serach 466 []
serach 4663 [(gone, ), (good, ), ]
serach 2 [(a, ), ]
serach 22 []
```

The result is slightly different, please refer to the Haskell section for the reason of this difference. It is very easy to modify the program to output the very same result as Trie's one.

Recursive algorithm of T9 looking up

T9 implemented in Haskell

In Haskell, we first define a map from key pad to English letter. When user input a key pad number sequence, we take each number and check from the Trie. All children match the number should be investigated. Below is a Haskell program to realize T9 input.

```
mapT9 = [('2', "abc"), ('3', "def"), ('4', "ghi"), ('5', "jkl"),
         ('6', "mno"), ('7', "pqrs"), ('8', "tuv"), ('9', "wxyz")]

lookupT9 :: Char → [(Char, b)] → [(Char, b)]
lookupT9 c children = case lookup c mapT9 of
        Nothing → []
        Just s  → foldl f [] s where
```

```
            f lst x = case lookup x children of
                Nothing → lst
                Just t  → (x, t):lst

-- T9-find in Trie
findT9:: Trie a → String → [(String, Maybe a)]
findT9 t [] = [("", Trie.value t)]
findT9 t (k:ks) = foldl f [] (lookupT9 k (children t))
    where
      f lst (c, tr) = (mapAppend c (findT9 tr ks)) ++ lst
```

findT9 is the main function, it takes 2 parameters, a Trie and a number sequence string. In non-trivial case, it calls lookupT9 function to examine all children which match the first number.

For each matched child, the program recursively calls findT9 on it with the left numbers, and we use mapAppend to insert the currently finding letter in front of all results. The program use foldl to combine all these together.

Function lookupT9 is used to filtered all possible children who match a number. It first call lookup function on mapT9, so that a string of possible English letters can be identified. Next we call lookup for each candidate letter to see if there is a child can match the letter. We use foldl to collect all such child together.

This program can be verified by using some simple test cases.

```
testFindT9 = "press 4: " ++ (show $ take 5 $ findT9 t "4")++
             "λnpress 46: " ++ (show $ take 5 $ findT9 t "46")++
             "λnpress 4663: " ++ (show $ take 5 $ findT9 t "4663")++
             "λnpress 2: " ++ (show $ take 5 $ findT9 t "2")++
             "λnpress 22: " ++ (show $ take 5 $ findT9 t "22")
    where
      t = Trie.fromList lst
      lst = [("home", 1), ("good", 2), ("gone", 3), ("hood", 4),
             ("a", 5), ("another", 6), ("an", 7)]
```

The program will output below result.

```
press 4: [("g",Nothing),("h",Nothing)]
press 46: [("go",Nothing),("ho",Nothing)]
press 4663: [("gone",Just 3),("good",Just 2),("home",Just 1),("hood",Just 4)]
press 2: [("a",Just 5)]
press 22: []
```

The value of each child is just for illustration, we can put empty value instead and only returns candidate keys for a real input application.

Tries consumes to many spaces, we can provide a Patricia version as alternative.

```
findPrefixT9' :: String → [(String, b)] → [(String, b)]
findPrefixT9' s lst = filter f lst where
    f (k, _) = (toT9 k) `Data.List.isPrefixOf` s

toT9 :: String → String
toT9 [] = []
toT9 (x:xs) = (unmapT9 x mapT9):(toT9 xs) where
```

```
    unmapT9 x (p:ps) = if x `elem` (snd p) then (fst p) else unmapT9 x ps

findT9' :: Patricia a → String → [(String, Maybe a)]
findT9' t [] = [("", value t)]
findT9' t k = foldl f [] (findPrefixT9' k (children t))
    where
      f lst (s, tr) = (mapAppend' s (findT9' tr (k `diff` s))) ++ lst
      diff x y = drop (length y) x
```

In this program, we don't check one digit at a time, we take all the digit sequence, and we examine all children of the Patricia node. For each child, the program convert the prefix string to number sequence by using function toT9, if the result is prefix of what user input, we go on search in this child and append the prefix in front of all further results.

If we tries the same test case, we can find the result is a bit different.

```
press 4: []
press 46: [("go",Nothing),("ho",Nothing)]
press 466: []
press 4663: [("good",Just 2),("gone",Just 3),("home",Just 1),("hood",Just 4)]
press 2: [("a",Just 5)]
press 22: []
```

If user press key '4', because the dictionary (represent by Patricia) doesn't contain any candidates matches it, user will get an empty candidates list. The same situation happens when he enters "466". In real input method implementation, such user experience isn't good, because it displays nothing although user presses the key several times. One improvement is to predict what user will input next by display a partial result. This can be easily achieved by modify the above program. (Hint: not only check

```
findPrefixT9' s lst = filter f lst where
    f (k, _) = (toT9 k) `Data.List.isPrefixOf` s
```

but also check

```
    f (k, _) = s `Data.List.isPrefixOf` (toT9 k)
```

)

T9 implemented in Scheme/Lisp

In Scheme/Lisp, T9 map is defined as a list of pairs.

```
(define map-T9 (list '("2" "abc") '("3" "def") '("4" "ghi") '("5" "jkl")
                     '("6" "mno") '("7" "pqrs") '("8" "tuv") '("9" "wxyz")))
```

The main searching function is implemented as the following.

```
(define (find-T9 t k) ;; return [(key value)]
  (define (accumulate-find lst child)
    (append (map-string-append (key child) (find-T9 (tree child) (string-cdr k)))
            lst))
  (define (lookup-child lst c) ;; lst, list of childen [(key tree)], c, char
    (let ((res (find-matching-item map-T9 (lambda (x) (string=? c (car x))))))
      (if (not res) '()
```

```
          (filter (lambda (x) (substring? (key x) (cadr res))) lst))))
  (if (string-null? k) (list (cons k (value t)))
      (fold-left accumulate-find '() (lookup-child (children t) (string-car k)))))
```

This function contains 2 inner functions. If the string is empty, the program returns a one element list. The element is a string value pair. For the none trivial case, the program will call inner function to find in each child and then put them together by using fold-left high order function.

To test this T9 search function, a very simple dictionary is established by using Trie insertion. Then we test by calling find-T9 function on several digits sequences.

```
(define dict-T9 (list '("home" ()) '("good" ()) '("gone" ()) '("hood" ())
                      '("a" ()) '("another" ()) '("an" ())))

(define (test-trie-T9)
  (define t (list→trie dict-T9))
  (display "find␣4:␣") (display (find-T9 t "4")) (newline)
  (display "find␣46:␣") (display (find-T9 t "46")) (newline)
  (display "find␣4663:␣") (display (find-T9 t "4663")) (newline)
  (display "find␣2:␣") (display (find-T9 t "2")) (newline)
  (display "find␣22:␣") (display (find-T9 t "22")) (newline))
```

Evaluate this test function will output below result.

```
find 4: ((g) (h))
find 46: ((go) (ho))
find 4663: ((gone) (good) (hood) (home))
find 2: ((a))
find 22: ()
```

In order to be more space effective, Patricia can be used to replace Trie. The search program modified as the following.

```
(define (find-T9 t k)
  (define (accumulate-find lst child)
    (append (map-string-append (key child) (find-T9 (tree child) (string- k (key child))))
            lst))
  (define (lookup-child lst k)
    (filter (lambda (child) (string-prefix? (str→t9 (key child)) k)) lst))
  (if (string-null? k) (list (cons "" (value t)))
      (fold-left accumulate-find '() (lookup-child (children t) k))))
```

In this program a string helper function 'string-' is defined to get the different part of two strings. It is defined like below.

```
(define (string- x y)
  (string-tail x (string-length y)))
```

Another function is 'str-¿t9' it will convert a alphabetic string back to digit sequence base on T9 mapping.

```
(define (str→t9 s)
  (define (unmap-t9 c)
    (car (find-matching-item map-T9 (lambda (x) (substring? c (cadr x))))))
  (if (string-null? s) ""
      (string-append (unmap-t9 (string-car s)) (str→t9 (string-cdr s)))))
```

We can feed the almost same test cases, and the result is output as the following.

```
find 4: ()
find 46: ((go) (ho))
find 466: ()
find 4663: ((good) (gone) (home) (hood))
find 2: ((a))
find 22: ()
```

Note the result is a bit different, the reason is described in Haskell section. It is easy to modify the program, so that Trie and Patricia approach give the very same result.

5.8 Short summary

In this post, we start from the Integer base trie and Patricia, the map data structure based on integer patricia plays an important role in Compiler implementation. Next, alphabetic Trie and Patricia are given, and I provide a example implementation to illustrate how to realize a predictive e-dictionary and a T9 input method. Although they are far from the real implementation in commercial software. They show a very simple approach of manipulating text. There are still some interesting problem can not be solved by Trie or Patrica directly, how ever, some other data structures such as suffix tree have close relationship with them. I'll note something about suffix tree in other post.

5.9 Appendix

All programs provided along with this article are free for downloading.

5.9.1 Prerequisite software

GNU Make is used for easy build some of the program. For C++ and ANSI C programs, GNU GCC and G++ 3.4.4 are used. I use boost triple to reduce the amount of our code lines, boost library version I am using is 1.33.1. The path is in CXX variable in Makefile, please change it to your path when compiling. For Haskell programs GHC 6.10.4 is used for building. For Python programs, Python 2.5 is used for testing, for Scheme/Lisp program, MIT Scheme 14.9 is used.

all source files are put in one folder. Invoke 'make' or 'make all' will build C++ and Haskell program.

Run 'make Haskell' will separate build Haskell program. There will be two executable file generated one is htest the other is happ (with .exe in Window like OS). Run htest will test functions in IntTrie.hs, IntPatricia.hs, Trie.hs and Patricia.hs. Run happ will execute the editionary and T9 test cases in EDict.hs.

Run 'make cpp' will build c++ program. It will create a executable file named cpptest (with .exe in Windows like OS). Run this program will test inttrie.hpp, intpatricia.hpp, trie.hpp, patricia.hpp, and edict.hpp.

Run 'make c' will build the ANSI C program for Trie. It will create a executable file named triec (with .exe in Windows like OS).

Python programs can run directly with interpreter.

Scheme/Lisp program need be loaded into Scheme evaluator and evaluate the final function in the program. Note that patricia.scm will hide some functions defined in trie.scm.

Here is a detailed list of source files

5.9.2 Haskell source files

- IntTrie.hs, Haskell version of little-endian integer Trie.
- IntPatricia.hs, integer Patricia tree implemented in Haskell.
- Trie.hs, Alphabetic Trie, implemented in Haskell.
- Patricia.hs, Alphabetic Patricia, implemented in Haskell.
- TestMain.hs, main module to test the above 4 programs.
- EDict.hs, Haskell program for e-dictionary and T9.

5.9.3 C++/C source files

- inttrie.hpp, Integer base Trie;
- intpatricia.hpp, Integer based Patricia tree;
- trie.c, Alphabetic Trie only for lowercase English language, implemented in ANSI C.
- trie.hpp, Alphabetic Trie;
- patricia.hpp, Alphabetic Patricia;
- trieutil.hpp, Some generic utilities;
- edit.hpp, e-dictionary and T9 implemented in C++;
- test.cpp, main program to test all above programs.

5.9.4 Python source files

- inttrie.py, Python version of little-endian integer Trie, with test cases;
- intpatricia.py, integer Patricia tree implemented in Python;
- trie.py, Alphabetic Trie, implemented in Python;
- patricia.py, Alphabetic Patricia implemented in Python;
- trieutil.py, Common utilities;
- edict.py, e-dictionary and T9 implemented in Python.

5.9.5 Scheme/Lisp source files

- inttrie.scm, Little-endian integer Trie, implemented in Scheme/Lisp;
- intpatricia.scm, Integer based Patricia tree;
- trie.scm, Alphabetic Trie;
- patricia.scm, Alphabetic Patricia, reused many definitions in Trie;
- trieutil.scm, common functions and utilities.

5.9.6 Tools

Besides them, I use graphviz to draw most of the figures in this post. In order to translate the Trie, Patrica and Suffix Tree output to dot language scripts. I wrote a python program. it can be used like this.

```
trie2dot.py -o foo.dot -t patricia "1:x, 4:y, 5:z"
trie2dot.py -o foo.dot -t trie "001:one, 101:five, 100:four"
```

This helper scripts can also be downloaded with this article.
download position: http://sites.google.com/site/algoxy/trie/trie.zip

Bibliography

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

[2] Chris Okasaki and Andrew Gill. "Fast Mergeable Integer Maps". Workshop on ML, September 1998, pages 77-86, http://www.cse.ogi.edu/ andy/pub-/finite.htm

[3] D.R. Morrison, "PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric", Journal of the ACM, 15(4), October 1968, pages 514-534.

[4] Suffix Tree, Wikipedia. http://en.wikipedia.org/wiki/Suffix_tree

[5] Trie, Wikipedia. http://en.wikipedia.org/wiki/Trie

[6] T9 (predictive text), Wikipedia. http://en.wikipedia.org/wiki/T9_(predictive_text)

[7] Predictive text, Wikipedia. http://en.wikipedia.org/wiki/Predictive_text

[8] Bjorn Karlsson. "Beyond the C++ Standard Library: An Introduction to Boost". Addison Wesley Professional, August 31, 2005, ISBN: 0321133544

Suffix Tree with Functional and imperative implementation
Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

Chapter 6

Suffix Tree with Functional and imperative implementation

6.1 abstract

Suffix Tree is an important data structure. It is quite powerful in string and DNA information manipulations. Suffix Tree is introduced in 1973. The latest on-line construction algorithm was found in 1995. This post collects some existing result of suffix tree, including the construction algorithms as well as some typical applications. Some imperative and functional implementation are given. There are multiple programming languages used, including C++, Haskell, Python and Scheme/Lisp.

There may be mistakes in the post, please feel free to point out.

Keywords: Suffix Tree

6.2 Introduction

Suffix Tree is a special Patricia. There is no such a chapter in CLRS book. To introduce suffix tree together with Trie and Patricia will be a bit easy to understand.

As a data structure, Suffix tree allows for particularly fast implementation

of many important string operations[2]. And it is also widely used in bio-information area such as DNA pattern matching[3].

The suffix tree for a string S is a Patricia tree, with each edges are labeled with some sub-string of S. Each suffix of S corresponds to exactly one path from root to a leaf. Figure 6.1 shows the suffix tree for an English word 'banana'.

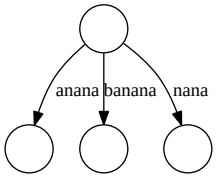


Figure 6.1: The suffix tree for 'banana'

Note that all suffixes, 'banana', 'anana', 'nana', 'ana', 'na', 'a', '' can be looked up in the above tree. Among them the first 3 suffixes are explicitly shown; others are implicitly represented. The reason for why 'ana, 'na', 'a', and '' are not shown explicitly is because they are prefixes of some edges. In order to make all suffixes shown explicitly, we can append a special pad terminal symbol, which is not seen to the string. Such terminator is typically denoted as '$'. With this method, no suffix will be a prefix of the others. In this post, we won't use terminal symbol for most cases.

It's very interesting that compare to the simple suffix tree for 'banana', the suffix tree for 'bananas' is quite different as shown in figure 6.2.

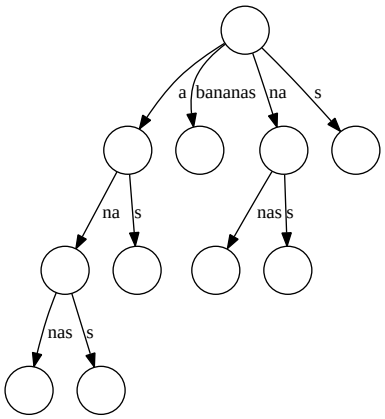


Figure 6.2: The suffix tree for 'bananas'

In this post, I'll first introduce about suffix Trie, and give the trivial method about how to construct suffix Trie and tree. Trivial methods utilize the insertion algorithm for normal Trie and Patricia. They need much of computation and spaces. Then, I'll explain about the on-line construction for suffix Trie by using suffix link concept. After that, I'll show Ukkonen's method, which is a linear time on-line construction algorithm. For both suffix Trie and suffix tree, functional approach is provided as well as the imperative one. In the last section,

I'll list some typical string manipulation problems and show how to solve them with suffix tree.

This article provides example implementation in C, C++, Haskell, Python, and Scheme/Lisp languages.

All source code can be downloaded in appendix 8.7, please refer to appendix for detailed information about build and run.

6.3 Suffix Trie

Just likes the relationship between Trie and Patricia, Suffix Trie has much simpler structure than suffix tree. Figure 6.3 shows the suffix Trie of 'banana'.

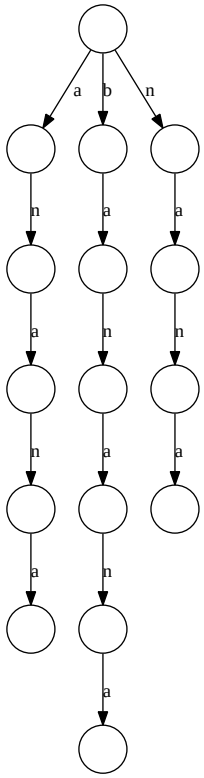


Figure 6.3: Suffix Trie of 'banana'

Compare with figure 6.1, the difference is that, instead of representing a word for each edge, one edge in suffix Trie only represents one character. Thus suffix Trie need more spaces. If we pack all nodes which has only one child, the suffix Trie can be turned into a suffix tree.

Suffix Trie can be a good start point for explaining the suffix tree construction algorithm.

6.3.1 Trivial construction methods of Suffix Tree

By repeatedly applying the insertion algorithms[5] for Trie and Patricia on each suffixes of a word, Suffix Trie and tree can be built in a trivial way.

Below algorithm illustrates this approach for suffix tree.

1: **function** TRIVIAL-SUFFIX-TREE(S)
2: $T \leftarrow NIL$
3: **for** i from 1 to $LENGTH(S)$ **do**
4: $T \leftarrow PATRICIA - INSERT(T, RIGHT(S, i))$
5: **return** T

Where function $RIGHT(S, i)$ will extract sub-string of S from left to right most. Similar functional algorithm can also be provided in this way.

1: **function** TRIVIAL-SUFFIX-TREE'(S)
2: **return** $FOLD - LEFT(PATRICIA - INSERT, NIL, TAILS(S))$

Function TAILS() returns a list of all suffixes for string S. In Haskell, Module Data.List provides this function already. In Scheme/Lisp, it can be implemented as below.

```
(define (tails s)
  (if (string-null? s)
      '("")
      (cons s (tails (string-tail s 1)))))
```

The trivial suffix Trie/tree construction method takes $O(n^2)$ time, where n is the length of the word. Although the string manipulation can be very fast by using suffix tree, slow construction will be the bottleneck of the whole process.

6.3.2 On-line construction of suffix Trie

Analysis of construction for suffix Trie can be a good start point in finding the linear time suffix tree construction algorithm. In Ukkonen's paper[1], finite-state automation, transition function and suffix function are used to build the mathematical model for suffix Trie/tree.

In order to make it easy for understanding, let's explain the above concept with the elements of Trie data structure.

With a set of alphabetic, a string with length n can be defined as $S = s_1 s_2 ... s_n$. And we define $S[i] = s_1 s_2 ... s_i$, which contains the first i characters.

In a suffix Trie, each node represents a suffix string. for example in figure 6.4, node X represents suffix 'a', by adding a character 'c', node X transfers to node Y which represents suffix 'ac'. We say node X and edge labeled 'c' transfers to node Y. This relationship can be denoted in pseudo code as below.

$Y \leftarrow CHILDREN(X)[c]$

It's equal to the following C++ and Python code.

```
y = x.children[c]
```

We also say that node x has a c-child y.

If a node A in a suffix Trie represents for a suffix $s_i s_{i+1} ... s_n$, and node B represents for suffix $s_{i+1} s_{i+2} ... s_n$, we say node B represents for the suffix of node A. We can create a link from A to B. This link is defined as the suffix link of node A. In this post, we show suffix link in dotted style. In figure 6.4, suffix link of node A points to node B, and suffix link of node B points to node

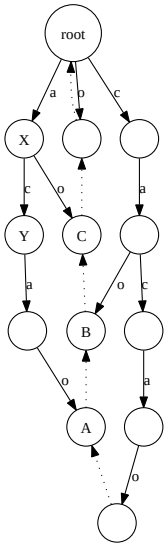


Figure 6.4: node $X \leftarrow$ "a", node $Y \leftarrow$ "ac", X transfers to Y with character 'c'

suffix string
$s_1 s_2 ... s_i$
$s_2 s_3 ... s_i$
...
$s_{i-1} s_i$
s_i
""

Table 6.1: suffixes for S_i

C. Suffix link is an important tool in Ukkonen's on-line construction algorithm, it is also used in some other algorithms running on the suffix tree.

on-line construction algorithm for suffix Trie

For a string S, Suppose we have construct suffix Trie for its i-th prefix $s_1 s_2 ... s_i$. We denote the suffix Trie for this i-th prefix as $SuffixTrie(S_i)$. Let's consider how can we obtain $SuffixTrie(S_{i+1})$ from $SuffixTrie(S_i)$.

If we list all suffixes corresponding to $SuffixTrie(S_i)$, from the longest S_i to the shortest empty string, we get table 6.1. There are total $i+1$ suffixes.

The most straightforward way is to append s_{i+1} to each of the suffix in above table, and add a new empty suffix. This operation can be implemented as create a new node, and append the new node as a child with edge bind to character s_{i+1}.

Algorithm 1 Initial version of update $SuffixTrie(S_i)$ to $SuffixTrie(S_{i+1})$.

1: **for** each *node* in $SuffixTrie(S_i)$ **do**
2: $\quad CHILDREN(node)[s_{i+1}] \leftarrow CREATE - NEW - NODE()$

However, some node in $SuffixTrie(S_i)$ may have already s_{i+1}-child. For example, in figure 6.5, Node X and Y are corresponding for suffix 'cac' and 'ac', they don't have 'a'-child. While node Z, which represents for suffix 'c' has 'a'-child already.

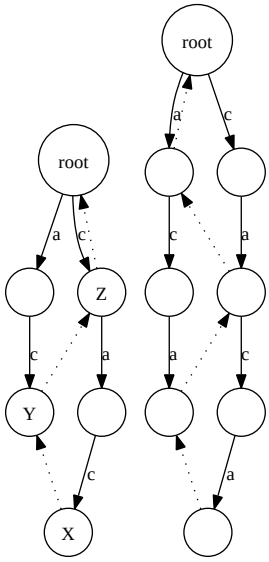


Figure 6.5: Suffix Trie of "cac" and "caca"

When we append s_{i+1}, in this case it is 'a', to $SuufixTrie(S_i)$. We need create a new node and append the new node to X and Y, however, we needn't create new node to Z, because node Z has already a child node with edge 'a'. So $SuffixTrie(S_{i+1})$, in this case it is for "caca", is shown in right part of figure 6.5.

If we check each node as the same order as in table 6.1, we can stop immediately once we find a node which has a s_{i+1}-child. This is because if a node X in $SuffixTrie(S_i)$ has already a s_{i+1}-child, then according to the definition of suffix link, all suffix nodes X' of X in $SuffixTrie(S_i)$ must also have s_{i+1}-child. In other words, let $c = s_{i+1}$, if wc is a sub-string of S_i, then every suffix of wc is also a sub-string of S_i [1]. The only exception is root node, which represents for empty string "".

According to this fact, we can refine the algorithm 1 to

Algorithm 2 Revised version of update $SuffixTrie(S_i)$ to $SuffixTrie(S_{i+1})$.

```
1: for each node in SuffixTrie(S_i) in descending order of suffix length do
2:     if CHILDREN(node)[s_{i+1}] = NIL then
3:         CHILDREN(node)[s_{i+1}] ← CREATE − NEW − NODE()
4:     else
5:         break
```

The next unclear question is how to iterate all nodes in $SuffixTrie(S_i)$ in descending order of suffix string length? We can define the top of a suffix Trie as

the deepest leaf node, by using suffix link for each node, we can traverse suffix Trie until the root. Note that the top of $SuffixTrie(NIL)$ is root, so we can get a final version of on-line construction algorithm for suffix Trie.

function INSERT(top, c)
  if $top = NIL$ **then**
    $top \leftarrow CREATE - NEW - NODE()$
  $node \leftarrow top$
  $node' \leftarrow CREATE - NEW - NODE()$
  while $node \neq NIL \wedge CHILDREN(node)[c] = NIL$ **do**
    $CHILDREN(node)[c] \leftarrow CREATE - NEW - NODE()$
    $SUFFIX - LINK(node') \leftarrow CHILDREN(node)[c]$
    $node' \leftarrow CHILDREN(node)[c]$
    $node \leftarrow SUFFIX - LINK(node)$
  if $node \neq NIL$ **then**
    $SUFFIX - LINK(node') \leftarrow CHILDREN(node)[c]$
  return $CHILDREN(top)[c]$

The above function $INSERT()$, can update $SuffixTrie(S_i)$ to $SuffixTrie(S_{i+1})$. It receives two parameters, one is the top node of $SuffixTrie(S_i)$, the other is the character of s_{i+1}. If the top node is NIL, which means that there is no root node yet, it create the root node then. Compare to the algorithm given by Ukkonen [1], I use a dummy node $node'$ to keep tracking the previous created new node. In the main loop, the algorithm check each node to see if it has s_{i+1}-child, if not, it will create new node, and bind the edge to character s_{i+1}. Then it go up along the suffix link until either arrives at root node, or find a node which has s_{i+1}-child already. After the loop, if the node point to some where in the Trie, the algorithm will make the last suffix link point to that place. The new top position is returned as the final result.

and the main part of the algorithm is as below:

```
1: procedure SUFFIX-TRIE(S)
2:     t ← NIL
3:     for i from 1 to LENGTH(S) do
4:         t ← INSERT(t, s_i)
```

Figure 6.6 shows the phases of on-line construction of suffix Trie for "cacao". Only the last layer of suffix links are shown.

According to the suffix Trie on-line construction process, the computation time is proportion to the size of suffix Trie. However, in the worse case, this is $O(n^2)$, where $n = LENGTH(S)$. One example is $S = a^n b^n$, that there are n characters of a and n characters of b.

Suffix Trie on-line construction program in Python and C++

The above algorithm can be easily implemented with imperative languages such as C++ and Python. In this section, I'll first give the defitions of the suffix Trie node. After that, I'll show the algorithms. In order to test and verify the programs, I'll provide some helper functions to print the Trie as human readable strings and give some look-up function as well.

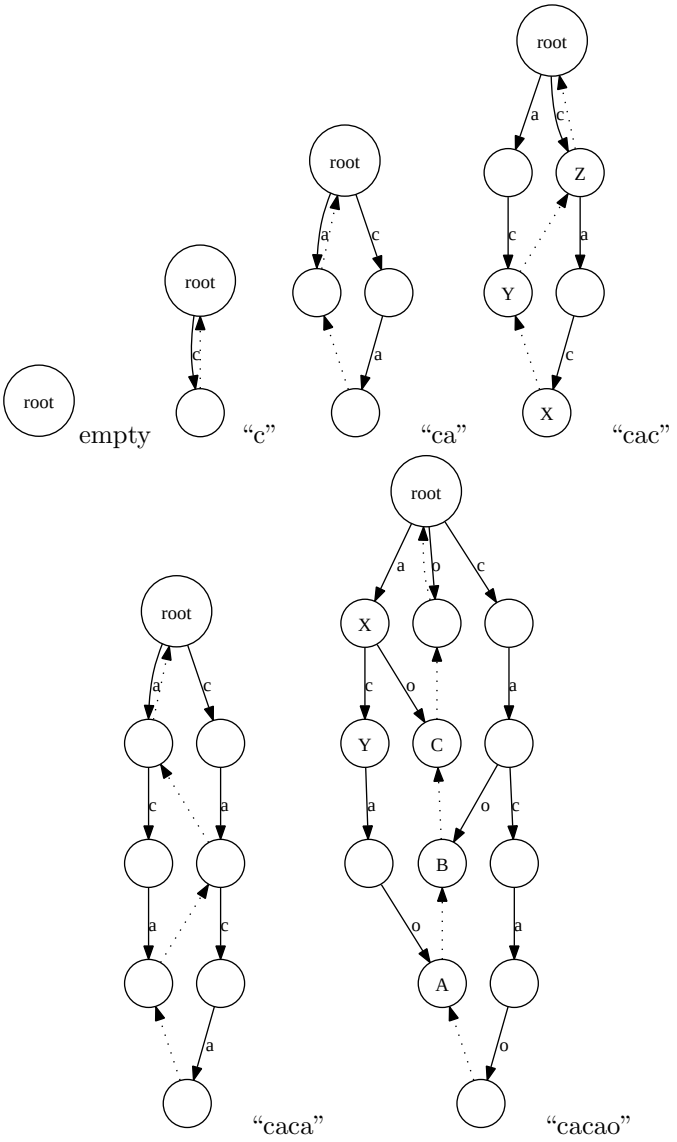


Figure 6.6: Construction of suffix Trie for "cacao", the 6 phases are shown, only the last layer of suffix links are shown in dotted arrow.

Definition of suffix Trie node in Python

With Python programming language, we can define the node in suffix Trie with two fields, one is a dictionary of children, the key is the character binding to an edge, and the value is the child node. The other field is suffix link, it points to a node which represents for the suffix string of this one.

```
class STrie:
    def __init__(self, suffix=None):
        self.children = {}
        self.suffix = suffix
```

By default, the suffix link for a node is initialized as empty, it will be set during the main construction algorithm.

Definition of suffix Trie node in C++

Suffix Trie on-line construction algorithm in Python

The main algorithm of updating $SuffixTrie(S_i)$ to $SuffixTrie(S_{i+1})$ is as below. It takes the top position node and the character to be updated as parameters.

```
def insert(top, c):
    if top is None:
        top=STrie()
    node = top
    new_node = STrie() #dummy init value
    while (node is not None) and (c not in node.children):
        new_node.suffix = node.children[c] = STrie(node)
        new_node = node.children[c]
        node = node.suffix
    if node is not None:
        new_node.suffix = node.children[c]
    return top.children[c] #update top
```

The main entry point of the program iterates all characters of a given string, and call the insert() function repeatedly.

```
def suffix_trie(str):
    t = None
    for c in str:
        t = insert(t, c)
    return root(t)
```

Because insert() function returns the updated top position, the program calls a function to return the root node as the final result. This function is implemented as the following.

```
def root(node):
    while node.suffix is not None:
        node = node.suffix
    return node
```

It will go along with the suffix links until reach the root node.

In order to verify the program, we need convert the suffix Trie to human readable string. This is realized in a recursive way for easy illustration purpose.

```
def to_lines(t):
    if len(t.children)==0:
        return [""]
    res = []
    for c, tr in sorted(t.children.items()):
        lines = to_lines(tr)
        lines[0] = "|--"+c+"-->"+lines[0]
        if len(t.children)>1:
            lines[1:] = map(lambda l: "|      "+l, lines[1:])
        else:
            lines[1:] = map(lambda l: "       "+l, lines[1:])
        if res !=[]:
            res.append("|")
        res += lines
    return res

def to_str(t):
    return "\n".join(to_lines(t))
```

With the to_str() helper function, we can test our program with some simple cases.

```
class SuffixTrieTest:
    def __init__(self):
        print "start suffix trie test"

    def run(self):
        self.test_build()

    def __test_build(self, str):
        print "Suffix Trie (",str,"):\n", to_str(suffix_trie(str)),"\n"

    def test_build(self):
        str="cacao"
        for i in range(len(str)):
            self.__test_build(str[:i+1])
```

Run this test program will output the below result.

```
start suffix trie test
Suffix Trie (c):
|--c-->

Suffix Trie (ca):
|--a-->
|
|--c-->|--a-->

Suffix Trie (cac):
|--a-->|--c-->
|
|--c-->|--a-->|--c-->

Suffix Trie (caca):
```

```
|--a-->|--c-->|--a-->
|
|--c-->|--a-->|--c-->|--a-->

Suffix Trie (cacao):
|--a-->|--c-->|--a-->|--o-->
|      |
|      |--o-->
|
|--c-->|--a-->|--c-->|--a-->|--o-->
|             |
|             |--o-->
|
|--o-->
```

Compare with figure 6.6, we can find that the results are identical.

Suffix Trie on-line construction algorithm in C++

With ISO C++, we define the suffix Trie node as a struct.

```
struct Node{
  typedef std::string::value_type Key;
  typedef std::map<Key, Node*> Children;

  Node(Node* suffix_link=0):suffix(suffix_link){}
  ~Node(){
    for(Children::iterator it=children.begin();
        it!=children.end(); ++it)
      delete it->second;
  }

  Children children;
  Node* suffix;
};
```

The difference between a standard Trie node definition is the suffix link member pointer.

The insert function will updating from the top position of the suffix Trie.

```
Node* insert(Node* top, Node::Key c){
  if(!top)
    top = new Node();
  Node dummy;
  Node *node(top), *prev(&dummy);
  while(node && (node->children.find(c)==node->children.end())){
    node->children[c] = new Node(node);
    prev->suffix = node->children[c];
    node = node->suffix;
  }
  if(node)
    prev->suffix = node->children[c];
  return top->children[c];
}
```

If the top points to null pointer, it means the Trie hasn't been initialized yet. Instead of using sentinel as Ukkonen did in his paper, I explicitly test if the loop can be terminated when it goes back along the suffix link to the root node. A dummy node is used for simplify the logic. At the end of the program, the new top position is returned.

In order to find the root node of a suffix Trie, a helper function is provided as below.

```
Node* root(Node* node){
  for(; node→suffix; node=node→suffix);
  return node;
}
```

The main entry for the suffix Trie on-line construction is defined like the following.

```
Node* suffix_trie(std::string s){
  return root(std::accumulate(s.begin(), s.end(), (Node*)0,
                              std::ptr_fun(insert)));
}
```

This C++ program will generate the same result as the Python program, the output/printing part is skipped here. Please refer to section 6.4.1 for the details about how to convert suffix Trie to string.

6.3.3 Alternative functional algorithm

Because functional approach isn't suitable for on-line updating. I'll provide a declarative style suffix Trie build algorithm in later section together with suffix tree building algorithm.

6.4 Suffix Tree

Suffix Trie is helpful when study the on-line construction algorithm. However, instead of suffix Trie, suffix tree is commonly used in real world. The above suffix Trie on-line construction algorithm is $O(n^2)$, and need a lot of memory space. One trivial solution is to compress the suffix Trie to suffix tree[6], but it is possible to find much better method than it.

In this section, an $O(n)$ on-line construction algorithm for suffix tree is introduced based on Ukkonen's work[1].

6.4.1 On-line construction of suffix tree

Active point and end point

Although the suffix Trie construction algorithm is $O(n^2)$, it shows very important facts about what happens when $SuffixTrie(S_i)$ is updated to $SuffixTrie(S_{i+1})$. Let's review the last 2 trees when we construct for "cacao".

We can find that there are two different types of updating.

1. All leaves are appended with a new node of s_{i+1}-child;
2. Some non-leaf nodes are branched out with a new node of s_{i+1}-child.

The first type of updating is trivial, because for all new coming characters, we need do this trivial work anyway. Ukkonen defined leaf as 'open' node.

The second type of updating is important. We need figure out which internal nodes need to be branched out. We only focus on these nodes and apply our updating.

Let's review the main algorithm for suffix Trie. We start from top position of a Trie, process and go along with suffix link. If a node hasn't s_{i+1}-child, we create a new child, then update the suffix link and go on traverse with suffix link until we arrive at a node which has s_{i+1}-child already or it is root node.

Ukkonen defined the path along suffix link from top to the end as 'boundary path'. All nodes in boundary path are denoted as, $n_1, n_2, ..., n_j, ..., n_k$. These nodes start from leaf node (the first one is the top position), after the j-th node, they are not leaves any longer, we need do branching from this time point until we stop at the k-th node.

Ukkonen defined the first none-leaf node n_j as 'active point' and the last one n_k as 'end point'. Please note that end point can be the root node.

Reference pair

In suffix Trie, we define a node X transfer to node Y by edge labeled with a character c as $Y \leftarrow CHILDREN(X)[c]$, However, If we compress the Trie to Patricia, we can't use this transfer concept anymore.

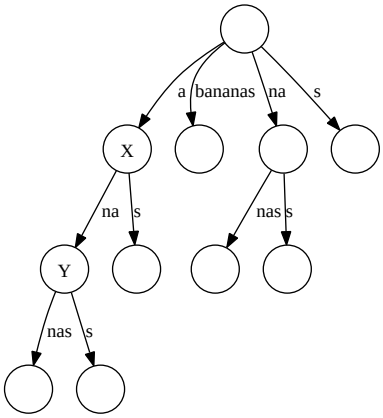


Figure 6.7: Suffix tree of "bananas". X transfer to Y with sub-string "na".

Figure 6.7 shows the suffix tree of English word "bananas". Node X represents for suffix "a", by adding a sub-string "na", node X transfers to node Y, which represents for suffix "ana". Such transfer relationship can be denoted like $Y \leftarrow CHILDREN(X)[w]$, where $w =$ "ana". In other words, we can represent Y with a pair of node and a sub-string, like (X, w). Ukkonen defined such kind of pair as *reference pair*. Not only the explicit node, but also the implicit position in suffix tree can be represented with reference pair. For example, $(X,$ "n") represents to a position which is not an explicit node. By using reference pair, we can represent every position in a suffix Trie for suffix tree.

In order to save spaces, Ukkonen found that given a string S all sub-strings

can be represented as a pair of index (l, r), where l is the left index and r is the right index of character for the sub-string. For instance, if $S =$ "$bananas$", and the index from 1, sub-string "na" can be represented with pair $(3, 4)$. As the result, there will be only one copy of the complete string, and all position in a suffix tree will be refined as $(node, (l, r))$. This is the final form for reference pair.

Let's define the node transfer for suffix tree as the following.

$CHILDREN(X)[s_l] \leftarrow ((l, r), Y) \iff Y \leftarrow (X, (l, r))$

If $s_i = c$, we say that node X has a c-child. Each node can have at most one c-child.

canonical reference pair

It's obvious that the one position in a suffix tree has multiple reference pairs. For example, node Y in Figure 6.7 can be either denoted as $(X, (3, 4))$ or $(root, (2, 4))$. And if we define empty string $\epsilon = (i, i - 1)$, Y can also be represented as (Y, ϵ).

Ukkonen defined the canonical reference pair as the one which has the closest node to the position. So among the reference pairs of $(root, (2, 3))$ and $(X, (3, 3))$, the latter is the canonical reference pair. Specially, in case a position is an explicit node, the canonical reference pair is $(node, \epsilon)$, so (Y, ϵ) is the canonical reference pair of position corresponding to node Y.

It's easy to provide an algorithm to convert a reference pair $(node, (l, r))$ to canonical reference pair $(node', (l', r))$. Note that r won't be changed, so this algorithm can only return $(node', l')$ as the result.

Algorithm 3 Convert reference pair to canonical reference pair

```
1:  function CANONIZE(node, (l, r))
2:      if node = NIL then
3:          if (l, r) = ϵ then
4:              return (NIL, l)
5:          else
6:              return CANONIZE(root, (l + 1, r))
7:      while l ≤ r do
8:          ((l', r'), node') ← CHILDREN(node)[s_l]
9:          if r − l ≥ r' − l' then
10:             l ← l + LENGTH(l', r')
11:             node ← node'
12:         else
13:             break
14:     return (node, l)
```

In case the node parameter is NIL, it means a very special case, typically it is something like the following.

$CANONIZE(SUFFIX - LINK(root), (l, r))$

Because the suffix link of root points to NIL, the result should be $(root, (l + 1, r))$ if (l, r) is not ϵ. Else (NIL, ϵ) is returned to indicate a terminal position.

I'll explain this special case in more detail later.

The algorithm

In 6.4.1, we mentioned, all updating to leaves is trivial, because we only need append the new coming character to the leaf. With reference pair, it means, when we update $SuffixTree(S_i)$ to $SuffixTree(S_{i+1})$, For all reference pairs with form $(node, (l, i))$, they are leaves, they will be change to $(node, (l, i+1))$ next time. Ukkonen defined leaf as $(node, (l, \infty))$, here ∞ means "open to grow". We can omit all leaves until the suffix tree is completely constructed. After that, we can change all ∞ to the length of the string.

So the main algorithm only cares about *positions* from active point to end point. However, how to find the active point and end point?

When we start from the very beginning, there is only a root node, there are no branches nor leaves. The active point should be $(root, \epsilon)$, or $(root, (1, 0))$ (the string index start from 1).

About the end point, it's a position we can finish updating $SuffixTree(S_i)$. According to the algorithm for suffix Trie, we know it should be a *position* which has s_{i+1}-child already. Because a position in suffix Trie may not be an explicit node in suffix tree. If $(node, (l, r))$ is the end point, there are two cases.

1. $(l, r) = \epsilon$, it means node itself an end point, so node has a s_{i+1}-child. Which means $CHILDREN(node)[s_{i+1}] \neq NIL$

2. otherwise, $l \leq r$, end point is an implicit position. It must satisfy $s_{i+1} = s_{l'+|(l,r)|}$, where $CHILDREN(node)[s_l] = ((l', r'), node')$. and $|(l, r)|$ means the length of string (l, r). it is equal to $r - l + 1$. This is illustrate in figure 6.8. We can also say that $(node, (l, r))$ has a s_{i+1}-child implicitly.

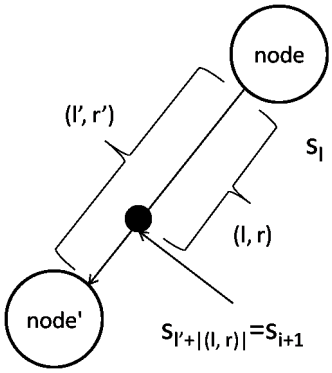


Figure 6.8: Implicit end point

Ukkonen found a very important fact that if the $(node, (l, i))$ is the end point of $SuffixTree(S_i)$, then $(node, (l, i+1))$ is the active point $SuffixTree(S_{i+1})$.

This is because if $(node, (l, i))$ is the end point of $SuffixTree(S_i)$, It must have a s_{i+1}-child (either explicitly or implicitly). If the suffix this end point represents is $s_k s_{k+1} ... s_i$, it is the longest suffix in $SuffxiTree(S_i)$ which satisfies $s_k s_{k+1} ... s_i s_{i+1}$ is a sub-string of S_i. Consider S_{i+1}, $s_k s_{k+1} ... s_i s_{i+1}$ must

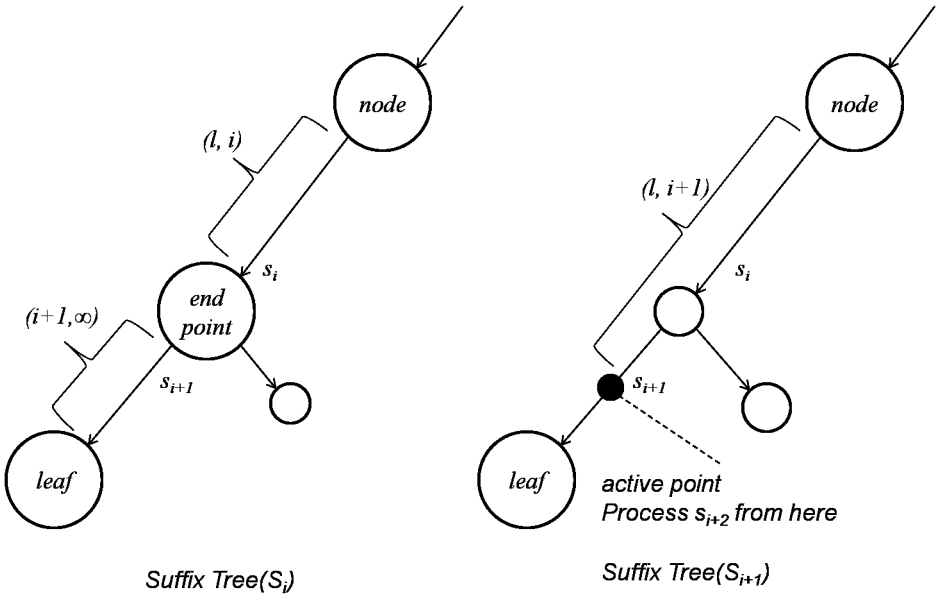


Figure 6.9: End point in $SuffixTree(S_i)$ and active point in $SuffixTree(S_{i+1})$.

occurs at least twice in S_{i+1}, so position $(node, (l, i+1))$ is the active point of $SuffixTree(S_{i+1})$. Figure 6.9 shows about this truth.

At this time point, the algorithm of Ukkonen's on-line construction can be given as the following.

```
1: function UPDATE(node, (l, i))
2:     prev ← CREATE − NEW − NODE()
3:     while TRUE do
4:         (finish, node′) ← END − POINT − BRANCH?(node, (l, i − 1), s_i)
5:         if finish = TRUE then
6:             BREAK
7:         CHILDREN(node′)[s_i] ← ((i, ∞), CREATE − NEW − NODE())
8:         SUFFIX − LINK(prev) ← node′
9:         prev ← node′
10:        (node, l) = CANONIZE(SUFFIX − LINK(node), (l, i − 1))
11:    SUFFIX − LINK(prev) ← node
12:    return (node, l)
```

This algorithm takes a parameter as reference pair $(node, (l, i))$, note that position $(node, (l, i-1)$ is the active point for $SuffixTree(S_{i-1})$. Next we enter a loop, this loop will go along with the suffix link until it found the current position $(node, (l, i-1))$ is the end point. If it is not end point, the function $END - POINT - BRANCH?()$ will return a position, from where the new leaf node will be branch out.

$END - POINT - BRANCH?()$ algorithm is implemented as below.

```
function END-POINT-BRANCH?(node, (l, r), c)
    if (l, r) = ϵ then
```

if $node = NIL$ **then**
return $(TRUE, root)$
else
return $(CHILDREN(node)[c] = NIL?, node)$
else
$((l', r'), node') \leftarrow CHILDREN(node)[s_l]$
$pos \leftarrow l' + |(l, r)|$
if $s_{pos} = c$ **then**
return $(TRUE, node)$
else
$p \leftarrow CREATE - NEW - NODE()$
$CHILDREN(node)[s_{l'}] \leftarrow ((l', pos - 1), p)$
$CHILDREN(p)[s_{pos}] \leftarrow ((pos, r'), node')$
return $(FALSE, p)$

If the position is $(root, \epsilon)$, which means we have gone along suffix links to the root, we return $TRUE$ to indicate the updating can be finished for this round. If the position is in form of $(node, \epsilon)$, it means the reference pair represents an explicit node, we just test if this node has already c-child, where $c = s_i$. and if not, we can just branching out a leaf from this node.

In other case, which means the position $(node, (l, r))$ points to a implicit node. We need find the exact position next to it to see if it is c-child implicitly. If yes, we meet a end point, the updating loop can be finished, else, we make the position an explicit node, and return it for further branching.

With the previous defined $CANONIZE()$ function, we can finalize the Ukkonen's algorithm.

1: **function** SUFFIX-TREE(S)
2: $root \leftarrow CREATE - NEW - NODE()$
3: $node \leftarrow root, l \leftarrow 0$
4: **for** $i \leftarrow 1$ to $LENGTH(S)$ **do**
5: $(node, l) = UPDATE(node, (l, i))$
6: $(node, l) = CANONIZE(node, (l, i))$
7: **return** $root$

Figure 6.10 shows the phases when constructing the suffix tree for string "cacao" with Ukkonen's algorithm.

Note that it needn't setup suffix link for leaf nodes, only branch nodes have been set suffix links.

Implementation of Ukkonen's algorithm in imperative languages

The 2 main features of Ukkonen's algorithm are intense use of suffix link and on-line update. So it will be very suitable to implement in imperative language.

Ukkonen's algorithm in Python

The node definition is as same as the suffix Trie, however, the exact meaning for children field are not same.

```
class Node:
    def __init__(self, suffix=None):
        self.children = {} # 'c':(word, Node), where word = (l, r)
```

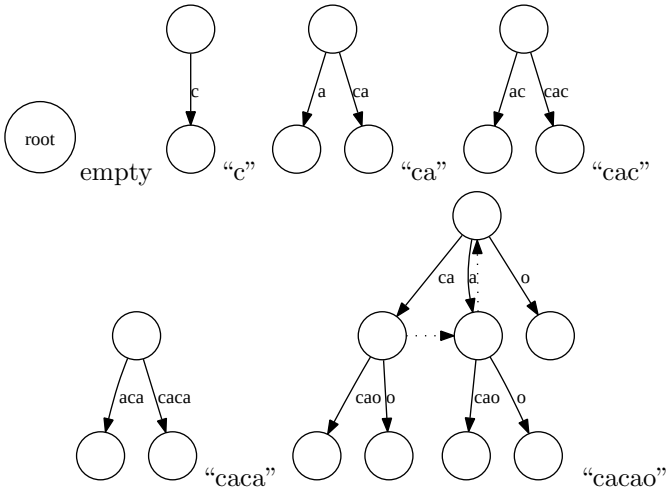


Figure 6.10: Construction of suffix tree for "cacao", the 6 phases are shown, only the last layer of suffix links are shown in dotted arrow.

```
        self.suffix = suffix
```

The children for suffix tree actually represent to the node transition with reference pair. if the transition is $CHILDREN(node)[s_l] \leftarrow ((l, r)node')$, The key type of the children is the character, which is corresponding to s_l, the data; the data type of the children is the reference pair.

Because there is only one copy of the complete string, all sub-strings are represent in $(left, right)$ pairs, and the leaf are open pairs as $(left, \infty)$, so we provide a tree definition in Python as below.

```
class STree:
    def __init__(self, s):
        self.str = s
        self.infinity = len(s)+1000
        self.root = Node()
```

The infinity is defined as the length of the string plus a big number, we'll benefit from python's list[a:b] expression that if the right index exceed to the length of the list, the result is from left to the end of the list.

For convenience, I provide 2 helper functions for later use.

```
def substr(str, str_ref):
    (l, r)=str_ref
    return str[l:r+1]

def length(str_ref):
    (l, r)=str_ref
    return r-l+1
```

The main entry for Ukkonen's algorithm is implemented as the following.

```
def suffix_tree(str):
    t = STree(str)
    node = t.root # init active point is (root, Empty)
```

```
    l = 0
    for i in range(len(str)):
        (node, l) = update(t, node, (l, i))
        (node, l) = canonize(t, node, (l, i))
    return t
```

In the main entry, we initialize the tree and let the node points to the root, at this time point, the active point is $(root, \epsilon)$, which is (root, (0, -1)) in Python. we pass the active point to update() function in a loop from the left most index to the right most index of the string. Inside the loop, update() function returns the end point, and we need convert it to canonical reference pair for the next time update.

the update() function is realized like the following.

```
def update(t, node, str_ref):
    (l, i) = str_ref
    c = t.str[i] # current char
    prev = Node() # dummy init
    while True:
        (finish, p) = branch(t, node, (l, i-1), c)
        if finish:
            break
        p.children[c]=((i, t.infinity), Node())
        prev.suffix = p
        prev = p
        # go up along suffix link
        (node, l) = canonize(t, node.suffix, (l, i-1))
    prev.suffix = node
    return (node, l)
```

Different with Ukkonen's original program, I didn't use sentinel node. The reference passed in is $(node, (l, i)$, the active point is $(node, (l, i-1))$ actually, we passed the active point to branch() function. If it is end point, branch() function will return true as the first element in the result. we then terminate the loop immediately. Otherwise, branch() function will return the node which need to branch out a new leaf as the second element in the result. The program then create the new leaf, set it as open pair, and then go up along with suffix link. The prev variable first point to a dummy node, this can simplify the logic, and it used to record the position along the boundary path. by the end of the loop, we'll finish the last updating of the suffix link and return the end point. Since the end point is always in form of (node, (l, i-1)), only (node, l) is returned.

Function branch() is used to test if a position is the end point and turn the implicit node to explicit node if necessary.

```
def branch(t, node, str_ref, c):
    (l, r) = str_ref
    if length(str_ref)≤0: # (node, empty)
        if node is None: #_|_
            return (True, t.root)
        else:
            return ((c in node.children), node)
    else:
        ((l1, r1), node1) = node.children[t.str[l]]
        pos = l1+length(str_ref)
```

```
        if t.str[pos]==c:
            return (True, node)
        else:              # node-->branch_node-->node1
            branch_node = Node()
            node.children[t.str[l1]]=((l1, pos-1), branch_node)
            branch_node.children[t.str[pos]] = ((pos, r1), node1)
            return (False, branch_node)
```

Because I don't use sentinel node, the special case is handled in the first if-clause.

The canonize() function helps to convert a reference pair to canonical reference pair.

```
def canonize(t, node, str_ref):
    (l, r) = str_ref
    if node is None:
        if length(str_ref)<=0:
            return (None, l)
        else:
            return canonize(t, t.root, (l+1, r))
    while l<=r: # str_ref is not empty
        ((l1, r1), child) = node.children[t.str[l]] # node--(l', r')-->child
        if r-l >= r1-l1: #node--(l',r')-->child-->...
            l += r1-l1+1 # remove |(l',r')| chars from (l, r)
            node = child
        else:
            break
    return (node, l)
```

Before testing the suffix tree construction algorithm, some helper functions to convert the suffix tree to human readable string are given.

```
def to_lines(t, node):
    if len(node.children)==0:
        return [""]
    res = []
    for c, (str_ref, tr) in sorted(node.children.items()):
        lines = to_lines(t, tr)
        edge_str = substr(t.str, str_ref)
        lines[0] = "|--"+edge_str+"-->"+lines[0]
        if len(node.children)>1:
            lines[1:] = map(lambda l: "|"+" "*(len(edge_str)+5)+l, lines[1:])
        else:
            lines[1:] = map(lambda l: " "+" "*(len(edge_str)+6)+l, lines[1:])
        if res !=[]:
            res.append("|")
        res += lines
    return res

def to_str(t):
    return "\n".join(to_lines(t, t.root))
```

They are quite similar to the helper functions for suffix Trie print. The different part is mainly cause by the reference pair of string.

In order to verify the implementation, some very simple test cases are feed to the algorithm as below.

```
class SuffixTreeTest:
    def __init__(self):
        print "start␣suffix␣tree␣test"

    def run(self):
        strs = ["cacao", "mississippi", "banana$"] #$ special terminator
        for s in strs:
            self.test_build(s)

    def test_build(self, str):
        for i in range(len(str)):
            self.__test_build(str[:i+1])


    def __test_build(self, str):
        print "Suffix␣Tree␣("+str+"):λn", to_str(suffix_tree(str)),"λn"
```

Here is a result snippet which shows the construction phases for string "cacao"

```
Suffix Tree (c):
|--c-->

Suffix Tree (ca):
|--a-->
|
|--ca-->

Suffix Tree (cac):
|--ac-->
|
|--cac-->

Suffix Tree (caca):
|--aca-->
|
|--caca-->

Suffix Tree (cacao):
|--a-->|--cao-->
|      |
|      |--o-->
|
|--ca-->|--cao-->
|       |
|       |--o-->
|
|--o-->
```

The result is identical to the one which is shown in figure 6.10.

Ukkonen's algorithm in C++

Ukkonen's algorithm has much use of pairs, including reference pair and sub-string index pair. Although STL provided std::pair tool, it is lack of variable binding ability, for example (x, y)=apair like assignment isn't legal C++ code. boost::tuple provides a handy tool, tie(). I'll give a mimic tool like boost::tie so that we can bind two variables to a pair.

```
template<typename T1, typename T2>
struct Bind{
  Bind(T1& r1, T2& r2):x1(r1), x2(r2){}
  Bind(const Bind& r):x1(r.x1), x2(r.x2){}

  // Support implicit type conversion
  template<typename U1, typename U2>
  Bind& operator=(const std::pair<U1, U2>& p){
    x1 = p.first;
    x2 = p.second;
    return *this;
  }
  T1& x1;
  T2& x2;
};

template<typename T1, typename T2>
Bind<T1, T2> tie(T1& r1, T2& r2){
  return Bind<T1, T2>(r1, r2);
}
```

With this tool, we can tie variables like the following.

```
int l, r;
tie(l, r) = str_pair;
```

We define sub-string index pair and reference pair like below. First is string index reference pair.

```
struct StrRef: public std::pair<int, int>{
  typedef std::pair<int, int> Pair;
  static std::string str;
  StrRef():Pair(){}
  StrRef(int l, int r):Pair(l, r){}
  StrRef(const Pair& ref):Pair(ref){}

  std::string substr(){
    int l, r;
    tie(l, r) = *this;
    return str.substr(l, len());
  }

  int len(){
    int l, r;
    tie(l, r) = *this;
    return r-l+1;
  }
};
```

```
std::string StrRef::str="";
```

Because there is only one copy of the complete string, a static variable is used to store it. substr() function is used to convert a pair of left, right index into the sub-string. Function len() is used to calculate the length of a sub-string.

Ukkonen's reference pair is defined in the same way.

```
struct Node;

struct RefPair: public std::pair<Node*, StrRef>{
  typedef std::pair<Node*, StrRef> Pair;
  RefPair():Pair(){}
  RefPair(Node* n, StrRef s):Pair(n, s){}
  RefPair(const Pair& p):Pair(p){}
  Node* node(){ return first; }
  StrRef str(){ return second; }
};
```

With these definition, the node type of suffix tree can be defined.

```
struct Node{
  typedef std::string::value_type Key;
  typedef std::map<Key, RefPair> Children;

  Node():suffix(0){}
  ~Node(){
    for(Children::iterator it=children.begin();
        it!=children.end(); ++it)
      delete it->second.node();
  }

  Children children;
  Node* suffix;
};
```

The children of a node is defined as a map with reference pairs stored. In order for easy memory management, a recursive approach is used.

The final suffix tree is defined with a string and a root node.

```
struct STree{
  STree(std::string s):str(s),
                       infinity(s.length()+1000),
                       root(new Node)
  { StrRef::str = str; }

  ~STree() { delete root; }
  std::string str;
  int infinity;
  Node* root;
};
```

the infinity is defined as the length of the string plus a big number. Infinity will be used for leaf node as the 'open to append' meaning.

Next is the main entry of Ukkonen's algorithm.

```
STree* suffix_tree(std::string s){
```

```
  STree* t=new STree(s);
  Node* node = t→root; // init active point as (root, empty)
  for(unsigned int i=0, l=0; i<s.length(); ++i){
    tie(node, l) = update(t, node, StrRef(l, i));
    tie(node, l) = canonize(t, node, StrRef(l, i));
  }
  return t;
}
```

The program start from the initialized active point, and repeatedly call update(), the returned end point will be canonized and used for the next active point.

Function update() is implemented as below.

```
std::pair<Node*, int> update(STree* t, Node* node, StrRef str){
  int l, i;
  tie(l, i)=str;
  Node::Key c(t→str[i]); //current char
  Node dummy, *p;
  Node* prev(&dummy);
  while((p=branch(t, node, StrRef(l, i-1), c))!=0){
    p→children[c]=RefPair(new Node(), StrRef(i, t→infinity));
    prev→suffix = p;
    prev = p;
    // go up along suffix link
    tie(node, l) = canonize(t, node→suffix, StrRef(l, i-1));
  }
  prev→suffix = node;
  return std::make_pair(node, l);
}
```

In this function, pair (node, (l, i-1)) is the real active point position. It is fed to branch() function. If the position is end point, branch will return NULL pointer, so the while loop terminates. else a node for branching out a new leaf is returned. Then the program will go up along with suffix links and update the previous suffix link accordingly. The end point will be returned as the result of this function.

Function branch() is implemented as the following.

```
Node* branch(STree* t, Node* node, StrRef str, Node::Key c){
  int l, r;
  tie(l, r) = str;
  if(str.len()≤0){ // (node, empty)
    if(node && node→children.find(c)==node→children.end())
      return node;
    else
      return 0; // either node is empty (_|_), or is EP
  }
  else{
    RefPair rp = node→children[t→str[l]];
    int l1, r1;
    tie(l1, r1) = rp.str();
    int pos = l1+str.len();
    if(t→str[pos]==c)
      return 0;
```

```
    else{ // node-->branch_node-->node1
      Node* branch_node = new Node();
      node->children[t->str[l1]]=RefPair(branch_node, StrRef(l1, pos-1));
      branch_node->children[t->str[pos]] = RefPair(rp.node(), StrRef(pos, r1));
      return branch_node;
    }
  }
}
```

If the position is (NULL, empty), it means the program arrive at the root position, NULL pointer is returned to indicate the updating can be terminated. If the position is in form of $(node, \epsilon)$, it then check if the node has already a s_i-child. In other case, it means the position point to a implicit node, we need extra process to test if it is end point. If not, a splitting happens to convert the implicit node to explicit one.

The function to canonize a reference pair is given below.

```
std::pair<Node*, int> canonize(STree* t, Node* node, StrRef str){
  int l, r;
  tie(l, r)=str;
  if(!node)
    if(str.len()<=0)
      return std::make_pair(node, l);
    else
      return canonize(t, t->root, StrRef(l+1, r));
  while(l<=r){ //str isn't empty
    RefPair rp = node->children[t->str[l]];
    int l1,r1;
    tie(l1,r1) = rp.str();
    if(r-l >= r1-l1){
      l += rp.str().len(); // remove len() from (l,r)
      node = rp.node();
    }
    else
      break;
  }
  return std::make_pair(node,l);
}
```

In order to test the program, some helper functions are provided to represent the suffix tree as string. Among them, some are very common tools.

```
// map (x+) coll in Haskell
// boost lambda: transform(first, last, first, x+_1)
template<class Iter, class T>
void map_add(Iter first, Iter last, T x){
  std::transform(first, last, first,
                 std::bind1st(std::plus<T>(), x));
}

// x ++ y in Haskell
template<class Coll>
void concat(Coll& x, Coll& y){
  std::copy(y.begin(), y.end(),
            std::insert_iterator<Coll>(x, x.end()));
}
```

map_add() will add a value to every element in a collection. concat can concatenate tow collections together.

the suffix tree to string function is finally provided like this.

```
std::list<std::string> to_lines(Node* node){
  typedef std::list<std::string> Result;
  Result res;
  if(node->children.empty()){
    res.push_back("");
    return res;
  }
  for(Node::Children::iterator it = node->children.begin();
      it!=node->children.end(); ++it){
    RefPair rp = it->second;
    Result lns = to_lines(rp.node());
    std::string edge = rp.str().substr();
    *lns.begin() = "|--" + edge + "-->" + (*lns.begin());
    map_add(++lns.begin(), lns.end(),
            std::string("|")+std::string(edge.length()+5, ' '));
    if(!res.empty())
      res.push_back("|");
    concat(res, lns);
  }
  return res;
}

std::string to_str(STree* t){
  std::list<std::string> ls = to_lines(t->root);
  std::ostringstream s;
  std::copy(ls.begin(), ls.end(),
            std::ostream_iterator<std::string>(s, "\n"));
  return s.str();
}
```

After that, the program can be verified by some simple test cases.

```
class SuffixTreeTest{
public:
  SuffixTreeTest(){
    std::cout<<"Start suffix tree test\n";
  }
  void run(){
    test_build("cacao");
    test_build("mississippi");
    test_build("banana$"); //$ as special terminator
  }
private:
  void test_build(std::string str){
    for(unsigned int i=0; i<str.length(); ++i)
      test_build_step(str.substr(0, i+1));
  }

  void test_build_step(std::string str){
    STree* t = suffix_tree(str);
    std::cout<<"Suffix Tree ("<<str<<"):\n"
```

```
            <<to_str(t)<<"\n";
    delete t;
  }
};
```

Below are snippet of suffix tree construction phases.

```
Suffix Tree (b):
|--b-->

Suffix Tree (ba):
|--a-->
|
|--ba-->

Suffix Tree (ban):
|--an-->
|
|--ban-->
|
|--n-->

Suffix Tree (bana):
|--ana-->
|
|--bana-->
|
|--na-->

Suffix Tree (banan):
|--anan-->
|
|--banan-->
|
|--nan-->

Suffix Tree (banana):
|--anana-->
|
|--banana-->
|
|--nana-->

Suffix Tree (banana$):
|--$-->
|
|--a-->|--$-->
|      |
|      |--na-->|--$-->
|      |       |
|      |       |--na$-->
```

```
|
|--banana$-->
|
|--na-->|--$-->
|       |
|       |--na$-->
```

Functional algorithm for suffix tree construction

Ukkonen's algorithm is in a manner of on-ling updating and suffix link plays very important role. Such properties can't be realized in a functional approach. Giegerich and Kurtz found Ukkonen's algorithm can be transformed to McCreight's algorithm[7]. These two and the algorithm found by Weiner are all $O(n)$-algorithms. They also conjecture (although it isn't proved) that any sequential suffix tree construction not base on the important concepts, such as suffix links, active suffixes, etc., will fail to meet $O(n)$-criterion.

There is implemented in PLT/Scheme[10] based on Ukkonen's algorithm, However it updates suffix links during the processing, so it is not pure functional approach.

A lazy suffix tree construction method is discussed in [8]. And this method is contribute to Haskell Hackage by Bryan O'Sullivan. [9].

This method benefit from the lazy evaluation property of Haskell programming languages, so that the tree won't be constructed until it is traversed. However, I think it is still a kind of brute-force method. In other functional programming languages such as ML. It can't be $O(n)$ algorithm.

I will provide a pure brute-force implementation which is similar but not 100% same in this post.

Brute-force suffix tree construction in Haskell

For brute-force implementation, we needn't suffix link at all. The definition of suffix node is plain straightforward.

```
data Tr = Lf | Br [(String, Tr)] deriving (Eq, Show)
type EdgeFunc = [String]→(String, [String])
```

The edge function plays interesting role. it takes a list of strings, and extract a prefix of these strings, the prefix may not be the longest prefix, and empty string is also possible. Whether extract the longest prefix or just return them trivially can be defined by different edge functions.

For easy implementation, we limit the character set as below.

```
alpha = ['a'..'z']++['A'..'Z']
```

This is only for illustration purpose, only the English lower case and upper case letters are included. We can of course includes other characters if necessary.

The core algorithm is given in list comprehension style.

```
lazyTree::EdgeFunc → [String] → Tr
lazyTree edge = build where
    build [[]] = Lf
    build ss = Br [(a:prefix, build ss') |
                        a←alpha,
                        xs@(x:_) ←[[cs | c:cs←ss, c==a]],
```

```
(prefix, ss')←[edge xs]]
```

lazyTree function takes a list of string, it will generate a radix tree (for example a Trie or a Patricia) from these string.

It will categorize all strings with the first letter in several groups, and remove the first letter for each elements in every group. For example, for the string list ["acac", "cac", "ac", "c"] the categorized group will be [('a', ["cac", "c"]), ('c', ["ac", ""])]. For easy understanding, I left the first letter, and write the groups as tuple. then all strings with same first letter (removed) will be fed to edge function.

Different edge function produce different radix trees. The most trivial one will build a Trie.

```
edgeTrie::EdgeFunc
edgeTrie ss = ("", ss)
```

If the edge function extract the longest common prefix, then it will build a Patricia.

```
-- ex:
--   edgeTree ["an", "another", "and"] = ("an", ["", "other", "d"])
--   edgeTree ["bool", "foo", "bar"] = ("", ["bool", "foo", "bar"])
--
-- some helper comments
--   let awss@((a:w):ss) = ["an", "another", "and"]
--       (a:w) = "an",  ss = ["another", "and"]
--       a=='a', w=="n"
--       rests awss = w:[u| _:u←ss] = ["n", "nother", "nd"]
--
edgeTree::EdgeFunc
edgeTree [s] = (s, [[]])
edgeTree awss@((a:w):ss) | null [c|c:_←ss, a/=c] = (a:prefix, ss')
                         | otherwise             = ("", awss)
                         where (prefix, ss') = edgeTree (w:[u| _:u←ss])
edgeTree ss = ("", ss) -- (a:w):ss can't be match ≤⇒ head ss == ""
```

We can build suffix Trie and suffix tree with the above two functions.

```
suffixTrie::String→Tr
suffixTrie = lazyTree edgeTrie ∘ tails -- or init ∘ tails

suffixTree::String→Tr
suffixTree = lazyTree edgeTree ∘ tails
```

Below snippet is the result of constructing suffix Trie/tree for string "mississippi".

```
SuffixTree("mississippi")=Br [("i",Br [("ppi",Lf),("ssi",Br [("ppi",Lf),
("ssippi",Lf)])]),("mississippi",Lf),("p",Br [("i",Lf),("pi",Lf)]),("s",
Br [("i",Br [("ppi",Lf),("ssippi",Lf)]),("si",Br [("ppi",Lf),("ssippi",
Lf)])])]
SuffixTrie("mississippi")=Br [("i",Br [("p",Br [("p",Br [("i",Lf)])]),
("s",Br [("s",Br [("i",Br [("p",Br [("p",Br [("i",Lf)])])]),("s",Br [("s",
Br [("i",Br [("p",Br [("p",Br [("i",Lf)])])])])])])])]),("m",Br [("i",
Br [("s",Br [("s",Br [("i",Br [("s",Br [("s",Br [("i",Br [("p",Br [("p",
Br [("i",Lf)])])])])])])])])]),("p",Br [("i",Lf),("p",Br [("i",Lf)])]),
```

```
("s",Br [("i",Br [("p",Br [("p",Br [("i",Lf)])]),("s",Br [("s",Br [("i",
Br [("p",Br [("p",Br [("i",Lf)])])])])]),("s",Br [("i",Br [("p",Br
[("p",Br [("i",Lf)])]),("s",Br [("s",Br [("i",Br [("p",Br [("p",Br
[("i",Lf)])])])])])])])]
```

Function lazyTree is common for all radix trees, the normal Patricia and Trie can also be constructed with it.

```
trie::[String]→Tr
trie = lazyTree edgeTrie

patricia::[String]→Tr
patricia = lazyTree edgeTree
```

Let's test it with some simple cases.

```
trie ["zoo", "bool", "boy", "another", "an", "a"]
patricia ["zoo", "bool", "boy", "another", "an", "a"]
```

The results are as below.

```
Br [("a",Br [("n",Br [("o",Br [("t",Br [("h",Br [("e",
Br [("r",Lf)])])])])])]),("b",Br [("o",Br [("o",Br [("l",Lf)]),
("y",Lf)])]),("z",Br [("o",Br [("o",Lf)])])]

Br [("another",Lf),("bo",Br [("ol",Lf),("y",Lf)]),("zoo",Lf)]
```

This is reason why I think the method is brute-force.

Brute-force suffix tree construction in Scheme/Lisp

.

The Functional implementation in Haskell utilizes list comprehension, which is a handy syntax tool. In Scheme/Lisp, we use functions instead.

In MIT scheme, there are special functions to manipulate strings, which is a bit different from list. Below are helper functions to simulate car and cdr function for string.

```
(define (string-car s)
  (if (string=? s "")
      ""
      (string-head s 1)))

(define (string-cdr s)
  (if (string=? s "")
      ""
      (string-tail s 1)))
```

The edge functions will extract common prefix from a list of strings. For Trie, only the first common character will be extracted.

```
;; (edge-trie '("an" "another" "and"))
;;  ⟹ ("a" "n" "nother" "nd")
(define (edge-trie ss)
  (cons (string-car (car ss)) (map string-cdr ss)))
```

While for suffix tree, we need extract the longest common prefix.

```
;; (edge-tree '("an" "another" "and"))
;;   ⟹ ("an" "" "other" "d")
(define (edge-tree ss)
  (cond ((= 1 (length ss)) (cons (car ss) '()))
        ((prefix? ss)
         (let* ((res (edge-tree (map string-cdr ss)))
                (prefix (car res))
                (ss1 (cdr res)))
           (cons (string-append (string-car (car ss)) prefix) ss1)))
        (else (cons "" ss))))

;; test if a list of strings has common prefix
;; (prefix '("an" "another" "and")) ⟹ true
;; (prefix '("" "other" "d")) ⟹ false
(define (prefix? ss)
  (if (null? ss)
      '()
      (let ((c (string-car (car ss))))
        (null? (filter (lambda (s) (not (string=? c (string-car s))))
                       (cdr ss))))))
```

For some old version of MIT scheme, there isn't definition for partition function, so I defined one like below.

```
;; overwrite the partition if not support SRFI 1
;; (partition (> 5) '(1 6 2 7 3 9 0))
;;   ⟹((6 7 9) 1 2 3 0)
(define (partition pred lst)
  (if (null? lst)
      (cons '() '())
      (let ((res (partition pred (cdr lst))))
        (if (pred (car lst))
            (cons (cons (car lst) (car res)) (cdr res))
            (cons (car res) (cons (car lst) (cdr res)))))))
```

Function groups can group a list of strings based on the first letter of each string.

```
;; group a list of strings based on first char
;; ss shouldn't contains "" string, so filter should be done first.
;; (groups '("an" "another" "bool" "and" "bar" "c"))
;;  ⟹ (("an" "another" "and") ("bool" "bar") ("c"))
(define (groups ss)
  (if (null? ss)
      '()
      (let* ((c (string-car (car ss)))
             (res (partition (lambda (x) (string=? c (string-car x))) (cdr ss))))
        (append (list (cons (car ss) (car res)))
                (groups (cdr res))))))
```

Function remove-empty can remove the empty string from the string list.

```
(define (remove-empty ss)
  (filter (lambda (s) (not (string=? "" s))) ss))
```

With all the above tools, the core brute-force algorithm can be implemented like the following.

```
(define (make-tree edge ss)
  (define (bld-group grp)
    (let* ((res (edge grp))
           (prefix (car res))
           (ss1 (cdr res)))
      (cons prefix (make-tree edge ss1))))
  (let ((ss1 (remove-empty ss)))
    (if (null? ss1) '()
        (map bld-group (groups ss1)))))
```

The final suffix tree and suffix Trie construction algorithms can be given.

```
(define (suffix-tree s)
  (make-tree edge-tree (tails s)))

(define (suffix-trie s)
  (make-tree edge-trie (tails s)))
```

Below snippet are quick verification of this program.

```
(suffix-trie "cacao")
;Value 66: (("c" ("a" ("c" ("a" ("o"))) ("o"))) ("a" ("c" ("a" ("o"))) ("o")) ("o"))
(suffix-tree "cacao")
;Value 67: (("ca" ("cao") ("o")) ("a" ("cao") ("o")) ("o"))
```

6.5 Suffix tree applications

Suffix tree can help to solve a lot of string/DNA manipulation problems particularly fast. For typical problems will be list in this section.

6.5.1 String/Pattern searching

There a plenty of string searching problems, among them includes the famous KMP algorithm. Suffix tree can perform same level as KMP[11], that string searching in $O(m)$ complexity, where m is the length of the sub-string.However, O(n) time is required to build the suffix tree in advance[12].

Not only sub-string searching, but also pattern matching, including regular expression matching can be solved with suffix tree. Ukkonen summarize this kind of problems as sub-string motifs, and he gave the result that *For a string S, $SuffixTree(S)$ gives complete occurrence counts of all sub-string motifs of S in $O(n)$ time, although S may have $O(n^2)$ sub-strings.*

Note the facts of a $SuffixTree(S)$ that all internal nodes is corresponding to a repeating sub-string of S and the number of leaves of the sub-tree of a node for string P is the number of occurrence of P in S.[13]

Algorithm of finding the number of sub-string occurrence

The algorithm is almost as same as the Patricia looking up algorithm, please refer to [5] for detail, the only difference is that the number of the children is returned when a node matches the pattern.

Find number of sub-string occurrence in Python

In Ukkonen's algorithm, there is only one copy of string, and all edges are represent with index pairs. There are some changes because of this reason.

```
def lookup_pattern(t, node, s):
    f = (lambda x: 1 if x==0 else x)
    while True:
        match = False
        for _, (str_ref, tr) in node.children.items():
            edge = t.substr(str_ref)
            if string.find(edge, s)==0: #s `isPrefixOf` edge
                return f(len(tr.children))
            elif string.find(s, edge)==0: #edge `isPrefixOf` s
                match = True
                node = tr
                s = s[len(edge):]
                break
        if not match:
            return 0
    return 0 # not found
```

In case a branch node matches the pattern, it means there is at least one occurrence even if the number of children is zero. That's why a local lambda function is defined.

I added a member function in STree to convert a string index pair to string as below.

```
class STree:
    #...
    def substr(self, sref):
        return substr(self.str, sref)
```

In lookup_pattern() function, it takes a suffix tree which is built from the string. A node is passed as the position to be looked up, it is root node when starting. Parameter s is the string to be searched.

The algorithm iterate all children of the node, it convert the string index reference pair to edge sub-string, and check if s is prefix of the the edge string, if matches, then the program can be terminated, the number of the branches of this node will be returned as the number of the occurrence of this sub-string. Note that no branch means there is only 1 occurrence. In case the edge is prefix of s, we then updated the node and string to be searched and go on the searching.

Because construction of the suffix tree is expensive, so we only do it when necessary. We can do a lazy initialization as below.

```
TERM1 = '$' # $: special terminator

class STreeUtil:
    def __init__(self):
        self.tree = None

    def find_pattern(self, str, pattern):
        if self.tree is None or self.tree.str!=str+TERM1:
            self.tree = stree.suffix_tree(str+TERM1)
        return lookup_pattern(self.tree, self.tree.root, pattern)
```

We always append special terminator to the string, so that there won't be any suffix becomes the prefix of the other[2].

Some simple test cases are given to verify the program.

```
class StrSTreeTest:
    def run(self):
        self.test_find_pattern()

    def test_find_pattern(self):
        util=STreeUtil()
        self.__test_pattern__(util, "banana", "ana")
        self.__test_pattern__(util, "banana", "an")
        self.__test_pattern__(util, "banana", "anan")
        self.__test_pattern__(util, "banana", "nana")
        self.__test_pattern__(util, "banana", "ananan")

    def __test_pattern__(self, u, s, p):
        print "find␣pattern", p, "in", s, ":", u.find_pattern(s, p)
```

And the output is like the following.

```
find pattern ana in banana : 2
find pattern an in banana : 2
find pattern anan in banana : 1
find pattern nana in banana : 1
find pattern ananan in banana : 0
```

Find the number of sub-string occurrence in C++

In C++, do-while is used as the repeat-until structure, the program is almost as same as the standard Patricia looking up function.

```
int lookup_pattern(const STree* t, std::string s){
  Node* node = t->root;
  bool match(false);
  do{
    match=false;
    for(Node::Children::iterator it = node->children.begin();
        it!=node->children.end(); ++it){
      RefPair rp = it->second;
      if(rp.str().substr().find(s)==0){
        int res = rp.node()->children.size();
        return res == 0? 1 : res;
      }
      else if(s.find(rp.str().substr())==0){
        match = true;
        node = rp.node();
        s = s.substr(rp.str().substr().length());
        break;
      }
    }
  }while(match);
  return 0;
}
```

An utility class is defined and it support lazy initialization to save the cost of construction of suffix tree.

```
class STreeUtil{
public:
  STreeUtil():t(0){}
  ~STreeUtil(){ delete t; }

  int find_pattern(std::string s, std::string pattern){
    lazy(s);
    return lookup_pattern(t, pattern);
  }
private:
  void lazy(std::string s){
    if((!t) || t→str != s+TERM1){
      delete t;
      t = suffix_tree(s+TERM1);
    }
  }
  STree* t;
};
```

The same test cases can be feed to this C++ program.

```
class StrSTreeTest{
public:
  void test_find_pattern(){
    __test_pattern("banana", "ana");
    __test_pattern("banana", "an");
    __test_pattern("banana", "anan");
    __test_pattern("banana", "nana");
    __test_pattern("banana", "ananan");
  }
pivate:
  void __test_pattern(std::string s, std::string ptn){
    std::cout<<"find␣pattern␣"<<ptn<<"␣in␣"<<s<<":␣"
             <<util.find_pattern(s, ptn)<<"\n";
  }

  STreeUtil util;
};
```

And the same result will be obtained like the Python program.

Find the number of sub-string occurrence in Haskell

The Haskell program is just turn the looking up into recursive way.

```
lookupPattern :: Tr → String → Int
lookupPattern (Br lst) ptn = find lst where
    find [] = 0
    find ((s, t):xs)
         | ptn `isPrefixOf` s = numberOfBranch t
         | s `isPrefixOf` ptn = lookupPattern t (drop (length s) ptn)
         | otherwise = find xs
    numberOfBranch (Br ys) = length ys
```

```
    numberOfBranch _ = 1

findPattern :: String → String → Int
findPattern s ptn = lookupPattern (suffixTree $ s++"$") ptn
```

To verify it, the test cases are fed to the program as the following

```
testPattern = ["find pattern "++p++" in banana: "++
               (show $ findPattern "banana" p)
                   | p← ["ana", "an", "anan", "nana", "anana"]]
```

Launching GHCi, evaluate the instruction can output the same result as the above programs.

```
putStrLn $ unlines testPattern
```

Find the number of sub-string occurrence in Scheme/Lisp

Because the underground data structure of suffix tree is list in Scheme/Lisp program, we needn't define a inner find function as in Haskell program.

```
(define (lookup-pattern t ptn)
  (define (number-of-branches node)
    (if (null? node) 1 (length node)))
  (if (null? t) 0
      (let ((s (edge (car t)))
            (tr (children (car t))))
        (cond ((string-prefix? ptn s)(number-of-branches tr))
              ((string-prefix? s ptn)
               (lookup-pattern tr (string-tail ptn (string-length s))))
              (else lookup-pattern (cdr t) ptn)))))
```

The test cases are fed to this program via a list.

```
(define (test-pattern)
  (define (test-ptn t s)
    (cons (string-append "find␣pattern␣" s "␣in␣banana" )
          (lookup-pattern t s)))
  (let ((t (suffix-tree "banana")))
    (map (lambda (x) (test-ptn t x)) '("ana" "an" "anan" "nana" "anana"))))
```

Evaluate this test function can generate a result list as the following.

```
(test-pattern)
;Value 16: (("find pattern ana in banana"∘ "ana") ("find pattern an in banana"∘ "an") ("find p
```

Complete pattern search

For search pattern like "a**n" with suffix tree, please refer to [13] and [14].

6.5.2 Find the longest repeated sub-string

If we go one step ahead from 6.5.1, below result can be found.

After adding a special terminator character to string S, The longest repeated sub-string can be found by searching the deepest branches in suffix tree.

Consider the example suffix tree shown in figure 6.11

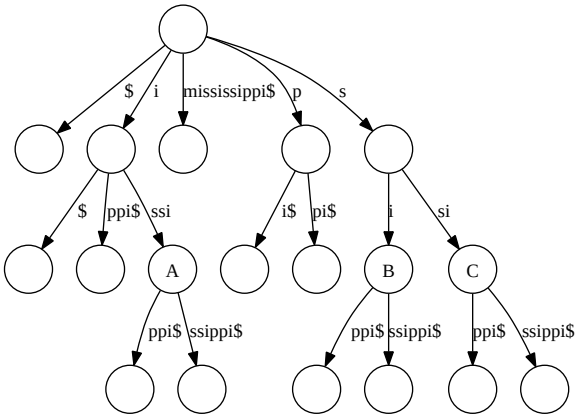


Figure 6.11: The suffix tree for 'mississippi$'

There are 3 branch nodes, A, B, and C which depth is 3. However, A represents the longest repeated sub-string "issi". B and C represent for "si", "ssi", they are all shorter than A.

This example tells us that the "depth" of the branch node should be measured by the number of characters traversed from the root.

Find the longest repeated sub-string in imperative approach

According to the above analysis, to find the longest repeated sub-string can be turned into a BFS (Bread First Search) in a suffix tree.

```
1: function LONGEST-REPEATED-SUBSTRING(T)
2:     Q ← (NIL, ROOT(T))
3:     R ← NIL
4:     while Q is not empty do
5:         (s, node) ← POP(Q)
6:         for each ((l, r), node') in CHILDREN(node) do
7:             if node' is not leaf then
8:                 s' ← CONCATENATE(s, (l, r))
9:                 PUSH(Q, (s', node'))
10:                UPDATE(R, s')
11:    return R
```

where algorithm $UPDATE()$ will compare the longest repeated sub-string candidates. If two candidates have the same length, one simple solution is just take one as the final result, the other solution is to maintain a list contains all candidates with same length.

```
1: function UPDATE(l, x)
2:     if l = NIL or LENGTH(l[1]) < LENGTH(x) then
3:         return l ← [x]
4:     else if LENGTH(l[1]) = LENGTH(x) then
5:         return APPEND(l, x)
```

Note that the index of a list starts from 1 in this algorithm. This algorithm will first initialize a queue with a pair of an empty string and the root node. Then it will repeatedly pop from the queue, examine the candidate node until the queue is empty.

For each node, the algorithm will expand all children, if it is a branch node (which is not a leaf), the node will be pushed back to the queue for future examine. And the sub-string represented by this node will be compared to see if it is a candidate of the longest repeated sub-string.

Find the longest repeated sub-string in Python

The above algorithm can be translated into Python program as the following.

```
def lrs(t):
    queue = [("", t.root)]
    res = []
    while len(queue)>0:
        (s, node) = queue.pop(0)
        for _, (str_ref, tr) in node.children.items():
            if len(tr.children)>0:
                s1 = s+t.substr(str_ref)
                queue.append((s1, tr))
                res = update_max(res, s1)
    return res

def update_max(lst, x):
    if lst ==[] or len(lst[0]) < len(x):
        return [x]
    elif len(lst[0]) == len(x):
        return lst + [x]
    else:
        return lst
```

In order to verify this program, some simple test cases are fed.

```
class StrSTreeTest:
    #...

    def run(self):
        #...
        self.test_lrs()

    def test_lrs(self):
        self.__test_lrs__("mississippi")
        self.__test_lrs__("banana")
        self.__test_lrs__("cacao")
        self.__test_lrs__("foofooxbarbar")

    def __test_lrs__(self, s):
        print "longest␣repeated␣substrings␣of", s, "=", self.util.find_lrs(s)
```

By running the test case, the result like below can be obtained.

```
longest repeated substrings of mississippi = ['issi']
longest repeated substrings of banana = ['ana']
```

```
longest repeated substrings of cacao = ['ca']
longest repeated substrings of foofooxbarbar = ['bar', 'foo']
```

Find the longest repeated sub-string in C++

With C++, we can utilize the STL queue library in the implementation of BFS(Bread First Search).

```
typedef std::list<std::string> Strings;

Strings lrs(const STree* t){
  std::queue<std::pair<std::string, Node*> > q;
  Strings res;
  q.push(std::make_pair(std::string(""), t->root));
  while(!q.empty()){
    std::string s;
    Node* node;
    tie(s, node) = q.front();
    q.pop();
    for(Node::Children::iterator it = node->children.begin();
        it!=node->children.end(); ++it){
      RefPair rp = it->second;
      if(!(rp.node()->children.empty())){
        std::string s1 = s + rp.str().substr();
        q.push(std::make_pair(s1, rp.node()));
        update_max(res, s1);
      }
    }
  }
  return res;
}
```

Firstly, the empty string and root node is pushed to the queue as initialized value. Then the program repeatedly pop from the queue, examine it to see if any child of the node is not a leaf node, push it back to the queue and check if it is the deepest one.

The function "update_max()" is implemented to record all the longest strings.

```
void update_max(Strings& res, std::string s){
  if(res.empty() || (*res.begin()).length() < s.length()){
    res.clear();
    res.push_back(s);
    return;
  }
  if((*res.begin()).length() == s.length())
    res.push_back(s);
}
```

Since the cost of construction a suffix tree is big ($O(n)$ with Ukkonen's algorithm), some lazy initialization approach is used in the main entrance of the finding program.

```
const char TERM1 = '$';

class STreeUtil{
public:
```

```
  STreeUtil():t(0){}
  ~STreeUtil(){ delete t; }

  Strings find_lrs(std::string s){
    lazy(s);
    return lrs(t);
  }

private:
  void lazy(std::string s){
    if((!t) || t→str != s+TERM1){
      delete t;
      t = suffix_tree(s+TERM1);
    }
  }
  STree* t;
};
```

In order to verify the program, some test cases are provided. output for list of strings can be easily realized by overloading "operator¡¡".

```
class StrSTreeTest{
public:
  StrSTreeTest(){
    std::cout<<"start␣string␣manipulation␣over␣suffix␣tree␣testλn";
  }

  void run(){
    test_lrs();
  }

  void test_lrs(){
    __test_lrs("mississippi");
    __test_lrs("banana");
    __test_lrs("cacao");
    __test_lrs("foofooxbarbar");
  }
private:
  void __test_lrs(std::string s){
    std::cout<<"longest␣repeated␣substirng␣of␣"<<s<<"="
             <<util.find_lrs(s)<<"λn";
  }
  STreeUtil util;
};
```

Running these test cases, we can obtain the following result.

```
start string manipulation over suffix tree test
longest repeated substirng of mississippi=[issi, ]
longest repeated substirng of banana=[ana, ]
longest repeated substirng of cacao=[ca, ]
longest repeated substirng of foofooxbarbar=[bar, foo, ]
```

Find the longest repeated sub-string in functional approach

Searching the deepest branch can also be realized in functional way. If the tree is just a leaf node, empty string is returned, else the algorithm will try to find the longest repeated sub-string from the children of the tree.

```
1: function LONGEST-REPEATED-SUBSTRING'(T)
2:     if T is leaf then
3:         return Empty
4:     else
5:         return PROC(CHILDREN(T))
```

```
1: function PROC(L)
2:     if L is empty then
3:         return Empty
4:     else
5:         (s, node) ← FIRST(L)
6:         x ← s + LONGEST − REPEATED − SUBSTRING'(T)
7:         y ← PROC(REST(L))
8:         if LENGTH(x) > LENGTH(y) then
9:             return x
10:        else
11:            return y
```

In $PROC$ function, the first element, which is a pair of edge string and a child node, will be examine firstly. We recursively call the algorithm to find the longest repeated sub-string from the child node, and append it to the edge string. Then we compare this candidate sub string with the result obtained from the rest of the children. The longer one will be returned as the final result.

Note that in case x and y have the same length, it is easy to modify the program to return both of them.

Find the longest repeated sub-string in Haskell

We'll provide 2 versions of Haskell implementation. One version just returns the first candidate in case there are multiple sub-strings which have the same length as the longest sub-string. The other version returns all possible candidates.

```
isLeaf::Tr → Bool
isLeaf Lf = True
isLeaf _ = False

lrs'::Tr→String
lrs' Lf = ""
lrs' (Br lst) = find $ filter (not ∘ isLeaf ∘ snd) lst where
    find [] = ""
    find ((s, t):xs) = maximumBy (compare `on` length) [s++(lrs' t), find xs]
```

In this version, we used the maximumBy function provided in Data.List module. it will only return the first maximum value in a list. In order to return all maximum candidates, we need provide a customized function.

```
maxBy::(Ord a)⇒(a→a→Ordering)→[a]→[a]
maxBy _ [] = []
maxBy cmp (x:xs) = foldl maxBy' [x] xs where
```

```
    maxBy' lst y = case cmp (head lst) y of
                     GT → lst
                     EQ → lst ++ [y]
                     LT → [y]

lrs::Tr→[String]
lrs Lf = [""]
lrs (Br lst) = find $ filter (not ∘ isLeaf ∘ snd) lst where
    find [] = [""]
    find ((s, t):xs) = maxBy (compare `on` length)
                             ((map (s++) (lrs t)) ++ (find xs))
```

We can feed some simple test cases and compare the results of these 2 different program to see their difference.

```
testLRS s = "LRS(" ++ s ++ ")=" ++ (show $ lrs $ suffixTree (s++"$")) ++ "λn"

testLRS' s = "LRS'(" ++ s ++ ")=" ++ (lrs' $ suffixTree (s++"$")) ++ "λn"


test = concat [ f s | s←["mississippi", "banana", "cacao", "foofooxbarbar"],
                      f←[testLRS, testLRS']]
```

Below are the results printed out.

```
LRS(mississippi)=["issi"]
LRS'(mississippi)=issi
LRS(banana)=["ana"]
LRS'(banana)=ana
LRS(cacao)=["ca"]
LRS'(cacao)=ca
LRS(foofooxbarbar)=["bar","foo"]
LRS'(foofooxbarbar)=foo
```

Find the longest repeated sub-string in Scheme/Lisp

Because the underground data structure is list in Scheme/Lisp, in order to access the suffix tree components easily, some helper functions are provided.

```
(define (edge t)
  (car t))

(define (children t)
  (cdr t))

(define (leaf? t)
  (null? (children t)))
```

Similar with the Haskell program, a function which can find all the maximum values on a special measurement rules are given.

```
(define (compare-on func)
  (lambda (x y)
    (cond ((< (func x) (func y)) 'lt)
          ((> (func x) (func y)) 'gt)
          (else 'eq))))
```

```
(define (max-by comp lst)
  (define (update-max xs x)
    (case (comp (car xs) x)
      ('lt (list x))
      ('gt xs)
      (else (cons x xs))))
  (if (null? lst)
      '()
      (fold-left update-max (list (car lst)) (cdr lst))))
```

Then the main function for searching the longest repeated sub-strings can be implemented as the following.

```
(define (lrs t)
  (define (find lst)
    (if (null? lst)
        '("")
        (let ((s (edge (car lst)))
              (tr (children (car lst))))
          (max-by (compare-on string-length)
                  (append
                   (map (lambda (x) (string-append s x)) (lrs tr))
                   (find (cdr lst)))))))
  (if (leaf? t)
      '("")
      (find (filter (lambda (x) (not (leaf? x))) t))))

(define (longest-repeated-substring s)
  (lrs (suffix-tree (string-append s TERM1))))
```

Where TERM1 is defined as "$" string.

Same test cases can be used to verify the results.

```
(define (test-main)
  (let ((fs (list longest-repeated-substring))
        (ss '("mississippi" "banana" "cacao" "foofooxbarbar")))
    (map (lambda (f) (map f ss)) fs)))
```

This test program can be easily extended by adding new test functions as a element of fs list. the result of the above function is as below.

```
(test-main)
;Value 16: ((("issi") ("ana") ("ca") ("bar" "foo")))
```

6.5.3 Find the longest common sub-string

The longest common sub-string of two strings, can also be quickly found by using suffix tree. A typical solution is to build a generalized suffix tree for two strings. If the two strings are denoted as txt_1 and txt_2, a generalized suffix tree is $SuffixTree(txt_1\$_1 txt_2\$_2)$. Where $\$_1$ is a special terminator character for txt_1, and $\$_2$ is another special terminator character for txt_2.

The longest common sub-string is indicated by the deepest branch node, with two forks corresponding to both "...$\$_1$..." and "...$\$_2$"(no $\$_1$). The definition of

the *deepest* node is as same as the one for the longest repeated sub-string, it is the number of characters traversed from root.

If a node has "...$\$_1$..." beneath it, then the node must represent to a sub-string of txt_1, as $\$_1$ is the terminator of txt_1. On the other hand, since it also has "...$\$_2$" (without $\$_1$) child, this node must represent to a sub-string of txt_2 too. Because of it's a deepest one satisfied this criteria. The node indicates to the longest common sub-string.

Find the longest common sub-string imperatively

Based on the above analysis, a BFS (bread first search) algorithm can be used to find the longest common sub-string.

```
1: function LONGEST-COMMON-SUBSTRING(T)
2:     Q ← (NIL, ROOT(T))
3:     R ← NIL
4:     while Q is not empty do
5:         (s, node) ← POP(Q)
6:         if MATCH − FORK(node) then
7:             UPDATE(R, s)
8:         for each ((l, r), node′) in CHILDREN(node) do
9:             if node′ is not leaf then
10:                s′ ← CONCATENATE(s, (l, r))
11:                PUSH(Q, (s′, node′))
12:    return R
```

The most part is as same as the algorithm for finding the longest repeated sub-sting. The function $MATCH - FORK()$ will check if the children of a node satisfy the common sub-string criteria.

Find the longest common sub-string in Python

By translate the imperative algorithm in Python, the following program can be obtained.

```
def lcs(t):
    queue = [("", t.root)]
    res = []
    while len(queue)>0:
        (s, node) = queue.pop(0)
        if match_fork(t, node):
            res = update_max(res, s)
        for _, (str_ref, tr) in node.children.items():
            if len(tr.children)>0:
                s1 = s + t.substr(str_ref)
                queue.append((s1, tr))
    return res
```

Where we define the function match_fork() as below.

```
def is_leaf(node):
    return node.children=={}

def match_fork(t, node):
```

```
    if len(node.children)==2:
        [(_, (str_ref1, tr1)), (_, (str_ref2, tr2))]=node.children.items()
        return is_leaf(tr1) and is_leaf(tr2) and λ
            (t.substr(str_ref1).find(TERM2)!=-1) != λ
            (t.substr(str_ref2).find(TERM2)!=-1)
    return False
```

In this function, it checks if the two children of a node are both leaf, and one contains TERM2 character, while the other doesn't. This is because if one child node is a leaf, it will always contains TERM1 character according to the definition of suffix tree.

Note, the main interface of the function is to add TERM2 to the first string and append TERM1 to the second string.

```
class STreeUtil:
    def __init__(self):
        self.tree = None

    def __lazy__(self, str):
        if self.tree is None or self.tree.str!=str+TERM1:
            self.tree = stree.suffix_tree(str+TERM1)

    def find_lcs(self, s1, s2):
        self.__lazy__(s1+TERM2+s2)
        return lcs(self.tree)
```

We can test this program like below:

```
util = STreeUtil()
print "longest␣common␣substring␣of␣ababa␣and␣baby␣=", util.find_lcs("ababa", "baby")
```

And the output will be something like:

```
longest common substring of ababx and baby = ['bab']
```

Find the longest common sub-string in C++

In C++ implementation, we first define the special terminator characters for the generalized suffix tree of two strings.

```
const char TERM1 = '$';
const char TERM2 = '#';
```

Since the program need frequently test if a node is a branch node or leaf node, a helper function is provided.

```
bool is_leaf(Node* node){
  return node→children.empty();
}
```

The criteria for a candidate node is that it has two children. One in pattern "...#...", the other in pattern "...$".

```
bool match_fork(Node* node){
  if(node→children.size() == 2){
    RefPair rp1, rp2;
    Node::Children::iterator it = node→children.begin();
    rp1 = (it++)→second;
```

```
    rp2 = it→second;
    return (is_leaf(rp1.node()) && is_leaf(rp2.node())) &&
      (rp1.str().substr().find(TERM2)!=std::string::npos)!=
      (rp2.str().substr().find(TERM2)!=std::string::npos);
  }
  return false;
}
```

The main program in BFS(Bread First Search) approach is given as below.

```
Strings lcs(const STree* t){
  std::queue<std::pair<std::string, Node*> > q;
  Strings res;
  q.push(std::make_pair(std::string(""), t→root));
  while(!q.empty()){
    std::string s;
    Node* node;
    tie(s, node) = q.front();
    q.pop();
    if(match_fork(node))
      update_max(res, s);
    for(Node::Children::iterator it = node→children.begin();
        it!=node→children.end(); ++it){
      RefPair rp = it→second;
      if(!is_leaf(rp.node())){
        std::string s1 = s + rp.str().substr();
        q.push(std::make_pair(s1, rp.node()));
      }
    }
  }
  return res;
}
```

After that we can finalize the interface in a lazy way as the following.

```
class STreeUtil{
public:
  //...
  Strings find_lcs(std::string s1, std::string s2){
    lazy(s1+TERM2+s2);
    return lcs(t);
  }
//...
```

This C++ program can generate similar result as the Python one if same test cases are given.

```
longest common substring of ababa, baby =[bab, ]
```

Find the longest common sub-string recursively

The longest common sub-string finding algorithm can also be realized in functional way.

1: **function** LONGEST-COMMON-SUBSTRING'(T)
2: **if** T is leaf **then**
3: **return** *Empty*

```
4:    else
5:        return PROC(CHILDREN(T))
```

If the generalized suffix tree is just a leaf, empty string is returned to indicate the trivial result. In other case, we need process the children of the tree.

```
1: function PROC(L)
2:    if L is empty then
3:        return Empty
4:    else
5:        (s, node) ← FIRST(L)
6:        if MATCH − FORK(node) then
7:            x ← s
8:        else
9:            x ← LONGEST − COMMON − SUBSTRING′(node)
10:           if x ≠ Empty then
11:               x ← s + x
12:       y ← PROC(LEFT(L))
13:       if LENGTH(x) > LENGTH(y) then
14:           return x
15:       else
16:           return y
```

If the children list is empty, the algorithm returns empty string. In other case, the first element, as a pair of edge string and a child node, is first picked, if this child node match the fork criteria (one is in pattern "...$_1$...", the other in pattern "...$_2$" without $_1$), then the edge string is a candidate. The algorithm will process the rest children list and compare with this candidate. The longer one will be returned as the final result. If it doesn't match the fork criteria, we need go on find the longest common sub-string from this child node recursively. and do the similar comparison afterward.

Find the longest common sub-string in Haskell

Similar as the longest repeated sub-string problem, there are two alternative, one is to just return the first longest common sub-string. The other is to return all the candidates.

```
lcs::Tr→[String]
lcs Lf = []
lcs (Br lst) = find $ filter (not ∘ isLeaf ∘ snd) lst where
    find [] = []
    find ((s, t):xs) = maxBy (compare `on` length)
                       (if match t
                        then s:(find xs)
                        else  (map (s++) (lcs t)) ++ (find xs))
```

Most of the program is as same as the one for finding the longest repeated sub-string. The "match" function is defined to check the fork criteria.

```
match (Br [(s1, Lf), (s2, Lf)]) = ("#" `isInfixOf` s1) /= ("#" `isInfixOf` s2)
match _ = False
```

If the function "maximumBy" defined in Data.List is used, only the first candidate will be found.

```
lcs'::Tr→String
lcs' Lf = ""
lcs' (Br lst) = find $ filter (not ∘ isLeaf ∘ snd) lst where
    find [] = ""
    find ((s, t):xs) = maximumBy (compare `on` length)
                       (if match t then [s, find xs]
                           else [tryAdd s (lcs' t), find xs])
    tryAdd x y = if y==="" then "" else x++y
```

We can test this program by some simple cases, below are the snippet of the result in GHCi.

```
lcs $ suffixTree "baby#ababa$"
["bab"]
```

Find the longest common sub-string in Scheme/Lisp

It can be found from the Haskell program, that the common structure of the lrs and lcs are very similar to each other, this hint us that we can abstract to a common search function.

```
(define (search-stree t match)
  (define (find lst)
    (if (null? lst)
        '()
        (let ((s (edge (car lst)))
              (tr (children (car lst))))
          (max-by (compare-on string-length)
                  (if (match tr)
                      (cons s (find (cdr lst)))
                      (append
                       (map (lambda (x) (string-append s x)) (search-stree tr match))
                       (find (cdr lst))))))))
  (if (leaf? t)
      '()
      (find (filter (lambda (x) (not (leaf? x))) t))))
```

This function takes a suffix tree and a function to test if a node match a certain criteria. It will filter out all leaf node first, then repeatedly check if each branch node match the criteria. If matches the function will compare the edge string to see if it is the longest one, else, it will recursively check the child node until either fails or matches.

The longest common sub-string function can be then implemented with this function.

```
(define (xor x y)
  (not (eq? x y)))

(define (longest-common-substring s1 s2)
  (define (match-fork t)
    (and (eq? 2 (length t))
         (and (leaf? (car t)) (leaf? (cadr t)))
         (xor (substring? TERM2 (edge (car t)))
              (substring? TERM2 (edge (cadr t))))))
  (search-stree (suffix-tree (string-append s1 TERM2 s2 TERM1)) match-fork))
```

We can test this function with some simple cases:

```
(longest-common-substring "xbaby" "ababa")
;Value 11: ("bab")

(longest-common-substring "ff" "bb")
;Value: ()
```

6.5.4 Find the longest palindrome in a string

A palindrome is a string, S, such that $S = reverse(S)$, for instance, in English, "level", "rotator", "civic" are all palindrome.

The longest palindrome in a string $s_1 s_2 ... s_n$ can be found in $O(n)$ time with suffix tree. The solution can be benefit from the longest common sub-string problem.

For string S, if a sub-string w is a palindrome, then it must be sub-string of $reverse(S)$ too. for instance, "issi" is a palindrome, it is a sub-string of "mississippi". When turns it reversed to "ippississim", we found that "issi" is also a sub-string.

Based on this truth, we can find get the longest palindrome by finding the longest common sub-string for S and $reverse(S)$.

The algorithm is straightforward for both imperative and functional approach.

function LONGEST-PALINDROME(S)
  return $LONGEST-COMMON-SUBSTRING(SUFFIX-TREE(S+REVERSE(S)))$

Find the longest palindrome in Python

In Python we can reverse a string s like: s[::-1], which means we start from the beginning to ending with step -1.

```
class STreeUtil:
    #...
    def find_lpalindrome(self, s):
        return self.find_lcs(s, s[::-1]) #l[::-1] = reverse(l)
```

We can feed some simple test cases to check if the program can find the palindrome.

```
class StrSTreeTest:
    def test_lpalindrome(self):
        self.__test_lpalindrome__("mississippi")
        self.__test_lpalindrome__("banana")
        self.__test_lpalindrome__("cacao")
        self.__test_lpalindrome__("Woolloomooloo")

    def __test_lpalindrome__(self, s):
        print "longest␣palindrome␣of", s, "=", self.util.find_lpalindrome(s)
```

The result is something like the following.

```
longest palindrome of mississippi = ['ississi']
longest palindrome of banana = ['anana']
```

```
longest palindrome of cacao = ['aca', 'cac']
longest palindrome of Woolloomooloo = ['loomool']
```

Find the longest palindrome in C++

C++ program is just delegate the call to longest common sub-string function.

```
Strings find_lpalindrome(std::string s){
  std::string s1(s);
  std::reverse(s1.begin(), s1.end());
  return find_lcs(s, s1);
}
```

The test cases are added as the following.

```
class StrSTreeTest{
public:
  //...
  void test_lrs(){
    __test_lrs("mississippi");
    __test_lrs("banana");
    __test_lrs("cacao");
    __test_lrs("foofooxbarbar");
  }
private:
  //...
  void __test_lpalindrome(std::string s){
    std::cout<<"longest␣palindrome␣of␣"<<s<<"␣=="
             <<util.find_lpalindrome(s)<<"λn";
  }
```

Running the test cases generate the same result.

```
longest palindrome of mississippi =[ississi, ]
longest palindrome of banana =[anana, ]
longest palindrome of cacao =[aca, cac, ]
longest palindrome of Woolloomooloo =[loomool, ]
```

Find the longest palindrome in Haskell

Haskell program of finding the longest palindrome is implemented as below.

```
longestPalindromes s = lcs $ suffixTree (s++"#"++(reverse s)++"$")
```

If some strings are fed to the program, results like the following can be obtained.

```
longest palindrome(mississippi)=["ississi"]
longest palindrome(banana)=["anana"]
longest palindrome(cacao)=["aca","cac"]
longest palindrome(foofooxbarbar)=["oofoo"]
```

Find the longest palindrome in Scheme/Lisp

Scheme/Lisp program of finding the longest palindrome is realized as the following.

```
(define (longest-palindrome s)
  (longest-common-substring (string-append s TERM2)
                            (string-append (reverse-string s) TERM1)))
```

We can just add this function to the fs list in test-main program, so that the test will automatically done.

```
(define (test-main)
  (let ((fs (list longest-repeated-substring longest-palindrome))
        (ss '("mississippi" "banana" "cacao" "foofooxbarbar")))
    (map (lambda (f) (map f ss)) fs)))
```

The relative result snippet is as below.

```
(test-main)
;Value 12: (... (("ississi") ("anana") ("aca" "cac") ("oofoo")))
```

6.5.5 Others

Suffix tree can also be used in data compression, Burrows-Wheeler transform, LZW compression (LZSS) etc. [2]

6.6 Notes and short summary

Suffix Tree was first introduced by Weiner in 1973 [**?**]. In 1976, McCreight greatly simplified the construction algorithm. McCreight construct the suffix tree from right to left. And in 1995, Ukkonen gave the first on-line construction algorithms from left to right. All the three algorithms are linear time ($O(n)$). And some research shows the relationship among these 3 algorithms. [7]

6.7 Appendix

All programs provided along with this article are free for downloading.

6.7.1 Prerequisite software

GNU Make is used for easy build some of the program. For C++ and ANSI C programs, GNU GCC and G++ 3.4.4 are used. For Haskell programs GHC 6.10.4 is used for building. For Python programs, Python 2.5 is used for testing, for Scheme/Lisp program, MIT Scheme 14.9 is used.

all source files are put in one folder. Invoke 'make' or 'make all' will build C++ and Haskell program.

Run 'make Haskell' will separate build Haskell program. the executable file is "happ" (with .exe in Window like OS). It is also possible to run the program in GHCi.

6.7.2 Tools

Besides them, I use graphviz to draw most of the figures in this post. In order to translate the Trie, Patrica and Suffix Tree output to dot language scripts. I wrote a python program. it can be used like this.

```
st2dot -o filename.dot -t type "string"
```

Where filename.dot is the output file for the dot script, type can be either trie or tree, the default value is tree. it can generate suffix Trie/tree from the string input and turns the tree/Trie into dot script.

This helper scripts can also be downloaded with this article.

download position: http://sites.google.com/site/algoxy/stree/stree.zip

Bibliography

[1] Esko Ukkonen. "On-line construction of suffix trees". Algorithmica 14 (3): 249–260. doi:10.1007/BF01206331. http://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf

[2] Suffix Tree, Wikipedia. http://en.wikipedia.org/wiki/Suffix_tree

[3] Esko Ukkonen. "Suffix tree and suffix array techniques for pattern analysis in strings". http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

[4] Trie, Wikipedia. http://en.wikipedia.org/wiki/Trie

[5] Liu Xinyu. "Trie and Patricia, with Functional and imperative implementation". http://sites.google.com/site/algoxy/trie

[6] Suffix Tree (Java). http://en.literateprograms.org/Suffix_tree_(Java)

[7] Robert Giegerich and Stefan Kurtz. "From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction". Science of Computer Programming 25(2-3):187-218, 1995. http://citeseer.ist.psu.edu/giegerich95comparison.html

[8] Robert Giegerich and Stefan Kurtz. "A Comparison of Imperative and Purely Functional Suffix Tree Constructions". Algorithmica 19 (3): 331–353. doi:10.1007/PL00009177. www.zbh.uni-hamburg.de/pubs/pdf/GieKur1997.pdf

[9] Bryan O'Sullivan. "suffixtree: Efficient, lazy suffix tree implementation". http://hackage.haskell.org/package/suffixtree

[10] Danny. http://hkn.eecs.berkeley.edu/ dyoo/plt/suffixtree/

[11] Zhang Shaojie. "Lecture of Suffix Trees". http://www.cs.ucf.edu/ shzhang/Combio09/lec3.pdf

[12] Lloyd Allison. "Suffix Trees". http://www.allisons.org/ll/AlgDS/Tree/Suffix/

[13] Esko Ukkonen. "Suffix tree and suffix array techniques for pattern analysis in strings". http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

[14] Esko Ukkonen "Approximate string-matching over suffix trees". Proc. CPM 93. Lecture Notes in Computer Science 684, pp. 228-242, Springer 1993. http://www.cs.helsinki.fi/u/ukkonen/cpm931.ps

B-Trees with Functional and imperative implementation
Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

Chapter 7

B-Trees with Functional and imperative implementation

7.1 abstract

B-Tree is introduced by "Introduction to Algorithms" book[3] as one of the advanced data structures. It is important to the modern file systems, some of them are implemented based on B+ tree, which is extended from B-tree. It is also widely used in many database systems. This post provides some implementation of B-trees both in imperative way as described in [3] and in functional way with a kind of modify-and-fix approach. There are multiple programming languages used, including C++, Haskell, Python and Scheme/Lisp.

There may be mistakes in the post, please feel free to point out.

This post is generated by LaTeX 2_ε, and provided with GNU FDL(GNU Free Documentation License). Please refer to http://www.gnu.org/copyleft/fdl.html for detail.

Keywords: B-Trees

7.2 Introduction

In "Introduction to Algorithm" book, B-tree is introduced with the the problem of how to access a large block of data on magnetic disks or secondary storage devices[3]. B-tree is commonly used in databases and file-systems.

It is also helpful to understand B-tree as a generalization of balanced binary search tree[2].

Refer to the Figure 7.1, It is easy to found the difference and similarity of B-tree regarding to binary search tree.

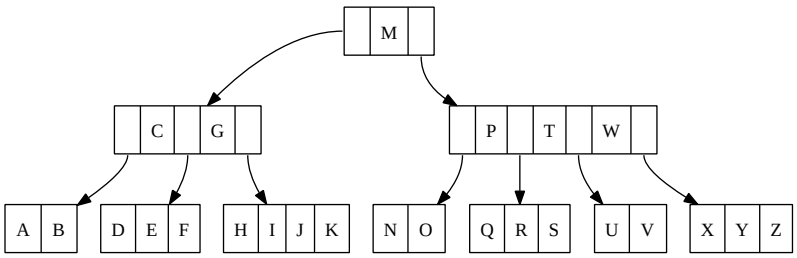


Figure 7.1: An example of B-Tree

Let's review the definition of binary search tree [3].
A binary search tree is

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

An it satisfies the constraint that.

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

The constraint can be represented as the following. for any node n, it satisfies the below equation.

$$\forall x \in LEFT(n), \forall y \in RIGHT(n) \Rightarrow VALUE(x) < VALUE(n) < VALUE(y) \tag{7.1}$$

If we extend this definition to allow multiple keys and children, we get the below definition.
A B-tree is

- either an empty node;
- or a node contains n keys, and $n+1$ children, each child is also a B-Tree, we denote these keys and children as $key_1, key_2, ..., key_n$ and $c_1, c_2, ..., c_n, c_{n+1}$.

Figure 7.2 illustrates a B-Tree node.

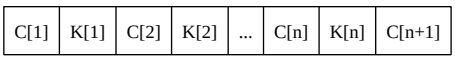


Figure 7.2: A B-Tree node

The keys and children in a node satisfy the following order constraints.

- Keys are stored in non-decreasing order. that is $key_1 \leq key_2 \leq ... \leq key_n$;
- for each key_i, all values stored in child c_i are no bigger than key_i, while all values stored in child c_{i+1} are no less than key_i.

The constraints can be represented as in equation refeq:btree-order as well.

$$\forall x_i \in c_i, i = 0, ..., n, \Rightarrow x_1 \leq key_1 \leq x_2 \leq key_2 \leq ... \leq x_n \leq key_n \leq x_{n+1} \tag{7.2}$$

Finally, if we added some constraints to make the tree balanced, we get the complete definition of B-tree.

- All leaves have the same depth;
- We define an integral number, t, as the *minimum degree* of a B-tree;
  - each node can have at most $2t - 1$ keys;
  - each node can have at least $t - 1$ keys, except the root;

In this post, I'll first introduce How to generate B-trees by insertion algorithm. Two different methods will be explained. One method is discussed in [3] book, the other is a kind of modify-fix approach which quite similar to the algorithm Okasaki used in red-black tree[4]. This method is also discussed in wikipedia[2]. After that, how to delete element from B-tree is explained. As the last part, algorithm for searching in B-tree is also provided.

This article provides example implementation in C, C++, Haskell, Python, and Scheme/Lisp languages.

All source code can be downloaded in appendix 8.7, please refer to appendix for detailed information about build and run.

7.3 Definition

Similar as Binary search tree, B-tree can be defined recursively. Because there are multiple of keys and children, a collection container can be used to store them.

Definition of B-tree in C++

ISO C++ support using const integral number as template parameter. This feature can used to define B-tree with different minimum degree as different type.

```
// t: minimum degree of B-tree
template<class K, int t>
struct BTree{
  typedef K key_type;
  typedef std::vector<K> Keys;
  typedef std::vector<BTree*> Children;

  BTree(){}
```

```
  ~BTree(){
    for(typename Children::iterator it=children.begin();
        it!=children.end(); ++it)
      delete (*it);
  }

  bool full(){ return keys.size() == 2*t-1; }

  bool leaf(){
    return children.empty();
  }

  Keys keys;
  Children children;
};
```

In order to support random access to keys and children, the inner data structure uses STL vector. The node will recursively release all its children. and a two simple auxiliary member functions "full" and "leaf" are provided to test if a node is full or is a leaf node.

Definition of B-tree in Python

If the minimum degree is 2, the B-tree is commonly called as 2-3-4 tree. For illustration purpose, I set 2-3-4 tree as default.

```
TREE_2_3_4 = 2 #by default, create 2-3-4 tree

class BTreeNode:
    def __init__(self, t=TREE_2_3_4, leaf=True):
        self.leaf = leaf
        self.t = t
        self.keys = [] #self.data =○ ..
        self.children = []
```

It's quite OK for B-tree not only store keys, but also store satellite data. However, satellite data is omitted in this post.

Also there are some auxiliary member functions defined

```
class BTreeNode:
    #...
    def is_full(self):
        return len(self.keys) == 2*self.t-1
```

This member function is used to test if a node is full.

Definition of B-tree in Haskell

In Haskell, record syntax is used to define BTree, so that keys and children can be access easily later on. Some auxiliary functions are also provided.

```
data BTree a = Node{ keys :: [a]
                   , children :: [BTree a]
                   , degree :: Int} deriving (Eq, Show)

-- Auxiliary functions
```

```
empty deg = Node [] [] deg

full::BTree a → Bool
full tr = (length $ keys tr) > 2*(degree tr)-1
```

Definition of B-tree in Scheme/Lisp

In Scheme/Lisp, because a list can contain both children and keys at same time, we can organize a B-tree with children and keys interspersed in list. for instance, below list represents a B-tree, the root has one key "c" and two children, the left child is a leaf node, with keys "A" and "B", while the right child is also a leaf with keys "D" and "E".

```
(("A" "B") "C" ("D" "E"))
```

However, this definition doesn't hold the information of minimum degree t. The solution is to pass t as an argument for all operations.

Some auxiliary functions are provided so that we can access and test a B-tree easily.

```
(define (keys tr)
  (if (null? tr)
      '()
      (if (list? (car tr))
          (keys (cdr tr))
          (cons (car tr) (keys (cdr tr))))))

(define (children tr)
  (if (null? tr)
      '()
      (if (list? (car tr))
          (cons (car tr) (children (cdr tr)))
          (children (cdr tr)))))

(define (leaf? tr)
  (or (null? tr)
      (not (list? (car tr)))))
```

Here we assume the key is a simple value, such as a number, or a string, but not a list. In case we find a element is a list, it represents a child B-tree. All above functions are defined based on this assumption.

7.4 Insertion

Insertion is the basic operation to B-tree, a B-tree can be created by inserting keys repeatedly. The essential idea of insertion is similar to the binary search tree. If the keys to be inserted is x, we examine the keys in a node to find a position where all the keys on the left are less than x, while all the keys on the right hand are greater than x. after that we can recursively insert x to the child node at this position.

However, this basic idea need to be fine tuned. The first thing is what the recursion termination criteria is. This problem can be easily solved by define the rule that, in case the node to be inserted is a leaf node. We needn't do

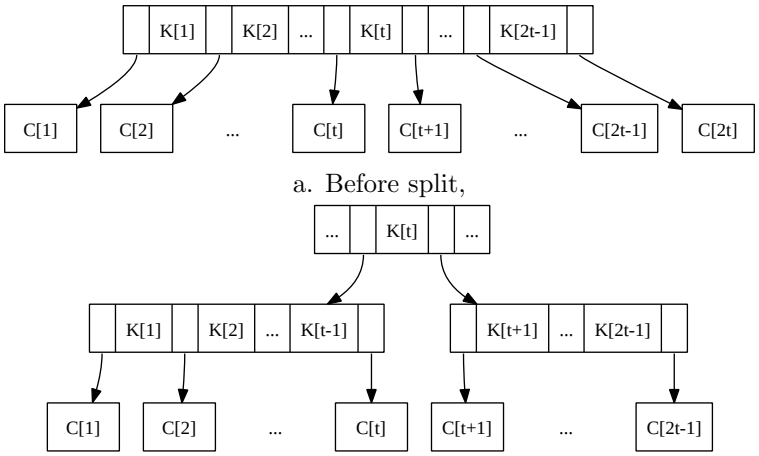
inserting recursively. This is because leaf node don't have children at all. We can just put the x between all left hand keys and right hand keys, which cause the keys number of a leaf node increased by one.

The second thing is how to keep the balance properties of a B-tree when inserting. if a leaf has already $2t-1$ keys, it will break the rule of ' each node can has at most $2t-1$ keys' after we insert x to it. Below sections will show 2 major methods to solve this problem.

7.4.1 Splitting

Regarding to the problem of insert a key to a node, which has already $2t-1$ keys, one solution is to split the node before insertion.

In this case, we can divide the node into 3 parts as shown in Figure 7.3. the left part contains first $t-1$ keys and t children, while the right part contains the last $t-1$ keys and t children. Both left part and right part are valid B-tree nodes. the middle part is just the t-th key. It is pushed up to its parent node (if it already root node, then the t-th key, with 2 children turn be the new root).



a. Before split,

b. After split,

Figure 7.3: Split node

Imperative splitting

If we skip the disk accessing part as explained in [3]. The imperative splitting algorithms can be shown as below.

1: **procedure** B-TREE-SPLIT-CHILD($node, i$)
2: $x \leftarrow CHILDREN(node)[i]$
3: $y \leftarrow CREATE-NODE()$
4: $INSERT(KEYS(node), i, KEYS(x)[t])$
5: $INSERT(CHILDREN(node), i+1, y)$
6: $KEYS(y) \leftarrow KEYS(x)[t+1...2t-1]$
7: $KEYS(x) \leftarrow KEYS(x)[1...t-1]$
8: **if** y is not leaf **then**

9: $CHILDREN(y) \leftarrow CHILDREN(x)[t+1...2t]$
10: $CHILDREN(x) \leftarrow CHILDREN(x)[1...t]$

This algorithm take 2 parameters, one is a B-tree node, the other is the index to indicate which child of this node will be split.

Split implemented in C++

The algorithm can be implemented in C++ as a member function of B-tree node.

```
template<class K, int t>
struct BTree{

  //...

  void split_child(int i){
    BTree<K, t>* x = children[i];
    BTree<K, t>* y = new BTree<K, t>();
    keys.insert(keys.begin()+i, x->keys[t-1]);
    children.insert(children.begin()+i+1, y);
    y->keys = Keys(x->keys.begin()+t, x->keys.end());
    x->keys = Keys(x->keys.begin(), x->keys.begin()+t-1);
    if(!x->leaf()){
      y->children = Children(x->children.begin()+t, x->children.end());
      x->children = Children(x->children.begin(), x->children.begin()+t);
    }
  }
```

Split implemented in Python

We can define splitting operation as a member method of B-tree as the following.

```
class BTreeNode:
    #...
    def split_child(self, i):
        t = self.t
        x = self.children[i]
        y = BTreeNode(t, x.leaf)
        self.keys.insert(i, x.keys[t-1])
        self.children.insert(i+1, y)
        y.keys = x.keys[t:]
        x.keys = x.keys[:t-1]
        if not y.leaf:
            y.children = x.children[t:]
            x.children = x.children[:t]
```

Functional splitting

For functional algorithm, splitting will return a tuple, which contains the left part and right as B-Trees, along with a key.

1: **function** B-TREE-SPLIT($node$)
2: $ks \leftarrow KEYS(node)[1...t-1]$
3: $ks' \leftarrow KEYS(node)[t+1...2t-1]$

4: **if** *node* is not leaf **then**
5: $cs \leftarrow CHILDREN(node)[1...t]$
6: $cs' \leftarrow CHILDREN(node)[t...2t]$
7: **return** $(CREATE-B-TREE(ks, cs), KEYS(node)[t], CREATE-B-TREE(ks', cs'))$

Split implemented in Haskell

Haskell prelude provide take/drop functions to get the part of the list. These functions just returns empty list if the list passed in is empty. So there is no need to test if the node is leaf.

```
split :: BTree a → (BTree a, a, BTree a)
split (Node ks cs t) = (c1, k, c2) where
    c1 = Node (take (t-1) ks) (take t cs) t
    c2 = Node (drop t ks) (drop t cs) t
    k  = head (drop (t-1) ks)
```

Split implemented in Scheme/Lisp

As mentioned previously, the minimum degree t is passed as an argument. The splitting is performed according to t.

```
(define (split tr t)
  (if (leaf? tr)
      (list (list-head tr (- t 1))
            (list-ref tr (- t 1))
            (list-tail tr t))
      (list (list-head tr (- (* t 2) 1))
            (list-ref tr (- (* t 2) 1))
            (list-tail tr (* t 2)))))
```

When splitting a leaf node, because there is no child at all, the program simply take the first $t-1$ keys and the last $t-1$ keys to form two child, and left the t-th key as the only key of the new node. It will return these 3 parts in a list. When splitting a branch node, children must be also taken into account, that's why the first $2t-1$ and the last $2t-1$ elements are taken.

7.4.2 Split before insert method

Note that the split solution will push a key up to its parent node, It is possible that the parent node be full if it has already $2t-1$ keys.

Regarding to this issue, the [3] provides a solution to check every node from root along the path until leaf, in case there is a node in this path is full. the split is applied. Since the parent of this node has been examined except the root node, which ensure the parent node has less than $2t-1$ keys, the pushing up of one key won't make the parent full. This approach need only a single pass down the tree without need of any back-tracking.

The main insert algorithm will first check if the root node need split. If yes, it will create a new node, and set the root as the only child, then performs splitting. and set the new node as the root. After that, the algorithm will try to insert the key to the non-full node.

```
1: function B-TREE-INSERT(T, k)
2:     r ← T
3:     if r is full then
4:         s ← CREATE − NODE()
5:         APPEND(CHILDREN(s), r)
6:         B − TREE − SPLIT − CHILD(s, 1)
7:         r ← s
8:     B − TREE − INSERT − NONFULL(r, k)
9:     return r
```

The algorithm $B-TREE-INSERT-NONFUL$ assert that the node passed in is not full. If it is a leaf node, the new key is just inserted to the proper position based on its order. If it is a branch node. The algorithm finds a proper child node to which the new key will be inserted. If this child node is full, the splitting will be performed firstly.

```
1: procedure B-TREE-INSERT-NONFUL(T, k)
2:     if T is leaf then
3:         i ← 1
4:         while i ≤ LENGTH(KEYS(T)) and k > KEYS(T)[i] do
5:             i ← i + 1
6:         INSERT(KEYS(T), i, k)
7:     else
8:         i ← LENGTH(KEYS(T))
9:         while i > 1andk < KEYS(T)[i] do
10:            i ← i − 1
11:        if CHILDREN(T)[i] is full then
12:            B − TREE − SPLIT − CHILD(T, i)
13:            if k > KEYS(T)[i] then
14:                i ← i + 1
15:        B − TREE − INSERT − NONFULL(CHILDREN(T)[i], k)
```

Note that this algorithm is actually recursive. Consider B-tree typically has minimum degree t relative to magnetic disk structure, and it is balanced tree, Even small depth can support huge amount of data (with $t = 10$, maximum to 10 billion data can be stored in a B-tree with height of 10). Of course it is easy to eliminate the recursive call to improve the algorithm.

In the below language specific implementations, I'll eliminate recursion in C++ program, and show the recursive version in Python program.

Insert implemented in C++

The main insert program in C++ examine if the root is full and performs splitting accordingly. Then it will call insert_nonfull to do the further process.

```
template<class K, int t>
BTree<K, t>* insert(BTree<K, t>* tr, K key){
  BTree<K, t>* root(tr);
  if(root→full()){
    BTree<K, t>* s = new BTree<K, t>();
    s→children.push_back(root);
    s→split_child(0);
    root = s;
```

```
  }
  return insert_nonfull(root, key);
}
```

The recursion is eliminated in insert_nonfull function. If the current node is leaf, it will call ordered_insert to insert the key to the correct position. If it is branch node, the program will find the proper child tree and set it as the current node in next loop. Splitting is performed if the child tree is full.

```
template<class K, int t>
BTree<K, t>* insert_nonfull(BTree<K, t>* tr, K key){
  typedef typename BTree<K, t>::Keys Keys;
  typedef typename BTree<K, t>::Children Children;

  BTree<K, t>* root(tr);
  while(!tr→leaf()){
    unsigned int i=0;
    while(i < tr→keys.size() && tr→keys[i] < key)
      ++i;
    if(tr→children[i]→full()){
      tr→split_child(i);
      if(key > tr→keys[i])
        ++i;
    }
    tr = tr→children[i];
  }
  ordered_insert(tr→keys, key);
  return root;
}
```

Where the ordered_insert is defined as the following.

```
template<class Coll>
void ordered_insert(Coll& coll, typename Coll::value_type x){
  typename Coll::iterator it = coll.begin();
  while(it != coll.end() && *it < x)
    ++it;
  coll.insert(it, x);
}
```

For convenience, I defined auxiliary functions to convert a list of keys into the B-tree.

```
template<class T>
T* insert_key(T* t, typename T::key_type x){
  return insert(t, x);
}

template<class Iterator, class T>
T* list_to_btree(Iterator first, Iterator last, T* t){
  return std::accumulate(first, last, t,
                         std::ptr_fun(insert_key<T>));
}
```

In order to print the result as human readable string, a recursive convert function is provided.

```
template<class T>
std::string btree_to_str(T* tr){
  typename T::Keys::iterator k;
  typename T::Children::iterator c;

  std::ostringstream s;
  s<<"(";
  if(tr->leaf()){
    k=tr->keys.begin();
    s<<*k++;
    for(; k!=tr->keys.end(); ++k)
      s<<", "<<*k;
  }
  else{
    for(k=tr->keys.begin(), c=tr->children.begin();
        k!=tr->keys.end(); ++k, ++c)
      s<<btree_to_str(*c)<<", "<<*k<<", ";
    s<<btree_to_str(*c);
  }
  s<<")";
  return s.str();
}
```

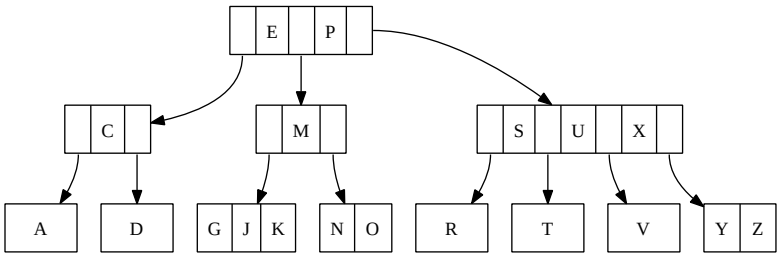
With all the above defined program, some simple test cases can be fed to verify the program.

```
const char* ss[] = {"G", "M", "P", "X", "A", "C", "D", "E", "J", "K", λ
                    "N", "O", "R", "S", "T", "U", "V", "Y", "Z"};
BTree<std::string, 2>* tr234=list_to_btree(ss, ss+sizeof(ss)/sizeof(char*),
                                           new BTree<std::string, 2>);
std::cout<<"2-3-4 tree of ";
std::copy(ss, ss+sizeof(ss)/sizeof(char*),
          std::ostream_iterator<std::string>(std::cout, ", "));
std::cout<<"λn"<<btree_to_str(tr234)<<"λn";
delete tr234;

BTree<std::string, 3>* tr = list_to_btree(ss, ss+sizeof(ss)/sizeof(char*),
                                          new BTree<std::string, 3>);
std::cout<<"B-tree with t=3 of ";
std::copy(ss, ss+sizeof(ss)/sizeof(char*),
          std::ostream_iterator<std::string>(std::cout, ", "));
std::cout<<"λn"<<btree_to_str(tr)<<"λn";
delete tr;
```

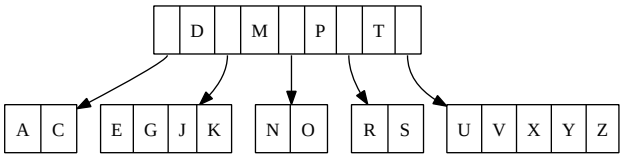
Run these lines will generate the following result:

```
2-3-4 tree of G, M, P, X, A, C, D, E, J, K, N, O, R, S, T, U, V, Y, Z,
(((A), C, (D)), E, ((G, J, K), M, (N, O)), P, ((R), S, (T), U, (V), X, (Y, Z)))
B-tree with t=3 of G, M, P, X, A, C, D, E, J, K, N, O, R, S, T, U, V, Y, Z,
((A, C), D, (E, G, J, K), M, (N, O), P, (R, S), T, (U, V, X, Y, Z))
```

Figure 7.4 shows the result.



a. Insert result of a 2-3-4 tree,



b. Insert result of a B-tree with minimum degree of 3.

Figure 7.4: insert result

Insert implemented in Python

Implement the above insertion algorithm in Python is straightforward, we change the index starts from 0 instead of 1.

```
def B_tree_insert(tr, key): # + data parameter
    root = tr
    if root.is_full():
        s = BTreeNode(root.t, False)
        s.children.insert(0, root)
        s.split_child(0)
        root = s
    B_tree_insert_nonfull(root, key)
    return root
```

And the insertion to non-full node is implemented as the following.

```
def B_tree_insert_nonfull(tr, key):
    if tr.leaf:
        ordered_insert(tr.keys, key)
        #disk_write(tr)
    else:
        i = len(tr.keys)
        while i>0 and key < tr.keys[i-1]:
            i = i-1
        #disk_read(tr.children[i])
        if tr.children[i].is_full():
            tr.split_child(i)
            if key>tr.keys[i]:
                i = i+1
        B_tree_insert_nonfull(tr.children[i], key)
```

Where the function "ordered_insert" function is used to insert an element to an ordered list. Since Python standard list don't support order information. The program is written as below.

```
def ordered_insert(lst, x):
    i = len(lst)
    lst.append(x)
    while i>0 and lst[i]<lst[i-1]:
        (lst[i-1], lst[i]) = (lst[i], lst[i-1])
        i=i-1
```

For the array based collection, append on the tail is much more effective than insert in other position, because the later takes $O(n)$ time, if the length of the collection is n. This program will first append the new element at the end of the existing collection, then iterate from the last element to the first one, and check if the current two elements next to each other are ordered. If not, these two elements will be swapped.

For easily creating a B-tree from a list of keys, we can write a simple helper function.

```
def list_to_B_tree(l, t=TREE_2_3_4):
    tr = BTreeNode(t)
    for x in l:
        tr = B_tree_insert(tr, x)
    return tr
```

By default, this function will create a 2-3-4 tree, and user can specify the minimum degree as the second parameter. The first parameter is a list of keys. This function will repeatedly insert every key into the B-tree which starts from an empty tree.

In order to print the B-tree out for verification, an auxiliary printing function is provided.

```
def B_tree_to_str(tr):
    res = "("
    if tr.leaf:
        res += ",␣".join(tr.keys)
    else:
        for i in range(len(tr.keys)):
            res+= B_tree_to_str(tr.children[i]) + ",␣" + tr.keys[i] + ",␣"
        res += B_tree_to_str(tr.children[len(tr.keys)])
    res += ")"
    return res
```

After that, some smoke test cases can be use to verify the insertion program.

```
class BTreeTest:
    def run(self):
        self.test_insert()

    def test_insert(self):
        lst = ["G", "M", "P", "X", "A", "C", "D", "E", "J", "K", λ
               "N", "O", "R", "S", "T", "U", "V", "Y", "Z"]
        tr = list_to_B_tree(lst)
        print B_tree_to_str(tr)
        print B_tree_to_str(list_to_B_tree(lst, 3))
```

Run the test cases prints two different B-trees. They are identical to the C++ program outputs.

```
(((A), C, (D)), E, ((G, J, K), M, (N, O)), P, ((R), S, (T), U, (V), X, (Y, Z)))
((A, C), D, (E, G, J, K), M, (N, O), P, (R, S), T, (U, V, X, Y, Z))
```

7.4.3 Insert then fix method

Another approach to implement B-tree insertion algorithm is just find the position for the new key and insert it. Since such insertion may violate B-tree properties. We can then apply a fixing procedure after that. If a leaf contains too much keys, we split it into 2 leafs and push a key up to the parent branch node. Of course this operation may cause the parent node violate the B-tree properties, so the algorithm need traverse from leaf to root to perform the fixing.

By using recursive implementation these fixing method can also be realized from top to bottom.

```
1: function B-TREE-INSERT'(T, k)
2:     return FIX − ROOT(RECURSIVE − INSERT(T, k))
```

Where $FIX-ROOT$ examine if the root node contains too many keys, and do splitting if necessary.

```
1: function FIX-ROOT(T)
2:     if FULL?(T) then
3:         T ← B − TREE − SPLIT(T)
4:     return T
```

And the inner function $INSERT(T, k)$ will first check if T is leaf node or branch node. It will do directly insertion for leaf and recursively do insertion for branch.

```
1: function RECURSIVE-INSERT(T, k)
2:     if LEAF?(T) then
3:         INSERT(KEYS(T), k)
4:         return T
5:     else
6:         initialize empty arrays of k', k'', c', c''
7:         i ← 1
8:         while i <= LENGTH(KEYS(T)) and KEYS(T)[i] < k do
9:             APPEND(k', KEYS(T)[i])
10:            APPEND(c', CHILDREN(T)[i])
11:            i ← i + 1
12:        k'' ← KEYS(T)[i...LENGTH(KEYS(T))]
13:        c'' ← CHILDREN(T)[i + 1...LENGTH(CHILDREN(T))]]
14:        c ← CHILDREN(T)[i]
15:        left ← (k', c')
16:        right ← (k'', c'')
17:        return MAKE−B−TREE(left, RECURSIVE−INSERT(c, k), right)
```

Figure 7.5 shows the branch case. The algorithm first locates the position. for certain key k_i, if the new key k to be inserted satisfy $k_{i-1} < k < k_i$, Then we need recursively insert k to child c_i.

This position divides the node into 3 parts, the left part, the child c_i and the right part.

The procedure $MAKE - B - TREE$ take 3 parameters, which relative to the left part, the result after insert k to c_i and right part. It tries to merge these

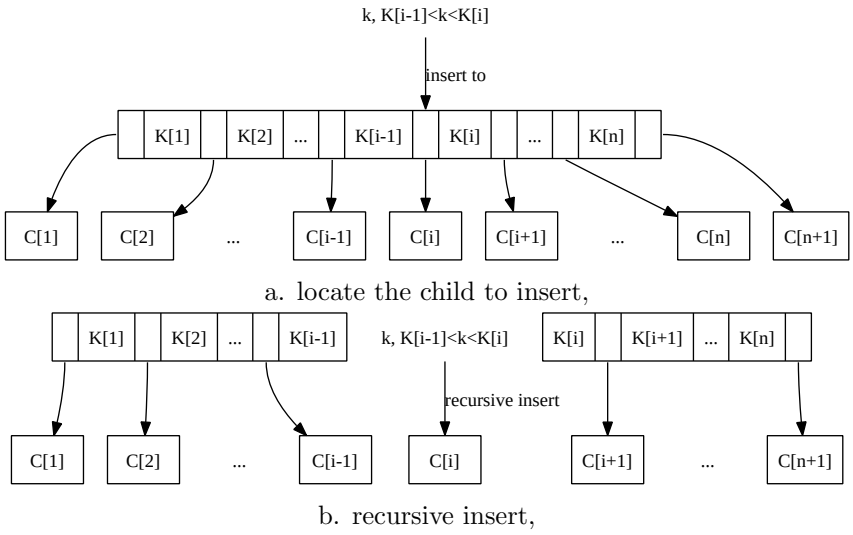


Figure 7.5: Insert a key to a branch node

3 parts into a new B-tree branch node.

However, insert key into a child may make this child violate the B-tree property if it exceed the limitation of the number of keys a node can have. $MAKE - B - TREE$ will detect such situation and try to fix the problem by splitting.

```
1: function MAKE-B-TREE(L, C, R)
2:     if FULL?(C) then
3:         return FIX − FULL(L, C, R)
4:     else
5:         T ← CREATE − NEW − NODE()
6:         KEYS(T) ← KEYS(L) + KEYS(R)
7:         CHILDREN(T) ← CHILDREN(L) + [C] + CHILDREN(R)
8:         return T
```

Where $FIX - FULL$ just calls splitting process.

```
1: function FIX-FULL(L, C, R)
2:     (C′, K, C″) ← B − TREE − SPLIT(C)
3:     T ← CREATE − NEW − NODE()
4:     KEYS(T) ← KEYS(L) + [K] + KEYS(R)
5:     CHILDREN(T) ← CHILDREN(L) + [C′, C″] + CHILDREN(R)
6:     return T
```

Note that splitting may push one extra key up to the parent node. However, even the push-up causes the violation of B-tree property, it will be recursively fixed.

Insert implemented in Haskell

Realize the above recursive algorithm in Haskell can implement this insert-fixing program.

The main program is provided as the following.

```
insert :: (Ord a)⇒ BTree a → a → BTree a
insert tr x = fixRoot $ ins tr x
```

It will just call an auxiliary function 'ins' then examine and fix the root node if contains too many keys.

```
import qualified Data.List as L

--...

ins :: (Ord a) ⇒ BTree a → a → BTree a
ins (Node ks [] t) x = Node (L.insert x ks) [] t
ins (Node ks cs t) x = make (ks', cs') (ins c x) (ks'', cs'')
    where
      (ks', ks'') = L.partition (<x) ks
      (cs', (c:cs'')) = L.splitAt (length ks') cs
```

The 'ins' function uses pattern matching to handle the two different cases. If the node to be inserted is leaf, it will call insert function defined in Haskell standard library, which can insert the new key x into the proper position to keep the order of the keys.

If the node to be inserted is a branch node, the program will recursively insert the key to the child which has the range of keys cover x. After that, it will call 'make' function to combine the result together as a new node. the examine and fixing are performed also by 'make' function.

The function 'fixRoot' first check if the root node contains too many keys, if it exceeds the limit, splitting will be applied. The split result will be used to make a new node, so the total height of the tree increases.

```
fixRoot :: BTree a → BTree a
fixRoot (Node [] [tr] _) = tr -- shrink height
fixRoot tr = if full tr then Node [k] [c1, c2] (degree tr)
             else tr
    where
      (c1, k, c2) = split tr
```

The following is the implementation of 'make' function.

```
make :: ([a], [BTree a]) → BTree a → ([a], [BTree a]) → BTree a
make (ks', cs') c (ks'', cs'')
    | full c = fixFull (ks', cs') c (ks'', cs'')
    | otherwise = Node (ks'++ks'') (cs'++[c]++cs'') (degree c)
```

While 'fixFull' are given like below.

```
fixFull :: ([a], [BTree a]) → BTree a → ([a], [BTree a]) → BTree a
fixFull (ks', cs') c (ks'', cs'') = Node (ks'++[k]++ks'')
                                         (cs'++[c1,c2]++cs'') (degree c)
    where
      (c1, k, c2) = split c
```

In order to print B-tree content out, an auxiliary function 'toString' is provided to convert a B-tree to string.

```
toString :: (Show a)⇒BTree a → String
toString (Node ks [] _) = "("++(L.intercalate ", " (map show ks))++")"
```

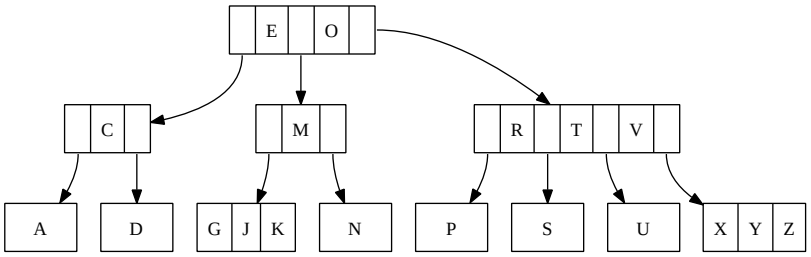
```
toString tr = "("++(toStr (keys tr) (children tr))++")" where
    toStr (k:ks) (c:cs) = (toString c)++", "++(show k)++", "++(toStr ks cs)
    toStr [] [c] = toString c
```

With all the above definition, the insertion program can be verified with some simple test cases.

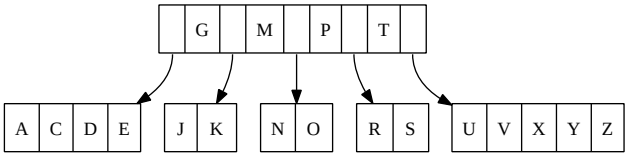
```
listToBTree::(Ord a)⇒[a]→Int→BTree a
listToBTree lst t = foldl insert (empty t) lst

testInsert = do
  putStrLn $ toString $ listToBTree "GMPXACDEJKNORSTUVYZ" 3
  putStrLn $ toString $ listToBTree "GMPXACDEJKNORSTUVYZ" 2
```

Run 'testInsert' will generate the following result.

```
(('A', 'C', 'D', 'E'), 'G', ('J', 'K'), 'M', ('N', 'O'), 'P', ('R', 'S'),
'T', ('U', 'V', 'X', 'Y', 'Z'))
((('A'), 'C', ('D')), 'E', (('G', 'J', 'K'), 'M', ('N')), 'O', (('P'),
'R', ('S'), 'T', ('U'), 'V', ('X', 'Y', 'Z')))
```



a. Insert result of a 2-3-4 tree (insert-fixing method),



b. Insert result of a B-tree with minimum degree of 3 (insert-fixing method).

Figure 7.6: insert and fixing results

Compare the results output by C++ or Python program with this one, as shown in figure 7.6 we can found that there are different points. However, the B-tree built by Haskell program is still valid because all B-tree properties are satisfied. The main reason for this difference is because of the approaching change.

Insert implemented in Scheme/Lisp

The main function for insertion in Scheme/Lisp is given as the following.

```
(define (btree-insert tr x t)
  (define (ins tr x)
```

```
    (if (leaf? tr)
        (ordered-insert tr x) ;;leaf
        (let* ((res (partition-by tr x))
               (left (car res))
               (c (cadr res))
               (right (caddr res)))
          (make-btree left (ins c x) right t))))
  (fix-root (ins tr x) t))
```

The program simply calls an internal function and performs fixing on it. The internal 'ins' function examine if the current node is a leaf node. In case the node is a leaf, it only contains keys, we can located the position and insert the new key there. Otherwise, we partition the node into 3 parts, the left part, the child which the recursive insertion will performed on, and the right part. The program will do the recursive insertion and then combine these three part to a new node. fixing will be happened during the combination.

Function 'ordered-insert' can help to traverse a ordered list and insert the new key to proper position as below.

```
(define (ordered-insert lst x)
  (define (insert-by less-p lst x)
    (if (null? lst)
        (list x)
        (if (less-p x (car lst))
            (cons x lst)
            (cons (car lst) (insert-by less-p (cdr lst) x)))))
  (if (string? x)
      (insert-by string<? lst x)
      (insert-by < lst x)))
```

In order to deal with B-trees with key types both as string and as number, we abstract the less-than function as a parameter and pass it to an internal function.

Function 'partition-by' uses a similar approach.

```
(define (partition-by tr x)
  (define (part-by pred tr x)
    (if (= (length tr) 1)
        (list '() (car tr) '())
        (if (pred (cadr tr) x)
            (let* ((res (part-by pred (cddr tr) x))
                   (left (car res))
                   (c (cadr res))
                   (right (caddr res)))
              (list (cons-pair (car tr) (cadr tr) left) c right))
            (list '() (car tr) (cdr tr)))))
  (if (string? x)
      (part-by string<? tr x)
      (part-by < tr x)))
```

Where 'cons-pair' is a helper function which can put a key, a child in front of a B-tree.

```
(define (cons-pair c k lst)
  (cons c (cons k lst)))
```

In order to fixing the root of a B-tree, which contains too many keys, a 'fix-root' function is provided.

```
(define (full? tr t) ;; t: minimum degree
  (> (length (keys tr))
     (- (* 2 t) 1)))

(define (fix-root tr t)
  (cond ((full? tr t) (split tr t))
        (else tr)))
```

When we turn the recursive insertion result to a new node, we need do fixing if the result node contains too many keys.

```
(define (make-btree l c r t)
  (cond ((full? c t) (fix-full l c r t))
        (else (append l (cons c r)))))

(define (fix-full l c r t)
  (append l (split c t) r))
```

With all above facilities, we can test the program for verification.

In order to build the B-tree easily from a list of keys, some simple helper functions are given.

```
(define (list→btree lst t)
  (fold-left (lambda (tr x) (btree-insert tr x t)) '() lst))

(define (str→slist s)
  (if (string-null? s)
      '()
      (cons (string-head s 1) (str→slist (string-tail s 1)))))
```

A same simple test case as the Haskell one is feed to our program.

```
(define (test-insert)
  (list→btree (str→slist "GMPXACDEJKNORSTUVYZBFHIQW") 3))
```

Evaluate 'test-insert' function can get a B-tree.

```
((("A" "B") "C" ("D" "E" "F") "G" ("H" "I" "J" "K")) "M"
(("N" "O") "P" ("Q" "R" "S") "T" ("U" "V") "W" ("X" "Y" "Z")))
```

It is as same as the result output by the Haskell program.

7.5 Deletion

Deletion is another basic operation of B-tree. Delete a key from a B-tree may cause violating of B-tree balance properties, that a node can't contains too few keys (no less than $t - 1$ keys, where t is minimum degree).

Similar to the approaches for insertion, we can either do some preparation so that the node from where the key will be deleted contains enough keys; or do some fixing after the deletion if the node has too few keys.

7.5.1 Merge before delete method

In textbook[3], the delete algorithm is given as algorithm description. The pseudo code is left as exercises. The description can be used as a good reference when writing the pseudo code.

Merge before delete algorithm implemented imperatively

The first case is the trivial, if the key k to be deleted can be located in node x and x is a leaf node. we can directly remove k from x.

Note that this is a terminal case. For most B-trees which have not only a leaf node as the root. The program will first examine non-leaf nodes.

The second case states that, the key k can be located in node x, however, x isn't a leaf node. In this case, there are 3 sub cases.

- If the child node y precedes k contains enough keys (more than t). We replace k in node x with k', which is the predecessor of k in child y. And recursively remove k' from y.

  The predecessor of k can be easily located as the last key of child y.

- If y doesn't contains enough keys, while the child node z follows k contains more than t keys. We replace k in node x with k'', which is the successor of k in child z. And recursively remove k'' from z.

  The successor of k can be easily located as the first key of child z.

- Otherwise, if neither y, nor z contains enough keys, we can merge y, k and z into one new node, so that this new node contains $2t-1$ keys. After that, we can then recursively do the removing.

  Note that after merge, if the current node doesn't contain any keys, which means k is the only key in x, y and z are the only two children of x. we need shrink the tree height by one.

The case 2 is illustrated as in figure 7.7, 7.8, and 7.9.

Note that although we use recursive way to delete keys in case 2, the recursion can be turned into pure imperative way. We'll show such program in C++ implementation.

the last case states that, if k can't be located in node x, the algorithm need try to find a child node c_i of x, so that sub-tree c_i may contains k. Before the deletion is recursively applied in c_i, we need be sure that there are at least t keys in c_i. If there are not enough keys, we need do the following adjustment.

- We check the two sibling of c_i, which are c_{i-1} and c_{i+1}. If either one of them contains enough keys (at least t keys), we move one key from x down to c_i, and move one key from the sibling up to x. Also we need move the relative child from the sibling to c_i.

  This operation makes c_i contains enough keys OK for deletion. we can next try to delete k from c_i recursively.

- In case neither one of the two siblings contains enough keys, we then merge c_i, a key from x, and either one of the sibling into a new node, and do the deletion on this new node.

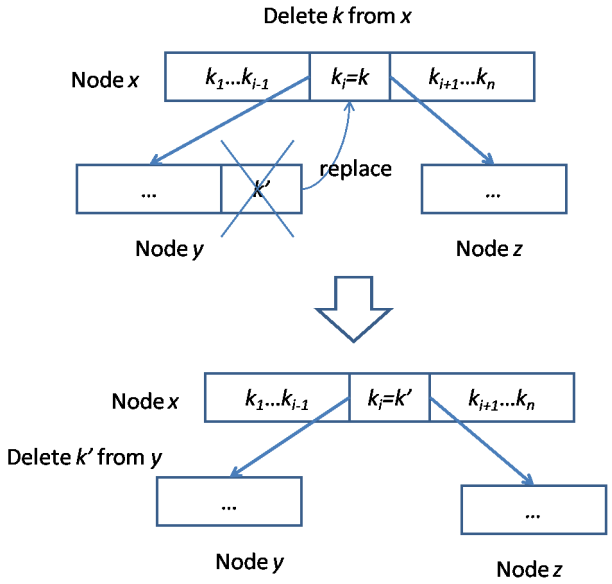


Figure 7.7: case 2a. Replace and delete from predecessor.

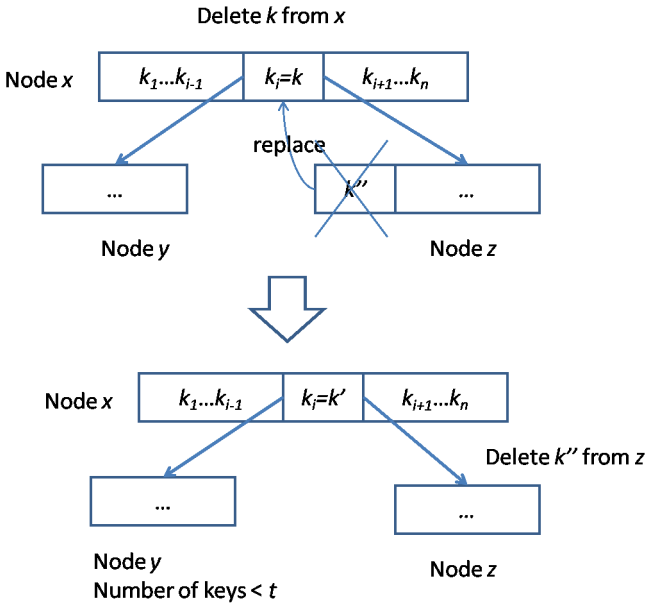


Figure 7.8: case 2b. Replace and delete from successor.

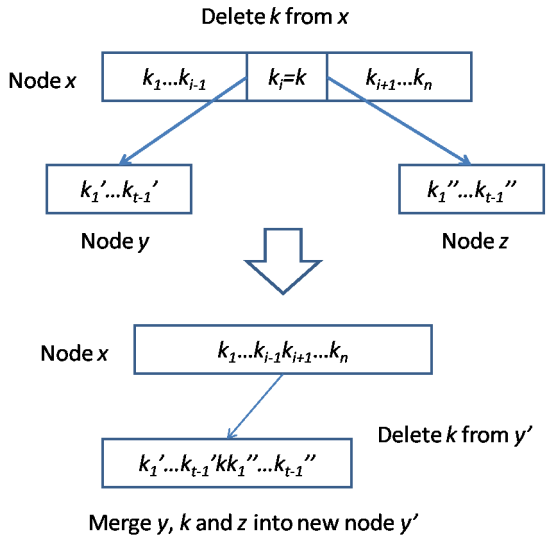


Figure 7.9: case 2c. Merge and delete.

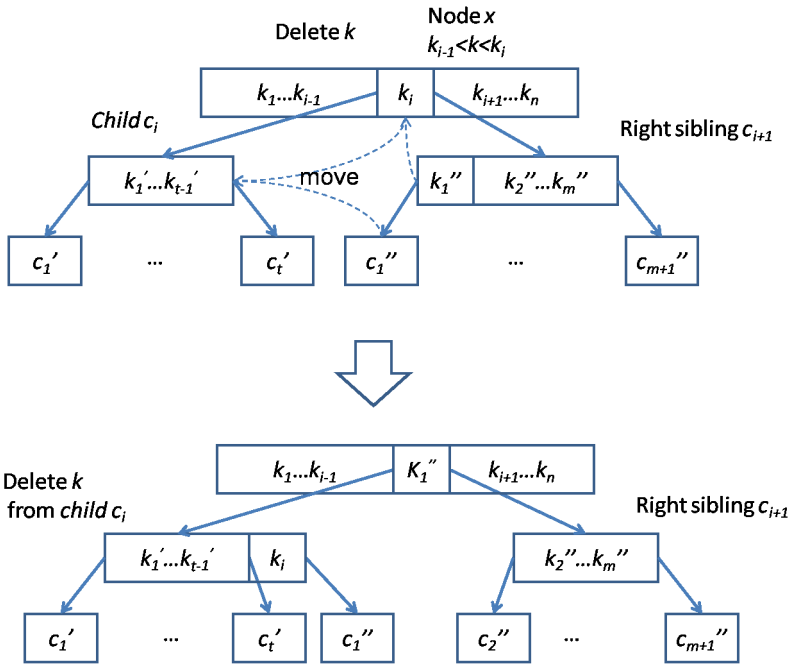


Figure 7.10: case 3a. Borrow from left sibling.

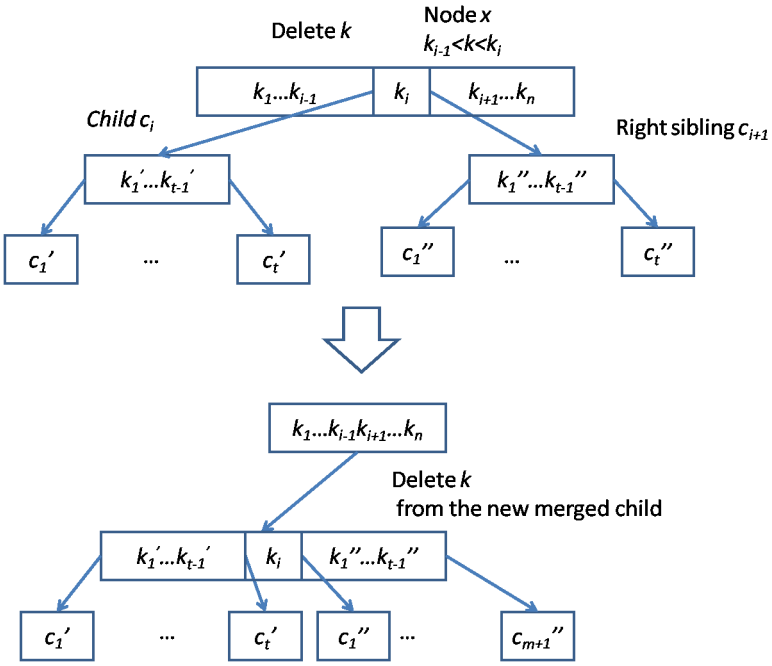


Figure 7.11: case 3b. Borrow Merge and delete.

Case 3 is illustrated in figure 7.10, 7.11.

By implementing the above 3 cases into pseudo code, the B-tree delete algorithm can be given as the following.

First there are some auxiliary functions to do some simple test and operations on a B-tree.

```
1: function CAN-DEL(T)
2:     return number of keys of T ≥ t
```

Function $CAN-DEL$ test if a B-tree node contains enough keys (no less than t keys).

```
1: procedure MERGE-CHILDREN(T, i)          ▷ Merge children i and i + 1
2:     x ← CHILDREN(T)[i]
3:     y ← CHILDREN(T)[i + 1]
4:     APPEND(KEYS(x), KEYS(T)[i])
5:     CONCAT(KEYS(x), KEYS(y))
6:     CONCAT(CHILDREN(x), CHILDREN(y)
7:     REMOVE(KEYS(T), i)
8:     REMOVE(CHILDREN(T), i + 1)
```

Procedure $MERGE-CHILDREN$ merges the i-th child, the i-th key, and $i+1$-th child of node T into a new child, and remove the i-th key and $i+1$-th child after merging.

With these helper functions, the main algorithm of B-tree deletion is described as below.

```
1: function B-TREE-DELETE(T, k)
```

```
2:    i ← 1
3:    while i <= LENGTH(KEYS(T)) do
4:        if k = KEYS(T)[i] then
5:            if T is leaf then                                   ▷ case 1
6:                REMOVE(KEYS(T), k)
7:            else                                                ▷ case 2
8:                if CAN − DEL(CHILDREN(T)[i]) then               ▷ case 2a
9:                    KEYS(T)[i] ← LAST − KEY(CHILDREN(T)[i])
10:                   B−TREE−DELETE(CHILDREN(T)[i], KEYS(T)[i])
11:               else if CAN − DEL(CHILDREN(T)[i+1]) then ▷ case 2b
12:                   KEYS(T)[i] ← FIRST − KEY(CHILDREN(T)[i+1])
13:                   B−TREE−DELETE(CHILDREN(T)[i+1], KEYS(T)[i])
14:               else                                            ▷ case 2c
15:                   MERGE − CHILDREN(T, i)
16:                   B − TREE − DELETE(CHILDREN(T)[i], k)
17:                   if KEYS(T) = NIL then
18:                       T ← CHILDREN(T)[i]                      ▷ Shrinks height
19:           return T
20:       else if k < KEYS(T)[i] then
21:           BREAK
22:       else
23:           i ← i + 1

24:   if T is leaf then
25:       return T                                  ▷ k doesn't exist in T at all
26:   if not CAN − DEL(CHILDREN(T)[i]) then                       ▷ case 3
27:       if i > 1 and CAN − DEL(CHILDREN(T)[i − 1]) then ▷ case 3a:
left sibling
28:           INSERT(KEYS(CHILDREN(T)[i]), KEYS(T)[i − 1])
29:           KEYS(T)[i−1] ← POP−BACK(KEYS(CHILDREN(T)[i−
1]))
30:           if CHILDREN(T)[i] isn't leaf then
31:               c ← POP − BACK(CHILDREN(CHILDREN(T)[i − 1]))
32:               INSERT(CHILDREN(CHILDREN(T)[i]), c)
33:       else if i <= LENGTH(CHILDREN(T)) and CAN−DEL(CHILDREN(T)[i+
1] then                                       ▷ case 3a: right sibling
34:           APPEND(KEYS(CHILDREN(T)[i]), KEYS(T)[i])
35:           KEYS(T)[i] ← POP − FRONT(KEYS(CHILDREN(T)[i +
1]))
36:           if CHILDREN(T)[i] isn't leaf then
37:               c ← POP−FRONT(CHILDREN(CHILDREN(T)[i+1]))
38:               APPEND(CHILDREN(CHILDREN(T)[i]), c)
39:       else                                                    ▷ case 3b
40:           if i > 1 then
41:               MERGE − CHILDREN(T, i − 1)
42:           else
43:               MERGE − CHILDREN(T, i)
44:   B − TREE − DELETE(CHILDREN(T)[i], k)          ▷ recursive delete
45:   if KEYS(T) = NIL then                                       ▷ Shrinks height
```

46: $T \leftarrow CHILDREN(T)[1]$
47: **return** T

Merge before deletion algorithm implemented in C++

The C++ implementation given here isn't simply translate the above pseudo code into C++. The recursion can be eliminated in a pure imperative program.

In order to simplify some B-tree node operation, some auxiliary member functions are added to the B-tree node class definition.

```
template<class K, int t>
struct BTree{
  //...
  // merge children[i], keys[i], and children[i+1] to one node
  void merge_children(int i){
    BTree<K, t>* x = children[i];
    BTree<K, t>* y = children[i+1];
    x->keys.push_back(keys[i]);
    concat(x->keys, y->keys);
    concat(x->children, y->children);
    keys.erase(keys.begin()+i);
    children.erase(children.begin()+i+1);
    y->children.clear();
    delete y;
  }

  key_type replace_key(int i, key_type key){
    keys[i]=key;
    return key;
  }

  bool can_remove(){ return keys.size() >=t; }
  //...
```

Function 'replace_key' can update the i-th key of a node with a new value. Typically, this new value is pulled from a child node as described in deletion algorithm. It will return the new value.

Function 'can_remove' will test if a node contains enough keys for further deletion.

Function 'merge_children' can merge the i-th child, the i-th key, and the $i+1$-th children into one node. This operation is reverse operation of splitting, it can double the keys of a node, so that such adjustment can ensure a node has enough keys for further deleting.

Note that, unlike the other languages equipped with GC, in C++ program, the memory must be released after merging.

This function uses 'concat' function to concatenate two collections. It is defined as the following.

```
template<class Coll>
void concat(Coll& x, Coll& y){
  std::copy(y.begin(), y.end(),
            std::insert_iterator<Coll>(x, x.end()));
}
```

With these helper functions, the main program of B-tree deleting is given as below.

```
template<class T>
T* del(T* tr, typename T::key_type key){
  T* root(tr);
  while(!tr->leaf()){
    unsigned int i = 0;
    bool located(false);
    while(i < tr->keys.size()){
      if(key == tr->keys[i]){
        located = true;
        if(tr->children[i]->can_remove()){ //case 2a
          key = tr->replace_key(i, tr->children[i]->keys.back());
          tr->children[i]->keys.pop_back();
          tr = tr->children[i];
        }
        else if(tr->children[i+1]->can_remove()){ //case 2b
          key = tr->replace_key(i, tr->children[i+1]->keys.front());
          tr->children[i+1]->keys.erase(tr->children[i+1]->keys.begin());
          tr = tr->children[i+1];
        }
        else{ //case 2c
          tr->merge_children(i);
          if(tr->keys.empty()){ //shrinks height
            T* temp = tr->children[0];
            tr->children.clear();
            delete tr;
            tr = temp;
          }
        }
        break;
      }
      else if(key > tr->keys[i])
        i++;
      else
        break;
    }
    if(located)
      continue;
    if(!tr->children[i]->can_remove()){ //case 3
      if(i>0 && tr->children[i-1]->can_remove()){
        // case 3a: left sibling
        tr->children[i]->keys.insert(tr->children[i]->keys.begin(),
                                     tr->keys[i-1]);
        tr->keys[i-1] = tr->children[i-1]->keys.back();
        tr->children[i-1]->keys.pop_back();
        if(!tr->children[i]->leaf()){
          tr->children[i]->children.insert(tr->children[i]->children.begin(),
                                           tr->children[i-1]->children.back());
          tr->children[i-1]->children.pop_back();
        }
      }
      else if(i<tr->children.size() && tr->children[i+1]->can_remove()){
        // case 3a: right sibling
```

```
        tr→children[i]→keys.push_back(tr→keys[i]);
        tr→keys[i] = tr→children[i+1]→keys.front();
        tr→children[i+1]→keys.erase(tr→children[i+1]→keys.begin());
        if(!tr→children[i]→leaf()){
          tr→children[i]→children.push_back(tr→children[i+1]→children.front());
          tr→children[i+1]→children.erase(tr→children[i+1]→children.begin());
        }
      }
      else{
        if(i>0)
          tr→merge_children(i-1);
        else
          tr→merge_children(i);
      }
    }
    tr = tr→children[i];
  }
  tr→keys.erase(remove(tr→keys.begin(), tr→keys.end(), key),
                tr→keys.end());
  if(root→keys.empty()){ //shrinks height
    T* temp = root→children[0];
    root→children.clear();
    delete root;
    root = temp;
  }
  return root;
}
```

Please note how the recursion be eliminated. The main loop terminates only if the current node which is examined is a leaf. Otherwise, the program will go through the B-tree along the path which may contains the key to be deleted, and do proper adjustment including borrowing keys from other nodes, or merging to make the candidate nodes along this path all have enough keys to perform deleting.

In order to verify this program, a quick and simple parsing function which can turn a B-tree description string into a B-tree is provided. Error handling of parsing is omitted for illusion purpose.

```
template<class T>
T* parse(std::string::iterator& first, std::string::iterator last){
  T* tr = new T;
  ++first; //'('
  while(first!=last){
    if(*first=='('){ //child
      tr→children.push_back(parse<T>(first, last));
    }
    else if(*first == ',' || *first == '␣')
      ++first; //skip deliminator
    else if(*first == ')'){
      ++first;
      return tr;
    }
    else{ //key
      typename T::key_type key;
```

```
      while(*first!=',' && *first!=')')
        key+=*first++;
      tr->keys.push_back(key);
    }
  }
  //should never run here
  return 0;
}

template<class T>
T* str_to_btree(std::string s){
  std::string::iterator first(s.begin());
  return parse<T>(first, s.end());
}
```

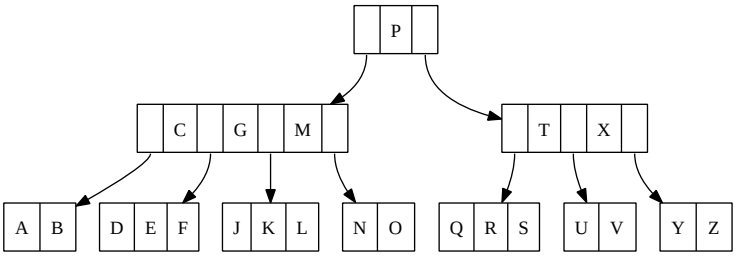
After that, the testing can be performed as below.

```
  void test_delete(){
    std::cout<<"test delete...\n";
    const char* s="(((A,B),C,(D,E,F),G,(J,K,L),M,(N,O)),P,((Q,R,S),T,(U,V),X,(Y,Z)))";
    typedef BTree<std::string, 3> BTr;
    BTr* tr = str_to_btree<BTr>(s);
    std::cout<<"before delete:\n"<<btree_to_str(tr)<<"\n";
    const char* ks[] = {"F", "M", "G", "D", "B", "U"};
    for(unsigned int i=0; i<sizeof(ks)/sizeof(char*); ++i)
      tr=__test_del__(tr, ks[i]);
    delete tr;
  }

  template<class T>
  T* __test_del__(T* tr, typename T::key_type key){
    std::cout<<"delete "<<key<<"==>\n";
    tr = del(tr, key);
    std::cout<<btree_to_str(tr)<<"\n";
    return tr;
  }
```

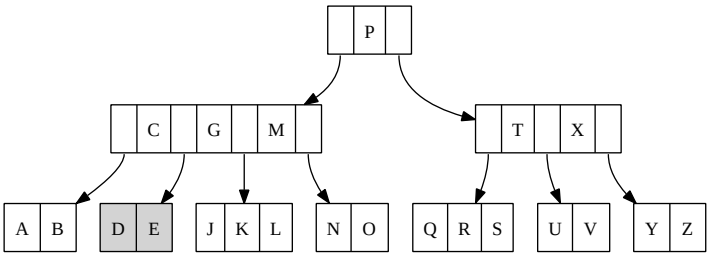
Run 'test_delete' will generate the below result.

```
test delete...
before delete:
(((A, B), C, (D, E, F), G, (J, K, L), M, (N, O)), P, ((Q, R, S), T, (U, V), X, (Y, Z)))
delete F==>
(((A, B), C, (D, E), G, (J, K, L), M, (N, O)), P, ((Q, R, S), T, (U, V), X, (Y, Z)))
delete M==>
(((A, B), C, (D, E), G, (J, K), L, (N, O)), P, ((Q, R, S), T, (U, V), X, (Y, Z)))
delete G==>
(((A, B), C, (D, E, J, K), L, (N, O)), P, ((Q, R, S), T, (U, V), X, (Y, Z)))
delete D==>
((A, B), C, (E, J, K), L, (N, O), P, (Q, R, S), T, (U, V), X, (Y, Z))
delete B==>
((A, C), E, (J, K), L, (N, O), P, (Q, R, S), T, (U, V), X, (Y, Z))
delete U==>
((A, C), E, (J, K), L, (N, O), P, (Q, R), S, (T, V), X, (Y, Z))
```

Figure 7.12, 7.13, and 7.14 show this deleting test process step by step. The nodes modified are shaded. The first 5 steps are as same as the example shown in textbook[3] figure 18.8.



a. A B-tree before performing deleting;



b. After delete key 'F', case 1;

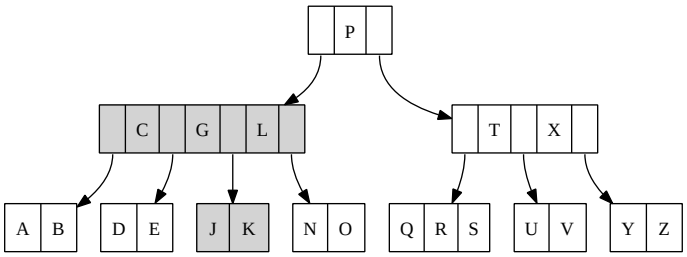
Figure 7.12: Result of B-tree deleting program (1)

Merge before deletion algorithm implemented in Python

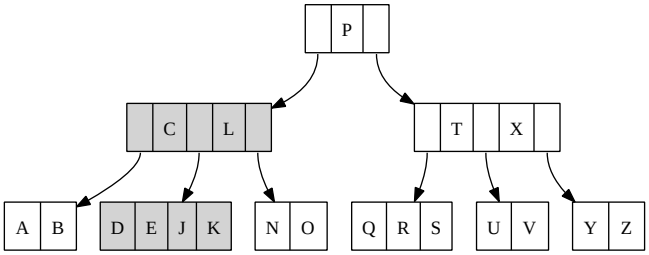
In Python implementation, detailed memory management can be handled by GC. Similar as the C++ program, some auxiliary member functions are added to B-tree node definition.

```
class BTreeNode:
    #...
    def merge_children(self, i):
        #merge children[i] and children[i+1] by pushing keys[i] down
        self.children[i].keys += [self.keys[i]]+self.children[i+1].keys
        self.children[i].children += self.children[i+1].children
        self.keys.pop(i)
        self.children.pop(i+1)

    def replace_key(self, i, key):
        self.keys[i] = key
        return key

    def can_remove(self):
        return len(self.keys) ≥ self.t
```

The member function names are same with the C++ program, so that the meaning for each of them can be referred in previous sub section.

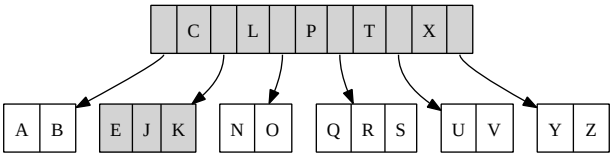


c. After delete key 'M', case 2a;

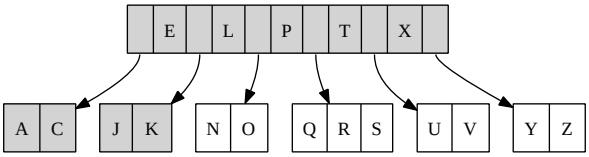


d. After delete key 'G', case 2c;

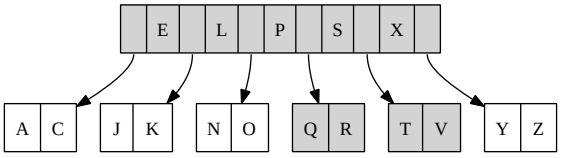
Figure 7.13: Result of B-tree deleting program (2)



e. After delete key 'D', case 3b, and height is shrunk;



f. After delete key 'B', case 3a, borrow from right sibling;



g. After delete key 'U', case 3a, borrow from left sibling;

Figure 7.14: Result of B-tree deleting program (3)

In contrast to the C++ program, a recursion approach similar to the pseudo code is used in this Python program.

```
def B_tree_delete(tr, key):
    i = len(tr.keys)
    while i>0:
        if key == tr.keys[i-1]:
            if tr.leaf:  # case 1 in CLRS
                tr.keys.remove(key)
                #disk_write(tr)
            else: # case 2 in CLRS
                if tr.children[i-1].can_remove(): # case 2a
                    key = tr.replace_key(i-1, tr.children[i-1].keys[-1])
                    B_tree_delete(tr.children[i-1], key)
                elif tr.children[i].can_remove(): # case 2b
                    key = tr.replace_key(i-1, tr.children[i].keys[0])
                    B_tree_delete(tr.children[i], key)
                else: # case 2c
                    tr.merge_children(i-1)
                    B_tree_delete(tr.children[i-1], key)
                    if tr.keys==[]: # tree shrinks in height
                        tr = tr.children[i-1]
            return tr
        elif key > tr.keys[i-1]:
            break
        else:
            i = i-1
    # case 3
    if tr.leaf:
        return tr #key doesn't exist at all
    if not tr.children[i].can_remove():
        if i>0 and tr.children[i-1].can_remove(): #left sibling
            tr.children[i].keys.insert(0, tr.keys[i-1])
            tr.keys[i-1] = tr.children[i-1].keys.pop()
            if not tr.children[i].leaf:
                tr.children[i].children.insert(0, tr.children[i-1].children.pop())
        elif i<len(tr.children) and tr.children[i+1].can_remove(): #right sibling
            tr.children[i].keys.append(tr.keys[i])
            tr.keys[i]=tr.children[i+1].keys.pop(0)
            if not tr.children[i].leaf:
                tr.children[i].children.append(tr.children[i+1].children.pop(0))
        else: # case 3b
            if i>0:
                tr.merge_children(i-1)
            else:
                tr.merge_children(i)
    B_tree_delete(tr.children[i], key)
    if tr.keys==[]: # tree shrinks in height
        tr = tr.children[0]
    return tr
```

In order to verify the deletion program, similar test cases are fed to the function.

```
def test_delete():
    print "test␣delete"
```

```
    t = 3
    tr = BTreeNode(t, False)
    tr.keys=["P"]
    tr.children=[BTreeNode(t, False), BTreeNode(t, False)]
    tr.children[0].keys=["C", "G", "M"]
    tr.children[0].children=[BTreeNode(t), BTreeNode(t), BTreeNode(t), BTreeNode(t)]
    tr.children[0].children[0].keys=["A", "B"]
    tr.children[0].children[1].keys=["D", "E", "F"]
    tr.children[0].children[2].keys=["J", "K", "L"]
    tr.children[0].children[3].keys=["N", "O"]
    tr.children[1].keys=["T", "X"]
    tr.children[1].children=[BTreeNode(t), BTreeNode(t), BTreeNode(t)]
    tr.children[1].children[0].keys=["Q", "R", "S"]
    tr.children[1].children[1].keys=["U", "V"]
    tr.children[1].children[2].keys=["Y", "Z"]
    print B_tree_to_str(tr)
    lst = ["F", "M", "G", "D", "B", "U"]
    reduce(__test_del__, lst, tr)

def __test_del__(tr, key):
    print "delete", key
    tr = B_tree_delete(tr, key)
    print B_tree_to_str(tr)
    return tr
```

In this test case, the B-tree is constructed manually. It is identical to the B-tree built in C++ deleting test case. Run the test function will generate the following result.

```
test delete
(((A, B), C, (D, E, F), G, (J, K, L), M, (N, O)), P, ((Q, R, S), T, (U, V), X, (Y, Z)
delete F
(((A, B), C, (D, E), G, (J, K, L), M, (N, O)), P, ((Q, R, S), T, (U, V), X, (Y, Z)))
delete M
(((A, B), C, (D, E), G, (J, K), L, (N, O)), P, ((Q, R, S), T, (U, V), X, (Y, Z)))
delete G
(((A, B), C, (D, E, J, K), L, (N, O)), P, ((Q, R, S), T, (U, V), X, (Y, Z)))
delete D
((A, B), C, (E, J, K), L, (N, O), P, (Q, R, S), T, (U, V), X, (Y, Z))
delete B
((A, C), E, (J, K), L, (N, O), P, (Q, R, S), T, (U, V), X, (Y, Z))
delete U
((A, C), E, (J, K), L, (N, O), P, (Q, R), S, (T, V), X, (Y, Z))
```

This result is as same as the one output by C++ program.

7.5.2 Delete and fix method

From previous sub-sections, we see how complex is the deletion algorithm, There are several cases, and in each case, there are sub cases to deal.

Another approach to design the deleting algorithm is a kind of delete-then-fix way. It is similar to the insert-then-fix strategy.

When we need delete a key from a B-tree, we firstly try to locate which node this key is contained. This will be a traverse process from the root node towards leaves. We start from root node, If the key doesn't exist in the node, we'll traverse deeper and deeper until we rich a node.

If this node is a leaf node, we can remove the key directly, and then examine if the deletion makes the node contains too few keys to maintain the B-tree balance properties.

If it is a branch node, removing the key will break the node into two parts, we need merge them together. The merging is a recursive process which can be shown in figure 7.15.

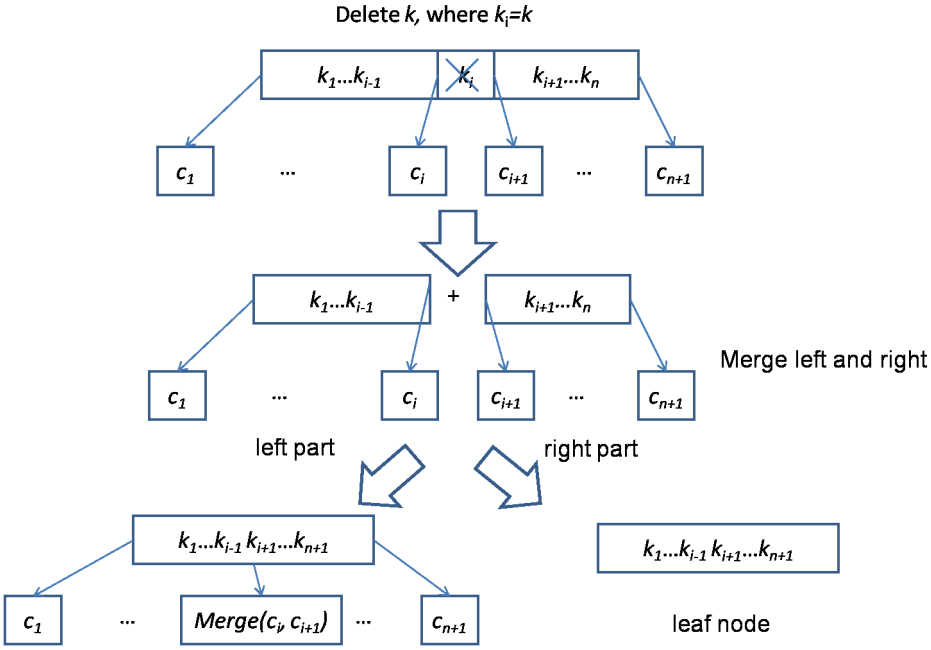


Figure 7.15: Delete a key from a branch node. Removing k_i breaks the node into 2 parts, left part and right part. Merging these 2 parts is a recursive process. When the two parts are leaves, the merging terminates.

When do merging, if the two nodes are not leaves, we merge the keys together, and recursively merge the last child of the left part and the first child of the right part as one new child node. Otherwise, if they are leaves, we merely put all keys together.

Till now, we do the deleting in straightforward way. However, deleting will decrease the number of keys of a node, and it may result in violating the B-tree balance properties. The solution is to perform a fixing along the path we traversed from root.

When we do recursive deletion, the branch node is broken into 3 parts. The left part contains all keys less than k, say $k_1, k_2, ..., k_{i-1}$, and children $c_1, c_2, ..., c_{i-1}$, the right part contains all keys greater than k, say $k_i, k_{i+1}, ..., k_{n+1}$, and children $c_{i+1}, c_{i+2}, ..., c_{n+1}$, the child c_i which recursive deleting applied becomes c_i'. We need make these 3 parts to a new node as shown in figure 7.16.

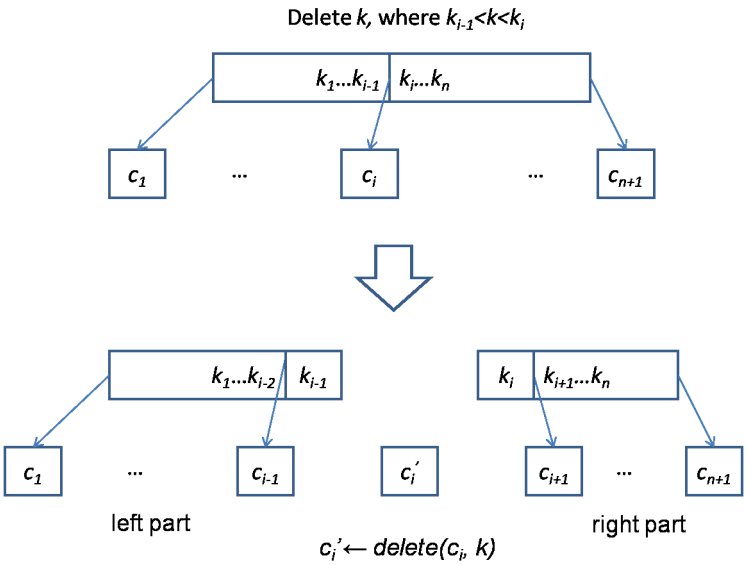


Figure 7.16: Denote c'_i as the result of recursively deleting key k, from child c_i, we should do fixing when making the left part, c'_i and right part together to a new node.

At this time point, we can examine if c'_i contains enough keys, it the number of keys is to less (less than $t-1$, but not t in contrast to merge and delete approach), we can either borrow a key-child pair from left part or right part, and do a inverse operation of splitting. Figure 7.17 shows an example of borrow from left part.

In case both left part and right part is empty, we can simply push c'_i up.

Delete and fix algorithm implemented functionally

By summarize all above analysis, we can draft the delete and fix algorithm.

```
1: function B-TREE-DELETE'(T, k)
2:     return FIX − ROOT(DEL(T, k))
3: function DEL(T, k)
4:     if CHILDREN(T) = NIL then                          ▷ leaf node
5:         DELETE(KEYS(T), k)
6:         return T
7:     else                                               ▷ branch node
8:         n ← LENGTH(KEYS(T))
9:         i ← LOWER − BOUND(KEYS(T), k)
10:        if KEYS(T)[i] = k then
11:            k_l ← KEYS(T)[1, ..., i − 1]
12:            k_r ← KEYS(T)[i + 1, ..., n]
13:            c_l ← CHILDREN(T)[1, ..., i]
14:            c_r ← CHILDREN(T)[i + 1, ..., n + 1]
15:            return MERGE(CREATE − B − TREE(k_l, c_l), CREATE −
```

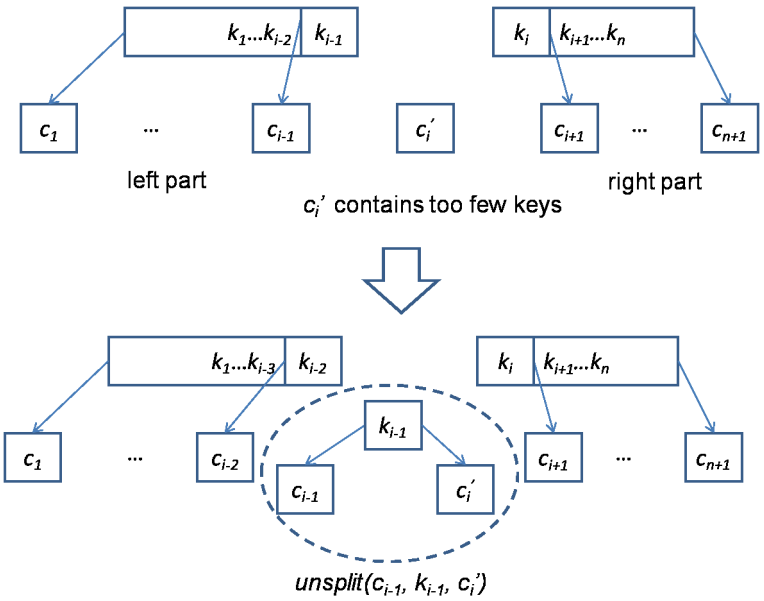


Figure 7.17: Borrow a key-child pair from left part and un-split to a new child.

$B - TREE(k_r, c_r))$

16: **else**
17: $k_l \leftarrow KEYS(T)[1, ..., i-1]$
18: $k_r \leftarrow KEYS(T)[i, ..., n]$
19: $c \leftarrow CHILDREN(T)[i]$
20: $c_l \leftarrow CHILDREN(T)[1, ..., i-1]$
21: $c_r \leftarrow CHILDREN(T)[i+1, ..., n+1]$
22: **return** $MAKE((k_l, c_l), c, (k_r, c_r))$

The main delete function will call an internal DEL function to performs the work, after that, it will apply $FIX - ROOT$ to check if need to shrink the tree height. So the $FIX - ROOT$ function we defined in insertion section should be updated as the following.

1: **function** FIX-ROOT(T)
2: **if** $KEYS(T) = NIL$ **then** ▷ Single child, shrink the height
3: $T \leftarrow CHILDREN(T)[1]$
4: **else if** $FULL?(T)$ **then**
5: $T \leftarrow B - TREE - SPLIT(T)$
6: **return** T

For the recursive merging, the algorithm is given as below. The left part and right part are passed as parameters. If they are leaves, we just put all keys together. Otherwise, we recursively merge the last child of left and the first child of right to a new child, and make this new merged child and the other two parts it breaks into a new node.

1: **function** MERGE(L, R)
2: **if** L, R are leaves **then**

```
3:          T ← CREATE − NEW − NODE()
4:          KEYS(T) ← KEYS(L) + KEYS(R)
5:          return T
6:      else
7:          mgetsLENGTH(KEYS(L))
8:          ngetsLENGTH(KEYS(R))
9:          k_l ← KEYS(L)
10:         k_r ← KEYS(R)
11:         c_l ← CHILDREN(L)[1, ..., m − 1]
12:         c_r ← CHILDREN(R)[2, ..., n]
13:         c ← MERGE(CHILDREN(L)[m], CHILDREN(R)[1])
14:         return MAKE − B − TREE((k_l, c_l), c, (k_r, c_r))
```

In order to make the three parts, the left L, the right R and the child c_i' into a node, we need examine if c_i contains enough keys, together with the process of ensure it contains not too much keys during insertion, we updated the algorithm like the following.

```
1: function MAKE-B-TREE(L, C, R)
2:     if FULL?(C) then
3:         return FIX − FULL(L, C, R)
4:     else if LOW?(C) then
5:         return FIX − LOW(L, C, R)
6:     else
7:         T ← CREATE − NEW − NODE()
8:         KEYS(T) ← KEYS(L) + KEYS(R)
9:         CHILDREN(T) ← CHILDREN(L) + [C] + CHILDREN(R)
10:        return T
```

Where $FIX - LOW$ is defined as the following. In case the left part isn't empty, it will borrow a key-child pair from the left, and do un-split to make the child contains enough keys, then recursively call $MAKE - B - TREE$; If the left part is empty, it will try to borrow key-child pair from the right part, and if both sides are empty, it will returns the child node as result, so that the height shrinks.

```
1: function FIX-LOW(L, C, R)
2:     k_l, c_l ← L
3:     k_r, c_r ← R
4:     m ← LENGTH(k_l)
5:     n ← LENGTH(k_r)
6:     if k_l ≠ NIL then
7:         k_l' ← k_l[1, ..., m − 1]
8:         c_l' ← c_l[1, ..., m − 1]
9:         C' ← UN − SPLIT(c_l[m], k_l[m], C)
10:        return MAKE − B − TREE((k_l', c_l'), C', R)
11:    else if k_r ≠ NIL then
12:        k_r' ← k_r[2, ..., n]
13:        c_r' ← c_r[2, ..., n]
14:        C' ← UN − SPLIT(C, k_r[1], c_r[1])
15:        return MAKE − B − TREE(L, C', (k_r', c_r'))
16:    else
17:        return C
```

Function $UN - SPLIT$ defines as the inverses operation of splitting.

```
1: function UN-SPLIT(L, k, R)
2:     T ← CREATE − B − TREE − NODE()
3:     KEYS(T) ← KEYS(L) + [k] + KEYS(R)
4:     CHILDREN(T) ← CHILDREN(L) + CHILDREN(R)
5:     return T
```

Delete and fix algorithm implemented in Haskell

Based on the analysis of delete-then-fixing approach, a Haskell program can be provided accordingly.

The core deleting function is simple, it just call an internal removing function, then examine the root node to see if the height of the tree can be shrunk.

```
import qualified Data.List as L

delete :: (Ord a) ⇒ BTree a → a → BTree a
delete tr x = fixRoot $ del tr x

del:: (Ord a) ⇒ BTree a → a → BTree a
del (Node ks [] t) x = Node (L.delete x ks) [] t
del (Node ks cs t) x =
    case L.elemIndex x ks of
      Just i → merge (Node (take i ks) (take (i+1) cs) t)
                     (Node (drop (i+1) ks) (drop (i+1) cs) t)
      Nothing → make (ks', cs') (del c x) (ks'', cs'')
    where
      (ks', ks'') = L.partition (<x) ks
      (cs', (c:cs'')) = L.splitAt (length ks') cs
```

Let's focus on the 'del' function, if try to delete a key from a leaf node, it just calls delete function defined in Data.List library. If the key doesn't exist at all, the pre-defined delete function will simply return the list without any modification. For the case of deleting a key from a branch node, it will first examine if the key can be located in this node, and apply recursive merge after remove this key. Otherwise, it will locate the proper child and do recursive delete-then-fixing on this child.

Note that 'partition' and 'splitAt' functions defined in Data.List can help to split the key and children list at the position that all elements on the left is less than the key while the right part are greater than the key.

The recursive merge program has two patterns, merge two leaves and merge two branches. It is given as the following.

```
merge :: BTree a → BTree a → BTree a
merge (Node ks [] t) (Node ks' [] _) = Node (ks++ks') [] t
merge (Node ks cs t) (Node ks' cs' _) = make (ks, init cs)
                                             (merge (last cs) (head cs'))
                                             (ks', tail cs')
```

Where 'init', 'last', 'tail' functions are used to manipulate list which are defined in Haskell prelude.

The fixing part of delete-then-fixing is defined inside 'make' function.

```
make :: ([a], [BTree a]) → BTree a → ([a], [BTree a]) → BTree a
```

```
make (ks', cs') c (ks'', cs'')
    | full c = fixFull (ks', cs') c (ks'', cs'')
    | low c  = fixLow  (ks', cs') c (ks'', cs'')
    | otherwise = Node (ks'++ks'') (cs'++[c]++cs'') (degree c)
```

Where function 'low' is used to test if a node contains too few keys.

```
low::BTree a → Bool
low tr = (length $ keys tr) < (degree tr)-1
```

The real fixing is implemented by try to borrow keys either from left sibling or right sibling as the following.

```
fixLow :: ([a], [BTree a]) → BTree a → ([a], [BTree a]) → BTree a
fixLow (ks'@(_:_), cs') c (ks'', cs'') = make (init ks', init cs')
                                              (unsplit (last cs') (last ks') c)
                                              (ks'', cs'')
fixLow (ks', cs') c (ks''@(_:_), cs'') = make (ks', cs')
                                              (unsplit c (head ks'') (head cs''))
                                              (tail ks'', tail cs'')
fixLow _ c _ = c
```

Note that by using 'x@(_:_)' like pattern can help to ensure 'x' is not empty. Here function 'unsplit' is used which will do inverse splitting operation like below.

```
unsplit :: BTree a → a → BTree a → BTree a
unsplit c1 k c2 = Node ((keys c1)++[k]++(keys c2))
                       ((children c1)++(children c2)) (degree c1)
```

In order to verify the Haskell program, we can provide some simple test cases.

```
import Control.Monad (foldM_, mapM_)

testDelete = foldM_ delShow (listToBTree "GMPXACDEJKNORSTUVYZBFHIQW" 3) "EGAMU"
  where
    delShow tr x = do
      let tr' = delete tr x
      putStrLn $ "delete "++(show x)
      putStrLn $ toString tr'
      return tr'
```

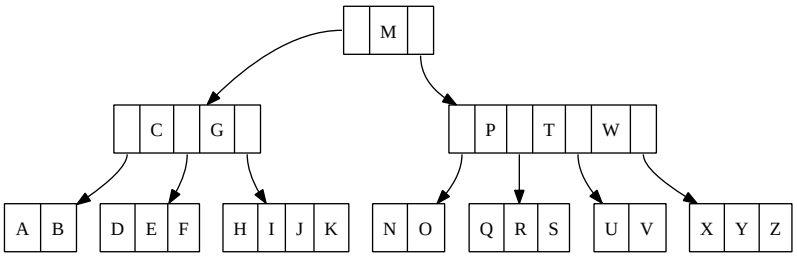
Where function 'listToBTree' and 'toString' are defined in previous section when we explain insertion algorithm.

Run this function will generate the following result.

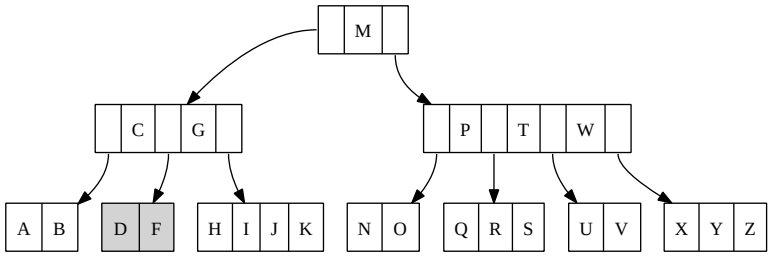
```
delete 'E'
(((('A', 'B'), 'C', ('D', 'F'), 'G', ('H', 'I', 'J', 'K')), 'M',
(('N', 'O'), 'P', ('Q', 'R', 'S'), 'T', ('U', 'V'), 'W', ('X', 'Y', 'Z')))
delete 'G'
(((('A', 'B'), 'C', ('D', 'F'), 'H', ('I', 'J', 'K')), 'M',
(('N', 'O'), 'P', ('Q', 'R', 'S'), 'T', ('U', 'V'), 'W', ('X', 'Y', 'Z')))
delete 'A'
(('B', 'C', 'D', 'F'), 'H', ('I', 'J', 'K'), 'M', ('N', 'O'),
'P', ('Q', 'R', 'S'), 'T', ('U', 'V'), 'W', ('X', 'Y', 'Z'))
delete 'M'
```

```
(('B', 'C', 'D', 'F'), 'H', ('I', 'J', 'K', 'N', 'O'), 'P',
('Q', 'R', 'S'), 'T', ('U', 'V'), 'W', ('X', 'Y', 'Z'))
delete 'U'
(('B', 'C', 'D', 'F'), 'H', ('I', 'J', 'K', 'N', 'O'), 'P',
('Q', 'R', 'S', 'T', 'V'), 'W', ('X', 'Y', 'Z'))
```

If we try to delete the same key from the same B-tree as in merge and fixing approach, we can found that the result is different by using delete-then-fixing methods. Although the results are not as same as each other, both satisfy the B-tree properties, so they are all correct.



a. A B-tree before performing deleting;



b. After delete key 'E'

Figure 7.18: Result of delete-then-fixing (1)

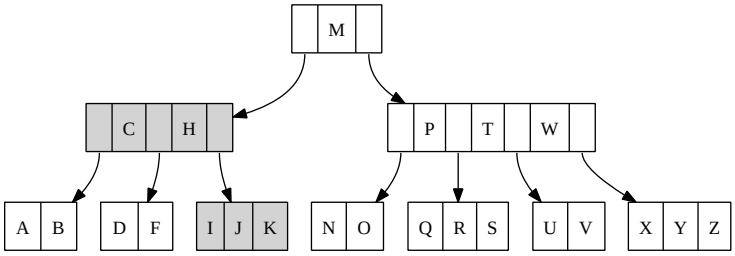
Delete and fix algorithm implemented in Scheme/Lisp

In order to implement delete program in Scheme/Lisp, we provide an extra function to test if a node contains too few keys after deletion.

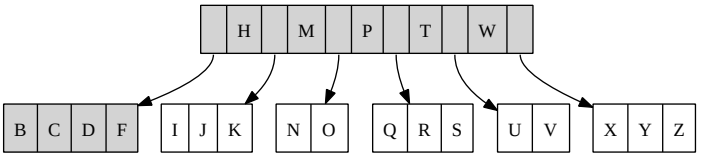
```
(define (low? tr t) ;; t: minimum degree
  (< (length (keys tr))
     (- t 1)))
```

And some general purpose list manipulation functions are defined.

```
(define (rest lst k)
  (list-tail lst (- (length lst) k)))

(define (except-rest lst k)
  (list-head lst (- (length lst) k)))

(define (first lst)
```

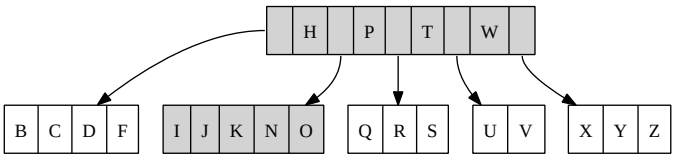


c. After delete key 'G';

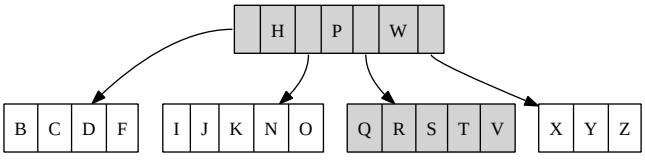


d. After delete key 'A';

Figure 7.19: Result of delete-then-fixing (2)



e. After delete key 'M';



f. After delete key 'U';

Figure 7.20: Result of delete-then-fixing (3)

```
  (if (null? lst) '() (car lst)))

(define (last lst)
  (if (null? lst) '() (car (last-pair lst))))

(define (inits lst)
  (if (null? lst) '() (except-last-pair lst)))
```

Function 'rest' can extract the last k elements from a list, while 'except-rest' used to extract all except the last k elements. 'first' can be treat as a safe 'car', it will return empty list but not throw exception when the list is empty. Function 'last' returns the last element of a list, and if the list is empty, it will return empty result. Function 'inits' returns all excluding the last element.

And a inversion operation of splitting is provided.

```
(define (un-split lst)
  (let ((c1 (car lst))
        (k (cadr lst))
        (c2 (caddr lst)))
    (append c1 (list k) c2)))
λend{lstlising}

The main function of deletion is defined as the following.

λbegin{lstlisting}
(define (btree-delete tr x t)
  (define (del tr x)
    (if (leaf? tr)
        (delete x tr)
        (let* ((res (partition-by tr x))
               (left (car res))
               (c (cadr res))
               (right (caddr res)))
          (if (equal? (first right) x)
              (merge-btree (append left (list c)) (cdr right) t)
              (make-btree left (del c x) right t)))))
  (fix-root (del tr x) t))
```

It is implemented in a similar way as the insertion, call an internal defined 'del' function then apply fixing process on it. In the internal deletion fiction, if the B-tree is a leaf node, the standard list deleting function defined in standard library is applied. If it is a branch node, we call the 'partition-by' function defined previously. This function will divide the node into 3 parts, all children and keys less than x as the left part, a child node next, all keys not less than (greater than or equal to) x and children s the right part.

If the first key in right part is equal to x, it means x can be located in this node, we remove x from right and then call 'merge-btree' to merge left+c, right-x to one new node.

```
(define (merge-btree tr1 tr2 t)
  (if (leaf? tr1)
      (append tr1 tr2)
      (make-btree (inits tr1)
                  (merge-btree (last tr1) (car tr2) t)
                  (cdr tr2)
```

```
                t)))
```

Otherwise, x may be located in c, so we need recursively try to delete x from c.

Function 'fix-root' is updated to handle the cases for deletion as below.

```
(define (fix-root tr t)
  (cond ((null? tr) '()) ;; empty tree
        ((full? tr t) (split tr t))
        ((null? (keys tr)) (car tr)) ;; shrink height
        (else tr)))
```

We added one case to handle if a node contains too few keys after deleting in 'make-btree'.

```
(define (make-btree l c r t)
  (cond ((full? c t) (fix-full l c r t))
        ((low? c t) (fix-low l c r t))
        (else (append l (cons c r)))))
```

Where 'fix-low' is defined to try to borrow a key and a child either from left sibling or right sibling.

```
(define (fix-low l c r t)
  (cond ((not (null? (keys l)))
         (make-btree (except-rest l 2)
                     (un-split (append (rest l 2) (list c)))
                     r t))
        ((not (null? (keys r)))
         (make-btree l
                     (un-split (cons c (list-head r 2)))
                     (list-tail r 2) t))
        (else c)))
```

In order to verify the the deleting program, a simple test is fed to the above defined function.

```
(define (test-delete)
  (define (del-and-show tr x)
    (let ((r (btree-delete tr x 3)))
      (begin (display r) (display "λn") r)))
  (fold-left del-and-show
             (list→btree (str→slist "GMPXACDEJKNORSTUVYZBFHIQW") 3)
             (str→slist "EGAMU")))
```

Run the test will generate the following result.

```
(((A B) C (D F) G (H I J K)) M ((N O) P (Q R S) T (U V) W (X Y Z)))
(((A B) C (D F) H (I J K)) M ((N O) P (Q R S) T (U V) W (X Y Z)))
((B C D F) H (I J K) M (N O) P (Q R S) T (U V) W (X Y Z))
((B C D F) H (I J K N O) P (Q R S) T (U V) W (X Y Z))
((B C D F) H (I J K N O) P (Q R S T V) W (X Y Z))
```

Compare with the output by the Haskell program in previous section, it can be found they are same.

7.6 Searching

Although searching in B-tree can be considered as a generalized form of tree search which extended from binary search tree, it's good to mention that in disk access case, instead of just returning the satellite data corresponding to the key, it's more meaningful to return the whole node, which contains the key.

7.6.1 Imperative search algorithm

When searching in Binary tree, there are only 2 different directions, left and right to go further searching, however, in B-tree, we need extend the search directions to cover the number of children in a node.

```
1: function B-TREE-SEARCH(T, k)
2:     loop
3:         i ← 1
4:         while i ≤ LENGTH(KEYS(T)) and k > KEYS(T)[i] do
5:             k ← k + 1
6:         if i ≤ LENGTH(KEYS(T)) and k = KEYS(T)[i] then
7:             return (T, i)
8:         if T is leaf then
9:             return NIL                          ▷ k doesn't exist at all
10:        else
11:            T ← CHILDREN(T)[i]
```

When doing search, the program examine each key from the root node by traverse from the smallest towards the biggest one. in case it find a matched key, it returns the current node as well as the index of this keys. Otherwise, if it finds this key satisfying $k_i < k < k_{i+1}$, The program will update the current node to be examined as child node c_{i+1}. If it fails to find this key in a leaf node, empty value is returned to indicate the fail case.

Note that in "Introduction to Algorithm", this program is described with recursion, Here the recursion is eliminated.

search program in C++

In C++ implementation, we can use pair provided in STL library as the return type.

```
template<class T>
std::pair<T*, unsigned int> search(T* t, typename T::key_type k){
  for(;;){
    unsigned int i(0);
    for(; i < t->keys.size() && k > t->keys[i]; ++i);
    if(i < t->keys.size() && k == t->keys[i])
      return std::make_pair(t, i);
    if(t->leaf())
      break;
    t = t->children[i];
  }
  return std::make_pair((T*)0, 0); //not found
}
```

And the test cases are given as below.

```
void test_search(){
  std::cout<<"test␣search...λn";
  const char* ss[] = {"G", "M", "P", "X", "A", "C", "D", "E", "J", "K", λ
                      "N", "O", "R", "S", "T", "U", "V", "Y", "Z"};
  BTree<std::string, 3>* tr = list_to_btree(ss, ss+sizeof(ss)/sizeof(char*),
                                            new BTree<std::string, 3>);
  std::cout<<"λn"<<btree_to_str(tr)<<"λn";
  for(unsigned int i=0; i<sizeof(ss)/sizeof(char*); ++i)
    __test_search(tr, ss[i]);
  __test_search(tr, "W");
  delete tr;
}

template<class T>
void __test_search(T* t, typename T::key_type k){
  std::pair<T*, unsigned int> res = search(t, k);
  if(res.first)
    std::cout<<"found␣"<<res.first→keys[res.second]<<"λn";
  else
    std::cout<<"not␣found␣"<<k<<"λn";
}
```

Run 'test_search' function will generate the following result.

```
test search...
((A, C), D, (E, G, J, K), M, (N, O), P, (R, S), T, (U, V, X, Y, Z))
found G
found M
...
found Z
not found W
```

Here the program can find all keys we inserted.

search program in Python

Change a bit the above algorithm in Python gets the program corresponding to the pseudo code mentioned in "Introduction to Algorithm" textbook.

```
def B_tree_search(tr, key):
    for i in range(len(tr.keys)):
        if key≤ tr.keys[i]:
            break
    if key == tr.keys[i]:
        return (tr, i)
    if tr.leaf:
        return None
    else:
        if key>tr.keys[-1]:
            i=i+1
        #disk_read
        return B_tree_search(tr.children[i], key)
```

There is a minor modification from the original pseudo code. We uses for-loop to iterate the keys, the the boundary check is done by compare the last key in the node and adjust the index if necessary.

Let's feed some simple test cases to this program.

```
def test_search():
    lst = ["G", "M", "P", "X", "A", "C", "D", "E", "J", "K", λ
           "N", "O", "R", "S", "T", "U", "V", "Y", "Z"]
    tr = list_to_B_tree(lst, 3)
    print "test␣searchλn", B_tree_to_str(tr)
    for i in lst:
        __test_search__(tr, i)
    __test_search__(tr, "W")

def __test_search__(tr, k):
    res = B_tree_search(tr, k)
    if res is None:
        print k, "not␣found"
    else:
        (node, i) = res
        print "found", node.keys[i]
```

Run the function 'test_search' will generate the following result.

```
found G
found M
...
found Z
W not found
```

7.6.2 Functional search algorithm

The imperative algorithm can be turned into Functional by performing recursive search on a child in case key can't be located in current node.

1: **function** B-TREE-SEARCH(T, k)
2: $i \leftarrow FIND - FIRST(\lambda_x x >= k, KEYS(T))$
3: **if** i exists and $k = KEYS(T)[i]$ **then**
4: **return** (T, i)
5: **if** T is leaf **then**
6: **return** NIL ▷ k doesn't exist at all
7: **else**
8: **return** $B - TREE - SEARCH(CHILDREN(T)[i], k)$

Search program in Haskell

In Haskell program, we first filter out all keys less than the key to be searched. Then check the first element in the result. If it matches, we return the current node along with the index as a tuple. Where the index start from '0'. If it doesn't match, We then do recursive search till leaf node.

```
search :: (Ord a)⇒ BTree a → a → Maybe (BTree a, Int)
search tr@(Node ks cs _) k
    | matchFirst k $ drop len ks = Just (tr, len)
    | otherwise = if null cs then Nothing
                  else search (cs !! len) k
    where
```

```
      matchFirst x (y:_) = x==y
      matchFirst x _ = False
      len = length $ filter (<k) ks
```

The verification test cases are provided as the following.

```
testSearch = mapM_ (showSearch (listToBTree lst 3)) $ lst++"L"
    where
      showSearch tr x = do
        case search tr x of
          Just (_, i) → putStrLn $ "found" ++ (show x)
          Nothing → putStrLn $ "not found" ++ (show x)
      lst = "GMPXACDEJKNORSTUVYZBFHIQW"
```

Here we construct a B-tree from a series of string, then we check if each element in this string can be located. Finally, an non-existed element "L" is fed to verify the failure case.

Run this test function generates the following results.

```
found'G'
found'M'
...
found'W'
not found'L'
```

Search program in Scheme/Lisp

Because we intersperse children and keys in one list in Scheme/Lisp B-tree definition, the search function just move one step a head to locate the key in a node.

```
(define (btree-search tr x)
  ;; find the smallest index where keys[i]≥ x
  (define (find-index tr x)
    (let ((pred (if (string? x) string≥? ≥)))
      (if (null? tr)
          0
          (if (and (not (list? (car tr))) (pred (car tr) x))
              0
              (+ 1 (find-index (cdr tr) x))))))
  (let ((i (find-index tr x)))
    (if (and (< i (length tr)) (equal? x (list-ref tr i)))
        (cons tr i)
        (if (leaf? tr) #f (btree-search (list-ref tr (- i 1)) x)))))
```

The program defines an inner function to find the index of the first element which is greater or equal to the key we are searching.

If the key pointed by this index matches, we are done. Otherwise, this index points to a child which may contains this key. The program will return false result in case the current node is a leaf node.

We can run the below testing function to verify this searching program.

```
(define (test-search)
  (define (search-and-show tr x)
    (if (btree-search tr x)
```

```
        (display (list "found␣" x))
        (display (list "not␣found␣" x))))
  (let* ((lst (str→slist "GMPXACDEJKNORSTUVYZBFHIQW"))
         (tr (list→btree lst 3)))
    (map (lambda (x) (search-and-show tr x)) (cons "L" lst))))
λend{lstlisitng}

A non-existed key ‘‘L’’ is firstly fed, and then all elements
which used to form the B-tree are looked up for verification.

λbegin{lstlisting}
(not found  L)(found  G)(found  M) ○ .. (found  W)
```

7.7 Notes and short summary

In this post, we explained the B-tree data structure as a kind of extension from binary search tree. The background knowledge of magnetic disk access is skipped, user can refer to [3] for detail. For the three main operations, insertion, deletion, and searching, both imperative and functional algorithms are illustrated. The complexity isn't discussed here, However, since B-tree are defined to maintain the balance properties, all operations mentioned here perform $O(lgN)$ where N is the number of the keys in a B-tree.

7.8 Appendix

All programs provided along with this article are free for downloading.

7.8.1 Prerequisite software

GNU Make is used for easy build some of the program. For C++ and ANSI C programs, GNU GCC and G++ 3.4.4 are used. For Haskell programs GHC 6.10.4 is used for building. For Python programs, Python 2.5 is used for testing, for Scheme/Lisp program, MIT Scheme 14.9 is used.

all source files are put in one folder. Invoke 'make' or 'make all' will build C++ and Haskell program.

Run 'make Haskell' will separate build Haskell program. the executable file is "htest" (with .exe in Window like OS). It is also possible to run the program in GHCi.

7.8.2 Tools

Besides them, I use graphviz to draw most of the figures in this post. In order to translate the B-tree output to dot script. A Haskell tool is provided. It can be used like this.

```
bt2dot filename.dot "string"
```

Where filename.dot is the output file for the dot script. It can parse the string which describes B-tree content and translate it into dot script.

This source code of this tool is BTr2dot.hs, it can also be downloaded with this article.

download position: http://sites.google.com/site/algoxy/btree/btree.zip

Bibliography

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

[2] B-tree, Wikipedia. http://en.wikipedia.org/wiki/B-tree

[3] Liu Xinyu. "Comparison of imperative and functional implementation of binary search tree". http://sites.google.com/site/algoxy/bstree

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

Part II

Heaps

Binary Heaps with Functional and imperative implementation

Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

Chapter 8

Binary Heaps with Functional and imperative implementation

8.1 abstract

Heap is one of the elementary data structure. It is widely used to solve some practical problems, such as sorting, prioritized scheduling, and graph algorithms[2].

Most popular implementations of heap are using a kind of implicit binary heap by array, which is described in "Introduction to Algorithm" textbook[3]. Examples include C++/STL heap and Python heapq.

However, heaps can be generalized and implemented with varies of other data structure besides array. In this post, explicit binary tree is used to realize heaps. It leads to Leftist heaps, Skew heaps, and Splay heaps, which are suitable for pure functional implementation as shown by Okasaki[6].

There are multiple programming languages used, including C++, Haskell, Python and Scheme/Lisp.

There may be mistakes in the post, please feel free to point out.

Keywords: Binary Heaps, Leftist Heaps, Skew Heaps, Splay Heaps

8.2 Introduction

Heap is an important elementary data structure. Most of the algorithm text-books introduce heap, especially about binary heap and heap sort.

Some popular implementation, such as C++/STL heap and Python heapq are based on binary heaps (implicit binary heap by array more precisely). And the fastest heap sort algorithm is also written with binary heap as proposed by R. W. Floyd [3] [5].

In this post, we use a general definition of heap, so that varies of under-ground data structures can be used for implementation. And binary heap is also extended to a wide concept under this definition.

A heap is a data structure that satisfies the following *heap property*.

- Top operation always returns the minimum (maximum) element;
- Pop operation removes the top element from the heap while the heap property should be kept, so that the new top element is still the minimum (maximum) one;
- Insert a new element to heap should keep the heap property. That the new top is still the minimum (maximum) element;
- Other operations including merge etc should all keep the heap property.

This is a kind of recursive definition, while it doesn't limit the under ground data structure.

We call the heap with top returns the minimum element *min-heap*, while if top returns the maximum element, we call it *max-heap*.

In this post, I'll first give the definition of binary heap. Then I'll review the traditional imperative way of implicit heap by array. After that, by considering explicit heap by binary trees, I'll explain the Leftist heap, Skew heap, Splay heap, and provide pure functional implementation for them based on Okasaki's result[6].

As the last part, the computation complexity will be mentioned, and I'll show in what situation, Leftist heap performs bad.

I'll introduce some other heaps, including Binomial heaps, Fibonacci heaps, and Pairing heaps in a separate post.

This article provides example implementation in C++, Haskell, Python, and Scheme/Lisp languages.

All source code can be downloaded in appendix 8.7, please refer to appendix for detailed information about build and run.

8.3 Implicit binary heap by array

Considering the heap definition in previous section, one option to implement heap is by using trees. A straightforward solution is to store the minimum (maximum) element in the root node of the tree, so for 'top' operation, we simply return the root as the result. And for 'pop' operation, we can remove the root and rebuild the tree from the children.

If the tree which is used to implement heap is a binary tree, we can call it *binary heap*. There are three types of binary heap implementation explained in this post. All of them are based on binary tree.

8.3.1 Definition

The first one is implicit binary tree indeed. Consider the problem how to represent a complete binary tree with array. (For example, try to represent a complete binary tree in a programming language doesn't support structure or record data type, so that only array can be used). One solution is to pack all element from top level (root) down to bottom level (leaves).

Figure 8.1 shows a complete binary tree and its corresponding array representation.

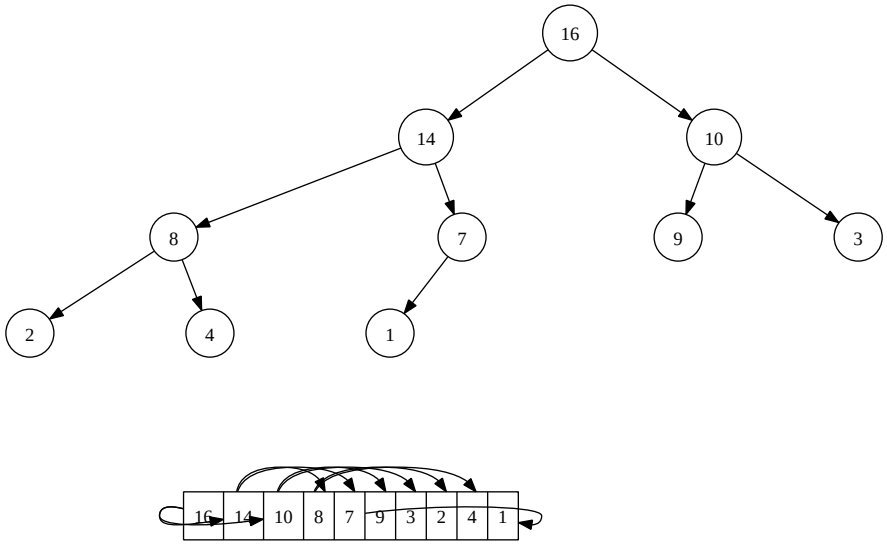


Figure 8.1: Mapping between a complete binary tree and array

This mapping relationship between tree and array can be denote as the following equations (Note that the array index starts from 1).

1: **function** PARENT(i)
2: **return** $\lfloor \frac{i}{2} \rfloor$

3: **function** LEFT(i)
4: **return** $2i$

5: **function** LEFT(i)
6: **return** $2i + 1$

For a given tree node which is represented as the i-th element in an array, since the tree is complete, we can easily find its parent node as the $\lfloor i/2 \rfloor$-th element in the array; Its left child with index of $2i$ and right child with index of $2i + 1$. If the index of the child exceeds the length of the array, it simply means this node don't have such child.

This mapping calculation can be performed fast if bit-wise operation is used.

Definition of implicit binary heap by array in C++

In C++ language, array index starts from zero, but not one. The mapping from array to binary tree should be adjusted accordingly.

```
template<class T>
T parent(T i){ return ((i+1)>>1)-1; }

template<class T>
T left(T i){ return (i<<1)+1; }

template<class T>
T right(T i){ return (i+1)<<1; }
```

The type T must support bit-wise operation.

Definition of implicit binary heap by array in Python

Similar as C/C++, the array index in Python starts from 0, so we provide the mapping functions as below.

```
def parent(i):
    return (i+1)//2-1

def left(i):
    return  2*i+1

def right(i):
    return 2*(i+1)
```

8.3.2 Heapify

The most important thing for heap algorithm is to maintain the heap property, that the top element should be the minimum (maximum) one.

For the implicit binary heap by array, it means for a given node, which is represented as the i-th index, we must develop a algorithm to check if all its two children conform to this property and in case there is violation, we need swap the parent and child to fix the problem.

In "Introduction to Algorithms" book[3], this algorithm is given in a recursive way, here we show a pure imperative solution. Let's take min-heap for example.

```
1: function HEAPIFY(A, i)
2:     n ← LENGTH(A)
3:     loop
4:         l ← LEFT(i)
5:         r ← RIGHT(i)
6:         smallest ← i
7:         if l < n and A[l] < A[i] then
8:             smallest ← l
9:         if r < n and A[r] < A[smallest] then
10:            smallest ← r
11:        if smallest ≠ i then
12:            exchangeA[i] ↔ A[smallest]
```

13: $i \leftarrow smallest$
14: **else**
15: **return**

This algorithm assume that for a given node, the children all conform to the heap property, however, we are not sure if the value of this node is the smallest compare to its tow children.

For array A and a given index i, we need check none its left child or right child is bigger than $A[i]$, in case we find violation, we pick the smallest one, and set it as the new value for $A[i]$, the previous value of $A[i]$ is then set as the new value of the child, and we need go along the child tree to repeat this check and fixing process until we either reach a leaf node or there is no heap property violation.

Note that the $HEAPIFY$ algorithm takes $O(\lg N)$ time.

Heapify in C++

For C++ program, we need it cover both traditional C compatible array, and the modern container abstraction. There are several options to realize this requirement.

One method is to pass iterators as argument to heap algorithm. C++/STL implementation (at the time the author wrote this post) uses this approach.

The advantage of using iterator is that some random access iterator is just implemented as C pointers, so it compatible with C array well.

However, this method need us change the algorithm from a array index style to pointer operation style. It's hard to reflect the above pseudo code quite clear in such style. Because of this problem, we won't use this approach here. Reader can refer to STL source code for detailed information.

Another option is to pass the array or container as well as the number of elements as arguments. However, we need abstract the comparison anyway so that the algorithm works for both max-heap and min-heap.

```
template<class T> struct MinHeap: public std::less<T>{};
template<class T> struct MaxHeap: public std::greater<T>{};
```

Here we define MinHeap and MaxHeap as a kind of alias of less-than and greater-than logic comparison functor template.

The heapify algorithm can be implemented as the following.

```
template<class Array, class LessOp>
void heapify(Array& a, unsigned int i, unsigned int n, LessOp lt){
  while(true){
    unsigned int l=left(i);
    unsigned int r=right(i);
    unsigned int smallest=i;
    if(l < n && lt(a[l], a[i]))
      smallest = l;
    if(r < n && lt(a[r], a[smallest]))
      smallest = r;
    if(smallest != i){
      std::swap(a[i], a[smallest]);
      i = smallest;
    }
    else
```

```
      break;
  }
}
```

The program accepts the reference of the array (both reference of the container, and reference to pointer are OK), the index from where we want to adjust so that all children of it confirms to the heap property; the number of elements in the array, and a comparison functor. It checks from the node which is indexed as i down to the leaf until it find a node that both children are "less than" the value of the node based on the comparison functor. Otherwise, it will locate the "smallest" one and swap it with the node value.

It is also possible to create a concept of range and realize the algorithm with it. Some C++ library, such as boost has already support range. Here we can develop a light weight range only for random access container.

```
template<class RIter> // random access iterator
struct Range{
  typedef typename std::iterator_traits<RIter>::value_type value_type;
  typedef typename std::iterator_traits<RIter>::difference_type size_t;
  typedef typename std::iterator_traits<RIter>::reference  reference;
  typedef RIter iterator;

  Range(RIter left, RIter right):first(left), last(right){}

  reference  operator[](size_t i){ return *(first+i); }
  size_t size() const { return last-first; }

  RIter first;
  RIter last;
};
```

For a given left index l, and right index r, a range represents $[l, r)$, So it is easy to construct a range with iterators as well as the pointer of the array and its length.

Two overloaded auxiliary function templates are provided to create range easily.

```
template<class Iter>
Range<Iter> range(Iter left, Iter right){ return Range<Iter>(left, right); }

template<class Iter>
Range<Iter> range(Iter left, typename Range<Iter>::size_t n){
  return Range<Iter>(left, left+n);
}
```

The above algorithm can be implemented with range like below.

```
template<class R, class LessOp>
void heapify(R a, typename R::size_t i, LessOp lt){
  typename R::size_t l, r, smallest;
  while(true){
    l = left(i);
    r = right(i);
    smallest = i;
    if( l < a.size() && lt(a[l], a[i]))
      smallest = l;
```

```
    if( r < a.size() && lt(a[r], a[smallest]))
      smallest = r;
    if( smallest != i){
      std::swap(a[i], a[smallest]);
      i = smallest;
    }
    else
      break;
  }
}
```

Almost everything is as same as the former one except that the number of elements can be given by the size of the range.

In order to verify the program, a same test case as in figure 6.2 in [3] is fed to our function.

```
// test c-array
const int a[] = {16, 4, 10, 14, 7, 9, 3, 2, 8, 1};
const unsigned int n = sizeof(a)/sizeof(a[0]);
int x[n];
std::copy(a, a+n, x);
heapify(x, 1, n, MaxHeap<int>());
print_range(x, x+n);

// test random access container
std::vector<short> y(a, a+n);
heapify(y, 1, n, MaxHeap<short>());
print_range(y.begin(), y.end());
```

The same test case can also be applied to the "range" version of program.

```
heapify(range(x, n), 1, MaxHeap<int>());

//...

heapify(range(y.begin(), y.end()), 1, MaxHeap<short>());
```

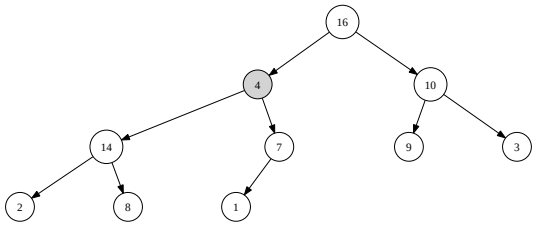
Where "print_range" is a helper function to output all elements in a container, a C array or a range.

```
template<class Iter>
void print_range(Iter first, Iter last){
  for(; first!=last; ++first)
    std::cout<<*first<<",␣";
  std::cout<<"\n";
}

template<class R>
void print_range(R a){
  print_range(a.first, a.last);
}
```

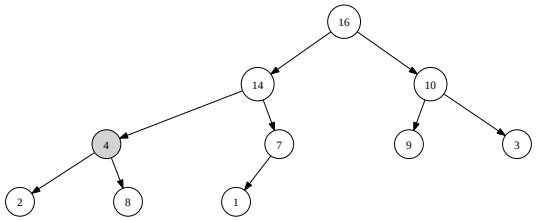
The above test code can output a result as below:

```
16, 14, 10, 8, 7, 9, 3, 2, 4, 1,
16, 14, 10, 8, 7, 9, 3, 2, 4, 1,
```

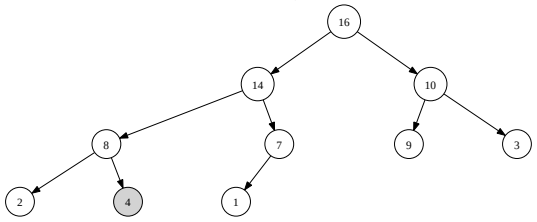
Figure 8.2 shows how this algorithm works.



a. Step 1, 14 is the biggest element among 4, 14, and 7. Swap 4 with the left child;



b. Step 2, 8 is the biggest element among 2, 4, and 8. Swap 4 with the right child;



c. 4 is the leaf node. It hasn't any children. Process terminates.

Figure 8.2: Heapify example, a max-heap case.

Heapify in Python

In order to cover both min-heap and max-heap, we abstract the comparison as the following lambda expressions.

```
MIN_HEAP = lambda a, b: a < b
MAX_HEAP = lambda a, b: a > b
```

By passing the above defined comparison operation as an argument, the "Heapify" algorithm is given as below.

```
def heapify(x, i, less_p = MIN_HEAP):
    n = len(x)
    while True:
        l = left(i)
        r = right(i)
        smallest = i
        if l < n and less_p(x[l], x[i]):
            smallest = l
        if r < n and less_p(x[r], x[smallest]):
            smallest = r
        if smallest != i:
            (x[i], x[smallest])=(x[smallest], x[i])
            i  = smallest
        else:
            break
```

We can use the same test case as presents in Figure 6.2 of [3].

```
l = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
heapify(l, 1, MAX_HEAP)
print l
```

The result is something like this.

```
[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
```

This result is as same as the one presented in 8.2.

8.3.3 Build a heap

With heapify algorithm, it is easy to build a heap from an arbitrary array. Observe that the number of nodes in a complete binary tree for each level is a list like:

$1, 2, 4, 8, ..., 2^i, ...$

The only exception is the last level. Since the tree may not full (note that complete binary tree doesn't mean full binary tree), the last level contains at most 2^{p-1} nodes, where $2^p \leq n$ and n is the length of the array.

Heapify algorithm doesn't take any effect on leave node, which means we can skip applying heapify for all nodes. In other words, all leaf nodes have already satisfied heap property. We only need start checking and maintaining heap property from the last branch node. the Index of the last branch node is no greater than $\lfloor n/2 rfloor$.

Based on this fact, we can build a heap with the following algorithm. (Assume the heap is min-heap).

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

Although the complexity of $HEAPIFY$ is $O(\lg N)$, the running time of $BUILD\_HEAP$ doesn't bound to $O(N \lg N)$ but to $O(N)$, so this is a linear time algorithm. Please refer to [3] for the detailed proof.

Build a heap in C++

The only adjustment in C++ program from the above algorithm is about the starting index from 1 to 0.

```
template<class Array, class LessOp>
void build_heap(Array& a, unsigned int n, LessOp lt){
  unsigned int i = (n-1)>>1;
  while(true){
    heapify(a, i, n, lt);
    if(i==0) break; // this is a trick: unsigned int always ≥0
    --i;
  }
}
```

Note that since the unsigned type is used to represent index, It can't lower than zero. We can't just use a for loop as below.

```
for(unsigned int i = (n-1)>>1; i≥0; --i) //wrong, i always ≥0
```

This program can be easily adjusted with range concept.

```
template<class RangeType, class LessOp>
void build_heap(RangeType a, LessOp lt){
  typename RangeType::size_t i = (a.size()-1)>>1;
  while(true){
    heapify(a, i, lt);
    if(i==0) break;
    --i;
  }
}
```

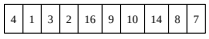
We can test our program with the same data as in Figure 6.3 in [3].

```
// test c-array
const int a[] = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7};
const unsigned int n = sizeof(a)/sizeof(a[0]);
int x[n];
std::copy(a, a+n, x);
build_heap(range(x, n), MaxHeap<int>());
print_range(x, x+n);

// test random access container
std::vector<int> y(a, a+n);
build_heap(range(y.begin(), y.end()), MaxHeap<short>());
print_range(y.begin(), y.end());
```

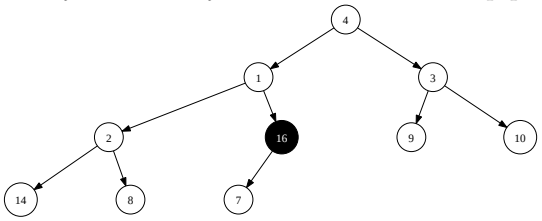
Running results are printed in console like the following.

```
16, 14, 10, 8, 7, 9, 3, 2, 4, 1,
16, 14, 10, 8, 7, 9, 3, 2, 4, 1,
```

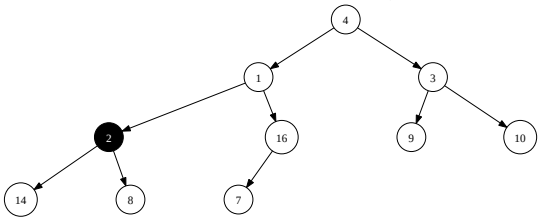
Figure 8.3 and 8.4 show the steps when build a heap from an arbitrary array. The node in black color is the one we will apply $HEAPIFY$ algorithm, the nodes in gray color are swapped during $HEAPIFY$.



a. An array in arbitrary order before build-heap process;



b. Step 1, The array is mapped to binary tree. The first branch node, which is 16 is to be examined;



c. Step 2, 16 is the largest element in current sub tree, next is to check node with value 2;

Figure 8.3: Build a heap from an arbitrary array. Gray nodes are changed in each step, black node is the one to be processed next step.

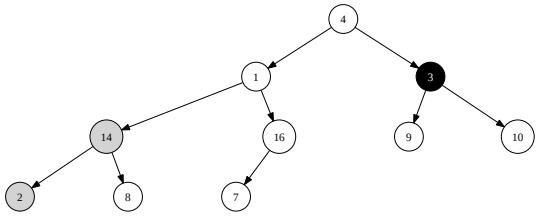
Build a heap in Python

Like the C++ heap building program, we check from the last non-leaf node and apply $HEAPIFY$ algorithm, and repeat the process back to the root node. However, we use explicit calculation (divided by 2) instead of using bit-wise shifting.

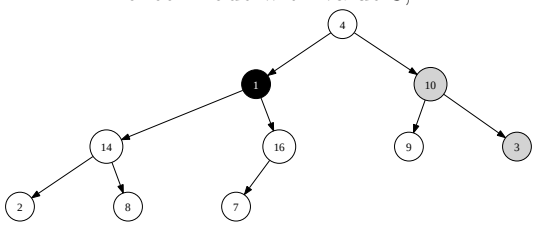
```
def build_heap(x, less_p = MIN_HEAP):
    n = len(x)
    for i in reversed(range(n//2)):
        heapify(x, i, less_p)
```

We can feed the similar test case to this program as below:

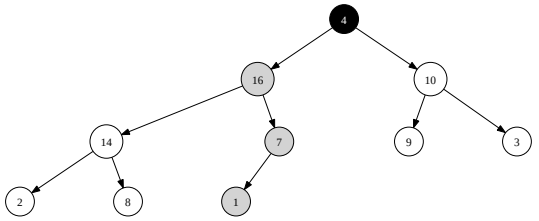
```
l = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
```



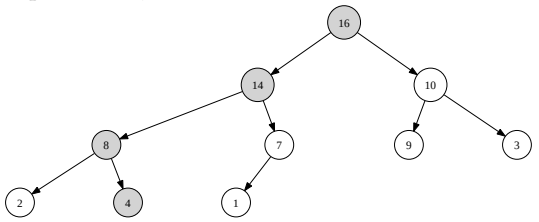
d. Step 3, 14 is the largest value in the sub-tree, swap 14 and 2; next is to check node with value 3;



e. Step 4, 10 is the largest value in the sub-tree, swap 10 and 3; next is to check node with value 1;



f. Step 5, 16 is the largest value in current node, swap 16 and 1 first; then similarly, swap 1 and 7; next is to check the root node with value 4;



g. Step 6, Swap 4 and 16, then swap 4 and 14, and then swap 4 and 8; And the whole build process finish.

Figure 8.4: Build a heap from an arbitrary array. Gray nodes are changed in each step, black node is the one to be processed next step.

```
build_heap(l, MAX_HEAP)
print l
```

It will output the same result as the C++ program.

```
[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
```

8.3.4 Basic heap operations

From the generic definition of heap (not necessarily binary heap), It's essential to provides basic operations so that user can access the data and modify it.

The most important operations included accessing the top element (find the minimum or maximum element), pop one element (the minimum one or the maximum one depends on the type of the heap) from the heap, find the top N elements, decrease a key (note this is for min-heap, and it will be increase a key for max-heap), and insertion.

For binary tree, most of this operation is bound to $O(\lg N)$ worst-case, some of them, such as top is $O(1)$ time.

Access the top element (minimum)

According to the definition of heap, there must be an operation to return the top element. for implicit binary tree by array, it is the root node which stores the minimum (maximum) value.

1: **function** TOP(A)
2: **return** $A[0]$

This operation is trivial. It takes $O(1)$ time.

Access the top element in C++

By translate the above algorithm directly in C++, we get the following program.

```
template<class T>
typename ValueType<T>::Result heap_top(T a){ return a[0]; }
```

There is a small trick to get the type of the element store in array no matter if the array is STL container or plain C like array.

```
template<class T> struct ValueType{
  typedef typename T::value_type Result;
};

template<class T> struct ValueType<T*>{
  typedef T Result; // c-pointer type
};

template<class T, unsigned int n> struct ValueType<T[n]>{
  typedef T Result; // c-array type
};
```

Not that the C++ template meta programming support to specialize for a certain type.

Here we skip the error handling of empty heap case. If the heap is empty, one option is just to raise exception.

Access the top element in Python

The python version of this program is also simple, we omit the error handling for empty heap as well.

```
def heap_top(x):
    return x[0] #ignore empty case
```

Heap Pop (delete minimum)

Different from the top operation, pop operation is a bit complex, because the heap property has to be maintained after the top element is removed.

The solution is to apply $HEAPIFY$ algorithm immediately to the next node to the root node which has been removed.

A quick but slow algorithm based on this idea may look like the following.

1: **function** POP-SLOW(A)
2: $x \leftarrow TOP(A)$
3: $REMOVE(A, 1)$
4: **if** A is not empty **then**
5: $HEAPIFY(A, 1)$
6: **return** x

This algorithm first remember the top element in x, then it removes the first element from the array, the size of this array reduced by one. After that if the array isn't empty, $HEAPIFY$ will applied to the modified array on the first element (previous the second element).

Removing an element from array takes $O(N)$ time, where N is the length of the array. Removing the first element need shift all the rest values one by one. Because of this bottle neck, it slows the whole algorithm to $O(N)$.

In order to solve this program, one alternative way is to just swap the first element and the last one in the array, then shrink the array size by one.

1: **function** POP(A)
2: $x \leftarrow TOP(A)$
3: $SWAP(A[1], A[HEAP - SIZE(A)])$
4: $REMOVE(A, HEAP - SIZE(A))$
5: **if** A is not empty **then**
6: $HEAPIFY(A, 1)$
7: **return** x

Note that remove the last element from the array takes only $O(1)$ time, and $HEAPIFY$ is bound to $O(\lg N)$. The whole algorithm is bound to $O(\lg N)$ time.

Pop in C++

In C++ program, we abstract the min-heap and max-heap as heap type template parameter, and pass it explicitly.

First is the 'reference + size' approach.

```
template<class T, class LessOp>
typename ValueType<T>::Result heap_pop(T& a, unsigned int& n, LessOp lt){
  typename ValueType<T>::Result top = heap_top(a);
  a[0] = a[n-1];
```

```
  heapify(a, 0, --n, lt);
  return top;
}
```

And it can be adapted to "range" abstraction as well.

```
template<class R, class LessOp>
typename R::value_type heap_pop(R& a, LessOp lt){
  typename R::value_type top = heap_top(a);
  std::swap(a[0], a[a.size()-1]);
  --a.last;
  heapify_(a, 0, lt);
  return top;
}
```

Pop in Python

Python provides pop() function to get rid of the last element, so the program can be developed as below.

```
def heap_pop(x, less_p = MIN_HEAP):
    top = heap_top(x)
    x[0] = x[-1] # this is faster than top = x.pop(0)
    x.pop()
    if x!=[]:
        heapify(x, 0, less_p)
    return top
```

Find the first K biggest (smallest) element

With pop operation, it is easy to implement algorithm to find the top K elements. In order to find the biggest K values form an array, we can build a max-heap, then perform pop operation K times.

1: **function** TOP-K(A, k)
2: $BUILD - HEAP(A)$
3: **for** $i \leftarrow 1, MIN(k, LENGTH(A))$ **do**
4: $APPEND(Result, POP(A))$
5: **return** *Result*

Note that if K is bigger than the length of the array, it means we need return the whole array as the result. That's why it need use the MIN function in the algorithm.

Find the first K biggest (smallest) element in C++

In C++ program, we can pass the iterator for output, so the "reference - size" version looks like below.

```
template<class Iter, class Array, class LessOp>
void heap_top_k(Iter res, unsigned int k,
                Array& a, unsigned int& n, LessOp lt){
  build_heap(a, n, lt);
  unsigned int count = std::min(k, n);
  for(unsigned int i=0; i<count; ++i)
```

```
    *res++=heap_pop(a, n, lt);
}
```

When we adapt to 'range' concept, it is possible to manipulate the data in place, so that we can put the top K element in first K positions in the array.

```
template<class R, class LessOp>
void heap_top_k(R a, typename R::size_t k, LessOp lt){
  typename R::size_t count = std::min(k, a.size());
  build_heap(a, lt);
  while(count--){
    ++a.first;
    heapify(a, 0, lt);
  }
}
```

The algorithm doesn't utilize 'pop' function, instead, after the heap is built, the first element is the top one, it adjusts the range one position next, then apply heapify to the new range. This process is repeated for K times so the first K elements are the result.

A simple test cases can be fed to the program for verification.

```
const int a[] = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7};
unsigned int n = sizeof(a)/sizeof(a[0]);
std::vector<int> x(a, a+n);
heap_top_k(range(x.begin(), x.end()), 3, MaxHeap<int>());
print_range(range(x.begin(), 3));
```

The result is printed in console like below.

```
16, 14, 10,
```

Find the first K biggest (smallest) element in Python

In Python we can put 'pop' function to list comprehension to get the top K elements like the following.

```
def top_k(x, k, less_p = MIN_HEAP):
    build_heap(x, less_p)
    return [heap_pop(x, less_p) for i in range(min(k, len(x)))]
```

The testing and result are shown as the following.

```
l = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
res = top_k(l, 3, MAX_HEAP)
print res
```

Evaluate the code led to below line.

```
[16, 14, 10]
```

Modification: Decrease key

Heap can be used to implement priority queue, because of this, it is important to modify the key stored in heap. One typical operation is to increase the priority of a tasks so that it can be performed earlier.

Here we present the decrease key operation for a min-heap. The corresponding operation is increase key for max-heap.

Once we modified a key by decreasing it in a min-heap, it can make the node conflict with the heap property, that the key may be less than some values in its ancestors. In order to maintain the invariant, an auxiliary algorithm is provided to fix the heap property.

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: $i \leftarrow PARENT[i]$

This algorithm repeatedly examine the key of parent node and the key in current node. It will swap nodes in case the parent contains the smaller key. This process is performed from current node towards the root node till it find that the parent node holds the smaller key.

With this auxiliary algorithm, decrease key can be realized easily.

1: **function** DECREASE-KEY(A, i, k)
2: **if** $k < A[i]$ **then**
3: $A[i] \leftarrow k$
4: $HEAP - FIX(A, i)$

Note that the algorithm only takes effect when the new key is less than the original key.

Decrease key in C++

In order to support both min-heap and max-heap, the comparison function object is passed as an argument in C++ implementation.

```
template<class Array, class LessOp>
void heap_fix(Array& a, unsigned int i, LessOp lt){
  while(i>0 && lt(a[i], a[parent(i)])){
    std::swap(a[i], a[parent(i)]);
    i = parent(i);
  }
}

template<class Array, class LessOp>
void heap_decrease_key(Array& a,
                       unsigned int i,
                       typename ValueType<Array>::Result key,
                       LessOp lt){
  if(lt(key, a[i])){
    a[i] = key;
    heap_fix(a, i, lt);
  }
}
```

Some very simple verification case can be fed to the program. Here we use the example presented in [3] Figure 6.5.

```
const int a[] = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1};
const unsigned int n = sizeof(a)/sizeof(a[0]);
int x[n];
```

```
std::copy(a, a+n, x);
heap_decrease_key(x, 8, 15, MaxHeap<int>());
print_range(x, x+n);
```

Run the above lines will generate the following output.

```
16, 15, 10, 14, 7, 9, 3, 2, 8, 1,
```

In this max-heap example, we try to increase the key of the 9-th node from 4 to 15. As shown in figure 8.5

Decrease key in Python

The Python version decrease key program is similar as well. It first checks if the new key is "less than" the original one, if yes it modifies the value, and performs the fixing process.

```
def heap_decrease_key(x, i, key, less_p = MIN_HEAP):
    if less_p(key, x[i]):
        x[i] = key
        heap_fix(x, i, less_p)

def heap_fix(x, i, less_p = MIN_HEAP):
    while i>0 and less_p(x[i],x[parent(i)]):
        (x[parent(i)], x[i]) = (x[i], x[parent(i)])
        i = parent(i)
```

We can use the same test case to verify the program.

```
l = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
heap_decrease_key(l, 8, 15, MAX_HEAP)
print l
```

It will output the same result as the C++ program.

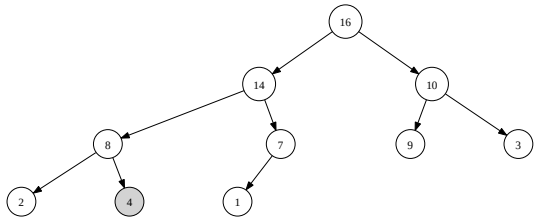
```
[16, 15, 10, 14, 7, 9, 3, 2, 8, 1]
```

Insertion

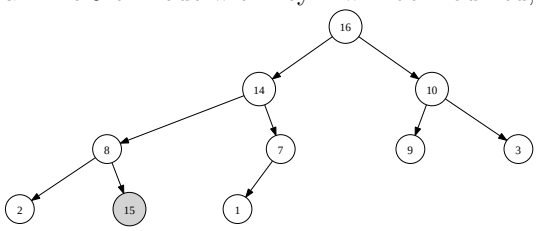
In [3], insertion is implemented by using $DECREASE - KEY$. The approach is to first insert a node with infinity key. According to the min-heap property, the node should be the last element in the under ground array. After that, the key is decreased to the value to be inserted, so that we can call decrease-key to finish the process.

Instead of reuse $DECREASE - KEY$, we can reuse $HEAP - FIX$ to implement insertion. The new key is directly appended at the end of the array, and the $HEAP - FIX$ is applied on this new node.

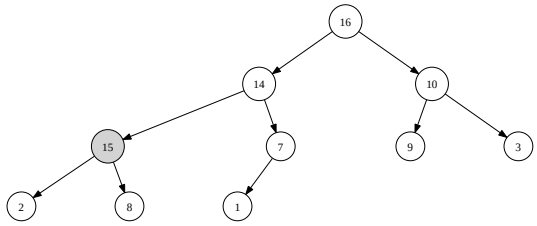
1: **function** HEAP-PUSH(A, k)
2: $APPEND(A, k)$
3: $HEAP - FIX(A, SIZE(A))$



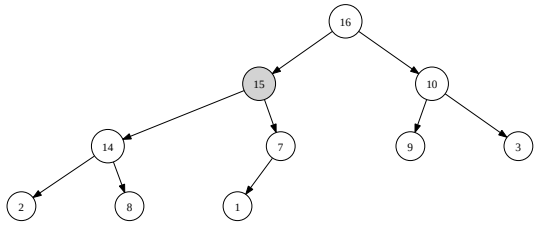
a. The 9-th node with key 4 will be modified;



b. The key is modified to 15, which is greater than its parent;



c. According the max-heap property, 8 and 15 are swapped.



d. Since 15 is greater than 14, which is the key of its parent node, 15 and 14 are swapped. Because 15 is less than 16, the algorithm terminates.

Figure 8.5: Example process when increase a key in a max-heap.

Insertion by decreasing key method in C++

In C++, traditional C array is static sized, so appending a new element to the array has to be managed properly. In order to simplify the problem, we assume the necessary has already been allocated (by client program).

First is the "reference + size" version of program.

```
template<class Array, class LessOp>
void heap_push(Array& a,
               unsigned int& n,
               typename ValueType<Array>::Result key,
               LessOp lt){
  a[n] = key;
  heap_fix(a, n, lt);
  ++n;
}
```

Note that the size is explicitly increased. so after calling this function, the count of the element is increased by one.

It is also possible to provide equivalent program by using 'range'.

```
template<class R, class LessOp>
void heap_push(R a, typename R::value_type key, LessOp lt){
  *a.last++ = key;
  heap_fix(a, a.size()-1, lt);
}
```

We can test the insert program with a very simple case.

```
const int a[] = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1};
unsigned int n = sizeof(a)/sizeof(a[0]);
std::vector<int> x(a, a+n);
x.push_back(0);
heap_push(range(x.begin(), n), 17, MaxHeap<int>());
print_range(x.begin(), x.end());
```

Note that, in client program (this test program), we reserved the memory in advance, or it will cause access violation problem. Running these lines will output the following result.

```
17, 16, 10, 8, 14, 9, 3, 2, 4, 1, 7,
```

We can found the new element 17 is inserted at the proper position of the heap.

Insertion directly in Python

In python program, append a new element to a list is build-in supported. So client program doesn't need to take care of the similar problem as described in C++ implementation.

```
def heap_insert(x, key, less_p = MIN_HEAP):
    i = len(x)
    x.append(key)
    heap_fix(x, i, less_p)
```

If the same test case is fed to the above function, we can get the output like the following.

```
l = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
heap_insert(l, 17, MAX_HEAP)
print l
```

```
[17, 16, 10, 8, 14, 9, 3, 2, 4, 1, 7
```

8.3.5 Heap sort

Heap sort algorithm is an interesting application of heap. According to the heap property, the min(max) element can be easily accessed by from the top of the heap. So a straightforward way to sort an arbitrary of values is to first build a heap from them, then continuously pops the smallest element from the heap till the heap is empty.

The algorithm based on this strategy is something like below.

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

Robert. W. Floyd found a very fast implementation of heap sort. The idea is to build a max-heap instead of min-heap, so the first element is the biggest one. Then this biggest element is swapped with the last element in the array, so that it is in the right position after sorting. Now the last element becomes the top of the heap, it may violate the heap property. We can perform $HEAPIFY$ on it with the heap size shrink by one. This process is repeated till there is only one element left in the heap.

1: **function** HEAP-SORT-FAST(A)
2: $BUILD - MAX - HEAP(A)$
3: **while** $SIZE(A) > 1$ **do**
4: Exchange $A[1] \leftrightarrow A[SIZE(A)]$
5: $SIZE(A) \leftarrow SIZE(A) - 1$
6: $HEAPIFY(A, 1)$

Note that this algorithm is in-place algorithm. It's the fastest heap sort algorithm by far.

In terms of complexity, $BUILD-HEAP$ is bound to $O(N)$. Since $HEAPIFY$ is $O(\lg N)$, and it is called $O(N)$ times, so both above algorithms take $O(N \lg N)$ time to run.

Floyd's heap sort algorithm in C++

Only Floyd's algorithm is given in C++ in this post.

```
template<class Array, class GreaterOp>
void heap_sort(Array& a, unsigned int n, GreaterOp gt){
  for(build_heap(a, n, gt); n>1; --n){
    std::swap(a[0], a[n-1]);
    heapify(a, 0, n-1, gt);
  }
}
```

A very simple test case is used for verification.

```
const int a[] = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1};
std::vector<int> y(a, a+n);
heap_sort(range(y.begin(), y.end()), MaxHeap<int>());
print_range(y.begin(), y.end());
```

The result is output as we expect.

```
1, 2, 3, 4, 7, 8, 9, 10, 14, 16,
```

General heap sort algorithm in Python

Instead of Floyd method, we'll show the straightforward 'popping N times' algorithm in Python. Please refer to [5] for Floyd algorithm in Python.

```
def heap_sort(x, less_p = MIN_HEAP):
    res = []
    build_heap(x, less_p)
    while x!=[]:
        res.append(heap_pop(x, less_p))
    return res
```

And the test case is as same as the one we used in C++ program.

```
l = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
res = heap_sort(l)
print res
```

The result is output as the following.

```
[1, 2, 3, 4, 7, 8, 9, 10, 14, 16]
```

8.4 Leftist heap and Skew heap, explicit binary heaps

Instead of using implicit binary tree by array, it is natural to consider why we can't use explicit binary tree to realize heap?

There are some problems must be solved if we turn into explicit binary tree as the under ground data strucutre for heap.

The first problem is about the $HEAP - POP$ or $DELETE - MIN$ operation. If the explicit binary tree is represent as the form of (left value right), which is shown in figure 8.6

If k is the top element, all values in left and right children are less than k. After k is popped, only left and right children are left. They have to be merged to a new tree. Since heap property should be maintained after merge, so the new root element is the smallest one.

Since both left child and right child are heaps in binary tree, the two trivial cases can be found immediately.

1: **function** MERGE(L, R)
2: **if** $L = NIL$ **then**
3: **return** R
4: **else if** $R = NIL$ **then**

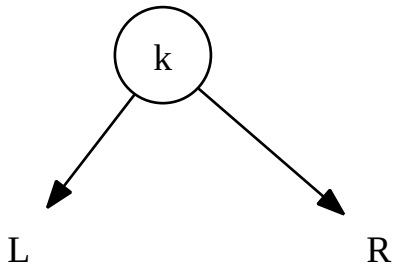


Figure 8.6: A binary tree, all values in left child and right child are smaller than k.

```
5:          return L
6:      else
7:          ...
```

If neither left child nor right child is empty tree, because they all fit heap property, the top element of them are all minimum value respectively. One solution is to compare the root value of the left and right children, select the smaller one as the new root of the merged heap, and recursively merget the other child to one of the children of the smaller one. For instance if $L = (AxB)$ and $R = (A'yB')$, where A, A', B, B' are all sub trees, and $x < y$. There are two candidate results according to this strategy.

- $(MERGE(A, R)xB)$
- $(AxMERGE(B, R))$

Both are correct result. One simplified solution is only merge on right sub tree. Leftist tree provides a systematically approach based on this idea.

8.4.1 Definition

The heap implemented by Leftist tree is called Leftist heap. Leftist tree is first introduced by C. A. Crane in 1972[6].

Rank (S-value)

In Leftist tree, a rank value (or S value) is defined to each node. Rank is the distance to the nearest external node. Where external node is a NIL concept extended from leaf node.

For example, in figure 8.7, the rank of NIL is defined 0, consider the root node 4, The nearest leaf node is the child of node 8. So the rank of root node

4 is 2. Because node 6 and node 8 are all only contain NIL, so the rank values are 1. Although node 5 has non-NIL left child, However, since the right child is NIL, so the rank value, which is the minimum distance to leaf is still 1.

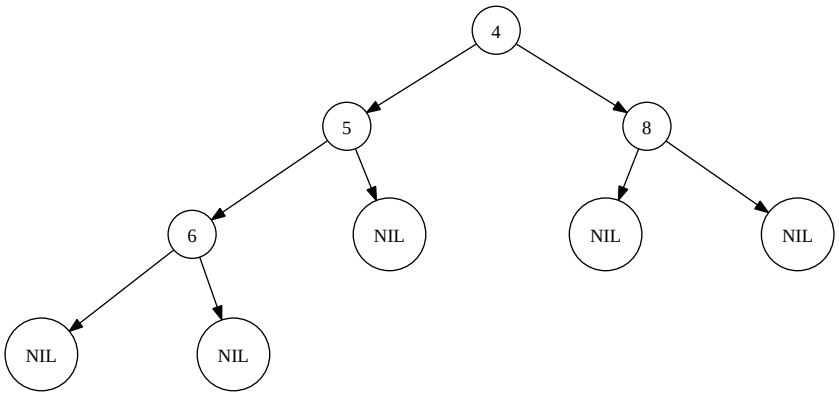


Figure 8.7: rank(4) = 2, rank(6) = rank(8) = rank(5) = 1.

Leftist property

With rank defined, we can create a strategy when merging.

- Every time when merging, we always merge to right child; Denote the rank of the new right sub tree as r_r;
- Compare the ranks of the left and right children, if the rank of left sub tree is r_l and $r_l < r_r$, we swap the left and right children.

We call this 'Leftist property'. Generally speaking, a Leftist tree always has the shortest path to an external node on the right.

A Leftist tree tends to be very unbalanced, However, it ensures an important property as specified in the following theorem.

If a Leftist tree T contains N internal nodes, the path from root to the rightmost external node contains at most $\lfloor \log(N+1) \rfloor$ nodes.

For the proof of these theorem, please refer to [7] and [1].

With this theorem, algorithms operate along this path are all bound to $O(\lg N)$.

Definition of Leftist heap in Haskell

In Haskell the definition of Leftist tree is almost as same as the binary search tree except there is a rank field added.

```
data LHeap a = E -- Empty
             | Node Int a (LHeap a) (LHeap a) -- rank, element, left, right
               deriving (Eq, Show)
```

In order to access the rank field, a helper function is provided.

```
rank::LHeap a → Int
rank E = 0
rank (Node r _ _ _) = r
```

Definition of Leftist heap in Scheme/Lisp

In Scheme/Lisp, list is used to represent the Leftist tree. In order to output the tree easily in in-order format, a node is arranged as (left rank element right). Some auxiliary functions are defined to access these fields of a node.

```
(define (left t)
  (if (null? t) '() (car t)))

(define (rank t)
  (if (null? t) 0 (cadr t)))

(define (elem t)
  (if (null? t) '() (caddr t)))

(define (right t)
  (if (null? t) '() (cadddr t)))
```

And a construction function is provided so that a node can be built explicitly.

```
(define (make-tree l s x r) ;; l: left, s: rank, x: elem, r: right
  (list l s x r))
```

8.4.2 Merge

In order to realize 'merge', an auxiliary algorithm is given to compare the ranks and do swapping if necessary.

1: **function** LEFTIFY(T)
2: $l \leftarrow LEFT(T), r \leftarrow RIGHT(T)$
3: $k \leftarrow KEY(T)$
4: **if** $RANK(l) < RANK(r)$ **then**
5: $RANK(T) \leftarrow RANK(L) + 1$
6: **else**
7: $RANK(T) \leftarrow RANK(R) + 1$
8: Exchange $l \leftrightarrow r$

The algorithm compares the rank of the left and right sub trees, pick the less one and add it by one as the rank of the modified node. If the rank of left side is greater, it will also swap the left and right children.

The reason why rank need to be increased by one is because there is a new key added on top of the tree, which causes the rank increase.

With $LEFTIFY$ defined, merge algorithm can be provided as the following.

1: **function** MERGE(L, R)
2: **if** $L = NIL$ **then**
3: **return** R
4: **else if** $R = NIL$ **then**
5: **return** L

```
6:    else
7:        T ← CREATE − NEW − NODE()
8:        if KEY(L) < KEY(R) then
9:            KEY(T) ← KEY(L)
10:           LEFT(T) ← LEFT(L)
11:           RIGHT(T) ← MERGE(RIGHT(L), R)
12:       else
13:           KEY(T) ← KEY(R)
14:           LEFT(T) ← LEFT(R)
15:           RIGHT(T) ← MERGE(L, RIGHT(R))
16:       LEFTIFY(T)
17:   return T
```

Note that $MERGE$ algorithm always operate on right side, and call $LEFITFY$ to ensure the 'Leftist property, so that this algorithm is bound to $O(\lg N)$.

Merge in Haskell

Translate the algorithm to Haskell lead to the following program. Here we modified the pseudo code to a pure functional style.

```
merge::(Ord a)⇒LHeap a → LHeap a → LHeap a
merge E h = h
merge h E = h
merge h1@(Node _ x l r) h2@(Node _ y l' r') =
    if x < y then makeNode x l (merge r h2)
    else makeNode y l' (merge h1 r')

makeNode::a → LHeap a → LHeap a → LHeap a
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 in Scheme/Lisp

In Scheme/Lisp, the $LEFTIFY$ algorithm can be defined as an inner function inside $MERGE$.

```
(define (merge t1 t2)
  (define (make-node x a b)
    (if (< (rank a) (rank b))
        (make-tree b (+ (rank a) 1) x a)
        (make-tree a (+ (rank b) 1) x b)))
  (cond ((null? t1) t2)
        ((null? t2) t1)
        ((< (elem t1) (elem t2)) (make-node (elem t1) (left t1) (merge (right t1) t2)))
        (else (make-node (elem t2) (left t2) (merge t1 (right t2))))))
```

Merge operation in implicit binary heap by array

In most cases implicit binary heap by array performs very fast, and it fits modern computer with cache technology well. However, merge is the algorithm bounds to $O(N)$ time. The best you can do is concatenate two arrays together and make a heap of the result [13].

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

We omit the implementation of this algorithm in C++ and Python because they are trivial.

8.4.3 Basic heap operations

Most of the basic heap operations can be implemented easily with $MERGE$ algorithm define above.

Find minimum (top) and delete minimum (pop)

Since we keep the smallest element in root node, finding the minimum value (top element) is trivial. It's a $O(1)$ operation.

1: **function** TOP(T)
2: **return** $KEY(T)$

While if the top element popped, left and right children are merged so the heap updated.

1: **function** POP(T)
2: **return** $MERGE(LEFT(T), RIGHT(T))$

Note that pop operation on Leftist heap takes $O(\lg N)$ time.

Find minimum (top) and delete minimum in Haskell

We skip the error handling of operation on an empty Leftist heap.

```
findMin :: LHeap a → a
findMin (Node _ x _ _) = x

deleteMin :: (Ord a) ⇒ LHeap a → LHeap a
deleteMin (Node _ _ l r) = merge l r
```

Find minimum (top) and delete minimum in Scheme/Lisp

With merge function defined, these operations are trivial to implement in Scheme/Lisp.

```
(define (find-min t)
  (elem t))

(define (delete-min t)
  (merge (left t) (right t)))
```

Insertion

To insert a new key to the heap, one solution is to create a single leaf node from the key, and perform merge with this leaf node and the Leftist tree.

1: **function** INSERT(T, k)
2: $x \leftarrow CREATE - NEW - NODE()$
3: $KEY(x) \leftarrow k$
4: $RAKN(x) \leftarrow 1$
5: $LEFT(x), RIGHT(x) \leftarrow NIL$

6: **return** $MERGE(x, T)$

Since insert still call merge inside, the algorithm is bound to $O(\lg N)$ time.

Insertion in Haskell

Translating the above algorithm to Haskell is trivial.

```
insert::(Ord a)⇒ LHeap a → a → LHeap a
insert h x = merge (Node 1 x E E) h
```

In order to provide a convenient way to build a Leftist heap from a list, an auxiliary function is given as the following.

```
fromList :: (Ord a) ⇒ [a] → LHeap a
fromList = foldl insert E
```

This function can be used like this.

```
fromList [9, 4, 16, 7, 10, 2, 14, 3, 8, 1]
```

It will create a Leftist heap as below.

```
Node 1 1 (Node 3 2 (Node 2 4 (Node 2 7 (Node 1 16 E E)
(Node 1 10 E E)) (Node 1 9 E E)) (Node 2 3 (Node 1 14 E E)
(Node 1 8 E E))) E
```

Figure 8.8 shows the result respectively.

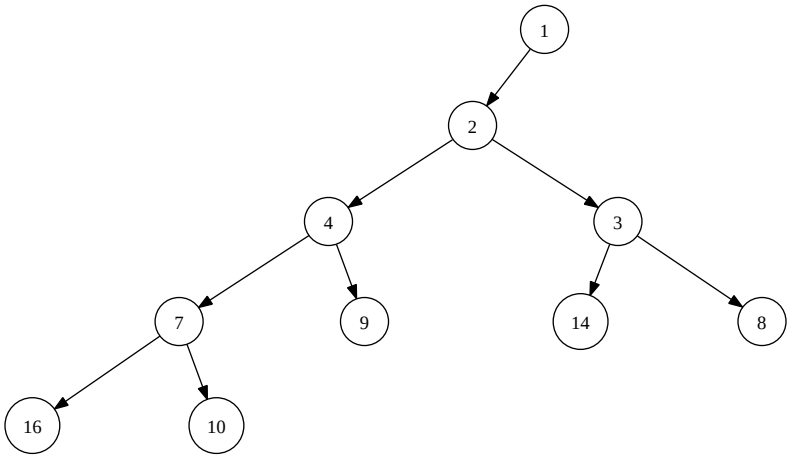


Figure 8.8: A Leftist tree.

Insertion in Scheme/Lisp

Based on the algorithm, once inserting a new element, a leaf node is created and merge to the heap.

```
(define (insert t x)
  (merge (make-tree '() 1 x '()) t))
```

This function can be verified by continuously insert all elements from a list so that a Leftist heap can be built as a result.

```
(define (from-list lst)
  (fold-left insert '() lst))
```

One example test case which is equivalent to the Haskell program is shown like the following.

```
(define (test-from-list)
  (from-list '(16 14 10 8 7 9 3 2 4 1)))
```

Evaluate this function yields a Leftist tree, which can be output in in-order as below.

```
(((((((() 1 16 ()) 1 14 ()) 1 10 ()) 1 8 ()) 2 7 (() 1 9 ())) 1 3
()) 2 2 (() 1 4 ())) 1 1 ())
```

8.4.4 Heap sort by Leftist Heap

With all the basic operations defined, it's straightforward to implement heap sort in a N-popping way. Once we need sort a list of elements, we first build a Leftist heap from the list. Then repeatedly pop the minimum element from the heap until heap is empty.

First is the algorithm to build the Leftist heap by insert all elements to an empty heap.

1: **function** BUILD-HEAP(A)
2: $H \leftarrow NIL$
3: **for** each x in A **do**
4: $T \leftarrow INSERT(T, x)$
5: **return** T

And the heap sort algorithm is as same as the generic one presented in 8.3.5.

Note this algorithm is bound to $O(N \lg N)$ time.

Heap sort in Haskell

In Haskell program, since we have already defined the 'fromList' auxiliary function to build Leftist heap from a list, the heap sort algorithm can utilize it.

```
heapSort :: (Ord a) ⇒ [a] → [a]
heapSort = hsort ∘ fromList where
    hsort E = []
    hsort h = (findMin h):(hsort $ deleteMin h)
```

Here is an example case which is used in previous C++ and Python programs.

```
heapSort [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
```

It will output the same result as the following.

```
[1,2,3,4,7,8,9,10,14,16]
```

Heap sort in Scheme/Lisp

In Scheme/Lisp program, we can first use 'from-list' function to turn a list of element into a Leftist heap, then repeatedly pop the smallest one to a result list.

```
(define (heap-sort lst)
  (define (hsort t)
    (if (null? t) '() (cons (find-min t) (hsort (delete-min t)))))
  (hsort (from-list lst)))
```

Here is a simple test case.

```
heap-sort '(16 14 10 8 7 9 3 2 4 1))
```

Evaluate the case will out put an ordered list.

```
(1 2 3 4 7 8 9 10 14 16)
```

8.4.5 Skew heaps

The problem with Leftist heap is that, it performs bad in some cases. For example, if we examine the Leftist heap behind the above heap sort test case, it's a very unbalanced binary tree as shown in figure 8.9[1].

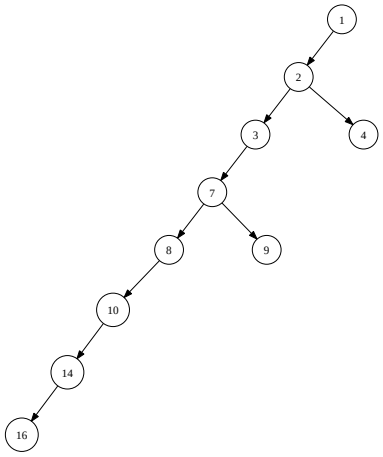


Figure 8.9: A very unbalanced Leftist tree build from list [16, 14, 10, 8, 7, 9, 3, 2, 4, 1].

The binary tree is almost turned to be a linked-list. The worst case is when feed a ordered list for building Leftist tree, since the tree changed to linked-list, the time bound degrade from $O(\lg N)$ to $O(N)$.

Skew heap (or *self-adjusting heap*) is one step ahead simplified Leftist heap [9] [10].

[1] run Haskell Leftist tree function: fromList [16, 14, 10, 8, 7, 9, 3, 2, 4, 1] will generates the result: Node 1 1 (Node 2 2 (Node 1 3 (Node 2 7 (Node 1 8 (Node 1 10 (Node 1 14 (Node 1 16 E E) E) E) E) (Node 1 9 E E)) E) (Node 1 4 E E)) E

Remind the Leftist heap, we swap the left and right children during merge when the rank on left side is less than right side. This comparison strategy doesn't work when one of the sub tree has only one child. Because in such case, the rank of the sub tree is always 1 no matter how big it is. A Brute-force approach is to swap the left and right children every time when merge. This idea leads to Skew heap.

Definition of Skew heap

A Skew heap is a heap implemented with Skew tree. A Skew tree is a special binary tree. The minimum element is stored in root node. Every sub tree is also a skew tree.

Based on above discussion, there is no use to keep the rank (or S-value) field, so the Skew heap definition is as same as the binary tree from the programming language point of view.

Definition of Skew heap in Haskell

After removing rank from Leftist heap definition, we can get the Skew heap one.

```
data SHeap a = E -- Empty
             | Node a (SHeap a) (SHeap a) -- element, left, right
               deriving (Eq, Show)
```

Definition of Skew heap in Scheme/Lisp

Since Skew heap is just a special kind of binary tree, the definition is as same as binary tree's. In Scheme/Lisp, the inner data structure is list. We organize it in in-order for easy output purpose.

Some auxiliary access functions and a simple constructor is provided.

```
(define (left t)
  (if (null? t) '() (car t)))

(define (elem t)
  (if (null? t) '() (cadr t)))

(define (right t)
  (if (null? t) '() (caddr t)))

;; constructor
(define (make-tree l x r) ;; l: left, x: element, r: right
  (list l x r))
```

Merge

The merge algorithm tends to be very simple. When we merge two Skew trees, we compare the root element of each tree, and pick the smaller one as the new root, we then merge the other tree contains bigger element onto the right sub tree and swap the left and right children.

1: **function** MERGE(L, R)
2: **if** $L = NIL$ **then**

```
3:        return R
4:    else if R = NIL then
5:        return L
6:    else
7:        T ← CREATE − EMPTY − NODE()
8:        if KEY(L) < KEY(R) then
9:            KEY(T) ← KEY(L)
10:           LEFT(T) ← MERGE(R, RIGHT(L))
11:           RIGHT(T) ← LEFT(L)
12:       else
13:           KEY(T) ← KEY(R)
14:           LEFT(T) ← MERGE(L, RIGHT(R))
15:           RIGHT(T) ← LEFT(R)
16:       return T
```

Skew heap in Haskell

Translating the above algorithm into Haskell gets a simple merge program.

```
merge::(Ord a)⇒SHeap a → SHeap a → SHeap a
merge E h = h
merge h E = h
merge h1@(Node x l r) h2@(Node y l' r') =
    if x < y then Node x (merge r h2) l
    else Node y (merge h1 r') l'
```

All the rest programs are as same as the Leftist heap except we needn't provide rank value when construct a node.

```
insert::(Ord a)⇒ SHeap a → a → SHeap a
insert h x = merge (Node x E E) h

findMin :: SHeap a → a
findMin (Node x _ _) = x

deleteMin :: (Ord a) ⇒ SHeap a → SHeap a
deleteMin (Node _ l r) = merge l r
```

If we feed a completely ordered list to Skew heap, it will results a fairly balanced binary trees as shown in figure 8.10.

```
*SkewHeap>fromList [1..10]
Node 1 (Node 2 (Node 6 (Node 10 E E) E) (Node 4 (Node 8 E E) E))
(Node 3 (Node 5 (Node 9 E E) E) (Node 7 E E))
```

Skew heap in Scheme/Lisp

In merge program, if any one of the tree to be merged, it just returns the other one. For non-trivial case, the program select the smaller one as the new root element, merge the tree contains the bigger element to the right child, then swap the two children.

```
(define (merge t1 t2)
  (cond ((null? t1) t2)
```

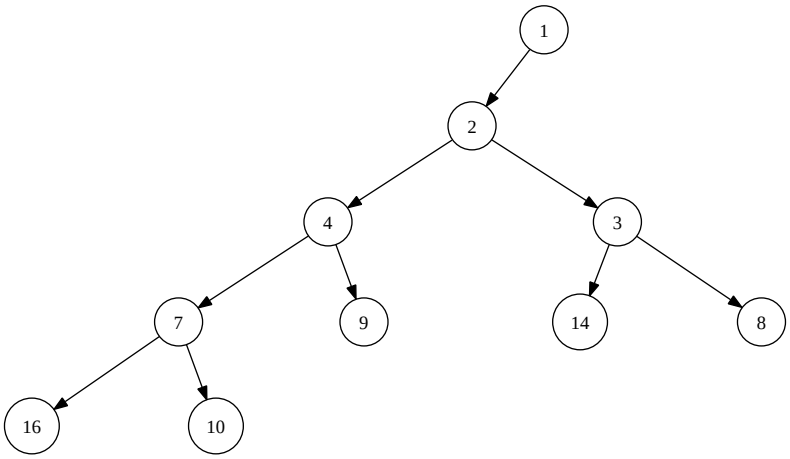


Figure 8.10: Skew tree is still balanced even the input is an ordered list.

```
((null? t2) t1)
((< (elem t1) (elem t2))
 (make-tree (merge (right t1) t2)
            (elem t1)
            (left t1)))
(else
 (make-tree (merge t1 (right t2))
            (elem t2)
            (left t2)))))
```

With merge function defined, insert can be treated as just a special merge case, that one tree is a leaf which contains the value to be inserted.

```
(define (insert t x)
  (merge (make-tree '() x '()) t))
```

The find minimum, delete minimum and heap sort functions are all as same as the leftist heap programs, that they can be put to a generic module.

We only show the testing result in this section.

```
(load "skewheap.scm")
;Loading "skewheap.scm"... done
;Value: insert

(test-from-list)
;Value 13: ((((() 4 ()) 2 ((((() 9 ()) 7 (((((() 16 ()) 14 ()) 10 ())
8 ())) 3 ())) 1 ())

(test-sort)
;Value 14: (1 2 3 4 7 8 9 10 14 16)
```

8.5 Splay heap, another explicit binary heap

Leftist heap presents that it's quite possible to implement heap data structure with explicit binary tree. Skew heap shows one method to solve the balance problem. Splay heap on the other hand, shows another balance approach.

Although Leftist heap and Skew heap use binary trees, they are not Binary Search tree (BST). If we turn the underground data structure to binary search tree, the minimum(maximum) element isn't located in root node. It takes $O(\lg N)$ time to find the minimum(maximum) element.

Binary search tree becomes inefficient if it isn't well balanced, operations degrades to $O(N)$ in the worst case. Although it's quite OK to use red-black tree to implement binary heap, Splay tree provides a light weight implementation with acceptable dynamic balancing result.

8.5.1 Definition

Splay tree uses a cache-like approach that it keeps rotating the current access node close to the top, so that the node can be accessed fast next time. It defines such kinds of operation as "Splay". For an unbalanced binary search tree, after several times of splay operation, the tree tends to be more and more balanced. Most basic operation of Splay tree have amortized $O(\lg N)$ time. Splay tree was invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985[11] [12].

Splaying

There are two kinds of splaying approach method. The first one is a bit complex, However, it can be implemented fairly simple with pattern matching. The second one is simple, but the implementation is a bit complex.

Denote the node is currently accessed as X, it's parent node as P, and it's grand parent node (if has) as G. There are 3 steps for splaying. Each step contains 2 symmetric cases. For illustration purpose, only one case is shown for each step.

- *Zig-zig step.* As shown in figure 8.11, in this case, both X, and its parent P are either left children or right children. By rotating 2 times, X becomes the new root.
- *Zig-zag step.* As shown in figure 8.12, in this case, X is the right child of its parent while P is the left child. Or X is the left child of P, and P is the right child of G. After rotation, X becomes the new root, P and G become siblings.
- *Zig step.* As shown in figure 8.13, in this case, P is the root, we perform one rotate, so that X becomes new root. Note this is the last step in splay operation.

Okasaki found a simple rule for Splaying [6], that every time we follow two left branches in a row, or two right branches in a row, we rotate those two nodes.

Based on this rule, the Splaying can be realized in such a way. When we access node for a key x (can be during the process of inserting a node, or looking up a node, or deleting a node), if we found we traverse two left branches (or two

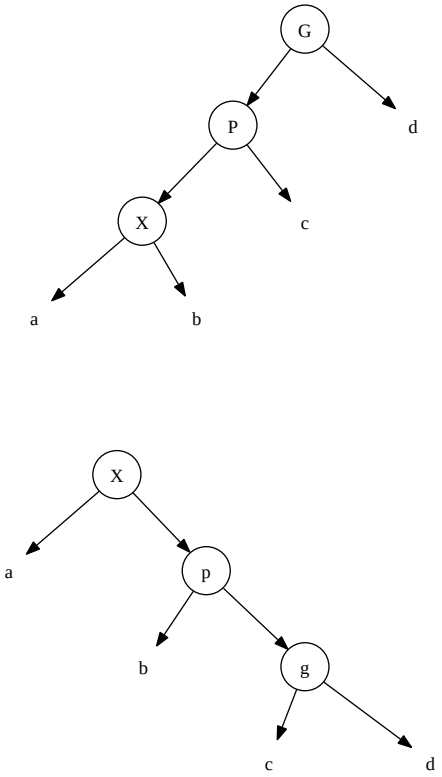


Figure 8.11: Zig-zig case.

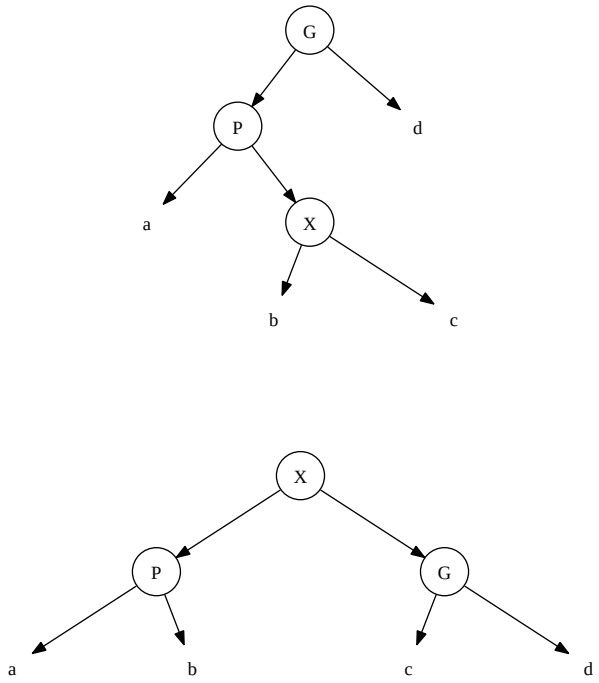


Figure 8.12: Zig-zag case.

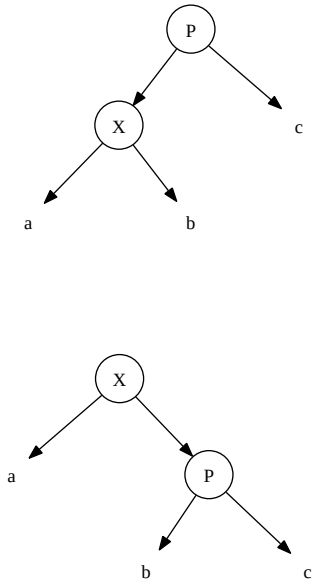


Figure 8.13: Zig case.

right branches), we partition the tree in two part L and R, where L contains all nodes smaller than x, and R contains all nodes bigger than x. We can then create a new tree (for instance in insertion), with x as the root, L as the left child and R is the right child. Note the partition process is recursive, because it will do splaying inside.

```
1: function PARTITION(T, pivot)
2:     if T = NIL then
3:         return (NIL, NIL)
4:     x ← KEY(T)
5:     l ← LEFT(T)
6:     r ← RIGHT(T)
7:     L ← NIL
8:     R ← NIL
9:     if x < pivot then
10:        if r = NIL then
11:            L ← T
12:        else
13:            x' ← KEY(r)
14:            l' ← LEFT(r)
15:            r' ← RIGHT(r)
16:            if x' < pivot then
17:                (small, big) ← PARTITION(r', pivot)
18:                L ← CREATE−NODE(CREATE−NODE(l, x, r), x', small)
```

```
19:                 R ← big
20:             else
21:                 (small, big) ← PARTITION(l', pivot)
22:                 L ← CREATE − NODE(l, x, small)
23:                 R ← CREATE − NODE(big, x', r')
24:     else
25:         if l = NIL then
26:             L ← NIL
27:         else
28:             x' ← KEY(L)
29:             l' ← LEFT(L)
30:             r' ← RIGHT(L)
31:             if x' > pivot then
32:                 (small, big) ← PARTITION(l', pivot)
33:                 L ← small
34:                 R ← CREATE−NODE(l', x', CREATE−NODE(r', x, r))
35:             else
36:                 (small, big) ← PARTITION(r', pivot)
37:                 L ← CREATE − NODE(l', x, small)
38:                 R ← CREATE − NODE(big, x, r)
39:     return (L, R)
40: function CREATE-NODE(l, x, r)
41:     T ← CREATE − NEW − NODE()
42:     KEY(T) ← x
43:     LEFT(T) ← l
44:     RIGHT(T) ← r
45:     return T
```

Definition of Splay heap in Haskell

Since Splay tree is a special binary search tree, the definition of them are same.

```
data STree a = E -- Empty
             | Node (STree a) a (STree a) -- left, element, right
               deriving (Eq, Show)
```

Translate the above algorithm into Haskell gets the following partition program.

```
partition :: (Ord a) ⇒ STree a → a → (STree a, STree a)
partition E _ = (E, E)
partition t@(Node l x r) y
    | x < y =
        case r of
          E → (t, E)
          Node l' x' r' →
              if x' < y then
                  let (small, big) = partition r' y in
                  (Node (Node l x l') x' small, big)
              else
                  let (small, big) = partition l' y in
                  (Node l x small, Node big x' r')
```

```
    | otherwise =
        case l of
          E → (E, t)
          Node l' x' r' →
              if y < x' then
                  let (small, big) = partition l' y in
                  (small, Node l' x' (Node r' x r))
              else
                  let (small, big) = partition r' y in
                  (Node l' x' small, Node big x r)
```

In a language which supports 'pattern matching' such as Haskell, Splay can be implemented in a very simple and straightforward style by translating the figure 8.11, 8.12 and 8.13 directly into patterns. Note that for each step, there are two left-right symmetric cases.

```
-- splay by pattern matching
splay :: (Eq a) ⇒ STree a → a →STree a
-- zig-zig
splay t@(Node (Node (Node a x b) p c) g d) y =
    if x == y then Node a x (Node b p (Node c g d)) else t
splay t@(Node a g (Node b p (Node c x d))) y =
    if x == y then Node (Node (Node a g b) p c) x d else t
-- zig-zag
splay t@(Node (Node a p (Node b x c)) g d) y =
    if x == y then Node (Node a p b) x (Node c g d) else t
splay t@(Node a g (Node (Node b x c) p d)) y =
    if x == y then Node (Node a g b) x (Node c p d) else t
-- zig
splay t@(Node (Node a x b) p c) y = if x == y then Node a x (Node b p c) else t
splay t@(Node a p (Node b x c)) y = if x == y then Node (Node a p b) x c else t
-- otherwise
splay t _ = t
```

Definition of Splay heap in Scheme/Lisp

Since Splay heap is essentially binary search tree. The definition is same. In order to output the tree in in-order. we arrange it as (left element right) format.

Some auxiliary functions to access left child, element, right child and constructor are defined as same as in refskew-heap-def-lisp.

The function which can partition the tree into 2 parts according to a pivot value based on algorithm *PARTITION* is defined as the following. It's a bit complex, but not hard. We compared the pivot and the element and traverse the tree based on binary search tree property. In case there are two left (or right) children traversed, the tree is rotated by splaying. It makes the tree more and more balanced.

```
(define (partition t pivot)
  (if (null? t)
      (cons '() '())
      (let ((l (left t))
            (x (elem t))
            (r (right t)))
        (if (< x pivot)
```

```
    (if (null? r)
        (cons t '())
        (let ((l1 (left r))
              (x1 (elem r))
              (r1 (right r)))
          (if (< x1 pivot)
              (let* ((p (partition r1 pivot))
                     (small (car p))
                     (big (cdr p)))
                (cons (make-tree (make-tree l x l1) x1 small) big))
              (let* ((p (partition l1 pivot))
                     (small (car p))
                     (big (cdr p)))
                (cons (make-tree l x small) (make-tree big x1 r1))))))
    (if (null? l)
        (cons '() t)
        (let ((l1 (left l))
              (x1 (elem l))
              (r1 (right l)))
          (if (> x1 pivot)
              (let* ((p (partition l1 pivot))
                     (small (car p))
                     (big (cdr p)))
                (cons small (make-tree l1 x1 (make-tree r1 x r))))
              (let* ((p (partition r1 pivot))
                     (small (car p))
                     (big (cdr p)))
                (cons (make-tree l1 x1 small) (make-tree big x r)))))))))
```

Although there is no direct pattern matching language feature supporting in Scheme/Lisp, it possible to provide a splay function by guard clause. Compare to the Haskell one, it is a bit more complex.

```
(define (splay l x r y)
  (cond ((eq? y (elem (left l))) ;; zig-zig
         (make-tree (left (left l))
                    (elem (left l))
                    (make-tree (right (left l))
                               (elem l)
                               (make-tree (right l) x r))))
        ((eq? y (elem (right r))) ;; zig-zig
         (make-tree (make-tree (make-tree l x (left r))
                               (elem r)
                               (left (right r)))
                    (elem (right r))
                    (right (right r))))
        ((eq? y (elem (right l))) ;; zig-zag
         (make-tree (make-tree (left l) (elem l) (left (right l)))
                    (elem (right l))
                    (make-tree (right (right l)) x r)))
        ((eq? y (elem (left r))) ;; zig-zag
         (make-tree (make-tree l x (left (left r)))
                    (elem (left r))
                    (make-tree (right (left r)) (elem r) (right r))))
        ((eq? y (elem l)) ;; zig
```

```
 (make-tree (left l) (elem l) (make-tree (right l) x r)))
((eq? y (elem r)) ;; zig
 (make-tree (make-tree l x (left r)) (elem r) (right r)))
(else (make-tree l x r))))
```

8.5.2 Basic heap operations

There are two methods to implement basic heap operations for Splay heap. One is by using $PARTITION$ algorithm we defined, the other is to utilize a $SPLAY$ process, which can be realized in pattern matching way in languages equipped with this feature.

Insertion

If using $PARTITION$ algorithm, once we want to insert a new x into a heap T, we can first partition it into two trees, L and R. Where L contains all nodes smaller than x, and R contains all bigger ones. We then construct a new node, with x as the root and L, R as children.

1: **function** INSERT(T, x)
2: $(L, R) \leftarrow PARTITION(T, x)$
3: **return** $CREATE - NODE(L, x, R)$

If there is a $SPLAY$ algorithm defined, the insert can be done in a recursive way as the following.

1: **function** INSERT'(T, x)
2: **if** $T = NIL$ **then**
3: **return** $CREATE - NODE(NIL, x, NIL)$
4: **if** $KEY(T) < x$ **then**
5: $LEFT(T) \leftarrow INSERT'(LEFT(T), x)$
6: **else**
7: $RIGHT(T) \leftarrow INSERT'(RIGHT(T), x)$
8: **return** $SPLAY(T, x)$

Insertion in Haskell

It's easy to translate the above algorithms into Haskell.

```
insert :: (Ord a) ⇒ STree a → a → STree a
insert t x = Node small x big where (small, big) = partition t x
```

And the pattern matching one is in recursive manner as below.

```
insert' :: (Ord a) ⇒ STree a → a → STree a
insert' E y = Node E y E
insert' (Node l x r) y
    | x > y     = splay (Node (insert' l y) x r) y
    | otherwise = splay (Node l x (insert' r y)) y
```

Insertion in Scheme/Lisp

By using partition method, when insert a new element to the Splay heap, we use this element as the pivot to partition the tree. After that we set this new

element as the new root, the sub tree contains all elements smaller than root as the left child, and the others as the right child. Note that we intend not handle the duplicated elements case, because it quite possible to contains them in Splay heap.

```
(define (insert t x)
  (let* ((p (partition t x))
         (small (car p))
         (big (cdr p)))
    (make-tree small x big)))
```

And if use splay function, the insert can be implemented straightforward like binary search tree, except that we need apply splaying recursively after that.

```
(define (insert-splay t x)
  (cond ((null? t) (make-tree '() x '()))
        ((> (elem t) x)
         (splay (insert-splay (left t) x) (elem t) (right t) x))
        (else
         (splay (left t) (elem t) (insert-splay (right t) x) x))))
```

Verify how Splay improve the balance

In order to show how Splaying improves the balance of binary search tree, we first insert a list of ordered element to the tree and then performs a large number of arbitrary node access.

A look-up algorithm is provided with Splaying operation inside.

1: **function** LOOKUP(T, x)
2: **if** $KEY(T) = x$ **then**
3: **return** T
4: **else if** $KEY(T) > x$ **then**
5: $LEFT(T) \leftarrow LOOKUP(LEFT(T), x)$
6: **else**
7: $RIGHT(T) \leftarrow LOOKUP(RIGHT(T), x)$
8: **return** $SPLAY(T, x)$

Translate this algorithm into Haskell yields the following program[2].

```
lookup' :: (Ord a) ⇒ STree a → a → STree a
lookup' E _ = E
lookup' t@(Node l x r) y
    | x == y    = t
    | x > y     = splay (Node (lookup' l y) x r) y
    | otherwise = splay (Node l x (lookup' r y)) y
```

Next we can create a Splay heap by inserting sequence number from 1 to 10. After that, We random select from a number from this range, and perform looking up 1000 times as below.

```
testSplay = do
  xs ← sequence (replicate 1000 (randomRIO(1, 10)))
  putStrLn $ show (foldl lookup' t xs)
      where
        t = foldl insert' (E::STree Int) [1..10]
```

[2] $LOOKUP$ algorithm in other language is skipped.

Run these test will make the tree quite balance. Below is an example result. Figure 8.14 shows the tree after splaying.

```
Node (Node (Node (Node E 1 E) 2 E) 3 E) 4 (Node (Node E 5 E) 6 (Node
(Node (Node E 7 E) 8 E) 9 (Node E 10 E)))
```

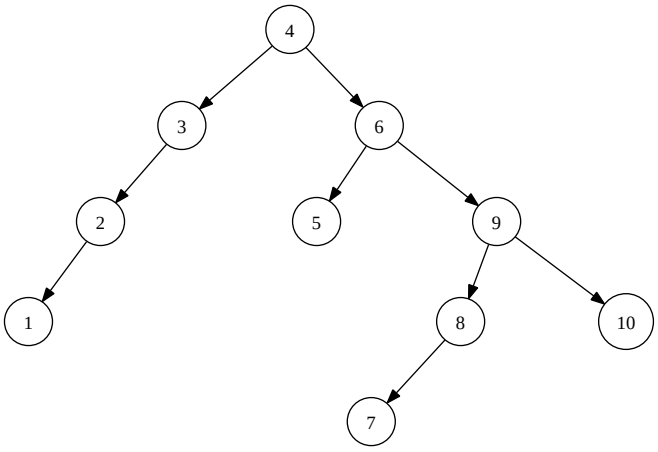


Figure 8.14: Splaying helps improving the balance.

Find minimum (top) and delete minimum (pop)

Since Splay tree is just a special binary search tree, so the minimum element is stored in the left most node. We need keep traversing the left child.

```
1: function FIND-MIN(T)
2:     if LEFT(T) = NIL then
3:         return KEY(T)
4:     else
5:         return FIND − MIN(LEFT(T))
```

And for pop operation, the algorithm need keep traversing in left and also remove the minimum element from the tree. In case there are two left nodes traversed, a splaying operation should be performed.

```
1: function DEL-MIN(T)
2:     if LEFT(T) = NIL then
3:         return RIGHT(T)
4:     else if LEFT(LEFT(T)) = NIL then
5:         LEFT(T) ← RIGHT(LEFT(T))
6:         return T
7:     else                                              ▷ Splaying
8:         l ← LEFT(T)
9:         r ← CREATE − NEW − NODE()
```

10: $LEFT(r) \leftarrow RIGHT(l)$
11: $KEY(r) \leftarrow KEY(T)$
12: $RIGHT(r) \leftarrow RIGHT(T)$
13: $T' \leftarrow CREATE - NEW - NODE()$
14: $LEFT(T') \leftarrow DEL - MIN(LEFT(l))$
15: $KEY(T') \leftarrow KEY(l)$
16: $RIGHT(T') \leftarrow r$
17: **return** T'

Note that the find minimum and delete minimum algorithms both are bound to $O(\lg N)$.

Find minimum (top) and delete minimum in Haskell

When translate the above algorithms into Haskell, one option is to use pattern matching.

```
findMin :: STree a → a
findMin (Node E x _) = x
findMin (Node l x _) = findMin l

deleteMin :: STree a → STree a
deleteMin (Node E x r) = r
deleteMin (Node (Node E x' r') x r) = Node r' x r
deleteMin (Node (Node l' x' r') x r) = Node (deleteMin l') x' (Node r' x r)
```

Find minimum (top) and delete minimum in Scheme/Lisp

In Scheme/Lisp, finding minimum element means keep traversing to left; while for max-heap, we need change the program to keep traversing to right.

```
(define (find-min t)
  (if (null? (left t))
      (elem t)
      (find-min (left t))))
```

And for delete minimum program, except for trivial case, splaying also need be performed if we traverse in left twice.

```
(define (delete-min t)
  (cond ((null? (left t)) (right t))
        ((null? (left (left t))) (make-tree (right (left t)) (elem t) (right t)))
        (else (make-tree (delete-min (left (left t)))
                         (elem (left t))
                         (make-tree (right (left t)) (elem t) (right t))))))
```

Merge

Merge is another basic operation for heaps as it is widely used in Graph algorithms. By using $PARTITION$ algorithm, merge can be realized in $O(\lg N)$ time.

When merging two Splay trees, for non-trivial case, we can take the root element of the first tree as the new root element, then partition the second tree with the new root as the pivot value. After that we recursively merge the

children of the first tree with the partition result. This algorithm is shown as the following.

```
1: function MERGE(T1, T2)
2:     if T1 = NIL then
3:         return T2
4:     else
5:         L ← LEFT(T1)
6:         R ← RIGHT(T1)
7:         k ← KEY(T1)
8:         (L', R') ← PARTITION(T2, k)
9:         return CREATE − NODE(MERGE(L, L'), kMERGE(R, R'))
```

Merge two Splay heaps in Haskell

In Haskell, we can handle the trivial and non-trivial case by pattern matching.

```
merge :: (Ord a) ⇒ STree a → STree a → STree a
merge E t = t
merge (Node l x r) t = Node (merge l l') x (merge r r')
    where (l', r') = partition t x
```

Merge two Splay heaps in Scheme/Lisp

In Scheme/Lisp, we translate the above algorithm strictly as the following.

```
(define (merge t1 t2)
  (if (null? t1)
      t2
      (let* ((p (partition t2 (elem t1)))
             (small (car p))
             (big (cdr p)))
        (make-tree (merge (left t1) small) (elem t1) (merge (right t1) big)))))
```

8.5.3 Heap sort

Since the internal implementation of the Splay heap is completely transparent to the heap interface, the heap sort algorithm can be reused. It means that the heap sort algorithm is generic no matter what the underground data structure is.

8.6 Notes and short summary

In this post, we reviewed the definition of binary heap, and adjust it a bit so that as long as the heap property is maintained, all binary representation of data structures can be used to implement binary heap.

This enable us not only limit to the popular implicit binary heap by array, but also extend to explicit binary heaps including Leftist heap, Skew heap and Splay heap. Note that, the implicit binary heap by array is particularly convenient for imperative implementation because it intense uses random index access which can be mapped to a completely binary tree. It's hard to directly find functional counterpart in this way.

However, by using explicit binary tree, functional implementation can be easily achieved, most of them have $O(\lg N)$ worst case performance, and some of them even reach $O(1)$ amortize time. Okasaki in [6] shows good analysis of these data structures.

In this post, Only pure functional realization for Leftist heap, Skew heap, and Splay heap are explained, they are all possible to be implemented in imperative way. I skipped them only for the purpose of presenting comparable functional algorithm to the implicit binary heap by array.

It's very natural to extend the concept from binary tree to K-ary (K-way) tree, which leads to other useful heap concept such as Binomial heaps, Fibonacci heaps and pairing heaps. I'll introduce them in a separate post later.

8.7 Appendix

All programs provided along with this article are free for downloading.

8.7.1 Prerequisite software

GNU Make is used for easy build some of the program. For C++ and ANSI C programs, GNU GCC and G++ 3.4.4 are used. For Haskell programs GHC 6.10.4 is used for building. For Python programs, Python 2.5 is used for testing, for Scheme/Lisp program, MIT Scheme 14.9 is used.

all source files are put in one folder. Invoke 'make' or 'make all' will build C++ Program.

There is no separate Haskell main program module, however, it is possible to run the program in GHCi.

- bheap.hpp. This is the C++ source file contains binary heap defintion and functions, There are two types of approach, one is the "reference + size" way, the other is "range" representation.
- test.cpp. This is the main C++ program to test bheap.hpp module.
- bheap.py. This is the Python source file for binary heap implementation. It's a self-contained program with test cases embedded. run it directly will performs all test cases. It is also possible to import it as a module.
- LeftistHeap.hs. This is the Haskell program for Leftist Heap with some simple test cases as well. It can be loaded to GHCi directly.
- SkewHeap.hs. This is the Haskell program for Skew heap defintion. Some very simple test cases are provided.
- SplayHeap.hs. This is the Haskell proram for Splay heap definition.
- lefitst.scm. This is the Scheme/Lisp program for Leftist heap.
- skewheap.scm. This is the Scheme/Lisp program for Skew heap. Same as leftist.scm, some gerneric functions are reused.
- genheap.scm. This is the Scheme/Lisp general heap function utilities which are all same for varies of heaps. It can be overwritten afterwards.

- splayheap.scm. This is the Scheme/Lisp program for Splay heap definition.

download position: http://sites.google.com/site/algoxy/btree/bheap.zip

Bibliography

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

[2] Heap (data structure), Wikipedia. http://en.wikipedia.org/wiki/Heap_(data_structure)

[3] Heapsort, Wikipedia. http://en.wikipedia.org/wiki/Heapsort

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

[5] Sorting algorithms/Heapsort. Rosetta Code. http://rosettacode.org/wiki/Sorting_algorithms/Heapsort

[6] Leftist Tree, Wikipedia. http://en.wikipedia.org/wiki/Leftist_tree

[7] Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in Java. http://www.brpreiss.com/books/opus5/index.html

[8] Donald E. Knuth. "The Art of Computer Programming. Volume 3: Sorting and Searching.". Addison-Wesley Professional; 2nd Edition (October 15, 1998). ISBN-13: 978-0201485417. Section 5.2.3 and 6.2.3

[9] Skew heap, Wikipedia. http://en.wikipedia.org/wiki/Skew_heap

[10] Sleator, Daniel Dominic; Jarjan, Robert Endre. "Self-adjusting heaps" SIAM Journal on Computing 15(1):52-69. doi:10.1137/0215004 ISSN 00975397 (1986)

[11] Splay tree, Wikipedia. http://en.wikipedia.org/wiki/Splay_tree

[12] Sleator, Daniel D.; Tarjan, Robert E. (1985), "Self-Adjusting Binary Search Trees", Journal of the ACM 32(3):652 - 686, doi: 10.1145/3828.3835

[13] NIST, "binary heap". http://xw2k.nist.gov/dads//HTML/binaryheap.html

From grape to the world cup, the evolution of selection sort

Liu Xinyu **Liu Xinyu**

Email: liuxinyu95@gmail.com

Chapter 9

From grape to the world cup, the evolution of selection sort

9.1 Introduction

We have introduced the 'hello world' sorting algorithm, insertion sort. In this short chapter, we explain another straightforward sorting method, selection sort. The basic version of selection sort doesn't perform as good as the divide and conqueror methods, e.g. quick sort and merge sort. We'll use the same approaches in the chapter of insertion sort, to analyze why it's slow, and try to improve it by varies of attempts till reach the best bound of comparison based sorting, $O(N \lg N)$, by evolving to heap sort.

The idea of selection sort can be illustrated by a real life story. Consider a kid eating a bunch of grapes. There are two types of children according to my observation. One is optimistic type, that the kid always eats the biggest grape he/she can ever find; the other is pessimistic, that he/she always eats the smallest one.

The first type of kids actually eat the grape in an order that the size decreases monotonically; while the other eat in a increase order. The kid *sorts* the grapes in order of size in fact, and the method used here is selection sort.

Based on this idea, the algorithm of selection sort can be directly described as the following.

In order to sort a series of elements:

- The trivial case, if the series is empty, then we are done, the result is also empty;
- Otherwise, we find the smallest element, and append it to the tail of the result;

Note that this algorithm sorts the elements in increase order; It's easy to sort in decrease order by picking the biggest element instead; We'll introduce about passing a comparator as a parameter later on.

Figure 9.1: Always picking the smallest grape.

This description can be formalized to a equation.

$$sort(A) = \begin{cases} \Phi & : \quad A = \Phi \\ \{m\} \cup sort(A') & : \quad otherwise \end{cases} \tag{9.1}$$

Where m is the minimum element among collection A, and A' is all the rest elements except m:

$$m = min(A)$$
$$A' = A - \{m\}$$

We don't limit the data structure of the collection here. Typically, A is an array in imperative environment, and a list (singly linked-list particularly) in functional environment, and it can even be other data struture which will be introduced later.

The algorithm can also be given in imperative manner.

function SORT(A)
  $X \leftarrow \Phi$
  while $A \neq \Phi$ **do**
    $x \leftarrow$ MIN(A)
    $A \leftarrow$ DEL(A, x)
    $X \leftarrow$ APPEND(X, x)
  return X

Figure 9.2 depicts the process of this algorithm.

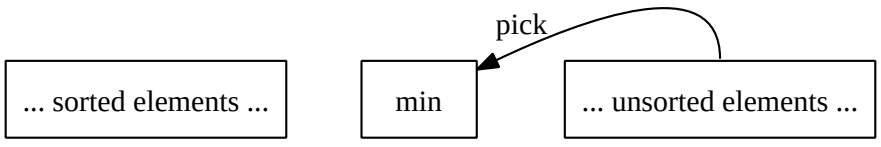


Figure 9.2: The left part is sorted data, continuously pick the minimum element in the rest and append it to the result.

We just translate the very original idea of 'eating grapes' line by line without considering any expense of time and space. This realization stores the result in

X, and when an selected element is appended to X, we delete the same element from A. This indicates that we can change it to 'in-place' sorting to reuse the spaces in A.

The idea is to store the minimum element in the first cell in A (we use term 'cell' if A is an array, and say 'node' if A is a list); then store the second minimum element in the next cell, then the third cell, ...

One solution to realize this sorting strategy is swapping. When we select the i-th minimum element, we swap it with the element in the i-th cell:

```
function SORT(A)
    for i ← 1 to |A| do
        m ← MIN(A[i...])
        EXCHANGE A[i] ↔ m
```

Denote $A = \{a_1, a_2, ..., a_N\}$. At any time, when we process the i-th element, all elements before i, as $\{a_1, a_2, ..., a_{i-1}\}$ have already been sorted. We locate the minimum element among the $\{a_i, a_{i+1}, ..., a_N\}$, and exchange it with a_i, so that the i-th cell contains the right value. The process is repeatedly executed until we arrived at the last element.

This idea can be illustrated by figure 9.3.

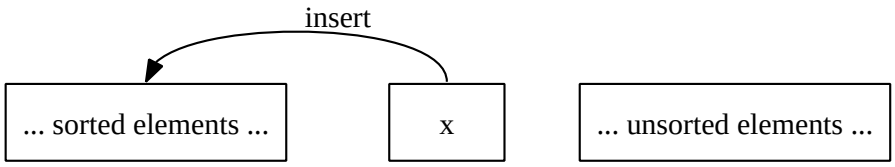


Figure 9.3: The left part is sorted data, continuously pick the minimum element in the rest and put it to the right position.

9.2 Finding the minimum

We haven't completely realized the selection sort, because we take the operation of finding the minimum (or the maximum) element as a black box. It's a puzzle how does a kid locate the biggest or the smallest grape. And this is an interesting topic for computer algorithms.

The easiest but not so fast way to find the minimum in a collection is to perform a scan. There are several ways to interpret this scan process. Consider that we want to pick the biggest grape. We start from any grape, compare it with another one, and pick the bigger one; then we take a next grape and compare it with the one we selected so far, pick the bigger one and go on the take-and-compare process, until there are not any grapes we haven't compared.

It's easy to get loss in real practice if we don't mark which grape has been compared. There are two ways to to solve this problem, which are suitable for different data-structures respectively.

9.2.1 Labeling

Method 1 is to label each grape with a number: $\{1, 2, ..., N\}$, and we systematically perform the comparison in the order of this sequence of labels. That we first compare grape number 1 and grape number 2, pick the bigger one; then we take grape number 3, and do the comparison, ... We repeat this process until arrive at grape number N. This is quite suitable for elements stored in an array.

function MIN(A)
  $m \leftarrow A[1]$
  for $i \leftarrow 2$ to $|A|$ **do**
    if $A[i] < m$ **then**
      $m \leftarrow A[i]$
  return m

With MIN defined, we can complete the basic version of selection sort (or naive version without any optimization in terms of time and space).

However, this algorithm returns the value of the minimum element instead of its location (or the label of the grape), which needs a bit tweaking for the in-place version. Some languages such as ISO C++, support returning the reference as result, so that the swap can be achieved directly as below.

```
template<typename T>
T& min(T* from, T* to) {
    T* m;
    for (m = from++; from != to; ++from)
        if (*from < *m)
            m = from;
    return *m;
}

template<typename T>
void ssort(T* xs, int n) {
    int i;
    for (i = 0; i < n; ++i)
        std::swap(xs[i], min(xs+i, xs+n));
}
```

In environments without reference semantics, the solution is to return the location of the minimum element instead of the value:

function MIN-AT(A)
  $m \leftarrow$ FIRST-INDEX(A)
  for $i \leftarrow m + 1$ to $|A|$ **do**
    if $A[i] < A[m]$ **then**
      $m \leftarrow i$
  return m

Note that since we pass $A[i...]$ to MIN-AT as the argument, we assume the first element $A[i]$ as the smallest one, and examine all elements $A[i + 1], A[i + 2], ...$ one by one. Function FIRST-INDEX() is used to retrieve i from the input parameter.

The following Python example program, for example, completes the basic in-place selection sort algorithm based on this idea. It explicitly passes the range information to the function of finding the minimum location.

```
def ssort(xs):
    n = len(xs)
    for i in range(n):
        m = min_at(xs, i, n)
        (xs[i], xs[m]) = (xs[m], xs[i])
    return xs

def min_at(xs, i, n):
    m = i;
    for j in range(i+1, n):
        if xs[j] < xs[m]:
            m = j
    return m
```

9.2.2 Grouping

Another method is to group all grapes in two parts: the group we have examined, and the rest we haven't. We denote these two groups as A and B; All the elements (grapes) as L. At the beginning, we haven't examine any grapes at all, thus A is empty (Φ), and B contains all grapes. We can select arbitrary two grapes from B, compare them, and put the loser (the smaller one for example) to A. After that, we repeat this process by continuously picking arbitrary grapes from B, and compare with the winner of the previous time until B becomes empty. At this time being, the final winner is the minimum element. And A turns to be $L - \{min(L)\}$, which can be used for the next time minimum finding.

There is an invariant of this method, that at any time, we have $L = A \cup \{m\} \cup B$, where m is the winner so far we hold.

This approach doesn't need the collection of grapes being indexed (as being labeled in method 1). It's suitable for any traversable data structures, including linked-list etc. Suppose b_1 is an arbitrary element in B if B isn't empty, and B' is the rest of elements with b_1 being removed, this method can be formalized as the below auxiliary function.

$$min'(A, m, B) = \begin{cases} (m, A) & : \quad B = \Phi \\ min'(A \cup \{m\}, b_1, B') & : \quad b_1 < m \\ min'(A \cup \{b_1\}, m, B') & : \quad otherwise \end{cases} \tag{9.2}$$

In order to pick the minimum element, we call this auxiliary function by passing an empty A, and use an arbitrary element (for instance, the first one) to initialize m:

$$extractMin(L) = min'(\Phi, l_1, L') \tag{9.3}$$

Where L' is all elements in L except for the first one l_1. The algorithm $extractMin$) doesn't not only find the minimum element, but also returns the updated collection which doesn't contain this minimum. Summarize this minimum extracting algorithm up to the basic selection sort definition, we can create a complete functional sorting program, for example as this Haskell code snippet.

```
sort [] = []
sort xs = x : sort xs' where
  (x, xs') = extractMin xs
```

```
extractMin (x:xs) = min' [] x xs where
  min' ys m [] = (m, ys)
  min' ys m (x:xs) = if m < x then min' (x:ys) m xs else min' (m:ys) x xs
```

The first line handles the trivial edge case that the sorting result for empty list is obvious empty; The second clause ensures that, there is at least one element, that's why the `extractMin` function needn't other pattern-matching.

One may think the second clause of `min'` function should be written like below:

```
min' ys m (x:xs) = if m < x then min' ys ++ [x] m xs
                            else min' ys ++ [m] x xs
```

Or it will produce the updated list in reverse order. Actually, it's necessary to use 'cons' instead of appending here. This is because appending is linear operation which is proportion to the length of part A, while 'cons' is constant $O(1)$ time operation. In fact, we needn't keep the relative order of the list to be sorted, as it will be re-arranged anyway during sorting.

It's quite possible to keep the relative order during sorting, while ensure the performance of finding the minimum element not degrade to quadratic. The following equation defines a solution.

$$extractMin(L) = \begin{cases} (l_1, \Phi) & : & |L| = 1 \\ (l_1, L') & : & l_1 < m, (m, L'') = extractMin(L') \\ (m, l_1 \cup L'') & : & otherwise \end{cases} \tag{9.4}$$

If L is a singleton, the minimum is the only element it contains. Otherwise, denote l_1 as the first element in L, and L' contains the rest elements except for l_1, that $L' = \{l_2, l_3, ...\}$. The algorithm recursively finding the minimum element in L', which yields the intermediate result as (m, L''), that m is the minimum element in L', and L'' contains all rest elements except for m. Comparing l_1 with m, we can determine which of them is the final minimum result.

The following Haskell program implements this version of selection sort.

```
sort [] = []
sort xs = x : sort xs' where
  (x, xs') = extractMin xs

extractMin [x] = (x, [])
extractMin (x:xs) = if x < m then (x, xs) else (m, x:xs') where
  (m, xs') = extractMin xs
```

Note that only 'cons' operation is used, we needn't appending at all because the algorithm actually examines the list from right to left. However, it's not free, as this program need book-keeping the context (via call stack typically). The relative order is ensured by the nature of recursion. Please refer to the appendix about tail recursion call for detailed discussion.

9.2.3 performance of the basic selection sorting

Both the labeling method, and the grouping method need examine all the elements to pick the minimum in every round; and we totally pick up the minimum

element N times. Thus the performance is around $N+(N-1)+(N-2)+...+1$ which is $\frac{N(N+1)}{2}$. Selection sort is a quadratic algorithm bound to $O(N^2)$ time.

Compare to the insertion sort, which we introduced previously, selection sort performs same in its best case, worst case and average case. While insertion sort performs well in best case (that the list has been reverse ordered, and it is stored in linked-list) as $O(N)$, and the worst performance is $O(N^2)$.

In the next sections, we'll examine, why selection sort performs poor, and try to improve it step by step.

Exercise 9.1

- Implement the basic imperative selection sort algorithm (the none in-place version) in your favorite programming language. Compare it with the in-place version, and analyze the time and space effectiveness.

9.3 Minor Improvement

9.3.1 Parameterize the comparator

Before any improvement in terms of performance, let's make the selection sort algorithm general enough to handle different sorting criteria.

We've seen two opposite examples so far, that one may need sort the elements in ascending order or descending order. For the former case, we need repeatedly finding the minimum, while for the later, we need find the maximum instead. They are just two special cases. In real world practice, one may want to sort things in varies criteria, e.g. in terms of size, weight, age, ...

One solution to handle them all is to passing the criteria as a compare function to the basic selection sort algorithms. For example:

$$sort(c, L) = \begin{cases} \Phi & : \quad L = \Phi \\ m \cup sort(c, L'') & : \quad otherwise, (m, L'') = extract(c, L') \end{cases} \tag{9.5}$$

And the algorithm $extract(c, L)$ is defined as below.

$$extract(c, L) = \begin{cases} (l_1, \Phi) & : \quad |L| = 1 \\ (l_1, L') & : \quad c(l_1, m), (m, L'') = extract(c, L') \\ (m, \{l_1\} \cup L'') & : \quad \neg c(l_1, m) \end{cases} \tag{9.6}$$

Where c is a comparator function, it takes two elements, compare them and returns the result of which one is preceding of the other. Passing 'less than' operator ($<$) turns this algorithm to be the version we introduced in previous section.

Some environments require to pass the total ordering comparator, which returns result among 'less than', 'equal', and 'greater than'. We needn't such strong condition here, that c only tests if 'less than' is satisfied. However, as the minimum requirement, the comparator should meet the strict weak ordering as following [3]:

- Irreflexivity, for all x, it's not the case that $x < x$;
- Asymmetric, For all x and y, if $x < y$, then it's not the case $y < x$;
- Transitivity, For all x, y, and z, if $x < y$, and $y < z$, then $x < z$;

The following Scheme/Lisp program translates this generic selection sorting algorithm. The reason why we choose Scheme/Lisp here is because the lexical scope can simplify the needs to pass the 'less than' comparator for every function calls.

```
(define (sel-sort-by ltp? lst)
  (define (ssort lst)
    (if (null? lst)
        lst
        (let ((p (extract-min lst)))
          (cons (car p) (ssort (cdr p))))))
  (define (extract-min lst)
    (if (null? (cdr lst))
        lst
        (let ((p (extract-min (cdr lst))))
          (if (ltp? (car lst) (car p))
              lst
              (cons (car p) (cons (car lst) (cdr p)))))))
  (ssort lst))
```

Note that, both `ssort` and `extract-min` are inner functions, so that the 'less than' comparator `ltp?` is available to them. Passing '<' to this function yields the normal sorting in ascending order:

```
(sel-sort-by < '(3 1 2 4 5 10 9))
;Value 16: (1 2 3 4 5 9 10)
```

It's possible to pass varies of comparator to imperative selection sort as well. This is left as an exercise to the reader.

For the sake of brevity, we only consider sorting elements in ascending order in the rest of this chapter. And we'll not pass comparator as a parameter unless it's necessary.

9.3.2 Trivial fine tune

The basic in-place imperative selection sorting algorithm iterates all elements, and picking the minimum by traversing as well. It can be written in a compact way, that we inline the minimum finding part as an inner loop.

```
procedure SORT(A)
    for i ← 1 to |A| do
        m ← i
        for j ← i + 1 to |A| do
            if A[i] < A[m] then
                m ← i
        EXCHANGE A[i] ↔ A[m]
```

Observe that, when we are sorting N elements, after the first $N-1$ minimum ones are selected, the left only one, is definitely the N-th big element, so that

we need NOT find the minimum if there is only one element in the list. This indicates that the outer loop can iterate to $N-1$ instead of N.

Another place we can fine tune, is that we needn't swap the elements if the i-th minimum one is just $A[i]$. The algorithm can be modified accordingly as below:

```
procedure SORT(A)
    for i ← 1 to |A| − 1 do
        m ← i
        for j ← i + 1 to |A| do
            if A[i] < A[m] then
                m ← i
        if m ≠ i then
            EXCHANGE A[i] ↔ A[m]
```

Definitely, these modifications won't affects the performance in terms of big-O.

9.3.3 Cock-tail sort

Knuth gave an alternative realization of selection sort in [1]. Instead of selecting the minimum each time, we can select the maximum element, and put it to the last position. This method can be illustrated by the following algorithm.

```
procedure SORT'(A)
    for i ← |A| down-to 2 do
        m ← i
        for j ← 1 to i − 1 do
            if A[m] < A[i] then
                m ← i
        EXCHANGE A[i] ↔ A[m]
```

As shown in figure 9.4, at any time, the elements on right most side are sorted. The algorithm scans all unsorted ones, and locate the maximum. Then, put it to the tail of the unsorted range by swapping.

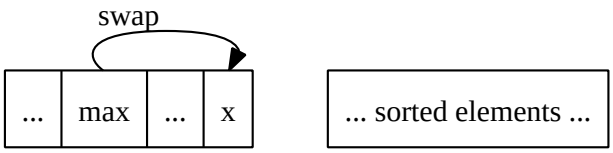


Figure 9.4: Select the maximum every time and put it to the end.

This version reveals the fact that, selecting the maximum element can sort the element in ascending order as well. What's more, we can find both the minimum and the maximum elements in one pass of traversing, putting the minimum at the first location, while putting the maximum at the last position. This approach can speed up the sorting slightly (halve the times of the outer loop).

```
procedure SORT(A)
    for i ← 1 to ⌊|A|/2⌋ do
```

$min \leftarrow i$
$max \leftarrow |A| + 1 - i$
if $A[max] < A[min]$ **then**
    EXCHANGE $A[min] \leftrightarrow A[max]$
for $j \leftarrow i + 1$ to $|A| - i$ **do**
    if $A[j] < A[min]$ **then**
        $min \leftarrow j$
    if $A[max] < A[j]$ **then**
        $max \leftarrow j$
EXCHANGE $A[i] \leftrightarrow A[min]$
EXCHANGE $A[|A| + 1 - i] \leftrightarrow A[max]$

This algorithm can be illustrated as in figure 9.5, at any time, the left most and right most parts contain sorted elements so far. That the smaller sorted ones are on the left, while the bigger sorted ones are on the right. The algorithm scans the unsorted ranges, located both the minimum and the maximum positions, then put them to the head and the tail position of the unsorted ranges by swapping.

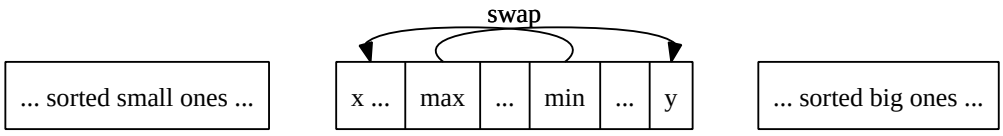


Figure 9.5: Select both the minimum and maximum in one pass, and put them to the proper positions.

Note that it's necessary to swap the left most and right most elements before the inner loop if they are not in correct order. This is because we scan the range excluding these two elements. Another method is to initialize the first element of the unsorted range as both the maximum and minimum before the inner loop. However, since we need two swapping operations after the scan, it's possible that the first swapping moves the maximum or the minimum from the position we just found, which leads the second swapping malfunctioned. How to solve this problem is left as exercise to the reader.

The following Python example program implements this cock-tail sort algorithm.

```
def cocktail_sort(xs):
    n = len(xs)
    for i in range(n / 2):
        (mi, ma) = (i, n - 1 -i)
        if xs[ma] < xs[mi]:
            (xs[mi], xs[ma]) = (xs[ma], xs[mi])
        for j in range(i+1, n - 1 - i):
            if xs[j] < xs[mi]:
                mi = j
            if xs[ma] < xs[j]:
                ma = j
        (xs[i], xs[mi]) = (xs[mi], xs[i])
        (xs[n - 1 - i], xs[ma]) = (xs[ma], xs[n - 1 - i])
```

```
return xs
```

It's possible to realize cock-tail sort in functional approach as well. An intuitive recursive description can be given like this:

- Trivial edge case: If the list is empty, or there is only one element in the list, the sorted result is obviously the origin list;
- Otherwise, we select the minimum and the maximum, put them in the head and tail positions, then recursively sort the rest elements.

This algorithm description can be formalized by the following equation.

$$sort(L) = \begin{cases} L & : & |L| \leq 1 \\ \{l_{min}\} \cup sort(L'') \cup \{l_{max}\} & : & otherwise \end{cases} \tag{9.7}$$

Where the minimum and the maximum are extracted from L by a function $select(L)$.

$$(l_{min}, L'', l_{max}) = select(L)$$

Note that, the minimum is actually linked to the front of the recursive sort result. Its semantic is a constant $O(1)$ time 'cons' (refer to the appendix of this book for detail). While the maximum is appending to the tail. This is typically a linear $O(N)$ time expensive operation. We'll optimize it later.

Function $select(L)$ scans the whole list to find both the minimum and the maximum. It can be defined as below:

$$select(L) = \begin{cases} (min(l_1, l_2), max(l_1, l_2)) & : & L = \{l_1, l_2\} \\ (l_1, \{l_{min}\} \cup L'', l_{max}) & : & l_1 < l_{min} \\ (l_{min}, \{l_{max}\} \cup L'', l_1) & : & l_{max} < l_1 \\ (l_{min}, \{l_1\} \cup L'', l_{max}) & : & otherwise \end{cases} \tag{9.8}$$

Where $(l_{min}, L'', l_{max}) = select(L')$ and L' is the rest of the list except for the first element l_1. If there are only two elements in the list, we pick the smaller as the minimum, and the bigger as the maximum. After extract them, the list becomes empty. This is the trivial edge case; Otherwise, we take the first element l_1 out, then recursively perform selection on the rest of the list. After that, we compare if l_1 is less then the minimum or greater than the maximum candidates, so that we can finalize the result.

Note that for all the cases, there is no appending operation to form the result. However, since selection must scan all the element to determine the minimum and the maximum, it is bound to $O(N)$ linear time.

The complete example Haskell program is given as the following.

```
csort [] = []
csort [x] = [x]
csort xs = mi : csort xs' ++ [ma] where
  (mi, xs', ma) = extractMinMax xs

extractMinMax [x, y] = (min x y, [], max x y)
extractMinMax (x:xs) | x < mi = (x, mi:xs', ma)
                     | ma < x = (mi, ma:xs', x)
                     | otherwise = (mi, x:xs', ma)
  where (mi, xs', ma) = extractMinMax xs
```

We mentioned that the appending operation is expensive in this intuitive version. It can be improved. This can be achieved in two steps. The first step is to convert the cock-tail sort into tail-recursive call. Denote the sorted small ones as A, and sorted big ones as B in figure 9.5. We use A and B as accumulators. The new cock-tail sort is defined as the following.

$$sort'(A, L, B) = \begin{cases} A \cup L \cup B & : \quad L = \Phi \vee |L| = 1 \\ sort'(A \cup \{l_{min}\}, L'', \{l_{max}\} \cup B) & : \quad otherwise \end{cases} \tag{9.9}$$

Where l_{min}, l_{max} and L'' are defined as same as before. And we start sorting by passing empty A and B: $sort(L) = sort'(\Phi, L, \Phi)$.

Besides the edge case, observing that the appending operation only happens on $A \cup \{l_{min}\}$; while l_{max} is only linked to the head of B. This appending occurs in every recursive call. To eliminate it, we can store A in reverse order as $\overleftarrow{A}$, so that l_{max} can be 'cons' to the head instead of appending. Denote $cons(x, L) = \{x\} \cup L$ and $append(L, x) = L \cup \{x\}$, we have the below equation.

$$\begin{aligned} append(L, x) &= reverse(cons(x, reverse(L))) \\ &= reverse(cons(x, \overleftarrow{L})) \end{aligned} \tag{9.10}$$

Finally, we perform a reverse to turn $\overleftarrow{A}$ back to A. Based on this idea, the algorithm can be improved one more step as the following.

$$sort'(A, L, B) = \begin{cases} reverse(A) \cup B & : \quad L = \Phi \\ reverse(\{l_1\} \cup A) \cup B & : \quad |L| = 1 \\ sort'(\{l_{min}\} \cup A, L'', \{l_{max}\} \cup B) & : \end{cases} \tag{9.11}$$

This algorithm can be implemented by Haskell as below.

```
csort' xs = cocktail [] xs [] where
  cocktail as [] bs = reverse as ++ bs
  cocktail as [x] bs = reverse (x:as) ++ bs
  cocktail as xs bs = let (mi, xs', ma) = extractMinMax xs
                      in cocktail (mi:as) xs' (ma:bs)
```

Exercise 9.2

- Realize the imperative basic selection sort algorithm, which can take a comparator as a parameter. Please try both dynamic typed language and static typed language. How to annotate the type of the comparator as general as possible in a static typed language?
- Implement Knuth's version of selection sort in your favorite programming language.
- An alternative to realize cock-tail sort is to assume the i-th element both the minimum and the maximum, after the inner loop, the minimum and maximum are found, then we can swap the the minimum to the i-th position, and the maximum to position $|A|+1-i$. Implement this solution in your favorite imperative language. Please note that there are several special edge cases should be handled correctly:

- $A = \{max, min, ...\}$;
- $A = \{..., max, min\}$;
- $A = \{max, ..., min\}$.

Please don't refer to the example source code along with this chapter before you try to solve this problem.

9.4 Major improvement

Although cock-tail sort halves the numbers of loop, the performance is still bound to quadratic time. It means that, the method we developed so far handles big data poorly compare to other divide and conquer sorting solutions.

To improve selection based sort essentially, we must analyze where is the bottle-neck. In order to sort the elements by comparison, we must examine all the elements for ordering. Thus the outer loop of selection sort is necessary. However, must it scan all the elements every time to select the minimum? Note that when we pick the smallest one at the first time, we actually traverse the whole collection, so that we know which ones are relative big, and which ones are relative small partially.

The problem is that, when we select the further minimum elements, instead of re-using the ordering information we obtained previously, we drop them all, and blindly start a new traverse.

So the key point to improve selection based sort is to re-use the previous result. There are several approaches, we'll adopt an intuitive idea inspired by football match in this chapter.

9.4.1 Tournament knock out

The football world cup is held every four years. There are 32 teams from different continent play the final games. Before 1982, there were 16 teams compete for the tournament finals[4].

For simplification purpose, let's go back to 1978 and imagine a way to determine the champion: In the first round, the teams are grouped into 8 pairs to play the game; After that, there will be 8 winner, and 8 teams will be out. Then in the second round, these 8 teams are grouped into 4 pairs. This time there will be 4 winners after the second round of games; Then the top 4 teams are divided into 2 pairs, so that there will be only two teams left for the final game.

The champion is determined after the total 4 rounds of games. And there are actually $8 + 4 + 2 + 1 = 16$ games. Now we have the world cup champion, however, the world cup game won't finish at this stage, we need to determine which is the silver medal team.

Readers may argue that isn't the team beaten by the champion at the final game the second best? This is true according to the real world cup rule. However, it isn't fair enough in some sense.

We often heard about the so called 'group of death', Let's suppose that Brazil team is grouped with Deutch team at the very beginning. Although both teams are quite strong, one of them must be knocked out. It's quite possible

that even the team loss that game can beat all the other teams except for the champion. Figure 9.6 illustrates such case.

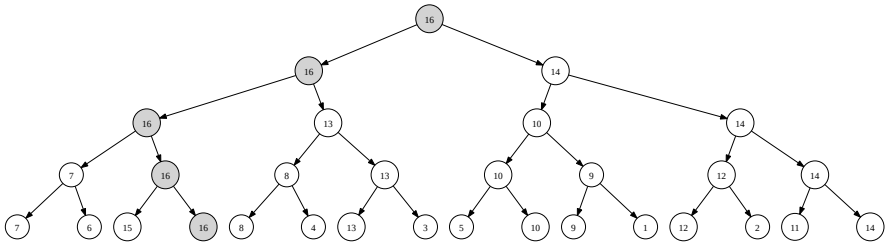


Figure 9.6: The element 15 is knocked out in the first round.

Imagine that every team has a number. The bigger the number, the stronger the team. Suppose that the stronger team always beats the team with smaller number, although this is not true in real world. But this simplification is fair enough for us to develop the tournament knock out solution. This maximum number which represents the champion is 16. Definitely, team with number 14 isn't the second best according to our rules. It should be 15, which is knocked out at the first round of comparison.

The key question here is to find an effective way to locate the second maximum number in this tournament tree. After that, what we need is to apply the same method to select the third, the fourth, ..., to accomplish the selection based sort.

One idea is to assign the champion a very small number (for instance, $-\infty$), so that it won't be selected next time, and the second best one, becomes the new champion. However, suppose there are 2^m teams for some natural number m, it still takes $2^{m-1} + 2^{m-2} + ... + 2 + 1 = 2^m$ times of comparison to determine the new champion. Which is as slow as the first time.

Actually, we needn't perform a bottom-up comparison at all since the tournament tree stores plenty of ordering information. Observe that, the second best team must be beaten by the champion at sometime, or it will be the final winner. So we can track the path from the root of the tournament tree to the leaf of the champion, examine all the teams along with this path to find the second best team.

In figure 9.6, this path is marked in gray color, the elements to be examined are $\{14, 13, 7, 15\}$. Based on this idea, we refine the algorithm like below.

1. Build a tournament tree from the elements to be sorted, so that the champion (the maximum) becomes the root;

2. Extract the root from the tree, perform a top-down pass and replace the maximum with $-\infty$;

3. Perform a bottom-up back-track along the path, determine the new champion and make it as the new root;

4. Repeat step 2 until all elements have been extracted.

Figure 9.7, 9.8, and 9.9 show the steps of applying this strategy.

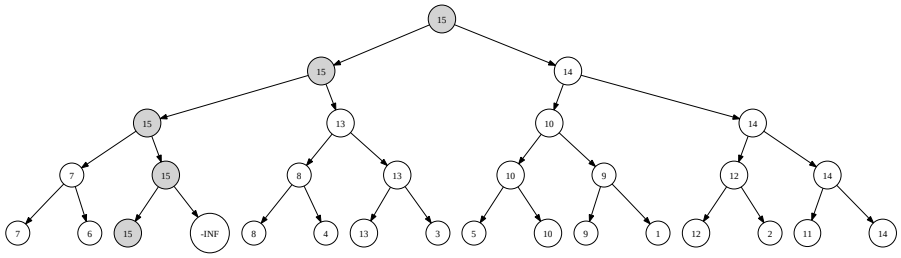


Figure 9.7: Extract 16, replace it with $-\infty$, 15 sifts up to root.

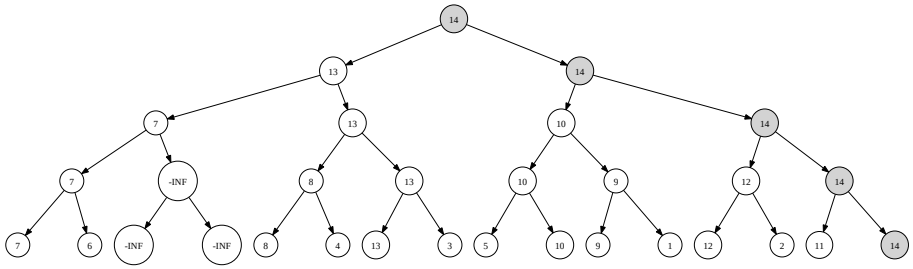


Figure 9.8: Extract 15, replace it with $-\infty$, 14 sifts up to root.

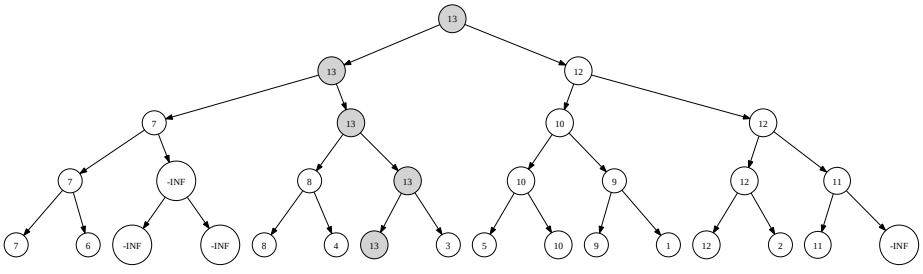


Figure 9.9: Extract 14, replace it with $-\infty$, 13 sifts up to root.

We can reuse the binary tree definition given in the first chapter of this book to represent tournament tree. In order to back-track from leaf to the root, every node should hold a reference to its parent (concept of pointer in some environment such as ANSI C):

```
struct Node {
  Key key;
  struct Node *left, *right, *parent;
};
```

To build a tournament tree from a list of elements (suppose the number of elements are 2^m for some m), we can first wrap each element as a leaf, so that we obtain a list of binary trees. We take every two trees from this list, compare their keys, and form a new binary tree with the bigger key as the root; the two trees are set as the left and right children of this new binary tree. Repeat this operation to build a new list of trees. The height of each tree is increased by 1. Note that the size of the tree list halves after such a pass, so that we can keep reducing the list until there is only one tree left. And this tree is the finally built tournament tree.

function BUILD-TREE(A)
  $T \leftarrow \Phi$
  for each $x \in A$ **do**
    $t \leftarrow$ CREATE-NODE
    KEY(t) $\leftarrow x$
    APPEND(T, t)
  while $|T| > 1$ **do**
    $T' \leftarrow \Phi$
    for every $t_1, t_2 \in$ T **do**
      $t \leftarrow$ CREATE-NODE
      KEY(t) $\leftarrow$ MAX(KEY(t_1), KEY(t_2))
      LEFT(t) $\leftarrow t_1$
      RIGHT(t) $\leftarrow t_2$
      PARENT(t_1) $\leftarrow t$
      PARENT(t_2) $\leftarrow t$
      APPEND(T', t)
    $T \leftarrow T'$
  return $T[1]$

Suppose the length of the list A is N, this algorithm firstly traverses the list to build tree, which is linear to N time. Then it repeatedly compares pairs, which loops proportion to $N + \frac{N}{2} + \frac{N}{4} + ... + 2 = 2N$. So the total performance is bound to $O(N)$ time.

The following ANSI C program implements this tournament tree building algorithm.

```
struct Node* build(const Key* xs, int n) {
    int i;
    struct Node *t, **ts = (struct Node**) malloc(sizeof(struct Node*) * n);
    for (i = 0; i < n; ++i)
        ts[i] = leaf(xs[i]);
    for (; n > 1; n /= 2)
        for (i = 0; i < n; i += 2)
```

```
            ts[i/2] = branch(max(ts[i]→key, ts[i+1]→key), ts[i], ts[i+1]);
    t = ts[0];
    free(ts);
    return t;
}
```

The type of key can be defined somewhere, for example:

```
typedef int Key;
```

Function `leaf(x)` creats a leaf node, with value `x` as key, and sets all its fields, left, right and parent to `NIL`. While function `branch(key, left, right)` creates a branch node, and links the new created node as parent of its two children if they are not empty. For the sake of brevity, we skip the detail of them. They are left as exercise to the reader, and the complete program can be downloaded along with this book.

Some programming environments, such as Python provides tool to iterate every two elements at a time, for example:

```
for x, y in zip(*[iter(ts)]*2):
```

We skip such language specific feature, readers can refer to the Python example program along with this book for details.

When the maximum element is extracted from the tournament tree, we replace it with $-\infty$, and repeatedly replace all these values from the root to the leaf. Next, we back-track to root through the parent field, and determine the new maximum element.

function EXTRACT-MAX(T)
  $m \leftarrow$ KEY(T)
  KEY(T) $\leftarrow -\infty$
  while $\neg$ LEAF?(T) **do** ▷ The top down pass
    if KEY(LEFT(T)) $= m$ **then**
      $T \leftarrow$ LEFT(T)
    else
      $T \leftarrow$ RIGHT(T)
    KEY(T) $\leftarrow -\infty$
  while PARENT(T) $\neq \Phi$ **do** ▷ The bottom up pass
    $T \leftarrow$ PARENT(T)
    KEY(T) $\leftarrow$ MAX(KEY(LEFT(T)), KEY(RIGHT(T)))
  return m

This algorithm returns the extracted maximum element, and modifies the tournament tree in-place. Because we can't represent $-\infty$ in real program by limited length of word, one approach is to define a relative negative big number, which is less than all the elements in the tournament tree, for example, suppose all the elements are greater than -65535, we can define negative infinity as below:

```
#define N_INF -65535
```

We can implements this algorithm as the following ANSI C example program.

```
Key pop(struct Node* t) {
    Key x = t→key;
    t→key = N_INF;
    while (!isleaf(t)) {
```

```
        t = t→left→key == x ? t→left : t→right;
        t→key = N_INF;
    }
    while (t→parent) {
        t = t→parent;
        t→key = max(t→left→key, t→right→key);
    }
    return x;
}
```

The behavior of EXTRACT-MAX is quite similar to the pop operation for some data structures, such as queue, and heap, thus we name it as `pop` in this code snippet.

Algorithm EXTRACT-MAX process the tree in tow passes, one is top-down, then a bottom-up along the path that the 'champion team wins the world cup'. Because the tournament tree is well balanced, the length of this path, which is the height of the tree, is bound to $O(\lg N)$, where N is the number of the elements to be sorted (which are equal to the number of leaves). Thus the performance of this algorithm is $O(\lg N)$.

It's possible to realize the tournament knock out sort now. We build a tournament tree from the elements to be sorted, then continuously extract the maximum. If we want to sort in monotonically increase order, we put the first extracted one to the right most, then insert the further extracted elements one by one to left; Otherwise if we want to sort in decrease order, we can just append the extracted elements to the result. Below is the algorithm sorts elements in ascending order.

procedure SORT(A)
  $T \leftarrow$ BUILD-TREE(A)
  for $i \leftarrow |A|$ down to 1 **do**
    $A[i] \leftarrow$ EXTRACT-MAX(T)

Translating it to ANSI C example program is straightforward.

```
void tsort(Key* xs, int n) {
    struct Node* t = build(xs, n);
    while(n)
        xs[--n] = pop(t);
    release(t);
}
```

This algorithm firstly takes $O(N)$ time to build the tournament tree, then performs N pops to select the maximum elements so far left in the tree. Since each pop operation is bound to $O(\lg N)$, thus the total performance of tournament knock out sorting is $O(N \lg N)$.

Refine the tournament knock out

It's possible to design the tournament knock out algorithm in purely functional approach. And we'll see that the two passes (first top-down replace the champion with $-\infty$, then bottom-up determine the new champion) in pop operation can be combined in recursive manner, so that we needn't the parent field any more. We can re-use the functional binary tree definition as the following example Haskell code.

```
data Tr a = Empty | Br (Tr a) a (Tr a)
```

Thus a binary tree is either empty or a branch node contains a key, a left sub tree and a right sub tree. Both children are again binary trees.

We've use hard coded big negative number to represents $-\infty$. However, this solution is ad-hoc, and it forces all elements to be sorted are greater than this pre-defined magic number. Some programming environments support algebraic type, so that we can define negative infinity explicitly. For instance, the below Haskell program setups the concept of infinity [1].

```
data Infinite a = NegInf | Only a | Inf deriving (Eq, Ord)
```

From now on, we switch back to use the $min()$ function to determine the winner, so that the tournament selects the minimum instead of the maximum as the champion.

Denote function $key(T)$ returns the key of the tree rooted at T. Function $wrap(x)$ wraps the element x into a leaf node. Function $tree(l, k, r)$ creates a branch node, with k as the key, l and r as the two children respectively.

The knock out process, can be represented as comparing two trees, picking the smaller key as the new key, and setting these two trees as children:

$$branch(T_1, T_2) = tree(T_1, min(key(T_1), key(T_2)), T_2) \tag{9.12}$$

This can be implemented in Haskell word by word:

```
branch t1 t2 = Br t1 (min (key t1) (key t2)) t2
```

There is limitation in our tournament sorting algorithm so far. It only accepts collection of elements with size of 2^m, or we can't build a complete binary tree. This can be actually solved in the tree building process. Remind that we pick two trees every time, compare and pick the winner. This is perfect if there are always even number of trees. Considering a case in football match, that one team is absent for some reason (sever flight delay or whatever), so that there left one team without a challenger. One option is to make this team the winner, so that it will attend the further games. Actually, we can use the similar approach.

To build the tournament tree from a list of elements, we wrap every element into a leaf, then start the building process.

$$build(L) = build'(\{wrap(x) | x \in L\}) \tag{9.13}$$

The $build'(\mathbb{T})$ function terminates when there is only one tree left in $\mathbb{T}$, which is the champion. This is the trivial edge case. Otherwise, it groups every two trees in a pair to determine the winners. When there are odd numbers of trees, it just makes the last tree as the winner to attend the next level of tournament and recursively repeats the building process.

$$build'(\mathbb{T}) = \begin{cases} \mathbb{T} & : & |\mathbb{T}| \leq 1 \\ build'(pair(\mathbb{T})) & : & otherwise \end{cases} \tag{9.14}$$

[1] The order of the definition of 'NegInf', regular number, and 'Inf' is significant if we want to derive the default, correct comparing behavior of 'Ord'. Anyway, it's possible to specify the detailed order by make it as an instance of 'Ord'. However, this is Language specific feature which is out of the scope of this book. Please refer to other textbook about Haskell.

Note that this algorithm actually handles another special cases, that the list to be sort is empty. The result is obviously empty.

Denote $\mathbb{T} = \{T_1, T_2, ...\}$ if there are at least two trees, and $\mathbb{T}'$ represents the left trees by removing the first two. Function $pair(\mathbb{T})$ is defined as the following.

$$pair(\mathbb{T}) = \begin{cases} \{branch(T_1, T_2)\} \cup pair(\mathbb{T}') & : & |\mathbb{T}| \geq 2 \\ \mathbb{T} & : & otherwise \end{cases} \tag{9.15}$$

The complete tournament tree building algorithm can be implemented as the below example Haskell program.

```
fromList :: (Ord a) ⇒ [a] → Tr (Infinite a)
fromList = build ∘ (map wrap) where
  build [] = Empty
  build [t] = t
  build ts = build $ pair ts
  pair (t1:t2:ts) = (branch t1 t2):pair ts
  pair ts = ts
```

When extracting the champion (the minimum) from the tournament tree, we need examine either the left child sub-tree or the right one has the same key, and recursively extract on that tree until arrive at the leaf node. Denote the left sub-tree of T as L, right sub-tree as R, and K as its key. We can define this popping algorithm as the following.

$$pop(T) = \begin{cases} tree(\Phi, \infty, \Phi) & : & L = \Phi \wedge R = \Phi \\ tree(L', min(key(L'), key(R)), R) & : & K = key(L), L' = pop(L) \\ tree(L, min(key(L), key(R')), R') & : & K = key(R), R' = pop(R) \end{cases} \tag{9.16}$$

It's straightforward to translate this algorithm into example Haskell code.

```
pop (Br Empty _ Empty) = Br Empty Inf Empty
pop (Br l k r) | k == key l = let l' = pop l in Br l' (min (key l') (key r)) r
               | k == key r = let r' = pop r in Br l (min (key l) (key r')) r'
```

Note that this algorithm only removes the current champion without returning it. So it's necessary to define a function to get the champion at the root node.

$$top(T) = key(T) \tag{9.17}$$

With these functions defined, tournament knock out sorting can be formalized by using them.

$$sort(L) = sort'(build(L)) \tag{9.18}$$

Where $sort'(T)$ continuously pops the minimum element to form a result list

$$sort'(T) = \begin{cases} \Phi & : & T = \Phi \vee key(T) = \infty \\ \{top(T)\} \cup sort'(pop(T)) & : & otherwise \end{cases} \tag{9.19}$$

The rest of the Haskell code is given below to complete the implementation.

```
top = only ∘ key

tsort :: (Ord a) ⇒ [a] → [a]
tsort = sort' ∘ fromList where
    sort' Empty = []
    sort' (Br _ Inf _) = []
    sort' t = (top t) : (sort' $ pop t)
```

And the auxiliary function `only`, `key`, `wrap` accomplished with explicit infinity support are list as the following.

```
only (Only x) = x
key (Br _ k _ ) = k
wrap x = Br Empty (Only x) Empty
```

Exercise 9.3

- Implement the helper function `leaf()`, `branch`, `max() lsleaf()`, and `release()` to complete the imperative tournament tree program.
- Implement the imperative tournament tree in a programming language support GC (garbage collection).
- Why can our tournament tree knock out sort algorithm handle duplicated elements (elements with same value)? We say a sorting algorithm stable, if it keeps the original order of elements with same value. Is the tournament tree knock out sorting stable?
- Design an imperative tournament tree knock out sort algorithm, which satisfies the following:
  - Can handle arbitrary number of elements;
  - Without using hard coded negative infinity, so that it can take elements with any value.
- Compare the tournament tree knock out sort algorithm and binary tree sort algorithm, analyze efficiency both in time and space.
- Compare the heap sort algorithm and binary tree sort algorithm, and do same analysis for them.

9.4.2 Final improvement by using heap sort

We manage improving the performance of selection based sorting to $O(N \lg N)$ by using tournament knock out. This is the limit of comparison based sort according to [1]. However, there are still rooms for improvement. After sorting, there lefts a complete binary tree with all leaves and branches hold useless infinite values. This isn't space efficient at all. Can we release the nodes when popping?

Another observation is that if there are N elements to be sorted, we actually allocate about $2N$ tree nodes. N for leaves and N for branches. Is there any better way to halve the space usage?

The final sorting structure described in equation 9.19 can be easily uniformed to a more general one if we treat the case that the tree is empty if its root holds infinity as key:

$$sort'(T) = \begin{cases} \Phi & : \quad T = \Phi \\ \{top(T)\} \cup sort'(pop(T)) & : \quad otherwise \end{cases} \tag{9.20}$$

This is exactly as same as the one of heap sort we gave in previous chapter. Heap always keeps the minimum (or the maximum) on the top, and provides fast pop operation. The binary heap by implicit array encodes the tree structure in array index, so there aren't any extra spaces allocated except for the N array cells. The functional heaps, such as leftist heap and splay heap allocate N nodes as well. We'll introduce more heaps in next chapter which perform well in many aspects.

9.5 Short summary

In this chapter, we present the evolution process of selection based sort. selection sort is easy and commonly used as example to teach students about embedded looping. It has simple and straightforward structure, but the performance is quadratic. In this chapter, we do see that there exists ways to improve it not only by some fine tuning, but also fundamentally change the data structure, which leads to tournament knock out and heap sort.

Bibliography

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

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

[3] Wikipedia. "Strict weak order". http://en.wikipedia.org/wiki/Strict_weak_order

[4] Wikipedia. "FIFA world cup". http://en.wikipedia.org/wiki/FIFA_World_Cup

Binomial heap, Fibonacci heap, and pairing heap
Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

Chapter 10

Binomial heap, Fibonacci heap, and pairing heap

10.1 Introduction

In previous chapter, we mentioned that heaps can be generalized and implemented with varies of data structures. However, only binary heaps are focused so far no matter by explicit binary trees or implicit array.

It's quite natural to extend the binary tree to K-ary [1] tree. In this chapter, we first show Binomial heaps which is actually consist of forest of K-ary trees. Binomial heaps gain the performance for all operations to $O(\lg N)$, as well as keeping the finding minimum element to $O(1)$ time.

If we delay some operations in Binomial heaps by using lazy strategy, it turns to be Fibonacci heap.

All binary heaps we have shown perform no less than $O(\lg N)$ time for merging, we'll show it's possible to improve it to $O(1)$ with Fibonacci heap, which is quite helpful to graph algorithms. Actually, Fibonacci heap achieves almost all operations to good amortized time bound as $O(1)$, and left the heap pop to $O(\lg N)$.

Finally, we'll introduce about the pairing heaps. It has the best performance in practice although the proof of it is still a conjecture for the time being.

10.2 Binomial Heaps

10.2.1 Definition

Binomial heap is more complex than most of the binary heaps. However, it has excellent merge performance which bound to $O(\lg N)$ time. A binomial heap is consist of a list of binomial trees.

Binomial tree

In order to explain why the name of the tree is 'binomial', let's review the famous Pascal's triangle (in China, we call it Yang Hui's triangle) [4].

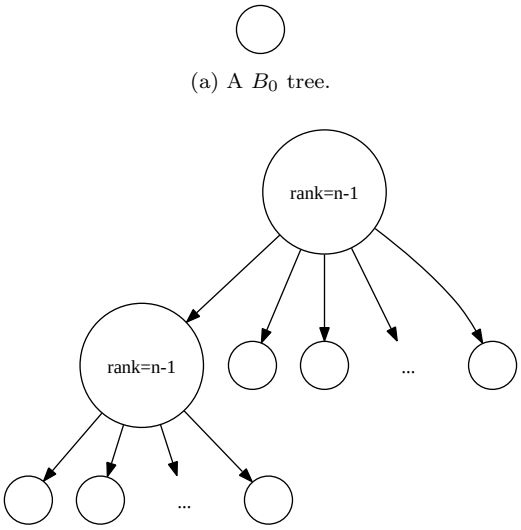
```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
...
```

In each row, the numbers are all binomial coefficients. There are many ways to gain a series of binomial coefficient numbers. One of them is by using recursive composition. Binomial trees, as well, can be defined in this way as the following.

- A binomial tree of rank 0 has only a node as the root;
- A binomial tree of rank N is consist of two rank $N-1$ binomial trees, Among these 2 sub trees, the one has the bigger root element is linked as the leftmost child of the other.

We denote a binomial tree of rank 0 as B_0, and the binomial tree of rank n as B_n.

Figure 10.1 shows a B_0 tree and how to link 2 B_{n-1} trees to a B_n tree.



(a) A B_0 tree.

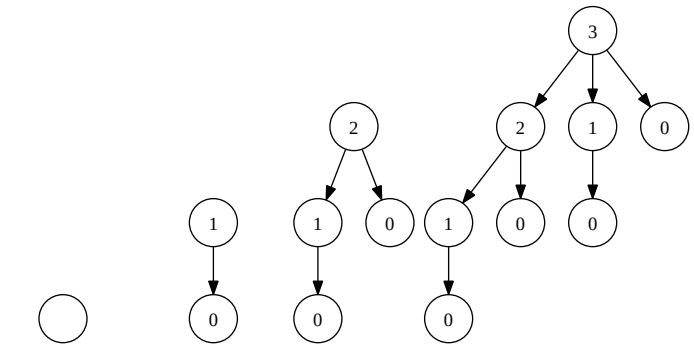
(b) Linking 2 B_{n-1} trees yields a B_n tree.

Figure 10.1: Recursive definition of binomial trees

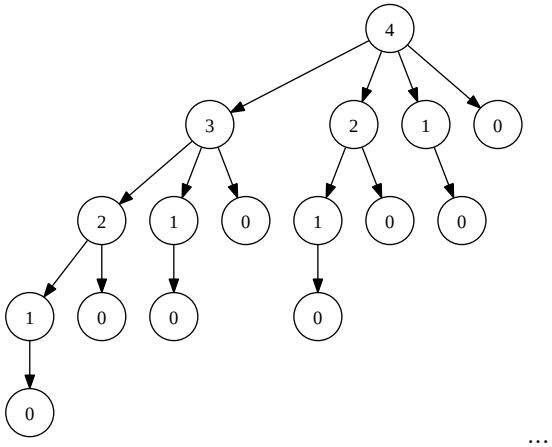
With this recursive definition, it easy to draw the form of binomial trees of rank 0, 1, 2, ..., as shown in figure 10.2

Observing the binomial trees reveals some interesting properties. For each rank N binomial tree, if counting the number of nodes in each row, it can be found that it is the binomial number.

For instance for rank 4 binomial tree, there is 1 node as the root; and in the second level next to root, there are 4 nodes; and in 3rd level, there are 6 nodes; and in 4-th level, there are 4 nodes; and the 5-th level, there is 1 node. They



(a) B_0 tree; (b) B_1 tree; (c) B_2 tree; (d) B_3 tree;



...

(e) B_4 tree;

Figure 10.2: Forms of binomial trees with rank = 0, 1, 2, 3, 4, ...

are exactly 1, 4, 6, 4, 1 which is the 5th row in Pascal's triangle. That's why we call it binomial tree.

Another interesting property is that the total number of node for a binomial tree with rank N is 2^N. This can be proved either by binomial theory or the recursive definition directly.

Binomial heap

With binomial tree defined, we can introduce the definition of binomial heap. A binomial heap is a set of binomial trees (or a forest of binomial trees) that satisfied the following properties.

- Each binomial tree in the heap conforms to *heap property*, that the key of a node is equal or greater than the key of its parent. Here the heap is actually min-heap, for max-heap, it changes to 'equal or less than'. In this chapter, we only discuss about min-heap, and max-heap can be equally applied by changing the comparison condition.

- There is at most one binomial tree which has the rank r. In other words, there are no two binomial trees have the same rank.

This definition leads to an important result that for a binomial heap contains N elements, and if convert N to binary format yields $a_0, a_1, a_2, ..., a_m$, where a_0 is the LSB and a_m is the MSB, then for each $0 \leq i \leq m$, if $a_i = 0$, there is no binomial tree of rank i and if $a_i = 1$, there must be a binomial tree of rank i.

For example, if a binomial heap contains 5 element, as 5 is '(LSB)101(MSB)', then there are 2 binomial trees in this heap, one tree has rank 0, the other has rank 2.

Figure 10.3 shows a binomial heap which have 19 nodes, as 19 is '(LSB)11001(MSB)' in binary format, so there is a B_0 tree, a B_1 tree and a B_4 tree.

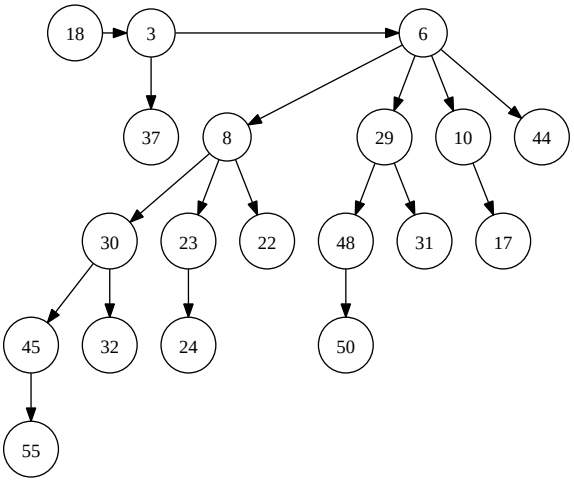


Figure 10.3: A binomial heap with 19 elements

Data layout

There are two ways to define K-ary trees imperatively. One is by using 'left-child, right-sibling' approach[3]. It is compatible with the typical binary tree structure. For each node, it has two fields, left field and right field. We use the left field to point to the first child of this node, and use the right field to point to the sibling node of this node. All siblings are represented as a single directional linked list. Figure 10.4 shows an example tree represented in this way.

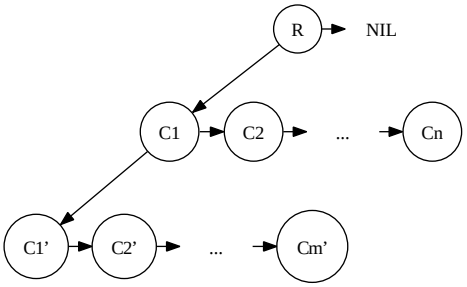


Figure 10.4: Example tree represented in 'left-child, right-sibling' way. R is the root node, it has no sibling, so it right side is pointed to NIL. $C_1, C_2, ..., C_n$ are children of R. C_1 is linked from the left side of R, other siblings of C_1 are linked one next to each other on the right side of C_1. $C'_2, ..., C'_m$ are children of C_1.

The other way is to use the library defined collection container, such as array or list to represent all children of a node.

Since the rank of a tree plays very important role, we also defined it as a field.

For 'left-child, right-sibling' method, we defined the binomial tree as the following.[1]

```
class BinomialTree:
    def __init__(self, x = None):
        self.rank = 0
        self.key = x
        self.parent = None
        self.child = None
        self.sibling = None
```

When initialize a tree with a key, we create a leaf node, set its rank as zero and all other fields are set as NIL.

It quite nature to utilize pre-defined list to represent multiple children as below.

```
class BinomialTree:
    def __init__(self, x = None):
        self.rank = 0
        self.key = x
        self.parent = None
        self.children = []
```

[1] C programs are also provided along with this book.

For purely functional settings, such as in Haskell language, binomial tree are defined as the following.

```
data BiTree a = Node { rank :: Int
                     , root :: a
                     , children :: [BiTree a]}
```

While binomial heap are defined as a list of binomial trees (a forest) with ranks in monotonically increase order. And as another implicit constraint, there are no two binomial trees have the same rank.

```
type BiHeap a = [BiTree a]
```

10.2.2 Basic heap operations

Linking trees

Before dive into the basic heap operations such as pop and insert, We'll first realize how to link two binomial trees with same rank into a bigger one. According to the definition of binomial tree and heap property that the root always contains the minimum key, we firstly compare the two root values, select the smaller one as the new root, and insert the other tree as the first child in front of all other children. Suppose function $Key(T)$, $Children(T)$, and $Rank(T)$ access the key, children and rank of a binomial tree respectively.

$$link(T_1, T_2) = \begin{cases} node(r+1, x, \{T_2\} \cup C_1) & : \quad x < y \\ node(r+1, y, \{T_1\} \cup C_2) & : \quad otherwise \end{cases} \tag{10.1}$$

Where

$$\begin{aligned} x &= Key(T_1) \\ y &= Key(T_2) \\ r &= Rank(T_1) = Rank(T_2) \\ C_1 &= Children(T_1) \\ C_2 &= Children(T_2) \end{aligned}$$

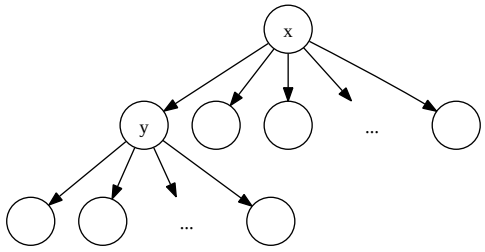


Figure 10.5: Suppose $x < y$, insert y as the first child of x.

Note that the link operation is bound to $O(1)$ time if the $\cup$ is a constant time operation. It's easy to translate the link function to Haskell program as the following.

```
link :: (Ord a) ⇒ BiTree a → BiTree 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)
```

It's possible to realize the link operation in imperative way. If we use 'left child, right sibling' approach, we just link the tree which has the bigger key to the left side of the other's key, and link the children of it to the right side as sibling. Figure 10.6 shows the result of one case.

1: **function** LINK(T_1, T_2)
2: **if** KEY(T_2) < KEY(T_1) **then**
3: Exchange $T_1 \leftrightarrow T_2$
4: SIBLING(T_2) $\leftarrow$ CHILD(T_1)
5: CHILD(T_1) $\leftarrow T_2$
6: PARENT(T_2) $\leftarrow T_1$
7: RANK(T_1) $\leftarrow$ RANK(T_1) + 1
8: **return** T_1

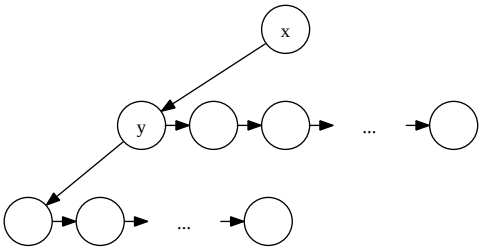


Figure 10.6: Suppose $x < y$, link y to the left side of x and link the original children of x to the right side of y.

And if we use a container to manage all children of a node, the algorithm is like below.

1: **function** LINK'(T_1, T_2)
2: **if** KEY(T_2) < KEY(T_1) **then**
3: Exchange $T_1 \leftrightarrow T_2$
4: PARENT(T_2) $\leftarrow T_1$
5: INSERT-BEFORE(CHILDREN(T_1), T_2)
6: RANK(T_1) $\leftarrow$ RANK(T_1) + 1
7: **return** T_1

It's easy to translate both algorithms to real program. Here we only show the Python program of LINK' for illustration purpose [2].

```
def link(t1, t2):
    if t2.key < t1.key:
        (t1, t2) = (t2, t1)
    t2.parent = t1
    t1.children.insert(0, t2)
    t1.rank = t1.rank + 1
    return t1
```

[2]The C and C++ programs are also available along with this book

Exercise 10.1

Implement the tree-linking program in your favorite language with left-child, right-sibling method.

We mentioned linking is a constant time algorithm and it is true when using left-child, right-sibling approach, However, if we use container to manage the children, the performance depends on the concrete implementation of the container. If it is plain array, the linking time will be proportion to the number of children. In this chapter, we assume the time is constant. This is true if the container is implemented in linked-list.

Insert a new element to the heap (push)

As the rank of binomial trees in a forest is monotonically increasing, by using the *link* function defined above, it's possible to define an auxiliary function, so that we can insert a new tree, with rank no bigger than the smallest one, to the heap which is a forest actually.

Denote the non-empty heap as $H = \{T_1, T_2, ..., T_n\}$, we define

$$insertT(H, T) = \begin{cases} \{T\} & : & H = \phi \\ \{T\} \cup H & : & Rank(T) < Rank(T_1) \\ insertT(H', link(T, T_1)) & : & otherwise \end{cases} \tag{10.2}$$

where

$$H' = \{T_2, T_3, ..., T_n\}$$

The idea is that for the empty heap, we set the new tree as the only element to create a singleton forest; otherwise, we compare the ranks of the new tree and the first tree in the forest, if they are same, we link them together, and recursively insert the linked result (a tree with rank increased by one) to the rest of the forest; If they are not same, since the pre-condition constraints the rank of the new tree, it must be the smallest, we put this new tree in front of all the other trees in the forest.

From the binomial properties mentioned above, there are at most $O(\lg N)$ binomial trees in the forest, where N is the total number of nodes. Thus function *insertT* performs at most $O(\lg N)$ times linking, which are all constant time operation. So the performance of *insertT* is $O(\lg N)$. [3]

The relative Haskell program is given as below.

```
insertTree :: (Ord a) ⇒ BiHeap a → BiTree a → BiHeap a
insertTree [] t = [t]
insertTree ts@(t':ts') t = if rank t < rank t' then t:ts
                           else insertTree ts' (link t t')
```

[3] There is interesting observation by comparing this operation with adding two binary numbers. Which will lead to topic of *numeric representation*[6].

With this auxiliary function, it's easy to realize the insertion. We can wrap the new element to be inserted as the only leaf of a tree, then insert this tree to the binomial heap.

$$insert(H, x) = insertT(H, node(0, x, \phi)) \tag{10.3}$$

And we can continuously build a heap from a series of elements by folding. For example the following Haskell define a helper function 'fromList'.

```
fromList = foldl insert []
```

Since wrapping an element as a singleton tree takes $O(1)$ time, the real work is done in $insertT$, the performance of binomial heap insertion is bound to $O(\lg N)$.

The insertion algorithm can also be realized with imperative approach.

Algorithm 4 Insert a tree with 'left-child-right-sibling' method.

```
1: function INSERT-TREE(H, T)
2:     while H ≠ φ∧ RANK(HEAD(H)) = RANK(T) do
3:         (T_1, H) ← EXTRACT-HEAD(H)
4:         T ← LINK(T, T_1)
5:     SIBLING(T) ← H
6:     return T
```

Algorithm 4 continuously linking the first tree in a heap with the new tree to be inserted if they have the same rank. After that, it puts the linked-list of the rest trees as the sibling, and returns the new linked-list.

If using a container to manage the children of a node, the algorithm can be given in Algorithm 5.

Algorithm 5 Insert a tree with children managed by a container.

```
1: function INSERT-TREE'(H, T)
2:     while H ≠ φ∧ RANK(H[0]) = RANK(T) do
3:         T_1 ← POP(H)
4:         T ← LINK(T, T_1)
5:     HEAD-INSERT(H, T)
6:     return H
```

In this algorithm, function POP removes the first tree $T_1 = H[0]$ from the forest. And function HEAD-INSERT, insert a new tree before any other trees in the heap, so that it becomes the first element in the forest.

With either INSERT-TREE or INSERT-TREE' defined. Realize the binomial heap insertion is trivial.

Algorithm 6 Imperative insert algorithm

```
1: function INSERT(H, x)
2:     return INSERT-TREE(H, NODE(0, x, φ))
```

The following python program implement the insert algorithm by using a container to manage sub-trees. the 'left-child, right-sibling' program is left as an exercise.

```
def insert_tree(ts, t):
    while ts !=[] and t.rank == ts[0].rank:
        t = link(t, ts.pop(0))
    ts.insert(0, t)
    return ts

def insert(h, x):
    return insert_tree(h, BinomialTree(x))
```

Exercise 10.2

Write the insertion program in your favorite imperative programming language by using the 'left-child, right-sibling' approach.

Merge two heaps

When merge two binomial heaps, we actually try to merge two forests of binomial trees. According to the definition, there can't be two trees with the same rank and the ranks are in monotonically increasing order. Our strategy is very similar to merge sort. That in every iteration, we take the first tree from each forest, compare their ranks, and pick the smaller one to the result heap; if the ranks are equal, we then perform linking to get a new tree, and recursively insert this new tree to the result of merging the rest trees.

Figure 10.7 illustrates the idea of this algorithm. This method is different from the one given in [3].

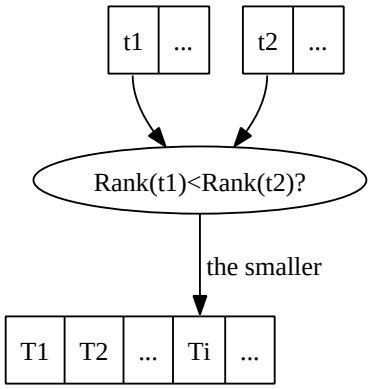
We can formalize this idea with a function. For non-empty cases, we denote the two heaps as $H_1 = \{T_1, T_2, ...\}$ and $H_2 = \{T'_1, T'_2, ...\}$. Let $H'_1 = \{T_2, T_3, ...\}$ and $H'_2 = \{T'_2, T'_3, ...\}$.

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ \{T_1\} \cup merge(H'_1, H_2) & : & Rank(T_1) < Rank(T'_1) \\ \{T'_1\} \cup merge(H_1, H'_2) & : & Rank(T_1) > Rank(T'_1) \\ insertT(merge(H'_1, H'_2), link(T_1, T'_1)) & : & otherwise \end{cases} \tag{10.4}$$

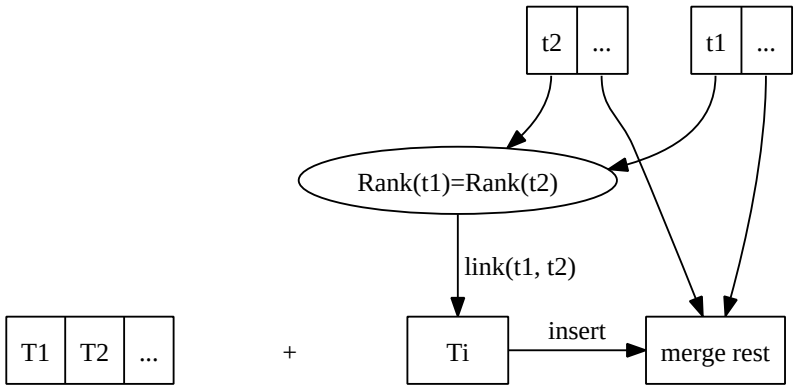
To analysis the performance of merge, suppose there are m_1 trees in H_1, and m_2 trees in H_2. There are at most $m_1 + m_2$ trees in the merged result. If there are no two trees have the same rank, the merge operation is bound to $O(m_1 + m_2)$. While if there need linking for the trees with same rank, $insertT$ performs at most $O(m_1 + m_2)$ time. Consider the fact that $m_1 = 1 + \lfloor \lg N_1 \rfloor$ and $m_2 = 1 + \lfloor \lg N_2 \rfloor$, where N_1, N_2 are the numbers of nodes in each heap, and $\lfloor \lg N_1 \rfloor + \lfloor \lg N_2 \rfloor \leq 2\lfloor \lg N \rfloor$, where $N = N_1 + N_2$, is the total number of nodes. the final performance of merging is $O(\lg N)$.

Translating this algorithm to Haskell yields the following program.

```
merge:: (Ord a) ⇒ BiHeap a → BiHeap a → BiHeap a
merge ts1 [] = ts1
merge [] ts2 = ts2
merge ts1@(t1:ts1') ts2@(t2:ts2')
    | rank t1 < rank t2 = t1:(merge ts1' ts2)
    | rank t1 > rank t2 = t2:(merge ts1 ts2')
    | otherwise = insertTree (merge ts1' ts2') (link t1 t2)
```



(a) Pick the tree with smaller rank to the result.



(b) If two trees have same rank, link them to a new tree, and recursively insert it to the merge result of the rest.

Figure 10.7: Merge two heaps.

Merge algorithm can also be described in imperative way as shown in algorithm 7.

Algorithm 7 imperative merge two binomial heaps

```
1:  function MERGE(H_1, H_2)
2:      if H_1 = φ then
3:          return H_2
4:      if H_2 = φ then
5:          return H_1
6:      H ← φ
7:      while H_1 ≠ φ ∧ H_2 ≠ φ do
8:          T ← φ
9:          if RANK(H_1) < RANK(H_2) then
10:             (T, H_1) ← EXTRACT-HEAD(H_1)
11:         else if RANK(H_2) < RANK(H_1) then
12:             (T, H_2) ← EXTRACT-HEAD(H_2)
13:         else                                          ▷ Equal rank
14:             (T_1, H_1) ← EXTRACT-HEAD(H_1)
15:             (T_2, H_2) ← EXTRACT-HEAD(H_2)
16:             T ← LINK(T_1, T_2)
17:         APPEND-TREE(H, T)
18:     if H_1 ≠ φ then
19:         APPEND-TREES(H, H_1)
20:     if H_2 ≠ φ then
21:         APPEND-TREES(H, H_2)
22:     return H
```

Since both heaps contain binomial trees with rank in monotonically increasing order. Each iteration, we pick the tree with smallest rank and append it to the result heap. If both trees have same rank we perform linking first. Consider the APPEND-TREE algorithm, The rank of the new tree to be appended, can't be less than any other trees in the result heap according to our merge strategy, however, it might be equal to the rank of the last tree in the result heap. This can happen if the last tree appended are the result of linking, which will increase the rank by one. In this case, we must link the new tree to be inserted with the last tree. In below algorithm, suppose function LAST(H) refers to the last tree in a heap, and APPEND(H, T) just appends a new tree at the end of a forest.

```
1: function APPEND-TREE(H, T)
2:     if H ≠ φ∧ RANK(T) = RANK(LAST(H)) then
3:         LAST(H) ← LINK(T, LAST(H))
4:     else
5:         APPEND(H, T)
```

Function APPEND-TREES repeatedly call this function, so that it can append all trees in a heap to the other heap.

```
1: function APPEND-TREES(H_1, H_2)
2:     for each T ∈ H_2 do
3:         H_1 ← APPEND-TREE(H_1, T)
```

The following Python program translates the merge algorithm.

```
def append_tree(ts, t):
    if ts != [] and ts[-1].rank == t.rank:
        ts[-1] = link(ts[-1], t)
    else:
        ts.append(t)
    return ts

def append_trees(ts1, ts2):
    return reduce(append_tree, ts2, ts1)

def merge(ts1, ts2):
    if ts1 == []:
        return ts2
    if ts2 == []:
        return ts1
    ts = []
    while ts1 != [] and ts2 != []:
        t = None
        if ts1[0].rank < ts2[0].rank:
            t = ts1.pop(0)
        elif ts2[0].rank < ts1[0].rank:
            t = ts2.pop(0)
        else:
            t = link(ts1.pop(0), ts2.pop(0))
        ts = append_tree(ts, t)
    ts = append_trees(ts, ts1)
    ts = append_trees(ts, ts2)
    return ts
```

Exercise 10.3

The program given above uses a container to manage sub-trees. Implement the merge algorithm in your favorite imperative programming language with 'left-child, right-sibling' approach.

Pop

Among the forest which forms the binomial heap, each binomial tree conforms to heap property that the root contains the minimum element in that tree. However, the order relationship of these roots can be arbitrary. To find the minimum element in the heap, we can select the smallest root of these trees. Since there are $\lg N$ binomial trees, this approach takes $O(\lg N)$ time.

However, after we locate the minimum element (which is also know as the top element of a heap), we need remove it from the heap and keep the binomial property to accomplish heap-pop operation. Suppose the forest forms the binomial heap consists trees of $B_i, B_j, ..., B_p, ..., B_m$, where B_k is a binomial tree of rank k, and the minimum element is the root of B_p. If we delete it, there will be p children left, which are all binomial trees with ranks $p-1, p-2, ..., 0$.

One tool at hand is that we have defined $O(\lg N)$ merge function. A possible approach is to reverse the p children, so that their ranks change to monotonically increasing order, and forms a binomial heap H_p. The rest of trees is still a

binomial heap, we represent it as $H' = H - B_p$. Merging H_p and H' given the final result of pop. Figure 10.8 illustrates this idea.

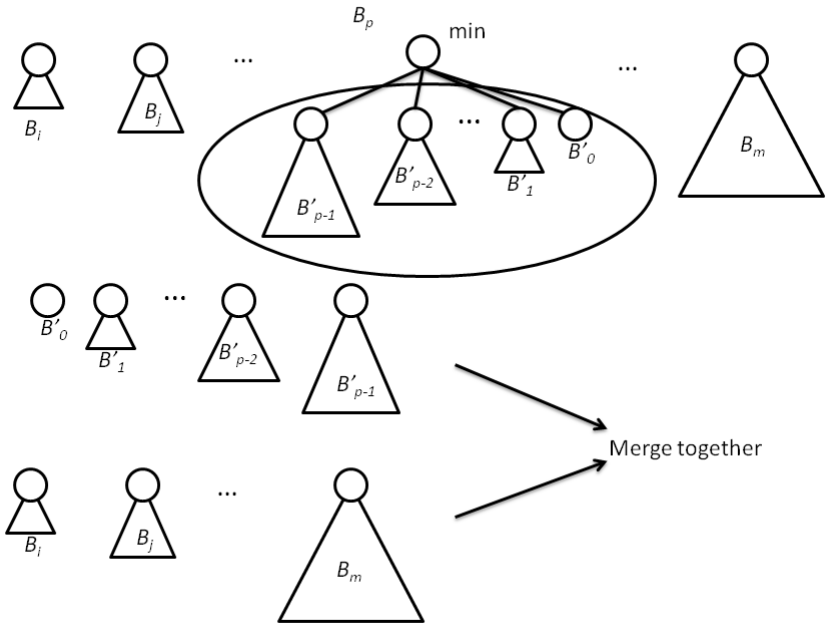


Figure 10.8: Pop the minimum element from a binomial heap.

In order to realize this algorithm, we first need to define an auxiliary function, which can extract the tree contains the minimum element at root from the forest.

$$extractMin(H) = \begin{cases} (T, \phi) & : & \text{H is a singleton as } \{T\} \\ (T_1, H') & : & Root(T_1) < Root(T') \\ (T', \{T_1\} \cup H'') & : & otherwise \end{cases} \tag{10.5}$$

where

$$\begin{array}{ll} H = \{T_1, T_2, ...\} & \text{for the non-empty forest case;} \\ H' = \{T_2, T_3, ...\} & \text{is the forest without the first tree;} \\ (T', H'') = extractMin(H') & \end{array}$$

The result of this function is a tuple. The first part is the tree which has the minimum element at root, the second part is the rest of the trees after remove the first part from the forest.

This function examine each of the trees in the forest thus is bound to $O(\lg N)$ time.

The relative Haskell program can be give respectively.

```
extractMin :: (Ord a) ⇒ BiHeap a → (BiTree a, BiHeap a)
```

```
extractMin [t] = (t, [])
extractMin (t:ts) = if root t < root t' then (t, ts)
                    else (t', t:ts')
    where
      (t', ts') = extractMin ts
```

With this function defined, to return the minimum element is trivial.

```
findMin :: (Ord a) ⇒ BiHeap a → a
findMin = root ∘ fst. extractMin
```

Of course, it's possible to just traverse forest and pick the minimum root without remove the tree for this purpose. Below imperative algorithm describes it with 'left child, right sibling' approach.

```
1: function FIND-MINIMUM(H)
2:     T ← HEAD(H)
3:     min ← ∞
4:     while T ≠ φ do
5:         if KEY(T)< min then
6:             min ← KEY(T)
7:         T ← SIBLING(T)
8:     return min
```

While if we manage the children with collection containers, the link list traversing is abstracted as to find the minimum element among the list. The following Python program shows about this situation.

```
def find_min(ts):
    min_t = min(ts, key=lambda t: t.key)
    return min_t.key
```

Next we define the function to delete the minimum element from the heap by using $extractMin$.

$$delteMin(H) = merge(reverse(Children(T)), H') \tag{10.6}$$

where

$$(T, H') = extractMin(H)$$

Translate the formula to Haskell program is trivial and we'll skip it.

To realize the algorithm in procedural way takes extra efforts including list reversing etc. We left these details as exercise to the reader. The following pseudo code illustrate the imperative pop algorithm

```
1: function EXTRACT-MIN(H)
2:     (T_min, H) ← EXTRACT-MIN-TREE(H)
3:     H ← MERGE(H, REVERSE(CHILDREN(T_min)))
4:     return (KEY(T_min), H)
```

With pop operation defined, we can realize heap sort by creating a binomial heap from a series of numbers, than keep popping the smallest number from the heap till it becomes empty.

$$sort(xs) = heapSort(fromList(xs)) \tag{10.7}$$

And the real work is done in function *heapSort*.

$$heapSort(H) = \begin{cases} \phi & : & H = \phi \\ \{findMin(H)\} \cup heapSort(deleteMin(H)) & : & otherwise \end{cases} \tag{10.8}$$

Translate to Haskell yields the following program.

```
heapSort :: (Ord a) ⇒ [a] → [a]
heapSort = hsort ∘ fromList where
    hsort [] = []
    hsort h = (findMin h):(hsort $ deleteMin h)
```

Function fromList can be defined by folding. Heap sort can also be expressed in procedural way respectively. Please refer to previous chapter about binary heap for detail.

Exercise 10.4

- Write the program to return the minimum element from a binomial heap in your favorite imperative programming language with 'left-child, right-sibling' approach.
- Realize the EXTRACT-MIN-TREE() Algorithm.
- For 'left-child, right-sibling' approach, reversing all children of a tree is actually reversing a single-direct linked-list. Write a program to reverse such linked-list in your favorite imperative programming language.

More words about binomial heap

As what we have shown that insertion and merge are bound to $O(\lg N)$ time. The results are all ensure for the *worst case*. The amortized performance are $O(1)$. We skip the proof for this fact.

10.3 Fibonacci Heaps

It's interesting that why the name is given as 'Fibonacci heap'. In fact, there is no direct connection from the structure design to Fibonacci series. The inventors of 'Fibonacci heap', Michael L. Fredman and Robert E. Tarjan, utilized the property of Fibonacci series to prove the performance time bound, so they decided to use Fibonacci to name this data structure.[3]

10.3.1 Definition

Fibonacci heap is essentially a lazy evaluated binomial heap. Note that, it doesn't mean implementing binomial heap in lazy evaluation settings, for instance Haskell, brings Fibonacci heap automatically. However, lazy evaluation setting does help in realization. For example in [5], presents a elegant implementation.

Fibonacci heap has excellent performance theoretically. All operations except for pop are bound to amortized $O(1)$ time. In this section, we'll give an algorithm different from some popular textbook[3]. Most of the ideas present here are based on Okasaki's work[6].

Let's review and compare the performance of binomial heap and Fibonacci heap (more precisely, the performance goal of Fibonacci heap).

operation	Binomial heap	Fibonacci heap
insertion	$O(\lg N)$	$O(1)$
merge	$O(\lg N)$	$O(1)$
top	$O(\lg N)$	$O(1)$
pop	$O(\lg N)$	amortized $O(\lg N)$

Consider where is the bottleneck of inserting a new element x to binomial heap. We actually wrap x as a singleton leaf and insert this tree into the heap which is actually a forest.

During this operation, we inserted the tree in monotonically increasing order of rank, and once the rank is equal, recursively linking and inserting will happen, which lead to the $O(\lg N)$ time.

As the lazy strategy, we can postpone the ordered-rank insertion and merging operations. On the contrary, we just put the singleton leaf to the forest. The problem is that when we try to find the minimum element, for example the top operation, the performance will be bad, because we need check all trees in the forest, and there aren't only $O(\lg N)$ trees.

In order to locate the top element in constant time, we must remember where is the tree contains the minimum element as root.

Based on this idea, we can reuse the definition of binomial tree and give the definition of Fibonacci heap as the following Haskell program for example.

```
data BiTree a = Node { rank :: Int
                     , root :: a
                     , children :: [BiTree a]}
```

The Fibonacci heap is either empty or a forest of binomial trees with the minimum element stored in a special one explicitly.

```
data FibHeap a = E | FH { size :: Int
                        , minTree :: BiTree a
                        , trees :: [BiTree a]}
```

For convenient purpose, we also add a size field to record how many elements are there in a heap.

The data layout can also be defined in imperative way as the following ANSI C code.

```
struct node{
  Key key;
  struct node *next, *prev, *parent, *children;
  int degree; /* As known as rank */
  int mark;
};

struct FibHeap{
  struct node *roots;
  struct node *minTr;
  int n; /* number of nodes */
```

```
};
```

For generality, Key can be a customized type, we use integer for illustration purpose.

```
typedef int Key;
```

In this chapter, we use the circular doubly linked-list for imperative settings to realize the Fibonacci Heap as described in [3]. It makes many operations easy and fast. Note that, there are two extra fields added. A *degree* also known as *rank* for a node is the number of children of this node; Flag *mark* is used only in decreasing key operation. It will be explained in detail in later section.

10.3.2 Basic heap operations

As we mentioned that Fibonacci heap is essentially binomial heap implemented in a lazy evaluation strategy, we'll reuse many algorithms defined for binomial heap.

Insert a new element to the heap

Recall the insertion algorithm of binomial tree. It can be treated as a special case of merge operation, that one heap contains only a singleton tree. So that the inserting algorithm can be defined by means of merging.

$$insert(H, x) = merge(H, singleton(x)) \tag{10.9}$$

where singleton is an auxiliary function to wrap an element to a one-leaf-tree.

$$singleton(x) = FibHeap(1, node(1, x, \phi), \phi)$$

Note that function $FibHeap()$ accepts three parameters, a size value, which is 1 for this one-leaf-tree, a special tree which contains the minimum element as root, and a list of other binomial trees in the forest. The meaning of function $node()$ is as same as before, that it creates a binomial tree from a rank, an element, and a list of children.

Insertion can also be realized directly by appending the new node to the forest and updated the record of the tree which contains the minimum element.

1: **function** INSERT(H, k)
2: $x \leftarrow$ SINGLETON(k) ▷ Wrap x to a node
3: append x to root list of H
4: **if** $T_{min}(H) = NIL \vee k <$ KEY($T_{min}(H)$) **then**
5: $T_{min}(H) \leftarrow x$
6: N(H) $\leftarrow$ N(H)+1

Where function $T_{min}()$ returns the tree which contains the minimum element at root.

The following C source snippet is a translation for this algorithm.

```
struct FibHeap* insert_node(struct FibHeap* h, struct node* x){
  h = add_tree(h, x);
  if(h→minTr == NULL || x→key < h→minTr→key)
    h→minTr = x;
  h→n++;
```

```
  return h;
}
```

Exercise 10.5

Implement the insert algorithm in your favorite imperative programming language completely. This is also an exercise to circular doubly linked list manipulation.

Merge two heaps

Different with the merging algorithm of binomial heap, we post-pone the linking operations later. The idea is to just put all binomial trees from each heap together, and choose one special tree which record the minimum element for the result heap.

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ FibHeap(s_1 + s_2, T_{1min}, \{T_{2min}\} \cup \mathbb{T}_1 \cup \mathbb{T}_2) & : & root(T_{1min}) < root(T_{2min}) \\ FibHeap(s_1 + s_2, T_{2min}, \{T_{1min}\} \cup \mathbb{T}_1 \cup \mathbb{T}_2) & : & otherwise \end{cases} \tag{10.10}$$

where s_1 and s_2 are the size of H_1 and H_2; T_{1min} and T_{2min} are the special trees with minimum element as root in H_1 and H_2 respectively; $\mathbb{T}_1 = \{T_{11}, T_{12}, ...\}$ is a forest contains all other binomial trees in H_1; while $\mathbb{T}_2$ has the same meaning as $\mathbb{T}_1$ except that it represents the forest in H_2. Function $root(T)$ return the root element of a binomial tree.

Note that as long as the $\cup$ operation takes constant time, these *merge* algorithm is bound to $O(1)$. The following Haskell program is the translation of this algorithm.

```
merge:: (Ord a) ⇒ FibHeap a → FibHeap a → FibHeap a
merge h E = h
merge E h = h
merge h1@(FH sz1 minTr1 ts1) h2@(FH sz2 minTr2 ts2)
    | root minTr1 < root minTr2 = FH (sz1+sz2) minTr1 (minTr2:ts2++ts1)
    | otherwise = FH (sz1+sz2) minTr2 (minTr1:ts1++ts2)
```

Merge algorithm can also be realized imperatively by concatenating the root lists of the two heaps.

```
1: function MERGE(H1, H2)
2:     H ← Φ
3:     ROOT(H) ← CONCAT(ROOT(H1), ROOT(H2))
4:     if KEY(Tmin(H1)) < KEY(Tmin(H2)) then
5:         Tmin(H) ← Tmin(H1)
6:     else
7:         Tmin(H) ← Tmin(H2)
       N(H) = N(H1) + N(H2)
8:     release H1 and H2
9:     return H
```

This function assumes neither H_1, nor H_2 is empty. And it's easy to add handling to these special cases as the following ANSI C program.

```
struct FibHeap* merge(struct FibHeap* h1, struct FibHeap* h2){
  struct FibHeap* h;
  if(is_empty(h1))
    return h2;
  if(is_empty(h2))
    return h1;
  h = empty();
  h→roots = concat(h1→roots, h2→roots);
  if(h1→minTr→key < h2→minTr→key)
    h→minTr = h1→minTr;
  else
    h→minTr = h2→minTr;
  h→n = h1→n + h2→n;
  free(h1);
  free(h2);
  return h;
}
```

With *merge* function defined, the $O(1)$ insertion algorithm is realized as well. And we can also give the $O(1)$ time top function as below.

$$top(H) = root(T_{min}) \tag{10.11}$$

Exercise 10.6

Implement the circular doubly linked list concatenation function in your favorite imperative programming language.

Extract the minimum element from the heap (pop)

The pop (delete the minimum element) operation is the most complex one in Fibonacci heap. Since we postpone the tree consolidation in merge algorithm. We have to compensate it somewhere. Pop is the only place left as we have defined, insert, merge, top already.

There is an elegant procedural algorithm to do the tree consolidation by using an auxiliary array[3]. We'll show it later in imperative approach section.

In order to realize the purely functional consolidation algorithm, let's first consider a similar number puzzle.

Given a list of numbers, such as $\{2, 1, 1, 4, 8, 1, 1, 2, 4\}$, we want to add any two values if they are same. And repeat this procedure till all numbers are unique. The result of the example list should be $\{8, 16\}$ for instance.

One solution to this problem will as the following.

$$consolidate(L) = fold(meld, \phi, L) \tag{10.12}$$

Where $fold()$ function is defined to iterate all elements from a list, applying a specified function to the intermediate result and each element. it is sometimes called as *reducing*. Please refer to the chapter of binary search tree for it.

$L = \{x_1, x_2, ..., x_n\}$, denotes a list of numbers; and we'll use $L' = \{x_2, x_3, ..., x_n\}$ to represent the rest of the list with the first element removed. Function $meld()$

Table 10.1: Steps of consolidate numbers

number	intermediate result	result
2	2	2
1	1, 2	1, 2
1	(1+1), 2	4
4	(4+4)	8
8	(8+8)	16
1	1, 16	1, 16
1	(1+1), 16	2, 16
2	(2+2), 16	4, 16
4	(4+4), 16	8, 16

is defined as below.

$$meld(L, x) = \begin{cases} \{x\} & : & L = \phi \\ meld(L', x + x_1) & : & x = x_1 \\ \{x\} \cup L & : & x < x_1 \\ \{x_1\} \cup meld(L', x) & : & otherwise \end{cases} \tag{10.13}$$

The *consolidate*() function works as the follows. It maintains an ordered result list L, contains only unique numbers, which is initialized from an empty list ϕ. Each time it process an element x, it firstly check if the first element in L is equal to x, if so, it will add them together (which yields $2x$), and repeatedly check if $2x$ is equal to the next element in L. This process won't stop until either the element to be melt is not equal to the head element in the rest of the list, or the list becomes empty. Table 10.1 illustrates the process of consolidating number sequence $\{2, 1, 1, 4, 8, 1, 1, 2, 4\}$. Column one lists the number 'scanned' one by one; Column two shows the intermediate result, typically the new scanned number is compared with the first number in result list. If they are equal, they are enclosed in a pair of parentheses; The last column is the result of meld, and it will be used as the input to next step processing.

The Haskell program can be give accordingly.

```
consolidate = foldl meld [] where
    meld [] x = [x]
    meld (x':xs) x | x == x' = meld xs (x+x')
                   | x < x'  = x:x':xs
                   | otherwise = x': meld xs x
```

We'll analyze the performance of consolidation as a part of pop operation in later section.

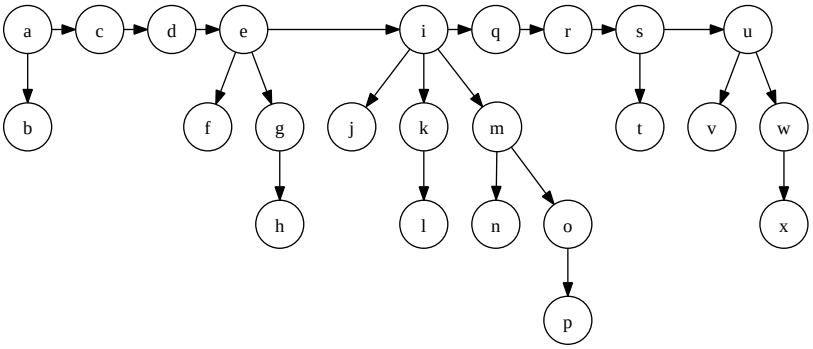
The tree consolidation is very similar to this algorithm except it performs based on rank. The only thing we need to do is to modify *meld*() function a bit, so that it compare on ranks and do linking instead of adding.

$$meld(L, x) = \begin{cases} \{x\} & : & L = \phi \\ meld(L', link(x, x_1)) & : & rank(x) = rank(x_1) \\ \{x\} \cup L & : & rank(x) < rank(x_1) \\ \{x_1\} \cup meld(L', x) & : & otherwise \end{cases} \tag{10.14}$$

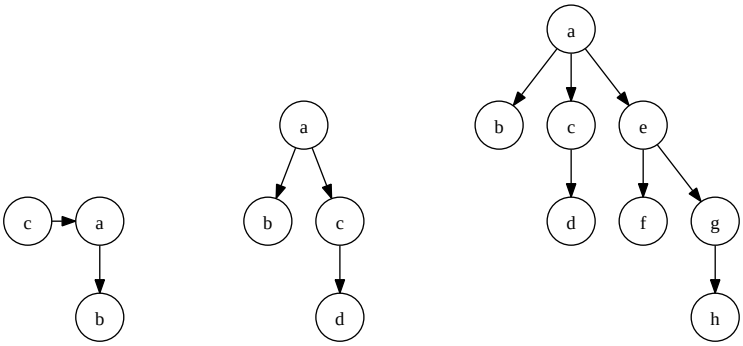
The final consolidate Haskell program changes to the below version.

```
consolidate :: (Ord a) ⇒ [BiTree a] → [BiTree a]
consolidate = foldl meld [] where
    meld [] t = [t]
    meld (t':ts) t | rank t == rank t' = meld ts (link t t')
                   | rank t <  rank t' = t:t':ts
                   | otherwise = t' : meld ts t
```

Figure 10.9 and 10.10 show the steps of consolidation when processing a Fibonacci Heap contains different ranks of trees. Comparing with table 10.1 reveals the similarity.



(a) Before consolidation



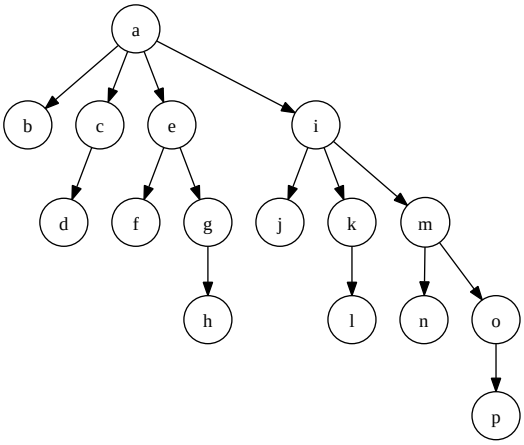
(b) Step 1, 2 (c) Step 3, 'd' is firstly linked to 'c', then repeatedly linked to 'a'. (d) Step 4

Figure 10.9: Steps of consolidation

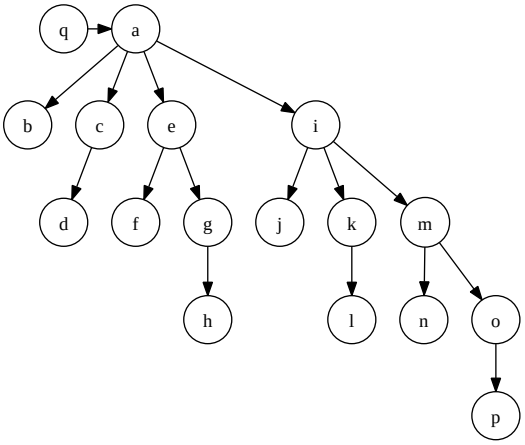
After we merge all binomial trees, including the special tree record for the minimum element in root, in a Fibonacci heap, the heap becomes a Binomial heap. And we lost the special tree, which gives us the ability to return the top element in $O(1)$ time.

It's necessary to perform a $O(\lg N)$ time search to resume the special tree. We can reuse the function $extractMin()$ defined for Binomial heap.

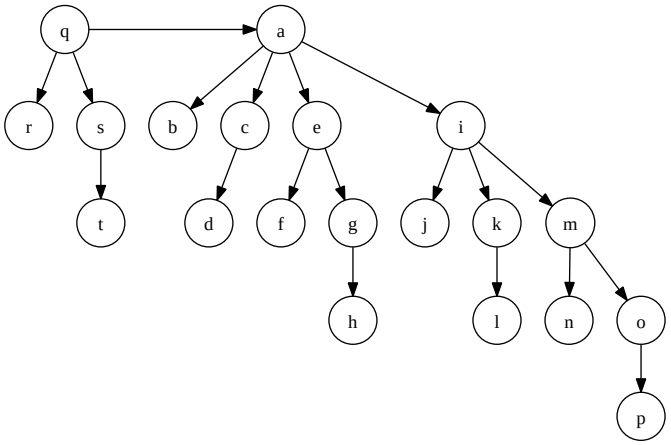
It's time to give the final pop function for Fibonacci heap as all the sub problems have been solved. Let T_{min} denote the special tree in the heap to record the minimum element in root; $\mathbb{T}$ denote the forest contains all the other trees except for the special tree, s represents the size of a heap, and function



(a) Step 5



(b) Step 6



(c) Step 7, 8, 'r' is firstly linked to 'q', then 's' is linked to 'q'.

Figure 10.10: Steps of consolidation

$children()$ returns all sub trees except the root of a binomial tree.

$$deleteMin(H) = \begin{cases} \phi & : \mathbb{T} = \phi \wedge children(T_{min}) = \phi \\ FibHeap(s-1, T'_{min}, \mathbb{T}') & : otherwise \end{cases} \tag{10.15}$$

Where

$$(T'_{min}, \mathbb{T}') = extractMin(consolidate(children(T_{min}) \cup \mathbb{T}))$$

Translate to Haskell yields the below program.

```
deleteMin :: (Ord a) ⇒ FibHeap a → FibHeap a
deleteMin (FH _ (Node _ x []) []) = E
deleteMin h@(FH sz minTr ts) = FH (sz-1) minTr' ts' where
    (minTr', ts') = extractMin $ consolidate (children minTr ++ ts)
```

The main part of the imperative realization is similar. We cut all children of T_{min} and append them to root list, then perform consolidation to merge all trees with the same rank until all trees are unique in term of rank.

```
1: function DELETE-MIN(H)
2:     x ← T_min(H)
3:     if x ≠ NIL then
4:         for each y ∈ CHILDREN(x) do
5:             append y to root list of H
6:             PARENT(y) ← NIL
7:         remove x from root list of H
8:         N(H) ← N(H) - 1
9:         CONSOLIDATE(H)
10:    return x
```

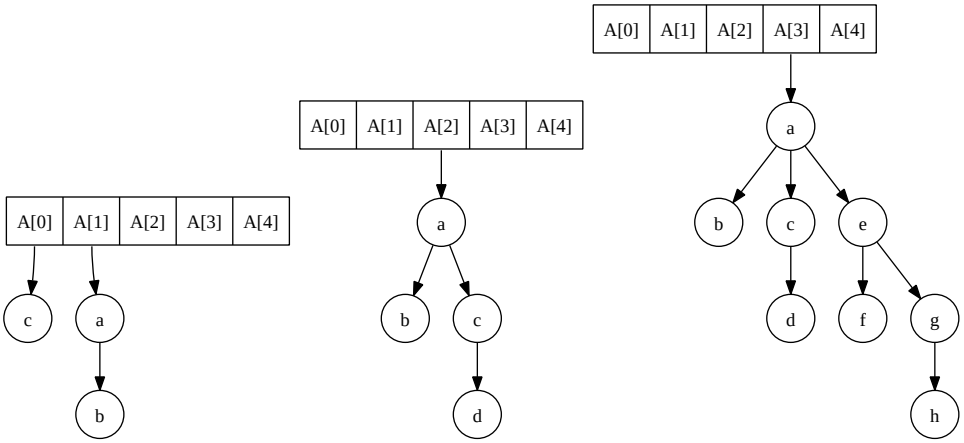
Algorithm CONSOLIDATE utilizes an auxiliary array A to do the merge job. Array $A[i]$ is defined to store the tree with rank (degree) i. During the traverse of root list, if we meet another tree of rank i, we link them together to get a new tree of rank $i+1$. Next we clean $A[i]$, and check if $A[i+1]$ is empty and perform further linking if necessary. After we finish traversing all roots, array A stores all result trees and we can re-construct the heap from it.

```
1: function CONSOLIDATE(H)
2:     D ← MAX-DEGREE(N(H))
3:     for i ← 0 to D do
4:         A[i] ← NIL
5:     for each x ∈ root list of H do
6:         remove x from root list of H
7:         d ← DEGREE(x)
8:         while A[d] ≠ NIL do
9:             y ← A[d]
10:            x ← LINK(x, y)
11:            A[d] ← NIL
12:            d ← d + 1
13:        A[d] ← x
14:    T_min(H) ← NIL                    ▷ root list is NIL at the time
15:    for i ← 0 to D do
```

16: **if** $A[i] \neq NIL$ **then**
17: append $A[i]$ to root list of H.
18: **if** $T_{min} = NIL \vee$ KEY$(A[i]) <$ KEY$(T_{min}(H))$ **then**
19: $T_{min}(H) \leftarrow A[i]$

The only unclear sub algorithm is MAX-DEGREE, which can determine the upper bound of the degree of any node in a Fibonacci Heap. We'll delay the realization of it to the last sub section.

Feed a Fibonacci Heap shown in Figure 10.9 to the above algorithm, Figure 10.11, 10.12 and 10.13 show the result trees stored in auxiliary array A in every steps.



(a) Step 1, 2

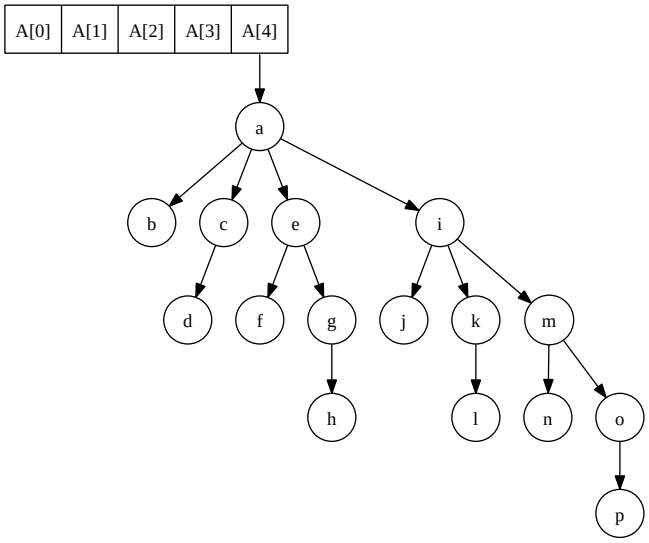
(b) Step 3, Since $A_0 \neq NIL$, 'd' is firstly linked to 'c', and clear A_0 to NIL. Again, as $A_1 \neq NIL$, 'c' is linked to 'a' and the new tree is stored in A_2.

(c) Step 4

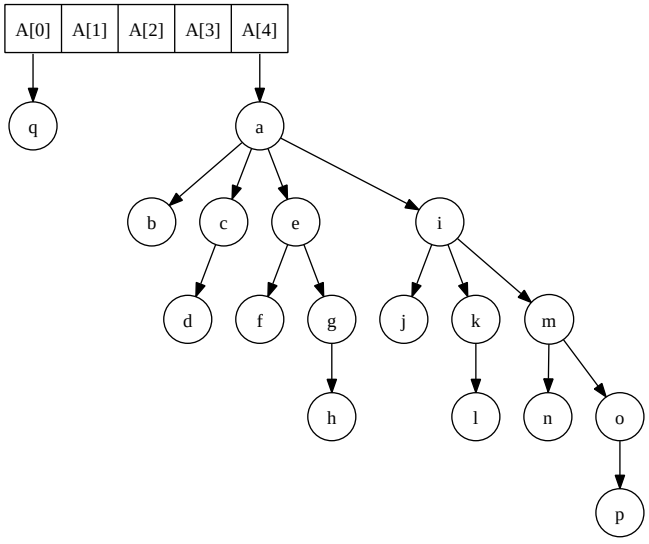
Figure 10.11: Steps of consolidation

Translate the above algorithm to ANSI C yields the below program.

```
void consolidate(struct FibHeap* h){
  if(!h→roots)
    return;
  int D = max_degree(h→n)+1;
  struct node *x, *y;
  struct node** a = (struct node**)malloc(sizeof(struct node*)*(D+1));
  int i, d;
  for(i=0; i≤D; ++i)
    a[i] = NULL;
  while(h→roots){
    x = h→roots;
    h→roots = remove_node(h→roots, x);
    d= x→degree;
    while(a[d]){
      y = a[d];  /* Another node has the same degree as x */
      x = link(x, y);
```

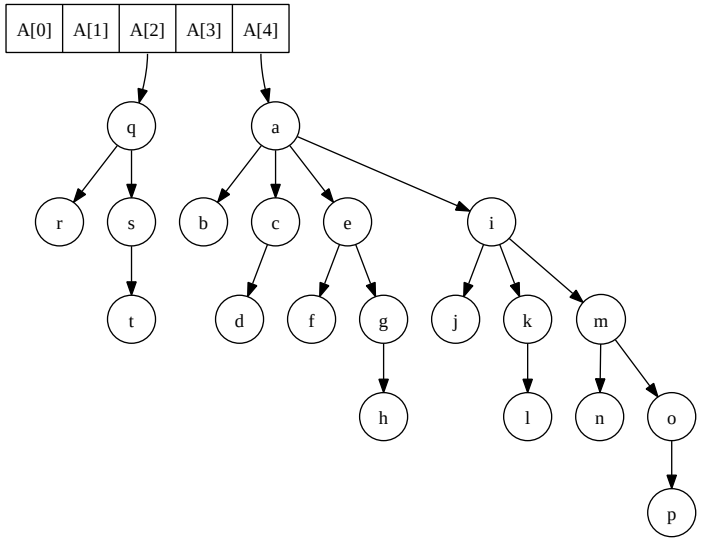


(a) Step 5



(b) Step 6

Figure 10.12: Steps of consolidation



(a) Step 7, 8, Since $A_0 \neq NIL$, 'r' is firstly linked to 'q', and the new tree is stored in A_1 (A_0 is cleared); then 's' is linked to 'q', and stored in A_2 (A_1 is cleared).

Figure 10.13: Steps of consolidation

```
      a[d++] = NULL;
    }
    a[d] = x;
  }
  h→minTr = h→roots = NULL;
  for(i=0; i≤D; ++i)
    if(a[i]){
      h→roots = append(h→roots, a[i]);
      if(h→minTr == NULL || a[i]→key < h→minTr→key)
        h→minTr = a[i];
    }
  free(a);
}
```

Exercise 10.7

Implement the remove function for circular doubly linked list in your favorite imperative programming language.

10.3.3 Running time of pop

In order to analyze the amortize performance of pop, we adopt potential method. Reader can refer to [3] for a formal definition. In this chapter, we only give a intuitive illustration.

Remind the gravity potential energy, which is defined as

$$E = M \cdot g \cdot h$$

Suppose there is a complex process, which moves the object with mass M up and down, and finally the object stop at height h'. And if there exists friction resistance W_f, We say the process works the following power.

$$W = M \cdot g \cdot (h' - h) + W_f$$

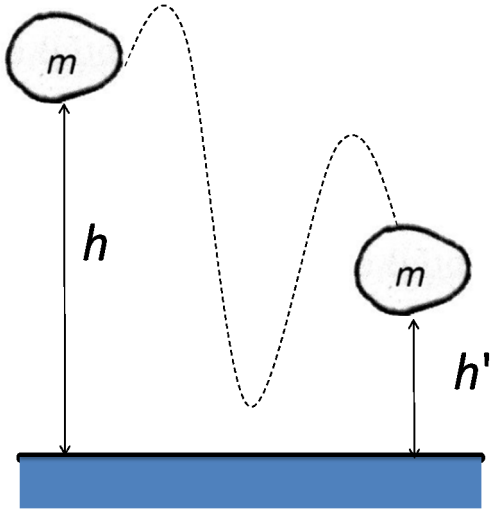


Figure 10.14: Gravity potential energy.

Figure 10.14 illustrated this concept.

We treat the Fibonacci heap pop operation in a similar way, in order to evaluate the cost, we firstly define the potential $\Phi(H)$ before extract the minimum element. This potential is accumulated by insertion and merge operations executed so far. And after tree consolidation and we get the result H', we then calculate the new potential $\Phi(H')$. The difference between $\Phi(H')$ and $\Phi(H)$ plus the contribution of consolidate algorithm indicates the amortized performance of pop.

For pop operation analysis, the potential can be defined as

$$\Phi(H) = t(H) \tag{10.16}$$

Where $t(H)$ is the number of trees in Fibonacci heap forest. We have $t(H) = 1 + length(\mathbb{T})$ for any non-empty heap.

For the N-nodes Fibonacci heap, suppose there is an upper bound of ranks for all trees as $D(N)$. After consolidation, it ensures that the number of trees in the heap forest is at most $D(N) + 1$.

Before consolidation, we actually did another important thing, which also contribute to running time, we removed the root of the minimum tree, and concatenate all children left to the forest. So consolidate operation at most processes $D(N) + t(H) - 1$ trees.

Summarize all the above factors, we deduce the amortized cost as below.

$$\begin{aligned} T &= T_{consolidation} + \Phi(H') - \Phi(H) \\ &= O(D(N) + t(H) - 1) + (D(N) + 1) - t(H) \\ &= O(D(N)) \end{aligned} \tag{10.17}$$

If only insertion, merge, and pop function are applied to Fibonacci heap. We ensure that all trees are binomial trees. It is easy to estimate the upper limit $D(N)$ if $O(\lg N)$. (Suppose the extreme case, that all nodes are in only one Binomial tree).

However, we'll show in next sub section that, there is operation can violate the binomial tree assumption.

10.3.4 Decreasing key

There is a special heap operation left. It only makes sense for imperative settings. It's about decreasing key of a certain node. Decreasing key plays important role in some Graphic algorithms such as Minimum Spanning tree algorithm and Dijkstra's algorithm [3]. In that case we hope the decreasing key takes O(1) amortized time.

However, we can't define a function like $Decrease(H, k, k')$, which first locates a node with key k, then decrease k to k' by replacement, and then resume the heap properties. This is because the time for locating phase is bound to $O(N)$ time, since we don't have a pointer to the target node.

In imperative setting, we can define the algorithm as DECREASE-KEY(H, x, k). Here x is a node in heap H, which we want to decrease its key to k. We needn't perform a search, as we have x at hand. It's possible to give an amortized $O(1)$ solution.

When we decreased the key of a node, if it's not a root, this operation may violate the property Binomial tree that the key of parent is less than all keys of children. So we need to compare the decreased key with the parent node, and if this case happens, we can cut this node and append it to the root list. (Remind the recursive swapping solution for binary heap which leads to $O(\lg N)$)

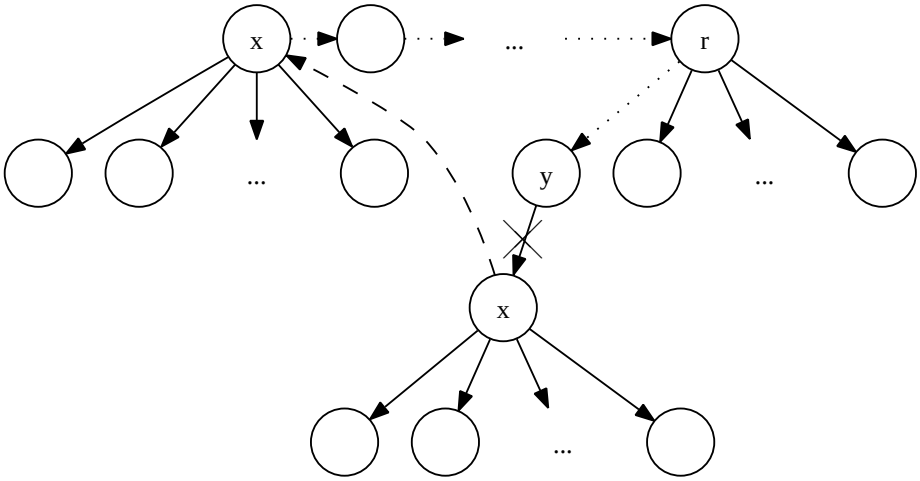


Figure 10.15: $x < y$, cut tree x from its parent, and add x to root list.

Figure 10.15 illustrates this situation. After decreasing key of node x, it is less than y, we cut x off its parent y, and 'past' the whole tree rooted at x to root list.

Although we recover the property of that parent is less than all children, the tree isn't any longer a Binomial tree after it losses some sub tree. If a tree losses too many of its children because of cutting, we can't ensure the performance of merge-able heap operations. Fibonacci Heap adds another constraints to avoid such problem:

If a node losses its second child, it is immediately cut from parent, and added to root list

The final DECREASE-KEY algorithm is given as below.

```
1: function DECREASE-KEY(H, x, k)
2:     KEY(x) ← k
3:     p ← PARENT(x)
4:     if p ≠ NIL ∧ k < KEY(p) then
5:         CUT(H, x)
6:         CASCADING-CUT(H, p)
7:     if k < KEY(T_min(H)) then
8:         T_min(H) ← x
```

Where function CASCADING-CUT uses the mark to determine if the node is losing the second child. the node is marked after it losses the first child. And the mark is cleared in CUT function.

```
1: function CUT(H, x)
2:     p ← PARENT(x)
3:     remove x from p
4:     DEGREE(p) ← DEGREE(p) - 1
5:     add x to root list of H
6:     PARENT(x) ← NIL
7:     MARK(x) ← FALSE
```

During cascading cut process, if x is marked, which means it has already lost one child. We recursively performs cut and cascading cut on its parent till reach to root.

```
1: function CASCADING-CUT(H, x)
2:     p ← PARENT(x)
3:     if p ≠ NIL then
4:         if MARK(x) = FALSE then
5:             MARK(x) ← TRUE
6:         else
7:             CUT(H, x)
8:             CASCADING-CUT(H, p)
```

The relevant ANSI C decreasing key program is given as the following.

```
void decrease_key(struct FibHeap* h, struct node* x, Key k){
  struct node* p = x→parent;
  x→key = k;
  if(p && k < p→key){
    cut(h, x);
    cascading_cut(h, p);
  }
  if(k < h→minTr→key)
    h→minTr = x;
}
```

```
void cut(struct FibHeap* h, struct node* x){
  struct node* p = x→parent;
  p→children = remove_node(p→children, x);
  p→degree--;
  h→roots = append(h→roots, x);
  x→parent = NULL;
  x→mark = 0;
}

void cascading_cut(struct FibHeap* h, struct node* x){
  struct node* p = x→parent;
  if(p){
    if(!x→mark)
      x→mark = 1;
    else{
      cut(h, x);
      cascading_cut(h, p);
    }
  }
}
```

Exercise 10.8

Prove that DECREASE-KEY algorithm is amortized $O(1)$ time.

10.3.5 The name of Fibonacci Heap

It's time to reveal the reason why the data structure is named as 'Fibonacci Heap'.

There is only one undefined algorithm so far, MAX-DEGREE(N). Which can determine the upper bound of degree for any node in a N nodes Fibonacci Heap. We'll give the proof by using Fibonacci series and finally realize MAX-DEGREE algorithm.

Lemma 10.3.1. *For any node x in a Fibonacci Heap, denote $k = degree(x)$, and $|x| = size(x)$, then*

$$|x| \geq F_{k+2} \tag{10.18}$$

Where F_k is Fibonacci series defined as the following.

$$F_k = \begin{cases} 0 & : \quad k = 0 \\ 1 & : \quad k = 1 \\ F_{k-1} + F_{k-2} & : \quad k \geq 2 \end{cases}$$

Proof. Consider all k children of node x, we denote them as $y_1, y_2, ..., y_k$ in the order of time when they were linked to x. Where y_1 is the oldest, and y_k is the youngest.

Obviously, $y_i \geq 0$. When we link y_i to x, children $y_1, y_2, ..., y_{i-1}$ have already been there. And algorithm LINK only links nodes with the same degree. Which indicates at that time, we have

$$degree(y_i) = degree(x) = i - 1$$

After that, node y_i can at most lost 1 child, (due to the decreasing key operation) otherwise, if it will be immediately cut off and append to root list after the second child loss. Thus we conclude

$$degree(y_i) \geq i - 2$$

For any $i = 2, 3, ..., k$.

Let s_k be the *minimum possible size* of node x, where $degree(x) = k$. For trivial cases, $s_0 = 1$, $s_1 = 2$, and we have

$$\begin{aligned} |x| &\geq s_k \\ &= 2 + \sum_{i=2}^{k} s_{degree(y_i)} \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \end{aligned}$$

We next show that $s_k > F_{k+2}$. This can be proved by induction. For trivial cases, we have $s_0 = 1 \geq F_2 = 1$, and $s_1 = 2 \geq F_3 = 2$. For induction case $k \geq 2$. We have

$$\begin{aligned} |x| &\geq s_k \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i \end{aligned}$$

At this point, we need prove that

$$F_{k+2} = 1 + \sum_{i=}^{k} F_i \tag{10.19}$$

This can also be proved by using induction:

- Trivial case, $F_2 = 1 + F_0 = 2$
- Induction case,

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &= 1 + \sum_{i=0}^{k-1} F_i + F_k \\ &= 1 + \sum_{i=0}^{k} F_i \end{aligned}$$

Summarize all above we have the final result.

$$N \geq |x| \geq F_k + 2 \tag{10.20}$$

□

Recall the result of AVL tree, that $F_k \geq \Phi^k$, where $\Phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. We also proved that pop operation is amortized $O(\lg N)$ algorithm.

Based on this result. We can define Function $MaxDegree$ as the following.

$$MaxDegree(N) = 1 + \lfloor \log_\Phi N \rfloor \tag{10.21}$$

The imperative MAX-DEGREE algorithm can also be realized by using Fibonacci sequences.

```
1: function MAX-DEGREE(N)
2:     F_0 ← 0
3:     F_1 ← 1
4:     k ← 2
5:     repeat
6:         F_k ← F_{k_1} + F_{k_2}
7:         k ← k + 1
8:     until F_k < N
9:     return k − 2
```

Translate the algorithm to ANSI C given the following program.

```
int max_degree(int n){
  int k, F;
  int F2 = 0;
  int F1 = 1;
  for(F=F1+F2, k=2; F<n; ++k){
    F2 = F1;
    F1 = F;
    F = F1 + F2;
  }
  return k-2;
}
```

10.4 Pairing Heaps

Although Fibonacci Heaps provide excellent performance theoretically, it is complex to realize. People find that the constant behind the big-O is big. Actually, Fibonacci Heap is more significant in theory than in practice.

In this section, we'll introduce another solution, Pairing heap, which is one of the best heaps ever known in terms of performance. Most operations including insertion, finding minimum element (top), merging are all bounds to $O(1)$ time, while deleting minimum element (pop) is conjectured to amortized $O(\lg N)$ time [7] [6]. Note that this is still a conjecture for 15 years by the time I write this chapter. Nobody has been proven it although there are much experimental data support the $O(\lg N)$ amortized result.

Besides that, pairing heap is simple. There exist both elegant imperative and functional implementations.

10.4.1 Definition

Both Binomial Heaps and Fibonacci Heaps are realized with forest. While a pairing heaps is essentially a K-ary tree. The minimum element is stored at root. All other elements are stored in sub trees.

The following Haskell program defines pairing heap.

```
data PHeap a = E | Node a [PHeap a]
```

This is a recursive definition, that a pairing heap is either empty or a K-ary tree, which is consist of a root node, and a list of sub trees.

Pairing heap can also be defined in procedural languages, for example ANSI C as below. For illustration purpose, all heaps we mentioned later are minimum-heap, and we assume the type of key is integer [4]. We use same linked-list based left-child, right-sibling approach (aka, binary tree representation[3]).

```
typedef int Key;

struct node{
  Key key;
  struct node *next, *children, *parent;
};
```

Note that the parent field does only make sense for decreasing key operation, which will be explained later on. we can omit it for the time being.

10.4.2 Basic heap operations

In this section, we first give the merging operation for pairing heap, which can be used to realize the insertion. Merging, insertion, and finding the minimum element are relative trivial compare to the extracting minimum element operation.

Merge, insert, and find the minimum element (top)

The idea of merging is similar to the linking algorithm we shown previously for Binomial heap. When we merge two pairing heaps, there are two cases.

- Trivial case, one heap is empty, we simply return the other heap as the result;
- Otherwise, we compare the root element of the two heaps, make the heap with bigger root element as a new children of the other.

Let H_1, and H_2 denote the two heaps, x and y be the root element of H_1 and H_2 respectively. Function $Children()$ returns the children of a K-ary tree. Function $Node()$ can construct a K-ary tree from a root element and a list of children.

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ Node(x, \{H_2\} \cup Children(H_1)) & : & x < y \\ Node(y, \{H_1\} \cup Children(H_2)) & : & otherwise \end{cases} \tag{10.22}$$

[4] We can parametrize the key type with C++ template, but this is beyond our scope, please refer to the example programs along with this book

Where

$$x = Root(H_1)$$
$$y = Root(H_2)$$

It's obviously that merging algorithm is bound to $O(1)$ time [5]. The *merge* equation can be translated to the following Haskell program.

```
merge :: (Ord a) ⇒ PHeap a → PHeap a → PHeap a
merge h E = h
merge E h = h
merge h1@(Node x hs1) h2@(Node y hs2) =
    if x < y then Node x (h2:hs1) else Node y (h1:hs2)
```

Merge can also be realized imperatively. With left-child, right sibling approach, we can just link the heap, which is in fact a K-ary tree, with larger key as the first new child of the other. This is constant time operation as described below.

1: **function** MERGE(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: Insert H_2 in front of CHILDREN(H_1)
9: PARENT(H_2) $\leftarrow H_1$
10: **return** H_1

Note that we also update the parent field accordingly. The ANSI C example program is given as the following.

```
struct node* merge(struct node* h1, struct node* h2){
  if(h1 == NULL)
    return h2;
  if(h2 == NULL)
    return h1;
  if(h2→key < h1→key)
    swap(&h1, &h2);
  h2→next = h1→children;
  h1→children = h2;
  h2→parent = h1;
  h1→next = NULL; /*Break previous link if any*/
  return h1;
}
```

Where function swap() is defined in a similar way as Fibonacci Heap.

With merge defined, insertion can be realized as same as Fibonacci Heap in Equation 10.9. Definitely it's $O(1)$ time operation. As the minimum element is always stored in root, finding it is trivial.

$$top(H) = Root(H) \tag{10.23}$$

[5] Assume $\cup$ is constant time operation, this is true for linked-list settings, including 'cons' like operation in functional programming languages.

Same as the other two above operations, it's bound to $O(1)$ time.

Exercise 10.9

Implement the insertion and top operation in your favorite programming language.

Decrease key of a node

There is another trivial operation, to decrease key of a given node, which only makes sense in imperative settings as we explained in Fibonacci Heap section.

The solution is simple, that we can cut the node with the new smaller key from it's parent along with all its children. Then merge it again to the heap. The only special case is that if the given node is the root, then we can directly set the new key without doing anything else.

The following algorithm describes this procedure for a given node x, with new key k.

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

The following ANSI C program translates this algorithm.

```
struct node* decrease_key(struct node* h, struct node* x, Key key){
  x→key = key; /* Assume key ≤ x→key */
  if(x→parent)
    x→parent→children = remove_node(x→parent→children, x);
  x→parent = NULL;
  return merge(h, x);
}
```

Exercise 10.10

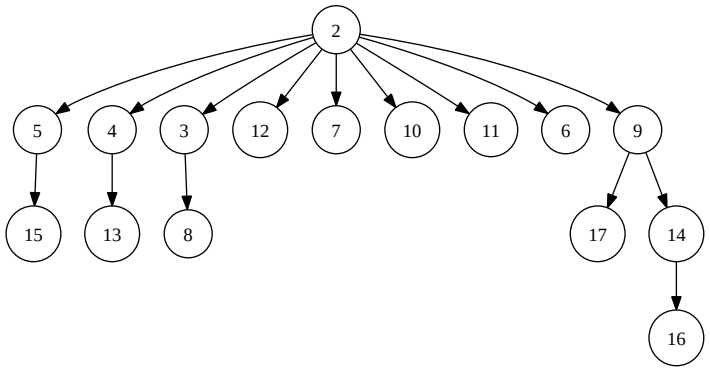
Implement the program of removing a node from the children of its parent in your favorite imperative programming language. Consider how can we ensure the overall performance of decreasing key is O(1) time? Is left-child, right sibling approach enough?

Delete the minimum element from the heap (pop)

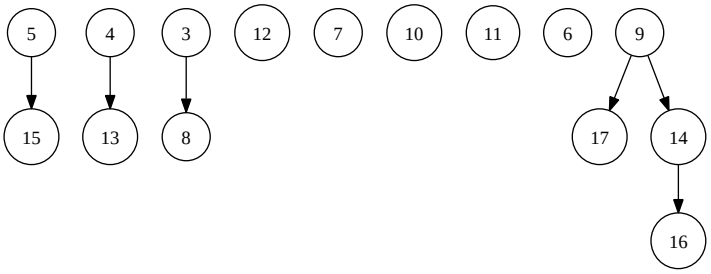
Since the minimum element is always stored at root, after delete it during popping, the rest things left are all sub-trees. These trees can be merged to one big tree.

$$pop(H) = mergePairs(Children(H)) \tag{10.24}$$

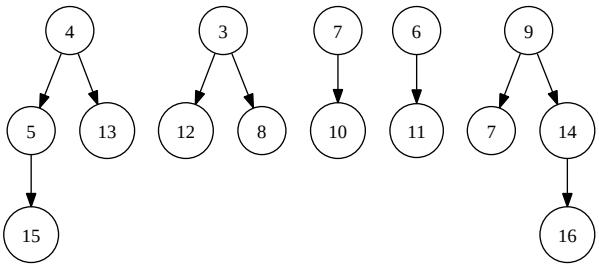
Pairing Heap uses a special approach that it merges every two sub-trees from left to right in pair. Then merge these paired results from right to left which forms a final result tree. The name of 'Pairing Heap' comes from the characteristic of this pair-merging.



(a) A pairing heap before pop.

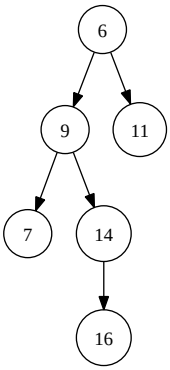


(b) After root element 2 being removed, there are 9 sub-trees left.

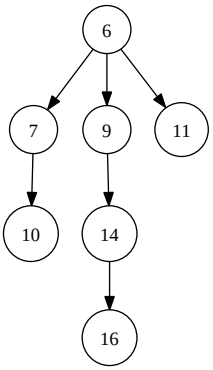


(c) Merge every two trees in pair, note that there are odd number trees, so the last one needn't merge.

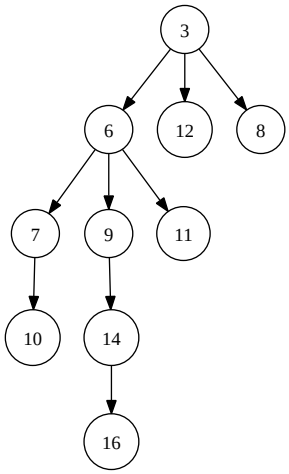
Figure 10.16: Remove the root element, and merge children in pairs.



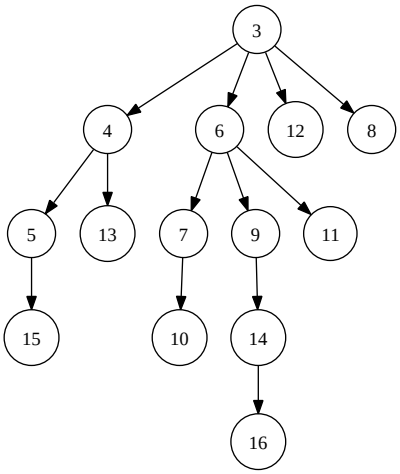
(a) Merge tree with 9, and tree with root 6.



(b) Merge tree with root 7 to the result.



(c) Merge tree with root 3 to the result.



(d) Merge tree with root 4 to the result.

Figure 10.17: Steps of merge from right to left.

Figure 10.16 and 10.17 illustrate the procedure of pair-merging.

The recursive pair-merging solution is quite similar to the bottom up merge sort[6]. Denote the children of a pairing heap as A, which is a list of trees of $\{T_1, T_2, T_3, ..., T_m\}$ for example. The $mergePairs()$ function can be given as below.

$$mergePairs(A) = \begin{cases} \Phi & : & A = \Phi \\ T_1 & : & A = \{T_1\} \\ merge(merge(T_1, T_2), mergePairs(A')) & : & otherwise \end{cases} \tag{10.25}$$

where

$$A' = \{T_3, T_4, ..., T_m\}$$

is the rest of the children without the first two trees.

The relative Haskell program of popping is given as the following.

```
deleteMin :: (Ord a) ⇒ PHeap a → PHeap a
deleteMin (Node _ hs) = mergePairs hs where
    mergePairs [] = E
    mergePairs [h] = h
    mergePairs (h1:h2:hs) = merge (merge h1 h2) (mergePairs hs)
```

The popping operation can also be explained in the following procedural algorithm.

1: **function** POP(H)
2: $L \leftarrow NIL$
3: **for** every 2 trees T_x, $T_y \in$ CHILDREN(H) from left to right **do**
4: Extract x, and y from CHILDREN(H)
5: $T \leftarrow$ MERGE(T_x, T_y)
6: Insert T at the beginning of L
7: $H \leftarrow$ CHILDREN(H) ▷ H is either NIL or one tree.
8: **for** $\forall T \in L$ from left to right **do**
9: $H \leftarrow$ MERGE(H, T)
10: **return** H

Note that L is initialized as an empty linked-list, then the algorithm iterates every two trees in pair in the children of the K-ary tree, from left to right, and performs merging, the result is inserted at the beginning of L. Because we insert to front end, so when we traverse L later on, we actually process from right to left. There may be odd number of sub-trees in H, in that case, it will leave one tree after pair-merging. We handle it by start the right to left merging from this left tree.

Below is the ANSI C program to this algorithm.

```
struct node* pop(struct node* h){
  struct node *x, *y, *lst = NULL;
  while((x = h→children) != NULL){
    if((h→children = y = x→next) != NULL)
      h→children = h→children→next;
    lst = push_front(lst, merge(x, y));
  }
  x = NULL;
```

```
  while((y = lst) != NULL){
    lst = lst→next;
    x = merge(x, y);
  }
  free(h);
  return x;
}
```

The pairing heap pop operation is conjectured to be amortized $O(\lg N)$ time [7].

Exercise 10.11

Write a program to insert a tree at the beginning of a linked-list in your favorite imperative programming language.

Delete a node

We didn't mention delete in Binomial heap or Fibonacci Heap. Deletion can be realized by first decreasing key to minus infinity ($-\infty$), then performing pop. In this section, we present another solution for delete node.

The algorithm is to define the function $delete(H, x)$, where x is a node in a pairing heap H [6].

If x is root, we can just perform a pop operation. Otherwise, we can cut x from H, perform a pop on x, and then merge the pop result back to H. This can be described as the following.

$$delete(H, x) = \begin{cases} pop(H) & : \quad x \quad \text{is root of} \quad H \\ merge(cut(H, x), pop(x)) & : \quad otherwise \end{cases} \tag{10.26}$$

As delete algorithm uses pop, the performance is conjectured to be amortized $O(1)$ time.

Exercise 10.12

- Write procedural pseudo code for delete algorithm.
- Write the delete operation in your favorite imperative programming language
- Consider how to realize delete in purely functional setting.

10.5 Notes and short summary

In this chapter, we extend the heap implementation from binary tree to more generic approach. Binomial heap and Fibonacci heap use Forest of K-ary trees as under ground data structure, while Pairing heap use a K-ary tree to represent heap. It's a good point to post pone some expensive operation, so that the over all amortized performance is ensured. Although Fibonacci Heap gives good

[6] Here the semantic of x is a reference to a node.

performance in theory, the implementation is a bit complex. It was removed in some latest textbooks. We also present pairing heap, which is easy to realize and have good performance in practice.

The elementary tree based data structures are all introduced in this book. There are still many tree based data structures which we can't covers them all and skip here. We encourage the reader to refer to other textbooks about them. From next chapter, we'll introduce generic sequence data structures, array and queue.

Bibliography

[1] K-ary tree, Wikipedia. http://en.wikipedia.org/wiki/K-ary_tree

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

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

[4] Wikipedia, "Pascal's triangle". http://en.wikipedia.org/wiki/Pascal's_triangle

[5] Hackage. "An alternate implementation of a priority queue based on a Fibonacci heap.", http://hackage.haskell.org/packages/archive/pqueue-mtl/1.0.7/doc/html/src/Data-Queue-FibQueue.html

[6] Chris Okasaki. "Fibonacci Heaps." http://darcs.haskell.org/nofib/gc/fibheaps/orig

[7] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. "The Pairing Heap: A New Form of Self-Adjusting Heap" Algorithmica (1986) 1: 111-129.

Part III

Queues and Sequences

Queue, not so simple as it was thought

Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

Chapter 11

Queue, not so simple as it was thought

11.1 Introduction

It seems that queues are relative simple. A queue provides FIFO (first-in, first-out) data manipulation support. There are many options to realize queue includes singly linked-list, doubly linked-list, circular buffer etc. However, we'll show that it's not so easy to realize queue in purely functional settings if it must satisfy abstract queue properties.

In this chapter, we'll present several different approaches to implement queue. And in next chapter, we'll explain how to realize sequence.

A queue is a FIFO data structure satisfies the following performance constraints.

- Element can be added to the tail of the queue in $O(1)$ constant time;
- Element can be removed from the head of the queue in $O(1)$ constant time.

These two properties must be satisfied. And it's common to add some extra goals, such as dynamic memory allocation etc.

Of course such abstract queue interface can be implemented with doubly-linked list trivially. But this is a overkill solution. We can even implement imperative queue with singly linked-list or plain array. However, our main question here is about how to realize a purely functional queue as well?

We'll first review the typical queue solution which is realized by singly linked-list and circular buffer in first section; Then we give a simple and straightforward functional solution in the second section. While the performance is ensured in terms of amortized constant time, we need find real-time solution (or worst-case solution) for some special case. Such solution will be described in the third and the fourth section. Finally, we'll show a very simple real-time queue which depends on lazy evaluation.

Most of the functional contents are based on Chris, Okasaki's great work in [6]. There are more than 16 different types of purely functional queue given in that material.

11.2 Queue by linked-list and circular buffer

11.2.1 Singly linked-list solution

Queue can be implemented with singly linked-list. It's easy to add and remove element at the front end of a linked-list in $O(1)$ time. However, in order to keep the FIFO order, if we execute one operation on head, we must perform the inverse operation on tail.

For plain singly linked-list, we must traverse the whole list before adding or removing. Traversing is bound to $O(N)$ time, where N is the length of the list. This doesn't match the abstract queue properties.

The solution is to use an extra record to store the tail of the linked-list. A sentinel is often used to simplify the boundary handling. The following ANSI C [1] code defines a queue realized by singly linked-list.

```
typedef int Key;

struct Node{
  Key key;
  struct Node* next;
};

struct Queue{
  struct Node *head, *tail;
};
```

Figure 11.1 illustrates an empty list. Both head and tail point to the sentinel NIL node.

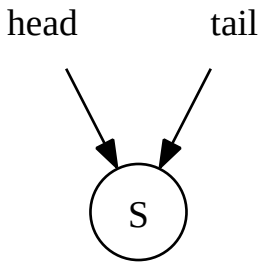


Figure 11.1: The empty queue, both head and tail point to sentinel node.

We summarize the abstract queue interface as the following.

function EMPTY ▷ Create an empty queue
function EMPTY?(Q) ▷ Test if Q is empty
function ENQUEUE(Q, x) ▷ Add a new element x to queue Q
function DEQUEUE(Q) ▷ Remove element from queue Q
function HEAD(Q) ▷ get the next element in queue Q in FIFO order

[1] It's possible to parameterize the type of the key with C++ template. ANSI C is used here for illustration purpose.

Note the difference between DEQUEUE and HEAD. HEAD only retrieve next element in FIFO order without removing it, while DEQUEUE performs removing.

In some programming languages, such as Haskell, and most object-oriented languages, the above abstract queue interface can be ensured by some definition. For example, the following Haskell code specifies the abstract queue.

```
class Queue q where
    empty :: q a
    isEmpty :: q a → Bool
    push :: q a → a → q a -- aka 'snoc' or append, or push_back
    pop :: q a → q a -- aka 'tail' or pop_front
    front :: q a → a -- aka 'head'
```

To ensure the constant time ENQUEUE and DEQUEUE, we add new element to head and remove element from tail.[2]

function ENQUEUE(Q, x)
  $p \leftarrow$ CREATE-NEW-NODE
  KEY(p) $\leftarrow x$
  NEXT(p) $\leftarrow NIL$
  NEXT(TAIL(Q)) $\leftarrow p$
  TAIL(Q) $\leftarrow p$

Note that, as we use the sentinel node, there are at least one node, the sentinel in the queue. That's why we needn't check the validation of of the tail before we append the new created node p to it.

function DEQUEUE(Q)
  $x \leftarrow$ HEAD(Q)
  NEXT(HEAD(Q)) $\leftarrow$ NEXT(x)
  if $x =$ TAIL(Q) **then** ▷ Q gets empty
    TAIL(Q) $\leftarrow$ HEAD(Q)
  return KEY(x)

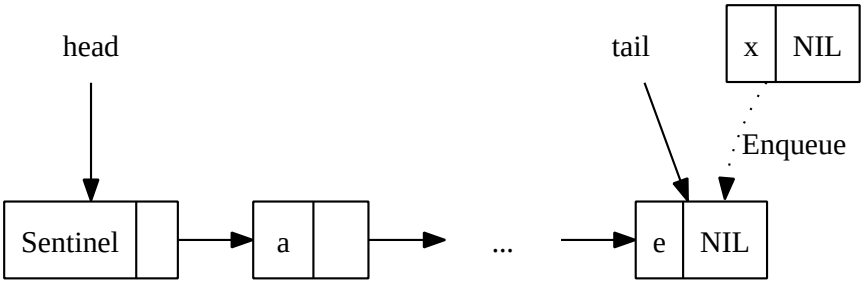
As we always put the sentinel node in front of all the other nodes, function HEAD actually returns the next node to the sentinel.

Figure 11.2 illustrates ENQUEUE and DEQUEUE process with sentinel node.

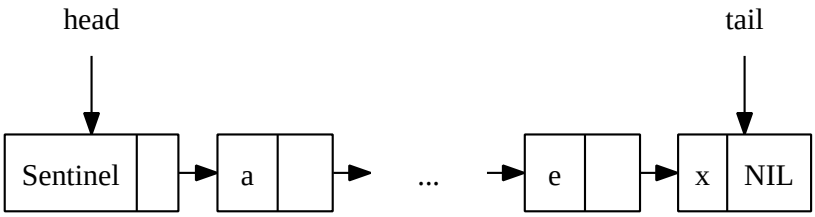
Translating the pseudo code to ANSI C program yields the below code.

```
struct Queue* enqueue(struct Queue* q, Key x){
  struct Node* p = (struct Node*)malloc(sizeof(struct Node));
  p→key = x;
  p→next = NULL;
  q→tail→next = p;
  q→tail = p;
  return q;
}

Key dequeue(struct Queue* q){
  struct Node* p = head(q); /*gets the node next to sentinel*/
  Key x = key(p);
  q→head→next = p→next;
  if(q→tail == p)
```

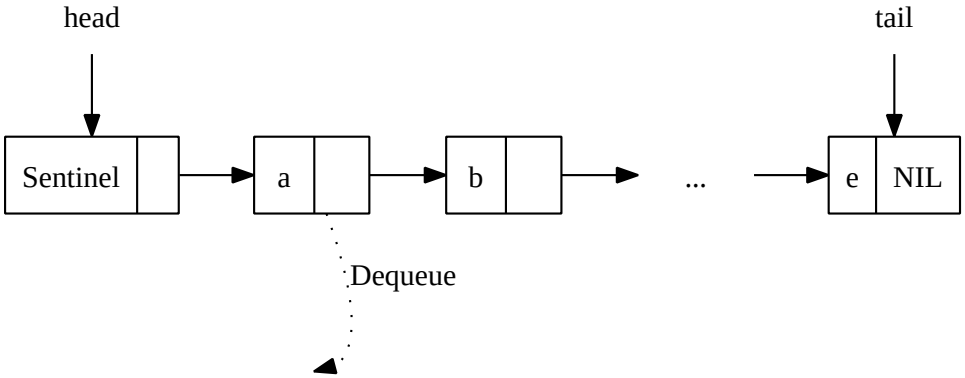
[2]It's possible to add new element to the tail, while remove element from head, but the operations are more complex than this approach.



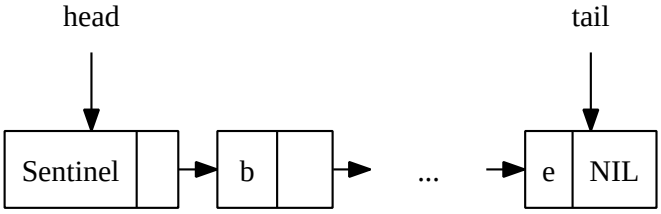
(a) Before ENQUEUE x to queue



(b) After ENQUEUE x to queue



(c) Before DEQUEUE x to queue



(d) After DEQUEUE x to queue

Figure 11.2: ENQUEUE and DEQUEUE to linked-list queue.

```
    q→tail = q→head;
  free(p);
  return x;
}
```

This solution is simple and robust. It's easy to extend this solution even to the concurrent environment (e.g. multicores). We can assign a lock to the head and use another lock to the tail. The sentinel helps us from being dead-locked due to the empty case [1] [2].

Exercise 11.1

- Realize the EMPTY? and HEAD algorithms for linked-list queue.
- Implement the singly linked-list queue in your favorite imperative programming language. Note that you need provide functions to initialize and destroy the queue.

11.2.2 Circular buffer solution

Another typical solution to realize queue is to use plain array as a circular buffer (also known as ring buffer). Oppose to linked-list, array support appending to the tail in constant $O(1)$ time if there are still spaces. Of course we need re-allocate spaces if the array is fully occupied. However, Array performs poor in $O(N)$ time when removing element from head and packing the space. This is because we need shift all rest elements one cell ahead. The idea of circular buffer is to reuse the free cells before the first valid element after we remove elements from head.

The idea of circular buffer can be described in figure 11.3 and 11.4.

If we set a maximum size of the buffer instead of dynamically allocate memories, the queue can be defined with the below ANSI C code.

```
struct Queue{
  Key* buf;
  int head, tail, size;
};
```

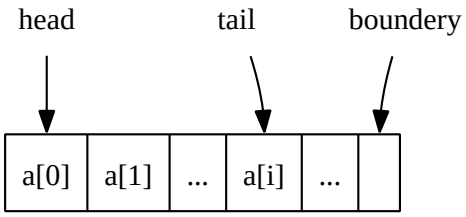
When initialize the queue, we are explicitly asked to provide the maximum size as argument.

```
struct Queue* createQ(int max){
  struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
  q→buf = (Key*)malloc(sizeof(Key)*max);
  q→size = max;
  q→head = q→tail = 0;
  return q;
}
```

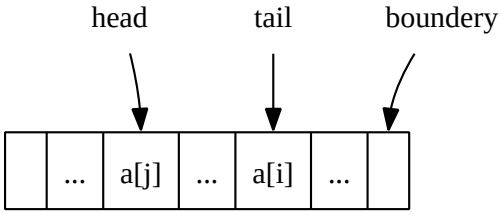
To test if a queue is empty is trivial.

function EMPTY?(Q)
  return HEAD(Q) = TAIL(Q)

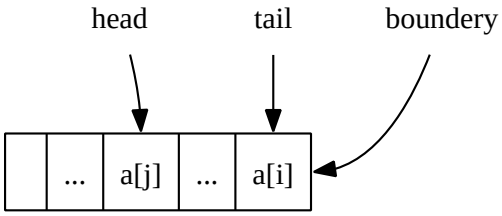
One brute-force implementation for ENQUEUE and DEQUEUE is to calculate the modular of index blindly as the following.



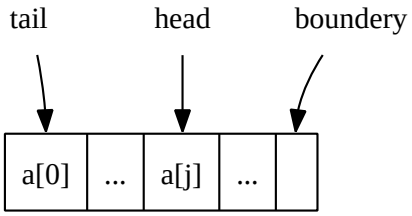
(a) Continuously add some elements.



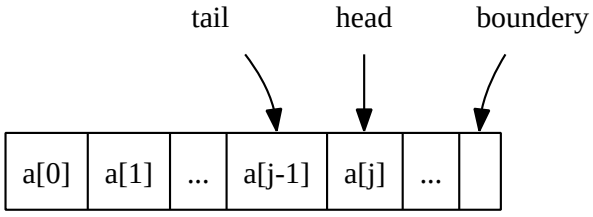
(b) After remove some elements from head, there are free cells.



(c) Go on adding elements till the boundary of the array.



(d) The next element is added to the first free cell on head.



(e) All cells are occupied. The queue is full.

Figure 11.3: A queue is realized with ring buffer.

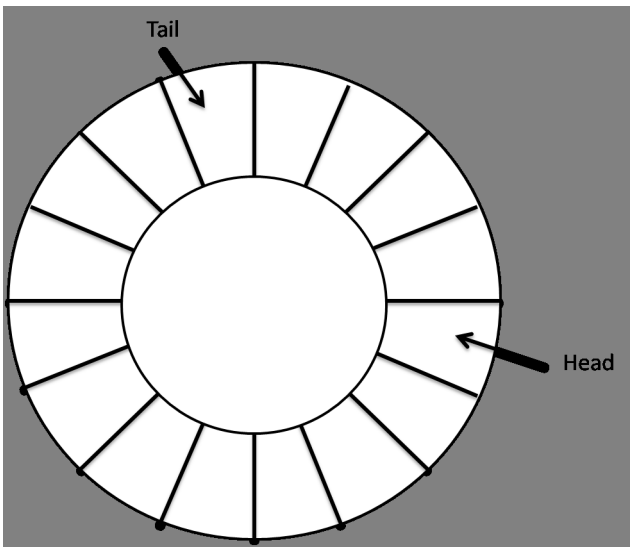


Figure 11.4: The circular buffer.

function ENQUEUE(Q, x)
  if $\neg$ FULL?(Q) **then**
    TAIL(Q) $\leftarrow$ (TAIL(Q) + 1) mod SIZE(Q)
    BUFFER(Q)[TAIL(Q)] $\leftarrow x$

function HEAD(Q)
  if $\neg$ EMPTY?(Q) **then**
    return BUFFER(Q)[HEAD(Q)]

function DEQUEUE(Q)
  if $\neg$ EMPTY?(Q) **then**
    HEAD(Q) $\leftarrow$ (HEAD(Q) + 1) mod SIZE(Q)

However, modular is expensive and slow depends on some settings, so one may replace it by some adjustment. For example as in the below ANSI C program.

```
void enQ(struct Queue* q, Key x){
  if(!fullQ(q)){
    q→buf[q→tail++] = x;
    q→tail -= q→tail< q→size ? 0 : q→size;
  }
}

Key headQ(struct Queue* q){
  return q→buf[q→head]; /* Assume queue isn't␣empty␣*/
}

Key␣deQ(struct␣Queue*␣q){
␣␣Key␣x␣=␣headQ(q);
␣␣q→head++;
␣␣q→head␣-=␣q→head<␣q→size␣?␣0␣:␣q→size;
␣␣return␣x;
```

```
}
```

Exercise 11.2

As the circular buffer is allocated with a maximum size parameter, please write a function to test if a queue is full to avoid overflow. Note there are two cases, one is that the head is in front of the tail, the other is on the contrary.

11.3 Purely functional solution

11.3.1 Paired-list queue

We can't just use a list to implement queue, or we can't satisfy abstract queue properties. This is because singly linked-list, which is the back-end data structure in most functional settings, performs well on head in constant $O(1)$ time, while it performs in linear $O(N)$ time on tail, where N is the length of the list. Either dequeue or enqueue will perform proportion to the number of elements stored in the list as shown in figure 11.5.

(a) DeQueue performs poorly.

(b) EnQueue performs poorly.

Figure 11.5: DeQueue and EnQueue can't perform both in constant $O(1)$ time with a list.

We neither can add a pointer to record the tail position of the list as what we have done in the imperative settings like in the ANSI C program, because of the nature of purely functional.

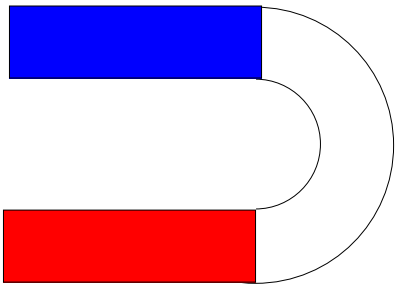
Chris Okasaki mentioned a simple and straightforward functional solution in [6]. The idea is to maintain two linked-lists as a queue, and concatenate these two lists in a tail-to-tail manner. The shape of the queue looks like a horseshoe magnet as shown in figure 11.6.

With this setup, we push new element to the head of the rear list, which is ensure to be $O(1)$ constant time; on the other hand, we pop element from the head of the front list, which is also $O(1)$ constant time. So that the abstract queue properties can be satisfied.

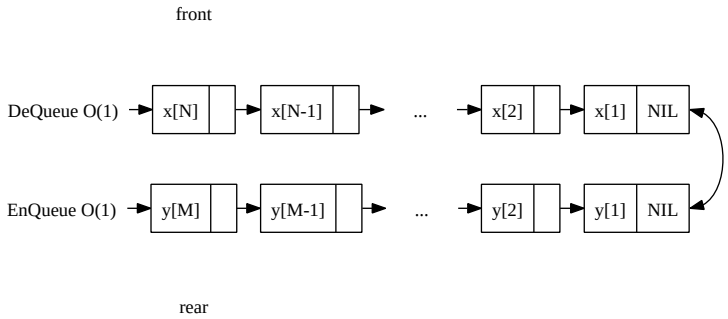
The definition of such paired-list queue can be expressed in the following Haskell code.

```
type Queue a = ([a], [a])

empty = ([], [])
```

Suppose function $front(Q)$ and $rear(Q)$ return the front and rear list in such setup, and $Queue(F, R)$ create a paired-list queue from two lists F and R.



(a) a horseshoe magnet.



(b) concatenate two lists tail-to-tail.

Figure 11.6: A queue with front and rear list shapes like a horseshoe magnet.

The EnQueue (push) and DeQueue (pop) operations can be easily realized based on this setup.

$$push(Q, x) = Queue(front(Q), \{x\} \cup rear(Q)) \tag{11.1}$$

$$pop(Q) = Queue(tail(front(Q)), rear(Q)) \tag{11.2}$$

where if a list $X = \{x_1, x_2, ..., x_n\}$, function $tail(X) = \{x_2, x_3, ..., x_n\}$ returns the rest of the list without the first element.

However, we must next solve the problem that after several pop operations, the front list becomes empty, while there are still elements in rear list. One method is to rebuild the queue by reversing the rear list, and use it to replace front list.

Hence a balance operation will be execute after popping. Let's denote the front and rear list of a queue Q as $F = front(Q)$, and $R = fear(Q)$.

$$balance(F, R) = \begin{cases} Queue(reverse(R), \Phi) & : \quad F = \Phi \\ Q & : \quad otherwise \end{cases} \tag{11.3}$$

Thus if front list isn't empty, we do nothing, while when the front list becomes empty, we use the reversed rear list as the new front list, and the new rear list is empty.

The new enqueue and dequeue algorithms are updated as below.

$$push(Q, x) = balance(F, \{x\} \cup R) \tag{11.4}$$

$$pop(Q) = balance(tail(F), R) \tag{11.5}$$

Sum up the above algorithms and translate them to Haskell yields the following program.

```
balance :: Queue a → Queue a
balance ([], r) = (reverse r, [])
balance q = q

push :: Queue a → a → Queue a
push (f, r) x = balance (f, x:r)

pop :: Queue a → Queue a
pop ([], _) = error "Empty"
pop (_:f, r) =  balance (f, r)
```

However, although we only touch the heads of front list and rear list, the overall performance can't be kept always as $O(1)$. Actually, the performance of this algorithm is amortized $O(1)$. This is because the reverse operation takes time proportion to the length of the rear list. it's bound $O(N)$ time, where $N = |R|$. We left the prove of amortized performance as an exercise to the reader.

11.3.2 Paired-array queue - a symmetric implementation

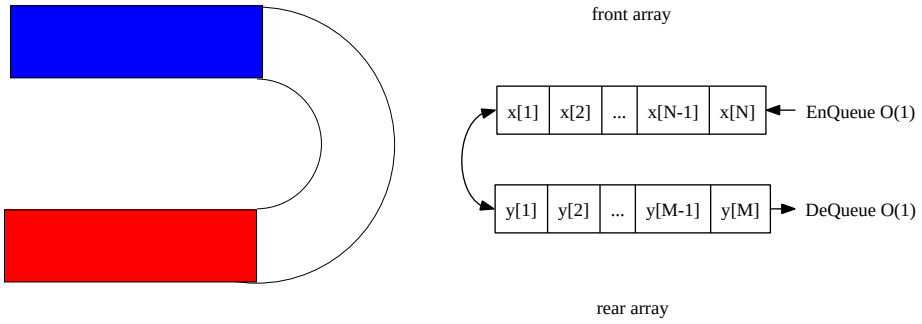
There is an interesting implementation which is symmetric to the paired-list queue. In some old programming languages, such as legacy version of BASIC, There is array supported, but there is no pointers, nor records to represent linked-list. Although we can use another array to store indexes so that we can represent linked-list with implicit array, there is another option to realized amortized $O(1)$ queue.

Compare the performance of array and linked-list. Below table reveals some facts (Suppose both contain N elements).

operation	Array	Linked-list
insert on head	$O(N)$	$O(1)$
insert on tail	$O(1)$	$O(N)$
remove on head	$O(N)$	$O(1)$
remove on tail	$O(1)$	$O(N)$

Note that linked-list performs in constant time on head, but in linear time on tail; while array performs in constant time on tail (suppose there is enough memory spaces, and omit the memory reallocation for simplification), but in linear time on head. This is because we need do shifting when prepare or eliminate an empty cell in array. (see chapter 'the evolution of insertion sort' for detail.)

The above table shows an interesting characteristic, that we can exploit it and provide a solution mimic to the paired-list queue: We concatenate two arrays, head-to-head, to make a horseshoe shape queue like in figure 11.7.



(a) a horseshoe magnet.

(b) concatenate two arrays head-to-head.

Figure 11.7: A queue with front and rear arrays shapes like a horseshoe magnet.

We can define such paired-array queue like the following Python code [3]

```
class Queue:
    def __init__(self):
        self.front = []
        self.rear = []
```

[3]Legacy Basic code is not presented here. And we actually use list but not array in Python to illustrate the idea. ANSI C and ISO C++ programs are provides along with this chapter, they show more in a purely array manner.

```
def is_empty(q):
    return q.front == [] and q.rear == []
```

The relative PUSH() and POP() algorithm only manipulate on the tail of the arrays.

function PUSH(Q, x)
  APPEND(REAR(Q), x)

Here we assume that the APPEND() algorithm append element x to the end of the array, and handle the necessary memory allocation etc. Actually, there are multiple memory handling approaches. For example, besides the dynamic re-allocation, we can initialize the array with enough space, and just report error if it's full.

function POP(Q)
  if FRONT(Q) $= \Phi$ **then**
    FRONT(Q) $\leftarrow$ REVERSE(REAR(Q))
    REAR(Q) $\leftarrow \Phi$
  $N \leftarrow$ LENGTH(FRONT(Q))
  $x \leftarrow$ FRONT(Q)[N]
  LENGTH(FRONT(Q)) $\leftarrow N - 1$
  return x

For simplification and pure illustration purpose, the array isn't shrunk explicitly after elements removed. So test if front array is empty (Φ) can be realized as check if the length of the array is zero. We omit all these details here.

The enqueue and dequeue algorithms can be translated to Python programs straightforwardly.

```
def push(q, x):
    q.rear.append(x)

def pop(q):
    if q.front == []:
        q.rear.reverse()
        (q.front, q.rear) = (q.rear, [])
    return q.front.pop()
```

Similar to the paired-list queue, the performance is amortized $O(1)$ because the reverse procedure takes linear time.

Exercise 11.3

- Prove that the amortized performance of paired-list queue is $O(1)$.
- Prove that the amortized performance of paired-array queue is $O(1)$.

11.4 A small improvement, Balanced Queue

Although paired-list queue is amortized $O(1)$ for popping and pushing, the solution we proposed in previous section performs poor in the worst case. For

example, there is one element in the front list, and we push N elements continuously to the queue, here N is a big number. After that executing a pop operation will cause the worst case.

According to the strategy we used so far, all the N elements are added to the rear list. The front list turns to be empty after a pop operation. So the algorithm starts to reverse the rear list. This reversing procedure is bound to $O(N)$ time, which is proportion to the length of the rear list. Sometimes, it can't be acceptable for a very big N.

The reason why this worst case happens is because the front and rear lists are extremely unbalanced. We can improve our paired-list queue design by making them more balanced. One option is to add a balancing constraint.

$$|R| \leq |F| \tag{11.6}$$

Where $R = Rear(Q)$, $F = Front(Q)$, and $|L|$ is the length of list L. This constraint ensure the length of the rear list is less than the length of the front list. So that the reverse procedure will be executed once the rear list grows longer than the front list.

Here we need frequently access the length information of a list. However, calculate the length takes linear time for singly linked-list. We can record the length to a variable and update it as adding and removing elements. This approach enables us to get the length information in constant time.

Below example shows the modified paired-list queue definition which is augmented with length fields.

```
data BalanceQueue a = BQ [a] Int [a] Int
```

As we keep the invariant as specified in (11.6), we can easily tell if a queue is empty by testing the length of the front list.

$$F = \Phi \Leftrightarrow |F| = 0 \tag{11.7}$$

In the rest part of this section, we suppose the length of a list L, can be retrieved as $|L|$ in constant time.

Push and pop are almost as same as before except that we check the balance invariant by passing length information and performs reversing accordingly.

$$push(Q, x) = balance(F, |F|, \{x\} \cup R, |R| + 1) \tag{11.8}$$

$$pop(Q) = balance(tail(F), |F| - 1, R, |R|) \tag{11.9}$$

Where function $balance()$ is defined as the following.

$$balance(F, |F|, R, |R|) = \begin{cases} Queue(F, |F|, R, |R|) & : \quad |R| \leq |F| \\ Queue(F \cup reverse(R), |F| + |R|, \Phi, 0) & : \quad otherwise \end{cases} \tag{11.10}$$

Note that the function $Queue()$ takes four parameters, the front list along with its length (recorded), and the rear list along with its length, and forms a paired-list queue augmented with length fields.

We can easily translate the equations to Haskell program. And we can enforce the abstract queue interface by making the implementation an instance of the Queue type class.

```
instance Queue BalanceQueue where
    empty = BQ [] 0 [] 0

    isEmpty (BQ _ lenf _ _) = lenf == 0

    -- Amortized O(1) time push
    push (BQ f lenf r lenr) x = balance f lenf (x:r) (lenr + 1)

    -- Amortized O(1) time pop
    pop (BQ (_:f) lenf r lenr) = balance f (lenf - 1) r lenr

    front (BQ (x:_) _ _ _) = x

balance f lenf r lenr
    | lenr ≤ lenf = BQ f lenf r lenr
    | otherwise = BQ (f ++ (reverse r)) (lenf + lenr) [] 0
```

Exercise 11.4

Write the symmetric balance improvement solution for paired-array queue in your favorite imperative programming language.

11.5 One more step improvement, Real-time Queue

Although the extremely worst case can be avoided by improving the balancing as what has been presented in previous section, the performance of reversing rear list is still bound to $O(N)$, where $N = |R|$. So if the rear list is very long, the instant performance is still unacceptable poor even if the amortized time is $O(1)$. It is particularly important in some real-time system to ensure the worst case performance.

As we have analyzed, the bottleneck is the computation of $F \cup reverse(R)$. This happens when $|R| > |F|$. Considering that $|F|$ and $|R|$ are all integers, so this computation happens when

$$|R| = |F| + 1 \tag{11.11}$$

Both F and the result of $reverse(R)$ are singly linked-list, It takes $O(|F|)$ time to concatenate them together, and it takes extra $O(|R|)$ time to reverse the rear list, so the total computation is bound to $O(|N|)$, where $N = |F| + |R|$. Which is proportion to the total number of elements in the queue.

In order to realize a real-time queue, we can't computing $F \cup reverse(R)$ monolithic. Our strategy is to distribute this expensive computation to every pop and push operations. Thus although each pop and push get a bit slow, we may avoid the extremely slow worst pop or push case.

Incremental reverse

Let's examine how functional reverse algorithm is implemented typically.

$$reverse(X) = \begin{cases} \Phi & : & X = \Phi \\ reverse(X') \cup \{x_1\} & : & otherwise \end{cases} \tag{11.12}$$

Where $X' = tail(X) = \{x_2, x_3, ...\}$.

This is a typical recursive algorithm, that if the list to be reversed is empty, the result is just an empty list. This is the edge case; otherwise, we take the first element x_1 from the list, reverse the rest $\{x_2, x_3, ..., x_n\}$, to $\{x_n, x_{n-1}, .., x_3, x_2\}$ and append x_1 after it.

However, this algorithm performs poor, as appending an element to the end of a list is proportion to the length of the list. So it's $O(N^2)$, but not a linear time reverse algorithm.

There exists another implementation which utilizes an accumulator A, like below.

$$reverse(X) = reverse'(X, \Phi) \tag{11.13}$$

Where

$$reverse'(X, A) = \begin{cases} A & : \quad X = \Phi \\ reverse'(X', \{x_1\} \cup A) & : \quad otherwise \end{cases} \tag{11.14}$$

We call A as *accumulator* because it accumulates intermediate reverse result at any time. Every time we call $reverse'(X, A)$, list X contains the rest of elements wait to be reversed, and A holds all the reversed elements so far. For instance when we call $reverse'()$ at i-th time, X and A contains the following elements:

$$X = \{x_i, x_{i+1}, ..., x_n\} \quad A = \{x_{i-1}, x_{i-2}, ...x_1\}$$

In every non-trivial case, we takes the first element from X in $O(1)$ time; then put it in front of the accumulator A, which is again $O(1)$ constant time. We repeat it N times, so this is a linear time ($O(N)$) algorithm.

The latter version of reverse is obviously a *tail-recursion* algorithm, see [5] and [6] for detail. Such characteristic is easy to change from monolithic algorithm to incremental manner.

The solution is state transferring. We can use a state machine contains two types of stat: reversing state S_r to indicate that the reverse is still on-going (not finished), and finish state S_f to indicate the reverse has been done (finished). In Haskell programming language, it can be defined as a type.

```
data State a = | Reverse [a] [a]
               | Done [a]
```

And we can schedule (slow-down) the above $reverse'(X, A)$ function with these two types of state.

$$step(S, X, A) = \begin{cases} (S_f, A) & : \quad S = S_r \wedge X = \Phi \\ (S_r, X', \{x_1\} \cup A) & : \quad S = S_r \wedge X \neq \Phi \end{cases} \tag{11.15}$$

Each step, we examine the state type first, if the current state is S_r (on-going), and the rest elements to be reversed in X is empty, we can turn the algorithm to finish state S_f; otherwise, we take the first element from X, put it in front of A just as same as above, but we do NOT perform recursion, instead, we just finish this step. We can store the current state as well as the resulted

X and A, the reverse can be continued at any time when we call 'next' *step* function in the future with the stored state, X and A passed in.

Here is an example of this step-by-step reverse algorithm.

$$\begin{aligned} step(S_r, "hello", \Phi) &= (S_r, "ello", "h") \\ step(S_r, "ello", "h") &= (S_r, "llo", "eh") \\ ... \\ step(S_r, "o", "lleh") &= (S_r, \Phi, "olleh") \\ step(S_r, \Phi, "olleh") &= (S_f, "olleh") \end{aligned}$$

And in Haskell code manner, the example is like the following.

```
step $ Reverse "hello" [] = Reverse "ello" "h"
step $ Reverse "ello" "h" = Reverse "llo" "eh"
...
step $ Reverse "o" "lleh" = Reverse [] "olleh"
step $ Reverse [] "olleh" = Done "olleh"
```

Now we can distribute the reverse into steps in every pop and push operations. However, the problem is just half solved. We want to break down $F \cup reverse(R)$, and we have broken $reverse(R)$ into steps, we next need to schedule(slow-down) the list concatenation part $F \cup ...$, which is bound to $O(|F|)$, into incremental manner so that we can distribute it to pop and push operations.

Incremental concatenate

It's a bit more challenge to implement incremental list concatenation than list reversing. However, it's possible to re-use the result we gained from increment reverse by a small trick: In order to realize $X \cup Y$, we can first reverse X to $\overleftarrow{X}$, then take elements one by one from $\overleftarrow{X}$ and put them in front of Y just as what we have done in $reverse'$.

$$\begin{aligned} X \cup Y &\equiv reverse(reverse(X)) \cup Y \\ &\equiv reverse'(reverse(X), \Phi) \cup Y \\ &\equiv reverse'(reverse(X), Y) \\ &\equiv reverse'(\overleftarrow{X}, Y) \end{aligned} \tag{11.16}$$

This fact indicates us that we can use an extra state to instruct the $step()$ function to continuously concatenating $\overleftarrow{F}$ after R is reversed.

The strategy is to do the total work in two phases:

1. Reverse both F and R in parallel to get $\overleftarrow{F} = reverse(F)$, and $\overleftarrow{R} = reverse(R)$ incrementally;
2. Incrementally take elements from $\overleftarrow{F}$ and put them in front of $\overleftarrow{R}$.

So we define three types of state: S_r represents reversing; S_c represents concatenating; and S_f represents finish.

In Haskell, these types of state are defined as the following.

```
data State a = Reverse [a] [a] [a] [a]
             | Concat [a] [a]
             | Done [a]
```

Because we reverse F and R simultaneously, so reversing state takes two pairs of lists and accumulators.

The state transfering is defined according to the two phases strategy described previously. Denotes that $F = \{f_1, f_2, ...\}$, $F' = tail(F) = \{f_2, f_3, ...\}$, $R = \{r_1, r_2, ...\}$, $R' = tail(R) = \{r_2, r_3, ...\}$. A state $\mathcal{S}$, contains it's type S, which has the value among S_r, S_c, and S_f. Note that $\mathcal{S}$ also contains necessary parameters such as F, $\overleftarrow{F}$, X, A etc as intermediate results. These parameters vary according to the different states.

$$next(\mathcal{S}) = \begin{cases} (S_r, F', \{f_1\} \cup \overleftarrow{F}, R', \{r_1\} \cup \overleftarrow{R}) & : \quad S = S_r \wedge F \neq \Phi \wedge R \neq \Phi \\ (S_c, \overleftarrow{F}, \{r_1\} \cup \overleftarrow{R}) & : \quad S = S_r \wedge F = \Phi \wedge R = \{r_1\} \\ (S_f, A) & : \quad S = S_c \wedge X = \Phi \\ (S_c, X', \{x_1\} \cup A) & : \quad S = S_c \wedge X \neq \Phi \end{cases} \tag{11.17}$$

The relative Haskell program is list as below.

```
next (Reverse (x:f) f' (y:r) r') = Reverse f (x:f') r (y:r')
next (Reverse [] f' [y] r') = Concat f' (y:r')
next (Concat 0 _ acc) = Done acc
next (Concat (x:f') acc) = Concat f' (x:acc)
```

All left to us is to distribute these incremental steps into every pop and push operations to implement a real-time $O(1)$ purely functional queue.

Sum up

Before we dive into the final real-time queue implementation. Let's analyze how many incremental steps are taken to achieve the result of $F \cup reverse(R)$. According to the balance variant we used previously, $|R| = |F|+1$, Let's denotes $M = |F|$.

Once the queue gets unbalanced due to some push or pop operation, we start this incremental $F \cup reverse(R)$. It needs $M + 1$ steps to reverse R, and at the same time, we finish reversing the list F within these steps. After that, we need extra $M + 1$ steps to execute the concatenation. So there are $2M + 2$ steps.

It seems that distribute one step inside one pop or push operation is the natural solution, However, there is a critical question must be answered: Is it possible that before we finish these $2M + 2$ steps, the queue gets unbalanced again due to a series push and pop?

There are two facts about this question, one is good news and the other is bad news.

Let's first show the good news, that luckily, continuously pushing can't make the queue unbalanced again before we finish these $2M + 2$ steps to achieve $F \cup reverse(R)$. This is because once we start re-balancing, we can get a new front list $F' = F \cup reverse(R)$ after $2M+2$ steps. While the next time unbalance is triggered when

$$\begin{aligned} |R'| &= |F'| + 1 \\ &= |F| + |R| + 1 \\ &= 2M + 2 \end{aligned} \tag{11.18}$$

That is to say, even we continuously pushing as mush elements as possible after the last unbalanced time, when the queue gets unbalanced again, the

front copy	on-going computation	new rear
$\{f_i, f_{i+1}, ..., f_M\}$	$(S_r, \tilde{F}, ..., \tilde{R}, ...)$	$\{...\}$
first $i-1$ elements popped	intermediate result $\overleftarrow{F}$ and $\overleftarrow{R}$	new elements pushed

Table 11.1: Intermediate state of a queue before first M steps finish.

$2M + 2$ steps exactly get finished at that time point. Which means the new front list F' is calculated OK. We can safely go on to compute $F' \cup reverse(R')$. Thanks to the balance invariant which is designed in previous section.

But, the bad news is that, pop operation can happen at anytime before these $2M + 2$ steps finish. The situation is that once we want to extract element from front list, the new front list $F' = F \cup reverse(R)$ hasn't been ready yet. We don't have a valid front list at hand.

One solution to solve this problem is to keep a copy of original front list F, during the time we are calculating $reverse(F)$ which is described in phase 1 of our incremental computing strategy. So that we are still safe even if user continuously performs first M pop operations. So the queue looks like in table 11.1 at some time after we start the incremental computation and before phase 1 (reverse F and R simultaneously) ending[4].

After these M pop operations, the copy of F is exhausted. And we just start incremental concatenation phase at that time. What if user goes on popping?

The fact is that since F is exhausted (becomes Φ), we needn't do concatenation at all. Since $F \cup \overleftarrow{R} = \Phi \cup \overleftarrow{R} = \overleftarrow{R}$.

It indicates us, when doing concatenation, we only need to concatenate those elements haven't been popped, which are still left in F. As user pops elements one by one continuously from the head of front list F, one method is to use a counter, record how many elements there are still in F. The counter is initialized as 0 when we start computing $F \cup reverse(R)$, it's increased by one when we reverse one element in F, which means we need concatenate thie element in the future; and it's decreased by one every time when pop is performed, which means we can concatenate one element less; of course we need decrease this counter as well in every steps of concatenation. If and only if this counter becomes zero, we needn't do concatenations any more.

We can give the realization of purely functional real-time queue according to the above analysis.

We first add an idle state S_0 to simplify some state transfering. Below Haskell program is an example of this modified state definition.

```
data State a = Empty
             | Reverse Int [a] [a] [a] [a] -- n, f', acc_f' r, acc_r
             | Append Int [a] [a]          -- n, rev_f', acc
             | Done [a] -- result: f ++ reverse r
```

[4]One may wonder that copying a list takes linear time to the length of the list. If so the whole solution would make no sense. Actually, this linear time copying won't happen at all. This is because the purely functional nature, the front list won't be mutated either by popping or by reversing. However, if trying to realize a symmetric solution with paired-array and mutate the array in-place, this issue should be stated, and we can perform a 'lazy' copying, that the real copying work won't execute immediately, instead, it copies one element every step we do incremental reversing. The detailed implementation is left as an exercise.

And the data structure is defined with three parts, the front list (augmented with length); the on-going state of computing $F \cup reverse(R)$; and the rear list (augmented with length).

Here is the Haskell definition of real-time queue.

```
data RealtimeQueue a = RTQ [a] Int (State a) [a] Int
```

The empty queue is composed with empty front and rear list together with idle state S_0 as $Queue(\Phi, 0, S_0, \Phi, 0)$. And we can test if a queue is empty by checking if $|F| = 0$ according to the balance invariant defined before. Push and pop are changed accordingly.

$$push(Q, x) = balance(F, |F|, \mathcal{S}, \{x\} \cup R, |R| + 1) \tag{11.19}$$

$$pop(Q) = balance(F', |F| - 1, abort(\mathcal{S}), R, |R|) \tag{11.20}$$

The major difference is $abort()$ function. Based on our above analysis, when there is popping, we need decrease the counter, so that we can concatenate one element less. We define this as aborting. The details will be given after $balance()$ function.

The relative Haskell code for push and pop are listed like this.

```
push (RTQ f lenf s r lenr) x = balance f lenf s (x:r) (lenr + 1)
pop (RTQ (_:f) lenf s r lenr) = balance f (lenf - 1) (abort s) r lenr
```

The $balance()$ function first check the balance invariant, if it's violated, we need start re-balance it by starting compute $F \cup reverse(R)$ incrementally; otherwise we just execute one step of the unfinished incremental computation.

$$balance(F, |F|, \mathcal{S}, R, |R|) = \begin{cases} step(F, |F|, \mathcal{S}, R, |R|) & : \quad |R| \leq |F| \\ step(F, |F| + |R|, (S_r, 0, F, \Phi, R, \Phi)\Phi, 0) & : \quad otherwise \end{cases} \tag{11.21}$$

The relative Haskell code is given like below.

```
balance f lenf s r lenr
    | lenr ≤ lenf =  step f lenf s r lenr
    | otherwise = step f (lenf + lenr) (Reverse 0 f [] r []) [] 0
```

The $step()$ function typically transfer the state machine one state ahead, and it will turn the state to idle (S_0) when the incremental computation finishes.

$$step(F, |F|, \mathcal{S}, R, |R|) = \begin{cases} Queue(F', |F|, S_0, R, |R|) & : \quad S' = S_f \\ Queue(F, |F|, \mathcal{S}', R, |R|) & : \quad otherwise \end{cases} \tag{11.22}$$

Where $\mathcal{S}' = next(\mathcal{S})$ is the next state transferred; $F' = F \cup reverse(R)$, is the final new front list result from the incremental computing. The real state transferring is implemented in $next()$ function as the following. It's different from previous version by adding the counter field n to record how many elements

left we need to concatenate.

$$next(\mathcal{S}) = \begin{cases} (S_r, n+1, F', \{f_1\} \cup \overleftarrow{F}, R', \{r_1\} \cup \overleftarrow{R}) & : & S = S_r \wedge F \neq \Phi \\ (S_c, n, \overleftarrow{F}, \{r_1\} \cup \overleftarrow{R}) & : & S = S_r \wedge F = \Phi \\ (S_f, A) & : & S = S_c \wedge n = 0 \\ (S_c, n-1, X', \{x_1\} \cup A) & : & S = S_c \wedge n \neq 0 \\ \mathcal{S} & : & otherwise \end{cases} \tag{11.23}$$

And the corresponding Haskell code is like this.

```
next (Reverse n (x:f) f' (y:r) r') = Reverse (n+1) f (x:f') r (y:r')
next (Reverse n [] f' [y] r') = Concat n f' (y:r')
next (Concat 0 _ acc) = Done acc
next (Concat n (x:f') acc) = Concat (n-1) f' (x:acc)
next s = s
```

Function $abort()$ is used to tell the state machine, we can concatenate one element less since it is popped.

$$abort(\mathcal{S}) = \begin{cases} (S_f, A') & : & S = S_c \wedge n = 0 \\ (S_c, n-1, X'A) & : & S = S_c \wedge n \neq 0 \\ (S_r, n-1, F, \overleftarrow{F}, R, \overleftarrow{R}) & : & S = S_r \\ \mathcal{S} & : & otherwise \end{cases} \tag{11.24}$$

Note that when $n = 0$ we actually rollback one concatenated element by return A' as the result but not A. (Why? this is left as an exercise.)

The Haskell code for abort function is like the following.

```
abort (Concat 0 _ (_:acc)) = Done acc -- Note! we rollback 1 elem
abort (Concat n f' acc) = Concat (n-1) f' acc
abort (Reverse n f f' r r') = Reverse (n-1) f f' r r'
abort s = s
```

It seems that we've done, however, there is still one tricky issue hidden behind us. If we push an element x to an empty queue, the result queue will be:

$$Queue(\Phi, 1, (S_c, 0, \Phi, \{x\}), \Phi, 0)$$

If we perform pop immediately, we'll get an error! We found that the front list is empty although the previous computation of $F \cup reverse(R)$ has been finished. This is because it takes one more extra step to transfer from the state $(S_c, 0, \Phi, A)$ to (S_f, A). It's necessary to refine the $\mathcal{S}'$ in $step()$ function a bit.

$$\mathcal{S}' = \begin{cases} next(next(\mathcal{S})) & : & F = \Phi \\ next(\mathcal{S}) & : & otherwise \end{cases} \tag{11.25}$$

The modification reflects to the below Haskell code:

```
step f lenf s r lenr =
    case s' of
      Done f' → RTQ f' lenf Empty r lenr
      s' → RTQ f lenf s' r lenr
    where s' = if null f then next $ next s else next s
```

Note that this algorithm differs from the one given by Chris Okasaki in [6]. Okasaki's algorithm executes two steps per pop and push, while the one presents in this chapter executes only one per pop and push, which leads to more distributed performance.

Exercise 11.5

- Why need we rollback one element when $n = 0$ in $abort()$ function?
- Realize the real-time queue with symmetric paired-array queue solution in your favorite imperative programming language.
- In the footnote, we mentioned that when we start incremental reversing with in-place paired-array solution, copying the array can't be done monolithic or it will lead to linear time operation. Implement the lazy copying so that we copy one element per step along with the reversing.

11.6 Lazy real-time queue

The key to realize a real-time queue is to break down the expensive $F \cup reverse(R)$ to avoid monolithic computation. Lazy evaluation is particularly helpful in such case. In this section, we'll explore if there is some more elegant solution by exploit laziness.

Suppose that there exits a function $rotate()$, which can compute $F \cup reverse(R)$ incrementally. that's to say, with some accumulator A, the following two functions are equivalent.

$$rotate(X, Y, A) \equiv X \cup reverse(Y) \cup A \tag{11.26}$$

Where we initialized X as the front list F, Y as the rear list R, and the accumulator A is initialized as empty Φ.

The trigger of rotation is still as same as before when $|F| + 1 = |R|$. Let's keep this constraint as an invariant during the whole rotation process, that $|X| + 1 = |Y|$ always holds.

It's obvious to deduce to the trivial case:

$$rotate(\Phi, \{y_1\}, A) = \{y_1\} \cup A \tag{11.27}$$

Denote $X = \{x_1, x_2, ...\}$, $Y = \{y_1, y_2, ...\}$, and $X' = \{x_2, x_3, ...\}$, $Y' = \{y_2, y_3, ...\}$ are the rest of the lists without the first element for X and Y respectively. The recursion case is ruled out as the following.

$$\begin{array}{lcll} rotate(X, Y, A) & \equiv & X \cup reverse(Y) \cup A & \text{Definition of (11.26)} \\ & \equiv & \{x_1\} \cup (X' \cup reverse(Y) \cup A) & \text{Associative of } \cup \\ & \equiv & \{x_1\} \cup (X' \cup reverse(Y') \cup (\{y_1\} \cup A)) & \text{Nature of reverse and associative of } \cup \\ & \equiv & \{x_1\} \cup rotate(X', Y', \{y_1\} \cup A) & \text{Definition of (11.26)} \end{array} \tag{11.28}$$

Summarize the above two cases, yields the final incremental rotate algorithm.

$$rotate(X, Y, A) = \begin{cases} \{y_1\} \cup A & : & X = \Phi \\ \{x_1\} \cup rotate(X', Y', \{y_1\} \cup A) & : & otherwise \end{cases} \tag{11.29}$$

If we execute $\cup$ lazily instead of strictly, that is, execute $\cup$ once pop or push operation is performed, the computation of *rotate* can be distribute to push and pop naturally.

Based on this idea, we modify the paired-list queue definition to change the front list to a lazy list, and augment it with a computation stream. [5]. When the queue triggers re-balance constraint by some pop/push, that $|F| + 1 = |R|$, The algorithm creates a lazy rotation computation, then use this lazy rotation as the new front list F'; the new rear list becomes Φ, and a copy of F' is maintained as a stream.

After that, when we performs every push and pop; we consume the stream by forcing a $\cup$ operation. This results us advancing one step along the stream, $\{x\} \cup F''$, where $F' = tail(F')$. We can discard x, and replace the stream F' with F''.

Once all of the stream is exhausted, we can start another rotation.

In order to illustrate this idea clearly, we turns to Scheme/Lisp programming language to show example codes, because it gives us explicit control of laziness.

In Scheme/Lisp, we have the following three tools to deal with lazy stream.

```
(define (cons-stream a b) (cons a (delay b)))

(define stream-car car)

(define (stream-cdr s) (cdr (force s)))
```

So 'cons-stream' constructs a 'lazy' list from an element x and an existing list L without really evaluating the value of L; The evaluation is actually delayed to 'stream-cdr', where the computation is forced. delaying can be realized by lambda calculus, please refer to [5] for detail.

The lazy paired-list queue is defined as the following.

```
(define (make-queue f r s)
  (list f r s))

;; Auxiliary functions
(define (front-lst q) (car q))

(define (rear-lst q) (cadr q))

(define (rots q) (caddr q))
```

A queue is consist of three parts, a front list, a rear list, and a stream which represents the computation of $F \cup reverse(R)$. Create an empty queue is trivial as making all these three parts null.

```
(define empty (make-queue '() '() '()))
```

Note that the front-list is also lazy stream actually, so we need use stream related functions to manipulate it. For example, the following function test if the queue is empty by checking the front lazy list stream.

```
(define (empty? q) (stream-null? (front-lst q)))
```

The push function is almost as same as the one given in previous section. That we put the new element in front of the rear list; and then examine the balance invariant and do necessary balancing works.

$$push(Q, x) = balance(\mathcal{F}, \{x\} \cup R, \mathcal{R}_s) \tag{11.30}$$

Where $\mathcal{R}$ represents the lazy stream of front list; $\mathcal{R}_s$ is the stream of rotation computation. The relative Scheme/Lisp code is give below.

```
(define (push q x)
  (balance (front-lst q) (cons x (rear q)) (rots q)))
```

While pop is a bit different, because the front list is actually lazy stream, we need force an evaluation. All the others are as same as before.

$$pop(Q) = balance(\mathcal{F}', R, \mathcal{R}_s) \tag{11.31}$$

Here $\mathcal{F}'$, force one evaluation to $\mathcal{F}$, the Scheme/Lisp code regarding to this equation is as the following.

```
(define (pop q)
  (balance (stream-cdr (front-lst q)) (rear q) (rots q)))
```

For illustration purpose, we skip the error handling (such as pop from an empty queue etc) here.

And one can access the top element in the queue by extract from the front list stream.

```
(define (front q) (stream-car (front-lst q)))
```

The balance function first checks if the computation stream is completely exhausted, and starts new rotation accordingly; otherwise, it just consumes one evaluation by enforcing the lazy stream.

$$balance(Q) = \begin{cases} Queue(\mathcal{F}', \Phi, \mathcal{F}') & : \quad \mathcal{R}_s = \Phi \\ Queue(\mathcal{F}, R, \mathcal{R}'_s) & : \quad otherwise \end{cases} \tag{11.32}$$

Here $\mathcal{F}'$ is defined to start a new rotation.

$$\mathcal{F}' = rotate(F, R, \Phi) \tag{11.33}$$

The relative Scheme/Lisp program is listed accordingly.

```
(define (balance f r s)
  (if (stream-null? s)
      (let ((newf (rotate f r '())))
    (make-queue newf '() newf))
      (make-queue f r (stream-cdr s))))
```

The implementation of incremental rotate function is just as same as what we analyzed above.

```
(define (rotate xs ys acc)
  (if (stream-null? xs)
      (cons-stream (car ys) acc)
      (cons-stream (stream-car xs)
           (rotate (stream-cdr xs) (cdr ys)
               (cons-stream (car ys) acc)))))
```

We used explicit lazy evaluation in Scheme/Lisp. Actually, this program can be very short by using lazy programming languages, for example, Haskell.

```
data LazyRTQueue a = LQ [a] [a] [a] -- front, rear, f ++ reverse r

instance Queue LazyRTQueue where
    empty = LQ [] [] []

    isEmpty (LQ f _ _) = null f

    -- O(1) time push
    push (LQ f r rot) x = balance f (x:r) rot

    -- O(1) time pop
    pop (LQ (_:f) r rot) = balance f r rot

    front (LQ (x:_) _ _) = x

balance f r [] = let f' = rotate f r [] in LQ f' [] f'
balance f r (_:rot) = LQ f r rot

rotate [] [y] acc = y:acc
rotate (x:xs) (y:ys) acc = x : rotate xs ys (y:acc)
```

11.7 Notes and short summary

Just as mentioned in the beginning of this book in the first chapter, queue isn't so simple as it was thought. We've tries to explain algorithms and data structures both in imperative and in function approaches; Sometimes, it gives impression that functional way is simpler and more expressive in most time. However, there are still plenty of areas, that more studies and works are needed to give equivalent functional solution. Queue is such an important topic, that it links to many fundamental purely functional data structures.

That's why Chris Okasaki made intensively study and took a great amount of discussions in [6]. With purely functional queue solved, we can easily implement dequeue with the similar approach revealed in this chapter. As we can handle elements effectively in both head and tail, we can advance one step ahead to realize sequence data structures, which support fast concatenate, and finally we can realize random access data structures to mimic array in imperative settings. The details will be explained in later chapters.

Note that, although we haven't mentioned priority queue, it's quite possible to realized it with heaps. We have covered topic of heaps in several previous chapters.

Exercise 11.6

- Realize dequeue, wich support adding and removing elements on both sides in constant $O(1)$ time in purely functional way.
- Realize dequeue in a symmetric solution only with array in your favorite imperative programming language.

Bibliography

[1] Maged M. Michael and Michael L. Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms”. http://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html

[2] Herb Sutter. “Writing a Generalized Concurrent Queue”. Dr. Dobb’s Oct 29, 2008. http://drdobbs.com/cpp/211601363?pgno=1

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

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

[5] Wikipedia. “Tail-call”. http://en.wikipedia.org/wiki/Tail_call

[6] Wikipedia. “Recursion (computer science)”. http://en.wikipedia.org/wiki/Recursion_(computer_science)#Tail-recursive_functions

[7] Harold Abelson, Gerald Jay Sussman, Julie Sussman. “Structure and Interpretation of Computer Programs, 2nd Edition”. MIT Press, 1996, ISBN 0-262-51087-1

Sequences, The last brick
Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

Chapter 12

Sequences, The last brick

12.1 Introduction

In the first chapter of this book, which introduced binary search tree as the 'hello world' data structure, we mentioned that neither queue nor array is simple if realized not only in imperative way, but also in functional approach. In previous chapter, we explained functional queue, which achieves the similar performance as its imperative counterpart. In this chapter, we'll dive into the topic of array-like data structures.

We have introduced several data structures in this book so far, and it seems that functional approaches typically bring more expressive and elegant solution. However, there are some areas, people haven't found competitive purely functional solutions which can match the imperative ones. For instance, the Ukkonen linear time suffix tree construction algorithm. another examples is Hashing table. Array is also among them.

Array is trivial in imperative settings, it enables randomly accessing any elements with index in constant $O(1)$ time. However, this performance target can't be achieved directly in purely functional settings as there is only list can be used.

In this chapter, we are going to abstract the concept of array to sequences. Which support the following features

- Element can be inserted to or removed from the head of the sequence quickly in $O(1)$ time;
- Element can be inserted to or removed from the head of the sequence quickly in $O(1)$ time;
- Support concatenate two sequences quickly (faster than linear time);
- Support randomly access and update any element quickly;
- Support split at any position quickly;

We call these features abstract sequence properties, and it easy to see the fact that even array (here means plain-array) in imperative settings can't meet them all at the same time.

We'll provide three solutions in this chapter. Firstly, we'll introduce a solution based on binary tree forest and numeric representation; Secondly, we'll show a concatenate-able list solution; Finally, we'll give the finger tree solution.

Most of the results are based on Chris, Okasaki's work in [6].

12.2 Binary random access list

12.2.1 Review of plain-array and list

Let's review the performance of plain-array and singly linked-list so that we know how they perform in different cases.

operation	Array	Linked-list
operation on head	$O(N)$	$O(1)$
operation on tail	$O(1)$	$O(N)$
access at random position	$O(1)$	average $O(N)$
remove at given position	average $O(N)$	$O(1)$
concatenate	$O(N_2)$	$O(N_1)$

Because we hold the head of linked list, operations on head such as insert and remove perform in constant time; while we need traverse to the end to perform removing or appending on tail; Given a position i, it need traverse i elements to access it. Once we are at that position, removing element from there is just bound to constant time by modifying some pointers. In order to concatenate two linked-lists, we need traverse to the end of the first one, and link it to the second one, which is bound to the length of the first linked-list;

On the other hand, for array, we must prepare free cell for inserting a new element to the head of it, and we need release the first cell after the first element being removed, all these two operations are achieved by shifting all the rest elements forward or backward, which costs linear time. While the operations on the tail of array are trivial constant time. Array also support accessing random position i by nature; However, removing the element at that position causes shifting all elements after it one position ahead. In order to concatenate two arrays, we need copy all elements from the second one to the end of the first one (ignore the memory re-allocation details), which is proportion to the length of the second array.

In the chapter about binomial heaps, we have explained the idea of using forest, which is a list of trees. It brings us the merit that, for any given number N, by representing it in binary number, we know how many binomial trees need to hold them. That each bit of 1 represents a binomial tree of that rank of bit. We can go one step ahead, if we have a N nodes binomial heap, for any given index $1 < i < N$, we can quickly know which binomial tree in the heap holds the i-th node.

12.2.2 Represent sequence by trees

One solution to realize a random-access sequence is to manage the sequence with a forest of complete binary trees. Figure 12.1 shows how we attach such trees to a sequence of numbers.

Here two trees t_1 and t_2 are used to represent sequence $\{x_1, x_2, x_3, x_4, x_5, x_6\}$. The size of binary tree t_1 is 2. The first two elements $\{x_1, x_2\}$ are leaves of t_1;

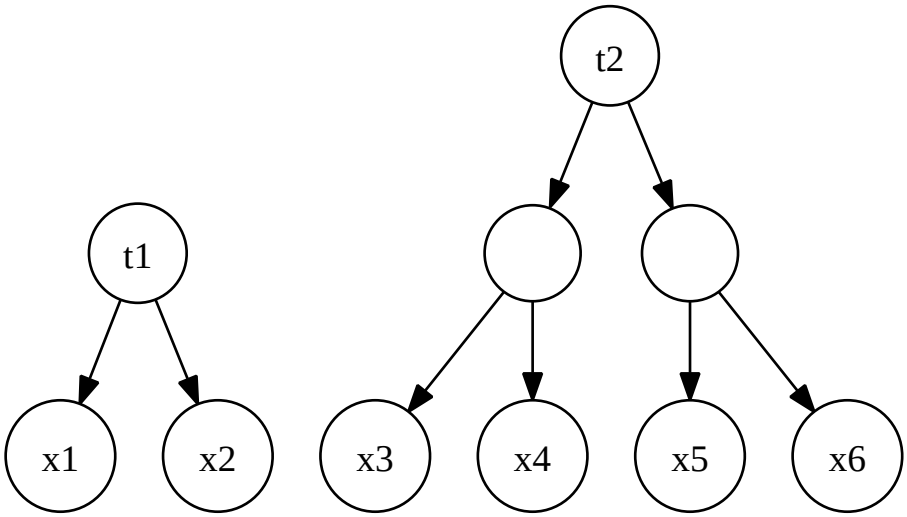


Figure 12.1: A sequence of 6 elements can be represented in a forest.

the size of binary tree t_2 is 4. The next four elements $\{x_3, x_4, x_5, x_6\}$ are leaves of t_2.

For a complete binary tree, we define the depth as 0 if the tree has only a leaf. The tree is denoted as as t_i if its depth is $i + 1$. It's obvious that there are 2^i leaves in t_i.

For any sequence contains N elements, it can be turned to a forest of complete binary trees in this manner. First we represent N in binary number like below.

$$N = 2^0 e_0 + 2^1 e_1 + ... + 2^M e_M \tag{12.1}$$

Where e_i is either 1 or 0, so $N = (e_M e_{M-1} ... e_1 e_0)_2$. If $e_i \neq 0$, we then need a complete binary tree with size 2^i, For example in figure 12.1, as the length of sequence is 6, which is $(110)_2$ in binary. The lowest bit is 0, so we needn't a tree of size 1; the second bit is 1, so we need a tree of size 2, which has depth of 2; the highest bit is also 1, thus we need a tree of size 4, which has depth of 3.

This method represents the sequence $\{x_1, x_2, ..., x_N\}$ to a list of trees $\{t_0, t_1, ..., t_M\}$ where t_i is either empty if $e_i = 0$ or a complete binary tree if $e_i = 1$. We call this representation as *Binary Random Access List* [6].

We can reused the definition of binary tree. For example, the following Haskell program defines the tree and the binary random access list.

```
data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)  -- size, left, right

type BRAList a = [Tree a]
```

The only difference from the typical binary tree is that we augment the size information to the tree. This enable us to get the size without calculation at every time. For instance.

```
size (Leaf _) = 1
size (Node sz _ _) = sz
```

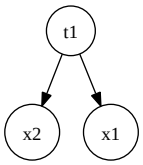
12.2.3 Insertion to the head of the sequence

The new forest representation of sequence enables many operation effectively. For example, the operation of inserting a new element y in front of sequence can be realized as the following.

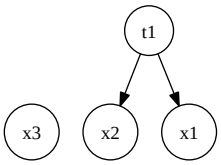
1. Create a tree t', with y as the only one leaf;
2. Examine the first tree in the forest, compare its size with t', if its size is greater than t', we just let t' be the new head of the forest, since the forest is a linked-list of tree, insert t' to its head is trivial operation, which is bound to constant $O(1)$ time;
3. Otherwise, if the size of first tree in the forest is equal to t', let's denote this tree in the forest as t_i, we can construct a new binary tree t'_{i+1} by linking t_i and t' as its left and right children. After that, we recursively try to insert t'_{i+1} to the forest.

Figure 12.2 and 12.3 illustrate the steps of inserting element $x_1, x_2, ..., x_6$ to an empty tree.

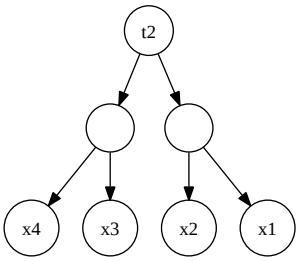
(a) A singleton leaf of x_1



(b) Insert x_2. It causes linking, results a tree of height 1.



(c) Insert x_3. the result is two trees, t_1 and t_2



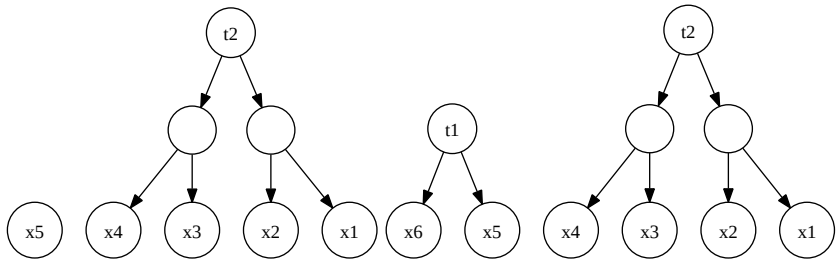
(d) Insert x_4. It first causes linking two leafs to a binary tree, then it performs linking again, which results a final tree of height 2.

Figure 12.2: Steps of inserting elements to an empty list, 1

As there are at most M trees in the forest, and M is bound to $O(\lg N)$, so the insertion to head algorithm is ensured to perform in $O(\lg N)$ even in worst case. We'll prove the amortized performance is $O(1)$ later.

Let's formalize the algorithm to equations. we define the function of inserting an element in front of a sequence as $insert(S, x)$.

$$insert(S, x) = insertTree(S, leaf(x)) \tag{12.2}$$



(a) Insert x_5. The forest is a leaf (t_0) and t_2.

(b) Insert x_6. It links two leaf to t_1.

Figure 12.3: Steps of inserting elements to an empty list, 2

This function just wrap element x to a singleton tree with a leaf, and call $insertTree$ to insert this tree to the forest. Suppose the forest $F = \{t_1, t_2, ...\}$ if it's not empty, and $F' = \{t_2, t_3, ...\}$ is the rest of trees without the first one.

$$insertTree(F, t) = \begin{cases} \{t\} & : & F = \Phi \\ \{t\} \cup F & : & size(t) < size(t_1) \\ insertTree(F', link(t, t_1)) & : & otherwise \end{cases} \tag{12.3}$$

Where function $link(t_1, t_2)$ create a new tree from two small trees with same size. Suppose function $tree(s, t_1, t_2)$ create a tree, set its size as s, makes t_1 as the left child, and t_2 as the right child, linking can be realized as below.

$$link(t_1, t_2) = tree(size(t_1) + size(t_2), t_1, t_2) \tag{12.4}$$

The relative Haskell programs can be given by translating these equations.

```
cons :: a → BRAList a → BRAList a
cons x ts = insertTree ts (Leaf x)

insertTree :: BRAList a → Tree a → BRAList a
insertTree [] t = [t]
insertTree (t':ts) t = if size t < size t' then  t:t':ts
                       else insertTree ts (link t t')

-- Precondition: rank t1 = rank t2
link :: Tree a → Tree a → Tree a
link t1 t2 = Node (size t1 + size t2) t1 t2
```

Here we use the Lisp tradition to name the function that insert an element before a list as 'cons'.

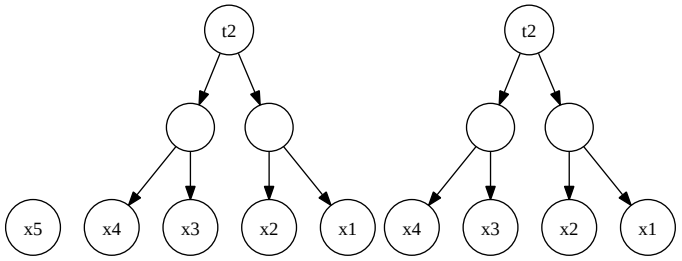
Remove the element from the head of the sequence

It's not complex to realize the inverse operation of 'cons', which can remove element from the head of the sequence.

- If the first tree in the forest is a singleton leaf, remove this tree from the forest;

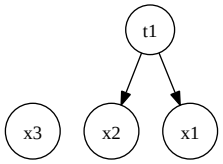
- otherwise, we can halve the first tree by unlinking its two children, so the first tree in the forest becomes two trees, we recursively halve the first tree until it turns to be a leaf.

Figure 12.4 illustrates the steps of removing elements from the head of the sequence.



(a) A sequence of 5 elements

(b) Result of removing x_5, the leaf is removed.



(c) Result of removing x_4, As there is not leaf tree, the tree is firstly divided into two sub trees of size 2. The first tree is next divided again into two leafs, after that, the first leaf, which contains x_4 is removed. What left in the forest is a leaf tree of x_3, and a tree of size 2 with elements x_2, x_1.

Figure 12.4: Steps of removing elements from head

If we assume the sequence isn't empty, so that we can skip the error handling such as trying to remove an element from an empty sequence, this can be expressed with the following equation. We denote the forest $F = \{t_1, t_2, ...\}$ and the trees without the first one as $F' = \{t_2, t_3, ...\}$

$$extractTree(F) = \begin{cases} (t_1, F') & : \quad t_1 \quad \text{is leaf} \\ extractTree(\{t_l, t_r\} \cup F') & : \quad otherwise \end{cases} \tag{12.5}$$

where $\{t_l, t_r\} = unlink(t_1)$ are the two children of t_1.
It can be translated to Haskell programs like below.

```
extractTree (t@(Leaf x):ts) = (t, ts)
extractTree (t@(Node _ t1 t2):ts) = extractTree (t1:t2:ts)
```

With this function defined, it's convenient to give $head()$ and $tail()$ functions, the former returns the first element in the sequence, the latter return the rest.

$$head(S) = key(first(extractTree(S))) \tag{12.6}$$

$$tail(S) = second(extractTree(S)) \tag{12.7}$$

Where function $first()$ returns the first element in a paired-value (as known as tuple); $second()$ returns the second element respectively. function $key()$ is used to access the element inside a leaf. Below are Haskell programs corresponding to these two equations.

```
head' ts = x where (Leaf x, _) = extractTree ts
tail' = snd ∘ extractTree
```

Note that as `head` and `tail` functions have already been defined in Haskell standard library, we given them apostrophes to make them distinct. (another option is to hide the standard ones by importing. We skip the details as they are language specific).

Random access the element in binary random access list

As trees in the forest help managing the elements in blocks, giving an arbitrary index, it's easy to locate which tree this element is stored, after that performing a search in the tree yields the result. As all trees are binary (more accurate, complete binary tree), the search is essentially binary search, which is bound to the logarithm of the tree size. This brings us a faster random access capability than linear search in linked-list setting.

Given an index i, and a sequence S, which is actually a forest of trees, the algorithm is executed as the following [1].

1. Compare i with the size of the first tree T_1 in the forest, if i is less than or equal to the size, the element exists in T_1, perform looking up in T_1;
2. Otherwise, decrease i by the size of T_1, and repeat the previous step in the rest of the trees in the forest.

This algorithm can be represented as the below equation.

$$get(S, i) = \begin{cases} lookupTree(T_1, i) & : \quad i \leq |T_1| \\ get(S', i - |T_1|) & : \quad otherwise \end{cases} \tag{12.8}$$

Where $|T| = size(T)$, and $S' = \{T_2, T_3, ...\}$ is the rest of trees without the first one in the forest. Note that we don't handle out of bound error case, this is left as an exercise to the reader.

Function $lookupTree()$ is just a binary search algorithm, if the index i is 1, we just return the root of the tree, otherwise, we halve the tree by unlinking, if i is less than or equal to the size of the halved tree, we recursively look up the left tree, otherwise, we look up the right tree.

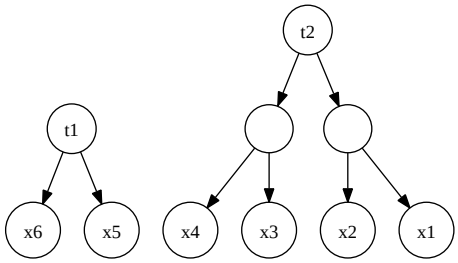
$$lookupTree(T, i) = \begin{cases} root(T) & : \quad i = 1 \\ lookupTree(left(T)) & : \quad i \leq \lfloor \frac{|T|}{2} \rfloor \\ lookupTree(right(T)) & : \quad otherwise \end{cases} \tag{12.9}$$

Where function $left()$ returns the left tree T_l of T, while $right()$ returns T_r. The corresponding Haskell program is given as below.

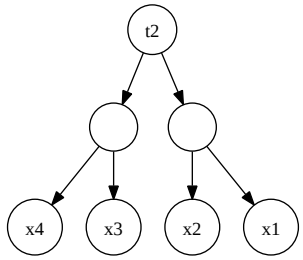
[1] We follow the tradition that the index i starts from 1 in algorithm description; while it starts from 0 in most programming languages

```
getAt (t:ts) i = if i < size t then lookupTree t i
                 else getAt ts (i - size t)

lookupTree (Leaf x) 0 = x
lookupTree (Node sz t1 t2) i = if i < sz `div` 2 then lookupTree t1 i
                               else lookupTree t2 (i - sz `div` 2)
```

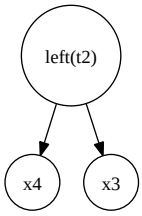
Figure 12.5 illustrates the steps of looking up the 4-th element in a sequence of size 6. It first examine the first tree, since the size is 2 which is smaller than 4, so it goes on looking up for the second tree with the updated index $i' = 4 - 2$, which is the 2nd element in the rest of the forest. As the size of the next tree is 4, which is greater than 2, so the element to be searched should be located in this tree. It then examines the left sub tree since the new index 2 is not greater than the half size 4/2=2; The process next visits the right grand-child, and the final result is returned.



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



(b) $getAt(S', 4-2) \Rightarrow lookupTree(t_2, 2)$



(c) $2 \leq \lfloor size(t_2)/2 \rfloor \Rightarrow lookupTree(left(t_2), 2)$

(d) $lookupTree(right(left(t_2)), 1)$, x_3 is returned.

Figure 12.5: Steps of locating the 4-th element in a sequence.

By using the similar idea, we can update element at any arbitrary position i. We first compare the size of the first tree T_1 in the forest with i, if it is less than i, it means the element to be updated doesn't exist in the first tree. We recursively examine the next tree in the forest, comparing it with $i - |T_1|$, where $|T_1|$ represents the size of the first tree. Otherwise if this size is greater than or equal to i, the element is in the tree, we halve the tree recursively until to get a leaf, at this stage, we can replace the element of this leaf with a new one.

$$set(S, i, x) = \begin{cases} \{updateTree(T_1, i, x)\} \cup S' & : \quad i < |T_1| \\ \{T_1\} \cup set(S', i - |T_1|, x) & : \quad otherwise \end{cases} \tag{12.10}$$

Where $S' = \{T_2, T_3, ...\}$ is the rest of the trees in the forest without the first one.

Function $setTree(T, i, x)$ performs a tree search and replace the i-th element with the given value x.

$$setTree(T, i, x) = \begin{cases} leaf(x) & : & i = 0 \wedge |T| = 1 \\ tree(|T|, setTree(T_l, i, x), T_r) & : & i < \lfloor \frac{|T|}{2} \rfloor \\ tree(|T|, T_l, setTree(T_r, i - \lfloor \frac{|T|}{2} \rfloor, x)) & : & otherwise \end{cases} \tag{12.11}$$

Where T_l and T_r are left and right sub tree of T respectively. The following Haskell program translates the equation accordingly.

```
setAt :: BRAList a → Int → a → BRAList a
setAt (t:ts) i x = if i < size t then (updateTree t i x):ts
                   else t:setAt ts (i-size t) x

updateTree :: Tree a → Int → a → Tree a
updateTree (Leaf _) 0 x = Leaf x
updateTree (Node sz t1 t2) i x =
    if i < sz `div` 2 then Node sz (updateTree t1 i x) t2
    else Node sz t1 (updateTree t2 (i - sz `div` 2) x)
```

As the nature of complete binary search tree, for a sequence with N elements, which is represented by binary random access list, the number of trees in the forest is bound to $O(\lg N)$. Thus it takes $O(\lg N)$ time to locate the tree for arbitrary index i, that contains the element. the followed tree search is bound to the heights of the tree, which is $O(\lg N)$ as well. So the total performance of random access is $O(\lg N)$.

Exercise 12.1

1. The random access algorithm given in this section doesn't handle the error such as out of bound index at all. Modify the algorithm to handle this case, and implement it in your favorite programming language.

2. It's quite possible to realize the binary random access list in imperative settings, which is benefited with fast operation on the head of the sequence. the random access can be realized in two steps: firstly locate the tree, secondly use the capability of constant random access of array. Write a program to implement it in your favorite imperative programming language.

12.3 Numeric representation for binary random access list

In previous section, we mentioned that for any sequence with N elements, we can represent N in binary format so that $N = 2^0 e_0 + 2^1 e_1 + ... + 2^M e_M$. Where e_i is the i-th bit, which can be either 0 or 1. If $e_i \neq 0$ it means that there is a complete binary tree with size 2^i.

This fact indicates us that there is an explicit relationship between the binary form of N and the forest. Insertion a new element on the head can be simulated

by increasing the binary number by one; while remove an element from the head mimics the decreasing of the corresponding binary number by one. This is as known as *numeric representation* [6].

In order to represent the binary access list with binary number, we can define two states for a bit. That $Zero$ means there is no such a tree with size which is corresponding to the bit, while One, means such tree exists in the forest. And we can attach the tree with the state if it is One.

The following Haskell program for instance defines such states.

```
data Digit a = Zero
             | One (Tree a)

type RAList a = [Digit a]
```

Here we reuse the definition of complete binary tree and attach it to the state One. Note that we cache the size information in the tree as well.

With digit defined, forest can be treated as a list of digits. Let's see how inserting a new element can be realized as binary number increasing. Suppose function $one(t)$ creates a One state and attaches tree t to it. And function $getTree(s)$ get the tree which is attached to the One state s. The sequence S is a list of digits of states that $S = \{s_1, s_2, ...\}$, and S' is the rest of digits with the first one removed.

$$insertTree(S, t) = \begin{cases} \{one(t)\} & : & S = \Phi \\ \{one(t)\} \cup S' & : & s_1 = Zero \\ \{Zero\} \cup insertTree(S', link(t, getTree(s_1))) & : & otherwise \end{cases} \tag{12.12}$$

When we insert a new tree t to a forest S of binary digits, If the forest is empty, we just create a One state, attach the tree to it, and make this state the only digit of the binary number. This is just like $0 + 1 = 1$;

Otherwise if the forest isn't empty, we need examine the first digit of the binary number. If the first digit is $Zero$, we just create a One state, attach the tree, and replace the $Zero$ state with the new created One state. This is just like $(...digits...0)_2 + 1 = (...digits...1)_2$. For example $6+1 = (110)_2+1 = (111)_2 = 7$.

The last case is that the first digit is One, here we make assumption that the tree t to be inserted has the same size with the tree attached to this One state at this stage. This can be ensured by calling this function from inserting a leaf, so that the size of the tree to be inserted grows in a series of $1, 2, 4, ..., 2^i, ...$ In such case, we need link these two trees (one is t, the other is the tree attached to the One state), and recursively insert the linked result to the rest of the digits. Note that the previous One state has to be replaced with a $Zero$ state. This is just like $(...digits...1)_2 + 1 = (...digits'...0)_2$, where $(...digits'...)_2 = (...digits...)_2 + 1$. For example $7 + 1 = (111)_2 + 1 = (1000)_2 = 8$

Translating this algorithm to Haskell yields the following program.

```
insertTree :: RAList a → Tree a → RAList a
insertTree [] t = [One t]
insertTree (Zero:ts) t = One t : ts
insertTree (One t' :ts) t = Zero : insertTree ts (link t t')
```

All the other functions, including $link(), cons()$ etc. are as same as before.

Next let's see how removing an element from a sequence can be represented as binary number deduction. If the sequence is a singleton *One* state attached with a leaf. After removal, it becomes empty. This is just like $1 - 1 = 0$;

Otherwise, we examine the first digit, if it is *One* state, it will be replaced with a *Zero* state to indicate that this tree will be no longer exist in the forest as it being removed. This is just like $(...digits...1)_2 - 1 = (...digits...0)_2$. For example $7 - 1 = (111)_2 - 1 = (110)_2 = 6$;

If the first digit in the sequence is a *Zero* state, we have to borrow from the further digits for removal. We recursively extract a tree from the rest digits, and halve the extracted tree to its two children. Then the *Zero* state will be replaced with a *One* state attached with the right children, and the left children is removed. This is something like $(...digits...0)_2 - 1 = (...digits'...1)_2$, where $(...digits'...)_2 = (...digits)_2 - 1$. For example $4 - 1 = (100)_2 - 1 = (11)_2 = 3$.The following equation illustrated this algorithm.

$$extractTree(S) = \begin{cases} (t, \Phi) & : & S = \{one(t)\} \\ (t, S') & : & s_1 = one(t) \\ (t_l, \{one(t_r)\} \cup S'') & : & otherwise \end{cases} \tag{12.13}$$

Where $(t', S'') = extractTree(S')$, t_l and t_r are left and right sub-trees of t'. All other functions, including $head()$, $tail()$ are as same as before.

Numeric representation doesn't change the performance of binary random access list, readers can refer to [2] for detailed discussion. Let's take for example, analyze the average performance (or amortized) of insertion on head algorithm by using aggregation analysis.

Considering the process of inserting $N = 2^m$ elements to an empty binary random access list. The numeric representation of the forest can be listed as the following.

i	forest (MSB ... LSB)
0	0, 0, ..., 0, 0
1	0, 0, ..., 0, 1
2	0, 0, ..., 1, 0
3	0, 0, ..., 1, 1
...	...
$2^m - 1$	1, 1, ..., 1, 1
2^m	1, 0, 0, ..., 0, 0
bits changed	1, 1, 2, ... 2^{m-1}. 2^m

The LSB of the forest changed every time when there is a new element inserted, it costs 2^m units of computation; The next bit changes every two times due to a linking operation, so it costs 2^{m-1} units; the bit next to MSB of the forest changed only one time which links all previous trees to a big tree as the only one in the forest. This happens at the half time of the total insertion process, and after the last element is inserted, the MSB flips to 1.

Sum these costs up yield to the total cost $T = 1+1+2+3+...+2^{m_1}+2^m = 2^{m+1}$ So the average cost for one insertion is

$$O(T/N) = O(\frac{2^{m+1}}{2^m}) = O(1) \tag{12.14}$$

Which proves that the insertion algorithm performs in amortized $O(1)$ constant time. The proof for deletion are left as an exercise to the reader.

12.3.1 Imperative binary access list

It's trivial to implement the binary access list by using binary trees, and the recursion can be eliminated by updating the focused tree in loops. This is left as an exercise to the reader. In this section, we'll show some different imperative implementation by using the properties of numeric representation.

Remind the chapter about binary heap. Binary heap can be represented by implicit array. We can use similar approach that use an array of 1 element to represent the leaf; use an array of 2 elements to represent a binary tree of height 1; and use an array of 2^m to represent a complete binary tree of height m.

This brings us the capability of accessing any element with index directly instead of divide and conquer tree search. However, the tree linking operation has to be implemented as array copying as the expense.

The following ANSI C code defines such a forest.

```
#define M sizeof(int) * 8
typedef int Key;

struct List {
  int n;
  Key* tree[M];
};
```

Where n is the number of the elements stored in this forest. Of course we can avoid limiting the max number of trees by using dynamic arrays, for example as the following ISO C++ code.

```
template<typename Key>
struct List {
  int n;
  vector<vector<key> > tree;
};
```

For illustration purpose only, we use ANSI C here, readers can find the complete ISO C++ example programs along with this book.

Let's review the insertion process, if the first tree is empty (a *Zero* digit), we simply set the first tree as a leaf of the new element to be inserted; otherwise, the insertion will cause tree linking anyway, and such linking may be recursive until it reach a position (digit) that the corresponding tree is empty. The numeric representation reveals an important fact that if the first, second, ..., $(i-1)$-th trees all exist, and the i-th tree is empty, the result is creating a tree of size 2^i, and all the elements together with the new element to be inserted are stored in this new created tree. What's more, all trees after position i are kept as same as before.

Is there any good methods to locate this i position? As we can use binary number to represent the forest of N element, after a new element is inserted, N increases to $N+1$. Compare the binary form of N and $N+1$, we find that all bits before i change from 1 to 0, the i-th bit flip from 0 to 1, and all the bits after i keep unchanged. So we can use bit-wise exclusive or ($\oplus$) to detect this bit. Here is the algorithm.

function NUMBER-OF-BITS(N)
  $i \leftarrow 0$
  while $\lfloor \frac{N}{2} \rfloor \neq 0$ **do**

```
        N ← ⌊N/2⌋
        i ← i + 1
    return i
```

$i \leftarrow$ NUMBER-OF-BITS$(N \oplus (N+1))$

And it can be easily implemented with bit shifting, for example the below ANSI C code.

```
int nbits(int n) {
  int i=0;
  while(n >>= 1)
    ++i;
  return i;
}
```

So the imperative insertion algorithm can be realized by first locating the bit which flip from 0 to 1, then creating a new array of size 2^i to represent a complete binary tree, and moving content of all trees before this bit to this array as well as the new element to be inserted.

```
function INSERT(L, x)
    i ← NUMBER-OF-BITS(N ⊕ (N + 1))
    TREE(L)[i + 1] ← CREATE-ARRAY(2^i)
    l ← 1
    TREE(L)[i + 1][l] ← x
    for j ∈ [1, i] do
        for k ∈ [1, 2^j] do
            l ← l + 1
            TREE(L)[i + 1][l] ← TREE(L)[j][k]
        TREE(L)[j] ← NIL
    SIZE(L) ← SIZE(L) + 1
    return L
```

The corresponding ANSI C program is given as the following.

```
struct List insert(struct List a, Key x) {
  int i, j, sz;
  Key* xs;
  i = nbits( (a.n+1) ^ a.n );
  xs = a.tree[i] = (Key*)malloc(sizeof(Key)*(1<<i));
  for(j=0, *xs++ = x, sz = 1; j<i; ++j, sz <<= 1) {
    memcpy((void*)xs, (void*)a.tree[j], sizeof(Key)*(sz));
    xs += sz;
    free(a.tree[j]);
    a.tree[j] = NULL;
  }
  ++a.n;
  return a;
}
```

However, the performance in theory isn't as good as before. This is because the linking operation downgrade from $O(1)$ constant time to linear array copying.

We can again calculate the average (amortized) performance by using aggregation analysis. When insert $N = 2^m$ elements to an empty list which is

represented by implicit binary trees in arrays, the numeric presentation of the forest of arrays are as same as before except for the cost of bit flipping.

i	forest (MSB ... LSB)
0	0, 0, ..., 0, 0
1	0, 0, ..., 0, 1
2	0, 0, ..., 1, 0
3	0, 0, ..., 1, 1
...	...
$2^m - 1$	1, 1, ..., 1, 1
2^m	1, 0, 0, ..., 0, 0
bit change cost	1×2^m, $1 \times 2^{m-1}$, $2 \times 2^{m-2}$, ... $2^{m-2} \times 2$, $2^{m-1} \times 1$

The LSB of the forest changed every time when there is a new element inserted, however, it creates leaf tree and performs copying only it changes from 0 to 1, so the cost is half of N unit, which is 2^{m-1}; The next bit flips as half as the LSB. Each time the bit gets flipped to 1, it copies the first tree as well as the new element to the second tree. the the cost of flipping a bit to 1 in this bit is 2 units, but not 1; For the MSB, it only flips to 1 at the last time, but the cost of flipping this bit, is copying all the previous trees to fill the array of size 2^m.

Summing all to cost and distributing them to the N times of insertion yields the amortized performance as below.

$$\begin{aligned} O(T/N) &= O(\frac{1\times 2^m + 1\times 2^{m-1} + 2\times 2^{m-2} + ... + 2^{m-1}\times 1}{2^m}) \\ &= O(1 + \frac{m}{2}) \\ &= O(m) \end{aligned} \tag{12.15}$$

As $m = O(\lg N)$, so the amortized performance downgrade from constant time to logarithm, although it is still faster than the normal array insertion which is $O(N)$ in average.

The random accessing gets a bit faster because we can use array indexing instead of tree search.

function GET(L, i)
  for each $t \in$ TREES(L) **do**
    if $t \neq$ NIL **then**
      if $i \leq$ SIZE(t) **then**
        return t[i]
      else
        $i \leftarrow i-$ SIZE(t)

Here we skip the error handling such as out of bound indexing etc. The ANSI C program of this algorithm is like the following.

```
Key get(struct List a, int i) {
  int j, sz;
  for(j = 0, sz = 1; j < M; ++j, sz <<= 1)
    if(a.tree[j]) {
      if(i < sz)
        break;
      i -= sz;
    }
  return a.tree[j][i];
}
```

The imperative removal and random mutating algorithms are left as exercises to the reader.

Exercise 12.2

1. Please implement the random access algorithms, including looking up and updating, for binary random access list with numeric representation in your favorite programming language.

2. Prove that the amortized performance of deletion is $O(1)$ constant time by using aggregation analysis.

3. Design and implement the binary random access list by implicit array in your favorite imperative programming language.

12.4 Imperative paired-array list

12.4.1 Definition

In previous chapter about queue, a symmetric solution of paired-array is presented. It is capable to operate on both ends of the list. Because the nature that array supports fast random access. It can be also used to realize a fast random access sequence in imperative setting.

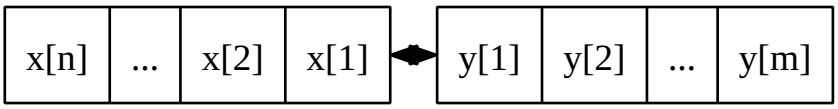


Figure 12.6: A paired-array list, which is consist of 2 arrays linking in head-head manner.

Figure 12.6 shows the design of paired-array list. Tow arrays are linked in head-head manner. To insert a new element on the head of the sequence, the element is appended at the end of front list; To append a new element on the tail of the sequence, the element is appended at the end of rear list;

Here is a ISO C++ code snippet to define the this data structure.

```
template<typename Key>
struct List {
  int n, m;
  vector<Key> front;
  vector<Key> rear;

  List() : n(0), m(0) {}
  int size() { return n + m; }
};
```

Here we use vector provides in standard library to cover the dynamic memory management issues, so that we can concentrate on the algorithm design.

12.4.2 Insertion and appending

Suppose function FRONT(L) returns the front array, while REAR(L) returns the rear array. For illustration purpose, we assume the arrays are dynamic allocated. inserting and appending can be realized as the following.

function INSERT(L, x)
  $F \leftarrow$ FRONT(L)
  SIZE(F) $\leftarrow$ SIZE(F) + 1
  F[SIZE(F)] $\leftarrow x$

function APPEND(L, x)
  $R \leftarrow$ REAR(L)
  SIZE(R) $\leftarrow$ SIZE(R) + 1
  R[SIZE(R)] $\leftarrow x$

As all the above operations manipulate the front and rear array on tail, they are all constant $O(1)$ time. And the following are the corresponding ISO C++ programs.

```
template<typename Key>
void insert(List<Key>& xs, Key x) {
  ++xs.n;
  xs.front.push_back(x);
}

template<typename Key>
void append(List<Key>& xs, Key x) {
  ++xs.m;
  xs.rear.push_back(x);
}
```

12.4.3 random access

As the inner data structure is array (dynamic array as vector), which supports random access by nature, it's trivial to implement constant time indexing algorithm.

function GET(L, i)
  $F \leftarrow$ FRONT(L)
  $N \leftarrow$ SIZE(F)
  if $i \leq N$ **then**
    return $F[N - i + 1]$
  else
    REAR(L)[$i - N$]

Here the index $i \in [1, |L|]$ starts from 1. If it is not greater than the size of front array, the element is stored in front. However, as front and rear arrays are connect head-to-head, so the elements in front array are in reverse order. We need locate the element by subtracting the size of front array by i; If the index i is greater than the size of front array, the element is stored in rear array. Since elements are stored in normal order in rear, we just need subtract the index i by an offset which is the size of front array.

Here is the ISO C++ program implements this algorithm.

```
template<typename Key>
Key get(List<Key>& xs, int i) {
  if( i < xs.n )
    return xs.front[xs.n-i-1];
  else
    return xs.rear[i-xs.n];
}
```

The random mutating algorithm is left as an exercise to the reader.

12.4.4 removing and balancing

Removing isn't as simple as insertion and appending. This is because we must handle the condition that one array (either front or rear) becomes empty due to removal, while the other still contains elements. In extreme case, the list turns to be quite unbalanced. So we must fix it to resume the balance.

One idea is to trigger this fixing when either front or rear array becomes empty. We just cut the other array in half, and reverse the first half to form the new pair. The algorithm is described as the following.

function BALANCE(L)
  $F \leftarrow$ FRONT(L), $R \leftarrow$ REAR(L)
  $N \leftarrow$ SIZE(F), $M \leftarrow$ SIZE(R)
  if $F = \Phi$ **then**
    $F \leftarrow$ REVERSE($R[1 \ldots \lfloor \frac{M}{2} \rfloor]$)
    $R \leftarrow R[\lfloor \frac{M}{2} \rfloor + 1 \ldots M]$
  else if $R = \Phi$ **then**
    $R \leftarrow$ REVERSE($F[1 \ldots \lfloor \frac{N}{2} \rfloor]$)
    $F \leftarrow F[\lfloor \frac{N}{2} \rfloor + 1 \ldots N]$

Actually, the operations are symmetric for the case that front is empty and the case that rear is empty. Another approach is to swap the front and rear for one symmetric case and recursive resumes the balance, then swap the front and rear back. For example below ISO C++ program uses this method.

```
template<typename Key>
void balance(List<Key>& xs) {
  if(xs.n == 0) {
    back_insert_iterator<vector<Key> > i(xs.front);
    reverse_copy(xs.rear.begin(), xs.rear.begin() + xs.m/2, i);
    xs.rear.erase(xs.rear.begin(), xs.rear.begin() +xs.m/2);
    xs.n = xs.m/2;
    xs.m -= xs.n;
  }
  else if(xs.m == 0) {
    swap(xs.front, xs.rear);
    swap(xs.n, xs.m);
    balance(xs);
    swap(xs.front, xs.rear);
    swap(xs.n, xs.m);
  }
}
```

With BALANCE algorithm defined, it's trivial to implement remove algorithm both on head and on tail.

```
function REMOVE-HEAD(L)
    BALANCE(L)
    F ← FRONT(L)
    if F = Φ then
        REMOVE-TAIL(L)
    else
        SIZE(F) ← SIZE(F) - 1

function REMOVE-TAIL(L)
    BALANCE(L)
    R ← REAR(L)
    if R = Φ then
        REMOVE-HEAD(L)
    else
        SIZE(R) ← SIZE(R) - 1
```

There is an edge case for each, that is even after balancing, the array targeted to perform removal is still empty. This happens that there is only one element stored in the paired-array list. The solution is just remove this singleton left element, and the overall list results empty. Below is the ISO C++ program implements this algorithm.

```
template<typename Key>
void remove_head(List<Key>& xs) {
  balance(xs);
  if(xs.front.empty())
    remove_tail(xs); //remove the singleton elem in rear
  else {
    xs.front.pop_back();
    --xs.n;
  }
}

template<typename Key>
void remove_tail(List<Key>& xs) {
  balance(xs);
  if(xs.rear.empty())
    remove_head(xs); //remove the singleton elem in front
  else {
    xs.rear.pop_back();
    --xs.m;
  }
}
```

It's obvious that the worst case performance is $O(N)$ where N is the number of elements stored in paired-array list. This happens when balancing is triggered, and both reverse and shifting are linear operation. However, the amortized performance of removal is still $O(1)$, the proof is left as exercise to the reader.

Exercise 12.3

1. Implement the random mutating algorithm in your favorite imperative programming language.

2. We utilized vector provided in standard library to manage memory dynamically, try to realize a version using plain array and manage the memory allocation manually. Compare this version and consider how does this affect the performance?

3. Prove that the amortized performance of removal is $O(1)$ for paired-array list.

12.5 Concatenate-able list

By using binary random access list, we realized sequence data structure which supports $O(\lg N)$ time insertion and removal on head, as well as random accessing element with a given index.

However, it's not so easy to concatenate two lists. As both lists are forests of complete binary trees, we can't merely merge them (Since forests are essentially list of trees, and for any size, there is at most one tree of that size. Even concatenate forests directly is not fast). One solution is to push the element from the first sequence one by one to a stack and then pop those elements and insert them to the head of the second one by using 'cons' function. Of course the stack can be implicitly used in recursion manner, for instance:

$$concat(s_1, s_2) = \begin{cases} s_2 & : & s_1 = \Phi \\ cons(head(s_1), concat(tail(s_1), s_2)) & : & otherwise \end{cases} \tag{12.16}$$

Where function $cons()$, $head()$ and $tail()$ are defined in previous section.

If the length of the two sequence is N, and M, this method takes $O(N \lg N)$ time repeatedly push all elements from the first sequence to stacks, and then takes $\Omega(N \lg(N + M))$ to insert the elements in front of the second sequence. Note that Ω means the upper limit, There is detailed definition for it in [3].

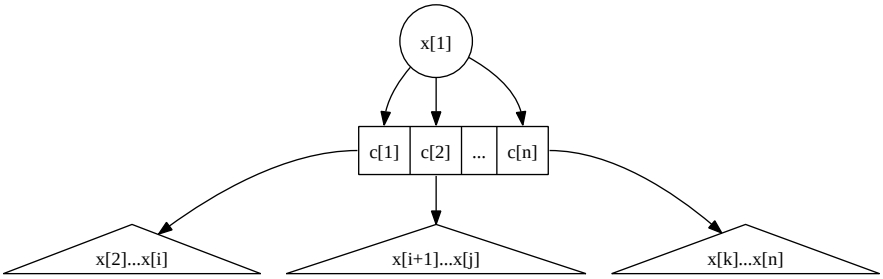
We have already implemented the real-time queue in previous chapter. It supports $O(1)$ time pop and push. If we can turn the sequence concatenation to a kind of pushing operation to queue, the performance will be improved to $O(1)$ as well. Okasaki gave such realization in [6], which can concatenate lists in constant time.

To represent a concatenate-able list, the data structure designed by Okasaki is essentially a K-ary tree. The root of the tree stores the first element in the list. So that we can access it in constant $O(1)$ time. The sub-trees or children are all small concatenate-able lists, which are managed by real-time queues. Concatenating another list to the end is just adding it as the last child, which is in turn a queue pushing operation. Appending a new element can be realized as that, first wrapping the element to a singleton tree, which is a leaf with no children. Then, concatenate this singleton to finalize appending.

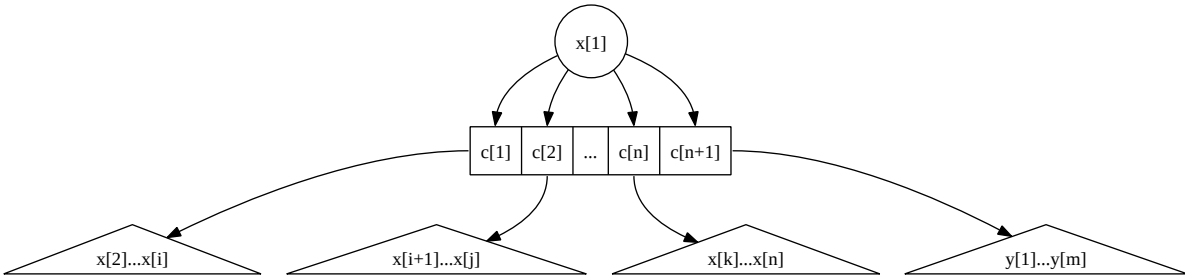
Figure 12.7 illustrates this data structure.

Such recursively designed data structure can be defined in the following Haskell code.

```
data CList a = Empty | CList a (Queue (CList a)) deriving (Show, Eq)
```



(a) The data structure for list $\{x_1, x_2, ..., x_n\}$



(b) The result after concatenated with list $\{y_1, y_2, ..., y_m\}$

Figure 12.7: Data structure for concatenate-able list

It means that a concatenate-able list is either empty or a K-ary tree, which again consists of a queue of concatenate-able sub-lists and a root element. Here we reuse the realization of real-time queue mentioned in previous chapter.

Suppose function $clist(x, Q)$ constructs a concatenate-able list from an element x, and a queue of sub-lists Q. While function $root(s)$ returns the root element of such K-ary tree implemented list. and function $queue(s)$ returns the queue of sub-lists respectively. We can implement the algorithm to concatenate two lists like this.

$$concat(s_1, s_2) = \begin{cases} s_1 & : & s_2 = \Phi \\ s_2 & : & s_1 = \Phi \\ clist(x, push(Q, s_2)) & : & otherwise \end{cases} \tag{12.17}$$

Where $x = root(s_1)$ and $Q = queue(s_1)$. The idea of concatenation is that if either one of the list to be concatenated is empty, the result is just the other list; otherwise, we push the second list as the last child to the queue of the first list.

Since the push operation is $O(1)$ constant time for a well realized real-time queue, the performance of concatenation is bound to $O(1)$.

The *concat*() function can be translated to the below Haskell program.

```
concat x Empty = x
concat Empty y = y
concat (CList x q) y = CList x (push q y)
```

Besides the good performance of concatenation, this design also brings satisfied features for adding element both on head and tail.

$$cons(x, s) = concat(clist(x, \Phi), s) \tag{12.18}$$

$$append(s, x) = concat(s, clist(x, \Phi)) \tag{12.19}$$

It's a bit complex to realize the algorithm that removes the first element from a concatenate-able list. This is because after the root, which is the first element in the sequence got removed, we have to re-construct the rest things, a queue of sub-lists, to a K-ary tree.

Before diving into the re-construction, let's solve the trivial part first. Getting the first element is just returning the root of the K-ary tree.

$$head(s) = root(s) \tag{12.20}$$

As we mentioned above, after root being removed, there left all children of the K-ary tree. Note that all of them are also concatenate-able list, so that one natural solution is to concatenate them all together to a big list.

$$concatAll(Q) = \begin{cases} \Phi & : \quad Q = \Phi \\ concat(front(Q), concatAll(pop(Q))) & : \quad otherwise \end{cases} \tag{12.21}$$

Where function $front()$ just returns the first element from a queue without removing it, while $pop()$ does the removing work.

If the queue is empty, it means that there is no children at all, so the result is also an empty list; Otherwise, we pop the first child, which is a concatenate-able list, from the queue, and recursively concatenate all the rest children to a list; finally, we concatenate this list behind the already popped first children.

With $concatAll()$ defined, we can then implement the algorithm of removing the first element from a list as below.

$$tail(s) = linkAll(queue(s)) \tag{12.22}$$

The corresponding Haskell program is given like the following.

```
head (CList x _) = x
tail (CList _ q) = linkAll q

linkAll q | isEmptyQ q = Empty
          | otherwise = link (front q) (linkAll (pop q))
```

Function `isEmptyQ` is used to test a queue is empty, it is trivial and we omit its definition. Readers can refer to the source code along with this book.

$linkAll()$ algorithm actually traverses the queue data structure, and reduces to a final result. This remind us of *folding* mentioned in the chapter of binary search tree. readers can refer to the appendix of this book for the detailed

description of folding. It's quite possible to define a folding algorithm for queue instead of list[2] [8].

$$foldQ(f, e, Q) = \begin{cases} e & : \quad Q = \Phi \\ f(front(Q), foldQ(f, e, pop(Q))) & : \quad otherwise \end{cases} \tag{12.23}$$

Function $foldQ()$ takes three parameters, a function f, which is used for reducing, an initial value e, and the queue Q to be traversed.

Here are some examples to illustrate folding on queue. Suppose a queue Q contains elements $\{1, 2, 3, 4, 5\}$ from head to tail.

$$\begin{aligned} foldQ(+, 0, Q) &= 1 + (2 + (3 + (4 + (5 + 0)))) = 15 \\ foldQ(\times, 1, Q) &= 1 \times (2 \times (3 \times (4 \times (5 \times 1)))) = 120 \\ foldQ(\times, 0, Q) &= 1 \times (2 \times (3 \times (4 \times (5 \times 0)))) = 0 \end{aligned}$$

Function $linkAll$ can be changed by using $foldQ$ accordingly.

$$linkAll(Q) = foldQ(link, \Phi, Q) \tag{12.24}$$

The Haskell program can be modified as well.

```
linkAll = foldQ link Empty

foldQ :: (a → b → b) → b → Queue a → b
foldQ f z q | isEmptyQ q = z
            | otherwise = (front q) `f` foldQ f z (pop q)
```

However, the performance of removing can't be ensured in all cases. The worst case is that, user keeps appending N elements to a empty list, and then immediately performs removing. At this time, the K-ary tree has the first element stored in root. There are $N - 1$ children, all are leaves. So $linkAll()$ algorithm downgrades to $O(N)$ which is linear time.

The average case is amortized $O(1)$, if the add, append, concatenate and removing operations are randomly performed. The proof is left as en exercise to the reader.

Exercise 12.4

1. Can you figure out a solution to append an element to the end of a binary random access list?

2. Prove that the amortized performance of removal operation is $O(1)$. Hint: using the banker's method.

3. Implement the concatenate-able list in your favorite imperative language.

[2]Some functional programming language, such as Haskell, defined type class, which is a concept of monoid so that it's easy to support folding on a customized data structure.

12.6 Finger tree

We haven't been able to meet all the performance targets listed at the beginning of this chapter.

Binary random access list enables to insert, remove element on the head of sequence, and random access elements fast. However, it performs poor when concatenates lists. There is no good way to append element at the end of binary access list.

Concatenate-able list is capable to concatenates multiple lists in a fly, and it performs well for adding new element both on head and tail. However, it doesn't support randomly access element with a given index.

These two examples bring us some ideas:

- In order to support fast manipulation both on head and tail of the sequence, there must be some way to easily access the head and tail position;
- Tree like data structure helps to turn the random access into divide and conquer search, if the tree is well balance, the search can be ensured to be logarithm time.

12.6.1 Definition

Finger tree[6], which was first invented in 1977, can help to realize efficient sequence. And it is also well implemented in purely functional settings[5].

As we mentioned that the balance of the tree is critical to ensure the performance for search. One option is to use balanced tree as the under ground data structure for finger tree. For example the 2-3 tree, which is a special B-tree. (readers can refer to the chapter of B-tree of this book).

A 2-3 tree either contains 2 children or 3. It can be defined as below in Haskell.

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

In imperative settings, node can be defined with a list of sub nodes, which contains at most 3 children. For instance the following ANSI C code defines node.

```
union Node {
  Key* keys;
  union Node* children;
};
```

Note in this definition, a node can either contain $2 \sim 3$ keys, or $2 \sim 3$ sub nodes. Where key is the type of elements stored in leaf node.

We mark the left-most none-leaf node as the front finger and the right-most none-leaf node as the rear finger. Since both fingers are essentially 2-3 trees with all leafs as children, they can be directly represented as list of 2 or 3 leafs. Of course a finger tree can be empty or contain only one element as leaf.

So the definition of a finger tree is specified like this.

- A finger tree is either empty;
- or a singleton leaf;

- or contains three parts: a left finger which is a list contains at most 3 elements; a sub finger tree; and a right finger which is also a list contains at most 3 elements.

Note that this definition is recursive, so it's quite possible to be translated to functional settings. The following Haskell definition summaries these cases for example.

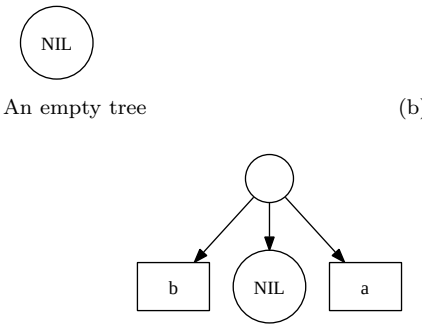
```
data Tree a = Empty
            | Lf a
            | Tr [a] (Tree (Node a)) [a]
```

In imperative settings, we can define the finger tree in a similar manner. What's more, we can add a parent field, so that it's possible to back-track to root from any tree node. Below ANSI C code defines finger tree accordingly.

```
struct Tree {
  union Node* front;
  union Node* rear;
  Tree* mid;
  Tree* parent;
};
```

We can use NIL pointer to represent an empty tree; and a leaf tree contains only one element in its front finger, both its rear finger and middle part are empty;

Figure 12.8 and 12.9 show some examples of figure tree.



(a) An empty tree

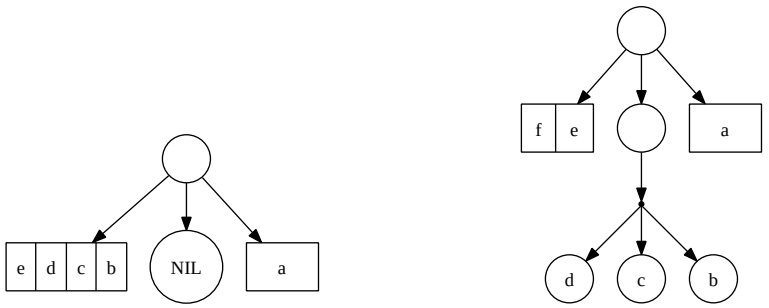
(b) A singleton leaf

(c) Front finger and rear finger contain one element for each, the middle part is empty

Figure 12.8: Examples of finger tree, 1

The first example is an empty finger tree; the second one shows the result after inserting one element to empty, it becomes a leaf of one node; the third example shows a finger tree contains 2 elements, one is in front finger, the other is in rear;

If we continuously insert new elements, to the tree, those elements will be put in the front finger one by one, until it exceeds the limit of 2-3 tree. The 4-th example shows such condition, that there are 4 elements in front finger, which isn't balanced any more.



(a) After inserting extra 3 elements to front finger, it exceeds the 2-3 tree constraint, which isn't balanced any more

(b) The tree resumes balancing. There are 2 elements in front finger; The middle part is a leaf, which contains a 3-branches 2-3 tree.

Figure 12.9: Examples of finger tree, 2

The last example shows that the finger tree gets fixed so that it resumes balancing. There are two elements in the front finger. Note that the middle part is not empty any longer. It's a leaf of a 2-3 tree. The content of the leaf is a tree with 3 branches, each contains an element.

We can express these 5 examples as the following Haskell expression.

```
Empty
Lf a
[b] Empty [a]
[e, d, c, b] Empty [a]
[f, e] Lf (Br3 d c b) [a]
```

As we mentioned that the definition of finger tree is recursive. The middle part besides the front and rear finger is a deeper finger tree, which is defined as $Tree(Node(a))$. Every time we go deeper, the $Node()$ is embedded one more level. if the element type of the first level tree is a, the element type for the second level tree is $Node(a)$, the third level is $Node(Node(a))$, ..., the n-th level is $Node(Node(Node(...(a))...)) = Node^n(a)$, where ^{n} indicates the $Node()$ is applied n times.

12.6.2 Insert element to the head of sequence

The examples list above actually reveal the typical process that the elements are inserted one by one to a finger tree. It's possible to summarize these examples to some cases for insertion on head algorithm.

When we insert an element x to a finger tree T,

- If the tree is empty, the result is a leaf which contains the singleton element x;
- If the tree is a singleton leaf of element y, the result is a new finger tree. The front finger contains the new element x, the rear finger contains the previous element y; the middle part is a empty finger tree;

- If the number of elements stored in front finger isn't bigger than the upper limit of 2-3 tree, which is 3, the new element is just inserted to the head of front finger;
- otherwise, it means that the number of elements stored in front finger exceeds the upper limit of 2-3 tree. the last 3 elements in front finger is wrapped in a 2-3 tree and recursively inserted to the middle part. the new element x is inserted in front of the rest elements in front finger.

Suppose that function $leaf(x)$ creates a leaf of element x, function $tree(F, T', R)$ creates a finger tree from three part: F is the front finger, which is a list contains several elements. Similarity, R is the rear finger, which is also a list. T' is the middle part which is a deeper finger tree. Function $tr3(a, b, c)$ creates a 2-3 tree from 3 elements a, b, c; while $tr2(a, b)$ creates a 2-3 tree from 2 elements a and b.

$$insertT(x, T) = \begin{cases} leaf(x) & : & T = \Phi \\ tree(\{x\}, \Phi, \{y\}) & : & T = leaf(y) \\ tree(\{x, x_1\}, insertT(tr3(x_2, x_3, x_4), T'), R) & : & T = tree(\{x_1, x_2, x_3, x_4\}, T', R) \\ tree(\{x\} \cup F, T', R) & : & otherwise \end{cases} \tag{12.25}$$

The performance of this algorithm is dominated by the recursive case. All the other cases are constant $O(1)$ time. The recursion depth is proportion to the height of the tree, so the algorithm is bound to $O(h)$ time, where h is the height. As we use 2-3 tree to ensure that the tree is well balanced, $h = O(\lg N)$, where N is the number of elements stored in the finger tree.

More analysis reveal that the amortized performance of $insertT()$ is $O(1)$ because we can amortize the expensive recursion case to other trivial cases. Please refer to [6] and [5] for the detailed proof.

Translating the algorithm yields the below Haskell program.

```
cons :: a → Tree a → Tree a
cons a Empty = Lf a
cons a (Lf b) = Tr [a] Empty [b]
cons a (Tr [b, c, d, e] m r) = Tr [a, b] (cons (Br3 c d e) m) r
cons a (Tr f m r) = Tr (a:f) m r
```

Here we use the LISP naming convention to illustrate inserting a new element to a list.

The insertion algorithm can also be implemented in imperative approach. Suppose function TREE() creates an empty tree, that all fields, including front and rear finger, the middle part inner tree and parent are empty. Function NODE() creates an empty node.

```
function PREPEND-NODE(n, T)
    r ← TREE()
    p ← r
    CONNECT-MID(p, T)
    while FULL?(FRONT(T)) do
        F ← FRONT(T)                          ▷ F = {n_1, n_2, n_3, ...}
        FRONT(T) ← {n, F[1]}                  ▷ F[1] = n_1
        n ← NODE()
```

```
    CHILDREN(n) ← F[2..]                          ▷ F[2..] = {n_2, n_3, ...}
    p ← T
    T ← MID(T)
  if T = NIL then
    T ← TREE()
    FRONT(T)← {n}
  else if | FRONT(T) | = 1 ∧ REAR(T) = Φ then
    REAR(T) ← FRONT(T)
    FRONT(T) ← {n}
  else
    FRONT(T) ← {n}∪ FRONT(T)
  CONNECT-MID(p, T) ← T
  return FLAT(r)
```

Where the notation $L[i..]$ means a sub list of L with the first $i-1$ elements removed, that if $L = \{a_1, a_2, ..., a_n\}$, $L[i..] = \{a_i, a_{i+1}, ..., a_n\}$.

Functions FRONT(), REAR(), MID(), and PARENT() are used to access the front finger, the rear finger, the middle part inner tree and the parent tree respectively; Function CHILDREN() accesses the children of a node.

Function CONNECT-MID(T_1, T_2), connect T_2 as the inner middle part tree of T_1, and set the parent of T_2 as T_1 if T_2 isn't empty.

In this algorithm, we performs a one pass top-down traverse along the middle part inner tree if the front finger is full that it can't afford to store any more. The criteria for full for a 2-3 tree is that the finger contains 3 elements already. In such case, we extract all the elements except the first one off, wrap them to a new node (one level deeper node), and continuously insert this new node to its middle inner tree. The first element is left in the front finger, and the element to be inserted is put in front of it, so that this element becomes the new first one in the front finger.

After this traversal, the algorithm either reach an empty tree, or the tree still has room to hold more element in its front finger. We create a new leaf for the former case, and perform a trivial list insert to the front finger for the latter.

During the traversal, we use p to record the parent of the current tree we are processing. So any new created tree are connected as the middle part inner tree to p.

Finally, we return the root of the tree r. The last trick of this algorithm is the FLAT() function. In order to simplify the logic, we create an empty 'ground' tree and set it as the parent of the root. We need eliminate this extra 'ground' level before return the root. This flatten algorithm is realized as the following.

```
function FLAT(T)
  while T ≠ NIL ∧T is empty do
    T ← MID(T)
  if T ≠ Φ then
    PARENT(T) ← Φ
  return T
```

The while loop test if T is trivial empty, that it's not NIL($= \Phi$), while both its front and rear fingers are empty.

Below Python code implements the insertion algorithm for finger tree.

```
def insert(x, t):
    return prepend_node(wrap(x), t)

def prepend_node(n, t):
    root = prev = Tree()
    prev.set_mid(t)
    while frontFull(t):
        f = t.front
        t.front = [n] + f[:1]
        n = wraps(f[1:])
        prev = t
        t = t.mid
    if t is None:
        t = leaf(n)
    elif len(t.front)==1 and t.rear == []:
        t = Tree([n], None, t.front)
    else:
        t = Tree([n]+t.front, t.mid, t.rear)
    prev.set_mid(t)
    return flat(root)

def flat(t):
    while t is not None and t.empty():
        t = t.mid
    if t is not None:
        t.parent = None
    return t
```

The implementation of function '`set_mid()`', '`frontFull()`', '`wrap()`', '`wraps()`', '`empty()`', and tree constructor are trivial enough, that we skip the detail of them here. Readers can take these as exercises.

12.6.3 Remove element from the head of sequence

It's easy to implement the reverse operation that remove the first element from the list by reversing the $insertT()$ algorithm line by line.

Let's denote $F = \{f_1, f_2, ...\}$ is the front finger list, M is the middle part inner finger tree. $R = \{r_1, r_2, ...\}$ is the rear finger list of a finger tree, and $R' = \{r_2, r_3, ...\}$ is the rest of element with the first one removed from R.

$$extractT(T) = \begin{cases} (x, \Phi) & : & T = leaf(x) \\ (x, leaf(y)) & : & T = tree(\{x\}, \Phi, \{y\}) \\ (x, tree(\{r_1\}, \Phi, R')) & : & T = tree(\{x\}, \Phi, R) \\ (x, tree(toList(F'), M', R)) & : & T = tree(\{x\}, M, R), (F', M') = extractT(M) \\ (f_1, tree(\{f_2, f_3, ...\}, M, R)) & : & otherwise \end{cases} \tag{12.26}$$

Where function $toList(T)$ converts a 2-3 tree to plain list as the following.

$$toList(T) = \begin{cases} \{x, y\} & : & T = tr2(x, y) \\ \{x, y, z\} & : & T = tr3(x, y, z) \end{cases} \tag{12.27}$$

Here we skip the error handling such as trying to remove element from empty tree etc. If the finger tree is a leaf, the result after removal is an empty tree; If

the finger tree contains two elements, one in the front rear, the other in rear, we return the element stored in front rear as the first element, and the resulted tree after removal is a leaf; If there is only one element in front finger, the middle part inner tree is empty, and the rear finger isn't empty, we return the only element in front finger, and borrow one element from the rear finger to front; If there is only one element in front finger, however, the middle part inner tree isn't empty, we can recursively remove a node from the inner tree, and flatten it to a plain list to replace the front finger, and remove the original only element in front finger; The last case says that if the front finger contains more than one element, we can just remove the first element from front finger and keep all the other part unchanged.

Figure 12.10 shows the steps of removing two elements from the head of a sequence. There are 10 elements stored in the finger tree. When the first element is removed, there is still one element left in the front finger. However, when the next element is removed, the front finger is empty. So we 'borrow' one tree node from the middle part inner tree. This is a 2-3 tree. it is converted to a list of 3 elements, and the list is used as the new finger. the middle part inner tree change from three parts to a singleton leaf, which contains only one 2-3 tree node. There are three elements stored in this tree node.

Below is the corresponding Haskell program for 'uncons'.

```
uncons :: Tree a → (a, Tree a)
uncons (Lf a) = (a, Empty)
uncons (Tr [a] Empty [b]) = (a, Lf b)
uncons (Tr [a] Empty (r:rs)) = (a, Tr [r] Empty rs)
uncons (Tr [a] m r) = (a, Tr (nodeToList f) m' r) where (f, m') = uncons m
uncons (Tr f m r) = (head f, Tr (tail f) m r)
```

And the function *nodeToList* is defined like this.

```
nodeToList :: Node a → [a]
nodeToList (Br2 a b) = [a, b]
nodeToList (Br3 a b c) = [a, b, c]
```

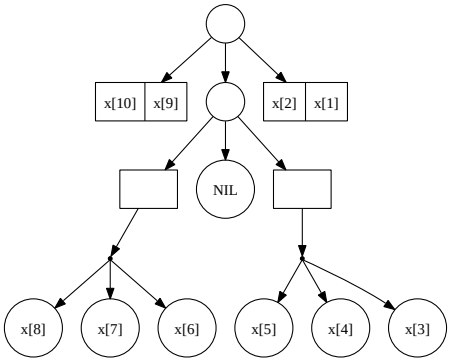
Similar as above, we can define *head* and *tail* function from *uncons*.

```
head = fst ∘ uncons
tail = snd ∘ uncons
```

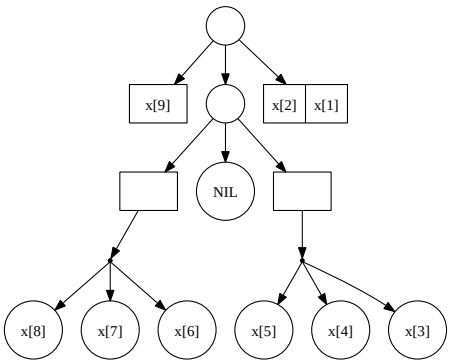
12.6.4 Handling the ill-formed finger tree when removing

The strategy used so far to remove element from finger tree is a kind of removing and borrowing. If the front finger becomes empty after removing, we borrows more nodes from the middle part inner tree. However there exists cases that the tree is ill-formed, for example, both the front fingers of the tree and its middle part inner tree are empty. Such ill-formed tree can result from imperatively splitting, which we'll introduce later.

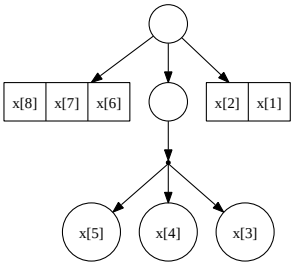
Here we developed an imperative algorithm which can remove the first element from finger tree even it is ill-formed. The idea is first perform a top-down traverse to find a sub tree which either has a non-empty front finger or both its front finger and middle part inner tree are empty. For the former case, we can safely extract the first element which is a node from the front finger; For



(a) A sequence of 10 elements represented as a finger tree



(b) The first element is removed. There is one element left in front finger.



(c) Another element is remove from head. We borrowed one node from the middle part inner tree, change the node, which is a 2-3 tree to a list, and use it as the new front finger. the middle part inner tree becomes a leaf of one 2-3 tree node.

Figure 12.10: Examples of remove 2 elements from the head of a sequence

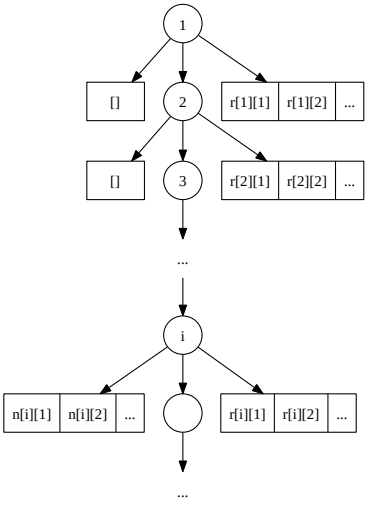


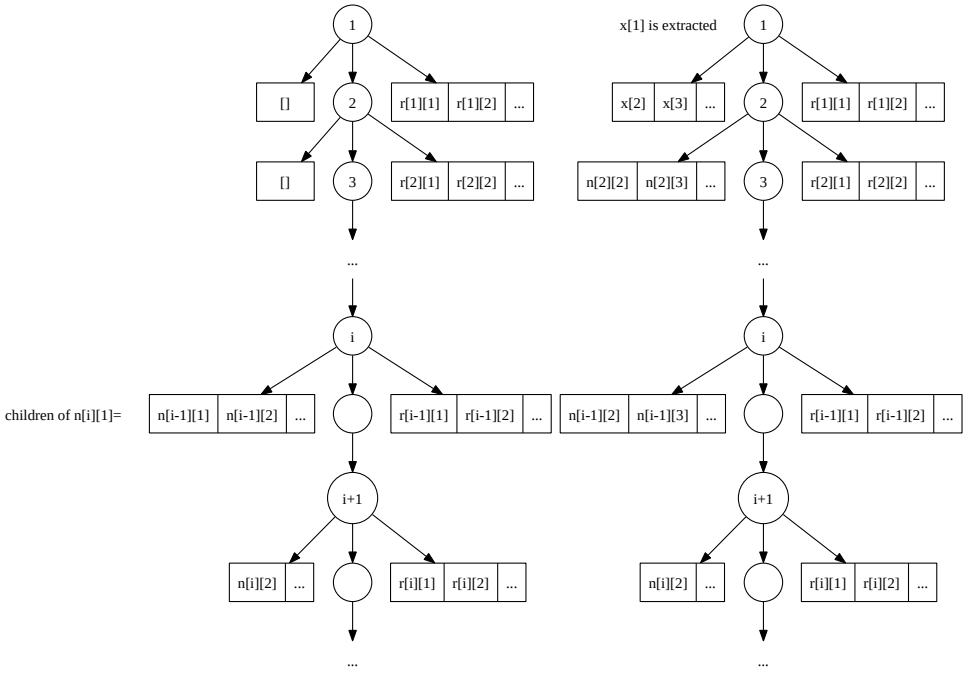
Figure 12.11: Example of an ill-formed tree. The front finger of the i-th level sub tree isn't empty.

the latter case, since only the rear finger isn't empty, we can swap it with the empty front finger, and change it to the former case.

After that, we need examine the node we extracted from the front finger is leaf node (How to do that? this is left as an exercise to the reader). If not, we need go on extracting the first sub node from the children of this node, and left the rest of other children as the new front finger to the parent of the current tree. We need repeatedly go up along with the parent field till the node we extracted is a leaf. At that time point, we arrive at the root of the tree. Figure 12.12 illustrates this process.

Based on this idea, the following algorithm realizes the removal operation on head. The algorithm assumes that the tree passed in isn't empty.

function EXTRACT-HEAD(T)
  $r \leftarrow$ TREE()
  CONNECT-MID(r, T)
  while FRONT(T) $= \Phi \wedge$ MID(T) $\neq$ NIL **do**
    $T \leftarrow$ MID(T)
  if FRONT(T) $= \Phi \wedge$ REAR(T) $\neq \Phi$ **then**
    EXCHANGE FRONT(T) $\leftrightarrow$ REAR(T)
  $n \leftarrow$ NODE()
  CHILDREN(n) $\leftarrow$ FRONT(T)
  repeat
    $L \leftarrow$ CHILDREN(n) ▷ $L = \{n_1, n_2, n_3, ...\}$
    $n \leftarrow L[1]$ ▷ $n \leftarrow n_1$
    FRONT(T) $\leftarrow L[2..]$ ▷ $L[2..] = \{n_2, n_3, ...\}$
    $T \leftarrow$ PARENT(T)
    if MID(T) becomes empty **then**
      MID(T) $\leftarrow$ NIL
  until n is a leaf



(a) Extract the first element n[i][1] and put its children to the front finger of upper level tree.

(b) Repeat this process i times, and finally x[1] is extracted.

Figure 12.12: Traverse bottom-up till a leaf is extracted.

return (ELEM(n), FLAT(r))

Note that function ELEM(n) returns the only element stored inside leaf node n. Similar as imperative insertion algorithm, a stub 'ground' tree is used as the parent of the root, which can simplify the logic a bit. That's why we need flatten the tree finally.

Below Python program translates the algorithm.

```
def extract_head(t):
    root = Tree()
    root.set_mid(t)
    while t.front == [] and t.mid is not None:
        t = t.mid
    if t.front == [] and t.rear != []:
        (t.front, t.rear) = (t.rear, t.front)
    n = wraps(t.front)
    while True: # a repeat-until loop
        ns = n.children
        n = ns[0]
        t.front = ns[1:]
        t = t.parent
        if t.mid.empty():
            t.mid.parent = None
            t.mid = None
        if n.leaf:
            break
    return (elem(n), flat(root))
```

Member function `Tree.empty()` returns true if all the three parts - the front finger, the rear finger and the middle part inner tree - are empty. We put a flag `Node.leaf` to mark if a node is a leaf or compound node. The exercise of this section asks the reader to consider some alternatives.

As the ill-formed tree is allowed, the algorithms to access the first and last element of the finger tree must be modified, so that they don't blindly return the first or last child of the finger as the finger can be empty if the tree is ill-formed.

The idea is quite similar to the EXTRACT-HEAD, that in case the finger is empty while the middle part inner tree isn't, we need traverse along with the inner tree till a point that either the finger becomes non-empty or all the nodes are stored in the other finger. For instance, the following algorithm can return the first leaf node even the tree is ill-formed.

function FIRST-LF(T)
  while FRONT(T) $= \Phi \wedge$ MID(T) $\neq$ NIL **do**
    $T \leftarrow$ MID(T)
  if FRONT(T) $= \Phi \wedge$ REAR(T) $\neq \Phi$ **then**
    $n \leftarrow$ REAR(T)[1]
  else
    $n \leftarrow$ FRONT(T)[1]
  while n is NOT leaf **do**
    $n \leftarrow$ CHILDREN(n)[1]
  return n

Note the second loop in this algorithm that it keeps traversing on the first sub-node if current node isn't a leaf. So we always get a leaf node and it's trivial

to get the element inside it.

function FIRST(T)
  return ELEM(FIRST-LF(T))

The following Python code translates the algorithm to real program.

```
def first(t):
    return elem(first_leaf(t))

def first_leaf(t):
    while t.front == [] and t.mid is not None:
        t = t.mid
    if t.front == [] and t.rear != []:
        n = t.rear[0]
    else:
        n = t.front[0]
    while not n.leaf:
        n = n.children[0]
    return n
```

To access the last element is quite similar, and we left it as an exercise to the reader.

12.6.5 append element to the tail of the sequence

Because finger tree is symmetric, we can give the realization of appending element on tail by referencing to $insertT()$ algorithm.

$$appendT(T, x) = \begin{cases} leaf(x) & : & T = \Phi \\ tree(\{y\}, \Phi, \{x\}) & : & T = leaf(y) \\ tree(F, appendT(M, tr3(x_1, x_2, x_3)), \{x_4, x\}) & : & T = tree(F, M, \{x_1, x_2, x_3, x_4\}) \\ tree(F, M, R \cup \{x\}) & : & otherwise \end{cases} \tag{12.28}$$

Generally speaking, if the rear finger is still valid 2-3 tree, that the number of elements is not greater than 4, the new elements is directly appended to rear finger. Otherwise, we break the rear finger, take the first 3 elements in rear finger to create a new 2-3 tree, and recursively append it to the middle part inner tree. If the finger tree is empty or a singleton leaf, it will be handled in the first two cases.

Translating the equation to Haskell yields the below program.

```
snoc :: Tree a → a → Tree a
snoc Empty a = Lf a
snoc (Lf a) b = Tr [a] Empty [b]
snoc (Tr f m [a, b, c, d]) e = Tr f (snoc m (Br3 a b c)) [d, e]
snoc (Tr f m r) a = Tr f m (r++[a])
```

Function name '`snoc`' is mirror of '`cons`', which indicates the symmetric relationship.

Appending new element to the end imperatively is quite similar. The following algorithm realizes appending.

function APPEND-NODE(T, n)
  $r \leftarrow$ TREE()

$p \leftarrow r$
CONNECT-MID(p, T)
while FULL?(REAR(T)) **do**
  $R \leftarrow$ REAR(T) ▷ $R = \{n_1, n_2, ..., , n_{m-1}, n_m\}$
  REAR$(T) \leftarrow \{n,$ LAST$(R)\ \}$ ▷ last element n_m
  $n \leftarrow$ NODE()
  CHILDREN$(n) \leftarrow R[1...m-1]$ ▷ $\{n1, n2, ..., n_{m-1}\}$
  $p \leftarrow T$
  $T \leftarrow$ MID(T)
if $T =$ NIL **then**
  $T \leftarrow$ TREE()
  FRONT$(T) \leftarrow \{n\}$
else if $\mid$ REAR$(T) \mid = 1 \wedge$ FRONT$(T) = \Phi$ **then**
  FRONT$(T) \leftarrow$ REAR(T)
  REAR$(T) \leftarrow \{n\}$
else
  REAR$(T) \leftarrow$ REAR$(T) \cup \{n\}$
CONNECT-MID$(p, T) \leftarrow T$
return FLAT(r)

And the corresponding Python programs is given as below.

```
def append_node(t, n):
    root = prev = Tree()
    prev.set_mid(t)
    while rearFull(t):
        r = t.rear
        t.rear = r[-1:] + [n]
        n = wraps(r[:-1])
        prev = t
        t = t.mid
    if t is None:
        t = leaf(n)
    elif len(t.rear) == 1 and t.front == []:
        t = Tree(t.rear, None, [n])
    else:
        t = Tree(t.front, t.mid, t.rear + [n])
    prev.set_mid(t)
    return flat(root)
```

12.6.6 remove element from the tail of the sequence

Similar to *appendT*(), we can realize the algorithm which remove the last element from finger tree in symmetric manner of *extractT*().

We denote the non-empty, non-leaf finger tree as $tree(F, M, R)$, where F is

the front finger, M is the middle part inner tree, and R is the rear finger.

$$removeT(T) = \begin{cases} (\Phi, x) & : & T = leaf(x) \\ (leaf(y), x) & : & T = tree(\{y\}, \Phi, \{x\}) \\ (tree(init(F), \Phi, last(F)), x) & : & T = tree(F, \Phi, \{x\}) \wedge F \neq \Phi \\ (tree(F, M', toList(R')), x) & : & T = tree(F, M, \{x\}), (M', R') = removeT(M) \\ (tree(F, M, init(R)), last(R)) & : & otherwise \end{cases} \tag{12.29}$$

Function $toList(T)$ is used to flatten a 2-3 tree to plain list, which is defined previously. Function $init(L)$ returns all elements except for the last one in list L, that if $L = \{a_1, a_2, ..., a_{n-1}, a_n\}$, $init(L) = \{a_1, a_2, ..., a_{n-1}\}$. And Function $last(L)$ returns the last element, so that $last(L) = a_n$. Please refer to the appendix of this book for their implementation.

Algorithm $removeT()$ can be translated to the following Haskell program, we name it as '`unsnoc`' to indicate it's the reverse function of '`snoc`'.

```
unsnoc :: Tree a → (Tree a, a)
unsnoc (Lf a) = (Empty, a)
unsnoc (Tr [a] Empty [b]) = (Lf a, b)
unsnoc (Tr f@(_:_) Empty [a]) = (Tr (init f) Empty [last f], a)
unsnoc (Tr f m [a]) = (Tr f m' (nodeToList r), a) where (m', r) = unsnoc m
unsnoc (Tr f m r) = (Tr f m (init r), last r)
```

And we can define a special function '`last`' and '`init`' for finger tree which is similar to their counterpart for list.

```
last = snd ∘ unsnoc
init = fst ∘ unsnoc
```

Imperatively removing the element from the end is almost as same as removing on the head. Although there seems to be a special case, that as we always store the only element (or sub node) in the front finger while the rear finger and middle part inner tree are empty (e.g. $Tree(\{n\}, NIL, \Phi)$), it might get nothing if always try to fetch the last element from rear finger.

This can be solved by swapping the front and the rear finger if the rear is empty as in the following algorithm.

function EXTRACT-TAIL(T)
  $r \leftarrow$ TREE()
  CONNECT-MID(r, T)
  while REAR(T) $= \Phi \wedge$ MID(T) $\neq$ NIL **do**
    $T \leftarrow$ MID(T)
  if REAR(T) $= \Phi \wedge$ FRONT(T) $\neq \Phi$ **then**
    EXCHANGE FRONT(T) $\leftrightarrow$ REAR(T)
  $n \leftarrow$ NODE()
  CHILDREN(n) $\leftarrow$ REAR(T)
  repeat
    $L \leftarrow$ CHILDREN(n) ▷ $L = \{n_1, n_2, ..., n_{m-1}, n_m\}$
    $n \leftarrow$ LAST(L) ▷ $n \leftarrow n_m$
    REAR(T) $\leftarrow L[1...m-1]$ ▷ $\{n_1, n_2, ..., n_{m-1}\}$
    $T \leftarrow$ PARENT(T)
    if MID(T) becomes empty **then**

MID(T) $\leftarrow$ NIL
until n is a leaf
return (ELEM(n), FLAT(r))

How to access the last element as well as implement this algorithm to working program are left as exercises.

12.6.7 concatenate

Consider the none-trivial case that concatenate two finger trees $T_1 = tree(F_1, M_1, R_1)$ and $T_2 = tree(F_2, M_2, R_2)$. One natural idea is to use F_1 as the new front finger for the concatenated result, and keep R_2 being the new rear finger. The rest of work is to merge M_1, R_1, F_2 and M_2 to a new middle part inner tree.

Note that both R_1 and F_2 are plain lists of node, so the sub-problem is to realize a algorithm like this.

$$merge(M_1, R_1 \cup F_2, M_2) =?$$

More observation reveals that both M_1 and M_2 are also finger trees, except that they are one level deeper than T_1 and T_2 in terms of $Node(a)$, where a is the type of element stored in the tree. We can recursively use the strategy that keep the front finger of M_1 and the rear finger of M_2, then merge the middle part inner tree of M_1, M_2, as well as the rear finger of M_1 and front finger of M_2.

If we denote function $front(T)$ returns the front finger, $rear(T)$ returns the rear finger, $mid(T)$ returns the middle part inner tree. the above $merge()$ algorithm can be expressed for non-trivial case as the following.

$$\begin{aligned} &merge(M_1, R_1 \cup F_2, M_2) = tree(front(M_1), S, rear(M_2)) \\ &S = merge(mid(M_1), rear(M_1) \cup R_1 \cup F_2 \cup front(M_2), mid(M_2)) \end{aligned} \tag{12.30}$$

If we look back to the original concatenate solution, it can be expressed as below.

$$concat(T_1, T_2) = tree(F_1, merge(M_1, R_1 \cup R_2, M_2), R_2) \tag{12.31}$$

And compare it with equation 12.30, it's easy to note the fact that concatenating is essentially merging. So we have the final algorithm like this.

$$concat(T_1, T_2) = merge(T_1, \Phi, T_2) \tag{12.32}$$

By adding edge cases, the $merge()$ algorithm can be completed as below.

$$merge(T_1, S, T_2) = \begin{cases} foldR(insertT, T_2, S) & : & T_1 = \Phi \\ foldL(appendT, T_1, S) & : & T_2 = \Phi \\ merge(\Phi, \{x\} \cup S, T_2) & : & T_1 = leaf(x) \\ merge(T_1, S \cup \{x\}, \Phi) & : & T_2 = leaf(x) \\ tree(F_1, merge(M_1, nodes(R_1 \cup S \cup F_2), M2), R_2) & : & otherwise \end{cases} \tag{12.33}$$

Most of these cases are straightforward. If any one of T_1 or T_2 is empty, the algorithm repeatedly insert/append all elements in S to the other tree;

Function $foldL$ and $foldR$ are kinds of for-each process in imperative settings. The difference is that $foldL$ processes the list S from left to right while $foldR$ processes from right to left.

Here are their definition. Suppose list $L = \{a_1, a_2, ..., a_{n-1}, a_n\}$, $L' = \{a_2, a_3, ..., a_{n-1}, a_n\}$ is the rest of elements except for the first one.

$$foldL(f, e, L) = \begin{cases} e & : & L = \Phi \\ foldL(f, f(e, a_1), L') & : & otherwise \end{cases} \tag{12.34}$$

$$foldR(f, e, L) = \begin{cases} e & : & L = \Phi \\ f(a_1, foldR(f, e, L')) & : & otherwise \end{cases} \tag{12.35}$$

They are detailed explained in the appendix of this book.

If either one of the tree is a leaf, we can insert or append the element of this leaf to S, so that it becomes the trivial case of concatenating one empty tree with another.

Function $nodes()$ is used to wrap a list of elements to a list of 2-3 trees. This is because the contents of middle part inner tree, compare to the contents of finger, are one level deeper in terms of $Node()$. Consider the time point that transforms from recursive case to edge case. Let's suppose M_1 is empty at that time, we then need repeatedly insert all elements from $R_1 \cup S \cup F_2$ to M_2. However, we can't directly do the insertion. If the element type is a, we can only insert $Node(a)$ which is 2-3 tree to M_2. This is just like what we did in the $insertT()$ algorithm, take out the last 3 elements, wrap them in a 2-3 tree, and recursive perform $insertT()$. Here is the definition of $nodes()$.

$$nodes(L) = \begin{cases} \{tr2(x_1, x_2)\} & : & L = \{x_1, x_2\} \\ \{tr3(x_1, x_2, x_3)\} & : & L = \{x_1, x_2, x_3\} \\ \{tr2(x_1, x_2), tr2(x_3, x_4)\} & : & L = \{x_1, x_2, x_3, x_4\} \\ \{tr3(x_1, x_2, x_3)\} \cup nodes(\{x_4, x_5, ...\}) & : & otherwise \end{cases} \tag{12.36}$$

Function $nodes()$ follows the constraint of 2-3 tree, that if there are only 2 or 3 elements in the list, it just wrap them in singleton list contains a 2-3 tree; If there are 4 elements in the lists, it split them into two trees each is consist of 2 branches; Otherwise, if there are more elements than 4, it wraps the first three in to one tree with 3 branches, and recursively call $nodes()$ to process the rest.

The performance of concatenation is determined by merging. Analyze the recursive case of merging reveals that the depth of recursion is proportion to the smaller height of the two trees. As the tree is ensured to be balanced by using 2-3 tree. it's height is bound to $O(\lg N')$ where N' is the number of elements. The edge case of merging performs as same as insertion, (It calls $insertT()$ at most 8 times) which is amortized $O(1)$ time, and $O(\lg M)$ at worst case, where M is the difference in height of the two trees. So the overall performance is bound to $O(\lg N)$, where N is the total number of elements contains in two finger trees.

The following Haskell program implements the concatenation algorithm.

```
concat :: Tree a → Tree a → Tree a
concat t1 t2 = merge t1 [] t2
```

Note that there is '`concat`' function defined in prelude standard library, so we need distinct them either by hiding import or take a different name.

```
merge :: Tree a → [a] → Tree a → Tree a
merge Empty ts t2 = foldr cons t2 ts
merge t1 ts Empty = foldl snoc t1 ts
merge (Lf a) ts t2 = merge Empty (a:ts) t2
merge t1 ts (Lf a) = merge t1 (ts++[a]) Empty
merge (Tr f1 m1 r1) ts (Tr f2 m2 r2) = Tr f1 (merge m1 (nodes (r1 ++ ts ++
f2)) m2) r2
```

And the implementation of *nodes*() is as below.

```
nodes :: [a] → [Node a]
nodes [a, b] = [Br2 a b]
nodes [a, b, c] = [Br3 a b c]
nodes [a, b, c, d] = [Br2 a b, Br2 c d]
nodes (a:b:c:xs) = Br3 a b c:nodes xs
```

To concatenate two finger trees T_1 and T_2 in imperative approach, we can traverse the two trees along with the middle part inner tree till either tree turns to be empty. In every iteration, we create a new tree T, choose the front finger of T_1 as the front finger of T; and choose the rear finger of T_2 as the rear finger of T. The other two fingers (rear finger of T_1 and front finger of T_2) are put together as a list, and this list is then balanced grouped to several 2-3 tree nodes as N. Note that N grows along with traversing not only in terms of length, the depth of its elements increases by one in each iteration. We attach this new tree as the middle part inner tree of the upper level result tree to end this iteration.

Once either tree becomes empty, we stop traversing, and repeatedly insert the 2-3 tree nodes in N to the other non-empty tree, and set it as the new middle part inner tree of the upper level result.

Below algorithm describes this process in detail.

function CONCAT(T_1, T_2)
  return MERGE(T_1, Φ, T_2)

function MERGE(T_1, N, T_2)
  $r \leftarrow$ TREE()
  $p \leftarrow r$
  while $T_1 \neq$ NIL $\wedge T_2 \neq$ NIL **do**
    $T \leftarrow$ TREE()
    FRONT(T) $\leftarrow$ FRONT(T_1)
    REAR(T) $\leftarrow$ REAR(T_2)
    CONNECT-MID(p, T)
    $p \leftarrow T$
    $N \leftarrow$ NODES(REAR(T_1) $\cup N \cup$ FRONT(T_2))
    $T_1 \leftarrow$ MID(T_1)
    $T_2 \leftarrow$ MID(T_2)
  if $T_1 =$ NIL **then**
    $T \leftarrow T_2$
    for each $n \in$ REVERSE(N) **do**
      $T \leftarrow$ PREPEND-NODE(n, T)
  else if $T_2 =$ NIL **then**

$T \leftarrow T_1$
for each $n \in N$ **do**
$T \leftarrow$ APPEND-NODE(T, n)
CONNECT-MID(p, T)
return FLAT(r)

Note that the for-each loops in the algorithm can also be replaced by folding from left and right respectively. Translating this algorithm to Python program yields the below code.

```
def concat(t1, t2):
    return merge(t1, [], t2)

def merge(t1, ns, t2):
    root = prev = Tree() #sentinel dummy tree
    while t1 is not None and t2 is not None:
        t = Tree(t1.size + t2.size + sizeNs(ns), t1.front, None, t2.rear)
        prev.set_mid(t)
        prev = t
        ns = nodes(t1.rear + ns + t2.front)
        t1 = t1.mid
        t2 = t2.mid
    if t1 is None:
        prev.set_mid(foldR(prepend_node, ns, t2))
    elif t2 is None:
        prev.set_mid(reduce(append_node, ns, t1))
    return flat(root)
```

Because Python only provides folding function from left as `reduce()`, a folding function from right is given like what we shown in pseudo code, that it repeatedly applies function in reverse order of the list.

```
def foldR(f, xs, z):
    for x in reversed(xs):
        z = f(x, z)
    return z
```

The only function in question is how to balanced-group nodes to bigger 2-3 trees. As a 2-3 tree can hold at most 3 sub trees, we can firstly take 3 nodes and wrap them to a ternary tree if there are more than 4 nodes in the list and continuously deal with the rest. If there are just 4 nodes, they can be wrapped to two binary trees. For other cases (there are 3 trees, 2 trees, 1 tree), we simply wrap them all to a tree.

Denote node list $L = \{n_1, n_2, ...\}$, The following algorithm realizes this process.

function NODES(L)
$N = \Phi$
while $|L| > 4$ **do**
$n \leftarrow$ NODE()
CHILDREN$(n) \leftarrow L[1..3]$ ▷ $\{n_1, n_2, n_3\}$
$N \leftarrow N \cup \{n\}$
$L \leftarrow L[4...]$ ▷ $\{n_4, n_5, ...\}$
if $|L| = 4$ **then**
$x \leftarrow$ NODE()

```
        CHILDREN(x) ← {L[1], L[2]}
        y ← NODE()
        CHILDREN(y) ← {L[3], L[4]}
        N ← N ∪ {x, y}
    else if L ≠ Φ then
        n ← NODE()
        CHILDREN(n) ← L
        N ← N ∪ {n}
    return N
```

It's straight forward to translate the algorithm to below Python program. Where function `wraps()` helps to create an empty node, then set a list as the children of this node.

```
def nodes(xs):
    res = []
    while len(xs) > 4:
        res.append(wraps(xs[:3]))
        xs = xs[3:]
    if len(xs) == 4:
        res.append(wraps(xs[:2]))
        res.append(wraps(xs[2:]))
    elif xs != []:
        res.append(wraps(xs))
    return res
```

Exercise 12.5

1. Implement the complete finger tree insertion program in your favorite imperative programming language. Don't check the example programs along with this chapter before having a try.

2. How to determine a node is a leaf? Does it contain only a raw element inside or a compound node, which contains sub nodes as children? Note that we can't distinguish it by testing the size, as there is case that node contains a singleton leaf, such as $node(1, \{node(1, \{x\}\})$. Try to solve this problem in both dynamic typed language (e.g. Python, lisp etc) and in strong static typed language (e.g. C++).

3. Implement the EXTRACT-TAIL algorithm in your favorite imperative programming language.

4. Realize algorithm to return the last element of a finger tree in both functional and imperative approach. The later one should be able to handle ill-formed tree.

5. Try to implement concatenation algorithm without using folding. You can either use recursive methods, or use imperative for-each method.

12.6.8 Random access of finger tree

size augmentation

The strategy to provide fast random access, is to turn the looking up into tree-search. In order to avoid calculating the size of tree many times, we augment an extra field to tree and node. The definition should be modified accordingly, for example the following Haskell definition adds size field in its constructor.

```
data Tree a = Empty
            | Lf a
            | Tr Int [a] (Tree (Node a)) [a]
```

And the previous ANSI C structure is augmented with size as well.

```
struct Tree {
  union Node* front;
  union Node* rear;
  Tree* mid;
  Tree* parent;
  int size;
};
```

Suppose the function $tree(s, F, M, R)$ creates a finger tree from size s, front finger F, rear finger R, and middle part inner tree M. When the size of the tree is needed, we can call a $size(T)$ function. It will be something like this.

$$size(T) = \begin{cases} 0 & : & T = \Phi \\ ? & : & T = leaf(x) \\ s & : & T = tree(s, F, M, R) \end{cases}$$

If the tree is empty, the size is definitely zero; and if it can be expressed as $tree(s, F, M, R)$, the size is s; however, what if the tree is a singleton leaf? is it 1? No, it can be 1 only if $T = leaf(a)$ and a isn't a tree node, but a raw element stored in finger tree. In most cases, the size is not 1, because a can be again a tree node. That's why we put a '?' in above equation.

The correct way is to call some size function on the tree node as the following.

$$size(T) = \begin{cases} 0 & : & T = \Phi \\ size'(x) & : & T = leaf(x) \\ s & : & T = tree(s, F, M, R) \end{cases} \tag{12.37}$$

Note that this isn't a recursive definition since $size \neq size'$, the argument to $size'$ is either a tree node, which is a 2-3 tree, or a plain element stored in the finger tree. To uniform these two cases, we can anyway wrap the single plain element to a tree node of only one element. So that we can express all the situation as a tree node augmented with a size field. The following Haskell program modifies the definition of tree node.

```
data Node a = Br Int [a]
```

The ANSI C node definition is modified accordingly.

```
struct Node {
  Key key;
  struct Node* children;
  int size;
};
```

We change it from union to structure. Although there is a overhead field 'key' if the node isn't a leaf.

Suppose function $tr(s, L)$, creates such a node (either one element being wrapped or a 2-3 tree) from a size information s, and a list L. Here are some example.

$tr(1, \{x\})$	a tree contains only one element
$tr(2, \{x, y\})$	a 2-3 tree contains two elements
$tr(3, \{x, y, z\})$	a 2-3 tree contains three elements

So the function $size'$ can be implemented as returning the size information of a tree node. We have $size'(tr(s, L)) = s$.

Wrapping an element x is just calling $tr(1, \{x\})$. We can define auxiliary functions $wrap$ and $unwrap$, for instance.

$$\begin{aligned} wrap(x) &= tr(1, \{x\}) \\ unwrap(n) &= x \quad : \quad n = tr(1, \{x\}) \end{aligned} \tag{12.38}$$

As both front finger and rear finger are lists of tree nodes, in order to calculate the total size of finger, we can provide a $size''(L)$ function, which sums up size of all nodes stored in the list. Denote $L = \{a_1, a_2, ...\}$ and $L' = \{a_2, a_3, ...\}$.

$$size''(L) = \begin{cases} 0 & : \quad L = \Phi \\ size'(a_1) + size''(L') & : \quad otherwise \end{cases} \tag{12.39}$$

It's quite OK to define $size''(L)$ by using some high order functions. For example.

$$size''(L) = sum(map(size', L)) \tag{12.40}$$

And we can turn a list of tree nodes into one deeper 2-3 tree and vice-versa.

$$\begin{aligned} wraps(L) &= tr(size''(L), L) \\ unwraps(n) &= L \quad : \quad n = tr(s, L) \end{aligned} \tag{12.41}$$

These helper functions are translated to the following Haskell code.

```
size (Br s _) = s

sizeL = sum ∘ (map size)

sizeT Empty = 0
sizeT (Lf a) = size a
sizeT (Tr s _ _ _) = s
```

Here are the wrap and unwrap auxiliary functions.

```
wrap x = Br 1 [x]
unwrap (Br 1 [x]) = x
wraps xs = Br (sizeL xs) xs
unwraps (Br _ xs) = xs
```

We omitted their type definitions for illustration purpose.

In imperative settings, the size information for node and tree can be accessed through the size field. And the size of a list of nodes can be summed up for this field as the below algorithm.

function SIZE-NODES(L)
  $s \leftarrow 0$
  for $\forall n \in L$ **do**
    $s \leftarrow s+$ SIZE(n)
  return s

The following Python code, for example, translates this algorithm by using standard `sum()` and `map()` functions provided in library.

```
def sizeNs(xs):
    return sum(map(lambda x: x.size, xs))
```

As NIL is typically used to represent empty tree in imperative settings, it's convenient to provide a auxiliary size function to uniformed calculate the size of tree no matter it is NIL.

function SIZE-TR(T)
  if $T =$ NIL **then**
    return 0
  else
    return SIZE(T)

The algorithm is trivial and we skip its implementation example program.

Modification due to the augmented size

The algorithms have been presented so far need to be modified to accomplish with the augmented size. For example the $insertT()$ function now inserts a tree node instead of a plain element.

$$insertT(x, T) = insertT'(wrap(x), T) \tag{12.42}$$

The corresponding Haskell program is changed as below.

```
cons a t = cons' (wrap a) t
```

After being wrapped, x is augmented with size information of 1. In the implementation of previous insertion algorithm, function $tree(F, M, R)$ is used to create a finger tree from a front finger, a middle part inner tree and a rear finger. This function should also be modified to add size information of these three arguments.

$$tree'(F, M, R) = \begin{cases} fromL(F) & : & M = \Phi \wedge R = \Phi \\ fromL(R) & : & M = \Phi \wedge F = \Phi \\ tree'(unwraps(F'), M', R) & : & F = \Phi, (F', M') = extractT'(M \\ tree'(F, M', unwraps(R')) & : & R = \Phi, (M', R') = removeT'(M \\ tree(size''(F) + size(M) + size''(R), F, M, R) & : & otherwise \end{cases} \tag{12.43}$$

Where $fromL()$ helps to turn a list of nodes to a finger tree by repeatedly inserting all the element one by one to an empty tree.

$$fromL(L) = foldR(insertT', \Phi, L)$$

Of course it can be implemented in pure recursive manner without using folding as well.

The last case is the most straightforward one. If none of F, M, and R is empty, it adds the size of these three part and construct the tree along with this size information by calling $tree(s, F, M, R)$ function. If both the middle part inner tree and one of the finger is empty, the algorithm repeatedly insert all elements stored in the other finger to an empty tree, so that the result is constructed from a list of tree nodes. If the middle part inner tree isn't empty, and one of the finger is empty, the algorithm 'borrows' one tree node from the middle part, either by extracting from head if front finger is empty or removing from tail if rear finger is empty. Then the algorithm unwraps the 'borrowed' tree node to a list, and recursively call $tree'()$ function to construct the result.

This algorithm can be translated to the following Haskell code for example.

```
tree f Empty [] = foldr cons' Empty f
tree [] Empty r = foldr cons' Empty r
tree [] m r = let (f, m') = uncons' m in tree (unwraps f) m' r
tree f m [] = let (m', r) = unsnoc' m in tree f m' (unwraps r)
tree f m r = Tr (sizeL f + sizeT m + sizeL r) f m r
```

Function $tree'()$ helps to minimize the modification. $insertT'()$ can be realized by using it like the following.

$$insertT'(x, T) = \begin{cases} leaf(x) & : \quad T = \Phi \\ tree'(\{x\}, \Phi, \{y\}) & : \quad T = leaf(x) \\ tree'(\{x, x_1\}, insertT'(wraps(\{x_2, x_3, x_4\}), M), R) & : \quad T = tree(s, \{x_1, x_2, x_3, x_4\}, M, R) \\ tree'(\{x\} \cup F, M, R) & : \quad otherwise \end{cases} \tag{12.44}$$

And it's corresponding Haskell code is a line by line translation.

```
cons' a Empty = Lf a
cons' a (Lf b) = tree [a] Empty [b]
cons' a (Tr _ [b, c, d, e] m r) = tree [a, b] (cons' (wraps [c, d, e]) m) r
cons' a (Tr _ f m r) = tree (a:f) m r
```

The similar modification for augment size should also be tuned for imperative algorithms, for example, when a new node is prepend to the head of the finger tree, we should update size when traverse the tree.

function PREPEND-NODE(n, T)
  $r \leftarrow$ TREE()
  $p \leftarrow r$
  CONNECT-MID(p, T)
  while FULL?(FRONT(T)) **do**
    $F \leftarrow$ FRONT(T)
    FRONT(T) $\leftarrow \{n, F[1]\}$
    SIZE(T) $\leftarrow$ SIZE(T) + SIZE(n) ▷ update size
    $n \leftarrow$ NODE()
    CHILDREN(n) $\leftarrow F[2..]$
    $p \leftarrow T$
    $T \leftarrow$ MID(T)
  if $T =$ NIL **then**
    $T \leftarrow$ TREE()
    FRONT(T)$\leftarrow \{n\}$
  else if $\mid$ FRONT(T) $\mid = 1 \wedge$ REAR(T) $= \Phi$ **then**

```
        REAR(T) ← FRONT(T)
        FRONT(T) ← {n}
    else
        FRONT(T) ← {n}∪ FRONT(T)
    SIZE(T) ← SIZE(T) + SIZE(n)                    ▷ update size
    CONNECT-MID(p, T) ← T
    return FLAT(r)
```

The corresponding Python code are modified accordingly as below.

```
def prepend_node(n, t):
    root = prev = Tree()
    prev.set_mid(t)
    while frontFull(t):
        f = t.front
        t.front = [n] + f[:1]
        t.size = t.size + n.size
        n = wraps(f[1:])
        prev = t
        t = t.mid
    if t is None:
        t = leaf(n)
    elif len(t.front)==1 and t.rear == []:
        t = Tree(n.size + t.size, [n], None, t.front)
    else:
        t = Tree(n.size + t.size, [n]+t.front, t.mid, t.rear)
    prev.set_mid(t)
    return flat(root)
```

Note that the tree constructor is also modified to take a size argument as the first parameter. And the `leaf()` helper function does not only construct the tree from a node, but also set the size of the tree with the same size of the node inside it.

For simplification purpose, we skip the detailed description of what are modified in $extractT()'$, $appendT()$, $removeT()$, and $concat()$ algorithms. They are left as exercises to the reader.

Split a finger tree at a given position

With size information augmented, it's easy to locate a node at given position by performing a tree search. What's more, as the finger tree is constructed from three part F, M, and R; and it's nature of recursive, it's also possible to split it into three sub parts with a given position i: the left, the node at i, and the right part.

The idea is straight forward. Since we have the size information for F, M, and R. Denote these three sizes as S_f, S_m, and S_r. if the given position $i \leq S_f$, the node must be stored in F, we can go on seeking the node inside F; if $S_f < i \leq S_f + S_m$, the node must be stored in M, we need recursively perform search in M; otherwise, the node should be in R, we need search inside R.

If we skip the error handling of trying to split an empty tree, there is only one edge case as below.

$$splitAt(i, T) = \begin{cases} (\Phi, x, \Phi) & : & T = leaf(x) \\ ... & : & otherwise \end{cases}$$

Splitting a leaf results both the left and right parts empty, the node stored in leaf is the resulting node.

The recursive case handles the three sub cases by comparing i with the sizes. Suppose function $splitAtL(i, L)$ splits a list of nodes at given position i into three parts: $(A, x, B) = splitAtL(i, L)$, where x is the i-th node in L, A is a sub list contains all nodes before position i, and B is a sub list contains all rest nodes after i.

$$splitAt(i, T) = \begin{cases} (\Phi, x, \Phi) & : & T = leaf(x) \\ (fromL(A), x, tree'(B, M, R)) & : & i \leq S_f, (A, x, B) = splitAtL(i, F) \\ (tree'(F, M_l, A), x, tree'(B, M_r, R)) & : & S_f < i \leq S_f + S_m \\ (tree'(F, M, A), x, fromL(B)) & : & otherwise, (A, x, B) = splitAtL(i - S_f - S_m, R) \end{cases} \tag{12.45}$$

Where M_l, x, M_r, A, B in the thrid case are calculated as the following.

$$\begin{aligned} &(M_l, t, M_r) = splitAt(i - S_f, M) \\ &(A, x, B) = splitAtL(i - S_f - size(M_l), unwraps(t)) \end{aligned}$$

And the function $splitAtL()$ is just a linear traverse, since the length of list is limited not to exceed the constraint of 2-3 tree, the performance is still ensured to be constant $O(1)$ time. Denote $L = \{x_1, x_2, ...\}$ and $L' = \{x_2, x_3, ...\}$.

$$splitAtL(i, L) = \begin{cases} (\Phi, x_1, \Phi) & : & i = 0 \wedge L = \{x_1\} \\ (\Phi, x_1, L') & : & i < size'(x_1) \\ (\{x_1\} \cup A, x, B) & : & otherwise, (A, x, B) = splitAtL(i - size'(x_1), L') \end{cases} \tag{12.46}$$

The solution of splitting is a typical divide and conquer strategy. The performance of this algorithm is determined by the recursive case of searching in middle part inner tree. Other cases are all constant time as we've analyzed. The depth of recursion is proportion to the height of the tree h, so the algorithm is bound to $O(h)$. Because the tree is well balanced (by using 2-3 tree, and all the insertion/removal algorithms keep the tree balanced), so $h = O(\lg N)$ where N is the number of elements stored in finger tree. The overall performance of splitting is $O(\lg N)$.

Let's first give the Haskell program for $splitAtL()$ function

```
splitNodesAt 0 [x] = ([], x, [])
splitNodesAt i (x:xs) | i < size x = ([], x, xs)
                      | otherwise = let (xs', y, ys) = splitNodesAt (i-size x) xs
                                    in (x:xs', y, ys)
```

Then the program for $splitAt()$, as there is already function defined in standard library with this name, we slightly change the name by adding a apostrophe.

```
splitAt' _ (Lf x) = (Empty, x, Empty)
splitAt' i (Tr _ f m r)
    | i < szf = let (xs, y, ys) = splitNodesAt i f
```

```
            in ((foldr cons' Empty xs), y, tree ys m r)
  | i < szf + szm = let (m1, t, m2) = splitAt' (i-szf) m
                        (xs, y, ys) = splitNodesAt (i-szf - sizeT m1) (unwraps t)
                    in (tree f m1 xs, y, tree ys m2 r)
  | otherwise = let (xs, y, ys) = splitNodesAt (i-szf -szm) r
                in (tree f m xs, y, foldr cons' Empty ys)
  where
    szf = sizeL f
    szm = sizeT m
```

Random access

With the help of splitting at any arbitrary position, it's trivial to realize random access in $O(\lg N)$ time. Denote function $mid(x)$ returns the 2-nd element of a tuple, $left(x)$, and $right(x)$ return the first element and the 3-rd element of the tuple respectively.

$$getAt(S, i) = unwrap(mid(splitAt(i, S))) \tag{12.47}$$

It first splits the sequence at position i, then unwraps the node to get the element stored inside it. When mutate the i-th element of sequence S represented by finger tree, we first split it at i, then we replace the middle to what we want to mutate, and re-construct them to one finger tree by using concatenation.

$$setAt(S, i, x) = concat(L, insertT(x, R)) \tag{12.48}$$

where

$$(L, y, R) = splitAt(i, S)$$

What's more, we can also realize a $removeAt(S, i)$ function, which can remove the i-th element from sequence S. The idea is first to split at i, unwrap and return the element of the i-th node; then concatenate the left and right to a new finger tree.

$$removeAt(S, i) = (unwrap(y), concat(L, R)) \tag{12.49}$$

These handy algorithms can be translated to the following Haskell program.

```
getAt t i = unwrap x where (_, x, _) = splitAt' i t
setAt t i x = let (l, _, r) = splitAt' i t in concat' l (cons x r)
removeAt t i = let (l, x, r) = splitAt' i t in (unwrap x, concat' l r)
```

Imperative random access

As we can directly mutate the tree in imperative settings, it's possible to realize GET-AT(T, i) and SET-AT(T, i, x) without using splitting. The idea is firstly implement a algorithm which can apply some operation to a given position. The following algorithm takes three arguments, a finger tree T, a position index at i which ranges from zero to the number of elements stored in the tree, and a function f, which will be applied to the element at i.

function APPLY-AT(T, i, f)
  while SIZE$(T) > 1$ **do**
    $S_f \leftarrow$ SIZE-NODES(FRONT(T))

```
        S_m ← SIZE-TR(MID(T))
        if i < S_f then
            return LOOKUP-NODES(FRONT(T), i, f)
        else if i < S_f + S_m then
            T ← MID(T)
            i ← i − S_f
        else
            return LOOKUP-NODES(REAR(T), i − S_f − S_m, f)
    n ← FIRST-LF(T)
    x ← ELEM(n)
    ELEM(n) ← f(x)
    return x
```

This algorithm is essentially a divide and conquer tree search. It repeatedly examine the current tree till reach a tree with size of 1 (can it be determined as a leaf? please consider the ill-formed case and refer to the exercise later). Every time, it checks the position to be located with the size information of front finger and middle part inner tree.

If the index i is less than the size of front finger, the location is at some node in it. The algorithm call a sub procedure to look-up among front finger; If the index is between the size of front finger and the total size till middle part inner tree, it means that the location is at some node inside the middle, the algorithm goes on traverse along the middle part inner tree with an updated index reduced by the size of front finger; Otherwise it means the location is at some node in rear finger, the similar looking up procedure is called accordingly.

After this loop, we've got a node, (can be a compound node) with what we are looking for at the first leaf inside this node. We can extract the element out, and apply the function f on it and store the new value back.

The algorithm returns the previous element before applying f as the final result.

What hasn't been factored is the algorithm LOOKUP-NODES(L, i, f). It takes a list of nodes, a position index, and a function to be applied. This algorithm can be implemented by checking every node in the list. If the node is a leaf, and the index is zero, we are at the right position to be looked up. The function can be applied on the element stored in this leaf, and the previous value is returned; Otherwise, we need compare the size of this node and the index to determine if the position is inside this node and search inside the children of the node if necessary.

```
function LOOKUP-NODES(L, i, f)
    loop
        for ∀n ∈ L do
            if n is leaf ∧i = 0 then
                x ← ELEM(n)
                ELEM(n) ← f(x)
                return x
            if i < SIZE(n) then
                L ← CHILDREN(n)
                break
            i ← i− SIZE(n)
```

The following are the corresponding Python code implements the algorithms.

```
def applyAt(t, i, f):
    while t.size > 1:
        szf = sizeNs(t.front)
        szm = sizeT(t.mid)
        if i < szf:
            return lookupNs(t.front, i, f)
        elif i < szf + szm:
            t = t.mid
            i = i - szf
        else:
            return lookupNs(t.rear, i - szf - szm, f)
    n = first_leaf(t)
    x = elem(n)
    n.children[0] = f(x)
    return x

def lookupNs(ns, i, f):
    while True:
        for n in ns:
            if n.leaf and i == 0:
                x = elem(n)
                n.children[0] = f(x)
                return x
            if i < n.size:
                ns = n.children
                break
            i = i - n.size
```

With auxiliary algorithm that can apply function at a given position, it's trivial to implement the GET-AT() and SET-AT() by passing special function for applying.

function GET-AT(T, i)
  return APPLY-AT($T, i, \lambda_x.x$)

function SET-AT(T, i, x)
  return APPLY-AT($T, i, \lambda_y.x$)

That is we pass `id` function to implement getting element at a position, which doesn't change anything at all; and pass constant function to implement setting, which set the element to new value by ignoring its previous value.

Imperative splitting

It's not enough to just realizing APPLY-AT algorithm in imperative settings, this is because removing element at arbitrary position is also a typical case.

Almost all the imperative finger tree algorithms so far are kind of one-pass top-down manner. Although we sometimes need to book keeping the root. It means that we can even realize all of them without using the parent field.

Splitting operation, however, can be easily implemented by using parent field. We can first perform a top-down traverse along with the middle part inner tree as long as the splitting position doesn't located in front or rear finger. After that, we need a bottom-up traverse along with the parent field of the two split trees to fill out the necessary fields.

```
function SPLIT-AT(T, i)
    T_1 ← TREE()
    T_2 ← TREE()
    while S_f ≤ i < S_f + S_m do                          ▷ Top-down pass
        T'_1 ← TREE()
        T'_2 ← TREE()
        FRONT(T'_1) ← FRONT(T)
        REAR(T'_2) ← REAR(T)
        CONNECT-MID(T_1, T'_1)
        CONNECT-MID(T_2, T'_2)
        T_1 ← T'_1
        T_2 ← T'_2
        i ← i − S_f
        T ← MID(T)
    if i < S_f then
        (X, n, Y) ← SPLIT-NODES(FRONT(T), i)
        T'_1 ← FROM-NODES(X)
        T'_2 ← T
        SIZE(T'_2) ← SIZE(T) - SIZE-NODES(X) - SIZE(n)
        FRONT(T'_2) ← Y
    else if S_f + S_m ≤ i then
        (X, n, Y) ← SPLIT-NODES(REAR(T), i − S_f − S_m)
        T'_2 ← FROM-NODES(Y)
        T'_1 ← T
        SIZE(T'_1) ← SIZE(T) - SIZE-NODES(Y) - SIZE(n)
        REAR(T'_1) ← X
    CONNECT-MID(T_1, T'_1)
    CONNECT-MID(T_2, T'_2)
    i ← i− SIZE-TR(T'_1)
    while n is NOT leaf do                                ▷ Bottom-up pass
        (X, n, Y) ← SPLIT-NODES(CHILDREN(n), i)
        i ← i− SIZE-NODES(X)
        REAR(T_1) ← X
        FRONT(T_2) ← Y
        SIZE(T_1) ← SUM-SIZES(T_1)
        SIZE(T_2) ← SUM-SIZES(T_2)
        T_1 ← PARENT(T_1)
        T_2 ← PARENT(T_2)
    return (FLAT(T_1), ELEM(n), FLAT(T_2))
```

The algorithm first creates two trees T_1 and T_2 to hold the split results. Note that they are created as 'ground' trees which are parents of the roots. The first pass is a top-down pass. Suppose S_f, and S_m retrieve the size of the front finger and the size of middle part inner tree respectively. If the position at which the tree to be split is located at middle part inner tree, we reuse the front finger of T for new created T'_1, and reuse rear finger of T for T'_2. At this time point, we can't fill the other fields for T'_1 and T'_2, they are left empty, and we'll finish filling them in the future. After that, we connect T_1 and T'_1 so the latter becomes the middle part inner tree of the former. The similar connection is done for T_2 and T'_2 as well. Finally, we update the position by deducing it by the size of front

finger, and go on traversing along with the middle part inner tree.

When the first pass finishes, we are at a position that either the splitting should be performed in front finger, or in rear finger. Splitting the nodes in finger results a tuple, that the first part and the third part are lists before and after the splitting point, while the second part is a node contains the element at the original position to be split. As both fingers hold at most 3 nodes because they are actually 2-3 trees, the nodes splitting algorithm can be performed by a linear search.

function SPLIT-NODES(L, i)
  for $j \in [1,$ LENGTH(L) $]$ **do**
    if $i <$ SIZE($L[j]$) **then**
      return ($L[1...j-1]$, $L[j]$, $L[j+1...$ LENGTH(L) $]$)
    $i \leftarrow i-$ SIZE($L[j]$)

We next create two new result trees T_1' and T_2' from this tuple, and connected them as the final middle part inner tree of T_1 and T_2.

Next we need perform a bottom-up traverse along with the result trees to fill out all the empty information we skipped in the first pass.

We loop on the second part of the tuple, the node, till it becomes a leaf. In each iteration, we repeatedly splitting the children of the node with an updated position i. The first list of nodes returned from splitting is used to fill the rear finger of T_1; and the other list of nodes is used to fill the front finger of T_2. After that, since all the three parts of a finger tree – the front and rear finger, and the middle part inner tree – are filled, we can then calculate the size of the tree by summing these three parts up.

function SUM-SIZES(T)
  return SIZE-NODES(FRONT(T)) + SIZE-TR(MID(T)) + SIZE-NODES(REAR(T))

Next, the iteration goes on along with the parent fields of T_1 and T_2. The last 'black-box' algorithm is FROM-NODES(L), which can create a finger tree from a list of nodes. It can be easily realized by repeatedly perform insertion on an empty tree. The implementation is left as an exercise to the reader.

The example Python code for splitting is given as below.

```
def splitAt(t, i):
    (t1, t2) = (Tree(), Tree())
    while szf(t) ≤ i and i < szf(t) + szm(t):
        fst = Tree(0, t.front, None, [])
        snd = Tree(0, [], None, t.rear)
        t1.set_mid(fst)
        t2.set_mid(snd)
        (t1, t2) = (fst, snd)
        i = i - szf(t)
        t = t.mid

    if i < szf(t):
        (xs, n, ys) = splitNs(t.front, i)
        sz = t.size - sizeNs(xs) - n.size
        (fst, snd) = (fromNodes(xs), Tree(sz, ys, t.mid, t.rear))
    elif szf(t) + szm(t) ≤ i:
        (xs, n, ys) = splitNs(t.rear, i - szf(t) - szm(t))
        sz = t.size - sizeNs(ys) - n.size
```

```
        (fst, snd) = (Tree(sz, t.front, t.mid, xs), fromNodes(ys))
    t1.set_mid(fst)
    t2.set_mid(snd)

    i = i - sizeT(fst)
    while not n.leaf:
        (xs, n, ys) = splitNs(n.children, i)
        i = i - sizeNs(xs)
        (t1.rear, t2.front) = (xs, ys)
        t1.size = sizeNs(t1.front) + sizeT(t1.mid) + sizeNs(t1.rear)
        t2.size = sizeNs(t2.front) + sizeT(t2.mid) + sizeNs(t2.rear)
        (t1, t2) = (t1.parent, t2.parent)

    return (flat(t1), elem(n), flat(t2))
```

The program to split a list of nodes at a given position is listed like this.

```
def splitNs(ns, i):
    for j in range(len(ns)):
        if i < ns[j].size:
            return (ns[:j], ns[j], ns[j+1:])
        i = i - ns[j].size
```

With splitting defined, removing an element at arbitrary position can be realized trivially by first performing a splitting, then concatenating the two result tree to one big tree and return the element at that position.

function REMOVE-AT(T, i)
  $(T_1, x, T_2) \leftarrow$ SPLIT-AT(T, i)
  return (x, CONCAT(T_1, T_2))

Exercise 12.6

1. Another way to realize $insertT'()$ is to force increasing the size field by one, so that we needn't write function $tree'()$. Try to realize the algorithm by using this idea.

2. Try to handle the augment size information as well as in $insertT'()$ algorithm for the following algorithms (both functional and imperative): $extractT()'$, $appendT()$, $removeT()$, and $concat()$. The *head*, *tail*, *init* and *last* functions should be kept unchanged. Don't refer to the downloadable programs along with this book before you take a try.

3. In the imperative APPLY-AT() algorithm, it tests if the size of the current tree is greater than one. Why don't we test if the current tree is a leaf? Tell the difference between these two approaches.

4. Implement the FROM-NODES(L) in your favorite imperative programming language. You can either use looping or create a folding-from-right sub algorithm.

12.7 Notes and short summary

Although we haven't been able to give a purely functional realization to match the $O(1)$ constant time random access as arrays in imperative settings. The result finger tree data structure achieves an overall well performed sequence. It manipulates fast in amortized $O(1)$ time both on head an on tail, it can also concatenates two sequence in logarithmic time as well as break one sequence into two sub sequences at any position. While neither arrays in imperative settings nor linked-list in functional settings satisfies all these goals. Some functional programming languages adopt this sequence realization in its standard library [7].

Just as the title of this chapter, we've presented the last corner stone of elementary data structures in both functional and imperative settings. We needn't concern about being lack of elementary data structures when solve problems with some typical algorithms.

For example, when writing a MTF (move-to-front) encoding algorithm[8], with the help of the sequence data structure explained in this chapter. We can implement it quite straightforward.

$$mtf(S, i) = \{x\} \cup S'$$

where $(x, S') = removeAt(S, i)$.

In the next following chapters, we'll first explains some typical divide and conquer sorting methods, including quick sort, merge sort and their variants; then some elementary searching algorithms, and string matching algorithms; finally, we'll give a real-world example of algorithms, BWT (Burrows-Wheeler transform) compressor, which is one of the best compression tool in the world.

Bibliography

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

[2] Chris Okasaki. “Purely Functional Random-Access Lists”. Functional Programming Languages and Computer Architecture, June 1995, pages 86-95.

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

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

[5] Ralf Hinze and Ross Paterson. “Finger Trees: A Simple General-purpose Data Structure.” in Journal of Functional Programming16:2 (2006), pages 197-217. http://www.soi.city.ac.uk/ ross/papers/FingerTree.html

[6] Guibas, L. J., McCreight, E. M., Plass, M. F., Roberts, J. R. (1977), ”A new representation for linear lists”. Conference Record of the Ninth Annual ACM Symposium on Theory of Computing, pp. 49C60.

[7] Generic finger-tree structure. http://hackage.haskell.org/packages/archive/fingertree/0.0/doc/html/Data-FingerTree.html

[8] Wikipedia. Move-to-front transform. http://en.wikipedia.org/wiki/Move-to-front_transform

Part IV

Appendix

Appendices

Lists
Liu Xinyu **Liu Xinyu**
Email: liuxinyu95@gmail.com

Appendix A

Lists

A.1 Introduction

This book intensely uses recursive list manipulations in purely functional settings. List can be treated as a counterpart to array in imperative settings, which are the bricks to many algorithms and data structures.

For the readers who are not familiar with functional list manipulation, this appendix provides a quick reference. All operations listed in this appendix are not only described in equations, but also implemented in both functional programming languages as well as imperative languages as examples. We also provide a special type of implementation in C++ template meta programming similar to [3] for interesting in next appendix.

Besides the elementary list operations, this appendix also contains explanation of some high order function concepts such as mapping, folding etc.

A.2 List Definition

Like arrays in imperative settings, lists play a critical role in functional setting[1]. Lists are built-in support in some programming languages like Lisp families and ML families so it needn't explicitly define list in those environment.

List, or more precisely, singly linked-list is a data structure that can be described below.

- A *list* is either empty;
- Or contains an element and a *list*.

Note that this definition is recursive. Figure A.1 illustrates a list with N nodes. Each node contains two part, a key element and a sub list. The sub list contained in the last node is empty, which is denoted as 'NIL'.

[1]Some reader may argue that 'lambda calculus plays the most critical role'. Lambda calculus is somewhat as assembly languages to the computation world, which is worthy studying from the essence of computation model to the practical programs. However, we don't dive into the topic in this book. Users can refer to [4] for detail.

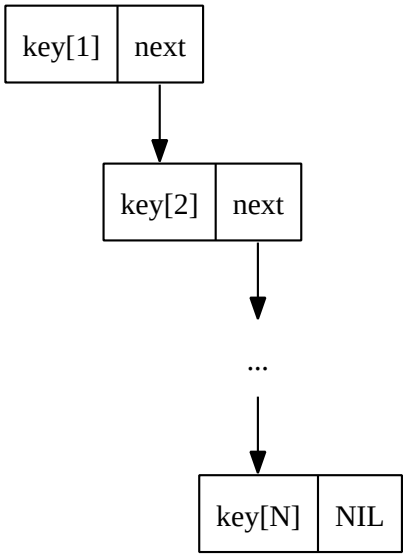


Figure A.1: A list contains N nodes

This data structure can be explicitly defined in programming languages support record (or compound type) concept. The following ISO C++ code defines list[2].

```
template<typename T>
struct List {
  T key;
  List* next;
};
```

A.2.1 Empty list

It's worth to mention about 'empty' list a bit more in detail. In environment supporting the nil concept, for example, C or java like programming languages, empty list can have two different representations. One is the trivial 'NIL' (or null, or 0, which varies from languages); the other is an non-NIL empty list as {}, the latter is typically allocated with memory but filled with nothing. In Lisp dialects, the empty is commonly written as `'()`. In ML families, it's written as `[]`. We use Φ to denote empty list in equations and use 'NIL' in pseudo code sometimes to describe algorithms in this book.

A.2.2 Access the element and the sub list

Given a list L, two functions can be defined to access the element stored in it and the sub list respectively. They are typically denoted as $first(L)$, and $rest(L)$ or $head(L)$ and $tail(L)$ for the same meaning. These two functions are named as `car` and `cdr` in Lisp for historic reason about the design of machine

[2]We only use template to parameterize the type of the element in this chapter. Except this point, all imperative source code are in ANSI C style to avoid language specific features.

registers [5]. In languages support Pattern matching (e.g. ML families, Prolog and Erlang etc.) These two functions are commonly realized by matching the `cons` which we'll introduced later. for example the following Haskell program:

```
head (x:xs) = x
tail (x:xs) = xs
```

If the list is defined in record syntax like what we did above, these two functions can be realized by accessing the record fields [3].

```
template<typename T>
T first(List<T> *xs) { return xs->key; }

template<typename T>
List<T>* rest(List<T>* xs) { return xs->next; }
```

In this book, L' is used to denote the $rest(L)$ sometimes, also we uses l_1 to represent $first(L)$ in the context that the list is literately given in form $L = \{l_1, l_2, ..., l_N\}$.

More interesting, as far as in an environment support recursion, we can define List. The following example define a list of integers in C++ compile time.

```
struct Empty;

template<int x, typename T> struct List {
  static const int first = x;
  typedef T rest;
};
```

This line constructs a list of $\{1, 2, 3, 4, 5\}$ in compile time.

```
typedef List<1, List<2, List<3, List<4 List<5, Empty> > > > > A;
```

A.3 Basic list manipulation

A.3.1 Construction

The last C++ template meta programming example actually shows literate construction of a list. A list can be constructed from an element with a sub list, where the sub list can be empty. We denote function $cons(x, L)$ as the constructor. This name is used in most Lisp dialects. In ML families, there are 'cons' operator defined as `::`, (in Haskell it's `:`).

We can define `cons` to create a record as we defined above in ISO C++, for example[4].

```
template<typename T>
List<T>* cons(T x, List<T>* xs) {
  List<T>* lst = new List<T>;
  lst->key = x;
  lst->next = xs;
  return lst;
}
```

[3] They can be also named as 'key' and 'next' or be defined as class methods.

[4] It's often defined as a constructor method for the class template, However, we define it as a standalone function for illustration purpose.

A.3.2 Empty testing and length calculating

It's trivial to test if a list is empty. If the environment contains nil concept, the testing should also handle nil case. Both Lisp dialects and ML families provide `null` testing functions. Empty testing can also be realized by pattern-matching with empty list if possible. The following Haskell program shows such example.

```
null [] = True
null _ = False
```

In this book we will either use $empty(L)$ or $L = \Phi$ where empty testing happens.

With empty testing defined, it's possible to calculate length for a list. In imperative settings, LENGTH is often implemented like the following.

function LENGTH(L)
  $n \leftarrow 0$
  while $L \neq NIL$ **do**
    $n \leftarrow n + 1$
    $L \leftarrow$ NEXT(L)

This ISO C++ code translates the algorithm to real program.

```
template<typename T>
int length(List<T>* xs) {
  int n = 0;
  for (; xs; ++n, xs = xs->next);
  return n
}
```

However, in purely funcitonal setting, we can't mutate a counter variable. the idea is that, if the list is empty, then its size is zero; otherwise, we can recursively calculate the length of the sub list, then add it by one to get the length of this list.

$$length(L) = \begin{cases} 0 & : \quad L = \Phi \\ 1 + length(L') & : \quad otherwise \end{cases} \tag{A.1}$$

Here $L' = rest(L)$ as mentioned above, it's $\{l_2, l_3, ..., l_N\}$ for list contains N elements. Note that both L and L' can be empty Φ. In this equation, we also use '=' to test if list L is empty. In order to know the length of a list, we need traverse all the elements from the head to the end, so that this algorithm is proportion to the number of elements stored in the list. It's a linear algorithm bound to $O(N)$ time.

Below are two programs in Haskell and in Scheme/Lisp realize this recursive algorithm.

```
length [] = 0
length (x:xs) = 1 + length xs
```

```
(define (length lst)
  (if (null? lst) 0 (+ 1 (length (cdr lst)))))
```

How to testing if two lists are identical is left as exercise to the reader.

A.3.3 indexing

One big difference between array and list (singly-linked list accurately) is that array supports random access. Many programming languages support using `x[i]` to access the i-th element stored in array in constant $O(1)$ time. The index typically starts from 0, but it's not the all case. Some programming languages using 1 as the first index. In this appendix, we treat index starting from 0. However, we must traverse the list with i steps to reach the target element. The traversing is quite similar to the length calculation. Thus it's commonly expressed as below in imperative settings.

function GET-AT(L, i)
  while $i \neq 0$ **do**
    $L \leftarrow$ NEXT(L)
  return FIRST(L)

Note that this algorithm doesn't handle the error case such that the index isn't within the bound of the list. We assume that $0 \leq i < |L|$, where $|L| = length(L)$. The error handling is left as exercise to the reader. The following ISO C++ code is a line-by-line translation of this algorithm.

```
template<typename T>
T getAt(List<T>* lst, int n) {
  while(n--)
    lst = lst→next;
  return lst→key;
}
```

However, in purely functional settings, we turn to recursive traversing instead of while-loop.

$$getAt(L, i) = \begin{cases} First(L) & : \quad i = 0 \\ getAt(Rest(L), i - 1) & : \quad otherwise \end{cases} \tag{A.2}$$

In order to *get the i-th element*, the algorithm does the following:

- if i is 0, then we are done, the result is the first element in the list;
- Otherwise, the result is to *get the $(i-1)$-th element* from the sub-list.

This algorithm can be translated to the following Haskell code.

```
getAt i (x:xs) = if i == 0 then x else getAt i-1 xs
```

Note that we are using pattern matching to ensure the list isn't empty, which actually handles all out-of-bound cases with un-matched pattern error. Thus if $i > |L|$, we finally arrive at a edge case that the index is $i - |L|$, while the list is empty; On the other hand, if $i < 0$, minus it by one makes it even farther away from 0. We finally end at the same error that the index is some negative, while the list is empty;

The indexing algorithm takes time proportion to the value of index, which is bound to $O(N)$ linear time. This section only address the read semantics. How to mutate the element at a given position is explained in later section.

A.3.4 Access the last element

Although accessing the first element and the rest list L' is trivial, the opposite operations, that retrieving the last element and the initial sub list need linear time without using a tail pointer. If the list isn't empty, we need traverse it till the tail to get these two components. Below are their imperative descriptions.

```
function LAST(L)
    x ← NIL
    while L ≠ NIL do
        x ← FIRST(L)
        L ← REST(L)
    return x

function INIT(L)
    L' ← NIL
    while REST(L) ≠ NIL do
        L' ← APPEND(L', FIRST(L))
        L ← REST(L)
    return L'
```

The algorithm assumes that the input list isn't empty, so the error handling is skipped. Note that the INIT() algorithm uses the appending algorithm which will be defined later.

Below are the corresponding ISO C++ implementation. The optimized version by utilizing tail pointer is left as exercise.

```
template<typename T>
T last(List<T>* xs) {
  T x; /* Can be set to a special value to indicate empty list err. */
  for (; xs; xs = xs→next)
    x = xs→key;
  return x;
}

template<typename T>
List<T>* init(List<T>* xs) {
  List<T>* ys = NULL;
  for (; xs→next; xs = xs→next)
    ys = append(ys, xs→key);
  return ys;
}
```

While these two algorithms can be implemented in purely recursive manner as well. When we want to access *the last element*.

- If the list contains only one element (the rest sub-list is empty), the result is this very element;
- Otherwise, the result is *the last element* of the rest sub-list.

$$last(L) = \begin{cases} First(L) & : & Rest(L) = \Phi \\ last(Rest(L)) & : & otherwise \end{cases} \tag{A.3}$$

The similar approach can be used to *get a list contains all elements except for the last one.*

- The edge case: If the list contains only one element, then the result is an empty list;
- Otherwise, we can first *get a list contains all elements except for the last one* from the rest sub-list, then construct the final result from the first element and this intermediate result.

$$init(L) = \begin{cases} \Phi & : & L' = \Phi \\ cons(l_1, init(L')) & : & otherwise \end{cases} \tag{A.4}$$

Here we denote l_1 as the first element of L, and L' is the rest sub-list. This recursive algorithm needn't use appending, It actually construct the final result list from right to left. We'll introduce a high-level concept of such kind of computation later in this appendix.

Below are Haskell programs implement $last()$ and $init()$ algorithms by using pattern matching.

```
last [x] = x
last (_:xs) = last xs

init [x] = []
init (x:xs) = x : init xs
```

Where `[x]` matches the singleton list contains only one element, while `(_:xs)` matches any non-empty list, and the underscore (`_`) is used to indicate that we don't care about the element. For the detail of pattern matching, readers can refer to any Haskell tutorial materials, such as [8].

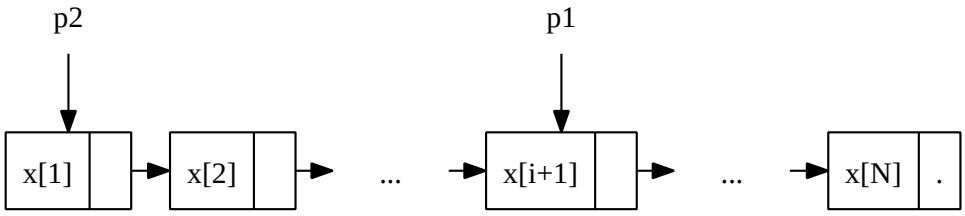
A.3.5 Reverse indexing

Reverse indexing is a general case for $last()$, finding the i-th element in a singly-linked list with the minimized memory spaces is interesting, and this problem is often used in technical interview in some companies. A naive implementation takes 2 rounds of traversing, the first round is to determine the length of the list N, then, calculate the left-hand index by $N - i - 1$. Finally a second round of traverse is used to access the element with the left-hand index. This idea can be give as the following equation.

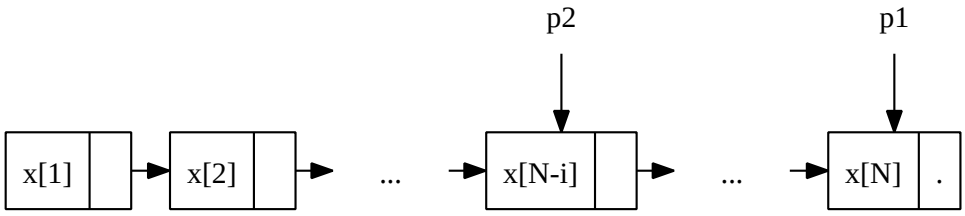
$$getAtR(L, i) = getAt(L, length(L) - i - 1)$$

There exists better imperative solution. For illustration purpose, we omit the error cases such as index is out-of-bound etc. The idea is to keep two pointers p_1, p_2, with the distance of i between them, that $rest^i(p_2) = p_1$, where $rest^i(p_1)$ means repleatedly apply $rest()$ function i times. It says that succeeds i steps from p_2 gets p_1. We can start p_2 from the head of the list and advance the two pointers in parallel till one of them (p_1) arrives at the end of the list. At that time point, pointer p_2 exactly arrived at the i-th element from right. Figure A.2 illustrates this idea.

It's straightforward to realize the imperative algorithm based on this 'double pointers' solution.



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



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

Figure A.2: Double pointers solution to reverse indexing.

function GET-AT-R(L, i)
  $p \leftarrow L$
  while $i \neq 0$ **do**
    $L \leftarrow \text{REST}(L)$
    $i \leftarrow i - 1$
  while $\text{REST}(L) \neq$ NIL **do**
    $L \leftarrow \text{REST}(L)$
    $p \leftarrow \text{REST}(p)$
  return $\text{FIRST}(p)$

The following ISO C++ code implements the 'double pointers' right indexing algorithm.

```
template<typename T>
T getAtR(List<T>* xs, int i) {
  List<T>* p = xs;
  while(i--)
    xs = xs→next;
  for(; xs→next; xs = xs→next, p = p→next);
  return p→key;
}
```

The same idea can be realized recursively as well. If we want to access the i-th element of list L, we can examine the two lists L and $S = \{l_i, l_{i+1}, ..., l_N\}$ simultaneously, where S is a sub-list of L without the first i elements.

- The edge case: If S is a singleton list, then the i-th element from right is the first element in L;
- Otherwise, we drop the first element from L and S, and recursively exam-

ine L' and S'.

This algorithm description can be formalized as the following equations.

$$getAtR(L, i) = examine(L, drop(i, L)) \tag{A.5}$$

Where function $examine(L, S)$ is defined as below.

$$examine(L, S) = \begin{cases} first(L) & : & |S| = 1 \\ examine(rest(L), rest(S)) & : & otherwise \end{cases} \tag{A.6}$$

We'll explain the detail of *drop*() function in later section about list mutating operations. Here it can be implemented as repeatedly call *rest*() with specified times.

$$drop(n, L) = \begin{cases} L & : & n = 0 \\ drop(n - 1, rest(L)) & : & otherwise \end{cases}$$

Translating the equations to Haskell yields this example program.

```
atR :: [a] → Int → a
atR xs i = get xs (drop i xs) where
  get (x:_) [_] = x
  get (_:xs) (_:ys) = get xs ys
  drop n as@(_:as') = if n == 0 then as else drop (n-1) as'
```

Here we use dummy variable `_` as the placeholders for components we don't care.

A.3.6 Mutating

Strictly speaking, we can't mutate the list at all in purely functional settings. Unlike in imperative settings, mutate is actually realized by creating new list. Almost all functional environments support garbage collection, the original list may either be persisted for reusing, or released (dropped) at sometime (Chapter 2 in [6]).

Appending

Function *cons* can be viewed as building list by insertion element always on head. If we chains multiple *cons* operations, it can repeatedly construct a list from right to the left. Appending on the other hand, is an operation adding element to the tail. Compare to *cons* which is trivial constant time $O(1)$ operation, We must traverse the whole list to locate the appending position. It means that appending is bound to $O(N)$, where N is the length of the list. In order to speed up the appending, imperative implementation typically uses a field (variable) to record the tail position of a list, so that the traversing can be avoided. However, in purely functional settings we can't use such 'tail' pointer. The appending has to be realized in recursive manner.

$$append(L, x) = \begin{cases} \{x\} & : & L = \Phi \\ cons(first(L), append(rest(L), x)) & : & otherwise \end{cases} \tag{A.7}$$

That the algorithm handles two different appending cases:

- If the list is empty, the result is a singleton list contains x, which is the element to be appended. The singleton list notion $\{x\} = cons(x, \Phi)$, is a simplified form of *cons* the element with an empty list Φ;
- Otherwise, for the none-empty list, the result can be achieved by first appending the element x to the rest sub-list, then construct the first element of L with the recursive appending result.

For the none-trivial case, if we denote $L = \{l_1, l_2, ...\}$, and $L' = \{l_2, l_3, ...\}$ the equation can be written as.

$$append(L, x) = \begin{cases} \{x\} & : \quad L = \Phi \\ cons(l_1, append(L', x)) & : \quad otherwise \end{cases} \tag{A.8}$$

We'll use both forms in the rest of this appendix.

The following Scheme/Lisp program implements this algorithm.

```
(define (append lst x)
  (if (null? lst)
      (list x)
      (cons (car lst) (append (cdr lst) x))))
```

Even without the tail pointer, it's possible to traverse the list imperatively and append the element at the end.

function APPEND(L, x)
  if $L =$ NIL **then**
    return CONS(x, NIL)
  $H \leftarrow L$
  while REST(L) $\neq$ NIL **do**
    $L \leftarrow$ REST(L)
  REST(L) $\leftarrow$ CONS(x, NIL)
  return H

The following ISO C++ programs implements this algorithm. How to utilize a tail field to speed up the appending is left as exercise to the reader for interesting.

```
template<typename T>
List<T>* append(List<T>* xs, T x) {
  List<T> *tail, *head;
  for (head = tail = xs; xs; xs = xs→next)
    tail = xs;
  if (!head)
    head = cons<T>(x, NULL);
  else
    tail→next = cons<T>(x, NULL);
  return head;
}
```

Mutate element at a given position

Although we have defined random access algorithm $getAt(L, i)$, we can't just mutate the element returned by this function in a sense of purely functional

settings. It's quite common to provide reference semantics in imperative programming languages and in some 'almost' functional environment. Readers can refer to [4] for detail. For example, the following ISO C++ example returns a reference instead of a value in indexing program.

```
template<typename T>
T& getAt(List<T>* xs, int n) {
  while (n--)
    xs = xs->next;
  return xs->key;
}
```

So that we can use this function to mutate the 2nd element as below.

```
List<int>* xs = cons(1, cons(2, cons<int>(3, NULL)));
getAt(xs, 1) = 4;
```

In an impure functional environment, such as Scheme/Lisp, to set the i-th element to a given value can be implemented by mutate the referenced cell directly as well.

```
(define (set-at! lst i x)
  (if (= i 0)
      (set-car! lst x)
      (set-at! (cdr lst) (- i 1) x)))
```

This program first checks if the index i is zero, if so, it mutate the first element of the list to given value x; otherwise, it deduces the index i by one, and tries to mutate the rest of the list at this new index with value x. This function doesn't return meaningful value. It is for use of side-effect. For instance, the following code mutates the 2nd element in a list.

```
(define lst '(1 2 3 4 5))
(set-at! lst 1 4)
(display lst)

(1 4 3 4 5)
```

In order to realize a purely functional $setAt(L, i, x)$ algorithm, we need avoid directly mutating the cell, but creating a new one:

- Edge case: If we want to set the value of the first element ($i = 0$), we construct a new list, with the new value and the sub-list of the previous one;
- Otherwise, we construct a new list, with the previous first element, and a new sub-list, which has the $(i-1)$-th element set with the new value.

This recursive description can be formalized by the following equation.

$$setAt(L, i, x) = \begin{cases} cons(x, L') & : & i = 0 \\ cons(l_1, setAt(L', i-1, x)) & : & otherwise \end{cases} \tag{A.9}$$

Comparing the below Scheme/Lisp implementation to the previous one reveals the difference from imperative mutating.

```
(define (set-at lst i x)
  (if (= i 0)
      (cons x (cdr lst))
      (cons (car lst) (set-at (cdr lst) (- i 1) x))))
```

Here we skip the error handling for out-of-bound error etc. Again, similar to the random access algorithm, the performance is bound to linear time, as traverse is need to locate the position to set the value.

insertion

There are two semantics about list insertion. One is to insert an element at a given position, which can be denoted as $insert(L, i, x)$. The algorithm is close to $setAt(L, i, x)$; The other is to insert an element to a sorted list, so that the the result list is still sorted.

Let's first consider how to insert an element x at a given position i. The obvious thing is that we need firstly traverse i elements to get to the position, the rest of work is to construct a new sub-list with x being the head of this sub-list. Finally, we construct the whole result by attaching this new sub-list to the end of the first i elements.

The algorithm can be described accordingly to this idea. If we want to insert an element x to a list L at i.

- Edge case: If i is zero, then the insertion turns to be a trivial 'cons' operation – $cons(x, L)$;
- Otherwise, we recursively *insert* x to the sub-list L' at position $i - 1$; then construct the first element with this result.

Below equation formalizes the insertion algorithm.

$$insert(L, i, x) = \begin{cases} cons(x, L) & : & i = 0 \\ cons(l_1, insert(L', i - 1, x)) & : & otherwise \end{cases} \tag{A.10}$$

The following Haskell program implements this algorithm.

```
insert xs 0 y = y:xs
insert (x:xs) i y = x : insert xs (i-1) y
```

This algorithm doesn't handle the out-of-bound error. However, we can interpret the case, that the position i exceeds the length of the list as appending. Readers can considering about it in the exercise of this section.

The algorithm can also be designed imperatively: If the position is zero, just construct the new list with the element to be inserted as the first one; Otherwise, we record the head of the list, then start traversing the list i steps. We also need an extra variable to memorize the previous position for the later list insertion operation. Below is the pseudo code.

function INSERT(L, i, x)
  if $i = 0$ **then**
    return CONS(x, L)
  $H \leftarrow L$
  $p \leftarrow L$

while $i \neq 0$ **do**
$p \leftarrow L$
$L \leftarrow \textsc{Rest}(L)$
$i \leftarrow i - 1$
$\textsc{Rest}(p) \leftarrow \textsc{Cons}(x, L)$
return H

And the ISO C++ example program is given by translating this algorithm.

```
template<typename T>
List<T>* insert(List<T>* xs, int i , int x) {
  List<T> *head, *prev;
  if (i == 0)
    return cons(x, xs);
  for (head = xs; i; --i, xs = xs→next)
    prev = xs;
  prev→next = cons(x, xs);
  return head;
}
```

If the list L is sorted, that is for any position $1 \leq i \leq j \leq N$, we have $l_i \leq l_j$. We can design an algorithm which inserts a new element x to the list, so that the result list is still sorted.

$$insert(x, L) = \begin{cases} cons(x, \Phi) & : & L = \Phi \\ cons(x, L) & : & x < l_1 \\ cons(l_1, insert(x, L')) & : & otherwise \end{cases} \tag{A.11}$$

The idea is that, to insert an element x to a sorted list L:

- If either L is empty or x is less than the first element in L, we just put x in front of L to construct the result;
- Otherwise, we recursively insert x to the sub-list L'.

The following Haskell program implements this algorithm. Note that we use $\leq$, to determine the ordering. Actually this constraint can be loosened to the strict less ($<$), that if elements can be compare in terms of $<$, we can design a program to insert element so that the result list is still sorted. Readers can refer to the chapters of sorting in this book for details about ordering.

```
insert y [] = [y]
insert y xs@(x:xs') = if y ≤ x then y : xs else x : insert y xs'
```

Since the algorithm need compare the elements one by one, it's also a linear time algorithm. Note that here we use the 'as' notion for pattern matching in Haskell. Readers can refer to [8] and [7] for details.

This ordered insertion algorithm can be designed in imperative manner, for example like the following pseudo code[5].

function $\textsc{Insert}(x, L)$
if $L = \Phi \vee x < \textsc{First}(L)$ **then**
return $\textsc{Cons}(x, L)$

[5]Reader can refer to the chapter 'The evolution of insertion sort' in this book for a minor different one

$H \leftarrow L$
while $\textsc{Rest}(L) \neq \Phi \wedge \textsc{First}(\textsc{Rest}(L)) < x$ **do**
$L \leftarrow \textsc{Rest}(L)$
$\textsc{Rest}(L) \leftarrow \textsc{Cons}(x, \textsc{Rest}(L))$
return H

If either the list is empty, or the new element to be inserted is less than the first element in the list, we can just put this element as the new first one; Otherwise, we record the head, then traverse the list till a position, where x is less than the rest of the sub-list, and put x in that position. Compare this one to the 'insert at' algorithm shown previously, the variable p uses to point to the previous position during traversing is omitted by examine the sub-list instead of current list. The following ISO C++ program implements this algorithm.

```
template<typename T>
List<T>* insert(T x, List<T>* xs) {
  List<T> *head;
  if (!xs || x < xs→key)
    return cons(x, xs);
  for (head = xs; xs→next && xs→next→key < x; xs = xs→next);
  xs→next = cons(x, xs→next);
  return head;
}
```

With this linear time ordered insertion defined, it's possible to implement quadratic time insertion-sort by repeatedly inserting elements to an empty list as formalized in this equation.

$$sort(L) = \begin{cases} \Phi & : \quad L = \Phi \\ insert(l_1, sort(L')) & : \quad otherwise \end{cases} \tag{A.12}$$

This equation says that if the list to be sorted is empty, the result is also empty, otherwise, we can firstly recursively sort all elements except for the first one, then ordered insert the first element to this intermediate result. The corresponding Haskell program is given as below.

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

And the imperative linked-list base insertion sort is described in the following. That we initialize the result list as empty, then take the element one by one from the list to be sorted, and ordered insert them to the result list.

function $\textsc{Sort}(L)$
$L' \leftarrow \Phi$
while $L \neq \Phi$ **do**
$L' \leftarrow \textsc{Insert}(\textsc{First}(L), L')$
$L \leftarrow \textsc{Rest}(L)$
return L'

Note that, at any time during the loop, the result list is kept sorted. There is a major difference between the recursive algorithm (formalized by the equation) and the procedural one (described by the pseudo code), that the former process the list from right, while the latter from left. We'll see in later section about 'tail-recursion' how to eliminate this difference.

The ISO C++ version of linked-list insertion sort is list like this.

```
template<typename T>
List<T>* isort(List<T>* xs) {
  List<T>* ys = NULL;
  for(; xs; xs = xs→next)
    ys = insert(xs→key, ys);
  return ys;
}
```

There is also a dedicated chapter discusses insertion sort in this book. Please refer to that chapter for more details including performance analysis and fine-tuning.

deletion

In purely functional settings, there is no deletion at all in terms of mutating, the data is persist, what the semantic deletion means is actually to create a 'new' list with all the elements in previous one except for the element being 'deleted'.

Similar to the insertion, there are also two deletion semantics. One is to delete the element at a given position; the other is to find and delete elements of a given value. The first can be expressed as $delete(L, i)$, while the second is $delete(L, x)$.

In order to design the algorithm $delete(L, i)$ (or 'delete at'), we can use the idea which is quite similar to random access and insertion, that we first traverse the list to the specified position, then construct the result list with the elements we have traversed, and all the others except for the next one we haven't traversed yet.

The strategy can be realized in a recursive manner that in order to *delete the i-th element from list L*,

- If i is zero, that we are going to delete the first element of a list, the result is obviously the rest of the list;
- If the list to be removed element is empty, the result is anyway empty;
- Otherwise, we can recursively *delete the $(i-1)$-th element from the sub-list L'*, then construct the final result from the first element of L and this intermediate result.

Note there are two edge cases, and the second case is major used for error handling. This algorithm can be formalized with the following equation.

$$delete(L, i) = \begin{cases} L' & : \quad i = 0 \\ \Phi & : \quad L = \Phi \\ cons(l_1, delete(L', i-1)) & : \end{cases} \tag{A.13}$$

Where $L' = rest(L), l_1 = first(L)$. The corresponding Haskell example program is given below.

```
del (_:xs) 0 = xs
del [] _ = []
del (x:xs) i = x : del xs (i-1)
```

This is a linear time algorithm as well, and there are also alternatives for implementation, for example, we can first split the list at position $i-1$, to get 2 sub-lists L_1 and L_2, then we can concatenate L_1 and L_2'.

The 'delete at' algorithm can also be realized imperatively, that we traverse to the position by looping:

```
function DELETE(L, i)
    if i = 0 then
        return REST(L)
    H ← L
    p ← L
    while i ≠ 0 do
        i ← i − 1
        p ← L
        L ← REST(L)
    REST(p) ← REST(L)
    return H
```

Different from the recursive approach, The error handling for out-of-bound is skipped. Besides that the algorithm also skips the handling of resource releasing which is necessary in environments without GC (Garbage collection). Below ISO C++ code for example, explicitly releases the node to be deleted.

```
template<typename T>
List<T>* del(List<T>* xs, int i) {
  List<T> *head, *prev;
  if (i == 0)
    head = xs→next;
  else {
    for (head = xs; i; --i, xs = xs→next)
      prev = xs;
    prev→next = xs→next;
  }
  xs→next = NULL;
  delete xs;
  return head;
}
```

Note that the statement `xs->next = NULL` is neccessary if the destructor is designed to release the whole linked-list recursively.

The 'find and delete' semantic can further be represented in two ways, one is just find the first occurrence of a given value, and delete this element from the list; The other is to find *ALL* occurrence of this value, and delete these elements. The later is more general case, and it can be achieved by a minor modification of the former. We left the 'find all and delete' algorithm as an exercise to the reader.

The algorithm can be designed exactly as the term 'find and delete' but not 'find then delete', that the finding and deleting are processed in one pass traversing.

- If the list to be dealt with is empty, the result is obviously empty;
- If the list isn't empty, we examine the first element of the list, if it is identical to the given value, the result is the sub list;
- Otherwise, we keep the first element, and recursively find and delete the element in the sub list with the given value. The final result is a list constructed with the kept first element, and the recursive deleting result.

This algorithm can be formalized by the following equation.

$$delete(L, x) = \begin{cases} \Phi & : & L = \Phi \\ L' & : & l_1 = x \\ cons(l_1, delete(L', x)) & : & otherwise \end{cases} \tag{A.14}$$

This algorithm is bound to linear time as it traverses the list to find and delete element. Translating this equation to Haskell program yields the below code, note that, the first edge case is handled by pattern-matching the empty list; while the other two cases are further processed by if-else expression.

```
del [] _ = []
del (x:xs) y = if x == y then xs else x : del xs y
```

Different from the above imperative algorithms, which skip the error handling in most cases, the imperative 'find and delete' realization must deal with the problem that the given value doesn't exist.

```
function DELETE(L, x)
    if L = Φ then                                    ▷ Empty list
        return Φ
    if FIRST(L) = x then
        H ← REST(L)
    else
        H ← L
        while L ≠ Φ ∧ FIRST(L) ≠ x do                ▷ List isn't empty
            p ← L
            L ← REST(L)
        if L ≠ Φ then                                ▷ Found
            REST(p) ← REST(L)
    return H
```

If the list is empty, the result is anyway empty; otherwise, the algorithm traverses the list till either finds an element identical to the given value or to the end of the list. If the element is found, it is removed from the list. The following ISO C++ program implements the algorithm. Note that there are codes release the memory explicitly.

```
template<typename T>
List<T>* del(List<T>* xs, T x) {
  List<T> *head, *prev;
  if (!xs)
    return xs;
  if (xs→key == x)
    head = xs→next;
  else {
    for (head = xs; xs && xs→key != x; xs = xs→next)
      prev = xs;
    if (xs)
      prev→next = xs→next;
  }
  if (xs) {
    xs→next = NULL;
    delete xs;
  }
```

```
  return head;
}
```

concatenate

Concatenation can be considered as a general case for appending, that appending only adds one more extra element to the end of the list, while concatenation adds multiple elements.

However, It will lead to quadratic algorithm if implement concatenation naively by appending, which performs poor. Consider the following equation.

$$concat(L_1, L_2) = \begin{cases} L_1 & : & L_2 = \Phi \\ concat(append(L_1, first(L_2)), rest(L_2)) & : & otherwise \end{cases}$$

Note that each appending algorithm need traverse to the end of the list, which is proportion to the length of L_1, and we need do this linear time appending work $|L_2|$ times, so the total performance is $O(|L_1| + (|L_1| + 1) + ... + (|L_1| + |L_2|)) = O(|L_1||L_2| + |L_2|^2)$.

The key point is that the linking operation of linked-list is fast (constant $O(1)$ time), we can traverse to the end of L_1 only one time, and link the second list to the tail of L_1.

$$concat(L_1, L_2) = \begin{cases} L_2 & : & L_1 = \Phi \\ cons(first(L_1), concat(rest(L_1), L_2)) & : & otherwise \end{cases} \tag{A.15}$$

This algorithm only traverses the first list one time to get the tail of L_1, and then linking the second list with this tail. So the algorithm is bound to linear $O(|L_1|)$ time.

This algorithm is described as the following.

- If the first list is empty, the concatenate result is the second list;
- Otherwise, we concatenate the second list to the sub-list of the first one, and construct the final result with the first element and this intermediate result.

Most functional languages provide built-in functions or operators for list concatenation, for example in ML families `++` is used for this purpose.

```
[] ++ ys = ys
xs ++ [] = xs
(x:xs) ++ ys = x : xs ++ ys
```

Note we add another edge case that if the second list is empty, we needn't traverse to the end of the first one and perform linking, the result is merely the first list.

In imperative settings, concatenation can be realized in constant $O(1)$ time with the augmented tail record. We skip the detailed implementation for this method, reader can refer to the source code which can be download along with this appendix.

The imperative algorithm without using augmented tail record can be described as below.

```
function CONCAT(L_1, L_2)
    if L_1 = Φ then
        return L_2
    if L_2 = Φ then
        return L_1
    H ← L_1
    while REST(L_1) ≠ Φ do
        L_1 ← REST(L_1)
    REST(L_1) ← L_2
    return H
```

And the corresponding ISO C++ example code is given like this.

```
template<typename T>
List<T>* concat(List<T>* xs, List<T>* ys) {
  List<T>* head;
  if (!xs)
    return ys;
  if (!ys)
    return xs;
  for (head = xs; xs→next; xs = xs→next);
  xs→next = ys;
  return head;
}
```

A.3.7 sum and product

Recursive sum and product

It's common to calculate the sum or product of a list of numbers. They are quite similar in terms of algorithm structure. We'll see how to abstract such structure in later section.

In order to calculate the *sum of a list*:

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

Formalize this description gives the following equation.

$$sum(L) = \begin{cases} 0 & : \quad L = \Phi \\ l_1 + sum(L') & : \quad otherwise \end{cases} \tag{A.16}$$

However, we can't merely replace plus to times in this equation to achieve product algorithm, because it always returns zero. We can define the product of empty list as 1 to solve this problem.

$$product(L) = \begin{cases} 1 & : \quad L = \Phi \\ l_1 \times product(L') & : \quad otherwise \end{cases} \tag{A.17}$$

The following Haskell program implements sum and product.

```
sum [] = 0
sum (x:xs) = x + sum xs

product [] = 1
product (x:xs) = x * product xs
```

Both algorithms traverse the whole list during calculation, so they are bound to $O(N)$ linear time.

Tail call recursion

Note that both sum and product algorithms actually compute the result from right to left. We can change them to the normal way, that calculate the *accumulated* result from left to right. For example with sum, the result is actually accumulated from 0, and adds element one by one to this accumulated result till all the list is consumed. Such approach can be described as the following.

When accumulate result of a list by summing:

- If the list is empty, we are done and return the accumulated result;
- Otherwise, we take the first element from the list, accumulate it to the result by summing, and go on processing the rest of the list.

Formalize this idea to equation yields another version of sum algorithm.

$$sum'(A, L) = \begin{cases} A & : \quad L = \Phi \\ sum'(A + l_1, L') & : \quad otherwise \end{cases} \tag{A.18}$$

And sum can be implemented by calling this function by passing start value 0 and the list as arguments.

$$sum(L) = sum'(0, L) \tag{A.19}$$

The interesting point of this approach is that, besides it calculates the result in a normal order from left to right; by observing the equation of $sum'(A, L)$, we found it needn't remember any intermediate results or states when perform recursion. All such states are either passed as arguments (A for example) or can be dropped (previous elements of the list for example). So in a practical implementation, such kind of recursive function can be optimized by eliminating the recursion at all.

We call such kind of function as 'tail recursion' (or 'tail call'), and the optimization of removing recursion in this case as 'tail recursion optimization'[10] because the recursion happens as the final action in such function. The advantage of tail recursion optimization is that the performance can be greatly improved, so that we can avoid the issue of stack overflow in deep recursion algorithms such as sum and product.

Changing the sum and product Haskell programs to tail-recursion manner gives the following modified programs.

```
sum = sum' 0 where
    sum' acc [] = acc
    sum' acc (x:xs) = sum' (acc + x) xs

product = product' 1 where
    product' acc [] = acc
    product' acc (x:xs) = product' (acc * x) xs
```

In previous section about insertion sort, we mentioned that the functional version sorts the elements form right, this can also be modified to tail recursive realization.

$$sort'(A, L) = \begin{cases} A & : \quad L = \Phi \\ sort'(insert(l_1, A), L') & : \quad otherwise \end{cases} \tag{A.20}$$

The the sorting algorithm is just calling this function by passing empty list as the accumulator argument.

$$sort(L) = sort'(\Phi, L) \tag{A.21}$$

Implementing this tail recursive algorithm to real program is left as exercise to the reader.

As the end of this sub-section, let's consider an interesting problem, that how to design an algorithm to compute b^N effectively? (refer to problem 1.16 in [5].)

A naive brute-force solution is to repeatedly multiply b for N times from 1, which leads to a linear $O(N)$ algorithm.

function POW(b, N)
  $x \leftarrow 1$
  loop N times
    $x \leftarrow x \times b$
  return x

Actually, the solution can be greatly improved. Consider we are trying to calculate b^8. By the first 2 iterations in above naive algorithm, we got $x = b^2$. At this stage, we needn't multiply x with b to get b^3, we can directly calculate x^2, which leads to b^4. And if we do this again, we get $(b^4)^2 = b^8$. Thus we only need looping 3 times but not 8 times.

An algorithm based on this idea to compute b^N if $N = 2^M$ for some non-negative integer M can be shown in the following equation.

$$pow(b, N) = \begin{cases} b & : \quad N = 1 \\ pow(b, \frac{N}{2})^2 & : \quad otherwise \end{cases}$$

It's easy to extend this divide and conquer algorithm so that N can be any non-negative integer.

- For the trivial case, that N is zero, the result is 1;
- If N is even number, we can halve N, and compute $b^{\frac{N}{2}}$ first. Then calculate the square number of this result.
- Otherwise, N is odd. Since $N-1$ is even, we can first recursively compute b^{N-1}, the multiply b one more time to this result.

Below equation formalizes this description.

$$pow(b, N) = \begin{cases} 1 & : \quad N = 0 \\ pow(b, \frac{N}{2})^2 & : \quad 2|N \\ b \times pow(b, N-1) & : \quad otherwise \end{cases} \tag{A.22}$$

However, it's hard to turn this algorithm to tail-recursive mainly because the 2nd clause. In fact, the 2nd clause can be alternatively realized by squaring the base number, and halve the exponent.

$$pow(b, N) = \begin{cases} 1 & : & N = 0 \\ pow(b^2, \frac{N}{2}) & : & 2|N \\ b \times pow(b, N-1) & : & otherwise \end{cases} \tag{A.23}$$

With this change, it's easy to get a tail-recursive algorithm as the following, so that $b^N = pow'(b, N, 1)$.

$$pow'(b, N, A) = \begin{cases} A & : & N = 0 \\ pow'(b^2, \frac{N}{2}, A) & : & 2|N \\ pow'(b, N-1, A \times b) & : & otherwise \end{cases} \tag{A.24}$$

Compare to the naive brute-force algorithm, we improved the performance to $O(\lg N)$. Actually, this algorithm can be improved even one more step.

Observe that if we represent N in binary format $N = (a_m a_{m-1} ... a_1 a_0)_2$, we clear know that the computation for b^{2^i} is necessary if $a_i = 1$. This is quite similar to the idea of Binomial heap (reader can refer to the chapter of binomial heap in this book). Thus we can calculate the final result by multiplying all of them for bits with value 1.

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

1. calculate b^1, which is b;
2. Get b^2 from previous result;
3. Get b^{2^2} from step 2;
4. Get b^{2^3} from step 3.

Finally, we multiply the results of step 1, 2, and 4 which yields b^{11}.

Summarize this idea, we can improve the algorithm as below.

$$pow'(b, N, A) = \begin{cases} A & : & N = 0 \\ pow'(b^2, \frac{N}{2}, A) & : & 2|N \\ pow'(b^2, \lfloor \frac{N}{2} \rfloor, A \times b) & : & otherwise \end{cases} \tag{A.25}$$

This algorithm essentially shift N to right for 1 bit each time (by dividing N by 2). If the LSB (Least Significant Bit, which is the lowest bit) is 0, it means N is even. It goes on computing the square of the base, without accumulating the final product (Just like the 3rd step in above example); If the LSB is 1, it means N is odd. It squares the base and accumulates it to the product A; The edge case is when N is zero, which means we exhaust all the bits in N, thus the final result is the accumulator A. At any time, the updated base number b', the shifted exponent number N', and the accumulator A satisfy the invariant that $b^N = b'^{N'} A$.

This algorithm can be implemented in Haskell like the following.

```
pow b n = pow' b n 1 where
  pow' b n acc | n == 0 = acc
               | even n = pow' (b*b) (n `div` 2) acc
               | otherwise = pow' (b*b) (n `div` 2) (acc*b)
```

Compare to previous algorithm, which minus N by one to change it to even when N is odd, this one halves N every time. It exactly runs m rounds, where m is the number of bits of N. However, the performance is still bound to $O(\lg N)$. How to implement this algorithm imperatively is left as exercise to the reader.

Imperative sum and product

The imperative sum and product are just applying plus and times while traversing the list.

function SUM(L)
  $s \leftarrow 0$
  while $L \neq \Phi$ **do**
    $s \leftarrow s +$ FIRST(L)
    $L \leftarrow$ REST(L)
  return s

function PRODUCT(L)
  $p \leftarrow 1$
  while $L \neq \Phi$ **do**
    $p \leftarrow p \times$ FIRST(L)
    $L \leftarrow$ REST(L)
  return p

The corresponding ISO C++ example programs are list as the following.

```
template<typename T>
T sum(List<T>* xs) {
  T s;
  for (s = 0; xs; xs = xs->next)
    s += xs->key;
  return s;
}

template<typename T>
T product(List<T>* xs) {
  T p;
  for (p = 1; xs; xs = xs->next)
    p *= xs->key;
  return p;
}
```

One interesting usage of product algorithm is that we can calculate factorial of N by calculating the product of $\{1, 2, ..., N\}$ that $N! = product([1..N])$.

A.3.8 maximum and minimum

Another very useful use case is to get the minimum or maximum element of a list. We'll see that their algorithm structures are quite similar again. We'll

generalize this kind of feature and introduce about higher level abstraction in later section. For both maximum and minimum algorithms, we assume that the given list isn't empty.

In order to find the minimum element in a list.

- If the list contains only one element, (a singleton list), the minimum element is this one;
- Otherwise, we can firstly find the minimum element of the rest list, then compare the first element with this intermediate result to determine the final minimum value.

This algorithm can be formalized by the following equation.

$$min(L) = \begin{cases} l_1 & : & L = \{l_1\} \\ l_1 & : & l_1 \leq min(L') \\ min(L') & : & otherwise \end{cases} \tag{A.26}$$

In order to get the maximum element instead of the minimum one, we can simply replace the $\leq$ comparison to $\geq$ in the above equation.

$$max(L) = \begin{cases} l_1 & : & L = \{l_1\} \\ l_1 & : & l_1 \geq max(L') \\ max(L') & : & otherwise \end{cases} \tag{A.27}$$

Note that both maximum and minimum actually process the list from right to left. It remind us about tail recursion. We can modify them so that the list is processed from left to right. What's more, the tail recursion version brings us 'on-line' algorithm, that at any time, we hold the minimum or maximum result of the list we examined so far.

$$min'(L, a) = \begin{cases} a & : & L = \Phi \\ min(L', l_1) & : & l_1 < a \\ min(L', a) & : & otherwise \end{cases} \tag{A.28}$$

$$max'(L, a) = \begin{cases} a & : & L = \Phi \\ max(L', l_1) & : & a < l_1 \\ max(L', a) & : & otherwise \end{cases} \tag{A.29}$$

Different from the tail recursion sum and product, we can't pass constant value to min', or max' in practice, this is because we have to pass infinity ($min(L, \infty)$) or negative infinity ($max(L, -\infty)$) in theory, but in a real machine neither of them can be represented since the length of word is limited.

Actually, there is workaround, we can instead pass the first element of the list, so that the algorithms become applicable.

$$\begin{aligned} min(L) &= min(L', l_1) \\ max(L) &= max(L', l_1) \end{aligned} \tag{A.30}$$

The corresponding real programs are given as the following. We skip the none tail recursion programs, as they are intuitive enough. Reader can take them as exercises for interesting.

```
min (x:xs) = min' xs x where
    min' [] a = a
    min' (x:xs) a = if x < a then min' xs x else min' xs a

max (x:xs) = max' xs x where
    max' [] a = a
    max' (x:xs) a = if a < x then max' xs x else max' xs a
```

The tail call version can be easily translated to imperative min/max algorithms.

function MIN(L)
  $m \leftarrow$ FIRST(L)
  $L \leftarrow$ REST(L)
  while $L \neq \Phi$ **do**
    if FIRST(L) $< m$ **then**
      $m \leftarrow$ FIRST(L)
    $L \leftarrow$ REST(L)
  return m

function MAX(L)
  $m \leftarrow$ FIRST(L)
  $L \leftarrow$ REST(L)
  while $L \neq \Phi$ **do**
    if $m <$ FIRST(L) **then**
      $m \leftarrow$ FIRST(L)
    $L \leftarrow$ REST(L)
  return m

The corresponding ISO C++ programs are given as below.

```
template<typename T>
T min(List<T>* xs) {
  T x;
  for (x = xs->key; xs; xs = xs->next)
    if (xs->key < x)
      x = xs->key;
  return x;
}

template<typename T>
T max(List<T>* xs) {
  T x;
  for (x = xs->key; xs; xs = xs->next)
    if (x < xs->key)
      x = xs->key;
  return x;
}
```

Another method to achieve tail-call maximum(and minimum) algorithm is by discarding the smaller element each time. The edge case is as same as before; for recursion case, since there are at least two elements in the list, we can take the first two for comparing, then drop one and go on process the rest. For a list with more than two elements, denote L'' as $rest(rest(L)) = \{l_3, l_4, ...\}$, we have

the following equation.

$$max(L) = \begin{cases} l_1 & : & |L| = 1 \\ max(cons(l_1, L'')) & : & l_2 < l_1 \\ max(L') & : & otherwise \end{cases} \tag{A.31}$$

$$min(L) = \begin{cases} l_1 & : & |L| = 1 \\ min(cons(l_1, L'')) & : & l_1 < l_2 \\ min(L') & : & otherwise \end{cases} \tag{A.32}$$

The relative example Haskell programs are given as below.

```
min [x] = x
min (x:y:xs) = if x < y then min (x:xs) else min (y:xs)

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

Exercise A.1

- Given two lists L_1 and L_2, design a algorithm $eq(L_1, L_2)$ to test if they are equal to each other. Here equality means the lengths are same, and at the same time, every elements in both lists are identical.
- Consider varies of options to handle the out-of-bound error case when randomly access the element in list. Realize them in both imperative and functional programming languages. Compare the solutions based on exception and error code.
- Augment the list with a 'tail' field, so that the appending algorithm can be realized in constant $O(1)$ time but not linear $O(N)$ time. Feel free to choose your favorite imperative programming language. Please don't refer to the example source code along with this book before you try it.
- With 'tail' field augmented to list, for which list operations this field must be updated? How it affects the performance?
- Handle the out-of-bound case in insertion algorithm by treating it as appending.
- Write the insertion sort algorithm by only using less than ($<$).
- Design and implement the algorithm that find all the occurrence of a given value and delete them from the list.
- Reimplenent the algorithm to calculate the length of a list in tail-call recursion manner.
- Implement the insertion sort in tail recursive manner.
- Implement the $O(\lg N)$ algorithm to calculate b^N in your favorite imperative programming language. Note that we only need accumulate the intermediate result when the bit is not zero.

A.4 Transformation

In previous section, we list some basic operations for linked-list. In this section, we focus on the transformation algorithms for list. Some of them are corner stones of abstraction for functional programming. We'll show how to use list transformation to solve some interesting problems.

A.4.1 mapping and for-each

It's every-day programming routine that, we need output something as readable string. If we have a list of numbers, and we want to print the list to console like '3 1 2 5 4'. One option is to convert the numbers to strings, so that we can feed them to the printing function. One such trivial conversion program may like this.

$$toStr(L) = \begin{cases} \Phi & : & L = \Phi \\ cons(str(l_1), toStr(L')) & : & otherwise \end{cases} \tag{A.33}$$

The other example is that we have a dictionary which is actually a list of words grouped in their initial letters, for example: [[a, an, another, ...], [bat, bath, bool, bus, ...], ..., [zero, zoo, ...]]. We want to know the frequency of them in English, so that we process some English text, for example, 'Hamlet' or the 'Bible' and augment each of the word with a number of occurrence in these texts. Now we have a list like this:

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

If we want to find which word in each initial is used most, how to write a program to work this problem out? The output is a list of words that every one has the most occurrence in the group, which is categorized by initial, something like '[a, but, can, ...]'. We actually need a program which can transfer a list of group of augmented words into a list of words.

Let's work it out step by step. First, we need define a function, which takes a list of word - number pairs, and find the word has the biggest number augmented. Sorting is overkill. What we need is just a special $max'()$ function, Note that the $max()$ function developed in previous section can't be used directly. Suppose for a pair of values $p = (a, b)$, function $fst(p) = a$, and $snd(p) = b$ are accessors to extract the values, $max'()$ can be defined as the following.

$$max'(L) = \begin{cases} l_1 & : & |L| = 1 \\ l_1 & : & snd(max'(L')) < snd(l_1) \\ max'(L') & : & otherwise \end{cases} \tag{A.34}$$

Alternatively, we can define a dedicated function to compare word-number of occurrence pair, and generalize the $max()$ function by passing a compare function.

$$less(p_1, p_2) = snd(p_1) < snd(p_2) \tag{A.35}$$

$$maxBy(cmp, L) = \begin{cases} l_1 & : & |L| = 1 \\ l_1 & : & cmp(l_1, maxBy(cmp, L')) \\ maxBy(cmp, L') & : & otherwise \end{cases} \tag{A.36}$$

Then $max'()$ is just a special case of $maxBy()$ with the compare function comparing on the second value in a pair.

$$max'(L) = maxBy(less, L) \tag{A.37}$$

Here we write all functions in purely recursive way, they can be modified in tail call manner. This is left as exercise to the reader.

With $max'()$ function defined, it's possible to complete the solution by processing the whole list.

$$solve(L) = \begin{cases} \Phi & : & L = \Phi \\ cons(fst(max'(l_1)), solve(L')) & : & otherwise \end{cases} \tag{A.38}$$

Map

Compare the $solve()$ function in (A.38) and $toStr()$ function in (A.33), it reveals very similar algorithm structure. although they targets on very different problems, and one is trivial while the other is a bit complex.

The structure of $toStr()$ applies the function $str()$ which can turn a number into string on every element in the list; while $solve()$ first applies $max'()$ function to every element (which is actually a list of pairs), then applies $fst()$ function, which essentially turns a list of pairs into a string. It's not hard to abstract such common structure like the following equation, which is called as *mapping*.

$$map(f, L) = \begin{cases} \Phi & : & L = \Phi \\ cons(f(l_1)), map(f, L')) & : & otherwise \end{cases} \tag{A.39}$$

Because map takes a 'converter' function f as argument, it's called a kind of high-order function. In functional programming environment such as Haskell, mapping can be implemented just like the above equation.

```
map :: (a→b)→[a]→[b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

The two concrete cases we discussed above can all be represented in high order mapping.

$$\begin{aligned} toStr &= map \quad str \\ solve &= map \quad (fst \cdot max') \end{aligned}$$

Where $f \cdot g$ means function composing, that we first apply g then apply f. For instance function $h(x) = f(g(x))$ can be represented as $h = f \cdot g$, reading like function h is composed by f and g. Note that we use Curried form to omit the argument L for brevity. Informally speaking, If we feed a function which needs 2 arguments, for instance $f(x, y) = z$ with only 1 argument, the result turns to be a function which need 1 argument. For instance, if we feed f with

only argument x, it turns to be a new function take one argument y, defined as $g(y) = f(x, y)$, or $g = fx$. Note that x isn't a free variable any more, as it is bound to a value. Reader can refer to any book about functional programming for details about function composing and Currying.

Mapping can also be understood from the domain theory point of view. Consider function $y = f(x)$, it actually defines a mapping from domain of variable x to the domain of value y. (x and y can have different types). If the domains can be represented as set X, and Y, we have the following relation.

$$Y = \{f(x)|x \in X\} \tag{A.40}$$

This type of set definition is called Zermelo Frankel set abstraction [7]. The different is that here the mapping is from a list to another list, so there can be duplicated elements. In languages support list comprehension, for example Haskell and Python etc (Note that the Python list is a built-in type, but not the linked-list we discussed in this appendix), mapping can be implemented as a special case of list comprehension.

```
map f xs = [ f x | x ← xs]
```

List comprehension is a powerful tool. Here is another example that realizes the permutation algorithm in list comprehension. Many textbooks introduce how to implement all-permutation for a list, such as [7], and [9]. It's possible to design a more general version $perm(L, r)$, that if the length of the list L is N, this algorithm permutes r elements from the total N elements. We know that there are $P_N^r = \frac{N!}{(N-r)!}$ solutions.

$$perm(L, r) = \begin{cases} \{\Phi\} & : \quad r = 0 \vee |L| < r \\ \{\{l\} \cup P | l \in L, P \in perm(L - \{l\}, r - 1)\} & : \quad otherwise \end{cases} \tag{A.41}$$

In this equation, $\{l\} \cup P$ means $cons(l, P)$, and $L - \{l\}$ denotes $delete(L, l)$, which is defined in previous section. If we take zero element for permutation, or there are too few elements (less than r), the result is a list contains a empty list; Otherwise for non-trivial case, the algorithm picks one element l from the list, and recursively permutes the rest $N - 1$ elements by picking up $r - 1$ ones; then it puts all the possible l in front of all the possible $r - 1$ permutations. Here is the Haskell implementation of this algorithm.

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

We'll go back to the list comprehension later in section about filtering.

Mapping can also be realized imperatively. We can apply the function while traversing the list, and construct the new list from left to right. Since that the new element is appended to the result list, we can track the tail position to achieve constant time appending, so the mapping algorithms is linear in terms of the passed in function.

function MAP(f, L)
  $L' \leftarrow \Phi$
  $p \leftarrow \Phi$
  while $L \neq \Phi$ **do**

```
        if p = Φ then
            p ← Cons(f( First(L) ), Φ)
            L' ← p
        else
            Next(p) ← Cons(f( First(L) ), Φ)
            p ← Next(p)
        L ← Next(L)
    return L'
```

Because It's a bit complex to annotate the type of the passed-in function in ISO C++, as it involves some detailed language specific features. See [11] for detail. In fact ISO C++ provides the very same mapping concept as in `std::transform`. However, it needs the reader have knowledge of function object, iterator etc, which are out of the scope of this book. Reader can refer to any ISO C++ STL materials for detail.

For brevity purpose, we switch to Python programming language for example code. So that the type inference can be avoid in compile time. The definition of a simple singly linked-list in Python is give as the following.

```
class List:
    def __init__(self, x = None, xs = None):
        self.key = x
        self.next = xs

def cons(x, xs):
    return List(x, xs)
```

The mapping program, takes a function and a linked-list, and maps the functions to every element as described in above algorithm.

```
def mapL(f, xs):
    ys = prev = List()
    while xs is not None:
        prev.next = List(f(xs.key))
        prev = prev.next
        xs = xs.next
    return ys.next
```

Different from the pseudo code, this program uses a dummy node as the head of the resulting list. So it needn't test if the variable stores the last appending position is NIL. This small trick makes the program compact. We only need drop the dummy node before returning the result.

For each

For the trivial task such as printing a list of elements out, it's quite OK to just print each element without converting the whole list to a list of strings. We can actually simplify the program.

```
function Print(L)
    while L ≠ Φ do
        print First(L)
        L ← Rest(L)
```

More generally, we can pass a procedure such as printing, to this list traverse, so the procedure is performed *for each* element.

function FOR-EACH(L, P)
  while $L \neq \Phi$ **do**
    P(FIRST(L))
    $L \leftarrow$ REST(L)

For-each algorithm can be formalized in recursive approach as well.

$$foreach(L, p) = \begin{cases} u & : & L = \Phi \\ do(p(l_1), foreach(L', p)) & : & otherwise \end{cases} \tag{A.42}$$

Here u means unit, it's can be understood as doing nothing, The type of it is similar to the 'void' concept in C or java like programming languages. The $do()$ function evaluates all its arguments, discards all the results except for the last one, and returns the last result as the final value of $do()$. It's equivalent to `(begin ...)` in Lisp families, and `do` block in Haskell in some sense. For the details about unit type, please refer to [4].

Note that the for-each algorithm is just a simplified mapping, there are only two minor difference points:

- It needn't form a result list, we care the 'side effect' rather than the returned value;
- For each focus more on traversing, while mapping focus more on applying function, thus the order of arguments are typically arranged as $map(f, L)$ and $foreach(L, p)$.

Some Functional programming facilities provides options for both returning the result list or discarding it. For example Haskell Monad library provides both `mapM`, `mapM_` and `forM`, `forM_`. Readers can refer to language specific materials for detail.

Examples for mapping

We'll show how to use mapping by an example, which is a problem of ACM/ICPC[12]. For sake of brevity, we modified the problem description a bit. Suppose there are N lights in a room, all of them are off. We execute the following process N times:

1. We switch all the lights in the room, so that they are all on;
2. We switch the 2, 4, 6, ... lights, that every other light is switched, if the light is on, it will be off, and it will be on if the previous state is off;
3. We switch every third lights, that the 3, 6, 9, ... are switched;
4. ...

And at the last round, only the last light (the N-th light) is switched.

The question is how many lights are on finally?

Before we show the best answer to this puzzle, let's first work out a naive brute-force solution. Suppose there are N lights, which can be represented as a list of 0, 1 numbers, where 0 means the light is off, and 1 means on. The initial state is a list of N zeros: $\{0, 0, ..., 0\}$.

We can label the light from 1 to N. A mapping can help us to turn the above list into a labeled list[6].

$$map(\lambda_i \cdot (i, 0), \{1, 2, 3, ...N\})$$

This mapping augments each natural number with zero, the result is a list of pairs: $L = \{(1, 0), (2, 0), ..., (N, 0)\}$.

Next we operate this list of pairs N times from 1 to N. For every time i, we switch the second value in this pair if the first label can be divided by i. Consider the fact that $1 - 0 = 1$, and $1 - 1 = 0$, we can realize switching of 0, 1 value x by $1 - x$. At the i-th operation, for light (j, x), if $i|j$, (or $j \mod i = 0$), we then perform switching, otherwise, we leave the light untouched.

$$switch(i, (j, x)) = \begin{cases} (j, 1 - x) & : \quad j \mod i = 0 \\ (j, x) & : \quad otherwise \end{cases} \tag{A.43}$$

The i-th operation on all lights can be realized as mapping again:

$$map(switch(i), L) \tag{A.44}$$

Note that, here we use Curried form of $switch()$ function, which is equivalent to

$$map(\lambda_{(j,x)} \cdot switch(i, (j, x)), L)$$

Here we need define a function $proc()$, which can perform the above mapping on L over and over by N times. One option is to realize it in purely recursive way as the following, so that we can call it like $proc(\{1, 2, ..., N\}, L)$[7].

$$proc(I, L) = \begin{cases} L & : \quad I = \Phi \\ operate(I', map(switch(i_1), L)) & : \quad otherwise \end{cases} \tag{A.45}$$

Where $I = cons(i_1, I')$ if I isn't empty.

At this stage, we can sum the second value of each pair in list L to get the answer. The sum function has been defined in previous section, so the only thing left is mapping.

$$solve(N) = sum(map(snd, proc(\{1, 2, ..., N\}, L))) \tag{A.46}$$

Translating this naive brute-force solution to Haskell yields below program.

```
solve' = sum ∘ (map snd) ∘ proc  where
    proc n = operate [1..n] $ map (λi → (i, 0)) [1..n]
    operate [] xs = xs
    operate (i:is) xs = operate is (map (switch i) xs)
```

Let's see what's the answer for there are 1, 2, ..., 100 lights.

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

[6] Readers who are familiar with functional programming, may use zipping to achieve this. We'll explain zipping in later section.

[7] This can also be realized by folding, which will be explained in later section.

This result is interesting:

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

It seems that the i^2-th to the $((i+1)^2-1)$-th answers are i. Actually, we can prove this fact as the following.

Proof. Given N lights, labeled from 1 to N, consider which lights are on finally. Since the initial states for all lights are off, we can say that, the lights which are manipulated odd times are on. For every light i, it will be switched at the j round if i can be divided by j (denote as $j|i$). So only the lights which have odd number of factors are on at the end.

So the key point to solve this puzzle, is to find all numbers which have odd number of factors. For any positive integer N, denote S the set of all factors of N. S is initialized to Φ. if p is a factor of N, there must exist a positive integer q that $N = pq$, which means q is also a factor of N. So we add 2 different factors to the set S if and only if $p \neq q$, which keeps $|S|$ even all the time unless $p = q$. In such case, N is a perfect square number, and we can only add 1 factor to the set S, which leads to an odd number of factors. □

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

$$solve(N) = \lfloor \sqrt{N} \rfloor \tag{A.47}$$

The next Haskell command verifies that the answer for 1, 2, ..., 100 lights are as same as above.

```
map (floor.sqrt) [1..100]
[1,1,1,2,2,2,2,2,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,5,5,
6,6,6,6,6,6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,
8,8,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,10]
```

Mapping is generic concept that it doesn't only limit in linked-list, but also can be applied to many complex data structures. The chapter about binary search tree in this book explains how to map on trees. As long as we can traverse a data structure in some order, and the empty data structure can be identified, we can use the same mapping idea. We'll return to this kind of high-order concept in the section of folding later.

A.4.2 reverse

How to reverse a singly linked-list with minimum space is a popular technical interview problem in some companies. The pointer manipulation must be arranged carefully in imperative programming languages such as ANSI C. However, we'll show that, there exists an easy way to write this program:

1. Firstly, write a pure recursive straightforward solution;

2. Then, transform the pure recursive solution to tail-call manner;
3. Finally, translate the tail-call solution to pure imperative pointer operations.

The pure recursive solution is simple enough that we can write it out immediately. In order to *reverse a list* L.

- If L is empty, the reversed result is empty. This is the trivial edge case;
- Otherwise, we can first reverse the rest of the sub-list, then append the first element to the end.

This idea can be formalized to the below equation.

$$reverse(L) = \begin{cases} \Phi & : & L = \Phi \\ append(reverse(L'), l_1) & : & otherwise \end{cases} \tag{A.48}$$

Translating it to Haskell yields below program.

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

However, this solution doesn't perform well, as appending has to traverse to the end of list, which leads to a quadratic time algorithm. It's not hard to improve this program by changing it to tail-call manner. That we can use a accumulator to store the intermediate reversed result, and initialize the accumulated result as empty. So the algorithm is formalized as $reverse(L) = reverse'(L, \Phi)$.

$$reverse'(L, A) = \begin{cases} A & : & L = \Phi \\ reverse'(L', \{l_1\} \cup A) & : & otherwise \end{cases} \tag{A.49}$$

Where $\{l_1\} \cup A$ means $cons(l_1, A)$. Different from appending, it's a constant $O(1)$ time operation. The core idea is that we repeatedly take the element one by one from the head of the original list, and put them in front the accumulated result. This is just like we store all the elements in a stack, them pop them out. This is a linear time algorithm.

Below Haskell program implements this tail-call version.

```
reverse' [] acc = acc
reverse' (x:xs) acc = reverse' xs (x:acc)
```

Since the nature of tail-recursion call needn't book-keep any context (typically by stack), most modern compilers are able to optimize it to a pure imperative loop, and reuse the current context and stack etc. Let's manually do this optimization so that we can get a imperative algorithm.

function REVERSE(L)
  $A \leftarrow \Phi$
  while $L \neq \Phi$ **do**
    $A \leftarrow$ CONS(FIRST(L), A)
    $L \leftarrow$ REST(L)

However, because we translate it directly from a functional solution, this algorithm actually produces a new reversed list, but does not mutate the original one. It's not hard to change it to an in-place solution by reusing L. For example, the following ISO C++ program implements the in-place algorithm. It takes $O(1)$ memory space, and reverses the list in $O(N)$ time.

```
template<typename T>
List<T>* reverse(List<T>* xs) {
  List<T> *p, *ys = NULL;
  while (xs) {
    p = xs;
    xs = xs→next;
    p→next = ys;
    ys = p;
  }
  return ys;
}
```

Exercise A.2

- Implement the algorithm to find the maximum element in a list of pair in tail call approach in your favorite programming language.

A.5 Extract sub-lists

Different from arrays which are capable to slice a continuous segment fast and easily, It needs more work to extract sub lists from singly linked list. Such operations are typically linear algorithms.

A.5.1 take, drop, and split-at

Taking first N elements from a list is semantically similar to extract sub list from the very left like $sublist(L, 1, N)$, where the second and the third arguments to $sublist$ are the positions the sub-list starts and ends. For the trivial edge case, that either N is zero or the list is empty, the sub list is empty; Otherwise, we can recursively take the first $N - 1$ elements from the rest of the list, and put the first element in front of it.

$$take(N, L) = \begin{cases} \Phi & : \quad L = \Phi \vee N = 0 \\ cons(l_1, take(N - 1, L')) & : \quad otherwise \end{cases} \tag{A.50}$$

Note that the edge cases actually handle the out-of-bound error. The following Haskell program implements this algorithm.

```
take _ [] = []
take 0 _ = []
take n (x:xs) = x : take (n-1) xs
```

Dropping on the other hand, drops the first N elements and returns the left as result. It is equivalent to get the sub list from right like $sublist(L, N+1, |L|)$,

where $|L|$ is the length of the list. Dropping can be designed quite similar to taking by discarding the first element in the recursive case.

$$drop(N, L) = \begin{cases} \Phi & : & L = \Phi \\ L & : & N = 0 \\ drop(N-1, L') & : & otherwise \end{cases} \tag{A.51}$$

Translating the algorithm to Haskell gives the below example program.

```
drop _ [] = []
drop 0 L = L
drop n (x:xs) = drop (n-1) xs
```

The imperative taking and dropping are quite straight-forward, that they are left as exercises to the reader.

With taking and dropping defined, extracting sub list at arbitrary position for arbitrary length can be realized by calling them.

$$sublist(L, from, count) = take(count, drop(from - 1, L)) \tag{A.52}$$

or in another semantics by providing left and right boundaries:

$$sublist(L, from, to) = drop(from - 1, take(to, L)) \tag{A.53}$$

Note that the elements in range $[from, to]$ is returned by this function, with both ends included. All the above algorithms perform in linear time.

take-while and drop-while

Compare to taking and dropping, there is another type of operation, that we either keep taking or dropping elements as far as a certain condition is met. The taking and dropping algorithms can be viewed as special cases for take-while and drop-while.

Take-while examines elements one by one as far as the condition is satisfied, and ignore all the rest of elements even some of them satisfy the condition. This is the different point from filtering which we'll explained in later section. Take-while stops once the condition tests fail; while filtering traverses the whole list.

$$takeWhile(p, L) = \begin{cases} \Phi & : & L = \Phi \\ \Phi & : & \neg p(l_1) \\ cons(l_1, takeWhile(p, L')) & : & otherwise \end{cases} \tag{A.54}$$

Take-while accepts two arguments, one is the predicate function p, which can be applied to element in the list and returns Boolean value as result; the other argument is the list to be processed.

It's easy to define the drop-while symmetrically.

$$dropWhile(p, L) = \begin{cases} \Phi & : & L = \Phi \\ L & : & \neg p(l_1) \\ dropWhile(p, L') & : & otherwise \end{cases} \tag{A.55}$$

The corresponding Haskell example programs are given as below.

```
takeWhile _ [] = []
takeWhile p (x:xs) = if p x then x : takeWhile p xs else []

dropWhile _ [] = []
dropWhile p xs@(x:xs') = if p x then dropWhile p xs' else xs
```

split-at

With taking and dropping defined, splitting-at can be realized trivially by calling them.

$$splitAt(i, L) = (take(i, L), drop(i, L)) \tag{A.56}$$

A.5.2 breaking and grouping

breaking

Breaking can be considered as a general form of splitting, instead of splitting at a given position, breaking examines every element for a certain predicate, and finds the longest prefix of the list for that condition. The result is a pair of sub-lists, one is that longest prefix, the other is the rest.

There are two different breaking semantics, one is to pick elements satisfying the predicate as long as possible; the other is to pick those don't satisfy. The former is typically defined as *span*, while the later as *break*.

Span can be described, for example, in such recursive manner: In order to span a list L for predicate p:

- If the list is empty, the result for this edge trivial case is a pair of empty lists (Φ, Φ);
- Otherwise, we test the predicate against the first element l_1, if l_1 satisfies the predicate, we denote the intermediate result for spanning the rest of list as $(A, B) = span(p, L')$, then we put l_1 in front of A to get pair $(\{l_1\} \cup A, B)$, otherwise, we just return (Φ, L) as the result.

For breaking, we just test the negate of predicate and all the others are as same as spanning. Alternatively, one can define breaking by using span as in the later example program.

$$span(p, L) = \begin{cases} (\Phi, \Phi) & : & L = \Phi \\ (\{l_1\} \cup A, B) & : & p(l_1) = True, (A, B) = span(p, L') \\ (\Phi, L) & : & otherwise \end{cases} \tag{A.57}$$

$$break(p, L) = \begin{cases} (\Phi, \Phi) & : & L = \Phi \\ (\{l_1\} \cup A, B) & : & \neg p(l_1), (A, B) = break(p, L') \\ (\Phi, L) & : & otherwise \end{cases} \tag{A.58}$$

Note that both functions only find the longest *prefix*, they stop immediately when the condition is fail even if there are elements in the rest of the list meet the predicate (or not). Translating them to Haskell gives the following example program.

```
span _ [] = ([], [])
span p xs@(x:xs') = if p x then let (as, bs) = span xs' in (x:as, bs) else ([], xs)

break p = span (not ∘ p)
```

Span and break can also be realized imperatively as the following.

function SPAN(p, L)
  $A \leftarrow \Phi$
  while $L \neq \Phi \wedge p(l_1)$ **do**
    $A \leftarrow$ CONS(l_1, A)
    $L \leftarrow$ REST(L)
  return (A, L)

function BREAK(p, L)
  return SPAN($\neg p, L$)

This algorithm creates a new list to hold the longest prefix, another option is to turn it into in-place algorithm to reuse the spaces as in the following Python example.

```
def span(p, xs):
    ys = xs
    last = None
    while xs is not None and p(xs.key):
        last = xs
        xs = xs.next
    if last is None:
        return (None, xs)
    last.next = None
    return (ys, xs)
```

Note that both span and break need traverse the list to test the predicate, thus they are linear algorithms bound to $O(N)$.

grouping

Grouping is a commonly used operation to solve the problems that we need divide the list into some small groups. For example, Suppose we want to group the string 'Mississippi', which is actual a list of char { 'M', 's', 's', 'i', 's', 's', 'i', 'p', 'p', 'i'}. into several small lists in sequence, that each one contains consecutive identical characters. The grouping operation is expected to be:

```
group('Mississippi') = { 'M'', 'i', 'ss', 'i', 'ss', 'i', 'pp', 'i'}
```

Another example, is that we have a list of numbers:

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

We want to divide it into several small lists, that each sub-list is ordered descending. The grouping operation is expected to be :

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

Both cases play very important role in real algorithms. The string grouping is used in creating Trie/Patricia data structure, which is a powerful tool in string searching area; The ordered sub-list grouping can be used in nature merge sort. There are dedicated chapters in this book explain the detail of these algorithms.

It's obvious that we need abstract the grouping condition so that we know where to break the original list into small ones. This predicate can be passed to the algorithm as an argument like $group(p, L)$, where predicate p accepts two consecutive elements and test if the condition matches.

The first idea to solve the grouping problem is traversing – takes two elements at each time, if the predicate test succeeds, put both elements into a small group; otherwise, only put the first one into the group, and use the second one to initialize another new group. Denote the first two elements (if there are) are l_1, l_2, and the sub-list without the first element as L'. The result is a list of list $G = \{g_1, g_2, ...\}$, denoted as $G = group(p, L)$.

$$group(p, L) = \begin{cases} \{\Phi\} & : & L = \Phi \\ \{\{l_1\}\} & : & |L| = 1 \\ \{\{l_1\} \cup g'_1, g'_2, ...\} & : & p(l_1, l_2), G' = group(p, L') = \{g'_1, g'_2, ...\} \\ \{\{l_1\}, g'_1, g'_2, ...\} & : & otherwise \end{cases} \tag{A.59}$$

Note that $\{l_1\} \cup g'_1$ actually means $cons(l_1, g'_1)$, which performs in constant time. This is a linear algorithm performs proportion to the length of the list, it traverses the list in one pass which is bound to $O(N)$. Translating this program to Haskell gives the below example code.

```
group _ [] = [[]]
group _ [x] = [[x]]
group p (x:xs@(x':_)) | p x x' = (x:ys):yss
                      | otherwise = [x]:r
  where
    r@(ys:yss) = group p xs
```

It's possible to implement this algorithm in imperative approach, that we initialize the result groups as $\{l_1\}$ if L isn't empty, then we traverse the list from the second one, and append to the last group if the two consecutive elements satisfy the predicate; otherwise we start a new group.

function GROUP(p, L)
  if $L = \Phi$ **then**
    return $\{\Phi\}$
  $x \leftarrow$ FIRST(L)
  $L \leftarrow$ REST(L)
  $g \leftarrow \{x\}$
  $G \leftarrow \{g\}$
  while $L \neq \Phi$ **do**
    $y \leftarrow$ FIRST(L)
    if $p(x, y)$ **then**
      $g \leftarrow$ APPEND(g, y)
    else
      $g \leftarrow \{y\}$
      $G \leftarrow$ APPEND(G, g)

$x \leftarrow y$
$L \leftarrow \text{NEXT}(L)$
return G

However, different from the recursive algorithm, this program performs in quadratic time if the appending function isn't optimized by storing the tail position. The corresponding Python program is given as below.

```
def group(p, xs):
    if xs is None:
        return List(None)
    (x, xs) = (xs.key, xs.next)
    g = List(x)
    G = List(g)
    while xs is not None:
        y = xs.key
        if p(x, y):
            g = append(g, y)
        else:
            g = List(y)
            G = append(G, g)
        x = y
        xs = xs.next
    return G
```

With the grouping function defined, the two example cases mentioned at the beginning of this section can be realized by passing different predictions.

$$group(=, \{m, i, s, s, i, s, s, i, p, p, i\}) = \{\{M\}, \{i\}, \{ss\}, \{i\}, \{ss\}, \{i\}, \{pp\}, \{i\}\}$$

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

Another solution is to use the *span* function we have defined to realize grouping. We pass a predicate to span, which will break the list into two parts: The first part is the longest sub-list satisfying the condition. We can repeatedly apply the span with the same predication to the second part, till it becomes empty.

However, the predicate function we passed to span is *an unary function*, that it takes an element as argument, and test if it satisfies the condition. While in grouping algorithm, the predicate function is *a binary function*. It takes two adjacent elements for testing. The solution is that, we can use currying and pass the first element to the binary predicate, and use it to test the rest of elements.

$$group(p, L) = \begin{cases} \{\Phi\} & : & L = \Phi \\ \{\{l_1\} \cup A\} \cup group(p, B) & : & otherwise \end{cases} \tag{A.60}$$

Where $(A, B) = span(\lambda_x \cdot p(l_1, x), L')$ is the result of spanning on the rest sub-list of L.

Although this new defined grouping function can generate correct result for the first case as in the following Haskell code snippet.

```
groupBy (==) "Mississippi"
["m","i","ss","i","ss","i","pp","i"]
```

However, it seems that this algorithm can't group the list of numbers into ordered sub lists.

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

The reason is because that, as the first element 15 is used as the left parameter to $\geq$ operator for span, while 15 is the maximum value in this list, so the span function ends with putting all elements to A, and B is left empty. This might seem a defect, but it is actually the correct behavior if the semantic is to group equal elements together.

Strictly speaking, the equality predicate must satisfy three properties: reflexive, transitive, and symmetric. They are specified as the following.

- Reflexive. $x = x$, which says that any element is equal to itself;
- Transitive. $x = y, y = z \Rightarrow x = z$, which says that if two elements are equal, and one of them is equal to another, then all the tree are equal;
- Symmetric. $x = y \Leftrightarrow y = x$, which says that the order of comparing two equal elements doesn't affect the result.

When we group character list "Mississippi", the equal (=) operator is used, which obviously conforms these three properties. So that it generates correct grouping result. However, when passing ($\geq$) as equality predicate, to group a list of numbers, it violets both reflexive and symmetric properties, that is reason why we get wrong grouping result.

This fact means that the second algorithm we designed by using span, limits the semantic to strictly equality, while the first one does not. It just tests the condition for every two adjacent elements, which is much weaker than equality.

Exercise A.3

1. Implement the in-place imperative taking and dropping algorithms in your favorite programming language, note that the out of bound cases should be handled. Please try both languages with and without GC (Garbage Collection) support.
2. Implement take-while and drop-while in your favorite imperative programming language. Please try both dynamic type language and static type language (with and without type inference). How to specify the type of predicate function as generic as possible in static type system?
3. Consider the following definition of span.

$$span(p, L) = \begin{cases} (\Phi, \Phi) & : \quad L = \Phi \\ (\{l_1\} \cup A, B) & : \quad p(l_1) = True, (A, B) = span(p, L') \\ (A, \{l_1\} \cup B) & : \quad otherwise \end{cases}$$

What's the difference between this algorithm and the the one we've shown in this section?

4. Implement the grouping algorithm by using span in imperative way in your favorite programming language.

A.6 Folding

We are ready to introduce one of the most critical concept in high order programming, folding. It is so powerful tool that almost all the algorithms so far in this appendix can be realized by folding. Folding is sometimes be named as reducing (the abstracted concept is identical to the buzz term 'map-reduce' in cloud computing in some sense). For example, both STL and Python provide reduce function which realizes partial form of folding.

A.6.1 folding from right

Remind the sum and product definition in previous section, they are quite similar actually.

$$sum(L) = \begin{cases} 0 & : \quad L = \Phi \\ l_1 + sum(L') & : \quad otherwise \end{cases}$$

$$product(L) = \begin{cases} 1 & : \quad L = \Phi \\ l_1 \times product(L') & : \quad otherwise \end{cases}$$

It's obvious that they have same structure. What's more, if we list the insertion sort definition, we can find that it also shares this structure.

$$sort(L) = \begin{cases} \Phi & : \quad L = \Phi \\ insert(l_1, sort(L')) & : \quad otherwise \end{cases}$$

This hint us that we can abstract this essential common structure, so that we needn't repeat it again and again. Observing *sum*, *product*, and *sort*, there are two different points which we can parameterize.

- The result of the trivial edge case varies. It's zero for sum, 1 for product, and empty list for sorting.
- The function applied to the first element and the intermediate result varies. It's plus for sum, multiply for product, and ordered-insertion for sorting.

If we parameterize the result of trivial edge case as initial value z (stands for abstract zero concept), the function applied in recursive case as f (which takes two parameters, one is the first element in the list, the other is the recursive result for the rest of the list), this common structure can be defined as something like the following.

$$proc(f, z, L) = \begin{cases} z & : \quad L = \Phi \\ f(l_1, proc(f, z, L')) & : \quad otherwise \end{cases}$$

That's it, and we should name this common structure a better name instead of the meaningless 'proc'. Let's see the characteristic of this common structure. For list $L = \{x_1, x_2, ..., x_N\}$, we can expand the computation like the following.

$$\begin{aligned} proc(f, z, L) &= f(x_1, proc(f, z, L') \\ &= f(x_1, f(x_2, proc(f, z, L'')) \\ &... \\ &= f(x_1, f(x_2, f(..., f(x_N, f(f, z, \Phi))...) \\ &= f(x_1, f(x_2, f(..., f(x_N, z))...) \end{aligned}$$

Since f takes two parameters, it's a binary function, thus we can write it in infix form. The infix form is defined as below.

$$x \oplus_f y = f(x, y) \tag{A.61}$$

The above expanded result is equivalent to the following by using infix notion.

$$proc(f, z, L) = x_1 \oplus_f (x_2 \oplus_f (...(x_N \oplus_f z))...)$$

Note that the parentheses are necessary, because the computation starts from the right-most $(x_N \oplus_f z)$, and repeatedly fold to left towards x_1. This is quite similar to folding a Chinese hand-fan as illustrated in the following photos. A Chinese hand-fan is made of bamboo and paper. Multiple bamboo frames are stuck together with an axis at one end. The arc shape paper is fully expanded by these frames as shown in Figure A.3 (a); The fan can be closed by folding the paper. Figure A.3 (b) shows that some part of the fan is folded from right. After these folding finished, the fan results a stick, as shown in Figure A.3 (c).

We can considered that each bamboo frame along with the paper on it as an element, so these frames forms a list. A unit process to close the fan is to rotate a frame for a certain angle, so that it lays on top of the collapsed part. When we start closing the fan, the initial collapsed result is the first bamboo frame. The close process is folding from one end, and repeatedly apply the unit close steps, till all the frames is rotated, and the folding result is a stick closed form.

Actually, the sum and product algorithms exactly do the same thing as closing the fan.

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

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

In functional programming, we name this process *folding*, and particularly, since we execute from the most inner structure, which starts from the right-most one. This type of folding is named *folding right*.

$$foldr(f, z, L) = \begin{cases} z & : \quad L = \Phi \\ f(l_1, foldr(f, z, L')) & : \quad otherwise \end{cases} \tag{A.62}$$

(a) A folding fan fully opened.

(b) The fan is partly folded on right.

(c) The fan is fully folded, closed to a stick.

Figure A.3: Folding a Chinese hand-fan

Let's see how to use fold-right to realize sum and product.

$$\begin{aligned}\sum_{i=1}^{N} x_i &= x_1 + (x_2 + (x_3 + ... + (x_{N_1} + x_N))...) \\ &= foldr(+, 0, \{x_1, x_2, ..., x_N\})\end{aligned} \tag{A.63}$$

$$\begin{aligned}\prod_{i=1}^{N} x_i &= x_1 \times (x_2 \times (x_3 \times ... + (x_{N_1} \times x_N))...) \\ &= foldr(\times, 1, \{x_1, x_2, ..., x_N\})\end{aligned} \tag{A.64}$$

The insertion-sort algorithm can also be defined by using folding right.

$$sort(L) = foldr(insert, \Phi, L) \tag{A.65}$$

A.6.2 folding from left

As mentioned in section of 'tail recursive' call. Both pure recursive sum and product compute from right to left and they must book keep all the intermediate results and contexts. As we abstract fold-right from the very same structure, folding from right does the book keeping as well. This will be expensive if the list is very long.

Since we can change the realization of sum and product to tail-recursive call manner, it quite possible that we can provide another folding algorithm, which processes the list from left to right in normal order, and enable the tail-call optimization by reusing the same context.

Instead of induction from sum, product and insertion, we can directly change the folding right to tail call. Observe that the initial value z, actually represents the intermediate result at any time. We can use it as the accumulator.

$$foldl(f, z, L) = \begin{cases} z & : \quad L = \Phi \\ foldl(f, f(z, l_1), L') & : \quad otherwise \end{cases} \tag{A.66}$$

Every time when the list isn't empty, we take the first element, apply function f on the accumulator z and it to get a new accumulator $z' = f(z, l_1)$. After that we can repeatedly folding with the very same function f, the updated accumulator z', and list L'.

Let's verify that this tail-call algorithm actually folding from left.

$$\begin{aligned}\sum_{i=1}^{5} i &= foldl(+, 0, \{1, 2, 3, 4, 5\}) \\ &= foldl(+, 0 + 1, \{2, 3, 4, 5\}) \\ &= foldl(+, (0 + 1) + 2\{3, 4, 5\} \\ &= foldl(+, ((0 + 1) + 2) + 3, \{4, 5\}) \\ &= foldl(+, (((0 + 1) + 2) + 3) + 4, \{5\}) \\ &= foldl(+, ((((0 + 1) + 2 + 3) + 4 + 5, \Phi) \\ &= 0 + 1 + 2 + 3 + 4 + 5\end{aligned}$$

Note that, we actually delayed the evaluation of $f(z, l_1)$ in every step. (This is the exact behavior in system support lazy-evaluation, for instance, Haskell. However, in strict system such as standard ML, it's not the case.) Actually, they will be evaluated in sequence of $\{1, 3, 6, 10, 15\}$ in each call.

Generally, folding-left can be expanded in form of

$$foldl(f, z, L) = f(f(...(f(f(z, l_1), l_2), ..., l_N) \tag{A.67}$$

Or in infix manner as

$$foldl(f, z, L) = ((...(z \oplus_f l_1) \oplus_f l_2) \oplus_f ...) \oplus l_N \tag{A.68}$$

With folding from left defined, sum, product, and insertion-sort can be transparently implemented by calling $foldl$ as $sum(L) = foldl(+, 0, L)$, $product(L) = foldl(+, 1, L)$, and $sort(L) = foldl(insert, \Phi, L)$. Compare with the folding-right version, they are almost same at first glares, however, the internal implementation differs.

Imperative folding and generic folding concept

The tail-call nature of folding-left algorithm is quite friendly for imperative settings, that even the compiler isn't equipped with tail-call recursive optimization, we can anyway implement the folding in while-loop manually.

function FOLD(f, z, L)
  while $L \neq \Phi$ **do**
    $z \leftarrow f(z,$ FIRST(L))
    $L \leftarrow$ REST(L)
  return z

Translating this algorithm to Python yields the following example program.

```
def fold(f, z, xs):
    for x in xs:
        z = f(z, x)
    return z
```

Actually, Python provides built-in function 'reduce' which does the very same thing. (in ISO C++, this is provided as reduce algorithm in STL.) Almost no imperative environment provides folding-right function because it will cause stack overflow problem if the list is too long. However, there still exist cases that the folding from right semantics is necessary. For example, one defines a container, which only provides insertion function to the head of the container, but there is no any appending method, so that we want such a $fromList$ tool.

$$fromList(L) = foldr(insertHead, empty, L)$$

Calling $fromList$ with the insertion function as well as an empty initialized container, can turn a list into the special container. Actually the singly linked-list is such a container, which performs well on insertion to the head, but poor to linear time if appending on the tail. Folding from right is quite nature when duplicate a linked-list while keeps the elements ordering. While folding from left will generate a reversed list.

In such cases, there exists an alternative way to implement imperative folding right by first reverse the list, and then folding the reversed one from left.

function FOLD-RIGHT(f, z, L)
  return FOLD($f, z,$ REVERSE(L))

Note that, here we must use the tail-call version of reversing, or the stack overflow issue still exists.

One may think that folding-left should be chosen in most cases over folding-right because it's friendly for tail-recursion call optimization, suitable for both

functional and imperative settings, and it's an online algorithm. However, folding-right plays a critical role when the input list is infinity and the binary function f is lazy. For example, below Haskell program wraps every element in an infinity list to a singleton, and returns the first 10 result.

```
take 10 $ foldr (λx xs →[x]:xs) [] [1..]
[[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]
```

This can't be achieved by using folding left because the outer most evaluation can't be finished until all the list being processed. The details is specific to lazy evaluation feature, which is out of the scope of this book. Readers can refer to [13] for details.

Although the main topic of this appendix is about singly linked-list related algorithms, the folding concept itself is generic which doesn't only limit to list, but also can be applied to other data structures.

We can fold a tree, a queue, or even more complicated data structures as long as we have the following:

- The empty data structure can be identified for trivial edge case; (e.g. empty tree)
- We can traverse the data structure (e.g. traverse the tree in pre-order).

Some languages provide this high-level concept support, for example, Haskell achieve this via *monoid*, readers can refer to [8] for detail.

There are many chapters in this book use the widen concept of folding.

A.6.3 folding in practice

We have seen that *max*, *min*, and insertion sort all can be realized in folding. The brute-force solution for 'drunk jailer' puzzle shown in mapping section can also be designed by mixed use of mapping and folding.

Remind that we create a list of pairs, each pair contains the number of the light, and the on-off state. After that we process from 1 to N, switch the light if the number can be divided. The whole process can be viewed as folding.

$$fold(step, \{(1,0),(2,0),...,(N,0)\}, \{1,2,...,N\})$$

The initial value is the very first state, that all the lights are off. The list to be folding is the operations from 1 to N. Function *step* takes two arguments, one is the light states pair list, the other is the operation time i. It then maps on all lights and performs switching. We can then substitute the *step* with mapping.

$$fold(\lambda_{L,i} \cdot map(switch(i), L), \{(1,0),(2,0),...,(N,0)\}, \{1,2,...,N\})$$

We'll simplify the λ notation, and directly write $map(switch(i), l)$ for brevity purpose. The result of this folding is the final states pairs, we need take the second one of the pair for each element via mapping, then calculate the summation.

$$sum(map(snd, fold(map(switch(i), L), \{(1,0),(2,0),...,(N,0)\}, \{1,2,...,N\}))) \tag{A.69}$$

There are materials provides plenty of good examples of using folding, especially in [1], folding together with fusion law are well explained.

concatenate a list of list

In previous section A.3.6 about concatenation, we explained how to concatenate two lists. Actually, concatenation of lists can be considered equivalent to summation of numbers. Thus we can design a general algorithm, which can concatenate multiple lists into one big list.

What's more, we can realize this general concatenation by using folding. As sum can be represented as $sum(L) = foldr(+, 0, L)$, it's straightforward to write the following equation.

$$concats(L) = foldr(concat, \Phi, L) \tag{A.70}$$

Where L is a list of list, for example $\{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}, ...\}$. Function $concat(L_1, L_2)$ is what we defined in section A.3.6.

In some environments which support lazy-evaluation, such as Haskell, this algorithm is capable to concatenate infinite list of list, as the binary function `++` is lazy.

Exercise A.4

- What's the performance of *concats* algorithm? is it linear or quadratic?
- Design another linear time *concats* algorithm without using folding.
- Realize mapping algorithm by using folding.

A.7 Searching and matching

Searching and matching are very important algorithms. They are not only limited to linked list, but also applicable to a wide range of data structures. We just scratch the surface of searching and matching in this appendix. There are dedicated chapters explain about them in this book.

A.7.1 Existence testing

The simplest searching case is to test if a given element exists in a list. A linear time traverse can solve this problem. In order to determine element x exists in list L:

- If the list is empty, it's obvious that the element doesn't exist in L;
- If the first element in the list equals to x, we know that x exists;
- Otherwise, we need recursively test if x exists in the rest sub-list L';

This simple description can be directly formalized to equation as the following.

$$x \in L = \begin{cases} False & : & L = \Phi \\ True & : & l_1 = x \\ x \in L' & : & otherwise \end{cases} \tag{A.71}$$

This is definitely a linear algorithm which is bound to $O(N)$ time. The best case happens in the two trivial clauses that either the list is empty or the first element is what we are finding; The worst case happens when the element doesn't exist at all or it is the last element. In both cases, we need traverse the whole list. If the probability is equal for all the positions, the average case takes about $\frac{N+1}{2}$ steps for traversing.

This algorithm is so trivial that we left the implementation as exercise to the reader. If the list is ordered, one may expect to improve the algorithm to logarithm time but not linear. However, as we discussed, since list doesn't support constant time random accessing, binary search can't be applied here. There is a dedicated chapter in this book discusses how to evolve the linked list to binary tree to achieve quick searching.

A.7.2 Looking up

One extra step from existence testing is to find the interesting information stored in the list. There are two typical methods to augment extra data to the element. Since the linked list is chain of nodes, we can store satellite data in the node, then provide $key(n)$ to access the key of the node, $rest(n)$ for the rest sub-list, and $value(n)$ for the augmented data. The other method, is to pair the key and data, for example $\{(1, hello), (2, world), (3, foo), ...\}$. We'll introduce how to form such pairing list in later section.

The algorithm is almost as same as the existence testing, that it traverses the list, examines the key one by one. Whenever it finds a node which has the same key as what we are looking up, it stops, and returns the augmented data. It's obvious that this is linear strategy. If the satellite data is augmented to the node directly, the algorithm can be defined as the following.

$$lookup(x, L) = \begin{cases} \Phi & : & L = \Phi \\ value(l_1) & : & key(l_1) = x \\ lookup(x, L') & : & otherwise \end{cases} \tag{A.72}$$

In this algorithm, L is a list of nodes which are augmented with satellite data. Note that the first case actually means looking up failure, so that the result is empty. Some functional programming languages, such as Haskell, provide `Maybe` type to handle the possibility of fail. This algorithm can be slightly modified to handle the key-value pair list as well.

$$lookup(x, L) = \begin{cases} \Phi & : & L = \Phi \\ snd(l_1) & : & fst(l_1) = x \\ lookup(x, L') & : & otherwise \end{cases} \tag{A.73}$$

Here L is a list of pairs, functions $fst(p)$ and $snd(p)$ access the first part and second part of the pair respectively.

Both algorithms are in tail-call manner, they can be transformed to imperative looping easily. We left this as exercise to the reader.

A.7.3 finding and filtering

Let's take one more step ahead, looking up algorithm performs linear search by comparing the key of an element is equal to the given value. A more general case is to find an element matching a certain predicate. We can abstract this matching condition as a parameter for this generic linear finding algorithm.

$$find(p, L) = \begin{cases} \Phi & : & L = \Phi \\ l_1 & : & p(l_1) \\ find(p, L') & : & otherwise \end{cases} \tag{A.74}$$

The algorithm traverses the list by examining if the element satisfies the predicate p. It fails if the list is empty while there is still nothing found. This is handled in the first trivial edge case; If the first element in the list satisfies the condition, the algorithm returns the whole element (node), and user can further handle it as he like (either extract the satellite data or do whatever); otherwise, the algorithm recursively perform finding on the rest of the sub-list. Below is the corresponding Haskell example program.

```
find _ [] = Nothing
find p (x:xs) = if p x then Just x else find p xs
```

Translating this to imperative algorithm is straightforward. Here we use 'NIL' to represent the fail case.

function FIND(p, L)
  while $L \neq \Phi$ **do**
    if p(FIRST(L)) **then**
      return FIRST(L)
    $L \leftarrow$ REST(L)
  return NIL

And here is the Python example of finding.

```
def find(p, xs):
    while xs is not None:
        if p(xs.key):
            return xs
        xs = xs.next
    return None
```

It's quite possible that there are multiple elements in the list which satisfy the precondition. The finding algorithm designed so far just picks the first one it meets, and stops immediately. It can be considered as a special case of finding all elements under a certain condition.

Another viewpoint of finding all elements with a given predicate is to treat the finding algorithm as a black box, the input to this box is a list, while the output is another list contains all elements satisfying the predicate. This can be called as filtering as shown in the below figure.

This figure can be formalized in another form in taste of set enumeration. However, we actually enumerate among list instead of a set.

$$filter(p, L) = \{x | x \in L \wedge p(x)\} \tag{A.75}$$

Some environment such as Haskell (and Python for any iterable), supports this form as list comprehension.

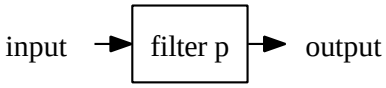


Figure A.4: The input is the original list $\{x_1, x_2, ..., x_N\}$, the output is a list $\{x'_1, x'_2, ..., x'_M\}$, that for $\forall x'_i$, predicate $p(x'_i)$ is satisfied.

```
filter p xs = [ x | x ← xs, p x]
```

And in Python for built-in list as

```
def filter(p, xs):
    return [x for x in xs if p(x)]
```

Note that the Python built-in list isn't singly-linked list as we mentioned in this appendix.

In order to modify the finding algorithm to realize filtering, the found elements are appended to a result list. And instead of stopping the traverse, all the rest of elements should be examined with the predicate.

$$filter(p, L) = \begin{cases} \Phi & : \quad L = \Phi \\ cons(l_1, filter(p, L')) & : \quad p(l_1) \\ filter(p, L') & : \quad otherwise \end{cases} \tag{A.76}$$

This algorithm returns empty result if the list is empty for trivial edge case; For non-empty list, suppose the recursive result of filtering the rest of the sub-list is A, the algorithm examine if the first element satisfies the predicate, it is put in front of A by a 'cons' operation ($O(1)$ time).

The corresponding Haskell program is given as below.

```
filter _ [] = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

Although we mentioned that the next found element is 'appended' to the result list, this algorithm actually constructs the result list from the right most to the left, so that appending is avoided, which ensure the linear $O(N)$ performance. Compare this algorithm with the following imperative quadratic realization reveals the difference.

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

As the comment of appending statement, it's typically proportion to the length of the result list if the tail position isn't memorized. This fact indicates that directly transforming the recursive filter algorithm into tail-call form will downgrade the performance from $O(N)$ to $O(N^2)$. As shown in the below equation, that $filter(p, L) = filter'(p, L, \Phi)$ performs as poorly as the imperative

one.

$$filter'(p, L, A) = \begin{cases} A & : & L = \Phi \\ filter'(p, L', A \cup \{l_1\}) & : & p(l_1) \\ filter'(p, L', A) & : & otherwise \end{cases} \tag{A.77}$$

One solution to achieve linear time performance imperatively is to construct the result list in reverse order, and perform the $O(N)$ reversion again (refer to the above section) to get the final result. This is left as exercise to the reader.

The fact of construction the result list from right to left indicates the possibility of realizing filtering with folding-right concept. We need design some combinator function f, so that $filter(p, L) = foldr(f, \Phi, L)$. It requires that function f takes two arguments, one is the element iterated among the list; the other is the intermediate result constructed from right. $f(x, A)$ can be defined as that it tests the predicate against x, if succeed, the result is updated to $cons(x, A)$, otherwise, A is kept same.

$$f(x, A) = \begin{cases} cons(x, A) & : & p(x) \\ A & : & otherwise \end{cases} \tag{A.78}$$

However, the predicate must be passed to function f as well. This can be achieved by using currying, so f actually has the prototype $f(p, x, A)$, and filtering is defined as following.

$$filter(p, L) = foldr(\lambda_{x,A} \cdot f(p, x, A), \Phi, L) \tag{A.79}$$

Which can be simplified by η-conversion. For detailed definition of η-conversion, readers can refer to [2].

$$filter(p, L) = foldr(f(p), \Phi, L) \tag{A.80}$$

The following Haskell example program implements this equation.

```
filter p = foldr f [] where
    f x xs = if p x then x : xs else xs
```

Similar to mapping and folding, filtering is actually a generic concept, that we can apply a predicate on any traversable data structures to get what we are interesting. readers can refer to the topic about monoid in [8] for further reading.

A.7.4 Matching

Matching generally means to find a given pattern among some data structures. In this section, we limit the topic within list. Even this limitation will leads to a very wide and deep topic, that there are dedicated chapters in this book introduce matching algorithms. So we only select the algorithm to test if a given list exists in another (typically longer) list.

Before dive into the algorithm of finding the sub-list at any position, two special edge cases are used for warm up. They are algorithms to test if a given list is either prefix or suffix of another.

In the section about span, we have seen how to find a prefix under a certain condition. prefix matching can be considered as a special case in some sense.

That it compares each of the elements between the two lists from the beginning until meets any different elements or pass the end of one list. Define $P \subseteq L$ if P is prefix of L.

$$P \subseteq L = \begin{cases} True & : & P = \Phi \\ False & : & p_1 \neq l_1 \\ P' \subseteq L' & : & otherwise \end{cases} \tag{A.81}$$

This is obviously a linear algorithm. However, We can't use the very same approach to test if a list is suffix of another because it isn't cheap to start from the end of the list and keep iterating backwards. Arrays, on the other hand which support random access can be easily traversed backwards.

As we only need the yes-no result, one solution to realize a linear suffix testing algorithm is to reverse both lists, (which is linear time), and use prefix testing instead. Define $L \supseteq P$ if P is suffix of L.

$$L \supseteq P = reverse(P) \subseteq reverse(L) \tag{A.82}$$

With $\subseteq$ defined, it enables to test if a list is infix of another. The idea is to traverse the target list, and repeatedly applying the prefix testing till any success or arrives at the end.

```
function Is-Infix(P, L)
    while L ≠ Φ do
        if P ⊆ L then
            return TRUE
        L ← Rest(L)
    return FALSE
```

Formalize this algorithm to recursive equation leads to the below definition.

$$infix?(P, L) = \begin{cases} True & : & P \subseteq L \\ False & : & L = \Phi \\ infix?(P, L') & : & otherwise \end{cases} \tag{A.83}$$

Note that there is a tricky implicit constraint in this equation. If the pattern P is empty, it is definitely the infix of any target list. This case is actually covered by the first condition in the above equation because empty list is also the prefix of any list. In most programming languages support pattern matching, we can't arrange the second clause as the first edge case, or it will return false for $infix?(\Phi, \Phi)$. (One exception is Prolog, but this is a language specific feature, which we won't covered in this book.)

Since prefix testing is linear, and it is called while traversing the list, this algorithm is quadratic $O(N * M)$. where N and M are the length of the pattern and target lists respectively. There is no trivial way to improve this 'position by position' scanning algorithm to linear even if the data structure changes from linked-list to randomly accessible array.

There are chapters in this book introduce several approaches for fast matching, including suffix tree with Ukkonen algorithm, Knuth-Morris-Pratt algorithm and Boyer-Moore algorithm.

Alternatively, we can enumerate all suffixes of the target list, and check if the pattern is prefix of any these suffixes. Which can be represented as the

following.

$$infix?(P, L) = \exists S \in suffixes(L) \wedge P \subseteq S \tag{A.84}$$

This can be represented as list comprehension, for example the below Haskell program.

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

Where function `isPrefixOf` is the prefixing testing function defined according to our previous design. function `tails` generate all suffixes of a list. The implementation of `tails` is left as an exercise to the reader.

Exercise A.5

- Implement the linear existence testing in both functional and imperative approaches in your favorite programming languages.
- Implement the looking up algorithm in your favorite imperative programming language.
- Realize the linear time filtering algorithm by firstly building the result list in reverse order, and finally reverse it to resume the normal result. Implement this algorithm in both imperative looping and functional tail-recursion call.
- Implement the imperative algorithm of prefix testing in your favorite programming language.
- Implement the algorithm to enumerate all suffixes of a list.

A.8 zipping and unzipping

It's quite common to construct a list of paired elements. For example, in the naive brute-force solution for 'Drunk jailer' puzzle which is shown in section of mapping, we need to represent the state of all lights. It is initialized as $\{(1, 0), (2, 0), ..., (N, 0)\}$. Another example is to build a key-value list, such as $\{(1, a), (2, an), (3, another), ...\}$.

In 'Drunk jailer' example, the list of pairs is built like the following.

$$map(\lambda_i \cdot (i, 0), \{1, 2, ..., N\})$$

The more general case is that, There have been already two lists prepared, what we need is a handy 'zipper' method.

$$zip(A, B) = \begin{cases} \Phi & : \quad A = \Phi \vee B = \Phi \\ cons((a_1, b_1), zip(A', B')) & : \quad otherwise \end{cases} \tag{A.85}$$

Note that this algorithm is capable to handle the case that the two lists being zipped have different lengths. The result list of pairs aligns with the shorter one. And it's even possible to zip an infinite list with another one with

limited length in environment support lazy evaluation. For example with this auxiliary function defined, we can initialize the lights state as

$$zip(\{0, 0, ...\}, \{1, 2, ..., N\}$$

In some languages support list enumeration, such as Haskell (Python provides similar `range` function, but it manipulates built-in list, which isn't linked-list actually), this can be expressed as `zip (repeat 0) [1..n]`. Given a list of words, we can also index them with consecutive numbers as

$$zip(\{1, 2, ...\}, \{a, an, another, ...\})$$

Note that the zipping algorithm is linear, as it uses constant time 'cons' operation in each recursive call. However, directly translating *zip* into imperative manner would down-grade the performance to quadratic unless the linked-list is optimized with tail position cache or we in-place modify one of the passed-in list.

function ZIP(A, B)
  $C \leftarrow \Phi$
  while $A \neq \Phi \wedge B \neq \Phi$ **do**
    $C \leftarrow$ APPEND(C, (FIRST(A), FIRST(B)))
    $A \leftarrow$ REST(A)
    $B \leftarrow$ REST(B)
  return C

Note that, the appending operation is proportion to the length of the result list C, so it will get more and more slowly along with traversing. There are three solutions to improve this algorithm to linear time. The first method is to use a similar approach as we did in infix-testing, that we construct the result list of pairs in reverse order by always insert the paired elements on head; then perform a linear reverse operation before return the final result; The second method is to modify one passed-in list, for example A, in-place while traversing. Translate it from list of elements to list of pairs; The third method is to remember the last appending position. Please try these solutions as exercise.

The key point of linear time zipping is that the result list is actually built from right to left, which is similar to the infix-testing algorithm. So it's quite possible to provide a folding-right realization. This is left as exercise to the reader.

It's natural to extend the zipper algorithm so that multiple lists can be zipped to one list of multiple-elements. For example, Haskell standard library provides, `zip`, `zip3`, `zip4`, ..., till `zip7`. Another typical extension to zipper is that, sometimes, we don't want to list of pairs (or tuples more generally), instead, we want to apply some combinator function to each pair of elements.

For example, consider the case that we have a list of unit prices for every fruit: apple, orange, banana, ..., as $\{1.00, 0.80, 10.05, ...\}$, with same unit of Dollar; And the cart of customer holds a list of purchased quantity, for instance $\{3, 1, 0, ...\}$, means this customer, put 3 apples, an orange in the cart. He doesn't take any banana, so the quantity of banana is zero. We want to generate a list of cost for the customer, contains how much should pay for apple, orange, banana,... respectively.

The program can be written from scratch as below.

$$paylist(U, Q) = \begin{cases} \Phi & : \quad U = \Phi \vee Q = \Phi \\ cons(u_1 \times q_1, paylist(U', Q')) & : \quad otherwise \end{cases}$$

Compare this equation with the zipper algorithm. It's easy to find the common structure of the two, and we can parameterize the combinator function as f, so that the 'generic' zipper algorithm can be defined as the following.

$$zipWith(f, A, B) = \begin{cases} \Phi & : \quad A = \Phi \vee B = \Phi \\ cons(f(a_1, b_1), zipWith(f, A', B')) & : \quad otherwise \end{cases} \tag{A.86}$$

Here is an example that defines the inner-product (or dot-product)[14] by using $zipWith$.

$$A \cdot B = sum(zipWith(\times, A, B)) \tag{A.87}$$

It's necessary to realize the inverse operation of zipping, that converts a list of pairs, to different lists of elements. Back to the purchasing example, It's quite possible that the unit price information is stored in a association list like $U = \{(apple, 1.00), (orange, 0.80), (banana, 10.05), ...\}$, so that it's convenient to look up the price with a given product name, for instance, $lookup(melon, U)$. Similarly, the cart can also be represented clearly in such manner, for example, $Q = \{(apple, 3), (orange, 1), (banana, 0), ...\}$.

Given such a 'product - unit price' list and a 'product - quantity' list, how to calculate the total payment?

One straight forward idea derived from the previous solution is to extract the unit price list and the purchased quantity list, then calculate the inner-product of them.

$$pay = sum(zipWith(\times, snd(unzip(P)), snd(unzip(Q)))) \tag{A.88}$$

Although the definition of $unzip$ can be directly written as the inverse of zip, here we give a realization based on folding-right.

$$unzip(L) = foldr(\lambda_{(a,b),(A,B)} \cdot (cons(a, A), cons(b, B)), (\Phi, \Phi), L) \tag{A.89}$$

The initial result is a pair of empty list. During the folding process, the head of the list, which is a pair of elements, as well as the intermediate result are passed to the combinator function. This combinator function is given as a lambda expression, that it extracts the paired elements, and put them in front of the two intermediate lists respectively. Note that we use implicit pattern matching to extract the elements from pairs. Alternatively this can be done by using fst, and snd functions explicitly as

$$\lambda_{p,P} \cdot (cons(fst(p), fst(P)), cons(snd(p), snd(P)))$$

The following Haskell example code implements $unzip$ algorithm.

```
unzip = foldr λ(a, b) (as, bs) → (a:as, b:bs) ([], [])
```

Zip and unzip concepts can be extended more generally rather than only limiting within linked-list. It's quite useful to zip two lists to a tree, where the data stored in the tree are paired elements from both lists. General zip and unzip can also be used to track the traverse path of a collection to mimic the 'parent' pointer in imperative implementations. Please refer to the last chapter of [8] for a good treatment.

Exercise A.6

- Design and implement iota (I) algorithm, which can enumerate a list with some given parameters. For example:
  - $iota(..., N) = \{1, 2, 3, ..., N\}$;
  - $iota(M, N) = \{M, M + 1, M + 2, ..., N\}$, Where $M \leq N$;
  - $iota(M, M + a, ..., N) = \{M, M + a, M + 2a, ..., N\}$;
  - $iota(M, M, ...) = repeat(M) = \{M, M, M, ...\}$;
  - $iota(M, ...) = \{M, M + 1, M + 2, ...\}$.

  Note that the last two cases demand generate infinite list essentially. Consider how to represents infinite list? You may refer to the streaming and lazy evaluation materials such as [5] and [8].
- Design and implement a linear time imperative zipper algorithm.
- Realize the zipper algorithm with folding-right approach.
- For the purchase payment example, suppose the quantity association list only contains those items with the quantity isn't zero, that instead of a list of $Q = \{(apple, 3), (banana, 0), (orange, 1), ...\}$, it hold a list like $Q = \{(apple, 3), (orange, 1), ...\}$. The 'banana' information is filtered because the customer doesn't pick any bananas. Write a program, taking the unit-price association list, and this kind of quantity list, to calculate the total payment.

A.9 Notes and short summary

In this appendix, a quick introduction about how to build, manipulate, transfer, and searching singly linked list is briefed in both purely functional and imperative approaches. Most of the modern programming environments have been equipped with tools to handle such elementary data structures. However, such tools are designed for general purpose cases, Serious programming shouldn't take them as black-boxes.

Since linked-list is so critical that it builds the corner stones for almost all functional programming environments, just like the importance of array to imperative settings. We take this topic as an appendix to the book. It's quite OK that the reader starts with the first chapter about binary search tree, which is a kind of 'hello world' topic, and refers to this appendix when meets any unfamiliar list operations.

Bibliography

[1] Richard Bird. “Pearls of Functional Algorithm Design”. Cambridge University Press; 1 edition (November 1, 2010). ISBN: 978-0521513388

[2] Simon L. Peyton Jones. “The Implementation of Functional Programming Languages”. Prentice-Hall International Series in Computer Since. Prentice Hall (May 1987). ISBN: 978-0134533339

[3] Andrei Alexandrescu. “Modern C++ design: Generic Programming and Design Patterns Applied”. Addison Wesley February 01, 2001, ISBN 0-201-70431-5

[4] Benjamin C. Pierce. “Types and Programming Languages”. The MIT Press, 2002. ISBN:0262162091

[5] Harold Abelson, Gerald Jay Sussman, Julie Sussman. “Structure and Interpretation of Computer Programs, 2nd Edition”. MIT Press, 1996, ISBN 0-262-51087-1

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

[7] Fethi Rabhi, Guy Lapalme. “Algorithms: a functional programming approach”. Second edition. Addison-Wesley, 1999. ISBN: 0201-59604-0

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

[9] Joe Armstrong. “Programming Erlang: Software for a Concurrent World”. Pragmatic Bookshelf; 1 edition (July 18, 2007). ISBN-13: 978-1934356005

[10] Wikipedia. “Tail call”. https://en.wikipedia.org/wiki/Tail_call

[11] SGI. “transform”. http://www.sgi.com/tech/stl/transform.html

[12] ACM/ICPC. “The drunk jailer.” Peking University judge online for ACM/ICPC. http://poj.org/problem?id=1218.

[13] Haskell wiki. “Haskell programming tips”. 4.4 Choose the appropriate fold. http://www.haskell.org/haskellwiki/Haskell_programming_tips

[14] Wikipedia. “Dot product”. http://en.wikipedia.org/wiki/Dot_product

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a sec-

tion 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