Part IV

Queues and Sequences

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.

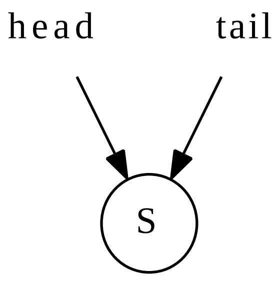


Figure 11.1: The empty queue, both head and tail point to sentinel node.

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($x$)
```

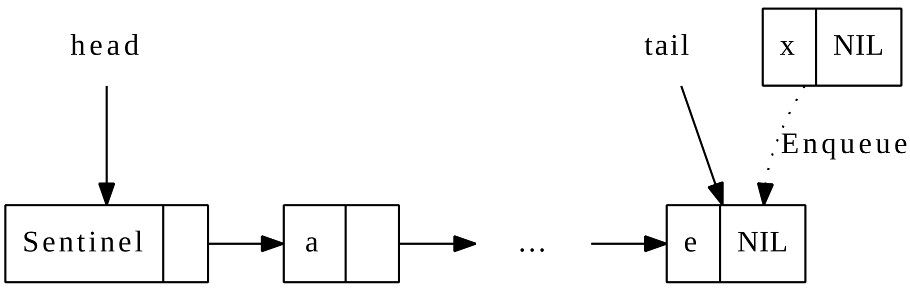
As we always put the sentinel node in front of all the other nodes, function HEAD actually returns the next node to the sentinel.

Figure 11.2 illustrates ENQUEUE and DEQUEUE process with sentinel node.

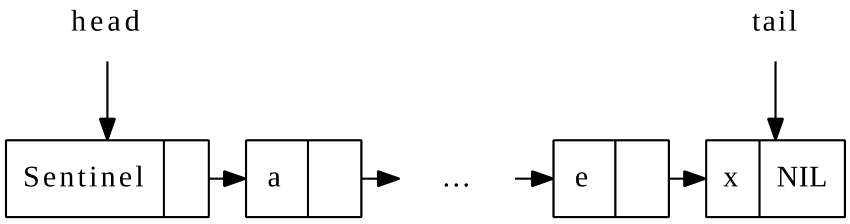
Translating the pseudo code to ANSI C program yields the below code.

```
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)
```

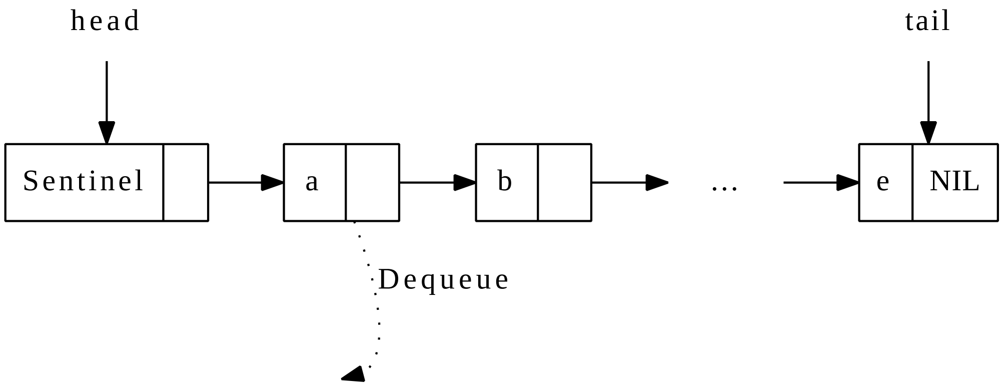
[2]It's possible to add new element to the tail, while remove element from head, but the operations are more complex than this approach.



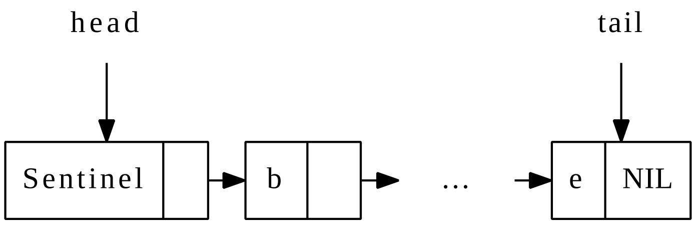
(a) Before ENQUEUE x to queue



(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 reallocate spaces if the array is fully occupied. However, Array performs poor in $O(N)$ time when removing element from head and packing the space. This is because we need shift all rest elements one cell ahead. The idea of circular buffer is to reuse the free cells before the first valid element after we remove elements from head.

The idea of circular buffer can be described in figure 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.

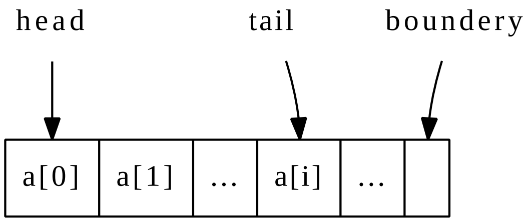
```
struct Queue∗ createQ(int max){
  struct Queue∗ q = (struct Queue∗)malloc(sizeof(struct Queue));
  q→buf = (Key∗)malloc(sizeof(Key)∗max);
  q→size = max;
  q→head = q→tail = 0;
  return q;
}
```

To test if a queue is empty is trivial.

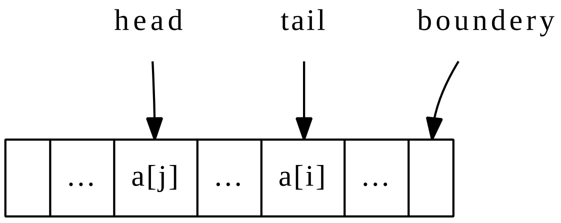
```
function EMPTY?($Q$)
    return HEAD($Q$) = TAIL($Q$)
```

One brute-force implementation for ENQUEUE and DEQUEUE is to calculate the modular of index blindly as the following.

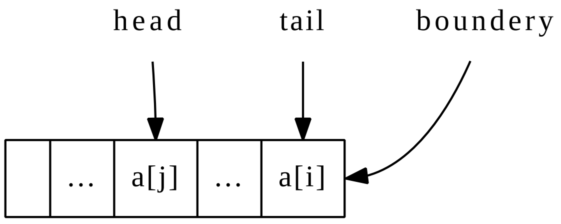
```
function ENQUEUE($Q, x$)
```



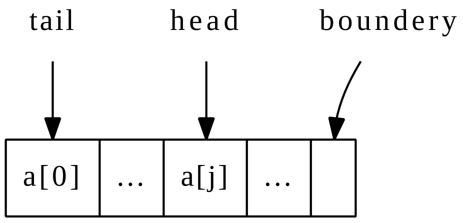
(a) Continuously add some elements.



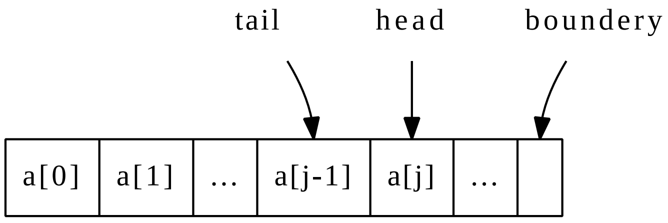
(b) After remove some elements from head, there are free cells.



(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.

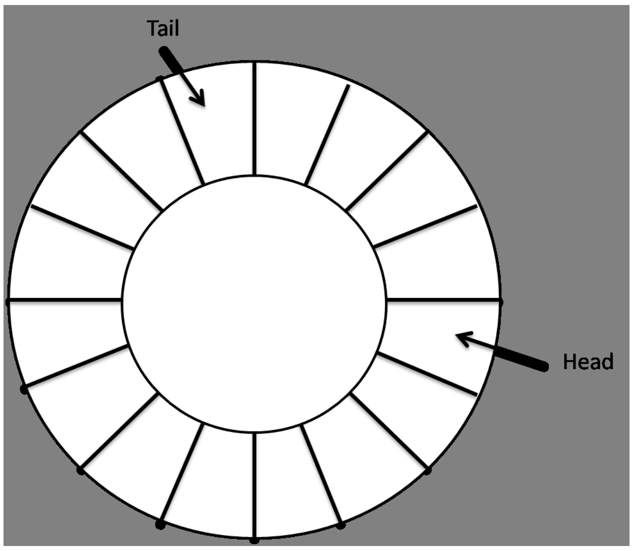


Figure 11.4: The circular buffer.

```
    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;
}
```

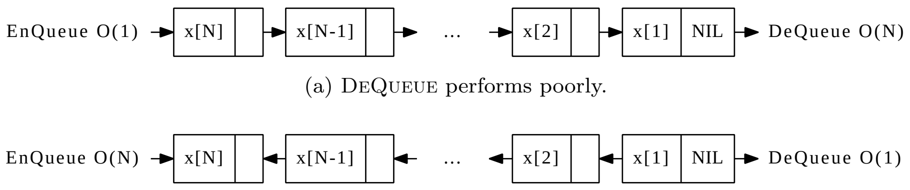
Exercise 11.2

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.

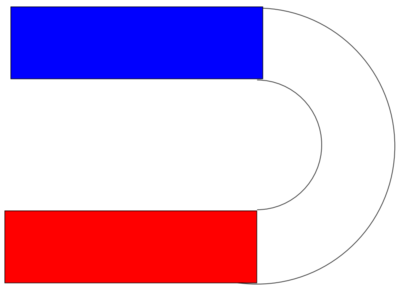
With this setup, we push new element to the head of the rear list, which is ensure to be $O(1)$ constant time; on the other hand, we pop element from the head of the front list, which is also $O(1)$ constant time. So that the abstract queue properties can be satisfied.

The definition of such paired-list queue can be expressed in the following Haskell code.

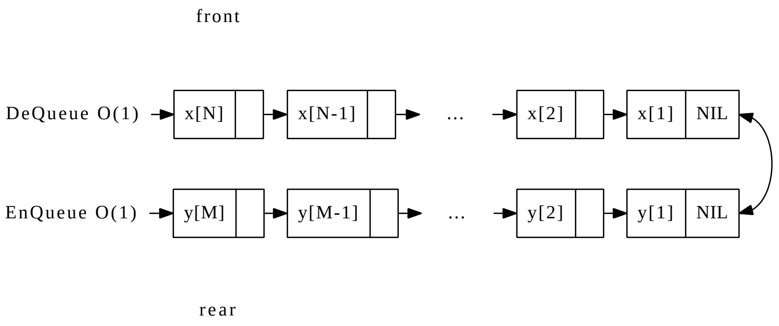
```haskell
type Queue a = ([a], [a])

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. The EnQueue (push) and DeQueue (pop) operations can be easily realized based on this setup.

$$push(Q, x) = Queue(front(Q), \{x\} \cup rear(Q)) \tag{11.1}$$



(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.

$$pop(Q) = Queue(tail(front(Q)), rear(Q)) \tag{11.2}$$

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 $R = fear(Q)$.

$$balance(F, R) = \begin{cases} Queue(reverse(R), \Phi) & : F = \Phi \\ Q & : otherwise \end{cases} \tag{11.3}$$

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

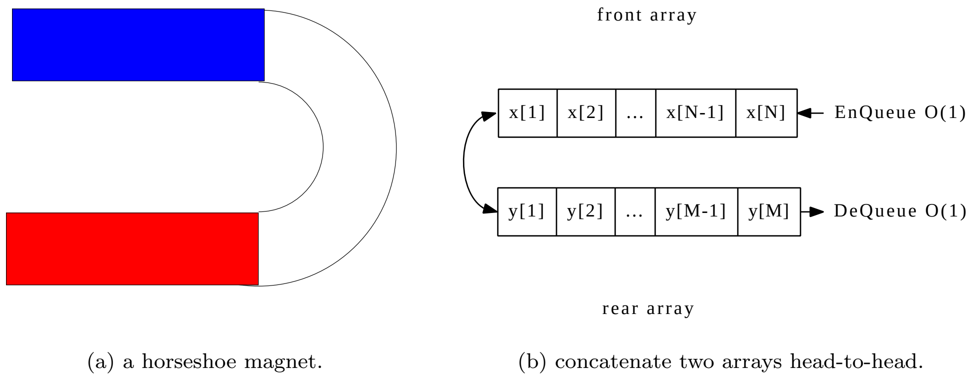
There is an interesting implementation which is symmetric to the paired-list queue. In some old programming languages, such as legacy version of BASIC, There is array supported, but there is no pointers, nor records to represent linked-list. Although we can use another array to store indexes so that 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 = []

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($Q, x$)
```

[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.

```
    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.

$$balance(F, |F|, R, |R|) = \begin{cases} Queue(F, |F|, R, |R|) & : |R| \leq |F| \\ Queue(F \cup reverse(R), |F| + |R|, \Phi, 0) & : otherwise \end{cases} \tag{11.10}$$

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

$$reverse'(X, A) = \begin{cases} A & : X = \Phi \\ reverse'(X', \{x_1\} \cup A) & : otherwise \end{cases} \tag{11.14}$$

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.

$$step(S, X, A) = \begin{cases} (S_f, A) & : S = S_r \wedge X = \Phi \\ (S_r, X', \{x_1\} \cup A) & : S = S_r \wedge X \neq \Phi \end{cases} \tag{11.15}$$

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.

$$\begin{array}{lcl} step(S_r, "hello", \Phi) & = & (S_r, "ello", "h") \\ step(S_r, "ello", "h") & = & (S_r, "llo", "eh") \\ ... & & \\ step(S_r, "o", "lleh") & = & (S_r, \Phi, "olleh") \\ step(S_r, \Phi, "olleh") & = & (S_f, "olleh") \end{array}$$

And in Haskell code manner, the example is like the following.

```
step $ Reverse "hello" [] = Reverse "ello" "h"
step $ Reverse "ello" "h" = Reverse "llo" "eh"
...
step $ Reverse "o" "lleh" = Reverse [] "olleh"
step $ Reverse [] "olleh" = Done "olleh"
```

Now we can distribute the reverse into steps in every pop and push operations. However, the problem is just half solved. We want to break down $F \cup reverse(R)$, and we have broken $reverse(R)$ into steps, we next need to schedule(slow-down) the list concatenation part $F \cup ...$, which is bound to $O(|F|)$, into incremental manner so that we can distribute it to pop and push operations.

Incremental concatenate

It's a bit more challenge to implement incremental list concatenation than list reversing. However, it's possible to re-use the result we gained from increment reverse by a small trick: In order to realize $X \cup Y$, we can first reverse X to $\overleftarrow{X}$, then take elements one by one from $\overleftarrow{X}$ and put them in front of Y just as what we have done in $reverse'$.

$$
\begin{array}{rcl}
X \cup Y & \equiv & reverse(reverse(X)) \cup Y \\
 & \equiv & reverse'(reverse(X), \Phi) \cup Y \\
 & \equiv & reverse'(reverse(X), Y) \\
 & \equiv & reverse'(\overleftarrow{X}, Y)
\end{array}
\tag{11.16}
$$

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.

$$
next(\mathcal{S}) = \begin{cases}
(S_r, F', \{f_1\} \cup \overleftarrow{F}, R', \{r_1\} \cup \overleftarrow{R}) & : \quad S = S_r \land F \neq \Phi \land R \neq \Phi \\
(S_c, \overleftarrow{F}, \{r_1\} \cup \overleftarrow{R}) & : \quad S = S_r \land F = \Phi \land R = \{r_1\} \\
(S_f, A) & : \quad S = S_c \land X = \Phi \\
(S_c, X', \{x_1\} \cup A) & : \quad S = S_c \land X \neq \Phi
\end{cases} \tag{11.17}
$$

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

$$
\begin{array}{rcl}
|R'| & = & |F'| + 1 \\
 & = & |F| + |R| + 1 \\
 & = & 2M + 2
\end{array} \tag{11.18}
$$

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 $2M + 2$ 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.

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.

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
```

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

[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.

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.

$$balance(F, |F|, \mathcal{S}, R, |R|) = \begin{cases} step(F, |F|, \mathcal{S}, R, |R|) & : |R| \leq |F| \\ step(F, |F| + |R|, (S_r, 0, F, \Phi, R, \Phi)\Phi, 0) & : otherwise \end{cases} \tag{11.21}$$

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.

$$step(F, |F|, \mathcal{S}, R, |R|) = \begin{cases} Queue(F', |F|, S_0, R, |R|) & : S' = S_f \\ Queue(F, |F|, \mathcal{S}', R, |R|) & : otherwise \end{cases} \tag{11.22}$$

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.

$$next(\mathcal{S}) = \begin{cases} (S_r, n+1, F', \{f_1\} \cup \overleftarrow{F}, R', \{r_1\} \cup \overleftarrow{R}) & : S = S_r \land F \neq \Phi \\ (S_c, n, \overleftarrow{F}, \{r_1\} \cup \overleftarrow{R}) & : S = S_r \land F = \Phi \\ (S_f, A) & : S = S_c \land n = 0 \\ (S_c, n-1, X', \{x_1\} \cup A) & : S = S_c \land n \neq 0 \\ \mathcal{S} & : otherwise \end{cases} \tag{11.23}$$

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.

$$abort(\mathcal{S}) = \left\{ \begin{array}{rcl} (S_f, A') & : & S = S_c \land n = 0 \\ (S_c, n-1, X'A) & : & S = S_c \land n \neq 0 \\ (S_r, n-1, F, \overleftarrow{F}, R, \overleftarrow{R}) & : & S = S_r \\ \mathcal{S} & : & otherwise \end{array} \right. \quad (11.24)$$

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.

$$\mathcal{S}' = \left\{ \begin{array}{rcl} next(next(\mathcal{S})) & : & F = \Phi \\ next(\mathcal{S}) & : & otherwise \end{array} \right. \quad (11.25)$$

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.

$$\begin{array}{rcll} rotate(X, Y, A) & \equiv & X \cup reverse(Y) \cup A & \text{Definition of (11.26)} \\ & \equiv & \{x_1\} \cup (X' \cup reverse(Y) \cup A) & \text{Associative of } \cup \\ & \equiv & \{x_1\} \cup (X' \cup reverse(Y') \cup (\{y_1\} \cup A)) & \text{Nature of reverse and associative of } \cup \\ & \equiv & \{x_1\} \cup rotate(X', Y', \{y_1\} \cup A) & \text{Definition of (11.26)} \end{array} \tag{11.28}$$

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.

$$balance(Q) = \begin{cases} Queue(\mathcal{F}', \Phi, \mathcal{F}') & : \mathcal{R}_s = \Phi \\ Queue(\mathcal{F}, R, \mathcal{R}_s') & : otherwise \end{cases} \tag{11.32}$$

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

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;

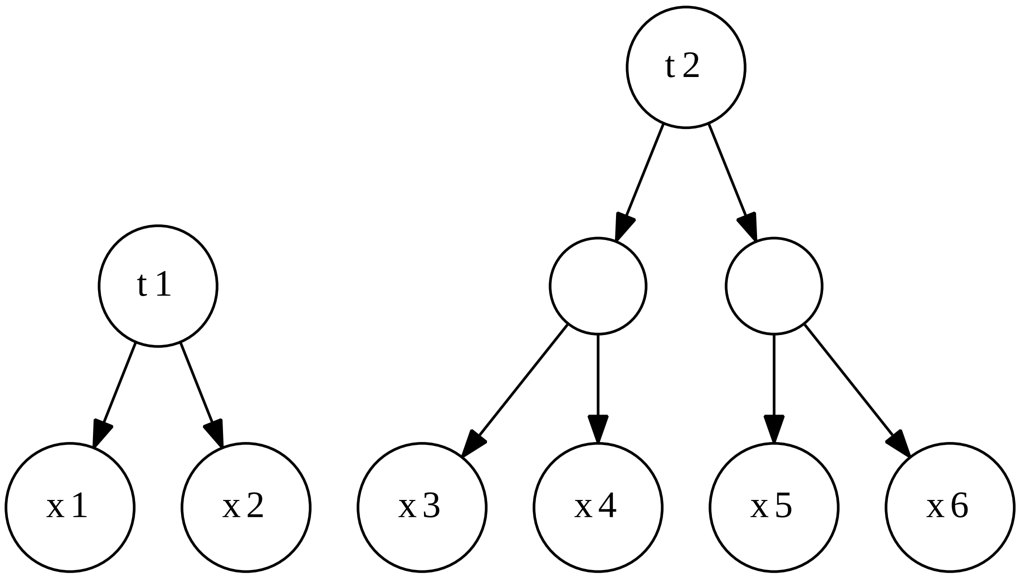


Figure 12.1: A sequence of 6 elements can be represented in a forest.

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.

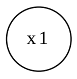
```
size (Leaf _) = 1
size (Node sz _ _) = sz
```

12.2.3 Insertion to the head of the sequence

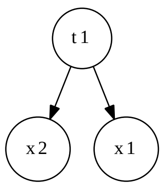
The new forest representation of sequence enables many operation effectively. For example, the operation of inserting a new element y in front of sequence can be realized as the following.

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.

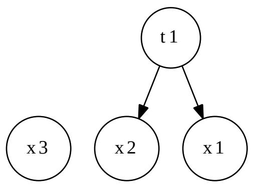
Figure 12.2 and 12.3 illustrate the steps of inserting element $x_1, x_2, ..., x_6$ to an empty tree.



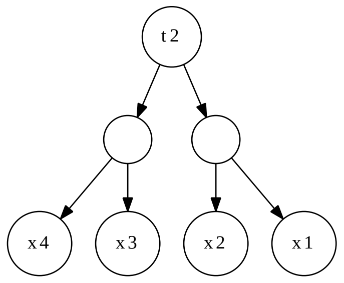
(a) A singleton leaf of x_1



(b) Insert x_2. It causes linking, results a tree of height 1.



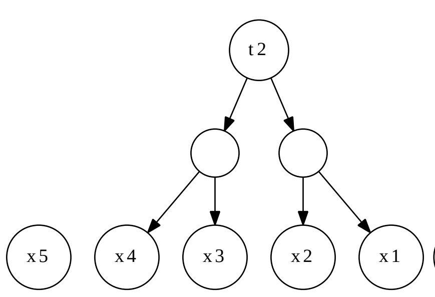
(c) Insert x_3. the result is two trees, t_1 and t_2



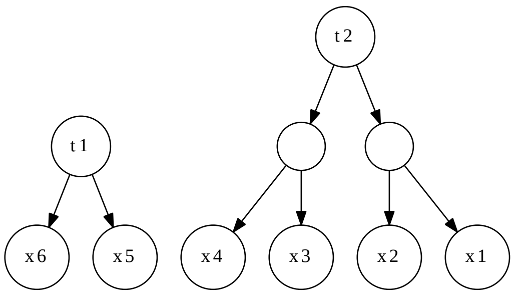
(d) Insert x_4. It first causes linking two leafs to a binary tree, then it performs linking again, which results a final tree of height 2.

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.



(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

Let's formalize the algorithm to equations. we define the function of inserting an element in front of a sequence as $insert(S, x)$.

$$insert(S, x) = insertTree(S, leaf(x)) \tag{12.2}$$

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
```

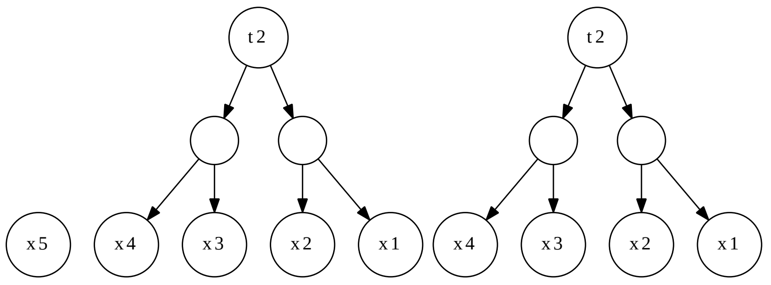
Here we use the Lisp tradition to name the function that insert an element before a list as 'cons'.

Remove the element from the head of the sequence

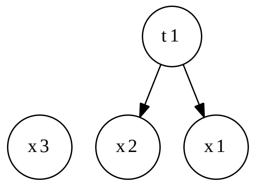
It's not complex to realize the inverse operation of 'cons', which can remove element from the head of the sequence.

- 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 $F' = \{t_2, t_3, ...\}$

$$extractTree(F) = \begin{cases} (t_1, F') & : t_1 \text{ is leaf} \\ extractTree(\{t_l, t_r\} \cup F') & : otherwise \end{cases} \tag{12.5}$$

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

[1] We follow the tradition that the index i starts from 1 in algorithm description; while it starts from 0 in most programming languages

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.

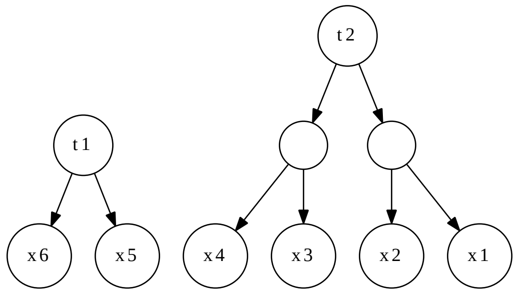
$$lookupTree(T, i) = \begin{cases} root(T) & : i = 1 \\ lookupTree(left(T)) & : i \leq \lfloor \frac{|T|}{2} \rfloor \\ lookupTree(right(T)) & : otherwise \end{cases} \tag{12.9}$$

Where function $left()$ returns the left tree T_l of T, while $right()$ returns T_r.

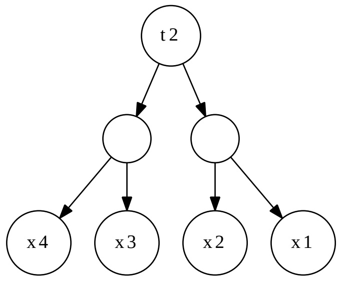
The corresponding Haskell program is given as below.

```
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)
```

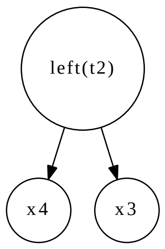
Figure 12.5 illustrates the steps of looking up the 4-th element in a sequence of size 6. It first examine the first tree, since the size is 2 which is smaller than 4, so it goes on looking up for the second tree with the updated index $i' = 4 - 2$, which is the 2nd element in the rest of the forest. 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.



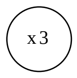
(a) $getAt(S, 4))$, $4 > size(t_1) = 2$



(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.

$$setTree(T, i, x) = \begin{cases} leaf(x) & : \quad i = 0 \land |T| = 1 \\ tree(|T|, setTree(T_l, i, x), T_r) & : \quad i < \lfloor \frac{|T|}{2} \rfloor \\ tree(|T|, T_l, setTree(T_r, i - \lfloor \frac{|T|}{2} \rfloor, x)) & : \quad otherwise \end{cases} \tag{12.11}$$

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.

$$extractTree(S) = \begin{cases} (t, \Phi) & : & S = \{one(t)\} \\ (t, S') & : & s_1 = one(t) \\ (t_l, \{one(t_r)\} \cup S'' & : & otherwise \end{cases} \quad (12.13)$$

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
        $N \leftarrow \lfloor \frac{N}{2} \rfloor$
        $i \leftarrow i + 1$
    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($L, x$)
    $i \leftarrow$ NUMBER-OF-BITS($N \oplus (N + 1)$)
    TREE($L$)[$i + 1$] $\leftarrow$ CREATE-ARRAY($2^i$)
    $l \leftarrow 1$
    TREE($L$)[$i + 1$][$l$] $\leftarrow x$
    for $j \in [1, i]$ do
        for $k \in [1, 2^j]$ do
            $l \leftarrow l + 1$
            TREE($L$)[$i + 1$][$l$] $\leftarrow$ TREE($L$)[$j$][$k$]
        TREE($L$)[$j$] $\leftarrow$ NIL
    SIZE($L$) $\leftarrow$ SIZE($L$) + 1
    return $L$
```

The corresponding ANSI C program is given as the following.

```
struct List insert(struct List a, Key x) {
  int i, j, sz;
  Key* xs;
  i = nbits( (a.n+1) ^ a.n );
  xs = a.tree[i] = (Key*)malloc(sizeof(Key)*(1<<i));
  for(j=0, *xs++ = x, sz = 1; j<i; ++j, sz <<= 1) {
    memcpy((void*)xs, (void*)a.tree[j], sizeof(Key)*(sz));
    xs += sz;
    free(a.tree[j]);
    a.tree[j] = NULL;
  }
  ++a.n;
  return a;
}
```

However, the performance in theory isn't as good as before. This is because the linking operation 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.

$$\begin{array}{rcl} O(T/N) & = & O(\frac{1 \times 2^m + 1 \times 2^{m-1} + 2 \times 2^{m-2} + ... + 2^{m-1} \times 1}{2^m}) \\ & = & O(1 + \frac{m}{2}) \\ & = & O(m) \end{array} \tag{12.15}$$

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.

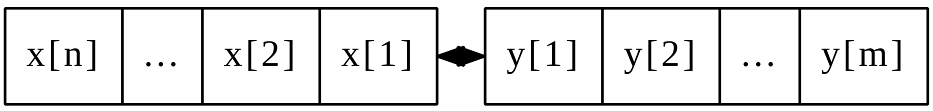


Figure 12.6: A paired-array list, which is consist of 2 arrays linking in head-head manner.

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($R[1 ... \lfloor \frac{M}{2} \rfloor]$)
        $R \leftarrow R[\lfloor \frac{M}{2} \rfloor + 1 ... M]$
    else if $R = \Phi$ then
        $R \leftarrow$ Reverse($F[1 ... \lfloor \frac{N}{2} \rfloor]$)
        $F \leftarrow F[\lfloor \frac{N}{2} \rfloor + 1 ... N]$
```

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)
    $F \leftarrow$ FRONT(L)
    if $F = \Phi$ then
        REMOVE-TAIL(L)
    else
        SIZE(F) $\leftarrow$ SIZE(F) - 1

function REMOVE-TAIL(L)
    BALANCE(L)
    $R \leftarrow$ REAR(L)
    if $R = \Phi$ then
        REMOVE-HEAD(L)
    else
        SIZE(R) $\leftarrow$ SIZE(R) - 1
```

There is an edge case for each, that is even after balancing, the array targeted to perform removal is still empty. This happens that there is only one element stored in the paired-array list. The solution is just remove this singleton left element, and the overall list results empty. Below is the ISO C++ program implements this algorithm.

```
template<typename Key>
void remove_head(List<Key>& xs) {
  balance(xs);
  if(xs.front.empty())
    remove_tail(xs); //remove the singleton elem in rear
  else {
    xs.front.pop_back();
    --xs.n;
  }
}

template<typename Key>
void remove_tail(List<Key>& xs) {
  balance(xs);
  if(xs.rear.empty())
    remove_head(xs); //remove the singleton elem in front
  else {
    xs.rear.pop_back();
    --xs.m;
  }
}
```

It's obvious that the worst case performance is $O(N)$ where N is the number of elements stored in paired-array list. This happens when balancing is triggered, and both reverse and shifting are linear operation. However, the amortized performance of removal is still $O(1)$, the proof is left as exercise to the reader.

Exercise 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:

$$concat(s_1, s_2) = \begin{cases} s_2 & : s_1 = \Phi \\ cons(head(s_1), concat(tail(s_1), s_2)) & : otherwise \end{cases} \tag{12.16}$$

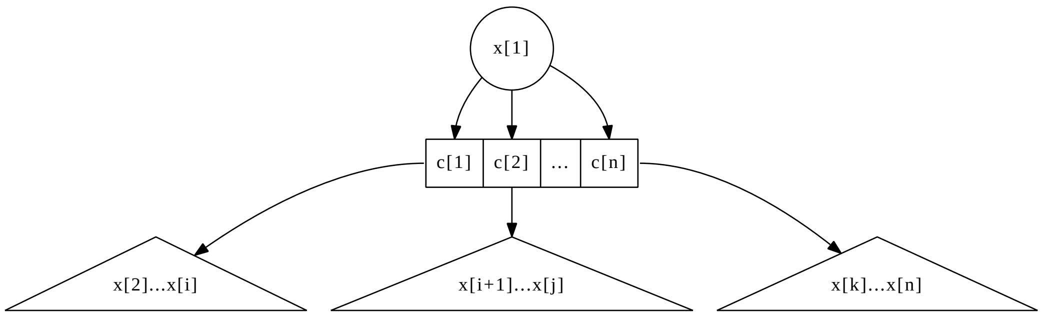
Where function $cons()$, $head()$ and $tail()$ are defined in previous section.

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 [2].

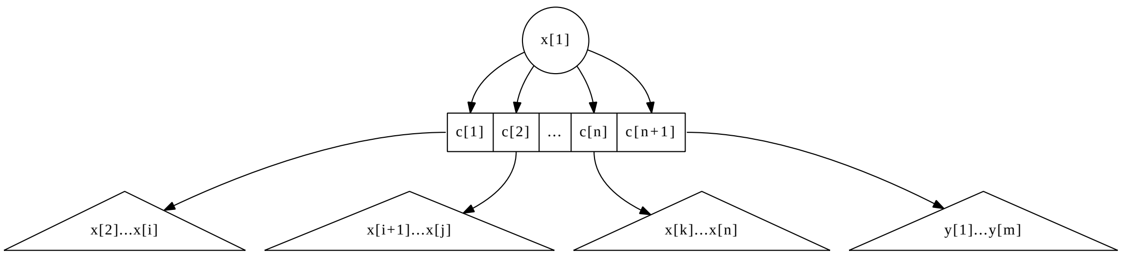
We have already implemented the real-time queue in previous chapter. It supports $O(1)$ time pop and push. If we can turn the sequence concatenation to a kind of pushing operation to queue, the performance will be improved to $O(1)$ as well. Okasaki gave such realization in [6], which can concatenate lists in constant time.

To represent a concatenate-able list, the data structure designed by Okasaki is essentially a K-ary tree. The root of the tree stores the first element in the list. So that we can access it in constant $O(1)$ time. The sub-trees or children are all small concatenate-able lists, which are managed by real-time queues. Concatenating another list to the end is just adding it as the last child, which is in turn a queue pushing operation. Appending a new element can be realized as that, first wrapping the element to a singleton tree, which is a leaf with no children. Then, concatenate this singleton to finalize appending.

Figure 12.7 illustrates this data structure.



(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

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)
```

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.

$$concatAll(Q) = \begin{cases} \Phi & : Q = \Phi \\ concat(front(Q), concatAll(pop(Q))) & : otherwise \end{cases} \tag{12.21}$$

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.

$$\begin{array}{l} foldQ(+, 0, Q) = 1 + (2 + (3 + (4 + (5 + 0)))) = 15 \\ foldQ(\times, 1, Q) = 1 \times (2 \times (3 \times (4 \times (5 \times 1)))) = 120 \\ foldQ(\times, 0, Q) = 1 \times (2 \times (3 \times (4 \times (5 \times 0)))) = 0 \end{array}$$

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]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.

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.

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.

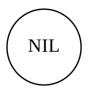
```
data Tree a = Empty
            | Lf a
            | Tr [a] (Tree (Node a)) [a]
```

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.

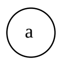
```
struct Tree {
  union Node∗ front;
  union Node∗ rear;
  Tree∗ mid;
  Tree∗ parent;
};
```

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;

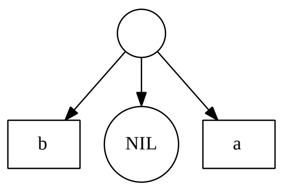
Figure 12.8 and 12.9 show some examples of figure tree.



(a) An empty tree



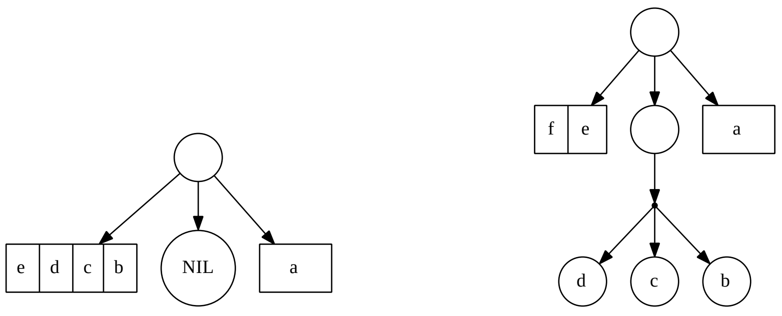
(b) A singleton leaf



(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



(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

example shows a finger tree contains 2 elements, one is in front finger, the other is in rear;

If we continuously insert new elements, to the tree, those elements will be put in the front finger one by one, until it exceeds the limit of 2-3 tree. The 4-th example shows such condition, that there are 4 elements in front finger, which isn't balanced any more.

The last example shows that the finger tree gets fixed so that it resumes balancing. There are two elements in the front finger. Note that the middle part is not empty any longer. It's a leaf of a 2-3 tree. 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.

```haskell
cons :: a → Tree a → Tree a
cons a Empty = Lf a
cons a (Lf b) = Tr [a] Empty [b]
cons a (Tr [b, c, d, e] m r) = Tr [a, b] (cons (Br3 c d e) m) r
cons a (Tr f m r) = Tr (a:f) m r
```

Here we use the LISP naming convention to illustrate inserting a new element to a list.

The insertion algorithm can also be implemented in imperative approach. Suppose function TREE() creates an empty tree, that all fields, including front and rear finger, the middle part inner tree and parent are empty. Function NODE() creates an empty node.

```
function PREPEND-NODE($n, T$)
    $r \leftarrow$ TREE()
    $p \leftarrow r$
    CONNECT-MID($p, T$)
    while FULL?(FRONT($T$)) do
        $F \leftarrow$ FRONT($T$)                          ▷ $F = \{n_1, n_2, n_3, ...\}$
        FRONT($T$) $\leftarrow \{n, F[1]\}$                 ▷ $F[1] = n_1$
        $n \leftarrow$ NODE()
        CHILDREN($n$) $\leftarrow F[2..]$                   ▷ $F[2..] = \{n_2, n_3, ...\}$
        $p \leftarrow T$
        $T \leftarrow$ MID($T$)
    if $T =$ NIL then
        $T \leftarrow$ TREE()
        FRONT($T$)$\leftarrow \{n\}$
    else if | FRONT($T$) | $= 1 \land$ REAR($T$) $= \Phi$ then
        REAR($T$) $\leftarrow$ FRONT($T$)
        FRONT($T$) $\leftarrow \{n\}$
    else
        FRONT($T$) $\leftarrow \{n\} \cup$ FRONT($T$)
    CONNECT-MID($p, T$) $\leftarrow T$
    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 $T \neq$ NIL $\land T$ is empty do
        $T \leftarrow$ MID($T$)
    **if** $T \neq \Phi$ **then**
        PARENT($T$) $\leftarrow \Phi$
    **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.

$$extractT(T) = \left\{ \begin{array}{rcl} (x, \Phi) & : & T = leaf(x) \\ (x, leaf(y)) & : & T = tree(\{x\}, \Phi, \{y\}) \\ (x, tree(\{r_1\}, \Phi, R')) & : & T = tree(\{x\}, \Phi, R) \\ (x, tree(toList(F'), M', R)) & : & T = tree(\{x\}, M, R), (F', M') = extractT(M) \\ (f_1, tree(\{f_2, f_3, ...\}, M, R)) & : & otherwise \end{array} \right. \tag{12.26}$$

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'.

```haskell
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.

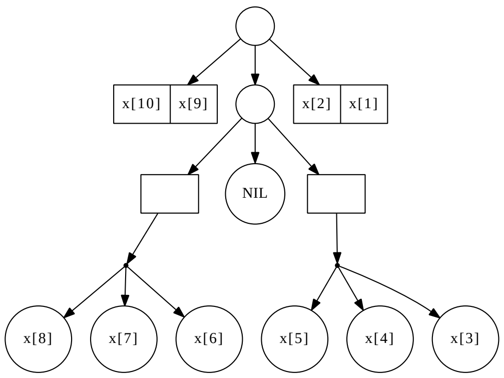
```haskell
nodeToList :: Node a → [a]
nodeToList (Br2 a b) = [a, b]
nodeToList (Br3 a b c) = [a, b, c]
```

Similar as above, we can define $head$ and $tail$ function from $uncons$.

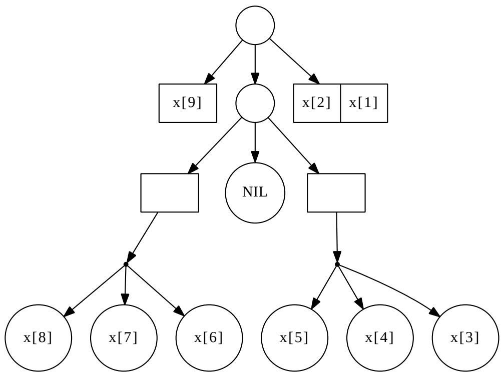
```haskell
head = fst ∘ uncons
tail = snd ∘ uncons
```

12.6.4 Handling the ill-formed finger tree when removing

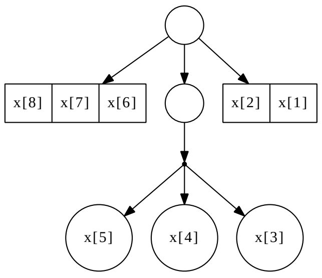
The strategy used so far to remove element from finger tree is a kind of removing and borrowing. If the front finger becomes empty after removing, we borrows more nodes from the middle part inner tree. However there exists cases that the tree is ill-formed, for example, both the front fingers of the tree and its middle part inner tree are empty. Such ill-formed tree can result from imperatively splitting, which we'll introduce later.



(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

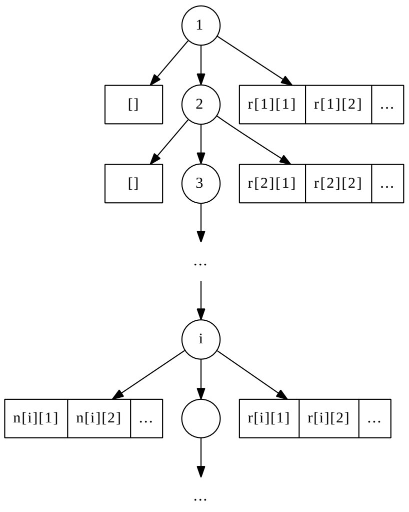


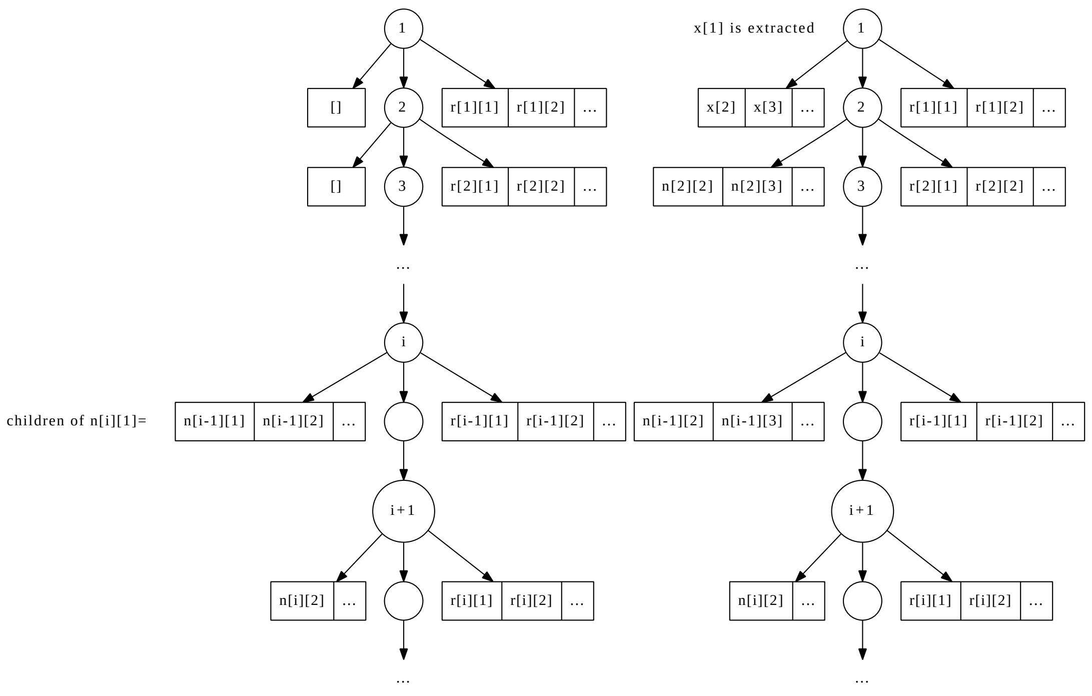
Figure 12.11: Example of an ill-formed tree. The front finger of the i-th level sub tree isn't empty.

Here we developed an imperative algorithm which can remove the first element from finger tree even it is ill-formed. The idea is first perform a top-down traverse to find a sub tree which either has a non-empty front finger or both its front finger and middle part inner tree are empty. For the former case, we can safely extract the first element which is a node from the front finger; For the latter case, since only the rear finger isn't empty, we can swap it with the empty front finger, and change it to the former case.

After that, we need examine the node we extracted from the front finger is leaf node (How to do that? this is left as an exercise to the reader). If not, we need go on extracting the first sub node from the children of this node, and left the rest of other children as the new front finger to the parent of the current tree. We need repeatedly go up along with the parent field till the node we extracted is a leaf. At that time point, we arrive at the root of the tree. Figure 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($n$)                    $\triangleright L = \{n_1, n_2, n_3, ...\}$
        $n \leftarrow L[1]$                              $\triangleright n \leftarrow n_1$
```



(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.

```
        FRONT($T$) ← $L[2..]$                                    ▷ $L[2..] = \{n_2, n_3, ...\}$
        $T$ ← PARENT($T$)
        **if** MID($T$) becomes empty **then**
            MID($T$) ← NIL
    **until** $n$ is a leaf
    **return** (ELEM($n$), FLAT($r$))
```

Note that function ELEM(n) returns the only element stored inside leaf node n. Similar as imperative insertion algorithm, a stub 'ground' tree is used as the parent of the root, which can simplify the logic a bit. That's why we need flatten the tree finally.

Below Python program translates the algorithm.

```
def extract_head(t):
    root = Tree()
    root.set_mid(t)
    while t.front == [] and t.mid is not None:
        t = t.mid
    if t.front == [] and t.rear != []:
        (t.front, t.rear) = (t.rear, t.front)
    n = wraps(t.front)
    while True: # a repeat-until loop
        ns = n.children
        n = ns[0]
        t.front = ns[1:]
        t = t.parent
        if t.mid.empty():
            t.mid.parent = None
            t.mid = None
        if n.leaf:
            break
    return (elem(n), flat(root))
```

Member function `Tree.empty()` returns true if 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**
        $T$ ← MID($T$)
    **if** FRONT($T$) $= \Phi \wedge$ REAR($T$) $\neq \Phi$ **then**
        $n$ ← REAR($T$)[1]
    **else**
        $n$ ← 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 $|$ REAR($T$) $| = 1 \wedge$ FRONT($T$) $= \Phi$ then
        FRONT($T$) $\leftarrow$ REAR($T$)
        REAR($T$) $\leftarrow \{n\}$
    else
        REAR($T$) $\leftarrow$ REAR($T$) $\cup \{n\}$
    CONNECT-MID($p, T$) $\leftarrow T$
    return FLAT($r$)
```

And the corresponding Python programs is given as below.

```
def append_node(t, n):
    root = prev = Tree()
    prev.set_mid(t)
    while rearFull(t):
        r = t.rear
        t.rear = r[-1:] + [n]
        n = wraps(r[:-1])
        prev = t
        t = t.mid
    if t is None:
        t = leaf(n)
    elif len(t.rear) == 1 and t.front == []:
        t = Tree(t.rear, None, [n])
    else:
        t = Tree(t.front, t.mid, t.rear + [n])
    prev.set_mid(t)
    return flat(root)
```

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($T$) ← 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.

$$\begin{array}{l} merge(M_1, R_1 \cup F_2, M_2) = tree(front(M_1), S, rear(M_2)) \\ S = merge(mid(M_1), rear(M_1) \cup R_1 \cup F_2 \cup front(M_2), mid(M_2)) \end{array} \tag{12.30}$$

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($T_1, \Phi, T_2$)

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($x$) $\leftarrow \{L[1], L[2]\}$
        $y \leftarrow$ NODE()
        CHILDREN($y$) $\leftarrow \{L[3], L[4]\}$
        $N \leftarrow N \cup \{x, y\}$
    else if $L \neq \Phi$ then
        $n \leftarrow$ NODE()
        CHILDREN($n$) $\leftarrow L$
        $N \leftarrow N \cup \{n\}$
    return $N$
```

It's straight forward to translate the algorithm to below Python program. Where function `wraps()` helps to create an empty node, then set a list as the children of this node.

```
def nodes(xs):
    res = []
    while len(xs) > 4:
        res.append(wraps(xs[:3]))
        xs = xs[3:]
    if len(xs) == 4:
        res.append(wraps(xs[:2]))
        res.append(wraps(xs[2:]))
    elif xs != []:
        res.append(wraps(xs))
    return res
```

Exercise 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.

$$\begin{array}{l} wrap(x) = tr(1, \{x\}) \\ unwrap(n) = x \quad : \quad n = tr(1, \{x\}) \end{array} \tag{12.38}$$

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 $L' = \{a_2, a_3, ...\}$.

$$size''(L) = \left\{ \begin{array}{rcl} 0 & : & L = \Phi \\ size'(a_1) + size''(L') & : & otherwise \end{array} \right. \tag{12.39}$$

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.

$$\begin{array}{l} wraps(L) = tr(size''(L), L) \\ unwraps(n) = L \quad : \quad n = tr(s, L) \end{array} \tag{12.41}$$

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.

$$tree'(F, M, R) = \begin{cases} fromL(F) & : M = \Phi \land R = \Phi \\ fromL(R) & : M = \Phi \land F = \Phi \\ tree'(unwraps(F'), M', R) & : F = \Phi, (F', M') = extractT'(\\ tree'(F, M', unwraps(R')) & : R = \Phi, (M', R') = removeT'(\\ tree(size''(F) + size(M) + size''(R), F, M, R) & : otherwise \end{cases} \tag{12.43}$$

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.

$$insertT'(x, T) = \begin{cases} leaf(x) & : T = \Phi \\ tree'(\{x\}, \Phi, \{y\}) & : T = leaf(x) \\ tree'(\{x, x_1\}, insertT'(wraps(\{x_2, x_3, x_4\}), M), R) & : T = tree(s, \{x_1, x_2, x_3, x_4\}, M, R) \\ tree'(\{x\} \cup F, M, R) & : otherwise \end{cases} \quad (12.44)$$

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 $|$ FRONT($T$) $| = 1 \land$ REAR($T$) $= \Phi$ then
```

```
        REAR($T$) $\leftarrow$ FRONT($T$)
        FRONT($T$) $\leftarrow \{n\}$
    else
        FRONT($T$) $\leftarrow \{n\} \cup$ FRONT($T$)
    SIZE($T$) $\leftarrow$ SIZE($T$) + SIZE($n$)                    ▷ update size
    CONNECT-MID($p, T$) $\leftarrow T$
    return FLAT($r$)
```

The corresponding Python code are modified accordingly as below.

```
def prepend_node(n, t):
    root = prev = Tree()
    prev.set_mid(t)
    while frontFull(t):
        f = t.front
        t.front = [n] + f[:1]
        t.size = t.size + n.size
        n = wraps(f[1:])
        prev = t
        t = t.mid
    if t is None:
        t = leaf(n)
    elif len(t.front)==1 and t.rear == []:
        t = Tree(n.size + t.size, [n], None, t.front)
    else:
        t = Tree(n.size + t.size, [n]+t.front, t.mid, t.rear)
    prev.set_mid(t)
    return flat(root)
```

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.

$$
\begin{array}{l} (M_l, t, M_r) = splitAt(i - S_f, M) \\ (A, x, B) = splitAtL(i - S_f - size(M_l), unwraps(t)) \end{array}
$$

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($T$))
        $S_m \leftarrow$ SIZE-TR(MID($T$))
        if $i < S_f$ then
            return LOOKUP-NODES(FRONT($T$), $i$, $f$)
        else if $i < S_f + S_m$ then
            $T \leftarrow$ MID($T$)
            $i \leftarrow i - S_f$
        else
            return LOOKUP-NODES(REAR($T$), $i - S_f - S_m$, $f$)
    $n \leftarrow$ FIRST-LF($T$)
    $x \leftarrow$ ELEM($n$)
    ELEM($n$) $\leftarrow f(x)$
    return $x$
```

This algorithm is essentially a divide and conquer tree search. It repeatedly examine the current tree till reach a tree with size of 1 (can it be determined as a leaf? please consider the ill-formed case and refer to the exercise later). Every time, it checks the position to be located with the size information of front finger and middle part inner tree.

If the index i is less than the size of front finger, the location is at some node in it. The algorithm call a sub procedure to look-up among front finger; If the index is between the size of front finger and the total size till middle part inner tree, it means that the location is at some node inside the middle, the algorithm goes on traverse along the middle part inner tree with an updated index reduced by the size of front finger; Otherwise it means the location is at some node in rear finger, the similar looking up procedure is called accordingly.

After this loop, we've got a node, (can be a compound node) with what we are looking for at the first leaf inside this node. We can extract the element out, and apply the function f on it and store the new value back.

The algorithm returns the previous element before applying f as the final result.

What hasn't been factored is the algorithm LOOKUP-NODES(L, i, f). It takes a list of nodes, a position index, and a function to be applied. This algorithm can be implemented by checking every node in the list. If the node is a leaf, and the index is zero, we are at the right position to be looked up. The function can be applied on the element stored in this leaf, and the previous value is returned; Otherwise, we need compare the size of this node and the index to determine if the position is inside this node and search inside the children of the node if necessary.

```
function LOOKUP-NODES($L, i, f$)
    loop
        for $\forall n \in L$ do
            if $n$ is leaf $\wedge i = 0$ then
                $x \leftarrow$ ELEM($n$)
                ELEM($n$) $\leftarrow f(x)$
                return $x$
            if $i <$ SIZE($n$) then
                $L \leftarrow$ CHILDREN($n$)
                break
            $i \leftarrow i -$ SIZE($n$)
```

The following are the corresponding Python code implements the algorithms.

```
def applyAt(t, i, f):
    while t.size > 1:
        szf = sizeNs(t.front)
        szm = sizeT(t.mid)
        if i < szf:
            return lookupNs(t.front, i, f)
        elif i < szf + szm:
            t = t.mid
            i = i - szf
        else:
            return lookupNs(t.rear, i - szf - szm, f)
    n = first_leaf(t)
    x = elem(n)
    n.children[0] = f(x)
    return x

def lookupNs(ns, i, f):
    while True:
        for n in ns:
            if n.leaf and i == 0:
                x = elem(n)
                n.children[0] = f(x)
                return x
            if i < n.size:
                ns = n.children
                break
            i = i - n.size
```

With auxiliary algorithm that can apply function at a given position, it's trivial to implement the GET-AT() and SET-AT() by passing special 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($T, i$)
    $T_1 \leftarrow$ TREE()
    $T_2 \leftarrow$ TREE()
    while $S_f \leq i < S_f + S_m$ do                          ▷ Top-down pass
        $T'_1 \leftarrow$ TREE()
        $T'_2 \leftarrow$ TREE()
        FRONT($T'_1$) $\leftarrow$ FRONT($T$)
        REAR($T'_2$) $\leftarrow$ REAR($T$)
        CONNECT-MID($T_1, T'_1$)
        CONNECT-MID($T_2, T'_2$)
        $T_1 \leftarrow T'_1$
        $T_2 \leftarrow T'_2$
        $i \leftarrow i - S_f$
        $T \leftarrow$ MID($T$)
    if $i < S_f$ then
        $(X, n, Y) \leftarrow$ SPLIT-NODES(FRONT($T$), $i$)
        $T'_1 \leftarrow$ FROM-NODES($X$)
        $T'_2 \leftarrow T$
        SIZE($T'_2$) $\leftarrow$ SIZE($T$) - SIZE-NODES($X$) - SIZE($n$)
        FRONT($T'_2$) $\leftarrow Y$
    else if $S_f + S_m \leq i$ then
        $(X, n, Y) \leftarrow$ SPLIT-NODES(REAR($T$), $i - S_f - S_m$)
        $T'_2 \leftarrow$ FROM-NODES($Y$)
        $T'_1 \leftarrow T$
        SIZE($T'_1$) $\leftarrow$ SIZE($T$) - SIZE-NODES($Y$) - SIZE($n$)
        REAR($T'_1$) $\leftarrow X$
    CONNECT-MID($T_1, T'_1$)
    CONNECT-MID($T_2, T'_2$)
    $i \leftarrow i-$ SIZE-TR($T'_1$)
    while $n$ is NOT leaf do                                   ▷ Bottom-up pass
        $(X, n, Y) \leftarrow$ SPLIT-NODES(CHILDREN($n$), $i$)
        $i \leftarrow i-$ SIZE-NODES($X$)
        REAR($T_1$) $\leftarrow X$
        FRONT($T_2$) $\leftarrow Y$
        SIZE($T_1$) $\leftarrow$ SUM-SIZES($T_1$)
        SIZE($T_2$) $\leftarrow$ SUM-SIZES($T_2$)
        $T_1 \leftarrow$ PARENT($T_1$)
        $T_2 \leftarrow$ 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 will be covered.