Queue, not so simple as it was thought

Liu Xinyu *

July 23, 2012

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.

*Liu Xinyu
Email: liuxinyu95@gmail.com

Most of the functional contents are based on Chris, Okasaki's great work in [**?**]. There are more than 16 different types of purely functional queue given in that material.

2 Queue by linked-list and circular buffer

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 **??** illustrates an empty list. Both head and tail point to the sentinel NIL 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

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.

[1] It's possible to parameterize the type of the key with C++ template. ANSI C is used here for illustration purpose.

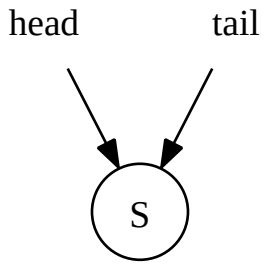


Figure 1: The empty queue, both head and tail point to sentinel node.

```
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)

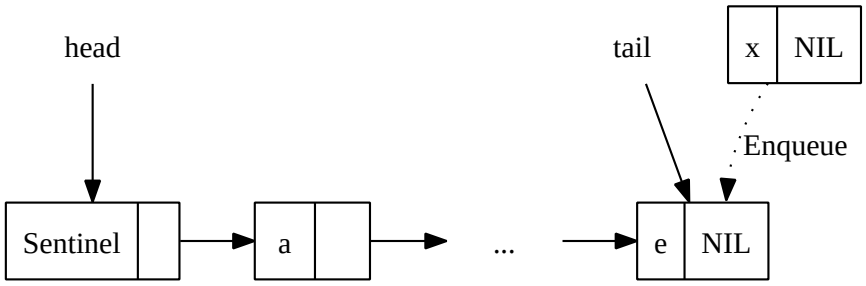
As we always put the sentinel node in front of all the other nodes, function HEAD actually returns the next node to the sentinel.

Figure **??** illustrates ENQUEUE and DEQUEUE process with sentinel node.

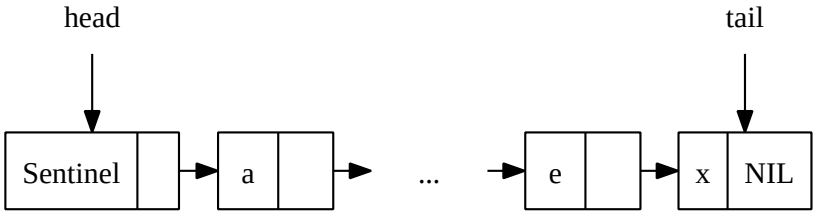
Translating the pseudo code to ANSI C program yields the below code.

```
struct Queue* enqueue(struct Queue* q, Key x){
  struct Node* p = (struct Node*)malloc(sizeof(struct Node));
```

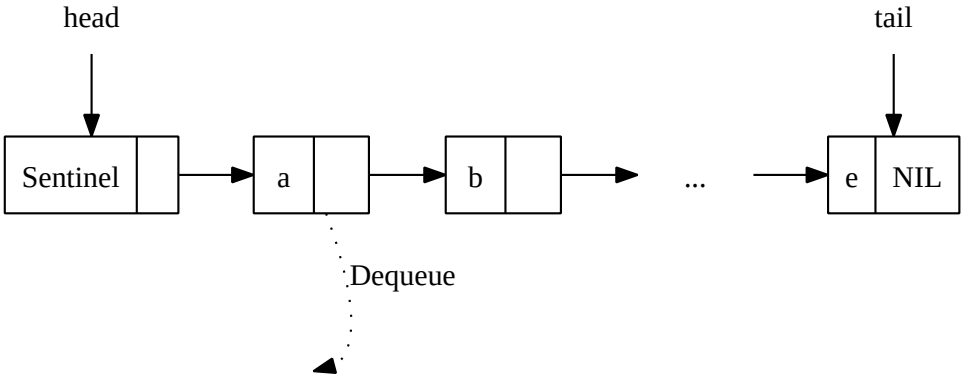
[2]It's possible to add new element to the tail, while remove element from head, but the operations are more complex than this approach.



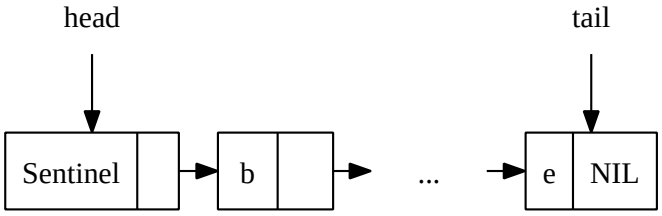
(a) Before ENQUEUE x to queue



(b) After ENQUEUE x to queue



(c) Before DEQUEUE x to queue



(d) After DEQUEUE x to queue

Figure 2: ENQUEUE and DEQUEUE to linked-list queue.

```
  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)
    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 [**?**] [**?**].

Exercise 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.

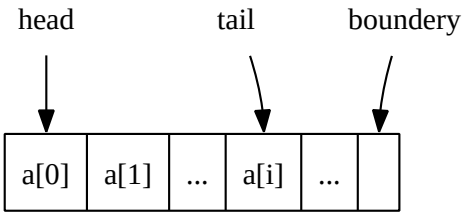
2.2 Circular buffer solution

Another typical solution to realize queue is to use plain array as a circular buffer (also known as ring buffer). Oppose to linked-list, array support appending to the tail in constant $O(1)$ time if there are still spaces. Of course we need re-allocate spaces if the array is fully occupied. However, Array performs poor in $O(N)$ time when removing element from head and packing the space. This is because we need shift all rest elements one cell ahead. The idea of circular buffer is to reuse the free cells before the first valid element after we remove elements from head.

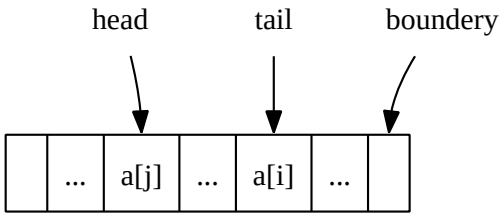
The idea of circular buffer can be described in figure **??** and **??**.

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.

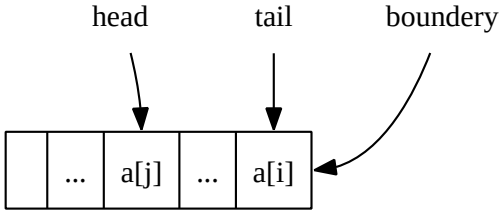
```
struct Queue{
  Key* buf;
  int head, tail, size;
};
```



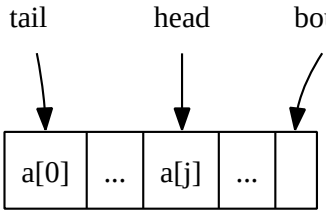
(a) Continuously add some elements.



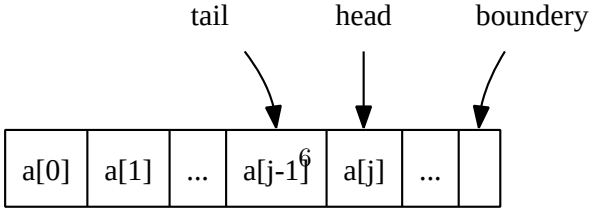
(b) After remove some elements from head, there are free cells.



(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 3: A queue is realized with ring buffer.

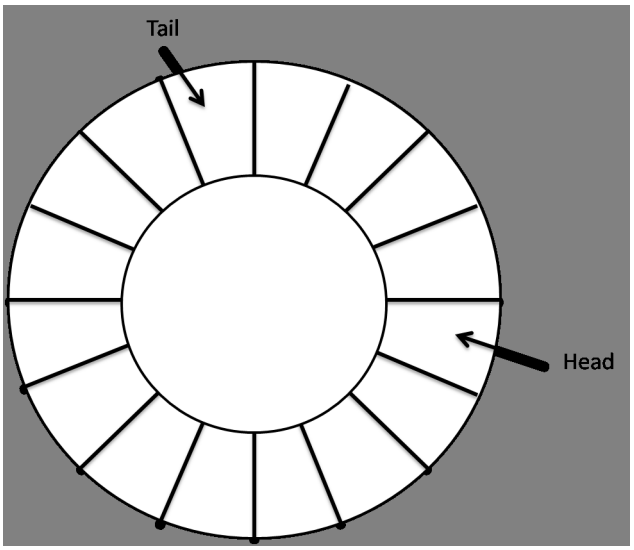


Figure 4: The circular buffer.

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)
  if ¬ FULL?(Q) **then**
    TAIL(Q) ← (TAIL(Q) + 1) mod SIZE(Q)
    BUFFER(Q)[TAIL(Q)] ← x

function HEAD(Q)
  if ¬ EMPTY?(Q) **then**
    return BUFFER(Q)[HEAD(Q)]

function DEQUEUE(Q)
  if ¬ EMPTY?(Q) **then**
    HEAD(Q) ← (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 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.

3 Purely functional solution

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 **??**.

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.

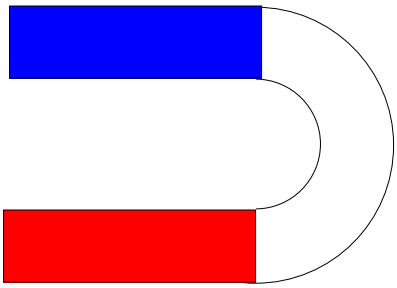
Chris Okasaki mentioned a simple and straightforward functional solution in [**?**]. The idea is to maintain two linked-lists as a queue, and concatenate these two lists in a tail-to-tail manner. The shape of the queue looks like a horseshoe magnet as shown in figure **??**.

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

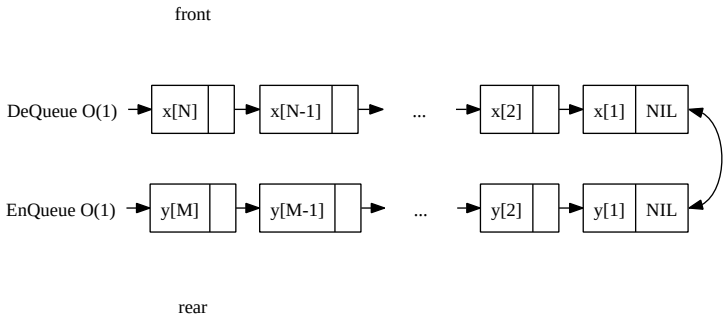
(a) DeQueue performs poorly.

(b) EnQueue performs poorly.

Figure 5: DeQueue and EnQueue can't perform both in constant $O(1)$ time with a list.



(a) a horseshoe magnet.



(b) concatenate two lists tail-to-tail.

Figure 6: A queue with front and rear list shapes like a horseshoe magnet.

head of the front list, which is also $O(1)$ constant time. So that the abstract queue properties can be satisfied.

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

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

empty = ([], [])
```

Suppose function $front(Q)$ and $rear(Q)$ return the front and rear list in such setup, and $Queue(F, R)$ create a paired-list queue from two lists F and R. The EnQueue (push) and DeQueue (pop) operations can be easily realized based on this setup.

$$push(Q, x) = Queue(front(Q), \{x\} \cup rear(Q)) \tag{1}$$

$$pop(Q) = Queue(tail(front(Q)), rear(Q)) \tag{2}$$

where if a list $X = \{x_1, x_2, ..., x_n\}$, function $tail(X) = \{x_2, x_3, ..., x_n\}$ returns the rest of the list without the first element.

However, we must next solve the problem that after several pop operations, the front list becomes empty, while there are still elements in rear list. One method is to rebuild the queue by reversing the rear list, and use it to replace front list.

Hence a balance operation will be execute after popping. Let's denote the front and rear list of a queue Q as $F = front(Q)$, and $R = fear(Q)$.

$$balance(F, R) = \begin{cases} Queue(reverse(R), \Phi) & : \quad F = \Phi \\ Q & : \quad otherwise \end{cases} \tag{3}$$

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.

$$push(Q, x) = balance(F, \{x\} \cup R) \tag{4}$$

$$pop(Q) = balance(tail(F), R) \tag{5}$$

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.

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.)

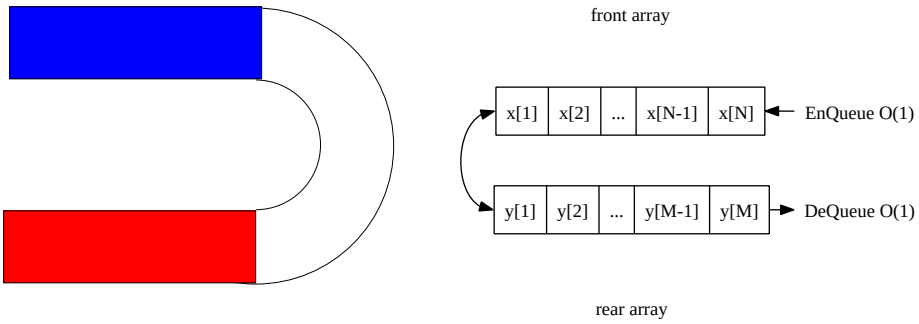
The above table shows an interesting characteristic, that we can exploit it and provide a solution mimic to the paired-list queue: We concatenate two arrays, head-to-head, to make a horseshoe shape queue like in figure **??**.

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 == []
```

[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.



(a) a horseshoe magnet. (b) concatenate two arrays head-to-head.

Figure 7: A queue with front and rear arrays shapes like a horseshoe magnet.

The relative PUSH() and POP() algorithm only manipulate on the tail of the arrays.

function PUSH(Q, x)
  APPEND(REAR(Q), x)

Here we assume that the APPEND() algorithm append element x to the end of the array, and handle the necessary memory allocation etc. Actually, there are multiple memory handling approaches. For example, besides the dynamic re-allocation, we can initialize the array with enough space, and just report error if it's full.

function POP(Q)
  if FRONT(Q) $= \Phi$ **then**
    FRONT(Q) $\leftarrow$ REVERSE(REAR(Q))
    REAR(Q) $\leftarrow \Phi$
  $N \leftarrow$ LENGTH(FRONT(Q))
  $x \leftarrow$ FRONT(Q)[N]
  LENGTH(FRONT(Q)) $\leftarrow N - 1$
  return x

For simplification and pure illustration purpose, the array isn't shrunk explicitly after elements removed. So test if front array is empty (Φ) can be realized as check if the length of the array is zero. We omit all these details here.

The enqueue and dequeue algorithms can be translated to Python programs straightforwardly.

```
def push(q, x):
    q.rear.append(x)

def pop(q):
    if q.front == []:
```

```
        q.rear.reverse()
        (q.front, q.rear) = (q.rear, [])
    return q.front.pop()
```

Similar to the paired-list queue, the performance is amortized $O(1)$ because the reverse procedure takes linear time.

Exercise 3

- Prove that the amortized performance of paired-list queue is $O(1)$.
- Prove that the amortized performance of paired-array queue is $O(1)$.

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.

$$|R| \leq |F| \tag{6}$$

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 (**??**), we can easily tell if a queue is empty by testing the length of the front list.

$$F = \Phi \Leftrightarrow |F| = 0 \tag{7}$$

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.

$$push(Q, x) = balance(F, |F|, \{x\} \cup R, |R| + 1) \tag{8}$$

$$pop(Q) = balance(tail(F), |F| - 1, R, |R|) \tag{9}$$

Where function $balance()$ is defined as the following.

$$balance(F, |F|, R, |R|) = \begin{cases} Queue(F, |F|, R, |R|) & : \quad |R| \leq |F| \\ Queue(F \cup reverse(R), |F| + |R|, \Phi, 0) & : \quad otherwise \end{cases} \tag{10}$$

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 4

Write the symmetric balance improvement solution for paired-array queue in your favorite imperative programming language.

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

$$|R| = |F| + 1 \tag{11}$$

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.

5.0.1 Incremental reverse

Let's examine how functional reverse algorithm is implemented typically.

$$reverse(X) = \begin{cases} \Phi & : \quad X = \Phi \\ reverse(X') \cup \{x_1\} & : \quad otherwise \end{cases} \tag{12}$$

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.

$$reverse(X) = reverse'(X, \Phi) \tag{13}$$

Where

$$reverse'(X, A) = \begin{cases} A & : \quad X = \Phi \\ reverse'(X', \{x_1\} \cup A) & : \quad otherwise \end{cases} \tag{14}$$

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 [**?**] and [**?**] for detail. Such characteristic is easy to change from monolithic algorithm to incremental manner.

The solution is state transferring. We can use a state machine contains two types of stat: reversing state S_r to indicate that the reverse is still on-going (not finished), and finish state S_f to indicate the reverse has been done (finished). In Haskell programming language, it can be defined as a type.

```
data State a = | Reverse [a] [a]
               | Done [a]
```

And we can schedule (slow-down) the above $reverse'(X, A)$ function with these two types of state.

$$step(S, X, A) = \begin{cases} (S_f, A) & : \quad S = S_r \wedge X = \Phi \\ (S_r, X', \{x_1\} \cup A) & : \quad S = S_r \wedge X \neq \Phi \end{cases} \tag{15}$$

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.

5.0.2 Incremental concatenate

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

$$\begin{aligned} X \cup Y &\equiv reverse(reverse(X)) \cup Y \\ &\equiv reverse'(reverse(X), \Phi) \cup Y \\ &\equiv reverse'(reverse(X), Y) \\ &\equiv reverse'(\overleftarrow{X}, Y) \end{aligned} \tag{16}$$

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.

$$next(\mathcal{S}) = \begin{cases} (S_r, F', \{f_1\} \cup \overleftarrow{F}, R', \{r_1\} \cup \overleftarrow{R}) & : & S = S_r \wedge F \neq \Phi \wedge R \neq \Phi \\ (S_c, \overleftarrow{F}, \{r_1\} \cup \overleftarrow{R}) & : & S = S_r \wedge F = \Phi \wedge R = \{r_1\} \\ (S_f, A) & : & S = S_c \wedge X = \Phi \\ (S_c, X', \{x_1\} \cup A) & : & S = S_c \wedge X \neq \Phi \end{cases} \tag{17}$$

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.

5.0.3 Sum up

Before we dive into the final real-time queue implementation. Let's analyze how many incremental steps are taken to achieve the result of $F \cup reverse(R)$. According to the balance variant we used previously, $|R| = |F|+1$, Let's denotes $M = |F|$.

Once the queue gets unbalanced due to some push or pop operation, we start this incremental $F \cup reverse(R)$. It needs $M + 1$ steps to reverse R, and at the same time, we finish reversing the list F within these steps. After that, we need extra $M + 1$ steps to execute the concatenation. So there are $2M + 2$ steps.

It seems that distribute one step inside one pop or push operation is the natural solution, However, there is a critical question must be answered: Is it possible that before we finish these $2M + 2$ steps, the queue gets unbalanced again due to a series push and pop?

There are two facts about this question, one is good news and the other is bad news.

Let's first show the good news, that luckily, continuously pushing can't make the queue unbalanced again before we finish these $2M + 2$ steps to achieve $F \cup reverse(R)$. This is because once we start re-balancing, we can get a new front list $F' = F \cup reverse(R)$ after $2M+2$ steps. While the next time unbalance is triggered when

$$\begin{aligned} |R'| &= |F'| + 1 \\ &= |F| + |R| + 1 \\ &= 2M + 2 \end{aligned} \tag{18}$$

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.

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 **??** at some time after we start the incremental computation and before phase

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 1: Intermediate state of a queue before first M steps finish.

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.

$$push(Q, x) = balance(F, |F|, \mathcal{S}, \{x\} \cup R, |R| + 1) \tag{19}$$

$$pop(Q) = balance(F', |F| - 1, abort(\mathcal{S}), R, |R|) \tag{20}$$

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.

$$balance(F, |F|, \mathcal{S}, R, |R|) = \begin{cases} step(F, |F|, \mathcal{S}, R, |R|) & : & |R| \leq |F| \\ step(F, |F| + |R|, (S_r, 0, F, \Phi, R, \Phi)\Phi, 0) & : & otherwise \end{cases} \tag{21}$$

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.

$$step(F, |F|, \mathcal{S}, R, |R|) = \begin{cases} Queue(F', |F|, S_0, R, |R|) & : & S' = S_f \\ Queue(F, |F|, \mathcal{S}', R, |R|) & : & otherwise \end{cases} \tag{22}$$

Where $\mathcal{S}' = next(\mathcal{S})$ is the next state transferred; $F' = F \cup reverse(R)$, is the final new front list result from the incremental computing. The real state transferring is implemented in $next()$ function as the following. It's different from previous version by adding the counter field n to record how many elements left we need to concatenate.

$$next(\mathcal{S}) = \begin{cases} (S_r, n + 1, F', \{f_1\} \cup \overleftarrow{F}, R', \{r_1\} \cup \overleftarrow{R}) & : & S = S_r \wedge F \neq \Phi \\ (S_c, n, \overleftarrow{F}, \{r_1\} \cup \overleftarrow{R}) & : & S = S_r \wedge F = \Phi \\ (S_f, A) & : & S = S_c \wedge n = 0 \\ (S_c, n - 1, X', \{x_1\} \cup A) & : & S = S_c \wedge n \neq 0 \\ \mathcal{S} & : & otherwise \end{cases} \tag{23}$$

And the corresponding Haskell code is like this.

```
next (Reverse n (x:f) f' (y:r) r') = Reverse (n+1) f (x:f') r (y:r')
next (Reverse n [] f' [y] r') = Concat n f' (y:r')
next (Concat 0 _ acc) = Done acc
next (Concat n (x:f') acc) = Concat (n-1) f' (x:acc)
next s = s
```

Function $abort()$ is used to tell the state machine, we can concatenate one element less since it is popped.

$$abort(\mathcal{S}) = \begin{cases} (S_f, A') & : \quad S = S_c \wedge n = 0 \\ (S_c, n-1, X'A) & : \quad S = S_c \wedge n \neq 0 \\ (S_r, n-1, F, \overleftarrow{F}, R, \overleftarrow{R}) & : \quad S = S_r \\ \mathcal{S} & : \quad otherwise \end{cases} \tag{24}$$

Note that when $n = 0$ we actually rollback one concatenated element by return A' as the result but not A. (Why? this is left as an exercise.)

The Haskell code for abort function is like the following.

```
abort (Concat 0 _ (_:acc)) = Done acc -- Note! we rollback 1 elem
abort (Concat n f' acc) = Concat (n-1) f' acc
abort (Reverse n f f' r r') = Reverse (n-1) f f' r r'
abort s = s
```

It seems that we've done, however, there is still one tricky issue hidden behind us. If we push an element x to an empty queue, the result queue will be:

$$Queue(\Phi, 1, (S_c, 0, \Phi, \{x\}), \Phi, 0)$$

If we perform pop immediately, we'll get an error! We found that the front list is empty although the previous computation of $F \cup reverse(R)$ has been finished. This is because it takes one more extra step to transfer from the state $(S_c, 0, \Phi, A)$ to (S_f, A). It's necessary to refine the $\mathcal{S}'$ in $step()$ function a bit.

$$\mathcal{S}' = \begin{cases} next(next(\mathcal{S})) & : \quad F = \Phi \\ next(\mathcal{S}) & : \quad otherwise \end{cases} \tag{25}$$

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 [**?**]. 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 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.

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.

$$rotate(X, Y, A) \equiv X \cup reverse(Y) \cup A \tag{26}$$

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:

$$rotate(\Phi, \{y_1\}, A) = \{y_1\} \cup A \tag{27}$$

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.

$$\begin{array}{rcll} rotate(X, Y, A) & \equiv & X \cup reverse(Y) \cup A & \text{Definition of (??)} \\ & \equiv & \{x_1\} \cup (X' \cup reverse(Y) \cup A) & \text{Associative of } \cup \\ & \equiv & \{x_1\} \cup (X' \cup reverse(Y') \cup (\{y_1\} \cup A)) & \text{Nature of reverse and associative of } \cup \\ & \equiv & \{x_1\} \cup rotate(X', Y', \{y_1\} \cup A) & \text{Definition of (??)} \end{array} \tag{28}$$

Summarize the above two cases, yields the final incremental rotate algorithm.

$$rotate(X, Y, A) = \begin{cases} \{y_1\} \cup A & : \quad X = \Phi \\ \{x_1\} \cup rotate(X', Y', \{y_1\} \cup A) & : \quad otherwise \end{cases} \tag{29}$$

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. [**?**]. 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 [**?**] 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.

$$push(Q, x) = balance(\mathcal{F}, \{x\} \cup R, \mathcal{R}_s) \tag{30}$$

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.

$$pop(Q) = balance(\mathcal{F}', R, \mathcal{R}_s) \tag{31}$$

Here $\mathcal{F}'$, force one evaluation to $\mathcal{F}$, the Scheme/Lisp code regarding to this equation is as the following.

```
(define (pop q)
  (balance (stream-cdr (front-lst q)) (rear q) (rots q)))
```

For illustration purpose, we skip the error handling (such as pop from an empty queue etc) here.

And one can access the top element in the queue by extract from the front list stream.

```
(define (front q) (stream-car (front-lst q)))
```

The balance function first checks if the computation stream is completely exhausted, and starts new rotation accordingly; otherwise, it just consumes one evaluation by enforcing the lazy stream.

$$balance(Q) = \begin{cases} Queue(\mathcal{F}', \Phi, \mathcal{F}') & : \quad \mathcal{R}_s = \Phi \\ Queue(\mathcal{F}, R, \mathcal{R}'_s) & : \quad otherwise \end{cases} \tag{32}$$

Here $\mathcal{F}'$ is defined to start a new rotation.

$$\mathcal{F}' = rotate(F, R, \Phi) \tag{33}$$

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)
```

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 [**?**]. 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 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.

References

[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