Exercise 3.23.

A *deque* (“double-ended queue”) is a sequence in which items can be inserted and deleted at either the front or the rear. Operations on deques are the constructor `make-deque`, the predicate `empty-deque?`, selectors `front-deque` and `rear-deque`, and mutators `front-insert-deque!`, `rear-insert-deque!`, `front-delete-deque!`, and `rear-delete-deque!`. Show how to represent deques using pairs, and give implementations of the operations.[1] All operations should be accomplished in $\Theta(1)$ steps.

Answer.

Figure 1 shows an initially empty double-ended queue in which the item `b` is inserted at the rear of the queue. Then `a` is inserted at the front, but sooner it is also removed at the front. `C` is inserted and removed at the rear, and `d` is inserted at the rear.

In terms of data abstraction, we can regard a double-ended queue as defined by the following set of operations:

• a constructor:
`(make-deque)`
returns an empty double-ended queue (a queue containing no items).

• three selectors:
`(empty-deque? <deque>)`
test if the doubled-ended queue is empty.

`(front-deque <deque>)`
returns the object at the front of the double-ended queue, signaling an error if the queue is empty; it does not modified the queue.

`(rear-deque <deque>)`
returns the object at the rear of the double-ended queue, signaling an error if the queue is empty; it does not modified the queue either.

• four mutators:
`(front-insert-deque! <deque> <item>)`
inserts the item at the front of the double-ended queue and returns the modified queue as its value.

`(rear-insert-deque! <deque> <item>)`
inserts the item at the rear of the double-ended queue and returns the modified queue as its value.

`(front-delete-deque! <deque>)`
removes the item at the front of the double-ended queue and returns the modified queue as its value, signaling an error if the queue is empty before the deletion.

`(rear-delete-deque! <deque>)`
removes the item at the rear of the double-ended queue and returns the modified queue as its value, signaling an error if the queue is empty before the deletion.

A double-ended queue is represented, much similar to the single-ended queue, as a pair of pointers, `front-ptr` and `rear-ptr`, which indicate, respectively, the first and last pairs in an ordinary list. Figure 2 illustrates this representation, which is almost the same as figure 3.19.

The procedures to select and to modify the front and rear pointers of a double-ended queue is the same as those of a single-ended queue:

```
(define (front-ptr deque) (car deque))

(define (rear-ptr deque) (cdr deque))

(define (set-front-ptr! deque item) (set-car! deque item))

(define (set-rear-ptr! deque item) (set-cdr! deque item))
```

1. Be careful not to make the interpreter try to print a structure that contains cycles. (See exercise 3.13.)

Operation	Resulting Queue
`(define dq (make-deque))`	
`(rear-insert-deque! dq 'b)`	`b`
`(front-insert-deque! dq 'a)`	`a b`
`(front-delete-deque! dq)`	`b`
`(rear-insert-deque! dq 'c)`	`b c`
`(rear-delete-deque! dq)`	`b`
`(rear-insert-deque! dq 'd)`	`b d`

Figure 1. Double-ended queue operations.

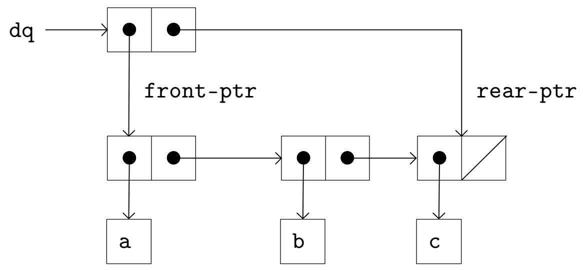

Figure 2. Implementation of a double-ended queue as a list with front and rear pointers.

Now we can implement the actual queue operations. We will consider a double-ended queue to be empty if its front pointer is the empty list:

```
(define (empty-deque? deque) (null? (front-ptr deque)))
```

The `make-deque` constructor returns, as an initially empty double-ended queue, a pair of empty list:

```
(define (make-deque) (cons '() '()))
```

To select the item at the front of the double-ended queue, we return the `car` of the pair indicated by the front pointers respectively:

```
(define (front-deque deque)
  (if (empty-deque? deque)
      (error "FRONT-DEQUE called with an empty double-ended queue" deque)
      (car (front-ptr deque))))
```

Selecting the item located at the rear of the doubled-ended queue is almost identical to that of the front:

```
(define (rear-deque deque)
  (if (empty-deque? deque)
      (error "REAR-DEQUE called with an empty double-ended queue" deque)
      (car (rear-ptr deque))))
```

To insert an item at the front of a double-ended queue, we follow the method whose result is indicated in figure 3. We first create a new pair whose `car` is the item to be inserted and whose `cdr` is the empty list. If the double-ended queue was initially empty, we set the front and rear pointers of the queue to this new pair. Otherwise, we modified this new pair to point to the first pair in the queue, and also set the front pointer to the new pair.

```
(define (front-insert-deque! deque item)
  (let ((new-pair (cons item '())))
    (cond ((empty-deque? deque)
           (set-front-ptr! deque new-pair)
           (set-rear-ptr! deque new-pair)
```

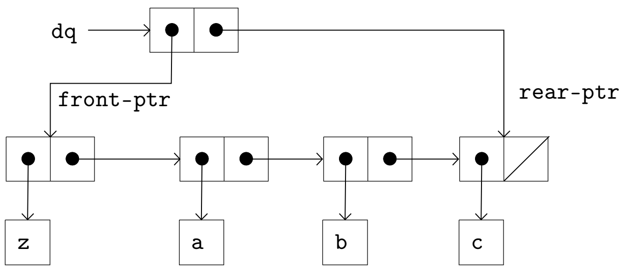

Figure 3. Result of using `(front-insert-deque! dq 'z)` on the queue of figure 2.

```
                deque)
               (else
                 (set-cdr! new-pair front-ptr)
                 (set-front-ptr! deque new-pair)
                 deque))))
```

Inserting an item at the rear of a double-ended queue is simply the same as that of a single-ended queue (see figure 4):

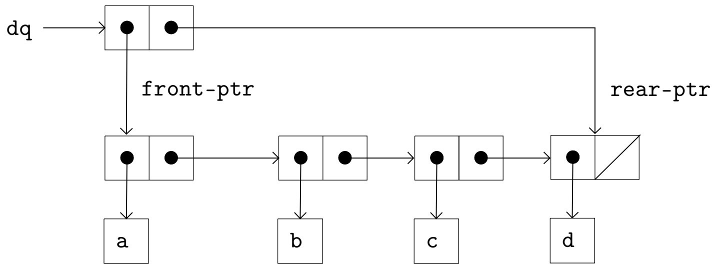

Figure 4. Result of using `(rear-insert-deque! dq 'd)` on the queue of figure 2.

```
(define (rear-insert-deque! deque item)
  (let ((new-pair (cons item '())))
    (cond ((empty-deque? deque)
           (set-front-ptr! deque new-pair)
           (set-rear-ptr! deque new-pair)
           deque)
          (else
            (set-cdr! (rear-ptr deque) new-pair)
            (set-rear-ptr! deque new-pair)
            deque))))
```

To delete the item at the front of the double-ended queue, we merely follow the same way we did in the single-ended queue (see figure 5):

```
(define (front-delete-deque! deque)
  (cond ((empty-deque? deque)
         (error "FRONT-DELETE-DEQUE! called with an empty queue" deque))
        (else
          (set-front-ptr! deque (cdr (front-ptr deque)))
          deque)))
```

Deleting an item at the rear of the double-ended queue, however, is a little bit complex. If the double-ended queue was initially empty, we just report an error and exit. Otherwise, if the queue contains only one item, we simply set the front pointer to the empty list. In the case where the queue is consisted of more than one item, we first traverse the queue to find the item just before the last item, set the rear pointer to it and assign its cdr to the empty list. Figure 6 illustrate the evolution of the double-ended queue in a more intuitive way.

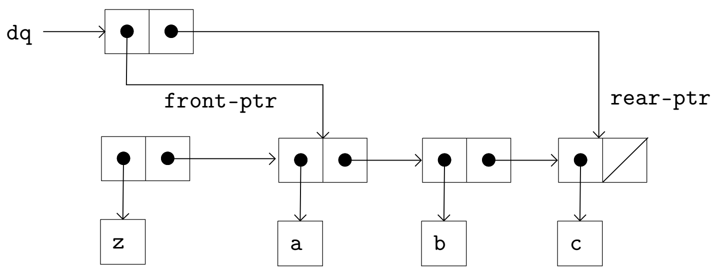

Figure 5. Result of using `(front-delete-deque! dq)` on the queue of figure 3.

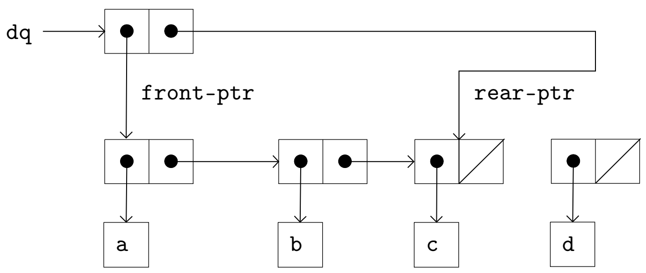

Figure 6. Result of using `(rear-delete-deque! dq)` on the queue of figure 4.

```scheme
(define (delete-rear-deque! deque)
  (define (find-one-before-last L)
    (cond ((null? L)
           (error "FIND-ONE-BEFORE-LAST empty list" L))
          ((null? (cdr (cdr L)))
           L)
          (else
            (find-one-before-last (cdr L)))))
  (cond ((empty-deque? deque)
         (error "REAR-DELETE-DEQUE! called with an empty queue" deque))
        ((null? (cdr (front-ptr deque))) ; the queue consist of only 1 item
         (set-front-ptr! deque '())
         deque)
        (else ; the queue is made up of more than 1 item
          (let (new-last (find-one-before-last (front-ptr deque)))
            (set-rear-ptr! deque new-last)
            (set-cdr! new-last '())
            deque))))
```