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 deque in which the item `b` is inserted at the rear of the sequence.

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. Operations on a deque.

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 deque as defined by the following set of operations:

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

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

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

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

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

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

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

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

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

• the printing procedure:
`(print-deque <deque>)`
prints items in the deque. If the deque is empty, just display the empty list.

Consider a deque that share the same representation with single-ended queues, but allow an additional selectot `rear-deque` and two more mutators: `front-insert-deque!` and `rear-delete-deque!`. We hope, as designated in the problem, that all operations can be accomplish in $\Theta(1)$ steps. Unfortunately, implementing `rear-delete-deque!` in this way is distined to disillusion us. Notice that in the ordinary list, which serves as the foundation to our queue representation, one can only access the precursor of the last item by successive `cdr` operation. This scanning requires $\Theta(n)$ steps for a list of n items, causes the `rear-delete-deque!` to be done in $\Theta(n)$ steps, rather than the $\Theta(1)$ steps we initially hoped.

The modification that ascertains all operations on a deque to be accomplished in $\Theta(1)$ steps is to represent the deque as a doubly linked list, together with an additional pointer that indicated the final pair in the list. That way, when we go to delete the last item in the deque, we can set the rear pointer to the precursor of the last item and so avoid scanning the list.

A deque is represented, then, as a pair of pointers, `front-ptr` and `rear-ptr`, which indicate, respectively, the first and last pairs in a doubly linked list. An typical item in the doubly linked list is made up of two pairs, where the `car` of the upper pair points to the lower pair and its `cdr` indicates the rest of the doubly linked list. The data is stored in the `car` of the lower pair whose `cdr` in this case, points to the precursor of that item. Figure 2 illustrate this representation.

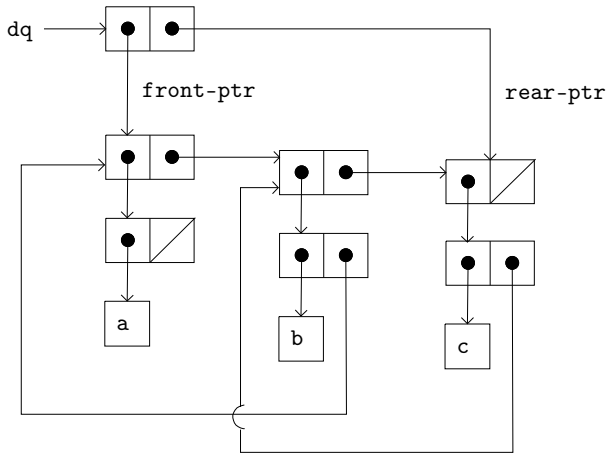

Figure 2. Implementation of a deque as a doubly linked list with front and rear pointers.

The procedures to select and to modify the front and rear pointers of a deque 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))
```

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

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

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

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

To select the item at the front of the deque, we return the `caar` of the pair indicated by the front pointers respectively:

```
(define (front-deque deque)
  (if (empty-deque? deque)
      (error "FRONT-DEQUE called with an empty deque" deque)
      (caar (front-ptr deque))))
```

Selecting the item located at the rear of the deque is almost identical to that of the front:

```
(define (rear-deque deque)
  (if (empty-deque? deque)
      (error "REAR-DEQUE called with an empty deque" deque)
      (caar (rear-ptr deque))))
```

To insert an item at the front of a deque, we follow the method whose result is indicated in figure 3.

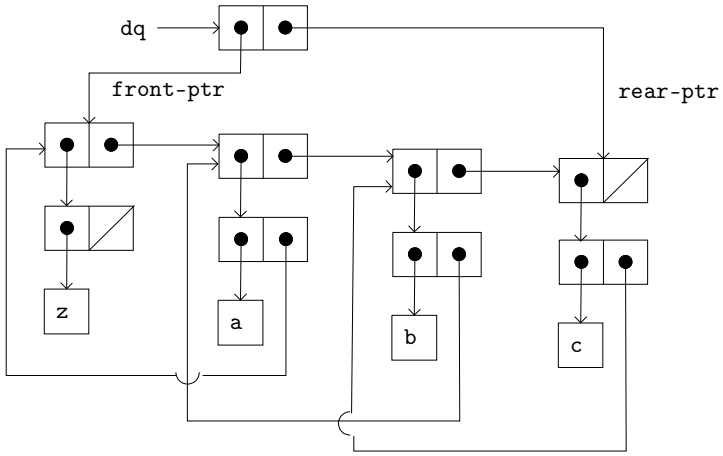

Figure 3. Result of using `(front-insert-deque! dq 'z)` on the deque of figure 2.

We first create a new pair whose `car` is another pair and whose `cdr` is the empty list. The item to be inserted is stored in the `car` of its subordinated pair, while the `cdr` of the latter pair is also the empty list. If the deque was initially empty, we set the front and rear pointers of the deque to this new pair. Otherwise, we first assign the `cdar` of the first pair to this new pair, then modified this new pair to point to the first pair in the deque, and also set the front pointer to the new pair.

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

Inserting an item at the rear of a deque follows almost same ways as we did in inserting items at the front. We begin by creating the a new pair that has the same structure as before. If the deque was initially empty, we set the front and rear pointers of the deque to this new pair. Otherwise, we first assign the `cdar` of the new pair to the last pair, then modified the last pair in the deque to point to this new pair, and also set the rear pointer to the new pair (see figure 4):

```
(define (rear-insert-deque! deque item)
```

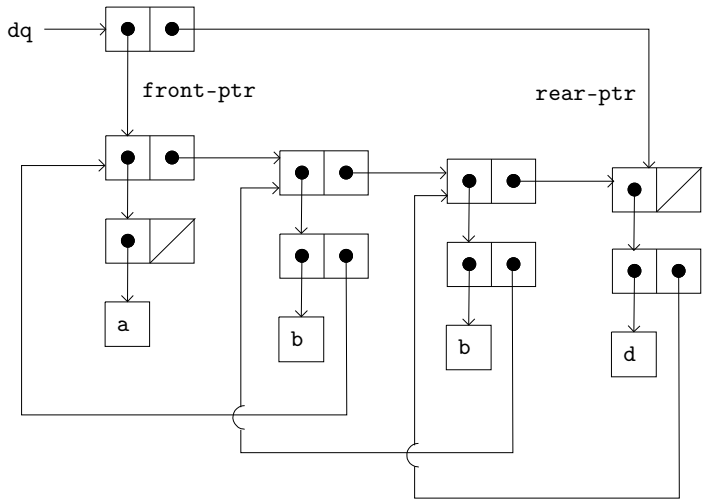

Figure 4. Result of using `(rear-insert-deque! dq 'd)` on the deque of figure 2.

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

To delete the item at the front of the deque, we first modify the front pointer so that it now points at the second item in the deque. Then, we set the `cdr` of the subordinated pair of the second pair of the deque to the empty list (see figure 5):

```
(define (front-delete-deque! deque)
  (cond ((empty-deque? deque)
         (error "FRONT-DELETE-DEQUE! called with an empty deque" deque))
        ((eq? (front-ptr deque) (rear-ptr deque)) ; the deque contains only 1 item
         (set-front-ptr! deque '())
         (set-rear-ptr! deque '())
         deque)
        (else ; the deque consists of more than 1 item
          (set-front-ptr! deque (cdr (front-ptr deque)))
          (set-cdr! (car (front-ptr deque)) '())
          deque)))
```

Deleting an item at the rear of the deque can be address in two cases. We simply report an error and exit whenever the delete procedure encounters an empty deque. Otherwise, we just modify the rear pointer of the deque to the precursor of the last item, togethre with setting the `cdr` of this new last pair to the empty list. Figure 6 illustrate the evolution of the deque in a more intuitive way.

```
(define (rear-delete-deque! deque)
  (cond ((empty-deque? deque)
         (error "REAR-DELETE-DEQUE! called with an empty deque" deque))
        ((eq? (front-ptr deque) (rear-ptr deque)) ; the deque contains only 1 item
         (set-front-ptr! deque '())
         (set-rear-ptr! deque '())
         deque)
        (else ; the deque consists of more than 1 item
```

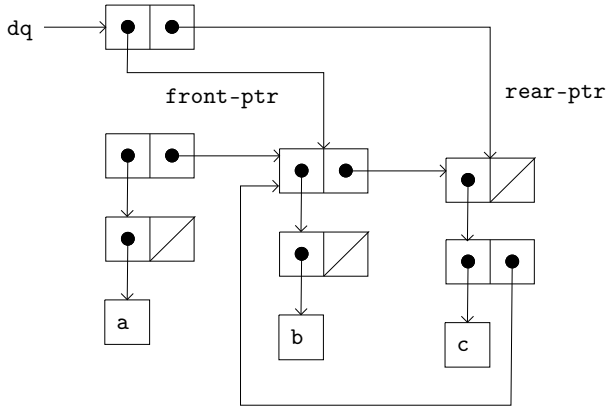

Figure 5. Result of using `(front-delete-deque! dq)` on the deque of figure 2.

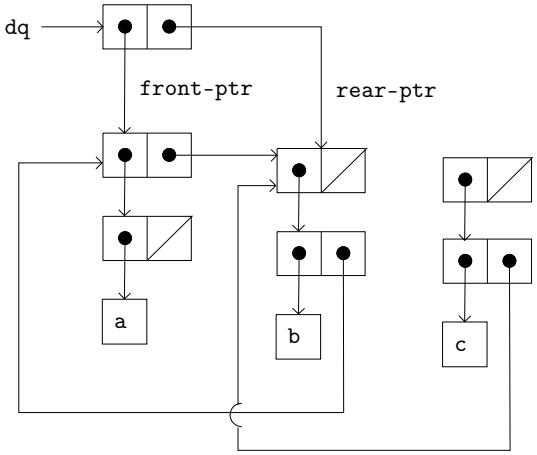

Figure 6. Result of using `(rear-delete-deque! dq)` on the deque of figure 2.

```
        (set-rear-ptr! deque (cdr (car (rear-ptr deque))))
        (set-cdr! (rear-ptr deque) '())
        deque)))
```

Finally, we have to implement the procedure for printing the deque in a user-friendly way:

```
(define (print-deque deque)
  (cond ((empty-deque? deque)
         '())
        (else
          (map (lambda (item) (car item))
               (front-ptr deque)))))
```