

### 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.<sup>1</sup> 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 double-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.

Consider a double-ended queue that share the same representation with single-ended queues, but allow an additional selector `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 destined 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  $\Theta(1)$  step we initially hope.

The modification that ascertains all operations on a double-ended queue to be accomplished in  $\Theta(1)$  steps is to represent the queue 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 queue, we can set the rear pointer to the precursor of the last item and so avoid scanning the list.

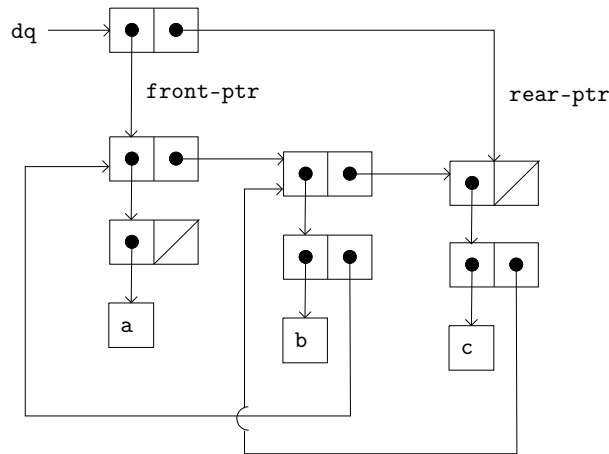
\*. Creative Commons  2013, Lawrence X. Amlord (颜世敏, aka 颜序).

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

Operation	Resulting Queue
<code>(define dq (make-dequeue))</code>	
<code>(rear-insert-dequeue! dq 'b)</code>	b
<code>(front-insert-dequeue! dq 'a)</code>	a b
<code>(front-delete-dequeue! dq)</code>	b
<code>(rear-insert-dequeue! dq 'c)</code>	b c
<code>(rear-delete-dequeue! dq)</code>	b
<code>(rear-insert-dequeue! dq 'd)</code>	b d

**Figure 1.** Double-ended queue operations.

A double-ended queue 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 list. The data is stored in the `car` of the lower pair whose `cdr`, however, points to the precursor of that item. Figure 2 illustrate this representation.



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

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))
```

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-dequeue? deque) (null? (front-ptr deque)))
```

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

```
(define (make-dequeue) (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-dequeue deque)
```

```
(if (empty-deque? deque)
  (error "FRONT-DEQUE called with an empty double-ended queue" deque)
  (caar (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)
    (caar (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 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 double-ended queue was initially empty, we set the front and rear pointers of the queue 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 queue, and also set the front pointer to the new pair.

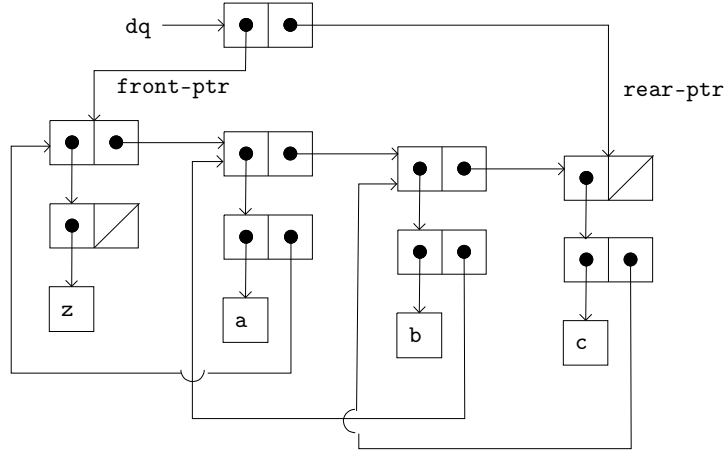
```
(define (front-insert-deque! deque item)
  (let ((new-pair (cons (cons item '())
                        '())))
    (cond ((empty-deque? deque)
           (set-front-ptr! deque new-pair)
           (set-rear-ptr! deque new-pair)
           deque)
          (else
           (set! (cdar (front-ptr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

Inserting an item at the rear of a double-ended queue 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 double-ended queue was initially empty, we set the front and rear pointers of the queue 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 queue to point to this new pair, and also set the rear pointer to the new pair (see figure 4):

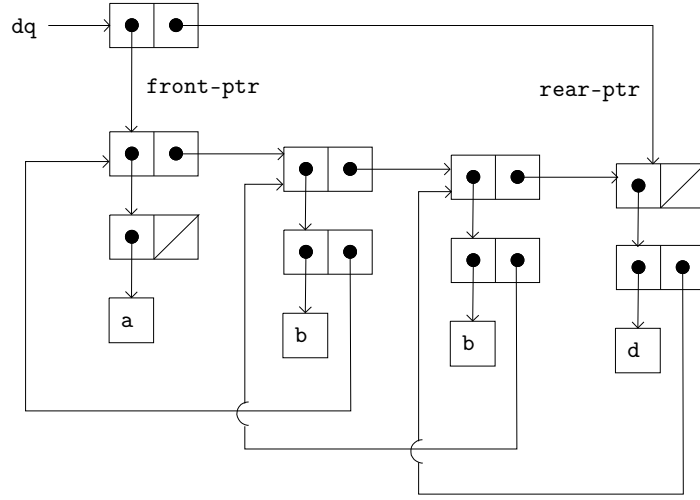
```
(define (rear-insert-deque! deque item)
  (let ((new-pair (cons (cons item '())
                        '())))
    (cond ((empty-deque? deque)
           (set-front-ptr! deque new-pair)
           (set-rear-ptr! deque new-pair)
           deque)
          (else
           (set! (cdar new-pair) (rear-ptr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

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

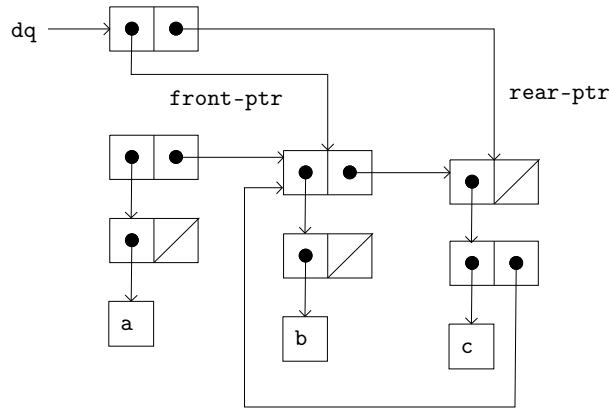
```
(define (front-delete-deque! deque)
  (cond ((empty-deque? deque)
        (error "FRONT-DELETE-DEQUE! called with an empty queue" deque))
        (else
         (set-front-ptr! deque (cdr (front-ptr deque)))
         (set! (cdar (front-ptr deque)) '())
         deque))))
```



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



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



**Figure 5.** Result of using (front-delete-dequeue! dq) on the queue of figure 2.

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

