SICP section 3.3.2

October 03, 2007 at 10:54 **Tags** SICP

The code for this section is in Common Lisp.

Here's all the queue code written before the exercises:

```
(defun front-ptr (queue)
  (car queue))

(defun rear-ptr (queue)
  (cdr queue))

(defun set-front-ptr! (queue item)
  (setf (car queue) item))

(defun set-rear-ptr! (queue item)
  (setf (cdr queue) item))

(defun empty-queue? (queue)
  (null (front-ptr queue)))

(defun make-queue ()
  (cons '() '()))

(defun front-queue (queue)
  (if (empty-queue? queue)
    (error "FRONT on empty queue")
    (car (front-ptr queue))))

(defun insert-queue! (queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
            (set-front-ptr! queue new-pair)
            (set-rear-ptr! queue new-pair)
            queue)
          (t
            (setf (cdr (rear-ptr queue)) new-pair)
            (set-rear-ptr! queue new-pair)
            queue))))

(defun delete-queue! (queue)
  (cond ((empty-queue? queue)
          (error "DELETE! on empty queue"))
        (t
          (set-front-ptr!
            queue
            (cdr (front-ptr queue)))
          queue)))
```

Note that although I've rewritten it in CL, I kept the Scheme convention of appending ? to predicates and ! to mutators. I think it's a useful naming convention and can be used in CL too.

Exercise 3.21

When printing a queue, the Lisp printer sees just a pair created with `cons`, so it prints its `car` and `cdr`. Its `car` is the front queue pointer, which points to a list. This is the first element printed. The rear pointer points to the last element of the list. Hence that last element is shown twice.

Here’s a printing function that just prints the queue itself:

```
(defun print-queue (queue)
  (format t "~a~%" (car queue)))
```

Exercise 3.22

```
(defun make-queue ()
  (let ((front '())
        (rear '()))
    (labels (
      (front-ptr ()
        front)
      (rear-ptr ()
        rear)
      (empty-queue? ()
        (null front))
      (set-front-ptr! (item)
        (setf front item))
      (set-rear-ptr! (item)
        (setf rear item))
      (front-queue ()
        (if (empty-queue?)
          (error "FRONT on empty queue")
          (car front-ptr)))
      (insert-queue! (item)
        (let ((new-pair (cons item '())))
          (cond ((empty-queue?)
                  (set-front-ptr! new-pair)
                  (set-rear-ptr! new-pair))
                (t
                  (setf (cdr (rear-ptr)) new-pair)
                  (set-rear-ptr! new-pair)))))
      (delete-queue! ()
        (cond ((empty-queue?)
                (error "DELETE! on empty queue"))
              (t
                (set-front-ptr!
                  (cdr (front-ptr))))))
      (print-queue ()
        (format t "~a~%" (front-ptr)))
      (dispatch (m)
        (case m
          ('front-ptr #'front-ptr)
          ('rear-ptr #'rear-ptr)
          ('empty-queue? #'empty-queue?)
          ('set-front-ptr! #'set-front-ptr!)
          ('set-rear-ptr! #'set-rear-ptr!)
          ('front-queue #'front-queue)
          ('insert-queue! #'insert-queue!)
          ('delete-queue! #'delete-queue!)
          ('print-queue #'print-queue)
          (otherwise (error "Bad dispatch ~A" m)))))

      #'dispatch)))
```

CL's semantics of function objects are less clean than Scheme's. We begin seeing it in the #' signs above, but it becomes much worse when we see how to call these methods:

```
(defvar q (make-queue))

(funcall (funcall q 'insert-queue!) 't)
(funcall (funcall q 'insert-queue!) 'a)

(funcall (funcall q 'print-queue))
```

In Scheme all those `funcall` calls wouldn't be there at all, but in CL they're a must because it's a Lisp-2 – function symbols and normal symbols "live" in different namespaces.

In reality, code is rarely written in this fashion in Common Lisp. When objects are needed, programmers prefer to use the Common Lisp Object System.

Exercise 3.23

This exercise at first looks simpler than it really is. Note that rear-deletion must be done in `O(1)` as well. This means we can no longer use singly linked lists, because when we delete the last element we have no `O(1)` way to get to the one-before-last element and point the rear pointer to it.

So, the solution is to use a doubly linked list. Each list element will now hold another pair in its `car` (and a pointer to the next element in the `cdr`, as before). In this pair, the `car` will be the data, and the `cdr` will be a back pointer to the previous element. Draw a box-and-pointer diagram if this isn't immediately clear.

Here's the code:

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

(defun front-ptr (deque)
  (car deque))

(defun rear-ptr (deque)
  (cdr deque))

(defun set-front-ptr! (deque item)
  (setf (car deque) item))

(defun set-rear-ptr! (deque item)
  (setf (cdr deque) item))

(defun empty-deque? (deque)
  (null (front-ptr deque)))

(defun front-deque (deque)
  (if (empty-deque? deque)
    (error "FRONT on empty deque")
    (caar (front-ptr deque))))

(defun rear-deque (deque)
  (if (empty-deque? deque)
    (error "REAR on empty deque")
```

```
    (caar (rear-ptr deque))))

(defun 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)
          (t
            (setf (cdr new-pair) (front-ptr deque))
            (setf (cdar (front-ptr deque)) new-pair)
            (set-front-ptr! deque new-pair)
            deque))))

(defun rear-insert-deque! (deque item)
  (let ((new-pair
          (cons
            (cons item (rear-ptr deque))
            '())))
    (cond ((empty-deque? deque)
            (set-front-ptr! deque new-pair)
            (set-rear-ptr! deque new-pair)
            deque)
          (t
            (setf (cdr (rear-ptr deque)) new-pair)
            (set-rear-ptr! deque new-pair)
            deque))))

(defun front-delete-deque! (deque)
  (cond ((empty-deque? deque)
          (error "FRONT-DELETE on empty deque"))
        (t
          (set-front-ptr!
            deque
            (cdr (front-ptr deque)))
          deque)))

(defun rear-delete-deque! (deque)
  (cond ((empty-deque? deque)
          (error "REAR-DELETE on empty deque"))
        (t
          (set-rear-ptr!
            deque
            (cdar (rear-ptr deque)))
          (setf (cdr (rear-ptr deque)) '())
          deque)))

(defun print-deque (deque)
  (format t "(")
  (mapcar
```

```
  (mapcar
    (lambda (e)
      (format t "~a " (car e)))
    (front-ptr deque))
  (format t ")"))
```

For comments, please send me an email.

Back to top