Mutation and Data Structures

A stack supports the following operations

- `(make-stack)` returns an empty stack
- `(insert elt stack)` adds an element to a stack and returns the new stack
- `(delete stack)` removes an element from a stack and returns the new stack
- `(top stack)` returns current top element of stack
- `(empty-stack? stack)` returns true if no elements, false otherwise

The stack satisfies the following contract: If `s` is a stack, created by `(make-stack)`, and `i` is the number of insertions and `j` is the number of deletions, then

1. If `j > i` it is an error.
2. If `j = i` then `(empty-stack? s)` is true and `(top s)`, and `(delete s)` are errors.
3. If `j < i` then `(empty-stack? s)` is false and `(top (delete (insert vals))) = (top s)` for any `vals`.
4. If `j <= i` then `(top (insert vals)) = vals` for any `vals`.

We can implement a stack in the following manner:

```scheme
(define (make-stack) '())
(define (empty-stack? stack) (null? stack))
(define (insert elt stack) (cons elt stack))
(define (delete stack)
  (if (empty-stack? stack)
      (error "Stack underflow -- delete")
      (cdr stack)))
(define (top stack)
  (if (empty-stack? stack)
      (error "Stack underflow -- top")
      (car stack)))
```
Now here is another implementation which doesn’t require grabbing hold of the new stack after deleting an element:

```scheme
(define (make-stack) (cons 'stack '()))

(define (empty-stack? stack) (null? (cdr stack)))

(define (insert elt stack)
  (set-cdr! stack (cons elt (cdr stack)))
  stack)

(define (delete stack)
  (if (empty-stack? stack)
      (error "Stack underflow -- delete")
      (set-cdr! stack (cddr stack)))
  stack)

(define (top stack)
  (if (empty-stack? stack)
      (error "Stack underflow -- top")
      (cadr stack)))
```

A **queue** supports the following operations:

- `(make-queue)` returns an empty queue
- `(insert elt queue)` adds an element to end of queue and returns the new queue
- `(delete queue)` removes an element from front of queue and returns the new queue
- `(head queue)` returns element at front of queue and returns the new queue
- `(empty-queue? queue)` returns true if no elements, false otherwise

The queue’s contract: If `q` is a queue, created by `(make-queue)`, `i` is the number of `insertions`, `j` is the number of `deletions`, and `x_i` is the `i`th item inserted into `q`, then

1. If `j > i` it is an error.
2. If `j = i` then `(empty-queue? q)` is true, and `(head q)`, and `(delete q)` are errors.
3. If `j < i` then `(head q) = x_{j+1}`.

A simple queue implementation:
(define (make-queue) '())

(define (empty-queue? queue) (null? queue))

(define (head queue)
  (if (empty-queue? queue)
      (error "Empty queue -- head")
      (car queue)))

(define (delete queue)
  (if (empty-queue? queue)
      (error "Empty queue -- delete")
      (cdr queue)))

(define (insert elt queue)
  (if (null? queue)
      (cons elt '())
      (cons (car queue)
            (insert elt (cdr queue)))))

An implementation with mutation. First some abstractions for the data structure which maintains the head and tail pointers.

(define (front-ptr queue) (car queue))

(define (rear-ptr queue) (cdr queue))

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

(define (set-rear-ptr! queue item) (set-cdr! queue item))

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

(define (make-queue) (cons '() '()))

Now we can implement the operations on queues:
(define (insert item queue)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
            (set-front-ptr! queue new-pair)
            (set-rear-ptr! queue new-pair)
            queue)
          (else (set-cdr! (rear-ptr queue) new-pair)
            (set-rear-ptr! queue new-pair)
            queue)))))

(define (head queue)
  (if (empty-queue? queue)
      (error "Empty queue -- head")
      (car (front-ptr queue))))

(define (delete queue)
  (cond ((empty-queue? queue)
         (error "Empty queue -- delete")
         ((eq? (front-ptr queue) (head-ptr queue))
          (set-front-ptr? queue '())
          (set-rear-ptr? queue '())
          queue)
       (else (set-front-ptr! queue (cdr (front-ptr queue)))
         queue))))