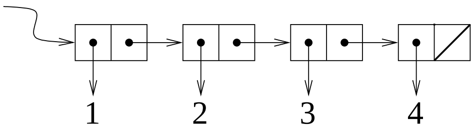
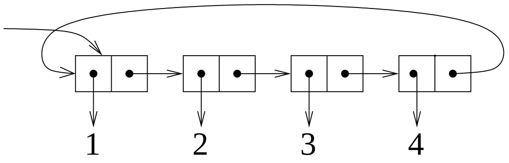
MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 15 — 10/26/2007 Solutions
Mutable Data Structures

Rings

Rings are a circular structure, similar to a list. Unlike a list however, the cdr of the last pair of a ring points back to the first element:



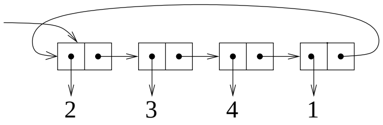


1. Write a function called `make-ring!` that takes a list and makes a ring out of it. You may want to start off writing a helper procdedure called `last-pair`.

```
(define (make-ring! ring-list)
  (define (last-pair lst)
    (if (null? (cdr lst))
        lst
        (last-pair (cdr lst))))
  (or (pair? ring-list) (error "cannot ringify ()"))
  (set-cdr! (last-pair ring-list) ring-list)
  ring-list)
```

2. Write a procedure `rotate-left` that takes a ring and returns a rotated version of the same ring. This procedure should take $\Theta(1)$ time, and not create any new cons cells.

A left-rotated version of the ring above:



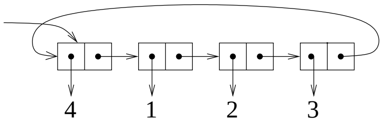
```
(define (rotate-left ring)
  (cdr ring))
```

3. Write a procedure `ring-length` which returns the length (number of elements) in a ring

```
(define (ring-length ring)
 (define (helper n here)
   (if (eq? here ring) n
       (helper (+ 1 n) (cdr here))))
 (helper 1 (cdr ring)))
```

4. Write a procedure `rotate-right` that rotates a ring to the right. Unlike `rotate-left`, `rotate-right` takes $\Theta(n)$ operations, though it still should not create any new cons cells.

A right-rotated version of the ring above:



```
(define (rotate-right ring)
  ((repeated rotate-left
             (- (ring-length ring) 1)) ring))
```

Ring Buffer

Using the ring procedures defined previously, design an ADT for a queue of fixed maximum capacity. It should have a constructor `(make-rb n)`, which creates a ring of `n` elements. `(rb-enqueue! x)` should add `x` to the queue, and `(rb-dequeue!)` should return the next element from the queue. Each enqueue or dequeue operation should take constant time, and not create any new cons cells. The queue may contain at most `n` elements at any one time. Adding more than `n` elements is an error.

For example:

```
(define rb (make-rb 2))          --> unspecified
(rb-enqueue! rb 1)               --> unspecified
(rb-enqueue! rb 2)               --> unspecified
(rb-dequeue! rb)                 --> 1
(rb-enqueue! rb 3)               --> unspecified
(rb-enqueue! rb 4)               --> error -- too many elements
```

1. Finish the definition of `make-rb`:

```
;tagged list (ring-buffer capacity number-filled next-to-read next-to-fill)
(define (make-rb n)
  (let ((rl ((repeated (lambda (x) (cons 'empty x)))
             '())))
    (make-ring! rl)
    (list 'ring-buffer n 0 rl rl)))
```

The definitions of ring selectors are as follows. Note that these are intended to be used only inside `ring-enqueue!` and `ring-dequeue!`, and they return pairs that contain the relevent data elements, rather than the actual values themselves.

```
(define (rb-capacity-pair rb)
  (cdr rb))
```

```
(define (rb-number-filled-pair rb)
  (cddr rb))

(define (rb-next-read-pair rb)
  (cdddr rb))

(define (rb-next-fill-pair rb)
  (cddddr rb))

(define (rb-empty? rb)
  (if (not (ring-buffer? rb))
      (error "not a ring buffer")
      (= (car (rb-number-filled-pair rb)) 0)))

(define (rb-full? rb)
  (if (not (ring-buffer? rb))
      (error "not a ring buffer")
      (= (car (rb-number-filled-pair rb))
         (car (rb-capacity-pair rb)))))
```

2. Complete `rb-enqueue!`.

```
(define (rb-enqueue! rb e)
  (cond ((not (ring-buffer? rb))
         (error "not a ring buffer"))
        ((rb-full? rb)
         (error "too many elements"))
        (else (set-car! (car (rb-next-fill-pair rb)) e)
              (set-car! (rb-next-fill-pair rb)
                        (rotate-left
                         (car (rb-next-fill-pair rb))))
              (set-car! (rb-number-filled-pair rb)
                        (+ 1 (car (rb-number-filled-pair rb)))))))
```

3. Complete `rb-dequeue!`.

```
(define (rb-dequeue! rb)
  (cond ((not (ring-buffer? rb))
         (error "not a ring buffer"))
        ((rb-empty? rb)
         (error "buffer empty"))
        (else
         (let ((val (caar (rb-next-read-pair rb))))
           (set-car! (car (rb-next-read-pair rb)) 'empty)
           (set-car! (rb-next-read-pair rb)
                     (rotate-left
                      (car (rb-next-read-pair rb))))
           (set-car! (rb-number-filled-pair rb)
                     (- (car (rb-number-filled-pair rb)) 1))
           val))))
```