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 procedure called `last-pair`.

   (define (make-ring! 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) ___)

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

   (define (ring-length 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)
```

![Ring Buffer Diagram]

**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

    (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 \texttt{ring-enqueue!} and \texttt{ring-dequeue!}, and they return pairs that contain the relevant data elements, rather than the actual values themselves.

$$\begin{align*}
&\text{(define (rb-capacity-pair rb)} \\
&\quad (\text{cdr } rb))
\end{align*}$$

$$\begin{align*}
&\text{(define (rb-number-filled-pair rb)} \\
&\quad (\text{cddr } rb))
\end{align*}$$

$$\begin{align*}
&\text{(define (rb-next-read-pair rb)} \\
&\quad (\text{cdddr } rb))
\end{align*}$$

$$\begin{align*}
&\text{(define (rb-next-fill-pair rb)} \\
&\quad (\text{cddddr } rb))
\end{align*}$$

$$\begin{align*}
&\text{(define (rb-empty? rb)} \\
&\quad (\text{if (not (ring-buffer? rb)} \\
&\quad\quad (\text{error "not a ring buffer"}) \\
&\quad\quad (= (\text{car (rb-number-filled-pair rb)}) 0))))
\end{align*}$$

$$\begin{align*}
&\text{(define (rb-full? rb)} \\
&\quad (\text{if (not (ring-buffer? rb)} \\
&\quad\quad (\text{error "not a ring buffer"}) \\
&\quad\quad (= (\text{car (rb-number-filled-pair rb)}) \\
&\quad\quad (\text{car (rb-capacity-pair rb)))))))
\end{align*}$$

2. Complete \texttt{rb-enqueue!}.

$$\begin{align*}
&\text{(define (rb-enqueue! rb e)} \\
&\quad (\text{cond ((not (ring-buffer? rb)} \\
&\quad\quad (\text{error "not a ring buffer"})) \\
&\quad\quad ((\text{rb-full? rb)} \\
&\quad\quad\quad (\text{error "too many elements"})) \\
&\quad\quad\text{else}}
\end{align*}$$
3. Complete \texttt{rb-dequeue}!.

\begin{verbatim}
(define (rb-dequeue! rb)
  (cond ((not (ring-buffer? rb))
         (error "not a ring buffer"))
        ((rb-empty? rb)
         (error "buffer empty"))
        (else
          \ldots))
\end{verbatim}