```
set-car!
set-cdr!
```

queues: FIFO
stacks: LIFO

Are set-car! set-cdr! special forms?
Yes, but only because they don't return a value!

In particular, set-car! and set-cdr! do require the evaluation of all subexpressions

Warm up

```
(define x (list (list 1 2) (list 1 2)))
(define y (let ((temp (list 1 2)))
  (list temp temp)))
x ➔ ((1 2) (1 2))
y ➔ ((1 2) (1 2))
(set-car! (car x) 3)
(set-car! (car y) 3)
x ==> ((3 2) (1 2))
y ==> ((3 2) (3 2))
(set-cdr! (car (cons 1 2)) 3)
```
➔ error! the first argument of set-cdr! is not a pair

```
(set-car! (cdr (list 1 2 3)) (cons 4 5))
```
➔ works ok, but nothing useful happens since the return value is unspecified and we lose the data structure since we didn't name it

Ring data structure

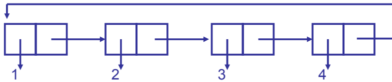
Rings are similar to lists data structures, but they are circular. In this exercise, we won't tag our data structure. If we define a ring using

```
(define r (make-ring! (list 1 2 3 4))
```

then we should get the following:

```
(nth 0 r) ➔ 1
(nth 1 r) ➔ 2
(nth 4 r) ➔ 1
```

Draw a box-and-arrow diagram of the data structure.



Write the function that takes a list as input and creates a ring structure. A helper function might help.

```
(define (last-pair lst)
  (cond ((not (pair? lst))
              (error "should be a pair!"))
        ((null? (cdr lst)) lst)
        (else (last-pair (cdr lst)))))
(define (make-ring! lst)
  (set-cdr! (last-pair lst) lst)
  lst)
(define (n-th n r)
  (if (= 0 n) (car r)
            (n-th (- n 1) (cdr r)))
```

Write the function print-n-elements that displays n elements of a ring like a list display.

```
(define (print-n-elements ring n)
  (cond ((null? ring)
             (error "not enough elements"))
        ((= n 0) (newline))
        (else (display (car ring))
              (display " ")
              (print-n-elements
                  (cdr ring) (- n 1)))))
```

Write the function ring-length that returns the number of elements in the ring.

```
(define (ring-length ring)
   (define (helper count current)
    (if (eq? current ring)
        count
        (helper (+ 1 count) (cdr current))))
  (helper 1 (cdr ring)))
```

Write the function forward that takes a ring and returns a ring shifted by one element.

```
(define (forward ring)
  (cdr ring))
```

Write the function backward.

```
(define (backward ring)
   (define (helper current)
     (if (eq? ring (forward current))
         current
         (helper (forward current))))
   (helper (forward ring)))
```

Order of growth of forward and backward?

forward is Θ(1) in time, while backward is Θ(n)

Write (change-nth! n elt ring) that changes the n-th element of the ring (the total number of elements is unchanged).

```
(define (change-nth! n elt ring)
  (if (= n 0)
    (set-car! ring elt)
    (change-nth! (- n 1) elt (cdr ring))))
```

```
(define (insert-after! elt ring)
  (let ((tmp (cons elt (cdr ring))))
    (set-cdr! ring tmp)))


(define (insert-before! elt ring)
  (insert-after! (backward ring) elt))
```

How could we change the data structure to make some operations more efficient?

use a doubly-linked list. each element points to the next element, like a regular list, but also to the previous element. this could be done with two cons cells

Mutating Append!

Recall the function append that takes two lists and returns a list of the two appended. Recall that to do this, we made a copy of the first list and cons'd it onto the second. Now that we had side effects, we can append two lists without creating a copy of one of them. Assuming you have the function last-pair that we wrote above, write the function append!.

```
(define (append! x y)
  (set-cdr! (last-pair x) y)
  x)
(define (last-pair lst)
  (cond ((not (pair? lst))
             (error "should be a pair!"))
         ((null? (cdr lst)) lst)
         (else (last-pair (cdr lst)))))
```

Mutating reverse!

```
(define (reverse! L)
  (define (helper cur prev)
    (if (null? cur) prev
      (let ((next (cdr cur)))
           (set-cdr! cur prev)
           (helper next cur))))
  (helper L ()))
```