Using a List Container

(define lc (make-list-container))

(for ([i (in-lines)])
  (add-to-front! lc i))

(for-each displayln (get-list lc))
A List Container

(define-struct container (ls) #:mutable)

(define (make-list-container)
  (make-container empty))

(define (add-to-front! lc i)
  (set-container-1s!
   lc
   (cons i (container-1s lc))))

(define (get-list lc)
  (container-1s lc))
List Container

Before:

\[
\text{(define } LC_1 \text{ (make-container (list 1)))}
\]
\[
\text{(add-to-front! } LC_1 \text{ 0)}
\]

After:

\[
\text{(define } LC_1 \text{ (make-container (list 0 1))})
\]
Using a List Container

```c
int main() {
    list_container lc;
    char buffer[256];

    lc = make_list_container();

    for (; fgets(buffer, 256, stdin); ) {
        add_to_front(lc, atoi(buffer));
    }

    print_list(get_list(lc));

    return 0;
}
```
A List Container

```c
struct container {
    list ls;
};
typedef struct container * list_container;

list_container make_list_container() {
    list_container lc;
    lc = (list_container)malloc(sizeof(struct container));
    lc->ls = NULL;
    return lc;
}

void add_to_front(list_container lc, int i) {
    lc->ls = cons(i, lc->ls);
}

list get_list(list_container lc) {
    return lc->ls;
}
```
List Container

Before:

```
add_to_front(lc, 0);
```

After:
Lab

Start with

http://www.eng.utah.edu/~cs2420-20/1c.c

Write tests for make_container(), add_to_front(), and get_list()
Adding to the End of a List

(define (add-to-back! lc i)
  (set-container-1s!
   lc
   (snoc i (container-1s lc))))

(define (snoc i ls)
  (cond
   [(empty? ls) (list i)]
   [else (cons (first ls)
                (snoc i (rest ls)))]))
Adding to the End of a List

**snoc** is painful to implement with a limited stack, so add to the end by finding and mutating the last **int_cons**:

```c
void add_to_back(list_container lc, int i) {
    if (lc->ls == NULL)
        lc->ls = cons(i, NULL);
    else {
        list ls;
        for (ls = lc->ls; ls->rest != NULL; ls = ls->rest) {
        }
        ls->rest = cons(i, NULL);
    }
}
```
Adding to the End of a List

Before:

```
add_to_back(1c, 2);
```

After:
Lab

Recreate

    void add_to_back(list_container lc, int i)

without consulting the previous slide

    Test it
Linked List Performance

on 10000 numbers

Racket:
- Add to front: 43 ms
- Add to back: 8957 ms

C:
- Add to front: 8 ms
- Add to front: 195 ms
Linked List Performance

on 20000 numbers

Racket:
• Add to front: 80 ms
• Add to back: 37195 ms

C:
• Add to front: 16 ms
• Add to front: 757 ms
List Performance: Why

Adding to the front:
- Allocate one cons cell: $O(1)$
- $n$ items: $O(n)$

Adding to the back:
- Traverse existing $n$ cons cells: $O(n)$
- $n$ items: $O(n^2)$
Adding to the Front And Back

```
struct container {
    list hd;
    list tl;
};
```
Adding to the Front And Back

Before:
```
add_to_front(lc, -1);
```

After:
Adding to the Front And Back

Before:

```
add_to_back(lc, 2);
```

After:
Adding to the Front And Back

Before:

\[
\text{add\_to\_back(1c, 1)};
\]

After:
Adding to the Front And Back

Before:

\[ \text{add\_to\_front(lc, 1);} \]

After:
Adding to the Front And Back

Before: \( \text{add\_to\_front}(lc, 0); \)

After:
Adding to the Front And Back

Before:

\[ \text{add\_to\_back}(1c, 2); \]

After:
The New List Container

```c
list_container make_list_container() {
    list_container lc;

    lc = (list_container)malloc(sizeof(struct container));
    lc->hd = NULL;
    lc->tl = NULL;

    return lc;
}
```
Adding to the New List Container

```c
void add_to_front(list_container lc, int i) {
    lc->hd = cons(i, lc->hd);
    if (lc->tl == NULL)
        lc->tl = lc->hd;
}

void add_to_back(list_container lc, int i) {
    if (lc->tl == NULL) {
        lc->hd = cons(i, NULL);
        lc->tl = lc->hd;
    } else {
        lc->tl->rest = cons(i, NULL);
        lc->tl = lc->tl->rest;
    }
}
```
Mutable Cons in Racket

(require racket/mpair)

(define ml (mlist 1 2 3))

(mcar ml) ; = 1
(mcdr ml) ; = (mlist 2 3)

(set-mcar! ml 0)
(mcar ml) ; = 0
ml ; = (mlist 0 2 3)

(set-mcdr! ml (mlist 5))
ml ; = (mlist 1 5)
New List Container

(define-struct container (hd tl) #:mutable)

(define (make-list-container) (make-container empty #f))

(define (add-to-front! lc i)
  (let ([p (mcons i (container-hd lc))]
         [tl (container-tl lc)])
    (unless tl
      (set-container-tl! lc p)
      (set-container-hd! lc p)))

(define (add-to-back! lc i)
  (let ([p (mcons i empty)]
         [tl (container-tl lc)])
    (if tl
        (set-mcdr! tl p)
        (set-container-hd! lc p))
        (set-container-tl! lc p)))

(define (get-list lc)
  (mlist->list (container-hd lc)))
New Linked List Performance

on 10000 numbers

Racket:
• Add to front: 43 ms
• Add to back: 43 ms

C:
• Add to front: 8 ms
• Add to front: 8 ms
Doubly Linked List

```c
struct int_node {
    int val;
    struct int_node * prev;
    struct int_node * next;
};

typedef struct int_node * node;
```
Doubly Linked List

Diagram of a doubly linked list with nodes labeled 0, 1, and 2, showing the direction of links between nodes.
Code is `doubly.c`