Using a List Container

(define lc (make-list-container))

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

(print-list (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-ls! lc
    (cons i (container-ls lc))))

(define (get-list lc)
  (container-ls lc))
List Container

Before:

```
(define LC₁ (make-container (list 1)))
(add-to-front! LC₁ 0)
```

After:

```
(define LC₁ (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:
Mini Lab

Start with 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

\texttt{snoc} is painful to implement with a limited stack, so add to the end by finding and mutating the last \texttt{int\_cons}:

\begin{verbatim}
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);
    }
}
\end{verbatim}
Adding to the End of a List

Before: \[ \text{add\_to\_back}(lc, 2); \]

After:
Mini Lab

Recreate

    void add_to_back(list_container lc, int i)

without consulting the previous slide

Test it
Linked List Performance

on 25000 numbers

Racket:
• Add to front: 25 ms
• Add to back: 7360 ms

C:
• Add to front: 6 ms
• Add to back: 709 ms
Linked List Performance

on 50000 numbers

Racket:
• Add to front: 38 ms
• Add to back: 36540 ms

C:
• Add to front: 13 ms
• Add to back: 2854 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

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

Before:

\[ \text{add_to_front}(lc, -1); \]

After:
Adding to the Front And Back

Before:

\[ \text{add_to_back}(lc, 2); \]

After:
Adding to the Front And Back

Before:

\[
\text{add}_\text{to}_\text{back}(\text{l}_\text{c}, 1);
\]

After:
Adding to the Front And Back

Before:

```
add_to_front(lc, 1);
```

After:
Adding to the Front And Back

Before:

```
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->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 25000 numbers

Racket:
• Add to front: 13 ms
• Add to back: 13 ms

C:
• Add to front: 6 ms
• Add to back: 6 ms
Removing from a Container

Which container variant supports a fast remove operation?

• From the **front**: both plain and head–tail containers

• From the **back**: **neither** plain nor head–tail containers
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
Code is *doubly.c*
Changing the Middle

What about adding or removing in the middle of a list?

• If you have to find the middle: **none** of our choices so far are fast

• If you’re already in the middle somehow: **doubly linked lists** can add and delete adjacent nodes quickly

Addition and deletion operations in a doubly linked list are normally expressed relative to a given node