Programming Abstractions

CS106X

Cynthia Lee

Topics this week:

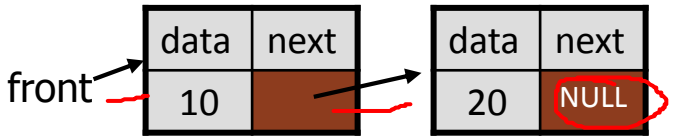
- **Memory and Pointers**
  - Revisit some topics from last week
  - Deeper look at what a pointer is
    - Hexadecimal!
    - Address-of operator: &
  - Dynamic Memory allocation
  - Dereference operator: *
  - Dynamic Memory with classes
    - The -> operator
  - Linked nodes
  - Linked List data structure
  - (if we have time) Priority Queue and Heap data structure

- **TODAY'S TOPICS NOT ON THE MIDTERM**

List code example: Draw a picture!

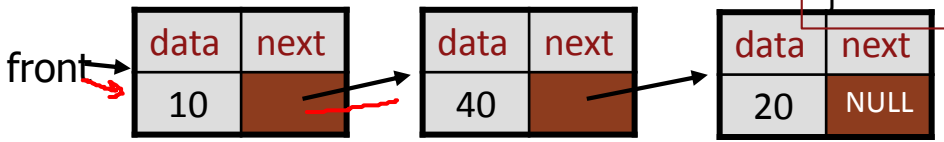
```
struct ListNode {
  ListNode(int d = 0,
           ListNode *n = NULL) {
    data = d;
    next = n;
  }
  int data;
  ListNode *next;
}
```

Before:

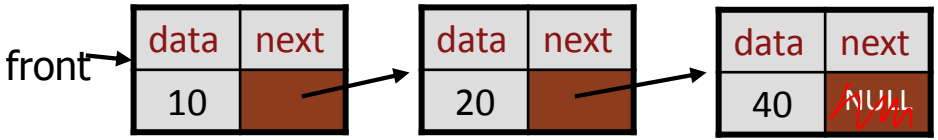



```
front->next->next = new LinkNode();
front->next->next->data = 40;
```

A. After:



B. After:



uninitialized

C. Using "next" that is NULL gives error

D. Other/none/more than one

FIRST RULE OF LINKED NODE/LISTS CLUB:

DRAW A PICTURE OF LINKED LISTS

Do no attempt to code linked nodes/lists without pictures!

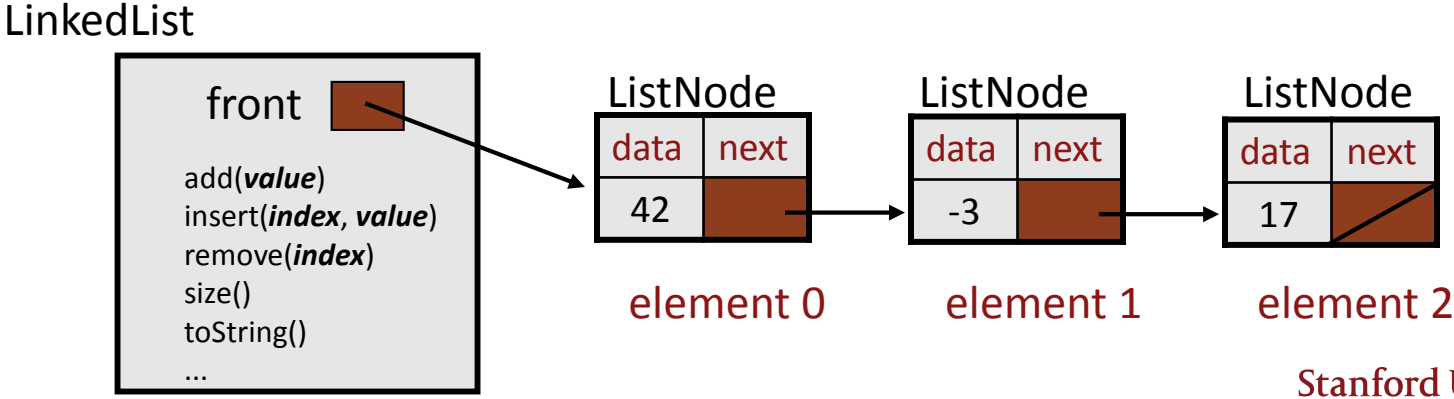
Linked List Data Structure

Putting the ListNode to use

A LinkedList class

Let's write a collection class named `LinkedList`.

- Has the same public members as `ArrayList`, `Vector`, etc.
  - `add`, `clear`, `get`, `insert`, `isEmpty`, `remove`, `size`, `toString`
- The list is internally implemented as a **chain of linked nodes**
  - The `LinkedList` keeps a pointer to its `front` node as a field
  - `NULL` is the end of the list; a `NULL` front signifies an empty list

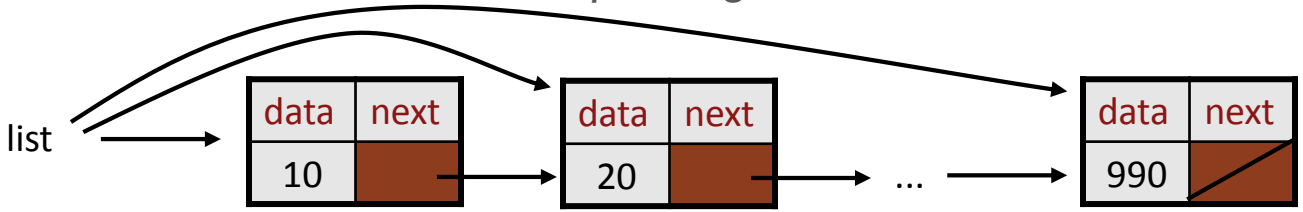


Traversing a list? (BUG version)

What's wrong with this approach to traverse and print the list?

```
while (list != NULL) {

    cout << list->data << endl;
    list = list->next;    // move to next node
}
```

- *It loses the linked list as it is printing it!*

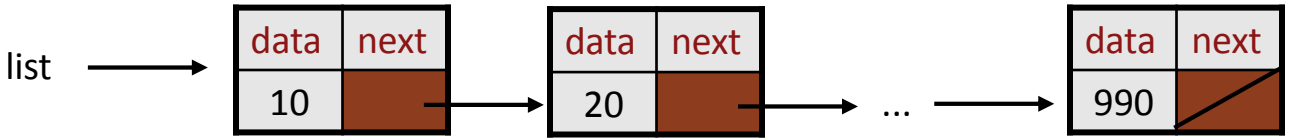


Traversing a list (12.2) (bug fixed version)

The correct way to print every value in the list:

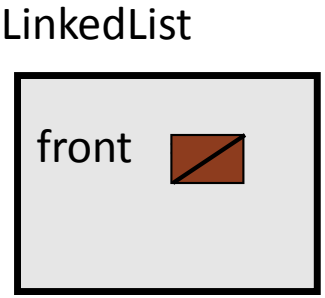
```
ListNode* current = list;
while (current != NULL) {
    cout << current->data << endl;
    current = current->next;  // move to next node
}
```

- Changing `current` does not damage the list.



LinkedList.h

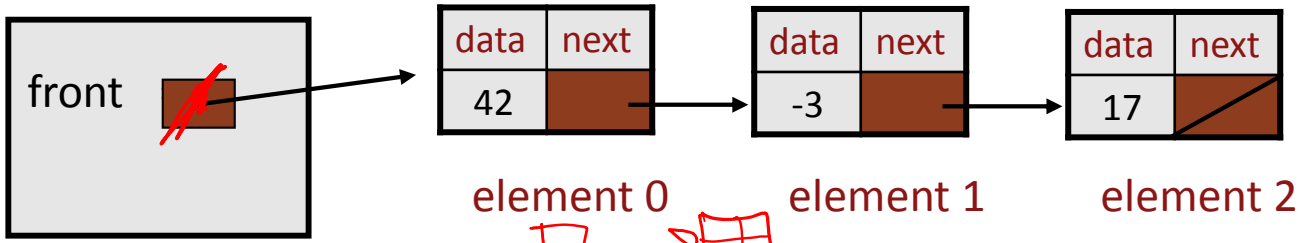
```
class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value);
    int size() const;

private:
    ListNode* front;
};
```



Implementing add

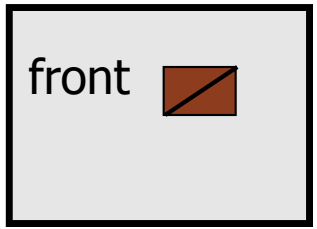
```
// Appends the given value to the end of the list.
void LinkedList::add(int value) {
    ...
}
```

- What pointer(s) must be changed to add a node to the end of a list?
- What different cases must we consider?

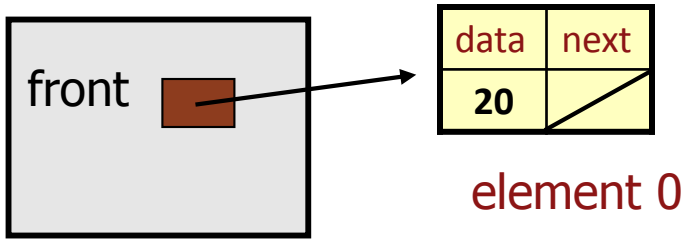


Case 1: Add to empty list

Before adding 20:



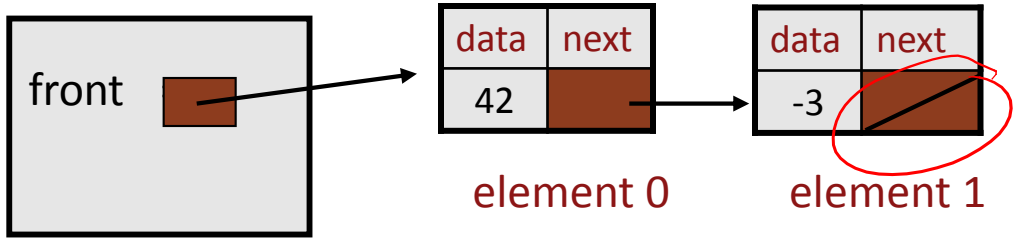
After:



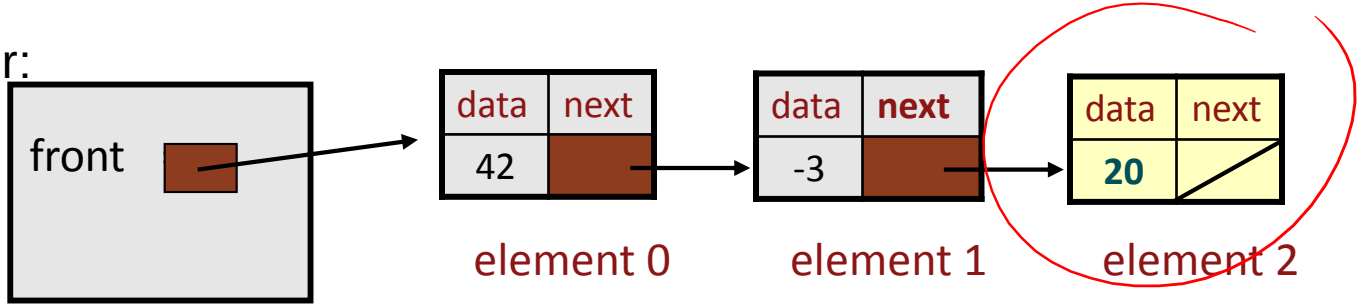
- We must create a new node and attach it to the list.
- For an empty list to become non-empty, we must change `front`.

Case 2: Non-empty list

Before adding value 20 to end of list:

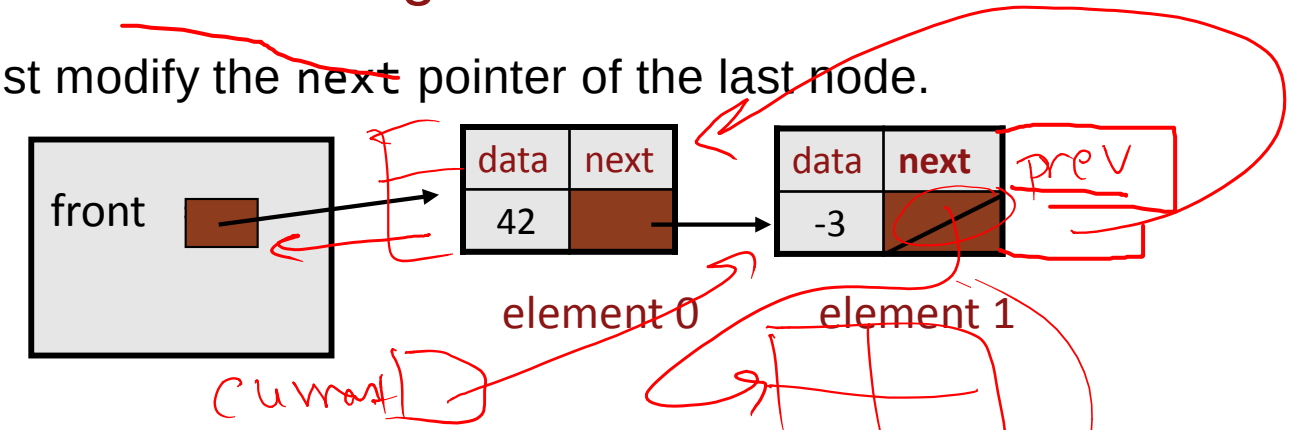


After:



Don't fall off the edge!

Must modify the ~~next~~ pointer of the last node.



- Where should `current` be pointing, to add 20 at the end?

Q: What loop test will stop us at this place in the list?

A. `while (current != NULL) { ...`

B. `while (front != NULL) { ...`

C. `while (current->next != NULL) { ...`

D. `while (front->next != NULL) { ...`

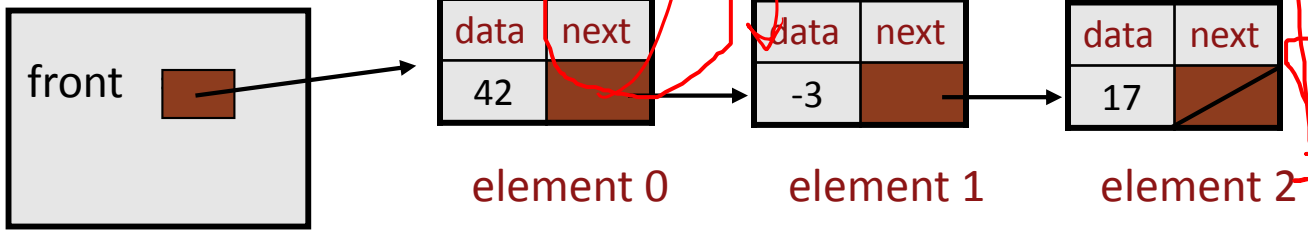
Code for add

```
// Adds the given value to the end of the list.
void LinkedList::add(int value) {
    if (front == NULL) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ListNode* current = front;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = new ListNode(value);
    }
}
```

Implementing get

Binary search log(n)

```
// Returns value in list at given index.
int LinkedList::get(int index) {
    ...
}
```



Vector access O(1)

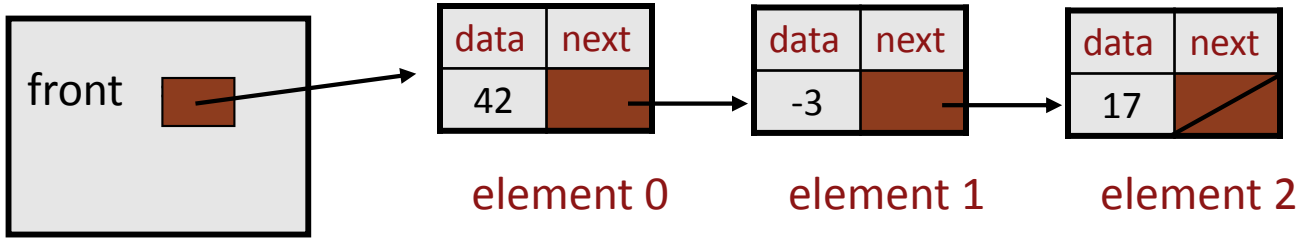
LinkedList access O(N)

Code for get

```
// Returns value in list at given index.
// Precondition: 0 <= index < size()
int LinkedList::get(int index) {
    ListNode* current = front;
    for (int i = 0; i < index; i++) {
        current = current->next;
    }
    return current->data;
}
```

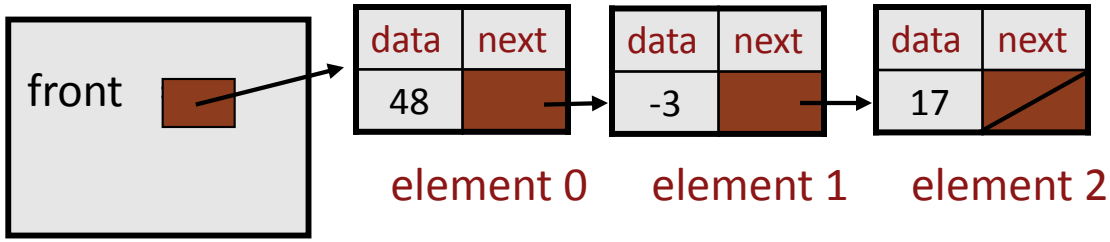
Implementing insert

```
// Inserts the given value at the given index.
void LinkedList::insert(int index, int value) {
    ...
}
```

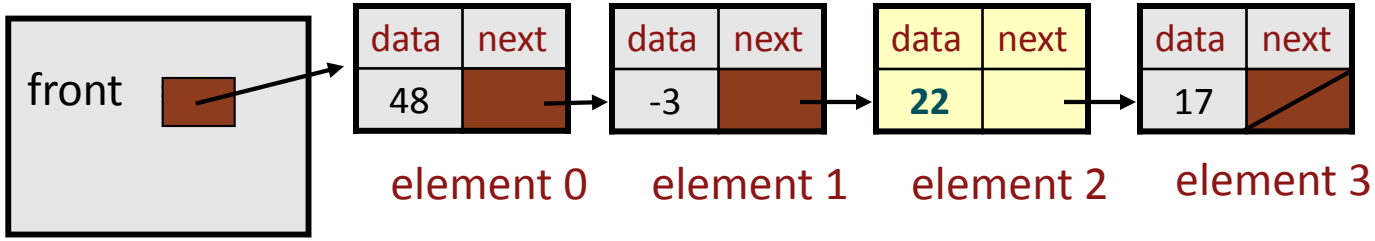


Inserting into a list

Before inserting element at index 2:



After:



Q: How many times to advance `current` to insert at index *i* ?

A. *i* - 1 **B.** *i* **C.** *i* + 1 **D.** none of the above

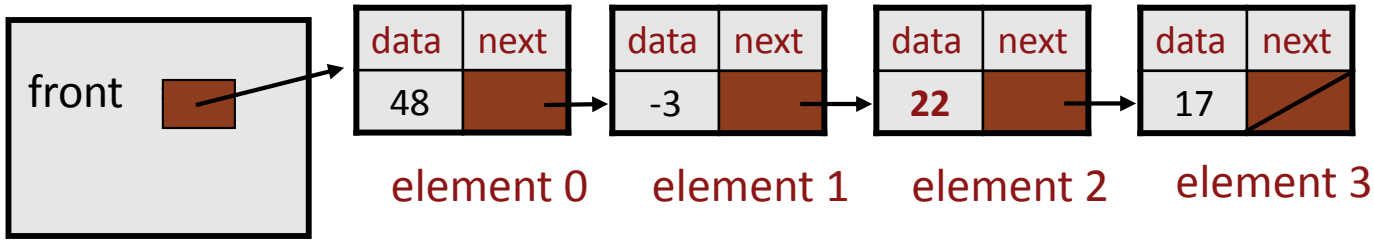
Code for insert

```
// Inserts the given value at the given index.
// Precondition: 0 <= index <= size()
void LinkedList::insert(int index, int value) {
    if (index == 0) {
        // adding to an empty list
        front = new ListNode(value, front);
    } else {
        // inserting into an existing list
        ListNode* current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current->next;
        }
        current->next =
            new ListNode(value, current->next);
    }
}
```

Implementing remove

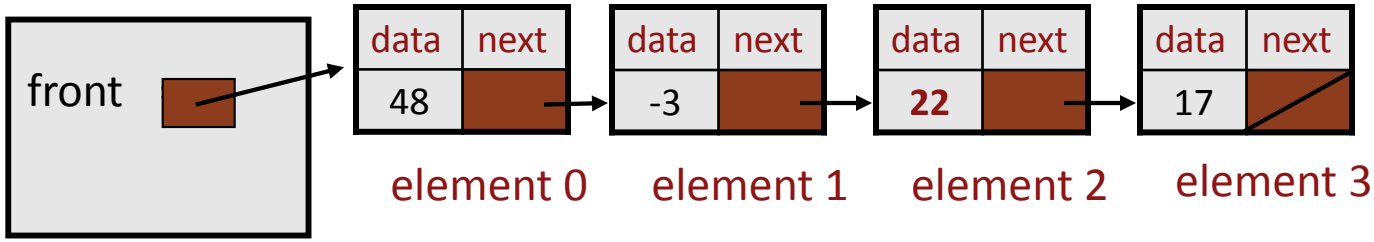
```
// Removes value at given index from list.
void LinkedList::remove(int index) {
    ...
}
```

- What pointer(s) must be changed to remove a node from a list?
- What different cases must we consider?

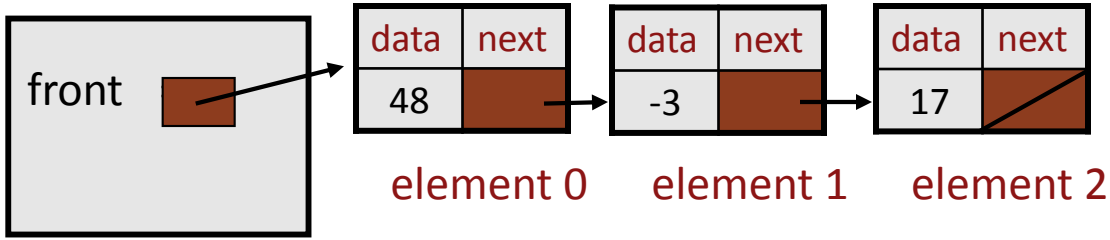


Removing from a list

Before removing element at index 2:

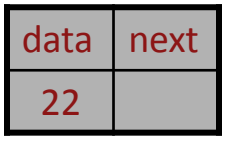


After:



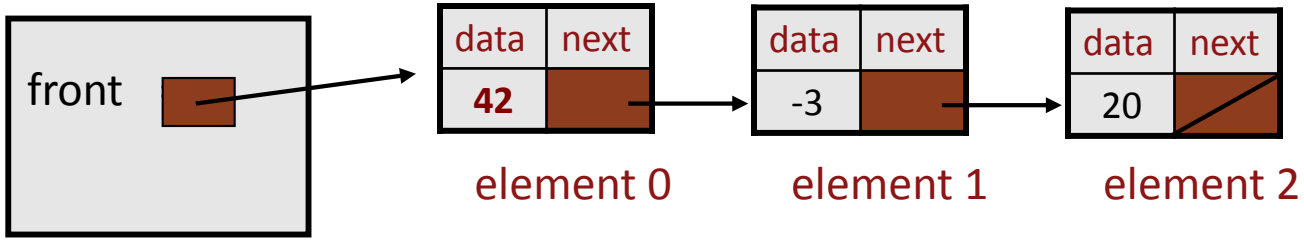
Where should `current` be pointing?

How many times should it advance from `front`?

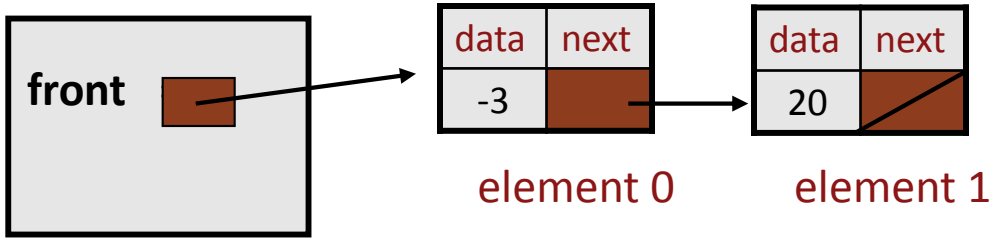


Removing from front

Before removing element at index 0:

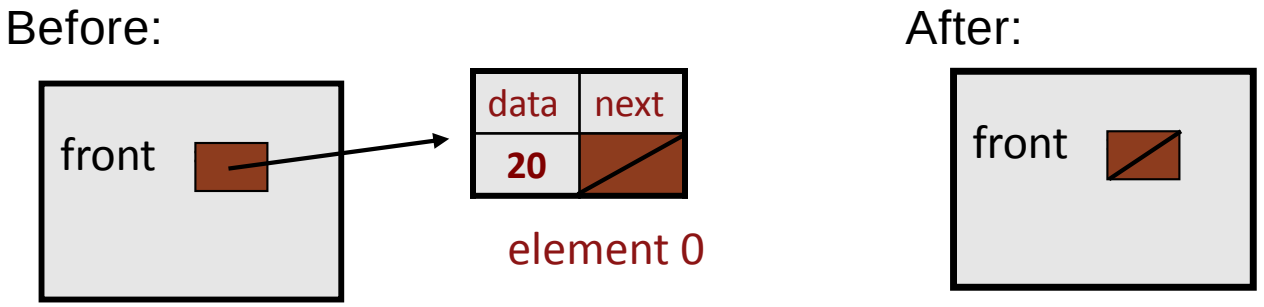


After:



To remove the first node, we must change `front`.

Removing the only element



- We must change the front field to store `NULL` instead of a node.
- Do we need a special case to handle this?

Code for remove

```
// Removes value at given index from list.
// Precondition: 0 <= index < size()
void LinkedList::remove(int index) {
    ListNode* trash;
    if (index == 0) {    // removing first element
        trash = front;
        front = front->next;
    } else {             // removing elsewhere in the list
        ListNode* current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current->next;
        }
        trash = current->next;
        current->next = current->next->next;
    }
    delete trash;
}
```

Other list features

Add the following public members to the `LinkedList`:

- `size()`
- `isEmpty()`
- `set(`***`index`***`, `***`value`***`)`
- `clear()`
- `toString()`

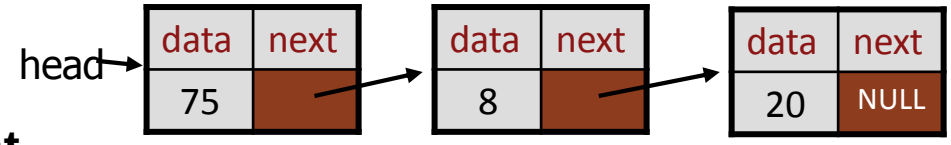
Add a `size` field to the list to return its size **more efficiently**.

Add preconditions and exception tests as appropriate.

Priority Queue

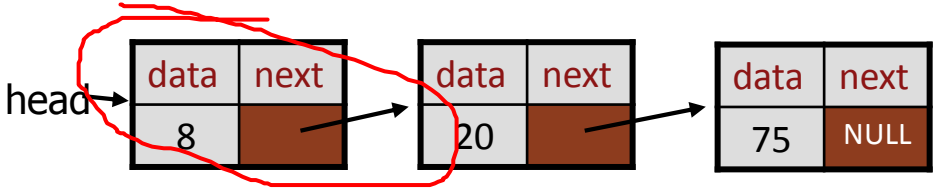
Emergency Department waiting room operates as a priority queue: patients are sorted according to priority (urgency), not “first come, first serve” (in computer science, “first in, first out” or FIFO).

Some priority queue implementation options



Unsorted linked list

- Insert new element in front
- **Remove by searching list for highest-priority item**

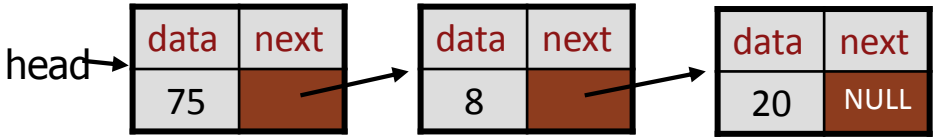


Sorted linked list

- Always insert new elements where they go in priority-sorted order
- **Remove from front (will be highest-priority because sorted)**

Priority queue implementations

Unsorted linked list

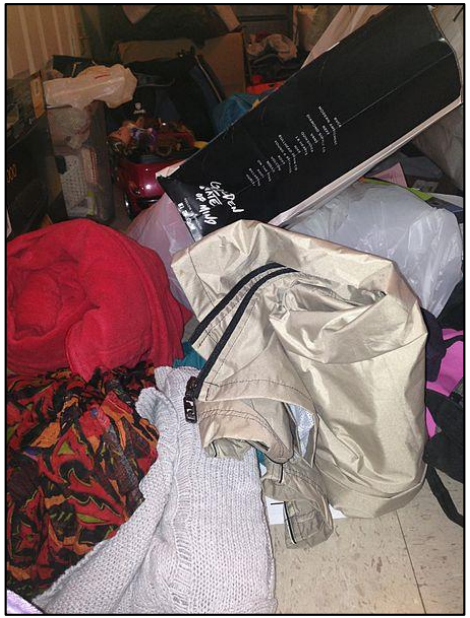


Add is FAST

- Just throw it in the list at the front
- **O(1)**

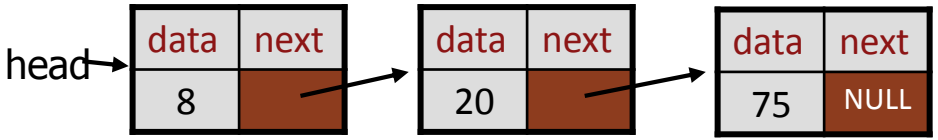
Remove/peek is SLOW

- Hard to find item the highest priority item—could be anywhere
- **O(N)**



Priority queue implementations

Sorted linked list



Add is SLOW

- Need to step through the list to find where item goes in priority-sorted order
- **O(N)**

Remove/peek is FAST

- Easy to find item you are looking for (first in list)
- **O(1)**

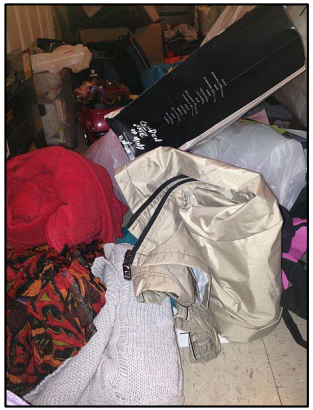
Image is in the public domain.
http://commons.wikimedia.org/wiki/File:Wall_Closet.jpg

Priority queue implementations

We want the best of both

Fast add AND fast remove/peek

We will investigate trees as a way to get the best of both worlds



Fast add

+

Fast remove/peek

=