Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Topics:

- **Wednesday: Link Nodes**
  - The `LinkNode` struct
  - Chains of link nodes
  - `LinkNode` operations
- **Today: Link Lists**
  - Providing a cohesive interface to chains of link nodes with a `LinkedList` class
  - `LinkedList` class implementation
  - `LinkedList` methods

Linked Nodes

A GREAT WAY TO EXERCISE YOUR POINTER UNDERSTANDING

FIRST RULE OF LINKED NODE/LISTS CLUB:

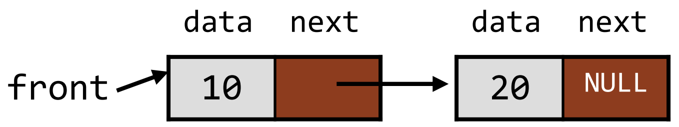
DRAW A PICTURE OF LINKED LISTS

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

List code example: Draw a picture!

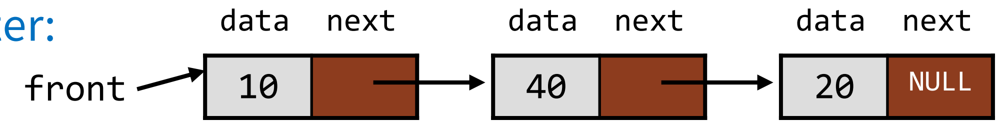
```
struct LinkNode {
   int data;
   LinkNode* next;
};
```

Before:

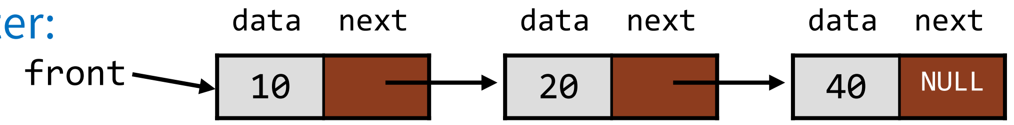


```
front->next->next = new LinkNode;
front->next->next->data = 40;
```

A. After:



B. After:



C. Using `next` that is `nullptr` gives an error

D. Other/none/more than one

Stanford University

Linked List Data Structure

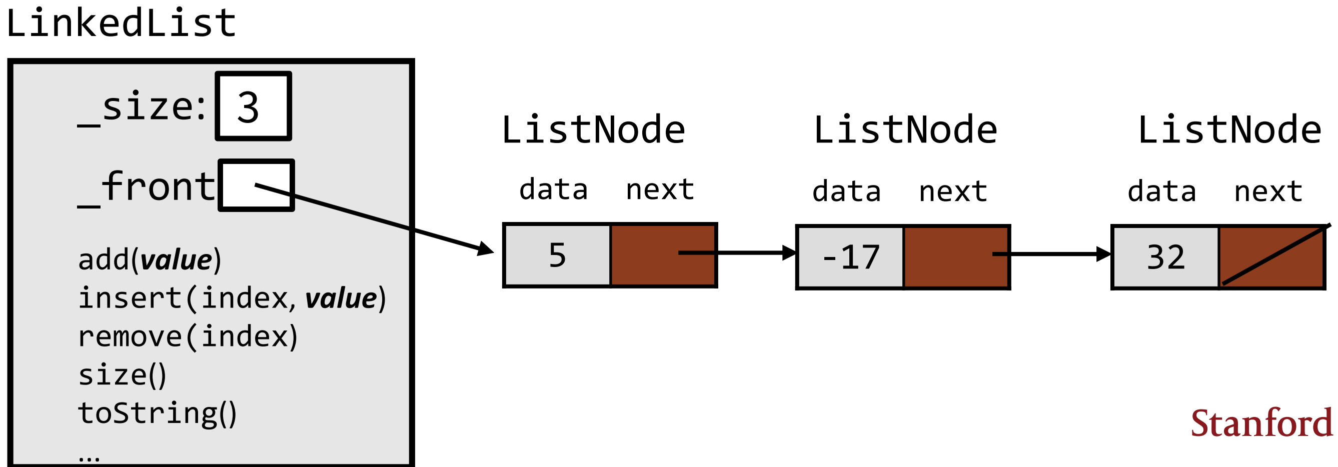
PUTTING THE `LISTNODE` TO USE

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

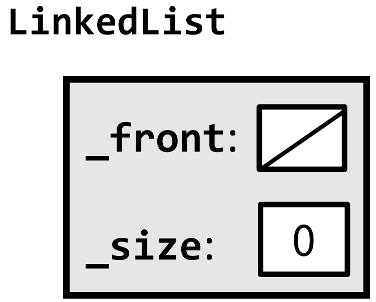
- The list is internally implemented as a **chain of linked nodes**
  - The LinkedList keeps a pointer to its _front node as a field
  - nullptr is the end of the list; a nullptr in _front signifies an empty list



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;
    int       _size;
};
```



Our first `LinkedList` Class Method

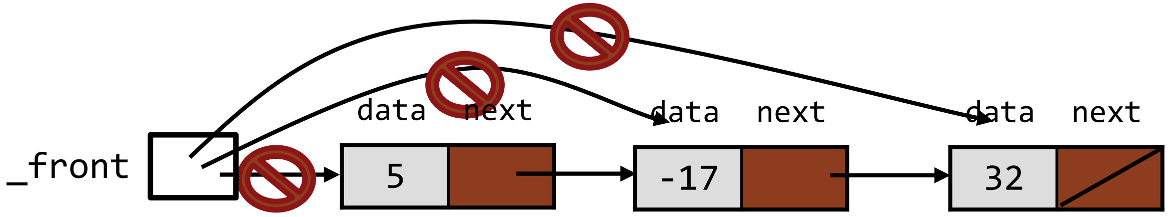
TOSTRING()

Traversing the list for `toString() // BUG VERSION`

- What's **wrong** with this approach to **traverse** the list?

```
string contents = "{";
while (_front != nullptr) {
    contents += (integerToString(_front->data) + ", ";
    _front = _front->next;    // move to next node
}
contents += "}";
```

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

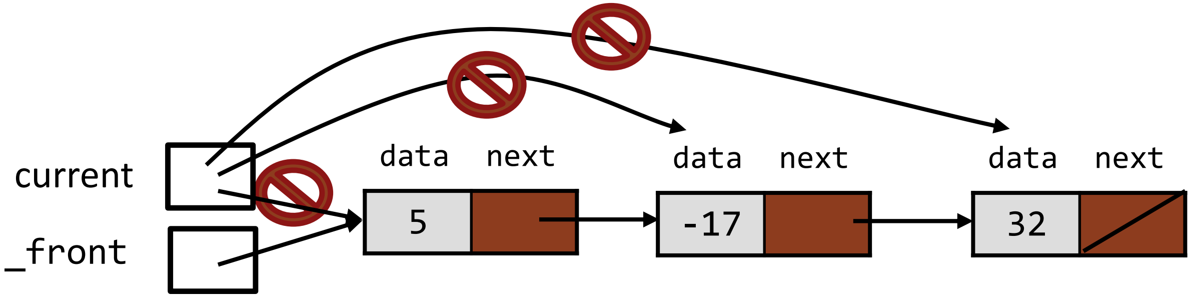


Traversing a list (12.2) (bug fixed version)

- The correct way to traverse the list:

```
string contents = "{";
ListNode* current = _front;
while (current != nullptr) {
    contents += (integerToString(current->data) + ", ";
    current = current->next;    // move to next node
}
contents += "}"; // TODO: should fix this to remove last extra , in list
```

- Changing the temporary variable `current` does not damage the list.



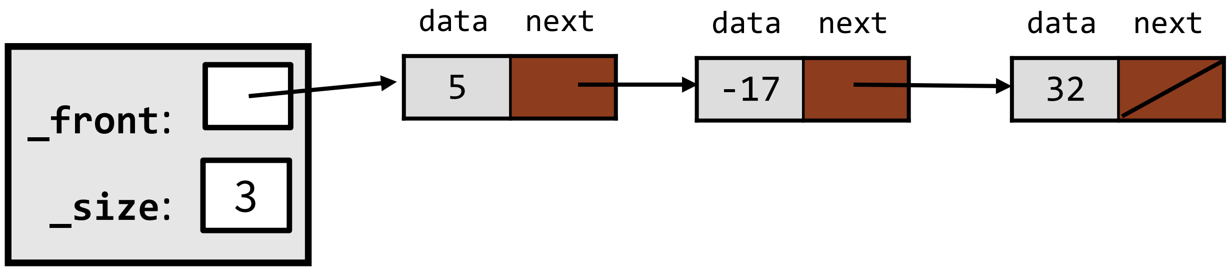
`LinkedList` Class `add()` Method

METHOD NUMBER TWO

Implementing add

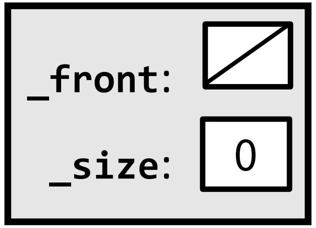
```
// Appends the given value to the end of the list.
void LinkedList::add(int value) {
    ...
}
```

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

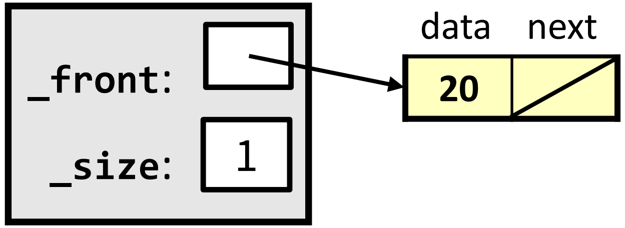


Case 1: Add to empty list

Before adding 20:



After:

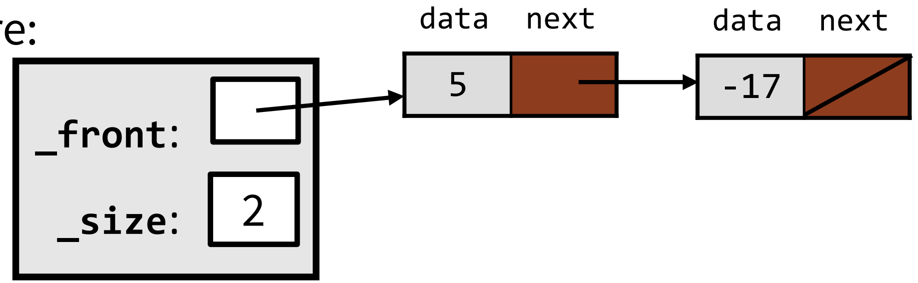


- We must create a new node and attach it to the list.
- For an empty list to become non-empty, we must change `_front`.

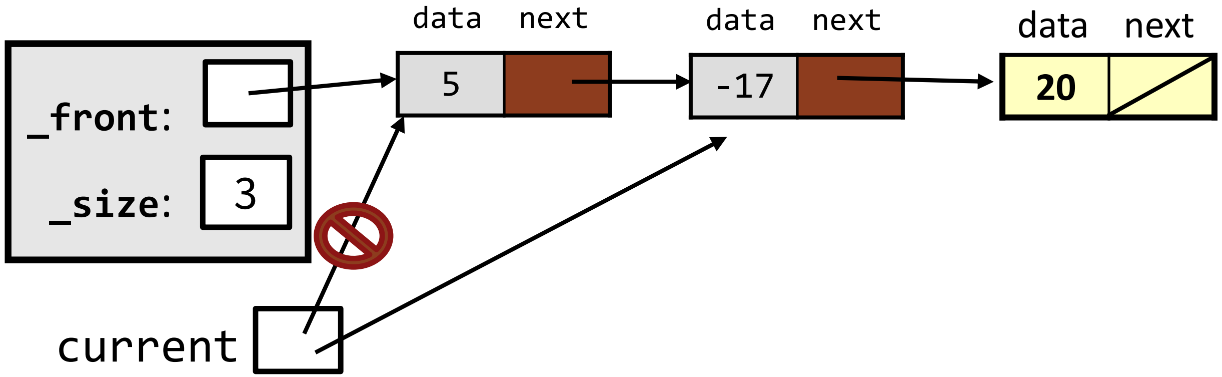
Case 2: Non-empty list

Before adding value 20 to end of list:

Before:



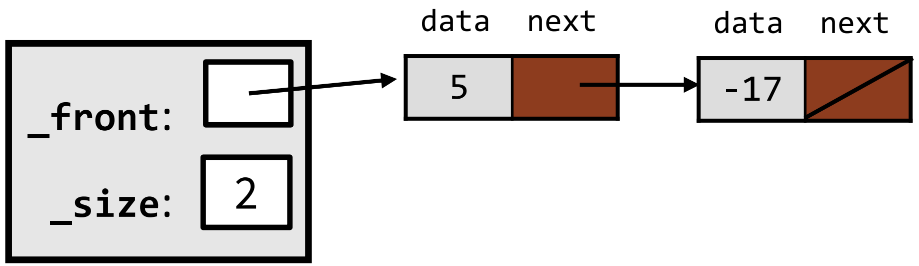
After:



Remember to use a temporary pointer for traversal to end

Managing our temporary pointer, `current`

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



- Think about where `current` should be pointing, to add 20 at the end

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

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

B. `while (_front != nullptr) { ...`

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

D. `while (_front->next != nullptr) { ...`

Code for add

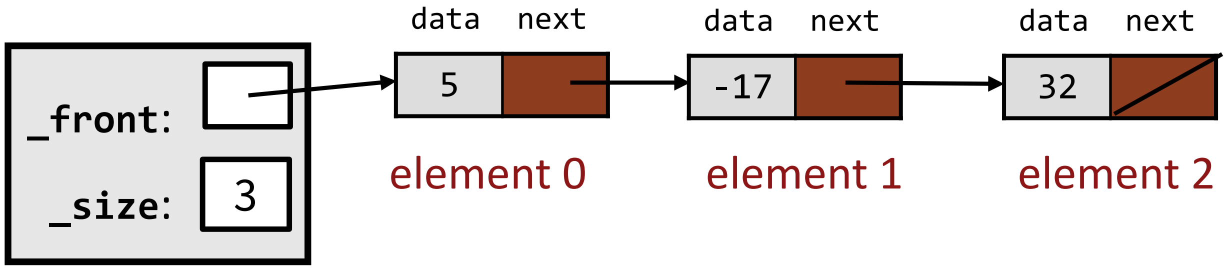
```
// (in linkedlist.cpp)
// Adds the given value to the end of the list.
void LinkedList::add(int value)
{
    if (_front == nullptr) {
        // adding to an empty list
        _front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ListNode* current =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

More `LinkedList` Class Methods!

GET(), INSERT(), REMOVE()

Implementing get

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



- **Fun tip:** we've been using a while loop to traverse our linked list (to go to the end for add). But for insert at a specified index, **a for loop** is handy to get us there in a defined number of steps

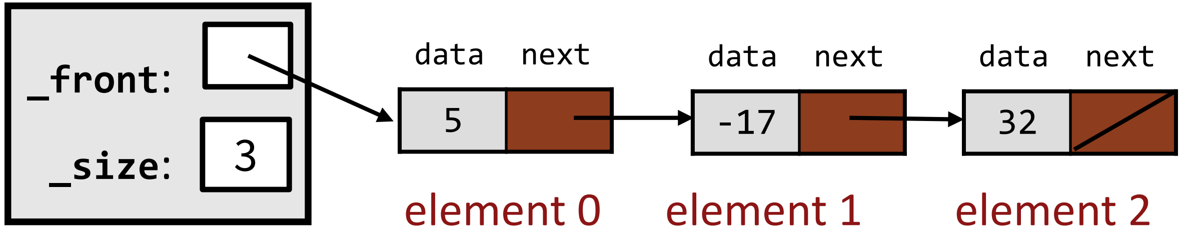
Code for get

```
// Returns value in list at given index.
int LinkedList::get(int index)
{
    if (index >= size()) {
        error("Index out of bounds!");
    }
    ListNode* current = _front;
    for (int i = 0; i < index; i++) {
        current = current->next;
    }
    return current->data;
}
```

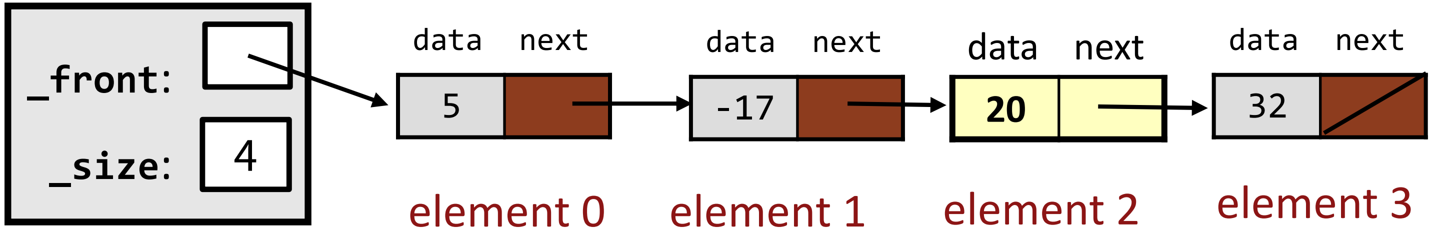
Implementing insert

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

Before `insert()` where `index = 2`, `value = 20 :`



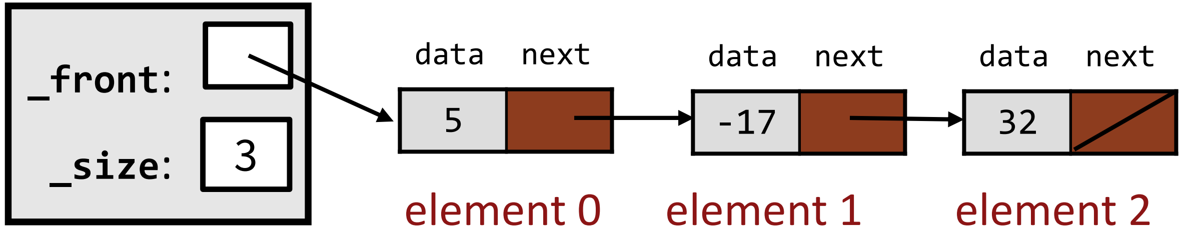
After:



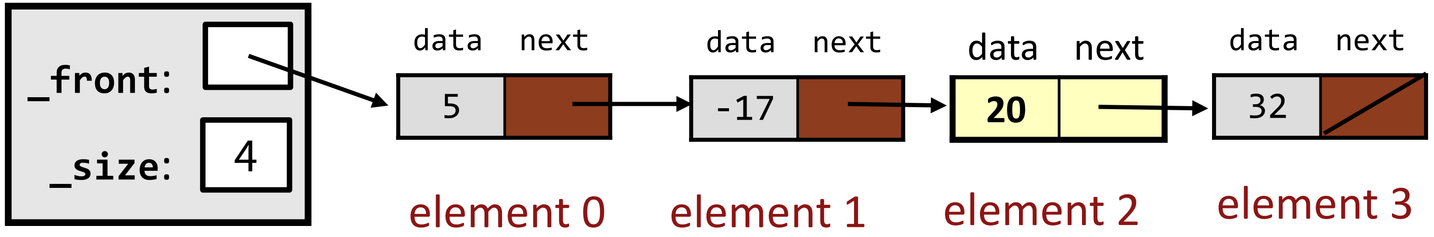
Stanford University

Inserting into a list

Before `insert()` where `index = 2`, `value = 20` :



After:



- **Your Turn:** If `current` starts out equal to `_front`, how many times do we <u>advance</u> `current` (in the for loop) to prepare for insert?

  A. `index – 1` times **B.** `index` times **C.** `index + 1` times **D.** Other

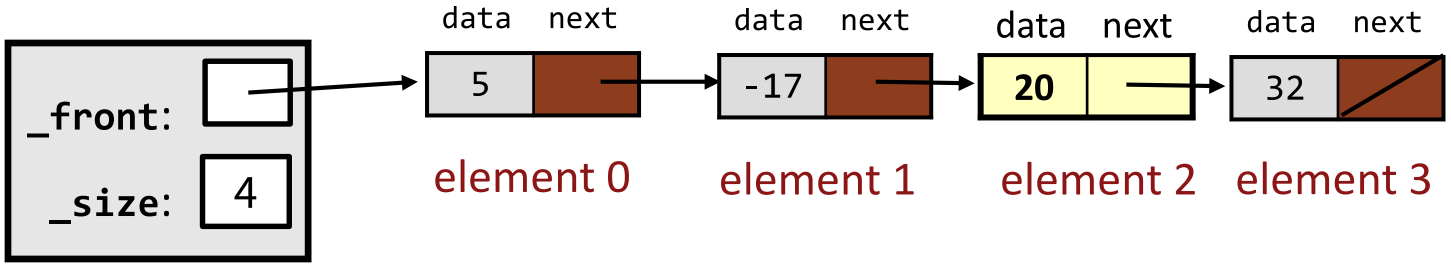
Stanford University

Implementing remove

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

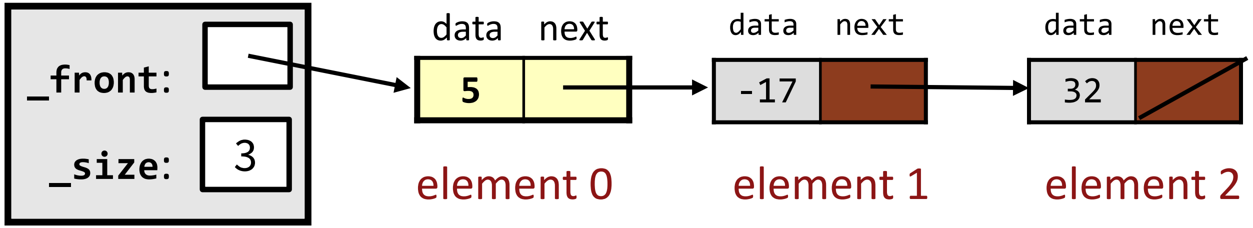
- What pointer(s) must be changed to remove a node from a list?
- What different cases must we consider?

Before remove with `index = 2`

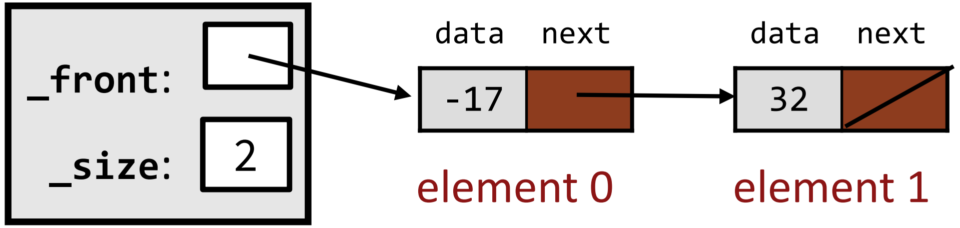


Case 1: Removing from front (index 0)

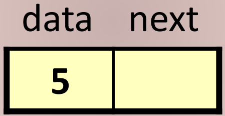
Before removing element at index 0:



After:



Be sure to delete this!



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

Code for remove

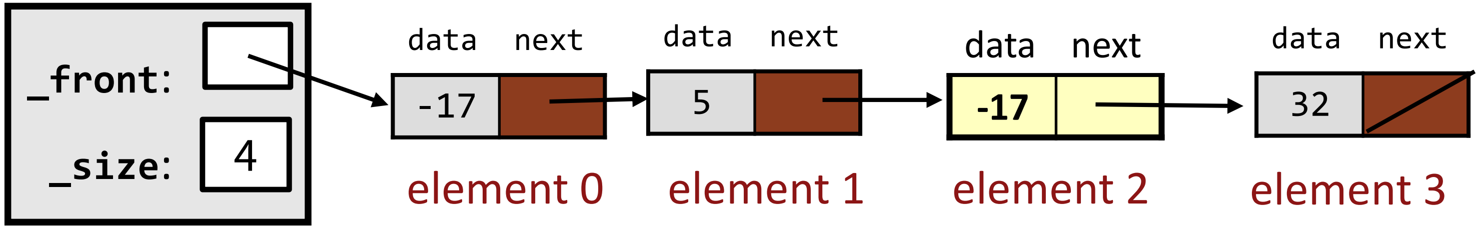
```
// Removes value at given index from list.
void LinkedList::remove(int index) {
   if (index >= size()) {
       error("Index out of bounds!");
   }
    ListNode* trash = nullptr;
    // removing first element
    if (index == 0) {
        trash = _front;
        _front = _front->next;
    // removing elsewhere in the list
    } else {
        // left for the reader ☺

    }
    delete trash;
    size--;
}
```

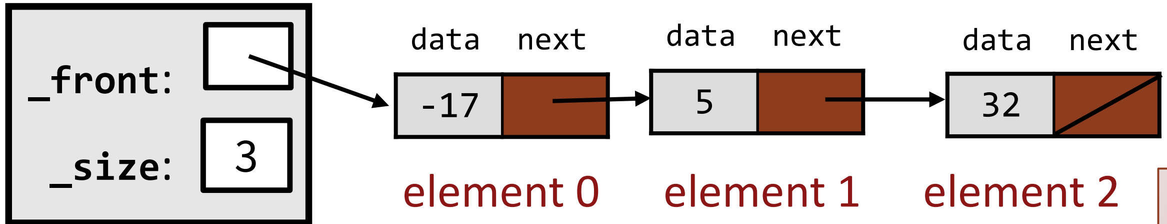
Stanford University

Case 2: Removing from "middle" of list (ex: index 2)

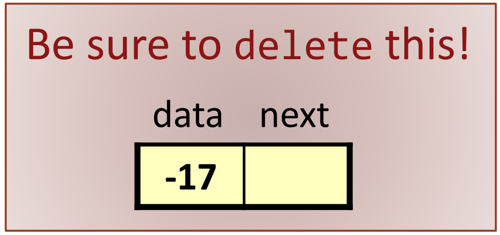
Before removing element at `index = 2`:



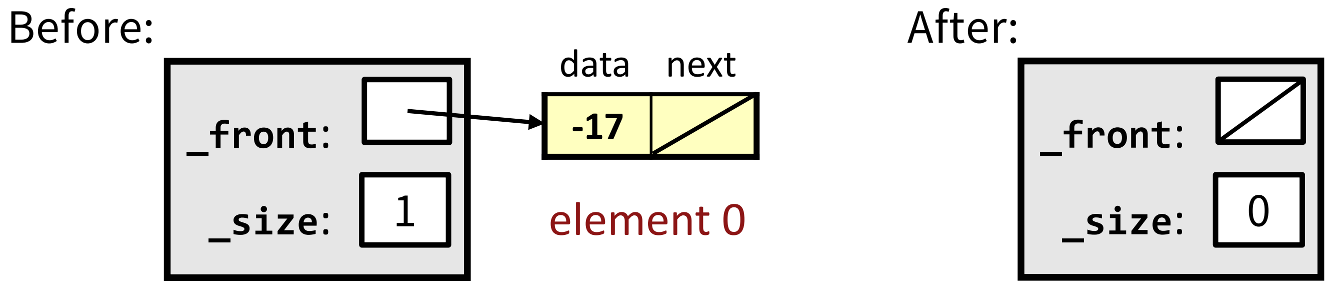
After:



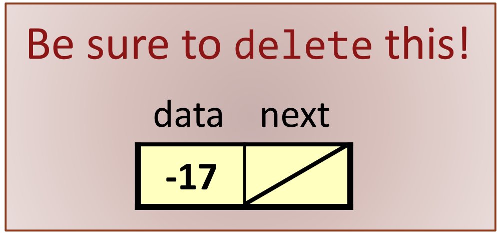
- Where should `current` be pointing?
- How many times should it advance from `_front`?



Case 3 (?): Removing the only element



- We must change the `_front` field to store `nullptr` instead of pointing to a node.
- Do we *really* need a special case to handle this?



Other list features

A nice `LinkedList` class will also want to have the following public member functions:

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