Topics:

- **Link Nodes**
  - LinkNode struct
  - Chains of link nodes
  - LinkNode operations

- **Link Lists**
  - Providing methods for operations on chains of link nodes
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!
Before:

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

A. After:

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

B. After:

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

C. Using “next” that is NULL gives error

D. Other/none/more than one
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

```
size: 3
front: 
```

```
add(value)
insert(index, value)
remove(index)
size()
toString()
```

```
42  
```

```
-3  
```

```
17  
```

```
element 0  element 1  element 2
```

Stanford University
Traversing a list? (BUG version)

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

while (front != NULL) {
    cout << front->data << endl;
    front = front->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:

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

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

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

After:

- data 20
- next
- element 0

front
size 0

front
size 1
Case 2: **Non-empty list**

Before adding value 20 to end of list:

Before:

- front
- size 2

After:

- front
- size 3

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 != NULL) { ...  
B. while (front != NULL) { ...  
C. while (current->next != NULL) { ...  
D. while (front->next != NULL) { ...
Code for add

// (in linkedlist.cpp)
// 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);
    }
    size++;
}
Implementing get

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

front
size 3

data 42
next

data -3
next

data 17
next

element 0
element 1
element 2
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) {
    ...
}

- 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
Inserting into a list

Before inserting element at index 2:

<table>
<thead>
<tr>
<th>Front</th>
<th>Size</th>
<th>Data</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>48</td>
<td>-3</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>Front</th>
<th>Size</th>
<th>Data</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>48</td>
<td>-3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>22</td>
</tr>
</tbody>
</table>

Q: If current starts out equal to front, how many times do we advance current (in the for loop) to prepare for insert at index i?

A. \(i - 1\) times  
B. \(i\) times  
C. \(i + 1\) times  
D. none of the above
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?
Case 1: Removing from front (index 0)

Before removing element at index 0:

To remove the first node, we must change front.

After:

Be sure to delete 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;
        delete trash;
    } else { // removing elsewhere in the list
        // left for the reader 😊
    }
    size--;
Case 2: Removing from “middle” of list (ex: index 2)

Before removing element at index 2:

<table>
<thead>
<tr>
<th>front</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>-</td>
</tr>
</tbody>
</table>

element 0  element 1  element 2  element 3

After:

<table>
<thead>
<tr>
<th>front</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>-</td>
</tr>
</tbody>
</table>

Be sure to delete this!

Where should current be pointing?
How many times should it advance from front?
Case 3 (?): Removing the only element

Before:

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

After:

Be sure to delete 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()