Outline

1. Learn how `LinkedIntList` is implemented
2. Learn about the different cases to deal with for `LinkedList`

LinkedIntList Fields

What fields does our `LinkedIntList` need?

A reference to the front of the list

```
LinkedIntList v1
public class LinkedIntList {
    private ListNode front;

    public LinkedIntList() {
        front = null;
    }
}
```

LinkedIntList toString()

Buggy toString()

```
public String toString() {
    String result = "[";
    ListNode current = this.front;
    while (current != null) {
        result += current.data + ", ";
        current = current.next;
    }
    return result + "]";
}
```

Our toString() puts a trailing comma. Fix it by stopping one early:

Fixed toString()

```
public String toString() {
    String result = "[";
    ListNode current = this.front;
    while (current != null) {
        result += current.data + ", ";
        current = current.next;
    }
    if (current != null) {
        result += current.data;
    }
    return result + "]";
}
```
Modify LinkedLists 4

Writing a LinkedList Method

1. Identify cases to consider...
   - Front/Empty
   - Middle
   - End
2. Draw pictures for each case
3. Write each case separately

```java
public int get(int index) {
    if (this.front == null) {
        return -1; // or null
    }
    int i = 0;
    ListNode current = this.front;
    while (current != null) {
        if (i == index) {
            return current.data;
        }
        current = current.next;
        i++;
    }
    return -1; // or null
}
```

Linked.LinkedList add() (Empty Case) 5

Cases to consider:
- Add to empty list
- Add to non-empty list

Add To An Empty List

What does an empty list look like?

```java
public void add(int value) {
    if (this.front == null) {
        this.front = new ListNode(value);
    } else {
        ListNode current = this.front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new ListNode(value);
    }
}
```

Linked.LinkedList add() (Non-empty Case) 6

Add To A Non-Empty List

Consider a non-empty list:

```java
public void add(int value) {
    ListNode current = this.front;
    while (current.next != null) {
        current = current.next;
    }
    current.next = new ListNode(value);
}
```

Linked.LinkedList get() 9

```java
public int get(int index) {
    if (this.front == null) {
        return -1; // or null
    }
    int i = 0;
    NodeList current = this.front;
    while (i < index) {
        current = current.next;
        i++;
    }
    return current.data;
}
```

Working with LinkedLists 8

There are only two ways to modify a LinkedList:

- Change front
- Change current.next for some ListNode, current

Setting “current” does NOTHING!
Some LinkedList Tips!

- Be able to deal with before-and-after ListNode pictures
- Know how to loop through a LinkedList
  - Use a while loop.
  - Don’t forget to create a ListNode current variable so we don’t destroy the original list.
  - Don’t forget to update the current variable.
- They both have the same functionality (add, remove, etc.)
- But they’re implemented differently (array vs. ListNode)
- With LinkedLists, you often have to stop one node before the one you want.
- DO NOT start coding LinkedList problems without drawing pictures first.
**What Are We Doing Again?**

**What Are We Doing...?**

We're building an alternative data structure to an ArrayList with different efficiencies.

---

**Today's Main Goals:**

- Get more familiarity with LinkedLists
- Write more LinkedList methods
- Learn how to "protect" against NullPointerExceptions

---

**A New LinkedList Constructor**

**First Attempt**

```
public LinkedList(int n) {
    /* Current State */
    ListNode current = this.front;
    for (int i = 1; i <= n; i++) {
        current = current.next;
    }
}
```

Remember, to edit a LinkedList, we MUST edit one of the following:
- front, or
- node.next (for some ListNode node)

In our code above, we edit current, which is neither.

---

**A New LinkedList Constructor**

**Second Attempt**

```
public LinkedList(int n) {
    /* Current State */
    if (n > 0) {
        this.front = new ListNode(1);
        ListNode current = this.front;
        for (int i = 1; i <= n; i++) {
            current = current.next;
        }
    }
}
```

What kind of loop should we use?

A for loop, because we have numbers we want to put in the list.

What cases should we worry about?

We're creating the list; so, there aren't really "cases".
This other solution works by going backwards. Before, we were editing the next fields. Here, we edit the front field instead.

```java
Different Solution!
public void addSorted(int value) {
    if (current.next.data < value) {
        current = current.next;
    }
}
```

We fell off the end of the LinkedList.

---

### Implementing addSorted

**addSorted**

Write a method addSorted(int value) that adds value to a sorted LinkedList and keeps it sorted. For example, if we call addSorted(10) on the following LinkedList, we would get:

```
front → 32 → 5
```

As always, we should approach this by considering the separate cases (and then drawing pictures):

- We’re supposed to insert at the front
- We’re supposed to insert in the middle
- We’re supposed to insert at the back

---

#### Case: Middle

**An Incorrect Solution**

```
public void addSorted(int value) { //Say value = 10...
    ListNode current = this.front;
    while (current.data < value) {
        current = current.next;
    }
    //the while loop continues...
}
```

Uh Oh! We went too far! We needed the next field BEFORE us.

---

#### Case: End

**Adding At The End?**

```
public void addSorted(int value) { //Say value = 40...
    ListNode current = this.front;
    while (current.next.data < value) {
        current = current.next;
    }
    current.next = new ListNode(value, null);
    current.next.next = null;
}
```

We fell off the end of the LinkedList.

Idea: Make sure current.next exists.

---

#### Case: End

**Adding At The End?**

```
public void addSorted(int value) {
    ListNode current = this.front;
    while (current.next.data < value) {
        current = current.next;
    }
    current.next = new ListNode(value, null);
    current.next.next = null;
}
```

---

#### Case: End

**A Real Fix!**

```
public void addSorted(int value) {
    ListNode current = this.front;
    while (current.next.data < value) {
        current = current.next;
    }
    current.next = new ListNode(value, null);
    current.next.next = null;
}
```

---

#### Case: End

**A Fix?**

```
public void addSorted(int value) {
    ListNode current = this.front;
    if (current.next.data > value) { //if we are making a check for current.next, we must be sure that current is not null.
        while (current.next.data < value) { //Since we want to keep going here, the check must be made in the while loop.
            current = current.next;
        }
    }
    current.next = new ListNode(value, null);
}
```
Case: Beginning

Our current code only sets current to a new ListNode. Importantly, this never updates front; so, we lose the new node.

Adding At The Beginning?

```
public void addSorted(int value) {
    //Say value = -10...
    if (value < front.data) {
        ListNode next = front;
        front = new ListNode(value, next);
    } else {
        while (current.next != null && current.next.data < value) {
            current = current.next;
        }
        ListNode next = current.next;
        current.next = new ListNode(value, next);
    }
}
```

Have we covered all of our cases now?

Protecting Our Tests!

With LinkedList code, every time we make a test (if, while, etc.), we need to make sure we’re protected. Our current code is:

```
public void addSorted(int value) {
    if (value < front.data) {
        front = new ListNode(value, next);
    } else {
        while (current.next != null && current.next.data < value) {
            current = current.next;
        }
        current.next = new ListNode(value, next);
    }
}
```

We’re “protected” if we know we won’t get a NullPointerException when trying the test. So, consider our tests:
- value < front.data
- current.next != null && current.next.data < value

So, Are We Protected?

Some LinkedList Tips!

- Make sure to try all the cases:
  - Empty List
  - Front of Non-empty List
  - Middle of Non-empty List
  - Back of Non-empty List

- To Edit a LinkedList, the assignment must look like:
  - this.front = <something>;
  - node.next = <something>;

- Protect All Of Your Conditionals! Make sure that nothing can accidentally be null.

- When protecting your conditionals, make sure the less complicated check goes first.