CSE 143
Computer Programming II
You love computers and play with them...

then you went college learn many... things.

you learn that programming is a state of the mind...

finally you learn the language that every programmer in knows: blasphemy.

Work you piece of $#!&;!!!
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
1. Get more familiarity with changing LinkedLists

2. Write more methods in the LinkedList class

3. Protecting Against NullPointerExceptions
A New LinkedList Constructor

New Constructor
Create a constructor

```java
public LinkedList(int n)
```

which creates the following LinkedList, when given $n$:

```
1 → 2 → 3 → ... → n
```

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”.
A New LinkedList Constructor

First Attempt

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

    if (n > 0) {
        // n is at least 1...
        this.front = new ListNode(1);
    
    ListNode current = this.front;

    for (int i = 1; i <= n; i++) {
        current.next = new ListNode(i);

        current = current.next;
    }
    
}
This other solution works by going backwards. Before, we were editing the next fields. Here, we edit the front field instead:

```java
public LinkedList(int n) {
    /* Current State */
    front

    for (int i = n; i > 0; i--) {
        ListNode next = this.front;

        this.front = new ListNode(i, next);
    }

    /* Second time through the loop (for demo)... */
    //ListNode next = this.front;

    //this.front = new ListNode(i, next);
}
```
1. Get more familiarity with changing LinkedLists
2. Write more methods in the LinkedList class
3. Protecting Against NullPointerExceptions
Implementing `addSorted`  

Write a method `addSorted(int value)` that adds value to a sorted `LinkedIntList` and **keeps it sorted**. For example, if we call `addSorted(10)` on the following `LinkedIntList`,

```
front
\downarrow
-8 \rightarrow 4 \rightarrow 32 \rightarrow 35
```

We would get:

```
front
\downarrow
-8 \rightarrow 4 \rightarrow 10 \rightarrow 32 \rightarrow 35
```

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
An Incorrect Solution

```java
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.
```java
public void addSorted(int value) { //Say value = 10...

    ListNode current = this.front;

    while (current.next.data < value) {
        current = current.next;
    }

    ...the while loop STOPS now...

    ListNode next = current.next;

    current.next = new ListNode(value, next);
}
```

Does this cover all the cases?
```java
public void addSorted(int value) { //Say value = 40...
    ListNode current = this.front;
    while (current.next.data < value) {
        current = current.next;
    }
    //the while loop continues...
    current.next.data → NullPointerExceptio!!
}
```

We fell off the end of the LinkedList.
Idea: Make sure `current.next` exists.
### Adding At The End?

```java
class ListNode {
    int data;
    ListNode next;
}

public void addSorted(int value) {
    ListNode current = this.front;
    /* 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 on going here,
         * the check must be made in the while loop.
         * current = current.next;
        }
    }
}
```

### A Fix?

```java
class ListNode {
    int data;
    ListNode next;
}

public void addSorted(int value) {
    ListNode current = this.front;
    /* The extra check here is useless...we’ve already checked
     * current.next by the time we get to it. */
    while (current.next.data < value && current.next != null) {
        current = current.next;
    }
}
```

### A Real Fix!

```java
class ListNode {
    int data;
    ListNode next;
}

public void addSorted(int value) {
    ListNode current = this.front;
    while (current.next != null && current.next.data < value) {
        current = current.next;
    }
}
```
Our current code only sets current to a new ListNode. Importantly, this never updates front; so, we lose the new node.

```java
public void addSorted(int value) {
    //Say value = -10...
    ListNode next = front;
    if (value < front.data) {
        -10 < -8 \rightarrow true
        front = new ListNode(value, next);
    } else {
        ...
    }
}
```

Have we covered all of our cases now?
1. Get more familiarity with changing LinkedLists

2. Write more methods in the LinkedList class

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

```java
public void addSorted(int value) {
    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);
    }
}
```

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?
Nope! What happens if `front == null`? We try to get the value of `front.data`, and get a `NullPointerException`. The fix:

```
public void addSorted(int value) {
    if (front == null || 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);
    }
}
```

Helpfully, this fix actually handles the empty list case correctly!
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>;`, or
  - `node.next = <something>;` (for some ListNode node in the list)

- 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.