CSE 143X

Accelerated Computer Programming I/II
Linked Lists I
1. Learn how LinkedIntList is implemented

2. Learn about the different cases to deal with for LinkedLists
Outline

1. Learn how LinkedIntList is implemented
2. Learn about the different cases to deal with for LinkedLists
No generics (only stores ints)

Fewer methods: add(value), add(index, value), get(index), set(index, value), size(), isEmpty(), remove(index), indexOf(value), contains(value), toString()

This is the same idea as when we implemented ArrayIntList!
What fields does our LinkedIntList need?

A reference to the front of the list

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

    public LinkedIntList() {
        front = null;
    }

    // ... other methods...
}
```
LinkedIntList toString()

Buggy toString()

```java
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()

```java
public String toString() {
    String result = "[";
    ListNode current = this.front;
    while (current != null && current.next != null) {
        result += current.data + ", ";
        current = current.next;
    }
    if (current != null) {
        result += current.data;
    }
    return result + "]";
}
```
1. Learn how LinkedIntList is implemented
2. Learn about the different cases to deal with for LinkedLists
Modifying LinkedLists

Writing a LinkedList Method

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

front middle end

```
1 -> 2 -> 3 -> ... -> 9 -> 10 -> 11 -> ... -> 42
```
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 the list is empty... */
    if (this.front == null) {
        this.front = new ListNode(value);
    }
    /* Other Cases ... */
}
```
Add To A Non-Empty List

Consider a non-empty list:

```
1 -> 2 -> 3 -> ... -> 100
```

```
/* Idea: We want to change the red arrow.
Loop until we’re at the last node. */
```

```
ListNode current = this.front;
```

```
while (current != null) {
    current = current.next;
}
```

```
current = new ListNode(value);
```

```
Add To A Non-Empty List (Fixed)

Consider a non-empty list:

```
front
1 --> 2 --> 3 --> ... --> 100
```

```java
/* Idea: We want to change the red arrow.
   Loop until we’re at the node before the last node */
ListNode current = this.front;

while (current.next != null) {
    current = current.next;
}

current.next = new ListNode(value);
```
There are only two ways to modify a LinkedList:

- **Change front**
  
  ![Diagram of changing front]
  
  \[
  \begin{align*}
  \text{front} & \quad 1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots \text{changing front} \cdots \\
  ( & \quad [1,2,3]; \quad \cdots \text{changing front} \cdots \quad [2,3])
  \end{align*}
  \]

- **Change current.next for some ListNode, current**
  
  ![Diagram of changing current.next]
  
  \[
  \begin{align*}
  \text{front} & \quad 1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots \text{changing current.next} \cdots \\
  ( & \quad [1,2,3]; \quad \cdots \text{changing current.next} \cdots \quad [1,3])
  \end{align*}
  \]

Settting “current” does NOTHING!
public int get(int index) {
    ListNode current = front;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
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. ListNodes)
- With LinkedLists, you often have to stop one node before the one you want.
- DO NOT start coding LinkedList problems without drawing pictures first.
You love computers and play with them...
then you went to college and learned 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 $#!&$!!!

Vlad Bazan (c) 2004
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
Outline

1. Get more familiarity with changing LinkedLists

2. Write more methods in the LinkedList class

3. Protecting Against NullPointerExceptions
New Constructor

Create a constructor

```java
public LinkedIntList(int n)
which creates the following LinkedIntList, when given \( n \):
```

![Diagram of a linked list with nodes from 1 to 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 */
    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);
}
```
Outline

1. Get more familiarity with changing LinkedLists
2. Write more methods in the LinkedList class
3. Protecting Against NullPointerExceptions
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
   ↓
  -8 → 4 → 32 → 35
```

We would get:

```
   front
   ↓
  -8 → 4 → 10 → 32 → 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
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.
public void addSorted(int value) { //Say value = 10...

ListNode current = this.front;

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

ListNode next = current.next;

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

Does this cover all the cases?
public void addSorted(int value) {
    // Say value = 40...

    ListNode current = this.front;

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

    ...the while loop continues...

    ...AND IT KEEPS ON GOING...

    current.next.data → NullPointerException!!!
}

We fell off the end of the LinkedList.

Idea: Make sure current.next exists.
Adding At The End?

```java
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
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
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...
    if (value < front.data) {
        ListNode next = front;
        front = new ListNode(value, next);
    } else {
        ...
    }
}
```

Have we covered all of our cases now?
Outline

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.