Lecture 9: Advanced Linked Lists

- Your assignment: Assassin
  - Demo
- Warning: this assignment is hard - linked lists are hard
  - Don't procrastinate on this assignment
  - As a medium hint, we'll consider another problem
  - LinkedIntList doesn't really need a second constructor like ArrayIntList (no capacity)
  - But what if we wanted a constructor that takes an integer *n* and counts down?

    `LinkedIntList(10) -----> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`
  - Well, let's consider the simplest case first
    - What if n is 0? What do we do? (draw)

```
ListNode temp;
temp = new ListNode(0);
front = temp;
```
  - Now what if n is 1? We start with what we had before, but need to add a 1 onto the front

```
temp = new ListNode(1, front);
front = temp;
```
  - Now what if n is 2? We start with what we had before, but add a 2 to the front

```
temp = new ListNode(2, front);
front = temp
```
  - What's the pattern? What do we want to do repeatedly (i.e. in a loop?)

```
for (int i = 1; i <= n; i++) {
    ListNode temp = new ListNode(i, front);
    front = temp;
}
```
  - What about the front case?
    - Outside the loop
    - But actually can go inside the loop - it doesn't hurt to use "null" in the 2-arg constructor
  - We actually don't need the temporary variable

```
front = new ListNode(i, front);
```
    - This might make you concerned - front on both sides!
    - The right-hand side is evaluated first - the object creation
    - Then the left-hand side (assignment) is evaluated
    - Equivalent to "x = x + 1"
    - It does work, but if you're uncomfortable stick with the 2-line version

```
public LinkedIntList(int n) {
    front = null;
    for (int i = 0; i <= n; i++) {
        front = new ListNode(i, front);
    }
}
```

- Another problem: addSorted
  - Assuming the list is already in sorted (nondecreasing) order, adds a value to the list to preserve the ordering

```
// pre : list is in sorted (nondecreasing) order
// post: given value is added to the list so as to preserve
//       (nondecreasing) order, duplicates allowed
public void addSorted(int value) {
    ...
}
```

  - This is a very complicated problem
- First: adding in the middle of the list
  - Ex. [2, 5, 12] and add 10
  - Where does it belong? Why?
  - How do we write this as a loop?
    - Does it belong in front of the 2? In front of the 5? In front of the 12?

```
ListNode current = front;
while (current.data < value)
    current = current.next;
```

  - This is the core of the right idea, but what's wrong?
    - Ends up positioning us at the wrong spot
    - Remember - have to change either FRONT or a ~.NEXT
    - So what do we need to change in this case? (the node w/5's .next)
  - Better:

```
ListNode current = front;
while (current.next.data < value)
    current = current.next;
```

  - Now we stop when our value is just barely less than the value we want to insert
  - Now we need to create a new node....

```
new ListNode(value, current.next)
```

  - ...and then link it into the list

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

  - (if you want, you can use a temporary variable)
- What if we want to add 13?
  - What will happen? (simulate) - NullPointerException
  - Problem: our code depends on the fact that there is a node with a value GREATER than the value we want to insert
  - So we need an extra test to make sure we don't run off the end:

```
while (current.next.data < value && current.next != null)
       ~~~~~~~~~~~~~~~~~~~~~~~~    ~~~~~~~~~~~~~~~~~~~
            sensitive test              robust test
```

  - This also doesn't work - because we'll still execute the first test and throw an exception
  - Combination of "sensitive" and "robust" tests
  - In order to make it work, reverse the tests
    - "Short-circuited evaluation" - Java stops as soon as the first test is FALSE

- Another case: what if we want to add 1 to the list?
  - What will happen? (simulate)
  - Will add AFTER the first node instead of before
  - Why is this different?
    - (must change FRONT rather than current.next)
  - OK, so in some cases we want to change front
    - Which cases?
    - What do we want to do in that case?
- So let's try it:
  - Create a new list, then call addSorted 2, 5, 12, 10, 1, 13
  - But throws a null pointer exception! uh oh
- This code will still break

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

  - What if the front is null? Then front.data test will fail
  - We need special test for the front being null
  - What do we do in this case?
    - Add to the front!
    - Extra case - BUT can actually be the same as adding before the first element
  - Test:

```
if (value <= front.data || front == null) {
```

  - But this is still wrong! Why?
    - Same example of a sensitive test - if front is null, front.data will throw an exception
  - Short-circuited evaluation works on OR tests as well
- So many cases to consider in this example!
  - middle
  - front
  - back
  - empty
  - (show parts of the tests pertaining to each case)
- Another approach: keep a "prev" pointer - "inchworm" approach

- Final 2 versions:

```
// pre : list is in sorted (non-decreasing) order
// post: given value inserted into list so as to preserve sorted order
public void addSorted(int value) {
    if (front == null || value <= front.data)
        front = new ListNode(value, front);
    else {
        ListNode current = front;
        while (current.next != null && current.next.data < value)
            current = current.next;
        current.next = new ListNode(value, current.next);
    }
}

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