Pre-Class Work 6: Linked Nodes with Loops

Linked Nodes with Loops [Background Reading]

So far, we have learned and practiced working with individual linked nodes. Now that we have a bit of experience with nodes and pointers, we can start think about problems where we might have a lot of linked nodes – enough to where we cannot "manually" move the pointer to where we want to go!

Instead, we can use while loops to help us advance through Linked Nodes, and help us stop where we want to.

Looping Paradigms

Reading

To start, let's consider the simplest case where we could use looping: we are given a bunch of linked nodes (we don't know how many) and we want to print out the value at every node

```
p -> 3 -> 2 -> 7 -> ... some  more nodes ... -> 6 -> 9 -> 5
```

Now, the question is: how do we get to that final node? Since we do not necessarily know how many nodes there are until the end, we can't use a `for` loop – this means that we need a `while` loop! Now, the question is, what condition are we going to use? As always, we want to think about the stopping condition – what is true about the end of the list?

Notice that if we follow the `next` field for each node, we will eventually get to `null` ! This gives us our reading loop pattern!

```
ListNode temp = p;
while (temp != null) {
    // Do something!
    temp = temp.next;
}
```

For printing out the value at every node, we would replace `// Do something!` with `System.out.println(temp.data);`

At the top of the while loop, we know that `temp != null` , so we know that we can access `temp.data` !

In general, we use this loop pattern every time we want to look at a list of Nodes *without* changing or adding any nodes!

Writing

Let's pretend we have the same list, but instead want to add a value to the end of the list. What would happen if we tried to use the previous loop? When we reach the end of loop, we know that the opposite of `temp != null` is true, so `temp` is null! This means the reading loop has gone too far. If we try to add a new node using `temp.next` at the end of the loop, we will have an error.

Since `temp` is set to `null`, calling `temp.next` would attempt to access the `next` of `null`, leading to the dreaded `NullPointerException`! In order to "stop earlier", we have to change our loop condition a bit.

This gives us the writing paradigm:

```
ListNode temp = p;
while (temp.next != null) {
    // Do something!
    temp = temp.next;
}
// Do something to the end of the list!
```

When we exit the while loop, we know that `temp.next == null` . This means that `temp` is referencing the very last node in the list – the only node that has its `next` field set to `null`!

On the other hand, inside the loop, we know that `temp.next != null` , so we have to be careful when we change `temp.next` inside of the loop – we could lose some nodes!

Main Points

- We can use while loops to help us advance through Linked Nodes, and help us stop where we want to.
- We want to think about the stopping condition – what is true about the end of the list?
- Our general loop pattern for **reading** will look like this:

```
ListNode temp = p;
while (temp != null) {
    // Do something!
    temp = temp.next;
}
```

- Our general loop pattern for **writing** will look like this:

```
ListNode temp = p;
while (temp.next != null) {
    // Do something!
    temp = temp.next;
}
// Do something to the end of the list!
```

- We have to be careful when we change `temp.next` inside of the loop – we could lose some nodes!

Approaching Linked List Problems [Background Reading]

In the previous slide, you learned two common approaches to linked list loops. ***However***, this does not mean that every linked node problem will look exactly like this! Linked Node problems are (notoriously) unintuitive and tricky.

Here are some questions you should ask yourself when approaching these problems. As an example, imagine that you are approaching the following problem.

Example Problem

Let `ListNode p` point to a sequence of `ListNodes` that are in nondecreasing sorted order. Here's a possible example:

```
p -> -1 -> 1 -> 5 -> 5 -> 13
```

Given an integer parameter, insert a new node into the list such that the list remains in nondecreasing order.

Linked Node Approach

1⃣ Reading or Writing?

Are we removing, moving, or adding nodes? If so, this is a writing problem. If we are only looking at nodes (never changing any nodes' `next` field), we have a reading problem. The answer to this question informs our while loop condition!

Since we are inserting a new node, this is a writing problem. So, we will use `temp.next` in our while loop condition.

2⃣ Where do we want to stop?

Do we want to do something at the end of the list? At the front? Somewhere in the middle? Or, does the "stopping location" change depending on the list and/or parameter? The answer to this question further informs our while loop condition, and where our code might go (before, in, or after the loop).

Since we might add our integer to the beginning, middle, or end of the list, there is no pre-determined place where we need to stop. This means that we will have to account for *every case* in our code!

3 What should happen when we stop?

Do we need to remove a node? Add a node? Switch a pair of nodes? What does this look like in a local "neighborhood" (like the problems we did in the first day of linked nodes in section).

When we reach a node whose value is greater than or equal to the value we want to insert, we would like to insert a node before it. If we reach the end of the list without having found a value greater, we should put it at the end of the list.

4 What cases should we consider?

Lists of linked nodes can take many different forms. Similarly to how we considered different cases to test using JUnit, when we write code for linked nodes, we should think about different potential states of the linked list. Also, just like with JUnit, these cases will be related to the specific problem we are working on!

Since we have different cases for inserting in the front, middle, and end, we should test out those three cases. We should also think about what we should do if our list is empty and we are trying to insert into it!

With these questions answered, let's solve this problem together!

Main Points

- When we approach Linked Node problems, we should ask ourselves these questions:
    - *Is this a reading or a writing problem?*
        - Reading is when we are only looking at nodes, and writing is when we are removing, moving, or adding nodes.
    - *Where do we want to stop?*
        - At the end, front, or middle? We will have to account for *every* case in our code.
    - *What should happen when we stop?*
        - Do we remove/add a node? Maybe switch nodes?
    - *What cases should we consider?*
        - We should think about potential different states of the linked list and certain edge cases e.g. if we are trying to insert into an empty list.

Insert Into Sorted Linked List [Practice]

Problem Spec:

Write a method `insertIntoSorted` that takes a `ListNode` parameter and an int parameter. The ListNode parameter points to `ListNode` s sorted in nondecreasing order.

Question 1

Which of the following is a correct method signature and method comment for `insertIntoSorted` ?

```
// pre: front is a linked list sorted in nondecreasing order
// post: a ListNode with data is in front, and front is still sorted
public static ListNode insertIntoSorted(ListNode front, int num) {}
```

```
// pre: front is a sorted list, data is non negative
// post: a ListNode with data is in front, and front is still sorted
public static ListNode insertIntoSorted(ListNode front, int num) {}
```

```
// pre: front is a linked list sorted in nondecreasing order
// post: a ListNode with data is in front, and front is still sorted
public static void insertIntoSorted(ListNode front, int num) {}
```

```
// pre: front is a sorted list, data is non negative
// post: a ListNode with data is in front, and front is still sorted
public static void insertIntoSorted(ListNode front, int num) {}
```

Question 2

A common way to approach Linked List problems is to solve each case separately, in order of simplest to most challenging. If `front` is an empty list (`front == null`) what lines of code should we execute?

```
front = data;
```

```
front.next = new ListNode(data);
```

```
front.next = data;
```

```
front = new ListNode(data);
```

Question 3

If front is not null, but the first element's `data` is equal to or greater than `num`, we should put a `ListNode` with `num` at the front of the list. What is the code to do so, without losing any nodes to the garbage collector? There are two correct options!

```
front = new ListNode(num);
```

```
front.next = new ListNode(num);
```

```
front.next = front;
front = new ListNode(num);
```

```
front = new ListNode(num, front);
```

```
ListNode temp = front;
front = new ListNode(num);
front.next = temp;
```

Question 4

Now, we will deal with the trickier cases: inserting into the middle and end. Let's assume that (by magic) we have a temp pointer pointing to the node *before* where we want to insert, for example

```
num : 10
```

```
front -> 2 -> 3 -> 3 -> 8 -> 11 -> 13
                        ^
                        |
temp - - - - - - - - - -
```

Which lines of code insert a ListNode with 10 into the right place? There are two right answers!

- [ ]
```
temp = temp.next;
temp = new ListNode(10, temp);
```

- [ ]
```
ListNode temp2 = temp.next;
temp.next = new ListNode(10);
temp.next.next = temp2;
```

- [ ]
```
temp.next = new ListNode(10, temp.next);
```

Question 5

Lastly, what should our code look like if we want to insert into the end of the list. In other words, if we reach this situation

```
num : 10


front -> 2 -> 3 -> 3 -> 8
                        ^
                        |
temp - - - - - - - - - -
```

What lines of code will add a `ListNode` with 10 to the list?

- [ ]
```
front.next = new ListNode(10);
```

- [ ]
```
temp.next = new ListNode(10);
```

- [ ]
```
temp = temp.next;
temp = new ListNode(10);
```

- [ ]
```
temp.next = new ListNode(10, temp.next.next);
```

- [ ]
```
temp.next = new ListNode(10, temp.next);
```

Question 6

Lastly, we need to find a way to get our `temp` pointer to that location. What is special about this location? Either

- `temp.next` is `null` (we are at the last element of the list)
- `temp.next`'s data is equal to or greater than `num`

Which lines of code brings `temp` to this location?

- ( )
```
ListNode temp = front;
while (temp.next != null || temp.next.data >= num) {
    temp = temp.next;
}
```

- ( )
```
while (temp.next != null && temp.next.data < num) {
    front = front.next;
}
```

- ( )
```
ListNode temp = front;
while (temp.next != null && temp.next.data < num) {
    temp = temp.next;
}
```

- ( )
```
ListNode temp = front;
while (temp.next.data < num && temp.next != null) {
    temp = temp.next;
}
```

Code for inserting into a sorted Linked List [Background Reading]

Now that we have gone through the problem-solving process, let's look at the complete solution for this problem

```
public static void insertIntoSorted(ListNode front, int num) {
    if (front == null) {
        front = new ListNode(num);
    } else if (front.data >= num) {
        front = new ListNode(num, front);
    } else {
        ListNode temp = front;
        while (temp.next != null && temp.next.data < num) {
            temp = temp.next;
        }
        temp.next = new ListNode(num, temp.next);
    }
}
```

Let's go through the steps in pseudo code:

1. If the list is empty, put our value into the list
2. If the list is not empty but the first value in the list is equal to or bigger than our value, put it in the front of the list
3. If the list is not empty and the first value is smaller than our value:
    1. Create a temp variable pointing to the start of the list
    2. Move the temp variable forward until we reach the last element or until we reach an element pointing at a bigger/equal element
    3. Insert our value after our temp variable, putting whatever is after our temp after our value (could be null)