San Francisco State University
CSC 220: Data Structures
Dawson Zhou (zhoud@sfsu.edu)

Assignment 4: Linked Lists

Overview

In this assignment, you'll use what you've learned about linked lists to solve four problems:

1. Inserting a new element into a singly linked list
2. Removing several elements from a singly linked list
3. Inserting a new element into a doubly linked list

Submission

Assignment 4 is due on **December 1, 2021** at **11:59 pm PT**.

Late submissions will be accepted with a penalty:

- December 2, 2021 12:00 am PT to December 2, 2021 11:59 pm PT will receive 75%
- December 3, 2021 12:00 am PT to December 3, 2021 11:59 pm PT will receive 50%

Submit your solution through iLearn. You will only need to submit **Program.java**.

Grading

100 pts total:

- 25 pts: **insertInSortedOrder** for singly linked lists
- 30 pts: **removeAllWithLength** for singly linked lists
- 25 pts: **insertInSortedOrder** for doubly linked lists
- 20 pts: Miscellaneous (coding style, no use of global or instance variables, etc.)

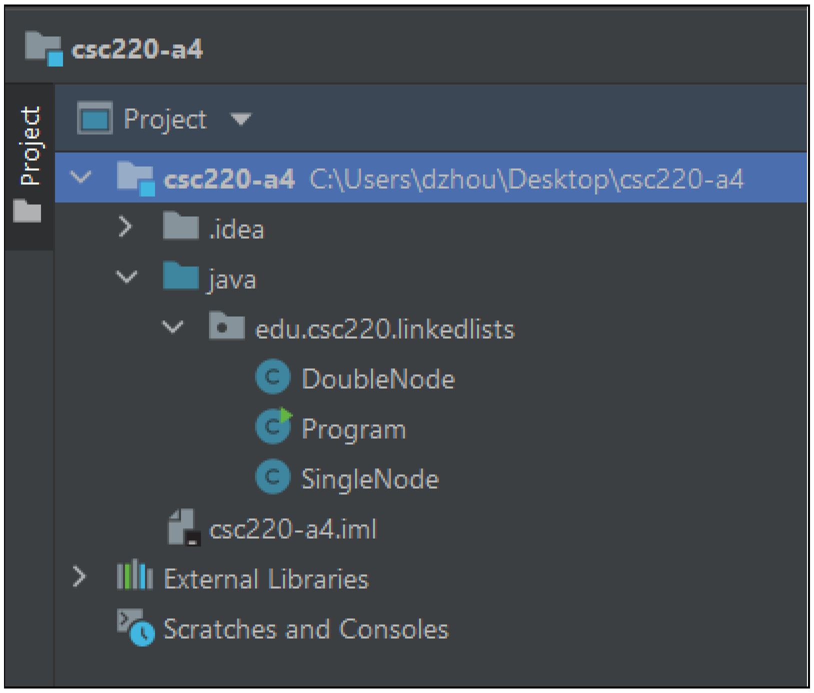
Assignment Setup

Download the folder named "Linked List Starter Code" from iLearn. It will be downloaded as a .zip file which can be extracted to whatever location you want on your computer. Make sure that the folder structure appears as follows:

```
> csc220-a4
    > java
```

Set up an IntelliJ project choosing “csc220-a4” as the root. For step-by-step instructions on setting up an IntelliJ project, see the “IntelliJ Setup Guide” handout on iLearn (at the top of the iLearn course page).

Once the project is set up, your project navigation panel should look something like this:



Please reach out ASAP if you have any problems setting the project up.

Detailed Requirements

Overview

The assignment starter project provides code in three files: **SingleNode.java**, **DoubleNode.java**, and **Program.java**. In all of our demos we worked through in lecture, we used the concept of a **Node** class to represent each node in a linked list, whether those nodes were for singly linked lists or doubly linked lists. In this assignment, because we need to work with both, we are naming them differently so they can both be used.

Our assignment will implement that store Strings instead of integers. **SingleNode** defines a node that can be used for a singly linked list:

```
public class SingleNode {
    private String element;
    private SingleNode next;

    // Constructor, getters, setters
}
```

DoubleNode defines a node that can be used for a doubly linked list:

```
public class DoubleNode {
    private String element;
    private DoubleNode next;
    private DoubleNode prev;

    // Constructor, getters, setters
}
```

Program defines several methods that you will need to implement to complete the assignment:

```
    public SingleNode insertInSortedOrder(SingleNode head, String newElement) {
        // ...
    }

    public SingleNode removeAllWithLength(SingleNode head, int length) {
        // ...
    }
```

```
public DoubleNode insertInSortedOrder(DoubleNode head, String newElement) {
    // ...
}
```

Note that there are two versions of **insertInSortedOrder**; one must be implemented both for singly linked lists (**SingleNode** version), and the other must be implemented for doubly linked lists (**DoubleNode** version).

insertInSortedOrder

Both versions of **insertInSortedOrder** take the head node of a linked list as well as a String representing the new element to insert into the list. The method assumes that the input linked list is already arranged in sorted alphabetical order, and it must find the correct location to insert the new element to maintain that order. Once the new element is inserted, the method should return the head of the list.

Note that this is a different insert method than what we worked through in our class demos. Instead of looking for a specific target to insert before/after, you're looking for the one correct place where this new node belongs based on comparisons with the existing Strings. After the new String is inserted, the resulting list should still be in sorted order.

The **compareTo** method allows you to compare two Strings to determine which one appears "first" in alphabetical order. Assuming that **firstString** and **secondString** are both String variables, it would be used as follows:

```
if (firstString.compareTo(secondString) < 0) {
    // firstString is "less than" secondString.
    // firstString would appear before secondString in sorted order.
}
```

...or, alternatively:

```
if (firstString.compareTo(secondString) > 0) {
    // firstString is "greater than" secondString.
    // secondString would appear before firstString in sorted order.
}
```

Of course, you will need to replace **firstString** and **secondString** in the code snippets above with the actual String variables you are comparing in your implementation.

Suppose you have a list of Strings in alphabetical order:

```
"Alice", "Bob", "Carlos"
```

If we want to insert the String `"Amelia"` into the list in sorted order, we can check each element in the list starting from the beginning. If we see a String that is “less than” the new element, we can’t really do anything, because we still don’t know what else follows. However, if we see a String that is “greater than” the new element, we know that our new element belongs in the spot ***right before***. In this example, when we see `"Bob"`, we should know to insert `"Amelia"` right before it.

I strongly recommend looking through our lecture material and demo code related to linked lists for ideas on how to approach list insertion.

removeAllWithLength

removeAllWithLength takes the head node of a singly linked list as well as a number named **length**. The method should remove every single String in the linked list where the String’s length is equal to the length parameter, and return the head node of the resulting linked list. If no Strings in the original list have the specified length, the list should be unchanged.

Important note: You *could* implement this by using code I’ve already provided in our demo examples to remove a single String, and just repeatedly do that until no elements are removed. However, doing so will only get you partial credit, as it’s an inefficient approach.

In order to receive full credit for each version, you must implement **removeAllWithLength** using only a single loop to go through the entire list. You can accomplish this by iterating through the list, removing a node when necessary *while still in the loop* (rather than breaking out of it), and allowing the loop to continue until the end of the list is reached.

Edge Cases

Remember to consider the various edge cases involved with all of our list operations. What happens if you’re inserting a new element at the beginning or the end of a list? What happens if you’re removing an element from the beginning or the end of a list? What if the list is empty to start with? How do these cases differ when dealing with singly linked lists or doubly linked lists?

Verification

I’ve provided code in the starter project to help with verifying your results. The **Program** class defines two versions of the **verify** method: one for singly linked lists, and one for doubly linked lists. You can call them from **main** to check if a linked list looks the way it’s expected:

```
    verify(ll1, "Alice", "Amelia", "Bob", "Carlos");
```

In this example, the linked list named `ll1` is being checked. If it doesn’t contain the Strings `"Alice", "Amelia", "Bob", "Carlos"` in that specific order, **verify** will throw an error with what the linked list actually contains. The doubly linked list version of **verify** will also check that the elements in reverse match the expected order.

Feel free to add your own test cases in **main**. Note that I have *not* provided all of the various edge cases in the starter code -- just because your submission passes the tests provided by default in **main**, doesn’t mean your implementation handles all cases correctly.