Lab Exercise 5
Doubly Linked Lists

1 Introduction

This time we will focus on doubly linked lists. A doubly linked list is a linked list in which each node has a pointer to both its successor and its predecessor. There is a link to the beginning of the list called head. The last node links to nothing i.e., there are no nodes after it. Also, there are no nodes before the first node. The pointer “before” of the first node and the pointer “after” of the last node are set to null. In this lab, you need to write some code to remove a particular node from such a doubly linked list.

2 Deleting From a List

There are 3 cases for removing a node:

- Removing a node from the beginning.
- Removing a node from the middle.
- Removing a node from the end.

We will examine each of these cases using the simple list below as an example.

Removing from the beginning

If the node to be removed is the first node of the list, the pointer to the beginning of the list must be adjusted and the before pointer of the new node at the beginning of the list must be set to null:

Removing from the middle

Removing a node from the middle requires that the preceding node skips over the node being removed, and the following node to skip over in the reverse direction. For example, removing the node with 2:
Removing from the end

Removing a node from the end requires that the preceding node becomes the new end of the list. This means that the pointer for the end of the list must be adjusted to the preceding node and after pointer of the preceding node needs to be null. For example, removing the node with 3:

3 Starting Up

Using Eclipse, import “Lab 5”:

```
/afs/cs.uwm.edu/users/classes/cs351/401/pantherid/git/lab5.git
```

You need to complete the implementation of `removeCurrent` method in `TaskList.java`. A correct implementation of `removeCurrent` method should allow the current node to be removed, and neighboring nodes adjusted as needed. This means traversal after a removal should be intact, i.e. if removing the middle node, you should be able to traverse from the first to the last element and back. After removal the current node pointer points to the beginning of the list.

4 Anonymous Classes

Anonymous classes allows for quick single implementation of a class or interface. Because the class has no name it cannot have a constructor. The advantage is if your class has a small body of implementation and only needs one instantiation, then you can define the class as it is needed. One use of this is the definition a single implementation of an adapter to an interface. The syntax for an anonymous class is as follows:

```
Comparator<Task> comparator = new Comparator<Task>() {
    // class body
    public int compare(Task o1, Task o2) {
        // implementation
    }
};
```

In specific example, you can create implementations of `Comparator` as you need, which allows us to compare to objects.

In `sortByImportance` method, create an anonymous class that implements `Comparator` that compares by the importance of the task. The larger the value for importance, the more important the task is, so we will sort them in descending order.
5 Bubble Sort

You may already be familiar with bubble sort, which is a simple but sometimes inefficient way to sort things. For bubble sort, you would start at the beginning of list and compare each element to the following neighbour. If they are out of order, we swap the current element with its neighbour. Take the following list of numbers:

\[ [10 \mid 12 \mid 7 \mid 5 \mid 9] \]

If you use bubble sort to sort by ascending order, you start with 10 and its neighbour 12. You can compare their values and see that in comparison to each other they are in order. So you proceed to check 12 and its neighbour 7. These two are not in order so you swap them giving the new list

\[ [10 \mid 7 \mid 12 \mid 5 \mid 9] \]

Continuing this you get the following

\[ [10 \mid 7 \mid 5 \mid 9 \mid 12] \]

Now as you can tell, the list isn’t quite sorted yet. So you start over again from the beginning and repeat the comparison with the following neighbour, swapping when necessary. You terminate once you iterate from beginning to end without swapping. We have provided the majority of the `bubbleSort` method, all you have to do is implement the comparison and utilize our swap method that takes in the current node to be swapped.

When you have completed the above steps, passed the supplied JUnit tests, and tested with the `TaskDriver.java` GUI, see your TA to receive credit for this lab.