CS 158 Lab 8 February 11, 2010
Lists Maintained In Order

1. Go into the project called Lists. Create a new ***package*** called **sllist**. Refactor/move all of the java classes you've built thus far into that package. Fix up any issues with missing package names and run all the tests.

2. Create another package called **orderedList**. In that package, create a new class called **OrderedList**. This class will be responsible for maintaining a list whose nodes are in order according to some ordering based on their data elements. As you know, Strings have an ordering that you can use with the **compareTo** method. Other classes can also have objects that can be compared to one another with that method. Your list should be able to work with any of them. The key is that any class that has objects that can be compared to one another must implement the **Comparable** interface. What this means for you is that any object that contains data for a node that is passed around via parameters will be declared as **Comparable** rather than just **Object**. This will come up when you work on the **add** method later. Right now, add protected instance variables to this class that represent the head node of a list and the size, and create a constructor that just instantiates the head node and sets the size of the list to zero. In order to use the **Node** class that we already have, you will need to import that class from our **sllist** package. Do not add any getters or setters yet.

3. Start a method called **add** that takes a **Comparable** argument. This method will be responsible for adding nodes to the list in such a way that they are always sorted. This method will be fairly complicated, the longest we've written. It will take several steps. So at this point, let's write the JUnit test for this method. Create the JUnit test class with a setup method and a stub for the **add** method. Set up an ordered list object for testing as usual. Think of your favorite sentence or saying (with at least 6 words in it), and in the test method, add each word (it will be easier if you use lower case exclusively for your words) to the list object using the **add** method (which, of course, hasn't been written yet). Create a **Node** object called **current** and initialize it to the list node that the head node is pointing to (*i.e.*, the first node of the list). Assert that the data in **current** is the alphabetically first word in your sentence. Increment **current** and assert that it now contains the alphabetically second word in your sentence. Repeat for each word you added. That's the test. You can run it if you wish. It should fail gloriously.

4. Go back to the **OrderedList** class and start working on the **add** method. First create a new node object, and put into its data component the data that comes in as a parameter. You can go ahead and increment the size instance variable now, so you don't forget to do it later. Treat the situation where you have an empty list as a special case. Take care of that case now (if the list is empty, do what is necessary to add that first node). Go back to the test class, comment out all the assertions except the first one. Run the test and it should work.

5. Back in the **add** method, the rest of this method will be what happens if the list is not empty. Our next task will be to find the insertion point for the new node. To do this, we will use two temporary node objects: one for the predecessor and one for the current node. Initialize the predecessor to the head node itself, and initialize the current node to what the head node points to (the first actual node of the list, which we know exists at this point). Also set up a boolean variable called **found** and initialize it to **false**. Using the **compareTo** method, see if the new node's data is smaller than or equal to the data in the first node of the list. If it is, you have found the insertion point, and you can set **found** to **true**. Either way, the next thing that happens is a loop that continues as long as the insertion point has not been found and the current node is not the last node of the list. Inside the loop, increment the predecessor and current nodes, and check to see if the new node's data is smaller than the data in the current node. When the loop terminates, it's either because the insertion point has been found, or you've come to the end of the list with all elements having smaller data. So first take care of the case where the insertion point was found. Insert the new node after the predecessor and before the current node by setting the appropriate "next" fields. Then take care of the case where the new node is to be inserted at the end of the list. That's it. Go back to the test class, put all the assertions back in, and verify that the tests pass.

6. Copy the **getNode** method that you've already written into this class, and verify that it works. Add a **getSize** method that does the appropriate thing.

7. In the Lists project, create a new package called **client**. Create a new class in this package called **UseOrderedList** that contains a main method. In that main method, create an ordered list object, and add to it a bunch of words (you can copy the same code you used to test the **add** method earlier). Write a loop using the **getNode** method that prints each data element of the list in the order that the list has them. Run the main method and see that all the words are printed as expected.

8. Write a **remove** method that takes a **Comparable** argument, returns a boolean, and removes an element from the list if found. Follow the pattern that we used in this lab. Create a test for this method, and then implement the method itself. This will be a little easier than the **add** method, because you're not creating any new nodes, you're only testing for equality, not comparing in order, and deleting a node involves fewer assignments.

9. Copy **OrderedList.java** to the shared drive.