CS 158 Lab 6 February 2, 2010
Introduction to Linked Lists

1. Create a new project called Lists. Create a new class called **Node**. Add two protected instance variables, the first an **Object** called **data** and the second a **Node** called **next**. Add a method called **setData** that takes an **Object** parameter and sets the **data** instance variable to it. Add a method called **getData** that returns the **data** variable. Add a method called **setNext** that takes a **Node** argument and changes **next** to that value. Add a method called **getNext** that returns **next**.

2. Create a new class called **SLList** (for "singly-linked list). Add two protected instance variables, one a **Node** called **head** and the other an integer called **size**. Create a constructor that instantiates the head node and sets its data to the string "Head node of singly linked list". Add a **getSize** method that does the obvious.

3. Create a method in **SLList** called **add** that takes a **Node** argument and returns nothing. This method must place the node that is provided (as an argument) into the list at the beginning and update the **size** variable. Since we are using a head node, this will be fairly simple. All you have to do is make the new node point to what the head node was pointing to, then make the head node's **next** field point to the new node (these two things must be done in that order -- note what will happen otherwise!), and update the size of the list. Draw a picture of this situation yourself. If you cannot, ask the instructor or the TA. This is an important skill. Create a JUnit test class for **SLList** that contains a test method for **add**. The test will be a bit simplistic at this point, but we will improve it later. For now, assert that the list has no elements to begin with, then add a node and assert that it has one element, and repeat a couple of more times.

4. We need a method that will return a node from the list, and there are many ways to set this up. For this method, we will expect the client to first find out how many elements are in the list (with the **getSize** method), and then ask for one of the elements by number. Please note: the head node will be considered element 0, and will not be returned. So the elements of the list that the client can have returned are numbered starting at 1 and ending with the size of the list. Create a method called **getNode** that takes an integer argument **n** and returns a **Node**. The first thing this method must do is verify that the argument **n** is valid, and if not, throw an **IllegalArgumentException**. If the argument is valid, the method will initialize a local **Node** variable **temp** to **head**, and then loop **n** times. Inside the loop you will "increment" **temp** in the following way: **temp = temp.getNext( );** When the loop terminates, **temp** will be pointing to the nth element of the list, and you can return it. Draw a picture of how this works by building a list one node at a time that has about 5 nodes in it, and then trace through what happens if someone does a **getNode(3)**.

5. Write a test method for **getNode**. Begin by asserting that the size of the list is 0. Then add about 10 nodes with data (Strings) that you can later test for. Then assert that the size is what you expect (this will duplicate your test for **add**, but the next part improves it). Now, get the data from the first node in the list. Remember that it was in the last node that was added. Assert that it is equal to what you put in last. Then get the data from the last node and assert that it is what it should be.

6. Start a new method in **SLList** called **remove** that takes an integer argument **n** and returns nothing. The rest of this lab will make this method work correctly. This method will remove node **n** in the list, and, like **getNode**, must validate the input. Begin by writing a test method. Test that you cannot remove a node from an empty list. Use the pattern involving try/catch where you expect an IllegalArgumentException, and you cause a fail if you don't get it. If you run the test now, it should fail, since you have not yet validated the argument in the **remove** method. Write the code in the **remove** method that validates the argument. Make sure the test now passes. Then add a node and remove it, and assert that the list is empty. This test should now fail, because you have not yet written the statements that actually remove a node.

7. After validating the argument in the **remove** method, the next step is to locate the ***predecessor*** (the node before) of the node to be deleted. This is fairly easy to do, since it's almost the same as the loop used in **getNode**; it just stops one step sooner. Once you have the predecessor to the node you want to remove, just set its next link (its successor, the node after it in the list) to the successor of its current successor. Draw a picture of what is going on. Finally decrease the **size** variable. The tests should now all pass.

8. In your test for the **remove** method, add 10 nodes with data that you control. Delete the first node and assert that the first node now contains what it is supposed to. Delete the last node and assert that the new last node contains what it is supposed to. Delete a node in the middle and make the appropriate assertion. Also, each time assert that the number of nodes in the list is the right number.

Copy all the Java source files to the shared drive by lab time on Thursday.