Lab 9 - Doubly Linked List - Lab Report

By: Daniel Williamson, Stephen White, Kelli Ruddy

Due Date: 4/26/16

Lab Report

**Problem Statement:**

For this lab our goal was to create a list that is doubly linked. Meaning it can move both backwards and forwards between nodes. We will do this by creating an abstract class and by defining a fully featured Java list implementation. Lastly we will be practicing unit testing by writing our linked list code to the requirements defined.

Requirements:

• You must be able to retrieve an iterator object with listIterator(int)

• You must be able to add to an empty list using an iterator

• You must be able to read the ﬁrst element of a list using an iterator

• You must be able to move forward and backwards through your list using an iterator

• You must be able to add arbitrary elements using an iterator

• You must be able to read arbitrary elements using an iterator

• You must be able to remove elements using an iterator

• You must be able to replace a value in the list using an iterator

• You must be able to determine if you are at the beginning or end of a list using an iterator

• You must be able to determine your current index using an iterator

• You must be able to retrieve iterator objects at a non-zero index from listIterator(int)

• You must be able to correctly retrieve the size of the list after adding and removing elements correctly

• You must be able to use addAll(Collection) to add every item in an existing collection.

• You must be able to read from the list by index

• You must be able to remove from the list by index

• You must be able to add to the list by index

• You must be able to replace an element by index

• Trying to iterate past the end of the list, or previous to the beginning should throw a NoSuchElementException

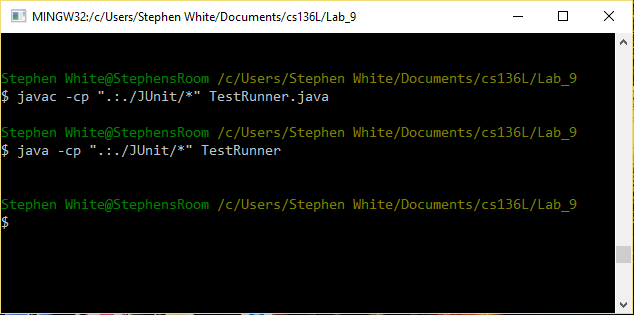
**Planning:**

We began this lab by discussing different ideas we each had on how to tackle this lab. We also made sure to look through the powerpoint slides provided to use in lecture. These outlined how we would be able to add and remove items in out list among other things. These slides proved to be extremely helpful. We also made sure to utilize the lab aides in helping us understand the concept of the lab and work through the many problems we encountered along the way. Throughout this process we made diagrams on paper to help us see how the mapping worked and how to rewire certain things. By drawing out what we were actually implementing helped us immensely in understanding what our code was actually doing. Also, being able to go back and look at these diagrams the next week in lab helped us to remember how we got to where we did the week prior. Since the topic that this lab implemented was relatively new, we were a bit nervous about it but once we broke down the problem and really thought step by step, we were able to figure it out. By utilizing the lab aides, the lecture powerpoint slides and our diagrams, we were able to complete this lab.

**Implementing and Testing:**

Our biggest friend in this lab was the use of pictures to piece together what exactly we needed to do next to get the code to work. By first looking at the slides that Ryan provided us, we set up all of the instance variables that we would need and looked at the implementations of each method. Essentially from that point forward, we wrote out expected outcomes of unit tests down on paper, wrote the test in code form and let the compiler errors lead us to our final code as well as pointing out any errors in the process. The unit testing was essential as it assured us that no matter what type of object was passed into the doubly linked list, the iterator would work 100% efficiently. Once we learned how to add elements to an empty list, we worked forward from there, implementing our add method, next(), and previous() to be sure that elements could be read, and then we tackled remove(). Because we were certain we could move throughout the list in both directions correctly, implementing remove() was very similar to adding, we just had to keep track of which direction the iterator had moved previously. The last portion of the basic code was being able to replace an element in the list. It was not that difficult, as we already understood how to reconnect nodes. Once the basic code had been completed, all we had left to do was be able to keep track of what index we were at, and be able to add(), remove(), and set() at a specified index. We did this through our doubly linked list instead of directly telling the iterator to go to indices. Below are screenshots of the test code, as well as proof that every test passed and the program was able to compile.





**Reflection:**

Overall, we found this lab to be pretty tricky but once we broke down the problem, we were able to see it just took some time and thought to figure out. This concept is definitely not the easiest to understand when looking at the problem as a whole but we made sure to look at each piece and break down how we would could accomplish each task. The most difficult part of this lab was probably just understanding what we were trying to do. Like stated previously, this concept is somewhat difficult to understand fully so we all agree that just starting was the most difficult part. Once we got going, we definitely still had issues but they were easier to tackle because we understood what we were trying to do. To refactor our code, we might try and change some variable names to make them more clear because we did have a few that were similar. This might have made it easier to read.