Vishal Yathish

9 November 2023

Project #3 Report

**PROJECT IMPLEMENTATION:**

For this project, we were asked to implement 6 methods into the List class provided in the skeleton.

The first method was size(), which would return the total number of data values found in the linked list. This was a relatively simple problem of traversing the list and incrementing a counter/output variable by one for each non-null node in the list. The only special case to this problem would be if the list were empty, for which we could use the useful IsEmpty() method already written, and the count was 0. For traversing the list, we use a while loop, with the node going on to the next node and incrementing the counter so long as the initial node value was not null (or had reached the end of the list).

The second method was position(), which would return the location of the first occurrence of an input string found in the List. If the data occurred multiple times on the List, we were to return the position of the first entry, and if the data was not present, we were to return -1. This was very similar to the first method in that it required only one traversal of the list using a while loop. I kept separate variables for the final output (with a default value of -1) and the counter variable, which would give the current position in the list. And if the value of the current node was ever equal to the input data, then the loop would break (and we would stop traversing the list), thus ensuring that we got the first entry of data in case of repeats, and the output variable would be set to the position of the entry.

The third method was before(), which would see if the first input string was present in the list before the second input string. If it was, then the method would return true, if not, then it would return false. In implementing this method, I simply used the position () method from earlier in the project. If both elements are in the list (and I did check for this case), then it would compare the position of the first element and the position of the second element. If the first position was less than (came before) the second one, then the method would return true. If not, then the method would return false.

The fourth method was get(), which sets the value of the parameter data to be the element located at a given (input) position of the List (if that position exists) and return true if the position exists. Otherwise, it would return false. This method also involved traversing the list with a while loop, using a counter variable to keep track of the current position. If the counter ever reached the given input element, the output would be set to true, the data would be set to that node’s element, and the loop would break.

The fifth method was min(), which would return the smallest item on the list. If the list is empty, the empty string should be returned. Again, this involved traversing the list with a while loop. The output (string) variable would be set to a default empty string if the list is empty. If not, then it would set to the first element of the list. As it traverses, it compares each element to the output variable; if it is smaller, then the output variable is adjusted to reflect that new smallest value. If not, the output variable is unchanged.

The final method was removeAllBiggerThan(), which would alter the list by removing all the items bigger than a given input data that were originally in it. This method was the most challenging, and I did require some assistance with it. My first approach was to again traverse the list with a while loop, and if the current element is larger than the input string, it would call the provided builtIn() method and remove it from the list. Then we would move on to the next node. However, this did not seem to work, as when the method removed one element from the list, the loop would stop, and it would not continue iterating through the rest of the list. I asked Howard about this during office hours, and he pointed out after calling the deleteItem() method, it did not make any sense to walk the node, since the pointer was deleted. So, what I to do was save the next node in a temporary variable before deleting the current node. Then assign the temp node to the current node. This implementation seemed to work.

**PROJECT TESTING/DEBUGGING:**

I tested my code on all of the test cases Howard gave us, and my methods all seemed to work. I also added a few of my own, to make sure that all methods outputted what I wanted them to.

Particularly, I tested the get() method on the following cases …

List l1;

string testString = "testing";

assert (l1.get(0, testString) == **false**);

assert (l1.get(1, testString) == **false**);

l1.addToFront("testing");

assert (l1.get(0, testString) == **true**);

assert (l1.get(1, testString) == **false**);

These seemed to have issues. The get method was accurately assigning new values to the node elements. However, it was having challenges with the final Boolean output. This mistake was relatively easy to correct however; in my implementation, I missed a command to assign a new Boolean value to the output variable, and I was able to add it in.