**CS2302 Data Structures**

**Lab Report No. 3**

Author: Bryan Ramos

Due: October 4th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

The purpose of this lab was to provide additional practice using linked lists which we have seen during class lectures and the past quizzes, the only difference is that in this case the element of this list will always be sorted in ascending over at all times and contain a series of functions that allow different operations to be performed on the list. In addition, there was a chance for additional practice with algorithm time complexity where we had to determine the time complexity for the functions in the *Sorted List* class and compare those runtimes to the regular *List* class we saw during the course.

**Proposed Solution Design & Implementation**

Before tackling the design of the assigned functions, first I imported common code from the List class (i.e. constructor) to create a linked list data structure. The instructions required that the list elements should always be sorted in ascending order, so I brainstormed several methods to make sure that condition was met. My method which I called *Append* has several checks to determine where the element belongs in conjunction with the existing elements so that everything stays in ascending order and inserts the integer in a node where it belongs.

I created a List and populated it with random integer values that I could manipulate with each of my operations and use the output in the console to debug and test. After that, I created a menu that shows each of the assigned functions as a menu option and implemented a switch using if-else-if conditions to execute each of the assigned operations separately in the console.

With that out of the way, I began working the assigned functions in the order they were assigned:

1. **Print:** This function needed to print the contents of the list. The print function accepts the list as a parameter and I used a loop to traverse the loop starting from the head all the way to the last element (tail), printing the data at each node.
2. **Insert:** Elements need to be inserted in ascending order. The insert function requires two parameters, the list and the integer to be added to the list. Because I had already designed the method Append when creating my lists for debugging and testing, I went ahead and called that function within this one.
3. **Delete:** The instructions state that this function should remove integer *i* from the list and if the integer *i* is not in the list it should do nothing. The way I approached writing this function was by using a try and except block. First, I have two variables, *previous* which is initially null and used to keep track of the previous element found before current. Since there is no previous element before the head it is initially null, and *t* which is the current node and initially is the head of the linked list. Next, my try-except block which traverses the list and if the element is not found returns None, the function stops there, thereby fulfilling the requirements. If it is found that element is removed, that is, that node’s reference is removed from the list.

For example, say that we are traversing the list and the current node the traversal is at is somewhere in the middle of the list and it happens to be the node storing the data we want to remove, then we remove the node by pointing the previous element’s (stored in *previous*) reference to the element after the current, if there is an element after current.

1. Merge: T