**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #3**

Due: October 4th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab we were tasked with creating a linked list that is always sorted. We then were tasked with creating several methods to manipulate the list such as adding an element to the list or checking for any duplicate numbers in the list.

**Proposed Solution Design and Implementation**

**Creating the Sorted List:**

I started the program by creating a Node object that contains the data of the node as well as the position of the next node. After that I created a new class called sorted list that would hold all the methods as well as an initiate method that starts the list.

1. **Print(self)**

For this method I would set a temporary variable “T” to be the head of the linked list. Then was long as T was not None the method would use a while loop to print every element of the list until T equaled None.

1. **Insert(self, i)**

For this method I first started by setting variable “H” as the head of the linked list, then created a node with the integer i. Next the program would check if the list was empty. If it was the node created would be set to the head and tail of the list. If it was not empty the program would check if the new node element was smaller then the head and if so, would place the new node as the head of the list. Otherwise it would just use a while loop to find where it is less then the next value of the list and insert is making sure to set the previous nodes next to itself and its node to the original next.

1. **Delete(self, i)**

For this method I start out by checking if the list is empty because if it is there is no need to do anything. After checking if the list is not empty the method then use a while loop to move through the list to the ith position and remove that node. If there is no ith position the program just returns.

1. **Merge(self, M)**

For this method I First checked if M.head was None in case we were trying to merge with an empty list. After that I checked if self.head was None and if so would just set the head and tail of M to be the head and tail of the list. If neither of those cases run then the program uses a while loop to sort through M and using the insert method it would add each element in its appropriate spot in the list.

1. **IndexOf(self, i)**

For this method I checked if the value of I was 0 and if so, would return the data of the head. If not I would use a while loop to go through the list to find and return the elemnt at index i. I would return -1 if there is no element i.

1. **Clear(self)**

For this method I reset the head.next of the list to none. Then I set the head and tail of the list to none ensuring that the list was completely reset

1. **Min(self)**

For this method since the list was already sorted, I just returned the value of the head.

1. **Max(self)**

For this method since the list was sorted, I just returned the value of the tail.

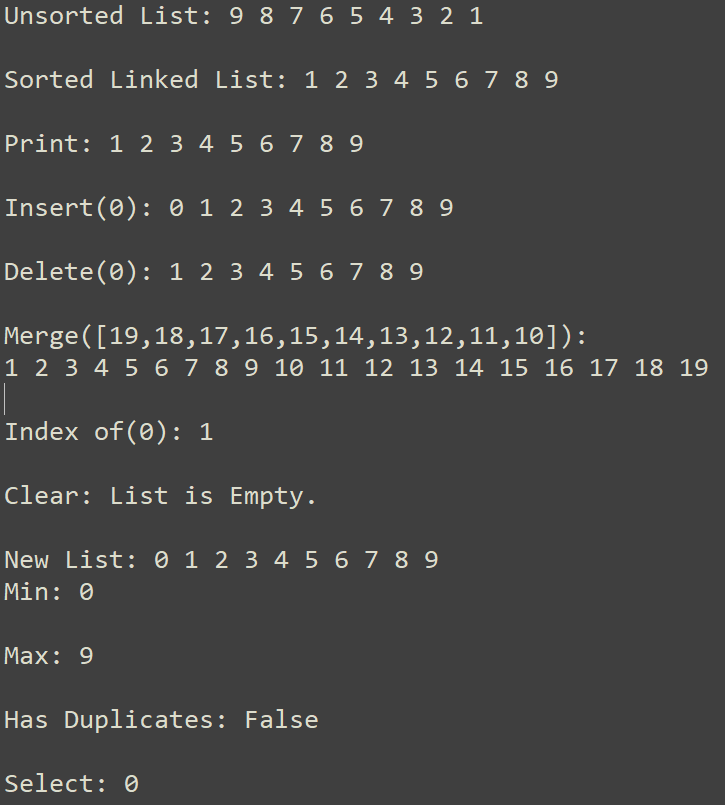
1. **HasDuplicates(self)**

For this method I first checked if the head of the list was empty and if so, returned false as there are technically no duplicates in the list. Then I created a list with the data of the head as well as a node value with the index of head.next. After that I would use a while loop and for loop to iterate through the list. While going through the list it would check if the temp nodes data is within the list of values already stored and if there is a matching value it would return true. Otherwise it would end after it finished iterating through the list.

1. **Select(self, k)**

For this method I first created two base cases, to see if the value of k was negative or if it was 0 and if so, would return the value of the head node. Otherwise It use a while loop it go through the list to return the kth smallest element as well as returning math.inf if k was larger then the length of the list.

**Experimental Results**



L1.AppendList([9,8,7,6,5,4,3,2,1])

L1.Print()

L1.Insert(0)

L1.Delete(0)

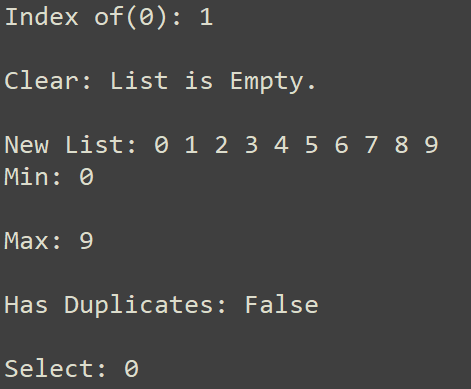
L2 = SortedList()

L3 = SortedList()

L2.AppendList([9,8,7,6,5,4,3,2,1])

L3.AppendList([19,18,17,16,15,14,13,12,11,10])

L2.Merge(L3)

L1.IndexOf(0)

L1.Clear()

L1.Append([9,8,7,6,5,4,3,2,1,0])

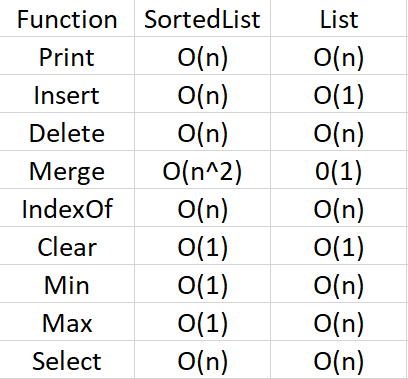
L1.Min()

L1.Max()

L1.HasDuplicates()

L1.Select(0)

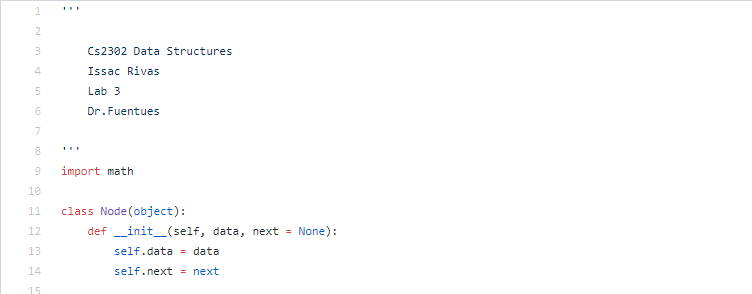
**Data**

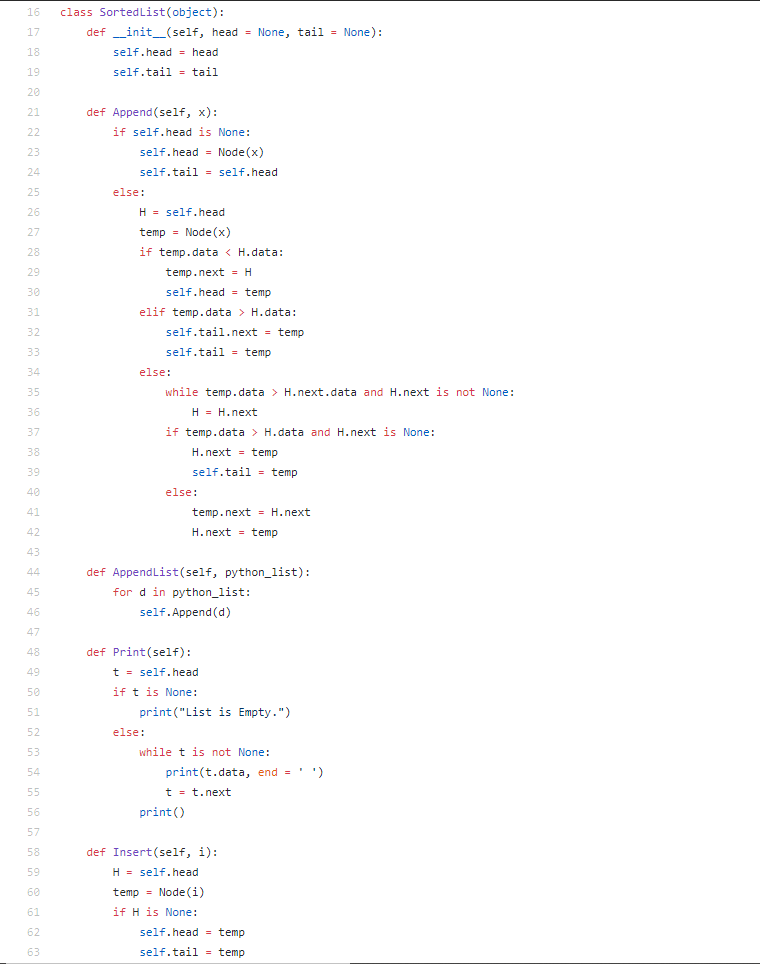


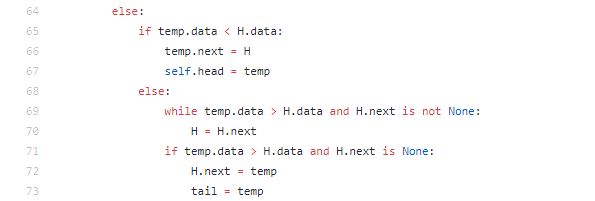
The data helps show how some tasks end up taking longer then a normal list suck as in the case of merge sort(O(n2)) were each item needs to be place in ascending order. While other tasks like Min and Max can be solved with a run time of O(1) because the list is already sorted.

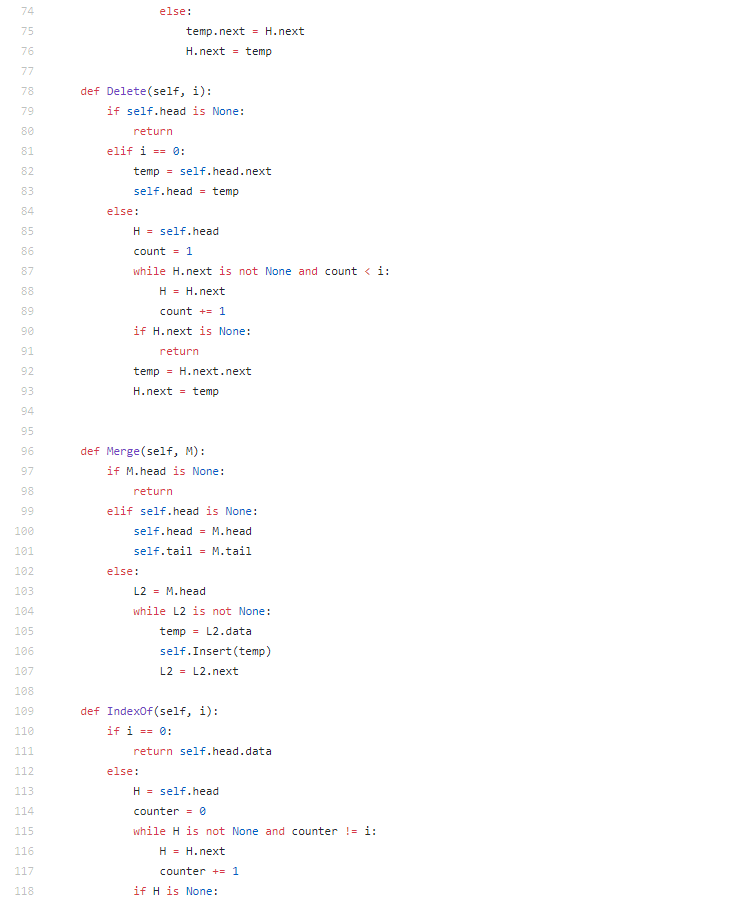
**Conclusion**

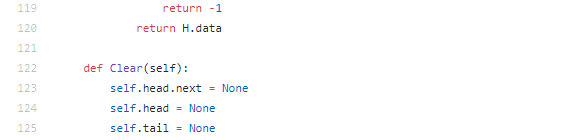
In conclusion through completing this lab I was able to see how a sorted linked list was able to have better run times in certain tasks then an unsorted linked list as well as how I can create methods that manipulate these linked lists such as if I wanted to create one for descending order.

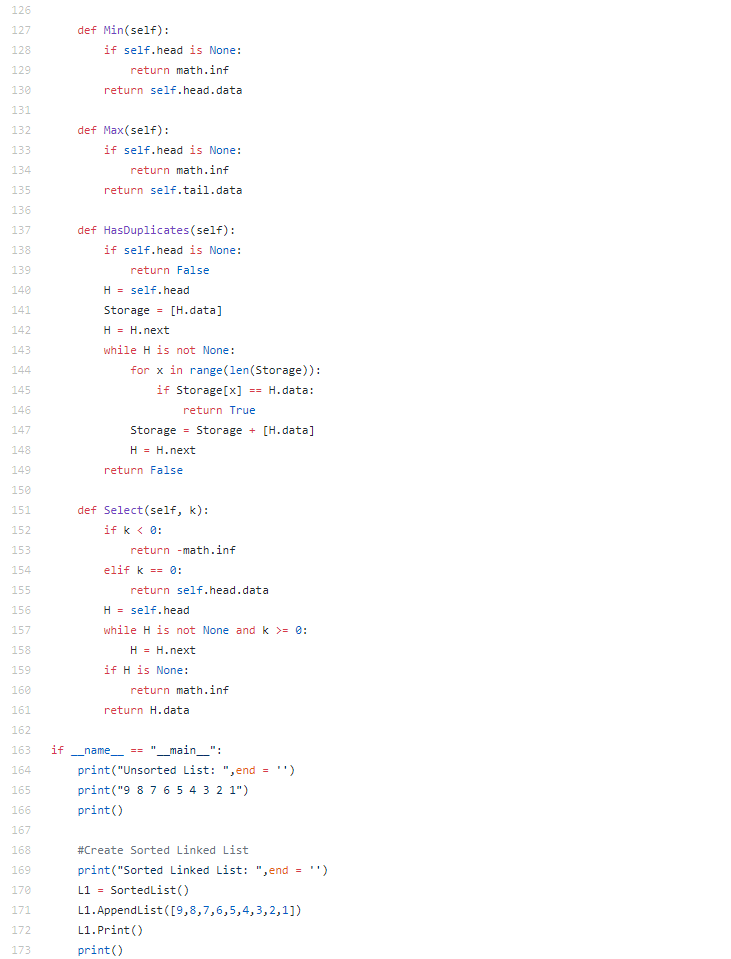
**Appendix**

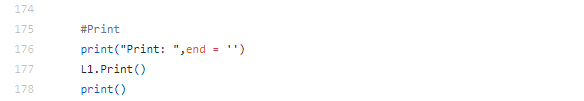


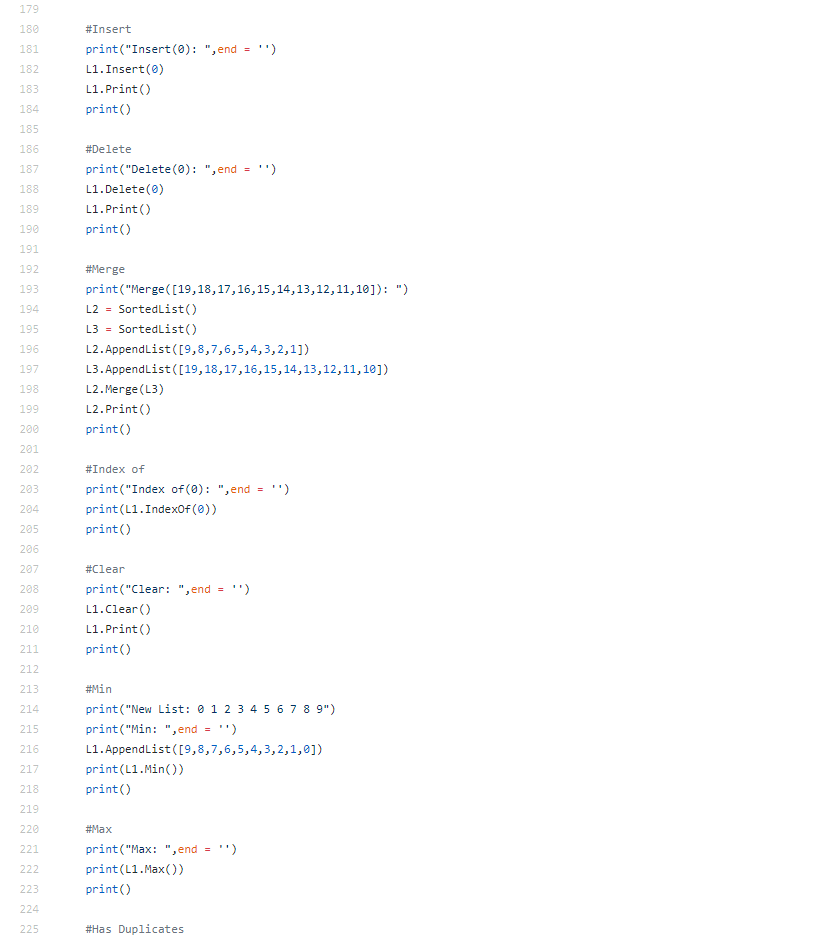


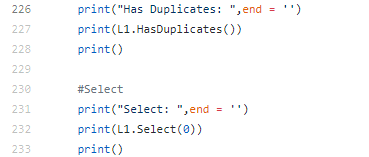














I Issac Rivas, certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, preformed the experiments, and wrote the report. I also certify that I did not share my code or report provided inappropriate assistance to any student in the class.