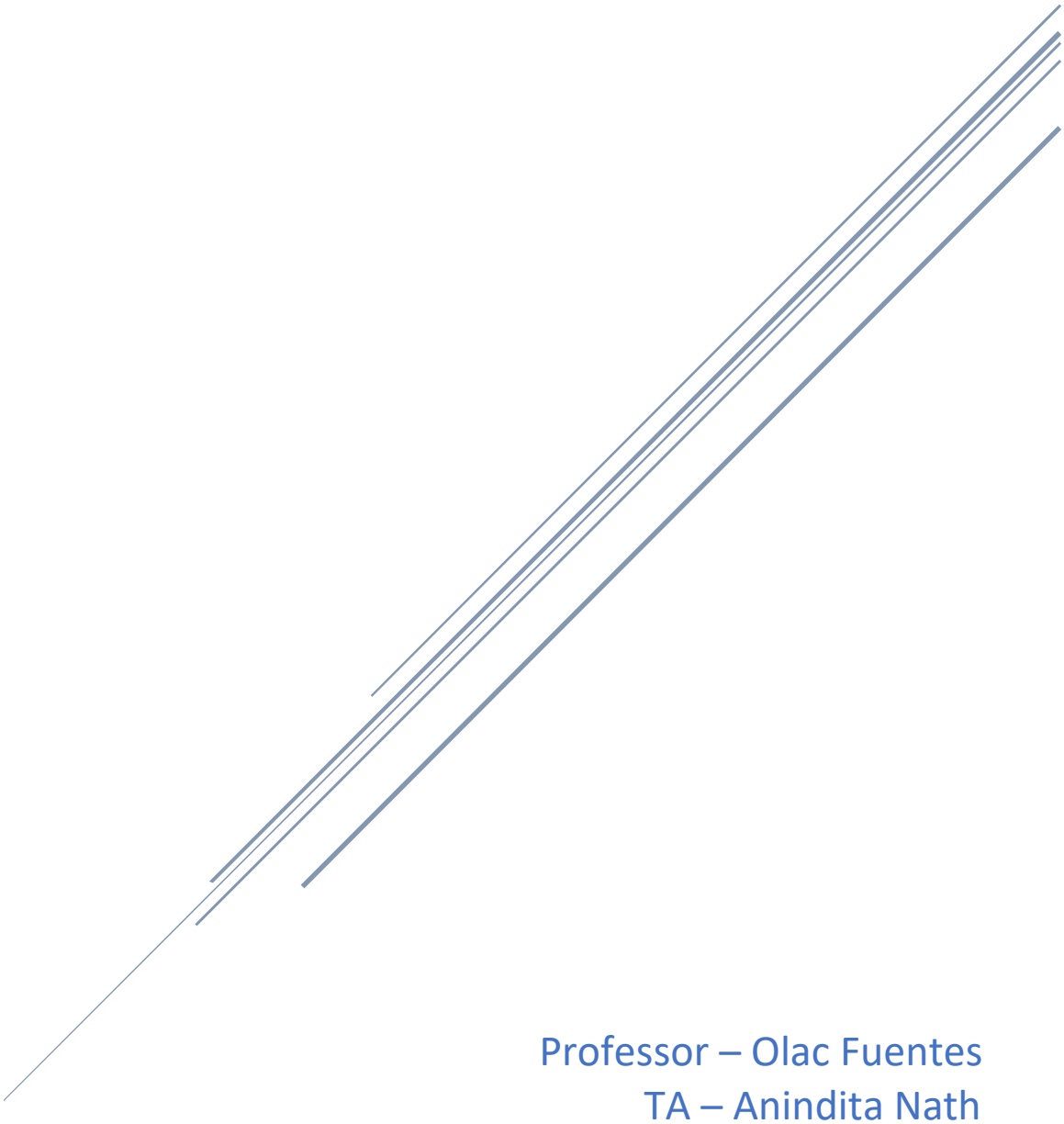
LAB REPORT #3

Sergio Ortiz
October 4, 2019



Professor – Olac Fuentes
TA – Anindita Nath
CS 2302 Data Structures – Fall 2019

Introduction

For this lab, we were asked to create the Sorted List class, which would be like a regular List class but the elements inside it would always be sorted. From here, we were asked to implement several functions inside the class such as Print(), Insert(), and Delete(). For this lab, it was essential to know how a regular Linked List worked such as how pointers and nodes work as well as a good understanding in asymptotic complexities.

Proposed Solution Design and Implementation

Print(self):

The idea for this method was quite simple as we have seen it several times. I decided to make a reference to the head node to traverse the list. From here, a while loop would end when the current node was empty, printing out the contents of the current node and advancing one place to the right.

Insert(self, i):

This method alone is what separates the regular Linked List from the Sorted List. I realized that when the user called this method, it would have to traverse the list and insert the item in its correct position rather than simply inserting it at the end. To optimize the function, I decided to cover some special cases. The first one would be when the list is empty. If so, then my function would create a new node with the given value and assign it the head and tail pointers. The next special case was if the given number was smaller than the value of the head node. In this case, the function would make the new node the head. Finally, the last special case was if the number was bigger than the value of the tail node. Here, we would make the new node the tail node. After this, I had to implement every other case, a situation where the new node had to be inserted somewhere in the middle of the List. From here, I made a reference to the head node and advanced it until it was smaller than the next element. Then, the new node would go after this node. After inserting it, I rerouted the assignment of the main head to the new reference I created.

Delete(self, i):

Deleting an item from a linked list is quite easy. It merely consists of moving the ‘next reference of the previous node to the one after the node we want to delete. I decided to traverse the list until I found the node with the given number and then delete it from the list. The only hurdle came when the numbers repeated, because my program would advance to the next node every time and check if its next was the given number. Therefore, if we wanted to delete ‘3’ and we had two ‘3’s, my program would stop at the node before the first 3, delete the 3, and advance to the next 3. Once at the next 3, it would check if its next was ‘3’, leaving that 3 intact. To solve this, I decided not to advance the pointer if we came across a node containing the value we wanted to delete.

Merge(self, M):

The idea for this function was that it would create a reference to the head node of list M, and traverse it until it was empty, calling the Insert method with the data of each node.

IndexOf(self, i):

For this method, I created a counter and a reference to the head node. Then, I would advance the pointer to the next node, incrementing the counter, until the program hit a node with the value that was given. At this point, the value of the counter would be returned. If we came to the end of the list without hitting the value, then the function would return -1.

Clear(self):

For this function, I simply made the head node None. This would not only delete the head node, but also the entire list behind it.

Min(self):

Another trivial method, I decided to simply return the value of the head node.

Max(self):

Same as the Min() method, but returning the value of the tail node instead of the head node.

HasDuplicates(self):

Since the list is in order, I traversed the list and returned True if the program ever came upon an instance where the data of the current node was equal to the data of the next node. Then, if we came to the end of the list, I would return False.

Select(self, k):

First, I decided to check if the list was empty, returning math.inf if so. Then, I created a reference to the head node and advanced it, decrementing k each time. Once k was one, the loop would exit and return the value of the data of the current node unless the loop exited and k was still not zero. In this case, we had reached the end of the list and k was bigger than the list, so the program would again return math.inf.

Experimental Results

It is worth noting that to test all these methods, I created a new class in which I imported the Sorted List and all its functions.

```
1  from SortedList import List
2
3  if __name__ == "__main__":
4      #all the test cases will go here
5
```

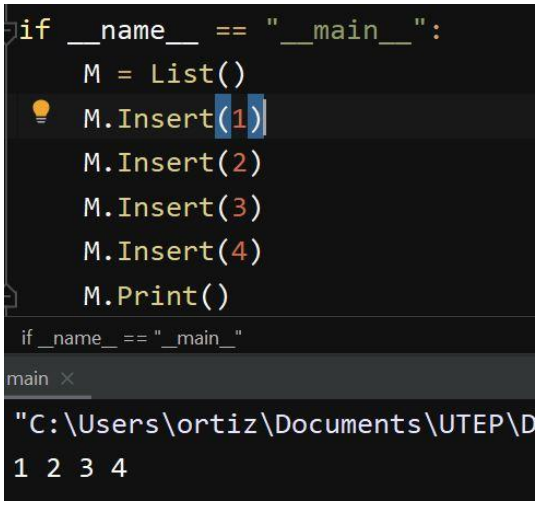
Print(self):

I really could not test this method until I had the Insert() method, but once I did I merely added elements to the list and printed them to see if the method actually printed the contents.

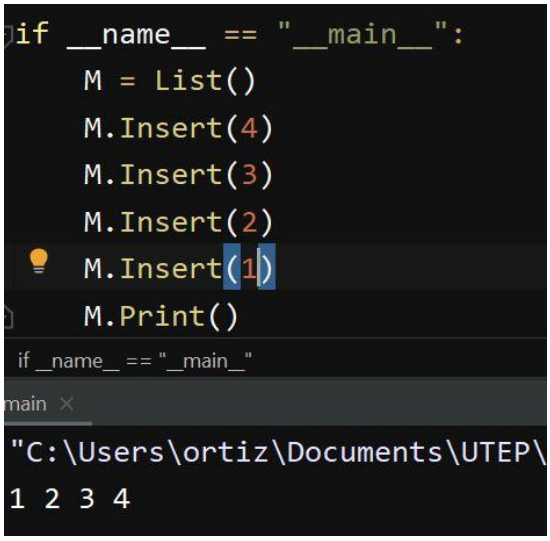
```
1  from SortedList import List
2
3  if __name__ == "__main__":
4      M = List()
5      M.Insert(5)
6      M.Insert(3)
7      M.Insert(6)
8      M.Print()

if __name__ == "__main__"
Run:  main
"C:\Users\ortiz\Documents\UTEP
3 5 6
```

Insert(self, i):

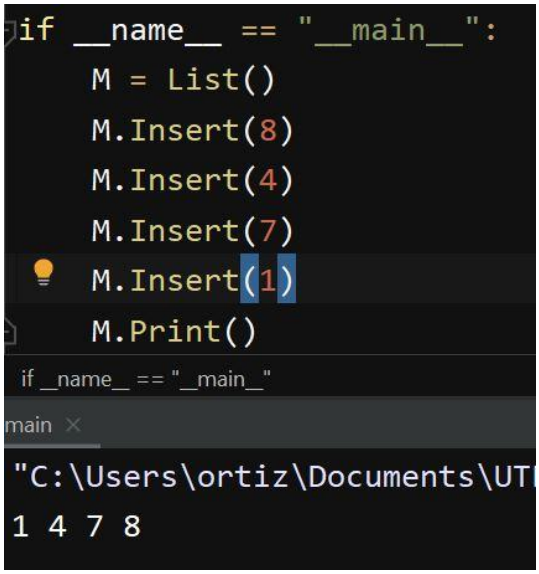
To test the insert method, I decided to do three test runs. The first one would consist of inserting a set of numbers already in order and see if it would print them out correctly. Here we can see the results.



Next, I decided to insert a set of numbers in reverse order to see if my insert function could reroute them all back in order.



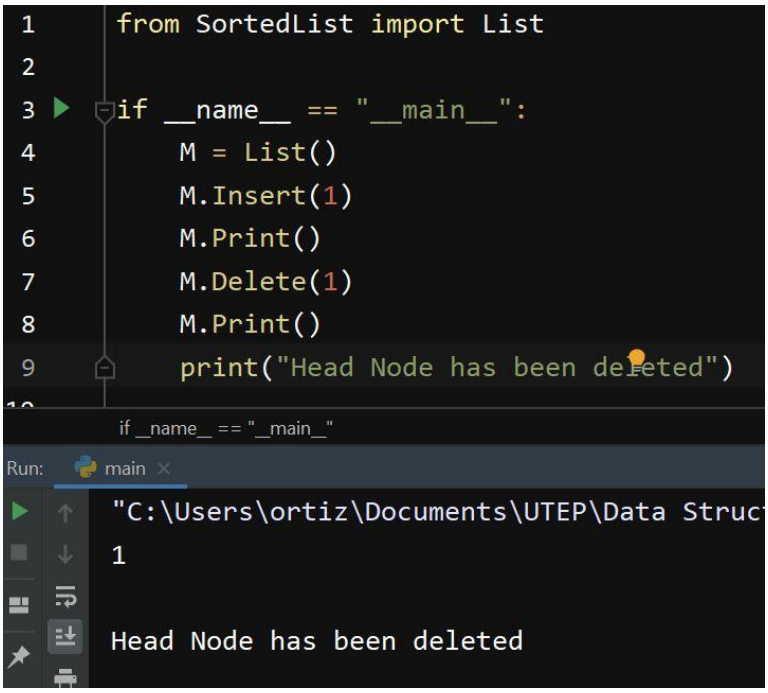
Finally, I tried with a set of randomly ordered numbers to see if the Insert method could put them all in order.



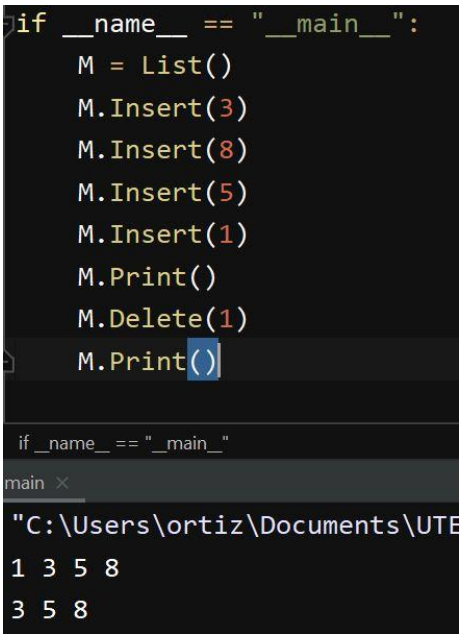
As we can see, the main meat of the Sorted List class works, maintaining ascending order in its elements at all time.

Delete(self, i):

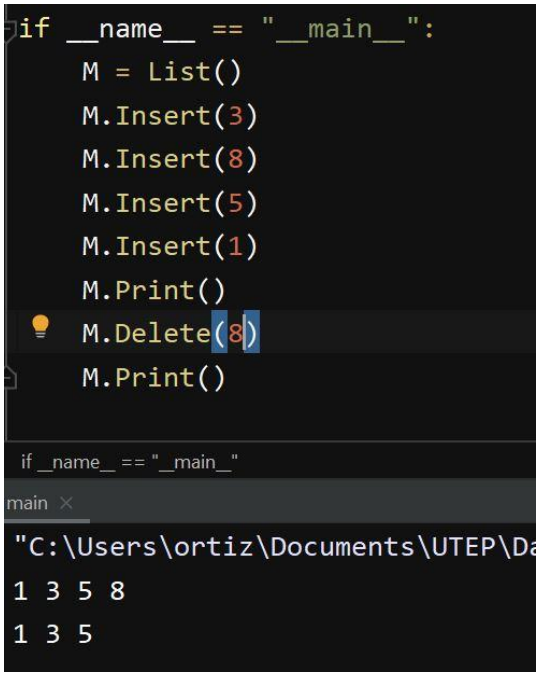
This method got quite messy because I had an idea and stuck with it but later had to take care of one too many special cases. Nonetheless, I decided to first test it in the case that the list only had one node and we decided to delete that one.



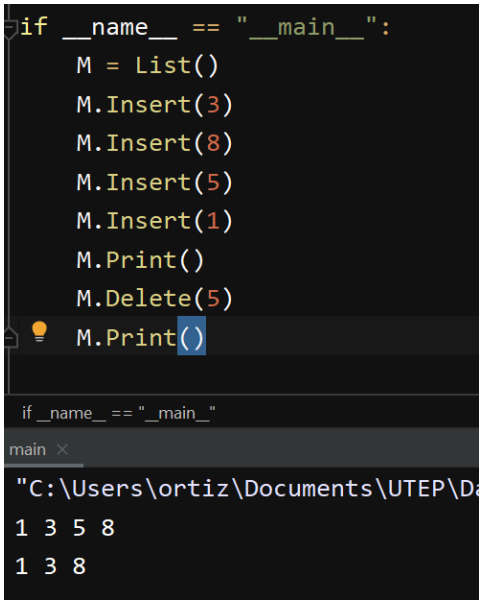
As we can see, the list is now empty. Next, I decided to test it in the case that the delete function called for the deletion of the head node of a list.



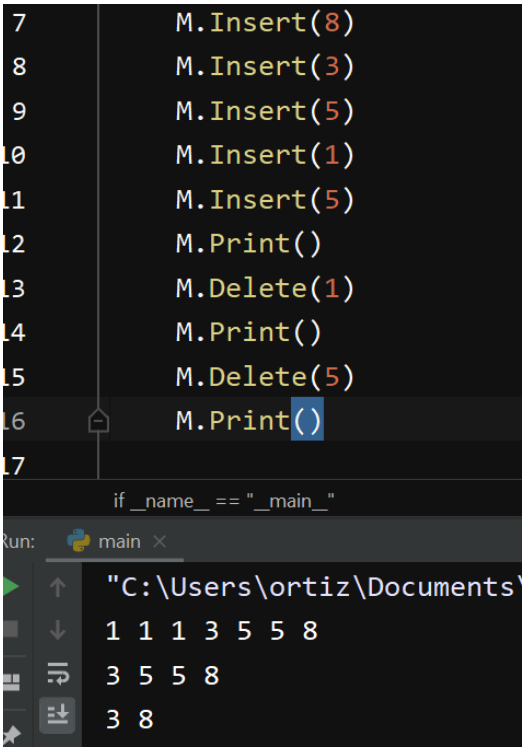
Afterwards, I decided to test it on the opposite case, when the tail node was removed. Logically, we should have less issues with this test case, but it is still good to check.



Then, I checked if the function worked just on a regular call to somewhere on the middle of the list.



Finally, I decided to check if the method would remove all the instances of a number in the case that it appeared more than once.



Merge(self, M):

For the first test run of merge, I decided to merge an existing list onto an empty one and see how the program would react.

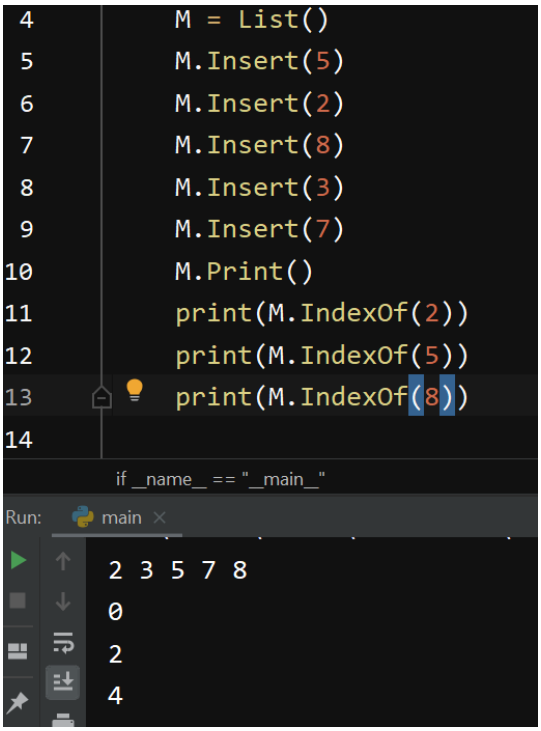
```
4       M = List()
5       M.Print()
6       M1 = List()
7       M1.Insert(5)
8       M1.Insert(3)
9       M1.Insert(7)
10      M1.Insert(8)
11      M1.Print()
12      M.Merge(M1)
13      M.Print()
14
        if __name__ == "__main__"
Run:    main
        "C:\Users\ortiz\Documents
        3 5 7 8
        3 5 7 8
```

Next, I tested to see how the program would merge two Sorted Lists, curious if the new list would maintain the property that all the data in it would be sorted in ascending order.

```
7       M.Insert(8)
8       M.Insert(7)
9       M.Print()
10      M1 = List()
11      M1.Insert(2)
12      M1.Insert(4)
13      M1.Insert(7)
14      M1.Insert(8)
15      M1.Print()
16      M1.Merge(M)
17      M1.Print()
        if __name__ == "__main__"
Run:    main
        "C:\Users\ortiz\Documents\
        3 5 7 8
        2 4 7 8
        2 3 4 5 7 7 8 8
```

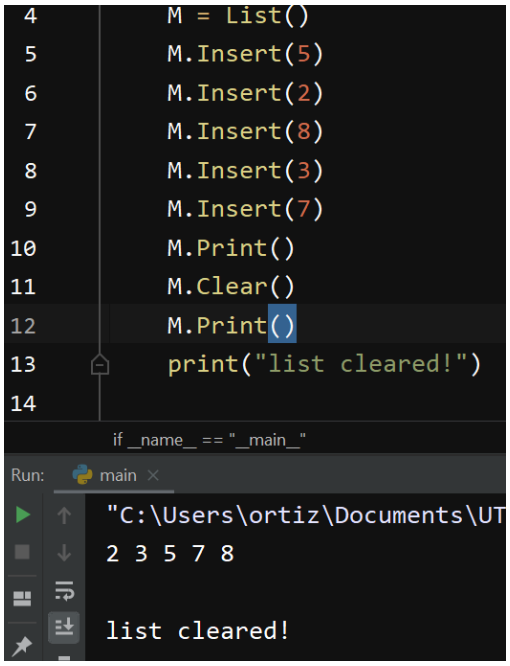
IndexOf(self, i):

To test this method, I checked if it would return the correct index for an element at the beginning of the list, one at the end, and one at the middle.



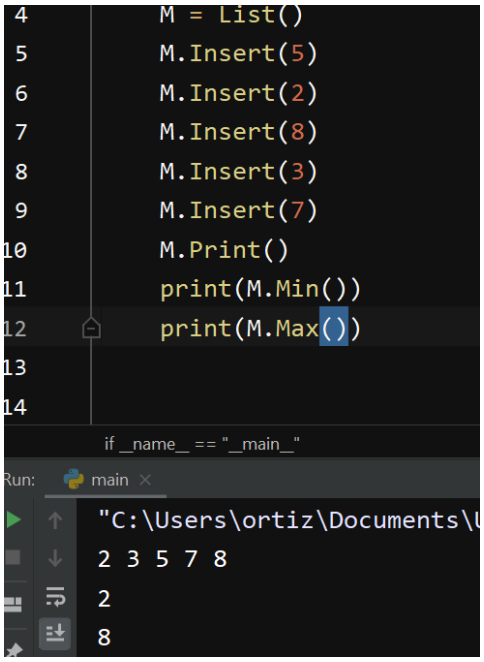
Clear(self):

For this method, I simply checked that the given list would be empty after calling it.



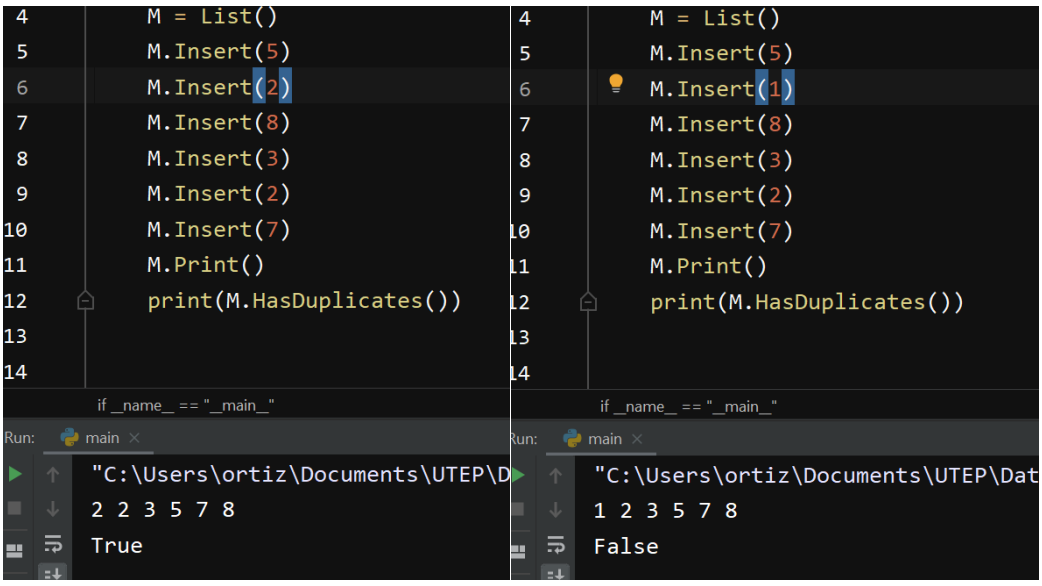
Min(self), Max(self):

Here, I checked if the functions would return the smallest and largest element in the list, respectively.



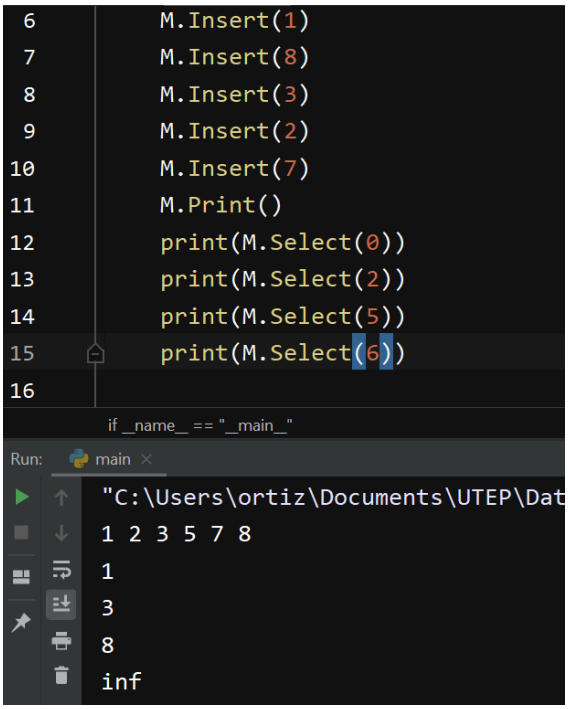
HasDuplicates(self, k):

I checked if a list with duplicates would result in the function returning true and false with a list that did not contain duplicates.



Select(self, k):

To test this method, I checked if it would return the smallest number in the list, the biggest, one in the middle, and how it would react when asked for an index that was not in the list.



Analysis:

Function	SortedList	List
Print()	O(n)	O(n)
Insert(i)	O(n)	O(1)
Delete(i)	O(n)	O(n)
Merge(M)	O(n)	O(n)
IndexOf(i)	O(n)	O(n)
Clear(i)	O(1)	O(1)
Min()	O(1)	O(n)
Max()	O(1)	O(n)
HasDuplicates()	O(n)	O(n)
Select(k)	O(n)	O(nlogn)

To complete, this part of the analysis, I implemented all these functions in a regular linked list. A lot of them could be implemented the same way, but I will now go over my implementation of those that couldn't. First off, perhaps the most obvious difference is the Insert method. The regular list implementation simply adds a new node to the end of the list or makes the new node the head if the list is empty. For this reason, the Insert method in a regular list is faster than in a Sorted List.

```
def Insert(self,x):
    if self.head is None:
        self.head = Node(x)
        self.tail = self.head
    else:
        self.tail.next = Node(x)
        self.tail = self.tail.next
```

The next difference comes when implementing the min() method. For the regular list, I had to traverse the entire list and then return the smallest rather than simply the head node. Due to this, the runtime complexity of this function is O(n).

```
def Min(self):
    if(self.head is None):
        return math.inf
    t = self.head
    smallest = t.data
    while(t is not None):
        if(t.data < smallest):
            smallest = t.data
        t = t.next
    return smallest
```

The same happens with the Max() method.

```
def Max(self):
    if(self.head is None):
        return math.inf
    t = self.head
    largest = t.data
    while(t is not None):
        if(t.data > largest):
            largest = t.data
        t = t.next
    return largest
```

My implementation for the HasDuplicates() in the regular list is quite bizarre. The asymptotic complexity is the same as the Sorted List in theory, but if were to look at the space complexity this algorithm takes much more. First, I traversed the list once to get the length of it. Then, I traversed the list again and added all the elements to a set. If there is a duplicate, the set will get

rid of it. Therefore, I would compare the length of the list to the length of the set and return True if they were different.

```
def HasDuplicates(self):
    if(self.head is None):
        return False
    else:
        counter = 0
        t = self.head
        while(t is not None):
            counter += 1
            t = t.next
        t = self.head
        new_set = set()
        while(t is not None):
            new_set.add(t.data)
            t = t.next
        if(len(new_set) < counter):
            return True
        else:
            return False
```

Finally, the implementation of the Select() method was also different. Perhaps my implementation was a bit cheap, because I added all the elements of the list to a traditional python list. Then, I sorted this traditional list using python's sort function. From here, I would return the value of the new list at k. I am assuming the runtime complexity of python's sort is nlogn, so the runtime of this function would be O(nlogn)

```
def Select(self, k):
    if(self.head is None):
        return math.inf
    new_list = []
    t = self.head
    while(t is not None):
        new_list.append(t.data)
        t = t.next
    new_list.sort()
    if(k > len(new_list)):
        return math.inf
    return new_list[k]
```

Conclusion

This lab was very fun because it taught us how to use pointers to implement all of these methods in a linked list. A lot of times, we like to visualize a linked list as an actual list with the elements actually connected when in reality, it is simply a group of pointers that make it all work. Due to this, we must keep in mind these pointers when approaching a linked list problem. This lab really helped me understand how to work with linked lists on a deeper level. Finally, we implemented a modified version of the linked list and saw if the trade offs of having a slower insert() method were worth it when implementing the other methods.

Appendix

```
1   import math
2
3   #Node object: used to hold the data
4   class Node(object):
5       def __init__(self, data, next = None):
6           self.data = data
7           self.next = next
8
9   #List class: has references between nodes,
10  #creating a list
11  class List(object):
12      def __init__(self):
13          self.head = None
14          self.tail = None
15
16  #Clear(): removes the list of all its elements
17      def Clear(self):
18          self.head = None
```

```
20      #Delete(): deletes a specific number
21      #input i: the specific value we want to delete
22          def Delete(self, i):
23              while(self.head.data == i):
24                  if(self.head.next is None):
25                      self.head = None
26                      return
27                  self.head = self.head.next
28              t = self.head
29              j = t
30              k = self.tail
31              while(t.next is not None):
32                  if(t.next.data == i):
33                      t.next = t.next.next
34                  else:
35                   t = t.next
36              k = t
37              self.head = j
```

```
38              self.tail = k
39
40      #HasDuplicates(): Checks if the list has duplicates
41          def HasDuplicates(self):
42              if(self.head is None):
43                  return False
44              t = self.head
45              while(t.next is not None):
46                  if(t.data == t.next.data):
47                      return True
48                  t = t.next
49              return False
50
51      #IndexOf(): returns the index of a specific number
52      #input i: the number whose index we are searching for
53          def IndexOf(self, i):
54              counter = 0
55              t = self.head
```

```
56          while(t.data != i):
57              counter += 1
58              t = t.next
59              if(t is None):
60                  return -1
61          return counter
62
63  #Insert(): The main meat of the Sorted List, inserts
64   #new data to list
65  #input x: the number we want to insert
66      def Insert(self, x):
67          if self.head is None:
68              self.head = Node(x)
69              self.tail = self.head
70          else:
71              if(x >= self.tail.data):
72                  self.tail.next = Node(x)
73                  self.tail = self.tail.next
```

```
74              elif(x <= self.head.data):
75                  i = self.head
76                  t = Node(x)
77                  t.next = i
78                  self.head = t
79              else:
80                  t = self.head
81                  i = t
82                  while(t is not None and t.next.data < x):
83                      t = t.next
84                  insertion = Node(x)
85                  insertion.next = t.next
86                  t.next = insertion
87                  self.head = i
88
89  #Max(): returns the max elemenet
90  #output: the max element
91      def Max(self):
```

```
92          if(self.head is None):
93              return math.inf
94          return self.tail.data
95
96      def Merge(self, M):
97          t = self.head
98          i = M.head
99          new_list = List()
100         while(t is not None):
101             new_list.Insert(t.data)
102             t = t.next
103         while(i is not None):
104             new_list.Insert(i.data)
105             i = i.next
106         self.head = new_list.head
107
108 #Min(): returns the minimum element
109 #output: the minimum element
```

```
110     def Min(self):
111         if(self.head is None):
112             return math.inf
113         return self.head.data
114 
115 #Print(): prints the contents of the list
116 #output: the contents of the list
117     def Print(self):
118         t = self.head
119         while t is not None:
120             print(t.data, end=' ')
121             t = t.next
122         print()
123 
124 #Select(): selects the kth smallest element
125  #Input k: the index we are searching for
126 #Output: the value of the list at k
127     def Select(self, k):
```

```
128         if(self.head is None):
129             return math.inf
130         t = self.head
131         while(t is not None  and k > 0):
132             t = t.next
133             k -= 1
134         if(t is None):
135             return math.inf
136         return t.data
```

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