Laurence Labayen

80358755

10/07/2019

Lab #3

CS2302 MW 10:30am

Description:

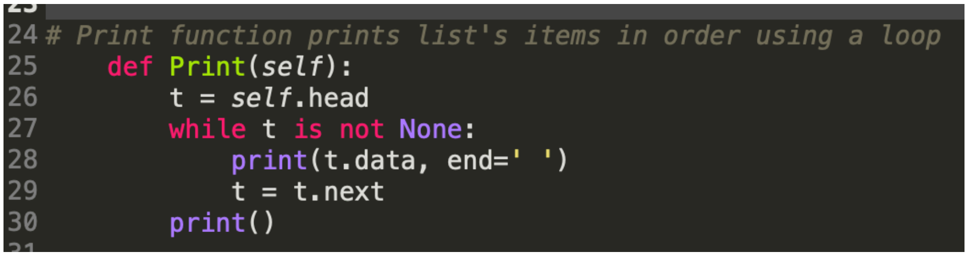
We were given the task of implementing different functions of a sorted linked list for this lab as well as including the running times of each function compared to an unsorted linked list. Using numbers as data inside the nodes, the list must be sorted at all times. Keeping that in mind, inserting and deleting nodes must be done in a way where the list will stay sorted.

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

Solution design and Implementation:

Print:

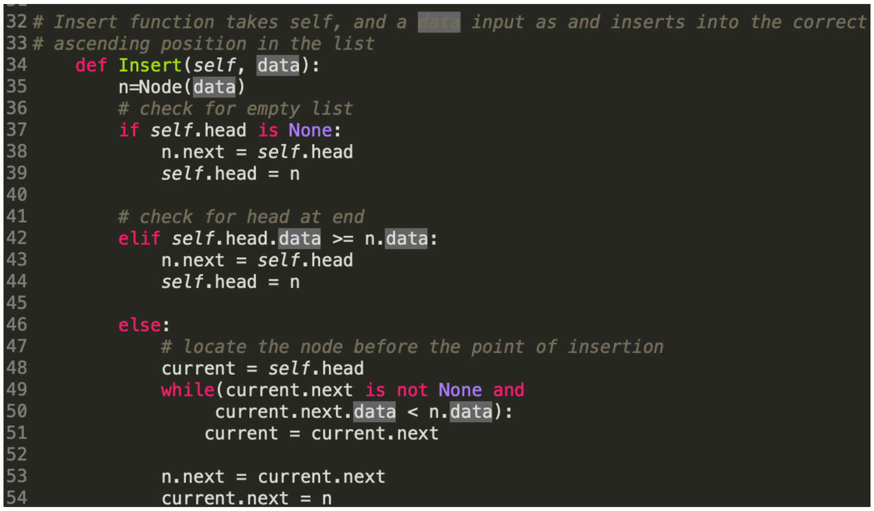
The print function is self explanatory, it will print all the nodes in the list. Since the list is always sorted, printing the list will always be in ascending order. To do this, I set a temporary node that takes the head. Then I set up a loop to keep going as long as the temporary pointer is not None/Null. As it iterates through the list, it prints out each data/element then moves to the next node.



```
# Print function prints list's items in order using a loop
    def Print(self):
        t = self.head
        while t is not None:
            print(t.data, end=' ')
            t = t.next
        print()
```

Insert:

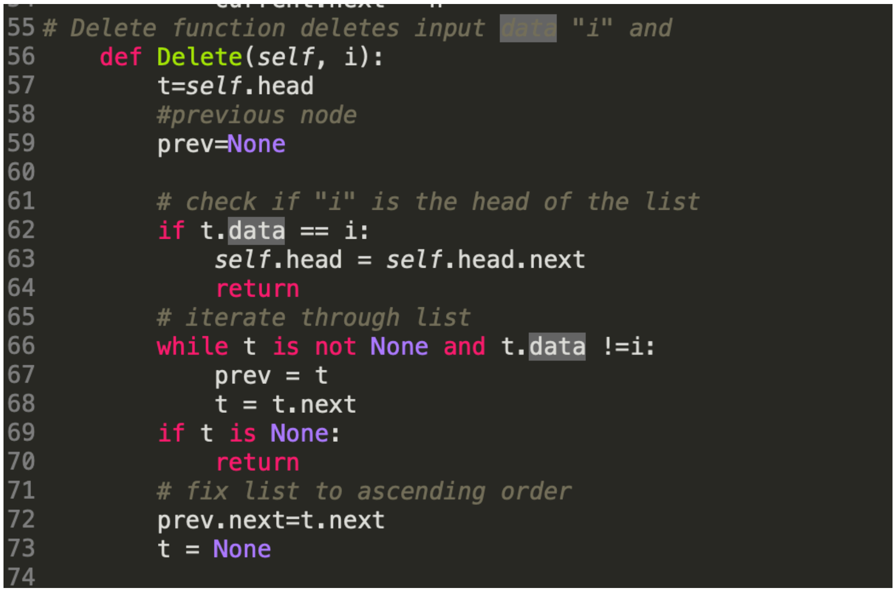
Insert is one of the functions that keeps the list sorted at all times. As it takes in a new node and data, it will correctly insert it in the list at the correct position. With that in mind, the first thing that needs to be done is to check if the list is empty. If it is, just point the head to a new node containing the new data. Otherwise, that means the list is not empty. Now we check if the data at the head node is more than or the same as the new node's data. If it is, insert the new node at the head. Finally if none of those conditions are met, we traverse the list to find the correct position for our new node.



```
# Insert function takes self, and a data input as and inserts into the correct
# ascending position in the list
    def Insert(self, data):
        n=Node(data)
        # check for empty list
        if self.head is None:
            n.next = self.head
            self.head = n

        # check for head at end
        elif self.head.data >= n.data:
            n.next = self.head
            self.head = n

        else:
            # locate the node before the point of insertion
            current = self.head
            while(current.next is not None and
                 current.next.data < n.data):
                current = current.next

            n.next = current.next
            current.next = n
```

Delete:

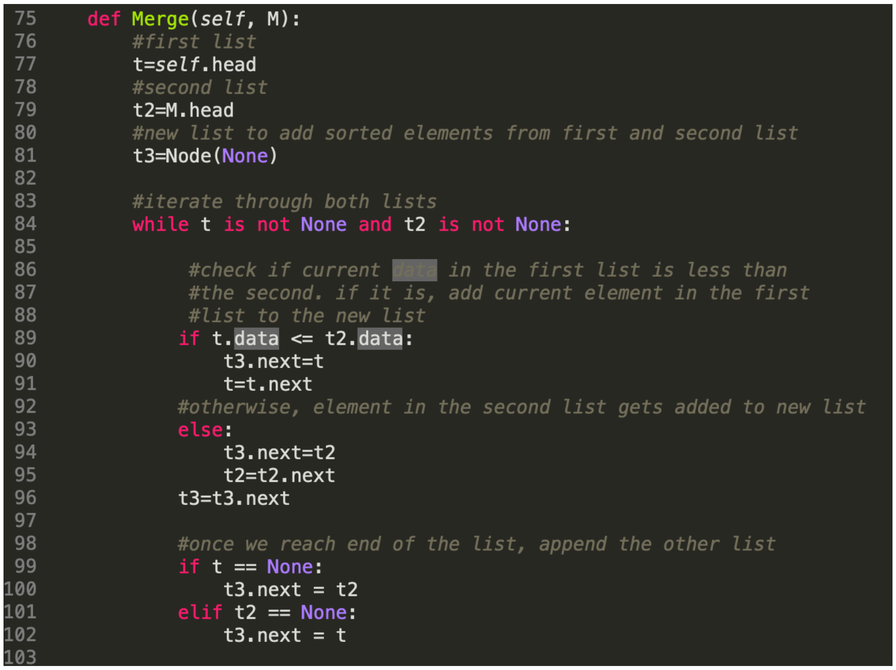
The delete function takes “i” as a parameter to be used as a key for the node that needs to be removed. This function keeps the list sorted as well by making sure the previous node points to the node after the deleted node. To do this, I set 2 temporary pointers, prev and t. If the list is empty, just return and do nothing. If the the key/“i” is the head of the list, move the head pointer to the next node. Otherwise, we traverse the list with the condition that we have not hit the end of the list and if the current data were looking at is not equal to the key/“i”. During each iteration, we will be advancing both pointers. Once we get out, that means we have found the node with “i”. We use our prev pointer and t pointer to fix the list in ascending order.



```python
# Delete function deletes input data "i" and
    def Delete(self, i):
        t=self.head
        #previous node
        prev=None

        # check if "i" is the head of the list
        if t.data == i:
            self.head = self.head.next
            return
        # iterate through list
        while t is not None and t.data !=i:
            prev = t
            t = t.next
        if t is None:
            return
        # fix list to ascending order
        prev.next=t.next
        t = None

```

Merge:

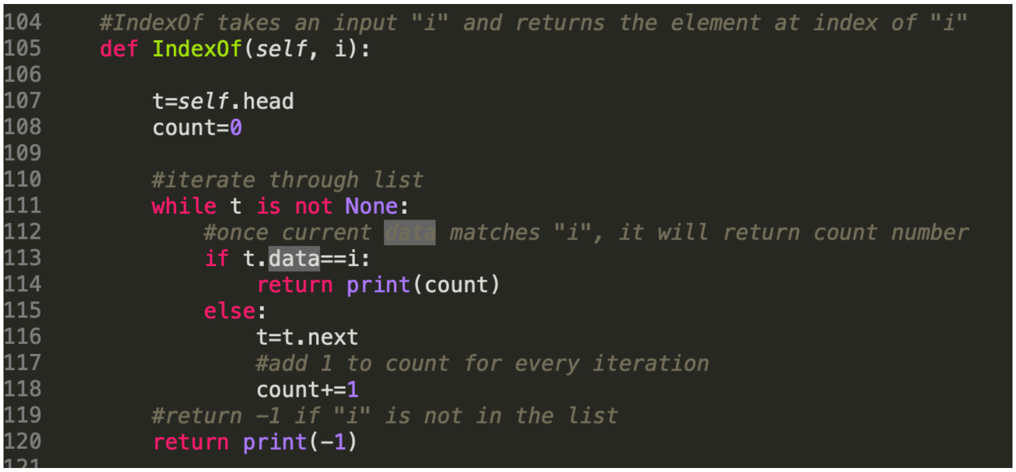
The Merge function takes another sorted list as an argument and merges it with the original list. To do this, I set temporary nodes for the head of the original, second list and created a new node for the third. Then it goes into a loop as long as the first and second temporary pointers are not none. While inside the loop, I compare the current node data on both lists. If the current data in the first list is smaller than the second, add the current element in the first list to the new list then advance the first list pointer to the next node. Otherwise, the current data in the second list goes into the new list, then advance the pointer for the second list. Once the end of the list is reached, append the other list.



```
    def Merge(self, M):
        #first list
        t=self.head
        #second list
        t2=M.head
        #new list to add sorted elements from first and second list
        t3=Node(None)

        #iterate through both lists
        while t is not None and t2 is not None:

             #check if current data in the first list is less than
             #the second. if it is, add current element in the first
             #list to the new list
            if t.data <= t2.data:
                t3.next=t
                t=t.next
            #otherwise, element in the second list gets added to new list
            else:
                t3.next=t2
                t2=t2.next
            t3=t3.next

            #once we reach end of the list, append the other list
            if t == None:
                t3.next = t2
            elif t2 == None:
                t3.next = t
```

IndexOf:

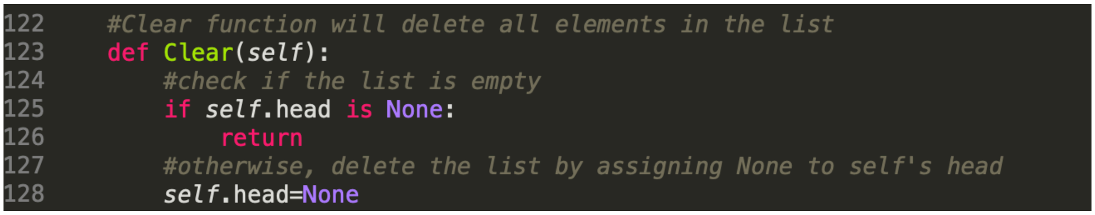
This function takes “i” as an argument, and returns the location or index of it. I set a temporary pointer t and created a counter variable. Then, it iterate through the list with a while loop until the end of the list. Inside the loop, I check if the current data is equal to the value of i, if it is, we return and print the count number. Otherwise, we keep iterating through the list by advancing the temporary pointer and add 1 to the counter. If it reaches the end of the list without finding i, we return and print -1.



```
    #IndexOf takes an input "i" and returns the element at index of "i"
    def IndexOf(self, i):

        t=self.head
        count=0

        #iterate through list
        while t is not None:
            #once current data matches "i", it will return count number
            if t.data==i:
                return print(count)
            else:
                t=t.next
                #add 1 to count for every iteration
                count+=1
        #return -1 if "i" is not in the list
        return print(-1)
```

Clear:

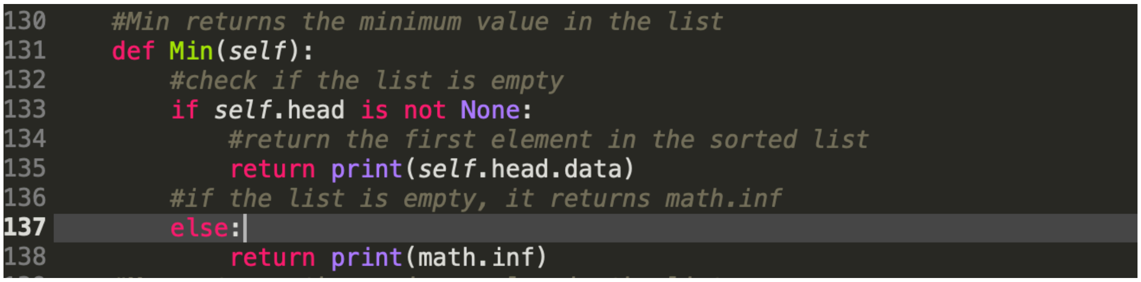
The Clear function deletes all the elements in the list. We check if the list is empty. If it is, we will return nothing. Otherwise, we just set the list head to None.



```
    #Clear function will delete all elements in the list
    def Clear(self):
        #check if the list is empty
        if self.head is None:
            return
        #otherwise, delete the list by assigning None to self's head
        self.head=None
```

Min:

Min function returns the minimum element in the list. Since this list is always sorted, we only need to return the first element in the list. If the list is not empty, we simply return the head's data. Otherwise, we return math.inf

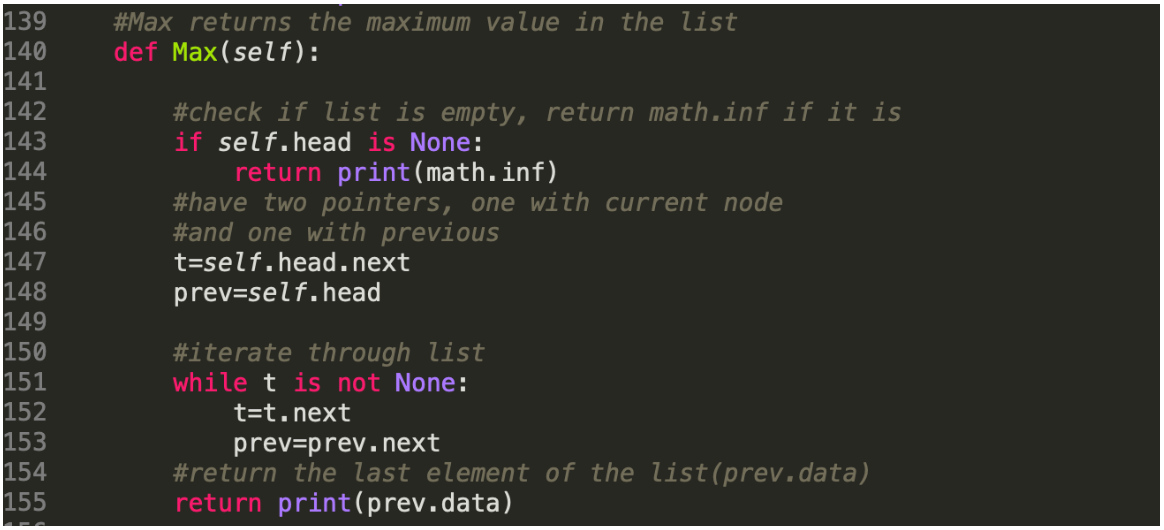


```
    #Min returns the minimum value in the list
    def Min(self):
        #check if the list is empty
        if self.head is not None:
            #return the first element in the sorted list
            return print(self.head.data)
        #if the list is empty, it returns math.inf
        else:
            return print(math.inf)
```

Max:

This function returns the biggest element in the list. Since this list is always sorted, we only need to return the last element in the list. We check if the list is empty, if it is, return math.inf. Otherwise, iterate through the list using 2 pointers with the current and previous nodes. Once the current node reaches the end, prev will be pointing to the last node. We can just return the data of the previous pointer.

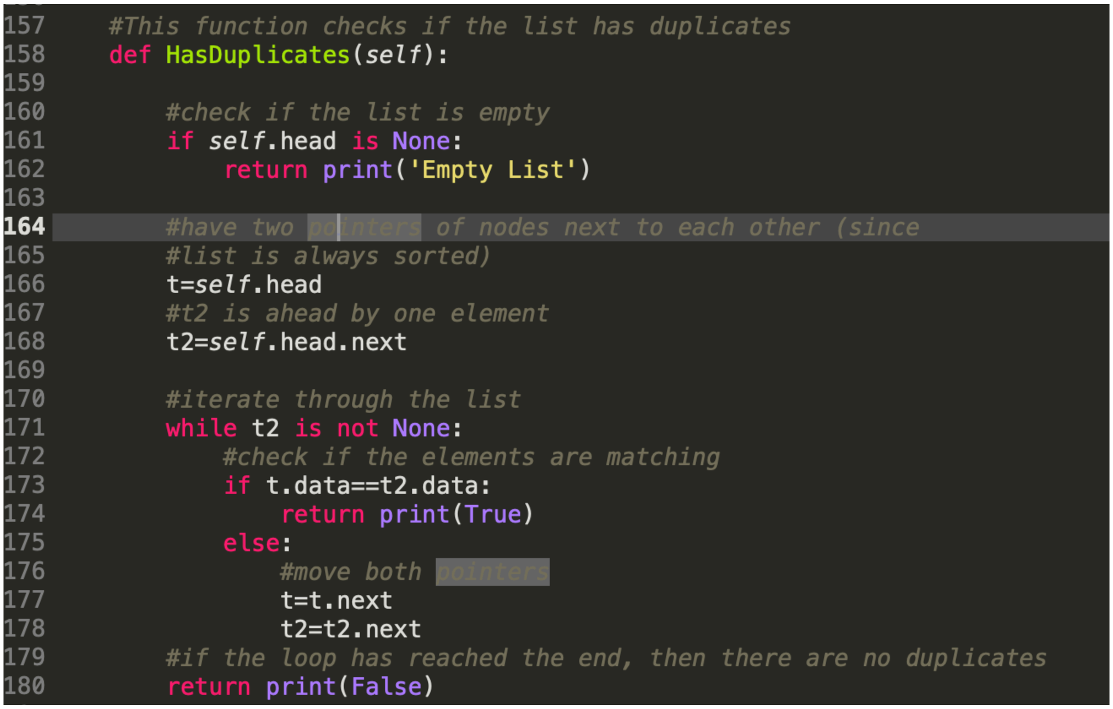
*This function could have been done at O(1) running time if a tail pointer was required and implemented



```
    #Max returns the maximum value in the list
    def Max(self):

        #check if list is empty, return math.inf if it is
        if self.head is None:
            return print(math.inf)
        #have two pointers, one with current node
        #and one with previous
        t=self.head.next
        prev=self.head

        #iterate through list
        while t is not None:
            t=t.next
            prev=prev.next
        #return the last element of the list(prev.data)
        return print(prev.data)
```

HasDuplicates:

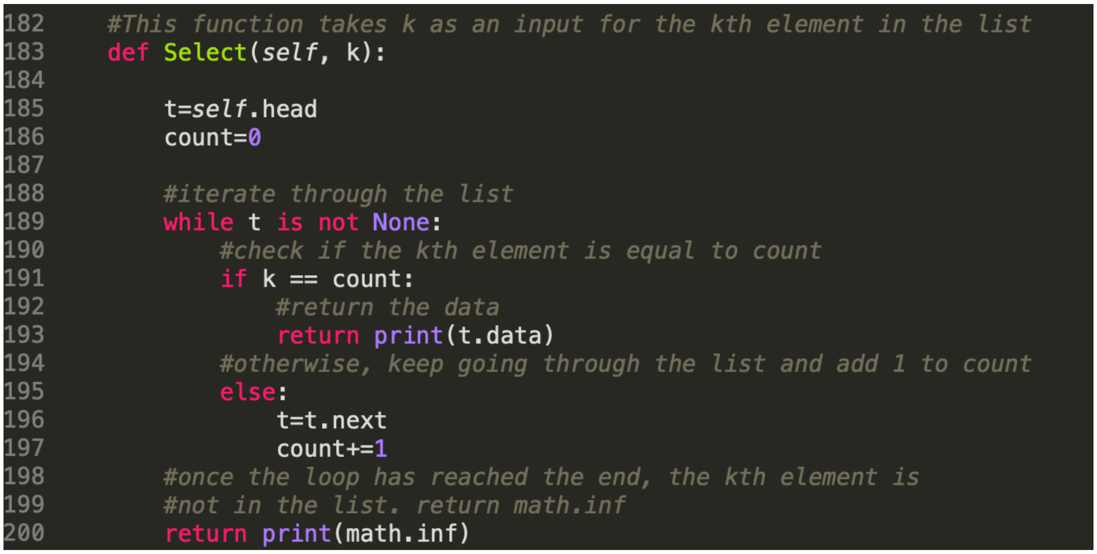
This function checks if the list has duplicates. It returns true if there are and false if otherwise. We first check if the list is empty. If it is, we return and print “Empty List”. We set 2 temporary pointers for the first and second node. Then we iterate thought the list, comparing adjacent elements if they match. If they do, we return True. Otherwise, we keep moving both pointers. Since this list is always sorted, duplicate elements will always be next to each other. If we reach the end of the list, we return False.



```
    #This function checks if the list has duplicates
    def HasDuplicates(self):

        #check if the list is empty
        if self.head is None:
            return print('Empty List')

        #have two pointers of nodes next to each other (since
        #list is always sorted)
        t=self.head
        #t2 is ahead by one element
        t2=self.head.next

        #iterate through the list
        while t2 is not None:
            #check if the elements are matching
            if t.data==t2.data:
                return print(True)
            else:
                #move both pointers
                t=t.next
                t2=t2.next
        #if the loop has reached the end, then there are no duplicates
        return print(False)
```

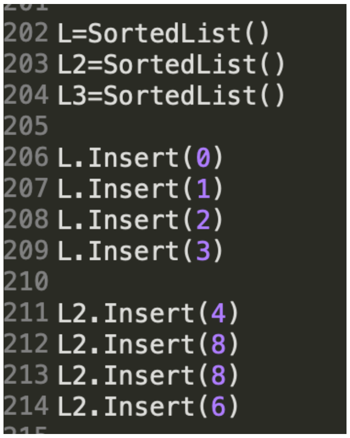
Select:

This function is similar to lab 2, it returns the kth smallest element in the list. We first start by setting a pointer to the head and initialize a counter. We then iterate through the list and check if k is equal to the counter, if its the first element, return the data. Otherwise, we keep going through the list and add 1 to the counter for every iteration. If we reach the end, we return math.inf



```python
    #This function takes k as an input for the kth element in the list
    def Select(self, k):

        t=self.head
        count=0

        #iterate through the list
        while t is not None:
            #check if the kth element is equal to count
            if k == count:
                #return the data
                return print(t.data)
            #otherwise, keep going through the list and add 1 to count
            else:
                t=t.next
                count+=1
        #once the loop has reached the end, the kth element is
        #not in the list. return math.inf
        return print(math.inf)
```

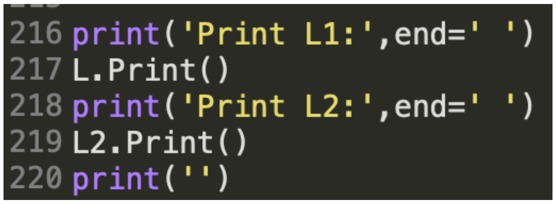
Experimental Results:



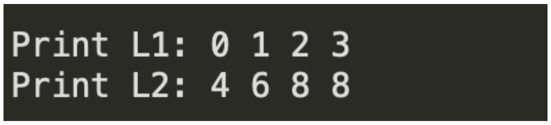
```python
L=SortedList()
L2=SortedList()
L3=SortedList()

L.Insert(0)
L.Insert(1)
L.Insert(2)
L.Insert(3)

L2.Insert(4)
L2.Insert(8)
L2.Insert(8)
L2.Insert(6)
```

Adding elements into the list using Insert function

Using the Print function to show the SortedList

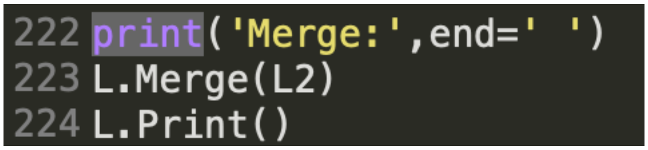


```python
print('Print L1:',end=' ')
L.Print()
print('Print L2:',end=' ')
L2.Print()
print('')
```

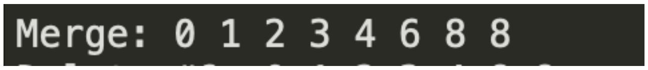


```
Print L1: 0 1 2 3
Print L2: 4 6 8 8
```

Merge function and printing list after calling

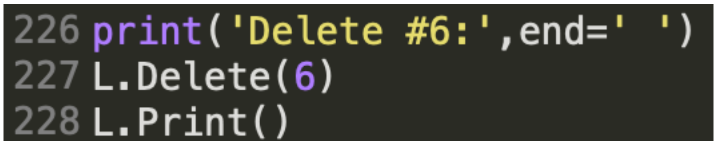


```python
print('Merge:',end=' ')
L.Merge(L2)
L.Print()
```

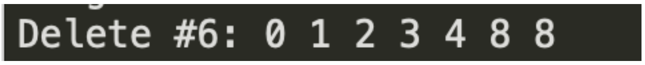


```
Merge: 0 1 2 3 4 6 8 8
```

Delete(6) and printing list after calling

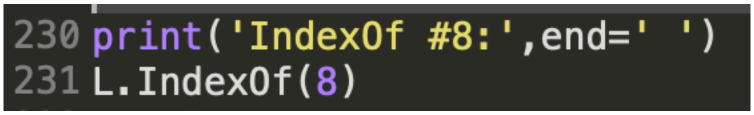


```python
print('Delete #6:',end=' ')
L.Delete(6)
L.Print()
```

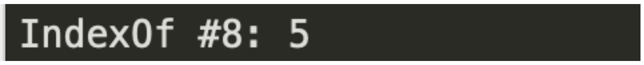


```
Delete #6: 0 1 2 3 4 8 8
```

IndexOf(8)

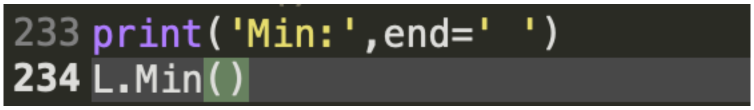


```python
print('IndexOf #8:',end=' ')
L.IndexOf(8)
```

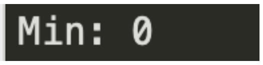


```
IndexOf #8: 5
```

Min Function

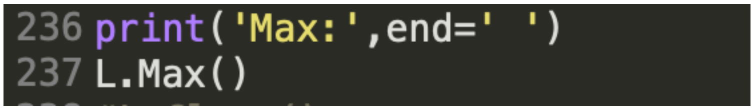


```
print('Min:',end=' ')
L.Min()
```

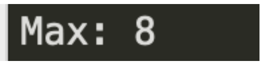


```
Min: 0
```

Max Function

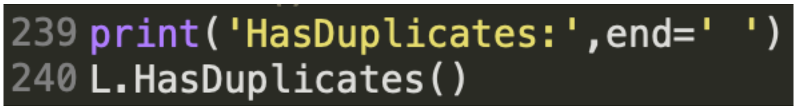


```
print('Max:',end=' ')
L.Max()
```

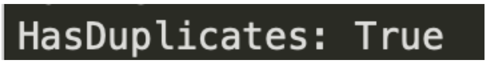


```
Max: 8
```

HasDuplicates Function

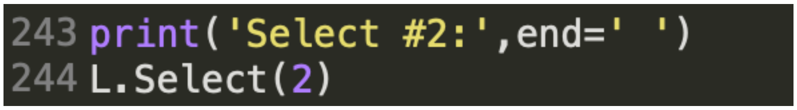


```
print('HasDuplicates:',end=' ')
L.HasDuplicates()
```

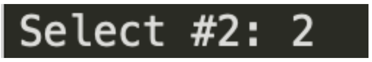


```
HasDuplicates: True
```

Select(2) Function

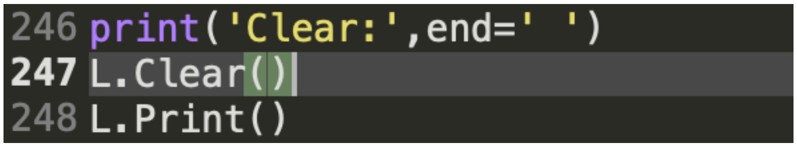


```
print('Select #2:',end=' ')
L.Select(2)
```

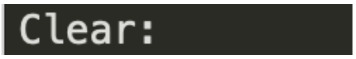


```
Select #2: 2
```

Clear Function



```
print('Clear:',end=' ')
L.Clear()
L.Print()
```



```
Clear:
```

Conclusion:

This lab was very helpful to learn different functions of linked lists. Although having the list sorted at all times makes easier for some functions to work more efficiently (min and max), most of the functions had the same running time as if the list was not sorted (with the exception of merge). Compared to other labs, this was much easier, less time consuming and less confusing.

Appendix:

```
"""
Laurence Justin Labayen
Lab 3
CS2302 Data Structures
MW 10:30
Professor: Olac Fuentes
TA: Anindita Nath
"""
import math

class Node(object):
    # Constructor
    def __init__(self, data, next=None):
        self.data = data
        self.next = next

#List Functions
class SortedList(object):
    # Constructor
    def __init__(self):
        self.head = None


# Print function prints list's items in order using a loop
    def Print(self):
        t = self.head
        while t is not None:
            print(t.data, end=' ')
            t = t.next
        print()

# Insert function takes self, and a data input as and inserts into the correct
# ascending position in the list
    def Insert(self, data):
        n=Node(data)
        # check for empty list
        if self.head is None:
            n.next = self.head
            self.head = n

        # check for head at end
        elif self.head.data >= n.data:
            n.next = self.head
            self.head = n

        else:
            # locate the node before the point of insertion
            current = self.head
            while(current.next is not None and
                 current.next.data < n.data):
                current = current.next

            n.next = current.next
            current.next = n
# Delete function deletes input data "i" and
    def Delete(self, i):
        t=self.head
        #previous node
        prev=None

        # check if "i" is the head of the list
        if t.data == i:
            self.head = self.head.next
            return
        # iterate through list
        while t is not None and t.data !=i:
            prev = t
            t = t.next
        if t is None:
            return
        # fix list to ascending order
        prev.next=t.next
        t = None

    def Merge(self, M):
        #first list
        t=self.head
        #second list
        t2=M.head
        #new list to add sorted elements from first and second list
        t3=Node(None)

        #iterate through both lists
        while t is not None and t2 is not None:

             #check if current data in the first list is less than
             #the second. if it is, add current element in the first
             #list to the new list
            if t.data <= t2.data:
                t3.next=t
                t=t.next
            #otherwise, element in the second list gets added to new list
            else:
                t3.next=t2
                t2=t2.next
            t3=t3.next

            #once we reach end of the list, append the other list
            if t == None:
                t3.next = t2
            elif t2 == None:
                t3.next = t

    #IndexOf takes an input "i" and returns the element at index of "i"
    def IndexOf(self, i):

        t=self.head
        count=0

        #iterate through list
        while t is not None:
            #once current data matches "i", it will return count number
            if t.data==i:
                return print(count)
            else:
                t=t.next
                #add 1 to count for every iteration
                count+=1
        #return -1 if "i" is not in the list
        return print(-1)

    #Clear function will delete all elements in the list
    def Clear(self):
        #check if the list is empty
        if self.head is None:
            return
        #otherwise, delete the list by assigning None to self's head
        self.head=None

    #Min returns the minimum value in the list
    def Min(self):
        #check if the list is empty
        if self.head is not None:
            #return the first element in the sorted list
            return print(self.head.data)
        #if the list is empty, it returns math.inf
        else:
            return print(math.inf)
    #Max returns the maximum value in the list
    def Max(self):

        #check if list is empty, return math.inf if it is
        if self.head is None:
            return print(math.inf)
        #have two pointers, one with current node
```

```
        #and one with previous
        t=self.head.next
        prev=self.head

        #iterate through list
        while t is not None:
            t=t.next
            prev=prev.next
        #return the last element of the list(prev.data)
        return print(prev.data)

    #This function checks if the list has duplicates
    def HasDuplicates(self):

        #check if the list is empty
        if self.head is None:
            return print('Empty List')

        #have two pointers of nodes next to each other (since
        #list is always sorted)
        t=self.head
        #t2 is ahead by one element
        t2=self.head.next

        #iterate through the list
        while t2 is not None:
            #check if the elements are matching
            if t.data==t2.data:
                return print(True)
            else:
                #move both pointers
                t=t.next
                t2=t2.next
        #if the loop has reached the end, then there are no duplicates
        return print(False)

    #This function takes k as an input for the kth element in the list
    def Select(self, k):

        t=self.head
        count=0

        #iterate through the list
        while t is not None:
            #check if the kth element is equal to count
            if k == count:
                #return the data
                return print(t.data)
            #otherwise, keep going through the list and add 1 to count
            else:
                t=t.next
                count+=1
        #once the loop has reached the end, the kth element is
        #not in the list. return math.inf
        return print(math.inf)

L=SortedList()
L2=SortedList()
L3=SortedList()

L.Insert(0)
L.Insert(1)
L.Insert(2)
L.Insert(3)

L2.Insert(4)
L2.Insert(8)
L2.Insert(8)
L2.Insert(6)
print()
print('Print L1:',end=' ')
L.Print()
print('Print L2:',end=' ')
L2.Print()
print('')

print('Merge:',end=' ')
L.Merge(L2)
L.Print()

print('Delete #6:',end=' ')
L.Delete(6)
L.Print()

print('IndexOf #8:',end=' ')
L.IndexOf(8)
#L.Clear()
print('Min:',end=' ')
L.Min()

print('Max:',end=' ')
L.Max()
#L.Clear()
print('HasDuplicates:',end=' ')
L.HasDuplicates()


print('Select #2:',end=' ')
L.Select(2)

print('Clear:',end=' ')
L.Clear()
L.Print()
```

Academic Honesty Statement:

“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.”

-Laurence Labayen