Nathan Aun

6/17/19

Lab #2 – Linked Lists

CS2302 – Data Structures

Summer 2019

Introduction

The problem that was posed for this lab wasn’t exactly a problem, but it was more of a direction, write methods that will be able to do the following operations: Append, Prepend, Insert After, Remove, Search, Print, Print Reverse, Sort, Is Empty, Get length, Copy, ItemAt, Pop, Count, Index, Clear, Sublist, and finally Reverse. The overall goal of this lab is to have a file that will be able to successfully execute all of these functions.

Proposed Solution

The proposed implementation for each of the methods is as follows:

Append: The point of this method is to add a new value to the end of a linked list. It checks if the list is empty, if it is then it adds the new node as the head and the tail, if not then it adds the new node to the tail’s next, then sets the new node to be the tail of the list.

Prepend: The point of this method is to add a new value to the front of a linked list. It checks if the list is empty, if it is then it adds the new node as the head and tail. If the list is not empty, it sets the new node’s next to point to the head and sets the new node to be the head of the list.

InsertAfter: The point of this method is to insert a new value after some existing value in the linked list. InsertAfter takes 3 values, a list, the value to be added after, and the value to be added to the list. It will check to see if the list is empty, and if it is it will notify the user. If it is not empty, it will go through the list until it finds the value to be added after, denoted by a “w”. If it finds w, it will set w’s next to be the new node and the new node’s next to be w’s next before we changed it. If it doesn’t find w, it will let the user know that it is not in the list.

Remove: The point of this method is to remove a value from the list. Remove will first check to see if the list is empty, if it is it will notify the user that you cannot remove a value from an empty list. If the list is not empty, it will iterate through the list until it finds that the next value is ‘x’, the value to be removed. If it is, then it will set the node before x to have a next pointer that is the same as x, basically skipping over ‘x’ when you go through the list. If x is never found it will notify the user.

Search: The point of this method is to find if an element is within the list. Search will check if the list is empty. If it is, execution will end. If not, it will iterate through the list until it finds ‘x’. If it does find it, it will return the value of x, otherwise it will let the user know that it did not find it.

Print: The purpose of this method is to print all the elements in a list. It will just go through the list and print until it runs out of elements to print

PrintReverse: The purpose of this method is to print all the elements in reverse order. This is done recursively, with our base case printing the current data and the recursive case will call the PrintReverse method and then after that executes it will print the current node.

Sort: The purpose of this method is to sort all the elements in the linked list to be in ascending order. The implementations follows bubble sort for the most part. It starts off doing one pass through the list in order to get the size, and after that it follows bubble sort exactly, by comparing two adjacent values and, if necessary, switching them around so that the larger one is on the right side. It will keep doing this until all elements have been sorted.

isEmpty: The point of this method is to check if the list is empty. A lot of previous methods rely on this one to make their code cleaner. It checks if L.head exists and if it doesn’t the list must be empty, so it returns true. If it does, then the list must not be empty so it returns false.

GetLength: The point of this method is to determine the length of the list. It will do this by iterating through the list and keeping count of every node it passes by.

Copy: The point of this method is to return a copy of the input list, and it does this by making a new list, and iterating through the original list. Every node it reaches, it appends the data from it to the new list, making it a copy of the original, and it returns the copy.

ItemAt: The point of this method is to return the value of an item at a given index in the list. I will iterate through the list as long as the index has not been reached and return the data value at that index.

Pop: The point of this method is to an item at a given index in a list. The program will iterate through until it reaches the node before this item. Then it will set the next of the previous node to the next of the node being removed, essentially skipping it in the list.

Count: The point of this method is to find how many times a certain variable appears in a list. It will, when given a list, iterate through and every time it sees the value it will increment one. It will then return this value.

Index: The point of this method is to find the index of a current value. The program will iterate through until it finds the data it needs, then it will return the index of that data point (where the head of the list is index 0)

Clear: The point of this method is to empty the list. It sets both the head and the tail to be None

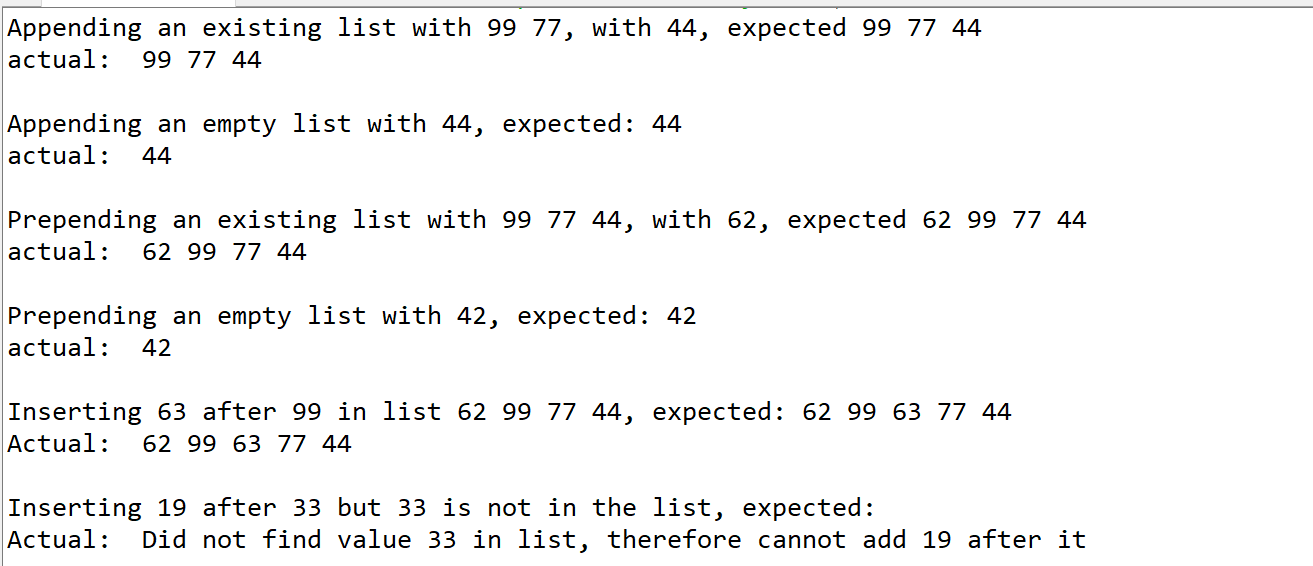
Sublist: The point of this method is to make a sublist from the original list and return it. It does it by going to where start is in the list, then making new list and adding a copy of every node up to where the end index is (non-inclusive). After that, return the new list.

Reverse: The point of this method is to reverse the values in a given list. The method does one pass and stores all the values in a native python list, then it does another pass through the linked list and changes the first value in the linked list with the last value in the native list, and iterates backwards through the native list and forwards through the linked list to successfully swap them around.

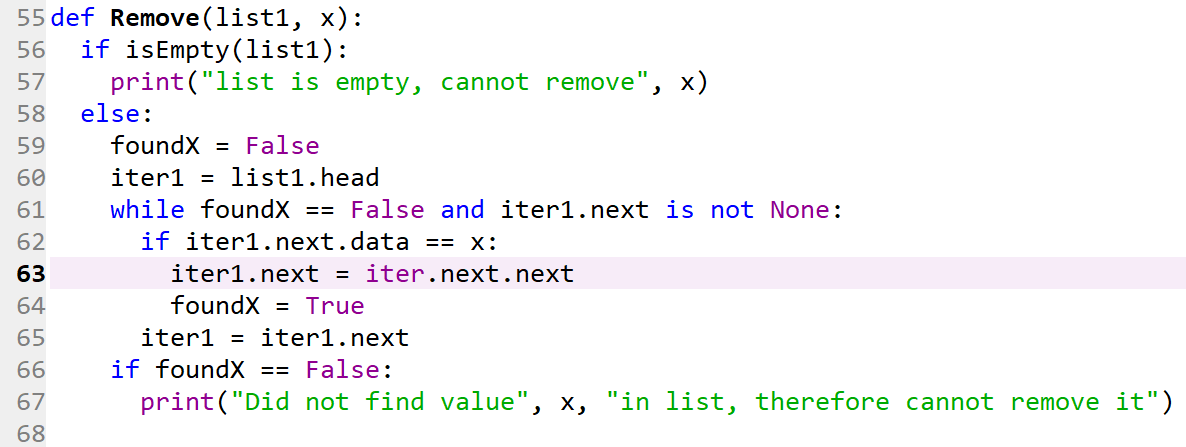
Experimental Results

In testing this lab after posting my code I have noticed a few errors in it that will not be able to be fixed in the final submission of the lab, however I will explain what went wrong in this session as well as how it could’ve been fixed. These will be brought up when they come up

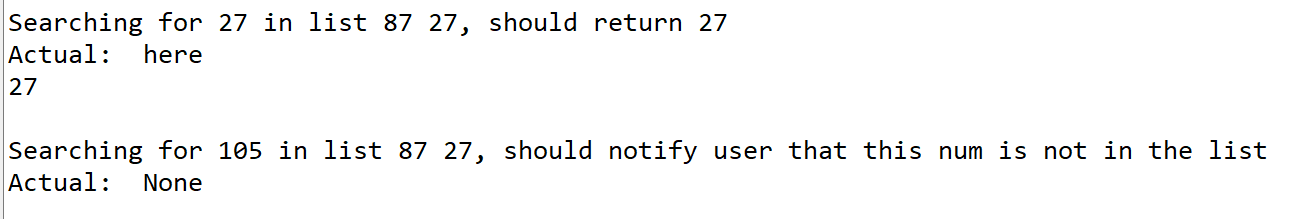
The first three methods, Append, Prepend, and InsertAfter all work as intended as evidenced by the following test cases:



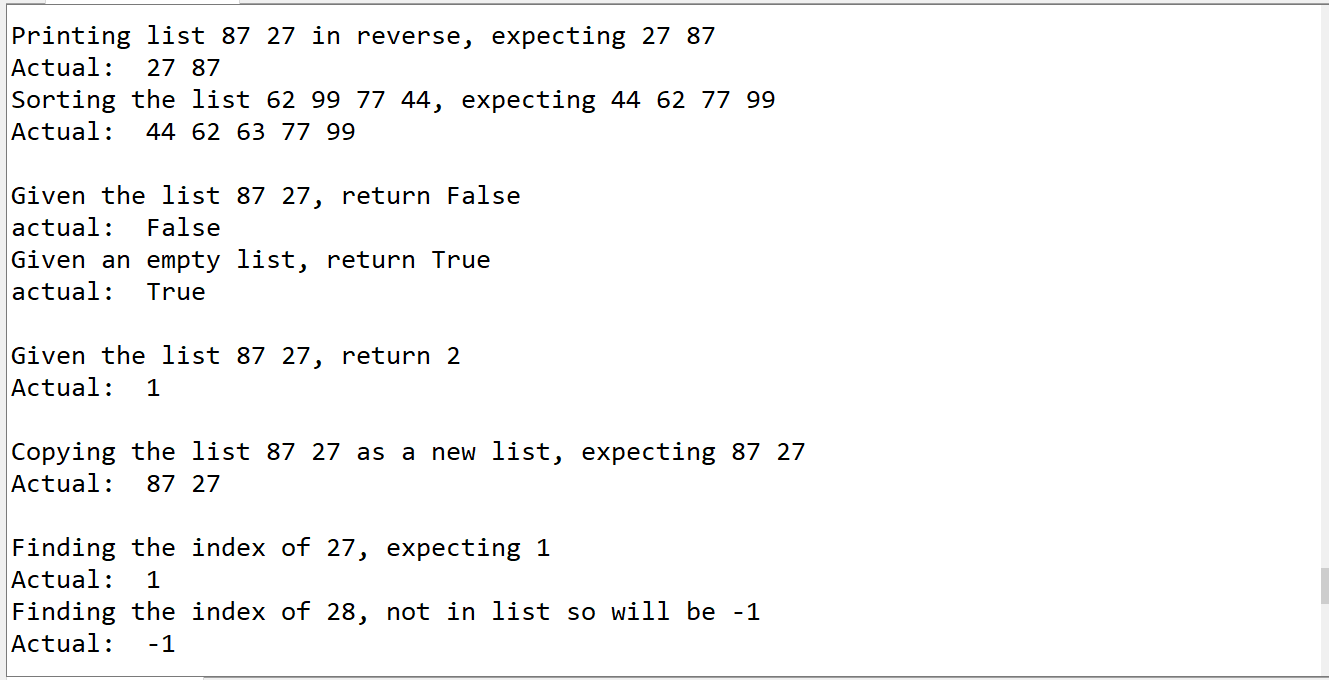
The first main error came from the Remove test case. I had to change a variable name at some point and misspelt one so the method does not compile.



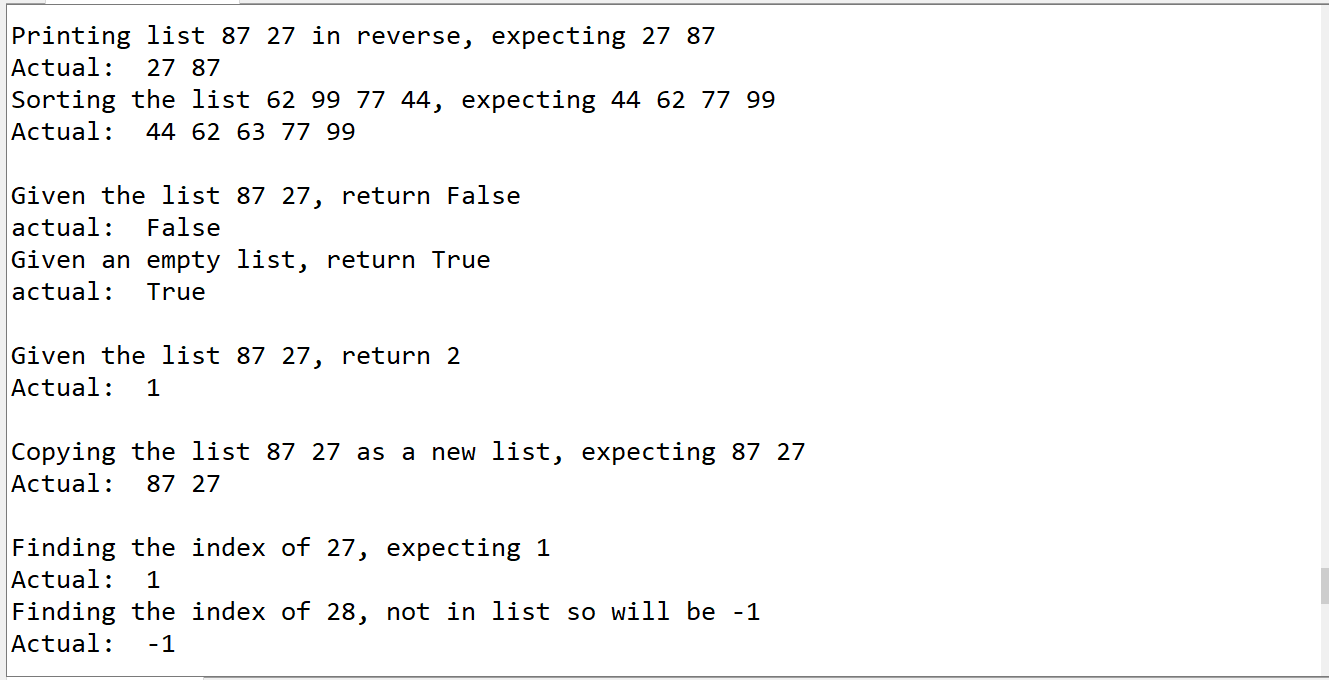
The problem here is on Line 63, where it should read iter1.next = iter1.next.next. The logic in this method works if the line is changed, however since the submitted code has this error this is the way it is represented in this section.

My search method works, however to does make an extra print statement that is a leftover of debugging. 

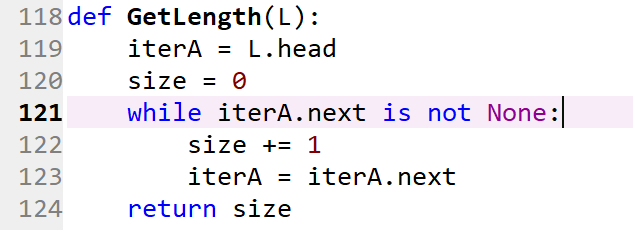
My print list method is used to demonstrate practically every single test case and as such I don’t feel it necessary to do a separate test on that alone.



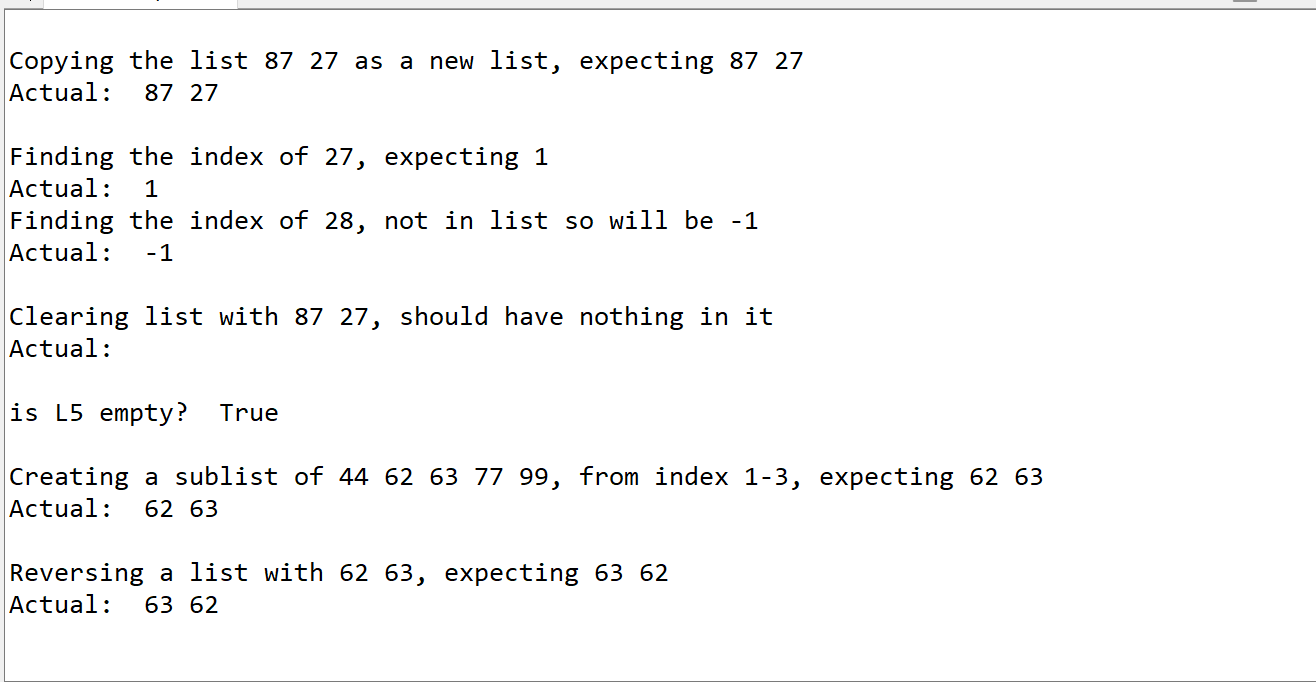
The methods PrintReverse, Sort, and IsEmpty are all functional as shown above.



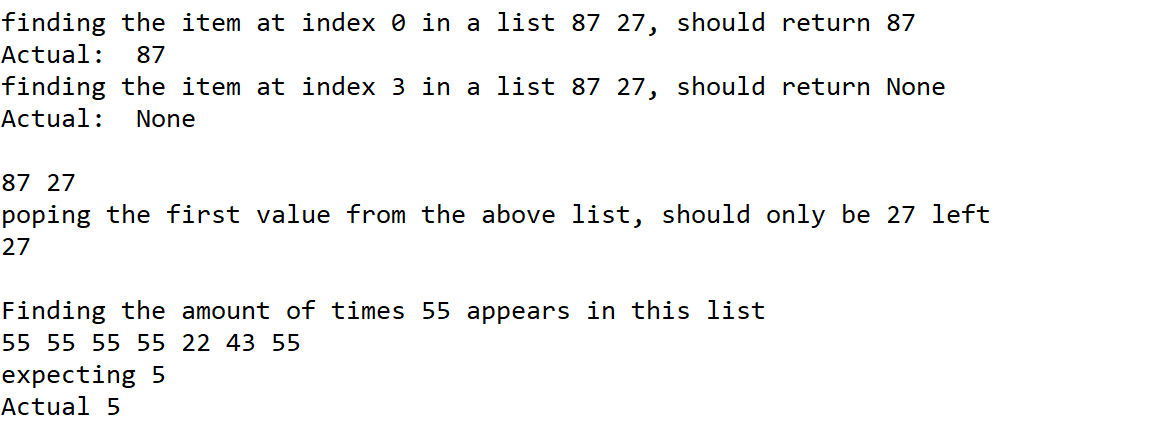
The method GetLength unfortunately has a small error where it will return 1 less than the length of the list. This error could be simply fixed by changing the while condition to be while iterA is not None: instead of the current one on line 121



The next methods all work as intended. Copying a list will return a new list that is the same as the old one, finding the index of a value will return the right value, clearing a list will successfully remove all elements from it, creating a Sublist will work from startIndex to endIndex – 1 and reversing a list will successfully flip all elements in the list.



Also the ItemAt, Pop, and count methods work as intended:



Conclusions

Overall this lab has taught me the importance of testing more than anything else. I did not have enough time during the course of the week to test my lab properly and as a result a good amount of my functions work almost as intended, but as with most industries almost working is not exactly good enough. It has been interesting seeing all the different possible functions that can be added to a linked list, and I’m sure there’s more that can be added but these seem to be the main ones. A lot of these I don’t think I would’ve been able to come up with myself, but after seeing them I can definitely see how they could be useful when using linked lists in a program.

Appendix

**Source code is as follows:**

# -\*- coding: utf-8 -\*-

"""

Course: CS2302 Data Structures

Author: Nathan Aun

Assignment: Lab 2 - Linked Lists

Instructor: Dr. Fuentes

TA: Ismael Villanueva-Miranda

Date of Last Modification: 6/27/2019

Purpose of Program: To implement many common functions of a linked list

"""

#======================================

#DO NOT MODIFY THIS CODE

class Node(object):

def \_\_init\_\_(self, data, next=None):

self.data = data

self.next = next

class List(object):

def \_\_init\_\_(self):

self.head = None

self.tail = None

def Append(L,x):

if L.head is None:

L.head = Node(x)

L.tail = L.head

else:

L.tail.next = Node(x)

L.tail = L.tail.next

#Use the BuildList function to create a reference-based list L

#from a native list pList

def BuildList(pList):

L = List()

for d in pList:

Append(L,d)

return L

#DO NOT MODIFY THIS CODE

#======================================

def Prepend(L, x):

N = Node(x,L.head)

L.head = N

if L.head.next is None:

L.tail = N

def InsertAfter(L, w, x):

foundW = False

iter1 = L.head

if isEmpty(L):

print("List is empty, cannot add", x, "after", w)

else:

while iter1.next is not None and foundW == False:

if iter1.data == w:

iter1.next = Node(x,iter1.next)

foundW = True

else:

iter1 = iter1.next

if foundW == False:

print("Did not find value", w, "in list, therefore cannot add", x,"after it")

def Remove(list1, x):

if isEmpty(list1):

print("list is empty, cannot remove", x)

else:

foundX = False

iter1 = list1.head

while foundX == False and iter1.next is not None:

if iter1.next.data == x:

iter1.next = iter.next.next

foundX = True

iter1 = iter1.next

if foundX == False:

print("Did not find value", x, "in list, therefore cannot remove it")

def Search(list1, x):

if isEmpty(list1):

return None

else:

iter1 = list1.head

while iter1 is not None:

if iter1.data == x:

print("here")

return x

iter1 = iter1.next

def Print(L):

temp = L.head

while temp is not None:

print(temp.data, end = " ")

temp = temp.next

print()

def PrintReverse(list1):

L2 = List()

L2.head = list1.head.next

if L2.head is None:

print(list1.head.data, end = " ")

else:

PrintReverse(L2)

print(list1.head.data, end = " ")

def Sort(L):

length = 1

temp = L.head

while temp is not None:

temp = temp.next

length += 1

length2 = length -1

for a in range(length):

temp = L.head

for b in range(length2 - 1):

if temp.data > temp.next.data:

temp.data, temp.next.data = temp.next.data, temp.data

temp = temp.next

length2 -= 1

def isEmpty(L):

if L.head is None:

return True

return False

#untested

def GetLength(L):

iterA = L.head

size = 0

while iterA.next is not None:

size += 1

iterA = iterA.next

return size

def Copy(L):

if isEmpty(L):

return None

L2 = List()

iterA = L.head

while iterA is not None:

Append(L2, iterA.data)

iterA = iterA.next

return L2

def ItemAt(L, n):

iterA = L.head

for i in range(n):

if iterA is None:

return None

iterA = iterA.next

return iterA.data

def Pop(L, i = 0):

if i == 0:

L.head = L.head.next

else:

iterA = L.head

for k in range(i - 1):

iterA = iterA.next

iterA.next = iterA.next.next

def Count(L, x):

numOccur = 0

iterA = L.head

while iterA is not None:

if iterA.data == x:

numOccur += 1

iterA = iterA.next

return numOccur

def Index(L, x):

iterA = L.head

index = 0

while iterA is not None:

if iterA.data == x:

return index

iterA = iterA.next

index += 1

return -1

def Clear(L):

L.head = None

L.tail = None

def Sublist(L, start = 0, end = -1):

L2 = List()

iterA = L.head

if end == -1:

end = GetLength(L)

i = 0

while iterA is not None and i != start:

iterA = iterA.next

i += 1

while iterA is not None and i != end:

Append(L2, iterA.data)

iterA = iterA.next

i += 1

return L2

def Reverse(L):

iterA = L.head

tempList = []

while iterA is not None:

tempList += [iterA.data]

iterA = iterA.next

iterA = L.head

index = len(tempList) - 1

while iterA is not None:

iterA.data = tempList[index]

iterA = iterA.next

index -= 1

def test():

L = List()

L2 = List()

L.head = Node(99,Node(77))

L.tail = L.head.next

#Appending test cases

print("Appending an existing list with 99->77, with 44, expected 99 77 44")

print("actual: ", end = ' ')

Append(L, 44)

Print(L)

print()

print("Appending an empty list with 44, expected: 44")

print("actual: ", end = ' ' )

Append(L2, 44)

Print(L2)

print()

#prepending test cases

L3 = List()

print("Prepending an existing list with 99->77->44, with 62, expected 62 99 77 44")

print("actual: ", end = ' ')

Prepend(L, 62)

Print(L)

print()

print("Prepending an empty list with 42, expected: 42")

print("actual: ", end = ' ' )

Prepend(L3, 42)

Print(L3)

print()

#insertAfter test cases

print("Inserting 63 after 99 in list 62 99 77 44, expected: 62 99 63 77 44")

print("Actual: ", end = ' ')

InsertAfter(L, 99, 63)

Print(L)

print("Inserting 19 after 33 but 33 is not in the list, expected: ")

print("Actual: ", end = ' ')

InsertAfter(L, 33, 19)

#remove test cases

InsertAfter(L, 77, 27)

InsertAfter(L, 24, 27)

InsertAfter(L2, 25, 38)

Remove(L2, 27)

print(Search(L,77))

print(Search(L,80))

Print(L)

PrintReverse(L)

print()

Sort(L)

Print(L)

L2 = Copy(L)

Print(L2)

print(ItemAt(L2,3))

Append(L2, 55)

Append(L2, 65)

Print(L2)

Pop(L2)

Print(L2)

Pop(L2, 2)

Print(L2)

print(Count(L2, 77))

Append(L2, 77)

Append(L2, 77)

Prepend(L2, 77)

print(Count(L2, 77))

Print(L2)

print(Index(L2, 55))

L3 = Sublist(L2, 1, 5)

Print(L3)

Reverse(L3)

Print(L3)

test()

Statement of Academic Honesty

“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.”

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_