**Dafne Bencomo**

July 1, 2019

CS2302

**Lab #2 Report**

The problem at hand in Lab 2 called to create multiple functions to manipulate a reference-based singly-linked lists. The code defining the reference-based singly-linked list was provided to us in class. We were asked to create a total of eight functions including ItemAt, Copy, Pop, Count, Index, Clear, SubList, and Reverse. Each of these functions had its own set of restricted perimeters and specific return elements. Because of this, the solution implemented varied from function to function.

The first function, ItemAt, ﻿received two parameters, a reference-based linked list 'L' and an integer 'i’ and returned the data in the Node at 'i' in the list. I took a quite simple approach when solving this problem. I decided that iterating through the list until the desired item was reached and then returned was the best choice. However, I had to take into consideration the multiple cases in which there could be an error. The first instance being if we received and empty linked list. If this was the case, the user would receive an error message indicating that the current list is empty and was hence unable to retrieve the desired item as shown below.

A screenshot of a cell phone

Description automatically generated

Another case to consider was an invalid input for ‘i.’ This being whether ‘i’ was less than 0 or greater than the total length of the linked list minus one. To facilitate this I decided to create an additional function GetLength which received a reference-based linked list and returned the total number of elements in the linked list by using a counter. In this case, the user would receive a similar error message as shown below.

A screenshot of a cell phone

Description automatically generated

If there were no errors found in the input the function would simply return the item at the desired index.

A close up of a logo

Description automatically generated

The second function, Copy, received a reference-based singly-linked list ‘L’ and returned an additional reference-based singly-linked list 'copy.' This function followed a similar approach. A new empty reference-based singly-linked list ‘copy’ was created and a for loop with a temp variable was used to traverse through the given linked list ‘L’ and the Append method was used to append each element of the given list onto our new list. Regardless of the linked list received this function would always return the newly created linked list ‘copy.’

A close up of a logo

Description automatically generated

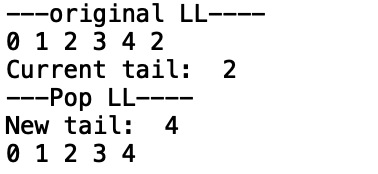
The third function ‘Pop’ was one of the more complex functions to solve in this lab. ‘﻿Pop’ received two parameters, a reference-based linked list 'L' and an optional integer 'ind' and removed or popped the Node at index 'ind.' However, the instructions indicated that the second parameter indicating the index should be an optional parameter. If it was not specified, we were to remove the first Node in the list. The complexity of this function was found in the multiple cases in which we could have an error as well as figuring out a way to specify and work with an optional parameter. After some research I decided that the best approach for an optional parameter in this situation would be for an \* to precede the optional parameter. The way I decided to make sure the optional parameter was present or not was by typecasting it into a string and checking the length. If the length was 0 then the first Node would be removed. Having the first Node removed also meant that the head of the list was going to have to be updated.

We would have the same output if the user’s input was 0. The example below shows the head being removed and updated as the optional parameter was not specified.

A picture containing object

Description automatically generated

In a similar manner, the same thing was to be expected if the user decided to pop the tail. We were to traverse the linked list using a temp variable until we reached the Node before the tail and made that the new tail and set the next element to None.



Another case that was to be taken into consideration was if the linked list given only had one element and that element was chosen to be popped whether it be by the appropriate user input or leaving that parameter empty. In this case, the head and tail were to be equal to None. However, instead of making them both None, I decided to use another one of the functions we were asked to create, Clear. In this case we would return an empty list.

A close up of a device

Description automatically generated

The fourth function Count ﻿received two parameters, a reference-based linked list 'L' and an

integer 'x' and returned an integer 'count' which represents the number of times the number 'x' was found in the list. This function used a for loop with a temp to traverse the linked list and a counter which kept track of the total number of times the chosen number ‘x’ was found in the list. To achieve this, a comparison of the data in each Node was made during the traversal. If the data equaled the chosen number ‘x’ then the counter was incremented by one and then returned.

A close up of a logo

Description automatically generated

The fifth function Index ﻿received two parameters, a reference-based linked list 'L' and an

integer 'x' and returned the index of the first item whose value is equal to ‘x’ in the list. I took a bit of a different approach to solve this problem. With the use of an additional function I created, Search, I was able to check if the input was present in the list at all. If it wasn’t, the user would receive an error message like the one shown below.

A screenshot of a cell phone

Description automatically generated

However, if this was not the case, the function would return the index of the first value equal to ‘x.’

A screenshot of a cell phone

Description automatically generated

The sixth function Clear ﻿receives a reference-based linked list 'L' and removes all elements in the list. This was the simplest function since we were required to basically lose the list. To do this we simply remove the pointer to both the head and tail and return the same list we were given.

A close up of a device

Description automatically generated

The seventh function SubList ﻿receives three parameters, a reference-based linked list 'L',

and two integers 'start' and 'end' and returns a new list 'sub.' This method makes a new list from the given list which only contains the elements indicated by the parameters 'start'(inclusive) and

'end' exclusive without modifying the original list 'L.' The most complicated portion of this function was to compute the logical comparisons that would bring about an error. The possible errors included if the variables start and end were less than 0 or greater than the total length of the list itself. Again, to calculate the length of the linked list I used the additional GetLength function I created. If any of these cases were to happen the user would receive an error message like the one shown below.

A screenshot of a cell phone

Description automatically generated

If none of those cases were to occur then the user would receive a list that contained the desired elements.

A screenshot of a cell phone

Description automatically generated

The eight and last function Reverse ﻿receives a reference-based linked list 'L' and simply reverses the elements in the given list 'L' and returns the same linked list after reversing the elements. This problem was approached in a fairly simple manner. Since we were not allowed to create an additional linked list as instructed this was done by using a temp variable to traverse the linked list and copy the data of said Node onto an array. Then another temp variable was used to traverse the linked list again and at the same time the elements in the array were accessed from the last element to the first. During each traversal the current element in the array would overwrite the current element in the linked list and then return the final linked list.

A close up of a logo

Description automatically generated

I learned a lot from completing this project. The research I had to do in order to learn how to approach certain aspects of this project such as handling an optional parameter allowed me to further expand my knowledge as a whole. I also learned the different situations in which having a recursive function would not be the best idea.

**Appendix**

﻿class Node(object):

# Constructor

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

self.data = data

self.next = next

class List(object):

# Constructor

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

def IsEmpty(L):

return L.head == None

def Print(L):

if IsEmpty(L) == True:

print("Empty List")

else:

temp = L.head

while temp is not None:

print(temp.data, end=' ')

temp = temp.next

print()

def GetLength(L):

count=0

if IsEmpty(L) == True:

return 0

else:

temp = L.head

while temp is not None:

count+=1

temp = temp.next

return count

def Search(L, x):

if IsEmpty(L) == True:

pass

else:

temp = L.head

while temp is not None:

if temp.data == x:

return True

temp = temp.next

return False

﻿def Copy(L):

copy = List()

temp = L.head

while temp is not None: #traverses through each Node from the original L

Append(copy, temp.data) #appends each Node into the new list copy

temp = temp.next

return copy

﻿def ItemAt(L, i):

if L.head is None:

print('Empty List')

if i > (GetLength(L)-1) or i < 0: #catches if a user chose an invalid i

print('LL does not contain', i, 'elements')

return

temp = L.head

for x in range(i):#iterates through list with temp until temp becomes desired Node at i

if temp is not None:

temp=temp.next

return temp.data

﻿def Pop(L, \*ind):

if L.head is None:

print('Empty List')

return

elif len(str(\*ind)) == 0 or int(\*ind)==0: #checks if parameter ind is specified

L.head = L.head.next #removes first element if not

return

elif int(\*ind) > (GetLength(L)-1) or int(\*ind) < 0: #checks if ind is a valid value

print('LL does not contain', \*ind, 'elements')

return

elif GetLength(L) == 1: #if list has only one element it will clear the list

Clear()

temp = L.head

count=0

while count != (int(\*ind)-1): #travels to the node right before one at 'ind'

temp = temp.next

count+=1

if temp.next == L.tail: #if 'ind' is the last element it updates the tail

temp.next = None

L.tail = temp

return

temp.next = temp.next.next

return

﻿def Count(L, x):

if L.head is None:

return 0

else:

temp = L.head

count=0

while temp is not None:

if temp.data == x: #compares wether the current Node data is x

count+=1

temp=temp.next

return count

﻿def Index(L, x):

if L.head is None:

print('Empty List')

if Search(L, x) == False: #makes sure the element is present in the list

print('Element not in List')

else:

temp = L.head

count=0 #keeps track of the index

while temp is not None:

if temp.data == x:

return count

else:

temp = temp.next

count+=1

return count

﻿def Clear(L):

L.head = None

L.tail = None

return L

﻿def Sublist(L, start, end):

sub = List()

if L.head is None: #returns empty tree is L is empty

return sub

elif start < 0 or start >= GetLength(L): #checks for any index out of bounds

print('Check indeces')

return sub

elif end < 0 or end > GetLength(L) or start > end:#checks for index out of bounds

print('Check indeces')

return sub

else:

temp = L.head

startcount=0 #keeps track of indeces

endcount = (end-start) #number of iterations after knowing what Node to start on

while startcount != start:

temp = temp.next

startcount+=1

for i in range(endcount):

Append(sub, temp.data)

temp = temp.next

return sub

﻿def Reverse(L):

temp = L.head

List = []

count = -1 #start from the end of the array back

while temp is not None: #used to copy data into array

List+=[temp.data]

temp = temp.next

t = L.head

while t is not None: #used to overwrite current data with new data from the array

t.data = List[count]

count-=1

t = t.next

return

﻿TestList = List()

for i in range(5):

Append(TestList, i)

Append(TestList, 2)

Append(TestList, 2)

print('---original LL----')

Print(TestList)

print('---copy LL----')

Print(Copy(TestList))

print('---ItemAt LL----')

x = ItemAt(TestList,4)

print(x)

print('---Count LL----')

print(Count(TestList, 2))

print('---Index LL----')

print(Index(TestList, 2))

print('---Sublist LL----')

Print(Sublist(TestList, 2, 5))

print('---List before Pop----')

Print(TestList)

print('---Pop LL----')

Pop(TestList,6)

Print(TestList)

print('---Reverse LL----')

Reverse(TestList)

Print(TestList)

print('---Clear LL----')

Print(Clear(TestList))

**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.”

* Dafne Bencomo