David Amparan

CS 2302

Lab 2

**Introduction**

Lab 2 expanded further the advantages of recursion as well as introducing an ADT (Abstract Data Type). The singly linked list is a type of data type that stores information and links it to one another much like a set of links that forms a chain. There are two types, a singly linked list which can only be traversed in one direction and a doubly linked list which as you guessed is traversed in two directions. Our task was simple, design three algorithms that sort the list that is provide and return the median of that list. A fourth algorithm was requested which did not fully sort an array but instead found its median based on the length of the list and where the middle of that length would be.

**Solution and Implementation**

Moving on, the three sorting algorithms were bubble sort, merge sort, and quick sort. My implementation of bubble sort included the comparison of the items to its neighbors up until we reach the end. If a switch was made, which I kept as a counter variable that would add up whenever a switch was made. If this counter was bigger than zero, then a switched occurred and therefore must check the list once again, until counter is 0 to be specific. A different approach could have been to make the counter a boolean variable which seems more natural but, in my perspective, it makes no change. Next up is merge sort, my algorithm for merge sort did not work. No to say that it did not work at all, but I faced a None Type error no matter what approach I took. My implementation is as follows, the list was split into two separate lists based on its midpoint. So, if a list of 10, 6, 7 was given the lists would be 10 and 6,7. After appending accordingly we compare each item in the list to an item of the other and append onto a new list which will be returned. Whichever item is appended only that list it belongs to moves to the next item.

The third sorting algorithm is called quick sort which in short takes a random number from the list, uses it as a pivot to create a list with numbers greater than or equal to and another list with numbers less than the pivot. Both quick sort and merge sort are recursive methods which required base cases and two recursive calls because of the two lists created in either method. The fourth algorithm needed is a modified quick sort method that does not sort the method but returns the median of the given list. I did not complete this method either, but my approach was as follows. We separate the list with a pivot like we do in regular quick sort but before doing our two recursive calls we compare the length of our list that is less than the pivot to the suspected location of the median in the list given (we pass n, n is the length(list)//2) . If the length of the small list is less than n, n is not located inside the small list so we recursive call MQS on the bigger list and n becomes n minus the length small minus 1. Our second case is if the length of small list is equal or greater than n then n must be within the small list so we call MQS(small list, n). Our third case is finally when neither of the above is true so n must be our pivot, so we return the pivot.

Furthermore, various methods such as getLength(L) which gets the length of a list and median(L) which return the median of a sorted list, were made to aid in the functionality of the program or because it was requested. Our main method simply creates a list of random numbers with a length that is given by the user. Following is a for loop that runs the working methods and displays its results.

**Experiment and Results**

Each sorting method has a different approach with its logic and basis. The way I tested my functioning algorithms was to increase the size of the list and timing how long it would take for it to produce the ordered list. Bubble sort is a very simple algorithm that compares one item to another and switches if they are not in order. This approach requires for a lot of comparisons to be made and thus bubble sort has a time complexity of O(n). As seen on the graph the growth rate of bubble sort steadily increases as the size of the list becomes larger. Quicksort on the other hand separates the list into two separate lists based off its randomly selected pivot. Since we are dividing the list into two separate list, we cut the work done by half and attain a complexity of O (n log n). With this growth rate we can see that it ever slightly increases as the size of n rises. Finally merge sort also has similar logic as it cuts the work in half and attains an O (n log n) time complexity, but since I could not complete these I cannot test it and graph it, this also applies to the modified quicksort method.

**Conclusion**

This project gave me more practice with creating complex recursive methods and exposed me more to the nature of python and Linked List. A lot of work was put into this project mainly due to merge sort and the modified quick sort methods not working how they were intended to. Besides serving its purpose to enhance and teach these sorting methods it also taught me the importance of relentless problem solving which one must undertake.

**Appendix: Source Code**

"""

Author: David Amparan

Course: CS 2302

Instructor: O. Fuentes

TA: Anindita Nath, Maliheh Zargaran

Last Modified: 2/25/2019

Purpose: This code is part of a lab assignemnt in which we must implement three types of sorting methods as well as a modified method which

must return the medium without fully sorting the array.

Down below are my codes as well as some of my codes that do not do what I inteded them to do. Functional methods include bubbleSort and quickSort but the

modified quick sort and mergeSort are not functional.

"""

#creating the class types

import random as rand

import time as tm

class Node( object ):

#constructor

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

self.item = item

self.next = next

class List(object):

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

self.head = None

self.tail = None

self.counter = 0 #keep track of those added for the getLength

#useful methods for each

"""

Method Name: isEmpty | Parameters: List | Return: Boolean

Functionality: Will take in a list and return whether the list is empty (being

empty at head)

"""

def isEmpty(L):

return L.head == None

"""

Method Name: printList | Parameters: L

Functionality: The method will print all the contents of the list if it is

not empty

"""

def printList(L):

temp = L.head #temp to not mess with og List

#loop to print all elements

while temp is not None:

print(temp.item, end=' ')

temp = temp.next

print()

"""

Method Name: appendL | Parameters: L, X (item)

Functionality: will append new nodes to the tail of the item

"""

def appendL(L,X):

if isEmpty(L):

L.head = Node(X)

L.tail = L.head

L.counter +=1

else:

L.tail.next = Node(X)

L.tail = L.tail.next

L.counter +=1

"""

Method Name: getLength | Parameters: L

Funtionality: The method attaisns the value of c (in the L object) and

returns it

"""

def getLength(L):

return int(L.counter)

"""

Method Name: Copy | Parameters: L

Functionality: Creates a copy of the given list and returns it

"""

def Copy(L):

if isEmpty(L):

return L

else:

toReturn = List()

toReturn.head = L.head

toReturn.tail = L.tail

toReturn.counter = L.counter

return toReturn

"""

Method Name: GetELementAt | Parameters: L, index

Functionality: The method will iterate throgh the list and attain the median of the list, this will be done by taking

half of the length and finding the number at that position

"""

def GetElementAt(L, index):

if isEmpty(L):

return L

else:

counter = 0 #counter will help us go through the list

temp = L.head

wanted = 0

while temp is not None:

if counter == index:

wanted = temp.item

temp = temp.next

counter += 1

return wanted

"""

Method Name: Median | Parameters: L

Functionality: The Median will return a list of the medians found which all come from

similar sorted lists which were sorted in a different manner

"""

def Median(L, use):

if isEmpty(L):

return L

else:

C = Copy(L)

#sorting method

if use == 0:#use bubblesort

bubbleSort(C)

#printList(C)

#print(getLength(C))

return GetElementAt(C,getLength(C)//2)

if use == 1: # use quicksort

C = quickSort(C)

#printList(L)

#print(getLength(L))

return GetElementAt(C, getLength(L)//2)

#if use == 2: # merge sort

#L = mergeSort(C)

#printList(L)

#print(getLength(L))

#return GetElementAt(L, getLength(L)//2)

"""

TESTING THE METHODS

theList = List()

print(getLength(theList))

appendL(theList, 45)

printList(theList)

for i in range(5):

appendL(theList, i\*10)

printList(theList)

print(getLength(theList))

"""

"""

Method Name: Bubble sort Parameters: List

Functionality: Bubble sort will compare every single item to its

neighbor and so forth until it reaches the end. A while loop

with a boolean condition will be used until there are no more

switches made

"""

def bubbleSort(L):

isOrder = False

changes = 0 #used to count the changes, if one is made then we have not finished

temp = L.head #dont want to mess with the original

#have to check incase our lst is empty meaning it is ordered

if isEmpty(L):

return

if getLength(L)==1:

return L

while isOrder is not True:

if temp.next is None :##Once we get to the end we want to reset

temp = L.head

changes = 0

if temp.item > temp.next.item:#comparison and increment in changes

tempValue = temp.item

temp.item = temp.next.item

temp.next.item = tempValue

changes = 1

temp = temp.next

if temp.next is None and changes == 0:#here we check incase we go through lsit and make no changes

isOrder = True

"""

Method Name: quickSort | Parameters: L

Functionality: Quick sort is a recursive method that will sort the list

by dividing into two seperate lists with the comparison to a pivot

then called consecutively until only one item exists and returned

Finally it will put these two lists together

"""

def quickSort(L):

if isEmpty(L):

return List()

if getLength(L) <= 1:

return L

if getLength(L) > 1:

smallerTP = List()

biggerTP = List()

pivot = L.head.item

temp = L.head.nex

#compare

while temp is not None:

if pivot > temp.item:

appendL(smallerTP, temp.item)

if pivot <= temp.item:

appendL(biggerTP, temp.item)

temp = temp.next

#recursive calls

smaller = quickSort(smallerTP)

bigger = quickSort(biggerTP)

#special conditions that can result

#if the list less than piuot is zero then we just the pvot in a list and append it

test = List()

if isEmpty(smaller):

appendL(test, pivot)

test.head.next = bigger.head

test.tail = bigger.tail

return test

#if the bigger is empty we simply append the pivot

if isEmpty(bigger):

appendL(smaller,pivot)

return smaller

#similar length so we merge

else:

appendL(smaller, pivot)

smaller.tail.next = bigger.head

smaller.tail = bigger.tail

return smaller

"""

Method Name: mergeSort | Parameters: L

Functionality: Divide into two equal pieces and eventually after successfully dividing the list into

one item which is ordered. After this we put everything back together in order

"""

def mergeSort(L):

#base cases

if getLength(L) == 0:

return L

if getLength(L) <= 1:

return L

#when our list is not just one thing

if getLength(L) > 1:

midpoint = getLength(L)//2

c = 0

LH = List()

RH = List()

temp = L.head

#here we split

while temp is not None:

if c < midpoint:

appendL(LH, temp.item)

#after our midpoint we must insert in Right

else:

appendL(RH, temp.item)

c +=1

temp = temp.next

#printList(LH)

#printList(RH)

Left = mergeSort(LH)

Right = mergeSort(RH)

n = getLength(Left) + getLength(Right)

#here we sort the array

toReturn = List()

while getLength(toReturn) != n:

#if our left side is empty

if isEmpty(Left):

appendL(toReturn, Right.head.item)

Right.head = Right.head.next

#if our right side is empty

if isEmpty(Right):

appendL(toReturn, Left.head.item)

Left.head = Left.head.next

#if neither is none then we must compare and append accordinly

if not isEmpty(Left) and not isEmpty(Right):

if Right.head.item > Left.head.item:

appendL(toReturn, Left.head.item)

Left.head = Left.head.next

if Right.head.item <= Left.head.item:

appendL(toReturn, Right.head.item)

Right.head = Right.head.next

return toReturn

"""

Method Name: modifiedQuickSort | Parameters: L, index

Functionality: much like the original quick sort it will sort the list but will get

rid of one of the extra lists it creates and will return the list where the median is known to exist

"""

def modifiedQuickSort(L, median):

if isEmpty(L):

return List()

if getLength(L) > 0:

smallerTP = List()

biggerTP = List()

pivot = L.head.item

temp = L.head.next

#we assume the rank of the possible median

#compare

while temp is not None:

if pivot > temp.item:

appendL(smallerTP, temp.item)

if pivot <= temp.item:

appendL(biggerTP, temp.item)

temp = temp.next

#determine where the median is based on the length of the lists

if getLength(smallerTP) < median: #n is smaller adn therefore must be within the small list

median = median - getLength(smallerTP)-1

return modifiedQuickSort(biggerTP, median)

if getLength(smallerTP) <= median: #if the len of smallerTP is smaller than n, then n cannot be within the small array

return modifiedQuickSort(smallerTP, median)

if getLength(smallerTP) == median:

return pivot

"""test = List()

n = int(input("How many numbers woud you like? "))

for i in range(n):

appendL(test, rand.randint(0,100))

for i in range(2):

print("This is the Unsorted List")

printList(test)

print("This is the median of that unsorted list")

print(Median(test,i))

""

#Test Cases

test = List()

n = int(input("How many numbers woud you like? "))

for i in range(n):

appendL(test, rand.randint(0,100)

start = tm.time()

quickSort(test)

end = tm.time()

print('Sorted List:', end=' ')

printList(test)

print('Size of the List: ', n)

print('Time taken: ', end-start)

**Academic Honesty Statement**

I, David Amparan certify that this code was written by me with no additional help from online sources or fellow classmates. Brain storming was done with classmates as allowed by the syllabus. If any type of plagiarism or suspicious pieces of code are found I will assume full responsibility but once again this was all written by myself.

David Amparan February 27, 2019