Robert Marc

80487972

CS 2302

MW 1:30-3:00

Lab 2 Report

Introduction

The purpose of this lab was to implement three different methods of sorting a singly linked list and then finding the median of that list. We were also tasked with implementing a modified version of the QuickSort method that should only search one half of the linked list for the median.

Proposed Solutions

The first sorting method we were tasked with implementing was BubbleSort. My solution uses a while loop to continuously traverse the given linked list while sorting the list. It does this by starting with the head of the list and traversing the list until it finds something smaller and placing the head in the place of the smaller item while picking up the smaller item. It proceeds through the list this way until it passes through the list without making any changes, breaking the loop.

The second sorting method was MergeSort. My solution splits the given list as evenly as possible into two smaller lists. It then recursively calls MergeSort on the smaller lists until only one element remains in each list. It then rebuilds the list using MergeLists while sorting the lists into ascending order.

The third method was QuickSort. It takes the head of the given list and sets it as a pivot point. It then sorts the given list into three separate lists, items smaller than the pivot, items equal to the pivot, and items larger than the pivot. It then rebuilds the full list from recursively sorted left and right lists along with all the items equal to and including the pivot.

The modified QuickSort method is similar to QuickSort except that it only sorts items into two lists, those smaller than the pivot and those larger or equal (called largequal in my method) to the pivot, excluding the pivot. I then uses a ranking system to determine the location of the median within the possible sorted list. For this case, it is the length of the given list divided by 2. Depending on the length of the list with smaller elements, it will either recursively call on the largequal list with the rank modified by the length of the smaller list if the length of the smaller list is less than the current rank. If the length of the smaller list is larger than the current rank, it recursively calls on the smaller list with an unadjusted rank. If the length of the smaller list is equal to the rank, it instead returns the pivot item.

Experimental Results

Average Number of Comparisons for a given N across 100 tests

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N = | 10 | 50 | 100 | 250 | 500 | 1000 |
| BubbleSort | 64.89 | 2071.23 | 8784.27 | 57546.39 | 235433.2 | 960398.6 |
| MergeSort | 22.91 | 221.76 | 541.77 | 1676.71 | 3853.09 | 8708.00 |
| QuickSort | 16.04 | 145.19 | 349.08 | 1091.45 | 2478.18 | 5846.03 |
| QuickSortMod | 19.7 | 136.16 | 307.93 | 830.39 | 1661.03 | 3430.63 |

MergeSort expected O(nlogn)

QuickSort expected O(nlogn)

QuickSortMod expected O(n)

Conclusions

In this lab, I struggled with implementing QuickSort and QuickSortMod appropriately. It took some heavy research to understand QuickSort and getting past a mental roadblock to understand the QuickSortMod algorithm given to us by Dr. Fuentes on Slack. However, understanding now how they work, I can understand how QuickSort and QuickSortMod have the lowest number of comparisons out of all the methods. In my research of QuickSort, I also learned that the QuickSortMod method we were asked to implement is very similar to a method of searching an unsorted list or array called QuickSelect.

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.”

Appendix

import random

#Node Functions

class Node(object):

# Constructor

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

self.item = item

self.next = next

def PrintNodes(N):

if N != None:

print(N.item, end=' ')

PrintNodes(N.next)

def PrintNodesReverse(N):

if N != None:

PrintNodesReverse(N.next)

print(N.item, end=' ')

#List Functions

class List(object):

# Constructor

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

self.head = None

self.tail = None

def IsEmpty(L):

return L.head == None

def Append(L,x):

# Inserts x at end of list L

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

else:

L.tail.next = Node(x)

L.tail = L.tail.next

def Print(L):

# Prints list L's items in order using a loop

temp = L.head

while temp is not None:

print(temp.item, end=' ')

temp = temp.next

print() # New line

def PrintRec(L):

# Prints list L's items in order using recursion

PrintNodes(L.head)

print()

def Remove(L,x):

# Removes x from list L

# It does nothing if x is not in L

if L.head==None:

return

if L.head.item == x:

if L.head == L.tail: # x is the only element in list

L.head = None

L.tail = None

else:

L.head = L.head.next

else:

# Find x

temp = L.head

while temp.next != None and temp.next.item !=x:

temp = temp.next

if temp.next != None: # x was found

if temp.next == L.tail: # x is the last node

L.tail = temp

L.tail.next = None

else:

temp.next = temp.next.next

def PrintReverse(L):

# Prints list L's items in reverse order

PrintNodesReverse(L.head)

print()

#Begin student code

def ListFill(n):

#Fills a list of size n with random integers between 0 and n

L = List()

for i in range(n):

Append(L,random.randint(0,n))

return L

def BubbleSort(L):

#Bubble sorts list L, swapping incorrectly ordered elements until sorted

changed = True

global bubbleCounter

while changed:

curNode = L.head

changed = False

while curNode.next != None:

bubbleCounter += 1

if curNode.item > curNode.next.item:

temp = curNode.item

curNode.item = curNode.next.item

curNode.next.item = temp

changed = True

curNode = curNode.next

def Length(L):

#Returns length of the list L

count = 0

temp = L.head

if temp != None:

count += 1

while temp.next != None:

count += 1

temp = temp.next

return count

def Copy(L):

#Copies list L to temp and returns temp

temp = List()

curNode = L.head

while curNode != None:

Append(temp,curNode.item)

curNode = curNode.next

return temp

def GetElementAt(L,n):

#Returns the item of element n in List L

curNode = L.head

pointer = 0

while pointer != n:

curNode = curNode.next

pointer += 1

return curNode.item

def MergeSort(L):

#Separates given list into left and right, until one element remains, then merges

#Checks that there is only one element in list L, base case

if L.head == None or L.head.next == None:

return L

len = Length(L)

left, right = SplitList(L)

left = MergeSort(left)

right = MergeSort(right)

return MergeLists(left, right)

def SplitList(L):

#Splits the given list into left and right

len = Length(L)

tempL = List()

tempR = List()

curNode = L.head

for i in range(len):

if i < len//2:

Append(tempL,curNode.item)

curNode = curNode.next

else:

Append(tempR,curNode.item)

curNode = curNode.next

return tempL, tempR

def MergeLists(L, R):

#Merges the two given lists while sorting them

global mergeCounter

temp = List()

#Makes sure that the temp list is not a None value and the head and tail are intitialzed properly

if L == None and R == None:

temp.head = None

temp.tail = None

elif L == None: #If the Left list is empty, fills temp with the right list

temp = Copy(L)

elif R == None: #If the right list is empty, fills temp with the left list

temp = Copy(R)

else:

curNodeL = L.head

curNodeR = R.head

while curNodeL and curNodeR: #Moves through both lists, sorting values into temp

if curNodeL.item < curNodeR.item:

mergeCounter += 1

Append(temp,curNodeL.item)

curNodeL = curNodeL.next

else:

mergeCounter += 1

Append(temp,curNodeR.item)

curNodeR = curNodeR.next

if curNodeL == None: #If left list is processed before right finishes, finish filling with right

while curNodeR != None:

Append(temp,curNodeR.item)

curNodeR = curNodeR.next

elif curNodeR == None: #If right list is processed before left finishes, finish filling with right

while curNodeL != None:

Append(temp,curNodeL.item)

curNodeL = curNodeL.next

return temp

def QuickSort(L):

#Sorts by splitting list into manageable sections and sorts before recursing quicksort on smaller segment

global quickCounter

#Base case

if Length(L) <= 1:

return L

smaller = List()

equal = List()

larger = List()

pivot = L.head

curNode = L.head

while curNode != None: #Sorts items into three lists, smaller, equal, and larger

if curNode.item < pivot.item:

quickCounter += 1

Append(smaller, curNode.item)

elif curNode.item == pivot.item:

quickCounter += 1

Append(equal, curNode.item)

else:

Append(larger, curNode.item)

curNode = curNode.next

return Concat(Concat(QuickSort(smaller),equal),QuickSort(larger)) #Rebuilds list with smaller pieces while recursively

#calling QuickSort on smaller and larger lists

def Concat(L1,L2):

#Function to build new list from two passed lists

curNodeL1 = L1.head

curNodeL2 = L2.head

temp = List()

while curNodeL1 != None:

Append(temp,curNodeL1.item)

curNodeL1 = curNodeL1.next

while curNodeL2 != None:

Append(temp,curNodeL2.item)

curNodeL2 = curNodeL2.next

return temp

def QuickSortMod(L,rank):

#Modified QuickSort algorithm to find median without fully sorting list

global modCounter

#Base case if algorithm fails and rank becomes less than 0

if Length(L) <= 1:

return L.head.item

smaller = List()

largequal = List()

pivot = L.head

curNode = L.head.next

while curNode != None: #Builds a list of elements smaller than the pivot and a list of elements

if curNode.item < pivot.item: #larger or equal to the pivot, not including the pivot itself

modCounter += 1

Append(smaller, curNode.item)

else:

modCounter += 1

Append(largequal, curNode.item)

curNode = curNode.next

if Length(smaller) < rank: #If List of smaller elements' length is less than rank, recur on largequal list

return QuickSortMod(largequal,rank-Length(smaller)-1)

elif Length(smaller) > rank: #If list of smaller elements' length is greater than rank, recur or smaller list

return QuickSortMod(smaller,rank)

elif Length(smaller) == rank: #If list of smaller elements' length is equal to rank, return the pivot's data point

return pivot.item

def MedianBubble(L):

#Makes a copy of passed list L, sorts using BubbleSort, then finds the median

C = Copy(L)

BubbleSort(C)

median = GetElementAt(C,Length(C)//2)

return median

def MedianMerge(L):

#Makes a copy of passed list L, sorts using MergeSort, then finds the median

C = Copy(L)

C = MergeSort(C)

median = GetElementAt(C,Length(C)//2)

return median

def MedianQuick(L):

#Makes a copy of passed list L, sorts using QuickSort, then finds the median

C = Copy(L)

C = QuickSort(C)

median = GetElementAt(C,Length(C)//2)

return median

#This code has been modified to facilitate testing of the sorting methods. It runs each sorting #method 100 times and calculates the average number of comparisons for each method.

bubbleCounter = 0

mergeCounter = 0

quickCounter = 0

modCounter = 0

for a in range (0,100):

test = ListFill(1000)

MedianBubble(test)

MedianMerge(test)

MedianQuick(test)

QuickSortMod(test,Length(test)//2)

print("BubbleCounter average: ",bubbleCounter/100)

print("MergeCounter average: ",mergeCounter/100)

print("QuickCounter average: ",quickCounter/100)

print("ModCounter average: ",modCounter/100)