Erick Perchez

80582912

CS 2302 Data Structures

Lab 2

**Introduction:** In this lab we were given one file that defined both a linked list called Node and a list called List. We were supposed to create lists of n length and fill them with random numbers, then sort them using three sorting algorithms.

**Proposed solution:**

1. MergeSort- The premise of merge sort was to split the list into two until there is one element left, then compare that to the next list then put them together. This would repeat until the lists were put into one and they were sorted. To split the list I created a method called SplitList that took the length of the list then divided it by two. The first half would be appended to a new list and the second half to a second list. This then would call the method merge. This method took two lists as parameters, then compared the head.item of each list. The smallest element of that comparison would get appended first, then that head would move onto the next element to be compared to the previous bigger element. This was repeated until one list was completely done, then the rest of the elements would be appended to the new list. In every comparison there was a count as well. O(nlogn)
2. QuickSort- The premise of quicksort was to get a pivot point which in my case was the head. Then the list would split into two, the elements smaller than the pivot would go to a list then the others to another list. After this was done, a recursive call would be done and the process would repeat to both lists until the lists were completely sorted. Then the actual pivot would be appended to the left list so it would be at the end of the list that was smaller than the pivot, so it would be in the right place. A counter was also added in every comparison. O(2^n)
3. BubbleSort- The premise of bubble sort was to go element by element starting at the beginning of a list and comparing the previous element with the next one. If the next element was smaller than the previous, they would swap places, if not, then the elements would keep moving forward in the list. This would also contain a counter to count the comparisons. O(n^2)

**Experimental Results:**

I ran the code with a list of n = 10 all the way up to n = 50. This went to show how efficient quicksort was compared to the other two sorting methods. This showed that the bubblesort method was the most inefficient.

**Conclusion:** I learned a lot on this lab. Efficiency really does make a difference when the method is being ran multiple times, especially when the number of times increases fast. This allowed me to see the effects of loops and to try to avoid them for efficiency,

**Appendix:**

'''

Course: CS2302

Author: Erick Perchez

Assignment: Lab 2

Instructor: Dr. Fuentes

TA: Andita Nath

Date: 02/26/2019

Purpose: To sort a list of nodes by ascending order.

It uses BubbleSort, MergeSort, and QuickSort

'''

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()

def GetLength(L):

if L is None:

return 0

temp = L.head

count = 0

while temp is not None:

temp = temp.next

count += 1

return count

def Copy(L):

C = List()

if IsEmpty(L):

return C

else:

temp = L.head

while temp is not None:

Append(C,temp.item)

temp = temp.next

return C

#Gets the middle element of a list of Nodes.

def Median(L):

C = Copy(L)

return ElementAt(C,GetLength(C)//2)

#Used by median, used to return the .item in the node

#given by median

def ElementAt(L,x):

count = 0

while L.head is not None:

if count is not x - 1:

L.head = L.head.next

count +=1

else:

return L.head.item

#Sorting method (least efficient)

def BubbleSort(L):

#Bubble Sort

#O(n^2)

global count

#boolean value used to initiate while loop

#changes to false then back to true if the

#list is sorted

change = True

count = 0

if IsEmpty(L):

return

while change:

temp = L.head

change = False

while temp.next is not None:

count += 1

#compares the previous element with with the next

#if its bigger, they swap places

if temp.item> temp.next.item:

temp2 = temp.item

temp.item = temp.next.item

temp.next.item = temp2

change = True

temp = temp.next

return L

#Sorting method thats the quickest out of the three in this code

def QuickSort(L):

#initializes a global variable to be able to access in the main method

global count

if GetLength(L) > 1:

#Selects head as pivot

pivot = L.head.item

temp = L.head.next

L1 = List()

L2 = List()

count = 0

while temp is not None:

count = count + 1

#splits list by whether the elements are less or more

#than pivot

if temp.item <= pivot:

count = count + 1

Append(L1,temp.item)

else:

Append(L2,temp.item)

count = count + 1

temp = temp.next

#recursive calls to edit the list

L1 = QuickSort(L1)

L2 = QuickSort(L2)

#Adds pivot to the middle

Append(L1, pivot)

#combines lists

return Concatenate(L1,L2)

else:

return L

#Merges two lists together, second list on top of first

def Concatenate(L1,L2):

if IsEmpty(L1):

return L2

if IsEmpty(L2):

return L1

L1.tail.next = L2.head

L1.tail = L2.tail

return L1

#QuickSort but modified to only give one recursion call

def ModifiedQuick(L):

if L.head is not None:

pivot = L.head.item

temp = L.head.next

L1, L2= List(), List()

count = 0

#the median will belong in the longer list

while temp is not None:

count += 1

if temp.item <= pivot:

Append(L1, temp.item)

else:

Append(L2, temp.item)

temp = temp.next

#Sorts the smaller list

if GetLength(L1) > GetLength(L2):

L2 = QuickSort(L1)

return L2

else:

L1 = QuickSort(L2)

return L1

else: return L

#splits the list into two

def SplitList(L):

temp = L.head

L1 = List()

L2 = List()

n = 0

#appends first half then second half of list to separate lists

while n < GetLength(L)//2:

Append(L1,temp.item)

n = n + 1

temp = temp.next

while n < GetLength(L):

Append(L2,temp.item)

n = n + 1

temp = temp.next

return L1, L2

#Third Sorting method, its the average running time

def MergeSort(L):

if L.head is not None and L.head.next is not None:

#unpacks the two lists given by SplitList

L1, L2 = SplitList(L)

#recursively calls the same method but on the two new lists

#eventually reducing the list to one element

L1 = MergeSort(L1)

L2 = MergeSort(L2)

#Sets sort to be the combination of both lists, but now sorted.

sort = Merge(L1,L2)

return sort

else: return L

def Merge(L1, L2):

global count

sort = List()

count = 0

current = L1.head

current2 = L2.head

#compares two elemnts of the lists, whoever is smallest gets appended

#then the following element of the list gets compared with the

#one that did not get appended

while current is not None and current2 is not None:

count += 1

if current.item < current2.item:

Append(sort, current.item)

current = current.next

else:

Append(sort, current2.item)

current2 = current2.next

#Appends any left over elements as the top while loop will only sort the

#lists until one list is completely gone

while current is not None:

Append(sort, current.item)

current = current.next

while current2 is not None:

Append(sort, current2.item)

current2 = current2.next

return sort

#Fills list using a given n value which is the length of the list

#and fills it with random values from 0 to 100

def ListFiller(n):

L = List()

for i in range(n):

Append(L,random.randint(0, 101))

return L

############# M A I N #################

L = ListFiller(50)

print("Unsorted List:", end = ' ')

Print(L)

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

Print(MergeSort(L))

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

print(Median(L))

print('Count: ', count)

print()

print('===================================================')

print("Unsorted List:", end = ' ')

Print(L)

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

a = (QuickSort(L))

Print(a)

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

print(Median(a))

print('Count: ', count)

print()

print('===================================================')

print("Unsorted List:", end = ' ')

Print(L)

print("Modified Quick Sorted List: ", end = ' ')

b = (ModifiedQuick(L))

Print(b)

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

print(Median(b))

print('Count: ', count)

print()

print('==================================================')

print("Unsorted List:", end = ' ')

Print(L)

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

c = (BubbleSort(L))

Print(c)

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

print(Median(c))

print('Count: ', count)

“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”

