Lab2

UTEP CS2302



Javier Soto

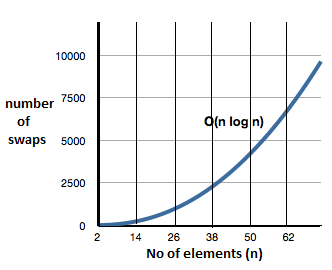
Lab 2 on sorting algorithms was a great exercise to gain hands on experience and visualization of time complexities and the grows of run times with different input sizes. The difficulties that I faced in this lab was the traversal and return of Professor Fuentes implementation of a singly linked list object. Through this lab I was also able to practice objected oriented programming by adding to the linked list class, functions that aided in my completion of this lab.

Starting with Bubble sort, my algorithm was to iterate from the head of the list to the tail. While visiting each node, I compare the item of the current node to the item of the next node. If the current node is larger than the following node then I then I call function “Swap” that will what the items of the node instead of trying to swap the entire node itself. If In the iteration of the entire linked list a swap at any index has occurred, then I set Boolean variable done to False. The traversal of the linked list terminates when the done variable is net set to false. At best case scenario my bubble sort implementation will take O(n) time. If the linked list past is already sorted, then the entire list must be traversed at least once. When the List is in random order my bubble sort will take O(N^2) since the list must be iterated till the list becomes sorted.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| bubble sort | 10 | 100 | 1000 | 10000 | 100000 |
|  |  | 0.001969 | 0.176679 | 19.55102 |  |

Unfortunately for the reminder of the lab I was unable make my code compile before the submission deadline. At the time of writing this report I was able to trace my code and trouble shoot the problems. Pertaining to my merge sort implementation, in the script I submitted I had a infinite loop with my recursive call since I had an incorrect base case. My intent with this sorting algorithms was to create 3 lists one that would temporarily hold the sorted list after it has been split and sorted. The other two lists would be to hold the newly created left and right lists once they have been split equally in two. The splitting of the list I handled by my helper method named “merge splitter” that takes a parameter of two new lists the current item, the size of the original list and a counter.

As mentioned above I have edited my linked lit class for ease of use for this lab, in this case I added an attribute to the List object called size that hold the number of nodes in the list. The “counter” argument is initiated in the original merge sort function that increments every time a new node is visited. If the counter is less than half the length of the original list, meaning the first half of the list. Then the current item is appended to the newly created left list and the latter when count is grater than half the length of the original length.



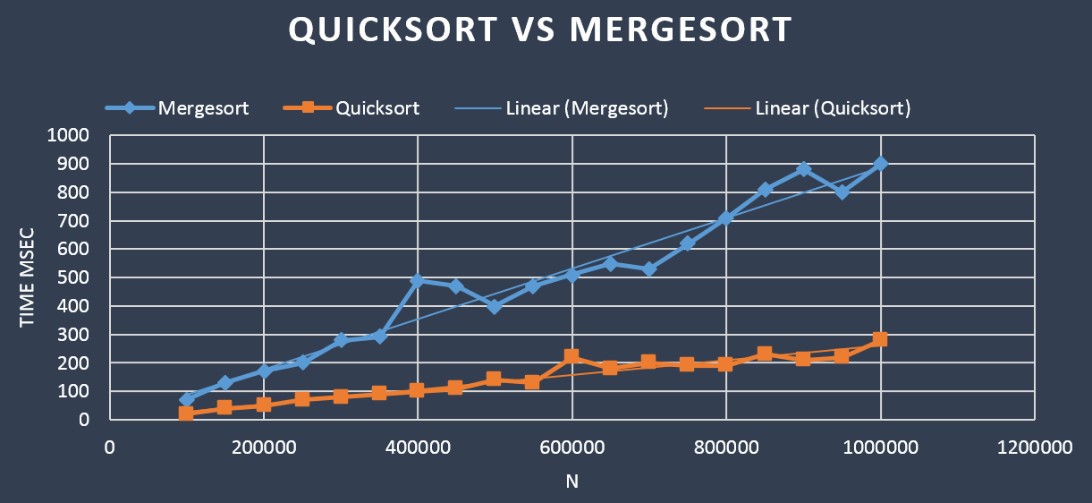
Once the two equally sized lists are created then my merge sort method is recursively called with twice once with each new list. The process is repeated till each list contains only one item. When every list contains only one item then we know every list is sorted now we must merge all lists into one list. I use a second helper method named merge that takes two lists and returns a single list contain all elements of both lists in sorted order. The way I merged the lists was by starting at the head of both left and right lists. I compare the two items and appended the smaller of the two to new list “Sorted list”. The list with the smaller node is then incremented to the next item and the process is repeated till both reach a null node.

Returning to the original merge sort function, all lists are merged from single item lists to a single list containing all nodes from original unsorted list. My merge helper method has a time complexity of O(2^n) my merge splitter has a constant time complexity and my merge sort has a recurrence equation of T(n)=2T(n/2)+2^n. I realize this I a terrible time complexity and that merge sort should have a time complexity of no longer than O(n log n) at worst case scenario. I am not satisfied with my implementation of Merge sort and would not want to provide my potential client with this implementation knowing that it can be done faster.

My implementation of Quick sort goes as follows. When called on with a List argument, if the list is greater than 1 then the pivot is set to the lists head item. The items of the list are starting with head. Next and each item is sent to my helper method. The helper method compares the current item with the pivot to sort items that are larger than the pivot and those who are smaller into two separate lists. With these two new lists, two recursive calls are made to quick sort with either list. Once all lists contain only one element then we concatenate the two lists while including the pivot in the middle of every list. The concatenation of the lists is handled by my concatenation definition that I added to the lists class.

My concatenation function takes two lists while the list to be appended is not empty we iterate every node and append the item of the node to the receiving list. At the end of the iteration the tail pointer of the first list is set to equal the tail list node of the second list. The concatenation function executes in O(n) time since only one iteration of one list is required to do the concatenation and the append function executes in constant time.

The last step in my quick sort method is to set the head of the original list to equal the head of the sorted to reflect the work done to the original list. My quick sort algorithm has a recurrence equation of T(n) = 2T(n/2)+n because of the concatenation process. The issue I had with this function is that I am not receiving all the nodes in the sorted list. After tracing my code I noticed that the nodes I’m missing are the ones that are larger than the pivot of every recursive call. After extensive testing I am confident that my concatenation function is not the function contacting the bug.



To complete the modified quick sort algorithm I needed to implement a function “rank” that returns the location of an item being search. To do so took me constant time with a loop. If the rank we are looking for is 0 we return the item of the head node. Else we increment the list pointer and subtract one from the desired rank till the rank is 0 in which case we will be at the node of that rank. I was unable to test my modified quick sort method since I relied on the same algorithm as my original quick sort method. I did make the required changes to meet the requirements of the lab as follows. The above-mentioned rank definition has been added to the quick sort algorithm, as before the original list is split into lists bigger and smaller than he pivot we then locate where the median of the list will be. if the pivot is at an index less than half length of the original list, meaning the current pivot is less than the median then we know the median will be in the list of numbers larger than the pivot. If the pivot is at an index larger than half the length of the original than the median is in the smaller list. With this information we can eliminate one of the recursive calls and only sort the list containing the median or return the pivot if it is the median.

In conclusion, I have found that my weakness in this lab was my ability to traverse and access data recursively with Professor Fuentes abstract data type. I know with a little more time I can master this implementation and be able to use the data type more effectively and efficiently. Though I was unable to get my script to run correctly, just by attempting the algorithms I was able to see the effect of loops and recursive calls have on time complexities and how in the big picture there can be a huge difference in run times if you choose to implement an algorithm that doesn’t work to well with you data type.

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

"""

Created on Mon Feb 18 19:00:04 2019

@author: yathatjavi

Professor Olac fuentes

TAs:

-Anindita Nath

-Maliheh Zaragaran

IA

-Eduardo Lara

Peer Leader

-Erick Macik

the puspose of this lab is to demonstrate my knowldge of implementing different

sort algorithms with a provided sigly linked list abstract data type as

provided by Professor Olac Fuentes

"""

import lists

from random import randint

from datetime import datetime as dt

def fillList(L):

n = int(input(' What size list would you like to test?'))

for i in range(n):

lists.Append(L,randint(0,100))

def qSortHelper(item,L1,L2,pivot):

#used to create two lists

#L! will hold items smaller than pivot

#L2 contains the latter

if item < pivot:

lists.Append(L1,item)

else:

lists.Append(L2,item)

return L1, L2

def quickSort(L):

#will take a list break that list into two samller with items greater

#or less than the item at index 0

#will then concatinate the two lists after all recursive calls

if L.size >1:

pivot = L.head.item

l1=lists.List()

l2=lists.List()

#for every element in L

temp =L.head.next

while temp is not None:

l1,l2 = qSortHelper(temp.item,l1,l2,pivot)

temp = temp.next

quickSort(l1)

quickSort(l2)

#lists.Append(l1,pivot)

lists.Prepend(l2,pivot)

lists.Concatenate(l1,l2)

L.head = l1.head

def fillTList(L):

lists.Append(L,1)

lists.Append(L,2)

lists.Append(L,3)

lists.Append(L,4)

lists.Append(L,5)

def swap(n1,n2):

#used to swap the items of two nodes withotu swaping the node

temp = n2.item

n2.item=n1.item

n1.item=temp

def bubbleSort(L):

#will take a list and iterate through swapping items if the current item is

#larger than the susseding item

#will return a sorted list of thoes items

if L.head== None:

return

done = False

while done is not True:

done = True

t = L.head

while t.next is not None:

if t.next.item < t.item:#if next item is smaller swap node.items

swap(t.next,t)

done = False

t= t.next

def merge(LL,RL):

#will take two lists and create a single list of order items contatined

#in the two smaller lists

t1 = LL.head

t2 = RL.head

SL =lists.List()

while t1 is not None or t2 is not None:

if t2 == None or t1.item < t2.item:

#if left list item small or right list is empty append

lists.Append(SL,t1.item)

t1=t1.next

else:

lists.Append(SL,t2.item)

t2=t2.next

return SL

def mergSplitter(LL,RL,count,item,size):

# will split the provied list in half with equal nodes in each list if

#given a a list with even amount of nodes

if count < size/2:

lists.Append(LL, item)

else:

lists.Append(RL,item)

def mergeSort(L):

#will take a list and break it down into two smaller lists and iterate the

#smaller lists appedning the smaller item of both sorted lsits to the now

#sorted list

SL = lists.List()

if L.size >1:

counter =0

temp = L.head

#split into 2 smaller lists

ll =lists.List()

rl =lists.List()

while temp is not None:

mergSplitter(ll,rl,counter,temp.item,L.size)

counter =+1

temp = temp.next

mergeSort(ll)

mergeSort(rl)

SL=merge(mergeSort(ll), mergeSort(rl))

#L.head= SL.head

def rank(L,n):

if L is not None and n < L.size:

t = L

if n == 0:

return t.head.item

else:

t.head= t.head.next

#temp = lists.List()

#temp.head=t

return rank(t,n-1)

def rank01(L,n):

if lists.IsEmpty(L):

return None

temp=L.head

while temp is not None and n > 0:

temp=temp.next

n= n-1

return temp.item

def quickSortMotified(L,n):

if L.size>1:

t=L.head

rnk = rank01(L,n)

pivot = L.head.item

smallList = lists.List()

largeList = lists.List()

while t is not None:

#print(t.item)

smallList,largeList = qSortHelper(t.item,smallList,largeList,pivot)

t = t.next

if(pivot == rnk): # rank n is pivot

return pivot

elif(lists.Contains(smallList,rnk)): # rank n in small list

nr = rank01(largeList,largeList.head.item)

return quickSortMotified(largeList,nr)

else: # rank n in large list

nr = rank01(smallList,smallList.head.item)

return quickSortMotified(smallList,nr)

def quickSM(L,n):

if L.size >1:

t=L.head

pivot = L.head.item

smallList = lists.List()

largeList = lists.List()

t = t.next

while t is not None:

smallList,largeList = qSortHelper(t.item,smallList,largeList,pivot)

t = t.next

if (smallList.size+1)==n:

return pivot

elif (smallList.size+1)<n:

return quickSM(largeList,0)

else:

return quickSM(largeList,0)

L= lists.List()

emptyList= lists.List()

testList= lists.List()

fillList(L)

fillTList(testList)

print('Original list: ', end=' ')

#lists.Print(L)

starttime=dt.now()

bubbleSort(L)

endtime=dt.now()-starttime

print('it took ' + str(endtime) + ' to execute')

#quickSort(L)

#mergeSort(L)

#print(quickSM(L,L.size/2))

#lists.Print(L)

'''

lists.Append(L,10)

lists.Append(L,90)

lists.Append(L,1)

lists.Print(L)'''

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

"""

Created on Thu Feb 28 13:49:58 2019

@author: yatha

"""

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

"""

Created on Mon Feb 18 18:37:20 2019

@author: yatha

Please see documentation on Lab2.py script

"""

#Provied by Olac Fuentes

#for CS2302

#edited and completed by yathatjavi

#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

self.size =0

def Contains(L,x):

temp = L.head

while temp is not None:

if temp.item == x:

return True

temp=temp.next

return False

def IsEmpty(L):

return L.head == None

def Concatenate(L1,L2):

t = L2.head

while t is not None:

Append(L1,t.item)

if t.next is None:

L1.tail=t

t=t.next

def Copy(L , theCopy):

temp = L.head

while temp is not None:

Append(theCopy,temp.item)

temp=temp.next

return theCopy

def isSorted(L):

t = L.head

while t.next != None:

if t.item > t.next.item:

return False

t = t.next

return True

def Append(L,x):

# Inserts x at end of list L

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

L.size += 1

else:

L.tail.next = Node(x)

L.tail = L.tail.next

L.size += 1

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

L.size -= 1

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 Prepend(L,x):

L.size += 1

if L.head == None:

L.head = Node(x)

L.tail = L.head

else:

temp = Node(x)

temp.next = L.head

L.head = temp

def GetLengthRec(L):

if L.head == None:

return 0

if L.head.next == None:

return 1

else:

L.head = L.head.next

return 1+GetLength(L)

def GetLength(L):

temp = L.head

size =0

while temp != None:

size+=1

temp = temp.next

return size

def InsertAfter(L, W,x):

L.size += 1

tempNode = Node(x)

if L.head.item == W:

tempNode.next = L.head.next

L.head.next = tempNode

temp = L.head

while temp.next != None and temp.next != W:

temp = temp.next

if temp.next != None:

tempNode.next = temp.next.next

temp.next.next = tempNode

if tempNode.next == None:

L.tail = tempNode

def getSize(L):

return L.size

'''

L = List()

#print(IsEmpty(L))

for i in range(5):

Append(L,i)

Print(L)

#Prepend(L,12)

#print(GetLength(L))

InsertAfter(L,2,15)

Print(L)

Print(L)

PrintRec(L)

PrintReverse(L)

Remove(L,2)

Print(L)

Remove(L,20)

Print(L)

Remove(L,0)

Print(L)

Remove(L,4)

Print(L)

print(L.head.item)

print(L.tail.item)

'''

Academic dishonesty

I, Javier Soto, certify that this script and lab report are of my own unless otherwise documented above.

