Lab 2 report – Andres Silva – 80588336

The task as hand for this assignment was to implement different sorting algorithms to linked list, and to compute the time it took each with different list sizes. The efficiency was then evaluated by returning the median of each sorted list. The different algorithms to be implemented were: QuickSort, Bubblesort, MergeSort, and a variation of QuickSort with a different number of recursive calls.

After understanding how each algorithm works and differs from the others, the task to “translate” the functionality of the algorithms from Python native list to singled linked list was started.

For the two algorithms completed in this assignment, their implementation was quite different. Bubblesort was a non-recursive solution, were the algorithm simply swaps 2 elements after comparing which element is larger. This is done multiple times until the list is sorted. Thetime complexity of this algorithm is O(n^2) and the case with 100,000 elements was not computed due to the inefficiency of the algorithm. References of the linked list are not modified with Bubblesort, only the contents of each node.

QuickSort implementation proved to be more difficult. The first part of the task was to split the given list into 2 lists based on a pivot. The lists were then smaller or greater than the pivot and were split once again recursively. This was done by 2 different methods that returned a list with the smaller (or larger) than pivot elements. After the recursive calls, it was time to ‘glue’ the lists and the pivots by making the original list the composition of the sorted lists. To do that, the original list head was now the head of the first half plus the pivot as the next element of the tail, and finally the tail would be set to that of the second half. The time complexity of Quicksort is Ꙩ(nLog n).

The experiments performed consisted of giving the algorithms lists of sizes 10, 100, 1000, and 100,000 (the last value only was passed to quicksort because bubble sort took too long to compute)

The following running times were recorded:

What I learned from the assignment is that different algorithms pose advantages and disadvantages. For example, Bubblesort can tell right away if a list is sorted, and will finish executing faster than Quicksort, yet this rarely happens in real applications. Another example is that Quicksort might not be efficient if the pivot chosen happens to be the smallest (or largest) element in the list, but then again this rarely happens. For most of these algorithms, recursion is more intuitive and easier to implement than doing it iteratively.

**Appendix**:

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

"""

Created on Mon Feb 18 13:15:17 2019

CS 2302 - Andres Silva

> Teacher: Olac Fuentes

> TAs: Anindita Nath & Maliheh Zargaran

> Lab #2

> The purpose of this lab is to implement different sorting algorithms and compare their efficiency based on the size

of the provided list to be sorted.

> LAST MODIFIED: FEB 27th, 2019

"""

import random

import time

from LinkedListModelCode import \*

counter = 0 #Counts number of comparasions.

def CreateLL(n):

LL = List()

for i in range(n):

LL.Append(random.randint(0,1000))

return LL

def MedianBB(L):

C = Copy(L)

BubbleSort(C)

return ElementAt(C, C.getLength()//2)

def MedianQS(L):

C = Copy(L)

QuickSort(C)

return ElementAt(C, C.getLength()//2 )

def Copy(L):

tempHead = L.head

C = List()

while L.head != None:

C.Append(L.head.item)

L.head = L.head.next

L.head = tempHead

return C

def ElementAt(L,n): #median is middle element when list is odd.

for i in range(n):

L.head = L.head.next

return L.head.item

def SplitLeft(L,p): #Returns List with items smaller than pivot

OgNode = L.head #Store original node to keep L intact

L1 = List() #Empty list to be filled

while L.head != None: #Append all items smaller than pivot

if L.head.item < p:

L1.Append(L.head.item)

L.head = L.head.next

L.head = OgNode #Repoint L's head

return L1 #Return L1, filled with items smaller than p

def SplitRight(L,p): #Returns List with items greater than pivot

OgNode = L.head #Store original node to keep L intact

L2 = List() #Empty list to be filled

while L.head != None: #Append all items greater than pivot

if L.head.item > p:

L2.Append(L.head.item)

L.head = L.head.next

L.head = OgNode#Repoint L's head

return L2 #Return L2, filled with items greater than p

def BubbleSort(L):

global counter

counter = 0

Status = False

while Status is not True: #Status variable that will check if list is sorted.

Status = True

temp = L.head

while temp.next is not None: #Go through all the list until the end is reached.

if temp.item > temp.next.item: #If the current item is larger thant he next:

counter += 1

# print(counter)

temp2 = temp.item #Swap items

temp.item = temp.next.item

temp.next.item = temp2

Status = False #Set status to false in order to tell code that list is not sorted yet.

temp = temp.next #Advance List.

def QuickSort(L):

global counter

temp = L #Work with temp to not modify L.

if L.getLength() > 1: #If current list has more than 1 element, it might not be sorted

tempNode = L.head #Store Head of Original List to repoint later and keep intact L.

while temp.head.next != None: #Take the last item as pivot

temp.head = temp.head.next

pivot = temp.head.item #Pivot is the last item of list

# DEBUG: print("\nPivot is: ", pivot, "\n")

temp.head = tempNode #repoint head to first node

L1 = SplitLeft(L,pivot) #Store all items smaller than pivot into L1

L2 = SplitRight(L,pivot) #Store all items greater than pivot into L2

QuickSort(L1) #Sort smaller lists.

QuickSort(L2)

#Glue list

if L.head == None:#If head is empty

counter += 1

L1.Append(pivot)#Insert pivot at end of List 1

else:

counter += 1

L2.Prepend(pivot) #If the head is not empty, insert it at the begining of List 2

if L1.head == None: #If L1 is empty,

counter += 1

L.head = L2.head #Make head of Original list the head of L2

L.tail = L2.tail #Point the tail to last element of L2

else:

L1.tail.next = L2.head #If L1 is not empty

L.head = L1.head #The head of L1 becomes the head of the original list

L.tail = L2.tail#The tail of the original list becomes the tail

else: #List has 1 element, it is sorted, return.

return

def MergeSort(L):

tempHead = L.head #save the node of the head

if L.getLength() > 1 :

# print("if: ")

L1 = List()

L2 = List()

for i in range(L.getLength()//2):

L1.Append(L.head.item)

L.head = L.head.next

while L.head != None:

L2.Append(L.head.item)

L.head = L.head.next

L.head = tempHead #Restore L's head.

print("L1:")

L1.Print()

print("L2:")

L2.Print()

tempL = List()

SmallestList = L1

LargestList = List()

if L1.getLength() > L2.getLength():

SmallestList = L2

LargestList = L1

else:

SmallestList = L1

LargestList = L2

MergeSort(L1)

MergeSort(L2)

# print("smallest List length: ",SmallestList.getLength())

# for i in range(SmallestList.getLength()):

# if L1.head.item < L2.head.item:

# print(L1.head.item, " < ", L2.head.item)

# tempL.Append(L1.head.item)

# L1.head = L.head.next

#

# print("temp inside loop")

# tempL.Print()

#

# else:

# tempL.Append(L2.head.item)

# L2.head = L2.head.next

#

# while LargestList.head != None:

# tempL.Append(LargestList.head.item)

# LargestList.head = LargestList.head.next

print("temp")

tempL.Print()

# L.head = tempL.head

else:

return

#Create a list of size n

A = CreateLL(100)

#A.Print()

start = time.time()

x = MedianBB(A)

end = time.time()

print("\nBubbleSort Total time: ", start - end )

print("Median: ", x)

print("Passes: ", counter)

counter = 0

start = time.time()

print("Median: ", MedianQS(A))

end = time.time()

print("Passes: ", counter)

print("\nQuickSort Total time: ", start - end )

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

"""

Created on Thu Feb 21 19:19:10 2019

CS 2302 - Andres Silva

> Teacher: Olac Fuentes

> TAs: Anindita Nath & Maliheh Zargaran

> Lab #2 - Linked list Class

> LAST MODIFIED: FEB 27th, 2019

"""

#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 L.head == None:

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.head == None:

#print("Empty head")

return 0

else:

tempHead = L.head

counter = 0

while L.head != None:

L.head = L.head.next

counter += 1

L.head = tempHead

return counter

def Prepend(L, x):#Insert data at begining of list

if L.head == None:

L.head = Node(x)

L.tail = L.head

else:

new\_node = Node(x)

new\_node.next = L.head

L.head = new\_node