Introduction:

The task of this lab is to implement several sorting algorithms for finding the median of a linked list. Additionally, to understand the different time complexities of each algorithm and how they work.

Proposed solution design and implementation:

Had to refer to some visual representation of these algorithms to fully understand what the algorithm could really do. Essentially each algorithm’s image and algorithm steps were reverse engineered to design the sorting method for linked lists.

**Bubble Sort**

Design:

Came up with a simple design for comparing the current node with the next, and whichever is a greater integer will remain towards the right of the list. I later found out that the method shouldn’t be recursive since having loops seemed fine for this type of situation. Therefore, having 2 ‘while’ loops or 2‘for’ loops was evident to check the whole list several times as long as the current node is greater than the next node (not in ascending order), and the list is not empty after traversing through a set of nodes.

Implementation:

Had pseudocode ready based on the algorithm designed before. However, after pseudocode, I Created 3 if statements, if L is empty it returns none, if L has only one node, and if L is NOT empty it does the rest of the algorithm.

**Merge Sort**

Design:

Came up with 3 different methods which each one does a distinct job of splitting or merging lists. After analyzing how to split the list into two parts every iteration, using recursion was the best option to consider. After looking splitting the list into two lists (left and right), they are stored and used to merge into a final list. Merge method does the logic behind the comparing of the numbers such as knowing when to switch values

Implementation:

Before programing in the IDE, pseudocode was written. But after that, the split method gets the node pointer of the center which helps separate the list into the two sections for the merge sort recursion. Merge method only compares the numbers from one list to the other recursively and obtains the correct ascending order from both lists and returns the result. This is done recursively until all the lists have been combined.

Unfinished algorithms:

Quicksort: Came up with a design of having a pointer select only the beginning of the linked list. Once the pointer would select the beginning of the linked list, it compares with the rest of the sorting linked list.

Experimental results:

**Bubble Sort**

Experiment: Checked if List is none, then set a temporary list equal to the head of the list and printed the head’s item.

Result: Returned random number from the head node

Experiment: Created two additional if statements testing for an empty list and if list has only one node.

Result: Returned None for empty list and the node item if there’s only one node in the

Experiment: Add two while loops where it keeps going through the list until temporary is not None or if current node’s number is greater than the next

Result: None

Experiment: Added if statement inside of the inner while loop to check if the current node’s value is greater than the next. And if true, I will swap the current node’s value with the next.

Result: Completes sorting algorithm. Tested a variable number of lists and they got sorted well.

**Merge Sort**

Experiment: Created Split list method to return the middle of the list.

Result: Returns reference of the middle node from the list

Experiment: I understood that after splitting the list into two parts, another method had to be made for merging or sorting both new lists recursively.

Result: None, it wasn’t implemented to Merge Sort method yet.

Experiment: Added Merge Sort method, had a node point to the middle of the list and split it as the right half of the list, and the left part just starting from the beginning of the list.

Result: Split all nodes into each recursive call (left or right)

Experiment: Set sorted list equal to Merge method to store the sorted list made so far and return

Result: Completes sorting algorithm. Tested a variable number of lists and they got sorted well.

Running times:

**Bubble Sort:**

List length = 100, Time=0.00298s

List length = 250, Time=0.01192s

List length = 500, Time=0.05186s

List length = 1000, Time=0.20142s

*Big O Time complexity=O(n^2)*

**Merge Sort:**

List length = 100, Time = 0.0s

List length = 250, Time= 0.001995s

List length = 500, Time = 0.001994s

List length = 1000, Time = 0.004987s

*Big O Time complexity = O(nlogn)*

Appendix

*# Course: 2302-001*

*# Author: Esteban Retana*

*# Assignment: Practice sorting algorithms*

*# Instructor: Olac Fuentes*

*# TA: Anindita Nath*

*# Date of last modification:2/23/19*

*# Purpose: Sort linked lists with different sorting algorithms and figure out their time complexity*

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()

*# Create a clone of the linked list*

def Copy(L):

temp = L.head

new\_copy = List()

while temp != None:

Append(new\_copy, temp.item)

temp = temp.next

return new\_copy

*# Obtain item by traversing through the list*

def ElementAt(L,x):

temp = L.head

for i in range(x):

temp = temp.next

return temp.item

*# Get the size of the Linked list*

def GetLength(L):

temp = L.head

i = 0

while temp != None:

i += 1

temp = temp.next

return i

def Median(L):

C = Copy(L)

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

*# Create random list of integers*

def random\_list():

L = List()

for x in range(5):

n = random.randint(0,46)

Append(L, n)

return L

*# Bubble sort algorithm*

def Bubble\_sort(L):

if IsEmpty(L):

return

if L.head != None and L.head.next == None :

return L.head.item

if L != None:

temp = L.head

is\_sorting = True

while is\_sorting:

temp = L.head

is\_sorting = False

while temp.next != None:

if temp.item > temp.next.item:

is\_sorting = True

int\_temp = temp.item

temp.item = temp.next.item

temp.next.item = int\_temp

temp = temp.next

return Median(L)

def Split\_list(L):

if L == None:

return L

after = L.next

before = L

while after != None:

after = after.next

if after != None:

before = before.next

after=after.next

return before

def Merge(L,K):

combined = None

*# If a list is shorter than the other*

if L == None:

return K

if K == None:

return L

*# L is less than, therefore L is inserted first*

if L.item <= K.item:

combined = L

combined.next = Merge(L.next, K)

*# K is less than, therefore K is inserted first*

else:

combined = K

combined.next = Merge(L, K.next)

return combined

def Merge\_sort(L):

*# If list is empty or only one element*

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

return L

if L != None:

mid = Split\_list(L)

r = mid.next

mid.next = None

left = Merge\_sort(L)

right = Merge\_sort(r)

*# j = Merge\_sort(right)*

sorted\_list = Merge(left,right)

*# final\_list = List()*

*# final\_list.head = sorted\_list*

return sorted\_list

C = random\_list()

*# Print(C)*

L = Copy(C)

*# Print(L)*

print(Bubble\_sort(L))

*# Print(L)*

L = Copy(C)

*# Print(L)*

k = List()

k.head = Merge\_sort(L.head)

print(Median(k))