Lab 2: Finding the Median of a Linked List

Manuel Ruvalcaba

CS2302 1:30-2:50

1. Introduction

There are many different ways to sort lists in Python, with the various types of sorting algorithms and the different types of lists. In this lab, we are trying to sort a linked list, which is a user-defined list, using Bubble Sort, Merge Sort, and Quick Sort. Then we will return the median of the list. Then we will implement a modified version of the Quick Sort that only sorts the side where the median resides and then return the median.

2. Proposed Solution Design and Implementation

To solve the problem regarding the Bubble Sort, we need a single method called *bubbleSort* that accepts a *list* as a parameter. We are going to need a *while* loop that will only be executed if there has been a change in the *list*. This can be done by setting a *change* variable equal to *True*, so we can enter the loop in the first place. Inside the loop, we will need a temporary variable that will point to the *head* of the list and set the *change* variable to *False*. We will also need a nested *while* loop that will only be executed if the current node's next node is not *None.* Inside this loop, the current node will be compared to the next node. If the current node's item is greater than the next node's item, the items will be swapped and *change* will be set to *True*, otherwise nothing will happen. Then we will set the current node to the next node, continuing the while loop, doing the same thing until the current node's next node is None. Once it equals *None*, the while loop will be exited and the outer loop will check if *change* is *True*, if it is, it will enter the loop again, repeating the process. Otherwise, it will exit the while loop and the method. The list is sorted.

When it comes to the Merge Sort, we will need two methods, one called *MergeSort* that accepts a *list* as a parameter and another called *Merge* that accepts the *left* and *right* lists as the parameters. The *MergeSort* method will have a base case that checks if the list is empty or has only one element, which will return the *list.* Otherwise, it will find the middle of the list and split the list into two lists with half the elements in each. Then we will call the method recursively with the left and right list. This method will then return the result of the *Merge* method that merges the left and right list. The *Merge* method will have two temporary variables that point to the *head* of each *list* and a new *list L*. Then we will need a *while* loop that is executed only if the temporary node is not *None*. Inside the loop, we will compare the items of the temporary nodes and append the item that is smaller to *L*. The temporary nodes will then point to the next node and continue the *while* loop. Upon exiting, the remaining items will be appended, if one of the temporary variables happened to point to *None.* This method will return the list *L* and the list is sorted.

For the Quick Sort, we will need a single method called *QuickSort* that accepts a *list* as a parameter. The base cases will check if the list is empty or contains only one element, which will return the *list*. Otherwise, we will need a pivot, which is the *head* node's item, two lists, that will contain elements less than or greater than the pivot, and a temporary variable that points to the node next to the head. We will then use a *while* loop that will be executed if the temporary node is not *None*. Inside the loop, the temporary node's item will be compared to the pivot. If the item has a value less than the pivot, it will be appended to the *small* list. Otherwise, it will be appended to the *large* list. The method will be called recursively for the *small* and *large* list, and then the pivot will be appended to the small list, where

it belongs. This method will then return the concatenation of both lists. The list is sorted.

For the modified version of Quick Sort, we will need a single method called *ModifiedQuick* that accepts a *list* as a parameter. The base cases will check if the list is empty or contains only one element, which will return the *list*. Otherwise, we will need a pivot, which is the *head* node's item, two lists, that will contain elements less than or greater than the pivot, and a temporary variable that points to the node next to the head. We will then use a *while* loop that will be executed if the temporary node is not *None*. Inside the loop, the temporary node's item will be compared to the pivot. If the item has a value less than the pivot, it will be appended to the *small* list. Otherwise, it will be appended to the *large* list. After this, we will recursively call the method with the *list* that contains more elements and then the pivot will be appended to the small list, where it belongs. This method will then return the concatenation of both lists. Note that while the list may not be sorted, the element in the middle will be the median.

I then implemented four methods, one for each sorting method, which accept a *list* as a parameter. These methods find and return the median, being the item in the node that is at the middle of the list.

I also implemented a method called *MakeList* that accepts an integer *n* as a parameter. This method creates and returns a list of size *n* with random numbers between zero and twice the value of *n*.

3. Experimental Results

To test the methods, I used the *MakeList* method to generate lists of size 20, 50, 100, and 200. I then used the sorting algorithms to find the median of the list and counters to find the number of comparisons made in each sorting algorithm (included in the code in the appendix). The Bubble Sort algorithm has a time complexity of O(n^2). Merge Sort has a time complexity of O(nlogn). While Quick Sort has an average time complexity of O(nlogn), the worst case can result in a O(n^2). The modified Quick Sort also seemed to have a O(n^2), with an average time complexity of O(n).

	List Size	Median	Number of Comparisons	Matches Big-O?
Bubble Sort	20	28	323	Yes
Merge Sort	20	28	64	Yes
Quick Sort	20	28	76	Yes
Modified Quick	20	28	60	Yes

	List Size	Median	Number of Comparisons	Matches Big-O?
Bubble Sort	50	69	2352	Yes
Merge Sort	50	69	221	Yes
Quick Sort	50	69	234	Yes
Modified Quick	50	69	124	Yes

	List Size	Median	Number of Comparisons	Matches Big-O?
Bubble Sort	100	105	7920	Yes
Merge Sort	100	105	540	Yes
Quick Sort	100	105	660	Yes
Modified Quick	100	105	394	Yes

	List Size	Median	Number of Comparisons	Matches Big-O?
Bubble Sort	200	203	34288	Yes
Merge Sort	200	203	1277	Yes
Quick Sort	200	203	1446	Yes
Modified Quick	200	203	669	Yes

	List Size	Median	Number of Comparisons	Matches Big-O?
Bubble Sort	400	419	152817	Yes
Merge Sort	400	419	2946	Yes
Quick Sort	400	419	3305	Yes
Modified Quick	400	419	1153	Yes

4. Conclusion

With this project, I have learned to use sorting algorithms to sort linked lists and find the median of the lists. I was also able to find the number of comparisons when sorting the lists and find out if it matches the Big-O of its respective algorithm.

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.

5. Appendix

```
"""
Course: CS2302 MW 1:30-2:50
Author: Manuel A. Ruvalcaba
Assignment: Lab #2
Instructor: Dr. Olac Fuentes
T.A.: Anindita Nath, Maliheh Zargaran
Date of Last Modification: February 22, 2019
Purpose of Program: The purpose of this program is to sort a list of n size and
                    return the median of the list.
"""
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 Prepend(L,x):
    if IsEmpty(L):
        L.head = Node(x)
        L.tail = L.head
    else:
        L.head = Node(x,L.head)

def Search(L,x):
    #searches for an element
    temp = L.head
    while temp is not None:
        if temp.item == x:
            return temp
        else:
            temp = temp.next
    return None

def GetLength(L):
    #returns the length of a list
```

```
    if IsEmpty(L):
        return 0
    temp = L.head
    global count
    count = 0
    while temp is not None:
        count += 1
        temp = temp.next
    return count

def InsertAfter(L,w,x):
    #inserts a node after the specified node
    if IsEmpty(L):
        L.head = Node(x)
        L.tail = L.head
    else:
        temp = L.head
        while temp is not None:
            if temp.item == w:
                temp.next = Node(x,temp.next)
            temp = temp.next
def Concatenate(L1,L2):
    #concatenates 2 lists
    if IsEmpty(L1):
        return L2
    elif IsEmpty(L2):
        return L1
    L1.tail.next = L2.head
    L1.tail = L2.tail
    return L1

def isSorted(L):
    #checks if a list is sorted and returns true or false
```

```
    if L.head == None:
        return True
    temp = L.head
    while temp.next is not None:
        if temp.item > temp.next.item:
            return False
        else:
            temp = temp.next
    return True

def Copy(L):
    #makes and returns a copy of a list
    temp = L.head
    Lt = List()
    while temp is not None:
        Append(Lt,temp.item)
        temp = temp.next
    return Lt

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 ElementAt(L,x):
    #finds and returns the element at a certain position
    if IsEmpty(L):
        return None
    elif x > GetLength(L):
        return None
```

```
    temp = L.head
    while x>0:
        temp = temp.next
        x-=1
    return temp.item

def Median(L):
    #finds the median of the list after it is sorted with Bubble Sort
    C = Copy(L)
    bubbleSort(C)
    return ElementAt(C,(GetLength(C)//2))

def Median2(L):
    #finds the median of the list after it is sorted with Merge Sort
    C = Copy(L)
    d = MergeSort(C)
    return ElementAt(d,(GetLength(d)//2))

def Median3(L):
    #finds the median of the list after it is sorted with Quick Sort
    C = Copy(L)
    d = QuickSort(C)
    return ElementAt(d,(GetLength(d)//2))

def Median4(L):
    #finds the median of the list after it is sorted with the Modified Quick Sort
    C = Copy(L)
    d = ModifiedQuick(C)
    return ElementAt(d,(GetLength(d)//2))

def bubbleSort(L):
    #sorts the list using bubble sort
    global count1
```

```
    count1 = 0
    change = True
    while change: #checks if there was a change
        t = L.head
        change = False
        while t.next is not None:
            count1 +=1
            if t.item > t.next.item:
                temp = t.item
                t.item = t.next.item
                t.next.item = temp
                change = True
            t = t.next
    print(count1)
def MergeSort(L):
    #sorts using merge sort
    if L.head is None or L.head.next is None:
        return L
    mid = (GetLength(L)//2)
    temp = L.head
    while mid>1: #finds the middle node
        temp = temp.next
        mid-=1
    #creates the left and right lists
    left = List()
    right = List()
    left.head = L.head
    right.head = temp.next
    temp.next = None
    #recursion
    leftlist = MergeSort(left)
    rightlist = MergeSort(right)
    return Merge(leftlist,rightlist)
```

```
x = 0
def Merge(left,right):
    #merges the lists
    global x
    L = List()
    temp = left.head
    temp2 = right.head
    while temp is not None and temp2 is not None:
        x +=1
        if temp.item < temp2.item:
            Append(L,temp.item)
            temp = temp.next
        else:
            Append(L,temp2.item)
            temp2 = temp2.next

    while temp2 is not None and temp is None:
        Append(L,temp2.item)
        temp2 = temp2.next
    while temp is not None and temp2 is None:
        Append(L,temp.item)
        temp = temp.next
    return L
p = 0
def QuickSort(L):
    global p
    #sorts using quick sort
    if IsEmpty(L):
        return L
    if L.head.next is None:
        return L
    pivot = L.head.item
```

```
    small = List()
    large = List()
    temp = L.head.next
    while temp is not None: #moves elements to either side depending on value
        p +=1
        if temp.item<pivot:
            Append(small,temp.item)
            temp = temp.next
        else:
            Append(large,temp.item)
            temp = temp.next

    small = QuickSort(small)
    large = QuickSort(large)
    Append(small,pivot)
    return Concatenate(small,large)
j = 0
def ModifiedQuick(L):
    global j
    #returns a list that has the median in the middle
    if IsEmpty(L):
        return L
    if L.head.next is None:
        return L
    pivot = L.head.item
    small = List()
    large = List()
    temp = L.head.next
    while temp is not None:
        j+=1
        if temp.item<pivot:
            Append(small,temp.item)
            temp = temp.next
```

```
        else:
            Append(large,temp.item)
            temp = temp.next
    lenS = GetLength(small)
    lenL = GetLength(large)
    if lenS >lenL:
        small = ModifiedQuick(small)
    elif lenS<lenL:
        large = ModifiedQuick(large)
    else:
        Append(small,pivot)
        return Concatenate(small,large)
    Append(small,pivot)
    return Concatenate(small,large)


def MakeList(n): #makes a list of n size
    L = List()
    i = n
    while i>0:
        Append(L,random.randint(0,n*2))
        i-=1
    return L

#this is where the testing of the code happens
l1 = MakeList(400)
#Print(l1)
c = Copy(l1)
l2 = Copy(l1)
l3 = Copy(l1)
print(Median(c))
print(Median2(l3))
print(Median3(l1))
```

```
print(Median4(l2))


print(x)
print(p)
print(j)
```