

Ivan Perez
80602918
Lab 2 Report

Introduction

This assignment has us finish some methods to the object type List. These methods are to create a list with size n, a copy linked list method, Finding an element at a certain part of the list, a sorting method, and a median method. In addition to this, 3 different types of sorting methods are to be created. These sorting methods are to be based on bubble sort, merge sort, and Quicksort. Another revised Quicksort method to be made is one where one recursive call is made. All these sorting methods are to sort the list and return the median.

Proposed solution design and implementation

To return the median of the list, I duplicated the Median code provided to us in the lab. I decided to make three of those, one for each sorting algorithm. I did this so that the sorting works independently so its more universal and not always returning the median if its not needed. Essentially, I felt that making a method to return the median by calling a sorting algorithm that it is named made more sense. If someone simply just wanted a list sorted, they can call the method without getting the median. If someone wants the median of a bubble sorted list, they can call Bubble_Median which calls the bubble sort algorithm but also returns the median

For bubble sort, comparing a value with the one next to it and shifting all the large ones to the left side was coded. The linked list is determined as sorted if theres no more shifting of values. Mergesort was accomplished recursively by cutting the list in half until one node was left. Once values were in their individual nodes, it would compare and reconstruct a sorted list by iterating through the left side and placing the right values where they belong. Quicksort was also done recursively by picking the last node as the pivot and creating 2 lists, one where the smaller elements are placed and one where the larger elements are placed. These 2 lists are also sent recursively until there is a single node. The list is then reconstructed by appending the pivot and concatenating the left and right.

Experimental results

I created a list of length 10, 100, and 1000 using create_list. I then called each sorting algorithm to sort the list of length 10 to make sure it was sorted (seen below).

Bubble Sort

78 82 66 38 9 91 88 85 12 32
9 12 32 38 66 78 82 85 88 91

Merge Sort

17 97 25 11 68 12 2 67 90 92
2 11 12 17 25 67 68 90 92 97

Quick Sort

82 48 59 32 88 95 1 60 91 4
1 4 82 48 59 32 88 95 60 91

Afterward I called to get the median of each list for each sorting algorithm for each list and got these results.

```
n=10
59
59
67
n=100
58
58
58
n=1000
51
51
87
```

Conclusions

I learned a more visual way to see how different sorting algorithms work. It also put into mind which sorting would be ideal for linked lists, arrays, or any other storage type of data structure. It also helped visualize recursion more and how to implement that into code. Overall the assignment helped understand different ways to sort and how efficient each sort is.

Appendix

```
#####
#####
#
# Course: CS2302 (MW - 1:30pm)
# Author: Ivan Perez
# Assignment: Lab 2
# Instructor: Olac Fuentes
# T.A.: Anindita Nath
# Last Modified: Feb 27, 2019
# Purpose: 2 object functions that consist of making a Node and a linked list
# of nodes. They contain methods to Print, append, prepend, remove nodes, and
# create a random linked list among other things. It also consists of different
# sorting algorithms to sort different sizes of lists.
#
#####
#####
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 Prepend(L, x):# Creates and places a node before a list
    if IsEmpty(L):
        L.head = Node(x)
        L.tail = L.head

    else:
        L.head = Node(x, L.head)

def Search(L, x): # Searches a list's node for item x
    temp = L.head
    while temp is not None:
        if temp.item == x:
            return temp
        temp = temp.next
    return None

def GetLength(L): # returns length of list
    temp = L.head
    count = 0
    while temp is not None:
        count += 1
        temp = temp.next
    return count

def insertAfter(L, w, x): # inserts a new node with item x after node w
    s = Search(L, w)
    if s is None:
        print("Location provided does not exist")
    else:
```

```
s.next = Node(x, s.next)
```

```
def Concatenate(L1, L2): # appends a list to a list
```

```
    if IsEmpty(L1):
```

```
        L1.head = L2.head
```

```
        L1.tail = L2.tail
```

```
    else:
```

```
        L1.tail.next = L2.head
```

```
        L1.tail = L2.tail
```

```
    return L1
```

```
#####
```

```
#
```

```
# Below are the methods required for the assignment
```

```
#
```

```
def createList(n): # creates a random list of integers with size n
```

```
    L = List()
```

```
    for i in range(n):
```

```
        Append(L, random.randint(1, 100))
```

```
    return L
```

```
def Copy(L): # makes a new list and copies the elements into it
```

```
    C = List()
```

```
    temp = L.head
```

```
    while temp is not None:
```

```
        Append(C, temp.item)
```

```
        temp = temp.next
```

```
    return C
```

```
def ElementAt(L, p): # returns the address of the element desired
```

```
    temp = L.head
```

```
    for i in range(p):
```

```
        if temp is None:
```

```
            print("Undefined Location")
```

```
    temp = temp.next
return temp
```

```
def Sort(L): # default sorting algorithm
    change = True
    while change:
        t = L.head
        change = False
        while t.next is not None:
            if t.item > t.next.item:
                temp = t.item
                t.item = t.next.item
                t.next.item = temp
                change = True
            t = t.next
```

```
def Bubble_Median(L): # Calls bubble sort and returns median of the list
    C = Copy(L)
    bubble_sort(C)
    return ElementAt(C, GetLength(C)//2)
```

```
def Merge_Median(L): # Calls merge sort and returns median of the list
    C = Copy(L)
    merge_sort(C)
    return ElementAt(C, GetLength(C)//2)
```

```
def Quick_Median(L): # Calls quick sort and returns median of the list
    C = Copy(L)
    quick_sort(C)
    return ElementAt(C, GetLength(C)//2)
```

```
##### Sorting Algorithms #####
```

```
# Bubble Sort
# Runtime:  $O(n^2)$ 
def bubble_sort(L):
    change = True
```

```

while change:
    t = L.head
    change = False # if if statement is run, sorting is not complete
    while t.next is not None:
        if t.item > t.next.item: # compares value of current node to the next node
            temp = t.item
            t.item = t.next.item
            t.next.item = temp
            change = True #if sorted, condition doesn't change
        t = t.next

```

```

# Merge Sort
# Runtime:  $O(n \log n)$ 
def merge_sort(L):
    if GetLength(L) > 1:
        R = List()
        tail = L.tail
        mid = ElementAt(L, GetLength(L)//2-1)
        R.head = mid.next
        mid.next = None
        L.tail = mid
        R.tail = tail

    if GetLength(L) > 0:
        merge_sort(L)
    if GetLength(R) > 0:
        merge_sort(R)

    if IsEmpty(L):
        Append(L, R.head.item)
        R.head = R.head.next

    if not IsEmpty(R):
        Rtemp = R.head
        Ltemp = L.head
        place = None
        while Rtemp is not None:
            if Ltemp is None:
                Rplace = Rtemp.next
                Append(L, Rtemp.item)

```



```

    Rtemp = Rplace
    Ltemp = L.head
    place = None
elif Rtemp.item < Ltemp.item:
    if place is None:
        Rplace = Rtemp.next
        place = Ltemp
        L.head = Rtemp
        Rtemp.next = place
        Ltemp = L.head
        place = None
        Rtemp = Rplace
    else:
        Rplace = Rtemp.next
        temp = place.next
        place.next = Rtemp
        Rtemp.next = temp
        Rtemp = Rplace
        Ltemp = L.head
        place = None

else:
    place = Ltemp
    Ltemp = Ltemp.next

```

#Quick sort

#Runtime: $O(n \log n)$

def quick_sort(L):

if GetLength(L) > 1:

s = List()

b = List()

temp = L.head

pivot = L.tail

while temp.next is not None:

if temp.item < pivot.item:

Append(s, temp.item)

else:

Append(b, temp.item)

temp = temp.next

```
temp = List()
if GetLength(s) > 0:
    quick_sort(s)
```

```
if GetLength(b) > 0:
    quick_sort(b)
```

```
Concatenate(temp, s)
Append(temp, pivot.item)
Concatenate(temp, b)
L = temp
return L
```

```
else:
    return L
```

```
#----- Test code -----#
```

```
l10 = createList(10)
l100 = createList(100)
l1000 = createList(1000)
```

```
# n = 10
print('n=10')
print(Bubble_Median(l10).item)
```

```
print(Merge_Median(l10).item)
```

```
print(Quick_Median(l10).item)
```

```
# n = 100
print('n=100')
print(Bubble_Median(l100).item)
```

```
print(Merge_Median(l100).item)
```

```
print(Quick_Median(l100).item)
```

```
# n = 1000
print('n=1000')
print(Bubble_Median(11000).item)

print(Merge_Median(11000).item)

print(Quick_Median(11000).item)
```

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.

A handwritten signature in black ink, appearing to read "Ivan Perez". The signature is fluid and cursive, with the first name "Ivan" and last name "Perez" clearly distinguishable.