Luis Renteria

ID 88740232

CS 2302

Dr. Fuentes

Lab Report 1

**Introduction**

Our task for Lab 2 was to sort a linked list, then return the median of said list using different methods. These methods were Bubble Sort, Merge Sort, Quicksort, and a modified version of Quicksort that returned the median without completely sorting the list.

**List Creation**

Before creating the methods, we had to create a list to sort. I decided to create a list based on the input of the user. The integer input is sent to the method BuildList, where a for loop takes an empty list and, using the Append method, fills the list with random integers between 1 and 100.

**Bubble Sort**

Our first sorting algorithm was the Bubble sort method. Bubble sort sorts an array or list by iterating through the list and swapping elements depending on whether the list is sorted in ascending or descending order. For this particular lab, all lists were sorted in ascending order.

For this method, I decided that iteration would be the simplest way to traverse and sort the list. First, I assigned a temporary variable, curr, so that the list could be traversed. Starting with the head, curr traversed the list and compared the current node in the list with the following node. If the current node was less than the following node, the two would be swapped by using another temporary variable to hold the element of the current node so that it is not lost during swapping. Once the swapping was complete, curr was pointed back to the head to ensure that the list is in order after the swap. After sorting the list, the median could be found by taking the list’s length and dividing it by two, then accessing the element at the location found.

**Merge Sort**

Our second sorting algorithm, Merge Sort, is a type of “divide and conquer” method. Merge Sort takes a list and divides it into two sublists by splitting the original list in half. This splitting continues until the list is split into many sublists of size 1. Once the splitting is complete, the list is put back together while continually being sorted.

In my Merge Sort method, two empty lists are created to hold the elements of the original list. By adding the attribute “length” to the linked list, I am able to keep track of the length of a sublist at all times. This allows me to find the midpoint of the list and split it accordingly. All elements to the left of the midpoint go into list L1, while the rest of the right side goes into L2. Merge Sort is then called recursively on each sublist until every sublist only has one element.

Once the splitting is complete, L1 and L2 are sent to the Merge method to be put back together and sorted. Two temporary variables are used to traverse the lists and compare the elements to sort them in ascending order. Using a while loop and a if/else statement, the two lists are combined into one list and returned to the MergeSort method. this process continues until all sublists are combined into one list that is completely sorted in ascending order. With the sorting done, the median can now be returned and printed in the same manner as Bubble Sort.

**Quicksort**

Our third algorithm, Quicksort, is also a type of “divide and conquer” algorithm that uses a pivot to compare the elements of a list and split the list. Much like Merge Sort, the splitting continues until the list is split into many sublists of size one. The list is then put back together in the same manner as Merge Sort.

For my algorithm, I decided to make the very first element in the list the “pivot”. Using a temporary variable, curr, I traversed the list and compared the elements to the pivot. If they were less than the pivot, they were sent to the “smaller” sublist, and if they were larger, they were sent to the “larger” sublist. After comparisons, if the larger sublist was left without any elements, I inserted the pivot, otherwise, I inserted the pivot into the smaller sublist. The quicksort method was called recursively with the smaller and larger sublists until every sublist had a length of one.

With the splitting complete, the smaller and larger sublists were sent to the Merge method. this method was not changed whatsoever, as the method of merging still remained the same. With the list complete and ordered, the median could be found and printed.

**Modified Quicksort**

Our fourth algorithm was a modified version of the previously completed Quicksort. This modified version used a “Rank” method to keep track of every element’s position in the list. This version was supposed to allow for the median to be returned without having to completely sort the entire list, making it the most efficient in terms of finding the median. Unfortunately, I was not able to completely comprehend the workings of this modified version, and as such, I failed to complete this part of the assignment. Due to this method being incomplete, it will not be included in the results table.

**Experimentation Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Sorting Method | Analytical Running Time | Number of Elements | Average Number of Comparisons |
| **Bubble Sort** | O(n^2) | 5 elements  10 elements  15 elements | 4 comparisons  22 comparisons  49 comparisons |
| **Merge Sort** | O(nlogn) | 5 elements  10 elements  15 elements | 5 comparisons  15 comparisons  28 comparisons |
| **Quicksort** | O(nlogn) | 5 elements  10 elements  15 elements | 4 comparisons  12 comparisons  20 comparisons |

As shown by this table, the analytical running times do not precisely match the number of elements as compared to the number of comparisons. Although Bubble sort is the slowest, at 15 elements, the number of comparisons is still far below 15^2. It can also be seen that, even though Merge Sort and Quicksort have the same analytical running time, Quicksort is slightly faster. This is due to the fact that Merge Sort is split in a way that will always yield the same number of comparisons depending on the size, whereas Quicksort can have less or more comparisons depending on each pivot. The worst-case scenario of Quicksort (O(n^2)) is unlikely to happen, resulting in a faster running time.

**Conclusion**

Overall, this lab taught us how different sorting algorithms work. It also demonstrated how efficient each algorithm is and how the method of sorting affects the running time. Although many pre-made sorting algorithms are already built into Python, it is beneficial to see the underlying mechanisms to better understand how running times are affected by different algorithms.

**Appendix**

This is the code that I used form the provided code on the class website.

#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()