Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: September 27, 2019

Instructor: Diego Aguirre

Assignment: Lab 2 Linked List

TA: Gerardo Barraza

**Introduction:**

In python, you can use Linked Lists to traverse through multiple sub lists and find what you are looking for, whether it is an integer or string for example. The task for this lab is to compare two text files, turn them into a single linked list and implement four methods that will help on finding repetitive integers from those files. This is exciting as it uses a real word problem like a merger of two companies to illustrate how we can use this problem in the real world.

**Proposed solution design and implementation:**

1. Compare Number: I would like to create a separate method needed to compare neighbor numbers as stated in solution 1. Especially because I would like to use this method so that I can use it after going through bubble sort and merge sort. I would like to keep this method simple and clean, with counting only how many duplicates are found within a given list.
2. Bubble Sort: There is not a lot of freedom when trying to design the implementation of bubble sort, the idea is to pass higher numbers to the end of the list over and over until we reach the actual sorted list. Perhaps the biggest challenge was to implement this algorithm using linked lists. Either way, I would have to find a way to alter the list with each term moving to the end if necessary, as well as making sure that if the list is correct, that the method would check its status once and end.
3. Merge Sort: There is not a lot of freedom when trying to design the implementation of merge sort, the idea is to split the values of the list by halves and sort each term to their respective place over and over until we reach the actual sorted list. Perhaps the biggest challenge was to implement this algorithm using linked lists, as well as finding a way to skim through the linked list depending of the size of length of the list. Either way, I would have to find a way to split the list into two, and a separate method to create the newly formed list. In order words, I would then need to use two methods for this particular function.
4. Search\_duplicates: For solution 4, I want to do something a little different, I want to find a way to not only find me the number of duplicated values, but to insert them in a list as well so that I can print that individual list when the program ends, and the user is able to see which Ids are duplicated from the combined file.

**Experimental results**:

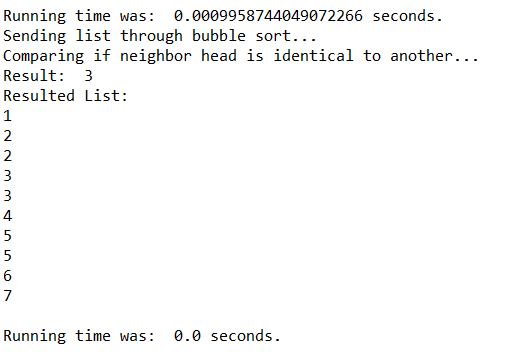
The example shown below was taken from two of my four custom text file, ‘act.txt’, and ‘viv.txt’, which only used 5 items on each list. I decided to portray this text files for my report as I thought it will help with illustrating how the program is supposed to run using a simple example.

1. compare\_num (0(: Originally, I wanted to have this method run in the normal indentation, but I found it faster to implement it as a method to be used in my Linked List class. The implementation of the method is pretty straight forward, plus I was able to add the nested loops like requested from the lab.

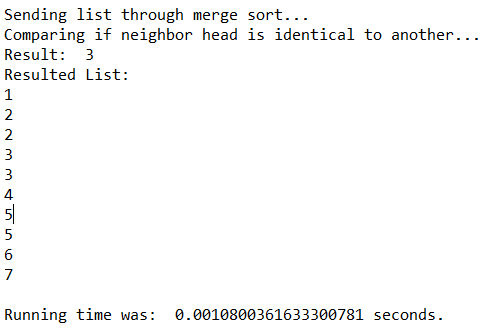
A screenshot of a social media post

Description automatically generated

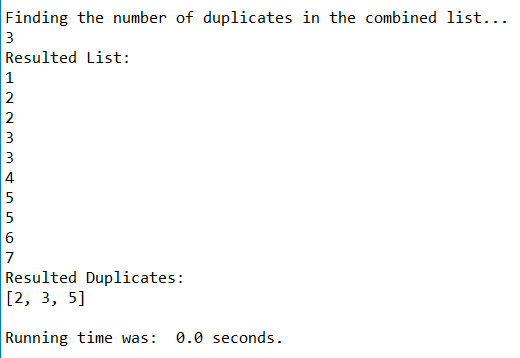
1. Bubble Sort(0(: or this method, I had to create a variable called Completed that would tell me whenever the list was completed in its sorting process. While Completed wasn’t true, the list would sort itself until the desired results were made, and finally returning the middle element in that list.



1. Merge Sort(0( For this method, I had to create three lists, one for the left terms, one for the right terms, and an empty one to become a vessel for the results of the other two lists. My process was pretty much straight forward from the implementation process, but I had to add a Merger method to combine my two new lists into one by indicating whenever the terms of one list is empty, to fill in the void for the other one. At the end, the new list will get its median, and the process was over. Also, I had to add a method that would get the length of the list, so that the partition would work correctly before the program even began.



1. Search\_duplicates (0(: This method was not that difficult to implement, the only issue I had was to use list as a parameter for the beginning, but other than that, the process is simple. The method creates an empty list that will append duplicated Ids into the list, and like mentioned on the lab, I created a parameter that will indicate if I have seen that number before in the list, and automatically add it if true.



The table below shows the name of the method on the top bar, while the name of the text files on the left-hand side. Then I used the time library to calculate how long it took for each method to run. The results are shown below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Files Used | Compare Number | Bubble Sort | Merge Sort | Search Duplicates |
| act.txt,  viv.txt | 0.0009958s | 0.00004s | 0.001080s | 0.000000000023s |
| act1.txt,  viv1.txt | 0.003988s | 0.003988s | 0.007033s | 0.003937s |
| activision.txt,  vivendi.txt | 8.32s | 16.31s | 12.61s | 0.75s |

As shown above, the fastest method was Search Duplicates, followed by the Compare Number, then Merge Sort, and finally Bubble Sort.

**Conclusions**:

With this lab, I was able to learn to code better using the Python language, including using many sorting algorithms to find multiple duplicated integers. I was able to learn to solve different problems by using Linked Lists and classes throughout my lab.

**Appendix :**

**Main.py**

'''

Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: September 24, 2019

Instructor: Diego Aguirre

Assingment: Lab 2 Linked Lists

TA: Gerardo Barraza

Purpose: To practice using linked lists and to apply the

knowledge of running times to compare different sorting

methods.

'''

#Used to calculate the time for each method

import time

#Node Functions given to us by the lab report

class Node(object):

item = -1

next = None

# Constructor

def \_\_init\_\_(self, item = -1, next = None):

self.item = item

self.next = next

def print\_list(self):

curr = self

while curr is not None:

print(curr.item)

curr = curr.next

#List Functions

class List(object):

# Constructor

head = None

tail = None

size = 0

#Default constructors

def \_\_init\_\_(self, node: Node = None):

if node is None:

self.head = None

self.tail = None

self.size = 0

return

if node.next is None:

self.head = node

self.tail = node

self.size = 1

#Method that adds a new head to the linked list

def add\_head(self, data):

if self.head is None:

self.head = Node(data)

self.tail = self.head

self.size += 1

return

curr\_node = Node(data)

curr\_node.next = self.head

self.head = curr\_node

self.size += 1

#Method that prints all the contents of a list

def print(self):

curr\_node = self.head

while curr\_node is not None:

print(curr\_node.item)

curr\_node = curr\_node.next

#Method that combines one list to another, we will use this to combine activision and vivendi

def combine\_list(self, list):

if list.size is 0:

return

if self.size is 0:

return

self.size += list.size

self.tail.next = list.head

self.tail = list.tail

#Method that finds the maximum number in the list

def find\_max(self) -> int:

if self.head is None:

return -1

max\_num = self.head.item

curr\_node = self.head

while curr\_node is not None:

if curr\_node.item > max\_num:

max\_num = curr\_node.item

curr\_node = curr\_node.next

return max\_num

# This function counts number of nodes in the linked list.

def getCount(self):

temp = self.head

count = 0

while (temp):

count += 1

temp = temp.next

return count

#Solution 1

#Method that compares elements of a list from its neighbor, and determine if they are duplicate. At the end it will return the number of duplicates found in the list.

def compare\_num(self):

duplicate = 0

cur\_node = self.head

while cur\_node is not None:

next\_node = cur\_node.next

while next\_node is not None:

if int(cur\_node.item) == int(next\_node.item):

duplicate += 1

break

next\_node = next\_node.next

cur\_node = cur\_node.next

return duplicate

#Makes a new empty list

def NewList(L):

L.head = None

L.tail = None

L.Len = 0

#Method that reads the content of a text file and turns into a brand new linked list

def Text\_to\_list(txt):

file = open(txt, 'r')

L = List()

for line in file:

L.add\_head(int(line))

file.close()

return L

#Sorting Algorithm Functions

#Solution 2

#Method that arranges a given list using bubble sort.

def BubbleSort(L):

if IsEmpty(L):

return None

else:

Current = L.head

Completed = False

while Completed != True:

Completed = True

Current = L.head

while Current.next is not None:

if Current.item > Current.next.item:

nextItem = Current.next.item

Current.next.item = Current.item

Current.item = nextItem

Completed = False

Current = Current.next

#Solution 3

#Method that arranges a given list using merge sort.

def MergeSort(L):

L.Len = L.getCount()

if L.Len > 1:

L1 = List()

L2 = List()

NewLength = L.Len//2

Current = L.head

for i in range(NewLength):

Append(L1, Current.item)

Current= Current.next

while Current != None:

Append(L2, Current.item)

Current = Current.next

MergeSort(L1)

MergeSort(L2)

NewList(L)

MergeList(L, L1, L2)

# Method that inserts x at end of list L

def Append(L,x):

L.Len = L.getCount()

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

L.Len = L.Len + 1

else:

L.tail.next = Node(x)

L.tail = L.tail.next

L.Len = L.Len + 1

#Method that states if the current List is empty or not

def IsEmpty(L):

return L.head == None

#Method that appends sorted Lists into a new list L.

def MergeList(L,L1,L2):

#Grabs the two head of each respective list. Called the variables current as it is the current item the algorithm is analyzing

Current1 = L1.head

Current2 = L2.head

while Current1 != None and Current2 != None:

#Adds the lowest term first of either list

if Current1.item < Current2.item:

Append(L, Current1.item)

Current1 = Current1.next

else:

Append(L, Current2.item)

Current2 = Current2.next

#Clarifies that if either list contains any elements, if so, they will add any remaining items to the new list

if Current1 is None:

while Current2 != None:

Append(L, Current2.item)

Current2 = Current2.next

if Current2 is None:

while Current1 != None:

Append(L, Current1.item)

Current1 = Current1.next

#Solution 4

#Method that searches for duplicates

def Search\_duplicates(list):

#Will use a list to see if the number is duplicate

listedID = []

for i in range(list.find\_max() + 1):

listedID.append(False)

#Will create a list that will add Duplicated items

Duplicate = []

curr\_node = list.head

while curr\_node is not None:

# If the ID is listed, then it will be added it to the duplicate list

if listedID[curr\_node.item]:

Duplicate.append(curr\_node.item)

listedID[curr\_node.item] = True

curr\_node = curr\_node.next

return Duplicate

print('Welcome to the Activision ID generator! Before we begin, please state the following... ')

L1 = input("Name of the first text file : ")

print(L1)

L2= input("Name of the second text file : ")

print(L2)

print('Excellent, commencing program now:')

print('')

activision = Text\_to\_list(L1)

vivendi = Text\_to\_list(L2)

L3 = activision

L3.combine\_list(vivendi)

L3\_COPY\_1 = L3

L3\_COPY\_2 = L3

#Solution 1

#Starts a timer that will calculate the total running time for the solution

start1 = time.time()

print('Beginning without sorting:')

print('Comparing if neighbor head is identical to another... ')

comparison1 = L3.compare\_num()

print('Result: ',comparison1)

print('Resulted List:')

L3.print()

print()

#Ends the timer used to calculate the running time

end1 = time.time()

print()

#Prints out the running time needed for this solution

print('Running time was: ', end1 - start1, 'seconds.')

#Solution 2

#Starts a timer that will calculate the total running time for the solution

start2 = time.time()

print('Sending list through bubble sort... ')

BubbleSort(L3\_COPY\_1)

print('Comparing if neighbor head is identical to another... ')

comparison2 = L3\_COPY\_1.compare\_num()

print('Result: ',comparison2)

print('Resulted List:')

L3.print()

print()

#Ends the timer used to calculate the running time

end2 = time.time()

#Prints out the running time needed for this solution

print('Running time was: ', end2 - start2, 'seconds.')

print()

#Solution 3

#Starts a timer that will calculate the total running time for the solution

start3 = time.time()

print('Sending list through merge sort... ')

MergeSort(L3\_COPY\_2)

print('Comparing if neighbor head is identical to another... ')

comparison3 = L3\_COPY\_2.compare\_num()

print('Result: ',comparison3)

print('Resulted List:')

L3.print()

print()

#Ends the timer used to calculate the running time

end3 = time.time()

#Prints out the running time needed for this solution

print('Running time was: ', end3 - start3, 'seconds.')

print()

#Solution 4

#Starts a timer that will calculate the total running time for the solution

start4 = time.time()

print('Finding the number of duplicates in the combined list... ')

duplicate\_ids = Search\_duplicates(L3)

print(len(duplicate\_ids))

print('Resulted List:')

L3.print()

print('Resulted Duplicates: ')

print(duplicate\_ids)

#Ends the timer used to calculate the running time

end4 = time.time()

print('')

#Prints out the running time needed for this solution

print('Running time was: ', end4 - start4, 'seconds.')

print()