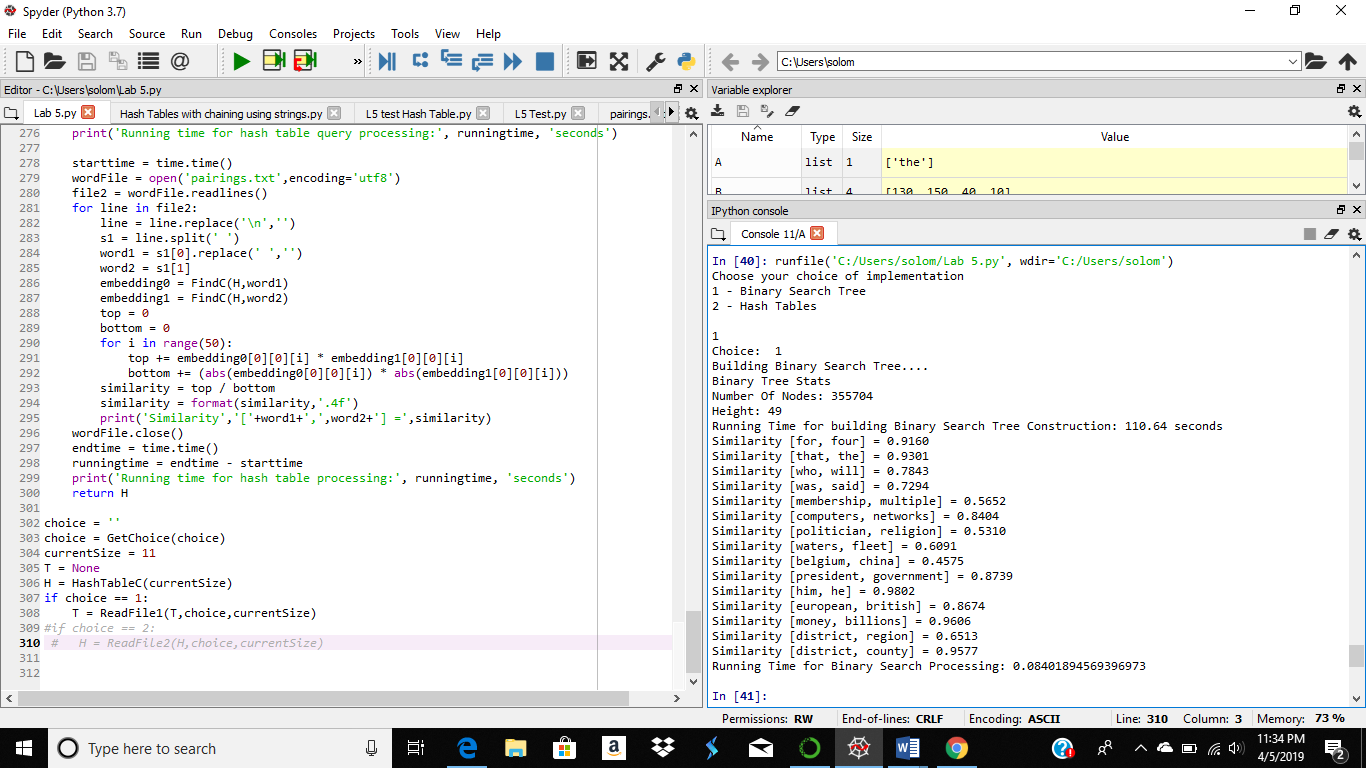
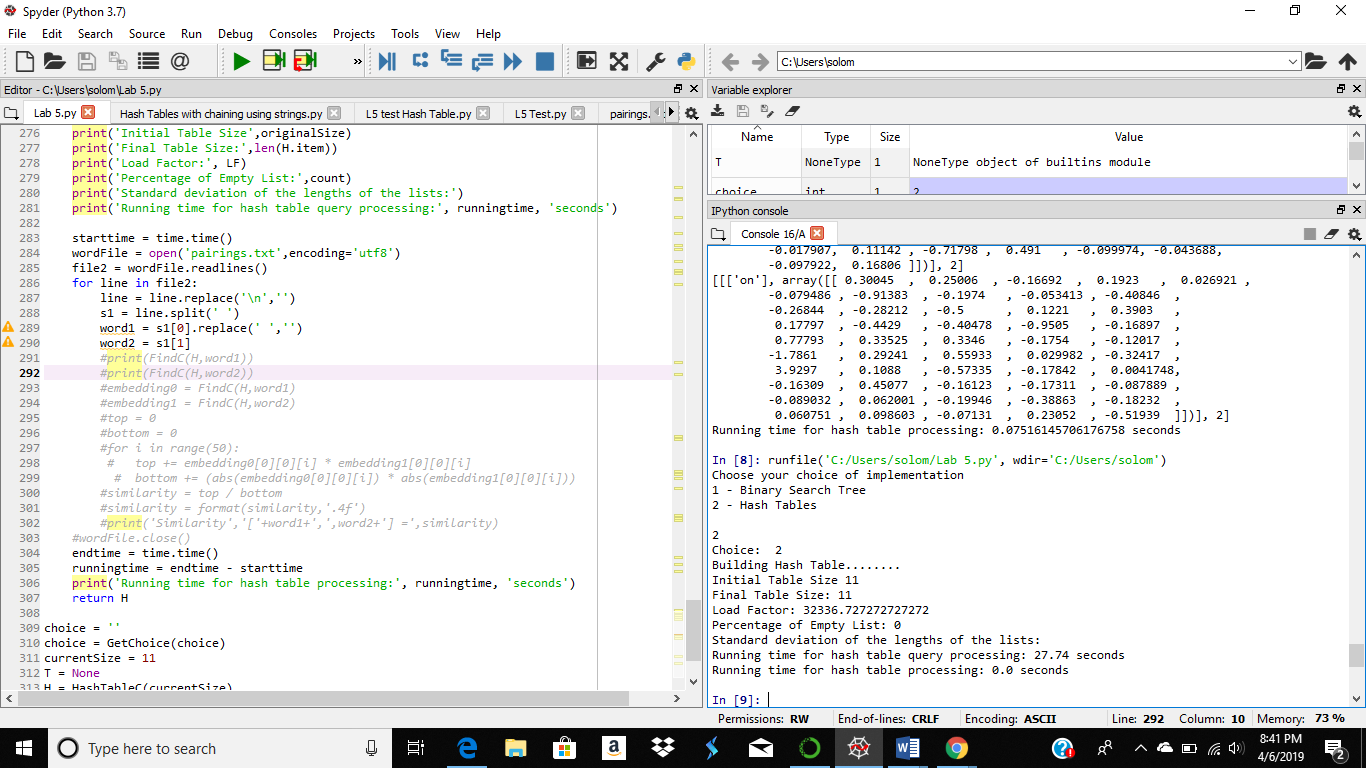
Solomon Davis Lab 5 Report

CS 2302 - MW 1:30 Spring 2019

For lab 5 we were supposed to get information from a file and input the information from the file into a binary search tree and hash table. I was unable to complete the hash table operations as I was able to insert information from the file into the tree but was unable to properly get the stats like the load factor, the standard deviation, and final size of the table. For the binary search tree, I was able to complete all the proper functions. First, I ask the user which choice they want to implement. When the user chooses 1 it builds the binary search tree from the data in the file. I inserted the variable called list which consisted of two list one made of the word that was at the start of every line in the file the other which consisted of 50 embedded values in after the word in the file. After this data was inputted into the binary search tree the data the running time was calculated to see how long it took to build the binary search tree. After this was done another file was read in order to calculate the similarity of the word pairings of the file. The similarity value was calculated by finding matching words and using the values added into the tree.





#Course: CS2302 - Spring 2019

#Author: Solomon Davis

#Lab Number: 5

#Instructor: Olac Fuentes

#Last Modified: April 6, 2019

#Due Date: April 3, 2019

#Description: This Lab will ask a user which choice of implementation 1 for

#a binary search and 2 for a hash table. It will then implement the user choice

# by taking the data from a file and input it into a binary search tree or

#a hash table. After inputtting the data into a hash table or a binary search

#tree it will take data from another file and find the embeddings from the

#binary search tree or hash table and calculate the "similarity" of the word

#pairings in the file.

import numpy as np

import time

# Implementation of hash tables with chaining using strings

class HashTableC(object):

# Builds a hash table of size 'size'

# Item is a list of (initially empty) lists

# Constructor

global num\_items,currentSize

num\_items = 0

currentSize = 0

def \_\_init\_\_(self,size):

self.item = []

#global num\_items

#num\_items = 0

for i in range(size):

self.item.append([])

#num\_items =

def Load(H):

count = 0

for i in range(len(H.item)):

pos = i%len(H.item)

#print(len(H.item[pos]))

if H.item[pos] != []:

if len(H.item[pos]) > 1:

count +=len(H.item[pos])

else:

count +=1

#print(count)

#print()

LoadFactor = count/len(H.item)

return LoadFactor

def InsertC(H,k,l,currentSize):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

global num\_items

#print(num\_items)

#global currentSize

num\_items +=1

#if num\_items == currentSize:

# currentSize = (2\*currentSize)+1

# H2 = HashTableC(currentSize)

# for i in range(len(H.item)):

# if H.item[i] != []:

# InsertC2(H2,H.item[i],len(H.item),currentSize)

#print(k[0][0])

b = h(k[0][0],len(H.item))

H.item[b].append([k,l])

#def InsertC2(H,k,l,currentSize):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

#print(k[0][0][0][0])

#H = HashTableC(currentSize)

#print(k[0][0][0][0])

#b = h(k[0][0][0][0],len(H.item))

#H.item[b].append([k,l])

def FindC(H,k):

# Returns bucket (b) and index (i)

# If k is not in table, i == -1

#print(k)

b = h(k,len(H.item))

for i in range(len(H.item[b])):

if H.item[b][i][0][0][0] == k:

#print(H.item[1:51])

return H.item[b][1]

#print(H.item[b][i][0])

return b, -1, -1

def h(s,n):

r = 0

#print(s[0])

for c in s:

r = (255 + ord(c))% n

return r

class BST(object):

# Constructor

def \_\_init\_\_(self, item, left=None, right=None):

self.item = item

self.left = left

self.right = right

def Insert(T,newList):

#Insets list data from file into the tree

if T is None:

T = BST(newList)

elif T.item > newList:

T.left = Insert(T.left,newList)

else:

T.right = Insert(T.right,newList)

return T

def Delete(T,del\_item):

if T is not None:

if del\_item < T.item:

T.left = Delete(T.left,del\_item)

elif del\_item > T.item:

T.right = Delete(T.right,del\_item)

else: # del\_item == T.item

if T.left is None and T.right is None: # T is a leaf, just remove it

T = None

elif T.left is None: # T has one child, replace it by existing child

T = T.right

elif T.right is None:

T = T.left

else: # T has two chldren. Replace T by its successor, delete successor

m = Smallest(T.right)

T.item = m.item

T.right = Delete(T.right,m.item)

return T

def InOrder(T):

# Prints items in BST in ascending order

if T is not None:

InOrder(T.left)

print(T.item,end = ' ')

InOrder(T.right)

def InOrderD(T,space):

# Prints items and structure of BST

if T is not None:

InOrderD(T.right,space+' ')

print(space,T.item)

InOrderD(T.left,space+' ')

def SmallestL(T):

# Returns smallest item in BST. Returns None if T is None

if T is None:

return None

while T.left is not None:

T = T.left

return T

def Smallest(T):

# Returns smallest item in BST. Error if T is None

if T.left is None:

return T

else:

return Smallest(T.left)

def Largest(T):

if T.right is None:

return T

else:

return Largest(T.right)

def Find(T,k):

# Returns the address of k in BST, or None if k is not in the tree

if T is None or T.item[0][0] == k:

return T.item[1:51]

if T.item[0][0]<k:

return Find(T.right,k)

return Find(T.left,k)

def FindAndPrint(T,k):

f = Find(T,k)

if f is not None:

print(f.item,'found')

else:

print(k,'not found')

def NumberOfNodes(T):

if T is None:

return 0

return 1 + NumberOfNodes(T.right) + NumberOfNodes(T.left)

def height(T):

if T is None:

return 0

leftHeight = height(T.left)

rightHeight = height(T.right)

if leftHeight > rightHeight:

return 1 + leftHeight

return 1 + rightHeight

def GetChoice(choice):

#Ask user for choice and returns the choice

print("Choose your choice of implementation")

print("1 - Binary Search Tree ")

print("2 - Hash Tables")

choice = int(input())

print("Choice: ",choice)

return choice

def ReadFile1(T,choice,currentSize):

#This functions is ran when the user chooses the option to implement the

#file using a Binary Search Tree. After its done inputing the data into the

#tree it the similarity of word pairuings in another file usings the

#embedding of the those words in the binary search tree.

print('Building Binary Search Tree....')

infoFile = open('glove.6B.50d.txt',encoding='utf8') #opens file

file = infoFile.readlines() #reads data from file

starttime = time.time()

for line in file:

if line[0:1] >= 'a' and line[0:1] <= 'z': #checks to see to see if the

#variable is a word

s = line.split(' ') #splits line between word and number array

numbers = np.array([s[1:51]],dtype=float) #stores number array

#into a list variable

#called numbers

word = [s[0]] #stores numbers into list variable called word

List = [word, numbers] #Puts the word and numbers variales into a

#list

T = Insert(T,List) #Inserts List into tree

endtime = time.time()

runningtime = endtime - starttime

runningtime = format(runningtime,'.2f')

print('Binary Tree Stats')

print('Number Of Nodes:', NumberOfNodes(T))

print('Height:', height(T))

print('Running Time for building Binary Search Tree Construction:', runningtime, 'seconds')

infoFile.close()

starttime = time.time()

wordFile = open('pairings.txt',encoding='utf8') #opens wordpairing file

file2 = wordFile.readlines()

for line in file2:

line = line.replace('\n','')

s1 = line.split(' ')

word1 = s1[0].replace(' ','')

word2 = s1[1]

embedding0 = Find(T,word1) #returns embedding for word 1

embedding1 = Find(T,word2) #returns embedding for word 2

top = 0

bottom = 0

#Calculates the similarity for each of the word pairings for the

#file and prints the calclualted value.

for i in range(50):

top += embedding0[0][0][i] \* embedding1[0][0][i]

bottom += (abs(embedding0[0][0][i]) \* abs(embedding1[0][0][i]))

similarity = top / bottom

similarity = format(similarity,'.4f')

print('Similarity','['+word1+',',word2+'] =',similarity)

wordFile.close()

endtime = time.time()

runningtime = endtime - starttime

print('Running Time for Binary Search Processing:', runningtime)

return T

def ReadFile2(H,choice,currentSize):

#This functions is ran when the user chooses the option to implement the file using

#a Hash Table.After its done inputing the data into the

#tree it the similarity of word pairuings in another file usings the

#embedding of the those words in the hash table.

print('Building Hash Table........')

originalSize = currentSize

LF = Load(H)

infoFile = open('glove.6B.50d.txt',encoding='utf8') #opens file

file = infoFile.readlines() #reads data from file

starttime = time.time()

for line in file:

if line[0:1] >= 'a' and line[0:1] <= 'z': #checks to see to see if the

#variable is a word

s = line.split(' ') #splits line between word and number array

numbers = np.array([s[1:51]],dtype=float) #stores number array

#into a list variable

#called numbers

word = [s[0]] #stores numbers into list variable called word

List = [word, numbers] #Puts the word and numbers variales into a

#list

InsertC(H,List,len(List),currentSize) #Inserts List into hash table

LF = Load(H)

count = 0

#for i in H:

# if i == []:

# count+1

endtime = time.time()

runningtime = endtime - starttime

runningtime = format(runningtime,'.2f')

print('Initial Table Size',originalSize)

print('Final Table Size:',len(H.item))

print('Load Factor:', LF)

print('Percentage of Empty List:',count)

print('Standard deviation of the lengths of the lists:')

print('Running time for hash table query processing:', runningtime, 'seconds')

starttime = time.time()

wordFile = open('pairings.txt',encoding='utf8')

file2 = wordFile.readlines()

for line in file2:

line = line.replace('\n','')

s1 = line.split(' ')

word1 = s1[0].replace(' ','')

word2 = s1[1]

#print(FindC(H,word1))

#print(FindC(H,word2))

#embedding0 = FindC(H,word1)

#embedding1 = FindC(H,word2)

#top = 0

#bottom = 0

#for i in range(50):

# top += embedding0[0][0][i] \* embedding1[0][0][i]

# bottom += (abs(embedding0[0][0][i]) \* abs(embedding1[0][0][i]))

#similarity = top / bottom

#similarity = format(similarity,'.4f')

#print('Similarity','['+word1+',',word2+'] =',similarity)

#wordFile.close()

endtime = time.time()

runningtime = endtime - starttime

print('Running time for hash table processing:', runningtime, 'seconds')

return H

choice = ''

choice = GetChoice(choice)

currentSize = 11

T = None

H = HashTableC(currentSize)

if choice == 1:

T = ReadFile1(T,choice,currentSize)

if choice == 2:

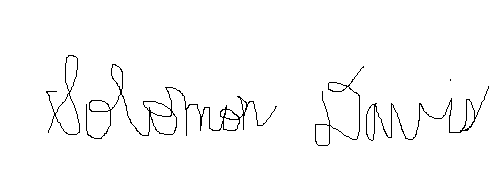
H = ReadFile2(H,choice,currentSize)

Running Times

|  |  |
| --- | --- |
| Binary Search Tree | 47.74 seconds |
| Hash Table | 27.74 seconds |

Academic Service Certificate:

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.



Solomon Davis