Rigoberto Quiroz

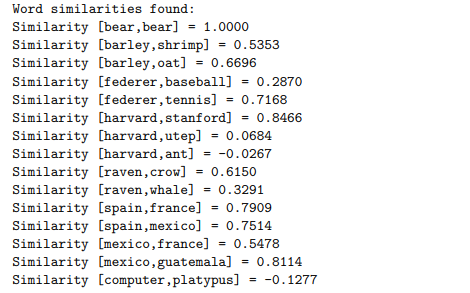
4/03/2019

Lab5 Report

CS2302 1:30 PM – 2:50 PM

Description:

For this lab I we went into Natural Language Processing or NLP in which we get embeddings from words and to see if they match with another word. To accomplish we had to build an NLP with BST and Hash Tables(chaining) data structures and compare how long it takes to build these data structures and how long it takes to retrieve the information of that word and make the comparisons, as shown below.



The way I was able to solve this lab was by altering the previous BST code from lab3 to take strings and embeddings as its information. I organized the information in the BST in alphabetical order. Each node in the BST will contain a string and the embedding that corresponds to that specific word. As for the Hash Table we would create a table with prime length and double it each time over load factor reached 1 or “num\_items” was equal to the length of the table. The way that I inserted data into our table was by making an array that contained the word and its embeddings. The data was dependent of its last character and other variables that would determine where in the table the information would be inserted. Once both BST and Hash table had its information, we had to create another file that contained two words per line. I would then, read that file, take those two strings and find them in each data structure, recording how fast we would acquire the data needed. In the end, Hash table was faster than the BST in making the comparisons and setting all the information.

**5.1 Prompt for User Implementation:**

For this section I prompt the user for an input

1. Hash Table Implementation
2. BST Implementation

Depending on what the user inputs we will create that data structure and extract the information to get the comparisons. If the user inputs some other number than the program will exit.

**5.2 Reading “glove.6B.50d.txt”**

**Hash Tables:**

For this part of the code, I will read line per line with a for loop until the file has no more information. As we are reading each line, I’m going to check if the string starts with a letter, if it does then we will take that word and the embedding and send it to our Insert method. Otherwise if it’s a symbol we will skip that string and its embedding all together and move to the next line. In our Insert method we will update a counter we will in our class method (keeps track of the number of nodes in Hash Table), we will find the index where the data will be inserted and add it to that index. Before we insert that data into our Table, we will divide the number of items by the length to determine if we should increase the size of our table. If we increase the size, we will double the size and add 1 to the table and we will update the index in which we are inserting that node. Once we have inserted the entire file we will re-enter all the values again in a new hash table with size of the previous table. Since we are always updating our first hash table we need to rehash all the old data for it to be placed in their correct positions.

**BST:**

Similarly, with the hash table, I used a for loop that will read each line of the file, check if it starts with a character, and insert the data into our BST with my Insert method. As I send information to the insert method, we will check the current parent or root and placing the data depending on the alphabet, moving left or right to insert data.

**5.3 Compute Stats:**

Number of Nodes: (BST)

For this method we will traverse the entire bst using recursion, adding 1 to our counter each time we move left or right, and stopping when we no longer have branches to extend to.

Height: (BST)

For this method we traverse the tree keeping track of our left and right branches using recursion and keeping a counter for both left and right branches. We will stop until we do not have anything in our bst. Once we are done we will compare both our counters and see which is bigger. We will return the bigger counter.

Initial Table Size: (hash table)

I was able to record the original size our tree saving that value into a variable before inserting any values into the table. Then using it when needed.

Final table Size: (hash table)

Since we have everything stored into our hash table, if we get the length of our table. This will tell us the number of nodes of lists that were created, giving us the final table size.

Load Factor: (hash table)

For this method, we will traverse the table using a for loop. As we traverse the table we will record the nodes or lists that have length greater than 0 and adding them together. Once done we will take our addition of all the length of the sub-list and divide that by the length of our entire lists, giving us our load factor.

Percentage Done (hash table):

For this method, we will traverse the table using a for loop and keep record of the number of empty sub-list we have, adding them together. Once done we will multiply that by 100 and divide that the length of the list. It is done like this because since 100 % is the length of the table, we want to know what the percentage of the empty sub-lists is, so we will cross multiply and divide to get our percentage.

Standard deviation: (hash table)

For this method we will take the average of the entire list, that is counting all the elements in our table (word and embedding count as 1) and then divide that by the total length of our list. Getting the average was done using a for loop. Then we will subtract each length of the sub list by the average and the square that. This was done by making an empty list and inserting each product of our equation into our list. I used a for loop too keep count. Then we will use another for loop to traverse our list created to add all the values and storing it in a value. Once that is done we will multiply that total addition by one over the length of our list, and the squaring it, getting our final answer. This method we will get us the location where most of the nodes are located.

**5.4 Comparisons and 5.5**

In order to make comparisons we have to make a.txt file that contains 2 words line. Once that is done we will read the file and store those two words, depending on the data structure we will search for those words…

Hash table:

Since the data was inserted using an equation we will have to search for that word in the same why. We will get the hash value of that word, access that hash value and go through all the list looking for the word using a for loop. If we find it, we will return its embedding otherwise we will return None (we didn’t find anything) This will be done twice for both words. Once we have the embeddings we will calculate the dot product (adding all the embeddings together from both words and multiplying them together) and getting both magnitudes (adding all the embeddings to the power of 2 from both list and then multiplying both magnitudes and find the square root). All of this is done using 2 for loops, as we read the file we will make those calculations. Then we will print how similar both words are.

BST:

This will work very similar to the comparisons made the hash table in terms of reading the file and getting the calculations of the words such as dot product, magnitudes, etc. The only difference would be how we search for our word. We will first access our root and see if the words match, if they don’t match then depending on where the word lies in the alphabet depending of the root, we will move to the left or right. Process will be repeated until word is found. If we find the word we will return the words embedding otherwise we will return None.

.txt File

bear bear

barley shrimp

barley oat

federer baseball

federer tennis

harvard stanford

harvard utep

harvard ant

raven crow

raven whale

spain france

spain mexico

mexico france

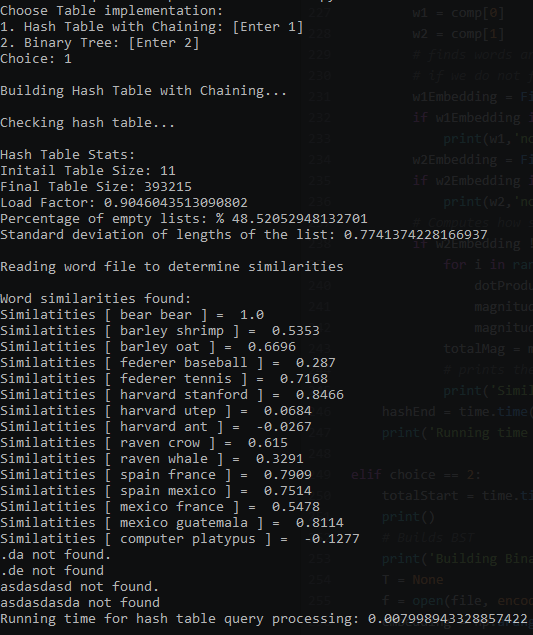
mexico guatemala

computer platypus

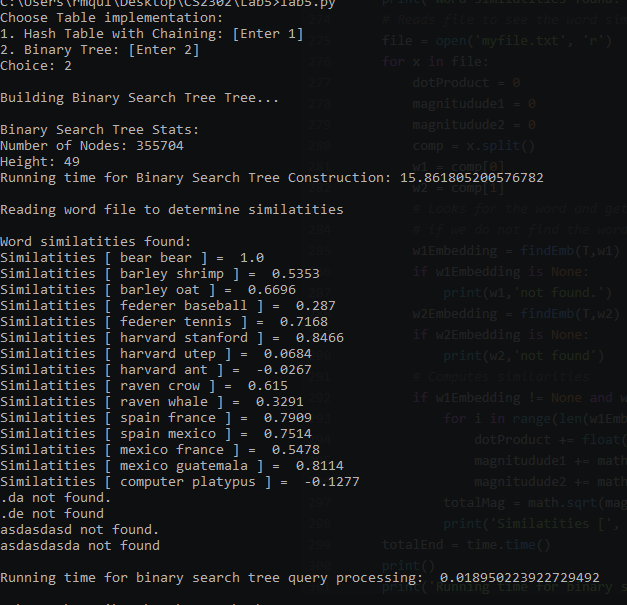
.da .de

asdasdasd asdasdasda

Output:

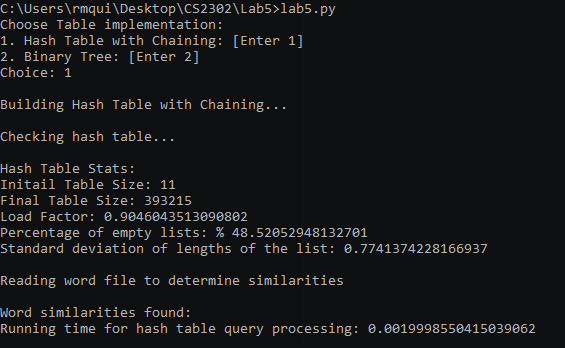


BST:

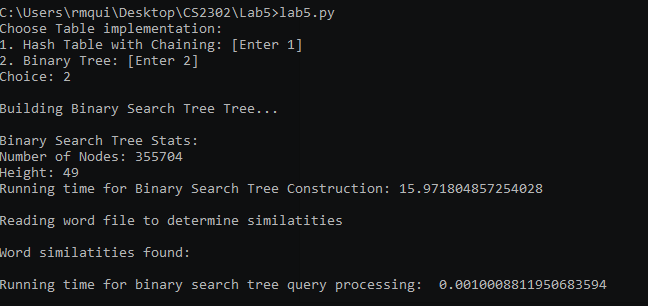


.txt file (empty)

Hash table:



BST:



.txt file

class canvas

canvas protest

protest texture

drum event

temper jeans

jeans mind

summer mind

team glue

crayon trade

event hot

glue back

office scale

texture death

scale back

head roof

mind fan

trade glue

downtown basin

hot fan

manager branch

branch decision

plough pocket

pocket pocket

level jeans

jeans month

calculator theory

view month

month fork

discovery fork

fork fork

value month

territory faucet

trip seashore

trains territory

grandmother meal

sack night

head distribution

canvas class

clover death

event mouth

humor pancake

low temper

crown volcano

cars knife

knife quiver

vacation transport

cart death

transport mouth

account account

kite transport

company company

punishment tooth

crush death

tooth crush

trousers death

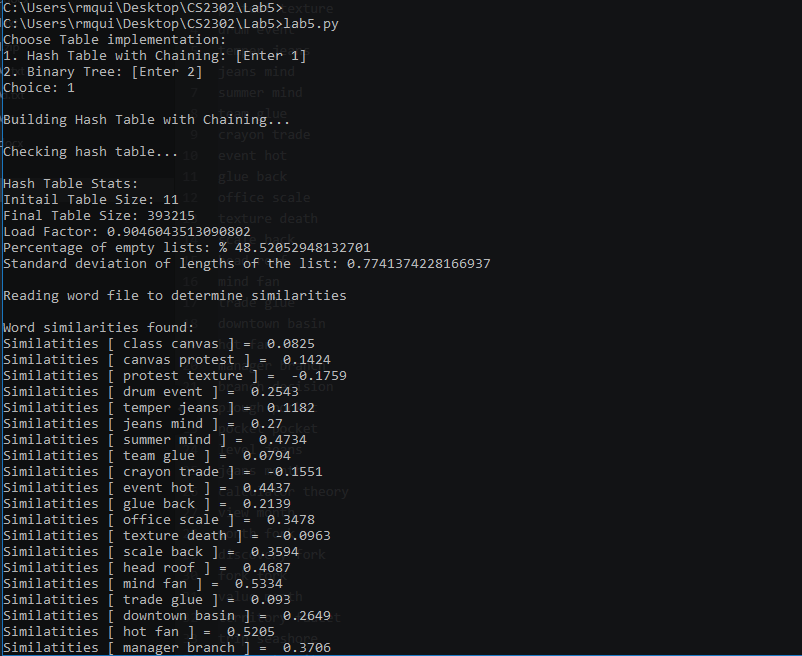
attempt account

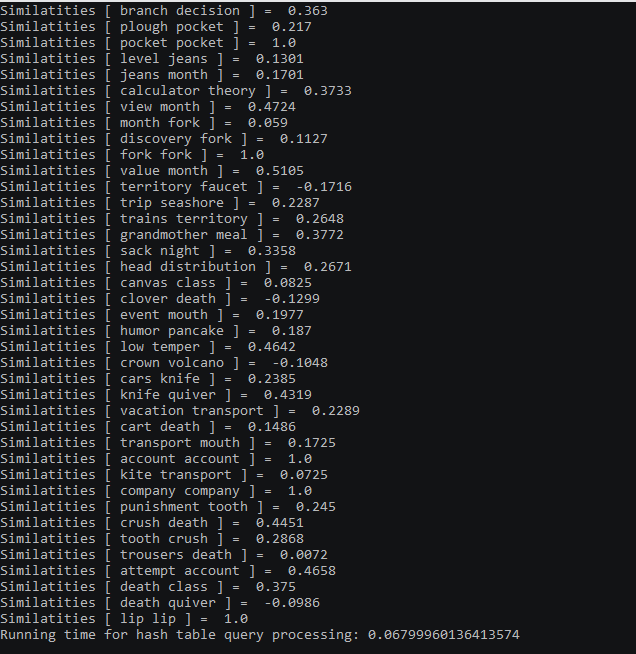
death class

death quiver

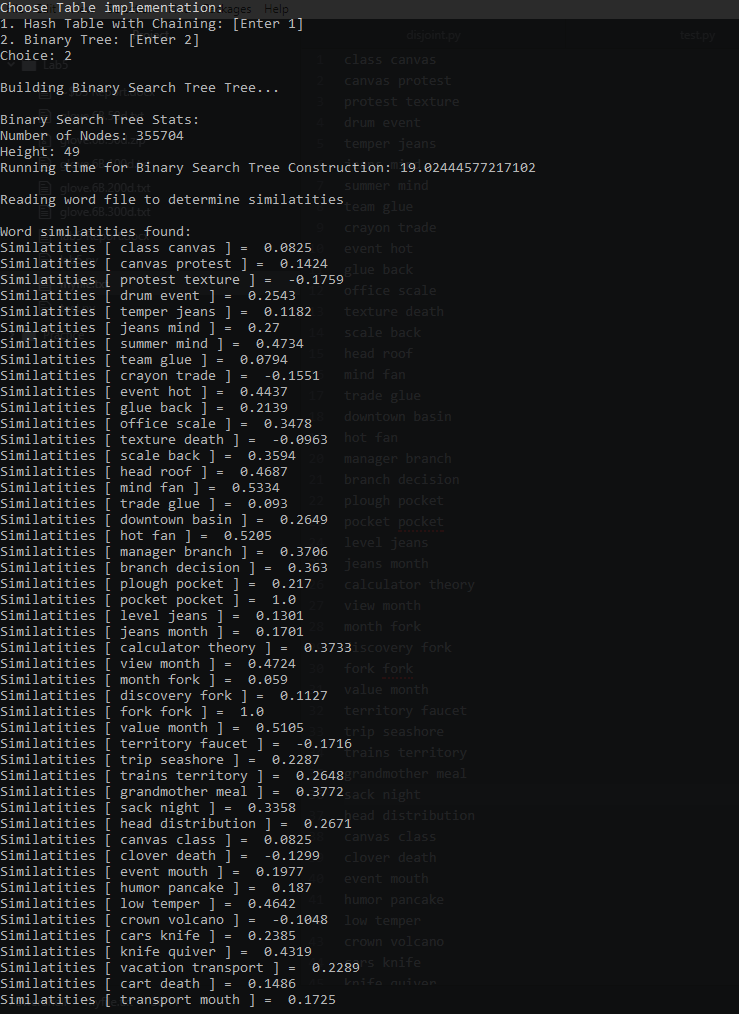
lip lip

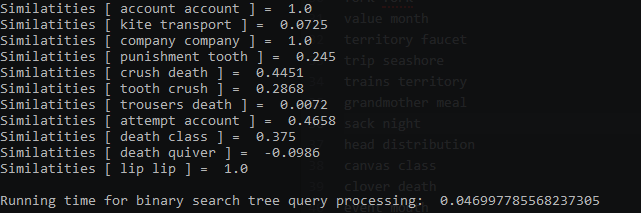
Hash Table:





BST:





In this lab I was able to learn how to implement BST and Hash Table to a much further extent. This was done by adding words and embeddings instead of only numbers. Also insert data from a heavy .txt file into a data structures (BST and Hash Tables) without any problems. Being able to search and extract information from these data structures to compute how similar they are. A better understanding of the running times for BST and Hash tables, how effective each can be when used correctly.

**Apendix:**

# Author: Rigoberto Quiroz

# Section: 1:30 PM - 2:50 PM

# This program will create a Hash table and a BST(Binary Search Tree) and will

# insert data from a file called glove.6B.50d.txt (word and embeddings), into

# these data structures. We will compute varies things such as height of BST,

# Number of nodes, pecentage of empty lists, etc. Once all of these fields have

# been computed we will search for words and their similarities. We will output

# how similary these two words are, the closer to 1, the more similar, the

# closer to -1 the less similar.

import numpy as np

import math

import time

class BST(object):

# Constructor

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

# self.item holds an array contaning a word and its embeddings

self.item = item

self.left = left

self.right = right

def Insert(T,newItem, l):

# If we find an empty stop in our bst we will add the word and embeddings

if T == None:

T = BST([newItem, l])

# Information is organzied in alphabetical order.

elif T.item[0] > newItem:

T.left = Insert(T.left,newItem, l)

else:

T.right = Insert(T.right,newItem, l)

return T

def InOrder(T):

# Prints items in BST in ascending order

if T is not None:

InOrder(T.left)

print(T.word, T.embeding, end = ' ')

InOrder(T.right)

def totalNodes(T):

if T is None:

return 0

if T is not None:

# adds 1 for each node we find in our bst

nodeCounter = 1 + totalNodes(T.left) + totalNodes(T.right)

return nodeCounter

def height(T):

if T is None:

return 0

leftBranch = 1+height(T.left)

rightBranch = 1+height(T.right)

# returns the height of bst

if leftBranch > rightBranch:

return leftBranch

return rightBranch

def InOrderD(T,space):

# Prints items and structure of BST

if T is not None:

print(space,T.word)

InOrderD(T.right,space+' ')

InOrderD(T.left,space+' ')

def findEmb(T, k):

if T is None:

return None

# We found the word we are looking for, returning its embedding

if T.item[0] == k:

return T.item[1]

# Move left if the word comes first than the word we are currently are

if T.item[0] > k:

return findEmb(T.left,k)

# moves right otherwise

return findEmb(T.right,k)

class HashTableC(object):

# Builds a hash table of size 'size'

# Item is a list of (initially empty) lists

# Constructor

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

self.item = []

for i in range(size):

self.item.append([])

# keeps record on how many items our hashTable has

self.num\_Items = 0

def InsertC(H,k,l):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

H.num\_Items += 1

# Locates the location where the information will be placed

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

# if load factor reaches 1 or a greater value

if H.num\_Items / len(H.item) >= 1.0:

# adds more space into our hash table

for i in range(len(H.item)+1):

H.item.append([])

# re-finds a location according to the new information about the hash

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

# adds to hash

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

def FindC(H,k):

# Returns bucket (b) and index (i)

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

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

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

# word found, returning embedding

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

return H.item[b][i][1]

return None

# equation for inserting items into our hash table

def h(s,n):

r = 0

for c in s:

r = (r\*27 + ord(c))% n

return r

def loadFactor(H):

if H is None:

return None

count = 0

# counts how many items we have in our hash table

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

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

# returns its load factor

return count/len(H.item)

def emptyPercentage(H):

if H is None:

return None

# checks how many emty list we have in our hash table

empty = 0

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

if len(H.item[i]) == 0:

empty += 1

# computes percentage and returns it

percentage = (empty \* 100) / len(H.item)

return percentage

def average(H):

if H is None:

return None

totalList = 0

lengths = 0

# computes the average of the length of list in our hashTable

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

lengths += len(H.item[i])

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

totalList += 1

return lengths / totalList

def standardDeviation(H):

if H is None:

return None

# Computes Standard deviation by:

# Getting Average --> subtracting the mean and then squaring it -->

# adding all the values from previous steps --> multiplies the total sum by

# 1/number of items we added --> squaring the final result and returns it

avg = average(H)

a = []

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

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

value = math.pow((len(H.item[i]) - avg),2)

a.append(value)

count = 0;

for j in range(len(a)):

count += a[j]

value2 = count\*(1/len(a))

final = math.sqrt(value2)

return final

# name of file

file = 'glove.6B.50d.txt'

# User has options, which data structure they wanna use

print('Choose Table implementation:')

print('1. Hash Table with Chaining: [Enter 1]')

print('2. Binary Tree: [Enter 2]')

choice = int(input('Choice: '))

# Hash Table Process

if choice == 1:

print()

#Builds hash table

print('Building Hash Table with Chaining...\n')

H = HashTableC(11)

originalSize = len(H.item)

f = open(file, encoding='utf8')

content = []

embedding = np.arange(50)

for x in f:

content = x.split(' ')

if ord(content[0][0]) >= 65 and ord(content[0][0]) <= 90 or ord(content[0][0]) >= 97 and ord(content[0][0]) <= 122:

word = content[0]

embedding = content[1:]

InsertC(H,word, embedding)

# Re-computes hashtable to set all keys in there correct positions

print('Checking hash table...')

print()

count = 0

H1 = HashTableC(len(H.item))

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

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

InsertC(H1,H.item[i][j][0],H.item[i][j][1])

print('Hash Table Stats:')

print('Initail Table Size:', originalSize)

print('Final Table Size:', len(H1.item))

print('Load Factor:', loadFactor(H1))

print('Percentage of empty lists: %', emptyPercentage(H))

print('Standard deviation of lengths of the list:', standardDeviation(H1))

print()

hashStart = time.time()

print('Reading word file to determine similarities')

print()

print('Word similarities found:')

# open file with words to compare

file = open('myfile.txt', 'r')

for x in file:

dotProduct = 0

magnitudude1 = 0

magnitudude2 = 0

comp = x.split()

w1 = comp[0]

w2 = comp[1]

# finds words and gets their embedding

# if we do not find the word then we will break from loop

w1Embedding = FindC(H1,w1)

if w1Embedding is None:

print(w1,'not found.')

w2Embedding = FindC(H1,w2)

if w2Embedding is None:

print(w2,'not found')

# Computes how similar the words are

if w2Embedding != None and w1Embedding != None:

for i in range(len(w1Embedding)):

dotProduct += float(w1Embedding[i]) \* float(w2Embedding[i])

magnitudude1 += math.pow(float(w1Embedding[i]),2)

magnitudude2 += math.pow(float(w2Embedding[i]),2)

totalMag = math.sqrt(magnitudude1 \* magnitudude2)

# prints the info of words

print('Similatities [', w1, w2, '] = ', round((dotProduct/totalMag),4))

hashEnd = time.time()

print('Running time for hash table query processing:', hashEnd - hashStart)

elif choice == 2:

print()

# Builds BST

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

T = None

f = open(file, encoding='utf8')

embedding = np.arange(50)

startTime = time.time()

for x in f:

content = x.split(' ')

if ord(content[0][0]) >= 65 and ord(content[0][0]) <= 90 or ord(content[0][0]) >= 97 and ord(content[0][0]) <= 122:

word = content[0]

embedding = content[1:]

T = Insert(T,word, embedding)

f.close()

endTime = time.time()

print('Binary Search Tree Stats:')

print('Number of Nodes:', totalNodes(T))

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

print('Running time for Binary Search Tree Construction:', endTime - startTime)

print()

totalStart = time.time()

print('Reading word file to determine similatities')

print()

print('Word similatities found:')

# Reads file to see the word similarities

file = open('myfile.txt', 'r')

for x in file:

dotProduct = 0

magnitudude1 = 0

magnitudude2 = 0

comp = x.split()

w1 = comp[0]

w2 = comp[1]

# Looks for the word and gets their embeddings

# if we do not find the word we will break from loop

w1Embedding = findEmb(T,w1)

if w1Embedding is None:

print(w1,'not found.')

w2Embedding = findEmb(T,w2)

if w2Embedding is None:

print(w2,'not found')

# Computes similarities

if w1Embedding != None and w2Embedding != None:

for i in range(len(w1Embedding)):

dotProduct += float(w1Embedding[i]) \* float(w2Embedding[i])

magnitudude1 += math.pow(float(w1Embedding[i]),2)

magnitudude2 += math.pow(float(w2Embedding[i]),2)

totalMag = math.sqrt(magnitudude1 \* magnitudude2)

print('Similatities [', w1, w2, '] = ', round((dotProduct/totalMag),4))

totalEnd = time.time()

print()

print('Running time for binary search tree query processing: ', totalEnd - totalStart)

else:

# if user selects something other than Hash table or BST

print('invalid input')