**Introduction**: For this Lab, I need to be able to create a Binary Search Tree (BST) that will store the information of two text files which one has the information for the similarities of each word that is being compared and the other one contains my own text file of words to be compared. After making my BST, I then need to ask the user if it should be an AVL tree or a Red-Black Tree.

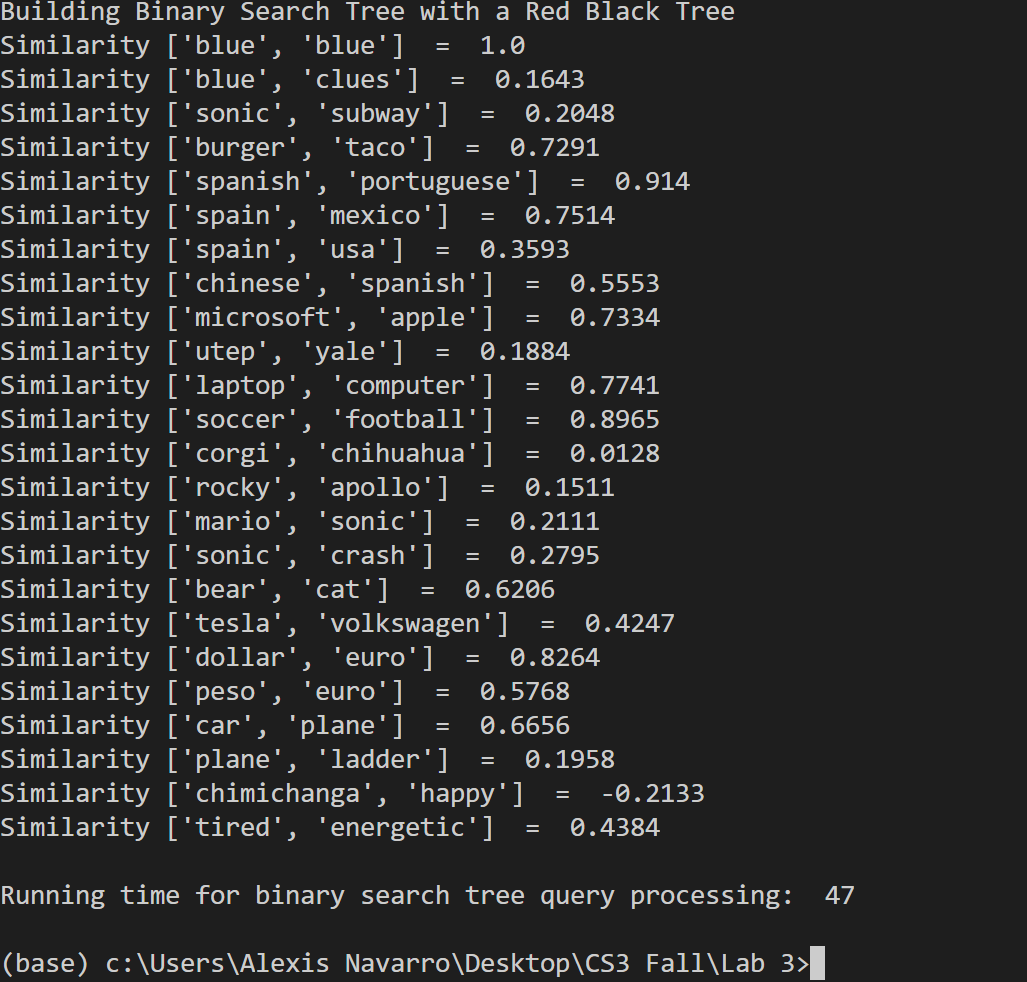
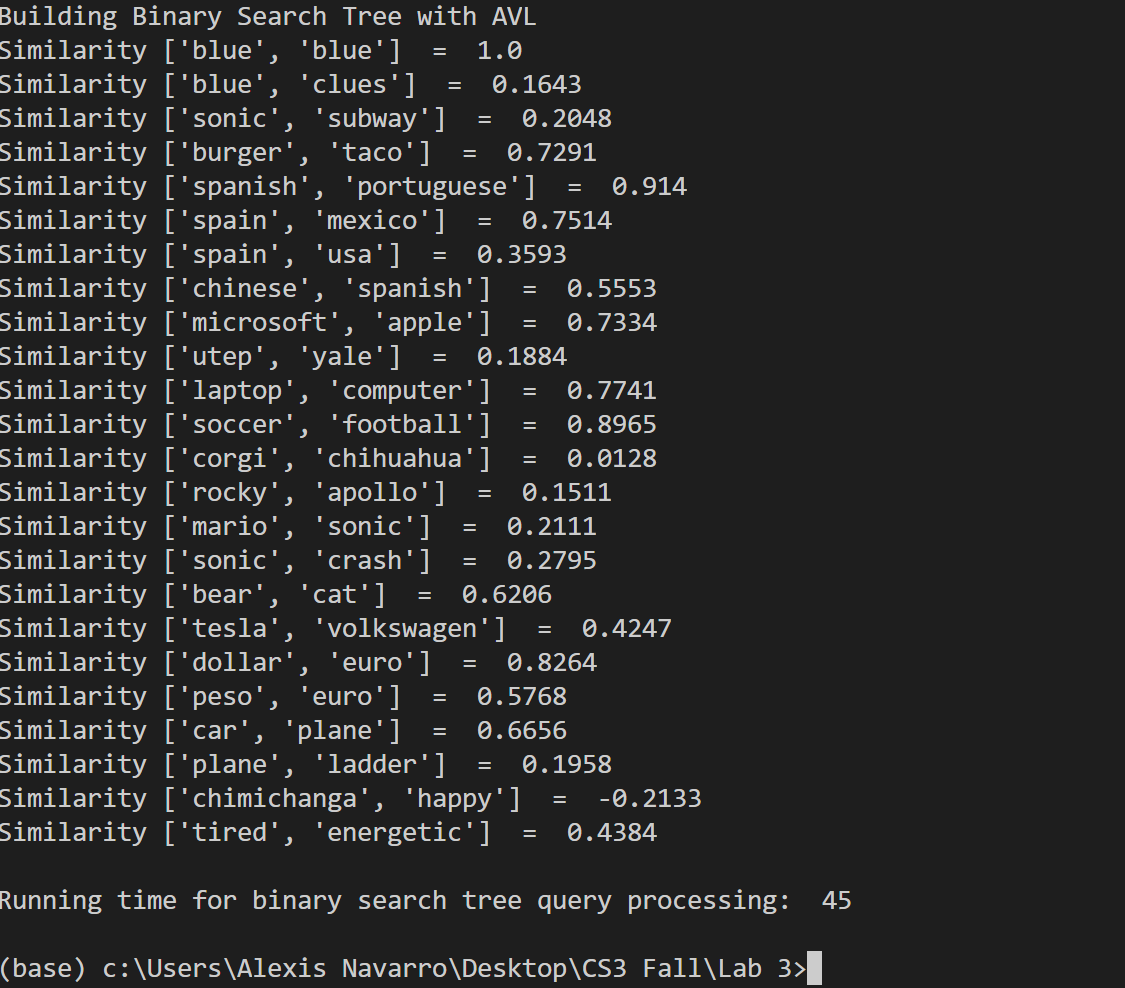
**Proposed** **Solution**: For this problem, I know that before I create an AVL or a Red-Black tree, I need to create a BST because if I don’t have that made then I won’t be able to create the rest of the trees. When making the BST, I need to read both text files and insert them inside the BST, also I need to check the similarities between words after creating my tree. After that is done, I ask the user to enter 1 to create an AVL tree or 2 to create a Red-Black tree. When creating the AVL tree, I first need to check if the BST from before is an AVL tree and to see if it is then I need to check the height of the tree which the height needs to have a difference of 1 or 0 between the left and right branches. If the tree doesn’t meet those conditions, then I need to do the rotations required until the height of the tree meets the conditions. When I’m making a Red-Black tree I need to add the functions of the Red-Black tree into my BST because when I’m checking if it’s Red-Black tree then I need to have to follow the order of the root always being black and the children being red, however the colors can’t be adjacent to each other which means if I have a node that is a color black then the next the children can’t be black, they need to be red.

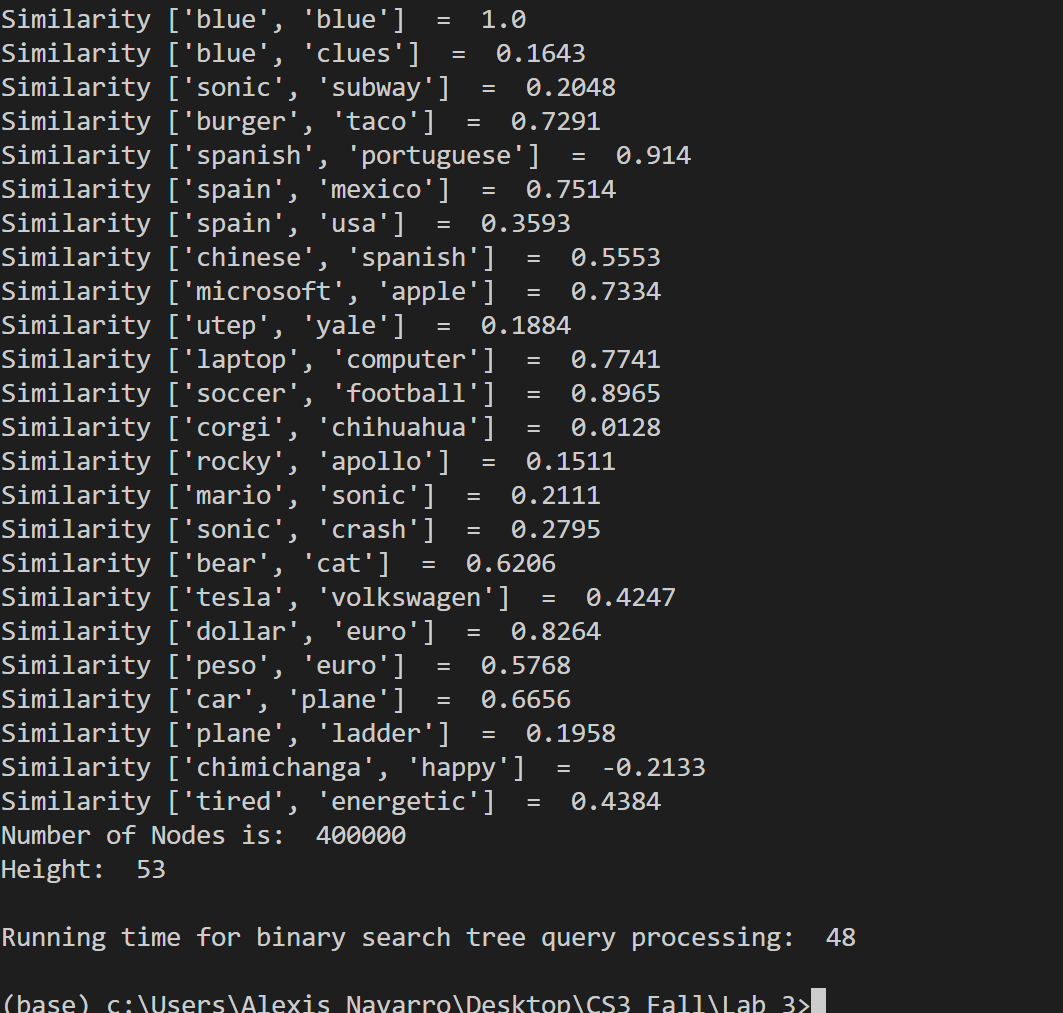
**Implementation**: When creating this lab, I first had to read my two text files which are the information that contains the similarities provided to me, then I had to read my text file which contains a list of words that need to be compared on how similar they are. Once they are read then those two files need to be inserted into the Binary Search Tree that needs to be made first. To create my BST I made a method that passes the two read text files and inserted the information inside that tree, but to do it I read the first file and inserted it to my BST first because the first file contains the similarities info. After that is read and inserted then I read the second file with the list of words, but when I would read it, I had to create a method to find the word in my second text file and check if its in the first file, which is done by iterating through the entire list. After it find the first word in the list then the program looks for the second word in the list, after the are both found I needed to use the equation provided which would find the amount of similarity between words. After that is all done with then I needed to return the BST for later use.

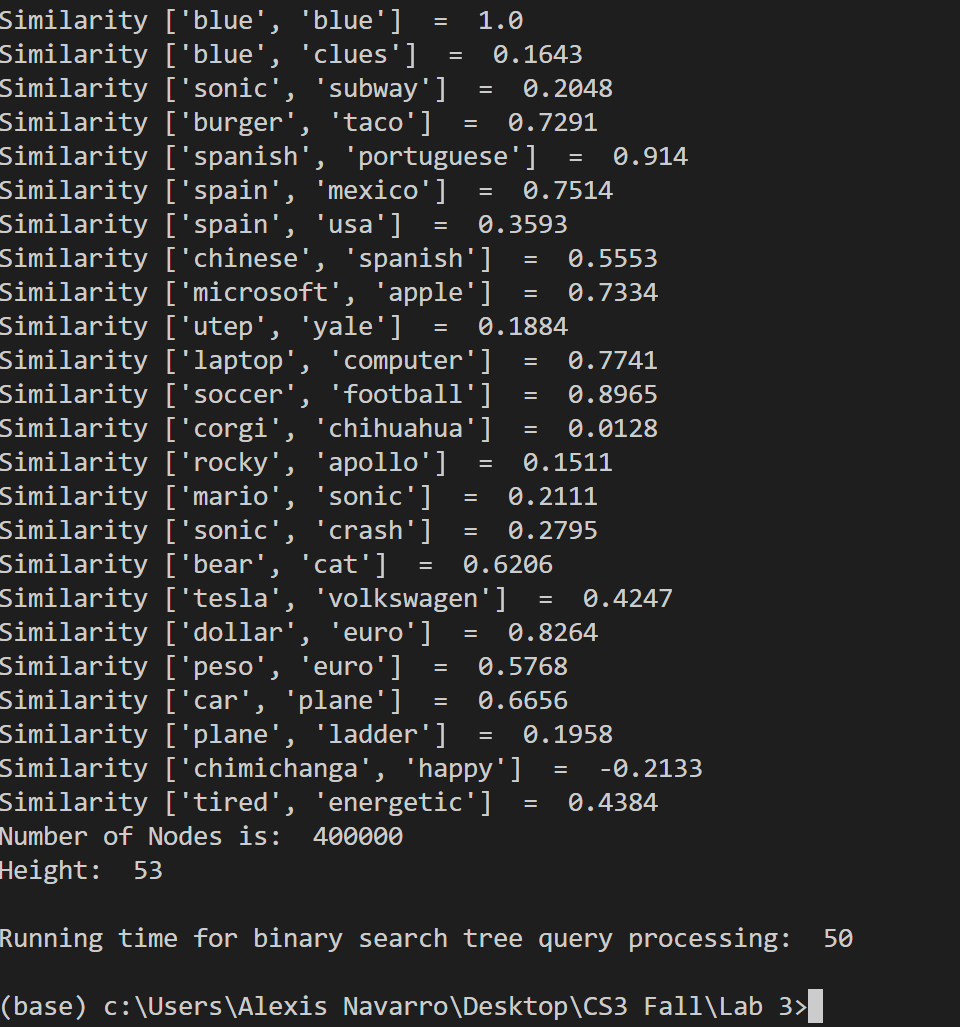
Next up, I need to make the AVL tree which checks if the BST is balanced by comparing the height of the subtrees which I explained earlier. If its not balanced then I need to rebalance the tree which the left height and the right height are being compared until they are at a height of 1 or 0 difference. If its not updated until its balanced then rotations need to be done to the left first then to the right if it doesn’t work. For the RBT the BST needs to take in its properties which include the comparison of colors with it. The BST now holds the colors in each node which the root is black and the children are red, however it checks if there are no colors that are the same and next to each other (if the parent is the same as the children). If they are the same then the tree needs to be changed with more rotations like the AVL tree, however it needs to declare the colors again for each of the updated nodes.

**Errors:** Had encountered errors when trying to write the tree on to an empty text file

**Conclusions:**

****

****



Academic dishonesty includes but is not limited to cheating, plagiarism and collusion. Cheating may involve copying from or providing information to another student, possessing unauthorized materials during a test, or falsifying data (for example program outputs) in laboratory reports. Plagiarism occurs when someone represents the work or ideas of another person as his/her own. Collusion involves collaborating with another person to commit an academically dishonest act.

Professors are required to - and will - report academic dishonesty and any other violation of the Standards of Conduct to the Dean of Students.

# -\*- coding: utf-8 -\*-

"""

Created on Wed Oct 16 16:17:35 2019

@author: Alexis Navarro

CS 2302 MW 1:30 PM -2:50 PM

Purpose: To learn how to create Binary search trees with a text file, but then to apply AVL trees and to use RED-BLACK TREES

"""

import numpy as np

import math

import time

#--------------------------BST TREE--------------------------------------------------------------------------------------------------------

class BST:

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

        self.root = root

        self.item = item

        self.left = left

        self.right = right

def insert(T,newItem):

    if T == None:

        T =  BST(newItem)

    elif T.item > newItem:

        T.left = insert(T.left,newItem)

    else:

        T.right = insert(T.right,newItem)

    return T

def build\_BST(f1,f2):

    tree=None

    f3 = open("tree\_holder.txt","w")

    for line in f1:

        info = line.split(' ')

        tree=insert(tree,[info[0],np.array(info[1:]).astype(np.float)]) #inserts the words and embeddings of the text file

    for line2 in f2:

        data = line2.split(',')

        e0 = findWord(tree,data[0])#returns the list when found

        e1 = findWord(tree,data[1])

        print("Similarity", data[0:2], " = ", round(np.sum(e0 \* e1) / (math.sqrt(np.sum(e0 \* e0)) \* math.sqrt(np.sum(e1 \* e1))), 4))  # compute the similarity

    print('Number of Nodes is: ',num\_Nodes(tree))

    print('Height: ', height(tree))

   #tried to write on to the file and insert my tree but it would not write anything or read the text file.

    #for line3 in f3:

        #f3.write("word in tree:",tree)

        #f3.close()

    #f3 = open("tree\_holder.txt","r")

    #print(f3.read())

    #f3.close()

    return tree

def num\_Nodes(T):

    if T is None:

        return 0

    else:

        return 1 + num\_Nodes(T.left)+num\_Nodes(T.right)

    return 0

def findWord(T,k):

    t = T

    while t is not None:

        if t.item[0] == k:

            #temp.item[1]

            return t.item[1]

        elif t.item[0] > k:

            t= t.left

        elif t.item[0]<k:

            t = t.right

    return None

def height(T):

    if T is None:

        return 0

    leftH = height(T.left)

    rightH = height(T.right)

    if rightH<leftH:

        return leftH+1

    else:

        return rightH+1

#--------------------------AVL-TREE--------------------------------------------------------------------------------------------------------

#The AVL code comes from zybooks and with some modifications added on to it.

class TreeNode(object):

    def \_\_init\_\_(self, val):

        self.val = val

        self.left = None

        self.right = None

        self.height = 1

#method to check if the current tree I have is an AVL, if not then it would construct it.

def build\_AVL(tree):

    AVL\_tree = None

    #print('goes in before if ')

    if is\_Balanced(tree) == True:

        AVL\_tree = tree

        #print('goes in')

        return AVL\_tree

    AVL\_tree=AVL\_rebalance(tree)

    return AVL\_tree

def AVL\_insert(tree,node):

    if tree.root == None:

        tree.root = node

        node.parent = None

        return

    curr = tree.root

    while curr == None:

        if node.item < curr.item:

            if curr.left == None:

                cur.left = node

                node.parent = curr

                curr = None

            else:

                cur = curr.left

        else:

            if curr.right == None:

                curr.right = node

                node.parent = curr

                curr = None

            else:

                curr = curr.right

        node = node.parent

        while node != None:

            AVL\_rebalance(tree,node)

            node = node.parent

def AVL\_rebalance(tree):

    AVL\_updateH(tree)

    if is\_Balanced(tree) == -2:

        if is\_Balanced(tree.right) ==1:

            AVL\_rotate\_right(tree.right)==1

        return AVL\_\_rotate\_left(tree)

    elif is\_Balanced:

        if is\_Balanced(tree.left):

            AVL\_\_rotate\_left(tree, tree.left)

        return AVL\_rotate\_right(tree)

    return tree

def AVL\_rotate\_right(self, z):

        y = z.left

        T3 = y.right

        # Perform rotation

        y.right = z

        z.left = T3

        # Update heights

        z.height = 1 + max(self.height(z.left),

                        self.height(z.right))

        y.height = 1 + max(self.height(y.left),

                        self.height(y.right))

        # Return the new root

        return y

def AVL\_\_rotate\_left(self, z):

            y = z.right

            T2 = y.left

            # Perform rotation

            y.left = z

            z.right = T2

            # Update heights

            z.height = 1 + max(self.height(z.left),

                            self.height(z.right))

            y.height = 1 + max(self.height(y.left),

                            self.height(y.right))

            # Return the new root

            return y

def AVL\_updateH(tree):

    left\_height = -1

    if tree.left!=None:

        left\_height = tree.left.height

    right\_height = -1

    if tree.right != None:

        right\_height = tree.right.height

    tree.height = max(left\_height, right\_height)+1

def AVL\_set\_child(parent, whichChild, child):

    if whichChild != 'left' and whichChild != 'right':

        return False

    if whichChild == 'left':

        parent.left = child

    else:

        parent.right = parent

    AVL\_updateH(parent)

    return True

def AVL\_replace\_Child(parent, curr\_child, new\_child):

    if parent.left == curr\_child:

        return AVL\_set\_child(parent,'left',new\_child)

    elif parent.right == curr\_child:

        return AVL\_set\_child(parent,'right', new\_child)

    return False

def is\_Balanced(tree):

    if tree is None:

        return True

    lh = height(tree.left)

    rh= height(tree.right)

    if (abs(lh - rh)<1) and is\_Balanced(tree.left) is True and is\_Balanced(tree.right) is True:

        return True

    return False

#--------------------------RED BLACK TREE--------------------------------------------------------------------------------------------------------

#RED-BLACK TREE methods come from zybooks with some modfications/additions of my own

class Node():

    def \_\_init\_\_(self, data):

        self.data = data

        self.parent = None

        self.left = None

        self.right = None

        self.color = 1 # 1 . Red, 0 . Black

class red\_black\_tree():

    def \_\_init\_\_(self):

        self.TNULL = Node(0)

        self.TNULL.color=0

        self.TNULL.left = None

        self.TNULL.right = None

        self.root = self.TNULL

    def tree\_balance(self,node):

        if node.parent == None:

            node.color = 0

            return

        if node.parent.color == 0:

            return

        parent = node.parent

        grandparent = GetGrandparent(node)

        uncle = RBTreeGetUncle(node)

        if uncle != None and uncle.color == 1:

                parent.color = uncle.color = 0

                grandparent.color = 1

                tree\_balance(node, grandparent)

                return

        if node == parent.right  and parent == grandparent.left:

            rotate\_left(node, parent)

            node = parent

            parent = node.parent

        elif node == parent.left and parent == grandparent.right:

            rotate\_left(node, parent)

            self = parent

            parent = node.parent

        parent.color = 0

        grandparent.color = 1

        if node == parent.left:

            rotate\_right(node, grandparent)

        else:

            rotate\_left(node, grandparent)

    def GetGrandparent(self):

        if self.parent == None:

            return None

        return self.parent.parent

    def RBTreeGetUncle(self):

        grandparent = None

        if self.parent != None:

            grandparent = self.parent.parent

        if grandparent == None:

            return None

        if grandparent.left == self.parent:

            return grandparent.right

        else:

            return grandparent.left

    def set\_child(self, which\_child,child):

        if which\_child != "left" and which\_child != "right":

            return False

        if which\_child == "left":

            self.left = child

        else:

            self.right = child

        if child != None:

            child.parent = self

        return True

    def replace\_child(self,current\_child,new\_child):

        if self.left == current\_child:

            return set\_child(self, "left", new\_child)

        elif self.right == current\_child:

            return set\_child(self, "right", new\_child)

        return False

    def rotate\_left(self, node):

        right\_left\_child = node.right.left

        if node.parent != None:

            replace\_child(node.parent, node, node.right)

        else:

            tree.root = node.right

            tree.root.parent = None

        set\_child(self.right, "left", node)

        set\_child(self, "right", right\_left\_child)

    def rotate\_right(self, node):

        left\_right\_child = node.left.right

        if node.parent != None:

            replace\_child(node.parent, node, node.left)

        else:

            tree.root = node.left

            tree.root.parent = None

        set\_child(node.left, "right", node)

        set\_child(node, "left", left\_right\_child)

    #--------------------------MAIN--------------------------------------------------------------------------------------------------------

def main():

    file\_info=[]

    my\_file\_info=[]

    tree = BST(None)

    node = Node(None)

    AVL = TreeNode(None)

    RBT = TreeNode(None)

    f1=open('glove.6B.50d.txt',encoding='utf-8') #open the file with the similarities

    f2 = open('List.txt',encoding='utf-8') #uses my own text file to be read later on

    print('Choose table implementation')

    x=int(input('Do you want a AVL Tree (AVL) or Red black trees (RBT)? select 1 for AVL or 2 for RBT: '))

    if x == 1:

        start\_Time=int(time.time())#starting time

        print('Building Binary Search Tree with AVL ')

        tree=build\_BST(f1,f2) # make the BST

        AVL = build\_AVL(tree.root) # checks if the BST is an AVL

        end\_Time = int(time.time())#ending time

        print('\nRunning time for binary search tree query processing: ',(end\_Time-start\_Time))

    elif x == 2:

        start\_Time2=int(time.time())#starting time

        print('Building Binary Search Tree with a Red Black Tree')

        tree = build\_BST(f1,f2)

        RBT= red\_black\_tree.tree\_balance(tree.root,node) #TAKES THE BST TREE TO BE MADE INTO A RED-BLACK TREE

        end\_Time2 = int(time.time())#ending time

        print('\nRunning time for binary search tree query processing: ',(end\_Time2-start\_Time2))

if \_\_name\_\_=="\_\_main\_\_":

    main()