**Introduction**

The purpose of this assignment was to delve shallowly into the topic of how artificial intelligence read and interpret spoken and written language. The way that the AI would understand language effectively regards the use of hash tables and/or binary search trees. These two data structures are most effective as traversing through them can be done in O(logn) time for binary trees and O(1) time for hash tables.

**Proposed Solution Design and Implementation**

The assignment calls for the use of either hash tables or binary trees, depending on which implementation the user chooses. This is accomplished by using a while-loop that accepts user-input. Once the user’s input is acquired successfully, the loop will break and the program then continues to an if/else case. The cases ensure that the implementation corresponds to the user’s input and moves onto the actual implementation.

This is where the program proceeds to read a text file and depending on which implementation chosen, the program creates a hash table or binary search tree of the words and their embeddings from the text file. The use of a for-loop is effective as the for-loop can obtain each the strings from the text file, store the first element/word from a line as a string, and the rest of the fifty embeddings can be stored into their own list as floats. These two objects can be stored into another list, such that every new list contains a word and a list of its embeddings.

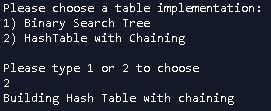
The third step is can be done with multiple functions. The statistics are given by the load factor, the percentage of empty lists, the height, and number of nodes. In reality, it was easiest to create a function with several function calls in print statements to other methods as this made it possible to produce all statistics under one function.

The fourth step consists of reading another file of other strings and embeddings. Creating and populating the second text file was simple as Python allows the user to read and write files simultaneously. Having to read and append two lines per line from the first text file onto the second file proved feasible, which was completed successfully to carry out the assigned instructions. From here, the only function left was the Sim function which computes the similarity between embeddings. To compare two words from the file, the Similarity function was implemented into the loop that reads the file, this ensures that the similarity between the two words’ embeddings are computed. The final step was not met, lack of time, understanding of the material, and lack of preparation kept from the formulation of the material.

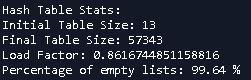
**Experimental Results**

**Hash Table:**

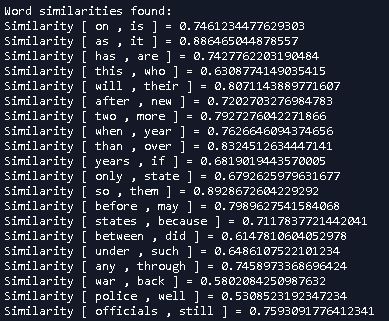
The Decision implementation:



The Stats:

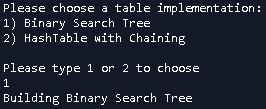


The Similarities:



**Binary Search Tree:**

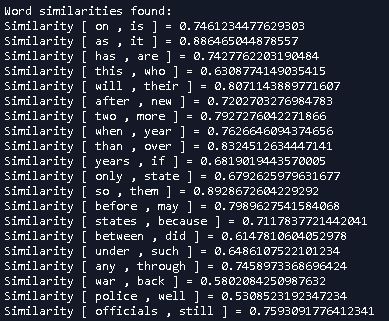
The decision implementation:



The Stats:



The Similarities:



**Conclusions**

The assignment was somewhat successfully completed. The implementation portion proved difficult at first, the reason being that it was not explicitly understood that in Python, lists could be added to a binary search tree without changing the implementation of the tree. It also took some time to understand how to convert string elements of a list into float types. Finally, researching into how to manipulate a language is very useful, as without that research, even just the implementation of hash tables over binary search trees would have been very difficult, if not impossible.

**Appendix**

"""

Course: CS 2302

Author: Jacob Montenegro

Lab: 5

Instructor: Dr. Olac Fuentes

T.A.: Anindita Nath, Maliheh Zargaran

Date of Last Modification: 4/1/2019

Program's Purpose: Using embeddings from a text file, compute the similarity

between 2 words, as well as store a significant amount of words into a hash table or BST, as chosen by the user.

"""

import sys,math

import numpy as np

# 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

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

self.item = []

self.initial = size

self.num\_items = 0

self.final = size

#load factor need H.final\*2 as well

for i in range(size):

self.item.append([])

def InsertC(H,k,l):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

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

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

H.num\_items +=1

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])):

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

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

return b, -1, -1

def h(s,n):

r = 0

for c in s:

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

return r

# Code to implement a binary search tree

# Programmed by Olac Fuentes

# Last modified February 27, 2019

class BST(object):

# Constructor

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

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 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[0])

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 == k:

return T

if T.item<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')

#returns the height of the tree

def Height(BST):

if BST is None:

return -1

LHeight = Height(BST.left)

RHeight = Height(BST.right)

if LHeight > RHeight:

return LHeight + 1

else:

return RHeight + 1

#Returns the number of nodes

def NumOfNodes(BST):

if BST is None:

return 0

return 1 + NumOfNodes(BST.left) + NumOfNodes(BST.right)

#computes the percentage of empty lists in the hash table

def EmptyListPercent(H):

c = 0

#counts each list that has length less than 1

for i in range(H.final):

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

c += 1

return c\*100/H.final

#computes Load Factor

def LoadFactor(H):

c = 0

for i in H.item:

c += len(i)

return c/len(H.item)

#doubles the size of the hash table and re-hashes the pre-existing hash table items into the new hash table

def DoubleHash(H):

#initializes a new table of 2 times +1 size bigger than the previous table

H2 = HashTableC(2\*H.final+1)

#loops through the pre-existing array

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

#ensures that the lists with values in them get hashed

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

#inserts the string, then appends its corresponding embedding

InsertC(H2,H.item[i][0][0],H.item[i][0][1])

#returns new hash table

return H2

#first window that displays the table implementation

def Implementation():

print('Please choose a table implementation:')

print('1) Binary Search Tree')

print('2) HashTable with Chaining')

#ensures that keyboard input is an integer

try:

c = int(input('Please type 1 or 2 to choose\n'))

except:

c = 0

#returns user input or 0 if anything other than an integer was input

return c

def BinaryTree(T):

try:

c = 0

#opens file to be read

with open('glove.6B.50d.txt','r',encoding='utf8') as file:

line = True

#loops while the file has a line to read

for line in file:

line = file.readline()

#checks to see if the first element of the string is a letter

if line.split()[0].isalpha():

#the next 4 lines store the word and the floats as floats into an array

a = line.split()[0]

array = line.split()[1:]

array = [float(i) for i in array]

final = [a,array]

#inserts the array into the tree

T = Insert(T,final)

c+=1

#ensures the loop only loops two hundred thousand times

if c == 200000:

break

except:

print('Error')

sys.exit(1)

return T

def HashTableChaining(H):

try:

c = 0

#reads file for the hash table implementation

with open('glove.6B.50d.txt','r',encoding="utf8") as file:

line = True

#initializes the size of the hash table to 13

H = HashTableC(13)

#loops while there is a line to read in the file

for line in file:

line = file.readline()

#ensures that the string to be implemented is a word

if line.split()[0].isalpha():

a = line.split()[0]

array = line.split()[1:]

#converts all the elements in the list (numbers) into floats

array = [float(i) for i in array]

#inserts the string and float array into the hash table

InsertC(H,a,array)

c+=1

#ensures that the number of items is not greater than the size of the hash table

if H.num\_items > len(H.item)\*2:

H = DoubleHash(H)

#ensures that the loop only loops two hundred thousand times

if c == 200000:

break

except:

print('Error')

sys.exit(1)

return H

def BinaryTreeStats(T):

print('Binary Search Tree Stats:')

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

print('Height:',Height(T),'\n')

def HashTableStats(H):

print('Hash Table Stats:')

print('Initial Table Size:',13)

print('Final Table Size:',H.final)

print('Load Factor:',LoadFactor(H))

print('Percentage of empty lists:',round(EmptyListPercent(H),2),'%\n')

def AnotherFile():

try:

array = None

bray = None

#Reads file 'WordPairs.txt'

with open('WordPairs.txt','r',encoding='utf8') as file:

#initializes line

line = True

print('Word similarities found:')

#checks that there is a line to read in the file

for line in file:

line = file.readline()

#Ensures that the words given are of the alphabet

if line.split()[0].isalpha() and line.split()[51].isalpha():

a = line.split()[0]

b = line.split()[51]

#Stores the embeddings in arrays

array = line.split()[1:50]

bray = line.split()[52:99]

#creates all strings from array into floats

array = [float(i) for i in array]

final = [a,array]

bray = [float(i) for i in bray]

binal = [b,bray]

#moves onto the Similiarity function that computes the Sim(w1,w2)

Similarity(final,binal)

except:

print('Error')

sys.exit(1)

def Sim(w1,w2):

#retreives and stores the floats of the list into a and b

a = w1[1][0:47]

b = w2[1][0:47]

#computes the dot product

q = np.dot(a,b)

u = 0

#both for loops compute the absolute value of the embeddings of both lists

for i in a:

i \*= i

u += i

v = 0

for i in b:

i \*= i

v += i

#returns the square root to produce a finished absolute value

r = math.sqrt(u)

x = math.sqrt(v)

#returns the dot product of the embeddings divided by the absolute value of a's and b's lists

return q/(r\*x)

def Similarity(w1,w2):

print('Similarity [',w1[0],',',w2[0],'] =', Sim(w1,w2))

decision = 0

while decision == 0:

decision = Implementation()

#ensures that the user can only choose either BST or hash table implementation

if decision == 1 or decision == 2:

break

decision = 0

print('Try again')

if decision == 1:

print('Building Binary Search Tree\n')

T = None

T = BinaryTree(T)

BinaryTreeStats(T)

else:

print('Building Hash Table with chaining\n')

H = None

H = HashTableChaining(H)

HashTableStats(H)

#this computes the second file's word pairs

AnotherFile()

I certify that this project is entirely my own work. I wrote, debugged, and tested the code 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. Signed, Jacob M. Montenegro.