Joaquin Hidalgo

CS 2302 MW @1:30pm

Lab 05

**Hash map w/ chaining vs BST**

**Introduction:** The problem at hand is to demonstrate how to insert data read from a file, write to file and gather information about the data structure to display the similarity of two words. The two data structures we will be using is a hash map with chaining and a BST. We will compare the running times of each program and display results.

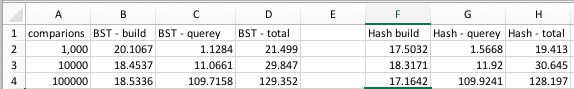
**Proposed solution design and implementation:** To compare 2 words and gather information from the 50-floating point array in python that it correlated to each word, we need to insert a BIG amount of data to have plenty of words to choose from. Here we will read a file and insert it to the BST or Hash map. When inserting data into the BST, I will be comparing words to each other to figure out where the node should be inserted into. For the hash map, I will insert the data by changing a string to a numerical value, find the modulo of the that value over the size of the hash map and determine what bucket my word and 50-floating point array should be inserted in.

After inserting data into the BST or Hash map, I will generate random keys from the list of possible keys read from the file, choose 2 words randomly and write those 2 words on a new file.

Then after writing into a text file, I will read that text file of just key of words, obtain the pairs and display the similarity to each word by using the formula provided using the dot product and magnitude of the arrays and display running times. When trying to find the word in the BST vs the hash map, theoretically, a hash map is supposed to be faster than a BST. Here are my running times:

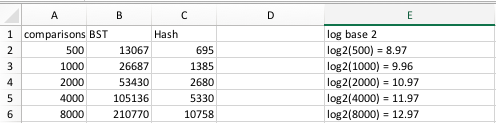
**Experimental results:**

**RUN TIMES (sec)**

****

Theoretically, the Hash map should be much faster than the BST, but in my results, the hash map is just about the same time as the BST to query but hash map is faster at building most of the time. I wanted to see the hash map be faster to find in constant time, versus a find in BST which is log(n).

**NUMBER OF COMPARISONS**

****

Here I decided to compare the amount of times in the BST, how many nodes are checked for the total amount of words. Here I found better results because on average, the BST takes log(n) to search and if you take the log(n) of the size of the list and multiply it times the amount of words we are searching, the total comparisons for the BST is just a little bit above log(n) for each word.  
Now a hash is supposed to insert and find at constant time O(1) and if you take the amount of comparisons we are doing, you will see that the amount of comparisons for each word is between 1 and 2 searches (closer to 1) which I think is great.

**Conclusion:** I learned how to read and write txt files to gather information from the data. From the data I entered it into the BST or Hash map with chaining. From here I learned more about SEO and comparing two words to determine their similarity. After displaying the similarity, I show the running times of each program and theoretically, a hash map that works in constant time should be much faster than the BST who takes log(n) to search for an item. I have to compare really big numbers to start to see the effect.

**Code:**

﻿﻿"""

By: Joaquin Hidalgo

Lab 05

Created Tue 26 Mar 18:48:16

Instructor: Olac Fuentes

TA: Dita and Mali

Last modified Mon 1 Apr 23:50:20

Implement BST vs Hash funtion to determine similarities between 2 words.

We practice to insert to either data structure while reading and writing txt files.

We search either data strcuture to find word and compare each of its own array

to each other, preform math on both vectors and display similarty and runtimes

"""

import random

import time

import math

import numpy as np

class BST(object):

def \_\_init\_\_(self, item=[]):

self.key = item[0]

self.vec = item[1]

self.left = None

self.right = None

def Insert\_BST(T, insertee):

if T == None:

T = BST(insertee)

else:

if T.key > insertee[0]:

T.left = Insert\_BST(T.left, insertee)

else:

T.right = Insert\_BST(T.right, insertee)

return T

def NumOfNodesBST(T):

if T is not None:

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

return 0

#Get height of BST

def GetHeightBST(T):

if T == None:

return 0

left = GetHeightBST(T.left)

right = GetHeightBST(T.right)

#At the last node of the BST (leaf)

if T.left == None and T.right == None:

if left > right:

return GetHeightBST(T.left)

else:

return GetHeightBST(T.right)

#Children node are not empty, add 1 and continue to count edges of path

else:

if left > right:

return GetHeightBST(T.left) + 1

else:

return GetHeightBST(T.right) +1

# Prints items and structure of BST

def InOrderD(T,space):

if T is not None:

InOrderD(T.right,space+' ')

print(space,T.key)

InOrderD(T.left,space+' ')

def WriteToTxtBST(list\_key, compar):

wfile = open('names\_BST.txt','w')

for j in range(compar):

g1 = random.randint(0,len(list\_key)-1)

g2 = random.randint(0,len(list\_key)-1)

wfile.write(list\_key[g1])

wfile.write(' ')

wfile.write(list\_key[g2])

wfile.write('\n')

wfile.close()

#Reads 2nd file and appends every pair to list

def ReadGetListBST(arr):

with open ('names\_BST.txt', 'r', encoding='utf-8') as keyFile:

for l in keyFile:

line = l.split(' ')

tWord = line[1]

tWord = tWord[0:-1]

arr.append(line[0])

arr.append(tWord)

return arr

def FindAndGetArrayBST(T, sear):

if T == None:

return None

else:

if T.key == sear:

return T.vec

elif T.key > sear:

return FindAndGetArrayBST(T.left, sear) #removed return

else:

return FindAndGetArrayBST(T.right, sear) # removed return

########################################################################################################################

class HashTableC(object):

# Builds a hash table of size 'size'

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

self.item = []

self.num\_size = 0

for i in range(size):

self.item.append([])

def InsertChain(H,inss):

global hash\_size

index = HashFunc(H, inss[0])

H.item[index].append(inss)

H.num\_size += 1

def InsertChainFull(H,inss):

global hash\_size

P = None

hash\_size = (hash\_size \* 2) + 1

P = HashTableC(hash\_size)

#checks every Bucket

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

#checks each node in the bucket

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

#H.item[i][j] is list and keyword

InsertChain( P, H.item[i][j] )

P.num\_size += 1

index = HashFunc(P, inss[0])

P.item[index].append(inss)

return P

def GetVecHash(H, word):

bucket = HashFunc(H, word)

# len(H.item[index]) == length of the bucket where the word is at

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

if word == H.item[bucket][i][0]:

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

return bucket, -1, -1

def HashFunc(H, word):

r = 1

n = len(H.item)

for c in word:

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

return r

# n = len(H.item)

# r = 3

# r = (r\*255 + ord(word[-1]))% n

# return r

def WriteToTxtHash(list\_key, compar):

wfile = open('names\_Hash.txt','w')

for j in range(compar):

g1 = random.randint(0,len(list\_key)-1)

g2 = random.randint(0,len(list\_key)-1)

wfile.write(list\_key[g1])

wfile.write(' ')

wfile.write(list\_key[g2])

wfile.write('\n')

wfile.close()

#return a list of all strings in the text file in pairs per each line

def ReadGetListHash(arr):

with open ('names\_Hash.txt', 'r', encoding='utf-8') as keyFile:

for l in keyFile:

line = l.split(' ')

tWord = line[1]

tWord = tWord[0:-1]

arr.append(line[0])

arr.append(tWord)

return arr

def Percentage\_Empty(H):

if H == None:

return 1.0

else:

counter = 0

for i in H.item:

if len(i) == 0:

counter +=1

return(round(counter / len(H.item), 4))

# User-Interface

########################################################################################################################

sim\_length = 100000

initial\_size = 63

hash\_size = initial\_size

choice = int(input('Choose table implementation\nType 1 for BST or 2 for hash table with chaining: '))

all\_letters = 'abcdefghijklmnopqrstuvwxyz'

A = None # BST with DATA

B = None # Hash map with DATA

keyBST = [] #holds keys

keyHash = [] #holds keys

dataBST = [] #Reads 2nd file and stores strings into list

dataHash = [] #Reads 2nd file and stores strings into list

dev1 = [] #Deviation list

##############################

# Main

if choice == 1:

startTotalBST = time.time()

startBuildBST= time.time()

with open ('/Users/joaquin/UTEP/Class/cs2302/lab05\_txt/glove.6B.50d.txt', 'r', encoding='utf-8') as txt\_file:

for l in txt\_file:

flag = False

allSplit = l.split(' ')

a = allSplit[0]

b = np.array(allSplit[1:], dtype = np.float)

#Checks if first string starts with a letter

for j in all\_letters:

if j == a[0]:

flag = True

#If word, insert to BST and append to keyBST

if flag == True:

keyBST.append(a)

temp = [a,b]

A = Insert\_BST(A, temp)

endBuildBST = time.time()

print('Choice: ', choice)

print('\nBuilding binary search tree')

print('\nBinary Search Tree stats:')

print('Number of nodes: ', NumOfNodesBST(A) )

# print('Height: ', GetHeightBST(A) )

print('Running time for binary search tree construction: ' ,round(endBuildBST- startBuildBST, 5))

print()

#write random keys to file to compare

WriteToTxtBST(keyBST, sim\_length)

#read 2nd file and save strings in array

dataBST = ReadGetListBST(dataBST)

startQueryBST = time.time()

#compare each string pair to find similarity and display

for i in range(1,len(dataBST), 2):

#gets array for corresponding word into (v1, v2)

v1 = FindAndGetArrayBST( A,dataBST[i-1] )

v2 = FindAndGetArrayBST( A,dataBST[i] )

#calculate similarity

dot\_p = v1\*v2

dot\_p = sum(dot\_p)

m0 = 0

m1 = 0

for j in v1:

m0 += j\*\*2

for j in v2:

m1 += j\*\*2

m0 = math.sqrt(m0 )

m1 = math.sqrt(m1 )

final\_result = dot\_p / (m0\*m1)

print('Similarities: ', dataBST[i-1], ' ', dataBST[i] , '=', end=' ')

print(round(final\_result,6))

endQueryBST = time.time()

endTotalBST = time.time()

print()

print('Running time for binary search tree construction: ' ,round(endBuildBST- startBuildBST, 5))

print('Running time for binary search tree query processing: ', round(endQueryBST-startQueryBST, 4))

print('Total runtime: ', round(endTotalBST-startTotalBST, 3))

##############################

if choice == 2:

startTotalHash = time.time()

B = HashTableC(hash\_size)

startBuildHash = time.time()

with open ('/Users/joaquin/UTEP/Class/cs2302/lab05\_txt/glove.6B.50d.txt', 'r', encoding='utf-8') as txt\_file:

for l in txt\_file:

flag = False

allSplit = l.split(' ')

a = allSplit[0]

b = np.array(allSplit[1:], dtype = np.float)

#Checks if first string, first char is a letter

for j in all\_letters:

if j == a[0]:

flag = True

#We found a correct 'KEY' to be inserted

#If word, insert to BST and append to keyBST

if flag == True:

keyHash.append(a)

temp = [a,b]

if B.num\_size/len(B.item) > 1:

P = None

hash\_size = (hash\_size \* 2) + 1

P = HashTableC(hash\_size)

#checks every Bucket

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

#checks each node in the bucket

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

#H.item[i][j] is list and keyword

InsertChain( P, B.item[i][j] )

P.num\_size += 1

index = HashFunc(P, temp[0])

P.item[index].append(temp)

B = None

B = P

InsertChain(B,temp)

else:

InsertChain(B,temp)

endBuildHash = time.time()

print('Choice: ', choice)

print('\nBuilding hash table with chaining')

print('\nHash table stats:\n')

print('number of elements: ', B.num\_size)

print('Initial table size: ', initial\_size )

print('Final table size: ', len(B.item))

print('Load factor: ', round(B.num\_size/len(B.item) , 4) )

print('Percentage of empty lists: ', Percentage\_Empty(B))

for i in B.item:

dev1.append(int(len(i)))

print('Standard deviation of the lengths of the lists: ', round(np.std(dev1),4) )

print('Build time: ', round(endBuildHash-startBuildHash, 4))

print('\nReading word file to determine similarities \n')

WriteToTxtHash(keyHash, sim\_length)

#read 2nd file and save strings in array

dataHash = ReadGetListHash(dataHash)

startQueryHash = time.time()

for i in range(1,len(dataHash), 2):

v1 = GetVecHash( B,dataHash[i-1] )

v2 = GetVecHash( B,dataHash[i] )

dot\_p = v1\*v2

dot\_p = sum(dot\_p)

m0 = 0

m1 = 0

for j in v1:

m0 += j\*\*2

for j in v2:

m1 += j\*\*2

m0 = math.sqrt(m0 )

m1 = math.sqrt(m1 )

final\_result = dot\_p / (m0\*m1)

print('Similarities: ', dataHash[i-1], ' ', dataHash[i] , '=', end=' ')

print(round(final\_result,6))

endQueryHash = time.time()

endTotalHash = time.time()

print('\nBuild time: ', round(endBuildHash-startBuildHash, 4))

print('Running time for Hash chain query processing: ', round(endQueryHash-startQueryHash, 4))

print('Total runtime: ', round(endTotalHash-startTotalHash, 3))

print(len(B.item))

**Standards of Conduct and Academic Dishonesty**

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.

* Joaquin Hidalgo