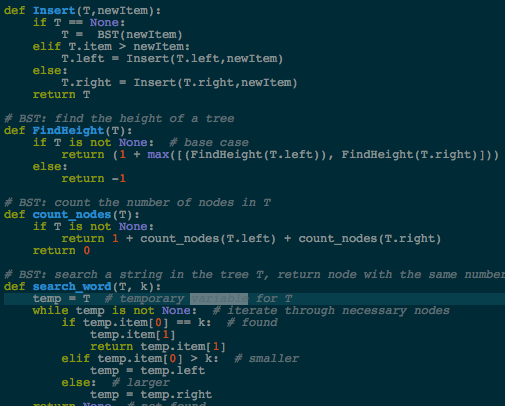
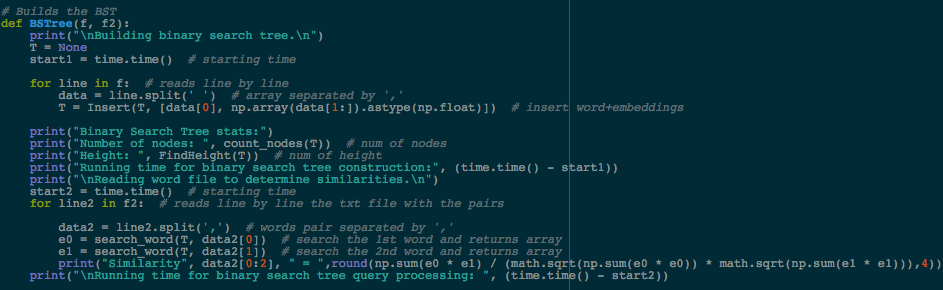
**Lab #5 Report**

**Introduction:**

In this lab we understood in a brief way how Natural Language Processing (NLP) works in apps or tools such as Siri and Alexa when they use hash tables to efficiently retrieve the embeddings given their corresponding words. The task for this lab was to compare the running times of two implementations of tables to retrieve word embeddings to enable the comparison of two words. One table will be provided with binary search tree and other with the use of hash tables with chaining. The main purpose of this lab is to practice and learn how to use binary search tree, hash tables, and also to read txt. files.

**BINARY SEARCH TREE’S METHODS**

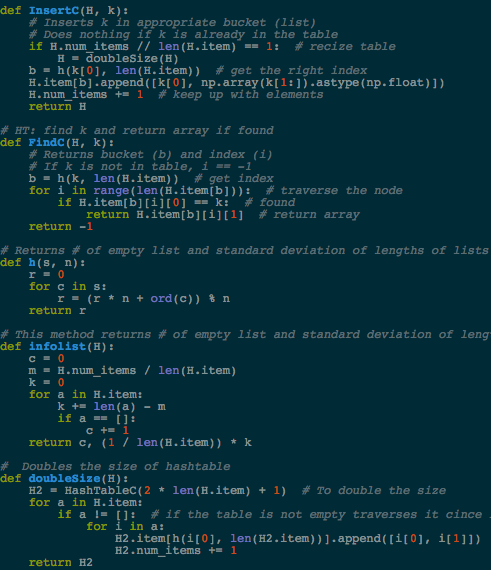
* As pre-methods I used the normal *insert()* method to insert the items in the tree.
* Secondly, I created a method *find\_height()* which is in charge of returning how many elements are in the actual tree. For this task we had to check if the tree is not empty, and if this happens we recursively add 1 plus the maximum between the left and the right side. If the tree is empty then it just returns -1.
* Then I created a method *count\_nodes****(****)* that tells how many elements the tree contains. Here, if the tree is not empty then recursively we add 1 plus the items in the left and the items in the right.
* Also, I did a method *search\_word()* which is in charge of searching the word and returning the array. In a while loop until the temporary variable is not null, if the item at the index 0 is the same as the key then it returns the temporary at an index of 1. if the item at index 0 is greater than k then the temporary goes to the left, otherwise it goes to the right.
* Finally. I made a method *BSTree()* that is the one that computes all questions and include all pre-methods. This method displays how many nodes are in the tree, the height, and also compute the time it took to construct the tree and the time it took for the query processing. Also, to find the similarities between the words, in a for loop we make the calculation that way each word can have its own comparison.

insert,FindHeight = O(n)

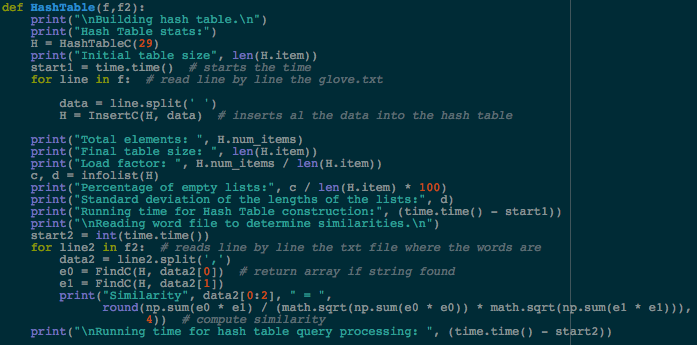
BSTree() = O(n) countNodes() = O(n^2) search\_word() = O(n)

**HASH TABLE WITH CHAINING:**

* For pre-methods I have the *InsertC()* that insert the items into the table, the method  *FindC()* that finds if an element is in the table, and also the method *h()* which is in charge of getting the number of comparison for each word, I have the method *infolist()* that ﻿returns the number of empty lists and standard deviation of lengths of lists.
* I have a method *doubleSize()* which doubles the size of the table in case that it needs more space to insert new items. For that, I first created another table in which its length is the double than the original. Then, in a for loop, if the table is not empty then it traverses the table and it appends the elements into the new table and finally returns the new table.
* Finally, I have a method *HashTable()* that displays all final calculations. First, in a for loop, the code reads the txt. that has all vectors and comparisons. Then we compute the load factor by the num\_item over the length of the table. Then we find the percentage of the empty lists by dividing the info list over the length of the table times 100. After it we take the time it took to create the table and the standard deviation of the lengths of the lists. Finally, we find the similarities between the words by reading the txt. where the words to compare are located. At the end we just get the time it took for the hash table query processing.

findC(), h(), infoList() = O(n)

insertC() = O(1) HashTable() = O(n) doublesize() = O(n^2)

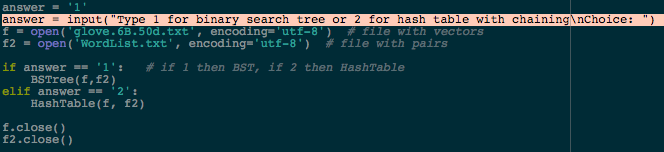


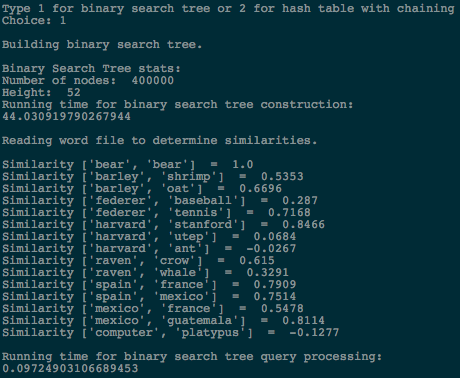
**“MAIN” METHOD:**

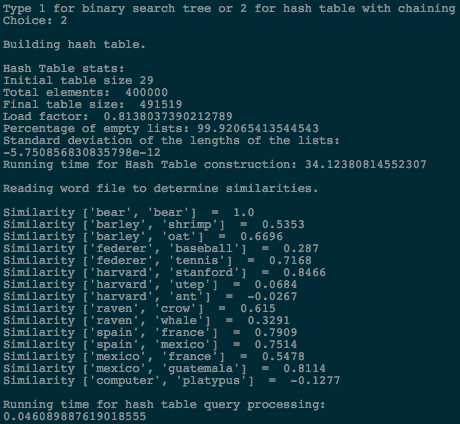
Finally, in our “main method” we ask the user to pick a number between 1 and 2, if the user selects 1 the binary search tree, otherwise will run the hash table method.

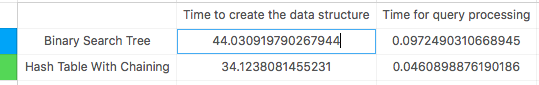
**Conclusion:**

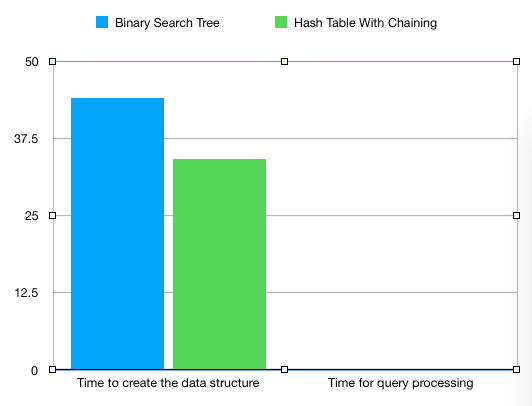
I think this lab assignment was not the harder one but was not the easier one. it was confusing since we had to deal with vectors and standard deviation. But in other point of view I think it was a good because I learnt how to read a file from the python code and also I learnt how to deal with hash tables.



**Output: **

****

****

****

**﻿**

import numpy as np

import time

import math

#------------------------- PRE-METHODS -----------------

# \*\*\*\* BST:

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

# BST: find the height of a tree

def FindHeight(T):

if T is not None: # base case

return (1 + max([(FindHeight(T.left)), FindHeight(T.right)])) # 1 + (the higher number)

else:

return -1

# BST: count the number of nodes in T

def count\_nodes(T):

if T is not None:

return 1 + count\_nodes(T.left) + count\_nodes(T.right)

return 0

# BST: search a string in the tree T, return node with the same number if it was found, None if not found

def search\_word(T, k):

temp = T # temporary variable for T

while temp is not None: # iterate through necessary nodes

if temp.item[0] == k: # found

temp.item[1]

return temp.item[1]

elif temp.item[0] > k: # smaller

temp = temp.left

else: # larger

temp = temp.right

return None # not found

# \*\*\* HASHTABLE:

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.num\_items = 0

for i in range(size):

self.item.append([])

def InsertC(H, k):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

if H.num\_items // len(H.item) == 1: # recize table

H = doubleSize(H)

b = h(k[0], len(H.item)) # get the right index

H.item[b].append([k[0], np.array(k[1:]).astype(np.float)])

H.num\_items += 1 # keep up with elements

return H

# HT: find k and return array if found

def FindC(H, k):

# Returns bucket (b) and index (i)

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

b = h(k, len(H.item)) # get index

for i in range(len(H.item[b])): # traverse the node

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

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

return -1

# Returns # of empty list and standard deviation of lengths of lists

def h(s, n):

r = 0

for c in s:

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

return r

# This method returns # of empty list and standard deviation of lengths of lists

def infolist(H):

c = 0

m = H.num\_items / len(H.item)

k = 0

for a in H.item:

k += len(a) - m

if a == []:

c += 1

return c, (1 / len(H.item)) \* k

# Doubles the size of hashtable

def doubleSize(H):

H2 = HashTableC(2 \* len(H.item) + 1) # To double the size

for a in H.item:

if a != []: # if the table is not empty traverses it cince it's chaining

for i in a:

H2.item[h(i[0], len(H2.item))].append([i[0], i[1]])

H2.num\_items += 1

return H2

#------------------------ METHODS FOR LAB ---------------

# Builds the BST

def BSTree(f, f2):

print("\nBuilding binary search tree.\n")

T = None

start1 = time.time() # starting time

for line in f: # reads line by line

data = line.split(' ') # array separated by ' '

T = Insert(T, [data[0], np.array(data[1:]).astype(np.float)]) # insert word+embeddings

print("Binary Search Tree stats:")

print("Number of nodes: ", count\_nodes(T)) # num of nodes

print("Height: ", FindHeight(T)) # num of height

print("Running time for binary search tree construction:", (time.time() - start1))

print("\nReading word file to determine similarities.\n")

start2 = time.time() # starting time

for line2 in f2: # reads line by line the txt file with the pairs

data2 = line2.split(',') # words pair separated by ','

e0 = search\_word(T, data2[0]) # search the 1st word and returns array

e1 = search\_word(T, data2[1]) # search the 2nd word and returns array

print("Similarity", data2[0:2], " = ",round(np.sum(e0 \* e1) / (math.sqrt(np.sum(e0 \* e0)) \* math.sqrt(np.sum(e1 \* e1))),4)) # compute the similarity

print("\nRunning time for binary search tree query processing: ", (time.time() - start2))

# Builds the HashTable

def HashTable(f,f2):

print("\nBuilding hash table.\n")

print("Hash Table stats:")

H = HashTableC(29)

print("Initial table size", len(H.item))

start1 = time.time() # starts the time

for line in f: # read line by line the glove.txt

data = line.split(' ')

H = InsertC(H, data) # inserts al the data into the hash table

print("Total elements: ", H.num\_items)

print("Final table size: ", len(H.item))

print("Load factor: ", H.num\_items / len(H.item))

c, d = infolist(H)

print("Percentage of empty lists:", c / len(H.item) \* 100)

print("Standard deviation of the lengths of the lists:", d)

print("Running time for Hash Table construction:", (time.time() - start1))

print("\nReading word file to determine similarities.\n")

start2 = int(time.time())

for line2 in f2: # reads line by line the txt file where the words are

data2 = line2.split(',')

e0 = FindC(H, data2[0]) # return array if string found

e1 = FindC(H, data2[1])

print("Similarity", data2[0:2], " = ",

round(np.sum(e0 \* e1) / (math.sqrt(np.sum(e0 \* e0)) \* math.sqrt(np.sum(e1 \* e1))),

4)) # compute similarity

print("\nRunning time for hash table query processing: ", (time.time() - start2))

# main method

answer = '1'

answer = input("Type 1 for binary search tree or 2 for hash table with chaining\nChoice: ")

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

f2 = open('WordList.txt', encoding='utf-8') # file with pairs

if answer == '1': # if 1 then BST, if 2 then HashTable

BSTree(f,f2)

elif answer == '2':

HashTable(f, f2)

f.close()

f2.close()

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.

* David A. Davis