**Lab Report 5: Hash Tables with Chaining and Linear Probing**

**I. Introduction**

This lab consisted on storing words and its embeddings, read from a file, on Hash Tables with Chaining and Linear Probing. After the words were stored, another file was read that contained pairs of words to be compared with their embeddings and returned how similar they were. There had to be methods to read the file into the respective table that the user chose and another to read the word file and return the similarities. The classes for the HTC and HTLP had to handle receiving the object WordEmbedding that in itself contained the word and a numpy array of the embedding.

**II. Proposed solution design and implementation**

The first step was creating the user interface in which it could choose the way that the words were going to be stored: Linear Probing, Chaining, or the previous trees from lab 4, plus the exit option. The program also asks the user to input the size of the table if they choose a hash table

After this was completed, my next step was to create a function that would read the file into the Tables. Just like in lab 4, this function read each line and stored the information on a WordEmbedding object, then on the table as it read. The Hash Table Chain class was modified to be able to receive a WordEmbedding and sort the words according to what the user chose. This was made with many if statements that chose which method to call for insertion.

The next step was to make the function that would read into a Hash Table with Linear Probing. This function was almost identical save to the call to insert the node with WordEmbedding that was for the htlp. The HTLP class was also modified to handle storing tobject types instead of integers. I also had to modify how it compared when a space was empty or being occupied by a word embedding.

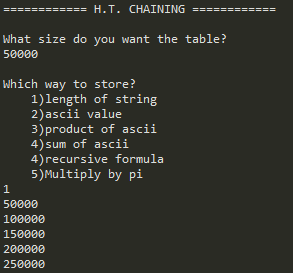
After these functions were successful counting the time that a reading and storing took was added. To make sure the tables were stored correctly, an optional message is sent to print the tables.

The last part was finding the similarities. This was solved with the same function used in lab 4 only adapted to call a method in the corresponding hash table and sending an extra parameter that has the choice of storage. The call to return the embedding of the word was with a function called embedding that looked for the position of the word in the table and returned the embedding numpy array once it found the word. After each embedding was returned for the pair of words, the formula given was applied to get the result of similarity. Time was also compared here and printed after the results were shown.

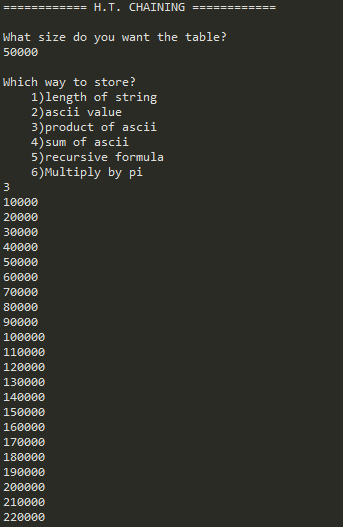
The program loops if the user wants to repeat any step or try the other tree function.

**III. Experimental results**

Reading HTC String:

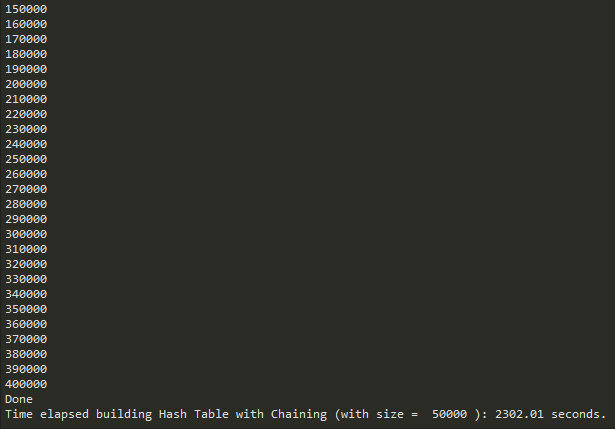
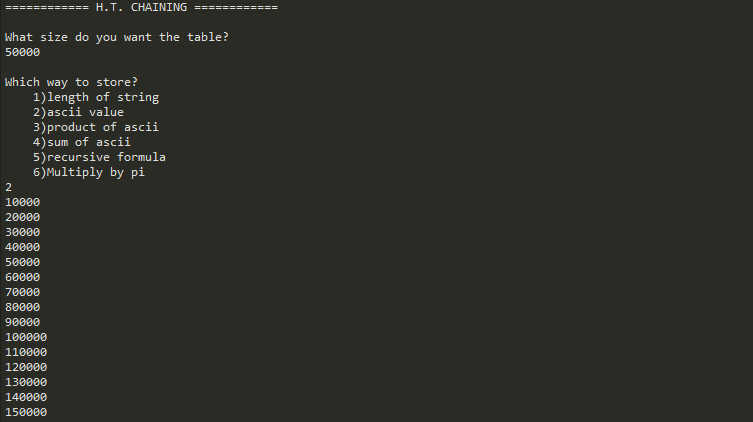
 **Too long to wait… (about an hour)**

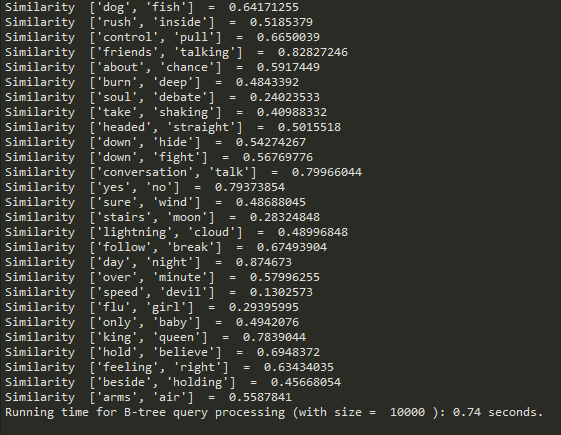
**By product of ASCII:**

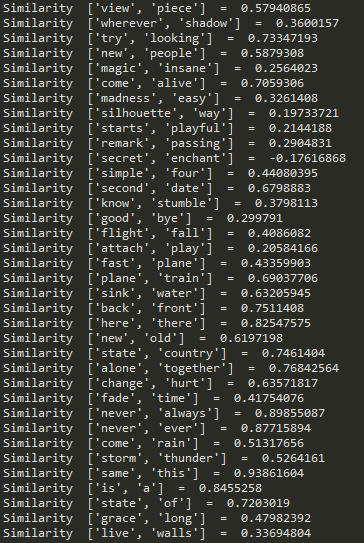
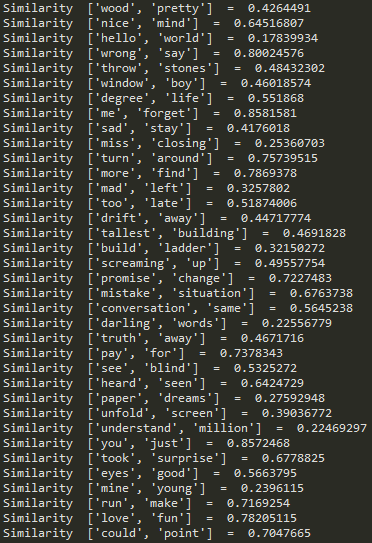
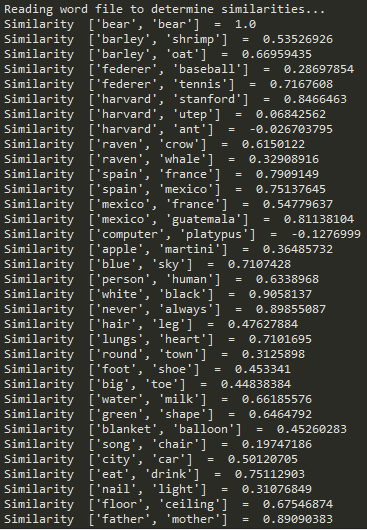
****

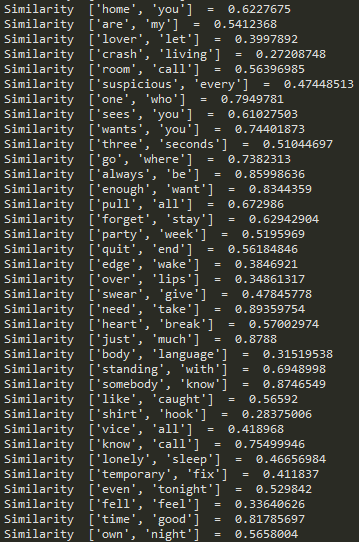
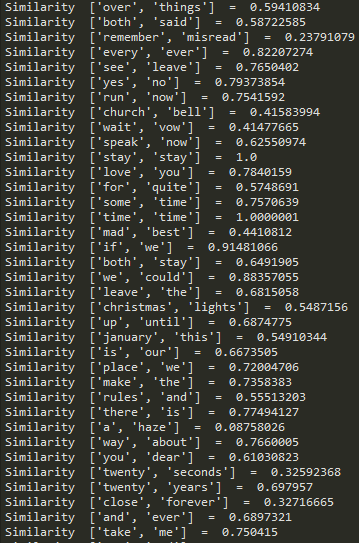
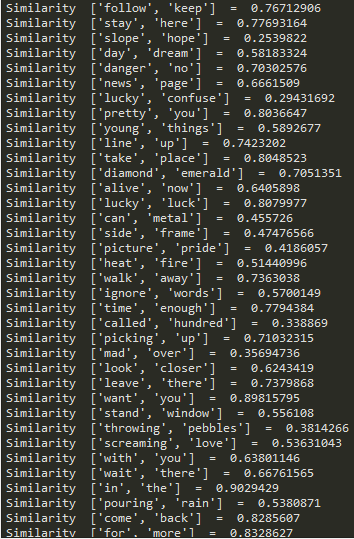
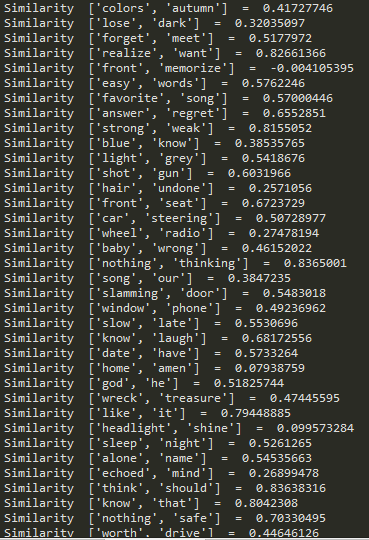
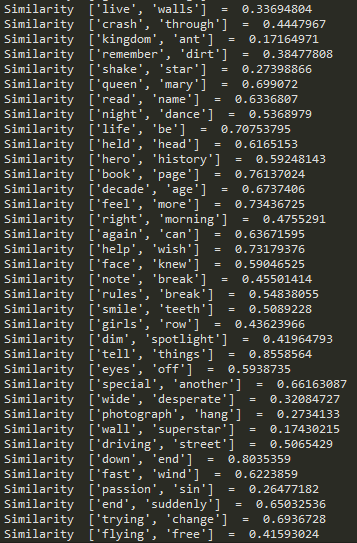
** 390.08 sec**

**By Ascii value:**

** 2302.01 sec**

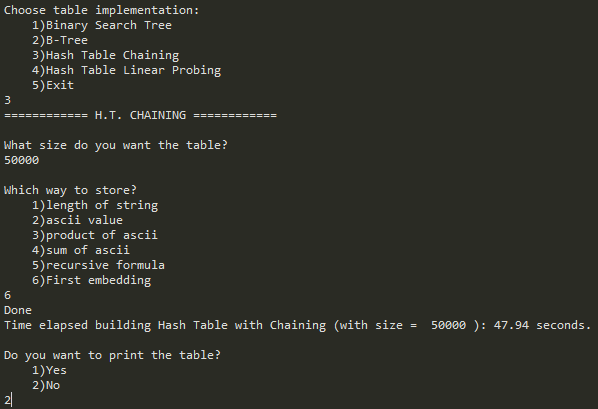
 **0.74 sec**



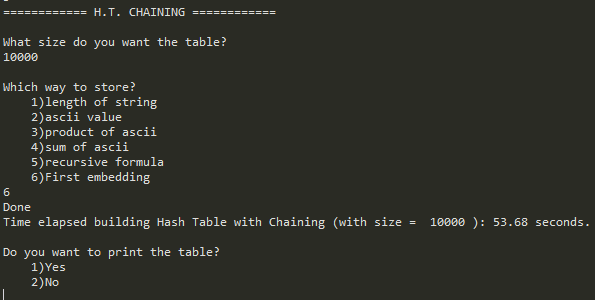


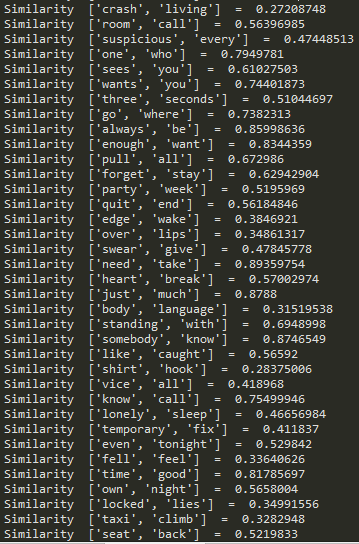
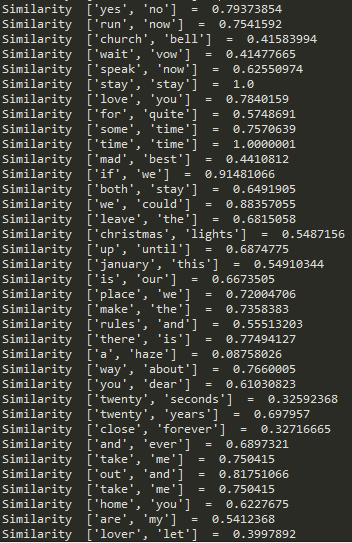
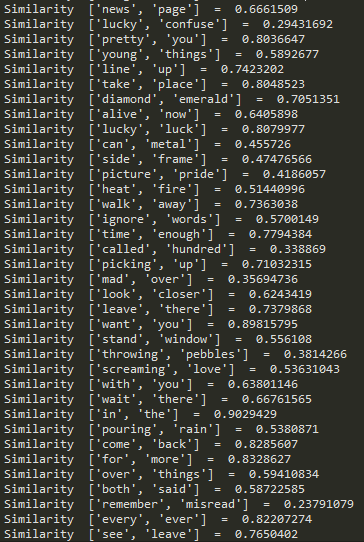
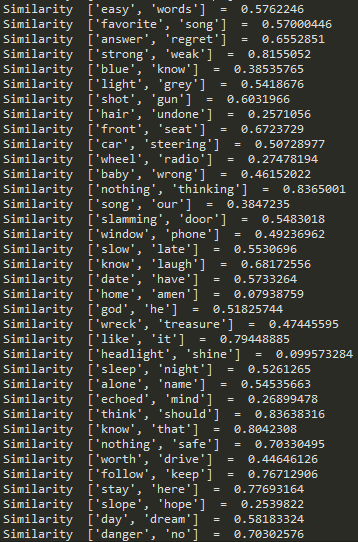
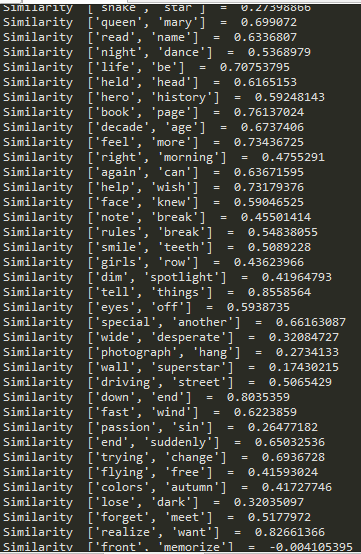
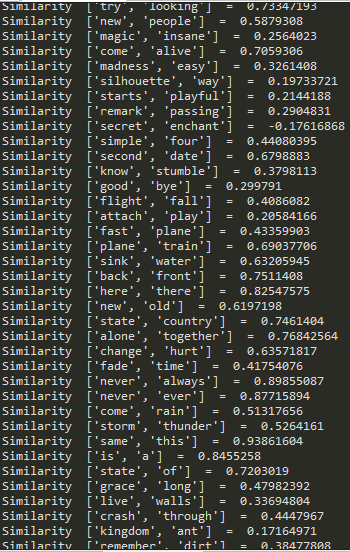
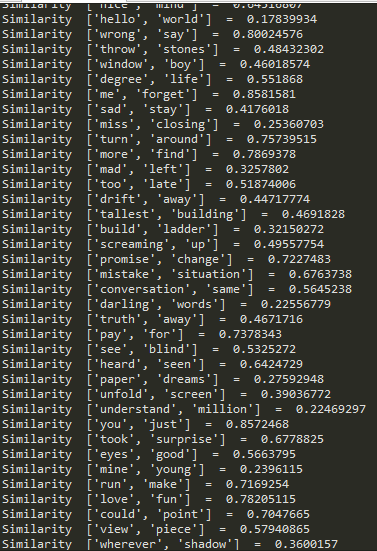
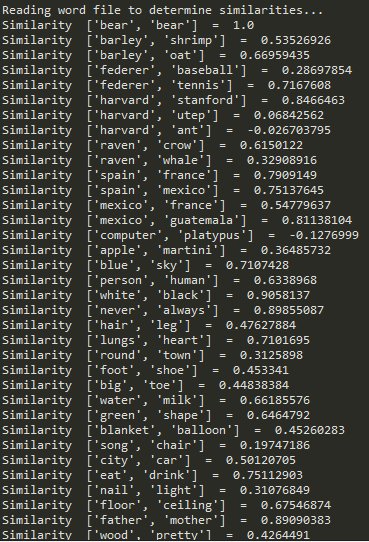
**Reading by embedding**

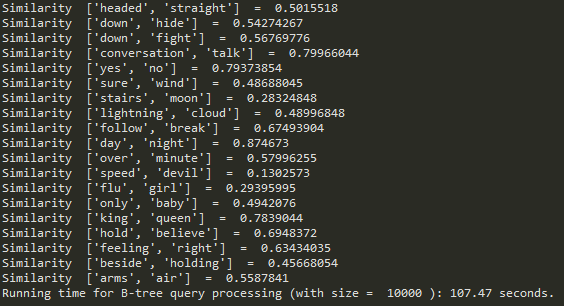
**Size 50000**

**** 47.94 sec

Size = 100000

 53.68



 **107.47sec**HTLC was extremely slow.  
**Table:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | **Length of String** | **ASCII value** | **Product of ASCII** | **Sum of ASCII** | **Recursive Formula** | **Embedding** |
| HTC 10000 | 1.88 | 1.27 | 0.77 | 0.71 | 11.28 | 0.94 |
| HTC 50000 | 2.42 | 1.69 | 1.17 | 1.13 | 11.17 | 1.31 |
| HTC 1000 | 2.09 | 1.23 | 0.79 | 1.28 | 10.81 | 0.95 |
| HTC 5000 | 1.97 | 1.34 | 0.77 | 0.70 | 10.70 | 0.94 |
| HTLP 1000 | 14.17 | 15.60 | 22.55 | 34.82 | 117.64 | 258.91 |
| HTLP 5000 | 56.57 | 63.60 | 67.97 | 132.73 | 341.00 | 743.21 |
| Similarities HTC | 0.84 | 0.74 | 0.60 | 0.54 | 5.12 | 50.04 |
| HTLP | 1.24 | 1.22 | 1.19 | 1.03 | 9.04 | 222.12 |
| BTREE | 1.09 | 0.23 |  |  |  |  |
| BST | 1.25 | 0.19 |  |  |  |  |

**\*\*\*LIMITING THE WORDS TO 10000**

**IV. Conclusions**

I learned that hashing can be very effective if the correct formula is found. For this lab it was very noticeable that trees had a better time complexity than hash tables. From both hash tables, chaining was better. My guess is that for linear hash tables, collisions are more expensive and more time consuming than for chaining. I absolutely learned that the more wide the number result for the hashing function is better. Storing by string length was a horrible idea because the limit is small. Storing by ascii value is better, but still extremely few for the amount of words that this lab took into account. Product was useful too, but by far the most effective from the assigned was de sum of de ascii. This gave more different possibilities of numbers and I believe this is the reason. The size of the table also influenced the velocity. The size could minimize the number of collisions that could happen and that is why the bigger the size te faster it got, but more memory was occupied.

The new function I created was very effective for storing in the table, because it used the first embedding number which was almost unique. Yet since for finding the word again our input is the word and not the word embedding, finding it was impossible using a hash table function and linear searching had to be made.

**V. Appendix – Source code**

# -\*- coding: utf-8 -\*-

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 5 - Hash Tables

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 11/01/2019

Program: Stores into hash tables linearly or by chaining

"""

import hash\_linear as htlp

import hash\_chain as htc

import bst

import btree

import WordEmbedding

import numpy as np

import time

#files to read

embed\_file = "glove.6B.50d.txt" #embeddings file

word\_file = "words.txt" #words with similarities file

#reading into a Hash Table with Chaining

def readIntoHashChain(size, choice):

h = htc.HashTableChain(size)

if choice == "1":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertString(n)

content = ef.readline()

elif choice == "2":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertAscii(n)

content = ef.readline()

elif choice == "3":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertPAscii(n)

content = ef.readline()

elif choice == "4":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertSAscii(n)

content = ef.readline()

elif choice == "5":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertRecursive(n)

elif choice == "6":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertFE(n)

content = ef.readline()

print("Done")

return h

#reading into a Hash Table with Linear Probing

def readIntoHashLP(size, choice):

h = htlp.HashTableLP(size)

if choice == "1":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertS(n)

content = ef.readline()

elif choice == "2":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertAscii(n)

content = ef.readline()

elif choice == "3":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertPAscii(n)

content = ef.readline()

elif choice == "4":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertSAscii(n)

content = ef.readline()

elif choice == "5":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertRecursive(n)

content = ef.readline()

elif choice == "6":

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

cnt = 0

for i in range(10000):

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

h.insertFE(n)

cnt += 1

if cnt % 10000 == 0:

print(cnt)

content = ef.readline()

print("Done")

return h

#similarities HT Chaining

def similaritiesHTC(T,c):

with open(word\_file) as wf:

content = wf.readline().split()

#reads every line in the file

while content:

#separates words

word1 = T.Embedding(T,content[0],c)

word2 = T.Embedding(T,content[1],c)

#returns similarities with similarities function

sim = (np.dot(word1,word2))/(np.linalg.norm(word1)\*np.linalg.norm(word2))

#Prints values found

print("Similarity ", content, " = ", sim)

content = wf.readline().split()

return

#similarities HT Linear Probing

def similaritiesHTLP(T,c):

with open(word\_file) as wf:

content = wf.readline().split()

#reads every line in the file

while content:

#separates words

word1 = T.Embedding(T,content[0],c)

word2 = T.Embedding(T,content[1],c)

#returns similarities with similarities function

sim = (np.dot(word1,word2))/(np.linalg.norm(word1)\*np.linalg.norm(word2))

#Prints values found

print("Similarity ", content, " = ", sim)

content = wf.readline().split()

return

#reading into a binary search tree

def readIntoBST():

tree = None

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

while content:

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

tree = bst.Insert(tree, n)

content = ef.readline()

print("Done")

return tree

#reading into a BTree

def readIntoBTree(max):

tree = btree.BTree([],max\_data=max)

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

while content:

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on btree

btree.Insert(tree, n)

btree.Print

content = ef.readline()

print("Done")

return tree

#prints bst in ascending order (inorder)

def printTree(T):

if T.left != None:

printTree(T.left)

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

if T.right != None:

printTree(T.right)

#finds similarities between words stored in a BST

def similaritiesBST(T):

with open(word\_file) as wf:

content = wf.readline().split()

#reads every line in the file

while content:

#separates words

word1 = bst.Embedding(T,content[0])

word2 = bst.Embedding(T,content[1])

#returns similarities with similarities function

sim = (np.dot(word1,word2))/(np.linalg.norm(word1)\*np.linalg.norm(word2))

#Prints values found

print("Similarity ", content, " = ", sim)

content = wf.readline().split()

return

#finds similarities between words stored in a BTree

def similaritiesBTree(T):

with open(word\_file) as wf:

content = wf.readline().split()

#reads every line in the file

while content:

word1 = btree.Embedding(T,content[0])

word2 = btree.Embedding(T,content[1])

#returns similarities with similarities function

sim = (np.dot(word1,word2))/(np.linalg.norm(word1)\*np.linalg.norm(word2))

#prints values found

print("Similarity ", content, " = ", sim)

content = wf.readline().split()

return

stay = True

#looping until exit

while(stay):

option = input("Choose table implementation:\n\t1)Binary Search Tree\n\t2)B-Tree\n\t3)Hash Table Chaining\n\t4)Hash Table Linear Probing\n\t5)Exit\n")

#BST option

if (option == "1"):

print("========= BINARY SEARCH TREE =========")

#call to method and time

start = time.time()

tree = readIntoBST()

end = time.time()

#results

print("Number of nodes: ", bst.countNodes(tree))

print("Height: ", bst.Height(tree))

print("Time elapsed building BST: "+"{:.2f}".format(end-start) + " seconds.")

#display tree option

if(input("Do you want to print the stored tree?\n\t1)Yes\n\t2)No\n") == "1"):

printTree(tree)

#similarities

print("Reading word file to determine similarities...")

start = time.time()

similaritiesBST(tree)

end = time.time()

print("Running time for binary search tree query processing: "+"{:.2f}".format(end-start) + " seconds.")

#BTree option

elif (option == "2"):

print("=============== B-TREE ===============")

max = int(input("Input the maximum number of items to store in a node:"))

#call to method and time

start = time.time()

tree = readIntoBTree(max)

end = time.time()

#results

print("Number of nodes: ", btree.countNodes(tree))

print("Height: ", btree.Height(tree))

print("Time elapsed building BTree (with max\_items = ", max, "): "+"{:.2f}".format(end-start) + " seconds.")

#display tree option

if(input("Do you want to print the stored tree?\n\t1)Yes\n\t2)No\n") == "1"):

btree.PrintD(tree, "")

#similarities

print("Reading word file to determine similarities...")

start = time.time()

similaritiesBTree(tree)

end = time.time()

print("Running time for B-tree query processing (with max\_items = ", max, "): "+"{:.2f}".format(end-start) + " seconds.")

elif (option == "3"):

print("============ H.T. CHAINING ============")

size = int(input("What size do you want the table?\n"))

choice = (input("Which way to store?\n\t1)length of string\n\t2)ascii value\n\t3)product of ascii\n\t4)sum of ascii\n\t5)recursive formula\n\t6)First embedding\n"))

#call to method and time

start = time.time()

table = readIntoHashChain(size, choice)

end = time.time()

print("Time elapsed building Hash Table with Chaining (with size = ", size, "): "+"{:.2f}".format(end-start) + " seconds.")

#display table option

if(input("Do you want to print the table?\n\t1)Yes\n\t2)No\n") == "1"):

table.print\_table()

#similarities

print("Reading word file to determine similarities...")

start = time.time()

similaritiesHTC(table, choice)

end = time.time()

print("Running time for HT query processing (with size = ", size, "): "+"{:.2f}".format(end-start) + " seconds.")

elif (option == "4"):

print("========= H.T. LINEAR PROBING =========")

size = int(input("What size do you want the table?\n"))

choice = (input("Which way to store?\n\t1)length of string\n\t2)ascii value\n\t3)product of ascii\n\t4)sum of ascii\n\t5)recursive formula\n\t6)First embedding\n"))

#call to method and time

start = time.time()

table = readIntoHashLP(size, choice)

end = time.time()

print("Time elapsed building Hash Table with Linear Probing (with size = ", size, "): "+"{:.2f}".format(end-start) + " seconds.")

#display table option

if(input("Do you want to print the table?\n\t1)Yes\n\t2)No\n") == "1"):

table.print\_table()

#similarities

print("Reading word file to determine similarities...")

start = time.time()

similaritiesHTLP(table,choice)

end = time.time()

print("Running time for HT query processing (with size = ", size, "): "+"{:.2f}".format(end-start) + " seconds.")

#Exit

elif (option == "5"):

print("Thank you for using this program! Goodbye!")

stay = False

else:

print("Choose 1, 2, 3, 4 or 5.")

**------HTC------**

# -\*- coding: utf-8 -\*-

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 5 - Hash Table Chain

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 11/01/2019

Program: --------------------

"""

class HashTableChain(object):

# Builds a hash table of size 'size'

# Item is a list of (initially empty) lists

# Constructor

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

self.bucket = [[] for i in range(size)]

def hS(self,k):

return len(k.word)%len(self.bucket)

def insertString(self,k):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

b = self.hS(k)

if not k in self.bucket[b]:

new = [k]

for i in self.bucket[b]:

new.append(i) #insert new item at the beginning

self.bucket[b]=new

#self.bucket[b].append(k) #Insert new item at the end

def hA(self,k):

return ord(k.word[0])%len(self.bucket)

def insertAscii(self,k):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

b = self.hA(k)

if not k in self.bucket[b]:

new = [k]

for i in self.bucket[b]:

new.append(i) #insert new item at the beginning

self.bucket[b]=new

#self.bucket[b].append(k) #Insert new item at the end

def hPA(self,k):

return (ord(k.word[0])\*ord(k.word[-1]))%len(self.bucket)

def insertPAscii(self,k):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

b = self.hPA(k)

if not k in self.bucket[b]:

new = [k]

for i in self.bucket[b]:

new.append(i) #insert new item at the beginning

self.bucket[b]=new

#self.bucket[b].append(k) #Insert new item at the end

def hSA(self,k):

sum=0

for i in k:

sum += ord(i)

return sum%len(self.bucket)

def insertSAscii(self,k):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

b = self.hSA(k.word)

if not k in self.bucket[b]:

new = [k]

for i in self.bucket[b]:

new.append(i) #insert new item at the beginning

self.bucket[b]=new

#self.bucket[b].append(k) #Insert new item at the end

def hR(self,k):

if len(k) == 0:

return 1

return (ord(k[0]) + 255\*self.hR(k[1:]))%len(self.bucket)

def insertRecursive(self,k):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

b = self.hR(k.word)

if not k in self.bucket[b]:

new = [k]

for i in self.bucket[b]:

new.append(i) #insert new item at the beginning

self.bucket[b]=new

#self.bucket[b].append(k) #Insert new item at the end

def hFE(self,k):

return int(round(k.emb[0]\*10000)%len(self.bucket))

def lookFe(self,k):

for i in self.bucket:

for j in i:

if j.word == k:

return j.emb

return None

def insertFE(self,k):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

b = self.hFE(k)

if not k in self.bucket[b]:

new = [k]

for i in self.bucket[b]:

new.append(i) #insert new item at the beginning

self.bucket[b]=new

#self.bucket[b].append(k) #Insert new item at the end

def find(self,k):

# Returns bucket (b) and index (i)

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

b = self.h(k)

try:

i = self.bucket[b].index(k)

except:

i = -1

if i > -1:

self.delete(k)

self.insert(k)

return b, i

def print\_table(self):

print('Table contents:')

for b in self.bucket:

print("[", end='')

for a in b:

print(a.word, end = ', ')

print("]")

def Embedding(self,T,k,c):

# Returns embedding of the word k, or None if k is not in the table

if c == "1":

b = self.hS(k) #bucket where k must be

for i in self.bucket[b]: #looks for index in bucket where k is

if i.word == k:

return i.emb

elif c == "2":

b = self.hA()

for i in self.bucket[b]:

if i.word == k:

return i.emb

elif c == "3":

b = self.hPA(k)

for i in self.bucket[b]:

if i.word == k:

return i.emb

elif c == "4":

b = self.hSA(k)

for i in self.bucket[b]:

if i.word == k:

return i.emb

elif c == "5":

b = self.hR(k)

for i in self.bucket[b]:

if i.word == k:

return i.emb

elif c == "6":

return self.lookFe(k)

return None #k was not found

**------HTLP------**

# -\*- coding: utf-8 -\*-

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 5 - Hash Tables Linear Probing

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 11/01/2019

Program: --------------------

"""

import numpy as np

class HashTableLP(object):

# Builds a hash table of size 'size', initilizes items to -1 (which means empty)

# Constructor

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

self.item = np.zeros(size,dtype=np.object)-1

def insertS(self,k):

# Inserts k in table unless table is full

# Returns the position of k in self, or -1 if k could not be inserted

for i in range(len(self.item)): #Despite for loop, running time should be constant for table with low load factor

pos = (len(k.word)+i)%len(self.item)

if isinstance(self.item[pos], int):

self.item[pos] = k

return pos

return -1

def insertAscii(self,k):

# Inserts k in table unless table is full

# Returns the position of k in self, or -1 if k could not be inserted

for i in range(len(self.item)): #Despite for loop, running time should be constant for table with low load factor

pos = (ord(k.word[0])+i)%len(self.item)

if isinstance(self.item[pos], int):

self.item[pos] = k

return pos

return -1

def hPA(self,k, i):

return (ord(k.word[0])\*ord(k.word[-1])+i)%len(self.item)

def insertPAscii(self,k):

# Inserts k in table unless table is full

# Returns the position of k in self, or -1 if k could not be inserted

for i in range(len(self.item)): #Despite for loop, running time should be constant for table with low load factor

pos = self.hPA(k,i)

if isinstance(self.item[pos], int):

self.item[pos] = k

return pos

return -1

def hSA(self,k,j):

sum=0

for i in k:

sum += ord(i)

return (sum+j)%len(self.item)

def insertSAscii(self,k):

# Inserts k in table unless table is full

# Returns the position of k in self, or -1 if k could not be inserted

for i in range(len(self.item)): #Despite for loop, running time should be constant for table with low load factor

pos = self.hSA(k.word,i)

if isinstance(self.item[pos], int):

self.item[pos] = k

return pos

return -1

def hR(self,k,i):

if len(k) == 0:

return 1

return ((ord(k[0]) + 255\*self.hR(k[1:],i))+i)%len(self.item)

def insertRecursive(self,k):

# Inserts k in table unless table is full

# Returns the position of k in self, or -1 if k could not be inserted

for i in range(len(self.item)): #Despite for loop, running time should be constant for table with low load factor

pos = self.hR(k.word,i)

if isinstance(self.item[pos], int):

self.item[pos] = k

return pos

return -1

def hFE(self,k,i):

return int((round(k.emb[0]\*10000)+i)%len(self.item))

def lookFe(self,k):

for i in self.item:

if i.word == k:

return i.emb

return None

def insertFE(self,k):

# Inserts k in table unless table is full

# Returns the position of k in self, or -1 if k could not be inserted

for i in range(len(self.item)): #Despite for loop, running time should be constant for table with low load factor

pos = self.hFE(k,i)

if isinstance(self.item[pos], int):

self.item[pos] = k

return pos

return -1

def find(self,k):

# Returns the position of k in table, or -1 if k is not in the table

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

pos = self.h(k+i)

if self.item[pos] == k:

return pos

if self.item[pos] == -1:

return -1

return -1

def delete(self,k):

# Deletes k from table. It returns the position where k was, or -1 if k was not in the table

# Sets table item where k was to -2 (which means deleted)

f = self.find(k)

if f >=0:

self.item[f] = -2

return f

def h(self,k):

return k%len(self.item)

def print\_table(self):

print('Table contents:')

for i in self.item:

print(i.word)

def load\_factor(self):

length =0

for i in self.item:

if i > 0:

length += 1

return length/11

def collisions(self):

collisions = 0

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

if self.h(self.item[i]) > 0 and self.h(self.item[i]) != i:

collisions += 1

print(self.item[i])

print( collisions)

return collisions

def Embedding(self,T,k,c):

# Returns embedding of the word k, or None if k is not in the table

if c == "1":

pos = (len(k.word))%len(self.item)#position where k must be

while isinstance(self.item[pos], object): #checks next places for k

if self.item[pos].word == k:

return self.item[pos].emb

pos+=1

return None #k was not found

elif c == "2":

pos = (ord(k.word[0]))%len(self.item)

while isinstance(self.item[pos], object):

if self.item[pos].word == k:

return self.item[pos].emb

pos+=1

return None

elif c == "3":

pos = self.hPA(k,0)

while isinstance(self.item[pos], object):

if self.item[pos].word == k:

return self.item[pos].emb

pos+=1

return None

elif c == "4":

pos = self.hSA(k,0)

while isinstance(self.item[pos], object):

if self.item[pos].word == k:

return self.item[pos].emb

pos+=1

return None

elif c == "5":

pos = self.hR(k,0)

while isinstance(self.item[pos], object):

if self.item[pos].word == k:

return self.item[pos].emb

pos+=1

return None

elif c == "6":

pos = self.hFE(k,0)

while isinstance(self.item[pos], object):

if self.item[pos].word == k:

return self.item[pos].emb

pos+=1

return None

**------WordEmbedding------**

import numpy as np

class WordEmbedding(object):

#constructor

def \_\_init\_\_(self, word, embedding):

self.word = word #string for the word

self.emb = np.array(embedding, dtype=np.float32) #np array for embeddings

**VI – Academic Honesty Certification**

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.

x- Elisa Jimenez Todd