

CS2302 Data Structures

Fall 2019

Lab Report 5

Author: Bryan Ramos

Due: November 1st, 2019

Professor: Dr. Olac Fuentes

T.A: Anindita Nath

Introduction

The purpose of this lab was to provide deliberate practice using hash tables to solve the same problem dealt with in the previous laboratory assignment. In lab 4, Word Embedding objects containing a word and its embedding, which was a vector used to determine the similarity of words, were stored as nodes in two tree structures: Binary Search Trees (BSTs) and B-Trees, and it was important to observe their construction and search runtimes especially with different sizes of data. In this lab, Word Embedding objects containing the same type of data are inserted into a hash table using different insert hash functions.

These different hash functions insert data into the hash table with different calculations. It's necessary to compare their different running times to build and search either hash table implementation to measure their efficiency especially with different amounts of data. We can use these calculations and information to compare the running times for hash functions in hash tables with the construction and searching of BSTs and B-Trees.

Proposed Solution Design & Implementation

Since the lab calls for solving the same problem as lab 4, this time with hash tables. Keeping a similar design as lab 4 in terms of user interface, first I dealt with creating the menus. The first menu asks the user to choose a hash table implementation. If invalid non-integer input is provided, the program terminates. If the user provides invalid integer input, that is an integer value that is not 1 or 2, the menu is shown again in the CLI. Once valid input is provided the user gets past this menu and the next menu with the hash functions is displayed. The CLI displays a brief description for each hash function. Again, the user must provide valid integer input, that is, integer values 1-6, else, error output will be displayed in the CLI like with the first menu (See invalid input in the experimental results).

Throughout writing the code for this lab, the use of if-else-if conditions is necessary, for example to split the code between each implementation of the hash tables and between each hash functions, depending on the user's preferences/choices. Anyhow, the next part of solving the problem was importing the code for hash tables with chaining and with linear probing from the class webpage. Just like how I edited the BST and B-Tree classes to accept WordEmbedding objects as data I did the same with the hash table classes. After modifying the code to accept WordEmbedding objects, I wrote the methods for insert and find for each of the respective hash functions in the lab instruction, including my own custom one (See the table in the experimental results).

Back to the main lab5 class. After the user makes his or her menu choices, I used if conditions to create the user's chosen hash table implementation. Either build hash table methods accept three parameters: the glove file filename, the chosen hash function, and the limit on the amount of lines to read from the file. When writing the code for this lab I was initially having trouble with the running times while debugging so I implemented a limit on the amount of lines that will be read from the file to make debugging possible. The structure of either building method is similar. In both methods, a hash table, with its respective type is created. The file is read line by line until the line count is met or the last line in the file is met. Each line is read, and the data is tokenized and stored. The word is detected because it begins with an alphabetical letter and is stored as a word in the WordEmbeddings object and the embeddings (vectors) are stored in a list in the WordEmbeddings object. Now depending on the chosen hash function type, the WordEmbedding object is inserted into the hash table utilizing that insert method for that function. The runtime during each insertion is calculated and added to a total runtime to be returned along with the filled hash table. If the line limit is reached, the hash table and runtime are both returned at that point. The file is closed to conserve memory. Next, like with trees, I read the word file to find words. Because the names of the insert and find functions are the same regardless of what hash table implementation, I was able to simplify my code to one method for this

step. The function takes three parameters: the built hash table, the filename of the similarities file, and the hash function number (1-6) chosen by user at the beginning of execution. The similarities file is opened inside of this function and read line by line. The function works the same as the one used in lab 4. To maintain the same functionality as lab 4, I imported my code from lab 4 that basically ensures that the similarities file has two words per line. The words are searched for in the built hash table. Two things can happen:

If the words are found, the WordEmbedding objects are appended to the list of embeddings. Otherwise, the strings are appended to the list. Like the building hash table functions, the find methods used to search through the hash table are respective to each hash function, so only the find method corresponding to the hash function the user specified to be used is used. The runtime is calculated with each search, and at the end, the total running time and filled in embeddings is returned to the main method.

In the main method, the similarities can be calculated. Here I made one change to the WordEmbedding class to improve comparisons between the strings. The code to print the similarities and the floating-point number or the no embeddings is exactly the same as the code I used for lab 4, the same for the similarities method.

At the very end, my program displays the returned running times from earlier so the user can see how long it took to construct the hash table and to then search it. For this lab, I did some research on how to more efficiently gather running time data from my lab program by looking at Medium.com, StackOverflow and GeekForGeeks to build a method that gets the running times for each of the hash functions and inserts them into either a CSV or SQL file, this time around I went with a CSV file.

Experimental Results

Demonstration of menus and output for both implementations:

- First menu asking user to choose table implementation

```
C:\Windows\System32\cmd.exe - python3 ./lab5.py
C:\Users\Bryan Ramos\Documents\CS2302\lab5>python3 ./lab5.py
Choose hash table implementation:
Type 1 for Hash Table Chaining or 2 Hash Table Probing
Hash Table Choice: 1
```

- Second menu asking user to choose hash function with description

```
C:\Windows\System32\cmd.exe - python3 ./lab5.py
C:\Users\Bryan Ramos\Documents\CS2302\lab5>python3 ./lab5.py
Choose hash table implementation:
Type 1 for Hash Table Chaining or 2 Hash Table Probing
Hash Table Choice: 1
Choose hash function:
1. The length of the string % n
2. The ascii value (ord(c)) of the first character in the string % n
3. The product of the ascii values of the first and last characters in the string % n
4. The sum of the ascii values of the characters in the string % n
5. The recursive formulation  $h("",n) = 1; h(S,n) = (\text{ord}(s[0]) + 255 * h(s[1:],n)) \% n$ 
6. (The length of the string // 2) % n
Hash function choice:
```

- Output if user choose choice 1 for hash table (hash table with chaining) and 1 for hash function (length of the string % n

```
C:\Windows\System32\cmd.exe
C:\Users\Bryan Ramos\Documents\CS2302\lab5>python3 ./lab5.py
Choose hash table implementation:
Type 1 for Hash Table Chaining or 2 Hash Table Probing
Hash Table Choice: 1
Choose hash function:
1. The length of the string % n
2. The ascii value (ord(c)) of the first character in the string % n
3. The product of the ascii values of the first and last characters in the string % n
4. The sum of the ascii values of the characters in the string % n
5. The recursive formulation  $h("",n) = 1; h(S,n) = (\text{ord}(s[0]) + 255 * h(s[1:],n)) \% n$ 
6. (The length of the string // 2) % n
Hash function choice: 1

Building Hash Table Chaining

Reading word file to determine similarities
```

```
C:\Windows\System32\cmd.exe
No embedding for continually or communist
No embedding for collectible or handmade
Similarity [greene,entrepreneurs] = -0.09559895098209381
No embedding for robots or grenada
No embedding for creations or jade
No embedding for scoop or acquisitions
No embedding for foul or keno
No embedding for gtk or earning
No embedding for mailman or sanyo
No embedding for nested or biodiversity
Similarity [excitement,somalia] = 0.010856878943741322
No embedding for movers or verbal
No embedding for blink or presently
Similarity [seas,carlo] = -0.07288948446512222
No embedding for workflow or mysterious
No embedding for novelty or bryant
No embedding for tiles or voyuer
```

```
C:\Windows\System32\cmd.exe
No embedding for ranger or smallest
No embedding for insulation or newman
Similarity [marsh,ricky] = 0.3432876765727997
No embedding for ctrl or scared
No embedding for theta or infringement
Similarity [bent,laos] = 0.07367416471242905
No embedding for subjective or monsters
No embedding for asylum or lightbox
No embedding for robbie or stake
No embedding for cocktail or outlets
No embedding for swaziland or varieties
No embedding for arbor or mediawiki
No embedding for configurations or poison
Running time for Hash Table Chaining construction: 3.6708133220672607
Running time for Hash Table Chaining searching: 2.7774665355682373

C:\Users\Bryan Ramos\Documents\CS2302\lab5>
```

Invalid Input:

```
C:\Windows\System32\cmd.exe
C:\Users\Bryan Ramos\Documents\CS2302\lab5>python3 ./lab5.py
Choose hash table implementation:
Type 1 for Hash Table Chaining or 2 Hash Table Probing
Hash Table Choice: a
Invalid input! Provide an integer value.

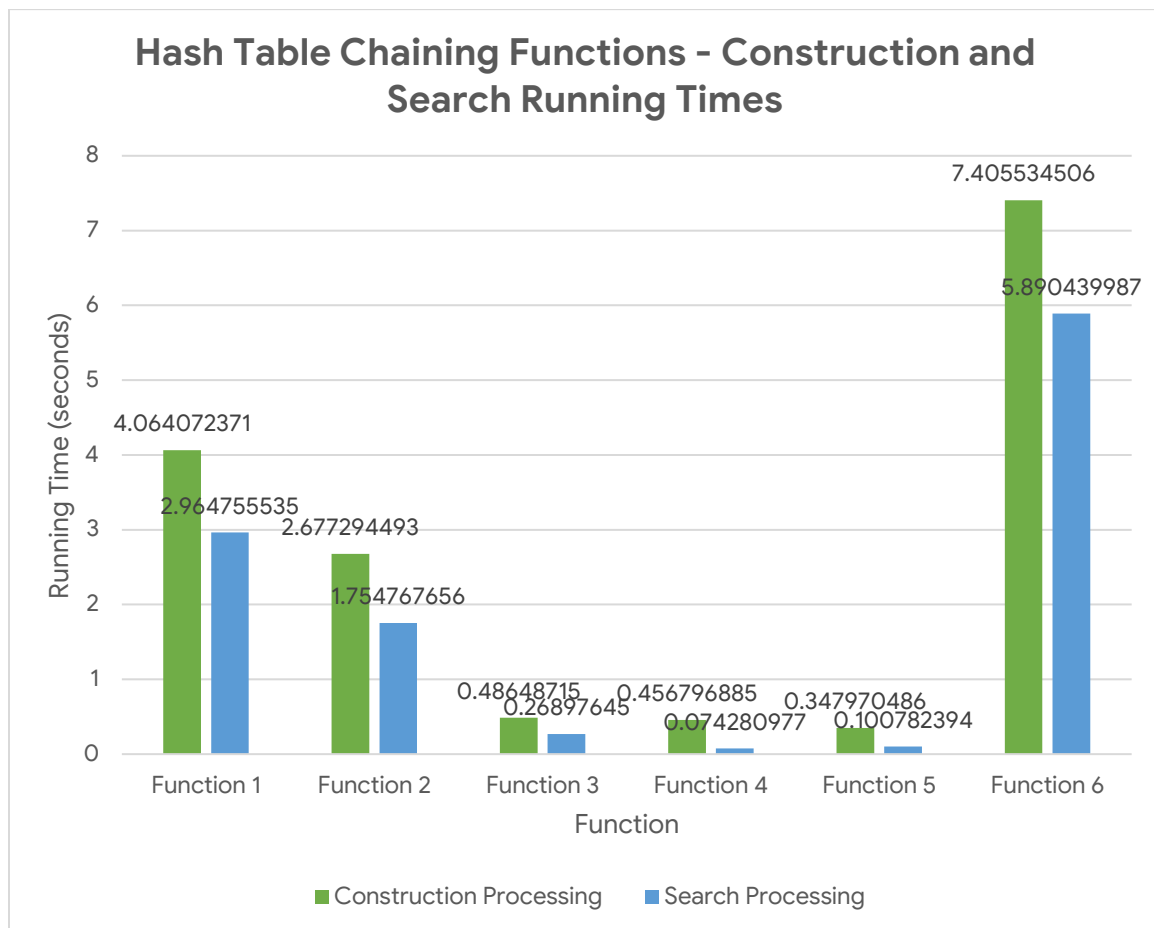
Building Hash Table Probing

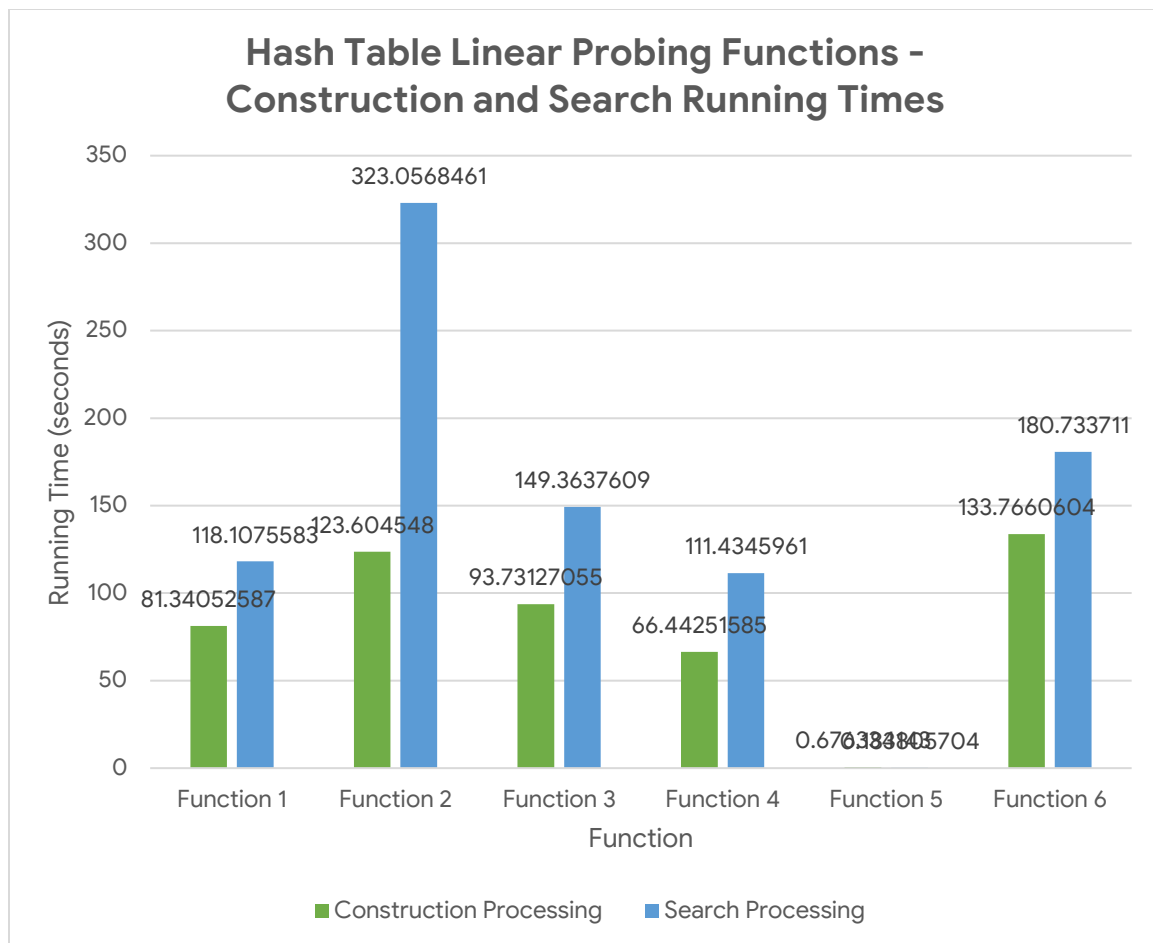
Reading word file to determine similarities

Running time for Hash Table Probing construction: 0
Running time for Hash Table Probing searching: 0

C:\Users\Bryan Ramos\Documents\CS2302\lab5>
```

Function Number	Description
1	The length of the string % n
2	The ascii value (ord(c)) of the first character in the string % n
3	The product of the ascii values of the first and last characters in the string % n
4	The sum of the ascii values of the characters in the string % n
5	The recursive formulation $h("",n) = 1$; $h(S,n) = (\text{ord}(s[0]) + 255 * h(s[1:],n)) \% n$
6	(The length of the string // 2) % n





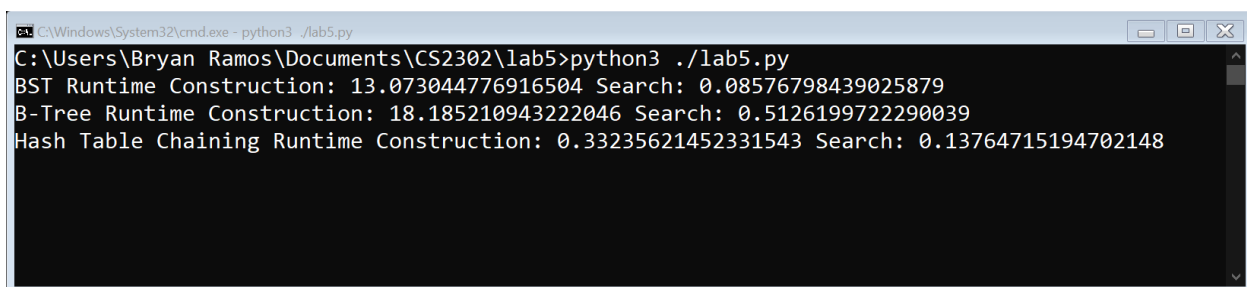
The table shows the functions that were used for lab 5. Functions 1-5 are predetermined based on the lab instructions. Function 6 was a custom-made hash function. Looking at the two graphs showing the running times for the construction and searching of either hash table with chaining or hash table with linear probing using different insert functions, linear probing is slower in all cases (both construction and search processing) of hash functions than chaining. The differences in running times are quite drastic. The construction and searching running times for hash tables with chaining are under 10 seconds. On the other hand, except for function 5, all the running times for hash tables with linear probing are above 60 seconds, or above a minute, especially function 2, with running times that reach 5 minutes.

Let's look closely at the hash table with chaining implementation. Although all the functions are extremely fast, 3-5 are extremely quick, being fractions of a second.

Looking at the running times for building and search processing, it's evident that building a hash table with chaining is slower than searching through it.

On the other hand, looking closely at the hash table with linear probing implementation, the functions are slow with the fastest being function 5. Function 2 is slow, taking upwards of 5 minutes for searching. Looking at the data for running times for building and search processing, building a hash table with probing is faster than searching through it. In all cases, the searching took longer than building. The running times increase as the table is filled with more data, searching for a word not in the hash table, or searching for a word in the hash table that is not there, and the table continues to be filled.

How does a hash table implementation face off against a tree implementation?



```
C:\Windows\System32\cmd.exe - python3 ./lab5.py
C:\Users\Bryan Ramos\Documents\CS2302\lab5>python3 ./lab5.py
BST Runtime Construction: 13.073044776916504 Search: 0.08576798439025879
B-Tree Runtime Construction: 18.185210943222046 Search: 0.5126199722290039
Hash Table Chaining Runtime Construction: 0.33235621452331543 Search: 0.13764715194702148
```

The hash table with chaining implementation construction surpasses the BST and B-Tree implementations in terms of speed. However, in terms of search, the Hash Table Chaining is only faster than the B-Tree Implementation, at search, the BST implementation is the fastest.

- hash operations generally have $O(1)$ time – with a good hash function
 - o worst case $O(n)$
- tree operations are $O(\log(n))$.

Conclusion

Hash tables operations are indeed very fast. As we learned during the class lectures, hash operations are normally $O(1)$ if a good hashing function is provided. The worst-case runtime for a hashing function is linear or $O(n)$. In terms of the hash functions used, hash table with chaining outperformed hash table with linear probing. It is a data

structure to consider when working on personal projects because of its fast insertions and searching. When compared to the tree implementations, hash tables with chaining are generally faster when construction and sometimes slower or faster when searching, again depending on the hash function used. Nonetheless, the data structures have the following runtimes:

- hash operations generally have $O(1)$ time – with a good hash function
 - o worst case $O(n)$
- tree operations are $O(\log(n))$.

Appendix

...

Course: CS2302 Data Structures Fall 2019

Author: Bryan Ramos [88760110]

Assignment: Lab 5

Instructor: Dr. Olac Fuentes

TA: Anindita Nath

Last Modified: November 5th 2019

Purpose: Model the use of hash tables with chaining and hash tables with linear probing to store Word Embedding objects using different hash functions and analyze their running times to compare them to that of trees.

...

```
import os
import numpy as np
from htc import HashTableChainLab5
from htlp import HashTableLP
import time
from wordEmbedding import WordEmbedding

# build hash table of Word Embedding objects using chaining
# parameters: name of file, hash function to build table, maximum
```

```

# amount of lines to deal with high amount of building time
def buildHashTableChaining(file_name, hashFunction, max):
    # catch file not found exception
    try:
        # file utilizes utf8 encoding
        f = open(file_name, "r", encoding="utf8")
        totaltime = 0

        H = HashTableChainLab5(400000)

        lines = 0 # compare to max amount of lines limit
        for line in f:
            tokens = line.split(" ")
            # store if value begins with alphabetic letter (A-Z, lowercase or uppercase)
            if tokens[0].isalpha():

                # The length of the string % n
                if hashFunction == 1:
                    start = time.time()
                    H.insert1(WordEmbedding(tokens[0], tokens[1:]))
                    end = time.time()
                    totaltime += end - start

                # The ascii value (ord(c)) of the first character in the string % n
                if hashFunction == 2:
                    start = time.time()
                    H.insert2(WordEmbedding(tokens[0], tokens[1:]))
                    end = time.time()
                    totaltime += end - start

                # The product of the ascii values of the first and last characters in the string % n
                if hashFunction == 3:
                    start = time.time()
                    H.insert3(WordEmbedding(tokens[0], tokens[1:]))
                    end = time.time()
                    totaltime += end - start

```

```

# The sum of the ascii values of the characters in the string % n
if hashFunction == 4:
    start = time.time()
    H.insert4(WordEmbedding(tokens[0], tokens[1:]))
    end = time.time()
    totaltime += end - start

# The recursive formulation  $h("",n) = 1$ ;  $h(S,n) = (\text{ord}(s[0]) + 255 \cdot h(s[1:],n))\%$ 
n

if hashFunction == 5:
    start = time.time()
    H.insert5(WordEmbedding(tokens[0], tokens[1:]))
    end = time.time()
    totaltime += end - start

# (The length of the string // 2) % n
if hashFunction == 6:
    start = time.time()
    H.insert6(WordEmbedding(tokens[0], tokens[1:]))
    end = time.time()
    totaltime += end - start

lines = lines + 1

# if the current line matches the maximum amount of lines allotted
if lines == max:
    return totaltime, H

f.close() # close glove file to save memory
return totaltime, H
except IOError:
    print("File {} was not found!".format(file_name))

# build hash table of Word Embedding objects using probing
# parameters: name of file, hash function to build table, maximum
# amount of lines to deal with high amount of building time

```

```

def buildHashTableProbing(file_name, hashFunction, max):
    # catch file not found exception
    try:
        # file utilizes utf8 encoding
        f = open(file_name, "r", encoding="utf8")
        totaltime = 0

        HTLP = HashTableLP(400000)

        lines = 0 # compare to max amount of lines limit
        for line in f:
            tokens = line.split(" ")
            # store if value begins with alphabetic letter (A-Z, lowercase or uppercase)
            if tokens[0].isalpha():

                # The length of the string % n
                if hashFunction == 1:
                    start = time.time()
                    HTLP.insert1(WordEmbedding(tokens[0], tokens[1:]))
                    end = time.time()
                    totaltime += end - start

                # The ascii value (ord(c)) of the first character in the string % n
                if hashFunction == 2:
                    start = time.time()
                    HTLP.insert2(WordEmbedding(tokens[0], tokens[1:]))
                    end = time.time()
                    totaltime += end - start

                # The product of the ascii values of the first and last characters in the string % n
                if hashFunction == 3:
                    start = time.time()
                    HTLP.insert3(WordEmbedding(tokens[0], tokens[1:]))
                    end = time.time()
                    totaltime += end - start

```

```

        # The sum of the ascii values of the characters in the string % n
        if hashFunction == 4:
            start = time.time()
            HTLP.insert4(WordEmbedding(tokens[0], tokens[1:]))
            end = time.time()
            totaltime += end - start

        # The recursive formulation  $h("",n) = 1$ ;  $h(S,n) = (\text{ord}(s[0]) + 255 \cdot h(s[1:],n))\%$ 
n
        if hashFunction == 5:
            start = time.time()
            HTLP.insert5(WordEmbedding(tokens[0], tokens[1:]))
            end = time.time()
            totaltime += end - start

        # (The length of the string // 2) % n
        if hashFunction == 6:
            start = time.time()
            HTLP.insert6(WordEmbedding(tokens[0], tokens[1:]))
            end = time.time()
            totaltime += end - start

    lines = lines + 1

    # if the current line matches the maximum amount of lines allotted
    if lines == max:
        return totaltime, HTLP

    f.close() # close glove file to save memory
    return totaltime, HTLP

except IOError:
    print("File {} was not found!".format(file_name))

# build spreadsheet data for lab report
def calculateRuntimes():
    f = open("runtimes.csv", "w")

```

```

for r in range(1, 7):
    runtime, H = buildHashTableChaining("glove.6B.50d.txt", int(r), 10000)
    totalTime, embeddingsList = embeddings(H, "similarities.txt", int(r))
    f.write("Hash Function " + str(r) + "," + str(runtime) + "," + str(totalTime) + "\n")

for r in range(1, 7):
    runtime, H = buildHashTableProbing("glove.6B.50d.txt", int(r), 10000)
    totalTime, embeddingsList = embeddings(H, "similarities.txt", int(r))
    f.write("Hash Function " + str(r) + "," + str(runtime) + "," + str(totalTime) + "\n")

f.close() # close file to save memory

# retrieve embeddings from the similarities file
def embeddings(H, file_name, hashFunction):
    # catch file not found exception - in main, there is a condition that guarantees
    # similarities file will be found - but nonetheless, an exception is provided
    try:
        f = open(file_name)
        totaltime = 0
        embeddings = []

        for line in f:
            # 1st remove remove new line character from line
            if "\n" in line:
                line = line[:-1]

            # split by comma tokenize into list
            words = line.split(",")

            # The length of the string % n
            if hashFunction == 1:
                start = time.time()
                firstWord = H.find1(words[0])
                secondWord = H.find1(words[1])

                if firstWord == None or secondWord == None:
                    embeddings.append(words)

```

```

        else:
            embeddings.append([firstWord, secondWord])
            totaltime = totaltime + (time.time() - start)

# The ascii value (ord(c)) of the first character in the string % n
if hashFunction == 2:
    start = time.time()
    firstWord = H.find2(words[0])
    secondWord = H.find2(words[1])

    if firstWord == None or secondWord == None:
        embeddings.append(words)
    else:
        embeddings.append([firstWord, secondWord])
        totaltime = totaltime + (time.time() - start)

# The product of the ascii values of the first and last characters in the string %
n
if hashFunction == 3:
    start = time.time()
    firstWord = H.find3(words[0])
    secondWord = H.find3(words[1])

    if firstWord == None or secondWord == None:
        embeddings.append(words)
    else:
        embeddings.append([firstWord, secondWord])
        totaltime = totaltime + (time.time() - start)

# The sum of the ascii values of the characters in the string % n
if hashFunction == 4:
    start = time.time()
    firstWord = H.find4(words[0])
    secondWord = H.find4(words[1])

    if firstWord == None or secondWord == None:
        embeddings.append(words)

```



```

        else:
            embeddings.append([firstWord, secondWord])
            totaltime = totaltime + (time.time() - start)

# The recursive formulation  $h("",n) = 1$ ;  $h(S,n) = (\text{ord}(s[0]) + 255 \cdot h(s[1:],n)) \% n$ 
if hashFunction == 5:
    start = time.time()
    firstWord = H.find5(words[0])
    secondWord = H.find5(words[1])

    if firstWord == None or secondWord == None:
        embeddings.append(words)
    else:
        embeddings.append([firstWord, secondWord])
        totaltime = totaltime + (time.time() - start)

# (The length of the string // 2) % n
if hashFunction == 6:
    start = time.time()
    firstWord = H.find6(words[0])
    secondWord = H.find6(words[1])

    if firstWord == None or secondWord == None:
        embeddings.append(words)
    else:
        embeddings.append([firstWord, secondWord])
        totaltime = totaltime + (time.time() - start)

f.close()
return totaltime, embeddings

except IOError:
    print("File {} was not found!".format(file_name))

# used in lab 4
def similarities(e1, e2):
    return np.dot(e1.emb, e2.emb)/(np.linalg.norm(e1.emb) * np.linalg.norm(e2.emb))

```

```

def hashTableType(H):
    import htc
    import htlp
    if type(H) == htc.HashTableChainLab5:
        return "Hash Table Chaining"
    if type(H) == htlp.HashTableLP:
        return "Hash Table Probing"

# get a file of words, create a new file, and write to
# the file having two words per line in the new file
def writeToSimilaritiesFile(file_name):
    try:

        # open file to read words from
        # create new file to write to
        f = open(file_name)
        new = open("similarities.txt", "w")

        word = 0
        for line in f:
            if "\n" in line:
                line = line[:-1]
            if word == 1:
                new.write(line + "\n")
                word = 0
            else:
                new.write(line + ",")
                word = word + 1

        f.close() # close the files to save memory
        new.close()
    except IOError:
        print("File", file_name, "not found!\n")

# main method
if __name__ == "__main__":

```

```

# txt file from nlp.stanford.edu
file_name = "glove.6B.50d.txt"

# calculateRuntimes()

# compare running times of hash table with chaining and BST

...

import lab4_source as lab4

bst_construct_time = 0
btree_construct_time = 0
table_construct_time = 0

# build bst
start = time.time()
binarySearchTree = lab4.buildBST(file_name)
bst_construct_time = time.time() - start

bst_search_time, bst_embeddings = lab4.getEmbeddings(binarySearchTree, "similarities.txt")

# build btree
start = time.time()
bTree = lab4.buildBTree(5, file_name)
btree_construct_time = time.time() - start

btree_search_time, btree_embeddings = lab4.getEmbeddings(bTree, "similarities.txt")

# build hash table

start = time.time()
table_construct_time, hash_table = buildHashTableChaining(file_name, 3, 10000)
table_search_time, embeddings = embeddings(hash_table, "similarities.txt", 3)

print("BST Runtime Construction: {} Search: {}".format(bst_construct_time, bst_search_time
))

```

```

print("B-
Tree Runtime Construction: {} Search: {}".format(btree_construct_time, btree_search_time))
    print("Hash Table Chaining Runtime Construction: {} Search: {}".format(table_construct_time, table_search_time))
print("\n\n\n\n")
...

# check if similarities text file exists
# if not - open a file containing English words and write to a new file
# two words per row to be used to find similarities in part 3
if not os.path.exists("similarities.txt"):
    writeToSimilaritiesFile("english-words.txt")

# vars
choice = 0
hashFunction = 0
hashFunctions = [
    "The length of the string % n",
    "The ascii value (ord(c)) of the first character in the string % n",
    "The product of the ascii values of the first and last characters in the string % n",
    "The sum of the ascii values of the characters in the string % n",
    "The recursive formulation h(",n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n",
    "(The length of the string // 2) % n"]
H = None # hash table initially null

# catch non-integer input from user
try:
    # valid input for choice: 1 (Hash Table Chaining) or 2 (Hash Table Probing)
    while choice < 1 or choice > 2:
        print("Choose hash table implementation:\nType 1 for Hash Table Chaining or 2 Hash
Table Probing")
        choice = int(input("Hash Table Choice: "))

    # hash functions from lab instructions
    # valid input for functions: 1, 2, 3, 4, 5, 6
    while hashFunction < 1 or hashFunction > 6:
        print("Choose hash function:")

```

```

        # print each of the strings for the types of hash functions options
        for i in range(len(hashFunctions)):
            print("{} . {}".format(i + 1, hashFunctions[i]))
        hashFunction = int(input("Hash function choice: "))

except ValueError:
    print("Invalid input! Provide an integer value.")

# build hash table
# hash table chaining
if choice == 1:
    print("\nBuilding Hash Table Chaining")
    runtime, H = buildHashTableChaining(file_name, hashFunction, 10000)
# hash table probing
else:
    print("\nBuilding Hash Table Probing")
    runtime, H = buildHashTableProbing(file_name, hashFunction, 10000)

file_name2 = "similarities.txt"
print("\nReading word file to determine similarities\n")

totalTime, embeddings = embeddings(H, file_name2, hashFunction)

for element in embeddings:
    if any(isinstance(words, str) for words in element):
        print("No embedding for {} or {}".format(element[0], element[1]))
    else:
        print("Similarity [{},{}] = {}".format(element[0].word, element[1].word, similarities(element[0], element[1])))

print("Running time for {} construction: {}".format(hashTableType(H), runtime))
print("Running time for {} searching: {}".format(hashTableType(H), totalTime))

```

I, Bryan Ramos, 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.