**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.

**Experimental Results**

|  |  |
| --- | --- |
| 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) = (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 of 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?

**Conclusion**

**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(400009)

        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) = (ord(s[0]) + 255\*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 alloted

            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(400009)

        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) = (ord(s[0]) + 255\*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 alloted

            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) = (ord(s[0]) + 255\*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\_\_":

    # calculateRuntimes()

    # txt file from nlp.stanford.edu

    file\_name = "glove.6B.50d.txt"

    # 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",

        "Another function of your choice"]

    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("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\n")

        runtime, H = buildHashTableChaining(file\_name, hashFunction, 10000)

    # hash table probing

    else:

        print("\nBuilding Hash Table Probing\n")

        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))