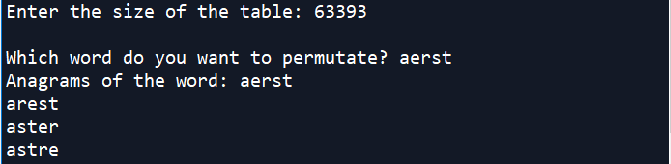
Ricardo Godoy

Lab Report

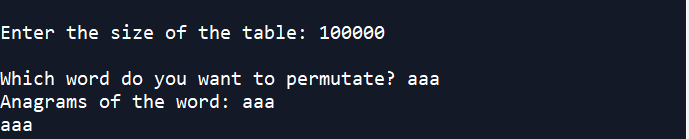
The purpose of this lab was to create a hash table with a specific number of English words, in this case 354,98, and then count how many anagrams does a specific word have and how many average comparisons per word it computes. I created the hash table on the size of the number of words, so each word is inserted into each bucket, leaving no empty buckets in the hash table. This was done with the purpose of making the running time of the program O(1), making the word easier to find. As the program previously created, this program also prompts the user to choose the desired word to perform these operations, along with the size of the list of words in the file.

In my HashTable class, I included five hash functions. One function to insert, another to remove, another to search for the word, one to get the average number of comparisons per word, and finally, another function to get the load factor. The purpose of this was to be able to perform all the operations available from the hash class and compute the average number, along with the load factor, from the HashTable’s class. The Hash table was created depending on the input of the user. The preferred size for the table was the number of the words that were actually in the file, in order to avoid collisions and improve the running time of the program, even though it uses more memory. The insert function was designed to append the item into each bucket, avoiding collisions. The average\_comparison function has a counter to count how many average comparisons the program computes for the words, having a running time of O(1). Along with that, I also implemented different functions extracted from the previous lab and adapted to hash table implementation, to print the anagrams of a specific word, count them, and get the word with most anagrams in the file.

The tests I produced for this program consisted in changing the size of the file and therefore, the size of the hash table itself. The operations performed therefore varied depending on the size of the table, and the user input. First, I used a table of size 63,393. Here is the output:



The second test was with a large input size. I decided to increment the table size to 100,000. This was the output produced:



The running time of this program is O(1). This is due to the large table size and the avoidance of collisions. The program is able to find a specific word in O(1).

In this lab- I learned how to implement hash tables to manage large input sizes and to implement different solutions of hash tables, along with how to implement or avoid chaining. It is difficult to find the best hashing function possible depending on the scenario and the computations necessary. In this case, because of my computer specs, I decided to go for more speed rather than save memory.

Appendix

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

"""

Created on Mon Nov 5 21:05:46 2018

@author: Ricardo

"""

# Course: CS2302

# Author: Ricardo Godoy

# Assignment: Lab 4 B

# T.A: Saha, Manoj

# Instructor: Diego Aguirre

# Date of last modification: 11/11/18

# This program reads a file that contains all english words and inserts them into a hash table, selected

# by the user. The program prints the anagrams of a specific word, only if it is in the hash table.

# Along with that, the program also computes the word with the most number of anagrams and the

# number of anagrams of each word

# HashTable class using chaining.

class HashTable:

# Constructor with optional initial capacity parameter.

# Assigns all buckets with an empty list.

def \_\_init\_\_(self, initial\_capacity=26):

# initialize the hash table with empty bucket list entries.

self.table = []

for i in range(initial\_capacity):

self.table.append([])

# Inserts a new item into the hash table.

def insert(self, item):

# get the bucket list where this item will go.

bucket = (ord(item[:1].lower())-97) % len(self.table)

# bucket = hash(item) % len(self.table)

bucket\_list = self.table[bucket]

# insert the item to the end of the bucket list.

bucket\_list.append(item.lower())

# Searches for an item with matching key in the hash table.

# Returns the item if found, or None if not found.

def search(self, key):

# get the bucket list where this key would be.

bucket = (ord(key[:1].lower())-97) % len(self.table)

bucket\_list = self.table[bucket]

# search for the key in the bucket list

if key in bucket\_list:

# find the item's index and return the item that is in the bucket list.

item\_index = bucket\_list.index(key)

return bucket\_list[item\_index]

else:

# the key is not found.

return None

# Removes an item with matching key from the hash table.

def remove(self, key):

# get the bucket list where this item will be removed from.

bucket = hash(key) % len(self.table)

bucket\_list = self.table[bucket]

# remove the item from the bucket list if it is present.

if key in bucket\_list:

bucket\_list.remove(key)

def avg\_comparisons(self):

count = 0

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

count += 1

print(count)

return str(count / len(self.table))

def load\_factor(self):

return self.size / len(self.tableq)

class Counter:

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

self.count = 0

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

def create\_hash(filename):

english\_words = HashTable()

with open (filename) as file:

for line in file:

ln = line.lower()

english\_words.insert(ln.replace("\n", ""))

return english\_words

def count\_anagrams(obj, word, prefix=""):

#Counts the anagrams found in the tree

if len(word) <= 1:

str = prefix + word

if english\_words.search(str):

obj.count = obj.count + 1

else:

for i in range(len(word)):

cur = word[i: i + 1]

before = word[0: i] # letters before cur

after = word[i + 1:] # letters after cur

if cur not in before: # Check if permutations of cur have not been generated.

count\_anagrams(obj, before + after, prefix + cur)

return obj.count

def max\_anagrams(obj, filename):

#Computes the word with the maximum number of anagrams

file = open(filename)

max\_count = 0

max\_word = ""

# num\_words = 4

for line in file:

obj = Counter()

obj.count = count\_anagrams(obj, line.replace("\n", "")) # -1 index means last one

# print(line[0:-1], obj.count)

# print(obj.count)

if obj.count > max\_count:

max\_count = obj.count

max\_word = line.replace("\n", "")

print("Word with more anagrams: " + max\_word)

print("Max number of anagrams: " + str(max\_count))

def print\_anagrams(word, prefix=""):

#Prints the anagrams of a specific word

if len(word) <= 1:

str = prefix + word

if english\_words.search(str):

print(prefix + word)

else:

for i in range(len(word)):

cur = word[i: i + 1]

before = word[0: i] # letters before cur

after = word[i + 1:] # letters after cur

if cur not in before: # Check if permutations of cur have not been generated.

print\_anagrams(before + after, prefix + cur)

def main():

global english\_words

english\_words = HashTable()

file = "words\_short.txt"

size = input("Enter the size of the table: ")

english\_words = HashTable(int(size))

english\_words = create\_hash(file)

anagram\_word = input("Which word do you want to permutate? ")

print("Anagrams of the word: " + anagram\_word)

print\_anagrams(anagram\_word)

print("Average number of comparisons: " + str(english\_words.avg\_comparisons))

print("Load factor: " + str(english\_words.load\_factor))

main()