Isaac Acosta | CS 2302 | Option B

Lab 4 Report

# Introduction

In this lab project, the objective is to borrow from the objective of the previous lab (lab 3) of using word Anagrams, and instead of using an AVL and Red-Black tree implementation, the algorithm must now use a Hash table implementation. The purpose of this change is to compare the time differences between using trees and Hash tables. The time complexities between both should be a major difference. It is hypothesized that the Hash table implementation should fare faster with this algorithm than the AVL and Red-Black tree implementations.

These are the things that are to be calculated from this lab:

* The time it takes to successfully retrieve a word string from the hash table.
* Compute the table’s load factor.
* Sort the words in the hash table in a form as if each word is a base-26 number.
* Chaining the hash table to solve the collision in the Hash table.

# Design and Implementation

The design and implementation that this lab procedure uses follows as is. The data structure that is now featured is a Hash table data structure. The hash table uses chaining to solve its collision problem. As it is, the hash table is expected to store more than 300,000 words, so the hash table will feature a large number table size. A large table size is vital to ensure that the time to search for a word string in the hash is as fast as possible, or at least faster than the AVL and Red-Black. Otherwise, the smaller the table size, the longer it will take to retrieve a string from the table. The table size that will be used in this lab will be of size 50,060.

In addiction to the hash table, it will have an insert function, a search function, a function to calculate the load factor, a function that grabs a string, treats it as a base base-26 number, and converts it to a base-10 number (This function is to complement the insertion process of the Hash table class). The insert function uses the modulo method to make insertions, and python can only perform a modulo on integers.

The main program features three additional functions, populate\_hash(), greatest\_anagrams(), and num\_of\_anagrams(). The first function uses an object text file and integer representing the table size as references, and creates a hash table, populates it by reading the object text file, and returns the hash table. The seconds function does the same procedure as its variant from the previous lab, it calculates the greatest number of anagrams in a text file containing a separate list of strings. The third function also does the task as its previous variant of the last lab, it calculates the number of valid anagrams a given word has. Both greatest\_anagrams() and num\_of\_anagrams() have been optimized for the use of a hash table structure.

So, with the above being said, the main program follows as is, the begins by reading a text file named words.txt that is expected to contain 354,984 words. But need to remind that the actual text file contains 466,507 words. After the text file is read, a variable engish\_words is initialized, and becomes a hash table when populate\_hash() is called. Afterwards, what follows is simply searching and retrieving for the word “blue” by calling engish\_words.search(“blue”). Third, another text file is read, this text file is expected to simply contain a random list of strings. After the text file is read, the word with the greatest number of anagrams is calculated by calling greatest\_anagrams(). Lastly, the program simply uses the word “spot” to calculate the number of anagrams by calling num\_of\_anagrams(“spot”). And for the sake of calculating the time complexity and comparing the running time of this implementation and the implementation of the previous lab, each step of the main programs is timed accordingly.

# Experimental Results

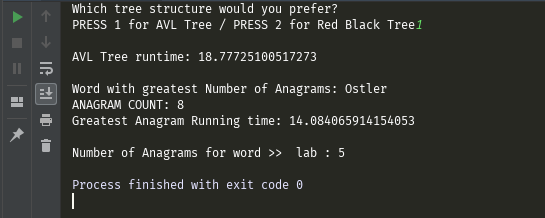
What could be observed and recorded from this implementation is that the search algorithm for a hash table was only faster as long as it had a large number table size. And it makes sense because the program was dealing with the allocation of 466,507 words. Otherwise, when the hash table was using a table size of merely 50 – 100 – 150, the program fared slow, and even quite slowed than the AVL and Red-Black tree implementations. When using table sizes of 5,000 10,000 or even 30,000 to 50,000 the hash table implementation proved to be much faster.

The difference in times and depending of the table size is possibly due to the way the hash table handles collisions. The less indexes the hash table had, the more time it took to traverse each chained index, or linked list. Little advantage is taken from taking the modulo of an integer to get to a specific index when there so few indexes and so many words stored in each index.

* To the table below, the “words” directory represents the number of words inside myWords2.txt.

Screenshot:

Table size of 100,000 and 104 words inside myWords2.txt.



# Conclusion

It can be concluded that a hash implementation can be faster than using a tree implementation. But it all depends on the table size of the hash table. The amount of time it takes to retrieve data from a hash table is significantly faster, given the right table size and collision handling method. And as far as storing strings inside a hash table, it can only be done properly when treating a string as a base-26 number. Otherwise, when dealing with any table size, storing a string in terms of their alphabetical order will only limit the use of hash table to an index size of 26, which is very inefficient. And as far as resource use goes, the bigger the hash table the more ram it will use, which is why chaining is used for this implementation. Storing a total of 400,000+ words in each hash index would be highly inefficient in terms of ram usage.

**Academic Honesty**

“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.”