PA4 Report: Recoding students' grades via a hash table

Minh Luu
CSCE 221-509
Date: April 13th, 2020

Objectives:

In this assignment, I am introduced to another data structure called hash table. I will write the data structure in C++ by implementing the quiz grade matching program where the quiz grades are read, store in a hash table, and match and output with the correct students in the roster file. I will have two classes: HashTable to store the data and the CSVEditor to read and write the CSV. Besides learning about the data structure, I am also introduced brief to regular expression which is a way of expressing pattern to be searched in a longer string. By the end of this assignment, I will be able to add another data structure and programming technique to my toolbox and hopefully use it on various applications in computer science soon.

Procedure:

This programming assignment is divided into three parts: building writeCSVToTable to read the file using regex, building the HashTable class to store the data, and writeCSVToFile to read roster, match quiz grades stored in HashTable to the correct UIN and output it to the output.csv file. I also display minimum, maximum, and average chain length for our hash table.

Hash table, in our case particularly, can be implemented by using an array of fixed size. To insert a key/value pair, the key is first hashed. Since hashes are just large integers, the hash is then taken modulo the size of the array, yielding an index. The key/value pair is then inserted into a bucket at that index. I use chaining method, an array of linked lists, so the first one to be inserted into a bucket will be at the top and subsequently, the next one that is also in the same bucket would be linked or chained to the last one. It features O(1) average search times, making it an efficient data structure to use for caching, indexing, and other time-critical operations.

As described above, I will read data from the input.csv and get the UIN and the quiz grade from each line using given regex pattern. Once I read it, I will use the convert UIN from string to integer and make UIN and quiz into a pair (int, string). I then insert pair into the hash table by hashing the key and put it in the right bin of the array and insert it into the right linked list. Once I read and store everything from input.csv. I read the roster.csv file using the same regex pattern, get the UIN and search the hash table to see whether that person has taken the quiz or not and output the line and the quiz grade if that person has one into input.csv. The restrictions I assume are the input and roster files are not empty and all students follow the same format and each has an UIN as key.

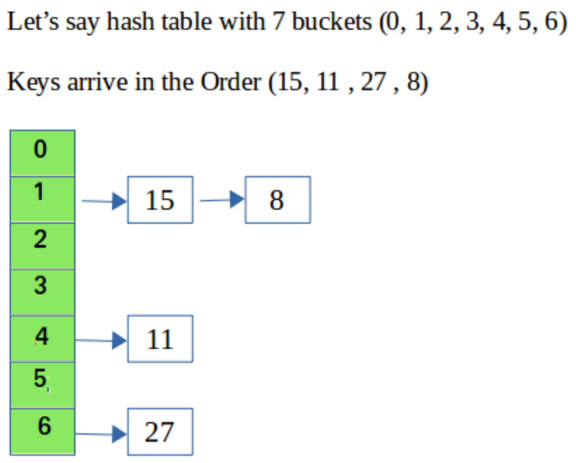
Let's say hash table with 7 buckets (0, 1, 2, 3, 4, 5, 6)
Keys arrive in the Order (15, 11 , 27 , 8)
0
1
2
3
4
5
6
15
8
11
27

Fig 1: A basic hash table. Source: geeksforgeeks.org

The way I test my program, I just call all my functions and run multiple trials to try to break the code. I also ran it through Valgrind to make sure there is no memory leak and it passed that test. A function for HashTable that prints out all the node in the way like figure 1 was added to aid testing. Additionally, I used two stl classes to make my work easier: list for the linked list and pair for the pairs of key and value.

Results and Discussion:

I tried to test out the efficiency by running the program through multiple size configurations. We can see that the higher the number of bins, the lower the chance of getting collisions and the lower the average chain length (aka average search time). This is true for the max and min chain length as well. However, as time complexity decreases, space complexity increase, but it is still worth it because time complexity decreases exponentially while space increases linearly.

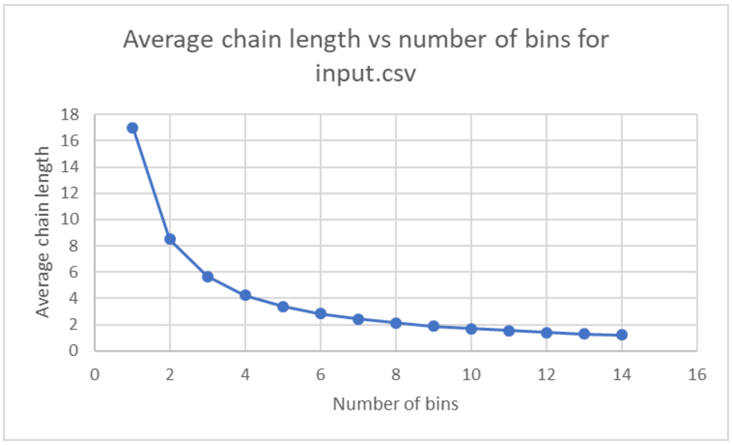

Fig 2: Average chain length vs array size. The average here includes empty bin(s) into the calculation.

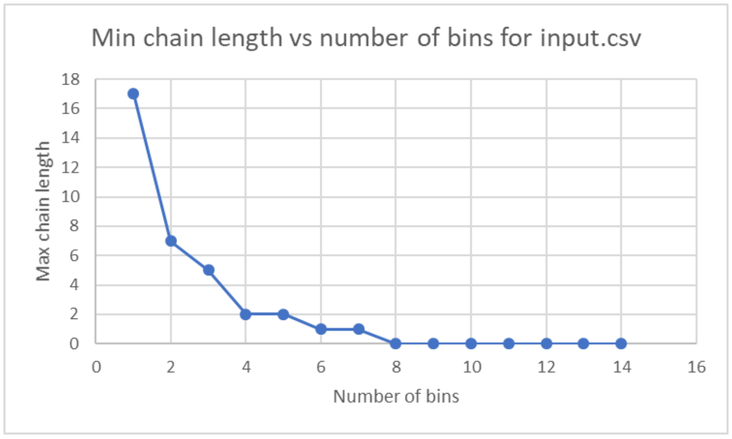

Fig 3: Min chain length vs array size

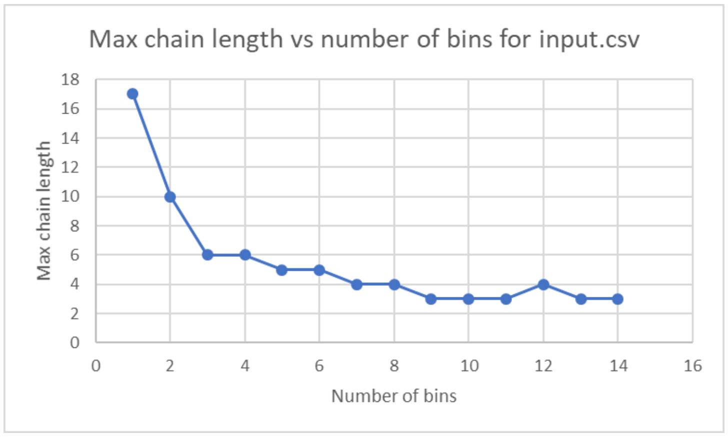

Fig 4: Max chain length vs array size

I expect this to happen because probability speaking, if we have two key-value pairs and the hash table has 1 bin, the chance of collision would 100%, 2 bins would be 50%, 3 bins would be 33.33%, 25%, and so on and so on until it reaches 0% when the array is infinitely large. This also means collisions are practically unavoidable, especially when dealing with large data, which makes it quite inefficient if there are not a lot of bins to accommodate. On the other size, if the number of bins is way larger than the number of values, it would the hash table a fancy array. So finding the balance between these two is necessary.

Conclusion:

In conclusion, I have successfully implemented a hash table. Through the implementation and search time analysis, I have figured out the advantages being store and retrieve data at almost constant time, although building a hash function in many situations can be difficult. Additionally, finding a balance between the time and space complexity for the structure is an important aspect to ensure it performs as efficient as possible. I was able to observe its behaviors with my own eyes which help me remembering many concepts for future considerations when working in the industry.