# California State University, Fresno

# DEPARTMENT OF COMPUTER SCIENCE

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Class: | **Algorithms & Data Structures** | | | Semester: | **Fall 2021** |
|  | | | | | |
| Points |  | Document author: | **Marlon Branham** | | |
|  | Author’s email: | **zekebranham@mail.fresnostate.edu** | | |
| Laboratory number: | **Laboratory 2** | | |
|  | | | | | |

**1. Statement of Objectives**

In this lab, we were to create and implement a hash table that uses linear probing to handle collisions. With the hash table implemented, we were able to learn more about how a hash table works and how to handle issues regarding the collisions and cost of each probe. In this report, I will be going over my experimental procedures, an analysis of the results, encountered problems, and a conclusion of the results.

**2. Experimental Procedure**

This lab was meant to show us how to implement a hash table and its functions. We were to create a program that reads a file, that file being the Romeo and Juliet script, and insert each word of the file into a hash table. I accomplished this by using an fstream function that reads from the file path and inserts each word into a string then to be converted to an ASCII sum that is inserted into the hash table. We were instructed to use linear probing as our method of collision handling, since there were many collisions, the table was quite large. I created a table that was 4001 elements in size and used the hash function that took the modulo of the ASCII value and the size of the table (being a prime number).

After the table implementation was complete, we were to create two separate arrays, updates[i] and cost[i]. The updates array was meant to hold a value that represented the amounts of updates made to the table, in this case being 1 primarily. The cost array was the count of the number of collisions at a given key value, incremented each time a collision occurred with a given value. We were to display the ratio of these two arrays at each index (updates[i]/cost[i]) and count the total number of unique numbers, in this case, the number of filled slots.

**A computer screen shot of a program code

Description automatically generated**

**3. Analysis**

My results from this program are conclusive in the principle of hash tables, each word was stored at a particular value, giving each unique word a slot, in this case being 3,684 unique words. As far as the ratio, the update[i] array was filled with 1’s, and my cost[i] array was filled with the total number of collisions/probes that occurred at each value, making the ratio outputted a bit unjustified since the ratio was equal to the cost[i] value given that the divisor of update[i] was 1’s all the way through the list. I cannot say this information and the data I gathered give me much insight into how efficient the program was, but the cost did not ever surpass 4000, meaning the cost number was less than the ‘n’ size of the array.

A screen shot of a computer

Description automatically generated

**4. Encountered Problems**

Unfortunately, enough, there were a number of issues in implementing the cost[i] and update[i] array given that it was difficult to determine where the incremental values should be implemented in which function and within which loop. To solve this problem, I decided that within the “while.occupied” loop within the “int update” function was the best place to implement the cost/probe incrementation. I did notice that the only way to implement a way to find the number of unique words in a file would be by using an open addressing method like linear/quadratic probing since with closed hashing it would only be able to count the number of unique ASCII sum values seen in the file, in this case, was much smaller than the number of unique actual words in the file.

A computer screen shot of a program code

Description automatically generated

**5. Conclusions**

In this lab, I learned the proper implementation of hash tables and used different methods of handling collisions, and open/closed addressing. I also learned that in order to find the number of unique items in a file or a hash table, you need to use open addressing so you can simply count the number of filled slots in the table. Using the update[i] and cost[i] arrays I was able to find that the cost of inserting an item with a high collision rate is very costly and ineffective if the collision is predicted to be common. Overall, I learned a lot in this lab and came out to be successful in implementing all that was needed.

**6. References**

<https://www.asciitable.com/>

<https://www.digitalocean.com/community/tutorials/hash-table-in-c-plus-plus>

<https://www.geeksforgeeks.org/c-program-hashing-chaining/>