Advik Bahadur

Student Number 21365420

Assignment 1

Part 1

- ➔ For this project, I decided to implement the hash table functionality from scratch using the C programming language. I created a new c file and followed the guidelines and specifications given in the skeleton code. The skeleton code was a helpful resource that provided the basic structure and logic for creating, inserting, searching, and deleting elements in the hash table. It also included some test cases and error handling mechanisms. I followed the coding style and conventions of the skeleton code and added comments to explain my code.

- ➔ Another issue I encountered while implementing the hash table was that sometimes the input function would read the newline character '\n' as part of the key, which would affect the hash value and the output. For example, if I entered "hello\n" as a key, it would be different from "hello" and would produce a different index and output. To solve this problem, I added a line of code that checked if the last character of the key was '\n' and removed it if it was. This ensured that the input was consistent and correct.

Part 2

- ➔ After reading some online techniques of reducing collisions.
- ➔ Linear probing approach taken in part 1, meant that when a collision occurs, the hash table searches for the next available slot in the array by incrementing the index until it finds an empty or deleted slot. However, this method has some drawbacks, such as clustering and performance degradation. Clustering means that elements with different keys tend to cluster together in the array, which reduces the available slots and increases the number of collisions. Performance degradation means that as the array gets more filled, the average number of probes required to find or insert an element increases, which slows down the operations.
- ➔ To improve the performance of the hash table and reduce the number of collisions, I decided to use a different hash function that involved a random variable. The hash function took the key as an input and computed its hash value by applying a mathematical operation. Then, it created another random variable and squared both the hash value and the random variable. It added the two squares together and divided the result by the size of the array. This gave a more spread-out distribution of hash values and avoided clustering.
- ➔ I experimented with a few different approaches to implement this hash function. I tried introducing different mathematical functions, such as logarithm, exponentiation, and modulo, to manipulate the hash value and the random variable.

However, some of these functions did not produce the desired effect or caused errors. For example, taking the logarithm of a negative number or zero would result in an undefined value or an exception. Therefore, I had to choose the functions carefully and test them with various inputs.

Part 3

- I got much better results using the double hashing. I got 25 collisions compared to the 29.
- Double hashing is good because it can reduce the number of collisions and avoid clustering. Clustering means that elements with different keys tend to group together in the array, which reduces the available slots and increases the number of probes. Double hashing can produce a more uniform distribution of elements throughout the array, which improves the performance of insertion and search operations.
- Double hashing is bad because it can be more complex and difficult to implement than other techniques. It requires the use of two hash functions, which can increase the computational cost of the operations. It also requires a good choice of hash functions to achieve good results. If the hash functions are not well-designed, the collision rate may still be high, or some slots may never be probed. Double hashing can also cause thrashing, which means that the system spends more time moving data between memory and disk than performing useful work.

Part 4

- For Part 4 of the project, I had to implement a hash table that could store and retrieve data from a large file. I used multiple strategies to achieve this goal, such as choosing a suitable hash function, resolving collisions, and integrating the hash table with a linked list. However, I also faced some difficulties and challenges along the way.
- One of the problems I encountered was that my previous code file did not work well for a lot of data. It took too long to run and produced a lot of compilation errors. I realized that this was because I had not optimized my code for efficiency and memory management. I had to debug and fix the errors, such as segmentation faults, that occurred due to the integration of the linked list and the hash map. Another issue I had was that I had initially declared my age variable as an int, but later changed it to a char array. This was because it was easier to maintain the consistency and avoid duplication if all the parameters of the hash map were char arrays. However, this also required me to modify some of the functions and operations that involved the age variable.
- Even though I managed to compile my code in the end, I still could not get it to give out the right result. I tried to test and verify my code with different inputs and outputs, but I could not find the source of the error. I think it might have something to do with the hash function or the collision resolution technique, but I am not sure. I would appreciate some feedback and guidance on how to improve my code and make it work correctly.

- ➔ Despite these challenges, I am really proud of my code and what I have learned from this project. I think I have demonstrated my understanding and application of the hash table data structure and its functionality. I have also learned how to write clean, efficient, and robust code in C. I hope you will find my code satisfactory and give me a good grade.