**LAB 5: HASH AND SEARCHING**

**Name: Trần Gia Bảo**

**ID: 22127034**

**EXERCISE 1:**

**Question 1.1: You will use djb2 hash function to convert a *string* to an *Int* in this exercise. Explain in report what you know about this hash function.**

**A computer screen with colorful text

Description automatically generated**

- The given C function is a hash function that takes a string as input and produces a 64-bit hash value as output. The hash function is based on the djb2 algorithm, created by Daniel J. Bernstein, which is a widely used and simple hash function. Let's break down the function step by step:

+ Initialization: The function starts by initializing the hash variable to the number 5381. This number was chosen because testing showed that it results in fewer collisions and better spreading of hash values.

+ Iteration: The function iterates through the characters of the input string s.

+ Hash Calculation: For each character s[c] in the string, the function performs the following operations:

* Multiplier: The hash variable is multiplied by 33. Instead of using the regular multiplication operator (\*), the function employs a bitwise left shift operation (hash << 5) followed by adding the original hash value to it (+ hash). This bitwise left shift by 5 positions is equivalent to multiplying the number by 32 (2^5) and then adding the original value, effectively resulting in multiplication by 33.
* Adding ASCII value: The ASCII value of the current character s[c] is added to the hash variable.

+ Finalization: After iterating through the entire string, the function returns the value held by hash, which represents the calculated hash value for the input string.

- Here's a summary of the hash function's properties:

+ Simplicity: The hash function is straightforward and easy to implement.

+ Speed: The use of bitwise operations for multiplication may offer faster computation on many CPUs compared to regular multiplication.

+ Avalanche effect: Changing a single character in the input string should lead to significant changes in the resulting hash value, ensuring a good avalanche effect.

+ Distribution: The function aims to distribute the hash values uniformly, reducing the likelihood of collisions for different input strings.

+ Bit length: The hash function produces a 64-bit hash value, which provides a wide range of possible hash codes.

- Despite its simplicity and effectiveness for many use cases, the choice of the magic number 33 (used for multiplication) has not been adequately explained by its creator, Daniel J. Bernstein. This choice is one of the characteristics that sets djb2 apart from some other hash functions. Nonetheless, the function has proven to be useful in various applications, such as hash tables and string hashing, due to its simplicity, reasonable distribution, and reduced collision rates.

**Question 1.2: Choose your own hash table size. Explain in report why you choose this size.**

- Given as input an N-element string array: size = N/0.7

- I call 0.7 is a load factor. Choosing a load factor of around 0.7 is a common practice in hash table design and implementation. The load factor represents the ratio of the number of elements stored in the hash table to the total number of buckets (or slots) in the table. In other words, it indicates how "full" the hash table is.

- Here's why I choose a load factor of 0.7:

+ Efficiency: A load factor in this range allows the hash table to strike a balance between memory usage and performance. It ensures that the table is not too sparse (wasting memory) or too dense (increasing the likelihood of collisions).

+ Reduced Collisions: A lower load factor (e.g., 0.5 or lower) can lead to a significant amount of unused space in the hash table. This may be wasteful, especially if memory is a concern. On the other hand, a very high load factor (e.g., 0.9 or higher) can cause more collisions, degrading the hash table's efficiency.

+ Collision Resolution: Although good hash functions aim to minimize collisions, they are still a possibility. By keeping the load factor around 0.7 , the impact of collisions is minimized, and the hash table can still handle collisions effectively with methods like chaining or open addressing.

+ Performance: With a moderate load factor, the average lookup, insertion, and deletion times remain relatively constant. Performance degradation due to collisions is less likely to occur in this range.

+ Expected Growth: Choosing a load factor around 0.7 allows for some level of future growth in the hash table without requiring frequent resizing. As the number of elements increases, the hash table can still maintain a reasonable load factor.

**Question 1.3: Experient with 4 ways to handle collision:**

**a. Chaining approach**

**b. Linear Probing**

**c. Quadratic Probing**

**d. Double Hashing (Choose your own 2nd Hash Function. Explain in report why you choose this hash function)**

**Compare the result from different method. Analyze and discussion.**

|  |  |
| --- | --- |
| **Methods** | **Run time (ms)** |
| **Linear Search** | 3184 |
| **Binary Search** | 20 |
| **Chaining Approach** | 2 |
| **Linear Probing** | 2 |
| **Quadratic Probing** | 2 |
| **Double Hash** | 2 |

- From the experiment results, it appears that linear search and binary search are included in the comparison, which are not directly related to the collision resolution techniques. However, we will focus on analyzing the collision resolution techniques mentioned (Linear Probing, Chaining Approach, Quadratic Probing, and Double Hashing) based on the time taken for each method.

- Linear Probing (2 milisec): Linear probing is a basic open addressing technique that sequentially searches for the next available slot in case of a collision. The result of 2 milliseconds indicates that the performance of linear probing was reasonably efficient for the given dataset and load factor.

- Chaining Approach (2 milisec): The chaining approach, where collisions are resolved by creating linked lists at each bucket, also took 2 milliseconds. This result suggests that chaining was able to handle collisions efficiently for the dataset and provided similar performance to linear probing.

- Quadratic Probing (2 milisec): Quadratic probing, another open addressing technique, uses a quadratic function to find the next available slot after a collision. The result of 2 milliseconds indicates that quadratic probing performed well and achieved the same level of efficiency as linear probing and chaining in this particular scenario.

- Double Hashing (2 milisec):Double hashing uses two hash functions to calculate the step size for probing after a collision. The result of 2 milliseconds suggests that double hashing, with the chosen second hash function, was effective in handling collisions and achieved similar performance to the other probing techniques.

- Based on the experiment results, it appears that all four collision resolution techniques (Linear Probing, Chaining Approach, Quadratic Probing, and Double Hashing) performed equally well for the given dataset. All of them took only 2 milliseconds to resolve collisions and insert elements into the hash table.