Chapter 4: Hashing and Hashtable

4.1 Introduction

We have learned array and linked list as primary data structures and then learned how we can use those primary data structures to create Stack and Queue. We have also learned the merits and demerits of those different data structures. One very important aspect is that in both array and linked list, our data is stored against some index value for our using purpose.

Another important concept of storing data is using **key-value** pairs to store data instead of storing against an index. The idea is to find the data in constant time using that key. You have already seen this type of data structure in the Python dictionary. The concept is also used in JSON (JavaScript Object Notation). Now the question is how we can achieve this. The problem with our existing knowledge is that if we use linear search in the array to find the object it takes linear time O(n) to find it. If we use a sorted array to use the concept of counting sort it also takes linear time (for simplicity) to sort the array which is O(n).

4.2 Structure of Hashtable

The idea of hashing and hash table data structure comes to solve this issue. The idea of hashing is that there will be a function that will map any key as an integer, that integer will be used as an index of an array, and both key and value will be stored in that index. The array where both key and values are stored is known as a hash table. For example, we may want to store “name”: “Naruto Uzumaki” where the key is “name” and the value is “Naruto Uzumaki”. It will be stored in a hash table with length 5 (which means the array’s length is 5). Now the hash function will convert the string “name” and map it to an integer. E.g.: using summation ASCII value of “name” and then mod it by 5 (in order to get a valid index). We get the index (110 + 97 + 109 + 101) = 417 % 5 = 2. Now in that particular index, the array will store both “name” as key and “Naruto Uzumaki” as value. Now you may wonder why we are storing both key and value, the reason is we need to distinguish between different keys which map to the same index. To illustrate, if we have another key “mean”, it will also map to index 2 and so we need to store data in such a way that the values of the “name” key and “mean” key are stored separately.

Now the question comes to how to search an item if multiple keys are mapped into the same index. One approach might be to use a hash function in such a way that all the keys are mapped to different distinguished indexes (one-to-one function). However, the problem with such an approach is that it will waste lots of space because many indexes will remain empty. To deal with this problem, hash functions usually use modulus operation with the array size to get a valid index. This creates the problem of multiple keys getting mapped into the same index as discussed above. The event is known as Collision and the hash table has some mechanism to deal with collision handling.

4.3 Collision Handle

One of the popular methods of collision handling is called Forward Chaining. The idea of Forward Chaining is to store a linked list in the index. As a result, whenever a collision occurs, the value is stored at the beginning of the linked list in that particular index. It is stored at the beginning of the linked list to decrease the insertion time. Whenever we search for an item, it will traverse the list and check the key to find the value. One limitation of this method is that the search time is longer if collisions occur frequently. For this reason, choosing the appropriate hash function to minimize the collision is very important.

Another method for collision handling is to use Linear Probing. It stores the data in the next available index once the collision has occurred. However, this process increases the chance of collision and so another technique is to use double hashing. The idea of this method is to use another hash function if we find any collision. The first hash function determines the key and the second hash function determines the jump from the given key if collision occurs.

4.4 Hashtable Example

Now we will see an example of hash table generation using forward chaining as a collision technique. In this example, we'll use a basic hash function to map keys to their corresponding hash index and use linked lists to handle collisions using forward chaining.

Let's assume we have a hash table of size 5, and the hash function is a simple modulo operation to convert keys to hash indices.

Hash Function: hash(key) = key % 5

Now, let's insert some key-value pairs into the hash table:

Insert (Key: 12, Value: "Apple")
Insert (Key: 5, Value: "Orange")
Insert (Key: 17, Value: "Banana")
Insert (Key: 10, Value: "Grapes")
Insert (Key: 22, Value: "Watermelon")
Insert (Key: 15, Value: "Pineapple")
Here's the resulting hash table after each insertion:

Step 1: Insert (Key: 12, Value: "Apple") index 2

Index	Key-Value Pair
2	(12, "Apple")

Step 2: Insert (Key: 5, Value: "Orange") index 0

Index	Key-Value Pair
2	(12, "Apple")
0	(5, "Orange")

Step 3: Insert (Key: 17, Value: "Banana") Collision at index 2

Index	Key-Value Pair
2	(17, "Banana") -> (12, "Apple") #Forward chaining with collision
0	(5, "Orange")

Step 4: Insert (Key: 10, Value: "Grapes") Collision at index 0

Index	Key-Value Pair
2	(17, "Banana") -> (12, "Apple")
0	(10, "Grapes") -> (5, "Orange")

Step 5: Insert (Key: 22, Value: "Watermelon") Collision at index 2

Index	Key-Value Pair
2	(22, "Watermelon") -> (17, "Banana") -> (12, "Apple")
0	(10, "Grapes") -> (5, "Orange")

Step 6: Insert (Key: 15, Value: "Pineapple") - Collision at index 0

Index	Key-Value Pair
2	(22, "Watermelon") -> (17, "Banana") -> (12, "Apple")
0	(15, "Pineapple") -> (10, "Grapes") -> (5, "Orange")

In this example, when a collision occurs, the new key-value pair is added to the linked list at the same index (using forward chaining) instead of overwriting the existing entry. This way, all key-value pairs with the same hash index can be accessed by traversing the linked list.

Forward chaining is a simple and effective way to handle collisions in a hash table, and it ensures that all key-value pairs are retained without overwriting or losing any data.

4.5 Advance Topic

Cryptographic Hashing: Cryptographic hashing, also known as one-way hashing or hash function, is a fundamental concept in computer science and cryptography. It is a mathematical function that takes an input (or "message") of arbitrary size and produces a fixed-size string of characters, typically represented in hexadecimal format. This fixed-size output is often referred to as the hash value, hash code, or simply hash.

Key properties of cryptographic hashing functions are:

Deterministic: For a given input, the hash function always produces the same output. This property is crucial for consistency and predictability.

Irreversibility (One-way): The hash function is designed to be irreversible, meaning it should be computationally infeasible to derive the original input from its hash value. This property ensures that the original data remains secure even if the hash is known.

Collision resistance: It should be extremely difficult to find two different inputs that produce the same hash value. In other words, it should be computationally infeasible to create a collision intentionally.

Cryptographic hashing has numerous applications in information security and computer science, including but not limited to:

1. Password storage: Storing passwords securely by hashing them, preventing direct exposure of user passwords in databases.

2. Data integrity verification: Verifying that data has not been tampered with by comparing the hash of the original data with the computed hash of the received data.

3. Digital signatures: Used in digital signature algorithms to ensure authenticity and integrity of messages or documents.

4. Secure storage and retrieval: Employed in blockchain technology to create secure links between blocks of data.

Common cryptographic hashing algorithms include MD5 (Message Digest Algorithm 5), SHA-1 (Secure Hash Algorithm 1), SHA-256 (part of the Secure Hash Algorithm 2 family), and SHA-3. It's important to note that as computing power increases and new cryptographic vulnerabilities are discovered, older hashing algorithms may become less secure, and it's advisable to use the most up-to-date and robust hashing functions available.