Table of contents

Hashmap

Hashing refers to the process of generating a small sized output (that can be used as index in a table) from an input of typically large and variable size. Hashing uses mathematical formulas known as hash functions to do the transformation. This technique determines an index or location for the storage of an item in a data structure called Hash Table.

Hash Table Data Structure Overview

- It is one of the most widely used data structure after arrays.
- It mainly supports search, insert and delete in **O(1)** time on average which is more efficient than other popular data structures like arrays, Linked List and Self Balancing BST.
- We use hashing for dictionaries, frequency counting, maintaining data for quick access by key, etc.
- Real World Applications include Database Indexing, Cryptography, Caches, Symbol Table and Dictionaries.
- There are mainly two forms of hash typically implemented in programming languages.
    - **Hash Set** : Collection of unique keys (Implemented as Set in Python, Set in JavaScript, unordered_set in C++ and HashSet in Java.
    - **Hash Map** : Collection of key value pairs with keys being unique (Implemented as dictionary in Python, Map in JavaScript, unordered_map in C++ and HashMap in Java)

Need for Hash data structure

The amount of data on the internet is growing exponentially every day, making it difficult to store it all effectively. In day-to-day programming, this amount of data might not be that big, but still, it needs to be stored, accessed, and processed easily and efficiently. A very common data structure that is used for such a purpose is the Array data structure.

Now the question arises if Array was already there, what was the need for a new data structure! The answer to this is in the word "efficiency". Though storing in Array takes **O(1)** time, searching in it takes at least **O(log n)** time. This time appears to be small, but for a large data set, it can cause a lot of problems and this, in turn, makes the Array data structure inefficient.

So now we are looking for a data structure that can store the data and search in it in constant time, i.e. in **O(1)** time. This is how Hashing data structure came into play. With the introduction of the Hash data structure, it is now possible to easily store data in constant time and retrieve them in constant time as well.

Components of Hashing

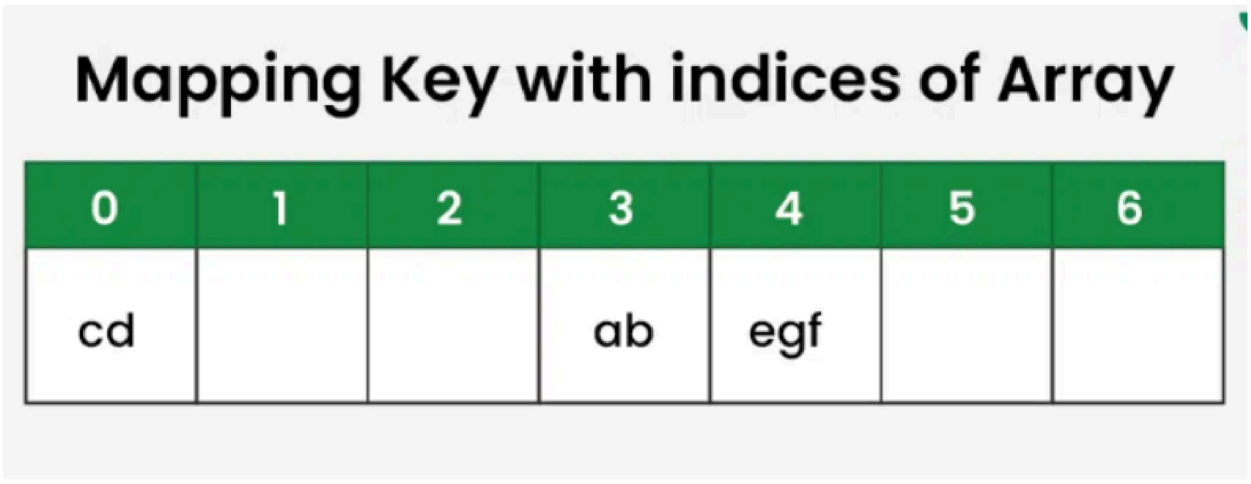
There are majorly three components of hashing:

1. **Key:** A Key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** Receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index .
3. **Hash Table:** Hash table is typically an array of lists. It stores values corresponding to the keys. Hash stores the data in an associative manner in an array where each data value has its own unique index.

How does Hashing work?

Suppose we have a set of strings {“ab”, “cd”, “efg”} and we would like to store it in a table.

- **Step 1:** We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.
- **Step 2:** So, let's assign
    - “a” = 1,
    - “b”=2, .. etc, to all alphabetical characters.
- **Step 3:** Therefore, the numerical value by summation of all characters of the string:
    - “ab” = 1 + 2 = 3,
    - “cd” = 3 + 4 = 7 ,
    - “efg” = 5 + 6 + 7 = 18
- **Step 4:** Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size** . We can compute the location of the string in the array by taking the **sum(string) mod 7** .
- **Step 5:** So we will then store
    - “ab” in 3 mod 7 = 3,
    - “cd” in 7 mod 7 = 0, and
    - “efg” in 18 mod 7 = 4.



Mapping Key with indices of Array

0	1	2	3	4	5	6
cd			ab	egf		

The above technique enables us to calculate the location of a given string by using a simple hash function and rapidly find the value that is stored in that location. Therefore the idea of hashing seems like a great way to store (key, value) pairs of the data in a table

What is a Hash function?

A hash function creates a mapping from an input key to an index in hash table, this is done through the use of mathematical formulas known as hash functions. For example: Consider phone numbers as keys and a hash table of size 100. A simple example hash function can be to consider the last two digits of phone numbers so that we have valid array indexes as output. A good hash function should have the following properties:

1. Efficient
2. Should uniformly distribute the keys to each index of hash table.
3. Should minimize collisions (This and the below are mainly derived from the above 2nd point)
4. Should have a low load factor (number of items in the table divided by the size of the table).

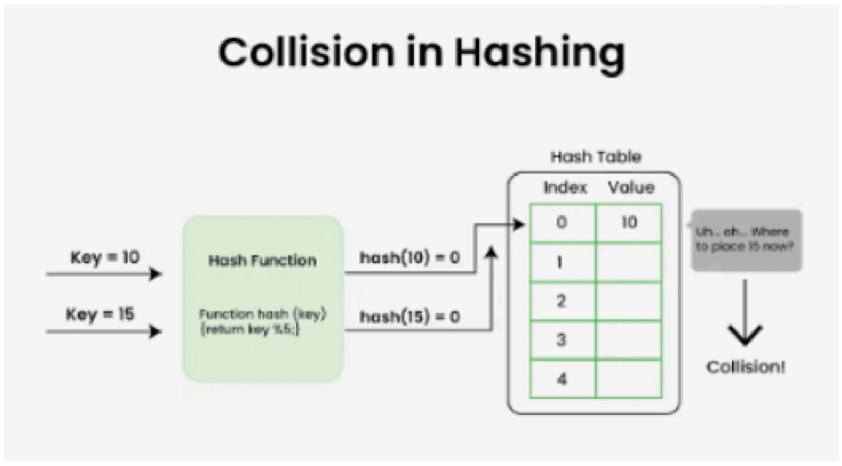
Advantages of Hashing in Data Structures

- **Key-value support:** Hashing is ideal for implementing key-value data structures.
- **Fast data retrieval:** Hashing allows for quick access to elements with constant-time complexity.
- **Efficiency:** Insertion, deletion, and searching operations are highly efficient.
- **Memory usage reduction:** Hashing requires less memory as it allocates a fixed space for storing elements.
- **Scalability:** Hashing performs well with large data sets, maintaining constant access time.
- **Security and encryption:** Hashing is essential for secure data storage and integrity verification.

What is Collision in Hashing?

When two or more keys have the same hash value, a **collision** happens. If we consider the above example, the hash function we used is the sum of the letters, but if we examined the hash function closely then the problem can be easily visualised that for different strings same hash value is being generated by the hash function.

For example: {“ab”, “ba”} both have the same hash value, and string {“cd”,”be”} also generate the same hash value, etc. This is known as **collision** and it creates problem in searching, insertion, deletion, and updating of value.



Collision in Hashing

The probability of a hash collision depends on the size of the algorithm, the distribution of hash values, and the efficiency of Hash function. To handle this collision, we use **Collision Resolution Techniques**.

What is meant by Load Factor in Hashing?

The load factor of the hash table can be defined as the number of items the hash table contains divided by the size of the hash table. Load factor is the decisive parameter that is used when we want to rehash the previous hash function or want to add more elements to the existing hash table.

It helps us in determining the efficiency of the hash function i.e. it tells whether the hash function which we are using is distributing the keys uniformly or not in the hash table.

Load Factor = Total elements in hash table/ Size of hash table

Ordered Map vs Unordered Map

Feature	**map**	**unordered_map**
Order	Sorted (by key)	No order
Time (Lookup)	O (log n)	O (1) average case
Use case	When order is needed	When speed is needed

C++ Built-In HashMap Functions

Function	Description
insert({key, value})	Inserts a new key-value pair into the map.
mp[key] = value	Inserts or updates the value associated with a key.
mp.count(key)	Returns 1 if key exists, 0 otherwise.
mp.find(key)	Returns an iterator to the key if found, else mp.end().
mp.erase(key)	Removes the key and its value from the map.
mp.clear()	Removes all elements from the map.
mp.empty()	Returns true if the map is empty.
mp.size()	Returns the number of key-value pairs in the map.
for (auto pair : mp)	Iterate through the entire map.