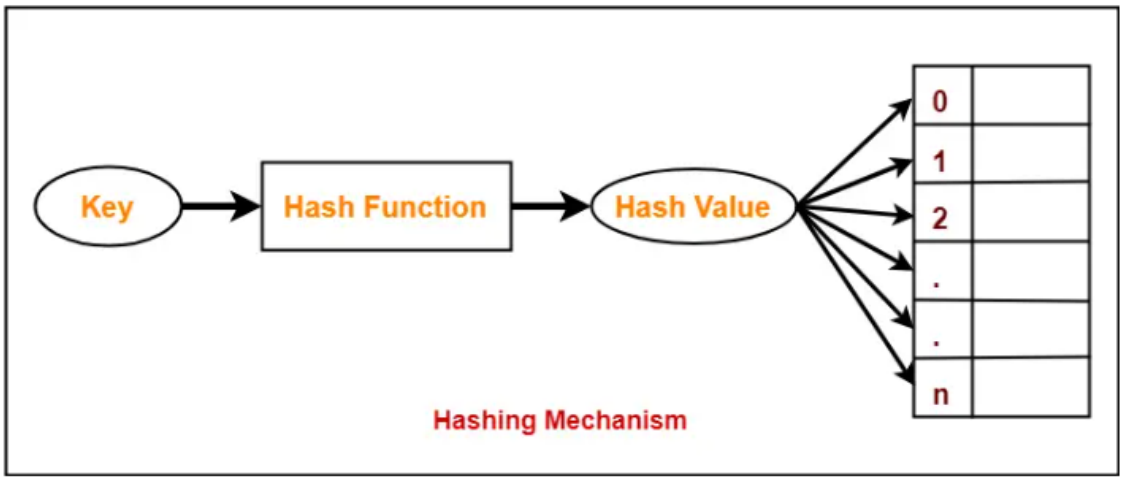
Hashing

Hashing involves utilising mathematical algorithms, known as **hash functions**, to produce a consistent and predetermined output size regardless of the size of the input. This method establishes an index or position for storing data within a data structure.



Need for Hash data structure

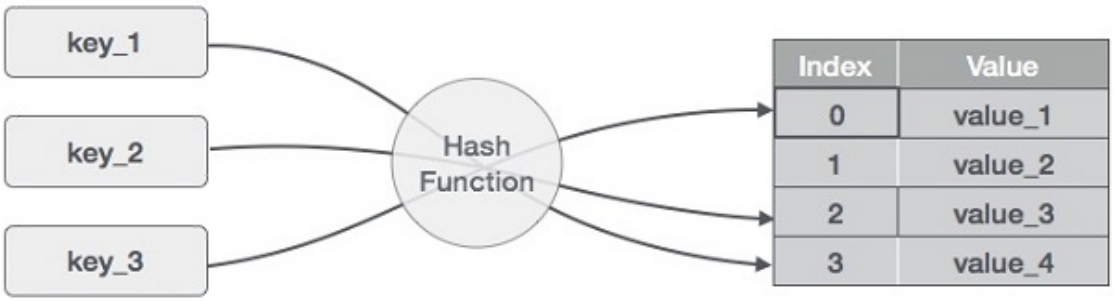
- **Efficient Data Retrieval**: Hashing allows for efficient data retrieval by generating a unique index or key for each data item. This index provides a direct path to the location where the item is stored, minimising the time required to search and retrieve it.
- **Fast Search Operations**: Hashing enables fast search operations by reducing the search space. Instead of examining every item in a collection, hash functions narrow down the search to a specific location or bucket, improving overall search performance.
- **Data Integrity and Security**: Hashing is commonly used for data integrity and security purposes. By generating a fixed-size hash value for a given input, it becomes possible to verify if the data has been altered or tampered with. Any modifications to the input will result in a different hash value, providing a reliable means of detecting changes.
- **Indexing and Databases**: Hashing is widely employed in indexing techniques used in databases. Hash functions facilitate quick indexing and retrieval of data based on specific criteria, such as primary keys or unique

identifiers. This accelerates database operations, including searching, sorting, and joining data.

- **Caching and Memoization:** Hashing is valuable in caching and memoization scenarios, where results of expensive calculations or data retrieval operations are stored for future reuse. Hashing allows efficient lookup and retrieval of previously computed results based on specific input parameters, reducing processing time and improving overall performance.

Components of Hashing

1. **Key:** The term"Key"refers to a string or integer that is provided as input to a hash function. This function is responsible for determining the index or storage location of an item within a data structure.
2. **Hash Function:** When the hash function takes the input key, it produces the hash index, which is essentially the index of an element within a designated array known as a hash table.
3. **Hash Table:** A hash table is a data structure that utilises a specific function, known as a hash function, to map keys to corresponding values. It organises and stores data in an array in such a way that each data value possesses its own distinct index, enabling an associative manner of data storage



Components of a hash hashing

What is a Hash function

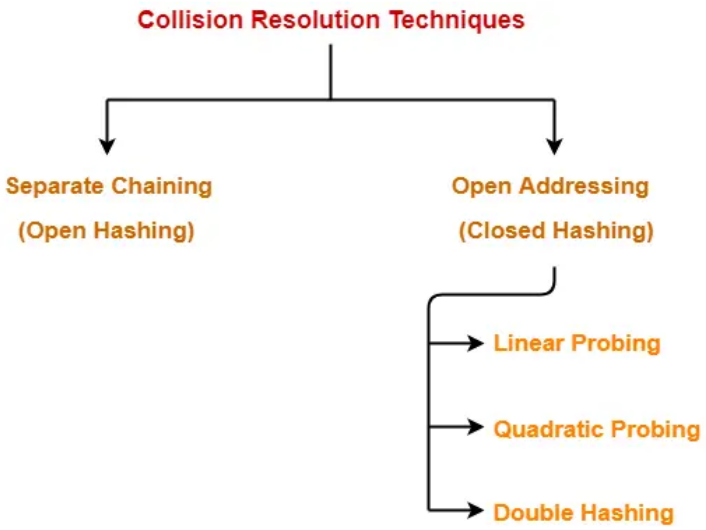
The hash function establishes a relationship between a key and its associated value by employing mathematical formulas called hash functions. The output generated by the hash function is commonly known as a hash value or simply a hash. **The hash value is a representation of the original string of characters but usually smaller than the original.**

What is a collision

Collisions can be problematic because they can lead to data loss or incorrect retrieval of information. Hashing finds frequent usage in data structures like hash tables or hash maps, facilitating the storage and retrieval of data based on its hash value. However, collisions can occur, leading to potential data overwriting or mixing, which can negatively impact the performance and accuracy of these data structures

How to handle Collisions

1. **Separate Chaining**
2. **Open Addressing**



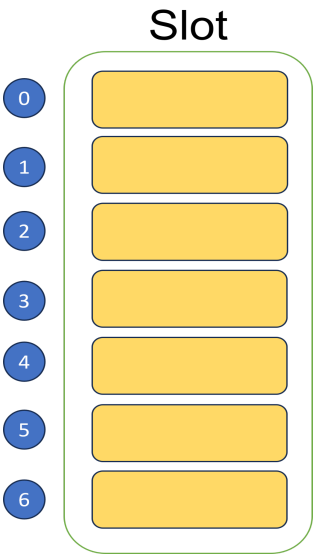
Separate Chaining

The concept revolves around assigning each cell of the hash table to a linked list consisting of records sharing the same hash function value. This approach, known as chaining, is straightforward but necessitates additional memory outside the table. In the context of inserting elements into the hash table, we are provided with a hash function and are tasked with incorporating the elements

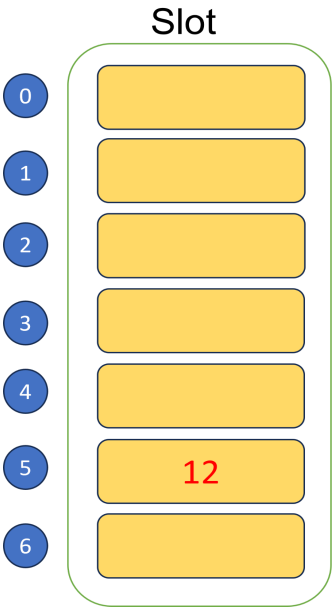
The linked list data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.

Hash function- key % 7, Element -12,22,50

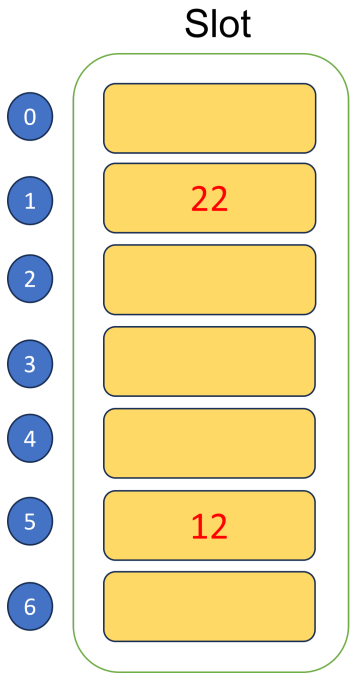
Step 1 : Draw the empty hash table which will have a possible range of hash values from 0 to 6 according to the hash function provided.



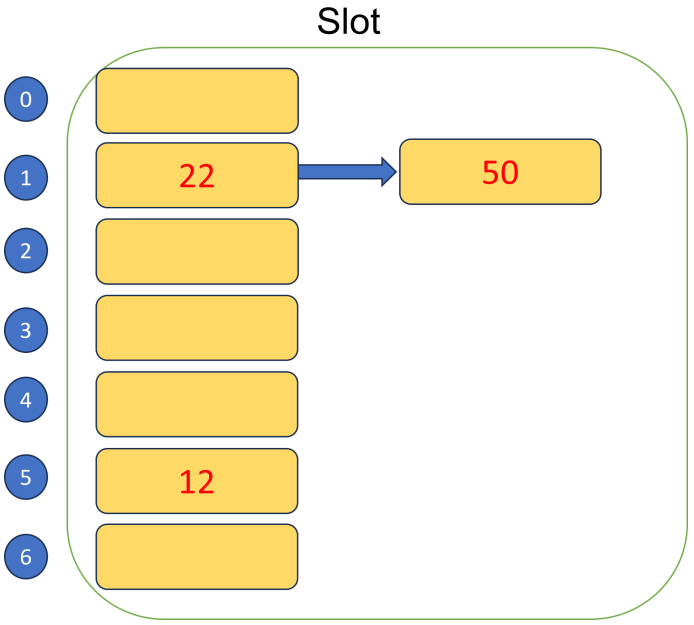
Step 2 The first key to be inserted is 12 which is mapped to bucket number 5 which is calculated by using the hash function 12%7=5.



Step 3: Now the next key is 22. It will map to the bucket number because 22%7=1.



Step 4 : Now, the next key is 50. It will map to the bucket number because 50 %7=1. But bucket 1 is already occupied.



Hence, in this way, the separate chaining method is used as the collision resolution technique.

Open Addressing

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

- Linear Probing
- Quadratic Probing
- Double Hashing

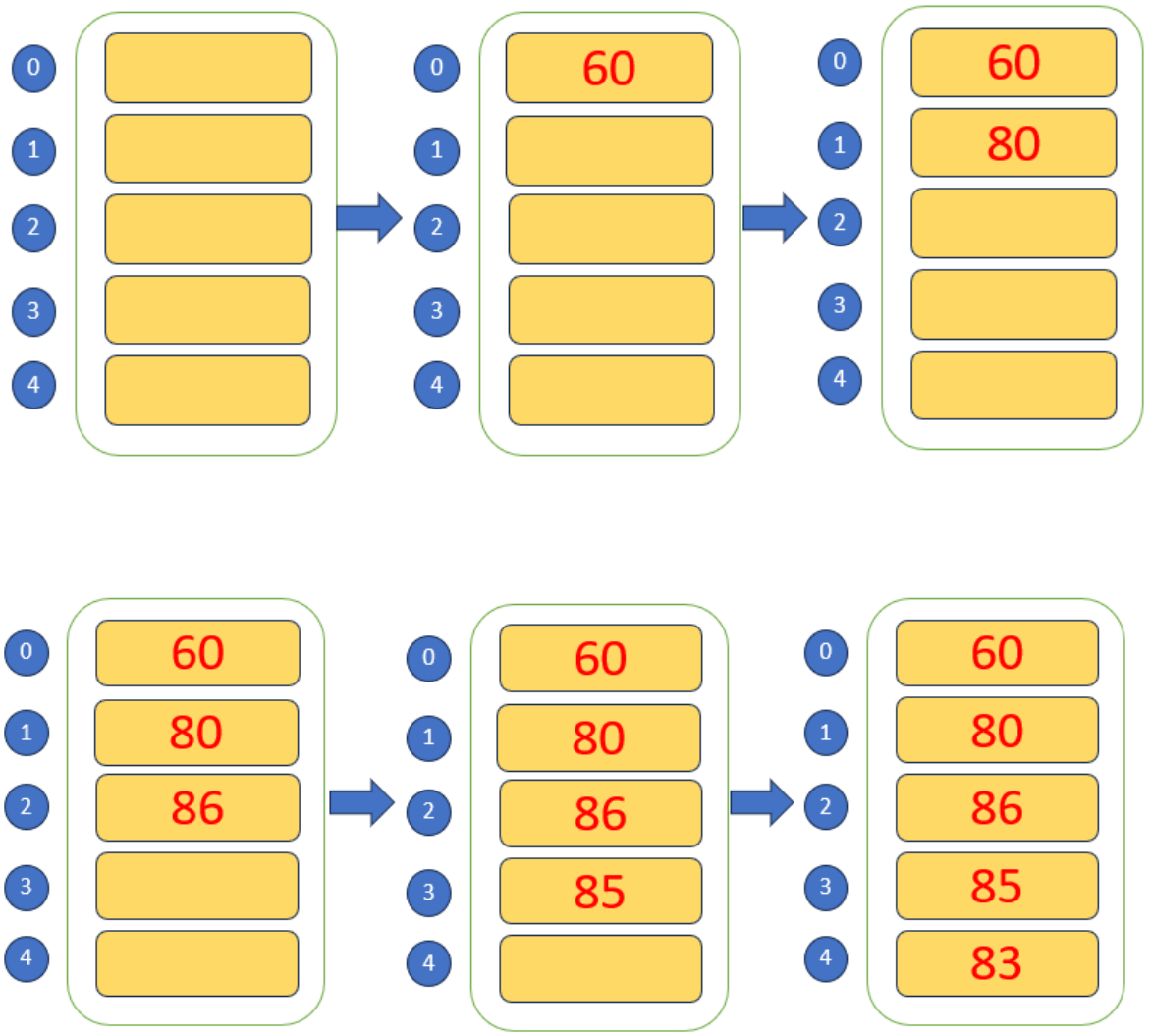
Linear Probing

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If the location that we get is already occupied, then we check for the next location.

Algorithm:

1. ***Find hash key using key = value % size of the bucket***
2. ***If hashtable[key] is empty -> hashtable[key]=data***
3. ***If hashtable[key] is not empty -> key = (key+1) % size***
4. ***Check if the next index is available hashtable[key] , then store the value . otherwise try for the next free index .***
5. ***Repeat the 1 to 4 points till we find the space .***

Problem statement : Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 60, 80, 86, 85, 83.



Quadratic Probing

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

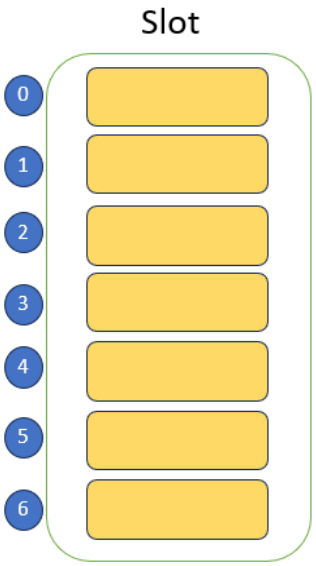
H + 1^2, H + 2^2, H + 3^2, ... H + k^2

Let hash(x) be the slot index computed using the hash function and n be the size of the hash table.

If the slot hash(x) % n is full, then we try (hash(x) + 12) % n.
If (hash(x) + 12) % n is also full, then we try (hash(x) + 22) % n.
If (hash(x) + 22) % n is also full, then we try (hash(x) + 32) % n.
This process will be repeated for all the values of i until an empty slot is found

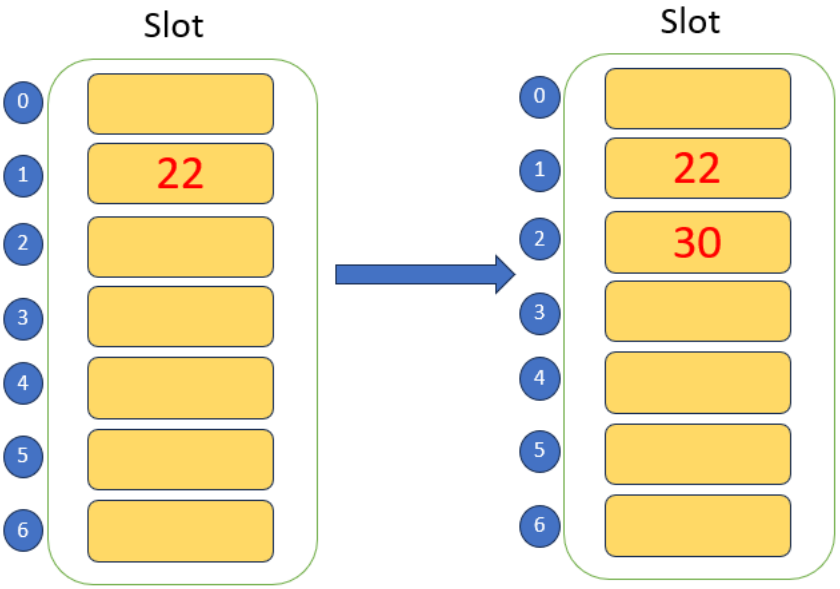
Example: Size of table = 7, hash function = x % 7 and collision resolution strategy to be f(i) = i2 . Insert = 22, 30, and 50

Step 1: Create an Empty table.

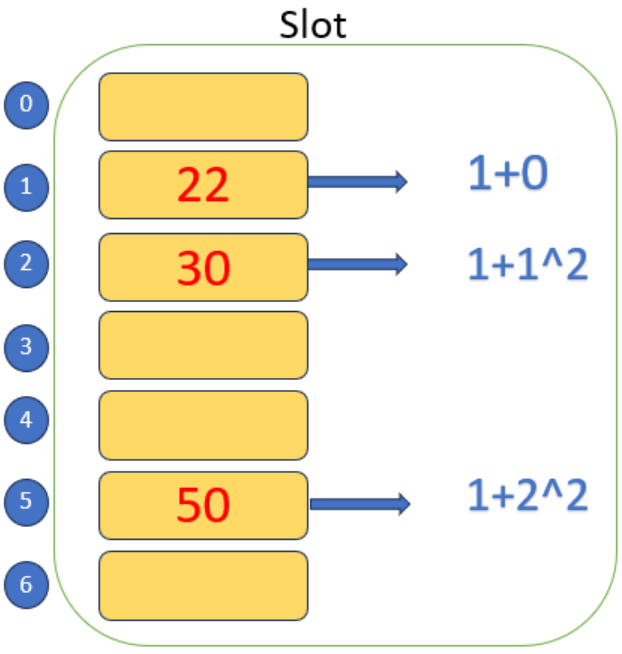


Step 2: Add elements 22 and 30 into the table.

- Hash (22): 22 % 7 = 1, Since the cell at index 1 is empty, we can insert 22 at slot 1.
- Hash (30) : 30 % 7 = 2, Since the cell at index 2 is empty, we can insert 30 at slot 2.



Step 3: Add elements 50 into the table. For this hash 50 % 7 =1 , hash table slot 1 is already occupied. So, we will search for slot 1+12, i.e. 1+1 = 2. Again slot 2 is found occupied, so we will search for cell 1+22, i.e.1+4 = 5. Now, cell 5 is not occupied so we will place 50 in slot 5.



Double hashing

Collision resolving technique in Open Addressed Hash tables. Double hashing makes use of two hash functions, The first hash function is **h1(k)** which takes the key and gives out a location on the hash table. But if the new location is not occupied or empty then we can easily place our key. But in case the location is occupied (collision) we will use a secondary hash-function **h2(k)** in combination with the first hash-function **h1(k)** to find the new location on the hash table.

This combination of hash functions is of the form

*h(k, i) = (h1(k) + i * h2(k)) % n*

where

- i is a non-negative integer that indicates a collision number,
- k = element/key which is being hashed
- n = hash table size.

Load Factor in Hashing

The load factor of a hash table refers to the ratio between the number of items stored in the hash table and its overall size. It serves as a critical parameter when considering scenarios such as rehashing with a new hash function or adding additional elements to the existing hash table.

The load factor plays a crucial role in evaluating the efficiency of the hash function. It indicates whether the distribution of keys in the hash table is uniform or not. By examining the load factor, we can assess how effectively the hash function is allocating and organising the keys within the hash table.

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

What is Rehashing?

Rehashing refers to the process of performing hashing again. When the load factor exceeds a predefined value (typically set at 0.75), it leads to increased complexity. To address this issue, the size of the array is expanded (doubled), and all the values are hashed once more and stored in the newly created array, which is twice the size of the original. This approach ensures that the load factor

remains low, thereby reducing complexity and maintaining efficient performance.

Advantages of Hashing

- Fast Data Retrieval: Hashing allows for fast retrieval of data from a collection, as it uses a hash function to directly map keys to their corresponding values. This results in constant-time average access time, making it efficient for large datasets.
- Memory Efficiency: Hashing is memory-efficient because it typically requires less memory compared to other data structures like arrays or linked lists. The memory usage is proportional to the number of elements stored and not the size of the data structure.
- Data Integrity: In cryptographic hashing, it provides a means to ensure data integrity. By generating a fixed-size hash value (digest) from the input data, any changes or tampering with the data will result in a completely different hash, allowing easy detection of alterations.
- Security Applications: Cryptographic hashing is widely used in security applications like password hashing. It ensures that even if the hash is leaked, it's challenging to reverse-engineer the original data, providing an additional layer of protection.

Disadvantages of Hashing:

- Collision Risk: Hashing can suffer from collisions, which occur when two different keys produce the same hash value. Collisions can lead to data loss or performance degradation if not handled properly.
- Lack of Order: Hash tables do not maintain any specific order among elements, which might be a disadvantage in some scenarios where the order of insertion or retrieval matters.
- Difficulty in Reconstructing Data: Unlike some other data structures, hashing does not easily support operations to reconstruct the original data or obtain a sorted list of elements, as the hash function is generally one-way.

- Hash Function Choice: In cryptographic hashing, the security of the hash relies heavily on the chosen hash function. If a weak or outdated hash function is used, it may be susceptible to attacks like collision attacks or preimage attacks.