DATA STRUCTURES & ALGORITHMS III

HASHING ALGORITHMS

TUTORIAL - 03

Hashing Algorithms

“ Hashing is a popular technique in computer science that involves **mapping large data sets** to **fixed-length values**. It is a process of converting a data set of **variable size** into a data set of a **fixed size**. The ability to perform **efficient lookup operations** makes hashing an essential concept in data structures.”

What is Hashing?

"A hashing algorithm is used to convert an **input** (such as a string or integer) into a **fixed-size output**. The data is then **stored** and retrieved using this **hash value** as an **index** in **an array** or **hash table**. The hash function must be deterministic, which guarantees that it will always yield the same result for a given input."

Real-World Applications of Hashing

- **Databases**

  Hashing is used in database **indexing** to quickly locate data without searching through **every row**.

- **Caches**

  Web caches and CPU caches use hashing to store **frequently accessed data** for quick retrieval.

- **Password Storage**

  Hash functions securely store **passwords** by converting them into hash codes.

- **Data De-duplication**

  Hashing helps in **identifying duplicate data** by comparing hash codes.

- **Load Balancing**

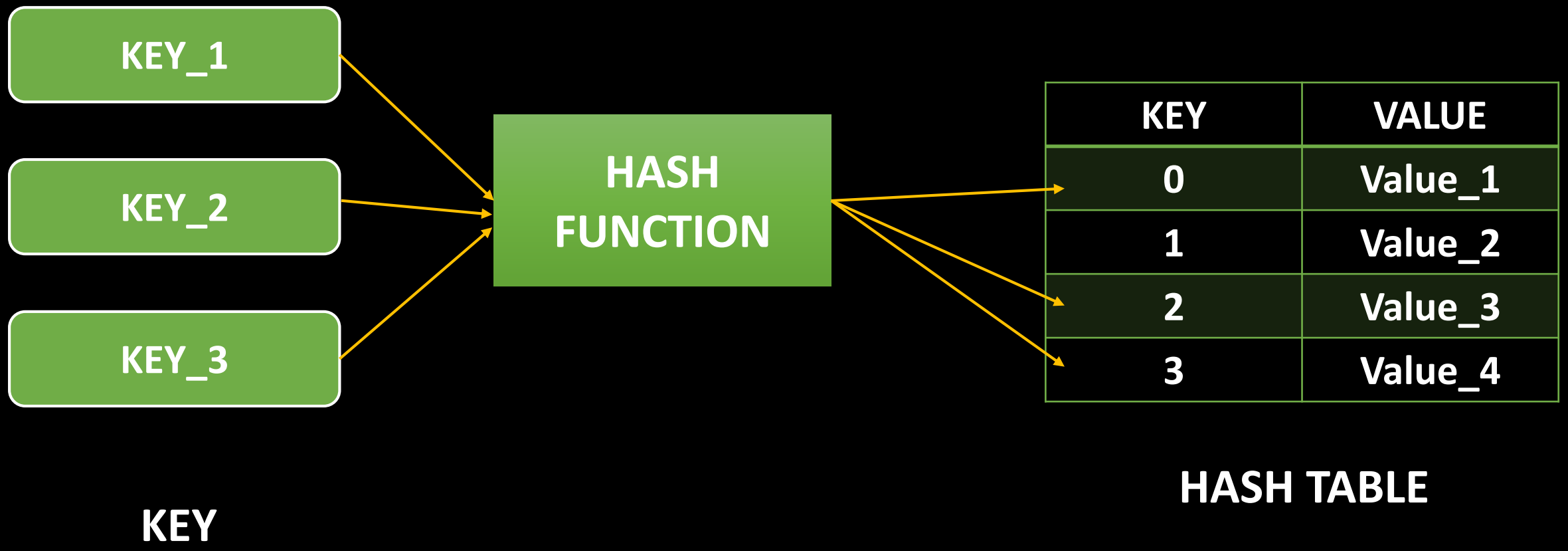
  Hashing algorithms **distribute network traffic** across multiple servers efficiently.

Need of Hashing Data Structure

Hashing is essential for various reasons:

- **Efficiency**: Provides **constant-time complexity (O(1))** for search, insert, and delete operations in the average case.
- **Speed**: Ideal for scenarios where speed is critical, such as databases and caching mechanisms.
- **Uniqueness**: Ensures unique storage locations for different keys through effective hash functions.

Components of Hashing



Components of Hashing

There are majorly three components of hashing:

- **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.
- **Hash Function:** The **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** .
- **Hash Table:** Hash table is a data structure that maps **keys** to **values** using a special function called a **hash function**. Hash stores the data in an associative manner in an array where each data value has its own **unique index**.

How Hashing Works?

The process of hashing can be broken down into three steps.

- **Input:** The data to be hashed is input into the **hashing algorithm**.
- **Hash Function:** The hashing algorithm takes the **input data** and applies a mathematical function to generate a **fixed-size hash value**. The hash function should be designed so that different input values produce different hash values, and small changes in the input produce large changes in the output.
- **Output:** The hash value is returned, which is used as **an index** to store or retrieve data in a data structure.

Types of Hash functions:

The hash function is a function that takes a key and returns an index into the hash table. The goal of a hash function is to distribute keys evenly across the hash table, minimizing collisions (when two keys map to the same index).

Common hash functions include:

Division Method: The division method involves dividing the key by **a prime number** and using the **remainder** as the hash value.

Multiplication Method: a constant AA (0 < A < 1) is used to multiply the key. The fractional part of the product is then multiplied by mm to get the hash value.

Mid-Square Method: in the mid-square method, the key is **squared**, and the middle digits of the result are taken as the hash value.

Example:

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

Objective: search or update the values stored in the table quickly in O(1) time and we are not concerned about the ordering of strings in the table.

Step 1: So, let’s assign

“a” = 1,

“b”=2, .. etc, to all alphabetical characters.

Step 2: Therefore, the numerical value by summation of all characters of the string:

“ab” = 1 + 2 = 3

“cd” = 3 + 4 = 7

Example:

Step 3: 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 4: So we will then store

"ab" in 3 mod 7 = 3,

"cd" in 7 mod 7 = 0,

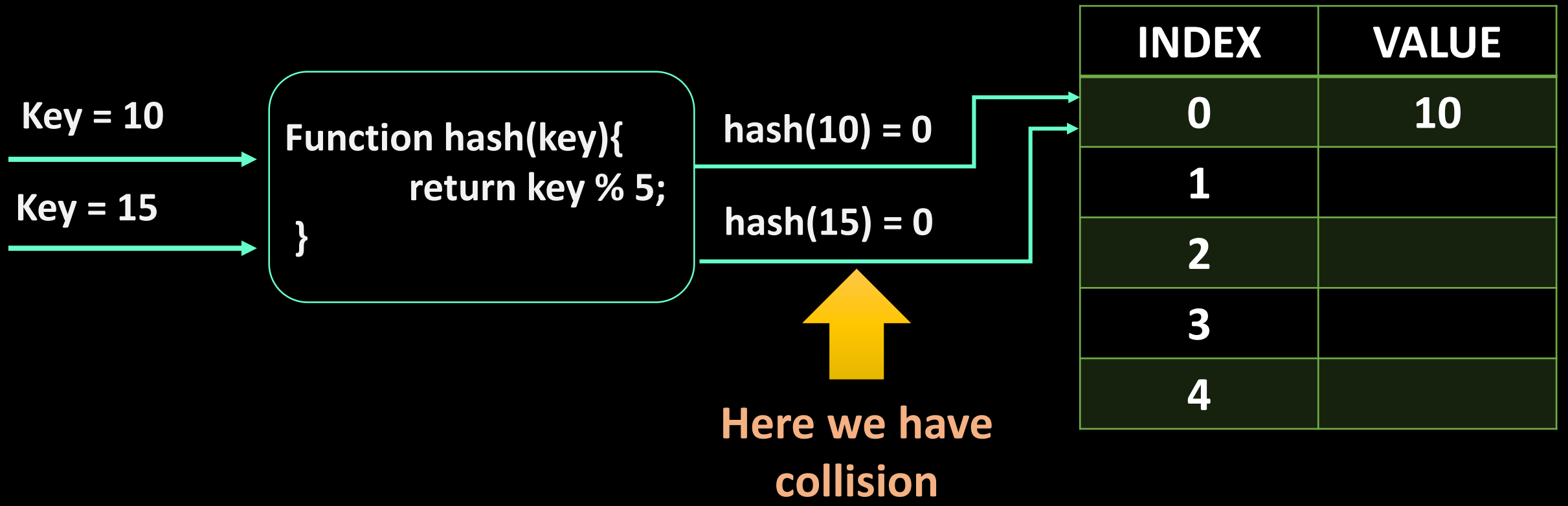
"efg" in 18 mod 7 = 4.

0	1	2	3	4	5	6
cd			ab	efg		

Problem with Hashing

One of the main challenges in hashing is handling collisions, which occur when two or more input values produce the same hash value.

This is known as **collision** and it creates problem in **searching**, **insertion**, **deletion**, and **updating** of value.



How to handle Collisions?

There are mainly two methods to handle collision:

1. Separate Chaining
2. Open Addressing
   - **Linear Probing**
   - **Quadratic Probing**
   - **Double Hashing**

1) Separate Chaining

The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

Example:

Hash function = key % 5,

Elements = 12, 15, 22, 25 and 37.

Step 01

0	
1	
2	
3	
4	

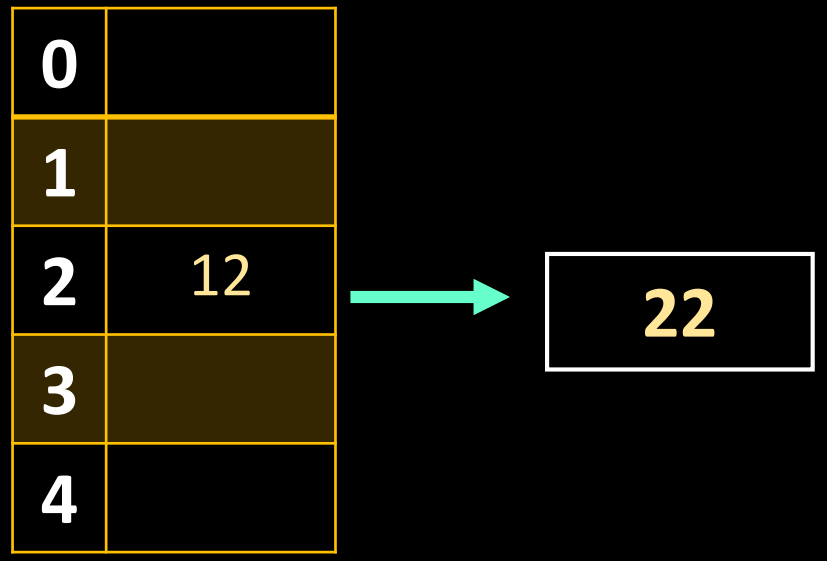
Empty hash table with range of hash values from **0 to 4** according to the hash function provided.

Step 02

0	
1	
2	12
3	
4	

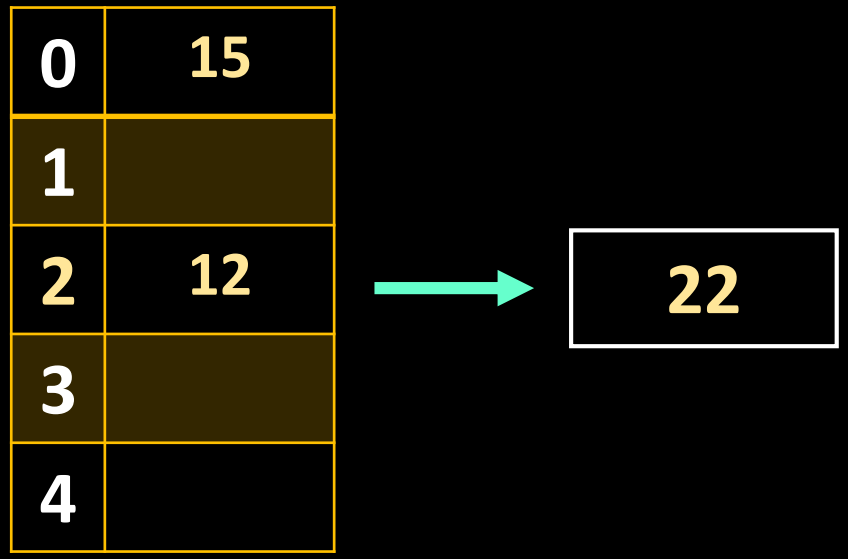
The first key to be inserted is 12 which is mapped to slot **2 (12%5)**

Step 03



The next key is 22 which is mapped to slot **2 (22%5)** but slot **2** is already occupied by key 12. Separate chaining handle this by creating a **linked list** to slot 2.

Step 04



The next key 15 which is mapped to slot **0 (15%5)**

2) 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.

2.a) Linear Probing

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

2.a) Linear Probing

1. Calculate the **hash key**. i.e. **key = data % size**
2. Check, if **hashTable[key]** is **empty**
   store the value directly by **hashTable[key] = data**
3. If the hash index already has **some value** then
   check for next index using **key = (key+1) % size**
4. Check, if the next index is available **hashTable[key]** then store the value. Otherwise try for next index.
5. Do the above process till we find the space.

2.a) Linear Probing

Example: Let us consider a simple hash function as **"key mod 5"** and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

Step 01

0	
1	
2	
3	
4	

Step 02

0	50
1	
2	
3	
4	

Empty hash table with range of hash values from **0 to 4** according to the hash function provided.

The first key to be inserted is 50 which is mapped to slot **0 (50%5)**

2.a) Linear Probing

Example: Let us consider a simple hash function as **"key mod 5"** and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

Step 03

0	50
1	70
2	
3	
4	

Step 04

0	50
1	70
2	76
3	
4	

The next key is 70 which is mapped to slot **0 (70%5)** but slot **0** is already occupied by key 50. so search for the next empty slot and insert it.

The next key is 76 which is mapped to slot **1 (76%5)** but slot **1** is already occupied by key 70. so search for the next empty slot and insert it.

2.a) Linear Probing

Example: Let us consider a simple hash function as **"key mod 5"** and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

Step 05

0	50
1	70
2	76
3	85
4	

The next key is 85 which is mapped to slot **0 (85%5)** but slot **0** is already occupied by key 50. so search for the next empty slot and insert it.

Step 04

0	50
1	70
2	76
3	85
4	93

The next key is 93 which is mapped to slot **3 (93%5)** but slot **3** is already occupied by key 85. so search for the next empty slot and insert it.

2.b) Quadratic Probing

Quadratic probing operates by taking the original hash index and adding successive **values of an arbitrary quadratic polynomial** until an open slot is found.
An example sequence using quadratic probing is:

$$H + 1^2, H + 2^2, H + 3^2, H + 4^2 \ldots\ldots\ldots\ldots\ldots H + k^2$$

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) + 1^2) % n**.
If (hash(x) + 1^2) % n is also full,
then we try **(hash(x) + 2^2) % n**.
If (hash(x) + 2^2) % n is also full,
then we try **(hash(x) + 3^2) % n**.

This process will be repeated for all the values of **i** until an empty slot is found

2.b) Quadratic Probing

Example: Let us consider **table Size = 7**, hash function as **Hash(x) = x % 7** and collision resolution strategy to be **f(i) = i ^{2}** . Insert = **22, 30, 50**

Step 01

0	
1	
2	
3	
4	
5	
6	

Empty hash table with range of hash values from **0 to 6** according to the hash function provided.

Step 02

0	
1	22
2	
3	
4	
5	
6	

The first key to be inserted is 22 which is mapped to slot **1 (22%7)**

2.b) Quadratic Probing

Example: Let us consider **table Size = 7**, hash function as **Hash(x) = x % 7** and collision resolution strategy to be **f(i) = i ^{2}** . Insert = **22, 30, 50**

Step 03

0	
1	22
2	30
3	
4	
5	
6	

The next key to be inserted is 30 which is mapped to slot **2 (30%7)**

Step 04

0	
1	22
2	30
3	
4	
5	50
6	

- Slot 1 + 1^2 = 2 **OCCUPIED**
- Slot 1 + 2^2 = 5 **NOT OCCUPIED**

The next key to be inserted is 50 which is mapped to slot **1 (50%7)** but already occupied.

2.C) Double Hashing

Double hashing make use of two hash function,

1. 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.
2. But in case the location is occupied (collision) we will use 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.

2.C) Double Hashing

Example: Insert the keys **27, 43, 692, 72** into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

Step 01

0	
1	
2	
3	
4	
5	
6	

Empty hash table with range of hash values from **0 to 6** according to the hash function provided.

Step 02

0	
1	
2	
3	
4	
5	
6	27

The first key to be inserted is 27 which is mapped to slot **6 (27%7)**

2.C) Double Hashing

Example: Insert the keys **27, 43, 692, 72** into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

Step 03

0	
1	43
2	
3	
4	
5	
6	27

The next key to be inserted is 43 which is mapped to slot **1 (43%7)**

Step 04

0	
1	43
2	
3	
4	
5	
6	27

The next key to be inserted is 692 which is mapped to slot **6 (692%7).** But it is occupied by 27

2.C) Double Hashing

Example: Insert the keys **27, 43, 692, 72** into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

Step 04

0	
1	43
2	692
3	
4	
5	
6	27

The next key to be inserted is 692 which is mapped to slot **6 (692%7).** But it is occupied by 27.

Hnew = [H1(692) + i* (H2(692))] % 7

= [6 + 1 * (1+692%5)] % 7

= 9 % 7

= **2**

Now as 2 is an empty slot you can insert 692 to the slot 2.

2.C) Double Hashing

Example: Insert the keys **27, 43, 692, 72** into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

Step 05

0	
1	43
2	692
3	
4	
5	72
6	27

The next key to be inserted is 72 which is mapped to slot **2 (72%7).** But it is occupied by 27.

Hnew = [H1(72) + i* (H2(72))] % 7

= [2 + 1 * (1+72%5)] % 7

= 5 % 7

= **5**

Now as 5 is an empty slot you can insert 72 to the slot 5.

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

What is Rehashing?

As the name suggests, rehashing means **hashing again**. Basically, when the load factor increases to more than its **predefined value** (the default value of the load factor is 0.75), the **complexity increases**.

So to overcome this, the **size of the array** is increased (doubled) and all the values are **hashed again** and **stored in the new double-sized array** to maintain a low load factor and low complexity.