Chapter 6
Searching Techniques

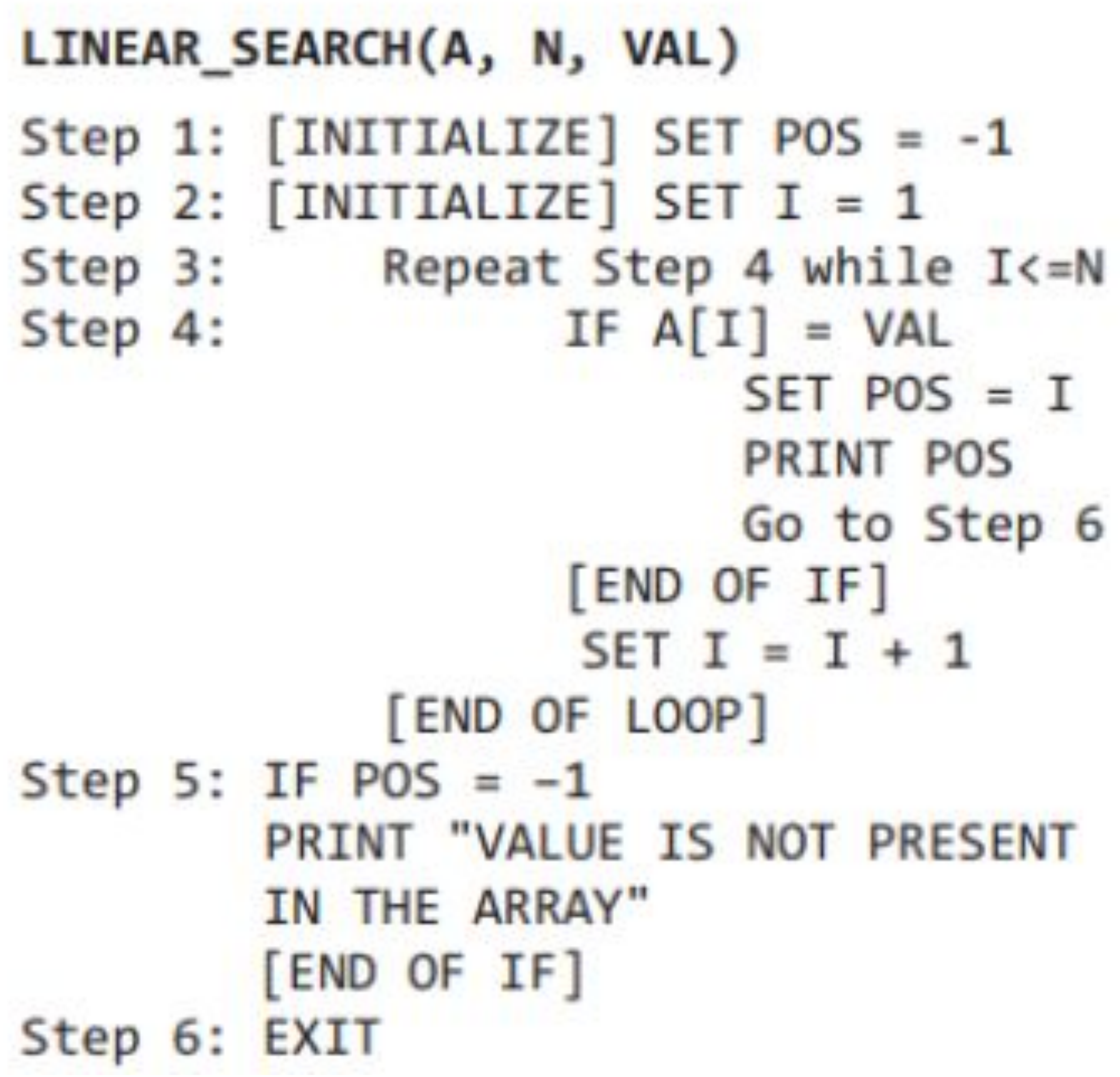
Searching

- Searching means to find whether a particular value is present in the array or not.
- If the value is present in the array then searching is said to be successful and the position of that value in the array is returned.
- If the value is not present then the searching is said to be unsuccessful.
- Two popular methods for searching :
  Linear Search
  Binary Search

Linear Search(Sequential Search)

- Works by comparing the value to be searched with every element of the array one by one in sequence until a match is found.
- Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).
- eg. int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5} and value to be searched is 7, then it returns POS = 3.
- Linear search executes in O(n) time where n is the number of elements in the array.

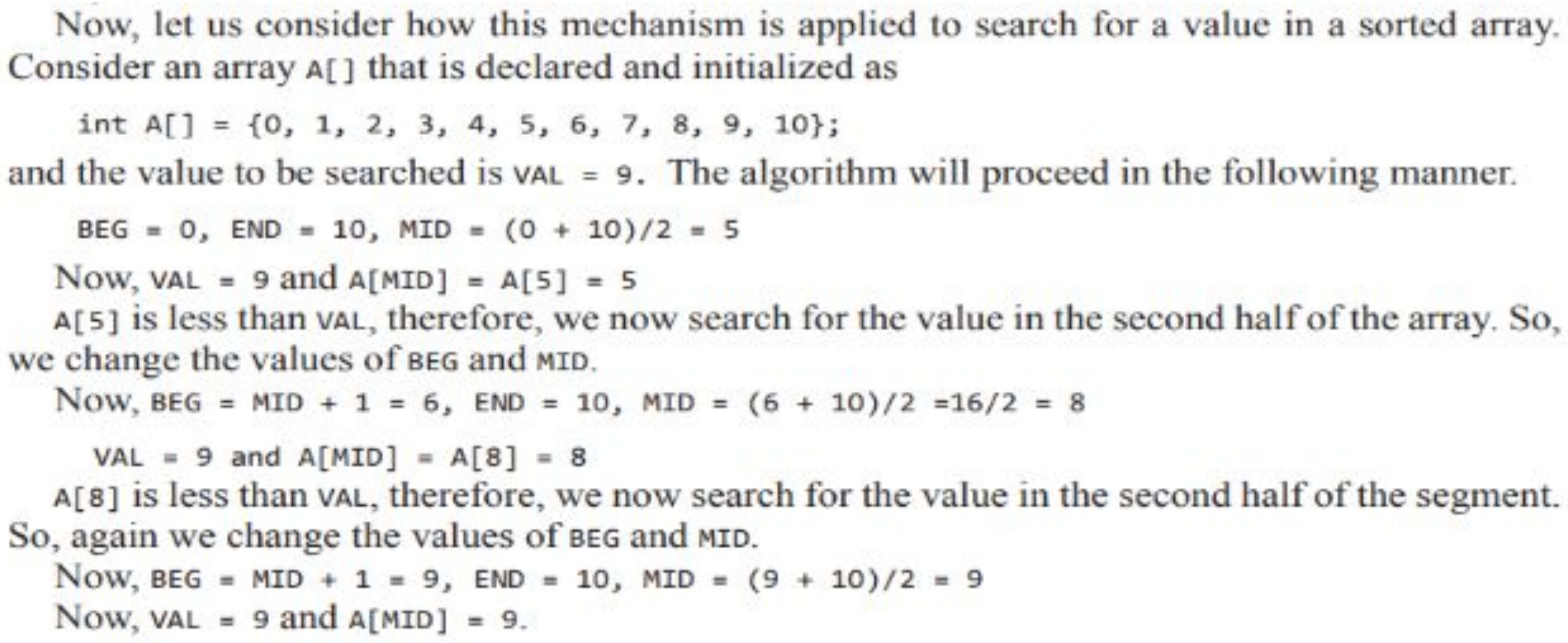
Algorithm



```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:             IF A[I] = VAL
                            SET POS = I
                            PRINT POS
                            Go to Step 6
                    [END OF IF]
                     SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Binary Search

- Binary search is a searching algorithm that works efficiently with a sorted list.



Now, let us consider how this mechanism is applied to search for a value in a sorted array. Consider an array `A[]` that is declared and initialized as

```
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

and the value to be searched is `VAL = 9`. The algorithm will proceed in the following manner.

```
BEG = 0, END = 10, MID = (0 + 10)/2 = 5
```

Now, `VAL = 9` and `A[MID] = A[5] = 5`

`A[5]` is less than `VAL`, therefore, we now search for the value in the second half of the array. So, we change the values of `BEG` and `MID`.

Now, `BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8`

```
VAL = 9 and A[MID] = A[8] = 8
```

`A[8]` is less than `VAL`, therefore, we now search for the value in the second half of the segment. So, again we change the values of `BEG` and `MID`.

Now, `BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9`

Now, `VAL = 9` and `A[MID] = 9`.

Working of Binary Search

In this algorithm, we see that `BEG` and `END` are the beginning and ending positions of the segment that we are looking to search for the element. `MID` is calculated as `(BEG + END)/2`. Initially, `BEG = lower_bound` and `END = upper_bound`. The algorithm will terminate when `A[MID] = VAL`. When the algorithm ends, we will set `POS = MID`. `POS` is the position at which the value is present in the array.

However, if `VAL` is not equal to `A[MID]`, then the values of `BEG`, `END`, and `MID` will be changed depending on whether `VAL` is smaller or greater than `A[MID]`.

(a) If `VAL < A[MID]`, then `VAL` will be present in the left segment of the array. So, the value of `END` will be changed as `END = MID - 1`.

(b) If `VAL > A[MID]`, then `VAL` will be present in the right segment of the array. So, the value of `BEG` will be changed as `BEG = MID + 1`.

Finally, if `VAL` is not present in the array, then eventually, `END` will be less than `BEG`. When this happens, the algorithm will terminate and the search will be unsuccessful.

Algorithm

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:           SET MID = (BEG + END)/2
Step 4:           IF A[MID] = VAL
                        SET POS = MID
                        PRINT POS
                        Go to Step 6
                  ELSE IF A[MID] > VAL
                        SET END = MID - 1
                  ELSE
                        SET BEG = MID + 1
                  [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Hashing

- In all search techniques like linear search, binary search and search trees, the time required to search an element depends on the total number of elements present in that data structure.
- In all these search techniques, as the number of elements increases the time required to search an element also increases linearly.
- Hashing is another approach in which time required to search an element doesn't depend on the total number of elements.
- Using hashing data structure, a given element is searched with **constant time complexity**. Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

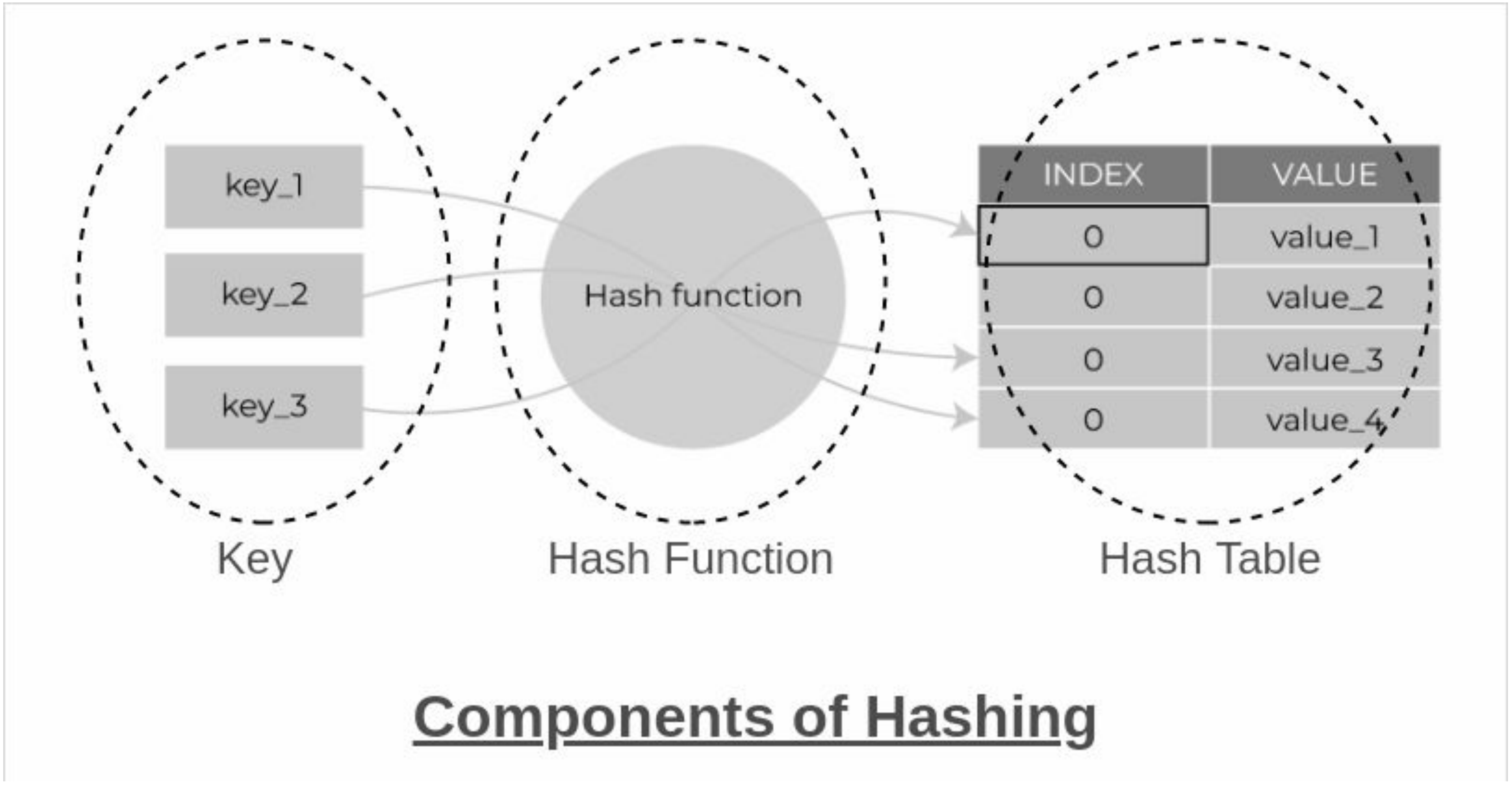
Hashing

- Hashing is defined as follows...
  - **Hashing is the process of indexing and retrieving element (data) in a data structure to provide a faster way of finding the element using a hash key.**
- Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.
- In this data structure, we use a concept called **Hash table** to store data. All the data values are inserted into the hash table based on the hash key value.
- The hash key value is used to map the data with an index in the hash table. And the hash key is generated for every data using a **hash function**. That means every entry in the hash table is based on the hash key value generated using the hash function.

Hashing

- Hash Table is defined as follows...
  - **Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations are performed with constant time complexity (i.e. O(1)).**
- Hash tables are used to perform insertion, deletion and search operations very quickly in a data structure.
- Using hash table concept, insertion, deletion, and search operations are accomplished in constant time complexity.
- Generally, every hash table makes use of a function called **hash function** to map the data into the hash table.
- A hash function is defined as follows...
- Hash function is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table.

Hashing Concept-Components of Hashing



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:** 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**.
3. **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.

Types of Hash functions

There are many hash functions that use numeric or alphanumeric keys.

1. Division Method.

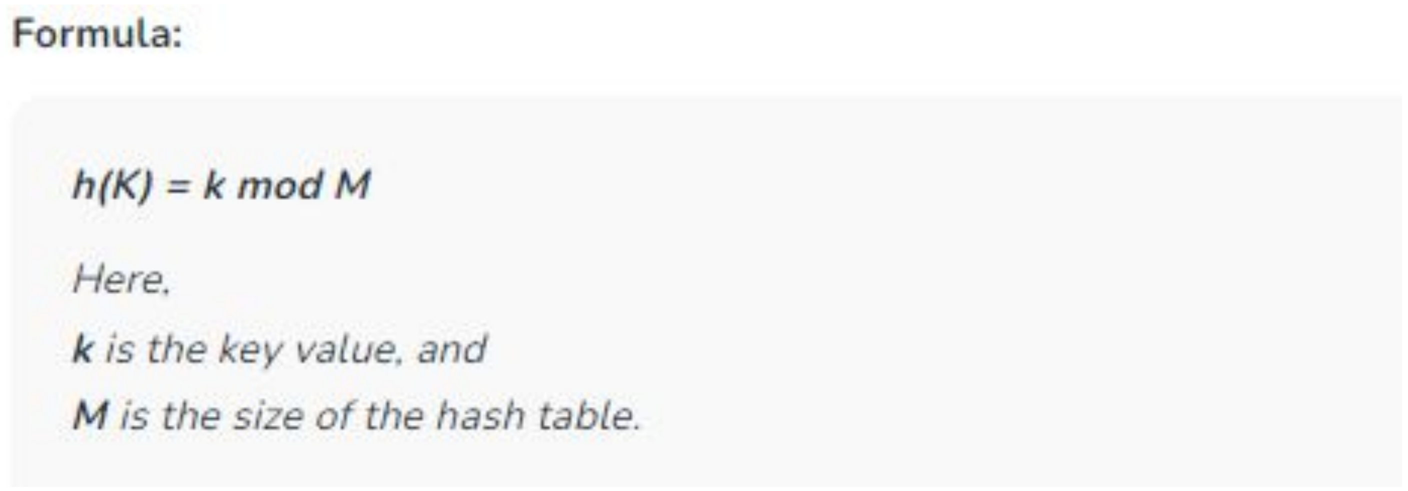
2. Mid Square Method.

3. Folding Method.

4. Multiplication Method.

Division Method:

- This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.



Formula:

h(K) = k mod M

Here,

***k** is the key value, and*

***M** is the size of the hash table.*

- It is best suited that **M** is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

Example:

$$k = 12345$$
$$M = 95$$
$$\begin{aligned} h(12345) &= 12345 \bmod 95 \\ &= 90 \end{aligned}$$

$$k = 1276$$
$$M = 11$$
$$\begin{aligned} h(1276) &= 1276 \bmod 11 \\ &= 0 \end{aligned}$$

Pros:

1. This method is quite good for any value of M.
2. The division method is very fast since it requires only a single division operation.

Cons:

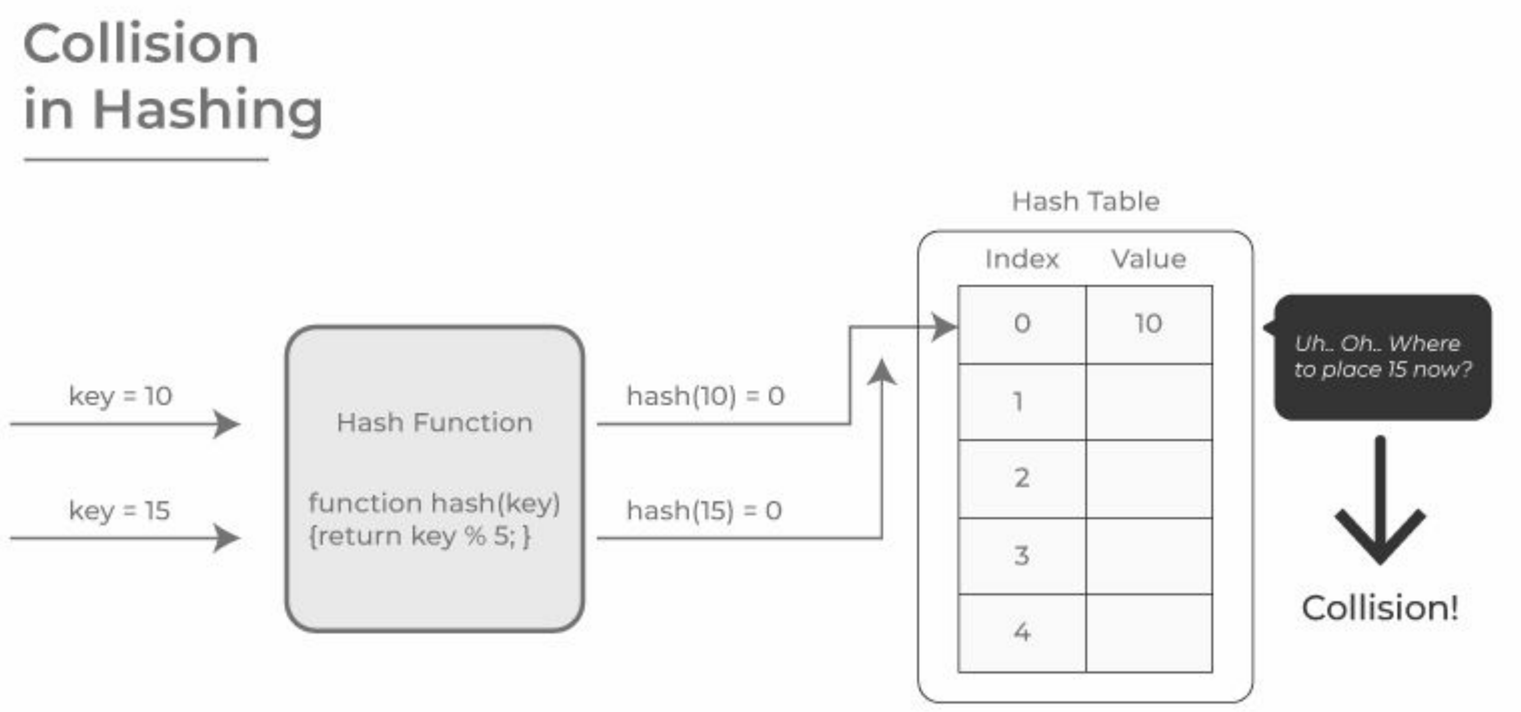
1. This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
2. Sometimes extra care should be taken to choose the value of M.

Problem with Hashing

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

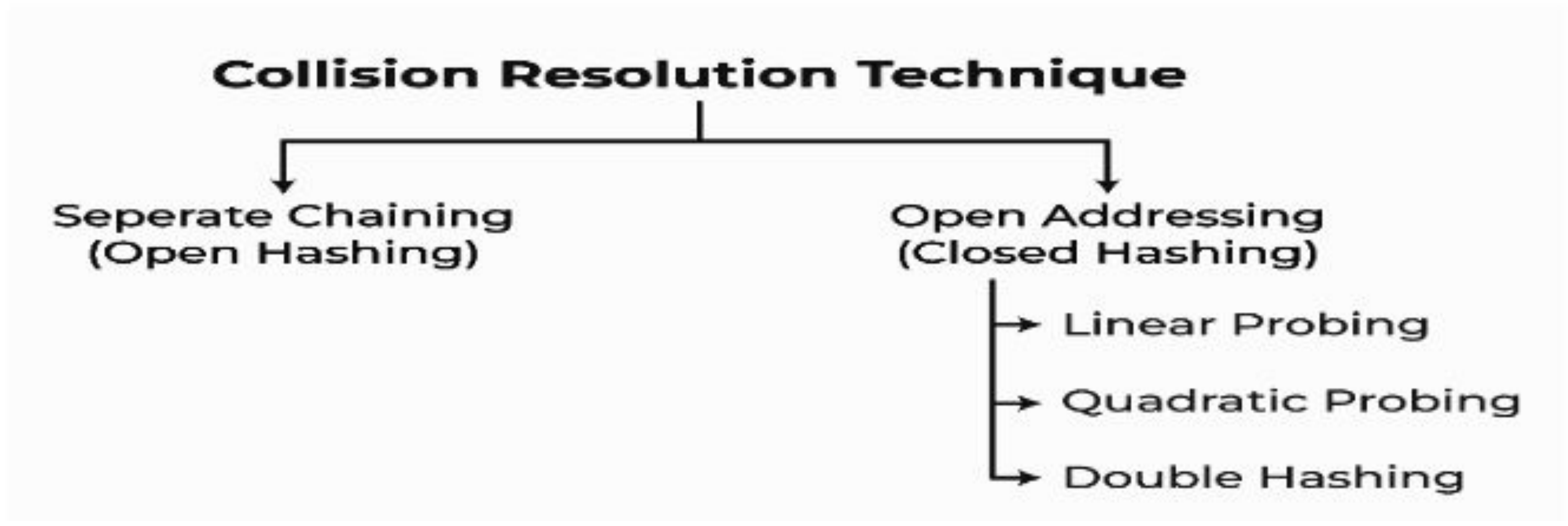
- The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.



How to handle Collisions?

There are mainly two methods to handle collision:

1. Separate Chaining:
2. Open Addressing:



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: We have given a hash function and we have to insert some elements in the hash table using a separate chaining method for collision resolution technique.

Hash function = key % 5,

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

- Let's see step by step approach to how to solve the above problem:
- **Step 1:** First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.
- **Step 2:** Now insert all the keys in the hash table one by one. The first key to be inserted is 12 which is mapped to bucket number 2 which is calculated by using the hash function 12%5=2.

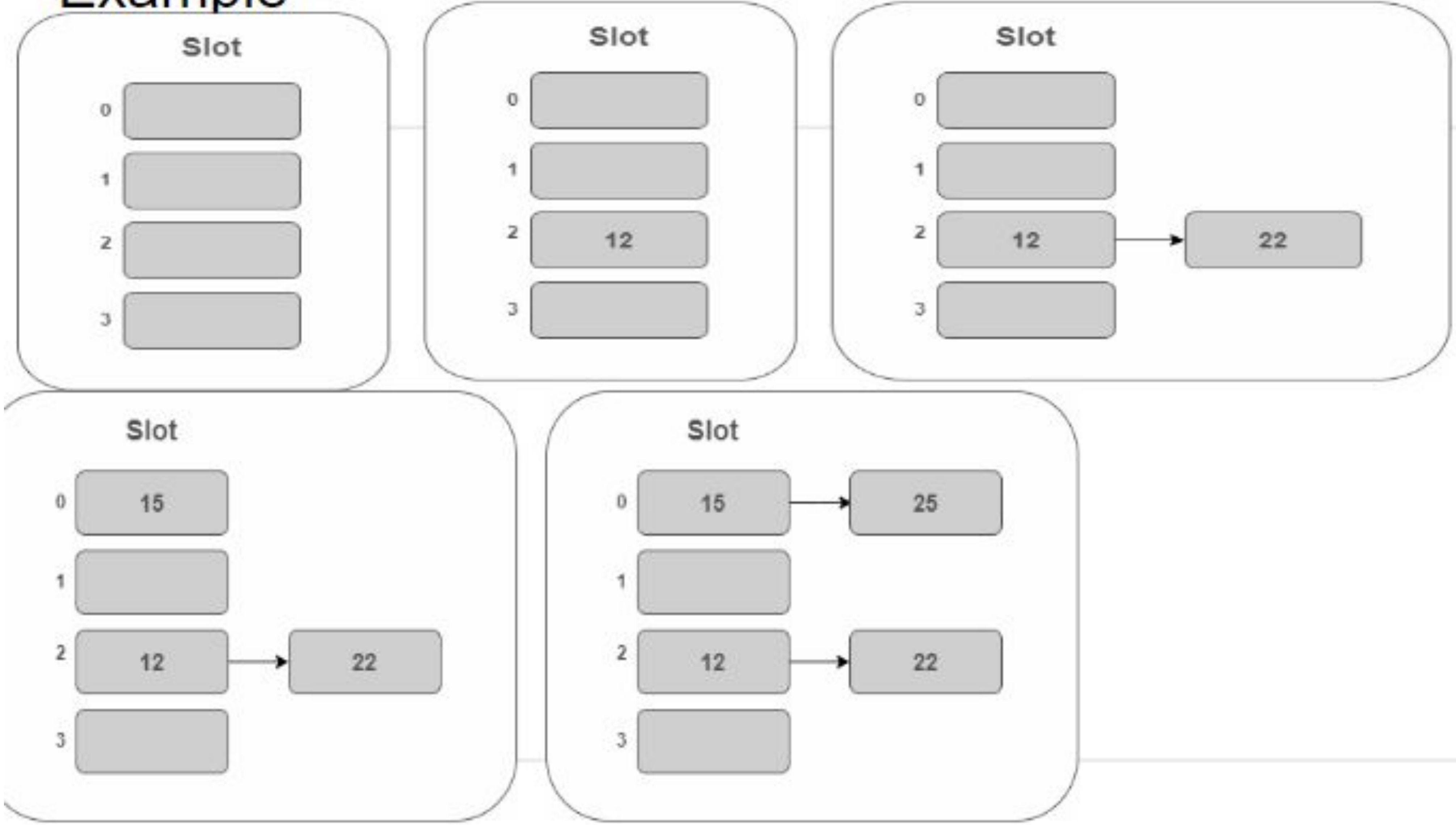
Separate Chaining

Step 3: The next key is 15. It will map to slot number 0 because 15%5=0. So insert it into bucket number 5.

Step 4: Now the next key is 22. It will map to bucket number 2 because 22%5=2. But bucket 2 is already occupied by key 12. So separate chaining method will handle this collision by creating a linked list to bucket 2

Step 5: Now the next key is 25. Its bucket number will be 25%5=0. But bucket 0 is already occupied by key 25. So separate chaining method will again handle the collision by creating a linked list to bucket 0.

Example


Slot
0
1
2
3
Slot
0
1
2
12
3
Slot
0
1
2
12
22
3
Slot
0
15
1
2
12
22
3
Slot
0
15
25
1
2
12
22
3

Open Addressing

- In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:
    - ***Insert(k):*** *Keep probing until an empty slot is found. Once an empty slot is found, insert k.*
    - ***Search(k):*** *Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.*
    - ***Delete(k)***: ***Delete operation is interesting***. *If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted".*

      The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

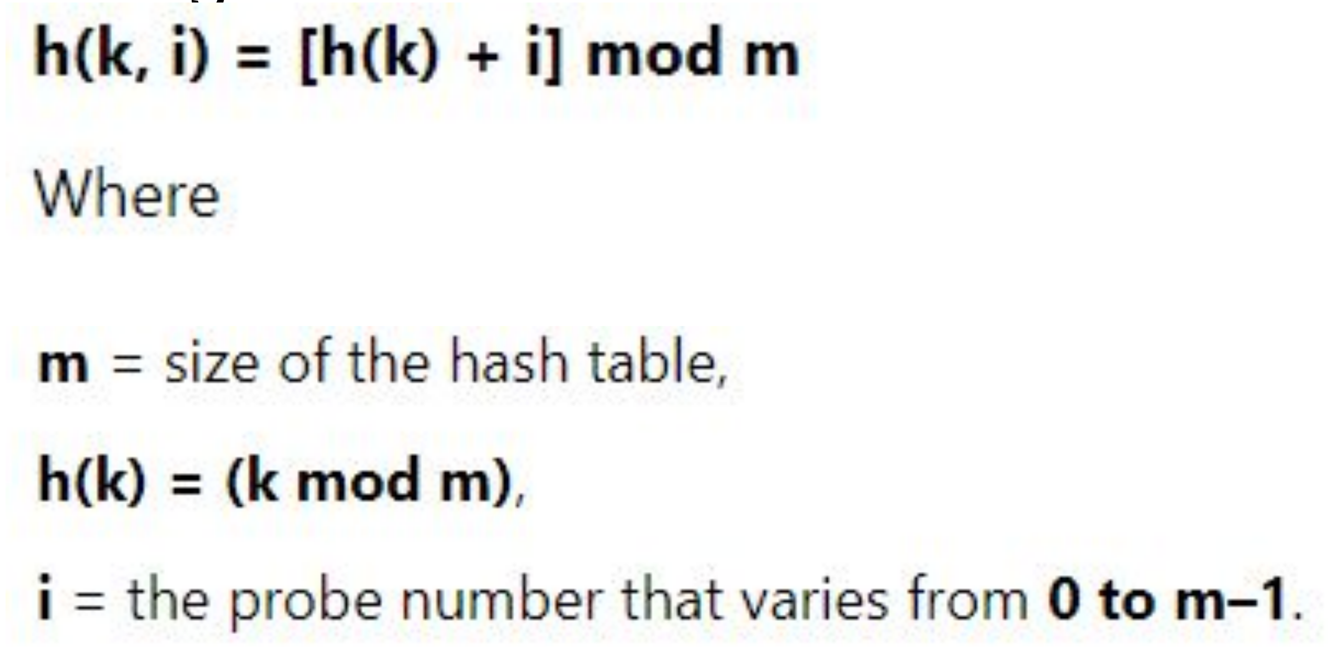
Linear Probing

- The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by h(k), it means collision occurred then we do a sequential search to find the empty location.
- Here the idea is to place a value in the next available position. Because in this approach searches are performed sequentially so it’s known as linear probing.
- Here array or hash table is considered circular because when the last slot reached an empty location not found then the search proceeds to the first location of the array.

Below is a hash function that calculates the next location.

If the location is empty, then store value otherwise find the next location.

Following hash function is used to resolve the collision in:



h(k, i) = [h(k) + i] mod m

Where

m = size of the hash table,

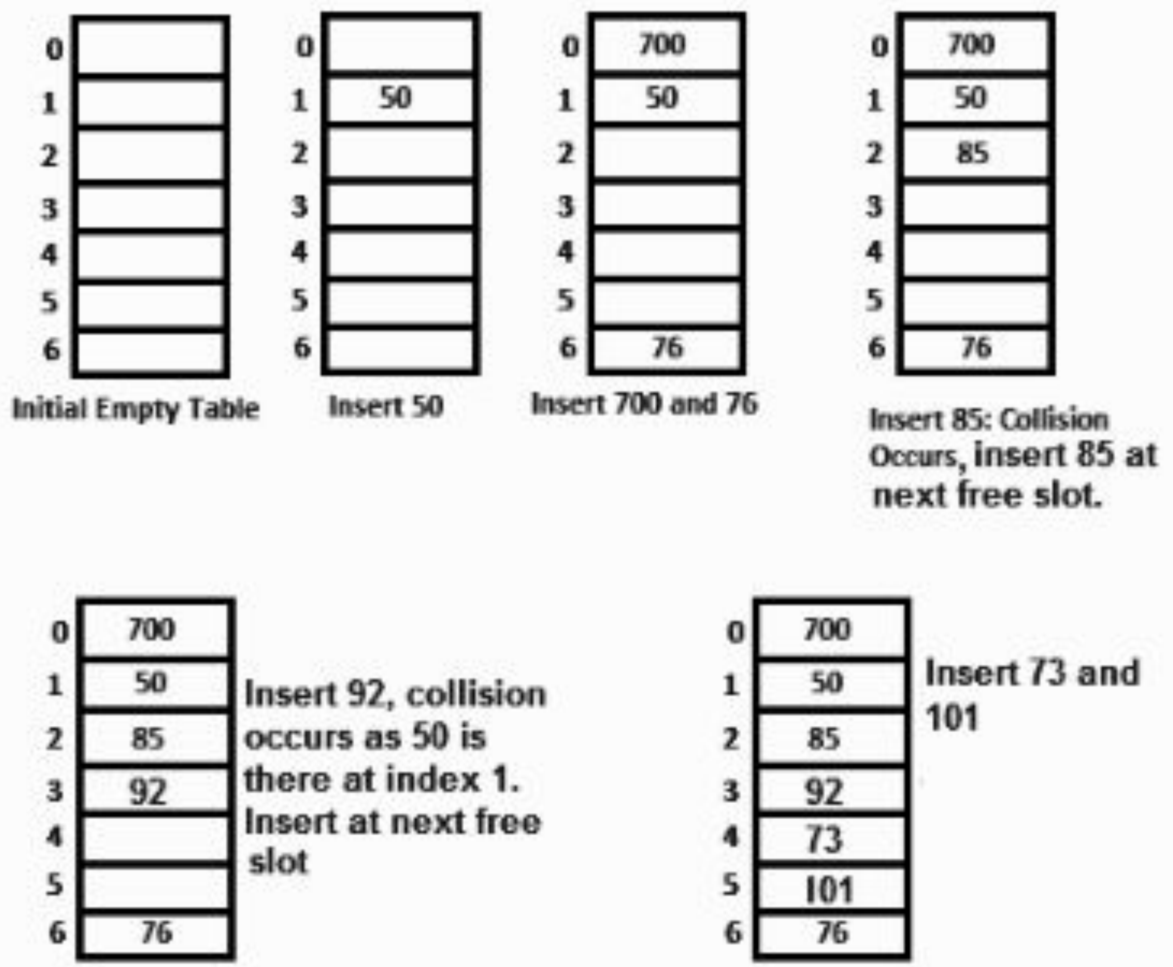
h(k) = (k mod m),

i = the probe number that varies from **0 to m–1**.

- Therefore, for a given key k, the first location is generated by **[h(k) + 0]** mod m, the first time i=0.
- If the location is free, the value is stored at this location. If value successfully **stores** then probe count is 1 means location is founded on the first go.
  If location is not free then second probe generates the address of the location given by **[h(k) + 1]mod m**.
  Similarly, if the generated location is occupied, then subsequent probes generate the address as **[h(k) + 2]mod m**, **[h(k) + 3]mod m**, **[h(k) + 4]mod m**, **[h(k) + 5]mod m**, and so on, until a free location is found.

Example

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.


Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, insert 85 at next free slot.
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
Insert 73 and 101
0 1 2 3 4 5 6
700
50
85
92
73
101
76

Linear Probing Example

Insert the following sequence of keys in the hash table of size 10

{72,27,36,24,63,81,92,101}

Quadratic Probing

- Quadratic probing is a method with the help of which we can solve the problem of clustering. This method is also known as the **mid-square** method. In this method, we look for the **i2‘th** slot in the **ith** iteration.
- We always start from the original hash location. If only the location is occupied then we check the other slots.

Let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S

If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S

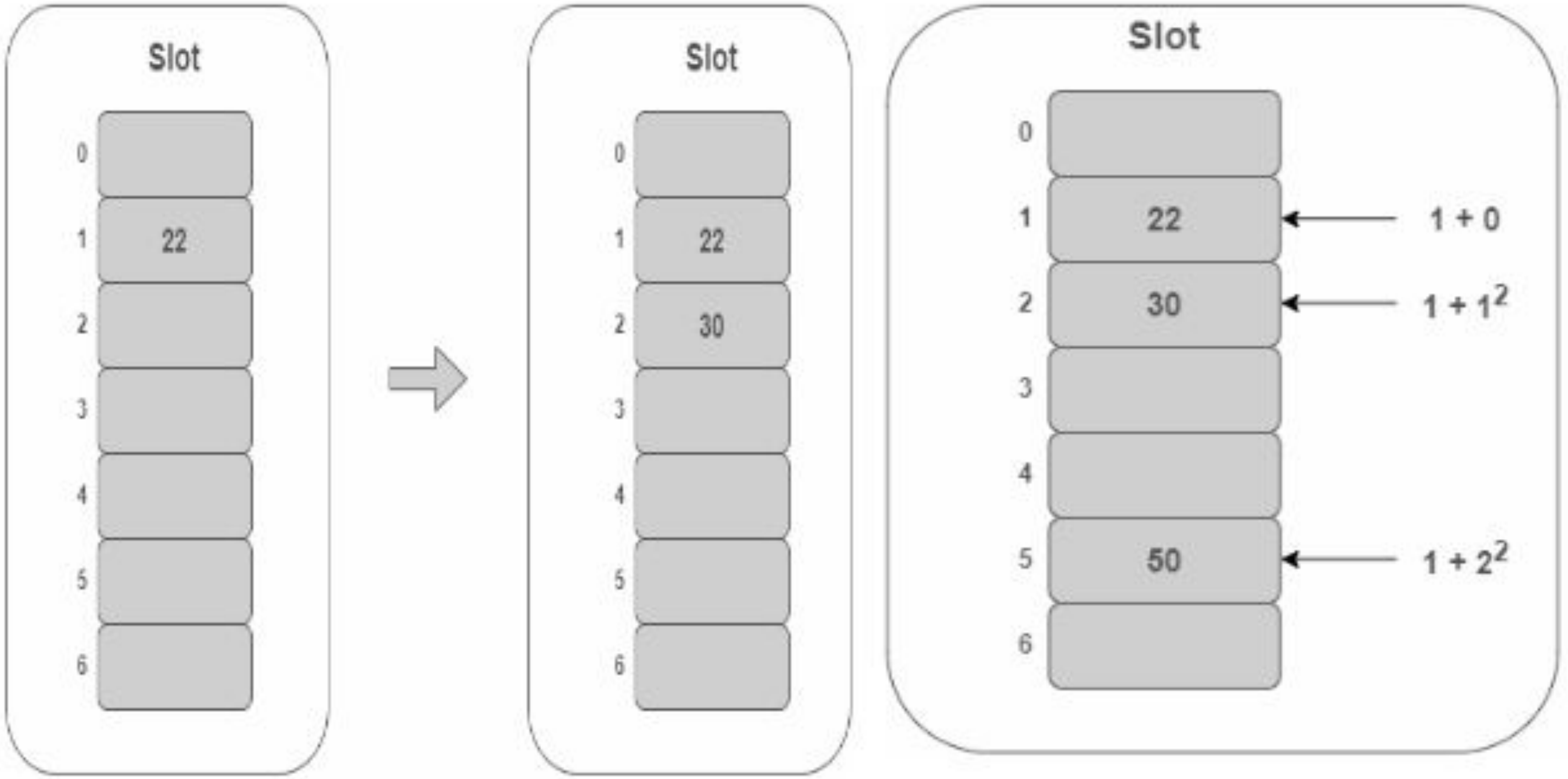
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

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, and 50.

- **Step 1:** Create a table of size 7.
- **Step 2** – Insert 22 and 30
    i. Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
    ii. Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.
- **Step 3:** Inserting 50
    a. Hash(50) = 50 % 7 = 1
    b. In our hash table slot 1 is already occupied. So, we will search for slot 1+12, i.e. 1+1 = 2,
    c. Again slot 2 is found occupied, so we will search for cell 1+22, i.e.1+4 = 5,
    d. Now, cell 5 is not occupied so we will place 50 in slot 5.

Example


Slot
0
1
22
2
3
4
5
6
Slot
0
1
22
2
30
3
4
5
6
Slot
0
1
22
2
30
3
4
5
50
6
1 + 0
1 + 1²
1 + 2²

Insert the following sequence of keys in the hash table of size 10

{72,27,36,24,63,81,101}

Double Hashing

- The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **ith** rotation.
  - let hash(x) be the slot index computed using hash function.
  - If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
  - If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
  - If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

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 1:** Insert 27
  - 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.
- **Step 2:** Insert 43
  - 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.
- **Step 3:** Insert 692
  - 692 % 7 = 6, but location 6 is already being occupied and this is a collision
  - So we need to resolve this collision using double hashing.

*h_{new} = [h1(692) + i * (h2(692)] % 7*

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

= 9% 7

= 2

Now, as 2 is an empty slot,

so we can insert 692 into 2nd slot.

- **Step 4:** Insert 72
    - 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
    - So we need to resolve this collision using double hashing.

*h_{new} = [h1(72) + i * (h2(72)] % 7*

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

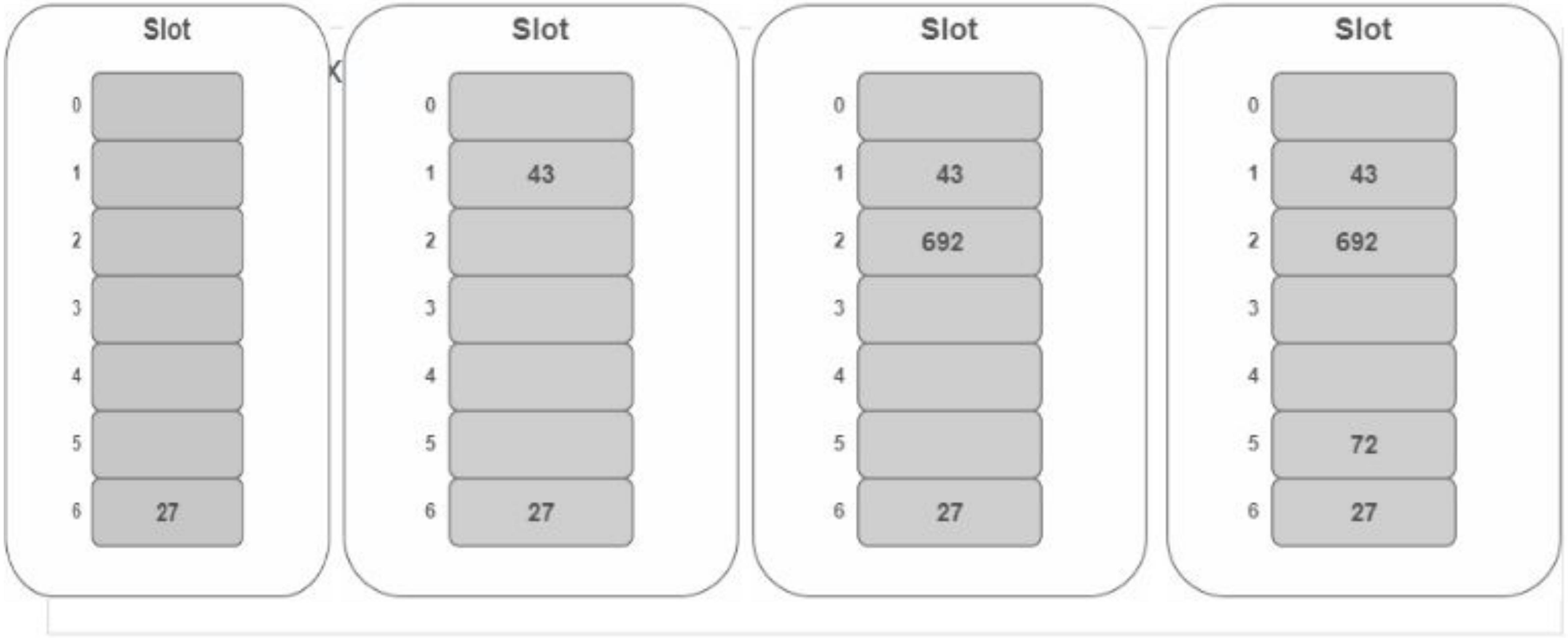
= 5 % 7

= 5,

Now, as 5 is an empty slot,

so we can insert 72 into 5th slot.

Example


Slot
0
1
2
3
4
5
6
27
Slot
0
1
43
2
3
4
5
6
27
Slot
0
1
43
2
692
3
4
5
6
27
Slot
0
1
43
2
692
3
4
5
72
6
27

Comparison

- Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
- Quadratic probing lies between the two in terms of cache performance and clustering.
- Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.