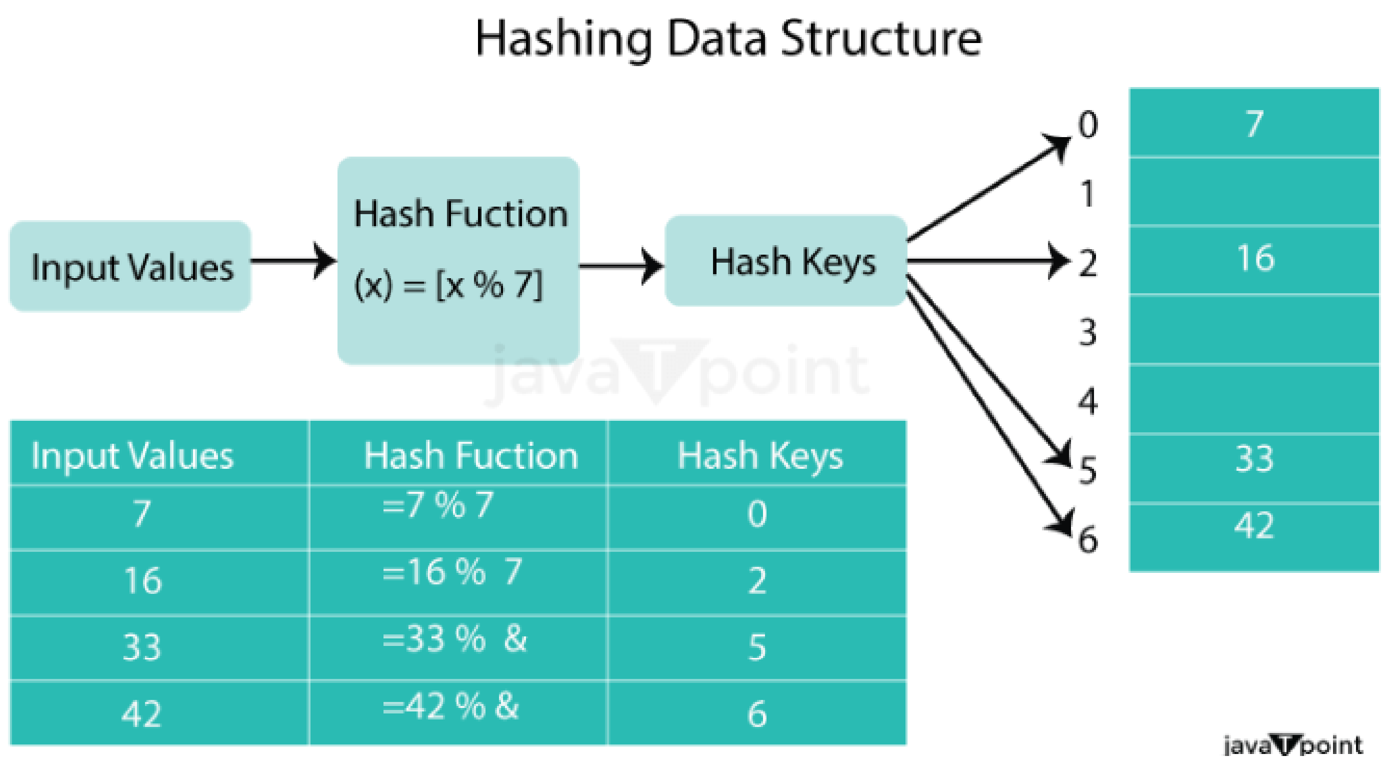
Hashing in Data Structure

Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access. It involves mapping data to a specific index in a hash table using a hash function that enables fast retrieval of information based on its key.

The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time.

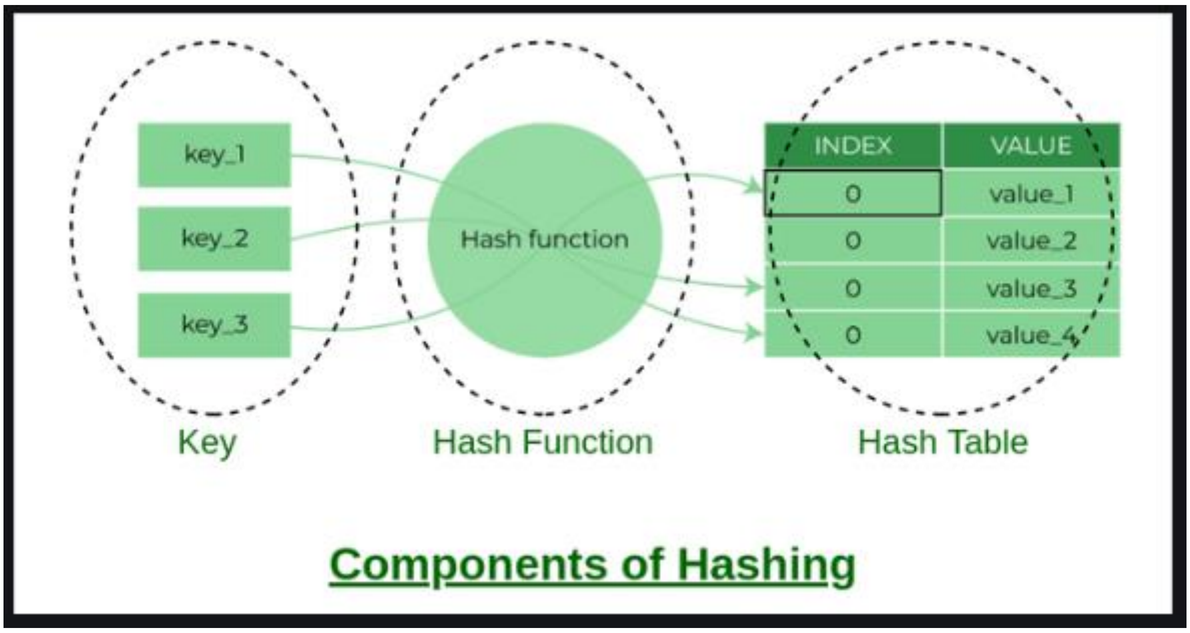


Input Values	Hash Fuction	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % &	5
42	=42 % &	6

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.



Hash Table in Data Structure

A hash table is also referred as a hash map (key value pairs) or a hash set (only keys). It uses a hash function to map keys to a fixed-size array, called a hash table.

What is a Hash function?

The hash function creates a mapping between key and value, this is done through the use of mathematical formulas known as hash functions. The result of the hash function is referred to as a hash value or hash.

Hash Function and types of Hash Function

A hash function is a function that takes an input (or 'message') and returns a fixed-size string of bytes. The output, typically a number, is called the hash code or hash value. The main purpose of a hash function is to efficiently map data of arbitrary size to fixed-size values, which are often used as indexes in hash tables.

Key Properties of Hash Functions

Deterministic: A hash function must consistently produce the same output for the same input.

Fixed Output Size: The output of a hash function should have a fixed size, regardless of the size of the input.

Efficiency: The hash function should be able to process input quickly.

Uniformity: The hash function should distribute the hash values uniformly across the output space to avoid clustering.

Collision Resistance: It should be difficult to find two different inputs that produce the same hash value.

Types of Hash Functions

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

- **Division Method.**
- **Mid-Square Method**
- **Folding Method**
- **Universal Hashing**

1. Division Method

- **Description**: This method uses the modulo operation to map a key to a hash value.
- **How It Works**: The hash value is computed as

  h(k)=k mod m

  h(k) = k % m

  where **k is the key** and **m is the size of the hash table**.

- **Example**: If k=12345 and m=100,

  The hash value h(12345) = 12345 mod 100 = **45h**

  Advantages: Simple and fast.

- **Disadvantages**: The choice of mmm is crucial; a poor choice can lead to many collisions.

2. Mid-Square Method

- **Description**: This method squares the key and then extracts the middle portion of the resulting value as the hash value.
- **How It Works**: The key k is squared, and the middle digits of the result are taken as the hash value.
- **Example**: If k=123,

then k^2 = **15129**.

If the table size is 100, taking the middle digits (12 in this case) could yield a **hash value of 12**.

- **Advantages**: Works well for smaller key values and tends to distribute keys more uniformly.
- **Disadvantages**: Not suitable for very large key values, as it may lead to overflow in certain cases.

3. Folding Method

- **Description**: This method breaks the key into several parts, adds them together, and then applies the modulo operation.
- **How It Works**: The key is divided into equal-sized pieces, possibly with the last piece being shorter. These pieces are added together, and then the modulo operation is applied with the hash table size.
- **Example**: If the key is 123456 and the table size is 100,

  the key might be split into **123** and **456**, and then the sum (**579**) mod 100 would give **79**.

- **Advantages**: Handles large keys well and tends to distribute values evenly.
- **Disadvantages**: The choice of splitting the key can affect performance.

Program:

```
#include <stdio.h>

#define TABLE_SIZE 10

// Function to create a simple hash from an array element

int hash_function(int key) {

    return key % TABLE_SIZE;

}


// Function to insert elements into the hash table (no collision handling)

void insert(int hash_table[], int arr[], int n) {

    for (int i = 0; i < n; i++) {

        int index = hash_function(arr[i]);
        // Directly insert the element at the hashed index (no collision handling)
        hash_table[index] = arr[i];
    }
}
// Function to search for an element in the hash table (no collision handling)
int search(int hash_table[], int key) {
    int index = hash_function(key);
    // Direct access since we assume no collisions
    if (hash_table[index] == key) {
        return index;  // Element found
    } else {
        return -1;  // Element not found
    }
}

// Function to delete an element from the hash table
void delete(int hash_table[], int key) {
    int index = search(hash_table, key);

    if (index != -1) {
        hash_table[index] = -1;  // Mark the slot as empty
        printf("Element %d deleted from index %d.\n", key, index);
    } else {
        printf("Element %d not found in the hash table.\n", key);
    }
}

// Function to display the hash table
void display_hash_table(int hash_table[]) {
    printf("Hash Table:\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hash_table[i] != -1) {
            printf("Index %d: %d\n", i, hash_table[i]);
        }
else {
            printf("Index %d: Empty\n", i);
        }
    }
}

int main() {
    int arr[] = {23, 43, 13, 27, 16};  // Example array
    int n = sizeof(arr) / sizeof(arr[0]);
    // Initialize the hash table with -1 (indicating empty slots)
    int hash_table[TABLE_SIZE];
    for (int i = 0; i < TABLE_SIZE; i++) {
        hash_table[i] = -1;
    }
    // Insert elements from the array into the hash table
    insert(hash_table, arr, n);
    // Display the hash table
    display_hash_table(hash_table);
    // Element to search and delete
    int key = 27;
    // Search for the element
    int result = search(hash_table, key);
    // Print the search result
    if (result != -1) {
        printf("Element %d found at index %d in the hash table.\n", key, result);
    } else {
        printf("Element %d not found in the hash table.\n", key);
    }
    // Delete the element
    delete(hash_table, key);
    // Display the hash table after deletion
    display_hash_table(hash_table);
}
```

Output:

```
Hash Table:
Index 0: Empty
Index 1: Empty
Index 2: Empty
Index 3: 13
Index 4: Empty
Index 5: Empty
Index 6: 16
Index 7: 27
Index 8: Empty
Index 9: 43
```

Search function =>Element 27 found at index 7 in the hash table.

Delete Function => Element 27 deleted from index 7.

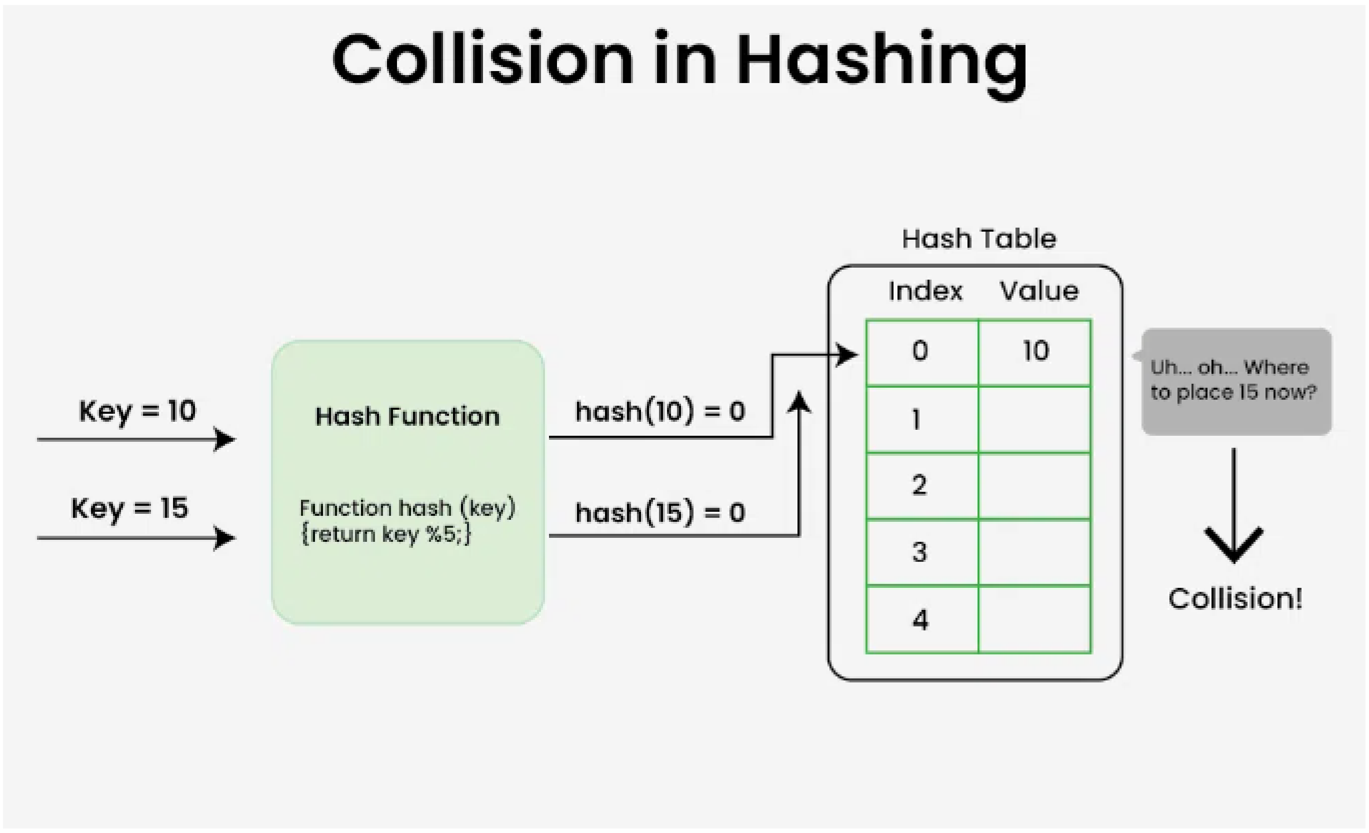
```
Hash Table:
Index 0: Empty
Index 1: Empty
Index 2: Empty
Index 3: 13
Index 4: Empty
Index 5: Empty
Index 6: 16
Index 7: Empty
Index 8: Empty
Index 9: 43
```

Collision in Hashing

What is Collision?

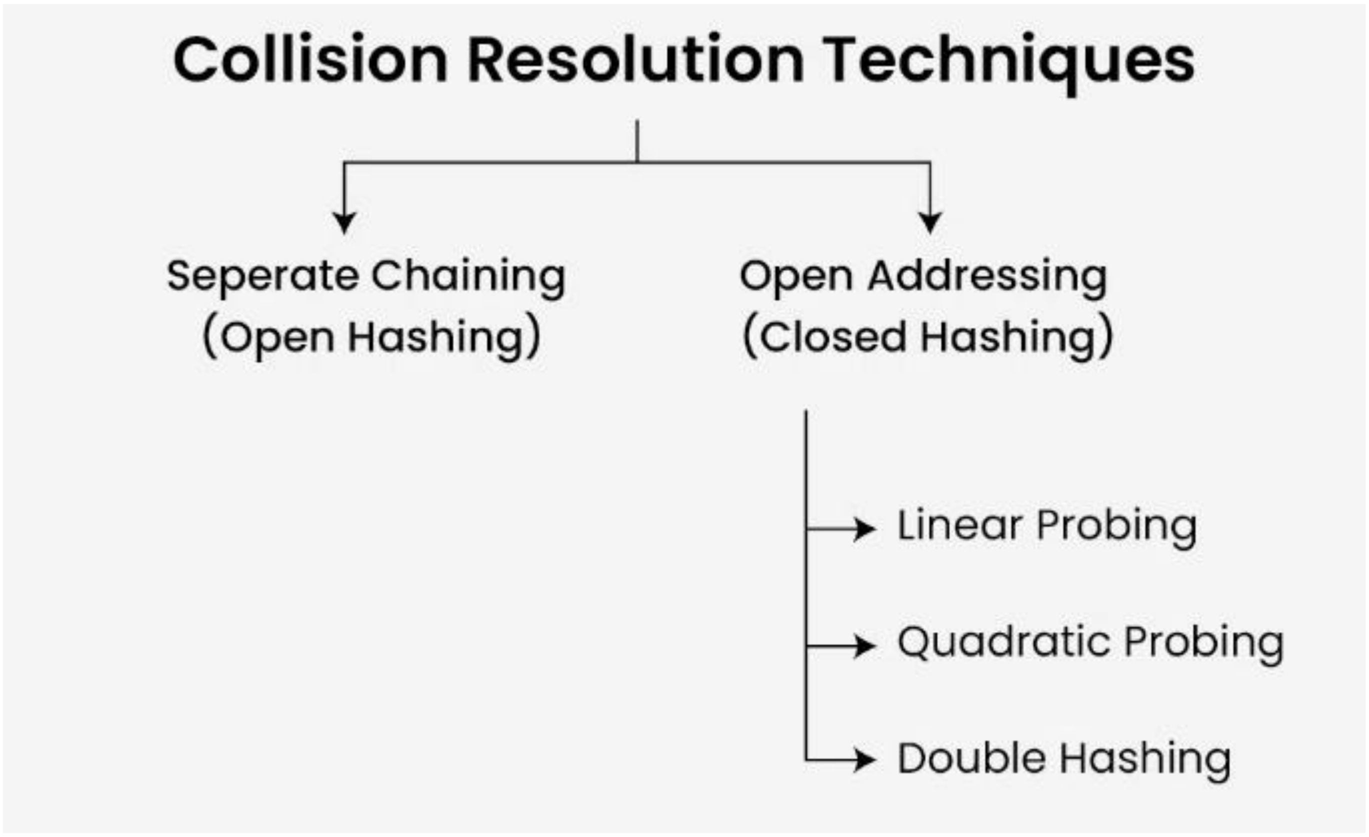
Collision in Hashing occurs when two different keys map to the same hash value. Hash collisions can be intentionally created for many hash algorithms. The probability of a hash collision depends on the size of the algorithm, the distribution of hash values and the efficiency of Hash function.

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:

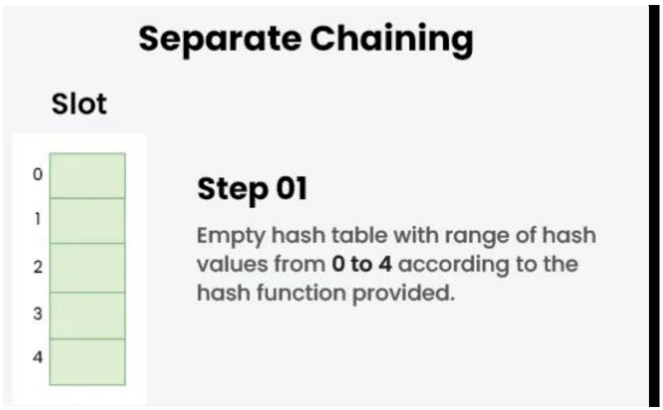

Collision Resolution Techniques
Seperate Chaining (Open Hashing)
Open Addressing (Closed Hashing)
Linear Probing
Quadratic Probing
Double Hashing

1) Separate Chaining

The Separate Chaining 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.

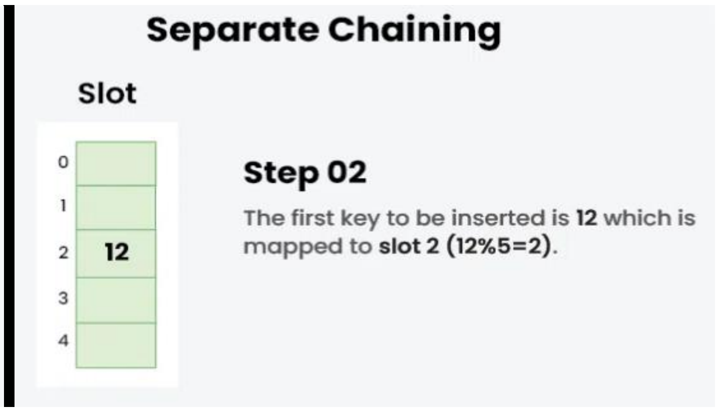
Example: We have given a hash function and we have to insert some elements in the hash table using a separate chaining method for collision resolution technique.

Hash function = key % 5,
Elements = 12, 15, 22, 25 and 37.



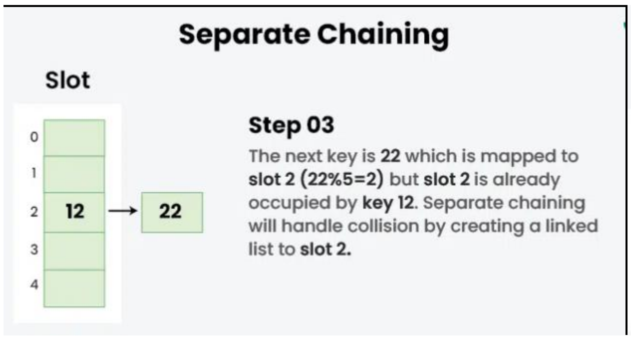
Step 01

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



Step 02

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

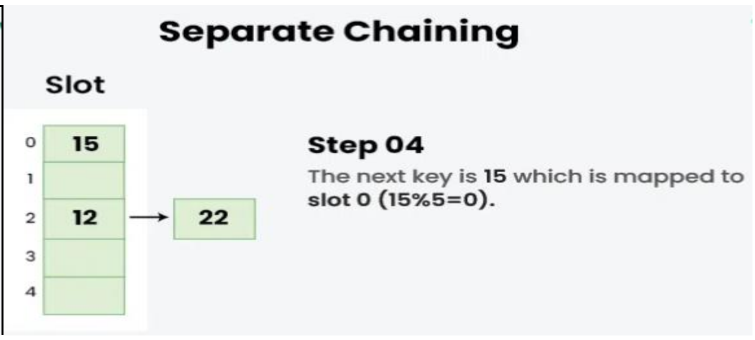


Separate Chaining

Slot

Step 03

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

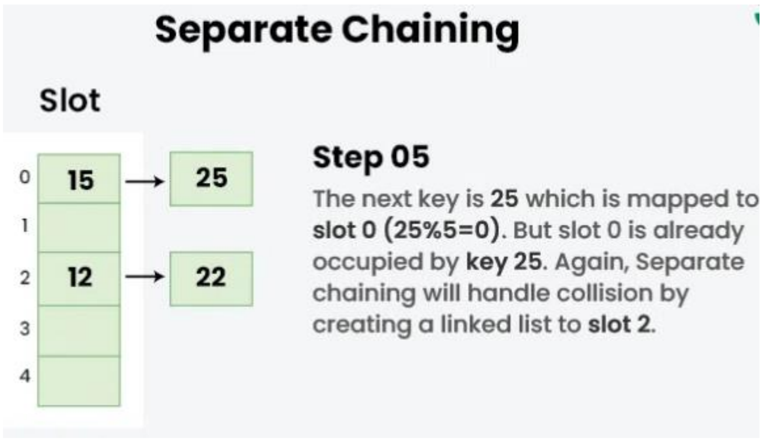


Separate Chaining

Slot

Step 04

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



Separate Chaining

Slot

Step 05

The next key is **25** which is mapped to **slot 0 (25%5=0)**. But slot 0 is already occupied by **key 25**. Again, Separate chaining will handle collision by creating a linked list to **slot 2**.

Program:
```c
#include <stdio.h>
#include <stdlib.h>

#define TABLE_SIZE 10  // Assume that the number of unique elements does not exceed TABLE_SIZE

// Node structure for the linked list (used in separate chaining)
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* create_node(int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

// Function to create a simple hash from an array element
int hash_function(int key) {
    return key % TABLE_SIZE;
}

// Function to insert elements into the hash table with separate chaining
void insert(struct Node* hash_table[], int arr[], int n) {
    for (int i = 0; i < n; i++) {
        int index = hash_function(arr[i]);

        // Create a new node for the element
        struct Node* new_node = create_node(arr[i]);

        // Insert the node at the beginning of the linked list at hash_table[index]
        new_node->next = hash_table[index];
        hash_table[index] = new_node;
    }
}
```

```
// Function to search for an element in the hash table with separate chaining
struct Node* search(struct Node* hash_table[], int key) {
    int index = hash_function(key);
    struct Node* temp = hash_table[index];
    while (temp != NULL) {
        if (temp->data == key) {
            return temp;
        }
        temp = temp->next;
    }
    return NULL;  // Key not found
}

// Function to delete an element from the hash table with separate chaining
void delete(struct Node* hash_table[], int key) {
    int index = hash_function(key);
    struct Node* temp = hash_table[index];
    struct Node* prev = NULL;

    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Element %d not found in the hash table.\n", key);
        return;
    }

    // If the element is found
    if (prev == NULL) {
        // Element is at the head of the linked list
        hash_table[index] = temp->next;
    } else {
        // Element is in the middle or end of the linked list
        prev->next = temp->next;
    }
    free(temp);
    printf("Element %d deleted from the hash table.\n", key);
}

// Function to print the hash table (for debugging/visualization purposes)
void print_hash_table(struct Node* hash_table[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("Index %d: ", i);
        struct Node* temp = hash_table[i];
        while (temp != NULL) {
            printf("%d -> ", temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

int main() {
    int arr[] = {23, 43, 13, 27, 16, 26, 12, 13, 11, 21, 22};  // Example array
    int n = sizeof(arr) / sizeof(arr[0]);

    // Initialize the hash table with NULL pointers (indicating empty lists)
    struct Node* hash_table[TABLE_SIZE] = {NULL};

    // Insert elements from the array into the hash table
    insert(hash_table, arr, n);

    // Print the hash table
    print_hash_table(hash_table);

    // Search for elements in the hash table
    int search_key = 16;
    struct Node* found_node = search(hash_table, search_key);
    if (found_node != NULL) {
        printf("Element %d found in the hash table.\n", search_key);
    } else {
        printf("Element %d not found in the hash table.\n", search_key);
    }
    // Delete elements from the hash table
    int delete_key = 27;
    delete(hash_table, delete_key);

    // Print the hash table after deletion
    print_hash_table(hash_table);

    // Clean-up: Free the allocated memory for the hash table
    for (int i = 0; i < TABLE_SIZE; i++) {
        struct Node* temp = hash_table[i];
        while (temp != NULL) {
            struct Node* to_free = temp;
            temp = temp->next;
            free(to_free);
        }
    }

    return 0;
}
```

Output

```
Index 0: NULL

Index 1: 11 -> NULL

Index 2: 22 -> 12 -> NULL

Index 3: 43 -> 23 -> NULL

Index 4: NULL

Index 5: NULL

Index 6: 26 -> 16 -> NULL

Index 7: 27 -> NULL

Index 8: NULL

Index 9: 13 -> NULL


Element 16 found in the hash table.
Element 27 deleted from the hash table.

Index 0: NULL
Index 1: 11 -> NULL
Index 2: 22 -> 12 -> NULL
Index 3: 43 -> 23 -> NULL
Index 4: NULL
Index 5: NULL
Index 6: 26 -> 16 -> NULL
Index 7: NULL
Index 8: NULL
Index 9: 13 -> NULL
```

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.

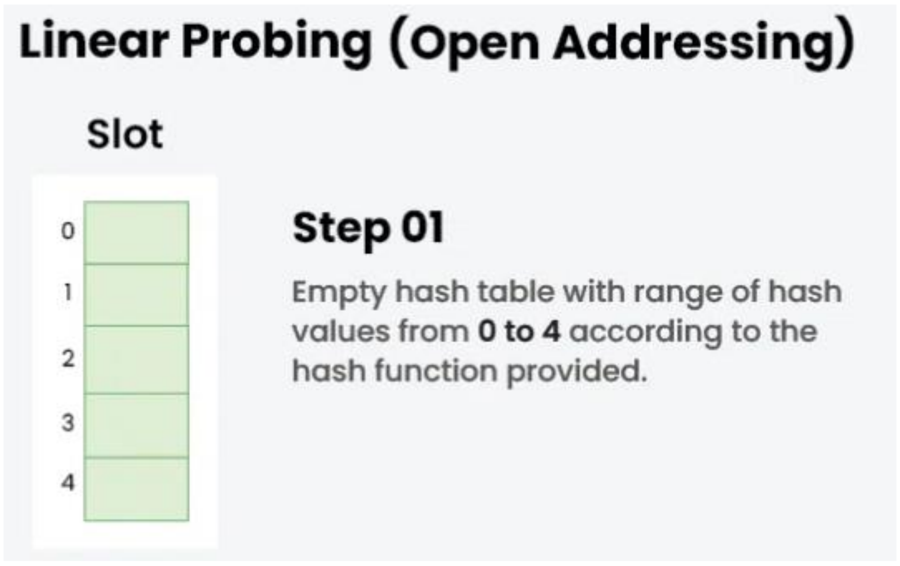
2. a) Linear Probing

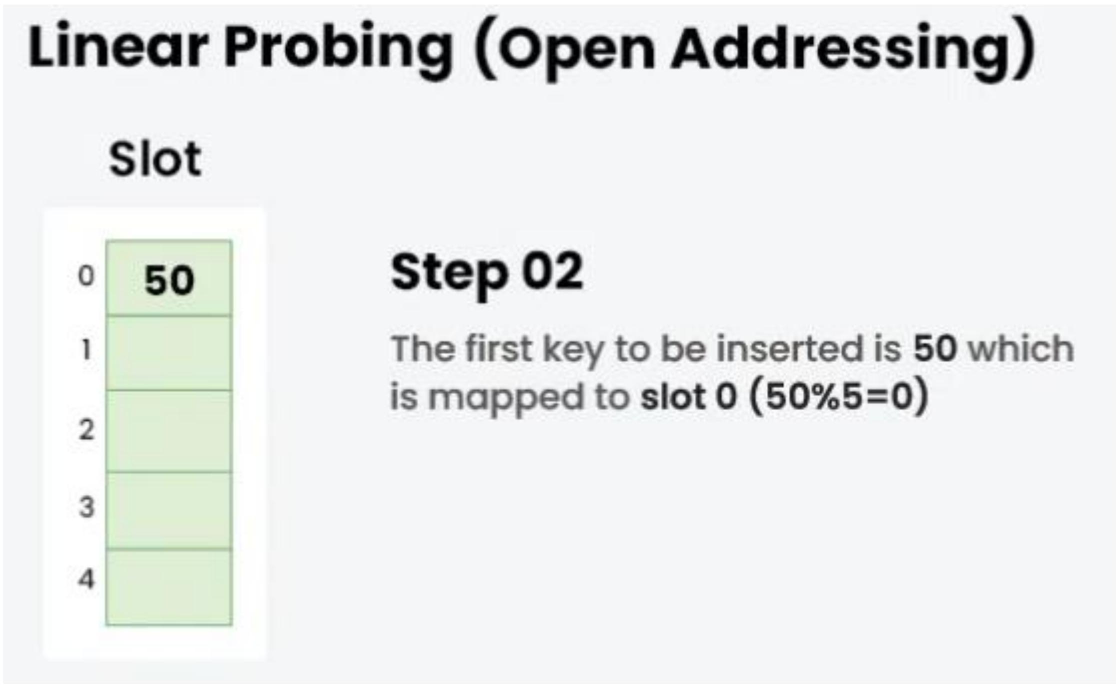
In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

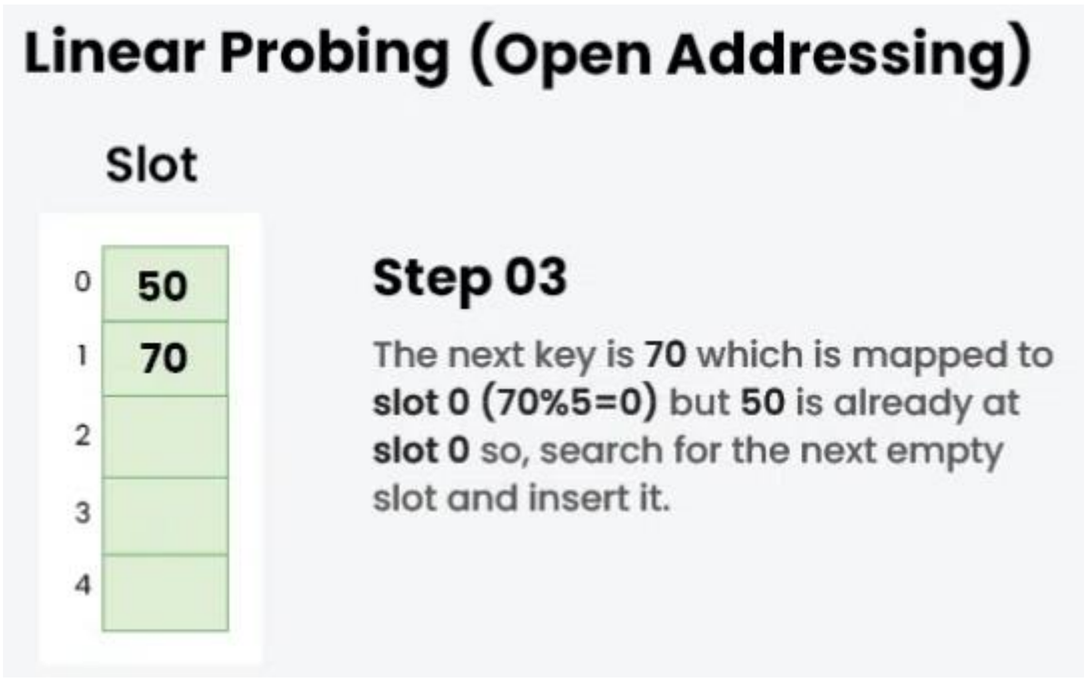
Algorithm:

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.


Linear Probing (Open Addressing)
Slot
0
1
2
3
4
Step 01
Empty hash table with range of hash values from **0 to 4** according to the hash function provided.


Linear Probing (Open Addressing)
Slot
0
50
1
2
3
4
Step 02
The first key to be inserted is **50** which is mapped to **slot 0 (50%5=0)**



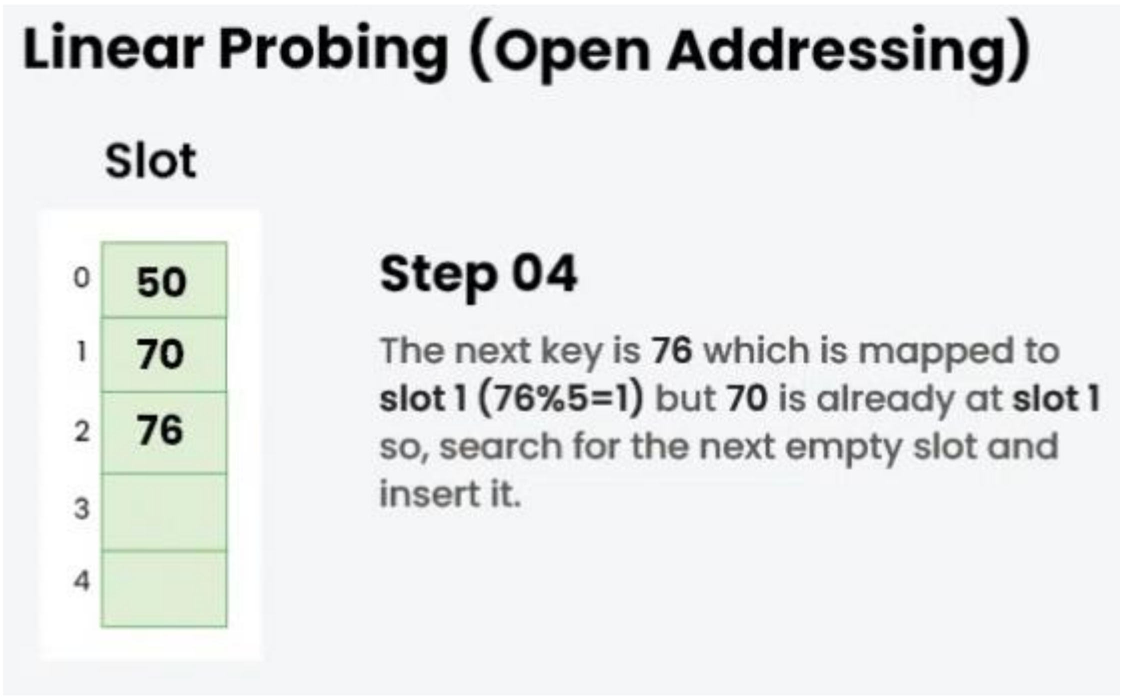
Linear Probing (Open Addressing)

Slot

0	50
1	70
2	
3	
4	

Step 03

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



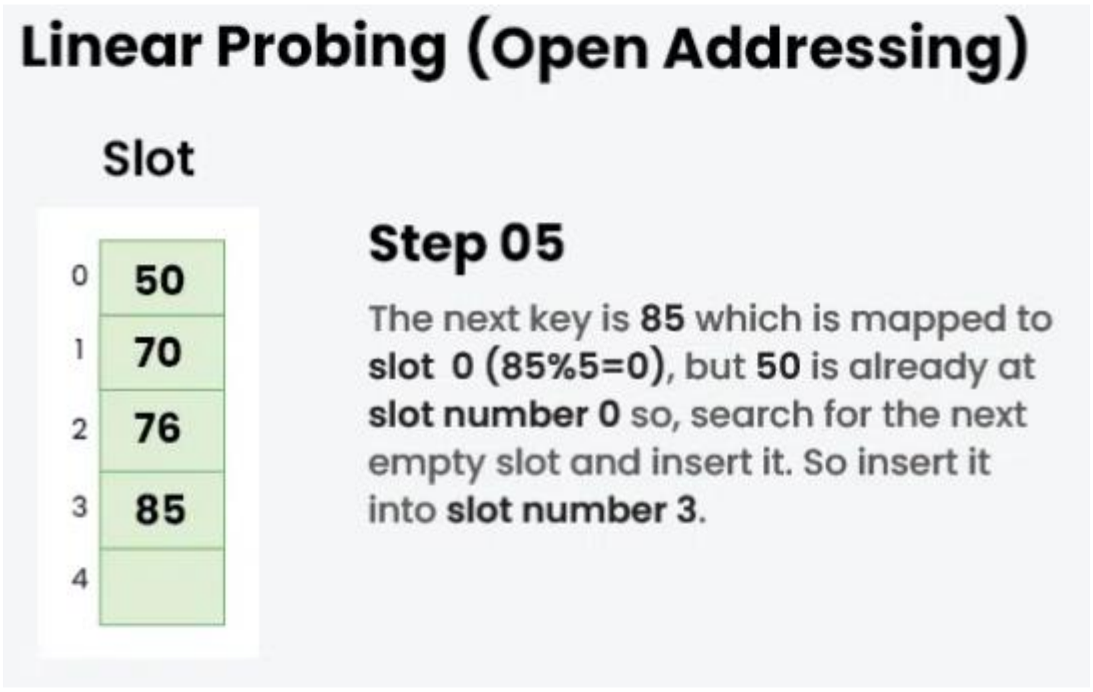
Linear Probing (Open Addressing)

Slot

0	50
1	70
2	76
3	
4	

Step 04

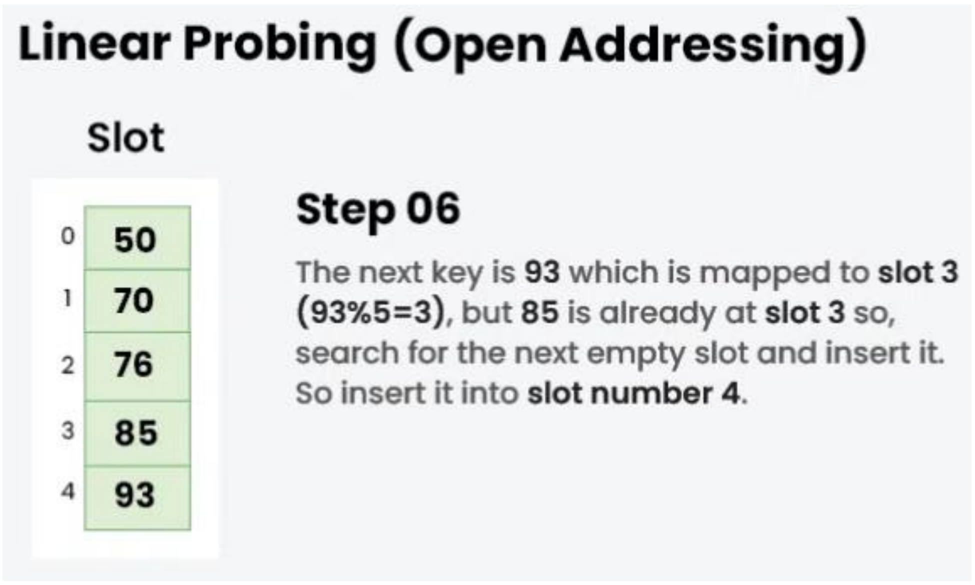
The next key is **76** which is mapped to **slot 1 (76%5=1)** but **70** is already at **slot 1** so, search for the next empty slot and insert it.



Linear Probing (Open Addressing)

Step 05

The next key is **85** which is mapped to **slot 0 (85%5=0)**, but **50** is already at **slot number 0** so, search for the next empty slot and insert it. So insert it into **slot number 3**.



Linear Probing (Open Addressing)

Step 06

The next key is **93** which is mapped to **slot 3 (93%5=3)**, but **85** is already at **slot 3** so, search for the next empty slot and insert it. So insert it into **slot number 4**.

Program:

```
#include <stdio.h>
#define table_size 10
int hash_function(int key)
{
    return key % table_size;
}
void insert(int hash_table[], int arr[], int length)
{
    for (int i = 0; i < length; i++)
    {
        int index = hash_function(arr[i]);
        // Handle collision using linear probing
        while (hash_table[index] != -1)
        {
            index = (index + 1) % table_size;
        }
        hash_table[index] = arr[i];
    }
}

void display(int hash_table[])
{
    printf("Hash Table:\n");
    for (int i = 0; i < table_size; i++)
    {
        if (hash_table[i] != -1)
        {
            printf("Index %d: %d\n", i, hash_table[i]);
        }
        else
        {
            printf("Index %d: Empty\n", i);
        }
    }
}

int search(int hash_table[], int key)
{
    int index = hash_function(key);
    // Search using linear probing
    while (hash_table[index] != -1)
    {
        if (hash_table[index] == key)
        {
            return index; // Return the index where the key is found
        }
        index = (index + 1) % table_size;

        // If we've looped back to the original index, the element isn't present
        if (index == hash_function(key))
        {
            break;
        }
    }
    return -1; // Key not found
}
```

```
void delete(int hash_table[], int key)
{
    int index = search(hash_table, key);
    if (index == -1)
    {
        printf("Element %d not found in the hash table.\n", key);
        return;
    }

    // Set the index as empty (using a sentinel value, e.g., -1)
    hash_table[index] = -1;
    printf("Element %d deleted from the hash table.\n", key);
    // Rehash all elements in the same cluster to avoid breaking the chain
    index = (index + 1) % table_size;
    while (hash_table[index] != -1)
    {
        int rehash_value = hash_table[index];
        hash_table[index] = -1;
        insert(hash_table, &rehash_value, 1);
        index = (index + 1) % table_size;
    }
}

int main()
{
    int arr[] = {23, 43, 13, 27, 16, 10, 11, 10};
    int length = sizeof(arr) / sizeof(arr[0]);
    int hash_table[table_size];
    // Initialize the hash table with -1 (indicating empty slots)
    for (int i = 0; i < table_size; i++)
    {
        hash_table[i] = -1;
    }
    insert(hash_table, arr, length);
    display(hash_table);
    // Search for an element
    int search_key = 27;
    int search_result = search(hash_table, search_key);
    if (search_result != -1)
    {
        printf("Element %d found at index %d.\n", search_key, search_result);
    }
    else
    {
        printf("Element %d not found in the hash table.\n", search_key);
    }

    // Delete an element
    int delete_key = 16;
    delete(hash_table, delete_key);

    // Display the hash table after deletion
    display(hash_table);

    return 0;
}
```

Output:

```
Hash Table:
Index 0: Empty
Index 1: 11
Index 2: Empty
Index 3: 23
Index 4: 43
Index 5: 13
Index 6: 16
Index 7: 27
Index 8: 10
Index 9: 10

Element 27 found at index 7.
Element 16 deleted from the hash table.


Hash Table:
Index 0: Empty
Index 1: 11
Index 2: Empty
Index 3: 23
Index 4: 43
Index 5: 13
Index 6: Empty
Index 7: 27
Index 8: 10
Index 9: 10
```

b) 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.

An example sequence using quadratic probing is:

H + 1 2 , H + 2 2 , H + 3 2 , H + 4 2 …………………. H + k 2

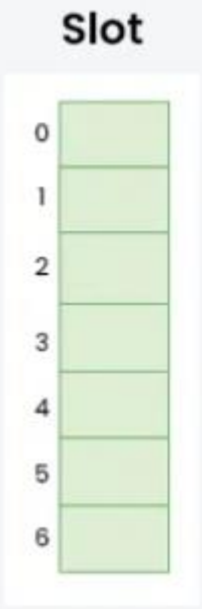
This method is also known as the mid-square method because in this method we look for i 2 ‘th probe (slot) in i’th iteration and the value of i = 0, 1, . . . n – 1. We always start from the original hash location. If only the location is occupied then we check the other slots.

Let hash(x) be the slot index computed using 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

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

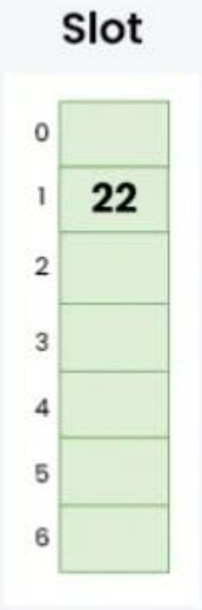
Quadratic Probing (Open Addressing)



Step 01

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

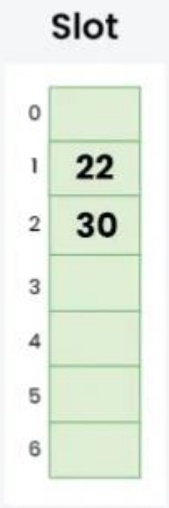
Quadratic Probing (Open Addressing)



Step 02

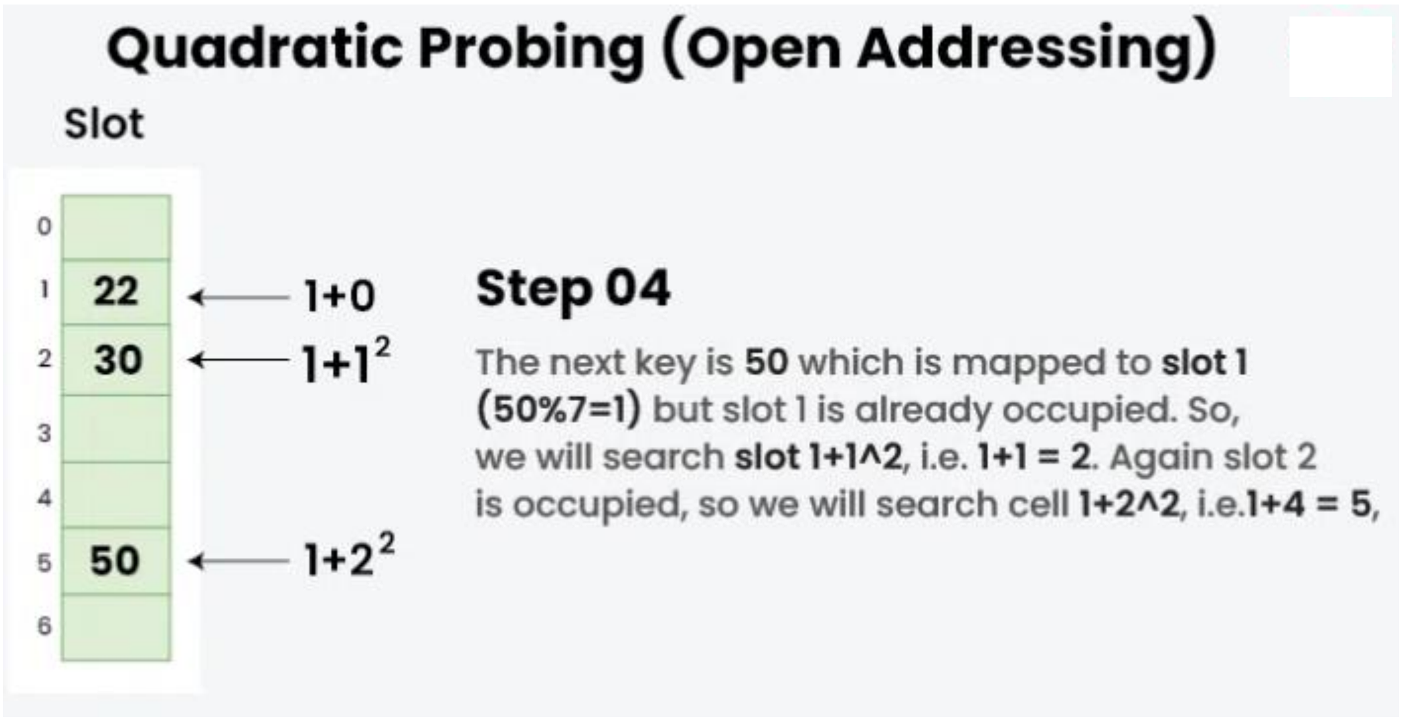
The first key to be inserted is **22** which is mapped to **slot 1 (22%7=1)**

Quadratic Probing (Open Addressing)



Step 03

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


Quadratic Probing (Open Addressing)
Slot
0
1
22
1+0
2
30
1+1^2
3
4
5
50
1+2^2
6
Step 04
The next key is 50 which is mapped to slot 1 (50%7=1) but slot 1 is already occupied. So, we will search slot 1+1^2, i.e. 1+1 = 2. Again slot 2 is occupied, so we will search cell 1+2^2, i.e.1+4 = 5,

c) Double Hashing
Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing make use of two hash function,

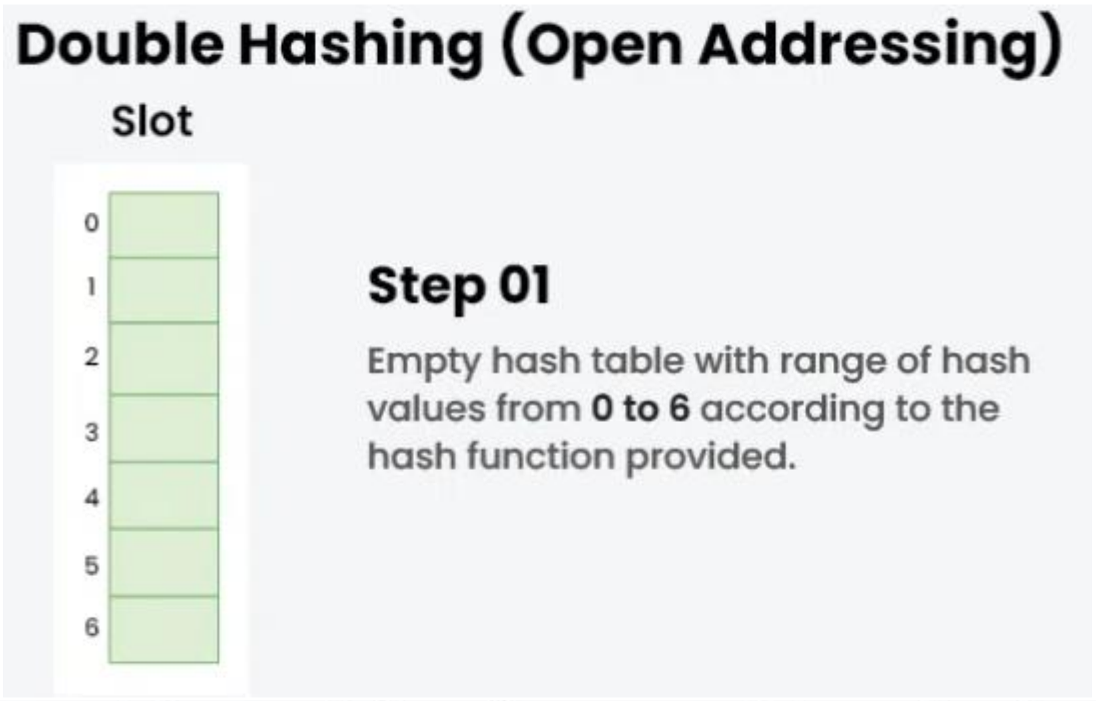
The first hash function is h1(k) which takes the key and gives out a location on the hash table. But if the new location is not occupied or empty then we can easily place our key.
But in case the location is occupied (collision) we will use secondary hash-function h2(k) in combination with the first 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.
Complexity of the Double hashing algorithm:

Time complexity: O(n)

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)

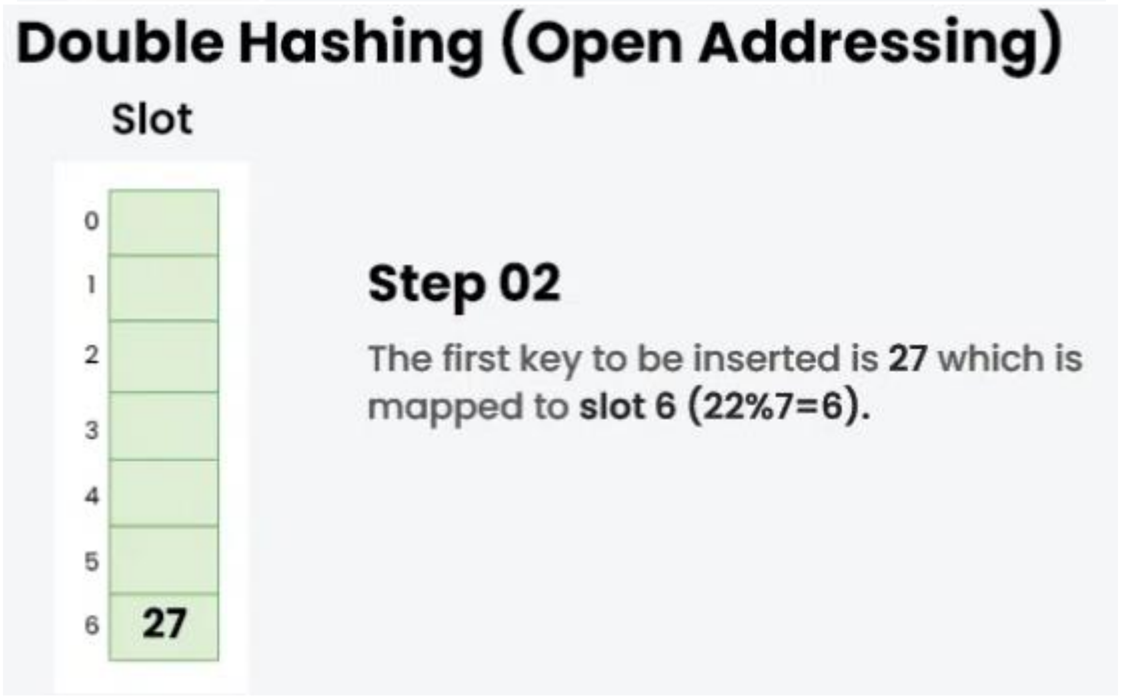


Double Hashing (Open Addressing)

Slot	
0	
1	
2	
3	
4	
5	
6	

Step 01

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

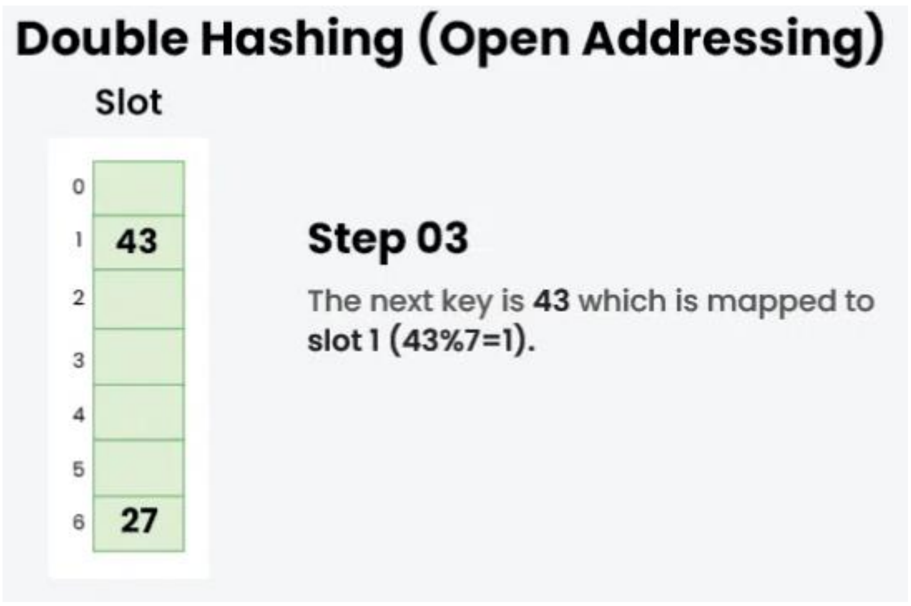


Double Hashing (Open Addressing)

Slot	
0	
1	
2	
3	
4	
5	
6	27

Step 02

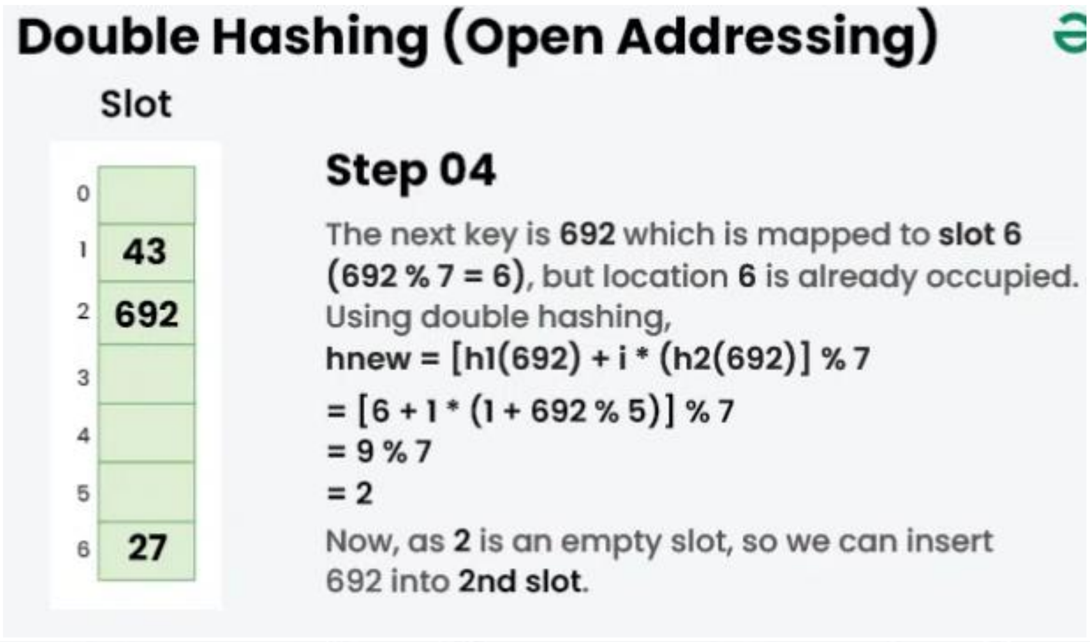
The first key to be inserted is **27** which is mapped to **slot 6 (22%7=6).**



Double Hashing (Open Addressing)

Step 03

The next key is **43** which is mapped to **slot 1 (43%7=1).**



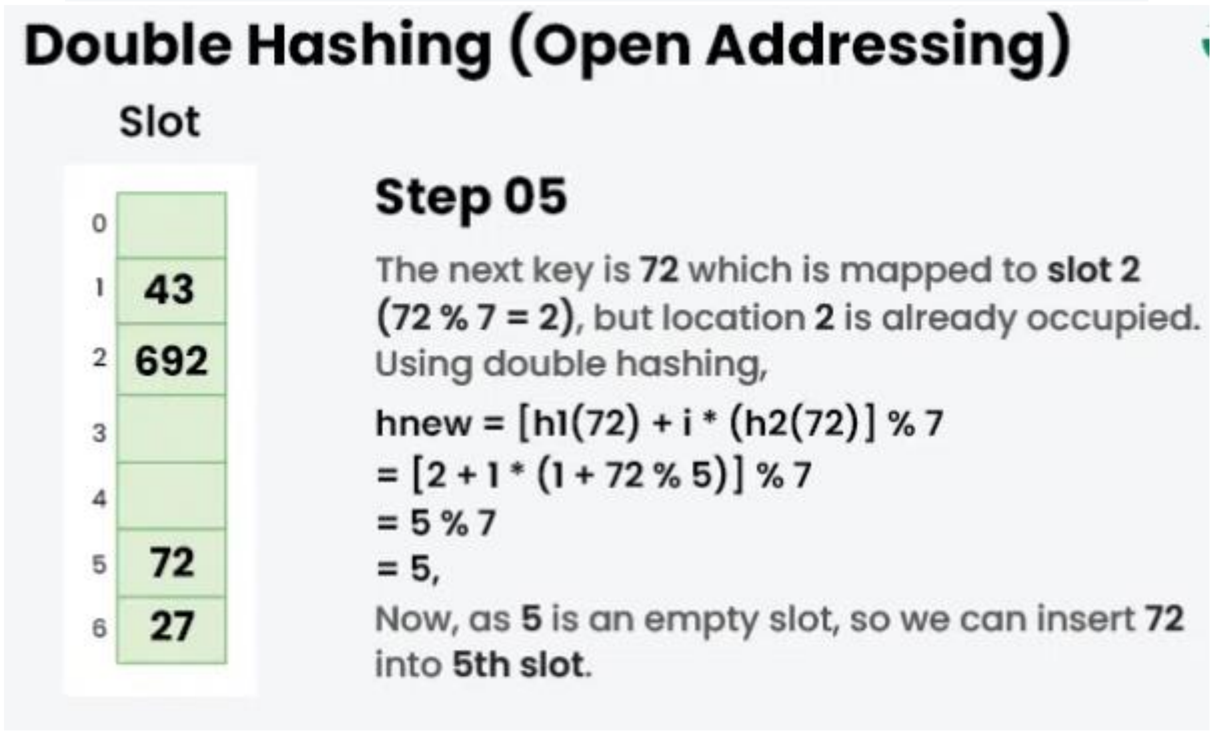
Double Hashing (Open Addressing)

Step 04

The next key is **692** which is mapped to **slot 6** **(692 % 7 = 6)**, but location **6** is already occupied. Using double hashing,

hnew = [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**.



Double Hashing (Open Addressing)

Step 05

The next key is **72** which is mapped to **slot 2** **(72 % 7 = 2)**, but location **2** is already occupied. Using double hashing,

hnew = [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**.

Static and Dynamic Hashing

Static hashing is a technique in which the size of the hash table remains fixed, and the hash function is static, meaning it does not change throughout the operations on the hash table. Static hashing is typically used when the total number of elements to be inserted is known in advance, and the hash table size can be pre-determined to minimize collisions.

Key Features of Static Hashing:

1. **Fixed Table Size**: The size of the hash table is set during the initialization and does not change.
2. **Hash Function**: The hash function is consistent and deterministic, meaning it always returns the same index for a given key.
3. **Collision Handling**: Since the table size is fixed, collisions (when two keys hash to the same index) are managed using techniques like:
    - **Separate Chaining**: Each index in the hash table points to a linked list (or another data structure) of all the elements that hash to that index.
    - **Open Addressing**: When a collision occurs, the algorithm searches for the next available slot using techniques like linear probing, quadratic probing, or double hashing.

Example of Static Hashing with Open Addressing (Linear Probing):

```
#include <stdio.h>
#define TABLE_SIZE 10
int hash_function(int key) {
    return key % TABLE_SIZE;
}

void insert(int hash_table[], int key) {
    int index = hash_function(key);

    while (hash_table[index] != -1) {
        index = (index + 1) % TABLE_SIZE;  // Linear probing
    }

    hash_table[index] = key;
}

int search(int hash_table[], int key) {
    int index = hash_function(key);
    while (hash_table[index] != -1) {
        if (hash_table[index] == key) {
            return index;
        }
        index = (index + 1) % TABLE_SIZE;
    }
    return -1;
}

void delete(int hash_table[], int key) {
    int index = search(hash_table, key);

    if (index == -1) {
        printf("Element %d not found in the hash table.\n", key);
        return;
    }

    hash_table[index] = -1;
    printf("Element %d deleted from the hash table.\n", key);

    // Rehash the subsequent elements in the cluster to avoid breaking the chain
    index = (index + 1) % TABLE_SIZE;
    while (hash_table[index] != -1) {
        int rehash_value = hash_table[index];
        hash_table[index] = -1;
        insert(hash_table, rehash_value);
        index = (index + 1) % TABLE_SIZE;
    }
}

void display(int hash_table[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hash_table[i] != -1) {
            printf("Index %d: %d\n", i, hash_table[i]);
        } else {
            printf("Index %d: Empty\n", i);
        }
    }
}

int main() {
    int hash_table[TABLE_SIZE];
    for (int i = 0; i < TABLE_SIZE; i++) {
        hash_table[i] = -1;
    }

    int arr[] = {23, 43, 13, 27, 16, 10, 11, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < n; i++) {
        insert(hash_table, arr[i]);
    }

    display(hash_table);

    // Search for a key
    int search_key = 27;
    int index = search(hash_table, search_key);
    if (index != -1) {
        printf("Element %d found at index %d\n", search_key, index);
    } else {
        printf("Element %d not found\n", search_key);
    }

    // Delete a key
    int delete_key = 16;
    delete(hash_table, delete_key);

    display(hash_table);

    return 0;
}
```

Dynamic hashing is a technique used to manage hash tables that can grow or shrink in size dynamically as the number of elements changes. Unlike static hashing, where the hash table size is fixed, dynamic hashing allows the table to expand or contract, reducing the likelihood of collisions and maintaining efficient search, insertion, and deletion operations.

Key Features of Dynamic Hashing:

1. **Table Size Adjustments**: The hash table can increase or decrease in size based on the number of elements.

2. **Rehashing**: When the table grows or shrinks, elements are rehashed to new positions based on the new table size.
3. **Load Factor**: The load factor (ratio of elements to table size) is used to determine when to resize the table. Commonly, when the load factor exceeds a threshold (e.g., 0.75), the table is resized.
4. **Collision Handling**: Similar to static hashing, collisions can be handled through techniques like separate chaining or open addressing.

Example of Dynamic Hashing with Open Addressing (Linear Probing):

```
#include <stdio.h>
#include <stdlib.h>

#define INITIAL_TABLE_SIZE 5
#define LOAD_FACTOR_THRESHOLD 0.75

typedef struct {
    int *table;
    int size;
    int count;
} HashTable;

int hash_function(int key, int size) {
    return key % size;
}

HashTable* create_table(int size) {
    HashTable *hash_table = (HashTable *)malloc(sizeof(HashTable));
    hash_table->table = (int *)malloc(size * sizeof(int));
    hash_table->size = size;
    hash_table->count = 0;
    for (int i = 0; i < size; i++) {
        hash_table->table[i] = -1;
    }
    return hash_table;
}

void resize(HashTable *hash_table) {
    int new_size = hash_table->size * 2;
    int *new_table = (int *)malloc(new_size * sizeof(int));
    for (int i = 0; i < new_size; i++) {
        new_table[i] = -1;
    }
    for (int i = 0; i < hash_table->size; i++) {
        if (hash_table->table[i] != -1) {
            int new_index = hash_function(hash_table->table[i], new_size);
            while (new_table[new_index] != -1) {
                new_index = (new_index + 1) % new_size;
            }
            new_table[new_index] = hash_table->table[i];
        }
    }

    free(hash_table->table);
    hash_table->table = new_table;
    hash_table->size = new_size;
}

void insert(HashTable *hash_table, int key) {
    if ((float)hash_table->count / hash_table->size >= LOAD_FACTOR_THRESHOLD) {
        resize(hash_table);
    }

    int index = hash_function(key, hash_table->size);

    while (hash_table->table[index] != -1) {
        index = (index + 1) % hash_table->size;
    }

    hash_table->table[index] = key;
    hash_table->count++;
}

int search(HashTable *hash_table, int key) {
    int index = hash_function(key, hash_table->size);

    while (hash_table->table[index] != -1) {
        if (hash_table->table[index] == key) {
            return index;
        }
        index = (index + 1) % hash_table->size;
    }

    return -1;
}
```

```
void delete(HashTable *hash_table, int key) {
    int index = search(hash_table, key);

    if (index == -1) {
        printf("Element %d not found in the hash table.\n", key);
        return;
    }

    hash_table->table[index] = -1;
    hash_table->count--;

    index = (index + 1) % hash_table->size;
    while (hash_table->table[index] != -1) {
        int rehash_value = hash_table->table[index];
        hash_table->table[index] = -1;
        hash_table->count--;
        insert(hash_table, rehash_value);
        index = (index + 1) % hash_table->size;
    }
}

void display(HashTable *hash_table) {
    for (int i = 0; i < hash_table->size; i++) {
        if (hash_table->table[i] != -1) {
            printf("Index %d: %d\n", i, hash_table->table[i]);
        } else {
            printf("Index %d: Empty\n", i);
        }
    }
}

int main() {
    HashTable *hash_table = create_table(INITIAL_TABLE_SIZE);

    int arr[] = {23, 43, 13, 27, 16, 10, 11, 22, 33, 44};
    int n = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < n; i++) {
        insert(hash_table, arr[i]);
    }

    display(hash_table);
    int search_key = 27;
    int index = search(hash_table, search_key);
    if (index != -1) {
        printf("Element %d found at index %d\n", search_key, index);
    } else {
        printf("Element %d not found\n", search_key);
    }

    int delete_key = 16;
    delete(hash_table, delete_key);

    display(hash_table);

    free(hash_table->table);
    free(hash_table);

    return 0;
}
```

Applications of Hash Data structure

- Hash is used in databases for indexing.
- Hash is used in disk-based data structures.
- In some programming languages like Python, JavaScript hash is used to implement objects.

Real-Time Applications of Hash Data structure

- Hash is used for cache mapping for fast access to the data.
- Hash can be used for password verification.
- Hash is used in cryptography as a message digest.
- Rabin-Karp algorithm for pattern matching in a string.
- Calculating the number of different substrings of a string.

Advantages of Hash Data structure

- Hash provides better synchronization than other data structures.
- Hash tables are more efficient than search trees or other data structures
- Hash provides constant time for searching, insertion, and deletion operations on average.

Disadvantages of Hash Data structure

- Hash is inefficient when there are many collisions.
- Hash collisions are practically not avoided for a large set of possible keys.
- Hash does not allow null values.

Here are a few examples of questions related to hashing in data structures:

1. Define a hash function.
2. What is a collision in hashing?
3. Give one example of a collision resolution technique.
4. **What is the load factor in a hash table?**

- **Answer**: The load factor in a hash table is the ratio of the number of elements (n) to the size of the hash table (m). It is calculated as Load Factor = n / m and is used to determine when the hash table should be resized.

5. Why is it important to choose a good hash function?
6. What is open addressing in hashing?
7. How does double hashing differ from linear probing?
8. Why is the size of the hash table often chosen as a prime number?
9. **What does it mean for a hash function to be deterministic?**

- **Answer**: A deterministic hash function is one that always produces the same output (index) for a given input (key). This consistency is crucial for ensuring that the same key always maps to the same location in the hash table.

10. Briefly describe the difference between separate chaining and open addressing.
11. What is the primary disadvantage of using linear probing?