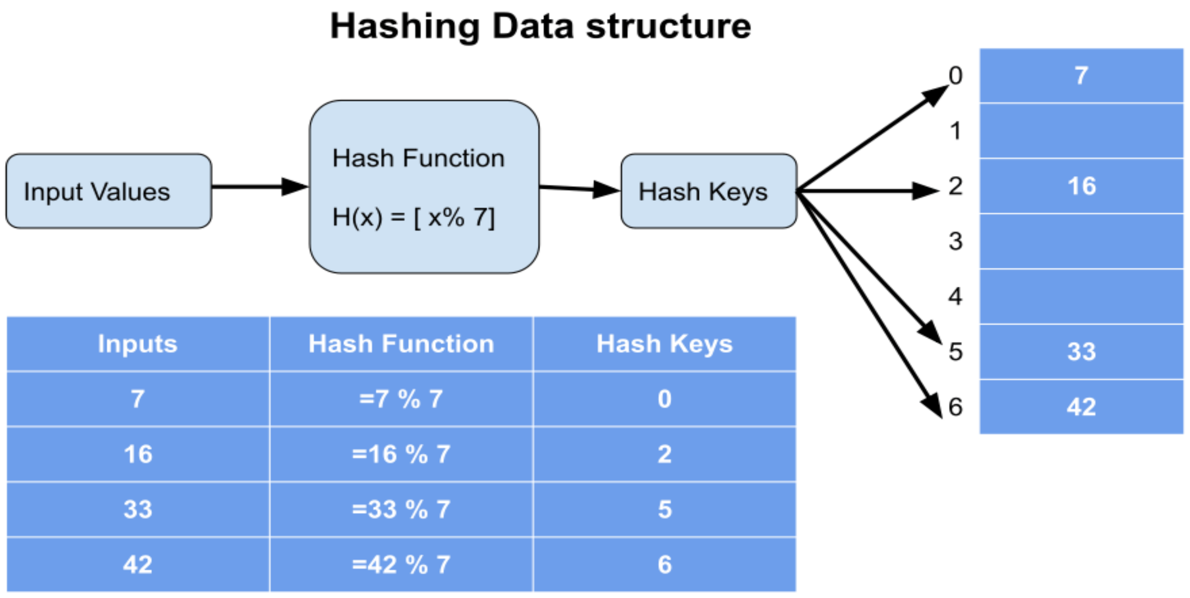
Data Structures Lab
Session 11

Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Sameer Faisal **T.A:**

- Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Get your lab checked at the end of the session.

HASHING

Hashing refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.



Inputs	Hash Function	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % 7	5
42	=42 % 7	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.

Inserting In A Hash Table:

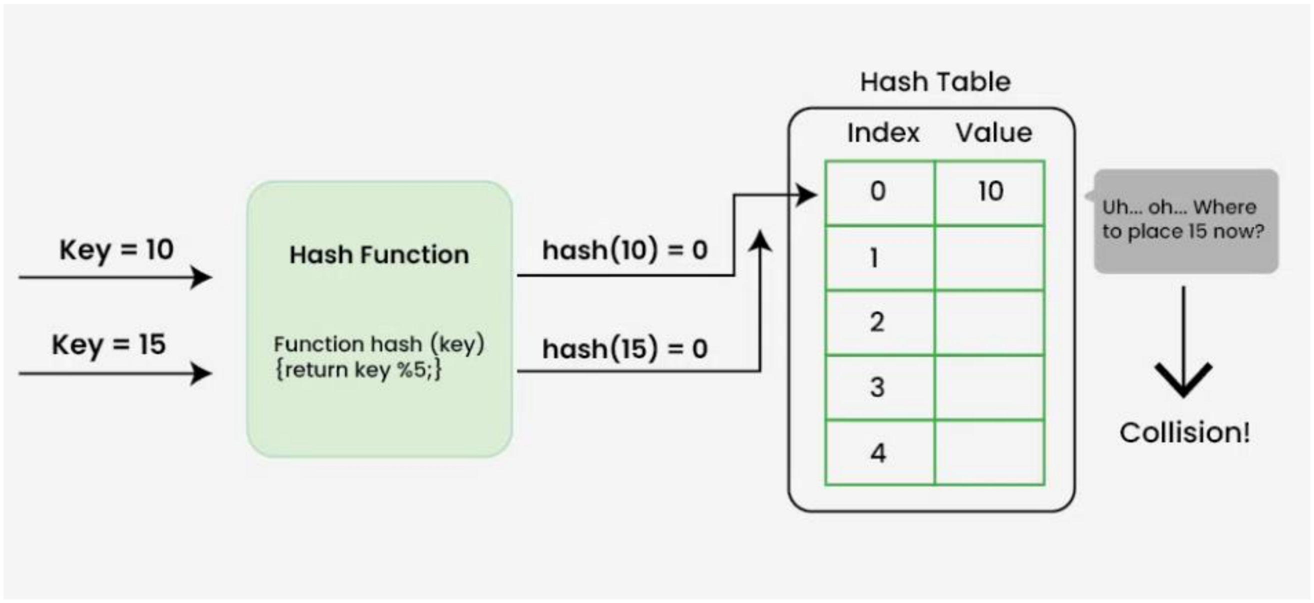
1. Choose a Hash Function: The first step is selecting or designing a hash function suitable for the data and the hash table size. The function should map input keys to indices within the range of the hash table size, ensuring uniform distribution.

   H(key) = key % sizeOfHashTable

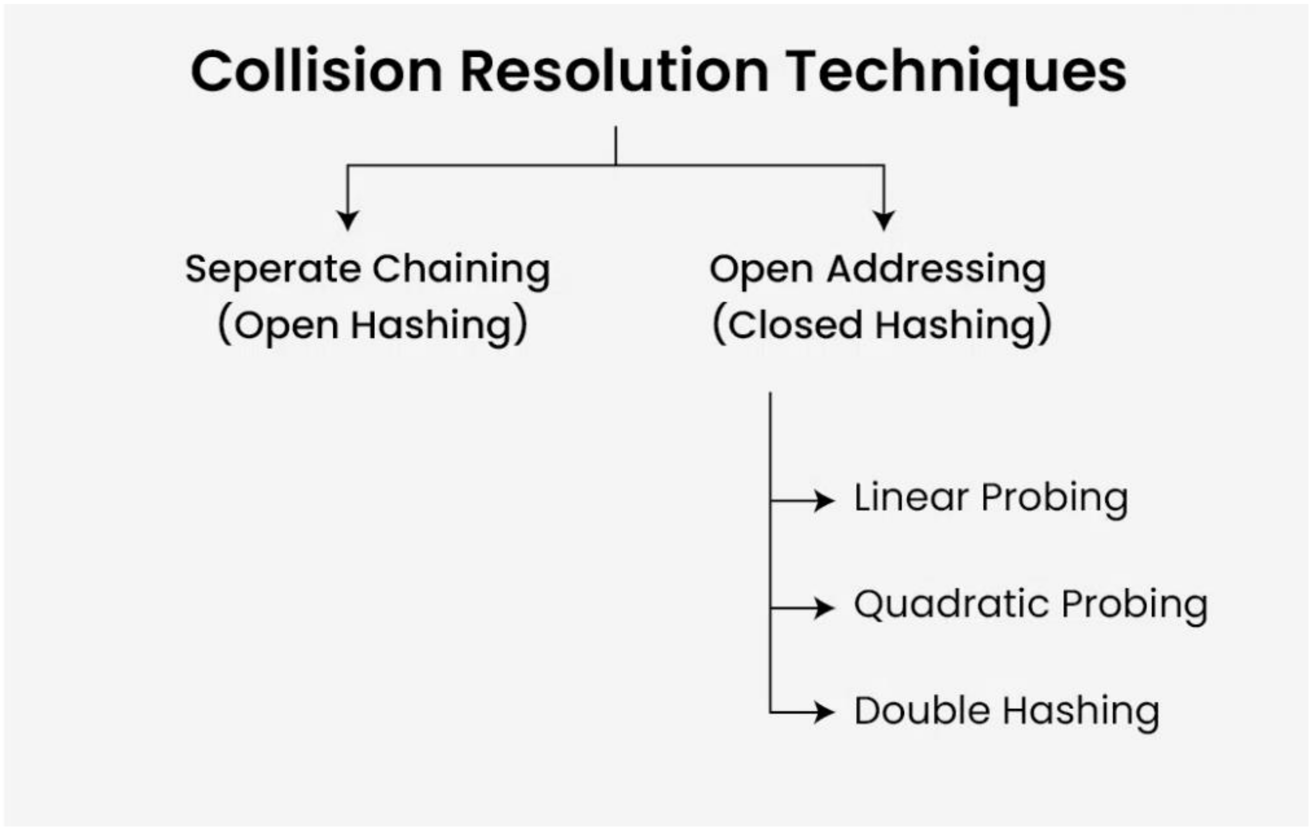
   The hash index must be within the range of hashtable size, so the key is usually taken modulo the table size to produce a valid index.

2. Calculate Hash Code: For a given key, apply the hash function to generate an index where that key will be inserted in the hashtable.

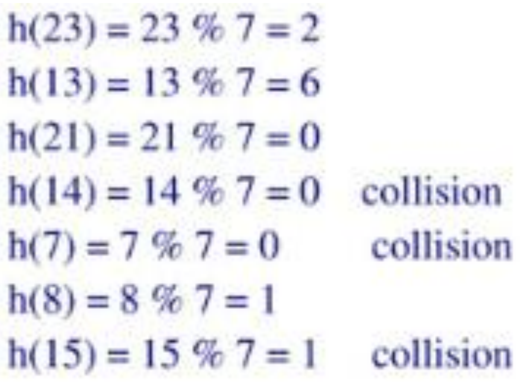
3. Insert Data:
   i) Calculate the index using the hash function.
   ii) Check if the computed index in the hash table is empty
      - If it's empty, place the key at that index.
      - If it's occupied (collision occurs), resolve the collision using a chosen method:
         - Separate Chaining: Add the key-value pair to a linked list at that index.
         - Open Addressing: Probe for the next available slot based on the probing technique (e.g., linear, quadratic, or double hashing).



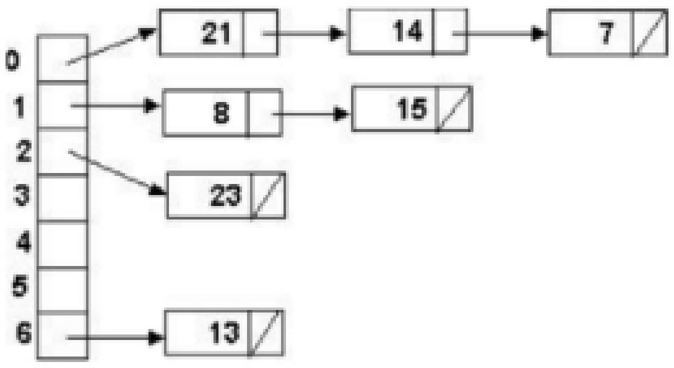
COLLISION RESOLUTION


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

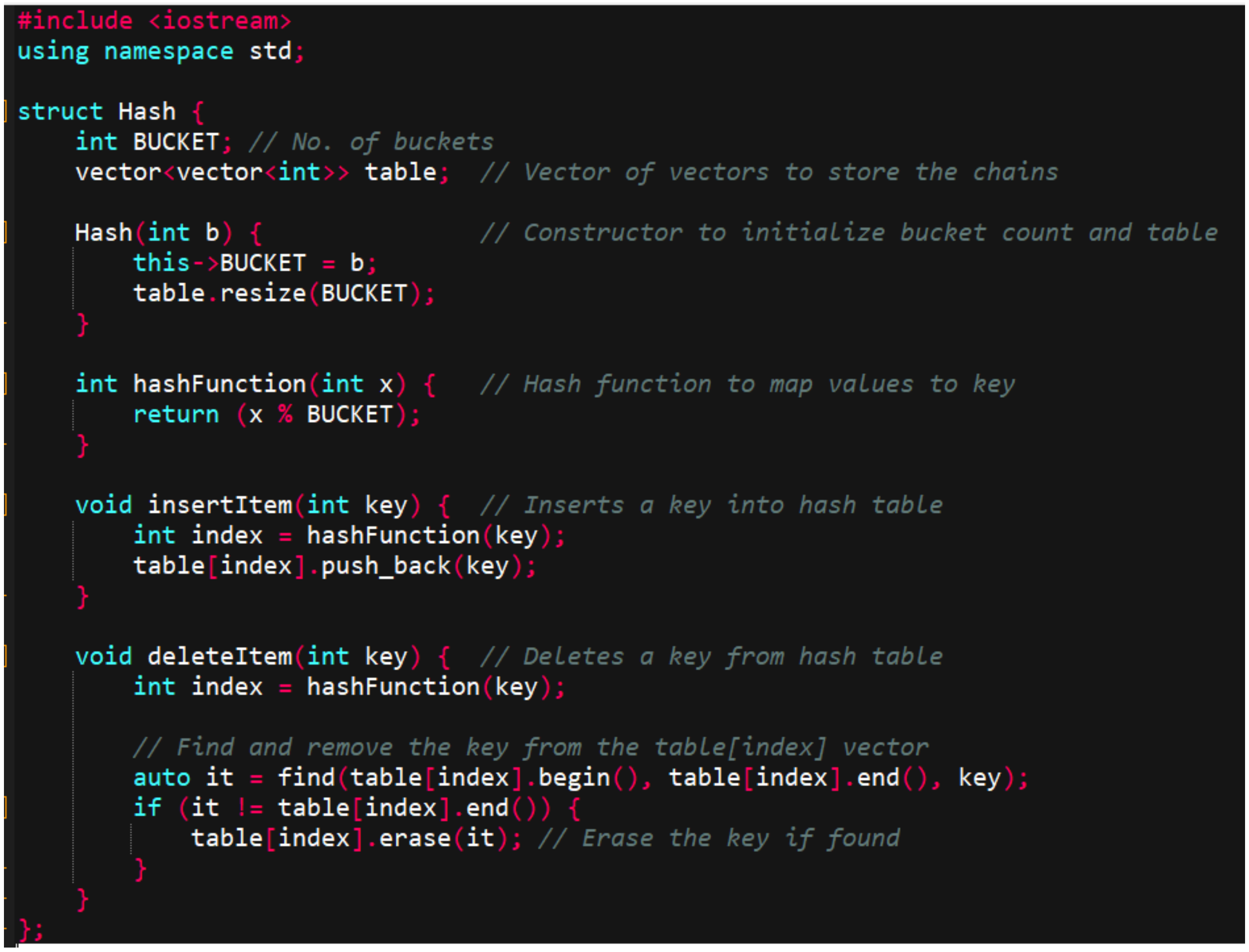
1. **SEPARATE CHAINING:** This method involves making a linked list out of the slot where the collision happened, then adding the new key to the list.
   - Time complexity: Its worst-case complexity for *searching* and *deletion* is o(n).
   - The hash table never fills full, so we can add more elements to the chain.
   - It requires more space for element links.



h(23) = 23 % 7 = 2
h(13) = 13 % 7 = 6
h(21) = 21 % 7 = 0
h(14) = 14 % 7 = 0   collision
h(7) = 7 % 7 = 0   collision
h(8) = 8 % 7 = 1
h(15) = 15 % 7 = 1   collision

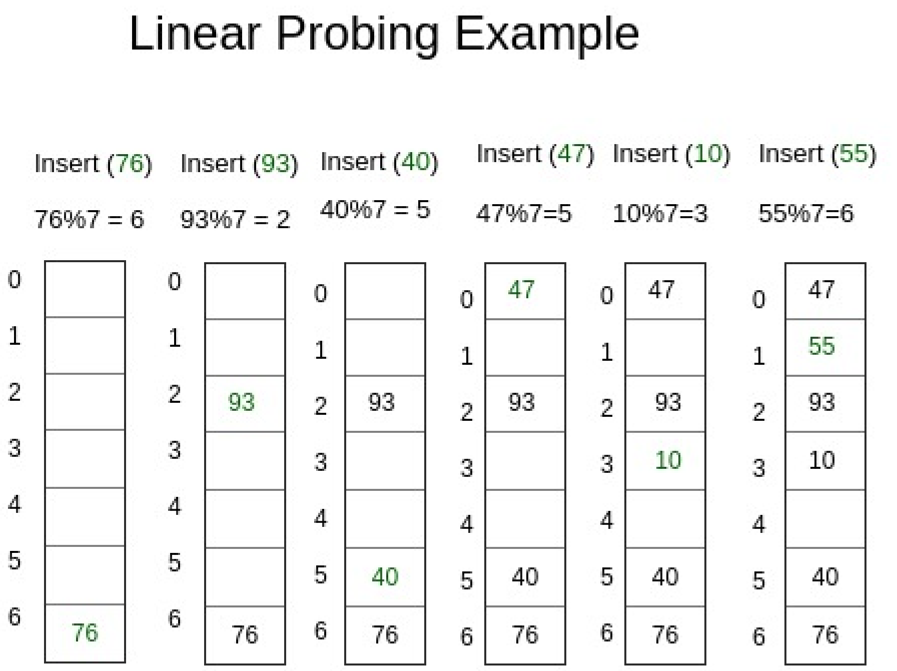


Code For Separate Chaining Using Vectors of Vectors:

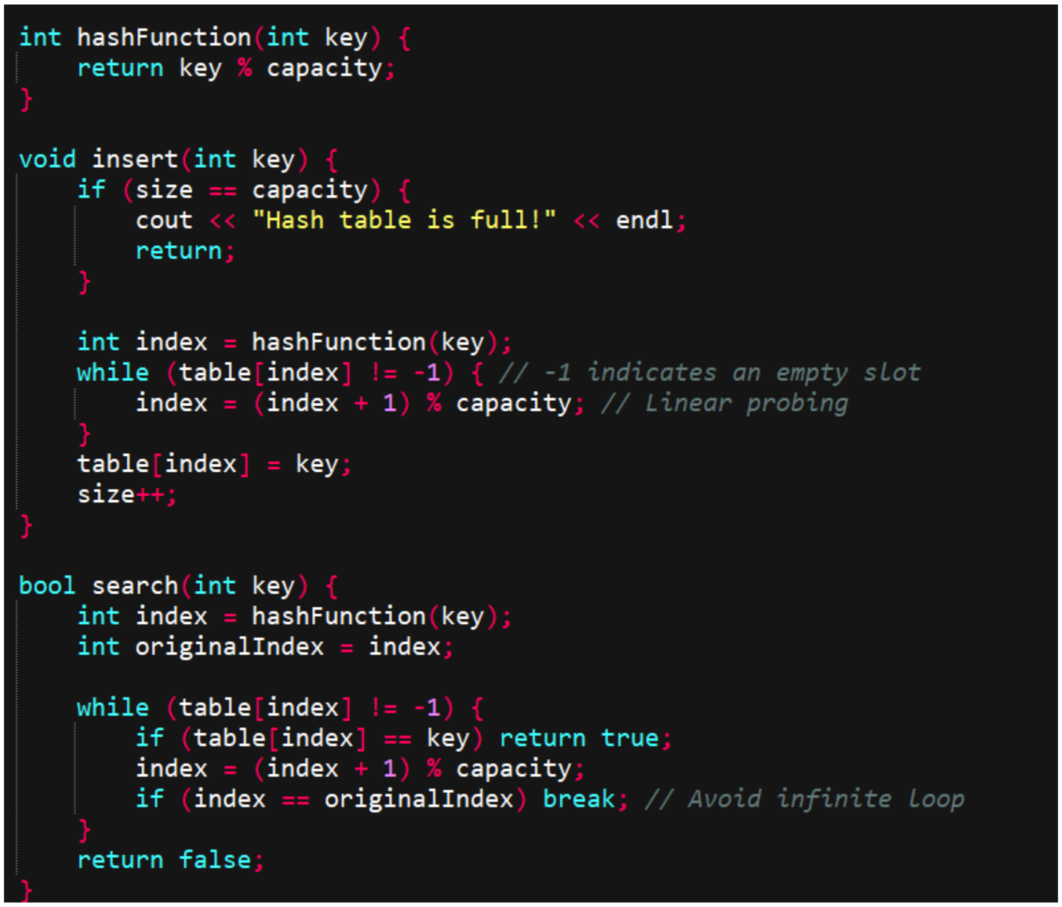


```cpp
#include <iostream>
using namespace std;

struct Hash {
    int BUCKET; // No. of buckets
    vector<vector<int>> table;  // Vector of vectors to store the chains

    Hash(int b) {               // Constructor to initialize bucket count and table
        this->BUCKET = b;
        table.resize(BUCKET);
    }

    int hashFunction(int x) {   // Hash function to map values to key
        return (x % BUCKET);
    }

    void insertItem(int key) {  // Inserts a key into hash table
        int index = hashFunction(key);
        table[index].push_back(key);
    }

    void deleteItem(int key) {  // Deletes a key from hash table
        int index = hashFunction(key);

        // Find and remove the key from the table[index] vector
        auto it = find(table[index].begin(), table[index].end(), key);
        if (it != table[index].end()) {
            table[index].erase(it); // Erase the key if found
        }
    }
};
```

2. **<u>OPEN ADDRESSING:</u>** To prevent collisions in the hashing table, open addressing is employed as a collision-resolution technique. No key is kept anywhere else besides the hash table. As a result, the hash table's size is never equal to or less than the number of keys. (Note that we can increase table size by copying old data if needed). Additionally known as closed hashing.



a) <u>Linear probing:</u> In linear probing, if a collision occurs at an index i, the algorithm checks the next slot (i + 1) % table_size, then (i + 2) % table_size, and so on until an empty slot is found. This technique often leads to **clustering**, where contiguous blocks of filled slots are formed, which can slow down search times

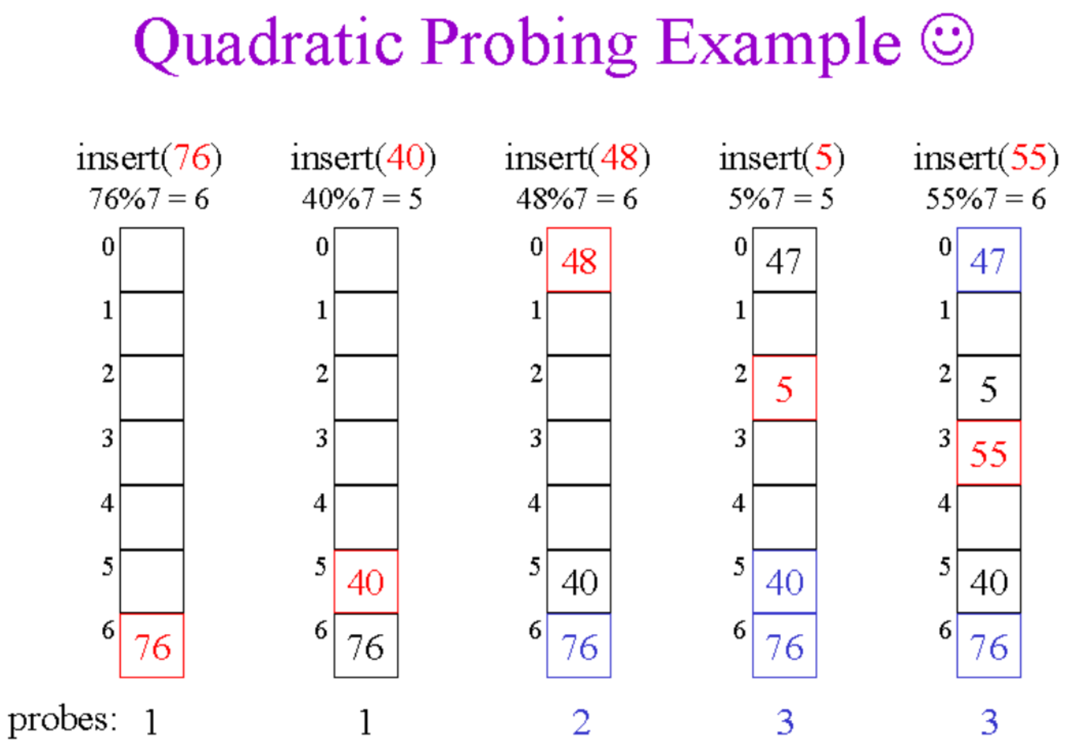


```
int hashFunction(int key) {
    return key % capacity;
}

void insert(int key) {
    if (size == capacity) {
        cout << "Hash table is full!" << endl;
        return;
    }

    int index = hashFunction(key);
    while (table[index] != -1) { // -1 indicates an empty slot
        index = (index + 1) % capacity; // Linear probing
    }
    table[index] = key;
    size++;
}

bool search(int key) {
    int index = hashFunction(key);
    int originalIndex = index;

    while (table[index] != -1) {
        if (table[index] == key) return true;
        index = (index + 1) % capacity;
        if (index == originalIndex) break; // Avoid infinite loop
    }
    return false;
}
```

b) Quadratic probing: When a collision occurs at index i, the next slots checked are

(i + 1^2) % table_size,
(i + 2^2) % table_size,
(i + 3^2) % table_size, and so on.

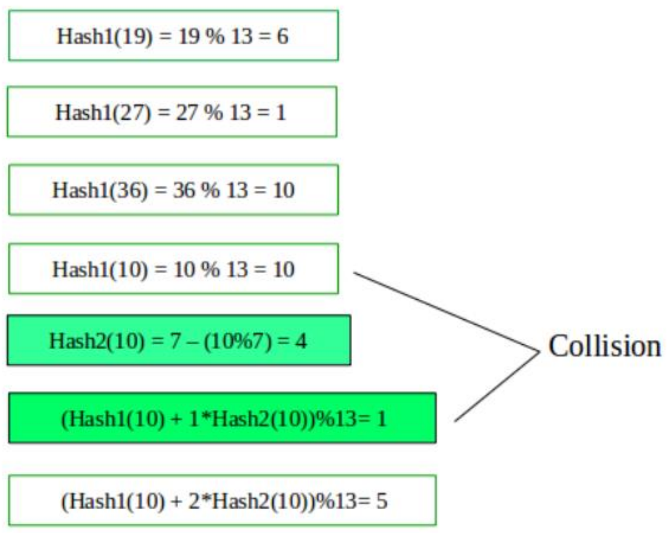
This spreads out the potential positions, reducing clustering but requiring a well-sized table to ensure all slots can be reached.



c) Double hashing: Double hashing uses a second hash function to calculate the step size for probing.When a collision occurs at index `i`, the next slot is determined by (i + j * hash2(key)) % table_size, where `hash2` is a secondary hash function, and `j` increments with each probe. Double hashing generally provides a good spread across the table and minimizes clustering.

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)



REHASHING

Rehashing is the process of resizing a hash table and reassigning all the elements to new positions within it. This is done to reduce the load factor, minimize collisions, and improve the performance of hash operations. In essence, rehashing involves creating a new, larger hash table and re-inserting each key-value pair from the old table using a new hash function or the same one adjusted to the new table size.

When is Rehashing Needed?

Rehashing is typically triggered when the load factor of the hash table reaches or exceeds a certain threshold, usually around 0.7 to 0.75. The load factor is defined as:

$$\text{Load Factor} = \frac{\text{Number of Elements in the Table}}{\text{Size of the Table}}$$

A high load factor means there are more elements relative to the number of slots, leading to a higher probability of collisions and therefore longer search times. Rehashing alleviates this by expanding the table size and redistributing elements.

Steps for Rehashing

1. Calculate the New Table Size
   - Generally, the new size is a prime number roughly double the current size. Prime numbers help in reducing clustering when using hash functions.
   - For instance, if the current table has 10 slots, the new table might have 23 or 29 slots.
2. Create the New Hash Table
   - Allocate a new hash table with the updated size.
3. Rehash All Existing Elements
   - For each element in the old hash table:
     - Calculate a new index using the hash function and the updated table size.
     - Insert the element at this new index in the new table.
   - This step can be time-consuming, as every element must be rehashed and inserted into the new table.
4. Replace the Old Table with the New Table
   - Once all elements have been rehashed into the new table, replace the old table with the new one.
   - Update any references to the old table, effectively freeing its memory.