NAME	Sujal Dingakar
UID no	2024301005
Experiment No	9

AIM:	Hashing Techniques: Write a program to implement Hash Table for the given input and solve thje collision using using quadratic probing and linear probing.

THEORY:

Why Do We Use Hashing?

- **Purpose of Hashing**: Explain why hashing is essential in data structures, especially in cases where fast data retrieval is required, like searching, inserting, or deleting elements.
- **Advantages Over Other Structures**: Describe how hashing provides constant-time complexity on average (O(1)) for operations, which is more efficient than structures like arrays or linked lists for large data sets.

What is Hashing and is type?

Hashing is a data storage and retrieval method that maps data to a fixed-size numerical value, called a hash, which serves as an index in a hash table. The hash function generates this value, allowing for quick data access in constant time, O(1), making hashing valuable for applications requiring fast lookup, such as databases and caches.

Types of Hashing Techniques for Collision Resolution

1. **Linear Probing:**
   - Concept: Moves sequentially to the next position until an empty spot is found.
   - Pros: Simple to implement.
   - Cons: May cause primary clustering, increasing lookup time as the table fills.
2. **Quadratic Probing:**
   - Concept: Moves at quadratic intervals (e.g., i+12,i+22i+1^2, i+2^2i+12,i+22) to reduce clustering.
   - Pros: Distributes data more evenly, reducing clustering.
   - Cons: Can lead to secondary clustering, with some slots possibly missed.
3. **Double Hashing:**
   - Concept: Uses a second hash function to calculate intervals, minimizing clustering.
   - Pros: Spreads elements more uniformly, reducing both primary and secondary clustering.
   - Cons: Requires two hash functions and the second hash must be non-zero to avoid loops.

What Makes a Good Hash Function?

	- Distribute Data Uniformly: Avoids clustering by spreading values evenly across the hash table. - Be Deterministic: Produces the same hash value for the same input every time. - Minimize Collisions: Reduces the likelihood of different inputs producing the same hash. - Be Efficient to Compute: Quickly generates the hash, keeping operations fast. - Use the Full Range of Table: Maximizes table utilization by covering all possible indices. **Advantages of Hashing Over Other Data Structures** **Efficiency:** Faster access to data compared to linear structures like linked lists or arrays. **Memory Usage:** Discuss the memory efficiency of hashing, especially when hash functions are well-optimized. **Suitability for Large Datasets:** Ideal for applications where high-speed access to large datasets is necessary. **Operations That Can Be Performed on Hashing** **Insertion**: How items are added to a hash table. **Searching:** How hashing allows for quick lookups of data. **Deletion:** How items are removed from a hash table and how the table manages collisions during deletions. **Handling Collisions:** Describe methods like chaining and open addressing to manage collisions, which occur when multiple items hash to the same index. **Applications of Hashing** **Data Indexing**: Used in databases to quickly locate records. **Cryptography:** Essential in encrypting data where a hash represents sensitive information. **Caching:** Used to cache data efficiently, such as in web applications to speed up access. **Checksum and Data Integrity:** Used to verify data integrity by comparing hashes. **What is Collision, and How is it Handled in Hashing?** A collision occurs in hashing when two different inputs produce the same hash value, causing them to map to the same index in a hash table. This can disrupt data retrieval, as it's unclear which value to access at the shared index. **Collisions are typically handled by collision resolution techniques such as:** **Chaining:** Stores multiple elements at the same index using linked lists. **Open Addressing:** Finds another open spot in the table based on a probing sequence (e.g., linear or quadratic probing, double hashing).

ALGORITHM:	**Graph Representation:** • Define a Node structure to represent each vertex's adjacency list. • Define a List structure to represent the linked list for each vertex. • Define a Graph structure that contains the number of vertices and an array of adjacency lists. **Queue Data Structure:** • Define a Queue structure for use in BFS traversal, including methods to enqueue and dequeue elements. **Graph Initialization:** • Create a graph using the createGraph function, which allocates memory for the graph and initializes the adjacency lists. **Adding Edges:** • Define the addEdge function to add directed edges between vertices. If you want an undirected graph, the same edge is added in both directions. **BFS Traversal:** • Initialize a queue to manage the BFS process. • Maintain an array to track the level of each vertex. • Use a visited array to keep track of visited vertices. • Start BFS from a specified vertex: o Dequeue a vertex, mark it as visited, and record its level. o Enqueue all unvisited adjacent vertices. • Print the BFS traversal order and levels of each vertex. **DFS Traversal:** • Initialize arrays to keep track of visited vertices, start times, and end times of each vertex. • Start DFS from a specified vertex: o Mark the vertex as visited, record its start time, and store it in the DFS traversal order. o Recursively visit all unvisited adjacent vertices. o Record the finish time for each vertex upon returning from the recursive call. • Print the DFS traversal order, start times, and end times for each vertex. **Input Handling:** • Get the number of vertices and edges from the user. • Read edges from the user to build the graph and adjacency matrix for BFS. • Ask the user for the starting vertex for both DFS and BFS. **Output:** • Print the adjacency list representation of the graph. • Print the BFS and DFS traversal orders and relevant timing information. **Memory Cleanup:** • Free allocated memory for the adjacency lists and the graph structure at the end of the program.
PROBLEM SOLVING:	1. Linear Probing solve.

* Linear Probing :-

key = 12, 22, 32, 50, 42, 60, 52, 70, 80, 62

① key = 12

12 % 10 = 2

Insert 12 at index 2

~~② key = 22~~

~~22 % 10 = 2 [collision]~~

~~h(k) 22~~

~~(h(22) + 0) mod 10 = 2 [collision]~~

~~(h(22) + 1) mod 10 = 2~~

~~2~~

② key = 22

22 % 10 = 2 [collision]

$\left.\begin{array}{l}(h(22)+0) \bmod 10 \\ (2+0) \bmod 10 = 2\end{array}\right\}$ Step 1

$(h(22) + 1) \bmod 10$
$\quad 2 + 1 \bmod 10 = 3$ } step 2

Insert 22 at 3

(3) key = 32

$32 \% 10 = 2$ (Collision)

$(h_2(32) + 0) \bmod 10$

$(2 + 0) \bmod 10 = 2$

$(h_2(32) + 1) \bmod 10$

$(2 + 1) \bmod 10 = 3$

$(h_2(32) + 2) \bmod 10$

$(2 + 2) \bmod 10 = 4$

Insert at key 32 at index 4

(4) key = 50

$50 \% 10 = 0$

Insert key 50 at index 0

(5) key = 42

$42 \% 10 = 2$ [collision]

$(h(42) + 0) \bmod 10$ [Formula]

$(2+0) \% 10 = 2$ [collision]

$(2+1) \% 10 = 3$ [collision]

$(2+2) \% 10 = 4$ [collision]

$(2+3) \% 10 = 5$

Insert 42 key at index 5

(6) key = 60

$60 \% 10 = 0$ [collision]

$(h(60) + 0) \% 10$ [Formula]

$(0+0) \% 10 = 0$ [collision]

$(0+1) \% 10 = 1$

Insert 60 at index 1

(7) Key = 52

$52 \% 10 = 2$ [collision]

$(h(52) + 0) \bmod 10$

$(2+0) \bmod 10 = 2$

$(2+1) \% 10 = 3$

$(2+2) \% 10 = 4$

$(2+3) \% 10 = 5$

[collision occur]

$(2+4) \% 10 = 6$

Insert 52 key at index 6

(8) Key = 70

$70 \% 10 = 0$ [collision]

$(h(70) + 0) \% 10 =$ (Formula)

$(0+0) \% 10 = 0$

$(0+1) \% 10 = 1$

$(0+2) \% 10 = 2$

$(0+3) \% 10 = 3$

[collision occur]

(5)

$(0+4)\%10 = 4$

$(0+5)\%10 = 5$

$(0+6)\%10 = 6$

$(0+7)\%10 = 7$

(collision occur)

Insert 70 key at index 7

5. key = 80

$80\%10 = 0$

$(h(80)+0) \bmod 10$ [Formula]

$(0+0) \bmod 10 = 0$

$(0+1) \bmod 10 = 1$

$\vdots$

$(0+7) \bmod 10 = 7$

collision occur

$(0+8) \bmod 10 = 8$

We insert 80 key at index 8

(10) key = 62

62 % 10 = 2 [collision]

~~h(62) % 10 =~~

(h(62) + i) mod 10 (Formula)

(2 + 0) mod 10 = 2

(2 + 1) mod 10 = 3

(2 + 2) mod 10 = 4

↓ [collision occur]

(2 + 6) mod 10 = 8

(2 + 7) mod 10 = 9

We insert key 62 at index 9

index	
0	50
1	60
2	12
3	22
4	32
5	42
6	52
7	70
8	80
9	62

Final hash table

2.Quadratic Probing..

Quadratic Probing :-

Keys => 12, 22, 32, 50, 42, 60, 52, 70, 80, 62

① key = 12

12 % 10 = 2

Insert 12 at index 2

② key = 22

22 % 10 = 2

Step 1 $(h(22) + i^2) \% 10$ [Formula]

$(2 + 0^2) \% 10 = 2$ [Collision occur]

Step 2 $(2 + 1^2) \% 10 = 3$

Insert 22 at index 3

③ key = 32

32 % 10 = 2

Step 1 $(h(32) + i^2) \% 10$ [Formula]

$(2 + 0^2) \% 10 = 2$

$(2 + 1^2) \% 10 = 3$ } [Collision]

$(2+2^2)\%10 = 6$

Insert x at index 6

(4) key = 50

$50\%10 = 0$

$\cancel{(h(50)+i^2) \bmod 10}$ ~~(Formula)~~

$\cancel{(0+0^2)\%10 = 0}$

$\cancel{(0+1^2)\%10 = 1}$

Insert 50 at index 0

(5) key = 42

$42\%10 = 2$ [collision]

$(h(42)+i^2) \bmod 10$ [Formula]

$$\left.\begin{array}{l}(2+0^2)\%10 = 2\\(2+1^2)\%10 = 3\\(2+2^2)\%10 = 6\end{array}\right\}\text{(collision occur)}$$

$(2+3^2)\%10 = 1$

Insert 42 at index 1

⑥ key = 60.

$60 \% 10 = 0$ [collision]

$(h(60) + i^2) \bmod 10$ [Formula]

$(0 + 0^2) \% 10 = 0$
$(0 + 1^2) \% 10 = 1$ } [collision over]

$(0 + 2^2) \% 10 = 4$

~~for~~ We insert 60 key at index 4

⑦ key = 52

$52 \% 10 = 2$ [collision]

$(h(52) + i^2) \bmod 10$ [Formula]

$(2+0^2) \% 10 = 2$
$(2+1^2) \% 10 = 3$
$(2+2^2) \% 10 = 6$
$(2+3^2) \% 10 = 1$ } [We got collosion]

$(2+4^2) \% 10 = 8$ [New index]

We insert 52 at index ~~5~~ 8

⑧ key = 70

$70 \% 10 = 0$ (collision)

$(h(70) + i^2) \bmod 10$ (Formula)

$$\left.\begin{aligned} (0 + 0^2) \% 10 &= 0 \\ (0 + 1^2) \% 10 &= 1 \\ (0 + 2^2) \% 10 &= 4 \end{aligned}\right\} \text{(collision)}$$

$(0 + 3^2) \% 10 = 9$

Insert key 70 at index 9.

⑨ key = 80

$80 \% 10 = 0$ (collision)

$(h(80) + i^2) \bmod 10$ (Formula)

$$\left.\begin{aligned} (0 + 0^2) \% 10 &= 0 \\ (0 + 1^2) \% 10 &= 1 \\ (0 + 2^2) \% 10 &= 4 \\ (0 + 3^2) \% 10 &= 9 \\ (0 + 4^2) \% 10 &= 6 \end{aligned}\right\} \text{(collision occur)}$$

$(0 + 5^2) \% 10 = 5$

we insert key 80 at index 5

	(18) key = 62 62 % 10 = 2. [collision] (h(62) + i^2) mod 10 [Formula] ~~(2 + 0^2) % 10 = 2~~ ~~(2 + 0^2~~ $(2+0^2)\ \%\ 10 = 2$ $(2+1^2)\ \%\ 10 = 3$ $(2+2^2)\ \%\ 10 = 6$ $(2+3^2)\ \%\ 10 = 1$ $(2+4^2)\ \%\ 10 = 8$ [collision occur] $(2+5^2)\ \%\ 10 = 7$ we insert 62 at index 7 Final hash table Index | key 0 | 50 1 | 42 2 | 12 3 | 22 4 | 60 5 | 80 6 | 32 7 | 62 8 | 52 9 | 70 3.Double Probing..

Double Hashing :-

keys ⇒ 12, 22, 32, 50, 42, 60, 52, 70, 80, 62

Here we use 2 hash function

* steps to solve :-

step1 ⇒ $h_1 = x \% 10$

step2 ⇒ h_2 = Prime − (key % Prime)

step3 ⇒ $(h_1(x) + i * h_2(x)) \% 10$

① key = 12

$h_1 = x \% 10 = 0$
$h_1 = 12 \% 10 = 2$

We insert 12 at index 2

② key = 22

$h_1 = x \% 10 = 0$
$h_1 = 22 \% 10 = 2$

Collision occur at index 2

Now we use 2^{nd} hash function.

$h_2 = 7 - (22 \% 7)$
$h_2 = 7 - 1$
$h_2 = 6$

$\cancel{(h_1(22) + 0 * 6) \%}$

$\cancel{(h_1(22) + 0 * 6) \% 10}$

$\cancel{h_1}$

$$\left.\begin{aligned} &(h_1(22) + 0 * 6) \% 10 \\ &(2 + 0) \% 10 \\ &2 \% 10 = 2 \end{aligned}\right\} \text{step 1}$$

Collision occur at index 2

$$\left.\begin{aligned} &(h_1(22) + 1 * 6) \% 10 \\ &(2 + 1 * 6) \% 10 \\ &(2 + 6) \% 10 \\ &8 \% 10 = 8 \end{aligned}\right\} \text{step 2}$$

we insert 8 index the element 22

③ keys = 32

$h_1 = x \% 10$
$= 32 \% 10$
$h_1 = 2$

Collision occur at index at 2

we go for 2nd hash function

$h_2 = Prime - (key \% Prime)$
$= 7 - (32 \% 7)$
$= 7 - 4$
$h_2 = 3$

* $(h_1(x) + i * h_2(x)) \% 10$

$(h_1(32) + 0 * 3) \% 10$
$(2 + 0) \% 10$
$2 \% 10 = 2$
step 1

Collision occur at index 2

$(h_1(32) + 1 * 3) \% 10$
$(2 + 3) \% 10$
$5 \% 10 = 5$
step 2

we insert 32 key at index 5

(4) key = 50

$h_1 = x \% 10$

$= 50 \% 10$

~~$h_1 = 5$~~

$h_1 = 0$

We insert 50 key at index 0

(5) key = 42

$h_1 = x \% 10$

$= 42 \% 10$

$h_1 = 2$

Collision occur at index 2

we use 2nd hash function

$h_2 = \text{Prime} - (\text{key} \% \text{Prime})$

$= 7 - (42 \% 7)$

$= 7 - 0$

$h_2 = 7$

~~$h_1(42) + i * h_2(x) =$~~

$h_1(x) + i * h_2(x) \% 10$

$(2+0*7)\%10$

$(2+0)\%10 = 2$ } Step 1

Collision occur at index 2

$= (2+1*7)\%10$
$= (2+7)\%10$
$= 9\%10$
$= 9$ } Step 2

We insert 42 key at index 9

⑥ key = 60

$h_1 = x\%10$
$= 60\%10$
$h_1 = 0$

Collision occur at index 0

$h_2 = Prime - (key\ \%\ Prime)$
$= 7 - (60\%7)$
$= 7 - 4$
$h_2 = 3$

$h_1(x) + i * h_2(x)\ \%10$

$((0) + 0*3)\%10$
$0 + 0\%10$
$0\%10 = 0$ } Step 1

$(0+1*3)\%10$

$(0+3)\%10$

$3\%10 = 3$

Step 2

(7) Key = 52

$h_1 = x\%10$

$= 52\%10$

$h_1 = 2$ Collision occur at index 2

$h_2 = \text{Prime} - (\text{Key} \% \text{Prime})$

$= 7 - (52 \% 7)$

$= 7 - 3$

$h_2 = 4$

$(h_1(x) + i * h_2(x))\%10$

$(2+0*4)\%10$

$2+0\%10$

$2\%10 = 2$

Collision occur at index 2

Step 1

$(2+1*4)\%10$

$(2+4)\%10$

$6\%10$

6

Step 2

so insert 52 at index 6

(8) key = 70

$h_1 = 70 \% 10 = 0$ Collision

$h_2 = 7 - (70 \div 7)$
$= 7 - (0)$
$h_2 = 7$

$h_1(x) + i * h_2(x) \% 10$ [Formula]

$\left.\begin{array}{l} ((0) + 0 * 7) \% 10 \\ 0 + 0 \% 10 \\ 0 \% 10 = 0 \\ \text{Collision occur at } 0 \end{array}\right\}$ step 1

$\left.\begin{array}{l} (0) + 1 * 7) \% 10 \\ 7 \% 10 = 7 \\ \text{Insert 70 key at index } 7 \end{array}\right\}$ step 2

(9) key = 80

$h_1 = x \% 10$
$= 80 \% 10$
$h_1 = 0$ [Collision]

$h_2 = 7 - (80 \% 7)$
$= 7 - 3$
$h_2 = 4$

$h_1(x) + i * h_2(x) \% 10$ [Formula]

$$\left.\begin{array}{l}(0 + 0 * 4) \% 10 \\ (0 + 0) \% 10 \\ 0 \% 10 = 0\end{array}\right\} \text{step 1}$$

Collision occur at 0

$$\left.\begin{array}{l}(0 + 1 * 4) \% 10 \\ 4 \% 10 = 4\end{array}\right\} \text{step 2}$$

Insert 80 key at index 4

(10) key = 62

$h_1(62) = 2$

$h_2(62) = 7 - (62 \bmod 7) = 1$

$((2) + 0 * 1) \% 10 = 2$ [Collision]

$(2 + 1 * 1) \% 10 = 3$ [Collision]

$(2 + 2 * 1) \% 10 = 4$ [Collision]

⋮

$(2 + 9 * 1) \% 10 = 1$

we insert 62 at index 1

0	50
1	62
2	12
3	60
4	80
5	32
6	52
7	70
8	22
9	42

Final hash table

PROGRAM:

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

#define TABLE_SIZE 10
#define PRIME 7

typedef struct {
    int *table;
    bool *occupied;
} HashTable;
```

```c
int hash(int key) {
    return key % TABLE_SIZE;
}

HashTable* createHashTable() {
    HashTable* ht = (HashTable*)malloc(sizeof(HashTable));
    ht->table = (int*)malloc(sizeof(int) * TABLE_SIZE);
    ht->occupied = (bool*)malloc(sizeof(bool) * TABLE_SIZE);
    for (int i = 0; i < TABLE_SIZE; i++) {
        ht->table[i] = -1;
        ht->occupied[i] = false;
    }
    return ht;
}


// Linear Probing
bool linearProbingInsert(HashTable* ht, int key) {
    int idx = hash(key);
    int original_idx = idx;
    int step = -1;

    // Show initial hash calculation
    printf("%d %% %d = %d\n", key, TABLE_SIZE, key % TABLE_SIZE);

    while (true) {
        // After collision, calculate next position using linear probing
        step++;
        idx = (original_idx + step) % TABLE_SIZE;
        printf("Step %d: (h(%d) + %d) mod %d = %d\n",
               step, key, step, TABLE_SIZE, idx);

        // Check if current position is empty
        if (!ht->occupied[idx]) {
            ht->table[idx] = key;
            ht->occupied[idx] = true;
            printf("Inserted %d at index %d\n", key, idx);
            return true;
        }

        // Position is occupied - show collision
        printf("Collision occurred: position %d is occupied by %d\n",
               idx, ht->table[idx]);

        // Check if table is full
        if (step >= TABLE_SIZE - 1) {
            printf("Table is full! Cannot insert %d\n", key);
            return false;
        }
```

```
    }
}
// Quadratic Probing with detailed index calculation steps
bool quadraticProbingInsert(HashTable* ht, int key) {
    int idx = hash(key);
    int original_idx = idx;
    int i = 0;

    while (ht->occupied[idx]) {
        printf("Collision occurred for %d at index %d\n", key, idx);
        printf("Step %d: Original index = %d\n", i, original_idx);

        int probe_value = (original_idx + i * i) % TABLE_SIZE;
        printf("Step %d: (%d + %d^2) mod %d = %d (new index)\n", i, original_idx, i,
TABLE_SIZE, probe_value);

        idx = probe_value;
        i++;

        if (i >= TABLE_SIZE) {
            return false;
        }
    }

    ht->table[idx] = key;
    ht->occupied[idx] = true;
    printf("Inserted %d at index %d\n", key, idx);
    return true;
}

// Double Hashing
int secondHash(int key) {
    return PRIME - (key % PRIME);
}

bool doubleHashingInsert(HashTable* ht, int key) {
    int idx = hash(key);
    int original_idx = idx;
    int step = secondHash(key);
    int i = 0;

    // Display initial hashing values
    printf("Inserting %d:\n", key);
    printf("h1(%d) = %d\n", key, idx);

    while (ht->occupied[idx]) {
        printf("Collision occurred for %d at index %d\n", key, idx);

        printf("h2(%d) = %d - (%d mod %d) = %d\n", key, PRIME, key, PRIME, step);
```

```
        // Calculate the new index
        idx = (original_idx + (i * step)) % TABLE_SIZE;

        // Show new index calculation
        printf("Step %d: (h(%d) + %d * %d) mod %d = %d\n", i + 1, original_idx, i, step,
TABLE_SIZE, idx);

        // Increment the probe count
        i++;

        // Check if we have looped back to the original index

    }

    // If the slot is empty, insert the key
    ht->table[idx] = key;
    ht->occupied[idx] = true;
    printf("Inserted %d at index %d\n", key, idx); // Print the final index
    return true;
}


// Display the hash table with a title
void displayHashTable(HashTable* ht, const char* title) {
    printf("%s:\n", title);
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (ht->occupied[i]) {
            printf("Index %d: %d\n", i, ht->table[i]);
        } else {
            printf("Index %d: Empty\n", i);
        }
    }
    printf("\n");
}

// Free the hash table
void freeHashTable(HashTable* ht) {
    free(ht->table);
    free(ht->occupied);
    free(ht);
}

int main() {
    HashTable* ht = createHashTable();
    int choice, numKeys, key;

    while (1) {
        printf("Menu:\n");
        printf("1. Insert elements into the hash table\n");
        printf("2. Print the hash table\n");
```

```
printf("3. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter the number of keys to insert (max %d): ", TABLE_SIZE);
        scanf("%d", &numKeys);
        if (numKeys > TABLE_SIZE) {
            printf("Number of keys exceeds table size.\n");
            break;
        }

        printf("Choose probing method:\n");
        printf("1. Linear Probing\n");
        printf("2. Quadratic Probing\n");
        printf("3. Double Hashing\n");
        printf("Enter your choice: ");
        int probingChoice;
        scanf("%d", &probingChoice);

        if (probingChoice < 1 || probingChoice > 3) {
            printf("Invalid probing method.\n");
            break;
        }

        // Prompt user for keys
        for (int i = 0; i < numKeys; i++) {
            printf("Enter key %d: ", i + 1);
            scanf("%d", &key);
            switch (probingChoice) {
                case 1:
                    linearProbingInsert(ht, key);
                    break;
                case 2:
                    quadraticProbingInsert(ht, key);
                    break;
                case 3:
                    doubleHashingInsert(ht, key);
                    break;
            }
        }
        break;

    case 2:
        displayHashTable(ht, "Hash Table");
        break;

    case 3:
        printf("Exist from loop..\n");
```

<table>
<tr><td></td><td>

```
            return(0);
            break;

        default:
            printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

</td></tr>
<tr><td>RESULT :-</td><td>1.Linear Probing..</td></tr>
</table>

```
Menu:
1. Insert elements into the hash table
2. Print the hash table
3. Exit
Enter your choice: 1
Enter the number of keys to insert (max 10): 10
Choose probing method:
1. Linear Probing
2. Quadratic Probing
3. Double Hashing
Enter your choice: 1
Enter key 1: 12
12 % 10 = 2
Step 0: (h(12) + 0) mod 10 = 2
Inserted 12 at index 2
Enter key 2: 22
22 % 10 = 2
Step 0: (h(22) + 0) mod 10 = 2
Collision occurred: position 2 is occupied by 12
Step 1: (h(22) + 1) mod 10 = 3
Inserted 22 at index 3
Enter key 3: 32
32 % 10 = 2
Step 0: (h(32) + 0) mod 10 = 2
Collision occurred: position 2 is occupied by 12
Step 1: (h(32) + 1) mod 10 = 3
Collision occurred: position 3 is occupied by 22
Step 2: (h(32) + 2) mod 10 = 4
Inserted 32 at index 4
Enter key 4: 50
50 % 10 = 0
Step 0: (h(50) + 0) mod 10 = 0
Inserted 50 at index 0
Enter key 5: 42
42 % 10 = 2
Step 0: (h(42) + 0) mod 10 = 2
Collision occurred: position 2 is occupied by 12
Step 1: (h(42) + 1) mod 10 = 3
Collision occurred: position 3 is occupied by 22
Step 2: (h(42) + 2) mod 10 = 4
Collision occurred: position 4 is occupied by 32
Step 3: (h(42) + 3) mod 10 = 5
Inserted 42 at index 5
```

```
Enter key 6: 60
60 % 10 = 0
Step 0: (h(60) + 0) mod 10 = 0
Collision occurred: position 0 is occupied by 50
Step 1: (h(60) + 1) mod 10 = 1
Inserted 60 at index 1
Enter key 7: 52
52 % 10 = 2
Step 0: (h(52) + 0) mod 10 = 2
Collision occurred: position 2 is occupied by 12
Step 1: (h(52) + 1) mod 10 = 3
Collision occurred: position 3 is occupied by 22
Step 2: (h(52) + 2) mod 10 = 4
Collision occurred: position 4 is occupied by 32
Step 3: (h(52) + 3) mod 10 = 5
Collision occurred: position 5 is occupied by 42
Step 4: (h(52) + 4) mod 10 = 6
Inserted 52 at index 6
```

	`` Enter key 8: 70 70 % 10 = 0 Step 0: (h(70) + 0) mod 10 = 0 Collision occurred: position 0 is occupied by 50 Step 1: (h(70) + 1) mod 10 = 1 Collision occurred: position 1 is occupied by 60 Step 2: (h(70) + 2) mod 10 = 2 Collision occurred: position 2 is occupied by 12 Step 3: (h(70) + 3) mod 10 = 3 Collision occurred: position 3 is occupied by 22 Step 4: (h(70) + 4) mod 10 = 4 Collision occurred: position 4 is occupied by 32 Step 5: (h(70) + 5) mod 10 = 5 Collision occurred: position 5 is occupied by 42 Step 6: (h(70) + 6) mod 10 = 6 Collision occurred: position 6 is occupied by 52 Step 7: (h(70) + 7) mod 10 = 7 Inserted 70 at index 7 Enter key 9: 80 80 % 10 = 0 Step 0: (h(80) + 0) mod 10 = 0 Collision occurred: position 0 is occupied by 50 Step 1: (h(80) + 1) mod 10 = 1 Collision occurred: position 1 is occupied by 60 Step 2: (h(80) + 2) mod 10 = 2 Collision occurred: position 2 is occupied by 12 Step 3: (h(80) + 3) mod 10 = 3 Collision occurred: position 3 is occupied by 22 Step 4: (h(80) + 4) mod 10 = 4 Collision occurred: position 4 is occupied by 32 Step 5: (h(80) + 5) mod 10 = 5 Collision occurred: position 5 is occupied by 42 Step 6: (h(80) + 6) mod 10 = 6 Collision occurred: position 6 is occupied by 52 Step 7: (h(80) + 7) mod 10 = 7 Collision occurred: position 7 is occupied by 70 Step 8: (h(80) + 8) mod 10 = 8 Inserted 80 at index 8 ``` ``` Enter key 10: 62 62 % 10 = 2 Step 0: (h(62) + 0) mod 10 = 2 Collision occurred: position 2 is occupied by 12 Step 1: (h(62) + 1) mod 10 = 3 Collision occurred: position 3 is occupied by 22 Step 2: (h(62) + 2) mod 10 = 4 Collision occurred: position 4 is occupied by 32 Step 3: (h(62) + 3) mod 10 = 5 Collision occurred: position 5 is occupied by 42 Step 4: (h(62) + 4) mod 10 = 6 Collision occurred: position 6 is occupied by 52 Step 5: (h(62) + 5) mod 10 = 7 Collision occurred: position 7 is occupied by 70 Step 6: (h(62) + 6) mod 10 = 8 Collision occurred: position 8 is occupied by 80 Step 7: (h(62) + 7) mod 10 = 9 Inserted 62 at index 9 ```

```
Menu:
1. Insert elements into the hash table
2. Print the hash table
3. Exit
Enter your choice: 2
Hash Table:
Index 0: 50
Index 1: 60
Index 2: 12
Index 3: 22
Index 4: 32
Index 5: 42
Index 6: 52
Index 7: 70
Index 8: 80
Index 9: 62
```

2.**Quadratic Probing…**

```
Menu:
1. Insert elements into the hash table
2. Print the hash table
3. Exit
Enter your choice: 1
Enter the number of keys to insert (max 10): 10
Choose probing method:
1. Linear Probing
2. Quadratic Probing
3. Double Hashing
Enter your choice: 2
Enter key 1: 12
Inserted 12 at index 2
Enter key 2: 22
Collision occurred for 22 at index 2
Step 0: Original index = 2
Step 0: (2 + 0^2) mod 10 = 2 (new index)
Collision occurred for 22 at index 2
Step 1: Original index = 2
Step 1: (2 + 1^2) mod 10 = 3 (new index)
Inserted 22 at index 3
Enter key 3: 32
Collision occurred for 32 at index 2
Step 0: Original index = 2
Step 0: (2 + 0^2) mod 10 = 2 (new index)
Collision occurred for 32 at index 2
Step 1: Original index = 2
Step 1: (2 + 1^2) mod 10 = 3 (new index)
Collision occurred for 32 at index 3
Step 2: Original index = 2
Step 2: (2 + 2^2) mod 10 = 6 (new index)
Inserted 32 at index 6
Enter key 4: 50
Inserted 50 at index 0
```

```
Enter key 5: 42
Collision occurred for 42 at index 2
Step 0: Original index = 2
Step 0: (2 + 0^2) mod 10 = 2 (new index)
Collision occurred for 42 at index 2
Step 1: Original index = 2
Step 1: (2 + 1^2) mod 10 = 3 (new index)
Collision occurred for 42 at index 3
Step 2: Original index = 2
Step 2: (2 + 2^2) mod 10 = 6 (new index)
Collision occurred for 42 at index 6
Step 3: Original index = 2
Step 3: (2 + 3^2) mod 10 = 1 (new index)
Inserted 42 at index 1
Enter key 6: 60
Collision occurred for 60 at index 0
Step 0: Original index = 0
Step 0: (0 + 0^2) mod 10 = 0 (new index)
Collision occurred for 60 at index 0
Step 1: Original index = 0
Step 1: (0 + 1^2) mod 10 = 1 (new index)
Collision occurred for 60 at index 1
Step 2: Original index = 0
Step 2: (0 + 2^2) mod 10 = 4 (new index)
Inserted 60 at index 4
```

```
Enter key 7: 52
Collision occurred for 52 at index 2
Step 0: Original index = 2
Step 0: (2 + 0^2) mod 10 = 2 (new index)
Collision occurred for 52 at index 2
Step 1: Original index = 2
Step 1: (2 + 1^2) mod 10 = 3 (new index)
Collision occurred for 52 at index 3
Step 2: Original index = 2
Step 2: (2 + 2^2) mod 10 = 6 (new index)
Collision occurred for 52 at index 6
Step 3: Original index = 2
Step 3: (2 + 3^2) mod 10 = 1 (new index)
Collision occurred for 52 at index 1
Step 4: Original index = 2
Step 4: (2 + 4^2) mod 10 = 8 (new index)
Inserted 52 at index 8
```

```
Enter key 8: 70
Collision occurred for 70 at index 0
Step 0: Original index = 0
Step 0: (0 + 0^2) mod 10 = 0 (new index)
Collision occurred for 70 at index 0
Step 1: Original index = 0
Step 1: (0 + 1^2) mod 10 = 1 (new index)
Collision occurred for 70 at index 1
Step 2: Original index = 0
Step 2: (0 + 2^2) mod 10 = 4 (new index)
Collision occurred for 70 at index 4
Step 3: Original index = 0
Step 3: (0 + 3^2) mod 10 = 9 (new index)
Inserted 70 at index 9
Enter key 9: 80
Collision occurred for 80 at index 0
Step 0: Original index = 0
Step 0: (0 + 0^2) mod 10 = 0 (new index)
Collision occurred for 80 at index 0
Step 1: Original index = 0
Step 1: (0 + 1^2) mod 10 = 1 (new index)
Collision occurred for 80 at index 1
Step 2: Original index = 0
Step 2: (0 + 2^2) mod 10 = 4 (new index)
Collision occurred for 80 at index 4
Step 3: Original index = 0
Step 3: (0 + 3^2) mod 10 = 9 (new index)
Collision occurred for 80 at index 9
Step 4: Original index = 0
Step 4: (0 + 4^2) mod 10 = 6 (new index)
Collision occurred for 80 at index 6
Step 5: Original index = 0
Step 5: (0 + 5^2) mod 10 = 5 (new index)
Inserted 80 at index 5
```

```
Enter key 10: 62
Collision occurred for 62 at index 2
Step 0: Original index = 2
Step 0: (2 + 0^2) mod 10 = 2 (new index)
Collision occurred for 62 at index 2
Step 1: Original index = 2
Step 1: (2 + 1^2) mod 10 = 3 (new index)
Collision occurred for 62 at index 3
Step 2: Original index = 2
Step 2: (2 + 2^2) mod 10 = 6 (new index)
Collision occurred for 62 at index 6
Step 3: Original index = 2
Step 3: (2 + 3^2) mod 10 = 1 (new index)
Collision occurred for 62 at index 1
Step 4: Original index = 2
Step 4: (2 + 4^2) mod 10 = 8 (new index)
Collision occurred for 62 at index 8
Step 5: Original index = 2
Step 5: (2 + 5^2) mod 10 = 7 (new index)
Inserted 62 at index 7
```

```
Menu:
1. Insert elements into the hash table
2. Print the hash table
3. Exit
Enter your choice: 2
Hash Table:
Index 0: 50
Index 1: 42
Index 2: 12
Index 3: 22
Index 4: 60
Index 5: 80
Index 6: 32
Index 7: 62
Index 8: 52
Index 9: 70
```

3.**Double Probing..**

```
Menu:
1. Insert elements into the hash table
2. Print the hash table
3. Exit
Enter your choice: 1
Enter the number of keys to insert (max 10): 10
Choose probing method:
1. Linear Probing
2. Quadratic Probing
3. Double Hashing
Enter your choice: 3
Enter key 1: 12
Inserting 12:
h1(12) = 2
Inserted 12 at index 2
Enter key 2: 22
Inserting 22:
h1(22) = 2
Collision occurred for 22 at index 2
h2(22) = 7 - (22 mod 7) = 6
Step 1: (h(2) + 0 * 6) mod 10 = 2
Collision occurred for 22 at index 2
h2(22) = 7 - (22 mod 7) = 6
Step 2: (h(2) + 1 * 6) mod 10 = 8
Inserted 22 at index 8
```

	`` Enter key 3: 32 Inserting 32: h1(32) = 2 Collision occurred for 32 at index 2 h2(32) = 7 - (32 mod 7) = 3 Step 1: (h(2) + 0 * 3) mod 10 = 2 Collision occurred for 32 at index 2 h2(32) = 7 - (32 mod 7) = 3 Step 2: (h(2) + 1 * 3) mod 10 = 5 Inserted 32 at index 5 Enter key 4: 50 Inserting 50: h1(50) = 0 Inserted 50 at index 0 Enter key 5: 42 Inserting 42: h1(42) = 2 Collision occurred for 42 at index 2 h2(42) = 7 - (42 mod 7) = 7 Step 1: (h(2) + 0 * 7) mod 10 = 2 Collision occurred for 42 at index 2 h2(42) = 7 - (42 mod 7) = 7 Step 2: (h(2) + 1 * 7) mod 10 = 9 Inserted 42 at index 9 Enter key 6: 60 Inserting 60: h1(60) = 0 Collision occurred for 60 at index 0 h2(60) = 7 - (60 mod 7) = 3 Step 1: (h(0) + 0 * 3) mod 10 = 0 Collision occurred for 60 at index 0 h2(60) = 7 - (60 mod 7) = 3 Step 2: (h(0) + 1 * 3) mod 10 = 3 Inserted 60 at index 3 ``` ``` Enter key 7: 52 Inserting 52: h1(52) = 2 Collision occurred for 52 at index 2 h2(52) = 7 - (52 mod 7) = 4 Step 1: (h(2) + 0 * 4) mod 10 = 2 Collision occurred for 52 at index 2 h2(52) = 7 - (52 mod 7) = 4 Step 2: (h(2) + 1 * 4) mod 10 = 6 Inserted 52 at index 6 Enter key 8: 70 Inserting 70: h1(70) = 0 Collision occurred for 70 at index 0 h2(70) = 7 - (70 mod 7) = 7 Step 1: (h(0) + 0 * 7) mod 10 = 0 Collision occurred for 70 at index 0 h2(70) = 7 - (70 mod 7) = 7 Step 2: (h(0) + 1 * 7) mod 10 = 7 Inserted 70 at index 7 ```

```
Enter key 9: 80
Inserting 80:
h1(80) = 0
Collision occurred for 80 at index 0
h2(80) = 7 - (80 mod 7) = 4
Step 1: (h(0) + 0 * 4) mod 10 = 0
Collision occurred for 80 at index 0
h2(80) = 7 - (80 mod 7) = 4
Step 2: (h(0) + 1 * 4) mod 10 = 4
Inserted 80 at index 4
Enter key 10: 62
```

```
Inserting 62:
h1(62) = 2
Collision occurred for 62 at index 2
h2(62) = 7 - (62 mod 7) = 1
Step 1: (h(2) + 0 * 1) mod 10 = 2
Collision occurred for 62 at index 2
h2(62) = 7 - (62 mod 7) = 1
Step 2: (h(2) + 1 * 1) mod 10 = 3
Collision occurred for 62 at index 3
h2(62) = 7 - (62 mod 7) = 1
Step 3: (h(2) + 2 * 1) mod 10 = 4
Collision occurred for 62 at index 4
h2(62) = 7 - (62 mod 7) = 1
Step 4: (h(2) + 3 * 1) mod 10 = 5
Collision occurred for 62 at index 5
h2(62) = 7 - (62 mod 7) = 1
Step 5: (h(2) + 4 * 1) mod 10 = 6
Collision occurred for 62 at index 6
h2(62) = 7 - (62 mod 7) = 1
Step 6: (h(2) + 5 * 1) mod 10 = 7
Collision occurred for 62 at index 7
h2(62) = 7 - (62 mod 7) = 1
Step 7: (h(2) + 6 * 1) mod 10 = 8
Collision occurred for 62 at index 8
h2(62) = 7 - (62 mod 7) = 1
Step 8: (h(2) + 7 * 1) mod 10 = 9
Collision occurred for 62 at index 9
h2(62) = 7 - (62 mod 7) = 1
Step 9: (h(2) + 8 * 1) mod 10 = 0
Collision occurred for 62 at index 0
h2(62) = 7 - (62 mod 7) = 1
Step 10: (h(2) + 9 * 1) mod 10 = 1
Inserted 62 at index 1
```

	Menu: 1. Insert elements into the hash table 2. Print the hash table 3. Exit Enter your choice: 2 Hash Table: Index 0: 50 Index 1: 62 Index 2: 12 Index 3: 60 Index 4: 80 Index 5: 32 Index 6: 52 Index 7: 70 Index 8: 22 Index 9: 42
CONCLUSION :-	After implementing hashing with linear probing, quadratic probing, and double hashing in C, I learned how each technique addresses collisions and affects performance. Linear probing is simple but can cause clustering; quadratic probing reduces clustering but may leave some spots unused; double hashing provides the best distribution with minimal clustering but requires an extra hash function. Overall, each method balances trade-offs between simplicity, speed, and even data distribution in hash tables.