**Exp No. 8**

## Title:

Searching and Hashing — Implement Hash Table using Array with Collision Handling via (A) Chaining and (B) Linear Probing

## Aim:

To implement a hash table using an array and handle collisions using (i) separate chaining with linked lists and (ii) open addressing via linear probing.

## Objectives:

• Understand hashing, load factor, and collision resolution strategies.  
• Implement insert, search, and delete operations for hash tables.  
• Compare memory and performance trade-offs between chaining and linear probing.

## Theory:

Hashing maps keys to indices of an array using a hash function h(key) = key % M (for integer keys). Collisions occur when multiple keys map to the same index. Two common strategies:  
1) Separate Chaining: Each table slot holds a linked list of entries. Insert at head/tail of list. Search/delete traverse the list.  
2) Linear Probing (Open Addressing): If slot is occupied, probe the next slot (i+1)%M repeatedly until an empty slot is found. Deletion uses a special marker (e.g., -2) to avoid breaking probe chains.  
  
Load factor α affects performance: higher α generally increases collisions and search costs, especially for open addressing.

## Algorithms (High Level):

A) Chaining (Separate Linked Lists)

• Insert(k): idx = h(k); push k to list at table[idx].  
• Search(k): traverse list at table[idx]; return found/not found.  
• Delete(k): find node in list at table[idx], unlink it.

B) Linear Probing (Open Addressing)

• Insert(k): idx = h(k); while slot is occupied (not -1 and not -2), idx = (idx+1)%M; place k.  
• Search(k): probe until empty slot (-1) encountered or key found.  
• Delete(k): upon finding k, mark slot as deleted (-2).

## Program Code A (C Language): Hash Table with Chaining

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define M 7 // Hash Table Size

// --- Linked List Node for Chaining ---

typedef struct Node {

    int key;

    struct Node \*next;

} Node;

// Hash Table is an array of Node pointers (heads of linked lists)

Node \*hash\_table[M];

// --- Hash Function ---

int hash(int key) {

    return key % M;

}

// --- Linked List Utility Functions ---

// Function to create a new node

Node\* create\_node(int key) {

    Node \*new\_node = (Node \*)malloc(sizeof(Node));

    if (new\_node == NULL) exit(EXIT\_FAILURE);

    new\_node->key = key;

    new\_node->next = NULL;

    return new\_node;

}

// --- Hash Table Operations (Chaining) ---

// Insert key into the hash table

void insert\_chaining(int key) {

    int index = hash(key);

    // Check for duplicates (optional, but good practice)

    Node \*current = hash\_table[index];

    while (current != NULL) {

        if (current->key == key) {

            printf("Key %d already exists.\n", key);

            return;

        }

        current = current->next;

    }

    // Insert at the head of the linked list

    Node \*new\_node = create\_node(key);

    new\_node->next = hash\_table[index];

    hash\_table[index] = new\_node;

    printf("Inserted %d at index %d (Chaining).\n", key, index);

}

// Search for a key in the hash table

bool search\_chaining(int key) {

    int index = hash(key);

    Node \*current = hash\_table[index];

    while (current != NULL) {

        if (current->key == key) {

            printf("Search %d: Found at index %d.\n", key, index);

            return true;

        }

        current = current->next;

    }

    printf("Search %d: Not Found.\n", key);

    return false;

}

// Delete a key from the hash table

void delete\_chaining(int key) {

    int index = hash(key);

    Node \*current = hash\_table[index];

    Node \*prev = NULL;

    while (current != NULL && current->key != key) {

        prev = current;

        current = current->next;

    }

    if (current == NULL) {

        printf("Delete %d: Not Found.\n", key);

        return;

    }

    // Found: unlink the node

    if (prev == NULL) {

        // Node to delete is the head

        hash\_table[index] = current->next;

    } else {

        // Node is in the middle or end

        prev->next = current->next;

    }

    printf("Delete %d: Removed from list at index %d.\n", key, index);

    free(current);

}

// Display the hash table

void display\_chaining() {

    printf("\n--- Hash Table (Chaining) ---\n");

    for (int i = 0; i < M; i++) {

        printf("Index %d: ", i);

        Node \*current = hash\_table[i];

        while (current != NULL) {

            printf("%d -> ", current->key);

            current = current->next;

        }

        printf("NULL\n");

    }

    printf("-----------------------------\n");

}

// Function to free the memory

void free\_chaining() {

    for (int i = 0; i < M; i++) {

        Node \*current = hash\_table[i];

        Node \*next;

        while (current != NULL) {

            next = current->next;

            free(current);

            current = next;

        }

        hash\_table[i] = NULL;

    }

}

int main() {

    // Initialize table to NULL

    for (int i = 0; i < M; i++) {

        hash\_table[i] = NULL;

    }

    printf("--- Hash Table with Chaining (M=%d) ---\n", M);

    // Insert: 10, 3, 17, 24 (All hash to index 3)

    insert\_chaining(10); // Index 3: 10 -> NULL

    insert\_chaining(3);  // Index 3: 3 -> 10 -> NULL

    insert\_chaining(17); // Index 3: 17 -> 3 -> 10 -> NULL

    insert\_chaining(24); // Index 3: 24 -> 17 -> 3 -> 10 -> NULL

    // Insert keys that map to other indices

    insert\_chaining(1); // Index 1

    insert\_chaining(8); // Index 1: 8 -> 1 -> NULL

    display\_chaining();

    search\_chaining(17); // Found

    search\_chaining(15); // Not Found

    delete\_chaining(3); // Deleted from index 3

    delete\_chaining(99); // Not Found

    display\_chaining();

    // Clean up memory

    free\_chaining();

    return 0;

}

## Program Code B (C Language): Hash Table with Linear Probing

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define M 7 // Hash Table Size

#define EMPTY -1     // Marker for an empty slot

#define DELETED -2   // Marker for a deleted slot (tombstone)

int hash\_table[M];

// --- Hash Function ---

int hash(int key) {

    return key % M;

}

// --- Hash Table Operations (Linear Probing) ---

// Find the index of a key (used by all other functions)

int find\_slot(int key, bool stop\_at\_empty, int \*probes) {

    int start\_index = hash(key);

    int current\_index = start\_index;

    \*probes = 0;

    for (int i = 0; i < M; i++) {

        (\*probes)++;

        if (hash\_table[current\_index] == EMPTY) {

            if (stop\_at\_empty) return current\_index; // Found an empty spot for insert/unsuccessful search

            // For successful search/delete, keep probing past EMPTY if not specified to stop

        }

        if (hash\_table[current\_index] == key) {

            return current\_index; // Found the key

        }

        // Move to the next slot (linear probing)

        current\_index = (start\_index + i + 1) % M; // Using i for probe step

    }

    return -1; // Not found/Table full/No empty/deleted slot

}

// Insert key into the hash table

bool insert\_probing(int key) {

    int probes;

    // Find a slot: stop\_at\_empty=true (look for EMPTY or DELETED)

    int index = -1;

    // Instead of using find\_slot, let's implement the search for an insert spot

    // which should ideally reuse DELETED slots

    int start\_index = hash(key);

    int current\_index = start\_index;

    int first\_deleted\_index = -1;

    for (int i = 0; i < M; i++) {

        if (hash\_table[current\_index] == EMPTY) {

            index = (first\_deleted\_index != -1) ? first\_deleted\_index : current\_index;

            break;

        } else if (hash\_table[current\_index] == DELETED) {

            if (first\_deleted\_index == -1) {

                first\_deleted\_index = current\_index;

            }

        } else if (hash\_table[current\_index] == key) {

            printf("Key %d already exists.\n", key);

            return false;

        }

        current\_index = (start\_index + i + 1) % M;

    }

    if (index == -1) {

        // If we found a DELETED slot, use it

        index = first\_deleted\_index;

    }

    if (index != -1) {

        hash\_table[index] = key;

        printf("Inserted %d at index %d (Probes: %d).\n", key, index, abs(index - start\_index) + 1); // Simple probe count

        return true;

    } else {

        printf("Overflow! Hash table is full.\n");

        return false;

    }

}

// Search for a key in the hash table

bool search\_probing(int key, int \*probes) {

    int start\_index = hash(key);

    int current\_index = start\_index;

    \*probes = 0;

    for (int i = 0; i < M; i++) {

        (\*probes)++;

        if (hash\_table[current\_index] == EMPTY) {

            printf("Search %d: Not Found (Probed %d slots).\n", key, \*probes);

            return false; // Stop search at EMPTY slot

        }

        if (hash\_table[current\_index] == key) {

            printf("Search %d: Found at index %d in %d probes.\n", key, current\_index, \*probes);

            return true; // Found the key

        }

        // Continue probing past DELETED

        current\_index = (start\_index + i + 1) % M;

    }

    printf("Search %d: Not Found (Probed %d slots - cycle completed).\n", key, \*probes);

    return false;

}

// Delete a key from the hash table

void delete\_probing(int key) {

    int probes;

    int index = -1;

    int start\_index = hash(key);

    int current\_index = start\_index;

    for (int i = 0; i < M; i++) {

        if (hash\_table[current\_index] == EMPTY) {

            break; // Stop at EMPTY

        }

        if (hash\_table[current\_index] == key) {

            index = current\_index;

            break;

        }

        current\_index = (start\_index + i + 1) % M;

    }

    if (index != -1) {

        hash\_table[index] = DELETED;

        printf("Delete %d: Marked DELETED at index %d.\n", key, index);

    } else {

        printf("Delete %d: Not Found.\n", key);

    }

}

// Display the hash table

void display\_probing() {

    printf("\n--- Hash Table (Linear Probing) ---\n");

    for (int i = 0; i < M; i++) {

        printf("Index %d: ", i);

        if (hash\_table[i] == EMPTY) {

            printf("EMPTY\n");

        } else if (hash\_table[i] == DELETED) {

            printf("DELETED\n");

        } else {

            printf("%d\n", hash\_table[i]);

        }

    }

    printf("-----------------------------------\n");

}

int main() {

    // Initialize table to EMPTY

    for (int i = 0; i < M; i++) {

        hash\_table[i] = EMPTY;

    }

    printf("--- Hash Table with Linear Probing (M=%d) ---\n", M);

    insert\_probing(10);

    insert\_probing(3);

    insert\_probing(17);

    insert\_probing(24);

    insert\_probing(31);

    insert\_probing(8);

    insert\_probing(15);

    display\_probing();

    int probes\_count;

    search\_probing(24, &probes\_count);

    search\_probing(17, &probes\_count);

    search\_probing(99, &probes\_count); // Not Found

    delete\_probing(17); // Mark DELETED at index 5

    display\_probing();

    search\_probing(24, &probes\_count); // Must probe past 5 (DELETED)

    return 0;

}

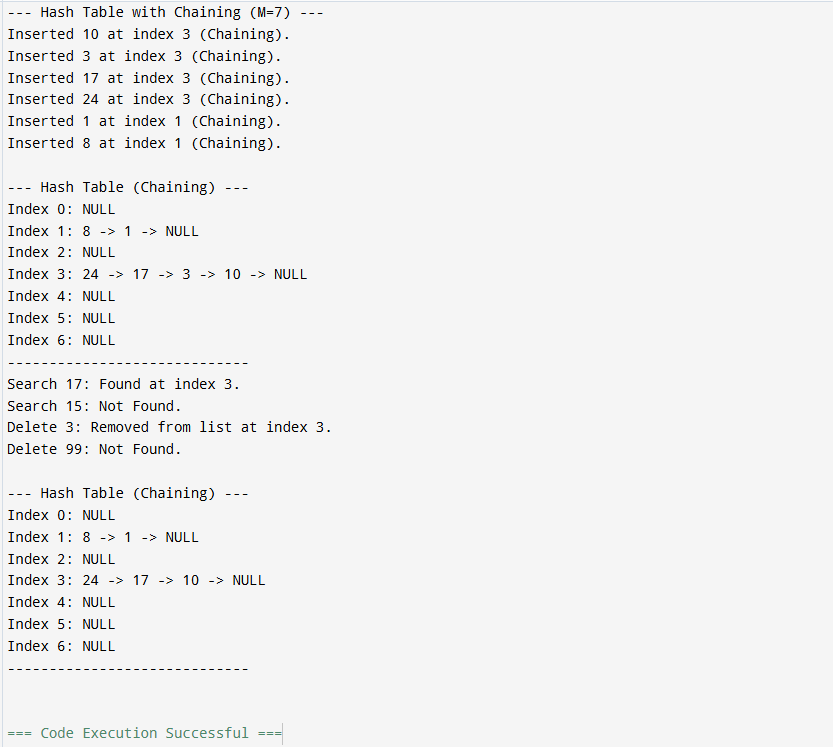
## Sample Input/Output:

Chaining Sample:  
Insert: 10, 3, 17, 24 (M=7) → indices: 3,3,3,3 (list grows)  
Search 17 → Found  
Delete 3 → Removed from list at index 3  
  
Linear Probing Sample:  
Insert: 10, 3, 17, 24 (M=7) → indices: 3,3,3,3 but stored at 3,4,5,6 via probing  
Search 24 → Found at index 6  
Delete 17 → Marked DELETED at its index

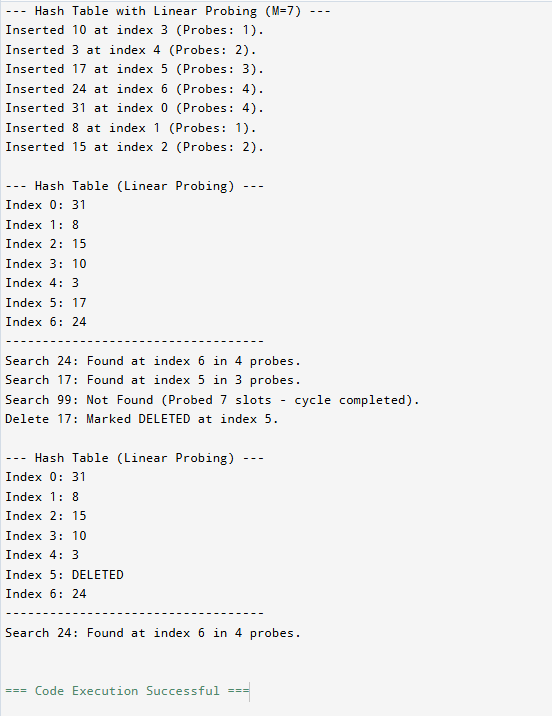
## Result:

Implemented hash tables with both chaining and linear probing collision resolution. Demonstrated insert, search, delete, and display operations.

Program A Output:



Program B Output:



## Conclusion:

Chaining uses extra memory for pointers but maintains expected O(1) average operations at moderate load. Linear probing is memory-compact, but clustering increases probe lengths as load factor grows; careful table sizing and rehashing are recommended in practice.

## Post-Lab Problem : Count Probes on Successful/Unsuccessful Search (Linear Probing)

Objective:

Modify the linear probing program to count and print the number of slots inspected during search (probes) for both successful and unsuccessful searches.

Description:

Add a counter inside the search routine to tally how many positions are checked before returning. Run multiple searches and compute the average probes for a given load factor.

Input:

Sequence of inserts followed by multiple search keys.

Output:

For each search, print "Found at index i in P probes" or "Not Found in P probes".

Constraints:

• Use table size M=7 or 11. • Keys are integers. • Stop search when EMPTY encountered or full cycle completed.

Sample Output:

Insert: 10 3 17 24 31  
Search 24 → Found at index 6 in 4 probes  
Search 99 → Not Found in 5 probes

Hints:

• For unsuccessful search, terminate when you hit an EMPTY slot or wrap to start.  
• Keep DELETED slots probing to avoid breaking chains.

Post Lab Program Code:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define M 7 // Hash Table Size (M=7 used)

#define EMPTY -1     // Marker for an empty slot

#define DELETED -2   // Marker for a deleted slot (tombstone)

int hash\_table[M];

// --- Hash Function (Reused) ---

int hash(int key) {

    return key % M;

}

// --- Insert Function (Reused from B) ---

bool insert\_probing(int key) {

    int start\_index = hash(key);

    int current\_index = start\_index;

    int first\_deleted\_index = -1;

    for (int i = 0; i < M; i++) {

        if (hash\_table[current\_index] == EMPTY) {

            int index = (first\_deleted\_index != -1) ? first\_deleted\_index : current\_index;

            hash\_table[index] = key;

            return true;

        } else if (hash\_table[current\_index] == DELETED) {

            if (first\_deleted\_index == -1) first\_deleted\_index = current\_index;

        } else if (hash\_table[current\_index] == key) {

            return false; // Key already exists

        }

        current\_index = (start\_index + i + 1) % M;

    }

    if (first\_deleted\_index != -1) {

        hash\_table[first\_deleted\_index] = key;

        return true;

    }

    return false; // Table full

}

// --- Search Function with Probe Count (Core Post-Lab) ---

bool search\_probing(int key, int \*probes) {

    int start\_index = hash(key);

    int current\_index = start\_index;

    \*probes = 0;

    for (int i = 0; i < M; i++) {

        (\*probes)++;

        if (hash\_table[current\_index] == EMPTY) {

            printf("Search %d -> Not Found in %d probes\n", key, \*probes);

            return false; // Stop search at EMPTY slot

        }

        if (hash\_table[current\_index] == key) {

            printf("Search %d -> Found at index %d in %d probes\n", key, current\_index, \*probes);

            return true; // Successful search

        }

        // Continue probing past DELETED

        current\_index = (start\_index + i + 1) % M;

    }

    printf("Search %d -> Not Found in %d probes (full cycle)\n", key, \*probes);

    return false;

}

// --- Display Function (Reused) ---

void display\_probing() {

    printf("\n--- Hash Table State (Linear Probing) ---\n");

    for (int i = 0; i < M; i++) {

        printf("Index %d: ", i);

        if (hash\_table[i] == EMPTY) {

            printf("EMPTY\n");

        } else if (hash\_table[i] == DELETED) {

            printf("DELETED\n");

        } else {

            printf("%d\n", hash\_table[i]);

        }

    }

    printf("-------------------------------------------\n");

}

int main() {

    // Initialize table to EMPTY

    for (int i = 0; i < M; i++) {

        hash\_table[i] = EMPTY;

    }

    printf("--- Post-Lab: Probe Counting in Linear Probing (M=%d) ---\n", M);

    // Insert Keys (as in sample I/O): 10, 3, 17, 24, 31 (All hash to 3)

    // 10(3) -> Index 3

    // 3(3) -> Index 4

    // 17(3) -> Index 5

    // 24(3) -> Index 6

    // 31(3) -> Index 0 (wraps around)

    insert\_probing(10);

    insert\_probing(3);

    insert\_probing(17);

    insert\_probing(24);

    insert\_probing(31);

    display\_probing();

    int probes\_count;

    // Successful Search 1 (Key 24: Hash 3, stored at 6)

    // Probes: 3 -> 4 -> 5 -> 6 (4 probes)

    search\_probing(24, &probes\_count);

    // Successful Search 2 (Key 31: Hash 3, stored at 0)

    // Probes: 3 -> 4 -> 5 -> 6 -> 0 (5 probes)

    search\_probing(31, &probes\_count);

    // Unsuccessful Search 1 (Key 99: Hash 1, but index 1 is EMPTY)

    // Probes: 1 (1 probe)

    search\_probing(99, &probes\_count);

    // Unsuccessful Search 2 (Key 4: Hash 4, but index 4 is occupied by 3)

    // Probes: 4 -> 5 -> 6 -> 0 -> 1 -> 2 (Index 2 is EMPTY) (6 probes)

    search\_probing(4, &probes\_count);

    return 0;

}

Post Lab Program Output:

