# ADS LAB ASSINGMENT 9

**Name-Sandesh Santosh Kadam**

**Class- SY IT-C Batch 1**

**PRN-12210695**

**Roll no-26**

Q. **Write C/C++ program to implement hashing using chaining**

# Theory-

Chaining involves using a hash function to map keys to indices in an array. When two or more keys hash to the same index, a collision occurs. Chaining is one way to handle collisions by maintaining a linked list at each array index. Each linked list contains all the keys that hash to the same index.

Here's a breakdown of key concepts:

Hash Function:

A hash function takes a key as input and produces an index as output. The goal is to distribute the keys uniformly across the array to minimize collisions.

Hash Table:

A hash table is an array that stores keys and associated values. The array size is typically determined by the number of possible hash indices.

Collision:

A collision occurs when two or more keys hash to the same index. Collisions are inevitable, and handling them is a critical aspect of designing a good hash table.

Chaining:

Chaining is a collision resolution technique where each array index maintains a linked list of keys that hash to the same index. If a collision occurs, the new key is simply appended to the linked list at the corresponding index.

Linked List:

In the context of hashing with chaining, a linked list is used to manage keys that collide. Each node in the list contains a key and a reference to the next node in the list.

Insertion:

To insert a key into the hash table, the hash function is applied to determine the index. The key is then inserted at the end of the linked list at that index.

Search:

To search for a key, the hash function is used to find the index. The linked list at that index is then traversed to check if the key is present.

Deletion:

To delete a key, the hash function is used to find the index, and the key is removed from the linked list at that index.

Load Factor:

The load factor is the ratio of the number of keys to the size of the array. A low load factor indicates a sparse array, while a high load factor may lead to increased collisions. It's important to resize the array if the load factor becomes too high.

Hashing with chaining provides a simple and effective way to handle collisions. The time complexity for insertion, search, and deletion operations is generally O(1) on average, assuming a good hash function and a reasonably balanced distribution of keys.

**Programming Lang. Used- C**

**Compiler Used- VS Code**

# CODE HASHING:-

#include <stdio.h>

#include <stdlib.h>

#define SIZE 10

struct Node {

    int data;

    struct Node\* next;

};

struct HashTable {

    struct Node\* table[SIZE];

};

int hashFunction(int key) {

    return key % SIZE;

}

void insert(struct HashTable\* ht, int key) {

    int index = hashFunction(key);

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = key;

    newNode->next = NULL;

    if (ht->table[index] == NULL) {

        ht->table[index] = newNode;

    } else {

        struct Node\* current = ht->table[index];

        while (current->next != NULL) {

            current = current->next;

        }

        current->next = newNode;

    }

}

void display(struct HashTable\* ht) {

    printf("Hash Table:\n");

    for (int i = 0; i < SIZE; i++) {

        printf("%d:", i);

        struct Node\* current = ht->table[i];

        while (current != NULL) {

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

            current = current->next;

        }

        printf("\n");

    }

}

int main() {

    struct HashTable ht;

    for (int i = 0; i < SIZE; i++) {

        ht.table[i] = NULL;

    }

    int n;

    printf("Enter the number of elements to insert: ");

    scanf("%d", &n);

    printf("Enter the elements:\n");

    for (int i = 0; i < n; i++) {

        int key;

        scanf("%d", &key);

        insert(&ht, key);

    }

    display(&ht);

    return 0;

}

# OUTPUT-

