Implement following hashing Techniques by assuming suitable input and Table Size.

1. Linear Probing

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define TABLE\_SIZE 10

struct Employee {

char employee\_name[50];

int emp\_no;

float emp\_salary;

};

struct Employee hashTable[TABLE\_SIZE];

// Hash function to generate an index

int hash(int key) {

return key % TABLE\_SIZE;

}

// Function to insert an employee into the hash table

void insert(struct Employee emp) {

int index = hash(emp.emp\_no);

int originalIndex = index;

// Linear probing to find an empty slot

while (hashTable[index].emp\_no != -1) {

index = (index + 1) % TABLE\_SIZE;

// If we come back to the original index, the table is full

if (index == originalIndex) {

printf("Hash table is full. Cannot insert.\n");

return;

}

}

hashTable[index] = emp;

printf("Inserted %s at index %d\n", emp.employee\_name, index);

}

// Function to display the hash table

void displayHashTable() {

printf("\nHash Table:\n");

printf("Index\tEmployee Name\tEmployee No.\tEmployee Salary\n");

for (int i = 0; i < TABLE\_SIZE; i++) {

printf("%d\t", i);

if (hashTable[i].emp\_no == -1) {

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

} else {

printf("%s\t\t%d\t\t%.2f\n", hashTable[i].employee\_name, hashTable[i].emp\_no, hashTable[i].emp\_salary);

}

}

}

int main() {

// Initialize the hash table with empty values

for (int i = 0; i < TABLE\_SIZE; i++) {

hashTable[i].emp\_no = -1;

}

// Insert some employees into the hash table

struct Employee emp1 = {"Alice", 101, 50000.0};

struct Employee emp2 = {"Bob", 201, 60000.0};

struct Employee emp3 = {"Charlie", 301, 70000.0};

insert(emp1);

insert(emp2);

insert(emp3);

// Display the hash table

displayHashTable();

return 0;

}