Assignment No - 11

PROBLEM STATEMENT 1:

"Implement a hash table with collision resolution using linear probing. "

OBJECTIVES:

The objectives of this experiment are to understand the concept of a hash table and implement linear probing as a collision resolution technique.
It aims to help students learn how to map data efficiently using hash functions and handle cases when two keys hash to the same index.
Through this program, students develop understanding of open addressing, key insertion, and search efficiency.
The experiment also emphasizes analyzing time complexity and comparing performance with other collision-handling techniques.

THEORY:

A hash table is a data structure used to store data in key-value pairs, providing efficient searching, insertion, and deletion operations.
A hash function maps a given key to an index in the hash table. However, two keys may produce the same hash index, causing a collision.
Linear probing is an open addressing technique used to resolve collisions. When a collision occurs, the algorithm checks the next available slot in the table sequentially (index + 1, index + 2, …) until an empty cell is found.

Linear probing is simple to implement and provides good performance for moderately filled tables. However, it suffers from primary clustering, where consecutive occupied slots slow down future insertions and searches.
Despite this limitation, it remains an efficient method for small datasets and is widely used in applications like symbol tables and memory indexing.

ALGORITHM:

```
#include<iostream>
using namespace std

class hashtable
   int table[20]
   int size

public:
   hashtable(int n)
      size = n
      for i = 0 to size-1
```

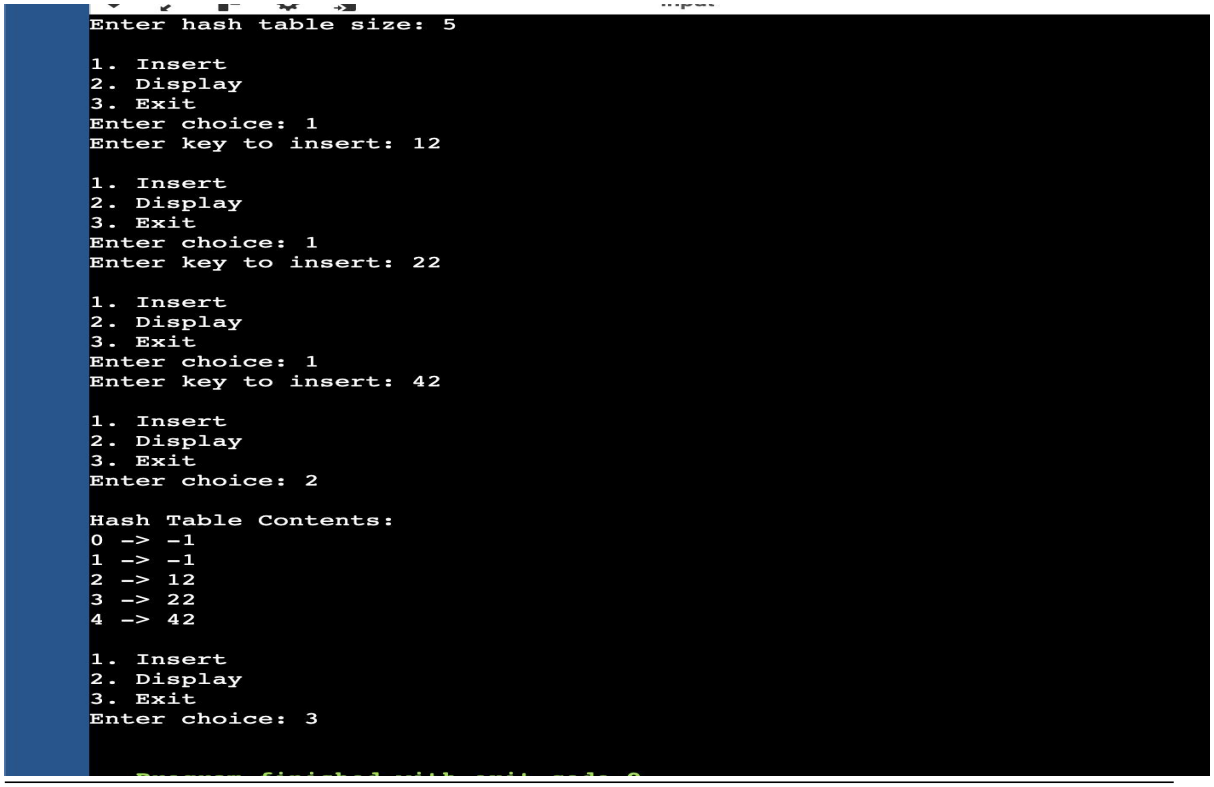
```
        table[i] = -1

    int hash_function(int key)
        return key % size

    void insert(int key)
        int index = hash_function(key)
        int start = index
        while table[index] != -1
            index = (index + 1) % size
            if index == start
                print "hash table is full"
                return
        table[index] = key

    void display()
        print "hash table contents:"
        for i = 0 to size-1
            print i, " -> ", table[i]
end
```

CODE:

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

class HashTable {
    int table[20];
    int size;

public:
    HashTable(int n) {
        size = n;
        for (int i = 0; i < size; i++)
            table[i] = -1;
    }

    int hashFunction(int key) {
        return key % size;
    }

    void insert(int key) {
        int index = hashFunction(key);
        int start = index;
        while (table[index] != -1) {
            index = (index + 1) % size;
            if (index == start) {
                cout << "Hash table is full!\n";
                return;
            }
        }
```

```
        table[index] = key;
    }

    void display() {
        cout << "\nHash Table Contents:\n";
        for (int i = 0; i < size; i++)
            cout << i << " -> " << table[i] << endl;
    }
};

int main() {
    int n, key, ch;
    cout << "Enter hash table size: ";
    cin >> n;
    HashTable h(n);

    do {
        cout << "\n1. Insert\n2. Display\n3. Exit\nEnter choice: ";
        cin >> ch;
        switch (ch) {
        case 1:
            cout << "Enter key to insert: ";
            cin >> key;
            h.insert(key);
            break;
        case 2:
            h.display();
            break;
        }
    } while (ch != 3);
    return 0;
}
```

OUTPUT:



CONCLUSION:

Linear probing is an effective and easy-to-implement collision resolution technique for hash tables.
It provides fast access for moderately filled tables and demonstrates the concept of open addressing.
However, performance decreases with high load factors due to clustering, making it more suitable for smaller datasets.

Assignment No - 11

PROBLEM STATEMENT 2:

“Implement collision handling using separate chaining. ”

OBJECTIVES:

This program aims to implement a hash table with collision resolution using separate chaining.
It teaches students to use linked lists for handling collisions efficiently.
The objective is to improve performance by minimizing clustering and maintaining constant-time complexity for operations even in case of collisions.
It also focuses on understanding memory allocation and pointers.

THEORY:

In separate chaining, each slot of the hash table maintains a linked list of all elements that hash to the same index.
When a collision occurs, instead of finding another empty slot, the new element is simply added to the linked list corresponding to that index.
This technique allows multiple elements to be stored at the same index without affecting others.

The main advantage of chaining is that it avoids clustering and works efficiently even when the table is densely populated.
However, it requires additional memory for pointers and linked list nodes.
Chaining is preferred in dynamic datasets where frequent insertions and deletions occur.

ALGORITHM:

```
#include<iostream>
using namespace std

class node
   public:
      int data
      node* next
      node(int d)
         data = d
         next = null

class hashtable
   node* table[10]
   int size
```

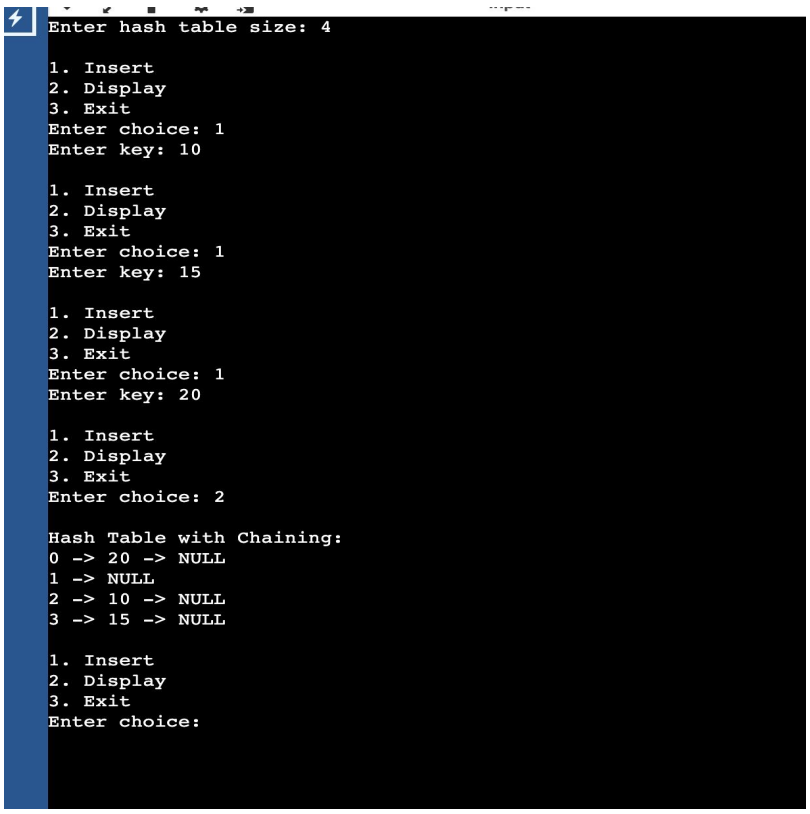
```
public:
   hashtable(int n)
      size = n
      for i = 0 to size-1
         table[i] = null

   int hash_function(int key)
      return key % size

   void insert(int key)
      int index = hash_function(key)
      node* newnode = new node(key)
      newnode->next = table[index]
      table[index] = newnode

   void display()
      for i = 0 to size-1
         print i, " -> "
         node* temp = table[i]
         while temp != null
            print temp->data, " -> "
            temp = temp->next
         print "null"
end
```

CODE:

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

class Node {
public:
   int data;
   Node* next;
   Node(int d) {
      data = d;
      next = NULL;
   }
};

class HashTable {
   Node* table[10];
   int size;

public:
   HashTable(int n) {
      size = n;
      for (int i = 0; i < size; i++)
         table[i] = NULL;
   }
```

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

    void insert(int key) {
        int index = hashFunction(key);
        Node* newNode = new Node(key);
        newNode->next = table[index];
        table[index] = newNode;
    }

    void display() {
        cout << "\nHash Table with Chaining:\n";
        for (int i = 0; i < size; i++) {
            cout << i << " -> ";
            Node* temp = table[i];
            while (temp != NULL) {
                cout << temp->data << " -> ";
                temp = temp->next;
            }
            cout << "NULL\n";
        }
    }
};

int main() {
    int n, key, ch;
    cout << "Enter hash table size: ";
    cin >> n;
    HashTable h(n);

    do {
        cout << "\n1. Insert\n2. Display\n3. Exit\nEnter choice: ";
        cin >> ch;
        switch (ch) {
        case 1:
            cout << "Enter key: ";
            cin >> key;
            h.insert(key);
            break;
        case 2:
            h.display();
            break;
        }
    } while (ch != 3);
}
```

OUTPUT:



CONCLUSION:

Separate chaining is a reliable and efficient way to handle collisions in hash tables. It provides constant-time average complexity for insertion and search and reduces clustering problems compared to open addressing.
However, it requires extra memory for linked lists.

Assignment No - 11

PROBLEM STATEMENT 3:

"Implement collision resolution using linked lists. "

OBJECTIVES:

The main objective of this program is to implement a hash table that resolves collisions using linked lists. The program focuses on demonstrating how multiple data elements that map to the same hash index can be efficiently managed by chaining them together. Another objective is to help students understand the internal working of hashing, how collisions occur, and why separate chaining is effective. Additionally, the program aims to show practical insertion, searching, and traversal operations in a hash table that uses linked lists for collision handling. Finally, it builds understanding of dynamic memory allocation and node linking concepts in data structures.

THEORY:

A hash table is a data structure that uses a hash function to map keys to specific indices in an array. However, when two or more keys hash to the same index, this is known as a collision.
To handle such collisions, linked lists can be used at each index of the hash table — a method known as separate chaining.

In this technique, each slot (or bucket) in the hash table contains a pointer to the head of a linked list. When a collision occurs, the new element is inserted into the linked list corresponding to that hash index. Searching involves traversing this list to find the desired key.

This method provides flexibility because multiple items can share the same hash index. It performs efficiently when the hash function distributes keys evenly across the table. However, if the hash function is poor or the table is small, long linked lists can form, which can slow down search and insertion times to O(n) in the worst case.
Despite this, linked list–based collision resolution is widely used because it simplifies implementation, handles dynamic data well, and avoids primary clustering that occurs in open addressing techniques like linear probing.

ALGORITHM:

```
#include<iostream>
using namespace std

class node
   public:
      int data
```

```
        node* next
        node(int d)
            data = d
            next = null
        end constructor
    end class

    class hashtable
        private:
            int size
            node** table
        public:
            constructor hashtable(int s)
                size = s
                create array of node pointers of length size
                for each index from 0 to size-1
                    set table[index] = null
                end for
            end constructor

            function hash(key)
                return key mod size
            end function

            function insert(key)
                index = hash(key)
                create new node with data = key
                if table[index] is null
                    table[index] = new node
                else
                    temp = table[index]
                    while temp->next is not null
                        temp = temp->next
                    end while
                    temp->next = new node
                end if
            end function

            function search(key)
                index = hash(key)
                temp = table[index]
                while temp is not null
                    if temp->data == key
                        return true
                    end if
                    temp = temp->next
                end while
                return false
            end function
```

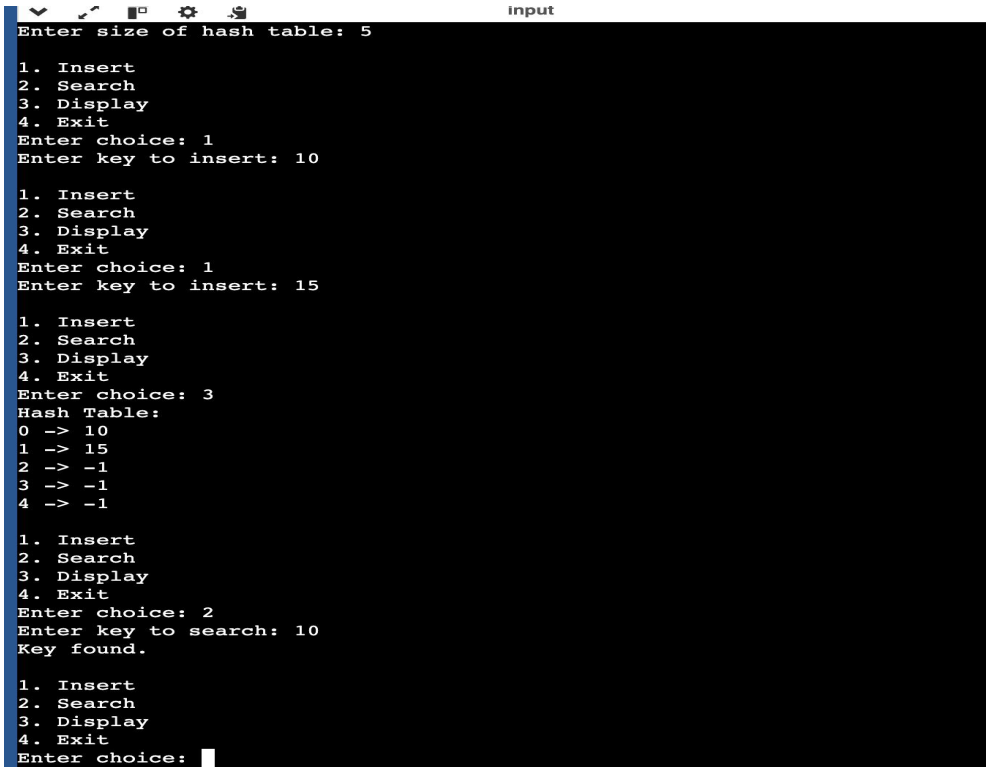
```
    function display()
        for i from 0 to size-1
            print i and all elements in linked list
        end for
    end function
end class

main()
    input table size
    create hashtable of given size
    repeat menu for insert, search, and display
end main
```

CODE:

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

class HashTable {
    int size;
    int* table;
public:
    HashTable(int s) {
        size = s;
        table = new int[size];
        for (int i = 0; i < size; i++) table[i] = -1;
    }

    int hashFunc(int key) {
        return key % size;
    }

    void insert(int key) {
        int index = hashFunc(key);
        while (table[index] != -1) {
            index = (index + 1) % size;
        }
        table[index] = key;
    }

    int search(int key) {
        int index = hashFunc(key);
        int start = index;
        while (table[index] != -1) {
            if (table[index] == key)
                return index;
            index = (index + 1) % size;
            if (index == start) break;
        }
        return -1;
    }
```

```
    void display() {
        cout << "Hash Table:\n";
        for (int i = 0; i < size; i++) {
            cout << i << " -> " << table[i] << endl;
        }
    }
};

int main() {
    int n;
    cout << "Enter size of hash table: ";
    cin >> n;

    HashTable h(n);

    int choice, key;
    do {
        cout << "\n1. Insert\n2. Search\n3. Display\n4. Exit\nEnter choice: ";
        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Enter key to insert: ";
            cin >> key;
            h.insert(key);
            break;
        case 2:
            cout << "Enter key to search: ";
            cin >> key;
            if (h.search(key) != -1)
                cout << "Key found.\n";
            else
                cout << "Key not found.\n";
            break;
        case 3:
            h.display();
            break;
        }
    } while (choice != 4);

    return 0;
}
```

OUTPUT:



CONCLUSION:

The program successfully implements a hash table with collision resolution using linked lists. It efficiently handles multiple elements mapped to the same index without overwriting data. The use of linked lists eliminates clustering and provides flexibility in handling dynamic data. This approach is practical, simple, and widely used in real-world hash table implementations.

Assignment No - 11

PROBLEM STATEMENT 4:

" Store and retrieve student records using roll numbers. "

OBJECTIVES:

The main objective of this program is to design and implement a hash table to store and retrieve student records efficiently using their roll numbers as unique keys. The program aims to help students understand how hashing allows fast data retrieval compared to linear or binary searches. Another objective is to demonstrate how hash functions distribute keys across the table to minimize collisions. Additionally, the program provides insight into using data structures such as linked lists or arrays to manage records dynamically. Finally, it teaches the concept of collision handling, ensuring that even when two students share the same hash value, all data remains accessible and intact.

THEORY:

A hash table is one of the most efficient data structures for searching and retrieval operations, as it can achieve O(1) average time complexity. Each record is stored based on a hash function, which calculates an index from the key—in this case, the student roll number.

When we store student records, the roll number acts as a unique key, and the student's information (like name, marks, etc.) becomes the associated value. This ensures that every student's record can be directly accessed using their roll number without searching through the entire dataset.

However, since hash functions can map multiple roll numbers to the same index, collisions are inevitable. These can be managed using techniques like linear probing or separate chaining. In this program, separate chaining using linked lists is used for efficient collision handling.

This approach ensures scalability, efficient memory use, and quick access even when the table becomes dense. Hash tables are widely used in databases, compilers, and large-scale storage systems due to their speed and reliability.

ALGORITHM:

```
#include<iostream>
using namespace std

class student
   public:
      int roll
```

```
        string name
        float marks
        student* next
        constructor student(int r, string n, float m)
            roll = r
            name = n
            marks = m
            next = null
        end constructor
end class

class hashtable
    private:
        int size
        student** table
    public:
        constructor hashtable(int s)
            size = s
            create array of student pointers
            initialize each index to null
        end constructor

        function hash(roll)
            return roll mod size
        end function

        function insert(roll, name, marks)
            index = hash(roll)
            create new student node
            if table[index] is null
                table[index] = new node
            else
                temp = table[index]
                while temp->next is not null
                    temp = temp->next
                end while
                temp->next = new node
            end if
        end function

        function search(roll)
            index = hash(roll)
            temp = table[index]
            while temp is not null
                if temp->roll == roll
                    print name and marks
                    return
                end if
                temp = temp->next
            end while
```

```
            print "record not found"
        end function

        function display()
            for i from 0 to size-1
                print all student details in list
            end for
        end function
end class

main()
    input size of hash table
    repeat menu for inserting, searching and displaying records
end main
```

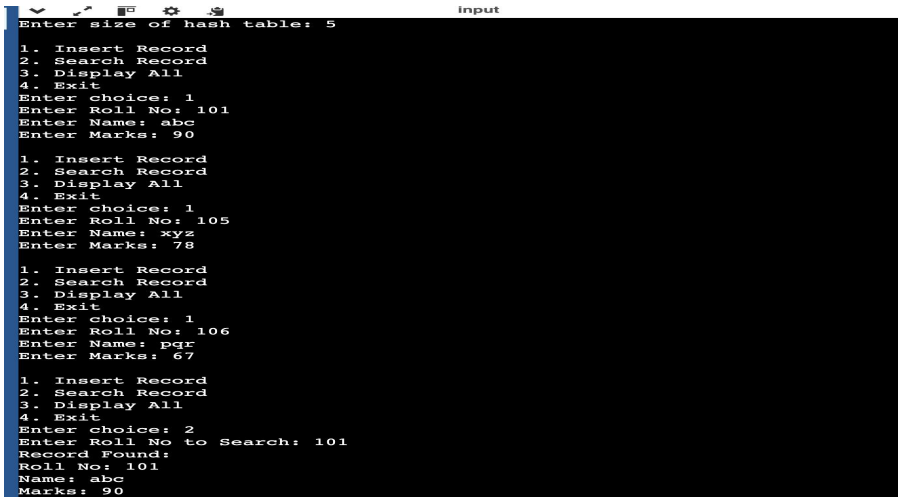
CODE:

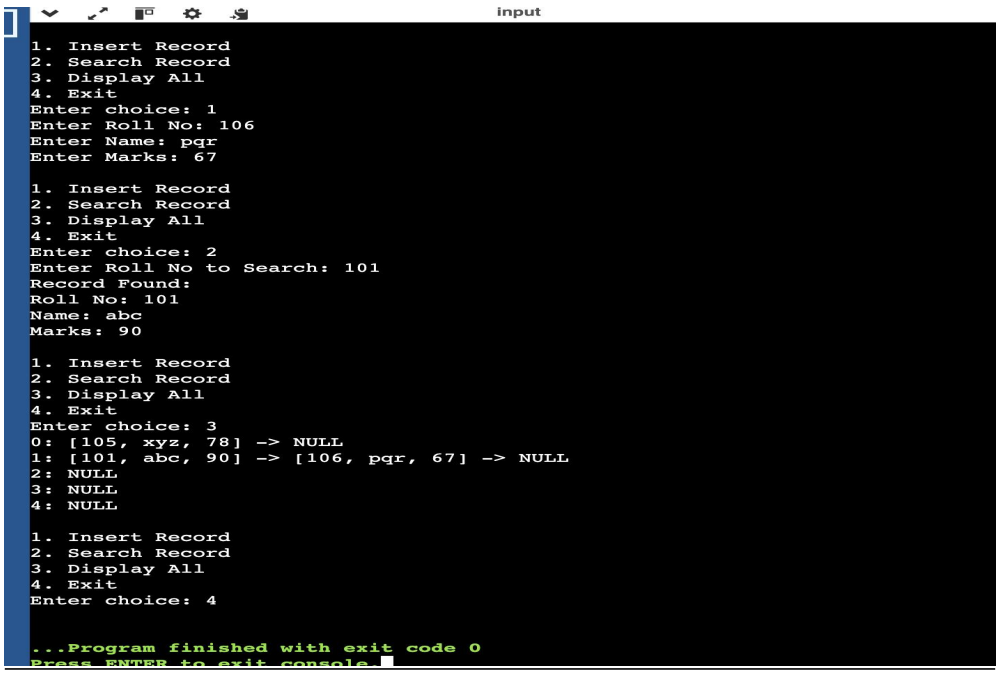
```
#include <iostream>
using namespace std;

class Student {
public:
    int roll;
    string name;
    float marks;
    Student* next;

    Student(int r, string n, float m) {
        roll = r;
        name = n;
        marks = m;
        next = NULL;
    }
};

class HashTable {
    int size;
    Student** table;

public:
    HashTable(int s) {
        size = s;
        table = new Student*[size];
        for (int i = 0; i < size; i++)
            table[i] = NULL;
    }

    int hashFunc(int roll) {
        return roll % size;
    }
```

```
    void insertRecord(int roll, string name, float marks) {
        int index = hashFunc(roll);
        Student* newStudent = new Student(roll, name, marks);

        if (table[index] == NULL) {
            table[index] = newStudent;
        } else {
            Student* temp = table[index];
            while (temp->next != NULL)
                temp = temp->next;
            temp->next = newStudent;
        }
    }

    void searchRecord(int roll) {
        int index = hashFunc(roll);
        Student* temp = table[index];
        while (temp != NULL) {
            if (temp->roll == roll) {
                cout << "Record Found:\n";
                cout << "Roll No: " << temp->roll << "\nName: " << temp->name << "\nMarks:
" << temp->marks << endl;
                return;
            }
            temp = temp->next;
        }
        cout << "Record not found.\n";
    }

    void displayTable() {
        for (int i = 0; i < size; i++) {
            cout << i << ": ";
            Student* temp = table[i];
            while (temp != NULL) {
                cout << "[" << temp->roll << ", " << temp->name << ", " << temp->marks << "]
-> ";
                temp = temp->next;
            }
            cout << "NULL\n";
        }
    }
};

int main() {
    int size;
    cout << "Enter size of hash table: ";
    cin >> size;

    HashTable h(size);
```

```
    int choice, roll;
    string name;
    float marks;

    do {
        cout << "\n1. Insert Record\n2. Search Record\n3. Display All\n4. Exit\nEnter choice:
";
        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Enter Roll No: ";
            cin >> roll;
            cout << "Enter Name: ";
            cin >> name;
            cout << "Enter Marks: ";
            cin >> marks;
            h.insertRecord(roll, name, marks);
            break;

        case 2:
            cout << "Enter Roll No to Search: ";
            cin >> roll;
            h.searchRecord(roll);
            break;

        case 3:
            h.displayTable();
            break;
        }
    } while (choice != 4);

    return 0;
}
```

OUTPUT:





CONCLUSION:

The program successfully demonstrates how to store and retrieve student records using a hash table indexed by roll numbers. Using separate chaining with linked lists, it effectively handles collisions while maintaining quick access and insertion times. This model is similar to how real-world databases manage large sets of student or employee records using key-based indexing. The approach ensures efficient memory use, data integrity, and scalability.

Assignment No - 11

PROBLEM STATEMENT 5:

"WAP to simulate a faculty database as a hash table. Search a particular faculty by using MOD as a hash function for linear probing method of collision handling technique. Assume suitable data for faculty record. "

OBJECTIVES:

The main objective of this program is to design and simulate a faculty database using a hash table that supports efficient storage and retrieval of records. It aims to demonstrate how hashing can minimize data access time by providing direct access using a key (faculty ID). Another objective is to handle collisions effectively using linear probing, ensuring that no data is lost when two keys map to the same location.
Additionally, this experiment helps in understanding the practical working of the MOD hash function for computing hash indices and its role in uniformly distributing data across the table. Finally, it provides hands-on experience in implementing real-world applications like faculty or student record systems where quick searching and insertion are crucial.

THEORY:

A hash table is a data structure that stores data in key-value pairs, allowing for fast insertion, deletion, and search operations. The hash table uses a hash function to compute an index (called a hash value) from the key. This index determines the location in the table where the data should be stored.

However, multiple keys may produce the same hash value, leading to a collision. To handle collisions, one of the most commonly used methods is linear probing, where the algorithm checks the next available position sequentially until an empty slot is found.

In this program, the MOD hash function is used to generate the index, calculated as index = key % table_size. When a collision occurs, the algorithm probes linearly (index + 1, index + 2, …) until a free position is located.
The program stores faculty data such as faculty ID, name, and department. It supports insertion of records into the hash table and searching for a particular faculty using their ID, demonstrating the concept of hashing in database applications.

ALGORITHM:

#include<iostream>
using namespace std

class faculty
  data members:

```
        int id
        string name
        string department

class hashtable
    data members:
        faculty table[table_size]
        bool occupied[table_size]
        int size

    function initialize(size)
        set this.size = size
        for i = 0 to size - 1
            occupied[i] = false

    function hash_function(key)
        return key mod size

    function insert_record(id, name, department)
        index = hash_function(id)
        original_index = index
        while occupied[index] is true
            index = (index + 1) mod size
            if index == original_index
                print "hash table is full"
                return
        table[index].id = id
        table[index].name = name
        table[index].department = department
        occupied[index] = true
        print "record inserted successfully"

    function search_record(id)
        index = hash_function(id)
        original_index = index
        while occupied[index] is true
            if table[index].id == id
                print "record found"
                print "id:", table[index].id, "name:", table[index].name, "department:",
table[index].department
                return
            index = (index + 1) mod size
            if index == original_index
                break
        print "record not found"

main function
    create object of hashtable
    initialize hash table with size from user
    repeat
```

display menu
1. insert record
2. search record
3. exit
enter user choice
perform appropriate operation
until choice is 3

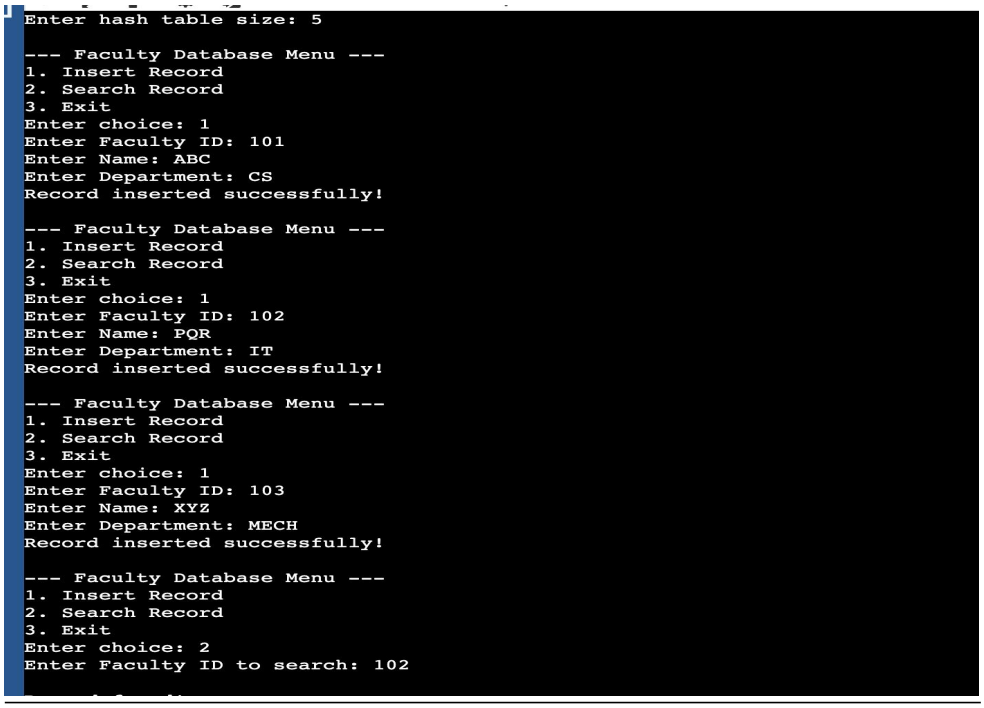
CODE:

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

class Faculty {
public:
    int id;
    string name;
    string department;
    Faculty() {
        id = -1;
        name = "";
        department = "";
    }
};

class HashTable {
    Faculty* table;
    bool* occupied;
    int size;

public:
    HashTable(int s) {
        size = s;
        table = new Faculty[size];
        occupied = new bool[size];
        for (int i = 0; i < size; i++)
            occupied[i] = false;
    }

    int hashFunction(int key) {
        return key % size;
    }

    void insertRecord(int id, string name, string dept) {
        int index = hashFunction(id);
        int original = index;

        while (occupied[index]) {
            index = (index + 1) % size;
            if (index == original) {
```

```
            cout << "Hash table is full!" << endl;
            return;
        }
    }

    table[index].id = id;
    table[index].name = name;
    table[index].department = dept;
    occupied[index] = true;
    cout << "Record inserted successfully!\n";
}

void searchRecord(int id) {
    int index = hashFunction(id);
    int original = index;

    while (occupied[index]) {
        if (table[index].id == id) {
            cout << "\nRecord found!" << endl;
            cout << "Faculty ID: " << table[index].id << endl;
            cout << "Name: " << table[index].name << endl;
            cout << "Department: " << table[index].department << endl;
            return;
        }
        index = (index + 1) % size;
        if (index == original) break;
    }

    cout << "Record not found!\n";
}
};

int main() {
    int size;
    cout << "Enter hash table size: ";
    cin >> size;

    HashTable ht(size);
    int choice, id;
    string name, dept;

    do {
        cout << "\n--- Faculty Database Menu ---\n";
        cout << "1. Insert Record\n2. Search Record\n3. Exit\nEnter choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter Faculty ID: ";
                cin >> id;
```

```
            cout << "Enter Name: ";
            cin >> name;
            cout << "Enter Department: ";
            cin >> dept;
            ht.insertRecord(id, name, dept);
            break;

        case 2:
            cout << "Enter Faculty ID to search: ";
            cin >> id;
            ht.searchRecord(id);
            break;

        case 3:
            cout << "Exiting program.\n";
            break;

        default:
            cout << "Invalid choice!\n";
      }
   } while (choice != 3);

   return 0;
}
```

OUTPUT:



```
Enter hash table size: 5

--- Faculty Database Menu ---
1. Insert Record
2. Search Record
3. Exit
Enter choice: 1
Enter Faculty ID: 101
Enter Name: ABC
Enter Department: CS
Record inserted successfully!

--- Faculty Database Menu ---
1. Insert Record
2. Search Record
3. Exit
Enter choice: 1
Enter Faculty ID: 102
Enter Name: PQR
Enter Department: IT
Record inserted successfully!

--- Faculty Database Menu ---
1. Insert Record
2. Search Record
3. Exit
Enter choice: 1
Enter Faculty ID: 103
Enter Name: XYZ
Enter Department: MECH
Record inserted successfully!

--- Faculty Database Menu ---
1. Insert Record
2. Search Record
3. Exit
Enter choice: 2
Enter Faculty ID to search: 102
```

```
--- Faculty Database Menu ---
1. Insert Record
2. Search Record
3. Exit
Enter choice: 1
Enter Faculty ID: 103
Enter Name: XYZ
Enter Department: MECH
Record inserted successfully!

--- Faculty Database Menu ---
1. Insert Record
2. Search Record
3. Exit
Enter choice: 2
Enter Faculty ID to search: 102

Record found!
Faculty ID: 102
Name: PQR
Department: IT

--- Faculty Database Menu ---
1. Insert Record
2. Search Record
3. Exit
Enter choice: 3
Exiting program.


...Program finished with exit code 0
```

CONCLUSION:

In this program, a faculty database was successfully implemented using a hash table with linear probing as a collision resolution technique. The use of the MOD hash function ensured uniform data distribution, while linear probing effectively handled collisions by finding the next available slot.
This experiment demonstrates the importance of hashing for fast data access and retrieval in databases and highlights how efficient collision resolution strategies are essential for maintaining performance in hash-based systems.