Assignment No - 12

PROBLEM STATEMENT 1:

"WAP to simulate a faculty database as a hash table. Search a particular faculty by using 'divide' as a hash function for linear probing with the chaining without replacement method of collision handling technique. Assume suitable data for faculty record"

OBJECTIVES:

The main objective is to simulate a faculty database using a hash table with a divide hash function.
It demonstrates linear probing with chaining to handle collisions efficiently.
The program shows how without replacement insertion works, where a new record is inserted in the next available position without replacing existing records.
Students learn how to search for a faculty record quickly using their faculty ID.
The program also teaches practical implementation of collision handling strategies in real-world database systems.

THEORY:

A hash table stores data for quick access using a hash function to compute an index. The divide method computes the hash as index = key / table_size (integer division), which distributes the records across the table. Collisions occur when two keys map to the same index.

In linear probing with chaining without replacement, each table index contains a linked list (chain) to hold all records that hash to the same index. The "without replacement" method ensures that no existing record is moved; instead, the new record is placed in the next available position or at the end of the chain.

This approach is useful for dynamic databases such as faculty or student records. Searching involves hashing the key and traversing the chain if necessary. It allows efficient storage, prevents data loss, and simplifies insertion logic compared to replacement methods.

ALGORITHM:

```
#include<iostream>
using namespace std

class faculty
   int id
   string name
   string dept
```

```
    faculty* next
    constructor faculty(int i, string n, string d)
        id = i
        name = n
        dept = d
        next = null

class hashtable
    int size
    faculty** table

    constructor hashtable(int s)
        size = s
        table = new array of faculty pointers
        initialize each index to null

    function hash_function(id)
        return id / size

    function insert_record(id, name, dept)
        index = hash_function(id)
        create new faculty node
        if table[index] is null
            table[index] = new node
        else
            temp = table[index]
            while temp->next != null
                temp = temp->next
            temp->next = new node

    function search_record(id)
        index = hash_function(id)
        temp = table[index]
        while temp != null
            if temp->id == id
                print record details
                return
            temp = temp->next
        print "record not found"

    function display()
        for i = 0 to size-1
            print all records in chain

main()
    create hash table object
    repeat menu for insert, search, display
```

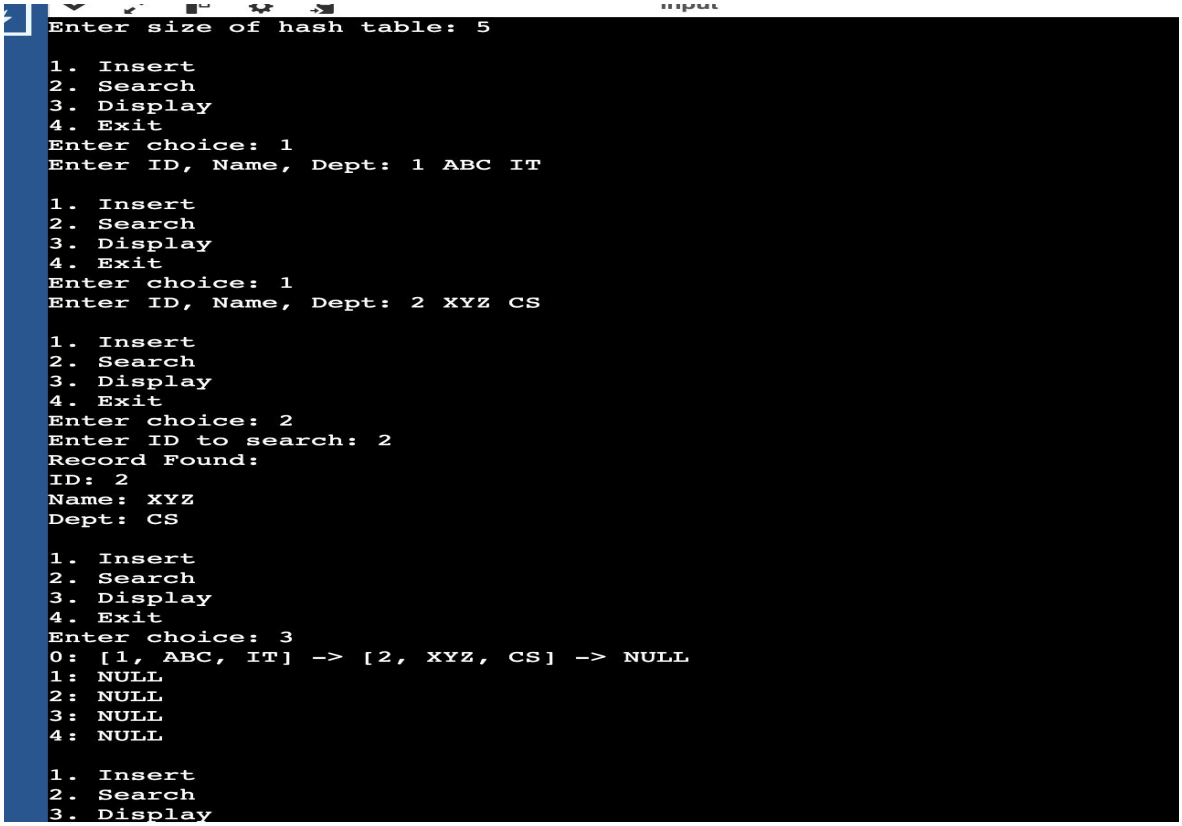
CODE:

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

class Faculty {
public:
    int id;
    string name;
    string dept;
    Faculty* next;
    Faculty(int i, string n, string d) {
        id = i;
        name = n;
        dept = d;
        next = NULL;
    }
};

class HashTable {
    int size;
    Faculty** table;

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

    int hashFunc(int id) {
        return id / size;
    }

    void insertRecord(int id, string name, string dept) {
        int index = hashFunc(id);
        Faculty* newNode = new Faculty(id, name, dept);

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

```
            temp->next = newNode;
        }
    }

    void searchRecord(int id) {
        int index = hashFunc(id);
        Faculty* temp = table[index];
        while (temp != NULL) {
            if (temp->id == id) {
                cout << "Record Found:\nID: " << temp->id << "\nName: " << temp-
>name << "\nDept: " << temp->dept << endl;
                return;
            }
            temp = temp->next;
        }
        cout << "Record not found.\n";
    }

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

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

    int choice, id;
    string name, dept;

    do {
        cout << "\n1. Insert\n2. Search\n3. Display\n4. Exit\nEnter choice: ";
```

```
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter ID, Name, Dept: ";
                cin >> id >> name >> dept;
                ht.insertRecord(id, name, dept);
                break;
            case 2:
                cout << "Enter ID to search: ";
                cin >> id;
                ht.searchRecord(id);
                break;
            case 3:
                ht.display();
                break;
        }
    } while (choice != 4);
    return 0;
}
```

OUTPUT:



CONCLUSION:

This program simulates a faculty database using linear probing with chaining without replacement. The divide method ensures keys are mapped to indices, while chaining allows multiple records at the same index. The "without replacement" approach keeps existing records intact, providing efficient and safe insertion and search operations.

Assignment No 12

PROBLEM STATEMENT 2 :

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

OBJECTIVE :

The main objective is to implement a faculty database using a hash table with the MOD hash function.

It demonstrates linear probing with chaining and replacement to handle collisions effectively.

The program shows how with replacement insertion works, where a new record may replace an existing record if it belongs to a different hash index.

It teaches students how to search efficiently in a hash table with replacement, maintaining minimal probing steps.

The program reinforces practical understanding of advanced collision handling techniques used in real-world database systems.

THEORY :

Hash tables provide fast insertion and retrieval using a hash function to compute indices. The MOD method calculates the hash index as index = key % table_size. Collisions occur when two keys map to the same index.

In linear probing with chaining with replacement, if a new record's hash index coincides with an existing record but the existing record belongs to a different hash index, the new record replaces the existing one, and the displaced record is reinserted using linear probing. This ensures the table remains more efficient and reduces long probe sequences.

Each index contains a linked list (chain) to store multiple records mapped to the same index. This method is widely used in databases where efficient insertion and search operations are critical. Searching involves computing the hash and traversing the chain if necessary. The "with replacement" technique optimizes table occupancy and reduces collisions during future insertions.

ALGORITHM :

```
#include<iostream>
using namespace std

class faculty
   int id
   string name
   string dept
   faculty* next
   constructor faculty(int i, string n, string d)
      id = i
      name = n
      dept = d
      next = null

class hashtable
   int size
   faculty** table
   bool* occupied

   constructor hashtable(int s)
      size = s
      table = new array of faculty pointers
      occupied = new bool array
      initialize each index to null and false

   function hash_function(id)
      return id mod size

   function insert_record(id, name, dept)
      index = hash_function(id)
      if occupied[index] is false
         table[index] = new faculty node
         occupied[index] = true
      else
         if hash_function(table[index].id) != index
            swap table[index] with new node
            reinsert displaced node using linear probing
         else
            temp = table[index]
            while temp->next != null
               temp = temp->next
            temp->next = new node

   function search_record(id)
      index = hash_function(id)
      temp = table[index]
      while temp != null
         if temp->id == id
            print record details
```

```
            return
        temp = temp->next
    print "record not found"

function display()
    for i = 0 to size-1
        print all records in chain

main()
    create hash table object
    repeat menu for insert, search, display
```

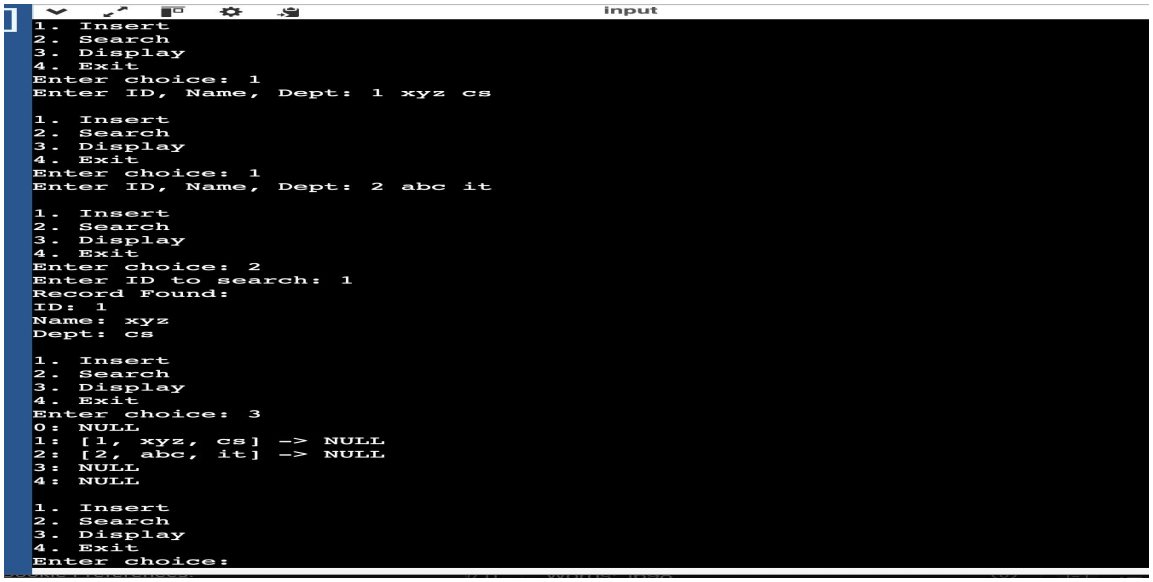
CODE :

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

class Faculty {
public:
    int id;
    string name;
    string dept;
    Faculty* next;
    Faculty(int i, string n, string d) {
        id = i;
        name = n;
        dept = d;
        next = NULL;
    }
};

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

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

    int hashFunc(int id) {
        return id % size;
```

```
    }

    void insertRecord(int id, string name, string dept) {
        int index = hashFunc(id);
        Faculty* newNode = new Faculty(id, name, dept);

        if (!occupied[index]) {
            table[index] = newNode;
            occupied[index] = true;
        } else {
            int existingHash = hashFunc(table[index]->id);
            if (existingHash != index) {
                // Swap with new node
                Faculty* temp = table[index];
                table[index] = newNode;
                // Reinsert displaced node
                int probe = (index + 1) % size;
                while (occupied[probe]) probe = (probe + 1) % size;
                table[probe] = temp;
                occupied[probe] = true;
            } else {
                Faculty* temp = table[index];
                while (temp->next != NULL)
                    temp = temp->next;
                temp->next = newNode;
            }
        }
    }

    void searchRecord(int id) {
        int index = hashFunc(id);
        Faculty* temp = table[index];
        while (temp != NULL) {
            if (temp->id == id) {
                cout << "Record Found:\nID: " << temp->id << "\nName: " << temp->name <<
"\nDept: " << temp->dept << endl;
                return;
            }
            temp = temp->next;
        }
        cout << "Record not found.\n";
    }

    void display() {
        for (int i = 0; i < size; i++) {
            cout << i << ": ";
            Faculty* temp = table[i];
            while (temp != NULL) {
                cout << "[" << temp->id << ", " << temp->name << ", " << temp->dept << "] ->
";
```

```
            temp = temp->next;
        }
        cout << "NULL\n";
    }
  }
};

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

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

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

    switch (choice) {
      case 1:
        cout << "Enter ID, Name, Dept: ";
        cin >> id >> name >> dept;
        ht.insertRecord(id, name, dept);
        break;
      case 2:
        cout << "Enter ID to search: ";
        cin >> id;
        ht.searchRecord(id);
        break;
      case 3:
        ht.display();
        break;
    }
  } while (choice != 4);

  return 0;
}
```

OUTPUT:



CONCLUSION :

This program successfully demonstrates a faculty database using linear probing with chaining WITH replacement. The MOD hash function provides efficient indexing, while the replacement strategy ensures that new records can occupy positions of less suitable records, reducing long probe chains and maintaining efficient access. This approach is highly practical for database systems where quick insertion and search are required.