lab_12.cpp

```cpp
//<----Lab 12 - Hashing---->

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

class HashMapTable
{
public:
    int table_size;
    list<int> *table;

    HashMapTable(int key)
    {
        this->table_size = key;
        table = new list<int>[table_size];
    }

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

    void insertElement(int key)
    {
        int index = hashFunction(key);
        table[index].push_back(key);
    }

    void deleteElement(int key)
    {
        int index = hashFunction(key);
        list<int>::iterator i = find(table[index].begin(), table[index].end(), key);

        if (i != table[index].end())
        {
            table[index].erase(i);
        }
    }

    void displayHashTable()
    {
        for (int i = 0; i < table_size; i++)
        {
            cout << i;
            for (auto j : table[i])
            {
                cout << " ==> " << j;
            }
            cout << endl;
        }
    }
```

```cpp
};

// Q1. Implement a hashing algorithm using Linear Probing.

class LinearProbingHashTable : public HashMapTable
{
public:
    LinearProbingHashTable(int key) : HashMapTable(key) {}

    void insertElementLinearProbing(int key)
    {
        int index = hashFunction(key);

        while (!table[index].empty())
        {
            index = (index + 1) % table_size;
        }

        table[index].push_back(key);
    }
};

// Q2. Implement a hashing algorithm using Quadratic Probing.

class QuadraticProbingHashTable : public HashMapTable
{
public:
    QuadraticProbingHashTable(int key) : HashMapTable(key) {}

    void insertElementQuadraticProbing(int key)
    {
        int index = hashFunction(key);
        int i = 1;

        while (!table[index].empty())
        {
            index = (index + i * i) % table_size;
            i++;
        }

        table[index].push_back(key);
    }
};

// Q3. Implement a hashing algorithm using Double Hashing.

class DoubleHashingHashTable : public HashMapTable
{
public:
    DoubleHashingHashTable(int key) : HashMapTable(key) {}

    int hashFunction2(int key)
    {
        return 1 + (key % (table_size - 2)); // Ensure step size is relatively prime to table_size
    }
```

```cpp

    void insertElementDoubleHashing(int key)
    {
        int index = hashFunction(key);
        int step = hashFunction2(key);

        // Iterate until an empty slot is found or a full cycle is completed
        while (!table[index].empty() && table[index].front() != key)
        { // Check for existing key
            index = (index + step) % table_size;
            if (index == hashFunction(key))
            { // Full cycle check
                cout << "Hash table is full\n";
                return;
            }
        }

        table[index].push_back(key);
    }
};

int main()
{
    int arr[] = {1,2,3,4,5,6};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Linear Probing
    LinearProbingHashTable linearTable(6);
    for (int i = 0; i < n; i++)
        linearTable.insertElementLinearProbing(arr[i]);

    cout << "Linear Probing Hash Table:" << endl;
    linearTable.displayHashTable();
    cout << endl;

    // Quadratic Probing
    QuadraticProbingHashTable quadraticTable(6);
    for (int i = 0; i < n; i++)
        quadraticTable.insertElementQuadraticProbing(arr[i]);

    cout << "Quadratic Probing Hash Table:" << endl;
    quadraticTable.displayHashTable();
    cout << endl;

    // Double Hashing
    DoubleHashingHashTable doubleHashingTable(6);
    for (int i = 0; i < n; i++)
        doubleHashingTable.insertElementDoubleHashing(arr[i]);

    cout << "Double Hashing Hash Table:" << endl;
    doubleHashingTable.displayHashTable();

    return 0;
}

```