# Department of Computing

# School of Electrical Engineering and Computer Science

**CS-250: Data Structure and Algorithms**

**Class: BSCS**

**Lab 12: Implementation of Hash Table**

**Name : Muhammad Haadhee Sheeraz Mian**

**Reg No 478359**

**Date: 7th August, 2025**

**Time: 01:00 pm – 4:50 pm**

# Lab 12: Implementation of Hash Table

**Objectives**

* Learn to implement basic hash table functionality.
* Understand collision resolution techniques: Linear Probing, Quadratic Probing, and Double Hashing.
* Compare lookup performance across techniques.

**Tools/Software Requirement**

Visual Studio C++

**Introduction**

A **hash table** is a data structure that allows quick access to data using a key. The key is passed through a **hash function** to determine its storage location in an array. However, two keys may generate the same location (a collision). To resolve this, we use techniques

* **Linear Probing,**
* **Quadratic Probing,**
* **Double Hashing.**

.

**Lab Tasks**

**Task 1: Direct Addressing (Simplified)**

* Implement a hash table using an array to store key-value pairs.
* Use keys as indices to directly access values.
* Write functions for:
  1. Inserting a key-value pair.
  2. Searching for a value using a key.
  3. Deleting a key-value pair.
* #include <iostream>
* using namespace std;
* #define maxSize 1000
* int map[maxSize];
* void initializeMap() {
* for (int i = 0; i < maxSize; i++) {
* map[i] = -1;
* }
* }
* void insert (int key, int value){
* if (key < 0 || key >= maxSize) {
* cerr << "Invalid key. Key must be between 0 and " << maxSize - 1 << "." << endl;
* return;
* }
* map[key] = value;
* cout << "Inserted key-value pair (" << key << ", " << value << ") into the map." << endl;
* }
* void search (int key) {
* if (key < 0 || key >= maxSize) {
* cerr << "Invalid key. Key must be between 0 and " << maxSize - 1 << "." << endl;
* return;
* }
* if (map[key] != -1) {
* cout << "Value for key " << key << " is: " << map[key] << endl;
* } else {
* cout << "Key " << key << " not found in the map, with value: " << map[key] << endl;
* }
* }
* void remove(int key, int value){
* if (key < 0 || key >= maxSize) {
* cerr << "Invalid key. Key must be between 0 and " << maxSize - 1 << "." << endl;
* return;
* }
* if (map[key] == value) {
* map[key] = -1; // Remove the key-value pair
* cout << "Removed key-value pair (" << key << ", " << value << ") from the map." << endl;
* } else {
* cout << "Key " << key << " not found with value: " << value << endl;
* }
* }
* int main() {
* initializeMap();
* // Test insert
* insert(42, 99);
* insert(5, 123);
* // Test search
* search(42);
* search(10); // should return -1
* // Test delete
* remove(42, 99);
* search(42); // should return -1
* return 0;
* }

A screenshot of a computer code

AI-generated content may be incorrect.

**Task 2: Hash Table with Linear Probing**

**Linear Probing**

Collisions occur when two keys hash to the same index. In Linear Probing, we resolve collisions by looking for the next available slot. If a collision occurs, check the next index, then the next, and so on, until a free slot is found

 Implement a hash table using a hash function (e.g., key % table\_size).

 Handle collisions using linear probing (i.e., checking the next available slot).

#include <iostream>

using namespace std;

const int TABLE\_SIZE = 7;

int hashTable[TABLE\_SIZE];

// hash function

int hashFunc(int key) {

    return key % TABLE\_SIZE;

}

void initTable() {

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

        hashTable[i] = -1;

    }

}

void insert(int key) {

    int index = hashFunc(key);

    int startIndex = index;

    while (hashTable[index] != -1 && hashTable[index] != -2) {

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

        if (index == startIndex) {

            cout << "Hash table is full!\n";

            return;

        }

    }

    hashTable[index] = key;

}

int search(int key) {

    int index = hashFunc(key);

    int startIndex = index;

    while (hashTable[index] != -1) {

        if (hashTable[index] == key) {

            return index;

        }

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

        if (index == startIndex) {

            break;

        }

    }

    return -1; // not found

}

void removeKey(int key) {

    int pos = search(key);

    if (pos == -1) {

        cout << "Key not found\n";

    } else {

        hashTable[pos] = -2; // mark as deleted

    }

}

void display() {

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

        cout << i << ": " << hashTable[i] << endl;

    }

}

int main() {

    initTable();

    insert(10);

    insert(20);

    insert(5);

    insert(15); // collision example

    display();

    cout << "Searching 15 → index " << search(15) << endl;

    removeKey(20);

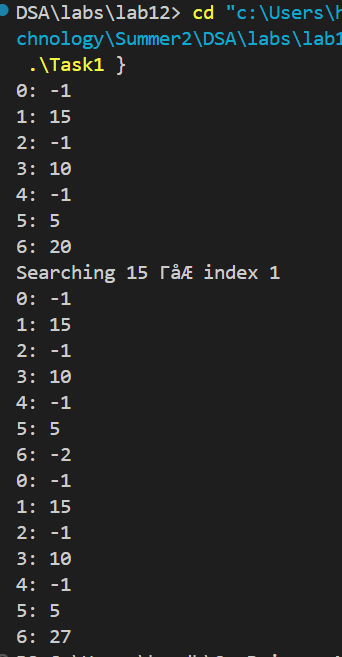
    display();

    insert(27); // should reuse deleted slot

    display();

    return 0;

}

****

**Task 3: Quadratic Probing**

Quadratic Probing is another way to resolve collisions. Instead of searching linearly, the step size increases quadratically. For example, if the initial collision happens at index i, the next indices checked are (i + 1²), (i + 2²), and so on

 Extend the hash table implementation to use quadratic probing for collisions.

#include <iostream>

using namespace std;

const int TABLE\_SIZE = 7;

int hashTable[TABLE\_SIZE];

// Hash function

int hashFunc(int key) {

    return key % TABLE\_SIZE;

}

// Initialize hash table

void initTable() {

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

        hashTable[i] = -1; // empty

    }

}

// Insert using Quadratic Probing

void insert(int key) {

    int index = hashFunc(key);

    if (hashTable[index] == -1 || hashTable[index] == -2) {

        hashTable[index] = key;

        return;

    }

    // Collision -> quadratic probing

    for (int i = 1; i < TABLE\_SIZE; i++) {

        int newIndex = (index + i \* i) % TABLE\_SIZE;

        if (hashTable[newIndex] == -1 || hashTable[newIndex] == -2) {

            hashTable[newIndex] = key;

            return;

        }

    }

    cout << "Hash table is full! Cannot insert " << key << endl;

}

// Search using Quadratic Probing

int search(int key) {

    int index = hashFunc(key);

    if (hashTable[index] == key) return index;

    for (int i = 1; i < TABLE\_SIZE; i++) {

        int newIndex = (index + i \* i) % TABLE\_SIZE;

        if (hashTable[newIndex] == key) return newIndex;

        if (hashTable[newIndex] == -1) return -1; // empty slot means not found

    }

    return -1;

}

// Delete key

void removeKey(int key) {

    int pos = search(key);

    if (pos == -1) {

        cout << "Key not found\n";

    } else {

        hashTable[pos] = -2; // mark as deleted

    }

}

// Display hash table

void display() {

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

        cout << i << ": " << hashTable[i] << endl;

    }

}

int main() {

    initTable();

    insert(10);

    insert(20);

    insert(5);

    insert(15);

    insert(7);

    insert(17); // will use quadratic probing

    cout << "Hash Table after insertions:\n";

    display();

    cout << "\nSearching 15 → index " << search(15) << endl;

    cout << "Searching 99 → index " << search(99) << endl;

    cout << "\nDeleting 20...\n";

    removeKey(20);

    display();

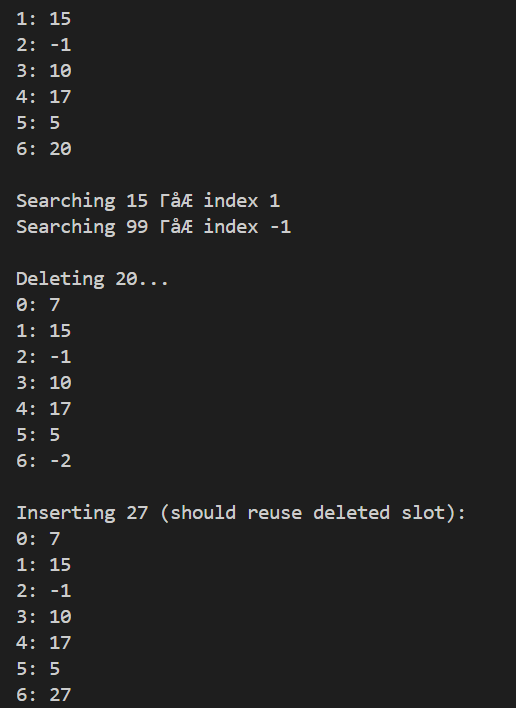
    cout << "\nInserting 27 (should reuse deleted slot):\n";

    insert(27);

    display();

    return 0;

}



**Task 4: Double Hashing**

Double hashing is a collision resolution technique used in open addressing. When a collision occurs, double hashing uses a second hash function to calculate a new step size or offset to probe the next slot in the hash table. This ensures better distribution of keys and reduces clustering (i.e., consecutive filled slots).

 Implement a hash table using double hashing.

 Define a second hash function to reduce clustering issues.

 Test the implementation with different hash functions.

#include <iostream>

using namespace std;

class DoubleHashing {

private:

    int\* table;

    int size;

    int prime; // for second hash function

    // First hash function

    int hash1(int key) {

        return key % size;

    }

    // Second hash function

    int hash2(int key) {

        return prime - (key % prime);

    }

public:

    DoubleHashing(int tableSize, int primeNum) {

        size = tableSize;

        prime = primeNum;

        table = new int[size];

        for (int i = 0; i < size; i++) {

            table[i] = -1; // -1 means empty

        }

    }

    void insert(int key) {

        int index = hash1(key);

        // If no collision

        if (table[index] == -1) {

            table[index] = key;

            return;

        }

        // Collision → apply double hashing

        int stepSize = hash2(key);

        int i = 1;

        while (i < size) {

            int newIndex = (index + i \* stepSize) % size;

            if (table[newIndex] == -1) {

                table[newIndex] = key;

                return;

            }

            i++;

        }

        cout << "Hash table is full, cannot insert " << key << endl;

    }

    void search(int key) {

        int index = hash1(key);

        int stepSize = hash2(key);

        int i = 0;

        while (table[(index + i \* stepSize) % size] != -1 && i < size) {

            if (table[(index + i \* stepSize) % size] == key) {

                cout << "Found " << key << " at index " << (index + i \* stepSize) % size << endl;

                return;

            }

            i++;

        }

        cout << key << " not found in hash table." << endl;

    }

    void display() {

        for (int i = 0; i < size; i++) {

            cout << i << " : " << table[i] << endl;

        }

    }

};

int main() {

    int tableSize = 7; // Table size

    int primeNum = 5;  // Prime number less than tableSize

    DoubleHashing dh(tableSize, primeNum);

    dh.insert(7);

    dh.insert(12);

    dh.insert(17);

    dh.insert(19);

    dh.insert(26);

    dh.insert(31);

    dh.insert(38); // should show full if table is filled

    cout << "\nHash Table:\n";

    dh.display();

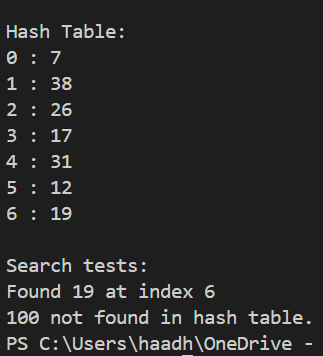
    cout << "\nSearch tests:\n";

    dh.search(19);

    dh.search(100);

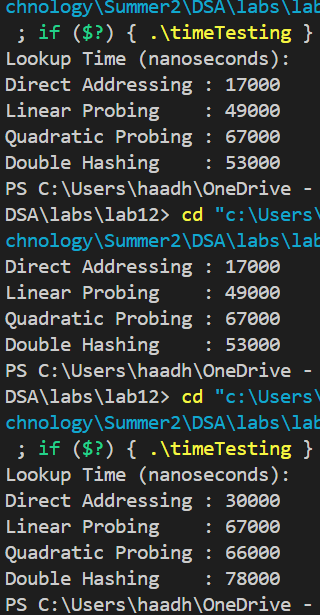
    return 0;

}



#### ****Task 4: Compare Lookup Time****

* Generate random keys and insert them into the hash table.
* Create a comparison function to measure the time taken for lookup operations in:
  + Direct Addressing
  + Linear Probing
  + Quadratic Probing
  + Double Hashing
* Create a simple table summarizing your results.
* #include <iostream>
* #include <vector>
* #include <chrono>
* #include <cstdlib>
* #include <ctime>
* using namespace std;
* using namespace std::chrono;
* // ======================= Direct Addressing =======================
* class DirectAddressing {
* private:
* vector<int> table;
* public:
* DirectAddressing(int maxKey) {
* table.assign(maxKey + 1, -1);
* }
* void insert(int key) {
* table[key] = key;
* }
* bool search(int key) {
* return table[key] != -1;
* }
* };
* // ======================= Linear Probing =======================
* class LinearProbing {
* private:
* vector<int> table;
* int size;
* public:
* LinearProbing(int s) {
* size = s;
* table.assign(size, -1);
* }
* int hash(int key) {
* return key % size;
* }
* void insert(int key) {
* int index = hash(key);
* int i = 0;
* while (table[(index + i) % size] != -1) {
* i++;
* }
* table[(index + i) % size] = key;
* }
* bool search(int key) {
* int index = hash(key);
* int i = 0;
* while (table[(index + i) % size] != -1 && i < size) {
* if (table[(index + i) % size] == key) return true;
* i++;
* }
* return false;
* }
* };
* // ======================= Quadratic Probing =======================
* class QuadraticProbing {
* private:
* vector<int> table;
* int size;
* public:
* QuadraticProbing(int s) {
* size = s;
* table.assign(size, -1);
* }
* int hash(int key) {
* return key % size;
* }
* void insert(int key) {
* int index = hash(key);
* int i = 0;
* while (table[(index + i\*i) % size] != -1 && i < size) {
* i++;
* }
* table[(index + i\*i) % size] = key;
* }
* bool search(int key) {
* int index = hash(key);
* int i = 0;
* while (table[(index + i\*i) % size] != -1 && i < size) {
* if (table[(index + i\*i) % size] == key) return true;
* i++;
* }
* return false;
* }
* };
* // ======================= Double Hashing =======================
* class DoubleHashing {
* private:
* vector<int> table;
* int size;
* int prime;
* public:
* DoubleHashing(int s, int p) {
* size = s;
* prime = p;
* table.assign(size, -1);
* }
* int hash1(int key) {
* return key % size;
* }
* int hash2(int key) {
* return prime - (key % prime);
* }
* void insert(int key) {
* int index = hash1(key);
* if (table[index] == -1) {
* table[index] = key;
* return;
* }
* int step = hash2(key);
* int i = 1;
* while (i < size) {
* int newIndex = (index + i \* step) % size;
* if (table[newIndex] == -1) {
* table[newIndex] = key;
* return;
* }
* i++;
* }
* }
* bool search(int key) {
* int index = hash1(key);
* int step = hash2(key);
* int i = 0;
* while (table[(index + i \* step) % size] != -1 && i < size) {
* if (table[(index + i \* step) % size] == key) return true;
* i++;
* }
* return false;
* }
* };
* // ======================= Main Program =======================
* int main() {
* srand(time(0));
* int numKeys = 1000;
* int tableSize = 2003; // prime number for better distribution
* int maxKeyValue = 10000;
* // Generate random keys
* vector<int> keys(numKeys);
* for (int i = 0; i < numKeys; i++) {
* keys[i] = rand() % maxKeyValue;
* }
* // Initialize all hash tables
* DirectAddressing da(maxKeyValue);
* LinearProbing lp(tableSize);
* QuadraticProbing qp(tableSize);
* DoubleHashing dh(tableSize, 1999); // prime < tableSize
* // Insert into all tables
* for (int key : keys) {
* da.insert(key);
* lp.insert(key);
* qp.insert(key);
* dh.insert(key);
* }
* // Measure lookup times
* auto measureTime = [&](auto& tableObj) {
* auto start = high\_resolution\_clock::now();
* for (int key : keys) {
* tableObj.search(key);
* }
* auto end = high\_resolution\_clock::now();
* return duration\_cast<nanoseconds>(end - start).count();
* };
* long long timeDA = measureTime(da);
* long long timeLP = measureTime(lp);
* long long timeQP = measureTime(qp);
* long long timeDH = measureTime(dh);
* // Print results
* cout << "Lookup Time (nanoseconds):\n";
* cout << "Direct Addressing : " << timeDA << "\n";
* cout << "Linear Probing    : " << timeLP << "\n";
* cout << "Quadratic Probing : " << timeQP << "\n";
* cout << "Double Hashing    : " << timeDH << "\n";
* return 0;
* }



| **Run #** | **Direct Addressing (ns)** | **Linear Probing (ns)** | **Quadratic Probing (ns)** | **Double Hashing (ns)** |
| --- | --- | --- | --- | --- |
| 1 | 24000 | 59000 | 79000 | 65000 |
| 2 | 17000 | 63000 | 61000 | 51000 |
| 3 | 17000 | 58000 | 43000 | 50000 |
| 4 | 31000 | 98000 | 71000 | 189000 |
| 5 | 17000 | 49000 | 67000 | 53000 |
| 6 | 30000 | 67000 | 66000 | 78000 |

**Important Note:** Practice your knowledge of OOP with C++ when creating a solution. Remember to comment your code properly. Inappropriate or no comment may result in deduction of marks.

**Deliverables**

C++ code file implementing the above tasks.

Brief comments in the code explaining the logic.

Word Documents includes the screenshots of code and output.

**Note:** Students are required to upload the lab on LMS before deadline.