Collision Handling, Load Factor & Rehashing, Hash Table Implementation

1. Collision-Handling Schemes

Introduction

A collision occurs in a hash table when two different keys generate the same index using the hash function.

Since a hash table has limited size, collisions are unavoidable in real-world applications.

Collision handling schemes are techniques used to resolve these conflicts so that all values can still be stored efficiently.

Why Collisions Happen

- Limited table size
- Non-uniform hash functions
- Large number of keys
- Poor distribution of hash values

Main Collision Handling Techniques

1. Separate Chaining

Each bucket stores a linked list of elements.

Index 3 → 12 → 22 → 32

2. Open Addressing

All elements are stored in the array itself.

Methods include:

- Linear probing
- Quadratic probing
- Double hashing

Linear Probing Example

If index is full, check next slot:

h(x) = x % 10

10 → index 0
20 → index 0 (collision)
→ place at index 1

Advantages

- Efficient average performance
- Simple implementation
- Works well with good hash functions

Disadvantages

- Clustering problem (open addressing)
- Extra memory for chaining
- Performance degrades at high load

2. Load Factor and Rehashing

Load Factor

Introduction

Load factor measures how full a hash table is.

It determines when the hash table should be resized to maintain efficiency.

Formula

Load Factor = Number of Elements / Table Size

Example

If:

- Elements = 8

- Table size = 10

Then:

Load Factor = 8 / 10 = 0.8

Importance

- Indicates performance of hash table
- Higher load factor → more collisions
- Lower load factor → wasted memory

Rehashing

Introduction

Rehashing is the process of increasing hash table size and redistributing all elements when load factor becomes too high.

It improves performance by reducing collisions.

Steps of Rehashing

1. Create a new larger table (usually double size)
2. Recalculate hash for each element
3. Insert elements into new table
4. Delete old table

When to Rehash

- Load factor > 0.7 or 0.75 (common threshold)

Advantages

- Reduces collisions
- Improves performance
- Maintains O(1) average complexity

Disadvantages

- Expensive operation (O(n))

- Temporary performance delay

3. C++ Hash Table Implementation

Features Included

- Hash function
- Collision handling (chaining)
- Insert
- Search
- Display
- Load factor
- Simple rehashing concept

C++ Implementation Program

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

class HashTable {
private:
   vector<list<int>> table;
   int size;
   int elements;

   // Hash Function
   int hashFunction(int key) {
      return key % size;
   }

public:
   HashTable(int s) {
      size = s;
      table.resize(size);
      elements = 0;
   }

   // Load Factor
   double loadFactor() {
      return (double)elements / size;
   }

   // Insert element
   void insert(int key) {
      int index = hashFunction(key);
```

```
        table[index].push_back(key);
        elements++;

        // Check load factor
        if (loadFactor() > 0.75) {
            cout << "\nRehashing needed (Load Factor > 0.75)\n";
            rehash();
        }
    }

    // Search element
    bool search(int key) {
        int index = hashFunction(key);

        for (int val : table[index]) {
            if (val == key)
                return true;
        }
        return false;
    }

    // Display table
    void display() {
        for (int i = 0; i < size; i++) {
            cout << i << " --> ";
            for (int val : table[i]) {
                cout << val << " ";
            }
            cout << endl;
        }
    }

    // Rehashing function
    void rehash() {
        vector<list<int>> oldTable = table;

        size = size * 2;
        table.clear();
        table.resize(size);
        elements = 0;

        for (auto bucket : oldTable) {
            for (int val : bucket) {
                insert(val);
            }
        }
    }
};

int main() {
    HashTable ht(5);
```

```
    ht.insert(10);
    ht.insert(20);
    ht.insert(30);
    ht.insert(40);
    ht.insert(50);

    cout << "\nHash Table:\n";
    ht.display();

    cout << "\nSearch 30: "
         << (ht.search(30) ? "Found" : "Not Found") << endl;

    cout << "Load Factor: " << ht.loadFactor() << endl;

    return 0;
}
```

Complexity Analysis

Operation	Average Case	Worst Case
Insert	O(1)	O(n)
Search	O(1)	O(n)
Rehashing	O(n)	O(n)

Comparison Table

Concept	Purpose
Collision Handling	Resolves index conflicts
Separate Chaining	Uses linked lists
Open Addressing	Finds next available slot
Load Factor	Measures table fullness
Rehashing	Resizes and rebuilds table

Key Takeaways

- Collision occurs when multiple keys map to same index
- Collision handling ensures all data is stored correctly
- Load factor determines hash table efficiency
- Rehashing improves performance by resizing table
- Chaining is simple and widely used
- Hash tables provide **O(1) average search time**
- Used in:
    - Databases
    - Compilers

- Caches
- STL unordered_map