1️⃣ Hash Table Basics

A **hash table** is a data structure that maps **keys** → **values** using a **hash function**.

- Hash function: Converts key into an **index** in an array.
- Ideal hash table operations:
  - `Insert` → $O(1)$ average
  - `Search` → $O(1)$ average
  - `Delete` → $O(1)$ average

C++ STL: `unordered_map` is a built-in hash table.
Example:

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

int main() {
    unordered_map<string, int> mp;

    // Insert key-value pairs
    mp["apple"] = 5;
    mp["banana"] = 3;
    mp["orange"] = 10;

    // Access value using key
    cout << "apple count: " << mp["apple"] << endl;

    // Check if key exists
    if (mp.find("banana") != mp.end()) {
        cout << "Banana is present" << endl;
    }

    // Iterate through hash table
    for (auto &p : mp) {
        cout << p.first << " -> " << p.second << endl;
    }

    return 0;
}
```

✅ **Output:**

```
apple count: 5
Banana is present
apple -> 5
banana -> 3
orange -> 10
```

2️⃣ Frequency Counting

We can count **occurrences of elements** easily using a hash table.

```cpp
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;

int main() {
    vector<int> arr = {1, 2, 2, 3, 1, 2, 4};
    unordered_map<int, int> freq;

    // Count frequency
    for (int x : arr) {
        freq[x]++;
    }

    // Print frequencies
    for (auto &p : freq) {
        cout << p.first << " occurs " << p.second << " times\n";
    }

    return 0;
}
```

✅ **Output:**

```
1 occurs 2 times
2 occurs 3 times
3 occurs 1 times
4 occurs 1 times
```

3️⃣ Hash Sets for Unique Elements

A **hash set** stores **only unique elements**.
In C++, we use `unordered_set`.

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

int main() {
    vector<int> arr = {1, 2, 2, 3, 4, 4, 5};
    unordered_set<int> s;

    // Insert elements
    for (int x : arr) {
        s.insert(x);
    }

    // Print unique elements
    for (int x : s) {
        cout << x << " ";
    }

    return 0;
}
```

✅ **Output:** (order may vary)

```
1 2 3 4 5
```

4️⃣ Collision Handling

Collisions happen when **two keys map to the same index**. There are **two main methods**:

A) Chaining

- Store a **list** at each bucket.
- If collision occurs, append to the list.

Simplified Example:

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

class HashTable {
    static const int SIZE = 10;
    list<pair<int, int>> table[SIZE];

public:
    int hashFunction(int key) { return key % SIZE; }

    void insert(int key, int value) {
        int idx = hashFunction(key);
        table[idx].push_back({key, value});
    }

    void display() {
        for (int i = 0; i < SIZE; i++) {
            cout << i << ": ";
            for (auto &p : table[i])
                cout << "(" << p.first << "," << p.second << ") ";
            cout << endl;
        }
    }
};

int main() {
    HashTable ht;
    ht.insert(1, 10);
    ht.insert(11, 20); // collision with 1
    ht.insert(2, 30);
    ht.display();
}
```

✅ **Output:**

```
0:
1: (1,10) (11,20)
2: (2,30)
3:
...
```

B) Open Addressing

- Store all elements in the array itself.
- If a spot is occupied, find **next available spot** (linear probing, quadratic probing, double hashing).

Example: Linear Probing

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

class HashTable {
    static const int SIZE = 10;
    int table[SIZE];
public:
    HashTable() {
        for(int i=0;i<SIZE;i++) table[i]=-1;
    }

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

    void insert(int key) {
        int idx = hashFunction(key);
        while(table[idx] != -1) { // collision
            idx = (idx + 1) % SIZE;
        }
        table[idx] = key;
    }

    void display() {
        for(int i=0;i<SIZE;i++)
            cout << i << " -> " << table[i] << endl;
    }
};

int main() {
    HashTable ht;
    ht.insert(1);
    ht.insert(11); // collision with 1
    ht.insert(21); // collision with 1 & 11
    ht.display();
}
```

✅ **Output:**

```
0 -> -1
1 -> 1
2 -> 11
3 -> 21
...
```

Summary:

1. **Hash Table** → key-value mapping.
2. **Frequency Counting** → `unordered_map`.
3. **Unique Elements** → `unordered_set`.
4. **Collisions** → Handled by:
    - **Chaining** (lists at each bucket)
    - **Open Addressing** (find next free spot)

1️⃣ What is a Hash Table?

- Think of a **hash table** like a **school locker system**:
    - Each locker has a number (index).
    - You store your books (values) in a locker using a **key** (like your name).
- **Key** → **Value** mapping.
- Fast way to **store** and **find** things.

Example in C++:

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

int main() {
    unordered_map<string, int> myLocker;

    // Put things in locker
    myLocker["apple"] = 5;
```

```
    myLocker["banana"] = 3;

    // Find how many apples
    cout << "Apples: " << myLocker["apple"] << endl;

    return 0;
}
```

✅ Output:

```
Apples: 5
```

2️⃣ Counting How Many Times Something Appears

If you have a list of numbers and want to know **how many times each number appears**, use a **hash map**.

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

int main() {
    int arr[] = {1, 2, 2, 3, 1, 2};
    unordered_map<int, int> freq;

    // Count frequency
    for(int x : arr) {
        freq[x]++; // add 1 every time we see x
    }

    // Print counts
    for(auto p : freq) {
        cout << p.first << " appears " << p.second << " times\n";
    }
}
```

✅ Output:

```
1 appears 2 times
2 appears 3 times
3 appears 1 times
```

3️⃣ Keeping Only Unique Elements

If you want **no duplicates**, use a **hash set**.

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

int main() {
    int arr[] = {1, 2, 2, 3, 4, 4, 5};
    unordered_set<int> s;

    for(int x : arr) {
        s.insert(x); // only keeps unique
    }

    // Print unique numbers
    for(int x : s) {
        cout << x << " ";
    }
}
```

✅ Output: (order may vary)

```
1 2 3 4 5
```

4️⃣ Collisions (Easy Explanation)

- Sometimes **two keys want the same locker** → collision.
- Two ways to fix:

A) Chaining (List in each locker)

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

int main() {
    list<int> lockers[5]; // 5 lockers

    // Put numbers in locker using key % 5
    int nums[] = {1, 6, 11}; // all go to locker 1 (1%5, 6%5, 11%5)
```

```
    for(int x : nums) {
        int locker = x % 5;
        lockers[locker].push_back(x);
    }

    // Print lockers
    for(int i = 0; i < 5; i++) {
        cout << "Locker " << i << ": ";
        for(int x : lockers[i]) cout << x << " ";
        cout << endl;
    }
}
```

✅ Output:

```
Locker 0:
Locker 1: 1 6 11
Locker 2:
Locker 3:
Locker 4:
```

Explanation: All numbers that go to the same locker are stored in a **list**.

B) Open Addressing (Find next empty locker)

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

int main() {
    int lockers[5] = {-1,-1,-1,-1,-1}; // -1 means empty

    int nums[] = {1, 6, 11};
    for(int x : nums) {
        int locker = x % 5;
        while(lockers[locker] != -1) { // if occupied, go next
            locker = (locker + 1) % 5;
        }
        lockers[locker] = x;
    }

    // Print lockers
    for(int i = 0; i < 5; i++) cout << "Locker " << i << ": " << lockers[i] <<
endl;
}
```

✅ Output:

```
Locker 0: -1
Locker 1: 1
Locker 2: 6
Locker 3: 11
Locker 4: -1
```

Explanation: If a locker is full, we move to the **next empty locker**.

Summary for Beginners ✅

Concept	Use	C++ STL
Hash Table	Key → Value storage	`unordered_map`
Frequency Counting	Count occurrences	`unordered_map`
Unique Elements	Keep only one copy	`unordered_set`
Collision Handling	When two keys go to same spot	Chaining / Open Addressing