Module 7 - Hash Map

1. Main Concept of Hash Map

Hashing is the process of converting data of any size (like a string, number, or object) into a fixed-length integer value. This integer value is called a **"hash value,"** "hash code," or simply "hash."

The primary data structure that uses this concept is called a **Hash Table**.

1.1. Components of a Hash System

A hash-based search system consists of three main components:

- **Hash Table:** This is fundamentally an array (or a `std::vector`). Each "row" or "slot" in this array is called a **bucket**. The size of this table (`m`) is a key factor in its performance.
- **Key:** This is the data you want to store or look up. The key is fed into the hash function to find its proper location. For example, in a phone book, the ***name*** is the key.
- **Hash Function:** This is the "engine" or mathematical function that takes your **key** and computes an **index** (a row number from `0` to `m-1`) within the array where your data should be stored.

The main goal of hashing is to achieve extremely fast data access. In the ideal case, the hash function provides a unique index for every key, allowing insertion, search, and deletion operations to be performed in constant time, or **`O(1)`**.

Hash Map Analogy:

Imagine a large file cabinet (**Hash Table**) with 100 drawers numbered 0-99. To store a document for "Budi," you don't search one by one. You use a rule (**Hash Function**), for example, "Take the first letter ('B' -> 2), store it in drawer #2." When you need to find "Budi," you compute the same rule and immediately jump to drawer #2.

1.2. Good Hash Function Properties

The **Hash Function** is the most critical part of the Hash Table. A good function must have the following properties:

- **Deterministic:** The same input (key) must *always* produce the same output (hash value).
- **Efficient:** The hash function must be fast to compute (ideally `O(1)`).
- **Uniform Distribution:** This is the most important. The hash function must spread keys as evenly as possible across all slots (indices) in the table. This minimizes collisions.

If the hash function doesn't distribute data well (e.g., it hashes all names starting with 'A', 'B', and 'C' to the same slot), it will cause **collisions**, and the performance will degrade significantly.

1.3. Common Hash Function Methods

There are several methods to design a hash function, each with different properties. The goal is always to create a **"well-distributed" hash** that avoids clustering keys in the same buckets.

1.3.1. Division Method

This is the most common and simplest method. It takes the key (which must be a numeric value or convertible to one) and finds the remainder after dividing by the table size.

- **Formula:** `h(key) = key % tableSize`
- **Pros:** It is extremely fast, involving only a single modular arithmetic operation.
- **Cons:** The performance is highly dependent on the choice of `tableSize`. If `tableSize` is poorly chosen (e.g., a power of 10 or 2), and the input keys have a pattern (e.g., all keys are even), this can lead to massive collisions. This is why it is strongly recommended to make `tableSize` a **prime number** that is not close to a power of 2

1.3.2. Multiplication Method

This method is a popular choice because it is far less sensitive to the `tableSize` (which can be any value, often a power of 2 for efficiency).

- **Formula:** `h(key) = floor( tableSize * ( (key * A) % 1 ) )`
- **Explanation:**
  - **`key * A`:** The key is multiplied by a constant A (where $0 < A < 1$). A common and effective value for A is related to the golden ratio.
  - **`( ... ) % 1:`** This operation extracts only the fractional part of the result. This fractional part is highly mixed and influenced by all digits of the key.
  - **`tableSize * ( ... )`:** This fractional part is then scaled up to the range of the table.
  - **`floor( ... )`:** The integer part is taken as the final index.
- **Pros:** This method effectively scrambles the input bits, meaning that similar keys (like 100 and 101) are likely to be mapped to very different indices. The choice of `tableSize` is not critical.

1.3.3. Mid-Square Method

This method is particularly effective at breaking up patterns in keys that are sequential or have similar prefixes/suffixes.

- **Concept:** The key is first squared. The resulting number is often very large. A set of digits is then extracted from the middle of this squared result to be used as the hash index.
- **Why it works:** The middle digits of a squared number are influenced by all the digits of the original key. Changes in either the beginning or end of the key will cause significant changes in the middle digits, effectively "folding" the key in on itself and scrambling any existing patterns.
- **Pros:** Good at breaking up non-random sequences in input keys.

1.3.4. Folding Method

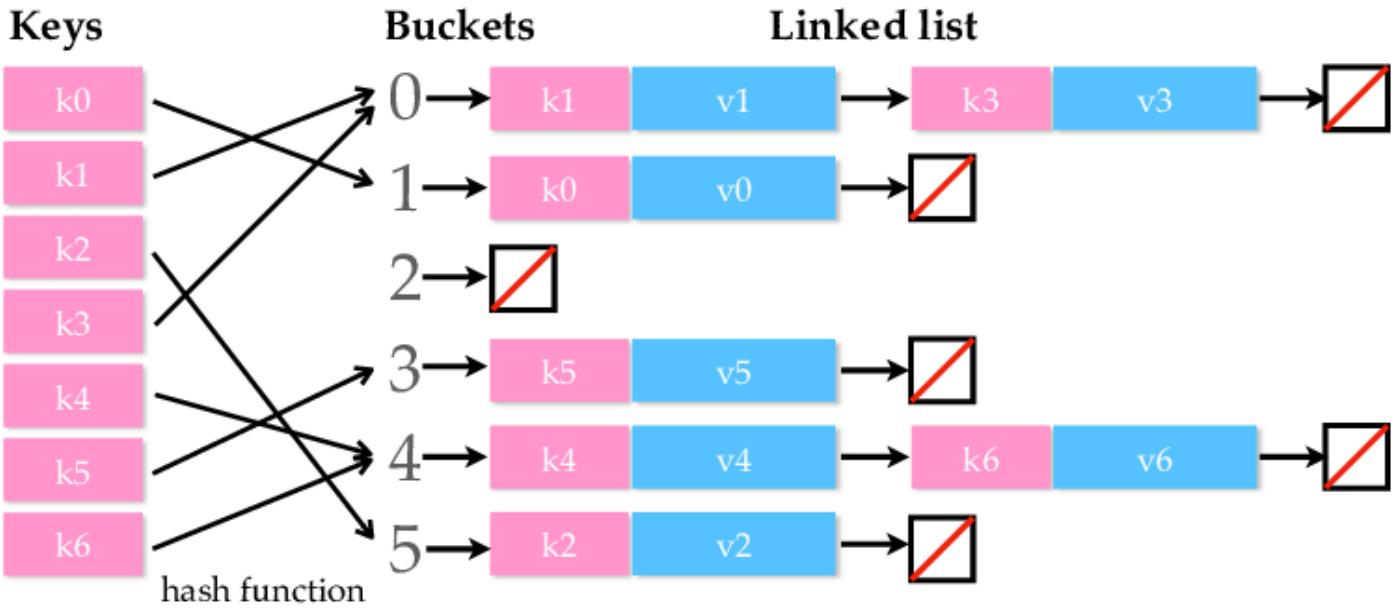
This method is excellent for hashing keys that are very long, such as account numbers, ISBNs, or long strings, where other methods might lead to overflow or only use a portion of the key.

- **Concept:** The key is divided into several equally-sized parts. These parts are then combined using a simple operation.
- **Operations:**
  - **Addition:** All parts are added together. The final result may need an extra modulo (`% tableSize`) if it exceeds the table size.
  - **XOR:** All parts are combined using the bitwise XOR operation. This is very fast and naturally stays within bounds if the parts are sized correctly.
- **Pros:** It ensures that all parts of a long key (beginning, middle, and end) contribute to the final hash value, making it highly distributive for long, structured keys.

2. Collision Handling

A **Collision** occurs when two or more different keys produce **the same hash value** (index). For example, "Budi" and "Dina" are both hashed to **row #5**. We can't overwrite Budi's data. We must have a strategy to handle this.

2.1. Separate Chaining



This is the most common strategy to avoid collisions.

- **Concept:** Instead of each table row storing a single value, each row stores a pointer to another data structure, usually a **Linked List**.
- **How it Works:**
  1. "Budi" hashes to row 5. We create a linked list at row 5 and add "Budi".
  2. "Dina" hashes to row 5. We add "Dina" to the linked list that already exists at row 5.
  3. Row 5 now contains: `[Budi] -> [Dina] -> NULL`
- **Search:** To find "Dina," we compute its hash (5), go to row 5, and then traverse the linked list at that row until we find "Dina." The worst-case search time becomes `O(k)` where `k` is the number of elements in the chain.

2.1.1. Manual Implementation: Insertion

The `insert` function handles adding a new key-value pair. Its logic is:

1. Hash the `key` to find the correct bucket `index`.
2. Get the linked list (the chain) at that `index`.

3. Traverse the list:
    - If the `key` is already in the list, **update** its `value` and stop.
    - If the end of the list is reached and the `key` was not found, **add** the new key-value pair to the end of the list.

```
void insert(std::string key, int value) {
    // Hash the key to get the bucket index
    int index = hashFunction(key);

    // Get the chain (linked list) at that index
    std::list<KeyValuePair>& chain = table[index];

    // Traverse the chain
    for (auto& pair : chain) {
        if (pair.key == key) {
            // Key found: update the value and return
            pair.value = value;
            return;
        }
    }

    // Key not found: add new pair to the end of the list
    chain.push_back(KeyValuePair(key, value));
}
```

2.1.2. Manual Implementation: Searching

The `search` function finds the value associated with a key. Its logic is:

1. Hash the `key` to find the bucket `index`.
2. Get the linked list at that `index`.
3. Traverse the list:
    - If the `key` is found, return its `value`.
    - If the end of the list is reached and the `key` was not found, return a sentinel value (like -1) or throw an exception.

```
int search(std::string key) {
    // Hash the key to get the bucket index
    int index = hashFunction(key);

    // Get the chain at that index
```

```
    std::list<KeyValuePair>& chain = table[index];

    // Traverse the chain
    for (auto& pair : chain) {
        if (pair.key == key) {
            // Key found: return the value
            return pair.value;
        }
    }

    // Key not found
    return -1;
}
```

2.1.3. Manual Implementation: Deletion

The remove function deletes a key-value pair. Its logic is:

1. Hash the `key` to find the bucket `index`.
2. Get the linked list at that `index`.
3. Traverse the list using an **iterator**. We must use an iterator because we need to know the position of the element to delete it.
4. If the `key` is found:
   - Call the list's `erase()` method, passing the iterator.
   - Stop and return.

```
void remove(std::string key) {
    // Hash the key to get the bucket index
    int index = hashFunction(key);

    // Get a reference to the chain
    auto& chain = table[index];

    // Traverse the chain using an iterator
    for (auto it = chain.begin(); it != chain.end(); ++it) {
        // 'it' is an iterator (position pointer)
        // 'it->key' accesses the key of the element at that position
        if (it->key == key) {
            // Key found: erase the element at this position
            chain.erase(it);
```

```
            return;
        }
    }
    // If loop finishes, key was not found. Do nothing.
}
```

2.2. Open Addressing (Probing)

This is an alternative strategy where all data is stored within the table itself. No linked lists are used.

- **Concept:** If a collision occurs (the slot is full), we "probe" for the next empty slot according to a specific rule and place the item there.
- **Types of Probing:**
  1. **Linear Probing:** This is the simplest rule. If slot `h(key)` is full, try `h(key) + 1`, then `h(key) + 2`, `h(key) + 3`, and so on, wrapping around the table if necessary. The **problem** of this type is it tends to **create clusters of occupied slots**, which degrades performance as the table fills up.
  2. **Quadratic Probing:** If slot `h(key)` is full, try `h(key) + 1^2`, then `h(key) + 2^2`, `h(key) + 3^2`, and so on. This helps spread out elements better than linear probing.
  3. **Double Hashing:** Uses a second hash function to determine the "step size" for probing, which is the most effective at avoiding clusters.

3. Load Factor and Rehashing

3.1. Load Factor (?)

The **Load Factor (λ)** is a measure of how full the hash table is. It is a critical metric for performance.

- **Formula:** `λ = m/n`
  - `n` = total number of items stored in the table.
  - `m` = total size of the hash table (number of buckets).
- **Impact on Performance:**
  - **Separate Chaining:** λ can be greater than 1. A higher λ means longer linked lists, and search time degrades towards `O(λ)` or `O(n)` in the worst case.
  - **Open Addressing:** λ cannot be greater than 1. As λ approaches 1, it becomes extremely difficult and slow to find an empty slot, and performance grinds to a halt

3.2. Rehashing

To maintain good performance (close to `O(1)`), we must keep the load factor low. When the load factor exceeds a certain threshold (e.g., **λ > 0.75**), the table is "rehashed."

Rehashing is the process of creating a new, larger hash table and re-inserting all existing elements into it. Here is the **process** of rehashing:

- **Trigger:** The load factor exceeds a predefined threshold (e.g., 0.75).
- **Allocate:** Create a new, larger hash table (e.g., `2 * m`, or the next prime number).
- **Re-insert:** Iterate through **every element** in the old table.
- **Re-hash**: For each element, compute its **new hash value** based on the **new, larger table size**.
- **Insert**: Place the element in its new correct slot in the new table.
- **Deallocate:** Free the memory of the old table.

Rehashing is an `O(n)` operation. It is slow and computationally expensive, but it happens infrequently. Its cost is "amortized" over many `O(1)` insertions, so the average insertion time remains `O(1)`.

3.3 Rehashing Implementation

```
// We must track item count 'n' and table size 'm'

int n = 0; // Number of items
int m = 10; // Initial table size
std::vector<std::list<KeyValuePair>> table;
const float MAX_LOAD_FACTOR = 0.75;

void rehash() {
    // Get old table and size
    std::vector<std::list<KeyValuePair>> oldTable = table;
    int oldSize = m;

    // Allocate new, larger table
    m = m * 2;
    table.clear();
    table.resize(m);
    n = 0; // Reset item count

    // Re-insert all elements from old table
    for (int i = 0; i < oldSize; ++i) {
        for (auto& pair : oldTable[i]) {
            // Re-hash and insert into new table
            insert(pair.key, pair.value);
        }
    }
}

void insert(std::string key, int value) {
    // Check load factor before inserting
    if ((float)n / m > MAX_LOAD_FACTOR) {
        rehash();
    }

    // Hash the key (uses the new 'm' if rehashed)
    int index = hashFunction(key);

    // Traverse chain to find or update
    for (auto& pair : table[index]) {
        if (pair.key == key) {
            pair.value = value;
```

```
            return;
        }
    }

    // Not found, add new pair and increment item count
    table[index].push_back(KeyValuePair(key, value));
    n++;
}
```

4. Example: Full Manual Implementation

This chapter combines all the concepts from **Chapters 2 and 3** into a single, complete `MyHashMap` class. This class handles insertion, searching, deletion, and automatic rehashing.

4.1. Manual Implementation Using Separate Chaining

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

// Data Node: Struct to store Key-Value pairs
struct KeyValuePair {
    std::string key;
    int value;
    KeyValuePair(std::string k, int v) : key(k), value(v) {}
};

class MyHashMap {
private:
    int n; // Number of items
    int m; // Number of buckets (table size)
    std::vector<std::list<KeyValuePair>> table;
    const float MAX_LOAD_FACTOR = 0.75;

    // Hash Function (Simple Division Method)
    int hashFunction(std::string key) {
        int hash = 0;
        // A simple hash: sum ASCII values
        for (char c : key) {
            hash = (hash + (int)c);
        }
```

```
        return hash % m; // Use current table size 'm'
    }

    // Rehashing Function
    void rehash() {
        // Get old table and size
        std::vector<std::list<KeyValuePair>> oldTable = table;
        int oldSize = m;

        // Allocate new, larger table
        m = m * 2;
        table.clear();
        table.resize(m);
        n = 0; // Reset item count

        std::cout << ". New size: " << m << std::endl;

        // Re-insert all elements from old table
        for (int i = 0; i < oldSize; ++i) {
            for (auto& pair : oldTable[i]) {
                // Re-hash and insert into new table
                insert(pair.key, pair.value);
            }
        }
    }

public:
    // Constructor: Initialize table
    MyHashMap(int size = 10) { // Default size of 10
        n = 0;
        m = size;
        table.resize(m);
    }

    // Insert Function (with rehashing)
    void insert(std::string key, int value) {
        // Check load factor before inserting
        if ((float)(n + 1) / m > MAX_LOAD_FACTOR) {
            rehash();
        }
```

```
    // Hash the key (uses the new 'm' if rehashed)
    int index = hashFunction(key);

    // Traverse chain to find or update
    for (auto& pair : table[index]) {
        if (pair.key == key) {
            pair.value = value;
            return;
        }
    }

    // Not found, add new pair and increment item count
    table[index].push_back(KeyValuePair(key, value));
    n++;
}

// Search Function
int search(std::string key) {
    int index = hashFunction(key);
    for (auto& pair : table[index]) {
        if (pair.key == key) {
            return pair.value;
        }
    }
    return -1; // Not found
}

// Remove Function
void remove(std::string key) {
    int index = hashFunction(key);
    auto& chain = table[index];

    for (auto it = chain.begin(); it != chain.end(); ++it) {
        if (it->key == key) {
            chain.erase(it); // Remove element at position 'it'
            n--; // Decrement item count
            return;
        }
    }
```

```
    }
};
```

4.2. Manual Implementation Using Open Addressing

```
#include <iostream>
#include <string>
#include <vector>

// Define the state for each slot
enum class SlotState {
    EMPTY,    // The slot has never been used
    OCCUPIED, // The slot contains an active key-value pair
    DELETED   // The slot used to have data, but it was removed
};

struct HashSlot {
    std::string key;
    int value;
    SlotState state;

    // Constructor to initialize new slots as EMPTY
    HashSlot() : key(""), value(0), state(SlotState::EMPTY) {}
};

class MyHashMap {
private:
    int n; // Number of items
    int m; // Number of buckets (table size)
    std::vector<HashSlot> table;
    const float MAX_LOAD_FACTOR = 0.75;

    // Hash Function (Simple Division Method)
    int hashFunction(std::string key) {
        int hash = 0;
        for (char c : key) {
            hash = (hash + (int)c);
        }
```

```
        // Ensure hash is positive before modulo
        return (hash & 0x7FFFFFFF) % m;
    }

    // Rehashing Function
    void rehash() {
        std::cout << "[Info] Rehashing triggered. Old size: " << m;

        // Save the old table
        std::vector<HashSlot> oldTable = table;
        int oldSize = m;

        // Create a new, larger table.
        m = m * 2;
        table.clear();
        table.resize(m);
        n = 0; // Reset item count

        std::cout << ". New size: " << m << std::endl;

        // 6. Re-insert all active elements from the old table
        for (int i = 0; i < oldSize; ++i) {
            // Only re-insert items that are currently occupied
            if (oldTable[i].state == SlotState::OCCUPIED) {
                insert(oldTable[i].key, oldTable[i].value);
            }
        }
    }

public:
    // Constructor: Initialize table
    MyHashMap(int size = 10) { // Default size of 10
        n = 0;
        m = size;
        table.resize(m);
    }

    // Insert Function
    void insert(std::string key, int value) {
        // Check load factor before inserting
```

```
if ((float)(n + 1) / m > MAX_LOAD_FACTOR) {
    rehash();
}

// Get the initial hash index
int index = hashFunction(key);
int firstDeletedSlot = -1; // To store the index of the first DELETED slot

// Start probing (loop max 'm' times)
for (int i = 0; i < m; ++i) {
    int probedIndex = (index + i) % m;
    auto& slot = table[probedIndex]; // Get a reference to the slot

    // Case 1: Slot is OCCUPIED and the key matches
    if (slot.state == SlotState::OCCUPIED && slot.key == key) {
        // Key already exists, just update the value
        slot.value = value;
        return;
    }

    // Case 2: Slot is EMPTY
    if (slot.state == SlotState::EMPTY) {
        // We found an empty slot, so the key is not in the table.
        // We should insert it here, OR in the first DELETED slot we found.
        int insertIndex = (firstDeletedSlot != -1) ? firstDeletedSlot : probedIndex;

        table[insertIndex].key = key;
        table[insertIndex].value = value;
        table[insertIndex].state = SlotState::OCCUPIED;
        n++; // Increment item count
        return;
    }

    // Case 3: Slot is DELETED
    if (slot.state == SlotState::DELETED) {
        // We can insert here, but we must keep searching
        // to see if the key already exists further down the probe chain.
        if (firstDeletedSlot == -1) {
            firstDeletedSlot = probedIndex;
        }
```

```
        }
    }
}

// Search Function
int search(std::string key) {
    // Get the initial hash index
    int index = hashFunction(key);

    // Start probing
    for (int i = 0; i < m; ++i) {
        int probedIndex = (index + i) % m;
        auto& slot = table[probedIndex];

        // Case 1: Slot is OCCUPIED and key matches
        if (slot.state == SlotState::OCCUPIED && slot.key == key) {
            return slot.value; // Item found
        }

        // Case 2: Slot is EMPTY
        if (slot.state == SlotState::EMPTY) {
            // If we hit an EMPTY slot, the key cannot be
            // further down the probe chain.
            return -1; // Not found
        }

        // Case 3: Slot is DELETED
        // We must continue probing, as the key we want
        // might have collided and be after this deleted slot.
        if (slot.state == SlotState::DELETED) {
            continue;
        }
    }

    // We looped through the entire table and didn't find it
    return -1;
}

// Remove Function
void remove(std::string key) {
```

```
        int index = hashFunction(key);

        // Start probing to find the key
        for (int i = 0; i < m; ++i) {
            int probedIndex = (index + i) % m;
            auto& slot = table[probedIndex];

            // Case 1: Slot is OCCUPIED and key matches
            if (slot.state == SlotState::OCCUPIED && slot.key == key) {
                // Found it. Set state to DELETED.
                slot.state = SlotState::DELETED;
                n--; // Decrement item count
                return;
            }

            // Case 2: Slot is EMPTY
            if (slot.state == SlotState::EMPTY) {
                // Key is not in the table
                return;
            }

            // Case 3: Slot is DELETED
            if (slot.state == SlotState::DELETED) {
                // Keep probing
                continue;
            }
        }
    }
};
```

5. Hashing Implementation with C++ STL

In C++, instead of creating a Hash Table manually, you could use **Standard Template Library (STL)**. The STL implementation **automatically** handles hash functions, collisions, and **rehashing** when the load factor gets too high.

5.1. `std::unordered_map` (Key-Value)

`std::unordered_map` is a Hash Map implementation for **Key-Value pairs**. The purpose of this template is to map Keys to Values. It acts like a **dictionary**, where you look up a word (key) to get its definition (value).

```
#include <iostream>
#include <string>
#include <unordered_map>

int main() {
    // Key: string (name), Value: int (grade)
    std::unordered_map<std::string, int> studentGrades;

    studentGrades["Budi"] = 90;
    studentGrades["Ani"] = 85;

    // Access value using key
    std::cout << "Budi's Grade: " << studentGrades["Budi"] << std::endl;
}
```

5.2. `std::unordered_set` (Unique Keys)

`std::unordered_set` is a Hash Set implementation that only stores **unique keys**. This template is used to check whether a **key is on the list or not**, like a **guest book**.

```
#include <iostream>
#include <string>
#include <unordered_set>

int main() {
    std::unordered_set<std::string> attendanceList;

    attendanceList.insert("Budi");
    attendanceList.insert("Ani");
    attendanceList.insert("Budi"); // Ignored, because "Budi" already exists

    // Check existence
    if (attendanceList.count("Budi") > 0) {
        std::cout << "Budi is present." << std::endl;
    }
}
```

5.3. Comparison: `map` vs. `set`

Aspect	`map`	`set`
Feature	`std::unordered_map<Key, Value>`	`std::unordered_map<Key>`
What is Stored	Key-Value pairs	Key only
Element Type	`std::pair<const Key, Value>`	`Key`
Main Purpose	Mapping Keys to Values	Storing unique Keys
Data Access	`map[key]` (get/set value)	`set.count(key)` (check existence)

6. Custom Struct Keys

A notable limitation of the C++ STL's `std::unordered_map` and `std::unordered_set` is their inability to natively support user-defined types (such as a `struct` or `class`) as keys. An attempt to instantiate a map with a custom struct, as shown below, will result in a compile-time error.

```
// This declaration will fail to compile
std::unordered_map<Mahasiswa, int> nilaiUjian;
```

This failure occurs because the STL hash containers have two fundamental requirements for their key types, which C++ cannot automatically generate for a custom `struct`:

- **A Hashing Function**: The container must have a mechanism to convert a `Key` object (in this case, `Mahasiswa`) into a `size_t` value. This resulting "hash code" is what determines the bucket where the element will be stored.
- **An Equality Function**: The container must know how to compare two Key objects for equality. This is not for sorting, but specifically for handling hash collisions. The map must be able to determine if the key it's looking for is truly present.

The compiler cannot, for instance, assume whether two `Mahasiswa` objects are equal if only their `nim` matches, or if both `nim` and `nama` must match. To resolve this, the programmer must explicitly "teach" C++ how to perform these two operations for the custom type.

6.1. Defining the Equality Operation

The first requirement is met by **overloading the equality operator** (`operator==`) for the custom struct. This is the standard C++ mechanism for defining identity and is used directly by the `unordered_map` to compare keys within a bucket.

```
struct Mahasiswa {
    std::string nama;
    int nim;
};

// This function defines what makes two Mahasiswa objects
// "equal" in the eyes of the hash map.
bool operator==(const Mahasiswa& a, const Mahasiswa& b) {
    return a.nama == b.nama && a.nim == b.nim;
}
```

6.2. Providing a Hashing Function via `std::hash` Specialization

The second requirement is met by providing a custom hash function. The standard mechanism for this is to create a **template specialization** of the `std::hash` struct, which resides in the std namespace.

This specialization "injects" our custom hashing logic into the STL, allowing `unordered_map` to find and use it automatically whenever it needs to hash a `Mahasiswa` object. This is achieved by defining `operator()` within the specialized struct.

```
#include <functional> // Required for std::hash

// We "inject" our hashing logic into the std namespace
namespace std {

// We declare a specialization of the 'hash' template for our type
template<>
struct hash<Mahasiswa> {

    // operator() is what the unordered_map will call.
    // It must be 'const' and return a 'size_t'.
    size_t operator()(const Mahasiswa& m) const {

        // Get the hash for each member individually.
        size_t hashNama = std::hash<std::string>{}(m.nama);
        size_t hashNim = std::hash<int>{}(m.nim);

        // Combine the individual hashes into one final hash.
        return hashNama ^ (hashNim << 1);
    }
};

} // End of namespace std
```

6.3. Usage Example

With both the equality operator and the `std::hash` specialization provided, the `std::unordered_map` now has all the components necessary to manage the Mahasiswa key. The following code will now compile and function as expected

```
int main() {
    std::unordered_map<Mahasiswa, int> nilaiUjian;

    Mahasiswa budi = {"Budi", 12345};
    nilaiUjian[budi] = 90;

    // The map can now hash "budi" to find a bucket
    // and use operator== to check for collisions.
}
```