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;
            }
        }
    }
};
```

Revision #3
Created 2025-11-04 15:26:11 UTC by RE
Updated 2025-11-05 09:59:13 UTC by RE