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

using namespace std;

// HashTable class for implementing a hash table using linear probing
class HashTable {
private:
    // Internal structure to represent a node in the hash table
    struct HashNode {
        string key;      // Key for the hash table
        int value;       // Associated value for the key
        bool isDeleted;  // Flag to mark if the node is deleted

        // Constructor to initialize a hash node
        HashNode(string k, int v) : key(k), value(v), isDeleted(false) {}
    };

    vector<HashNode*> table; // The hash table (a vector of HashNode pointers)
    int capacity;            // Current capacity of the hash table
    int size;                // Number of active elements in the hash table

    // Hash function to compute the index for a given key
    int hashFunction(const string& key) {
        int hash = 0;
        for (char ch : key) {
            hash = (hash * 31 + ch) % capacity; // Generate hash using a prime
multiplier
        }
        return hash; // Return index within the table bounds
    }

    // Resize the hash table when the load factor exceeds the threshold
    void resizeTable() {
        int oldCapacity = capacity;      // Store the old capacity
        capacity *= 2;                   // Double the capacity
        vector<HashNode*> oldTable = table; // Backup the old table

        // Clear the current table and reinitialize it with new capacity
        table.clear();
        table.resize(capacity, nullptr);
        size = 0; // Reset size since we'll re-insert all active elements

        // Rehash all non-deleted elements from the old table
        for (int i = 0; i < oldCapacity; ++i) {
            if (oldTable[i] && !oldTable[i]->isDeleted) {
                insert(oldTable[i]->key, oldTable[i]->value); // Reinsert into new
table
            }
            delete oldTable[i]; // Free memory for old nodes
        }
    }

    // Calculate the current load factor of the hash table
    float loadFactor() {
        return (float)size / capacity; // Size divided by capacity
    }
```

```
public:
    // Constructor to initialize hash table with a given capacity
    HashTable(int cap) : capacity(cap), size(0) {
        table.resize(capacity, nullptr); // Initialize table with nullptrs
    }

    // Destructor to clean up memory
    ~HashTable() {
        for (HashNode* node : table) {
            delete node; // Free memory for all nodes
        }
    }

    // Insert or update a key-value pair in the hash table
    void insert(const string& key, int value) {
        if (loadFactor() > 0.7) { // Check if resizing is needed
            resizeTable();
        }

        int index = hashFunction(key); // Get the hash index for the key

        // Linear probing to find an available slot or update existing key
        while (table[index] && !table[index]->isDeleted && table[index]->key !=
key) {
            index = (index + 1) % capacity; // Move to the next slot
        }

        // If slot is empty or marked as deleted, insert a new node
        if (!table[index] || table[index]->isDeleted) {
            if (!table[index]) { // Only increment size for truly empty slots
                ++size;
            }
            delete table[index]; // Clean up any old data at the slot
            table[index] = new HashNode(key, value); // Insert new node
        } else {
            table[index]->value = value; // Update value if key exists
        }
    }

    // Remove a key-value pair by marking it as deleted
    void remove(const string& key) {
        int index = hashFunction(key); // Get the hash index for the key

        // Linear probing to locate the key
        for (int i = 0; i < capacity; ++i) {
            int probeIndex = (index + i) % capacity;
            if (!table[probeIndex]) return; // Key not found, exit
            if (table[probeIndex]->key == key && !table[probeIndex]->isDeleted) {
                table[probeIndex]->isDeleted = true; // Mark as deleted
                --size; // Decrease size
                return;
            }
        }
    }

    // Retrieve the value associated with a given key
    int get(const string& key) {
        int index = hashFunction(key); // Get the hash index for the key
```

```
        // Linear probing to locate the key
        for (int i = 0; i < capacity; ++i) {
            int probeIndex = (index + i) % capacity;
            if (!table[probeIndex]) break; // Stop search if slot is empty
            if (table[probeIndex]->key == key && !table[probeIndex]->isDeleted) {
                return table[probeIndex]->value; // Return value if key is found
            }
        }
        throw runtime_error("Key not found"); // Key doesn't exist
    }

    // Display the current state of the hash table
    void display() {
        for (int i = 0; i < capacity; ++i) {
            cout << "Index " << i << ": ";
            if (!table[i]) {
                cout << "None"; // Slot is empty
            } else if (table[i]->isDeleted) {
                cout << "DELETED"; // Slot is marked as deleted
            } else {
                cout << table[i]->key << " -> " << table[i]->value; // Display key-
value pair
            }
            cout << endl;
        }
    }
};

int main() {
    // Create a hash table with initial capacity of 5
    HashTable ht(5);

    // Test cases from the assignment
    ht.insert("Alice", 25);
    cout << "After Inserting ('Alice', 25):" << endl;
    ht.display();

    ht.insert("Bob", 30);
    cout << "\nAfter Inserting ('Bob', 30):" << endl;
    ht.display();

    ht.insert("Charlie", 20);
    cout << "\nAfter Inserting ('Charlie', 20):" << endl;
    ht.display();

    ht.insert("David", 35);
    cout << "\nAfter Inserting ('David', 35):" << endl;
    ht.display();

    ht.insert("Eve", 50); // Causes resizing
    cout << "\nAfter Inserting ('Eve', 50):" << endl;
    ht.display();

    cout << "\nGet ('Charlie'): " << ht.get("Charlie") << endl;

    ht.remove("Bob");
    cout << "\nAfter Deleting ('Bob'):" << endl;
    ht.display();
```

```
    ht.insert("Bob", 40);
    cout << "\nAfter Re-Inserting ('Bob', 40):" << endl;
    ht.display();

    return 0;
}
```