```
// Write a program to implement Huffman Encoding using a greedy strategy.

#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>

using namespace std;

// A Huffman tree node
struct Node {
    char ch;
    int freq;
    Node *left, *right;

    Node(char character, int frequency) {
        ch = character;
        freq = frequency;
        left = right = nullptr;
    }
};

// Comparison object for min-heap priority queue
struct Compare {
    bool operator()(Node* left, Node* right) {
        return left->freq > right->freq;
    }
};

// Function to generate the Huffman codes from the Huffman Tree
void generateCodes(Node* root, string code, unordered_map<char, string> &huffmanCodes) {
```

```
    if (root == nullptr) return;

    // Leaf node; contains a character
    if (!root->left && !root->right) {
        huffmanCodes[root->ch] = code;
    }

    generateCodes(root->left, code + "0", huffmanCodes);
    generateCodes(root->right, code + "1", huffmanCodes);
}

// Main function to build the Huffman Tree and get the Huffman codes
unordered_map<char, string> huffmanEncoding(const unordered_map<char, int> &frequencies) {
    // Priority queue (min-heap) for building the Huffman Tree
    priority_queue<Node*, vector<Node*>, Compare> minHeap;

    // Create a leaf node for each character and add it to the priority queue
    for (auto pair : frequencies) {
        minHeap.push(new Node(pair.first, pair.second));
    }

    // Iterate until only one node remains in the priority queue
    while (minHeap.size() != 1) {
        Node *left = minHeap.top(); minHeap.pop();
        Node *right = minHeap.top(); minHeap.pop();

        // Combine two nodes with lowest frequency
        int combinedFreq = left->freq + right->freq;
        Node *newNode = new Node('\0', combinedFreq); // '\0' represents internal node
        newNode->left = left;
        newNode->right = right;
```

```
        minHeap.push(newNode);
    }

    // Root of the Huffman Tree
    Node* root = minHeap.top();

    // Traverse the Huffman Tree and generate codes
    unordered_map<char, string> huffmanCodes;
    generateCodes(root, "", huffmanCodes);
    return huffmanCodes;
}

int main() {
    // Sample input: frequencies of each character
    unordered_map<char, int> frequencies = {
        {'a', 5}, {'b', 9}, {'c', 12}, {'d', 13}, {'e', 16}, {'f', 45}
    };

    // Generate Huffman codes
    unordered_map<char, string> huffmanCodes = huffmanEncoding(frequencies);

    // Print the Huffman codes
    cout << "Huffman Codes for each character:" << endl;
    for (auto pair : huffmanCodes) {
        cout << pair.first << ": " << pair.second << endl;
    }

    return 0;
}
```

/*

Explanation of the Program

1. **Node Structure**: A `Node` struct is created to represent each node in the Huffman Tree, containing a character, its frequency, and pointers to left and right children.

2. **Min-Heap (Priority Queue)**: A `priority_queue` is used to implement a min-heap, with a custom comparator to prioritize nodes with lower frequencies.

3. **Building the Huffman Tree**:

   - All characters and their frequencies are added to the min-heap.

   - The two nodes with the lowest frequencies are removed from the heap, merged to create a new node with their combined frequency, and this new node is added back to the heap.

   - This process repeats until only one node (the root) remains.

4. **Generating Huffman Codes**: A recursive function `generateCodes` traverses the Huffman Tree to assign binary codes to each character. Left edges are labeled "0" and right edges "1".

5. **Output**: The program outputs Huffman codes for each character based on their frequency.

Complexity Analysis

- **Time Complexity**: $O(n \log n)$, where n is the number of unique characters, due to inserting and removing nodes from the priority queue.

- **Space Complexity**: $O(n)$ for storing the tree nodes and the resulting Huffman codes.

This implementation follows a greedy strategy by always combining the two nodes with the lowest frequencies, which helps minimize the overall cost of encoding.

*/