```
#include <iostream>
#include <unordered_map>
#include <queue>
#include <vector>
#include <fstream>
#include <string>

using namespace std;

// Make sure to update isLeaf appropriately when you combine nodes!
struct Node {
    char ch;
    int freq;
    Node* left;
    Node* right;
    bool isLeaf;

    Node(char ch, int freq) :
            ch(ch), freq(freq), left(nullptr), right(nullptr), isLeaf(true) {}
};

struct Compare {
    bool operator()(Node* l, Node* r) {
        return l->freq > r->freq;
    }
};

/*
 * The function should:
 * 1. Calculate the frequency of each character in the input text.
 * 2. Create a priority queue to store nodes of the Huffman tree, ordered by
frequency.
 * 3. Create a new node for each unique character and insert it into the priority
queue.
 * 4. While there is more than one node in the priority queue:
 *    a. Extract the two nodes with the smallest frequency.
 *    b. Create a new node with these two nodes as children and the sum of their
frequencies as the new frequency (make sure to update isleaf)
 *    c. Insert the new node back into the heap
 * 5. With only one node in the heap you are finished, that is your Huffman tree!
 */
Node* buildHuffmanTree(const string& fileText) {
    // 1: Calculate the frequency of each character in the input text.
    unordered_map<char, int> chars;
    for (char c : fileText)
    {
        chars[c]++;
    }

    // 2: Create a priority queue to store nodes of the Huffman tree, ordered by
frequency.
    priority_queue<Node*, vector<Node*>, Compare> pq;

    // 3: Create a new node for each unique character and insert it into the
priority queue.
    for (const auto& insert : chars) {
        pq.push(new Node(insert.first, insert.second));
    }
```

```
    // 4: While there is more than one node in the priority queue:
    while (pq.size() > 1) {
        // a. Extract the two nodes with the smallest frequency.
        Node* leftNode = pq.top();
        pq.pop();
        Node* rightNode = pq.top();
        pq.pop();

        // b. Create a new node with these two nodes as children and the sum of
their frequencies as the new frequency (Make sure to update isLeaf)
        Node* TreeNode = new Node('\0', leftNode->freq + rightNode->freq);
        TreeNode->left = leftNode;
        TreeNode->right = rightNode;
        TreeNode->isLeaf = false;

        // c. Insert the new node back into the heap.
        pq.push(TreeNode);
    }

    // 5: With only one node in the heap, you are finished, and that is your
Huffman tree!
    return pq.top();
}


/**
 * The function should:
 * 1. Check if the current node is null, if yes then return.
 * 2. If the current node is a leaf, associate the current Huffman code with the
node's character.
 * 3. Recursively call the function for the left child, adding "0" to the current
code.
 * 4. Recursively call the function for the right child, adding "1" to the current
code.
 */
void generateHuffmanCodes(Node* root, const string& str, unordered_map<char,
string>& codes) {
    // 1. Check if the current node is null, if yes then return.
    if (root == nullptr) {
        return;
    }
    // 2. If the current node is a leaf, associate the current Huffman code with
the node's character.
    if (root->isLeaf) {
        codes[root->ch] = str; // Step 2
    }

    // 3: Recursively call the function for the left child, adding "0" to the
current code.
    generateHuffmanCodes(root->left, str + "0", codes);

    // 4: Recursively call the function for the right child, adding "1" to the
current code.
    generateHuffmanCodes(root->right, str + "1", codes);
}

string encode(const string& text, const unordered_map<char, string>& codes) {
    string encoded = "";
    for (char ch : text) {
```

```
        encoded += codes.at(ch);
    }
    return encoded;
}

unordered_map<char, string> huffman(const string& input) {
    Node* root = buildHuffmanTree(input);
    unordered_map<char, string> codes;
    generateHuffmanCodes(root, "", codes);
    return codes;
}




int main() {
    cout << "Choose an option:\n";
    cout << "1: Input a string\n";
    cout << "2: Input a file path\n";
    int choice;
    cin >> choice;

    cin.ignore();

    string input; // either user entered or file contents

    if (choice == 1) {
        cout << "Enter your string: ";
        getline(cin, input);
    } else if (choice == 2) {
        string filePath;
        cout << "Enter the file path: ";
        getline(cin, filePath);
        ifstream file(filePath);
        if (file.is_open()) {
            getline(file, input, '\0');
            file.close();
        } else {
            cout << "Failed to open the file.\n";
            return 1;
        }
    } else {
        cout << "Invalid choice.\n";
        return 1;
    }

    int originalBits = input.size() * 8;
    int originalBytes = input.size();
    cout << "\nOriginal File Size: " << originalBytes << " bytes (" << originalBits
<< " bits)\n";

    unordered_map<char, string> codes = huffman(input);
    string encodedText = encode(input, codes);
    int compressedBits = encodedText.size();
    int compressedBytes = compressedBits / 8 + (compressedBits % 8 != 0);

    cout << "\nCharacter Codes:\n";
    for (auto& pair : codes) {
        cout << pair.first << ": " << pair.second << "\n";
```

```
    }

    cout << "\nCompressed File Size: " << compressedBytes << " bytes (" <<
compressedBits << " bits)\n";

    return 0;
}
```