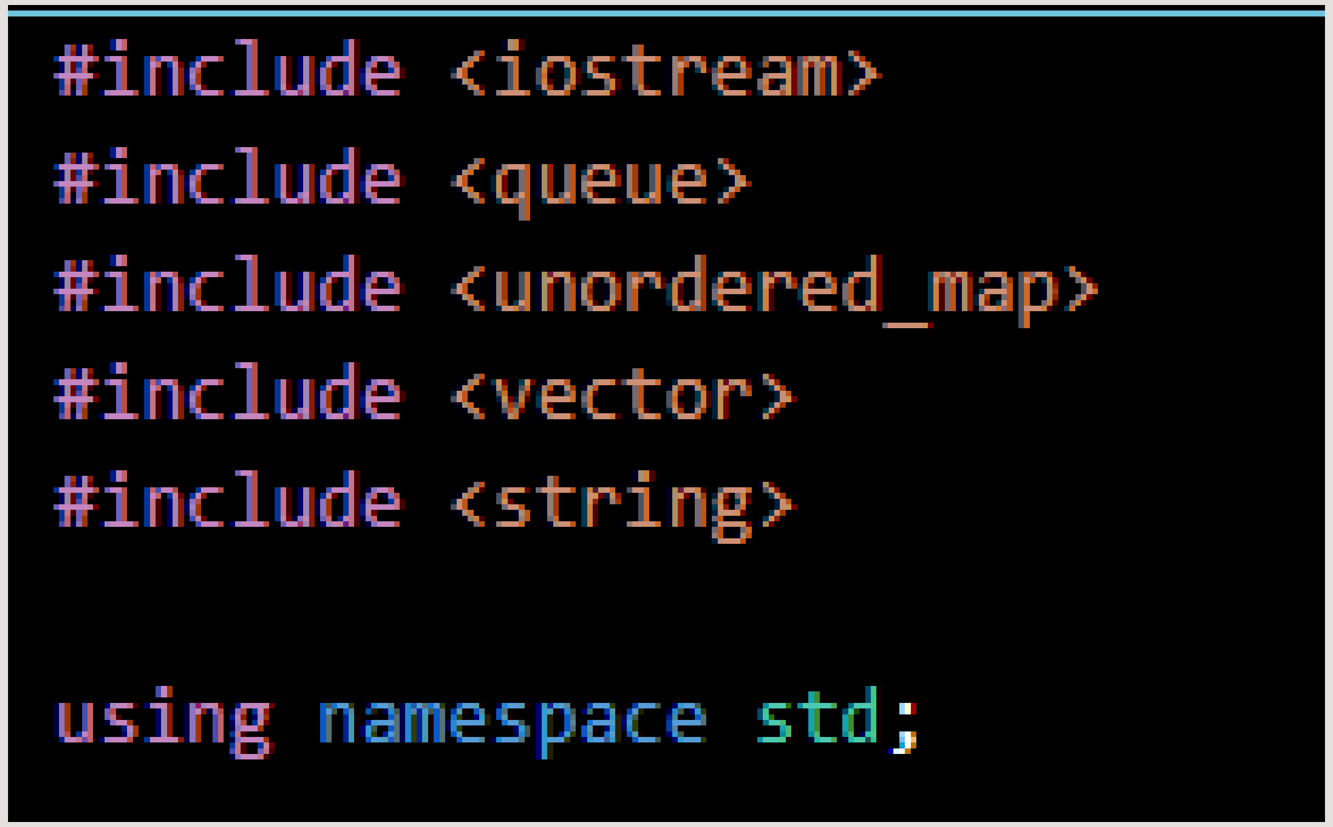
THE HUFFMAN HIGHWAY

MID TERM EVALUATION

Text Decompression using Huffman decoding

We were given coded text (binary numbers) and characters and its frequency and were supposed to decode it and find out what was the original text using huffman highway

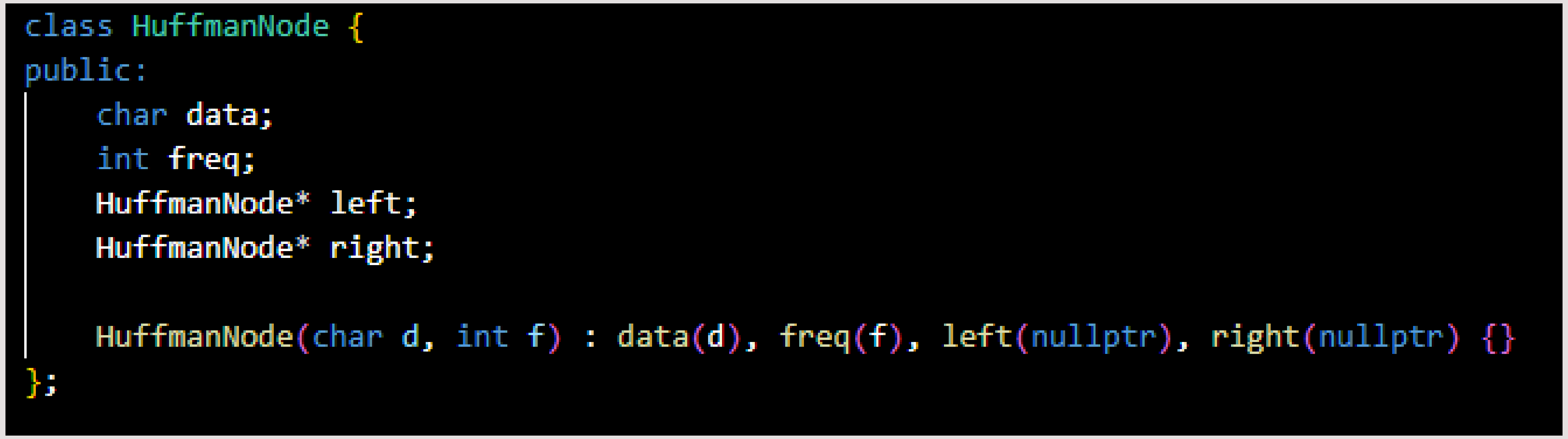
This is the link of my code used to decode the text: https://github.com/Anshika185/The-Huffman-Highway---230160



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

using namespace std;
```

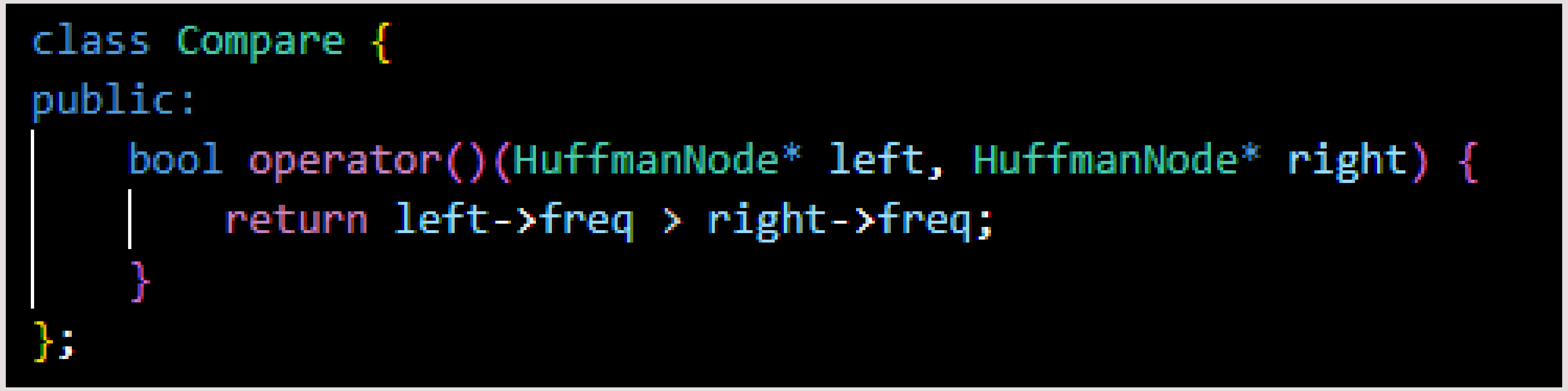
This includes some libraries we need to include for running our code.

The HuffmanNode class is a fundamental part of the Huffman coding algorithm.Each HuffmanNode object represents a node in the Huffman tree.A node contains a character (data), its frequency (freq), and pointers to its left and right children (left and right).

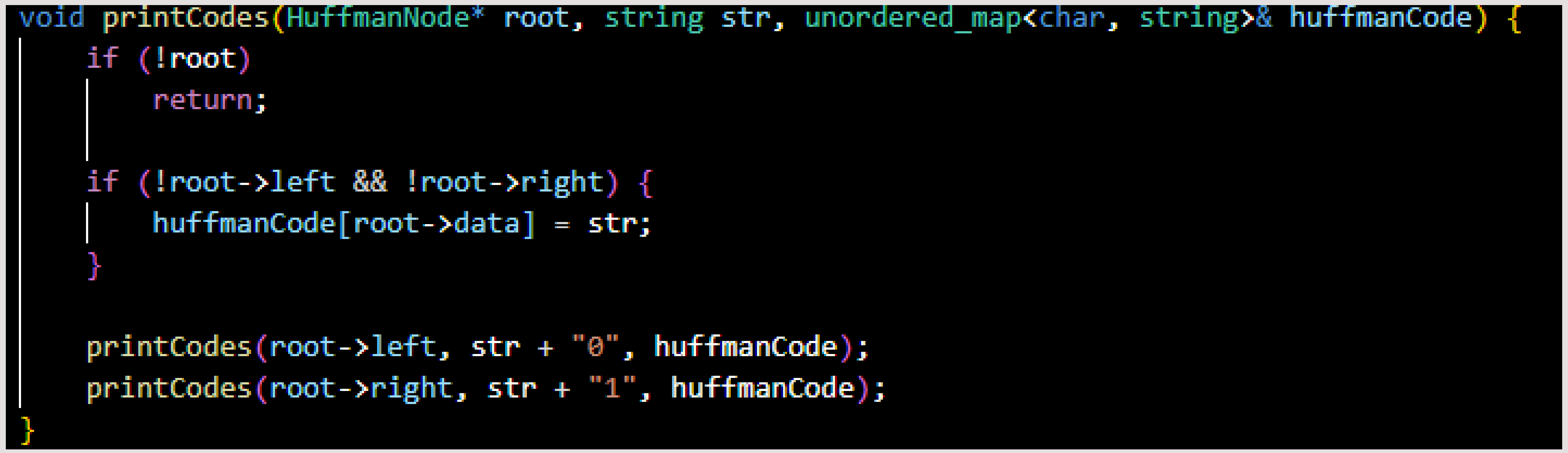


```
class HuffmanNode {
public:
    char data;
    int freq;
    HuffmanNode* left;
    HuffmanNode* right;

    HuffmanNode(char d, int f) : data(d), freq(f), left(nullptr), right(nullptr) {}
};
```

The Compare class in Huffman coding implementation is used to define a custom comparison operator for the priority queue. This operator ensures that the priority queue orders HuffmanNode objects by their frequency in ascending order, which is essential for building the Huffman tree correctly.

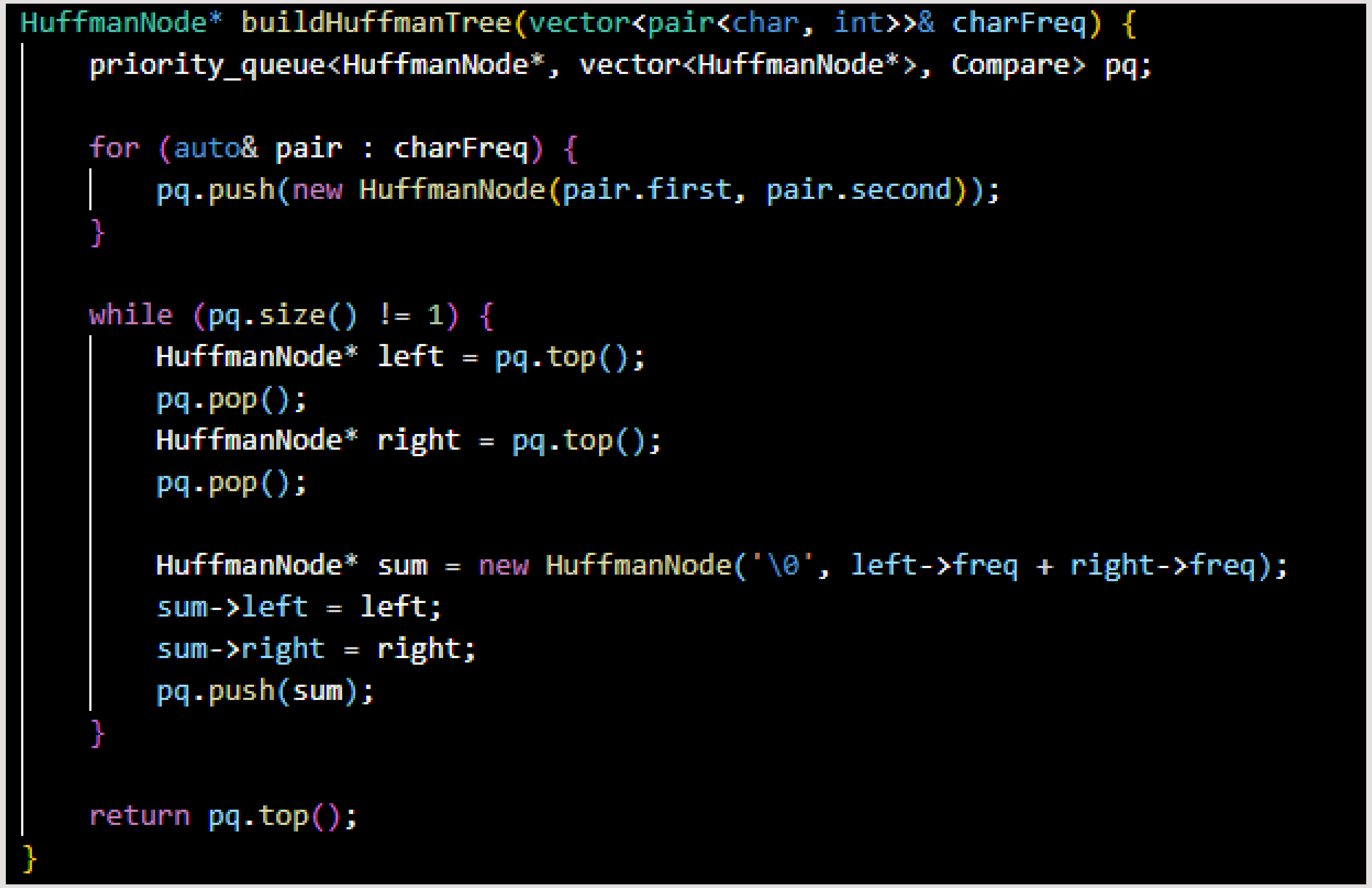


```cpp
class Compare {
public:
    bool operator()(HuffmanNode* left, HuffmanNode* right) {
        return left->freq > right->freq;
    }
};
```



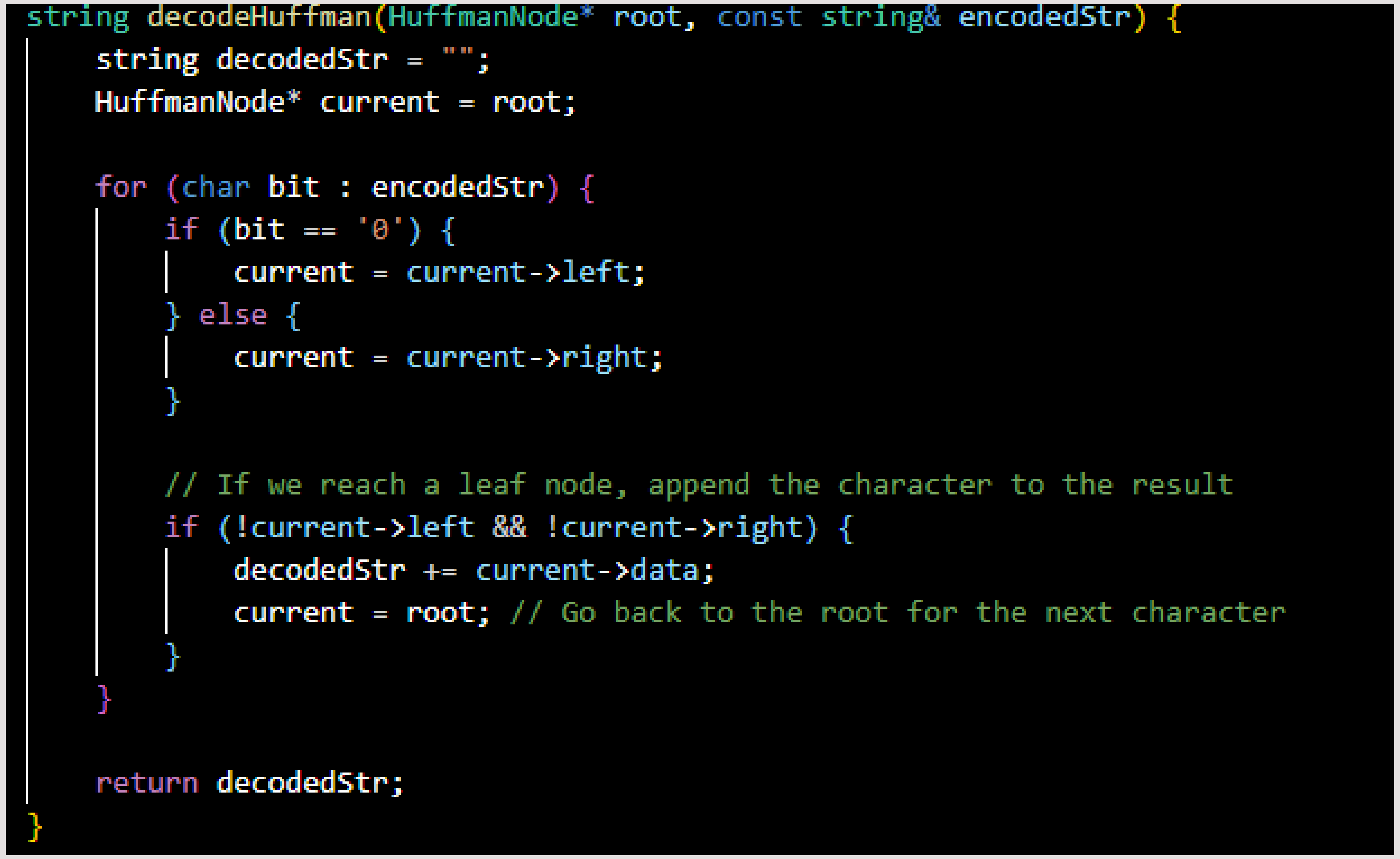
```cpp
void printCodes(HuffmanNode* root, string str, unordered_map<char, string>& huffmanCode) {
    if (!root)
        return;

    if (!root->left && !root->right) {
        huffmanCode[root->data] = str;
    }

    printCodes(root->left, str + "0", huffmanCode);
    printCodes(root->right, str + "1", huffmanCode);
}
```

The printCodes function performs a depth-first traversal of the Huffman tree, constructing the Huffman code for each character based on the path from the root to the leaf node. It appends '0' for left traversals and '1' for right traversals, thus building the unique binary codes for each character and storing them in the provided huffmanCode map. This map can then be used for encoding and decoding purposes in the Huffman coding algorithm.



```
HuffmanNode* buildHuffmanTree(vector<pair<char, int>>& charFreq) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> pq;

    for (auto& pair : charFreq) {
        pq.push(new HuffmanNode(pair.first, pair.second));
    }

    while (pq.size() != 1) {
        HuffmanNode* left = pq.top();
        pq.pop();
        HuffmanNode* right = pq.top();
        pq.pop();

        HuffmanNode* sum = new HuffmanNode('\0', left->freq + right->freq);
        sum->left = left;
        sum->right = right;
        pq.push(sum);
    }

    return pq.top();
}
```

The buildHuffmanTree function constructs the Huffman tree using a priority queue to repeatedly combine the two nodes with the smallest frequencies. This process continues until only one node remains, which serves as the root of the Huffman tree. The function returns this root node, which can then be used to generate Huffman codes or decode encoded data.



```
string decodeHuffman(HuffmanNode* root, const string& encodedStr) {
    string decodedStr = "";
    HuffmanNode* current = root;

    for (char bit : encodedStr) {
        if (bit == '0') {
            current = current->left;
        } else {
            current = current->right;
        }

        // If we reach a leaf node, append the character to the result
        if (!current->left && !current->right) {
            decodedStr += current->data;
            current = root; // Go back to the root for the next character
        }
    }

    return decodedStr;
}
```

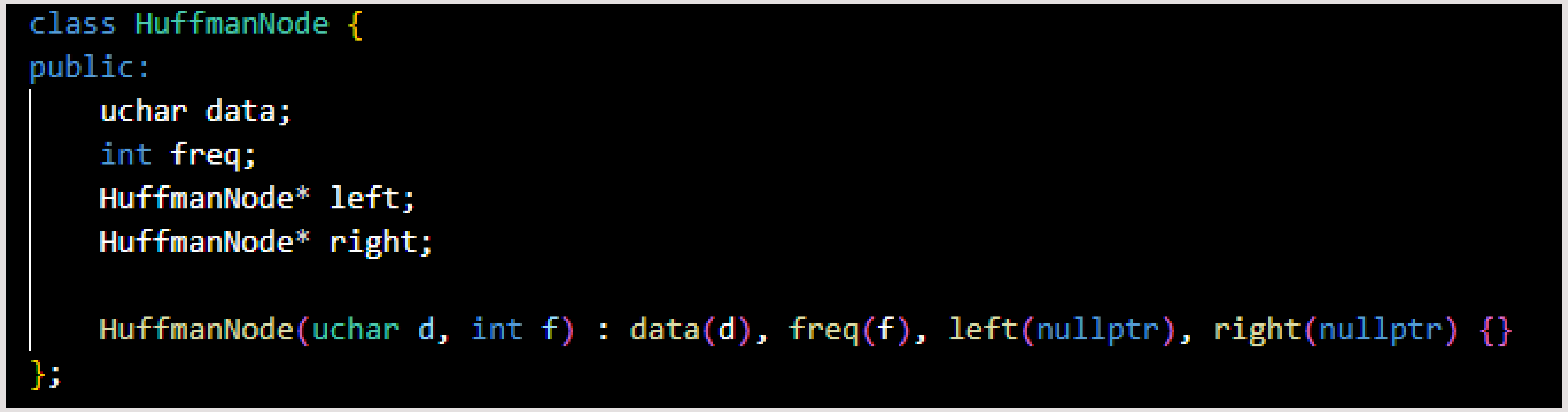
The decodeHuffman function decodes an encoded string by traversing the Huffman tree according to the bits in the encoded string. It appends the character found at each leaf node to the decoded result and then resets to the root to decode the next character. This process continues until the entire encoded string is decoded.

Then we have the main function in which firstly I have given the vector pairs of characters and its frequencies. After that using “buildHuffmanTree” function huffman tree is built using the character frequencies and codes are generated using “printCodes” function.Then we print the Huffman codes for each character stored in the huffmanCode map. It iterates through the map, printing each character and its corresponding Huffman code in a formatted manner. This output is useful for verifying the correctness of the Huffman coding process and for providing a visual representation of the codes generated. Then I input the given encoded string and ask compiler to decode it using “decodeHuffman” function and finally print the decoded string.

The decoded string:

Congratulations, you decoded the text and got the input file. I hope you learned a lot of concepts through this project. Keep up the good work and keep progressing.

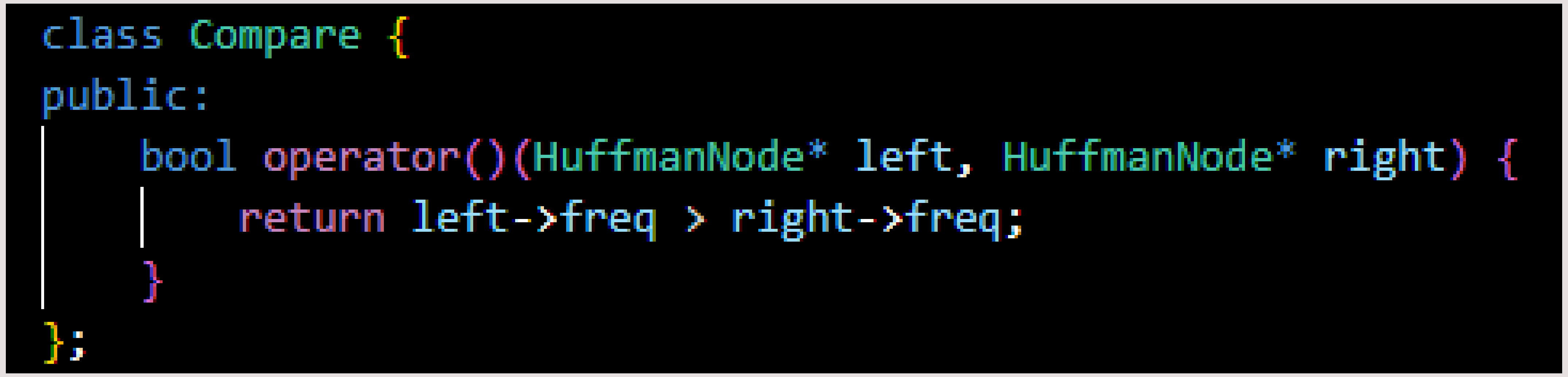
Image Compression using Huffman coding



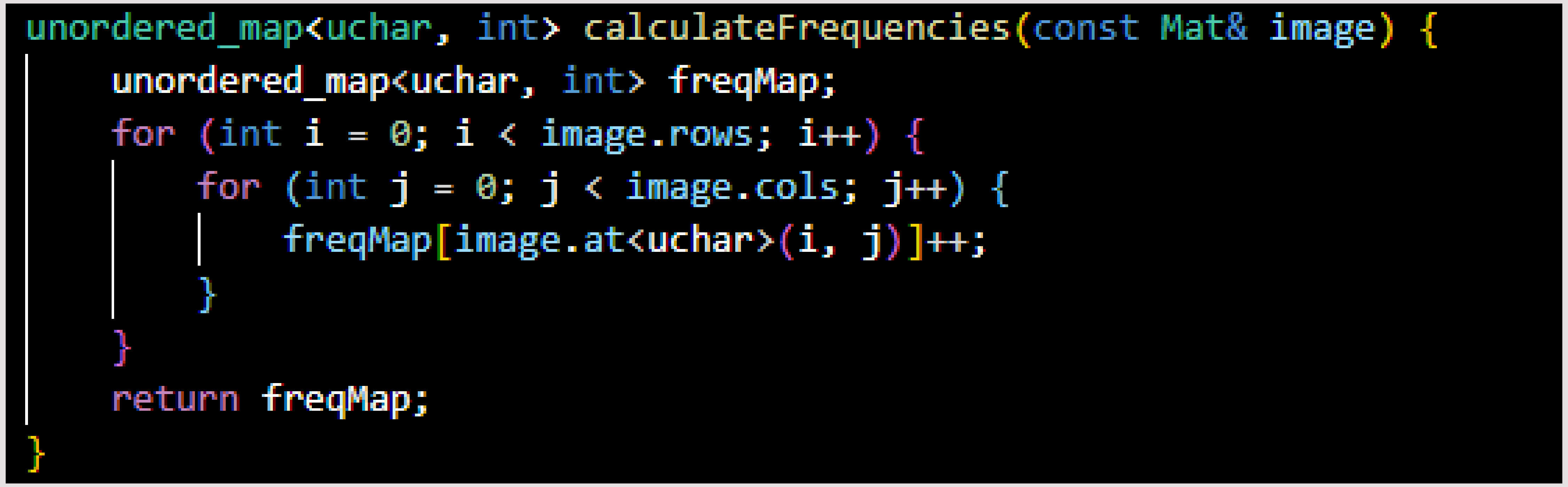
```cpp
class HuffmanNode {
public:
    uchar data;
    int freq;
    HuffmanNode* left;
    HuffmanNode* right;

    HuffmanNode(uchar d, int f) : data(d), freq(f), left(nullptr), right(nullptr) {}
};
```

The HuffmanNode class provides the basic structure to build a Huffman tree, facilitating efficient compression and decompression of data based on frequency analysis. Each node represents a potential symbol (like a pixel value in images) and its associated frequency

The Compare class in Huffman coding implementation is used to define a custom comparison operator for the priority queue. This operator ensures that the priority queue orders HuffmanNode objects by their frequency in ascending order, which is essential for building the Huffman tree correctly.

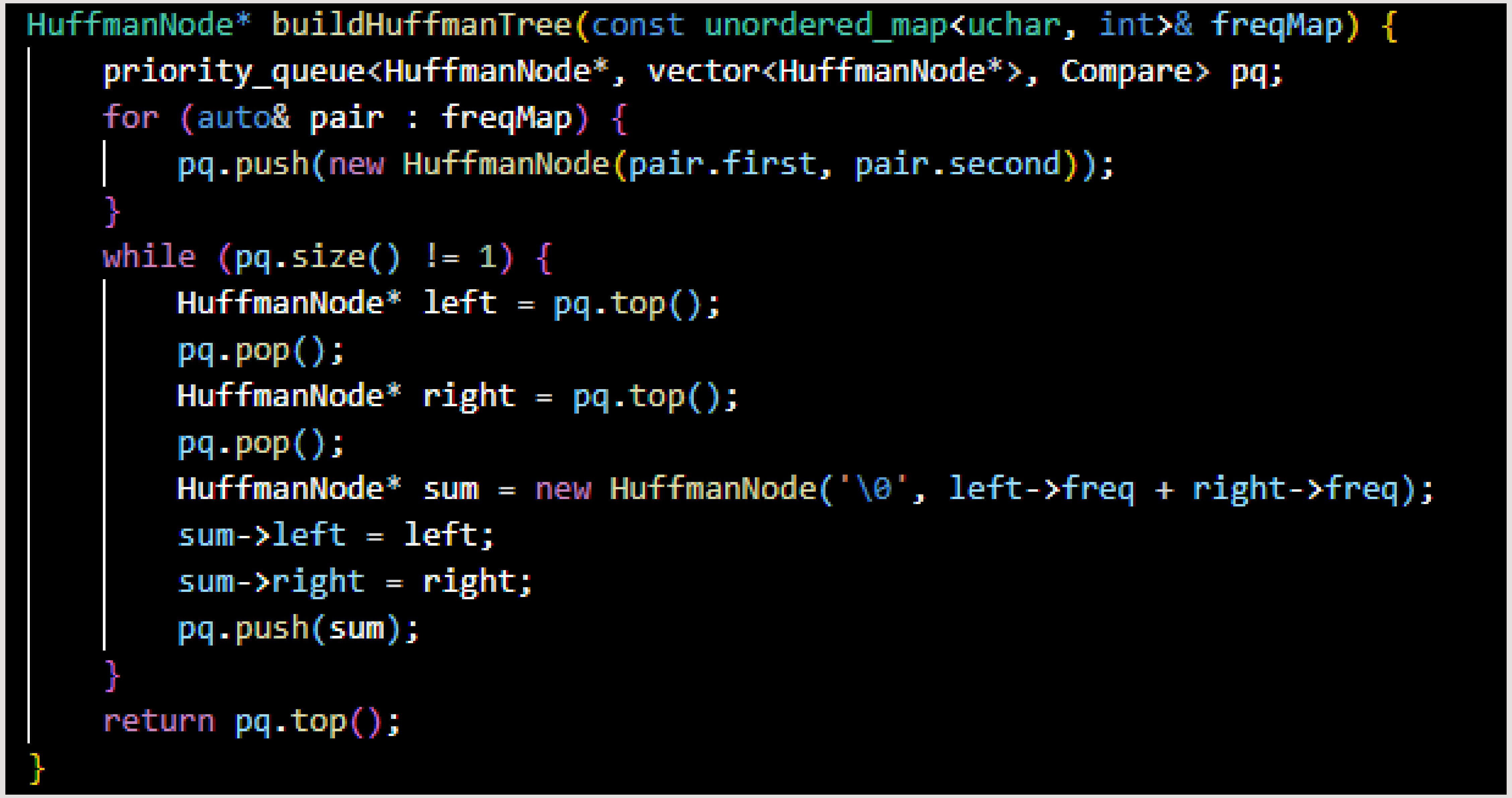


```cpp
class Compare {
public:
    bool operator()(HuffmanNode* left, HuffmanNode* right) {
        return left->freq > right->freq;
    }
};
```



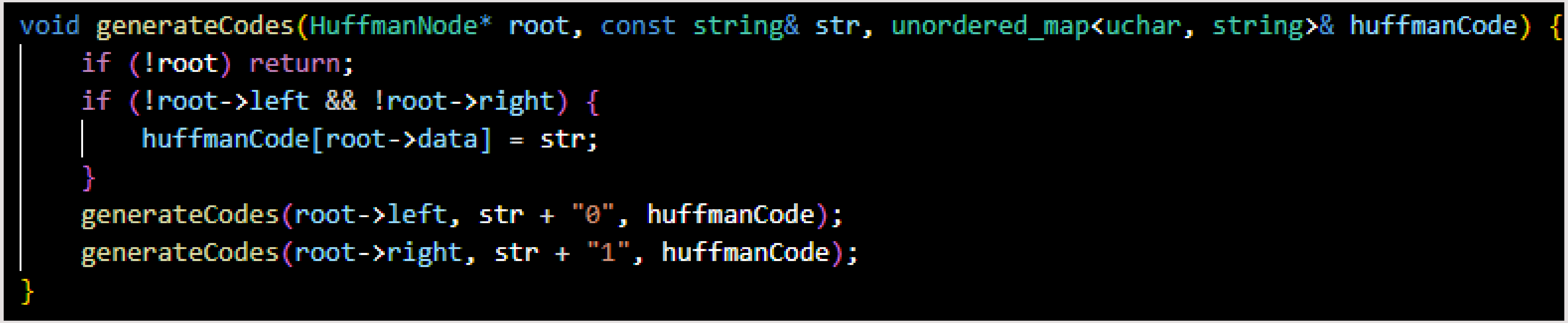
```
unordered_map<uchar, int> calculateFrequencies(const Mat& image) {
    unordered_map<uchar, int> freqMap;
    for (int i = 0; i < image.rows; i++) {
        for (int j = 0; j < image.cols; j++) {
            freqMap[image.at<uchar>(i, j)]++;
        }
    }
    return freqMap;
}
```

The calculateFrequencies function computes and returns a map that associates each grayscale pixel value in an image with its frequency of occurrence. First we initialize an empty unordered map named "freqMap", then used nested loops to iterate over each row and column and access the pixel value t that position.After processing all the pixels in image,the function returns the freqMap which contains frequency of each pixel value in the image.



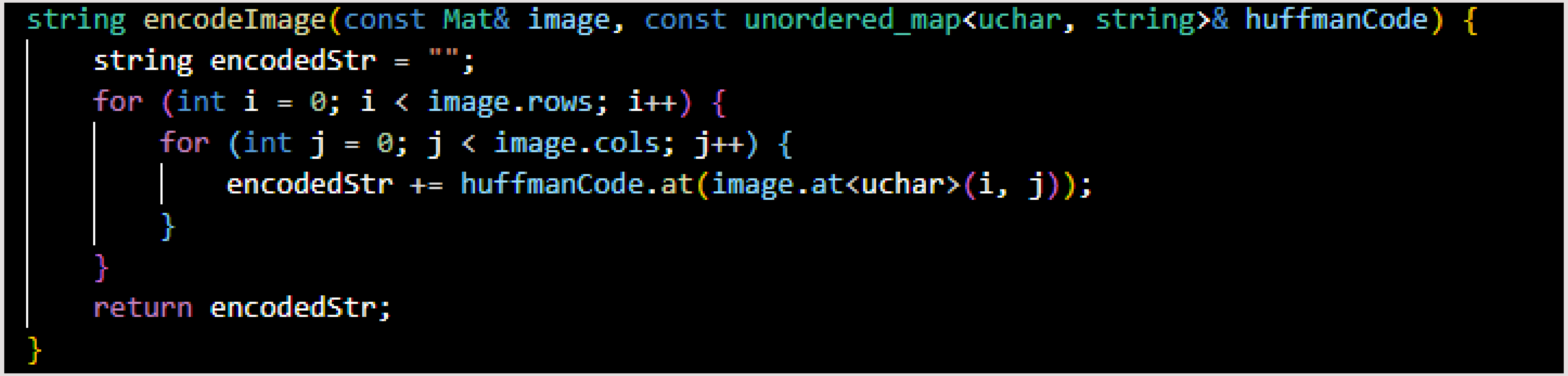
```
HuffmanNode* buildHuffmanTree(const unordered_map<uchar, int>& freqMap) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> pq;
    for (auto& pair : freqMap) {
        pq.push(new HuffmanNode(pair.first, pair.second));
    }
    while (pq.size() != 1) {
        HuffmanNode* left = pq.top();
        pq.pop();
        HuffmanNode* right = pq.top();
        pq.pop();
        HuffmanNode* sum = new HuffmanNode('\0', left->freq + right->freq);
        sum->left = left;
        sum->right = right;
        pq.push(sum);
    }
    return pq.top();
}
```

The buildHuffmanTree function constructs a Huffman tree using a priority queue (pq) based on the frequencies provided in freqMap.After calling buildHuffmanTree with a frequency map (freqMap), we obtain the root of the Huffman tree.Working is similar to the code in the first question.



```
void generateCodes(HuffmanNode* root, const string& str, unordered_map<uchar, string>& huffmanCode) {
    if (!root) return;
    if (!root->left && !root->right) {
        huffmanCode[root->data] = str;
    }
    generateCodes(root->left, str + "0", huffmanCode);
    generateCodes(root->right, str + "1", huffmanCode);
}
```

The generateCodes function is responsible for recursively traversing a Huffman tree (root) and generating Huffman codes for each leaf node (symbol) in the tree.It builds the Huffman codes incrementally (str + "0" for left, str + "1" for right) and stores them in an unordered_map (huffmanCode), mapping each symbol to its respective Huffman code.



```
string encodeImage(const Mat& image, const unordered_map<uchar, string>& huffmanCode) {
    string encodedStr = "";
    for (int i = 0; i < image.rows; i++) {
        for (int j = 0; j < image.cols; j++) {
            encodedStr += huffmanCode.at(image.at<uchar>(i, j));
        }
    }
    return encodedStr;
}
```

The encodeImage function takes an input grayscale image (image) and a map (huffmanCode) that associates each pixel value (represented as uchar) with its corresponding Huffman code (string). Its purpose is to encode the entire image into a single string of binary data based on the Huffman codes assigned to each pixel value.

Then we have the main function where first we load the image then firstly use the function unordered map to create a frequency map of pixels and its frequency. Then using buildHuffmanTree function we create a huffman tree using priority queue. After that we generate binary codes using generateCodes function for each pixel and finally encode the image. Then we print the encoded image string code.

THANK YOU