Huffman-Encoding

Huffman encoding is a lossless data compression technique that uses a greedy algorithm to compress data. In this case study, we'll try to understand how the greedy algorithm works and how it's used in Huffman encoding to compress data.

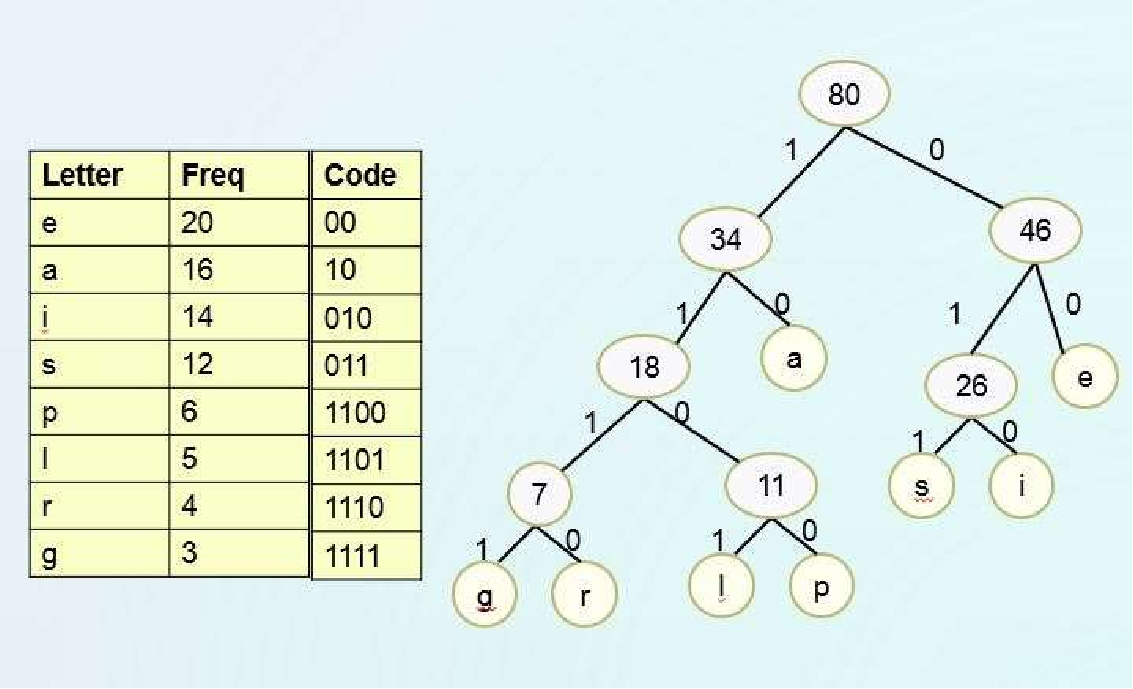
Background: Data compression is a technique used to reduce the size of data to save storage space and improve transmission efficiency. Huffman encoding is one of the most popular data compression techniques that can be used to compress text, images, and other types of data. The Huffman encoding technique assigns shorter codes to frequently occurring symbols and longer codes to less frequent symbols.

The Greedy Algorithm A greedy algorithm is an algorithmic technique that follows the problem-solving method of making the locally optimal choice at each stage. The algorithm chooses the best option at each step without considering the future consequences or alternatives.

How Does Huffman-Encoding work?

In Huffman encoding, the greedy algorithm is used to determine the optimal coding scheme for the symbols. The algorithm works as follows:

1. Calculate the frequency of each symbol in the data to be compressed.
2. Create a binary tree with each symbol as a leaf node and its frequency as the node weight.
3. Combine the two nodes with the lowest weight to create a new internal node with the sum of the weights as its weight.
4. Repeat step 3 until all the nodes are combined into a single tree.
5. Assign '0' to the left branch and '1' to the right branch of each internal node.
6. Traverse the tree to assign unique binary codes to each symbol.

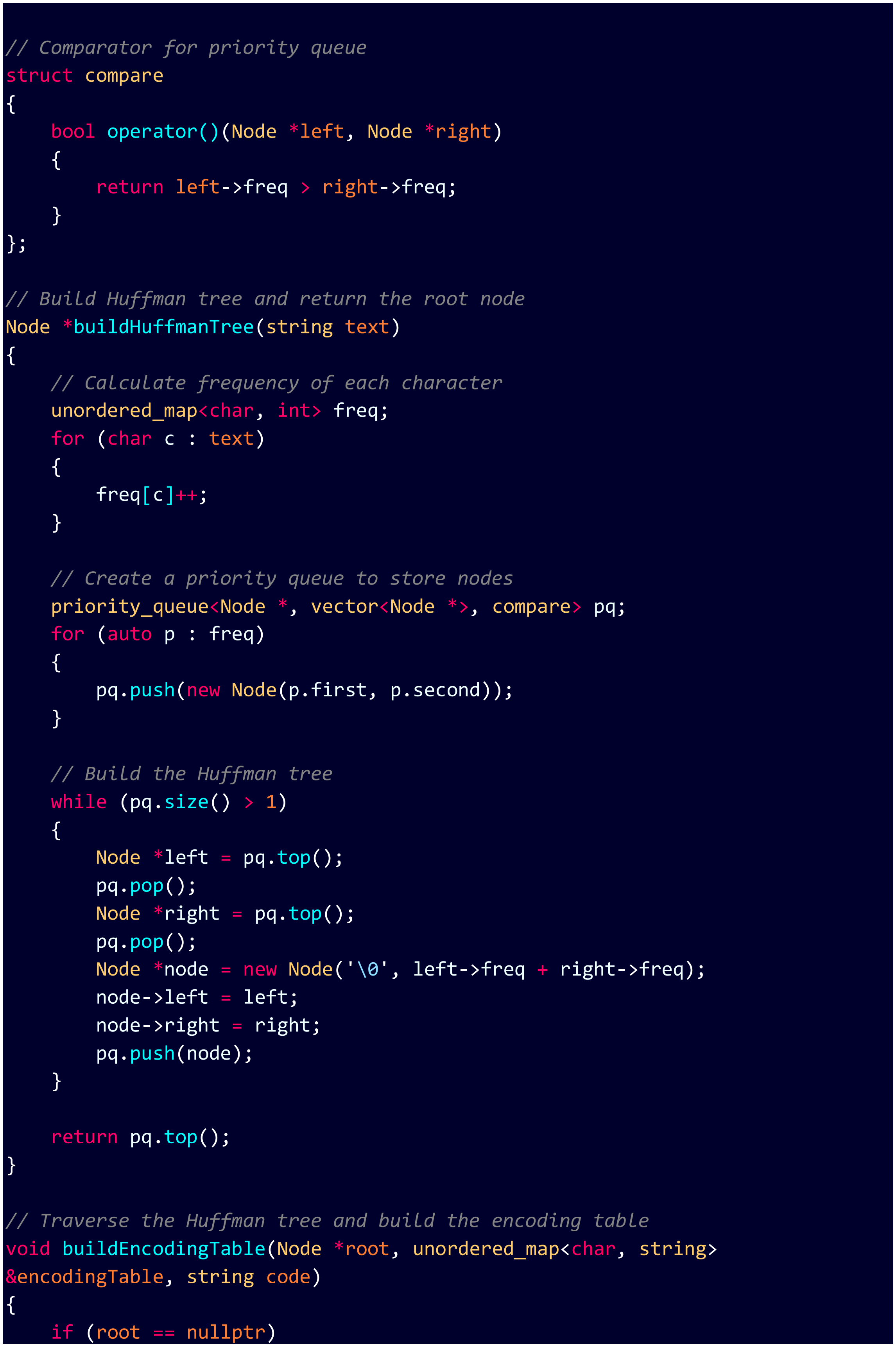


Letter	Freq	Code
e	20	00
a	16	10
i	14	010
s	12	011
p	6	1100
l	5	1101
r	4	1110
g	3	1111

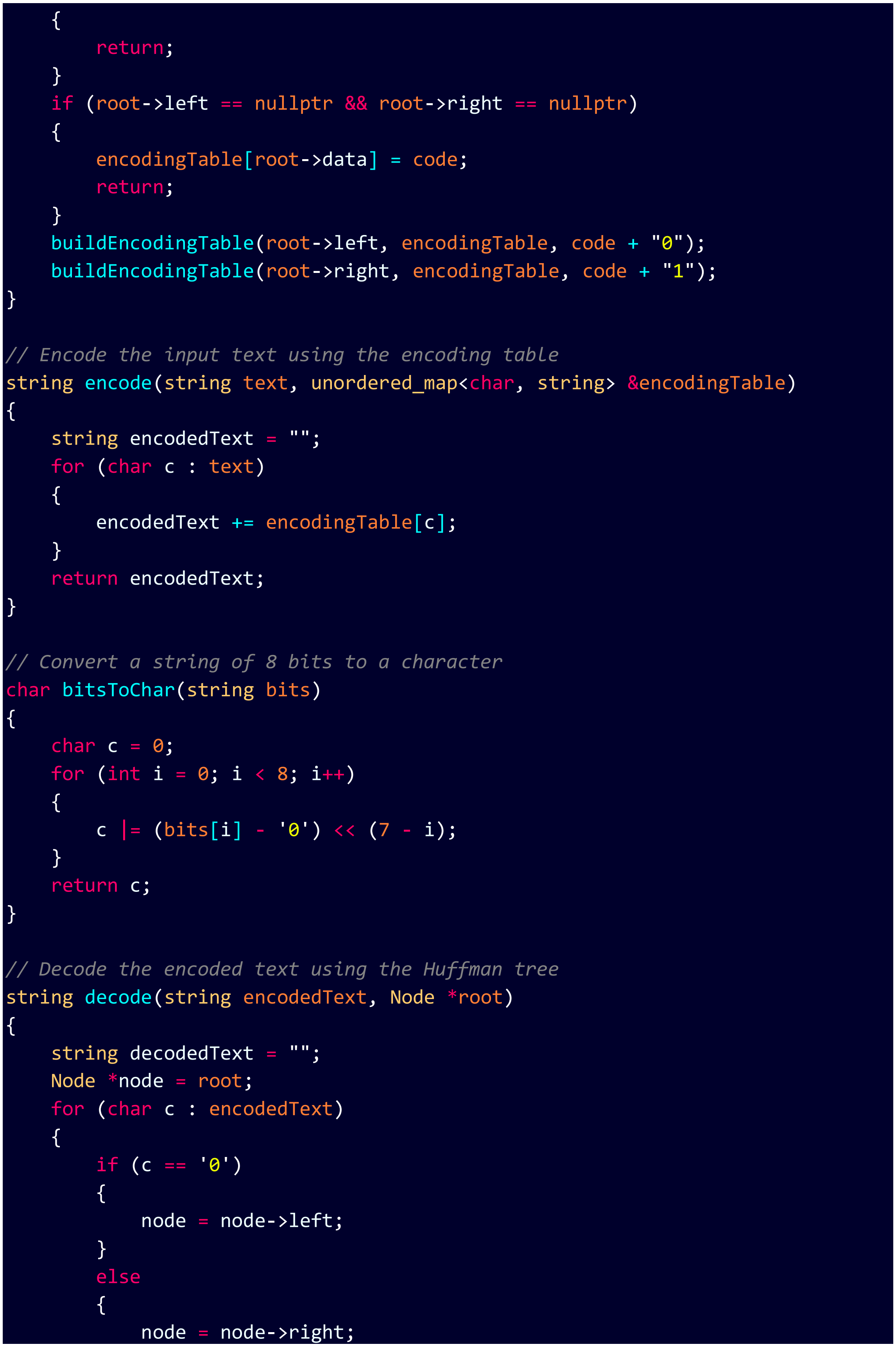
The Huffman Encoding After creating the Huffman tree, we can use it to compress data. Each symbol in the data is replaced by its corresponding Huffman code, resulting in a compressed representation of the original data.

Implementation in C++

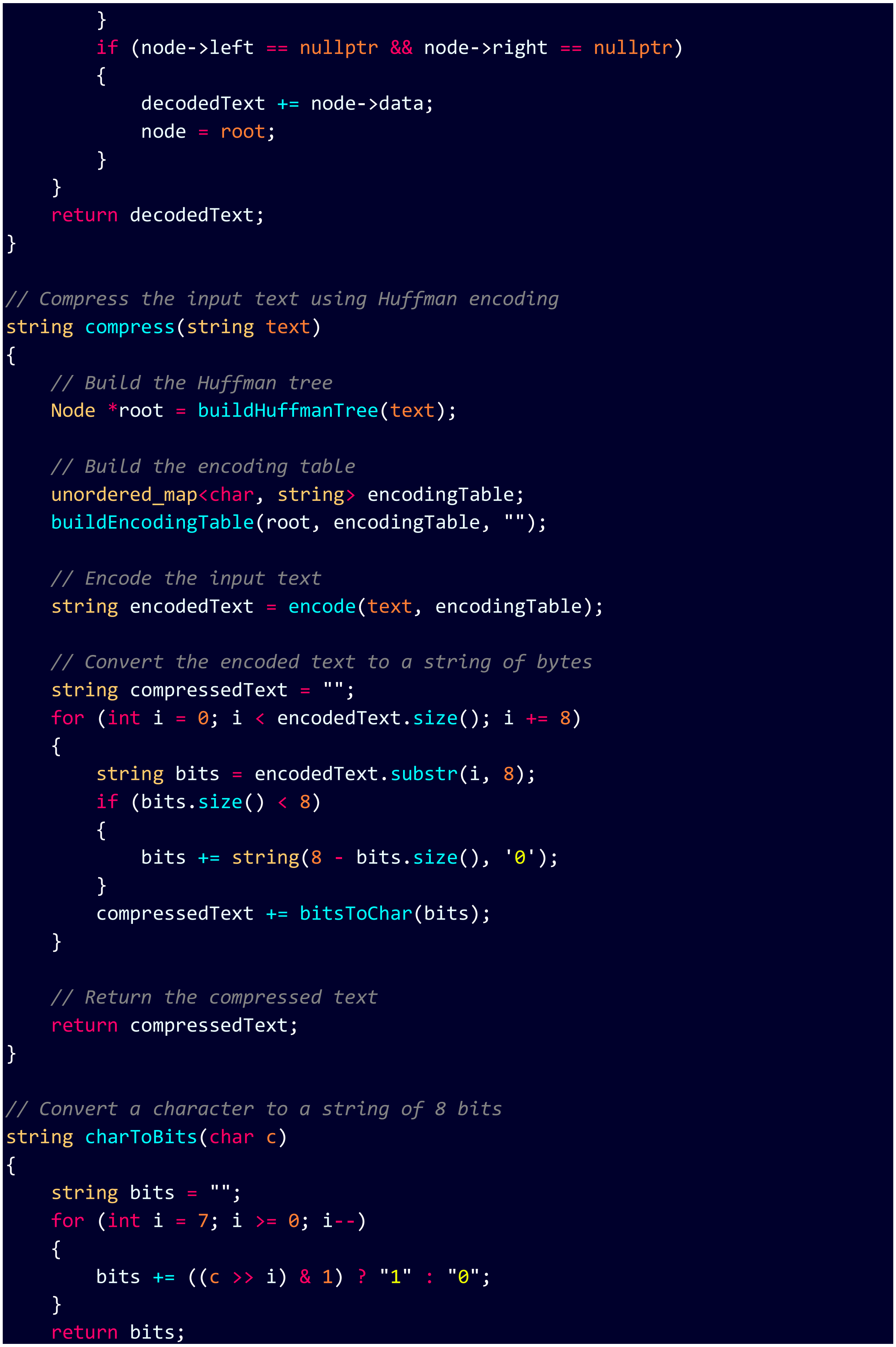
```cpp
#include <iostream>
#include <queue>
#include <unordered_map>
#include <string>
#include <bitset>

using namespace std;

// Huffman tree node
struct Node
{
    char data;
    int freq;
    Node *left;
    Node *right;
    Node(char data, int freq)
    {
        this->data = data;
        this->freq = freq;
        left = right = nullptr;
    }
};
```



```
// Comparator for priority queue
struct compare
{
    bool operator()(Node *left, Node *right)
    {
        return left->freq > right->freq;
    }
};

// Build Huffman tree and return the root node
Node *buildHuffmanTree(string text)
{
    // Calculate frequency of each character
    unordered_map<char, int> freq;
    for (char c : text)
    {
        freq[c]++;
    }

    // Create a priority queue to store nodes
    priority_queue<Node *, vector<Node *>, compare> pq;
    for (auto p : freq)
    {
        pq.push(new Node(p.first, p.second));
    }

    // Build the Huffman tree
    while (pq.size() > 1)
    {
        Node *left = pq.top();
        pq.pop();
        Node *right = pq.top();
        pq.pop();
        Node *node = new Node('\0', left->freq + right->freq);
        node->left = left;
        node->right = right;
        pq.push(node);
    }

    return pq.top();
}

// Traverse the Huffman tree and build the encoding table
void buildEncodingTable(Node *root, unordered_map<char, string>
&encodingTable, string code)
{
    if (root == nullptr)
```



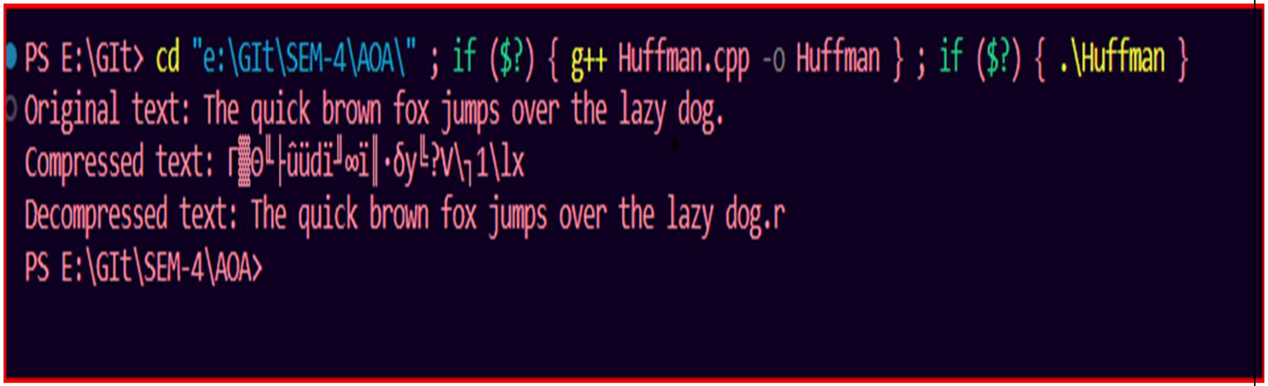
```
    {
        return;
    }
    if (root->left == nullptr && root->right == nullptr)
    {
        encodingTable[root->data] = code;
        return;
    }
    buildEncodingTable(root->left, encodingTable, code + "0");
    buildEncodingTable(root->right, encodingTable, code + "1");
}

// Encode the input text using the encoding table
string encode(string text, unordered_map<char, string> &encodingTable)
{
    string encodedText = "";
    for (char c : text)
    {
        encodedText += encodingTable[c];
    }
    return encodedText;
}

// Convert a string of 8 bits to a character
char bitsToChar(string bits)
{
    char c = 0;
    for (int i = 0; i < 8; i++)
    {
        c |= (bits[i] - '0') << (7 - i);
    }
    return c;
}

// Decode the encoded text using the Huffman tree
string decode(string encodedText, Node *root)
{
    string decodedText = "";
    Node *node = root;
    for (char c : encodedText)
    {
        if (c == '0')
        {
            node = node->left;
        }
        else
        {
            node = node->right;
```



```
        }
        if (node->left == nullptr && node->right == nullptr)
        {
            decodedText += node->data;
            node = root;
        }
    }
    return decodedText;
}

// Compress the input text using Huffman encoding
string compress(string text)
{
    // Build the Huffman tree
    Node *root = buildHuffmanTree(text);

    // Build the encoding table
    unordered_map<char, string> encodingTable;
    buildEncodingTable(root, encodingTable, "");

    // Encode the input text
    string encodedText = encode(text, encodingTable);

    // Convert the encoded text to a string of bytes
    string compressedText = "";
    for (int i = 0; i < encodedText.size(); i += 8)
    {
        string bits = encodedText.substr(i, 8);
        if (bits.size() < 8)
        {
            bits += string(8 - bits.size(), '0');
        }
        compressedText += bitsToChar(bits);
    }

    // Return the compressed text
    return compressedText;
}

// Convert a character to a string of 8 bits
string charToBits(char c)
{
    string bits = "";
    for (int i = 7; i >= 0; i--)
    {
        bits += ((c >> i) & 1) ? "1" : "0";
    }
    return bits;
```

```
}

// Decompress the compressed text using Huffman encoding
string decompress(string compressedText, Node *root)
{
    // Convert the compressed text to a string of bits
    string bits = "";
    for (char c : compressedText)
    {
        bits += charToBits(c);
    }
    // Decode the bits using the Huffman tree
    string decodedText = decode(bits, root);

    // Return the decoded text
    return decodedText;
}

int main()
{
    string text = "The quick brown fox jumps over the lazy dog.";
    // Compress the text
    string compressedText = compress(text);

    // Decompress the text
    string decompressedText = decompress(compressedText,
buildHuffmanTree(text));

    // Print the original text, compressed text, and decompressed text
    cout << "Original text: " << text << endl;
    cout << "Compressed text: " << compressedText << endl;
    cout << "Decompressed text: " << decompressedText << endl;

    return 0;
}
```

Output:



```
PS E:\GIt> cd "e:\GIt\SEM-4\AOA\" ; if ($?) { g++ Huffman.cpp -o Huffman } ; if ($?) { .\Huffman }
Original text: The quick brown fox jumps over the lazy dog.
Compressed text: Γ▓Θ╙├ûüdï╜∞ï║·δy╚?V\┐1\lx
Decompressed text: The quick brown fox jumps over the lazy dog.r
PS E:\GIt\SEM-4\AOA>
```

This code implements Huffman encoding, which is a technique used for lossless data compression. Here's what each function is doing:

1. **buildHuffmanTree**: This function takes a string **text** as input, and returns the root node of the Huffman tree. It first calculates the frequency of each character in the input text using an unordered map. Then, it creates a priority queue to store nodes of the Huffman tree, where each node represents a character and its frequency. The priority queue is sorted in ascending order of frequency, so the nodes with the lowest frequency are at the front of the queue. The function then builds the Huffman tree by repeatedly taking the two nodes with the lowest frequency from the priority queue, creating a new parent node with their combined frequency, and adding it back to the priority queue. This process continues until there is only one node left in the priority queue, which is the root node of the Huffman tree.

2. **buildEncodingTable**: This function takes the root node of a Huffman tree, and recursively traverses the tree to build an encoding table that maps each character to its corresponding Huffman code. The encoding table is stored in an unordered map with characters as keys and strings of 0s and 1s as values. The **code** argument is used to keep track of the current Huffman code as the function traverses the tree.

3. **encode**: This function takes a string **text** and an encoding table as input, and returns the Huffman-encoded version of the input text. It iterates over each character in the input text, and looks up its corresponding Huffman code in the encoding table. It then concatenates these codes together to form the encoded text.

4. **bitsToChar**: This function takes a string of 8 bits as input, and returns the corresponding character. It uses bit manipulation to convert the string of bits to a single character.

5. **decode**: This function takes an encoded text and the root node of a Huffman tree as input, and returns the decoded version of the input text. It iterates over each bit in the encoded text, and uses the Huffman tree to traverse down the tree until it reaches a leaf node. Once it reaches a leaf node, it adds the corresponding character to the decoded text, and resets the current node to the root node.

6. **compress**: This function takes a string **text** as input, and returns the Huffman-encoded and compressed version of the input text. It first builds the Huffman tree and encoding table using the **buildHuffmanTree** and **buildEncodingTable** functions. It then encodes the input text using the **encode** function, and converts the resulting string of bits to a compressed string of characters using the **bitsToChar** function.

7. **charToBits**: This function takes a character as input, and returns a string of 8 bits representing the input character. It uses bit manipulation to extract each bit of the input character and concatenate them together into a string.

8. **decompress**: This function takes a compressed text and the root node of a Huffman tree as input, and returns the decoded version of the compressed text. It first

converts the compressed text to a string of bits using the **charToBits** function. It then uses the **decode** function to decode the string of bits and return the decoded text.

9. **main**: This function demonstrates how to use the other functions to compress and decompress a sample text string. It first compresses the text using the **compress** function, and then decompresses the compressed text using the **decompress** function. It then prints out the original text, compressed text, and decompressed text for comparison.

Huffman encoding is widely used in our day to day life to compress data in a way that takes up less space and reduces the amount of time it takes to transmit the data. Here are a few examples of how Huffman encoding is used:

1. Video and audio compression: Huffman encoding is used in video and audio compression algorithms, such as MPEG, to reduce the size of video and audio files. This reduces the amount of storage space needed for these files and makes it easier to transmit them over the internet.
2. Text messaging: Huffman encoding is used in text messaging applications to reduce the amount of data that needs to be transmitted. For example, when you send a message using WhatsApp, the text is compressed using Huffman encoding before it is sent to the recipient.
3. Image compression: Huffman encoding is used in image compression algorithms, such as JPEG, to reduce the size of image files. This makes it easier to store and transmit images over the internet.
4. Computer programs: Huffman encoding is used in computer programs to compress executable files. This reduces the amount of time it takes to download and install the program, and also reduces the amount of storage space needed to store the program on the computer.
5. Electronic medical records: Huffman encoding is used in electronic medical records (EMR) to compress patient data, such as medical histories and lab results. This reduces the amount of storage space needed to store the data and makes it easier to transmit the data between healthcare providers.

6. streaming services use Huffman encoding to compress video data, which reduces the size of the video file and makes it easier to transmit over the internet.
   For example, when you watch a video on Netflix, the video data is compressed using various techniques, including Huffman encoding. This allows the video to be transmitted to your device more quickly and with less buffering.
   Huffman encoding is also used in the compression of audio files used in music streaming services. For example, when you listen to music on Spotify, the audio files

are compressed using various techniques, including Huffman encoding, to reduce the file size and make it easier to stream the music over the internet.

Overall, Huffman encoding is a useful technique for compressing data in a way that reduces storage space and transmission time, making it a valuable tool for a wide range of applications.

There are several extensions of Huffman encoding, which are designed to improve its performance or to address specific encoding requirements. Here are some of them:

1. Adaptive Huffman Encoding: In the basic Huffman encoding, the frequency of each symbol is fixed, and the Huffman tree is constructed based on the frequency information. However, in some cases, the frequency of symbols may change dynamically during the encoding process. Adaptive Huffman encoding is a technique that allows the Huffman tree to be dynamically updated as new symbols are encountered, to optimize the encoding efficiency.
2. Modified Huffman Encoding: Modified Huffman encoding is an extension of Huffman encoding that allows for efficient encoding of data that contains long runs of repeated symbols. In this technique, the Huffman tree is constructed in such a way that repeated symbols are grouped together and encoded more efficiently.
3. Block Sorting Compression: Block sorting compression is a technique that involves sorting the data into blocks and then encoding the blocks using Huffman encoding. This technique is particularly useful for compressing data that contains repetitive patterns, such as DNA sequences or text.
4. Burrows-Wheeler Transform (BWT): BWT is a technique that reorders the symbols in the input data to create a new sequence that is more amenable to compression using techniques such as Huffman encoding. The BWT creates a permutation of the input data that groups similar symbols together, making it easier to encode them efficiently.
5. Run-Length Encoding (RLE): RLE is a simple compression technique that is often used in combination with Huffman encoding. RLE involves replacing repeated sequences of symbols with a single symbol and a count of the number of times it occurs. This can significantly reduce the size of the input data and make it easier to compress further using Huffman encoding.