CQUPT – University at Albany

Computer Science – International College

**ICSI 403 --- Design and Analysis of Algorithms**

**Project 2 --- Spring 2025**

**Table of Contents**

1. **System documentation -------------------------------------------------------------1**
   * A high-level data flow diagram for the system-----------------------------1
   * A list of routines and their brief descriptions-------------------------------2
   * Implementation details --------------------------------------------------------3
2. **Test documentation------------------------------------------------------------------8**
   * How you tested your program -----------------------------------------------8
   * Testing outputs -----------------------------------------------------------------8
3. **User documentation ----------------------------------------------------------------11**
   * How to run your program ---------------------------------------------------11
   * Describe parameter (if any) -------------------------------------------------12
4. **Source Code--------------------------------------------------------------------------13**
   * Correctness --------------------------------------------------------------------13
   * Programming style -----------------------------------------------------------13

**I.** **System documentation**

**i.** **A high-level data flow diagram for the system**

图示

AI 生成的内容可能不正确。

图示

AI 生成的内容可能不正确。

**ii. A list of routines and their brief descriptions**

| **Layer** | **Function Name** | **Brief Description** |  |
| --- | --- | --- | --- |
| **Bit Manipulation** | void writeBit(std::ofstream& outFile, char bit) | Writes a single bit to the output file stream. |  |
|  | char readBit(std::ifstream& inFile) | Reads a single bit from the input file stream. |  |
|  | void writeByte(std::ofstream& outFile, unsigned char) | Writes a full byte to the output file. |  |
|  | unsigned char readByte(std::ifstream& inFile) | Reads a byte from the input file. |  |
|  | std::string charToBinary(char c) | Converts a character to its binary string form. |  |
|  | char binaryToChar(const std::string& binary) | Converts a binary string back to a character. |  |
| **Framing** | std::string frameData(const std::string& data) | Frames encoded data using SYN markers and length chunks. |  |
|  | std::vector<std::string> deframeData(...) | Extracts encoded data chunks from framed data. |  |
|  | std::string frameEncoding(...) | Frames the Huffman code table using SOH markers. |  |
|  | std::unordered\_map<char, std::string> deframeEncoding(...) | Parses and reconstructs Huffman code table from framed encoding. |  |
| **Application Logic** | void initializeFromFile(std::string filename) | Reads the file and counts character frequency. |  |
|  | void encodeFile(std::string inFile, std::string outFile) | Encodes input file content and writes compressed data. |  |
|  | void decodeFile(std::string inFile, std::string outFile) | Decodes compressed file content and writes original data. |  |
|  | int huffmanCode(std::string args) | Test/demo function for full encode-decode process. |  |
| **Huffman Tree Construction** | void MinHeapify(vector<Node\*>& heap, int i) | Maintains min-heap property at index i. |  |
|  | void BuildMinHeap(vector<Node\*>& heap) | Builds a min-heap from a vector of nodes. |  |
|  | Node\* ExtractMin(vector<Node\*>& heap) | Removes and returns the node with the smallest frequency. |  |
|  | void MinHeapInsert(vector<Node\*>& heap, Node\* node) | Inserts a new node into the min-heap. |  |
|  | Node\* buildHuffmanTree(vector<Node\*>& heap) | Builds the Huffman tree by merging nodes. |  |
|  | void generateCodes(Node\* root, std::string, unordered\_map<char, std::string>&) | Generates binary codes from the Huffman tree. |  |

**iii. Implementation details.**

The project is implemented in **C++** and divided into two core programs: **Transmitter** and **Receiver**. The architecture is modular and layered to promote separation of concerns.

**1. Layered Architecture**

The system is divided into three logical layers:

**Layer 1: Bit-level Operations**

Handles conversion of characters into '0'/'1' ASCII characters and writes them to file as a bit stream.

void **writeBitStreamToFile**(std::string bitstream, std::ofstream &out)

{

    for (char bit : bitstream)

    {

        out.**put**(bit); // bit is either '0' or '1'

    }

}

**Layer 2: Framing**

Adds framing to encoded data and encoding table:

* **Two SYN (ASCII 22) characters for data blocks.**
* **Two SOH (ASCII 1) characters for encoding table blocks.**
* **One byte for length (up to 16 encoded characters).**

outFile.**put**(22);                  // SYN

outFile.**put**(22);                  // SYN

outFile.**put**(encodedStr.**length**()); // length byte

outFile << encodedStr;            // up to 16 characters

**Layer 3: File I/O and Encoding Coordination**

Coordinates reading/writing files, invoking Huffman-related functions, and handling layers above.

**2. Huffman Tree Construction with Min-Heap**

The **Huffman tree** is built from character frequencies using a **Min-Heap**, implemented as a priority queue. Here's how key heap operations are implemented.

**Min-Heap Node Structure**

struct Node

{

    char ch;

    int freq;

    Node \*left;

    Node \*right;

**Node**(char c, int f) : **ch**(c), **freq**(f), **left**(nullptr), **right**(nullptr) {}

};

**Min-Heapify**

Ensures the heap maintains the min-heap property:

void **MinHeapify**(int idx)

{

    int smallest = idx;

    int left = 2 \* idx + 1;

    int right = 2 \* idx + 2;

    if (left < size && heap[left]->freq < heap[smallest]->freq)

        smallest = left;

    if (right < size && heap[right]->freq < heap[smallest]->freq)

        smallest = right;

    if (smallest != idx)

    {

        std::**swap**(heap[smallest], heap[idx]);

**MinHeapify**(smallest);

    }

}

**Build-Min-Heap**

Constructs the heap from an unordered array:

void **BuildMinHeap**()

{

    for (int i = size / 2 - 1; i >= 0; i--)

    {

**MinHeapify**(i);

    }

}

**Min-Heap-Insert**

Inserts a node and reorders heap:

void **MinHeapInsert**(Node \*node)

{

    heap.**push\_back**(node);

    int i = size++;

    while (i && heap[(i - 1) / 2]->freq > heap[i]->freq)

    {

        std::**swap**(heap[i], heap[(i - 1) / 2]);

        i = (i - 1) / 2;

    }

}

**Min-Heap-Extract-Min**

Removes and returns the node with the smallest frequency:

Node \***ExtractMin**()

{

    if (size <= 0)

        return nullptr;

    if (size == 1)

    {

        Node \*min = heap[0];

        heap.**pop\_back**();

        size--;

        return min;

    }

    Node \*min = heap[0];

    heap[0] = heap[size - 1];

    heap.**pop\_back**();

    size--;

**MinHeapify**(0);

    return min;

}

=

**3. Huffman Encoding and Framing**

After building the Huffman tree, codes are generated by traversing it.

void **generateCodes**(Node \*root, std::string code, std::unordered\_map<char, std::string> &table)

{

    if (!root)

        return;

    if (!root->left && !root->right)

    {

        table[root->ch] = code;

    }

**generateCodes**(root->left, code + "0", table);

**generateCodes**(root->right, code + "1", table);

}

Then, the content is encoded and written to a file in framed segments of max 16 characters.

**4. Decoding Process**

* **Receiver reads encoded file and decoding table (transmitted using SOH frames).**
* **Rebuilds Huffman tree from decoding table.**
* **Reads bit stream and traverses the tree to decode characters.**

Sample decoding loop:

Node \*current = root;

for (char bit : bitStream)

{

    if (bit == '0')

        current = current->left;

    else

        current = current->right;

    if (!current->left && !current->right)

    {

        outputFile.**put**(current->ch);

        current = root;

    }

}

**5. Statistics and Output**

* Transmitter prints:
  + Total characters read.
  + Frequency table.
  + Compression ratio.
* Receiver prints:
  + Characters received.
  + File sizes.
  + Confirms successful decompression.

**II Test documentation**

**i.** **How you tested your program**

I execute my program in Vscode, the program structure as follows

图形用户界面, 文本

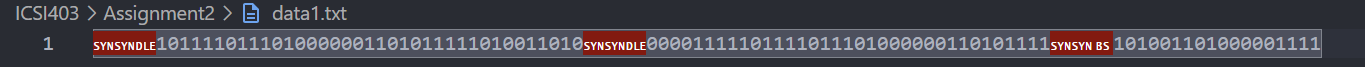
AI 生成的内容可能不正确。

I have four test files to test the program, after getting the results, I compare them with the answer which is calculated by myself.

**ii. Testing outputs**

project2-test1-input.txt图形用户界面, 文本, 应用程序

AI 生成的内容可能不正确。

data1.txt

encoding1.txt

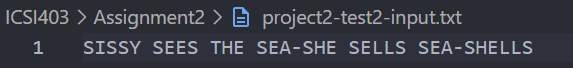
图形用户界面, 应用程序

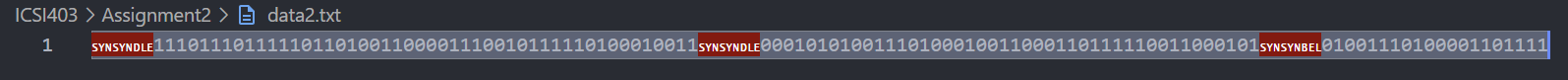
AI 生成的内容可能不正确。

output1.txt图形用户界面, 文本, 网站

AI 生成的内容可能不正确。

------------------------------------------------------------------------------------------------------------

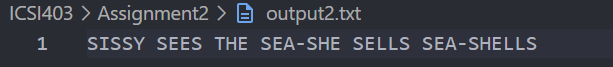
project2-test2-input.txt 

data2.txt 

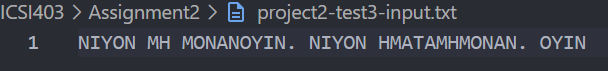
encoding2.txt

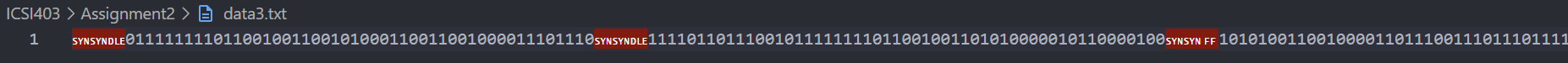
图形用户界面, 文本

AI 生成的内容可能不正确。

output2.txt 

------------------------------------------------------------------------------------------------------------

project2-test3-input.txt 

data3.txt 

encodin3.txt

图形用户界面, 文本, 网站

AI 生成的内容可能不正确。

output3.txt 图形用户界面, 文本

AI 生成的内容可能不正确。

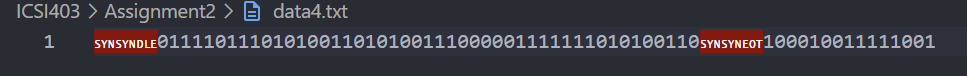
------------------------------------------------------------------------------------------------------------

project2-test4-input.txt

图形用户界面, 文本, 应用程序

AI 生成的内容可能不正确。

data4.txt



encoding4.txt

图形用户界面, 文本, 网站

AI 生成的内容可能不正确。

output4.txt

蓝色的标志

AI 生成的内容可能不正确。

**III. User documentation**

**i. How to run your program**

**1. Run Transmitter**

The Transmitter program is responsible for compressing the original text file using Huffman encoding.

**Steps:**

1. Input the path of the input file (project2-test2-input.txt).
2. Input the path of the data file where the compressed bits will be stored (data2.txt).
3. Input the path of the encoding file where the Huffman codes will be stored (encode2.txt).

文本

AI 生成的内容可能不正确。

**2. Run Receiver**

The Receiver program is responsible for decompressing the encoded file using the stored encoding.

**Steps:**

1. Input the path of the compressed data file (data2.txt).
2. Input the path of the encoding file (encoding2.txt).
3. Input the path of the result output file where the decoded message will be written (output2.txt).

文本

AI 生成的内容可能不正确。

**ii.** **Describe parameter (if any)**

The program requires the user to input file paths:

**Transmitter**

1. **Input file** – the original text file to compress
2. **Data file** – the output file for compressed data
3. **Encoding file** – the output file for Huffman codes

**Receiver**

1. **Data file** – the compressed file to decode
2. **Encoding file** – the Huffman code file
3. **Output file** – the file to save the decoded text

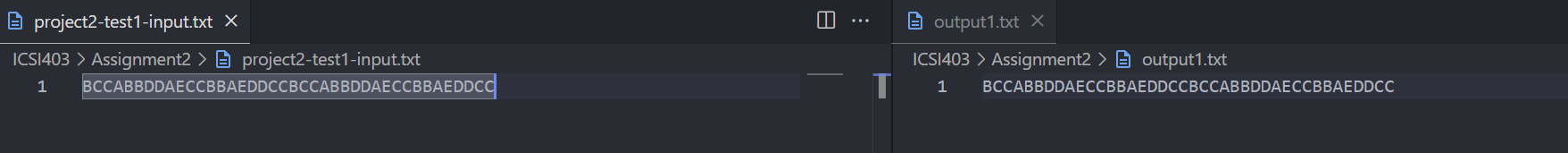
**IV. Source Code**

**Correctness:**

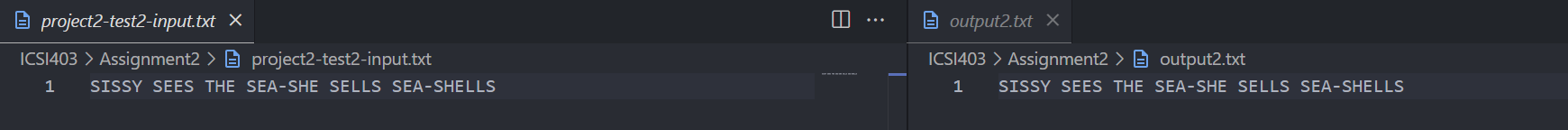
I execute my program to test the four example files, and the results are all **correct**

Layering. Readability.

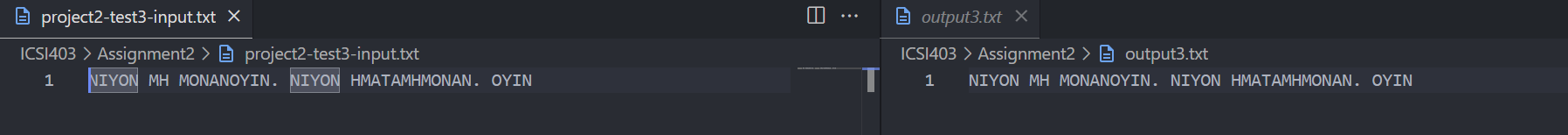
Test1



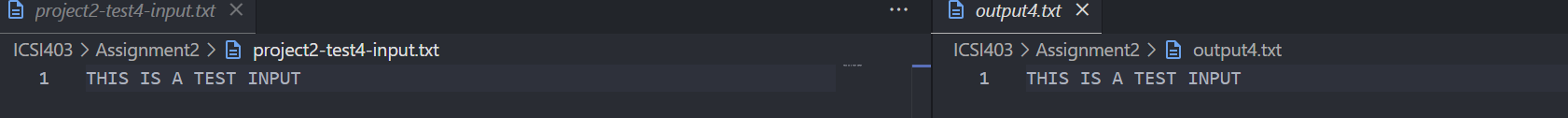
Test2



Test3



Test4



**Programming style:**

**Layering | Readability | Comments | Efficiency** are showing as follows

**Transmitter.cpp**

#include <iostream>

#include <fstream>

#include <map>

#include <vector>

#include <algorithm>

#include <climits>

#include <iomanip>

using namespace std;

// Huffman tree node structure

struct HuffmanNode

{

    char data;

    int freq;

    HuffmanNode \*left, \*right;

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

};

// Min-heap implementation (strictly following required interface names)

class MinHeap

{

    vector<HuffmanNode \*> A;

    int **Parent**(int i) { return (i - 1) / 2; }

    int **Left**(int i) { return 2 \* i + 1; }

    int **Right**(int i) { return 2 \* i + 2; }

public:

    // Maintain heap property for subtree rooted at index i

    void **Min\_Heapify**(vector<HuffmanNode \*> &A, int i)

    {

        int l = **Left**(i);

        int r = **Right**(i);

        int smallest = i;

        if (l < A.**size**() && A**[**l**]**->freq < A**[**i**]**->freq)

            smallest = l;

        if (r < A.**size**() && A**[**r**]**->freq < A**[**smallest**]**->freq)

            smallest = r;

        if (smallest != i)

        {

**swap**(A**[**i**]**, A**[**smallest**]**);

**Min\_Heapify**(A, smallest);

        }

    }

    // Build min-heap from unordered array

    void **Build\_Min\_Heap**(vector<HuffmanNode \*> &A, int n)

    {

        for (int i = n / 2 - 1; i >= 0; i--)

**Min\_Heapify**(A, i);

    }

    // Get minimum element without extraction

    HuffmanNode \***Min\_Heap\_Minimum**(vector<HuffmanNode \*> &A)

    {

        return A.**empty**() ? nullptr : A**[**0**]**;

    }

    // Extract and return minimum element

    HuffmanNode \***Min\_Heap\_Extract\_Min**(vector<HuffmanNode \*> &A)

    {

        if (A.**empty**())

            return nullptr;

        HuffmanNode \*min = A**[**0**]**;

        A**[**0**]** = A.**back**();

        A.**pop\_back**();

**Min\_Heapify**(A, 0);

        return min;

    }

    // Decrease key value at index i

    void **Min\_Heap\_Increase\_Key**(vector<HuffmanNode \*> &A, int i, HuffmanNode \*x)

    {

        if (x->freq < A**[**i**]**->freq)

        {

            A**[**i**]** = x;

            while (i > 0 && A**[Parent**(i)**]**->freq > A**[**i**]**->freq)

            {

**swap**(A**[**i**]**, A**[Parent**(i)**]**);

                i = **Parent**(i);

            }

        }

    }

    // Insert new element into heap

    void **Min\_Heap\_Insert**(vector<HuffmanNode \*> &A, HuffmanNode \*x, int &n)

    {

        A.**push\_back**(new **HuffmanNode**('\0', **INT\_MAX**));

        n = A.**size**();

**Min\_Heap\_Increase\_Key**(A, n - 1, x);

    }

};

// Generate Huffman codes from tree

void **generateCodes**(HuffmanNode \*root, string code, map<char, string> &huffmanCodes)

{

    if (!root)

        return;

    if (!root->left && !root->right) // Leaf node

    {

        huffmanCodes**[**root->data**]** **=** code;

    }

**generateCodes**(root->left, code **+** "0", huffmanCodes);

**generateCodes**(root->right, code **+** "1", huffmanCodes);

}

// Write compressed data file with 16-character blocks

void **writeDataFile**(const string &inputFile, const string &dataFile, const map<char, string> &huffmanCodes)

{

    ifstream **fin**(inputFile, ios::binary);

    ofstream **fout**(dataFile, ios::binary);

    vector<char> **buffer**(16);

    while (fin)

    {

        fin.**read**(buffer.**data**(), 16);

        streamsize count = fin.**gcount**();

        if (count == 0)

            break;

        // Write block header: 2 SYN chars + length

        fout **<<** char(22) **<<** char(22) **<<** char(count);

        // Write encoded data

        for (int i = 0; i < count; i++)

        {

            fout **<<** huffmanCodes.**at**(buffer**[**i**]**);

        }

    }

    fin.**close**();

    fout.**close**();

}

// Write encoding file with alphabetical sorting

void **writeEncodingFile**(const string &encodingFile, map<char, string> &huffmanCodes)

{

    ofstream **fout**(encodingFile, ios::binary);

    // Sort codes alphabetically

    vector<pair<char, string>> **sortedCodes**(huffmanCodes.**begin**(), huffmanCodes.**end**());

**sort**(sortedCodes.**begin**(), sortedCodes.**end**());

    // Write file header: 2 SOH chars + code count

    fout **<<** char(1) **<<** char(1) **<<** char(sortedCodes.**size**());

    // Write encoding table

    for (const auto &pair : sortedCodes)

    {

        fout **<<** pair.first **<<** pair.second;

    }

    fout.**close**();

}

int **main**()

{

    string inputFile, dataFile, encodingFile;

    cout **<<** "Enter input file path: ";

    cin **>>** inputFile;

    cout **<<** "Enter data file output path: ";

    cin **>>** dataFile;

    cout **<<** "Enter encoding file output path: ";

    cin **>>** encodingFile;

    // 1. Read file and calculate character frequencies

    ifstream **fin**(inputFile, ios::binary);

    map<char, int> freq;

    char ch;

    while (fin.**get**(ch))

        freq**[**ch**]**++;

    fin.**close**();

    // 2. Build Huffman tree using min-heap

    MinHeap minHeap;

    vector<HuffmanNode \*> heap;

    int n = 0;

    // Insert all characters into min-heap

    for (auto pair : freq)

    {

        minHeap.**Min\_Heap\_Insert**(heap, new **HuffmanNode**(pair.first, pair.second), n);

    }

    minHeap.**Build\_Min\_Heap**(heap, n);

    // Build Huffman tree by combining nodes

    while (heap.**size**() > 1)

    {

        HuffmanNode \*left = minHeap.**Min\_Heap\_Extract\_Min**(heap);

        HuffmanNode \*right = minHeap.**Min\_Heap\_Extract\_Min**(heap);

        HuffmanNode \*newNode = new **HuffmanNode**('$', left->freq + right->freq);

        newNode->left = left;

        newNode->right = right;

        minHeap.**Min\_Heap\_Insert**(heap, newNode, n);

    }

    HuffmanNode \*root = minHeap.**Min\_Heap\_Extract\_Min**(heap);

    // 3. Generate Huffman codes from tree

    map<char, string> huffmanCodes;

**generateCodes**(root, "", huffmanCodes);

    // 4. Write output files

**writeDataFile**(inputFile, dataFile, huffmanCodes);

**writeEncodingFile**(encodingFile, huffmanCodes);

    // Print statistics

    cout **<<** "\nCompression complete! Statistics:" **<<** **endl**;

    cout **<<** "----------------------------------------" **<<** **endl**;

    cout **<<** **left** **<<** **setw**(15) **<<** "Character" **<<** **setw**(15) **<<** "Frequency" **<<** "Code" **<<** **endl**;

    cout **<<** "----------------------------------------" **<<** **endl**;

    // Sort frequencies alphabetically for display

    vector<pair<char, int>> **sortedFreq**(freq.**begin**(), freq.**end**());

**sort**(sortedFreq.**begin**(), sortedFreq.**end**());

    for (const auto &pair : sortedFreq)

    {

        cout **<<** **setw**(15) **<<** pair.first

**<<** **setw**(15) **<<** pair.second

**<<** huffmanCodes**[**pair.first**]** **<<** **endl**;

    }

    cout **<<** "----------------------------------------" **<<** **endl**;

    cout **<<** "Data file saved to: " **<<** dataFile **<<** **endl**;

    cout **<<** "Encoding file saved to: " **<<** encodingFile **<<** **endl**;

    return 0;

}

Receiver.cpp

#include <iostream>

#include <fstream>

#include <map>

#include <vector>

using namespace std;

// Huffman tree node structure

struct HuffmanNode

{

    char data;

    HuffmanNode \*left, \*right;

**HuffmanNode**(char d) : **data**(d), **left**(nullptr), **right**(nullptr) {}

};

// Function to decode compressed file using Huffman coding

void **decodeFile**(const string &dataFile, const string &encodingFile, const string &outputFile)

{

    // 1. Read encoding file to get code-to-character mapping

    ifstream **encIn**(encodingFile, ios::binary);

    map<string, char> codeToChar;

    // Read file header (2 SOH characters + code count)

    char header[3];

    encIn.**read**(header, 3);

    if (header[0] != 1 || header[1] != 1)

    {

        cerr **<<** "Invalid encoding file header" **<<** **endl**;

        return;

    }

    // Read each character and its corresponding code

    int codeCount = static\_cast<unsigned char>(header[2]);

    for (int i = 0; i < codeCount; i++)

    {

        char ch;

        encIn.**get**(ch);

        string code;

        char bit;

        while (encIn.**get**(bit) && (bit == '0' || bit == '1'))

        {

            code **+=** bit;

        }

        encIn.**unget**();

        codeToChar**[**code**]** = ch;

    }

    encIn.**close**();

    // 2. Rebuild Huffman tree from codes

    HuffmanNode \*root = new **HuffmanNode**('\0');

    for (const auto &pair : codeToChar)

    {

        HuffmanNode \*current = root;

        for (char bit : pair.first)

        {

            if (bit == '0')

            {

                if (!current->left)

                    current->left = new **HuffmanNode**('\0');

                current = current->left;

            }

            else

            {

                if (!current->right)

                    current->right = new **HuffmanNode**('\0');

                current = current->right;

            }

        }

        current->data = pair.second;

    }

    // 3. Read and decode data file

    ifstream **dataIn**(dataFile, ios::binary);

    ofstream **out**(outputFile, ios::binary);

    // Process each data block (each starts with 2 SYN chars + length)

    vector<char> **blockHeader**(3);

    while (dataIn.**read**(blockHeader.**data**(), 3))

    {

        if (blockHeader**[**0**]** != 22 || blockHeader**[**1**]** != 22)

        {

            cerr **<<** "Invalid data block header" **<<** **endl**;

            break;

        }

        // Get number of characters in this block

        int length = static\_cast<unsigned char>(blockHeader**[**2**]**);

        HuffmanNode \*current = root;

        char bit;

        // Decode each character in the block

        for (int i = 0; i < length;)

        {

            dataIn.**get**(bit);

            if (bit == '0')

            {

                current = current->left;

            }

            else if (bit == '1')

            {

                current = current->right;

            }

            // When reaching a leaf node, write the character

            if (!current->left && !current->right)

            {

                out **<<** current->data;

                current = root;

                i++;

            }

        }

    }

    dataIn.**close**();

    out.**close**();

}

int **main**()

{

    string dataFile, encodingFile, outputFile;

    cout **<<** "Enter data file path: ";

    cin **>>** dataFile;

    cout **<<** "Enter encoding file path: ";

    cin **>>** encodingFile;

    cout **<<** "Enter output file path: ";

    cin **>>** outputFile;

**decodeFile**(dataFile, encodingFile, outputFile);

    cout **<<** "Decompression complete! Output file: " **<<** outputFile **<<** **endl**;

    return 0;

}