FINAL PROJECT:

BY

SIR FURQAN HUSSAIN ESSANI

PROJECT NAME:

FILE ZIPPER USING HUFFMAN CODING.

AYESHA YOUSUF (CR-004)
MARYAM KHAN (CR-021)
GHUFRAN RAZA(CR-028)

Report on Project: File Zipper Using Huffman Coding.

1. Introduction:

This code is an implementation of file zipping technique using Huffman coding algorithm in C++. The program covers vector, priority queue, heap memory and Tree. It efficiently manages data structures, file I/O, and dynamic memory allocation to achieve its goal while maintaining clarity and modularity.

2. Project Description:

The project implements Huffman coding for file compression and decompression. Huffman coding is a variable-length prefix coding algorithm used for lossless data compression. The program reads a file, constructs a Huffman tree based on character frequencies, generates Huffman codes, and then compresses or decompresses the input file accordingly. The Huffman tree and encoded data are stored in a binary file during compression and used during decompression. The priority queue is employed to efficiently build the Huffman tree based on character frequencies.

3. Functionalities:

The provided C++ code implements Huffman coding for file compression and decompression. Here are the main functionalities provided by the code.

1. Compression Functionality:

- **Reading Input File:** The program reads the input file specified in the command-line arguments.
- **Frequency Counting:** It counts the frequency of each character in the input file.
- **Min Heap Creation:** Constructs a Min Heap (priority queue) of nodes based on character frequencies.
- **Huffman Tree Construction:** Builds a Huffman tree using the Min Heap.
- **Huffman Code Generation:** Traverses the Huffman tree to generate Huffman codes for each character.

- **Saving Metadata (Huffman Tree):** Saves metadata of the Huffman tree (character data and codes) in the compressed file.
- **Encoding Characters:** Encodes each character in the input file using the generated Huffman codes.
- **Saving Encoded File:** Saves the encoded file, including the Huffman tree metadata and encoded characters.

2. De-compression Functionalities:

- **Reading Input File:** Reads the compressed input file specified in the command-line arguments.
- **Reconstructing Huffman Tree:** Reconstructs the Huffman tree based on the metadata stored in the compressed file.
- **Decoding Characters:** Decodes the Huffman-coded characters in the compressed file using the reconstructed Huffman tree.
- **Saving Decoded File:** Saves the decoded file, obtaining the original content from the Huffman-coded characters.

3. Additional Functionalities:

- **Binary-to-Decimal and Decimal-to-Binary Conversion:** Provides functions for converting binary strings to decimal values and vice versa.
- **Error Handling:** Checks if the correct number of command-line arguments is provided. Outputs an error message if the required files are not detected.

Overall:

The code provides a complete implementation of Huffman coding, including the construction of Huffman trees, generation of Huffman codes, and encoding/decoding of files.
The functionalities are encapsulated within the huffman class, promoting modularity and code organization.
The error handling ensures that the program gracefully handles incorrect usage and missing files.
The code aims to achieve efficient and lossless compression and decompression of files using Huffman coding.

4. Data Structures Used in the Code:

1. Vector:

- **Purpose:** Used to store a collection of nodes representing characters' ASCII values, each initialized with zero frequencies.
- **Reason:** Provides a dynamic array-like structure to easily access and manipulate individual nodes for frequency counting.

2. Priority Queue:

- **Purpose:** Used to maintain a Min Heap of nodes based on their frequencies during Huffman tree construction.
- **Reason:** Ensures that the node with the lowest frequency is always at the top of the heap, facilitating efficient extraction of nodes with the lowest frequencies.

3. Heap Memory:

- **Purpose:** Dynamically allocates memory for Node objects.
- **Reason:** Allows the creation and manipulation of nodes dynamically during the construction of the Huffman tree. Dynamic memory allocation is essential for managing nodes with varying frequencies efficiently.

4. Huffman Tree:

- **Purpose:** Used to construct a tree which later helps in assigning codes for each node.
- **Reason:** The primary purpose of the Huffman tree is to represent the encoding scheme for characters in the input data, where characters with higher frequencies are assigned shorter binary codes, and characters with lower frequencies are assigned longer binary codes.

The chosen data structures in the code are selected to efficiently manage and manipulate the information required for Huffman coding, including character frequencies, Huffman tree nodes, and input/output files. The use of these data structures contributes to the clarity, modularity, and efficiency of the implementation.

5. Contributions:

1. Ayesha Yousuf:

- Present the main idea.
- Implement the compression logic.
- Helped in debugging.
- Create a filling logic.
- Presented the idea of conditional compilation and worked on it.
- Project report.

2. Maryam Khan:

- Implement the de-compression logic.
- Designed a logic for Huffman class.
- Focus on maintaining the other functionalities.
- Idea for the features.
- Project planning.
- Debugging.
- Contributed in Error Handling.

3. Ghufran Raza:

- Idea for the features.
- Participated in overall implementation.
- Including necessary libraries.
- Testing of the code.
- Documentation and comments.

6. Code:

```cpp
//Header Guards to prevent header files from being included multiple times
#ifndef HUFFMAN_HPP
#define HUFFMAN_HPP
#include <string>
#include <vector>
#include <queue>
#include <fstream>
using namespace std;

//Defining Huffman Tree Node
struct Node {
    char data;
    unsigned freq;
    string code;
    Node *left, *right;

    Node() {
        left = right = NULL;
    }
};

class huffman {
    private:
        vector <Node*> arr;

        fstream inFile, outFile;
```

```cpp
        fstream inFile, outFile;

        string inFileName, outFileName;

        Node *root;

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

        priority_queue <Node*, vector<Node*>, Compare> minHeap;

        //Initializing a vector of tree nodes representing character's ascii value and initializing its frequency with
        void createArr();

        //Traversing the constructed tree to generate huffman codes of each present character
        void traverse(Node*, string);

        //Function to convert binary string to its equivalent decimal value
        int binToDec(string);
```

```cpp
        //Function to convert binary string to its equivalent decimal value
        int binToDec(string);

        //Function to convert a decimal number to its equivalent binary string
        string decToBin(int);

        //Reconstructing the Huffman tree while Decoding the file
        void buildTree(char, string&);

        //Creating Min Heap of Nodes by frequency of characters in the input file
        void createMinHeap();

        //Constructing the Huffman tree
        void createTree();

        //Generating Huffman codes
        void createCodes();

        //Saving Huffman Encoded File
        void saveEncodedFile();

        //Saving Decoded File to obtain the original File
        void saveDecodedFile();

        //Reading the file to reconstruct the Huffman tree
        void getTree();
```

```cpp
        //Saving Huffman Encoded File
        void saveEncodedFile();

        //Saving Decoded File to obtain the original File
        void saveDecodedFile();

        //Reading the file to reconstruct the Huffman tree
        void getTree();

    public:
        //Constructor
        huffman(string inFileName, string outFileName)
        {
            this->inFileName = inFileName;
            this->outFileName = outFileName;
            createArr();
        }
        //Compressing input file
        void compress();
        //Decompressing input file
        void decompress();
};
#endif
```

```cpp
#include "huffman.hpp"

void huffman::createArr() {
    for (int i = 0; i < 128; i++) {
        arr.push_back(new Node());
        arr[i]->data = i;
        arr[i]->freq = 0;
    }
}

void huffman::traverse(Node* r, string str) {
    if (r->left == NULL && r->right == NULL) {
        r->code = str;
        return;
    }

    traverse(r->left, str + '0');
    traverse(r->right, str + '1');
}

int huffman::binToDec(string inStr) {
    int res = 0;
    for (auto c : inStr) {
        res = res * 2 + c - '0';
    }
    return res;
```

```cpp
}

string huffman::decToBin(int inNum) {
    string temp = "", res = "";
    while (inNum > 0) {
        temp += (inNum % 2 + '0');
        inNum /= 2;
    }
    res.append(8 - temp.length(), '0');
    for (int i = temp.length() - 1; i >= 0; i--) {
        res += temp[i];
    }
    return res;
}

void huffman::buildTree(char a_code, string& path) {
    Node* curr = root;
    for (int i = 0; i < path.length(); i++) {
        if (path[i] == '0') {
            if (curr->left == NULL) {
                curr->left = new Node();
            }
            curr = curr->left;
        }
        else if (path[i] == '1') {
```

File Edit Selection View Go ··· DSAinC++

huffman.cpp

C: > Users > CPT > Desktop > project > huffman.cpp > createMinHeap()

```cpp
        else if (path[i] == '1' ) {
            if (curr->right == NULL) {
                curr->right = new Node();
            }
            curr = curr->right;
        }
    }
    curr->data = a_code;
}

void huffman::createMinHeap() {
    char id;
    inFile.open(inFileName, ios::in);
    inFile.get(id);
    //Incrementing frequency of characters that appear in the input file
    while (!inFile.eof()) {
        arr[id]->freq++;
        inFile.get(id);
    }
    inFile.close();
    //Pushing the Nodes which appear in the file into the priority queue (Min Heap)
    for (int i = 0; i < 128; i++) {
        if (arr[i]->freq > 0) {
            minHeap.push(arr[i]);
        }
    }
```

⊗ 0 ⚠ 0 DSAinC++ Debug 0 Ln 75, Col 10 Spaces: 4 UTF-8 LF {} C++ windows-gcc-x64

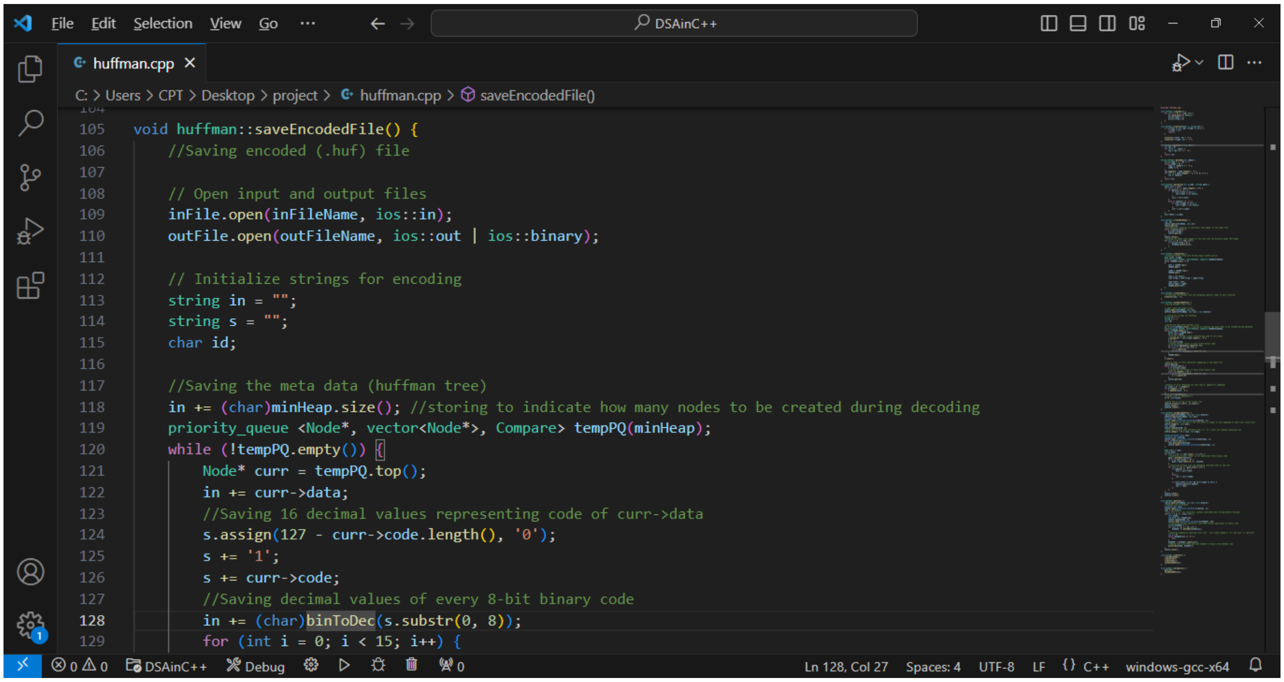
File Edit Selection View Go ··· DSAinC++

huffman.cpp

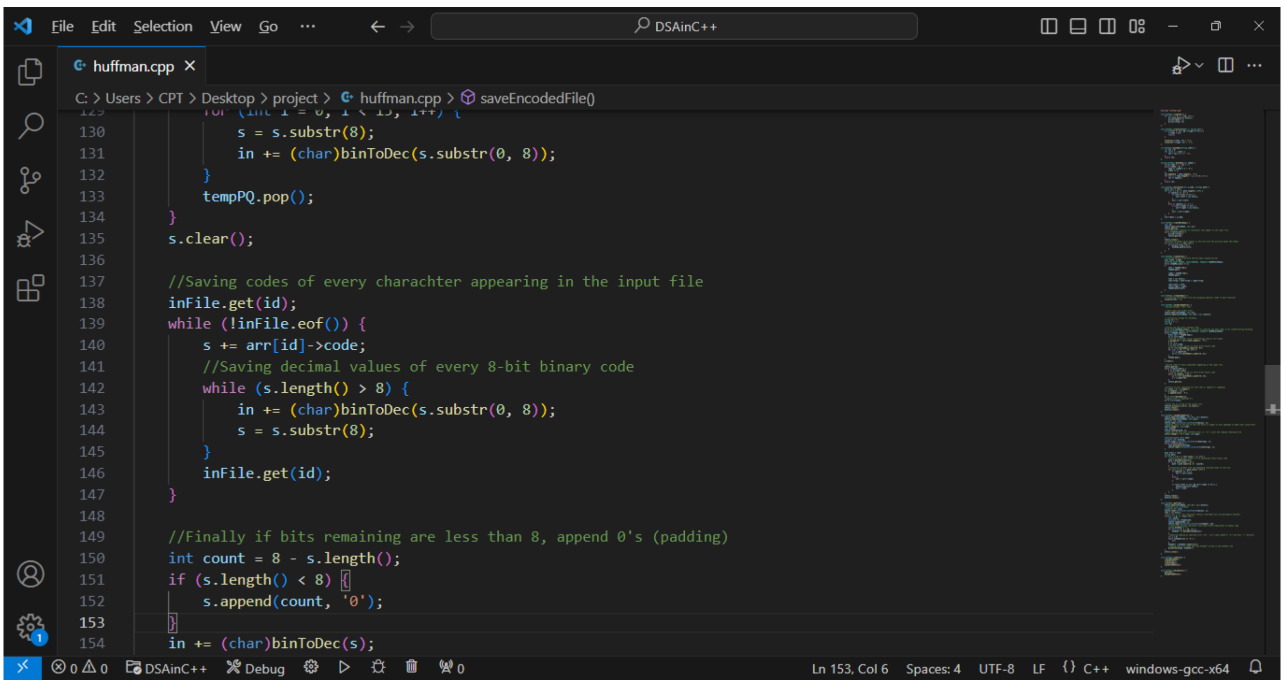
C: > Users > CPT > Desktop > project > huffman.cpp > createCodes()

```cpp
}

void huffman::createTree() {
    //Creating Huffman Tree with the Min Heap created earlier
    Node *left, *right;
    priority_queue <Node*, vector<Node*>, Compare> tempPQ(minHeap);
    while (tempPQ.size() != 1)
    {
        left = tempPQ.top();
        tempPQ.pop();

        right = tempPQ.top();
        tempPQ.pop();

        root = new Node();
        root->freq = left->freq + right->freq;

        root->left = left;
        root->right = right;
        tempPQ.push(root);
    }
}

void huffman::createCodes() {
    //Traversing the Huffman Tree and assigning specific codes to each character
    traverse(root, "");
```

⊗ 0 ⚠ 0 DSAinC++ Debug 0 Ln 101, Col 23 Spaces: 4 UTF-8 LF {} C++ windows-gcc-x64



```cpp
void huffman::saveEncodedFile() {
    //Saving encoded (.huf) file

    // Open input and output files
    inFile.open(inFileName, ios::in);
    outFile.open(outFileName, ios::out | ios::binary);

    // Initialize strings for encoding
    string in = "";
    string s = "";
    char id;

    //Saving the meta data (huffman tree)
    in += (char)minHeap.size(); //storing to indicate how many nodes to be created during decoding
    priority_queue <Node*, vector<Node*>, Compare> tempPQ(minHeap);
    while (!tempPQ.empty()) {
        Node* curr = tempPQ.top();
        in += curr->data;
        //Saving 16 decimal values representing code of curr->data
        s.assign(127 - curr->code.length(), '0');
        s += '1';
        s += curr->code;
        //Saving decimal values of every 8-bit binary code
        in += (char)binToDec(s.substr(0, 8));
        for (int i = 0; i < 15; i++) {
```

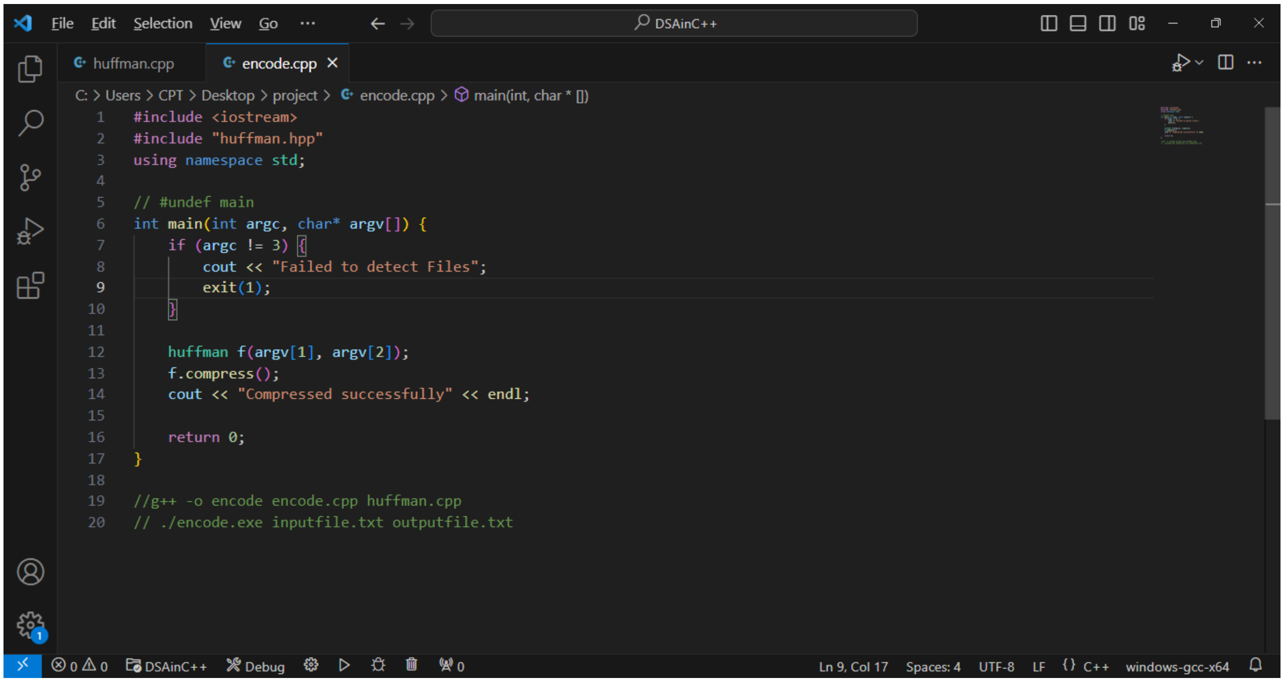


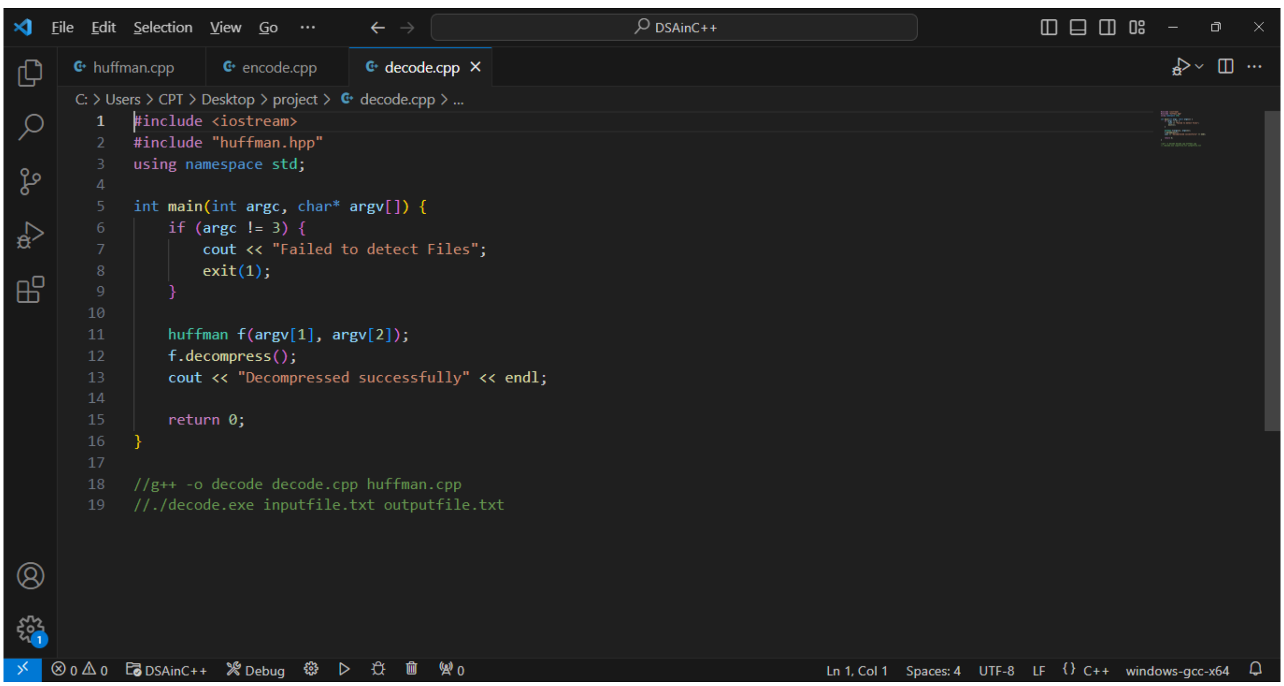
```cpp
            s = s.substr(8);
            in += (char)binToDec(s.substr(0, 8));
        }
        tempPQ.pop();
    }
    s.clear();

    //Saving codes of every charachter appearing in the input file
    inFile.get(id);
    while (!inFile.eof()) {
        s += arr[id]->code;
        //Saving decimal values of every 8-bit binary code
        while (s.length() > 8) {
            in += (char)binToDec(s.substr(0, 8));
            s = s.substr(8);
        }
        inFile.get(id);
    }

    //Finally if bits remaining are less than 8, append 0's (padding)
    int count = 8 - s.length();
    if (s.length() < 8) {
        s.append(count, '0');
    }
    in += (char)binToDec(s);
```

```cpp
    }
    in += (char)binToDec(s);
    //append count of appended 0's
    in += (char)count;

    //write the in string to the output file
    outFile.write(in.c_str(), in.size());
    inFile.close();
    outFile.close();
}

void huffman::saveDecodedFile() {
    inFile.open(inFileName, ios::in | ios::binary);
    outFile.open(outFileName, ios::out);
    unsigned char size;
    inFile.read(reinterpret_cast<char*>(&size), 1);
    //Reading count at the end of the file which is number of bits appended to make final value 8-bit
    inFile.seekg(-1, ios::end);
    char count0;
    inFile.read(&count0, 1);
    //Ignoring the meta data (huffman tree) (1 + 17 * size) and reading remaining file
    inFile.seekg(1 + 17 * size, ios::beg);

    vector<unsigned char> text;
    unsigned char textseg;
    inFile.read(reinterpret_cast<char*>(&textseg), 1);
    while (!inFile.eof()) {
        text.push_back(textseg);
        inFile.read(reinterpret_cast<char*>(&textseg), 1);
    }

    Node *curr = root;
    string path;
    for (int i = 0; i < text.size() - 1; i++) {
        //Converting decimal number to its equivalent 8-bit binary code
        path = decToBin(text[i]);
        if (i == text.size() - 2) {
            path = path.substr(0, 8 - count0);
        }
        //Traversing huffman tree and appending resultant data to the file
        for (int j = 0; j < path.size(); j++) {
            if (path[j] == '0') {
                curr = curr->left;
            }
            else {
                curr = curr->right;
            }

            if (curr->left == NULL && curr->right == NULL) {
                outFile.put(curr->data);
                curr = root;
```

```cpp
            }
        }
        inFile.close();
        outFile.close();
}

void huffman::getTree() {
    inFile.open(inFileName, ios::in | ios::binary);
    //Reading size of MinHeap
    unsigned char size;
    inFile.read(reinterpret_cast<char*>(&size), 1);
    root = new Node();
    //next size * (1 + 16) characters contain (char)data and (string)code[in decimal]
    for(int i = 0; i < size; i++) {
        char aCode;
        unsigned char hCodeC[16];
        inFile.read(&aCode, 1);
        inFile.read(reinterpret_cast<char*>(hCodeC), 16);
        //converting decimal characters into their binary equivalent to obtain code
        string hCodeStr = "";
        for (int i = 0; i < 16; i++) {
            hCodeStr += decToBin(hCodeC[i]);
        }
        //Removing padding by ignoring first (127 - curr->code.length()) '0's and next '1' character
        int j = 0;
```

```cpp
        //Removing padding by ignoring first (127 - curr->code.length()) '0's and next '1' character
        int j = 0;
        while (hCodeStr[j] == '0') {
            j++;
        }
        hCodeStr = hCodeStr.substr(j+1);
        //Adding node with aCode data and hCodeStr string to the huffman tree
        buildTree(aCode, hCodeStr);
    }
    inFile.close();
}

void huffman::compress() {
    createMinHeap();
    createTree();
    createCodes();
    saveEncodedFile();
}

void huffman::decompress() {
    getTree();
    saveDecodedFile();
}
```


File Edit Selection View Go
DSAinC++
huffman.cpp
encode.cpp
C: > Users > CPT > Desktop > project > encode.cpp > main(int, char * [])
#include <iostream>
#include "huffman.hpp"
using namespace std;
// #undef main
int main(int argc, char* argv[]) {
if (argc != 3) {
cout << "Failed to detect Files";
exit(1);
}
huffman f(argv[1], argv[2]);
f.compress();
cout << "Compressed successfully" << endl;
return 0;
}
//g++ -o encode encode.cpp huffman.cpp
// ./encode.exe inputfile.txt outputfile.txt
Ln 9, Col 17 Spaces: 4 UTF-8 LF C++ windows-gcc-x64


File Edit Selection View Go
DSAinC++
huffman.cpp
encode.cpp
decode.cpp
C: > Users > CPT > Desktop > project > decode.cpp > ...
#include <iostream>
#include "huffman.hpp"
using namespace std;
int main(int argc, char* argv[]) {
if (argc != 3) {
cout << "Failed to detect Files";
exit(1);
}
huffman f(argv[1], argv[2]);
f.decompress();
cout << "Decompressed successfully" << endl;
return 0;
}
//g++ -o decode decode.cpp huffman.cpp
//./decode.exe inputfile.txt outputfile.txt
Ln 1, Col 1 Spaces: 4 UTF-8 LF C++ windows-gcc-x64