Project Report

Youssef Ayman Hassan - Adham Elrouby - Hassan Ashraf -
Mohamed Elsayed - Mohamed Hazem Ali

December 7, 2024

§1 Abstract

This project presents a file compression system using Huffman coding to reduce file sizes while preserving data accuracy. The system performs the effective implementation of binary trees and priority queues by allocating shorter binary codes to the higher frequency characters. It can handle text files of large size, so it is scalable, flexible, and reliable. This project emphasizes how advanced data structures are used in real-world challenges involving file storage and transmission. Future development might entail the addition of efficient performance with very large datasets and additional data format support.

§2 Introduction

File compression plays a pivotal role in modern computing, enabling efficient storage and transmission of data. This project implements a file compression and decompression system based on Huffman coding, a lossless compression algorithm that assigns shorter binary codes to characters with higher frequencies. The system is built around the construction of Huffman trees using priority queues, ensuring efficient encoding and decoding of text files while preserving data integrity.

The primary goals of this project include achieving efficient compression, ensuring scalability for large datasets, and maintaining the reliability and flexibility needed for various use cases. By leveraging advanced data structures, this project highlights a practical application of computer science principles to address challenges in file storage and transmission.

This report details the implementation and evaluation of the proposed system, demonstrating its effectiveness in reducing file sizes while ensuring lossless recovery of the original data.

§3 Problem Definition

Handling large data volumes is a challenge in computing. While popular compression tools perform well in many cases, they often struggle with performance on large files, adaptability across different use cases, or fine-grained customization.
This project aims to address these limitations with the following goals:

- **Efficient Compression:** Minimize file sizes while preserving the exact original data.

- **Scalability:** Ensure the system can handle large datasets efficiently.
- **Flexibility:** Make the tool adaptable for various data formats and use cases.
- **Reliability:** Guarantee lossless decompression.

By focusing on these objectives, the project delivers a strong compression system that combines theoretical efficiency with practical utility.

§4 Methodology

§4.1 Encoding Methodology

1. Calculate Character Frequencies.
2. Build the Huffman Tree using the min-heap.
3. Assign Binary Codes for each character according to its frequency.
4. Encode the Input by replacing each character with its representative binary code.
5. Write Encoded Data by dividing the binary string into 8-bit chunks for storage and padding any leftover bits with zeros.

§4.2 Decoding Methodology

1. Read the Encoded Data from the compressed file.
2. Decode the Binary Data using the Huffman tree.
3. Remove Padding.

§4.3 Compression Methodology

1. Verify if the input is a .txt file.
2. Build the Huffman tree using the character frequencies from the input file.
3. Construct the header with character and Huffman code information.
4. Convert the file content into a binary string using the Huffman codes.

§4.4 Decompression Methodology

1. Extract the header and Huffman codes from the compressed file.
2. Use the header to reconstruct the Huffman tree.
3. Convert the binary string back into the original text using the Huffman tree.
4. Ensure proper data extraction by removing any padding.

§5 Specifications of Algorithms to be used

§5.1 Huffman Tree

First, to construct the Huffman tree, we will follow the following algorithm:

1. **Initialize the priority queue:** Every character which has a frequency not equal to zero becomes a leaf. The priority queue sorts those nodes according to their frequency; the lowest frequency is on top of it, using the custom comparator.
2. **Merge nodes:**
   - Two nodes are taken away one after another with a low frequency
   - Create a new inner node and its frequency being the sum of its previous two nodes.
   - Throw it back in the queue again.
3. **Set Root:** When only one node is left in the queue, that becomes the root of the Huffman Tree.
4. **Generate Codes:** Now, use the following recursive generate_codes function in order to generate the binary codes for each character using the root specified in the previous step.

Then, to generate the binary mask of each character, we will use the following algorithm. It takes the current node, the root will be inserted in the first call, and a string representing the binary code built so far. Then,

1. **Check if the node is a leaf:**
   - If the node is NULL, then it stops the base case.
   - If it is a leaf node, the character is not the special value 127 used for internal nodes, it saves the binary code for this character in the huffman_codes array.
2. **Recursive calls:**
   - Appending '0' to the code while going left.
   - Appending '1' when going right.

Using the previous algorithms of creating a Huffman tree, it becomes possible to encode and decode the characters in any text using the following encoding and decoding algorithms.

§5.2 Encoding algorithm

1. Create Temporary Storage
2. Iterate Over Each Character in originalText. For each character in originalText:
   - Its ASCII value is retrieved.
   - In case this character is within the valid range of ASCII (0 - 127), its corresponding Huffman code is appended to the string.
3. Convert Binary Data in Chunks of 8 Bits
   - The function processes the accumulated binary string s:

- Slices off the first 8 bits (byteStr).
- Converts this binary string into its decimal equivalent
- Converts the decimal into character and appends to it.
- Removes the processed 8 bits from the string.

4. Handle Remaining Bits
   - After processing all full bytes, whatever remaining bits (less than 8) are padded with zeros to make them a full byte.
   - This padded byte is converted to a character and appended to in.
5. Store Padding Information
   - The number of padding bits count is stored.
   - The process also ensures that during decoding, the exact number of valid bits in the last byte can be identified, together with the removal of padding.

§5.3 Decoding algorithm

1. Initialization:
   - decodedText: An empty string that shall store the final decoded text.
   - i: A pointer for the current position in the encoded text (codedText).
2. Iterative Decoding: The function goes through the encoded text character by character, and in each iteration:
   - Bits are appended one after the other to the current string from encoded text.
   - At each bit addition, the function calls linearSearchKey to verify whether the so far accumulated current string matches with any Huffman code.
3. Matching Code:
   - In case of a match -linearSearchKey returns a valid index, the character corresponding to the index is appended to decodedText.
   - The while-loop exits as the current Huffman code is completely decoded.
4. Update Position: Move position i to j + 1, so the next portion of the encoded text will be processed for decoding by the function.

§5.4 Compression

First, the algorithm checks if the input file is .txt. If it is not, the program outputs an error message and stops the algorithm. Otherwise, we read the input file and generate a frequency array that is used as an argument to generate the Huffman tree and map. Afterward, we use the Huffman Map to generate the header structure that will be used in decompression later on. In the header structure, the first byte is the number of characters used in the Huffman map (let it be n). The following segments consist of three parts:

1. Character Byte (the letter)
2. Huffman Code Size Byte (S)
3. Huffman Code encoded in ceil(S/8.0) bytes

Afterward, the content of the input file is converted into a large sequence of a binary string according to the Huffman codes for each character in the map. Afterward, The file content is encoded as described in the encoding algorithm.

§5.5 Decompression

Similarly, the algorithm first checks if the input file is .hassan. If it is not, the program gives an error message and stops the algorithm. Otherwise, the Huffman Map is first regenerated by reversing the process described in the compression algorithm and moving along the specified header structure. After that, the file content bytes are all converted to a binary string that is then processed to regenerate the original text. During the process of the binary string, we check if the current sequence of the binary string corresponds to an existing binary string in the Huffman Map (Note: this is true since in Huffman Tree, no two characters' codes have the same prefix). After retrieving the original text, a .txt file is created with the original content.

§6 Experimental Results

We made five different text files with varying sizes to test our compression algorithm. For each, we calculated the compression ratio (CR), which is defined as the following:

$$CR = \frac{\text{Size of the Compressed File}}{\text{Size of the Original File}} \times 100\%$$

Table 1 illustrates the sizes of the test cases and the corresponding compressed files and compression ratios.

Table 1: Test Results

Test case	Original Size (KB)	Compressed Size (KB)	Compression Ratio
T1	0.895	0.638	71.3%
T2	76.7	42.7	55.7%
T3	230	127	55.2%
T4	446	245	54.9%
T5	1613	895	55.4%

Figure 1 shows a graphical relationship between the input text file size and the output compression ratio.

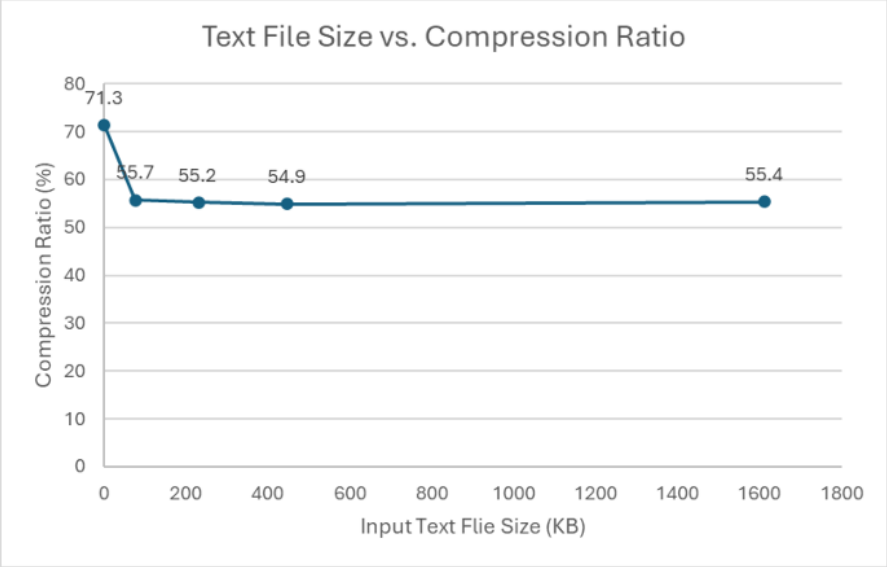

Figure 1: The relationship between text file size and the compression ratio

§7 Analysis and Critique

For the Huffman tree data structure, we represent the character held by internal nodes by the DEL character (ASCII code: 127). This is because it is not a printable character, so it will never need a Huffman code string.
For analyzing the compression/decompression algorithm, we find that the Huffman Map is an array of 128 strings, where each index represents the ASCII code for the 128 ASCII characters. Therefore, if a file contains a character that is outside those 128 characters or, for instance, a different language, then the compressed file will not take these into account, and these characters will not be retrieved after decompression. It would be difficult to account for characters outside of the 128 ASCII characters because some of these characters require a combination of two or more ASCII characters, which would require a significant change in the data structures used, although the algorithm will be the same.
If we measured the average compression ratio using the five test cases, we find that:

$$CR_{avg} = \frac{71.3 + 55.7 + 55.2 + 54.9 + 55.4}{5} = 58.5\%$$

However, we notice that the compression ratio at small file sizes is relatively large while starting from a threshold of 76.7 KB file size, the compression ratio (CR) stabilizes. This makes sense because the header used to regenerate the Huffman map during decompression takes a relatively fixed size regardless of the file input size. This fixed size becomes very small compared to the input file text when the input text file size increases, therefore, when we measure the average compression ratio after a threshold of 76.7 KB, then:

$$CR_{avg} = \frac{55.7 + 55.2 + 54.9 + 55.4}{4} = 55.3\%$$

This average compression ratio remains approximately the same for larger input file texts.

§8 Conclusion

The implementation of Huffman coding in this project demonstrates its effectiveness in file compression. By using binary trees and priority queues, the system achieves significant size reductions and handles large text files efficiently. The tool's adaptability and reliability make it suitable for diverse applications. Future work could explore optimizing algorithms for extreme-scale datasets or expanding support for more data types, further increasing its practical applicability in modern computing.

§9 Acknowledgements

We would like to express our sincere gratitude to Dr. Ashraf for his continuous support, invaluable guidance, and insightful feedback throughout the course of this project. His expertise and encouragement were instrumental in shaping our approach and ensuring our success.
We also extend our thanks to Dr. Dina for her constructive feedback, which helped us refine our work and overcome challenges.
A special thank you to Eng. Mohamed Ibrahim, our teaching assistant, for his unwavering support and guidance. His assistance in addressing technical issues and providing helpful advice greatly contributed to the completion of this project.
Lastly, we are grateful for the opportunity to work on this project and for the collaborative effort of all team members.

§10 References

[1] M. A. Weiss, Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2013. ISBN 978-0132847377.
[2] GeeksforGeeks, "Huffman Coding — Greedy Algo-3," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.
[3] J. Zelenski, K. Schwarz, and M. Stepp, "Huffman Encoding and Data Compression," Stanford University, [Online]. Available: Link

§11 Appendix

The code for the project can be found in this GitHub repository.

HVector.h

```
#ifndef HVECTOR_H
#define HVECTOR_H

#include <stdexcept>
using namespace std;
template<typename T>
class HVector
{
private:
    T *data;       // Pointer to the array holding elements
    int capacity;  // Current allocated size
    int size;      // Number of elements in the vector
```

```
    void resize(); // Resize the vector when capacity is reached

public:
    HVector(); // Constructor
    HVector(int initialCapacity, const T &initialValue);
    ~HVector(); // Destructor

    void push_back(const T &value);
    void pop_back();
    T &operator[](int index);
    const T &operator[](int index) const;
    T &front(); // Access the first element
    const T &front() const;
    T &back(); // Access the last element
    const T &back() const;
    int getSize() const;
    bool empty() const;
};

// Implementation of HVector template methods

template<typename T>
HVector<T>::HVector()
    : data(nullptr)
    , capacity(0)
    , size(0)
{}

template<typename T>
HVector<T>::HVector(int initialCapacity, const T &initialValue)
    : capacity(initialCapacity)
    , size(initialCapacity)
{
    data = new T[capacity];
    for (int i = 0; i < size; i++) {
        data[i] = initialValue;
    }
}

template<typename T>
HVector<T>::~HVector()
{
    delete[] data;
}

template<typename T>
void HVector<T>::push_back(const T &value)
{
    if (size == capacity) {
        resize();
```

```
    }
    data[size++] = value;
}

template<typename T>
void HVector<T>::pop_back()
{
    if (size == 0)
        throw out_of_range("HVector is empty. Cannot pop.");
    size--;
}

template<typename T>
T &HVector<T>::operator[](int index)
{
    if (index < 0 && index >= size)
        throw out_of_range("Index out of bounds.");
    return data[index];
}

template<typename T>
const T &HVector<T>::operator[](int index) const
{
    if (index < 0 && index >= size)
        throw out_of_range("Index out of bounds.");
    return data[index];
}

template<typename T>
T &HVector<T>::front()
{
    if (empty())
        throw out_of_range("HVector is empty. Cannot access front.");
    return data[0];
}

template<typename T>
const T &HVector<T>::front() const
{
    if (empty())
        throw out_of_range("HVector is empty. Cannot access front.");
    return data[0];
}

template<typename T>
T &HVector<T>::back()
{
    if (empty())
        throw out_of_range("HVector is empty. Cannot access back.");
    return data[size - 1];
```

```
}

template<typename T>
const T &HVector<T>::back() const
{
    if (empty())
        throw out_of_range("HVector is empty. Cannot access back.");
    return data[size - 1];
}

template<typename T>
int HVector<T>::getSize() const
{
    return size;
}

template<typename T>
bool HVector<T>::empty() const
{
    return size == 0;
}

template<typename T>
void HVector<T>::resize()
{
    capacity = (capacity == 0) ? 1 : capacity * 2;
    T *newData = new T[capacity];
    for (int i = 0; i < size; i++) {
        newData[i] = data[i];
    }
    delete[] data;
    data = newData;
}

#endif // HVECTOR_H
```

HPQ.h

```
#ifndef HPQ_H
#define HPQ_H

#include "HVector.h"
#include <functional>
#include <stdexcept>

using namespace std;

template<typename T, typename Compare = less<T>>
class HPQ
{
```

```
private:
    HVector<T> heap; // Use HVector as the underlying container
    Compare comp;    // Comparison object to determine priority

    void heapifyUp(int index);
    void heapifyDown(int index);

public:
    HPQ(); // Default constructor
    explicit HPQ(const Compare &compare);

    void push(const T &value);
    void pop();
    T &top();
    const T &top() const;
    bool empty() const;
    int size() const;
};

// Implementation of member functions

template<typename T, typename Compare>
HPQ<T, Compare>::HPQ()
    : comp(Compare())
{}

template<typename T, typename Compare>
HPQ<T, Compare>::HPQ(const Compare &compare)
    : comp(compare)
{}

template<typename T, typename Compare>
void HPQ<T, Compare>::push(const T &value)
{
    heap.push_back(value);
    heapifyUp(heap.getSize() - 1);
}

template<typename T, typename Compare>
void HPQ<T, Compare>::pop()
{
    if (empty())
        throw out_of_range("HPQ is empty. Cannot pop.");
    heap[0] = heap[heap.getSize() - 1];
    heap.pop_back();
    heapifyDown(0);
}

template<typename T, typename Compare>
T &HPQ<T, Compare>::top()
```

```
{
    if (empty())
        throw out_of_range("HPQ is empty. Cannot access top.");
    return heap[0];
}

template<typename T, typename Compare>
const T &HPQ<T, Compare>::top() const
{
    if (empty())
        throw out_of_range("HPQ is empty. Cannot access top.");
    return heap[0];
}

template<typename T, typename Compare>
bool HPQ<T, Compare>::empty() const
{
    return heap.getSize() == 0;
}

template<typename T, typename Compare>
int HPQ<T, Compare>::size() const
{
    return heap.getSize();
}

template<typename T, typename Compare>
void HPQ<T, Compare>::heapifyUp(int index)
{
    while (index > 0) {
        int parent = (index - 1) / 2;
        if (comp(heap[parent], heap[index])) {
            swap(heap[parent], heap[index]);
            index = parent;
        } else
            break;
    }
}

template<typename T, typename Compare>
void HPQ<T, Compare>::heapifyDown(int index)
{
    int leftChild, rightChild, largest;
    while (true) {
        leftChild = 2 * index + 1;
        rightChild = 2 * index + 2;
        largest = index;

        if (leftChild < heap.getSize() && comp(heap[largest],
        ↪  heap[leftChild]))
```

```
            largest = leftChild;
        if (rightChild < heap.getSize() && comp(heap[largest],
        ↪  heap[rightChild]))
            largest = rightChild;

        if (largest != index) {
            swap(heap[index], heap[largest]);
            index = largest;
        } else
            break;
    }
}

#endif // HPQ_H
```

Huffman.h

```
#ifndef HUFFMAN_H
#define HUFFMAN_H
#include<string>
using namespace std;
class Node
{
public:
    char c;
    int freq;
    Node *left, *right;
    Node(char t, int f)
    {
        c = t;
        freq = f;
        left = right = nullptr;
    }
};
class Huffman_tree
{
private:
    Node *root;
    struct custom_comparator
    {
        bool operator()(Node *a, Node *b);
    };
    void generate_codes(Node *current, string s = "");

public:
    string huffman_codes[128] = {""};
    Huffman_tree(const int freq_arr[128]);
};
```

```
#endif
```

Huffman.cpp

```
#include "Huffman.h"
#include "HPQ.h"
#include "HVector.h"
#include <queue>
using namespace std;

bool Huffman_tree::custom_comparator::operator()(Node *a, Node *b)
{
    return a->freq > b->freq; // Min-heap based on frequency
}
void Huffman_tree::generate_codes(Node *current, string s)
{
    if (current == NULL)
        return;
    if (current->c != 127)
        huffman_codes[int(current->c)] = s;
    generate_codes(current->left, s + '0');
    generate_codes(current->right, s + '1');
}

Huffman_tree::Huffman_tree(const int freq_arr[128])
{
    Node *top, *left, *right;

    HPQ<Node *, custom_comparator> q;
    for (int i = 0; i < 128; i++)
    {
        if (freq_arr[i] == 0)
            continue;
        q.push(new Node(char(i), freq_arr[i]));
    }

    while (q.size() > 1)
    {
        left = q.top();
        q.pop();
        right = q.top();
        q.pop();

        top = new Node(127, left->freq + right->freq);

        top->left = left;
        top->right = right;
        q.push(top);
    }
    root = q.top();
```

```
    generate_codes(root, "");
}
```

Header.h

```
#ifndef HEADER_H
#define HEADER_H
#include <fstream>
#include <string>

using namespace std;

// Class to manage the header for Huffman coding
class Header
{
private:
    string header; // Holds the constructed header string

    // Converts a binary string to a decimal integer
    int binary_to_decimal(string &in);

    // Converts a decimal integer to a binary string
    string decimal_to_binary(int in);

public:
    // Default constructor initializes an empty header
    Header();

    // Constructor that generates the header from Huffman codes
    Header(string huffman_codes[128]);

    // Generates the header based on the provided Huffman codes
    void generateHeader(string huffman_codes[128]);

    // Returns the constructed header
    string getHeader();

    // Writes the header to the output string
    void writeHeader(string &in);

    // Reads the header from a file and populates the Huffman codes
    void readHeader(ifstream &file, string huffman_codes[128]);
};

#endif HEADER_H
```

Header.cpp

```
#include <fstream>
#include <string>
#include <cmath>
```

```cpp
#include "Header.h"
using namespace std;

int Header::binary_to_decimal(string &in)
{
    int result = 0; // Resulting decimal value
    for (int i = 0; i < in.size(); i++)
        result = result * 2 + in[i] - '0'; // Convert binary to decimal
    return result;                          // Return the decimal result
}

string Header::decimal_to_binary(int in)
{
    string temp = "";   // Temporary binary string
    string result = ""; // Final binary string with leading zeros

    // Convert decimal to binary
    while (in > 0)
    {
        temp = to_string(in % 2) + temp; // Prepend binary digit
        in /= 2;                         // Divide by 2 for next
        ↪ iteration
    }

    // Add leading zeros to make the binary string 8 bits long
    int leadingZeros = 8 - temp.size();
    result.append(leadingZeros, '0'); // Append leading zeros
    result += temp;                   // Append the binary digits
    return result;                    // Return the binary string
}

Header::Header()
{
    header = "";
}

Header::Header(string huffman_codes[128])
{
    generateHeader(huffman_codes);
}

void Header::generateHeader(string huffman_codes[128])
{
    string tempHeader = ""; // Temporary header string
    unsigned char size = 0; // Size of the header

    // Iterate through all possible characters
    for (int i = 0; i < 128; i++)
    {
        if (huffman_codes[i].empty())
```

```
            continue;                                          // Skip empty
            ↪  codes
        tempHeader.push_back(i);                               // Add
        ↪  character to header
        tempHeader.push_back(huffman_codes[i].length()); // Add code
        ↪  length
        tempHeader += huffman_codes[i];                        // Add the
        ↪  actual Huffman code
        size++;                                                // Increment
        ↪  the size of the header
    }

    char size_char = size;           // Convert size to char
    header = size_char + tempHeader; // Construct the final header
}

string Header::getHeader()
{
    return header;
}

void Header::writeHeader(string &in)
{
    in += header[0]; // Add the size of the header
    int i = 1;       // Start index for reading the header

    // Loop through the header and write it to the output
    while (i < header.size())
    {
        char ch = header[i];                  // Current character
        in += header[i];                      // Add character to output
        int codeSize = header[++i];           // Get size of the Huffman
        ↪  code
        in += codeSize;                       // Add code size to output
        int numBytes = ceil(codeSize / 8.0); // Calculate number of
        ↪  bytes needed
        int padd = numBytes * 8 - codeSize;  // Calculate padding needed
        string s = "";                        // Temporary string for
        ↪  binary code

        // Collect the binary code
        int j;
        for (j = i + 1; j <= i + codeSize; j++)
        {
            s += header[j]; // Append each bit
        }
        s += string(padd, '0'); // Append padding zeros

        // Convert the binary string to bytes and add to output
        for (int k = 0; k < numBytes; k++)
```

```
        {
            string byteStr = s.substr(k * 8, 8);    // Get each byte (8
            ↪ bits)
            in += (char)binary_to_decimal(byteStr); // Convert to char
            ↪ and add to output
        }
        i = j; // Move index to the next character
    }
}

void Header::readHeader(ifstream &file, string huffman_codes[128])
{
    char headerSize;           // Variable to hold the size of the
    ↪ header
    file.read(&headerSize, 1); // Read the header size from file

    // Loop through each character in the header
    for (int i = 0; i < headerSize; i++)
    {
        char letter;                    // Current character
        file.read(&letter, 1);          // Read the character from file
        char letterCodeSize;            // Variable to hold the size of
        ↪ the Huffman code for the character
        file.read(&letterCodeSize, 1); // Read the size of the Huffman
        ↪ code

        // Calculate the number of bytes needed to store the code
        int numBytes = ceil(static_cast<unsigned char>(letterCodeSize) /
        ↪ 8.0);
        string binaryCode = ""; // String to hold the binary code

        // Read the bytes and convert them to a binary string
        for (int j = 0; j < numBytes; j++)
        {
            char byte;                                  // Variable to hold
            ↪ each byte
            file.read(&byte, 1);                        // Read a byte from
            ↪ the file
            unsigned char byte1 = byte;                 // Cast to unsigned
            ↪ char
            binaryCode += decimal_to_binary(byte1); // Convert byte to
            ↪ binary and append
        }

        // Trim the binary code to the correct length
        binaryCode = binaryCode.substr(0, letterCodeSize);
        huffman_codes[letter] = binaryCode; // Store the Huffman code
        ↪ for the character
    }
}
```

compression.cpp

```cpp
#include <iostream>
#include "Huffman.h"
#include <fstream>
#include "Header.h"

// Function to build a Huffman Tree from the content of a file
// and store the original text for compression.
Huffman_tree* readFromFile(ifstream& file, string& originalText) {
    if (!file) {
        cerr << "Failed to open the file.\n";
        return NULL;
    }

    char byte;
    int freq_arr[128] = {0}; // Array to store frequency of each
    ↪  character

    // Read the file content and count character frequencies
    while (file.get(byte)) {
        originalText += byte;
        freq_arr[int(byte)]++;
    }

    // Create Huffman Tree using character frequencies
    Huffman_tree* huffman = new Huffman_tree(freq_arr);

    file.close();
    return huffman;
}

// Function to convert a binary string to its decimal equivalent
int binary_to_decimal(string& in) {
    int result = 0;
    for (int i = 0; i < in.size(); i++)
        result = result * 2 + in[i] - '0'; // Calculate decimal value
        ↪  iteratively
    return result;
}

// Function to convert a decimal number to an 8-bit binary string
string decimal_to_binary(int in) {
    string temp = "";
    string result = "";

    while (in > 0) {
        temp = to_string(in % 2) + temp; // Get binary digits
        in /= 2;
    }
```

```
    // Pad with leading zeros to make it 8 bits
    int leadingZeros = 8 - temp.size();
    result.append(leadingZeros, '0');

    for (int i = 0; i < temp.size(); i++) {
        result += temp[i];
    }

    return result;
}

// Function to encode text using Huffman codes
void encodeText(string& in, const string& originalText, string
↪  huffman_codes[128]) {
    string s = ""; // Temporary string to hold binary data

    for (auto& byte : originalText) {
        int idx = int(byte);
        if(idx >= 0 && idx < 128) { // if the indxe is in valid ASCII
        ↪  range [0,127]
            s += huffman_codes[int(byte)]; // Append Huffman code for
            ↪  each character
        }

        // Write bytes to the output when we have at least 8 bits
        while (s.size() >= 8) {
            string byteStr = s.substr(0, 8);
            in += (char)binary_to_decimal(byteStr); // Convert to
            ↪  character
            s = s.substr(8); // Remove processed bits
        }
    }

    // Handle remaining bits
    int count = s.size() > 0 ? 8 - s.size() : 0;
    if (s.size() > 0) {
        s.append(count, '0'); // Pad remaining bits with zeros
        in += (char)binary_to_decimal(s); // Add to output
    }
    in += (char)count; // Store padding size
}

// Function to compress a text file using Huffman encoding
string compressFile(string fileIn) {
    int index = fileIn.rfind(".");
    if (index != string::npos && fileIn.substr(index + 1) != "txt") {
        cout << "The file is not a text file.\n";
        return "";
    }
```

```
    string fileOut = fileIn.substr(0, index) + "_compressed.hassan";
    ifstream file(fileIn, ios::binary);
    ofstream output(fileOut, ios::binary);

    if (!file || !output) {
        cerr << "Failed to open the file.\n";
        return "";
    }

    if (file.peek() == ifstream::traits_type::eof()) {
        cout << "The file is empty.\n";
        return "";
    }

    string originalText = "";
    Huffman_tree* huffman = readFromFile(file, originalText);
    if (huffman == NULL) {
        cout << "Failed to create the Huffman tree.\n";
        return "";
    }

    cout << "**** During Compression ******\n";

    cout << "Size of the original file: " << originalText.size() <<
    ↪  endl;

    Header h(huffman->huffman_codes);
    string in = "";
    h.writeHeader(in);

    cout << "Size of the header: " << in.size() << endl;

    encodeText(in, originalText, huffman->huffman_codes);

    cout << "Size of the compressed file: " << in.size() << endl;
    output.write(in.c_str(), in.size());

    file.close();
    output.close();
    string re = to_string((originalText.size() / 1024.0)) + "KB --> " +
    ↪  to_string((in.size() / 1024.0)) + "KB";
    return re;
}

// Helper function to find the character corresponding to a Huffman code
int linearSearchKey(string val, string huffman_codes[128]) {
    for (int i = 0; i < 128; i++) {
        if (huffman_codes[i] == val) return i;
    }
    return -1;
```

```
}

// Function to decode a compressed text using Huffman codes
string decodeText(string huffman_codes[128], const string& codedText) {
    size_t i = 0;
    string decodedText = "";
    while (i < codedText.size()) {
        string current = "";
        size_t j;
        for (j = i; j < codedText.size(); j++) {
            current += codedText[j];
            int idx = linearSearchKey(current, huffman_codes);
            if (idx != -1) {
                decodedText += char(idx); // Append decoded character
                break;
            }
        }
        i = j + 1;
    }

    return decodedText;
}

// Function to retrieve the coded content of a compressed file
string getCodedContent(ifstream& fileIn) {
    string fileContent;
    char byte;

    // Read all bytes from the file
    while (fileIn.get(byte)) {
        fileContent += byte;
    }

    int paddSize = fileContent.back(); // Padding size
    string codedText;

    for (int i = 0; i < fileContent.size() - 1; i++) {
        unsigned char byte1 = fileContent[i];
        string byteStr = decimal_to_binary(byte1); // Convert byte to
        ↪ binary
        codedText += byteStr;
    }

    // Remove padding bits
    codedText = codedText.substr(0, codedText.size() - paddSize);

    return codedText;
}

// Function to decompress a file encoded with Huffman compression
```

```
string decompressFile(string fileIn) {
    int index = fileIn.rfind(".");
    if (index != string::npos && fileIn.substr(index + 1) != "hassan") {
        cout << "The file is not a hassan file.\n";
        return "";
    }

    string fileOut = fileIn.substr(0, index) + "_original.txt";
    ofstream output(fileOut, ios::binary);
    ifstream file(fileIn, ios::binary);

    if (!file || !output) {
        cerr << "Failed to open the file.\n";
        return "";
    }

    if (file.peek() == ifstream::traits_type::eof()) {
        cout << "The file is empty.\n";
        return "";
    }

    Header* h;
    string huffman_codes[128];

    h->readHeader(file, huffman_codes);
    cout << "** During Decompression **\n";

    string codedText = getCodedContent(file);


    string decodedText = decodeText(huffman_codes, codedText);


    output.write(decodedText.c_str(), decodedText.size());

    file.close();

    // find compressed file size
    file.open(fileIn, ios::binary);
    file.seekg(0, std::ios::end);
    streampos fileInSize = file.tellg();
    cout << fileInSize << endl;
    // find decompressed file size
    streampos fileOutSize = output.tellp();
    cout << fileOutSize << endl;

    file.close();
    output.close();
    string re = to_string((fileInSize / 1024.0)) + "KB --> " +
    ↪  to_string((fileOutSize / 1024.0)) + "KB";
```

```
    return re;
}
```

MainWindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QFileSystemModel>

QT_BEGIN_NAMESPACE
namespace Ui {
class MainWindow;
}
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void on_pushButton_clicked();

    void on_Compress_clicked();

    void on_Decompress_clicked();

private:
    Ui::MainWindow *ui;
    QFileSystemModel * explorer;
    QString selected_path;
};
#endif // MAINWINDOW_H
```

MainWindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "cpp-code/compression.cpp"
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
```

```
    this->setFixedSize(1000,750);
    explorer = new QFileSystemModel(this);
    explorer->setRootPath("::{20D04FE0-3AEA-1069-A2D8-08002B30309D}");
    ui->treeView->setModel(explorer);
    ui->treeView->setColumnWidth(0, 500);
    selected_path = "";
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_pushButton_clicked()
{
    if(ui->treeView->selectionModel()->selectedIndexes().size()){
        QModelIndex index =
        ↪  ui->treeView->selectionModel()->selectedIndexes()[0];
        selected_path = explorer->filePath(index);
        ui->selected->setText("Selected File: " +
        ↪  index.data().toString());
        ui->notify->setText("");
        ui->size->setText("");
    }
}


void MainWindow::on_Compress_clicked()
{
    if(selected_path == ""){
        ui->notify->setText("Error: Please Select a File");
        return;
    }
    QString temp = selected_path;
    if(temp.size() < 4){
    ui->notify->setText("Error: Please Select a '.txt' File");
        return;
    }
    std::reverse(temp.begin(), temp.end());
    if(temp[0] != 't' || temp[1] != 'x' || temp[2] != 't' || temp[3] !=
    ↪  '.'){
        ui->notify->setText("Error: Please Select a '.txt' File");
        return;
    }
    ui->selected->setText("No File Selected");
    string size = compressFile(selected_path.toStdString());
    ui->notify->setText("Compression Succesful!");
    ui->size->setText(QString::fromStdString(size));
    selected_path = "";
}
```

```
void MainWindow::on_Decompress_clicked()
{
    if(selected_path == ""){
        ui->notify->setText("Error: Please Select a File");
        return;
    }
    QString temp = selected_path;
    if(temp.size() < 7){
        ui->notify->setText("Error: Please Select a '.hassan' File");
        return;
    }
    std::reverse(temp.begin(), temp.end());
    if(temp[0] != 'n' || temp[1] != 'a' || temp[2] != 's' || temp[3] !=
    ↪  's' || temp[4] != 'a' || temp[5] != 'h' || temp[6] != '.'){
        ui->notify->setText("Error: Please Select a '.hassan' File");
        return;
    }
    ui->selected->setText("No File Selected");
    string size = decompressFile(selected_path.toStdString());
    ui->notify->setText("Decompression Succesful!");
    ui->size->setText(QString::fromStdString(size));
    selected_path = "";
}
```

main.cpp

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```