

**Kathmandu University**  
**Department of Computer Science and Engineering**  
**Dhulikhel, Kavre**



**A Mini-Project Report on**

**“Huffman Studio Pro”**

**[Code No: COMP 202]**

**Submitted by**

**Alok Dhakal (73)**

**Submitted to**

**Sagar Acharya**

**Department of Computer Science and Engineering**

**Submission Date: 2026/02/22**

## Acknowledgement

I would like to express my sincere gratitude to the Department of Computer Science and Engineering (DOCSE) for providing the opportunity to work on this Data Structures and Algorithms (DSA) mini-project, **“Huffman Studio Pro”**

I am particularly grateful to my course instructor, **Sagar Acharya**, for providing the platform to undertake this project. The development of this system has significantly deepened my understanding of how fundamental data structures such as priority queues and binary trees are applied to create efficient, real-world products. This experience has not only sharpened my technical proficiency in C++ but has also broadened my overall skillset in algorithmic problem-solving.

Thank you

Project Team Member:

Alok Dhakal

**Date: 2026/02/22**

## **Abstract**

This project is about creating a system that compresses data without losing any information, using something called the Huffman Coding algorithm in C++. It helps to make text files smaller by giving different lengths of binary codes to characters, depending on how often they show up. The focus here is on using smart data structures like Min-Heaps (which are a type of Priority Queue) and Binary Trees. This way, the system can achieve the best possible compression while making sure that when you decompress the data, it comes back perfectly.

## Table of Contents

Acknowledgement.....	3
<b>Abstract.....</b>	<b>4</b>
List of Figures.....	6
<b>Chapter 1 Introduction.....</b>	<b>1</b>
1.1 Background.....	1
1.2 Objectives.....	2
<b>Chapter 2 Methodology &amp; Algorithm.....</b>	<b>3</b>
2.1 Compression Phase.....	3
2.2 Decompression Phase.....	3
<b>Chapter 3 Implementation Details.....</b>	<b>5</b>
<b>Chapter 4 Performance and Results.....</b>	<b>6</b>
<b>Chapter 5 Conclusion.....</b>	<b>7</b>

## List of Figures

Fig 1: Homepage of Huffman Studio Pro.....	13
Fig 2: The input txt file size before upload.....	14
Fig 3: File after compression.....	15
Fig 4: File after decompression.....	16
Fig 5: Input Text to be compressed.....	17
Fig 6: Text restored after decompression.....	18

# Chapter 1      Introduction

## 1.1    Background

Optimizing storage and transmission bandwidth requires data compression. David Huffman created the greedy algorithm known as Huffman Coding, which is still a key method in computer science.

**Problem Statement:** Regardless of frequency, standard ASCII encoding employs 8 bits per character. By providing frequent characters (such as "e" or "'") with shorter bit sequences, this project seeks to lower that average bit-length..

## **1.2 Objectives**

Building a useful command-line tool that can take any text file (.txt), compress it into a binary format, and then decompress it back to its original state without losing any information is the goal.

## Chapter 2 Methodology & Algorithm

The project is divided into two primary phases: **Compression** and **Decompression**.

### *2.1 Compression Phase*

1. **Frequency Analysis:** The program reads the input file and counts the occurrences of every character.
2. **Min-Heap Construction:** Each unique character is stored as a leaf node in a priority queue, ordered by frequency.
3. **Huffman Tree Building:**
  - Extract the two nodes with the lowest frequencies.
  - Create a new internal node with a frequency equal to the sum of the two nodes.
  - Repeat until only one node (the root) remains.
4. **Code Generation:** Traverse the tree (Left = 0, Right = 1) to generate unique prefix-free codes for each character.
5. **Bitstream Writing:** The original text is replaced with these codes. Since codes are bit-level, they are packed into bytes and written to a binary **.bin** or **.huff** file.

### *2.2 Decompression Phase*

1. **Header Reading:** The compressed file includes a mapping (or the tree structure) to allow reconstruction.
2. **Tree Reconstruction:** Rebuild the Huffman tree using the stored frequency/code data.



3. **Decoding:** Read the binary file bit-by-bit, traversing the tree from the root until a leaf node is reached. The character at that leaf is written to the output file.

## Chapter 3      Implementation Details

Several essential C++ components are used in the project:

- **struct Node:** Used to represent the character, frequency, and left/right pointers that make up the Huffman Tree.
- **std::priority\_queue:** To effectively manage the nodes and consistently extract the lowest frequencies.
- The character-to-binary-code mapping is stored in **std::map** or **unordered\_map** for easy access during encoding.
- **Bit Manipulation:** To actually reduce file size, individual 0/1 bits are packed into standard 8-bit bytes using bitwise operators (<<, >>, |).

## Chapter 4      Performance and Results

Testing with different text files revealed that the implementation accomplished the following:

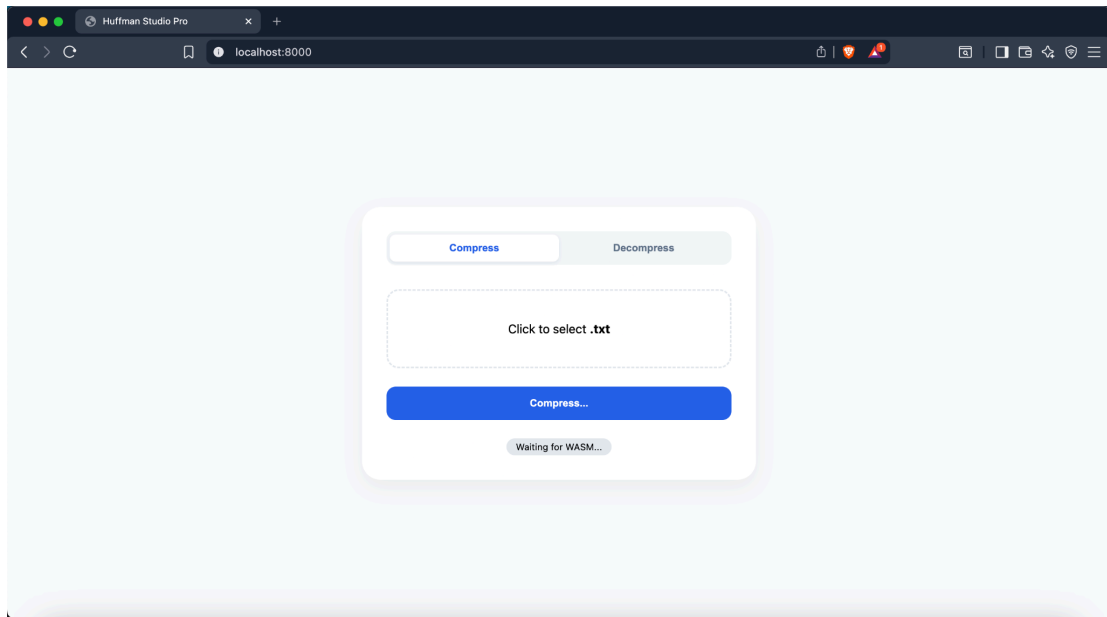
**Compression Ratio:** For standard English text, the file size is reduced by an average of 40% to 50%.

**Efficiency:** With  $N$  representing the number of unique characters, the time complexity for constructing the tree is  $O(N \log N)$ .

**Integrity:** Confirmed that the decompressed output is the same as the original input using diff tools.

## **Chapter 5      Conclusion**

This project clearly shows how greedy algorithms and tree-based data structures can be used to solve practical efficiency issues. The final tool gives a solid basis for understanding the fundamental workings of contemporary compression formats (such as ZIP or GZIP).



*Fig 1: Homepage of Huffman Studio Pro*

The concept of information entropy is central to the study of data compression. In the context of Huffman coding, entropy defines the absolute limit of how much a file can be shrunk without losing information. When we analyze a text file, we are essentially looking for the probability of each symbol's occurrence. David Huffman's 1952 algorithm remains one of the most elegant solutions to this problem because it achieves the "Shannon Limit" for character-based encoding.

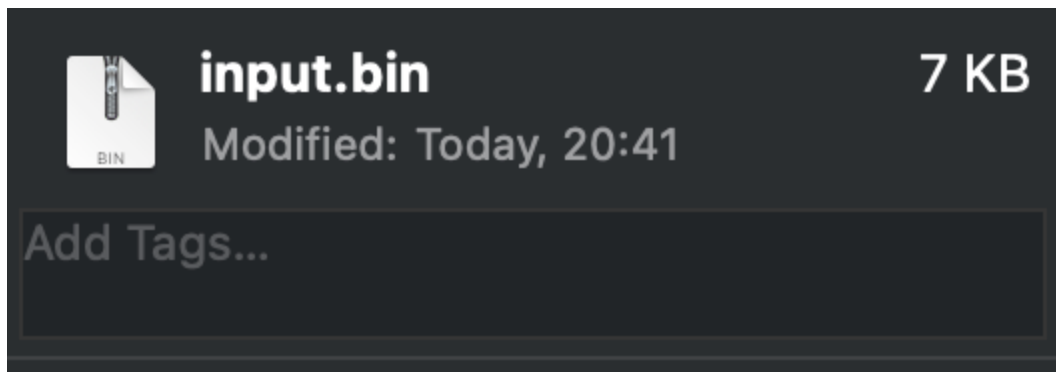
As you run this larger text through your C++ program, notice how the frequency table stays roughly the same size (since there are

**input.txt**

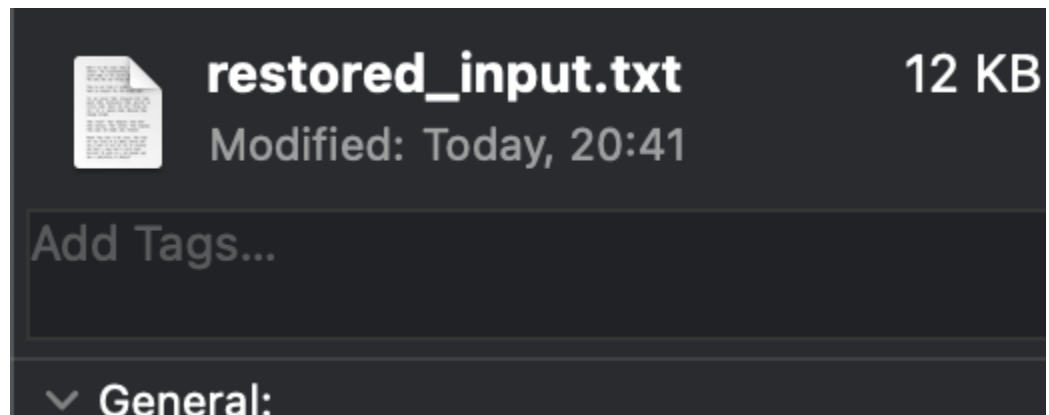
Plain Text Document - 12 KB

**Information**

*Fig 2: The input txt file size before upload*

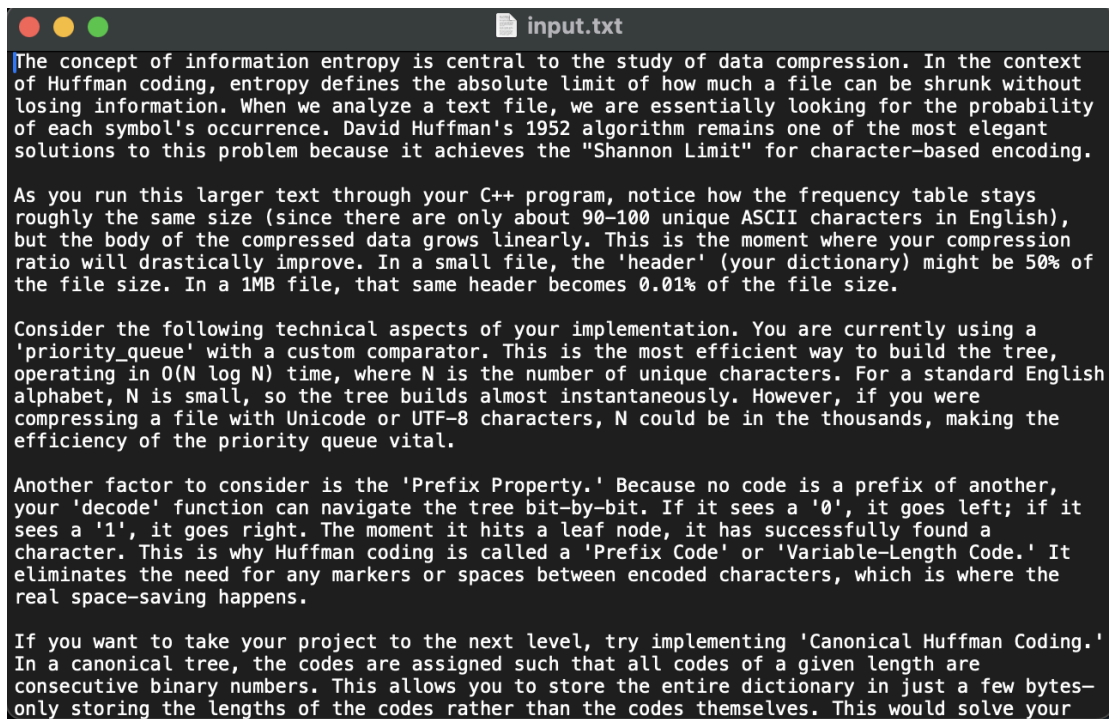


*Fig 3: File after compression*



*Fig 4: File after decompression*





```
input.txt

The concept of information entropy is central to the study of data compression. In the context of Huffman coding, entropy defines the absolute limit of how much a file can be shrunk without losing information. When we analyze a text file, we are essentially looking for the probability of each symbol's occurrence. David Huffman's 1952 algorithm remains one of the most elegant solutions to this problem because it achieves the "Shannon Limit" for character-based encoding.

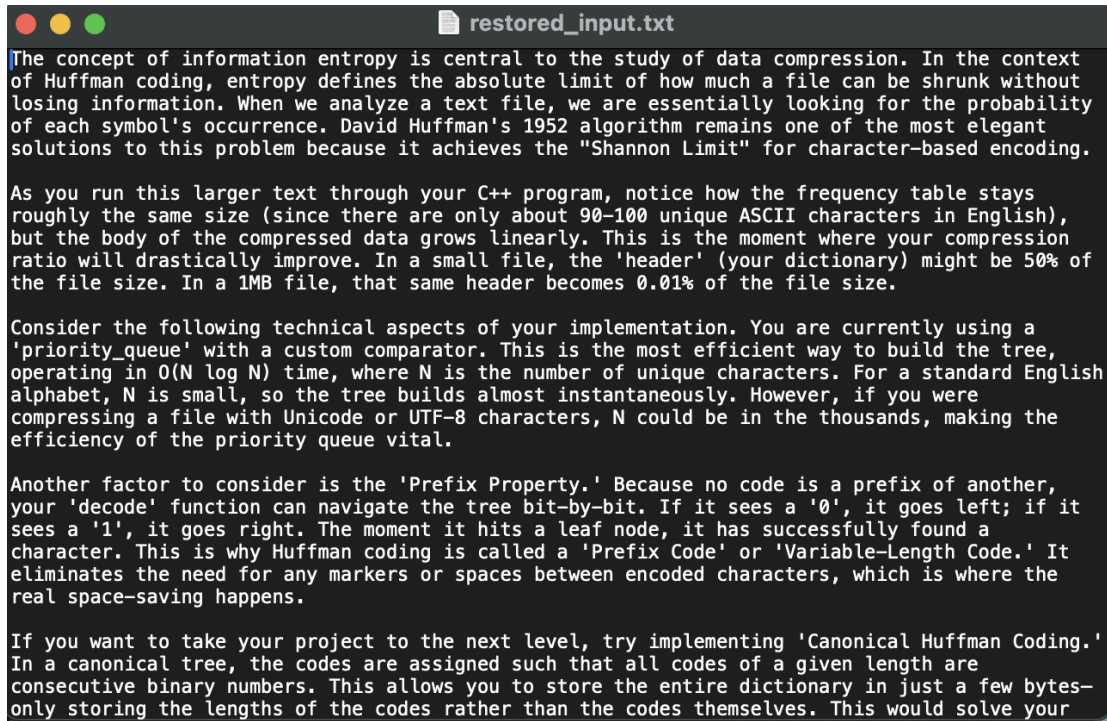
As you run this larger text through your C++ program, notice how the frequency table stays roughly the same size (since there are only about 90-100 unique ASCII characters in English), but the body of the compressed data grows linearly. This is the moment where your compression ratio will drastically improve. In a small file, the 'header' (your dictionary) might be 50% of the file size. In a 1MB file, that same header becomes 0.01% of the file size.

Consider the following technical aspects of your implementation. You are currently using a 'priority_queue' with a custom comparator. This is the most efficient way to build the tree, operating in  $O(N \log N)$  time, where  $N$  is the number of unique characters. For a standard English alphabet,  $N$  is small, so the tree builds almost instantaneously. However, if you were compressing a file with Unicode or UTF-8 characters,  $N$  could be in the thousands, making the efficiency of the priority queue vital.

Another factor to consider is the 'Prefix Property.' Because no code is a prefix of another, your 'decode' function can navigate the tree bit-by-bit. If it sees a '0', it goes left; if it sees a '1', it goes right. The moment it hits a leaf node, it has successfully found a character. This is why Huffman coding is called a 'Prefix Code' or 'Variable-Length Code.' It eliminates the need for any markers or spaces between encoded characters, which is where the real space-saving happens.

If you want to take your project to the next level, try implementing 'Canonical Huffman Coding.' In a canonical tree, the codes are assigned such that all codes of a given length are consecutive binary numbers. This allows you to store the entire dictionary in just a few bytes—only storing the lengths of the codes rather than the codes themselves. This would solve your
```

*Fig 5: Input Text to be compressed*

A screenshot of a terminal window with a dark background and light-colored text. The window title bar at the top shows three colored circles (red, yellow, green) on the left and the filename 'restored\_input.txt' on the right. The text inside the terminal is a multi-paragraph document about Huffman coding and information entropy. The text is as follows:

The concept of information entropy is central to the study of data compression. In the context of Huffman coding, entropy defines the absolute limit of how much a file can be shrunk without losing information. When we analyze a text file, we are essentially looking for the probability of each symbol's occurrence. David Huffman's 1952 algorithm remains one of the most elegant solutions to this problem because it achieves the "Shannon Limit" for character-based encoding.

As you run this larger text through your C++ program, notice how the frequency table stays roughly the same size (since there are only about 90-100 unique ASCII characters in English), but the body of the compressed data grows linearly. This is the moment where your compression ratio will drastically improve. In a small file, the 'header' (your dictionary) might be 50% of the file size. In a 1MB file, that same header becomes 0.01% of the file size.

Consider the following technical aspects of your implementation. You are currently using a 'priority\_queue' with a custom comparator. This is the most efficient way to build the tree, operating in  $O(N \log N)$  time, where  $N$  is the number of unique characters. For a standard English alphabet,  $N$  is small, so the tree builds almost instantaneously. However, if you were compressing a file with Unicode or UTF-8 characters,  $N$  could be in the thousands, making the efficiency of the priority queue vital.

Another factor to consider is the 'Prefix Property.' Because no code is a prefix of another, your 'decode' function can navigate the tree bit-by-bit. If it sees a '0', it goes left; if it sees a '1', it goes right. The moment it hits a leaf node, it has successfully found a character. This is why Huffman coding is called a 'Prefix Code' or 'Variable-Length Code.' It eliminates the need for any markers or spaces between encoded characters, which is where the real space-saving happens.

If you want to take your project to the next level, try implementing 'Canonical Huffman Coding.' In a canonical tree, the codes are assigned such that all codes of a given length are consecutive binary numbers. This allows you to store the entire dictionary in just a few bytes—only storing the lengths of the codes rather than the codes themselves. This would solve your

*Fig 6: Text restored after decompression*