Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report on

“FileZipper:Huffman File Compressor/Decompressor”

[Code No: COMP 202]

(For partial fulfilment of I/II Year/Semester in Computer Engineering)

Submitted by

Saksham Dallakoti (Reg No: 037965-24)

Submitted to

Er. Sagar Acharya
Department of Computer Science and Engineering

Submission Date: 25/02/2026

Acknowledgement

I would like to express my sincere gratitude to everyone who contributed to the successful completion of this mini project, "FileZipper: A Huffman Coding-Based Compression Utility".

First and foremost, I extend my deepest thanks to my project supervisor, Er. Sagar Acharya, from the Department of Computer Science and Engineering, Kathmandu University. His guidance on data structures and the complexities of greedy algorithms was invaluable throughout the development of this project. His encouragement to explore the practical applications of prefix-free encoding helped me gain a deeper understanding of binary trees, priority queues, and bitwise file I/O operations.

I am also thankful to the Department of Computer Science and Engineering, Kathmandu University, for providing the necessary resources and a supportive learning environment that made it possible to complete this project successfully.

Finally, I would like to thank my family and friends for their constant encouragement and feedback during the debugging and testing phases of this application.

Saksham Dallakoti

Regd. No. 037965-24

24th February 2026

Abstract

This project presents FileZipper, a command-line utility developed to implement efficient lossless data compression using Huffman coding. By leveraging Data Structures and Algorithms (DSA) concepts, specifically frequency tables and min-heaps, the system reduces the storage size of text files. The application provides a user-friendly menu for compressing and decompressing files while displaying real-time statistics such as the number of characters processed and the space-saving ratio achieved. This project demonstrates how fundamental DSA concepts like binary trees and priority queues can be applied to create a practical software tool.

Keywords: Huffman Coding, Data Compression, FileZipper, C++, Greedy Algorithm, Min-Heap, Binary Tree, Lossless Compression.

Table of Contents

Acknowledgement i
Abstract..ii
Table of Contents.. iii
List of Figures...iv
Abbreviations..v
Chapter 1: Introduction..1
1.1 Background..1
1.2 Objectives 1
1.3 Motivation and Significance 1
Chapter 2: Related Works..2
Chapter 3: Methodology 3
3.1System Overview 3
3.2 Frequency Analysis with Unordered Map..3
3.3 Huffman Tree Construction via Min-Heap 3
3.4 Recursive Code Generation...4
3.5 Encoding, Bit Packing, and Binary File I/O..5
3.6 Decompression…………..6
Chapter 4: Discussion on Achievement...8
4.1 Achievement Overview...8
4.2 Features Implemented 8
4.3 Challenges and Difficulties...9
Chapter 4: Conclusion..10
Chapter 5: References...11

List of Figures

Figure 1: Sample Huffman Tree…………………………......................................4
Figure 2: Code Table CSV output …………………...5
Figure 3: Compression Statistics Output on a Sample Text File.............................6
Figure 4: Decompresses File Verification Output…...7

Abbreviations

CLI	Command-Line Interface
DSA	Data Structures and Algorithms
C++	C Plus Plus (Programming Language)
I/O	Input / Output
O(n)	Order of n (Big-O Notation)
O(1)	Order of Constant Time
O(log n)	Order of Logarithmic Time
CSV	Comma-Separated Values
EOF	End of File
ASCII	American Standard Code for Information Interchange

Chapter 1: Introduction

1.1 Background

In the modern era of "Big Data," compression is a critical requirement for saving storage and reducing transmission latency. While many users use commercial tools like WinRAR or 7-Zip, the underlying logic of how a character is transformed into a bitstream is often abstracted away. This project aims to implement the Huffman algorithm, a cornerstone of data compression to provide a transparent view of how character frequencies dictate the structure of an optimal binary tree.

1.2 Objectives

1. To implement a lossless compression system using the Huffman coding greedy algorithm.
2. To manage character frequencies effectively using std::unordered_map and std::priority_queue.
3. To provide a command-line interface (CLI) for seamless file-based compression and decompression.
4. To calculate and display performance metrics, specifically the "Space Saved" ratio.

1.3 Motivation and Significance

The primary motivation for this project was to bridge the gap between theoretical algorithm analysis and physical file manipulation. Implementing a min-heap to build a Huffman tree provides deep insights into tree traversal and memory management in C++. This utility is significant as it transforms the abstract concept of "bits and bytes" into a tangible tool that reduces actual file sizes on a disk.

Chapter 2: Related Works

Several compression methodologies and tools influenced the development of FileZipper:

2.1 Lempel-Ziv-Welch (LZW)

Lempel-Ziv-Welch (LZW): A dictionary-based lossless compression algorithm used in GIF and TIFF files. While powerful, it differs from Huffman as it focuses on repeating sequences rather than individual character frequencies.

2.2 Run-Length Encoding (RLE)

RLE is one of the simplest forms of lossless data compression. Unlike Huffman Coding, which reduces size based on the statistical frequency of characters, RLE works by replacing "runs" of identical data values with a single value and a count (e.g., "AAAAA" becomes "5A"). While RLE is highly effective for data with many consecutive repetitions such as simple icons or system log files it is less efficient than Huffman Coding for standard text files where characters are frequent but rarely consecutive. Many modern systems use RLE as a "pre-processing" step before applying more complex algorithms like Huffman or LZW.

Chapter 3: Methodology

3.1 System Overview

The system consists of two main modules: Compression and Decompression. Input text undergoes frequency analysis to build a tree, and then encoded bits are used to represent characters.

3.2 Frequency Analysis with Unordered Map

The first step of compression is buildFrequencyTable(), which iterates over every character in the input string and increments its count in an unordered_map<char, int>. This hash map provides O(1) average-case lookup and insertion for each character, making the full frequency scan O(n) where n is the input length. The unordered_map was selected over a plain array indexed by ASCII value because it is more expressive, handles the full char range without manual index arithmetic, and makes the intent of the code immediately clear. The resulting frequency table serves as the sole input to the tree-building step.

3.3 Huffman Tree Construction via Min-Heap

The HuffmanTree::build() method constructs the Huffman Tree using a greedy algorithm backed by a min-heap priority queue. Every character in the frequency table is wrapped in a HuffmanNode struct containing the character, its frequency, and left/right child pointers, then pushed into a priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> where the custom Compare functor orders nodes by ascending frequency. At each iteration, the two nodes with the smallest frequencies are popped, merged into a new internal node whose frequency is their sum and whose data field is set to the null character '\0', and the merged node is pushed back into the heap. This process repeats until only one node remains, which becomes the root of the Huffman Tree. The total number of merge operations is n-1 where n is the number of distinct characters, giving O(n log n) overall complexity due to the O(log n) heap rebalancing on each push and pop.

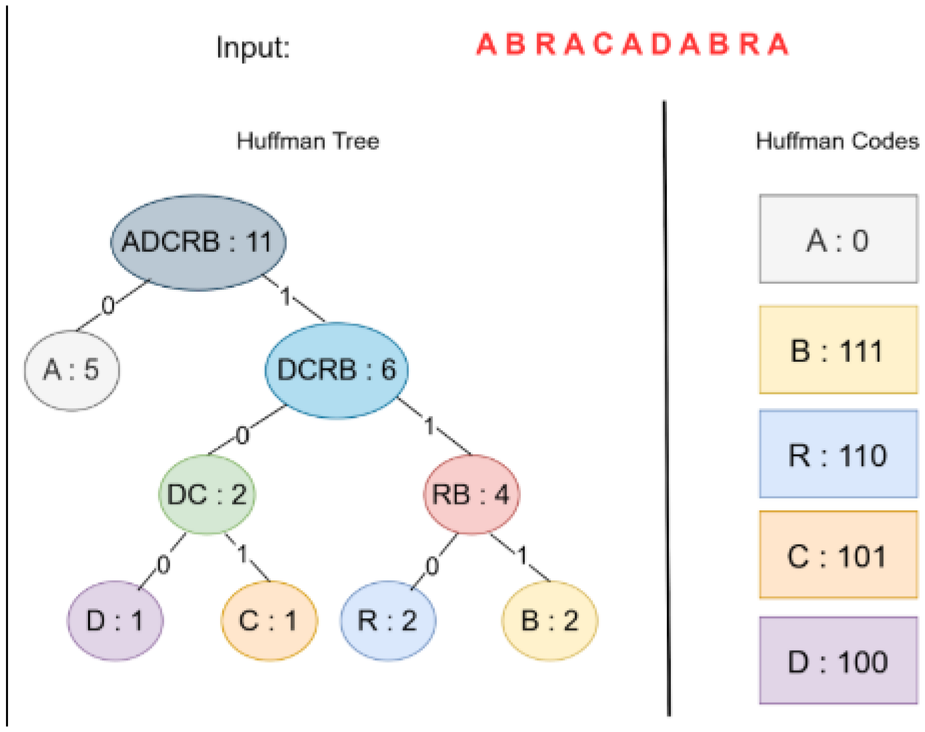
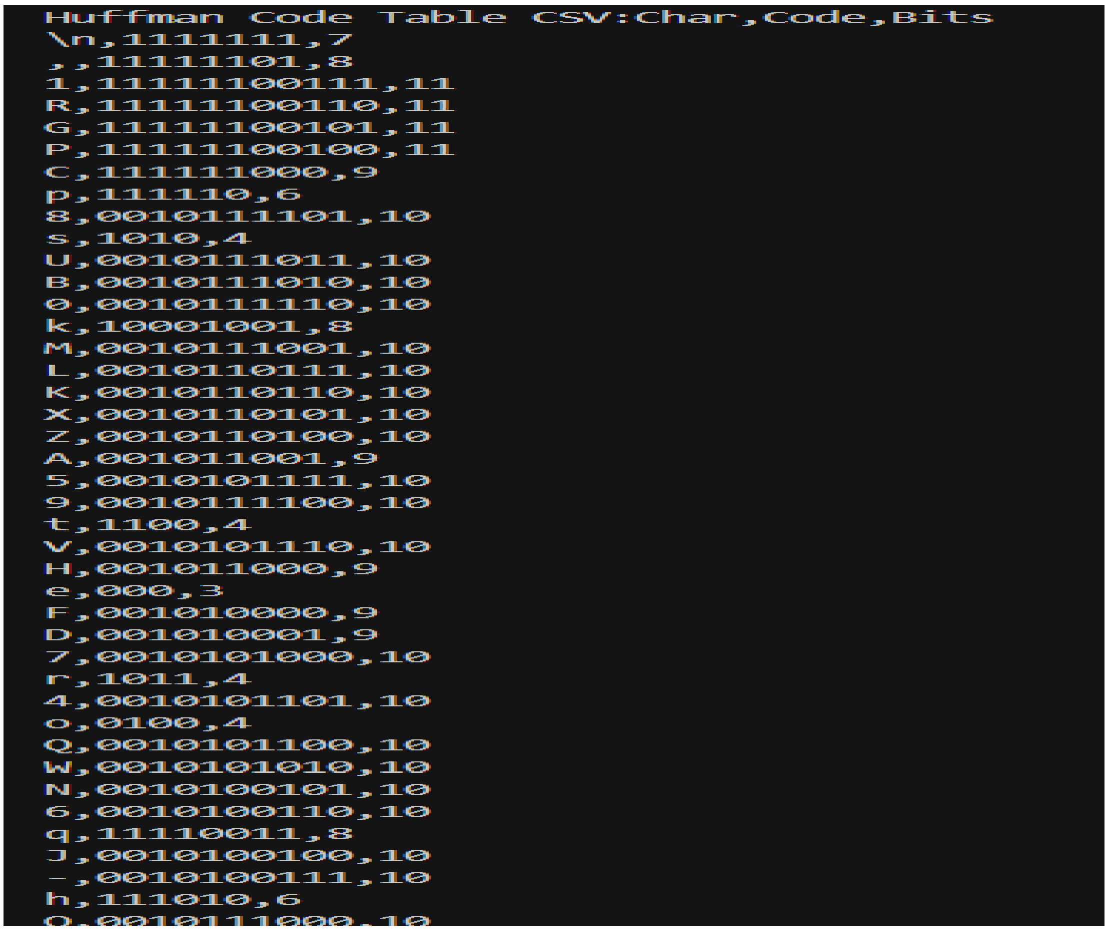


Figure 1: Sample Huffman Tree

3.4 Recursive Code Generation

After the tree is built, generateCodes() performs a recursive preorder traversal starting from the root. At each node, the current binary code string is extended with '0' when descending to the left child and '1' when descending to the right child. When a leaf node is reached (both children are null pointers), the accumulated code string is stored in two maps: codes[node->data] = code for encoding and reverse[code] = node->data for decoding. This dual-map design avoids repeated tree traversal during the character-by-character encoding step, keeping encode() at O(n * L) where L is the average code length. All node memory is allocated on the heap using new and freed in postorder by freeTree() in the destructor, demonstrating manual memory management.

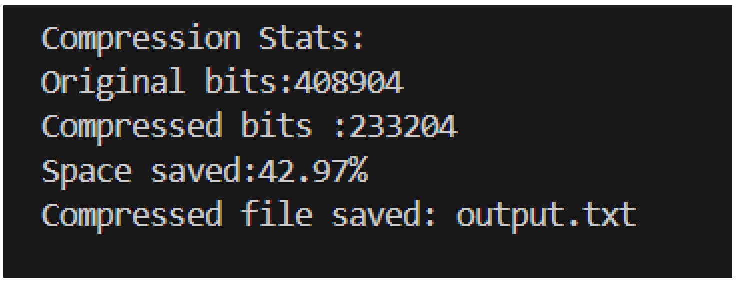


```
Huffman Code Table CSV:Char,Code,Bits
\n,1111111,7
,,11111101,8
1,11111100111,11
R,11111100110,11
G,11111100101,11
P,11111100100,11
C,111111000,9
p,111110,6
8,0010111101,10
s,1010,4
U,0010111011,10
B,0010111010,10
0,0010111110,10
k,10001001,8
M,0010111001,10
L,0010110111,10
K,0010110110,10
X,0010110101,10
Z,0010110100,10
A,001011001,9
5,0010101111,10
9,0010111100,10
t,1100,4
V,0010101110,10
H,001011000,9
e,000,3
F,001010000,9
D,001010001,9
7,0010101000,10
r,1011,4
4,0010101101,10
o,0100,4
Q,0010101100,10
W,0010101010,10
N,0010100101,10
6,0010100110,10
q,11110011,8
J,0010100100,10
-,0010100111,10
h,111010,6
O,0010111000,10
```

Figure 2: Code Table CSV Output Showing Character Codes and Bit Lengths

3.5 Encoding, Bit Packing, and Binary File I/O

The encode() method iterates over every character in the input text and concatenates its Huffman code from the codes map, producing a long binary string such as "01000001101110...". This raw bit string is then packed into bytes in saveCompressed(). Since the bit string length may not be a multiple of 8, padding zero bits are appended to complete the final byte. The number of padding bits is computed as (8 - length % 8) % 8 and written to the file so the decompressor knows how many trailing bits to discard. Each 8-character substring is converted to a byte using std::bitset<8> and static_cast<char>, and written to the file with out.write(). The binary output file uses the following format: [4-byte table size][char + 4-byte freq pairs][4-byte padding count][packed bytes]. This self-contained format allows decompression without any external metadata file.

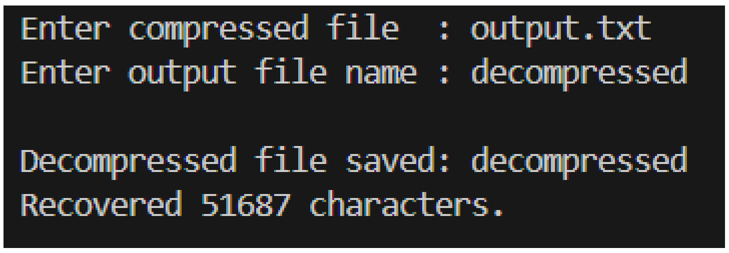


```
Compression Stats:
Original bits:408904
Compressed bits :233204
Space saved:42.97%
Compressed file saved: output.txt
```

Figure 3: Compression Statistics Output on a Sample Text File

3.6 Decompression

The loadCompressed() function reads the binary file and reconstructs the frequency table by reading the table size first, then iterating to read each character-frequency pair. It then reads the padding count, reads all remaining bytes, and converts each byte back to an 8-bit string using std::bitset<8>. The padding bits are stripped from the end of the reconstructed bit stream. The HuffmanTree is then rebuilt from the frequency table using the same build() method, producing an identical tree to the one used during compression. The decode() method traverses the tree bit by bit: a '0' moves to the left child and a '1' moves to the right child. When a leaf node is reached, its character is appended to the output and traversal restarts from the root. This produces the original text exactly.



```
Enter compressed file  : output.txt
Enter output file name : decompressed

Decompressed file saved: decompressed
Recovered 51687 characters.
```

Figure 4: Decompressed File Verification Output

Chapter 4: Discussion on Achievement

This chapter presents the results obtained from testing the Huffman File Compressor, reflects on the achievements of the implementation, discusses the features delivered, and describes the key challenges encountered during development along with how they were resolved.

4.1 Achievement Overview

The project successfully delivered a functional CLI-based file compression tool using standard C++17. The system provides a complete pipeline for Huffman-based compression and decompression, verified through a robust menu-driven interface. By correctly implementing character frequency mapping and deterministic Huffman Tree construction, the application ensures that all generated binary codes are prefix-free and logically sound. To maintain data integrity, the implementation includes a padding mechanism to handle bit-lengths that are not multiples of eight, allowing for a byte-for-byte identical reconstruction of the original file upon decompression.

4.2 Features Implemented

The following features were successfully implemented and verified in the final application:

- Lossless Compression: Ensures the recovered file is identical to the original.
- Custom Min-Heap Comparator: A Compare struct ensures the priority queue functions as a min-heap based on character frequency.
- Tree Visualization: Includes a printTree method that displays the generated Huffman tree rotated 90° for debugging purposes.
- Space Statistics: Calculates and displays the "Space Saved" percentage after compression.

4.3 Challenges and Difficulties

The implementation of the FileZipper presented several technical challenges, primarily regarding binary serialization and memory safety. A significant hurdle was mastering Binary I/O operations, specifically using reinterpret_cast<char*> to write non-character types like integers to the output stream. This required a deep understanding of C++ pointer aliasing rules to treat memory as raw bytes for serialization. Additionally, managing Bit-Padding proved complex; since encoded bit-lengths are rarely multiples of eight, a padding formula was implemented to fill the final byte. Storing this padding count was essential to prevent the decompressor from generating "spurious" characters at the end of the file.

Finally, Memory Management of dynamically allocated tree nodes required careful design. Because the Huffman Tree is built using raw pointers and the new keyword, a post-order traversal was implemented in the freeTree() function. This ensures that child nodes are deleted before their parents, preventing memory leaks by ensuring no node becomes unreachable before it is deallocated.

Chapter 4: Conclusion

This project successfully demonstrates the integration of an unordered map, a min-heap, and a binary tree into a functional CLI application that solves the real-world problem of lossless data compression. By utilizing Huffman Coding, the application achieves efficient character mapping and prefix-free encoding, resulting in significant space savings. The implementation—built entirely in standard C++17—features a self-contained binary file format with an embedded frequency table, ensuring files are independently decompressable without external metadata.

Despite its success, the application has limitations, such as high memory consumption due to loading entire files into RAM and a lack of cross-platform architectural portability. Future enhancements could include streaming data in chunks to handle larger files, adopting fixed-width integer types for compatibility, and implementing adaptive Huffman coding to eliminate the need for an embedded frequency table. Overall, the project provides a robust foundation for understanding the practical intersection of greedy algorithms and complex data structures.

Chapter 5: References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Huffman, D. A. (1952). *A method for the construction of minimum-redundancy codes*. Proceedings of the IRE, 40(9), 1098–1101

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison Wesley.

Sayood, K. (2017).*Introduction to Data Compression* (5th ed.). Morgan Kaufmann.