Bahria University,

Karachi Campus

COURSE: CSC-221 DATA STRUCTURES AND ALGORITHM
TERM: FALL 2025, CLASS: BSE-3(C)

ZipIt Smart

Group Members	**Enrollment No.**
Muhammad Hammad Asher	02-131242-066
Muhammad Affan Bin Aamir	02-131242-064
Syed Shayan Agha	02-131242-093
Samiullah Baig	02-131242-012

Engr. Majid/Engr. Saniya Sarim

Signed **Remarks:** **Score:**

Contents

Introduction And Problem

This project named ZipIT Smart aims to develop a C# based project, using algorithm of Huffman coding to compress and decompress different types of files. As the data storage and data transmission is the major problem due to large size of files, this tool will help reduce the size of different types of files for efficient data storage faster transfer of files without any delay or lose of original data. This tool implements the technique of converting the characters to shorter binary codes. Users can compress standard files (.txt, .doc, .docx, .ppt, .pptx, .pdf), images (.jpg, .png, .bmp, .gif, .tiff) and folders. This tool also supports decompression for any .huff compressed file.

Paradigms

1. Procedural Programming Paradigm

This paradigm focuses on step by step execution of instructions.

Usage in Huffman Program

- Reading files images or folders
- Calculating frequency of bytes
- Encoding and decoding data
- Writing compressed or decompressed output

The Huffman algorithm itself is naturally procedural because it follows a fixed sequence of operations from input to output.

2. Object Oriented Programming Paradigm

This paradigm is based on classes objects encapsulation and abstraction.

Usage in Huffman Program

- HuffmanNode class to store character frequency left and right nodes
- HuffmanTree class to build and traverse the tree
- HuffmanCompressor class to handle compression
- HuffmanDecompressor class to handle decompression
- FileHandler and FolderHandler classes

This paradigm improves modularity code reuse and maintainability.

3. Divide and Conquer Paradigm

This paradigm breaks a big problem into smaller subproblems.

Usage in Huffman Program

- Splitting data into individual symbols or bytes
- Building the Huffman tree by repeatedly combining smaller trees
- Encoding and decoding performed symbol by symbol

The Huffman tree construction is a classic example of divide and conquer.

4. Greedy Programming Paradigm

This paradigm chooses the best local option at each step.

Usage in Huffman Program

- Always selecting two nodes with minimum frequency
- Merging them to build the Huffman tree

Huffman algorithm is fundamentally greedy and guarantees optimal prefix codes.

5. Data Oriented Paradigm

This paradigm focuses on efficient handling of data.

Usage in Huffman Program

- Frequency tables
- Priority queues
- Binary trees
- Bit streams

Efficient data structures improve compression performance.

6. Structured Programming Paradigm

This paradigm emphasizes clear control structures.

Usage in Huffman Program

- Sequential execution
- Loops for reading data
- Conditional decisions during decoding
- Avoidance of goto statements

This makes the logic easy to follow and debug.

Algorithm and Explanation

Huffman Algorithm is a lossless compression technique. It reduces file size by assigning shorter binary codes to frequently occurring data and longer codes to less frequent data. Since it is lossless, the original data is fully recovered after decompression.

1. Role of the Huffman Class

The Huffman class is the core component that handles both compression and decompression. It performs tasks like

- Reading input data from files, images, or folders
- Calculating frequency of each byte
- Building the Huffman Tree
- Generating binary codes
- Encoding data into compressed form
- Decoding compressed data back to original form

2. File Compression and Decompression

Compression process

- Read file bytes
- Count frequency of each byte
- Build Huffman Tree
- Generate Huffman codes
- Replace bytes with binary codes
- Save compressed file with metadata

Decompression process

- Read compressed file
- Rebuild Huffman Tree using metadata
- Decode binary stream
- Restore original file

3. Folder Compression and Decompression

For folders, the Huffman class works recursively.

- Traverse all files inside the folder

- Compress each file individually
- Store folder structure information
- Combine everything into a single compressed archive

During decompression, the original folder hierarchy and files are restored exactly.

4. Image Compression and Decompression

Images are treated as binary data.

- Read image as raw bytes
- Apply Huffman compression on pixel data
- Preserve image headers separately
- Store compressed image file

After decompression, the image remains visually identical since Huffman is lossless.

Flowchart For Compression:

Start
↓
Read input file or image or folder
↓
Calculate frequency of bytes
↓
Build Huffman Tree
↓
Generate Huffman codes
↓
Encode data using codes
↓
Store metadata and compressed data
↓
End

Flowchart For Decompression:

Start
↓
Read compressed file
↓
Extract Huffman Tree metadata
↓
Rebuild Huffman Tree
↓
Decode bit stream
↓
Restore original data

↓

Save output file or folder or image

↓

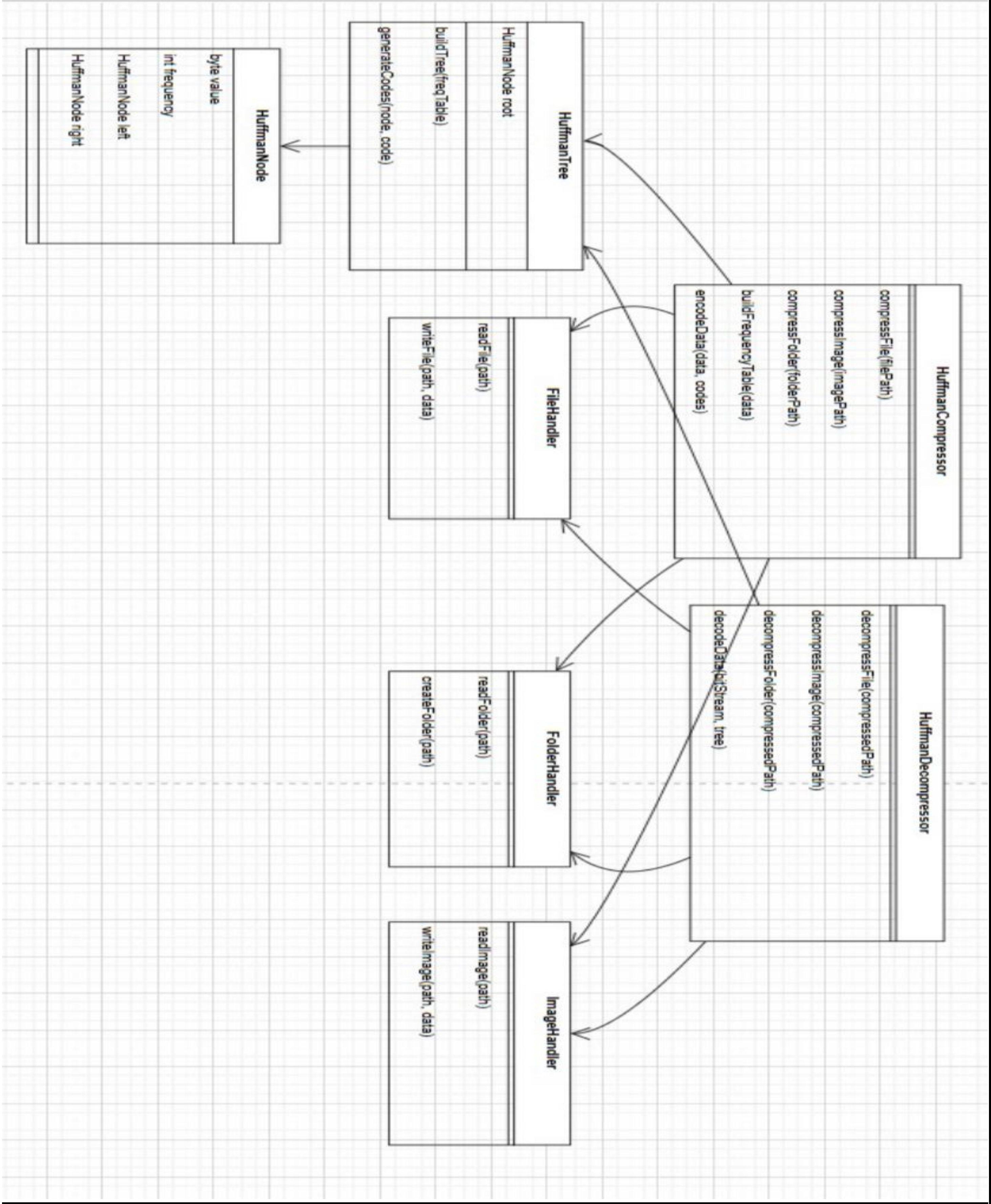
End

Algorithm Code

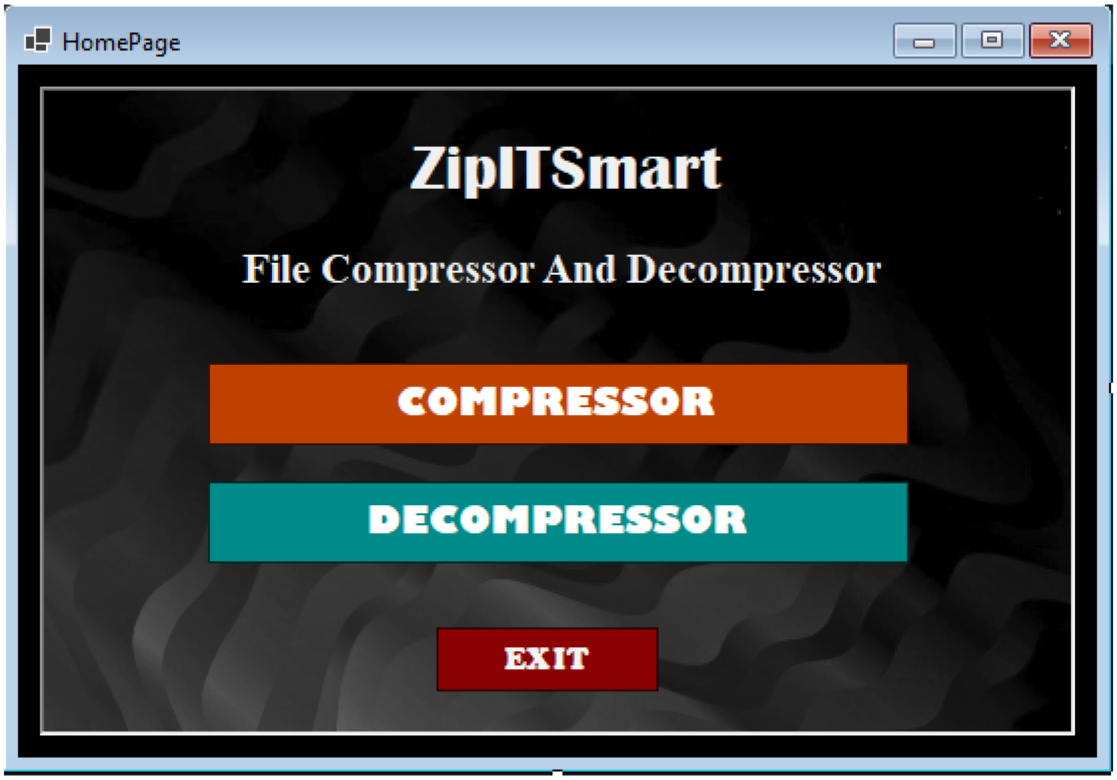
```
namespace ZipITSmart.Core.Huffman
{
    public class HuffmanTree
    {
        public HuffmanNode Root { get; private
set; }

        public Dictionary<byte, string>
Build(byte[] data)
        {
            var frequency = new
Dictionary<byte, int>();
            foreach (byte b in data)
            {
                if (!frequency.ContainsKey(b))
                    frequency[b] = 0;
                frequency[b]++;
            }

            var nodes = frequency
                .Select(kvp => new HuffmanNode
                {
                    Symbol = kvp.Key,
                    Frequency = kvp.Value
                })
                .ToList();
            if (nodes.Count == 1)
            {
                Root = new HuffmanNode
                {
                    Left = nodes[0],
                    Frequency =
nodes[0].Frequency
                };
            }
            else
            {
                while (nodes.Count > 1)
                {
var ordered = nodes.OrderBy(n =>
n.Frequency).ToList();

                    var left = ordered[0];
                    var right = ordered[1];

var parent = new HuffmanNode
                    {
                        Left = left,
                        Right = right,
Frequency = left.Frequency + right.Frequency
                    };
                    nodes.Remove(left);
                    nodes.Remove(right);
                    nodes.Add(parent);
                }
                Root = nodes[0];
            }
    var codes = new Dictionary<byte, string>();
            GenerateCodes(Root, "", codes);
            return codes;
        }
        private void GenerateCodes(HuffmanNode
node, string code, Dictionary<byte, string>
table)
        {
            if (node == null)
                return;
            if (node.IsLeaf &&
node.Symbol.HasValue)
            {
                table[node.Symbol.Value] =
code.Length > 0 ? code : "0";
            }
GenerateCodes(node.Left, code + "0", table);
GenerateCodes(node.Right, code + "1", table);
        }
    }
}
    public static class HuffmanService
    {
public static byte[] Compress(byte[] data)
        {
if (data == null || data.Length == 0)
return Array.Empty<byte>();
            var tree = new HuffmanTree();
            var codeTable = tree.Build(data);

            var bitString = new
StringBuilder();
            foreach (byte b in data)
                bitString.Append(codeTable[b]);
int padding = (8 - bitString.Length % 8) % 8;
            bitString.Append('0', padding);

            var bytes = new List<byte>();
            for (int i = 0; i <
bitString.Length; i += 8)
```

```
            {
bytes.Add(Convert.ToByte(bitString.ToString(i,
8), 2));
            }
            using (var ms = new MemoryStream())
            using (var bw = new
BinaryWriter(ms))
            {
                WriteTree(bw, tree.Root);
                bw.Write((byte)padding);
                bw.Write(bytes.ToArray());
                return ms.ToArray();
            }
        }
        public static byte[] Decompress(byte[]
compressed)
        {
            if (compressed == null ||
compressed.Length == 0)
                return Array.Empty<byte>();

            using (var ms = new
MemoryStream(compressed))
            using (var br = new
BinaryReader(ms))
            {
                var root = ReadTree(br);
                int padding = br.ReadByte();

                var bits = new StringBuilder();
                while (ms.Position < ms.Length)
                {

bits.Append(Convert.ToString(br.ReadByte(),
2).PadLeft(8, '0'));
                }

                if (padding > 0)
                    bits.Length -= padding;

                var output = new List<byte>();
                var current = root;

                foreach (char bit in
bits.ToString())
                {
                    current = bit == '0' ?
current.Left : current.Right;
                    if (current.IsLeaf)
                    {

output.Add(current.Symbol.Value);
                        current = root;
                        }
                    }

                    return output.ToArray();
                }
            }
        private static void
WriteTree(BinaryWriter bw, HuffmanNode node)
        {
            if (node.IsLeaf)
            {
                bw.Write(true);
                bw.Write(node.Symbol.Value);
            }
            else
            {
                bw.Write(false);
                WriteTree(bw, node.Left);
                WriteTree(bw, node.Right);
            }
        }

        private static HuffmanNode
ReadTree(BinaryReader br)
        {
            bool isLeaf = br.ReadBoolean();
            if (isLeaf)
            {
                return new HuffmanNode
                {
                    Symbol = br.ReadByte()
                };
            }

            return new HuffmanNode
            {
                Left = ReadTree(br),
                Right = ReadTree(br)
            };
        }
    }
}
public class HuffmanNode
{
    public byte? Symbol { get; set; }
    public int Frequency { get; set; }

    public HuffmanNode Left { get; set; }
    public HuffmanNode Right { get; set; }

    public bool IsLeaf => Left == null && Right
== null;
}
```

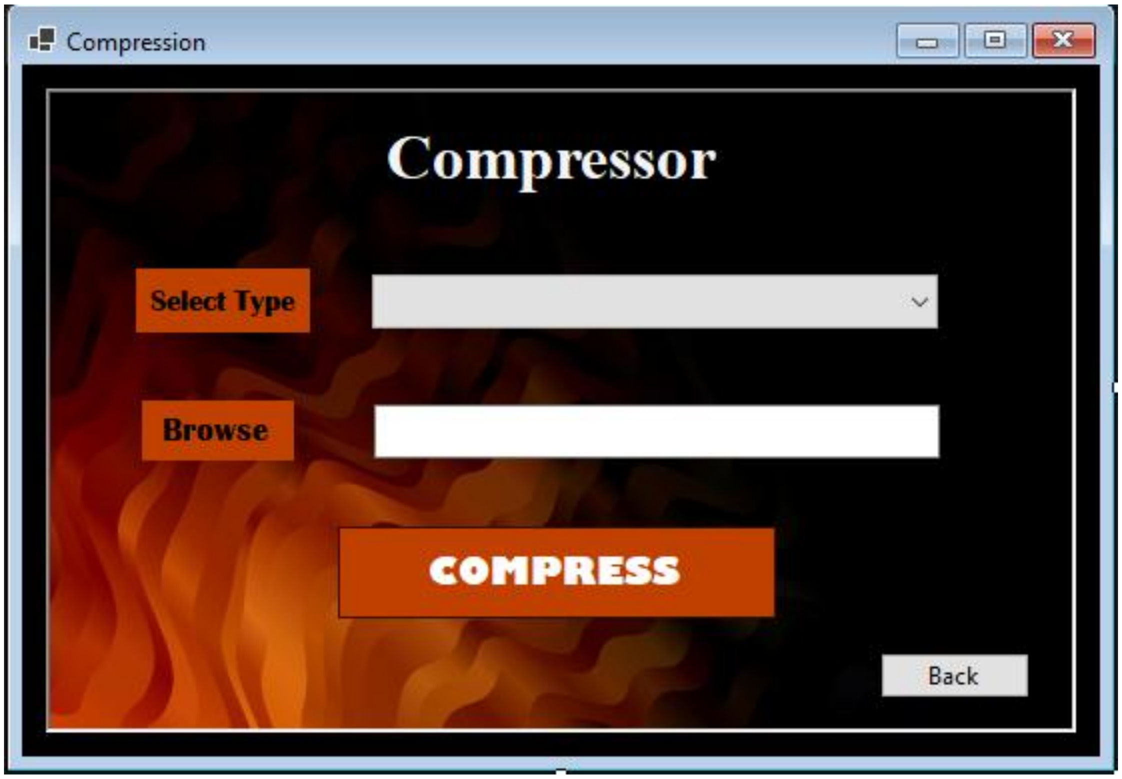
Sequence Diagram



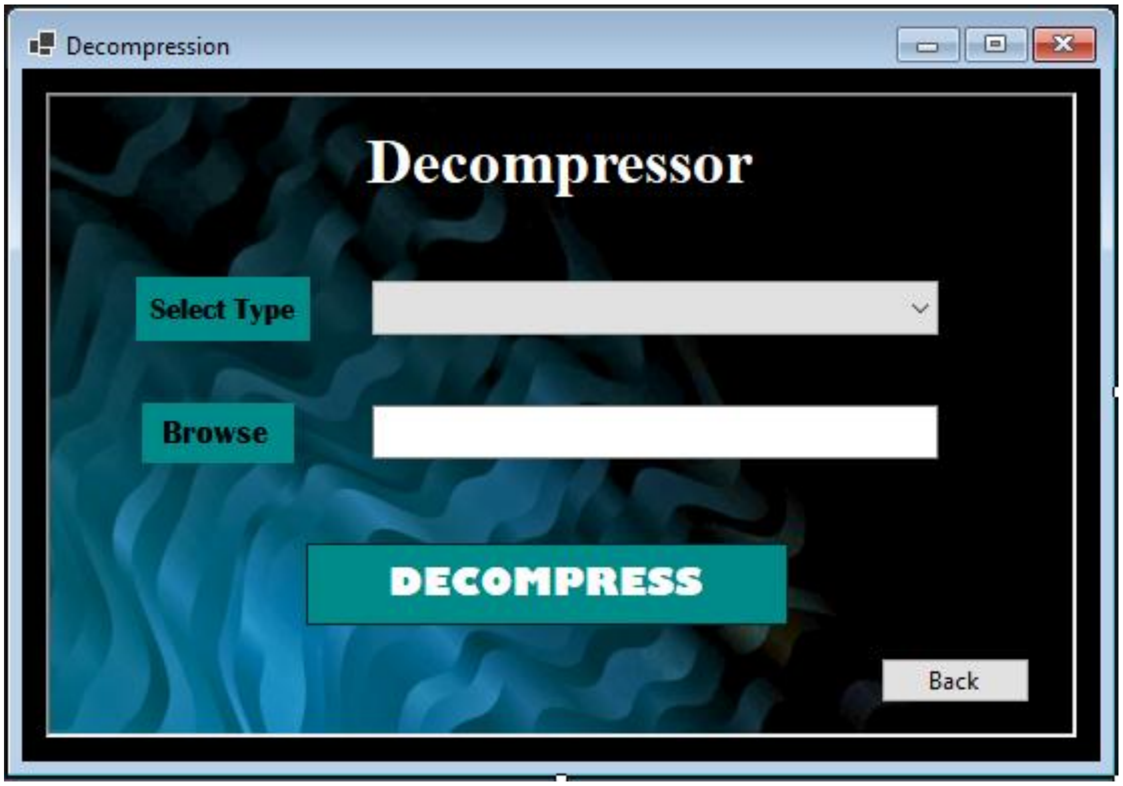
Interfaces



Home page with Compression/Decompression options.



Compression page with File/Image/Folder options.



Decompression page with folder/file selection.

Conclusion

This project bridges the gap between theoretical data structures and real-world software applications. By implementing Huffman Coding in C#, the team demonstrates how priority queues and binary trees can be utilized to optimize data storage. The modular distribution of tasks, from frequency analysis to UI development which ensures a comprehensive system capable of efficient file management and performance reporting.