

# **SPL-1 Project Report**

## **Image Compression Tool**

### **Submitted by**

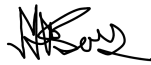
Kazi Farhana Faruque  
BSSE Roll No.: 1506  
Exam Roll: 231111

Registration No.: 2022116303

### **Submitted to**

Dr. B M Mainul Hossain  
Director & Professor

Institute of Information Technology  
University of Dhaka



**Institute of Information Technology**  
University of Dhaka

**25 March 2025**

# **SPL-1 Project Report**

## **Image Compression Tool**

### **Submitted to**

Dr. B M Mainul Hossain  
Director & Professor

Institute of Information Technology  
University of Dhaka

Supervisor Signature:

---

### **Submitted by**

Kazi Farhana Faruque  
BSSE Roll No.: 1506  
BSSE Session: 2022-23

Signature:

---

## Table of Contents

1. Introduction .....	4
1.2. Motivation Behind Image Compression .....	4
1.3. Importance of Lossless Compression .....	4
2. Background of the Project.....	5
2.1 Image File Structure .....	5
2.1.1. <i>PGM Image File Format</i> .....	5
2.1.2. <i>BMP Image File Format</i> .....	5
2.2 Algorithms .....	5
2.2.1. <i>Huffman Encoding</i> .....	5
2.2.2. <i>Run Length Encoding</i> .....	8
2.2.3. <i>Lempel-Ziv-Welch (LZW)</i> .....	8
3. Description of the Project .....	11
3.1 Image Read and Write .....	11
3.2 Features and Functionality .....	11
3.2.1. <i>Compression</i> .....	12
3.2.2. <i>Decompression</i> .....	18
4. Implementation and Testing .....	23
5. User Interface .....	24
6.Challenges Faced .....	26
6.1 Technical Difficulties .....	26
6.2 Solutions Implemented .....	26
7. Conclusion .....	27
7.1 Lessons Learned .....	27
7.2 Future Enhancements .....	27
8.Reference .....	28

# 1. Introduction

Compression is the process of reducing the size of data, files, or signals while maintaining their essential information. It reduces the size of files, allowing more data to be stored on a device or server. Smaller file sizes mean faster upload/download speeds, improving data transfer over networks. It reduces costs associated with storage hardware and data transmission. Many applications use compression to make files easier to share via email or messaging.

The **"Image Compression Tool"** is a software project designed to compress and decompress image files in two widely used formats: Portable Gray Map (PGM) and Bitmap (BMP). The tool implements three distinct compression algorithms—Run-Length Encoding (RLE), Lempel-Ziv-Welch (LZW), and Huffman Coding—to reduce the file size of images while preserving their quality for decompression. This project demonstrates the practical application of lossless compression techniques, enabling users to choose an algorithm based on their needs and the characteristics of the input image. The tool is written in C, leveraging file I/O operations and data structures to achieve efficient compression and decompression.

The primary goal of this project is to provide a user-friendly utility that supports both PGM and BMP image formats, offering flexibility in algorithm selection and demonstrating the effectiveness of each method through compression ratios and file size comparisons.

## 1.2. Motivation Behind Image Compression

In today's digital age, the exponential growth of multimedia content—such as images, videos, and audio—has made data compression an indispensable technology. Images, in particular, often contain large amounts of data that can quickly consume storage space and bandwidth. The "Image Compression Tool" addresses this challenge by providing an efficient solution to minimize file sizes without sacrificing quality. By implementing lossless compression techniques like Run-Length Encoding (RLE), Lempel-Ziv-Welch (LZW), and Huffman Coding, this tool showcases how compression can optimize storage and transmission efficiency. Written in C, the project combines algorithmic ingenuity with practical utility, making it a valuable resource for understanding and applying compression in real-world scenarios.

## 1.3. Importance of Lossless Compression

Compression comes in two primary forms: lossy and lossless. While lossy compression discards some data to achieve higher compression ratios, lossless compression ensures that every bit of original information can be perfectly reconstructed. The "Image Compression Tool" focuses exclusively on lossless methods, making it ideal for applications where data integrity is paramount, such as medical imaging, archival storage, and scientific visualization. By supporting PGM and BMP image formats and employing RLE, LZW, and Huffman Coding, this project highlights the balance between file size reduction and quality preservation. Its implementation in C ensures high performance, offering users a hands-on experience with compression techniques that power many modern technologies.

## 2. Background of the Project

### 2.1 Image File Structure

#### 2.1.1. PGM Image File Format

A PGM (Portable Gray Map) image is a simple and widely used file format for grayscale images. PGM images are often used in computer vision, image processing, and scientific applications.

PGM images can have different variations, including PGM P2 (ASCII) and PGM P5 (binary). In both variations, the image data consists of a header section followed by the pixel intensity values.

The header section of a PGM image contains information about the file format, image size, and maximum pixel intensity value. In the P2 (ASCII) format, the header is text-based and includes information such as the "P2" identifier, width and height of the image, and the maximum intensity value. In the P5 (binary) format, the header is stored in binary form, providing the same information in a more compact way.

#### 2.1.2. BMP Image File Format

BMP files are uncompressed raster images, typically resulting in large file sizes. BMP files store pixel data without built-in compression, making them ideal for applying external compression techniques. A BMP (Bitmap) image is a commonly used raster graphics file format for storing digital images. BMP images are uncompressed and store pixel data in a straightforward manner. They can be either monochrome (black and white) or color images.

The structure of a BMP file consists of a file header, an optional bitmap information header, and the pixel data. The file header contains information about the file format and size, while the bitmap information header specifies details about the image, such as its width, height, color depth, and compression method (if any). The pixel data in a BMP image is stored row by row, starting from the bottom-left corner of the image and moving horizontally to the right. Each pixel is represented by a certain number of bits depending on the color depth of the image. For example, in a 24-bit BMP image, each pixel is represented by 3 bytes (8 bits for each color channel: red, green, and blue), resulting in a total of 24 bits per pixel. This allows for a wide range of colors and more realistic representations.

## 2.2 Algorithms

#### 2.2.1. Huffman Encoding

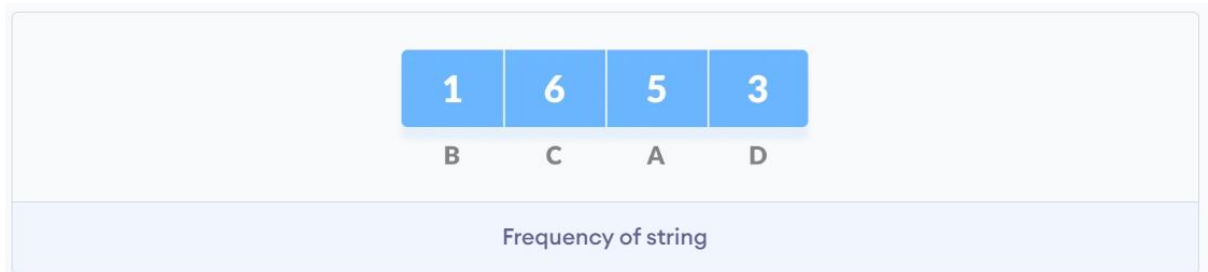
Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.



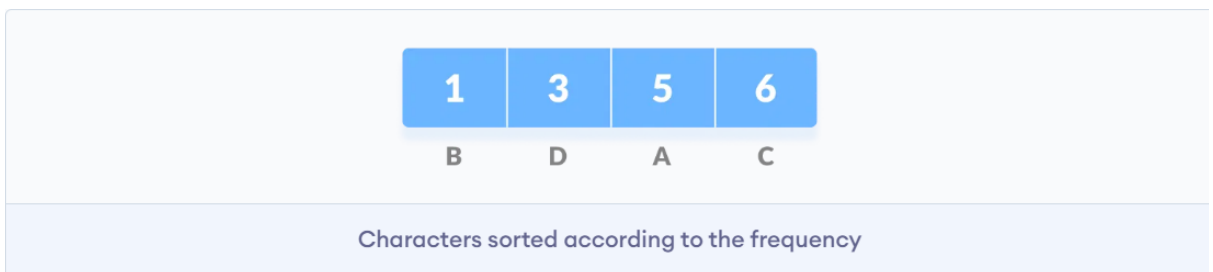
Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of  $8 * 15 = 120$  bits are required to send this string.

Huffman coding is done with the help of the following steps.

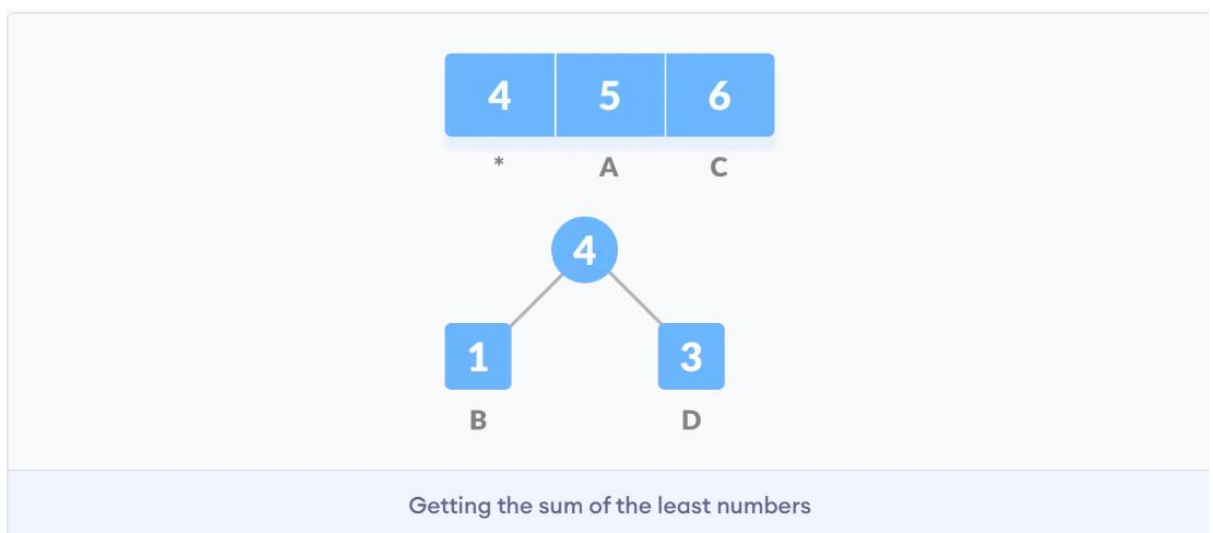
1. Calculate the frequency of each character in the string.



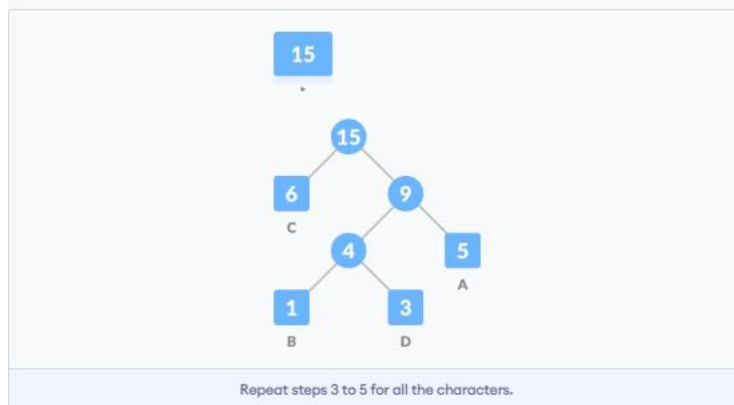
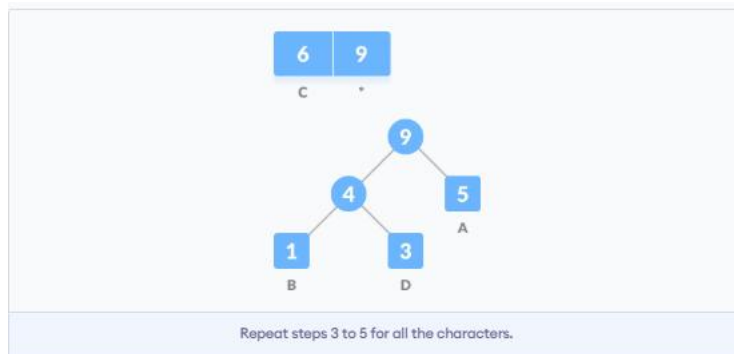
2. Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.



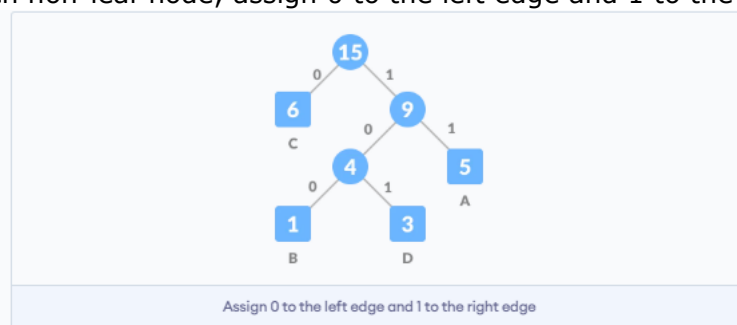
3. Make each unique character as a leaf node.
4. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.



5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies (\* denote the internal nodes in the figure above).
6. Insert node z into the tree.
7. Repeat steps 3 to 5 for all the characters.



8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

Character	Frequency	Code	Size
A	5	11	$5 \times 2 = 10$
B	1	100	$1 \times 3 = 3$
C	6	0	$6 \times 1 = 6$
D	3	101	$3 \times 3 = 9$
$4 \times 8 = 32$ bits	15 bits		28 bits

Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to  $32 + 15 + 28 = 75$ .

### 2.2.2. Run Length Encoding

**Run-length encoding (RLE)** is a form of lossless data compression in which runs of data (consecutive occurrences of the same data value) are stored as a single occurrence of that data value and a count of its consecutive occurrences, rather than as the original run. As an imaginary example of the concept, when encoding an image built up from coloured dots, the sequence "green green green green green green green green green " is shortened to "green x 9". This is most efficient on data that contains many such runs, for example, simple graphic images such as icons, line drawings, games, and animations. For files that do not have many runs, encoding them with RLE could increase the file size.

#### **Encoding algorithm :**

Run-length encoding compresses data by reducing the physical size of a repeating string of characters. This process involves converting the input data into a compressed format by identifying and counting consecutive occurrences of each character. The steps are as follows:

1. Traverse the input data.
2. Count the number of consecutive repeating characters (run length).
3. Store the character and its run length.

#### **Decoding algorithm :**

The decoding process involves reconstructing the original data from the encoded format by repeating characters according to their counts. The steps are as follows:

1. Traverse the encoded data.
2. For each count-character pair, repeat the character count times.
3. Append these characters to the result string.

Consider a screen containing plain black text on a solid white background. There will be many long runs of white pixels in the blank space, and many short runs of black pixels within the text. A hypothetical scan line, with B representing a black pixel and W representing white, might read as follows:

```
WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWW  
WWWWWWWWWWBWWWWWWWWWWWWWWWW
```

With a run-length encoding (RLE) data compression algorithm applied to the above hypothetical scan line, it can be rendered as follows:

```
12W1B12W3B24W1B14W
```

### 2.2.3. Lempel-Ziv-Welch (LZW)

LZW compression works by reading a sequence of symbols, grouping the symbols into strings, and converting the strings into codes. Because the codes take up less space than the strings they replace, we get compression. Characteristic features of LZW includes,



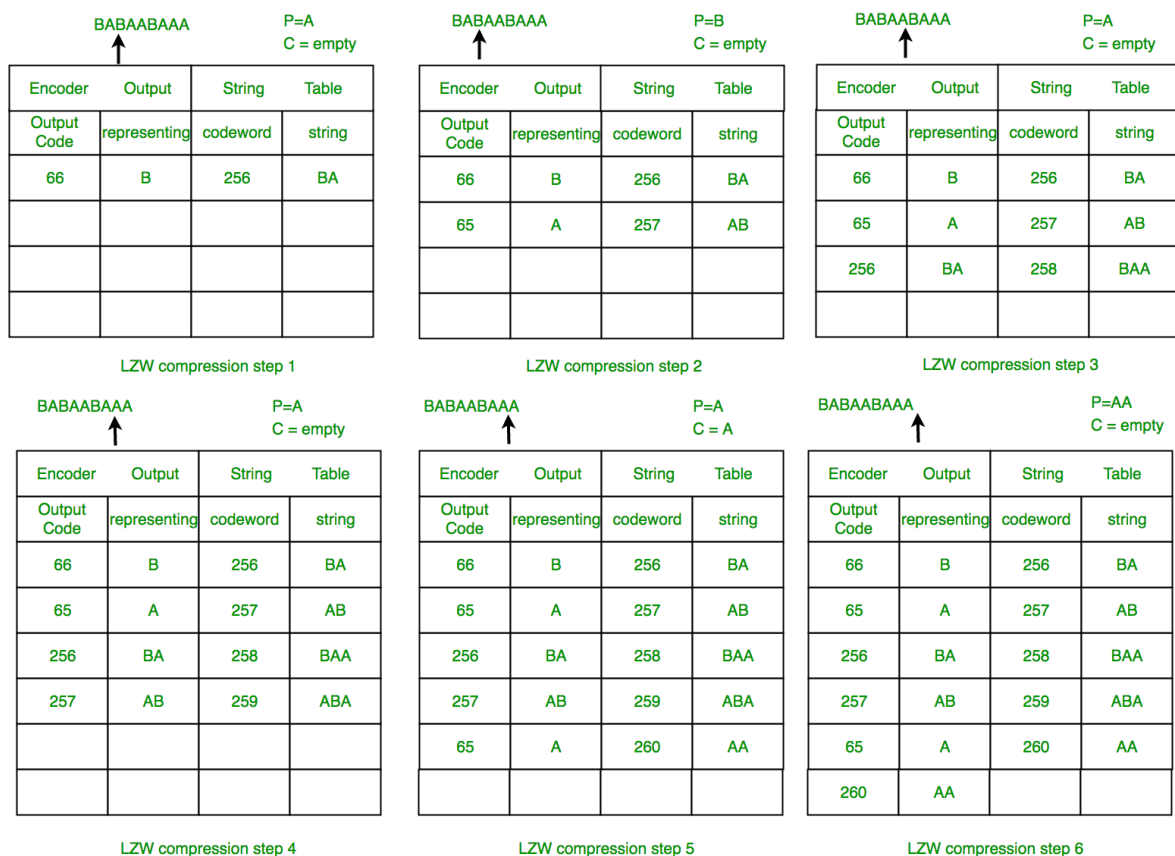
- LZW compression uses a code table, with 4096 as a common choice for the number of table entries. Codes 0-255 in the code table are always assigned to represent single bytes from the input file.
- When encoding begins the code table contains only the first 256 entries, with the remainder of the table being blanks. Compression is achieved by using codes 256 through 4095 to represent sequences of bytes.
- As the encoding continues, LZW identifies repeated sequences in the data and adds them to the code table.
- Decoding is achieved by taking each code from the compressed file and translating it through the code table to find what character or characters it represents.

Example: ASCII code. Typically, every character is stored with 8 binary bits, allowing up to 256 unique symbols for the data. This algorithm tries to extend the library to 9 to 12 bits per character. The new unique symbols are made up of combinations of symbols that occurred previously in the string. It does not always compress well, especially with short, diverse strings. But is good for compressing redundant data, and does not have to save the new dictionary with the data: this method can both compress and uncompress data.

### Compression using LZW

Example 1: Use the LZW algorithm to compress the string: **BABAABAAA**

The steps involved are systematically shown in the diagram below.

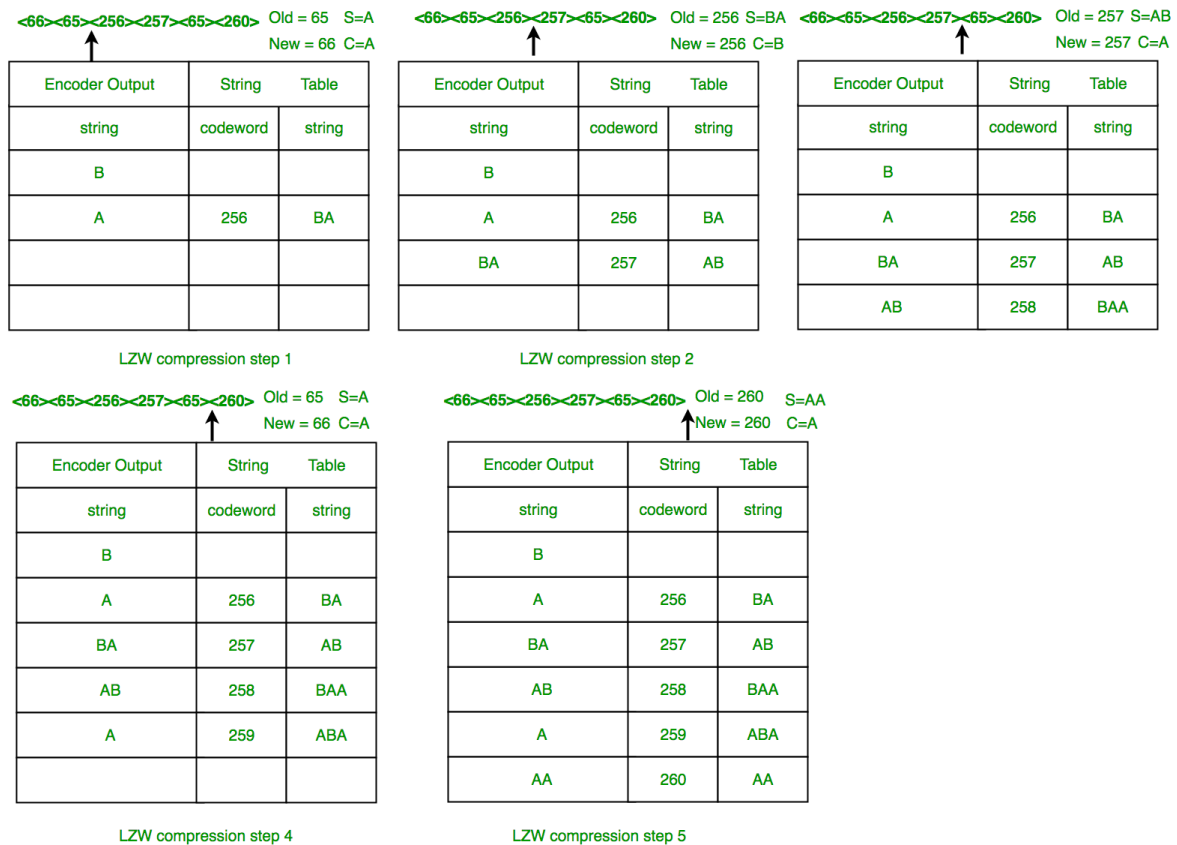


## LZW Decompression

The LZW decompressor creates the same string table during decompression. It starts with the first 256 table entries initialized to single characters. The string table is updated for each character in the input stream, except the first one. Decoding is achieved by reading codes and translating them through the code table being built.

LZW Decompression: Use LZW to decompress the output sequence of  
: <66><65><256><257><65><260>

The steps involved are systematically shown in the diagram below.



In this example, 72 bits are represented with 72 bits of data. After a reasonable string table is built, compression improves dramatically.

**LZW Summary:** This algorithm compresses repetitive sequences of data very well. Since the codewords are 12 bits, any single encoded character will expand the data size rather than reduce it.

## 3. Description of the Project

### 3.1 Image Read and Write

To read and write an image we must know the header structure of that particular image. In our case we used two types of image PGM and BMP these are the header structures of these two images :

Header structure for PGM image.

The header of a PGM image file consists of:

1. First line containing signature of the image file either p2 or p5.
2. Second line is the comment line
3. Third line provides the information about height and the width of the image.
4. Fourth line specifies maximum intensity level of the image.

Data follows the header information and is written in pixel values (in text or binary format). Data is in raster order, which indicates that all data for the first row of the image is written first, then for the second row, and so on. The origin of the coordinate system for a PGM image is located on the top left corner.

2. Header structure for BMP image :

- (a) File Header (14 Bytes).
- (b) Information Header (40 Bytes).
- (c) Color Table (4 \* number of color bytes).
- (d) Pixel Data (variable bytes).

### 3.2 Features and Functionality

The tool includes the following features:

1. Image Loading and Reading (Bitmap, PGM)
2. Image Compression using Huffman Coding, Run-Length Encoding and LZW
3. Bin File Creation
4. Image Decompression using Huffman Coding, Run-Length Encoding and LZW
5. Save Image Functionality

### 3.2.1. Compression

RLE Compression function for PGM Image file :



```
1 unsigned char current = id[0];
2 unsigned char count = 1;
3
4 for (long i = 1; i < tP; i++) {
5     if (id[i] == current && count < 255) {
6         count++;
7     } else {
8         if (fwrite(&current, sizeof(unsigned char), 1, output) != 1 ||
9             fwrite(&count, sizeof(unsigned char), 1, output) != 1) {
10             printf("Error writing RLE pair\n");
11             free(id);
12             fclose(output);
13             return 1;
14         }
15         current = id[i];
16         count = 1;
17     }
18 }
19 if (fwrite(&current, sizeof(unsigned char), 1, output) != 1 ||
20     fwrite(&count, sizeof(unsigned char), 1, output) != 1) {
21     printf("Error writing final RLE pair\n");
22     free(id);
23     fclose(output);
24     return 1;
25 }
```

LZW compression function for PGM Image file :



```
1 DictionaryEntry* dict = (DictionaryEntry*)malloc(MAX_DICT_SIZE * sizeof(DictionaryEntry));
2 int dictSize = MAX_SIZE;
3 for (int i = 0; i < MAX_SIZE; i++) {
4     dict[i].prefix = -1;
5     dict[i].value = (unsigned char)i;
6     dict[i].code = i;
7 }
```



```
1  int code = iD[0];
2      int nextCode = MAX_SIZE;
3      unsigned int bitBuffer = 0;
4      int bits = 0;
5
6      for (long i = 1; i < tP; i++) {
7          unsigned char nextChar = iD[i];
8          int j;
9          for (j = 0; j < dictSize; j++) {
10             if (dict[j].prefix == code && dict[j].value == nextChar) {
11                 code = dict[j].code;
12                 break;
13             }
14         }
15         if (j == dictSize) {
16             bitBuffer = (bitBuffer << 12) | code;
17             bits += 12;
18             while (bits >= 8) {
19                 unsigned char byte = (bitBuffer >> (bits - 8)) & 0xFF;
20                 fwrite(&byte, 1, 1, output);
21                 bits -= 8;
22             }
23             bitBuffer &= (1 << bits) - 1;
24
25             if (dictSize < MAX_DICT_SIZE) {
26                 dict[dictSize].prefix = code;
27                 dict[dictSize].value = nextChar;
28                 dict[dictSize].code = nextCode++;
29                 dictSize++;
30             }
31             code = nextChar;
32         }
33     }
34     bitBuffer = (bitBuffer << 12) | code;
35     bits += 12;
36     while (bits > 0) {
37         unsigned char byte = (bitBuffer >> (bits - 8)) & 0xFF;
38         fwrite(&byte, 1, 1, output);
39         bits -= 8;
40         if (bits < 0) break;
41     }
42
```

## Huffman compression function for PGM Image file :



```
1 unsigned int freq[MAX_SIZE] = {0};
2     for (long i = 0; i < tP; i++) {
3         freq[id[i]]++;
4     }
5
6     Node* root = buildHuffmanTree(freq);
7     if (!root) {
8         printf("Failed to build Huffman tree during compression\n");
9         free(id);
10        fclose(output);
11        return 1;
12    }
13
14    char codes[MAX_SIZE][MAX_SIZE] = {0};
15    int lengths[MAX_SIZE] = {0};
16    char code[MAX_SIZE];
17    generateCodes(root, code, 0, codes, lengths);
```



```
1 unsigned int bitBuffer = 0;
2     int bits = 0;
3     for (long i = 0; i < tP; i++) {
4         unsigned char pixel = id[i];
5         for (int j = 0; j < lengths[pixel]; j++) {
6             bitBuffer = (bitBuffer << 1) | (codes[pixel][j] - '0');
7             bits++;
8             if (bits == 8) {
9                 unsigned char byte = (unsigned char)bitBuffer;
10                if (fwrite(&byte, 1, 1, output) != 1) {
11                    printf("Failed to write compressed data\n");
12                    free(id);
13                    freeHuffmanTree(root);
14                    fclose(output);
15                    return 1;
16                }
17                bitBuffer = 0;
18                bits = 0;
19            }
20        }
21    }
22    if (bits > 0) {
23        bitBuffer <= (8 - bits);
24        unsigned char byte = (unsigned char)bitBuffer;
25        if (fwrite(&byte, 1, 1, output) != 1) {
26            printf("Failed to write final compressed byte\n");
27            free(id);
28            freeHuffmanTree(root);
29            fclose(output);
30            return 1;
31        }
32    }
33
```

RLE compression function for BMP Image file :



```
1  int padding = (4 - ((info.Width * 3) % 4)) % 4;
2      int rowSize = info.Width * 3 + padding;
3
4      fseek(in, file.Offbits, SEEK_SET);
5      unsigned char* pD = malloc(rowSize * info.Height); // pixel data
6      fread(pD, 1, rowSize * info.Height, in);
7
8      RLEEntry entry = {0, 0, 0, 0};
9      unsigned long cS = 0; // compressed size
10
11
12  for (int i = 0; i < info.Height * rowSize; i += 3) {
13      if (i % rowSize >= info.Width * 3) continue;
14      unsigned char b = pD[i];
15      unsigned char g = pD[i + 1];
16      unsigned char r = pD[i + 2];
17      if (entry.count == 0) {
18          entry.count = 1;
19          entry.b = b;
20          entry.g = g;
21          entry.r = r;
22      }
23      else if (entry.b == b && entry.g == g && entry.r == r && entry.count < 255) {
24          entry.count++;
25      }
26      else {
27          fwrite(&entry, sizeof(RLEEntry), 1, out);
28          cS += sizeof(RLEEntry);
29          entry.count = 1;
30          entry.b = b;
31          entry.g = g;
32          entry.r = r;
33      }
34  }
35  if (entry.count > 0) {
36      fwrite(&entry, sizeof(RLEEntry), 1, out);
37      cS += sizeof(RLEEntry);
38  }
```

## LZW compression function for BMP Image file :



```
1 unsigned int out_pos = 0;
2 unsigned short prefix = input[0];
3 unsigned int in_pos = 1;
4
5 while (in_pos < input_size) {
6     unsigned char next_char = input[in_pos++];
7     unsigned short current = 0;
8
9     for (current = 0; current < dict_size; current++) {
10         if (dict[current].prefix == prefix && dict[current].append == next_char) {
11             prefix = current;
12             break;
13         }
14     }
15
16     if (current == dict_size) {
17         output[out_pos++] = prefix & 0xFF;
18         output[out_pos++] = (prefix >> 8) & 0xFF;
19         if (dict_size < MAX_DICT_SIZE) {
20             dict[dict_size].prefix = prefix;
21             dict[dict_size].append = next_char;
22             dict_size++;
23         }
24         prefix = next_char;
25     }
26 }
27
28 output[out_pos++] = prefix & 0xFF;
29 output[out_pos++] = (prefix >> 8) & 0xFF;
30
31 *output_size = out_pos;
32 return output;
```



```
1 unsigned int com_size;
2 unsigned char* compressed = lzw_compress(pD, dS, &com_size);
3 if (!compressed) {
4     free(pD);
5     fclose(fin);
6     return -1;
7 }
8
9 file.Offbits = sizeof(BmpFile) + sizeof(BmpInfo) + sizeof(unsigned int);
10 file.Size = file.Offbits + com_size;
11 info.Compression = 1;
12 info.SizeImage = com_size;
```



## Huffman compression function for BMP Image file :

```
1 HuffmanNode* build_huffman_tree(unsigned int* freq) {
2     HuffmanNode* nodes[MAX_TREE_NODES];
3     int node_count = 0;
4
5     for (int i = 0; i < 256; i++) {
6         if (freq[i]) {
7             nodes[node_count++] = create_node((unsigned char)i, freq[i]);
8         }
9     }
10
11     while (node_count > 1) {
12         int min1 = 0, min2 = 1;
13         if (nodes[min2]->freq < nodes[min1]->freq) {
14             int temp = min1;
15             min1 = min2;
16             min2 = temp;
17         }
18         for (int i = 2; i < node_count; i++) {
19             if (nodes[i]->freq < nodes[min1]->freq) {
20                 min2 = min1;
21                 min1 = i;
22             } else if (nodes[i]->freq < nodes[min2]->freq) {
23                 min2 = i;
24             }
25         }
26
27         HuffmanNode* parent = create_node(0, nodes[min1]->freq + nodes[min2]->freq);
28         parent->left = nodes[min1];
29         parent->right = nodes[min2];
30
31         nodes[min1] = parent;
32         nodes[min2] = nodes[node_count - 1];
33         node_count--;
34     }
35
36     return nodes[0];
37 }
38
```

```
1 void generate_codes(HuffmanNode* root, unsigned int code, int length, unsigned int* codes, int* lengths) {
2     if (!root->left && !root->right) {
3         codes[root->symbol] = code;
4         lengths[root->symbol] = length;
5         return;
6     }
7     if (root->left) {
8         generate_codes(root->left, code << 1, length + 1, codes, lengths);
9     }
10    if (root->right) {
11        generate_codes(root->right, (code << 1) | 1, length + 1, codes, lengths);
12    }
13 }
14
```

```
1 unsigned int freq[256];
2 build_freq_table(pD, dS, freq);
3 HuffmanNode* root = build_huffman_tree(freq);
4
5 unsigned int codes[256] = {0};
6 int lengths[256] = {0};
7 generate_codes(root, 0, 0, codes, lengths);
8
9 file.Offsetbits = sizeof(BmpFile) + sizeof(BmpInfo) + sizeof(unsigned int) + 256 * sizeof(unsigned int);
10 info.Compression = 2;
11 BitBuffer bb;
12 init_bit_buffer(&bb, fout);
13
14 fwrite(&file, sizeof(BmpFile), 1, fout);
15 fwrite(&info, sizeof(BmpInfo), 1, fout);
16 fwrite(&dS, sizeof(unsigned int), 1, fout);
17 fwrite(freq, sizeof(unsigned int), 256, fout);
18
19 for (unsigned int i = 0; i < dS; i++) {
20     unsigned int code = codes[pD[i]];
21     int len = lengths[pD[i]];
22     for (int j = len - 1; j >= 0; j--) {
23         write_bit(&bb, (code >> j) & 1);
24     }
25 }
26 flush_bit_buffer(&bb);
27
28 unsigned int com_size = ftell(fout) - file.Offsetbits;
29 info.SizeImage = com_size;
30 file.Size = file.Offsetbits + com_size;
```

### 3.2.2. Decompression

RLE decompression function for PGM Image file :



```
1 unsigned char value;
2 unsigned char count;
3 long pixels = 0;
4
5 while (pixels < tP) {
6     if (fread(&value, sizeof(unsigned char), 1, input) != 1 ||
7         fread(&count, sizeof(unsigned char), 1, input) != 1) {
8         printf("Error reading RLE pair at pixel %ld\n", pixels);
9         free(dD);
10        fclose(input);
11        return 1;
12    }
13    for (int i = 0; i < count && pixels < tP; i++) {
14        dD[pixels++] = value;
15    }
16 }
17
18 fclose(input);
```



```
1 fprintf(output, "%s\n%d %d\n%d\n", pgm.sign, pgm.width, pgm.height, pgm.maxIntensity);
2
3 if (strcmp(pgm.sign, "P2") == 0) {
4     for (long i = 0; i < tP; i++) {
5         if (fprintf(output, "%d", dD[i]) < 0) {
6             printf("Error writing P2 data at pixel %ld\n", i);
7             fclose(output);
8             free(dD);
9             return 1;
10        }
11        if ((i + 1) % pgm.width == 0) {
12            fprintf(output, "\n");
13        } else {
14            fprintf(output, " ");
15        }
16    }
17 } else if (strcmp(pgm.sign, "P5") == 0) {
18     if (fwrite(dD, 1, tP, output) != tP) {
19         printf("Error writing P5 data\n");
20         fclose(output);
21         free(dD);
22         return 1;
23    }
24 } else {
25     printf("Invalid format in compressed file: %s\n", pgm.sign);
26     fclose(output);
27     free(dD);
28     return 1;
29 }
```

## LZW decompression function for PGM Image file :



```
1 unsigned int bitBuffer = 0;
2 int bits = 0;
3 long pw = 0; //pixelsWritten
4 int next = MAX_SIZE;
5
6 unsigned char byte;
7 if (fread(&byte, 1, 1, input) != 1) {
8     printf("Failed to read initial byte\n");
9     free(dd);
10    free(dict);
11    fclose(input);
12    return 1;
13 }
14 bitBuffer = byte;
15 bits = 8;
16 if (fread(&byte, 1, 1, input) != 1) {
17     printf("Failed to read second byte\n");
18     free(dd);
19     free(dict);
20     fclose(input);
21     return 1;
22 }
23 bitBuffer = (bitBuffer << 8) | byte;
24 bits += 8;
25
26 int code = (bitBuffer >> (bits - 12)) & 0xFFF;
27 bits -= 12;
28 dd[pw++] = (unsigned char)code;
29 int prev = code;
```



```
1 while (pw < tP) {
2     while (bits < 12) {
3         if (fread(&byte, 1, 1, input) != 1) {
4             printf("Unexpected end of file\n");
5             break;
6         }
7         bitBuffer = (bitBuffer << 8) | byte;
8         bits += 8;
9     }
10    code = (bitBuffer >> (bits - 12)) & 0xFFF;
11    bits -= 12;
12
13    unsigned char temp[MAX_SIZE];
14    int tempLen = 0;
15    int currentCode = code;
16
17    if (code >= dictSize) {
18        currentCode = prev;
19        temp[tempLen++] = dict[prev].value;
20    }
21
22    while (currentCode >= 0 && tempLen < MAX_SIZE) {
23        temp[tempLen++] = dict[currentCode].value;
24        currentCode = dict[currentCode].prefix;
25    }
26
27    for (int i = tempLen - 1; i >= 0 && pw < tP; i--) {
28        dd[pw++] = temp[i];
29    }
30
31    if (dictSize < MAX_DICT_SIZE) {
32        dict[dictSize].prefix = prev;
33        dict[dictSize].value = temp[tempLen - 1];
34        dict[dictSize].code = next++;
35        dictSize++;
36    }
37    prev = code;
38 }
39
```

## Huffman decompression function for PGM Image file :



```
1  unsigned int freq[MAX_SIZE] = {0};
2  unsigned char value;
3  int freqCount = 0;
4  while (fread(&value, 1, 1, input) == 1) {
5      if (value == 0) break;
6      unsigned int f;
7      if (fread(&f, sizeof(unsigned int), 1, input) != 1) {
8          printf("Error reading frequency value for byte %d\n", value);
9          free(dD);
10         fclose(input);
11         fclose(output);
12         return 1;
13     }
14     freq[value] = f;
15     freqCount++;
16 }

1  Node* current = root;
2  unsigned char byte;
3  long pW = 0; // pixelsWritten
4  int bits = 0;
5  while (pW < tP) {
6      if (bits == 0) {
7          if (fread(&byte, 1, 1, input) != 1) {
8              printf("Unexpected end of file at pixel %ld\n", pW);
9              break;
10         }
11         bits = 8;
12     }
13     int bit = (byte >> (bits - 1)) & 1;
14     bits--;
15
16     if (bit) {
17         if (!current->right) {
18             printf("Invalid Huffman code at pixel %ld\n", pW);
19             free(dD);
20             freeHuffmanTree(root);
21             fclose(input);
22             fclose(output);
23             return 1;
24         }
25         current = current->right;
26     } else {
27         if (!current->left) {
28             printf("Invalid Huffman code at pixel %ld\n", pW);
29             free(dD);
30             freeHuffmanTree(root);
31             fclose(input);
32             fclose(output);
33             return 1;
34         }
35         current = current->left;
36     }
37
38     if (!current->left && !current->right) {
39         dD[pW++] = current->data;
40         current = root;
41     }
42 }
```

## RLE decompression function for BMP Image file :



```
1  int padding = (4 - ((info.Width * 3) % 4)) % 4;
2      int rowSize = info.Width * 3 + padding;
3      unsigned char* row = calloc(rowSize, 1);
4
5      int Count = 0;
6      RLEEntry entry;
7      while (fread(&entry, sizeof(RLEEntry), 1, in) == 1) {
8          for (int i = 0; i < entry.count; i++) {
9              int pos = (Count % info.Width) * 3;
10                 row[pos] = entry.b;
11                 row[pos + 1] = entry.g;
12                 row[pos + 2] = entry.r;
13
14                 Count++;
15                 if (Count % info.Width == 0) {
16                     fwrite(row, 1, rowSize, out);
17                     memset(row, 0, rowSize);
18                 }
19             }
20         }
21         fseek(out, 0, SEEK_SET);
22         file.Size = sizeof(BmpFile) + sizeof(BmpInfo) + (rowSize * info.Height);
23         file.Offbits = 54;
24         fwrite(&file, sizeof(BmpFile), 1, out);
```

## LZW decompression function for BMP Image file :

```
1  unsigned int out_pos = 0;
2      unsigned int in_pos = 0;
3
4      unsigned short old = input[in_pos++] | (input[in_pos++] << 8);
5      output[out_pos++] = (unsigned char)old;
6
7      while (in_pos < input_size) {
8          unsigned short new = input[in_pos++] | (input[in_pos++] << 8);
9          unsigned char* temp = malloc(original_size);
10             unsigned int temp_pos = 0;
11
12             if (new >= dict_size) {
13                 temp[temp_pos++] = dict[old].append;
14                 unsigned short temp_code = old;
15                 while (temp_code != 0xFFFF) {
16                     temp[temp_pos++] = dict[temp_code].append;
17                     temp_code = dict[temp_code].prefix;
18                 }
19             } else {
20                 unsigned short temp_code = new;
21                 while (temp_code != 0xFFFF) {
22                     temp[temp_pos++] = dict[temp_code].append;
23                     temp_code = dict[temp_code].prefix;
24                 }
25             }
26
27             for (i = temp_pos; i > 0; i--) {
28                 output[out_pos++] = temp[i - 1];
29             }
30
31             if (dict_size < MAX_DICT_SIZE) {
32                 dict[dict_size].prefix = old;
33                 dict[dict_size].append = temp[temp_pos - 1];
34                 dict_size++;
35             }
36             old = new;
37             free(temp);
38         }
```



```
1 unsigned char* pD = lzw_decompress(compressed, info.SizeImage, og_size); // pixel data
2 if (!pD) {
3     free(compressed);
4     fclose(fin);
5     return -1;
6 }
7
8 file.Offbits = sizeof(BmpFile) + sizeof(BmpInfo);
9 file.Size = file.Offbits + og_size + (abs(info.Height) * ((4 - (info.Width * 3) % 4) % 4));
10 info.Compression = 0;
11 info.SizeImage = 0;
12
```

Huffman decompression function for BMP Image file :



```
1
2 unsigned int og_size;
3 fread(&og_size, sizeof(unsigned int), 1, fin);
4
5 unsigned int freq[256];
6 fread(freq, sizeof(unsigned int), 256, fin);
7
8 HuffmanNode* root = build_huffman_tree(freq);
9 unsigned char* pD = malloc(og_size); // pixel data
10 BitBuffer bb;
11 init_bit_buffer(&bb, fin);
12
13 unsigned int pos = 0;
14 HuffmanNode* current = root;
15 while (pos < og_size) {
16     int bit = read_bit(&bb);
17     current = bit ? current->right : current->left;
18     if (!current->left && !current->right) {
19         pD[pos++] = current->symbol;
20         current = root;
21     }
22 }
23
24 file.Offbits = sizeof(BmpFile) + sizeof(BmpInfo);
25 file.Size = file.Offbits + og_size + (abs(info.Height) * ((4 - (info.Width * 3) % 4) % 4));
26 info.Compression = 0;
27 info.SizeImage = 0;
```

## Key Features

- Compresses and decompresses images while preserving their original format.
- Calculates and displays original size, compressed size, and compression ratio.
- Supports user input for file names and operation selection (compress/decompress).
- Uses a consistent intermediate file ("compressed.bin") for storing compressed data.

## File Structure

- **image.h**: Defines common structures (e.g., BMP headers, PGM header) and utility functions.
- **huffpgm.h**: Huffman-specific structures and functions for PGM.
- Separate source files for each algorithm and format (e.g., Huffman BMP, LZW PGM).
- **main.c**: Orchestrates user interaction and algorithm selection.

## 4. Implementation and Testing

The project was developed using C programming languages in Visual Studio Code, with GitHub used for version control. The tool supports Windows and Linux environments, leveraging standard libraries for file handling and image processing.

Below is a detailed breakdown of the implementation and testing process:

### Implementation Details

#### 1. RLE:

- **PGM:** Reads pixel data, applies RLE by counting consecutive identical values, and writes (value, count) pairs to the output file.
- **BMP:** Compresses RGB triplets, handling row padding and updating BMP headers accordingly.
- Key Functions: `compressRLE()`, `decompressRLE()`, `compressBMP()`, `decompressBMP()`.

#### 2. LZW:

- **PGM:** Uses a dictionary initialized with 256 single-byte entries, expanding to 4096 entries with 12-bit codes.
- **BMP:** Employs a 16-bit code scheme, adapting the dictionary to RGB data.
- Key Functions: `compressLZW()`, `decompressLZW()`, `lzw_compress()`, `lzw_decompress()`.

#### 3. Huffman Coding:

- **PGM:** Constructs a min-heap-based Huffman tree, generates codes, and writes bit-packed data.
- **BMP:** Similar approach, with bit buffer management for RGB pixel data.
- Key Functions: `compressHuffman()`, `decompressHuffman()`, `compressBMP3()`, `decompressBMP3()`.

### Testing

#### • Test Cases:

- PGM: Tested with P2 and P5 files of varying sizes and pixel patterns (e.g., solid colours, gradients).
- BMP: Tested with 24-bit uncompressed BMP files, including images with repetitive and diverse pixel data.
- Each algorithm was tested for both compression and decompression, verifying file size reduction and lossless reconstruction.

#### • Metrics:

- Compression ratio calculated as  $(1 - \text{compressed\_size} / \text{original\_size}) * 100$ .

- Output files compared with originals using file comparison tools (e.g., diff for PGM, visual inspection for BMP).
- **Results:**
  - RLE excelled with uniform images (e.g., compression ratio > 50% for solid colours).
  - LZW performed well with moderate repetition (e.g., 20-40% reduction).
  - Huffman Coding was most effective for images with skewed pixel distributions (e.g., 30-50% reduction).

### **Error Handling**

- Checks for file opening failures, invalid formats, and memory allocation errors.
- Provides descriptive error messages to guide users.

## 5. User Interface

The tool features a command-line interface (CLI) designed for simplicity and clarity:

1. **Main Menu:**
  - Prompts: "What is your image type? 1. PGM 2. BMP"
  - Followed by: "Which algorithm? 1. Huffman 2. RLE 3. LZW"
2. **Operation Selection:**
  - "What do you want to do? 1. Compress 2. Decompress"
3. **File Input:**
  - Requests input file name for compression or output file name for decompression.
4. **Output:**
  - Displays progress ("Attempting to compress/decompress..."), success/failure messages, and file size statistics.

The interface is text-based, requiring users to input numbers and file names. It concludes with a thank-you message, enhancing user experience.



```
Welcome to the Image Compression Tool
This tool is used to compress the image using different algorithms.

What is your image type??
1.PGM image.
2.BMP image.
Enter your choice in number: 1

Which algorithm you want to use??
1.Huffman coding.
2.Run Length Encoding.
3.LZW.
Enter your choice in number: 1

What do you want to do??
1.Compress an image.
2.Decompress an image.
Enter your choice in number: 1

Enter the input PGM file name:a.ascii.pgm

Attempting to compress a.ascii.pgm...

Original size: 1458082 bytes
Compressed size: 66029 bytes
Compression ratio: 95.47%
Compression successful: a.ascii.pgm -> compressed.bin

Thank you for using the Image Compression Tool
```

```
Welcome to the Image Compression Tool
This tool is used to compress the image using different algorithms.

What is your image type??
1.PGM image.
2.BMP image.
Enter your choice in number: 1

Which algorithm you want to use??
1.Huffman coding.
2.Run Length Encoding.
3.LZW.
Enter your choice in number: 1

What do you want to do??
1.Compress an image.
2.Decompress an image.
Enter your choice in number: 2

Enter decompressed PGM file name: de.pgm

Attempting to decompress compressed.bin...

Decompression successful: compressed.bin -> de.pgm
Compressed size: 66029 bytes
Decompressed size: 1439173 bytes

Thank you for using the Image Compression Tool
```

## 6.Challenges Faced

### 6.1 Technical Difficulties

- 1. File Format Complexity:** Handling BMP padding and PGM variants (P2 vs. P5) required careful parsing and reconstruction logic.
- 2. Algorithm Adaptation:** Adapting LZW and Huffman to RGB data in BMP files was challenging due to the multi-byte nature of pixels.
- 3. Memory Management:** Ensuring proper allocation and deallocation of dynamic memory (e.g., pixel buffers, Huffman trees) to avoid leaks.
- 4. Bit-Level Operations:** Implementing bit buffers for Huffman Coding demanded precision to avoid data corruption.
- 5. Testing:** Verifying lossless compression across diverse image types was time-consuming, requiring extensive test cases.

### 6.2 Solutions Implemented

- 1.** For BMP, padding was calculated using the formula  $(4 - (\text{width} * 3) \% 4) \% 4$  and skipped during compression with `fseek()` while reading pixel data row-by-row (e.g., in `compressBMP3()`). During decompression, padding was reintroduced by writing zero bytes to align rows (e.g., in `decompressBMP()`).

For PGM, a robust header parsing function `readLine()` was implemented in `image.h` to skip comments and handle both P2 and P5 formats. The code checks the magic number (`strcmp(pgm.magic, "P2")` or `"P5"`) and adjusts reading logic—`fscanf()` for P2 and `fread()` for P5—ensuring accurate data extraction (e.g., in `compressRLE()`).

- 2.** For LZW in BMP (`lzw_compress()`), the dictionary was adapted to handle raw pixel data as a byte stream, treating RGB triplets as sequences without distinguishing color channels explicitly. The dictionary size was capped at 4096 (12-bit codes) to balance memory usage and compression efficiency.

For Huffman in BMP (`compressBMP3()`), a frequency table was built over the entire pixel data byte-by-byte, regardless of RGB boundaries, and a bit buffer (`BitBuffer`) was used to pack variable-length codes efficiently. This approach simplified the adaptation while maintaining good compression ratios.

- 3.** A consistent pattern of allocation and deallocation was enforced. For example, `malloc()` calls for pixel data (e.g., `pixelData` in `compressBMP3()`) were paired with `free()` calls after use. Huffman trees were recursively freed using `free_tree()` or `freeHuffmanTree()` (e.g., in `compressHuffman()`).

Error checks were added after each `malloc()` (e.g., if `(!imageData)` in `compressRLE()`), exiting gracefully with error messages to prevent crashes from failed allocations.

- 4.** A `BitBuffer` structure was defined with functions like `write_bit()`, `read_bit()`, and `flush_bit_buffer()` (e.g., in the Huffman BMP code). These managed an 8-bit buffer, writing or reading one bit at a time and flushing incomplete bytes with padding, ensuring accurate bit-level I/O.

During decompression (`decompressBMP3()`), the Huffman tree was traversed bit-by-bit using `read_bit()`, resetting to the root node upon reaching a leaf, which guaranteed correct symbol reconstruction.

5. Comprehensive error handling was integrated, such as checking file read/write success (e.g., if (`fread(...) != 1`) in `compressBMP2()`) and validating format compatibility (e.g., `biBitCount != 24` in `compressBMP3()`).

File size comparisons were automated in the code (e.g., using `ftell()` in `compressHuffman()` and `decompressHuffman()`), displaying original and compressed sizes along with compression ratios. This allowed quick verification of compression effectiveness.

Manual testing with sample images (e.g., solid colors for RLE, gradients for Huffman) ensured each algorithm's performance was validated, supplemented by visual inspection of decompressed BMPs and byte-by-byte comparison for PGMs.

## 7. Conclusion

### 7.1 Lessons Learned

The "Image Compression Tool" successfully achieves its goal of providing a versatile, lossless compression utility for PGM and BMP images. By implementing RLE, LZW, and Huffman Coding, it offers users flexibility in choosing an algorithm suited to their image data. The project demonstrates proficiency in C programming, file I/O, and compression techniques, while maintaining a user-friendly interface.

This project provided hands-on experience with compression algorithms, debugging large datasets, and optimizing code performance. Understanding the trade-offs between compression ratio and processing time was a key takeaway.

### 7.2 Future Enhancements

Future enhancements could include support for additional formats (e.g., JPEG, PNG), graphical user interface (GUI) integration, or hybrid compression methods for improved ratios and parallel processing for faster compression of large images. Overall, the tool serves as an effective educational and practical demonstration of image compression principles.

## 8.Reference

- Run Length Encoding , Wikipedia , [[https://en.wikipedia.org/wiki/Run-length\\_encoding](https://en.wikipedia.org/wiki/Run-length_encoding)].
- LZW Compression technique (21 May 2024), GeeksforGeeks , [<https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/>].
- Huffman Coding Algorithm, Programiz , [<https://www.programiz.com/dsa/huffman-coding>].
- Image Compression using Huffman Coding , GeeksforGeeks, [<https://www.geeksforgeeks.org/image-compression-using-huffman-coding/>].
- BMP file format, Wikipedia, [[https://en.wikipedia.org/wiki/BMP\\_file\\_format](https://en.wikipedia.org/wiki/BMP_file_format)].
- PGM Image file format - WPI , Worcester Polytechnic Institute, [<https://users.wpi.edu/~cfurlong/me-593n/pgmimage.html#:~:text=PGM%20is%20a%20standard%20bitmap,8%2Db it%20data%20per%20pixel.>].
- Rafael C Gonzalez & Richard E. Woods, (2002). *Digital Image Processing*. Prentice Hall.
- PGM Image download, Florida State University, [<https://people.sc.fsu.edu/~jburkardt/data/pgma/pgma.html>].
- BMP Image download, File Samples, [<https://filesamples.com/formats/bmp>], samples-files.com, [<https://samples-files.com/sample-bmp-file/>]