

---

---

# Image Compression using Huffman tree and FILE Handling


Submitted by: Ashish Virdi (2020csb1077) ,  
Avnish Kumar(2020csb1078)

Under the Guidance of: Dr. Anil Shukla

---

---

# Contents

- 
- 1 **Description**
  - 2 **Defining Inputs and Outputs of the Program**
  - 3 **File Encoding Methods**
  - 4 **File Decoding Method**
  - 5 **Conclusion**

# Description

- The project demonstrates the process of image compression using Huffman Tree and File handling.
- The Huffman tree is a binary tree associated with minimum external path weight i.e. minimum sum of weighted path lengths for the given set of leaves.
- Language used: c
- Libraries used: `stdio.h`, `stdlib.h`, `string.h`, `dirent.h`

# Defining Inputs and Outputs of our program.

## INPUTS REQUIRED

### 1. Input Directory path

- The path of folder containing the BMP Images.
- *FORMAT FOR PATH* - `"./InputFolder"`. (The folder should be in same directory as that of our Code).

### 1. Output Directory path

- The path of folder where we wish to see our Output i.e. encoded and decoded BMP images.
- *FORMAT FOR PATH* - `"./OutputFolder"`. (The folder should be in same directory as that of our Code).

# Defining Inputs and Outputs of our program.

## OUTPUTS GENERATED

1. **Encoded files** are generated into the Output Directory using the input files one by one.
2. **Compression.txt** file is generated in the Output Directory containing the Compression ratios of all the encoded files.
3. **Decoded files** are generated in the Output Directory one by one using the Encoded files in the Output Directory.

# File Encoding Method

(For FILE Compression using Huffman tree)

Most used/recommended Encoding method is :

1. Write the huffman tree formed into the Encoded file.

Since we only need Frequency array of characters appearing in the file to create a huffman tree. We will write the frequency array into the encoded file instead of the tree itself as huffman tree can be formed using this.

1. Write the offbits into the file.

Offbits denote the amount of bits we have to ignore for the last character as a character is of 8 bits and if our total length of encodings is not a multiple of 8 then some bits are required to be added at the end. So that a character can be written into the file.

1. Write the encodings formed into the file character by character.

When we run out of character we will write the last encoding with padding/offbits into the encoded file.

# ENCODING IS DONE FIRST

Encoding functions has 3 parts :

1. `freq_counter(fpr, ascii);`
2. `Build_Huffmantree(ascii, leafnodes);`
3. `encoding_buffer(fpr, fpw, leafnodes, &offbits);`

```
freq_counter(fpr, ascii);
```

Uses a while loop and `fgetc(fpr)` and counts the characters till the end of file, keeps updating the frequency of characters encountered

```
ascii[character]++;
```

We write frequency array in to encoded file.

```
// writing the frequency of ascii charaters into the encoded file.
for(int i = 0; i < 256; i++)
{
    fwrite(ascii + i, sizeof(unsigned int), 1, fpw);
}
```

We write temporary unsigned int (placeholder for offbits) and store the pointer location because we have to return to this position to write the actual off-bits of the last characters when all the encodings are written into the encoded file.

```
fwrite(&temp, sizeof(unsigned int), 1, fpw);
```

```
Build_Huffmantree(ascii, leafnodes);
```

THIS FUNCTIONS IS EXPLAINED IN BELOW SLIDES. Root node for our huffman tree is obtained from this functions.



# BUILDING HUFFMAN TREE FROM FREQUENCIES

- WE PASS THE FREQUENCY OF ASCII CHARACTERS AND 256 LEAF NODES WHERE EACH INDEX DENOTES FREQUENCY OF ASCII EQUIVALENT OF THAT VALUE.-> `Build_Huffmantree(ascii,leafnodes);`
- SINCE CHARACTERS ARE 8 BITS AND HAVE UNSIGNED INT EQUIVALENT VALUE BETWEEN 0 TO 255 . ASCII ARRAY WILL CONTAIN FREQUENCIES FROM 0 TO 255.
- HUFFMAN TREE IS BUILT USING TWO AUXILIARY ARRAYS i.e. leaf nodes and helper array, ----> Helper array acts as a parent array .
- \*LEAF NODES ARE SORTED IN INCREASING ORDER BASED ON THEIR FREQUENCIES(selection sort is used).

# Demonstration of BUILDING HUFFMAN TREE.

Let the leaf nodes and parent array be of size 4.

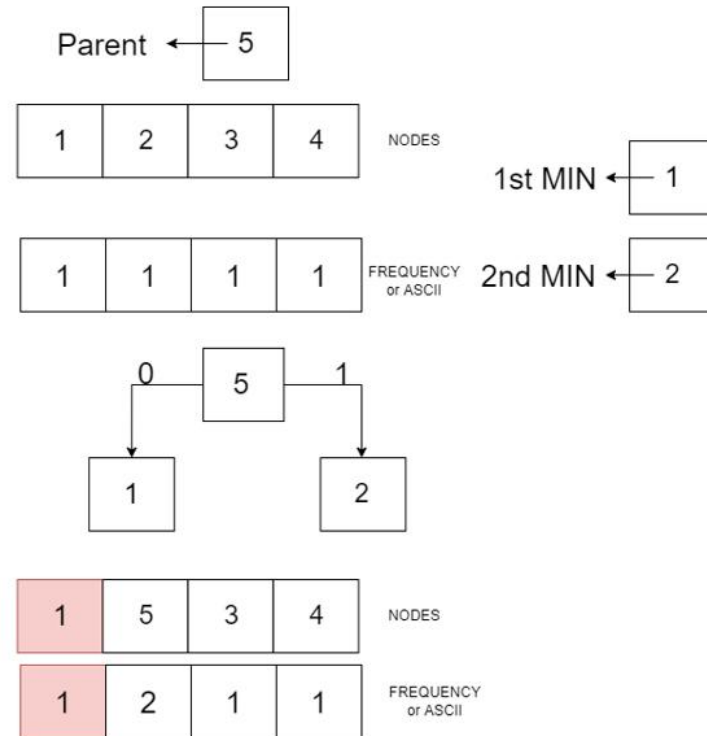
NODES is a pointer array, initially storing the location of leaf nodes.

i.e .  $\text{NODES}[i] = \text{leaf nodes}[i]$

1	2	3	4	NODES
1	1	1	1	FREQUENCY or ASCII
1	2	3	4	Leaf Nodes (character)
5	6	7	8	PARENTS

A TOTAL OF N-1 COMPARISONS ARE REQUIRED FOR GETTING THE ROOT NODE OF OUR HUFFMAN TREE.

### FIRST ITERATION



Nodes in the red will not be visited/used again.

SORTING NODE ARRAY ACC. TO FREQ AFTER  
PARENT NODE IS ADDED .

1	5	3	4
---	---	---	---

NODES

1	2	1	1
---	---	---	---

FREQUENCY  
or ASCII

1	3	5	4
---	---	---	---

NODES

1	1	2	1
---	---	---	---

FREQUENCY  
or ASCII

1	3	4	5
---	---	---	---

NODES

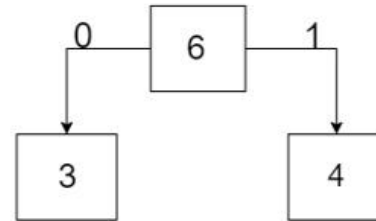
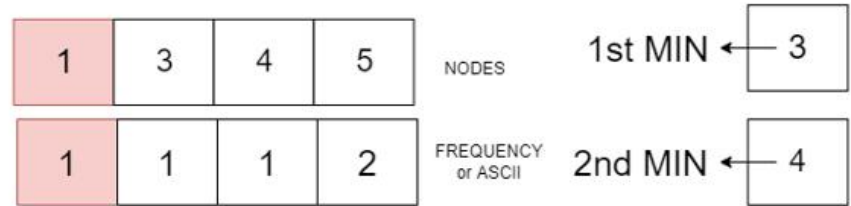
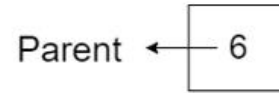
1	1	1	2
---	---	---	---

FREQUENCY  
or ASCII

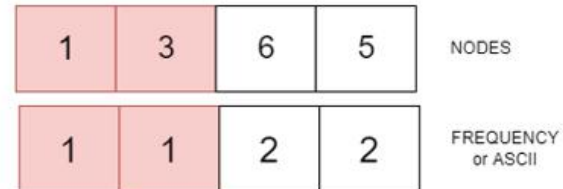
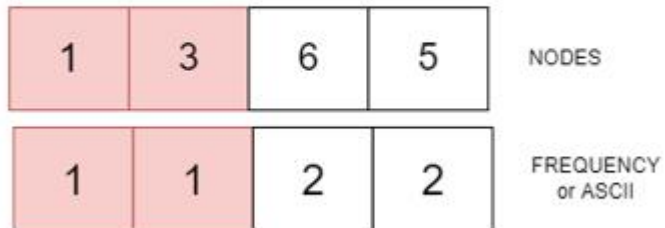
NODES ARRAY IS NOW SORTED  
GO TO NEXT ITERATION

## 2nd iteration:

### SECOND ITERATION

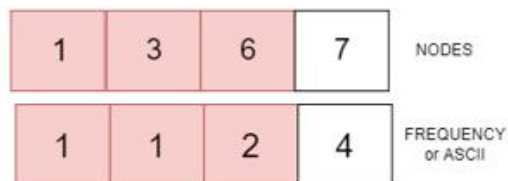
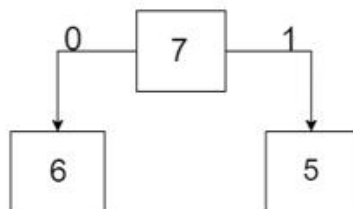
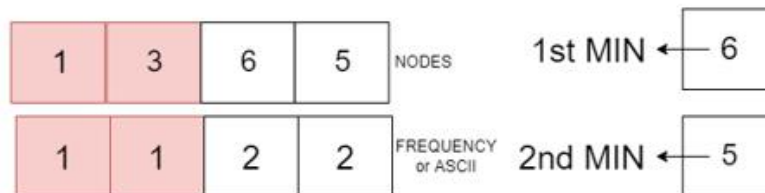
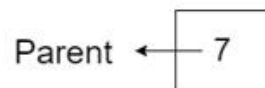


SORTING NODE ARRAY ACC. TO FREQ AFTER PARENT NODE IS ADDED .



## Third Iteration:

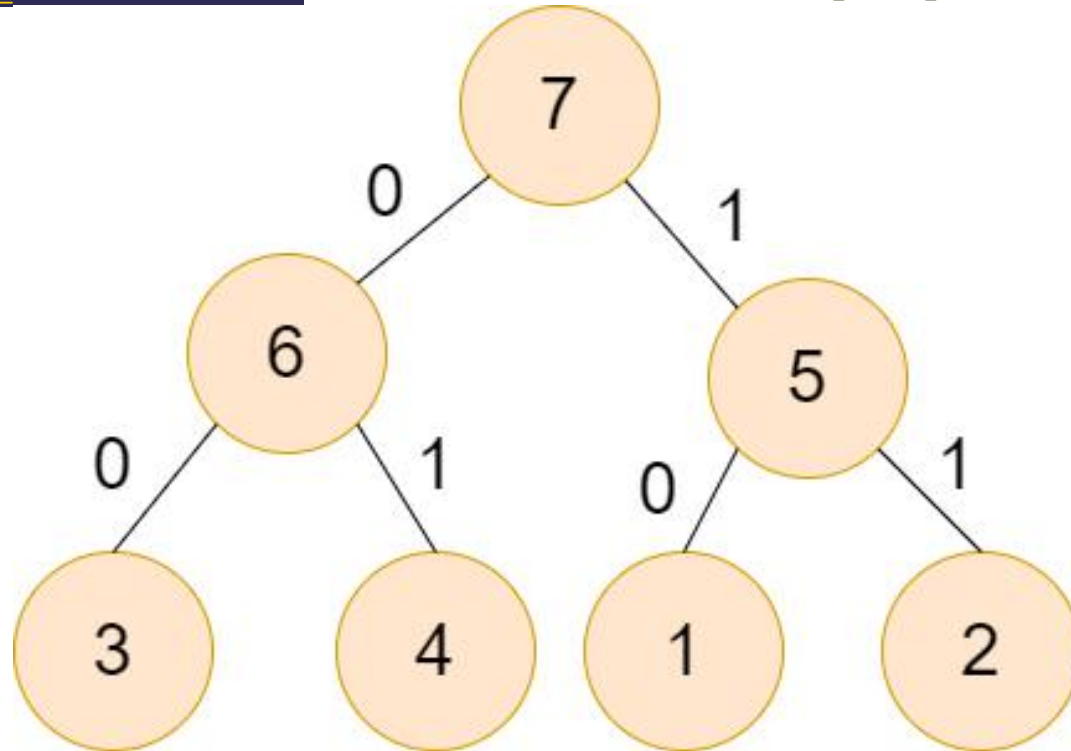
### THIRD ITERATION



AT LAST OF ITERATION WHEN WE REACH THE END OF NODES ARRAY  
WE HAVE OUR ROOT NODE WHICH HAS THE HIGHEST FREQUENCY

Huffman tree obtained for the above example will look like this .

`Build_Huffmantree()` function returns `nodes[last]` as the root node.



```
encoding_buffer(fpr, fpw, leafnodes , &offbits);
```

fpr is at the beginning of bmp file, fpw is after 1024+4 pos.

- For encoding we read a character. We take a temp pointer pointing to node at character value, example -> leafnode[0] denotes ascii[0] node in the tree. leafnode[36] denotes '\$' node in the tree. Once we have the temporary pointer to our character. We can traverse from bottom to top towards its parent and store the path taken.

- ```
form_encodings(temp, encodings, &height);
```

Encodings array stores the path of current character from the root node.



Initially height is 0. We will use the huffman tree example to explain this.

For example we call form\_encoding on 3.

3 = parent(3)->left i.e. 3 = 6->left, encoding[0] = 0

6 = parent(6)->left i.e. 6 = 7->left encoding[1] = 0

Encoding for 3 = [0,0] Encoding for 4 = [1,0]

Encoding for 1 = [0,1] Encoding for 2 = [1,1]

Height stores the current height of the encodings.

Height can never exceed 256 even if every node has frequency 1, height will still be 255.

```
while (head->parent!=NULL)
{
    if (head == head->parent->left) {
        encodings[*height] = 0;
    }
    else if (head == head->parent->right) {
        encodings[*height] = 1;
    }
    head = head->parent;
    (*height)++;
}
```

```
write_encodings(fpw,encodings,height,&buffer,&buffersize);
```

Now that we have the height and encoding of a character in bottom to top manner. We will write character in place of encoding when length of bits taken from encodings becomes 8.

We use an example to demonstrate this

Let take two characters a,b

encodings(a)=[1 , 0 , 1, 1 ] ,      encodings(b)=[1 , 1 , 0, 1, 0, 1 ]

Writing the encodings, we take an unsigned integer or char `buffer = 0`

----- 00000000 - buffer      encoding(a) 1011

----- 00000000 | 00000001      // WE NEED TO LEFT SHIFT ENCODING[HEIGHT-1]

----- BUFFER - 00000001

----- BUFFERSIZE - 1

----- 00000001 | 00000000

----- BUFFER - 00000001

----- BUFFERSIZE - 2

----- 00000001 | 00000100

----- BUFFER - 00000101

----- BUFFERSIZE - 3

----- 00000101 | 00001000

----- BUFFER - 00001101

----- BUFFERSIZE - 4

Buffer size and Buffer remains the same and fgetc for other character b is now called.

```
----- 00001101 - buffer      encoding(a) 110101
----- 00001101 | 00010000    // WE NEED TO LEFT SHIFT ENCODING[HEIGHT-1]
by BUFFERSIZE - 4

----- BUFFER - 00011101
----- BUFFERSIZE - 5
----- 00011101 | 00100000
----- BUFFER - 00111101
----- BUFFERSIZE - 6
----- 00011101 | 00000000
----- BUFFER - 00111101
----- BUFFERSIZE - 7
```

----- 10011101 | 10000000

----- BUFFER - 10111101

----- BUFFERSIZE - 8

Since buffer size - 8 we write this character into the encoded file.

putc(10111101) - putc(189)

BUFFERSIZE - 0 , BUFFER - 00000000

still height is not zero therefore we read BUFFER FROM ENCODING

----- 00000000 | 00000000      [ 0, 1] are left

----- BUFFER - 00000000

----- BUFFERSIZE - 1

----- 00000000 | 00000010

----- BUFFER - 00000010

----- BUFFERSIZE - 2

Now we dont have any character to read, while loop is ended

The last buffer is also written into the file, `fputc(00000010) - fputc(2)`

```
fputc(buffer, fpw);  
(*offbits) = 8 - buffersize;
```

Offbits will be  $8 - 2 = 6$ .

We got to the offbitset position in the encoded file to write the actual offbits.

Encoding of file is completed.

# DECODING IS DONE ON THE ENCODED FILE

Encoded file in the output directory is taken as input for decoding and target file is generated in the same directory as that of the encoded file.

Decoding functions has 2 parts :

1. `Build_Huffmantree(ascii, leafnodes);`
2. `encoding_buffer(fpr, fpw, leafnodes, &offbits);`

First we read the encoded file, the first thing we read is the frequency ascii array from the file using ,

```
for(int i = 0; i < 256; i++){  
    fread(&freq, sizeof(unsigned int), 1, fpr);  
    ascii[i] = freq;  
}
```

And we also read the offbits to determine how much characters will be read in the last character.

```
fread(&offbits, sizeof(unsigned int), 1, fpr);
```

1. `Build_Huffmantree(ascii, leafnodes);`

Build\_Huffman tree is exactly same as before. Returns the root node of the huffman tree formed.

1. `decoding_buffer(fpr, fpw, head, offbits);`

We now call this function. We calculate the length of file left to read i.e.

```
unsigned long int begin = ftell(fpr);, unsigned long int end = ftell(fpr);
```

now length - (end - begin).

we take an unsigned integer or char `buffer = 0,`

----- we read buffer from the encoded file.

To demonstrate we will use the same example of a and b



Buffer gets value 189 - 10111101

We read the MSB into unsigned char using & operator `bit = buffer & 1;`

We use this bit to traverse from the root node of huffman tree to retrieve our character from the huffman tree.

```
if(bit == 0)    // 0 for left
{
    temp = temp->left;
}
else           // 1 for right
{
    temp = temp->right;
}
```

We end our loop when we encounter a root node.

A root node encounter means that we have reached the desired ascii leafnode.

We put this leafnode's data into the decoded file.

```
fputc(temp->data, fpw);
```

For offbit our condition is

```
if(length == 1) { // special case where we reach the last offbit
char and we adjust the padding for it by using buffersize as 8 - offbits.
    buffersize = 8 - offbits;
    if(buffersize == 0){ return; }
}
```

Setting the buffersize as 8 - offbits helps us in ignoring these bits and end decoding the file.

Demonstration continued ->

Buffer gets value 189 - 10111101

Length - 2

Buffersize - 8

We read bit - buffer & 1 and we right shift our buffer every time so as to read the next significant bit.

We read 1 then traverse on the tree ,

Buffer - 01011110 , bit - 0

Buffer - 00101111 , bit - 1

Buffer - 00010111 , bit - 1 we reach leaf node and thus fputc(temp->data)

Buffersize - 4 is not 0 therefore we continue to read it.

Buffer - 00001011 , bit - 1

Buffer - 00000101 , bit - 1

Buffer - 00000010 , bit - 0

Buffer - 00000001 , bit - 1

Buffer - 00000000 , Buffer size - 0

We read next character, we notice that length = 1 that means we are at the last character which will contain offbits. We modify our buffersize to 8 - offbits.

Next character Buffer gets value 2 - 00000010, offbits - 6, buffersize - 2

Buffer - 00000010 , bit - 0

Buffer - 00000001 , bit - 1

we reach leaf node and thus `fputc(temp->data)`

We end our while loop because both length - 0 and buffersize - 0, i.e. we have nothing to write.

We have successfully encoded our file.

Compression ratio is given by the formula

$(\text{initialbitslength}/\text{encodedbitslength}) * 100$

Initialbitslength and encodedbitslength are given by `fseek(fp,0,SEEK_END)` in both the cases.

```

/*****
//      BMP HEADER STRUCTURE
*****/

struct BITMAP_header{
    char name[2];
    unsigned int size;
    int garbage;
    unsigned int image_offset;
};

```

```

struct DIB_header{
    unsigned int header_size;
    unsigned int width;
    unsigned int height;
    unsigned short int colorplanes;
    unsigned short int bitsperpixel;
    unsigned int compression;
    unsigned int image_size;
    unsigned int temp[4];
};

```

BITMAP\_header - 14 bits

DIB\_header - 40 bits.

Total length of BMP  
header is 54 bits.

# Conclusions with scope of improvement

- This compression technique can be improved by writing only those nodes which have ascii frequency  $> 0$  into the encoded file we will save.
- $(256 - \text{ascii\_frequency} > 0) * 4$ . The technique can be further improved using a proper priority queue/ Min heap implementation as that will make the time complexity to be  $O(n \log n)$ .
- Desired outcome is achieved.