RV COLLEGE OF ENGINEERING®
BENGALURU – 560059
(Autonomous Institution Affiliated to VTU, Belagavi)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Huffman Coding

Experiential Learning REPORT
Discrete Mathematical Structures (18CS36)
III SEMESTER

2021-2022

Submitted by

Naman N Karanth	1RV20CS091
Pydi Venkat	1RV20CS128
Meeth J Davda	1RV20CS087
Pari Raheja	1RV20CS104

Under the Guidance of

Anitha Sandeep,
Assistant Professor
Department of CSE, RVCE,
Bengaluru - 560059

RV COLLEGE OF ENGINEERING®, BENGALURU - 560059
(Autonomous Institution Affiliated to VTU, Belagavi)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Certified that the **Experiential Learning** work titled "**Huffman Coding**" has been carried out by **Naman N Karanth(1RV20CS091),Pydi Venkat(1RV20CS128),Meeth J Davda(1RV20CS087),Pari Raheja(1RV20CS104),** bonafide students of RV College of Engineering, Bengaluru, have submitted in partial fulfillment for the **Assessment of Course: Discrete Mathematical Structures (18CS36) – Experiential Learning** during the year 2021-2022. It is certified that all corrections/suggestions indicated for the internal assessment have been incorporated in the report.

Faculty Incharge
Department of CSE,
RVCE., Bengaluru –59

Head of Department
Department of CSE,
RVCE, Bengaluru–59

ACKNOWLEDGEMENT

We Would like to express our deepest gratitude to Our Professor **Anitha Sandeep , Assistant Professor Dept. of Computer Science and Engineering** for contributing her valuable time and efforts in helping us out with this project. Her suggestions and feedback have helped us a lot in improving the quality of the project.

We would also like to thank the HOD of the Computer Science Department **Dr.Ramakanth Kumar P** and Principal of RV College of Engineering **Dr. K N Subramanya** for giving us this opportunity to implement our project.

We would also like to thank our friends and family for their constant encouragement and support throughout the project.

Lastly, We like to thank all our supporters who have motivated us to fulfill their project before the timeline.

Abstract

Huge data system applications require storage of large volumes of data set, and the number of such applications is constantly increasing as the use of computers extends to new disciplines. At the same time, the proliferation of communication networks is resulting in massive transfer of data over communication links. Compressing data to be stored or transmitted reduces storage and communication costs. When the amount of data to be transmitted is reduced, the effect is that of increasing the capacity of the communication channel.

Here an efficient method for decoding the compressed data is proposed.A number of data compression techniques have been introduced to reduce the text/data storage and transmission costs. This paper describes the development of a data compression system that employs an adaptive Huffman method for generating variable-length codes. Construction of the tree is discussed for gathering latest information about the entered message. The encoder process of the system encodes frequently occurring characters with shorter bit codes and infrequently appearing characters with longer bit codes.

The decoder process expands the encoded text back to the original text and works very much like the encoder process. Experimental results are tabulated which demonstrate that the developed system is very effective for compressing database files (provides compression ratio up to 60%) in a real-time environment.

Table of Contents

List of Figures

List of Tables

GLOSSARY

CI	:	Compressed Image
DMS	:	Discrete Mathematical Structures
HC	:	Huffman Coding
HT	:	Huffman Tree
MH	:	Min-Heap
PQ	:	Priority Queue
SRS	:	Software Requirement Specification

Chapter 1

This chapter provides the introductory and essential concepts that are required henceforth in this paper. A brief history of Huffman coding is also mentioned with a reference to its developer. Further, we also describe how Huffman coding enters the domain of discrete mathematics.

Introduction

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code proceeds by means of Huffman coding, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes"

The objectives of our project are

1. To enunciate the concept of Huffman Coding and relate it to Discrete Mathematics
2. To implement Huffman trees using heaps and queues, and further compare their time complexities.
3. To apply and implement Huffman Coding for Image Compression

The utilisation of efficient huffman coding algorithms in areas of cryptography and better lossless image compression techniques for digital communication and storage.

Huffman coding is a method for the construction of minimum-redundancy codes.The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol.The algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol.Huffman Coding is a technique of compressing data to reduce its size without losing any of the details.Huffman code can be recognized with the help of Huffman coding trees or simply Huffman trees. Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix code ie. a code associated with a character should not be present in the prefix of any other code. The tree created helps in maintaining the property.

1.1 Discrete Mathematical Concepts

- Huffman Coding provides a scheme to encode data in such a way that it is not immediately perceivable to a potential threat.
- Huffman coding is an entropy encoding algorithm used for lossless data compression
- It can be extensively used to encrypt data along with its compression.
- Data can be either compressed and encrypted using existing compression algorithms or huffman compression can be applied with new symmetric cryptographic algorithms.
- Hence Huffman coding is so closely related to cryptography, which is the study of creating security structures and passwords.
- Also, since Huffman encoding happens entirely in binary digits and involves the storage and communication of digital information, it uses concepts of information theory and theoretical computer science , hence entering the domain of discrete mathematics.

1.2 Proposed System

1.2.1 Objectives

- To enunciate the concept of Huffman Coding and relate it to Discrete Mathematics
- To implement Huffman trees using heaps and queues, and further compare their time complexities.
- To apply and implement Huffman Coding for Image Compression

1.2.2 Methodology

- The concept of Huffman Coding involves the generation of code words in binary for a particular data set. Images and text files can be efficiently compressed without the loss of data using Huffman Trees. We attempt to enunciate its relation to discrete mathematical structures. It comes under the field of theoretical computer science and information theory in the domain of discrete structures. To explain the basic concept and development of the huffman trees we take an example and display the tree at each stage.

- The next step involves implementing the huffman tree concept using various data structures to compare their time complexities and extracting the most efficient one. First, huffman trees are realised with the help of min heaps. Next, we implement the same tree using queues. This approach is specific for sorted inputs. A clock function under the

<timer.h> header is utilised for the purpose of recording the execution time of the two algorithms. By displaying the time taken by the two algorithms we strive to prove the more efficient one.

- Finally, the concept of huffman coding is applied in a real life example, which is lossless image compression. An image file of 15x15 dimension is obtained in the bmp format. This is passed to the compression algorithm which generates a Huffman tree from the pixel values of the image. The Huffman codes are displayed and the image is encoded into a separate text file. Next, the proof of the compression is given considering the encoded file size and the size of the Huffman codes/tree. Further, the encoded file is decoded with the decompression algorithm and matched with the original image.

1.2.3 Scope

- Huffman Coding technique is used for image compression
- Huffman coding is used to compress data without any loss.It is a lossless data compression technique.
- Huffman encoding is widely used in compression formats like GZIP, PKZIP (winzip) and BZIP2.
- Multimedia codecs like JPEG, PNG and MP3 uses Huffman encoding (to be more precise the prefix codes)
- Huffman encoding includes conjunction with cryptography.

Chapter 2

Requirement Specification

In this chapter we have discussed the various software and hardware requirements for our implementation.

For the implementation of this project the minimum requirements are as follows
1. Windows/Mac OS/Linux
2. Text Editor
3. C compiler
4. Image/data for the processing

2.1 Hardware Requirements

The hardware requirements for the project are as follows
Laptop/PC with below configurations
1. Processor- intel i5 and above / M1
2. RAM- Min 4GB.
3. Memory- 2GB

2.2 Software Requirements

1. Windows/Mac OS/Linux/MINIX/XV6
2. gcc compiler
3. Microsoft Visual Studio Code/CodeBlocks/any Text Editor with terminal.
4. BMP Image Converter
5. Standard Libraries of C

Chapter 3

System Design and Implementation

This chapter focuses on the detailed method of implementing the mentioned objectives. It involves showcasing the underlying concept of Huffman trees and the two algorithms used to implement this concept. The time taken by the two implementations are compared to find the more time efficient one. We also apply the discussed concept in image compression and propose the algorithm required for it along with the proof of its compression.

3.1 Modular Description

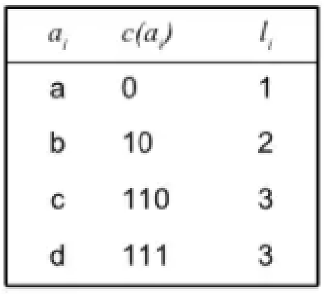
3.1.1 Concept Enumeration

The problem statement of Huffman coding is

Given
A set of symbols and their weights(probabilities or frequencies). This is not the data in the raw form but in the organised form. Input is generally in the form of the characters of a string or pixel intensity values of an image. These are passed to a function which maps each unique data element in the data set to its corresponding frequency of occurrence.

Find
From the given set of characters and their corresponding frequencies we are expected to find a prefix binary codeword of minimum expected length for each unique character or data element. Equivalently, a tree with minimum weighted path length from root.

Example
Given
A = { a , b , c , d }
P = { ½, ¼ ,⅛, ⅛} (Obtained from their frequencies)



a_i	$c(a_i)$	l_i
a	0	1
b	10	2
c	110	3
d	111	3

Table 3.1 Huffman Table

Huffman Table
a_i- Character
$c(a_i)$-Code
l_i- Number of bits

Compression

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, n. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain a weight, links to two child nodes and an optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and n-1internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths. The process begins with the leaf nodes containing the probabilities of the symbol they represent. Then, the process takes the two nodes with the smallest probability, and creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weight of the children. We then apply the process again, on the new internal node and on the remaining nodes (i.e., we exclude the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.

Decompression

Generally speaking, the process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be somehow reconstructed. In the simplest case, where character frequencies are fairly predictable, the tree can be preconstructed (and even statistically adjusted on each compression cycle) and thus reused every time, at the expense of at least some measure of compression efficiency. Otherwise,the information to reconstruct the tree must be sent a priori.

A simple conceptual example is portrayed below to explain the essence of Huffman coding at its core.

Suppose the string below is to be sent over a network.

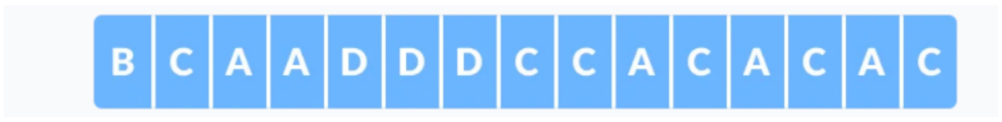


Fig 3.1

- Each character in Fig 3.1 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.
- Using the Huffman Coding technique, we can compress the string in Fig 3.1 to a smaller size.
- Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.
- Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

The following steps are to be followed to encode the above shown example.

1. Calculate the frequency of each character in the string as shown in Fig 3.2.

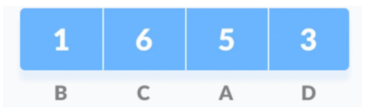


Fig 3.2

2. Sort the characters in increasing order of the frequency. These are stored in a priority queue Q which is displayed in Fig 3.3.

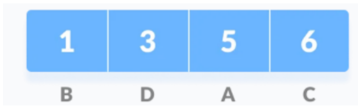


Fig 3.3

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 as indicated in Fig 3.4.(* denote the internal nodes in the figure above).

.

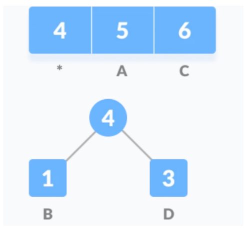
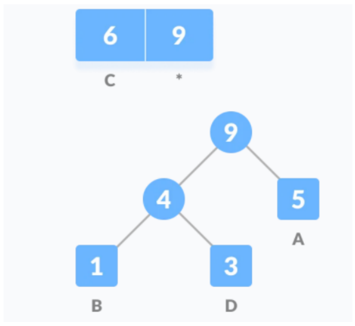


Fig 3.4

5. Remove these two minimum frequencies from the Priority Queue and add the sum into the list of frequencies.

6. Insert node z into the tree.

7. Repeat steps 3 to 5 for all the characters which are shown in Fig 3.5 and Fig 3.6.



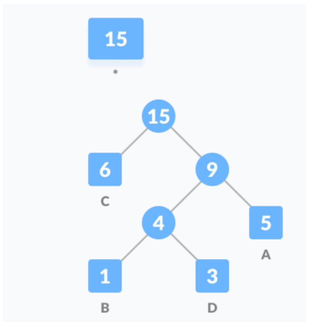


Fig 3.5 **Fig 3.6**

8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge. The final tree is displayed in Fig 3.7.

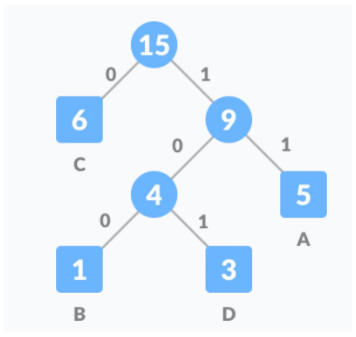
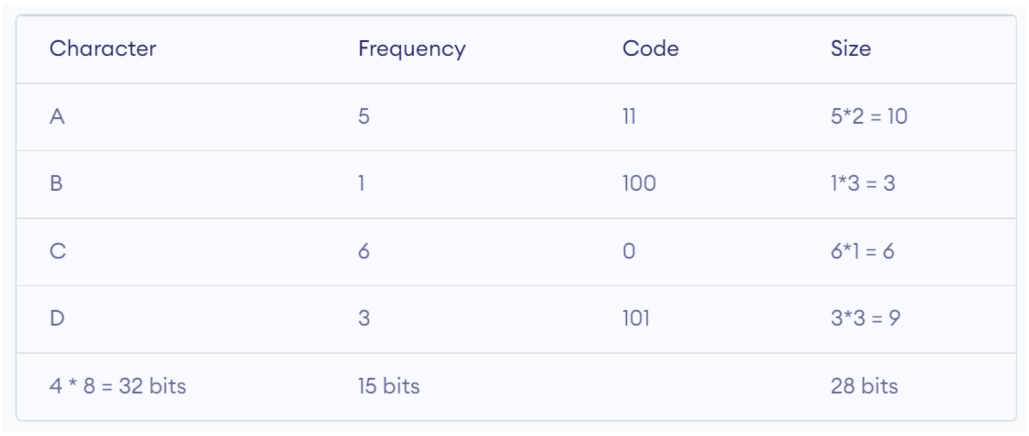


Fig 3.7

Using the huffman tree, the following huffman table is generated



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

Table 3.2

For sending the code across the network we have to send the encoded bits as well as the table shown in fig 8. Without encoding the string size was found to be 120 bits. But after encoding the new string size can be obtained from fig 8 as 32 + 15 + 28 = 75 bits which effectively is compression.

3.2 Comparison of Algorithms

3.2.1 Algorithm 1-Using Min Heaps

In a Min-Heap the key present at the root node must be less than or equal to among the keys present at all of its children. In a Min-Heap the minimum key element present at the root. A Min-Heap is a priority queue that uses the ascending priority i.e. in the construction of a Min-Heap, the smallest element has priority.

- Input the string that has to be encoded. The string is condensed into a unique character array and a frequency array
- A min heap is created with this data by first storing elements in an array and minheapifying it
- The extractMin function is repeatedly used to extract the first two minimum frequencies(along with the associated characters) from the min heap and hence these frequencies are removed from the heap
- A new node of the huffman tree is generated whose left and right children are respectively

the two minimum frequencies.

- The data of the node is fed to the min heap which is minHeapified once more.
- The above steps are repeated until there is only one node in the heap which is the root node.

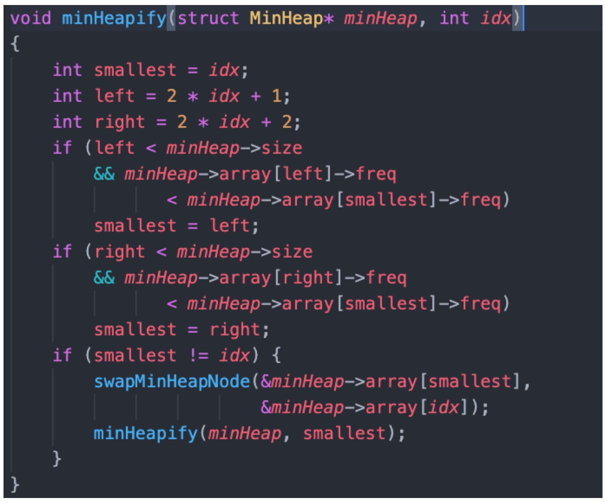
Major Functions and Structures Used:

- The minHeapNode structure is the basic element of the implemented minHeap. It also forms a node of the final huffman tree. It consists of a data field to store the character value and a frequency field to store the corresponding frequency. Most importantly, it holds the addresses of its left and right children as it is a node of a binary tree.

- The minHeap structure represents a min heap along with details of its properties such as size, capacity and the min heap array which represents the tree.

- createMinHeap() : This function dynamically allocates memory for the min heap and initializes its various attributes such as size, capacity(maximum size) and the min heap array which holds the tree.

- minHeapify() : This function is helpful in maintaining the property of the min heap by traversing through the heap and comparing all parent nodes with their children. If any violation of the min heap rule(parent is always lesser than child) is found the parent and child are interchanged and the child is checked with its child if it is a parent. Hence the min heap regains its property and the minimum node ends up in the beginning.



```c
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    if (left < minHeap->size
        && minHeap->array[left]->freq
                < minHeap->array[smallest]->freq)
        smallest = left;
    if (right < minHeap->size
        && minHeap->array[right]->freq
                < minHeap->array[smallest]->freq)
        smallest = right;
    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest],
                        &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}
```

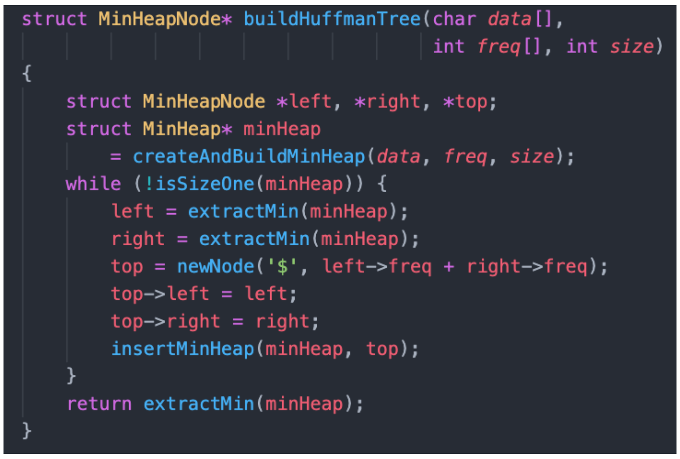
Img 3.2.1 minHeapify function

- extractMin() : This function simply extracts the value of the root of the min heap as the minimum value is predicted at the root from the structure of the min heap.

- buildMinHeap() : This function receives disorganised heap as its input along with the index of addition which brought up this chaos. The function calls the minHeapify function from this index which eliminates unnecessary sorting of the entire heap thereby improving performance and reducing computational cost.

- createAndBuildMinHeap() : It takes input as the unique character array and the corresponding frequency array. It calls the createMinHeap function to create min heap structure and allocate space. According to the inputs the nodes are inserted into the array attribute of the created min heap and minHeapified.

- buildHuffmanTree() : It is one of the main functions used to build a huffman tree from the existing min heap. A min heap is created by calling the createAndBuildMinHeap function and passing the required dataset. The two minimum frequencies of the min heap are extracted and combined according to the discussed Huffman Algorithm. This combination is added as a separate node to the min heap and the entire heap is min heapified to maintain its properties.

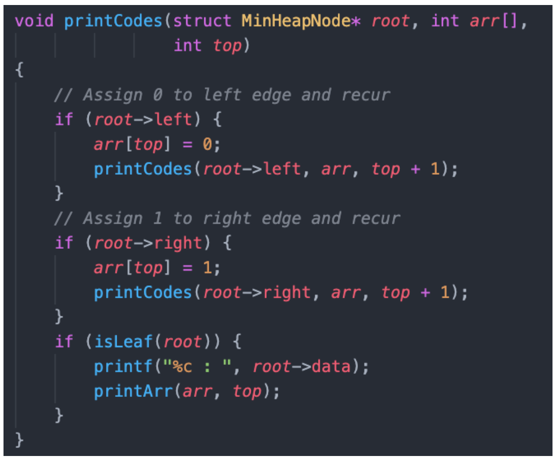


```
struct MinHeapNode* buildHuffmanTree(char data[],
                                     int freq[], int size)
{
    struct MinHeapNode *left, *right, *top;
    struct MinHeap* minHeap
        = createAndBuildMinHeap(data, freq, size);
    while (!isSizeOne(minHeap)) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
    }
    return extractMin(minHeap);
}
```

Img 3.2.2 buildHuffmanTree() function

- printBin() : This function takes in the huffman tree ,the character to be checked and the character array storing the encoded string in huffman codes. It does so by reading the character of the string as input and traversing the tree until it reaches the element at a leaf node of the tree. Here it checks whether the value at the leaf is the same as the character. If yes, then code is added to a character array given as input.

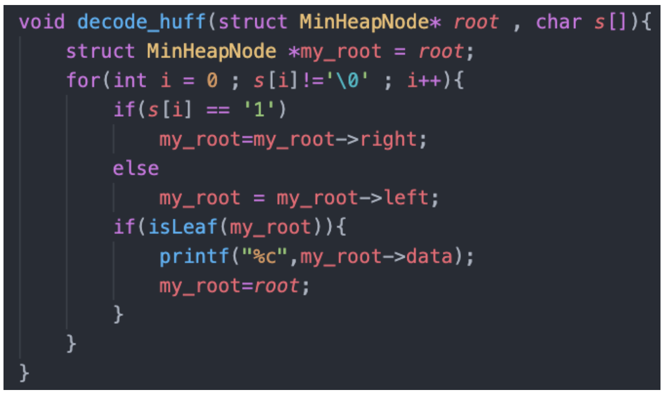
- printCodes() : It performs an operation similar to the printBin function but instead of taking single character inputs, it takes in the entire string as input. The string is traversed character by character. Each character is traversed through the tree with the general rule that 0 is assigned while traversing to the left node and 1 is assigned while traversing to the right node. The final code is obtained as a combination of all assignments upto a leaf node and is printed along with the corresponding character in the leaf node.



```c
void printCodes(struct MinHeapNode* root, int arr[],
                int top)
{
    // Assign 0 to left edge and recur
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }
    // Assign 1 to right edge and recur
    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }
    if (isLeaf(root)) {
        printf("%c : ", root->data);
        printArr(arr, top);
    }
}
```

Img 3.2.3 printCodes() function

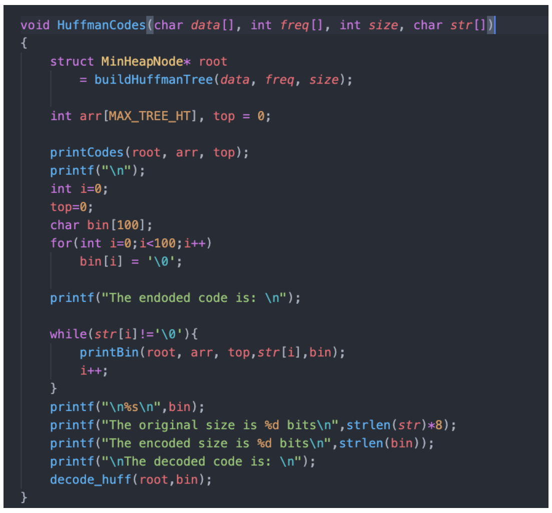
- decodeHuff() : This function works with the decoding algorithm taking the input as the encoded bits and the huffman tree. The encoded string is traversed along with the tree with the rule that a 0 bit indicated right traversal while a 1 bit indicated a left traversal. When a leaf node is reached it prints the character value of the leaf and repeats the tree traversal.



```c
void decode_huff(struct MinHeapNode* root , char s[]){
    struct MinHeapNode *my_root = root;
    for(int i = 0 ; s[i]!='\0' ; i++){
        if(s[i] == '1')
            my_root=my_root->right;
        else
            my_root = my_root->left;
        if(isLeaf(my_root)){
            printf("%c",my_root->data);
            my_root=root;
        }
    }
}
```

Img 3.2.4 decodeHuff() function

- HuffmanCodes() : This function integrates the working of all the other functions to give a sequential procedure of producing the huffman tree, printing the huffman codes and the encoded string and finally decoding it.



```c
void HuffmanCodes(char data[], int freq[], int size, char str[])
{
    struct MinHeapNode* root
        = buildHuffmanTree(data, freq, size);

    int arr[MAX_TREE_HT], top = 0;

    printCodes(root, arr, top);
    printf("\n");
    int i=0;
    top=0;
    char bin[100];
    for(int i=0;i<100;i++)
        bin[i] = '\0';

    printf("The endoded code is: \n");

    while(str[i]!='\0'){
        printBin(root, arr, top,str[i],bin);
        i++;
    }
    printf("\n%s\n",bin);
    printf("The original size is %d bits\n",strlen(str)*8);
    printf("The encoded size is %d bits\n",strlen(bin));
    printf("\nThe decoded code is: \n");
    decode_huff(root,bin);
}
```

Img 3.2.5 HuffmanCodes()

Decoding

Using Huffman tree/huffman table generated and the encoded bits of the given string, the original string can be decoded. Hence while decoding we require the encoded message as well as huffman table which has to be stored or sent along with the encoded bits.

The algorithm for decoding is:

- Take input as the encoded message and huffman tree
- The huffman tree is traversed according to the encoded message.
- Starting from the root of the tree, if the encoded bit is 0, the new root is the left child of the current root.
- If the encoded bit is 1, new root is the right child of the current root
- If we reach a leaf node, we print the value of the node and set the root to the actual root of the tree.

3.2.2 Algorithm-2 Using Queues

When is it applied?

- Applied when the given array is sorted
- Queues are used for the execution of the programs
- Leaf node is created for each character and frequency is increased on encounter with the character
- The time complexity of this algorithm is O(n)
- If the input is not sorted, it needs to be sorted first before it can be processed by the above algorithm. Sorting can be done using heap-sort or merge-sort both of which run in Theta(nlogn). So, the overall time complexity becomes O(nlogn) for unsorted input.
- This algorithm is applied only when the array is sorted else time complexity increases thus making it an efficient Huffman coding algorithm.

Algorithm

1. Create two empty queues.
2. Create a leaf node for each unique character and Enqueue it to the first queue in non-decreasing order of frequency. Initially the second queue is empty.
3. Dequeue two nodes with the minimum frequency by examining the front of both queues. Repeat following steps two times

   1. If the second queue is empty, dequeue from the first queue.

   2. If the first queue is empty, dequeue from the second queue.

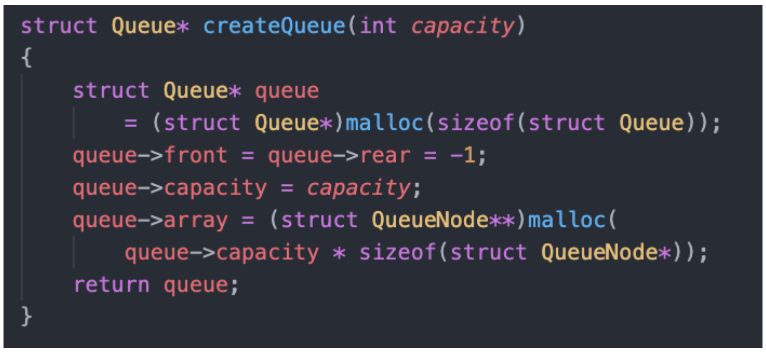
   3. Else, compare the front of two queues and dequeue the minimum.
4. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first Dequeued node as its left child and the second Dequeued node as right child. Enqueue this node to the second queue.
5. Repeat steps#3 and #4 while there is more than one node in the queues. The remaining node is the root node and the tree is complete.

The major functions used are:

The QueueNode structure represents the individual entity of the queue. It also functions as a node of the huffman tree with left and right attributes. The node contains fields for the unique character value it stores and the corresponding frequency.

The Queue structure represents a queue entity. It contains details of the front and rear ends of the queue and its capacity(maximum size). It also contains the array which implements the queue

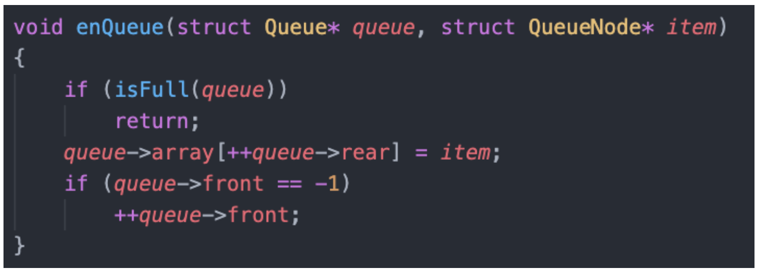
- createQueue() : An explicitly declared function which sets up a queue entity. It allocates memory for the Queue structure and assigns values to its parameters. The front and rear members are assigned -1, while the capacity is set to the maximum size value passed on to the function. A double pointer array is initialized to hold the queue nodes.



```
struct Queue* createQueue(int capacity)
{
    struct Queue* queue
        = (struct Queue*)malloc(sizeof(struct Queue));
    queue->front = queue->rear = -1;
    queue->capacity = capacity;
    queue->array = (struct QueueNode**)malloc(
        queue->capacity * sizeof(struct QueueNode*));
    return queue;
}
```

Img 3.2.6 createQueue()

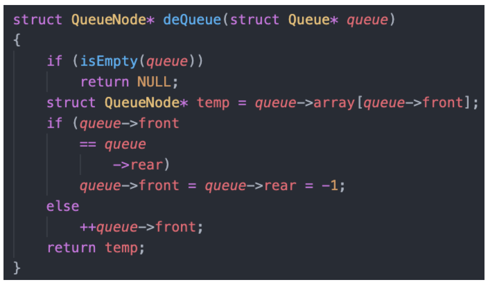
- enQueue() : This function enables the addition of a queue node item to the rear end of the queue after checking for the overflow condition.



```
void enQueue(struct Queue* queue, struct QueueNode* item)
{
    if (isFull(queue))
        return;
    queue->array[++queue->rear] = item;
    if (queue->front == -1)
        ++queue->front;
}
```

Img 3.2.7 enQueue()

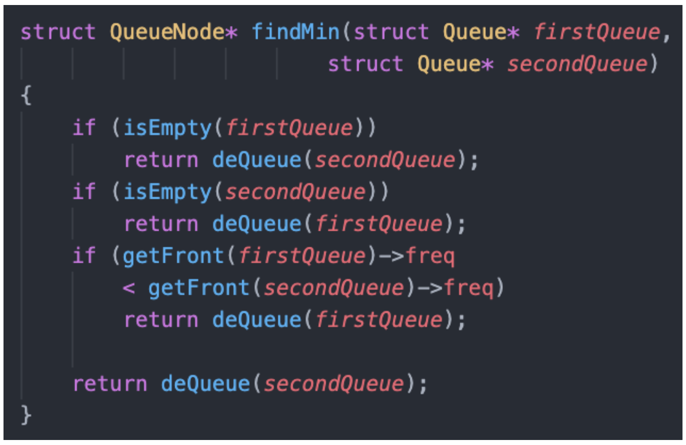
- deQueue() : Similar to enqueue, it targets to remove an element from the front end of the queue and increment the front pointer after checking for the queue empty condition.



```
struct QueueNode* deQueue(struct Queue* queue)
{
    if (isEmpty(queue))
        return NULL;
    struct QueueNode* temp = queue->array[queue->front];
    if (queue->front
        == queue
            ->rear)
        queue->front = queue->rear = -1;
    else
        ++queue->front;
    return temp;
}
```

Img 3.2.8 deQueue()

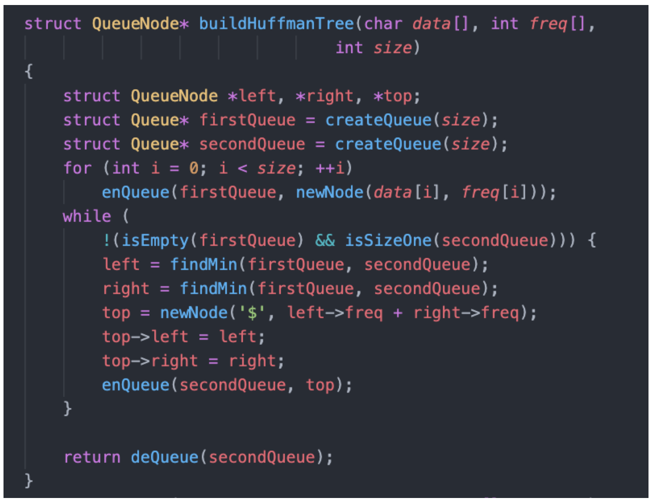
- findMin() : It is an essential function to find the minimum node values in the two queue scheme which is required for the huffman coding concept. Since the inputs are sorted the front of the queue has the minimum values. The function internally calls the dequeue function for those of the two queues which have the minimum value.



```
struct QueueNode* findMin(struct Queue* firstQueue,
                          struct Queue* secondQueue)
{
    if (isEmpty(firstQueue))
        return deQueue(secondQueue);
    if (isEmpty(secondQueue))
        return deQueue(firstQueue);
    if (getFront(firstQueue)->freq
        < getFront(secondQueue)->freq)
        return deQueue(firstQueue);

    return deQueue(secondQueue);
}
```

Img 3.2.7 findMin()

- buildHuffmanTree() : This function uses queues to build the huffman tree. The first queue stores the character nodes in their sorted input order of their frequencies. The second queue is initially empty. The two minimum values in the two queues are obtained by comparing their front values twice. These values are added and enqueued to the second queue. This process continues until only one node remains in the second queue.



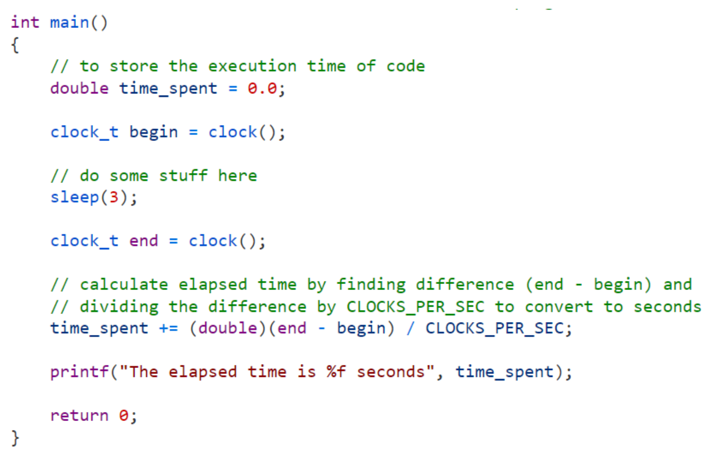
```
struct QueueNode* buildHuffmanTree(char data[], int freq[],
                                   int size)
{
    struct QueueNode *left, *right, *top;
    struct Queue* firstQueue = createQueue(size);
    struct Queue* secondQueue = createQueue(size);
    for (int i = 0; i < size; ++i)
        enQueue(firstQueue, newNode(data[i], freq[i]));
    while (
        !(isEmpty(firstQueue) && isSizeOne(secondQueue))) {
        left = findMin(firstQueue, secondQueue);
        right = findMin(firstQueue, secondQueue);
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        enQueue(secondQueue, top);
    }

    return deQueue(secondQueue);
}
```

IMG 3.2.8 buildHuffmanTree()

3.2.3 Comparing algorithm-1 and algorithm-2

The time complexities of the two algorithms are calculated and using the clock function the real time taken is measured.

The time of execution of the two algorithms is measured using the below mentioned clock function to compare their time complexities.



```
int main()
{
    // to store the execution time of code
    double time_spent = 0.0;

    clock_t begin = clock();

    // do some stuff here
    sleep(3);

    clock_t end = clock();

    // calculate elapsed time by finding difference (end - begin) and
    // dividing the difference by CLOCKS_PER_SEC to convert to seconds
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

    printf("The elapsed time is %f seconds", time_spent);

    return 0;
}
```

The clock() function is defined in the ctime header file. The clock() function returns the approximate processor time that is consumed by the program. The clock() time depends upon how the operating system allocates resources to the process, that's why clock() time may be slower or faster than the actual clock

3.3 Image compression using Huffman coding

Huffman coding proves to be an efficient technique to perform lossless compression of data, especially images. This is particularly useful in transmission and storage of data where bits represent data. The idea to generate variable length bit coded data helps to reduce redundancy and increase space required and the subsequent time.

The above comparisons of time complexities are enough proof for us to go with the queue implementation of the arrays. Alternatively, according to the form of data in this application(array of image pixels), we make use of the array implementation of the huffman tree. The need for the min Heapify function is eliminated by explicitly sorting the input pixel intensity values according to their frequencies which provides for a better time complexity effectively. The utilization of algorithm-1 or algorithm-2 is purely situation dependent. For example, in case of large data, the sorting algorithm may itself take a lot of time thereby reducing the performance of the encoding.

Algorithm for compression(encoding) of images using huffman code

- **Step 1 :**

  Read the Image into a 2D array(image) using bmp format of image
  Create a Histogram of the pixel intensity values present in the Image
  Find the number of pixel intensity values having non-zero frequency of occurrence
- **Step 2 :**

  Define a struct which will contain the pixel intensity values(pix), their corresponding frequencies, the pointer to the left(*left) and right(*right) child nodes and also the string array for the Huffman code word(code).
- **Step 3 :**

  Define another Struct which will contain the pixel intensity values(pix), their corresponding frequencies and an additional field, which will be used for storing the position of new generated nodes
- **Step 4 :**

  Declaring an array of structs. Each element of the array corresponds to a node in the Huffman Tree.
  Initialize the two arrays pix_freq and huffcodes with information of the leaf nodes.
  Sorting the huffcodes array according to the frequency of occurrence of the pixel intensity values
- **Step 5 :**

  Start by combining the two nodes with lowest frequencies of occurrence and then replacing the two nodes by the new node. This process continues until we have a root node. The first parent node formed will be stored at index node in the array pix_freq and the subsequent parent nodes obtained will be stored at higher values of index.
- **Step 6 :**

  Backtrack from the root to the leaf nodes to assign code words
  Encode the Image into a text file.

Decoding the Image
Using the decompression algorithm discussed earlier, using the generated Huffman tree and the encoded bits, we can obtain the decoded image array. This array is checked with the original array for validation of the decoding.

Main sections defined are :

- strconcat() : It takes in two strings and one character as the input. The second string argument is concatenated with the character and stored in the first string.

- Reading the image pixels into an image array using the fseek and fread functions. This image array is traversed and printed.

- A histogram array which stores the unique pixel values and their corresponding frequencies. It stores the zero as well as non zero occurring pixels.

- A pixfreq structure is defined which stores the pixel intensity value, the frequency for that pixel intensity, the left and right children of the node and the code for that pixel value.

- A huffcode node is defined with the attributes of pixel intensity, the corresponding frequency and the location of that pixel value in the pixfreq array.

- Two arrays, one having its individual element as huffcode node and the other having its individual element as pixfreq structure are initialized with each unique pixel value along with the corresponding frequency.

- The next section involves sorting the huffcode array in the descending order.

- The build huffman tree section involves making the huffman tree by traversing through the huffman codes sorted array and combining the minimum frequency nodes in the array. This combined frequency is inserted into the huffman code array such that its order is maintained. The combined frequency node is also appended to the pix freq array in this section setting the node's right and left attributes as the two minimum frequencies which gave the appended node.

- Using backtracking the huffman tree generated is iterated to obtain the huffman codes which are further displayed.

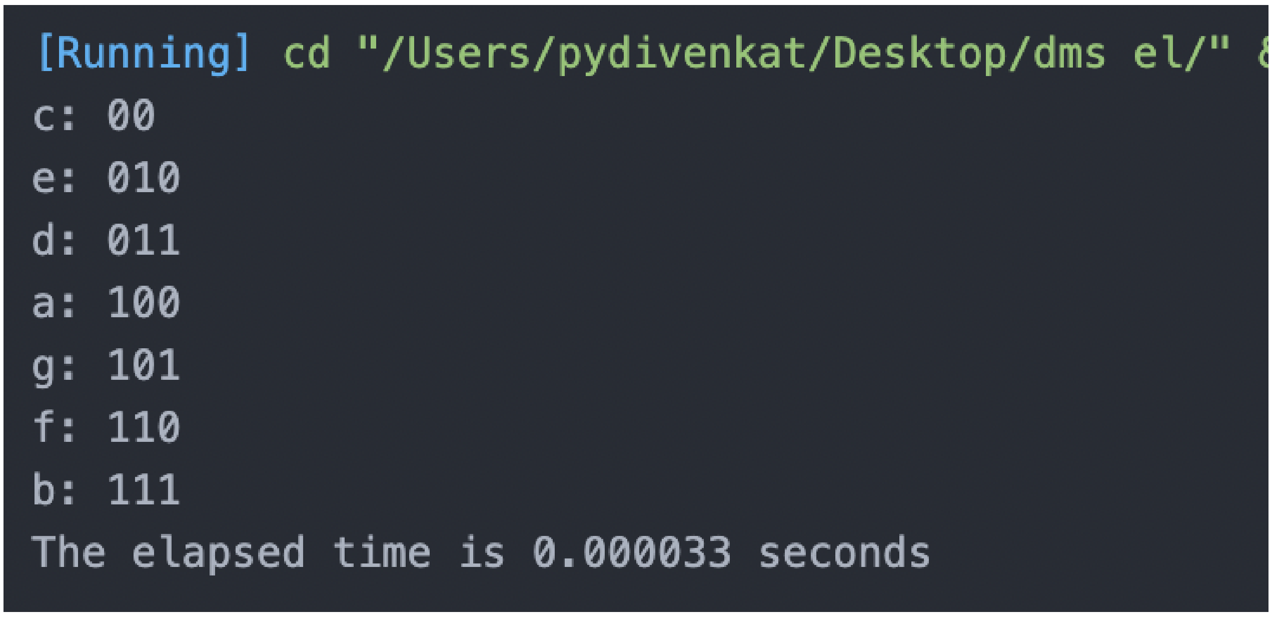
Chapter 4

Results and Snapshots

This chapter includes the final results of the results of the three programs that were run to

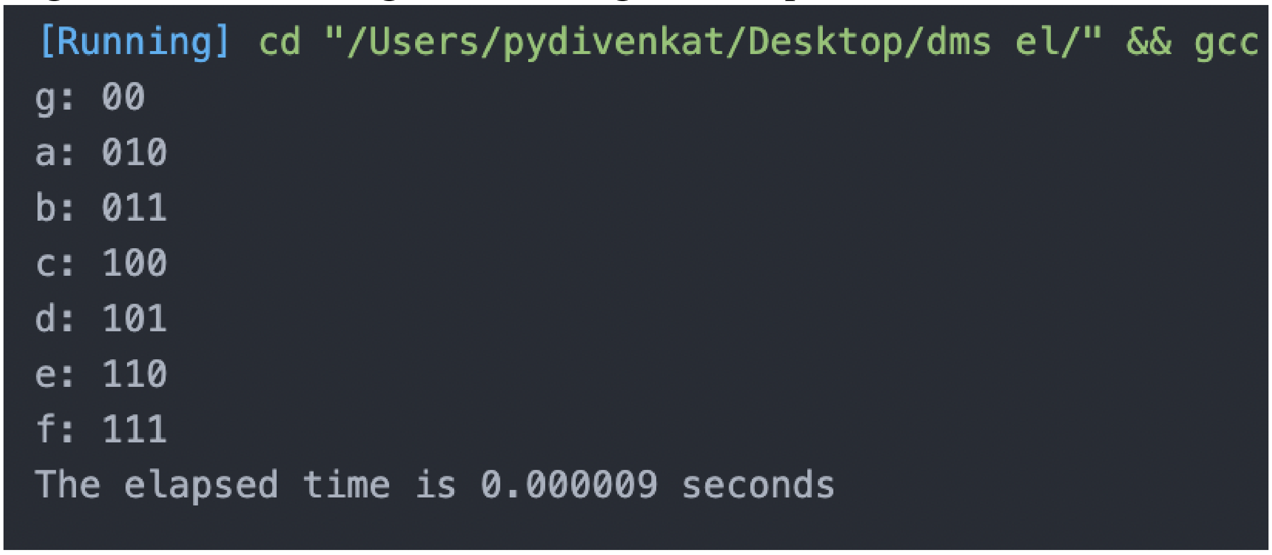
- Read the Huffman table generated using Min-Heaps and Queues
- Apply Huffman coding in encoding and decoding a given string
- Compare the time complexity in generating Huffman table generated using Min-Heaps and Queues
- To apply image compression and decompression using Huffman coding and verifying whether the decompressed image is the same image as the original.

4.1 Comparision of Time complexity



```
[Running] cd "/Users/pydivenkat/Desktop/dms el/" &
c: 00
e: 010
d: 011
a: 100
g: 101
f: 110
b: 111
The elapsed time is 0.000033 seconds
```

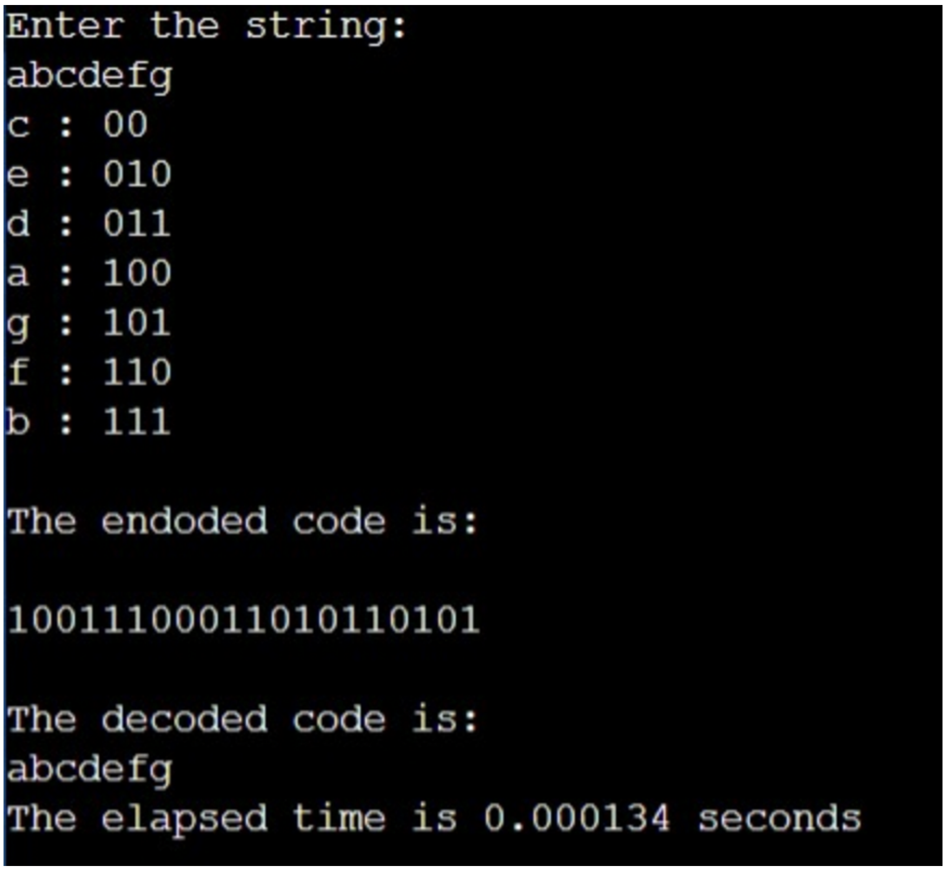
Img 4.1 Huffman Table generated using Min-Heaps



```
[Running] cd "/Users/pydivenkat/Desktop/dms el/" && gcc
g: 00
a: 010
b: 011
c: 100
d: 101
e: 110
f: 111
The elapsed time is 0.000009 seconds
```

Img 4.2 Huffman Table generated using Queues

It is found that the time taken in generating the Huffman Table using Min-Heaps is 0.000033 seconds whereas the time taken in generating the Huffman Table using Queues is 0.000009 seconds thus proving that Huffman Coding using queues is far more efficient than Huffman Coding using Min-Heaps and hence it can be named Efficient Huffman Coding.



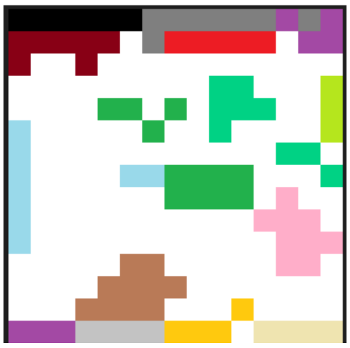
```
Enter the string:
abcdefg
c : 00
e : 010
d : 011
a : 100
g : 101
f : 110
b : 111

The endoded code is:

10011100011010110101

The decoded code is:
abcdefg
The elapsed time is 0.000134 seconds
```

Img 4.3 Encoding and decoding of string

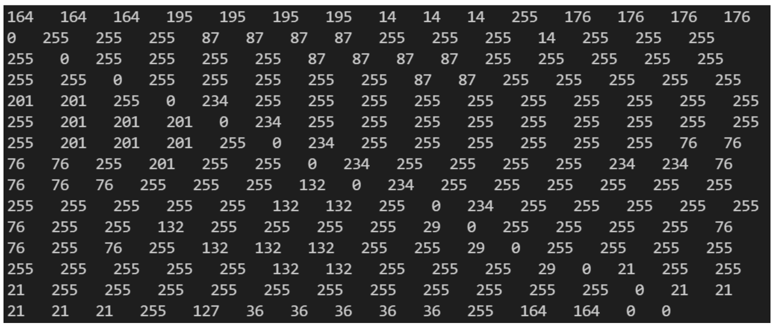
The above image is the output for encoding and decoding of a string which displays the Huffman Table and encoded code in binary and the decoded string which shows that there is no loss of data and hence proving Huffman coding is lossless technique and the above image also shows the time taken in printing the Huffman table and encoding and decoding the given string which is 0.000134 seconds.

4.2 Image Compression

Img 4.4

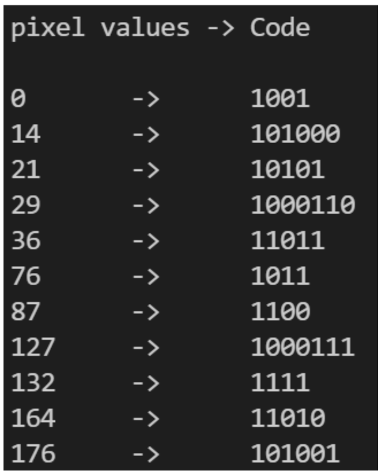


Img 4.5 Generated Image Matrix

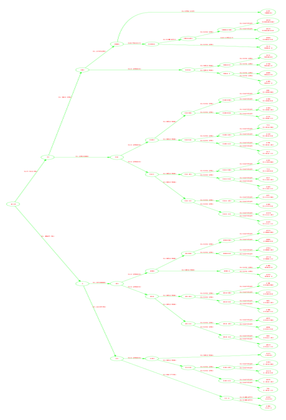


```
164   164   164   195   195   195   195   14   14   14   255   176   176   176   176
0   255   255   255   87   87   87   87   255   255   255   14   255   255   255
255   0   255   255   255   255   87   87   87   87   255   255   255   255   255
255   255   0   255   255   255   255   255   87   87   255   255   255   255   255
201   201   255   0   234   255   255   255   255   255   255   255   255   255   255
255   201   201   201   0   234   255   255   255   255   255   255   255   255   255
255   201   201   201   255   0   234   255   255   255   255   255   255   76   76
76   76   255   201   255   255   0   234   255   255   255   255   234   234   76
76   76   76   255   255   255   132   0   234   255   255   255   255   255   255
255   255   255   255   255   132   132   255   0   234   255   255   255   255   255
76   255   255   132   255   255   255   255   29   0   255   255   255   255   76
76   255   76   255   132   132   132   255   255   29   0   255   255   255   255
255   255   255   255   255   132   132   255   255   255   29   0   21   255   255
21   255   255   255   255   255   255   255   255   255   255   255   0   21   21
21   21   21   255   127   36   36   36   36   36   255   164   164   0   0
```

The output shown in the previous page displays the pixel values in the image array obtained from the original image in its bmp format. The original image shown is a 15 x 15 image taken for better understanding and easier. The huffman table generated is displayed below

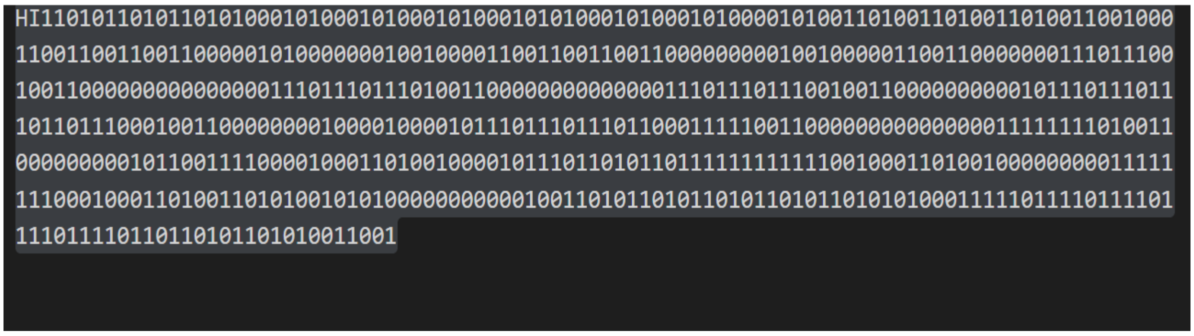


```
pixel values -> Code

0       ->      1001
14      ->      101000
21      ->      10101
29      ->      1000110
36      ->      11011
76      ->      1011
87      ->      1100
127     ->      1000111
132     ->      1111
164     ->      11010
176     ->      101001
```



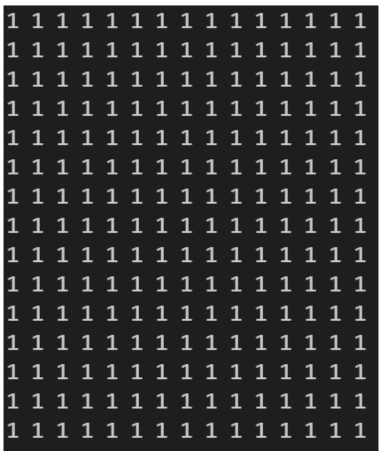
Img 4.6 Huffman Table **Img 4.7 Huffman Tree**

From the huffman table/tree generated we obtain the encoded bits



```
HI11010110101101010001010001010001010001010100010100010100001010011010011010011010011001000
1100110011001100000101000000010010000110011001100110000000001001000001100110000000111011100
1001100000000000000000111011101110100110000000000000001110111011100100110000000000101110111011
10110111000100110000000010000100001011101110111011000111110011000000000000000001111111010011
00000000101100111100001000110100100001011101101011011111111111100100011010010000000011111
11100010001101001101010010101000000000000100110101101011010110101101010100011111011110111101
1110111110110110101101010011001
```

The encoding is decoded into another array which is compared with the original array and the output is shown below. It is evident that the decoding is a perfect match.



```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Img 4.8 Testing the decided array

Chapter 5

Conclusion

Huffman Coding, with the use of Huffman Trees is a very efficient way to compress data without loss of quality or information. The greedy algorithm has proved to be one of the best methods of implementing Huffman code. Here, comparison has been drawn between the two different ways of implementation of the Huffman Coding using the greedy algorithm and the results have been analysed based on their time complexity. In the case of having sorted inputs, the algorithm using queues has proved more efficient than the one having minimum heap implementation. While the heap based approach gives a time complexity of O(nlogn), the queue based approach has a time complexity of O(n) for sorted inputs. However, the efficiency of the algorithm using queues drops drastically when the inputs are not in sorted order. Thus, the efficient algorithm finds its applications predominantly for sorted inputs.

Huffman coding has widespread applications, one of the major applications being image compression. The image compression algorithm implemented here shows the advantages of Huffman Coding in storing and transmitting data. Numerically, a data reduction of about 45 to 60% of the original image size can be attained using Huffman Coding. Although Huffman Coding is extensively used in various modern day applications, there is a lot of scope for future development. The Huffman coding approach can definitely be improvised to incorporate non-integers too in their codes. Different data structures such as priority queues and n-ary techniques can be employed in the implementation of the Huffman Code. Moreover, the speed of the encoding and decoding processes can be improved in order to increase the efficiency of an otherwise exceptional approach for data compression.

Image compression proof

Original size of image(Test2.bmp) : 311 bytes

= (311 x 8) bits

= 2488 bits

Number of bits after encoding = 574 bits

Size of the huffman tree/ huffman code table

= 4 + 15 x 8 + 74

= 198 bits

Net size of data transmitted/stored = 574 + 198 = 772 bits

Total compression relative to original size of image

= ((2488 - 772)/2488) x 100 = 65%(approx)

BIBLIOGRAPHY

References

[1] R. Channe, S. Ambatkar, S. Kakde and S. Kamble, "A Brief Review on Implementation of Lossless Color Image Compression," 2019 International Conference on Communication and Signal Processing (ICCSP), 2019, pp. 0131-0134, doi: 10.1109/ICCSP.2019.8698027.

[2] C. Narmatha, P. Manimegalai and S. Manimurugan, "A LS-compression scheme for grayscale images using pixel-based technique," 2017 International Conference on Innovations in Green Energy and Healthcare Technologies (IGEHT), 2017, pp. 1-5, doi: 10.1109/IGEHT.2017.8093980.

[3] K. Sharma and K. Gupta, "Lossless data compression technique with encryption based approach," 2017 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT), 2017, pp. 1-5, doi: 10.1109/ICCCNT.2017.8204117.

[4] Sharma, M.: “Compression Using Huffman Coding”. International Journal of Computer Science and Network Security, VOL.10 No.5, May 2010.

[5]Rabia Arshad, Adeel Saleem, Danista Khan,”Performance comparison of Huffman Coding and Double Huffman Coding”, 016 Sixth International Conference on Innovative Computing Technology (INTECH), doi:10.1109/INTECH.2016.7845058

[6]Shmuel T. Klein, Dana Shapira, ’’Huffman Coding with Non-sorted Frequencies’’, Data Compression Conference (dcc 2008), doi: 10.1109/DCC.2008.73

Appendix

The source code and the image required for this project have been uploaded to github which has the following files

1. ht_minHeap.c - Algorithm 1 for Huffman coding using Queues
2. efficeint.c - Algorithm 2 for Huffman coding using Queues
3. image_compress.c - Image Compression and Decompression using Huffman Coding
4. Test2.bmp - Test Image for Compression and Decompression
5. Team 4.pptx - PPT used for Presentation
6. Team 4.pdf - Project Report

The project can be accessed through the link given below or by scanning the QR Code Attached

https://github.com/pydivenkat/HUFFMAN-CODING-3RD-SEM-EL