**Department of Computing**

**CS250: Data Structures and Algorithms**

**Semester Project Report**

***Text File Encoder-Decoder Using Huffman Algorithm***

* **Name: Saqib Javed**
* **Reg.no: (4964)**
* **Class: BEE5-B**

Table of Contents

[1. Abstract: 3](#_Toc407579770)

[2. Introduction: 3](#_Toc407579771)

[2.1. Objective 3](#_Toc407579774)

[2.2. Problem Statement 3](#_Toc407579775)

[2.3. Scope 3](#_Toc407579777)

[3. Solution Breakdown 4](#_Toc407579778)

[3.1. Design Diagram 4](#_Toc407579780)

[3.2. Flow Chart 5](#_Toc407579781)

[3.3. Description 6](#_Toc407579782)

[4. Implementation 6](#_Toc407579783)

[4.1. Specific for each module 6](#_Toc407579788)

[5. Complexity Analysis for each algorithm 7](#_Toc407579789)

[6. Conclusion 8](#_Toc407579790)

[6.1. Pros 8](#_Toc407579793)

[6.2. Cons 8](#_Toc407579794)

[7. References 8](#_Toc407579795)

1. Abstract:  
    Huffman Encoding is a compression method whereby each character in the data is associated with a variable length prefix code. A character’s code length is determined by its relative frequency in the data. For Example, the vowels i.e. a, i, e, o, u occurs much more frequently in English Language as compared to other letters. The code word itself is derived from the Huffman tree, a binary tree whose leaves correspond to characters. The code word consists of 0s and 1s traced along the path from root to the character. Huffman Encoding is actually a technique that is used to compress files for saving memory while storing them which means that more frequently used symbols have shorter code words.
2. Introduction:

## Objective

The aim of the project was to compress the sequence of characters using Huffman Coding Algorithm. The data is encoded using Huffman tree. Then came the objective of compressing it. After decompression we achieve the sequence of 0’s and 1’s which are decoded using Huffman tree traversal.

The decoded file obtained at the end of whole execution is exact replica of original text file. Objective was to achieve such an encoded compressed file that must occupy less memory than the original.

## Problem Statement

## The main problem was to compress the file because encoding of the whole file was not a big problem whereas compression was. The size of the encoded file was increased many folds but if those 0’s and 1’s were to be considered as bits, the compression was successful. As this was not the right technique so we searched of how to write 0’s and 1’s as bits to a file but that was also not helpful.

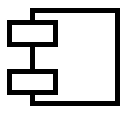
## Scope

We have implemented data structures like priority queues, binary trees (max heap) and arrays. This project also includes file handling. The algorithm that we employed is Huffman Algorithm for encoding, decoding and file compression.Although complete method for solution is given under next heading.

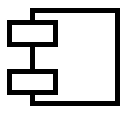
# Solution Breakdown



## Design Diagram

**compress** 

**Priority Queue**

**decompress** 

**Main control**

- \*front : Tree\_node

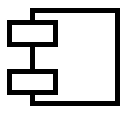
- \*rear : Tree\_node

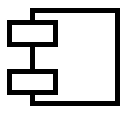
+ enqueue(Tree\_node \*) :void

+ dequeue() : Tree\_node\*

+ build\_tree() : Tree\_node\*

+ display() : void

**decode** 

**encoding** 

Priority queue is a class whereas compress, decompress, decode and encode are global functions also known as components.

The functions of class interacts with these global functions in order to accomplish the task of encoding, compression, decompression and decoding.

## Flow Chart

Start

**1**

Description of Huffman Algorithm

1- Description

**OR**

2-Compression

**2**

Text File

Read the whole file and then encode it

Compresses the encoded file

Decompresses the compressed file

End

Decode the decompressed file

## Description

First of all we built a priority queue out of the collected frequencies of all the characters in the text file.

Then we built Huffman tree out of it.

This Huffman tree was used to generate codes for characters.

Encoded file was written at the back end in text mode.

This encoded file helped in writing compressed file which is decompressed in later stage.

The decompressed file is then decoded with the help of Huffman tree.

The file obtained was copy of original text file.

# Implementation



## Specific for each module

**Building Priority Queues:**

In the beginning we have a text file whose name is being entered by the user. As he enters the name, the file is opened and read to count the frequencies of each character in the file. After that only those characters that occurred in the file at least once, they are sent to the enqueue function of priority queue class. Similarly, all the nodes are enqueued. Important thing to notice is that priority queue maintains the structure as to keep the high frequency character at the rear and low frequency character at front which will help later to make max heap easily.

**Huffman Tree:**

Easy way to make a Huffman tree out of the priority queue was to dequeue 2 nodes each time. Their frequencies were summed up and that sum was assigned to a new node. This new node was later on enqueued as well. At the last we were left with only single node in the queue that was the root of our Huffman tree which is max heap in its nature.

**Encoding Function:**

To build codes for each character, we had to traverse the Huffman tree. Every time we move left we append 0 with the code string and if we move right we append 1 with it. And when we reach leaf node we assign it that code string. This encoding function is recursive in nature.

**Compression:**

For the sake of compression, we picked 8 message bits each time from encoded file, created a char variable, whose bits were made equal to those message bits and then wrote that character in the new compressed file which is opened in binary mode. This procedure is followed for all the possible octets in the encode file. This file occupies less memory than that of encoded file.

**Decompression:**

To decompress the file, we picked each character from compressed file and declare a char variable with bits equal to 10000000 and do bit wise ‘AND’ with each bit of picked character. This helps us to identify sequence of 0’s and 1’s. These 0’s and 1’s are then written to a decompressed file. This sequence of steps were repeated till the end of compressed file. At the end, we have a decompressed file which is exact copy of encoded file.

**Decoding Function:**

In order to decode an encoded file, we simply have to traverse the Huffman tree in order of 0’s and 1’s that occur in the encoded file. For 0 we move left whereas for 1 we move right in the tree. Where ever we encounter the leaf node, we write its character in the decoded file. We repeat this sequence of steps until the whole encoded file is read. At last the decoded file is achieved that is exactly the replica of original text file.

# Complexity Analysis for each algorithm

**Reading Characters from file:**

O (n\*m) where n is number of sentences in the text file and m is the length of sentence

**Priority Queues:**

Insertion: O (n-1) where n is the number of nodes already present in it.

Deletion: O (1) because deletion is always at front end.

**Encoding:**

O (n\*h) where n is number of characters in the text file and h is the height of the Huffman tree.

**Compression:**

O (n + m) where n is number of octets in the encoded file and m is number of message bits that are left.

**Decompression:**

O (n + m) where n is number of octets in the encoded file and m is number of message bits that are left.

**Decoding:**

O (b) where b is number of bits (0’s and 1’s) in encoded file.

# Conclusion

## Pros

The strong point of our project lies in the concept that file compression is being done and file is decoded successfully. The two global functions that are COMPRESS and DECOMPRESS are the core functions that help in saving the file with less memory. The compresses file is written in binary mode so security aspect is also fulfilled.

## Cons

Although we have succeeded in compression but the way we did compression is not more efficient that it would have been if directly 0’s and 1’s were to be written in a file as BITS but not as BYTES. We did not find the way of how to implement it but if someone manages to store them in a file as BITS, the complexity of compressing function and decompressing function can be subtracted from the whole code complexity thereby making it more efficient.

# References

<http://michael.dipperstein.com/huffman/>

<http://www.cs.auckland.ac.nz/~jmor159/PLDS210/huffman.html>

<http://msdn.microsoft.com/en-us/library/336xbhcz.aspx>