# CHAPTER-1

**Introduction**

## Introduction

Text files can be compressed to make them smaller and faster to send, and unzipping files on devices has a low overhead. The process of encoding involves changing the representation of a file so that the compressed output takes less space to transmit while retaining the ability to reconstruct the original file exactly from its compressed representation.

The size of the text file can be reduced by compressing it, which converts the text to a smaller format that takes up less space. It typically works by locating similar strings/characters within a text file and replacing them with a temporary binary representation to reduce the overall size. There are two types of file compression, namely Lossy Compression and Lossless compression.

Lossy compression shrinks a file by permanently removing certain elements, particularly redundant elements.

Lossless compression can restore all elements of a file during decompression without sacrificing data and quality.

The main goal of this work is to build a file compression model using lossless compression using the Huffman coding algorithm. Huffman coding is a lossless data compression algorithm, a technique of compressing data to reduce the size of the data without any loss in the details present. Huffman coding is generally useful to compress the data in which there are frequently occurring characters.

## Existing System

* + 1. WinZip - One of the most famous names in the world of software utilities, WinZip is still going strong after nearly 30 years, and is still one of the best file compression tools around.

Features of WinZip:

* + - * Supports many file types.
      * Split large files.
      * Advanced management tool
    1. WinRar **-** As famous as WinZip in certain circles, WinRAR created a name for itself thanks to its proprietary RAR format, which offers incredible levels of compression. Most compression programs can extract RAR archives, but only WinRAR can (officially) create them.

WinRar features including:

* + - * High compression rate
      * Works with multiple formats
      * Create RAR files
    1. 7Zip - The first free option in this roundup, 7-Zip is another program with an excellent reputation. It can handle pretty much any compressed file format you care to throw at it.

Main options of 7Zip are:

* + - * Free software
      * Own formats
      * Great for huge files

* + 1. Zip Archiver - It might not be the first name that comes to mind when you think of file compression software, but Zip Archiver has a very healthy following thanks to its thoughtfully designed interface, excellent format support, and because it offers all this for free.

The advantages of Zip Archiver are:

* + - * Free Software.
      * Different formats.
      * Drag and Drop feature

## Problems in Existing Definition

* The application is open source and easy to breach.
* They are paid and charge high to their consumers. For those who don’t use these much often finds it quite expensive.
* Free software takes time for compression and for decompression as well.
* UI of these software are complex and user may find them hard to use. If a software isn’t simple and easy to use then user might want an alternate to switch to any other.
* These software are available to all so in case anyone wants to use them can directly purchase their services and use them.
* Some applications allows only a limited size file to be transferred, to overcome it we need a solution.
* Sharing information over the internet is not safe these days and we need a software that provides a more secure and confidential way to share information along with compression.

## Problem Definition

To design a software that can solve both the problems of compression of data and to retain the confidentiality of the software while sharing the information over the internet. So even in case of data break, the attacker won’t be able to retrieve the personal or confidential information.

## Feasibility Study

There are n number of compression software available in the market but they are available to all. They can easily be used by purchasing their services. Many of these are highly expensive or quite complex to use. Aimis to provide a simple software to overcome such problems and develop a better solution for the same.

## Motivation

The motivation to pursue this project comes from the existing systems that we use. Hard to retain the confidentiality of the data. In case a wrong recipient got the file, they might exploit it and use it for their own benefits.

## Project Overview

This project will try to solve the problem of confidentiality of the data along with the compression of it to save the data transmission. Some software have limited size files that can be transferred such as gmail, etc.

The technology used in order to build this project includes IDEs like Visual Studio Code, Programming languages like C++ and detailed knowledge about the data structures and algorithms.

## Hardware Specification

* A system with Windows/Linux operating System.
* Processor with more 1.7 GHz speed.
* Minimum 4 GB of Memory.
* Minimum 5 GB of empty Storage.
* An Integrated Graphic card.

## Software Specification

* + 1. **Visual Studio Code** - Visual Studio Code, additionally usually known as VS Code, is a source-code editor made via way of means of Microsoft for Windows, Linux and macOS. It consists of functions like aid for debugging, syntax highlighting, wise code completion, snippets, code refactoring, and embedded Git [1].

The setup process of Visual Studio Code is as follows:

* Open the official website of Visual Studio Code to download the software.

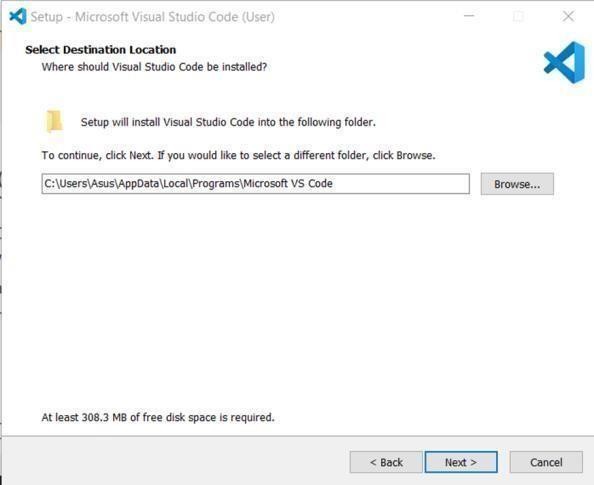
Figure 1.1 Shows the various versions available to download for different OS.



**Figure 1.1:** Downloading VS Code (*Source*:[self])

* Now select the version and OS you want to install for and click on download.
* After the file downloads click on setup and then agree to the license agreements.
* Select the destination folder for saving the VS code files and click on next.

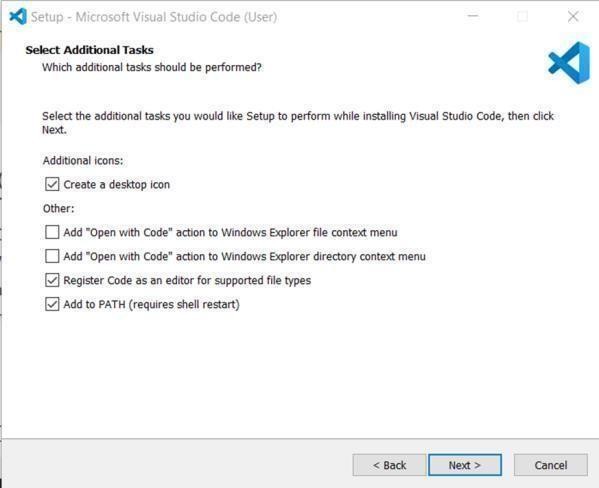
Figure 1.2 Shows the installation wizard window for VS Code.



**Figure 1.2:** Installing VS Code (*Source*:[self])

* In the additional tasks window confirm that the “Add to PATH” is selected.

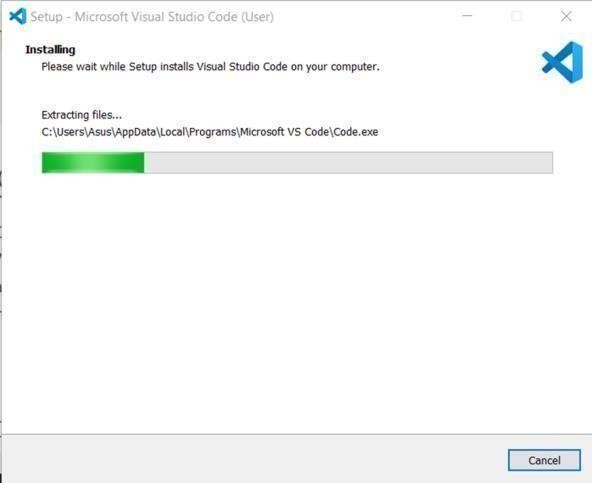
Figure 1.3 Shows the Additional tasks that can be selection during installation.



**Figure 1.3:** Configuring VS Code (*Source*:[self])

* Now complete the installation process by clicking on Install, it should take a few mins to finish installation.

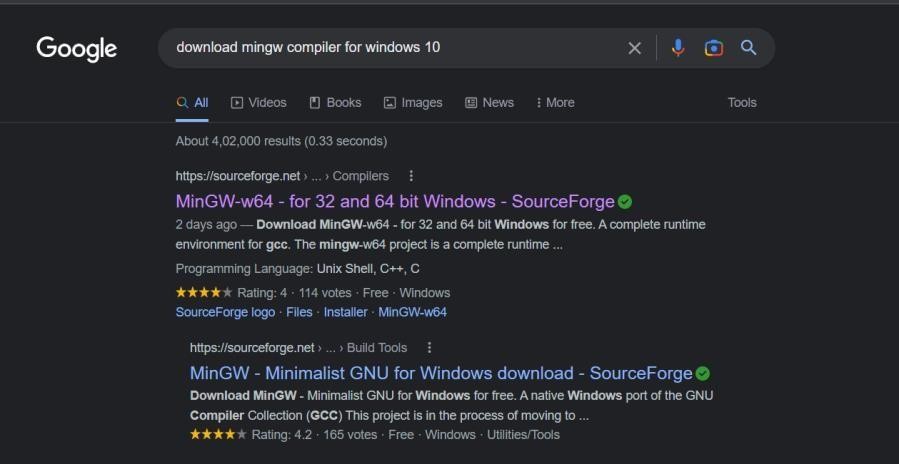
Figure 1.4 shows the progress bar of Visual Code installation.



**Figure 1.4:** Finishing VS Code Installation (*Source*:[1])

* + 1. **Mingw Compiler –** Mingw is the most widely used compiler used for various programming language such as C++, C and etc. We have to simply install it on our system.
* First search on google “Download mingw Compiler” and click on the first link.

Figure 1.5 shows the web result of downloading mingw command.

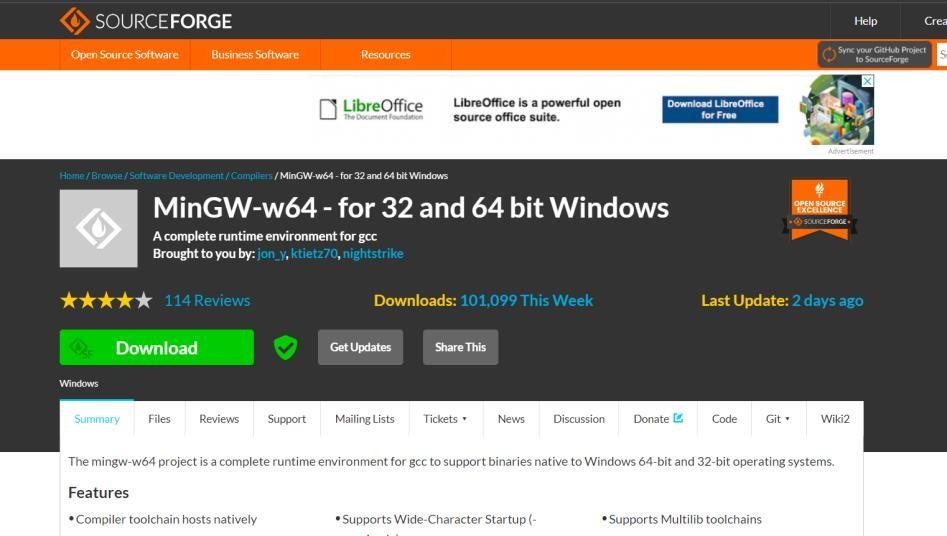


**Figure 1.5:** Looking for Mingw on web (*Source*:[self])

* Click on the green download button to download the mingw compiler setup.

Download the correct compiler as per the operating window bit whether it is 32-bit or 64-bit.

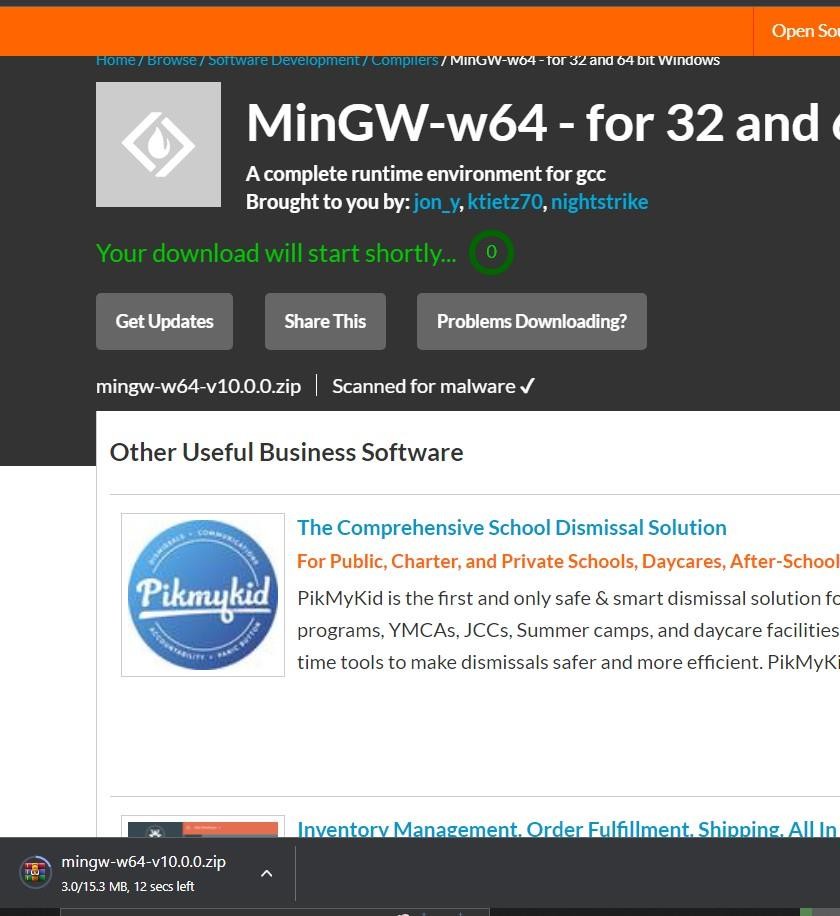
Figure 1.6 shows the website available to download mingw.



**Figure 1.6:** Selecting version of Mingw on web (*Source*:[self])

* A zip file will start to download on your system in bottom left corner and let it complete.

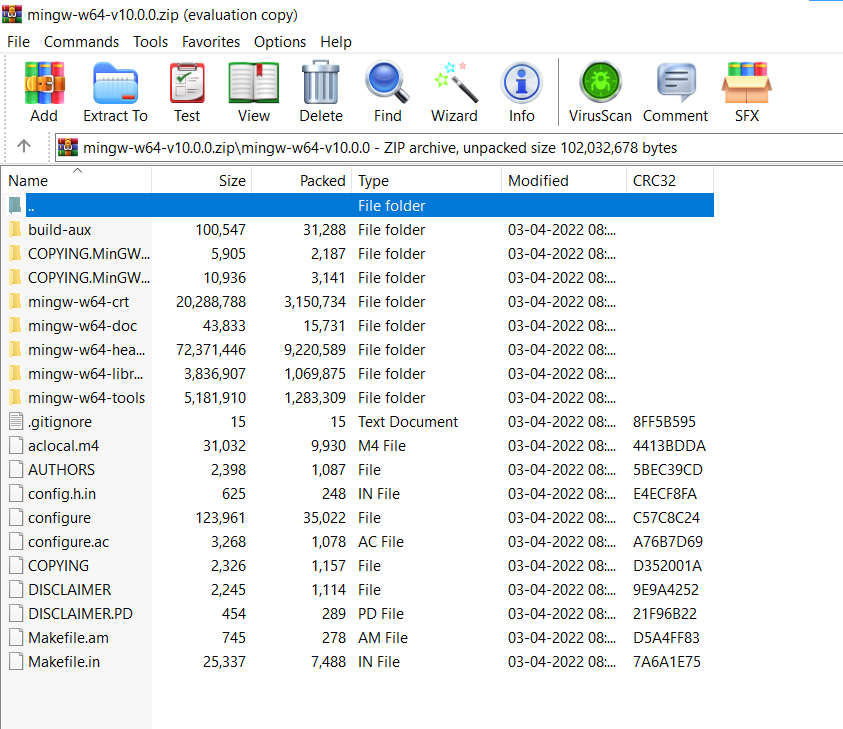
Figure 1.7 shows the downloading window.



**Figure 1.7:** Downloading Mingw (*Source*:[self])

* Now open the .zip file with the help of any too land extract the files.

Figure 1.8 shows the files available after downloading mingw setup.



**Figure 1.8:** Extracting file of MingW (*Source*:[self])

* Now begin the installation wizard and simply click on next to install the mingw compiler on your system.

## Overview of the report

This document is a comprehensive report on the existing compression software. It takes a short glance at their features as well as limitations. Additionally, it includes a catalogue of attributes that will be incorporated in our final product along with the construction steps, procedure, execution and culmination.

# Chapter 2

**System Analysis and Design**

## Requirement Specifications

This project “ZIPPER” is an application and therefore can be run on any device with any OS. It is recommended to use the latest versions of the OS to avoid and crashes. This application does not require the resources of the host machine for any sort of database storing or for performing any computing tasks.

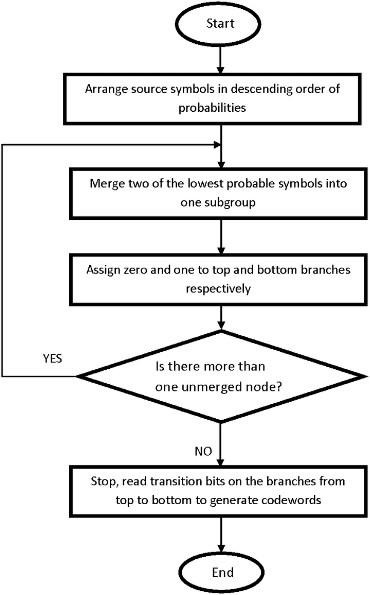
## Flowcharts

Given below is a visual representation of the workflow of the software as well as how the data is acquired:

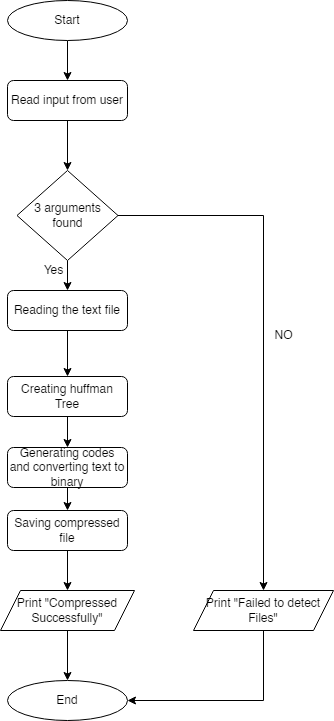
In Figure 2.1 the work flow of how the Huffman tree is created. It is created with the help of frequencies of the character stored in the min heap data structure and then that min heap helps in assigning the position of the each character in the Huffman tree.

Figure 2.2 shows the user workflow of the encoder application. The application simply checks for the number of arguments that user inputs and then on the basis of these arguments further flow of codes takes the respective functions are called.

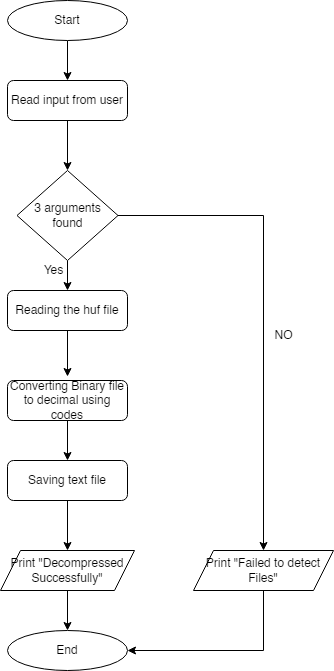
In Figure 2.3 shows the user workflow of the decoder application. The application simply checks for the number of arguments that user inputs and then on the basis of these arguments further flow of codes takes the respective functions are called to perform decryption of the compressed file.



**Figure 2.1:** Construction of Huffman Tree (*Source*:[Self])



**Figure 2.2:** Flowchart of Encode.cpp (*Source*:[Self])



**Figure 2.3:** Flowchart of Decode.cpp (*Source*:[Self])

## Design Steps

In order to fulfill the objectives proposed at the early stages of the project each step of the designing process needs to be meticulously carried out to avoid any issues in the later stages.

The steps carried out include:

* + - Coming up with an appropriate project title that can convey the primary objective of the application and sounds aesthetically pleasing.
    - Researching similar products and solutions offered by other companies and listing out their pros and cons.
    - Coming up with a basic list of features that need to be implemented in the prototype.
    - Creating a properly structured design for the software with details.
    - Finalizing the technology to be used after discussion and beginning the process of building web pages
    - Writing down the existing software and technologies that can be used to build the app according to the desired specifications and ideas.

## Pseudo Code

This section will comprise of the sections to be covered in the software.

## Steps for Building Huffman Tree

* + - * The input to the algorithm is the array of characters in the text file.
      * The frequency of occurrences of each character in the file is calculated.
      * Struct array is created where each element includes the character along with their frequencies. They are stored in a priority queue (min-heap), where the elements are compared using their frequencies.
      * To build the Huffman tree, two elements with minimum frequency are extracted from the min-heap.
      * The two nodes are added to the tree as left and right children to a new root node which contains a frequency equal to the sum of two frequencies. A lower frequency character is added to the left child node and the higher frequency character to the right child node.
      * The root node is then again added back to the priority queue.
      * Repeat step 4 until there is only one element left in the priority queue.
      * Finally, the tree’s left and right edges are numbered 0 and 1, respectively. For each leaf node, the entire tree is traversed, and the corresponding 1 and 0 are appended to their code until a leaf node is encountered.
      * Once we have the unique codes for each unique character in the text, we can replace the text characters with their codes. These codes will be stored in bit-by-bit form, which will take up less space than text.

## Encryption

Encryption is the method by which information is converted into secret code that hides the information's true meaning. The science of encrypting and decrypting information is called cryptography.

In computing, unencrypted data is also known as plaintext, and encrypted data is called cipher text. The formulas used to encode and decode messages are called encryption algorithms, or ciphers.

Encryption plays an important role in securing many different types of information technology (IT) assets. It provides the following:

* + - * Confidentiality encodes the message's content.
      * Authentication verifies the origin of a message.
      * Integrity proves the contents of a message have not been changed since it was sent.
      * Nonrepudiation prevents senders from denying they sent the encrypted message.

At the beginning of the encryption process, the sender must decide what cipher will best disguise the message's meaning and what variable to use to make the encoded message unique. The most widely used types of ciphers fall into two categories: symmetric and asymmetric.

## 2.4.2 Decryption

Decryption is the process of transforming data that has been rendered unreadable through encryption back to its unencrypted form. In decryption, the system extracts and converts the garbled data and transforms it into texts and images that are easily understandable not only by the reader but also by the system. Decryption may be accomplished manually or automatically. It may also be performed with a set of keys or passwords.

One of the foremost reasons for implementing an encryption-decryption system is privacy. As information travels over the World Wide Web, it becomes subject to scrutiny and access from unauthorized individuals or organizations. As a result, data is encrypted to reduce data loss and theft. Some of the common items that are encrypted include email messages, text files, images, user data, and directories. The person in charge of decryption receives a prompt or window in which a password may be entered to access encrypted information.

# Chapter 3 Implementation and Results

* 1. **Main Header File**- This is the main header file (Huffman.hpp) that is a custom designed header file for all the functions and header files that will be needed for the text compression. It is written in C++ programming language and uses the compiler to run the application. Furthermore it uses command prompt for the execution and create applications for smooth execution of multiple programs.

Here is the code file for the Huffman.hpp:

## huffman.hpp

//Header Guards to prevent header files from being included multiple times #ifndef HUFFMAN\_HPP

#define HUFFMAN\_HPP

#include <string> #include <vector> #include <queue> #include <fstream>

using namespace std;

//Defining Huffman Tree Node struct Node

{

char data; unsigned freq; string code;

Node \*left, \*right;

Node() {

left = right = NULL;

}

};

class huffman { private:

vector <Node\*> arr; fstream inFile, outFile;

string inFileName, outFileName; Node \*root;

class Compare { public:

bool operator() (Node\* l, Node\* r)

{

return l->freq > r->freq;

}

};

priority\_queue <Node\*, vector<Node\*>, Compare> minHeap;

//Initializing a vector of tree nodes representing character's ascii value and initializing its frequency with 0

void createArr();

//Traversing the constructed tree to generate huffman codes of each present character void traverse(Node\*, string);

//Function to convert binary string to its equivalent decimal value int binToDec(string);

//Function to convert a decimal number to its equivalent binary string string decToBin(int);

//Reconstructing the Huffman tree while Decoding the file void buildTree(char, string&);

//Creating Min Heap of Nodes by frequency of characters in the input file void createMinHeap();

//Constructing the Huffman tree void createTree();

//Generating Huffman codes void createCodes();

//Saving Huffman Encoded File void saveEncodedFile();

//Saving Decoded File to obtain the original File void saveDecodedFile();

//Reading the file to reconstruct the Huffman tree void getTree();

public:

//Constructor

huffman(string inFileName, string outFileName)

{

this->inFileName = inFileName; this->outFileName = outFileName; createArr();

}

//Compressing input file void compress();

//Decrompressing input file void decompress();

};

#endif

* 1. **Main Huffman Program –** This is the main program in which we defines the functions and the uses the header files that we have defined in the ‘huffman.hpp’ head file. All the function for encoding and decoding like creating the min heap, binary tree, saving codes, saving files are defined in this program.

## huffman.cpp

#include "huffman.hpp" // Custom header file void huffman::createArr()

{

for (int i = 0; i < 128; i++)

{

arr.push\_back(new Node()); arr[i]->data = i;

arr[i]->freq = 0;

}

}

void huffman::traverse(Node\* r, string str)

{

if (r->left == NULL && r->right == NULL)

{

r->code = str; return;

}

traverse(r->left, str + '0'); traverse(r->right, str + '1');

}

for (auto c : inStr)

{

res = res \* 2 + c - '0';

}

return res;

}

string huffman::decToBin(int inNum)

{

string temp = "", res = ""; while (inNum > 0)

{

temp += (inNum % 2 + '0'); inNum /= 2;

}

res.append(8 - temp.length(), '0');

for (int i = temp.length() - 1; i >= 0; i--)

{

res += temp[i];

}

return res;

}

void huffman::saveEncodedFile()

{

//Saving encoded (.huf) file inFile.open(inFileName, ios::in); outFile.open(outFileName, ios::out | ios::binary);

string in = ""; string s = "";

//Saving the meta data (huffman tree) in += (char)minHeap.size();

priority\_queue <Node\*, vector<Node\*>, Compare> tempPQ(minHeap);

while (!tempPQ.empty())

{

Node\* curr = tempPQ.top(); in += curr->data;

//Saving 16 decimal values representing code of curr->data s.assign(127 - curr->code.length(), '0');

s += '1';

s += curr->code;

//Saving decimal values of every 8-bit binary code in += (char)binToDec(s.substr(0, 8));

for (int i = 0; i < 15; i++)

{

s = s.substr(8);

in += (char)binToDec(s.substr(0, 8));

}

tempPQ.pop();

}

s.clear();

//Saving codes of every charachter appearing in the input file inFile.get(id);

while (!inFile.eof())

{

s += arr[id]->code;

in += (char)binToDec(s.substr(0, 8)); s = s.substr(8);

}

inFile.get(id);

}

//Finally if bits remaining are less than 8, append 0's int count = 8 - s.length();

if (s.length() < 8)

{

s.append(count, '0');

}

in += (char)binToDec(s);

//append count of appended 0's in += (char)count;

//write the in string to the output file outFile.write(in.c\_str(), in.size()); inFile.close();

outFile.close();

}

void huffman::saveDecodedFile()

{

inFile.open(inFileName, ios::in | ios::binary); outFile.open(outFileName, ios::out);

//Reading count at the end of the file which is number of bits appended to make final value 8- bit

inFile.seekg(-1, ios::end); char count0; inFile.read(&count0, 1);

//Ignoring the meta data (huffman tree) (1 + 17 \* size) and reading remaining file inFile.seekg(1 + 17 \* size, ios::beg);

vector<unsigned char> text; unsigned char textseg;

inFile.read(reinterpret\_cast<char\*>(&textseg), 1);

while (!inFile.eof())

{

text.push\_back(textseg); inFile.read(reinterpret\_cast<char\*>(&textseg), 1);

}

Node \*curr = root; string path;

for (int i = 0; i < text.size() - 1; i++)

{

//Converting decimal number to its equivalent 8-bit binary code path = decToBin(text[i]);

if (i == text.size() - 2)

{

path = path.substr(0, 8 - count0);

}

//Traversing huffman tree and appending resultant data to the file for (int j = 0; j < path.size(); j++) [j] == '0')

{

curr = curr->left;

}

else

{

curr = curr->right;

}

if (curr->left == NULL && curr->right == NULL)

{

outFile.put(curr->data); curr = root;

}

}

}

inFile.close(); outFile.close();

}

void huffman::getTree()

{

inFile.open(inFileName, ios::in | ios::binary);

//Reading size of MinHeap unsigned char size;

inFile.read(reinterpret\_cast<char\*>(&size), 1); root = new Node();

//next size \* (1 + 16) characters contain (char)data and (string)code[in decimal] for(int i = 0; i < size; i++)

{

char aCode;

unsigned char hCodeC[16]; inFile.read(&aCode, 1);

inFile.read(reinterpret\_cast<char\*>(hCodeC), 16);

//converting decimal characters into their binary equivalent to obtain code string hCodeStr = "";

for (int i = 0; i < 16; i++)

{

hCodeStr += decToBin(hCodeC[i]);

}

//Removing padding by ignoring first (127 - curr->code.length()) '0's and next '1' character int j = 0;

while (hCodeStr[j] == '0')

{

j++;

}

hCodeStr = hCodeStr.substr(j+1);

//Adding node with aCode data and hCodeStr string to the huffman tree buildTree(aCode, hCodeStr);

}

inFile.close();

}

void huffman::createTree()

{

//Creating Huffman Tree with the Min Heap created earlier Node \*left, \*right;

priority\_queue <Node\*, vector<Node\*>, Compare> tempPQ(minHeap);

while (tempPQ.size() != 1)

{

left = tempPQ.top(); tempPQ.pop();

root = new Node();

root->freq = left->freq + right->freq;

root->left = left; root->right = right; tempPQ.push(root);

}

}

void huffman::buildTree(char a\_code, string& path)

{

Node\* curr = root;

for (int i = 0; i < path.length(); i++)

{

if (path[i] == '0') {

if (curr->left == NULL)

{

curr->left = new Node();

}

curr = curr->left;

}

else if (path[i] == '1')

{

if (curr->right == NULL)

{

curr->right = new Node();

}

curr = curr->right;

}

}

curr->data = a\_code;

}

{

char id;

inFile.open(inFileName, ios::in); inFile.get(id);

//Incrementing frequency of characters that appear in the input file while (!inFile.eof())

{

arr[id]->freq++; inFile.get(id);

}

inFile.close();

//Pushing the Nodes which appear in the file into the priority queue (Min Heap) for (int i = 0; i < 128; i++)

{

if (arr[i]->freq > 0)

{

minHeap.push(arr[i]);

}

}

}

void huffman::createTree()

{

//Creating Huffman Tree with the Min Heap created earlier Node \*left, \*right;

priority\_queue <Node\*, vector<Node\*>, Compare> tempPQ(minHeap);

while (tempPQ.size() != 1)

{

left = tempPQ.top();

tempPQ.pop();

root = new Node();

root->freq = left->freq + right->freq;

root->left = left; root->right = right; tempPQ.push(root);

}

}

void huffman::createArr()

{

for (int i = 0; i < 128; i++)

{

arr.push\_back(new Node()); arr[i]->data = i;

arr[i]->freq = 0;

}

}

void huffman::traverse(Node\* r, string str)

{

if (r->left == NULL && r->right == NULL)

{

r->code = str; return;

}

traverse(r->left, str + '0'); traverse(r->right, str + '1');

}

for (auto c : inStr)

{

res = res \* 2 + c - '0';

}

return res;

}

string huffman::decToBin(int inNum)

{

string temp = "", res = ""; while (inNum > 0)

{

temp += (inNum % 2 + '0'); inNum /= 2;

}

res.append(8 - temp.length(), '0');

for (int i = temp.length() - 1; i >= 0; i--)

{

res += temp[i];

}

return res;

}

void huffman::saveEncodedFile()

{

//Saving encoded (.huf) file inFile.open(inFileName, ios::in); outFile.open(outFileName, ios::out | ios::binary);

string in = ""; string s = "";

//Saving the meta data (huffman tree) in += (char)minHeap.size();

priority\_queue <Node\*, vector<Node\*>, Compare> tempPQ(minHeap);

while (!tempPQ.empty())

{

Node\* curr = tempPQ.top(); in += curr->data;

//Saving 16 decimal values representing code of curr->data s.assign(127 - curr->code.length(), '0');

s += '1';

s += curr->code;

//Saving decimal values of every 8-bit binary code in += (char)binToDec(s.substr(0, 8));

for (int i = 0; i < 15; i++)

{

s = s.substr(8);

in += (char)binToDec(s.substr(0, 8));

}

tempPQ.pop();

}

s.clear();

//Saving codes of every charachter appearing in the input file inFile.get(id);

while (!inFile.eof())

{

s += arr[id]->code;

{

in += (char)binToDec(s.substr(0, 8)); s = s.substr(8);

}

inFile.get(id);

}

//Finally if bits remaining are less than 8, append 0's int count = 8 - s.length();

if (s.length() < 8)

{

s.append(count, '0');

}

in += (char)binToDec(s);

//append count of appended 0's in += (char)count;

//write the in string to the output file outFile.write(in.c\_str(), in.size()); inFile.close();

outFile.close();

}

void huffman::saveDecodedFile()

{

inFile.open(inFileName, ios::in | ios::binary); outFile.open(outFileName, ios::out);

//Reading count at the end of the file which is number of bits appended to make final value 8- bit

inFile.seekg(-1, ios::end); char count0; inFile.read(&count0, 1);

//Ignoring the meta data (huffman tree) (1 + 17 \* size) and reading remaining file inFile.seekg(1 + 17 \* size, ios::beg);

vector<unsigned char> text; unsigned char textseg;

inFile.read(reinterpret\_cast<char\*>(&textseg), 1);

while (!inFile.eof())

{

text.push\_back(textseg); inFile.read(reinterpret\_cast<char\*>(&textseg), 1);

}

Node \*curr = root; string path;

for (int i = 0; i < text.size() - 1; i++)

{

//Converting decimal number to its equivalent 8-bit binary code path = decToBin(text[i]);

if (i == text.size() - 2)

{

path = path.substr(0, 8 - count0);

}

//Traversing huffman tree and appending resultant data to the file

if (path[j] == '0')

{

curr = curr->left;

}

else

{

curr = curr->right;

}

if (curr->left == NULL && curr->right == NULL)

{

outFile.put(curr->data); curr = root;

}

}

}

inFile.close(); outFile.close();

}

void huffman::createCodes()

{

//Traversing the Huffman Tree and assigning specific codes to each character traverse(root, "");

}

void huffman::saveEncodedFile()

{

//Saving encoded (.huf) file inFile.open(inFileName, ios::in); outFile.open(outFileName, ios::out | ios::binary);

string s = "";

char id;

//Saving the meta data (huffman tree) in += (char)minHeap.size();

priority\_queue <Node\*, vector<Node\*>, Compare> tempPQ(minHeap);

while (!tempPQ.empty())

{

Node\* curr = tempPQ.top(); in += curr->data;

//Saving 16 decimal values representing code of curr->data s.assign(127 - curr->code.length(), '0');

s += '1';

s += curr->code;

//Saving decimal values of every 8-bit binary code in += (char)binToDec(s.substr(0, 8));

for (int i = 0; i < 15; i++)

{

s = s.substr(8);

in += (char)binToDec(s.substr(0, 8));

}

tempPQ.pop();

}

s.clear();

//Saving codes of every charachter appearing in the input file inFile.get(id);

s += arr[id]->code;

//Saving decimal values of every 8-bit binary code while (s.length() > 8)

{

in += (char)binToDec(s.substr(0, 8)); s = s.substr(8);

}

inFile.get(id);

}

//Finally if bits remaining are less than 8, append 0's int count = 8 - s.length();

if (s.length() < 8)

{

s.append(count, '0');

}

in += (char)binToDec(s);

//append count of appended 0's in += (char)count;

//write the in string to the output file outFile.write(in.c\_str(), in.size()); inFile.close();

outFile.close();

}

void huffman::saveDecodedFile()

{

outFile.open(outFileName, ios::out);

unsigned char size; inFile.read(reinterpret\_cast<char\*>(&size), 1);

//Reading count at the end of the file which is number of bits appended to make final value 8- bit

inFile.seekg(-1, ios::end); char count0; inFile.read(&count0, 1);

//Ignoring the meta data (huffman tree) (1 + 17 \* size) and reading remaining file inFile.seekg(1 + 17 \* size, ios::beg);

vector<unsigned char> text; unsigned char textseg;

inFile.read(reinterpret\_cast<char\*>(&textseg), 1);

while (!inFile.eof())

{

text.push\_back(textseg); inFile.read(reinterpret\_cast<char\*>(&textseg), 1);

}

Node \*curr = root; string path;

for (int i = 0; i < text.size() - 1; i++)

{

//Converting decimal number to its equivalent 8-bit binary code path = decToBin(text[i]);

if (i == text.size() - 2)

{

//Traversing huffman tree and appending resultant data to the file for (int j = 0; j < path.size(); j++)

{

if (path[j] == '0')

{

curr = curr->left;

}

else

{

curr = curr->right;

}

if (curr->left == NULL && curr->right == NULL)

{

outFile.put(curr->data); curr = root;

}

}

}

inFile.close(); outFile.close();

}

void huffman::getTree()

{

inFile.open(inFileName, ios::in | ios::binary);

//Reading size of MinHeap unsigned char size;

inFile.read(reinterpret\_cast<char\*>(&size), 1); root = new Node();

{

char aCode;

unsigned char hCodeC[16]; inFile.read(&aCode, 1);

inFile.read(reinterpret\_cast<char\*>(hCodeC), 16);

//converting decimal characters into their binary equivalent to obtain code string hCodeStr = "";

for (int i = 0; i < 16; i++)

{

hCodeStr += decToBin(hCodeC[i]);

}

//Removing padding by ignoring first (127 - curr->code.length()) '0's and next '1' character int j = 0;

while (hCodeStr[j] == '0')

{

j++;

}

hCodeStr = hCodeStr.substr(j+1);

//Adding node with aCode data and hCodeStr string to the huffman tree buildTree(aCode, hCodeStr);

}

inFile.close();

}

void huffman::createTree()

{

//Creating Huffman Tree with the Min Heap created earlier Node \*left, \*right;

priority\_queue <Node\*, vector<Node\*>, Compare> tempPQ(minHeap);

left = tempPQ.top(); tempPQ.pop();

right = tempPQ.top(); tempPQ.pop();

root = new Node();

root->freq = left->freq + right->freq;

root->left = left; root->right = right; tempPQ.push(root);

}

}

void huffman::createCodes()

{

//Traversing the Huffman Tree and assigning specific codes to each character traverse(root, "");

}

void huffman::saveEncodedFile()

{

//Saving encoded (.huf) file inFile.open(inFileName, ios::in); outFile.open(outFileName, ios::out | ios::binary);

string in = ""; string s = "";

char id;

//Saving the meta data (huffman tree)

while (!tempPQ.empty())

{

Node\* curr = tempPQ.top(); in += curr->data;

//Saving 16 decimal values representing code of curr->data s.assign(127 - curr->code.length(), '0');

s += '1';

s += curr->code;

//Saving decimal values of every 8-bit binary code in += (char)binToDec(s.substr(0, 8));

for (int i = 0; i < 15; i++)

{

s = s.substr(8);

in += (char)binToDec(s.substr(0, 8));

}

tempPQ.pop();

}

s.clear();

//Saving codes of every charachter appearing in the input file inFile.get(id);

while (!inFile.eof())

{

s += arr[id]->code;

//Saving decimal values of every 8-bit binary code while (s.length() > 8)

{

in += (char)binToDec(s.substr(0, 8)); s = s.substr(8);

inFile.get(id);

}

//Finally if bits remaining are less than 8, append 0's int count = 8 - s.length();

if (s.length() < 8)

{

s.append(count, '0');

}

in += (char)binToDec(s);

//append count of appended 0's in += (char)count;

//write the in string to the output file outFile.write(in.c\_str(), in.size()); inFile.close();

outFile.close();

}

void huffman::saveDecodedFile()

{

inFile.open(inFileName, ios::in | ios::binary); outFile.open(outFileName, ios::out);

unsigned char size; inFile.read(reinterpret\_cast<char\*>(&size), 1);

//Reading count at the end of the file which is number of bits appended to make final value 8- bit

inFile.seekg(-1, ios::end);

char count0; inFile.read(&count0, 1);

//Ignoring the meta data (huffman tree) (1 + 17 \* size) and reading remaining file inFile.seekg(1 + 17 \* size, ios::beg);

vector<unsigned char> text; unsigned char textseg;

inFile.read(reinterpret\_cast<char\*>(&textseg), 1);

while (!inFile.eof())

{

text.push\_back(textseg); inFile.read(reinterpret\_cast<char\*>(&textseg), 1);

}

Node \*curr = root; string path;

for (int i = 0; i < text.size() - 1; i++)

{

//Converting decimal number to its equivalent 8-bit binary code path = decToBin(text[i]);

if (i == text.size() - 2)

{

path = path.substr(0, 8 - count0);

}

//Traversing huffman tree and appending resultant data to the file for (int j = 0; j < path.size(); j++)

{

if (path[j] == '0')

{

curr = curr->left;

else

{

curr = curr->right;

}

if (curr->left == NULL && curr->right == NULL)

{

outFile.put(curr->data); curr = root;

}

}

}

inFile.close(); outFile.close();

}

void huffman::compress()

{

createMinHeap();

createTree();

createCodes();

saveEncodedFile();

}

void huffman::decompress()

{

getTree();

* 1. **Encode Program –** This program is of cpp consist of command line argument and takes input from the user through command prompt. User runs the application and input the name of original file to be compressed and then the name of new compressed file with extension .huf.

## encode.cpp

#include <iostream>

#include "huffman.hpp"

using namespace std;

int main(int argc, char\* argv[])

{

if (argc != 3) {

cout << "Failed to detect Files";exit(1);

}

huffman f(argv[1], argv[2]);

f.compress();

cout << "Compressed successfully" << endl;

return 0;

}

* 1. **Decode Program -** This program is of cpp consist of command line argument and takes input from the user through command prompt. User runs the application and input the name of compressed file and then the name of new decoded file with extension .txt.

## decode.cpp

#include <iostream>

#include "huffman.hpp"

using namespace std;

int main(int argc, char\* argv[])

{

if (argc != 3) {

cout << "Failed to detect Files";exit(1);

}

huffman f(argv[1], argv[2]);

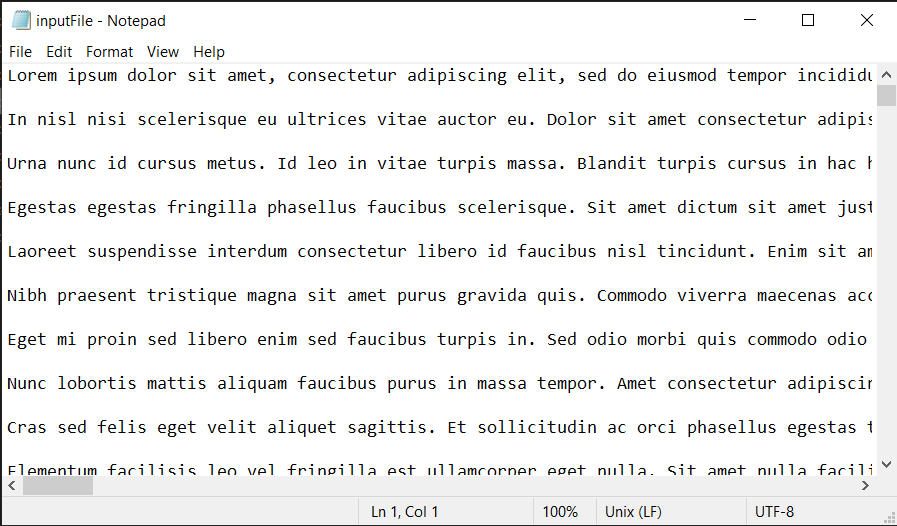
f.decompress(); cout << "Decompressed successfully" << endl;

return 0;

}

## Result –

This is our InputFile that we will be using for the compression. Figure 3.1 shows the original text file used for compression.

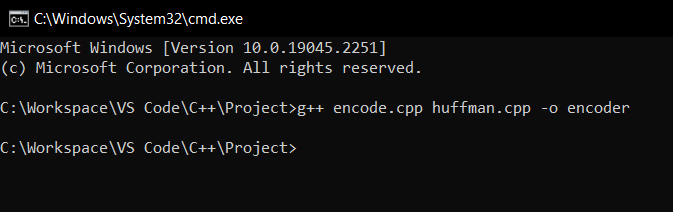


**Figure 3.1:** InputFile.txt (*Source*:[self])

First we have to encode the original text file that will create a new compressed file. Execute the command – “g++ encode.cpp huffman.cpp –o encoder.”

This command will create an application named ‘encoder’ that is created by the encode.cpp and Huffman.cpp program.

Figure 3.2 is the command line to be written in command prompt to create the Encoder application.

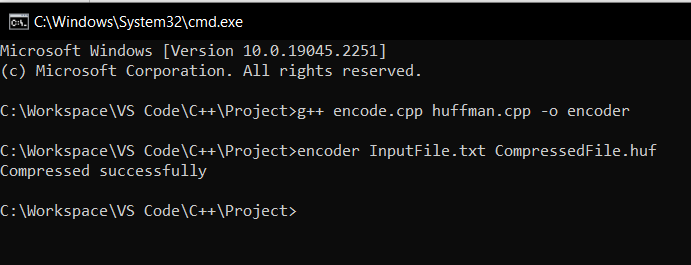


**Figure 3.2:** Creating Encoder Application (*Source*:[self])

Now simply use the encoder to create a compressed file.

Use command – “encoder InputFile.txt CompressedFile.huf ”

Figure 3.3 shows the command to compress the file in command prompt.

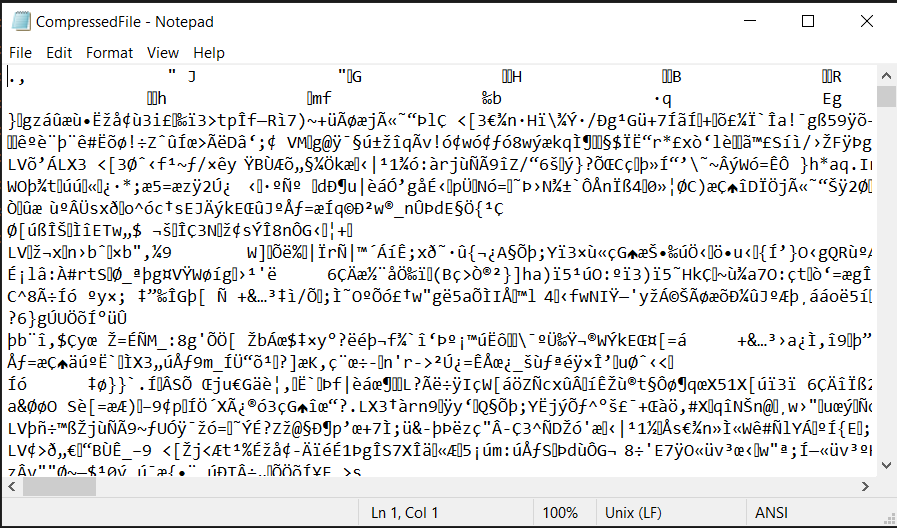


**Figure 3.3:** Compressing the file (*Source*:[self])

After successfully executing the encoding program with the help of application a compressed file will be generated with extension of ‘.huf’.

This is our compressed file where each character is converted to their respective binary codes assigned to them.

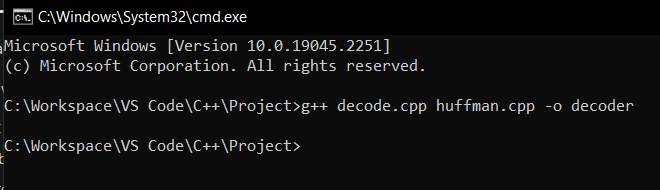
Figure 3.4 shows the compressed file that is created after compressing the original file.



**Figure 3.4:** CompressedFile.huf (*Source*:[self])

Now similarly for decoding of the compressed file first we will create an application that would be having the decode.cpp and Huffman.hpp program. Execute the command – “g++ decode.cpp huffman.cpp –o decoder.”

Figure 3.5 shows the command to be run in command prompt for creating the decoder application.

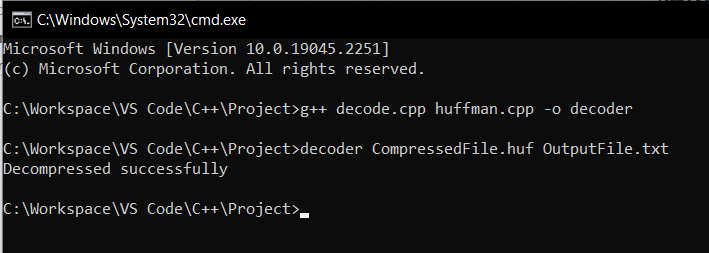


**Figure 3.5:** Creating Decoder Application (*Source*:[self])

Now simply use the decoder to decode the compressed file and get original file.

Use command – “decoder CompressedFile.huf OutputFile.txt”

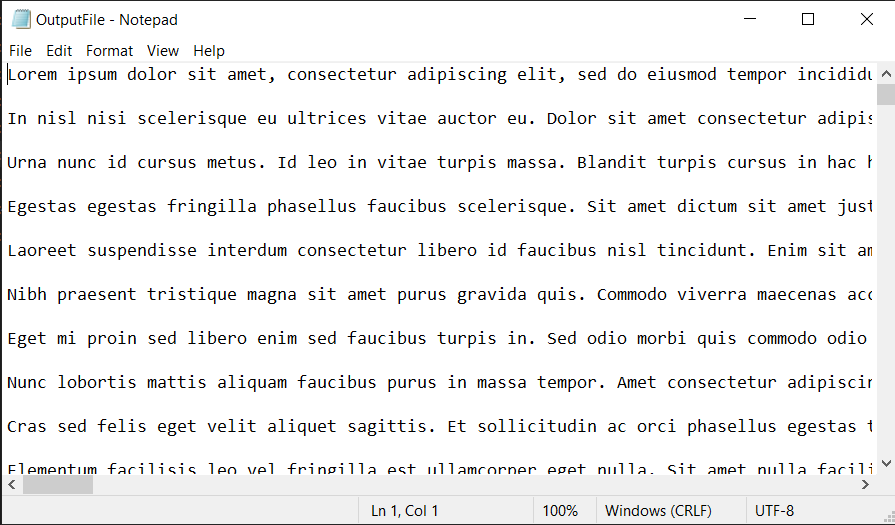
Figure 3.6 shows the command to be run in command prompt to decompressed the compressed .huf file,



**Figure 3.6:** Decompressing the file (*Source*:[self])

This will create a new text file which is same as our original file and thus our compressed file is decoded without any text loss or data loss.

Figure 3.7 shows the decompressed file that is same as our original file.



**Figure 3.7:** OutputFile.txt (*Source*:[self])

# Chapter 4

**Conclusion and Future Enhancements**

## Summary of Work done

At this point of time we have created a functional model that uses the Huffman algorithm. We create applications that contains the following:

* + - An application encoder that is used for the encoding the text file. It uses the C++ program ‘encode.cpp’ & ‘huffman.cpp’.
    - Use encoder application directly to the text file and generate the compressed file.
    - Another application decoder that is used for the decoding of the compressed file. It uses the C++ program ‘decode.cpp’ & ‘huffman.cpp’.
    - Use decoder application directly to decode the compressed file and get the original file without and loss of data.
    - Together both application can be used as a software to share the information confidentially and also save data during the transmission over the internet or web.

## Scope of Future Enhancements

The huﬀman coding, we have considered is simple binary Huﬀman coding but many variations of Huﬀman coding exist,

 n-ary Huﬀman coding: The n-ary Huﬀman algorithm uses the {0, 1, ..., n 1} alphabet to encode message and build an n-ary tree. This approach was considered by Huﬀman in his original paper. The same algorithm applies as for binary (n equals2) codes, except that the n least probable symbols are taken together, instead of just the 2 least probable. Note that for n greater than 2, not all sets of source words can properly form an n-ary tree for Huﬀman coding. In this case, additional 0-probability place holders must beadded. If the number of source words is congruent to 1 modulo n-1, then theset of source words will form a proper Huﬀman tree.

 Adaptive Huﬀman coding: A variation called adaptive Huﬀman coding calculates the probabilities dynamically based on recent actual frequencies in the source string. This is somewhat related to the LZ family of algorithms.

 Length-limited Huﬀman coding: Length-limited Huﬀman coding is a variant where the goal is still to achieve a minimum weighted path length, but there is an additional restriction that the length of each code word must be less than a given constant. The package-merge algorithm solves this problem with a simple greedy approach very similar to that used by Huﬀman’s algorithm. Its time complexity is O (nL), where L is the maximum length of a code word. No algorithm is known to solve this problem in linear or linear logarithmic time, unlike the pre-sorted and unsorted conventional Huﬀman problems, respectively.

* Huﬀman template algorithm:

Most often, the weights used in implementations of Huﬀman coding represent numeric probabilities, but the algorithm given above does not require this; it requires only a way to order weights and to add them. The Huﬀman template algorithm enables one to use any kind of weights (costs, frequencies, etc.).

# REFERENCES

[1]. A. Malik, N. Goyat and V. Saroha, “Greedy Algorithm: Huffman Algorithm,” International Journal of Advanced Research in Computer Science and Software Engineering, vol. 3, no.7, pp. 296-303,2013.

[2]. [A.](http://www.jetbrains.com/pycharm/) S. Sidhu and M. Garg, “Research Paper on Text Data Compression Algorithm using Hybrid Approach,” IJCSMC, vol. 3, no. 12, pp. 1-10, 2014

[3]. H. Al-Bahadili and S. M. Hussain, “A Bit-level Text Compression Scheme Based on the ACW Algorithm,” International Journal of Automation and Computing, pp. 123-131, 2010.

[4]. H. Altarawneh and M. Altarawneh, "Data Compression Techniques on Text Files: A Comparison Study” International Journal of Computer Applications, Vol 26– No.5, and July 2011 [5]. I. Akamn, H. Bayindir, S. Ozleme, Z. Akin and a. S. Misra, “Lossless Text Compression Technique Using Syllable Based Morphology,” International Arab Journal of Information Technology, vol. 8, no. 1, pp. 66-74, 2011.

[6] M. Schindler, “Practical Huffman coding,” 1998 [Online].

[7]. R.S. Brar and B. Singh, “A survey on different compression techniques and bit reduction Algorithm for compression of text data” International Journal of Advanced Research In Computer Science and Software Engineering (IJARCSSE) Volume 3, Issue 3, March 2013

[8].R. Kaur and M. Goyal, “An Algorithm for Lossless Text Data Compression” International Journal of Engineering Research & Technology (IJERT), Vol. 2 Issue 7, July - 2013

[9]. S. Porwal, Y. Chaudhary, J. Joshi and M. Jain, “Data Compression Methodologies for Lossless Data and Comparison between Algorithms” International Journal of Engineering Science and Innovative Technology (IJESIT) Volume 2, Issue 2, March 2013

[10]. S. Shanmugasundaram and R. Lourdusamy, “A Comparative Study of Text Compression Algorithms” International Journal of Wisdom Based Computing, Vol.1 (3), Dec 2011

[11]. S. Kapoor and A. Chopra, "A Review of Lempel Ziv Compression Techniques" IJCST Vol.4, Issue 2, April-June 2013

[12]. S.R. Kodituwakku and U. S. Amarasinghe, “Comparison of Lossless Data Compression Algorithms for Text Data “Indian Journal of Computer Science & Engineering Vol 1 No 4

[13]. U. Khurana and A. Koul, “Text Compression and Superfast Searching” Thapar Institute Of Engineering and Technology, Patiala, Punjab, India-147004