**A Project Report**

**On**

**File Compression and Encryption**

**For**

**Algorithm and Problem Solving Lab**

**(15B17CI471)**

Submitted by:

**Rahi Agarwal (9921103145.)**

**Priyansh Jain (9921103164)**

**Madhav Bansal (9921103097)**

**Naman Jhanwar (9921103191)**

Submitted to:

**Dr. Neeraj Jain**

**Dr. Laxmi Chaudhary**

****

**Department of CSE/IT**

**Jaypee Institute of Information Technology University, Noida**

**May, 2023**

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| 1. **Problem statement** | **3** |
| 1. **Introduction** | **4-5** |
| 1. **Detailed description of the project** | **6-7** |
| 1. **Implementation** | **8-21** |
| 1. **Results with snapshot of output** | **22-23** |
| 1. **Conclusion** | **24** |
| 1. **References** | **24** |

**PROBLEM STATEMENT**

Write a program in C++ to implement the following things :-

* Take input such as text file.
* Compress and decompress the file using Huffman Coding Algorithm
* Encrypt and Decrypt The file using Caeser Cypher

The problem that this report aims to address is to evaluate the effectiveness and efficiency of the Huffman coding algorithm and its implementation in constructing the Huffman tree for text and data compression and decompression.

**INTRODUCTION**

**MOTIVATION**

Huffman coding is a variable-length prefix coding scheme, which means that the prefix codes assigned to each character in the input file are uniquely decodable. This property ensures that the decoded output obtained from the compressed file is identical to the original input file.

The compression ratio achieved using Huffman coding depends on the frequency distribution of the characters in the input file. If the input file contains a large number of frequently occurring characters, the compression ratio will be higher.

The Caesar Cipher algorithm is a type of substitution cipher, which means that each character in the input file is replaced by another character according to a fixed rule. The algorithm is named after Julius Caesar, who used a simple shift cipher to encrypt his messages.

**OBJECTIVE AND SCOPE OF THE PROJECT:**

The project aims to implement Huffman coding for compressing and decompressing a text file and Caesar Cipher algorithm for encryption and decryption of the compressed file.

The goal of our project is to develop a program that can compress and decompress files using the Huffman coding algorithm. The program should take an input file of any type and compress it using variable-length codes assigned to each symbol in the input file. The compressed data should be written to a new output file.

To implement the Huffman coding algorithm, the program should first calculate the frequency of occurrence for each symbol in the input file. It should then use this frequency information to construct a binary tree that represents the Huffman code for the data. This tree should be stored in a compressed form along with the compressed data.

The program should also be able to decompress the compressed data back into its original form using the Huffman tree. The decompressed data should match the original input file exactly.

**Salient Features of the Project**

* User-friendly menu-driven options.
* Data Validation for all the inputs to be taken from the user to avoid ambiguous/wrong/invalid data entry.
* Easy to read and understand.
* Explanation provided wherever necessary.
* Thoroughly researched data which can be changed with time.

**CONTRIBUTION**

Huffman coding is a widely used compression algorithm that has made significant contributions in the field of data compression. Here are some of its contributions:

Efficient Data Compression: Huffman coding is an effective algorithm for compressing data and reducing file sizes. It assigns variable-length codes to each symbol in the input file based on its frequency of occurrence. This allows the data to be represented using fewer bits, resulting in a smaller file size.

Faster Data Transmission: Since the compressed data is smaller, it can be transmitted over networks or the internet at a faster rate, reducing the time it takes to transmit large amounts of data.

Reduced Storage Requirements: Huffman coding reduces the amount of storage space required to store data, allowing for more data to be stored on a given storage device.

Versatility: Huffman coding can be used for various types of data, including text, images, audio, and video. It is a universal compression algorithm that can be applied to a wide range of data types and formats.

Standardization: Huffman coding has been widely adopted as a standard for data compression in many industries, including computer hardware, software, and telecommunications. This has facilitated interoperability and compatibility across different systems and platforms.

In summary, Huffman coding has made significant contributions to data compression, enabling faster data transmission, reducing storage requirements, and facilitating interoperability and compatibility across different systems and platforms.

**DESCRIPTION**

We will be using various concepts we have learned throughout this semester and knowledge acquired in previous semesters to produce an easy-to-use working model of our idea.

Various concepts used in the project include and are not limited to:

* Basic C++/C concepts
* Arrays
* Vectors
* Linked List
* Stack and queue

Various methods for data retrieval and management in a more fast and more efficient way.

**Algorithms Used:**

**Huffman Coding:**

Huffman coding is a lossless data compression technique that assigns variable-length codes to characters based on their frequency of occurrence in the input file. The most frequently occurring characters are assigned the shortest codes, while the least frequently occurring characters are assigned longer codes. The compressed file is generated by replacing the characters in the input file with their respective codes.

The steps involved in implementing Huffman coding are as follows:

* **Frequency table generation:** A frequency table is generated by scanning the input file and counting the frequency of occurrence of each character in the file.
* **Construction of Huffman tree:** A binary tree is constructed using the frequency table generated in step 1. The two characters with the lowest frequencies are combined to form a new node, which is then inserted into the tree with a frequency equal to the sum of the frequencies of the two characters. This process is repeated until all the characters are included in the tree.
* **Generation of Huffman codes:** Huffman codes are generated by traversing the Huffman tree from the root node to each leaf node. A '0' is appended to the code if the left child of the node is visited, and a '1' is appended if the right child of the node is visited.
* **Encoding of the input file:** The input file is encoded using the Huffman codes generated in step 3. Each character in the input file is replaced by its corresponding Huffman code.
* **Writing the compressed file:** The compressed file is generated by writing the frequency table and the encoded input file to a new file.
* Decompression of the compressed file involves the reverse process of Huffman coding. The frequency table is extracted from the compressed file, and the Huffman tree is reconstructed. The encoded input file is then decoded using the Huffman tree to generate the original input file.

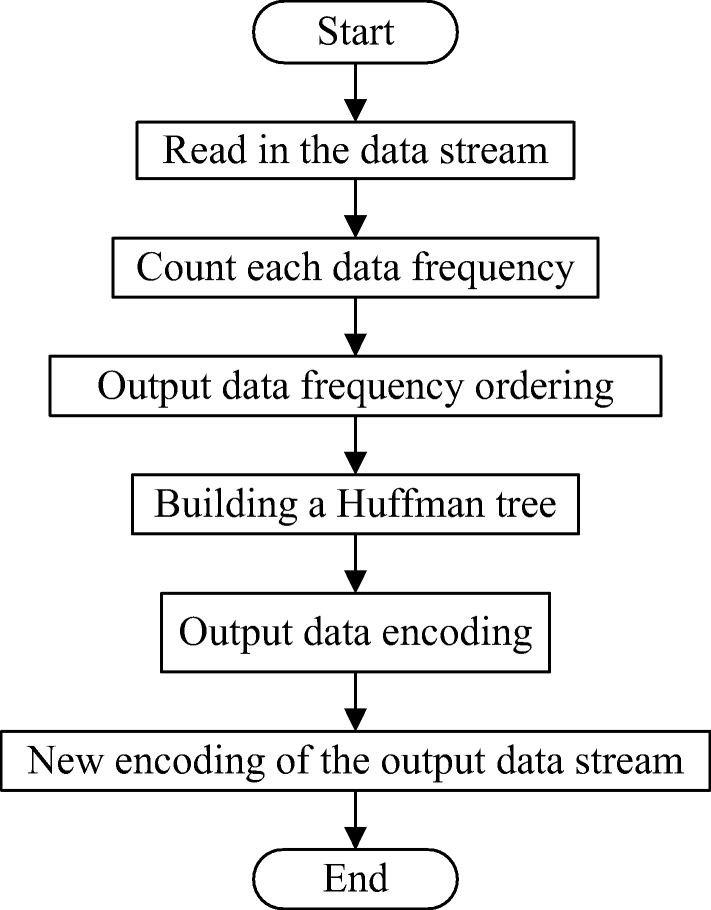
**Caesar Cipher:**

Caesar Cipher is a simple encryption algorithm that involves shifting each character in the input file by a fixed number of positions in the alphabet. For example, if the shift value is 3, then the letter 'A' is replaced by 'D', 'B' by 'E', and so on. The shift value is also known as the key for encryption.

The steps involved in implementing Caesar Cipher are as follows:

* **Reading the compressed file:** The compressed file generated by Huffman coding is read into the program.
* **Decryption using Caesar Cipher:** The compressed file is decrypted using the Caesar Cipher algorithm. Each character in the compressed file is shifted back by the same number of positions in the alphabet as the key used for encryption.
* **Writing the decompressed file:** The decompressed file is generated by writing the decrypted data to a new file.

**FLOW CHART AND BLOCK DIAGRAM:**

****

**Code:**

#include<iostream>

#include <string>

#include <vector>

#include <queue>

#include <fstream>

#include<cstring>

#include<sstream>

#include <typeinfo>

#include<windows.h>

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();

//Decompressing input file

void decompress();

};

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');

}

int huffman::binToDec(string inStr)

{

int res = 0;

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::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;

}

void huffman::createMinHeap()

{

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();

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)

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 character 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::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::compress()

{

createMinHeap();

createTree();

createCodes();

saveEncodedFile();

}

void huffman::decompress()

{

getTree();

saveDecodedFile();

}

bool fileExists(const std::string& fileName) {

std::ifstream file(fileName);

return file.good();

}

void encryptFile(string input,string encryptedFile)

{

//encryption of text data

ifstream File;

//clearing encryption.txt before editing

File.open(encryptedFile.c\_str(), std::ifstream::out | std::ifstream::trunc );

if (!File.is\_open() || File.fail())

{

File.close();

printf("\nError : failed to erase file content !");

}

File.close();

//reading plain text from input.txt

fstream newfile;

newfile.open(input,ios::in); //open a file to perform read operation using file object

if (newfile.is\_open()){ //checking whether the file is open

cout << "Reading plain text from " << input << " .........\n";

Sleep(1000);

string tp;

int key;

cout << "Enter key to encrypt text : ";

cin >> key;

cout << "Now encrypting ....\n";

Sleep(1000);

cout << "writing encrypted data in " << encryptedFile << " ..\n";

Sleep(1000);

cout<<endl;

while(getline(newfile, tp))

{

//read data from file object and put it into string.

string outputtext="";

int messlength=tp.length();

int i=0;

while(tp[i]!='\0')

{

outputtext += tp[i]+key;

i++;

}

//storing our encrypted data in encryption.aes

ofstream fout; // Create Object of Ofstream

ifstream fin;

fin.open(encryptedFile);

fout.open (encryptedFile,ios::app); // Append mode

if(fin.is\_open())

fout<< outputtext <<"\n"; // Writing data to file

fin.close();

fout.close();

}

cout << "Caesar Cipher encryption is done successfully \n";

cout << "Data has been appended to file " << encryptedFile << endl;

newfile.close(); //close the file object.

}

}

void decryptFile(string output,string encryptedFile)

{

cout << "Reading encrypted data from " << encryptedFile << " .........\n";

Sleep(1000);

string tp;

int key;

cout << "Enter decryption key : ";

cin >> key;

cout << "Now Decrypting ....\n";

Sleep(1000);

cout << "Writing decrypted data in " << output << " ..\n";

Sleep(1000);

//clearing output file

ifstream File;

File.open(output.c\_str(), std::ifstream::out | std::ifstream::trunc );

if (!File.is\_open() || File.fail())

{

File.close();

cout << "\nError : failed to erase file content !";

}

File.close();

//storing output text in output text file

string myText;

ifstream MyReadFile;

MyReadFile.open(encryptedFile, ios::in | ios::binary);

if(MyReadFile.is\_open())

{

while( getline (MyReadFile, myText))

{

string outputtext="";

int messlength=myText.length();

int i=0;

while ( myText[i]!='\0' )

{

outputtext += myText[i]-key;

i++;

}

ofstream fout; // Create Object of Ofstream

ifstream fin;

fin.open(output);

fout.open (output,ios::app); // Append mode

if(fin.is\_open())

fout << outputtext.substr(0,messlength-1) << endl; // Writing data to file

fin.close();

fout.close();

}

cout << endl;

cout << "Plain text has been appended to file " << output << endl;

//close the file object.

MyReadFile.close();

}

}

// This function is responsible for compressing a file using Huffman coding.

void compress()

{

string input,compressedFile;

// Get the names of the input file and the compressed file from the user.

cout << "Enter the name of the input file (without including extension ('.txt')): ";

cin >> input;

cout << "Enter the name of the compressed file (without including extension ('.huf')): ";

cin >> compressedFile;

if (fileExists(input + ".txt"))

{

// Create an instance of the Huffman class and call its compress method.

huffman f( input + ".txt", compressedFile + ".huf");

f.compress();

// Inform the user that the compression is successful.

cout << "Compressed successfully" << endl;

}

else

cout << input << ".txt file does not exist!!!" << endl;

}

// This function is responsible for decompressing a file using Huffman coding.

void decompress()

{

string output,compressedFile;

// Get the names of the compressed file and the output file from the user.

cout << "Enter the name of the compressed file (without including extension ('.huf')): ";

cin >> compressedFile;

cout << "Enter the name of the output file (without including extension ('.txt')): ";

cin >> output;

if (fileExists(compressedFile + ".huf"))

{

// Create an instance of the Huffman class and call its decompress method.

huffman f( compressedFile + ".huf", output + ".txt");

f.decompress();

// Inform the user that the decompression is successful.

cout << "Decompressed successfully" << endl;

}

else

cout << compressedFile << ".huf file does not exist!!!" << endl;

}

// This function is responsible for encrypting a file using a custom encryption algorithm.

void encrypt()

{

string input,encryptedFile;

// Get the names of the input file and the encrypted file from the user.

cout << "Enter the name of the input file (without including extension ('.txt')): ";

cin >> input;

cout << "Enter the name of the encrypted file (without including extension ('.txt')): ";

cin >> encryptedFile;

// Call the encryptFile function to encrypt the input file.

if (fileExists(input + ".txt"))

encryptFile(input + ".txt", encryptedFile + ".txt");

else

cout << input << ".txt file does not exist!!!" << endl;

}

// This function is responsible for decrypting a file using the same custom encryption algorithm used in encrypt().

void decrypt()

{

string output,encryptedFile;

// Get the names of the encrypted file and the output file from the user.

cout << "Enter the name of the encrypted file (without including extension ('.txt')): ";

cin >> encryptedFile;

cout << "Enter the name of the output file (without including extension ('.txt')): ";

cin >> output;

// Call the decryptFile function to decrypt the encrypted file.

if (fileExists(encryptedFile + ".txt"))

decryptFile(output + ".txt", encryptedFile + ".txt");

else

cout << encryptedFile << ".txt file does not exist!!!" << endl;

}

int main()

{

int ch;

while (1)

{

int ch;

// Display the menu to the user.

cout << "1. Compress a file" << endl;

cout << "2. Decompress a file" << endl;

cout << "3. Encrypt a file" << endl;

cout << "4. Decrypt a file" << endl;

cout << "5. Quit" << endl;

cout << "Enter your choice : ";

cin >> ch;

// Call the appropriate function based on the user's choice.

switch (ch)

{

case 1: compress();

break;

case 2: decompress();

break;

case 3: encrypt();

break;

case 4: decrypt();

break;

case 5: cout << "Thanks for joining!!!!" << endl;

return 0;

default:cout << "Invalid input!!!" << endl;

cout << "Please input a valid choice!!" << endl;

break;

}

system("pause");

system("CLS");

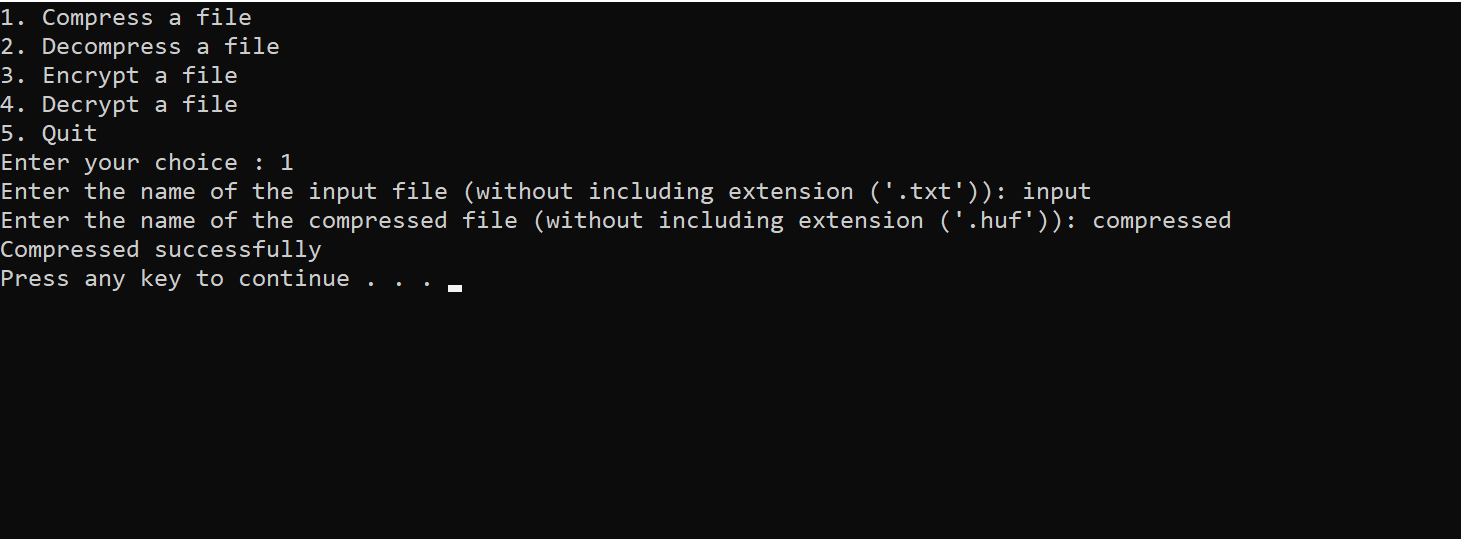
}

return 0;

}

**Output Screen:**

**Compressing a File**

****

**Decompressing a File**

**Text

Description automatically generated**

**Encrypting a File**

**A picture containing text

Description automatically generated**

**Decrypting a File**

**Text

Description automatically generated**

**Conclusion**

● Huffman Coding is generally useful to compress the data in which there are frequently

occurring characters.

● We can observe that the most frequent character gets the smallest code and the least

frequent character gets the largest code.

● The variable-length code i.e. a different number of bits for each character/symbol is

obtained by using the Huffman Code compression technique which is more advantageous

than fixed-length coding because it reduces the memory space and decreases the time

required for transmitting the data.

In conclusion, the Caesar cipher and Huffman coding are both important concepts in the field of cryptography and data compression.

The Caesar cipher is a simple encryption technique that was used in ancient times to protect messages. It works by shifting each letter in the message by a fixed number of positions in the alphabet. While it was effective at the time, it is no longer considered a secure encryption method as it can easily be cracked using modern techniques.

On the other hand, Huffman coding is an effective compression algorithm that assigns variable-length codes to each symbol in the input file based on its frequency of occurrence. This allows the data to be represented using fewer bits, resulting in a smaller file size. Huffman coding has made significant contributions to data compression, enabling faster data transmission, reducing storage requirements, and facilitating interoperability and compatibility across different systems and platforms.

While the Caesar cipher and Huffman coding are different in their purpose and functionality, they both demonstrate the importance of cryptography and data compression in our digital age. As technology advances, it is crucial to continue developing and improving these techniques to protect sensitive information and enhance the efficiency of data transmission and storage.

**References**

https://en.wikipedia.org/wiki/Huffman\_coding

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3

https://www.programiz.com/dsa/huffman-coding

https://www.researchgate.net/publication/304395425\_Huffman\_coding

<https://www.scaler.com/topics/huffman-coding/>

https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/