Data Compression

Muhammad Sallar Bin Aamir
&
Huzaifa Ilyas
(01-134211-066) & (01-134211-061)

26/06/2022

—

Data Structures and Algorithms
(Lab)

—

Mam Rabail Zahid

Introduction

What is Data Compression?

Data compression is a reduction in the number of bits needed to represent data. Compressing data can save storage capacity, speed up file transfer, and decrease costs for storage hardware and network bandwidth.
It enables reducing the storage size of one or more data instances or elements. Data compression is also known as source coding or bit-rate reduction.

Problem Statement:

This project's purpose is to build a data compression method. That is, we want to convey the same information in a smaller amount of space given data.

1. Read a Text File Build simple Tree based Huffman coding scheme and show the results.
2. The second task in this project is to use predefined priority queues to build an optimal Huffman tree. Your priority queue will maintain the current set of trees ordered by their frequencies. One challenge is to efficiently traverse the optimal Huffman tree to generate the code to be printed out.
3. At the end take a sample file and compress it.

Objectives:

- Read a Text File
- Create a Huffman Tree using STL Priority Queue.
- Compress the File.

Tools Used:

- Visual Studio 2022
- C++ 17
- GitHub (Version Control System)
- Text File
- Microsoft Word

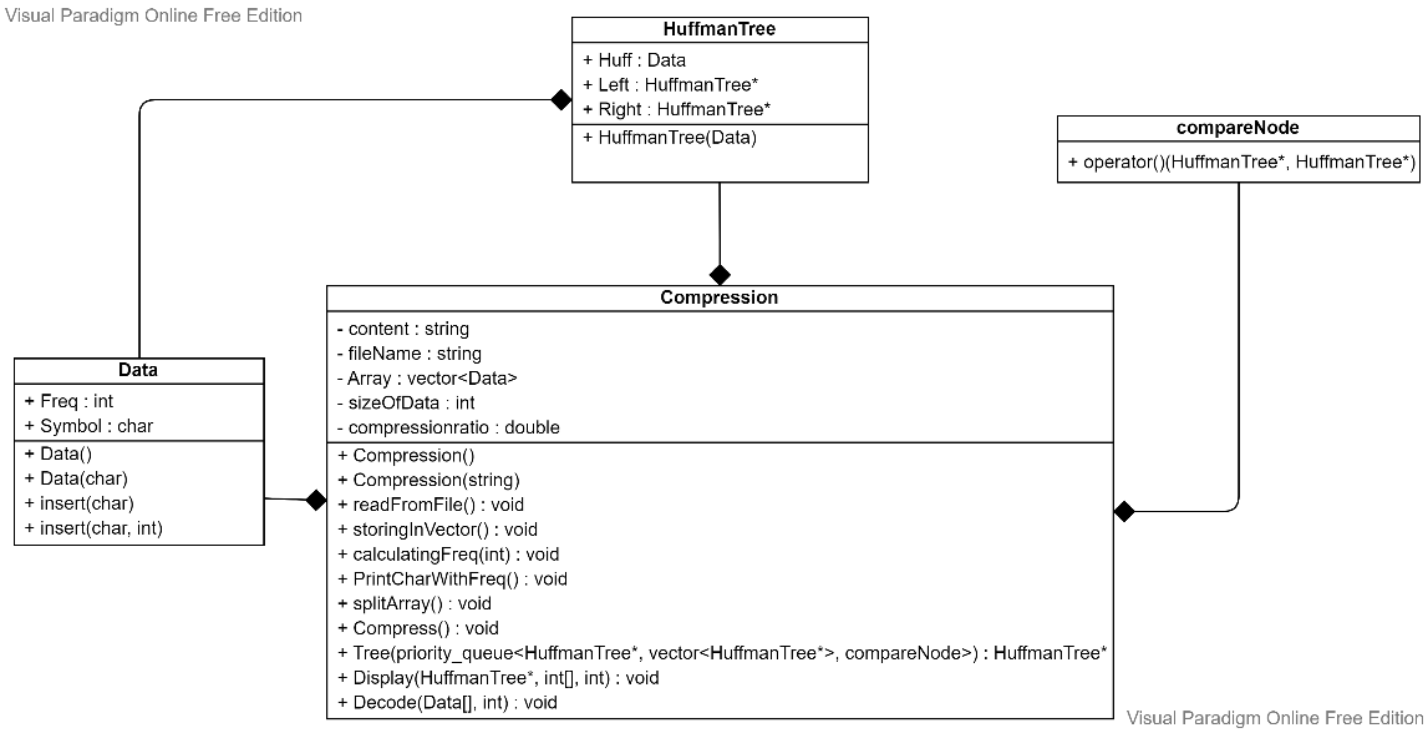
GitHub Repository Link:

"**https://github.com/sallar-ba/Data-Compression**"

Working:

- Unified Model Language Diagram (UML)
- Data Structures Used
- Algorithm
- Classes
- Source Code

Unified Model Language Diagram (UML Diagram):



*Made Using: online.visual-paradigm.com

Data Structures Used:

- **Priority Queue**
  We Used a Standard Template Library (STL) of Queue to implement Priority Queue.

- **Recursion**
  Minor Recursion is Used in the Code.

- **Huffman Algorithm**
  Huffman Algorithm is the main Algorithm used in this project (Data Compression).

- **Searching**
  Searching Algorithm is used in the Project.

Algorithm:

Algorithm is defined in plain simple English below:

- Reading a Text File.
- Storing the Text in the File into a String.
- Converting the String into a character and its frequency and storing into vector.
- Searching the Duplicates and Deleting the Duplicates while Incrementing the Frequency of Found Character.
- Creating a Huffman Tree.
- Calculating the Compression Ratio.
- Displaying Output.

Classes:

4 Main Classes Used in the Project.

1. Data Class
2. HuffmanTree Class
3. compareNode Class
4. Compression Class

Data Class:

Class consists of public data members and public member functions. Class is made to store the character and their respective frequency.

HuffmanTree Class:

This Class Consist of **Data** i.e Character and Frequency, With the Left and Right Nodes. These are Essential for making a Tree.

compareNode Class:

This Class Contains Only One Function and is used to Compare Frequencies of Two Nodes.

Compression Class:

This is the Main Class of the Project, This Contains the Most Data Members and Member Functions.
Most of The Work is Being Done in This Class. From Reading from a file to creating

Huffman Tree to Finding the Compression Ratio. This Class Does it All.

Source Code:

The Source Code is Divided into Different .h and .cpp files. GitHub Link is Provided for Better understanding.

Code Given Below:

Data.h

```
#pragma once
// -+-+-+-+-+--+-+-+-+-+--+-+-+-+-+--+-+-+-+-+-
class Data // data class
{
public:
	//Public Data Members
	int Freq; // For Storing Frequency
	char Symbol; // For Storing Symbol or Character

	// ----- Constructors ----
	Data();
	Data(char);

	////////////// PUBLIC MEMBER FUNCTIONS /////////
	// Functions
	void insert(char);
	void insert(char, int); //Overloaded Function
};
// -+-+-+-+-+--+-+-+-+-+--+-+-+-+-+--+-+-+-+-+-
```

Data.cpp

```
#include "Data.h"
//---------------- Constructors --------------------
Data::Data()
{
	//Assigning
	this->Freq = 0;
	this->Symbol = ' ';
}
Data::Data(char Symbol)
{
	//Assigning
	this->Freq = 1; // Assigned 1 Because Every character Will get Value 1
	this->Symbol = Symbol;
}
//-------------------------------------------------
//---------------- Insert Functions --------------------
void Data::insert(char Symbol)
{
```

```
	//Assigning
	this->Freq = 1;
	this->Symbol = Symbol;
}
//overloaded Function
void Data::insert(char Symbol, int Freq)
{
	//Assigning
	this->Freq = Freq;
	this->Symbol = Symbol;
}
//-----------------------------------------------
```

compareNode.h

```
#pragma once
#include"HuffmanTree.h" // including HuffmanTree Class
class compareNode // Class
{
public:
	//Public Member Function
	bool operator()(HuffmanTree*, HuffmanTree*); //Operator overloading
};
//-------------------------------------
```

compareNode.cpp

```
#include "compareNode.h"//Including Header
//+-+-+-+--+-+-+--+-+-+--+-+-+--+-+-+--+-+-+--+-+-+--+-+-
// overload operator to compare two nodes frequency
bool compareNode::operator()(HuffmanTree* First, HuffmanTree* Second)
{
	/*
	Checking First Frequency to be Greater than the Second
	and returing True else False.
	*/
	return First->Huff.Freq > Second->Huff.Freq;
}
//-+-+-+--+-+-+--+-+-+--+-+-+--+-+-+--+-+-+--+-+-+--+-+-+-
```

HuffmanTree.h

```
#pragma once
#include<iostream> // inclusing I/O Lib.
#include"Data.h" // including Data Class
using namespace std;
//------------------------------
class HuffmanTree //class
{
public:
	/*char Symbol;
	int Frequency;*/
	Data Huff; //Object
	//-----------------------
```

```
	HuffmanTree* Left;
	HuffmanTree* Right;
	//-----------------------
	//Constructor
	HuffmanTree(Data); //Parameterized
};
//-------------------------------------
```

HuffmanTree.cpp

```
#include "HuffmanTree.h" // Including Header
//-------------- Constructor ---------------------
HuffmanTree::HuffmanTree(Data Huff)
{
	//Assigning
	this->Huff = Huff;
	this->Left = this->Right = NULL;
}
//-------------------------------------------------
```

Compression.h

```
#pragma once
#include<iostream> // Including Input/Output Lib.
#include<string> // Including String Lib.
#include<vector> // Including Vector Lib.
#include<fstream> // File Lib.
#include<iomanip> // For Input/Ouput Manipulation
// -+-+-+-+-+--+-+-+-+-+--+-+-+-+-+--+-+-+-+-+-
//Including Self-made Classes
#include"Data.h"
#include"compareNode.h"
// -+-+-+-+-+--+-+-+-+-+--+-+-+-+-+--+-+-+-+-+-
#include<queue> // Including Queue Class For Priority Queue (STL)
#define Max_Size 100
using namespace std;
//-------------------------------------
class Compression // Compression Class
{
	//Private Data Members
private:
	string content;// the string which will read from the file
	string fileName; // to store File Name
	vector<Data> Array;// Modifiable Array to Store Data
	int sizeOfData; // To Store Len. of Data
	float compressionRatio;//  to store compression ratio
public:
	//Public Member Function Prototypes
	Compression();// default constructor
	Compression(string); // Parameterized Constructor

	// Reading text from the file
	void readFromFile();

	/*
	   Storing Data in Vector (Modifiable Array)
```

```
	*/
	void storingInVector();

	/*
	   Function to the Frequency of Each Character Present
	   in File, Which is Copied in content Data Member by
	 "readFromFile()" Function.
	*/
	void calculatingFreq(int);
	//Printing Function
	void PrintCharWithFreq();

	//Function to Splot Arrays
	void createArray();

	//Single Function to Compress all the Data
	void Compress();

	//Function To Make Huffman Tree
	HuffmanTree* Tree(priority_queue<HuffmanTree*, vector<HuffmanTree*>, compareNode>);

	//Display Function
	void Display(HuffmanTree*, int[], int);

	//Decoding into Huffman
	void Decode(Data[], int);
};
//--------------------------------------
```

Compression.cpp

```
#include "Compression.h"
//---------------------------------------- Constructors ------------------------------------
-------
//Default Constructor
Compression::Compression()
{
	// Giving a Default File
	fileName = "Huffman.txt";
}
// Parameterized Constructor
Compression::Compression(string fileName)
{
	//Assigning
	this->fileName = fileName;
}
//-------------------------------------------------------------------------------------------
---------
void Compression::readFromFile()
{
	// Creating an Object
	ifstream inFile(fileName);

	// If File is Not Present
	if (!inFile)
	{
		// Printing Error
		cout << "File Does Not Exist..." << endl;
		// Exiting
```

```
		exit(1);
	}
	else
	{
		// Creating an Object
		ifstream readFile(fileName, ios::in);
		// Run Till End of File
		while (!readFile.eof())
		{
			// Copy Data From readFile to Content String
			while (getline(readFile, content))
			{
				// getting the whole text file size
				sizeOfData = content.length();
			}
		}
		//Closing the File
		readFile.close();
	}

}
//-------------------------------------------------------------------------------------------
----------
void Compression::storingInVector()
{
	ifstream inFile(fileName); // Creating an Object
	Data Obj; // Creating Object of Data

	// Declaring Variables
	char character = ' ';
	int counter = 0; // Variable for Counting

	if (!inFile) // If File is Not Present
	{
		// Printing Error
		cout << "File Does Not Exist..." << endl;
		// Exiting
		exit(1);
	}
	else
	{
			/* the good function is to check whether
			   the file is good enough
			   to open or not.
			*/
			while (inFile.good())
			{
				// reading character by character
				inFile.get(character); // Getting Character From File
				// Did This Because It Was Printing an Extra Character
				if (counter != sizeOfData)
				{
					Obj.insert(character);//inserting Character into the Data Obj
					Array.push_back(Obj); // Pushing Data into vectors
					counter++; // Incrementing Counter
				}
			}
		inFile.close();
	}
}
//-----------------------------------------------------------
```

```
void Compression::calculatingFreq(int i = 2)
{
	//Made Recursive
	if (i == 0)
	{
		//Ending The Function
		return;
	}
	else
	{
		//Nesting For Loop
		for (int i = 0; i < Array.size() - 1; i++)
		{
			//For-Loop
			for (int j = i + 1; j < Array.size(); j++)
			{
				//Finding Duplicate Symbol
				if (Array[i].Symbol == Array[j].Symbol)
				{
					/*
					* Swapping the Found Symbol with the Last Element
					*/
					char Temp = Array[j].Symbol;
					Array[j].Symbol = Array[Array.size() - 1].Symbol;
					Array[Array.size() - 1].Symbol = Temp;
					//Incrementing Freq
					Array[i].Freq = Array[i].Freq + 1;
					// Removing Last
					Array.pop_back();
				}
			}
		}
		calculatingFreq(i - 1); //Calling The Function
	}
}
//-----------------------------------------------------------
void Compression::PrintCharWithFreq()
{
	cout << endl;
	//Telling That Printing Symbols
	cout << " Symbols:\tFrequency:\n" << endl;
	//For-Loop
	for (int i = 0; i < Array.size(); i++)
	{
		// Printing Symbols & Frequency
		cout << setw(4) << setfill(' ') << "\'" << Array[i].Symbol << "\'\t\t" << setw(5)
<< setfill(' ') << Array[i].Freq << endl;
	}
	cout << endl << endl;
}
//-----------------------------------------------------------
void Compression::Compress()
{
	//Function Call to Read Content From File
	readFromFile();
	//Function Call to Store Content Into Vector (Array)
	storingInVector();
	//Function Call To Calculate Frequency
	calculatingFreq();
	//Function Call To Print Characters With Frequency
	PrintCharWithFreq();
```

```
	//Splitting Array
	createArray();

}
//-----------------------------------------------------------
void Compression::createArray()
{
	//Creating an Array of Data Object of Size of Vector
	Data* Huff = new Data[Array.size()];
	//For-Loop
	for (int i = 0; i < Array.size(); i++)
	{
		//Storing Data i.e Symbol and Frequency in Data Array
		Huff[i].insert(Array[i].Symbol, Array[i].Freq);
	}
	//Calling the Decode Function
	Decode(Huff, Array.size());
}
//-----------------------------------------------------
HuffmanTree* Compression::Tree(priority_queue<HuffmanTree*, vector<HuffmanTree*>, compareNode>
PQue)
{
	/* This function makes tree untill the pQue size becomes 1
	which will be the root node of the whole tree */
	while (PQue.size() != 1)// function is used to get size of the priority queue
	{
		// saving the top node in left
		HuffmanTree* Left = PQue.top(); //function is used to reference the top element of
the priority queue

		// Poping the top node from Pque
		PQue.pop();

		//saving the second top node in right node
		HuffmanTree* Right = PQue.top(); //function is used to reference the top element
of the priority queue

		// poping that from Pque
		PQue.pop();
		Data Check;
		Check.insert('@', Left->Huff.Freq + Right->Huff.Freq);// making new data node
		HuffmanTree* newNode = new HuffmanTree(Check);
		// initiailizing the new node left and right child with the poped nodes
		newNode->Left = Left;
		newNode->Right = Right;

		// Pushing the new node into the pQue
		PQue.push(newNode);
	}

	/* at the end the root node will be    returned
	from where we will decode the tree in binary codes */
	return PQue.top();

}
//-------------------------------------------------------------------
void Compression::Display(HuffmanTree* root, int Arr[], int top)
{
	/* If the node have the left element till the symbol initialize the Array[top]
		0 (because according too the huffman algorithm ) */
	if (root->Left) //assign 0 to the left child path
```

```
	{
		Arr[top] = 0;
		Display(root->Left, Arr, top + 1);// recursively call with top + 1 till the leaf
node
	}
	/* If the node have the Right element till the symbol initialize the Array[top]
		1 (because according too the huffman algorithm ) */
	if (root->Right)// assign 1 to the right child path
	{
		Arr[top] = 1;
		Display(root->Right, Arr, top + 1);// recursively call with top + 1 till the leaf
node
	}

	/* if the node doesnot contain leftand right child
	then it will be the leaf node then print the saved codes in Array */
	if (!root->Left && !root->Right)// if the leaf node appear with no left and right child
	{
		int counter = 0;
		//Printing Symbol
		cout << setw(3) << setfill(' ') << "\'" << root->Huff.Symbol << "\' " << setw(15)
<< setfill(' ');
		for (int i = 0; i < top; i++)
		{
			cout << Arr[i];
			counter++;// counting the optimised codes per symbols
		}
		for (int i = 0; i < Array.size(); i++)
		{
			// checking for the symbol
			if (root->Huff.Symbol == Array[i].Symbol)
			{
				compressionRatio += Array[i].Freq * counter;// if found multiply with
its frequency and add in compression Ratio
			}
		}
		//cout << " ";
		cout << endl;
	}
}
//-----------------------------------------------------------------
void Compression::Decode(Data Huff[], int size)
{
	// priority Queue object
	priority_queue<HuffmanTree*, vector<HuffmanTree*>, compareNode>Pq;

	for (int i = 0; i < size; i++)
	{
		HuffmanTree* newNode = new HuffmanTree(Huff[i]);
		Pq.push(newNode);// pushing into the queue
	}
	// making huffman encoding tree
	HuffmanTree* root = Tree(Pq);
	int arr[Max_Size], top = 0;
	// print the optimized codes
	cout << " Symbol:\t    Codes:\n" << endl;
	Display(root, arr, top);
	// Printing final compression ratio
	cout << "\n  Compression Ratio: " << compressionRatio / Array.size() << endl;

}
```

```
//-----------------------------------------------------------------
```

Source.cpp

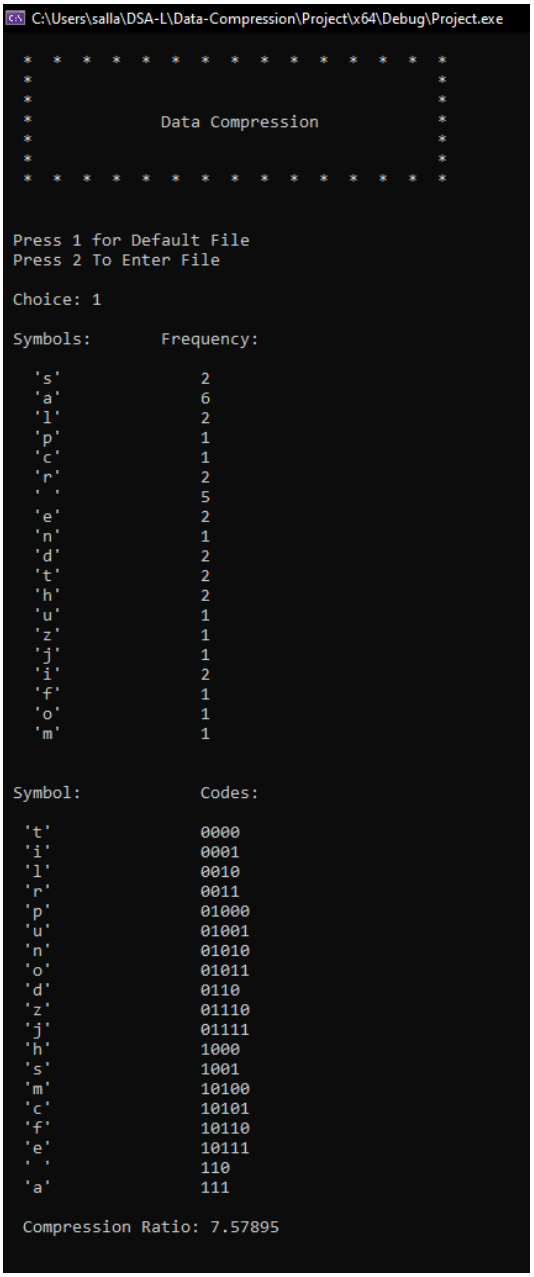
```
#include"Compression.h" // Adding Header
using namespace std;
int main()
{
	/*
	Styling
	*/
	int MAX_Row = 7, Max_Col = 15; // Defining Max Row and Col
	cout << endl; // End line
	for (int Row = 1; Row <= MAX_Row; Row++)  // For-Loop
	{
		cout << " ";//printing Space
		for (int Col = 1; Col <= Max_Col; Col++)//For-Loop
		{
			//For Printing the Box
			if (Row == 1 || Row == MAX_Row || Col == 1 || Col == Max_Col)
			{
				//Avoiding an Extra * At Printing Data Compression
				if (Row == 4 && Col == Max_Col)
				{
					cout << " "; //Printing Nothing
				}
				else
				{
					//Printing *
					cout << " * ";
				}
			}
			//For Printing Data Compression
			else if (Row == 4 && Col == 3)
			{
				//Printing Data Compression
				cout << "         Data Compression" << "\t    *"; // Last *
			}
			else
			{
				cout << "   ";//Printing Spaces
			}
		}
		cout << "\n";//Next Line
	}
	cout << endl << endl;
	//------------------ Styling Complete -----------------------------
	cout << " Press 1 for Default File\n Press 2 To Enter File" << endl;
	int choice = 0;//Declaring Variable
	while (choice != 1 && choice != 2)
	{
		cout << "\n Choice: "; cin >> choice;;//Input
		if (choice == 1)
		{
			Compression C; // Creating a Default Object
			C.Compress(); // Compressing
		}
		else if (choice == 2)
		{
```

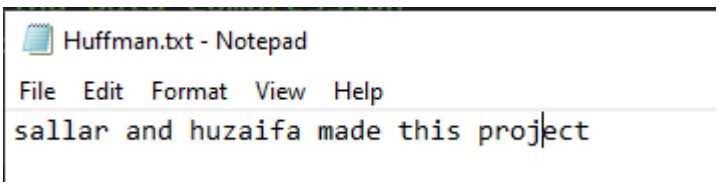
```
                    string fileName = " ";
                    cout << "\n Enter File Name: "; cin >> fileName; // Input
                    Compression C(fileName); // Creating Object with Input File Name
                    C.Compress();//Compressing File
                }
        }

        system("pause>0");
        return 0;
}
```

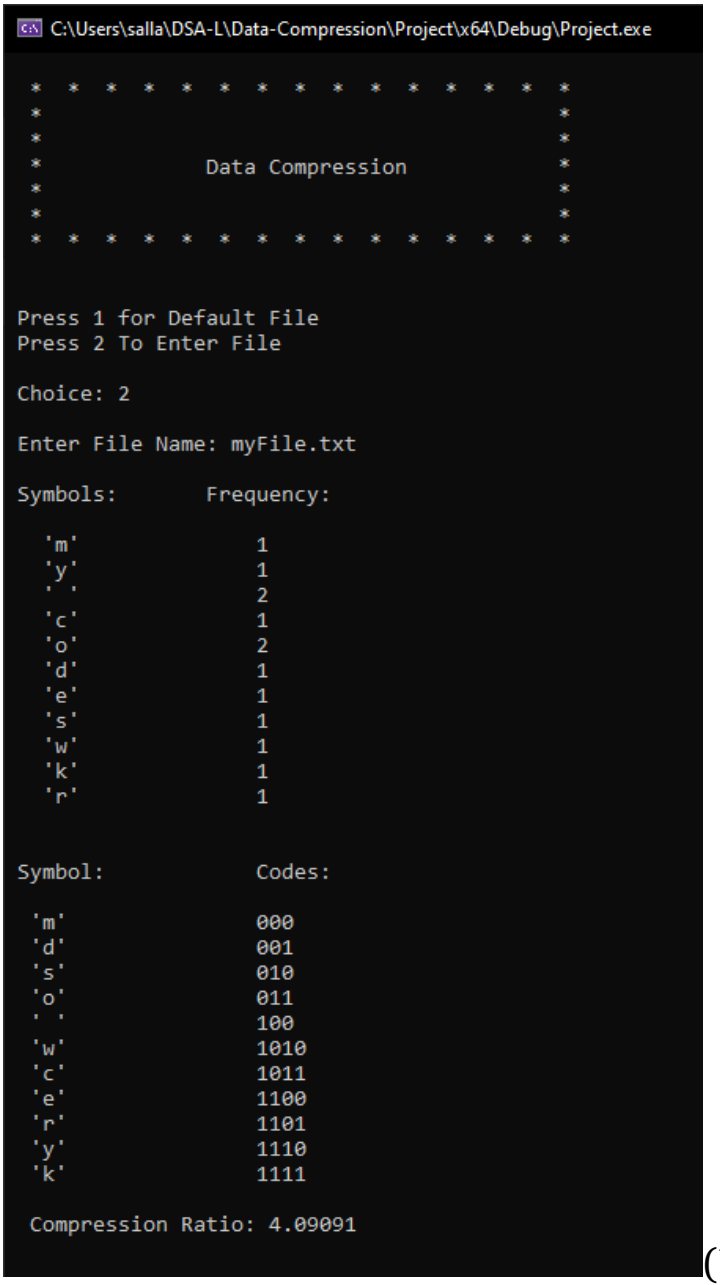
Output:

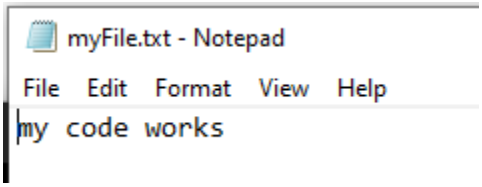
Two Outputs shown:





(Default File)





(User Defined/Input file)

Conclusion:

This Project includes the concept of basic computer programming, object-oriented programming (OOP) and Data Structures and Algorithms (DSA).

We built a data compression method. We convey the same information in a smaller amount of space given data. We Used Huffman Algorithm to Achieve this goal and compressed the Data.