**HUFFMAN FILE COMPRESSION ALGORITHM**

### Report submitted to Guru Gobind Singh Indraprastha University, Delhi (India)

In partial fulfillment of the requirement of the degree of

**B-TECH**

**In**

**COMPUTER SCIENCE**

**Submitted BY**

**HARSH PAHWA**

**Roll. No. 02415002717**

**To**



**DEPTT. OF COMPUTER SCIENCE**

**MAHARAJA SURAJMAL INSTITUTE OF TECHNOLOGY,**

**NEW DELHI-110058**

**BATCH 2017-2021**

**CANDIDATE DECLARATION**

I, Harsh Pahwa enrollment number 02415002717 student of B. Tech (Semester 5th) of Maharaja Surajmal Institute of Technology, New Delhi, hereby certify that the Summer Training Report entitled “HUFFMAN FILE COMPRESSON ALGORITHM” is my original work. The data provided in this document is authentic to the best of my knowledge. This is also certified that this study/data has not been submitted to any other institute for the award of any other degree/diploma.

Signature

**CERTIFICATE**

**ACKNOWLEDGEMENT**

I feel honoured to express my sincere gratitude and indebtedness to Mrs. Koyel Dutta HOD, Department of Computer Science , Maharaja Surajmal Institute of Technology, New Delhi for allowing me to undergo the Summer Training of six weeks at CODING NINJAS, PITAMPURA.

I owe my gratitude to my guide Ms. Nidhi Aggarwal for helping me to achieve the completion of the project which was assigned to me .Without her guidance, help and inspiration it would have been difficult for me to develop and complete the project successfully and in time.

I would fail in my duties if I do not sincerely thank and acknowledge the guidance and help of all the staff members of CODING NINJAS and making the training meaningful.

Submitted by:

HARSH PAHWA

Enrolment No: 02415002717

**ABSTRACT**

In data compression, source coding, or bit-rate reduction involves encoding information using fewer bits than the original representation. Compression can be either lossy or lossless. Lossless compression reduces bits by identifying and eliminating statistical redundancy. At some point we reach a physical limit of how fast we can send bits and if we want to send a large amount of information faster, we have to find a way to represent the same information with fewer bits - we must **compress** the data. Huffman Coding is an approach to text compression originally developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes". In computer science and information theory, it is one of many lossless data compression algorithms. It is a statistical compression method that converts characters into variable length bit strings and produces a prefix code. Most-frequently occurring characters are converted to shortest bit strings; least frequent, the longest. This project comprises the implementation of Huffman encoding algorithm and applying its application in real world. In this, we are traversing a text file and storing the characters and its frequency in a hashmap. From the hashmap we are constructing a priority queue by using a min heap. Now, using the least two frequencies repeatedly we form a binary tree called as Huffman tree.

Note : - A node containing the sum of the frequencies of the two nodes having least frequencies are added to the priority queue each time we remove two elements from the priority queue.

Now, we traverse the Huffman tree and assign 0 for each left and 1 for each right node and generate code for each character used in the file we store the character and its corresponding code in a hashmap now by this time we have generated a code for each character and now its ready for encoding. So for encoding we traverse the input file character by character and write the corresponding code in a new file that will be named as encoded file. After the full traversal of input file the encoded file is ready which will be the compressed file formed by using Huffman encoding algorithm.

For decoding we traverse the encoded file and search for the code then by using the code hashmap we find the corresponding character belonging to the code and write in a new file that will be named as decoded file.

**CONTENTS**

**S. No Title Page No.**

I CANDIDATE DECLARATION II

II CERTIFICATE (Organisation) III

III ACKNOWLEDGEMENT IV

IV ABSTRACT V

1 **CHAPTER 1: INTRODUCTION** 1- 5

1.1 NEED AND OBJECTIVE 1

1.2 METHODOLOGY 1

1.3 SOFTWARE USED 3

1.4 ABOUT ORGANISATION 4

2 **CHAPTER 2: PROJECT DESIGN** 5

2.1 BASICS OF FILE COMPRESSION

AND HUFFMAN ALGORITHM 5

2.2 DETAILED STEPS TO COMPRESS

TEXT FILE 7

3 **CHAPTER 3: IMPLEMENTATION** 14

3.1 HUFFMAN FILE COMPRESSION

CLASS 14

3.1.1 ENCODE FUNCTION 14

3.1.2 MAIN FUNCTION 14

3.1.2.1 CREATION OF FREQUENCY LIST 14

3.1.2.2 INSERTION INTO PRIORITY QUEUE 15

AND BUILDING A TREE

3.1.2.3 CREATING A LIST OF GENERATED

CODES 15

3.2 MIN PRIORITY QUEUE CLASS 17

3.2.1 IsEmpty, getSize, getMin FUNCTION 17

3.2.2 INSERT FUNCTION 18

3.2.3 REMOVE MIN FUNCTION 18

3.3 DATA\_CLASS CLASS 19

4 **CHAPTER 4: RESULT**

4.1 HUFFMAN\_FILE\_COMPRESSION 20

CLASS

4.2 MIN\_PRIORITY\_QUEUE\_CLASS 23

4.3 DATA\_CLASS 24

5 **CHAPTER** **5**

5.1 FUTURE SCOPE 27

5.2 CONCLUSION 27

6 REFERENCES 28

**LIST OF FIGURES**

**Figure no Title Page No.**

1.1 Eclipse IDE 4

1.2 Coding Ninjas logo 4

2.1 Step 1 9

2.2 Step 2 9

2.3 Step 3 9

2.4 Step 4 10

2.5 Step 5 11

2.6 Huffman tree 11

3.1 Encoding function. 14

3.2 Frequency list. 14

3.3 Min priority queue class 15

3.4 Addition of elements in heap 15

3.5 Encoding 16

3.6 Decoding 16

3.7 Exception handled 17

3.8 Min priority queue class 17

3.9 Insert function 18

3.10 Remove min element function 19

3.11 Character data storing class 19

4.1 Program code class 1 20-22

4.2 Program code class 2 23

4.3 Program code class 3 24

4.4 Console output 1 25

4.5 Console output 2 25

4.6 Console output 3 25

4.7 Input file 26

4.8 Encoded File 26

4.9 Decoded File 26

**LIST OF TABLES**

**Table no. Title Page No.**

2.1 Character codes 6

2.2 Hashmap 8

2.3 Priority queue 9

**Chapter 1**

**INTRODUCTION**

**1.1 NEED & OBJECTIVE**

Beginning in the 1970s, computer scientists used mathematical algorithms to search through computer code to find ways to reduce the file size. Since then, there has been an ever-growing need, spurred on by the development of the Internet, to create better compression schemes and reduce the size of any given file as much as possible.

Our project as the name says is also based on text file compression method. File compression system is basically an application of Huffman encoding or Huffman coding. We feel that there is an immense need of file compression these days because data generated in a computer is much bigger in size and to save space and for sustainable use of computer memory we compress the text files so that more space gets available for the data which is more important. For example, if an user installs a game into his computer, the setup and exe file will be taking enormous space, so to save space and download time we compress those files which are not much important to us. The most important factors for which we consider compressing the files is because data transfer from one place to another will be more easy as the size of the file will be lesser resulting in faster transfer and receiving of data.

**1.2 METHODOLOGY**

There are two main types of file compression, lossless and lossy. Lossless compression algorithms search for long strings of code and have a method to replace them with shorter strings. Lossless compression can recreate the entire file exactly as it was. Lossy compression algorithms search through the code to find pieces that it can delete. While lossy compression can't be used on program files, it can be used on multimedia files, where there is often information in the files that human senses cannot detect. When lossy compression is used, the file may appear to be identical, but it is very different at the code level.

This project is totally based on the lossless compression method called Huffman encoding where there is fixed length and variable length encoding, uniquely decodable codes, prefix rules and construction of Huffman Tree. We already know that every character is stored in sequence of O's and 1's using 8 bits. This is called fixed-length encoding as each character uses the same number of fixed bits storage.

*Given a text, how can we reduce the amount of space required to store a character?*

The idea is to use variable-length encoding. We can exploit the fact that some characters occurs more frequently than others in a text (refer this) to design an algorithm which can represent the same piece of text using lesser number

of bits. In variable-length encoding, we assign variable number of bits to characters depending upon their frequency in the given text. So, some character might end up taking 1 bit, some might end up taking two bits, some will be encoded using three bits, and so on.

*Given a sequence of bits, how can we decode it uniquely?*

Let’s consider the string "aabacdab". It has 8 characters in it and uses 64 bits storage (using fixed-length encoding). If we

Note, the frequency of characters 'a, 'b', 'c’ and 'd' are 4, 2, 1, 1 respectively. Let’s try to represent "aabacdab" using lesser

number of bits by using the fact that 'a' occurs more frequently than 'b' and 'b' occurs more frequently than C and 'd

We start by randomly assigning single code 0 to 'a', 2-bit code 11 to 'b' and 3-bit code 100 and 011 to characters

And ’d’ respectively.

a 0

b 11

c 100

d 011

So the string aabacdab will be encoded to (00110100011011 0110100011011) using above codes. But the real problem lies in decoding. If we try to decode the string 00110100011011, it will lead to ambiguity as it can be decoded to,

0|011 |0|100|011|0|11 adacdab

0|0|11|0|100 |0|11|011 aabacabd

0|011|0|100|0|11|0|11 adacabab

and so on…

To prevent ambiguities in decoding, we will ensure that your encoding satisfies what's called the prefix rule which will result into uniquely decodable codes. The prefix rule states that no code is a prefix of another code. By code, we mean the bits used for a particular character. In above example, 0 is prefix of 011 which violates the prefix rule. So if our codes satisfies the prefix rule, the decoding will be unambiguous (and vice versa). Let’s consider above example again. This time we assign codes that satisfies the prefix rule to characters 'a', b', ‘c’ and 'd'.

a 0

b 10

c 110

d 111

Using above codes, the string aabacdab will be encoded to 00100100011010 (0|0|10|0|100|011|0|10). Now we can uniquely decode 00100100011010 back to our original string aabacdab.

Now that we are clear on variable length encoding and prefix rule, let's talk about Huffman coding. The technique works by creating a binary tree of nodes. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the character itself, the weight (frequency of appearance) of the character. Internal nodes contain character weight and links to two child nodes. As a common convention, bit O' represents following the left child and bit'' represents following the right child. A finished tree has n leaf nodes and n-1 internal nodes. It is recommended that Huffman tree should discard unused characters in the text to produce the most optimal code lengths.

We will use priority queue for building Huffman tree where the node with lowest SOF

Below are the complete steps -

1. Create a leaf node for each character and add them to the priority queue.

2. While there is more than one node in the queue.

a. Remove the two nodes of highest priority (lowest frequency) from the queue.

b. Create a new internal node with these two nodes as children and with frequency

equal to the sum of the two nodes' frequencies.

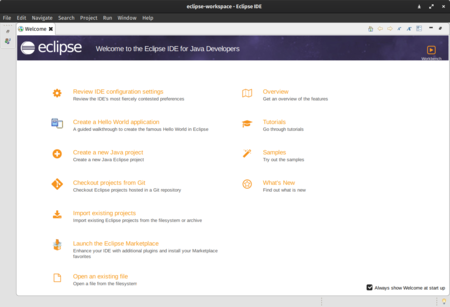
c. Add the new node to the priority queue.

3. The remaining node is the root node and the tree is complete.

**1.4 SOFTWARE USED**

The software used to implement this whole project is Eclipse IDE.

**Eclipse** is an integrated development environment (IDE) used in computer programming, and in 2014 was the most widely used Java IDE in one website's poll. It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java and its primary use is for developing Java applications.



**Fig 1.1: Eclipse IDE**

**1.5 ABOUT ORGANISATION**

****

**Fig 1.2: Coding Ninjas logo**

Coding Ninjas is an online as well offline coaching platform which provides various courses to the students who are willing to learn something from scratch in Java with ds algorithm , Python ML , Wed Development course etc.

Their main qualities which stands them above all leading institutes are as follows:-

**Exceptional Faculty**

Our faculty is from the best institutes around the world such as IITD, Stanford and more. They also have work experience in tech giants like Amazon, Facebook and Adobe which makes them exceptional.

**Mentor Support**

An incredible mentor support is provided where 1 teaching assistant is assigned to 10 students. The TA’s are top performers alumni dedicated to support and clear doubts at any point of time.

**CodeZen**

An online platform using proprietary technologies, social elements and well-crafted content curriculum delivering the finest learning experience.

**Placements**

We do not make false promise to provide 100 % placements rather help students based on their performance. Our students are placed at bigwigs like Adobe, Amazon, Microsoft and many more.

**Course curriculum**

Industry leading curriculum designed by expert developers turned educators who have invested time to create quality content infused with unique teaching methodology.

**Get Secured with Certificate**

The student will be provided the certificates which are signed by instructors and have our logo on it which will help them secure their career and increase job prospects with certificates signed by us.

**CHAPTER 2**

**PROJECT DESIGN**

**2.1 BASICS OF FILE COMPRESSION AND HUFFMAN ALGORITHM**

The total idea of Text file compression is based on Huffman encoding. Huffman encoding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing textual data to make a file occupy a smaller number of bytes. The idea of Huffman encoding is to abandon the rigid 8-bits-per-character requirement, and instead to use binary encodings of different lengths for different characters. The advantage of doing this is that if a character occurs frequently in the file, such as the very common letter 'e', it could be given a shorter encoding (i.e., fewer bits), making the overall file smaller. The tradeoff is that some characters may need to use encodings that are longer than 8 bits, but this is reserved for characters that occur infrequently, so the extra cost is worth it, on the balance. The table below compares ASCII values of various characters to possible Huffman encodings for some English text. Frequent characters such as space and 'e' have short encodings, while rarer characters (like 'z') have longer ones.

**Table 2.1: Character codes**

|  |  |  |  |
| --- | --- | --- | --- |
| Character | ASCII Value | ASCII Binary | Huffman Binary |
| **‘ ’** | 32 | 00100000 | 10 |
| **‘a’** | 97 | 01100001 | 0001 |
| **‘b’** | 98 | 01100010 | 0111010 |
| **‘c’** | 99 | 01100011 | 001100 |
| **‘e’** | 101 | 01100101 | 1100 |
| **‘z’** | 122 | 01111010 | 00100011010 |

The steps that include in the Algorithm of Huffman encoding is as follows:-

**1** – Read the file and calculate the frequency of each character that can be A-Z,

a-z , 0-9, special character like space etc. and store the information in a hashmap.

For example:-

We will store the information of frequencies in a hashmap as shown in the table.

**2 –** Build a minimum priority queue (heap) of this particular hashmap that means the character having lowest priority, its frequency will be on the top.

**3 –** Remove the 2 elements from minimum heap having lowest frequency and combine them to form a new entry. Reinsert into priority queue and store the deleted elements to form a binary tree and create a Huffman tree.

**4 –** Generation of the code and storing in a new hashmap called “coded hashmap”.

**5 –** Encoding the given text file using the help of coded hashmap and compressing it into smaller file size.

**6 –** Decoding the compressed file with the help of coded hashmap.

Now, let’s discuss about this whole procedure in a detailed manner.

**2.2 DETAILED STEPS TO COMPRESS TEXT FILE**

**STEP 1 - CALCULATING THE FREQUENCIES OF CHARACTERS AND CREATING A HASHMAP**

Let us suppose, we need to store a string of length 1000 that comprises characters a, e, n, and z. To storing it as 1-byte characters will require 1000 byte (or 8000 bits) of space. If the symbols in the string are encoded as (00=a, 01=e, 10=n, 11=z), then the 1000 symbols can be stored in 2000 bits saving 6000 bits of memory. The number of occurrence of a symbol in a string is called its frequency. When there is considerable difference in the frequencies of different symbols in a string, variable length codes can be assigned to the symbols based on their relative frequencies. The most common characters can be represented using shorter codes than are used for less common source symbols. More is the variation in the relative frequencies of symbols, it is more advantageous to use variable length codes for reducing the size of coded string.

After that we will create a hashmap of the frequencies of the characters occurring in a text file and it will be stored as a table format which is shown below as an example.

**Table 2.2: Hashmap**

|  |  |
| --- | --- |
| **KEY(CHARACTER)** | **OCCURRENCE OF THE CHARACTER IN THE TEXT FILE** |
| ‘a’ | 4 |
| ‘b’ | 5 |
| ‘c’ | 3 |
| ‘e’ | 2 |
| ‘f’ | 7 |

**STEP 2 – CREATING A MINIMUM PRIORITY QUEUE.**

Huffman’s algorithm builds a minimum priority queue as follows. First, we place our counts into hashmaps. Then, we put the key and values into a priority queue, which stores them in prioritized order, where smaller counts have a higher priority. This means that characters with lower counts will come out of the queue sooner.

Adding the key and values of the hashmap in the priority queue will sort the hashmap in ascending order, by ascending order it means the character with least frequency will be given highest priority and the character with highest frequency will be given least priority. In some cases where there is same frequency in two characters then that character will be added first which is first placed in the hashmap.

The above table will look like the table the shown below after adding the key and values to priority queue.

**Table 2.3: Priority Queue**

|  |  |
| --- | --- |
| ‘e’ | 2 |
| ‘c’ | 3 |
| ‘a’ | 4 |
| ‘b’ | 5 |
| ‘f’ | 7 |

**STEP 3** – **FORMATION OF HUFFMAN TREE**

We will use priority queue for building Huffman tree where the node with lowest frequency is given highest priority.

Below are the complete steps -

1. Create a leaf node for each character and add them to the priority queue.

2. While there is more than one node in the queue:

a. Remove the two nodes of highest priority (lowest frequency) from the queue

b. Create a new internal node with these two nodes as children and with

frequency equal to the sum of the two nodes' frequencies.

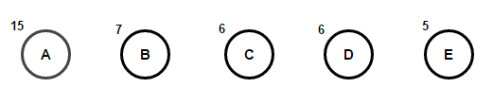
c. Add the new node to the priority queue.

3.The remaining node is the root node and the tree is complete.

Consider some text consisting of only 'A', 'B', 'C', 'D' and 'E character and their frequencies are 15,7,6,6,5 respectively.

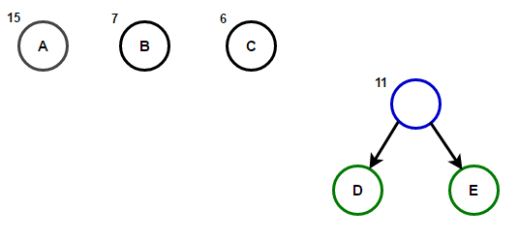
Below figures illustrate the steps followed by the algorithm –

**STEP 1**



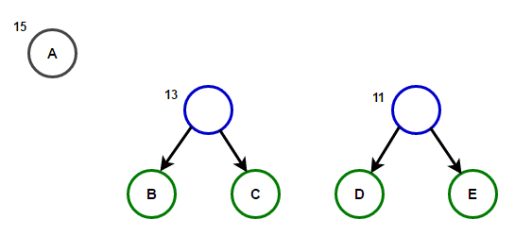
**Fig 2.1**

**STEP 2**



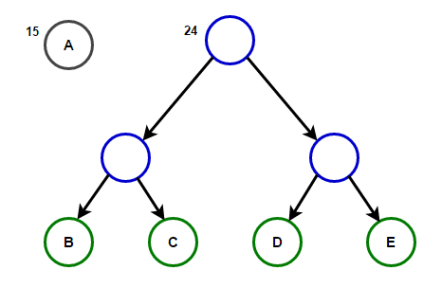
**Fig 2.2**

**STEP 3**

****

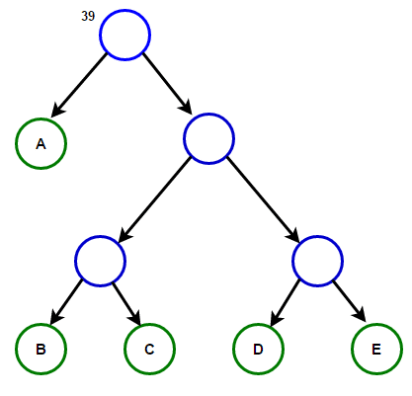
**Fig 2.3**

**STEP 4**



**Fig 2.4**

**STEP 5**



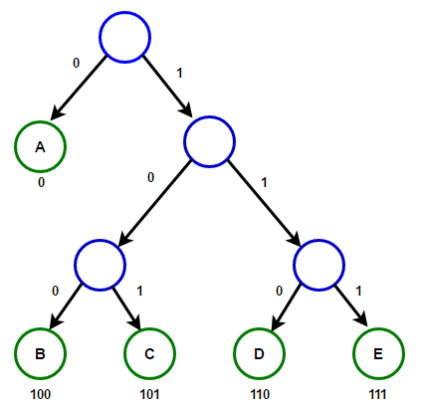
**Fig 2.5**

**HUFFMAN TREE IS CREATED**

**STEP 4 – GENERATION OF CODES AND STORING IN A HASHMAP.**

After we have constructed the Huffman tree, we will assign weights to all the edges. Let us assign weight ‘0’ to the left edges and weight ‘1’ to the right edges.

The above Huffman tree will be given the weights and the final tree will look as shown below.



**Fig 2.6**

After that, the text will be readed from the file character by character and appropriate codes will be assigned to each character using the Huffman tree and will be saved in a hashmap which will be called as “coded hashmap”.

**STEP 5 – ENCODING**

Using the encoding map, we can encode the file’s text into a shorter binary representation. Since the character encodings have different lengths, often the length of a Huffman-encoded file does not come out to an exact multiple of 8 bits. Files are stored as sequences of whole bytes, so in cases like this the remaining digits of the last bit are filled with 0s. You do not need to worry about implementing this; it is part of the underlying file system. It might worry you that the characters are stored without any delimiters between them, since their encodings can be different lengths and characters can cross byte boundaries, as with 'a' at the end of the second byte. But this will not cause problems in decoding the file, because Huffman encodings by definition have a useful prefix property where no character's encoding can ever occur as the start of another’s encoding.

**STEP 6 - DECODING**

You can use a Huffman tree to decode text that was previously encoded with its binary patterns. The decoding algorithm is to read each bit from the file, one at a time, and use this bit to traverse the Huffman tree. If the bit is a 0, you move left in the tree. If the bit is 1, you move right. You do this until you hit a leaf node. Leaf nodes represent characters, so once you reach a leaf, you output that character. For example, suppose we are given the same encoding tree above, and we are asked to decode a file containing the following bits: 1110010001001010011. Using the Huffman tree, we walk from the root until we find characters, then output them and go back to the root.

• We read a 1 (right), then a 1 (right). We reach 'b' and output b. Back to the root. 1110010001001010011

• We read a 1 (right), then a 0 (left). We reach 'a' and output a. Back to root. 1110010001001010011

• We read a 0 (left), then a 1 (right), then a 0 (left). We reach 'c' and output c.1 110010001001010011

• We read a 0 (left), then a 0 (left). We reach ' ' and output a space. 1110010001001010011

• We read a 1 (right), then a 0 (left). We reach 'a' and output a. 1110010001001010011

• We read a 0 (left), then a 1 (right), then a 0 (left). We reach 'c' and output c. 1110010001001010011

• We read a 1 (right), then a 0 (left). We reach 'a' and output a. 1110010001001010011

• We read a 0, 1, 1. This is our EOF encoding pattern, so we stop. The overall decoded text is “bac aca”.

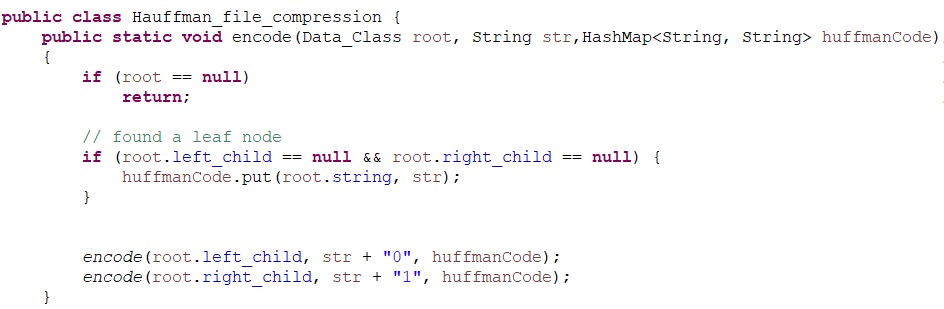
**CHAPTER 3**

**IMPLEMENTATION**

The detailed implementation of the java program is explained below.

**3.1 HUFFMAN FILE COMPRESSION CLASS**

**3.1.1 ENCODE FUNCTION**



**Fig 3.1: Encoding Function**

This function generates a code corresponding to each character and will store in a hashmap using recursion.

**3.1.2 MAIN FUNCTION**

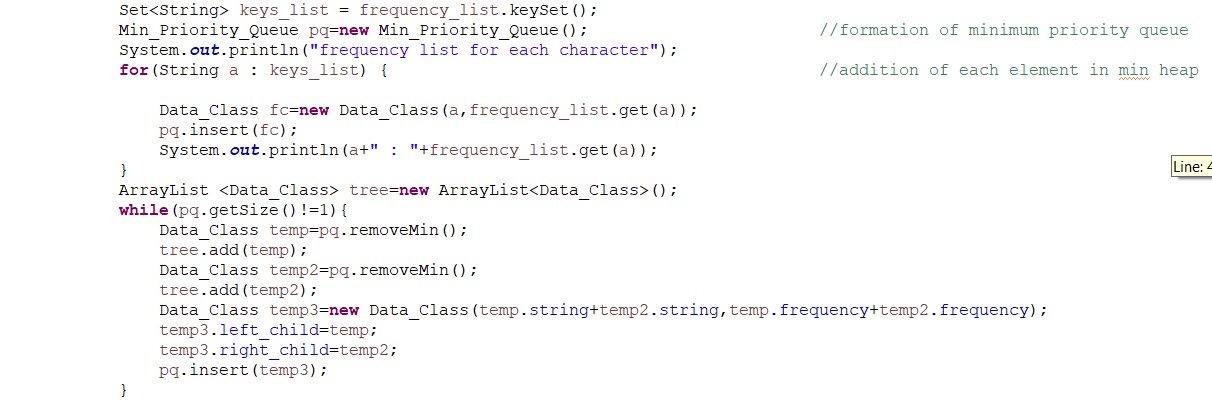
**3.1.2.1 CREATION OF FREQUENCY LIST**

****

**Fig 3.2: Frequency list creator**

In these lines we are reading the input file and creating a hashmap which contains the characters as a string with their respective frequencies in the text.

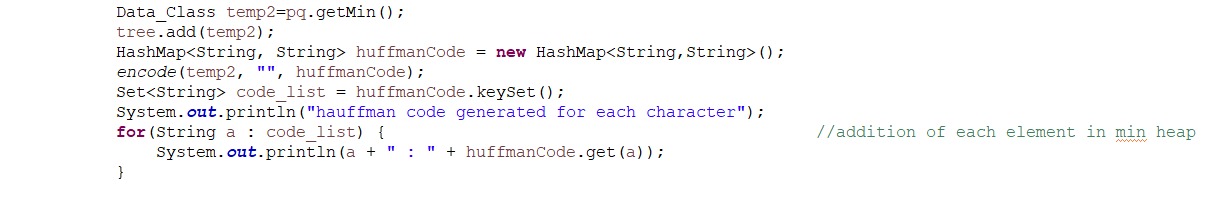
**3.1.2.2 INSERTION INTO PRIORITY QUEUE AND BUILDING A TREE**



**Fig 3.3: Min priority queue formation and addition of elements**

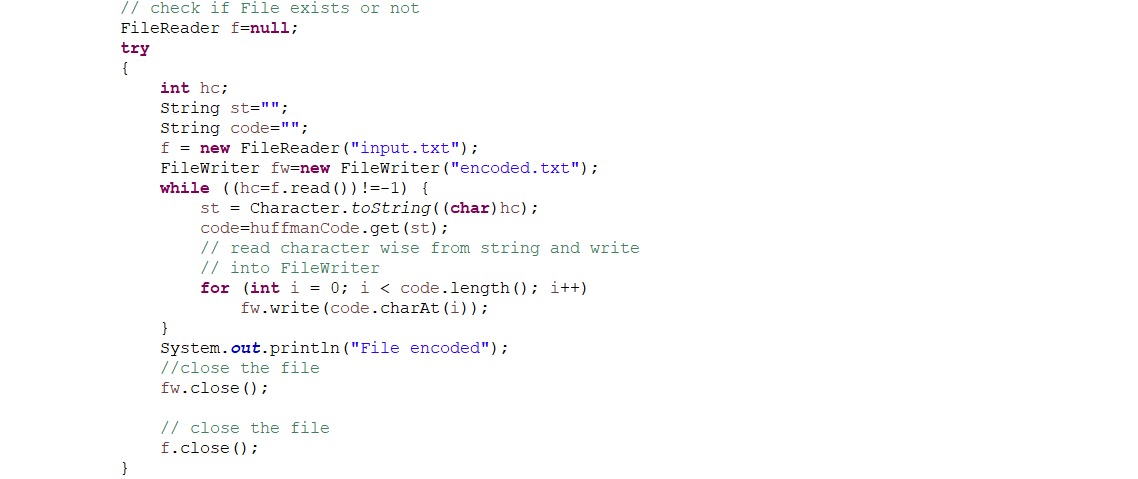
In these lines we are traversing the elements of frequency list and inserting them in a priority queue and after that we are removing 2 elements at a time in the priority queue and forming a tree.

**3.1.2.3 CREATING A LIST OF GENERATED CODES**

****

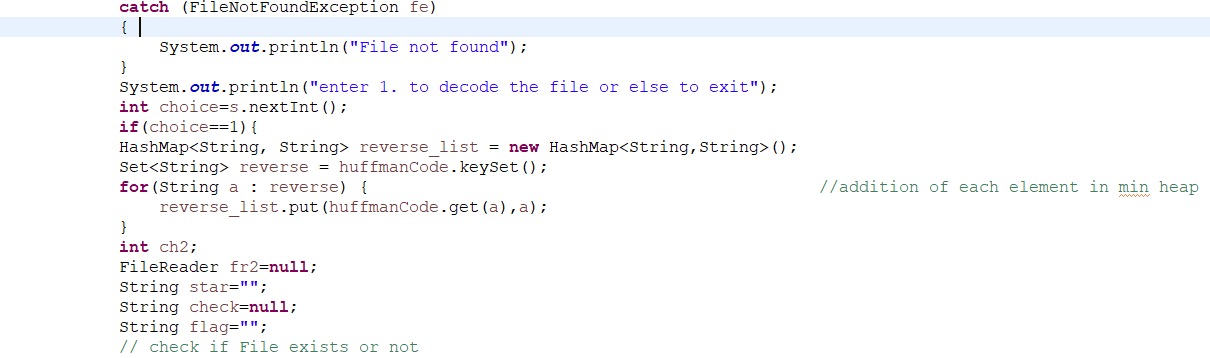
**Fig 3.4: Addition of elements in heap**

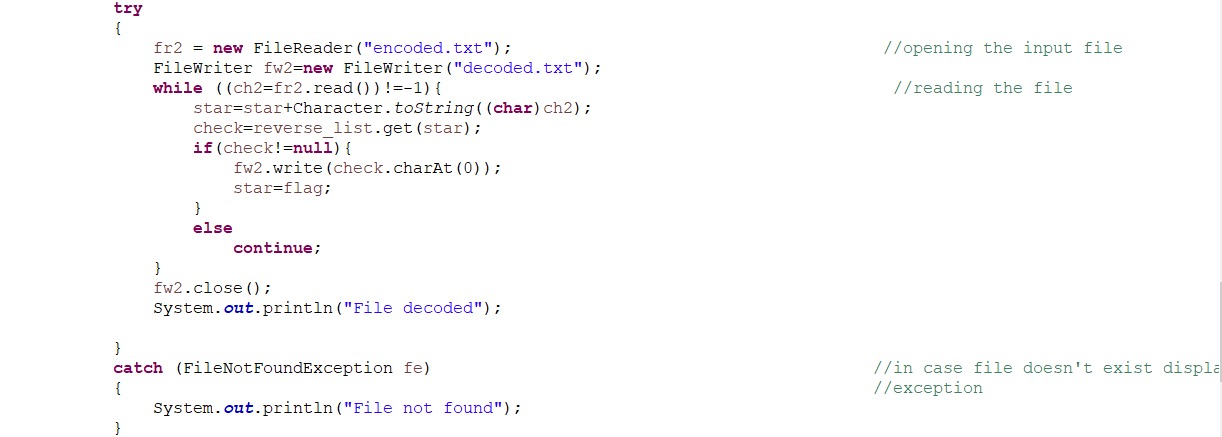
In these lines we are calling the encode function which inserts the codes for the respective characters in hashmap and thus printing the elements of the hashmap containing codes.



**Fig 3.5 Encoding**

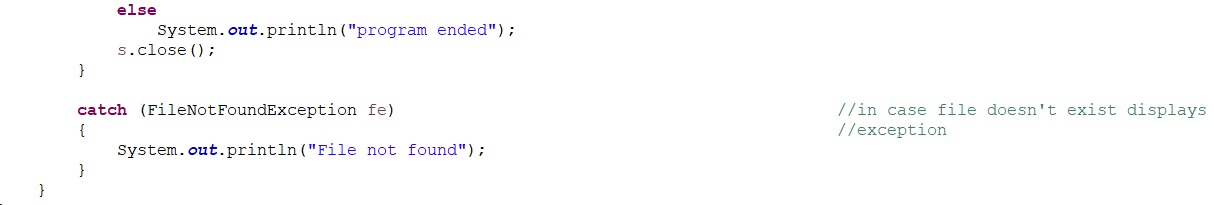
In fig 3.5 the encoded file is generated by traversing each character of input file and writing the corresponding code to the encoded file.





**Fig 3.6 Decoding**

In fig 3.6 we are traversing the encoded file by recognising the code we write the corresponding character in the decoded file.

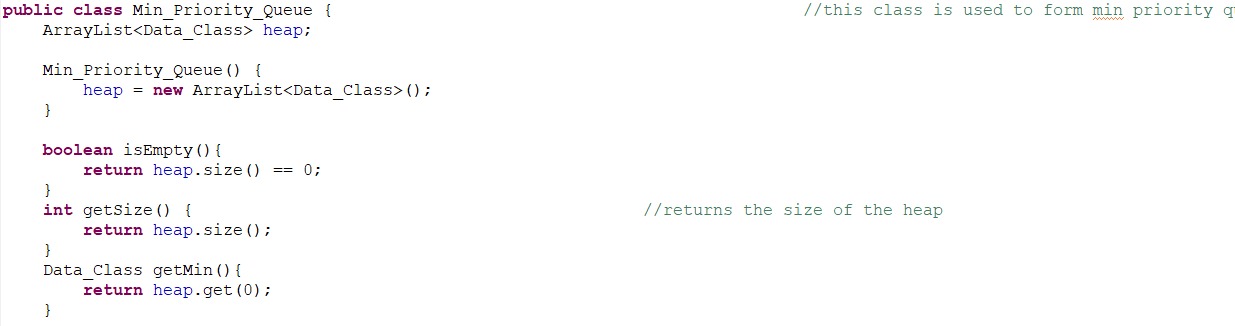


**Fig 3.7 Exception handled**

In fig 3.7 we are handling exception that is in case when file doesn’t exist this displays exception.

**3.2 MIN\_PRIORITY\_QUEUE CLASS**

**3.2.1 IsEmpty(), getSize(), getMin() FUNCTIONS**



**Fig 3.8: Min priority queue class**

This figure contains 3 functions which are as follows:-

1. IsEmpty() function: - It returns whether the heap is empty or not.
2. getSize () function: - It returns the size of the heap.
3. getMin () function: - It returns the minimum element of the heap.

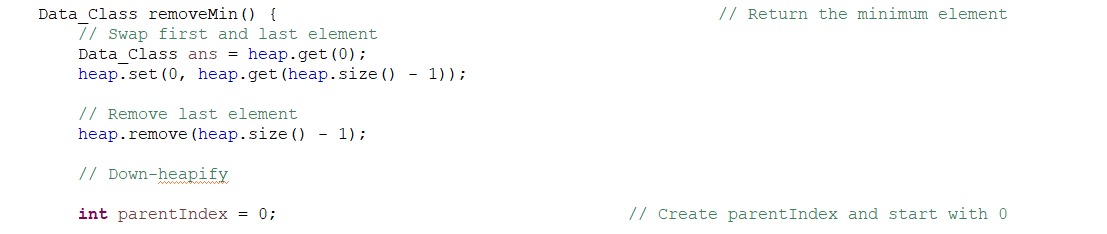
**3.2.2 INSERT FUNCTION**

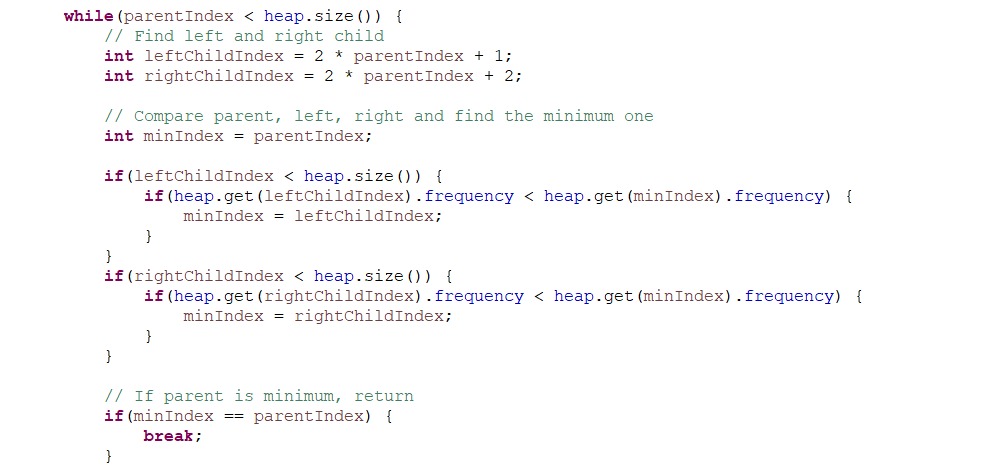


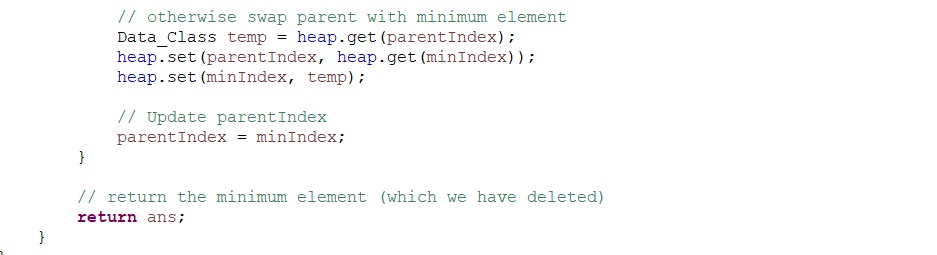
**Fig 3.9: Insert function**

This function receives the element and inserts that element to the priority queue.

**3.2.3 REMOVE MIN FUNCTION**

****

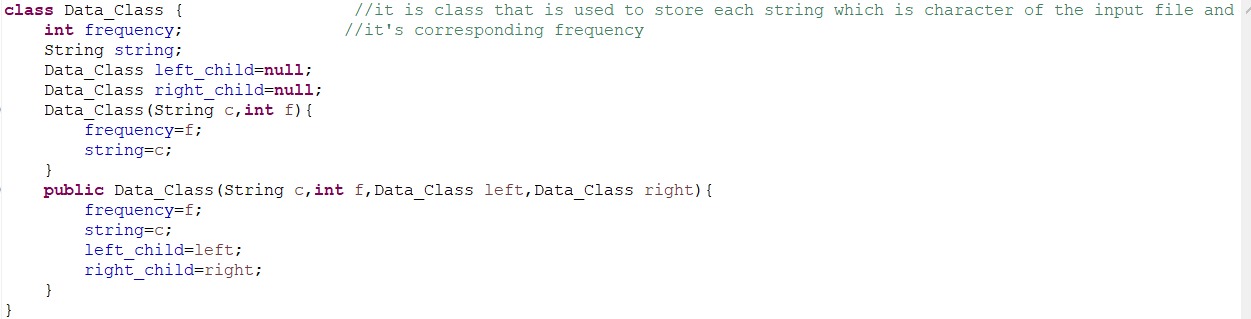
****

****

**Fig 3.10: Remove minimum element function**

This function removes the topmost element that is the minimum element of priority queue and returns it.

**3.3 DATA\_CLASS CLASS**

****

**Fig 3.11: Character data storing class**

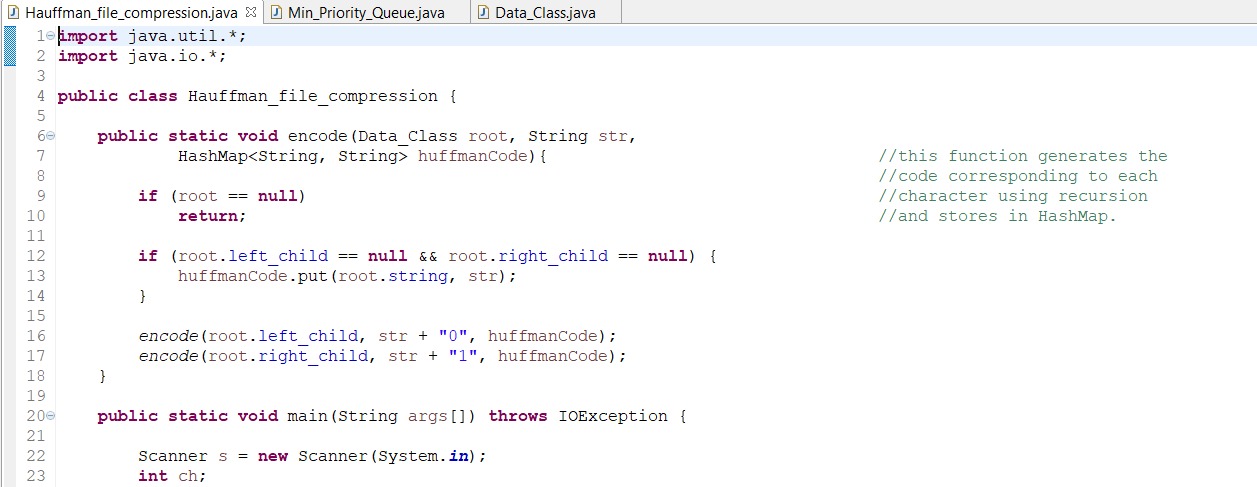
This class contains four members that is frequency, string, left\_child and right\_child.

These are used to store values of character as a string and their respective frequencies and further used in building priority queue and Huffman tree.

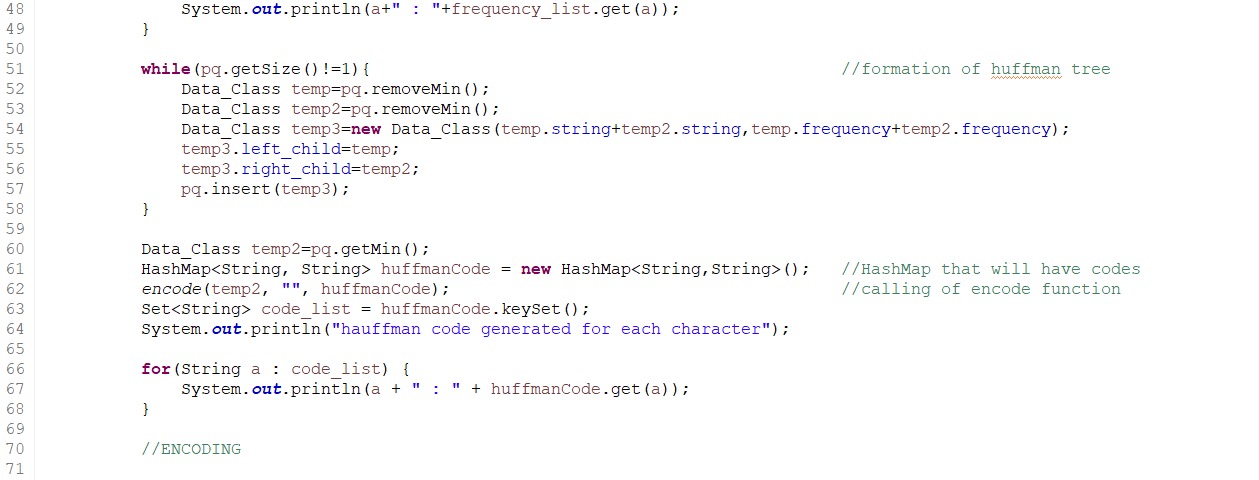
**Chapter 4**

**RESULT**

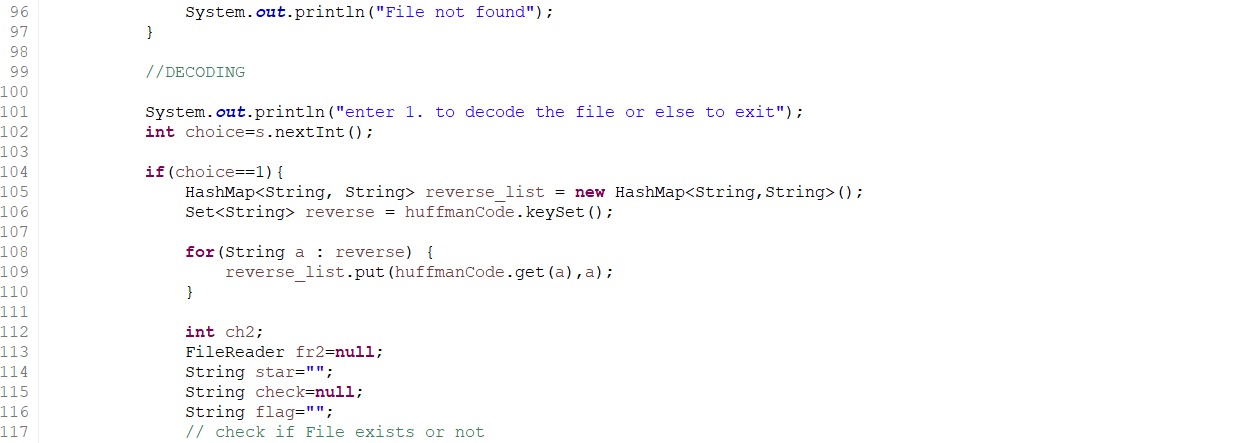
**4.1 HUFFMAN\_FILE\_COMPRESSION CLASS**

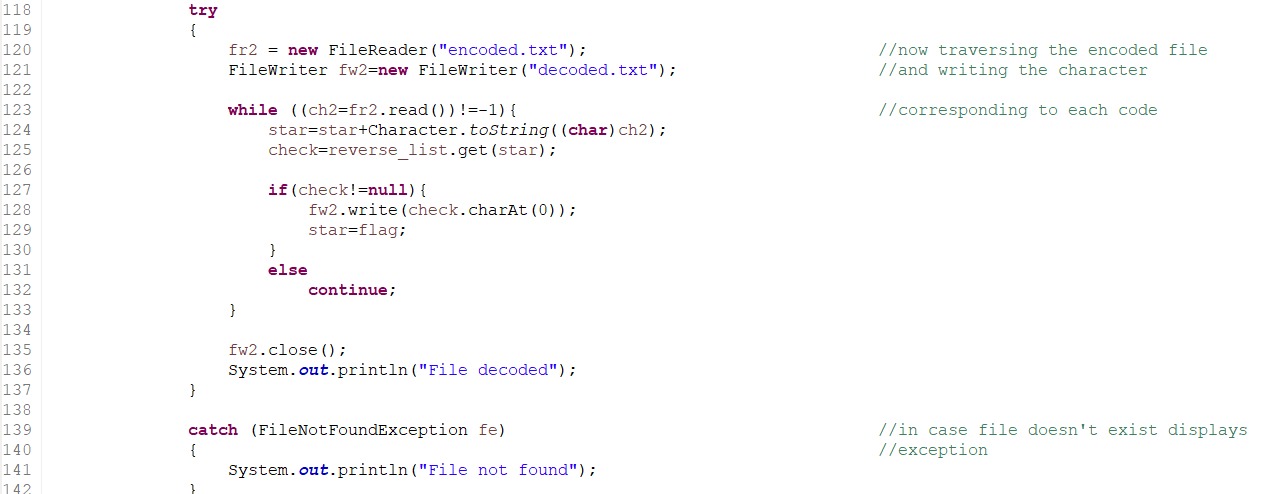


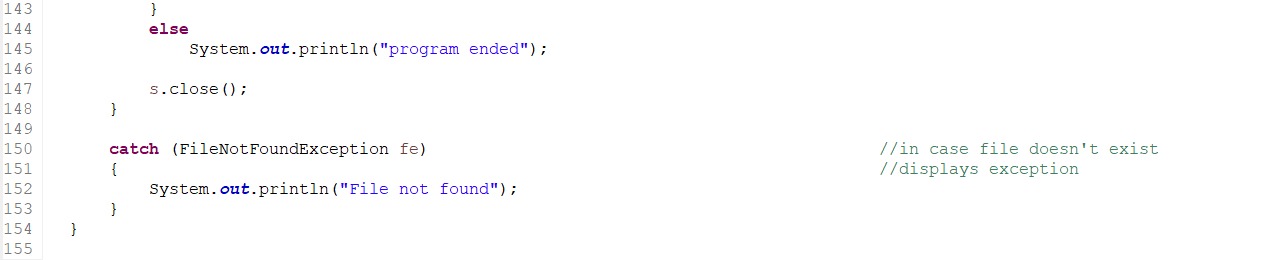






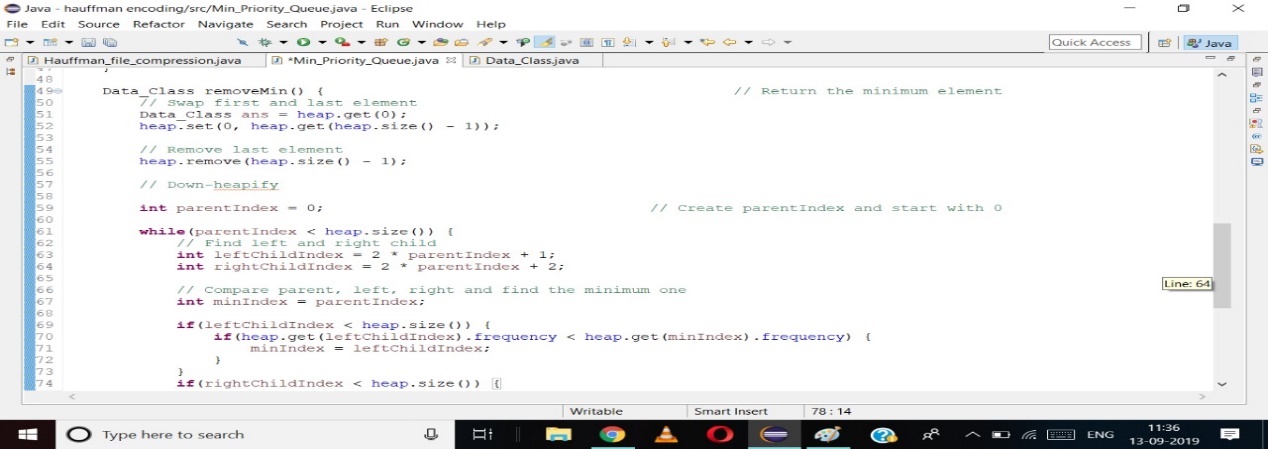
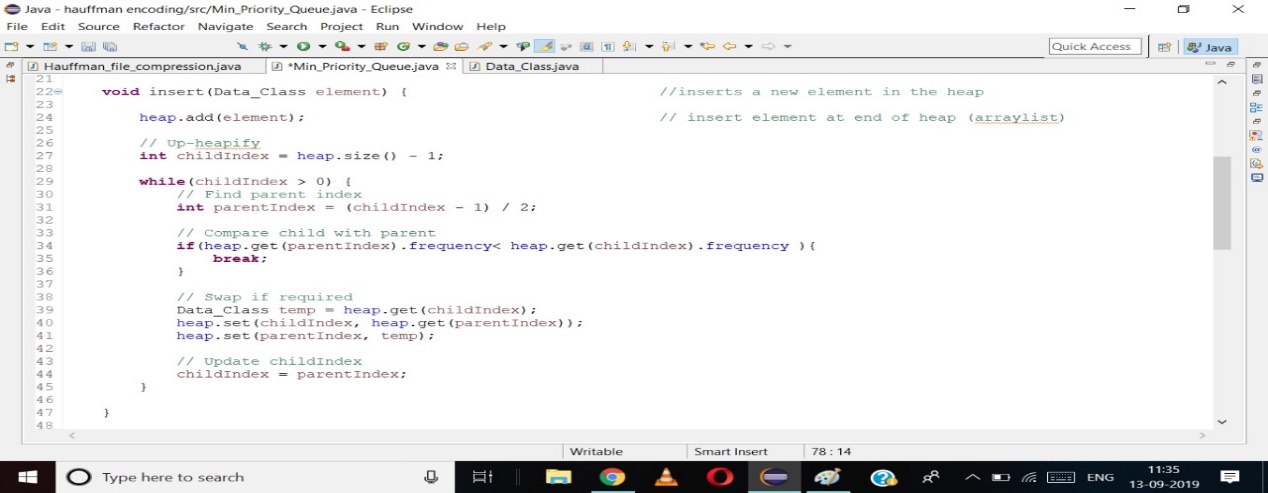


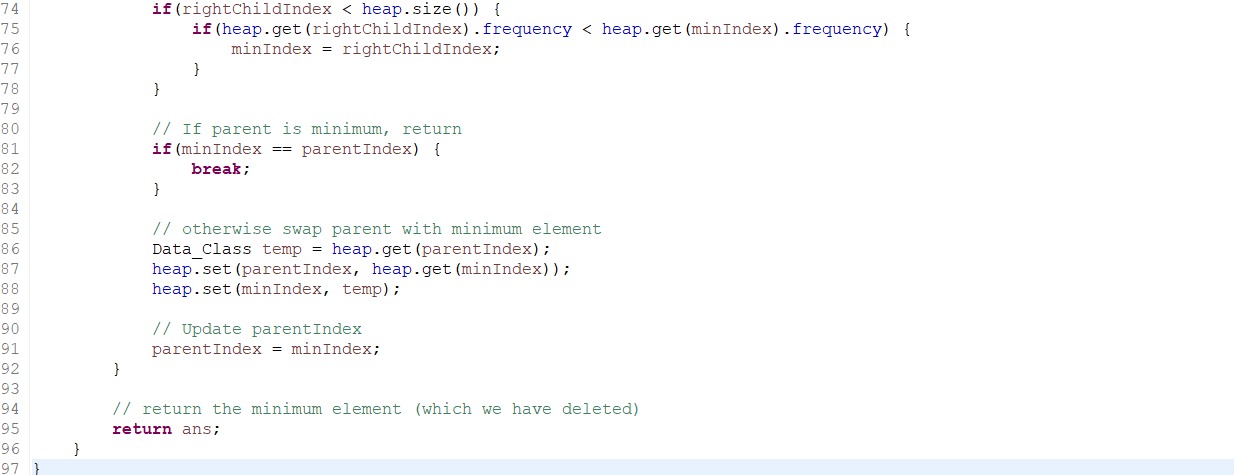




**Fig 4.1 Program code class1**

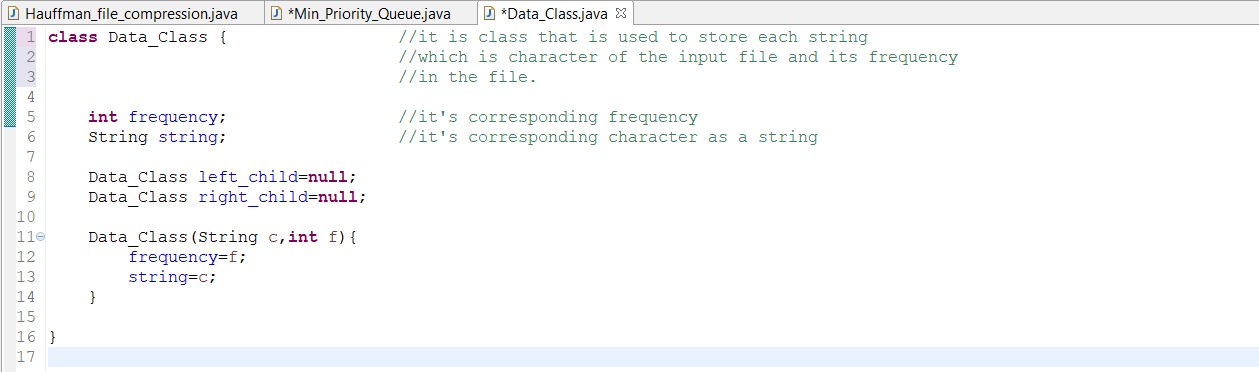
**4.2 MIN\_PRIORITY\_QUEUE CLASS**





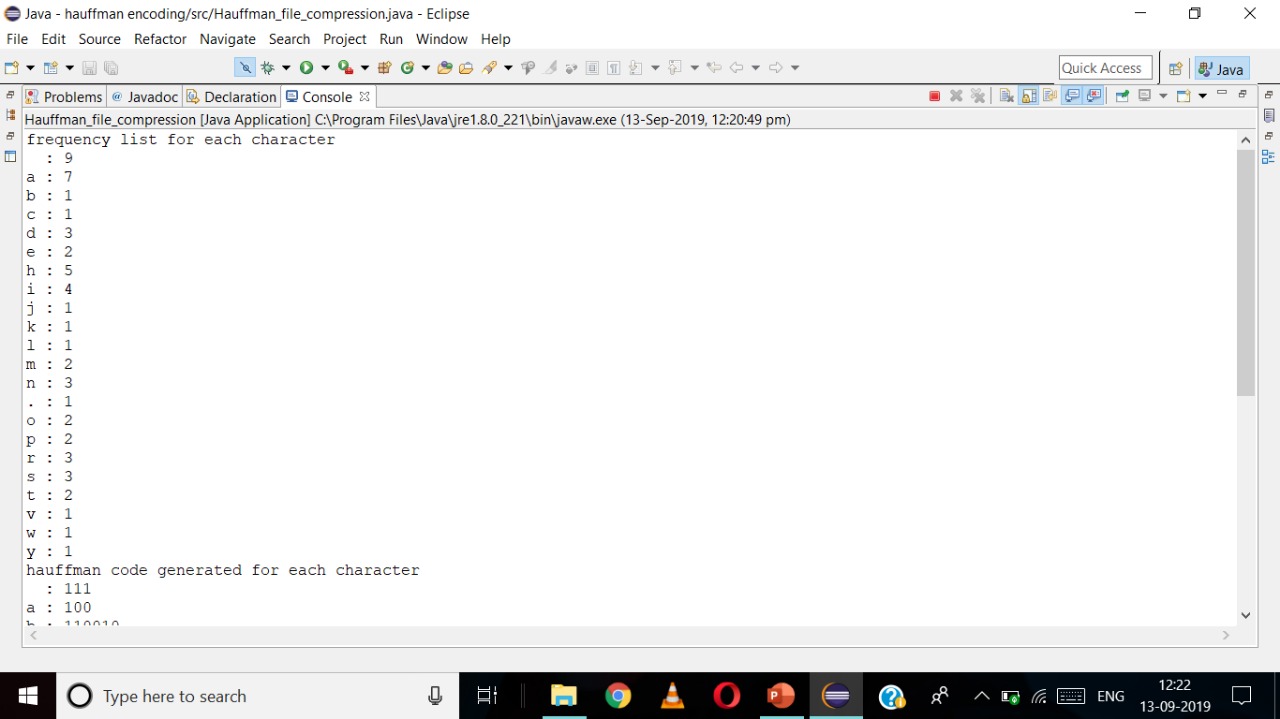
**Fig 4.2 program code class 2**

**4.3 DATA\_CLASS**



**Fig 4.3 program code class 3**

**4.4 OUTPUT**



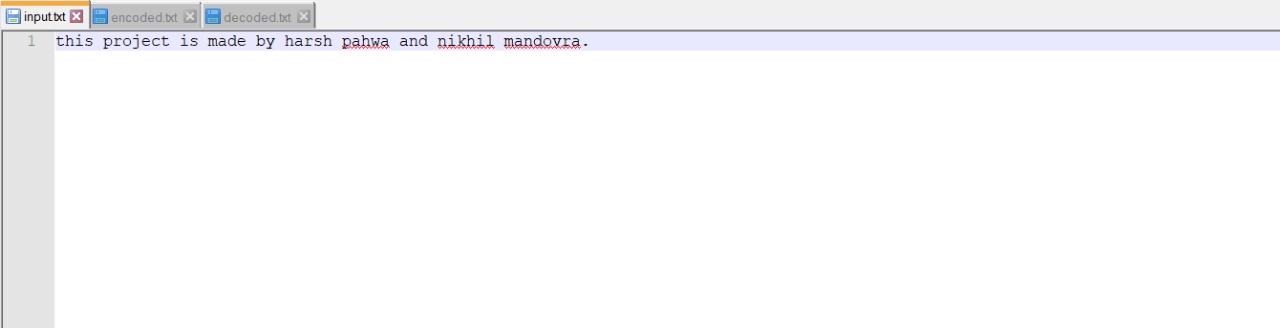
**Fig 4.4 console output1**



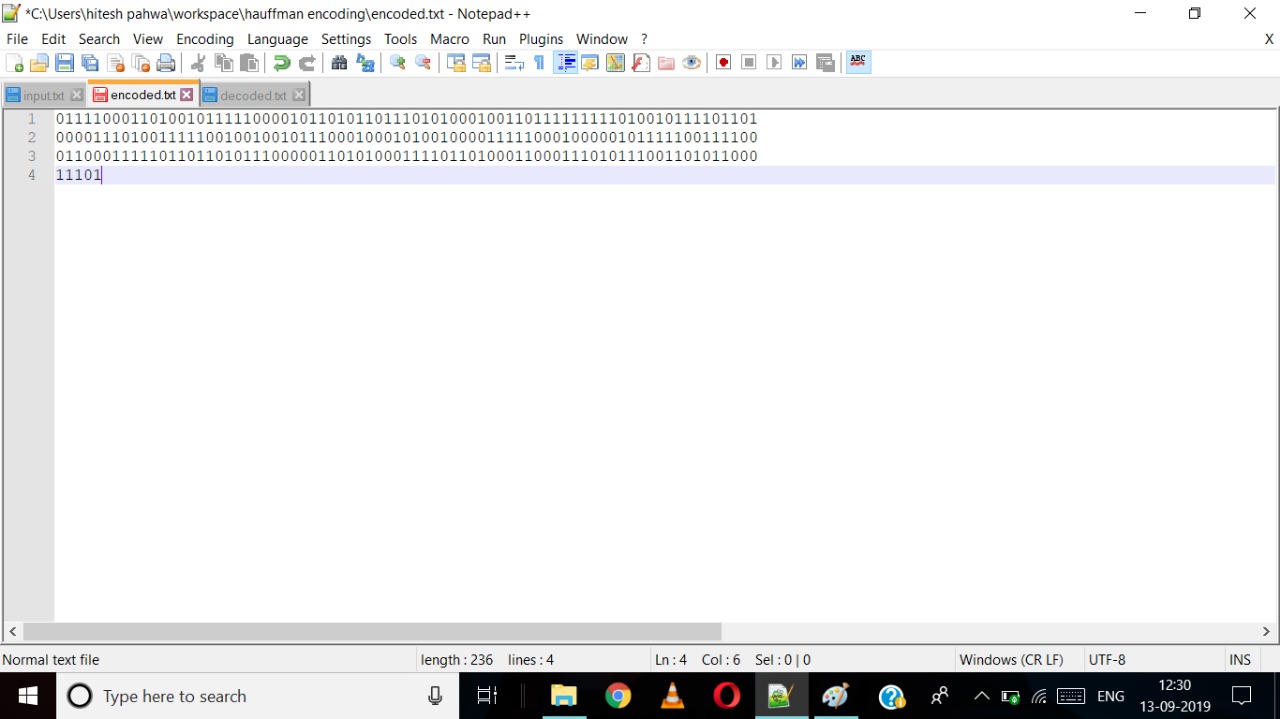
**Fig 4.5 console output 2**



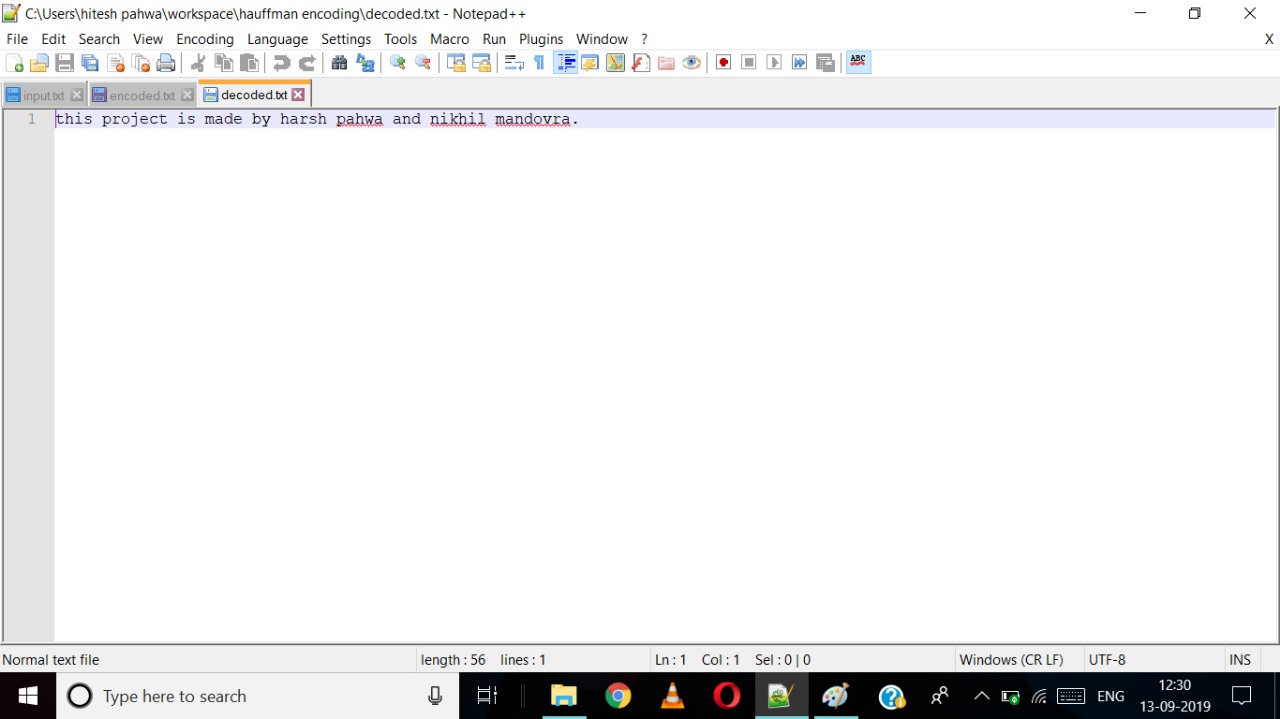
**Fig 4.6 console output 3**



**Fig 4.7 Input file**



**Fig 4.8 Encoded file**



**Fig 4.9 Decoded file**

**CHAPTER – 5**

**5.1 FUTURE SCOPE**

The scope of file compression will persist until there will be computer related technology in this world. And according to me, computers are never going to be evaded from this world because almost 95% of people in this world are somehow connected to technology and to implement latest technologies there is a strict need of computers among all of us.

As we all know that there is a limited source of space available in every electronic device, so to save the space and for optimal use of memory, file compression will always come into play and will save enormous space in a whole by compressing it in a proper way.

**5.2 CONCLUSION**

Well, the world always strive for betterment and improvement in every field, and into the field of Science there is a huge scope of improvement in primitive technologies so let’s hope that we might come up with a new and more improved compression system that compresses any file more efficiently and more space will be saved for other tasks.

**REFERENCES**

[1] Dr Naveen Garg, IIT-D on youtube.com, published on Sept 24, 2008 <https://www.youtube.com/watch?v=5wRPin4oxCo>

[2] Aashish Barnwal and GeeksforGeeks team, <https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/>

[3] Techie delight on techiedelight.com, published in 2017 <https://www.techiedelight.com/huffman-coding/>

[4]Huffman coding on Wikipedia.com, <https://en.wikipedia.org/wiki/Huffman_coding>