19AIE111

Data Structure and Algorithm - I

TEAM-7

HUFFMAN CODING ALGORITHM

1st year B. tech

CSE-AI

<u>Submitted by:</u>

Karthik D	**CB.EN.U4AIE19019**
Kaushik M	**CB.EN.U4AIE19036**
Prasanth S N	**CB.EN.U4AIE19046**
Shankar P	**CB.EN.U4AIE19049**
Vijai Simmon S	**CB.EN.U4AIE19068**

INTRODUCTION

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code proceeds by means of Huffman coding. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol . The algorithm derives this table from the estimated probability or frequency of occurrence for each possible value of the source symbol.

As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted. However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods it is replaced with arithmetic coding or asymmetric numeral systems if better compression ratio is required.

WORK BREAKDOWN STRUCTURE

The work breakdown structure of our project is as follows:

1) The TreeNode class is used to create the nodes was implemented by **Shankar P.**
2) The creation Huffman class with initialization of variables, constructor and finding the frequencies was done by **Vijai Simmon S.**
3) The heap and the creation of queue which is used to map character and its frequency was implemented by **Prasanth S N.**
4) The coding part for the purpose of encoding and decoding was done by **Kaushik M.**
5) The creation of test class, writing the encoded content in a file was done by **Karthik D.**

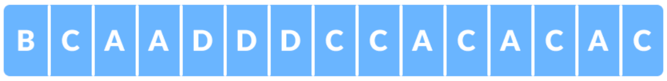
HUFFMAN CODING

Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.

Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

HOW DOES HUFFMAN CODING WORK

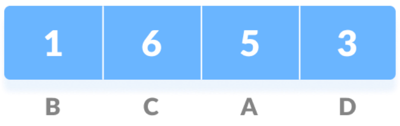
Suppose the string below is to be sent over a network.



Initial-String

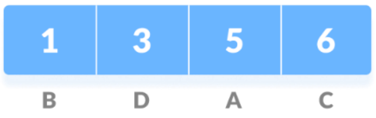
Each character occupies 8 bits. There is a total of 15 characters in the above string. Thus, a total of 8 * 15 = 120 bits is required to send this string.
Using the Huffman Coding technique, we can compress the string to a smaller size.
Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.
Once the data is encoded, it has to be decoded. Decoding is done using the same tree.
Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix code i.e. a code associated with a character should not be present in the prefix of any other code. The tree created above helps in maintaining the property.
Huffman coding is done with the help of the following steps.

1. Calculate the frequency of each character in the string.



Frequency of string

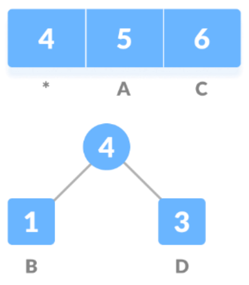
2. Sort the characters in increasing order of the frequency. These are stored in a queue Q.



Characters sorted according to the frequency

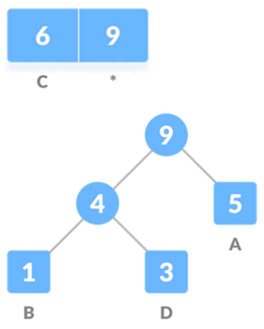
3. Make each unique character as a leaf node.

4. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.

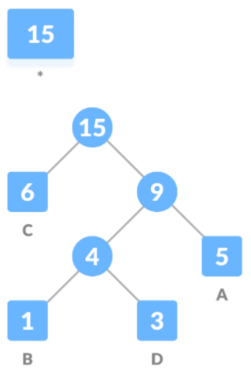


Getting the sum of the least numbers

5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure above).
6. Insert node z into the tree.
7. Repeat steps 3 to 5 for all the character

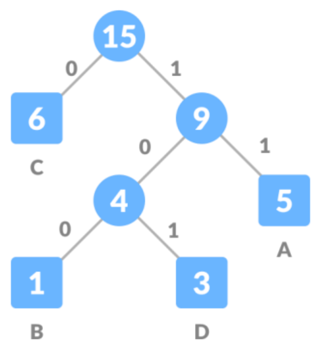


Repeat steps 3 to 5 for all the characters.



Repeat steps 3 to 5 for all characters.

8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



Assign 0 to the left edge and 1 to the right edge

For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
	15*8=120 bits		28 bits

So, finally we are able to compress the original data to a format in binary where we would be able to decode the same input string if we trace back the same code as stored in the output file,i.e corresponding to each binary code, we have a character and also the nodes are created based on the frequencies and thus we traverse through the output code. A '0' represents a left node and '1' represents a right node. So when we traverse throughout the code and if we reach a leaf node, then we append the character to the string corresponding to the code and thus we obtain the whole string back upon repetitive traversal through the tree based on the output encoded code.

IMPLEMENTATION IN ECLIPSE IDE

The TreeNode class is used to create nodes and thus construct a whole tree. It has 2 constructors-the first one is called when only frequency and character is known and the second constructor is invoked when frequency, character, left and right nodes are known.

The code for TreeNode class is given below:

```
package Project_DS1;
public class TreeNode {
	//All required variables are declared
	public char nodechar;
	static int size;
	int frequency;
	StringBuffer code;
	TreeNode left,right,root;
	//This constructor is used when we know only the character
and its frequency
	TreeNode(char ch,int fr){
		nodechar=ch;
		frequency=fr;
		size++;
		left = right = root = null;
	}
	//This constructor is used when we know the character and
its frequency along with right and left node
	TreeNode(char ch,int fr,TreeNode left,TreeNode right){
		nodechar=ch;
		frequency=fr;
		size++;
		this.left=left;
		this.right=right;
		root = this;
	}
}
```

The Huffman Tree class is the main implementation where we read the input file using the constructor. The letter_freq() deals with finding the frequency of each alphabet occurred in the text file. The makeQueue() is used to create the mapping for character and frequency based on heaps. The insertNode() creates a new node for each character. The buildTree() is used for creating the Huffman Tree. createCode() is where we create the binary string code for each character and store it in an output file. The printDecoded() deals with the output file and decodes the binary string back to corresponding character content.

The Huffman Tree Coding is given below:

```
package Project_DS1;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.nio.file.Files;
import java.nio.file.Paths;
public class Huffman {
     //All required variables are declared
     private String inputtext, outputtext;
     private byte[] byteinput;
     private int total_alphabets[], totalchars;
     public TreeNode tnodes[];
     private PrintWriter out;
     //Constructor to open the input file
     Huffman(String path){
          try {
     byteinput=Files.readAllBytes(Paths.get(path));//To take
everything that is in the text file
               inputtext=new
String(byteinput).replaceAll("[^a-z]", "");
          }
          catch (IOException e) {
               return;
          }
          outputtext = inputtext;
          total_alphabets = new int[26];//To store all
alphabets
     }
     //This function calculates the frequency of occurrence of
characters
     void letter_freq() {
          //iterating over the input text to find the
frequency of character in the input text
          for (char element : inputtext.toCharArray()) {
               try{
                    //maps the frequency to the corresponding
alphabet
                    total_alphabets[element - 'a']++;
               }
               catch (ArrayIndexOutOfBoundsException e){
                    return;
               }
          }
     }
     //this function is used to create a queue which helps in
building the tree
     void makeQueue(){
          for(int i=0;i<total_alphabets.length;i++){
               if(total_alphabets[i]!=0){
                    totalchars++;
               }
          }
          tnodes=new TreeNode[totalchars*2-1];
          for(int i=0;i<total_alphabets.length;i++){
               if(total_alphabets[i]!=0){
                    insertNode(new 
TreeNode((char)(i+'a'),total_alphabets[i]));
               }
          }
     }
     //This function helps in creating a node of a tree with 
max heap
     void insertNode(TreeNode node){
          int i, j;
          for(i = 0; i < TreeNode.size-1; i++){
               if(tnodes[i].frequency > node.frequency)
                    break;
          }
          for(j = TreeNode.size-2; j>=i; j--){
               tnodes[j+1]=tnodes[j];
          }
          tnodes[i] = node;
          maxHeapify(tnodes[tnodes.length-1]);
     }
     //Function to go to the Parent
     public TreeNode toParent(int pos) {
          return tnodes[(pos-1)/2];
     }
     //Making Max Heap based on node and length of tree node
     private void maxHeapify(TreeNode t1)
     {
          TreeNode r = t1;
          int pos = tnodes.length-1;
          while(r != null && pos != 0) {
               TreeNode p = toParent(pos);
               if(p.frequency <= r.frequency) {
                    TreeNode t = p;
                    p = r;
                    r = t;
               }
               r = p;
               pos = (pos-1)/2;
          }
     }
     //This function helps in building the complete tree
     void buildTree(){
          int looping = tnodes.length-3;
          //This iteration is used to insert the nodes of the 
tree that contains the character and the key
          for(int i = 0;i <= looping; i +=2){
               insertNode(new TreeNode('-',tnodes[i].frequency 
+ tnodes[i+1].frequency,tnodes[i],tnodes[i+1]));
          }
          // To display the character's corresponding
frequency and code like a table format.
          System.out.println("Character\tFrequency\tCode");
          createCode(tnodes[tnodes.length-1],new
StringBuffer());
     }
     //Here we create the binary codes based on movement from
the root
     void createCode(TreeNode node,StringBuffer codestring){
          //Moving on left from the parent we append 0
          if(node.left != null){
               codestring.append(0);
               createCode(node.left,codestring);
               codestring.deleteCharAt(codestring.length()-1);
          }
          //Moving on right from parent we append 1
          if(node.right != null){
               codestring.append(1);
               node.right.code=codestring;
               createCode(node.right,codestring);
               codestring.deleteCharAt(codestring.length()-1);
          }
          //If we reach a leaf node, we end the binary code
for the character and display it
          if(node.left == null&& node.right == null){
               node.code = codestring;
               outputtext =
outputtext.replaceAll(String.valueOf(node.nodechar),
codestring.toString());

     System.out.println(node.nodechar+"\t\t"+node.frequency+"\
t\t"+node.code);
          }}
     //This function is used to create an output file and
store the code for the characters.
     void makeOutputFile(String outputpath){
          try {
               out = new PrintWriter(new
FileWriter(outputpath));
               out.println(outputtext);
               System.out.println("Saved output in file " +
outputpath);
          } catch (IOException e) {
               System.out.println("cannot open/create file
:"+outputpath);
          }
          out.close();
     }
     //Function which is used to retrieve the data from the
encoded file and print the output
```

```
     void printDecoded(String outputfile, TreeNode node)
throws IOException {
          BufferedReader fileReader;
          fileReader = new BufferedReader(new
FileReader(outputfile));
          String s;
          String ans = "";
          TreeNode curr = node;
          boolean flag = true;
          while (flag == true)
          {
               s = fileReader.readLine();
               if (s == null)
                    flag = false;
               else {
                    flag = true;
                    for(int i = 0; i < s.length();i++) {
                         if(s.charAt(i)=='0')
                              curr = curr.left;
                         else if(s.charAt(i)=='1')
                              curr = curr.right;
                         if(curr.left == null && curr.right ==
null) {
                              ans = ans + curr.nodechar;
                              curr = node.root;
                         }
                    }
               }
          }
          fileReader.close();
          System.out.println(ans);
     }
     //Function is used to print the encoded text
     void printEncoded(){
          System.out.println("Output :\n" + outputtext);
     }

//Used to display Inorder traversal
     public void print(TreeNode r) {
          if(r != null)
          {
               print(r.left);
               if(r.frequency>0 && (r.nodechar >=97 &&
r.nodechar <=122))
                    System.out.print("Char:"+r.nodechar+"
,freq:"+r.frequency+";");
               print(r.right);

          }
          else
               return;
     }
}
```

The mainclass function is used to test the algorithm that we have coded. In this we have created an huffman object and giving a text file into it as input followed by calculating the frequency of each character. With the help of the frequency we make a queue with the help of maxheap function. Once it is created we began to build a tree and encode the data into an output file and followed by decoding it to ensure the result.

The code used for testing the algorithm is:

```
package Project_DS1;

import java.io.IOException;
//Testing class
public class MainClass {
     public static void main(String[] args) throws
IOException{
          //Create a Huffman Object
          Huffman h = new Huffman("C:\\Users\\Prasanth
.S.N\\Desktop\\DS1-Project\\input.txt");
          //We call the function to find frequency of each
character
          h.letter_freq();
          //Create the queue for enabling us to create a
MaxHeap
          h.makeQueue();
          //Building the tree based on the MaxHeap
          h.buildTree();
          //Create output file at specified path location
          h.makeOutputFile("C:\\Users\\Prasanth
.S.N\\Desktop\\DS1-Project\\output.txt");
          //Displaying the encoded code in the output
          h.printEncoded();
          //We retrieve the characters and display it calling
with the root.
          h.printDecoded("output.txt",
h.tnodes[h.tnodes.length-1]);
     }
```

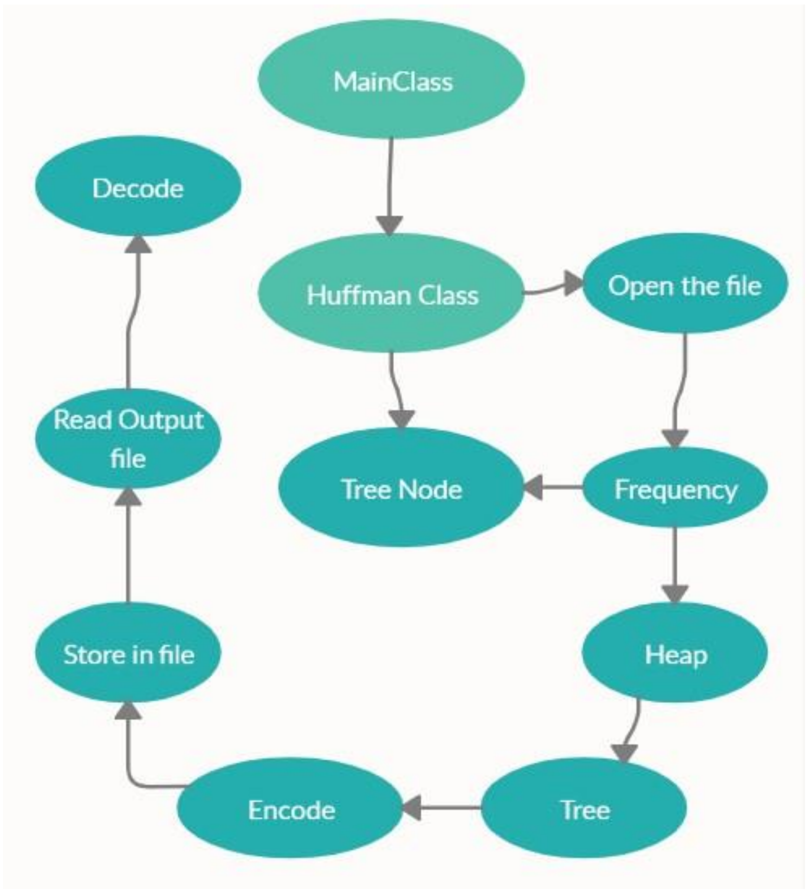
} WORKING OF THE ALGORITHM

The program works as:

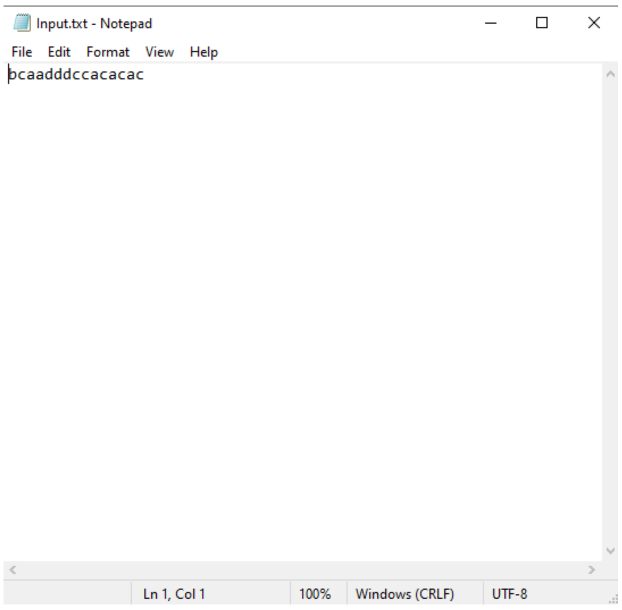
1. Read input file and calculate the count of all the characters filtering out the characters from `a' to `z' in lowercase, removing all other characters.
2. Create a TreeNode corresponding to each character and add it to sorted array.
3. Create the Huffman tree using the algorithm described above.

4. For each leaf in the resulting tree (corresponding to a character), calculate its codes as described above, as a String of 0s and 1s.
5. Finally, we decode the encoded file to retrieve back the original content.

<u>WORK FLOW DIAGRAM</u>

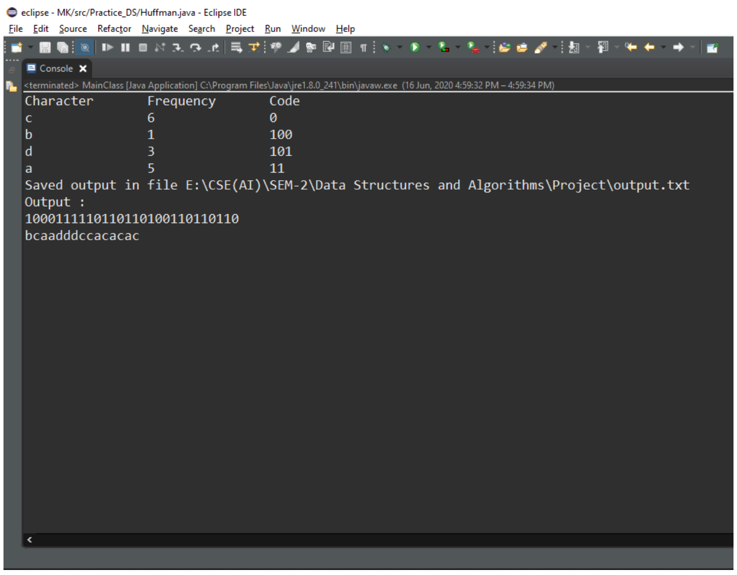


SCREENSHOTS



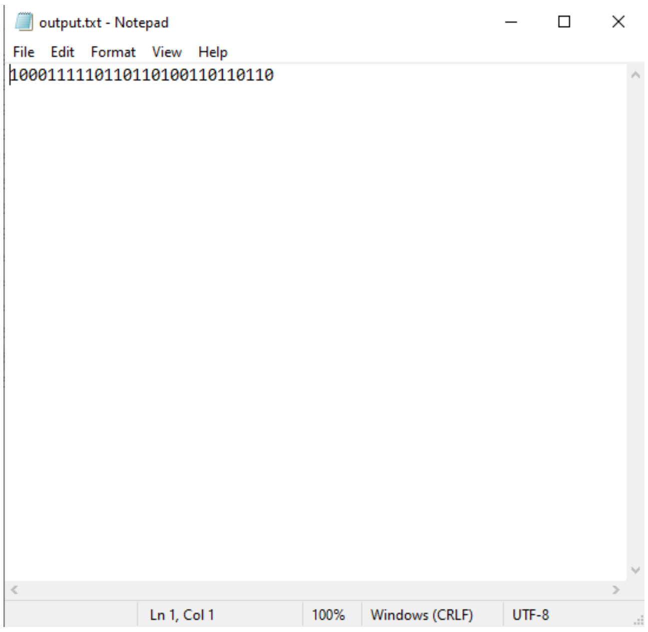
```
bcaadddccacacac
```

THE INPUT FILE


eclipse - MK/src/Practice_DS/Huffman.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Console

```
<terminated> MainClass [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (16 Jun, 2020 4:59:32 PM – 4:59:34 PM)
Character        Frequency        Code
c                6                0
b                1                100
d                3                101
a                5                11
Saved output in file E:\CSE(AI)\SEM-2\Data Structures and Algorithms\Project\output.txt
Output :
100011111011011010011011010
bcaadddccacacac
```

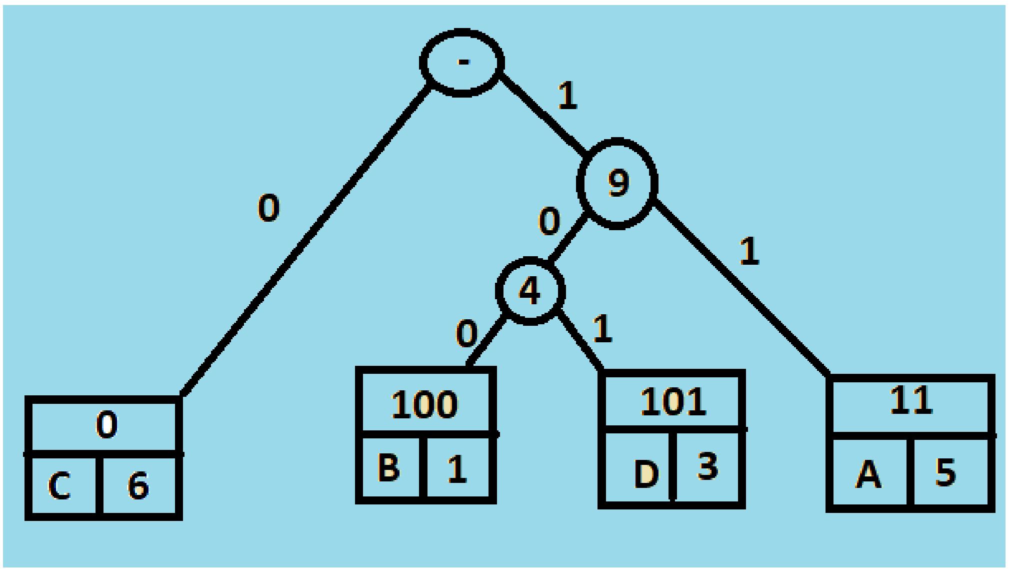
THE OUTPUT SHOWN IN THE CONSOLE.



```
1000111110110110100110110110
```

THE ENCODED DATA STORED IN THE FILE.

FINAL TREE STRUCTURE



PROPERITIES OF HUFFMAN CODE

- It is block code
- It is instantaneous
- It is uniquely decodable
- It is an optimal code

ADVANTAGES

- Produce a lossless compression of data
- Algorithm is easy to implement

DISADVANTAGES

- When a large number of symbols are to be coded, construction of Huffman code becomes very difficult.
- Cannot adapt to change in source statistics.
- Slower technique