Practical No -25

Aim:- Write a Program to Implement Huffman coding

Huffman Coding Overview

Huffman coding is a popular algorithm used for data compression. It is a lossless compression technique that assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters. This results in a more efficient representation of the data.

Key Concepts:

1. Frequency Table: Count the frequency of each character in the input data.
2. Priority Queue: Use a priority queue (or min-heap) to build a binary tree based on character frequencies.
3. Huffman Tree: Construct a binary tree where each leaf node represents a character and its frequency. The path from the root to a leaf node represents the binary code for that character.
4. Encoding: Generate binary codes for each character based on their position in the Huffman tree.
5. Decoding: Use the Huffman tree to decode the encoded data back to the original data.

Step-by-Step Explanation

1. Counting Character Frequencies

The first step in Huffman coding is to count the frequency of each character in the input string.

For the string **"huffman coding example"**, the frequency count might look like this:

| Character | Frequency | |-----------|-----------| | h | 1 | | u | 1 | | f | 2 | | m | 2 | | a | 2 | | n | 2 | | c | 1 | | o | 1 | | d | 1 | | i | 1 | | g | 1 | | e | 1 | | x | 1 | | l | 1 | | p | 1 | | space | 1 |

2. Building the Huffman Tree

Next, we build the Huffman tree using a priority queue (min-heap). Each character and its frequency are added as nodes to the priority queue.

- **Step 1**: Create nodes for each character based on their frequency.
- **Step 2**: While there is more than one node in the priority queue:
    - Remove the two nodes with the lowest frequency.
    - Create a new internal node with these two nodes as children and a frequency equal to the sum of their frequencies.
    - Add this new node back to the priority queue.

This process continues until there is only one node left in the priority queue, which becomes the root of the Huffman tree.

3. Generating Huffman Codes

Once the Huffman tree is built, we generate the Huffman codes for each character by traversing the tree:

- Assign a binary code of **0** for a left traversal and **1** for a right traversal.
- When a leaf node is reached, the path taken to reach that node represents the Huffman code for that character.

For example, the generated Huffman codes might look like this (the actual codes may vary based on the tree structure):

| Character | Huffman Code | |-----------|--------------| | h | 110 | | u | 1110 | | f | 10 | | m | 011 | | a | 010 | | n | 001 | | c | 11110 | | o | 11111 | | d | 0001 | | i | 0000 | | g | 00001 | | e | 0001 | | x | 00010 | | l | 00011 | | p | 000100 | | space | 000101 |

4. Encoding the Input Text

To encode the input text, we replace each character with its corresponding Huffman code. For example, the string **"huffman coding example"** would be encoded as a long binary string based on the Huffman codes generated.

5. Decoding the Encoded String

To decode the encoded string, we traverse the Huffman tree:

- Start from the root and read each bit of the encoded string.
- Move left for **0** and right for **1**.
- When a leaf node is reached, append the character to the decoded string and return to the root to continue decoding the next bits.

Program

```
import java.util.PriorityQueue;
```

```
import java.util.HashMap;
import java.util.Map;

class HuffmanNode implements Comparable<HuffmanNode>
{
    int frequency;
    char character;
    HuffmanNode left;
    HuffmanNode right;

    public HuffmanNode(char character, int frequency) {
        this.character = character;
        this.frequency = frequency;
        left = null;
        right = null;
    }

    @Override
    public int compareTo(HuffmanNode other) {
        return Integer.compare(this.frequency, other.frequency);
    }
}
```

```
class HuffmanCoding {
    private Map<Character, String> huffmanCodes = new HashMap<>();
    private HuffmanNode root;

    // Build the Huffman Tree
    public void buildHuffmanTree(String text) {
        // Count frequency of each character
        Map<Character, Integer> frequencyMap = new HashMap<>();
        for (char ch : text.toCharArray()) {
            frequencyMap.put(ch, frequencyMap.getOrDefault(ch, 0) + 1);
        }

        // Create a priority queue to hold the Huffman nodes
        PriorityQueue<HuffmanNode> priorityQueue = new PriorityQueue<>();
        for (Map.Entry<Character, Integer> entry : frequencyMap.entrySet()) {
            priorityQueue.add(new HuffmanNode(entry.getKey(), entry.getValue()));
        }
```

```
        // Build the Huffman tree
        while (priorityQueue.size() > 1) {
            HuffmanNode left = priorityQueue.poll();
            HuffmanNode right = priorityQueue.poll();
            HuffmanNode parent = new HuffmanNode('\0',
left.frequency + right.frequency);
            parent.left = left;
            parent.right = right;
            priorityQueue.add(parent);
        }

        root = priorityQueue.poll(); // The remaining node is the
root of the Huffman tree

        // Generate Huffman codes
        generateHuffmanCodes(root, "");
    }

    // Generate Huffman codes recursively
    private void generateHuffmanCodes(HuffmanNode node,
String code) {
        if (node == null) {
            return;
        }
```

```
        if (node.left == null && node.right == null) {
            huffmanCodes.put(node.character, code);
        }
        generateHuffmanCodes(node.left, code + "0");
        generateHuffmanCodes(node.right, code + "1");
    }

    // Encode the input text
    public String encode(String text) {
        StringBuilder encodedString = new StringBuilder();
        for (char ch : text.toCharArray()) {
            encodedString.append(huffmanCodes.get(ch));
        }
        return encodedString.toString();
    }

    // Decode the encoded string
    public String decode(String encodedString) {
        StringBuilder decodedString = new StringBuilder();
        HuffmanNode currentNode = root;
        for (char bit : encodedString.toCharArray()) {
            currentNode = (bit == '0') ? currentNode.left :
currentNode.right;
```

```
            if (currentNode.left == null && currentNode.right ==
null) {
                decodedString.append(currentNode.character);
                currentNode = root; // Go back to the root
            }
        }
        return decodedString.toString();
    }

    // Get the Huffman codes
    public Map<Character, String> getHuffmanCodes() {
        return huffmanCodes;
    }
}

public class HuffmanCodingExample {
    public static void main(String[] args) {
        String text = "huffman coding example";
        HuffmanCoding huffmanCoding = new
HuffmanCoding();

        // Build the Huffman Tree
        huffmanCoding.buildHuffmanTree(text);
```

```
        // Encode the text
        String encodedString = huffmanCoding.encode(text);
        System.out.println("Encoded String: " + encodedString);

        // Decode the encoded string
        String decodedString =
huffmanCoding.decode(encodedString);
        System.out.println("Decoded String: " + decodedString);

        // Display Huffman Codes
        System.out.println("Huffman Codes: " +
huffmanCoding.getHuffmanCodes());
    }
}
```

Output:-