CA6_832303206_陈竟镗

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



// Class representing a node in the Huffman tree
class HuffmanNode {
    int frequency; // Frequency of the character
    char character; // Character stored in the node (or '\0' for internal nodes)
    HuffmanNode left, right; // Left and right child nodes

    // Constructor for creating a Huffman node
    public HuffmanNode(char character, int frequency) {
        this.character = character;
        this.frequency = frequency;
    }
}

// Comparator for prioritizing nodes by frequency in the priority queue
class HuffmanComparator implements Comparator<HuffmanNode> {
    // Compares two nodes based on their frequencies

    @Override
    public int compare(HuffmanNode x, HuffmanNode y) {
        return x.frequency - y.frequency;
    }
}

public class HuffmanTemp {
    // Recursive function to store Huffman codes in a map
    public static void storeCodes(HuffmanNode root, String code, Map<Character,
String> huffmanCodes) {
        if (root == null) {
            return;
        }

        // Base case: if the node is a leaf, store the character and its code
        if (root.left == null && root.right == null) {
            huffmanCodes.put(root.character, code);
            return;
        }

        // Recursive call for the left child (append "0")
        storeCodes(root.left, code + "0", huffmanCodes);
        // Recursive call for the right child (append "1")
        storeCodes(root.right, code + "1", huffmanCodes);
    }

    // Recursive function to print Huffman codes from the tree
    public static void generateAndPrintCodes(HuffmanNode root, String code) {
```

```
        // Base case: if the node is a leaf, it contains a character
        if (root.left == null && root.right == null) {
            // Print the character and its corresponding code
            System.out.println("'" + root.character + "': " + code);
            return;
        }

        // Recursive call for the left child (append "0")
        generateAndPrintCodes(root.left, code + "0");
        // Recursive call for the right child (append "1")
        generateAndPrintCodes(root.right, code + "1");
    }

    // Recursive function to display the Huffman tree structure
    public static void displayTree(HuffmanNode root, String prefix, boolean
isLeft) {
        if (root == null) {
            return;
        }
        System.out.print(prefix);
        System.out.print(isLeft ? "├── " : "└── ");

        // Print node details: character (if leaf) and frequency
        if (root.left == null && root.right == null) {
            System.out.println("'" + root.character + "' [" + root.frequency +
"]");
        } else {
            System.out.println("Internal [" + root.frequency + "]");
        }

        // Recur for children, adjusting the prefix for visual indentation
        displayTree(root.left, prefix + (isLeft ? "|   " : "    "), true);
        displayTree(root.right, prefix + (isLeft ? "|   " : "    "), false);
    }


    public static void main(String[] args) {
        // As per the assignment, the test string is "Huffman is lossless"
        String input = "Huffman is lossless"; // Input string to encode

        // Step 1: Build Frequency Table
        // Use a HashMap to store characters and their frequencies.
        Map<Character, Integer> frequencyMap = new HashMap<>();
        for (char c : input.toCharArray()) {
            frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);
        }

        // Step 2: Initialize Priority Queue
        // A min-priority queue to store the nodes of the Huffman tree.
        // The custom comparator ensures nodes with lower frequencies have higher
priority.
        PriorityQueue<HuffmanNode> pq = new PriorityQueue<>(new
HuffmanComparator());

        // Create a leaf node for each character and add it to the priority
queue.
```

```
        for (Map.Entry<Character, Integer> entry : frequencyMap.entrySet()) {
            pq.add(new HuffmanNode(entry.getKey(), entry.getValue()));
        }

        // Store the root of the Huffman tree
        HuffmanNode root = null;

        // Step 3: Build Huffman Tree
        // Combine nodes until only one node (the root) remains in the queue.
        while (pq.size() > 1) {
            // Extract the two nodes with the minimum frequency from the queue.
            HuffmanNode left = pq.poll();
            HuffmanNode right = pq.poll();

            // Create a new internal node with a frequency equal to the sum of
the children's frequencies.
            // The character for internal nodes is set to a placeholder '\0'.
            HuffmanNode combinedNode = new HuffmanNode('\0', left.frequency +
right.frequency);
            combinedNode.left = left;
            combinedNode.right = right;

            // Add the new internal node back to the priority queue.
            pq.add(combinedNode);
        }
        // The last node in the queue is the root of the tree.
        root = pq.peek();


        // Step 4: Display the Huffman tree structure
        System.out.println("--- Huffman Tree ---");
        displayTree(root, "", false);
        System.out.println();


        // Step 5 & 6: Generate and Print Huffman codes for each character
        System.out.println("--- Huffman Codes ---");
        generateAndPrintCodes(root, "");
        System.out.println();


        Map<Character, String> huffmanCodes = new HashMap<>();
        storeCodes(root, "", huffmanCodes);

        System.out.println("--- Encoded String ---");
        StringBuilder encodedString = new StringBuilder();
        for (char c : input.toCharArray()) {
            encodedString.append(huffmanCodes.get(c));
        }


        System.out.println("Original String: " + input);
        System.out.println("Encoded (Binary) String: " +
encodedString.toString());
```

```
    }


}
```

```
--- Huffman Tree ---
└── Internal [19]
    ├── Internal [8]
    │   ├── Internal [4]
    │   │   ├── 'f' [2]
    │   │   └── Internal [2]
    │   │       ├── 'H' [1]
    │   │       └── 'o' [1]
    │   └── Internal [4]
    │       ├── Internal [2]
    │       │   ├── 'n' [1]
    │       │   └── 'i' [1]
    │       └── ' ' [2]
    └── Internal [11]
        ├── 's' [5]
        └── Internal [6]
            ├── Internal [2]
            │   ├── 'e' [1]
            │   └── 'm' [1]
            └── Internal [4]
                ├── 'l' [2]
                └── Internal [2]
                    ├── 'a' [1]
                    └── 'u' [1]
```

```
--- Huffman Codes ---
'f': 000
'H': 0010
'o': 0011
'n': 0100
'i': 0101
' ': 011
's': 10
'e': 1100
'm': 1101
'l': 1110
'a': 11110
'u': 11111

--- Encoded String ---
Original String: Huffman is lossless
Encoded (Binary) String: 0010111110000001101111100100011010110011111000111010111011001010
```