**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

# Experiment Number: 05

**Title:** Assignment Based on Greedy strategy. (Implement Huffman encoding algorithm)

|  |  |  |  |
| --- | --- | --- | --- |
| **Title of Experimentation** | **CO**  **Mapping** | **CO-Statements** | **PO**  **Mapping** |
| Assignment Based on Greedy strategy. (Implement Huffman encoding algorithm) | CO2, CO3 | CO2: To design efficient algorithms for computational problems using appropriate algorithmic paradigm  CO3: To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques | PO2, PO3, PO4 |

**Theory:**

* Greedy Strategy is a paradigm where local optimal choices are made at each step to find a global optimum.
* Huffman Encoding is a greedy algorithm that assigns variable-length binary codes to characters:
* Shorter codes → frequent characters.
* Longer codes → rare characters.
* Applications: Data compression in ZIP, JPEG, MP3, etc.
* **Time Complexity:**

1.Building priority queue: O(n)

2.Extract & merge steps: O(n log n)

Total: **O(n log n)**

**Input:**

Characters: {a, b, c, d, e, f}

Frequencies: {5, 9, 12, 13, 16, 45}

Huffman Codes (one possible solution):

a : 1100

b : 1101

c : 100

d : 101

e : 111

f : 0

Encoded String (for "face"):

f a c e → 0 1100 100 111 = 01100100111

**Output:**

Huffman Codes: {'f': '0', 'c': '100', 'd': '101', 'a': '1100', 'b': '1101', 'e': '111'}

Encoded: 01100100111

Decoded: face

**Objective of Experiment:**

* To implement Huffman Encoding using the Greedy strategy.
* To compress and decompress text data.
* To analyse space efficiency compared to fixed-length encoding

**Flow Chart/Pseudo Code/Algorithm:**

Algorithm:

1.Create a priority queue (min-heap) containing all characters with their frequencies.

2.While more than one node exists in the heap:

* Extract two nodes with the smallest frequency.
* Create a new internal node with these two as children.
* Insert the new node back into the heap.

3.The remaining node is the root of the Huffman tree.

4.Traverse the tree:

* Assign 0 for the left edge, 1 for the right edge.
* Generate codes for each character.

5.Encode the input string using generated codes.

6.Decode the encoded string using the Huffman tree.

**Flowchart:**

(You can insert a flowchart here showing recursive splitting and combining steps)

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

**Code:**

import java.util.\*;

class Node implements Comparable<Node> {

    char ch;

    int freq;

    Node left, right;

    Node(char ch, int freq) {

        this.ch = ch;

        this.freq = freq;

    }

    Node(int freq, Node left, Node right) {

        this.ch = '\0';

        this.freq = freq;

        this.left = left;

        this.right = right;

    }

    public int compareTo(Node other) {

        return this.freq - other.freq;

    }

}

public class HuffmanCoding {

    public static void generateCodes(Node root, String code, Map<Character, String> codes) {

        if (root == null)

            return;

        if (root.left == null && root.right == null) {

            codes.put(root.ch, code);

        }

        generateCodes(root.left, code + "0", codes);

        generateCodes(root.right, code + "1", codes);

    }

    public static Map<Character, String> buildHuffmanTree(char[] chars, int[] freq) {

        PriorityQueue<Node> pq = new PriorityQueue<>();

        for (int i = 0; i < chars.length; i++) {

            pq.add(new Node(chars[i], freq[i]));

        }

        while (pq.size() > 1) {

            Node left = pq.poll();

            Node right = pq.poll();

            Node merged = new Node(left.freq + right.freq, left, right);

            pq.add(merged);

        }

        Node root = pq.poll();

        Map<Character, String> codes = new HashMap<>();

        generateCodes(root, "", codes);

        return codes;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of characters:");

        int n = sc.nextInt();

        char[] chars = new char[n];

        int[] freq = new int[n];

        System.out.println("Enter characters:");

        for (int i = 0; i < n; i++) {

            chars[i] = sc.next().charAt(0);

        }

        System.out.println("Enter their frequencies:");

        for (int i = 0; i < n; i++) {

            freq[i] = sc.nextInt();

        }

        Map<Character, String> huffmanCodes = buildHuffmanTree(chars, freq);

        System.out.println("Huffman Codes:");

        for (char c : chars) {

            System.out.println(c + ": " + huffmanCodes.get(c));

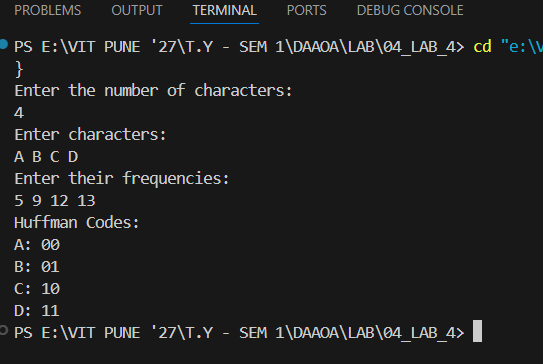
        }

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

## Time Complexity

* Building the priority queue takes O(n) for n characters.
* Each merge operation occurs n−1 times, and each operation involves extracting two min nodes and adding one new node, each costing O(logn) because of the priority queue.
* Total time complexity: **O(nlogn)**.

## Space Complexity

* The tree stores all nodes including merged internal nodes; a total of 2n−1 nodes for n characters.
* Codes are stored for each character.
* Priority queue space: O(n)
* Overall space complexity: **O(n)**

## Pseudocode with Complexity Comments

text

CLASS Node

VARIABLE ch, freq

VARIABLE left, right

FUNCTION Node(ch, freq) // Constructor for leaf node

this.ch ← ch // Time: +1

this.freq ← freq // Time: +1

ENDFUNCTION

FUNCTION Node(freq, left, right) // Constructor for internal node

this.ch ← '\0' // Time: +1

this.freq ← freq // Time: +1

this.left ← left // Time: +1

this.right ← right // Time: +1

ENDFUNCTION

FUNCTION compareTo(other) // For priority queue ordering

RETURN this.freq - other.freq // Time: +1

ENDFUNCTION

ENDCLASS

FUNCTION generateCodes(root, code, codes) // Generate Huffman codes by tree traversal

IF root == null // Time: +1 per call

RETURN

IF root.left == null AND root.right == null // Leaf node // Time: +1 per leaf

codes[root.ch] ← code // Store code for character // Time: +1

generateCodes(root.left, code + "0", codes) // Traverse left subtree // Time: O(n) overall

generateCodes(root.right, code + "1", codes) // Traverse right subtree // Time: O(n) overall

ENDFUNCTION

FUNCTION buildHuffmanTree(chars, freq) // chars, freq: arrays of size n

DECLARE priorityQueue pq // Space: +n for storing nodes

FOR i = 0 TO n-1 // Time: +n

pq.add(new Node(chars[i], freq[i])) // Time: O(log n) per insertion

WHILE pq.size() > 1 // Time: O(n log n) for building tree

left ← pq.poll() // Extract min node // Time: O(log n)

right ← pq.poll() // Extract next min // Time: O(log n)

merged ← new Node(left.freq + right.freq, left, right) // Create new node // Time: +1

pq.add(merged) // Insert merged node // Time: O(log n)

root ← pq.poll() // Root of Huffman Tree // Time: +1

DECLARE codes map // Space: +n for character codes

generateCodes(root, "", codes) // Time: O(n) to generate codes

RETURN codes // Map char → code // Time: +1

ENDFUNCTION

FUNCTION main

DECLARE scanner // Space: +1

PRINT "Enter the number of characters:" // Time: +1

INPUT n // Time: +1

DECLARE chars[n], freq[n] // Space: +2n

PRINT "Enter characters:" // Time: +1

FOR i = 0 TO n-1 // Time: +n

INPUT chars[i] // Time: +1 per input

PRINT "Enter their frequencies:" // Time: +1

FOR i = 0 TO n-1 // Time: +n

INPUT freq[i] // Time: +1 per input

huffmanCodes ← buildHuffmanTree(chars, freq) // Time: O(n log n), Space: O(n)

PRINT "Huffman Codes:" // Time: +1

FOR each c IN chars // Time: +n

PRINT c + ": " + huffmanCodes[c] // Time: +1 per output

CLOSE scanner // Time: +1

ENDFUNCTION