|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Pratik Pujari** | | |
| **UID no.** | **2020300054** | **Class:** | **Comps C Batch** |
| **Experiment No.** | 5 | | |

|  |  |
| --- | --- |
| **AIM:** | To implement Huffman encoding technique of greedy approach |
| **THEORY** | **What is Greedy Algorithm?**  A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph.  **Structure of a Greedy Algorithm**  Greedy algorithms take all of the data in a particular problem, and then set a rule for which elements to add to the solution at each step of the algorithm. In the animation above, the set of data is all of the numbers in the graph, and the rule was to select the largest number available at each level of the graph. The solution that the algorithm builds is the sum of all of those choices.  If both of the properties below are true, a greedy algorithm can be used to solve the problem.   * **Greedy choice property:** A global (overall) optimal solution can be reached by choosing the optimal choice at each step. * **Optimal substructure:** A problem has an optimal substructure if an optimal solution to the entire problem contains the optimal solutions to the sub-problems.   In other words, greedy algorithms work on problems for which it is true that, at every step, there is a choice that is optimal for the problem up to that step, and after the last step, the algorithm produces the optimal solution of the complete problem.  To make a greedy algorithm, identify an optimal substructure or subproblem in the problem. Then, determine what the solution will include (for example, the largest sum, the shortest path, etc.). Create some sort of iterative way to go through all of the subproblems and build a solution.  **Limitations of Greedy Algorithms**  Sometimes greedy algorithms fail to find the globally optimal solution because they do not consider all the data. The choice made by a greedy algorithm may depend on choices it has made so far, but it is not aware of future choices it could make.  **Huffman Coding**  Huffman encoding is another example of an algorithm where a greedy approach is successful. The Huffman algorithm analyzes a message and depending on the frequencies of the characters used in the message, it assigns a variable-length encoding for each symbol. A more commonly used symbol will have a shorter encoding while a rare symbol will have a longer encoding.  The Huffman coding algorithm takes in information about the frequencies or values of a particular symbol occurring. It begins to build the prefix tree from the bottom up, starting with the two least probable symbols in the list. It takes those symbols and forms a subtree containing them, and then removes the individual symbols from the list.  The algorithm sums the values of elements in a subtree and adds the subtree and its probability to the list. Next, the algorithm searches the list and selects the two symbols or subtrees with the smallest values. It uses those to make a new subtree, removes the original subtrees/symbols from the list, and then adds the new subtree and its combined probability to the list. This repeats until there is one tree and all elements have been added. At each subtree, the optimal encoding for each symbol is created and together composes the overall optimal encoding.  Huffman Encoding Example:  Suppose the string below is to be sent over a network.  string  Huffman coding is done with the help of the following steps.   1. Calculate the frequency of each character in the string   frequency of string   1. Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.   huffman coding   1. Make each unique character as a leaf node. 2. 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.   huffman coding   1. Remove these two minimum frequencies from Q and add the sum into the list of frequencies (\* denote the internal nodes in the figure above). 2. Insert node z into the tree. 3. Repeat steps 3 to 5 for all the characters   huffman coding     1. For each non-leaf node, 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.    **Decoding the code**  For decoding the code, we can take the code and traverse through the tree to find the character.  Let 101 is to be decoded, we can traverse from the root as in the figure below.  huffman coding |
| **PSEUDOCODE:** | **Huffman Algorithm**  1. Input:-Number of message with frequency count.  2. Output: - Huffman merge tree.  3. Begin  4. Let Q be the priority queue,  5. Q= {initialize priority queue with frequencies of all symbol or message}  6. Repeat n-1 times  7. Create a new node Z  8. X=extract\_min(Q)  9. Y=extract\_min(Q)  10. Frequency(Z) =Frequency(X) +Frequency(y);  11. Insert (Z, Q)  12. End repeat  13. Return (extract\_min(Q))  14. End. |
| **EXPERIMENT 1** | |
| **CODE:** | Huffman Code  import java.util.\*;  class HuffmanNode {  *// Huffman node class*      int data;      char character;  *// constructor*      HuffmanNode left, right;      HuffmanNode() {          this.left = null;          this.right = null;      }  *// constructor*      HuffmanNode(char ch, int data) {          this.left = null;          this.right = null;          this.data = data;          this.character = ch;      }  }  class MyComparator implements Comparator<HuffmanNode> {  *// Comparator class*      public int compare(HuffmanNode x, HuffmanNode y) {          return x.data - y.data;      }  }  public class Huffman {  *// Huffman class*      char charArray[];      int charFreq[];      int characters;  *// User input*      Scanner input = new Scanner(System.in);  *// Huffman tree*      PriorityQueue<HuffmanNode> queue;  *// hashmap to store the encodings*      HashMap<Character, String> map = new HashMap<Character, String>();  *// User input method*      public void userInput() {          System.out.print("\nEnter the number of characters to be read: ");          characters = input.nextInt();  *// Priority queue to store the Huffman nodes*          queue = new PriorityQueue<HuffmanNode>(characters, new MyComparator());          charArray = new char[characters];          charFreq = new int[characters];          System.out.print("\nEnter the characters below\n->");          for (int i = 0; i < characters; i++) {              charArray[i] = input.next().charAt(0);          }          System.out.print("\nEnter the Frequency of the Characters\n");          for (int i = 0; i < characters; i++) {              System.out.print("-> '" + charArray[i] + "' : ");              charFreq[i] = input.nextInt();          }      }      public void printArrays() {          System.out.println();          System.out.print("\n|   Characters\t|" + "   Frequency\t|\n");          System.out.print("-----------------------------------\n");          for (int i = 0; i < characters; i++) {              System.out.print("\n|        " + charArray[i] + "\t|" + "          " + charFreq[i] + "\t|");          }      }      public void setup(HashMap<Character, Integer> values) {  *// setup method*          characters = values.size();          charArray = new char[characters];          charFreq = new int[characters];          queue = new PriorityQueue<HuffmanNode>(characters, new MyComparator());          for (int i = 0; i < values.size(); i++) {              charArray[i] = (char) values.keySet().toArray()[i];              charFreq[i] = (int) values.values().toArray()[i];          }          printArrays();      }      public HuffmanNode makeTree() {  *// makeTree method*          System.out.print("\nStarted Making the Huffman Tree\n");          for (int i = 0; i < characters; i++) {  *// Adding the nodes to the queue*              HuffmanNode hNode = new HuffmanNode(charArray[i], charFreq[i]);              queue.add(hNode);          }          HuffmanNode root = null;          int counter = 1;  *// Making the tree*          while (queue.size() > 1) {              System.out.print("\n-----------------------------------\n");              System.out.print("\nStep " + counter);              HuffmanNode x = queue.peek();              queue.poll();              HuffmanNode y = queue.peek();              queue.poll();              HuffmanNode f = new HuffmanNode();  *// Combining the nodes*              f.data = x.data + y.data;              f.character = '+';  *// Adding the combined node to the queue*              f.left = x;              f.right = y;              root = f;              printNode(root, x, y);              queue.add(f);              counter++;          }          return root;      }      public void printNode(HuffmanNode root, HuffmanNode x, HuffmanNode y) {          System.out.print("\n\nParent Node: " + " " + root.data);          System.out.print("\n| |");          System.out.printf("\n| \\\_Left Child:\t|   " + x.character + "\t |   " + x.data + " \t|");          System.out.print("\n|");          System.out.printf("\n\\\_Right Child:\t|   " + y.character + "\t |   " + y.data + " \t|");      }      public void printTree(HuffmanNode root, String characters) {  *// System.out.println(root+" "+characters);*          if (root.left == null && root.right == null && Character.isLetter(root.character)) {              map.put(root.character, characters);              return;          }  *// Printing the left child*          printTree(root.left, characters + "0");          printTree(root.right, characters + "1");      }      public void displayMap() {  *// Displaying the map*          System.out.println("\n\nThe Encoding is: ");          for (Map.Entry m : map.entrySet())              System.out.println(m.getKey() + " " + m.getValue());      }  }  Driver Code  import java.util.HashMap;  public class Driver {      public static void main(String[] args) {          Huffman HM = new Huffman();  *// user INput*          System.out.print("\n1.Character Input\n2.String Input\nEnter your choice: ");          int choice = HM.input.nextInt();          switch (choice) {              case 1:  *// character input*                  HM.userInput();                  HM.printArrays();                  HuffmanNode root1 = HM.makeTree();                  HM.printTree(root1, "");                  HM.displayMap();                  break;              case 2:  *// string input*                  System.out.print("\nEnter the String to be encoded(without space)");                  System.out.print("\n->");                  String str = HM.input.next();                  HashMap<Character, Integer> chars = new HashMap<Character, Integer>();                  for (int i = 0; i < str.length(); i++) {  *// System.out.print("\n" + str.charAt(i));*                      if (Character.isLetter(str.toLowerCase().charAt(i)) && !chars.containsKey(str.charAt(i))) {                          chars.put(str.toLowerCase().charAt(i), 1);                      } else if (chars.containsKey(str.charAt(i))) {                          int value = chars.get(str.charAt(i));                          value++;                          chars.replace(str.charAt(i), value);                      }                  }                  System.out.print("\nAll the Characters in the String are: ");                  HM.setup(chars);                  HuffmanNode root2 = HM.makeTree();                  HM.printTree(root2, "");                  HM.displayMap();                  break;              default:                  break;          }  *// HM.printQueue();*      }  } |
| **OUTPUT:** | From the above example   * String is read and character is categorized according to the frequency * Priority queue (min heap) is used to store the Huffman node * Huffman tree is created using the queue * Every node is printed with its child data * In-order transversal through the tree goes through each node and assigns the encoding to it. |
| **TIME COMPLEXITY:** | Operation of the Huffman algorithm.  The time complexity of the Huffman algorithm is **O(nlogn)**. Using a heap to store the weight of each tree, each iteration requires **O(logn)** time to determine the cheapest weight and insert the new weight. There are **O(n)** iterations, one for each item. |
| **RESULT:** Things learnt during the procedural programming of the question   * Learnt about the Huffman encoding and decoding * Learnt on how to store Huffman node priority queue (min heap) * Learnt about the comparator class and used it in priority queue to find the min element * Used priority queue which acts as a min heap to find the minimum most element in the heap * Used HashMap to store the characters and frequency present in the string. | |