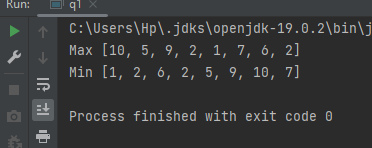
Lab10 22k4818

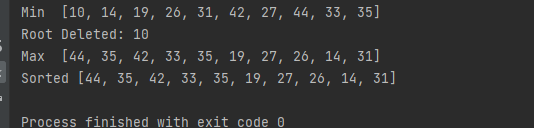
Q1

import java.util.Arrays;   
  
public class q1 {  
 public static void main(String[] args) {  
 int[] inputArray = {7, 1, 6, 2, 5, 9, 10, 2};  
  
  
 int[] maxHeap = *MaxHeap*(inputArray.clone());  
 System.*out*.println("Max " + Arrays.*toString*(maxHeap));  
 int[] minHeap = *MinHeap*(inputArray.clone());  
 System.*out*.println("Min " + Arrays.*toString*(minHeap));  
  
  
  
 }  
  
 public static int[] MinHeap(int[] arr) {  
 int n = arr.length;  
  
 for (int i = (n / 2) - 1; i >= 0; i--) {  
 *heapifyMin*(arr, n, i);  
 }  
  
 return arr;  
 }  
   
 public static void heapifyMax(int[] arr, int n, int i) {  
 int largest = i;  
 int leftChild = 2 \* i + 1;  
 int rightChild = 2 \* i + 2;  
  
 if (leftChild < n && arr[leftChild] > arr[largest]) {  
 largest = leftChild;  
 }  
  
 if (rightChild < n && arr[rightChild] > arr[largest]) {  
 largest = rightChild;  
 }  
  
 if (largest != i) {  
 int temp = arr[i];  
 arr[i] = arr[largest];  
 arr[largest] = temp;  
  
 *heapifyMax*(arr, n, largest);  
 }  
 }  
 private static void heapifyMin(int[] a, int n, int i) {  
 int smallest = i;  
 int left= 2 \* i + 1;  
 int rightC = 2 \* i + 2;  
  
 if (left < n && a[left] < a[smallest]) {  
 smallest = left;  
 }  
  
 if (rightC < n && a[rightC] < a[smallest]) {  
 smallest = rightC;  
 }  
  
 if (smallest != i) {  
 int temp = a[i];  
 a[i] = a[smallest];  
 a[smallest] = temp;  
  
 *heapifyMin*(a, n, smallest);  
 }  
 }  
 public static int[] MaxHeap(int[] arr) {  
 int n = arr.length;  
  
 for (int i = (n / 2) - 1; i >= 0; i--) {  
 *heapifyMax*(arr, n, i);  
 }  
  
 return arr;  
 }  
   
  
  
  
}



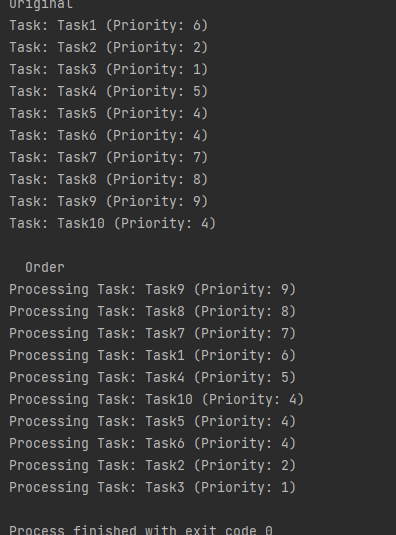
Q2

import java.util.Arrays;  
  
public class q2 {   
 public static void main(String[] args) {  
 int[] inputArray = {35, 33, 42, 10, 14, 19, 27, 44, 26, 31};  
   
  
 *Min*(inputArray);  
 System.*out*.println("Min " + Arrays.*toString*(inputArray));  
  
 int rootValue = *deleteRoot*(inputArray);  
 System.*out*.println("Root Deleted: " + rootValue);  
  
 *Max*(inputArray);  
 System.*out*.println("Max " + Arrays.*toString*(inputArray));  
  
 System.*out*.println("Sorted " + Arrays.*toString*(inputArray));  
 }  
  
 public static void Max(int[] a) {  
 int n = a.length;  
  
 for (int i = (n / 2) - 1; i >= 0; i--) {  
 *heapifyMax*(a, n, i);  
 }  
 }  
 public static void Min(int[] a) {  
 int n = a.length;  
  
 for (int i = (n / 2) - 1; i >= 0; i--) {  
 *heapifyMin*(a, n, i);  
 }  
 }  
   
 public static void heapifyMin(int[] a, int n, int i) {  
 int small = i;  
 int leftC = 2 \* i + 1;  
 int rightC = 2 \* i + 2;  
  
 if (leftC < n && a[leftC] < a[small]) {  
 small = leftC;  
 }  
  
 if (rightC < n && a[rightC] < a[small]) {  
 small = rightC;  
 }  
  
 if (small != i) {  
 int temp = a[i];  
 a[i] = a[small];  
 a[small] = temp;  
  
 *heapifyMin*(a, n, small);  
 }  
 }  
  
 public static int deleteRoot(int[] a) {  
 int n = a.length;  
  
 if (n == 0) {  
 throw new IllegalStateException("Cannot delete from an empty heap");  
 }  
  
 int rootValue = a[0];  
 a[0] = a[n - 1];  
 a = Arrays.*copyOf*(a, n - 1);  
  
 *heapifyMin*(a, n - 1, 0);  
  
 return rootValue;  
 }  
  
  
  
 private static void heapifyMax(int[] a, int n, int i) {  
 int large = i;  
 int leftC = 2 \* i + 1;  
 int rightC = 2 \* i + 2;  
  
 if (leftC < n && a[leftC] > a[large]) {  
 large = leftC;  
 }  
  
 if (rightC < n && a[rightC] > a[large]) {  
 large = rightC;  
 }  
  
 if (large != i) {  
 int temp = a[i];  
 a[i] = a[large];  
 a[large] = temp;  
  
 *heapifyMax*(a, n, large);  
 }  
 }  
   
  
}



Q3

import java.util.\*;   
  
class Task {  
 int priority;  
 String name;  
  
 public Task(String name, int priority) {  
 this.name = name;  
 this.priority = priority;  
 }  
}  
public class q3 {  
  
   
  
 private static List<Task> RandomTasks(int n) {  
 List<Task> tasks = new ArrayList<>();  
 Random random = new Random();  
  
 for (int i = 1; i <= n; i++) {  
 tasks.add(new Task("Task" + i, random.nextInt(10) + 1));  
 }  
  
 return tasks;  
 }  
  
  
 public static void main(String[] args) {  
 List<Task> tasks = *RandomTasks*(10);  
  
  
 System.*out*.println("Original");  
 *display*(tasks);  
  
  
 PriorityQueue<Task> priorityQueue = new PriorityQueue<>(Comparator.*comparingInt*(task -> -task.priority));  
 priorityQueue.addAll(tasks);  
  
  
 System.*out*.println("\n Order ");  
 while (!priorityQueue.isEmpty()) {  
 Task currentTask = priorityQueue.poll();  
 System.*out*.println("Processing Task: " + currentTask.name + " (Priority: " + currentTask.priority + ")");  
 }  
 }  
 public static void display(List<Task> tasks) {  
 for (Task task : tasks) {  
 System.*out*.println("Task: " + task.name + " (Priority: " + task.priority + ")");  
 }  
 }  
}



Q4

import java.util.HashMap;  
import java.util.Map;  
import java.util.PriorityQueue;   
  
  
  
public class q4 {   
 public static void main(String[] args) {  
 String inputString = "BSE-3B";  
  
 Map<Character, String> huffmanCodes = *encode*(inputString);  
  
 System.*out*.println("Original " + inputString);  
 System.*out*.println("Encoded " + *encodeString*(inputString, huffmanCodes));  
 }  
  
 public static Map<Character, String> encode(String inputString) {  
 Map<Character, Integer> frequencyMap = new HashMap<>();  
  
  
 for (char c : inputString.toCharArray()) {  
 frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);  
 }  
  
  
 PriorityQueue<HuffmanNode> priorityQueue = new PriorityQueue<>();  
 for (Map.Entry<Character, Integer> entry : frequencyMap.entrySet()) {  
 priorityQueue.add(new HuffmanNode(entry.getKey(), entry.getValue()));  
 }  
  
  
 while (priorityQueue.size() > 1) {  
 HuffmanNode left = priorityQueue.poll();  
 HuffmanNode right = priorityQueue.poll();  
  
 HuffmanNode internalNode = new HuffmanNode('\0', left.frequency + right.frequency);  
 internalNode.left = left;  
 internalNode.right = right;  
  
 priorityQueue.add(internalNode);  
 }  
  
  
 HuffmanNode root = priorityQueue.poll();  
 Map<Character, String> huffmanCodes = new HashMap<>();  
 *generateCodes*(root, "", huffmanCodes);  
  
 return huffmanCodes;  
 }  
 public static String encodeString(String inputString, Map<Character, String> huffmanCodes) {  
 StringBuilder encodedString = new StringBuilder();  
 for (char c : inputString.toCharArray()) {  
 encodedString.append(huffmanCodes.get(c));  
 }  
 return encodedString.toString();  
 }  
  
 public static void generateCodes(HuffmanNode root, String code, Map<Character, String> huffmanCodes) {  
 if (root != null) {  
 if (root.data != '\0') {  
 huffmanCodes.put(root.data, code);  
 }  
 *generateCodes*(root.left, code + "0", huffmanCodes);  
 *generateCodes*(root.right, code + "1", huffmanCodes);  
 }  
 }  
  
  
  
   
  
}  
  
class HuffmanNode implements Comparable<HuffmanNode> {  
 char data;  
 int frequency;  
 HuffmanNode left, right;  
  
 public HuffmanNode(char data, int frequency) {  
 this.data = data;  
 this.frequency = frequency;  
 left = right = null;  
 }  
  
 @Override  
 public int compareTo(HuffmanNode node) {  
 return this.frequency - node.frequency;  
 }  
}



Q5

import java.util.HashMap;  
import java.util.Map;  
import java.util.PriorityQueue;  
  
  
  
public class q5 {   
  
 private static String encodeString(String inputString, Map<Character, String> huffmanCodes) {  
 StringBuilder encodedString = new StringBuilder();  
 for (char c : inputString.toCharArray()) {  
 encodedString.append(huffmanCodes.get(c));  
 }  
 return encodedString.toString();  
 }  
 public static void main(String[] args) {  
 String inputString = "BSE-3B";  
  
 Map<Character, String> huffmanCodes = *encode*(inputString);  
 String encodedString = *encodeString*(inputString, huffmanCodes);  
  
 System.*out*.println("Original : " + inputString);  
 System.*out*.println("Encoded " + encodedString);  
  
 String decodedString = *decode*(encodedString, huffmanCodes);  
 System.*out*.println("Decoded " + decodedString);  
 }  
  
 private static Map<Character, String> encode(String inputString) {  
 Map<Character, Integer> frequencyMap = new HashMap<>();  
  
 for (char c : inputString.toCharArray()) {  
 frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);  
 }  
  
   
 PriorityQueue<HuffmanNode> priorityQueue = new PriorityQueue<>();  
 for (Map.Entry<Character, Integer> entry : frequencyMap.entrySet()) {  
 priorityQueue.add(new HuffmanNode(entry.getKey(), entry.getValue()));  
 }  
  
 while (priorityQueue.size() > 1) {  
 HuffmanNode left = priorityQueue.poll();  
 HuffmanNode right = priorityQueue.poll();  
  
 HuffmanNode internalNode = new HuffmanNode('\0', left.frequency + right.frequency);  
 internalNode.left = left;  
 internalNode.right = right;  
  
 priorityQueue.add(internalNode);  
 }  
  
 // Traverse the Huffman Tree and generate codes  
 HuffmanNode root = priorityQueue.poll();  
 Map<Character, String> huffmanCodes = new HashMap<>();  
 *generateCodes*(root, "", huffmanCodes);  
  
 return huffmanCodes;  
 }  
  
 private static void generateCodes(HuffmanNode root, String code, Map<Character, String> huffmanCodes) {  
 if (root != null) {  
 if (root.data != '\0') {  
 huffmanCodes.put(root.data, code);  
 }  
 *generateCodes*(root.left, code + "0", huffmanCodes);  
 *generateCodes*(root.right, code + "1", huffmanCodes);  
 }  
 }  
  
  
  
 private static String decode(String encodedString, Map<Character, String> huffmanCodes) {  
 StringBuilder decodedString = new StringBuilder();  
 int index = 0;  
  
 while (index < encodedString.length()) {  
 HuffmanNode current = *findDecodedCharacter*(encodedString, index, huffmanCodes);  
 decodedString.append(current.data);  
 index += huffmanCodes.get(current.data).length();  
 }  
  
 return decodedString.toString();  
 }  
  
 private static HuffmanNode findDecodedCharacter(String encodedString, int index, Map<Character, String> huffmanCodes) {  
 HuffmanNode root = new HuffmanNode('\0', 0);  
 HuffmanNode current = root;  
  
 while (index < encodedString.length()) {  
 char bit = encodedString.charAt(index);  
 if (bit == '0') {  
 if (current.left == null) {  
 current.left = new HuffmanNode('\0', 0);  
 }  
 current = current.left;  
 } else {  
 if (current.right == null) {  
 current.right = new HuffmanNode('\0', 0);  
 }  
 current = current.right;  
 }  
  
 index++;  
  
 if (huffmanCodes.containsKey(current.data)) {  
 break;  
 }  
 }  
  
 return current;  
 }  
}  
  
class HuffmanNode implements Comparable<HuffmanNode> {  
 char data;  
 int frequency;  
 HuffmanNode left, right;  
  
 public HuffmanNode(char data, int frequency) {  
 this.data = data;  
 this.frequency = frequency;  
 left = right = null;  
 }  
  
 @Override  
 public int compareTo(HuffmanNode node) {  
 return this.frequency - node.frequency;  
 }  
}

