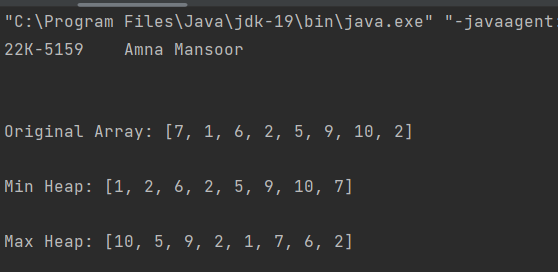
**22K-5159 Amna Mansoor BSE-3B LAB#10**

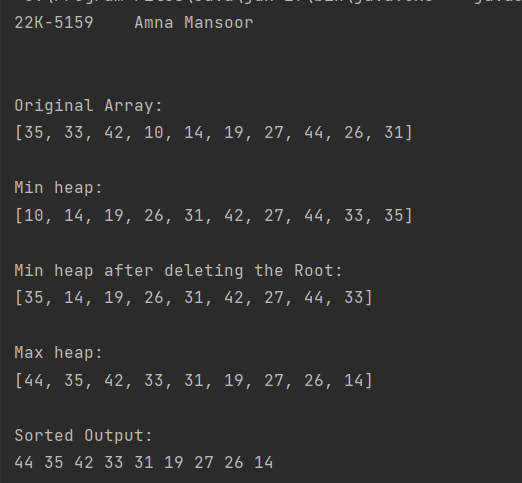
**Task 1:**

import java.util.Arrays;  
  
public class Task1 {  
 public static void main(String[] args) {  
 System.*out*.println("22K-5159 Amna Mansoor\n");  
 int[] arr={7,1,6,2,5,9,10,2};  
 System.*out*.println("\nOriginal Array: "+ Arrays.*toString*(arr));  
  
 int[] minHeap=*MinHeap*(arr.clone());  
 System.*out*.println("\nMin Heap: "+Arrays.*toString*(minHeap));  
  
 int[] maxHeap=*MaxHeap*(arr.clone());  
 System.*out*.println("\nMax Heap: "+Arrays.*toString*(maxHeap));  
  
  
 }  
 public static int[] MinHeap(int[] arr){  
 int n=arr.length;  
 for (int i=n/2-1;i>=0;i--){  
 *minHeapify*(arr,n,i);  
 }  
 return arr;  
 }  
 public static int[] MaxHeap(int[] arr){  
 int n=arr.length;  
 for (int i=n/2-1;i>=0;i--){  
 *maxHeapify*(arr,n,i);  
 }  
 return arr;  
 }  
  
  
 private static void minHeapify(int[] arr, int n, int i) {  
 int smallest=i;  
 int left=2\*i+1;  
 int right=2\*i+2;  
  
 if (left<n && arr[left]<arr[smallest]){  
 smallest=left;  
 }  
 if (right<n && arr[right]<arr[smallest]){  
 smallest=right;  
 }  
 if (smallest!=i){  
 int temp=arr[i];  
 arr[i]=arr[smallest];  
 arr[smallest]=temp;  
 *minHeapify*(arr, n, smallest);  
 }  
 }  
  
 private static void maxHeapify(int[] arr, int n, int i) {  
 int largest=i;  
 int left=2\*i+1;  
 int right=2\*i+2;  
  
 if (left<n && arr[left]>arr[largest]){  
 largest=left;  
 }  
 if (right<n && arr[right]>arr[largest]){  
 largest=right;  
 }  
 if (largest!=i){  
 int temp=arr[i];  
 arr[i]=arr[largest];  
 arr[largest]=temp;  
 *maxHeapify*(arr, n, largest);  
 }  
 }  
  
  
}

****

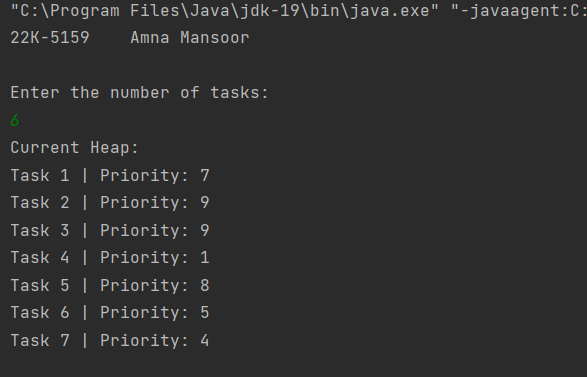
**Task 2:**

import java.util.Arrays;  
  
public class Task2 {  
 public static void main(String[] args) {  
 System.*out*.println("22K-5159 Amna Mansoor\n");  
 int[] arr={35,33,42,10,14,19,27,44,26,31};  
 System.*out*.println("\nOriginal Array: \n"+Arrays.*toString*(arr));  
  
 //creating minHeap  
 int[] minHeap=*MinHeap*(arr.clone());  
 System.*out*.println("\nMin heap: \n"+Arrays.*toString*(minHeap));  
  
 //deleting root of min heap  
 int[] DelRoot=*deleteRoot*(minHeap);  
 System.*out*.println("\nMin heap after deleting the Root: \n"+Arrays.*toString*(DelRoot));  
  
 //rebalance the tree to max heap  
 int[] maxHeap=*MaxHeap*(DelRoot.clone());  
 System.*out*.println("\nMax heap: \n"+Arrays.*toString*(maxHeap));  
  
 //printing the sorted output  
 System.*out*.println("\nSorted Output: ");  
 *printSorted*(maxHeap);  
  
 }  
  
  
 public static int[] MinHeap(int[] arr){  
 int n= arr.length;  
 for (int i=n/2-1;i>=0;i--){  
 *minHeapify*(arr,n,i);  
 }  
 return arr;  
 }  
 private static void minHeapify(int[] arr, int n, int i) {  
 int smallest=i;  
 int left=2\*i+1;  
 int right=2\*i+2;  
  
 if (left<n && arr[left]<arr[smallest]){  
 smallest=left;  
 }  
 if (right<n && arr[right]<arr[smallest]){  
 smallest=right;  
 }  
 if (smallest!=i){  
 int temp=arr[i];  
 arr[i]=arr[smallest];  
 arr[smallest]=temp;  
 *minHeapify*(arr, n, smallest);  
 }  
 }  
 public static int[] MaxHeap(int[] arr){  
 int n=arr.length;  
 for (int i=n/2-1;i>=0;i--){  
 *maxHeapify*(arr,n,i);  
 }  
 return arr;  
 }  
 private static void maxHeapify(int[] arr, int n, int i) {  
 int largest=i;  
 int left=2\*i+1;  
 int right=2\*i+2;  
  
 if (left<n && arr[left]>arr[largest]){  
 largest=left;  
 }  
 if (right<n && arr[right]>arr[largest]){  
 largest=right;  
 }  
 if (largest!=i){  
 int temp=arr[i];  
 arr[i]=arr[largest];  
 arr[largest]=temp;  
 *maxHeapify*(arr, n, largest);  
 }  
 }  
  
 public static int[] deleteRoot(int[] minHeap){  
 int n= minHeap.length;  
 if(n<=1){  
 return new int[0];  
 }  
 minHeap[0]=minHeap[n-1];  
 return Arrays.*copyOf*(minHeap,n-1);  
 }  
  
 private static void printSorted(int[] maxHeap) {  
 //since max heap is already sorted in descending order  
 for(int num:maxHeap){  
 System.*out*.print(num+" ");  
 }  
 System.*out*.println();  
 }  
}

****

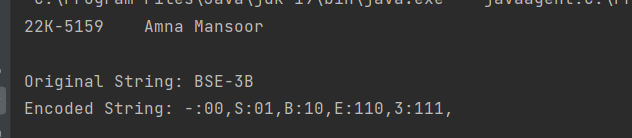
**Task 3:**

import java.util.Scanner;  
  
public class Task3 {  
 public static void main(String[] args) {  
 System.*out*.println("22K-5159 Amna Mansoor\n");  
 Scanner sc=new Scanner(System.*in*);  
 System.*out*.println("Enter the number of tasks: ");  
 int tasksNum=sc.nextInt();  
  
 Tasks taskHeap=new Tasks(tasksNum);  
 taskHeap.Display();  
  
 System.*out*.println("Heapifying the priority queue ->");  
 taskHeap.MaxHeap();  
 System.*out*.println("\nThe order is: \n");  
 taskHeap.Display();  
  
 for (int i=0;i<tasksNum;i++){  
 taskHeap.delMaxPriorityTask();  
 System.*out*.println("After Deleting, the order is: ");  
 taskHeap.Display();  
 }  
  
 }  
  
  
}  
class Tasks{ //inner class representation  
 class task{  
 static int *taskNum*=1;  
 int number;  
 int priority;  
 public task(){  
 int randomPriority=(int)(Math.*random*()\*10)+1;  
 number=*taskNum*;  
 priority=randomPriority;  
 *taskNum*++;  
 }  
 }  
  
 task[] tasks;  
 int heapSize;  
 public Tasks(int capacity){  
 tasks=new task[capacity+2];  
 for (int i=1;i< tasks.length;i++){  
 tasks[i]=new task();  
 }  
 heapSize=tasks.length-1;  
 }  
  
 public void MaxHeap(){  
 int n=heapSize;  
 for (int i=n/2-1;i>=0;i--){  
 maxHeapify(i);  
 }  
 }  
  
 public void maxHeapify(int index) {  
 int largest=index;  
 int left=index\*2;  
 int right=index\*2+1;  
  
 if (left<=heapSize && tasks[left] != null && tasks[left].priority>tasks[largest].priority){  
 largest=left;  
 }  
 if (right<=heapSize && tasks[right] != null && tasks[right].priority>tasks[largest].priority){  
 largest=right;  
 }  
 if (largest!=index && tasks[index]!=null){  
 swapTasks(index, largest);  
 maxHeapify(largest);  
 }  
 }  
  
 public void delMaxPriorityTask() {  
 if (heapSize > 0 && tasks[1] != null) {  
 System.*out*.println("Deleting task " + tasks[1].number + " with priority " + tasks[1].priority);  
 System.*out*.println("Heap size: " + heapSize);  
  
 tasks[1] = tasks[heapSize];  
 tasks[heapSize] = null;  
 heapSize--;  
  
 // Heapify to maintain heap property after removal  
 maxHeapify(1);  
 } else {  
 System.*out*.println("Heap is empty.");  
 }  
 }  
  
 public void swapTasks(int i, int j){  
 task temp=tasks[i];  
 tasks[i]=tasks[j];  
 tasks[j]=temp;  
 }  
  
 public void Display(){  
 System.*out*.println("Current Heap: ");  
 for (int i=1;i<=heapSize;i++){  
 System.*out*.println("Task "+tasks[i].number+" | Priority: "+tasks[i].priority);  
 }  
 System.*out*.println();  
 }  
}

****

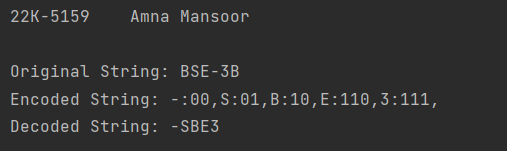
**Task 4:**

import java.util.PriorityQueue;  
  
public class Task4 {  
 public static String encode(String input) {  
 System.*out*.println("22K-5159 Amna Mansoor\n");  
 int[] frequency = new int[256]; // assume ASCII characters  
  
 // counting frequency of each character  
 for (char ch : input.toCharArray()) {  
 frequency[ch]++;  
 }  
  
 // creating priority queue/min heap by frequencies  
 PriorityQueue<HuffmanNode> minHeap = new PriorityQueue<>();  
 for (char ch = 0; ch < 256; ch++) {  
 if (frequency[ch] > 0) {  
 minHeap.offer(new HuffmanNode(ch, frequency[ch]));  
 }  
 }  
  
 // building the Huffman tree by combining nodes with the lowest frequencies  
 while (minHeap.size() > 1) {  
 HuffmanNode left = minHeap.poll();  
 HuffmanNode right = minHeap.poll();  
  
 HuffmanNode newNode = new HuffmanNode('\0', left.frequency + right.frequency);  
 newNode.left = left;  
 newNode.right = right;  
  
 minHeap.offer(newNode);  
 }  
  
 HuffmanNode root = minHeap.poll();  
 StringBuilder encodedString = new StringBuilder();  
 *build*(root, "", encodedString);  
  
 return encodedString.toString();  
 }  
  
 // recursive method to traverse the Huffman tree and build binary  
 private static void build(HuffmanNode node, String code, StringBuilder encodedString) {  
 if (node != null) {  
 // if the node is a leaf node, append the character and its binary code to the result  
 if (node.left == null && node.right == null) {  
 encodedString.append(node.character).append(":").append(code).append(",");  
 }  
  
 *build*(node.left, code + "0", encodedString);  
 *build*(node.right, code + "1", encodedString);  
 }  
 }  
 public static void main(String[] args) {  
 String input = "BSE-3B";  
 String encodedString = *encode*(input);  
  
 // Print the original string  
 System.*out*.println("Original String: " + input);  
  
 // Print the encoded string  
 System.*out*.println("Encoded String: " + encodedString);  
 }  
}  
  
class HuffmanNode implements Comparable<HuffmanNode> {  
 char character;  
 int frequency;  
 HuffmanNode left, right; // left and right children  
  
 public HuffmanNode(char character, int frequency) {  
 this.character = character;  
 this.frequency = frequency;  
 }  
  
 @Override  
 public int compareTo(HuffmanNode other) {  
 return this.frequency - other.frequency;  
 }  
}

****

**Task 5:**

import java.util.PriorityQueue;  
  
public class Task5 {  
 private static Huffman\_Node *root*;  
  
 public static String encode(String input) {  
 System.*out*.println("22K-5159 Amna Mansoor\n");  
 int[] frequency = new int[256]; // assume ASCII characters  
  
 // counting frequency of each character  
 for (char ch : input.toCharArray()) {  
 frequency[ch]++;  
 }  
  
 // creating priority queue/min heap by frequencies  
 PriorityQueue<Huffman\_Node> minHeap = new PriorityQueue<>();  
 for (char ch = 0; ch < 256; ch++) {  
 if (frequency[ch] > 0) {  
 minHeap.offer(new Huffman\_Node(ch, frequency[ch]));  
 }  
 }  
  
 // building the Huffman tree by combining nodes with the lowest frequencies  
 while (minHeap.size() > 1) {  
 Huffman\_Node left = minHeap.poll();  
 Huffman\_Node right = minHeap.poll();  
  
 Huffman\_Node newNode = new Huffman\_Node('\0', left.frequency + right.frequency);  
 newNode.left = left;  
 newNode.right = right;  
  
 minHeap.offer(newNode);  
 }  
  
 *root* = minHeap.poll();  
 StringBuilder encodedString = new StringBuilder();  
 *build*(*root*, "", encodedString);  
  
 return encodedString.toString();  
 }  
  
 private static void build(Huffman\_Node node, String code, StringBuilder encodedString) {  
 if (node != null) {  
 if (node.left == null && node.right == null) {  
 encodedString.append(node.character).append(":").append(code).append(",");  
 }  
  
 *build*(node.left, code + "0", encodedString);  
 *build*(node.right, code + "1", encodedString);  
 }  
 }  
  
 public static String decode(String encodedString) {  
 StringBuilder decodedString = new StringBuilder();  
 int index = 0;  
  
 while (index < encodedString.length()) {  
 char character = encodedString.charAt(index++);  
  
 // skip ':' character  
 index++;  
  
 StringBuilder code = new StringBuilder();  
 while (index < encodedString.length() && encodedString.charAt(index) != ',') {  
 code.append(encodedString.charAt(index++));  
 }  
  
 // Skip ',' character  
 index++;  
  
 // Initialize root for decoding  
 Huffman\_Node decodingRoot = *root*;  
  
 // Convert binary code to character  
 decodedString.append(*getCharacterFromCode*(decodingRoot, code.toString()));  
 }  
  
 return decodedString.toString();  
 }  
  
 private static char getCharacterFromCode(Huffman\_Node node, String code) {  
 for (char bit : code.toCharArray()) {  
 if (node == null) {  
 // Handle the case where node is null  
 throw new IllegalStateException("Invalid Huffman encoding");  
 }  
  
 if (bit == '0') {  
 node = node.left;  
 } else {  
 node = node.right;  
 }  
 }  
  
 if (node == null) {  
 // Handle the case where node is null  
 throw new IllegalStateException("Invalid Huffman encoding");  
 }  
  
 return node.character;  
 }  
  
 public static void main(String[] args) {  
 String input = "BSE-3B";  
 String encodedString = *encode*(input);  
  
 // Print the original string  
 System.*out*.println("Original String: " + input);  
  
 // Print the encoded string  
 System.*out*.println("Encoded String: " + encodedString);  
  
 // Decode the encoded string  
 String decodedString = *decode*(encodedString);  
 System.*out*.println("Decoded String: " + decodedString);  
 }  
}  
  
class Huffman\_Node implements Comparable<Huffman\_Node> {  
 char character;  
 int frequency;  
 Huffman\_Node left, right; // left and right children  
  
 public Huffman\_Node(char character, int frequency) {  
 this.character = character;  
 this.frequency = frequency;  
 }  
  
 @Override  
 public int compareTo(Huffman\_Node other) {  
 return this.frequency - other.frequency;  
 }  
}

****