# Assignment – 2

# for

**ADVANCED DATA STRUCTURES**

**AND ALGORITHMS (UNC601)**

## Submitted By: Vishav Singla Roll number: 102115204 Group: 3NC8

**Submitted to**

# Dr. Ram Kishan Dewangan

# Assistant Professor



**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**

**THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY, (A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB**

**INDIA**

**January – May**

**2024**

**1. Implement the Heapsort algorithm to arrange numbers in descending order.**

**CODE:**

public class HeapSort {

    public void heapify(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;

            heapify(arr, n, largest);

        }

    }

    public void heapSort(int arr[]) {

        int n = arr.length;

        for (int i = n / 2 - 1; i >= 0; i--) {

            heapify(arr, n, i);

        }

        for (int i = n - 1; i > 0; i--) {

            int temp = arr[0];

            arr[0] = arr[i];

            arr[i] = temp;

            heapify(arr, i, 0);

        }

    }

    public static void main(String args[]) {

        int[][] testCases = {

                { 4, 10, 3, 5, 1 },

                { 8, 15, 2, 7, 1, 9 },

                { 6, 20, 8, 12, 15, 4 }

        };

        for (int[] testCase : testCases) {

            int n = testCase.length;

            HeapSort heapSort = new HeapSort();

            heapSort.heapSort(testCase);

            System.out.println("Input array: " + Arrays.toString(testCase));

            System.out.println("Sorted array in descending order: " + Arrays.toString(testCase));

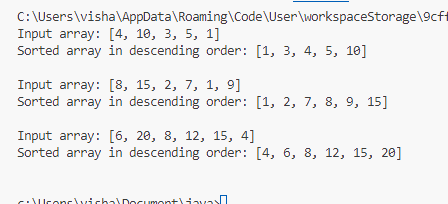
            System.out.println();

        }

    }

}

**OUTPUT:**

****

**2. Implement a min-priority queue with a min-heap. The program should have functions for the**

**following operations:**

**HEAP-MINIMUM to get the element with the smallest key,**

**HEAP-EXTRACT-MIN to remove and return the element with the smallest key,**

**HEAP-DECREASE-KEY decreases the value of the element to a new value, and**

**MIN-HEAP-INSERT to insert the element.**

**CODE:**

public class MinPriorityQueue {

    private int[] heap;

    private int size;

    private int capacity;

    public MinPriorityQueue(int capacity) {

        this.capacity = capacity;

        this.size = 0;

        this.heap = new int[capacity];

    }

    public int heapMinimum() {

        if (size == 0) {

            throw new IllegalStateException("Heap is empty");

        }

        return heap[0];

    }

    public int heapExtractMin() {

        if (size == 0) {

            throw new IllegalStateException("Heap is empty");

        }

        int min = heap[0];

        heap[0] = heap[size - 1];

        size--;

        minHeapify(0);

        return min;

    }

    public void heapDecreaseKey(int i, int newValue) {

        if (i >= size) {

            throw new IllegalArgumentException("Index out of bounds");

        }

        if (newValue > heap[i]) {

            throw new IllegalArgumentException("New value is greater than the current value");

        }

        heap[i] = newValue;

        while (i > 0 && heap[parent(i)] > heap[i]) {

            swap(i, parent(i));

            i = parent(i);

        }

    }

    public void minHeapInsert(int key) {

        if (size == capacity) {

            throw new IllegalStateException("Heap is full");

        }

        size++;

        int i = size - 1;

        heap[i] = Integer.MAX\_VALUE;

        heapDecreaseKey(i, key);

    }

    private void minHeapify(int i) {

        int left = leftChild(i);

        int right = rightChild(i);

        int smallest = i;

        if (left < size && heap[left] < heap[i]) {

            smallest = left;

        }

        if (right < size && heap[right] < heap[smallest]) {

            smallest = right;

        }

        if (smallest != i) {

            swap(i, smallest);

            minHeapify(smallest);

        }

    }

    private void swap(int i, int j) {

        int temp = heap[i];

        heap[i] = heap[j];

        heap[j] = temp;

    }

    private int parent(int i) {

        return (i - 1) / 2;

    }

    private int leftChild(int i) {

        return 2 \* i + 1;

    }

    private int rightChild(int i) {

        return 2 \* i + 2;

    }

    public static void main(String[] args) {

        // Test Case 1

        MinPriorityQueue minPriorityQueue1 = new MinPriorityQueue(10);

        minPriorityQueue1.minHeapInsert(4);

        minPriorityQueue1.minHeapInsert(2);

        minPriorityQueue1.minHeapInsert(6);

        minPriorityQueue1.minHeapInsert(1);

        System.out.println("Test Case 1:");

        System.out.println("Min element: " + minPriorityQueue1.heapMinimum());

        System.out.println("Extracted min element: " + minPriorityQueue1.heapExtractMin());

        minPriorityQueue1.heapDecreaseKey(2, 1);

        System.out.println("Min element after decrease key: " + minPriorityQueue1.heapMinimum());

        System.out.println();

        // Test Case 2

        MinPriorityQueue minPriorityQueue2 = new MinPriorityQueue(15);

        minPriorityQueue2.minHeapInsert(8);

        minPriorityQueue2.minHeapInsert(5);

        minPriorityQueue2.minHeapInsert(12);

        minPriorityQueue2.minHeapInsert(3);

        minPriorityQueue2.minHeapInsert(7);

        System.out.println("Test Case 2:");

        System.out.println("Min element: " + minPriorityQueue2.heapMinimum());

        System.out.println("Extracted min element: " + minPriorityQueue2.heapExtractMin());

        minPriorityQueue2.heapDecreaseKey(2, 4);

        System.out.println("Min element after decrease key: " + minPriorityQueue2.heapMinimum());

        System.out.println();

        // Test Case 3

        MinPriorityQueue minPriorityQueue3 = new MinPriorityQueue(20);

        minPriorityQueue3.minHeapInsert(10);

        minPriorityQueue3.minHeapInsert(15);

        minPriorityQueue3.minHeapInsert(20);

        minPriorityQueue3.minHeapInsert(5);

        minPriorityQueue3.minHeapInsert(25);

        System.out.println("Test Case 3:");

        System.out.println("Min element: " + minPriorityQueue3.heapMinimum());

        System.out.println("Extracted min element: " + minPriorityQueue3.heapExtractMin());

        minPriorityQueue3.heapDecreaseKey(2, 8);

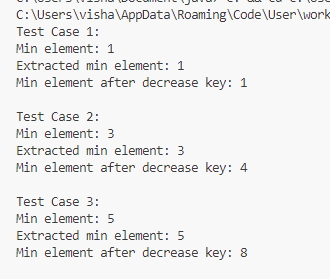
        System.out.println("Min element after decrease key: " +

minPriorityQueue3.heapMinimum());

    }

}

**OUTPUT:**

****

**3. Write a program to find the largest element in an unsorted array.**

**CODE:**

import java.util.Arrays;

import java.util.PriorityQueue;

public class LargestElement{

    public static void main(String[] args) {

        int[][] arrays = {

                {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5},

                {8, 12, 5, 7, 10, 15, 3, 11},

                {2, 4, 6, 8, 10},

                {15, 12, 9, 6, 3}

        };

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

            int[] arr = arrays[i];

            System.out.println("Input Array " + (i + 1) + ": " + Arrays.toString(arr));

            int result = findLargestElement(arr);

            System.out.println("The largest element is: " + result);

            System.out.println();

        }

    }

    public static int findLargestElement(int[] arr) {

        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

        for (int num : arr) {

            maxHeap.offer(num);

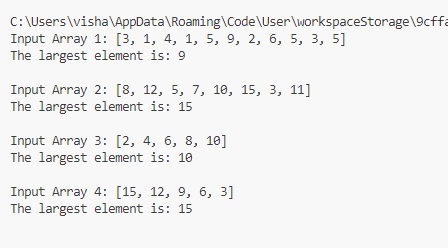
        }

        return maxHeap.poll();

    }

}

**OUTPUT:**

****

**4. Write a program to convert a binary search tree into a min-heap**

**CODE:**

import java.util.ArrayList;

import java.util.List;

class Node {

    int data;

    Node left, right;

    public Node(int item) {

        data = item;

        left = right = null;

    }

}

public class BSTtoMinHeap {

    Node root;

    private List<Integer> sortedNodes;

    public BSTtoMinHeap() {

        sortedNodes = new ArrayList<>();

    }

    private void inOrderTraversal(Node node) {

        if (node != null) {

            inOrderTraversal(node.left);

            sortedNodes.add(node.data);

            inOrderTraversal(node.right);

        }

    }

    private void arrayToMinHeap(Node node, int[] arr, int i) {

        if (node != null) {

            node.data = arr[i++];

            arrayToMinHeap(node.left, arr, i);

            arrayToMinHeap(node.right, arr, i);

        }

    }

    public void convertBSTtoMinHeap(Node root) {

        inOrderTraversal(root);

        int[] heapArray = new int[sortedNodes.size()];

        for (int i = 0; i < sortedNodes.size(); i++) {

            heapArray[i] = sortedNodes.get(i);

        }

        arrayToMinHeap(root, heapArray, 0);

    }

    public static void main(String args[]) {

        BSTtoMinHeap tree = new BSTtoMinHeap();

        tree.root = new Node(4);

        tree.root.left = new Node(2);

        tree.root.right = new Node(6);

        tree.root.left.left = new Node(1);

        tree.root.left.right = new Node(3);

        tree.root.right.left = new Node(5);

        tree.root.right.right = new Node(7);

        System.out.println("BST before conversion to Min-Heap:");

        printInOrder(tree.root);

        tree.convertBSTtoMinHeap(tree.root);

        System.out.println("\nMin-Heap after conversion from BST:");

        printInOrder(tree.root);

    }

    private static void printInOrder(Node root) {

        if (root != null) {

            printInOrder(root.left);

            System.out.print(root.data + " ");

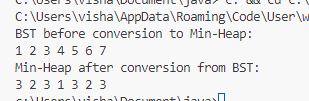
            printInOrder(root.right);

        }

    }

}

**OUTPUT:**

****