Tree Code(Insert ,Delete,Search)

package BST\_CODE;

import java.util.\*;

class Node {

    int data;

    Node left, right;

    public Node(int val) {

        data = val;

        left = right = null;

    }

}

public class BST {

    Node root;

    // Insert a node

    Node insert(Node root, int val) {

        if (root == null)

            return new Node(val);

        if (val < root.data)

            root.left = insert(root.left, val);

        else if (val > root.data)

            root.right = insert(root.right, val);

        return root;

    }

    // Search a node

    boolean search(Node root, int key) {

        if (root == null)

            return false;

        if (root.data == key)

            return true;

        if (key < root.data)

            return search(root.left, key);

        else

            return search(root.right, key);

    }

    // Find minimum node in right subtree

    Node findMin(Node root) {

        while (root.left != null)

            root = root.left;

        return root;

    }

    // Delete a node

    Node delete(Node root, int key) {

        if (root == null)

            return null;

        if (key < root.data)

            root.left = delete(root.left, key);

        else if (key > root.data)

            root.right = delete(root.right, key);

        else {

            // Node found

            if (root.left == null)

                return root.right;

            else if (root.right == null)

                return root.left;

            Node temp = findMin(root.right);

            root.data = temp.data;

            root.right = delete(root.right, temp.data);

        }

        return root;

    }

    // Inorder traversal

    void inorder(Node root) {

        if (root == null)

            return;

        inorder(root.left);

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

        inorder(root.right);

    }

    // Level Order Traversal using Queue

    void levelOrder(Node root) {

        if (root == null) return;

        Queue<Node> q = new LinkedList<>();

        q.add(root);

        while (!q.isEmpty()) {

            Node curr = q.poll();

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

            if (curr.left != null)

                q.add(curr.left);

            if (curr.right != null)

                q.add(curr.right);

        }

    }

    // Main method

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        BST tree = new BST();

        System.out.print("Enter number of elements: ");

        int n = sc.nextInt();

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

        for (int i = 0; i < n; i++) {

            int x = sc.nextInt();

            tree.root = tree.insert(tree.root, x);

        }

        System.out.print("\nInorder Traversal: ");

        tree.inorder(tree.root);

        System.out.print("\nLevel Order Traversal: ");

        tree.levelOrder(tree.root);

        System.out.print("\n\nEnter element to search: ");

        int key = sc.nextInt();

        System.out.println(tree.search(tree.root, key) ? "Found" : "Not Found");

        System.out.print("Enter element to delete: ");

        key = sc.nextInt();

        tree.root = tree.delete(tree.root, key);

        System.out.print("Inorder after deletion: ");

        tree.inorder(tree.root);

        System.out.println();

    }

}

Heap Sort

package Heap;

public class HeapSort {

    // Function to perform heap sort

    public void sort(int arr[]) {

        int n = arr.length;

        // Step 1: Build max heap

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

            heapify(arr, n, i);

        // Step 2: One by one extract elements from heap

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

            // Move current root to end

            int temp = arr[0];

            arr[0] = arr[i];

            arr[i] = temp;

            // Heapify the reduced heap

            heapify(arr, i, 0);

        }

    }

    // To heapify a subtree rooted with node i, n is size of heap

    void heapify(int arr[], int n, int i) {

        int largest = i; // Initialize largest as root

        int left = 2 \* i + 1; // left child

        int right = 2 \* i + 2; // right child

        // If left child is larger than root

        if (left < n && arr[left] > arr[largest])

            largest = left;

        // If right child is larger than largest so far

        if (right < n && arr[right] > arr[largest])

            largest = right;

        // If largest is not root

        if (largest != i) {

            int swap = arr[i];

            arr[i] = arr[largest];

            arr[largest] = swap;

            // Recursively heapify the affected subtree

            heapify(arr, n, largest);

        }

    }

    // Utility function to print array

    static void printArray(int arr[]) {

        for (int num : arr)

            System.out.print(num + " ");

        System.out.println();

    }

    // Main method to test

    public static void main(String args[]) {

        int arr[] = {12, 11, 13, 5, 6, 7};

        HeapSort hs = new HeapSort();

        System.out.println("Original array:");

        printArray(arr);

        hs.sort(arr);

        System.out.println("Sorted array:");

        printArray(arr);

    }

}