Code : 1  
package Topic\_16\_BinaryTree;

import java.io.IOException;

import java.util.ArrayDeque;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Stack;

class Practice\_BinaryTree {

public static class Node {

Node left;

Node right;

int data;

Node(Integer data, Node left, Node right) {

this.data = data;

this.left = left;

this.right = right;

}

}

public static void main(String[] args) throws NumberFormatException, IOException {

// int n = Integer.parseInt(br.readLine());

// String input = "50 25 12 n n 37 30 n n n 75 62 n 70 n n 87 n n";

// String[] values = input.split(" ");

String[] values = "50 25 12 n n 37 30 n n 40 n n 75 62 60 n n 70 n n 87 n n".split(" ");

Integer[] arr = new Integer[values.length];

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

if (values[i].equals("n")) {

arr[i] = null;

} else {

arr[i] = Integer.parseInt(values[i]);

}

}

BinaryTree tree = new BinaryTree();

Node root = tree.construct(arr);

tree.display(root);

int size = tree.size(root);

int sum = tree.sum(root);

int max = tree.max(root);

int ht = tree.height(root);

System.out.print("Size:");

System.out.println(size);

System.out.print("Sum:");

System.out.println(sum);

System.out.print("Max:");

System.out.println(max);

System.out.print("Height:");

System.out.println(ht);

System.out.println("Level Order:");

tree.levelOrder3rdApproach(root);

int data = 30;

System.out.println("Finding " + data + " in a tree and result is " + tree.find(root, data));

System.out.println("Node to root path");

ArrayList<Integer> list = tree.nodeToRootPath(root, data);

for (Integer i : list) {

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

}

System.out.println("\nTraversals");

tree.iterativePrePostInTraversal(root);

System.out.println();

int k = 3;

System.out.println("Print " + k + " Level down");

tree.printKLevelsDown(root, k);

System.out.println("Path to leaf root");

tree.pathToLeafFromRoot(root, "");

System.out.println("Path to leaf root In Range");

int lo = 150;

int hi = 250;

tree.pathToLeafFromRootInRange(root, "", 0, lo, hi);

System.out.println("createLeftCloneFromTree");

tree.createLeftCloneFromTree(root);

System.out.println("transBackFromLeftClonedTree");

tree.transBackFromLeftClonedTree(root);

System.out.println("Remove Leaves in Binary Tree");

Node tempRoot = tree.construct(arr);

Node root1 = tree.removeLeaves(tempRoot);

tree.display(root1);

tree.printSingleChildNodes(root, null);

System.out.println("Diameter: " + tree.diameter1(root));

tree.tilt(root);

System.out.println("Is Tilt: " + tree.tiltCalc);

tree.isBinarySearchTree(root);

BinaryTree.BSTPair p = tree.isBinarySearchTree(root);

System.out.println("Is Binary Search Tree: " + p.isBST);

BinaryTree.BalPair bp = tree.isBalanced(root);

System.out.println("Is balanced Tree: " + bp.isBal);

BinaryTree.BSTPair p1 = tree.isLargestBstSubtree(root);

System.out.println("LargestBstSubtree: ");

System.out.print(p1.root.data + "@" + p1.size);

}

public static class BinaryTree {

private static class Pair {

Node node;

int state;

public Pair(Node node, int state) {

this.node = node;

this.state = state;

}

}

private Node construct(Integer[] arr) {

Stack<Pair> stack = new Stack<>();

Node root = new Node(arr[0], null, null);

Pair pair = new Pair(root, 1);

stack.push(pair);

int idx = 1;

while (!stack.isEmpty()) {

Pair peek = stack.peek();

if (peek.state == 1) {

Integer data = arr[idx];

if (data != null) {

Node node = new Node(data, null, null);

peek.node.left = node;

Pair lp = new Pair(node, 1);

stack.push(lp);

}

peek.state++;

idx++;

} else if (peek.state == 2) {

Integer data = arr[idx];

if (data != null) {

Node node = new Node(data, null, null);

peek.node.right = node;

Pair rp = new Pair(node, 1);

stack.push(rp);

}

peek.state++;

idx++;

} else if (peek.state == 3) {

stack.pop();

}

}

return root;

}

private void display(Node node) {

if (node == null) {

return;

}

String root = "<-" + node.data + "->";

String left = node.left == null ? "." : node.left.data + "";

String right = node.right == null ? "." : node.right.data + "";

System.out.println(left + root + right);

display(node.left);

display(node.right);

}

public static int size(Node node) {

// write your code here

if (node == null) {

return 0;

}

int leftSize = size(node.left);

int rightSize = size(node.right);

return 1 + leftSize + rightSize;

}

public static int sum(Node node) {

if (node == null)

return 0;

return node.data + sum(node.left) + sum(node.right);

}

public static int max(Node node) {

if (node == null)

return 0;

int lmax = max(node.left);

int rmax = max(node.right);

return Math.max(lmax, Math.max(node.data, rmax));

// write your code here

}

public static int height(Node node) {

if (node == null)

return 0; // if need height with respect to edges "return -1" instead of "return 0"

int leftHeight = height(node.left);

int rightHeight = height(node.right);

return Math.max(leftHeight, rightHeight) + 1;

}

public static void levelOrder(Node node) { // Parent and child queue approach after that remove print and add

LinkedList<Node> main = new LinkedList<>();

LinkedList<Node> child = new LinkedList<>();

main.addFirst(node);

while (!main.isEmpty()) {

Node temp = main.removeFirst();

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

if (temp.left != null)

child.addLast(temp.left);

if (temp.right != null)

child.addLast(temp.right);

if (main.isEmpty()) {

main = child;

child = new LinkedList<>();

System.out.println();

}

}

}

public static void levelOrder2ndApproach(Node node) { // Count approach

LinkedList<Node> main = new LinkedList<>();

main.addFirst(node);

while (!main.isEmpty()) {

int count = main.size();

for (int i = 0; i < count; i++) {

Node temp = main.removeFirst();

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

if (temp.left != null)

main.addLast(temp.left);

if (temp.right != null)

main.addLast(temp.right);

}

System.out.println();

}

}

public static void levelOrder3rdApproach(Node node) { // Delimiter approach

LinkedList<Node> main = new LinkedList<>();

main.addFirst(node);

Node delimeterNode = new Node(-1, null, null);

main.add(delimeterNode);

while (!main.isEmpty()) {

Node temp = main.removeFirst();

if (temp.data == -1) {

System.out.println();

if (main.size() > 0) {

main.add(temp);

}

continue;

}

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

if (temp.left != null)

main.addLast(temp.left);

if (temp.right != null)

main.addLast(temp.right);

}

}

// Using Pair class when level getting print new line

static class LPair {

Node node;

int level;

}

public static void levelOrder4thApproach(Node node) {

ArrayDeque<LPair> q = new ArrayDeque<>();

LPair rp = new LPair();

rp.node = node;

rp.level = 1;

q.add(rp);

int level = 1;

while (q.size() > 0) {

LPair temp = q.remove();

if (temp.level > level) {

level = temp.level;

System.out.println();

}

System.out.print(temp.node.data + " ");

if (temp.node.left != null) {

LPair leftp = new LPair();

leftp.node = temp.node.left;

leftp.level = temp.level + 1;

q.add(leftp);

}

if (temp.node.right != null) {

LPair rightp = new LPair();

rightp.node = temp.node.right;

rightp.level = temp.level + 1;

q.add(rightp);

}

}

}

public static boolean find(Node node, int data) {

// write your code here

if (node == null) {

return false;

}

if (node.data == data) {

return true;

}

boolean left = find(node.left, data);

if (left) {

return true;

}

boolean right = find(node.right, data);

if (right) {

return true;

}

return false;

}

public static void iterativePrePostInTraversal(Node node) {

preOrderTraversal(node);

System.out.println();

inOrderTraversal(node);

System.out.println();

postOrderTraversal(node);

}

private static void preOrderTraversal(Node node) {

if (node == null)

return;

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

preOrderTraversal(node.left);

preOrderTraversal(node.right);

}

private static void inOrderTraversal(Node node) {

if (node == null)

return;

inOrderTraversal(node.left);

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

inOrderTraversal(node.right);

}

private static void postOrderTraversal(Node node) {

if (node == null)

return;

postOrderTraversal(node.left);

postOrderTraversal(node.right);

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

}

public static ArrayList<Integer> nodeToRootPath(Node node, int data) {

// write your code here

if (node == null) {

return new ArrayList<>();

}

if (node.data == data) {

ArrayList<Integer> list = new ArrayList<>();

list.add(node.data);

return list;

}

ArrayList<Integer> leftList = nodeToRootPath(node.left, data);

if (leftList.size() > 0) {

leftList.add(node.data);

return leftList;

}

ArrayList<Integer> rightList = nodeToRootPath(node.right, data);

if (rightList.size() > 0) {

rightList.add(node.data);

return rightList;

}

return new ArrayList<>();

}

public static Node createLeftCloneFromTree(Node node) {

if (node == null)

return null;

Node left = createLeftCloneFromTree(node.left);

Node right = createLeftCloneFromTree(node.right);

Node newNode = new Node(node.data, left, null);

node.left = newNode;

return node;

}

public static Node transBackFromLeftClonedTree(Node node) {

if (node == null) {

return null;

}

Node left = transBackFromLeftClonedTree(node.left.left);

Node right = transBackFromLeftClonedTree(node.right);

node.left = left;

return node;

}

public static Node removeLeaves(Node node) {

if (node == null)

return null;

if (node.left == null && node.right == null) {

return null;

}

node.left = removeLeaves(node.left);

node.right = removeLeaves(node.right);

return node;

}

static class DPair {

int ht;

int dia;

}

public static DPair diameter3(Node node) {

if (node == null) {

DPair bp = new DPair();

bp.ht = -1;

bp.dia = 0;

return bp;

}

DPair lp = diameter3(node.left);

DPair rp = diameter3(node.right);

DPair mp = new DPair();

mp.ht = Math.max(lp.ht, rp.ht) + 1;

mp.dia = Math.max(lp.ht + rp.ht + 2, Math.max(lp.dia, rp.dia));

return mp;

}

public static int diameter1(Node node) {

if (node == null) {

return 0;

}

int lh = height(node.left);

int rh = height(node.right);

int ld = diameter1(node.left);

int rd = diameter1(node.right);

return Math.max(lh + rh + 2, Math.max(ld, rd));

}

static int tiltCalc = 0;

public static int tilt(Node node) {

if (node == null) {

return 0;

}

int ls = tilt(node.left);

int rs = tilt(node.right);

int ts = ls + rs + node.data;

tiltCalc += Math.abs(ls - rs);

return ts;

}

public static BSTPair isBinarySearchTree(Node node) {

if (node == null) {

BSTPair bp = new BSTPair();

bp.min = Integer.MAX\_VALUE;

bp.max = Integer.MIN\_VALUE;

bp.isBST = true;

return bp;

}

BSTPair lp = isBinarySearchTree(node.left);

BSTPair rp = isBinarySearchTree(node.right);

BSTPair mp = new BSTPair();

mp.min = Math.min(node.data, Math.min(lp.min, rp.min));

mp.max = Math.max(node.data, Math.max(lp.max, rp.max));

mp.isBST = lp.isBST && rp.isBST && node.data >= lp.max && node.data <= rp.min;

return mp;

}

public static class BalPair {

int ht;

boolean isBal;

}

public static BalPair isBalanced(Node node) {

if (node == null) {

BalPair bp = new BalPair();

bp.ht = -1;

bp.isBal = true;

return bp;

}

BalPair lp = isBalanced(node.left);

BalPair rp = isBalanced(node.right);

BalPair mp = new BalPair();

mp.ht = Math.max(lp.ht, rp.ht) + 1;

mp.isBal = lp.isBal && rp.isBal && Math.abs(lp.ht - rp.ht) <= 1;

return mp;

}

public static class BSTPair {

boolean isBST;

int min;

int max;

Node root; //1

int size;

}

public static BSTPair isLargestBstSubtree(Node node) {

if (node == null) {

BSTPair bp = new BSTPair();

bp.isBST = true;

bp.min = Integer.MAX\_VALUE;

bp.max = Integer.MIN\_VALUE;

bp.root = null;

bp.size = 0;

return bp;

}

BSTPair lp = isLargestBstSubtree(node.left);

BSTPair rp = isLargestBstSubtree(node.right);

BSTPair mp = new BSTPair();

mp.isBST = lp.isBST && rp.isBST && (node.data >= lp.max && node.data <= rp.min);

mp.min = Math.min(node.data, Math.min(lp.min, rp.min));

mp.max = Math.max(node.data, Math.max(lp.max, rp.max));

if (mp.isBST) { //2

mp.root = node;

mp.size = lp.size + rp.size + 1;

} else if (lp.size > rp.size) { //3

mp.root = lp.root;

mp.size = lp.size;

} else { //4

mp.root = rp.root;

mp.size = rp.size;

}

return mp;

}

public static void printSingleChildNodes(Node node, Node parent) {

// write your code here

if (node == null) {

return;

}

if (parent != null && parent.left == null && parent.right == node) {

System.out.println(node.data);

} else if (parent != null && parent.right == null && parent.left == node) {

System.out.println(node.data);

}

printSingleChildNodes(node.left, node);

printSingleChildNodes(node.right, node);

}

public static void printKLevelsDown(Node node, int k) {

if (node == null) {

return;

}

if (k == 0) {

System.out.println(node.data);

return;

}

printKLevelsDown(node.left, k - 1);

printKLevelsDown(node.right, k - 1);

}

public static void pathToLeafFromRoot(Node node, String path) {

// write your code here

if (node == null) {

System.out.println(path);

return;

}

pathToLeafFromRoot(node.left, path + " " + node.data);

pathToLeafFromRoot(node.right, path + " " + node.data);

}

public static void pathToLeafFromRootInRange(Node node, String path, int sum, int lo, int hi) {

if (node == null) { //1

return;

}

if (node.left == null && node.right == null) { //2

sum += node.data; //3

if (sum >= lo && sum <= hi) { //4

System.out.println(path + node.data);

}

return;

}

pathToLeafFromRootInRange(node.left, path + node.data + " ", sum + node.data, lo, hi); //5

pathToLeafFromRootInRange(node.right, path + node.data + " ", sum + node.data, lo, hi);

}

}

}

Code : 2  
package Topic\_16\_BinaryTree;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.ArrayList;

import java.util.Stack;

class PrintKNodeFar {

public static class Node {

Node left;

Node right;

int data;

Node(Integer data, Node left, Node right) {

this.data = data;

this.left = left;

this.right = right;

}

public Node(int data) {

this.data = data;

}

}

public static void main(String[] args) throws NumberFormatException, IOException {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

// int n = Integer.parseInt(br.readLine());

// String[] values = "50 25 12 n n 37 30 n n n 75 62 n 70 n 85 n n 87 n n".split(" ");

String[] values = "50 25 12 n n 37 30 n n n 75 62 n 70 n n 87 n n".split(" ");

Integer[] arr = new Integer[values.length];

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

if (values[i].equals("n") == true) {

arr[i] = null;

} else {

arr[i] = Integer.parseInt(values[i]);

}

}

BinaryTreeKFar tree = new BinaryTreeKFar();

Node root = tree.construct(arr);

int k = 2;

int data = 37;

System.out.println("Printing " + k + " Nodes far from " + data);

tree.printKNodesFar(root, data, k);

}

static class BinaryTreeKFar {

private class Pair {

Node node;

int state;

public Pair(Node node, int state) {

this.node = node;

this.state = state;

}

}

private Node construct(Integer[] arr) {

Stack<Pair> stack = new Stack<>();

Node root = new Node(arr[0], null, null);

Pair pair = new Pair(root, 1);

stack.push(pair);

int idx = 1;

while (!stack.isEmpty()) {

Pair peek = stack.peek();

if (peek.state == 1) {

Integer data = arr[idx];

if (data != null) {

Node node = new Node(data, null, null);

peek.node.left = node;

Pair lp = new Pair(node, 1);

stack.push(lp);

}

peek.state++;

idx++;

} else if (peek.state == 2) {

Integer data = arr[idx];

if (data != null) {

Node node = new Node(data, null, null);

peek.node.right = node;

Pair rp = new Pair(node, 1);

stack.push(rp);

}

peek.state++;

idx++;

} else if (peek.state == 3) {

stack.pop();

}

}

return root;

}

public static ArrayList<Node> nodeToRootPath(Node node, int data) {

// write your code here

if (node == null) {

return new ArrayList<>();

}

if (node.data == data) {

ArrayList<Node> list = new ArrayList<>();

list.add(node);

return list;

}

ArrayList<Node> leftList = nodeToRootPath(node.left, data);

if (leftList.size() > 0) {

leftList.add(node);

return leftList;

}

ArrayList<Node> rightList = nodeToRootPath(node.right, data);

if (rightList.size() > 0) {

rightList.add(node);

return rightList;

}

return new ArrayList<>();

}

public static void printKLevelsDown(Node node, int k, Node blocker) {

if (node == null || k < 0 || node == blocker) {

return;

}

if (k == 0) {

System.out.println(node.data);

return;

}

printKLevelsDown(node.left, k - 1, blocker);

printKLevelsDown(node.right, k - 1, blocker);

}

public static void printKNodesFar(Node node, int data, int k) {

ArrayList<Node> nodeToRootPath = nodeToRootPath(node, data);

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

printKLevelsDown(nodeToRootPath.get(i), k - i, i > 0 ? nodeToRootPath.get(i - 1) : null);

}

}

}

}