Tree Problems — Full Notes + Java Implementations

TreeProblems30.java

This file contains complete Java implementations for 30 tree problems commonly asked in interviews. You can download the Java source file and compile it locally.

Files included:

- TreeProblems30.java (Java source)

Usage:

- Compile: `javac TreeProblems30.java`
- Run: `java TreeProblems30`

Complete Java Source Code (TreeProblems30.java)

```
/*
 * TreeProblems30.java
 *
 * Complete implementations for 30 binary tree problems commonly asked in interviews.
 * Includes problem descriptions, approaches, complexity notes and example usages in main().
 *
 * Compile & run:
 *   javac TreeProblems30.java && java TreeProblems30
 *
 * Notes:
 * - This single file is for learning and quick testing. For production, split into classes/tests.
 */

import java.util.*;

public class TreeProblems30 {

    // ---------------------------- Node Definition ----------------------------
    static class TreeNode {
        int val;
        TreeNode left, right;
        TreeNode(int val) { this.val = val; }
        @Override public String toString() { return String.valueOf(val); }
    }

    // ---------------------------- 1-20 (from original) ---

-------------------------
    // For brevity, I've included the already provided 20 problem implementations here.
    // (1) Inorder (recursive & iterative)
    public static List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        inorderHelper(root, res);
        return res;
    }
    private static void inorderHelper(TreeNode node, List<Integer> res) {
        if (node == null) return;
        inorderHelper(node.left, res);
        res.add(node.val);
        inorderHelper(node.right, res);
```

```
    }
    public static List<Integer> inorderIterative(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Deque<TreeNode> stack = new ArrayDeque<>();
        TreeNode curr = root;
        while (curr != null || !stack.isEmpty()) {

         while (curr != null) { stack.push(curr); curr = curr.left; }
            curr = stack.pop();
            res.add(curr.val);
            curr = curr.right;
        }
        return res;
    }

    // (2) Preorder
    public static List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        preorderHelper(root, res);
        return res;
    }
    private static void preorderHelper(TreeNode node, List<Integer> res) {
        if (node == null) return;
        res.add(node.val);
        preorderHelper(node.left, res);
        preorderHelper(node.right, res);
    }
    public static List<Integer> preorderIterative(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) return res;
        Deque<TreeNode> stack

= new ArrayDeque<>();
        stack.push(root);
        while (!stack.isEmpty()) {
            TreeNode node = stack.pop();
            res.add(node.val);
            if (node.right != null) stack.push(node.right);
            if (node.left != null) stack.push(node.left);
        }
        return res;
    }

    // (3) Postorder
    public static List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        postorderHelper(root, res);
        return res;
    }
    private static void postorderHelper(TreeNode node, List<Integer> res) {
        if (node == null) return;
        postorderHelper(node.left, res);
        postorderHelper(node.right, res);
        res.add(node.val);
    }
    public static List<Integer> postorderIterative(TreeNode root)

{
        List<Integer> res = new ArrayList<>();
        if (root == null) return res;
        Deque<TreeNode> stack = new ArrayDeque<>();
        stack.push(root);
        while (!stack.isEmpty()) {
            TreeNode node = stack.pop();
            res.add(node.val);
            if (node.left != null) stack.push(node.left);
            if (node.right != null) stack.push(node.right);
        }
        Collections.reverse(res);
        return res;
    }

    // (4) Level Order
    public static List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if (root == null) return res;
        Queue<TreeNode> q = new LinkedList<>();
```

```
        q.offer(root);
        while (!q.isEmpty()) {
            int size = q.size();
            List<Intege
r> level = new ArrayList<>();
            for (int i = 0; i < size; i++) {
                TreeNode node = q.poll();
                level.add(node.val);
                if (node.left != null) q.offer(node.left);
                if (node.right != null) q.offer(node.right);
            }
            res.add(level);
        }
        return res;
    }

    // (5) Zigzag Level Order
    public static List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if (root == null) return res;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        boolean leftToRight = true;
        while (!q.isEmpty()) {
            int size = q.size();
            LinkedList<Integer> level = new LinkedList<>();
            for (int i
= 0; i < size; i++) {
                TreeNode node = q.poll();
                if (leftToRight) level.addLast(node.val);
                else level.addFirst(node.val);
                if (node.left != null) q.offer(node.left);
                if (node.right != null) q.offer(node.right);
            }
            res.add(level);
            leftToRight = !leftToRight;
        }
        return res;
    }

    // (6) Height / Max Depth
    public static int height(TreeNode root) {
        if (root == null) return 0;
        return 1 + Math.max(height(root.left), height(root.right));
    }

    // (7) Diameter (node count)
    static int diameterAnswer;
    public static int diameter(TreeNode root) {
        diameterAnswer = 0;
        diameterHelper(root);
        return diameterAnswer;
}
    private static int diameterHelper(TreeNode node) {
        if (node == null) return 0;
        int left = diameterHelper(node.left);
        int right = diameterHelper(node.right);
        diameterAnswer = Math.max(diameterAnswer, left + right + 1);
        return 1 + Math.max(left, right);
    }

    // (8) Left View
    public static List<Integer> leftView(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) return res;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        while (!q.isEmpty()) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = q.poll();
                if (i == 0) res.add(node.val);
                if (node.left != null) q.offer(node.left);
```

```
            if (node.right != null) q.offer(node.right);
           }
        }
        return res;
    }

    // (9) Right View
    public static List<Integer> rightView(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) return res;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        while (!q.isEmpty()) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = q.poll();
                if (i == size - 1) res.add(node.val);
                if (node.left != null) q.offer(node.left);
                if (node.right != null) q.offer(node.right);
            }
        }
        return res;
    }

    // (10) Top View
    static class PairNode { TreeNode node; int hd; PairNo
de(TreeNode n,int h){node=n;hd=h;} }
    public static List<Integer> topView(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) return res;
        Map<Integer, Integer> map = new TreeMap<>();
        Queue<PairNode> q = new LinkedList<>();
        q.offer(new PairNode(root,0));
        while (!q.isEmpty()) {
            PairNode p = q.poll();
            if (!map.containsKey(p.hd)) map.put(p.hd, p.node.val);
            if (p.node.left != null) q.offer(new PairNode(p.node.left, p.hd-1));
            if (p.node.right != null) q.offer(new PairNode(p.node.right, p.hd+1));
        }
        for (Integer v : map.values()) res.add(v);
        return res;
    }

    // (11) Bottom View
    public static List<Integer> bottomView(TreeNode root) {
        List
<Integer> res = new ArrayList<>();
        if (root == null) return res;
        Map<Integer, Integer> map = new TreeMap<>();
        Queue<PairNode> q = new LinkedList<>();
        q.offer(new PairNode(root,0));
        while (!q.isEmpty()) {
            PairNode p = q.poll();
            map.put(p.hd, p.node.val);
            if (p.node.left != null) q.offer(new PairNode(p.node.left, p.hd-1));
            if (p.node.right != null) q.offer(new PairNode(p.node.right, p.hd+1));
        }
        for (Integer v : map.values()) res.add(v);
        return res;
    }

    // (12) Has Path Sum (root-to-leaf)
    public static boolean hasPathSum(TreeNode root, int targetSum) {
        if (root == null) return false;
        if (root.left == null && root.right == null) return root.val == targetSum
;
        int newSum = targetSum - root.val;
        return hasPathSum(root.left, newSum) || hasPathSum(root.right, newSum);
    }

    // (13) All root-to-leaf paths
    public static List<List<Integer>> allPaths(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if (root == null) return res;
```

```
        allPathsHelper(root, new ArrayList<>(), res);
        return res;
    }
    private static void allPathsHelper(TreeNode node, List<Integer> path, List<List<Integer>> res) {
        if (node == null) return;
        path.add(node.val);
        if (node.left == null && node.right == null) res.add(new ArrayList<>(path));
        else {
            allPathsHelper(node.left, path, res);
            allPathsHelper(node.right, path, res);
        }
        path.remove(path
.size()-1);
    }

    // (14) Lowest Common Ancestor (general)
    public static TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null) return null;
        if (root == p || root == q) return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if (left != null && right != null) return root;
        return left != null ? left : right;
    }

    // (15) Is Balanced
    public static boolean isBalanced(TreeNode root) { return checkHeight(root) != -1; }
    private static int checkHeight(TreeNode node) {
        if (node == null) return 0;
        int left = checkHeight(node.left); if (left == -1) return -1;
        int right = checkHeight(node.right); if (right
== -1) return -1;
        if (Math.abs(left-right) > 1) return -1;
        return 1 + Math.max(left,right);
    }

    // (16) Is Symmetric
    public static boolean isSymmetric(TreeNode root) {
        if (root == null) return true;
        return isMirror(root.left, root.right);
    }
    private static boolean isMirror(TreeNode a, TreeNode b) {
        if (a == null && b == null) return true;
        if (a == null || b == null) return false;
        if (a.val != b.val) return false;
        return isMirror(a.left, b.right) && isMirror(a.right, b.left);
    }

    // (17) Maximum Path Sum
    static int maxPathSumAns;
    public static int maxPathSum(TreeNode root) {
        maxPathSumAns = Integer.MIN_VALUE;
        maxPathSumHelper(root);
        return maxPathSumAns;
    }
    privat
e static int maxPathSumHelper(TreeNode node) {
        if (node == null) return 0;
        int left = Math.max(0, maxPathSumHelper(node.left));
        int right = Math.max(0, maxPathSumHelper(node.right));
        maxPathSumAns = Math.max(maxPathSumAns, node.val + left + right);
        return node.val + Math.max(left, right);
    }

    // (18) Serialize / Deserialize (level-order)
    public static String serialize(TreeNode root) {
        if (root == null) return "";
        StringBuilder sb = new StringBuilder();
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        while (!q.isEmpty()) {
            TreeNode node = q.poll();
            if (node == null) { sb.append("null,"); continue; }
            sb.append(node.val).append(',');
```

```
            q.offer(node.le
ft);
            q.offer(node.right);
        }
        String[] parts = sb.toString().split(",");
        int last = parts.length - 1;
        while (last >= 0 && parts[last].equals("null")) last--;
        StringBuilder cleaned = new StringBuilder();
        for (int i = 0; i <= last; i++) cleaned.append(parts[i]).append(',');
        if (cleaned.length() > 0) cleaned.setLength(cleaned.length()-1);
        return cleaned.toString();
    }
    public static TreeNode deserialize(String data) {
        if (data == null || data.isEmpty()) return null;
        String[] parts = data.split(",");
        Queue<TreeNode> q = new LinkedList<>();
        TreeNode root = new TreeNode(Integer.parseInt(parts[0]));
        q.offer(root);
        int i = 1;
        while (!q.isEmpty() && i < parts.lengt
h) {
            TreeNode node = q.poll();
            if (i < parts.length) {
                String leftVal = parts[i++];
                if (!leftVal.equals("null")) { TreeNode left = new TreeNode(Integer.parseInt(leftVal)); node.left = left; q.offer(left); }
            }
            if (i < parts.length) {
                String rightVal = parts[i++];
                if (!rightVal.equals("null")) { TreeNode right = new TreeNode(Integer.parseInt(rightVal)); node.right = right; q.offer(right); }
            }
        }
        return root;
    }

    // (19) Sorted Array to BST
    public static TreeNode sortedArrayToBST(int[] nums) {
        if (nums == null || nums.length == 0) return null;
        return sortedArrayToBSTHelper(nums, 0, nums.length-1);
    }
    private static TreeNod
e sortedArrayToBSTHelper(int[] nums, int l, int r) {
        if (l > r) return null;
        int mid = l + (r-l)/2;
        TreeNode root = new TreeNode(nums[mid]);
        root.left = sortedArrayToBSTHelper(nums, l, mid-1);
        root.right = sortedArrayToBSTHelper(nums, mid+1, r);
        return root;
    }

    // (20) Find Min and Max in Binary Tree
    public static int findMin(TreeNode root) {
        if (root == null) throw new IllegalArgumentException("Tree is empty");
        int min = root.val;
        if (root.left != null) min = Math.min(min, findMin(root.left));
        if (root.right != null) min = Math.min(min, findMin(root.right));
        return min;
    }
    public static int findMax(TreeNode root) {
        if (root == null) throw new IllegalArgumentException("Tree is
 empty");
        int max = root.val;
        if (root.left != null) max = Math.max(max, findMax(root.left));
        if (root.right != null) max = Math.max(max, findMax(root.right));
        return max;
    }

    // ---------------------------- 21-30 (additional problems) ----------------------------

    // 21. Validate Binary Search Tree (BST)
    // Use min/max bounds passed down recursion. Use long to avoid int overflow on extremes.
    public static boolean isValidBST(TreeNode root) {
        return isValidBSTHelper(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }
```

```
    private static boolean isValidBSTHelper(TreeNode node, long min, long max) {
        if (node == null) return true;
        if (node.val <= min || node.val >= max) return false;
        return isValidBSTHelper(node.left,
 min, node.val) && isValidBSTHelper(node.right, node.val, max);
    }

    // 22. Kth Smallest Element in BST (iterative inorder)
    public static int kthSmallest(TreeNode root, int k) {
        Deque<TreeNode> stack = new ArrayDeque<>();
        TreeNode curr = root;
        while (curr != null || !stack.isEmpty()) {
            while (curr != null) { stack.push(curr); curr = curr.left; }
            curr = stack.pop();
            if (--k == 0) return curr.val;
            curr = curr.right;
        }
        throw new IllegalArgumentException("k is larger than number of nodes");
    }

    // 23. Invert / Mirror Binary Tree
    public static TreeNode invertTree(TreeNode root) {
        if (root == null) return null;
        TreeNode left = invertTree(root.left);
        TreeNode right
= invertTree(root.right);
        root.left = right;
        root.right = left;
        return root;
    }

    // 24. Flatten Binary Tree to Linked List (in-place, preorder)
    static TreeNode flattenPrev = null;
    public static void flatten(TreeNode root) {
        flattenPrev = null;
        flattenHelper(root);
    }
    private static void flattenHelper(TreeNode node) {
        if (node == null) return;
        flattenHelper(node.right);
        flattenHelper(node.left);
        node.right = flattenPrev;
        node.left = null;
        flattenPrev = node;
    }

    // 25. Recover Binary Search Tree (two nodes swapped)
    static TreeNode recoverFirst = null, recoverSecond = null, recoverPrev = null;
    public static void recoverTree(TreeNode root) {
        recoverFirst = recov
erSecond = recoverPrev = null;
        recoverDfs(root);
        if (recoverFirst != null && recoverSecond != null) {
            int tmp = recoverFirst.val;
            recoverFirst.val = recoverSecond.val;
            recoverSecond.val = tmp;
        }
    }
    private static void recoverDfs(TreeNode node) {
        if (node == null) return;
        recoverDfs(node.left);
        if (recoverPrev != null && node.val < recoverPrev.val) {
            if (recoverFirst == null) recoverFirst = recoverPrev;
            recoverSecond = node;
        }
        recoverPrev = node;
        recoverDfs(node.right);
    }

    // 26. Path Sum III (any downward path) - count paths equal target
    public static int pathSumIII(TreeNode root, int target) {
        Map<Integer, Integer> prefix = new Hash
Map<>();
```

```
        prefix.put(0, 1);
        return pathSumIIIHelper(root, 0, target, prefix);
    }
    private static int pathSumIIIHelper(TreeNode node, int curr, int target, Map<Integer,Integer> prefix) {
        if (node == null) return 0;
        curr += node.val;
        int res = prefix.getOrDefault(curr - target, 0);
        prefix.put(curr, prefix.getOrDefault(curr, 0) + 1);
        res += pathSumIIIHelper(node.left, curr, target, prefix);
        res += pathSumIIIHelper(node.right, curr, target, prefix);
        prefix.put(curr, prefix.get(curr) - 1);
        return res;
    }

    // 27. Count Univalue Subtrees
    static int univalueCount;
    public static int countUnivalSubtrees(TreeNode root) {
        univalueCount = 0;
        isUnival(root);
        return univalueCount;

    }
    private static boolean isUnival(TreeNode node) {
        if (node == null) return true;
        boolean left = isUnival(node.left);
        boolean right = isUnival(node.right);
        if (!left || !right) return false;
        if (node.left != null && node.left.val != node.val) return false;
        if (node.right != null && node.right.val != node.val) return false;
        univalueCount++;
        return true;
    }

    // 28. Construct Binary Tree from Preorder and Inorder
    static int preIndex;
    public static TreeNode buildTreePreIn(int[] preorder, int[] inorder) {
        preIndex = 0;
        Map<Integer, Integer> idx = new HashMap<>();
        for (int i = 0; i < inorder.length; i++) idx.put(inorder[i], i);
        return buildPreInHelper(preorder, 0, inorder.length

- 1, idx);
    }
    private static TreeNode buildPreInHelper(int[] preorder, int inL, int inR, Map<Integer,Integer> idx) {
        if (inL > inR) return null;
        int rootVal = preorder[preIndex++];
        TreeNode root = new TreeNode(rootVal);
        int pos = idx.get(rootVal);
        root.left = buildPreInHelper(preorder, inL, pos - 1, idx);
        root.right = buildPreInHelper(preorder, pos + 1, inR, idx);
        return root;
    }

    // 29. Morris Inorder Traversal (O(1) extra space)
    public static List<Integer> morrisInorder(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        TreeNode curr = root;
        while (curr != null) {
            if (curr.left == null) {
                res.add(curr.val);
                curr = curr.right;
            } els

e {
                TreeNode pred = curr.left;
                while (pred.right != null && pred.right != curr) pred = pred.right;
                if (pred.right == null) {
                    pred.right = curr;
                    curr = curr.left;
                } else {
                    pred.right = null;
                    res.add(curr.val);
                    curr = curr.right;
                }
            }
        }
        return res;
```

```
    }

    // 30. Convert BST to Sorted Doubly Linked List (in-place)
    // Reuse left as prev and right as next. Return head of doubly linked list.
    static TreeNode dllPrev = null;
    public static TreeNode bstToDoublyList(TreeNode root) {
        dllPrev = null;
        if (root == null) return null;
        TreeNode head = bstToDoublyLi

stHelper(root);
        // Move to head
        while (head != null && head.left != null) head = head.left;
        return head;
    }
    private static TreeNode bstToDoublyListHelper(TreeNode node) {
        if (node == null) return null;
        bstToDoublyListHelper(node.left);
        // link prev <-> node
        node.left = dllPrev;
        if (dllPrev != null) dllPrev.right = node;
        dllPrev = node;
        bstToDoublyListHelper(node.right);
        return node;
    }

    // ------------------------------ Helper: Build Sample Tree ------------------------------
    public static TreeNode buildSampleTree() {
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.l

eft.right = new TreeNode(5);
        root.right.left = new TreeNode(6);
        root.right.right = new TreeNode(7);
        return root;
    }

    // ------------------------------ Main: Quick demonstration ------------------------------
    public static void main(String[] args) {
        TreeNode root = buildSampleTree();
        System.out.println("Inorder recursive: " + inorderTraversal(root));
        System.out.println("Inorder iterative: " + inorderIterative(root));
        System.out.println("Preorder recursive: " + preorderTraversal(root));
        System.out.println("Postorder recursive: " + postorderTraversal(root));
        System.out.println("Level Order: " + levelOrder(root));
        System.out.println("Zigzag: " + zigzagLevelOrder(root));
        System.out.println("Height:

" + height(root));
        System.out.println("Diameter: " + diameter(root));
        System.out.println("Left view: " + leftView(root));
        System.out.println("Right view: " + rightView(root));
        System.out.println("Top view: " + topView(root));
        System.out.println("Bottom view: " + bottomView(root));
        System.out.println("Has path sum 8: " + hasPathSum(root, 8));
        System.out.println("All paths: " + allPaths(root));
        System.out.println("LCA(4,5): " + lowestCommonAncestor(root, root.left.left, root.left.right));
        System.out.println("Is balanced: " + isBalanced(root));
        System.out.println("Is symmetric example: " + isSymmetric(root.left)); // not symmetric but demo
        TreeNode sumRoot = new TreeNode(-10); sumRoot.left = new TreeNode(9

); sumRoot.right = new TreeNode(20);
        sumRoot.right.left = new TreeNode(15); sumRoot.right.right = new TreeNode(7);
        System.out.println("Max path sum example: " + maxPathSum(sumRoot));

        String ser = serialize(root);
        System.out.println("Serialized: " + ser);
        TreeNode deser = deserialize(ser);
        System.out.println("Deserialized level order: " + levelOrder(deser));

        int[] sorted = {-10,-3,0,5,9};
        TreeNode bst = sortedArrayToBST(sorted);
        System.out.println("Sorted array->BST inorder: " + inorderTraversal(bst));
        System.out.println("Min: " + findMin(root) + " Max: " + findMax(root));
```

```
        // 21: Validate BST (use bst built from sorted array)
        System.out.println("Is valid BST: " + isValidBST(bst));

        //
22: kth smallest
        System.out.println("Kth smallest (k=3) in BST: " + kthSmallest(bst, 3));

        // 23: invert tree
        TreeNode inv = invertTree(buildSampleTree());
        System.out.println("Inverted inorder: " + inorderTraversal(inv));

        // 24: flatten
        TreeNode flatSample = buildSampleTree();
        flatten(flatSample);
        System.out.print("Flattened list (right pointers): "); TreeNode cur = flatSample;
        while (cur != null) { System.out.print(cur.val + " "); cur = cur.right; }
        System.out.println();

        // 25: recover tree - create swapped BST
        TreeNode r = new TreeNode(3); r.left = new TreeNode(1); r.right = new TreeNode(4); r.right.left = new TreeNode(2);
        System.out.println("Before recover inorder: " + inorderTraver
sal(r));
        // swap values of two nodes to simulate error
        int tmp = r.val; r.val = r.right.left.val; r.right.left.val = tmp;
        System.out.println("After swap inorder: " + inorderTraversal(r));
        recoverTree(r);
        System.out.println("After recover inorder: " + inorderTraversal(r));

        // 26: Path Sum III
        TreeNode pSum = buildSampleTree(); System.out.println("PathSumIII target=3: " + pathSumIII(pSum, 3));

        // 27: Count univalue subtrees
        TreeNode u = new TreeNode(5); u.left = new TreeNode(1); u.right = new TreeNode(5); u.right.left = new TreeNode(5); u.right.right = new TreeNode(5);
        System.out.println("Univalue subtree count: " + countUnivalSubtrees(u));

        // 28: build from preorder & inorder
        int[] pre = {3,9,
20,15,7}; int[] in = {9,3,15,20,7};
        TreeNode built = buildTreePreIn(pre, in);
        System.out.println("Built tree inorder (should be inorder array): " + inorderTraversal(built));

        // 29: morris inorder
        System.out.println("Morris inorder on sample: " + morrisInorder(buildSampleTree()));

        // 30: bst to doubly linked list
        TreeNode dll = bstToDoublyList(bst);
        System.out.print("BST->DLL inorder forward: "); TreeNode h = dll; while (h!=null) { System.out.print(h.val+" "); h = h.right; } System.out.println();

        System.out.println("--- End of demo ---");
    }
}
```