

```

1
2  /*
3   * TreeProblems30.java
4   *
5   * Complete implementations for 30 binary tree problems commonly asked in interviews.
6   * Includes problem descriptions, approaches, complexity notes and example usages in
7   * main().
8   *
9   * Compile & run:
10  *   javac TreeProblems30.java && java TreeProblems30
11  *
12  * Notes:
13  * - This single file is for learning and quick testing. For production, split into
14  *   classes/tests.
15  */
16
17  import java.util.*;
18
19  public class TreeProblems30 {
20
21      // ----- Node Definition -----
22      static class TreeNode {
23          int val;
24          TreeNode left, right;
25          TreeNode(int val) { this.val = val; }
26          @Override public String toString() { return String.valueOf(val); }
27      }
28
29      // ----- 1-20 (from original) -----
30      // For brevity, I've included the already provided 20 problem implementations here.
31      // (1) Inorder (recursive & iterative)
32      public static List<Integer> inorderTraversal(TreeNode root) {
33          List<Integer> res = new ArrayList<>();
34          inorderHelper(root, res);
35          return res;
36      }
37
38      private static void inorderHelper(TreeNode node, List<Integer> res) {
39          if (node == null) return;
40          inorderHelper(node.left, res);
41          res.add(node.val);
42          inorderHelper(node.right, res);
43      }
44
45      public static List<Integer> inorderIterative(TreeNode root) {
46          List<Integer> res = new ArrayList<>();
47          Deque<TreeNode> stack = new ArrayDeque<>();
48          TreeNode curr = root;
49          while (curr != null || !stack.isEmpty()) {
50              while (curr != null) { stack.push(curr); curr = curr.left; }
51              curr = stack.pop();
52              res.add(curr.val);
53              curr = curr.right;
54          }
55          return res;
56      }
57
58      // (2) Preorder
59      public static List<Integer> preorderTraversal(TreeNode root) {
60          List<Integer> res = new ArrayList<>();
61          preorderHelper(root, res);
62          return res;
63      }
64
65      private static void preorderHelper(TreeNode node, List<Integer> res) {
66          if (node == null) return;
67          res.add(node.val);
68          preorderHelper(node.left, res);
69          preorderHelper(node.right, res);
70      }
71
72      public static List<Integer> preorderIterative(TreeNode root) {
73          List<Integer> res = new ArrayList<>();

```

```

68     if (root == null) return res;
69     Deque<TreeNode> stack = new ArrayDeque<>();
70     stack.push(root);
71     while (!stack.isEmpty()) {
72         TreeNode node = stack.pop();
73         res.add(node.val);
74         if (node.right != null) stack.push(node.right);
75         if (node.left != null) stack.push(node.left);
76     }
77     return res;
78 }
79
80 // (3) Postorder
81 public static List<Integer> postorderTraversal(TreeNode root) {
82     List<Integer> res = new ArrayList<>();
83     postorderHelper(root, res);
84     return res;
85 }
86 private static void postorderHelper(TreeNode node, List<Integer> res) {
87     if (node == null) return;
88     postorderHelper(node.left, res);
89     postorderHelper(node.right, res);
90     res.add(node.val);
91 }
92 public static List<Integer> postorderIterative(TreeNode root) {
93     List<Integer> res = new ArrayList<>();
94     if (root == null) return res;
95     Deque<TreeNode> stack = new ArrayDeque<>();
96     stack.push(root);
97     while (!stack.isEmpty()) {
98         TreeNode node = stack.pop();
99         res.add(node.val);
100         if (node.left != null) stack.push(node.left);
101         if (node.right != null) stack.push(node.right);
102     }
103     Collections.reverse(res);
104     return res;
105 }
106
107 // (4) Level Order
108 public static List<List<Integer>> levelOrder(TreeNode root) {
109     List<List<Integer>> res = new ArrayList<>();
110     if (root == null) return res;
111     Queue<TreeNode> q = new LinkedList<>();
112     q.offer(root);
113     while (!q.isEmpty()) {
114         int size = q.size();
115         List<Integer> level = new ArrayList<>();
116         for (int i = 0; i < size; i++) {
117             TreeNode node = q.poll();
118             level.add(node.val);
119             if (node.left != null) q.offer(node.left);
120             if (node.right != null) q.offer(node.right);
121         }
122         res.add(level);
123     }
124     return res;
125 }
126
127 // (5) Zigzag Level Order
128 public static List<List<Integer>> zigzagLevelOrder(TreeNode root) {
129     List<List<Integer>> res = new ArrayList<>();
130     if (root == null) return res;
131     Queue<TreeNode> q = new LinkedList<>();
132     q.offer(root);
133     boolean leftToRight = true;
134     while (!q.isEmpty()) {
135         int size = q.size();
136         LinkedList<Integer> level = new LinkedList<>();

```

```

137         for (int i = 0; i < size; i++) {
138             TreeNode node = q.poll();
139             if (leftToRight) level.addLast(node.val);
140             else level.addFirst(node.val);
141             if (node.left != null) q.offer(node.left);
142             if (node.right != null) q.offer(node.right);
143         }
144         res.add(level);
145         leftToRight = !leftToRight;
146     }
147     return res;
148 }
149
150 // (6) Height / Max Depth
151 public static int height(TreeNode root) {
152     if (root == null) return 0;
153     return 1 + Math.max(height(root.left), height(root.right));
154 }
155
156 // (7) Diameter (node count)
157 static int diameterAnswer;
158 public static int diameter(TreeNode root) {
159     diameterAnswer = 0;
160     diameterHelper(root);
161     return diameterAnswer;
162 }
163 private static int diameterHelper(TreeNode node) {
164     if (node == null) return 0;
165     int left = diameterHelper(node.left);
166     int right = diameterHelper(node.right);
167     diameterAnswer = Math.max(diameterAnswer, left + right + 1);
168     return 1 + Math.max(left, right);
169 }
170
171 // (8) Left View
172 public static List<Integer> leftView(TreeNode root) {
173     List<Integer> res = new ArrayList<>();
174     if (root == null) return res;
175     Queue<TreeNode> q = new LinkedList<>();
176     q.offer(root);
177     while (!q.isEmpty()) {
178         int size = q.size();
179         for (int i = 0; i < size; i++) {
180             TreeNode node = q.poll();
181             if (i == 0) res.add(node.val);
182             if (node.left != null) q.offer(node.left);
183             if (node.right != null) q.offer(node.right);
184         }
185     }
186     return res;
187 }
188
189 // (9) Right View
190 public static List<Integer> rightView(TreeNode root) {
191     List<Integer> res = new ArrayList<>();
192     if (root == null) return res;
193     Queue<TreeNode> q = new LinkedList<>();
194     q.offer(root);
195     while (!q.isEmpty()) {
196         int size = q.size();
197         for (int i = 0; i < size; i++) {
198             TreeNode node = q.poll();
199             if (i == size - 1) res.add(node.val);
200             if (node.left != null) q.offer(node.left);
201             if (node.right != null) q.offer(node.right);
202         }
203     }
204     return res;
205 }

```

```

206
207 // (10) Top View
208 static class PairNode { TreeNode node; int hd; PairNode(TreeNode n,int h){node=n;hd=h;
;} }
209 public static List<Integer> topView(TreeNode root) {
210     List<Integer> res = new ArrayList<>();
211     if (root == null) return res;
212     Map<Integer, Integer> map = new TreeMap<>();
213     Queue<PairNode> q = new LinkedList<>();
214     q.offer(new PairNode(root,0));
215     while (!q.isEmpty()) {
216         PairNode p = q.poll();
217         if (!map.containsKey(p.hd)) map.put(p.hd, p.node.val);
218         if (p.node.left != null) q.offer(new PairNode(p.node.left, p.hd-1));
219         if (p.node.right != null) q.offer(new PairNode(p.node.right, p.hd+1));
220     }
221     for (Integer v : map.values()) res.add(v);
222     return res;
223 }
224
225 // (11) Bottom View
226 public static List<Integer> bottomView(TreeNode root) {
227     List<Integer> res = new ArrayList<>();
228     if (root == null) return res;
229     Map<Integer, Integer> map = new TreeMap<>();
230     Queue<PairNode> q = new LinkedList<>();
231     q.offer(new PairNode(root,0));
232     while (!q.isEmpty()) {
233         PairNode p = q.poll();
234         map.put(p.hd, p.node.val);
235         if (p.node.left != null) q.offer(new PairNode(p.node.left, p.hd-1));
236         if (p.node.right != null) q.offer(new PairNode(p.node.right, p.hd+1));
237     }
238     for (Integer v : map.values()) res.add(v);
239     return res;
240 }
241
242 // (12) Has Path Sum (root-to-leaf)
243 public static boolean hasPathSum(TreeNode root, int targetSum) {
244     if (root == null) return false;
245     if (root.left == null && root.right == null) return root.val == targetSum;
246     int newSum = targetSum - root.val;
247     return hasPathSum(root.left, newSum) || hasPathSum(root.right, newSum);
248 }
249
250 // (13) All root-to-leaf paths
251 public static List<List<Integer>> allPaths(TreeNode root) {
252     List<List<Integer>> res = new ArrayList<>();
253     if (root == null) return res;
254     allPathsHelper(root, new ArrayList<>(), res);
255     return res;
256 }
257 private static void allPathsHelper(TreeNode node, List<Integer> path, List<List<
Integer>> res) {
258     if (node == null) return;
259     path.add(node.val);
260     if (node.left == null && node.right == null) res.add(new ArrayList<>(path));
261     else {
262         allPathsHelper(node.left, path, res);
263         allPathsHelper(node.right, path, res);
264     }
265     path.remove(path.size()-1);
266 }
267
268 // (14) Lowest Common Ancestor (general)
269 public static TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
270     if (root == null) return null;
271     if (root == p || root == q) return root;
272     TreeNode left = lowestCommonAncestor(root.left, p, q);

```

```

273         TreeNode right = lowestCommonAncestor(root.right, p, q);
274         if (left != null && right != null) return root;
275         return left != null ? left : right;
276     }
277
278     // (15) Is Balanced
279     public static boolean isBalanced(TreeNode root) { return checkHeight(root) != -1; }
280     private static int checkHeight(TreeNode node) {
281         if (node == null) return 0;
282         int left = checkHeight(node.left); if (left == -1) return -1;
283         int right = checkHeight(node.right); if (right == -1) return -1;
284         if (Math.abs(left-right) > 1) return -1;
285         return 1 + Math.max(left, right);
286     }
287
288     // (16) Is Symmetric
289     public static boolean isSymmetric(TreeNode root) {
290         if (root == null) return true;
291         return isMirror(root.left, root.right);
292     }
293     private static boolean isMirror(TreeNode a, TreeNode b) {
294         if (a == null && b == null) return true;
295         if (a == null || b == null) return false;
296         if (a.val != b.val) return false;
297         return isMirror(a.left, b.right) && isMirror(a.right, b.left);
298     }
299
300     // (17) Maximum Path Sum
301     static int maxPathSumAns;
302     public static int maxPathSum(TreeNode root) {
303         maxPathSumAns = Integer.MIN_VALUE;
304         maxPathSumHelper(root);
305         return maxPathSumAns;
306     }
307     private static int maxPathSumHelper(TreeNode node) {
308         if (node == null) return 0;
309         int left = Math.max(0, maxPathSumHelper(node.left));
310         int right = Math.max(0, maxPathSumHelper(node.right));
311         maxPathSumAns = Math.max(maxPathSumAns, node.val + left + right);
312         return node.val + Math.max(left, right);
313     }
314
315     // (18) Serialize / Deserialize (level-order)
316     public static String serialize(TreeNode root) {
317         if (root == null) return "";
318         StringBuilder sb = new StringBuilder();
319         Queue<TreeNode> q = new LinkedList<>();
320         q.offer(root);
321         while (!q.isEmpty()) {
322             TreeNode node = q.poll();
323             if (node == null) { sb.append("null,"); continue; }
324             sb.append(node.val).append(',');
325             q.offer(node.left);
326             q.offer(node.right);
327         }
328         String[] parts = sb.toString().split(",");
329         int last = parts.length - 1;
330         while (last >= 0 && parts[last].equals("null")) last--;
331         StringBuilder cleaned = new StringBuilder();
332         for (int i = 0; i <= last; i++) cleaned.append(parts[i]).append(',');
333         if (cleaned.length() > 0) cleaned.setLength(cleaned.length() - 1);
334         return cleaned.toString();
335     }
336     public static TreeNode deserialize(String data) {
337         if (data == null || data.isEmpty()) return null;
338         String[] parts = data.split(",");
339         Queue<TreeNode> q = new LinkedList<>();
340         TreeNode root = new TreeNode(Integer.parseInt(parts[0]));
341         q.offer(root);

```

```

342     int i = 1;
343     while (!q.isEmpty() && i < parts.length) {
344         TreeNode node = q.poll();
345         if (i < parts.length) {
346             String leftVal = parts[i++];
347             if (!leftVal.equals("null")) { TreeNode left = new TreeNode(Integer.
                parseInt(leftVal)); node.left = left; q.offer(left); }
348         }
349         if (i < parts.length) {
350             String rightVal = parts[i++];
351             if (!rightVal.equals("null")) { TreeNode right = new TreeNode(Integer.
                parseInt(rightVal)); node.right = right; q.offer(right); }
352         }
353     }
354     return root;
355 }
356
357 // (19) Sorted Array to BST
358 public static TreeNode sortedArrayToBST(int[] nums) {
359     if (nums == null || nums.length == 0) return null;
360     return sortedArrayToBSTHelper(nums, 0, nums.length-1);
361 }
362 private static TreeNode sortedArrayToBSTHelper(int[] nums, int l, int r) {
363     if (l > r) return null;
364     int mid = l + (r-l)/2;
365     TreeNode root = new TreeNode(nums[mid]);
366     root.left = sortedArrayToBSTHelper(nums, l, mid-1);
367     root.right = sortedArrayToBSTHelper(nums, mid+1, r);
368     return root;
369 }
370
371 // (20) Find Min and Max in Binary Tree
372 public static int findMin(TreeNode root) {
373     if (root == null) throw new IllegalArgumentException("Tree is empty");
374     int min = root.val;
375     if (root.left != null) min = Math.min(min, findMin(root.left));
376     if (root.right != null) min = Math.min(min, findMin(root.right));
377     return min;
378 }
379 public static int findMax(TreeNode root) {
380     if (root == null) throw new IllegalArgumentException("Tree is empty");
381     int max = root.val;
382     if (root.left != null) max = Math.max(max, findMax(root.left));
383     if (root.right != null) max = Math.max(max, findMax(root.right));
384     return max;
385 }
386
387 // ----- 21-30 (additional problems)
388 // -----
389 // 21. Validate Binary Search Tree (BST)
390 // Use min/max bounds passed down recursion. Use long to avoid int overflow on
391 // extremes.
392 public static boolean isValidBST(TreeNode root) {
393     return isValidBSTHelper(root, Long.MIN_VALUE, Long.MAX_VALUE);
394 }
395 private static boolean isValidBSTHelper(TreeNode node, long min, long max) {
396     if (node == null) return true;
397     if (node.val <= min || node.val >= max) return false;
398     return isValidBSTHelper(node.left, min, node.val) && isValidBSTHelper(node.right,
        node.val, max);
399 }
400 // 22. Kth Smallest Element in BST (iterative inorder)
401 public static int kthSmallest(TreeNode root, int k) {
402     Deque<TreeNode> stack = new ArrayDeque<>();
403     TreeNode curr = root;
404     while (curr != null || !stack.isEmpty()) {
405         while (curr != null) { stack.push(curr); curr = curr.left; }

```

```

406         curr = stack.pop();
407         if (--k == 0) return curr.val;
408         curr = curr.right;
409     }
410     throw new IllegalArgumentException("k is larger than number of nodes");
411 }
412
413 // 23. Invert / Mirror Binary Tree
414 public static TreeNode invertTree(TreeNode root) {
415     if (root == null) return null;
416     TreeNode left = invertTree(root.left);
417     TreeNode right = invertTree(root.right);
418     root.left = right;
419     root.right = left;
420     return root;
421 }
422
423 // 24. Flatten Binary Tree to Linked List (in-place, preorder)
424 static TreeNode flattenPrev = null;
425 public static void flatten(TreeNode root) {
426     flattenPrev = null;
427     flattenHelper(root);
428 }
429 private static void flattenHelper(TreeNode node) {
430     if (node == null) return;
431     flattenHelper(node.right);
432     flattenHelper(node.left);
433     node.right = flattenPrev;
434     node.left = null;
435     flattenPrev = node;
436 }
437
438 // 25. Recover Binary Search Tree (two nodes swapped)
439 static TreeNode recoverFirst = null, recoverSecond = null, recoverPrev = null;
440 public static void recoverTree(TreeNode root) {
441     recoverFirst = recoverSecond = recoverPrev = null;
442     recoverDfs(root);
443     if (recoverFirst != null && recoverSecond != null) {
444         int tmp = recoverFirst.val;
445         recoverFirst.val = recoverSecond.val;
446         recoverSecond.val = tmp;
447     }
448 }
449 private static void recoverDfs(TreeNode node) {
450     if (node == null) return;
451     recoverDfs(node.left);
452     if (recoverPrev != null && node.val < recoverPrev.val) {
453         if (recoverFirst == null) recoverFirst = recoverPrev;
454         recoverSecond = node;
455     }
456     recoverPrev = node;
457     recoverDfs(node.right);
458 }
459
460 // 26. Path Sum III (any downward path) - count paths equal target
461 public static int pathSumIII(TreeNode root, int target) {
462     Map<Integer, Integer> prefix = new HashMap<>();
463     prefix.put(0, 1);
464     return pathSumIIHelper(root, 0, target, prefix);
465 }
466 private static int pathSumIIHelper(TreeNode node, int curr, int target, Map<Integer,
Integer> prefix) {
467     if (node == null) return 0;
468     curr += node.val;
469     int res = prefix.getOrDefault(curr - target, 0);
470     prefix.put(curr, prefix.getOrDefault(curr, 0) + 1);
471     res += pathSumIIHelper(node.left, curr, target, prefix);
472     res += pathSumIIHelper(node.right, curr, target, prefix);
473     prefix.put(curr, prefix.get(curr) - 1);

```

```

474         return res;
475     }
476
477     // 27. Count Unival Subtrees
478     static int univalueCount;
479     public static int countUnivalSubtrees(TreeNode root) {
480         univalueCount = 0;
481         isUnival(root);
482         return univalueCount;
483     }
484     private static boolean isUnival(TreeNode node) {
485         if (node == null) return true;
486         boolean left = isUnival(node.left);
487         boolean right = isUnival(node.right);
488         if (!left || !right) return false;
489         if (node.left != null && node.left.val != node.val) return false;
490         if (node.right != null && node.right.val != node.val) return false;
491         univalueCount++;
492         return true;
493     }
494
495     // 28. Construct Binary Tree from Preorder and Inorder
496     static int preIndex;
497     public static TreeNode buildTreePreIn(int[] preorder, int[] inorder) {
498         preIndex = 0;
499         Map<Integer, Integer> idx = new HashMap<>();
500         for (int i = 0; i < inorder.length; i++) idx.put(inorder[i], i);
501         return buildPreInHelper(preorder, 0, inorder.length - 1, idx);
502     }
503     private static TreeNode buildPreInHelper(int[] preorder, int inL, int inR, Map<
Integer,Integer> idx) {
504         if (inL > inR) return null;
505         int rootVal = preorder[preIndex++];
506         TreeNode root = new TreeNode(rootVal);
507         int pos = idx.get(rootVal);
508         root.left = buildPreInHelper(preorder, inL, pos - 1, idx);
509         root.right = buildPreInHelper(preorder, pos + 1, inR, idx);
510         return root;
511     }
512
513     // 29. Morris Inorder Traversal (O(1) extra space)
514     public static List<Integer> morrisInorder(TreeNode root) {
515         List<Integer> res = new ArrayList<>();
516         TreeNode curr = root;
517         while (curr != null) {
518             if (curr.left == null) {
519                 res.add(curr.val);
520                 curr = curr.right;
521             } else {
522                 TreeNode pred = curr.left;
523                 while (pred.right != null && pred.right != curr) pred = pred.right;
524                 if (pred.right == null) {
525                     pred.right = curr;
526                     curr = curr.left;
527                 } else {
528                     pred.right = null;
529                     res.add(curr.val);
530                     curr = curr.right;
531                 }
532             }
533         }
534         return res;
535     }
536
537     // 30. Convert BST to Sorted Doubly Linked List (in-place)
538     // Reuse left as prev and right as next. Return head of doubly linked list.
539     static TreeNode dllPrev = null;
540     public static TreeNode bstToDoublyList(TreeNode root) {
541         dllPrev = null;

```



```

542         if (root == null) return null;
543         TreeNode head = bstToDoublyListHelper(root);
544         // Move to head
545         while (head != null && head.left != null) head = head.left;
546         return head;
547     }
548     private static TreeNode bstToDoublyListHelper(TreeNode node) {
549         if (node == null) return null;
550         bstToDoublyListHelper(node.left);
551         // link prev <-> node
552         node.left = dllPrev;
553         if (dllPrev != null) dllPrev.right = node;
554         dllPrev = node;
555         bstToDoublyListHelper(node.right);
556         return node;
557     }
558
559     // ----- Helper: Build Sample Tree
560     -----
561     public static TreeNode buildSampleTree() {
562         TreeNode root = new TreeNode(1);
563         root.left = new TreeNode(2);
564         root.right = new TreeNode(3);
565         root.left.left = new TreeNode(4);
566         root.left.right = new TreeNode(5);
567         root.right.left = new TreeNode(6);
568         root.right.right = new TreeNode(7);
569         return root;
570     }
571
572     // ----- Main: Quick demonstration
573     -----
574     public static void main(String[] args) {
575         TreeNode root = buildSampleTree();
576         System.out.println("Inorder recursive: " + inorderTraversal(root));
577         System.out.println("Inorder iterative: " + inorderIterative(root));
578         System.out.println("Preorder recursive: " + preorderTraversal(root));
579         System.out.println("Postorder recursive: " + postorderTraversal(root));
580         System.out.println("Level Order: " + levelOrder(root));
581         System.out.println("Zigzag: " + zigzagLevelOrder(root));
582         System.out.println("Height: " + height(root));
583         System.out.println("Diameter: " + diameter(root));
584         System.out.println("Left view: " + leftView(root));
585         System.out.println("Right view: " + rightView(root));
586         System.out.println("Top view: " + topView(root));
587         System.out.println("Bottom view: " + bottomView(root));
588         System.out.println("Has path sum 8: " + hasPathSum(root, 8));
589         System.out.println("All paths: " + allPaths(root));
590         System.out.println("LCA(4,5): " + lowestCommonAncestor(root, root.left.left, root
591             .left.right));
592         System.out.println("Is balanced: " + isBalanced(root));
593         System.out.println("Is symmetric example: " + isSymmetric(root.left)); // not
594             symmetric but demo
595         TreeNode sumRoot = new TreeNode(-10); sumRoot.left = new TreeNode(9); sumRoot.
596             right = new TreeNode(20);
597         sumRoot.right.left = new TreeNode(15); sumRoot.right.right = new TreeNode(7);
598         System.out.println("Max path sum example: " + maxPathSum(sumRoot));
599
600         String ser = serialize(root);
601         System.out.println("Serialized: " + ser);
602         TreeNode deser = deserialize(ser);
603         System.out.println("Deserialized level order: " + levelOrder(deser));
604
605         int[] sorted = {-10,-3,0,5,9};
606         TreeNode bst = sortedArrayToBST(sorted);
607         System.out.println("Sorted array->BST inorder: " + inorderTraversal(bst));
608         System.out.println("Min: " + findMin(root) + " Max: " + findMax(root));
609
610         // 21: Validate BST (use bst built from sorted array)

```

```

606         System.out.println("Is valid BST: " + isValidBST(bst));
607
608         // 22: kth smallest
609         System.out.println("Kth smallest (k=3) in BST: " + kthSmallest(bst, 3));
610
611         // 23: invert tree
612         TreeNode inv = invertTree(buildSampleTree());
613         System.out.println("Inverted inorder: " + inorderTraversal(inv));
614
615         // 24: flatten
616         TreeNode flatSample = buildSampleTree();
617         flatten(flatSample);
618         System.out.print("Flattened list (right pointers): ");
619         while (cur != null) { System.out.print(cur.val + " "); cur = cur.right; }
620         System.out.println();
621
622         // 25: recover tree - create swapped BST
623         TreeNode r = new TreeNode(3); r.left = new TreeNode(1); r.right = new TreeNode(4);
624         r.right.left = new TreeNode(2);
625         System.out.println("Before recover inorder: " + inorderTraversal(r));
626         // swap values of two nodes to simulate error
627         int tmp = r.val; r.val = r.right.left.val; r.right.left.val = tmp;
628         System.out.println("After swap inorder: " + inorderTraversal(r));
629         recoverTree(r);
630         System.out.println("After recover inorder: " + inorderTraversal(r));
631
632         // 26: Path Sum III
633         TreeNode pSum = buildSampleTree();
634         System.out.println("PathSumIII target=3: " + pathSumIII(pSum, 3));
635
636         // 27: Count univalue subtrees
637         TreeNode u = new TreeNode(5); u.left = new TreeNode(1); u.right = new TreeNode(5);
638         u.right.left = new TreeNode(5); u.right.right = new TreeNode(5);
639         System.out.println("Univalue subtree count: " + countUnivalSubtrees(u));
640
641         // 28: build from preorder & inorder
642         int[] pre = {3,9,20,15,7}; int[] in = {9,3,15,20,7};
643         TreeNode built = buildTreePreIn(pre, in);
644         System.out.println("Built tree inorder (should be inorder array): " + inorderTraversal(built));
645
646         // 29: morris inorder
647         System.out.println("Morris inorder on sample: " + morrisInorder(buildSampleTree()));
648
649         // 30: bst to doubly linked list
650         TreeNode dll = bstToDoublyList(bst);
651         System.out.print("BST->DLL inorder forward: ");
652         while (h != null) { System.out.print(h.val + " "); h = h.right; }
653         System.out.println();
654
655         System.out.println("--- End of demo ---");
656     }
657 }

```