

```

1  /**
2  *  1. **Test if a Binary Tree is Height-Balanced**
3
4  public boolean isBalanced(TreeNode root) {
5      return checkHeight(root) != -1;
6  }
7
8  private int checkHeight(TreeNode node) {
9      if (node == null) return 0;
10     int left = checkHeight(node.left);
11     int right = checkHeight(node.right);
12     if (left == -1 || right == -1 || Math.abs(left - right) > 1) return -1;
13     return Math.max(left, right) + 1;
14 }
15
16 /**
17 *  2. **Test if a Binary Tree is Symmetric**
18
19 public boolean isSymmetric(TreeNode root) {
20     return root == null || isMirror(root.left, root.right);
21 }
22
23 private boolean isMirror(TreeNode t1, TreeNode t2) {
24     if (t1 == null || t2 == null) return t1 == t2;
25     return t1.val == t2.val && isMirror(t1.left, t2.right) && isMirror(t1.right, t2.left);
26 }
27
28 /**
29 *  3. **Compute the Lowest Common Ancestor in a Binary Tree**
30
31 public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
32     if (root == null || root == p || root == q) return root;
33     TreeNode left = lowestCommonAncestor(root.left, p, q);
34     TreeNode right = lowestCommonAncestor(root.right, p, q);
35     return left == null ? right : right == null ? left : root;
36 }
37
38 /**
39 *  4. **LCA with Parent Pointers**
40
41 public TreeNode getLCA(TreeNode a, TreeNode b) {
42     Set<TreeNode> ancestors = new HashSet<>();
43     while (a != null) {
44         ancestors.add(a);
45         a = a.parent;
46     }
47     while (b != null) {
48         if (ancestors.contains(b)) return b;
49         b = b.parent;
50     }
51     return null;
52 }
53
54 /**
55 *  5. **Sum of Root-to-Leaf Paths**
56
57 public int sumNumbers(TreeNode root) {
58     return dfs(root, 0);
59 }
60
61 private int dfs(TreeNode node, int sum) {
62     if (sum = sum * 10 + node.val;
63     if (node.left == null && node.right == null) return sum;
64     return dfs(node.left, sum) + dfs(node.right, sum);
65 }
66
67
68 /**
69 *  6. **Find Root-to-Leaf Path with Specified Sum**

```

```

69
70
71 public boolean hasPathSum(TreeNode root, int sum) {
72     if (root == null) return false;
73     if (root.left == null && root.right == null) return root.val == sum;
74     return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);
75 }
76
77
78 Recursion**
79
80 public List<Integer> inorderTraversal(TreeNode root) {
81     List<Integer> res = new ArrayList<>();
82     Stack<TreeNode> stack = new Stack<>();
83     TreeNode curr = root;
84     while (curr != null || !stack.isEmpty()) {
85         while (curr != null) {
86             stack.push(curr);
87             curr = curr.left;
88         }
89         curr = stack.pop();
90         res.add(curr.val);
91         curr = curr.right;
92     }
93     return res;
94 }
95
96
97 ### ☑ 8. **Preorder Traversal Without Recursion**
98 public List<Integer> preorderTraversal(TreeNode root) {
99     List<Integer> res = new ArrayList<>();
100    Stack<TreeNode> stack = new Stack<>();
101    if (root != null) stack.push(root);
102    while (!stack.isEmpty()) {
103        TreeNode node = stack.pop();
104        res.add(node.val);
105        if (node.right != null) stack.push(node.right);
106        if (node.left != null) stack.push(node.left);
107    }
108    return res;
109 }
110
111
112 ### ☑ 9. **Compute k-th Node in Inorder Traversal**
113 public TreeNode kthNode(TreeNode root, int k) {
114     Stack<TreeNode> stack = new Stack<>();
115     TreeNode curr = root;
116     while (curr != null || !stack.isEmpty()) {
117         while (curr != null) {
118             stack.push(curr);
119             curr = curr.left;
120         }
121         curr = stack.pop();
122         if (--k == 0) return curr;
123         curr = curr.right;
124     }
125     return null;
126 }
127
128
129 ### ☑ 10. **Compute Successor in BST**
130 public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
131     TreeNode succ = null;
132     while (root != null) {
133         if (p.val < root.val) {
134             succ = root;
135             root = root.left;
136         } else {

```

```

137         root = root.right;
138     }
139 }
140 return succ;
141 }
142
143 ### ☐ 11. **Inorder Traversal with O(1) Space (Morris Traversal)**
144
145 public List<Integer> morrisTraversal(TreeNode root) {
146     List<Integer> res = new ArrayList<>();
147     TreeNode curr = root;
148     while (curr != null) {
149         if (curr.left == null) {
150             res.add(curr.val);
151             curr = curr.right;
152         } else {
153             TreeNode pred = curr.left;
154             while (pred.right != null && pred.right != curr)
155                 pred = pred.right;
156             if (pred.right == null) {
157                 pred.right = curr;
158                 curr = curr.left;
159             } else {
160                 pred.right = null;
161                 res.add(curr.val);
162                 curr = curr.right;
163             }
164         }
165     }
166     return res;
167 }
168
169
170
171 ### ☐ 12. **Reconstruct a Binary Tree from Inorder and Preorder Traversal**
172
173
174 public TreeNode buildTree(int[] preorder, int[] inorder) {
175     Map<Integer, Integer> inMap = new HashMap<>();
176     for (int i = 0; i < inorder.length; i++)
177         inMap.put(inorder[i], i);
178     return build(preorder, 0, preorder.length - 1, 0, inMap);
179 }
180
181
182 private TreeNode build(int[] preorder, int preStart, int preEnd, int inStart, Map<Integer
183 , Integer> inMap) {
184     if (preStart > preEnd) return null;
185     TreeNode root = new TreeNode(preorder[preStart]);
186     int inIndex = inMap.get(root.val);
187     int leftSize = inIndex - inStart;
188     root.left = build(preorder, preStart + 1, preStart + leftSize, inStart, inMap);
189     root.right = build(preorder, preStart + leftSize + 1, preEnd, inIndex + 1, inMap);
190     return root;
191 }
192
193
194
195 int index = 0;
196
197 public TreeNode buildTreeWithMarkers(String[] preorder) {
198     if (index >= preorder.length || preorder[index].equals("#")) {
199         index++;
200         return null;
201     }
202     TreeNode node = new TreeNode(Integer.parseInt(preorder[index++]));
203     node.left = buildTreeWithMarkers(preorder);
204     node.right = buildTreeWithMarkers(preorder);

```

```

205     return node;
206 }
207
208
209 ###  14. **Form a Linked List from the Leaves of a Binary Tree**
210
211
212 public List<TreeNode> leafList(TreeNode root) {
213     List<TreeNode> leaves = new ArrayList<>();
214     collectLeaves(root, leaves);
215     return leaves;
216 }
217
218 private void collectLeaves(TreeNode node, List<TreeNode> leaves) {
219     if (node == null) return;
220     if (node.left == null && node.right == null) {
221         leaves.add(node);
222         return;
223     }
224     collectLeaves(node.left, leaves);
225     collectLeaves(node.right, leaves);
226 }
227
228
229 ###  15. **Compute the Exterior of a Binary Tree**
230
231
232 public List<TreeNode> exteriorBinaryTree(TreeNode root) {
233     List<TreeNode> result = new ArrayList<>();
234     if (root == null) return result;
235     result.add(root);
236     leftBoundary(root.left, result);
237     leaves(root.left, result);
238     leaves(root.right, result);
239     rightBoundary(root.right, result);
240     return result;
241 }
242
243 private void leftBoundary(TreeNode node, List<TreeNode> res) {
244     while (node != null) {
245         if (node.left != null || node.right != null) res.add(node);
246         node = (node.left != null) ? node.left : node.right;
247     }
248 }
249
250 private void rightBoundary(TreeNode node, List<TreeNode> res) {
251     Stack<TreeNode> stack = new Stack<>();
252     while (node != null) {
253         if (node.left != null || node.right != null) stack.push(node);
254         node = (node.right != null) ? node.right : node.left;
255     }
256     while (!stack.isEmpty()) res.add(stack.pop());
257 }
258
259 private void leaves(TreeNode node, List<TreeNode> res) {
260     if (node == null) return;
261     if (node.left == null && node.right == null) {
262         res.add(node);
263         return;
264     }
265     leaves(node.left, res);
266     leaves(node.right, res);
267 }
268
269
270 ###  16. **Compute the Right Sibling Tree (Next Right Pointer)**
271
272
273 public void connect(TreeNode root) {

```

```

274     if (root == null) return;
275     Queue<TreeNode> queue = new LinkedList<>();
276     queue.offer(root);
277     while (!queue.isEmpty()) {
278         int size = queue.size();
279         TreeNode prev = null;
280         for (int i = 0; i < size; i++) {
281             TreeNode curr = queue.poll();
282             if (prev != null) prev.next = curr;
283             prev = curr;
284             if (curr.left != null) queue.offer(curr.left);
285             if (curr.right != null) queue.offer(curr.right);
286         }
287     }
288 }
289
290
291 ###  17. **Implement Locking in a Binary Tree**
292
293
294 class LockableTreeNode {
295     int val;
296     LockableTreeNode left, right, parent;
297     boolean isLocked = false;
298     int lockedDescendants = 0;
299
300     public boolean isLocked() {
301         return isLocked;
302     }
303
304     public boolean lock() {
305         if (isLocked || lockedDescendants > 0 || hasLockedAncestor()) return false;
306         isLocked = true;
307         updateAncestors(1);
308         return true;
309     }
310
311     public boolean unlock() {
312         if (!isLocked) return false;
313         isLocked = false;
314         updateAncestors(-1);
315         return true;
316     }
317
318     private boolean hasLockedAncestor() {
319         LockableTreeNode curr = parent;
320         while (curr != null) {
321             if (curr.isLocked) return true;
322             curr = curr.parent;
323         }
324         return false;
325     }
326
327     private void updateAncestors(int delta) {
328         LockableTreeNode curr = parent;
329         while (curr != null) {
330             curr.lockedDescendants += delta;
331             curr = curr.parent;
332         }
333     }
334 }
335
336
337
338
339 /*
340 * TreeProblems30.java
341 *
342 * Complete implementations for 30 binary tree problems commonly asked in interviews.

```

```

343 */
344
345 import java.util.*;
346
347 public class TreeProblems30 {
348
349     // ----- Node Definition -----
350     static class TreeNode {
351         int val;
352         TreeNode left, right;
353         TreeNode(int val) { this.val = val; }
354         @Override public String toString() { return String.valueOf(val); }
355     }
356
357     // ----- 1-20 (from original) -----
358     // For brevity, I've included the already provided 20 problem implementations here.
359     // (1) Inorder (recursive & iterative)
360     public static List<Integer> inorderTraversal(TreeNode root) {
361         List<Integer> res = new ArrayList<>();
362         inorderHelper(root, res);
363         return res;
364     }
365     private static void inorderHelper(TreeNode node, List<Integer> res) {
366         if (node == null) return;
367         inorderHelper(node.left, res);
368         res.add(node.val);
369         inorderHelper(node.right, res);
370     }
371     public static List<Integer> inorderIterative(TreeNode root) {
372         List<Integer> res = new ArrayList<>();
373         Deque<TreeNode> stack = new ArrayDeque<>();
374         TreeNode curr = root;
375         while (curr != null || !stack.isEmpty()) {
376             while (curr != null) { stack.push(curr); curr = curr.left; }
377             curr = stack.pop();
378             res.add(curr.val);
379             curr = curr.right;
380         }
381         return res;
382     }
383
384     // (2) Preorder
385     public static List<Integer> preorderTraversal(TreeNode root) {
386         List<Integer> res = new ArrayList<>();
387         preorderHelper(root, res);
388         return res;
389     }
390     private static void preorderHelper(TreeNode node, List<Integer> res) {
391         if (node == null) return;
392         res.add(node.val);
393         preorderHelper(node.left, res);
394         preorderHelper(node.right, res);
395     }
396     public static List<Integer> preorderIterative(TreeNode root) {
397         List<Integer> res = new ArrayList<>();
398         if (root == null) return res;
399         Deque<TreeNode> stack = new ArrayDeque<>();
400         stack.push(root);
401         while (!stack.isEmpty()) {
402             TreeNode node = stack.pop();
403             res.add(node.val);
404             if (node.right != null) stack.push(node.right);
405             if (node.left != null) stack.push(node.left);
406         }
407         return res;
408     }
409
410     // (3) Postorder
411     public static List<Integer> postorderTraversal(TreeNode root) {

```

```

412     List<Integer> res = new ArrayList<>();
413     postorderHelper(root, res);
414     return res;
415 }
416 private static void postorderHelper(TreeNode node, List<Integer> res) {
417     if (node == null) return;
418     postorderHelper(node.left, res);
419     postorderHelper(node.right, res);
420     res.add(node.val);
421 }
422 public static List<Integer> postorderIterative(TreeNode root) {
423     List<Integer> res = new ArrayList<>();
424     if (root == null) return res;
425     Deque<TreeNode> stack = new ArrayDeque<>();
426     stack.push(root);
427     while (!stack.isEmpty()) {
428         TreeNode node = stack.pop();
429         res.add(node.val);
430         if (node.left != null) stack.push(node.left);
431         if (node.right != null) stack.push(node.right);
432     }
433     Collections.reverse(res);
434     return res;
435 }
436
437 // (4) Level Order
438 public static List<List<Integer>> levelOrder(TreeNode root) {
439     List<List<Integer>> res = new ArrayList<>();
440     if (root == null) return res;
441     Queue<TreeNode> q = new LinkedList<>();
442     q.offer(root);
443     while (!q.isEmpty()) {
444         int size = q.size();
445         List<Integer> level = new ArrayList<>();
446         for (int i = 0; i < size; i++) {
447             TreeNode node = q.poll();
448             level.add(node.val);
449             if (node.left != null) q.offer(node.left);
450             if (node.right != null) q.offer(node.right);
451         }
452         res.add(level);
453     }
454     return res;
455 }
456
457 // (5) Zigzag Level Order
458 public static List<List<Integer>> zigzagLevelOrder(TreeNode root) {
459     List<List<Integer>> res = new ArrayList<>();
460     if (root == null) return res;
461     Queue<TreeNode> q = new LinkedList<>();
462     q.offer(root);
463     boolean leftToRight = true;
464     while (!q.isEmpty()) {
465         int size = q.size();
466         LinkedList<Integer> level = new LinkedList<>();
467         for (int i = 0; i < size; i++) {
468             TreeNode node = q.poll();
469             if (leftToRight) level.addLast(node.val);
470             else level.addFirst(node.val);
471             if (node.left != null) q.offer(node.left);
472             if (node.right != null) q.offer(node.right);
473         }
474         res.add(level);
475         leftToRight = !leftToRight;
476     }
477     return res;
478 }
479
480 // (6) Height / Max Depth

```

```

481     public static int height(TreeNode root) {
482         if (root == null) return 0;
483         return 1 + Math.max(height(root.left), height(root.right));
484     }
485
486     // (7) Diameter (node count)
487     static int diameterAnswer;
488     public static int diameter(TreeNode root) {
489         diameterAnswer = 0;
490         diameterHelper(root);
491         return diameterAnswer;
492     }
493     private static int diameterHelper(TreeNode node) {
494         if (node == null) return 0;
495         int left = diameterHelper(node.left);
496         int right = diameterHelper(node.right);
497         diameterAnswer = Math.max(diameterAnswer, left + right + 1);
498         return 1 + Math.max(left, right);
499     }
500
501     // (8) Left View
502     public static List<Integer> leftView(TreeNode root) {
503         List<Integer> res = new ArrayList<>();
504         if (root == null) return res;
505         Queue<TreeNode> q = new LinkedList<>();
506         q.offer(root);
507         while (!q.isEmpty()) {
508             int size = q.size();
509             for (int i = 0; i < size; i++) {
510                 TreeNode node = q.poll();
511                 if (i == 0) res.add(node.val);
512                 if (node.left != null) q.offer(node.left);
513                 if (node.right != null) q.offer(node.right);
514             }
515         }
516         return res;
517     }
518
519     // (9) Right View
520     public static List<Integer> rightView(TreeNode root) {
521         List<Integer> res = new ArrayList<>();
522         if (root == null) return res;
523         Queue<TreeNode> q = new LinkedList<>();
524         q.offer(root);
525         while (!q.isEmpty()) {
526             int size = q.size();
527             for (int i = 0; i < size; i++) {
528                 TreeNode node = q.poll();
529                 if (i == size - 1) res.add(node.val);
530                 if (node.left != null) q.offer(node.left);
531                 if (node.right != null) q.offer(node.right);
532             }
533         }
534         return res;
535     }
536
537     // (10) Top View
538     static class PairNode { TreeNode node; int hd; PairNode(TreeNode n, int h) {node=n;hd=h} }
539     public static List<Integer> topView(TreeNode root) {
540         List<Integer> res = new ArrayList<>();
541         if (root == null) return res;
542         Map<Integer, Integer> map = new TreeMap<>();
543         Queue<PairNode> q = new LinkedList<>();
544         q.offer(new PairNode(root, 0));
545         while (!q.isEmpty()) {
546             PairNode p = q.poll();
547             if (!map.containsKey(p.hd)) map.put(p.hd, p.node.val);
548             if (p.node.left != null) q.offer(new PairNode(p.node.left, p.hd-1));

```

```

549         if (p.node.right != null) q.offer(new PairNode(p.node.right, p.hd+1));
550     }
551     for (Integer v : map.values()) res.add(v);
552     return res;
553 }
554
555 // (11) Bottom View
556 public static List<Integer> bottomView(TreeNode root) {
557     List<Integer> res = new ArrayList<>();
558     if (root == null) return res;
559     Map<Integer, Integer> map = new TreeMap<>();
560     Queue<PairNode> q = new LinkedList<>();
561     q.offer(new PairNode(root, 0));
562     while (!q.isEmpty()) {
563         PairNode p = q.poll();
564         map.put(p.hd, p.node.val);
565         if (p.node.left != null) q.offer(new PairNode(p.node.left, p.hd-1));
566         if (p.node.right != null) q.offer(new PairNode(p.node.right, p.hd+1));
567     }
568     for (Integer v : map.values()) res.add(v);
569     return res;
570 }
571
572 // (12) Has Path Sum (root-to-leaf)
573 public static boolean hasPathSum(TreeNode root, int targetSum) {
574     if (root == null) return false;
575     if (root.left == null && root.right == null) return root.val == targetSum;
576     int newSum = targetSum - root.val;
577     return hasPathSum(root.left, newSum) || hasPathSum(root.right, newSum);
578 }
579
580 // (13) All root-to-leaf paths
581 public static List<List<Integer>> allPaths(TreeNode root) {
582     List<List<Integer>> res = new ArrayList<>();
583     if (root == null) return res;
584     allPathsHelper(root, new ArrayList<>(), res);
585     return res;
586 }
587 private static void allPathsHelper(TreeNode node, List<Integer> path, List<List<Integer>> res) {
588     if (node == null) return;
589     path.add(node.val);
590     if (node.left == null && node.right == null) res.add(new ArrayList<>(path));
591     else {
592         allPathsHelper(node.left, path, res);
593         allPathsHelper(node.right, path, res);
594     }
595     path.remove(path.size()-1);
596 }
597
598 // (14) Lowest Common Ancestor (general)
599 public static TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
600     if (root == null) return null;
601     if (root == p || root == q) return root;
602     TreeNode left = lowestCommonAncestor(root.left, p, q);
603     TreeNode right = lowestCommonAncestor(root.right, p, q);
604     if (left != null && right != null) return root;
605     return left != null ? left : right;
606 }
607
608 // (15) Is Balanced
609 public static boolean isBalanced(TreeNode root) { return checkHeight(root) != -1; }
610 private static int checkHeight(TreeNode node) {
611     if (node == null) return 0;
612     int left = checkHeight(node.left); if (left == -1) return -1;
613     int right = checkHeight(node.right); if (right == -1) return -1;
614     if (Math.abs(left-right) > 1) return -1;
615     return 1 + Math.max(left,right);
616 }

```

```

617
618 // (16) Is Symmetric
619 public static boolean isSymmetric(TreeNode root) {
620     if (root == null) return true;
621     return isMirror(root.left, root.right);
622 }
623 private static boolean isMirror(TreeNode a, TreeNode b) {
624     if (a == null && b == null) return true;
625     if (a == null || b == null) return false;
626     if (a.val != b.val) return false;
627     return isMirror(a.left, b.right) && isMirror(a.right, b.left);
628 }
629
630 // (17) Maximum Path Sum
631 static int maxPathSumAns;
632 public static int maxPathSum(TreeNode root) {
633     maxPathSumAns = Integer.MIN_VALUE;
634     maxPathSumHelper(root);
635     return maxPathSumAns;
636 }
637 private static int maxPathSumHelper(TreeNode node) {
638     if (node == null) return 0;
639     int left = Math.max(0, maxPathSumHelper(node.left));
640     int right = Math.max(0, maxPathSumHelper(node.right));
641     maxPathSumAns = Math.max(maxPathSumAns, node.val + left + right);
642     return node.val + Math.max(left, right);
643 }
644
645 // (18) Serialize / Deserialize (level-order)
646 public static String serialize(TreeNode root) {
647     if (root == null) return "";
648     StringBuilder sb = new StringBuilder();
649     Queue<TreeNode> q = new LinkedList<>();
650     q.offer(root);
651     while (!q.isEmpty()) {
652         TreeNode node = q.poll();
653         if (node == null) { sb.append("null,"); continue; }
654         sb.append(node.val).append(',', ',');
655         q.offer(node.left);
656         q.offer(node.right);
657     }
658     String[] parts = sb.toString().split(",");
659     int last = parts.length - 1;
660     while (last >= 0 && parts[last].equals("null")) last--;
661     StringBuilder cleaned = new StringBuilder();
662     for (int i = 0; i <= last; i++) cleaned.append(parts[i]).append(',');
663     if (cleaned.length() > 0) cleaned.setLength(cleaned.length() - 1);
664     return cleaned.toString();
665 }
666 public static TreeNode deserialize(String data) {
667     if (data == null || data.isEmpty()) return null;
668     String[] parts = data.split(",");
669     Queue<TreeNode> q = new LinkedList<>();
670     TreeNode root = new TreeNode(Integer.parseInt(parts[0]));
671     q.offer(root);
672     int i = 1;
673     while (!q.isEmpty() && i < parts.length) {
674         TreeNode node = q.poll();
675         if (i < parts.length) {
676             String leftVal = parts[i++];
677             if (!leftVal.equals("null")) { TreeNode left = new TreeNode(Integer.
678                                         parseInt(leftVal)); node.left = left; q.offer(left); }
679         }
680         if (i < parts.length) {
681             String rightVal = parts[i++];
682             if (!rightVal.equals("null")) { TreeNode right = new TreeNode(Integer.
683                                         parseInt(rightVal)); node.right = right; q.offer(right); }
684         }
685     }
686 }

```

```

684     return root;
685 }
686
687 // (19) Sorted Array to BST
688 public static TreeNode sortedArrayToBST(int[] nums) {
689     if (nums == null || nums.length == 0) return null;
690     return sortedArrayToBSTHelper(nums, 0, nums.length-1);
691 }
692 private static TreeNode sortedArrayToBSTHelper(int[] nums, int l, int r) {
693     if (l > r) return null;
694     int mid = l + (r-1)/2;
695     TreeNode root = new TreeNode(nums[mid]);
696     root.left = sortedArrayToBSTHelper(nums, l, mid-1);
697     root.right = sortedArrayToBSTHelper(nums, mid+1, r);
698     return root;
699 }
700
701 // (20) Find Min and Max in Binary Tree
702 public static int findMin(TreeNode root) {
703     if (root == null) throw new IllegalArgumentException("Tree is empty");
704     int min = root.val;
705     if (root.left != null) min = Math.min(min, findMin(root.left));
706     if (root.right != null) min = Math.min(min, findMin(root.right));
707     return min;
708 }
709 public static int findMax(TreeNode root) {
710     if (root == null) throw new IllegalArgumentException("Tree is empty");
711     int max = root.val;
712     if (root.left != null) max = Math.max(max, findMax(root.left));
713     if (root.right != null) max = Math.max(max, findMax(root.right));
714     return max;
715 }
716
717 // ----- 21-30 (additional problems)
-----
```

718

```

719 // 21. Validate Binary Search Tree (BST)
720 // Use min/max bounds passed down recursion. Use long to avoid int overflow on
721 // extremes.
722 public static boolean isValidBST(TreeNode root) {
723     return isValidBSTHelper(root, Long.MIN_VALUE, Long.MAX_VALUE);
724 }
725 private static boolean isValidBSTHelper(TreeNode node, long min, long max) {
726     if (node == null) return true;
727     if (node.val <= min || node.val >= max) return false;
728     return isValidBSTHelper(node.left, min, node.val) && isValidBSTHelper(node.right,
729         node.val, max);
730 }
731
732 // 22. Kth Smallest Element in BST (iterative inorder)
733 public static int kthSmallest(TreeNode root, int k) {
734     Deque<TreeNode> stack = new ArrayDeque<>();
735     TreeNode curr = root;
736     while (curr != null || !stack.isEmpty()) {
737         while (curr != null) { stack.push(curr); curr = curr.left; }
738         curr = stack.pop();
739         if (--k == 0) return curr.val;
740         curr = curr.right;
741     }
742     throw new IllegalArgumentException("k is larger than number of nodes");
743 }
744
745 // 23. Invert / Mirror Binary Tree
746 public static TreeNode invertTree(TreeNode root) {
747     if (root == null) return null;
748     TreeNode left = invertTree(root.left);
749     TreeNode right = invertTree(root.right);
750     root.left = right;
751     root.right = left;
```

```

750     return root;
751 }
752
753 // 24. Flatten Binary Tree to Linked List (in-place, preorder)
754 static TreeNode flattenPrev = null;
755 public static void flatten(TreeNode root) {
756     flattenPrev = null;
757     flattenHelper(root);
758 }
759 private static void flattenHelper(TreeNode node) {
760     if (node == null) return;
761     flattenHelper(node.right);
762     flattenHelper(node.left);
763     node.right = flattenPrev;
764     node.left = null;
765     flattenPrev = node;
766 }
767
768 // 25. Recover Binary Search Tree (two nodes swapped)
769 static TreeNode recoverFirst = null, recoverSecond = null, recoverPrev = null;
770 public static void recoverTree(TreeNode root) {
771     recoverFirst = recoverSecond = recoverPrev = null;
772     recoverDfs(root);
773     if (recoverFirst != null && recoverSecond != null) {
774         int tmp = recoverFirst.val;
775         recoverFirst.val = recoverSecond.val;
776         recoverSecond.val = tmp;
777     }
778 }
779 private static void recoverDfs(TreeNode node) {
780     if (node == null) return;
781     recoverDfs(node.left);
782     if (recoverPrev != null && node.val < recoverPrev.val) {
783         if (recoverFirst == null) recoverFirst = recoverPrev;
784         recoverSecond = node;
785     }
786     recoverPrev = node;
787     recoverDfs(node.right);
788 }
789
790 // 26. Path Sum III (any downward path) - count paths equal target
791 public static int pathSumIII(TreeNode root, int target) {
792     Map<Integer, Integer> prefix = new HashMap<>();
793     prefix.put(0, 1);
794     return pathSumIIISHelper(root, 0, target, prefix);
795 }
796 private static int pathSumIIISHelper(TreeNode node, int curr, int target, Map<Integer, Integer> prefix) {
797     if (node == null) return 0;
798     curr += node.val;
799     int res = prefix.getOrDefault(curr - target, 0);
800     prefix.put(curr, prefix.getOrDefault(curr, 0) + 1);
801     res += pathSumIIISHelper(node.left, curr, target, prefix);
802     res += pathSumIIISHelper(node.right, curr, target, prefix);
803     prefix.put(curr, prefix.get(curr) - 1);
804     return res;
805 }
806
807 // 27. Count Unival Subtrees
808 static int univalCount;
809 public static int countUnivalSubtrees(TreeNode root) {
810     univalCount = 0;
811     isUnival(root);
812     return univalCount;
813 }
814 private static boolean isUnival(TreeNode node) {
815     if (node == null) return true;
816     boolean left = isUnival(node.left);
817     boolean right = isUnival(node.right);

```

```

818     if (!left || !right) return false;
819     if (node.left != null && node.left.val != node.val) return false;
820     if (node.right != null && node.right.val != node.val) return false;
821     univalueCount++;
822     return true;
823 }
824
825 // 28. Construct Binary Tree from Preorder and Inorder
826 static int preIndex;
827 public static TreeNode buildTreePreIn(int[] preorder, int[] inorder) {
828     preIndex = 0;
829     Map<Integer, Integer> idx = new HashMap<>();
830     for (int i = 0; i < inorder.length; i++) idx.put(inorder[i], i);
831     return buildPreInHelper(preorder, 0, inorder.length - 1, idx);
832 }
833 private static TreeNode buildPreInHelper(int[] preorder, int inL, int inR, Map<
834 Integer, Integer> idx) {
835     if (inL > inR) return null;
836     int rootVal = preorder[preIndex++];
837     TreeNode root = new TreeNode(rootVal);
838     int pos = idx.get(rootVal);
839     root.left = buildPreInHelper(preorder, inL, pos - 1, idx);
840     root.right = buildPreInHelper(preorder, pos + 1, inR, idx);
841     return root;
842 }
843
844 // 29. Morris Inorder Traversal (O(1) extra space)
845 public static List<Integer> morrisInorder(TreeNode root) {
846     List<Integer> res = new ArrayList<>();
847     TreeNode curr = root;
848     while (curr != null) {
849         if (curr.left == null) {
850             res.add(curr.val);
851             curr = curr.right;
852         } else {
853             TreeNode pred = curr.left;
854             while (pred.right != null && pred.right != curr) pred = pred.right;
855             if (pred.right == null) {
856                 pred.right = curr;
857                 curr = curr.left;
858             } else {
859                 pred.right = null;
860                 res.add(curr.val);
861                 curr = curr.right;
862             }
863         }
864     }
865     return res;
866 }
867
868 // 30. Convert BST to Sorted Doubly Linked List (in-place)
869 // Reuse left as prev and right as next. Return head of doubly linked list.
870 static TreeNode dllPrev = null;
871 public static TreeNode bstToDoublyList(TreeNode root) {
872     dllPrev = null;
873     if (root == null) return null;
874     TreeNode head = bstToDoublyListHelper(root);
875     // Move to head
876     while (head != null && head.left != null) head = head.left;
877     return head;
878 }
879 private static TreeNode bstToDoublyListHelper(TreeNode node) {
880     if (node == null) return null;
881     bstToDoublyListHelper(node.left);
882     // link prev <-> node
883     node.left = dllPrev;
884     if (dllPrev != null) dllPrev.right = node;
885     dllPrev = node;
886     bstToDoublyListHelper(node.right);

```

```

886     return node;
887 }
888
889 // ----- Helper: Build Sample Tree -----
890
891 public static TreeNode buildSampleTree() {
892     TreeNode root = new TreeNode(1);
893     root.left = new TreeNode(2);
894     root.right = new TreeNode(3);
895     root.left.left = new TreeNode(4);
896     root.left.right = new TreeNode(5);
897     root.right.left = new TreeNode(6);
898     root.right.right = new TreeNode(7);
899     return root;
900 }
901
902 // ----- Main: Quick demonstration -----
903
904 public static void main(String[] args) {
905     TreeNode root = buildSampleTree();
906     System.out.println("Inorder recursive: " + inorderTraversal(root));
907     System.out.println("Inorder iterative: " + inorderIterative(root));
908     System.out.println("Preorder recursive: " + preorderTraversal(root));
909     System.out.println("Postorder recursive: " + postorderTraversal(root));
910     System.out.println("Level Order: " + levelOrder(root));
911     System.out.println("Zigzag: " + zigzagLevelOrder(root));
912     System.out.println("Height: " + height(root));
913     System.out.println("Diameter: " + diameter(root));
914     System.out.println("Left view: " + leftView(root));
915     System.out.println("Right view: " + rightView(root));
916     System.out.println("Top view: " + topView(root));
917     System.out.println("Bottom view: " + bottomView(root));
918     System.out.println("Has path sum 8: " + hasPathSum(root, 8));
919     System.out.println("All paths: " + allPaths(root));
920     System.out.println("LCA(4,5): " + lowestCommonAncestor(root, root.left.left, root.left.right));
921     System.out.println("Is balanced: " + isBalanced(root));
922     System.out.println("Is symmetric example: " + isSymmetric(root.left)); // not
923     // symmetric but demo
924     TreeNode sumRoot = new TreeNode(-10); sumRoot.left = new TreeNode(9); sumRoot.
925     right = new TreeNode(20);
926     sumRoot.right.left = new TreeNode(15); sumRoot.right.right = new TreeNode(7);
927     System.out.println("Max path sum example: " + maxPathSum(sumRoot));
928
929     String ser = serialize(root);
930     System.out.println("Serialized: " + ser);
931     TreeNode deser = deserialize(ser);
932     System.out.println("Deserialized level order: " + levelOrder(deser));
933
934     int[] sorted = {-10,-3,0,5,9};
935     TreeNode bst = sortedArrayToBST(sorted);
936     System.out.println("Sorted array->BST inorder: " + inorderTraversal(bst));
937     System.out.println("Min: " + findMin(root) + " Max: " + findMax(root));
938
939     // 21: Validate BST (use bst built from sorted array)
940     System.out.println("Is valid BST: " + isValidBST(bst));
941
942     // 22: kth smallest
943     System.out.println("Kth smallest (k=3) in BST: " + kthSmallest(bst, 3));
944
945     // 23: invert tree
946     TreeNode inv = invertTree(buildSampleTree());
947     System.out.println("Inverted inorder: " + inorderTraversal(inv));
948
949     // 24: flatten
950     TreeNode flatSample = buildSampleTree();
951     flatten(flatSample);
952     System.out.print("Flattened list (right pointers): "); TreeNode cur = flatSample;
953     while (cur != null) { System.out.print(cur.val + " "); cur = cur.right; }

```

```

950     System.out.println();
951
952     // 25: recover tree - create swapped BST
953     TreeNode r = new TreeNode(3); r.left = new TreeNode(1); r.right = new TreeNode(4);
954     r.right.left = new TreeNode(2);
955     System.out.println("Before recover inorder: " + inorderTraversal(r));
956     // swap values of two nodes to simulate error
957     int tmp = r.val; r.val = r.right.left.val; r.right.left.val = tmp;
958     System.out.println("After swap inorder: " + inorderTraversal(r));
959     recoverTree(r);
960     System.out.println("After recover inorder: " + inorderTraversal(r));
961
962     // 26: Path Sum III
963     TreeNode pSum = buildSampleTree(); System.out.println("PathSumIII target=3: " +
964     pathSumIII(pSum, 3));
965
966     // 27: Count univalue subtrees
967     TreeNode u = new TreeNode(5); u.left = new TreeNode(1); u.right = new TreeNode(5);
968     u.right.left = new TreeNode(5); u.right.right = new TreeNode(5);
969     System.out.println("Univalue subtree count: " + countUnivalSubtrees(u));
970
971     // 28: build from preorder & inorder
972     int[] pre = {3,9,20,15,7}; int[] in = {9,3,15,20,7};
973     TreeNode built = buildTreePreIn(pre, in);
974     System.out.println("Built tree inorder (should be inorder array): " +
975     inorderTraversal(built));
976
977     // 29: morris inorder
978     System.out.println("Morris inorder on sample: " + morrisInorder(buildSampleTree(
979     )));
980
981     // 30: bst to doubly linked list
982     TreeNode dll = bstToDoublyList(bst);
983     System.out.print("BST->DLL inorder forward: "); TreeNode h = dll; while (h!=null)
984     { System.out.print(h.val+" "); h = h.right; } System.out.println();
985
986     System.out.println("--- End of demo ---");
987 }

```