

13
.1

Inorder Traversal

EASY

■ What You Will Learn

- ✓ What Inorder traversal means: Left → Root → Right
- ✓ Recursive approach — the simplest and most intuitive
- ✓ Iterative approach using an explicit Stack
- ✓ Why Inorder of a BST gives sorted output
- ✓ Both approaches with full dry runs

■ Intuition First

Think of a tree as a family tree. Inorder says: visit the entire left family first, then meet the person in the middle, then visit the entire right family. For a Binary Search Tree, this magical Left → Root → Right order always gives you numbers in sorted ascending order!

The word 'In' in Inorder means the root comes IN between the two subtrees.

■ The Pattern

- Inorder = Left → Root → Right
- Preorder = Root → Left → Right (root BEFORE subtrees)
- Postorder = Left → Right → Root (root AFTER subtrees)
- For BST: Inorder always produces a sorted array

■ Visual — Tree Diagram

Sample binary tree

```
1
 / \
2   3
 / \
4   5
```

Inorder traversal: 4 → 2 → 5 → 1 → 3

(Left-most node first, root in the middle, right side last)

Step-by-step visit order

```

Visit 4 (leftmost) ✓
Visit 2 (left subtree root) ✓
Visit 5 (right child of 2) ✓
Visit 1 (overall root) ✓
Visit 3 (right subtree) ✓
Result: [4, 2, 5, 1, 3]

```

■ ■ How It Works — Recursive

1. If node is null → return (base case)
2. Recursively call `inorder(node.left)` — go all the way left first
3. Visit / add `node.val` to result — process root
4. Recursively call `inorder(node.right)` — then visit right

■ Dry Run — Recursive Call Stack

```

Tree: 1
/ \
2 3
/ \
4 5

```

```

inorder(1) calls inorder(2), then visits 1, then calls inorder(3)
inorder(2) calls inorder(4), visits 2, calls inorder(5)
inorder(4) → null left, visits 4, null right → returns [4]
Back to 2 → visits 2 → [4,2]
inorder(5) → null left, visits 5, null right → [4,2,5]
Back to 1 → visits 1 → [4,2,5,1]
inorder(3) → null left, visits 3, null right → [4,2,5,1,3] ✓

```

■ Java Code — Recursive

TreeNode definition (used in all Binary Tree problems):

```

class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val) { this.val = val; }
}

```

```

public static List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    inorder(root, result);
    return result;
}

private static void inorder(TreeNode node, List<Integer> result) {
    if (node == null) return; // Base case

```

```
inorder(node.left, result); // L – go left first
result.add(node.val); // N – visit current node
inorder(node.right, result); // R – then go right
}
```

■ Java Code — Iterative (Stack-based)

The iterative approach manually simulates the call stack using an explicit Stack.

```
public static List<Integer> inorderIterative(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
    TreeNode curr = root;
    while (curr != null || !stack.isEmpty()) {
        // Go as far left as possible
        while (curr != null) {
            stack.push(curr);
            curr = curr.left;
        }
        // Process node (backtrack from left)
        curr = stack.pop();
        result.add(curr.val);
        // Move to right subtree
        curr = curr.right;
    }
    return result;
}
```

■ Iterative Dry Run

Step 1	curr=1. Push 1. curr=1.left=2. Push 2. curr=2.left=4. Push 4. curr=null
Step 2	Stack: [1,2,4]. Pop 4. result=[4]. curr=4.right=null
Step 3	Stack: [1,2]. Pop 2. result=[4,2]. curr=2.right=5. Push 5. curr=null
Step 4	Stack: [1,5]. Pop 5. result=[4,2,5]. curr=5.right=null
Step 5	Stack: [1]. Pop 1. result=[4,2,5,1]. curr=1.right=3. Push 3. curr=null
Step 6	Stack: [3]. Pop 3. result=[4,2,5,1,3]. curr=null. Stack empty → DONE ✓

■ Time & Space Complexity

Case	Time	Space	What it means
------	------	-------	---------------

All Cases

 $O(n)$ $O(n)$ Visit every node once. Stack/recursion depth = height = $O(n)$ worst, $O(\log n)$ balanced

■ Why Inorder Gives BST Sorted Output

- BST property: left child < root < right child
- Inorder visits left first → all smaller values first
- Then root → then right → all larger values last
- This naturally produces ascending order — a key BST verification technique!

■ Common Mistakes

- Confusing L-N-R (Inorder) with N-L-R (Preorder) — remember: 'In' = root in the middle
- Forgetting null check in recursive base case
- In iterative: forgetting to move `curr = curr.right` after popping

■ Visual Thinking

Picture a tree lying on its side. Inorder sweeps from the leftmost tip to the rightmost tip.

It's like reading a book shelf from left to right — but the shelf is arranged as a tree.

Animation: highlight nodes one by one left-to-right. Stack grows as you dive left, shrinks as you visit.

For BST specifically: watch numbers appear in sorted order — satisfying!

■ Practice Problems

[Easy] Inorder of a single-node tree [42]?

[Easy] Verify if a BST is valid using Inorder traversal

[Med] Binary Tree to Doubly Linked List using Inorder

[Med] Kth smallest element in BST using Inorder

[Hard] Morris Traversal — Inorder in $O(1)$ space (no stack/recursion)

■ Quick Revision Box

- Inorder = Left $\rightarrow$ Root $\rightarrow$ Right
- Recursive: `inorder(left)`, visit root, `inorder(right)`
- Iterative: push all left nodes, pop and process, move right
- $O(n)$ time, $O(n)$ space (call stack or explicit stack)
- Inorder of BST = sorted array — key insight!