CSC211 · Algorithms and Data Structures

19. Binary Tree Traversals

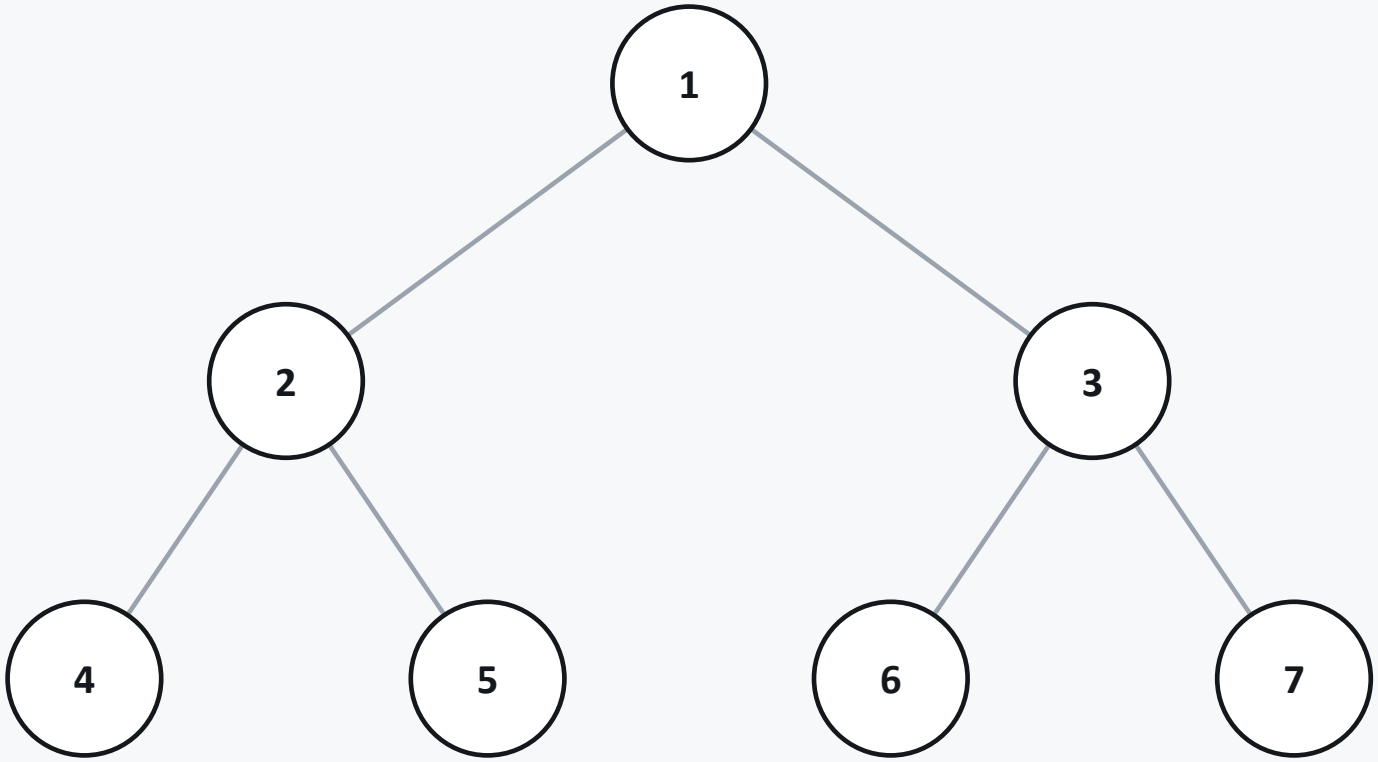
Pre-order, in-order, post-order, and level-order — four ways to visit every node

COMSATS University Islamabad, Lahore Campus

Agenda

1. Why a tree has more than one natural visiting order
2. Pre-order, in-order, and post-order (all depth-first)
3. Level-order traversal (breadth-first, from Lecture 17)
4. Proof: in-order traversal sorts a BST for free
5. A classic bug: when you print vs. which order you recurse
6. Comparing all four, and when to reach for each one

One Tree, Several Visiting Orders



?

At every node: go left first, process the node first, or go right first? Each choice defines a different traversal, each useful for a different real purpose.

Every traversal below runs on this exact tree

Three Functions, One Shape

tree_traversals.cpp

```
void preOrder(TreeNode* node) {
    if (node == nullptr) return;
    cout << node->data << " ";
    preOrder(node->left); preOrder(node->right);
}
void inOrder(TreeNode* node) {
    if (node == nullptr) return;
    inOrder(node->left);
    cout << node->data << " ";
    inOrder(node->right);
}
void postOrder(TreeNode* node) {
    if (node == nullptr) return;
    postOrder(node->left); postOrder(node->right);
    cout << node->data << " ";
}
```

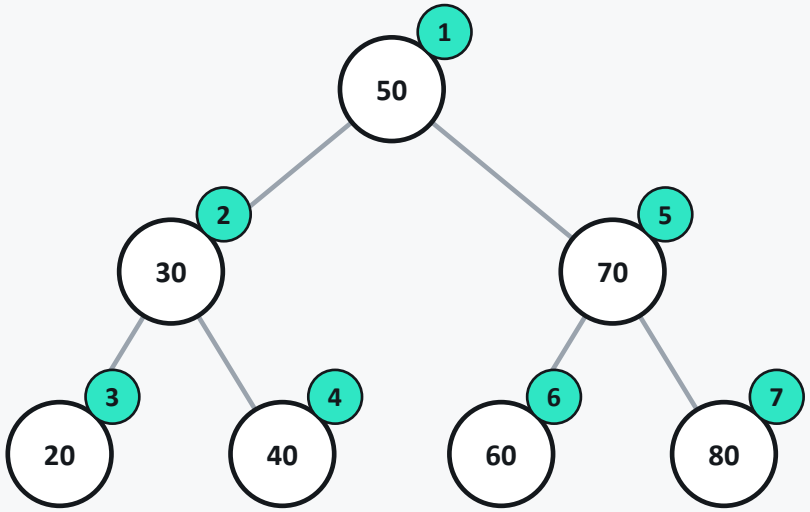
Same three lines, rearranged — only the position of the print (relative to the two recursive calls) changes.

Three Rules, Three Different Outputs

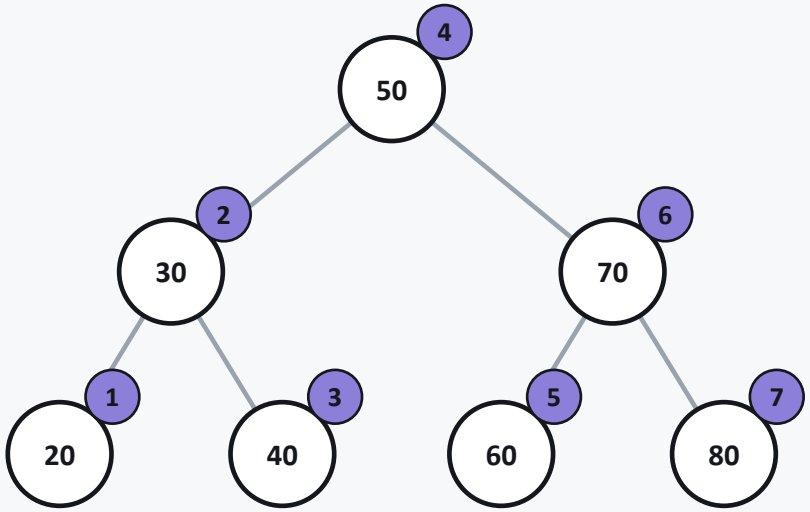
Traversal	Visit order rule	Output on this tree
Pre-order	Node → Left → Right	1 2 4 5 3 6 7
In-order	Left → Node → Right	4 2 5 1 6 3 7
Post-order	Left → Right → Node	4 5 2 6 7 3 1

Root 1 appears first in pre-order, last in post-order, and in the middle of in-order — exactly where “Node” sits in each rule. No two traversals ever match for a tree with more than one node.

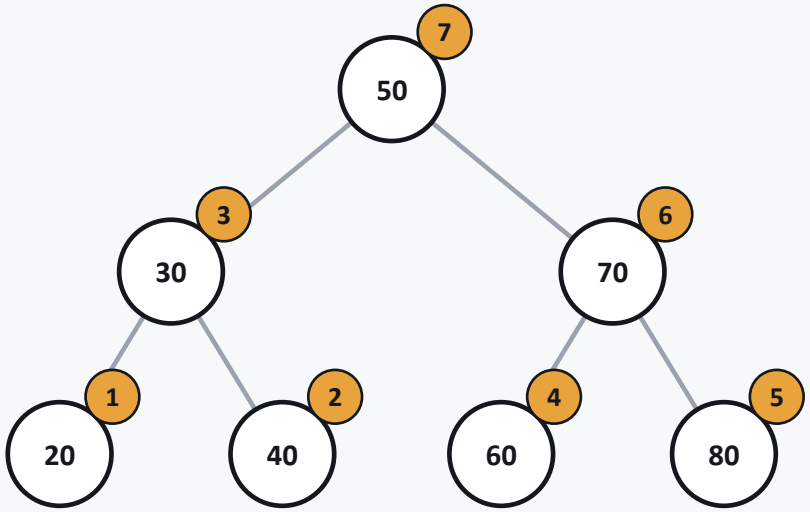
The Visit Order, Annotated on the Tree



Pre-order (Node, Left, Right)



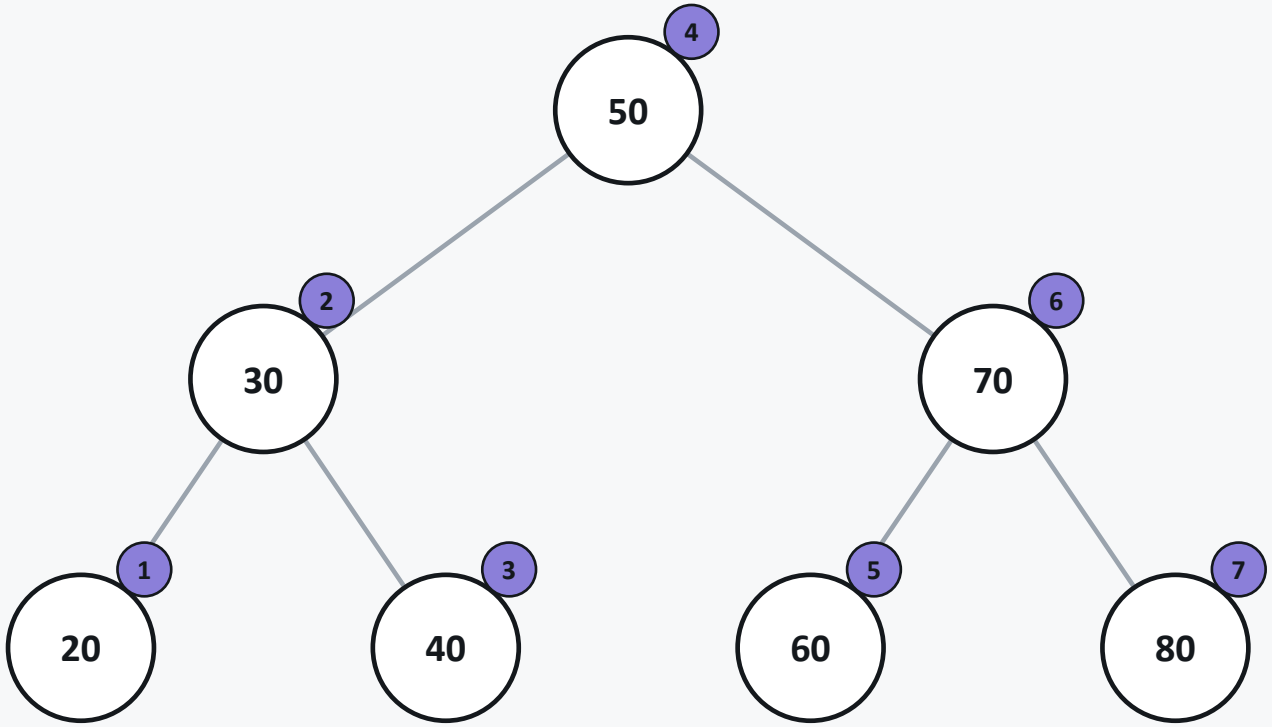
In-order (Left, Node, Right)



Post-order (Left, Right, Node)

Same BST, same 7 nodes — the orange/teal/purple badges show the position in each visit order.

In-Order Traversal Sorts a BST — For Free



In-order visit numbers shown in purple

Read left to right, bottom of tree:

```
20   30   40   50   60   70
80
```

...already sorted. Zero extra sorting work.

This is the single most important fact about in-order traversal in the course: left-subtree-smaller, right-subtree-larger, applied recursively at every node, plus “left, node, right”, produces sorted output — the whole reason BSTs (Lecture 20) are useful.

Verified With Real Code, Not Just the Diagram

```
bst_traversal_orders.cpp — same BST (50,30,70,20,40,60,80)
inOrder(root);
// Left, Node, Right — recurses into the
// smaller side first, prints, then the
// larger side — at every node, recursively.
```

Pre-order	50 30 20 40 70 60 80
In-order	20 30 40 50 60 70 80
Post-order	20 40 30 60 80 70 50

0

extra sorting operations performed — the order falls straight out of the BST property

Post-Order: Children Before Parent

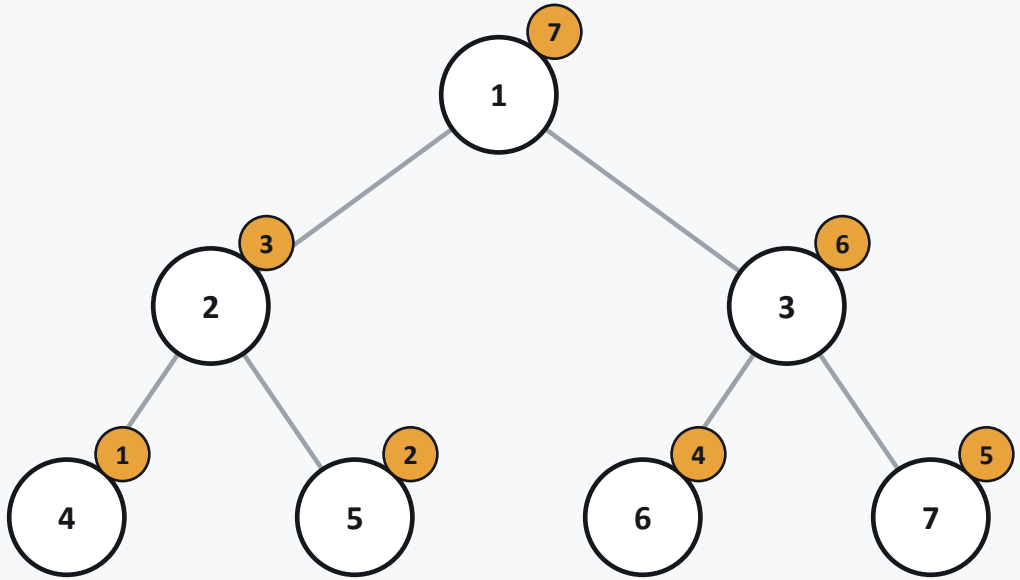
Left → Right → Node guarantees every node's children are fully processed before the node itself.

Safely deleting a tree

Free every child before freeing the parent — never touch freed memory.

Evaluating an expression tree

Bottom-up: operands must be computed before the operator above them (Lecture 24).



Post-order visit numbers — root visited last

Pitfall: When You Print vs. Which Order You Recurse

Two independent choices — mixing up either produces a real, different, wrong traversal that still compiles and runs cleanly.

inOrderCorrect ✓

```
void inOrderCorrect(TreeNode* n) {
  if (n == nullptr) return;
  inOrderCorrect(n->left);
  cout << n->data << " ";
  inOrderCorrect(n->right);
}
```

inOrderBuggy ✗ (print moved to top)

```
void inOrderBuggy(TreeNode* n) {
  if (n == nullptr) return;
  cout << n->data << " ";  // moved!
  inOrderBuggy(n->left);
  inOrderBuggy(n->right);
}
```

postOrderCorrect ✓

```
postOrderCorrect(n->left);
postOrderCorrect(n->right);
cout << n->data << " ";
```

postOrderBuggy ✗ (left/right swapped)

```
postOrderBuggy(n->right);  // swapped!
postOrderBuggy(n->left);
cout << n->data << " ";
```

Neither Bug Crashes — They Just Return the Wrong Answer

Function	Real output
inOrderCorrect	20 30 40 50 60 70 80
inOrderBuggy	50 30 20 40 70 60 80
postOrderCorrect	20 40 30 60 80 70 50
postOrderBuggy	80 60 70 40 20 30 50

inOrderBuggy ≡ preOrder

moving the print to the top didn't create a new traversal — it silently became pre-order

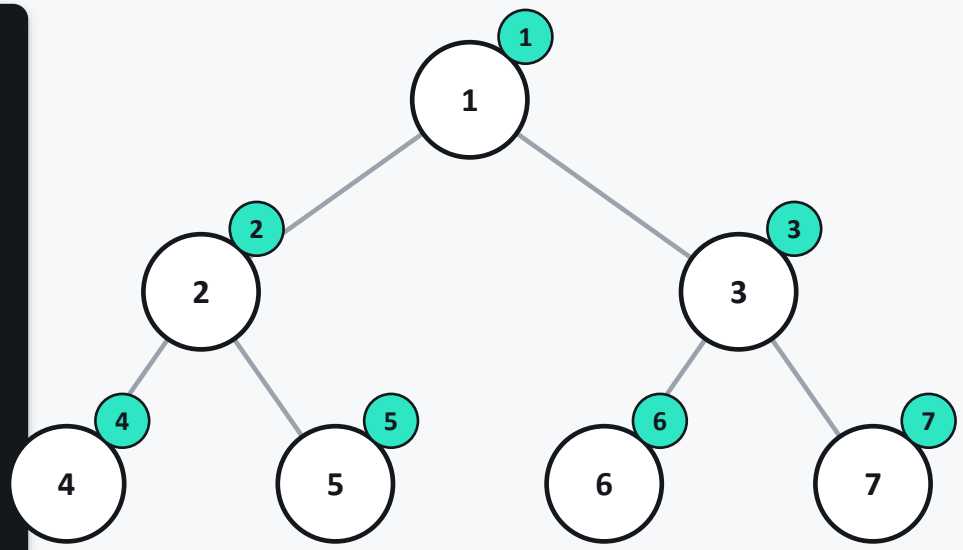
postOrderBuggy = mirrored

swapping the two calls visits every right subtree before every left one

Level-Order: The One Breadth-First Traversal

level_order.cpp

```
void levelOrder(TreeNode* root) {
    queue<TreeNode*> toVisit;
    toVisit.push(root);
    while (!toVisit.empty()) {
        TreeNode* cur = toVisit.front(); toVisit.pop();
        cout << cur->data << " ";
        if (cur->left)  toVisit.push(cur->left);
        if (cur->right) toVisit.push(cur->right);
    }
}
```



Depth 0, then depth 1, then depth 2 — via a queue, not recursion.

The only traversal that uses an explicit data structure instead of the call stack.

All Four Traversals, Side by Side

Traversal	Order	Uses	Typical application
Pre-order	Node, Left, Right	Recursion (call stack)	Copying/cloning a tree; serialization
In-order	Left, Node, Right	Recursion	Reading a BST's values in sorted order
Post-order	Left, Right, Node	Recursion	Safe deletion; expression-tree evaluation
Level-order	Depth 0, 1, 2, ...	A queue, explicitly	Printing shape; shortest path (BFS)

Pre-order, in-order, and post-order all secretly use the call stack (Lecture 12). **Level-order is the only one that reaches for an explicit data structure.**

Applications of Tree Traversal

In-order

Retrieves BST values in sorted order — zero extra sorting work.

Post-order

Required whenever children must be fully processed before their parent.

Pre-order

Natural way to copy a tree: create the root first, then attach copied subtrees.

Level-order

Prints a tree's actual shape; generalizes to BFS on graphs (Lecture 26).

Key Takeaways

- Pre-order, in-order, and post-order are depth-first, differing only in when the current node is processed relative to its subtrees.
- In-order traversal of a Binary Search Tree visits nodes in sorted order — the single most important fact in this lecture.
- Post-order guarantees children are processed before their parent — required for safe deletion and bottom-up expression evaluation.
- Level-order is the only breadth-first traversal, built on a queue rather than recursion.
- Getting a traversal wrong is a silent bug: it compiles and runs, but quietly produces the wrong sequence — always check against real output.