LECTURE 21

BST: Deletion and Analysis

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

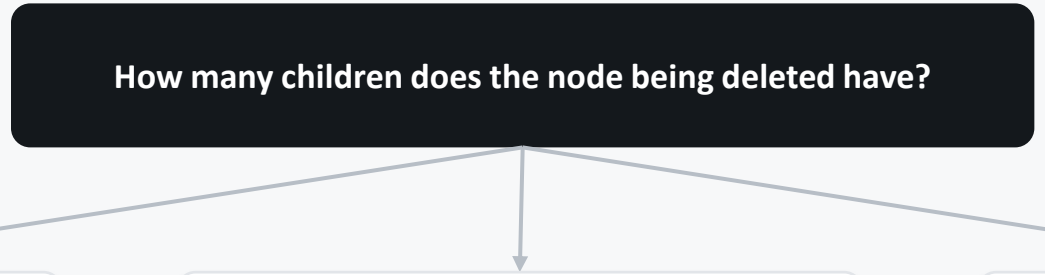
LECTURE 21

Agenda / Learning Objectives

1. The three cases of BST deletion: leaf, one child, two children
2. Why two children needs the in-order successor
3. A multi-level cascading deletion, traced with real output
4. The in-order predecessor as a symmetric alternative
5. Best-case / worst-case behavior tied to tree shape
6. Common pitfalls when implementing BST deletion

DELETION

BST Deletion: Three Cases



0

Leaf

Just remove it.
Nothing else to fix.

1

One Child

Splice it out — connect its parent directly to its child.

2

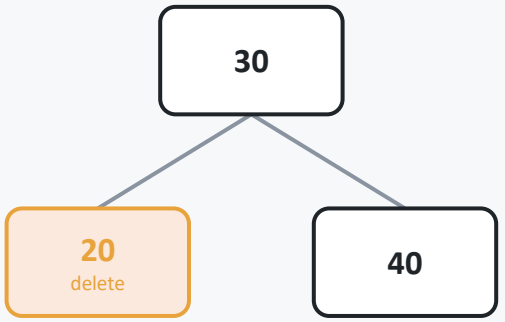
Two Children

Replace its value with its in-order successor's value, then delete that successor.

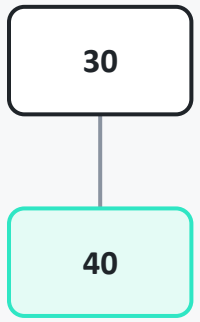
CASE 1

Deleting a Leaf Node

BEFORE



AFTER



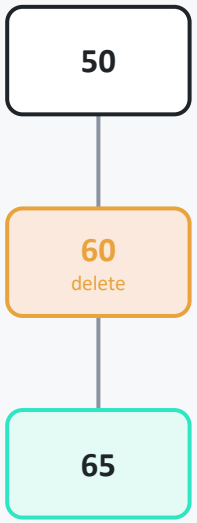
No children means nothing depends on this node — simply **remove it**, and set its parent's pointer to `nullptr`.

```
if (node->left == nullptr && node->right == nullptr) {
    delete node;   // Case 1: leaf node
    return nullptr;
}
```

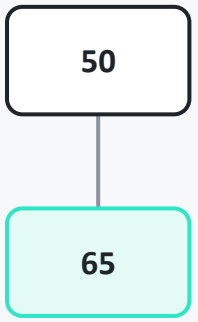
CASE 2

Deleting a Node With One Child

BEFORE



AFTER



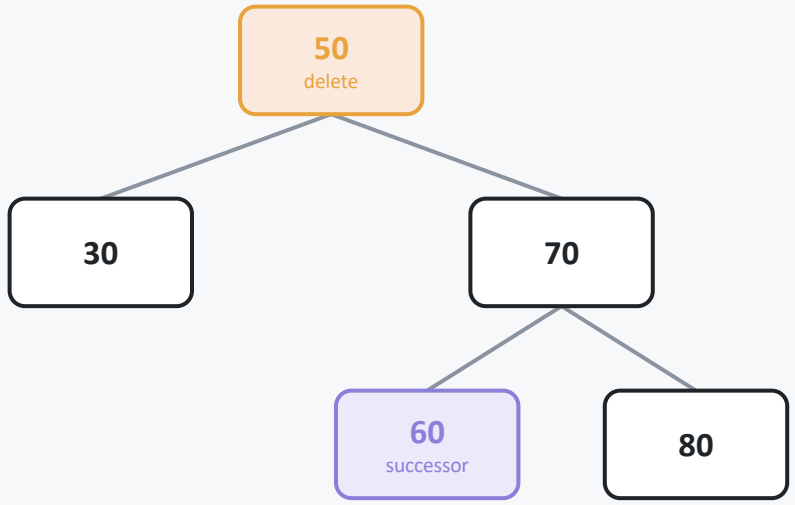
The node's single child can safely take its exact place — connect the deleted node's **parent** directly to its **child**, skipping over the deleted node entirely.

```
} else if (node->left == nullptr) {    // Case 2
    TreeNode* temp = node->right;
    delete node;
    return temp;    // child takes deleted node's place
```

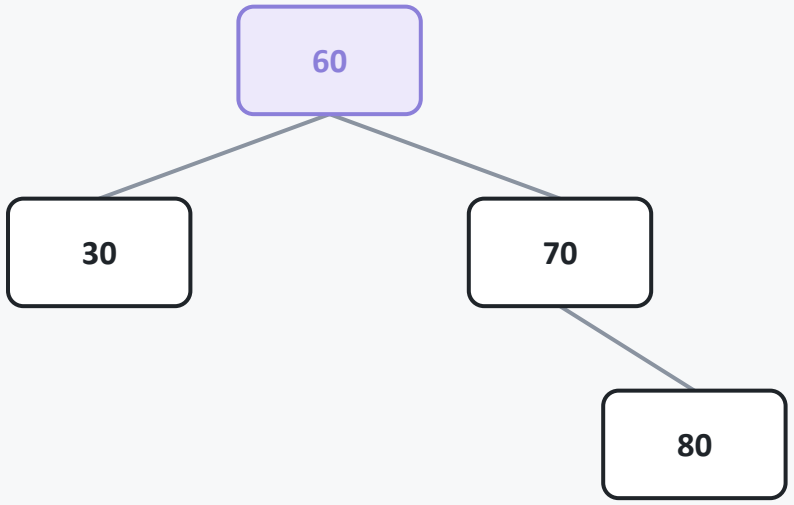
CASE 3

Deleting a Node With Two Children

BEFORE



AFTER



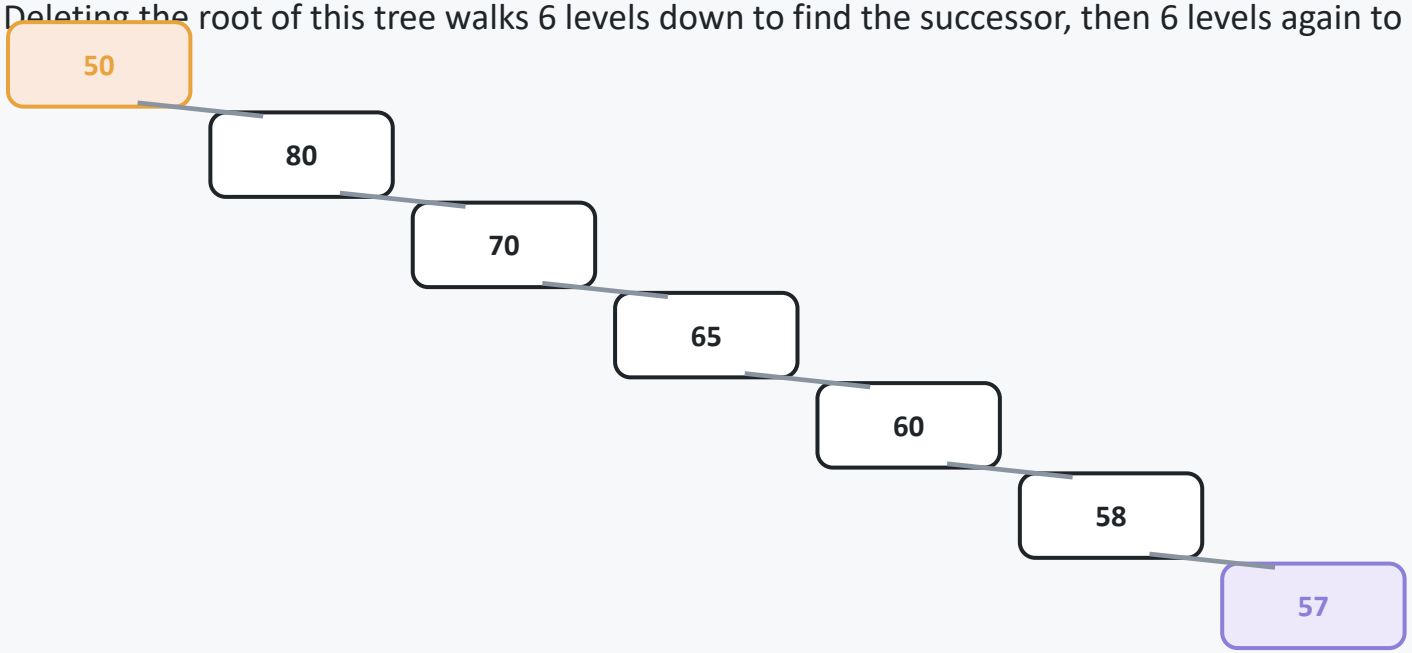
60's VALUE is copied into 50's node; the successor node 60 (leftmost of the right subtree) is the one actually freed — 50's original node survives, just relabeled.

```
TreeNode* successor = findMinNode(node->right);
node->data = successor->data;    // copy value up
node->right = deleteHelper(node->right, successor->data);
```

DEEP DIVE

A Multi-Level Cascading Deletion

Deleting the root of this tree walks 6 levels down to find the successor, then 6 levels again to remove it:



search down → (6 levels)

6 + 6
recursive calls: find successor, then delete it

O(h)
cost of Case 3's extra work: proportional to height

Two things worth noticing: the search for the successor and the delete of the successor walk the **exact same six-node path** twice. And 50's node survives — it's the real 57 node that's freed.

SYMMETRY

The In-Order Predecessor

- The in-order successor (min of right subtree) is one valid fix for Case 3.
- The in-order predecessor (max of left subtree) works identically, by symmetry.
- Found by walking node->left, then node->right repeatedly until right is nullptr.
- Guaranteed to have at most one child — same reduction to Case 1/2 applies.
- Neither convention is more “correct” — real codebases pick one and stay consistent.

find_predecessor.cpp

```
TreeNode* findPredecessor(TreeNode* node) {
    node = node->left;
    while (node->right != nullptr)
        node = node->right;
    return node;    // max of left subtree
}
```

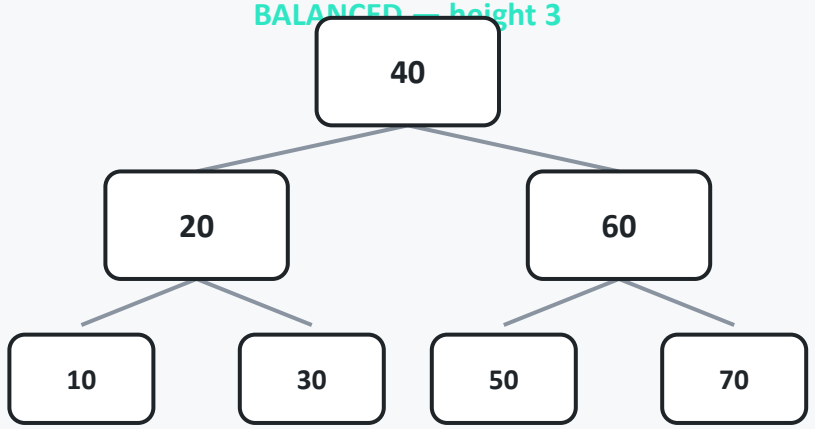
Symmetric, not superior: swapping successor for predecessor in deleteHelper's Case 3 still produces a valid, correctly-sorted BST after every deletion — just a different internal shape.

SHAPE MATTERS

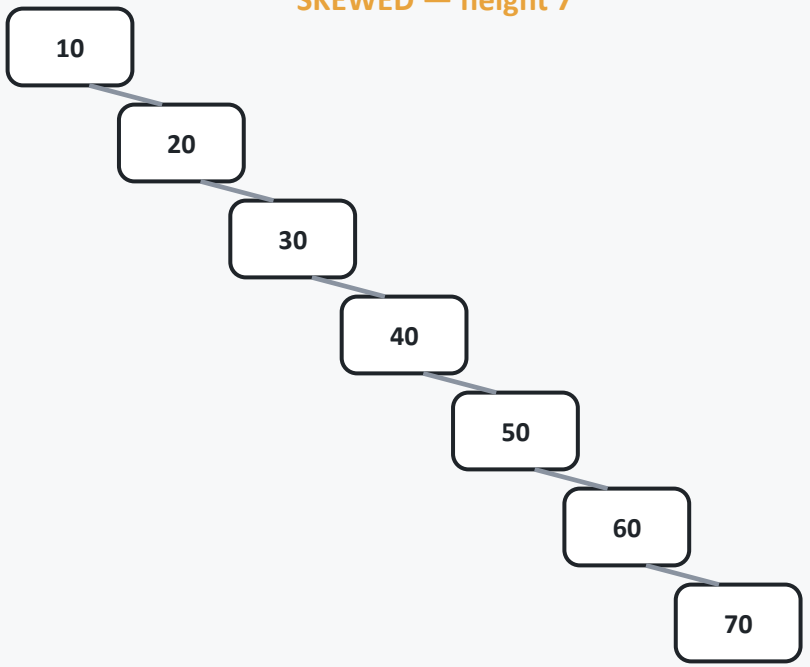
Balanced BST vs. Skewed BST

Same 7 values (10–70) — different insertion order produces a completely different shape:

BALANCED — height 3



SKEWED — height 7



	Balanced BST	Skewed BST
Height	O(log n)	O(n)
Search / Insert / Delete	O(log n)	O(n)

ANALYSIS

Deletion's Cost, Case by Case

Case	Children	Fix	Cost of the fix
1	0 (leaf)	Remove directly	O(1)
2	1	Splice parent to child	O(1)
3	2	Copy successor value up, delete successor	O(height) to find it

Every case bottoms out at O(1) work — Case 3 only “costs more” because finding the successor is a walk down the tree, exactly like the walk Cases 1 and 2 already do to find the node itself.

COMMON PITFALLS

- Freeing before reading — delete node before saving temp/successor->data corrupts the deletion.
- Forgetting two-children “reduces, doesn't finish” — the original successor node still needs its own delete.
- Assuming the successor is the right child directly — it's the leftmost node of the right subtree, arbitrarily far down.

BEYOND LOOKUP

Applications of BSTs

Range & Nearest-Neighbor Queries

“Find every value between 40 and 70” can skip entire subtrees provably outside that range.

Priority Scheduling With Ordering

When both “give me the smallest” and “search for a value” are needed together — a plain heap alone can't do this.

Looking ahead: a BST's Big-O is not a fixed property of “being a BST” — it's a property of the tree's actual shape. Lecture 22's AVL tree adds automatic rebalancing so the worst case simply can't happen.

Key Takeaways

1. BST deletion has three cases by child count: 0 (remove), 1 (splice out), 2 (replace with in-order successor, then delete it).
2. The in-order successor is always the leftmost node of the right subtree, and is guaranteed to have at most one child.
3. The node whose value changes and the node whose memory is freed are two different nodes in a two-children deletion.
4. The in-order predecessor (max of left subtree) is an equally valid, symmetric alternative.
5. A BST's performance depends entirely on its current shape, not just on being a BST — exactly what Lecture 22's AVL tree fixes.