LECTURE 22

AVL Trees

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

LECTURE 22

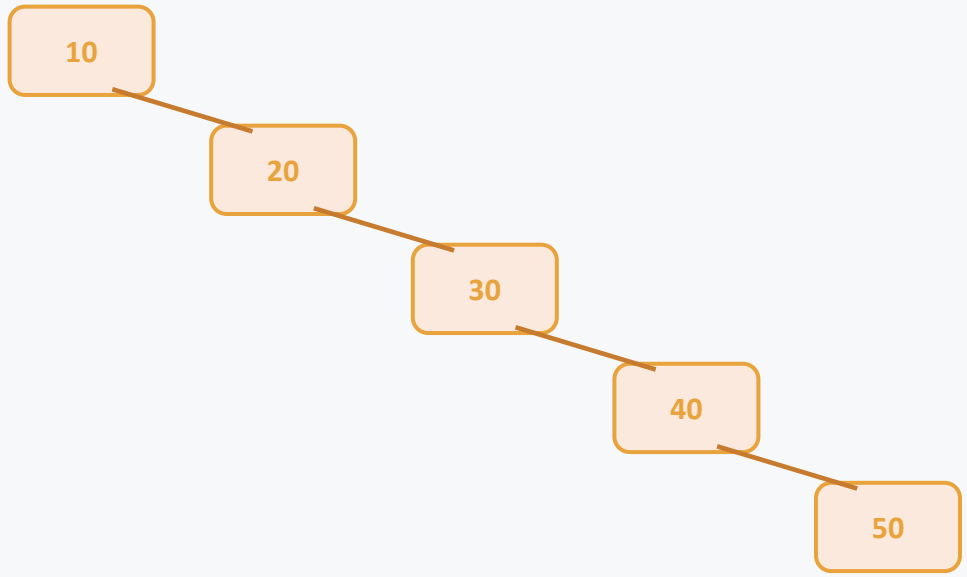
Agenda / Learning Objectives

1. Why balanced search trees are needed, precisely
2. The AVL balance factor and the height-balance rule it enforces
3. Four rotation cases: single (LL, RR) and double (LR, RL) — all verified with real output
4. Why a rotation is guaranteed to preserve the BST property
5. A complete AVL insertion and AVL deletion, both self-rebalancing
6. AVL trees vs. the red-black trees std::map/std::set actually use

MOTIVATION

Why Balanced Search Trees Are Needed

Insert 10, 20, 30, 40, 50 into a plain BST — every node lands with only a right child:



O(n)
height n − 1 — search degrades to a linked list

O(log n)
AVL guarantees this height, always

An AVL tree *(Adelson-Velsky and Landis)* **refuses to let this happen**: after every insertion, it checks whether the tree has become too lopsided, and if so, performs a **rotation** to restore balance immediately — guaranteeing O(log n) height no matter what order values arrive in.

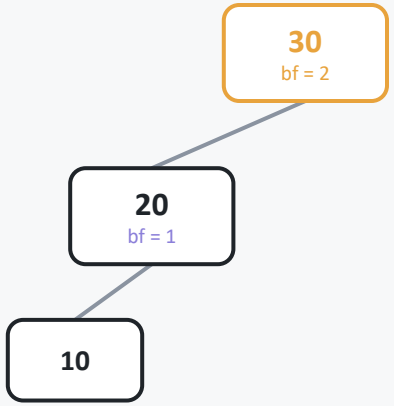
CORE CONCEPT

The AVL Balance Factor

```
balance factor =
height(left) - height(right)
```

The AVL property: every node's balance factor must be **−1, 0, or 1**. If an insertion ever pushes a node's balance factor to −2 or 2, the tree is out of balance at that node, and a rotation must fix it before moving on.

EXAMPLE




```
height(left of 30) = 2
height(right of 30) = 0
balance factor = 2 - 0 = 2   ← violates {-1,0,1}
```

This is the LL case — fixed by a single right rotation (next slides).

REBALANCING

The Four Rotation Cases

LL

Left-Left Heavy

Fix: single RIGHT rotation

RR

Right-Right Heavy

Fix: single LEFT rotation

LR

Left-Right Heavy

Fix: LEFT rotate the left child, then RIGHT rotate the node

RL

Right-Left Heavy

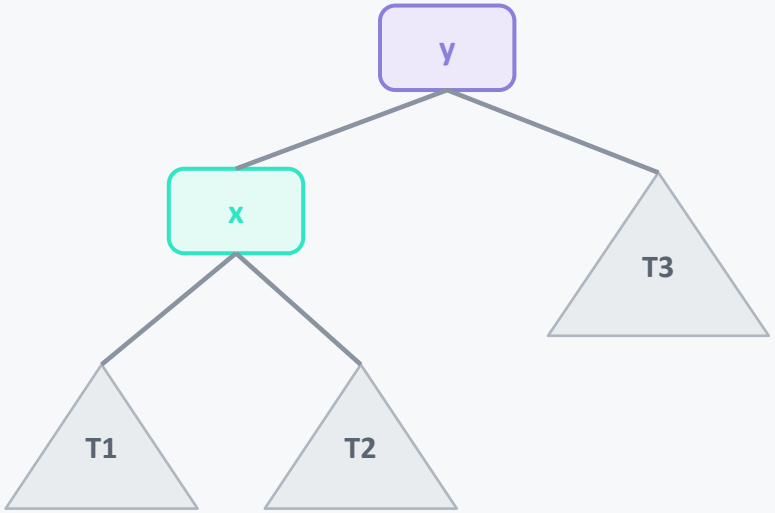
Fix: RIGHT rotate the right child, then LEFT rotate the node

Single vs. double: a single rotation (LL/RR) fixes a straight-line imbalance; a double rotation (LR/RL) fixes a zig-zag — the first rotation straightens it into a line, the second then fixes it like a single-rotation case.

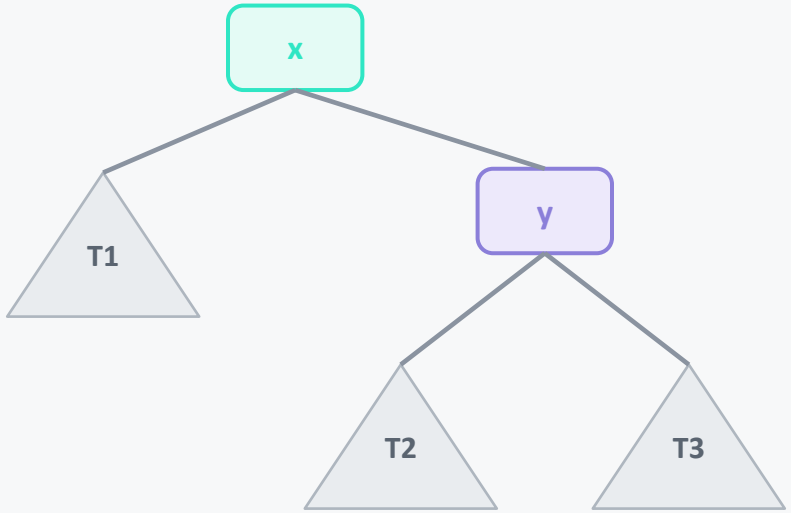
CORRECTNESS

Why a Rotation Preserves the BST Property

BEFORE: right rotation at y



AFTER



T1 < x < T2 < y < T3 already holds before the rotation. Only **T2** ever moves — from x's right subtree to y's left subtree — legal precisely because T2 < y already held. Only three pointers are reassigned; no value crosses a boundary it wasn't already inside.

IMPLEMENTATION

rotateRight and rotateLeft

rotateRight — fixes LL

```
AVLNode* rotateRight(AVLNode* y) {
    AVLNode* x = y->left;
    AVLNode* transferred = x->right;    // T2

    x->right = y;
    y->left = transferred;

    updateHeight(y);
    updateHeight(x);
    return x;    // new subtree root
}
```

rotateLeft — fixes RR

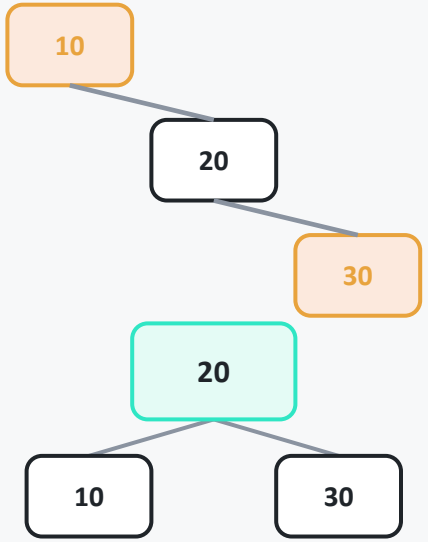
```
AVLNode* rotateLeft(AVLNode* x) {
    AVLNode* y = x->right;
    AVLNode* transferred = y->left;    // T2

    y->left = x;
    x->right = transferred;

    updateHeight(x);
    updateHeight(y);
    return y;    // new subtree root
}
```

Only three pointer reassignments each — x->right/y->left plus the caller re-linking the returned node in the old root's place.

VERIFIED

RR and LR Cases, Compiled and Traced

RR: insert 10, 20, 30

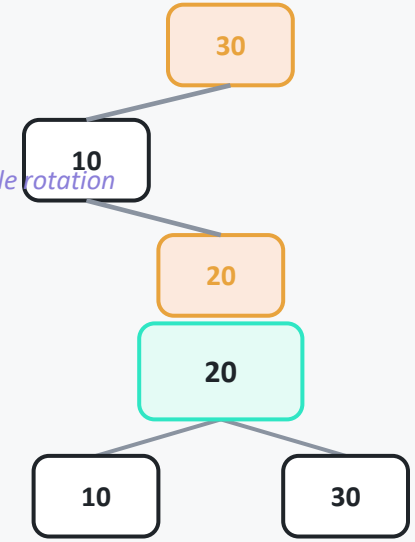


→ single LEFT rotation

height 2

root = 20 — single left rotation fixed it

LR: insert 30, 10, 20



→ double rotation

height 2

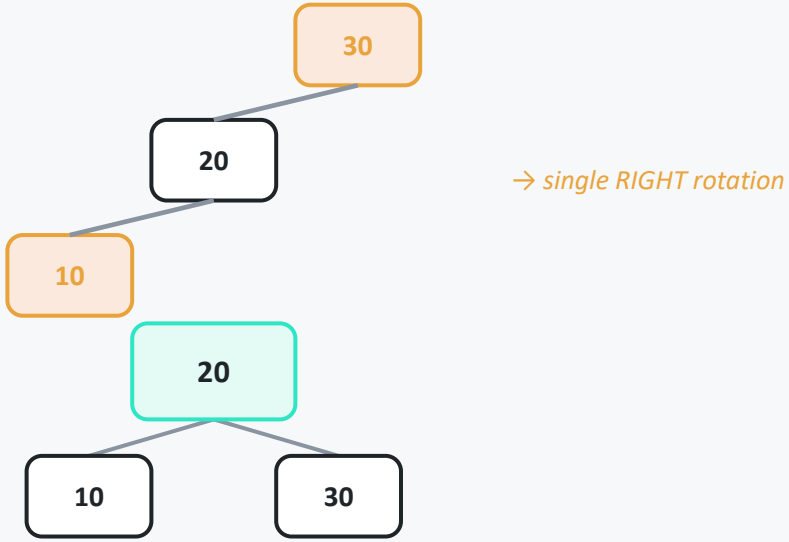
root = 20 — double rotation fixed it (rotate left at 10, then right at 30)

Both land on the identical balanced shape: root 20, children 10 and 30 — confirmed by real compiled balance-factor output, not hand-waving.

VERIFIED

LL and RL Cases: the Mirror Images

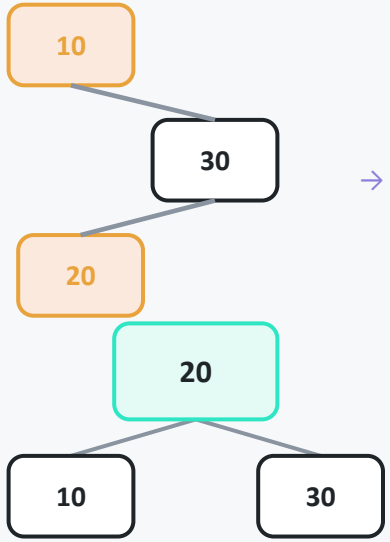
LL: insert 30, 20, 10



→ *single RIGHT rotation*

bf: 2 → 0
root=20, height 2 — single right rotation fired

RL: insert 10, 30, 20



→ *double rotation*

bf: -2 → 0
root=20, height 2 — right rotate 30, then left rotate 10

Three values, however they arrive, have exactly one balanced arrangement — all four cases (LL, RR, LR, RL) converge on root 20, children 10 and 30.

IMPLEMENTATION

AVL Insertion: Choosing the Case

Insertion picks LL/RR/LR/RL by comparing the newly inserted value against node->left->data / node->right->data:

avl_tree.cpp — insert()

```
updateHeight(node);
int balance = balanceFactor(node);
if (balance > 1  && value < node->left->data)  return rotateRight(node);    // LL
if (balance < -1 && value > node->right->data) return rotateLeft(node);     // RR
if (balance > 1  && value > node->left->data) {
    node->left = rotateLeft(node->left);    // LR: straighten first
    return rotateRight(node);
}
if (balance < -1 && value < node->right->data) {
    node->right = rotateRight(node->right);  // RL: straighten first
    return rotateLeft(node);
}
```

DELETION

AVL Deletion, With Rebalancing

- Ordinary BST deletion (Lecture 21's three cases), then the same rebalancing walk insertion uses.
- Key difference: insertion compares the newly inserted value; deletion has no such value — something disappeared, not arrived.
- Instead, the case is chosen from the unbalanced child's own balance factor.

avl_delete.cpp

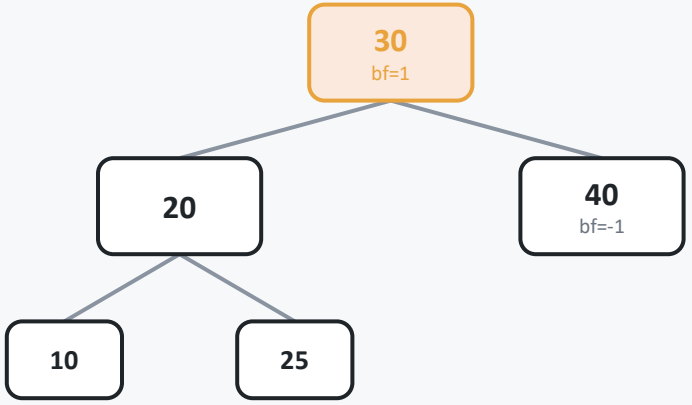
```
// LL / LR: from the LEFT child's own bf
if (balance > 1 && balanceFactor(node->left) >= 0)
    return rotateRight(node);
if (balance > 1) {
    node->left = rotateLeft(node->left);
    return rotateRight(node);
}
// RR / RL mirror this on node->right
```

Common pitfall: copying insertion's four value-comparison ifs into deletion doesn't work — there's no “value just inserted” to compare against after a deletion. Ask the unbalanced child for its own balance factor instead.

WORKED EXAMPLE

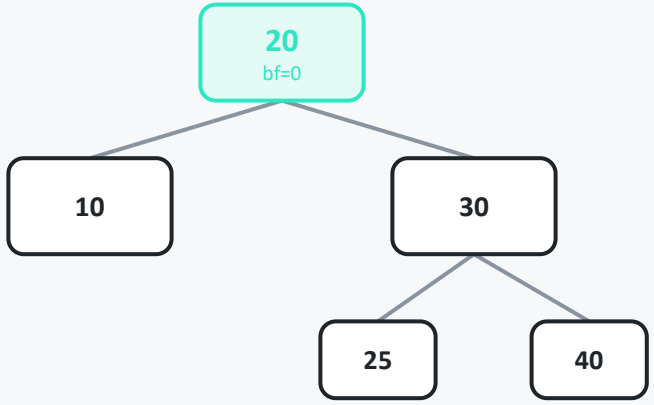
Deletion That Triggers a Rotation

BEFORE: deleting 50 (root bf = 1)



(50 already deleted — 40's only child is gone, pushing root 30's bf from 1 to 2)

AFTER: right rotation at 30



balanceFactor(node->left) = 1 (≥ 0) → single right rotation at 30

Deleting 35 (a leaf) earlier left 40 at bf=-1 — still legal, no rotation. Deleting 50 next removes 40's only child, pushing the root out of range — this is the same LL-style fix, just triggered by shrinking instead of growing.

GUARANTEES

AVL Tree Performance

	Plain BST (worst case)	AVL Tree (always)
Height	O(n)	O(log n) — guaranteed
Search / Insert / Delete	O(n)	O(log n) — guaranteed

"Guaranteed" is the entire point: a plain BST's O(log n) is best case, dependent on insertion order; an AVL tree's O(log n) is a mathematical property of the structure itself.

A TRADE-OFF, NOT A FREE LUNCH

	AVL Tree	Red-Black Tree
Balance guarantee	Strict — height ≤ ~1.44 $\log_2(n)$	Looser — height ≤ 2 $\log_2(n)$
Used by	Fast, lookup-heavy systems	std::map/std::set, Java TreeMap

Key Takeaways

1. An AVL tree is a BST with one added rule: every node's balance factor (left height − right height) must stay within {−1, 0, 1}.

2. Four rotation cases restore balance: LL/RR (single rotation) and LR/RL (double rotation) — all four verified with real balance-factor output.

3. A rotation only ever moves one subtree (T2) across a boundary the BST property already guaranteed was safe to cross.

4. Deletion rebalances the same way insertion does, but chooses its case from the unbalanced child's own balance factor, not a value comparison.

5. Real systems often use red-black trees instead — a looser balance guarantee traded for cheaper updates.