CS 112

Introduction to Data Structures

WEEK 12

AVL Trees: a brief introduction

Eric Araújo

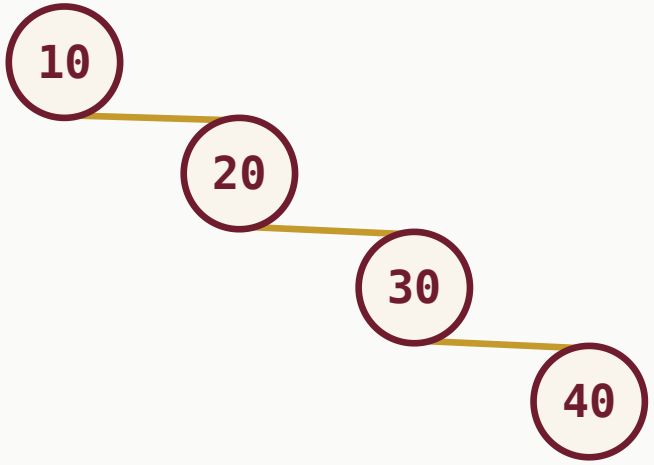
Calvin University · Fall 2026

This Week

1. Last week's problem
2. A self-balancing BST
3. Balance factor
4. A rotation
5. The four cases **KEY**
6. What it costs

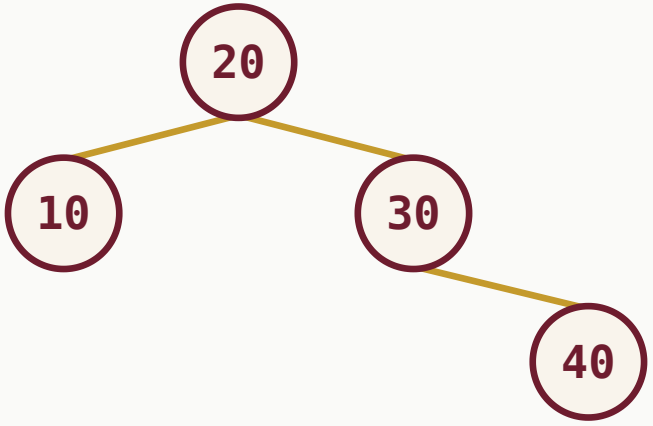
Last Week's Problem

10, 20, 30, 40 in order



height 3, find is **O(n)**

The same values, balanced



height 2, find is **O(lg n)**

> Sorted input is not exotic; it's the most natural way data arrives. A BST that degrades on sorted input degrades on real data.

A Self-Balancing BST

An **AVL tree** is a BST that repairs its own shape after every insert and remove.

- Still a BST: left smaller, right larger, all the old code still applies
- Plus one rule it never breaks
- Named for Adelson-Velsky and Landis, who published it in 1962

> The guarantee: height stays O(lg n) *no matter what order* the values arrive in.

Balance Factor

```
balanceFactor(node) = height(node->left) - height(node->right)
```

- **0**: both sides equally deep
- **+1**: the left is one deeper
- **−1**: the right is one deeper
- **±2 or worse**: the AVL rule is broken; fix it now

> The AVL invariant: **every node's balance factor is −1, 0, or +1**. Check it after each change, and repair the first place it fails on the way back up.

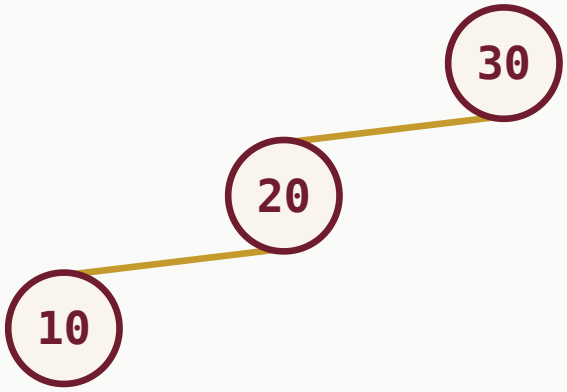
TALK TO YOUR NEIGHBOR · TTYN

A node's left subtree has height 3, its right subtree height 1. What now?

- **A.** Nothing: a difference of 2 is allowed
- **B.** Balance factor is +2, so this node needs rotating
- **C.** Balance factor is −2, so this node needs rotating
- **D.** The whole tree must be rebuilt

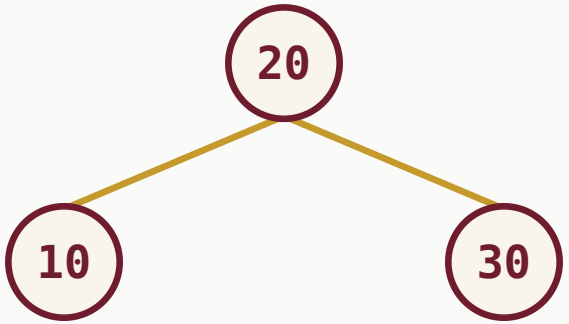
A Rotation

Before: left-heavy



30 has balance factor +2

After: rotate right



20 rises; 30 becomes its right child

Three pointer assignments. The in-order sequence is unchanged, 10, 20, 30 either way, so it is still a valid BST, just shorter.

Rotate Right, in Code

```
Node* rotateRight(Node *p) {
    Node *newRoot = p->myLeft;       // the child that rises
    p->myLeft  = newRoot->myRight;   // its right subtree moves across
    newRoot->myRight = p;            // the old root becomes the child
    updateHeight(p);
    updateHeight(newRoot);
    return newRoot;                  // caller re-links this in
}
```

Constant time, three pointers and two height updates, regardless of how big the subtrees are. That's why rebalancing is cheap.

The Four Cases

Case	Shape	Fix
LL	left child, left grandchild	one right rotation
RR	right child, right grandchild	one left rotation
LR	left child, right grandchild	left on the child, then right
RL	right child, left grandchild	right on the child, then left

Straight line → one rotation. Zig-zag → straighten it first, then rotate. That's the whole decision.

TALK TO YOUR NEIGHBOR · TTYN

Insert 10, then 30, then 20. Which case is this, and what fixes it?

- **A.** LL, one right rotation
- **B.** RR, one left rotation
- **C.** RL, right on the child, then left at the root
- **D.** No rotation needed

What It Costs

Operation	Plain BST (worst)	AVL tree
find	O(n)	**O(lg n)**
insert	O(n)	**O(lg n)**
remove	O(n)	**O(lg n)**
extra memory	none	a height per node
code complexity	modest	considerably more

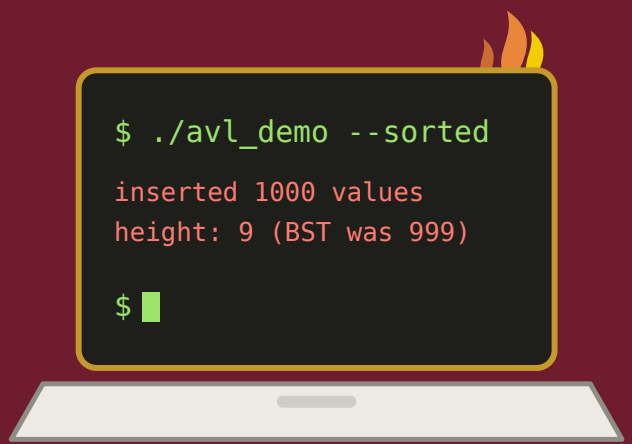
You pay in memory and in code you have to get right. You buy a guarantee that no input order can wreck your performance.

Insert, the AVL Way

- Insert exactly as in a plain BST: it becomes a leaf
- On the way back up the recursion, update each node's height
- At the **first** node whose balance factor hits ±2, rotate
- One rebalance is always enough after an insert

> The unwinding phase from week 10 is doing the work: the recursion already walks back up the exact path that could have become unbalanced.

Demo



Inserting sorted data into an AVL tree.

(the rotations paid for themselves)

Do You Need One?

Yes, when

- Input may arrive sorted or nearly so
- Worst-case latency matters
- Reads and writes are mixed

Probably not, when

- Data is loaded once, then only read
- A sorted array with binary search would do
- You can just use `std::map`

The STL's `map` and `set` are balanced trees, usually red-black rather than AVL, same guarantee, fewer rotations on insert. You'll meet them next week.

Week 12 Recap

- An AVL tree is a BST that **repairs its own shape**
- Balance factor = height(left) − height(right), and must stay in {−1, 0, +1}
- A rotation is three pointer moves: **constant time**
- Straight line → one rotation; zig-zag → two
- Every operation becomes O(lg n) no matter what order the data arrives

Next: a15 →

This deck stands on earlier CS112 materials by
Joel Adams and **Victor Norman** · adapted and extended by **Eric Araújo**