AVL Trees: the First Self-Balancing Search Tree

Lecture Script, Questions with Answers, Worksheet and Solutions
Z5007 Programming and Data Structures, MTech | Week 8, Monday 23 November 2026

Part 1. Overview

A plain search tree is fast only when its height is small, and sorted input turns it into a chain of height $n-1$. An AVL tree (Adelson-Velsky and Landis, 1962) enforces one rule at every node: the heights of the left and right subtrees differ by at most 1. That invariant keeps height bounded by roughly $1.44 \log_2 n$, guaranteeing $O(\log n)$ search, insertion, and deletion.

Check question. A chain of 1,000 keys versus a balanced tree of 1,000 keys: about how many comparisons can worst-case search take?

Answer. Chain: up to 1,000. Balanced tree: at most about 10, since $2^{10} = 1024$.

Part 2. Heights and Balance Factors

Every node stores its height $h(n) = 1 + \max(h(\text{left}), h(\text{right}))$, with empty tree height -1. Its balance factor is $bf(n) = h(\text{left}) - h(\text{right})$. Heights travel upward from children to parents; balance factors stay at their node as a local verdict in $\{-1, 0, +1\}$. Only nodes on the path back from an insert or delete can change height.

Part 3. Rotations and the Four Cases

Table 1: Selecting single vs. double rotations from lowest unbalanced node z ($|bf(z)| = 2$) and taller child y.

bf(z)	bf(y)	Case	Repair
+2	+1 (or 0)	Left-Left	Single right rotation at z
+2	−1	Left-Right	Left rotation at y, then right rotation at z
−2	−1 (or 0)	Right-Right	Single left rotation at z
−2	+1	Right-Left	Right rotation at y, then left rotation at z

```
def rebalance(n):
    update(n)
    b = height(n.left) - height(n.right)
    if b > 1:
        if balance(n.left) < 0:
            n.left = rotate_left(n.left)
        return rotate_right(n)
    if b < -1:
        if balance(n.right) > 0:
            n.right = rotate_right(n.right)
        return rotate_left(n)
    return n
```

Check question. Why can AVL deletion trigger rotations at multiple ancestors up to the root, whereas insertion stops after at most one repair?

Answer. An insertion repair restores the subtree's pre-insertion height, leaving ancestors unchanged. Deletion can shorten a subtree by 1 even after rotation, propagating imbalance upward.

Worksheet (Students Fill In)

1. Insert `60, 30, 40, 80, 70, 90, 75` into an empty AVL tree, computing balance factors after every insert.
2. Whenever a node hits ± 2, identify z and y, name the case, and draw the repaired tree.
3. Delete `30` from the final tree and report whether any rotation occurs plus the final inorder sequence.

Worksheet Solutions

Inserting `40` triggers LR at `60` ($bf(60) = +2, bf(30) = -1$) $\rightarrow$ `40(30, 60)`. Inserting `70` after `80` triggers RL at `60` ($bf(60) = -2, bf(80) = +1$) $\rightarrow$ `40(30, 70(60, 80))`. Inserting `90` triggers RR at `40` ($bf(40) = -2, bf(70) = -1$) $\rightarrow$ `70(40(30, 60), 80(-, 90))`. Inserting `75` needs no rotation. Deleting leaf `30` leaves $bf(40) = -1, bf(70) = 0$: balanced, final inorder `40 60 70 75 80 90`.

References

1. Adelson-Velsky, G. M. and Landis, E. M. (1962). An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146(2), 263–266.
2. Canning, J., Broder, A. and Lafore, R. (2022). *Data Structures and Algorithms in Python*. Addison-Wesley. Chapter 8.
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2022). *Introduction to Algorithms*, 4th ed. MIT Press. Chapter 13.
4. Knuth, D. E. (1998). *The Art of Computer Programming*, Vol. 3, 2nd ed. Addison-Wesley. Section 6.2.3.