

Indian Institute of Technology Madras
ZANZIBAR CAMPUS

Z5007 · PROGRAMMING AND DATA STRUCTURES

AVL Trees: the First Self-Balancing Search Tree

Week 8 · Monday 23 November 2026 · Canning, Broder & Lafore Ch 8

Dr. Innocent Nyalala | MTech Semester 1 | IIT Madras Zanzibar

Today's Plan

IIT Madras Zanzibar

1. (0–6 min) What an AVL tree is, who invented it, and why it matters
2. (6–14 min) Recap: heights and balance factors, exactly what we compute from what
3. (14–20 min) Deleting again: which replacement is correct, and why both can be
4. (20–33 min) Rotations: left, right, the four cases, a full insertion trace
5. (33–40 min) Deletion with rotations, the code, practice

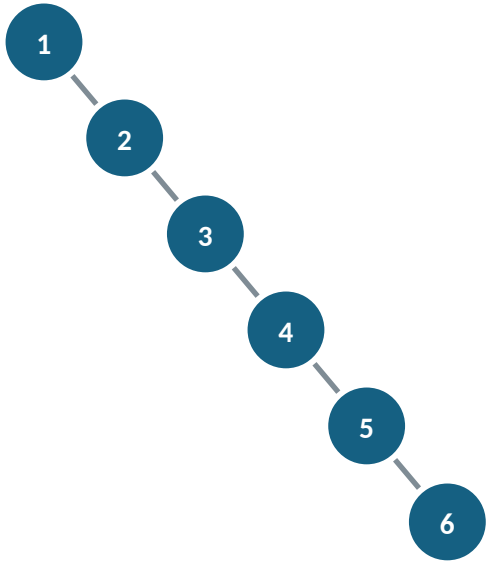
Learning Outcomes

IIT Madras Zanzibar

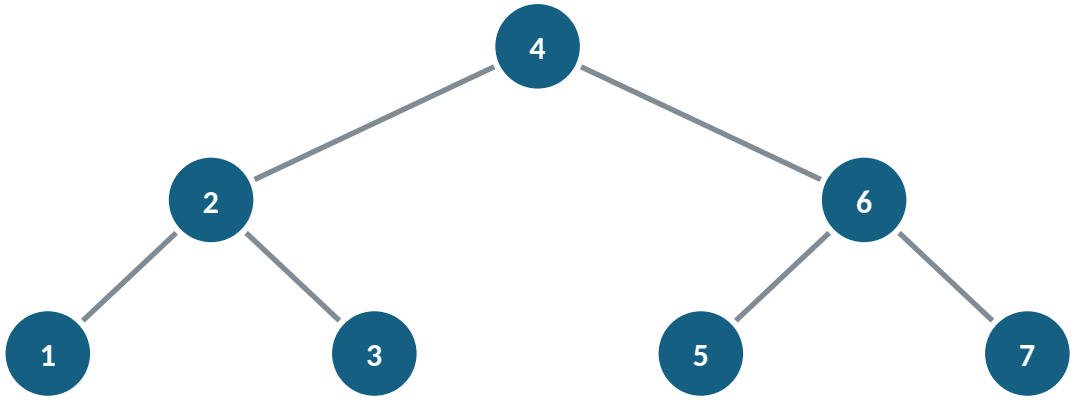
By the end of this session, you will be able to:

Explain	what an AVL tree guarantees, who invented it, and why it is worth the extra work
Compute	heights and balance factors correctly, using heights and never the balance factors of children
Delete	from a search tree with either valid replacement and check the result on your own drawing
Apply	single and double rotations to repair any of the four cases, for insertion and deletion

Where We Stopped on Friday



plain search tree, sorted input: height n − 1



balanced: height about $\log_2 n$

Goal: keep the height near $\log_2 n$ for every insertion order, automatically.

Think: A chain of 1,000 keys and a perfect tree of 1,000 keys: about how many comparisons does one worst-case search need in each?

Answer: Chain: up to 1,000. Perfect tree: about 10, because $2^{10} = 1{,}024$.

AVL Trees: the Big Picture

IIT Madras Zanzibar

1 What it is

a binary search tree where, at every node, the heights of the left and right subtrees differ by at most 1

2 Properties

still a search tree (inorder is sorted); height at most about 1.44 $\log_2 n$; every node keeps its height

3 Why we need it

plain trees degrade to chains on sorted input; AVL guarantees O(log n) search, insert, delete

4 Where it is used

ordered sets and maps, in-memory indexes, anywhere lookups dominate and order matters

5 Why it matters

the first structure that repairs itself with local rotations; the idea returns in red-black trees and B-trees

? **Think:** An AVL tree has 1,000,000 keys. At most how many comparisons can one search need?

Answer: Height at most 27 (Friday: N(27) is just under a million), so at most 28 comparisons. A chain needs up to 1,000,000.

Who Invented It, and Where the Name Comes From

IIT Madras Zanzibar

Georgy Adelson-Velsky

mathematician and computer scientist, 1922 to 2014

Evgenii Landis

mathematician, 1921 to 1997

AVL = A + V + L

the initials of their surnames

The paper

An algorithm for the organization of information

Doklady Akademii Nauk SSSR, 146 (1962), 263 to 266

Why it is famous

the first self-balancing binary search tree

the idea behind every later balanced tree: repair locally after each update

? **Think:** Why would two mathematicians in 1962 care about the height of a search tree?

Answer: Time and memory were scarce. Height n means n steps per search; height log n means a handful. For a million items that decides whether the system is usable.

Recap: Height and Balance Factor, Which Uses Which

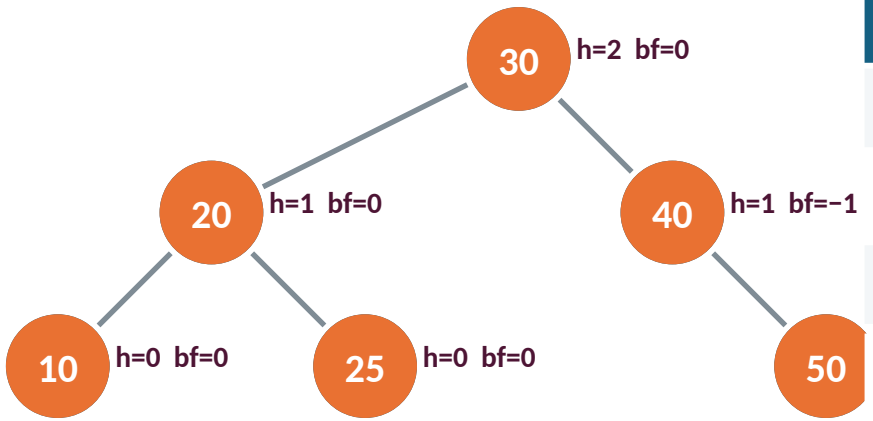
IIT Madras Zanzibar

HEIGHT of a node	BALANCE FACTOR of a node
edges on the longest path down to a leaf	a verdict on this one node
empty tree −1, leaf 0	left subtree height − right subtree height
uses the HEIGHTS of the two children: h = 1 + max(h left, h right)	uses the HEIGHTS of the two children, NEVER their balance factors
passed UP to the parent	not passed anywhere; only checked: −1, 0 or +1

? **Think:** Can two nodes have the same balance factor but very different heights?

Answer: Yes: a leaf and the root of a perfect tree of height 3 both have balance factor 0, with heights 0 and 3.

Recap, Step by Step: Heights First, Then Balance Factors



node	h left	h right	own height	bf = L − R
10	−1	−1	1+max(−1,−1)=0	−1 − (−1) = 0
25	−1	−1	1+max(−1,−1)=0	−1 − (−1) = 0
20	0	0	1+max(0,0)=1	0 − 0 = 0
50	−1	−1	1+max(−1,−1)=0	−1 − (−1) = 0
40	−1	0	1+max(−1,0)=1	−1 − 0 = −1
30	1	1	1+max(1,1)=2	1 − 1 = 0

Trap: bf(20) − bf(40) = 0 − (−1) = +1 is WRONG. From the heights: bf(30) = 1 − 1 = 0.

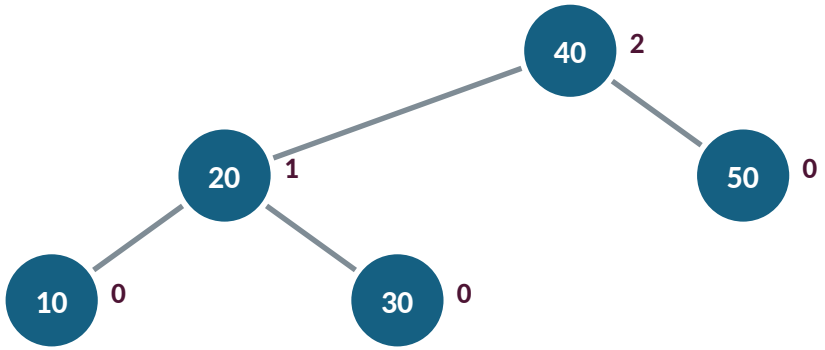
? **Think:** Which numbers do we pass up from children to a parent: the heights or the balance factors?

Answer: The heights. Balance factors stay at their own node and are only checked.

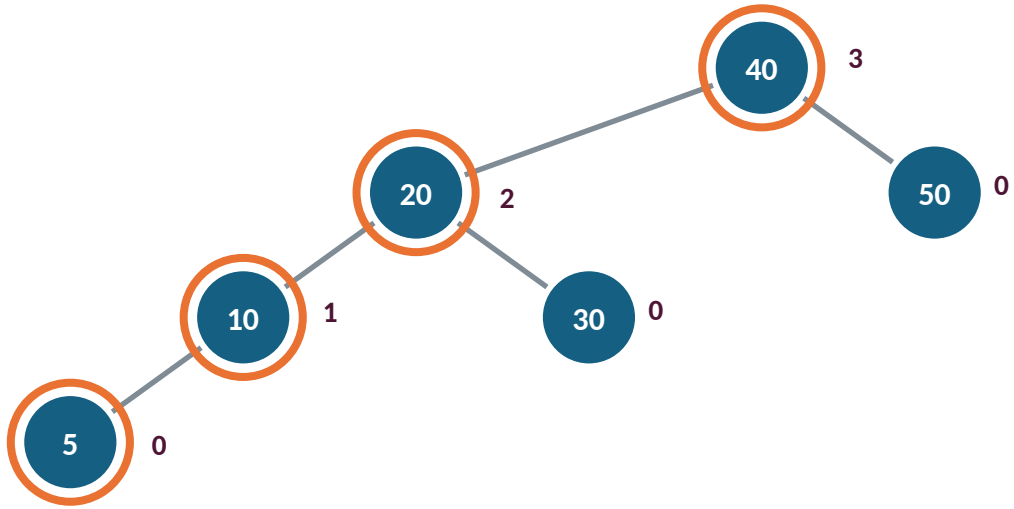
After a Change, Only the Path to the Root Needs New Heights

IIT Madras Zanzibar

before: heights on the nodes



after inserting 5: new heights



Path only: 5, 10, 20, 40 get new heights. 30 and 50 do not. Then bf(40) = 2 − 0 = +2: a violation.

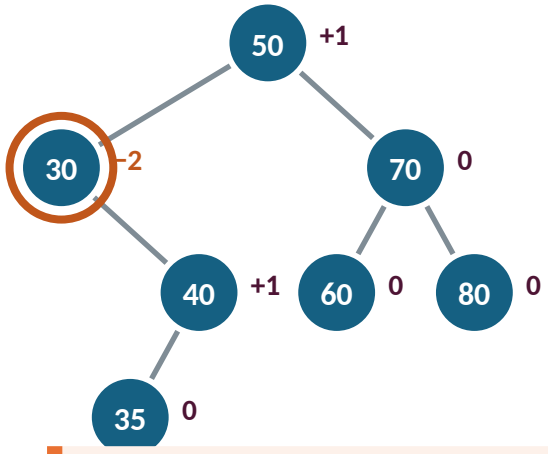
? **Think:** Why is it enough to check balance factors only along the path from the change to the root?

Answer: A balance factor depends only on the two subtrees' heights, and only subtrees on that path changed.

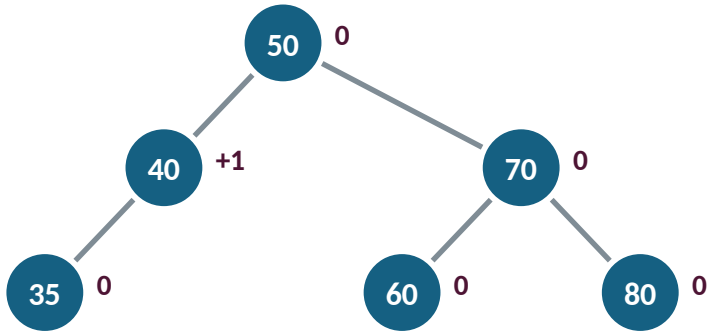
Delete a Node with One Child: Both Replacements Are Correct

IIT Madras Zanzibar

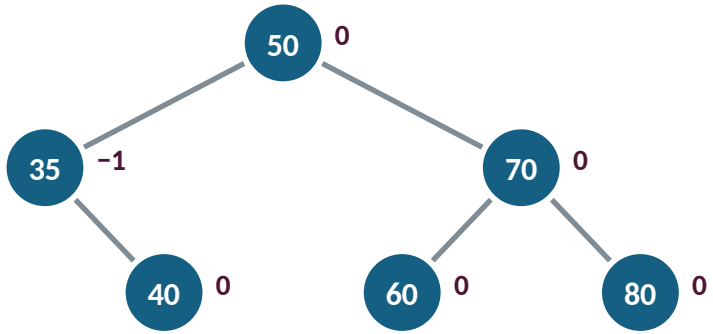
delete 20: bf(30) = −2



A: promote the child 40



B: copy successor 35



Both valid: inorder 35 40 50 60 70 80 either way. They differ only in that node's balance factor: +1 in A, −1 in B. Class rule: A (the child takes its parent's place).

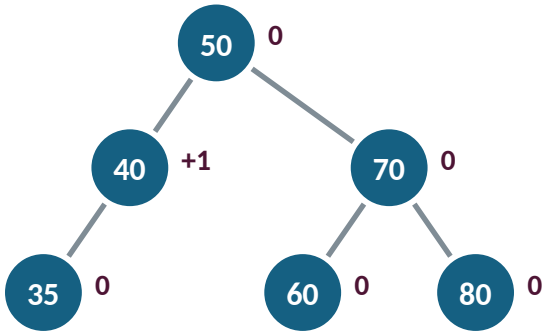
? **Think:** After deleting 20 and before deleting 30, is the tree balanced?

Answer: No: bf(30) = −1 − 1 = −2. It is balanced again only because the next delete removes node 30.

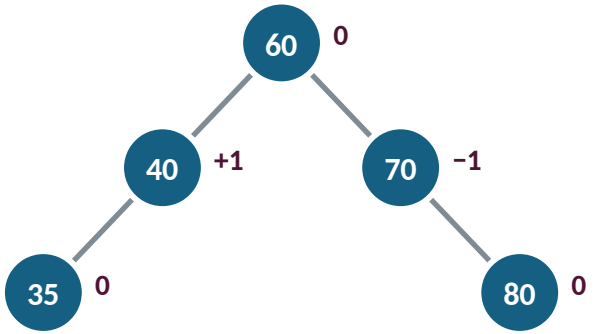
Delete a Node with Two Children: Successor or Predecessor

IIT Madras Zanzibar

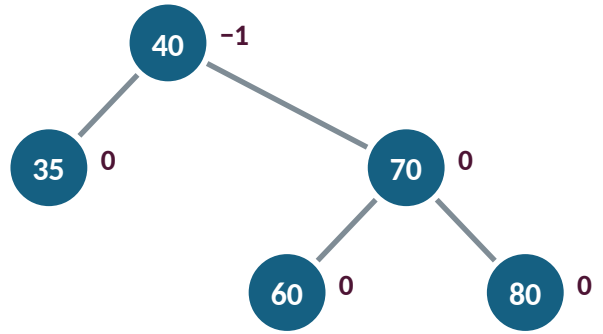
delete 50 from this tree



successor 60 (class rule)



predecessor 40



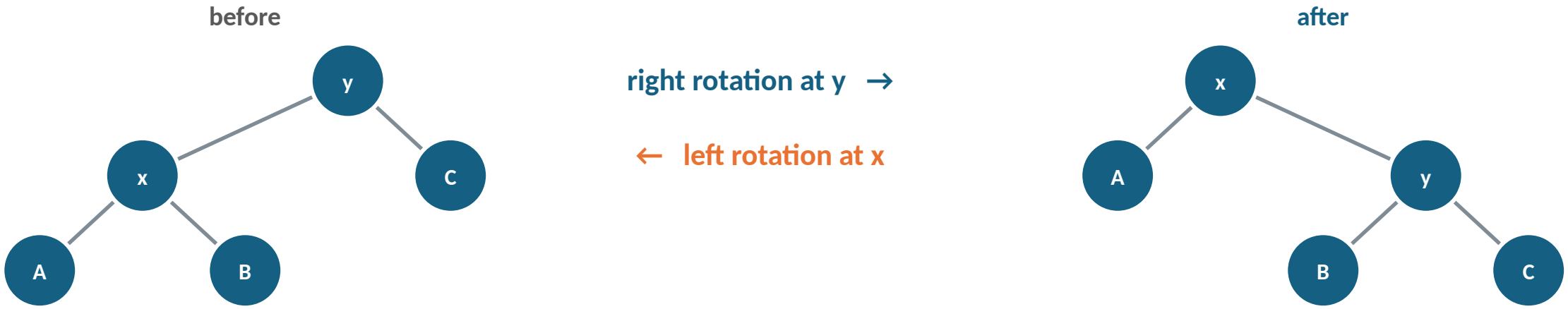
Rule: either neighbor of the deleted key in sorted order may take its place. Then recompute the balance factors on YOUR drawing. We use the successor so that we all draw the same tree.

?

Think: Where do the successor and the predecessor sit in the sorted order relative to the deleted key, and what does that tell you?

Answer: They are its two neighbors, one just above and one just below, so either can take its place without breaking the search rule.

A Rotation Is a Local Repair That Keeps the Order

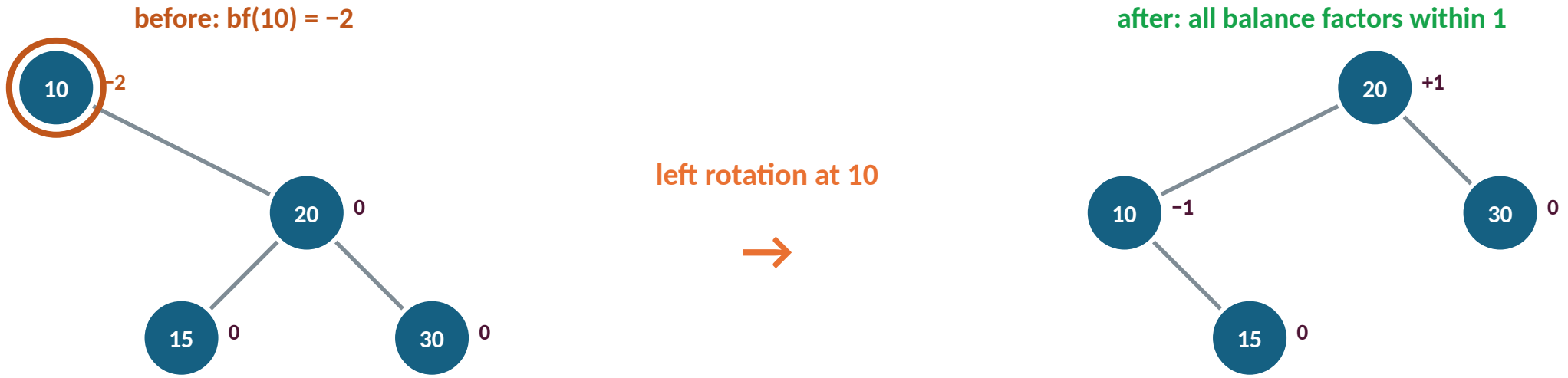


Only B moves: in sorted order A < x < B < y < C. B is between x and y, so it sits in the one slot that is free after the pivot: x's right slot before, y's left slot after. The inorder A x B y C is the same in both trees.

? **Think:** After a right rotation at y, how many nodes change their parent?

Answer: Three: x (it takes y's old parent), y (its new parent is x) and B (from x to y). Everything else keeps its parent.

Left Rotation, Step by Step



1. 20 moves up into 10's place
2. 10 moves down, to the left of 20
3. 15 (20's old left child) moves under 10, into 10's free right slot

? **Think:** Where is 15 relative to 10 and 20 before, and why does it fit under 10 afterwards?

Answer: Before: larger than 10, smaller than 20 (it is 20's left child). Afterwards it is 10's right child: larger than 10 and still smaller than 20. Order kept.

The Four Cases: Read the Two Balance Factors

IIT Madras Zanzibar

bf(z)	bf(y)	case	fix	shape
+2	+1 (or 0)	left-left	right rotation at z	z, y, x go down to the left twice
+2	−1	left-right	left at y, then right at z	zigzag: left, then right
−2	−1 (or 0)	right-right	left rotation at z	z, y, x go down to the right twice
−2	+1	right-left	right at y, then left at z	zigzag: right, then left

Rule of thumb: same signs: one rotation. Opposite signs: two (first at y to straighten the zigzag, then at z). z is the LOWEST violating node.

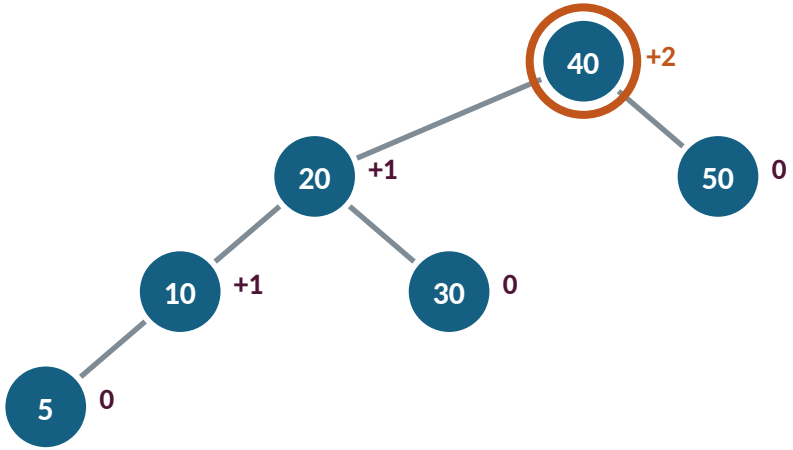
? **Think:** bf(z) = −2 and its right child y has bf(y) = +1. Which case, and which rotations?

Answer: Right-left (opposite signs): rotate right at y, then left at z.

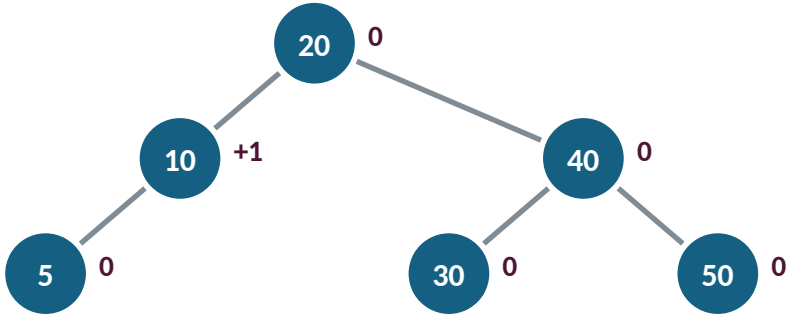
Case 1, Left-Left: Insert 5, One Right Rotation

IIT Madras Zanzibar

insert 5: bf(40) = +2



right rotation at 40



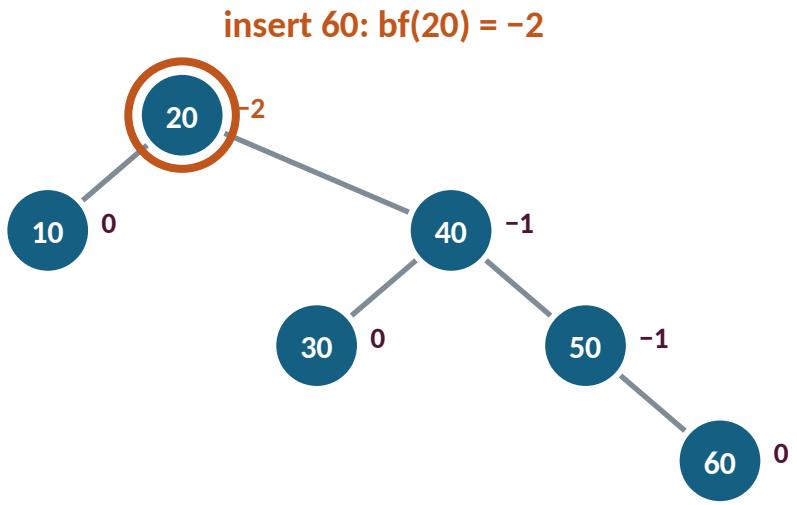
z = 40 (bf +2), y = 20 (bf +1): same signs, so ONE right rotation at 40.

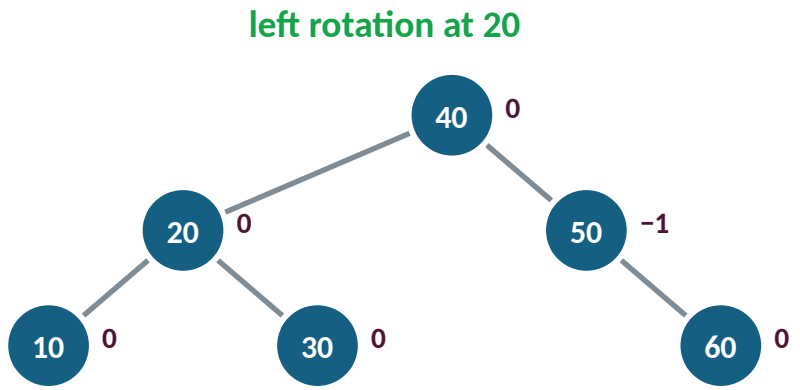
20 moves up, 40 moves down to its right, 30 moves from 20's right to 40's left.

? **Think:** Which node moves from 20 to 40, and why is that the only possible place for it?

Answer: Node 30: larger than 20, smaller than 40, and the only free slot is 40's left.

Case 2, Right-Right: Insert 60, One Left Rotation





z = 20 (bf −2), y = 40 (bf −1): same signs, so ONE left rotation at 20.

40 moves up, 20 moves down to its left, 30 moves from 40's left to 20's right.

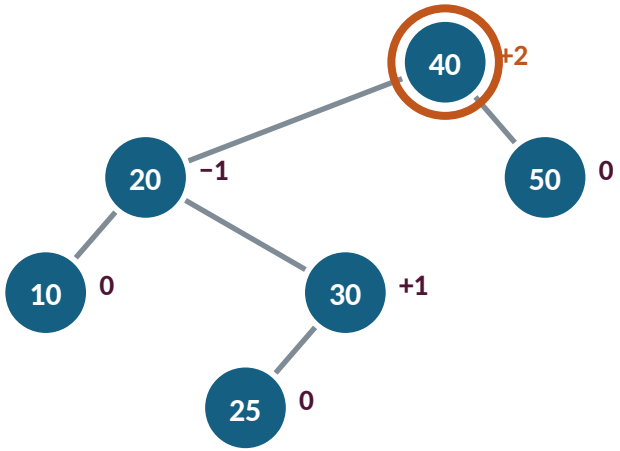
? **Think:** In a left rotation, which subtree changes parents, and where is it in sorted order?

Answer: y's left subtree (node 30 here). It lies between z = 20 and y = 40 in sorted order.

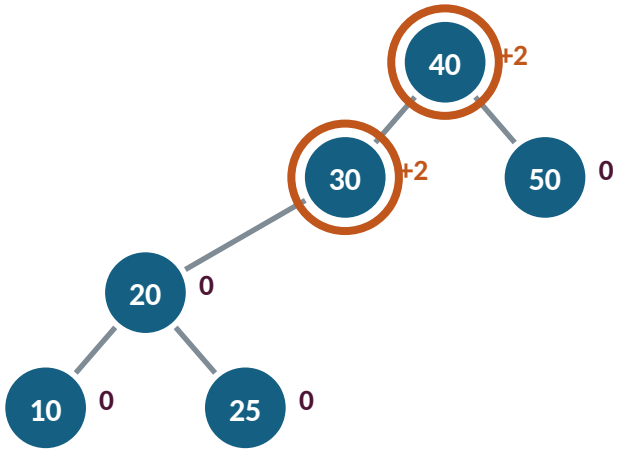
Case 3, Left-Right: Insert 25, Two Rotations

IIT Madras Zanzibar

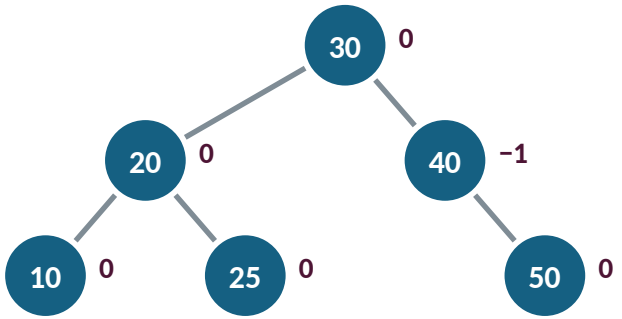
insert 25: bf(40) = +2, bf(20) = −1



1. left rotation at 20



2. right rotation at 40



z = 40 (bf +2), y = 20 (bf −1): opposite signs, so TWO rotations.

First a left rotation at y = 20: the tree becomes a left-left chain 40, 30, 20 (+2 at both 40 and 30 for now). Then a right rotation at z = 40 repairs both.

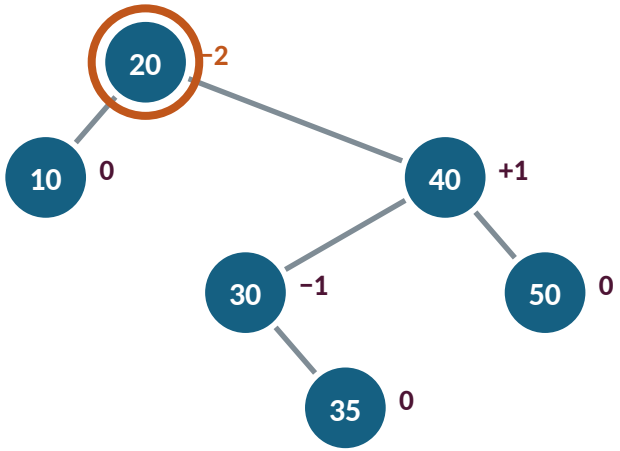
? **Think:** After the first rotation, what is bf(40), and which case is the tree in?

Answer: Still +2. The first rotation only turns the zigzag into a left-left chain; the second rotation repairs it.

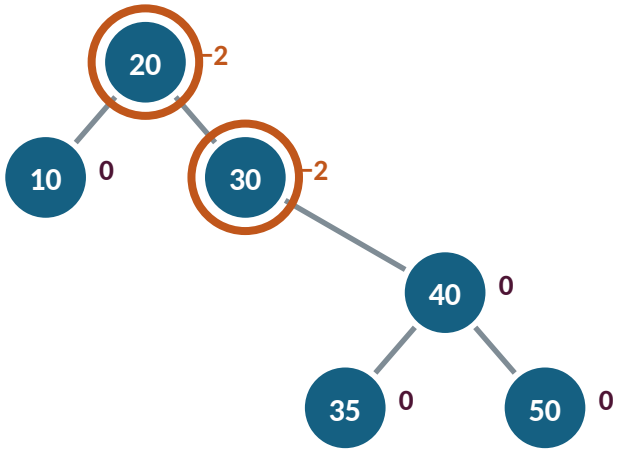
Case 4, Right-Left: Insert 35, Two Rotations

IIT Madras Zanzibar

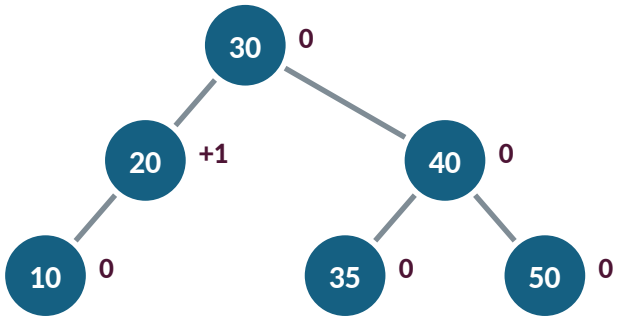
insert 35: bf(20) = −2, bf(40) = +1



1. right rotation at 40



2. left rotation at 20



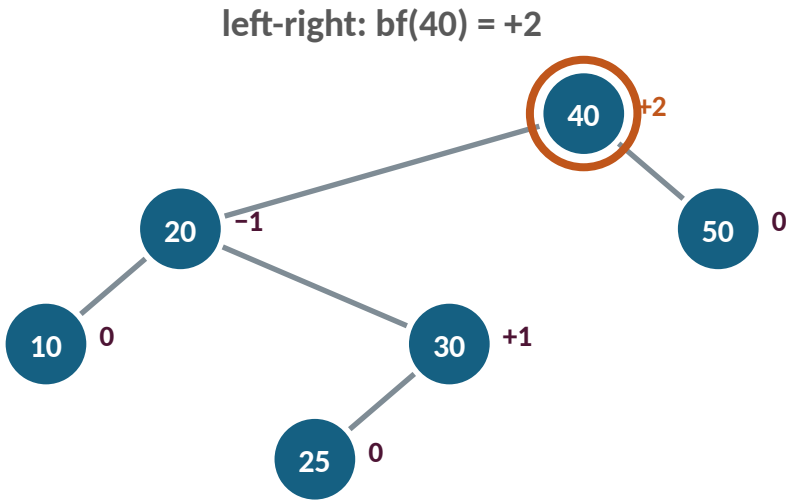
z = 20 (bf −2), y = 40 (bf +1): opposite signs, so TWO rotations.

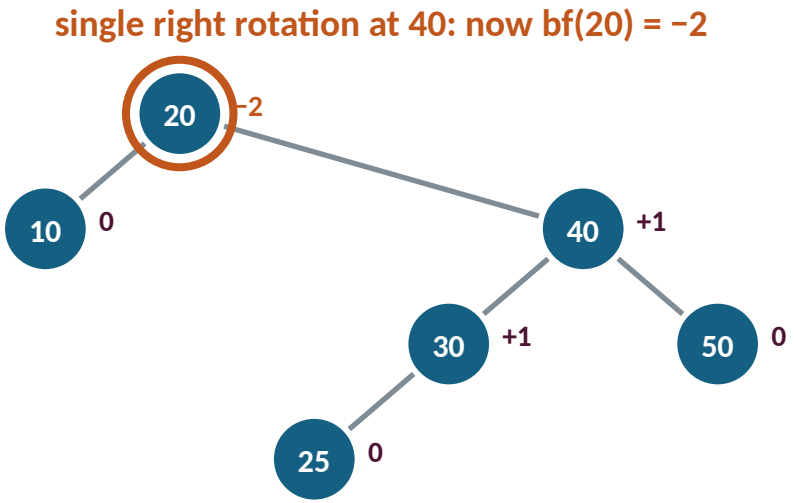
First a right rotation at y = 40: the tree becomes a right-right chain 20, 30, 40 (−2 at both 20 and 30 for now). Then a left rotation at z = 20 repairs both.

? **Think:** Case three used left at y, then right at z. Which pair does case four use?

Answer: The mirror: right rotation at y = 40, then left rotation at z = 20.

Why a Single Rotation Cannot Fix the Zigzag





Reason: the extra height is in the MIDDLE subtree (under 30). One rotation moves it to the other side but does not lower it.

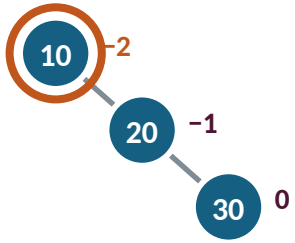
? **Think:** Where is the extra height in the left-right case, and why does one rotation not lower it?

Answer: In the middle subtree (under 30). One rotation only moves it across; two rotations lift it up.

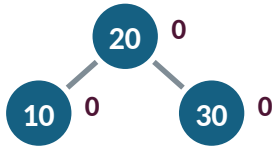
A Full Insertion Trace: 10, 20, 30, 40, 50, 25

IIT Madras Zanzibar

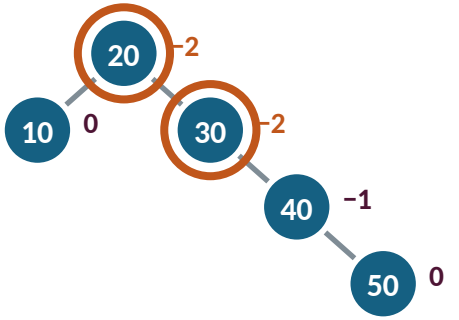
insert 30: bf(10) = −2 (right-right)



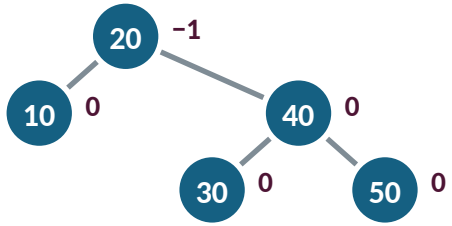
after left rotation at 10



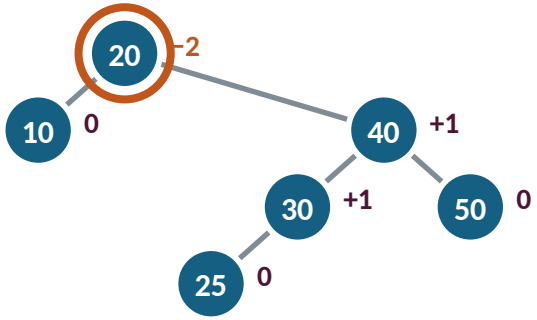
insert 40, 50: bf(30) = −2 (right-right)



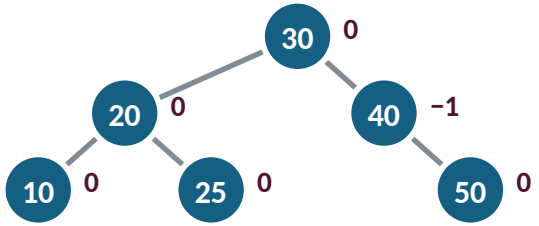
after left rotation at 30



insert 25: bf(20) = −2 (right-left)



after right at 40, then left at 20

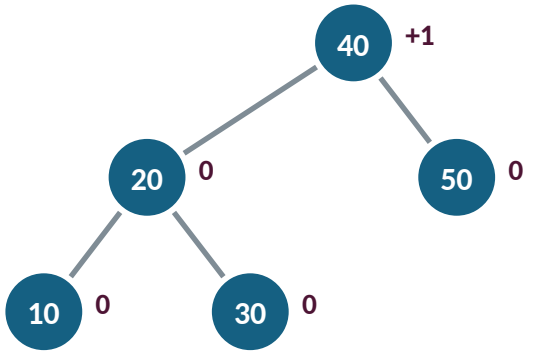


? **Think:** When 50 was inserted both 20 and 30 had bf = −2. Why do we repair 30 and not 20?

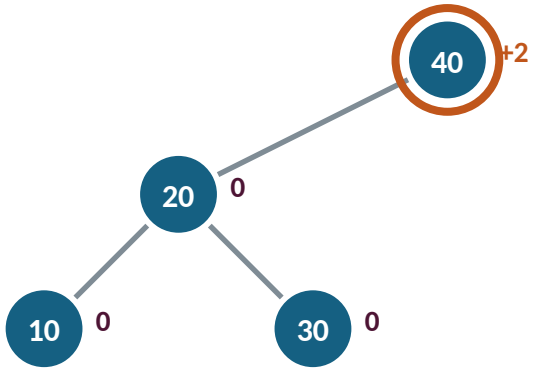
Answer: We repair the LOWEST violating node. Fixing 30 lowers the right side of 20 as well, so 20 is repaired too.

Deletion Needs Rotations Too, and a Fifth Situation Appears

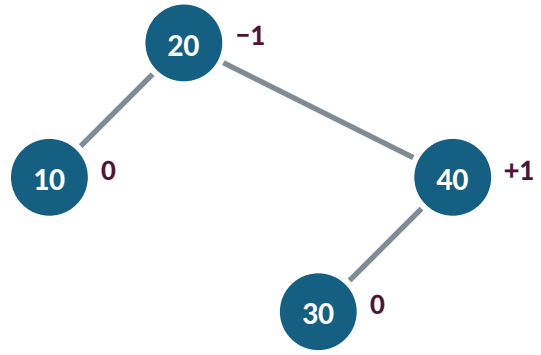
balanced



delete 50: bf(40) = +2, bf(20) = 0



right rotation at 40



New situation: in a deletion the taller child y can have bf = 0. One rotation (right at 40) is enough. The repaired subtree may be ONE LOWER than before, so keep checking upward: a deletion can need a repair at every level.

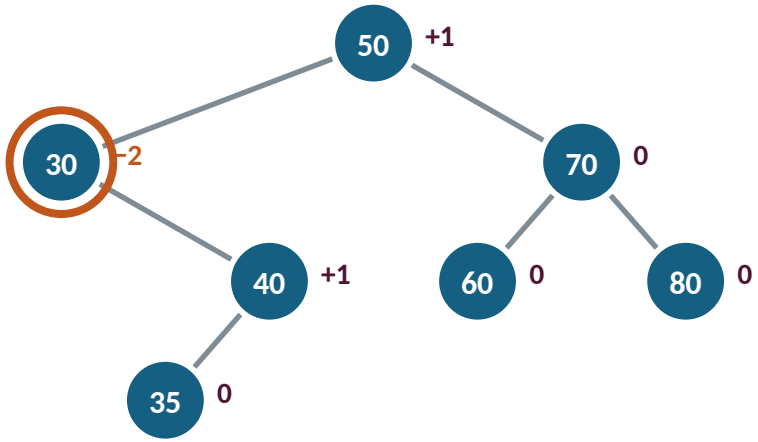
? **Think:** Why can a deletion need rotations at several levels, while an insertion needs at most one repair?

Answer: An insertion repair restores the old subtree height, so nothing above changes. A deletion repair can leave the subtree one lower, so the imbalance can move up.

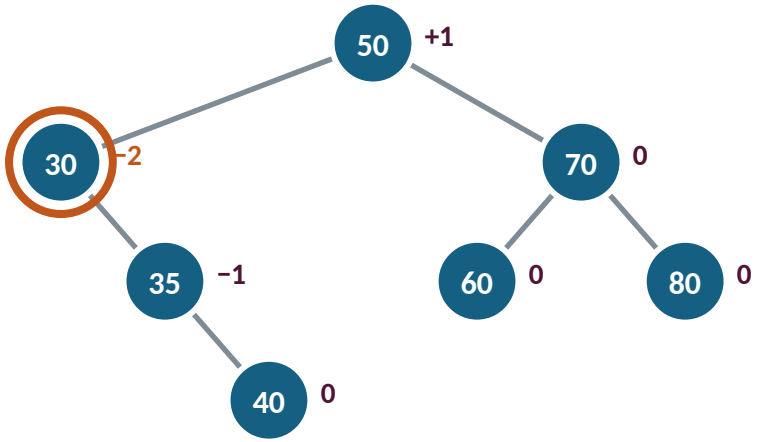
Friday's Tree, Done as a Real AVL Tree

IIT Madras Zanzibar

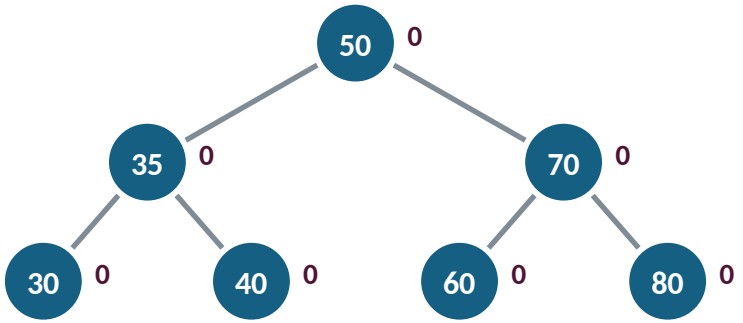
delete 20: bf(30) = −2, bf(40) = +1



1. right rotation at 40



2. left rotation at 30



So: a real AVL tree repairs at once. A search tree for the same keys is not unique; every AVL tree satisfies the rule, and the same inorder 30 35 40 50 60 70 80.

? **Think:** The three trees A, B and this one all hold the same keys. What do they share, and what differs?

Answer: They share the inorder (30 35 40 50 60 70 80), so all are valid search trees. They differ in shape and balance factors.

Code: Heights and Rotations

```
 1  def height(n):
 2      return n.h if n else -1
 3
 4  def update(n):
 5      n.h = 1 + max(height(n.left), height(n.right))
 6
 7  def rotate_right(y):
 8      x = y.left
 9      y.left = x.right
10      x.right = y
11      update(y); update(x)
12      return x
```

Line by line

Line 1-2 an empty side has height −1, otherwise the stored height

Line 4-5 recompute ONE node from its children's heights

Line 8 x is y's left child: it moves up

Line 9 x's right subtree (the middle one) becomes y's left

Line 10 y moves down to the right of x

Line 11 y first, since it is now x's child; then x

Line 12 return x: the caller hangs it where y was

? **Think:** What goes wrong if line 11 updated x before y?

Answer: x's height would be computed from y's old height, so it would be wrong, and so would every balance factor above.

Code: Insert with Rebalancing

```
 1  def insert(n, key):
 2      if n is None: return Node(key)
 3      if key < n.key: n.left = insert(n.left, key)
 4      else: n.right = insert(n.right, key)
 5      update(n)
 6      b = height(n.left) - height(n.right)
 7      if b > 1:
 8          if bf(n.left) < 0: n.left = rotate_left(n.left)
 9          return rotate_right(n)
10      if b < -1:
11          if bf(n.right) > 0: n.right = rotate_right(n.right)
12          return rotate_left(n)
13      return n
```

Line by line

Line 2-4 ordinary search tree insertion, recursively

Line 5-6 on the way back: new height, balance factor b of THIS node

Line 7-9 left-heavy: if the left child leans right (zigzag), rotate it left first; then rotate right at n

Line 10-12 the mirror for right-heavy

Line 13 balanced: nothing to do

Line 3-4 n.left = insert(...) stores a possibly NEW root of the subtree

? **Think:** Why does line 3 assign the result of insert back into n.left?

Answer: A rotation may have put a different node at the top of that subtree; the assignment hangs it back in place.

Cost and Guarantee

IIT Madras Zanzibar

	plain search tree	AVL tree	extra cost
height	up to n − 1	at most about 1.44 $\log_2 n$	store h in each node
search	O(h), up to n	O(log n)	none
insert	O(h)	O(log n), at most 2 rotations	recompute heights on the path
delete	O(h)	O(log n), up to log n rotations	check every level up

In practice: AVL is a little faster for lookups than red-black (tighter balance), a little slower for updates (more rotations). Wednesday: red-black trees.

? **Think:** Insert n sorted keys into an empty tree. Total cost with AVL, and with a plain search tree?

Answer: AVL: $n \times \log n = O(n \log n)$. Plain: $1 + 2 + \ldots + n = O(n^2)$, because the tree becomes a chain.

In-Class Exercise (7 minutes, in pairs)

IIT Madras Zanzibar

IN-CLASS TASK: insert, repair, delete

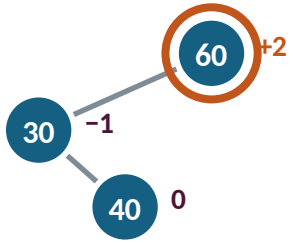
Three tasks (paper)

1. Insert 60, 30, 40, 80, 70, 90, 75 into an empty AVL tree, one key at a time. After EVERY insertion write the balance factors.
2. Whenever a node has +2 or −2, name z and y, write bf(z) and bf(y), name the case and draw the tree after the rotation(s).
3. Delete 30 from the final tree, using the successor if there is a choice. Is any repair needed? Give the final inorder.

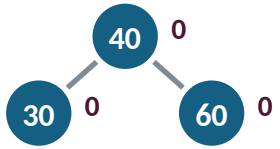
Solution: Insert 60, 30, 40, 80, 70, 90, 75

IIT Madras Zanzibar

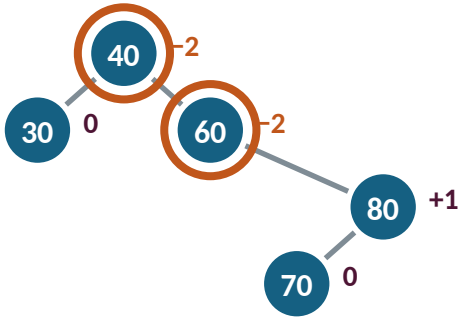
insert 40: LR at 60



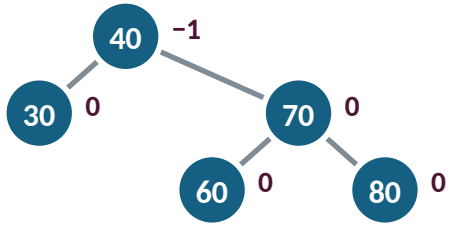
after: left at 30, right at 60



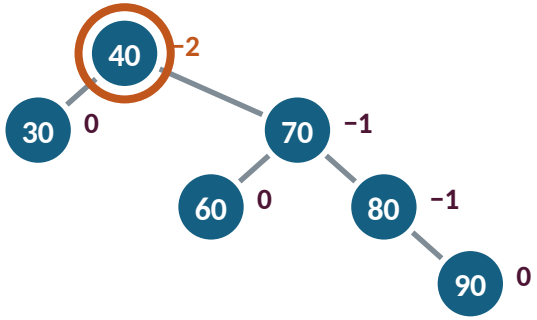
insert 70: RL at 60



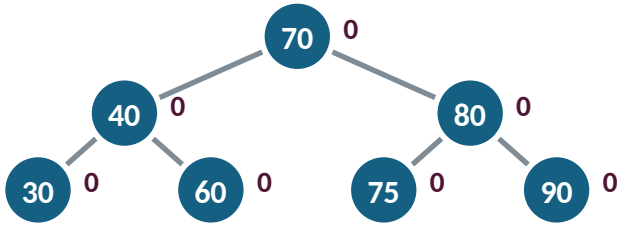
after: right at 80, left at 60



insert 90: RR at 40



after insert 75 (final)



Think: How many rotations did the seven insertions need in total, and what is the height of the final tree?

Answer: Five: two (LR) + two (RL) + one (RR). The final tree has height 2.

Summary and Exam Checkpoints

IIT Madras Zanzibar

Height vs balance factor

bf = h(left) − h(right), computed from HEIGHTS; heights go up, balance factors never do

Delete

one child: promote it; two children: successor or predecessor. Both valid; compute bf on your drawing

Four cases

z = lowest node with bf ±2; same signs one rotation, opposite signs two rotations

Deletion differs

child bf can be 0 (one rotation); repair can lower the subtree, so check up to the root

Next

Wednesday: Red-black trees: a looser balance with fewer rotations, as used in real libraries

Thursday: Heaps: min-heap, max-heap, heap sort and priority queues

Lab: AVL tree in Python, tested with random inserts and deletes

References and Further Reading

[1] Adelson-Velsky, G. M. & Landis, E. M. (1962). An algorithm for the organization of information. Doklady Akademii Nauk SSSR, 146(2), 263 to 266.

[2] Canning, J., Broder, A. & Lafore, R. (2022). Data Structures & Algorithms in Python. Addison-Wesley. Chapter 8.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press. Chapter 13 (balanced search trees).

[4] Knuth, D. E. (1998). The Art of Computer Programming, Vol. 3: Sorting and Searching (2nd ed.). Addison-Wesley. Section 6.2.3.