

Indian Institute of Technology Madras
ZANZIBAR CAMPUS

Z5007 · PROGRAMMING AND DATA STRUCTURES

# Deleting from a Search Tree, and Keeping the Tree Balanced

*Week 7 · Friday 20 November 2026 · Canning, Broder & Lafore Ch 8*

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

# Today's Plan

IIT Madras Zanzibar

1. (0–5 min) The two problems: deletion leaves a hole, and the shape can degrade
2. (5–17 min) Delete: three cases, the successor, the code, a worked sequence
3. (17–25 min) Why balance matters; the balance factor and the AVL rule
4. (25–35 min) Rotations, the four cases, and why AVL height is about log n
5. (35–40 min) Which structure for which job; practice

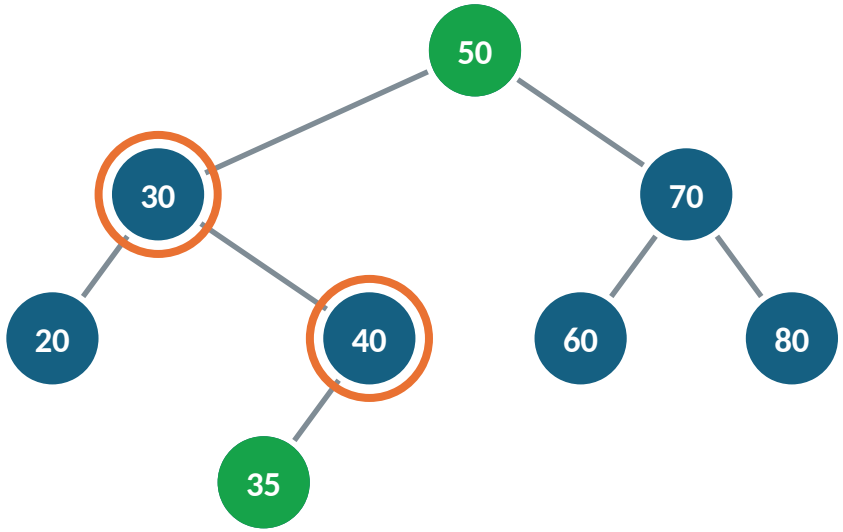
# Learning Outcomes

IIT Madras Zanzibar

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

| | |
|---|---|
| **Delete** | a key from a search tree by hand and in code, in each of the three cases |
| **Explain** | why the successor can replace a deleted node without breaking the rule |
| **Compute** | balance factors, and find where the AVL rule is violated |
| **Apply** | single and double rotations, and explain why AVL height is at most about $1.44 \log_2 n$ |

# Three Words We Need: Successor, Predecessor, Rotation




inorder (sorted order):

| 20 | 30 | 35 | 40 | 50 | 60 | 70 | 80 |
|---|---|---|---|---|---|---|---|

**Successor:** the next larger key, the next one in the inorder. Predecessor: the next smaller key.

**Right subtree:** go right once, then left as far as you can. Successor of 30: right to 40, left to 35.

**No right subtree:** go up to the nearest ancestor whose LEFT subtree holds the node. Successor of 40 is 50, above it.

**So:** the successor is not always on the right. Predecessor is the mirror: left once, then right; or up.

**Rotation:** a child moves up into its parent's place, the parent moves down. Inorder unchanged.

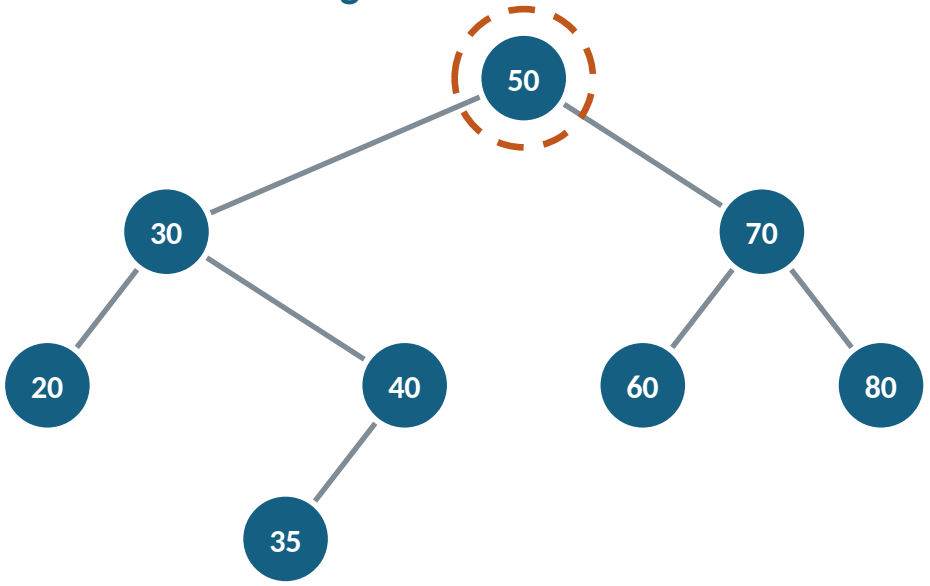
? **Think:** What is the successor of 35 in this tree, and is it below 35 or above it?

**Answer:** 35 has no right subtree, so look up: 35 is the LEFT child of 40. The successor is 40, an ancestor above 35, not a node to its right.

# Two Problems Left: a Hole, and a Bad Shape

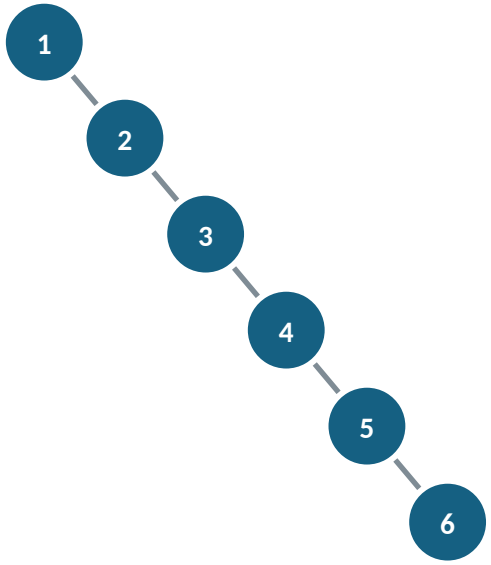
IIT Madras Zanzibar

### Problem 1: deleting leaves a hole




remove 50: who becomes the root?

### Problem 2: the shape can degrade

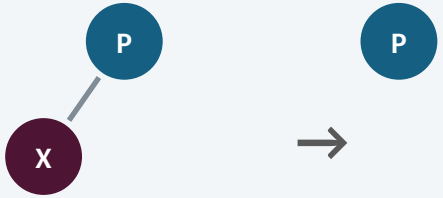



keys inserted in sorted order: every operation costs n

? **Think:** Why can't we simply remove a node with two children and hang both of its subtrees from its parent?

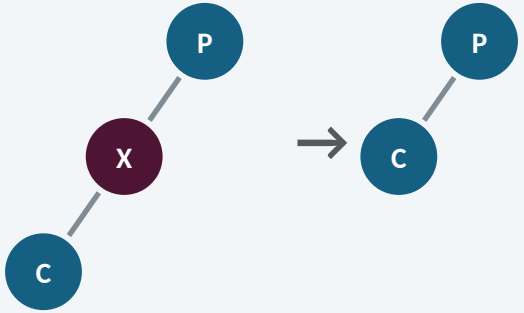
**Answer:** The parent has one free child slot but there are two subtrees. And the key that takes the place must exceed everything on the left and be below everything on the right.

# Delete: Three Cases



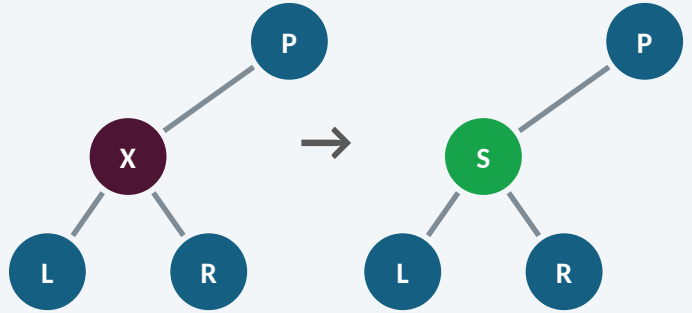

### Case 1: a leaf

remove it. Nothing else changes.




### Case 2: one child

the child takes the place of its parent.



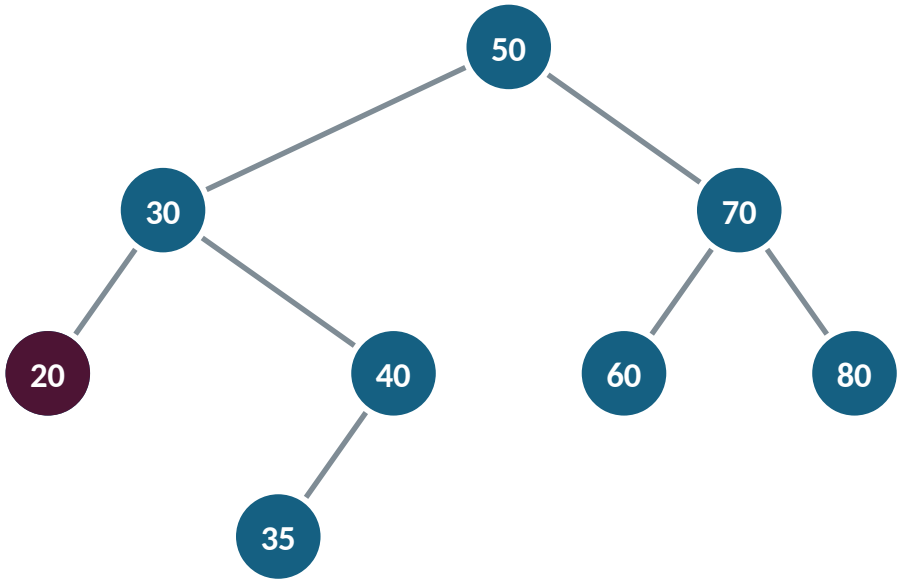

### Case 3: two children

copy the successor S into X, then delete S from the right subtree.

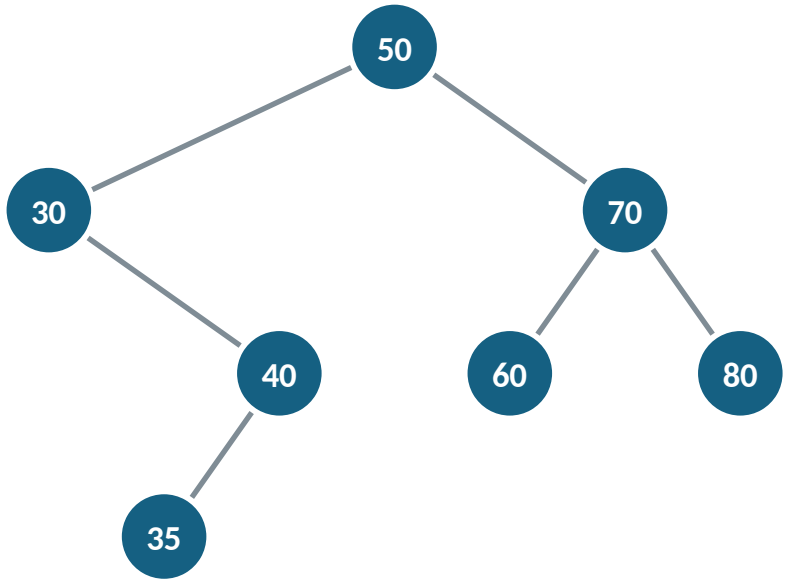
**?** **Think:** The successor of a node with two children has no left child. Why?

**Answer:** The successor is the leftmost node of the right subtree. A left child would be smaller, so it would be the leftmost node instead. So, deleting it is case 1 or 2.

# Delete Case 1: a Leaf




before



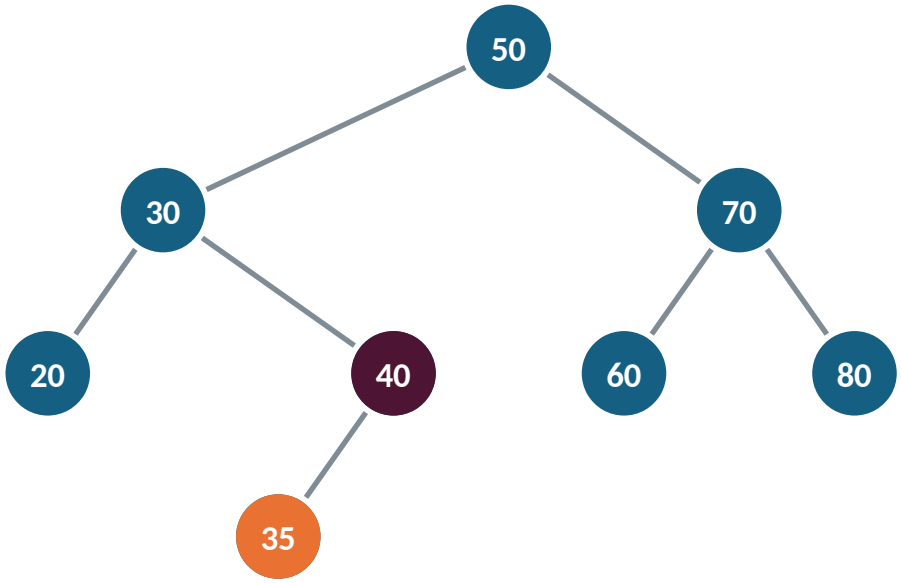

after

**delete 20: a leaf; just remove it**

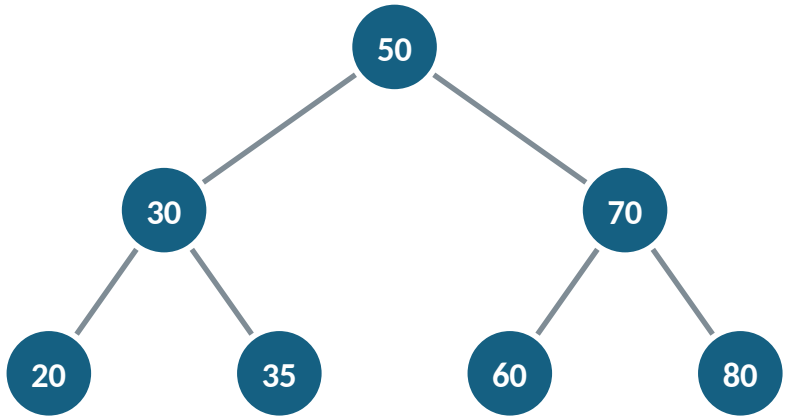
? **Think:** After deleting 20, what is the minimum key, and does the height of the tree change?

**Answer:** The minimum becomes 30, the new leftmost node. The height stays 3, because the path 50, 30, 40, 35 is untouched.

# Delete Case 2: One Child




before



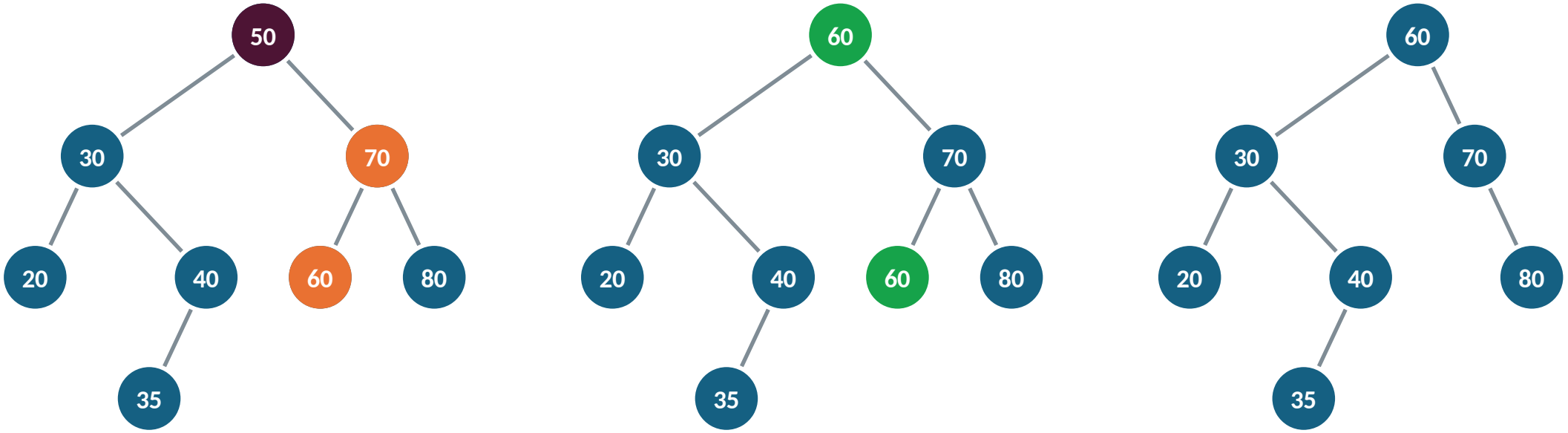

after

**delete 40: it has one child (35); the child takes its place**

**Think:** Now also delete 30 from this new tree. Which case is it, and which key takes its place?

**Answer:** Case 3: 30 now has two children (20 and 35). Its successor is 35, so 35 moves up into 30's place and the old 35 is removed.

# Delete Case 3: Two Children, Use the Successor




before copy the successor's key up after: the old 60 is removed

**delete 50: successor = go right (70), then left as far as possible (60)**

**Think:** Could we use the predecessor, the largest key in the left subtree, instead? Which key is it, and which case follows?

**Answer:** Yes: go left (30), then right as far as possible: 40. Copy 40 up; the old 40 has one child (35), so case 2 finishes the job.

# The delete Code, Line by Line

IIT Madras Zanzibar

```
 1  def delete(node, key):
 2      if node is None: return None  # Base case
 3      if key < node.value:
 4          node.left = delete(node.left, key)
 5      elif key > node.value:
 6          node.right = delete(node.right, key)
 7      else:
 8          if node.left is None: return node.right
 9          if node.right is None: return node.left
10          s = minimum(node.right)
11          node.value = s.value
12          node.right = delete(node.right, s.value)
13      return node
```

## Line by line

**Line 3–6** find the key: go left or right -Search Traversal

**Line 8–9** 0 or 1 child: return the other child

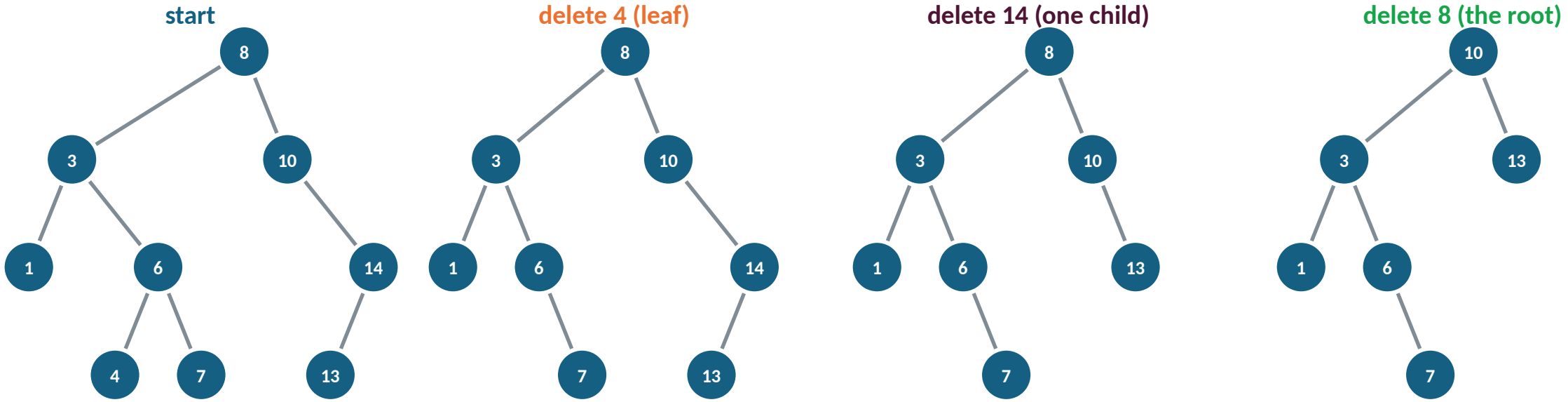
**Line 10–12** 2 children: copy the successor, delete it

**Line 13** return this node

? **Think:** Why does line 12 call delete on the right subtree instead of just unlinking the successor?

**Answer:** The successor may have a right child. The recursive call applies the same cases and lifts that child into the successor's place.

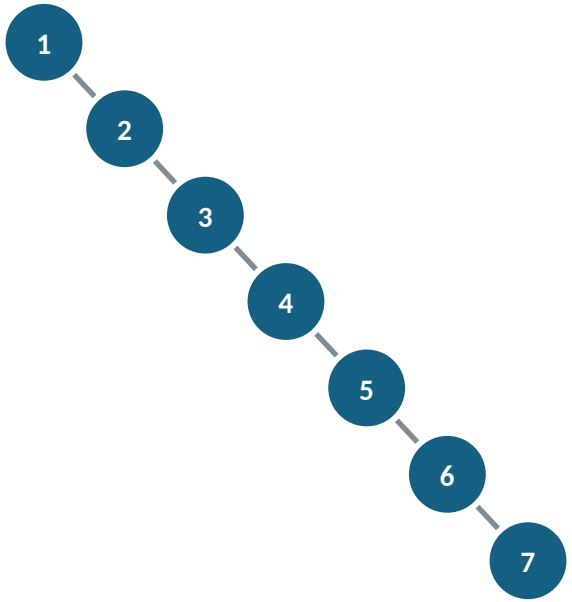
# A Worked Sequence: Delete 4, Then 14, Then 8



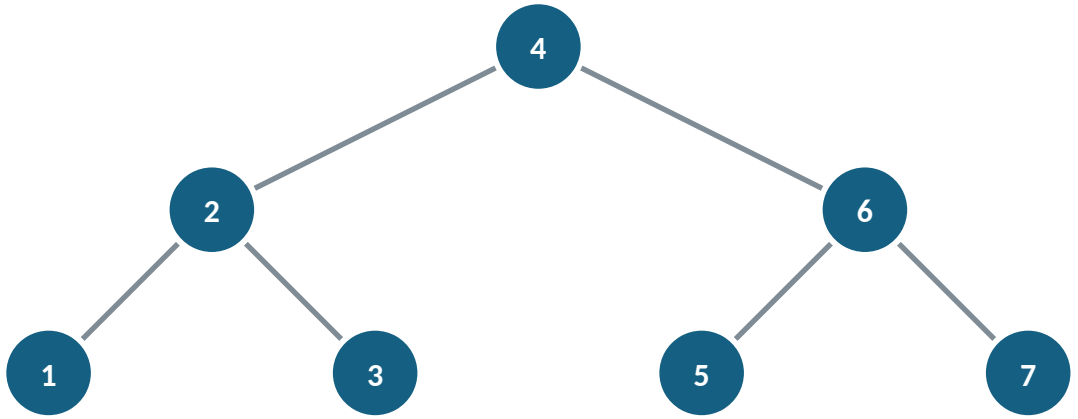

**Think:** Write the inorder traversal of the last tree. Which key is the root now, and why that one?

**Answer:** Inorder: 1 3 6 7 10 13, still sorted. The root is 10, the successor of 8: the smallest key of the old right subtree.

# Why Balance Matters




chain: height 6, search costs up to 7



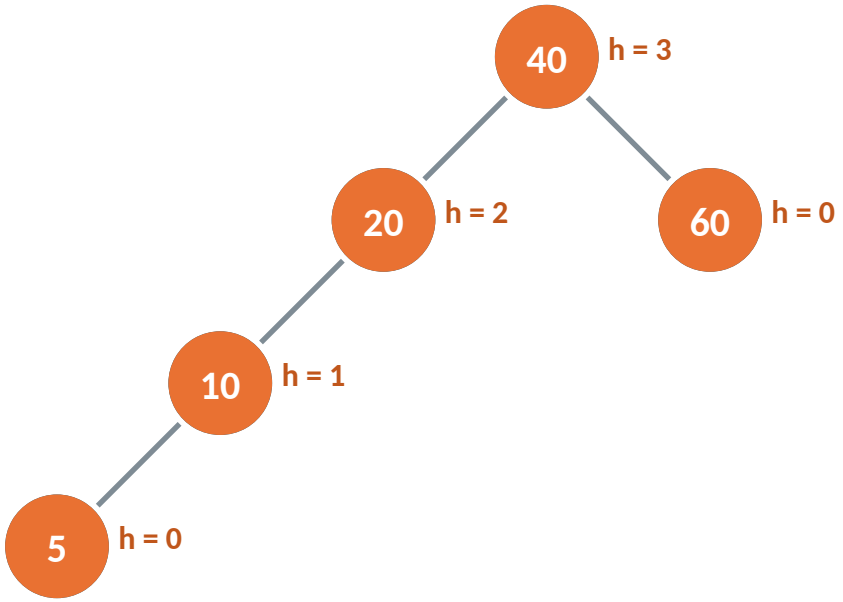

balanced: height 2, search at most 3

**1,000,000 keys:** up to 1,000,000 comparisons in a chain, about 20 in a balanced tree.

? **Think:** Insert the keys 1, 2, 3, …, n in order into a plain search tree. About how many comparisons in total, for n = 100,000? And for a balanced tree?

**Answer:** $1 + 2 + \ldots + n = n(n+1)/2$: about 5 billion for n = 100,000. A balanced tree needs about $n \log_2 n$, roughly 1.7 million.

# Balance Factor, Step 1: the Height of Every Node




**Height of a node:** edges on the longest path down to a leaf.
**empty tree:** −1 **leaf:** 0 **any other node:** 1 + the larger child height

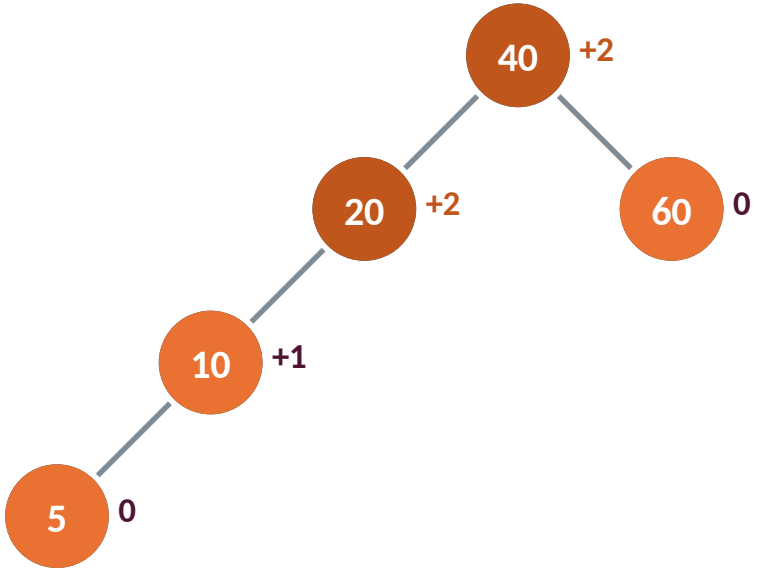
1. 5: a leaf, no children: height 0
2. 10: left (5) 0, right (empty) −1: 1 + max(0, −1) = 1
3. 20: left (10) 1, right (empty) −1: 1 + max(1, −1) = 2
4. 60: a leaf, no children: height 0
5. 40: left (20) 2, right (60) 0: 1 + max(2, 0) = 3

? **Think:** A node R has a left subtree of height 4 and an empty right side. What is the height of R?

**Answer:** 1 + max(4, −1) = 5. The empty side counts as −1, so it never wins.

# Balance Factor, Step 2: Left Minus Right

**Rule:** balance factor = height(left subtree) − height(right subtree). The AVL rule: every balance factor is −1, 0 or +1.

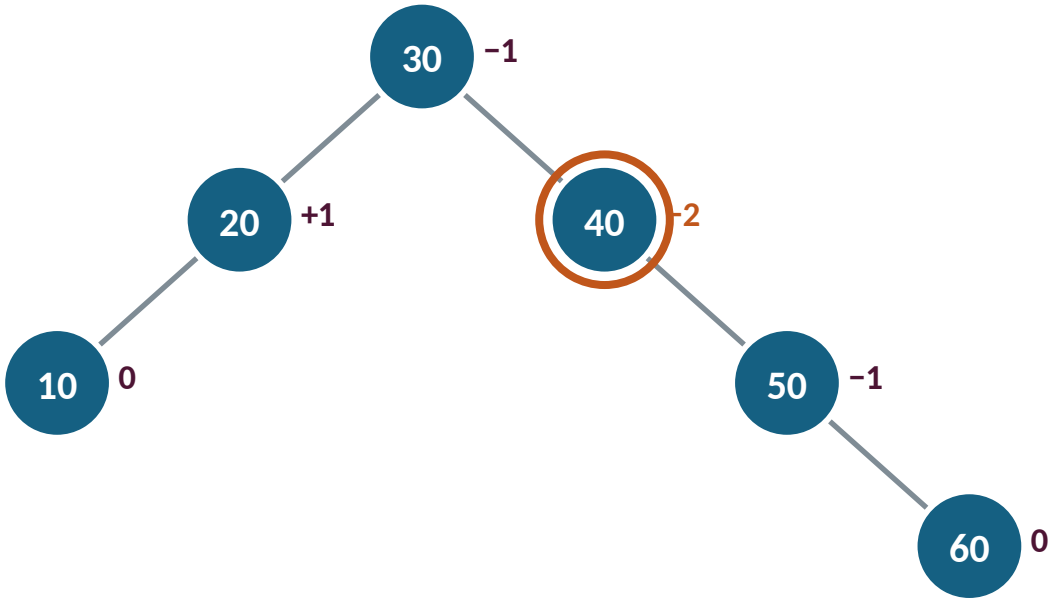



| node | height left | height right | left − right |
|---|---|---|---|
| **5** | −1 | −1 | −1 − (−1) = 0 |
| **10** | 0 | −1 | 0 − (−1) = +1 |
| **20** | 1 | −1 | 1 − (−1) = +2 violation |
| **60** | −1 | −1 | −1 − (−1) = 0 |
| **40** | 2 | 0 | 2 − 0 = +2 violation |

? **Think:** Which nodes violate the AVL rule, and which would you repair first?

**Answer:** Nodes 20 and 40, both +2. Repair the lowest violating node first (20); the repair also lowers the left side of 40.

# Your Turn: Compute the Balance Factors



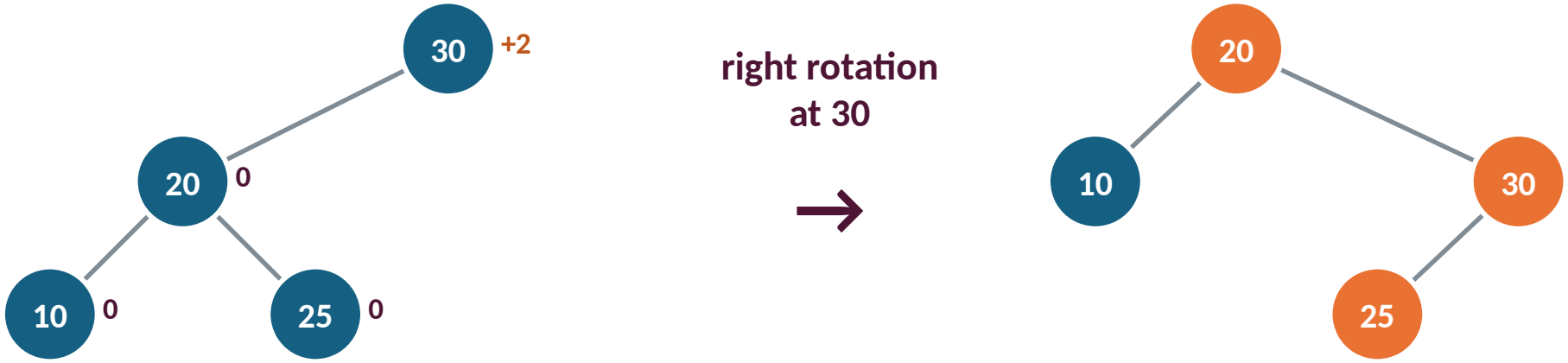

**Recipe**

1. height of every node, bottom up
2. balance factor = height(left) − height(right)
3. an empty side counts as −1
4. a violation is +2 or −2

? **Think:** Compute the balance factor of every node, bottom up. Which node violates the AVL rule, and what is its shape?

**Answer:** 10: 0, 60: 0, 20: +1, 50: −1, 40: −2, 30: −1. Only 40 violates: the chain 40, 50, 60 leans right twice (right-right).

# Rotation: What It Does, Step by Step




before: balance factor of 30 is +2

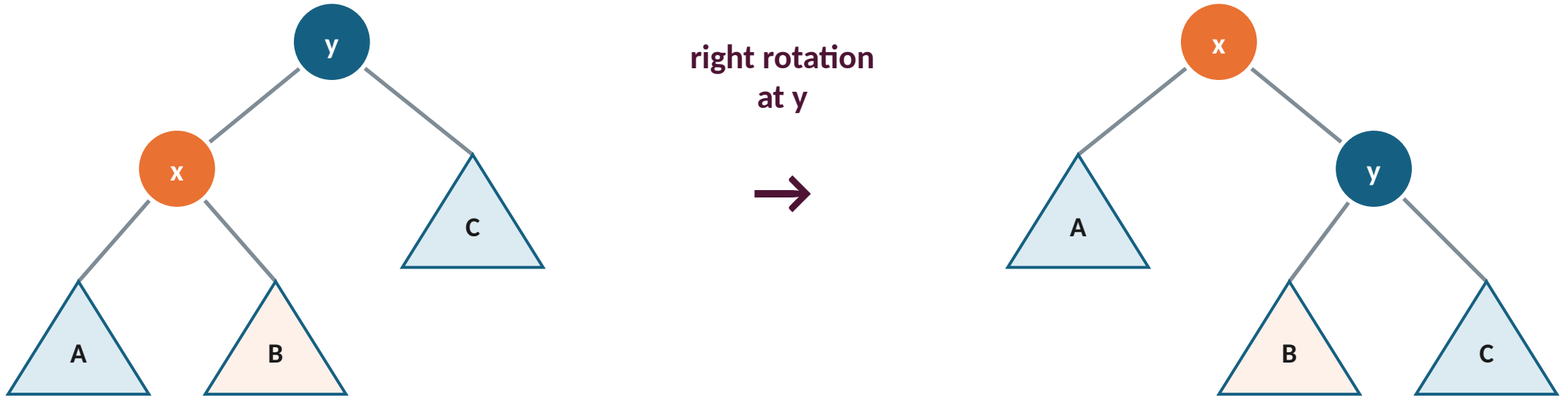
after: 20 on top; inorder still 10 20 25 30

1. 20, the left child of 30, moves UP into 30's place. It keeps its left child 10.
2. 30 moves DOWN, to the right of 20.
3. 25 was the right child of 20. It changes parent: it becomes the LEFT child of 30, the slot that 20 left behind.

? **Think:** After the rotation, compute the balance factors of 20, 30 and 10. Is the AVL rule satisfied again?

**Answer:** 20: left 0 − right 1 = −1. 30: left 0 − right (−1) = +1. 10: 0. All within −1 to +1, so the rule is restored.

# Rotation: the General Picture




1. x moves UP and becomes the top. It keeps its left subtree A.
2. y moves DOWN, to the right of x. It keeps its right subtree C.
3. B changes parent: from x (its old right subtree) to y (its new left subtree). Three links changed in total.

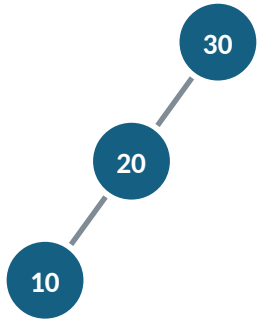
? **Think:** Why can B move to y's left without breaking the search rule?

**Answer:** The order is A < x < B < y < C. B's keys are larger than x and smaller than y: below y as its left subtree, and above x because y hangs on x's right.

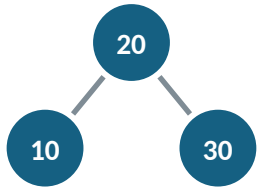
# The Four Cases: Single and Double Rotations

IIT Madras Zanzibar

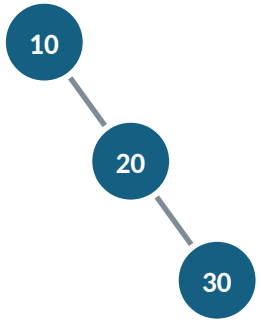
**left-left**



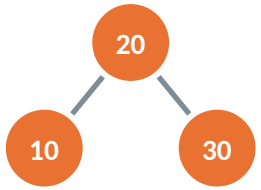

↓ right rotation at 30



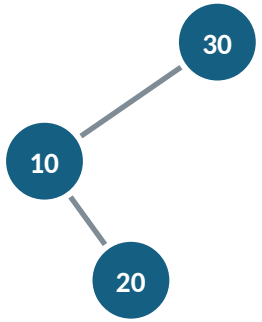

**right-right**



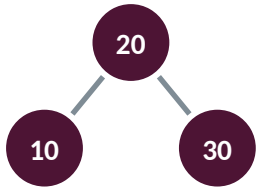

↓ left rotation at 10



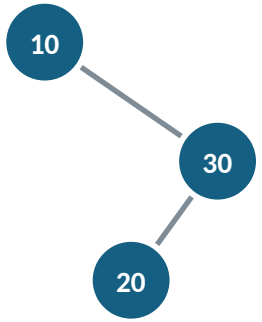

**left-right**



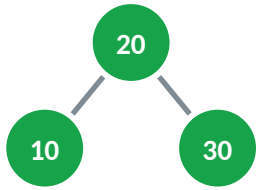

↓ left at 10, then right at 30




**right-left**




↓ right at 30, then left at 10




? **Think:** Insert 50, then 20, then 30 into an empty AVL tree. Which case occurs, and which rotations fix it?

**Answer:** 30 is the right child of 20, the left child of 50: left-right. Rotate left at 20, then right at 50. Result: 30 on top, 20 and 50 below.

# Why an AVL Tree Stays Short

IIT Madras Zanzibar

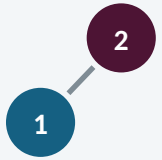
**Height h = edges on the longest path from the root down to a leaf. N(h) = the FEWEST nodes any AVL tree of height h can have.**

**height 0**



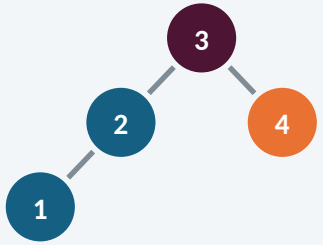

a root alone
N(0) = 1

**height 1**



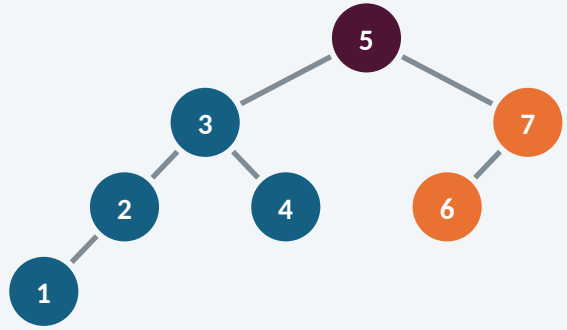

root + one child
N(1) = 2

**height 2**




root + N(1) + N(0)
= 1 + 2 + 1 = 4

**height 3**




root + N(2) + N(1)
= 1 + 4 + 2 = 7

**Why h − 1 and h − 2:** to reach height h, one subtree must have height h − 1; AVL lets the other be one shorter, h − 2; plus 1 for the root. N(h) = N(h−1) + N(h−2) + 1 gives 1, 2, 4, 7, 12, 20, 33: Fibonacci-like growth, so height ≤ about 1.44 $\log_2 n$ (1,000,000 keys: at most 27).

? **Think:** What is the fewest nodes an AVL tree of height 7 can have?

**Answer:** N(7) = N(6) + N(5) + 1 = 33 + 20 + 1 = 54.

# Which Structure for Which Job?

IIT Madras Zanzibar

| structure | search | insert | delete | sorted output |
|---|---|---|---|---|
| **unsorted list** | O(n) | O(1) | O(n) | O(n log n) |
| **sorted array** | O(log n) | O(n) | O(n) | O(n) |
| **plain search tree** | O(h): up to n | O(h) | O(h) | O(n) |
| **AVL tree** | O(log n) | O(log n) | O(log n) | O(n) |
| **red-black tree** | O(log n) | O(log n) | O(log n) | O(n) |
| **hash table (later)** | O(1) average | O(1) average | O(1) average | not ordered |

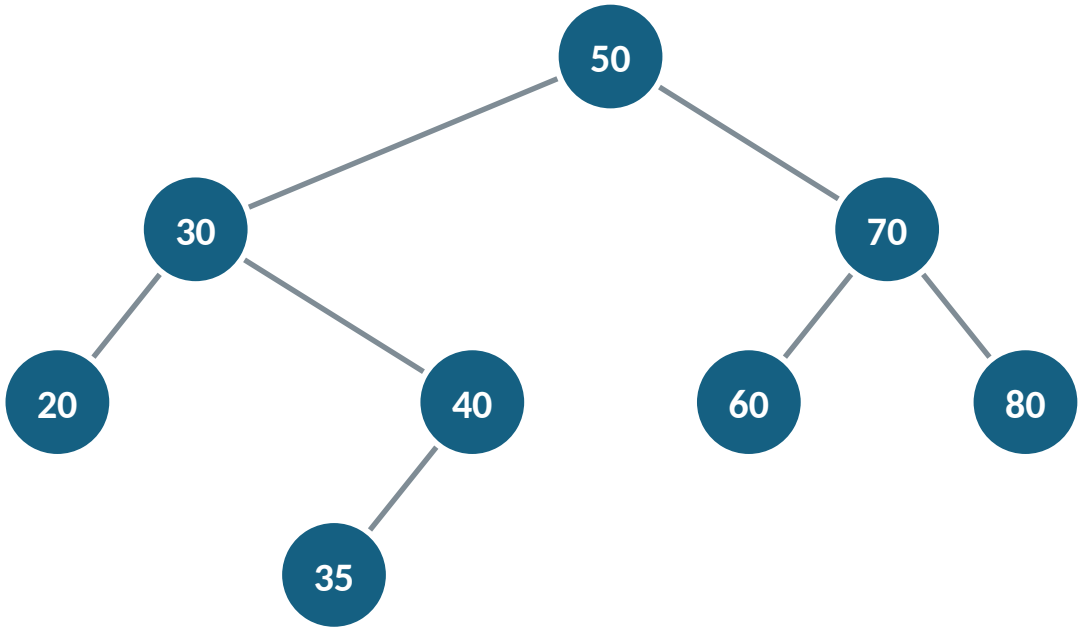
? **Think:** You need the smallest key larger than 40 in a changing set. Hash table or balanced tree, and why?

**Answer:** A balanced tree: it is ordered, so the successor of 40 is a short walk of about log n steps. A hash table has no order.

# In-Class Exercise (6 minutes, in pairs)

IIT Madras Zanzibar

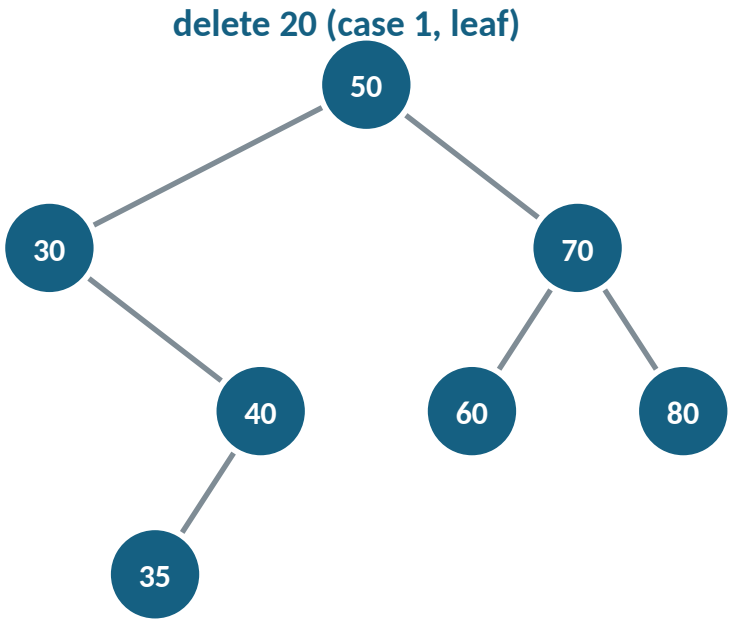
**IN-CLASS TASK: delete, balance, rotate**

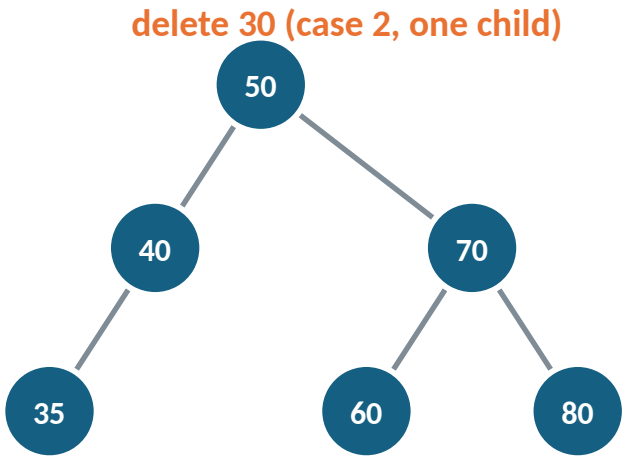


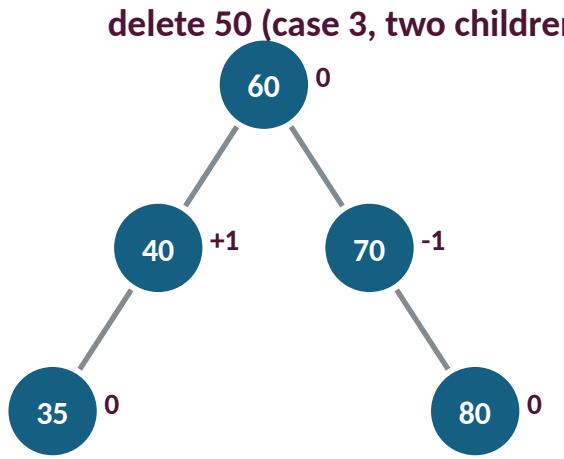

### Four tasks (paper)

1. Delete 20, then 30, then 50, in this order. Draw the tree after each delete and name the case.
2. Write the inorder traversal of the final tree.
3. Is the final tree an AVL tree? Give the balance factor of every node.
4. Insert 5, then 3, then 4 into an empty AVL tree. Which case occurs and which rotations repair it?

# Solution A: Three Deletes and the Balance Factors



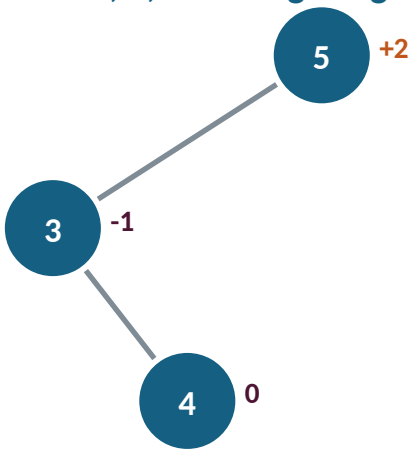




**Final:** inorder 35 40 60 70 80;  balance factors 0, +1, 0, −1, 0: the tree is balanced.

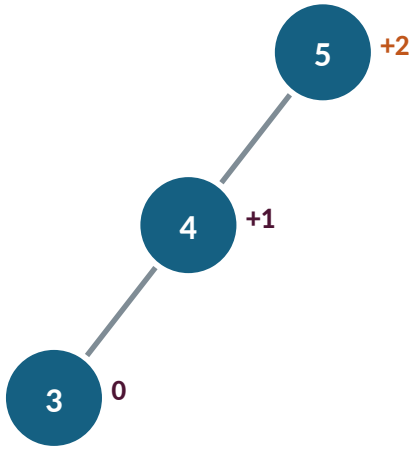
# Solution B: Inserting 5, 3, 4 Needs a Double Rotation

IIT Madras Zanzibar

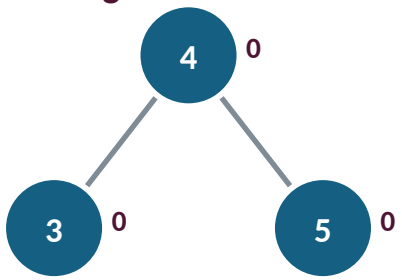
**insert 5, 3, 4: left-right zigzag**




**left rotation at 3**




**right rotation at 5**




**Check:** the inorder is 3 4 5 at every stage; the double rotation ends with all balance factors 0.

# Summary and Exam Checkpoints

IIT Madras Zanzibar

### Delete

leaf: remove; one child: promote it; two children: copy the successor, delete it

### Balance

bf = height(left) − height(right); AVL keeps it in −1, 0, 1

### Rotations

constant-time link changes that keep the inorder; four cases, two of them double

### Guarantee

AVL height at most about 1.44 $\log_2 n$: everything is O(log n)

## Next

**Mon 23 Nov:** heaps and priority queues

**Later:** hash tables: lookup without order

# References and Further Reading


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

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2022). Introduction to Algorithms, 4th ed. MIT Press. Chapter 12; Problem 13-3 (AVL trees).

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