CS 2100: Data Structures & Algorithms 1

Trees

~ AVL Trees ~

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Announcements / Reminders

- Quiz Retakes – Modules 4-6
  - Monday, March 28, 2022
    - Choose 1
    - No new Quiz!
    - Reminder: quiz retakes are 100% optional; if you're happy with your quiz scores, no need to retake anything

[See schedule on our webpage]

AVL Trees

Animation Tool

- A good **AVL tree animation** too is HERE
  - [https://visualgo.net/en/bst?mode=AVL]

- You’re welcome to play around with this visual animation of AVL trees as you review the material for this topic

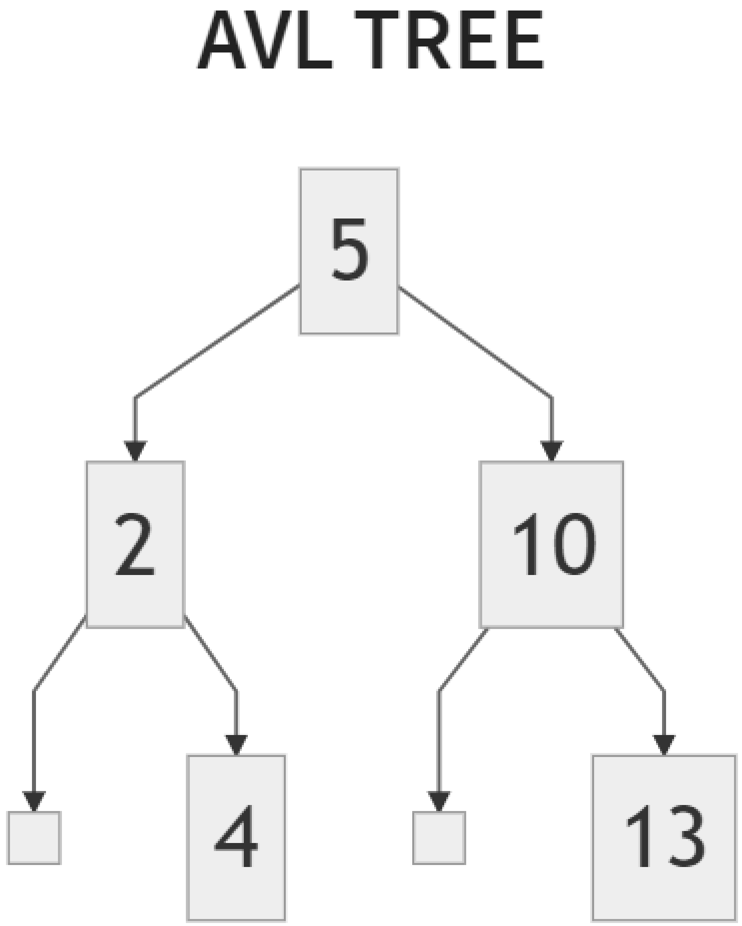
AVL Trees

- Motivation: to ***guarantee*** **Θ(log n)** running time on **find**, **insert**, and **remove**
- Idea: Keep tree **balanced** after each operation
- Solution: AVL trees
  - Named after the inventors, *Adelson-Velskii and Landis*

AVL Tree Structure Property

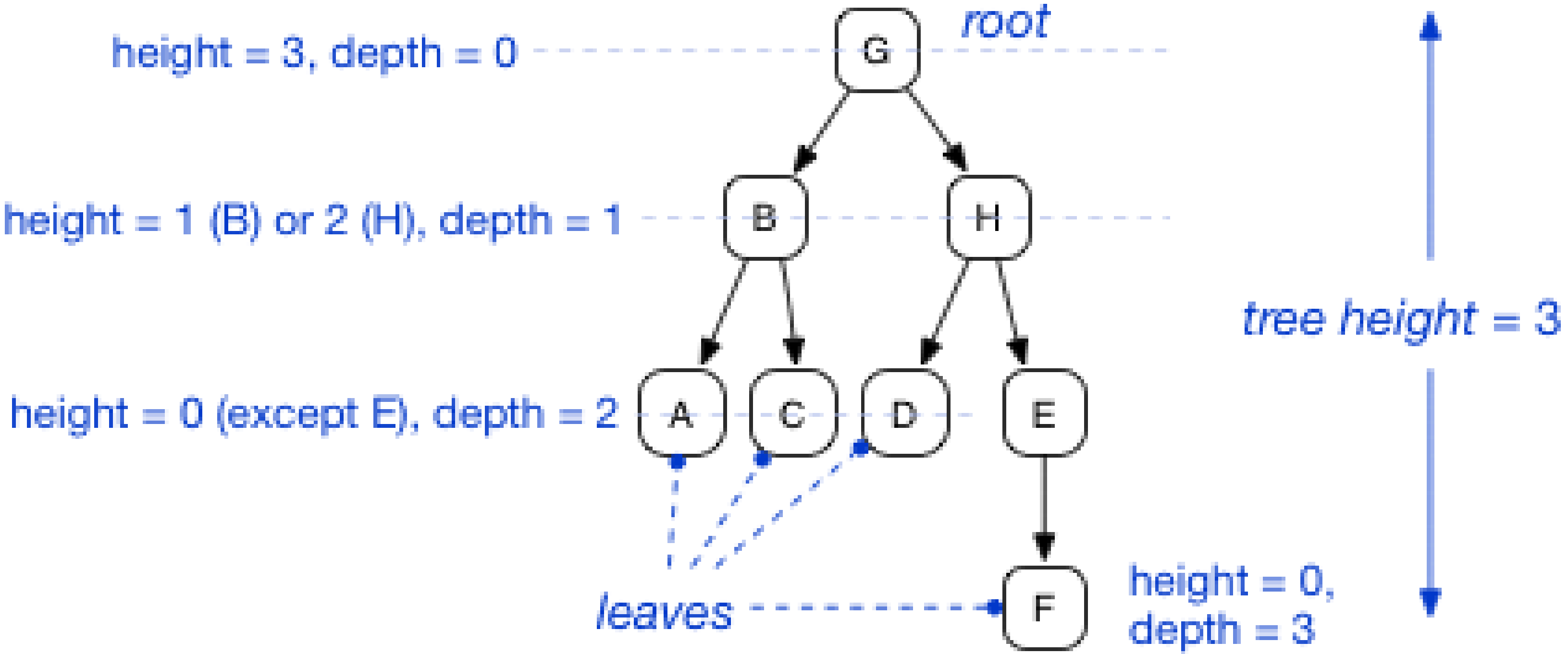
- An AVL Tree is a self-balancing Binary Search Tree
  - Where the difference between ***heights*** of left and right sub-trees cannot be more than 1 for all nodes

- Put another way…
  - For every node in the tree, the ***height*** of the left and right sub-trees differs at most by 1



Reminder About Height (Binary Tree)

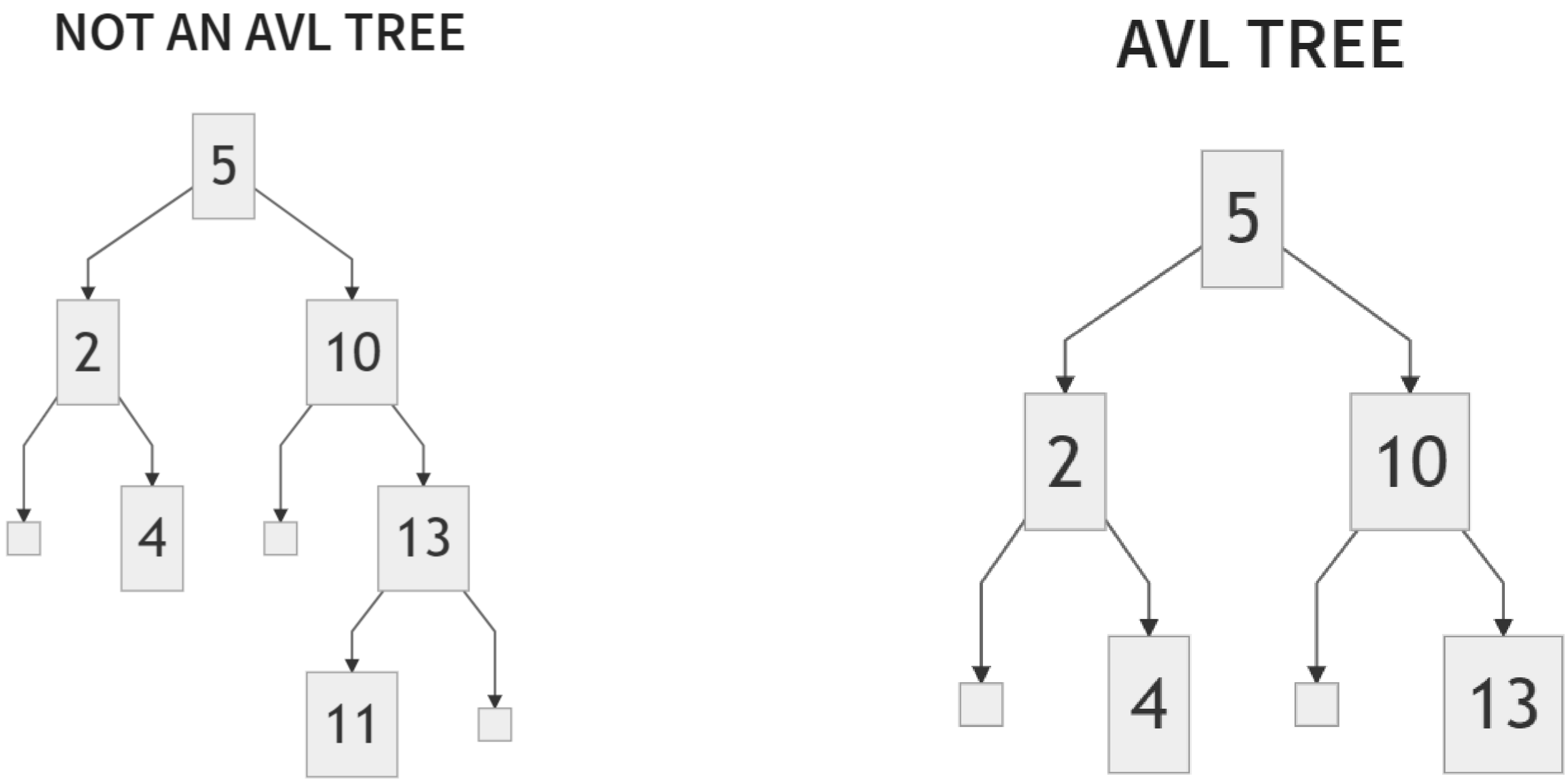
- **Height Definition**
  - **HEIGHT** of a **node**: is the *longest* **path** ***(# edges)*** from that node to a **leaf**
    - Thus, all **leaves** have a height of **zero** (0); Height of tree = height of the **root**



AVL Balance Factor

- Each node of a **BST** holds:
  - The data
  - Left and right child references

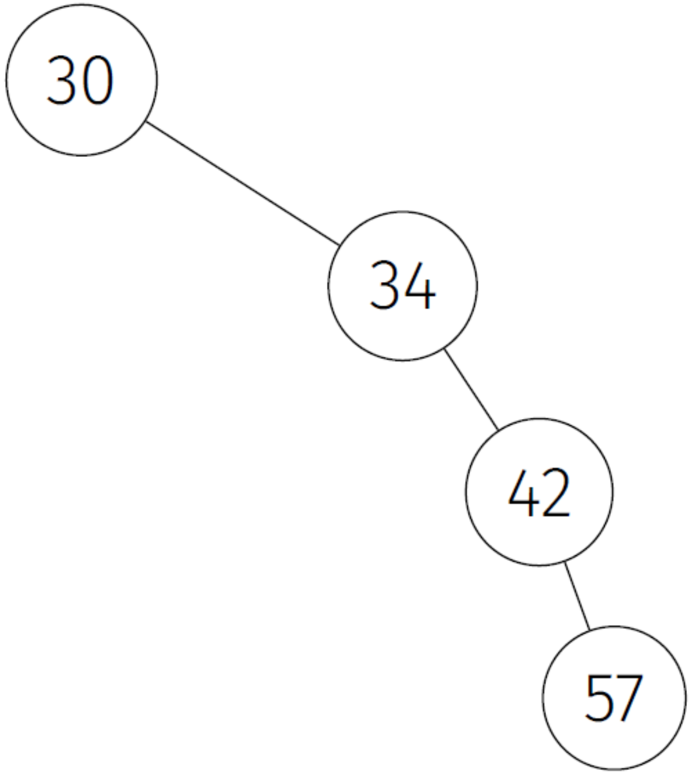
- An **AVL tree** also holds a **balance factor**
  - Balance factor = The height of the *right* sub-tree minus the height of the *left* sub-tree
  - Can be computed on the fly, as well, but that's VERY slow

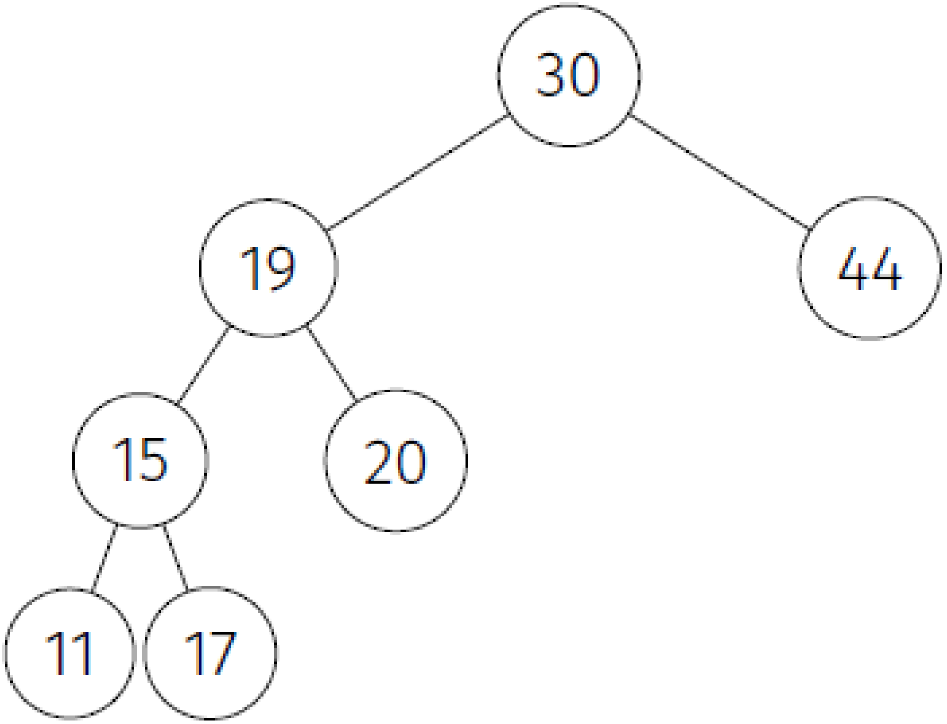


NOT AN AVL TREE

AVL TREE

Structure and Balanced Nature of **AVL Tree** (Example of an AVL Tree and a Tree that is not.)





Reminder: Both of these trees are examples of Binary Search Trees! (NOT AVL TREES)

AVL Tree Balance

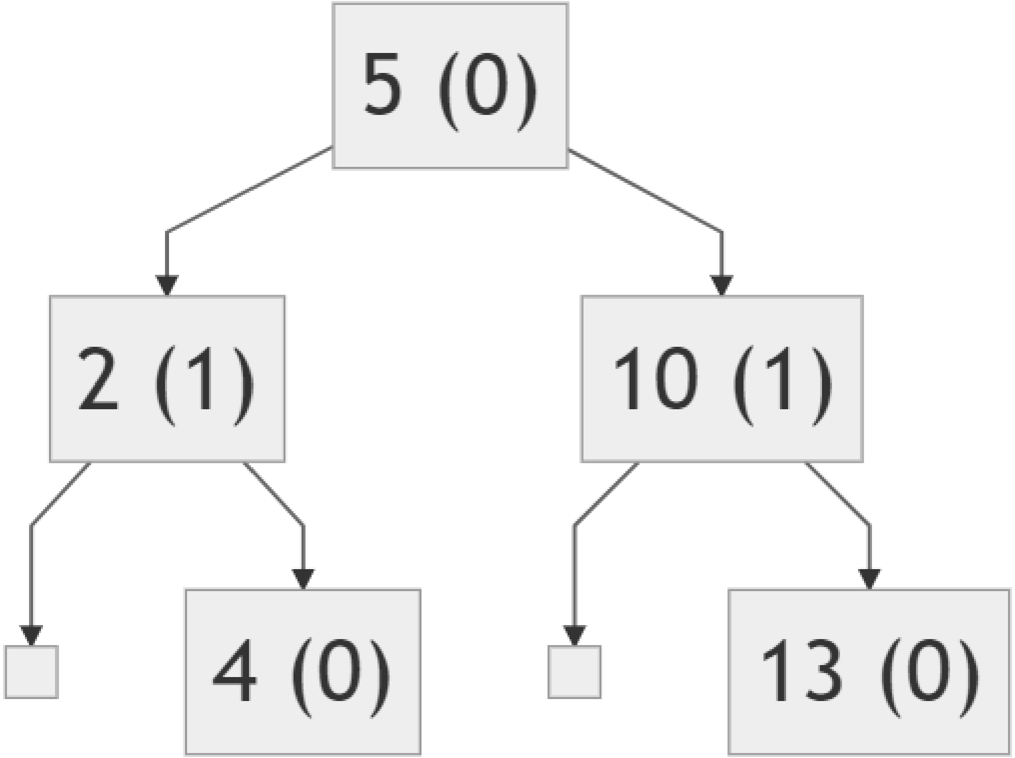
- “Balanced” trees
  - Balance Factor = 0 balanced
  - Balance Factor = 1 the **right** sub-tree is one longer than the left sub-tree
  - Balance Factor = -1 the **left** sub-tree is one longer than the right sub-tree

- “Unbalanced” trees
  - Balance Factor of 2 or -2
  - We will fix the tree once we discover an unbalanced tree (indicated by above Balance Factor)

- ***Question***: Will a node ever have a Balance Factor of **3** or **-3** (or more)?

AVL Tree With Balance Factors

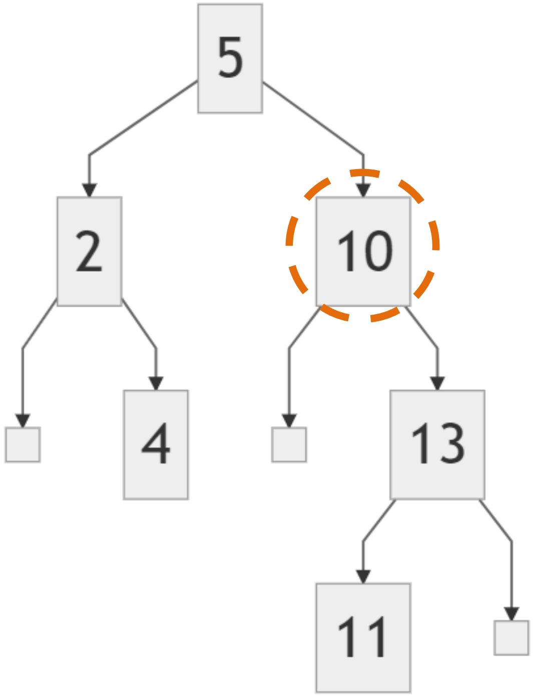
- Numbers in parenthesis represent the **Balance Factor** for each node in this AVL Tree:



Explanation Why This Tree is NOT an AVL Tree

- The **Balance Factor** for node “10” is **+2**
  - Left sub-tree height: 0
  - Right sub-tree height: 2
  - Difference > **1** !

NOT AN AVL TREE



AVL Trees: Find and Insert

- Find method: same as BST find
- Insert method: same as BST insert, except might need to "fix" the AVL tree after the insert (via rotations)

- Runtime analysis:
    - $\Theta(d)$, where ***d*** is the depth of the node being found/inserted

- ***Question:*** What is the **maximum height** of an n-node **AVL tree**?
- ***Question:*** What is the **maximum height** of an n-node **BST tree**?

AVL Trees: Find and Insert

- Find method: same as BST find
- Insert method: same as BST insert, except might need to “fix” the AVL tree after the insert (via rotations)

- Runtime analysis:
  - Θ(d), where ***d*** is the depth of the node being found/inserted

- ***Question:*** What is the **maximum height** of an n-node **AVL tree**? `log(n)`
- ***Question:*** What is the **maximum height** of an n-node **BST tree**? `(n-1)` or `log(n)`

AVL Tree Operations

- Perform the operation (insert, delete)
- Move back up to the root, **updating** the balance factors
  - *Why only those nodes?*
  - Because those are the only ones who have had their subtrees altered
    - Traversed one path to add or delete node, so check nodes on that path alone
    - No need to check left and right
    - Fix at the ***lowest*** imbalance
      - Fixing this will fix everything above it, too since they share the same sub-tree
- **Do tree rotations** where the balance factors are **2 or -2**

Quick Clarification…

- When performing insert or delete, we only insert or delete **ONE node** at a time

How Many Times To "Fix" The Tree?

- Any **single insert** will only modify the balance factor by **one**
  - So, we fix the lowest off-balance nodes
  - *Then everything above it is then balanced*
- This means that we will have to only look at the bottom two unbalanced nodes

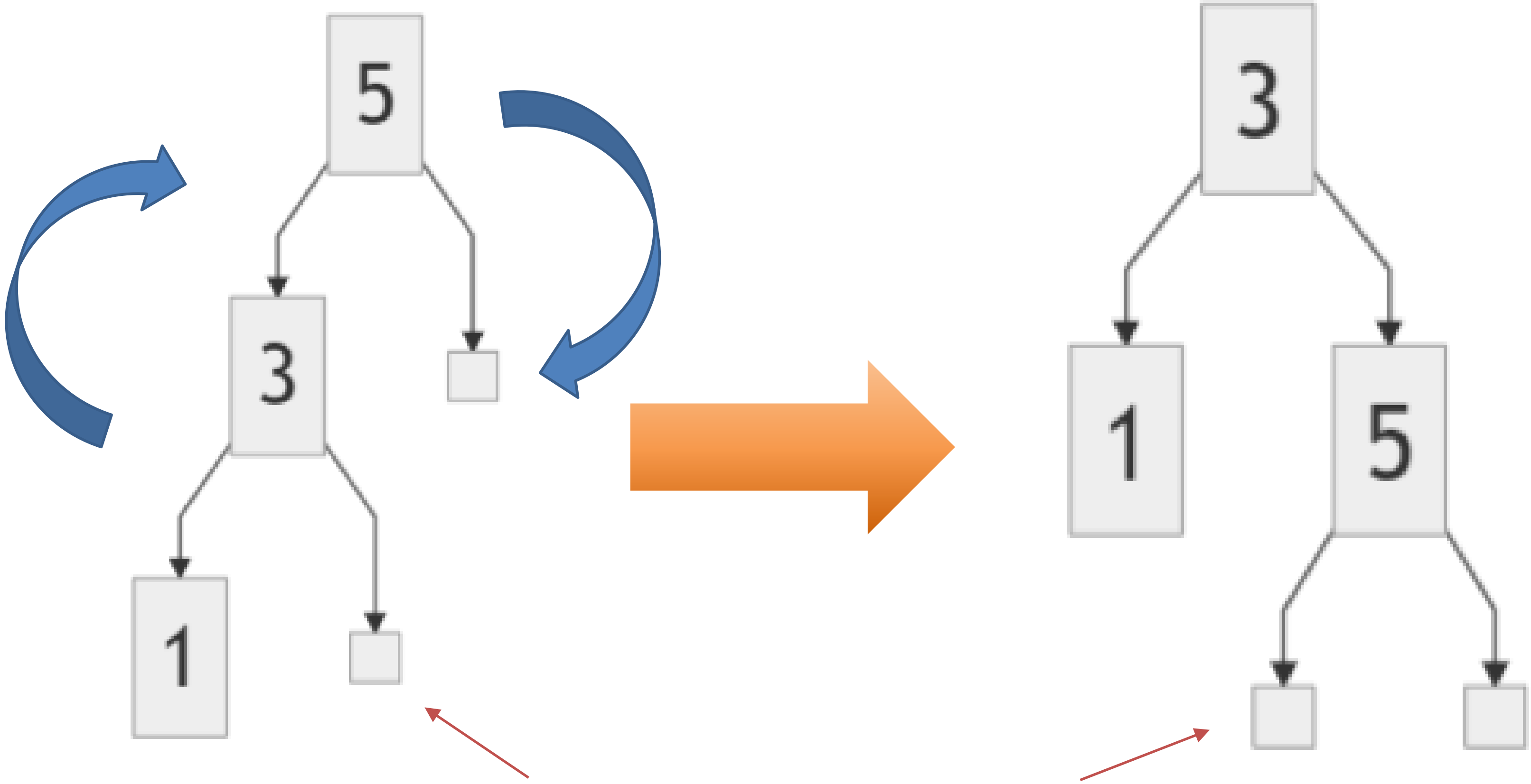
AVL Insert

- Let *x* be the *deepest* node where imbalance occurs
- Four cases where the insert happened:
  1. In the left subtree of the left child of x
  2. In the right subtree of the left child of x
  3. In the left subtree of the right child of x
  4. In the right subtree of the right child of x
- **Cases 1 & 4**: perform a single rotation
- **Cases 2 & 3**: perform a double rotation

Discussion of Cases

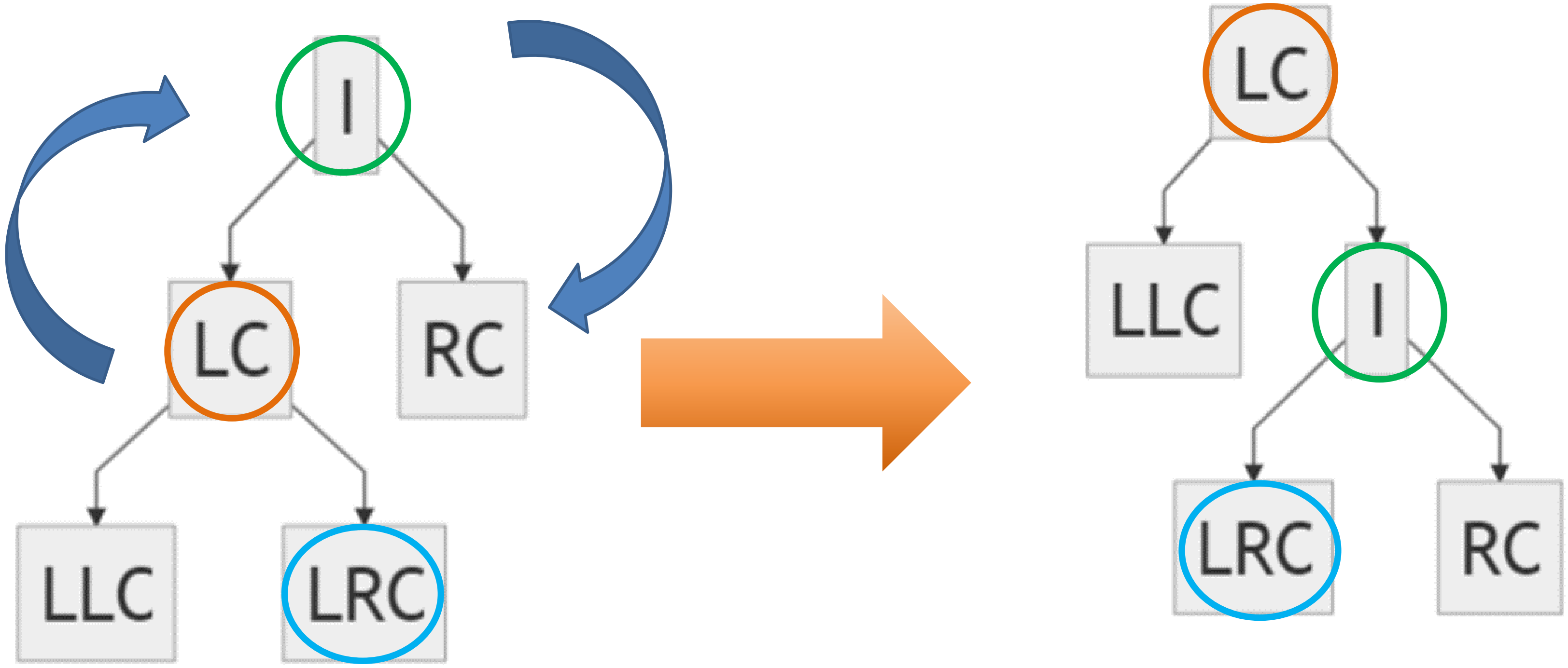
Rotation Cases 1 through 4 for an AVL Tree

5 is imbalanced, so we need to rotate on 5



Notice that the old right subtree of 3 becomes new left subtree of 5

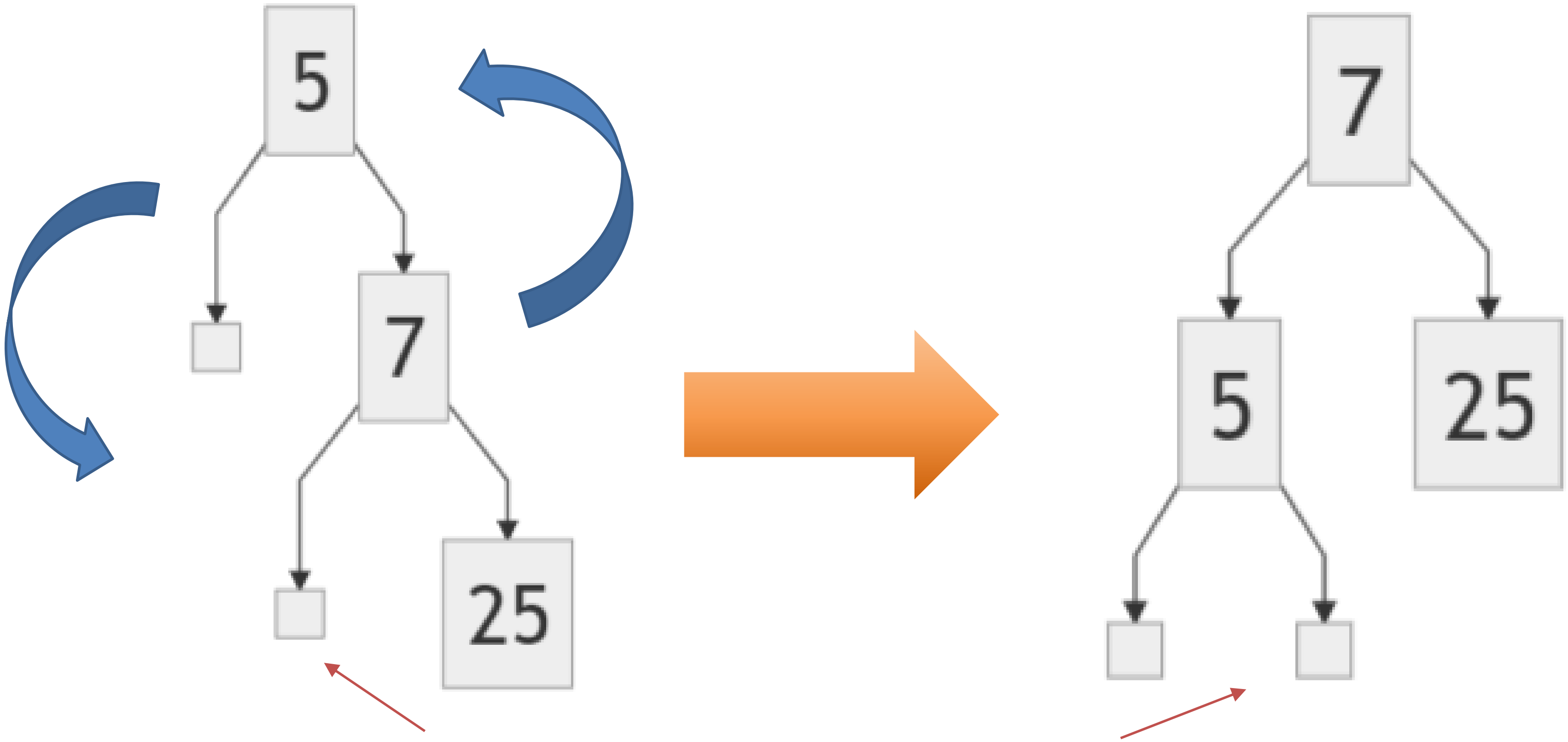
AVL Single Right Rotation



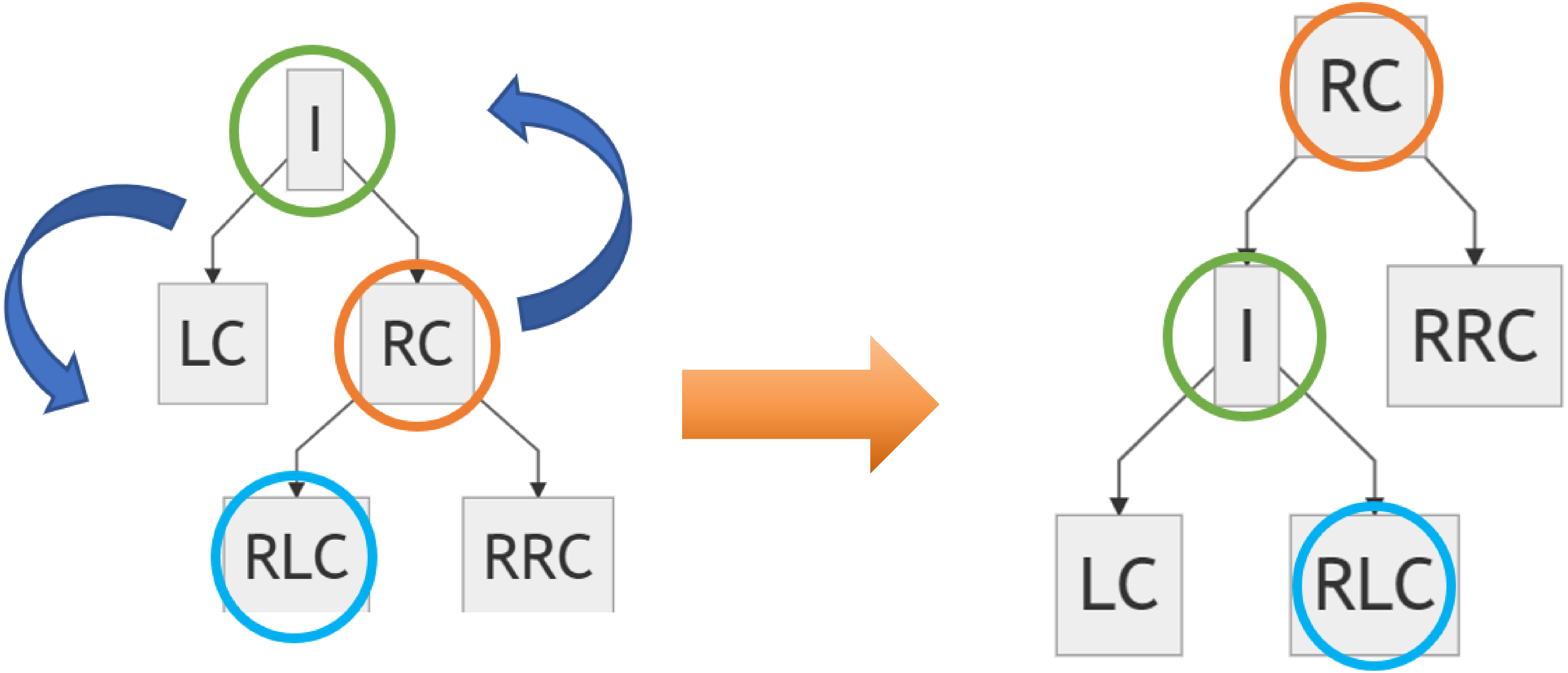
AVL Single Right Rotation: GENERAL CASE

AVL Single Left Rotation

5 is imbalanced, so we need to rotate on 5



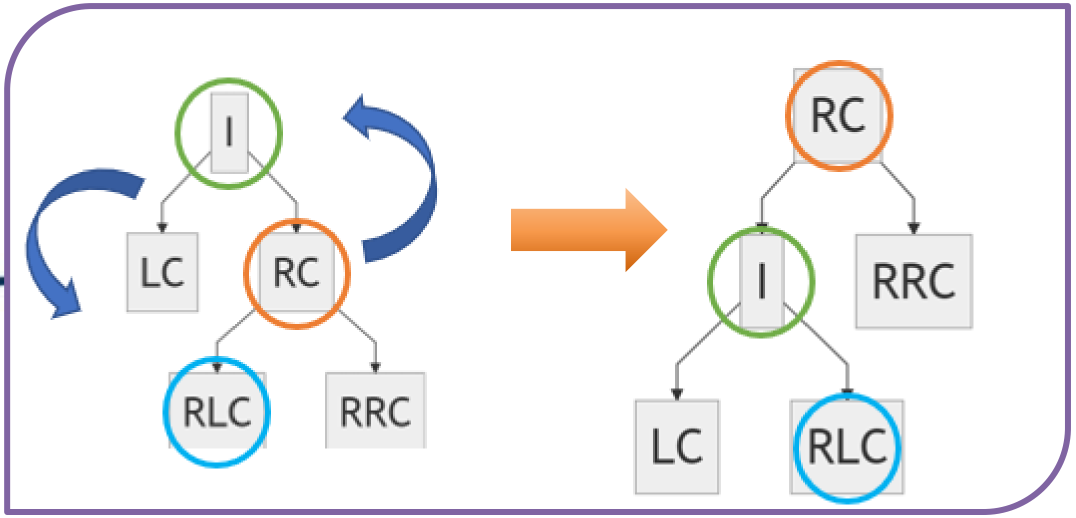
Notice that the old left subtree of 7 becomes new right subtree of 5



AVL Single Left Rotation: GENERAL CASE

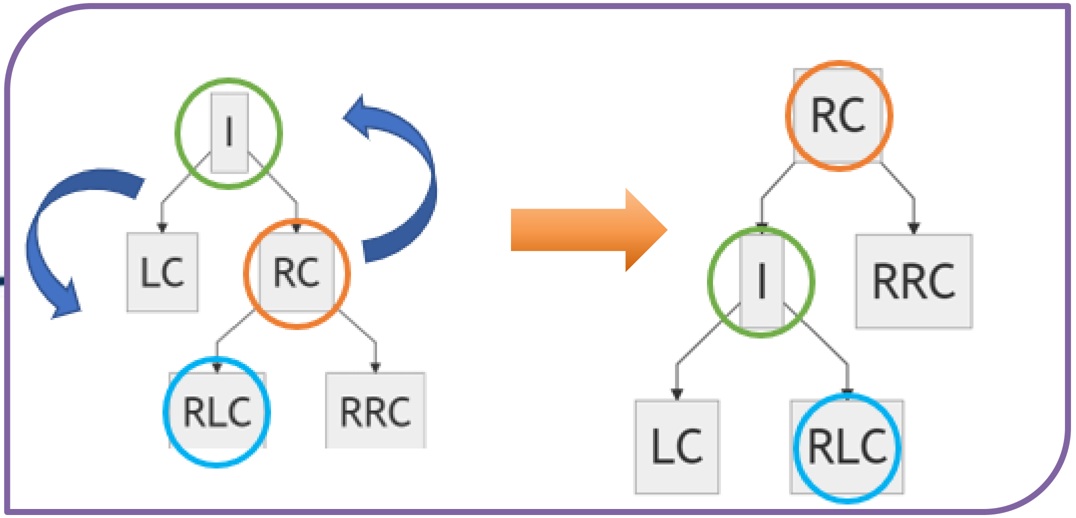
AVL Single Left Rotation:

ROTATELEFT METHOD



1. In a **new** node ("rNew") save **current Node's** (GREEN-I) **right node** (`curNode.right`) (**ORANGE-RC**)
2. In a **new** node ("**RL**") save the **left** subtree of the **current Node's** **right** node (`curNode.right.left`) (**BLUE-RLC**)
3. With the rotation, the **right** node ("rNew") (**ORANGE-RC**) will have the **current Node** (GREEN) as a **left** child
4. With the rotation, the **current Node's** (GREEN-I) has "**RL**" node as its **right** child

AVL Single Left Rotation: ROTATELEFT METHOD

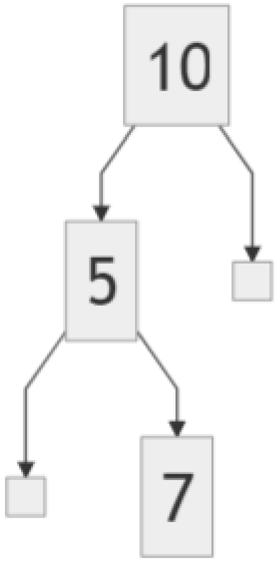


Now that we're done **update height**:

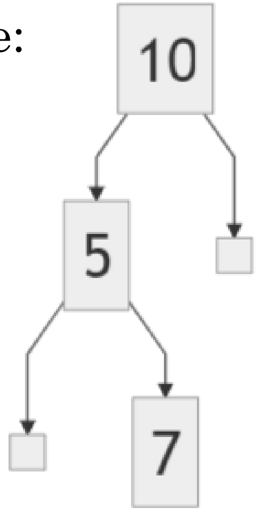
5. Set the **current Node's** (GREEN-I) **height** as the max of the height of the **left** and **right** child (plus 1 for itself)
6. Set the **right** node ("**rNew**") (**ORANGE-RC**) **height** as the max of the height of the **left** and **right** child (plus 1 for itself)
7. Finally, return the **right** node ("**rNew**") (**ORANGE-RC**) as it is the root of the sub-tree

Cases 2 and 3

- Attempt a single rotation on the following:

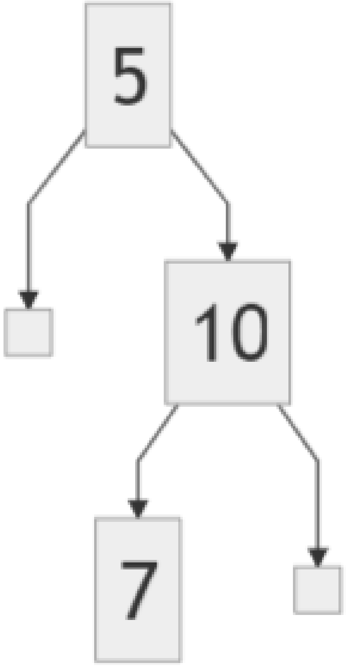


Before:



Cases 2 and 3

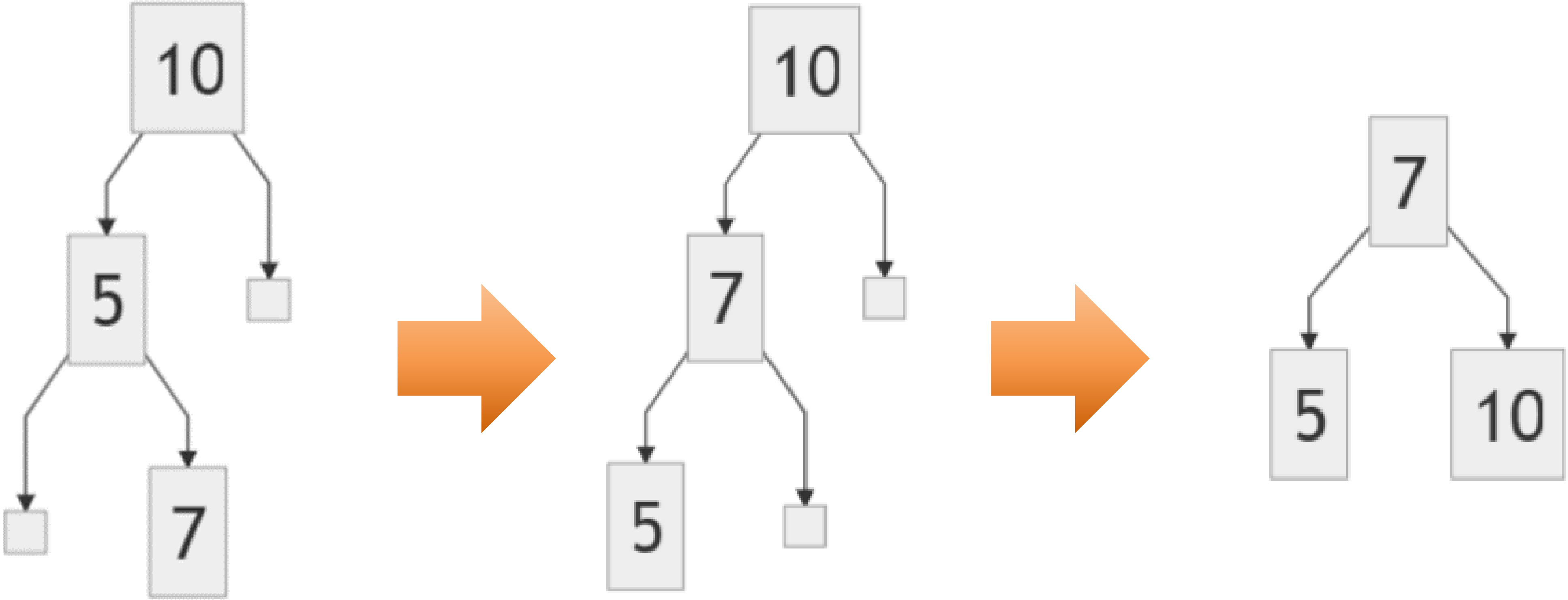
- Attempt a single rotation on the following:
  - What happened?? ***Still imbalanced!!***



DOUBLE Rotation

- A **double rotation** is used to get around this problem.

- **Double right rotation**:
  - rotate left on the **left child** of imbalanced node
  - then rotate right on the **imbalanced node**.



Double Right Rotation Example

Double right rotate on 10

Rotate left on **5**, then right on **10**

Double Left Rotation

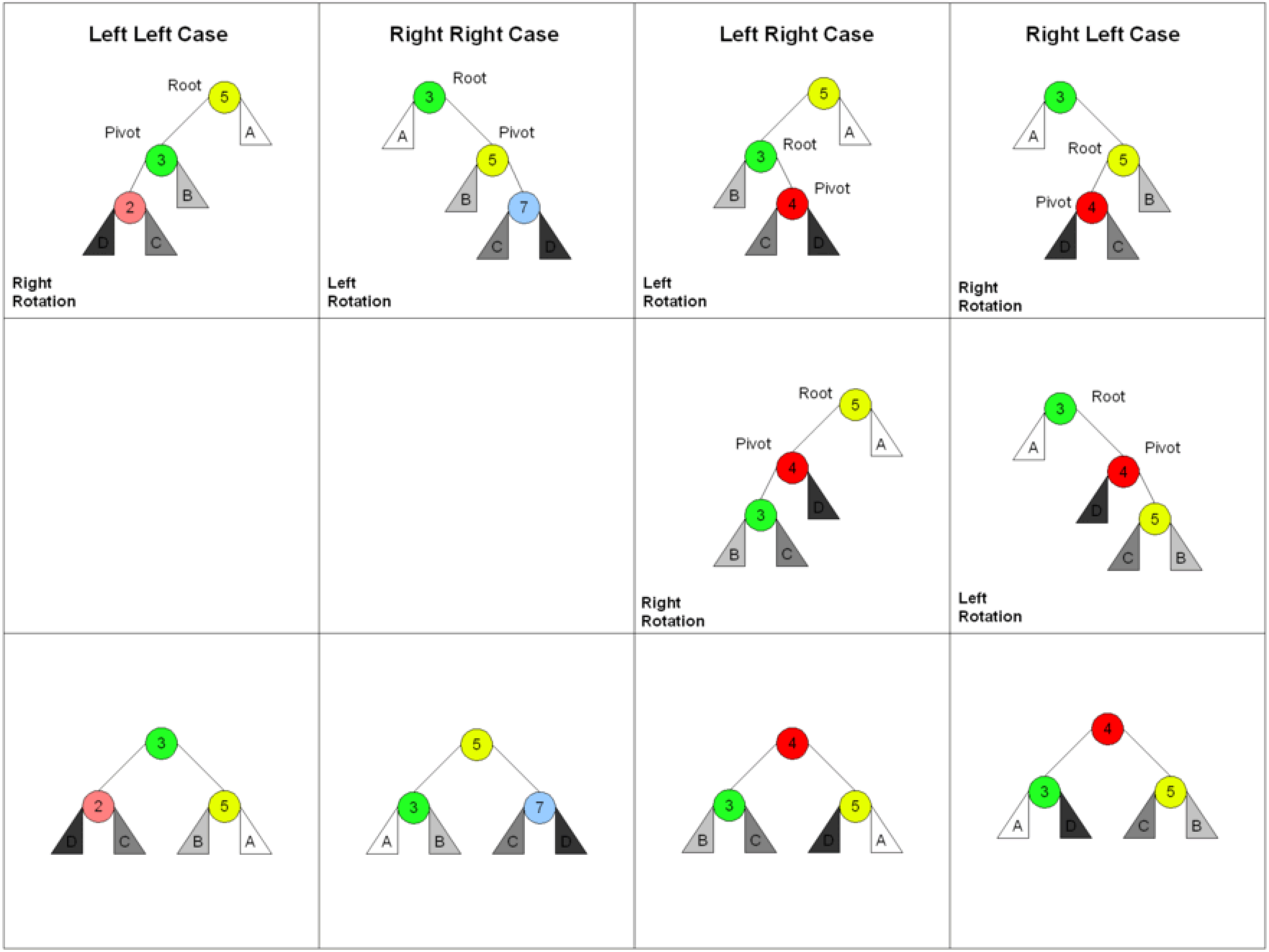
- Analogous to the other one

- **Double left rotation**:
  - rotate right on the **right child** of imbalanced node
  - then rotate left on the **imbalanced node**.

AVL Insert (one more time)

- Let *x* be the *deepest* node where imbalance occurs
- Four cases where the insert happened:
  1. In the left subtree of the left child of x
  2. In the right subtree of the left child of x
  3. In the left subtree of the right child of x
  4. In the right subtree of the right child of x
- **Cases 1 & 4**: perform a single rotation
- **Cases 2 & 3**: perform a double rotation

ALL THE TREE ROTATIONS



AVL Tree: Runtime Analysis

- **Find**: Θ(log n) time: **height** of tree is always **Θ(log n)**
- **Insert**: Θ(log n) time: find() takes Θ(log n), *then **may** have to visit every node on the path back up to root to perform up to **2 single rotations***
- **Remove**: Θ(log n): left as an exercise
- **Print**: Θ(n): no matter the data structure, it will *still take n steps* to print n elements