AVL Tree

Sur la base des cours de M. Gaetano &
CENG 213, Data Structures, Yusuf Sahillioğlu

an AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing binary search tree

MBF

Balanced BST

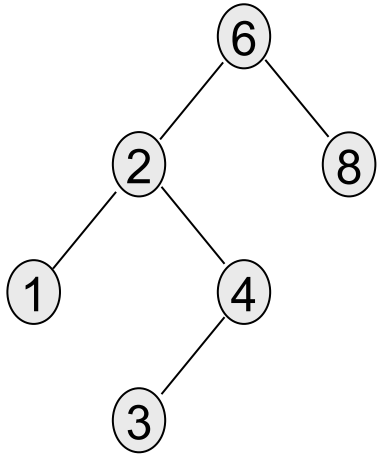
- Observation
  - BST: the shallower the better!
  - For a BST with n nodes inserted in arbitrary order
    - Average height is O(log n)
    - Worst case height is O(n)
- Simple cases, such as inserting in key order, lead to the worst-case scenario
- Solution: Require a **Balance Condition** that

  1. Ensures depth is always **O(log n)** – strong enough!
  2. Is efficient to maintain – not too strong!

AVL Trees

Definition:

An AVL tree is a binary search tree such that for any node in the tree, the height of the left and right subtrees can differ by at most 1.



Recap: height is the length of the longest path from root to a leaf, here height of the tree is 3, subtree rooted by node 2 is 2, by 8 is 0. Tree is unbalanced.

An AVL tree is a binary search tree with a *balance* condition.
AVL is named for its inventors: **A**del'son-**V**el'skii and **L**andis
AVL tree *approximates* the ideal tree (completely balanced tree).
AVL Tree maintains a height close to the minimum.

The AVL Balance Condition

- Left and right subtrees of every node have **heights differing by at most 1**
- Definition: **balance**(node) = height(node.left) – height(node.right)
- AVL property: for every node x, $-1 \leq \text{balance}(x) \leq 1$
- Ensures small depth
  - Will prove this by showing that an AVL tree of height h must have a number of nodes exponential in h
- Efficient to maintain using single and double rotations

The AVL Tree Data Structure

Structural properties

1. Binary tree property
2. Balance property: balance of every node is between -1 and 1

Result:

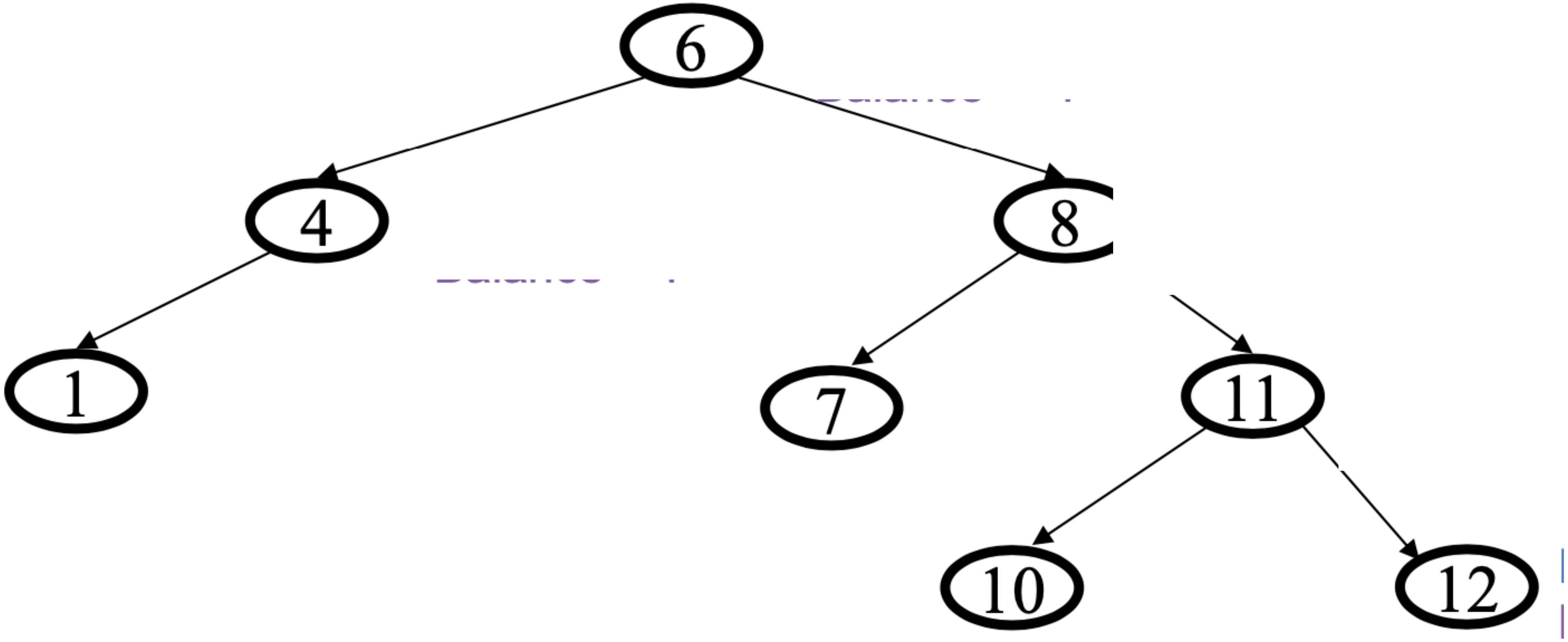
Worst-case depth is O(log n)

Ordering property

Same as for BST

empty height = -1

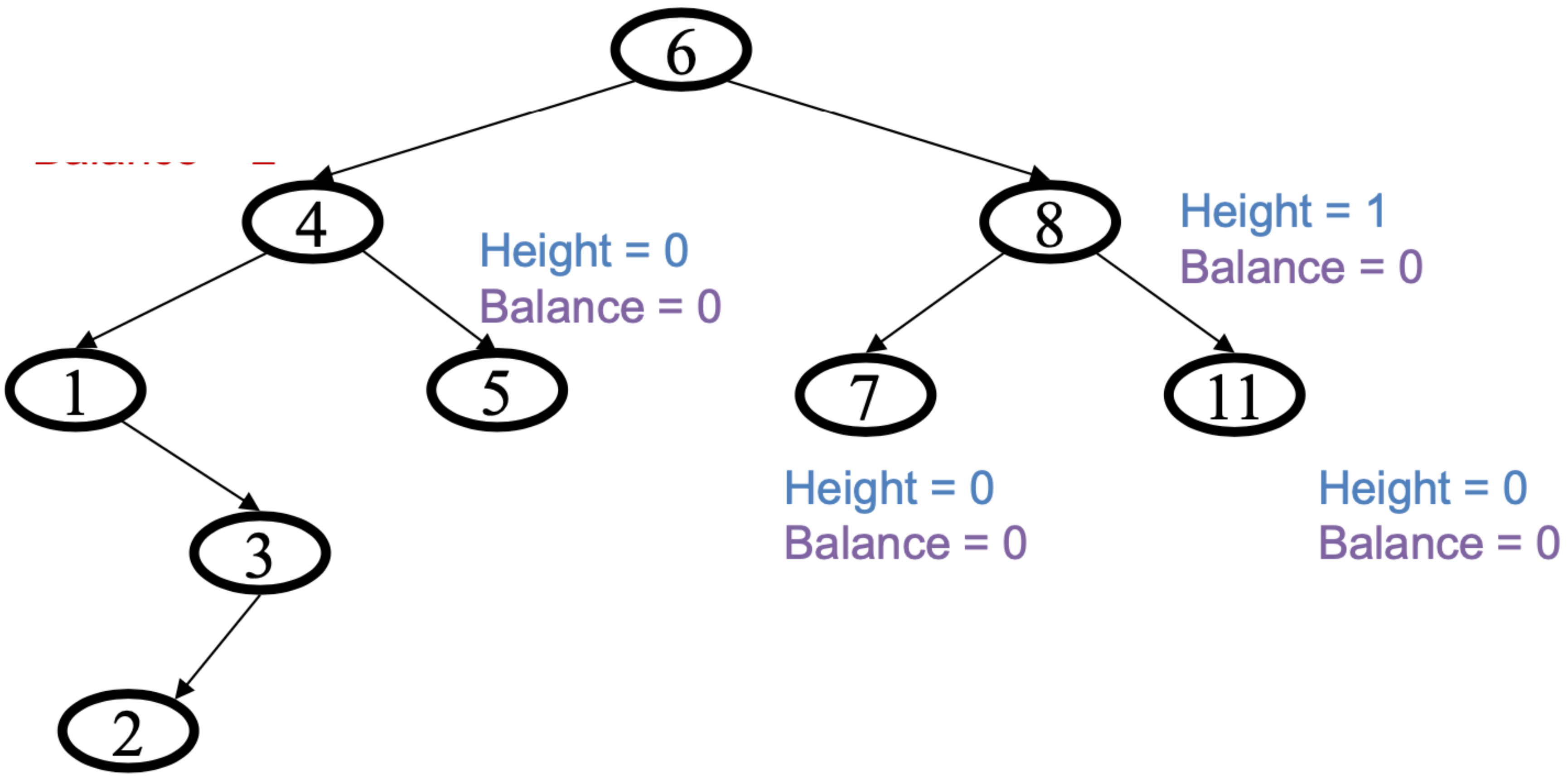
an AVL Tree ?



YES

an AVL Tree ?

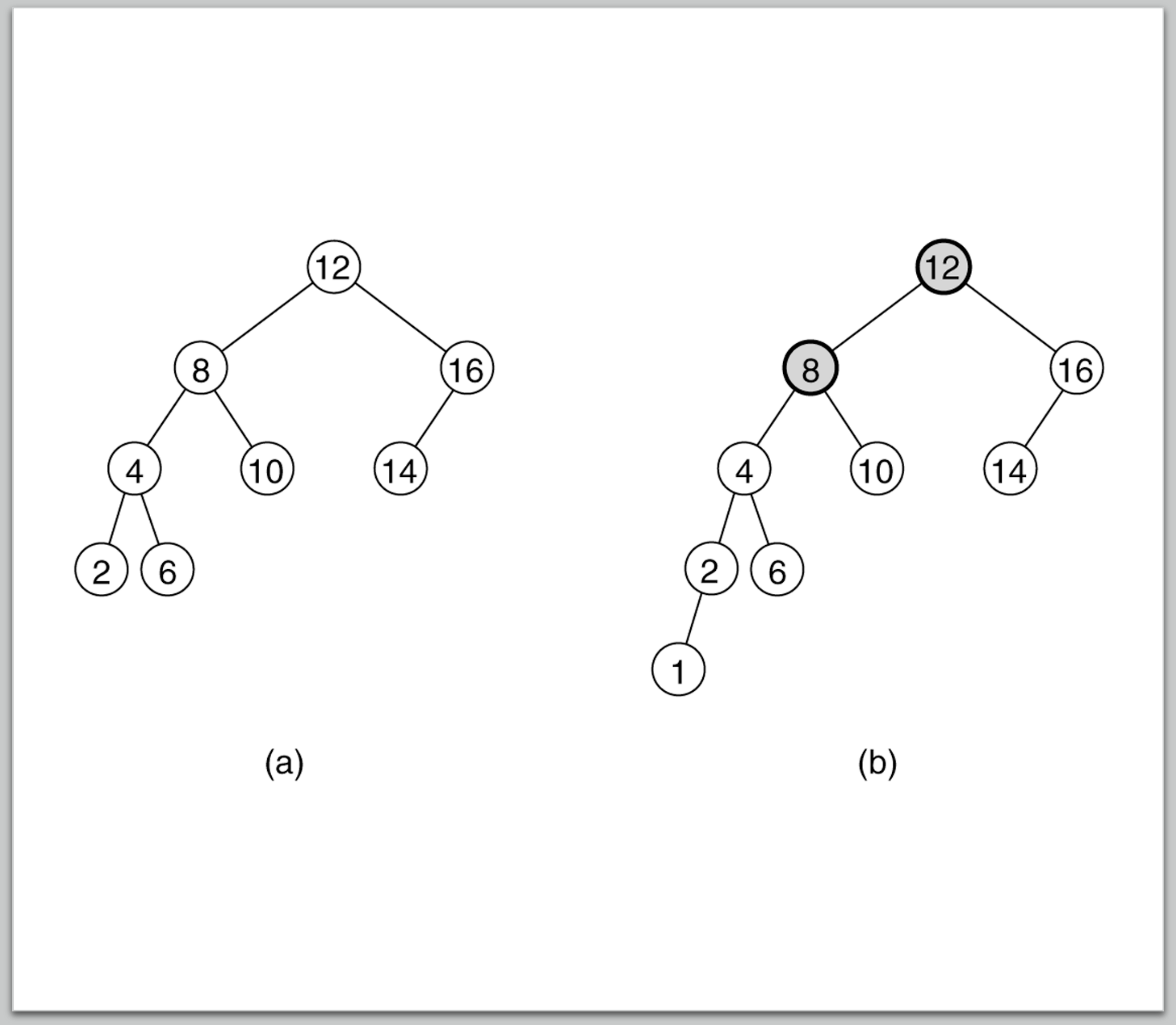
empty height = -1



Two binary search trees

(a) an AVL tree;

(b) not an AVL tree (unbalanced nodes are darkened)



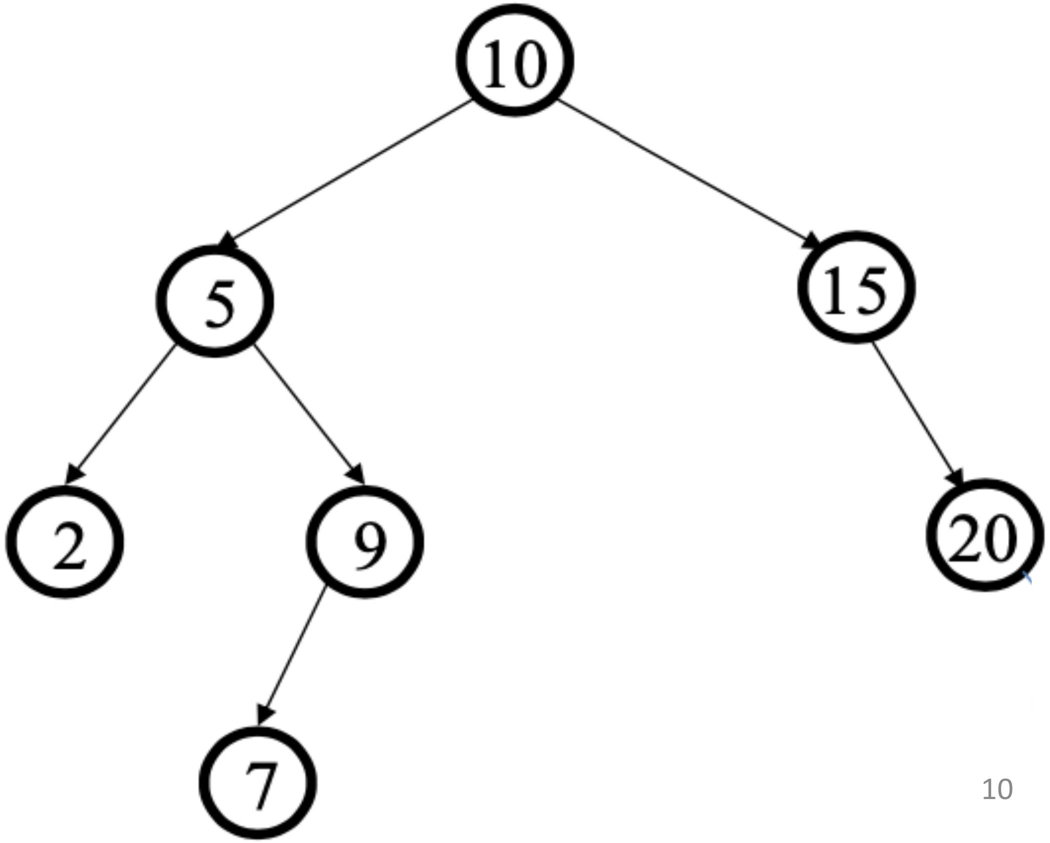
AVL Operations

If we have an AVL tree, the height is $O(\log n)$, so **find** is $O(\log n)$

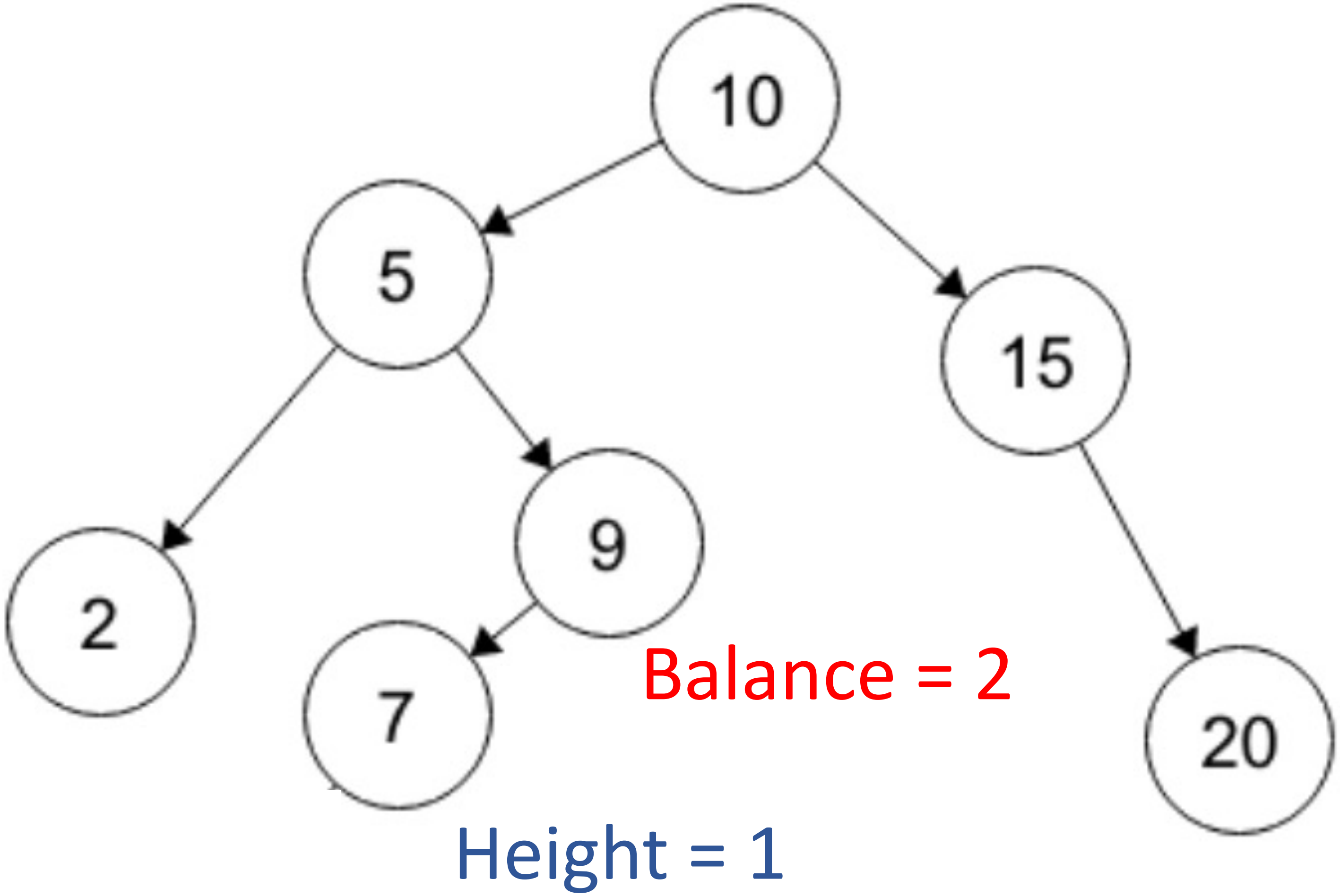
But as we insert and delete elements, we need to:

1. **Track balance**
2. **Detect imbalance**
3. **Restore balance**

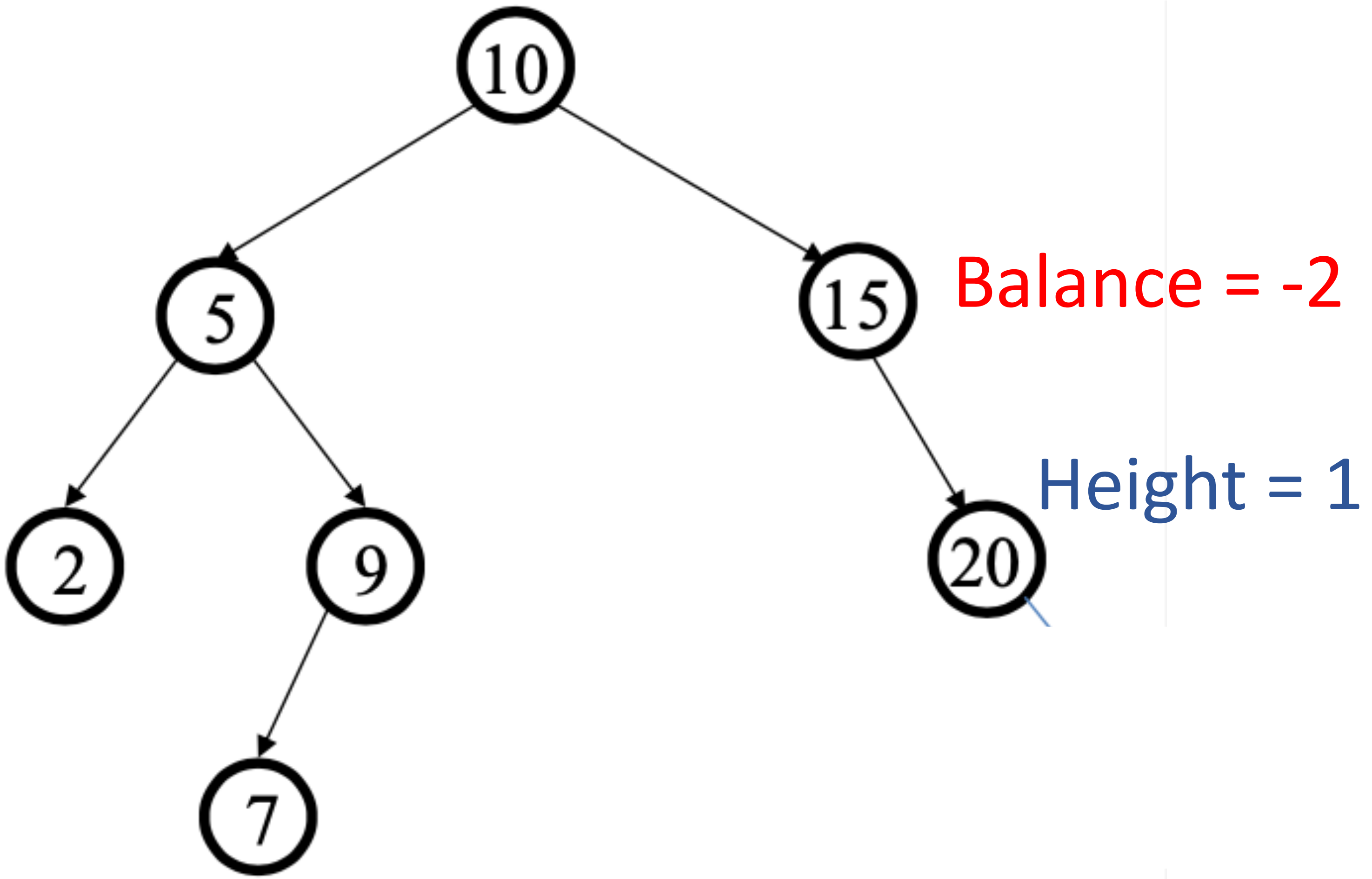
Is this AVL tree balanced?



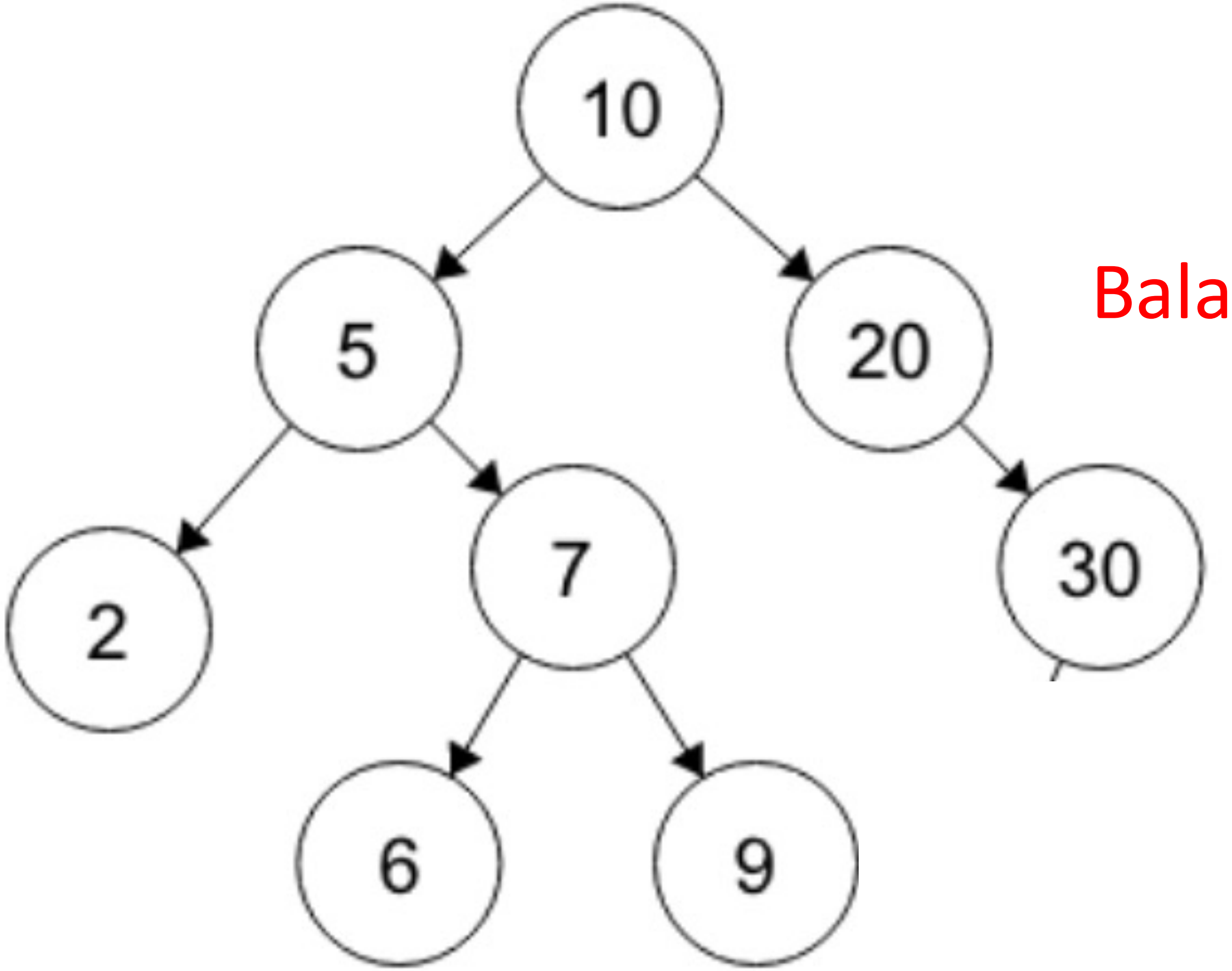
Insert 6, Balance of 9 is off !



Insert 30, Balance of 15 is off !



Insert 28, Balance of 20 is off !



Balance = -2

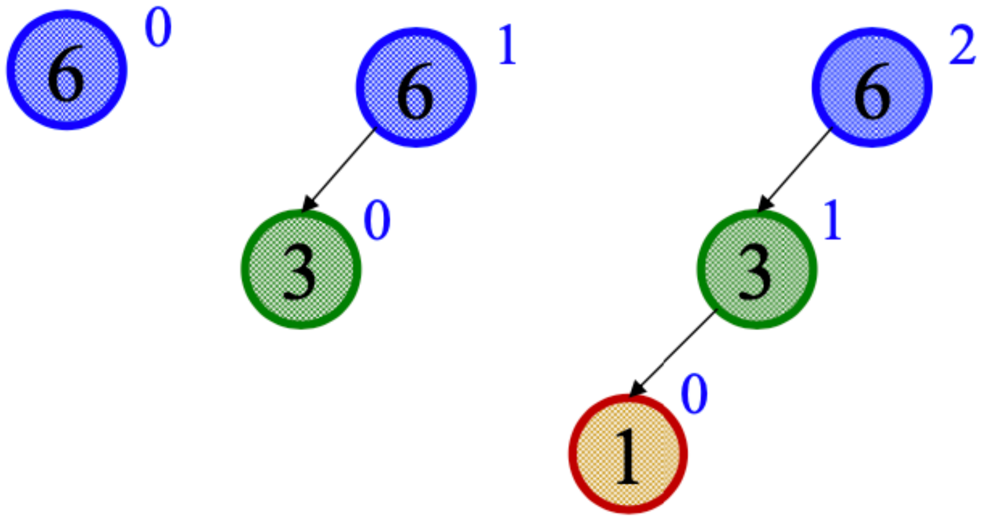
Height = 1

Case #1 : Example

Insert(6)

Insert(3)

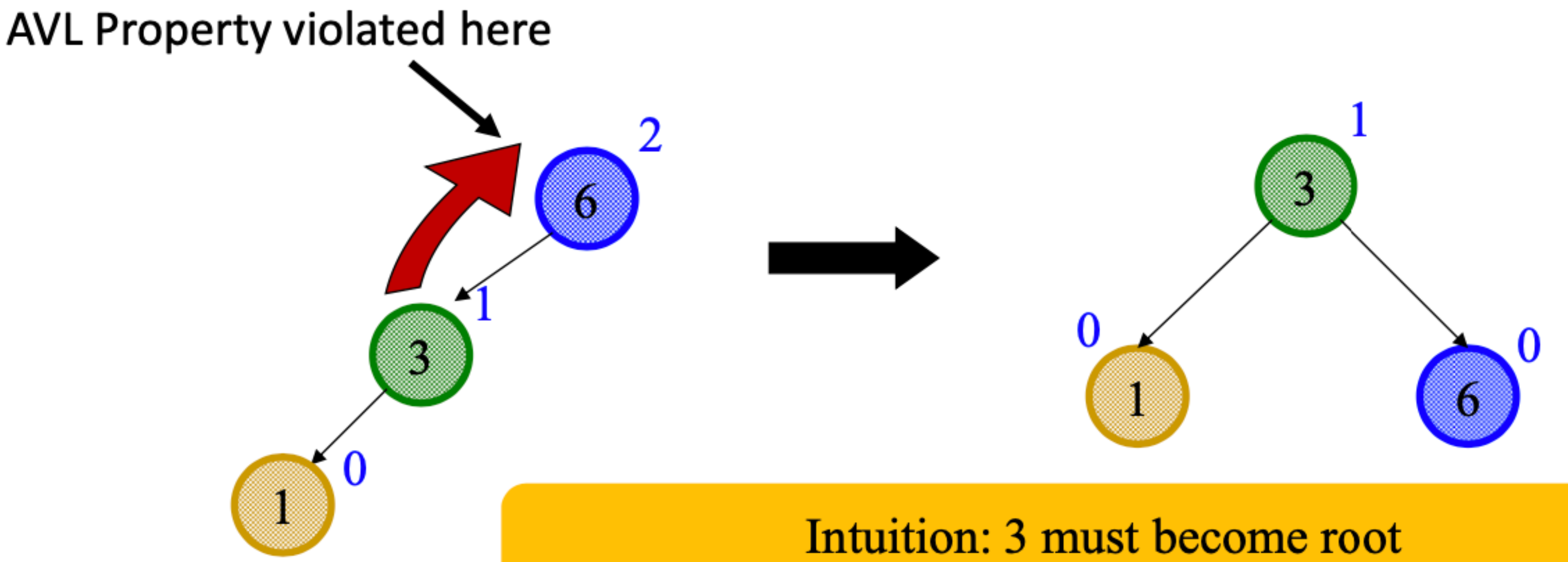
Insert(1)



- Third insertion violates balance property
- What is the only way to fix this (the only valid AVL tree with these nodes?)

Fix: Apply "Single Rotation"

- *Single rotation:* The basic operation we'll use to rebalance
  - Move child of unbalanced node into parent position
  - Parent becomes the "other" child (always okay in a BST!)
  - Other subtrees move in only way BST allows (next slide)

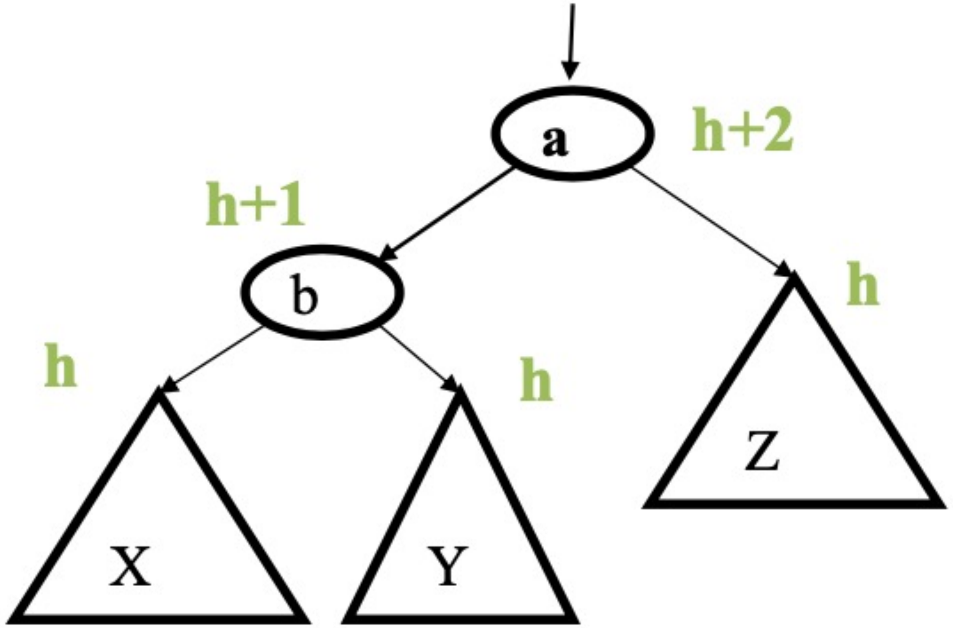


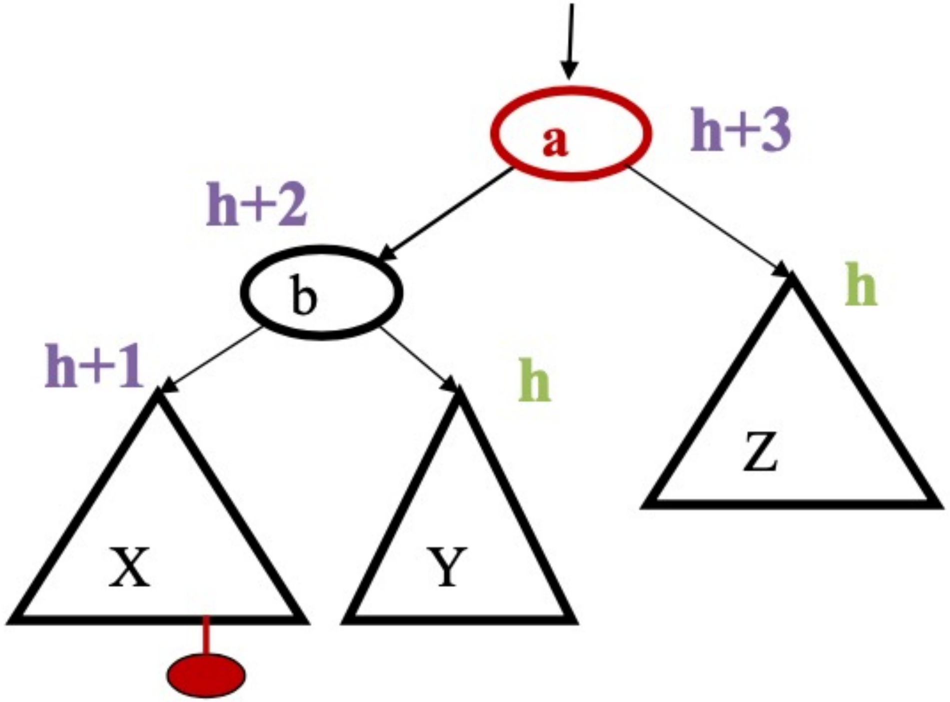
Intuition: 3 must become root

New parent height is now the old parent's height before insert

The example generalized

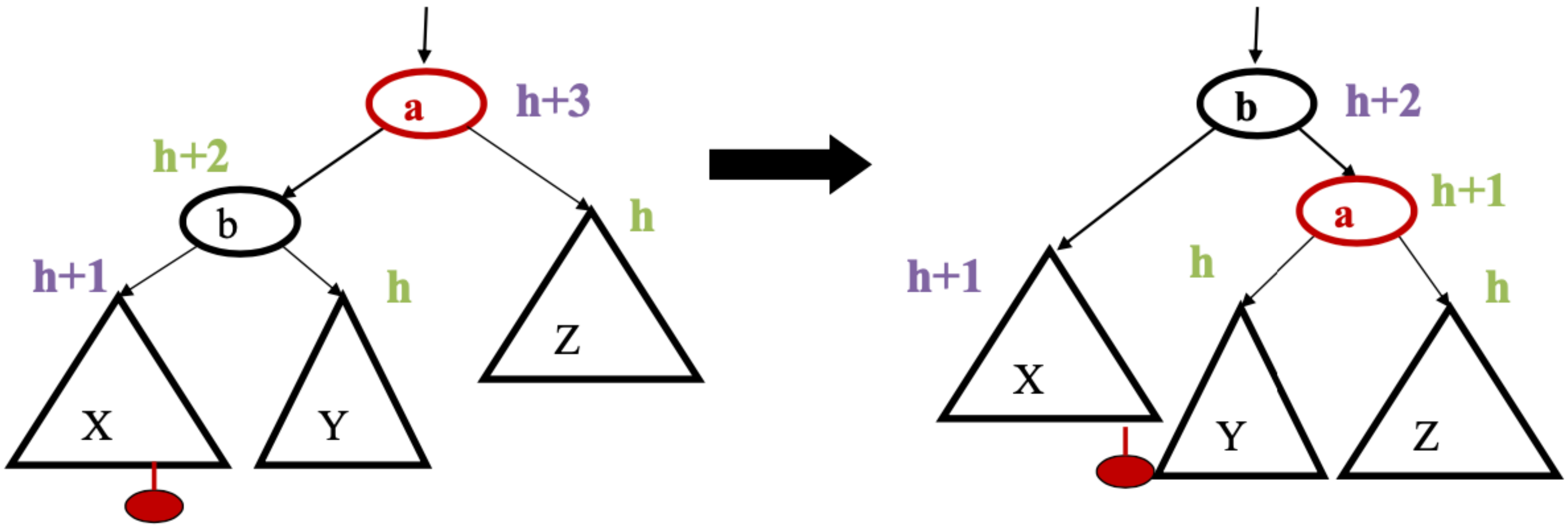
- Node imbalanced due to insertion *somewhere* in **left-left grandchild** that causes an increasing height
  - 1 of 4 possible imbalance causes (other three coming)
- First we did the insertion, which would make ***a*** imbalanced





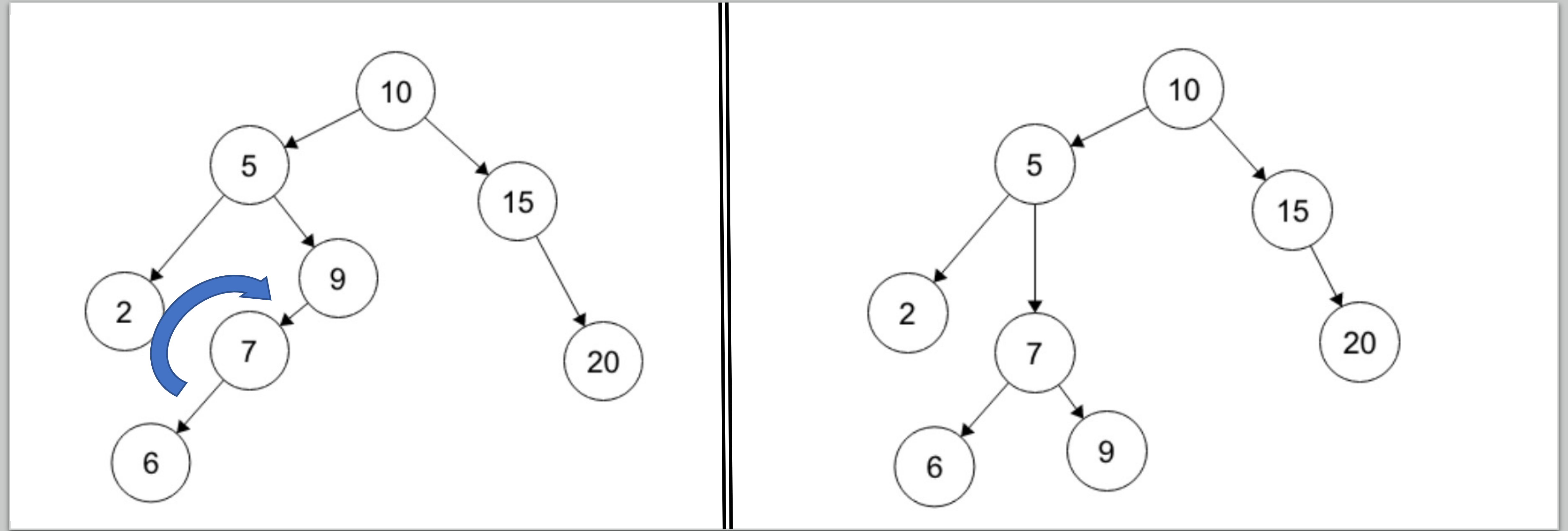
The general left-left case

- Node imbalanced due to insertion *somewhere* in **left-left grandchild**
  - 1 of 4 possible imbalance causes (other three coming)
- So we rotate at ***a***, using BST facts: X < b < Y < a < Z

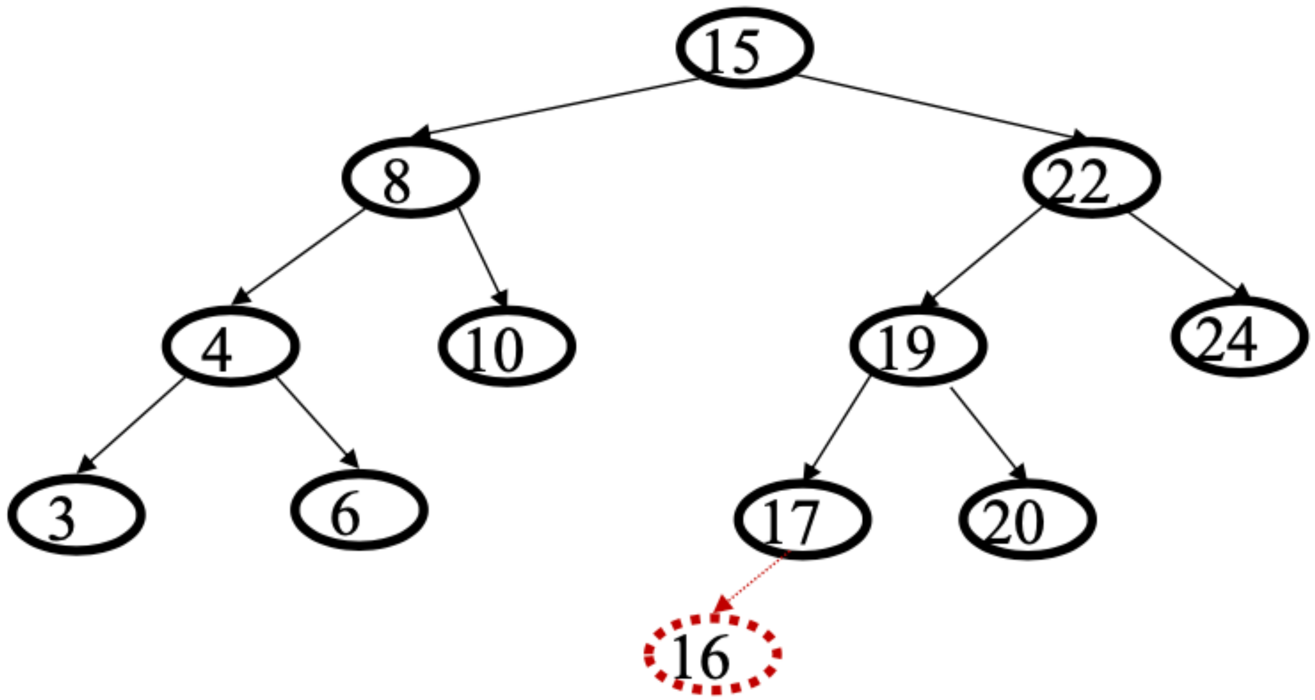


- A single rotation restores balance at the node
  - To same height as before insertion, so ancestors now balanced

Insert 6 and balance the tree

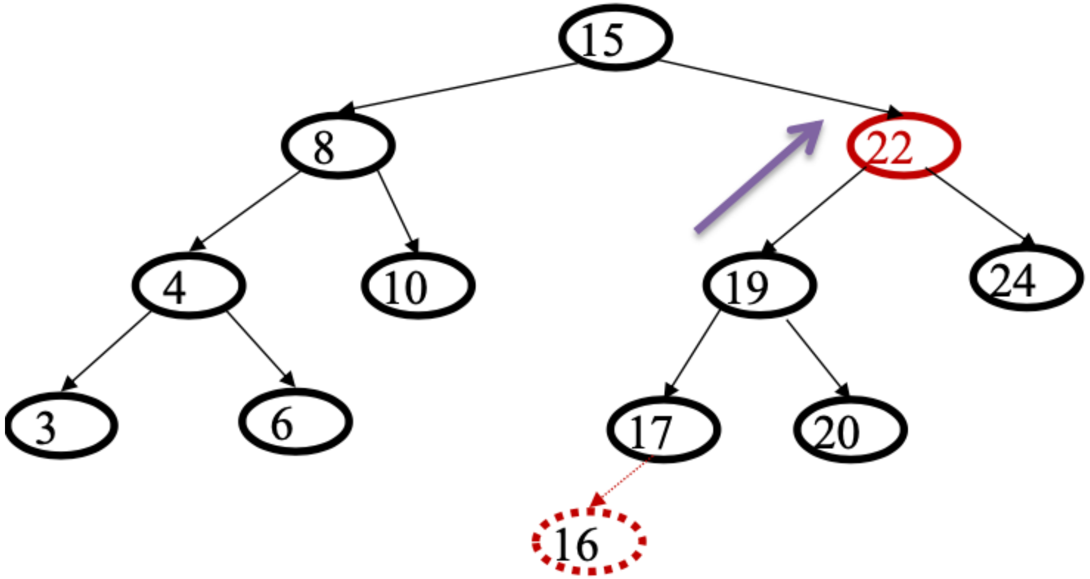


Another example: insert(16)

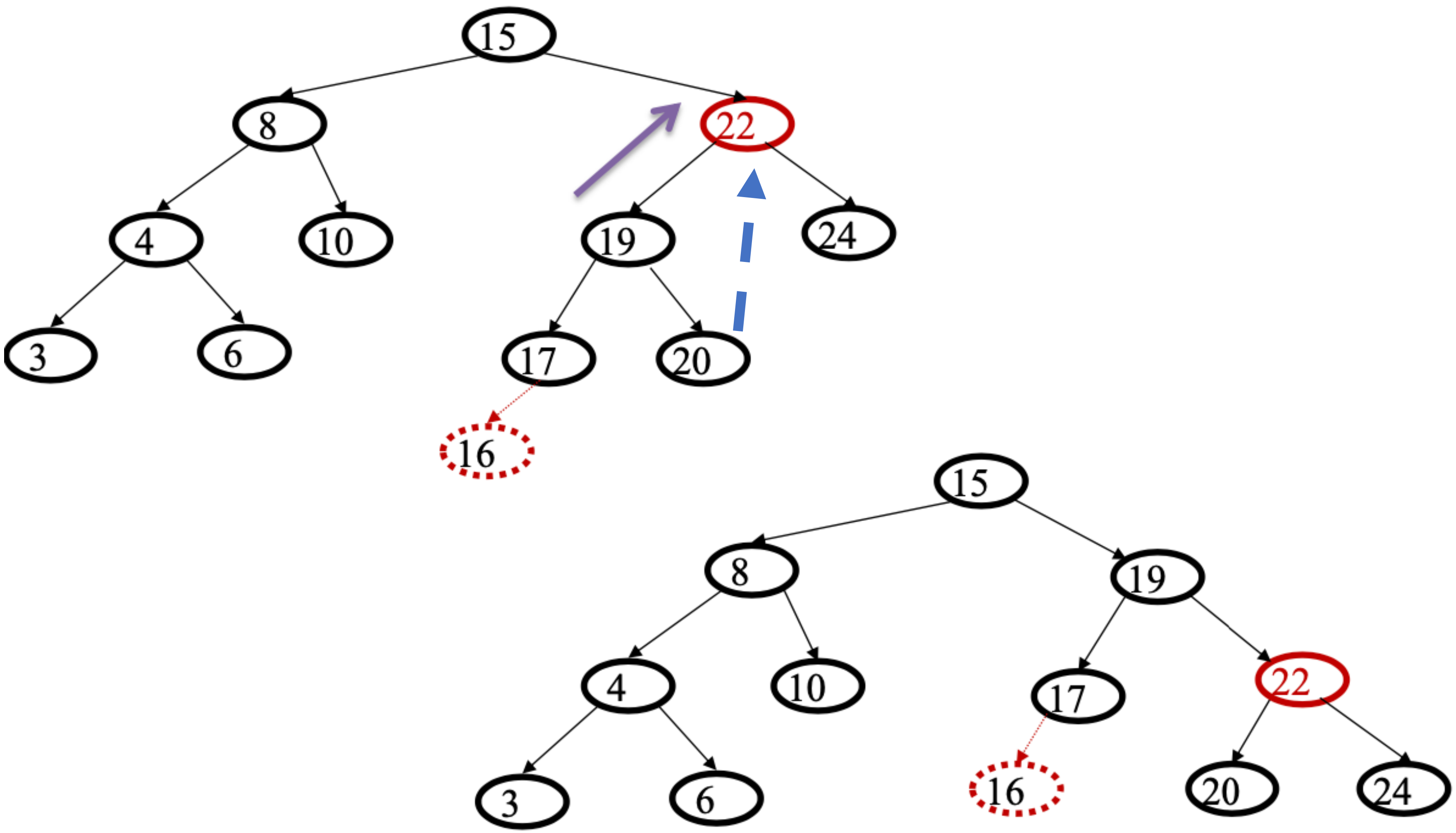


Where is the imbalance?

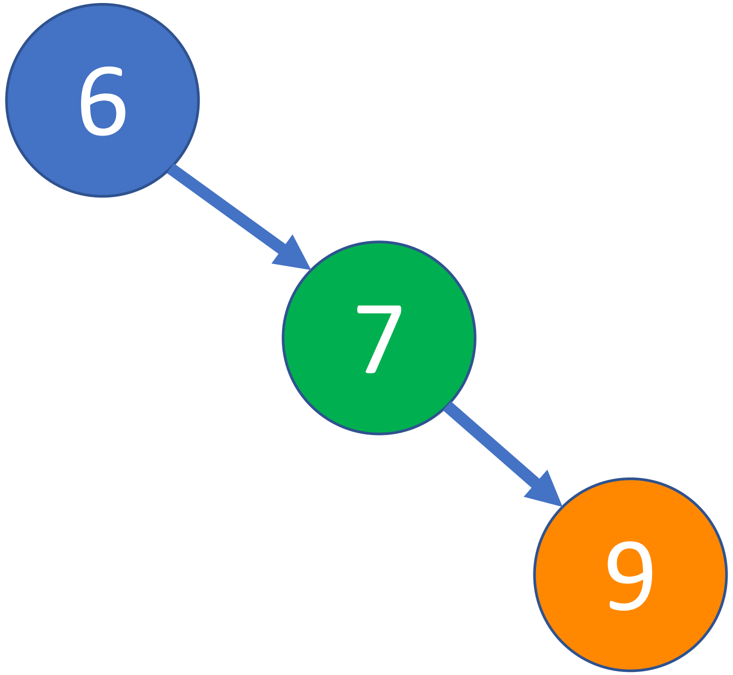
Another example: insert(16)



Another example: insert(16)



Case #2 : Example



Insert (6)

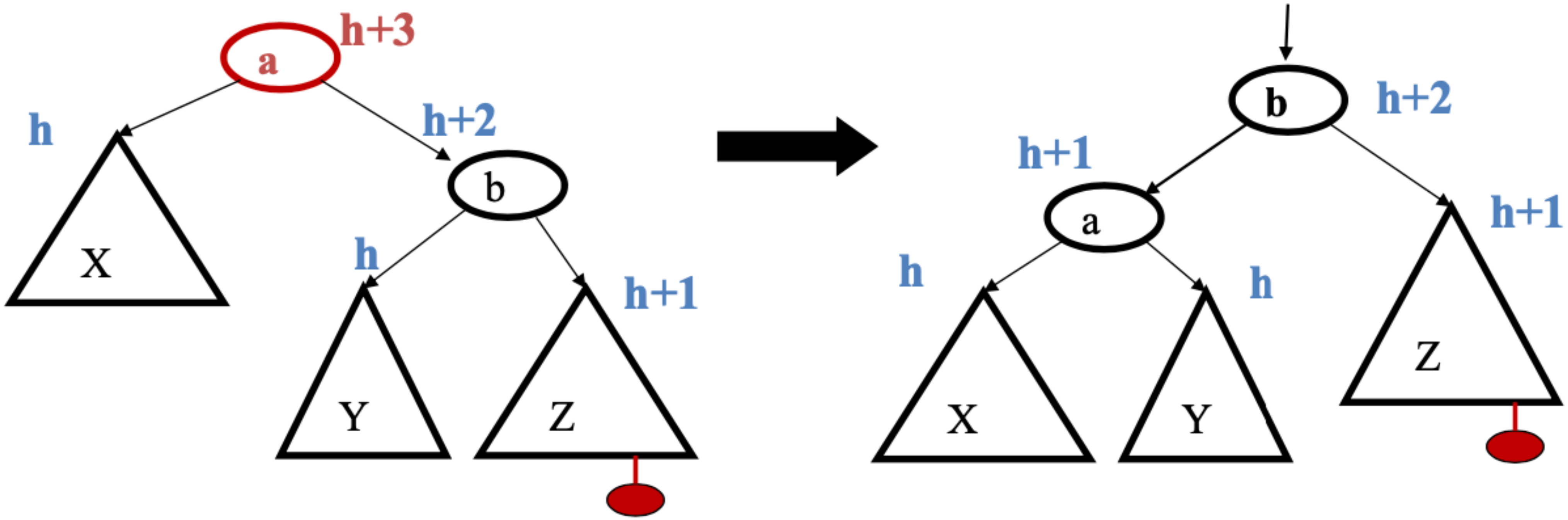
Insert (7)

Insert (9)

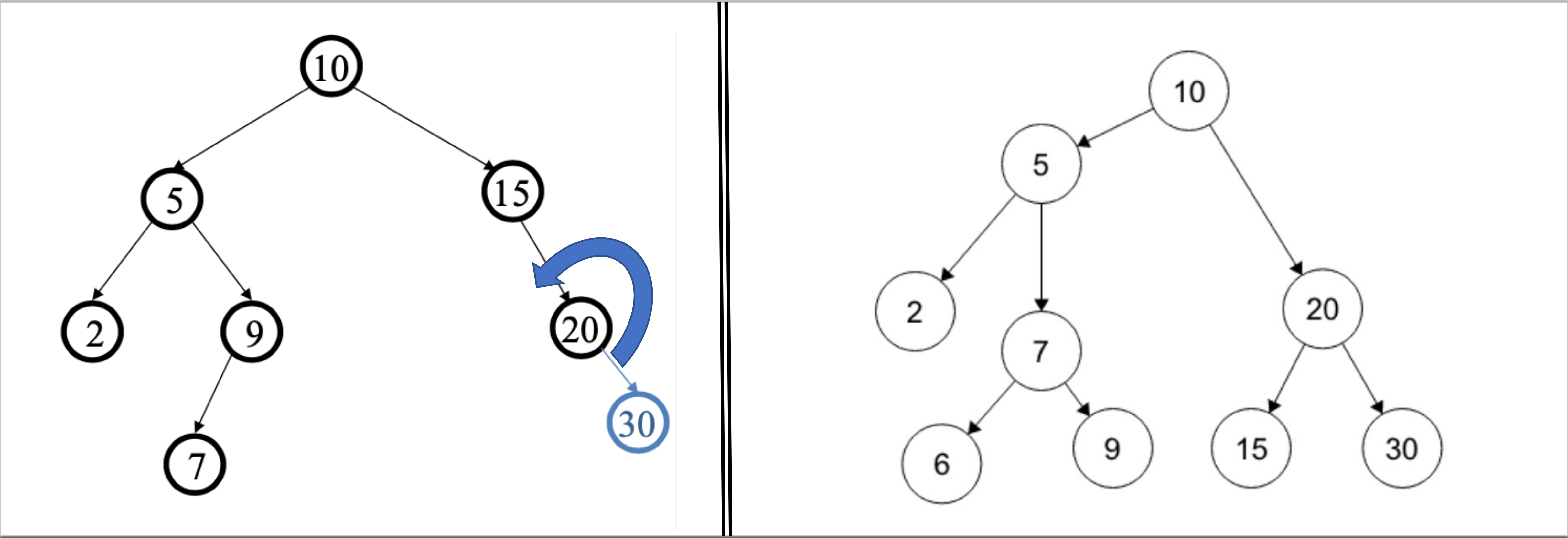
- Third insertion violates balance property
- What is the only way to fix this (the only valid AVL tree with these nodes?)

The general right-right case

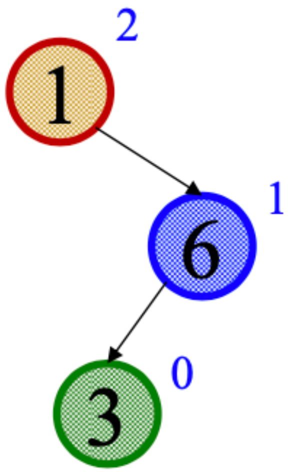
- Mirror image to left-left case, so you rotate the other way
  - Exact same concept, but need different code



Insert 30 and balance the tree

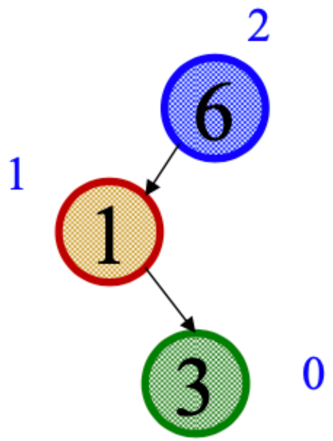


Case 3 & 4: left-right and right-left



Insert(1)
Insert(6)
Insert(3)

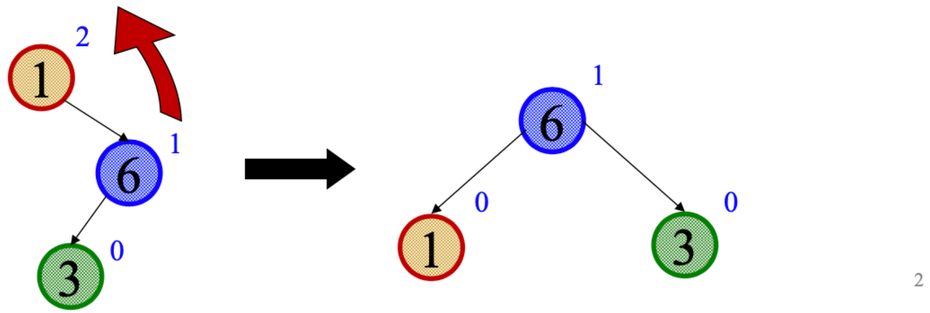
Is there a single rotation that can fix either tree?



Insert(6)
Insert(1)
Insert(3)

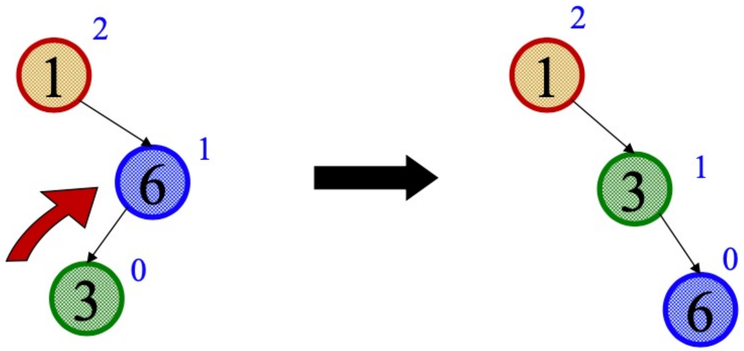
Case 3 & 4: left-right and right-left

– First wrong idea: single rotation like we did for left-left



Rotation violates the BST property

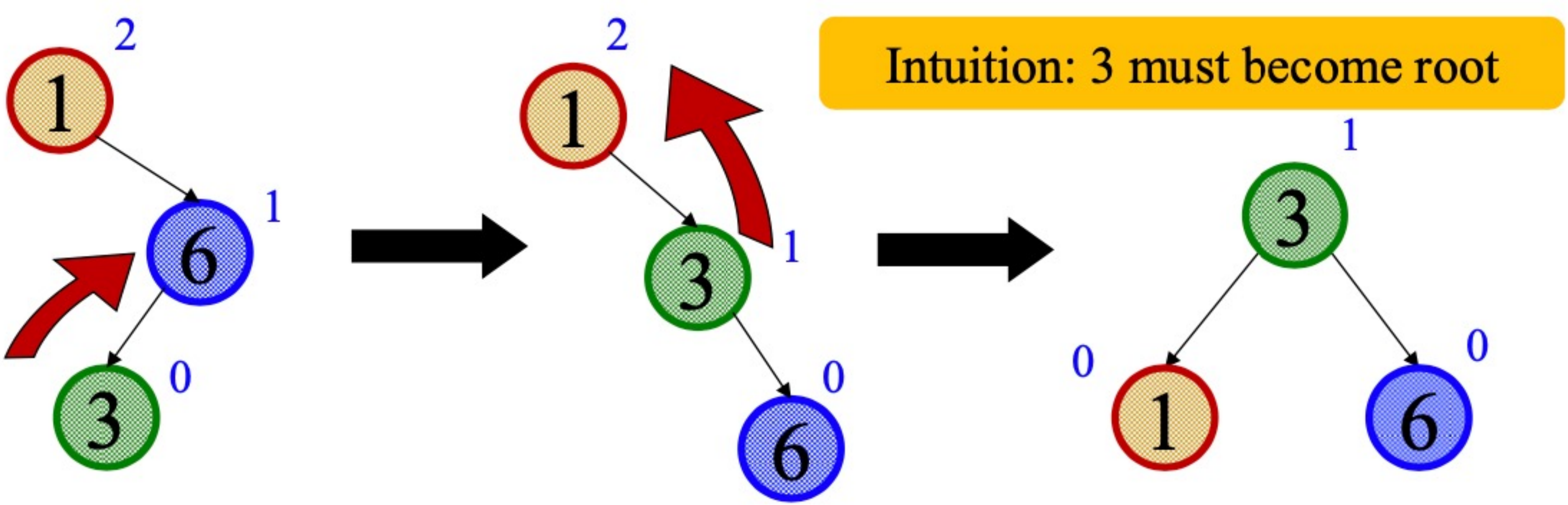
– Second wrong idea: single rotation on the child of the unbalanced node



Rotation doesn't fix balance

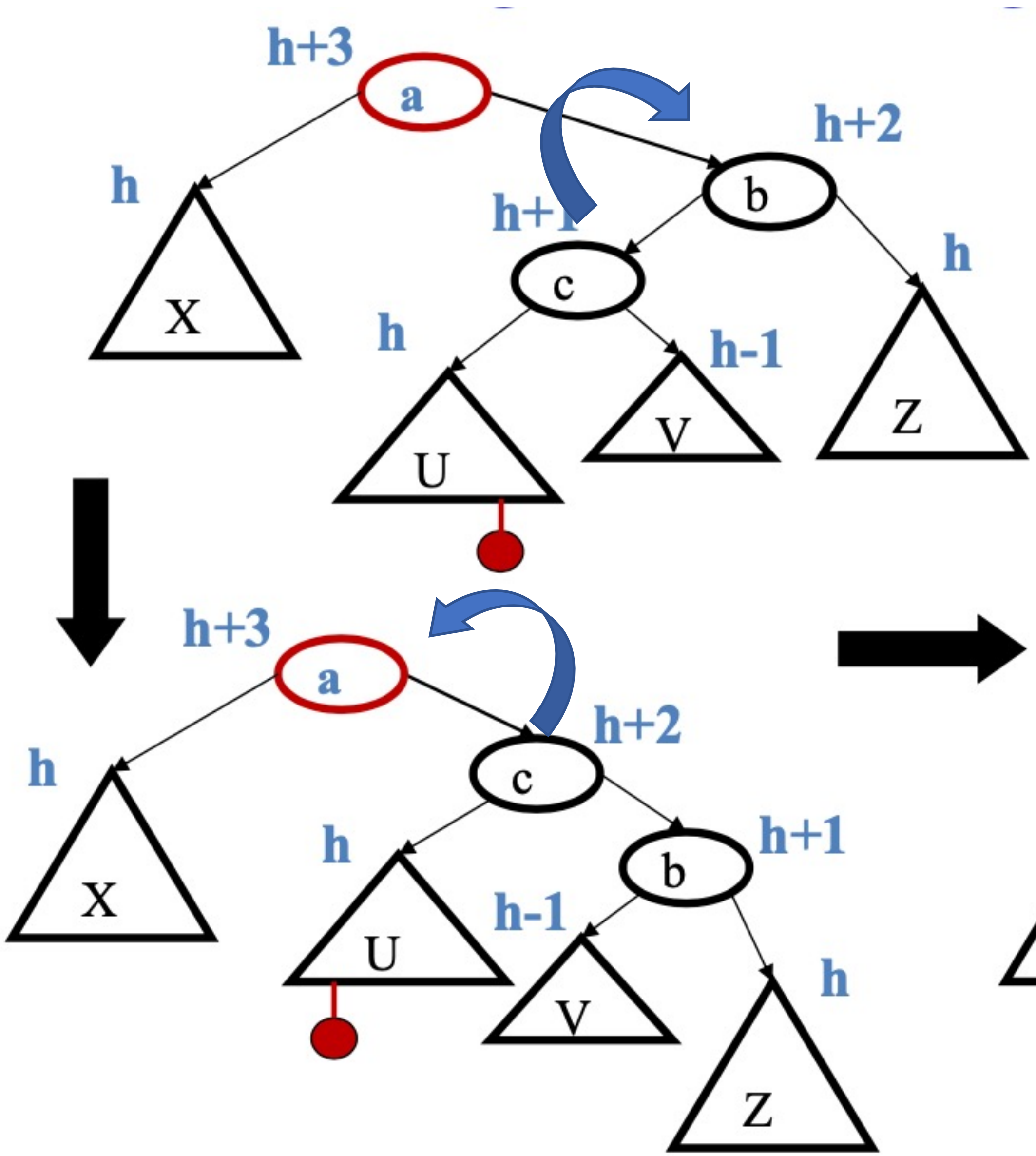
Case 3 & 4: Double rotations

1. Rotate problematic child and grandchild
2. Then rotate between self and new child



Insert(1) Insert(6) Insert(3)

The general right-left case

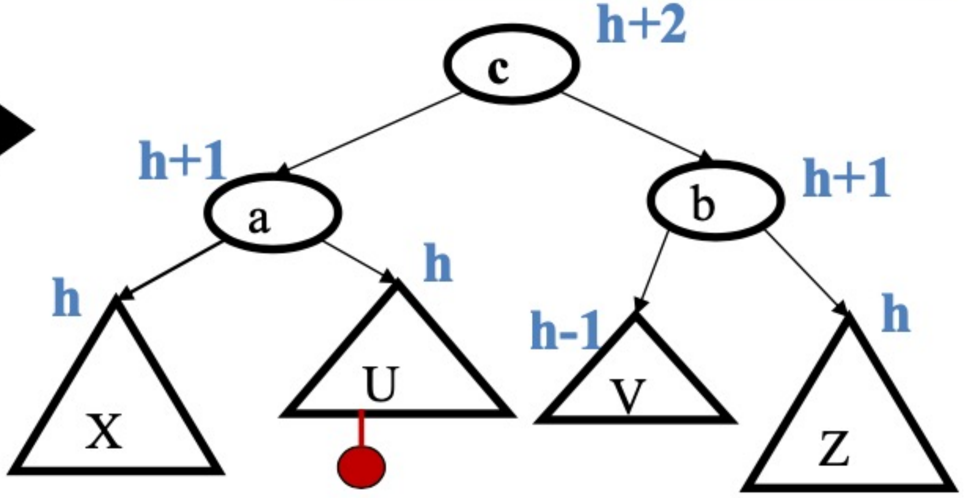


Rotation 1:

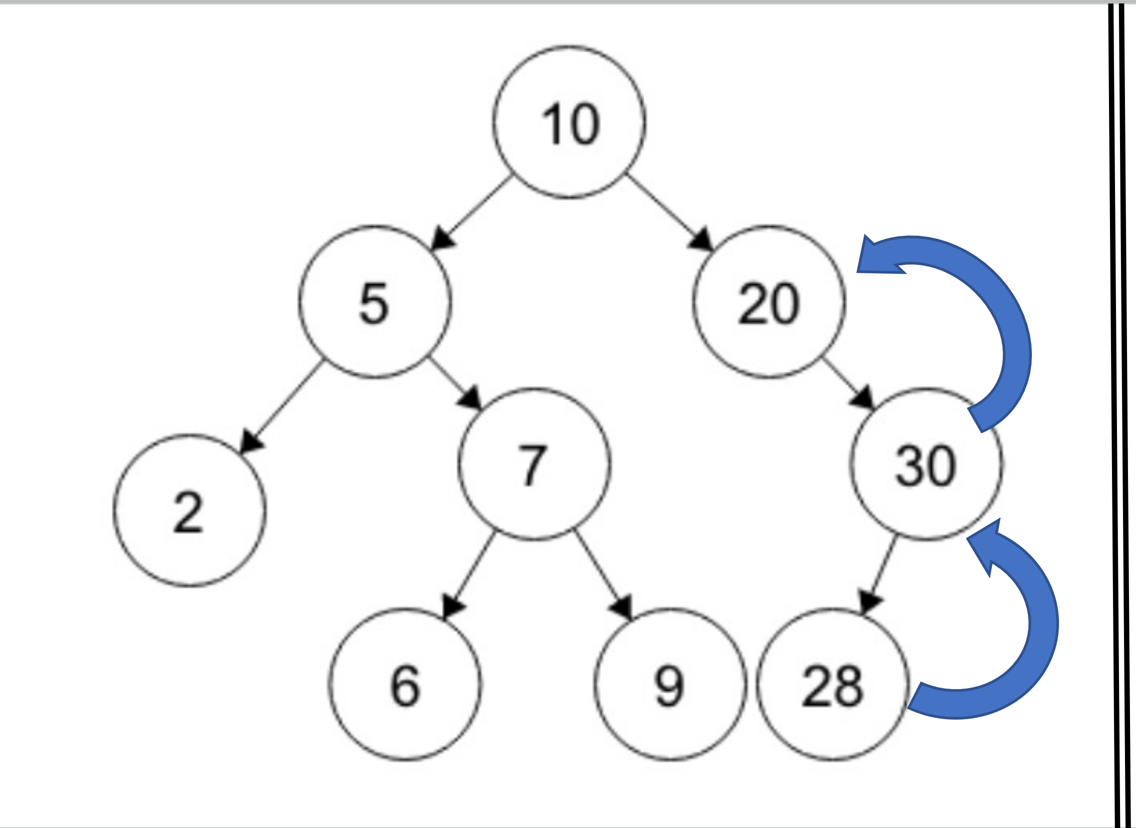
b.left = c.right
c.right = b
a.right = c

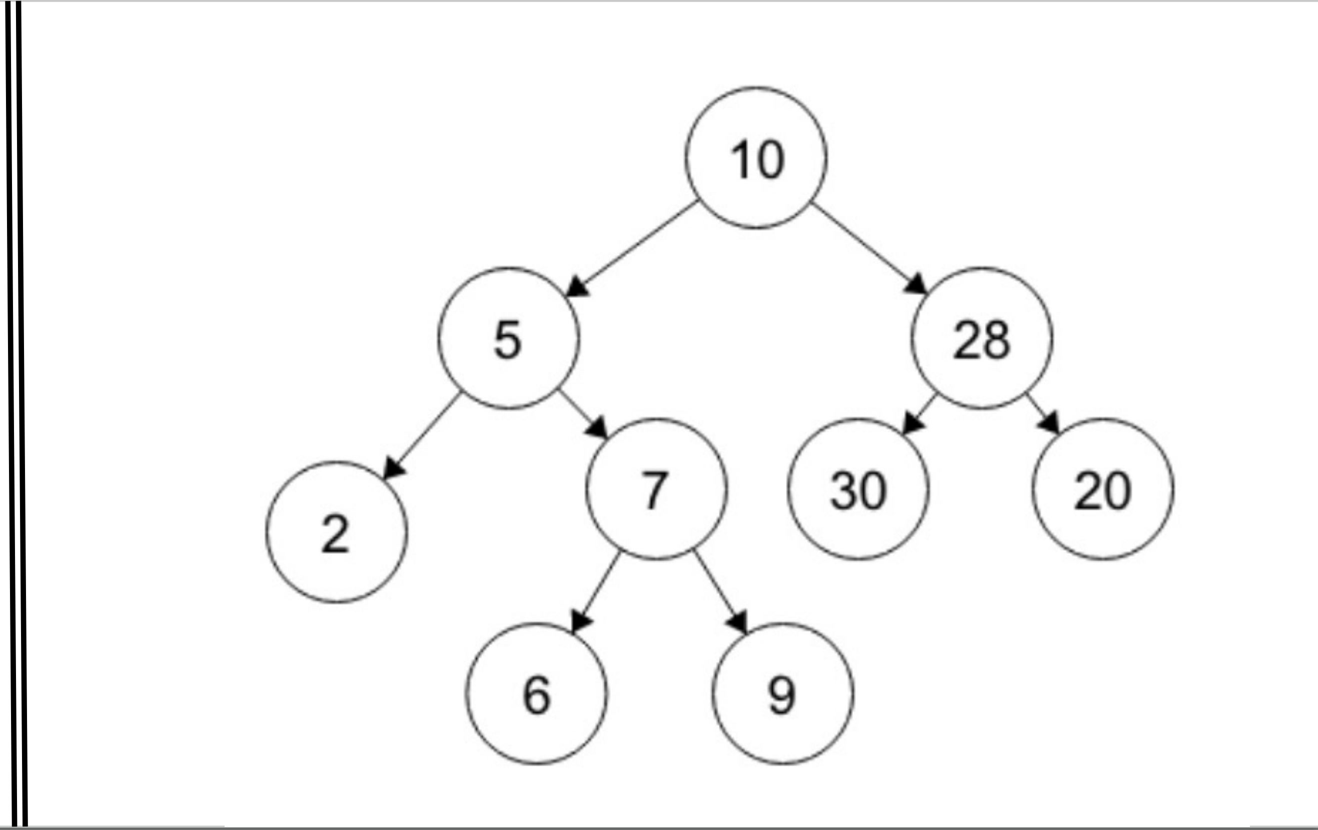
Rotation 2:

a.right = c.left
c.left = a
root = c



Insert 28 and balance the tree





Summary

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

The rebalancing is performed through four separate rotation algorithms.

AVL Trees in action

- AVL trees are commonly used in **compiler design** to implement symbol tables.
- **AVL-based database** indexes allow very fast search, insert and delete operations.
- AVLs are used in file systems to manage the **tree structure of directories** (e.g., ReiserFS file system)
- AVLs are used in **routing tables to store IP addresses** and corresponding routing information.
- AVLs are used in some sorting algorithms to improve performance (e.g., merge sort algorithm).

Pros and Cons of AVL Trees and other structures

Arguments for AVL trees:

1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(log n)
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (**e.g., B-trees**).
4. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast **(e.g., Splay trees**).

Balanced Search Trees ...

- AVL trees (Adelson-Velsii and Landis 1962) = équilibre sur la hauteur
- Red-black trees (Rudolf Bayer 1972) = équilibre hauteur + couleur
- Splay trees (Sleator and Tarjan 1985) = positionnement des nœuds les plus utilisés prêts de la racine
- Scapegoat trees (Galperin and Rivest 1993) = un facteur d'équilibrage à utiliser si bcp d'insertions et suppressions.
- Treaps (Seidel and Aragon 1996) = adapté à une priorisation des données qui seront placées plus haut dans l'arbre (par exemple les objets interactifs les plus proches dans un jeu vidéo)