AVL Tree

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

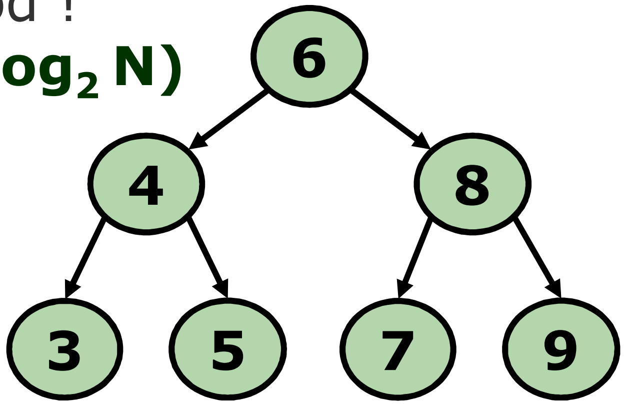
Lecture Outline

- **AVL Tree:**
  - AVL property
  - Tree Height
- **Rotation mechanism:**
  - Left and Right Rotation
- **Insertion:**
  - Single rotation
  - Double rotations

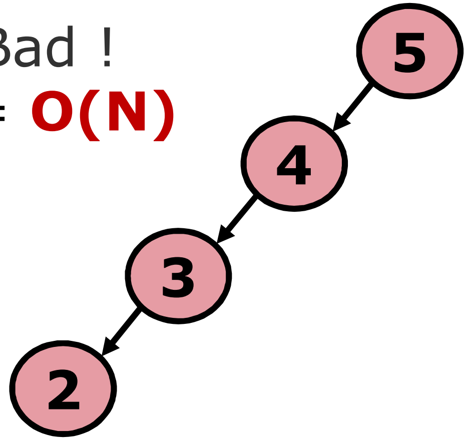
Binary Search Tree: Recap

- Operations are dependent on **BST Height**
  - *findMin* = O(h)
  - *search* = O(h)
  - *insert* = O(h)
  - *delete* = O(h)
- Height can differs **greatly:**

Good !
h = $O(\log_2 N)$



Bad !
h = $O(N)$



To Improve Performance

- Height range: $\mathbf{Log_2 N \leq h \leq N}$
  - Basic BST has no control over the height as the tree shape is determined by order of insertion
- This gives rise to a number of **self-balancing BST** (and other search trees):
  - AVL Tree (**covered**)
  - (2, 3, 4)-Tree and Red-Black Tree
  - Splay Tree, Treap, B+ Tree
  - etc…

Invented by Adel'son-Vel'skii and Landis in 1962

AVL TREE DEFINITIONS AND PROPERTIES

AVL Tree: Definition

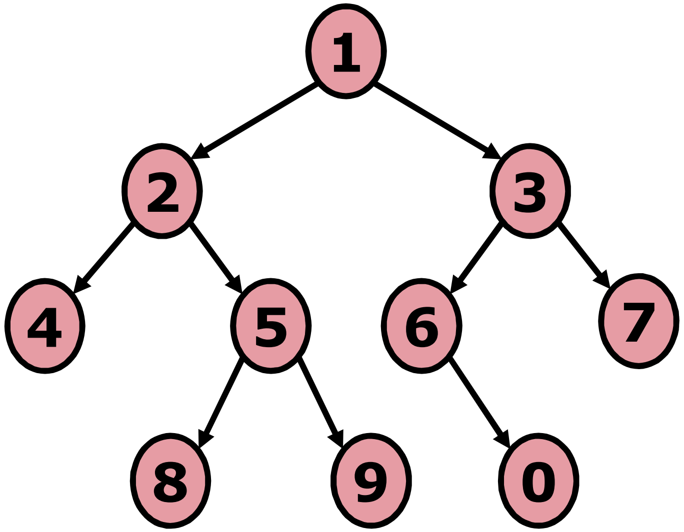
AVL Tree:

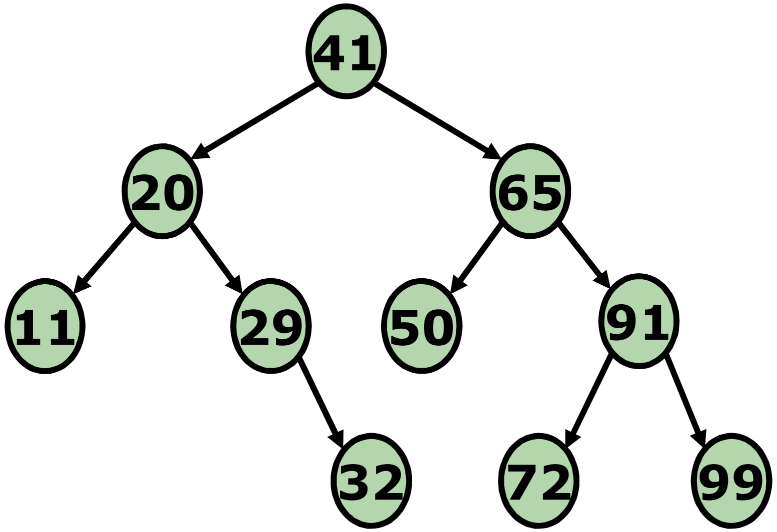
- is a **balanced BST:**
  - For all nodes, the **difference in height** between left and right subtree is **at most one:**

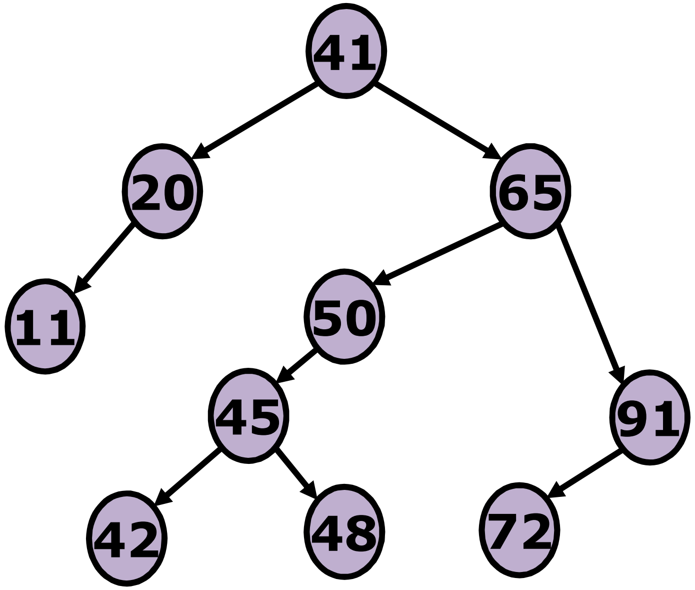
$$|H_l - H_r| \leq 1$$

  - Also known as the **AVL Property**

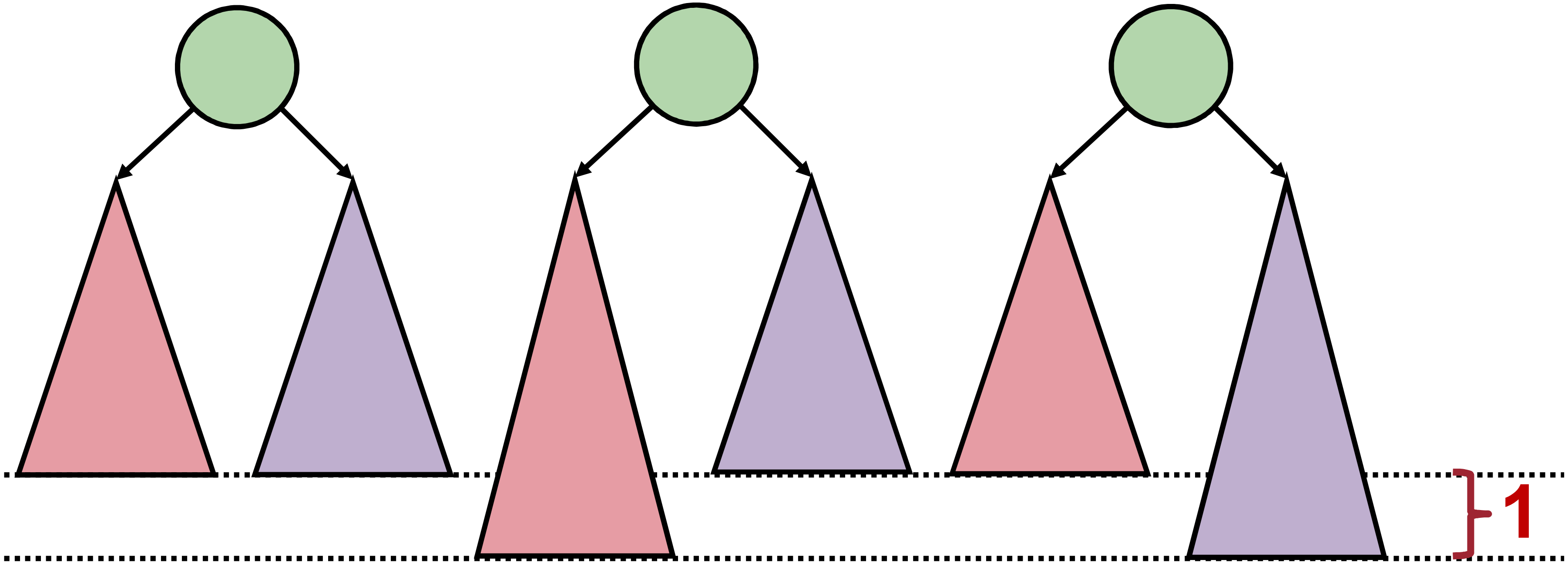
Check: **Which BST is AVL Tree?**







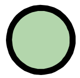
AVL Tree: Property



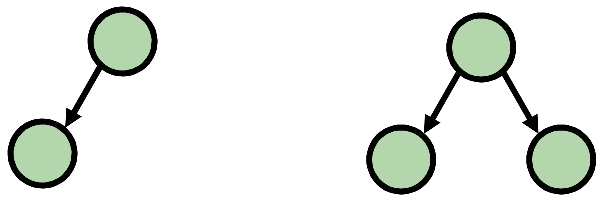
The difference between the levels of the two dotted lines is **at most one**

AVL Tree: Example

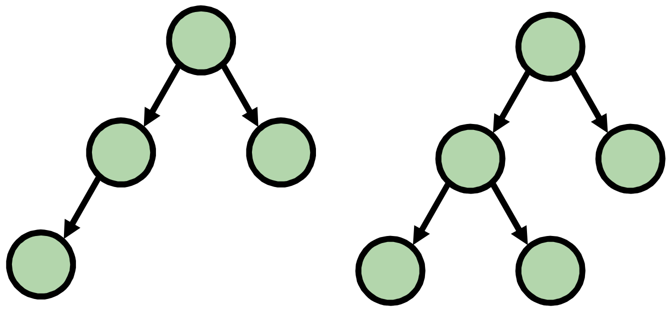
Height 1
AVL tree



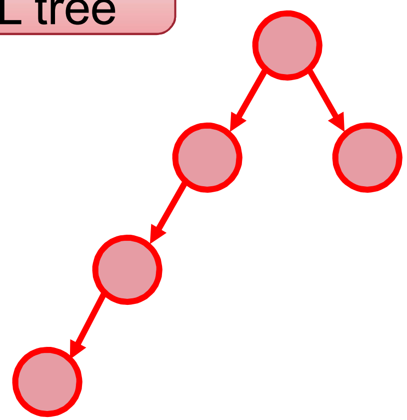
Height 2
AVL tree



Height 3
AVL tree



NOT
AVL tree



AVL Tree: Height

- Since AVL Tree is just a variant of BST
  - Operations similarly **depend on height**

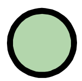
- **Important Question:**
  - **Given N nodes, what is the worst (tallest) AVL Tree?**
  - To answer the question, we first look at:

> What is the **smallest number of nodes** needed to build a **valid AVL tree of height H**?

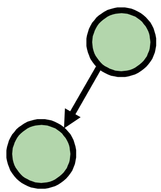
Definition: Minimal AVL Tree

Minimal AVL trees of **height h**:
AVL Trees having height h and **fewest possible number of nodes**

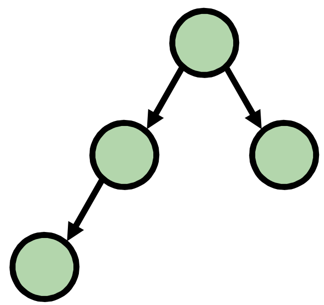
Height 1
Minimal AVL tree



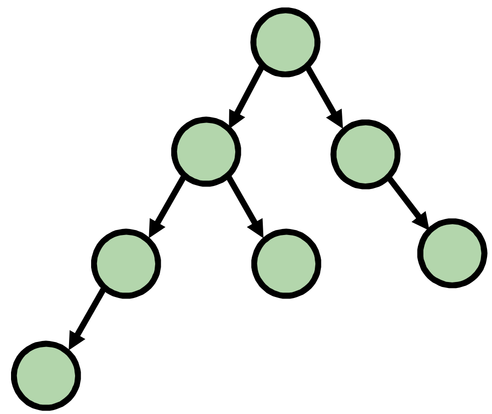
Height 2
Minimal AVL tree



Height 3
Minimal AVL tree

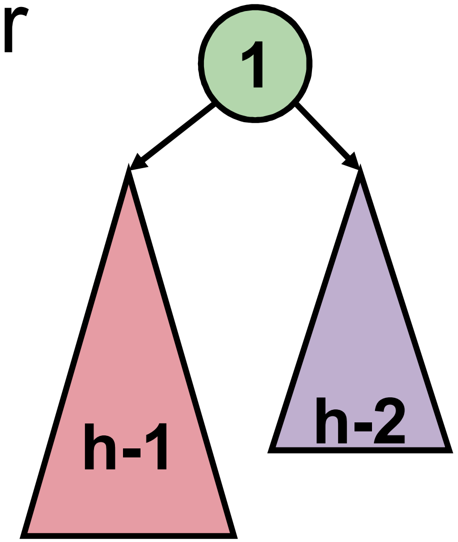


Height 4
Minimal AVL tree



Minimal AVL Tree: Height

- Given:
  - N: number of nodes in a AVL tree of height h
  - `n(h)`: number of nodes in a **minimal** AVL tree of height h
- Then:
  - `n(h)` $\leq$ N
- Assuming the left subtree is taller
  - `n(1) = 1`
  - `n(2) = 2`
  - `n(3) = 1 + n(2) + n(1)`
    `= 4`
  - `n(h) = 1 + n(h-1) + n(h-2)`



Minimal AVL Tree: Height

- $n(h) = 1 + n(h-1) + n(h-2)$

➔ $n(h) > 2\ n(h-2)$ Since $n(h-1) > n(h-2)$

- $n(h) > 2\ n(h-2)$

$> 2 * 2n(h-4)$ Since $n(h-2) > 2n(h-4)$

$> 2 * 2 * 2n(h-6)$

… …

$n(h) > 2^i\ n(h-2i)$

Minimal AVL Tree: Height

- Suppose *h* is an odd number:
  - $h-2i = 1 \rightarrow i = (h-1)/2$
  - and $n(h-2i) = n(1) = 1$

Try solving for when h is even?

$$\boxed{n(h) > 2^i\, n(h-2i)}$$

$$n(h) > 2^{(h-1)/2} \times n(1)$$

$$> 2^{(h-1)/2}$$

$$h < 2 \log_2 n(h) + 1$$

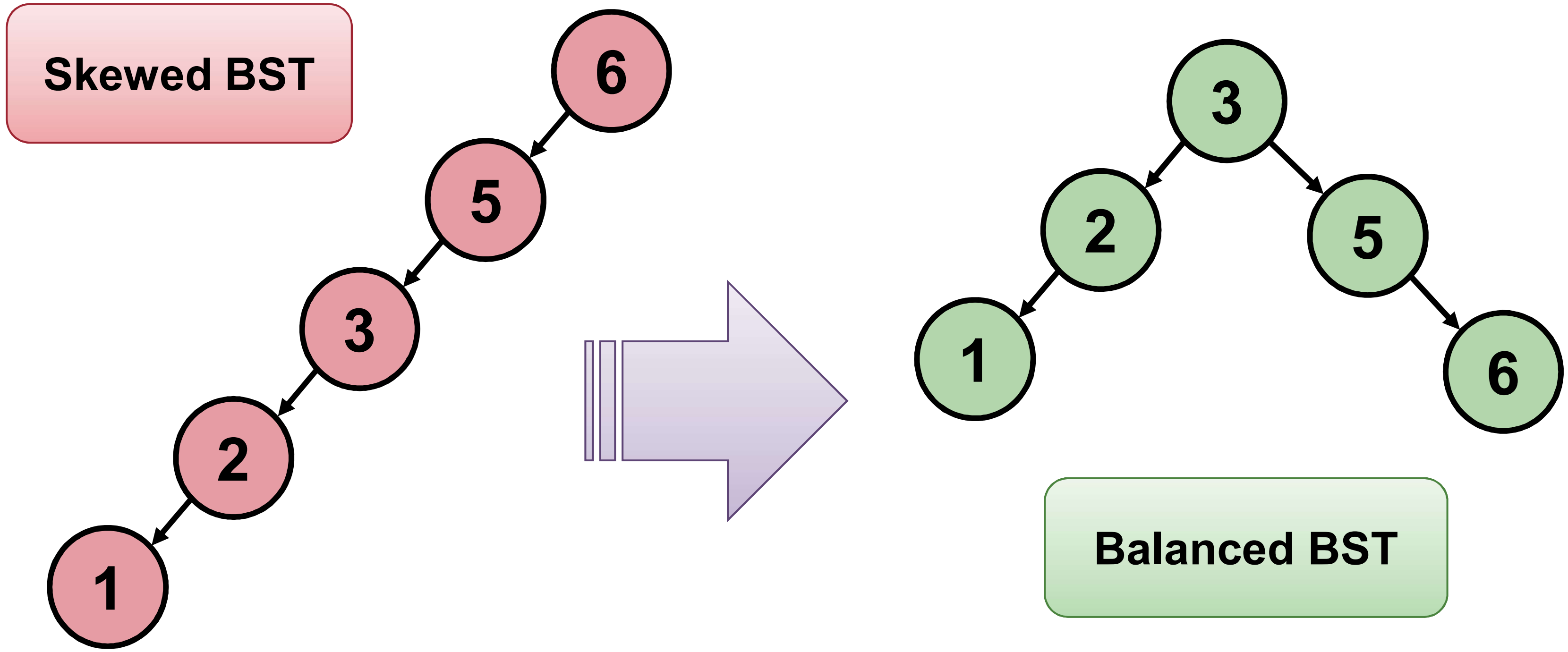
$$\boxed{h = O(\log_2 N)}$$

Conclusion:
height of AVL tree is $O(\log_2 N)$ even in the worst case

How to keep AVL tree in shape?

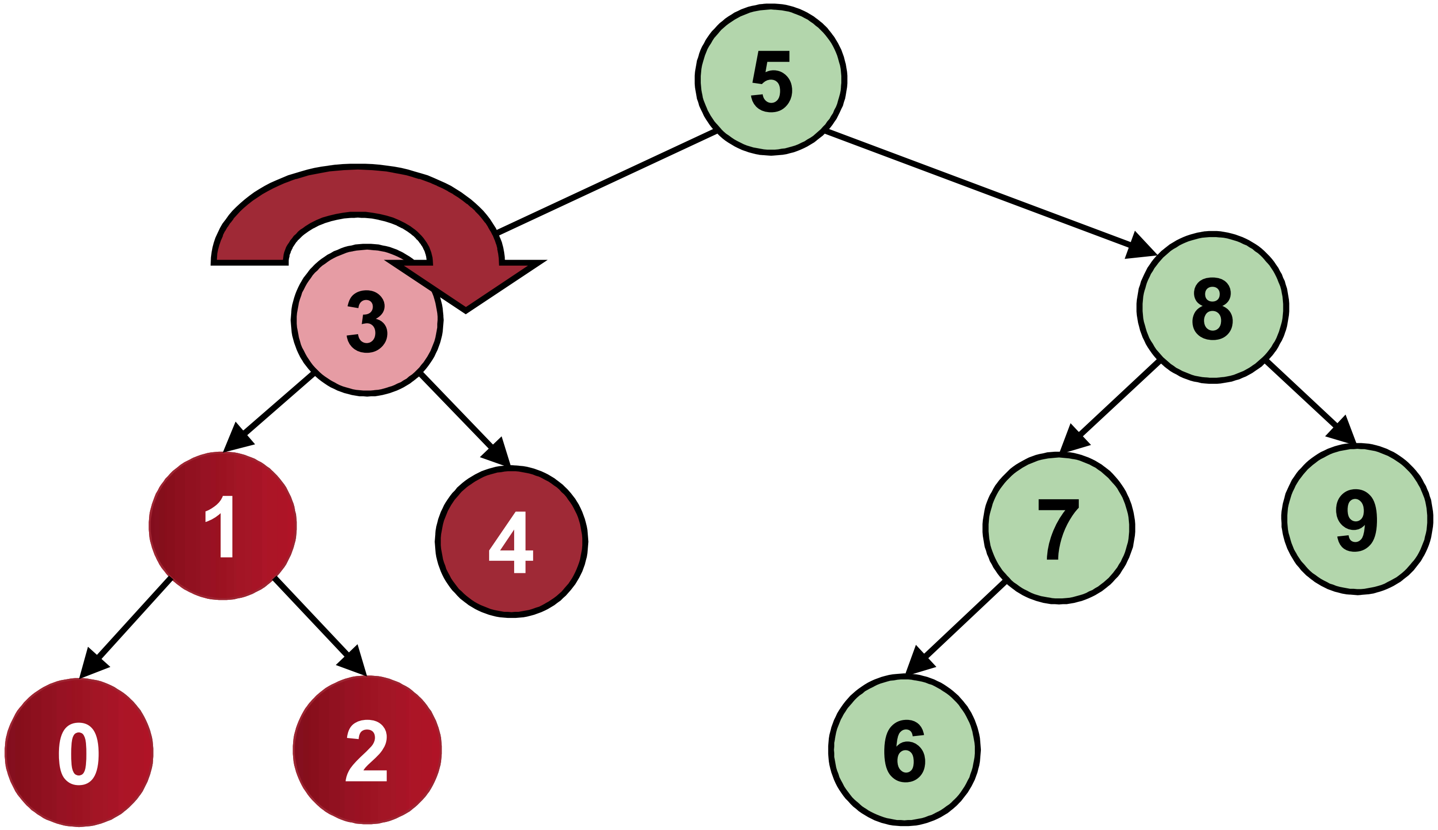
ROTATION MECHANISM

Rotation Mechanism: **Overview**



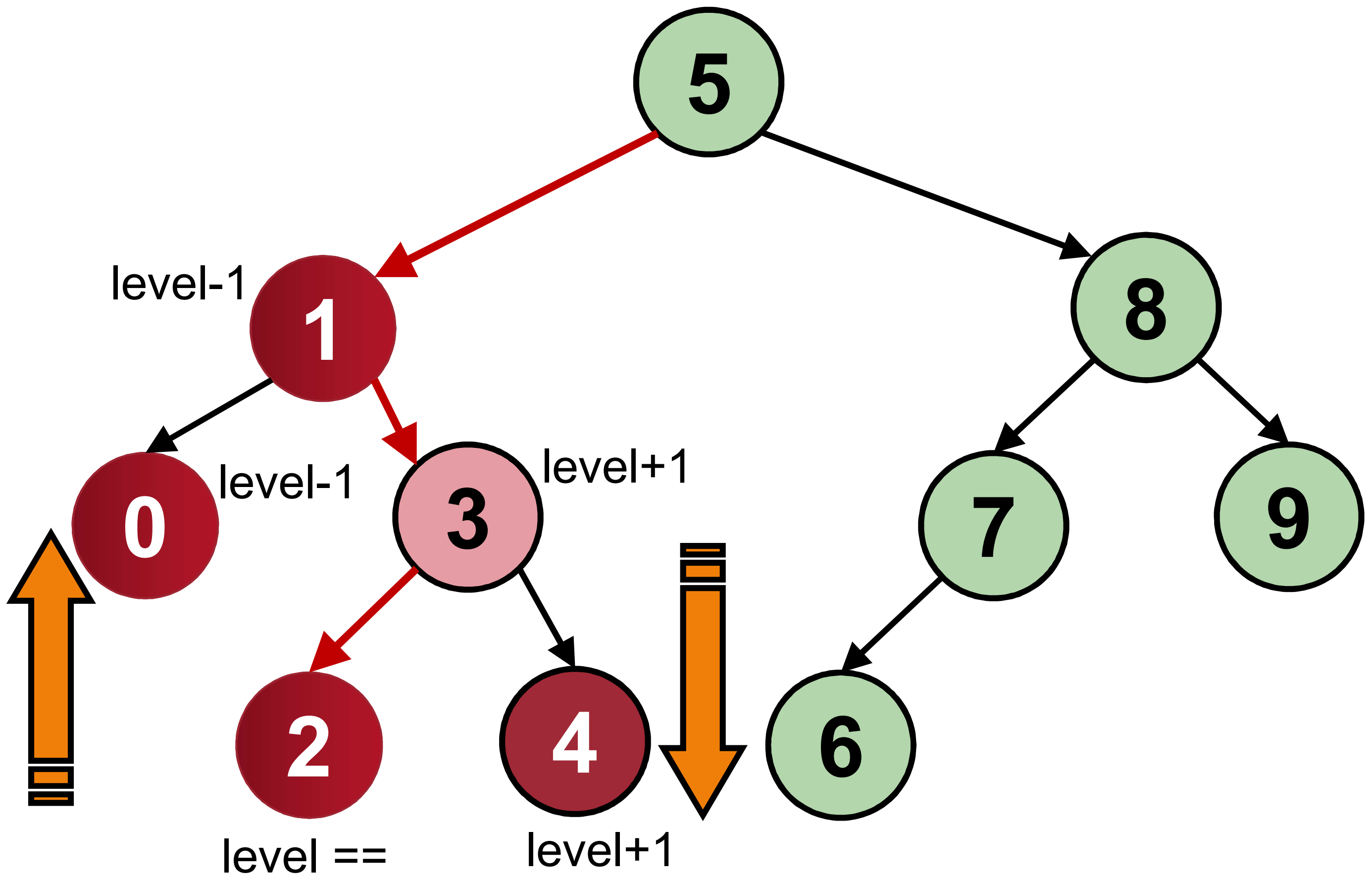
- The shape of a BST can be adjusted
  - Through a **series of rotations**
  - Note that rotation must **maintain BST property**

Right Rotation: Example



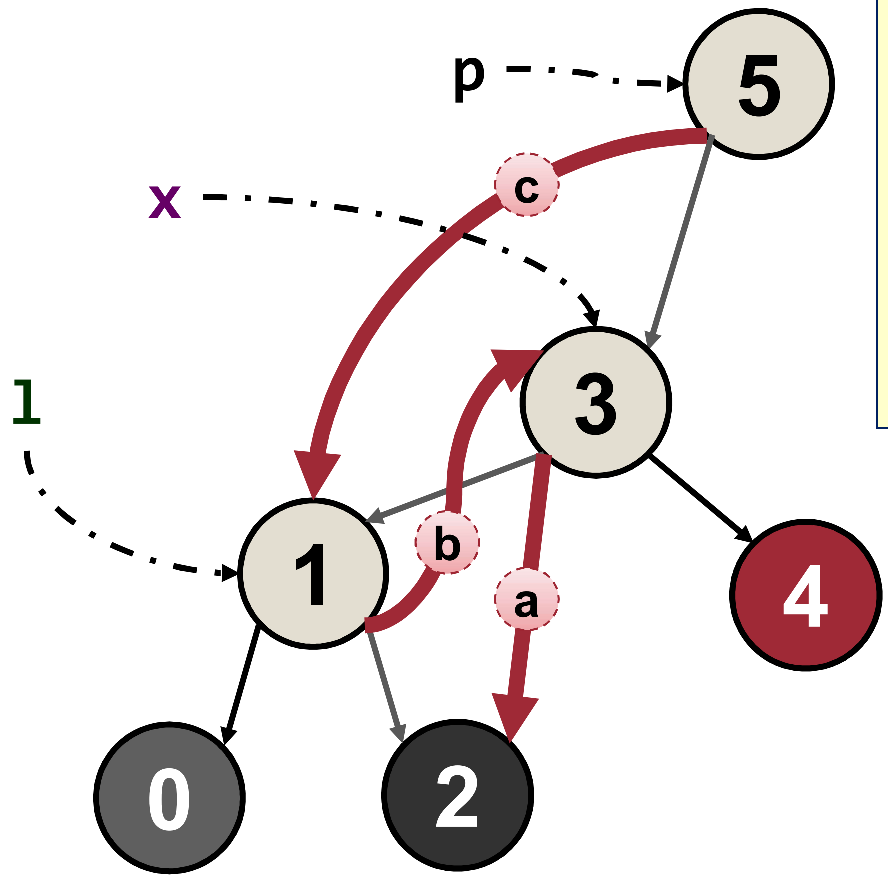
- Imagine rotating node "3" to the **right** (**clock-wise**)
  - How should we handle the child nodes?

AFTER Right Rotation: Example



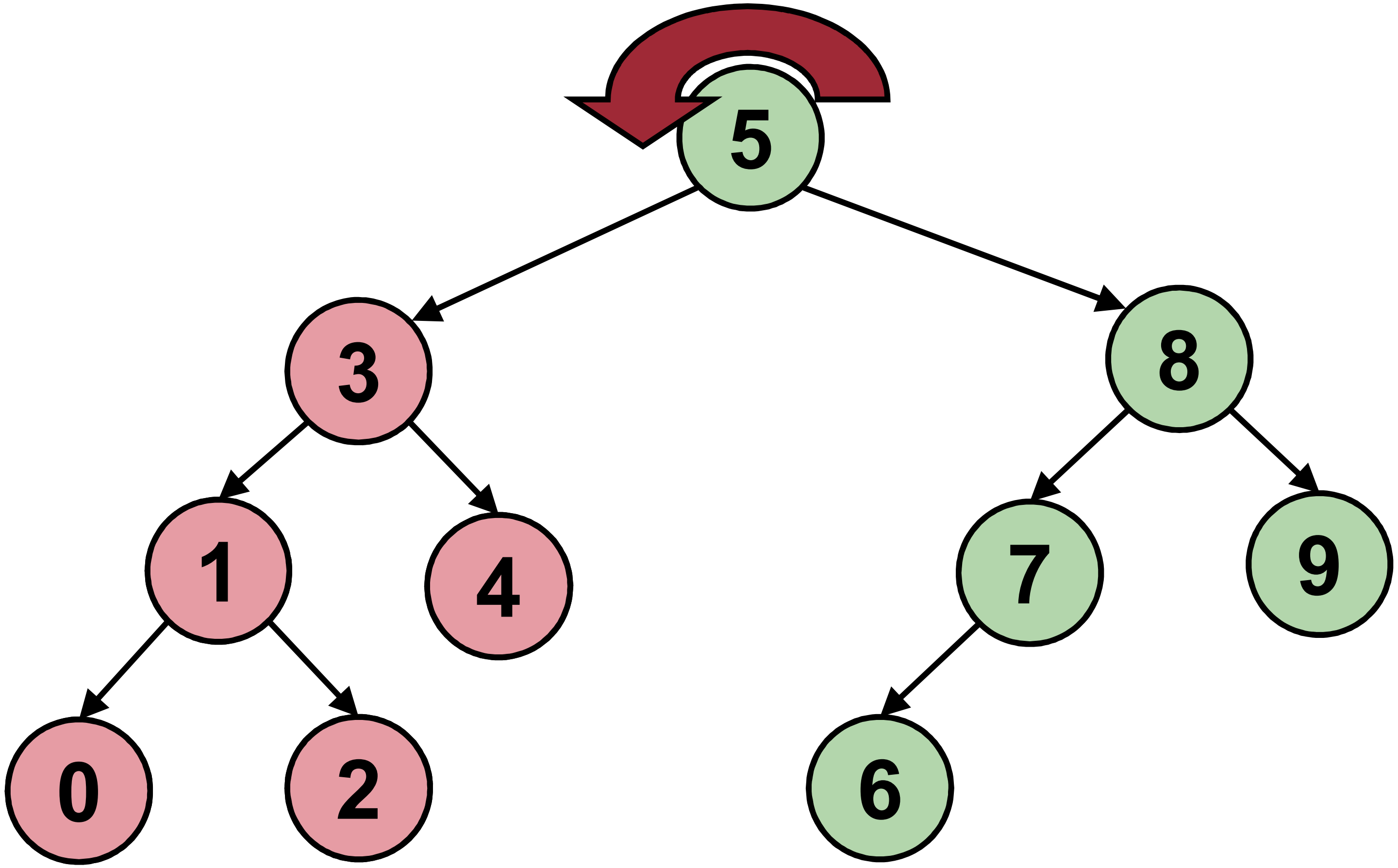
- Rotation changes the **heights** of some nodes
- Note the modified node references in **RED**

Right Rotation: **Psuedo-Code**



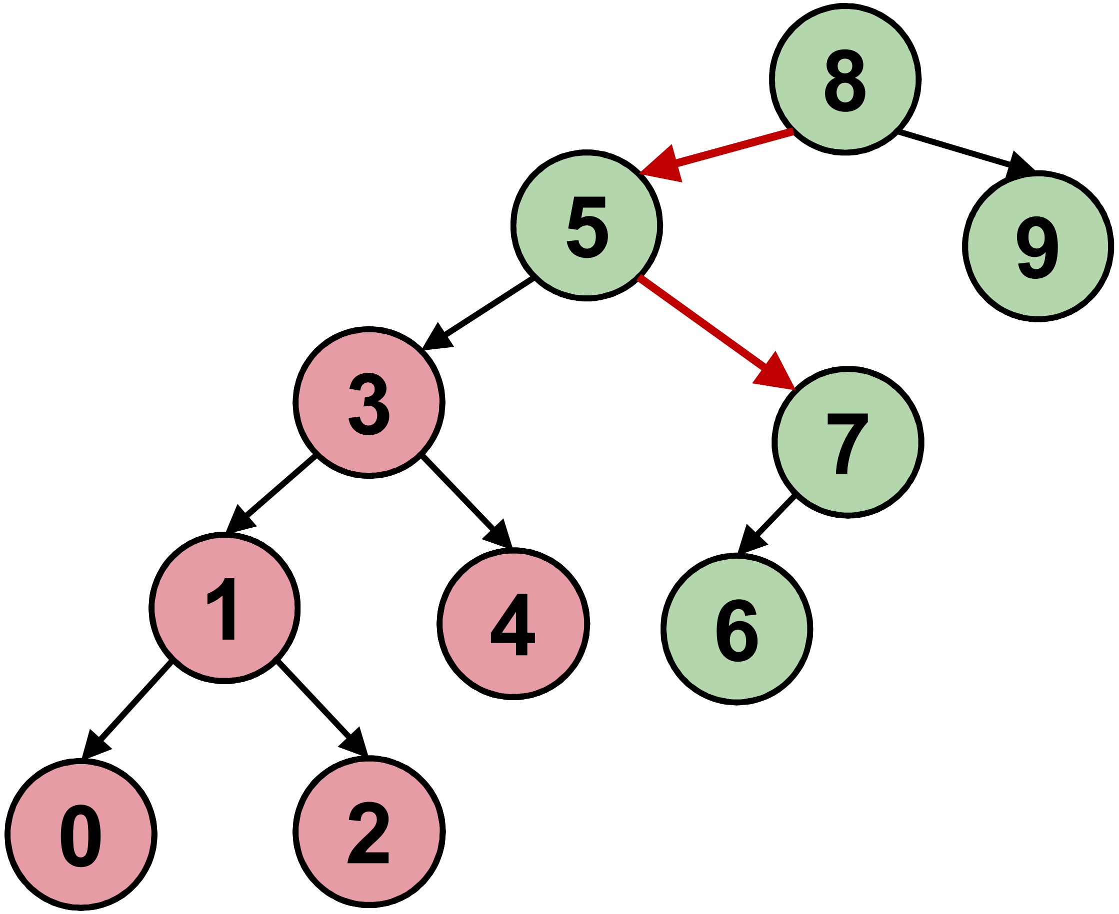

```
p.left = rotateRight( p.left )  c


def rotateRight( x ):
    l = x.left
    if l == None:    #cannot rotate!
        return x
    x.left = l.right  a
    l.right = x  b
    return l
```

Left Rotation: Example



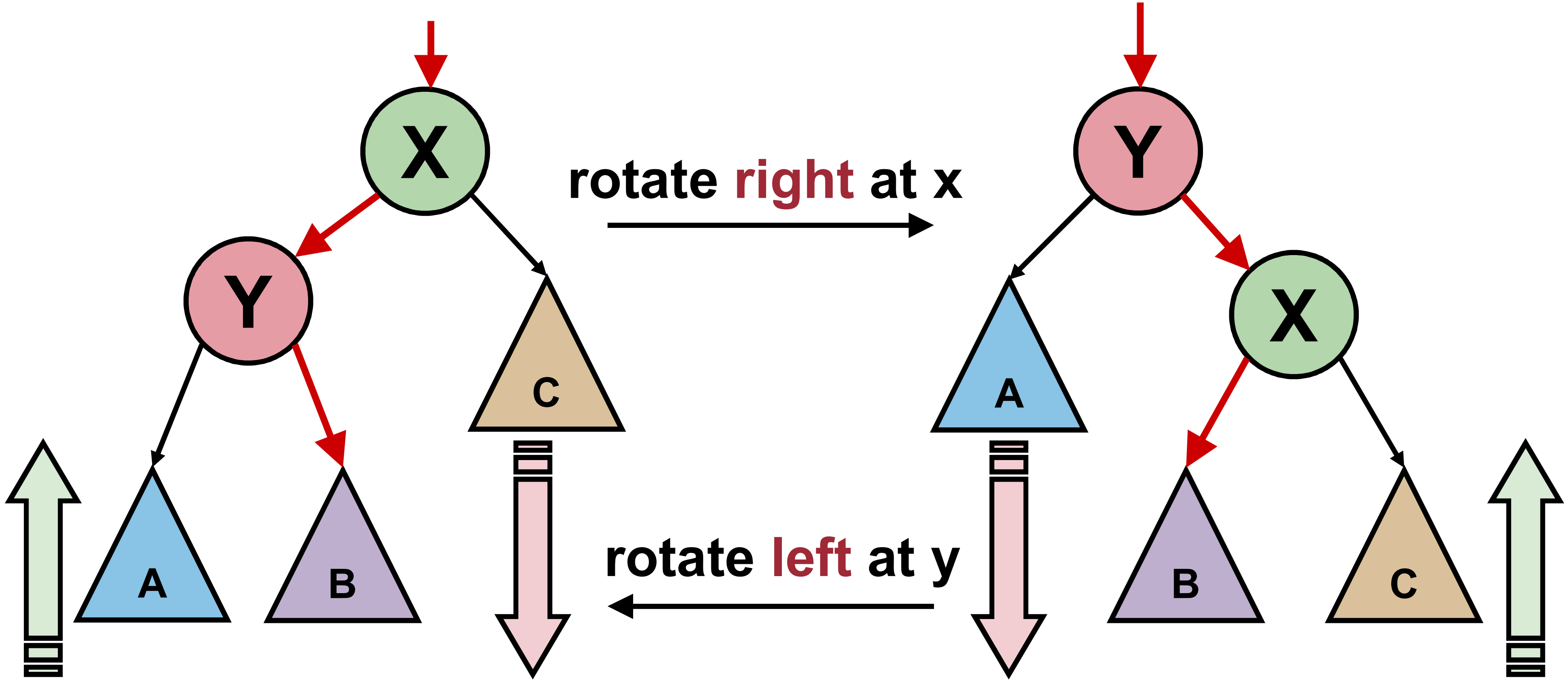
- Left rotation is similar (give it a try!)

AFTER Left Rotation: Example



- Figure out the pseudo-code for left rotation?

Rotation: Summary



- Rotation can change the tree shape while maintaining BST property
  - We'll use it heavily in maintaining AVL Tree next

MAJOR
AVL TREE OPERATIONS

AVL Tree Operations: **Overview**

- AVL Tree is also a BST

  ➔ The basic ideas of **insert**, **delete** and **search** are the same as BST

  - Additional task is to maintain AVL Tree property
    - Needed only for operations that **change the tree shape**

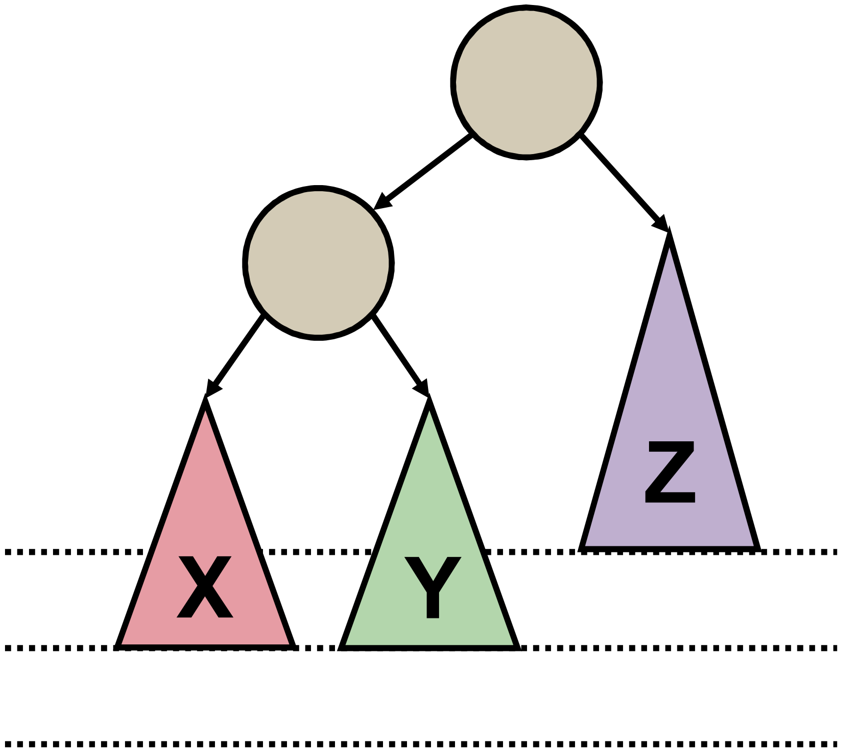
    ➔ e.g. **Search operation** is the exact same as BST tree's

- Basic idea **insertion / deletion**:
  1. Perform the operation normally
  2. Detect violation of AVL Tree property
  3. Perform rotation to restore AVL Tree property

AVL TREE INSERTION

AVL Insertion: Observations

- Given an AVL Tree ➔
  - Insertion into Z subtree never violates the AVL tree property
  - Insertion into X and Y subtree ***may*** cause a violation



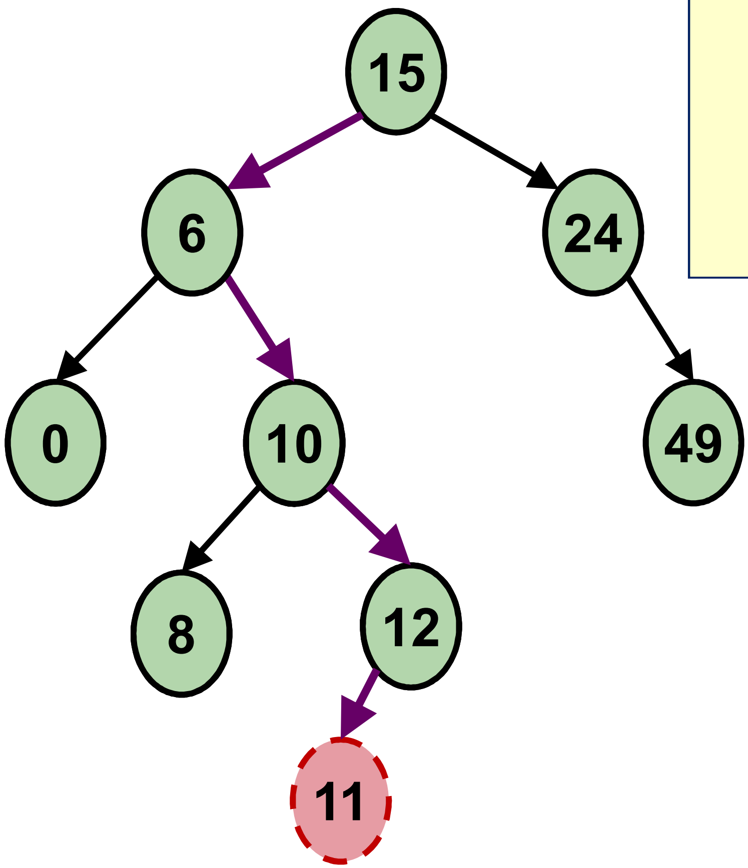
- Question:
  - If AVL Property is violated, at which node will we able to detect the issue?

AVL Property Violation

- BST insertion always **occurs at leaf node**
  - We have a path from root to the insertion point at the end of ***windup phase*** for the recursive insertion
  - Violation of AVL property **can only occur along this path**
- In the ***unwinding phase*** of the recursion
  - We will travel back from insertion point to root!
- Hence, we can:
  - **Add a check of AVL tree property** after insertion
  - **Correct violation** as we return to the parent caller

AVL Property Violation: **Detection**

```
def insert( T, key, data ):
    if T is empty:
        return TreeNode( key, data )
    if T➔key == key:
        Duplicate Key Error
    elif T➔key < key:
        T➔rightT = insert( T➔rightT, key, data )
    else:
        T➔leftT  = insert( T➔leftT, key, data )
    return T
```



- Note the path traversed during the insertion
- As we travel back from {11, 12, 10, 6, 15}, at which node can we detect the violation?

AVL Property Violation: **Detection**

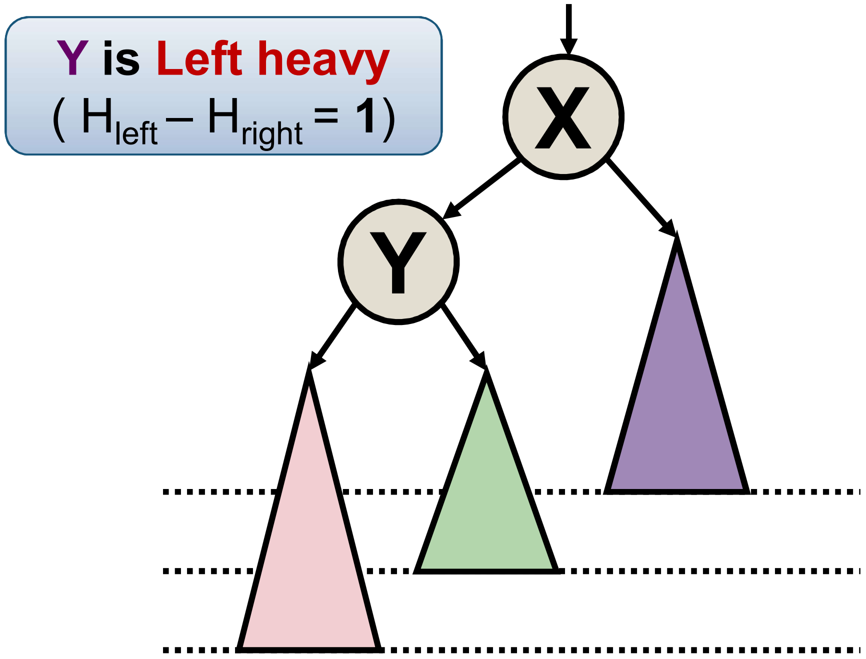
- For any node, height difference between left and right subtrees (**Balance Factor**) can be:

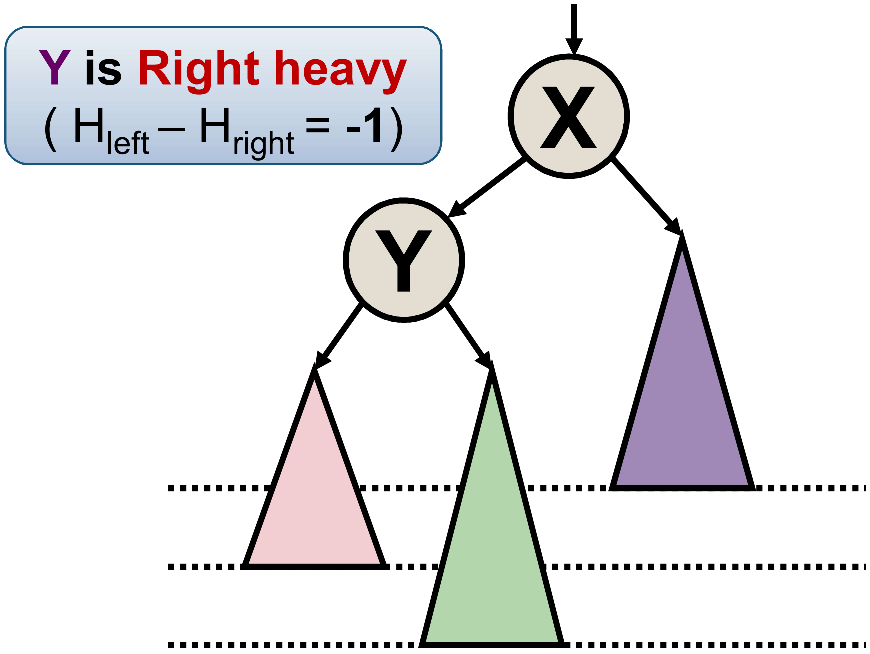
H_{left} - H_{right}	Remark
-2	**Right Skewed** (Need Correction!)
-1	**Right Heavy** (still ok)
0	**Balanced**
1	**Left Heavy** (still ok)
2	**Left Skewed** (Need Correction!)

- We'll use rotation to correct the skewed cases

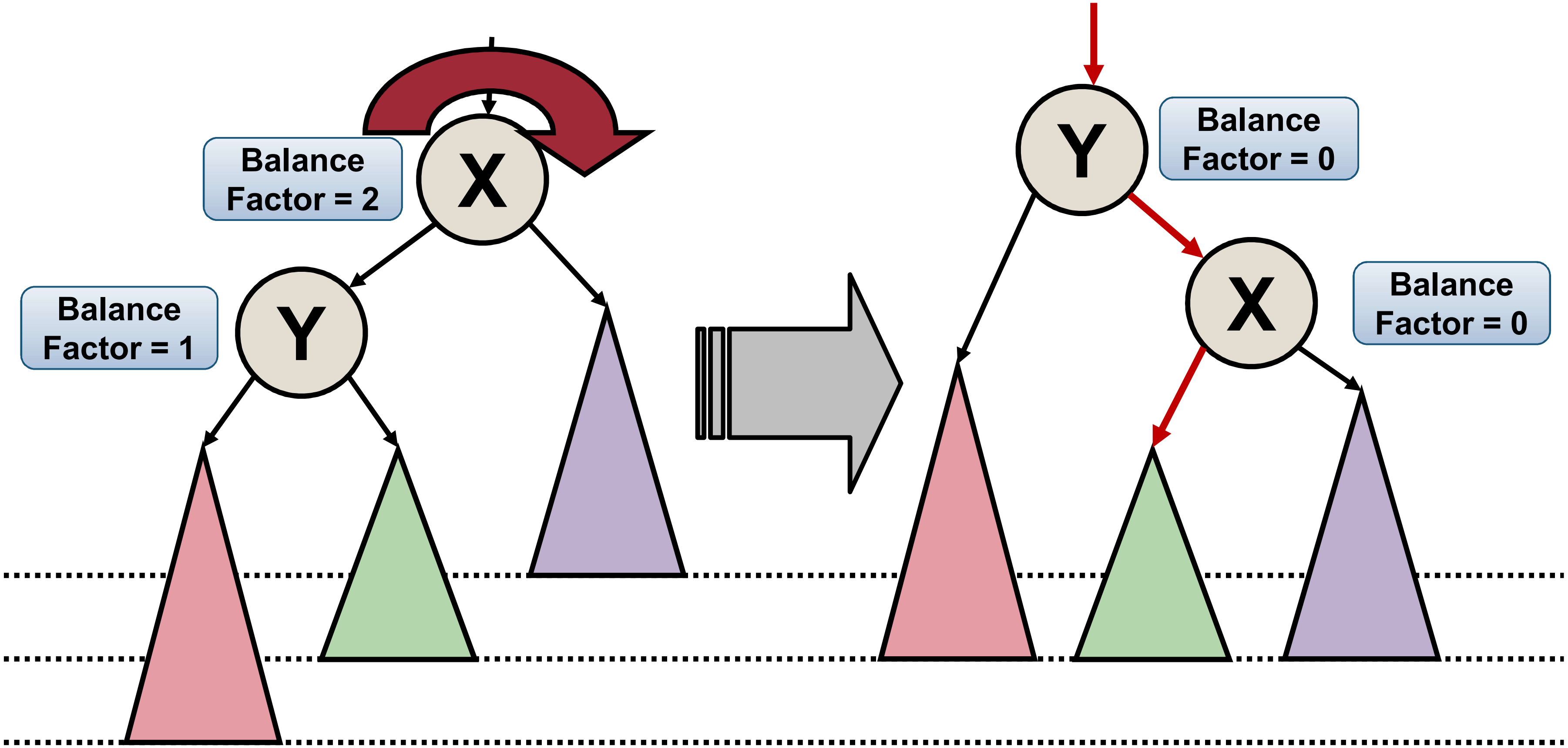
AVL Property Violation: **Cases**

- If node X is **left skewed** ($H_{left} - H_{right} = 2$):
  - The problem lies with the **left subtree of X**
  - The root of this left subtree, **Y** can have two possibilities:



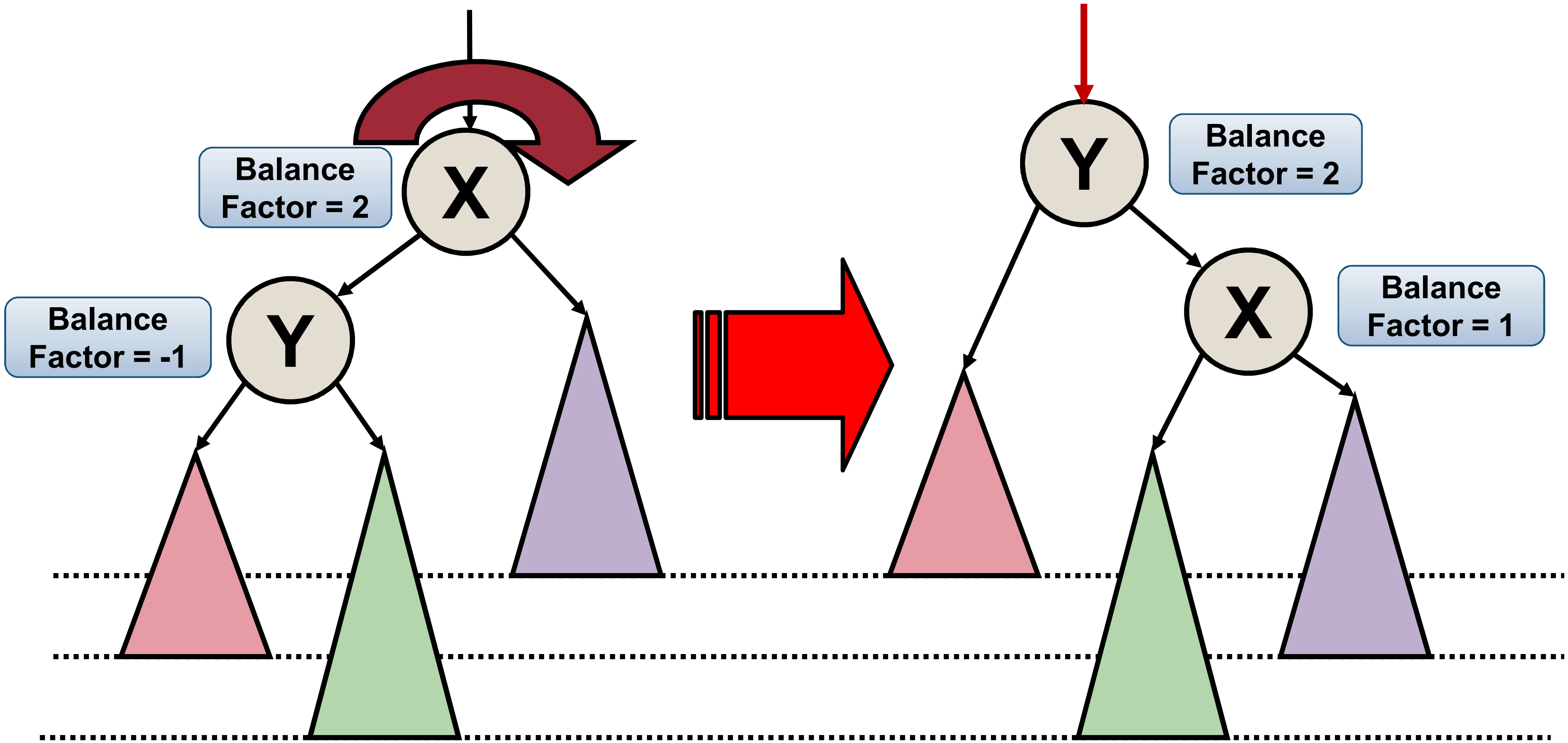


Case 1: **Left Skewed + Left Heavy**



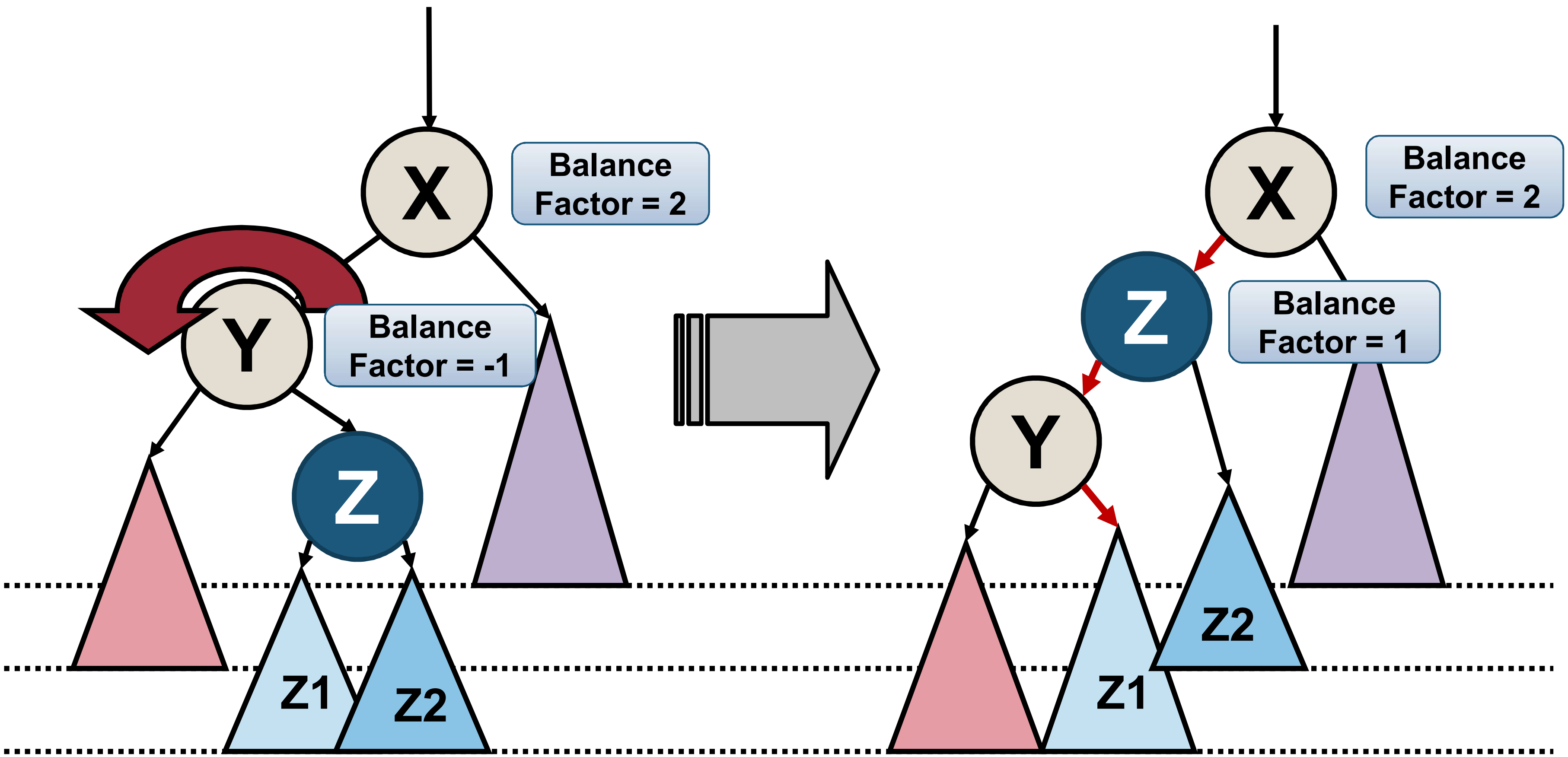
- A **Single Right Rotation** at X can restore the AVL tree property

Case 2: **Left Skewed + Right Heavy**



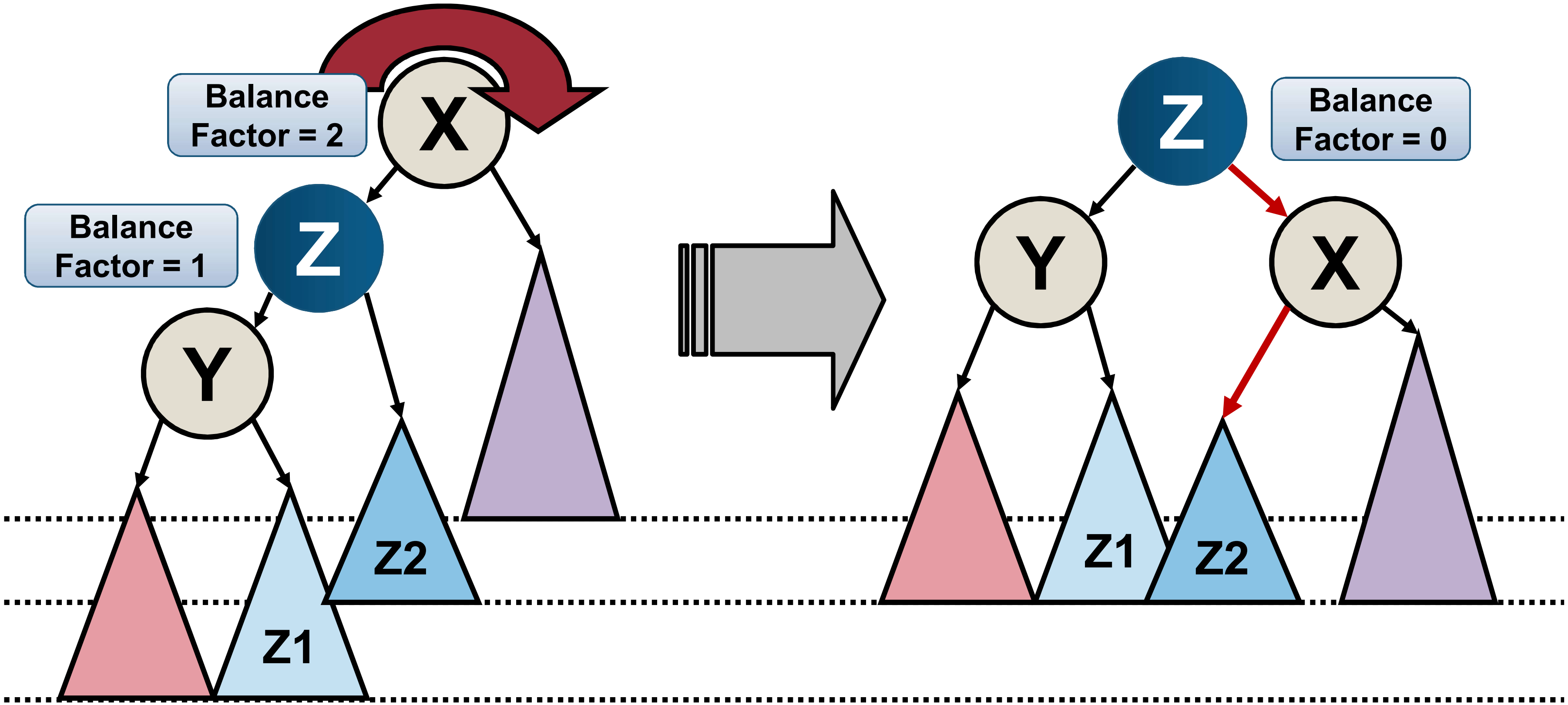
- A **Single Right Rotation** at X does **not** work!
- The height of the subtree remains unchanged

Case 2: **Left Skewed + Right Heavy**



- Let's do a **Single Left Rotation** at Y first
- Can you recognize the tree shape after rotation?

Case 2: **Become Case 1**



- Now, a **Single Right Rotation** at **X**

Left-Skewed Cases: Summary

Cases Left-Skewed at X Left Subtree of X is Y	Solution
Left Heavy at Y = **Outside Case** (Insertion on the left most branch) = **Zig-Zig Case** (Zig = left)	i. Right Rotation at **X**
Right Heavy at Y = **Inside Case** = **Zig-Zag Case** (Zag = right)	i. Left Rotation at Y ii. Right Rotation at **X**

- Other common name of the cases are also listed for your reference

Right-Skewed Cases: Summary

Cases Right-Skewed at X Right Subtree of X is Y	**Solution**
Right Heavy at Y = **Outside Case** (Insertion on the right most branch) = **Zag-Zag Case**	?
Left Heavy at Y = **Inside Case** = **Zag-Zig Case**	?

- Mirror Image for the right-skewed cases
- Can you figure out the correct solutions?

DESIGN CONSIDERATION & EXAMPLES

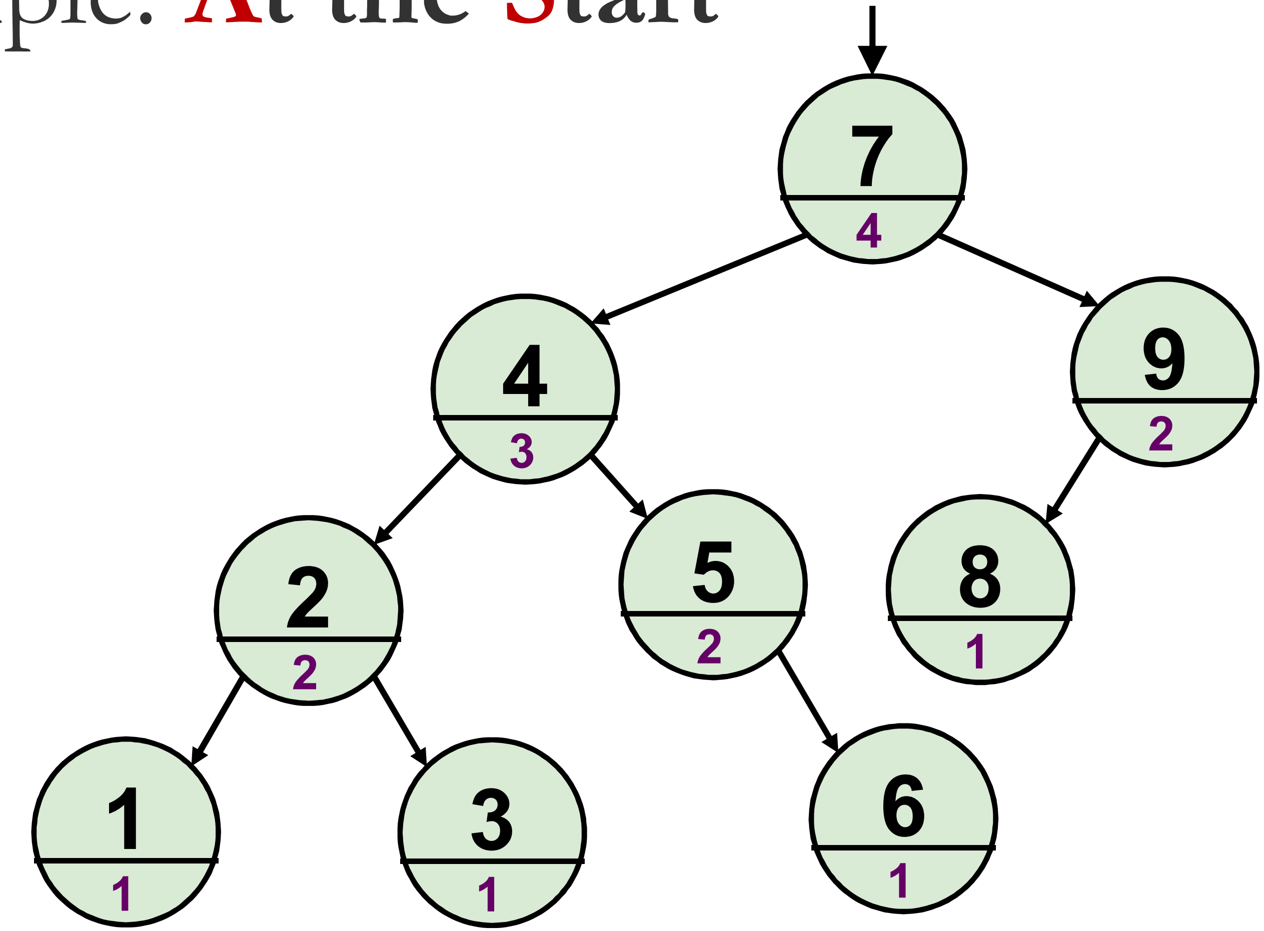
Design Considerations

- There are many ways to detect AVL Property violation:
  - Most solution requires additional attributes kept with each tree nodes
  - E.g. Keep the balance factor, tree height etc

- We will keep the "**tree height**" in each node:
  - i.e. the height of subtree at that node
    $\text{Height}_{\text{me}} = \max(\text{Height}_{\text{left}}, \text{Height}_{\text{right}}) + 1$

    ➔ The root node has the **tree height** for the entire tree

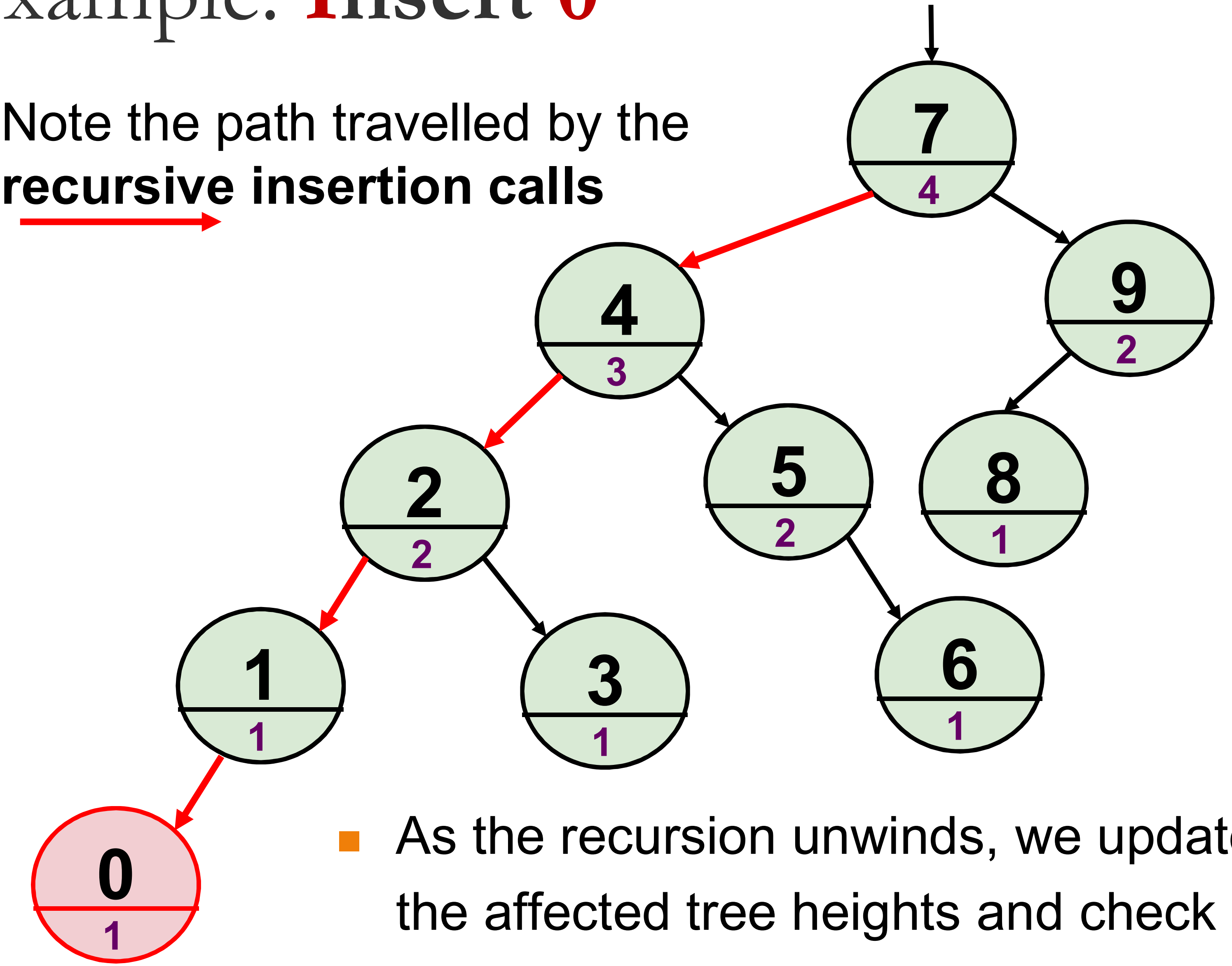
Example: At the Start



- The small number below the key in each node is the tree height
- We can use $|H_l - H_r|$ to find out the balance factor easily

Example: Insert 0

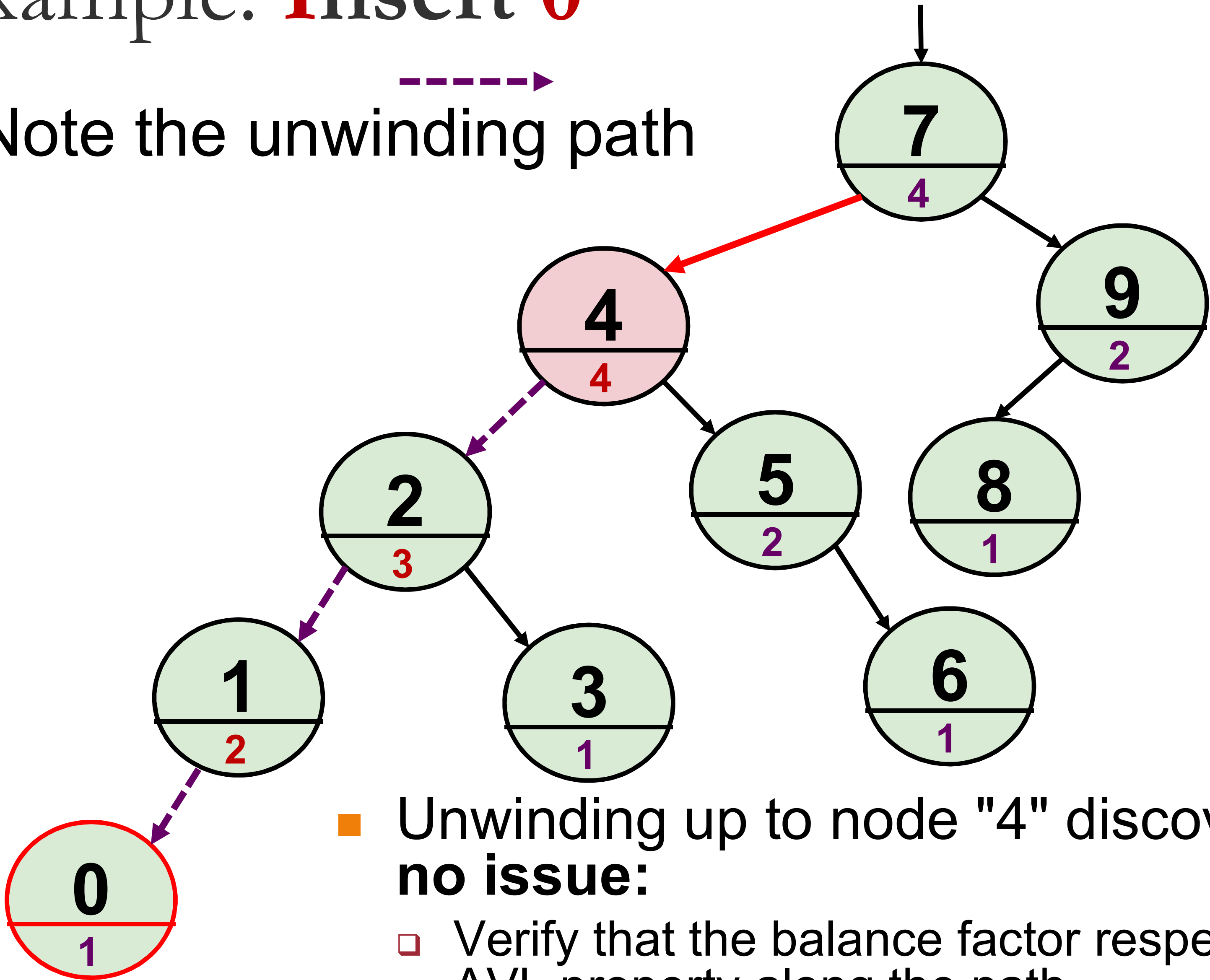
- Note the path travelled by the **recursive insertion calls**



- As the recursion unwinds, we update the affected tree heights and check the balance factor

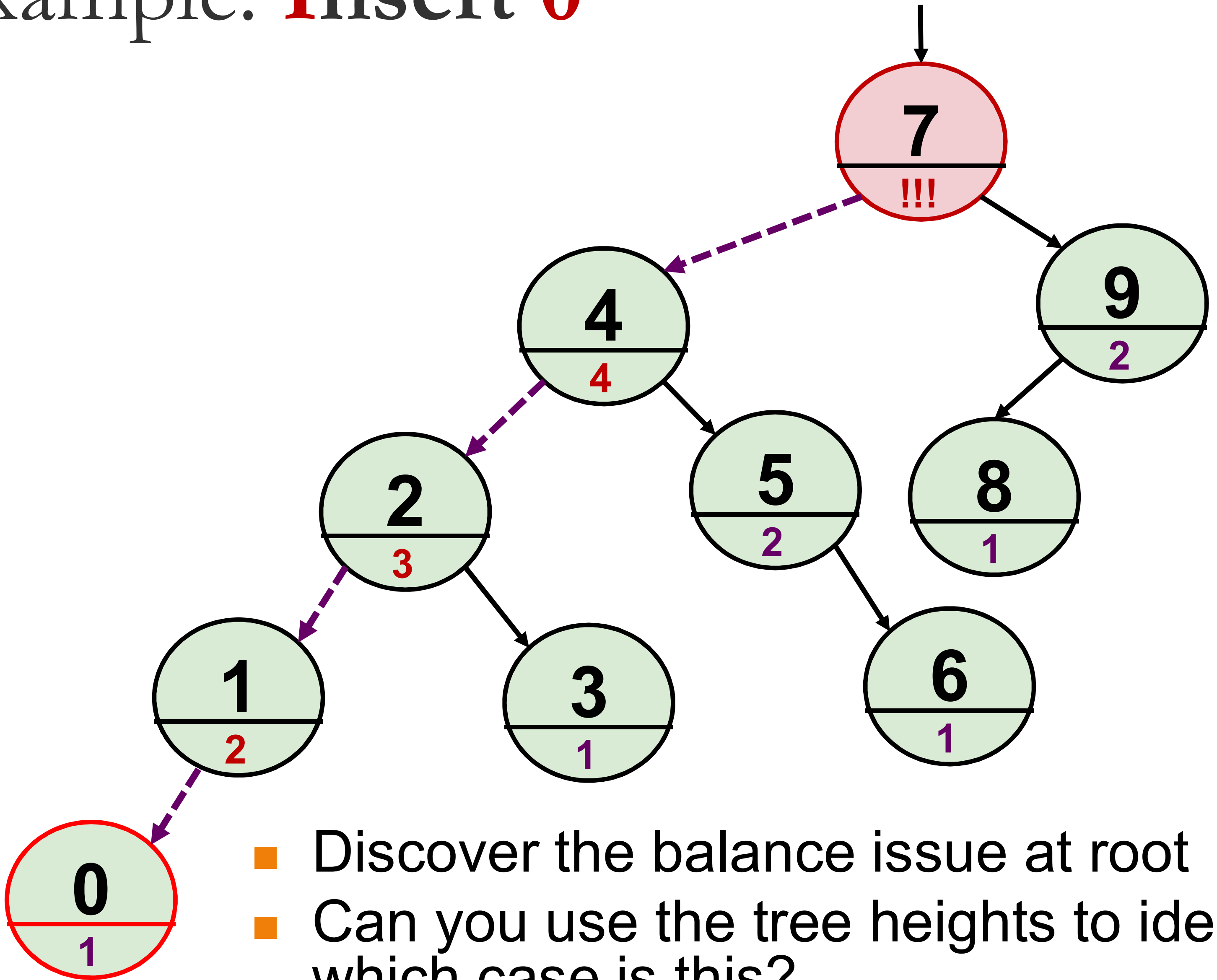
Example: Insert 0

- Note the unwinding path



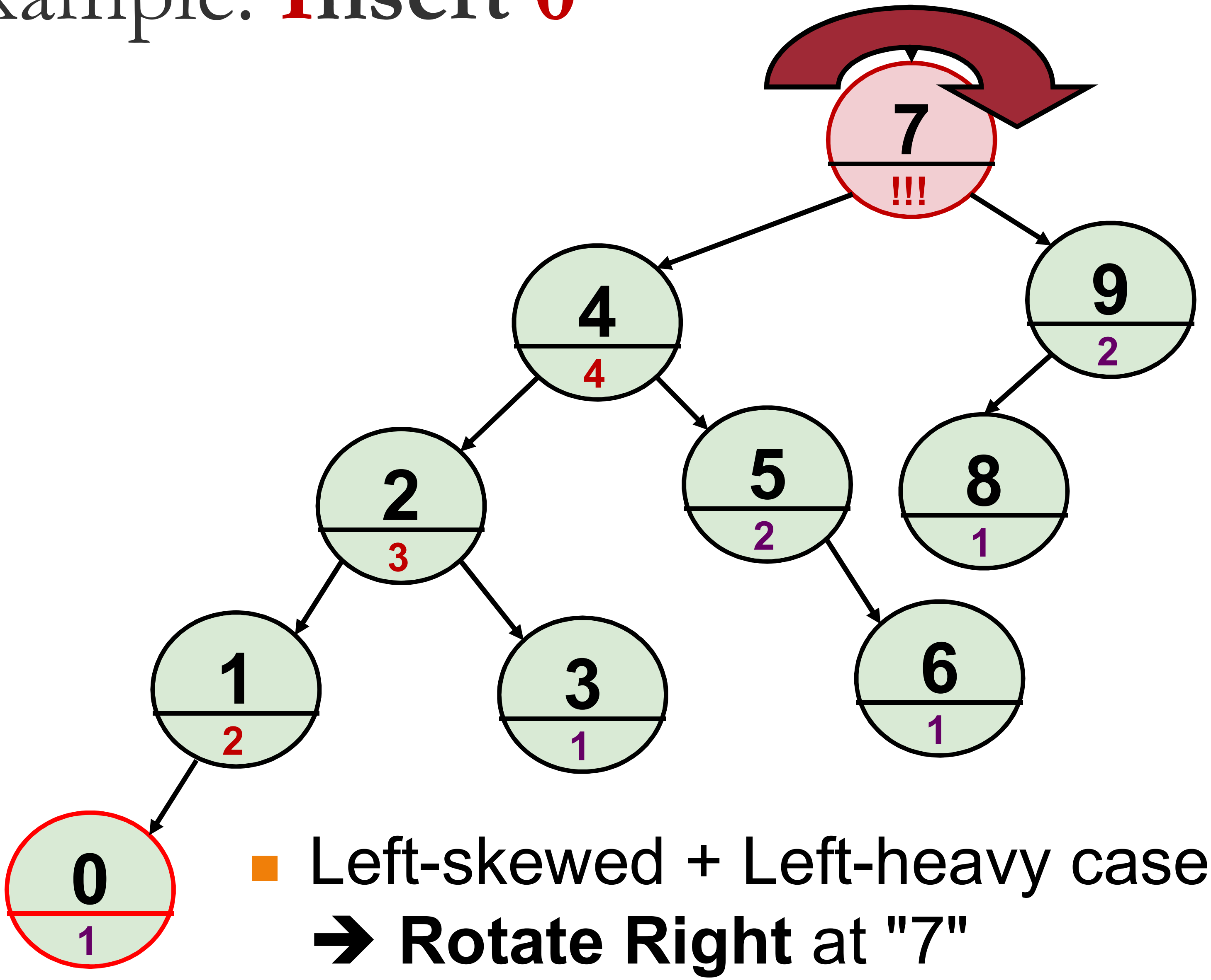
- Unwinding up to node "4" discover **no issue:**
  - Verify that the balance factor respects AVL property along the path

Example: Insert 0



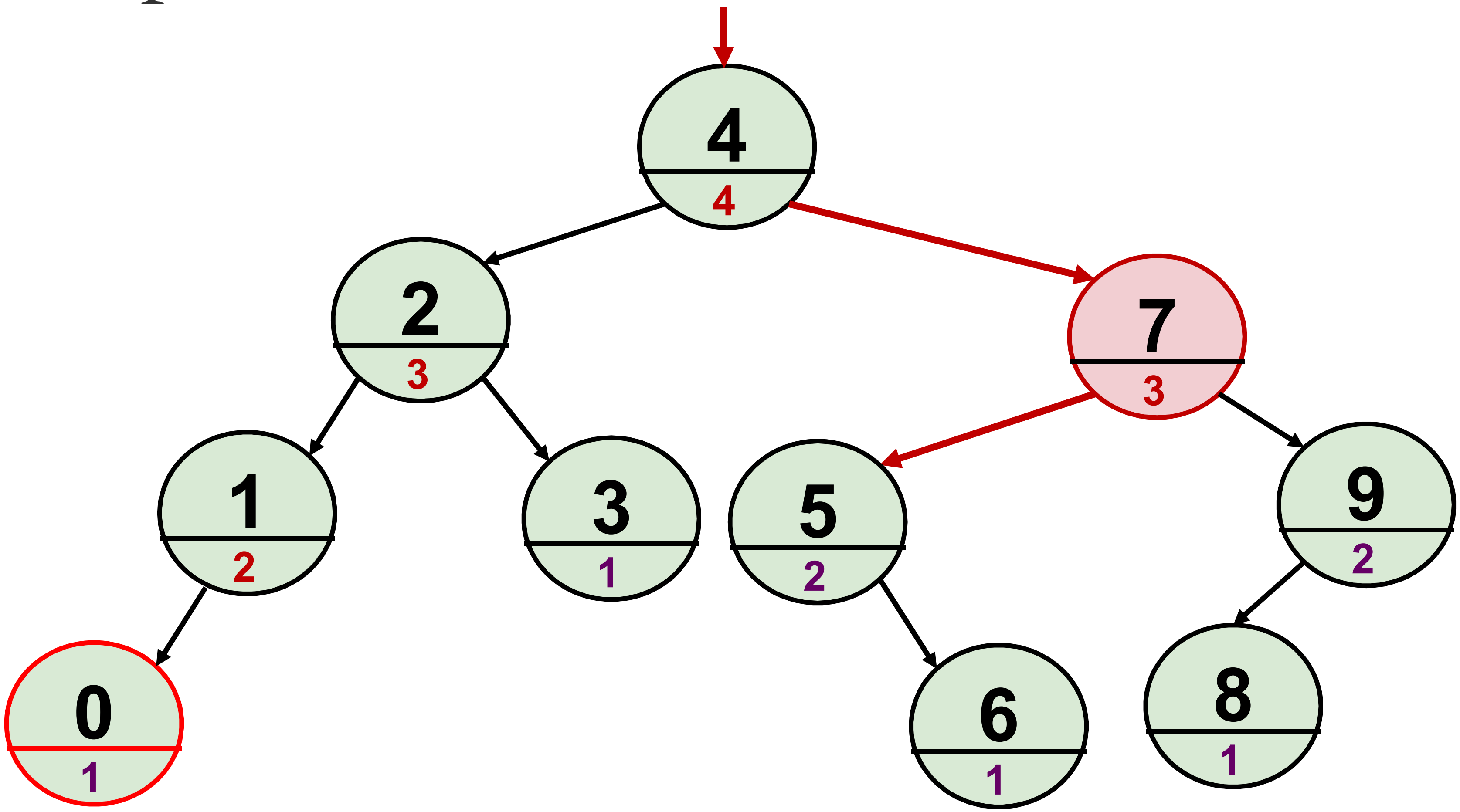
- Discover the balance issue at root
- Can you use the tree heights to identify which case is this?

Example: Insert 0



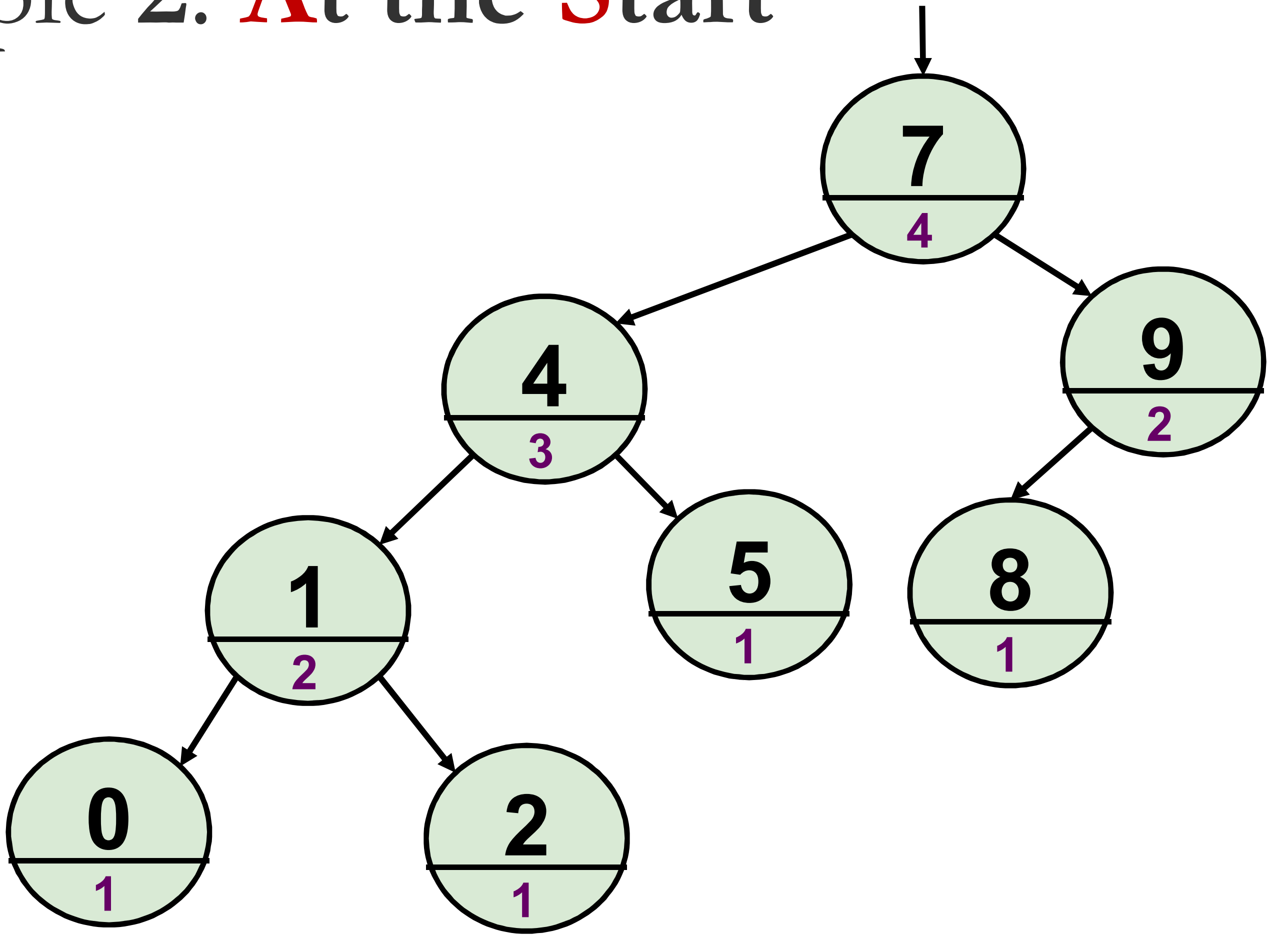
- Left-skewed + Left-heavy case
  ➔ **Rotate Right** at "7"

Example: Insert 0



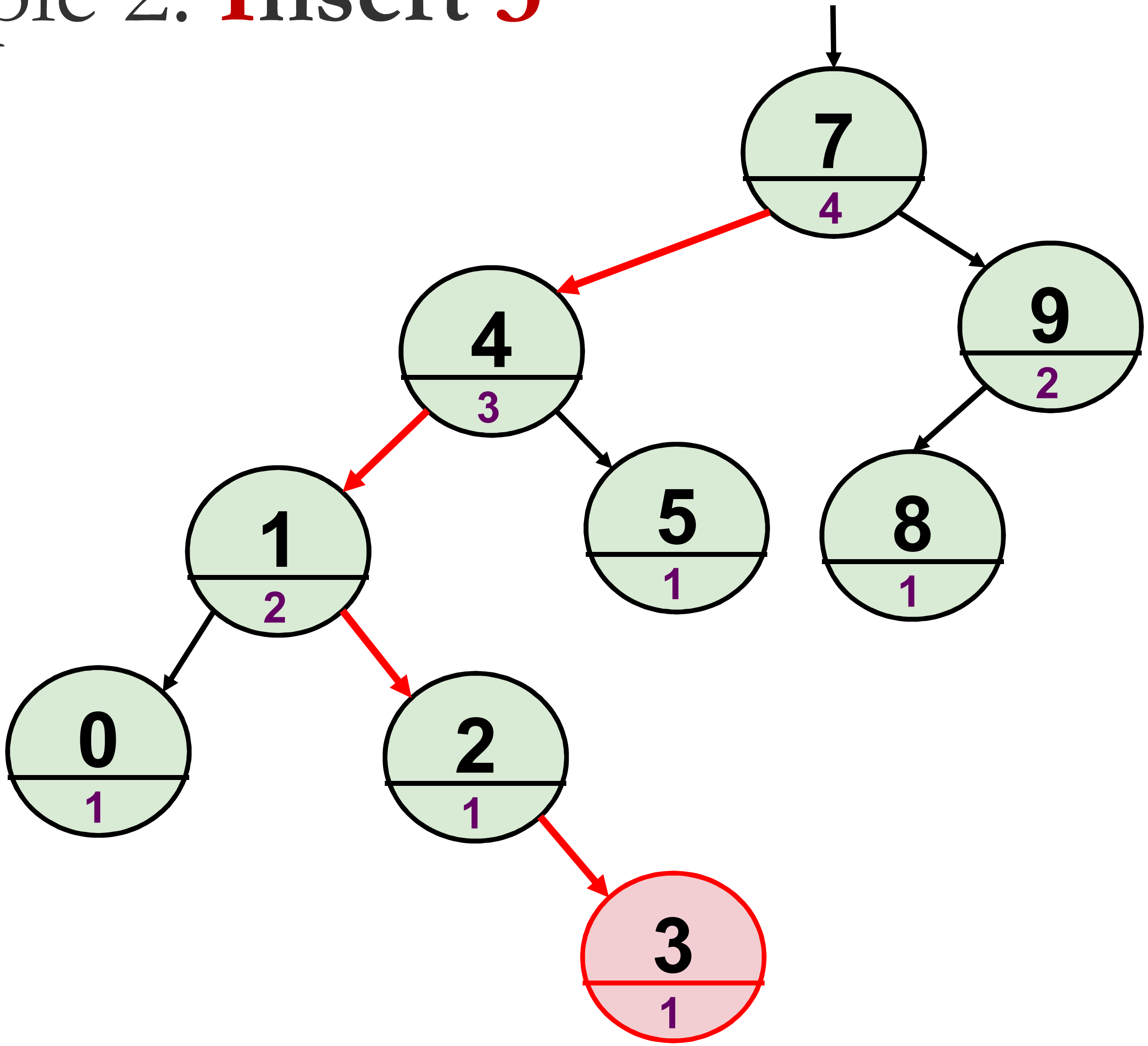
- Note the **changed references** and the **updated tree heights** for "7" and "4"

Example 2: At the Start

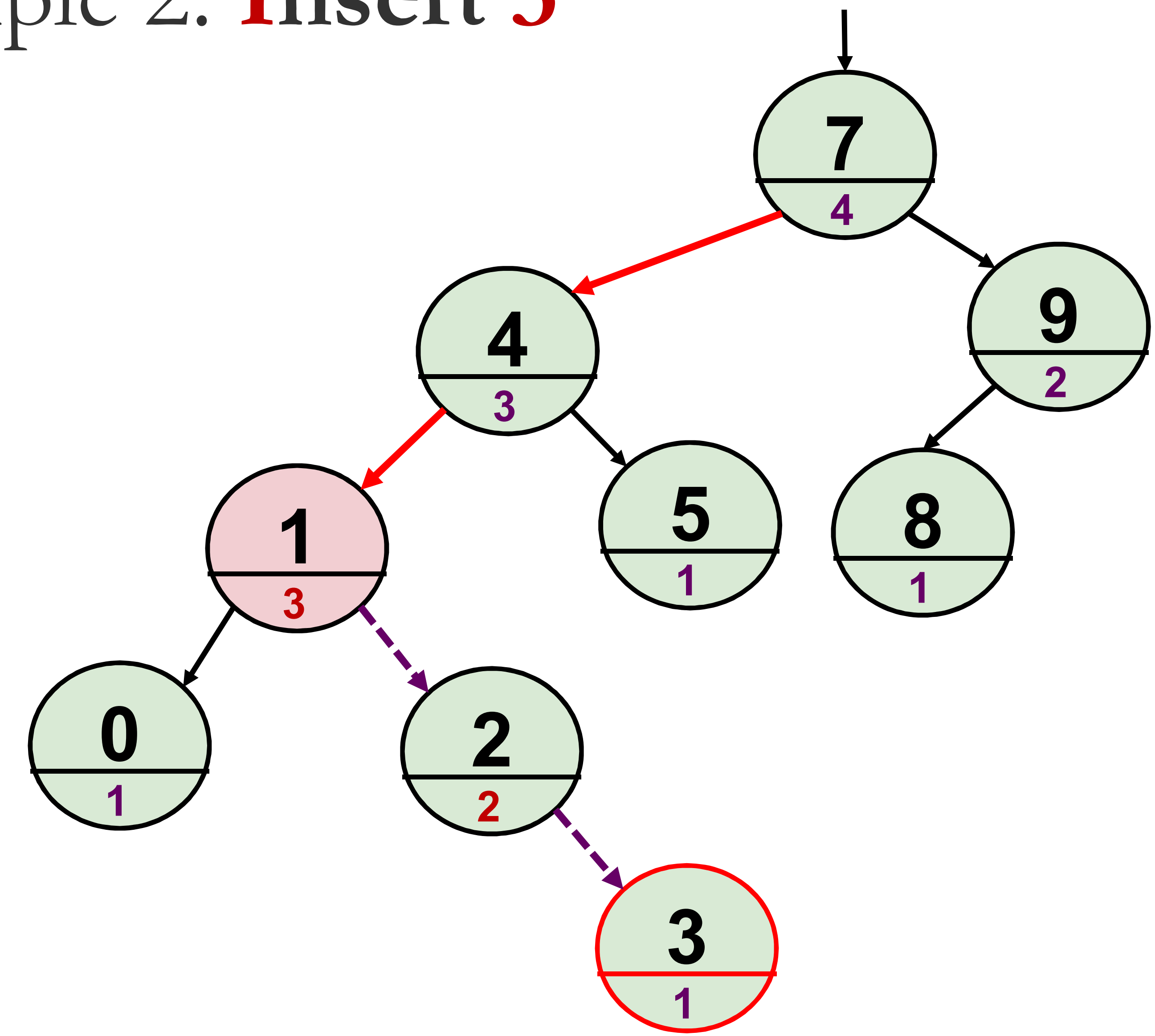


- Try insert **3** into the AVL Tree and perform the checking as shown before

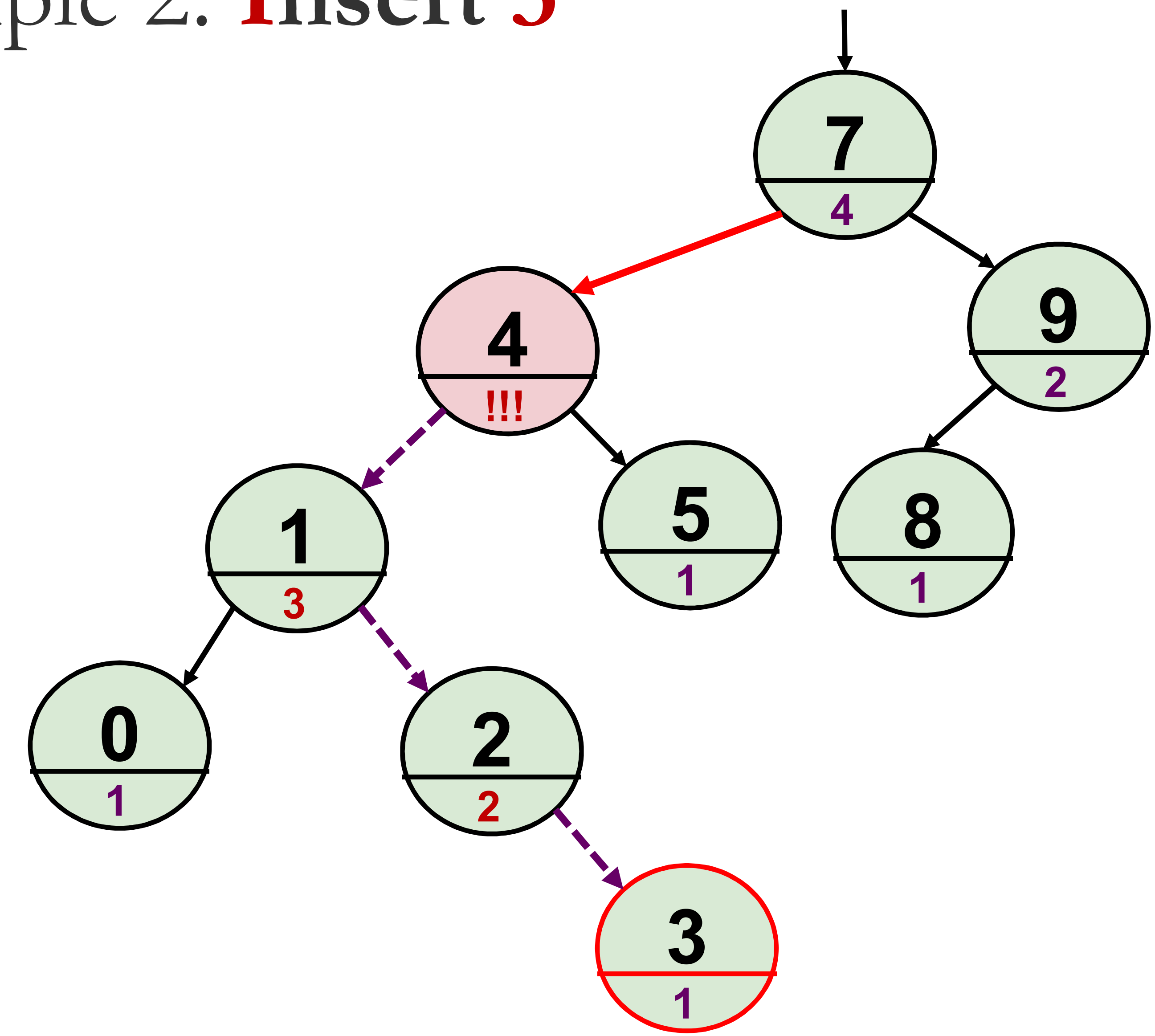
Example 2: Insert 3



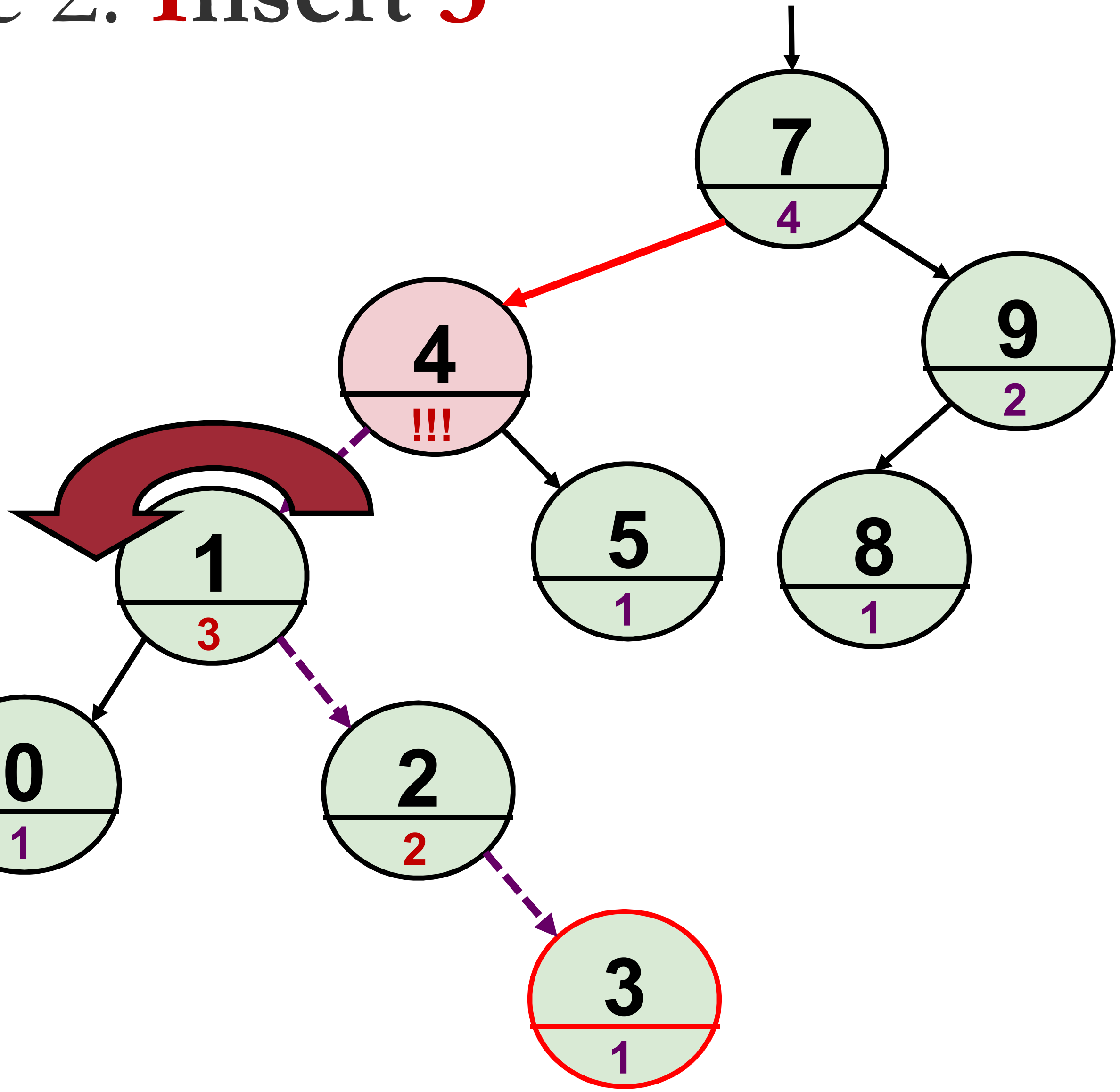
Example 2: Insert 3



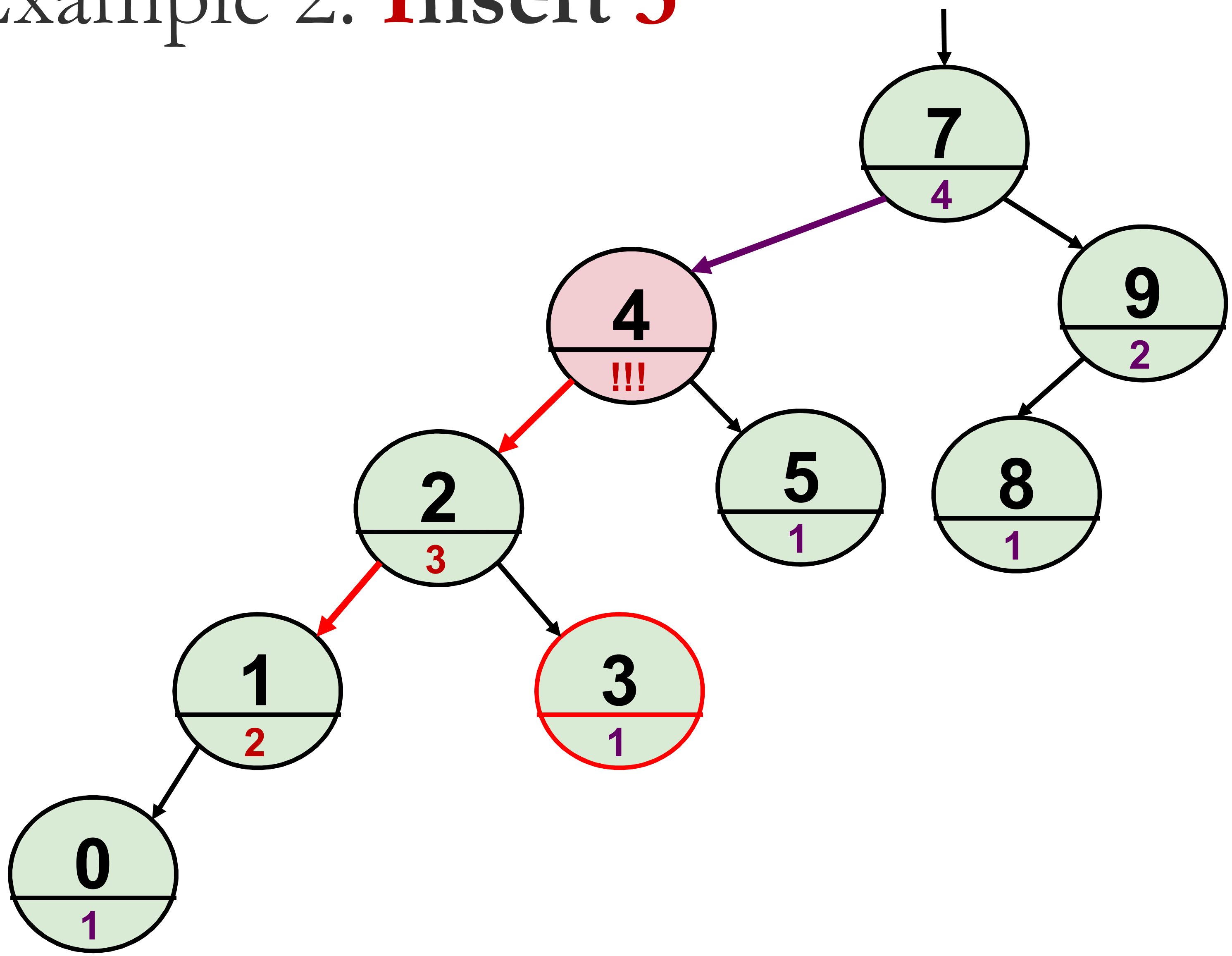
Example 2: Insert 3



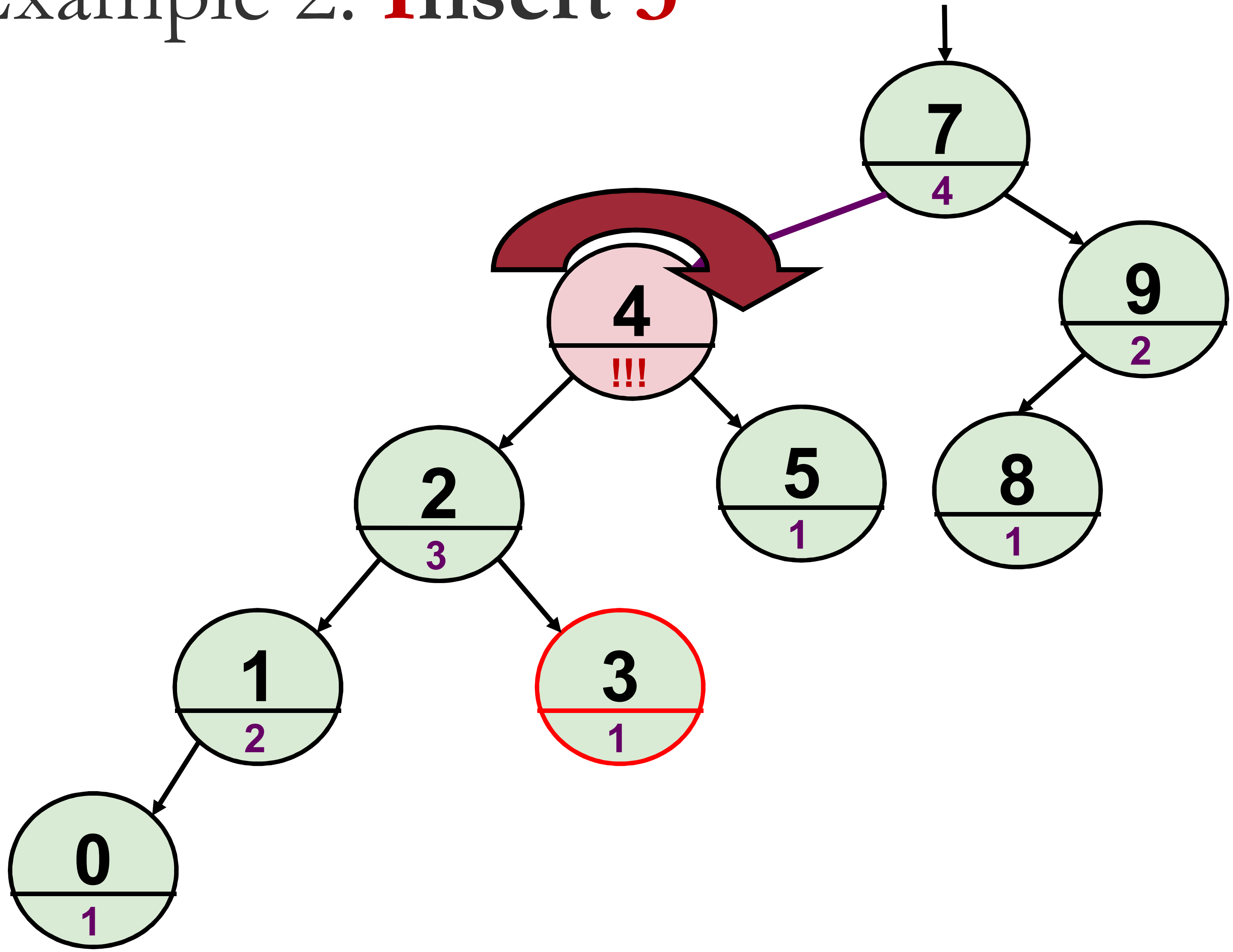
Example 2: Insert 3



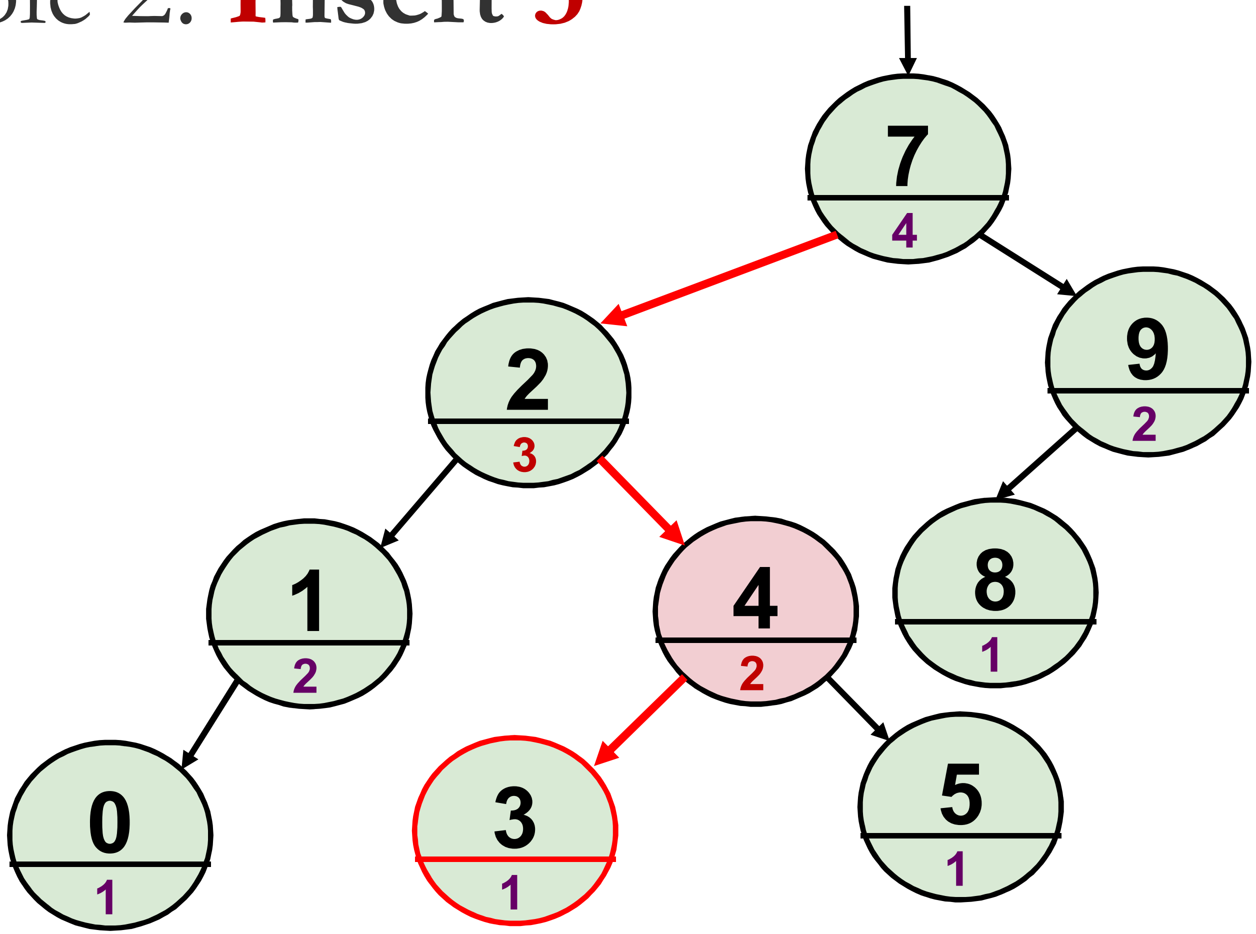
Example 2: Insert 3



Example 2: Insert 3



Example 2: Insert 3



Deletion: Can you generalize?

- Figure out the following:
  - What's the general flow of deletion in AVL Tree?
  - What are the cases in deletion?
    - Study one direction and use mirror generalization for the other
  - Use examples to illustrate the cases

- Important question:
  - How many rotation (at most) are needed for deletion?

Summary

- With the AVL tree property, major BST operations are now guaranteed at O(lg N)
- Rotation mechanism is general (i.e. can be used in BST as well)
  - AVL Tree use the rotations to ensure AVL Tree property after each insertion, rotation
- The AVL Rebalancing is performed along the insertion / deletion path

END