CSE 332 Autumn 2026
Lecture 9: AVL Trees, pt. 2

Ruth Anderson

http://www.cs.uw.edu/332

Announcements

- “Meet the staff” activity – Come meet us in office hours!
- Participate in Section, earn a late day (see Ed Board post)
- Exercise 2 Heaps: closes TONIGHT, Mon Oct 18
- Week 3 Survey in canvas: Due TOMORROW, Tuesday Oct 20
- Exercise 3 Recurrences: Due Fri Oct 23
- Exercise 4 AVL Trees: Due Friday Oct 30
- Midterm: 5-6pm Monday November 9

AVL Tree: A Balanced BST, a Dictionary

- A Binary Search tree that maintains that the left and right subtrees of every node have heights that differ by at most one.
  - height of left subtree and height of right subtree differ by at most 1
  - Not too weak (ensures trees are short)
  - Not too strong (works for any number of nodes)

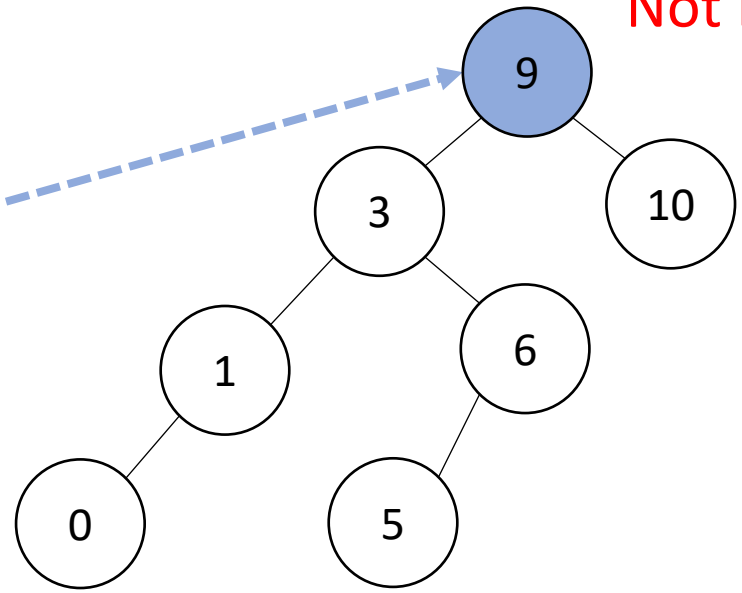
- Idea of AVL Tree:
  - When you insert/delete nodes, if tree is “out of balance” then modify the tree
  - Modification = “rotation”

Is it an AVL Tree? (Answers)

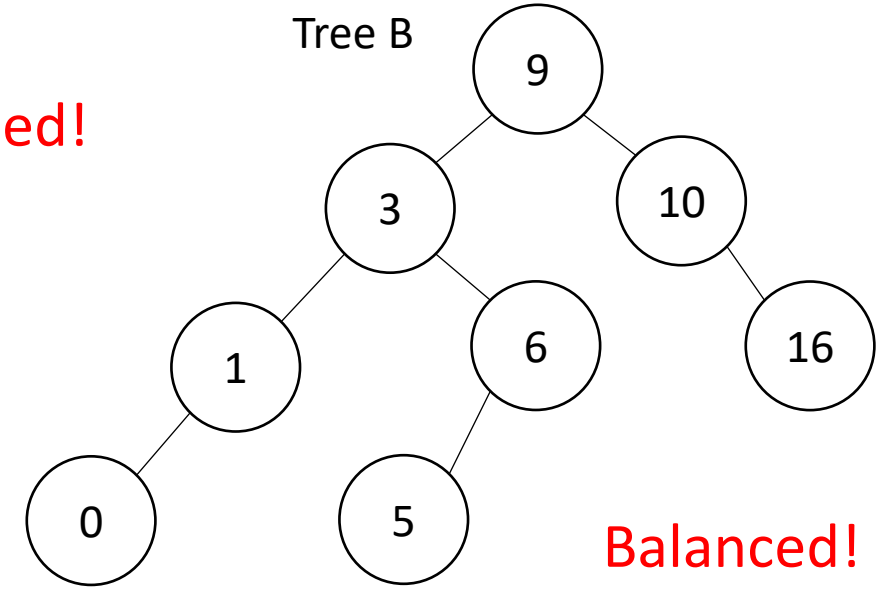
Tree A

Not Balanced!

"Problem" Node
Its children's heights differ by more than 1

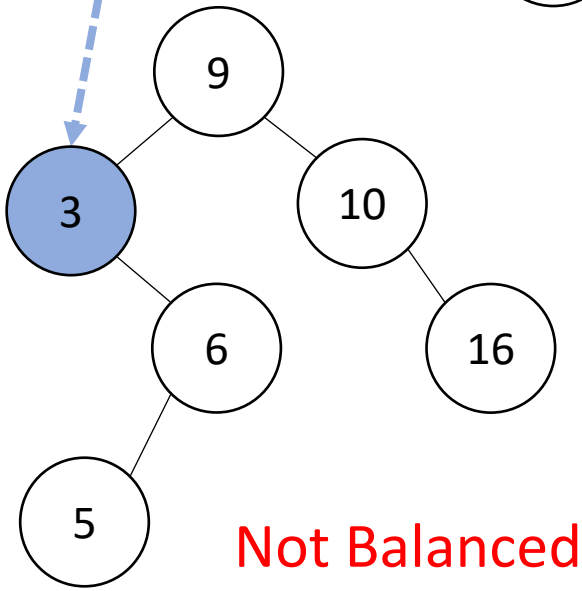


Tree B



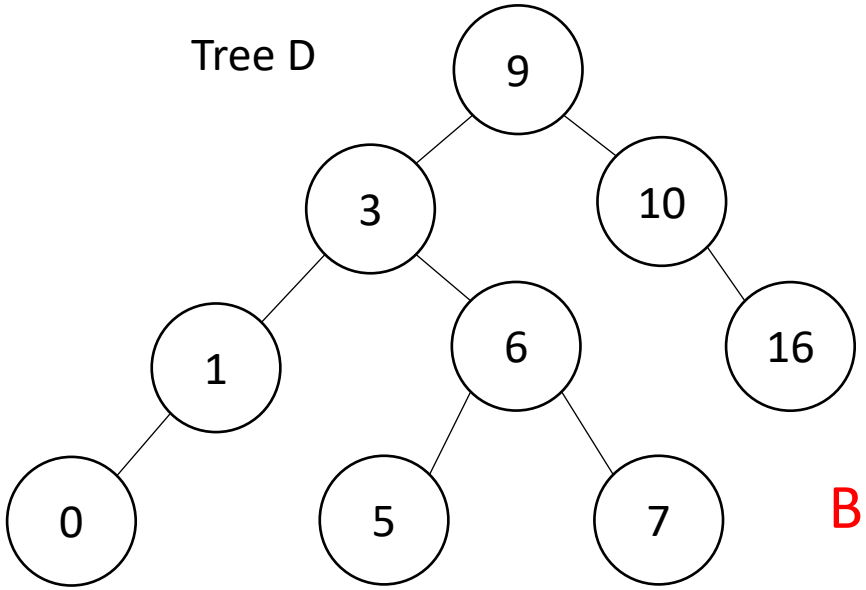
Balanced!

Tree C



Not Balanced!

Tree D

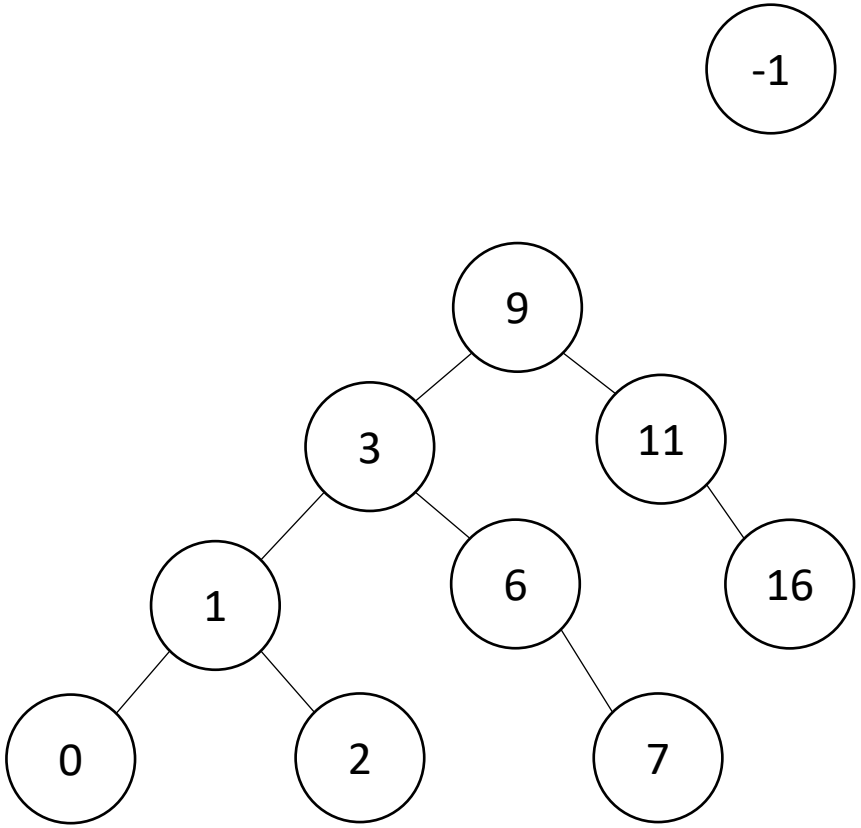


Balanced!

Inserting into an AVL Tree

- Starts out the same way as BST:
  - “Find” where the new node should go
  - Put it in the correct place (it will be a leaf)
- Next check the balance
  - If the tree is still balanced, you’re done!
  - Otherwise we need to do rotations

Insert Example (Insert -1)



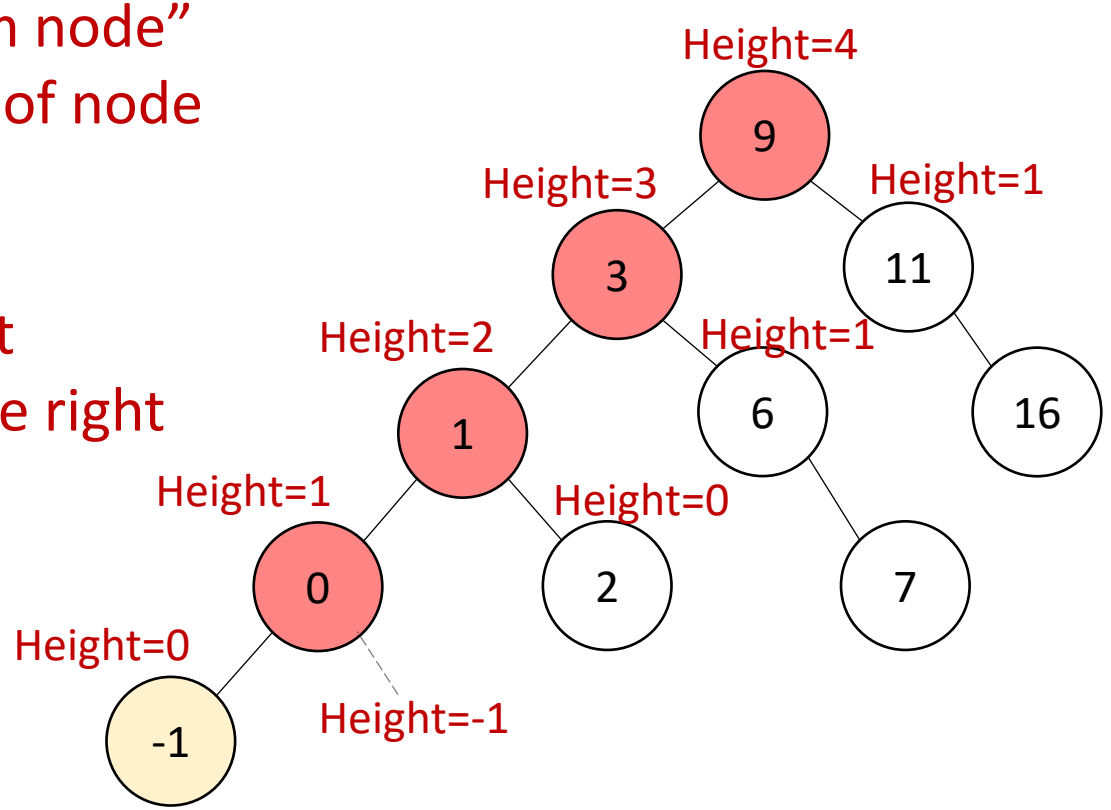
Insert -1, BST Insert Step

Not Balanced!

Node 9 is the “problem node”

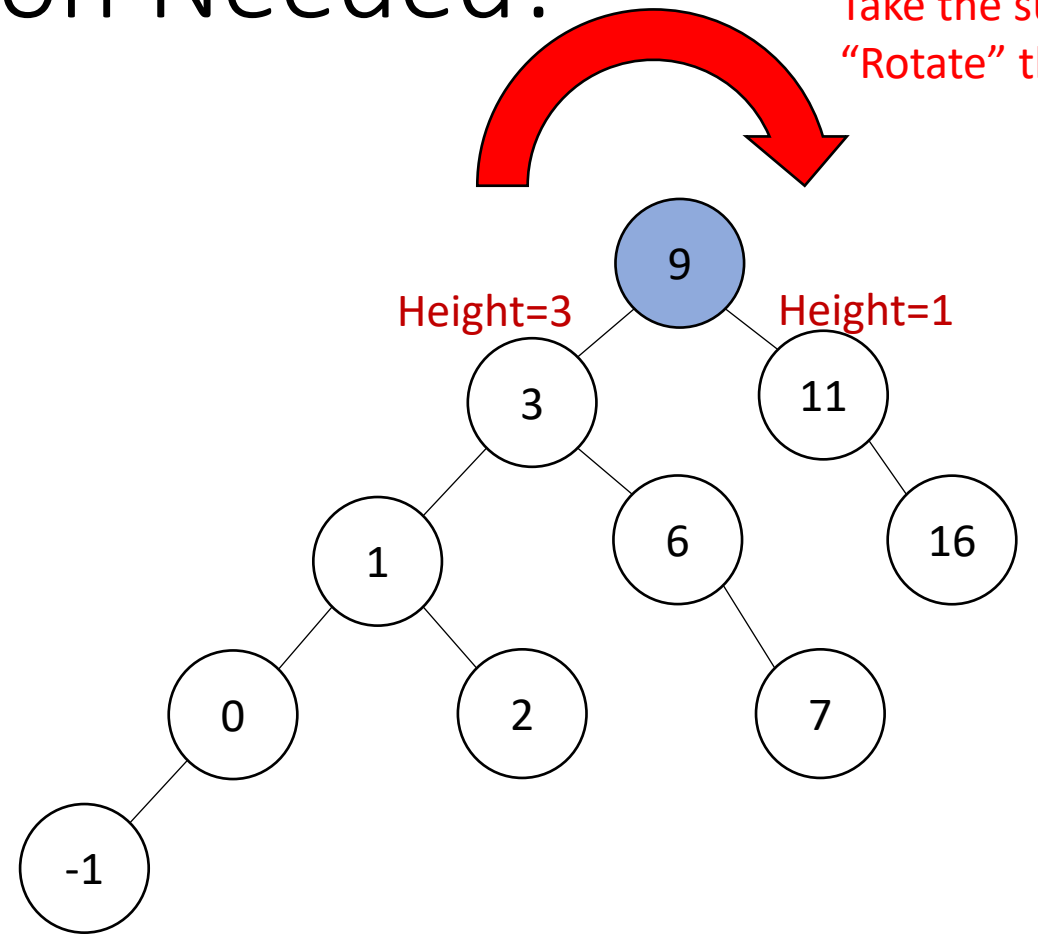
Left and right children of node 9 are different by 2.

Idea: “shorten” the left subtree, “lengthen” the right



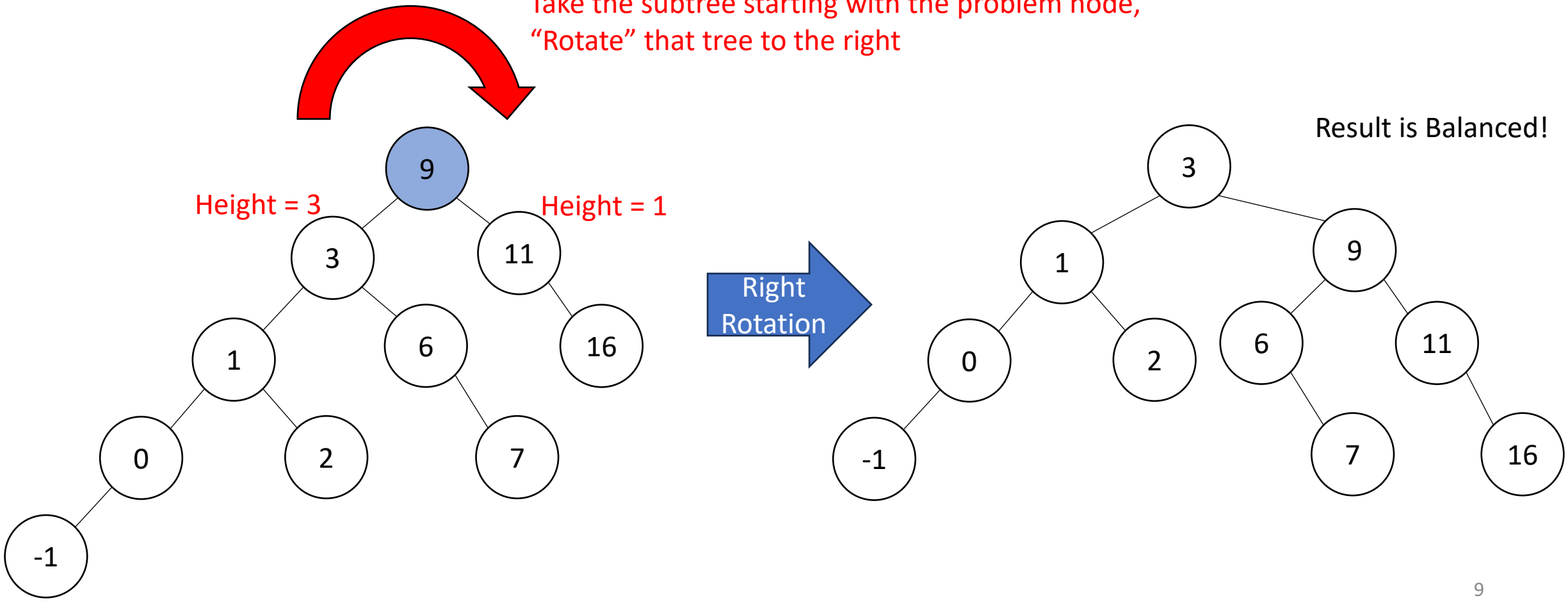
Right Rotation Needed!

Solution:
Take the subtree starting with the problem node,
“Rotate” that tree to the right

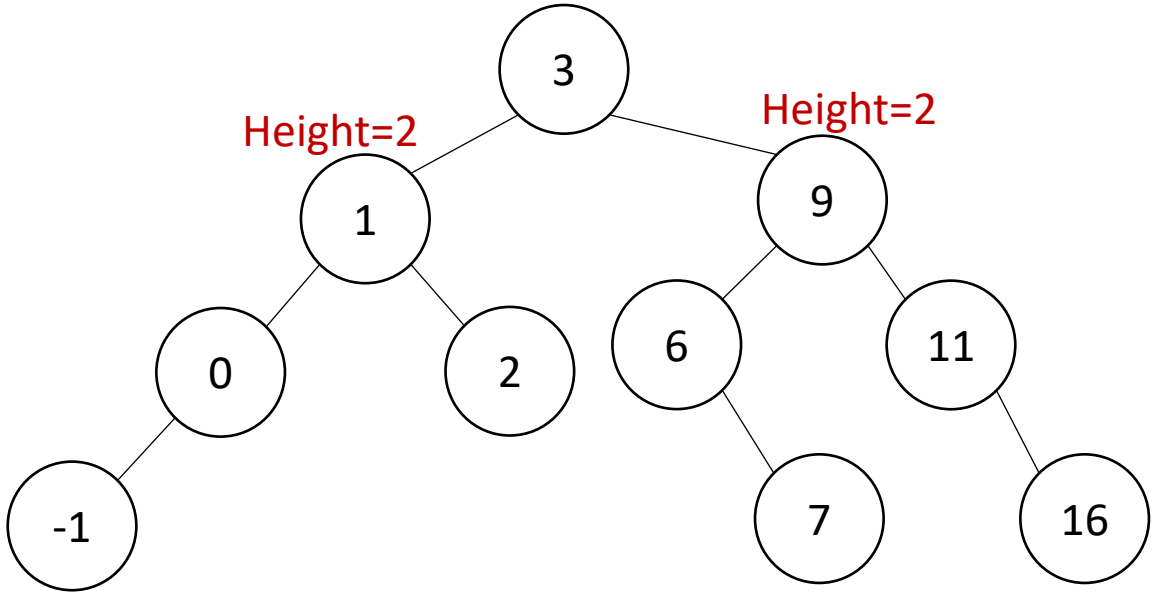


Insert -1, Right rotation process

Solution:
Take the subtree starting with the problem node,
“Rotate” that tree to the right

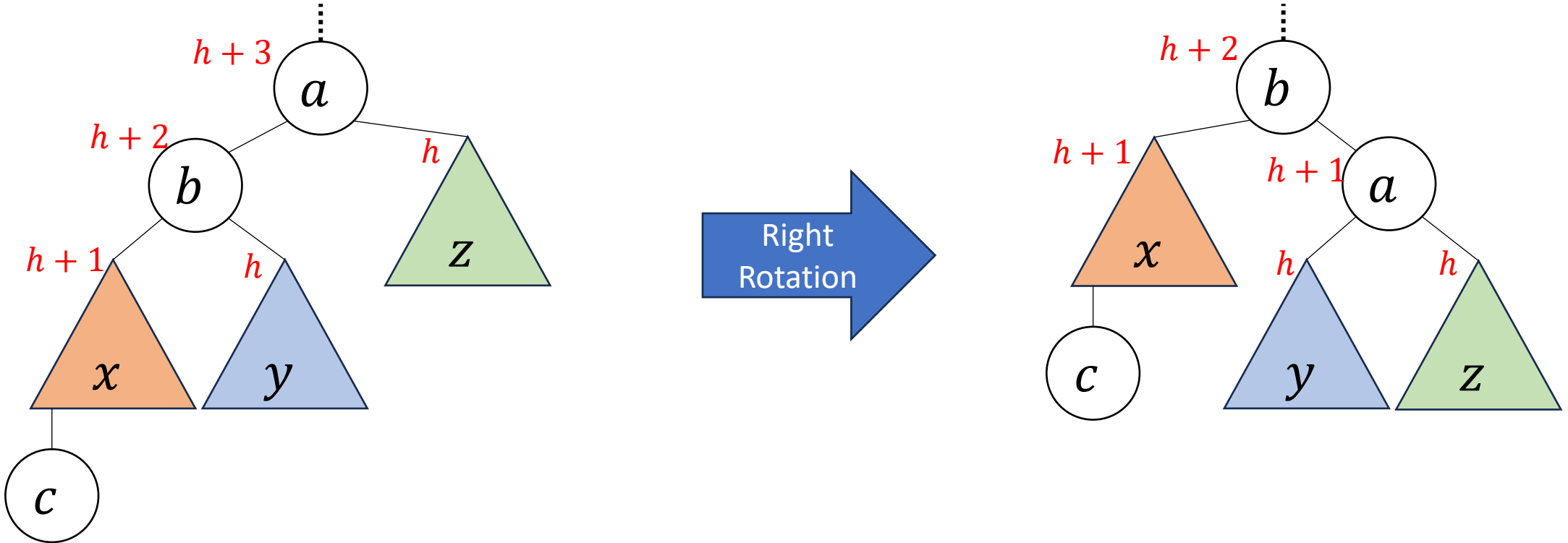


After the Right Rotation, Balanced!



Right Rotation - Definition

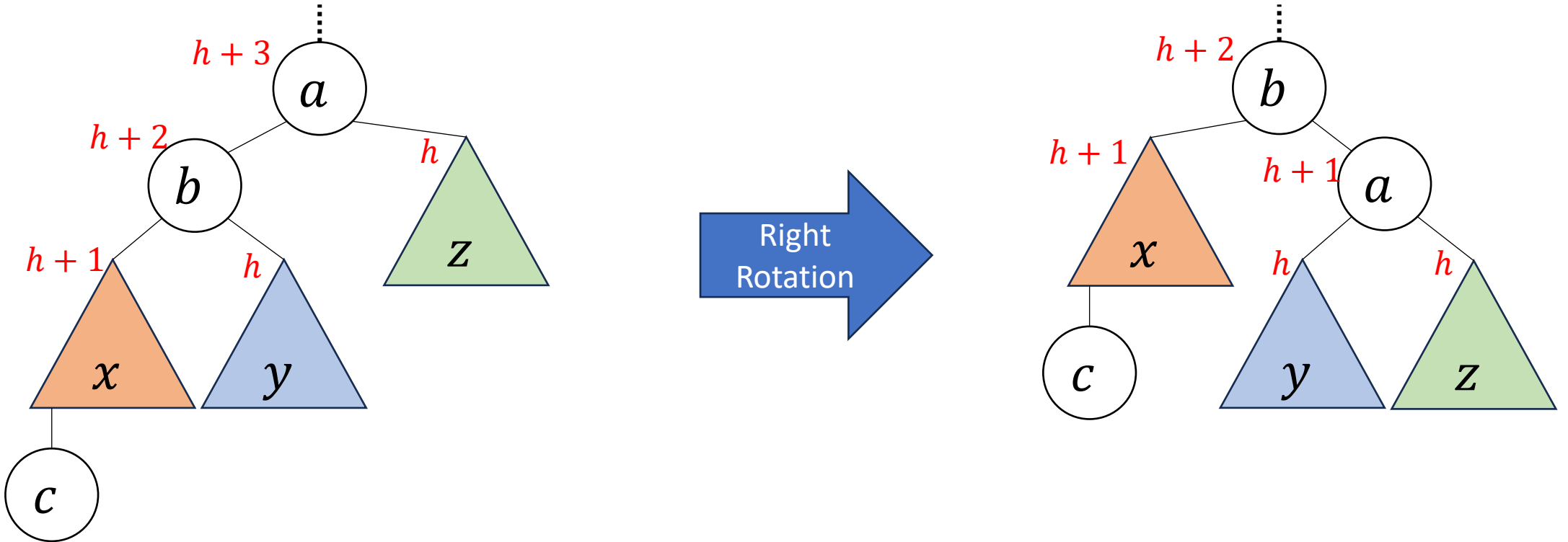
- We just inserted c, node a is the deepest "problem" node
- Make the left child the new root
- Make the old root the right child of the new
- Make the new root's right subtree the old root's left subtree



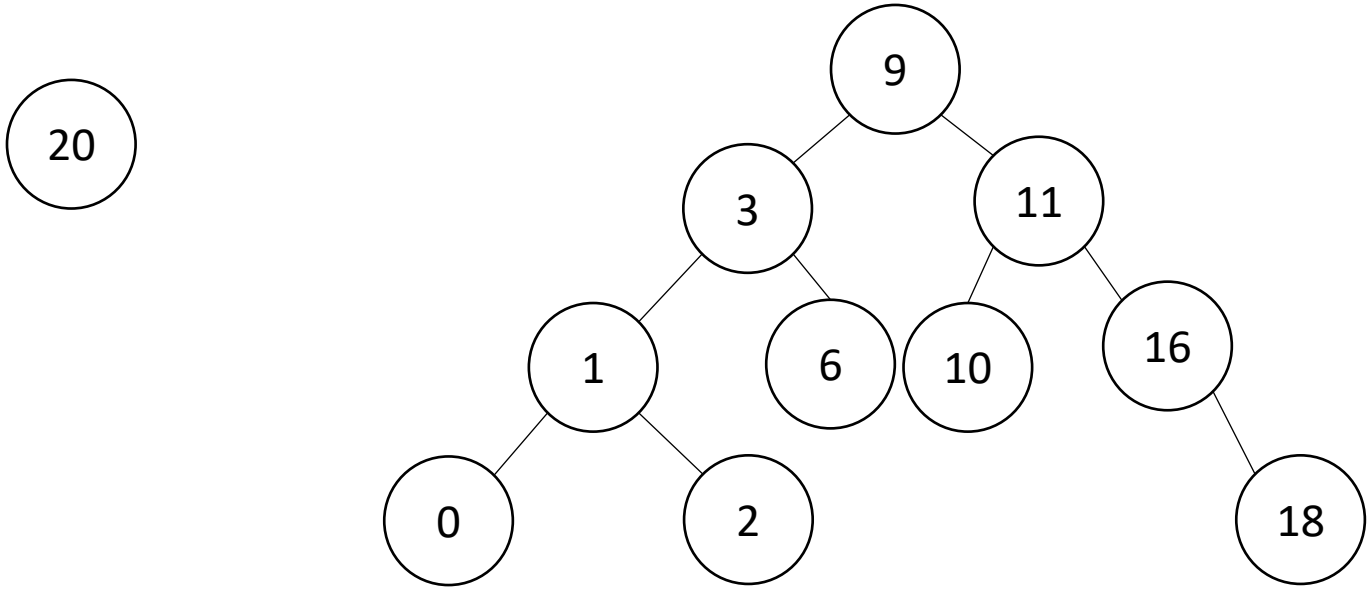
Right Rotation - Implementation

```
Node b = a.left
a.left = b.right
b.right = a
return b
```

Running time: $\Theta(1)$



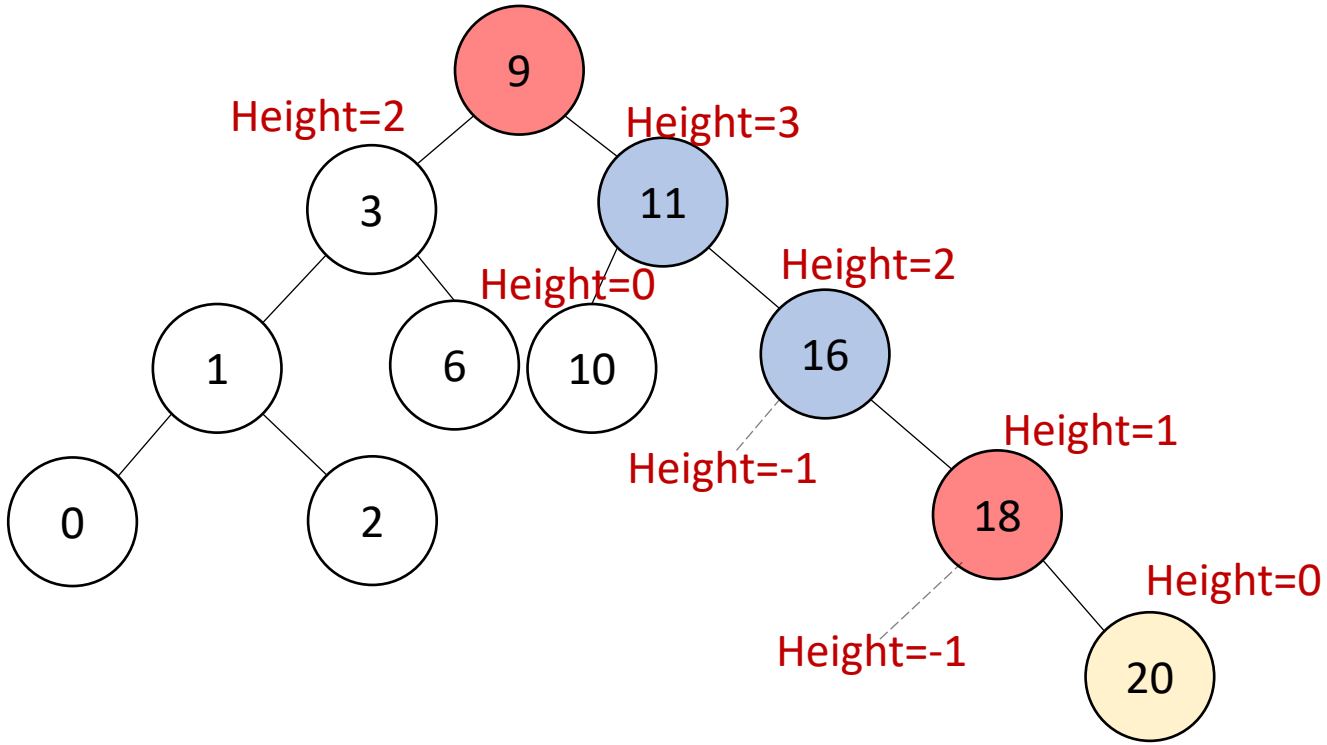
Insert Example (Insert 20)



Insert 20: Not Balanced! Multiple Problem Nodes

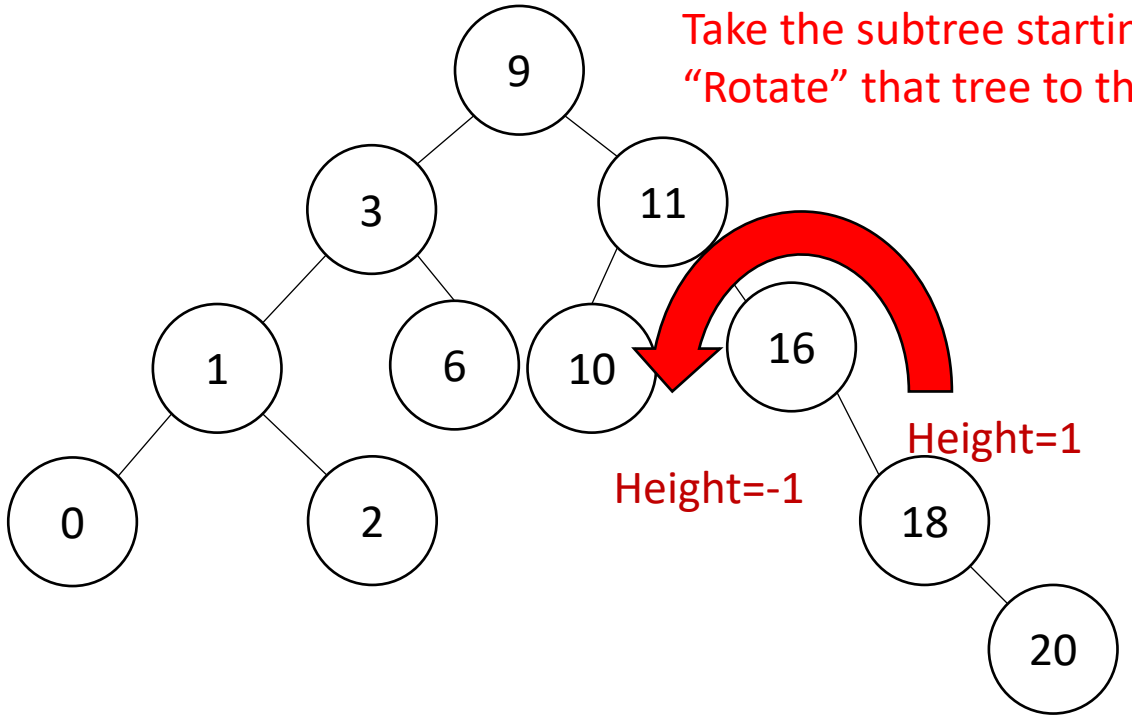
Here, nodes 11 and 16 are problem nodes.

There may be multiple places where we can do a rotation to rebalance the tree, but the *deepest* problem node *always works*!



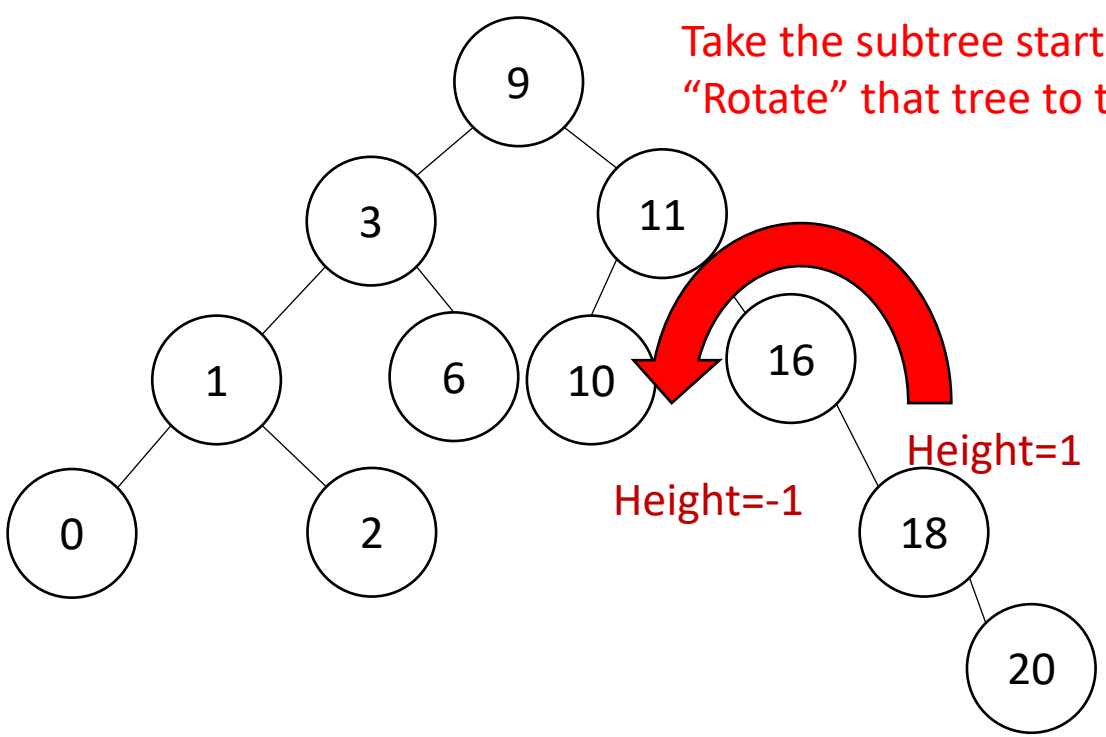
Left Rotation Needed!

Solution:
Take the subtree starting with the problem node,
“Rotate” that tree to the left

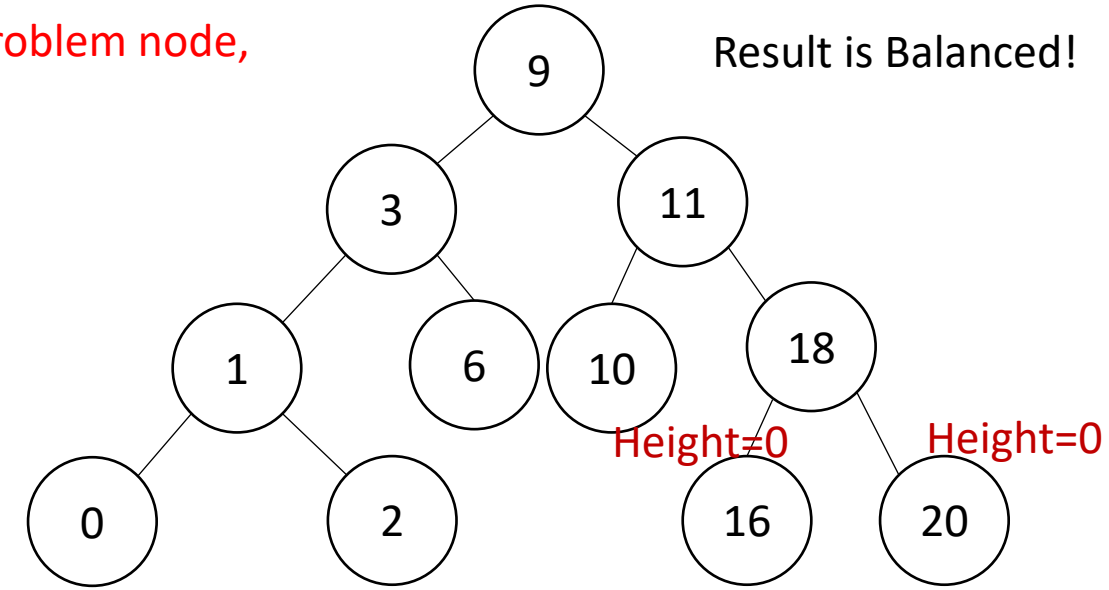


Inserting 20, Left rotation process

Solution:
Take the subtree starting with the problem node,
“Rotate” that tree to the left

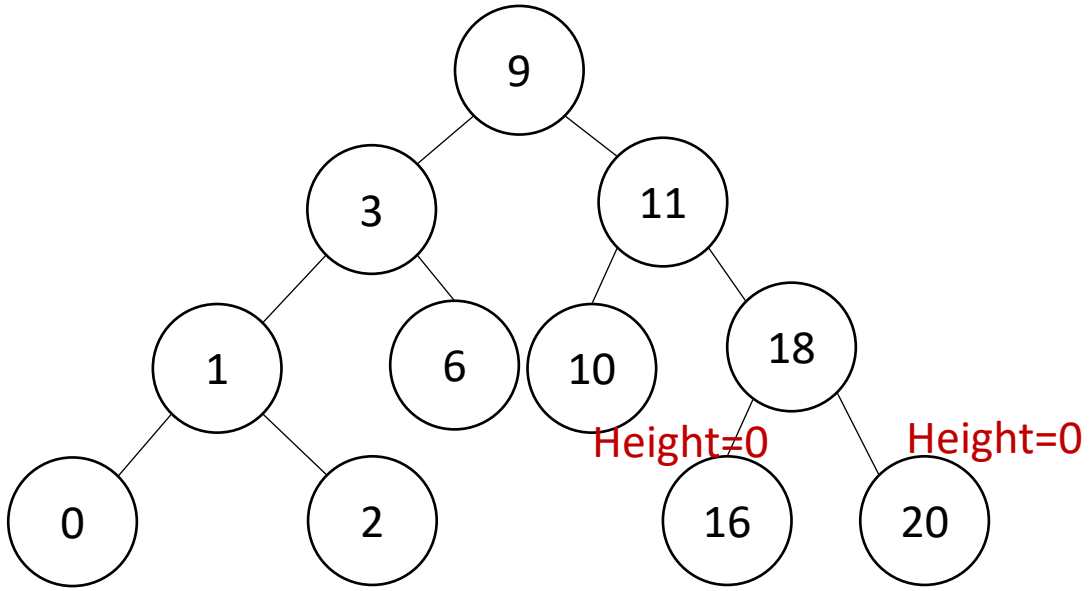


Left Rotation



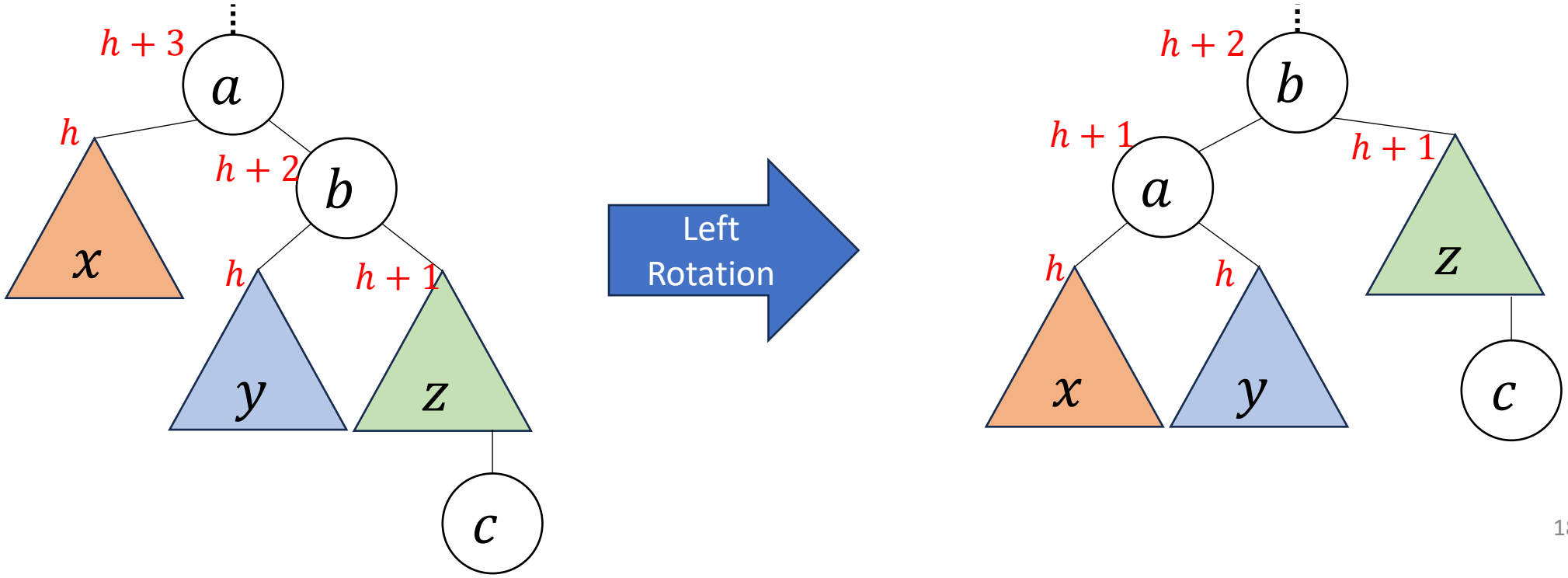
Result is Balanced!

After the Left Rotation, Balanced!



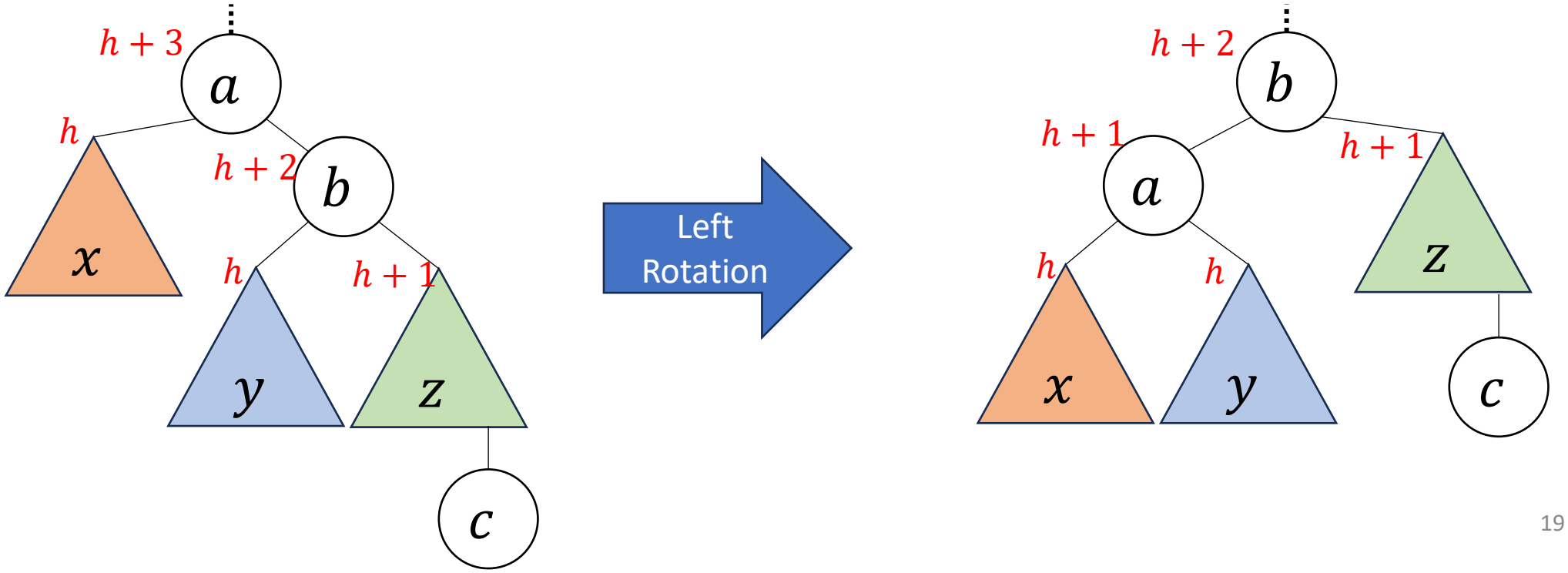
Left Rotation - Definition

- We just inserted c, node a is the deepest "problem" node
- Make the right child the new root
- Make the old root the left child of the new
- Make the new root's left subtree the old root's right subtree



Left Rotation (Implementation)

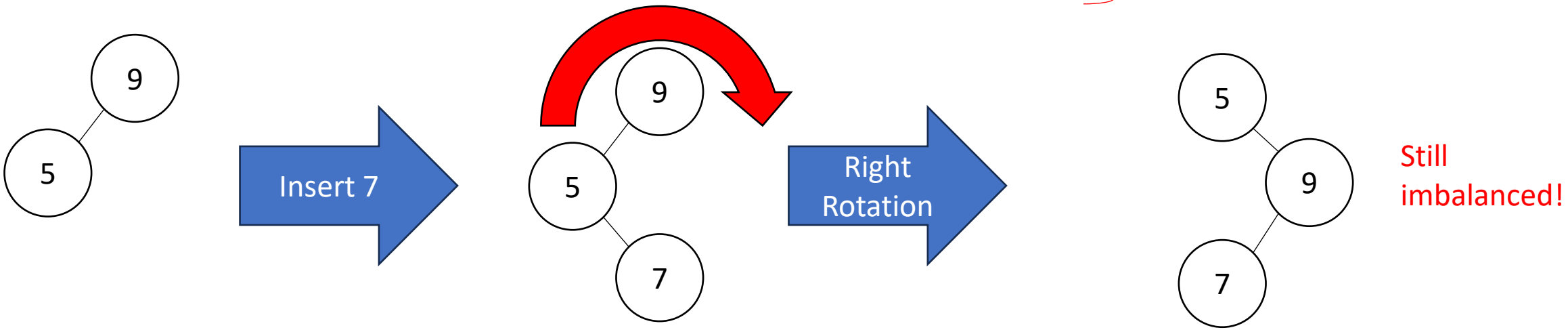
```
Node b = a.right
a.right = b.left
b.left = a
return b
```



Insertion Story So Far… still some missing pieces

- After insertion, update the heights of the node's ancestors
- Check for unbalance
- If unbalanced then at the deepest unbalanced root:
  - If the left subtree was deeper then rotate right
  - If the right subtree was deeper then rotate left

This is incomplete!
There are some cases where this doesn't work!



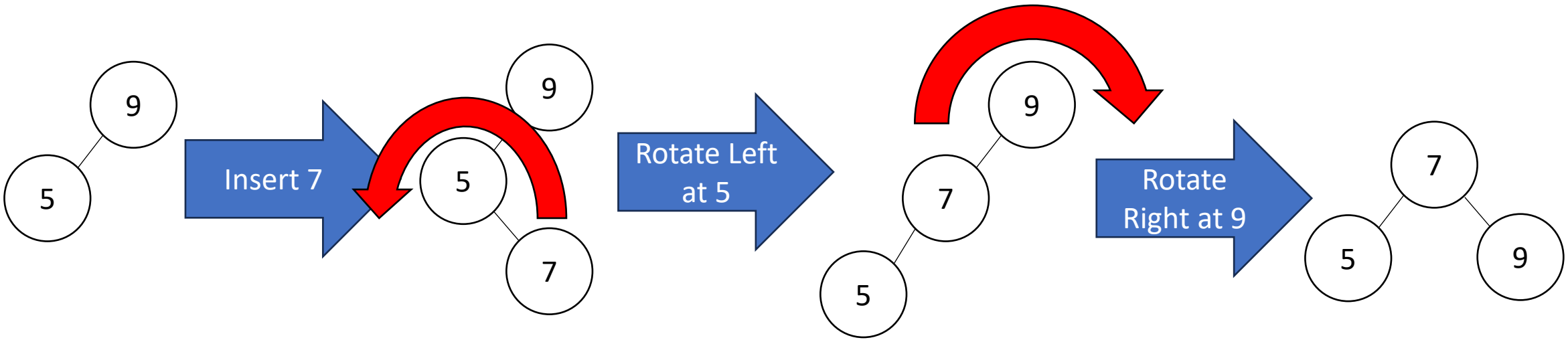
Insertion Cases

- After insertion, update the heights of the node's ancestors
- Check for unbalance
- If unbalanced then at the deepest unbalanced root:
  - Case LL: If we inserted in the **left** subtree of the **left** child then rotate right
  - Case RR: If we inserted in the **right** subtree of the **right** child then rotate left
  - Case LR: If we inserted into the **right** subtree of the **left** child then ???
  - Case RL: If we inserted into the **left** subtree of the **right** child then ???

Cases LR and RL require 2 rotations! (a double rotation)

Case LR

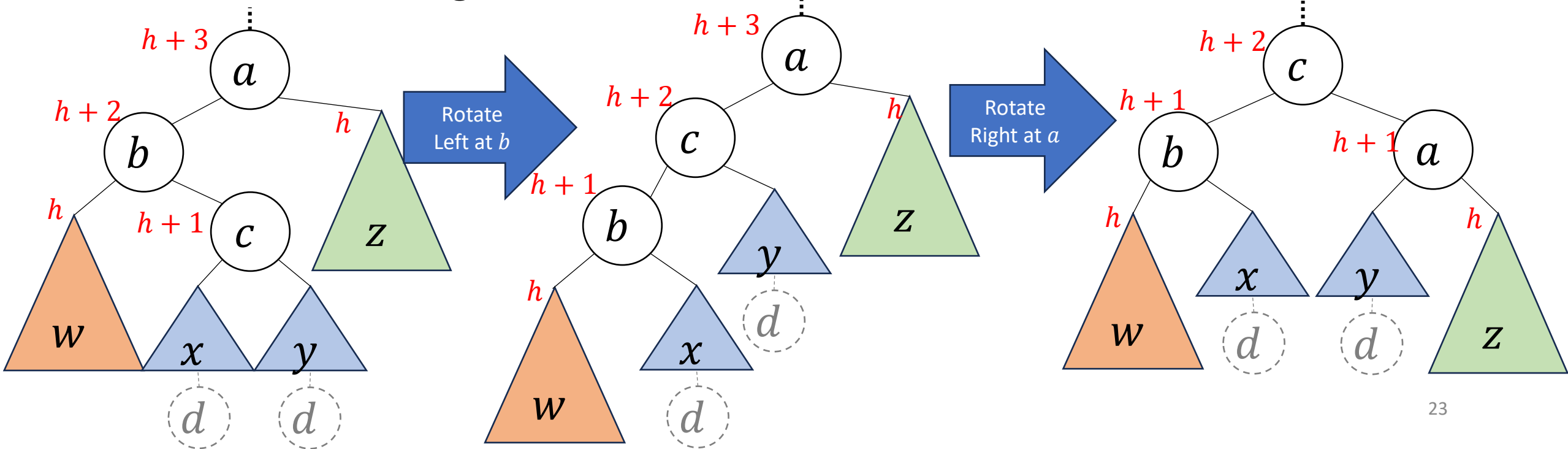
- From deepest problem node:
  1. Rotate left at the ***left child***
  2. Then, Rotate right at the problem node



Case LR - Definition

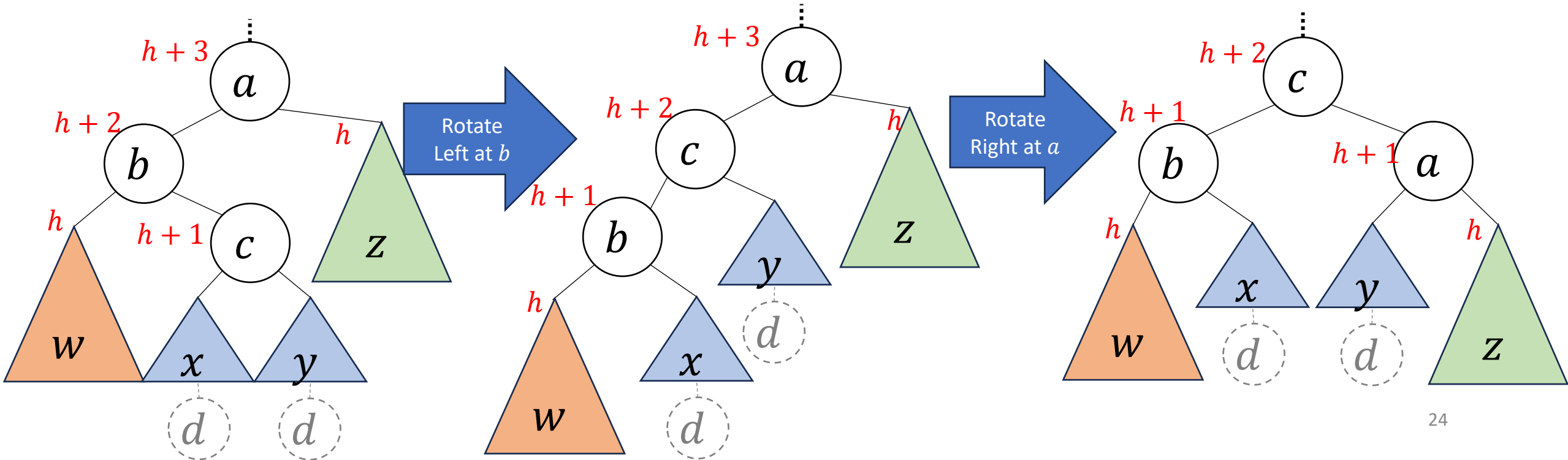
- We just inserted d, node a is the deepest "problem" node
- Imbalance caused by inserting in the left child's right subtree

1. Rotate left at the ***left child***
2. Then, rotate right at the unbalanced node



Case LR Implementation

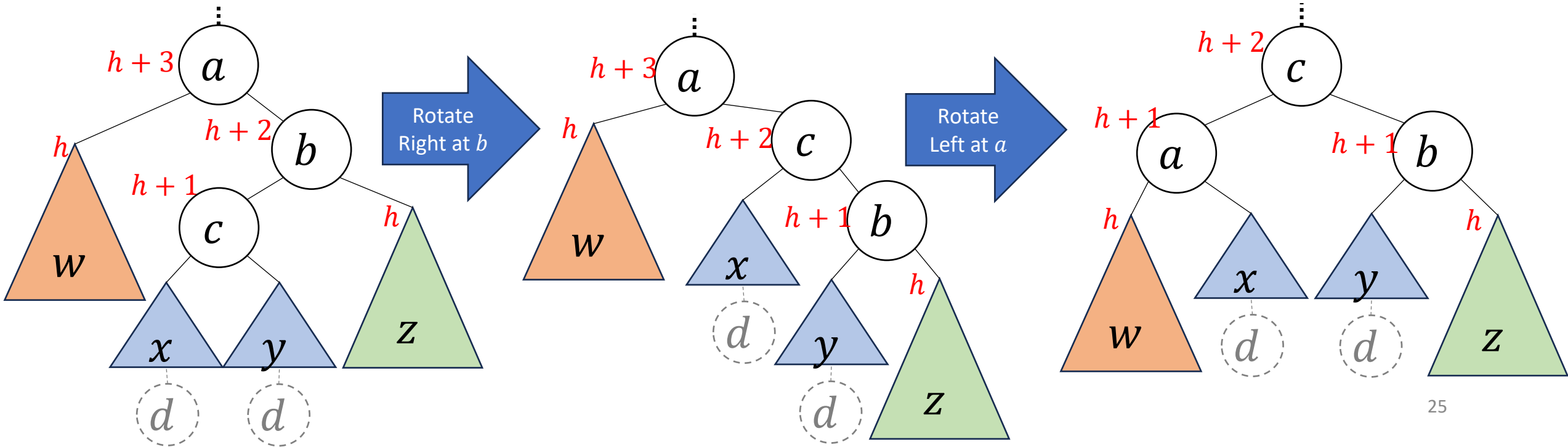
```
Node b = a.left
Node c = b.right
b.right = c.left
a.left = c.right
c.right = a
c.left = b
return c
```



Case RL - Definition

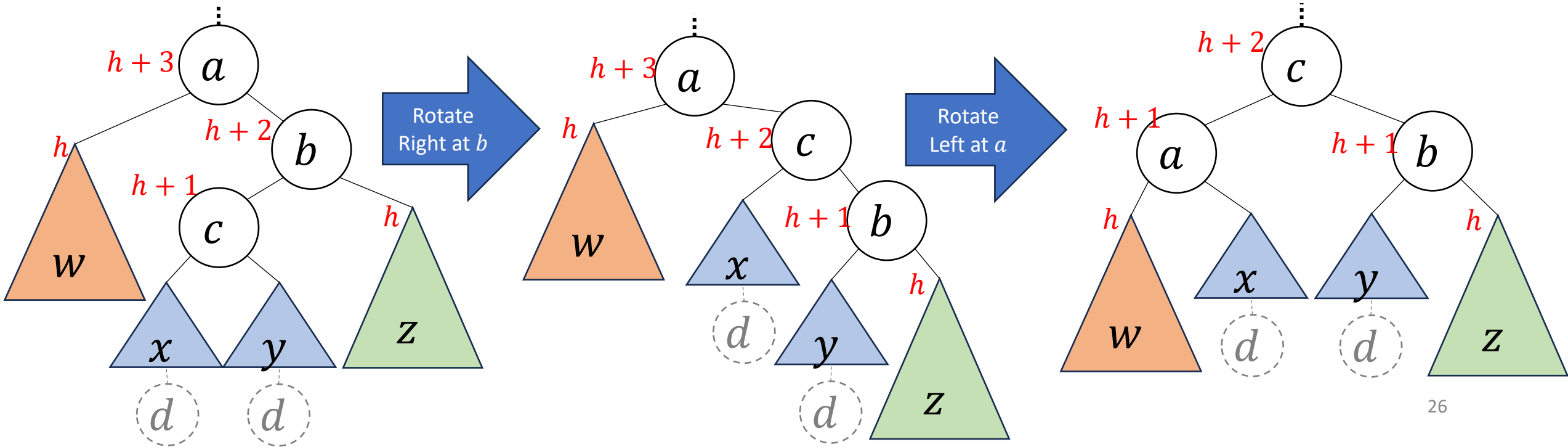
- We just inserted d, node a is the deepest "problem" node
- Imbalance caused by inserting in the right child's left subtree

1. Rotate right at the ***right child***
2. Then, rotate left at the unbalanced node



Case RL Implementation

```
Node b = a.right
Node c = b.left
b.left = c.right
a.right = c.left
c.left = a
c.right = b
return c
```

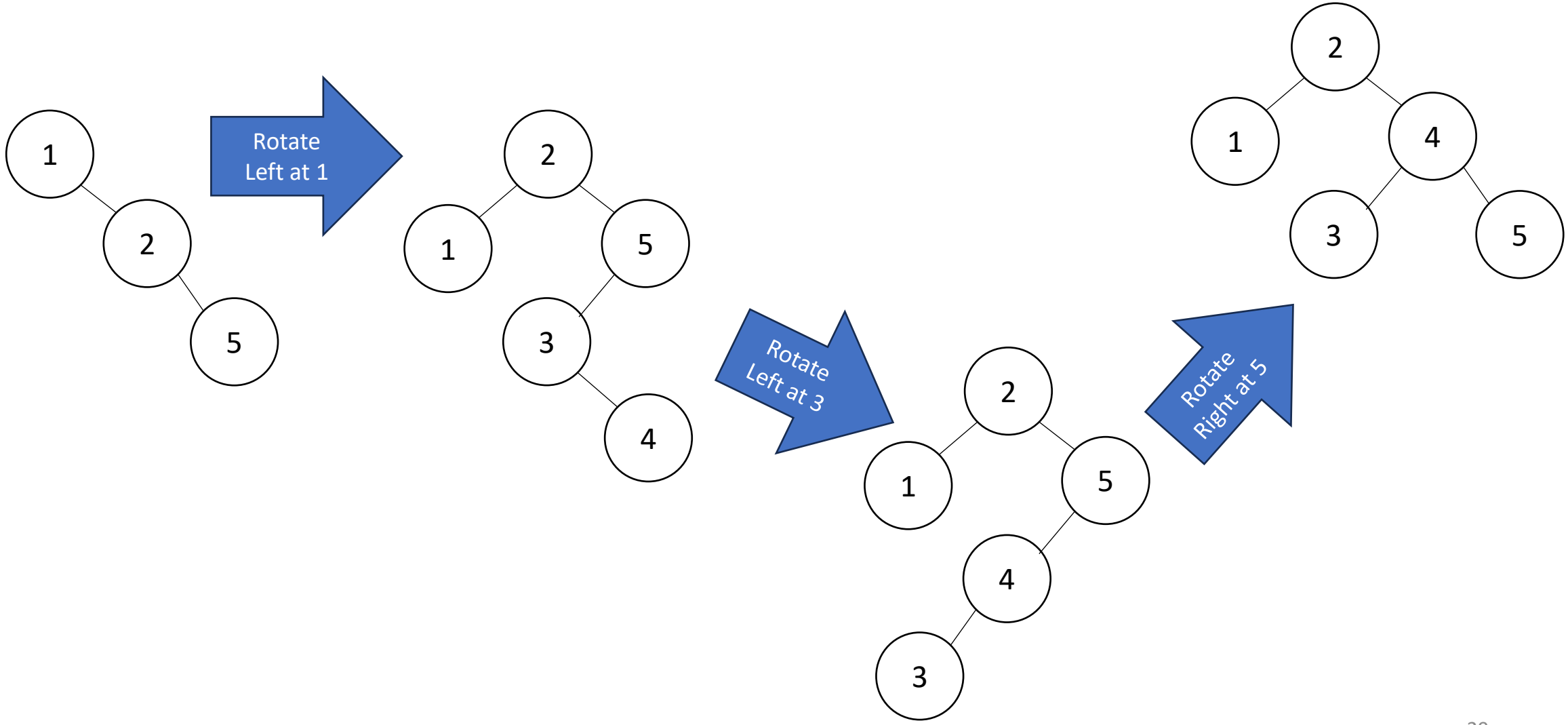


Insert Summary

- After a BST insertion, update the heights of the node's ancestors
- From leaf to root, check if each node is balanced
- If a node is unbalanced then at the deepest unbalanced node:
  - Case LL: If we inserted in the **left** subtree of the **left** child then: rotate right
  - Case RR: If we inserted in the **right** subtree of the **right** child then: rotate left
  - Case LR: If we inserted into the **right** subtree of the **left** child then: rotate left at the left child and then rotate right at the root
  - Case RL: If we inserted into the **left** subtree of the **right** child then: rotate right at the right child and then rotate left at the root
- Done after either reaching the root or applying **one** of the above cases

Insert 1, 2, 5, 3, 4 into an AVL tree

Insert 1, 2, 5, 3, 4 into an AVL tree (Answers)



Insert Operation (for AVL)

```
insert(key, value, root) {
        root = insertHelper(key, value, root);
}
insertHelper(key, value, root) {
        if (root == null)
                return new Node(key, value);
        if (root.key < key)
                root.right = insertHelper(key, value, root.right);
        else
                root.left = insertHelper(key, value, root.left);
        if (isUnbalanced(root))
                root = rotate(root);
        return root;
}
```

Bounding the Height

- Suppose you have a tree of height h, meeting the AVL condition:

> AVL condition: For every node, the height of its left subtree and right subtree differ by at most 1.

- What is the **minimum number of nodes** in the tree?

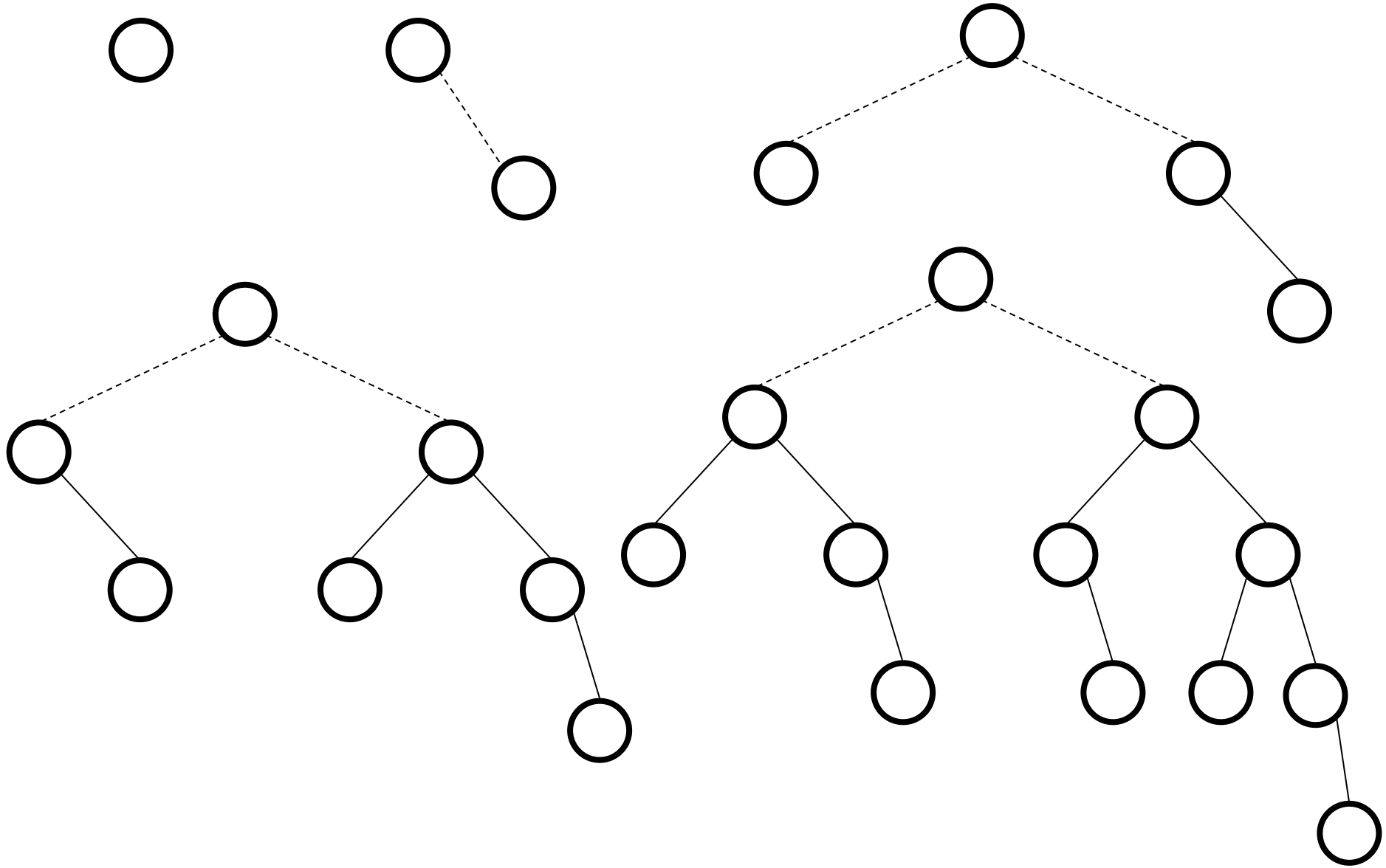
- If $h = 0$, then 1 node
- If $h = 1$, then 2 nodes.

- In general?

Let $\boldsymbol{M(h)}$ be the minimum # of nodes in an AVL tree of height $\boldsymbol{h}$, then:

$\boldsymbol{h}$	***Minimal AVL Tree***	$\boldsymbol{M(h)}$

Minimal AVL Trees



Why is this $\Theta(\log n)$ time?

- We get poor running times when $height \approx n$
- Let $M(h)$ be the minimum count of nodes in an AVL tree of height h
- An AVL tree of height h must have one subtree of height $h - 1$
- This means the other subtree has height at least $h - 2$
- $M(h) = M(h-1) + M(h-2) + 1$

Comparing to Fibonacci Sequence

- $M(h) = M(h-1) + M(h-2) + 1$
- $F(n) = F(n-1) + F(n-2)$
  - Fibonacci Sequence
- So $M(h) > F(h)$
- For large values of h, $F(h) \approx \phi^h$
  - ϕ being the golden ratio. $\phi > 1.6$
- This means that an AVL tree of height h has at least ϕ^h nodes
  - And not more than $2^{h+1} - 1$ nodes
- So a tree of n nodes has height at most $\log_\phi(n)$
  - We need $n \geq \phi^h$, so $\log_\phi n \geq h$
- All operations run in time $O(\log n)$
  - The maximum number of nodes for height h is $2^{h+1} - 1$, so they are also $\Omega(\log n)$