

CSE 332 Autumn 2026

Lecture 8: AVL Trees, pt. 1

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: due TONIGHT, Fri Oct 16
- Week 3 Survey in canvas: Due Tuesday Oct 20
- Exercise 3 Recurrences: due Fri Oct 23
- Exercise 4 AVL Trees, coming soon, Due Friday Oct 30
- Midterm: 5-6pm Monday November 9

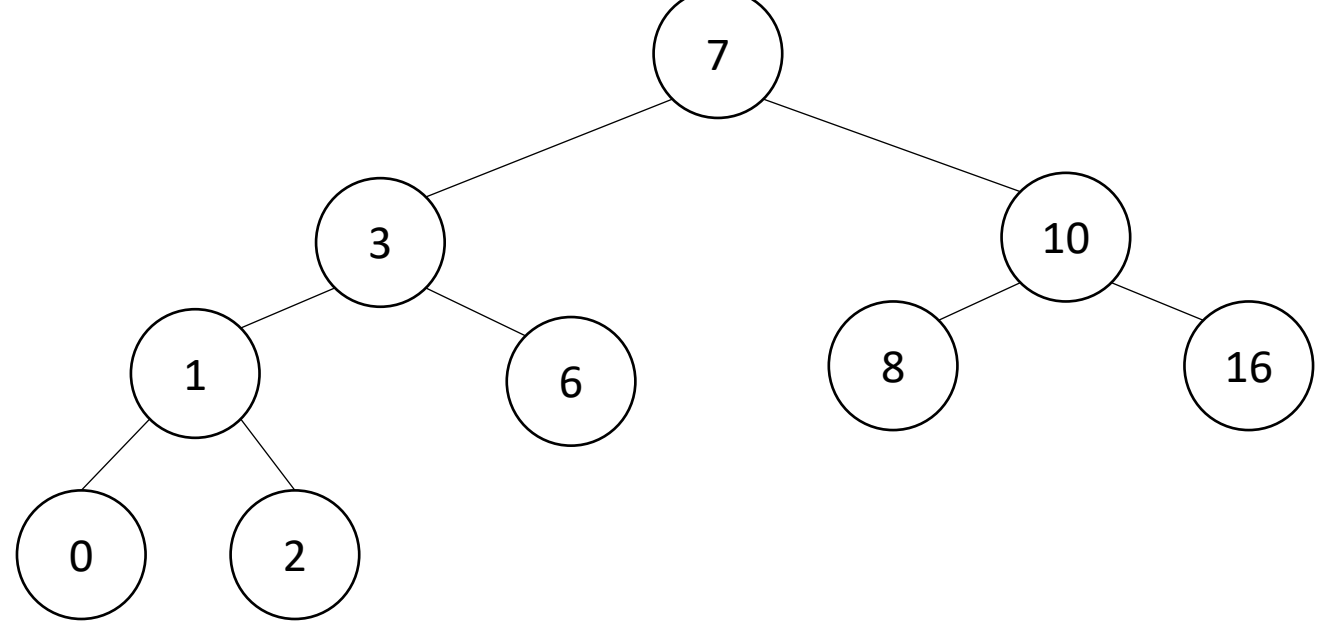
Dictionary (Map) ADT

- Contents:
 - Sets of key+value pairs
 - Keys must be:
 - Unique (always) and Comparable (for many dictionary implementations)
- Operations:
 - insert(key, value)
 - Adds the (key, value) pair into the dictionary
 - If the key already has a value, overwrite the old value
 - Consequence: Keys cannot be repeated
 - find(key)
 - Returns the value associated with the given key
 - delete(key)
 - Remove the key (and its associated value)

Naïve attempts (Answers)

Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Min Heap?	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (worst)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (expected)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

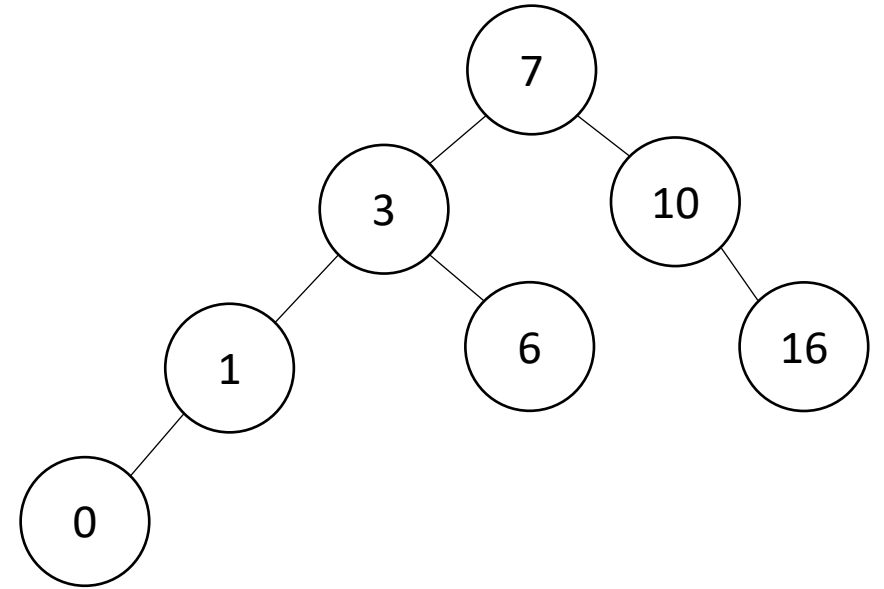
Binary Search Tree



- Binary Tree
 - Definition:
 - Tree where each node has at most 2 children
- Order Property
 - All keys in the left subtree are smaller than the root
 - All keys in the right subtree are larger than the root
 - Consequence: cannot have repeated values

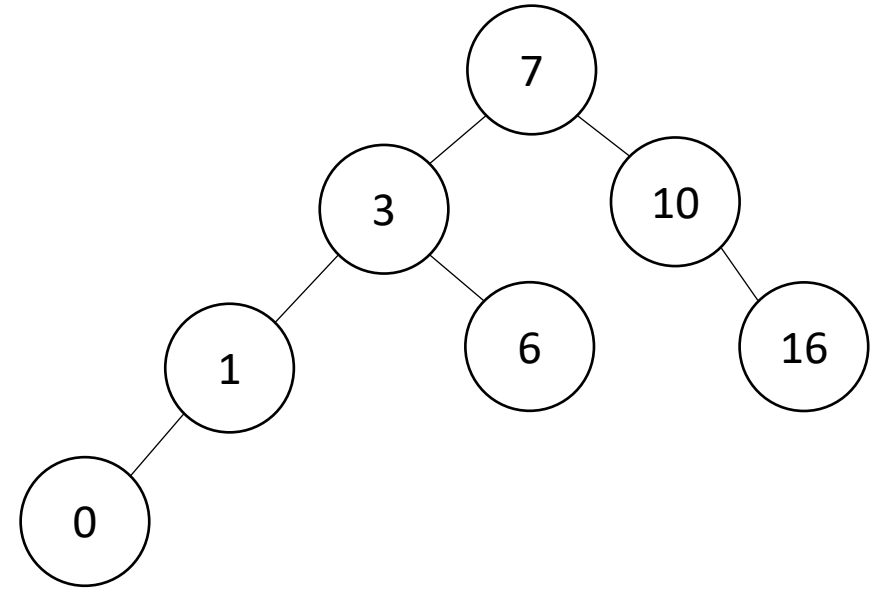
Find Operation (recursive)

```
find(key, root) {  
    if (root == Null) {  
        return Null;  
    }  
    if (key == root.key) {  
        return root.value;  
    }  
    if (key < root.key) {  
        return find(key, root.left);  
    }  
    if (key > root.key) {  
        return find(key, root.right);  
    }  
    return Null;  
}
```



Insert Operation (recursive)

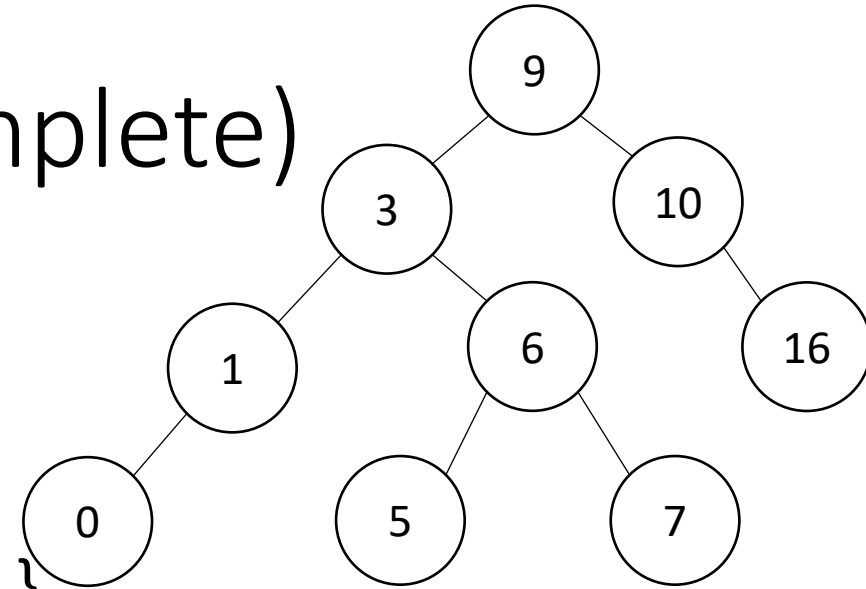
```
insert(key, value, root) {  
    root = insertHelper(key, value, root);  
}  
  
insertHelper(key, value, root) {  
    if (root == null)  
        return new Node(key, value);  
    if (root.key < key)  
        root.right = insertHelper(key, value, root.right);  
    else  
        root.left =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



Note: Insert happens only at the leaves!

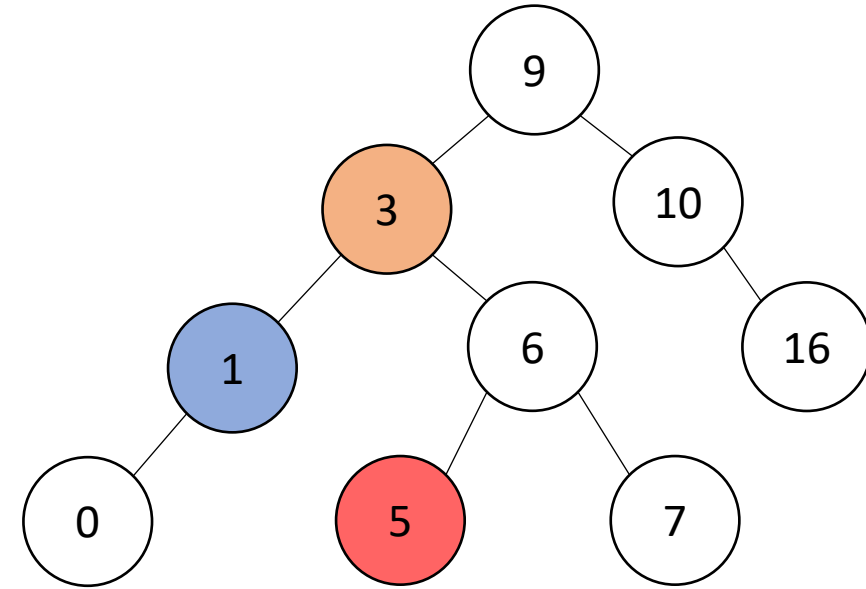
Delete Operation (iterative, incomplete)

```
delete(key, root) {  
    while (root != Null && key != root.key) {  
        if (key < root.key) { root = root.left; }  
        else if (key > root.key) { root = root.right; }  
    }  
    if (root == Null) { return; }  
    // Now root is the node to delete, what happens next?  
}
```



Delete – 3 Cases

- 0 Children (i.e. it's a leaf) (e.g. delete 5)
- 1 Child (e.g. delete 1)
 - Replace the deleted node with its child
- 2 Children (e.g. delete 3)
 - Replace the deleted with the largest node to its left or else the smallest node to its right

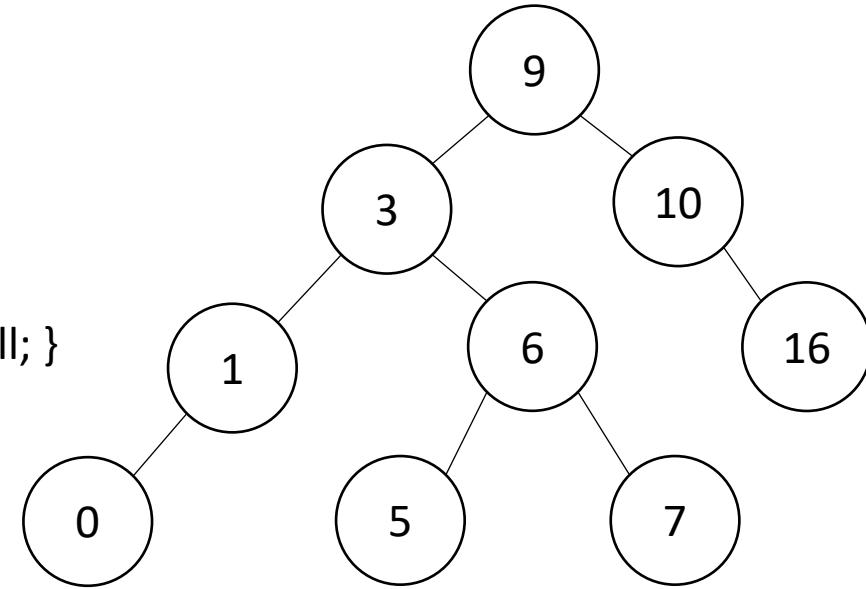


Finding the Max and Min

- Max of a BST:
 - Right-most Thing
- Min of a BST:
 - Left-most Thing

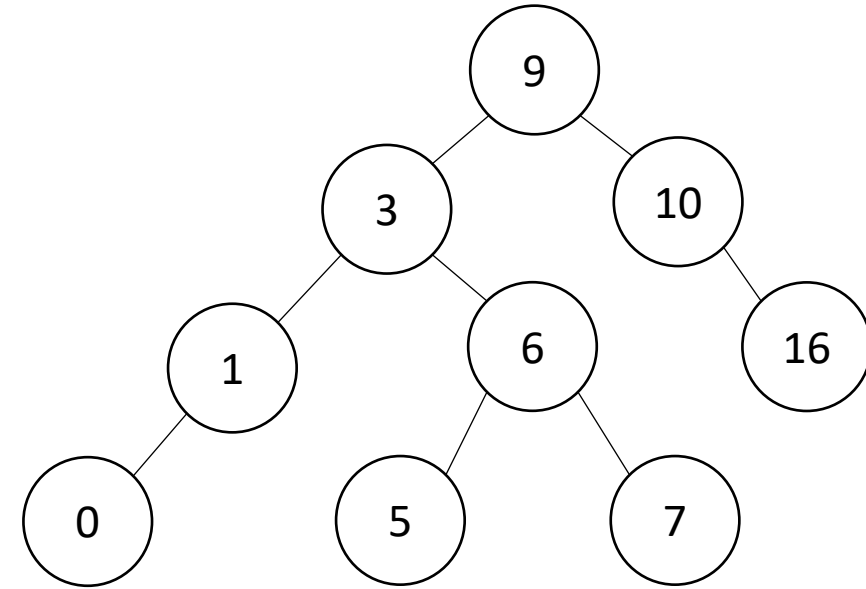
```
maxNode(root) {  
    if (root == Null) { return Null; }  
    while (root.right != Null) {  
        root = root.right;  
    }  
    return root;  
}
```

```
minNode(root) {  
    if (root == Null) { return Null; }  
    while (root.left != Null) {  
        root = root.left;  
    }  
    return root;  
}
```



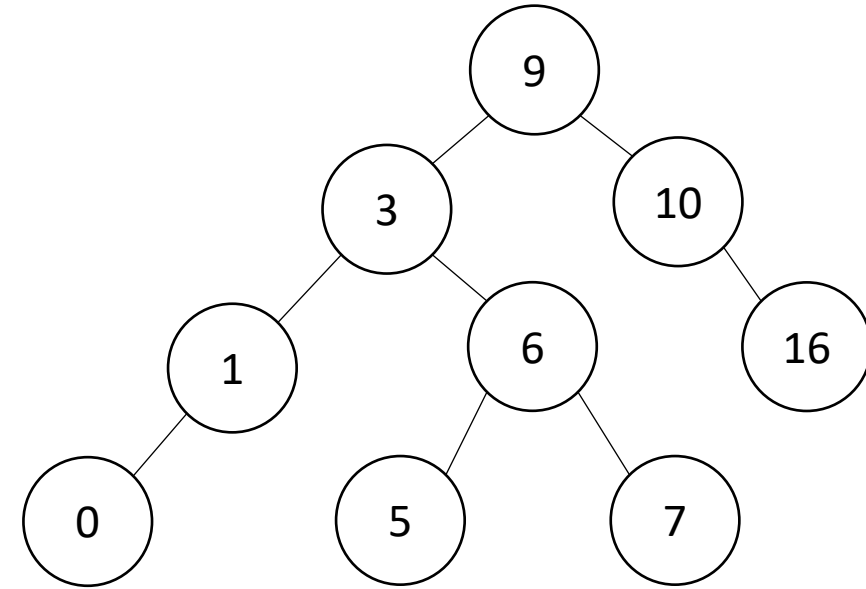
Delete Operation (iterative)

```
delete(key, root) {  
    while (root != Null && key != root.key) {  
        if (key < root.key) { root = root.left; }  
        else if (key > root.key) { root = root.right; }  
    }  
    if (root == Null) { return; }  
    if (root has no children) {  
        make parent point to Null Instead;  
    }  
    if (root has one child) {  
        make parent point to that child instead;  
    }  
    if (root has two children) {  
        make parent point to either the max from the left or min from the right  
    }  
}
```



Delete Operation (recursive)

```
delete(key, root) {  
    if (root == Null) { return; } // key not present  
    if (root.key == key) {  
        if (root has no children) { return Null; }  
        if (root has one child) { return that child; }  
        if (root has two children) { return removeMax(root.left); }  
    }  
    if (root.key < key) { root.right = delete(key, root.right); }  
    else { root.left = delete(key, root.left); }  
}
```



Worst Case Analysis

- For each of Find, insert, Delete:
 - Worst case running time matches height of the tree
- What is the maximum height of a BST with n nodes?
 - $\Theta(n)$

Binary Trees: Some Numbers (Answers)

Recall: height of a tree = longest path from root to leaf (count edges)

For binary tree of height h :

- max # of leaves: 2^h
- max # of nodes: $2^{(h+1)} - 1$
- min # of leaves: 1
- min # of nodes: $h + 1$

*For n nodes, we cannot do better than $O(\log n)$ height,
and we want to avoid $O(n)$ height*

Improving the worst case

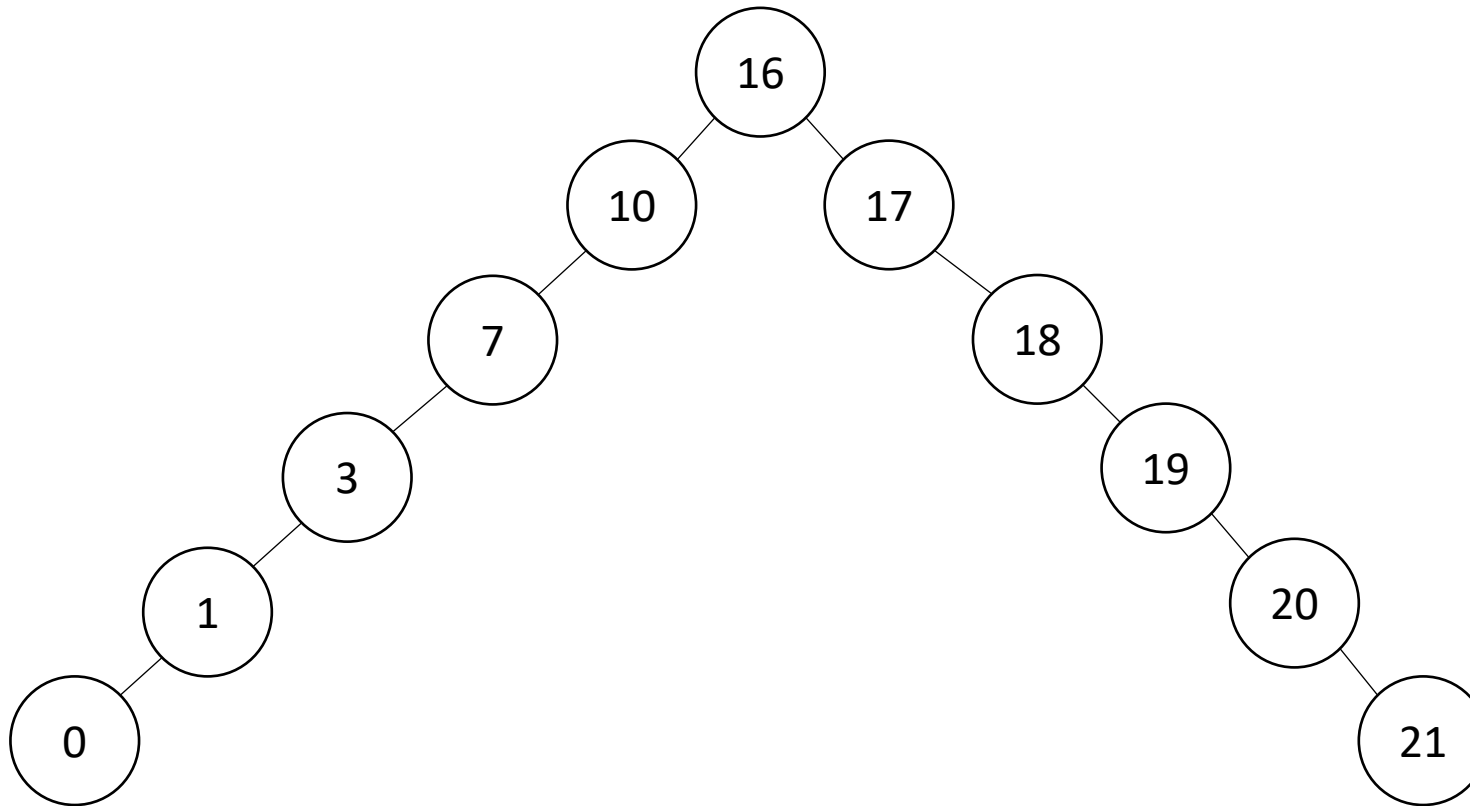
- How can we get a better worst case running time?
 - Add rules about the shape of our BST
- AVL Tree
 - A BST with some shape rules
 - Algorithms need to change to accommodate those

“Balanced” Binary Search Trees

- We get better running times by having “shorter” trees
- Trees get tall due to them being “sparse” (many one-child nodes)
- Idea: modify how we insert/delete to keep the tree more “full”
 - Encourage Branches!

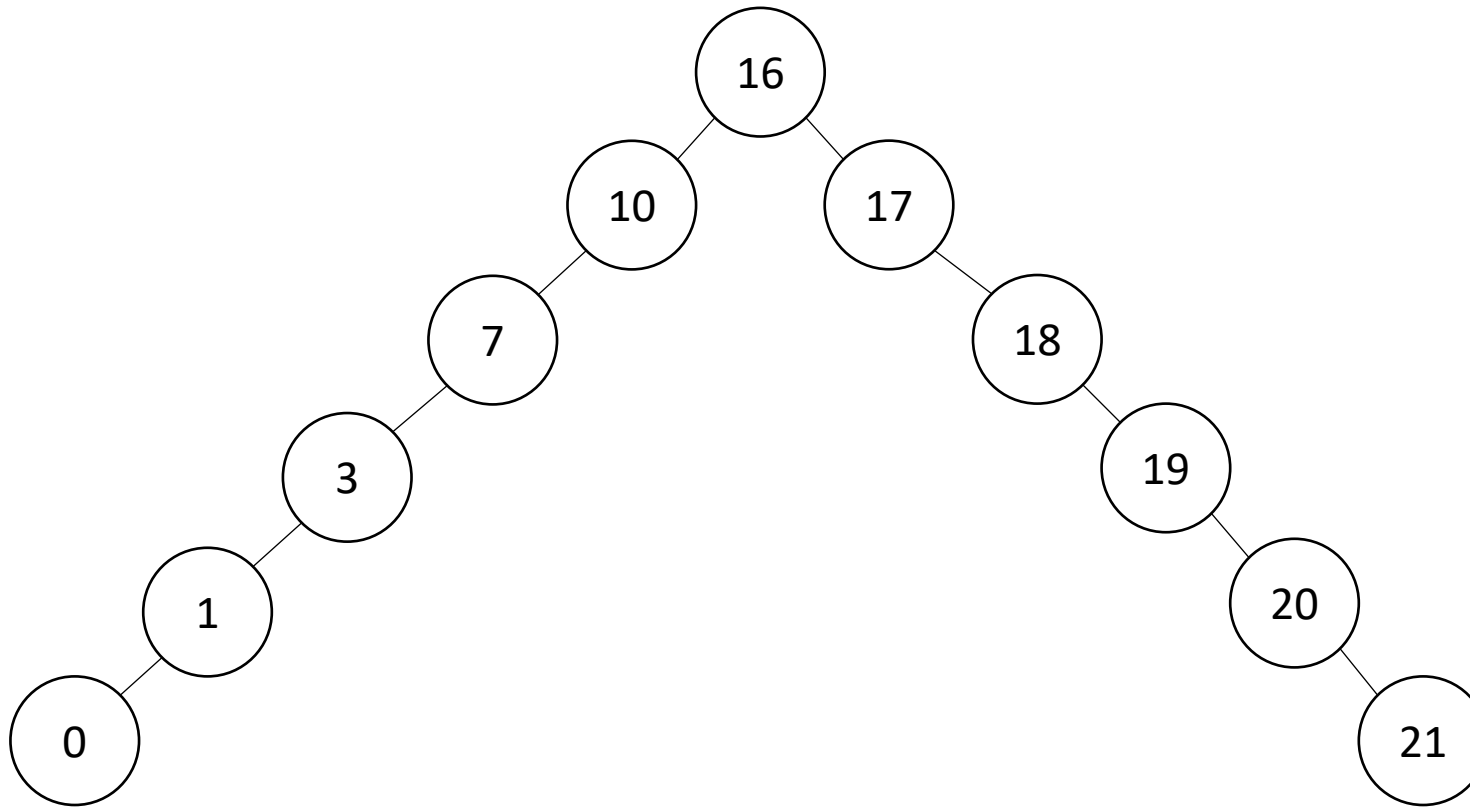
Idea 1: Both Subtrees of Root have same #
Nodes

Idea 1: Both Subtrees of Root have same # Nodes - Issue



Idea 2: Both Subtrees of Root have same height

Idea 2: Both Subtrees of Root have same height - Issue



Idea 3: Both Subtrees of every Node have same # Nodes

Idea 3: Both Subtrees of every Node have same # Nodes - Issue

Not all tree sizes are possible!

For example, cannot have a tree of size 2.

Idea 4: Both Subtrees of every Node have same height

Idea 4: Both Subtrees of every Node have same height - Issue

Not all tree sizes are possible!

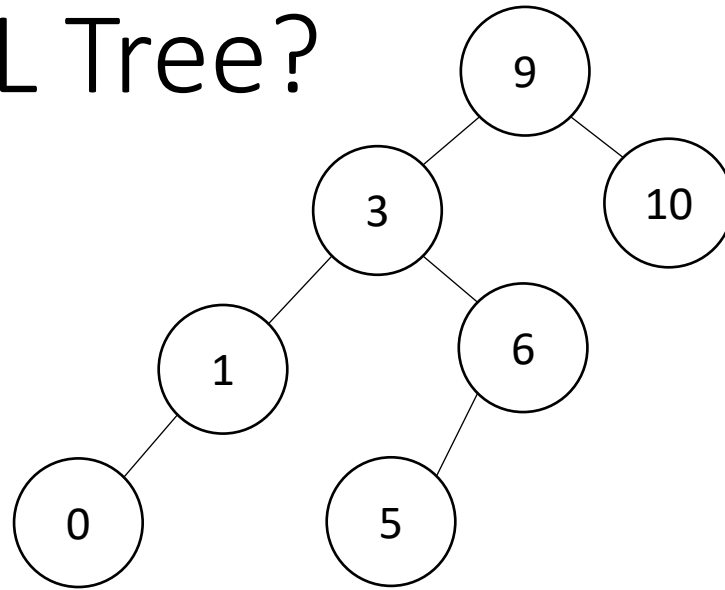
For example, cannot have a tree of size 2.

AVL Tree

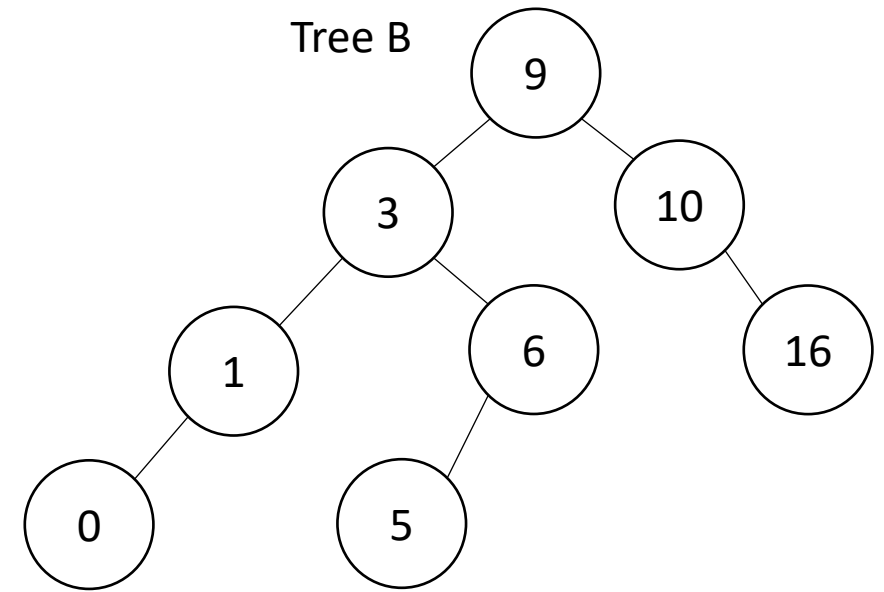
- A Binary Search tree that maintains that the left and right subtrees of every node have heights that differ by at most one.
 - height of left subtree and height of right subtree differ by at most 1
 - Not too weak (ensures trees are short)
 - Not too strong (works for any number of nodes)
- Idea of AVL Tree:
 - When you insert/delete nodes, if tree is “out of balance” then modify the tree
 - Modification = “rotation”

Is it an AVL Tree?

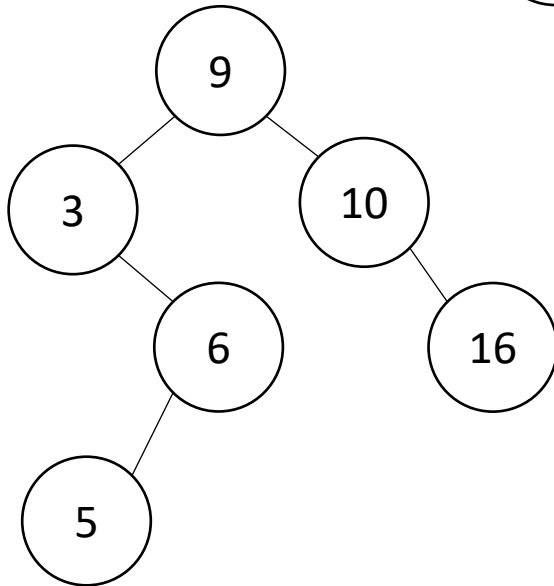
Tree A



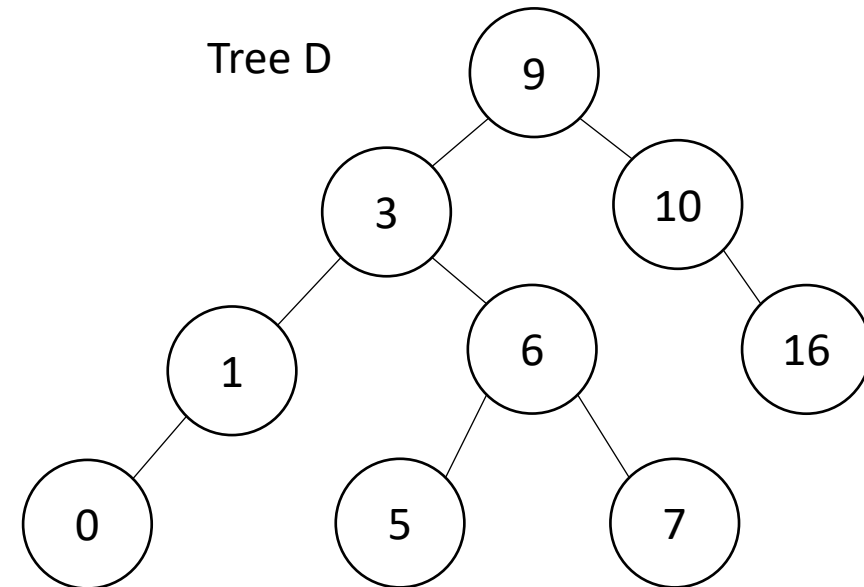
Tree B



Tree C



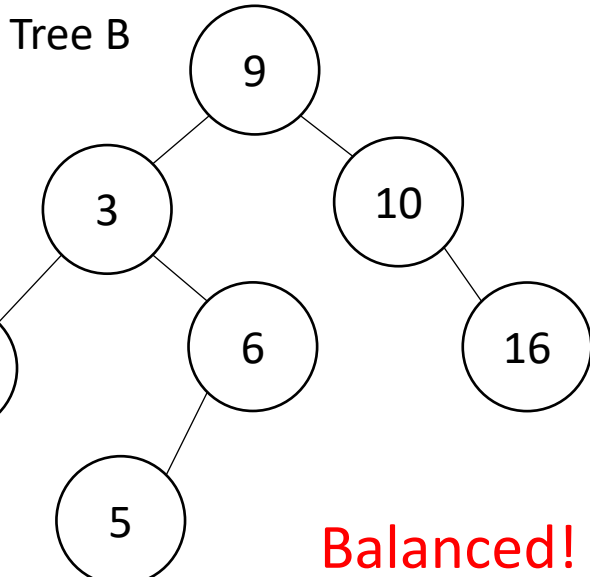
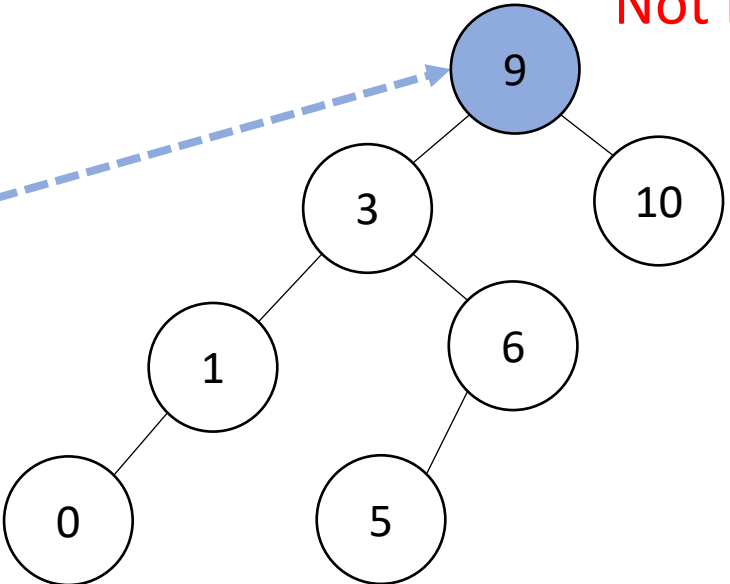
Tree D



Is it an AVL Tree? (Answers)

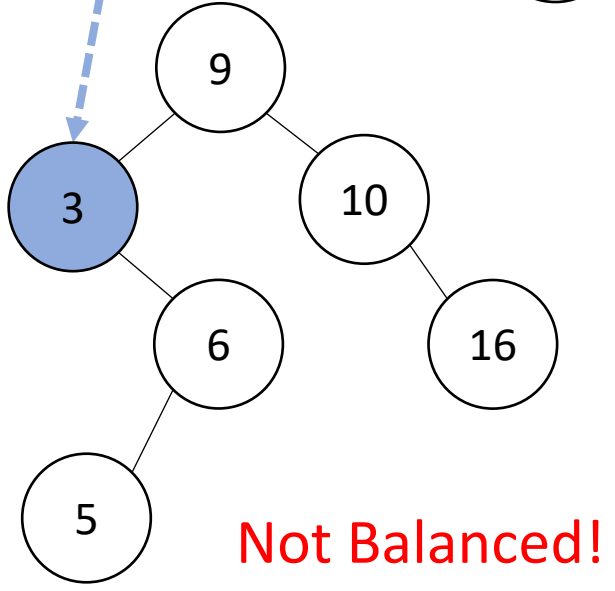
“Problem” Node
Its children’s heights differ by more than 1

Tree A
Not Balanced!



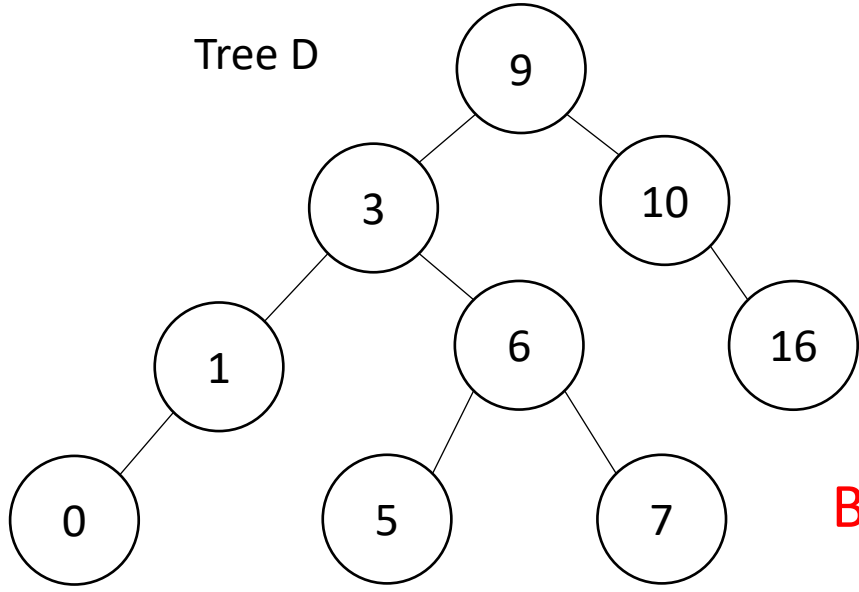
Balanced!

Tree C



Not Balanced!

Tree D

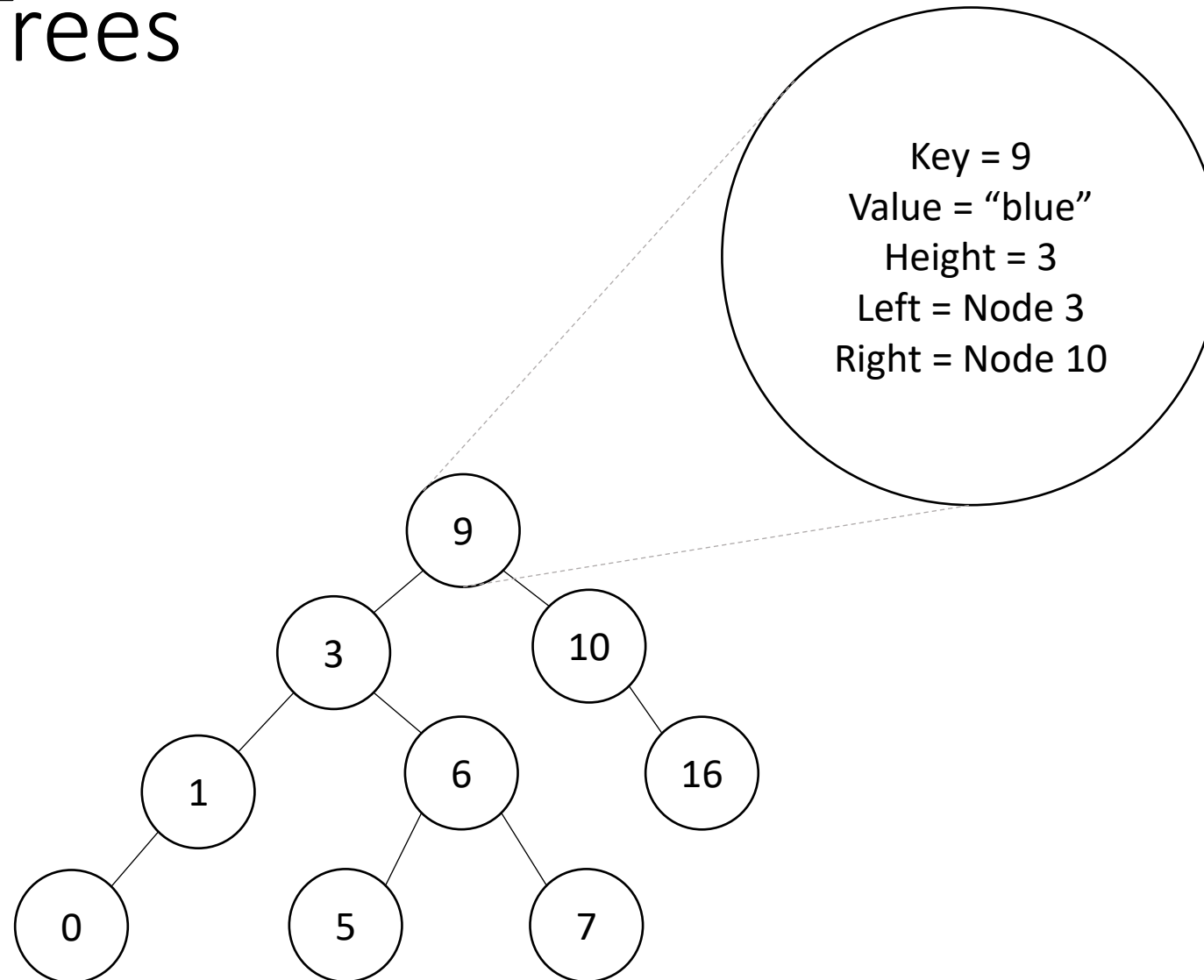


Balanced!

Using AVL Trees

- Each node has:

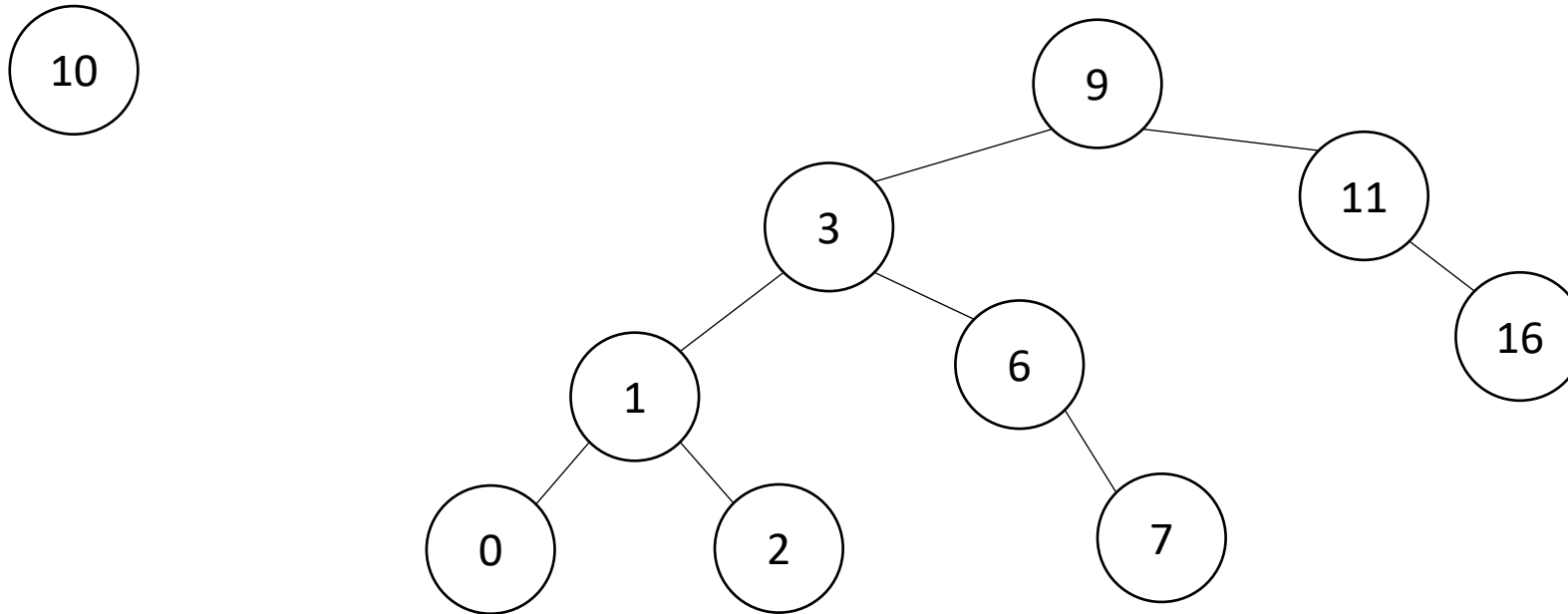
- Key
- Value
- Height
- Left child
- Right child



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 10)

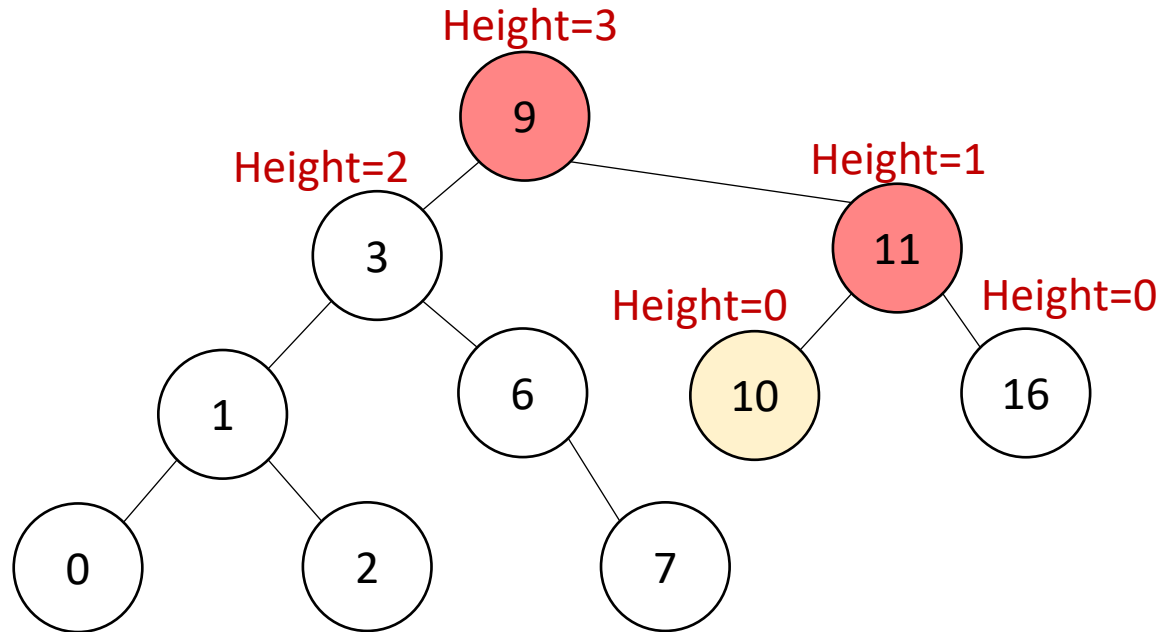


Insert 10 (After): No rotations needed

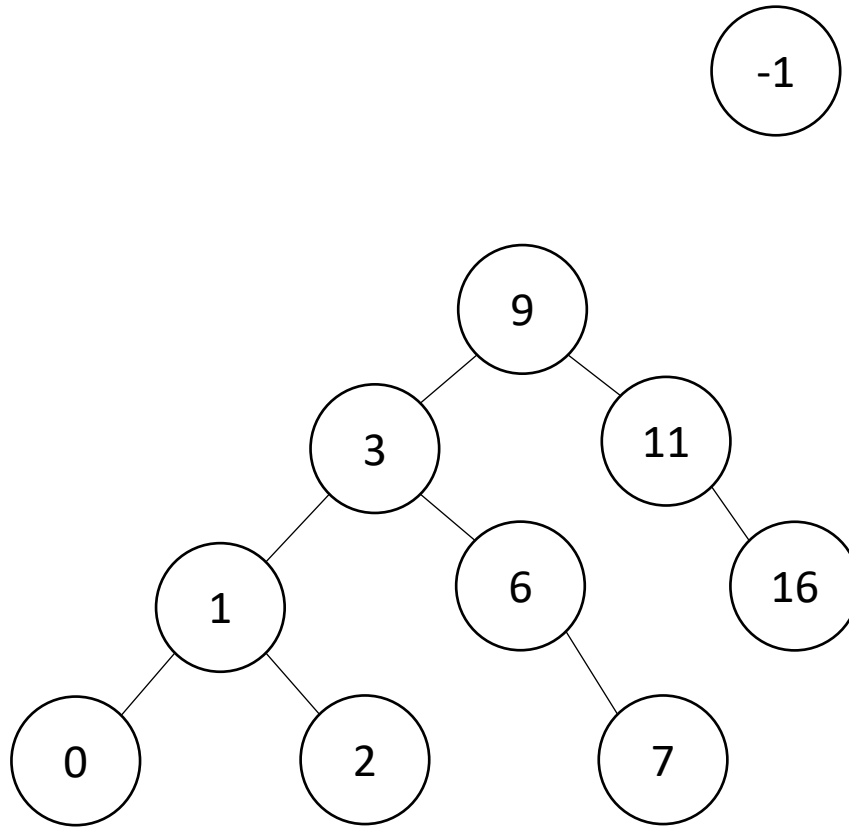
Is the tree still balanced?

To confirm we only need to check nodes in the path from root to the new node

Why? We assume the tree was balanced before the insert, so unchanged subtrees cannot be unbalanced.



Insert Example (Insert -1)



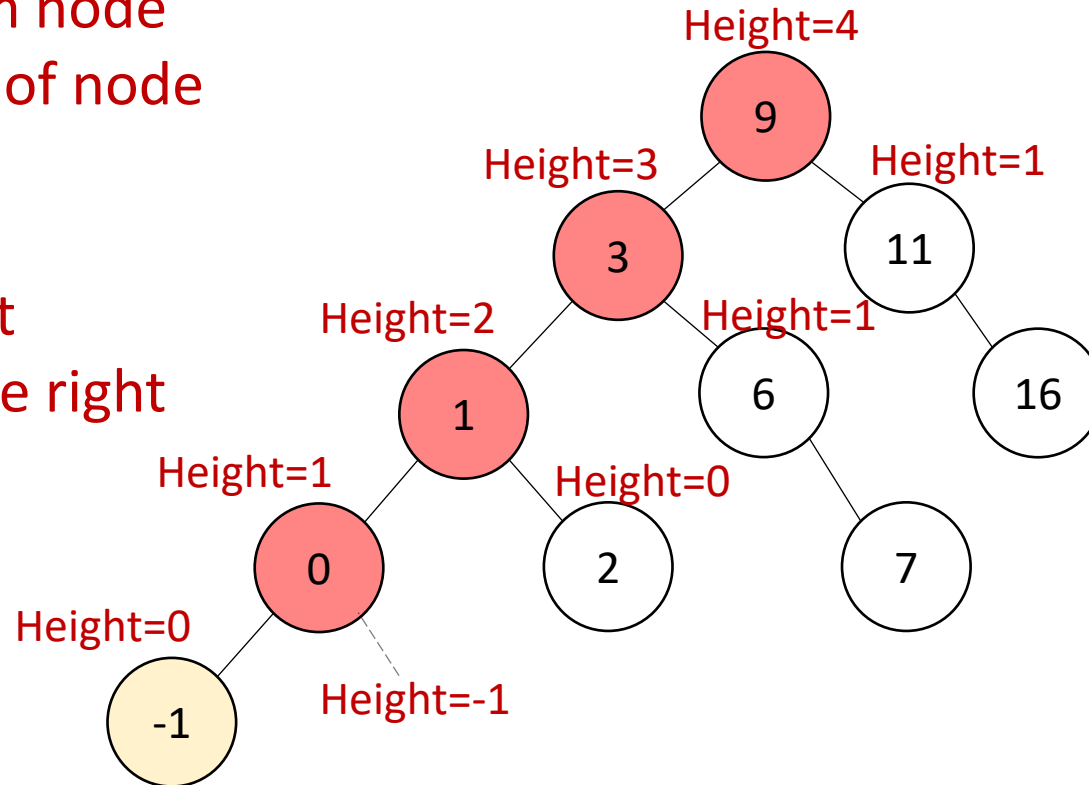
Insert -1, BST Insert Step

Not Balanced!

Node 9 is the “problem node”

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

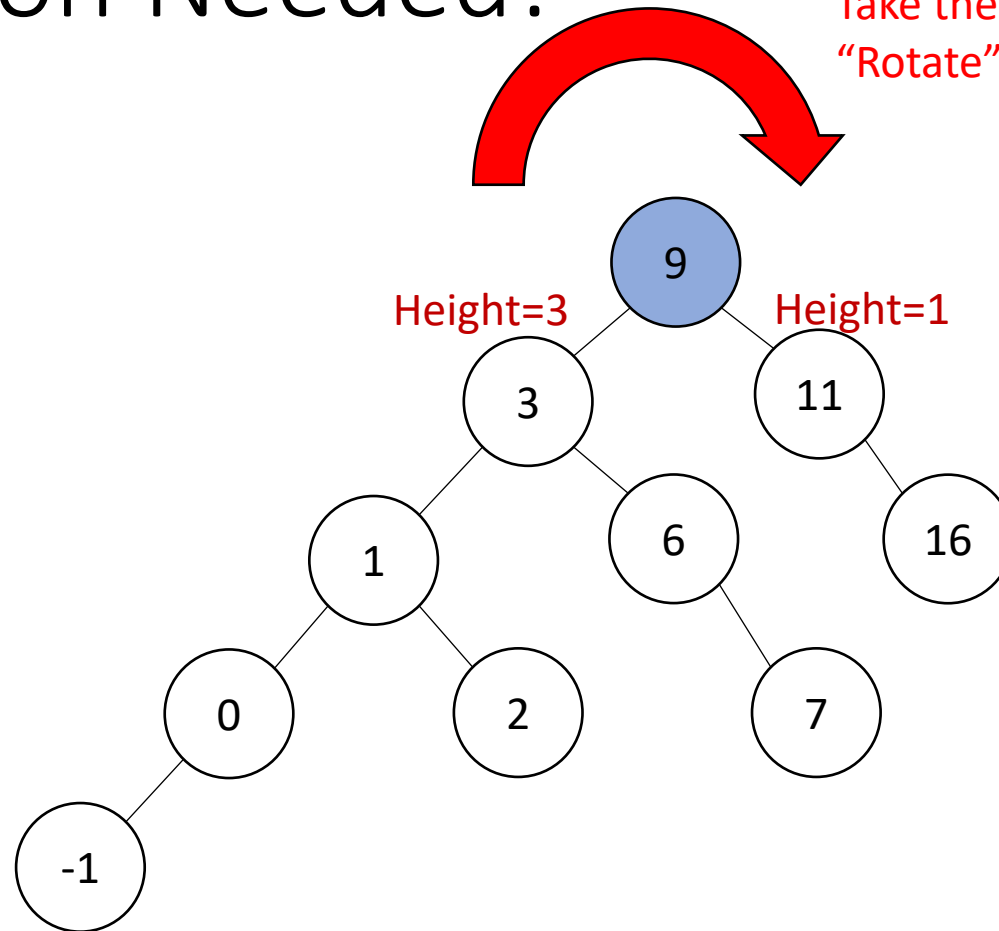
Idea: “shorten” the left subtree, “lengthen” the right



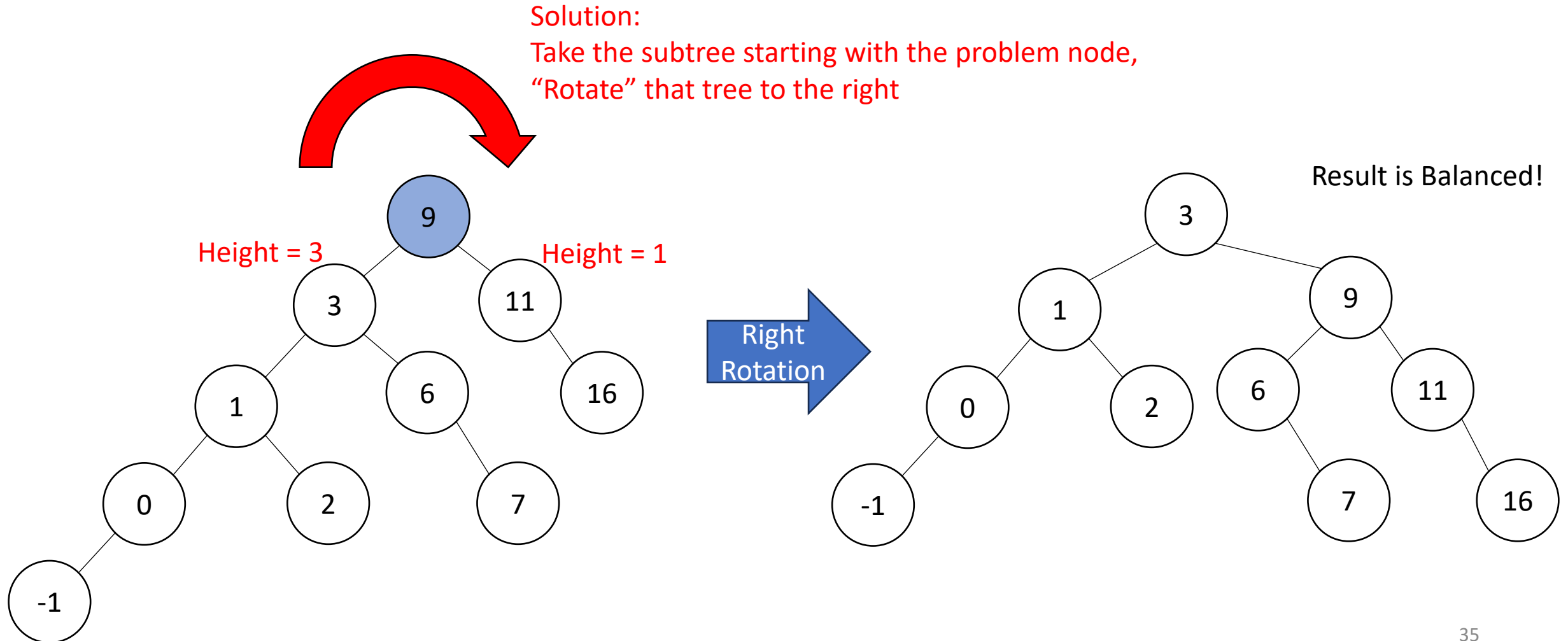
Right Rotation Needed!

Solution:

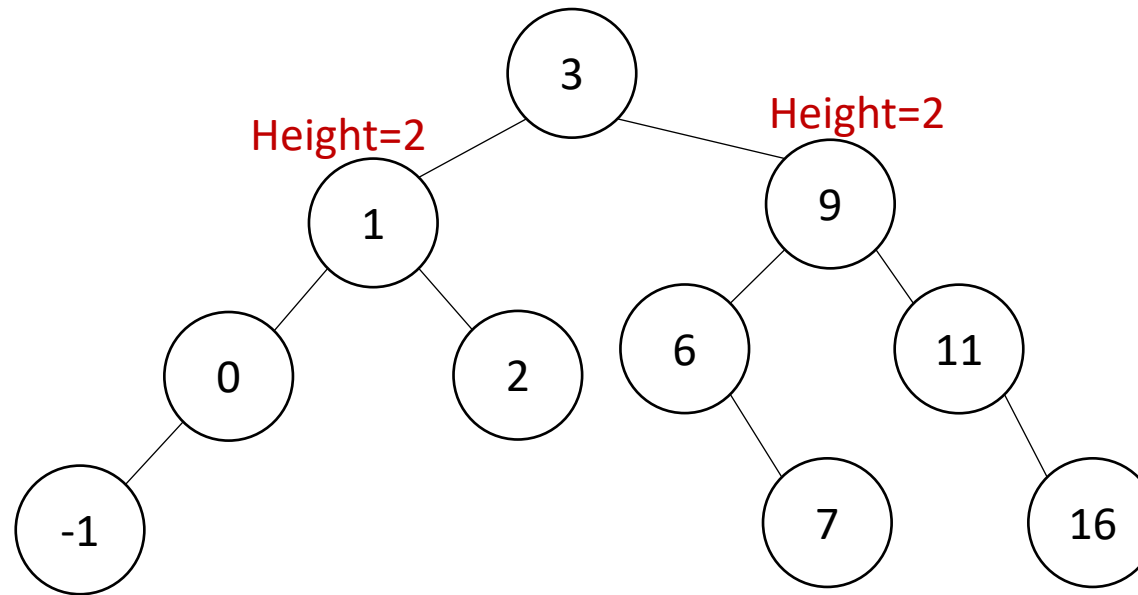
Take the subtree starting with the problem node, "Rotate" that tree to the right



Insert -1, Right rotation process

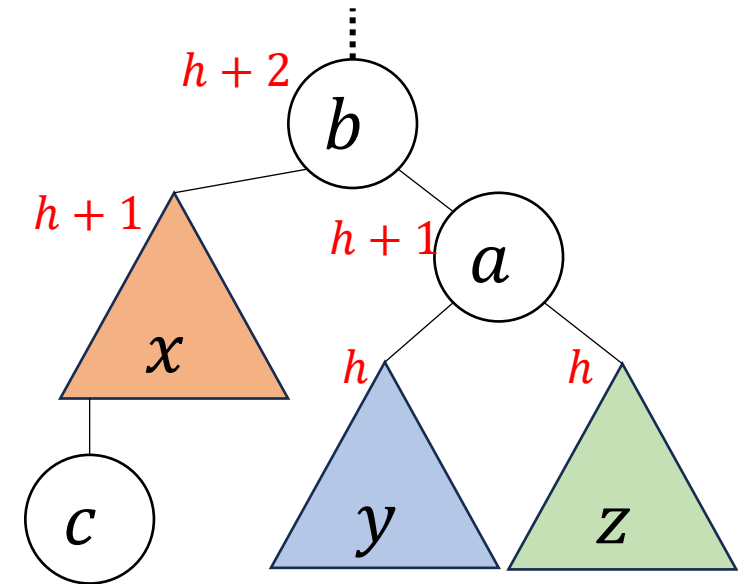
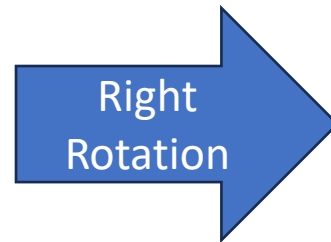
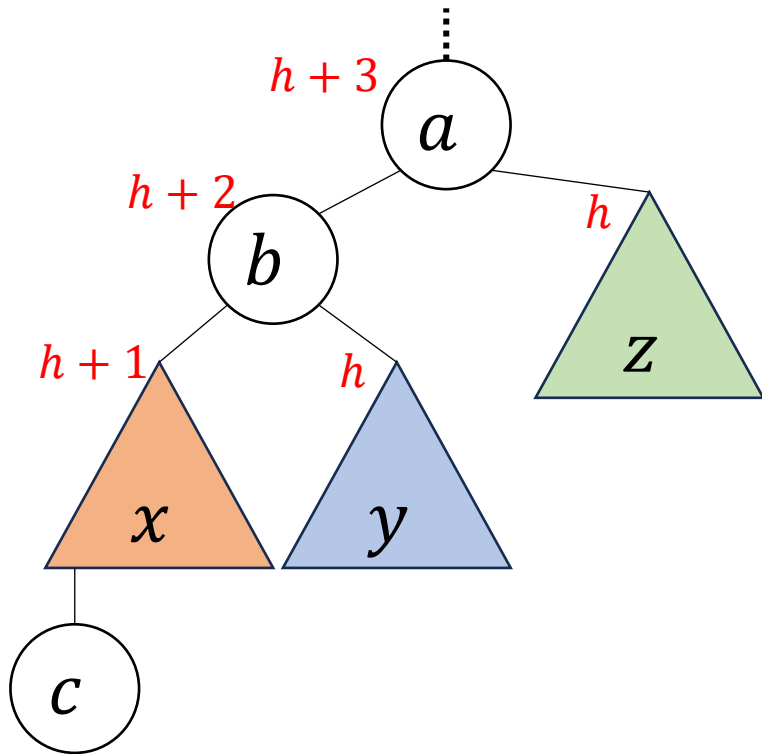


After the Right Rotation, Balanced!



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

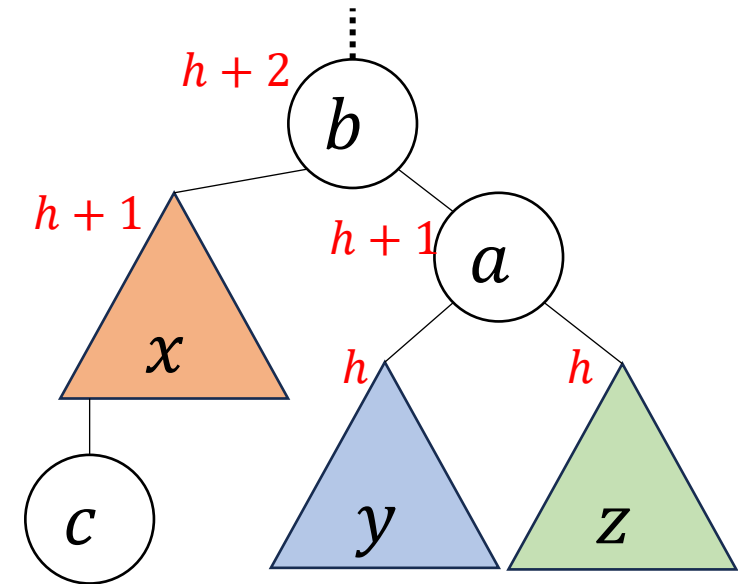
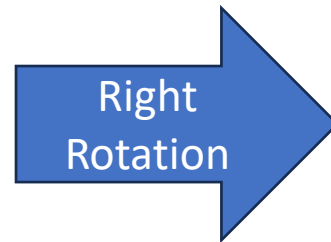
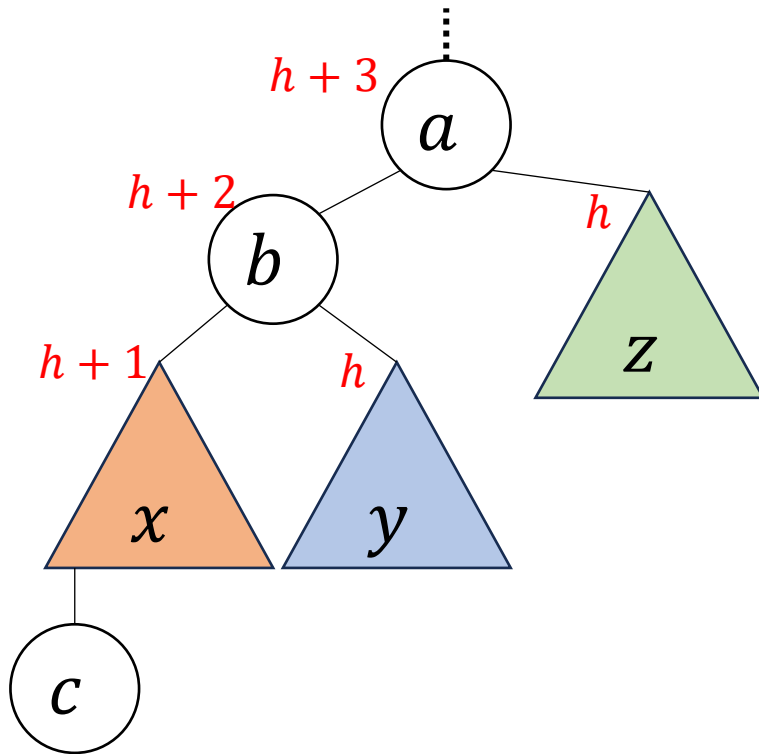
```
b = a.left
```

```
a.left = b.right
```

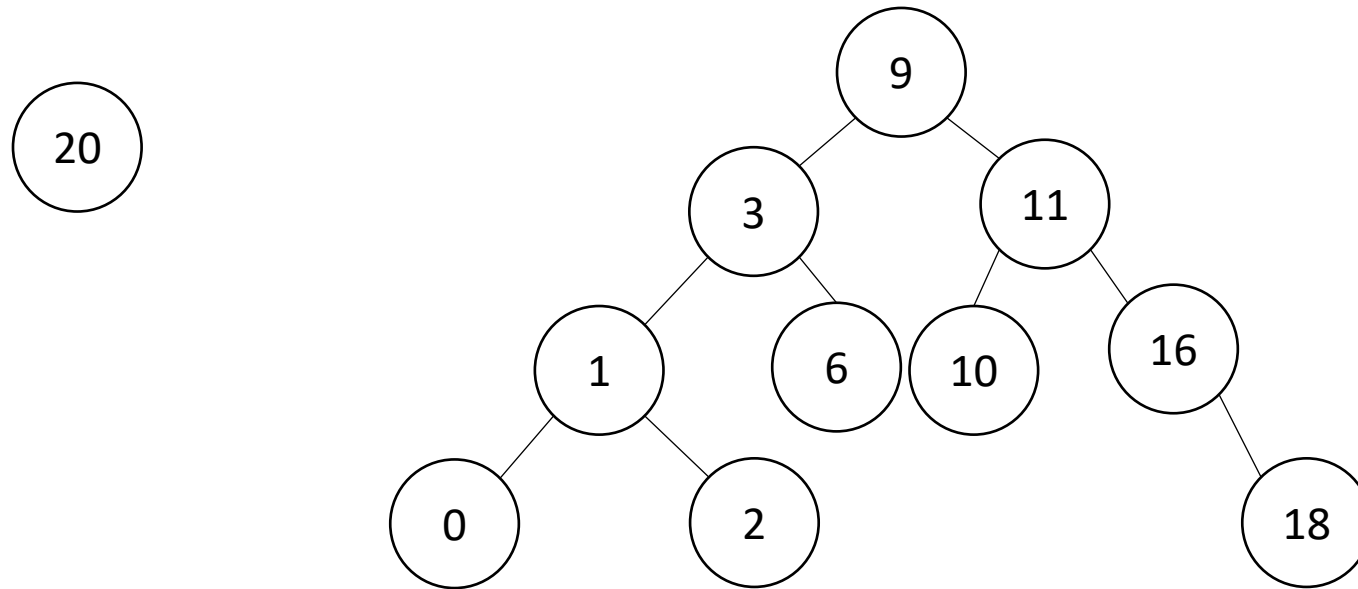
```
b.right = a
```

```
return b
```

Running time: $\Theta(1)$



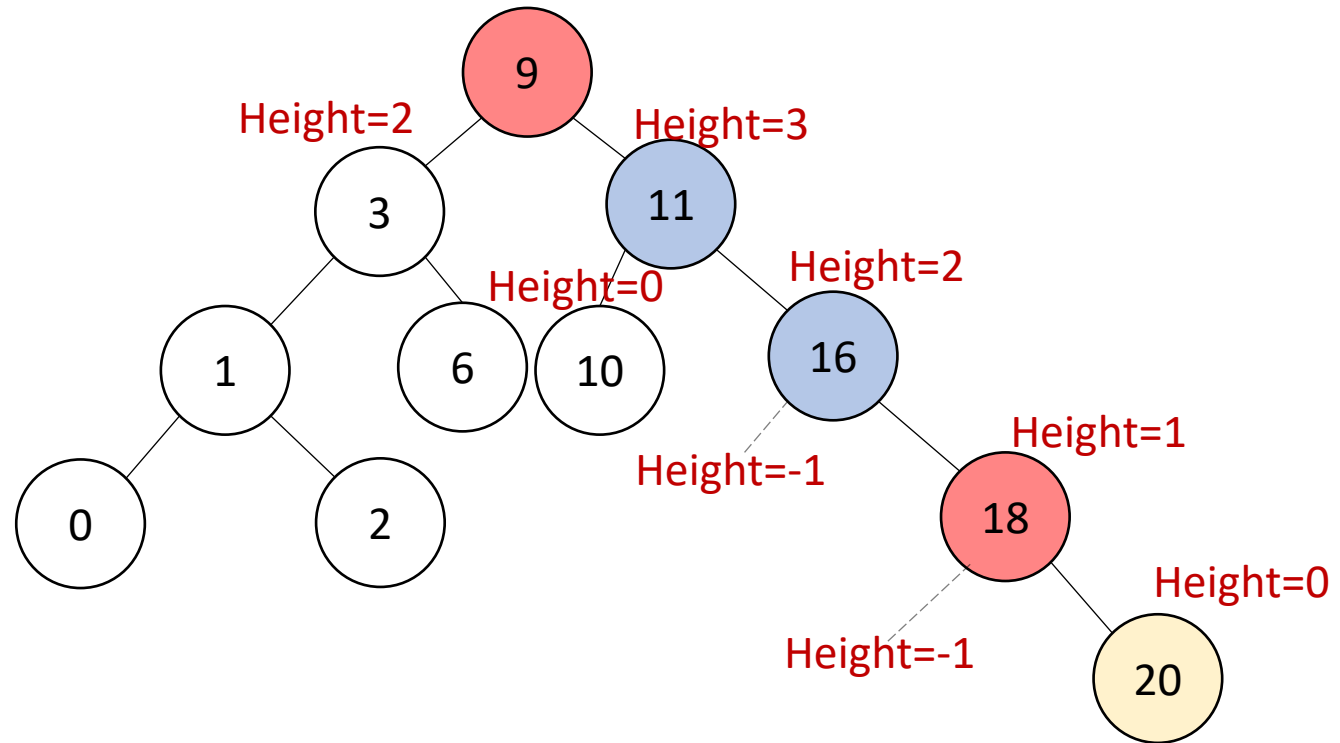
Insert Example (Insert 20)



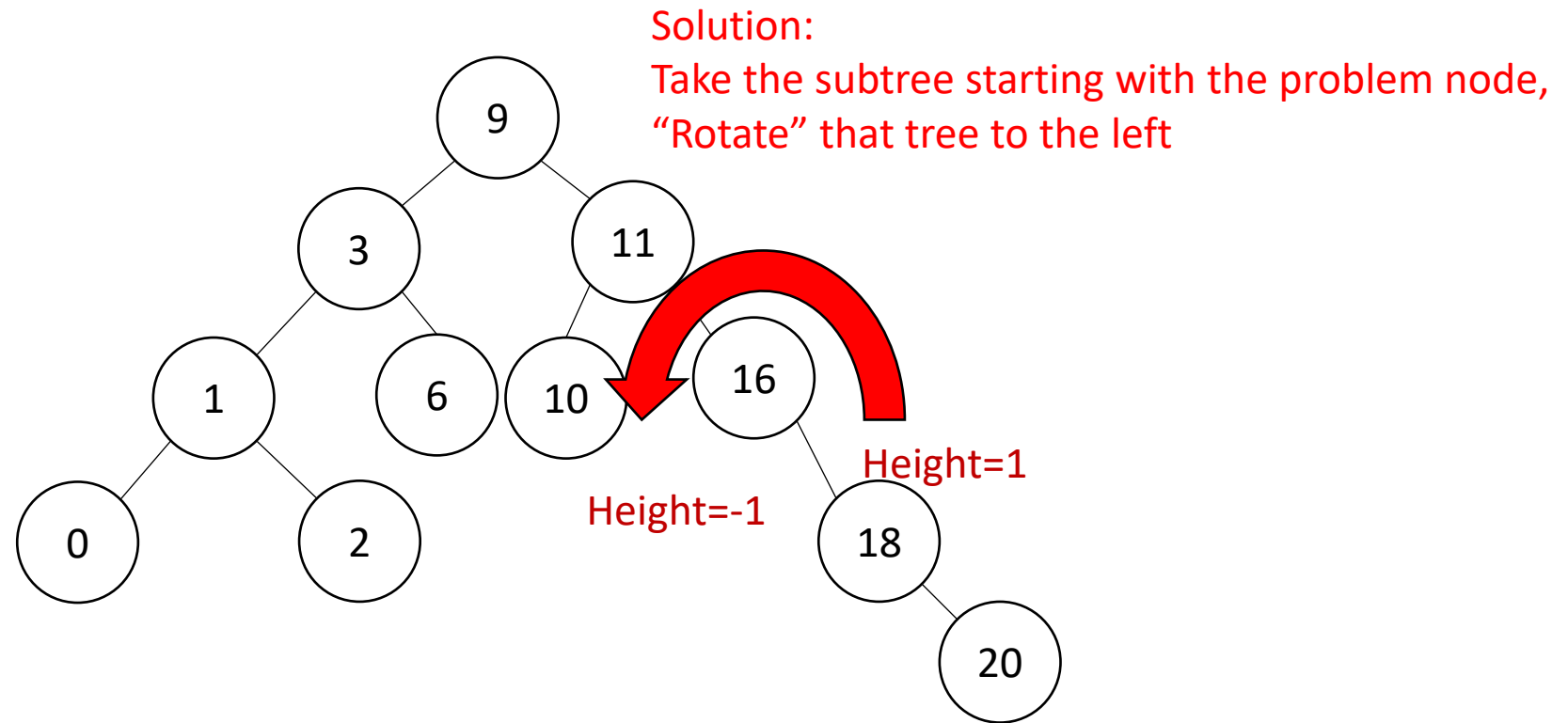
Insert 20: Not Balanced! Multiple Problem Nodes

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!



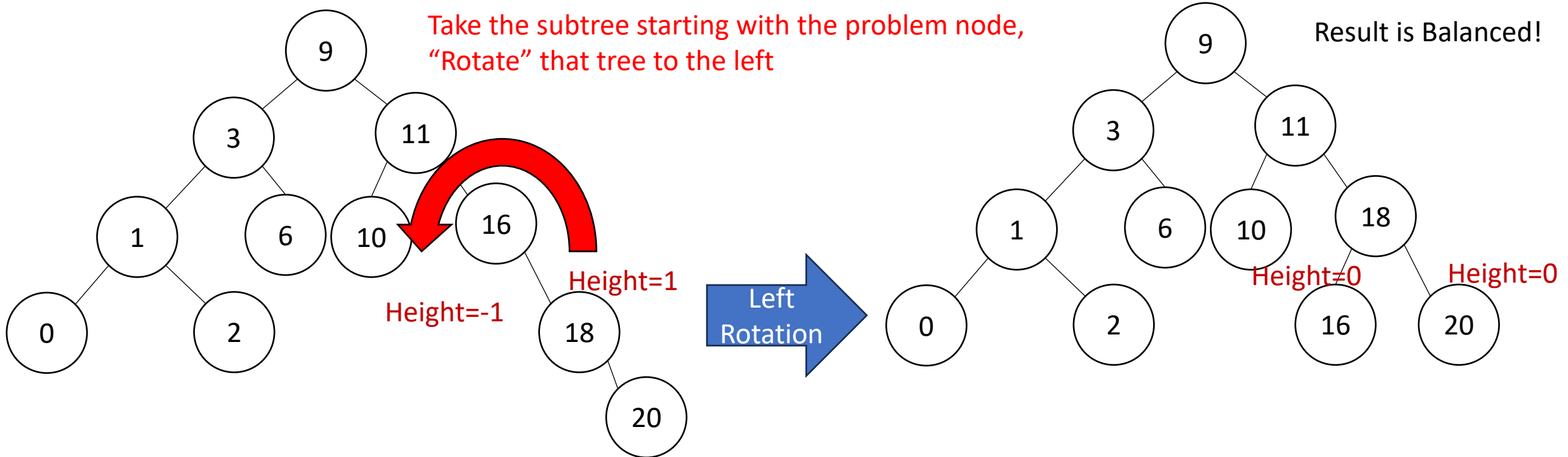
Left Rotation Needed!



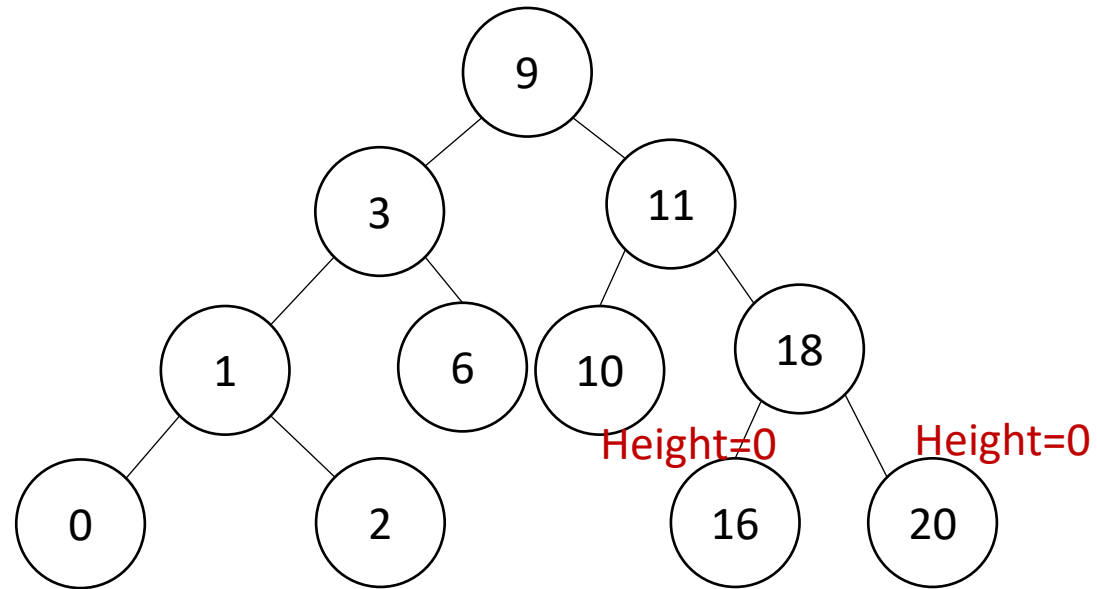
Inserting 20, Left rotation process

Solution:

Take the subtree starting with the problem node,
"Rotate" that tree to the left

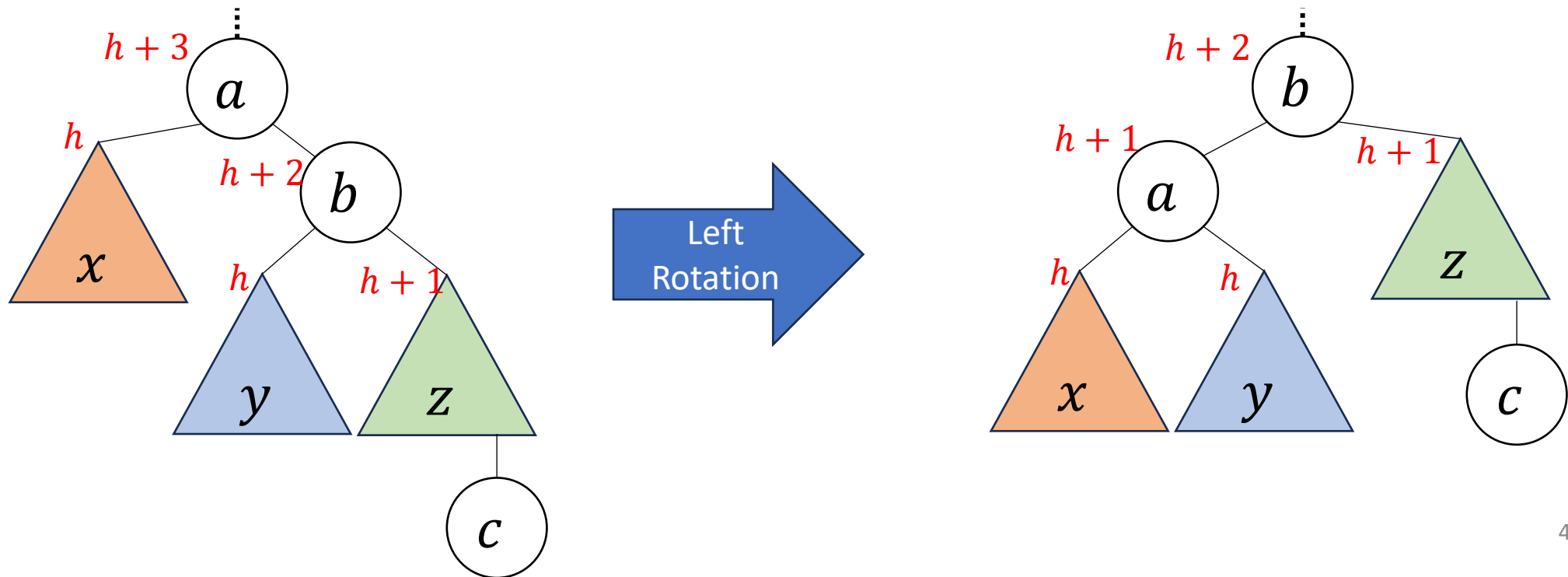


After the Left Rotation, Balanced!



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



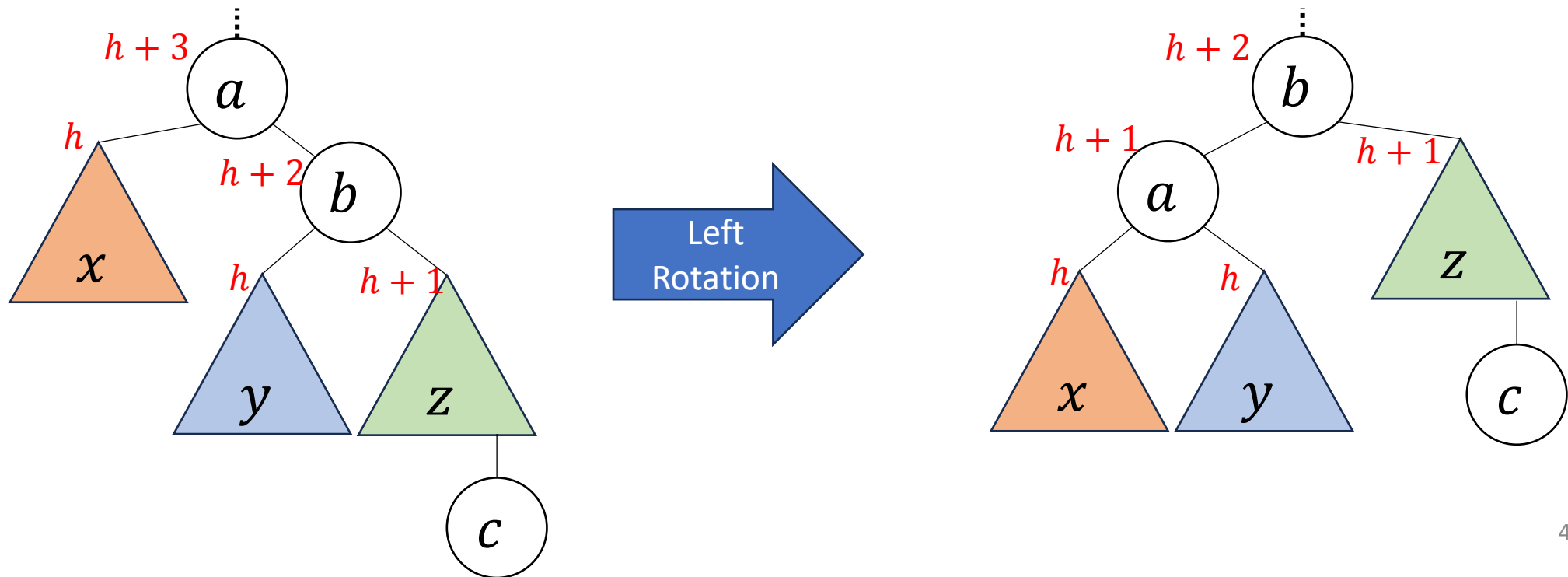
Left Rotation (Implementation)

`b = a.right`

`a.right = b.left`

`b.left = a`

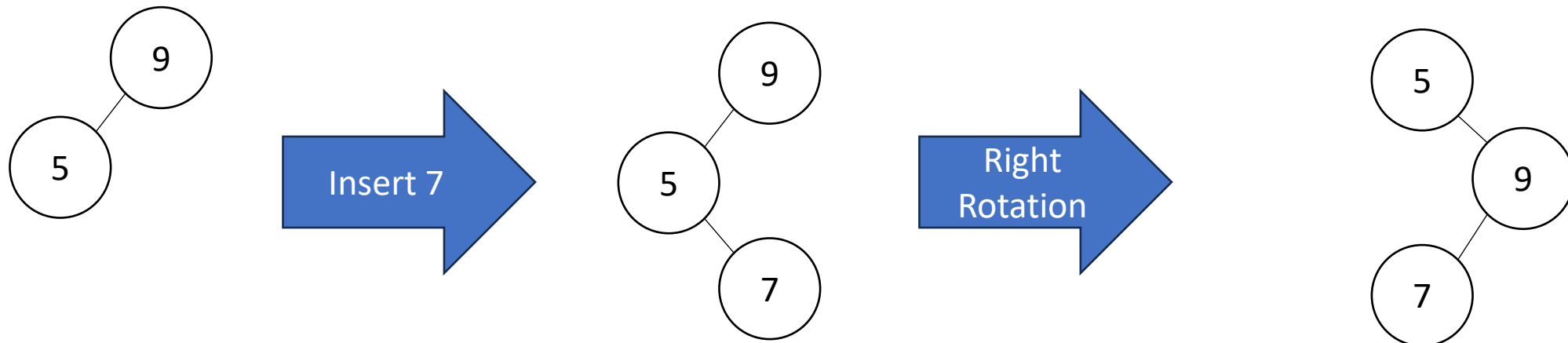
`return b`



Insertion Story So Far

- 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!



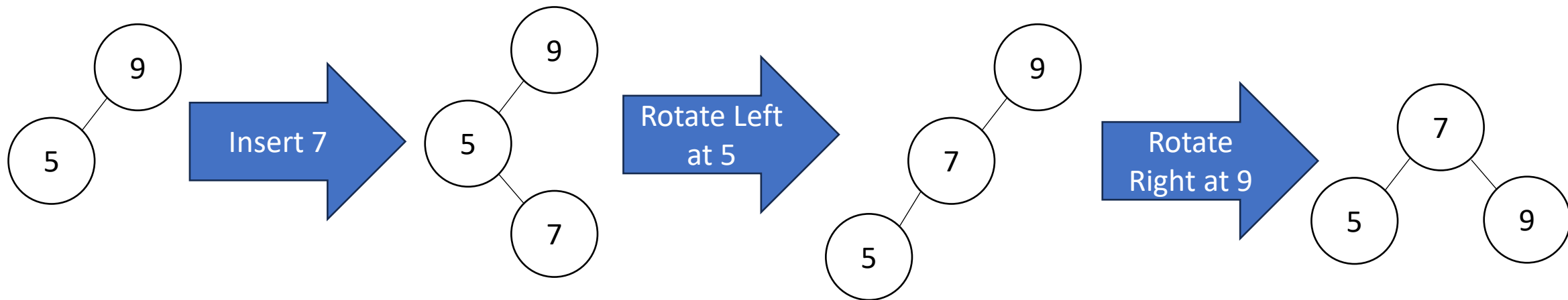
Insertion Cases

- 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!

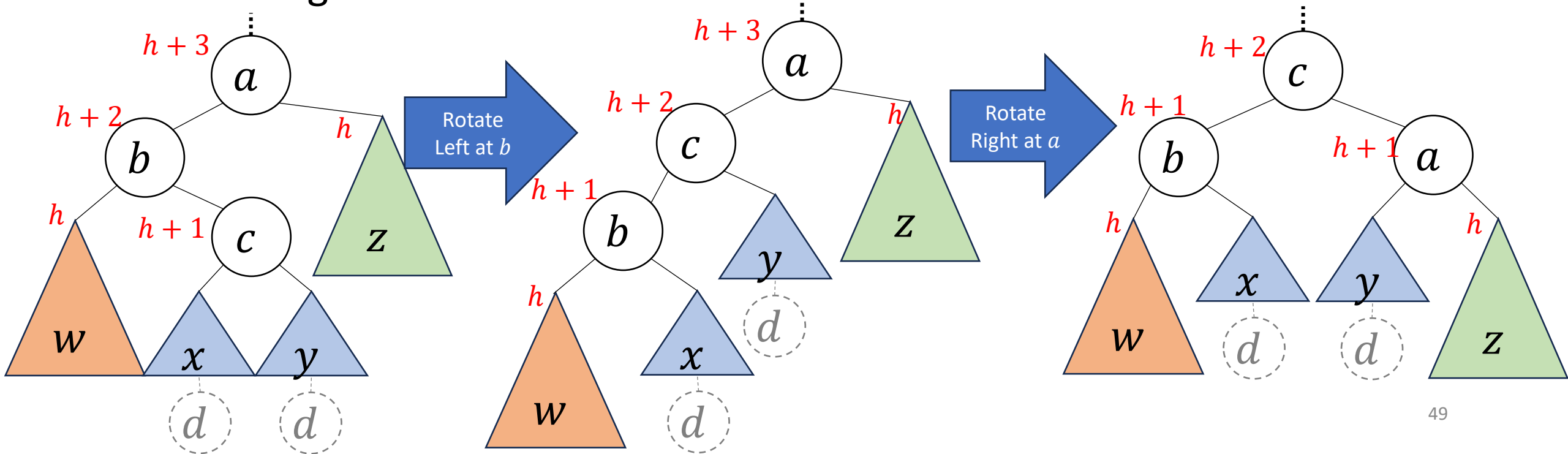
Case LR

- From deepest problem node:
 - Rotate left at the left child
 - Rotate right at the problem node



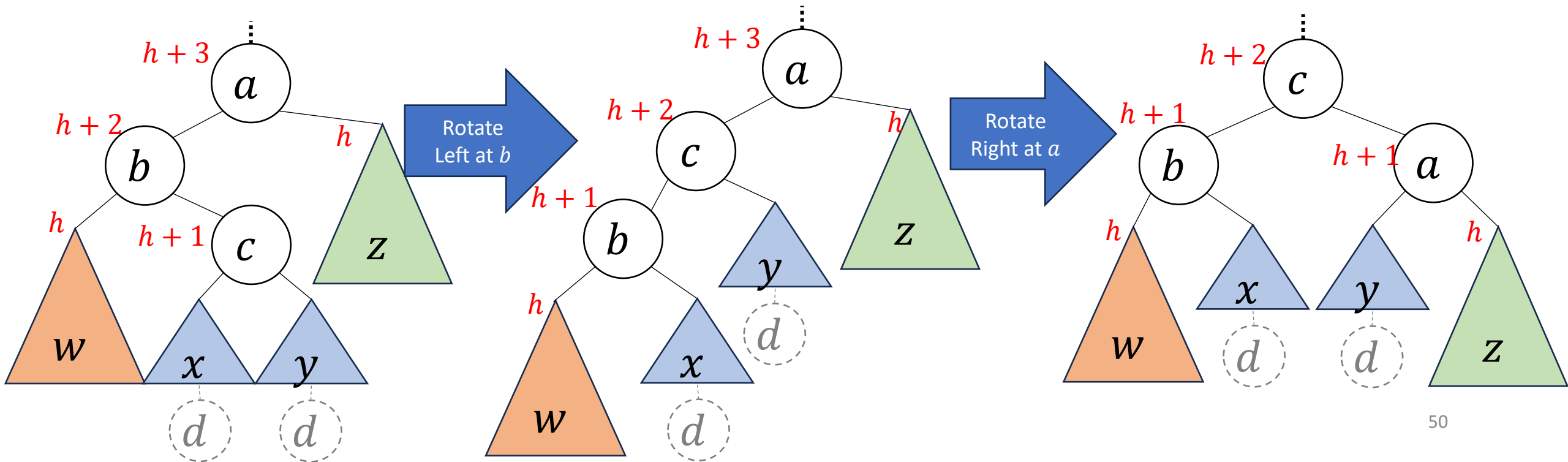
Case LR - Definition

- We just inserted d , node a is the deepest “problem” node
- Imbalance caused by inserting in the left child’s right subtree
- Rotate left at the left child
- Rotate right at the unbalanced node



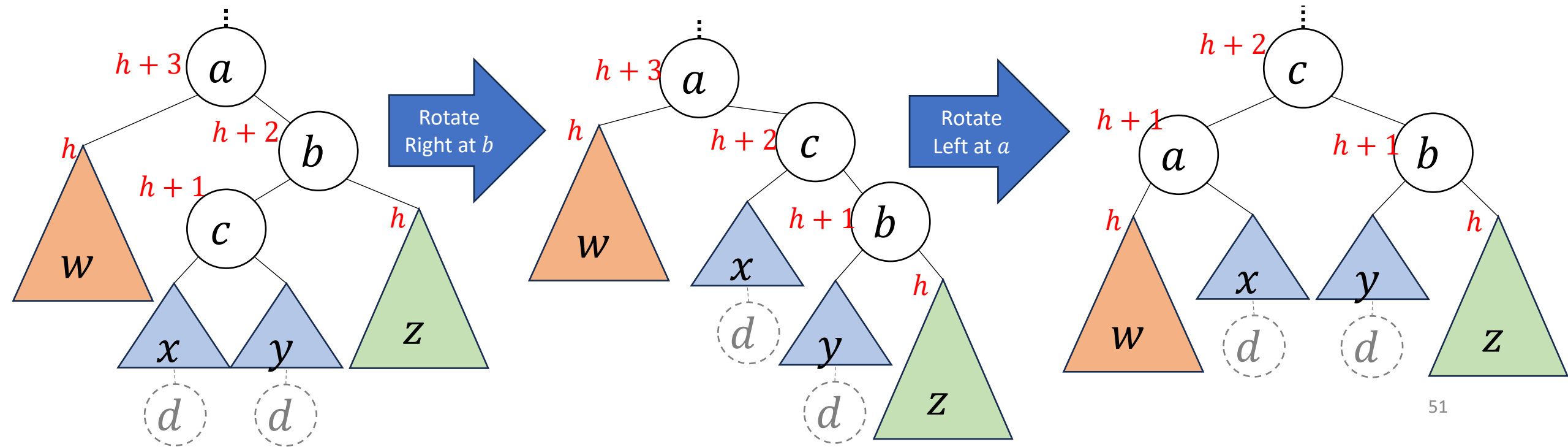
Case LR Implementation

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



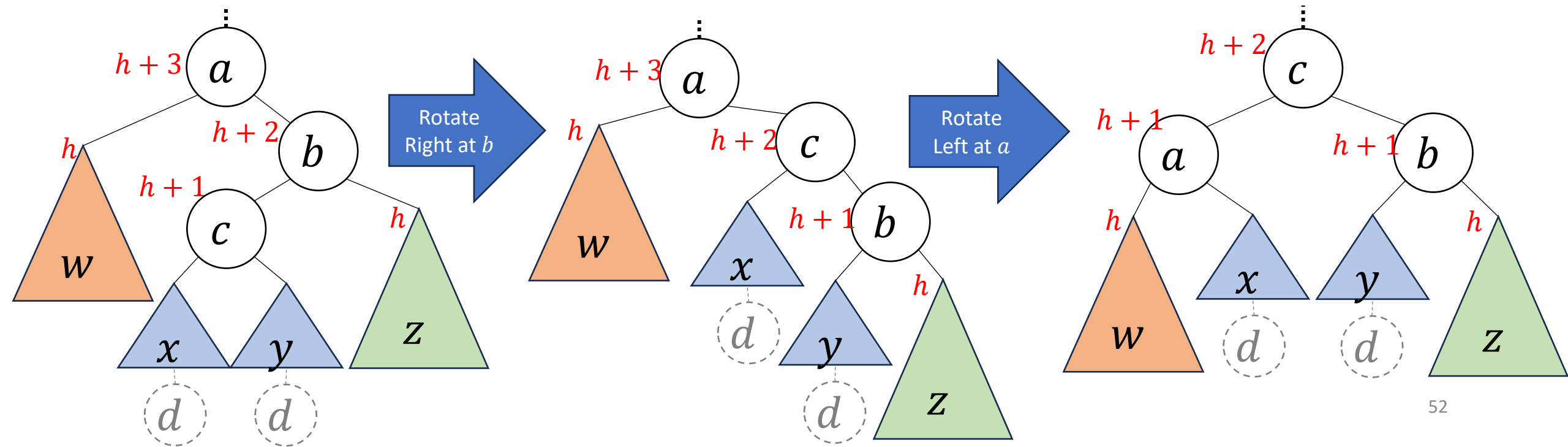
Case RL - Definition

- We just inserted d , node a is the deepest “problem” node
- Imbalance caused by inserting in the right child’s left subtree
- Rotate right at the right child
- Rotate left at the unbalanced node



Case RL Implementation

```
b = a.right  
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 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;  
}
```

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 $h - 2$**
- $M(h) = \textcolor{red}{M(h - 1)} + \textcolor{blue}{M(h - 2)} + 1$

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} 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 $\Omega(\log n)$