

CSE 332 Winter 2026

Lecture 8: AVL Trees

Nathan Brunelle

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

Dictionary (Map) ADT

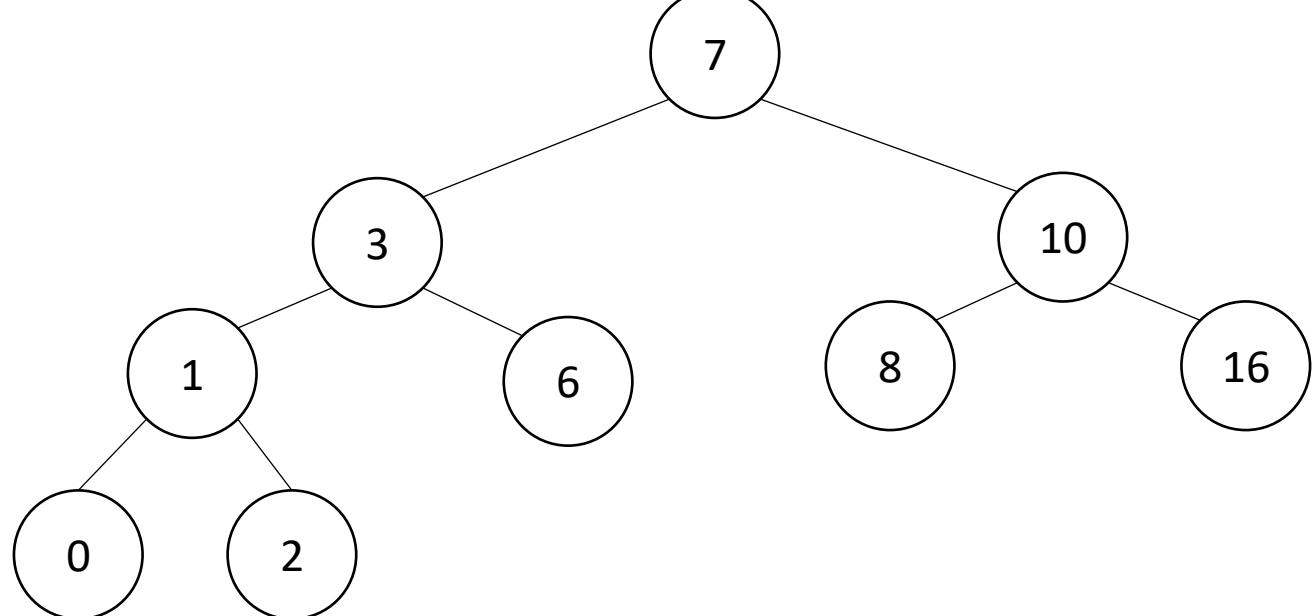
- **Contents:**
 - Sets of key+value pairs
 - Keys must be comparable
- **Operations:**
 - **insert(key, value)**
 - Adds the (key,value) pair into the dictionary
 - If the key already has a value, overwrite the old value
 - Consequence: Keys cannot be repeated
 - **find(key)**
 - Returns the value associated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

Naïve attempts

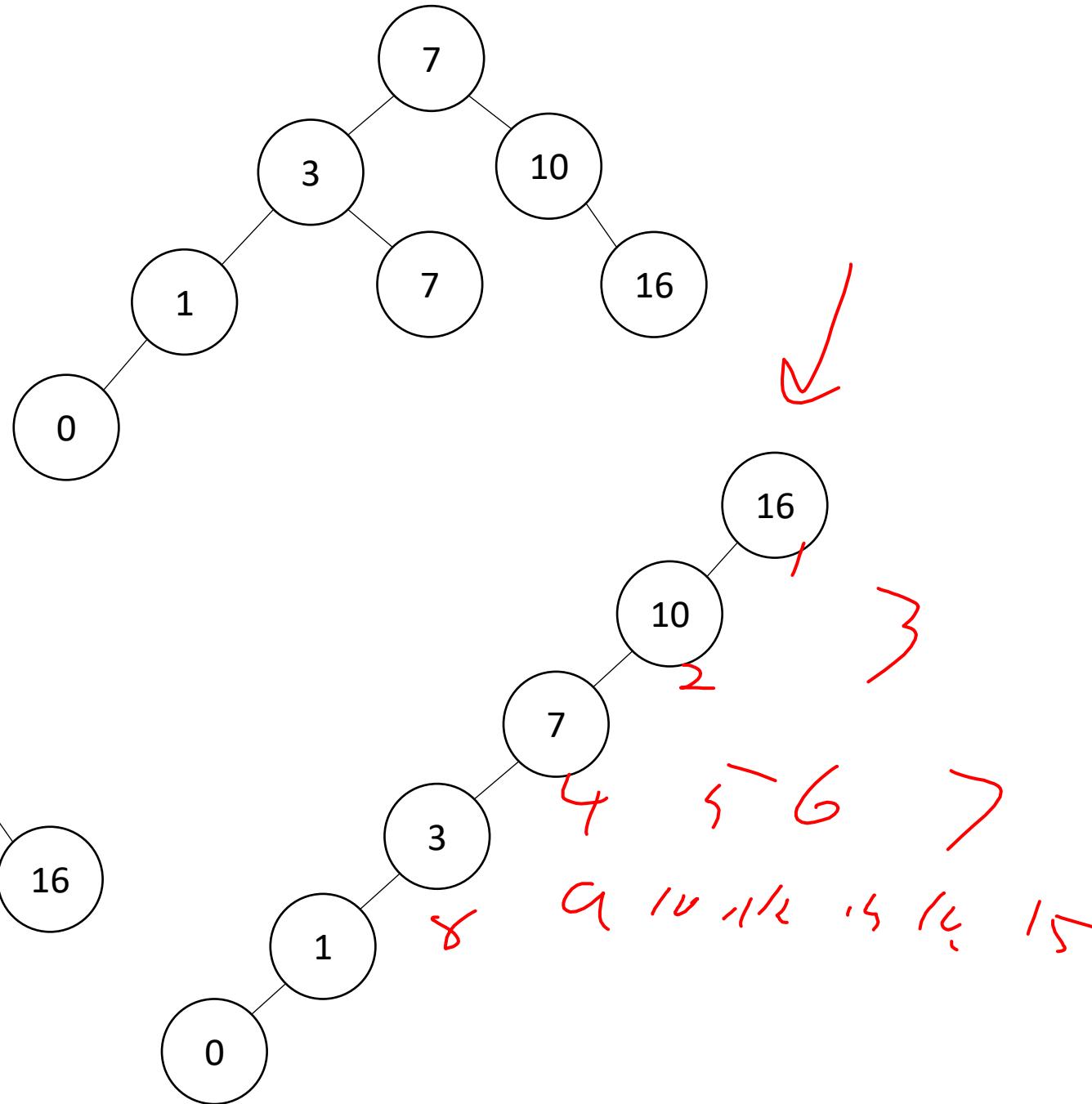
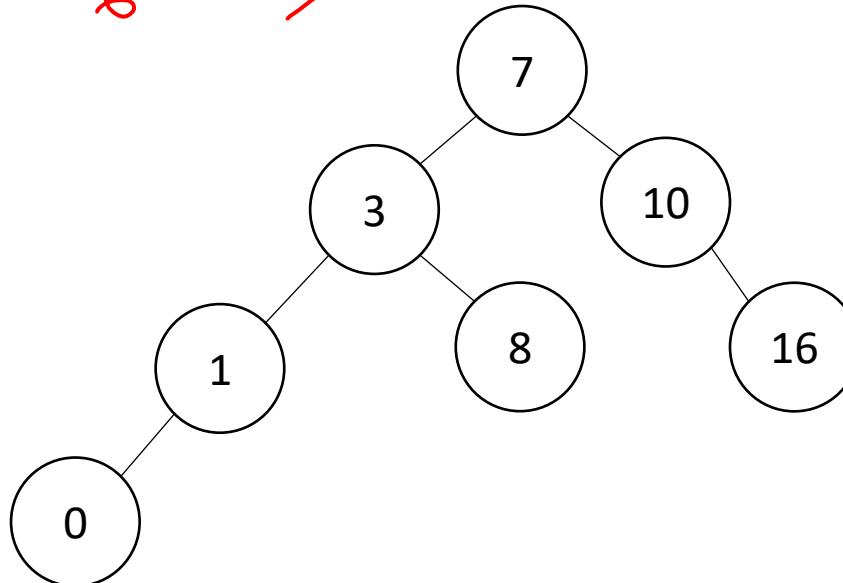
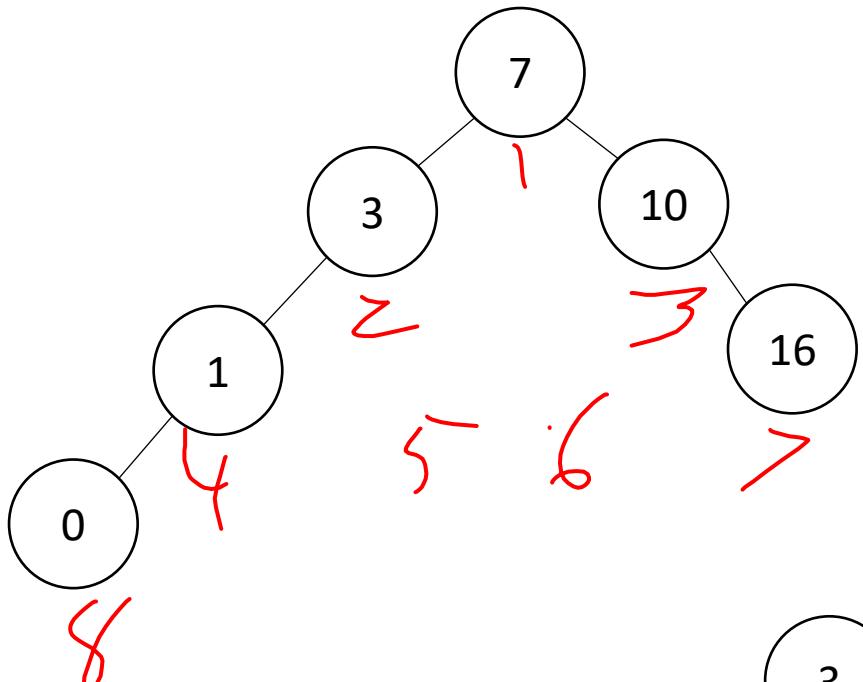
Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Heap	$\Theta(n)$	$\Theta(n)$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

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

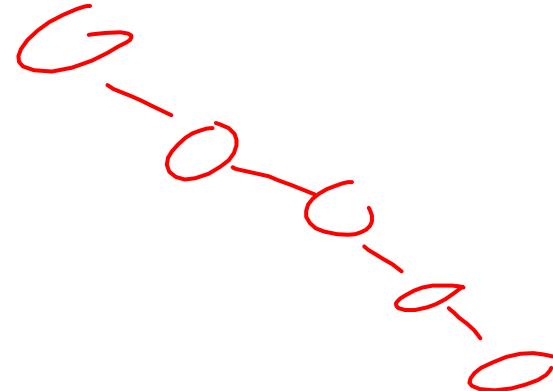


Are these BSTs?



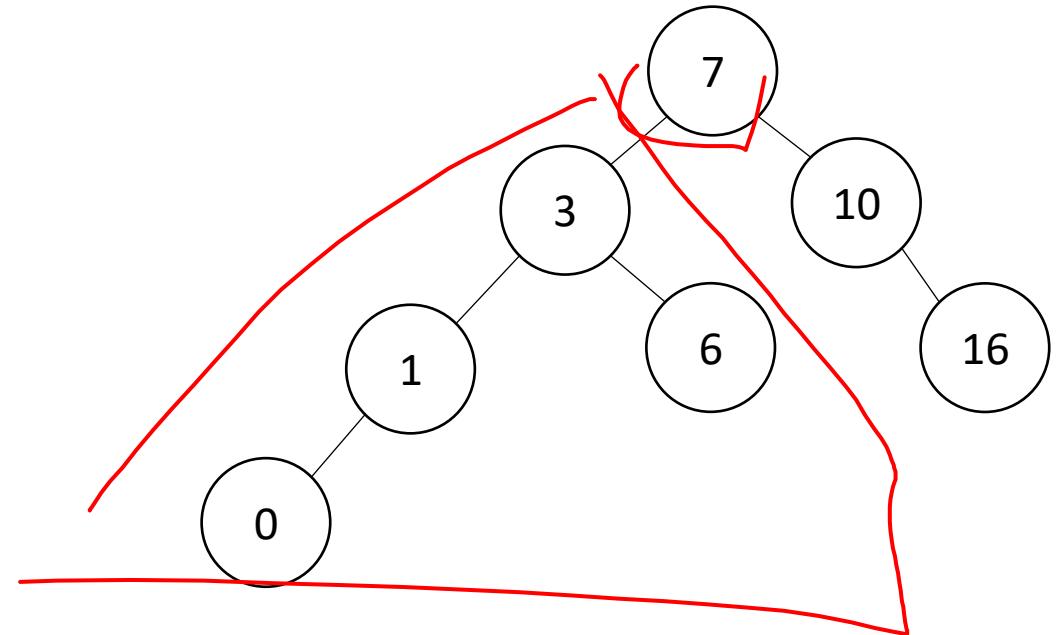
Aside: Why not use an array?

- We represented a heap using an array, finding children/parents by index
- We will represent BSTs with nodes and references. Why?
 - We might have “gaps” in our tree
 - Memory!
 - 2^n



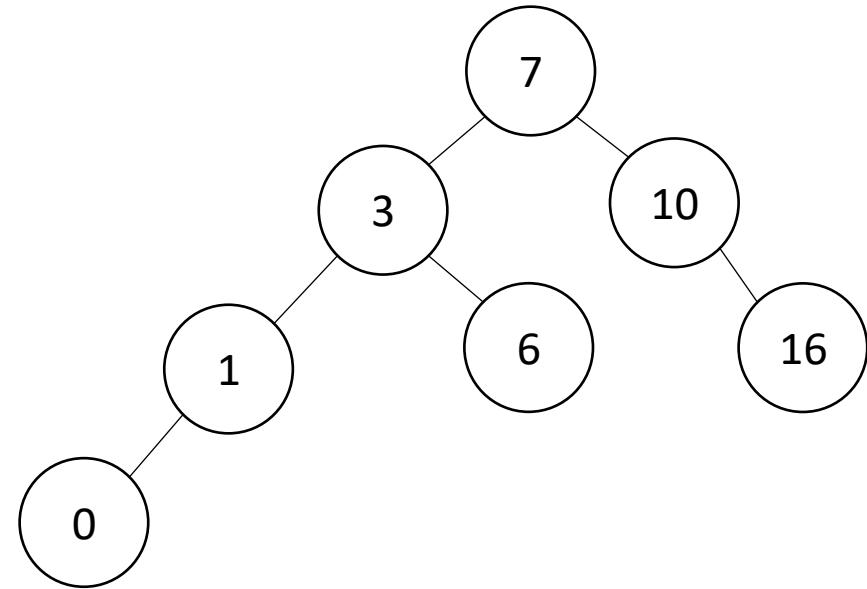
Find Operation (recursive)

```
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;  
}
```



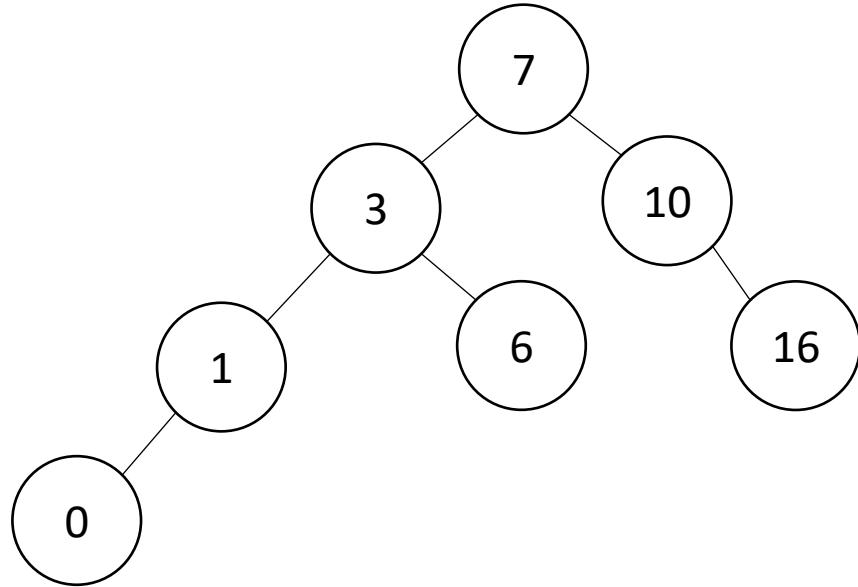
Find Operation (iterative)

```
find(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 Null;  
    }  
    return root.value;  
}
```



Insert Operation (recursive)

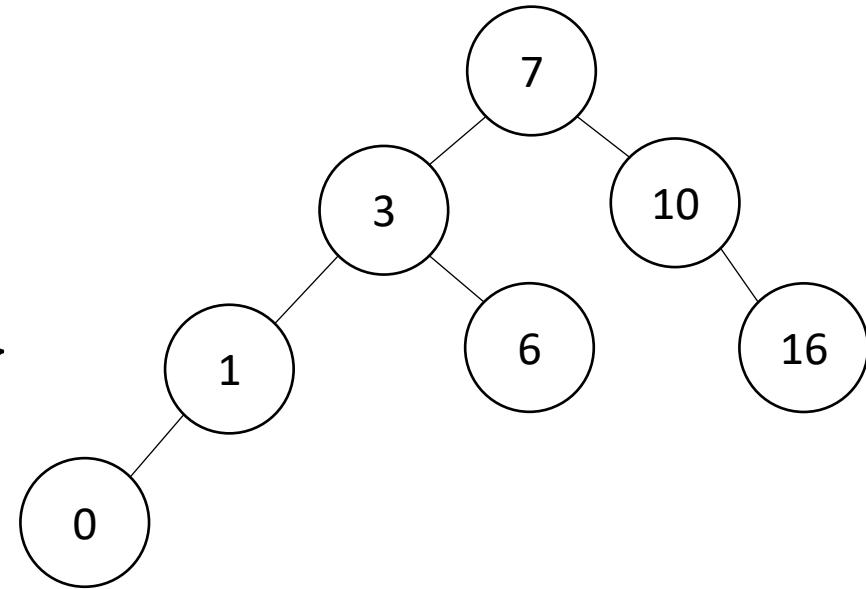
```
insert(key, value, root){  
    root = insertHelper(key, value, root);  
}  
insertHelper(key, value, root){  
    if(root == null)  
        return new Node(key, value);  
    if (root.key < key)  
        root.right = insertHelper(key, value, root.right);  
    else  
        root.left = insertHelper(key,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



Note: Insert happens only at the leaves!

Insert Operation (iterative)

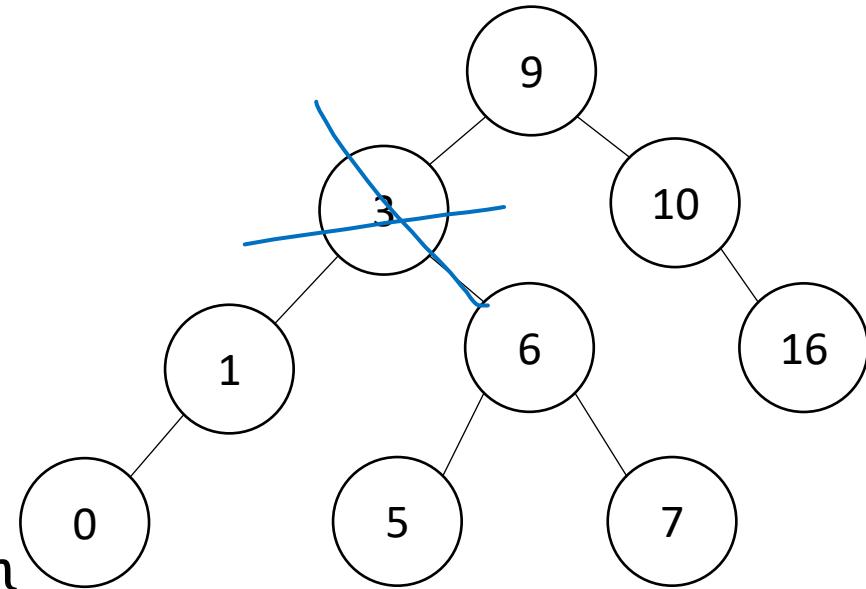
```
insert(key, value, root){  
    if (root == Null){ this.root = new Node(key, value); }  
    parent = Null;  
    while (root != Null && key != root.key){  
        parent = root;  
        if (key < root.key){ root = root.left; }  
        else if (key > root.key){ root = root.right; }  
    }  
    if (root != Null){ root.value = value; }  
    else if (key < parent.key){ parent.left = new Node(key, value); }  
    else{ parent.right = new Node (key, value); }  
}
```



Note: Insert happens only at the leaves!

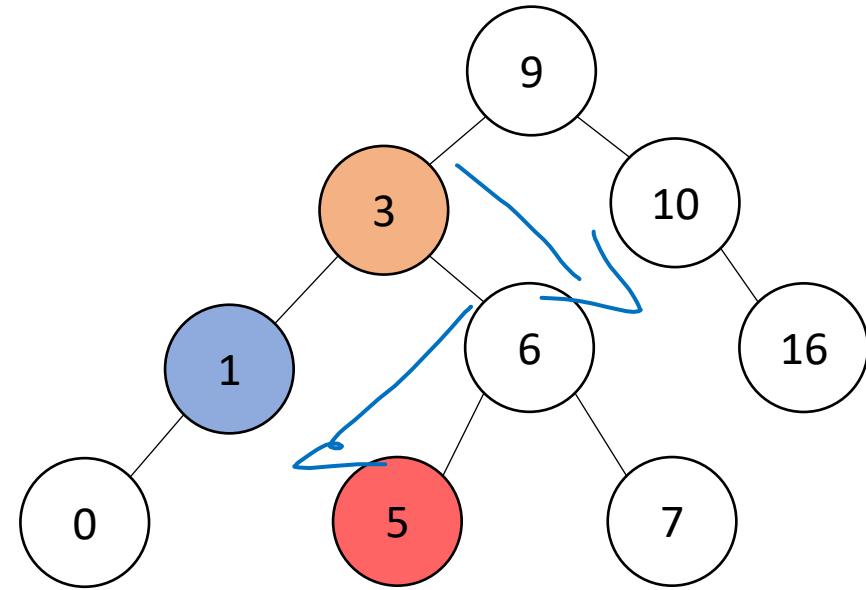
Delete Operation (iterative)

```
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; }  
    // Now root is the node to delete, what happens next?  
}
```



Delete – 3 Cases

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

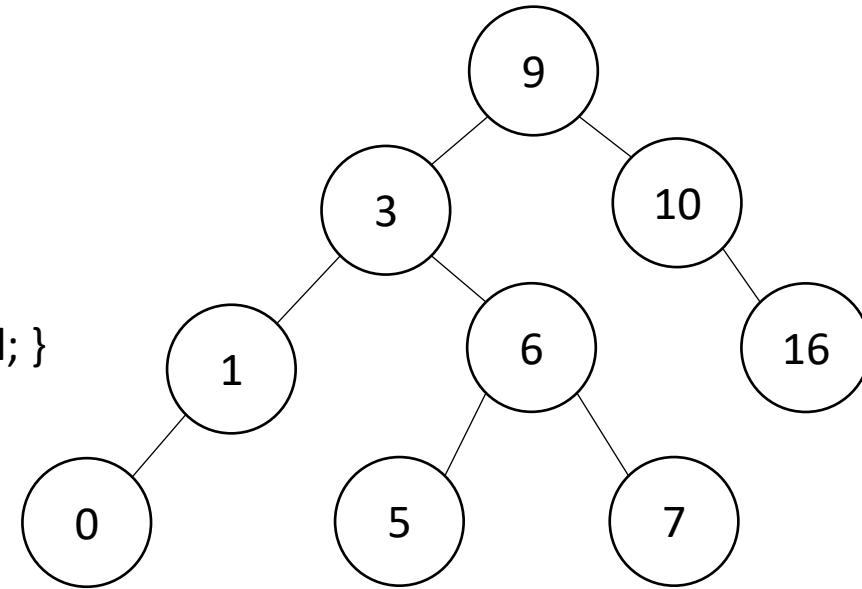


Finding the Max and Min

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

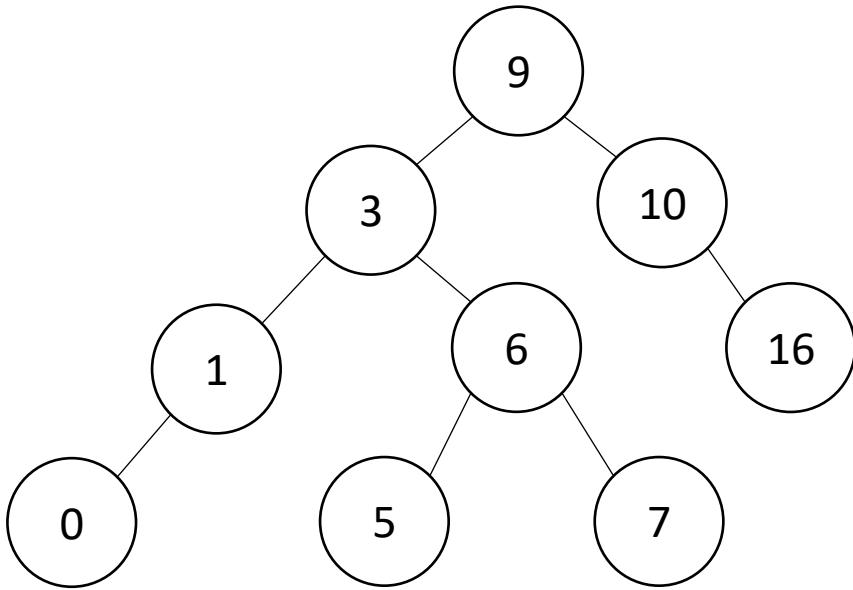
```
maxNode(root){  
    if (root == Null){ return Null; }  
    while (root.right != Null){  
        root = root.right;  
    }  
    return root;  
}
```

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



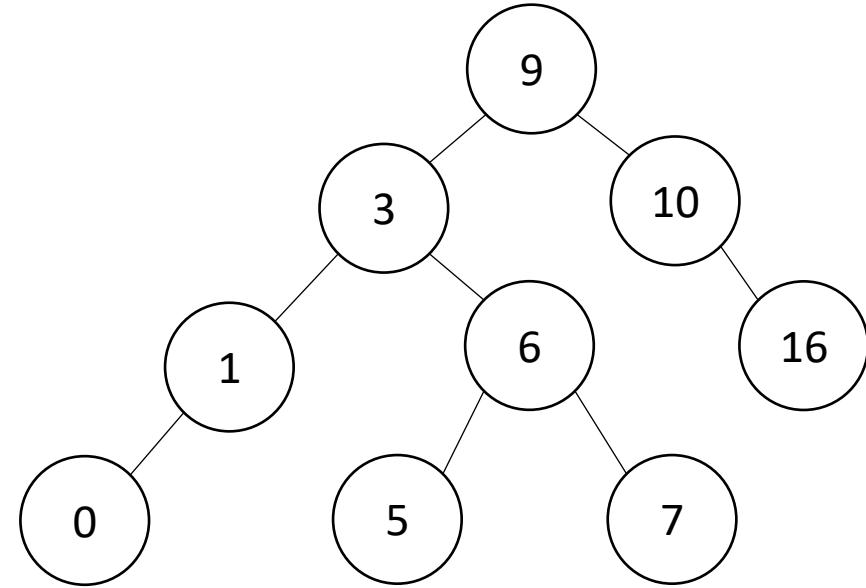
Delete Operation (iterative)

```
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?
 - $\Theta(n)$

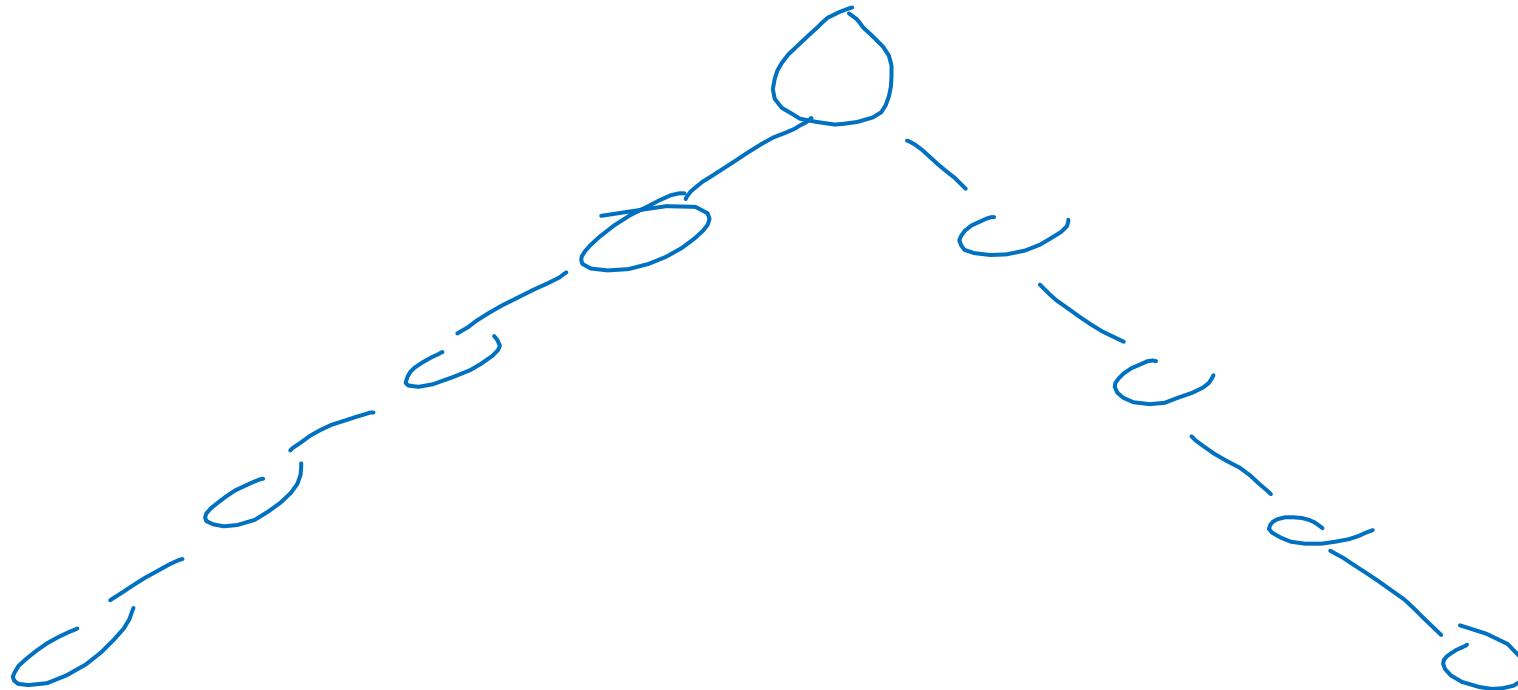
Improving the worst case

- 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

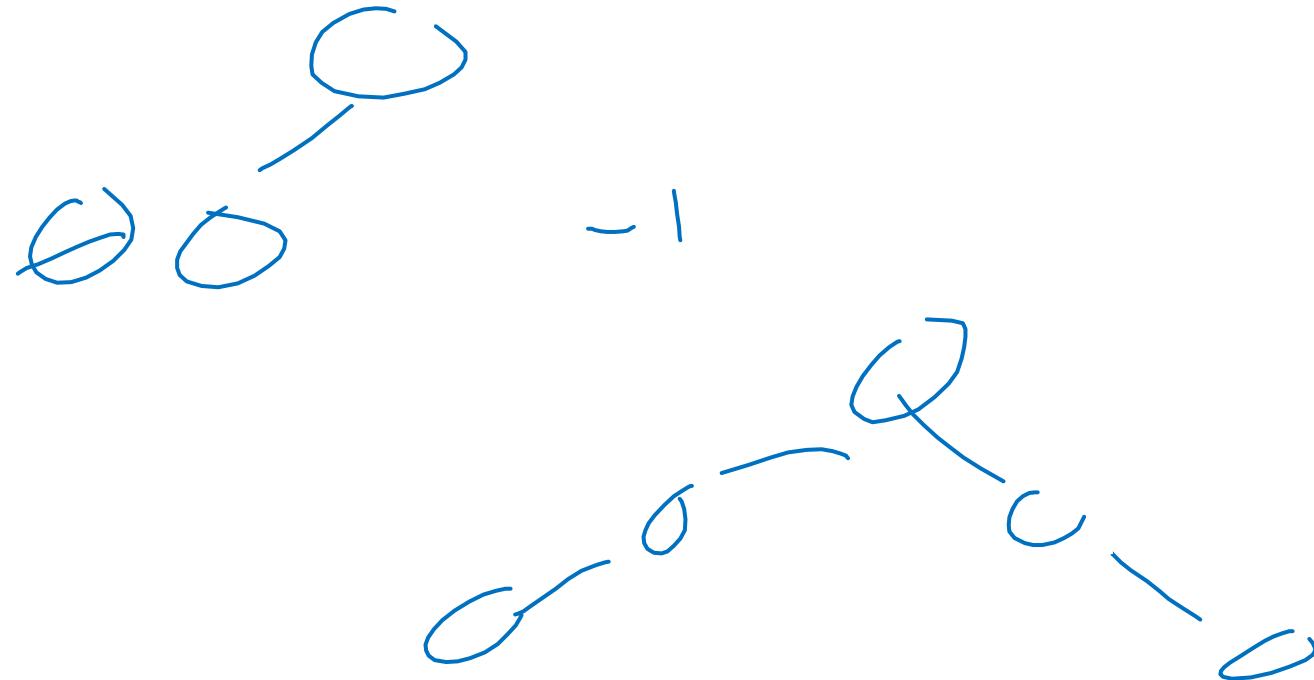
“Balanced” Binary Search Trees

- 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!

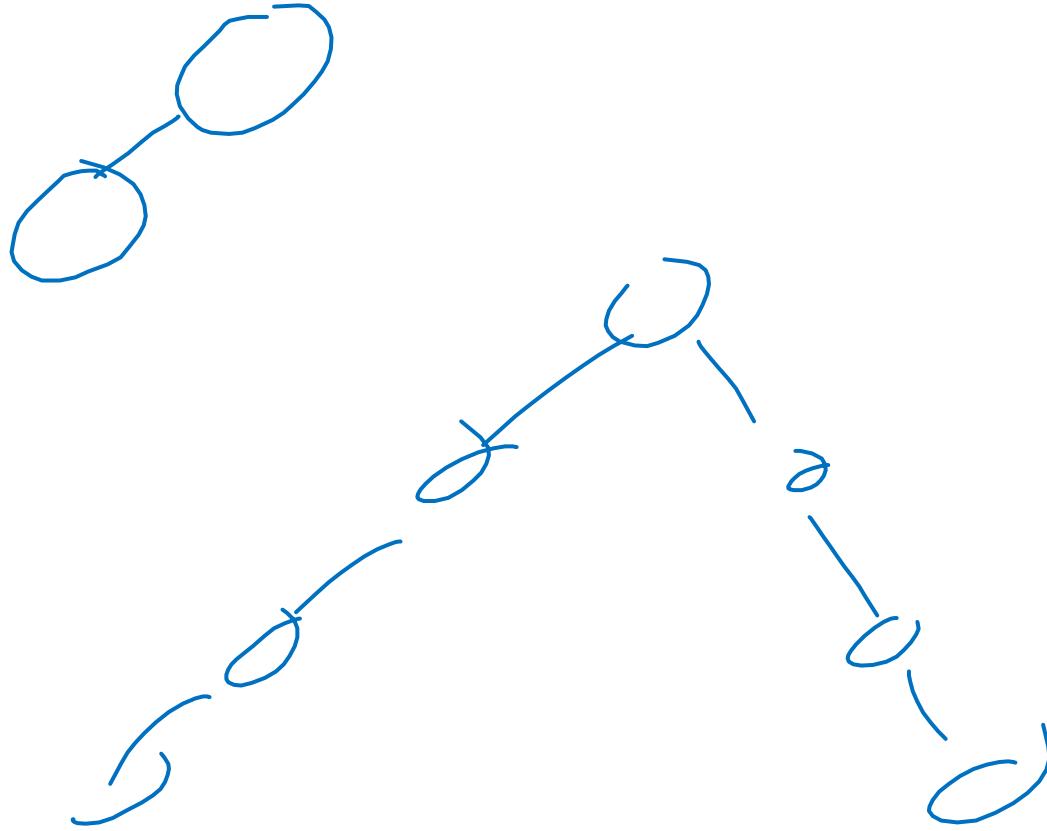
Idea 1: Both Subtrees of Root have same # Nodes



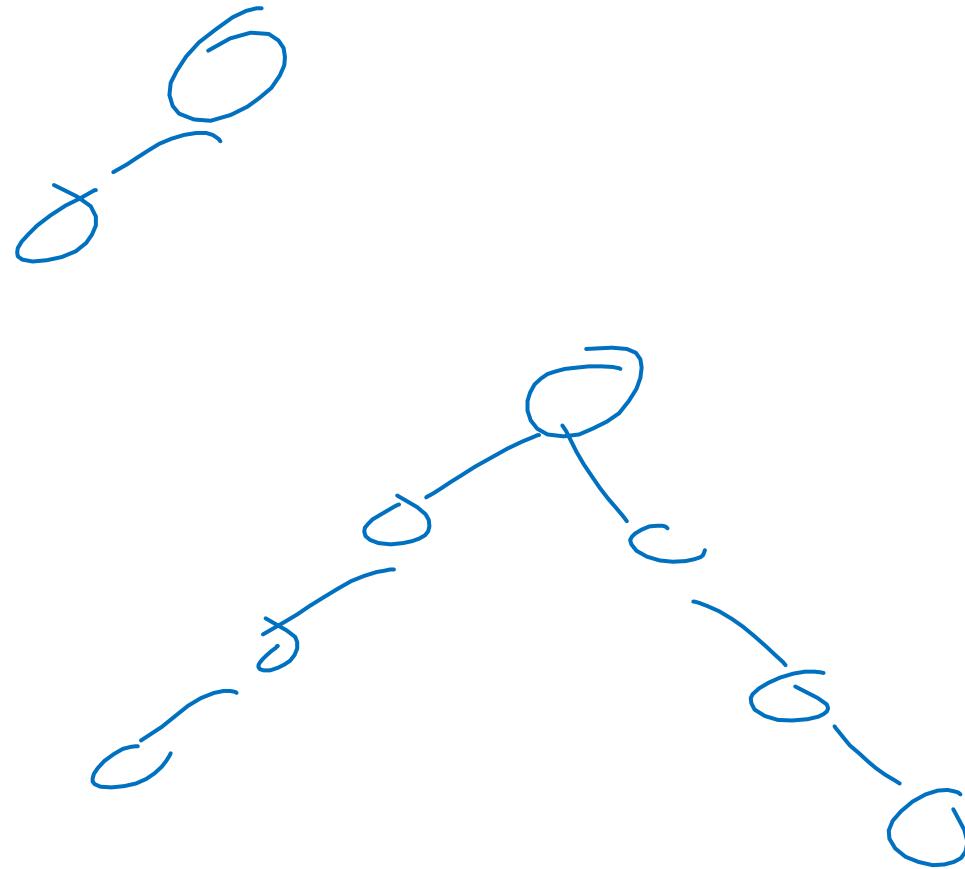
Idea 2: Both Subtrees of Root have same height



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



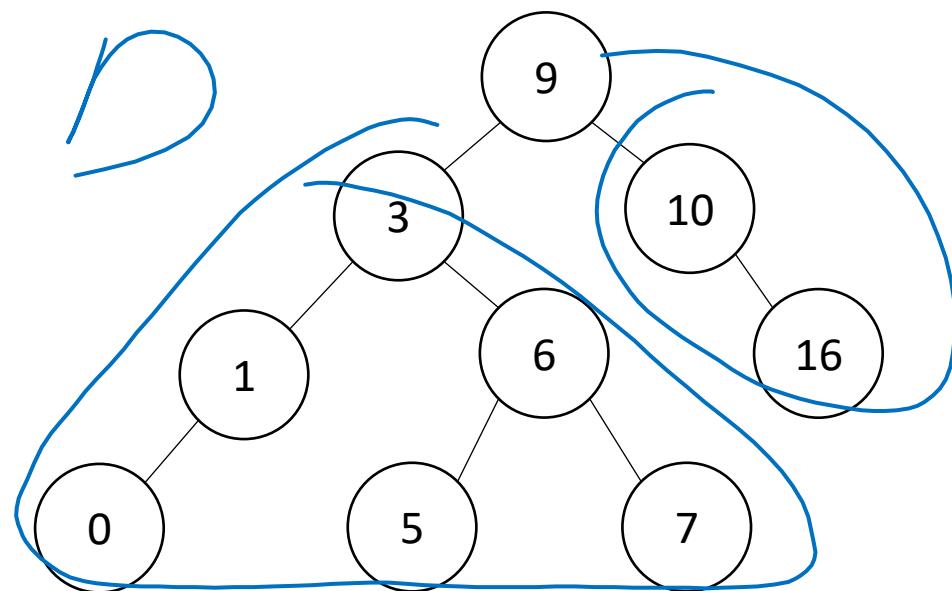
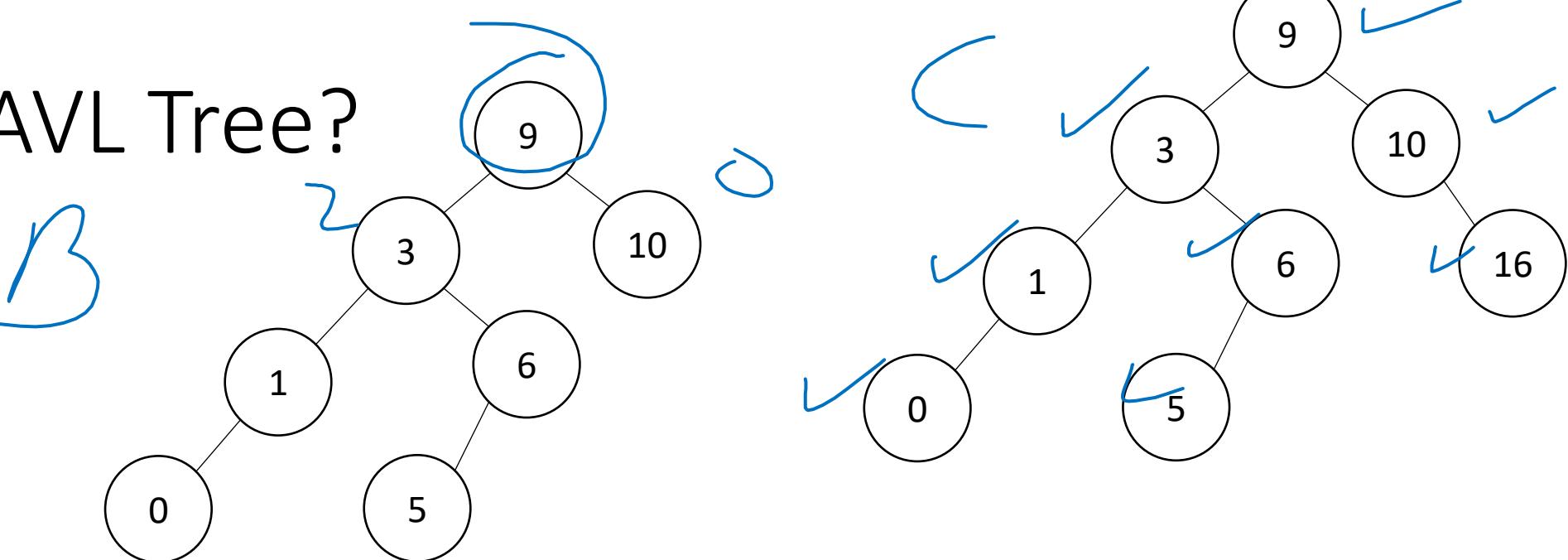
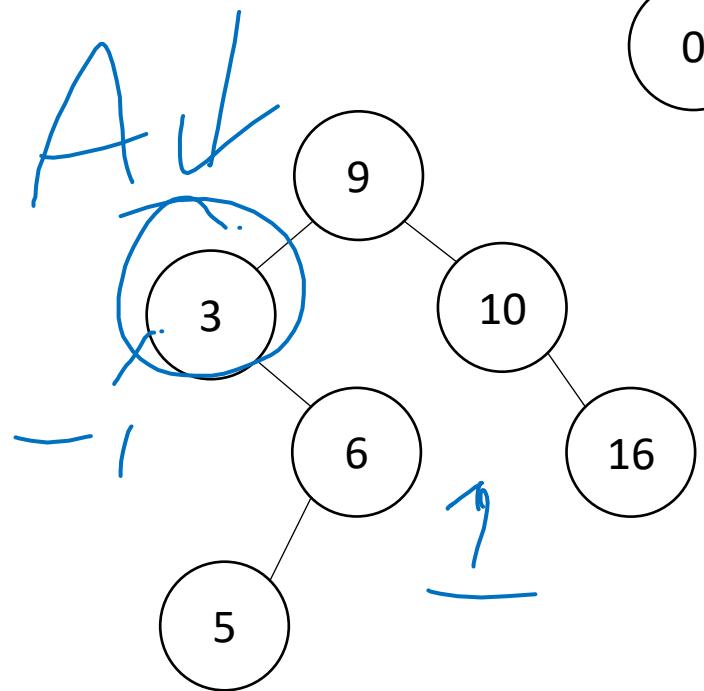
Idea 4: Both Subtrees of every Node have same height



AVL Tree

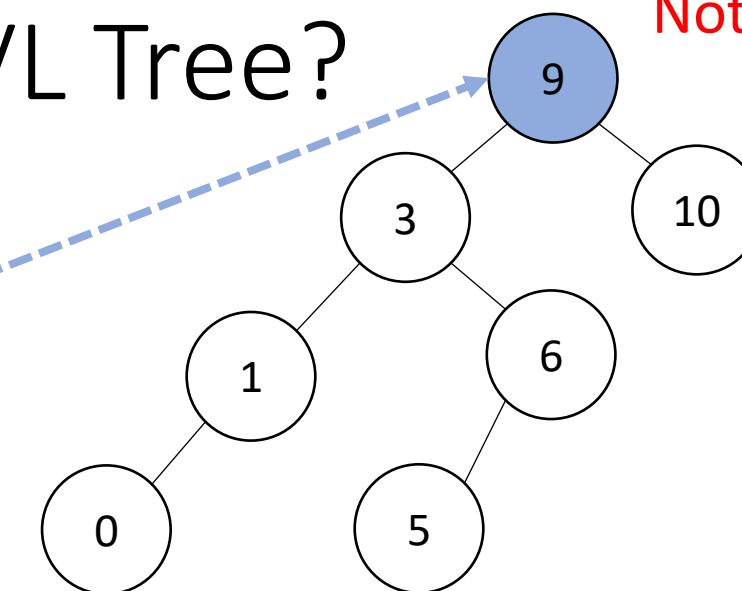
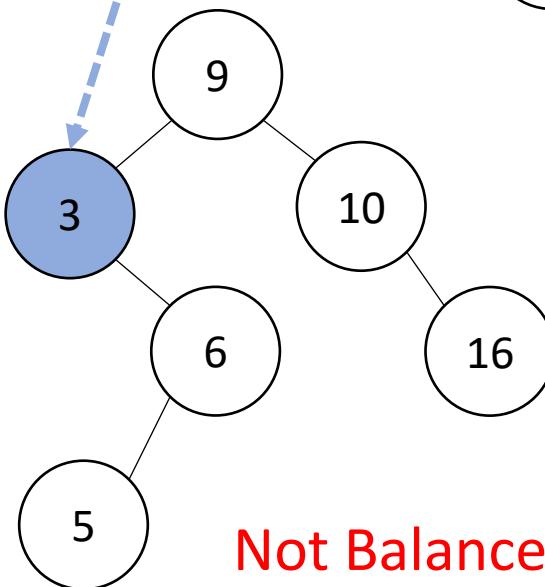
- A Binary Search tree that maintains that the left and right subtrees of every node have heights that differ by at most one.
 - height of left subtree and height of right subtree off by at most 1
 - Not too weak (ensures trees are short)
 - Not too strong (works for any number of nodes)
- Idea of AVL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

Is it an AVL Tree?

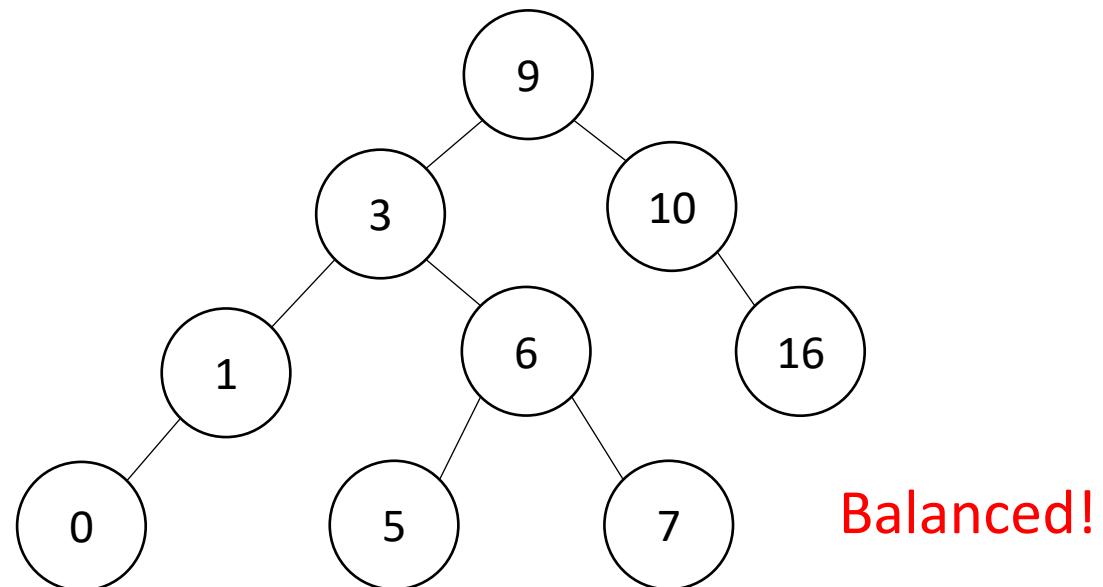
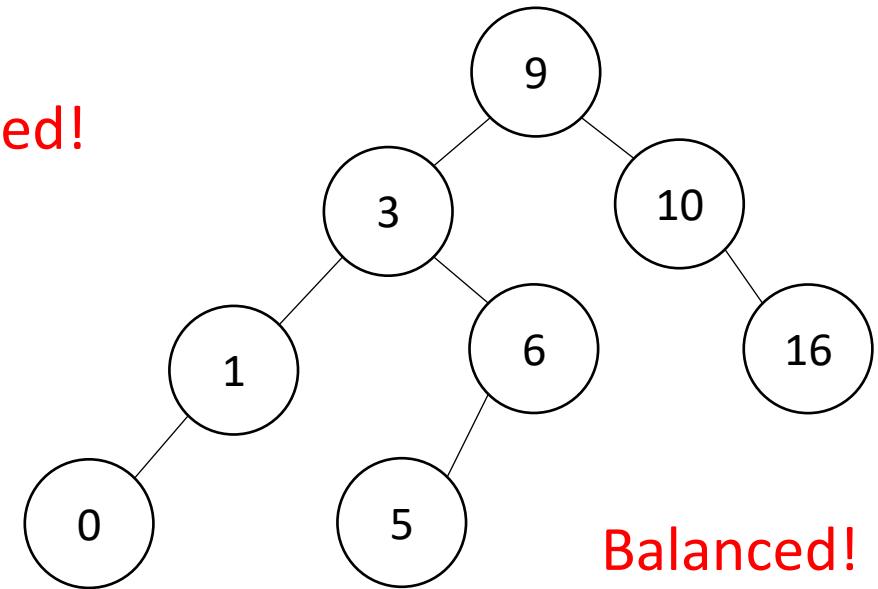


Is it an AVL Tree?

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

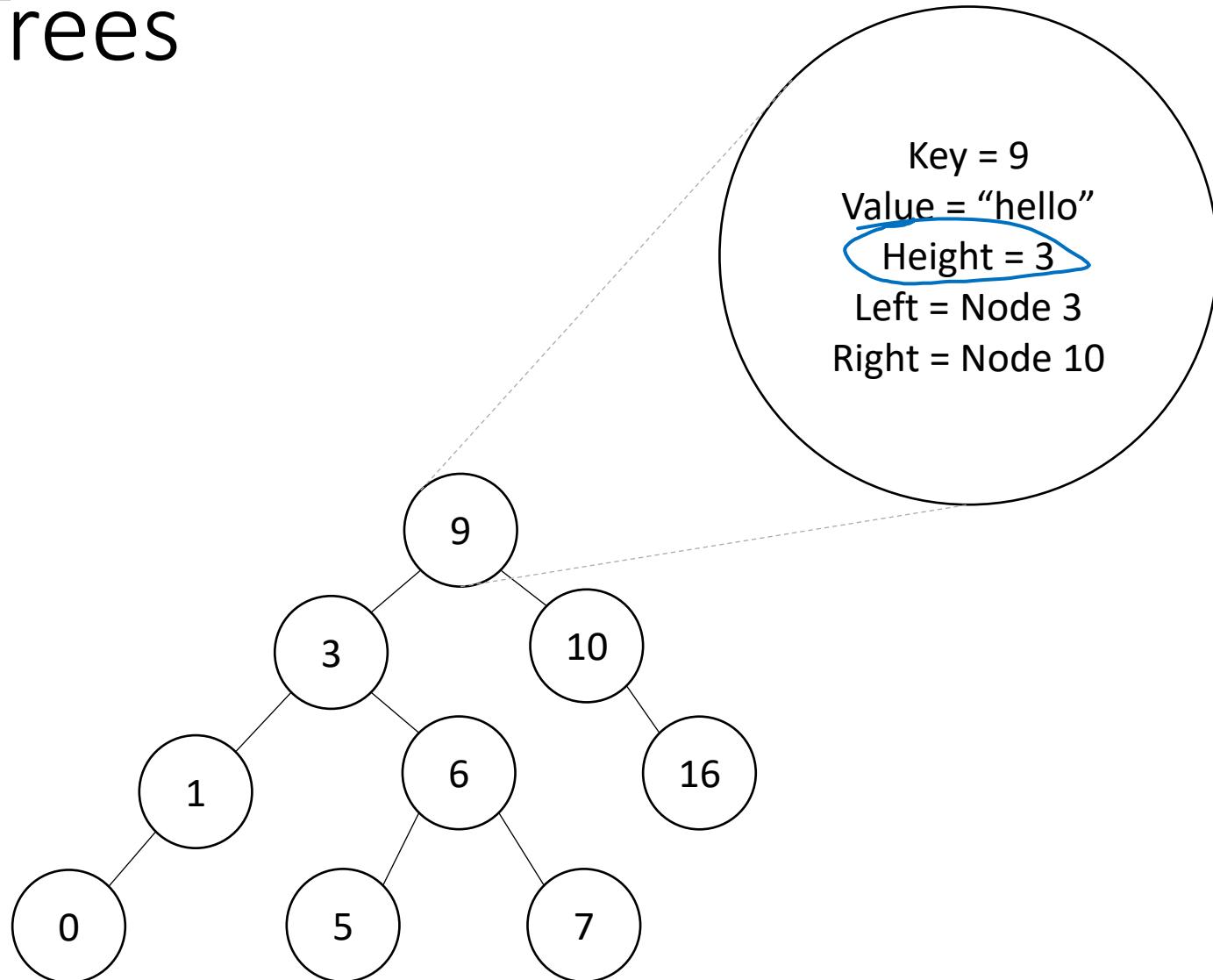


Not Balanced!



Using AVL Trees

- Each node has:
 - Key
 - Value
 - Height
 - Left child
 - Right child

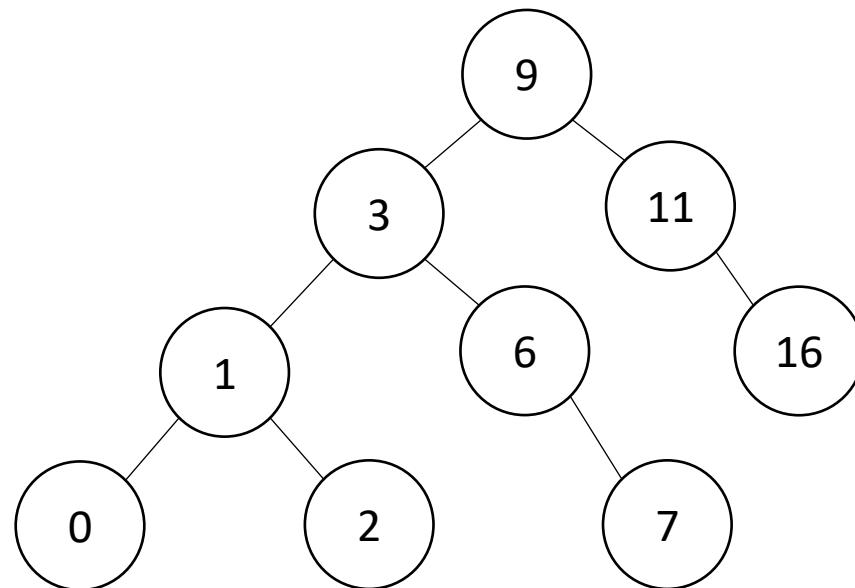


Inserting into an AVL Tree

- Starts out the same way as BST:
 - “Find” where the new node should go
 - Put it in the right 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

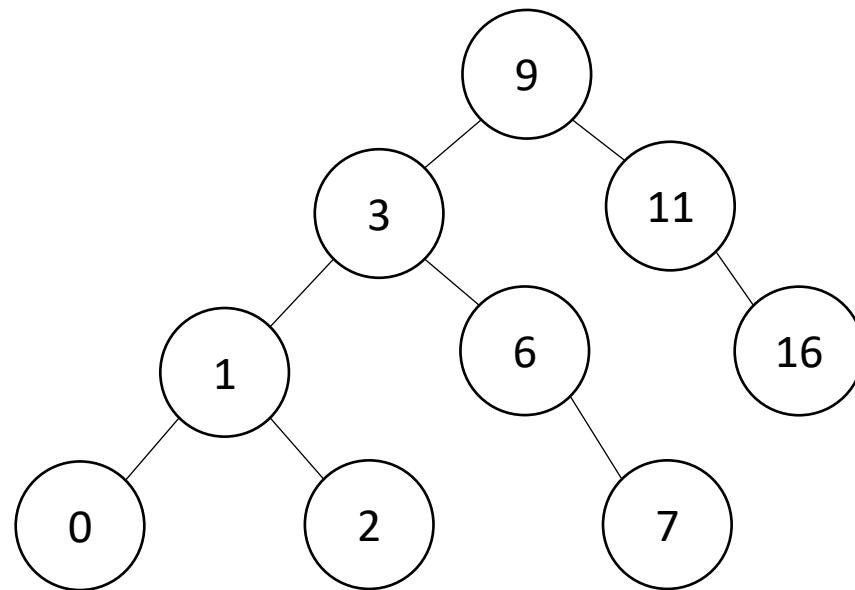
Insert Example

10



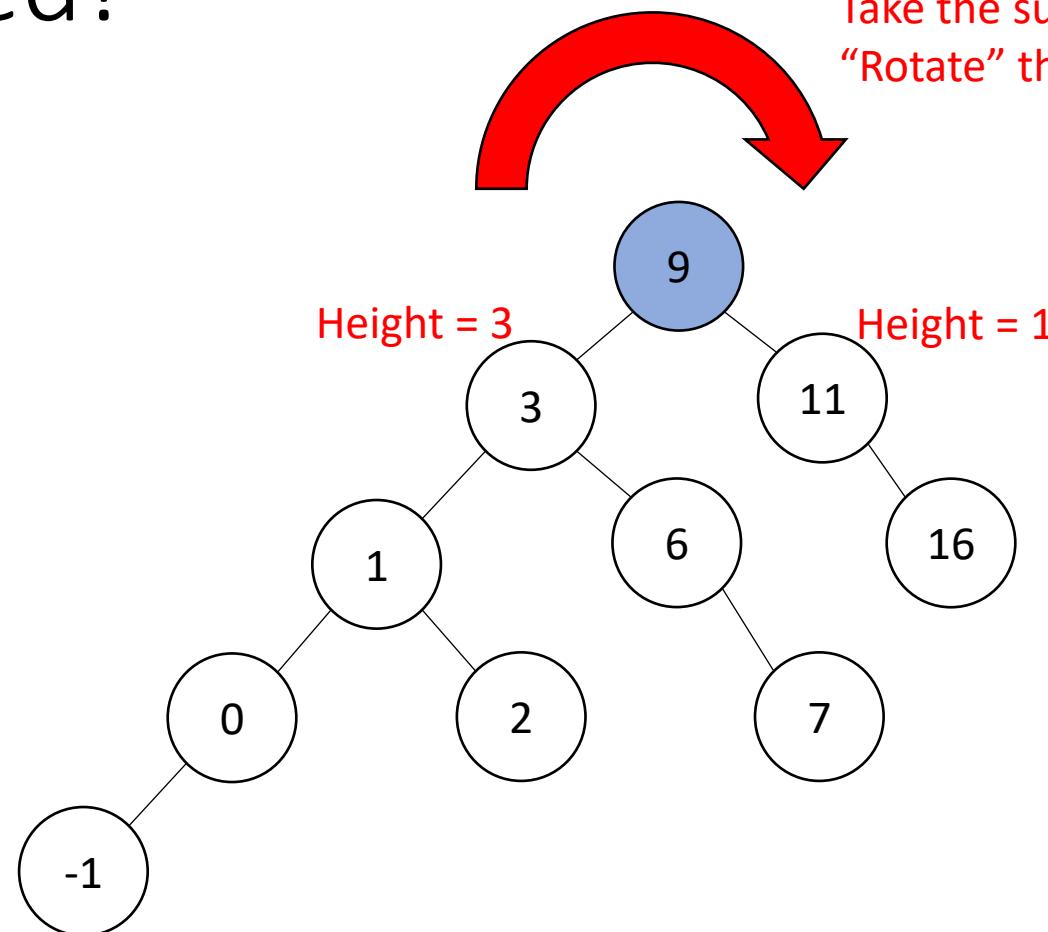
Insert Example

-1

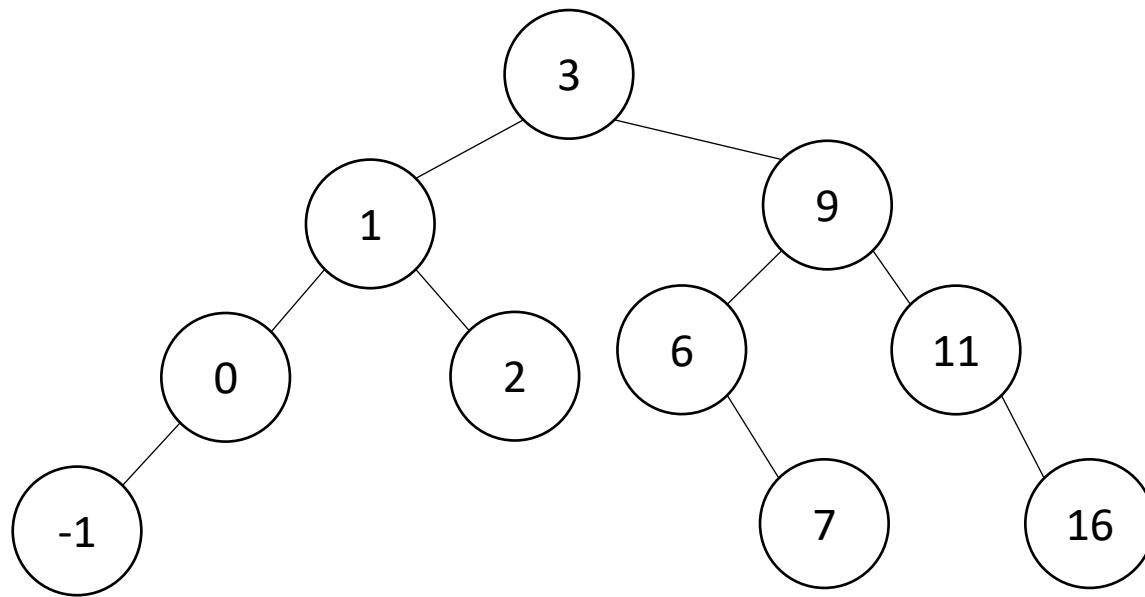


Not Balanced!

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

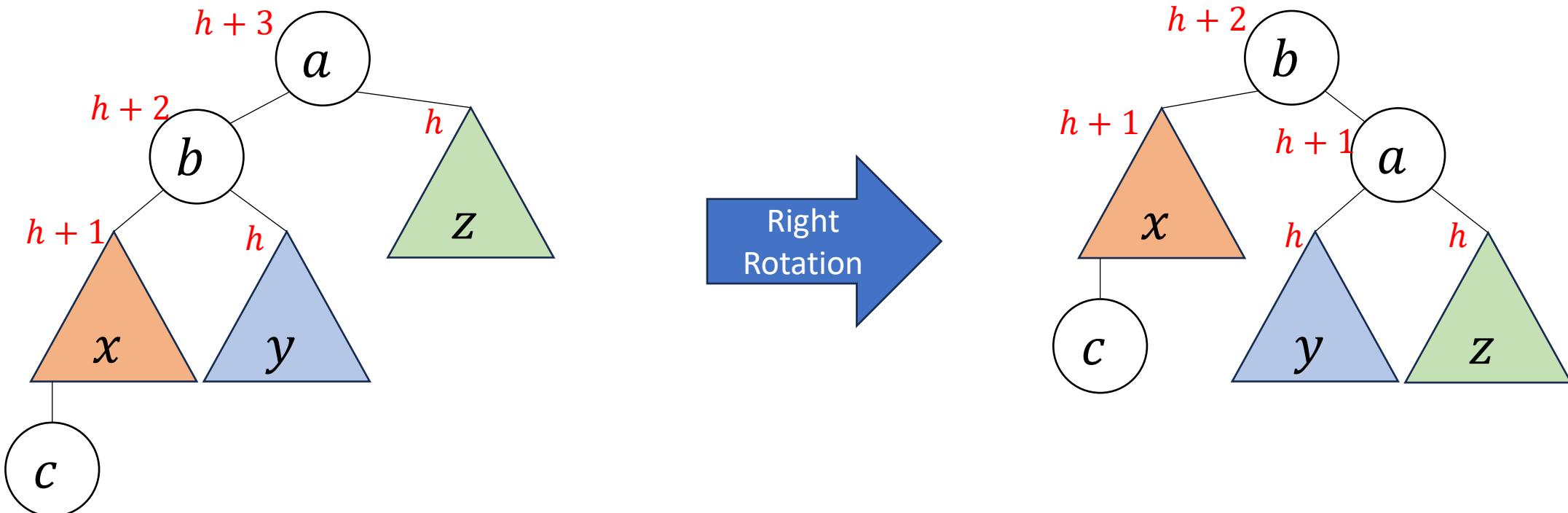


Balanced!

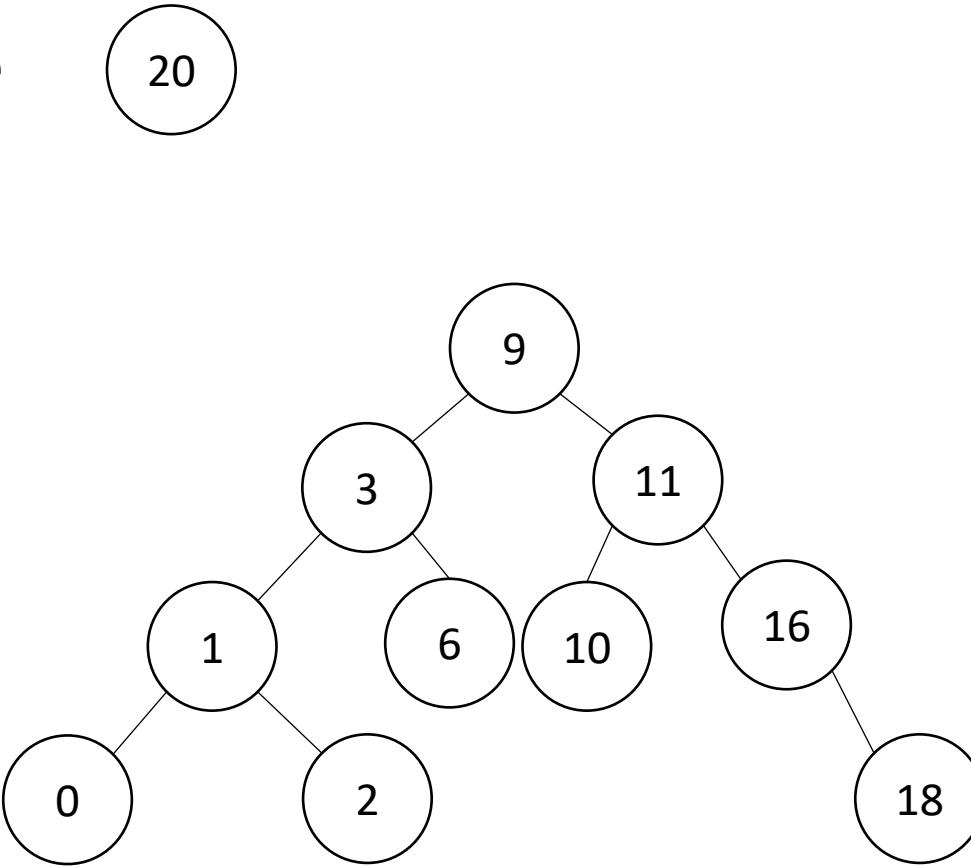


Right Rotation

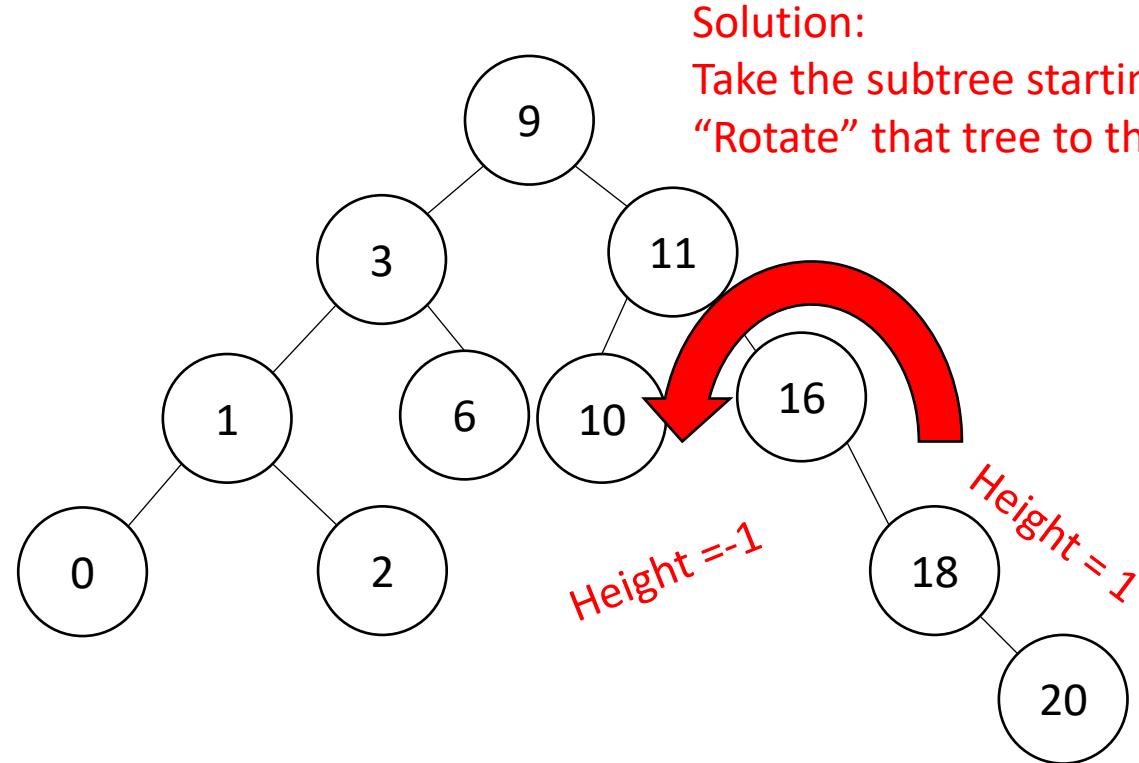
- We just inserted c , node a is the deepest “problem” node
- Make the left child the new root
- Make the old root the right child of the new
- Make the new root’s right subtree the old root’s left subtree



Insert Example



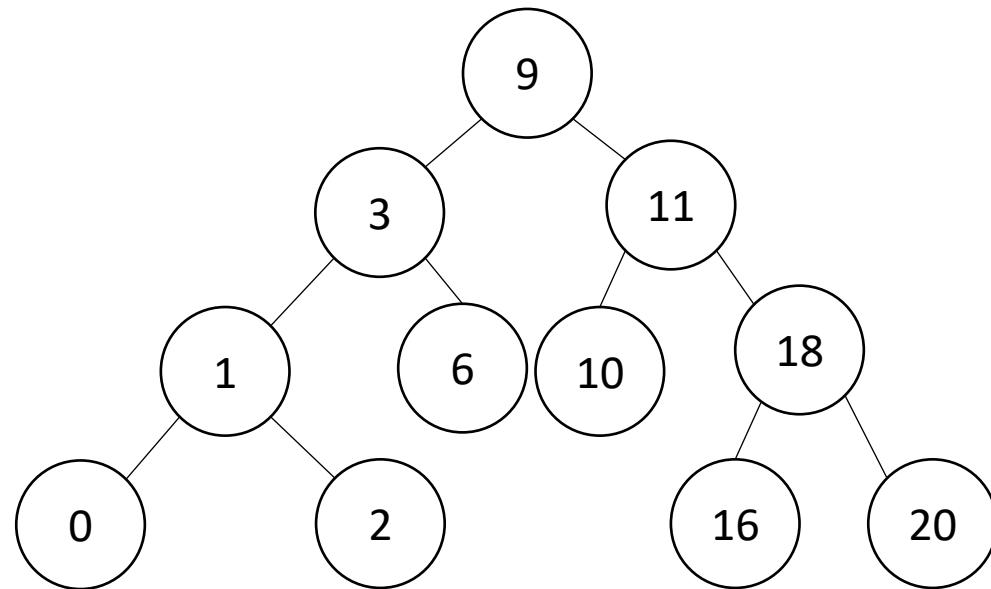
Not Balanced!



Solution:

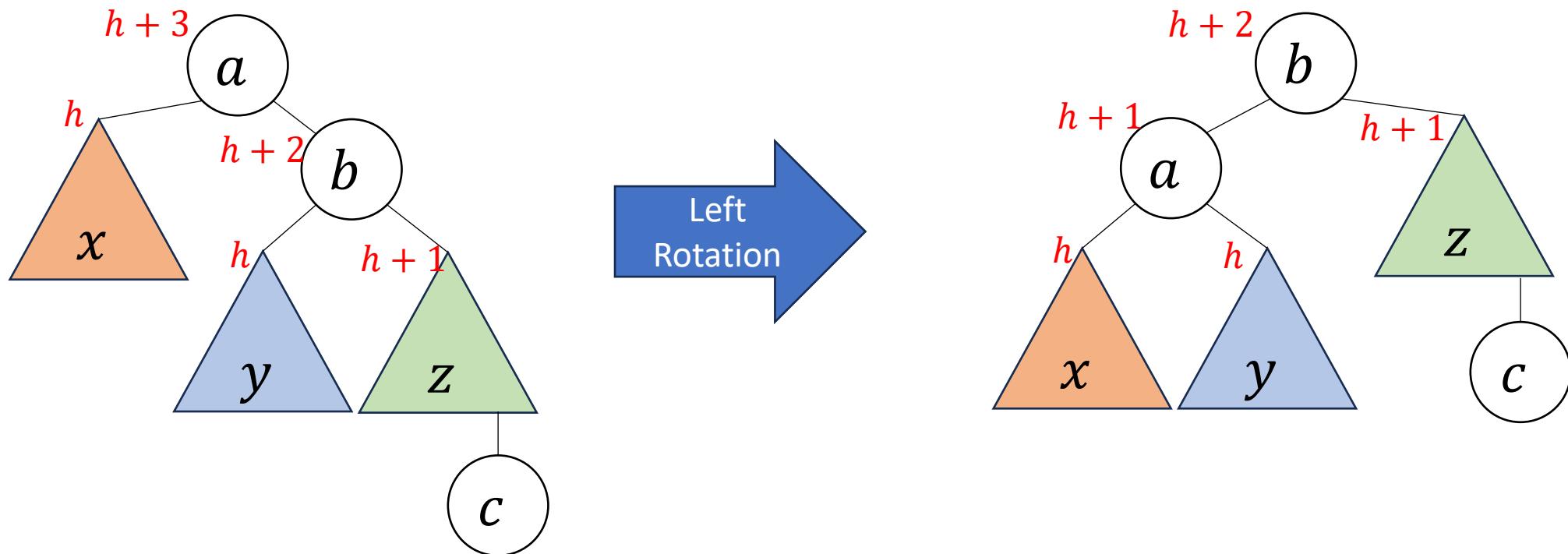
Take the subtree starting with the problem node,
"Rotate" that tree to the left

Balanced!



Left Rotation

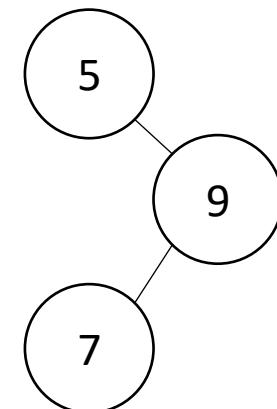
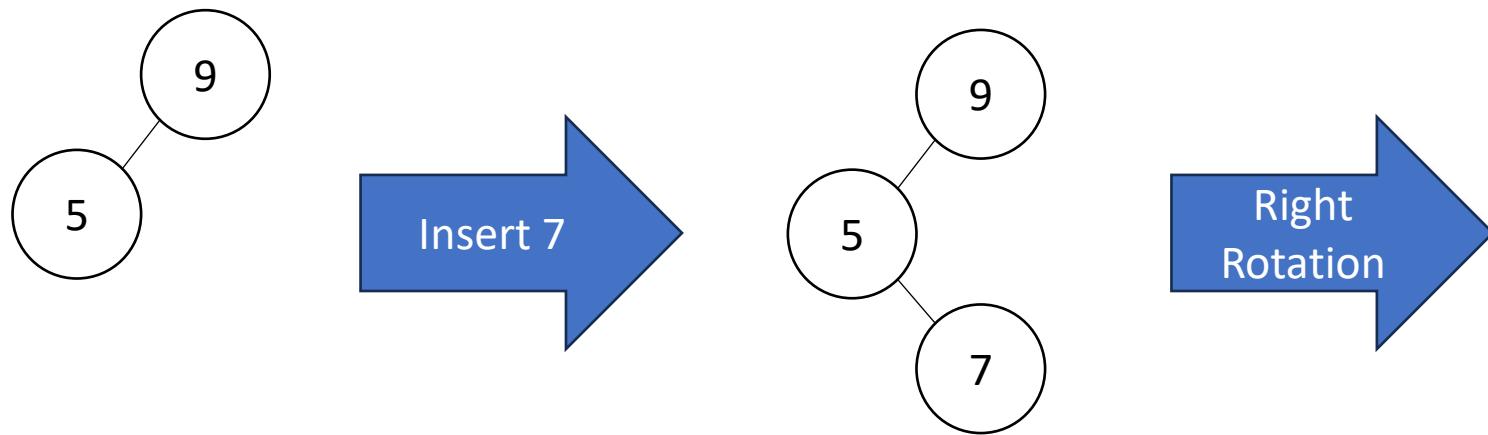
- We just inserted c , node a is the deepest “problem” node
- Make the right child the new root
- Make the old root the left child of the new
- Make the new root’s left subtree the old root’s right subtree



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!



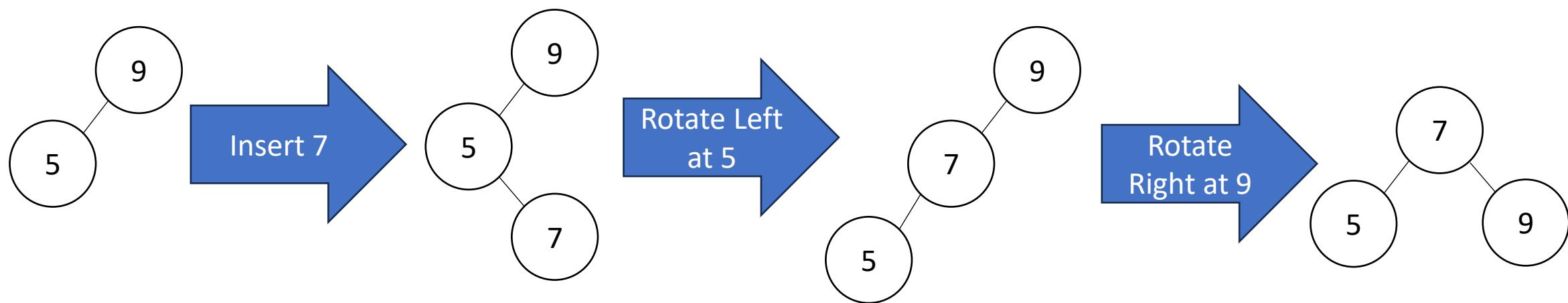
Insertion Story So Far

- 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!

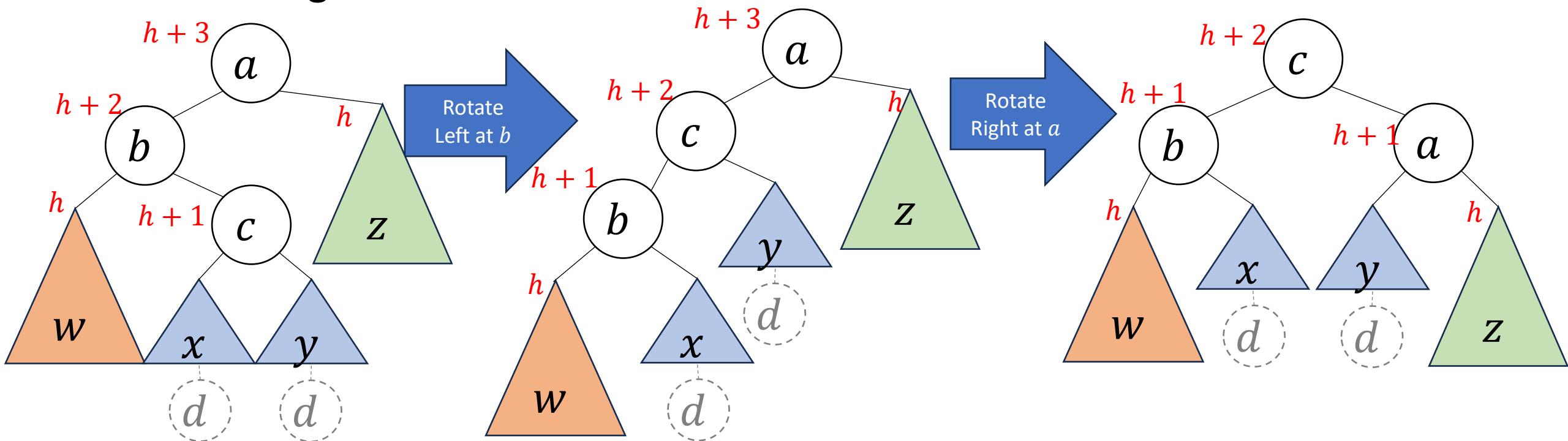
Case LR

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



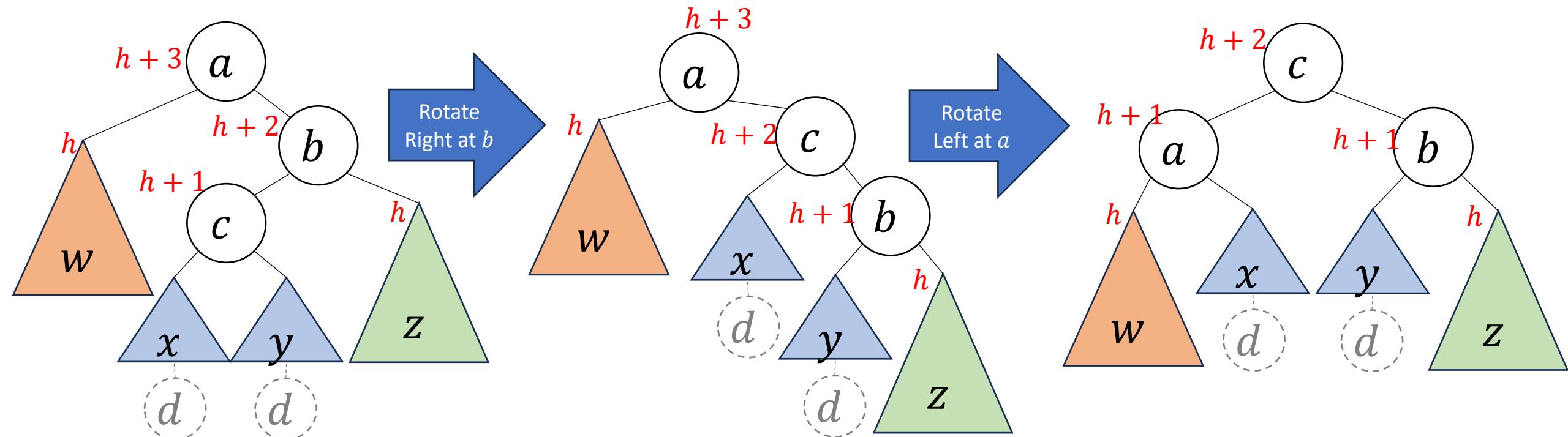
Case LR in General

- 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 RL in General

- 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



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

Delete Summary

- Tldr: same cases, reverse direction of rotation, may need to repeat with ancestors
- After a BST deletion, update the heights of the node's ancestors
- From leaf to root, check if each node is unbalanced
- If a node is unbalanced then at the deepest unbalanced node:
 - Case LL: If we deleted in the **left** subtree of the **left** child then: **rotate left**
 - Case RR: If we deleted in the **right** subtree of the **right** child then: **rotate right**
 - Case LR: If we deleted into the **right** subtree of the **left** child then: **rotate right** at the left child and then **rotate left** at the root
 - Case RL: If we deleted into the **left** subtree of the **right** child then: **rotate left** at the right child and then **rotate right** at the root
- Continue checking until reach the root