# CSE 332 Autumn 2023
# Lecture 9: 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)

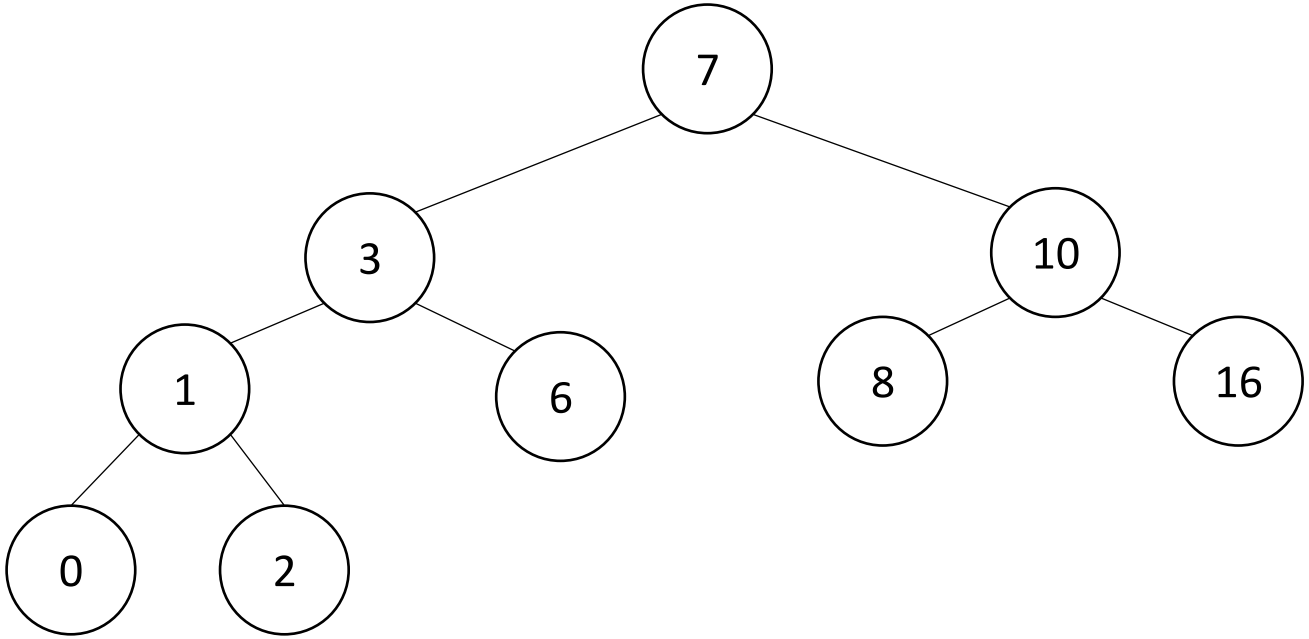
# Less Naïve attempts

- Binary Search Trees (Friday)
- Tries (Project)
- AVL Trees (Today)
- B-Trees (this week)
- HashTables (next week)
- Red-Black Trees (not included in this course)
- Splay Trees (not included in this course)

# Dictionary Data Structures

| 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 Search Tree | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| AVL Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ |

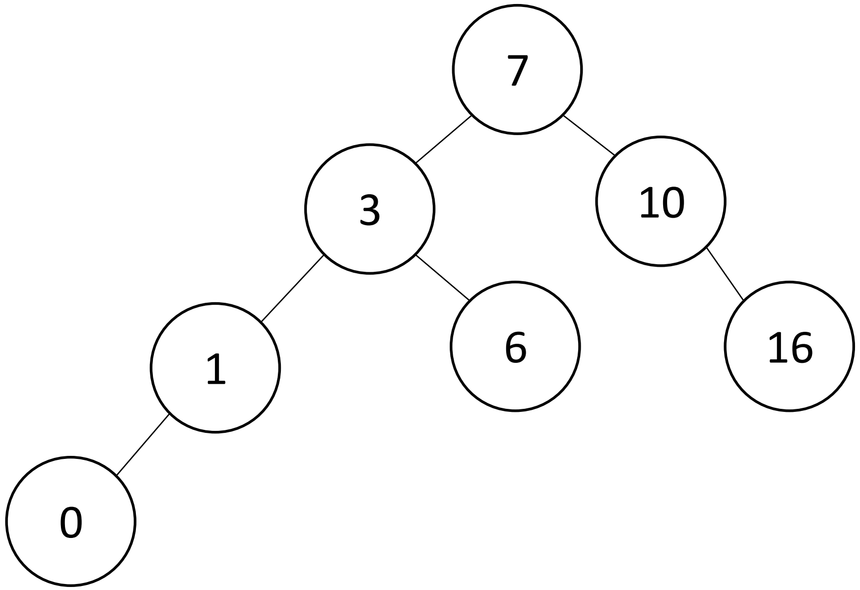
# Binary Search Tree




- Binary Tree
  - Definition:
  - Every 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
  - Apply recursively
- Why?
  - Makes searching quicker
    - Worst case: tree’s height

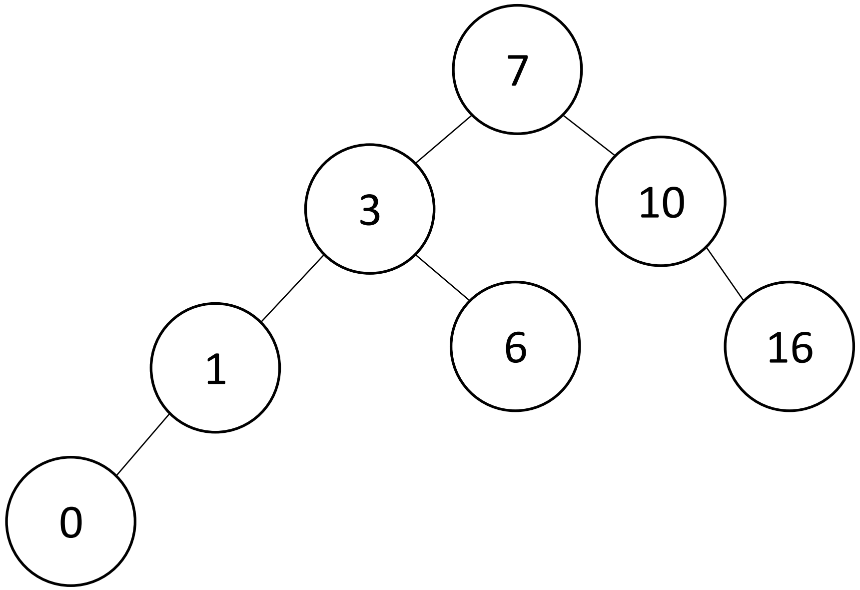
# Find Operation (recursive)

```
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;
}
```

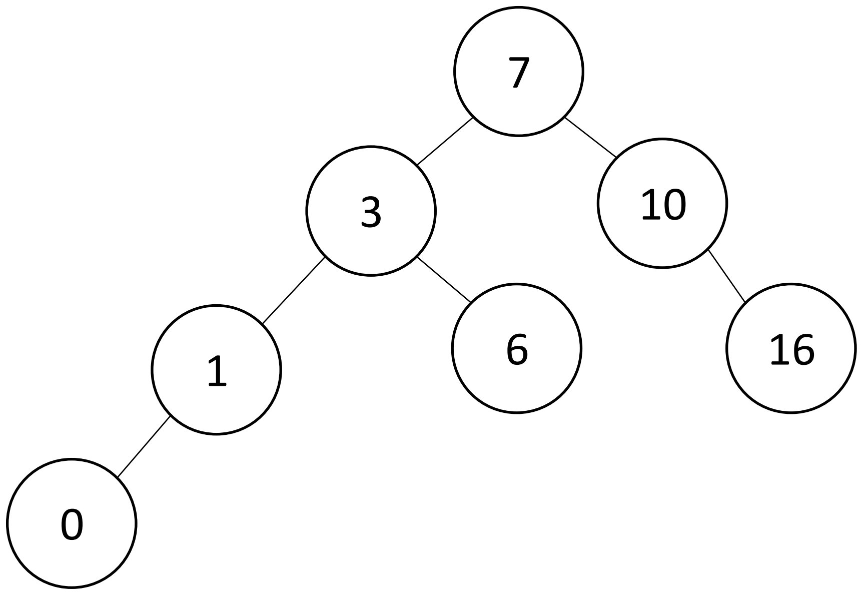


# Find Operation (iterative)

```
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 (iterative)

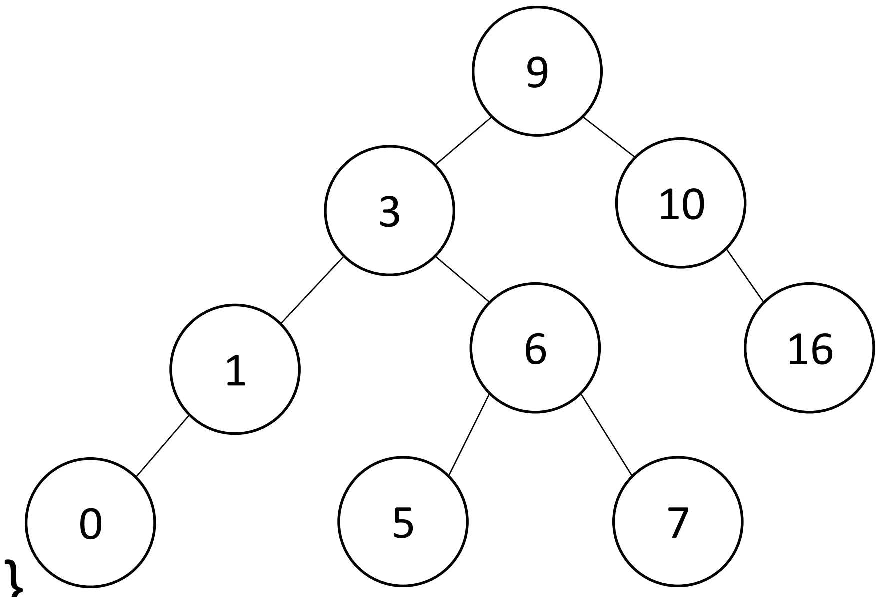



```
insert(key, value, root){
    if (root == Null){ this.root = new Node(key, value); }
    parent = Null;
    while (root != Null && key != root.key){
        parent = root;
        if (key < root.key){ root = root.left; }
        else if (key > root.key){ root = root.right; }
    }
    if (root !=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

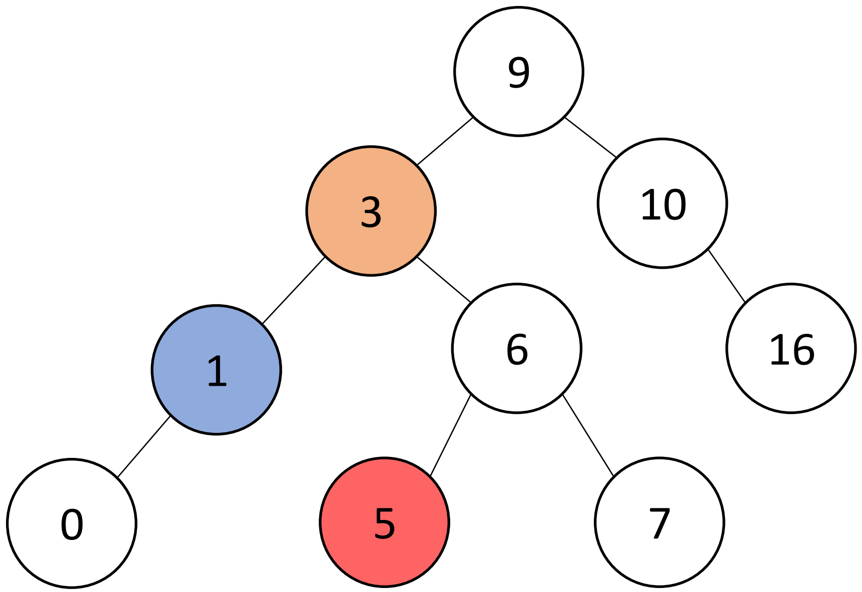
# Delete Operation (iterative)

```
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
- 2 Children



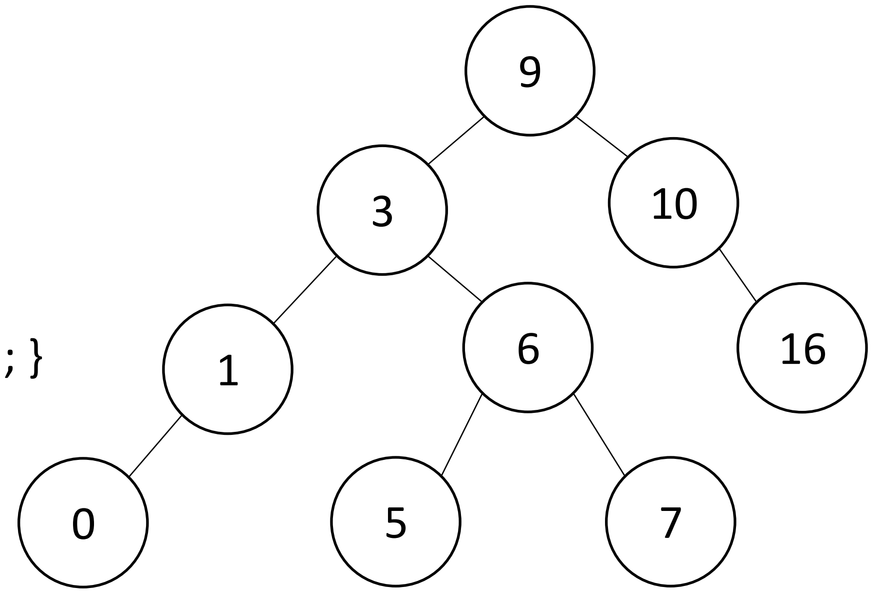
# Finding the Max and Min

- Max of a BST:
  - Right-most Thing

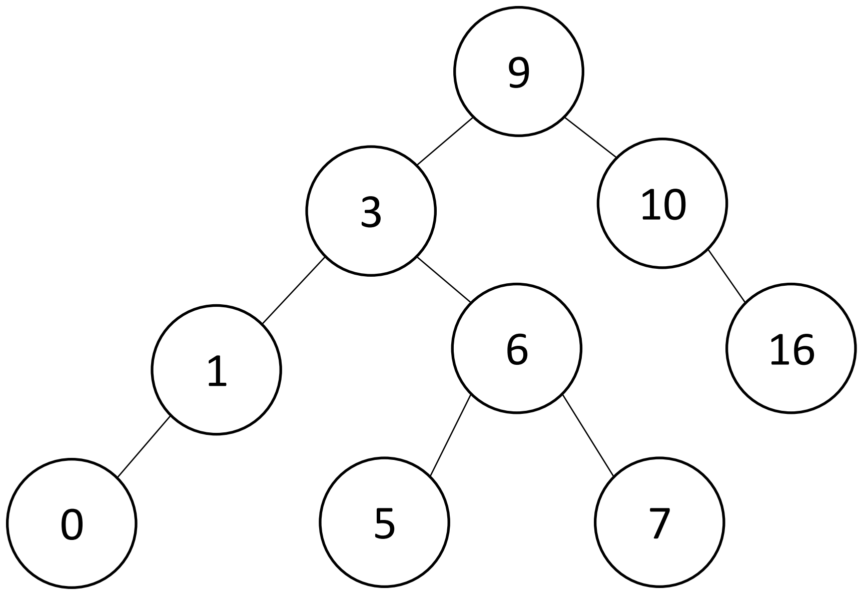
```
maxNode(root){
        if (root == Null){ return Null; }
        while (root.right != Null){
                root = root.right;
        }
        return root;
}
```

- Min of a BST:
  - Left-most Thing

```
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
        }
}
```

# Improving the worst case

- How can we get a better worst case running time?

# “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”

# 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

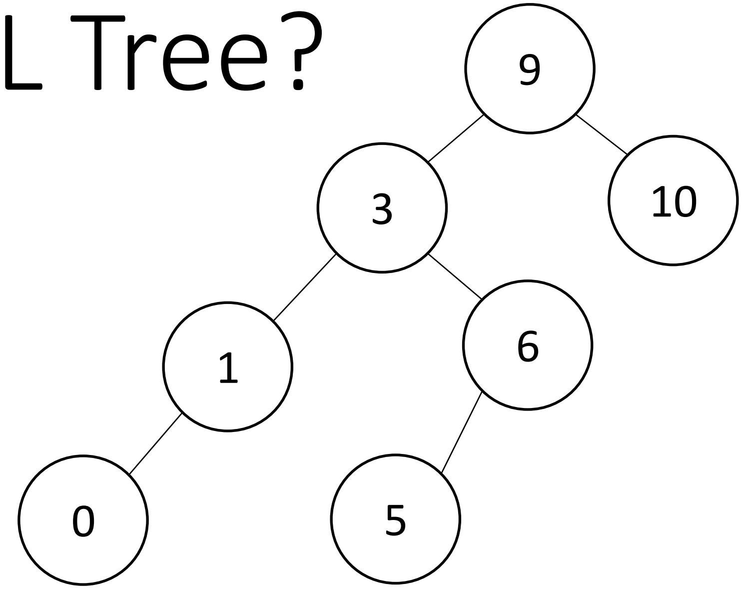
# Idea 4: Both Subtrees of every Node have same height

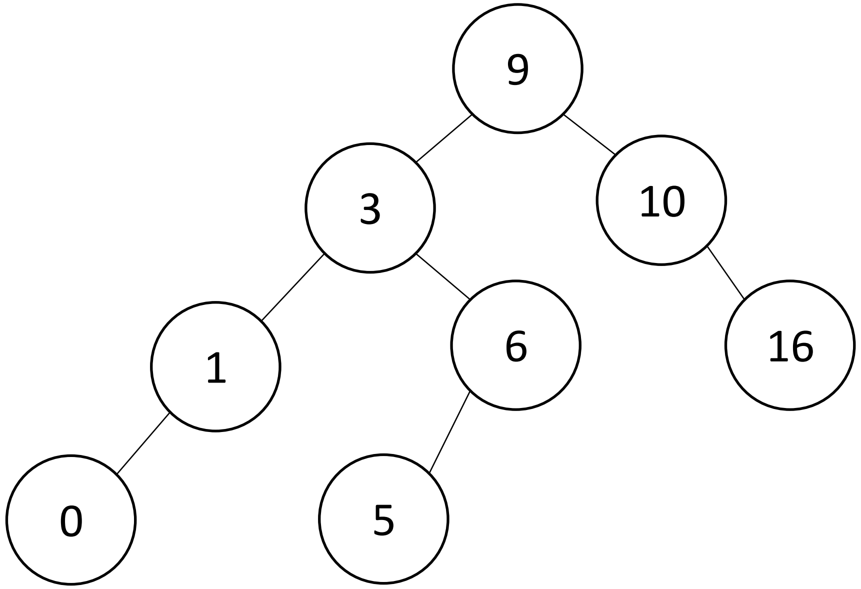
# AVL Tree

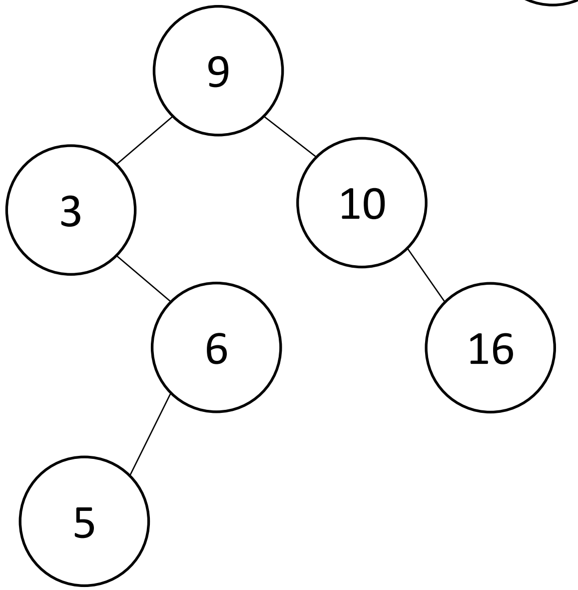
- A Binary Search tree that maintains that the left and right subtrees of every node have heights that differ by at most one.
  - height of left subtree and height of right subtree off by at most 1
  - Not too weak (ensures trees are short)
  - Not too strong (works for any number of nodes)

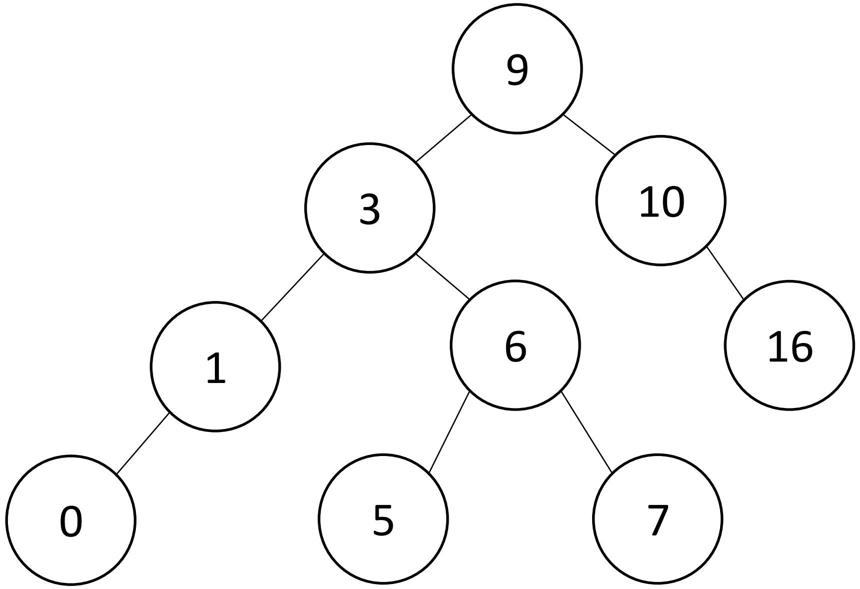
- Idea of AVL Tree:
  - When you insert/delete nodes, if tree is “out of balance” then modify the tree
  - Modification = “rotation”

# Is it an AVL Tree?



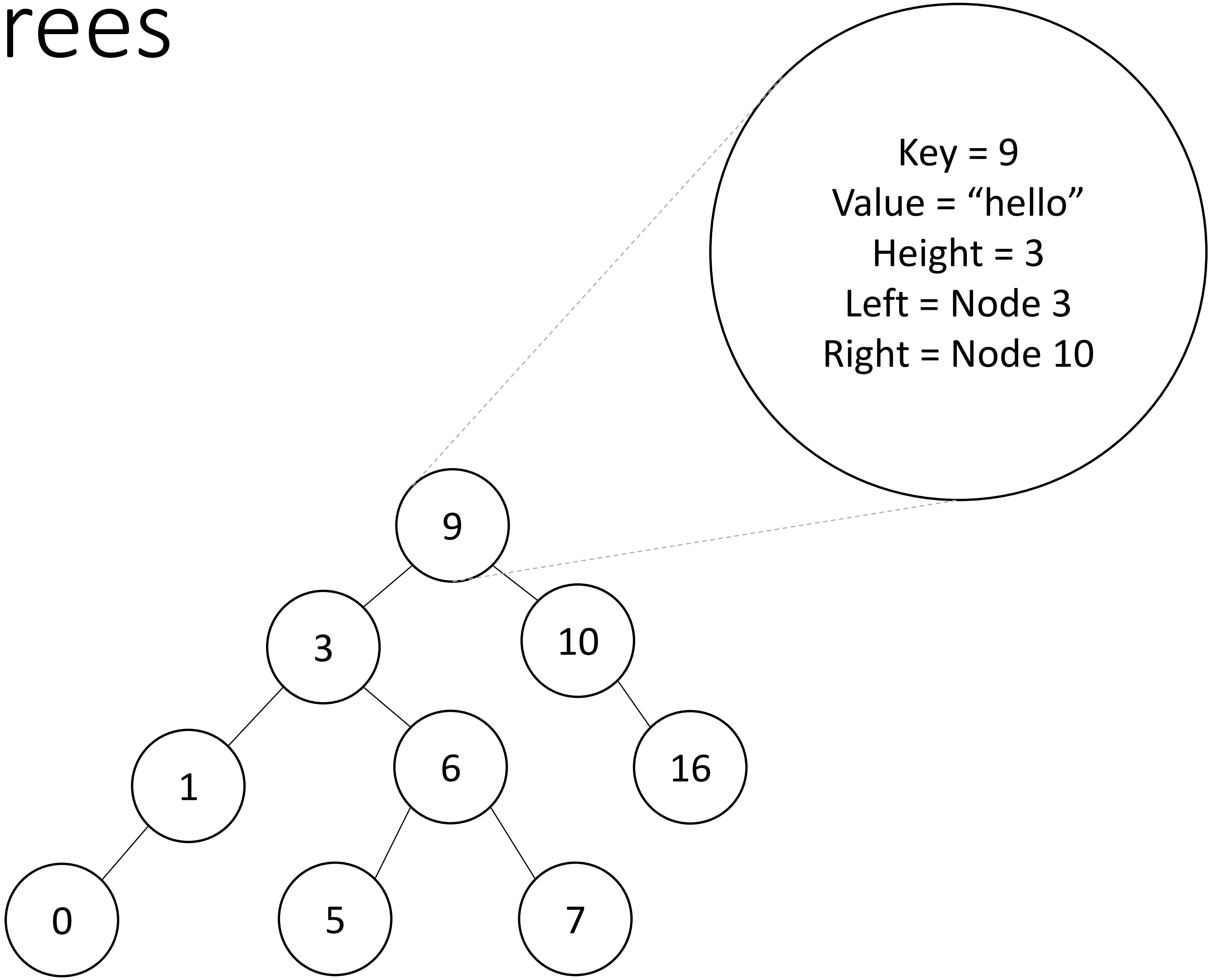



# Using AVL Trees

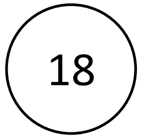
- Each node has:
  - Key
  - Value
  - Height
  - Left child
  - Right child

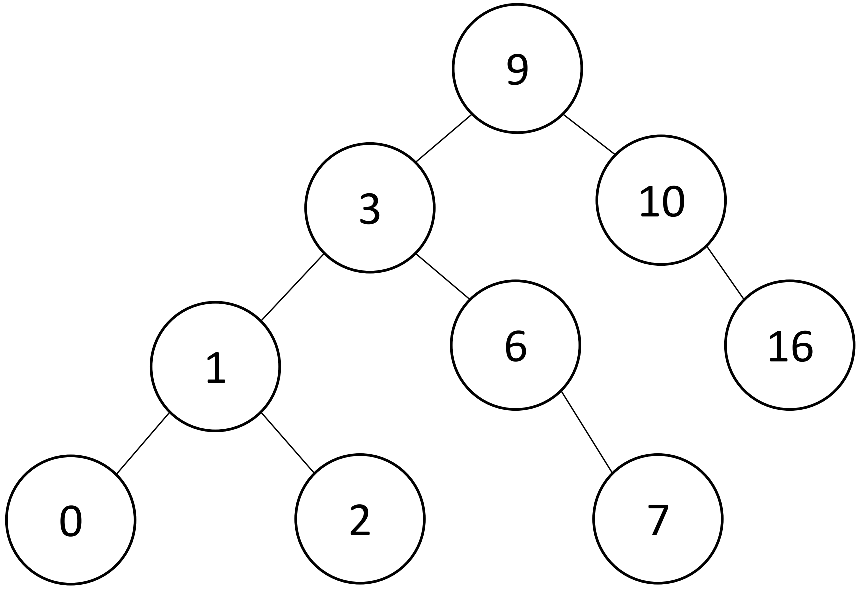


# Inserting into an AVL Tree

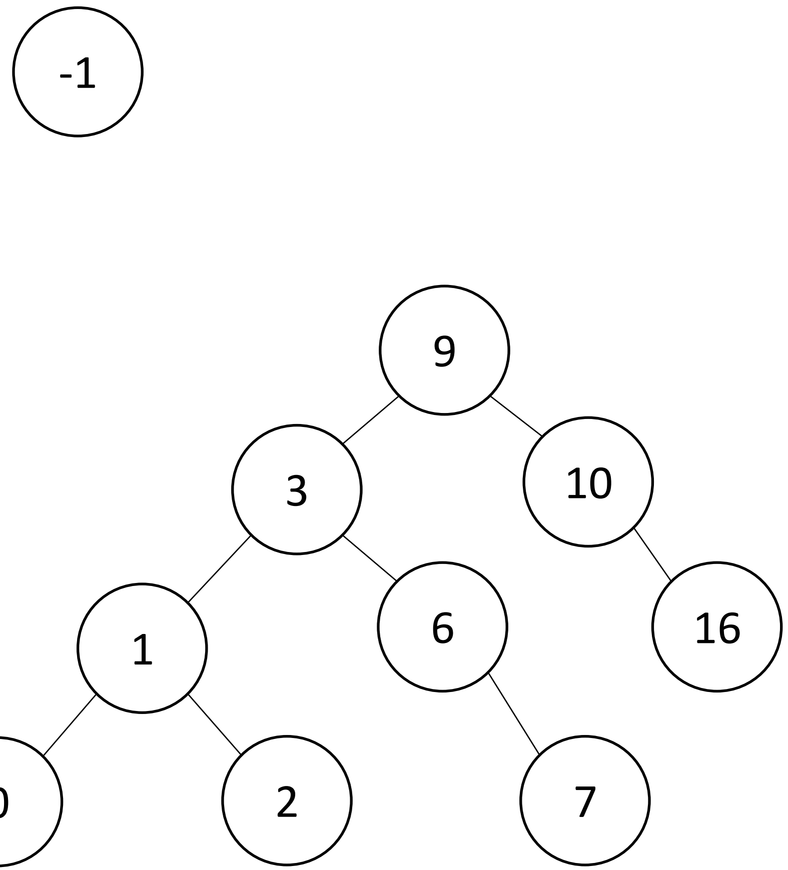
- Starts out the same way as BST:
  - “Find” where the new node should go
  - Put it in the right place (it will be a leaf)
- Next check the balance
  - If the tree is still balanced, you’re done!
  - Otherwise we need to do rotations

# Insert Example

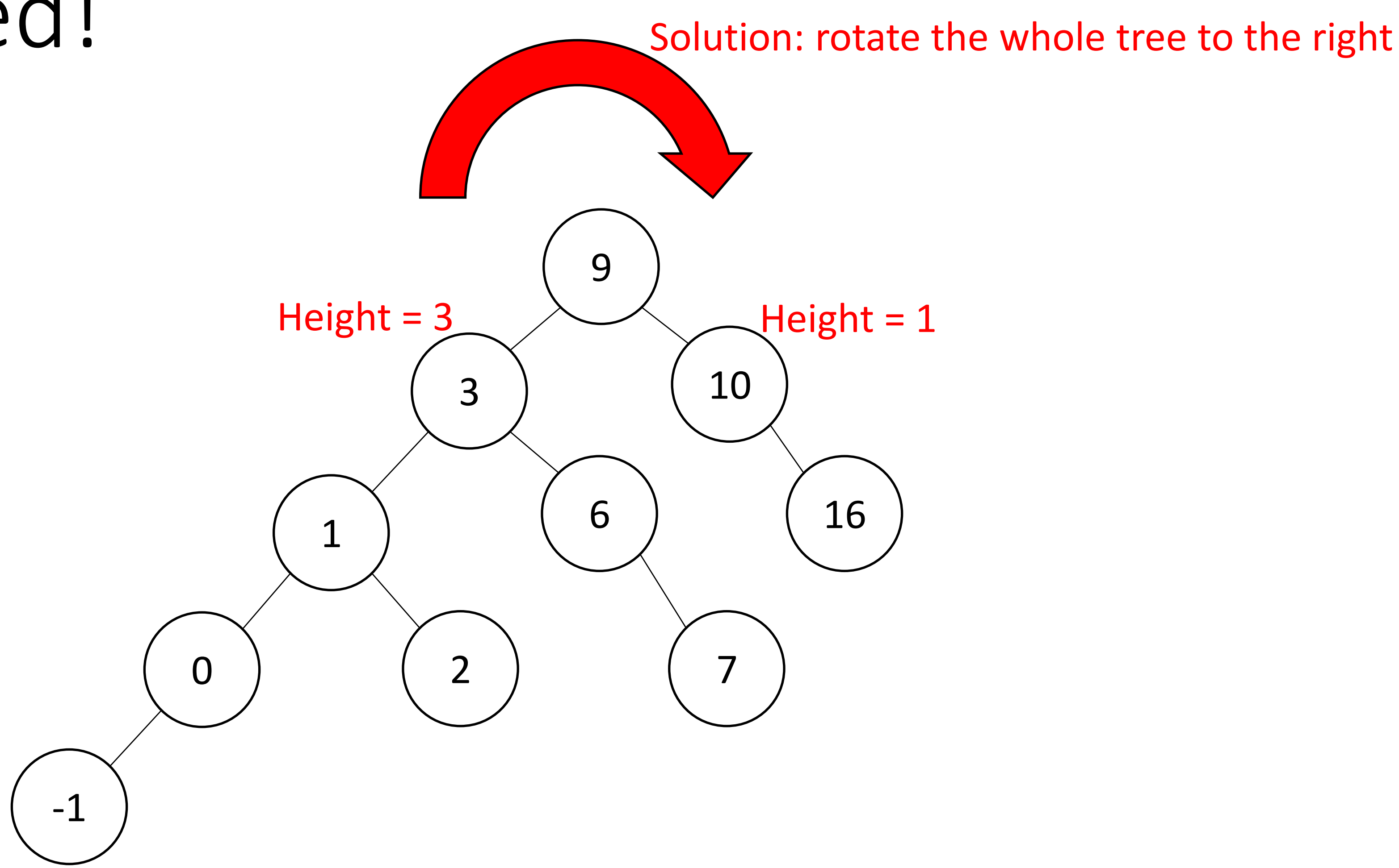


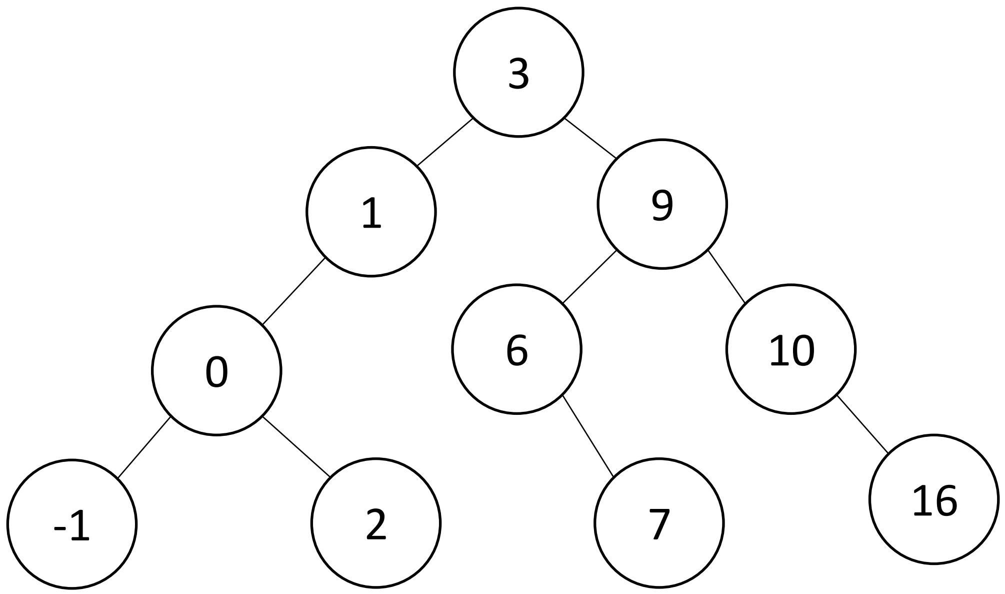
# Insert Example



# Not Balanced!

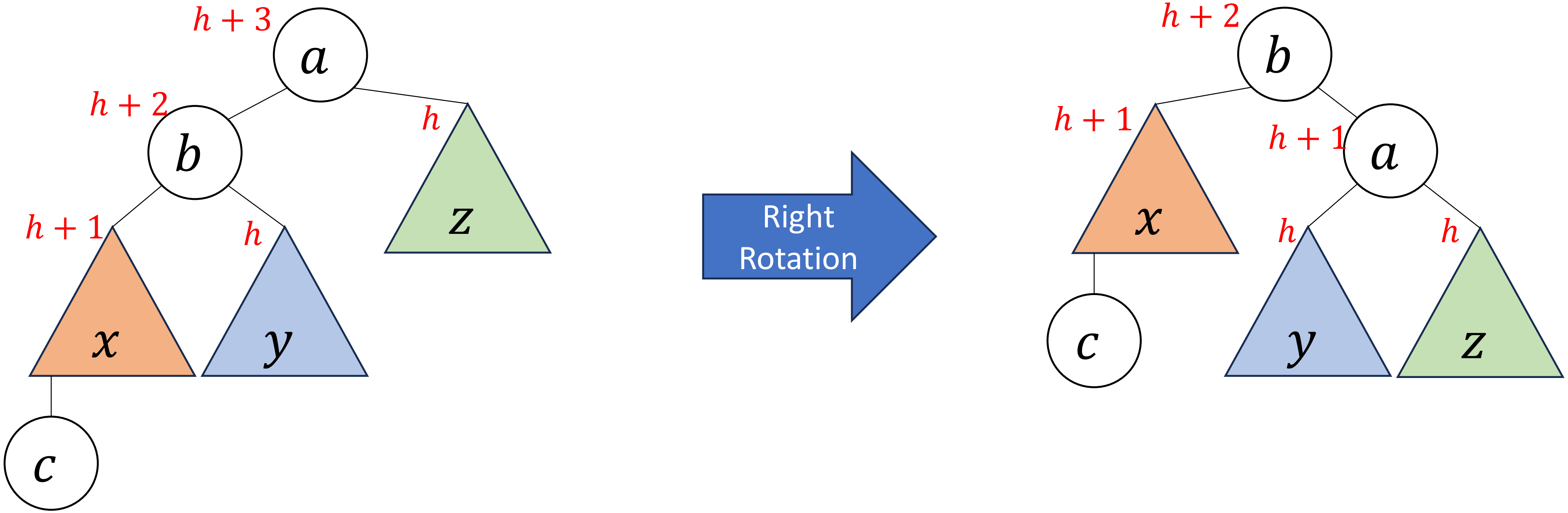


# Balanced!

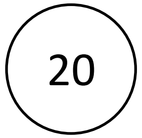


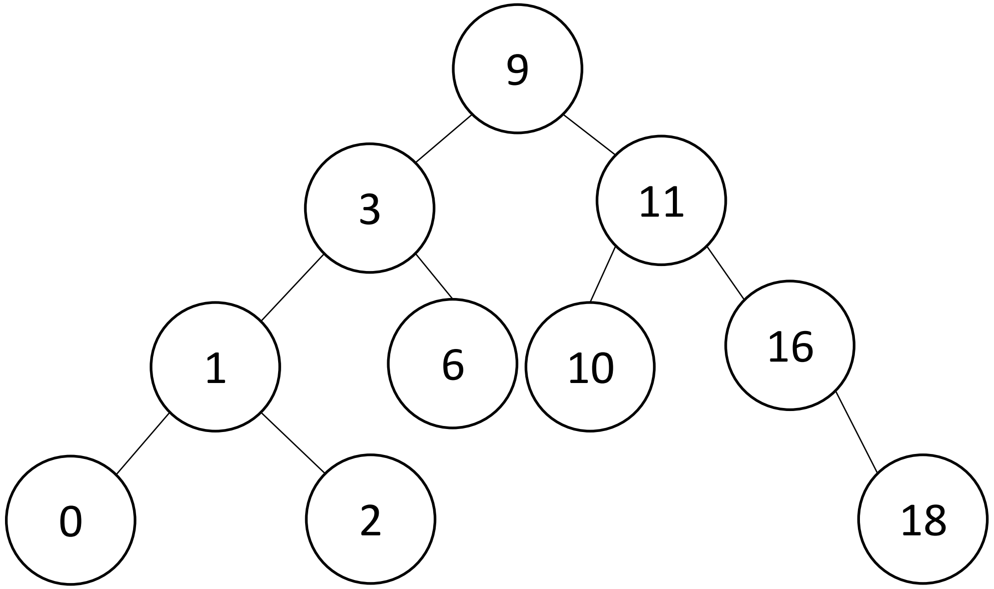
# Right Rotation

- 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

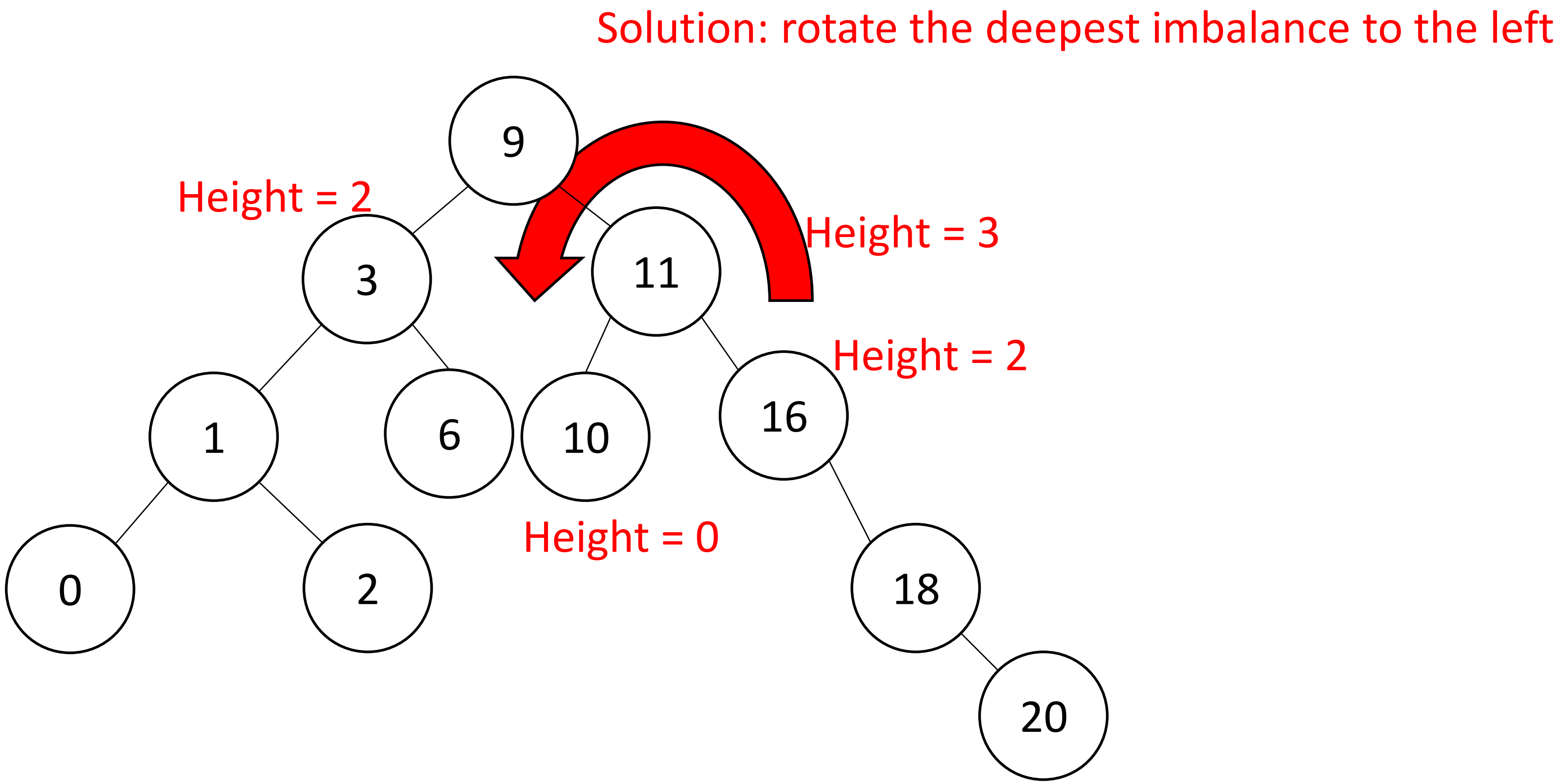


# Insert Example

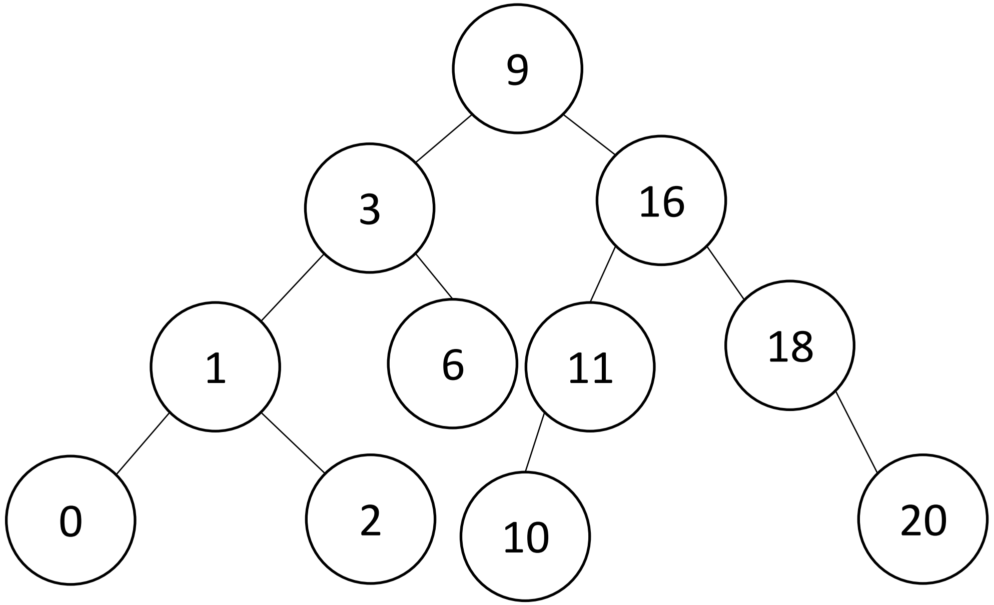


# Not Balanced!

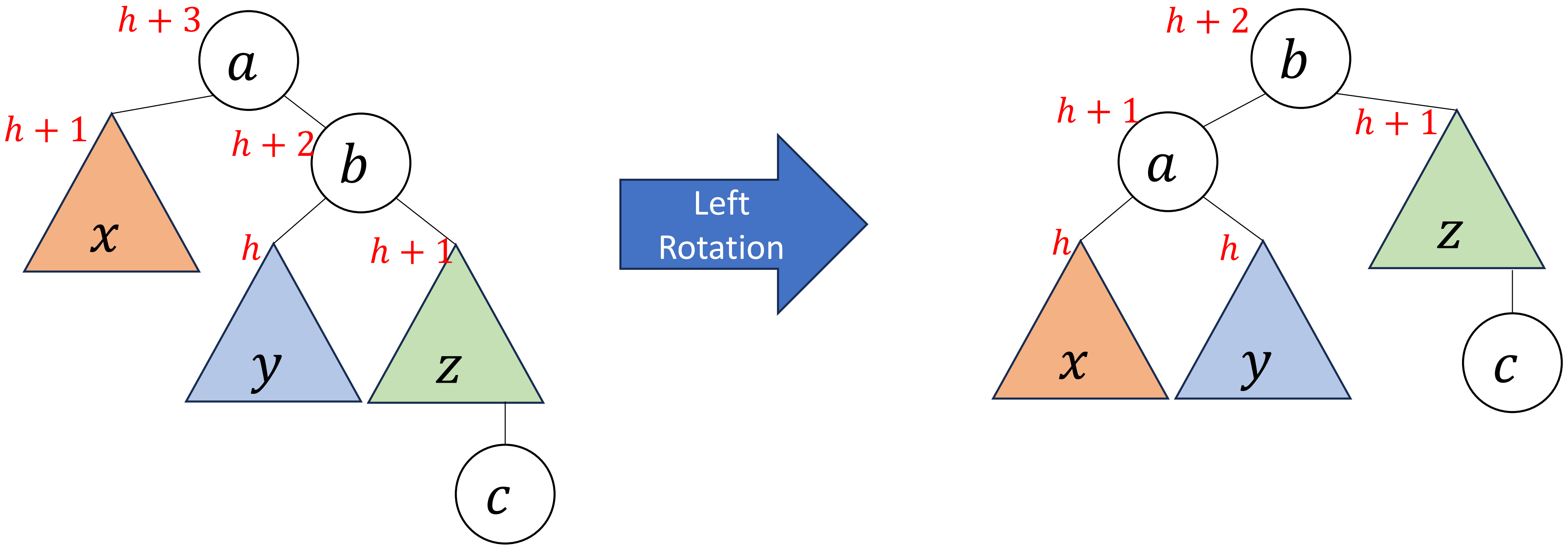


# Balanced!



# Left Rotation

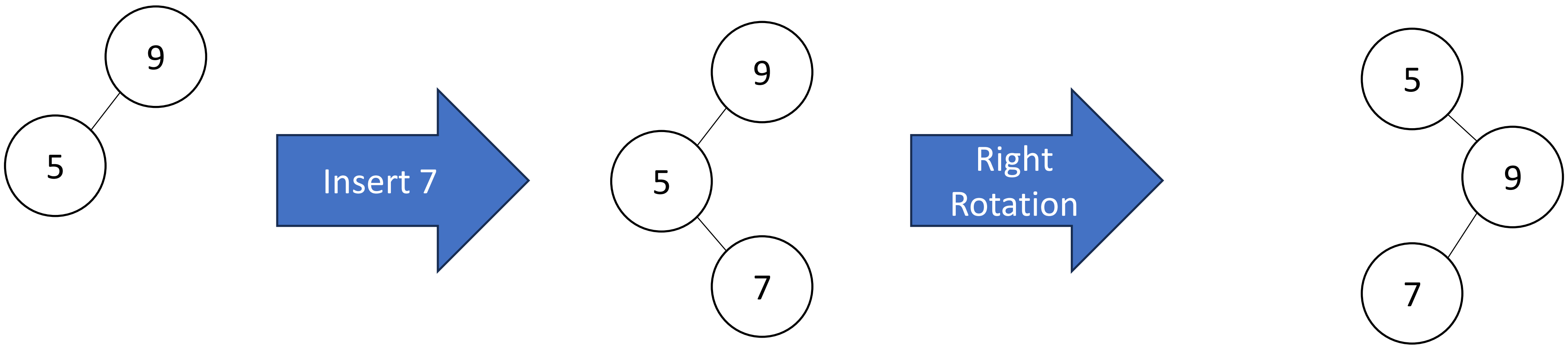
- Make the right child the new root
- Make the old root the left child of the new
- Make the new root's left subtree the old root's right subtree



# Insertion Story So Far

- After insertion, update the heights of the node's ancestors
- Check for imbalance
- If there's imbalance then at the deepest root of imbalance:
  - 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!



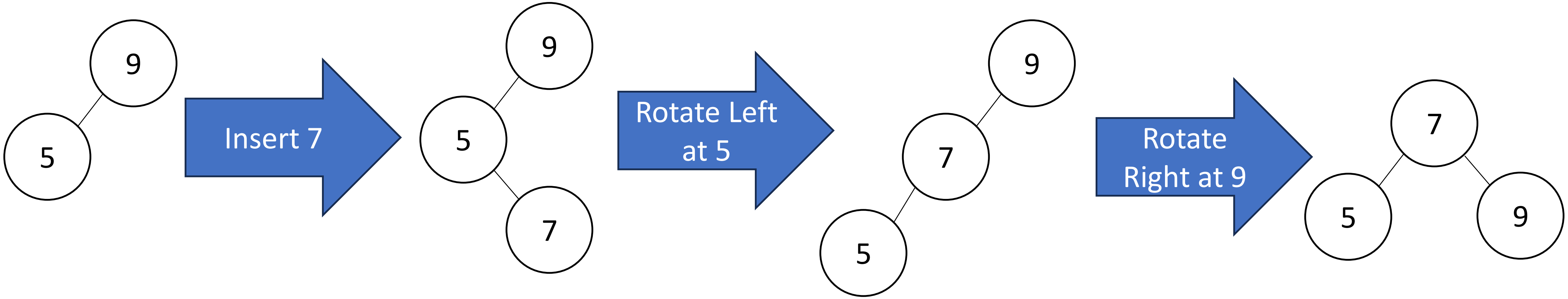
# Insertion Story So Far

- After insertion, update the heights of the node's ancestors
- Check for imbalance
- If there's imbalance then at the deepest root of imbalance:
  - 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!

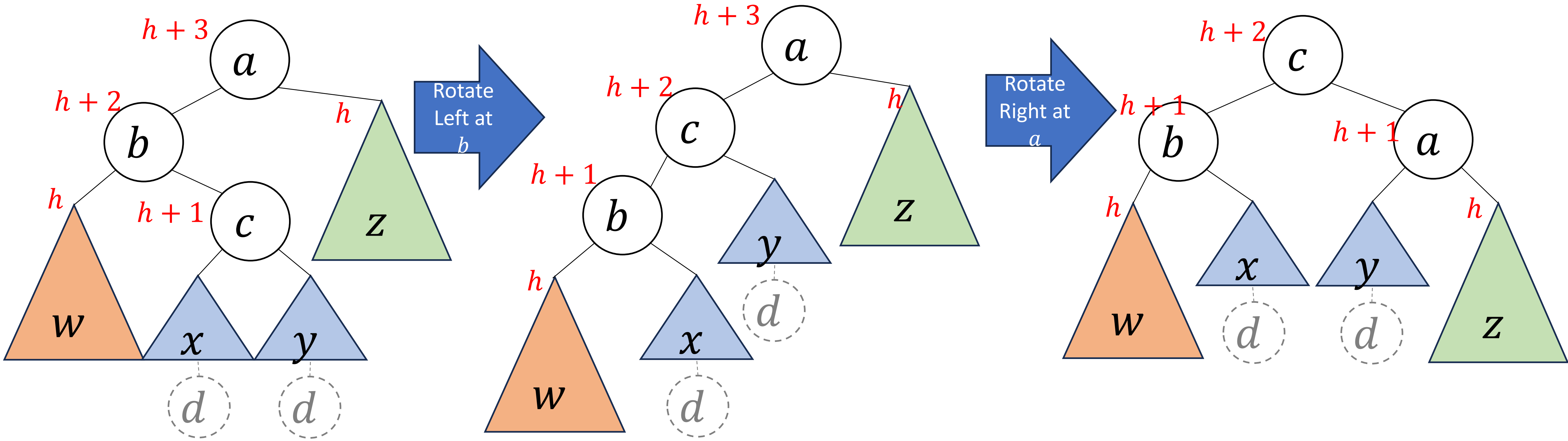
# Case LR

- From “root” of the deepest imbalance:
  - Rotate left at the left child
  - Rotate right at the root



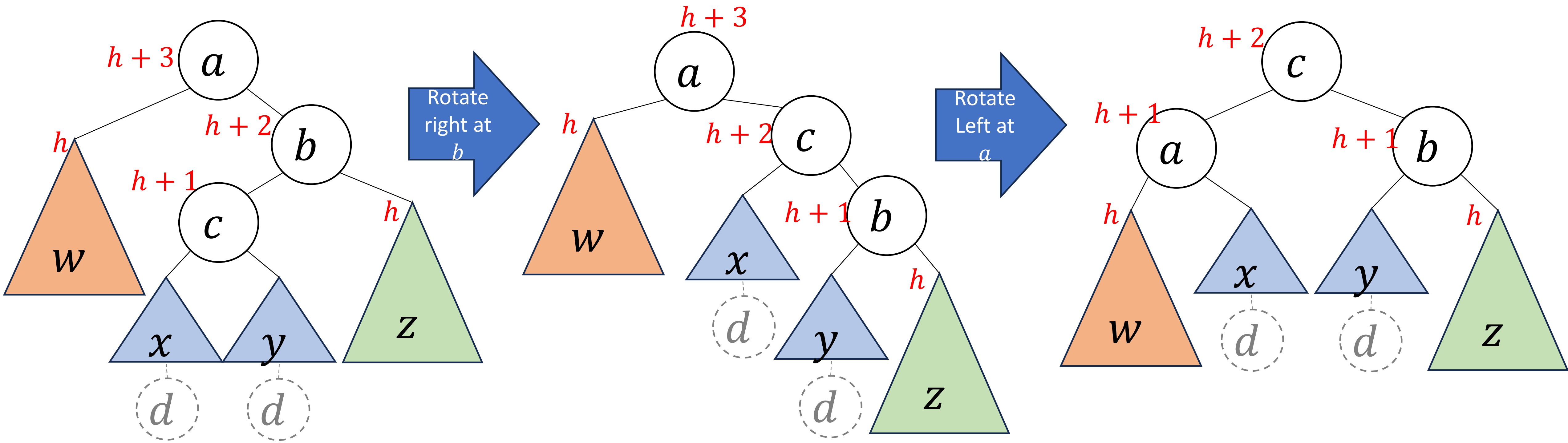
# Case LR in General

- Imbalance caused by inserting in the left child's right subtree
- Rotate left at the left child
- Rotate right at the imbalanced node



# Case RL in General

- Imbalance caused by inserting in the right child's left subtree
- Rotate right at the right child
- Rotate left at the imbalanced node



# Insert Summary

- After a BST insertion, update the heights of the node's ancestors
- Check for imbalance
- If there's imbalance then at the deepest root of imbalance:
    - 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