

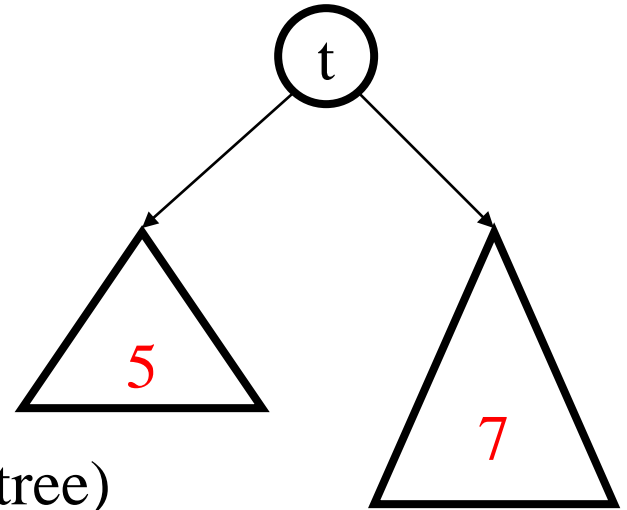
CPSC 221: Data Structures

Balanced BST (AVL Trees)

Alan J. Hu

(Using mainly Steve Wolfman's Slides)

Balance



- Balance

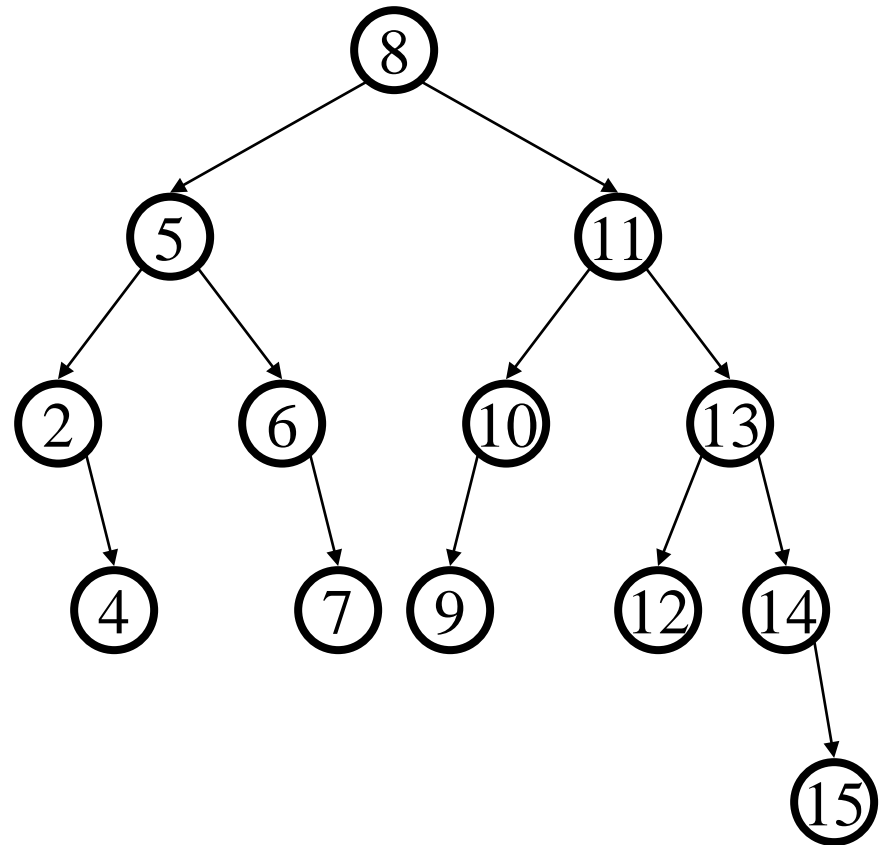
- $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$
- zero everywhere \Rightarrow perfectly balanced
- small everywhere \Rightarrow balanced enough

Balance between -1 and 1 everywhere \Rightarrow
maximum height of $\sim 1.44 \lg n$

AVL Tree

Dictionary Data Structure

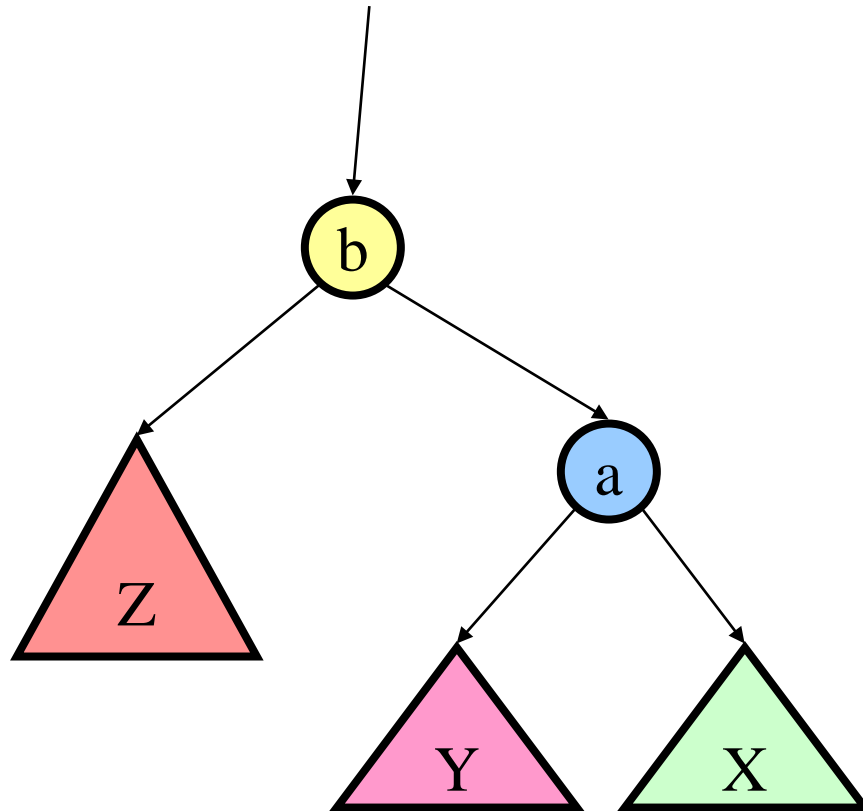
- Binary search tree properties
 - binary tree property
 - search tree property
- Balance property
 - balance of every node is:
-1, 0, or 1
 - result:
 - depth is $\Theta(\log n)$



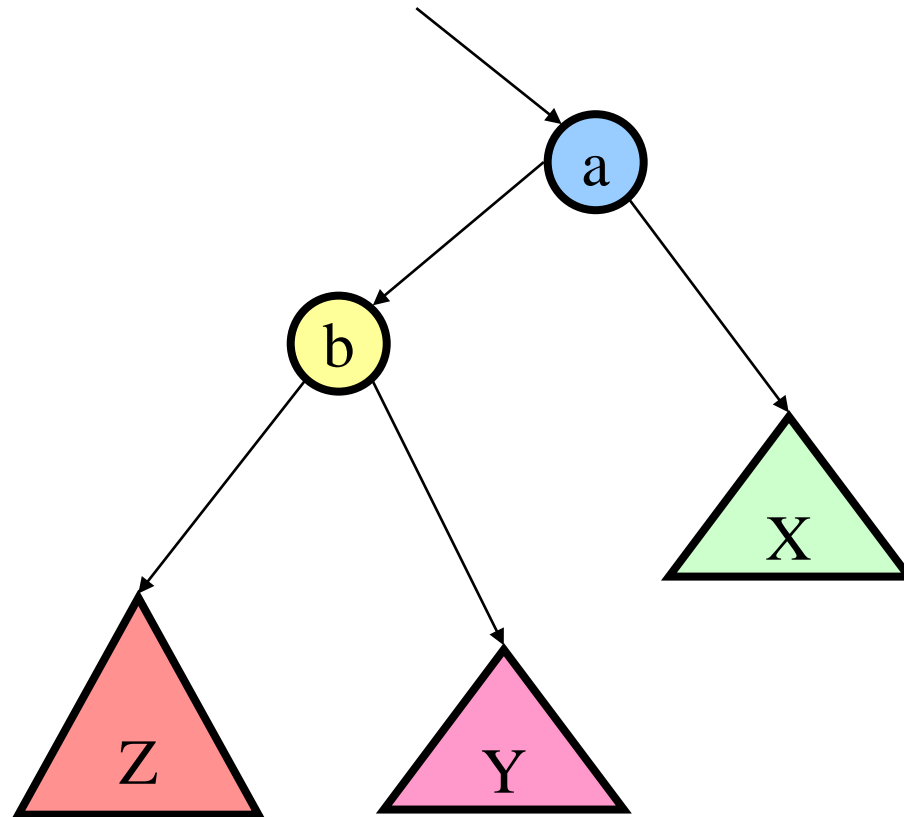
But, How Do We Stay Balanced?

- What do you do if you pick something up out of balance?

Rotation Intuition: Before



Rotation Intuition: After



Time Complexity of Rotation?

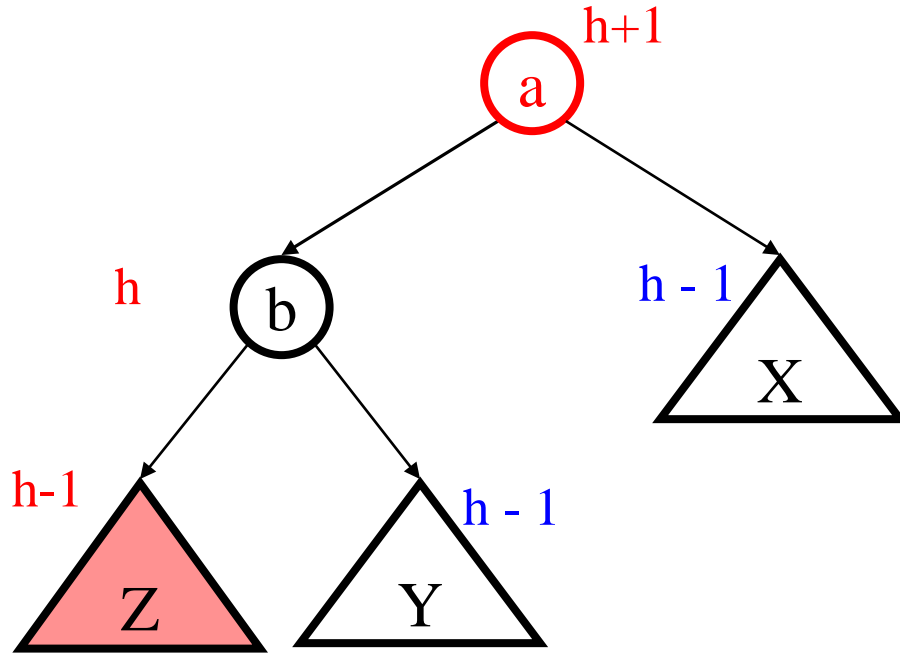
- $O(1)$?
- $O(\lg n)$?
- $O(n)$?
- $O(n \lg n)$?
- $O(n^2)$?
- All of the above?

AVL Tree Insertion

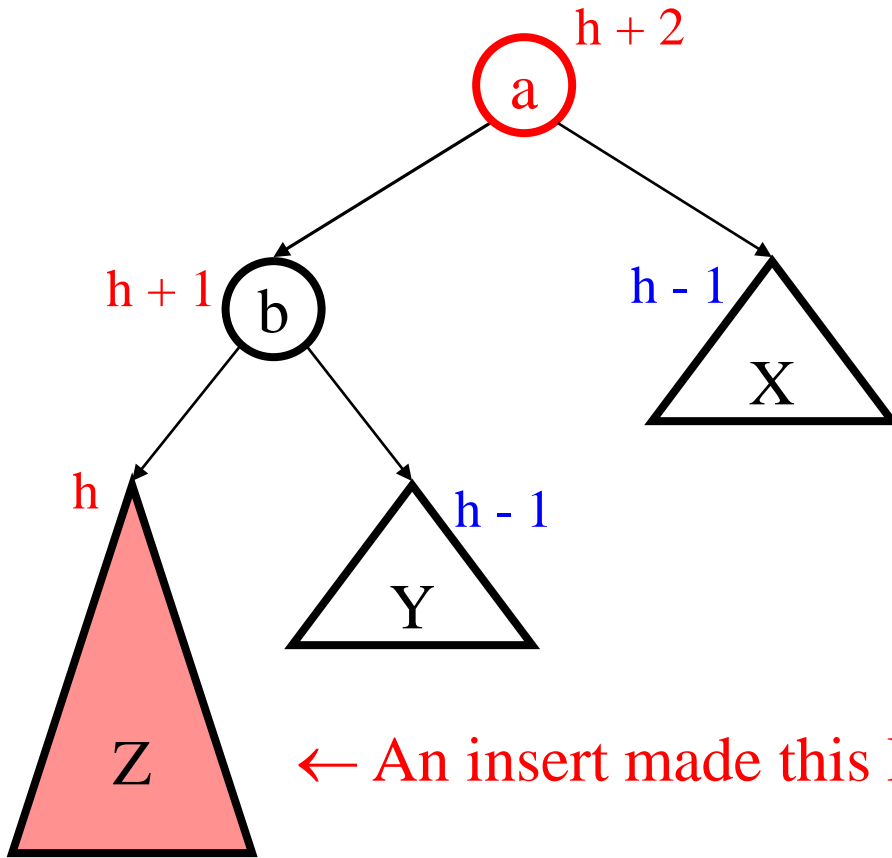
- Pre-Condition: Inserting an item into a correct AVL tree (binary, search tree, AVL balanced)
- Do a normal BST insert.
- This might upset balance property, so do rotation(s) as needed to restore AVL balance.

Turns out, you need at most two rotations!

Before Insertion (Single Rotation)

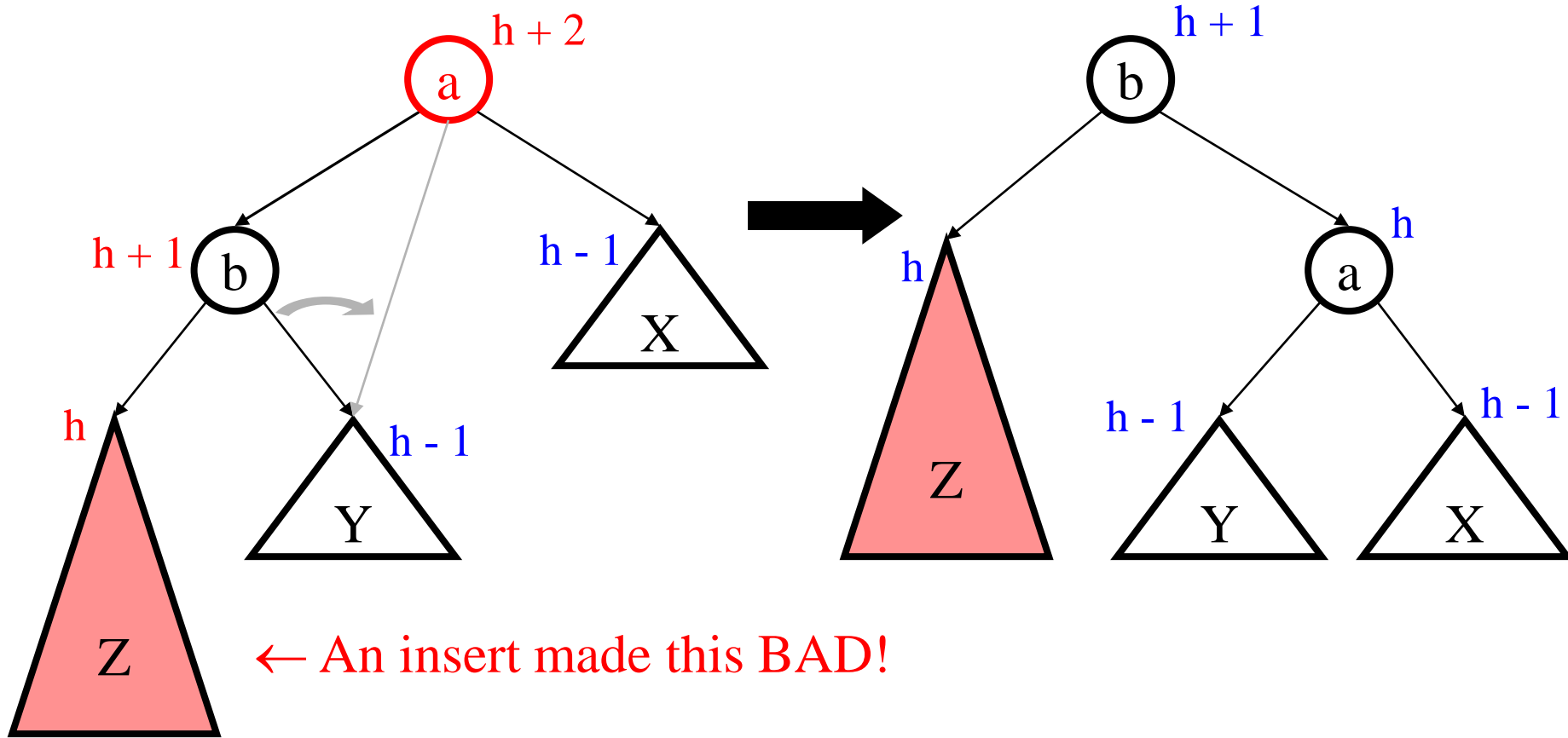


After Insertion (Single Rotation)



← An insert made this BAD!

General Single Rotation

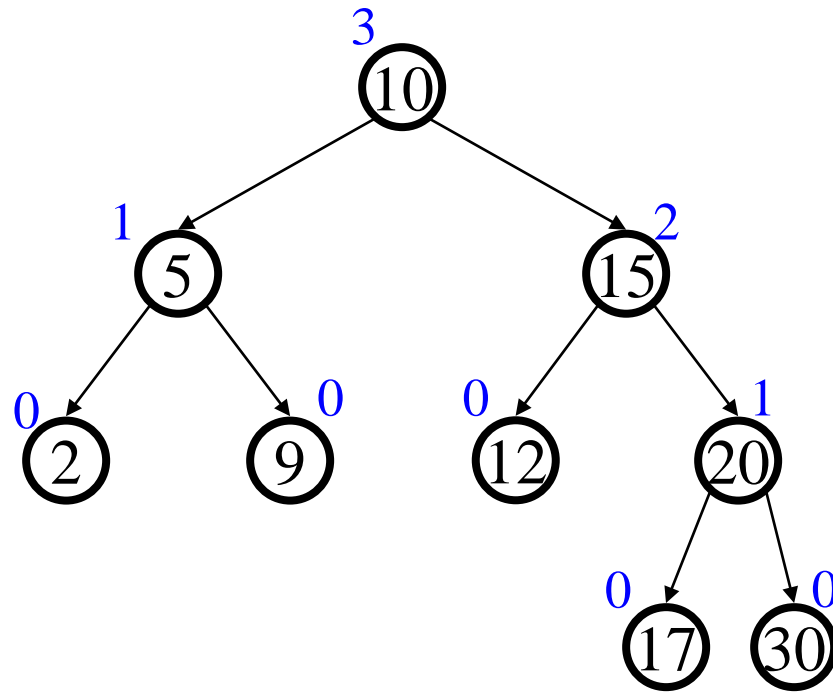


- After rotation, subtree's height same as before insert!
- Height of all ancestors unchanged.

So?

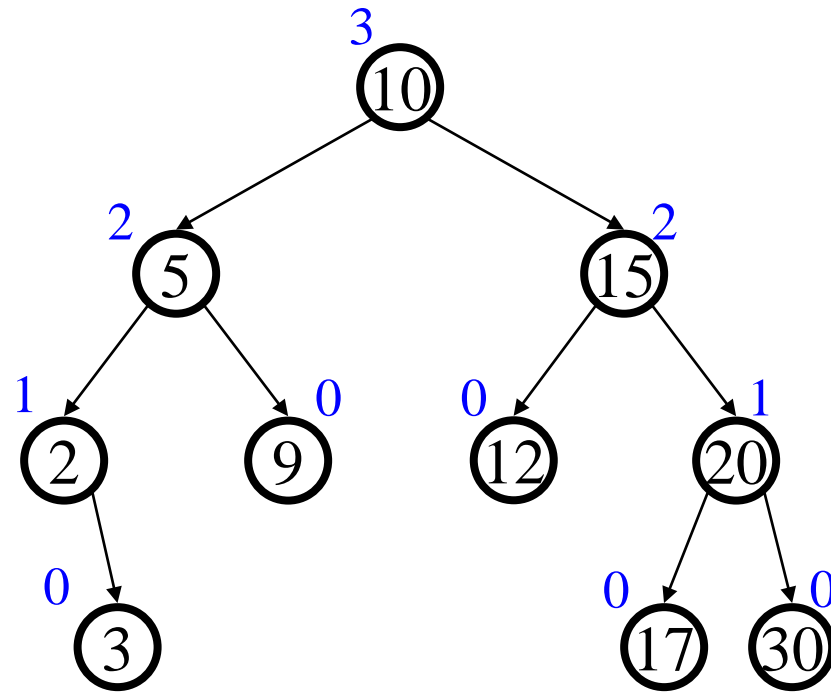
Example: Easy Insert

Insert(3)

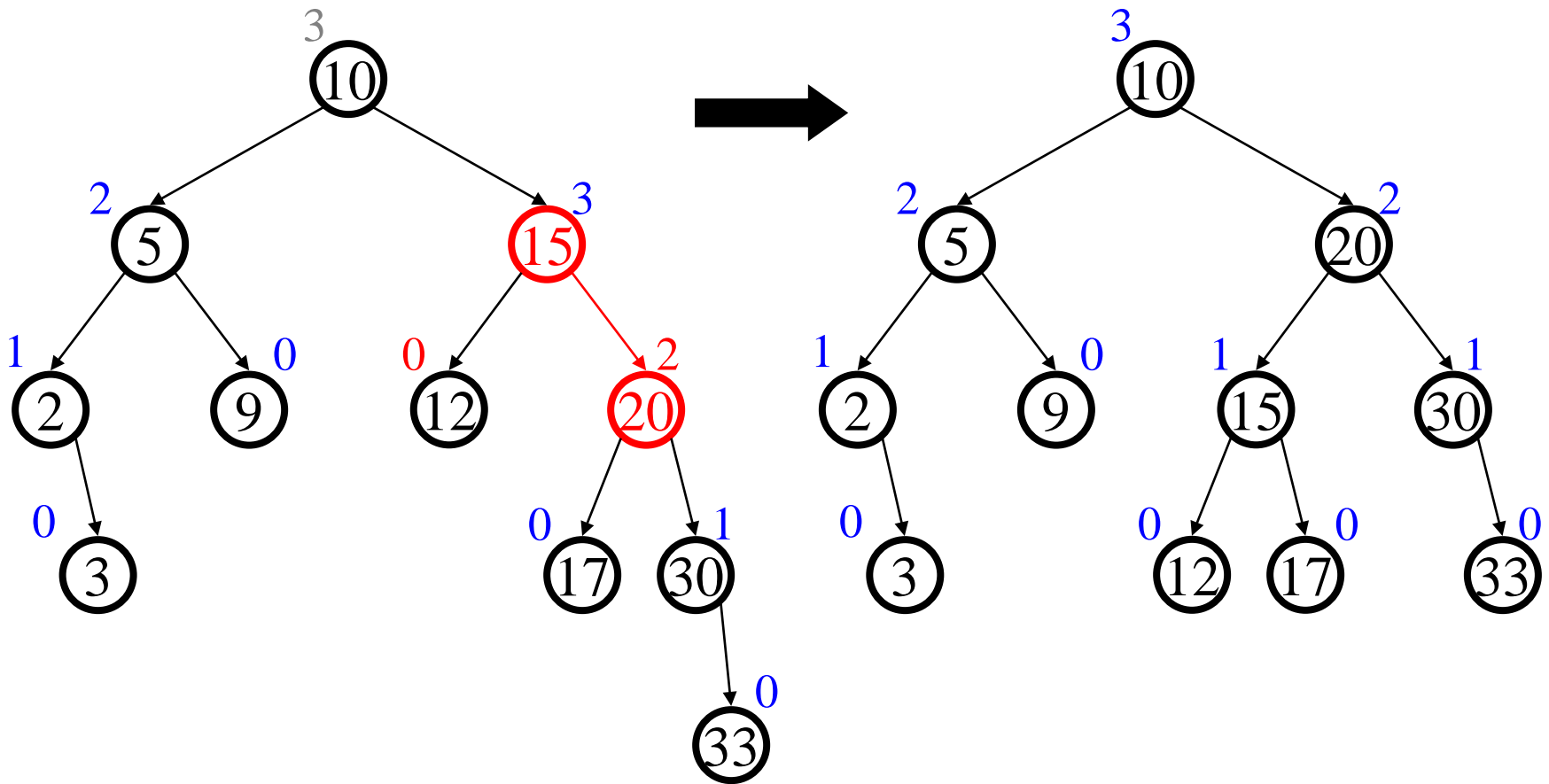


Hard Insert (Bad Case #1)

Insert(33)

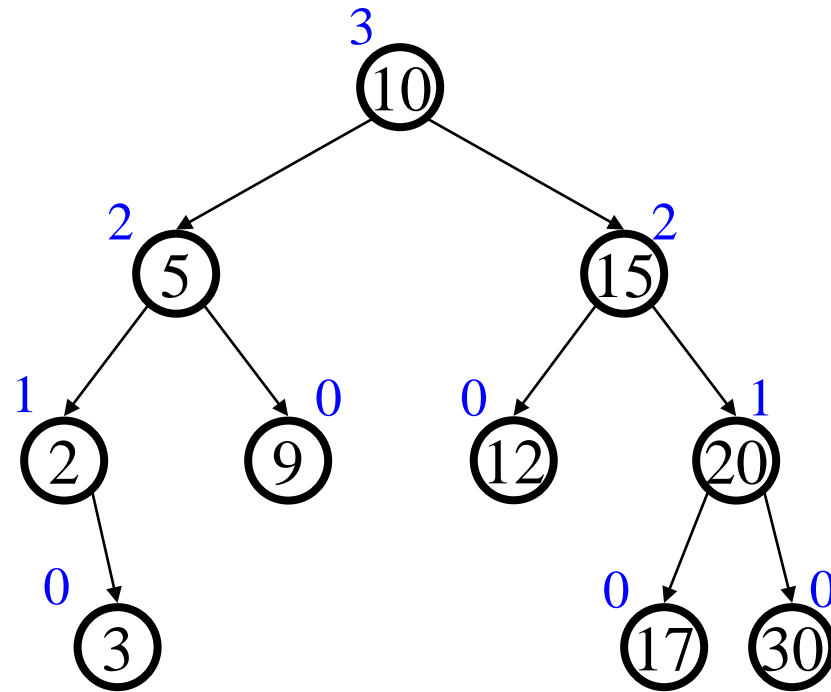


Single Rotation

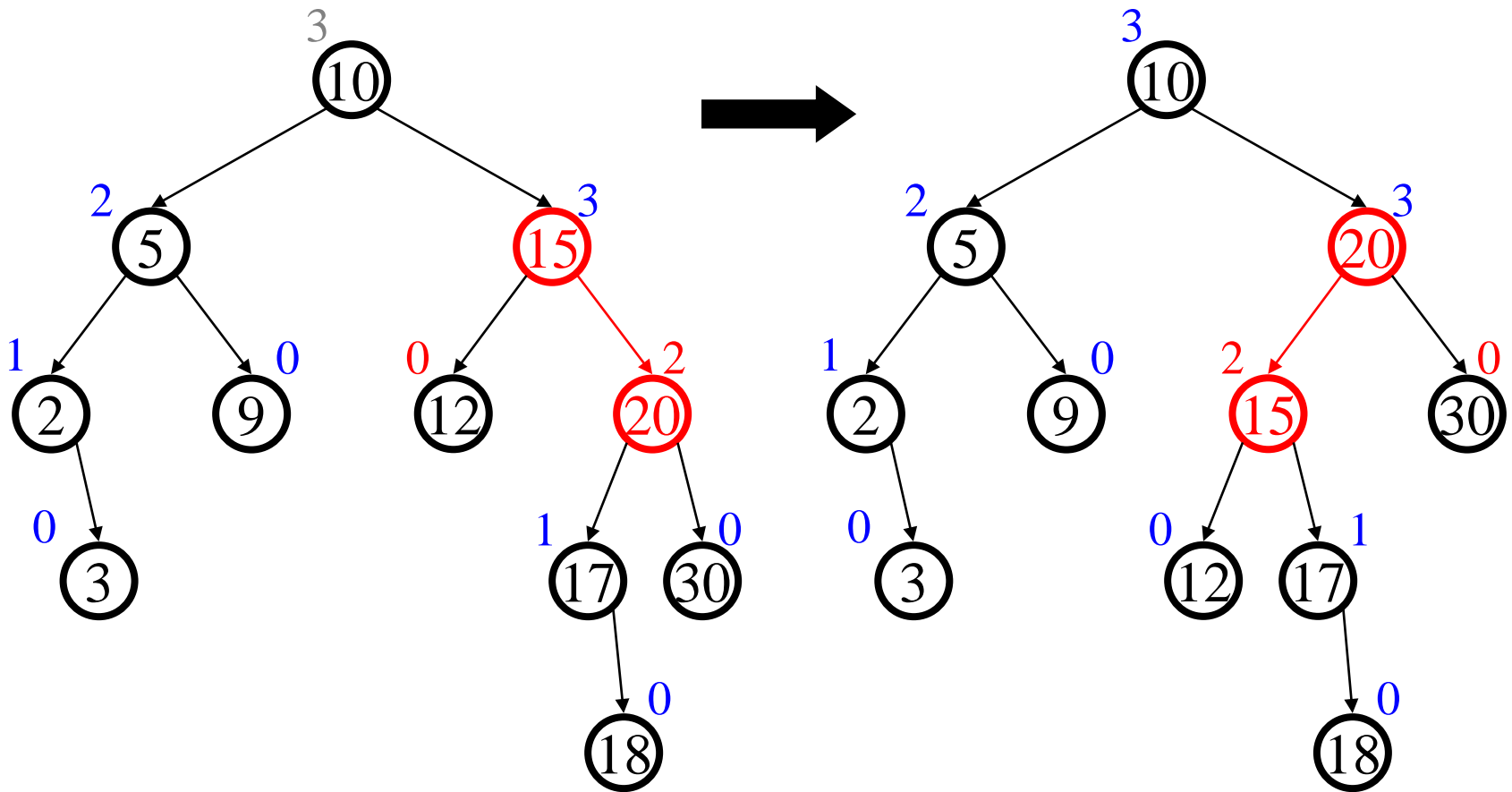


Hard Insert (Bad Case #2)

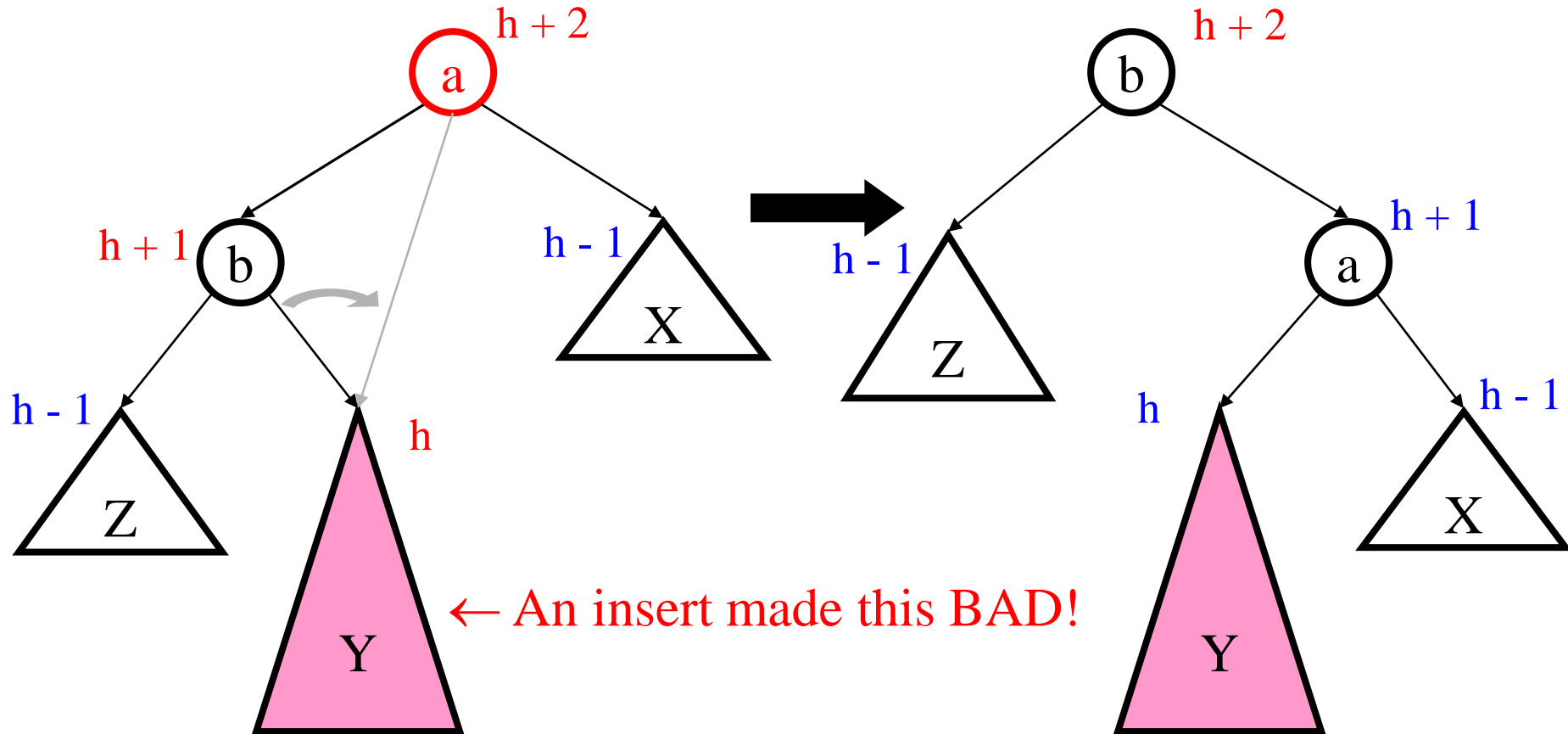
Insert(18)



Single Rotation (oops!)

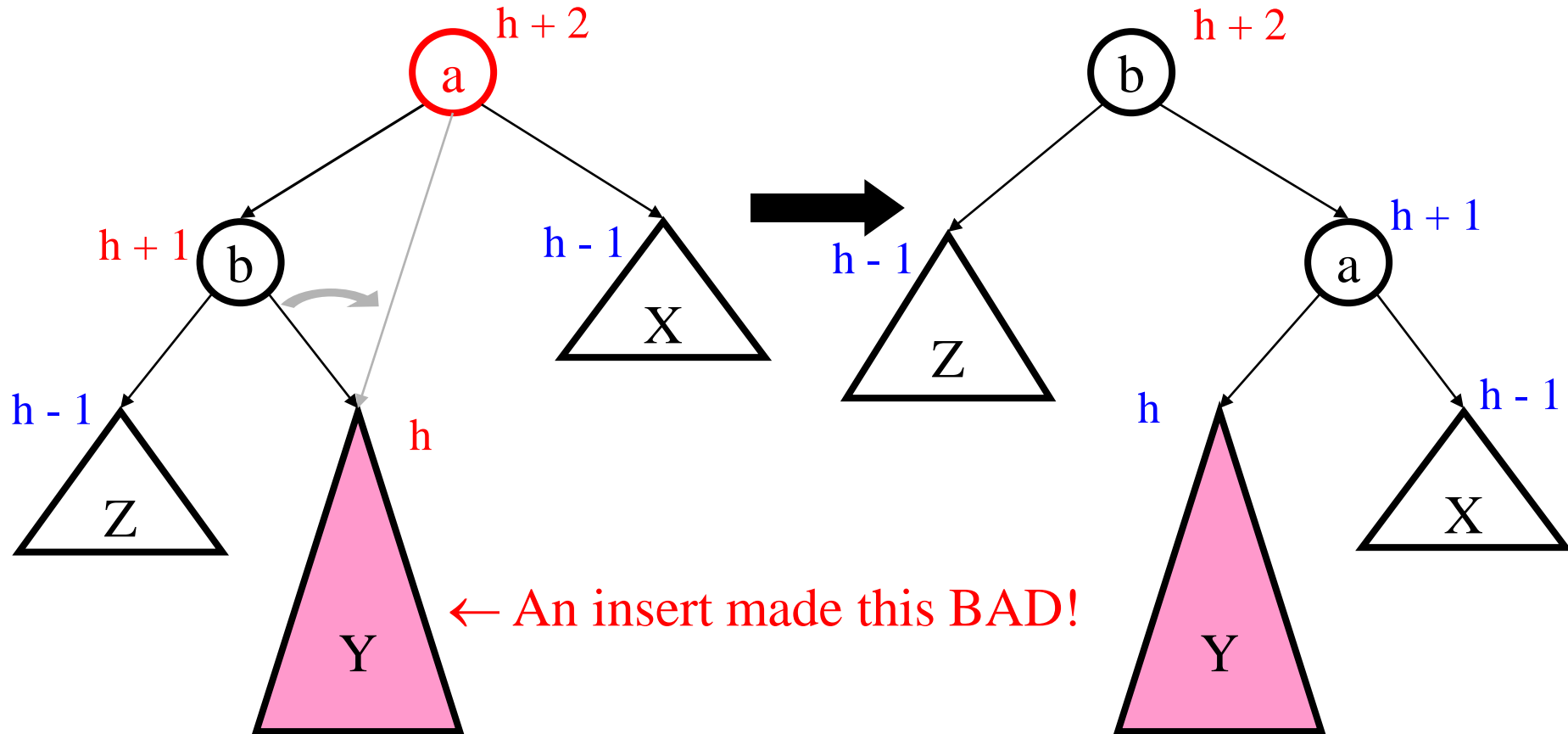


When Single Rotation Doesn't Help



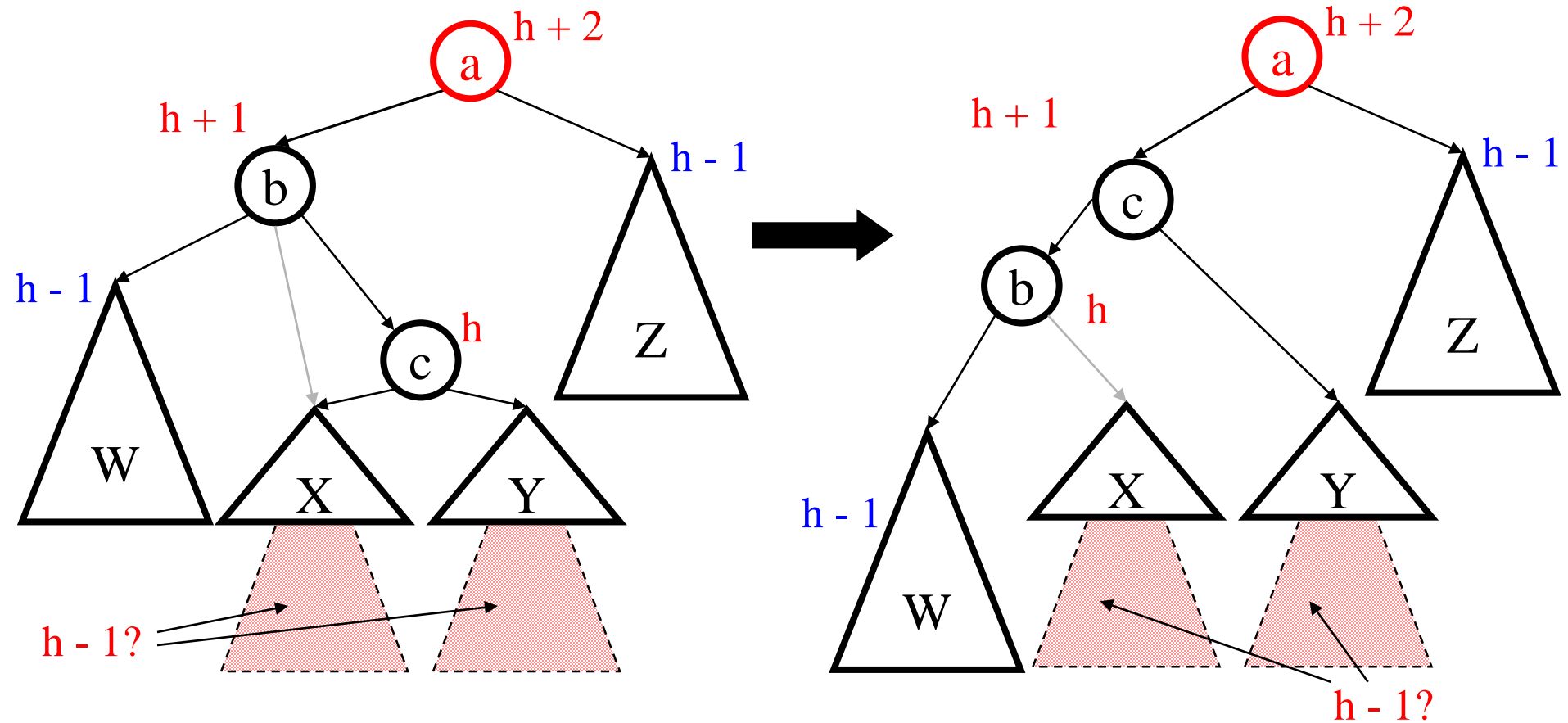
- After rotation, still unbalanced!
- What can you do?

When Single Rotation Doesn't Help



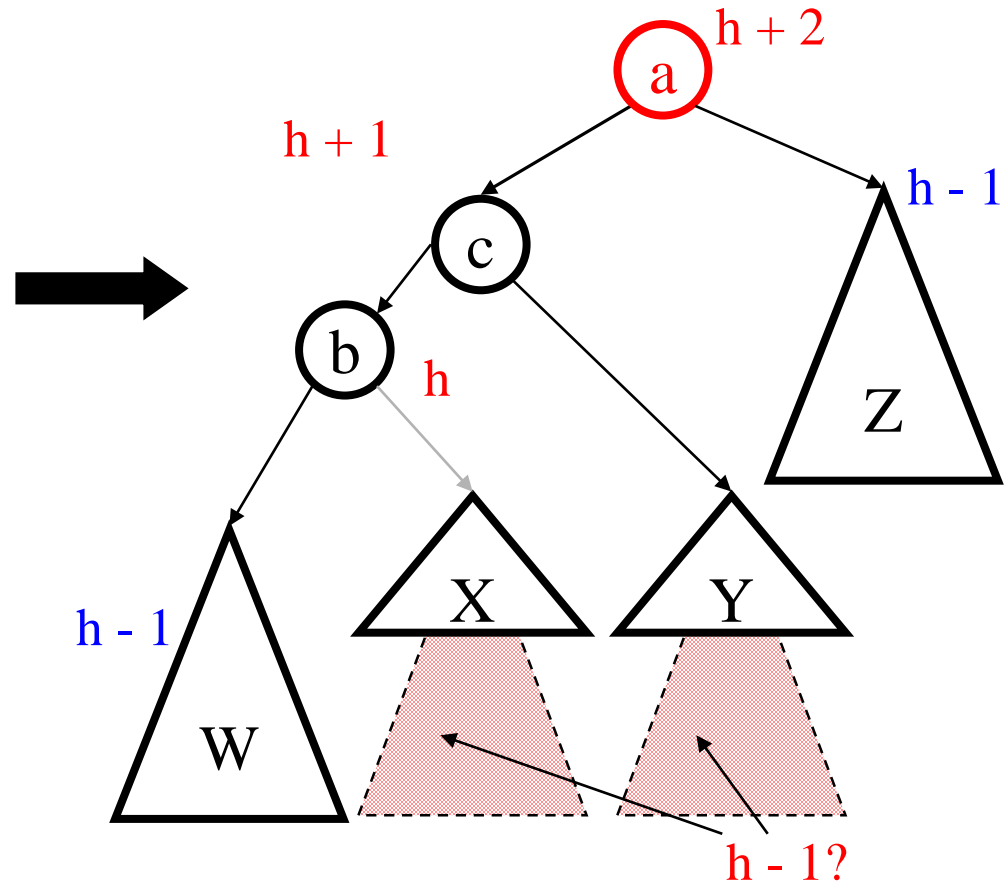
- After rotation, still unbalanced!
- The problem is **Y** is too heavy, so rotate stuff out of **Y**!

Double Rotation Part 1



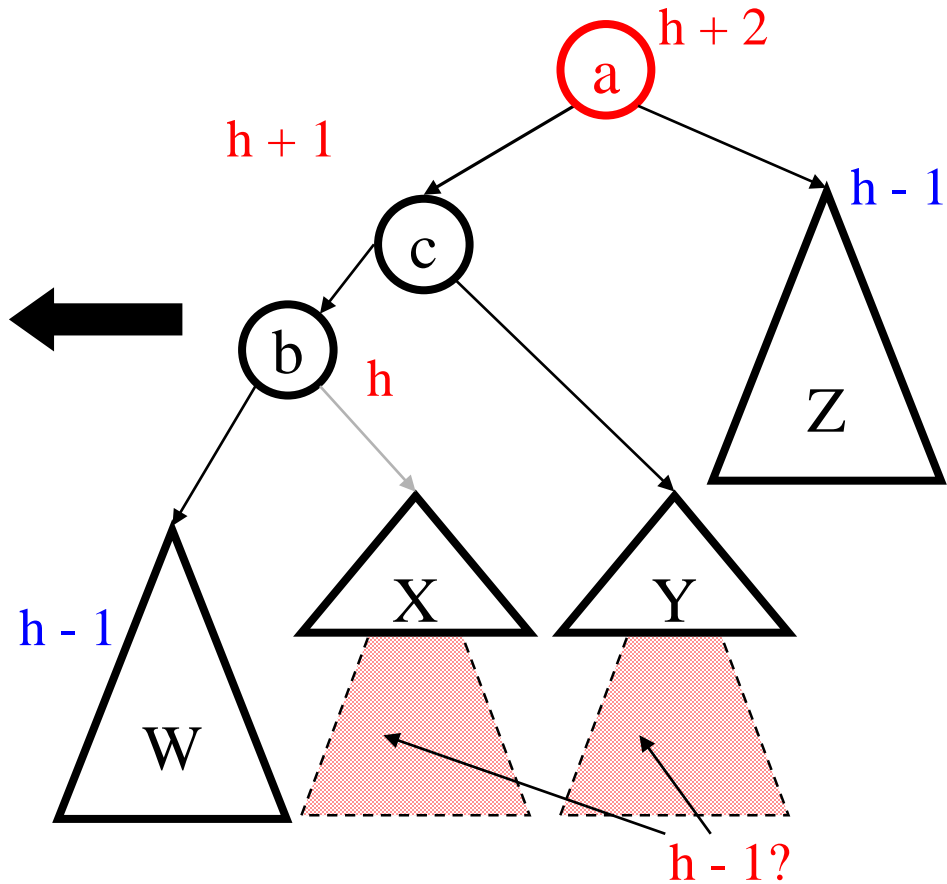
- First, do a single rotation farther down, to split up the big subtree.

Double Rotation Part 1



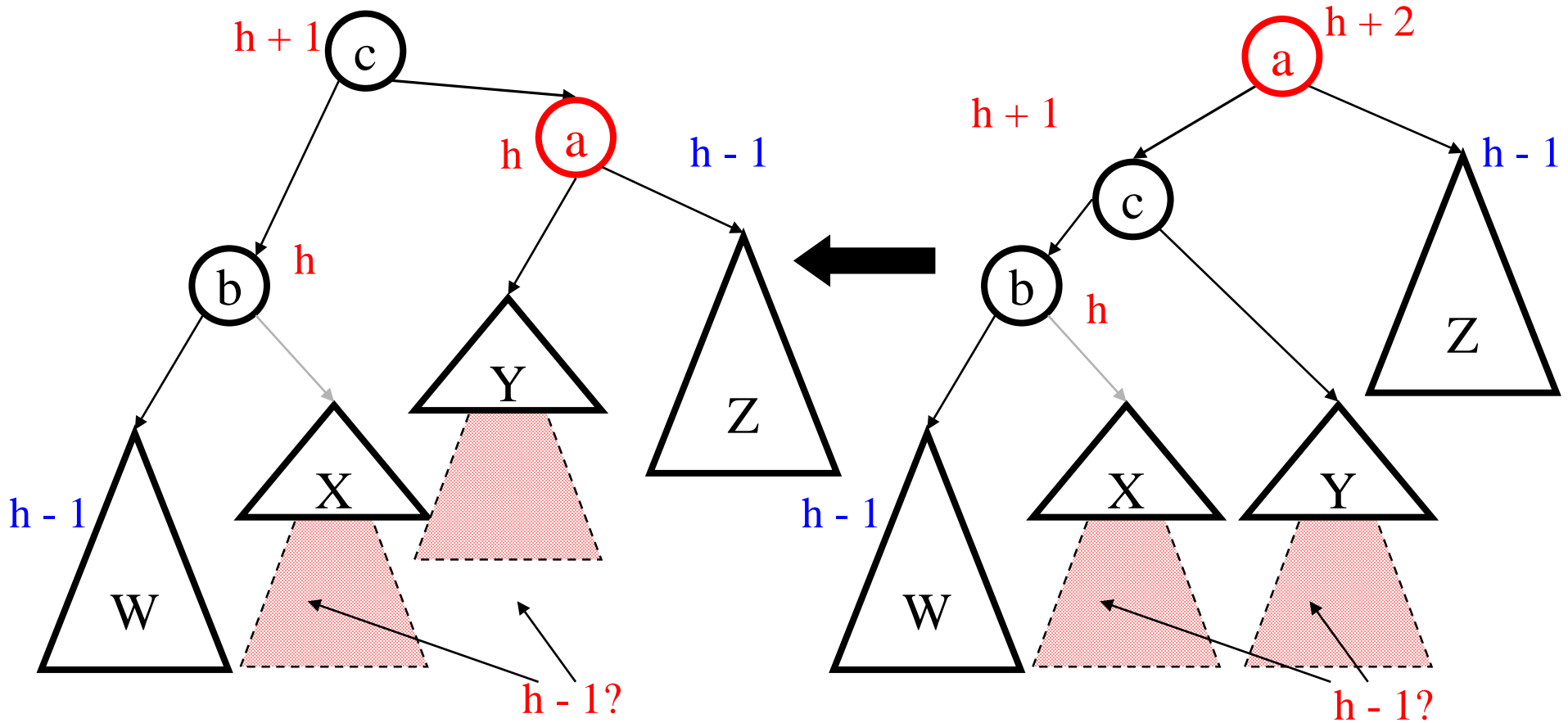
- First, do a single rotation farther down, to split up the big subtree.

Double Rotation Part 2



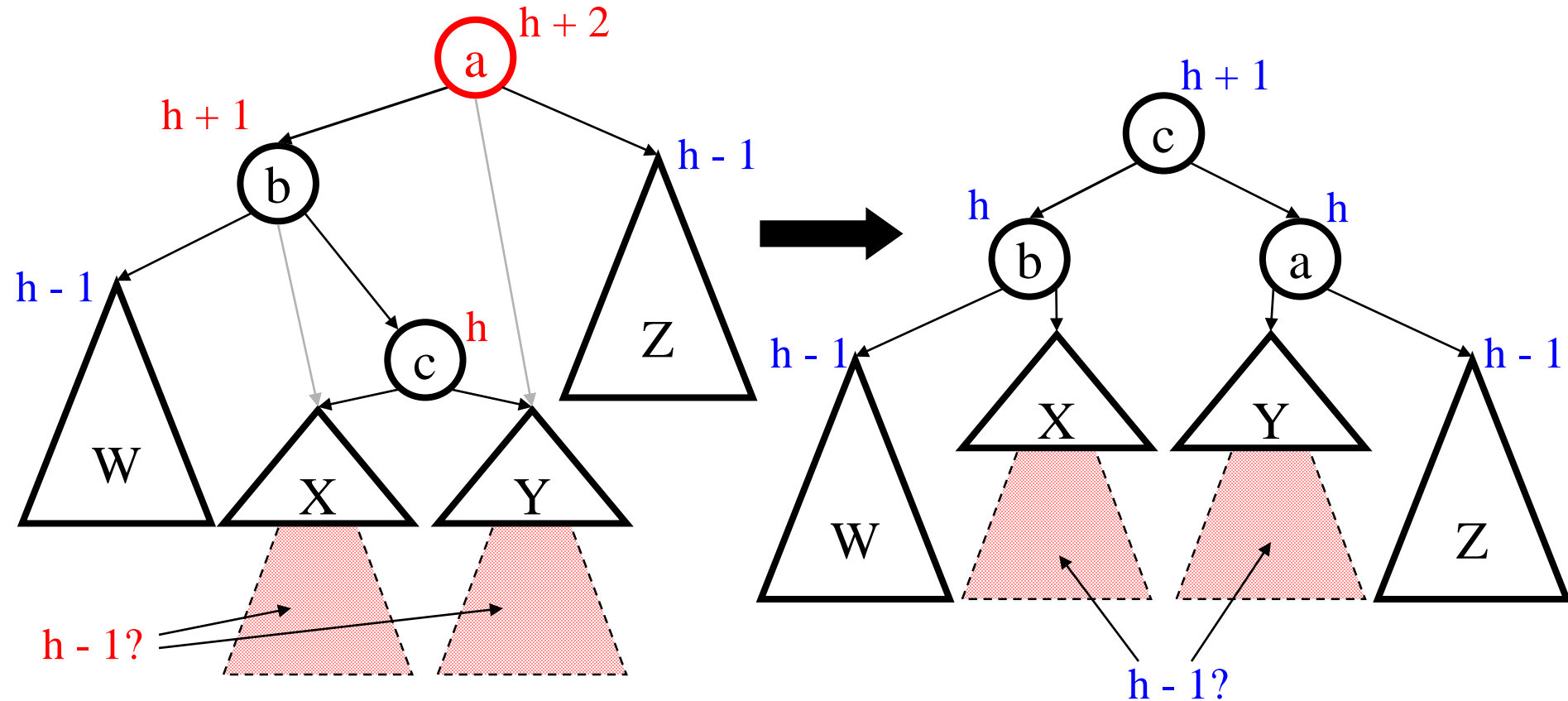
- Now, we can do the originally planned rotation, and not have too much height shift over...

Double Rotation Part 2



- Now, we can do the originally planned rotation, and not have too much height shift over...

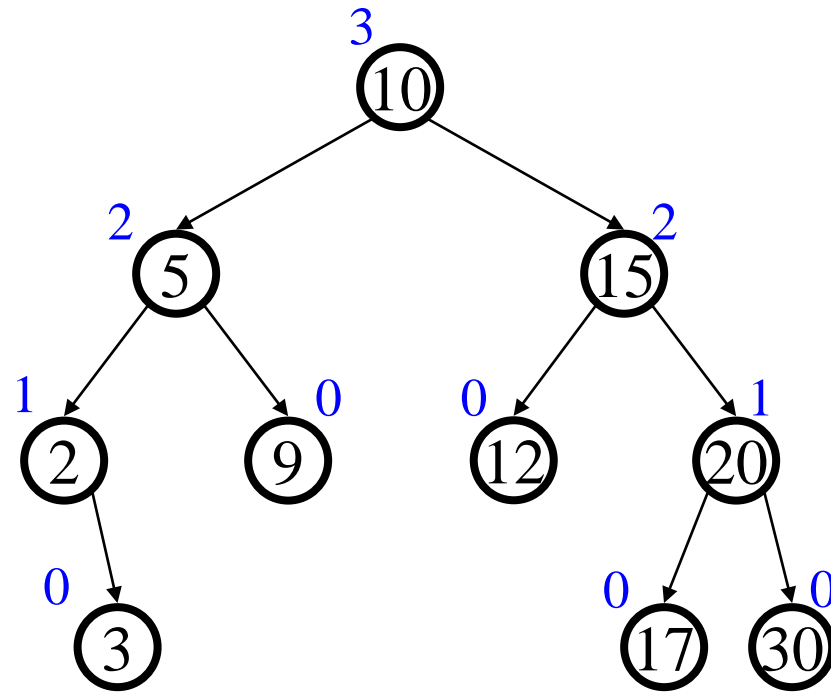
General Double Rotation



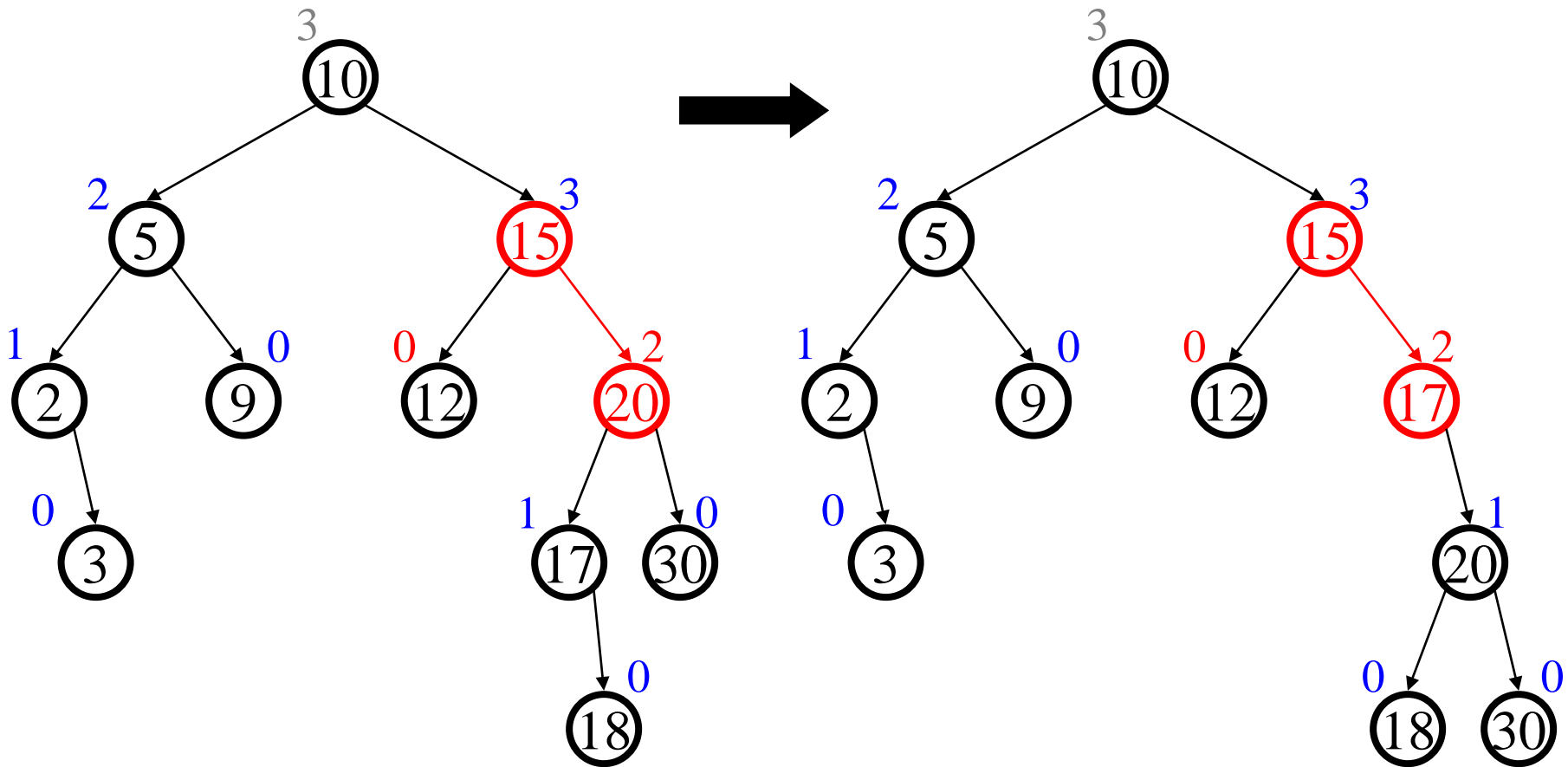
- Height of subtree **still** the same as it was before insert!
- Height of all ancestors unchanged.

Hard Insert (Bad Case #2)

Insert(18)

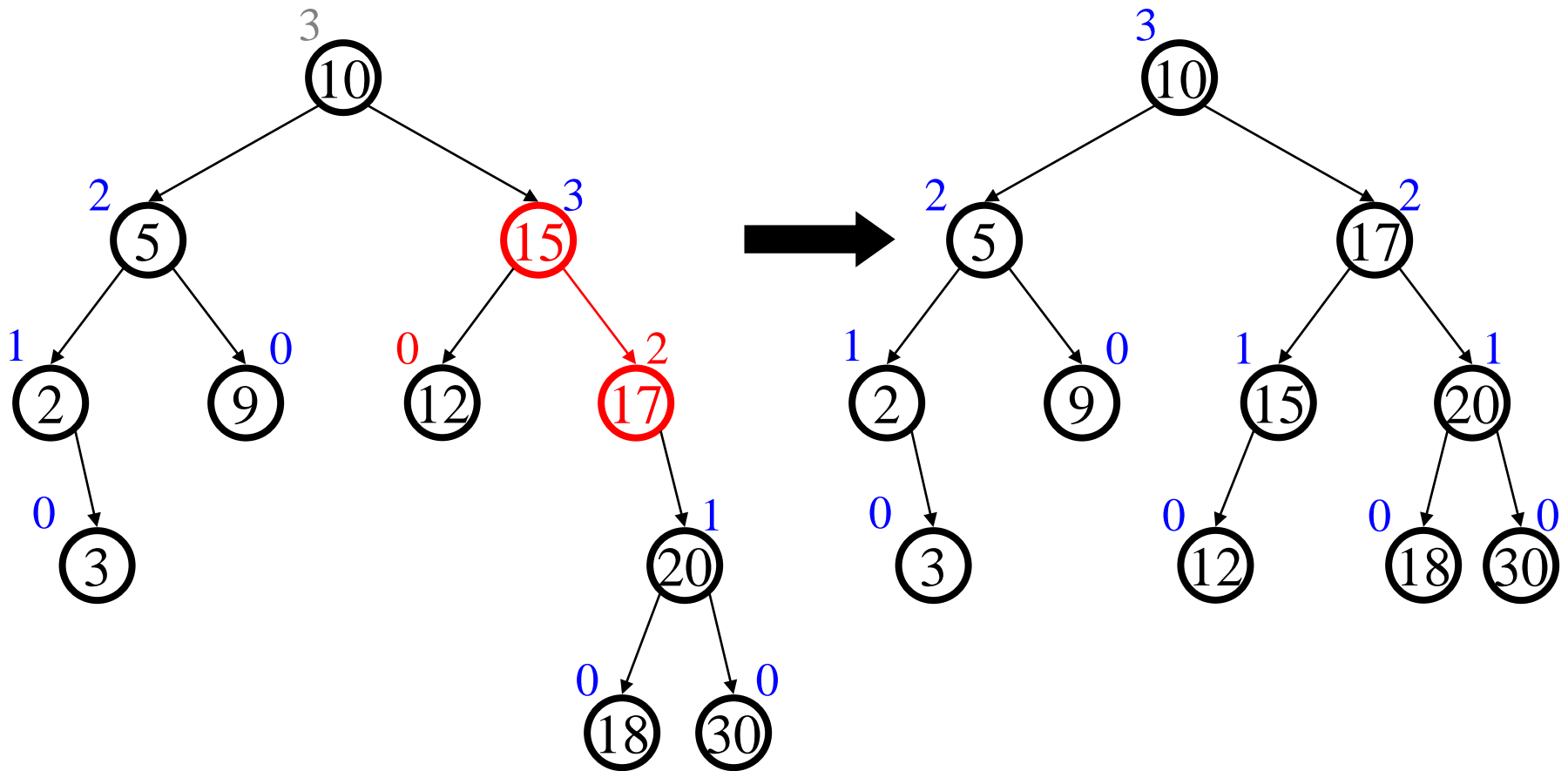


Double Rotation (Step #1)



Look familiar?

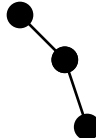
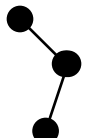
Double Rotation (Step #2)



Today's Outline

- Addressing one of our problems
- Single and Double Rotations
- **AVL Tree Implementation**

Insert Algorithm

- Find spot for value
- Hang new node
- Search back up for imbalance
- If there is an imbalance:
 -  case #1: Perform single rotation and exit
 -  case #2: Perform double rotation and exit

Mirrored cases also possible

AVL Algorithm Revisited

- Recursive

1. Search downward for spot
2. Insert node
3. Unwind stack, correcting heights
 - a. If imbalance #1, single rotate
 - b. If imbalance #2, double rotate

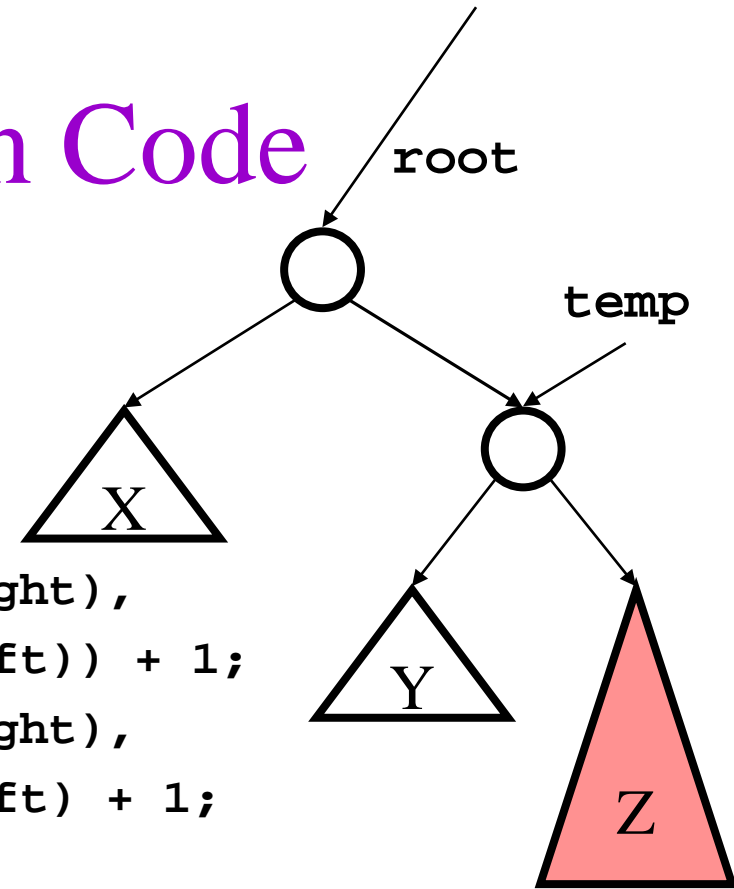
- Iterative

1. Search downward for spot, **stacking parent nodes**
2. Insert node
3. Unwind stack, correcting heights
 - a. If imbalance #1, single rotate **and exit**
 - b. If imbalance #2, double rotate **and exit**

(Parent pointers make iterative version easier.)

Single Rotation Code

```
void RotateLeft(Node *& root) {  
    Node * temp = root->right;  
    root->right = temp->left;  
    temp->left = root;  
    root->height = max(height(root->right),  
                       height(root->left)) + 1;  
    temp->height = max(height(temp->right),  
                     height(temp->left) + 1);  
    root = temp;  
}
```



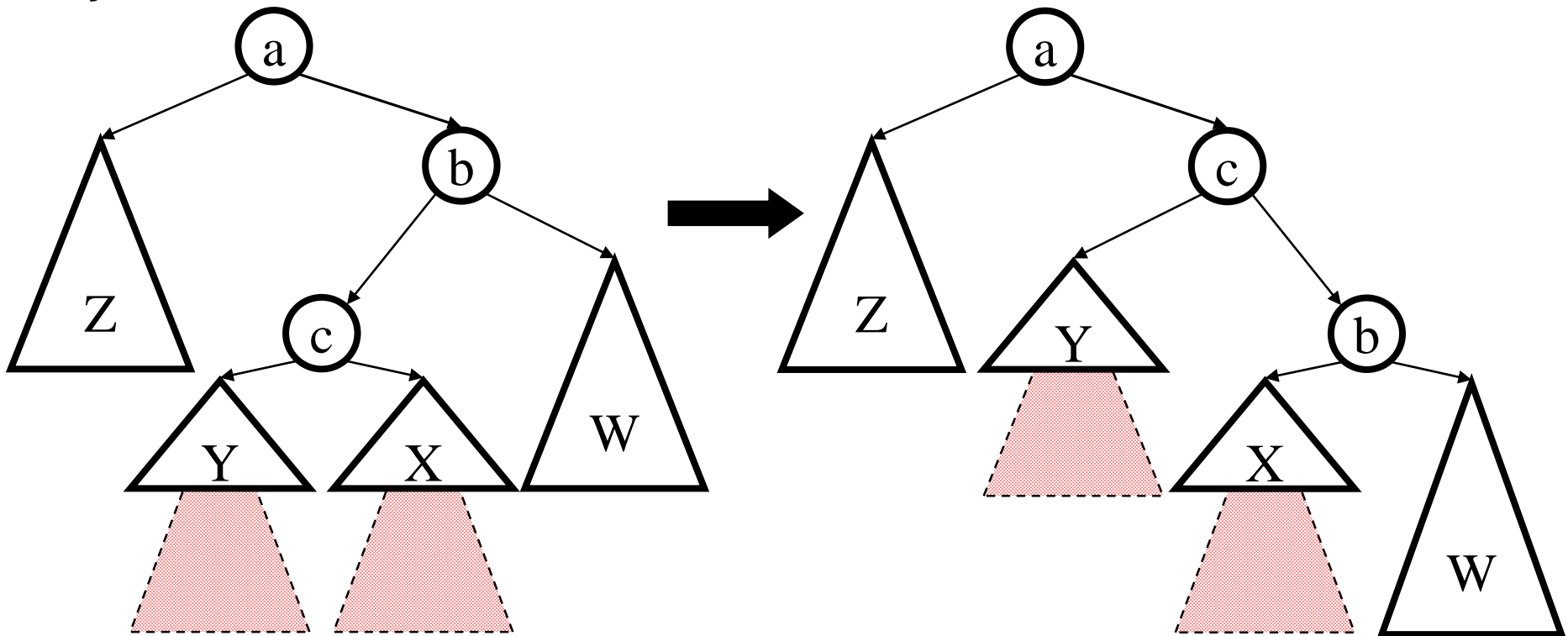
(The “height” function returns -1 for a NULL subtree or the “height” field of a non-NULL subtree.)

Notice that root is a reference parameter. MUST be the “correct” pointer.

Double Rotation Code

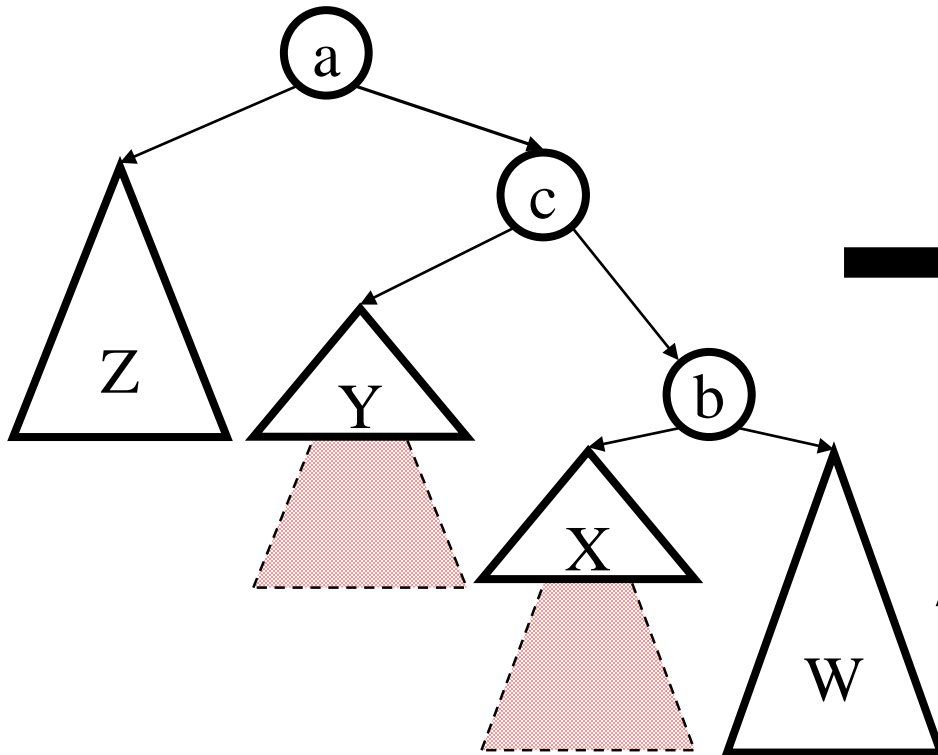
```
void DoubleRotateLeft(Node *& root) {  
    RotateRight(root->right);  
    RotateLeft(root);  
}
```

First Rotation



Double Rotation Completed

First Rotation



Second Rotation

