CSE 373: Data Structures & Algorithms

More AVL Trees

Riley Porter

Winter 2017

Course Logistics

- HW2 spec had a small typo

- Weekly summaries out soon, hopefully by tomorrow (sorry I got sick this weekend)

Review: AVL Trees

1. Values are correct: it's a BST

2. Structure is correct: AVL balance condition. Left and right subtrees of ***every node*** have heights differing by at most 1

   Result: **Worst-case** depth is O(log *n*)

Review: AVL Operations

If we have an AVL tree, the height is *O*(**log** *n*), so **find** is *O*(**log** *n*)

Maintenance: as we insert and delete elements, we need to keep the tree in balance. We do that with the following steps:

1. **Track balance**
2. **Detect imbalance**
3. **Restore balance**

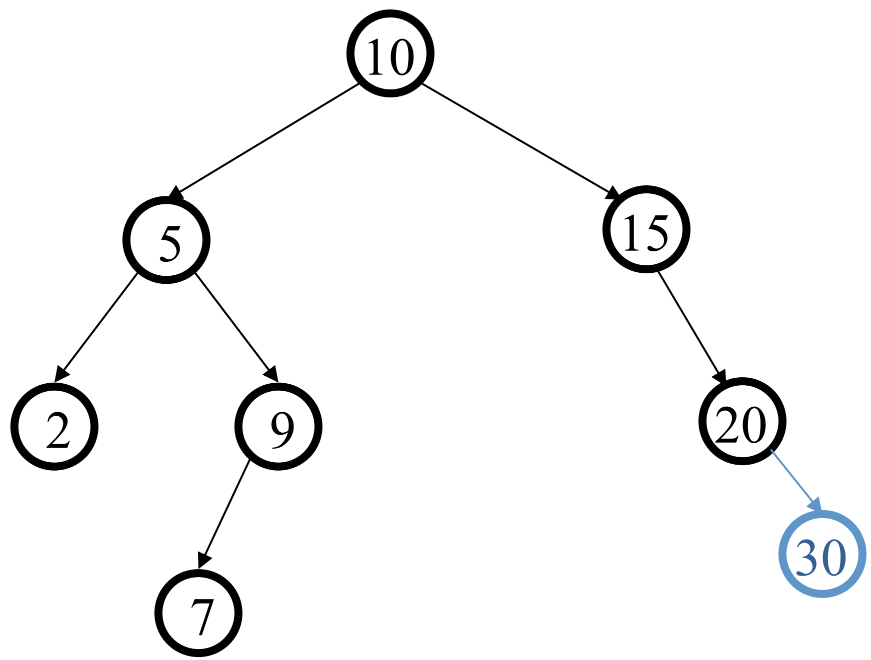
Is this AVL tree balanced?

Yep!

How about after `insert(30)`?

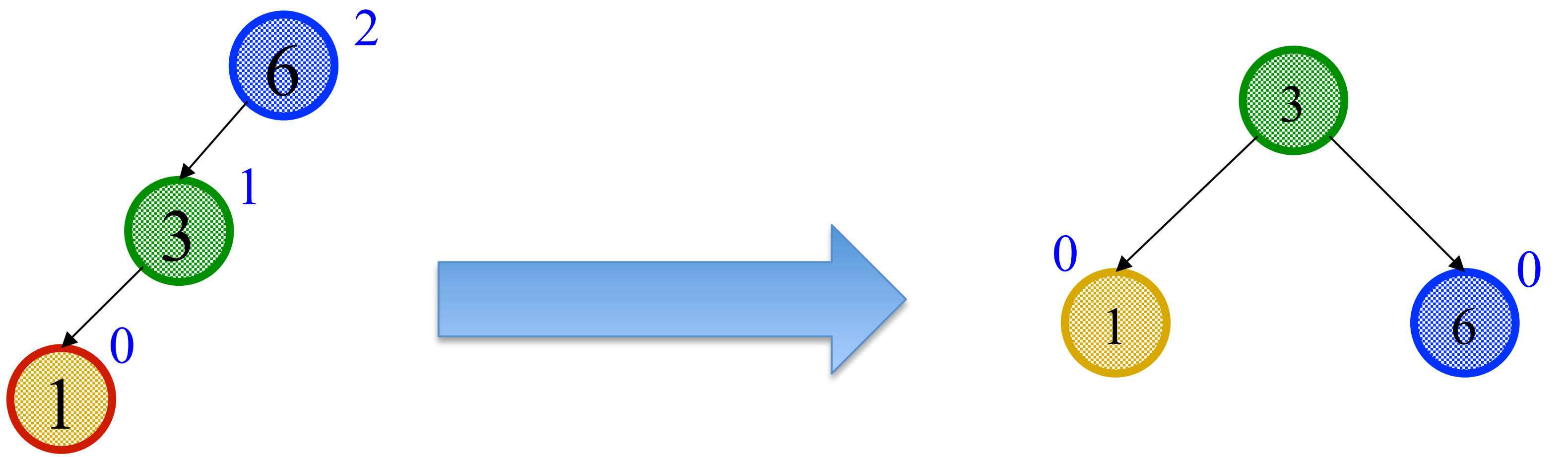
No, now the Balance of 15 is off

Perform a rotation of the subtree
15 -> 20 -> 30

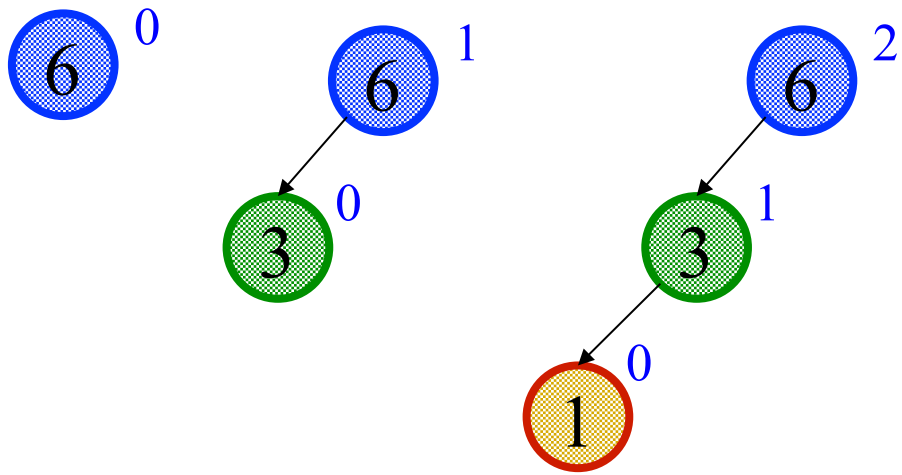


Review: AVL Insert

1. Insert the new node
2. Find and fix imbalances. We "fix" imbalances by doing rotation(s).



Case #1: Example



Insert(6)

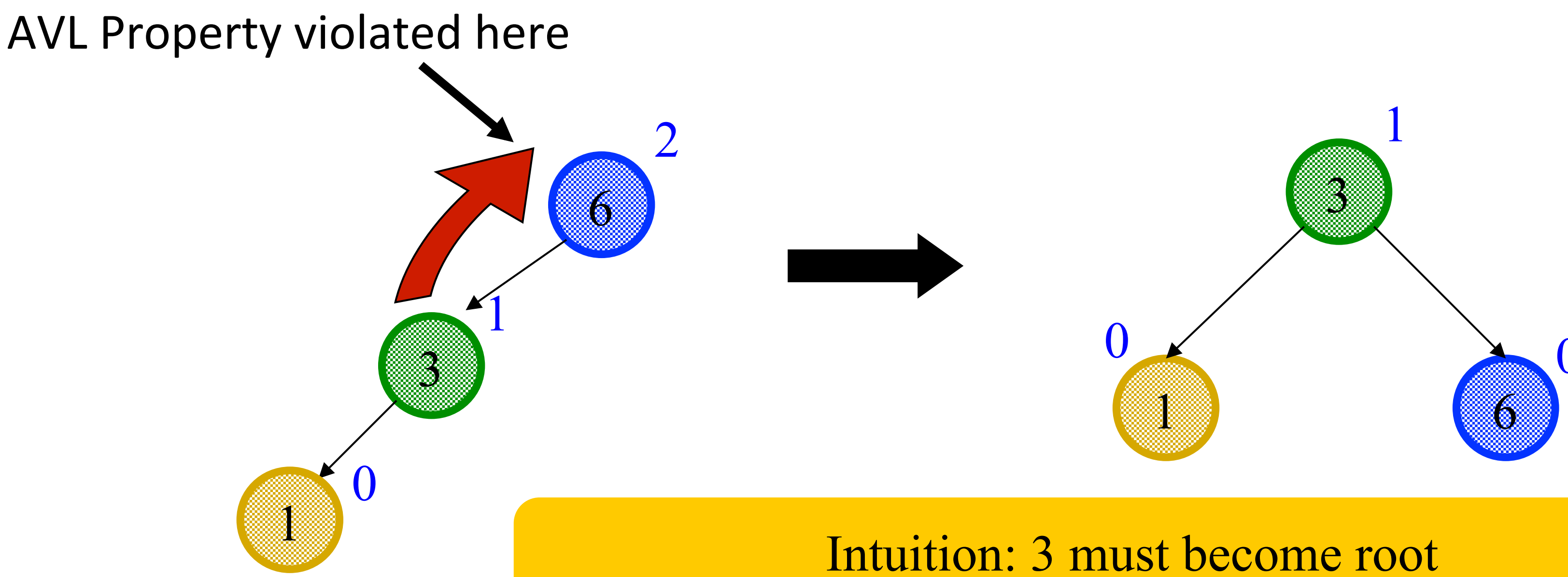
Insert(3)

Insert(1)

Third insertion violates balance property. To maintain balance, we fix the tree with a rotation.

Fix: Apply “Single Rotation”

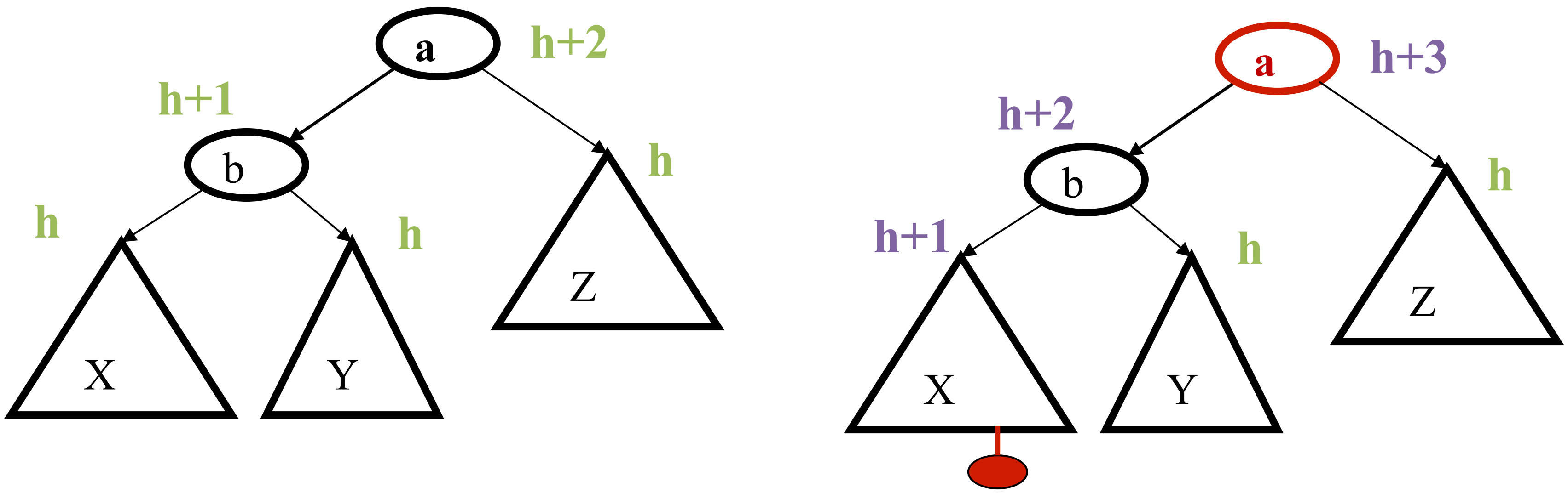
- *Single rotation:* The basic operation we’ll use to rebalance
  - Move child of unbalanced node into parent position
  - Parent becomes the “other” child (always okay in a BST!)
  - Other subtrees move in only way BST allows (next slide)



Intuition: 3 must become root
New parent height is now the old parent’s height before insert

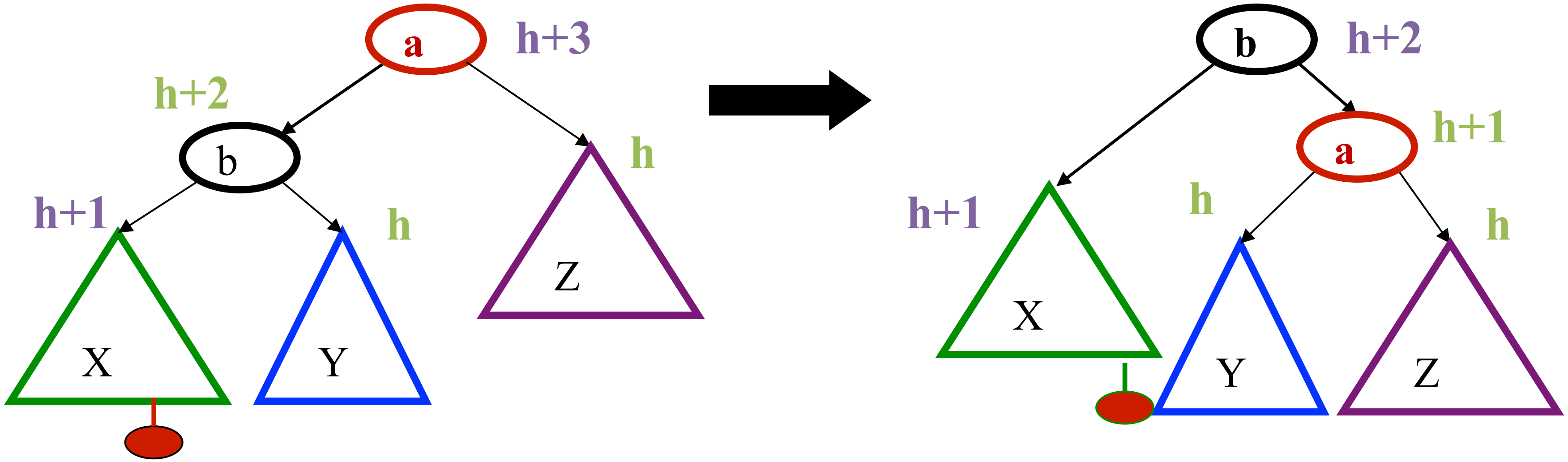
The example generalized

- Node imbalanced due to insertion *somewhere* in **left-left grandchild** that causes an increasing height
  - 1 of 4 possible imbalance causes (other three coming)
- First we did the insertion, which would make ***a*** imbalanced



The general left-left case

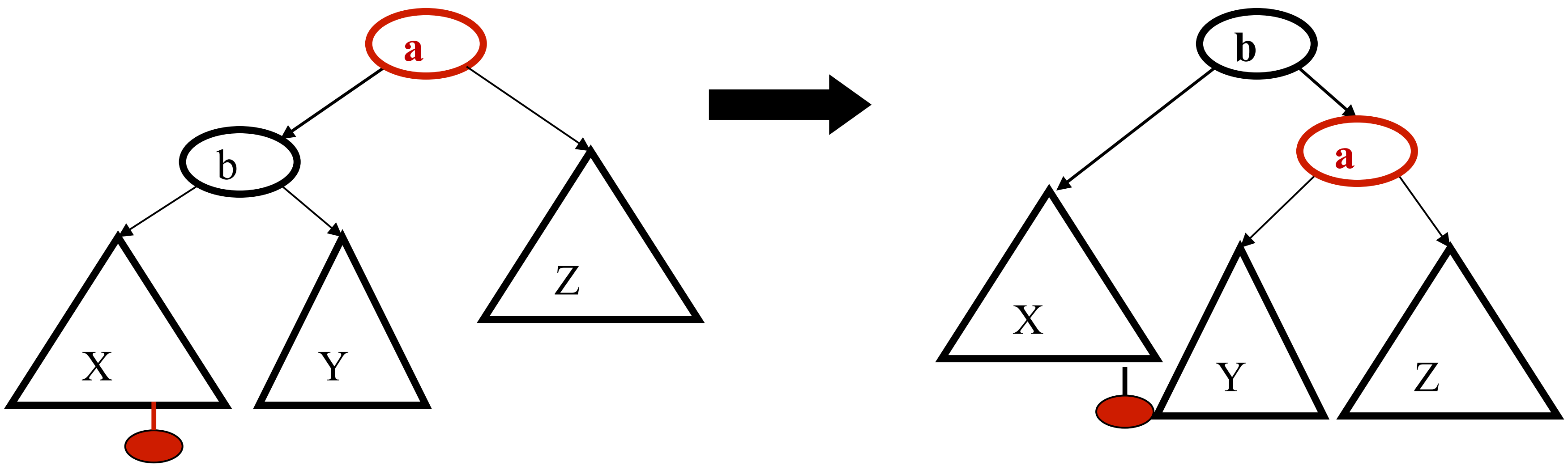
- Node imbalanced due to insertion *somewhere* in **left-left grandchild**
  - 1 of 4 possible imbalance causes (other three coming)
- So we rotate at ***a,***using BST facts: X < b < Y < a < Z



- A single rotation restores balance at the node
  - To same height as before insertion, so ancestors now balanced

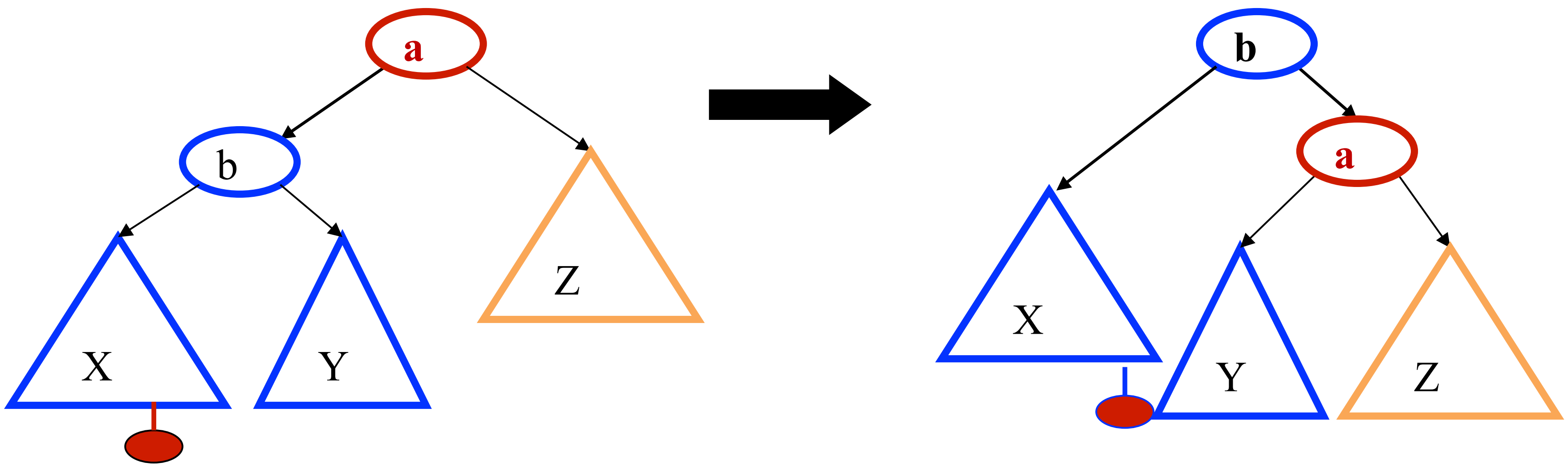
Why does this work: values

- Let's look at the BST property. Identify all values less than and greater than 'a'.



Why does this work: values

- Let's look at the BST property. All values < a are blue. All values > a are orange. They still hold for the BST order property after rotation.



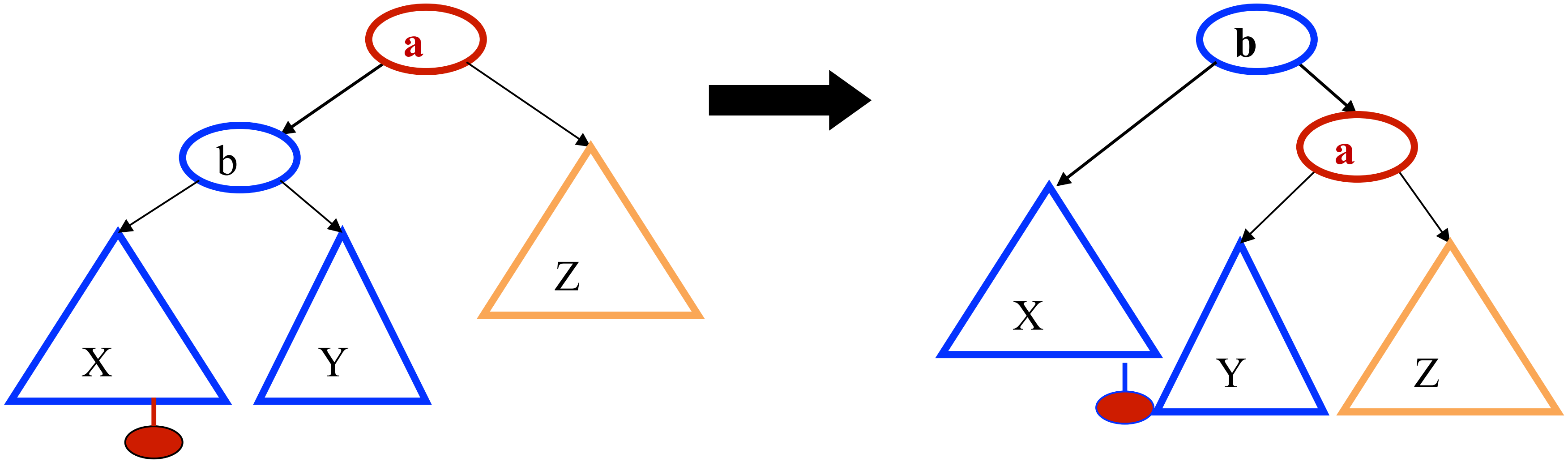
Why does this work: values

Before rotation:

- b < a
- X < a
- Y < a
- Z > a

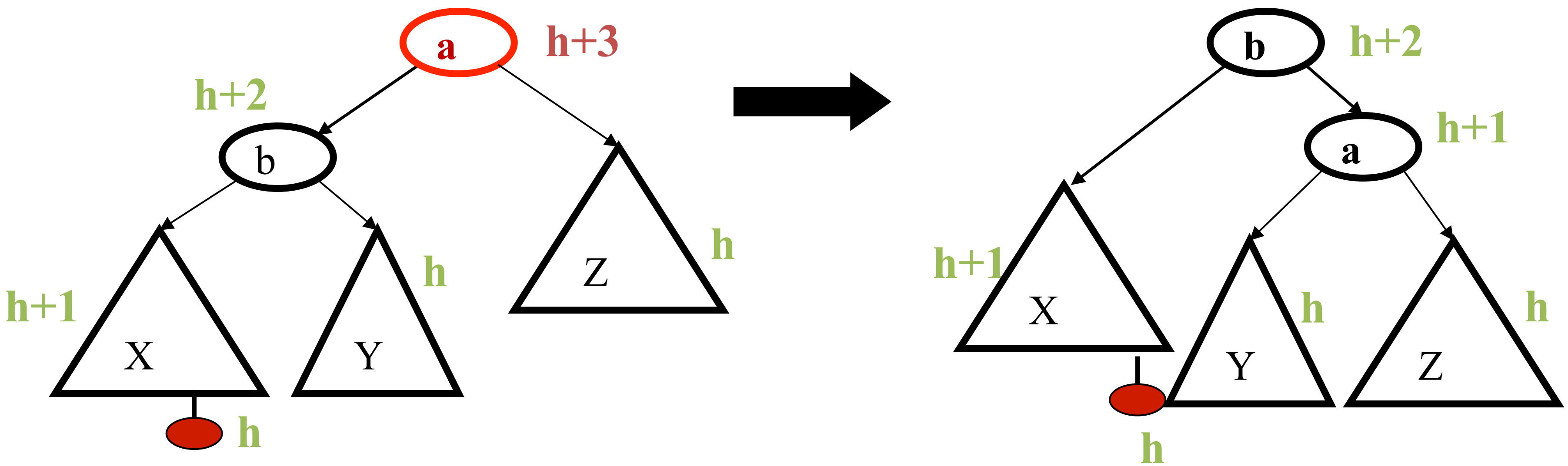
After rotation:

- b < a
- X < a
- Y < a
- Z > a

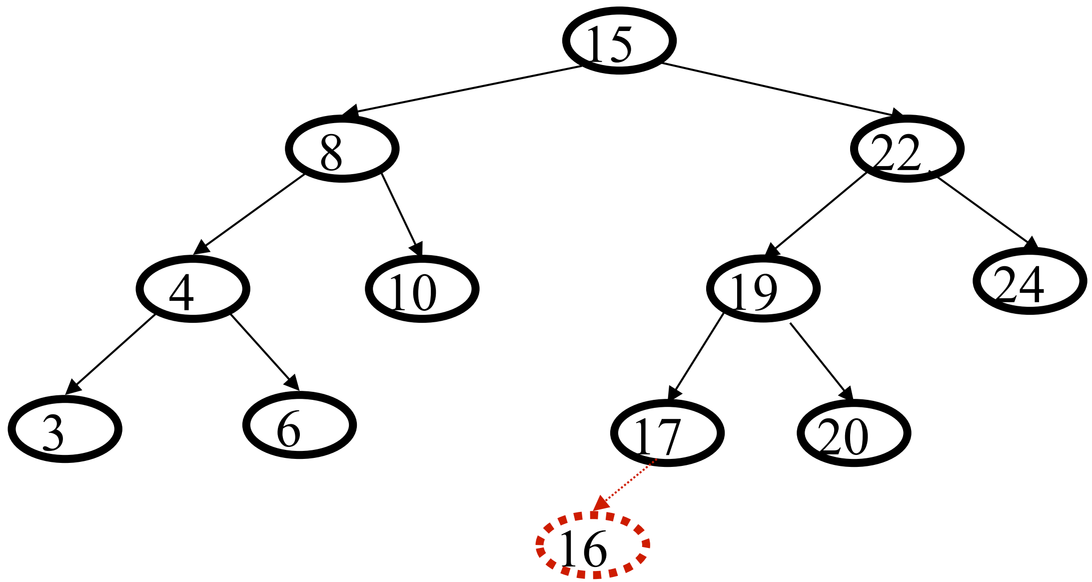


Why does this work: heights

- After inserting the new red node, identify the heights of each subtree.

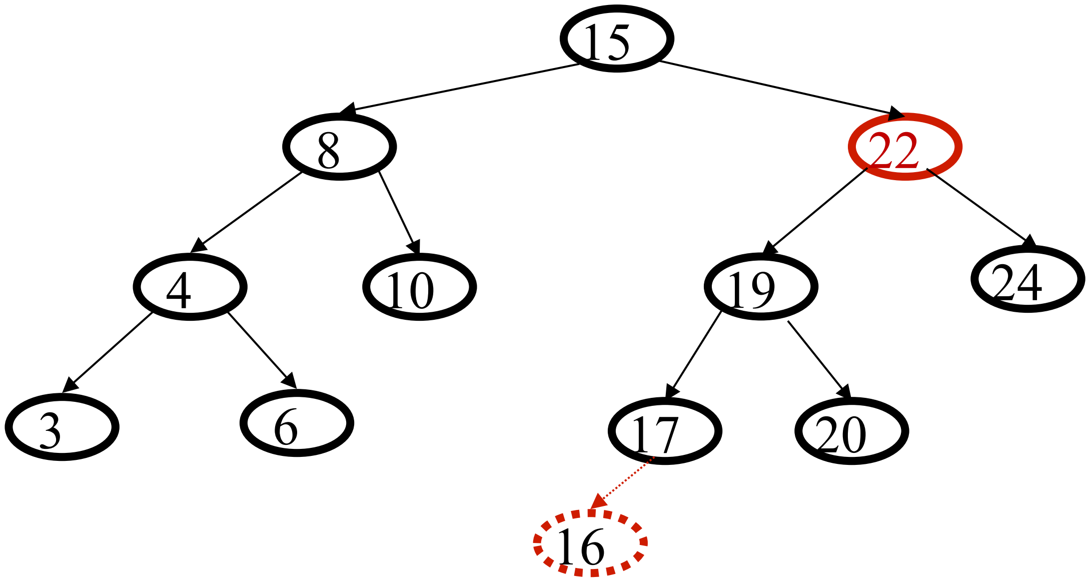


Another example: `insert(16)`



Where is the imbalance?

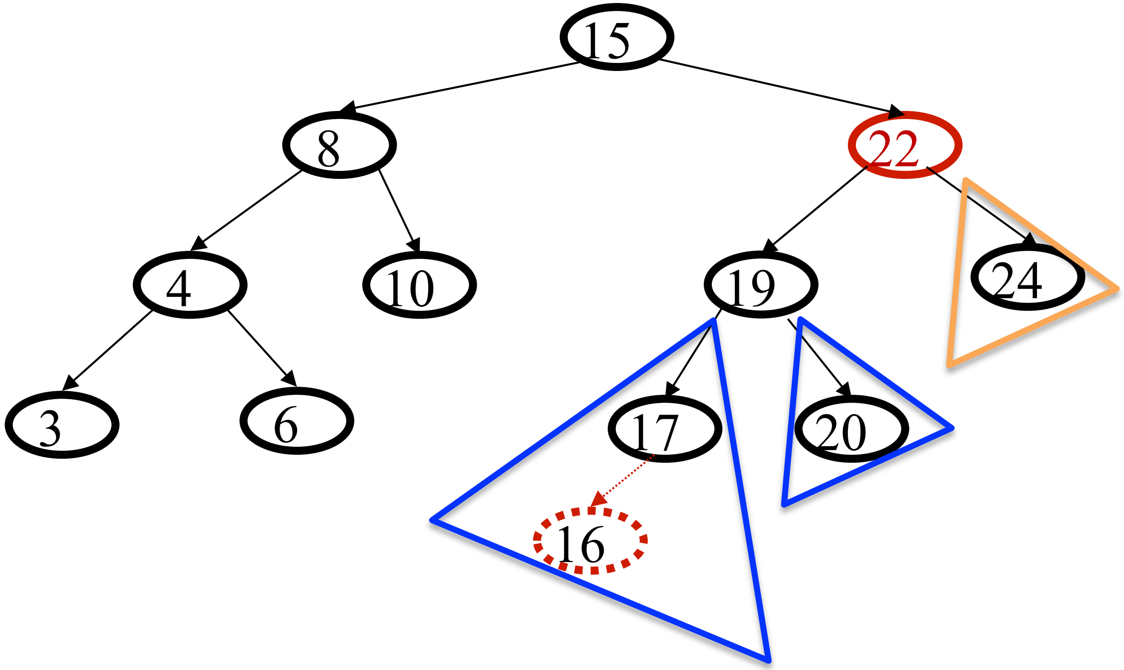
Another example: `insert(16)`



Where is the imbalance?

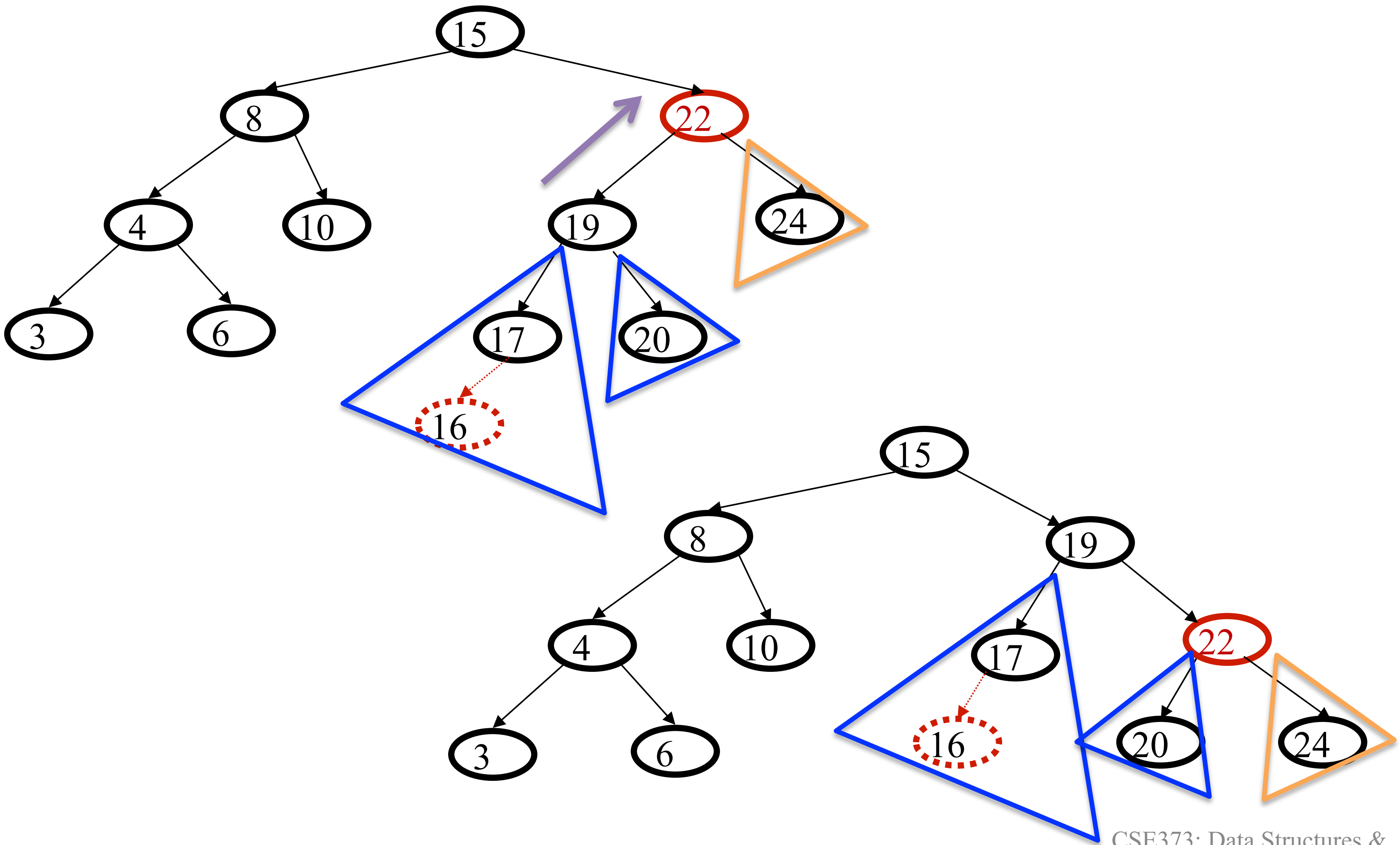
22

Another example: `insert(16)`



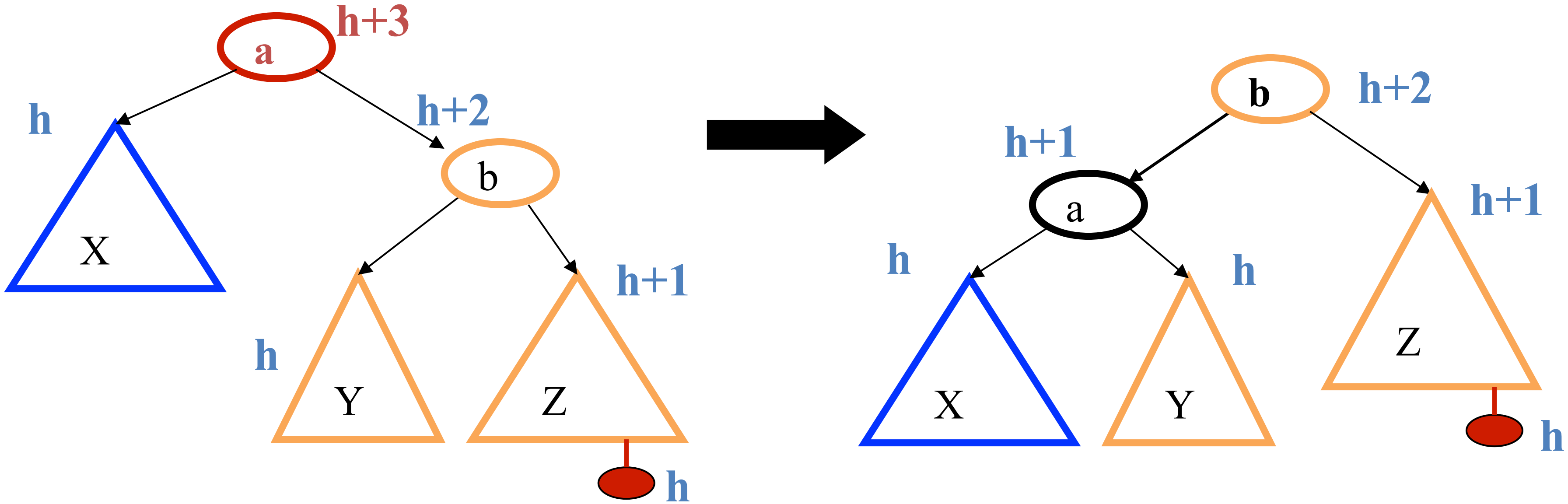
What are the generalized subtrees?

Another example: `insert(16)`



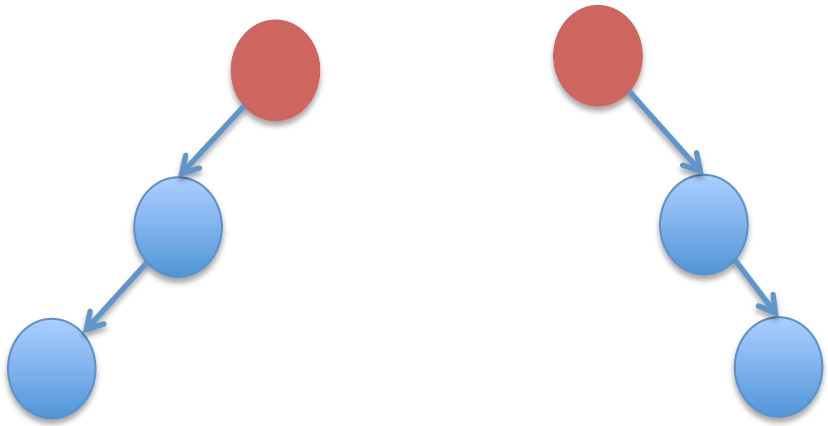
Case #2: right-right generalized

- Mirror image to left-left case, so you rotate the other way
  - Exact same concept, but different code

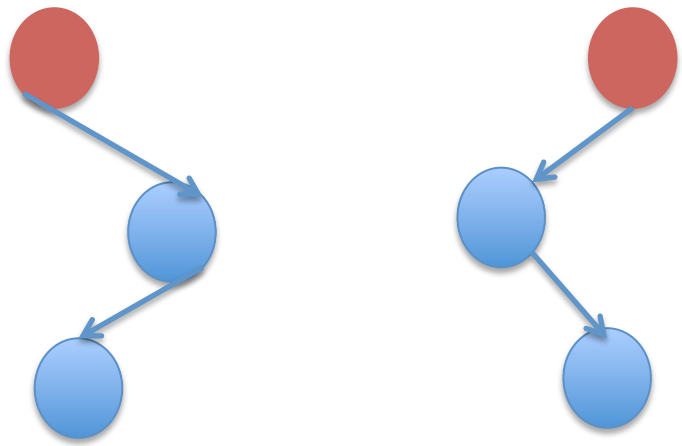


Rotations so far

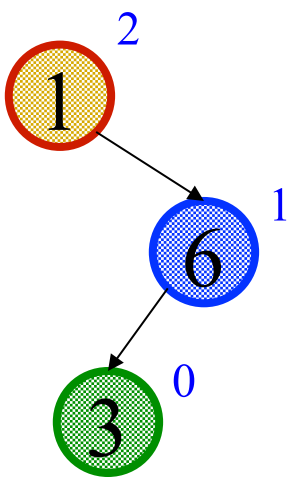
Cases we've covered:



Cases we haven't covered:



Case 3 & 4: left-right and right-left

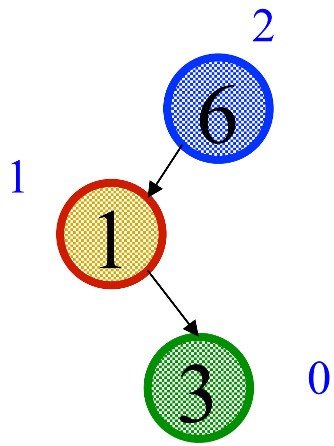


Insert(1)

Insert(6)

Insert(3)

Is there a single rotation that can fix either tree?



Insert(6)

Insert(1)

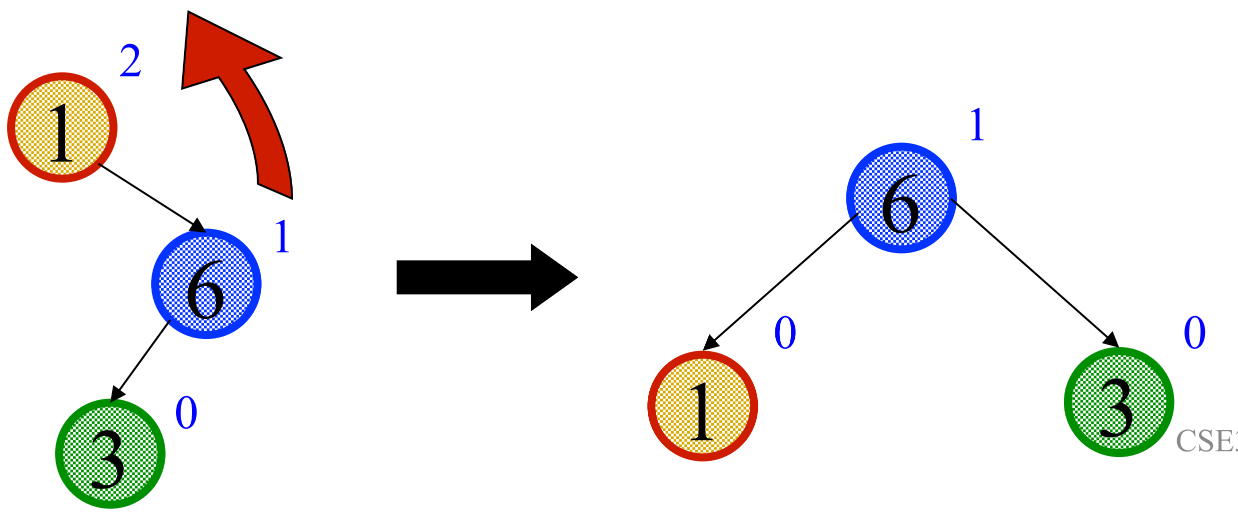
Insert(3)

Wrong rotation #1:

Unfortunately, single rotations are not enough for insertions in the **left-right** subtree or the **right-left** subtree

Simple example: **insert**(1), **insert**(6), **insert**(3)

- First wrong idea: single rotation like we did for left-left

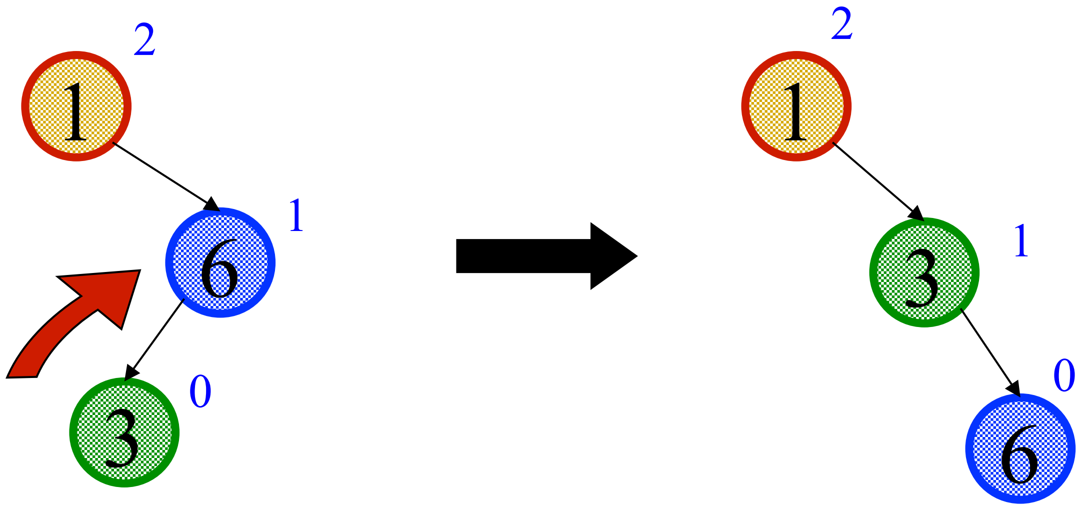


Wrong rotation #2:

Unfortunately, single rotations are not enough for insertions in the **left-right** subtree or the **right-left** subtree

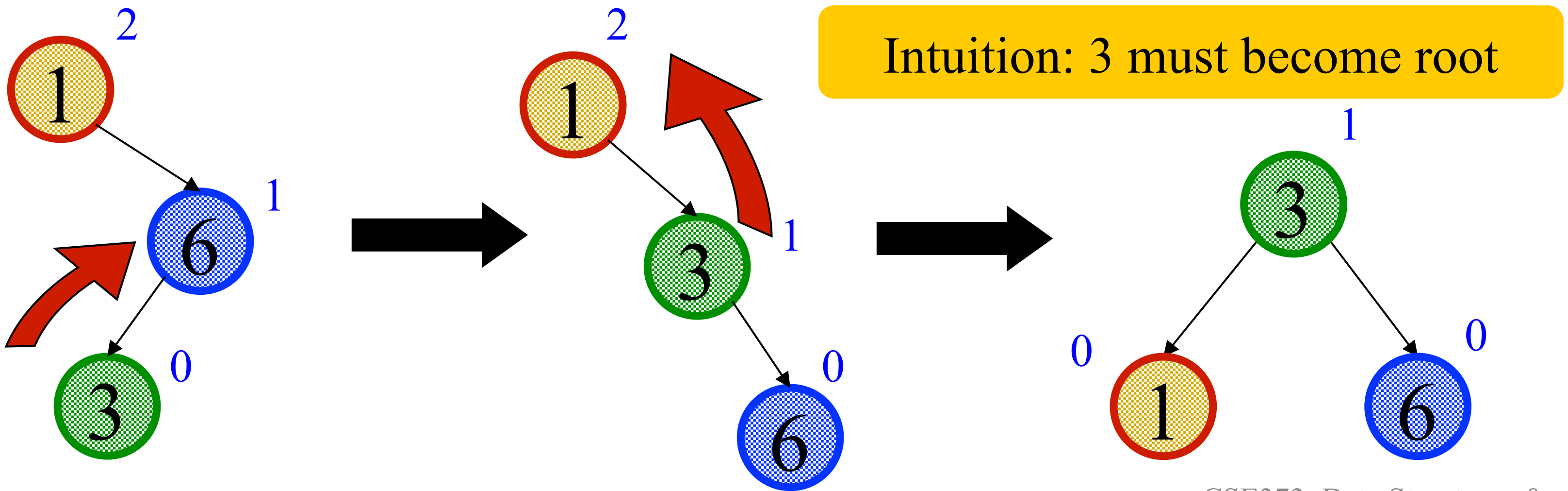
Simple example: **insert**(1), **insert**(6), **insert**(3)

- Second wrong idea: single rotation on the child of the unbalanced node

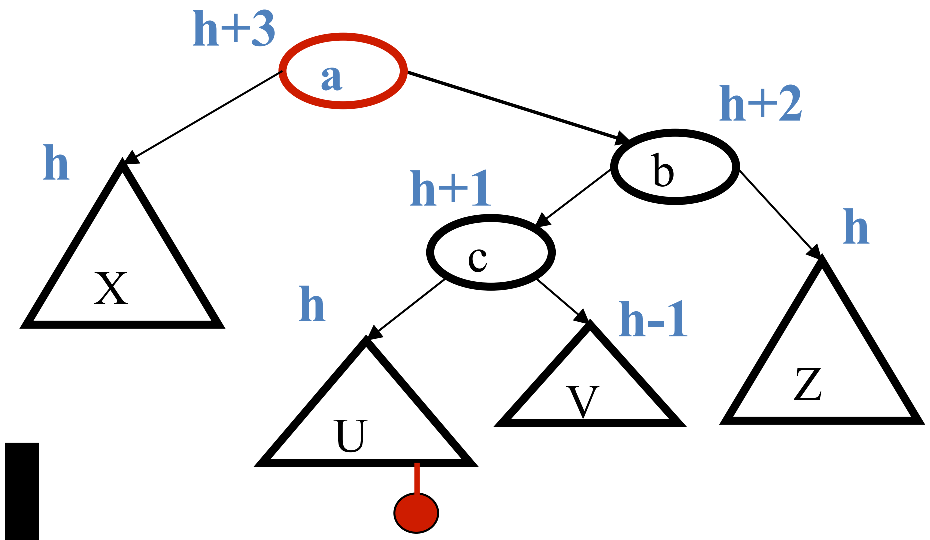


Sometimes two wrongs make a right

- First idea violated the BST property
- Second idea didn't fix balance
- But if we do both single rotations, starting with the second, it works! (And not just for this example.)
- Double rotation:
  1. Rotate problematic child and grandchild
  2. Then rotate between self and new child



The general right-left case

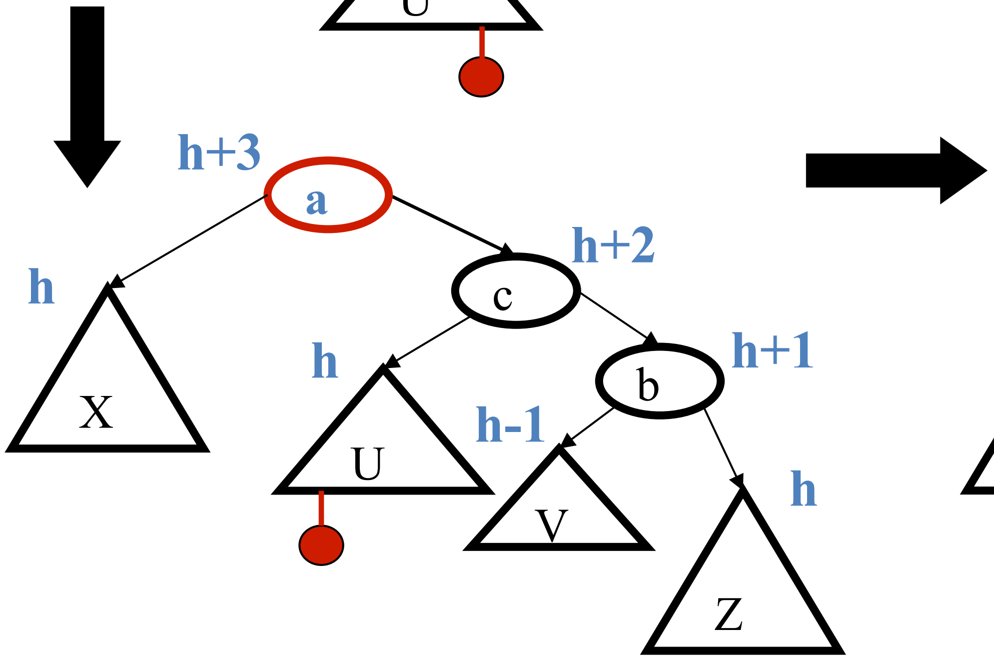


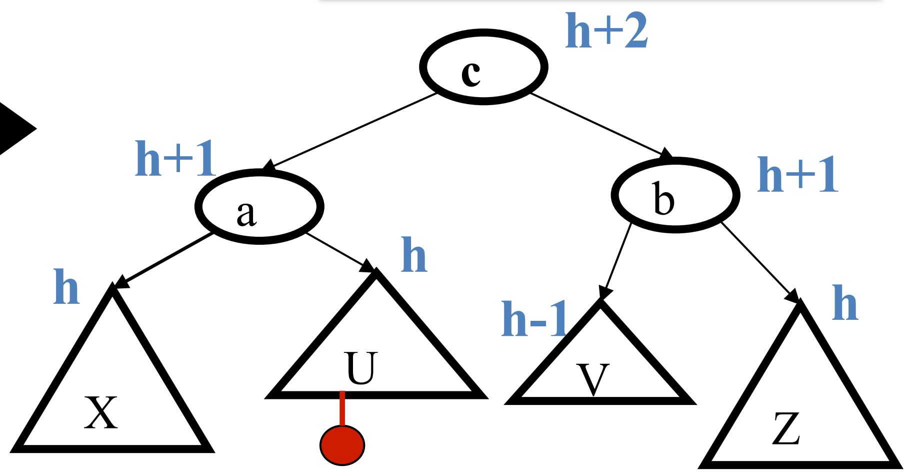
Rotation 1:

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

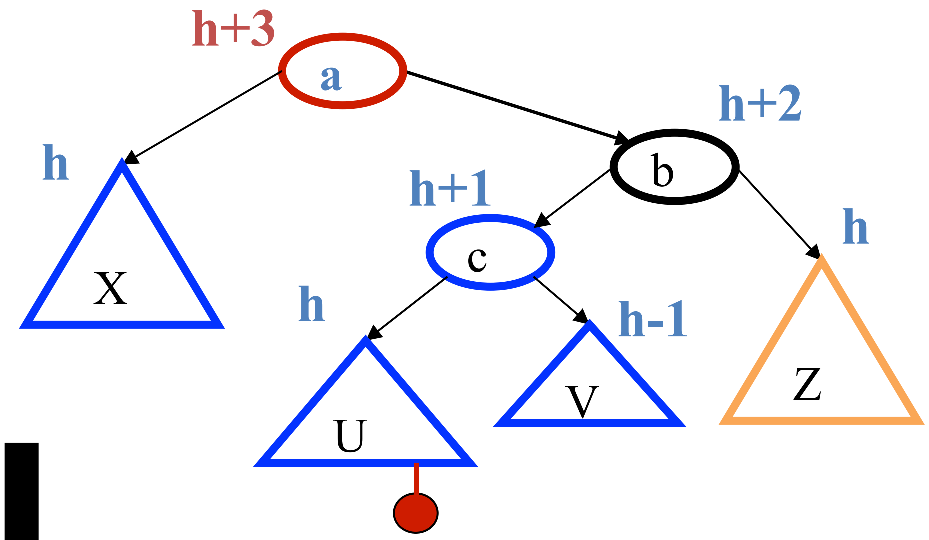
Rotation 2:

```
a.right = c.left
c.left = a
root = c
```



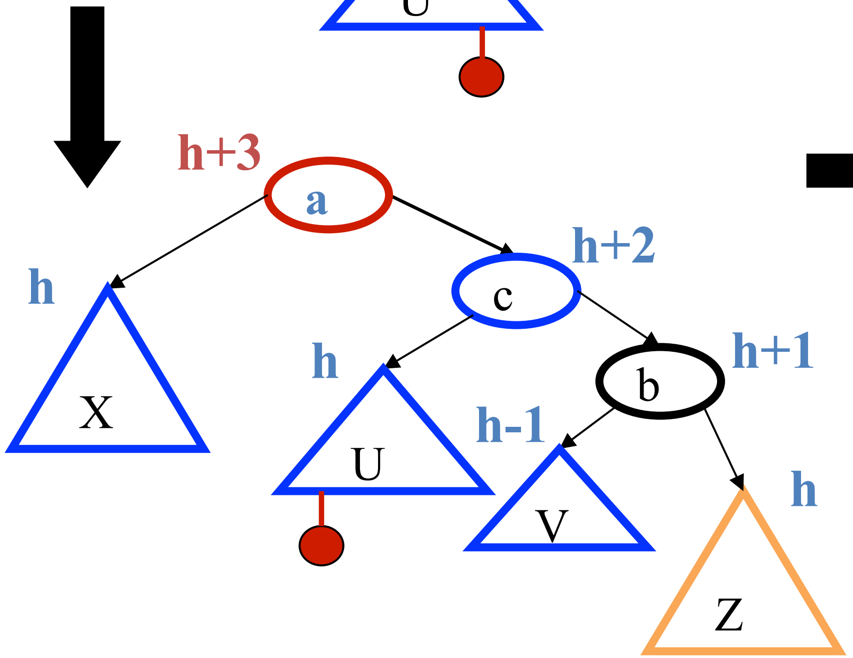


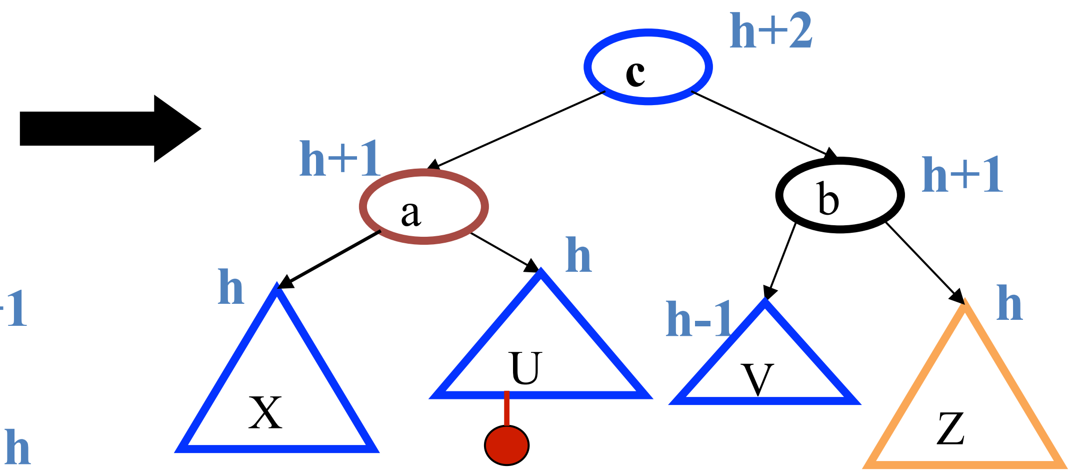
Keeping track of values



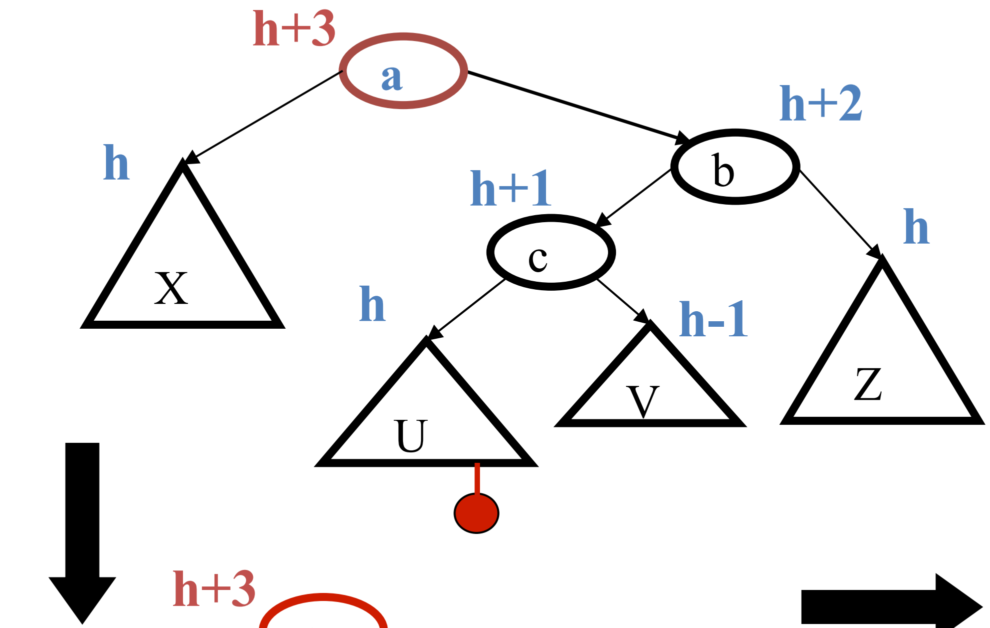
First rotation (around node ‘b’):

- Subtrees < ‘b’ in blue
- Subtrees > ‘b’ in orange



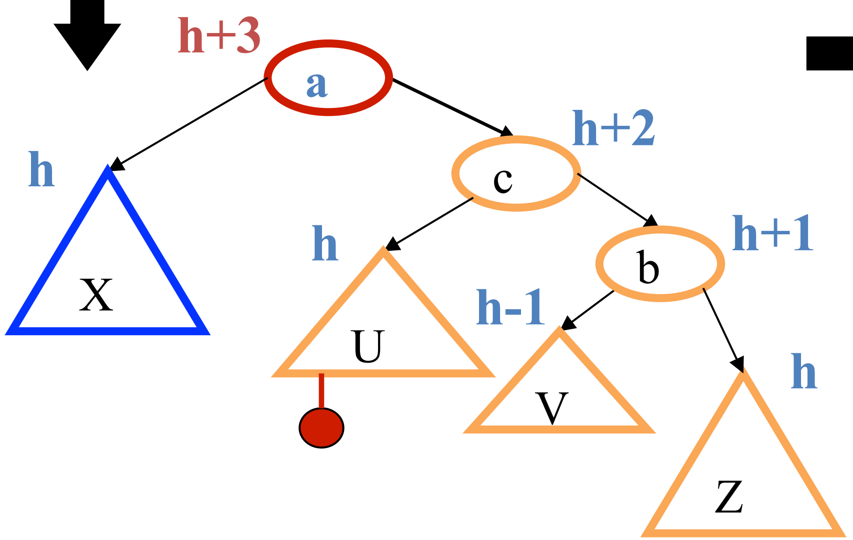


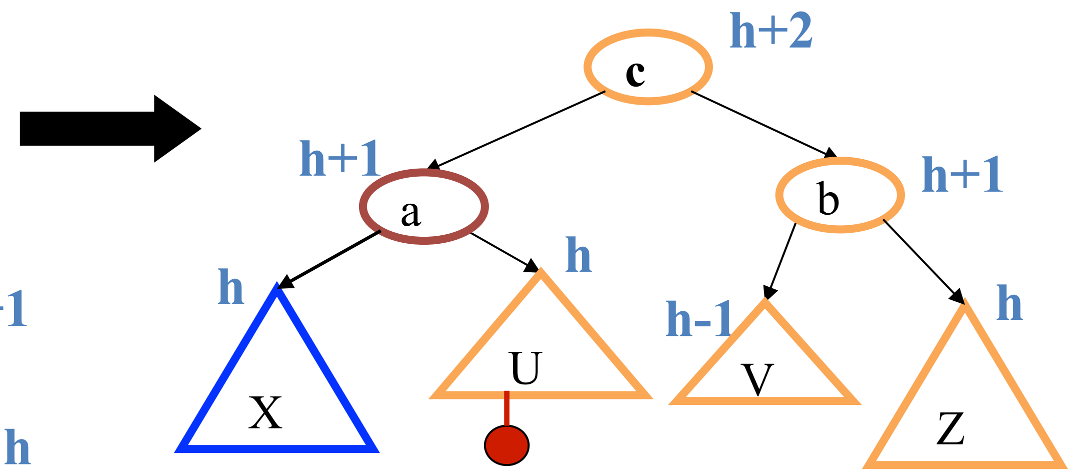
Keeping track of values 2



Second rotation (around node ‘a’):

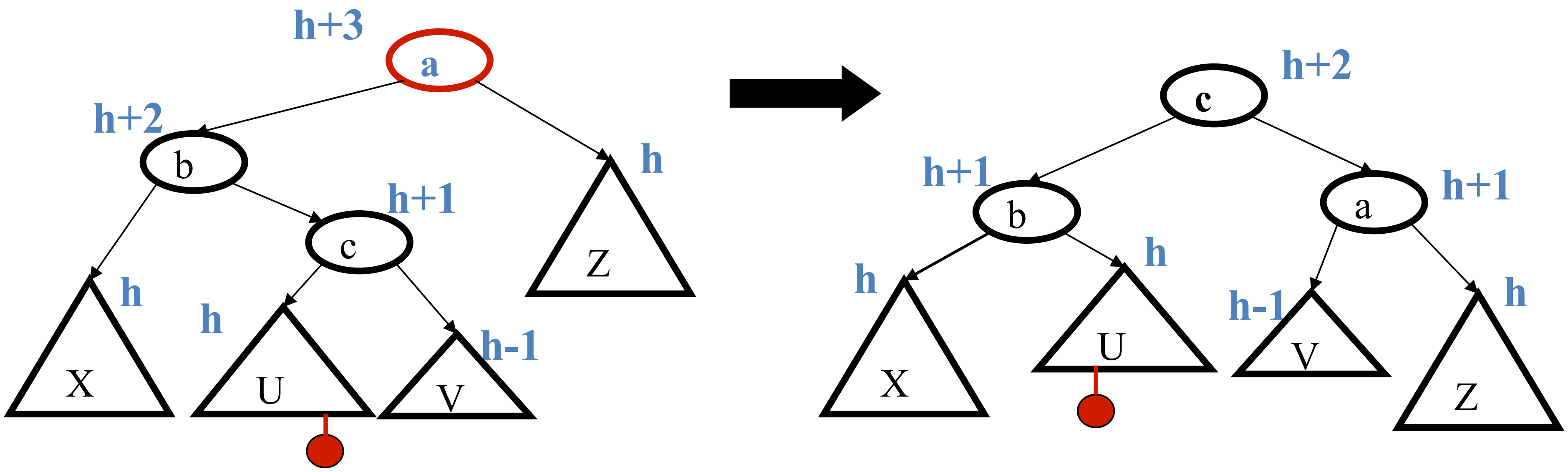
- Subtrees < ‘a’ in blue
- Subtrees > ‘a’ in orange





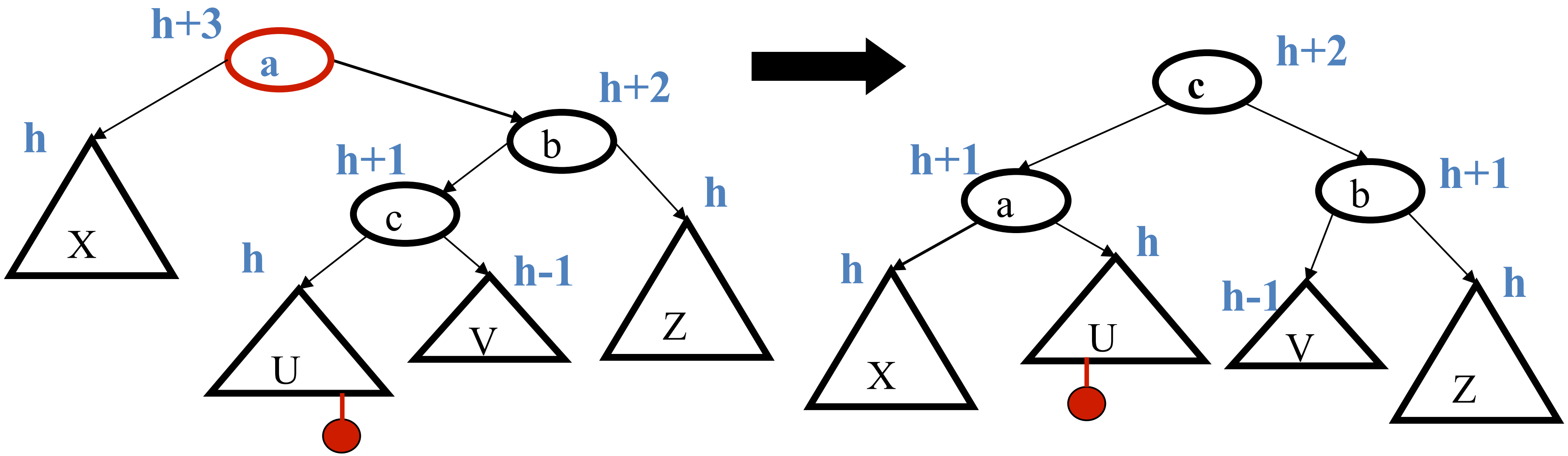
The last case: left-right

- Mirror image of right-left
  - Again, no new concepts, only new code to write



Comments

- Like in the left-left and right-right cases, the height of the subtree after rebalancing is the same as before the insert
  – So no ancestor in the tree will need rebalancing
- Does not have to be implemented as two rotations; can just do:



Easier to remember than you may think:

1) Move c to grandparent's position

2) Put a, b, X, U, V, and Z in the only legal positions for a BST

Insert, summarized

- Insert as in a BST
- Check back up path for imbalance, which will be 1 of 4 cases:
  - Node's left-left grandchild is too tall (**left-left single rotation**)
  - Node's left-right grandchild is too tall (**left-right double rotation**)
  - Node's right-left grandchild is too tall (**right-left double rotation)**
  - Node's right-right grandchild is too tall (**right-right double rotation**)
- Only one case occurs because tree was balanced before insert
- After the appropriate single or double rotation, the smallest-unbalanced subtree has the same height as before the insertion
  - So all ancestors are now balanced

Efficiency

- Worst-case complexity of **`find`**: $O($**`log`** $n)$
  - Tree is balanced
- Worst-case complexity of **`insert`**: $O($**`log`** $n)$
  - Tree starts balanced
  - A rotation is $O(1)$ and there's an $O($**`log`** $n)$ path to root
  - (Same complexity even without one-rotation-is-enough fact)
  - Tree ends balanced
- Worst-case complexity of **`buildTree`**: $O(n$ **`log`** $n)$

Takes some more rotation action to handle **`delete`**…

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced
2. Height balancing adds no more than a constant factor to the speed of insert and delete

Arguments against AVL trees:

1. Difficult to program & debug [but done once in a library!]
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. Most large searches are done in database-like systems on disk and use other structures (e.g., *B*-trees, a data structure in the text)
5. If *amortized* (later, I promise) logarithmic time is enough, use splay trees (also in text)

Today's Takeaways

- AVL trees:
  - understand the AVL balance condition
  - be able to identify AVL trees
  - intuition on why the height is O(logN)
  - understand AVL inserts and rotations:
    - single rotations
    - double rotations
  - understand complexity of AVL operations