# CSE 373: Data Structures & Algorithms

# Wrap up Amortized Analysis; AVL Trees

Riley Porter

Winter 2017

## Course Logistics

- Symposium offered by CSE department today
- HW2 released, Big-O, Heaps (lecture slides have pseudocode that will help a lot)
- Weekly summaries out soon

## Review: Amortized Complexity

- For a Stack implemented with an array can we claim **push** is *O*(**1**) time if resizing is *O*(**n**) time?
  - We *can't*, but we *can* claim it's an *O*(**1**) amortized operation

- Why is this good? Why do we care?
  - If amortized is good enough for our problem, then it's great to be able to say O(1) instead of O(n)

## Review: Amortized Complexity

- Like an Average.

N inserts at 1 unit cost + 1 insert at N unit cost

= N * 1 + 1 * N

2N overall cost for N + 1 insertions

= 2N cost / (N + 1) insertions

= O( 2N/(N+1) )

= O(1) amortized cost

# Not-Amortized Complexity

What if we only add 3 slots instead of N slots?

3 inserts at 1 unit cost + 1 insert at N unit cost

= 3 * 1 + 1 * N

= N + 3 overall cost for N + 1 insertions

= N + 3 cost / (3 + 1) insertions

= O( (N+3)/(3+1) )

= O(N)

Not Amortized! Didn't build up enough "credit"

## Example #1: Resizing stack

A stack implemented with an array where we double the size of the array if it becomes full

Claim: Any sequence of **push**/**pop**/**isEmpty** is amortized *O*(**1**)

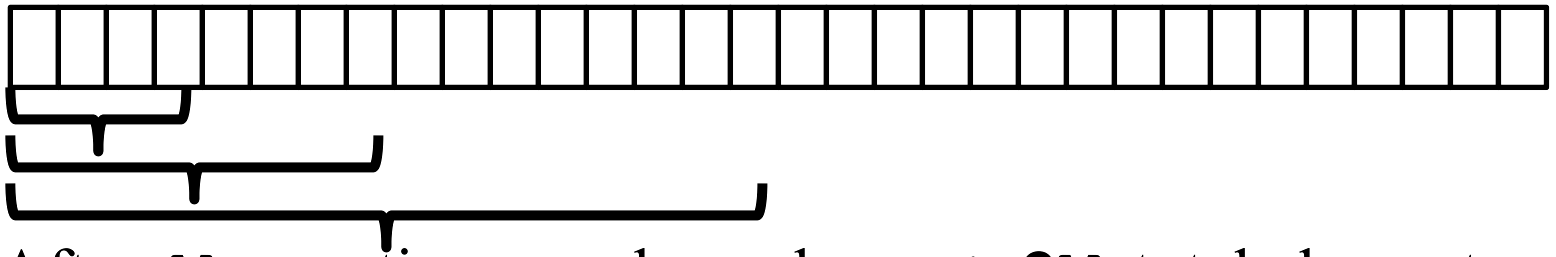
Need to show any sequence of **M** operations takes time *O*(**M**)

- Recall the non-resizing work is *O*(**M**) (i.e., **M****O*(**1**))
- The resizing work is proportional to the total number of element copies we do for the resizing
- So it suffices to show that:

  After **M** operations, we have done **< 2M** total element copies

  (So average number of copies per operation is bounded by a constant)

# Amount of copying



After **`M`** operations, we have done **`< 2M`** total element copies

Let **`n`** be the size of the array after **`M`** operations

- Then we have done a total of:

  **`n/2 + n/4 + n/8 + … INITIAL_SIZE < n`**

  element copies
- Because we must have done at least enough **`push`** operations to cause resizing up to size **`n`**:

  **`M ≥ n/2`**
- So

  **`2M ≥ n >`** *number of element copies*

## Have to be careful

- If array grows by a constant amount (say 1000), operations are not amortized $O(\mathbf{1})$
  - After $O(\mathbf{M})$ operations, you may have done $\Theta(\mathbf{M}^2)$ copies
- If array doubles when full and shrinks when 1/2 empty, operations are not amortized $O(\mathbf{1})$
  - Terrible case: **pop** once and shrink, **push** once and grow, **pop** once and shrink, …
- If array doubles when full and shrinks when 3/4 empty, it is amortized $O(\mathbf{1})$
  - Proof is more complicated, but basic idea remains: by the time an expensive operation occurs, many cheap ones occurred

## Amortized Complexity Summary

- Like an average
- You're building up "credit" with N cheap tasks proportional to 1 expensive N task
- Sometimes hard to prove, but useful if your application doesn't require every single operation to be cheap.

*There's another example on slides from Wednesday involving Queues, if you're curious*

## Review: Balanced BST

*Observation*

- BST: the shallower the better!
- For a BST with *n* nodes inserted in arbitrary order
  - Average height is $O(\log n)$ – see text for proof
  - Worst case height is $O(n)$
- Simple cases, such as inserting in key order, lead to the worst-case scenario

*Solution*: Require a **Balance Condition** that

1. Ensures depth is always $O(\log n)$ – strong enough!
2. Is efficient to maintain – not too strong!

## The AVL Balance Condition

Left and right subtrees of *every node* have *heights* **differing by at most 1**

*Definition*: **balance**(*node*) = height(*node*.left) – height(*node*.right)

AVL *property*: **for every node *x*, –1 ≤ balance(*x*) ≤ 1**

- Ensures small depth
  - Will prove this by showing that an AVL tree of height *h* must have a number of nodes *exponential* in *h*
- Efficient to maintain using single and double rotations

# The AVL Tree Data Structure

*Structural properties*

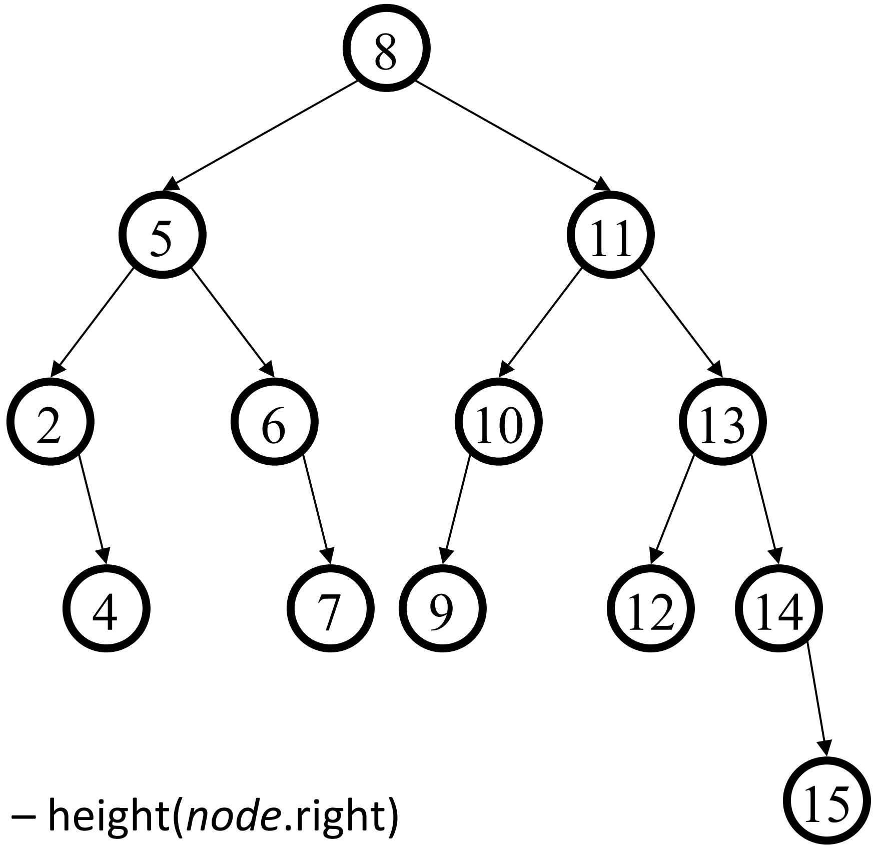
1. Binary tree property
2. Balance property:
   balance of every node is between -1 and 1

Result:

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

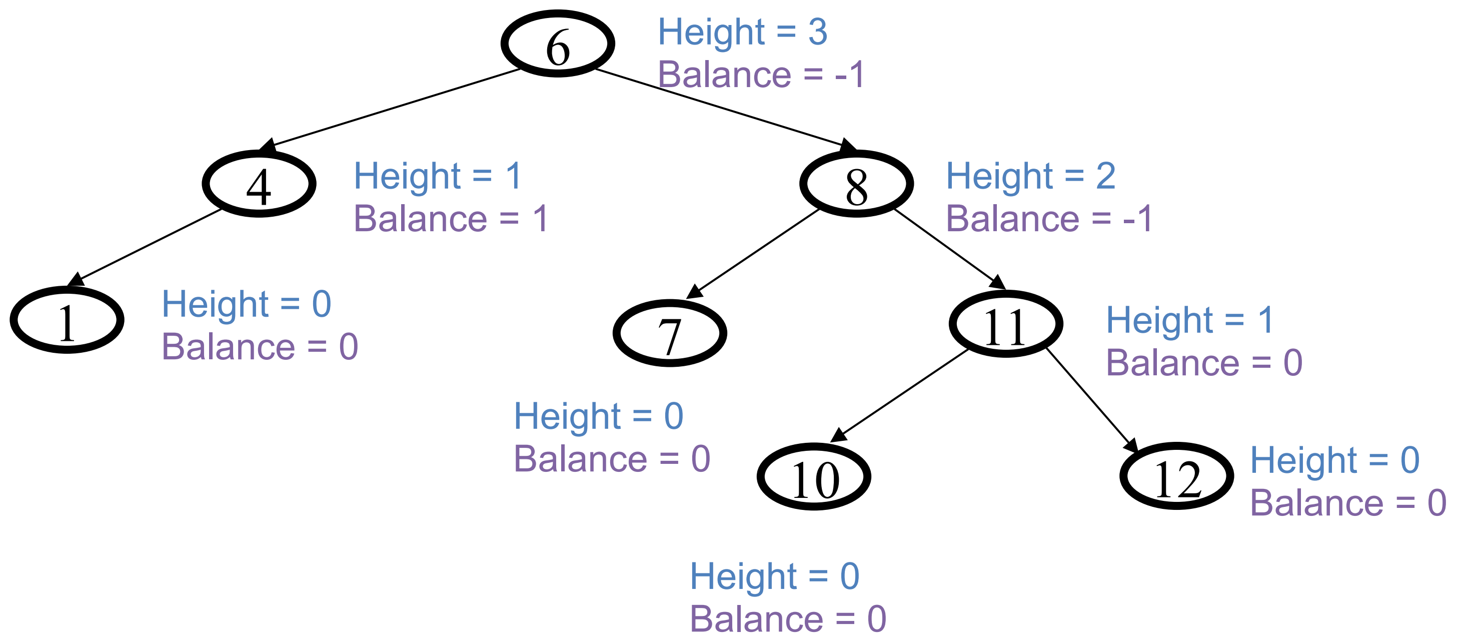
*Ordering property*

– Same as for BST

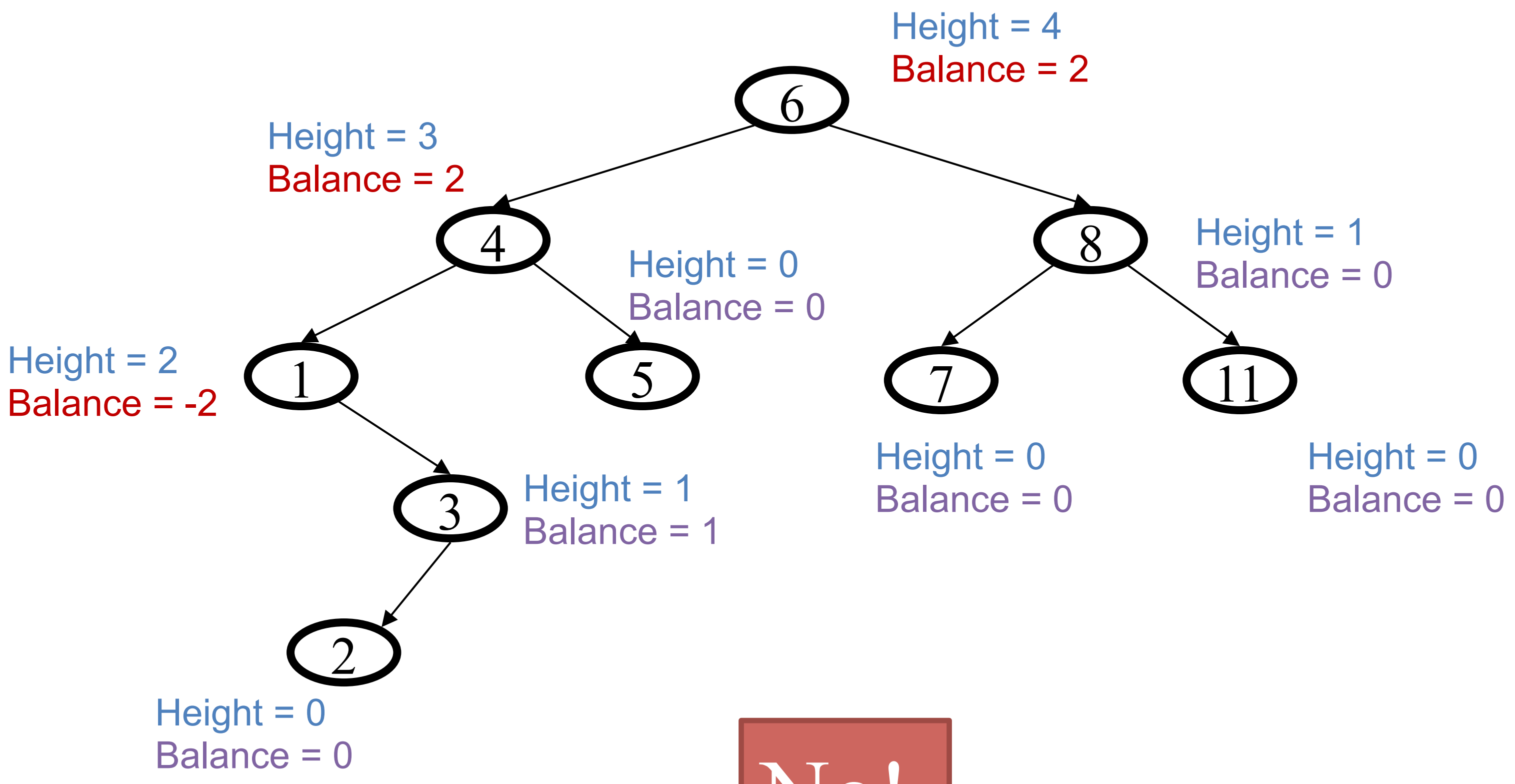


*Definition*: **balance**(*node*) = height(*node*.left) – height(*node*.right)

# An AVL tree?


6
Height = 3
Balance = -1
4
Height = 1
Balance = 1
8
Height = 2
Balance = -1
1
Height = 0
Balance = 0
7
Height = 0
Balance = 0
11
Height = 1
Balance = 0
10
Height = 0
Balance = 0
12
Height = 0
Balance = 0

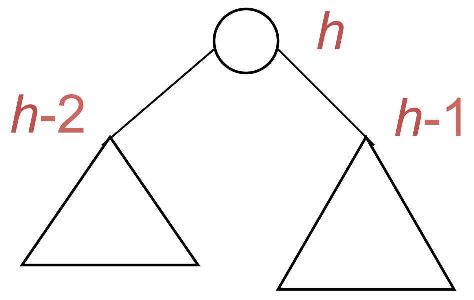
# An AVL tree?



No!

## Intuition: compactness

- If the heights differ by at most 1, your two subtrees are roughly the same size
- If this is true at **every** node, it's true all the way down
- If this is true all the way down, your tree winds up compact.
- Height is O(logN)



*We'll revisit the formal proof of this soon*

# AVL Operations

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

But as we insert and delete elements, we need to:

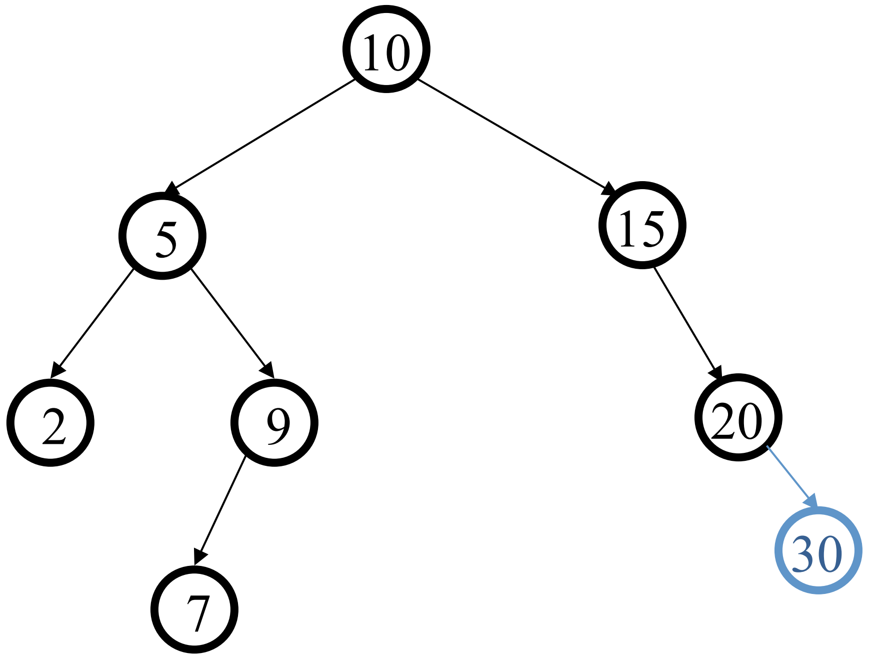
1. **Track balance**
2. **Detect imbalance**
3. **Restore balance**

Is this AVL tree balanced?

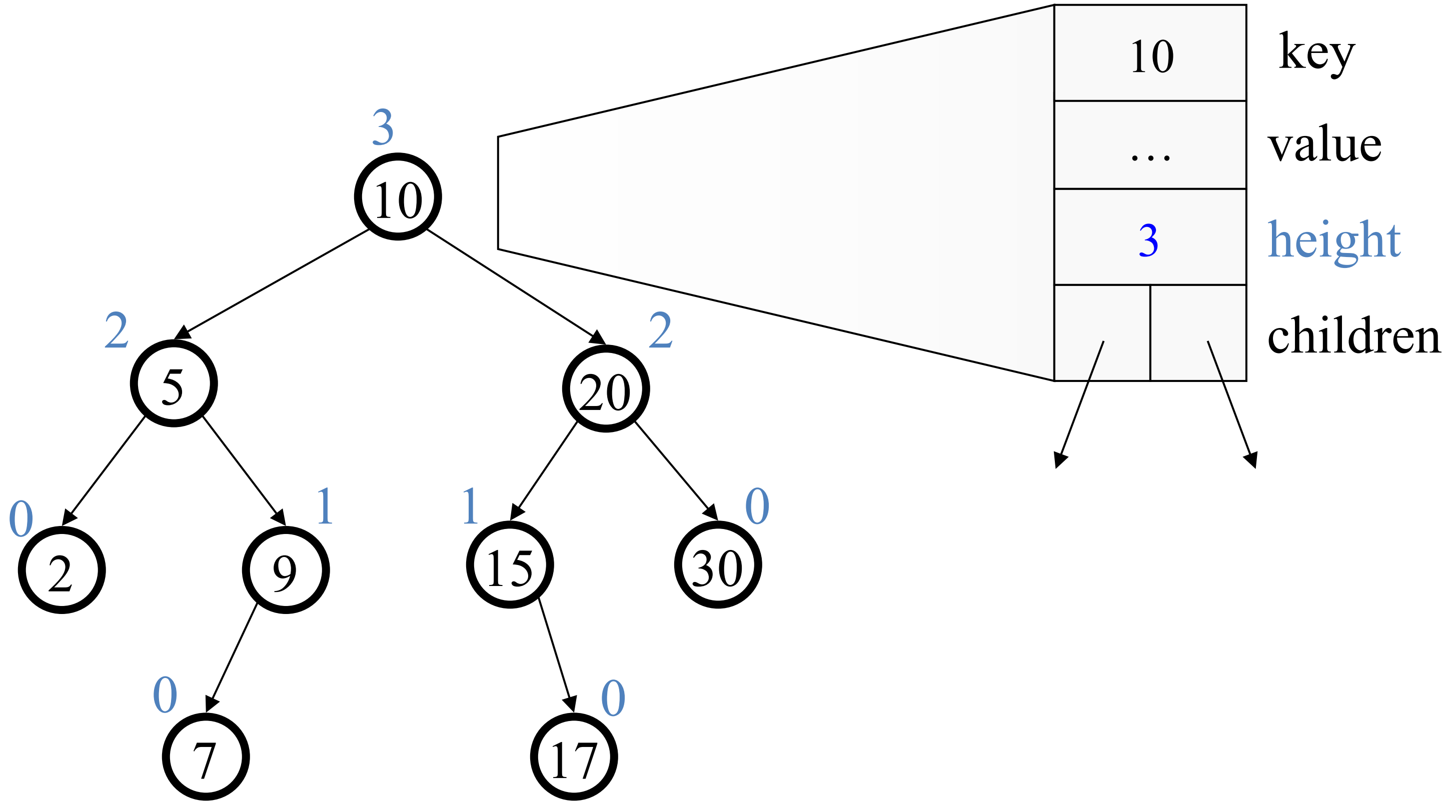
Yep!

How about after `insert(30)`?

No, now the Balance of 15 is off



# Keep the tree balanced



Track height at all times!

## AVL tree Operations

- **AVL `find`**:
  - Same as BST **`find`**
- **AVL `insert`**:
  - First BST **`insert`**, *then* check balance and potentially "fix" the AVL tree
  - Four different imbalance cases
- **AVL `delete`**:
  - The "easy way" is lazy deletion
  - Otherwise, do the deletion and then have several imbalance cases

# Insert: detect potential imbalance

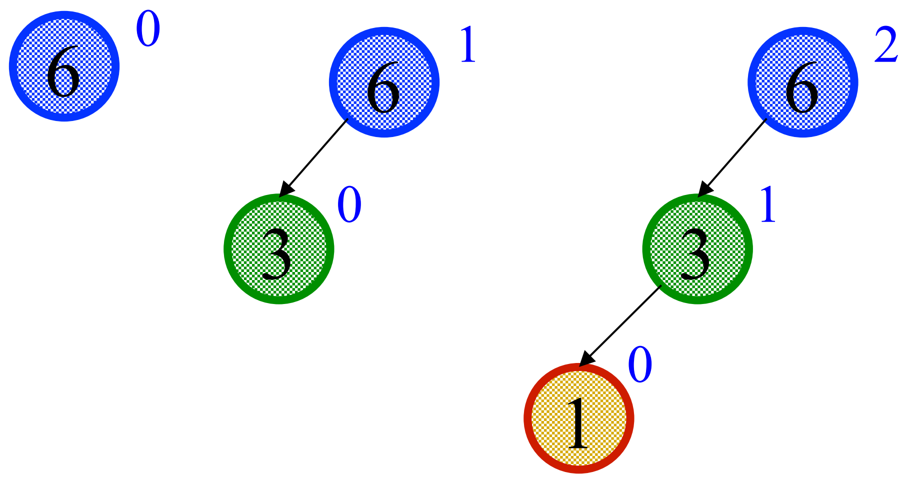
1. Insert the new node as in a BST (a new leaf)
2. For each node on the path from the root to the new leaf, the insertion may (or may not) have changed the node's height
3. So after recursive insertion in a subtree, detect height imbalance and perform a ***rotation*** to restore balance at that node

Type of rotation will depend on the location of the imbalance (if any)

**Facts about insert imbalances:**

- If there's an imbalance, there must be a deepest element that is imbalanced after the insert
- After rebalancing this deepest node, every node is balanced
- So at most one node needs to be rebalanced

## Case #1: Example



Insert(6)

Insert(3)

Insert(1)

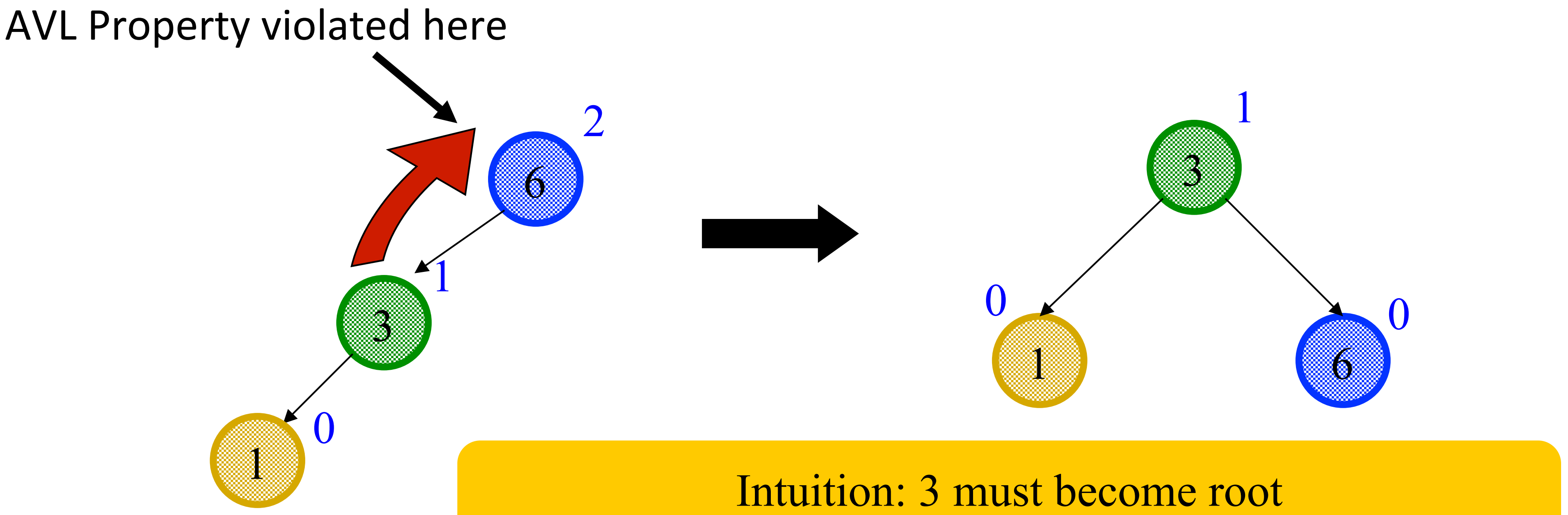
Third insertion violates balance property

- happens to be at the root

What is the only way to fix this (the only valid AVL tree with these nodes?

# 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)

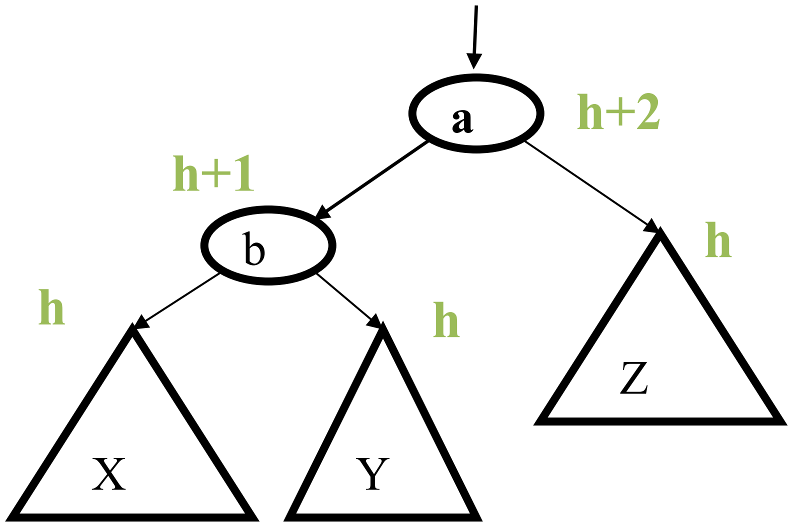


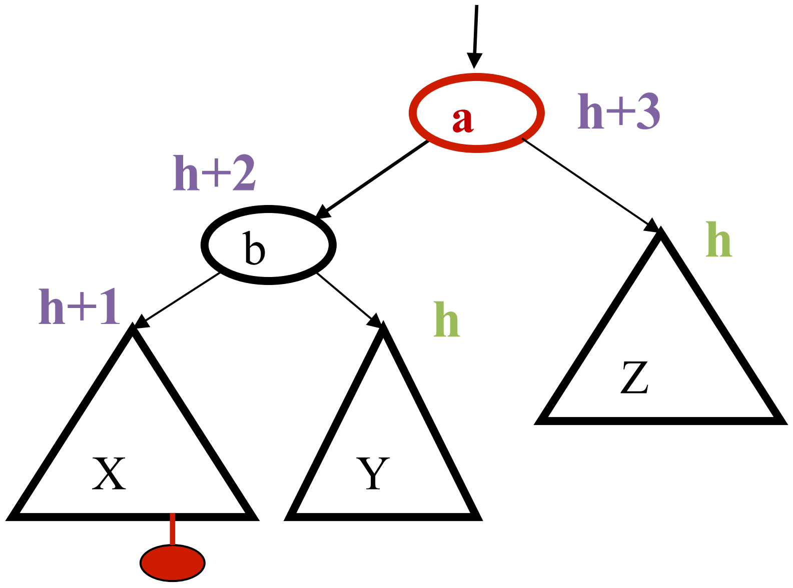
Intuition: 3 must become root

New parent height is now the old parent's height before insert

## The example generalized

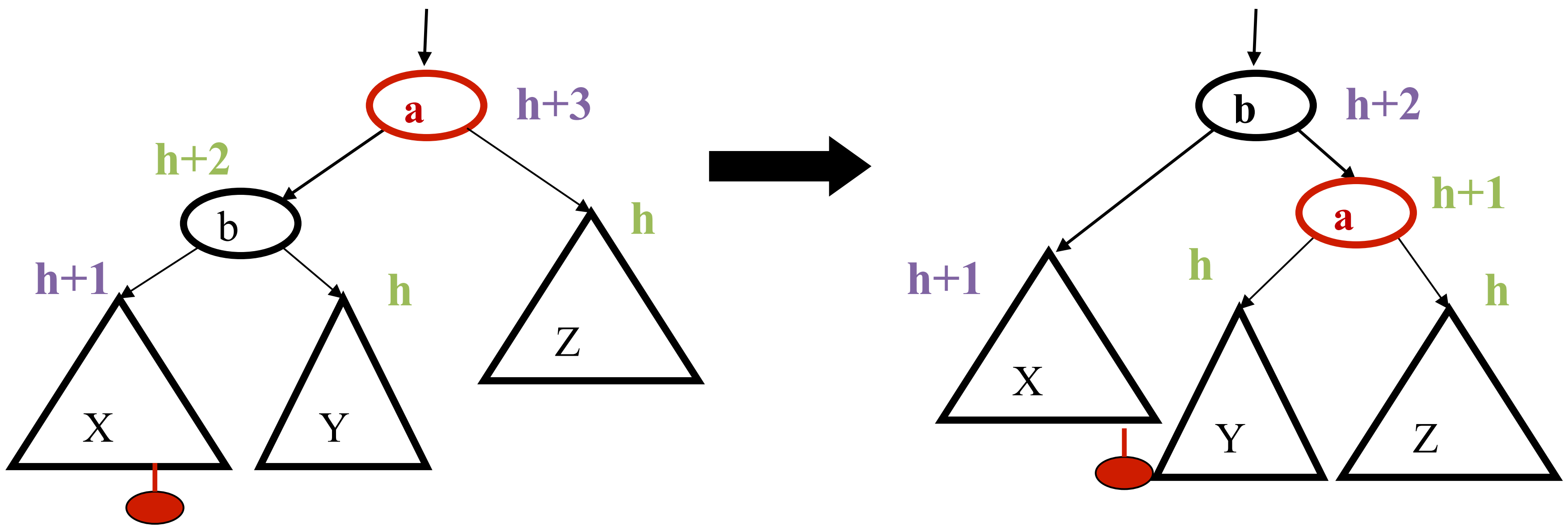
- Node imbalanced due to insertion *somewhere* in **left-left grandchild** that causes an increasing height
  - 1 of 4 possible imbalance causes (other three coming)
- First we did the insertion, which would make ***a*** imbalanced





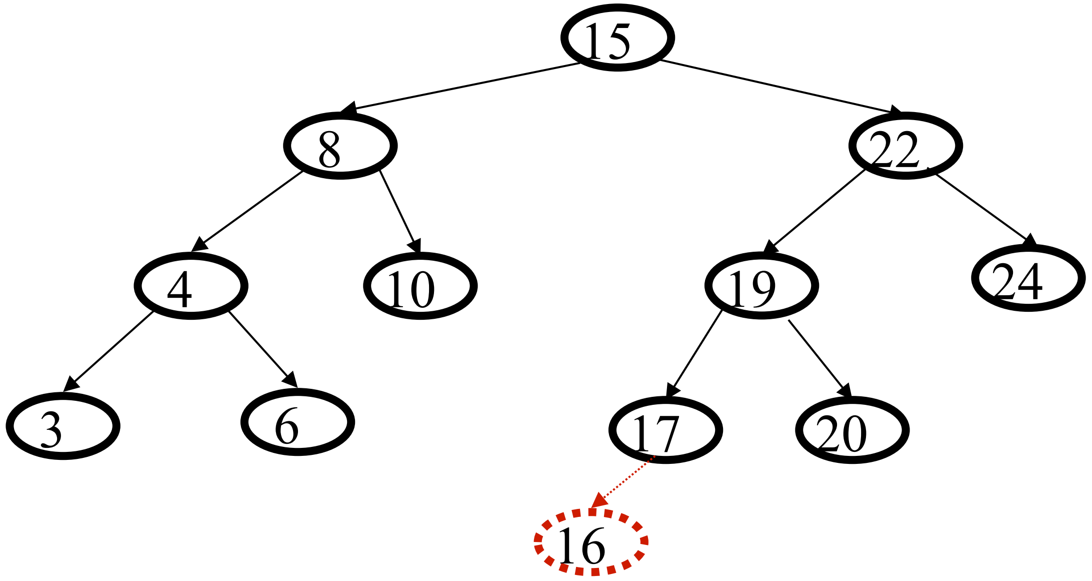
# The general left-left case

- 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



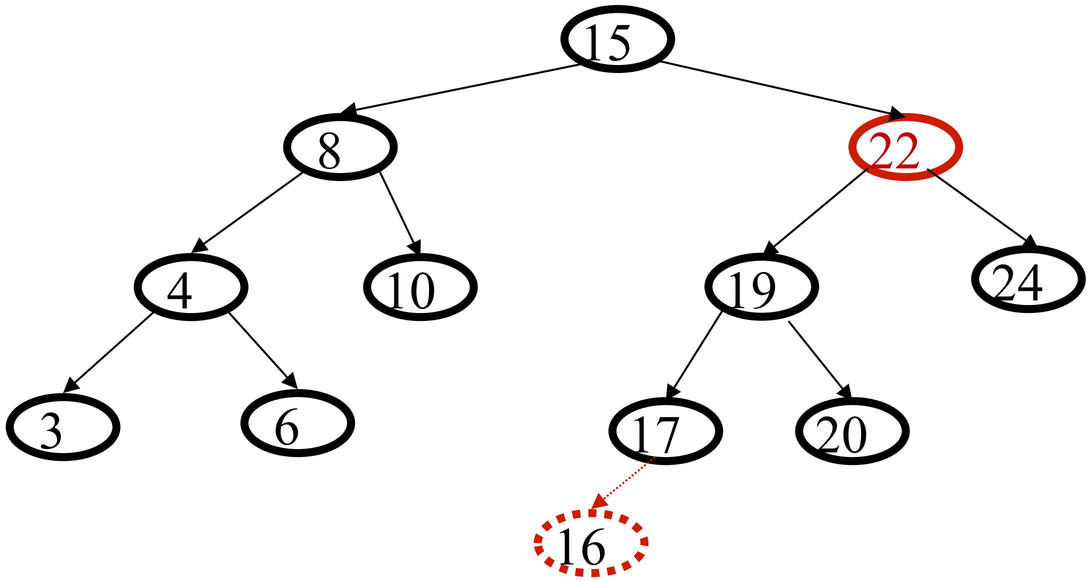
- A single rotation restores balance at the node
  - To same height as before insertion, so ancestors now balanced

## Another example: `insert(16)`



Where is the imbalance?

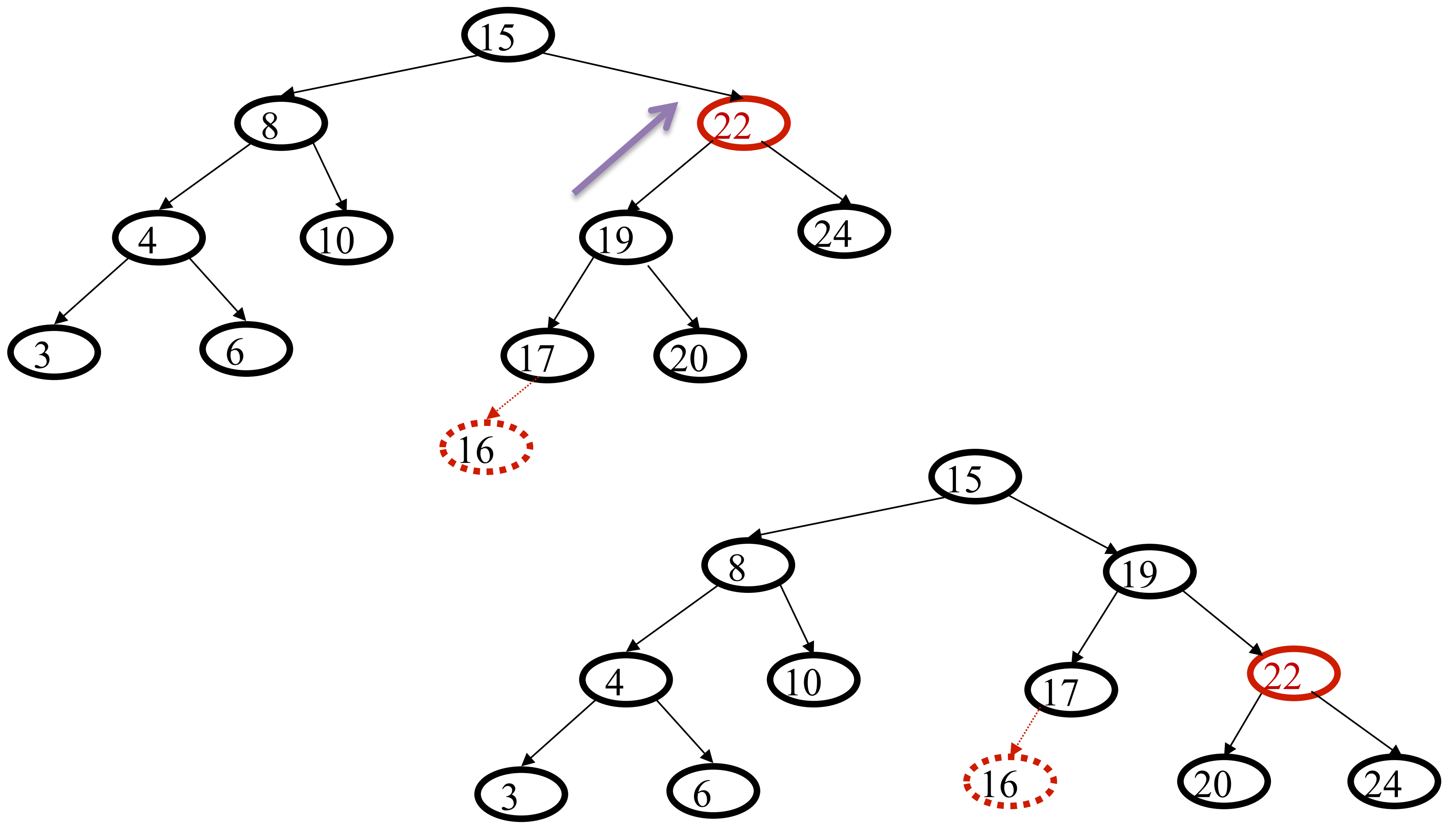
## Another example: `insert(16)`



Where is the imbalance?

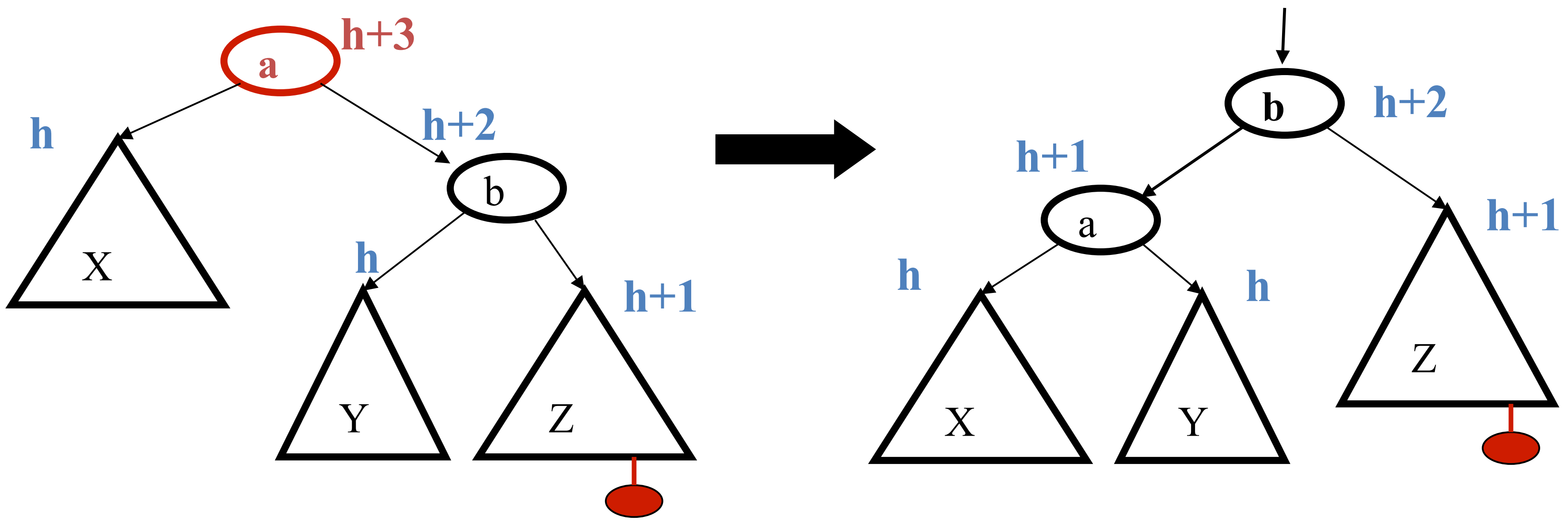
22

# Another example: `insert(16)`

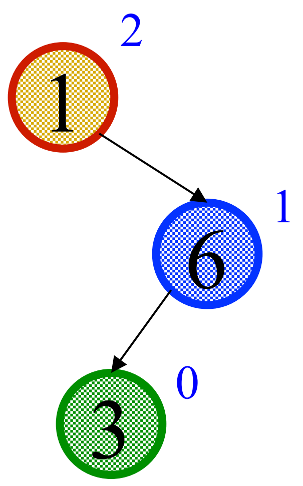


## The general right-right case

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



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

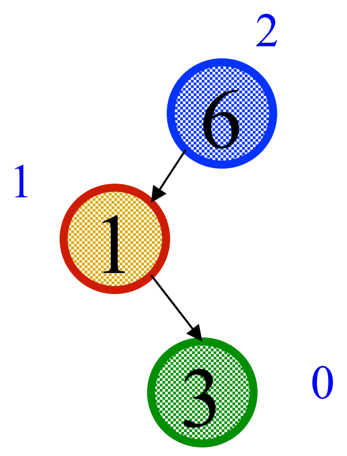


Insert(1)

Insert(6)

Insert(3)

Is there a single rotation that can fix either tree?



Insert(6)

Insert(1)

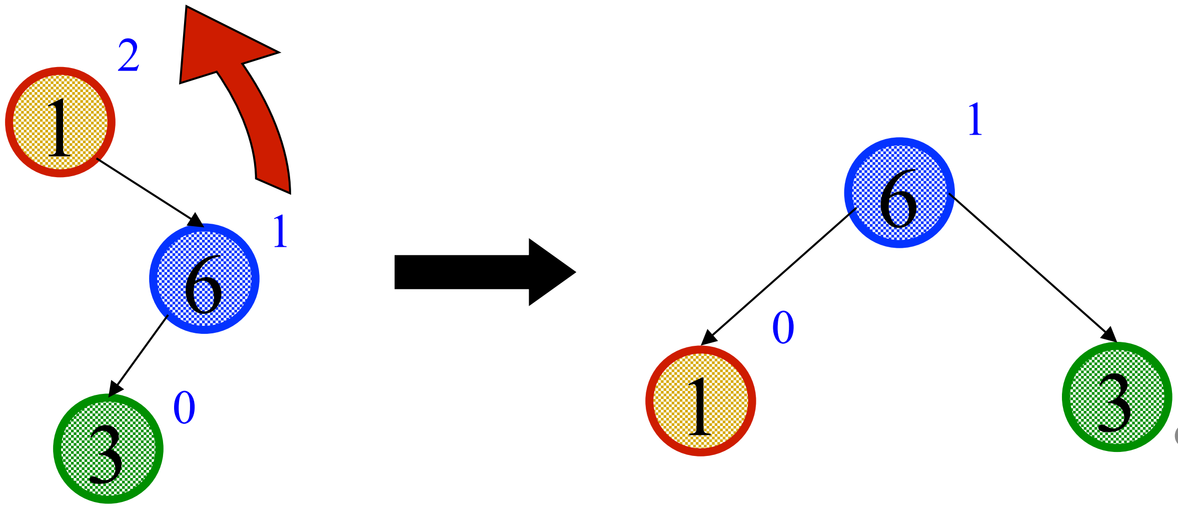
Insert(3)

# 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

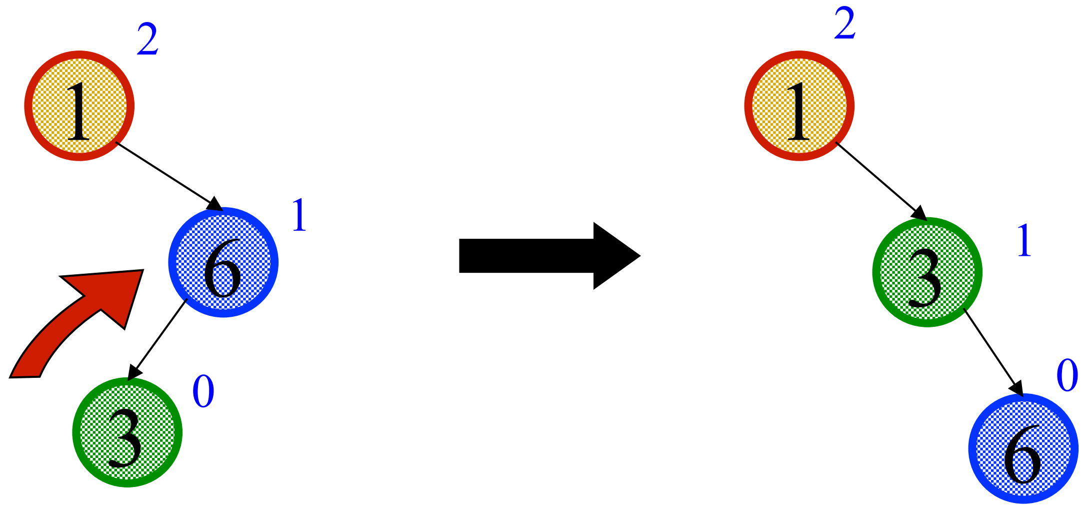


## Wrong rotation #2:

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

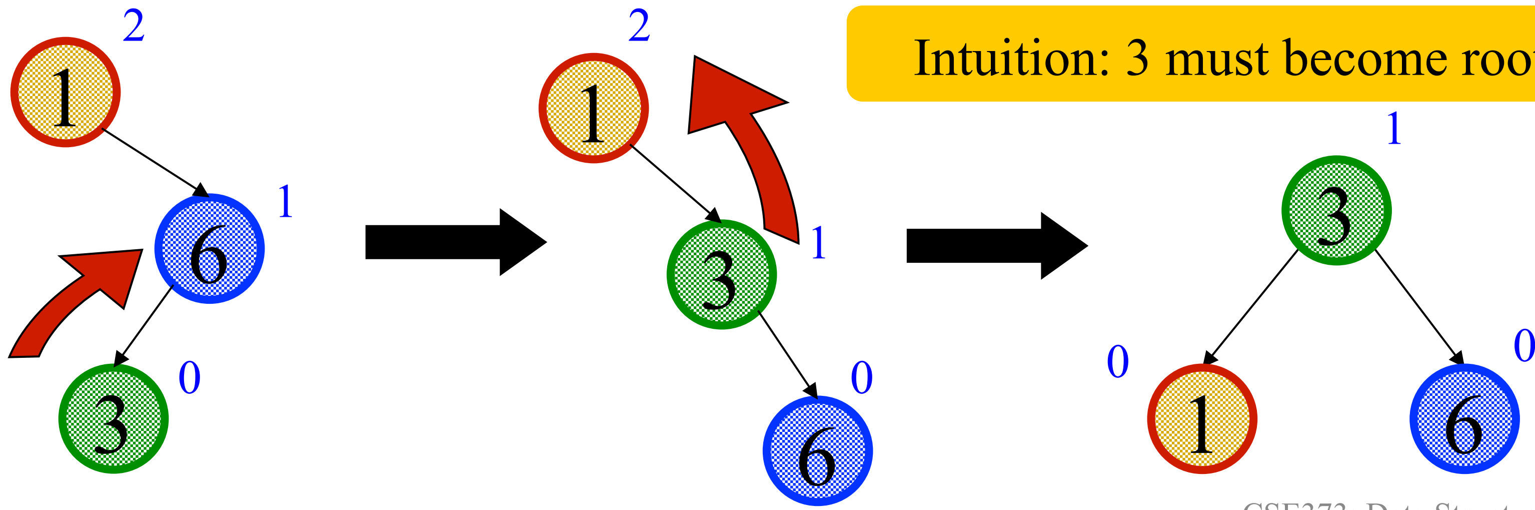
Simple example: **insert**(1), **insert**(6), **insert**(3)

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



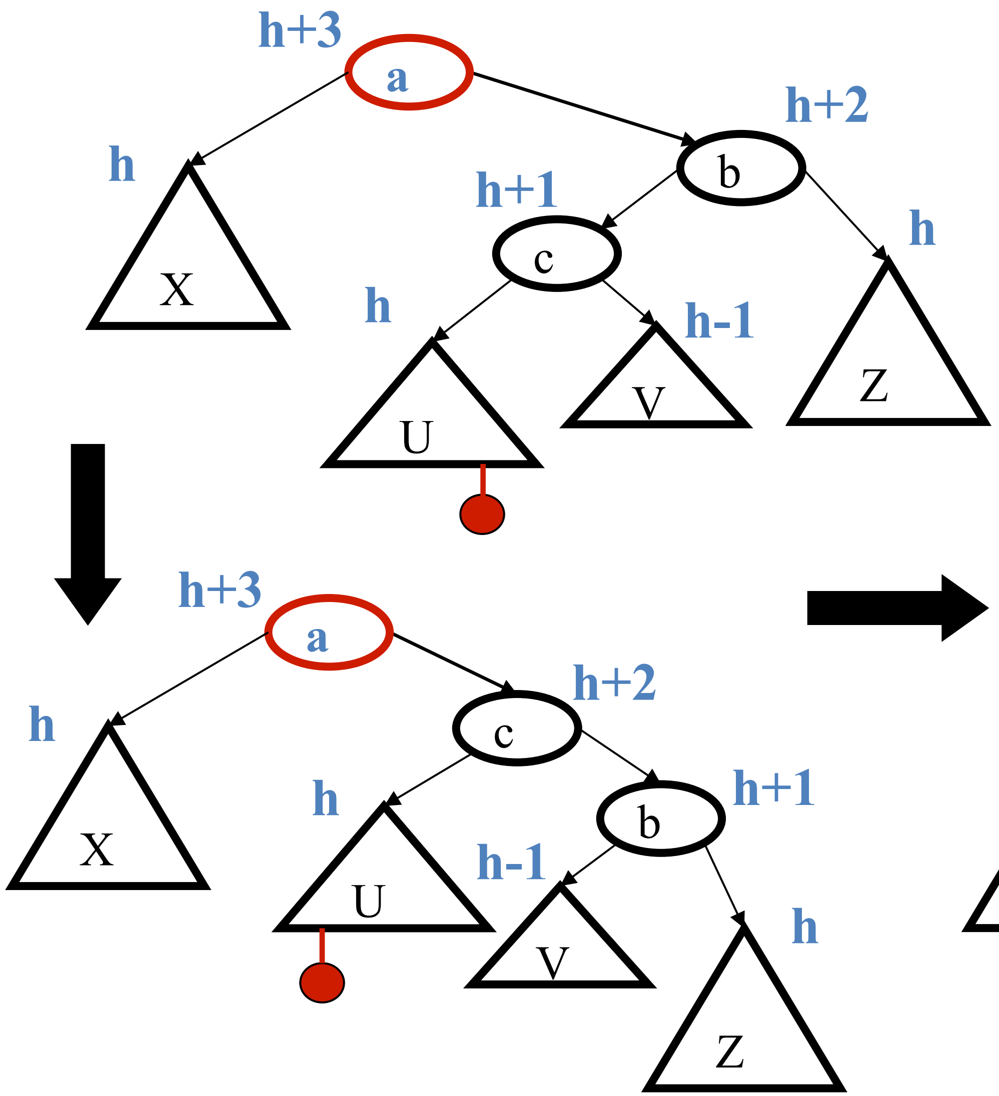
# Sometimes two wrongs make a right

- 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



Intuition: 3 must become root

# The general right-left case

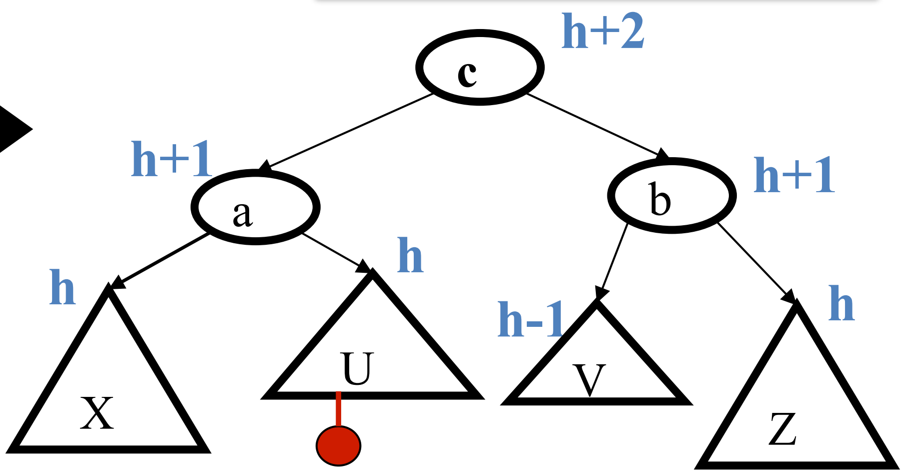


Rotation 1:

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

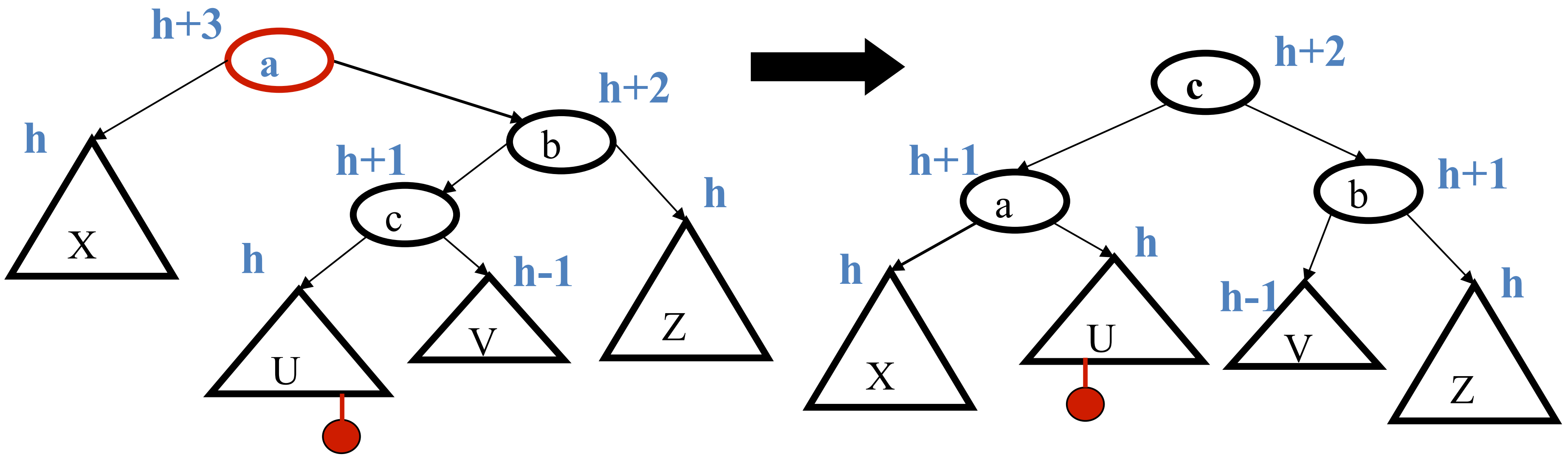
Rotation 2:

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



# 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:



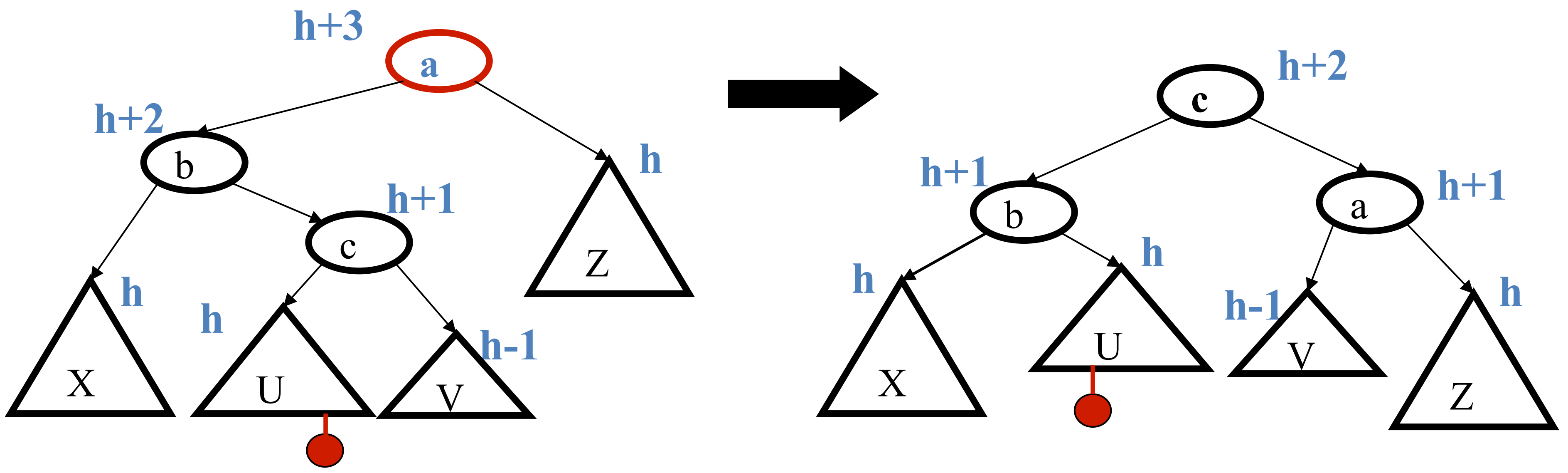
Easier to remember than you may think:

1) Move c to grandparent's position

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

## The last case: left-right

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



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

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

- Review of Amortized Analysis:
  - feel comfortable proving a runtime's amortized cost
- 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