CSE332: Data Abstractions

Lecture 7: AVL Trees

Tyler Robison

Summer 2010

The AVL Tree Data Structure

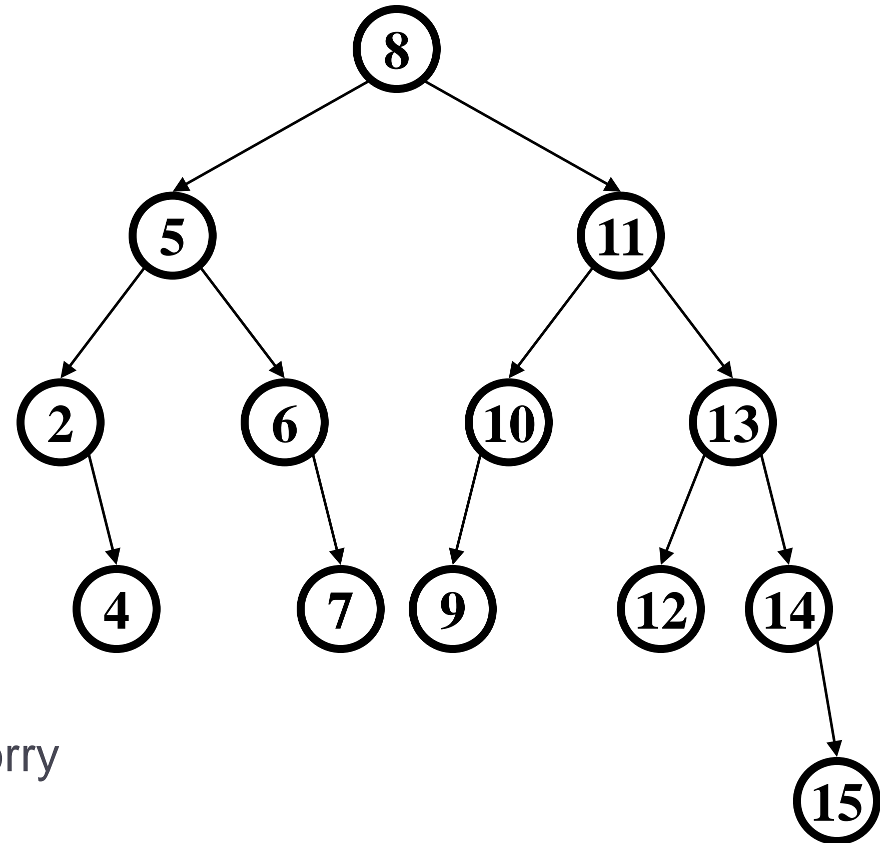
An AVL tree is a BST

In addition: Balance property:
balance of every node is between -1 and 1

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

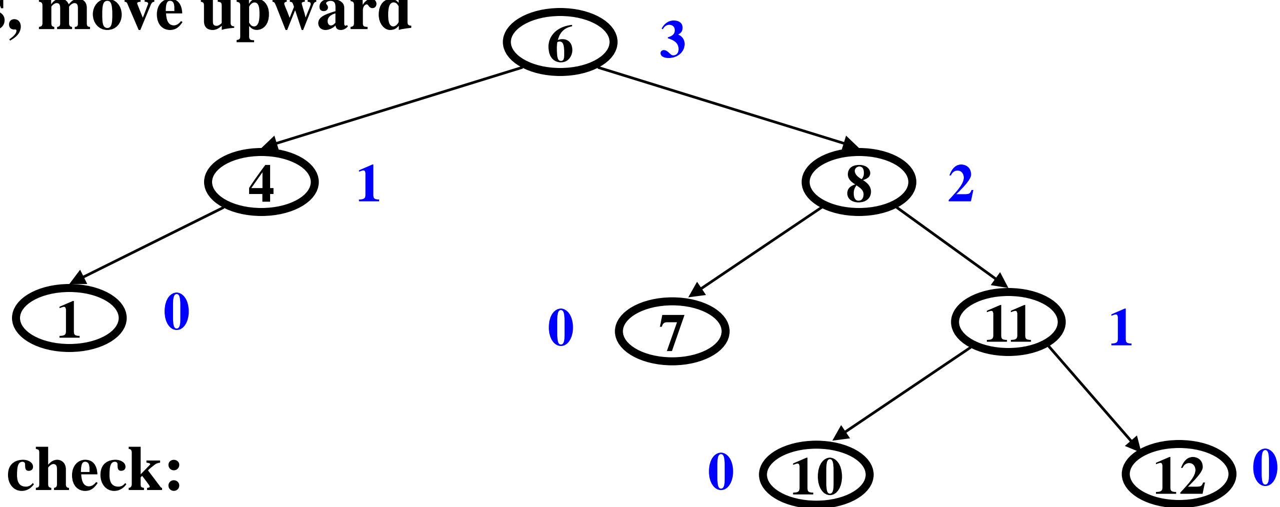
Result: **Worst-case** depth is O(log *n*)

How are we going to maintain this? Worry about that later…



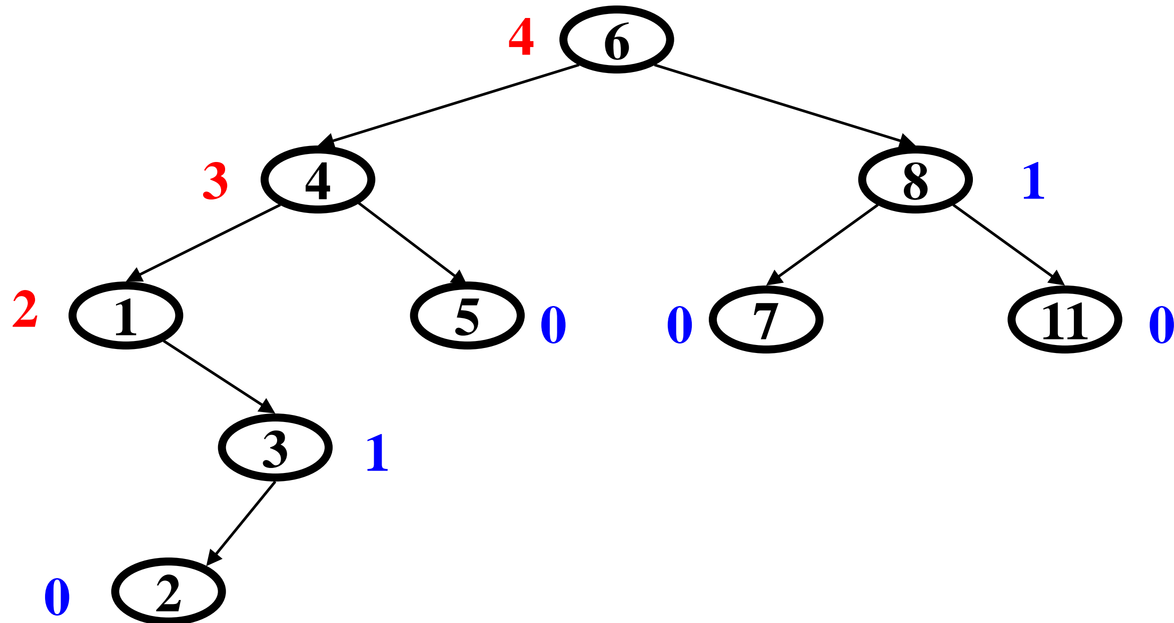
Is it an AVL tree?

BST? Check
Height/balance property?
Start at leaves, move upward



At each node, check:
Are heights of left & right within 1 of each other?

Is this an AVL tree?



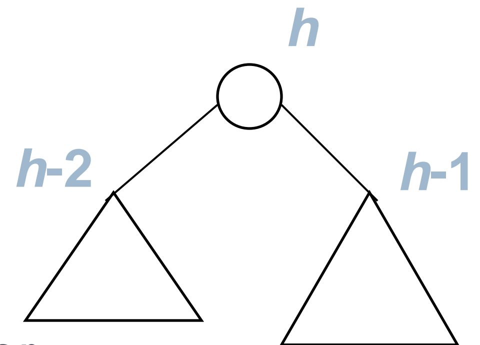
Null child has height of -1; this is an imbalance

The shallowness bound: Proving that the AVL balance property is 'enough'

Let $S(h)$ = the minimum number of nodes in an AVL tree of height h

- If we can prove that $S(h)$ grows exponentially in h, then a tree with n nodes has a logarithmic height

- Step 1: Define $S(h)$ inductively using AVL property
  - $S(-1)=0$, $S(0)=1$, $S(1)=2$
  - *For $h \geq 2$, $S(h) = 1+S(h-1)+S(h-2)$*
  - Build our minimal tree from smaller minimal trees, w/ heights within 1 of each other

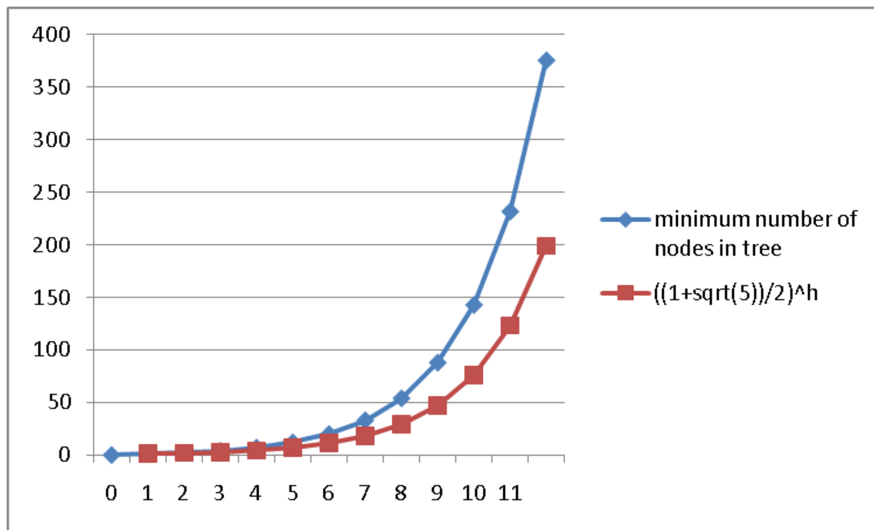


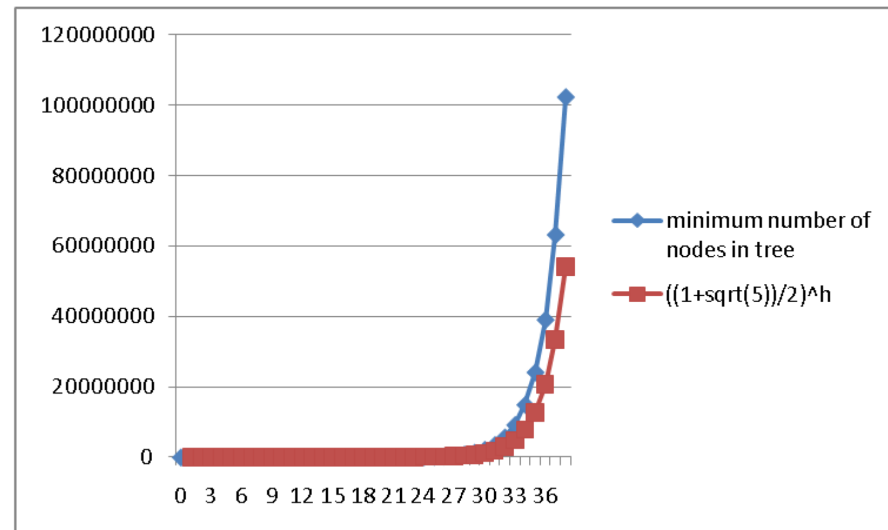
Notational note:
Oval: a node in the tree
Triangle: a subtree

- Step 2: Show this recurrence grows really fast
  - Similar to Fibonacci numbers
  - Can prove for all h, $S(h) > \phi^h - 1$ where ϕ is the golden ratio, $(1+\sqrt{5})/2$, about 1.62
  - Growing faster than 1.6^h is "plenty" exponential

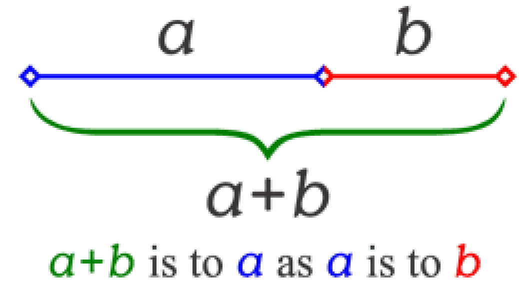
Before we prove it

- Good intuition from plots comparing:
  - $S(h)$ computed directly from the definition
  - $((1+\sqrt{5})/2)^h$
- $S(h)$ is always bigger
  - Graphs aren't proofs, so let's prove it





The Golden Ratio



$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.62$$

This is a special number: If **(a+b)/a = a/b**, then **a = ϕb**

- Aside: Since the Renaissance, many artists and architects have proportioned their work (e.g., length:height) to approximate the ***golden ratio***

- We will need one special arithmetic fact about ϕ :

$$
\begin{aligned}
\phi^2 &= ((1+5^{1/2})/2)^2 \\
&= (1 + 2*5^{1/2} + 5)/4 \\
&= (6 + 2*5^{1/2})/4 \\
&= (3 + 5^{1/2})/2 \\
&= 1 + (1 + 5^{1/2})/2 \\
&= 1 + \phi
\end{aligned}
$$

The proof

$S(-1)=0$, $S(0)=1$, $S(1)=2$
For $h \geq 2$, $S(h) = 1+S(h-1)+S(h-2)$

Theorem: For all $h \geq 0$, $S(h) > \phi^h - 1$

Proof: By induction on h

Base cases:

$S(0) = 1 > \phi^0 - 1 = 0$ $S(1) = 2 > \phi^1 - 1 \approx 0.62$

Inductive case ($k > 1$):

Inductive hypotheses: $S(k) > \phi^k - 1$ and $S(k-1) > \phi^{k-1} - 1$

Show $S(k+1) > \phi^{k+1} - 1$

$S(k+1) = 1 + S(k) + S(k-1)$ by definition of S

$> 1 + \phi^k - 1 + \phi^{k-1} - 1$ by inductive hypotheses

$= \phi^k + \phi^{k-1} - 1$ by arithmetic (1-1=0)

$= \phi^{k-1} (\phi + 1) - 1$ by arithmetic (factor ϕ^{k-1})

$= \phi^{k-1} \phi^2 - 1$ by special property of ϕ

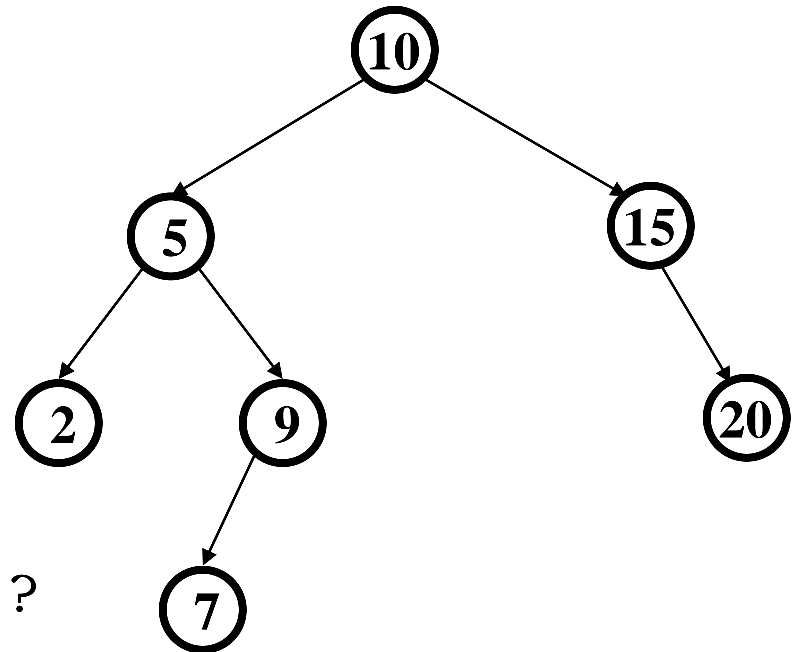
$= \phi^{k+1} - 1$ by arithmetic (add exponents)

Good news

Proof means that if we have an AVL tree, then **`find`** is $O($**`log`** $n)$

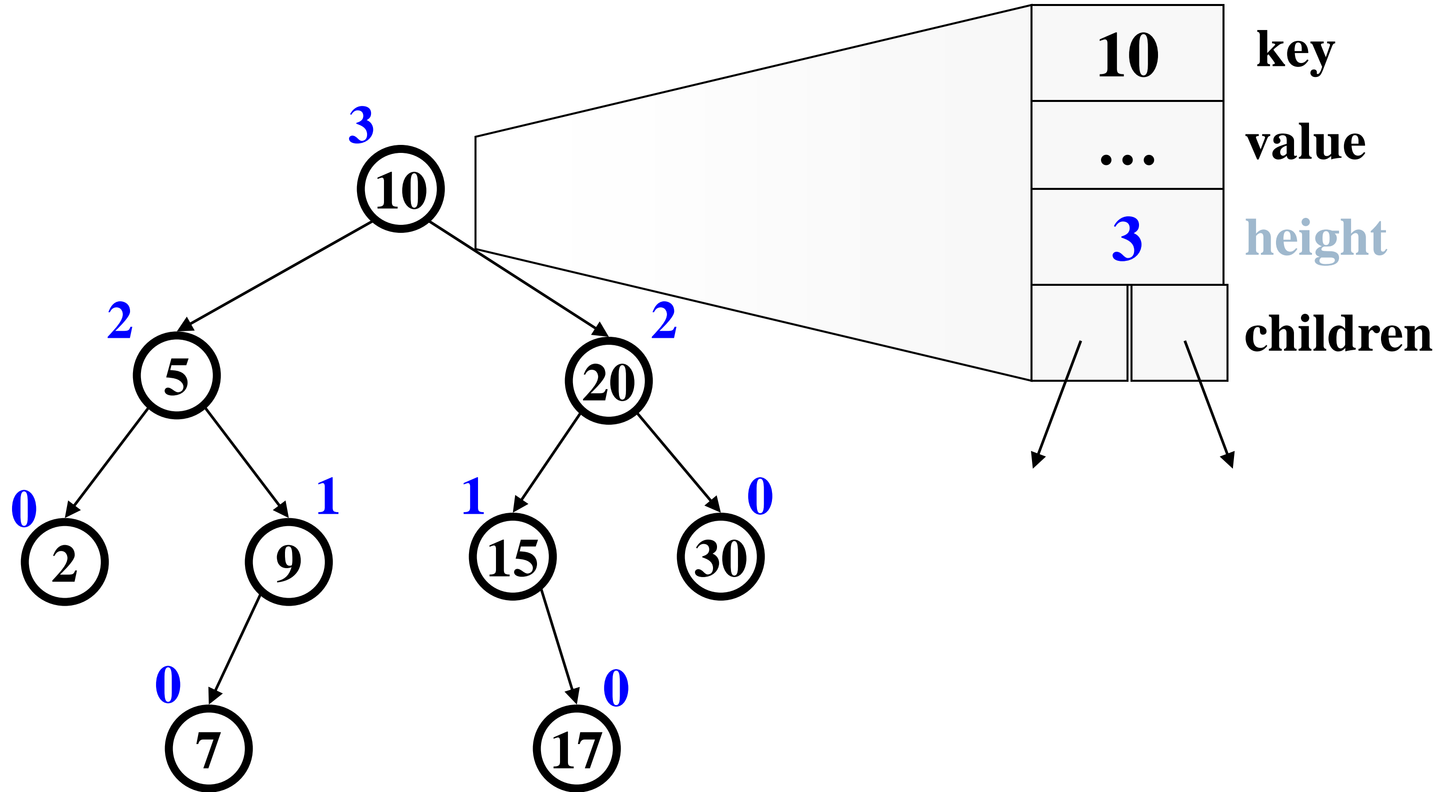
But as we insert and delete elements, we need to:

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



Is this AVL tree balanced?
How about after **`insert(30)`**?

An AVL Tree

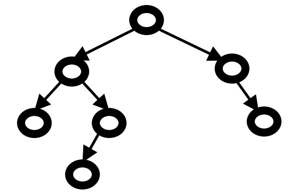


Track height at all times!

AVL tree operation overview

- **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, like insert we 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 when returning from insertion in a subtree:
   1. Update heights, if necessary
   2. Detect height imbalance
   3. Perform a *rotation* to restore balance at that node, if necessary

Fact that makes it a bit easier:

- We'll only need to do one type of rotation in one place to fix balance
- That is, a single 'move' can fix it all

Insertion Imbalance Example

Insert(6)

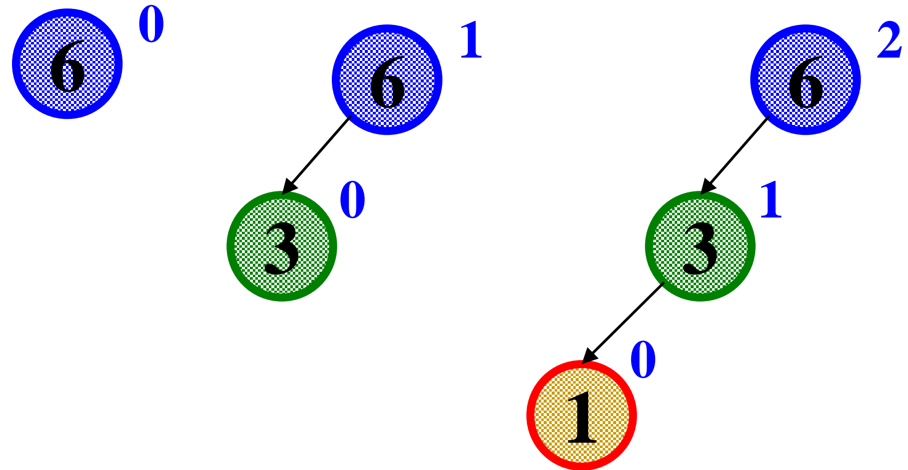
Insert(3)

Insert(1)

Third insertion violates balance property

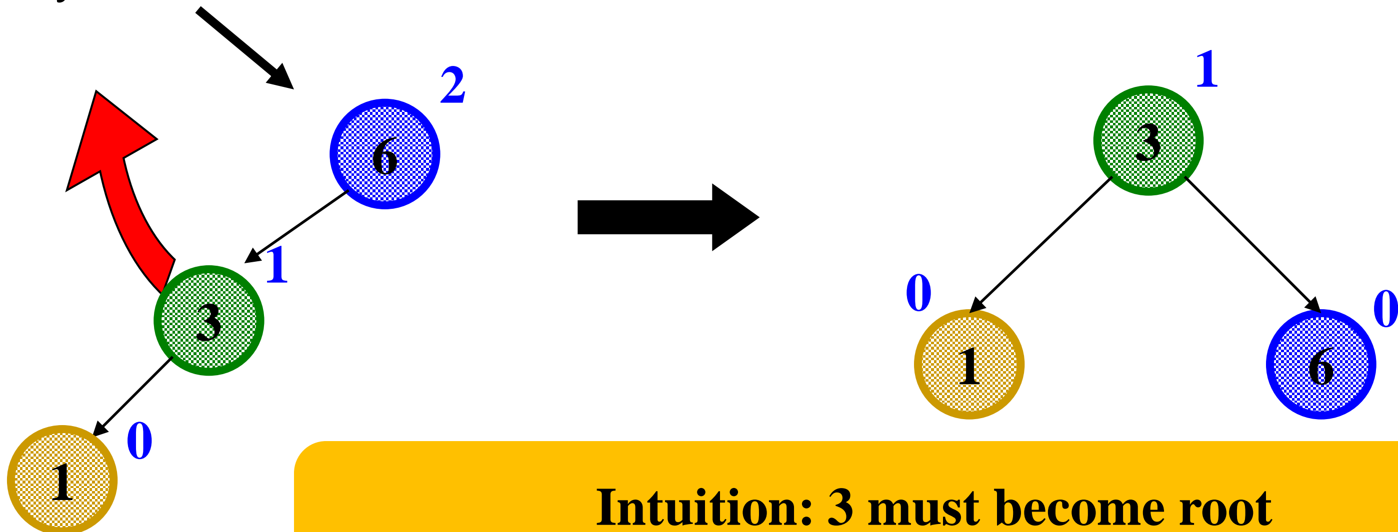
- happens to be detected at the root

What is the only way to fix this?



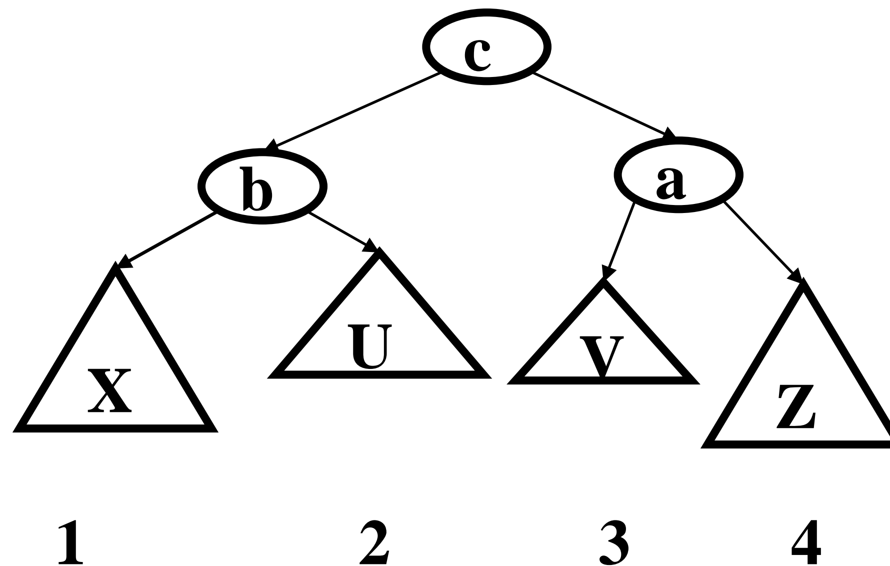
Fix for 'left-left' case: Apply 'Single Rotation'

- *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

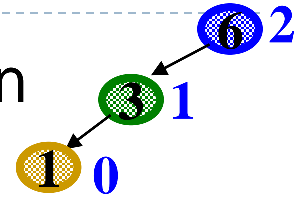


4 Different rotation cases

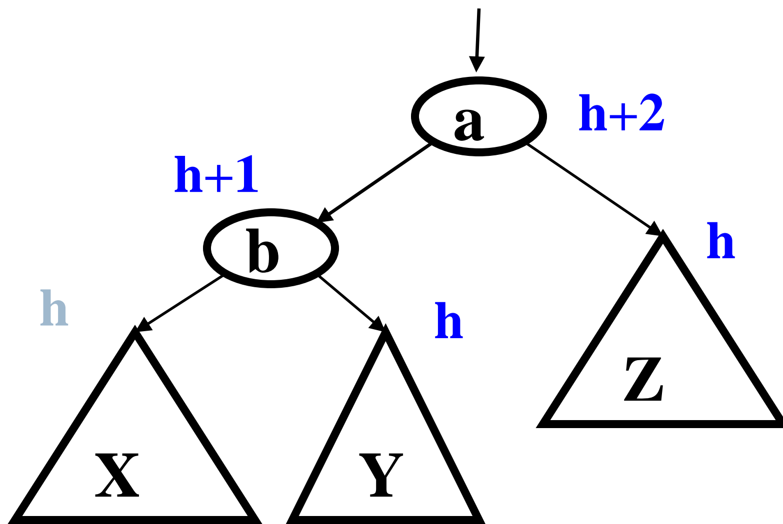
- Do the BST insertion, recurse back up the tree and check for an imbalance at each node
- If an imbalance is detected at node c, we'll perform 1 of 4 types of rotations to fix it, depending on which subtree the insertion was in

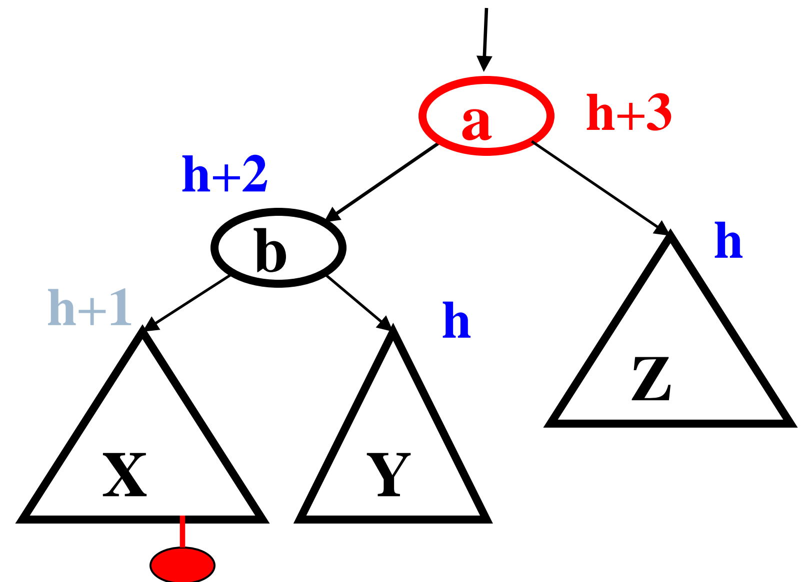


Back to our example (case 1)



- Node imbalanced due to insertion *somewhere* in **left-left grandchild** that increased the height
- First we did the insertion, which would make node ***a*** imbalanced



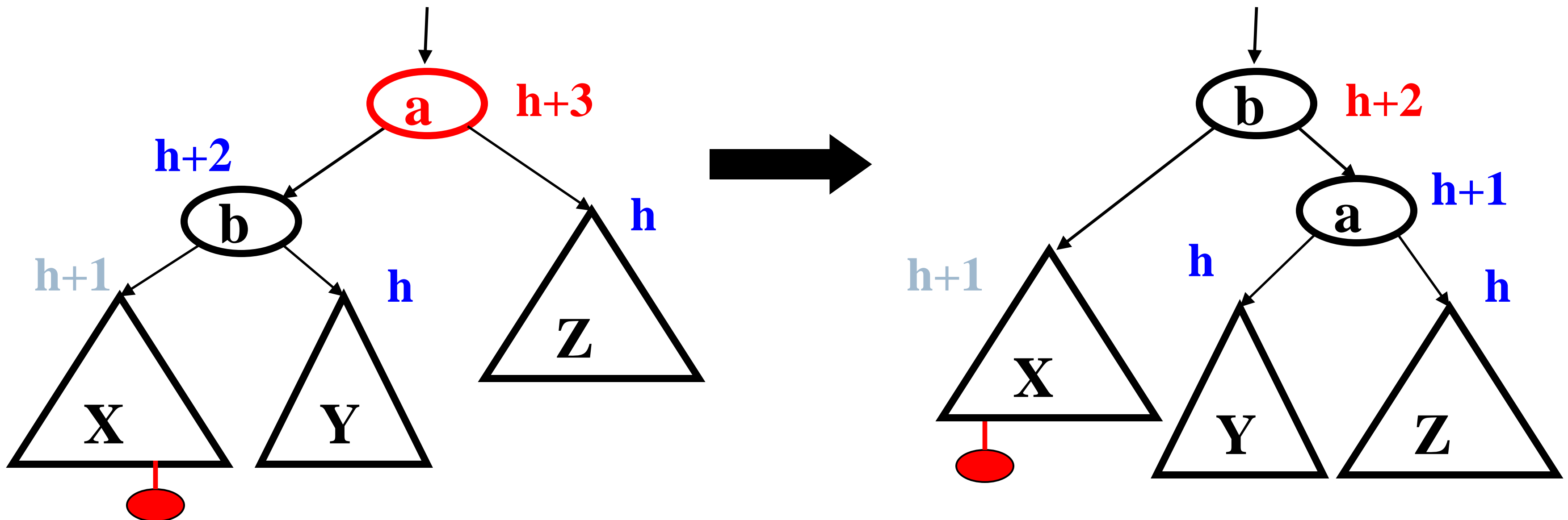


Before insertion **After insertion**

The general left-left case

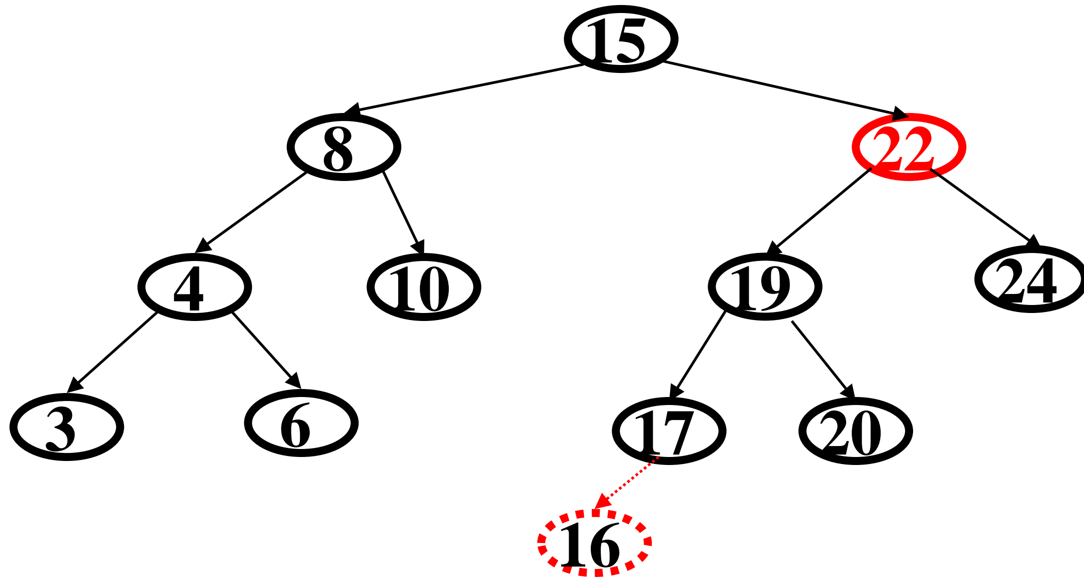
- Rotate at node `a`, using the fact that in a BST:

  X < b < Y < a < Z

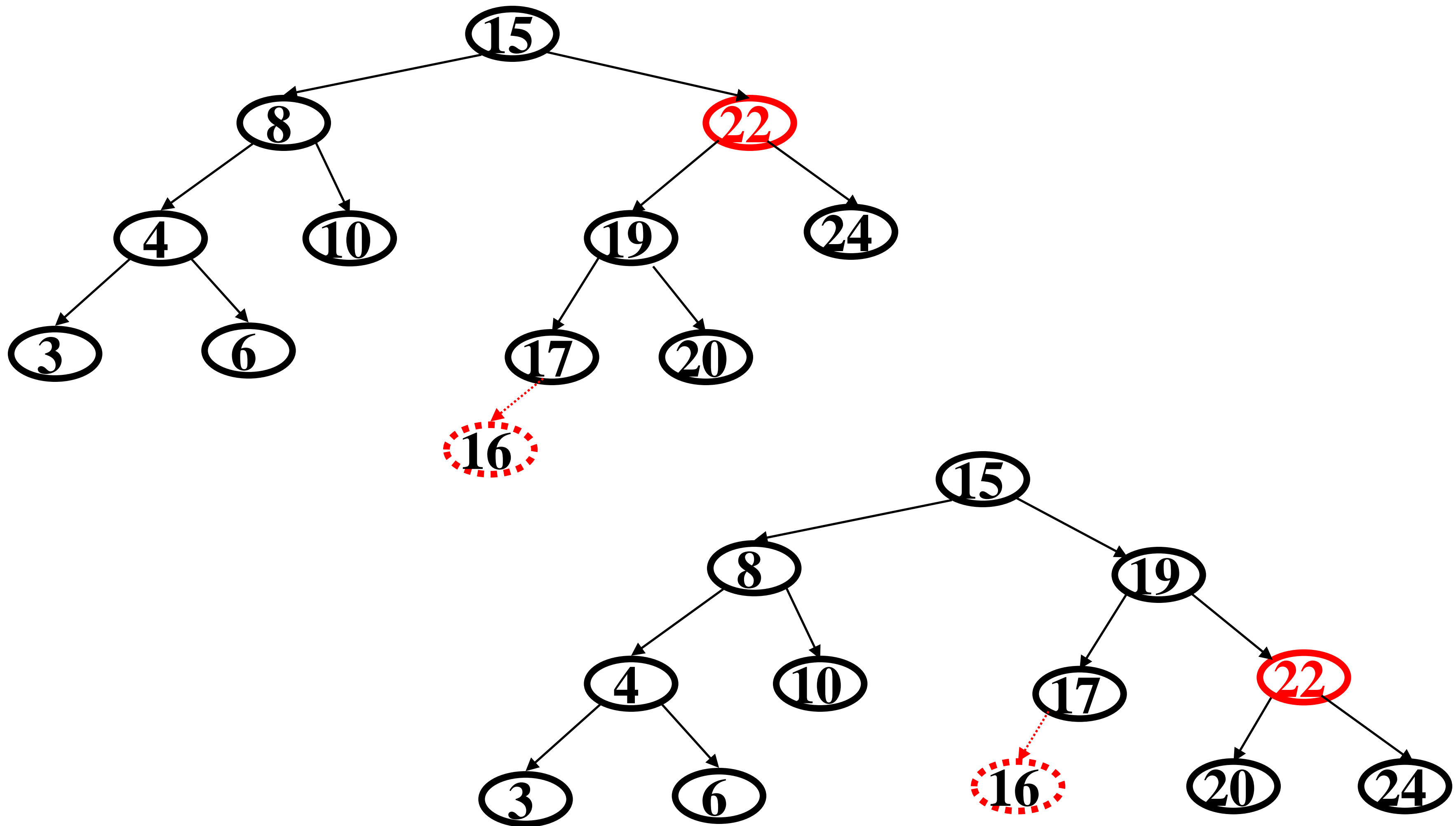


- A single rotation restores balance at the node
  - To same height as before insertion (so ancestors now balanced)

Another example: `insert(16)`

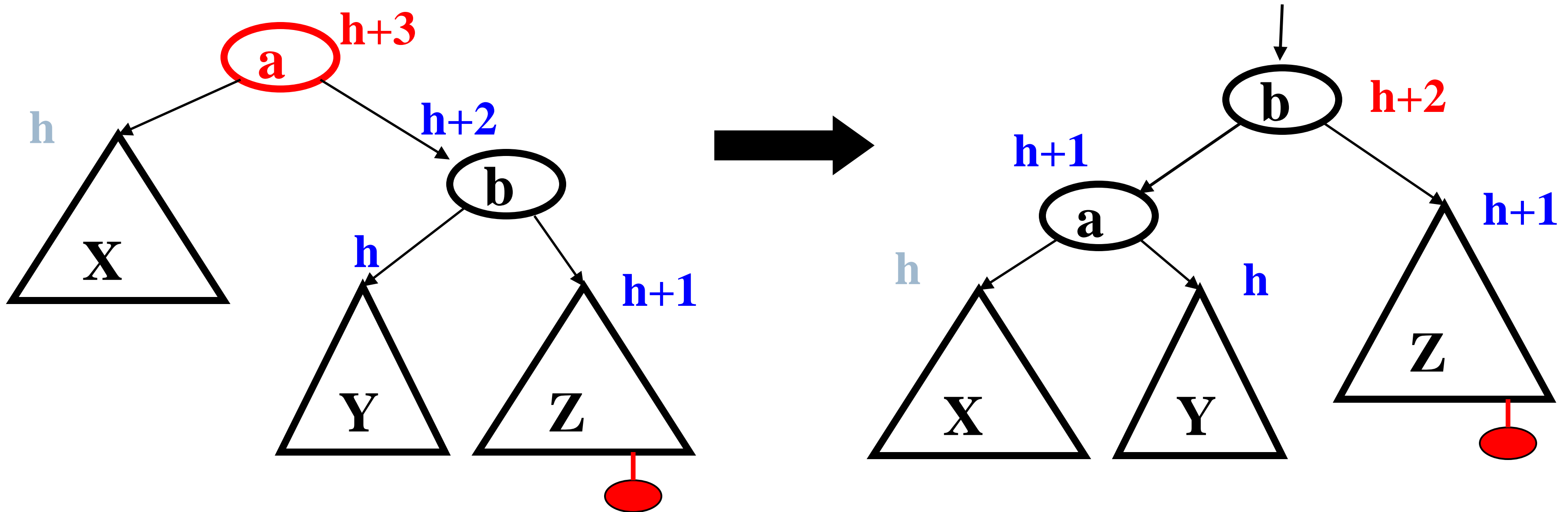


Another case 1 example: `insert(16)`



The general right-right case (case 4)

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

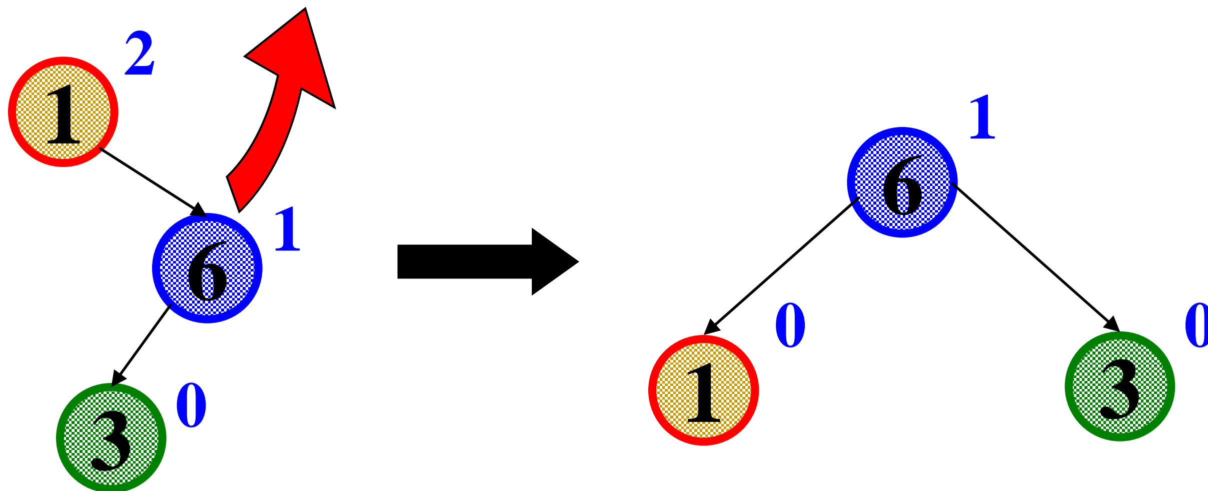


Two cases to go

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

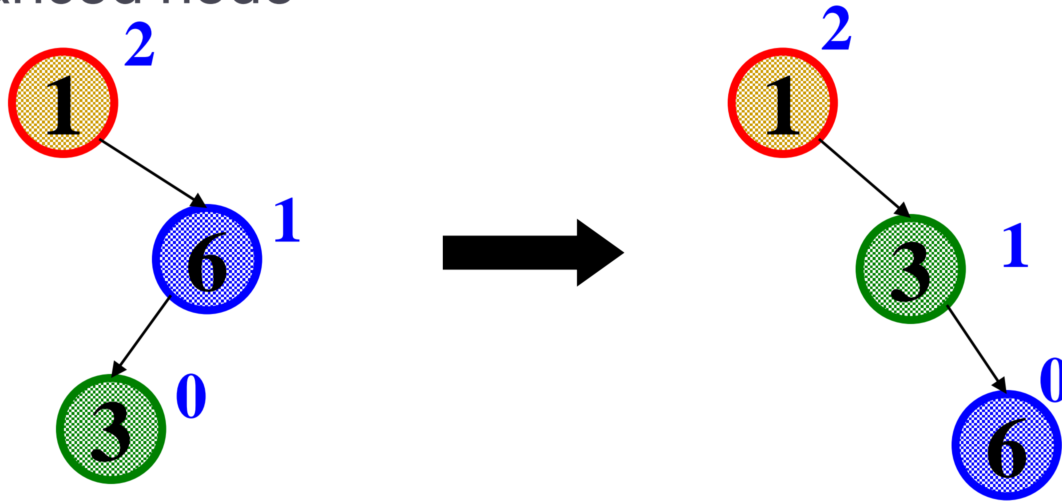


Two cases to go

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

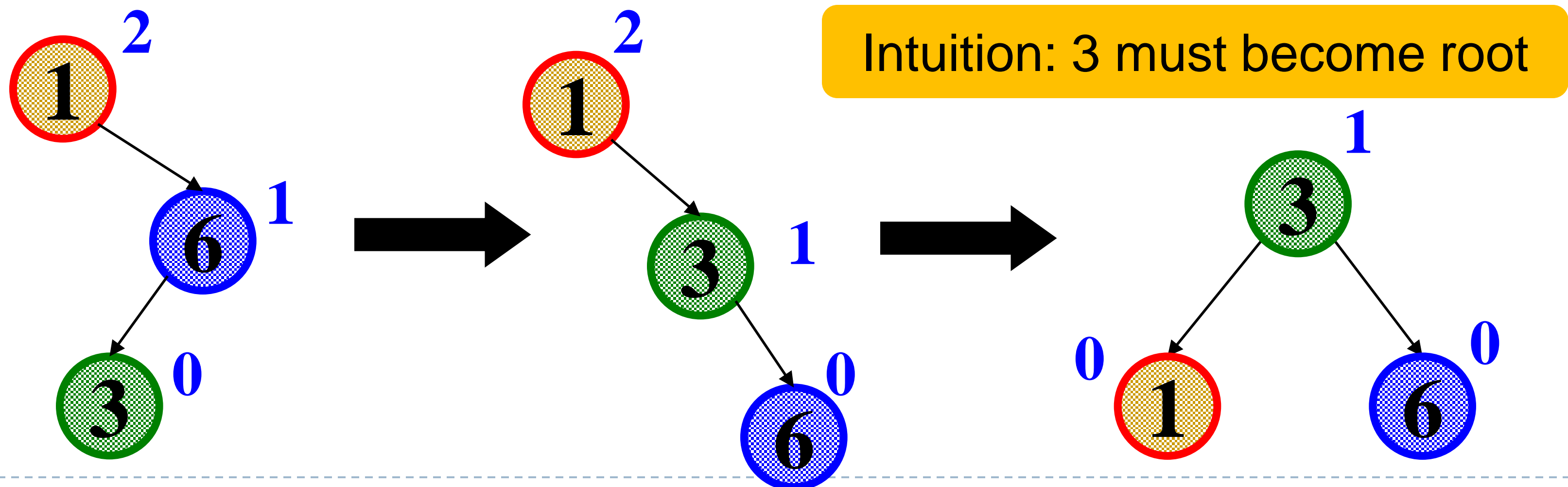
Simple example: **`insert`**(1), **`insert`**(6), **`insert`**(3)

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

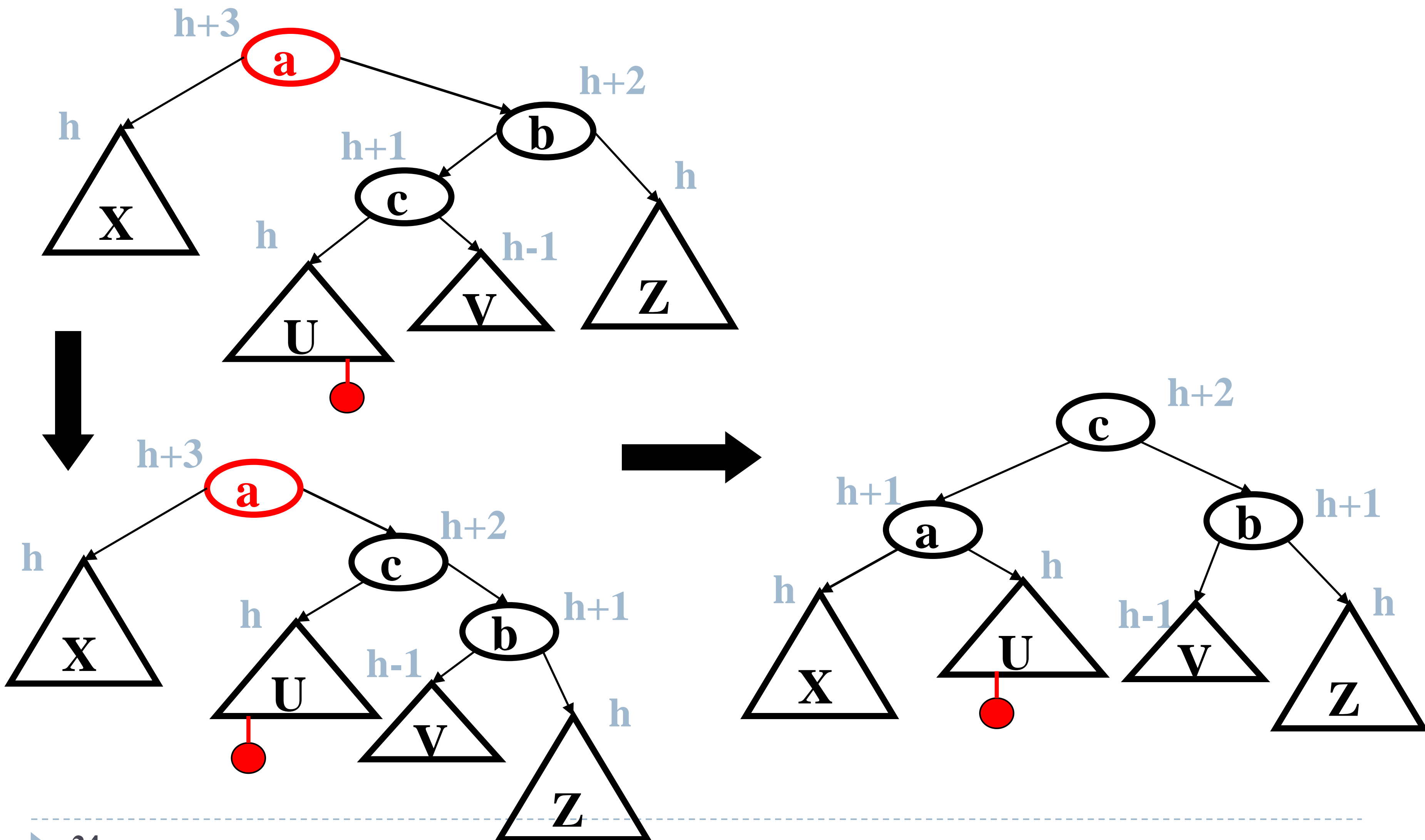


Sometimes two wrongs make a right ☺

- 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

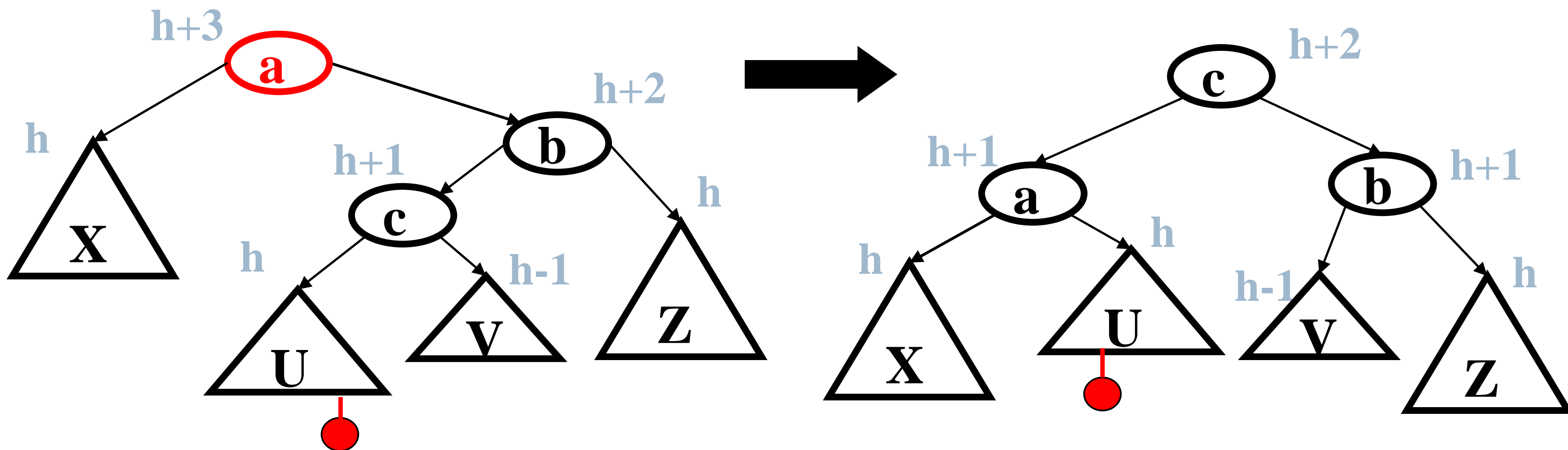


The general right-left case



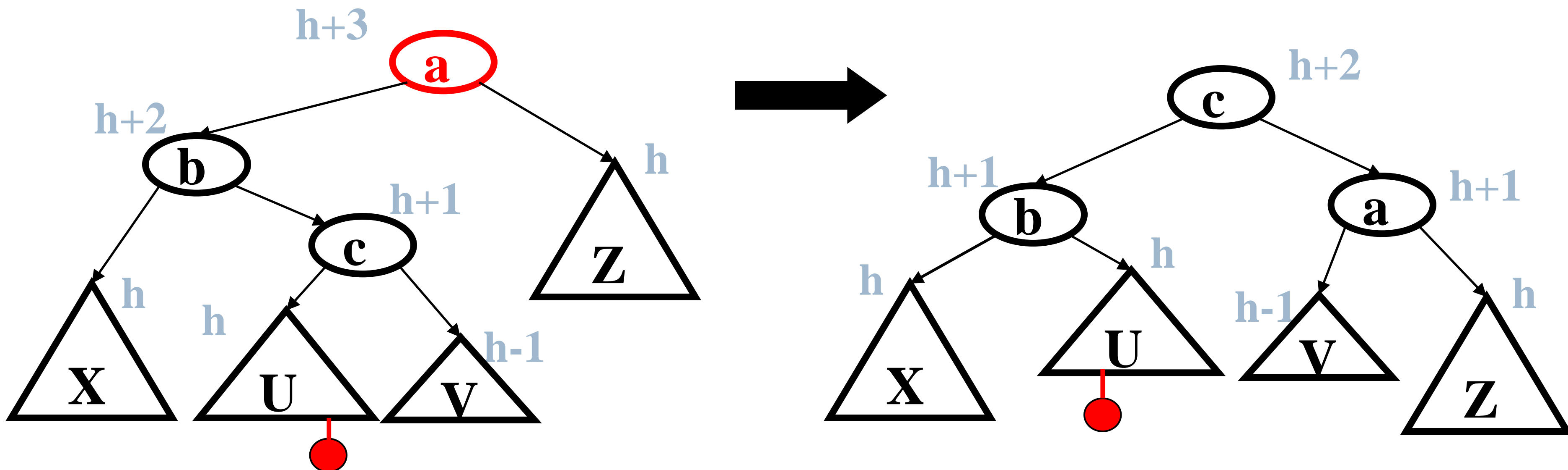
Comments

- 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:



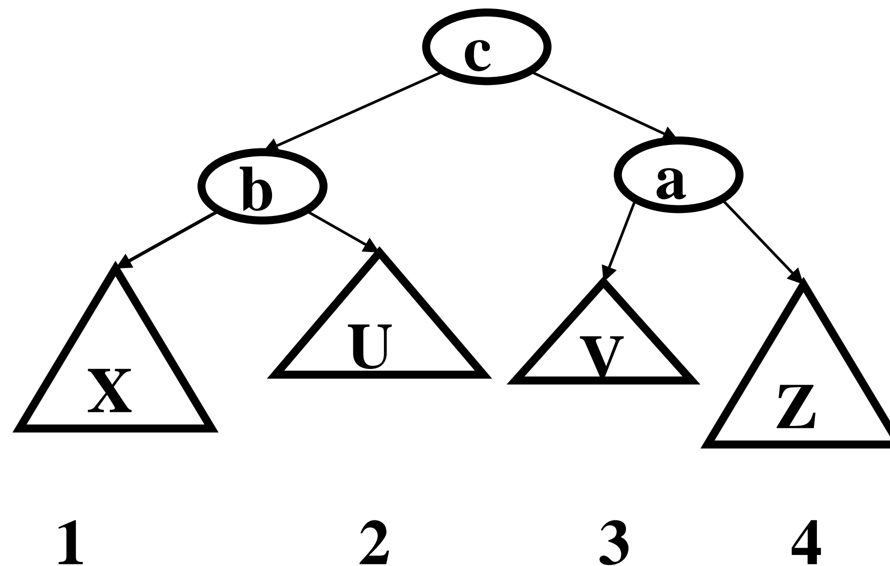
The last case: left-right

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



4 Different rotation cases

- Do the BST insertion, recurse back up the tree and check for an imbalance at each node
- If an imbalance is detected at node c, we'll perform 1 of 4 types of rotations to fix it, depending on which subtree the insertion was in



Insert, summarized

- Insert as in a BST
- Check back up path for imbalance, which will be 1 of 4 cases:
  - 1. node's left-left grandchild is too tall
  - 2. node's left-right grandchild is too tall
  - 3. node's right-left grandchild is too tall
  - 4. node's right-right grandchild is too tall
- 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

Now efficiency

Have argued rotations restore AVL property but do they produce an efficient data structure?

- Worst-case complexity of **`find`**: **O(logn)**
- Worst-case time to do a rotation? **O(1)**
- Worst-case complexity of **`insert`**: **O(logn)**

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced.
2. The 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
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, our next data structure)

What we're missing

- Splay Trees
  - BST
  - No balance condition (don't need to store height)
  - No O(logn) guarantee; can be O(n)
  - Instead, amortized guarantee: O(logn) over the course of a large number of operations
  - Neat caching behavior: when you access a node (find, insert), you 'splay' it to the top: it becomes the new root