CSE373: Data Structures & Algorithms

Lecture 7: AVL Trees

Linda Shapiro

Winter 2015

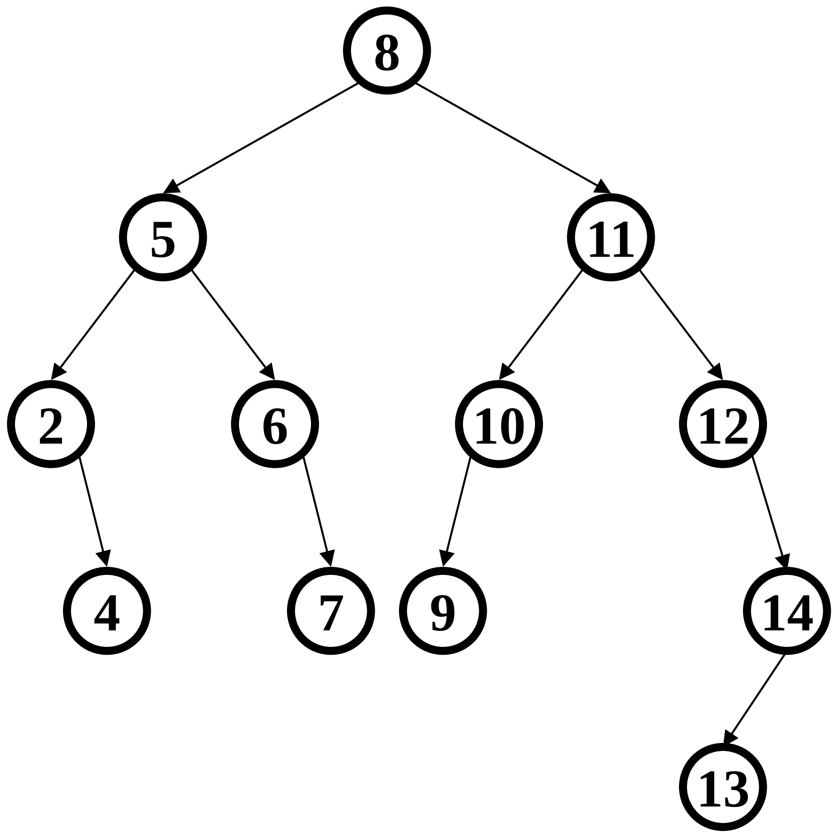
Announcements

- HW2 due TODAY
- TA sessions this week
  - Thursday: Binary Search Trees and AVL Trees

- Last lecture: Binary Search Trees
- Today… AVL Trees

Review: Binary Search Tree (BST)

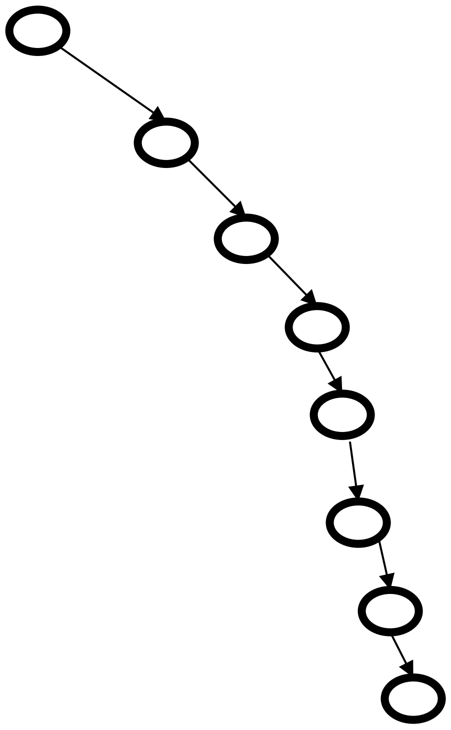
- Structure property (binary tree)
  - Each node has ≤ 2 children
  - Result: keeps operations simple
- Order property
  - All keys in left subtree smaller than node's key
  - All keys in right subtree larger than node's key
  - Result: easy to find any given key



BST: Efficiency of Operations?

- Problem: operations may be inefficient if BST is unbalanced.

- Find, insert, delete
  - O(n) in the worst case
- BuildTree
  - O(n^2) in the worst case



How can we make a BST efficient?

Observation

- BST: the shallower the better!

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

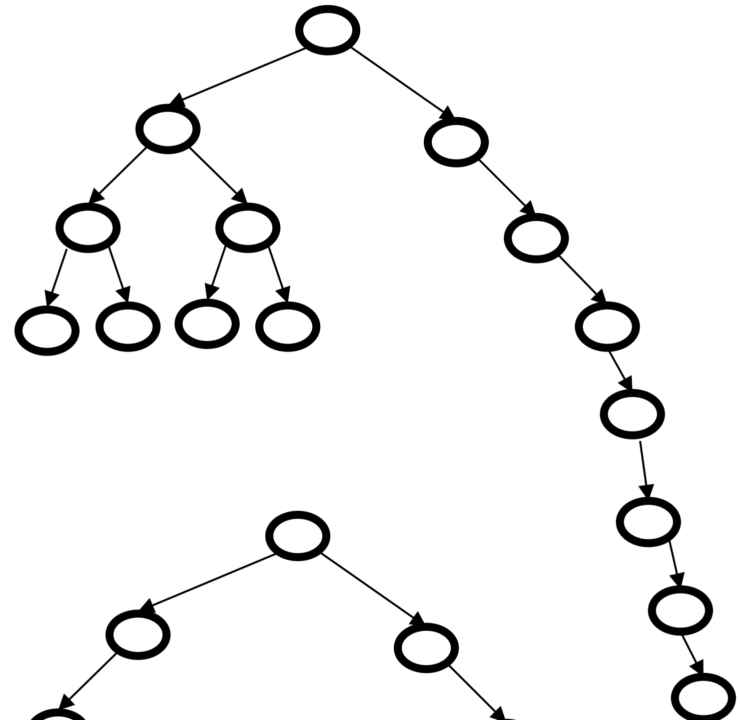
1. Ensures depth is always $O(\mathbf{log}\ n)$ – strong enough!
2. Is efficient to maintain – not too strong!

- When we build the tree, make sure it's balanced.
- BUT…Balancing a tree only at build time is insufficient because sequences of operations can eventually transform our carefully balanced tree into the *dreaded list* ☹
- So, we also need to also keep the tree balanced as we perform operations.

Potential Balance Conditions

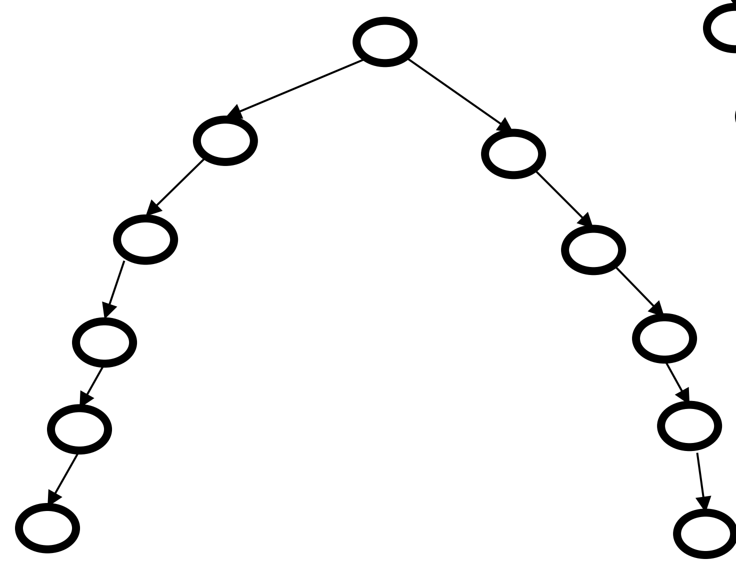
1. Left and right subtrees of the *root* have equal number of nodes

   Too weak!
   Height mismatch example:



2. Left and right subtrees of the *root* have equal *height*

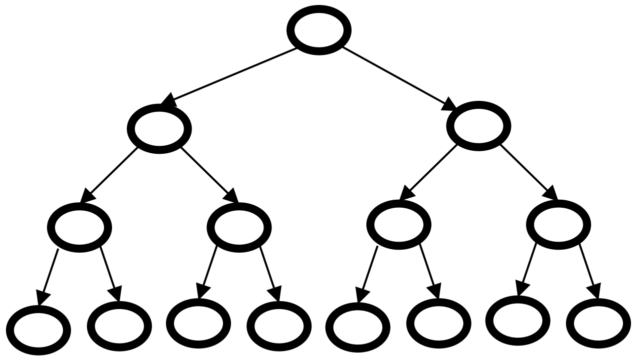
   Too weak!
   Double chain example:



Potential Balance Conditions

3. Left and right subtrees of every node have equal number of nodes

   Too strong!
   Only perfect trees (2^n – 1 nodes)



4. Left and right subtrees of every node have equal *height*

   Too strong!
   Only perfect trees (2^n – 1 nodes)

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
  - This is because an AVL tree of height *h* must have a number of nodes *exponential* in *h*

    *Thus height must be **log(number of nodes).***

- Efficient to maintain
  - Using single and double rotations

The AVL Tree Data Structure

An AVL tree is a self-balancing binary search tree.

Structural properties

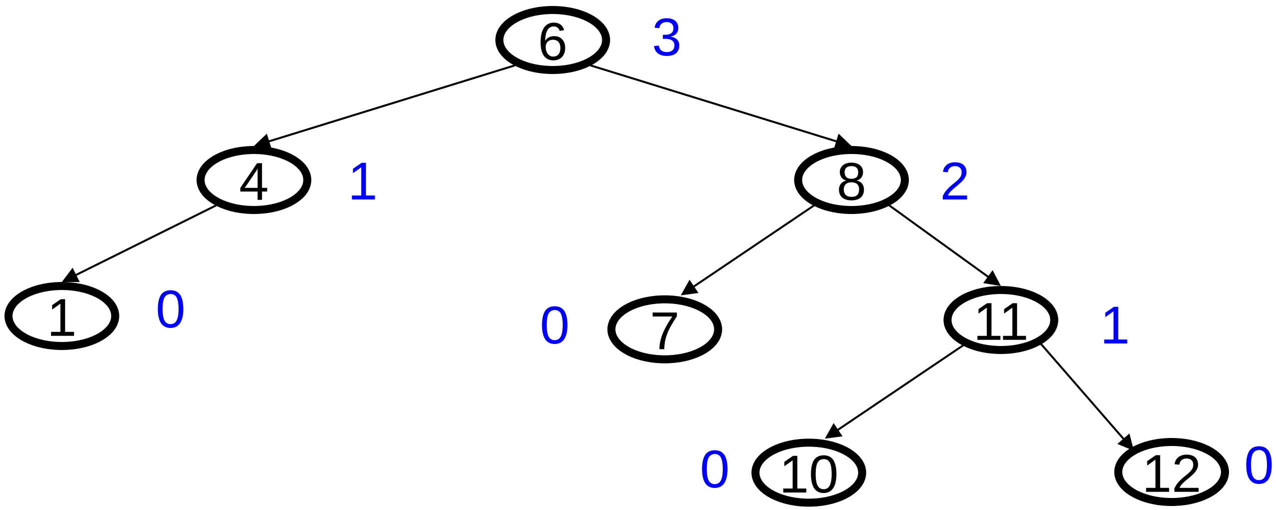
1. Binary tree property (same as BST)
2. Order property (same as for BST)

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

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

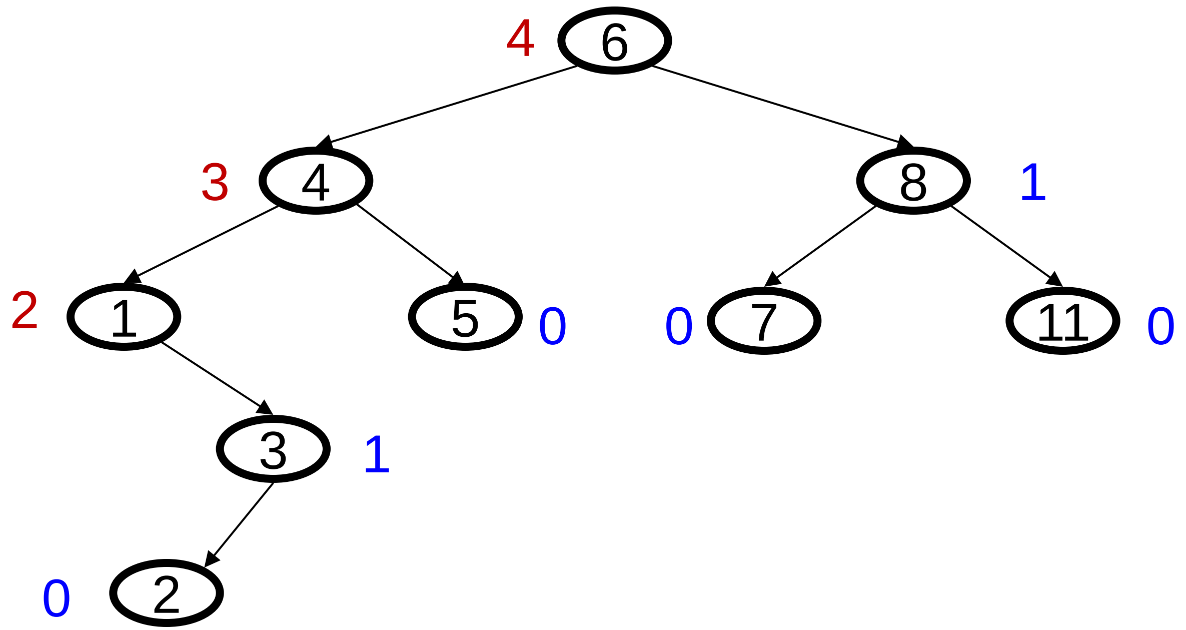
- Named after inventors Adelson-Velskii and Landis (AVL)
  - First invented in 1962

Is this an AVL tree?



Yes! Because the left and right subtrees of *every node* have *heights* differing by *at most 1*

Is this an AVL tree?



Nope! The left and right subtrees of some nodes (e.g. 1, 4, 6) have heights that differ by *more than 1*

What do AVL trees give us?

- If we have an AVL tree, then the number of nodes is an exponential function of the height.

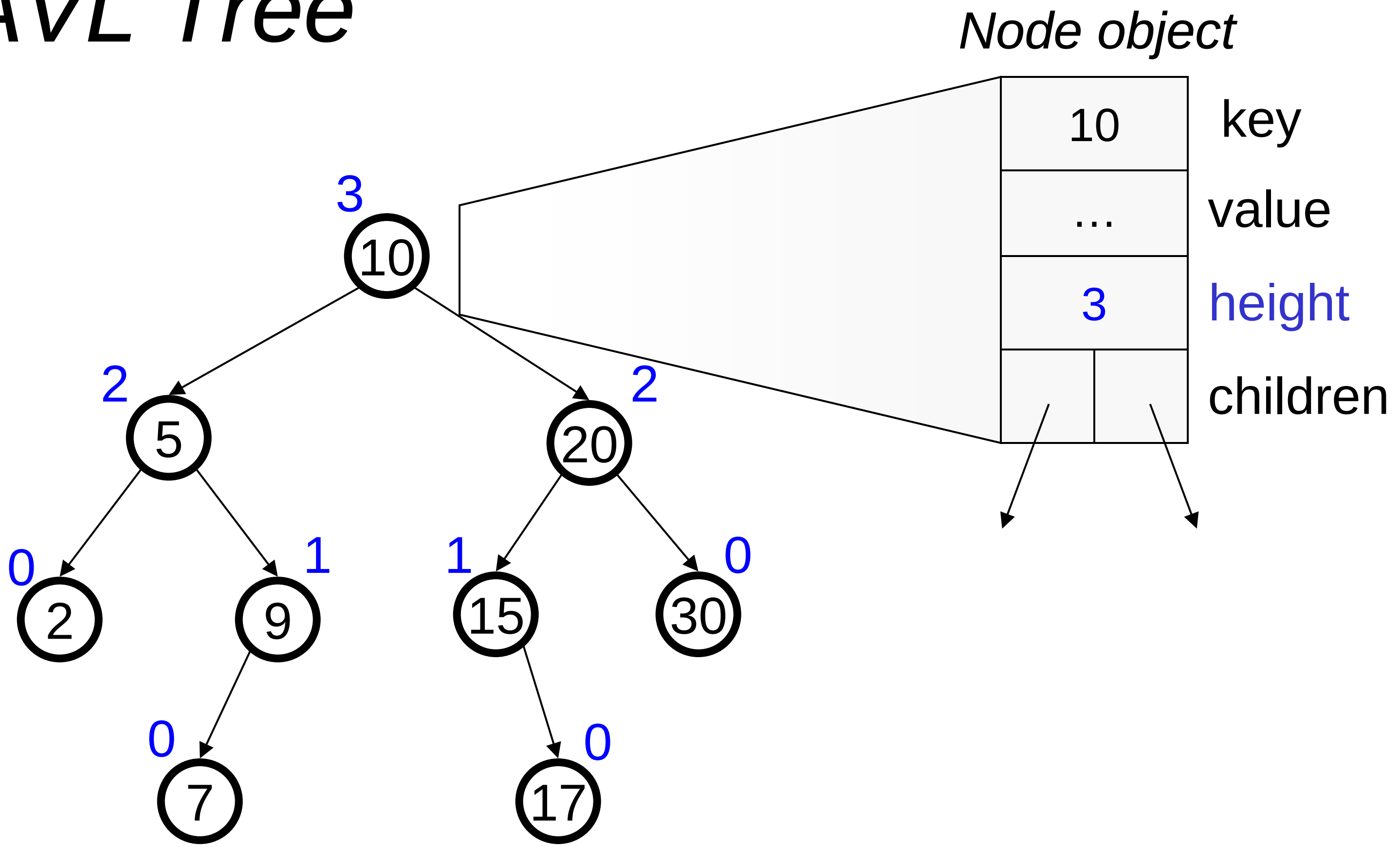
- **`Thus the height is a log function of the number of nodes!`**

- **`And thus find`** is $O(\mathbf{log}\ n)$

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

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

An AVL Tree



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 check for several imbalance cases (we will skip this)

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 insertion in a subtree, detect height imbalance and perform a *rotation* to restore balance at that node

All the action is in defining the correct rotations to restore balance

Fact that an implementation can ignore:

- There must be a deepest element that is imbalanced after the insert (all descendants still balanced)
- After rebalancing this deepest node, every node is balanced
- So at most one node needs to be rebalanced

Case #1: Example

Insert(6)

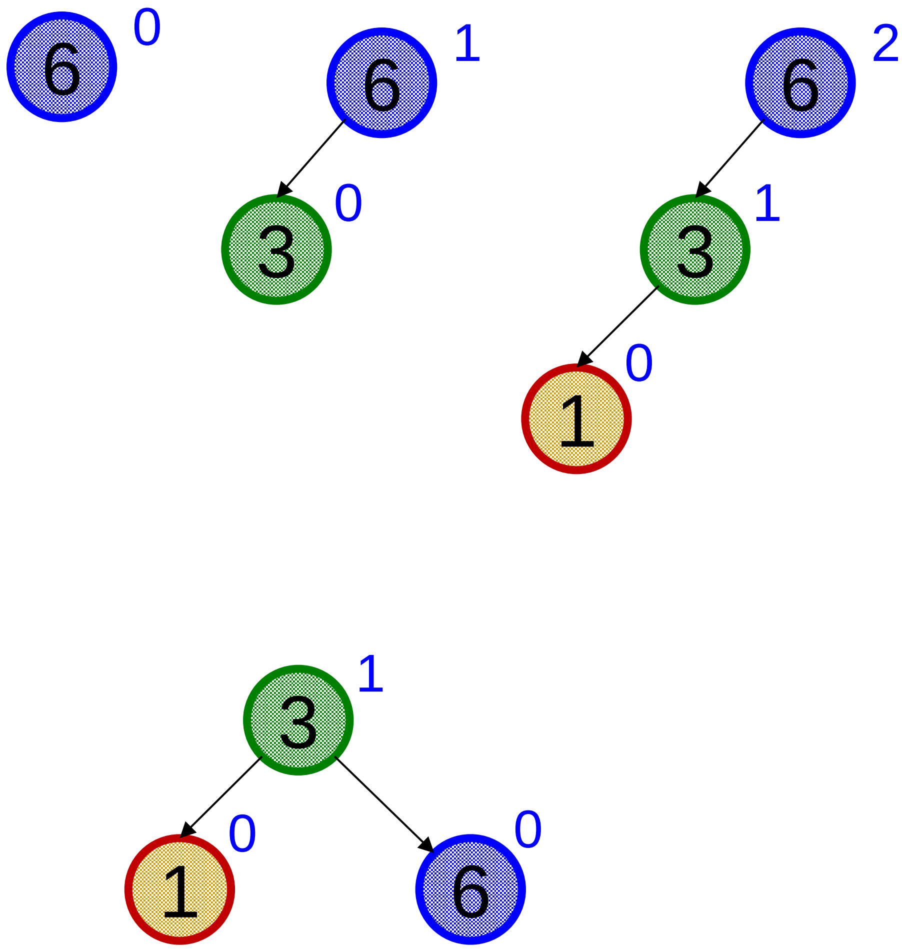
Insert(3)

Insert(1)

Third insertion violates balance property

- happens to be at the root

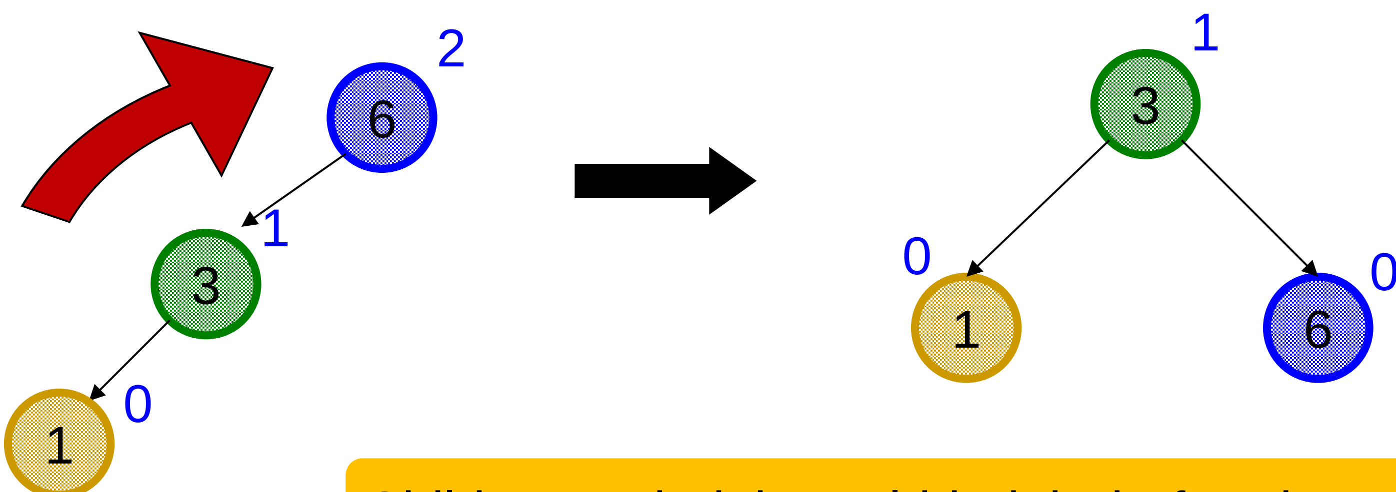
What is the only way to fix this?



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)

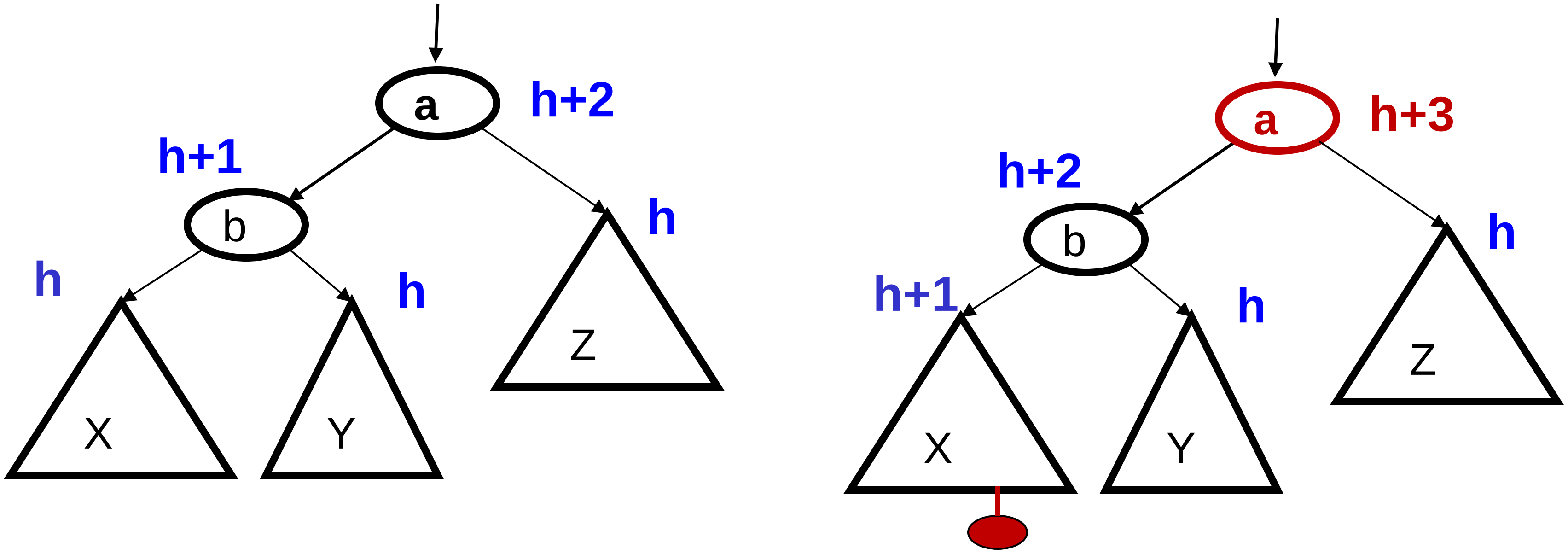
AVL Property violated at node 6



Child's new-height = old-height-before-insert

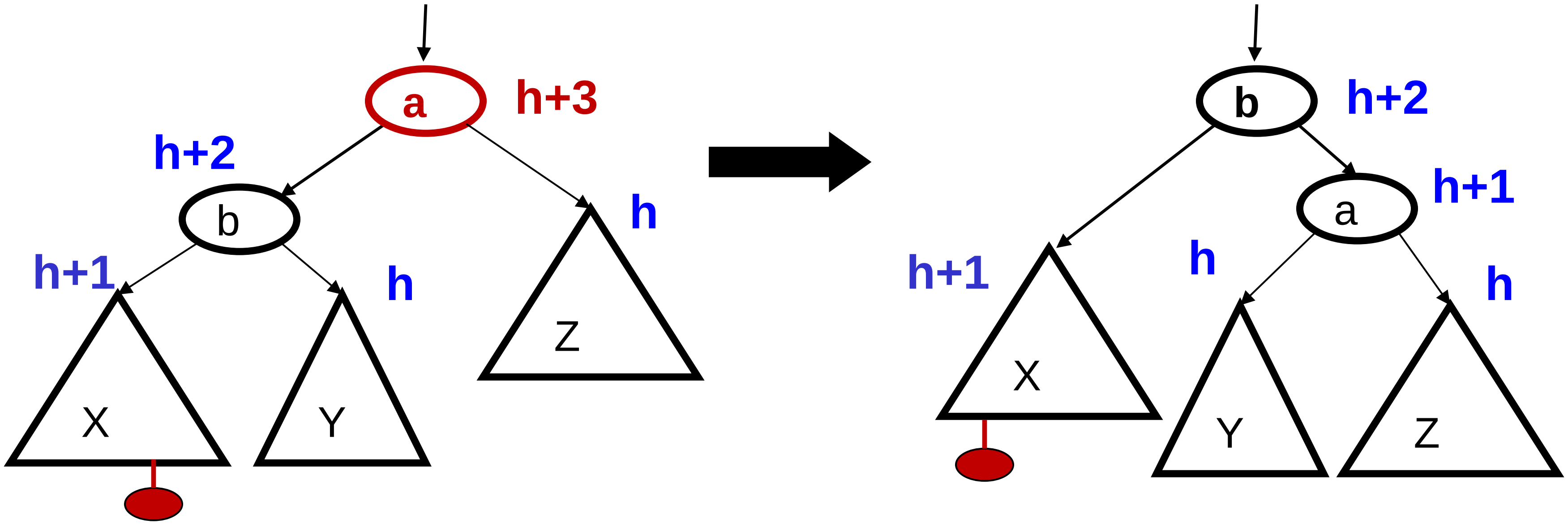
The example generalized: Left of Left

- Insertion into left-left grandchild causes an imbalance
  - 1 of 4 possible imbalance causes (other 3 coming up!)
- Creates an imbalance in the AVL tree (specifically **a** is imbalanced)



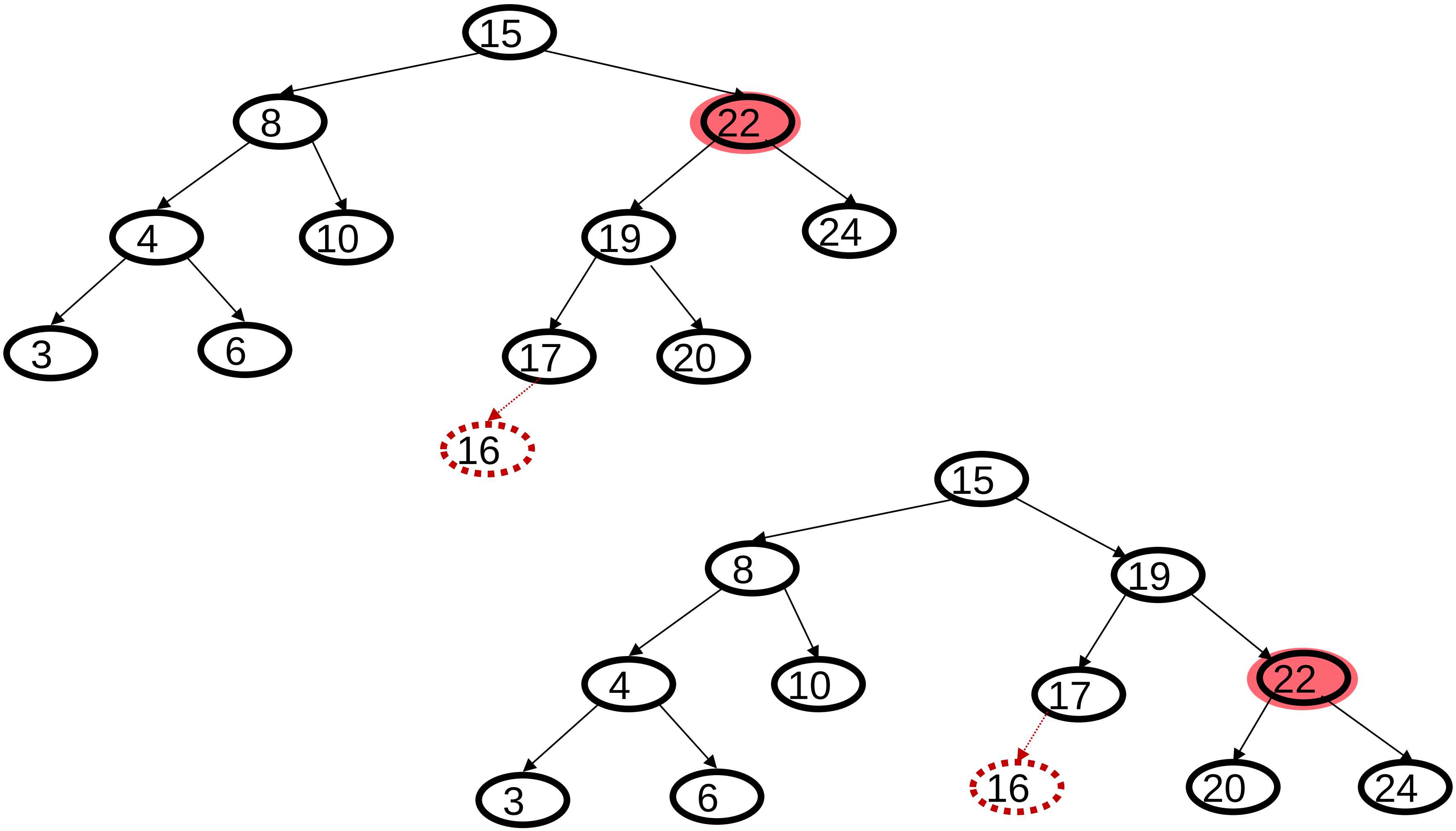
The general left-left case

- So we *rotate* at ***a***
  - Move left child of unbalanced node into parent position
  - Parent becomes the right child
  - Other sub-trees move in the only way BST allows:
    - using BST facts: X < b < Y < a < Z



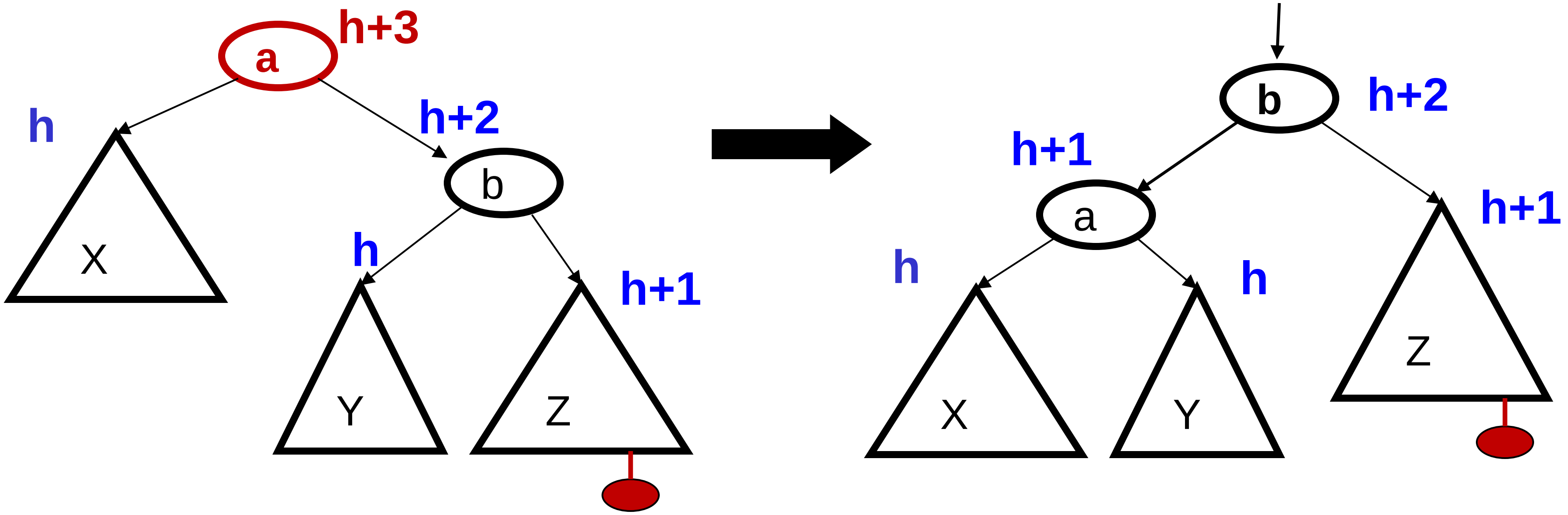
- A single rotation restores balance at the node
  - To same height as before insertion, so ancestors now balanced

Another example: **`insert(16)`**



The general right-right case

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

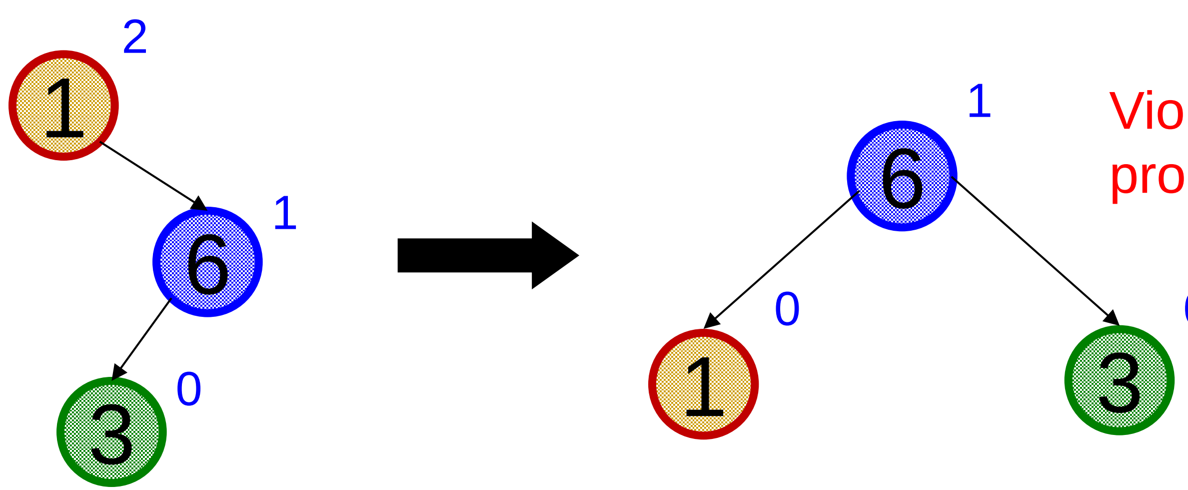


Two cases to go

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



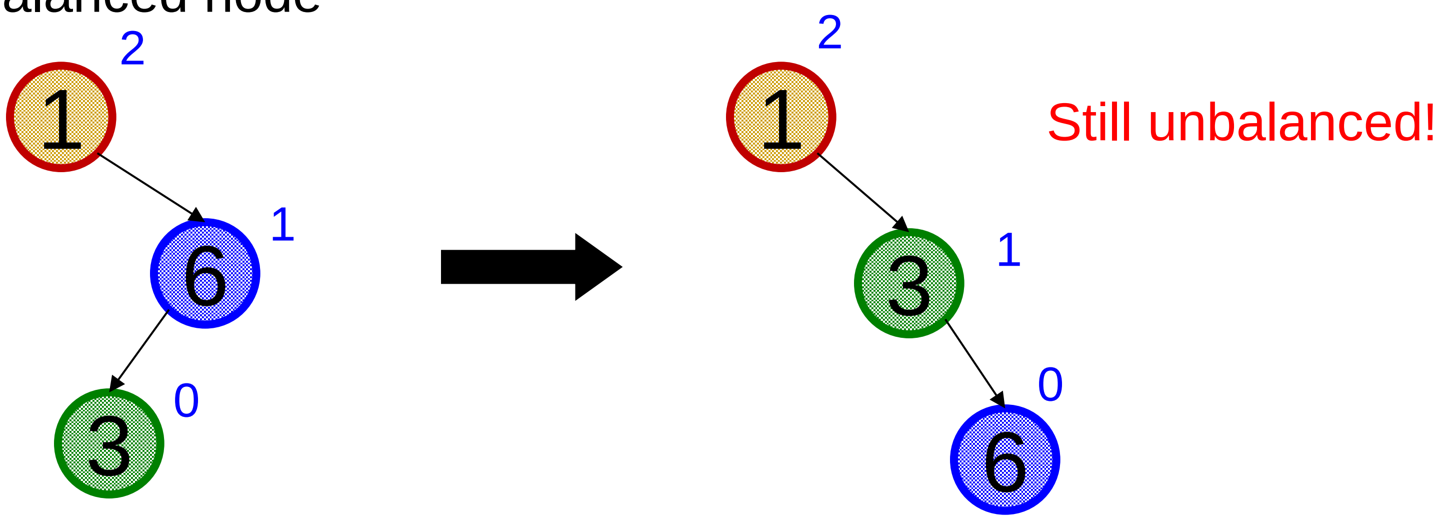
Violates order property!

Two cases to go

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

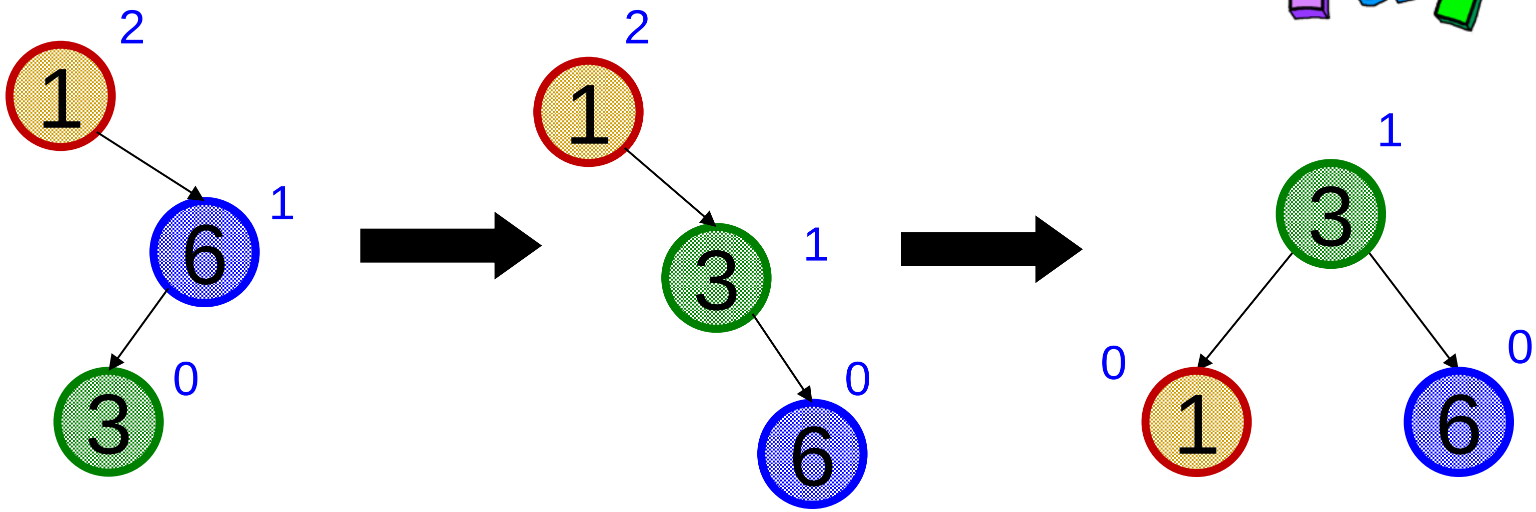
Simple example: **`insert`**(1), **`insert`**(6), **`insert`**(3)

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

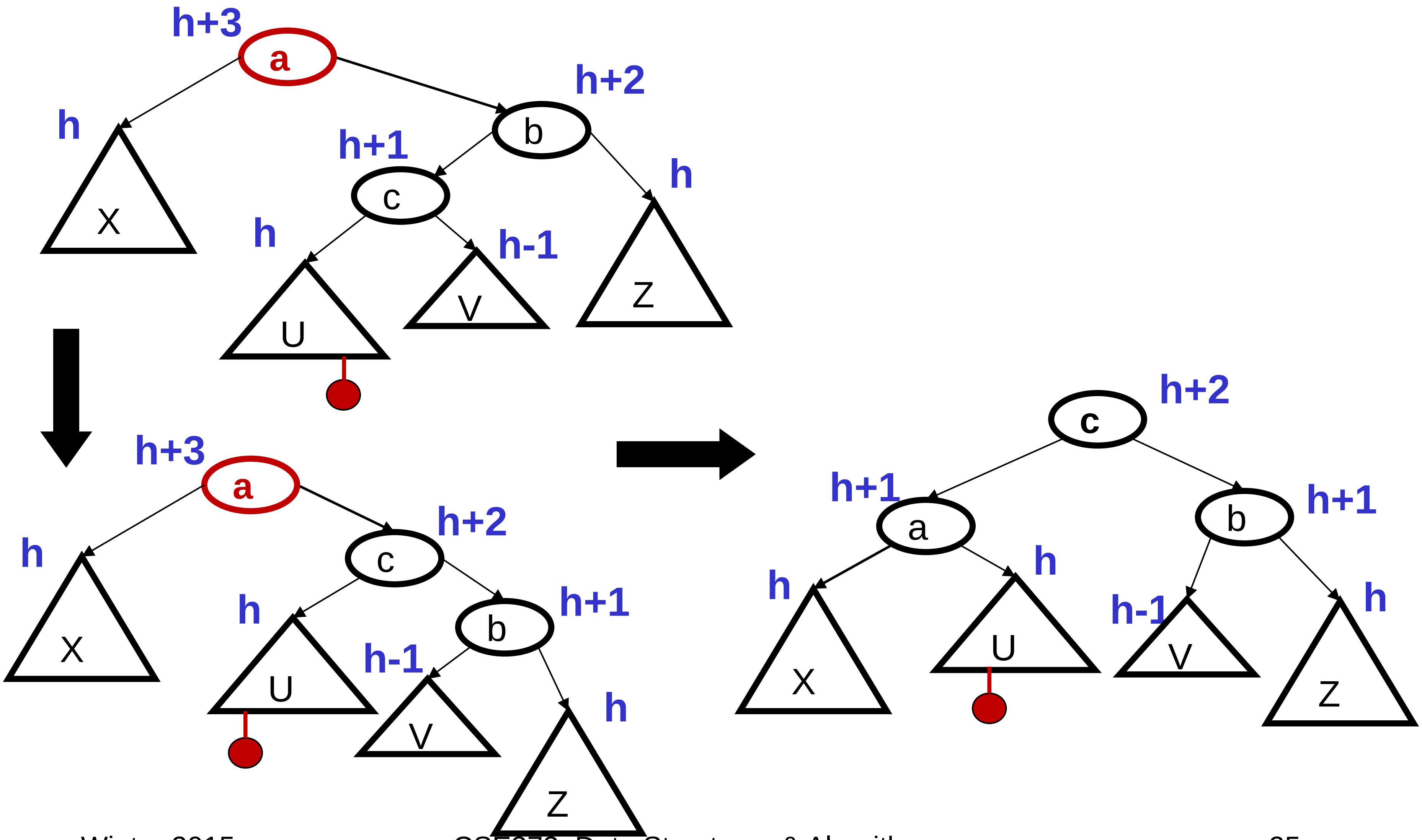


Sometimes two wrongs make a right ☺

- First idea violated the order 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

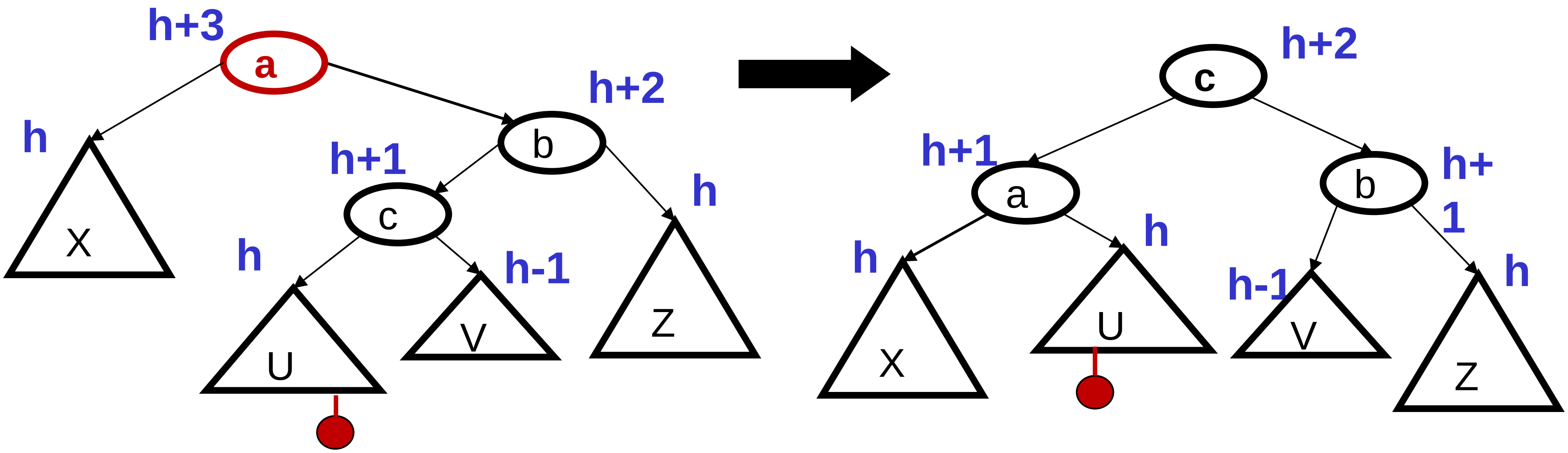


The general right-left case



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:



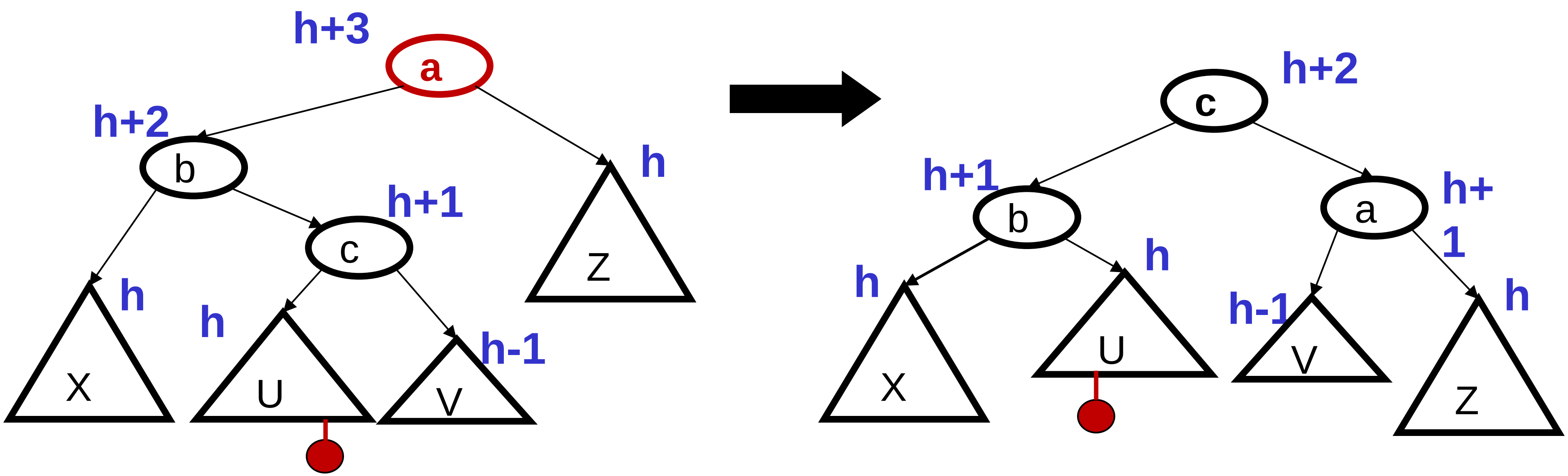
Easier to remember than you may think:

Move c to grandparent's position

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

The last case: left-right

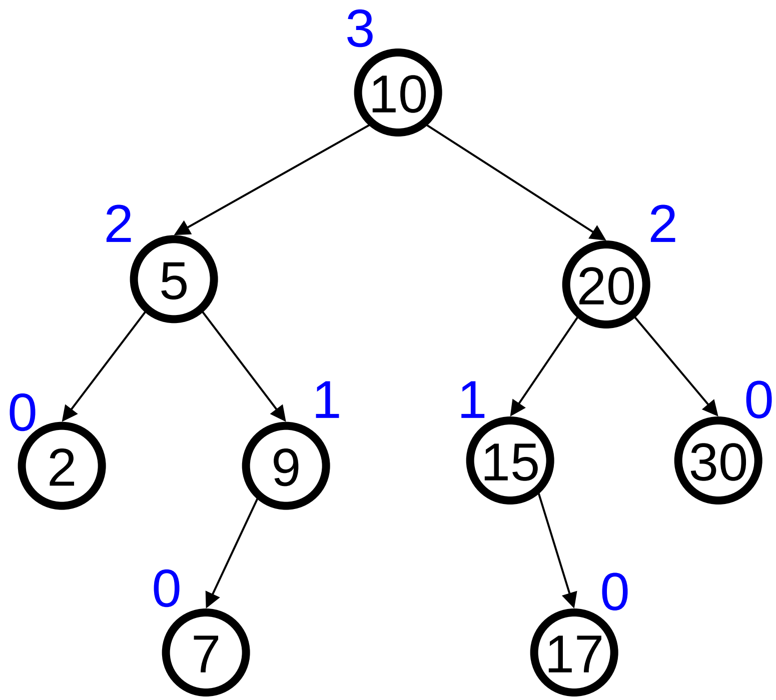
- Mirror image of right-left
  - Again, no new concepts, only new code to write



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
  - Node's left-right grandchild is too tall
  - Node's right-left grandchild is too tall
  - 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

Example

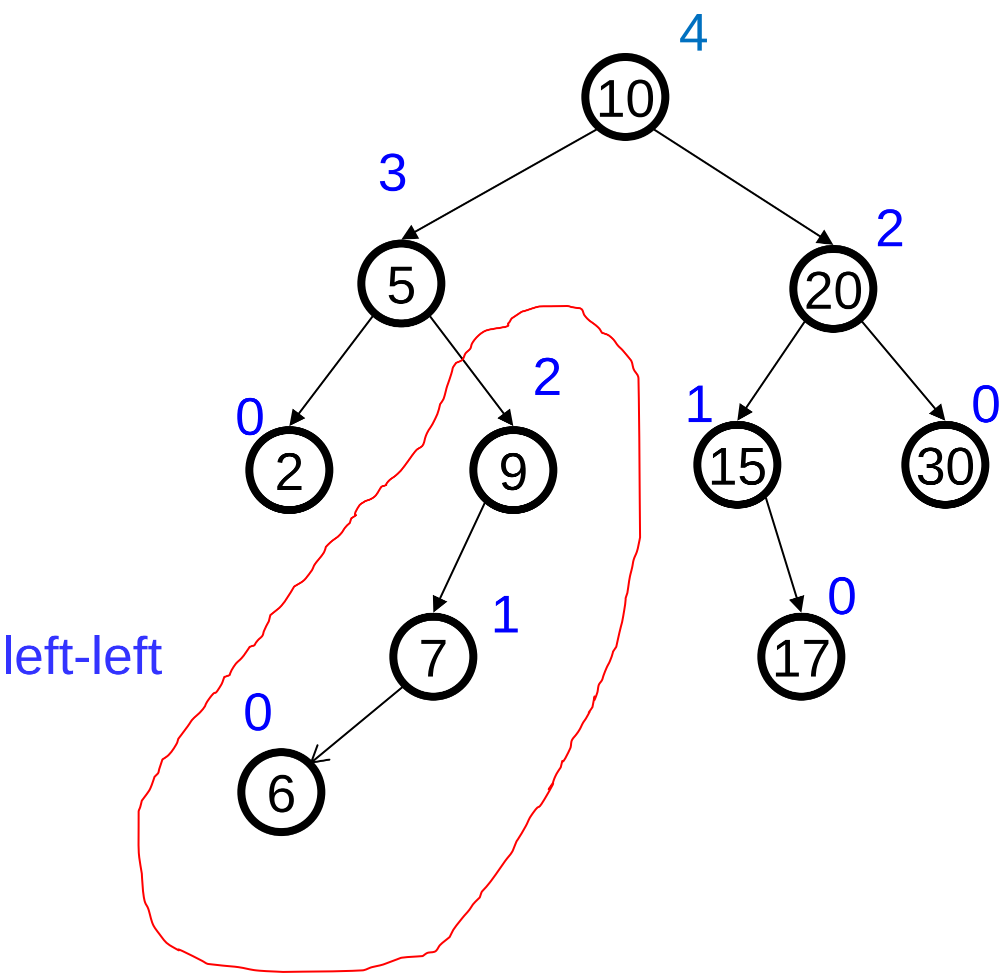


Insert a 6

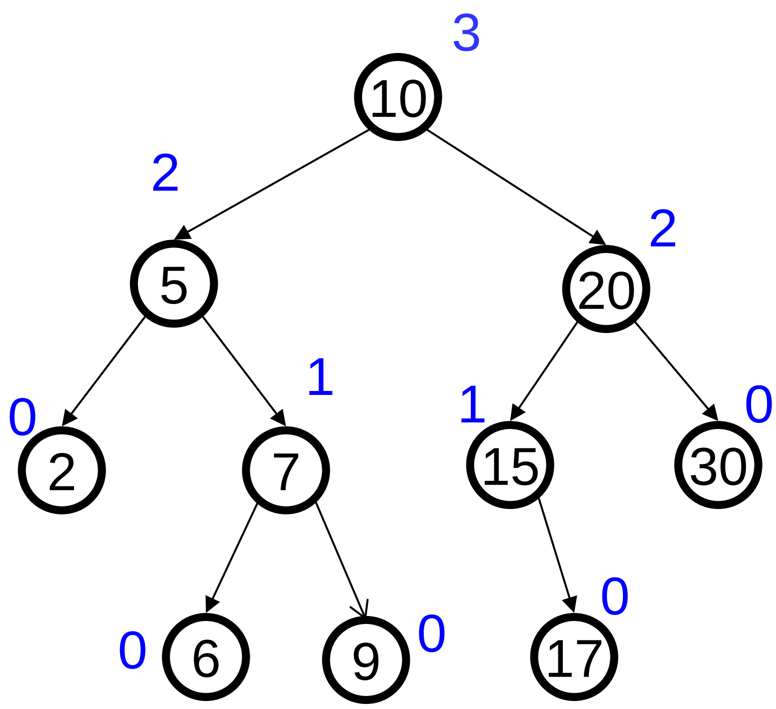
What's the deepest node that is unbalanced?

What's the case?

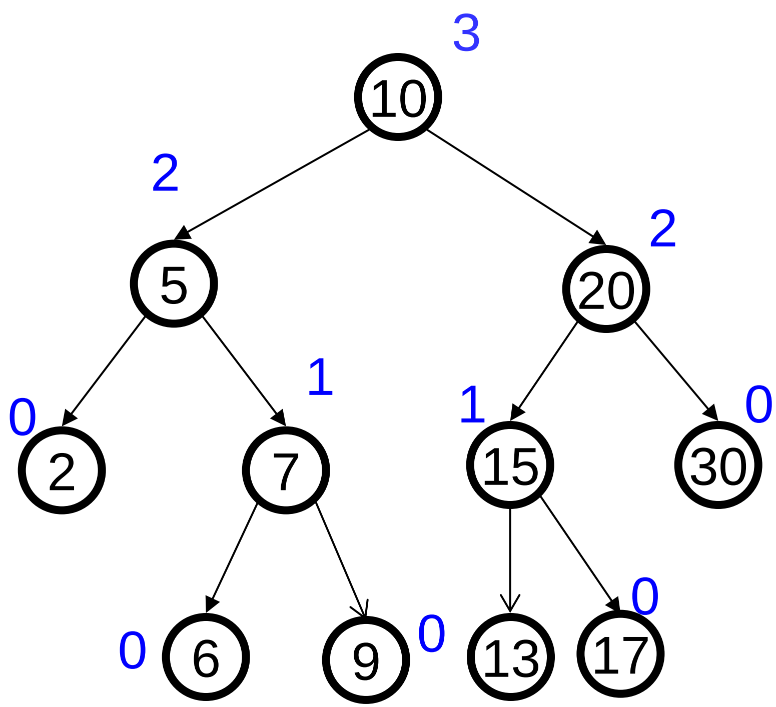
What do we do?



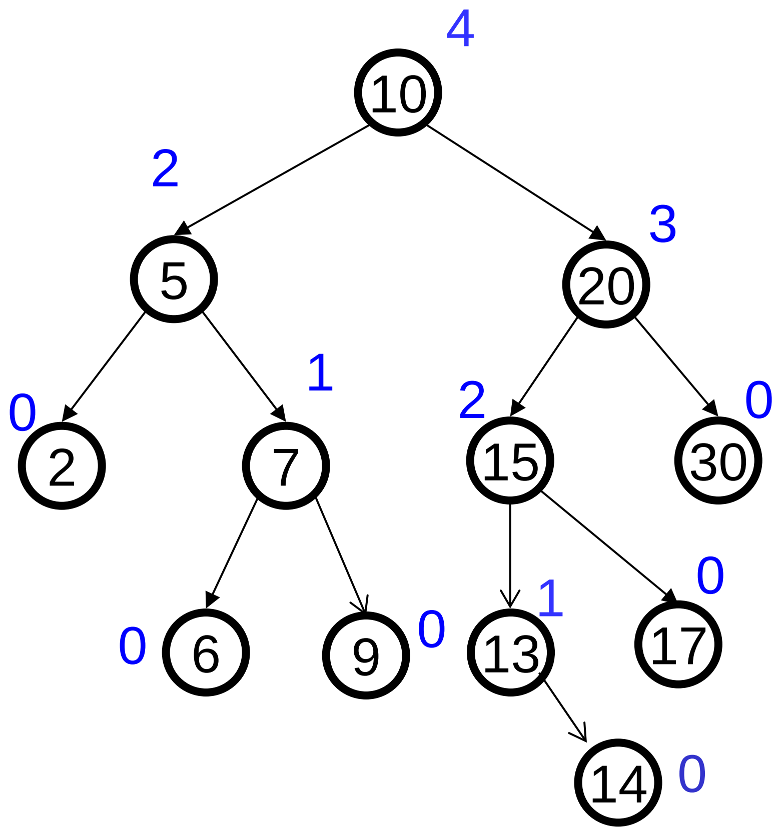
Insert a 6



Insert a 13

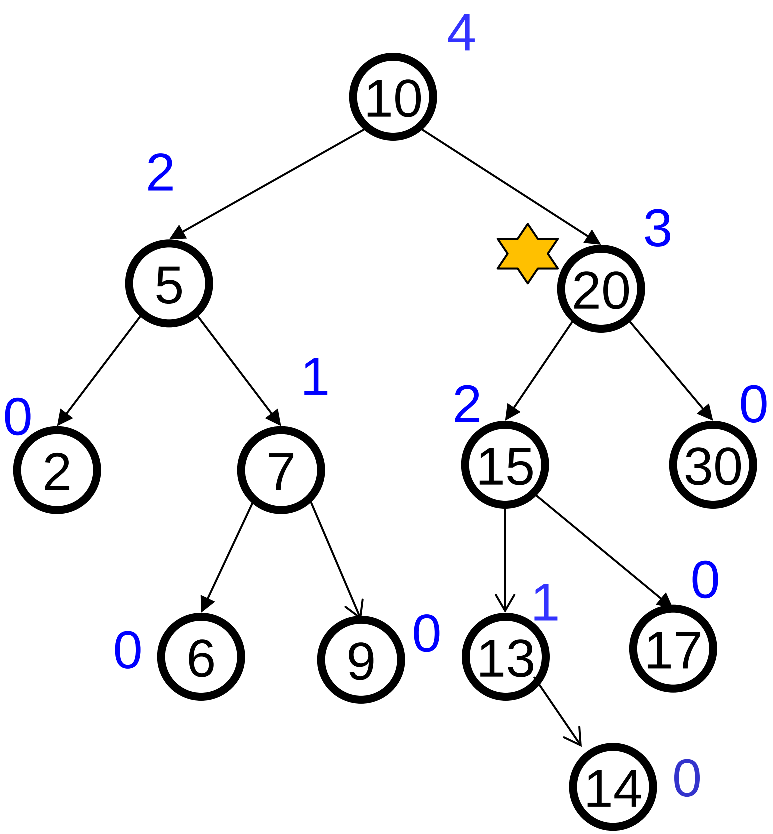


Insert a 14



What is the deepest unbalanced node?

Insert a 14

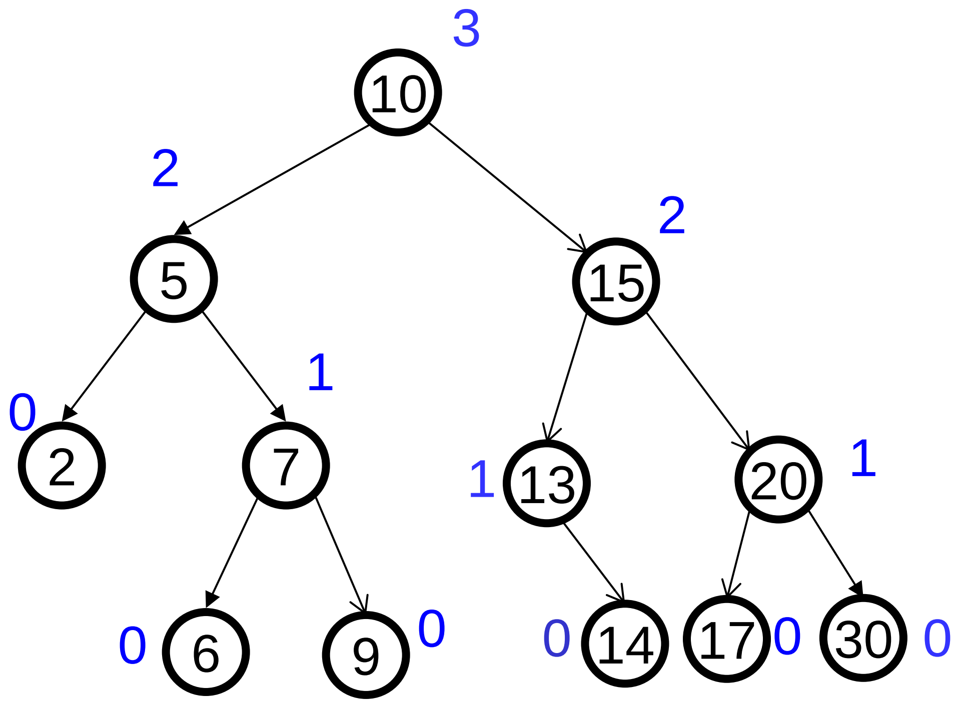


What is the deepest unbalanced node?

Which of the four cases is this?

Still left-left!
Single rotation

Insert a 14



Now 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
  - 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. More 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. If *amortized* (later) logarithmic time is enough, use splay trees (also in the text)