CSE 332: Data Structures and Parallelism

Fall 2022

Richard Anderson

Lecture 7: Адельсóн-Вéльский Лáндис деревья

Lecture 7: AVL Trees

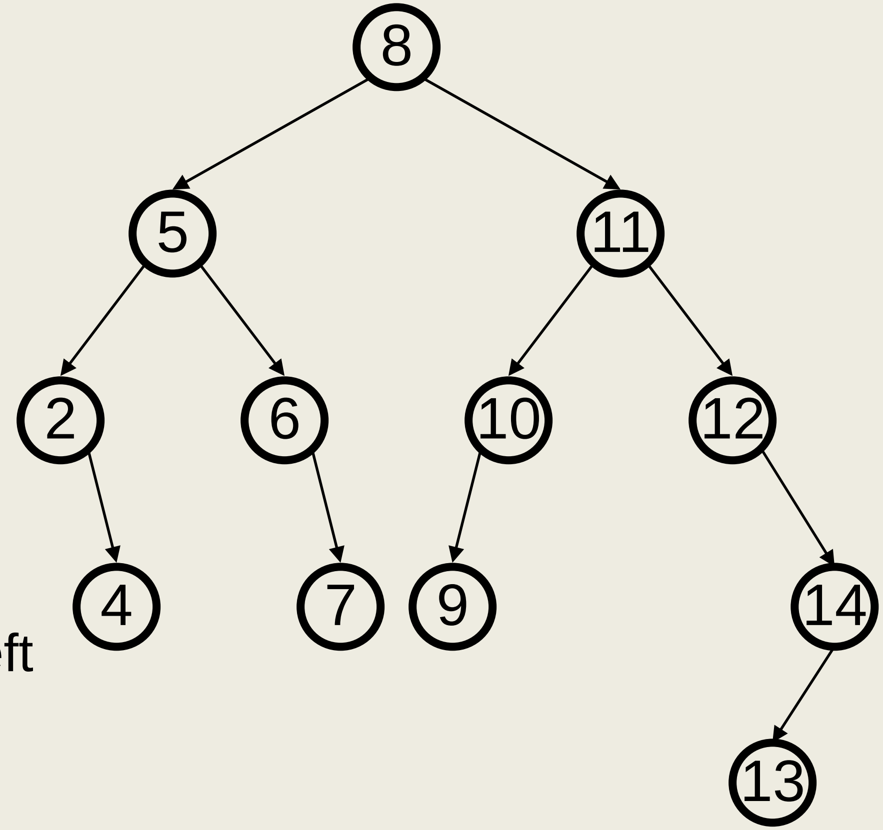
Announcements*

- 10/12: AVL Trees
- 10/14: AVL Trees
- 10/17: B-Trees
- 10/19: B-Trees
- 10/21: Hashing I
- 10/24: Hashing II
- 10/26: Sorting I
- 10/28: Sorting II
- 10/31: Sorting III
- 11/02: Splay Trees
- 11/04: Midterm

*Subject to change

Binary Search Tree Data Structure

- Structural property
  - each node has ≤ 2 children
- Order property
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
- Find / Insert
  - Compare with node value to go left or right
  - Runtime O(height)
- Works great, unless tree is unbalanced



Balanced binary trees

- Binary tree with guarantee on depths of leaves
- O(log n) insert and delete
- Many flavors
  - Red-black trees
  - Self-adjusting binary trees
  - 2-3 trees
  - AVL Trees

- Issues
  - Ensure height is bounded by c log n
  - Maintain this bound on insert and delete

Imbalance is a real issue

$\log_2 1{,}000 \approx 10$

$\log_2 1{,}000{,}000 \approx 20$

$\log_2 1{,}000{,}000{,}000 \approx 30$

Almost sorted input is a realistic case

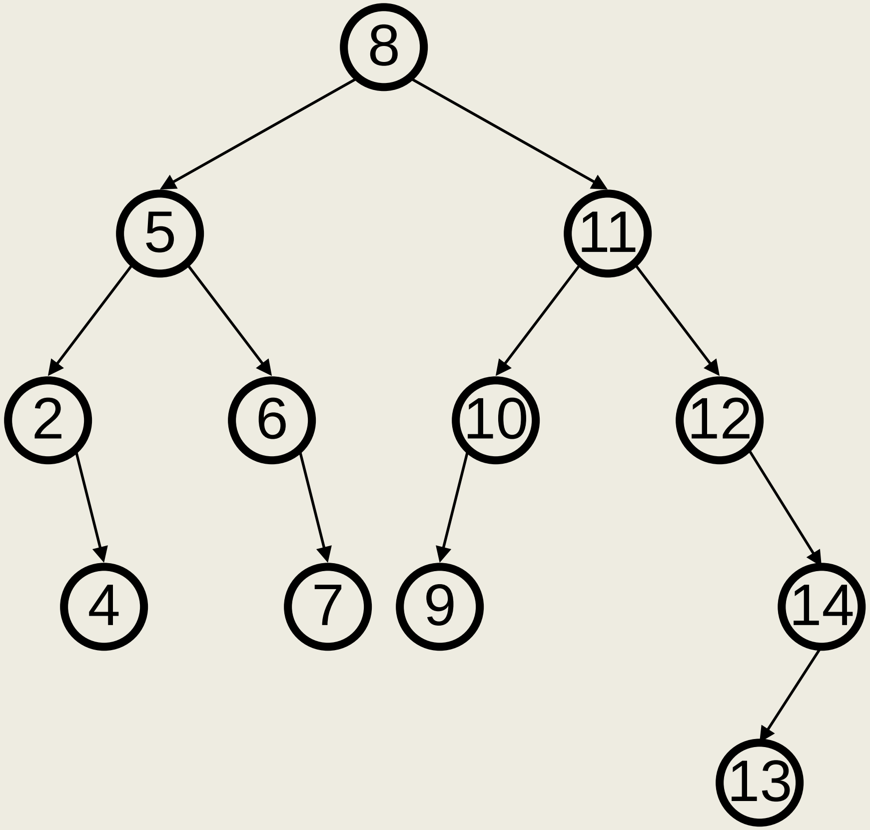
More on binary search trees

What does the following function do:

```
LV(Node root, int low, int high){
        if (root == null)
                return;
        if (low <= root.key)
                LV(root.left, low, high);
        if (low <= root.key && high >= root.key)
                Print(root.key);
        if (high >= root.key)
                LV(root.right, low, high);
}
```

What is the run time?

- Tree of height h
- A total of k nodes are output



AVL Trees

- Developed in 1962 by Soviet mathematicians Georgy Adelson-Velsky and Evgeny Landis (Адельсо́н-Ве́льский, Ла́ндис)
- Structural property on tree guarantees depth O(log n)
- Rebalance operation to ensure property
- Practical
- First published balanced search tree

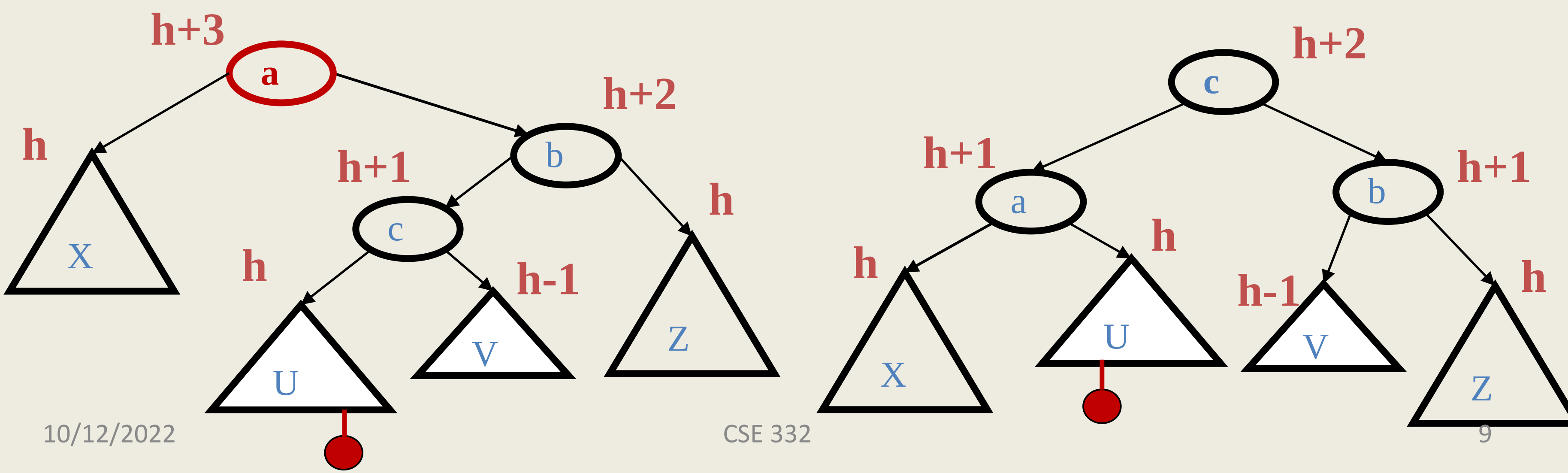
AN ALGORITHM FOR THE ORGANIZATION OF INFORMATION

G. M. ADEL'SON-VEL'SKIĬ AND E. M. LANDIS

An algorithm is proposed in which both the search and the recording are carried out in $C \lg N$ operations, where N is the number of information elements which have entered at a given moment.

AVL Tree overview

- Balance condition
- Depth bound
- Rotations to rebalance the tree



The AVL Tree Data Structure

Structural properties

1. Binary tree property
2. Balance:

   left.height – right.height

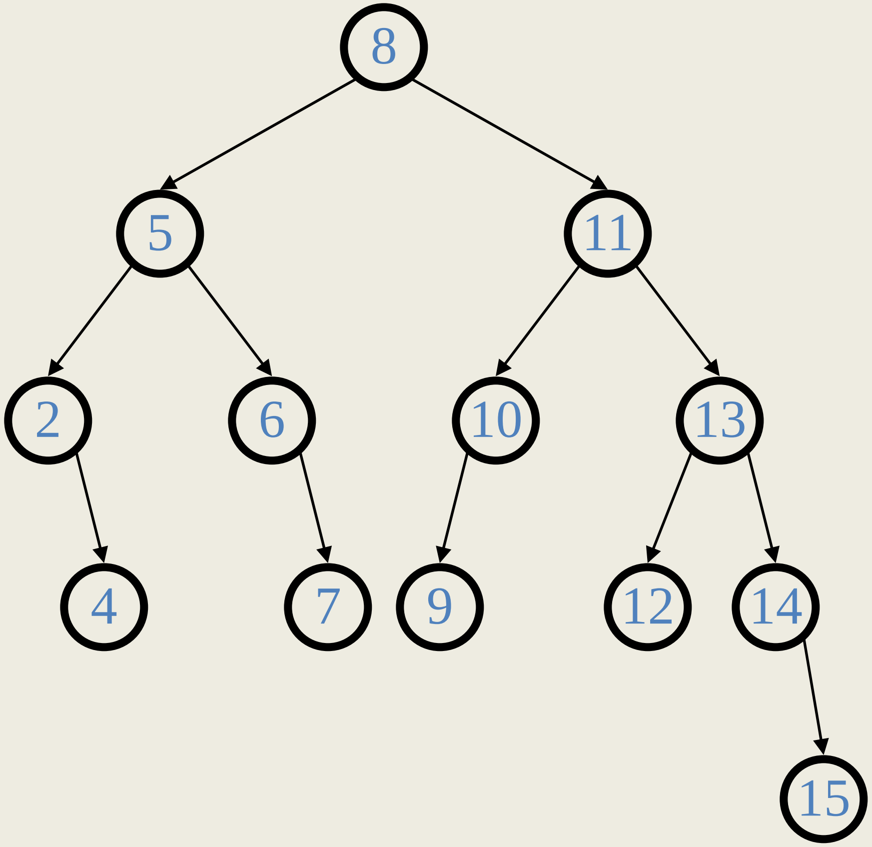
3. Balance property: balance of every node is between -1 and 1

Result:

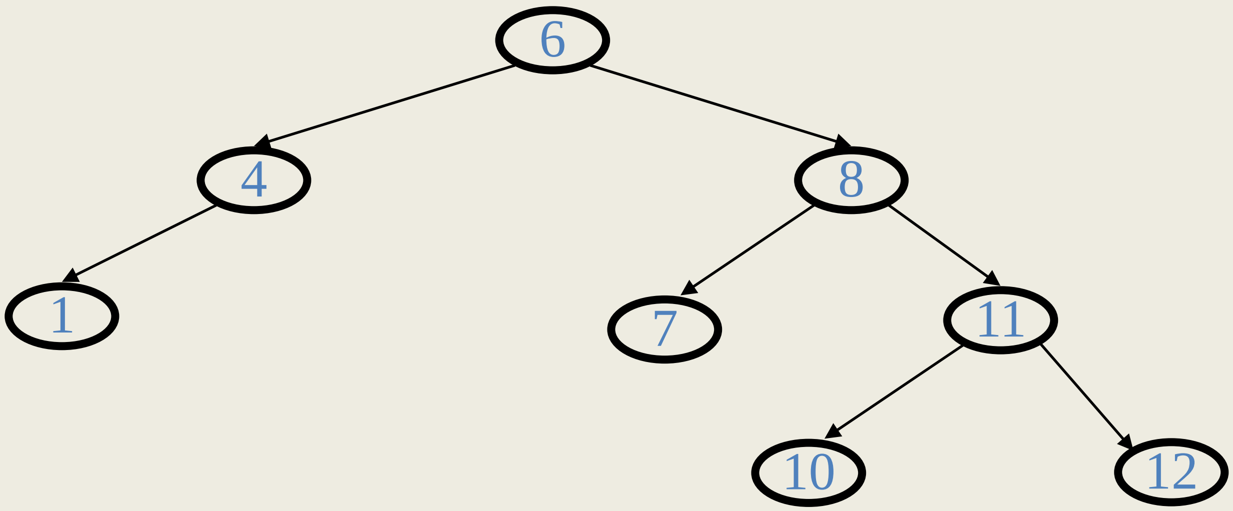
Worst-case depth is O(log *n*)

Ordering property

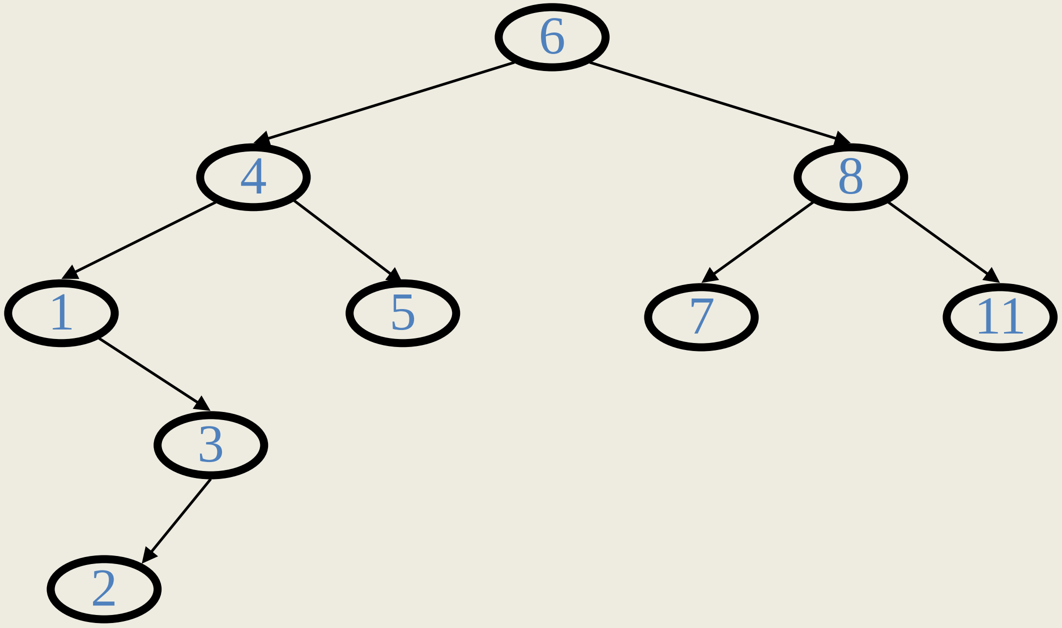
– Same as for BST



An AVL tree?



An AVL tree?

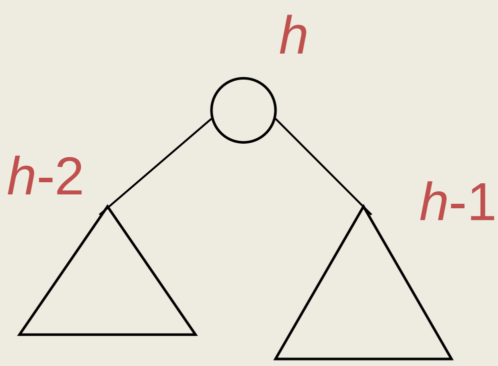


Height of an AVL Tree?

Using the AVL balance property, we can determine the minimum number of nodes in an AVL tree of height *h*

Let **S**(*h*)be the minimum # of nodes in an AVL tree of height *h*, then:

S(*h*) = **S**(*h*-1) + **S**(*h*-2) + 1
where **S**(0) = 1 and **S**(1) = 2



Solution of Recurrence: **S**(*h*) $\approx 1.62^h$

$$n \geq \phi^h \Longrightarrow \log_\phi n \geq h$$

Let S(h)be the minimum # of nodes in an AVL tree of height h, then:

$S(h) = S(h-1) + S(h-2) + 1$ where $S(0) = 1$ and $S(1) = 2$

h	*Minimal AVL Tree*	*S(h)*

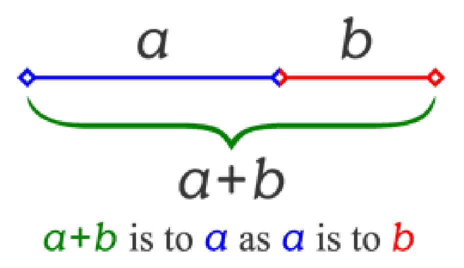
A simpler bound: $S_k \geq 2^{k/2}$

$S_0 = 1$, $S_1 = 2$, $S_k = 1 + S_{k-1} + S_{k-2}$

1, 2, 4, 7, 12, 20, 33, 54, . . .

Growing faster than the Fibonacci sequence

Observation: $S_k \geq 2\, S_{k-2}$



The Golden Ratio

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

This is a special number

- *Golden ratio*: If (a+b)/a = a/b, then a = ϕb

- We will need one special arithmetic fact about ϕ :

ϕ^2 = $((1+5^{1/2})/2)^2$
= $(1 + 2*5^{1/2} + 5)/4$
= $(6 + 2*5^{1/2})/4$
= $(3 + 5^{1/2})/2$
= $1 + (1 + 5^{1/2})/2$
= $1 + \phi$

$S(0)=1, S(1)=2, S(2)=4$

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

The Proof

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

Proof: By induction on h

Base cases:

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

Inductive case ($k > 1$):

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

$S(k+1) = 1 + S(k) + S(k-1)$	by definition of S
$> 1 + \phi^k - 1 + \phi^{k-1} - 1$	by induction
$= \phi^k + \phi^{k-1} - 1$	
$= \phi^{k-1}(\phi + 1) - 1$	by arithmetic (factor ϕ^{k-1})
$= \phi^{k-1}\phi^2 - 1$	by special property of ϕ
$= \phi^{k+1} - 1$	

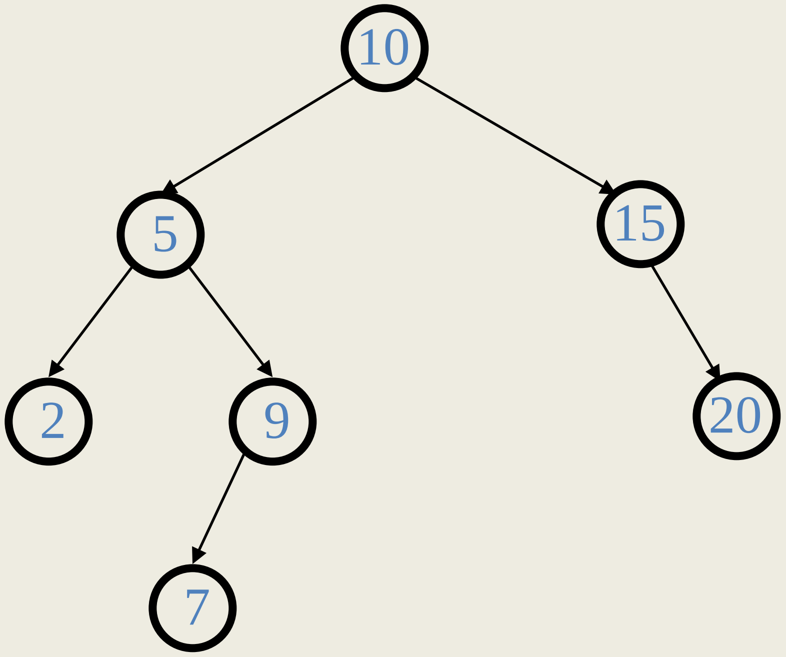
Good news

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

- Recall logarithms of different bases > 1 differ by only a constant factor

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

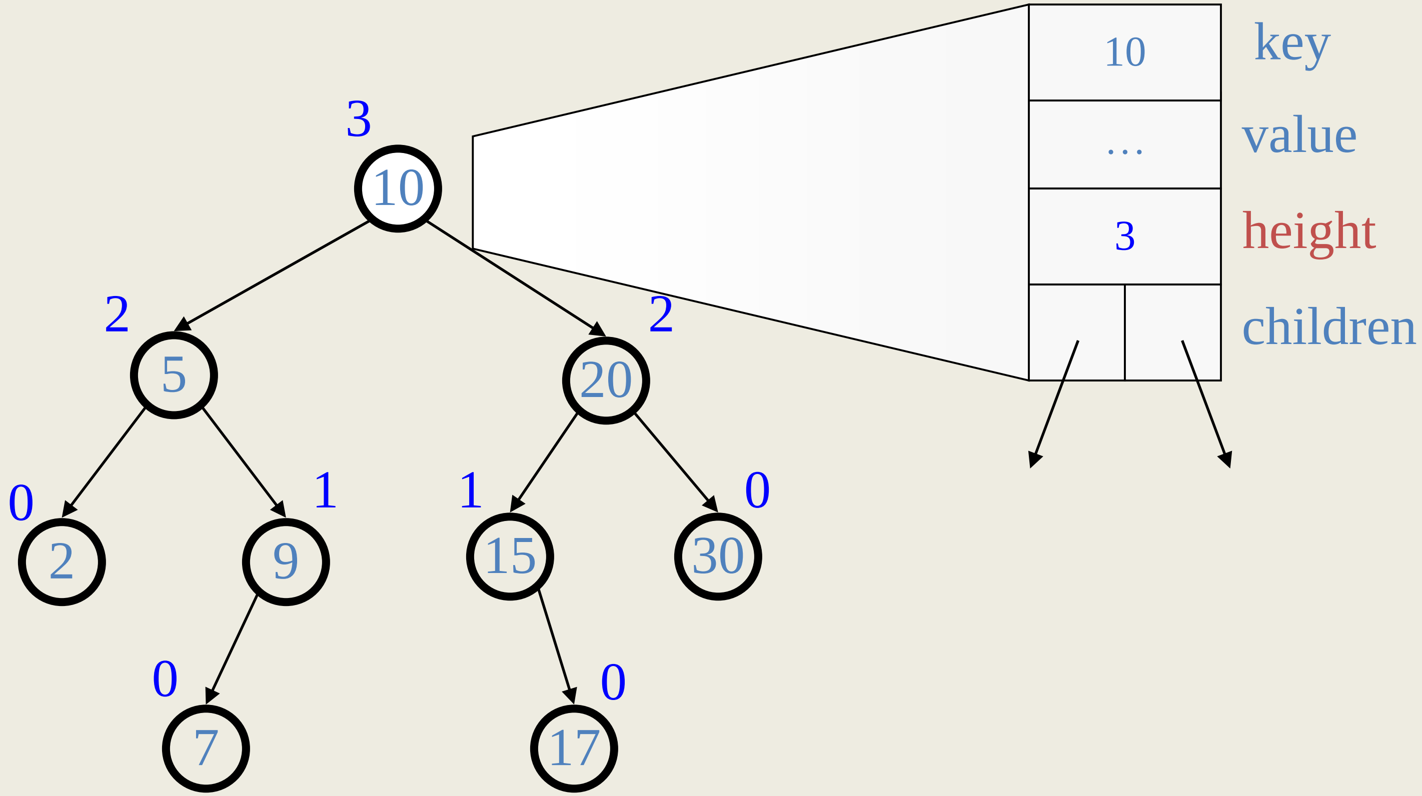
1. Track balance
2. Detect imbalance
3. Restore balance



Is this AVL tree balanced?

How about after insert(30)?

An AVL Tree



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 (next lecture, maybe)

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

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

Facts that an implementation can ignore:

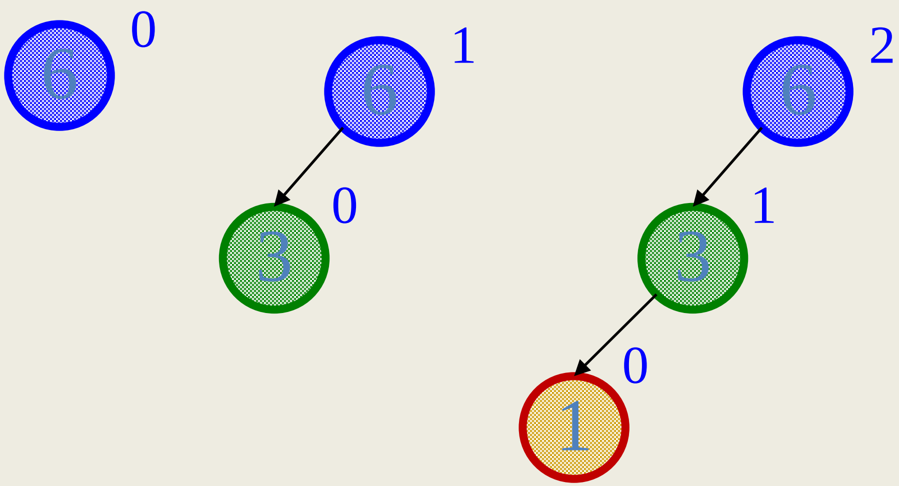
- There must be a deepest element that is imbalanced after the insert (all descendants still balanced)
- After rebalancing this deepest node, every node is balanced
- So at most one node needs to be rebalanced

Case #1: Example

Insert(6)

Insert(3)

Insert(1)



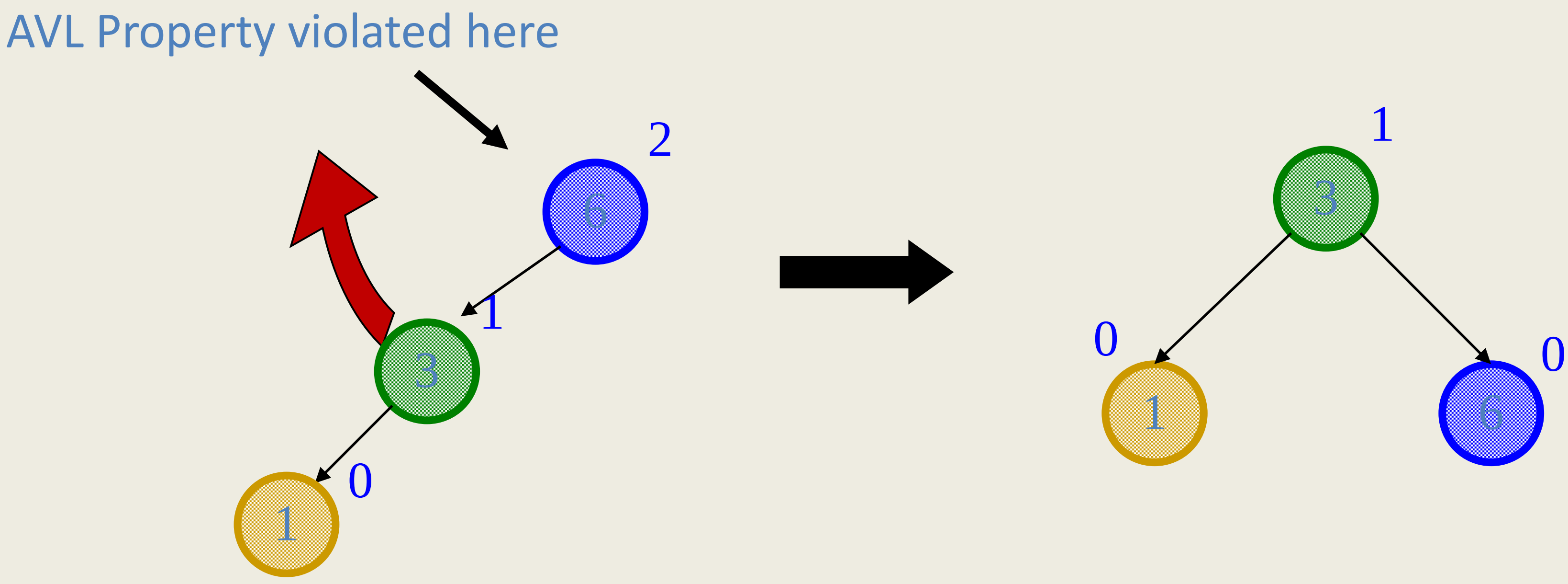
Third insertion violates balance property

- happens to be at the root

What is the only way to fix this?

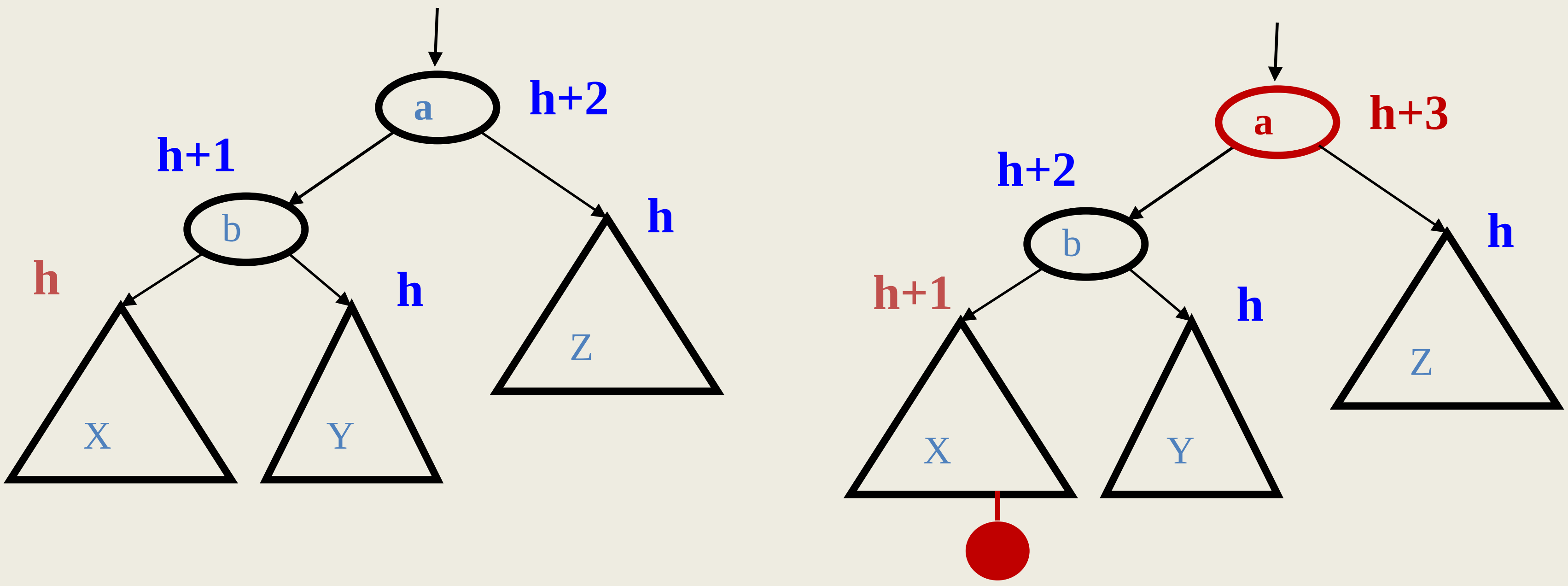
Fix: Apply "Single Rotation"

- *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)



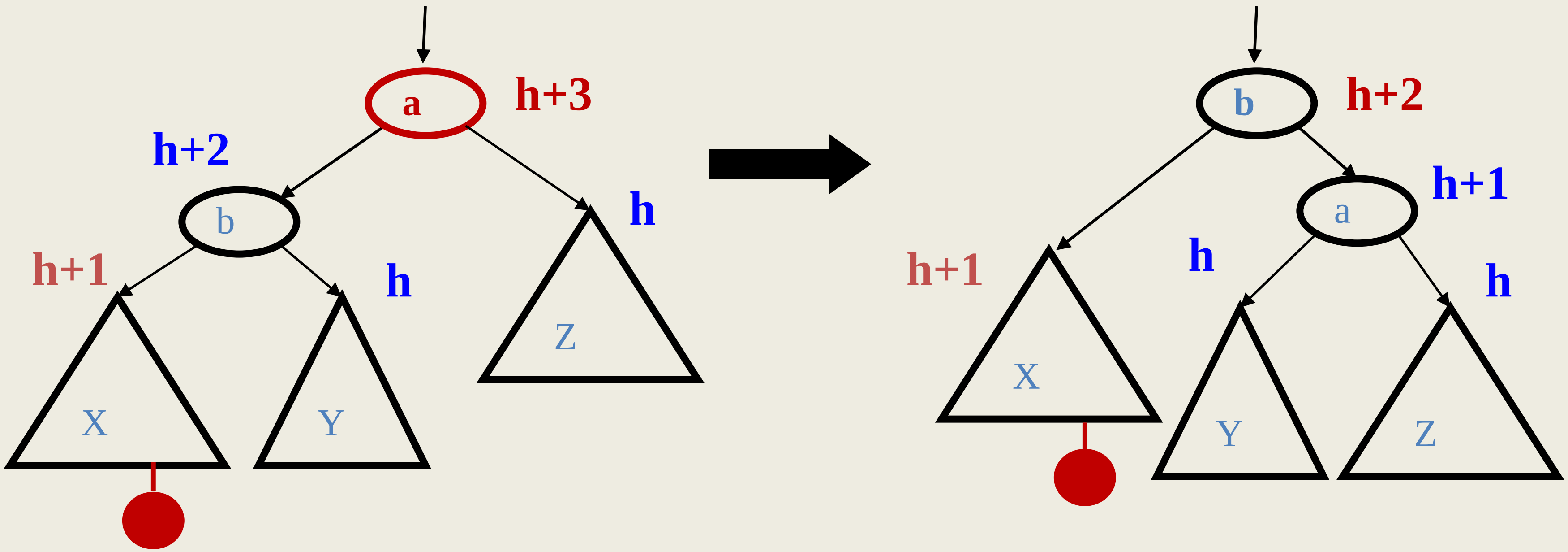
The example generalized

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



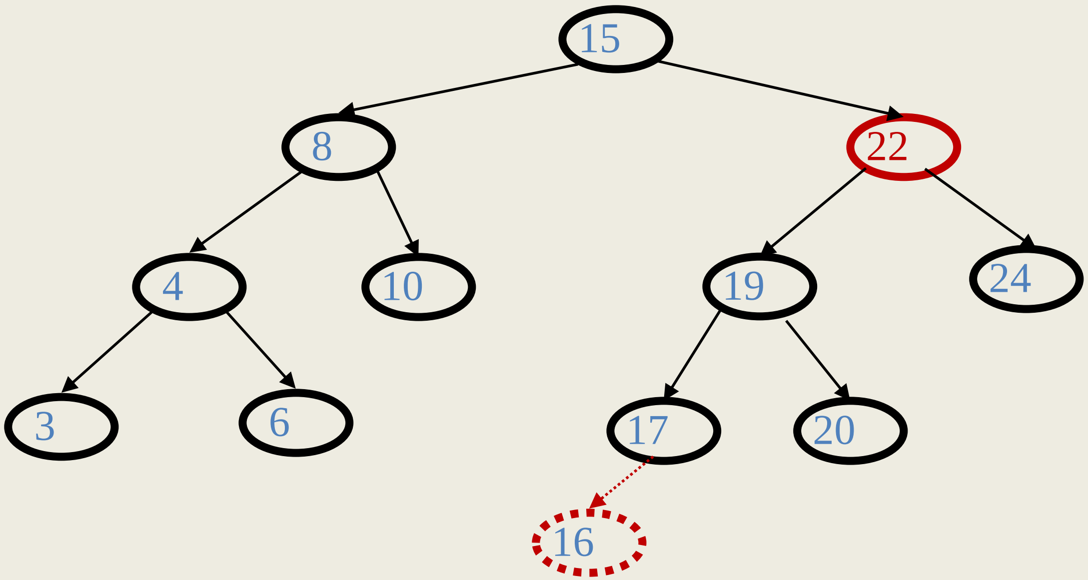
The general left-left case

- 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

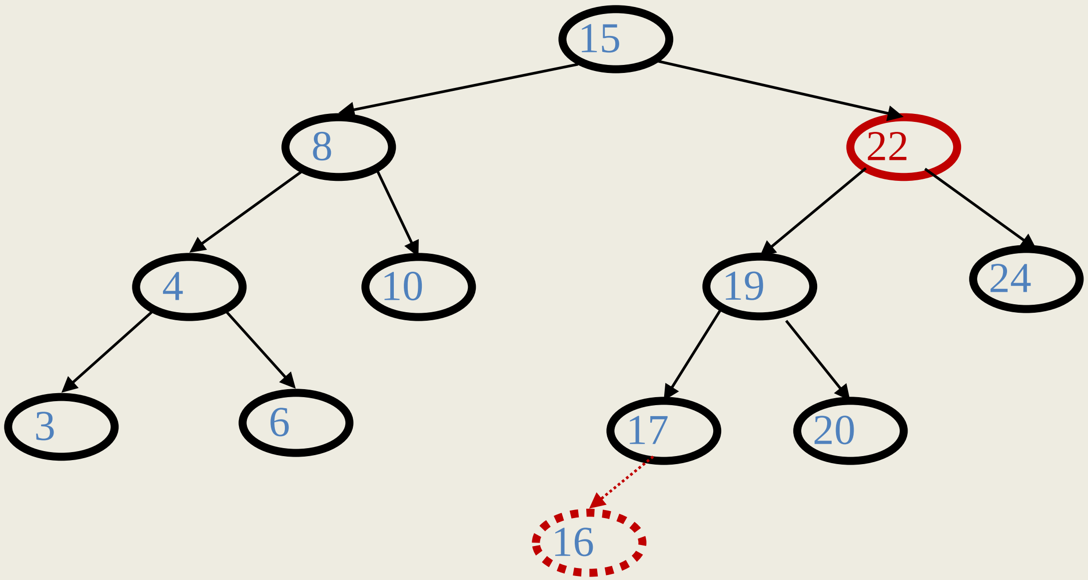


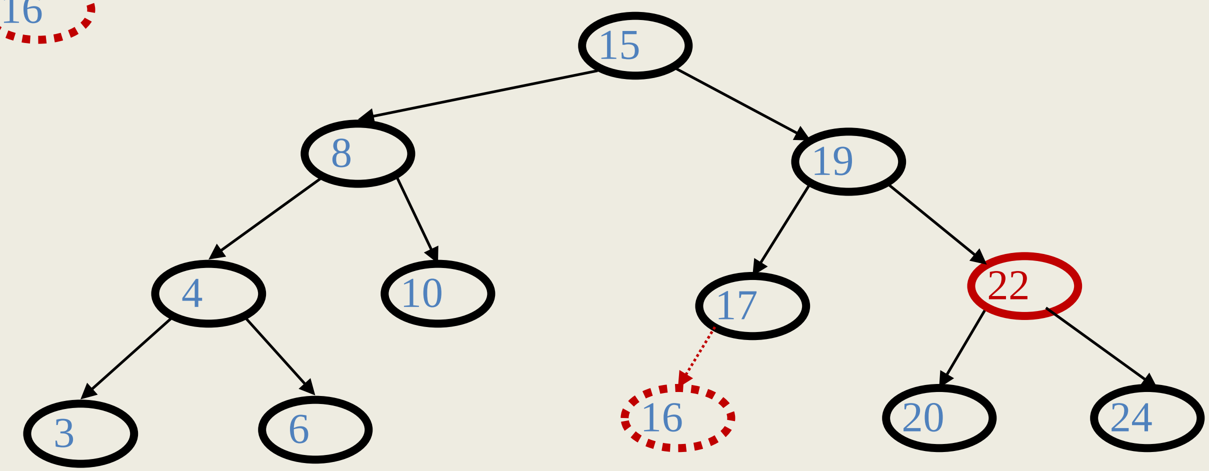
- A single rotation restores balance at the node
  - To same height as before insertion, so ancestors now balanced

Another example: `insert(16)`



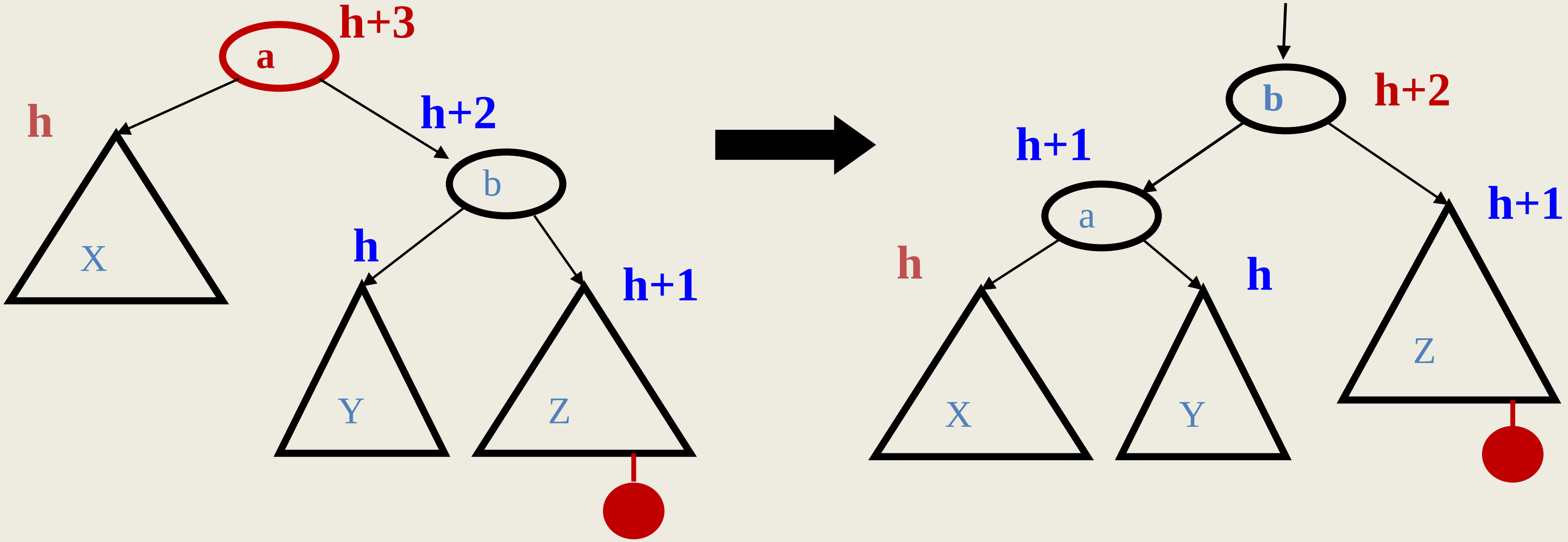
Another example: `insert(16)`





The general right-right case

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

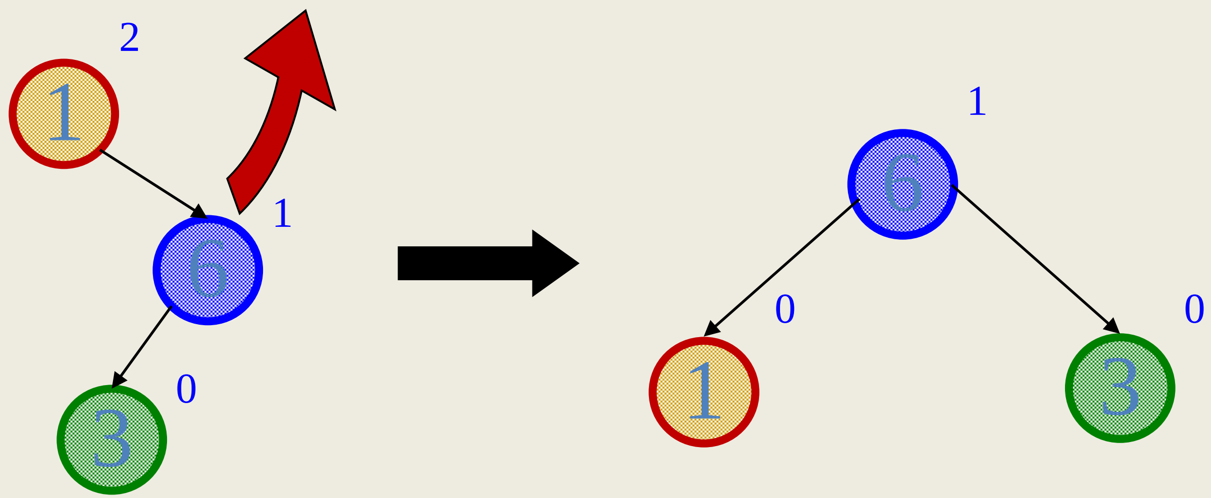


Two cases to go

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

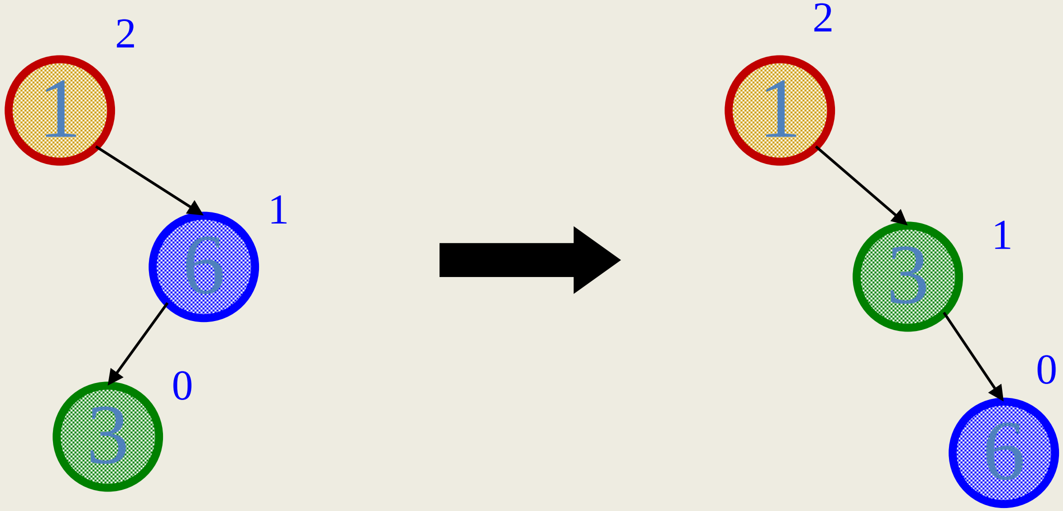


Two cases to go

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

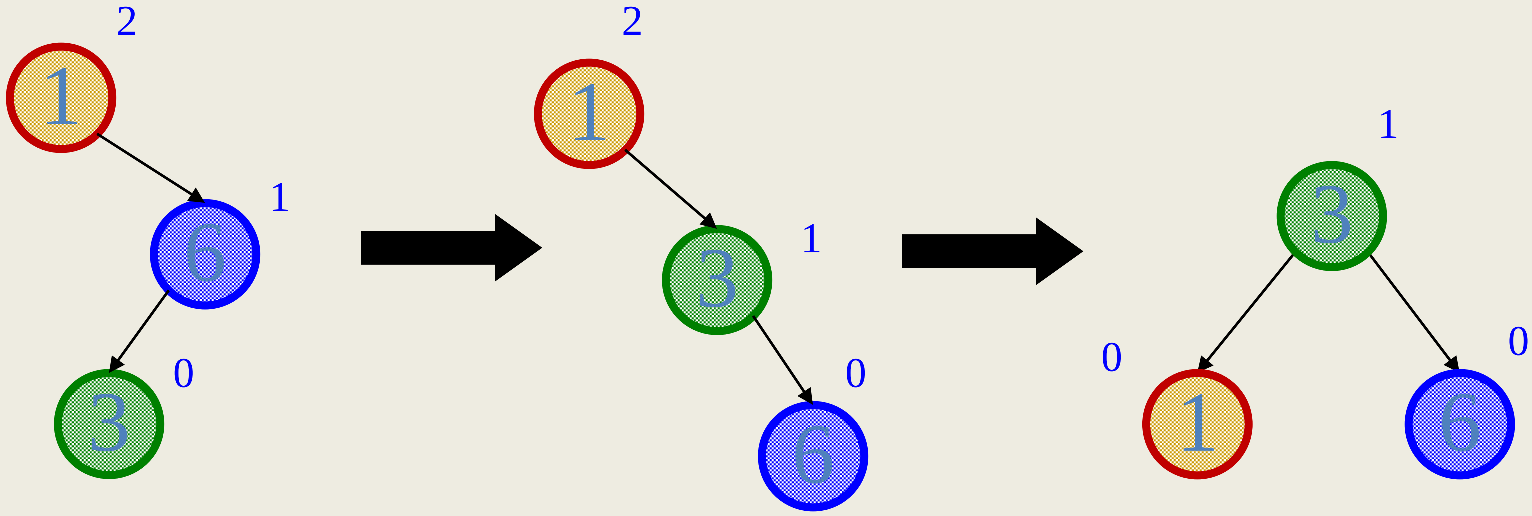
Simple example: **`insert`**(1), **`insert`**(6), **`insert`**(3)

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

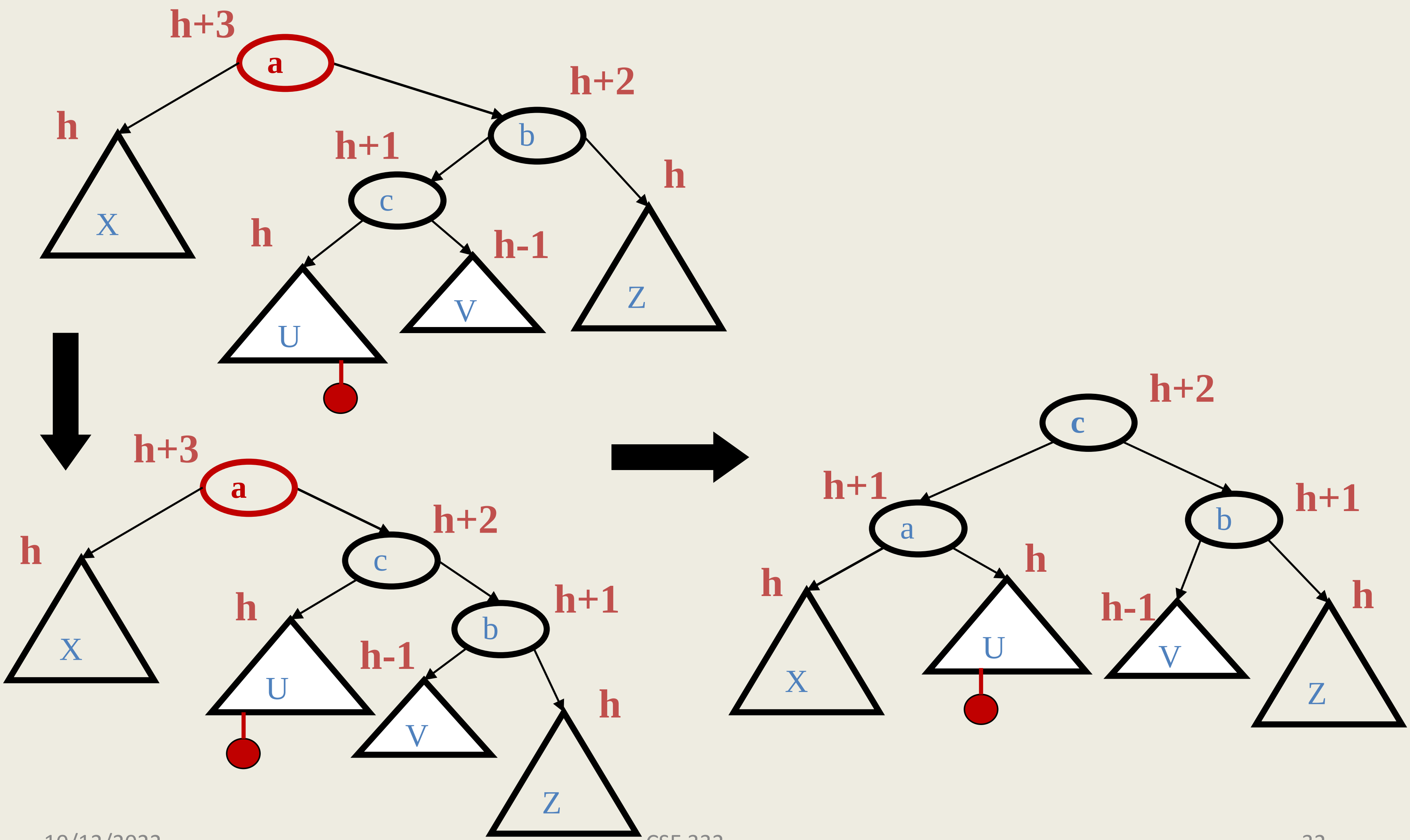


Sometimes two wrongs make a right ☺

- 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

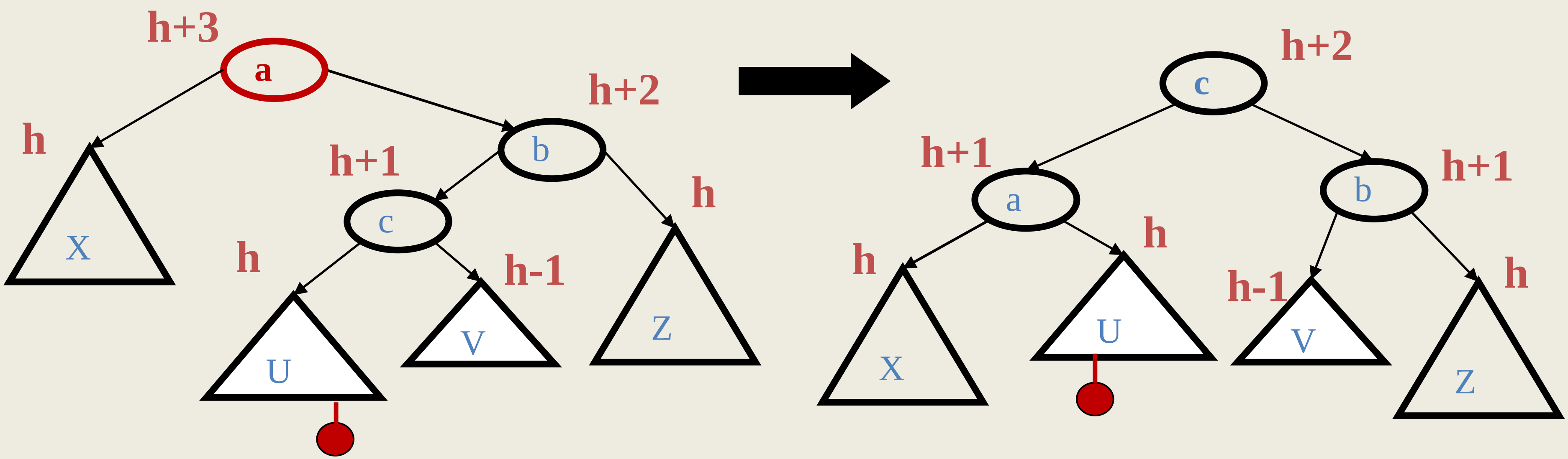


The general right-left case



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:



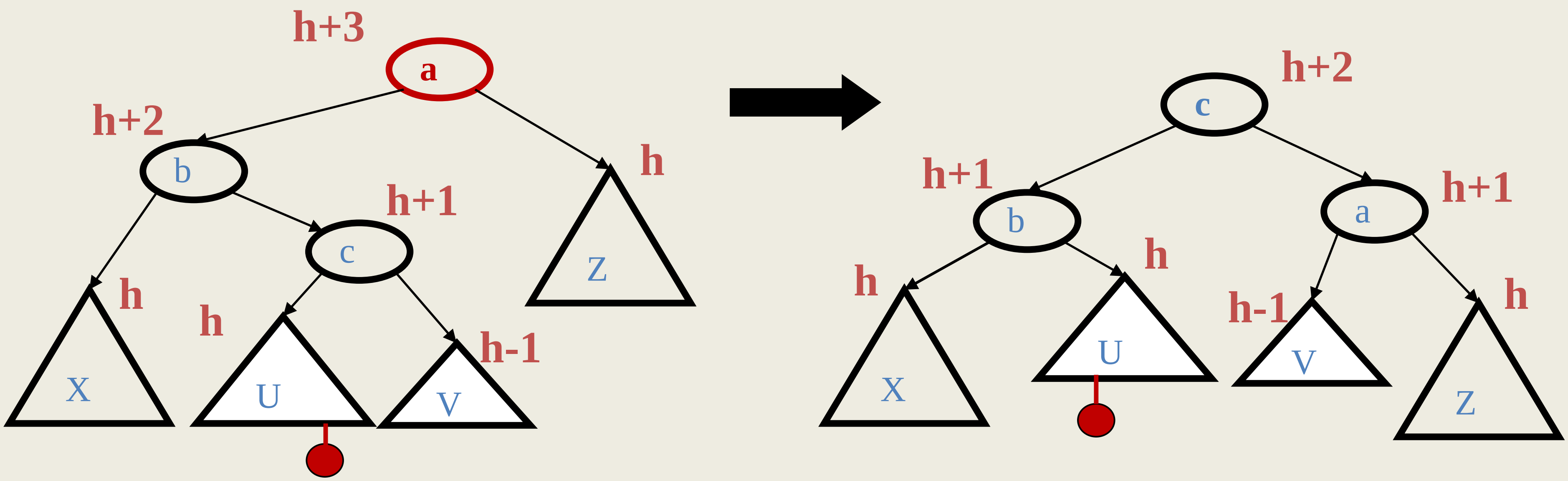
Easier to remember than you may think:

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