



## CSE332: Data Abstractions

### Lecture 7: AVL Trees

Dan Grossman  
Spring 2010

## The AVL Tree Data Structure

Structural properties

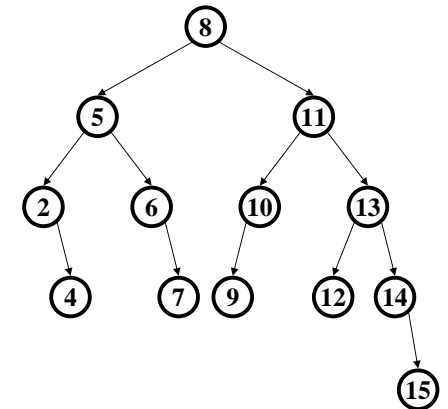
1. Binary tree property
2. Balance property:  
balance of every node is  
between -1 and 1

Result:

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

Ordering property

- Same as for BST

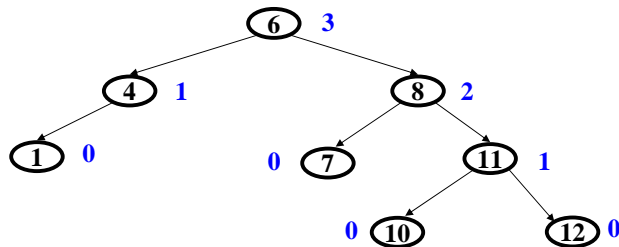


Spring 2010

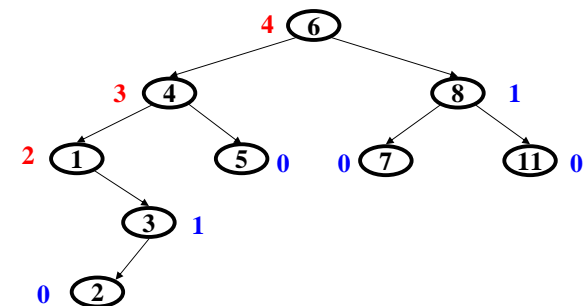
CSE332: Data Abstractions

2

*An AVL tree?*



*An AVL tree?*

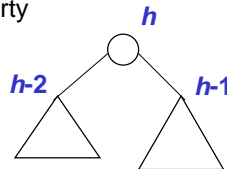


## The shallowness bound

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

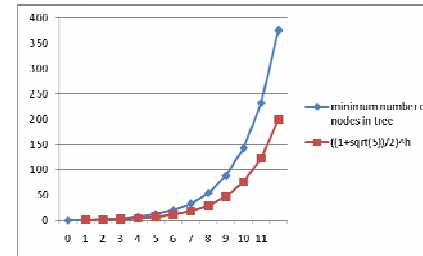


- Step 2: Show this recurrence grows really fast
  - Similar to Fibonacci numbers
  - Can prove for all  $h$ ,  $S(h) > \phi^h - 1$  where  $\phi$  is the golden ratio,  $(1+\sqrt{5})/2$ , about 1.62
  - Growing faster than  $1.6^h$  is “plenty” exponential

5

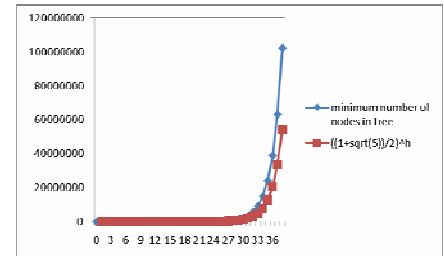
## Before we prove it

- 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



Spring 2010

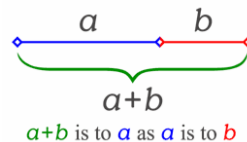
CSE332: Data Abstractions



6

## The Golden Ratio

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



This is a special number

- Aside: Since the Renaissance, many artists and architects have proportioned their work (e.g., length:height) to approximate the **golden ratio**: If  $(a+b)/a = a/b$ , then  $a = \phi b$
- We will need one special arithmetic fact about  $\phi$ :

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

7

## The proof

$$\begin{aligned} S(-1) &= 0, S(0) = 1, S(1) = 2 \\ \text{For } h \geq 2, S(h) &= 1 + S(h-1) + S(h-2) \end{aligned}$$

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

$$\begin{aligned} S(k+1) &= 1 + S(k) + S(k-1) && \text{by definition of } S \\ &> 1 + \phi^k - 1 + \phi^{k-1} - 1 && \text{by induction} \\ &= \phi^k + \phi^{k-1} - 1 && \text{by arithmetic } (1-1=0) \\ &= \phi^{k-1}(\phi + 1) - 1 && \text{by arithmetic (factor } \phi^{k-1}) \\ &= \phi^{k-1}\phi^2 - 1 && \text{by special property of } \phi \\ &= \phi^{k+1} - 1 && \text{by arithmetic (add exponents)} \end{aligned}$$

Spring 2010

CSE332: Data Abstractions

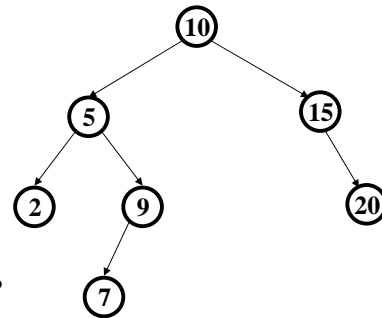
8

## Good news

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

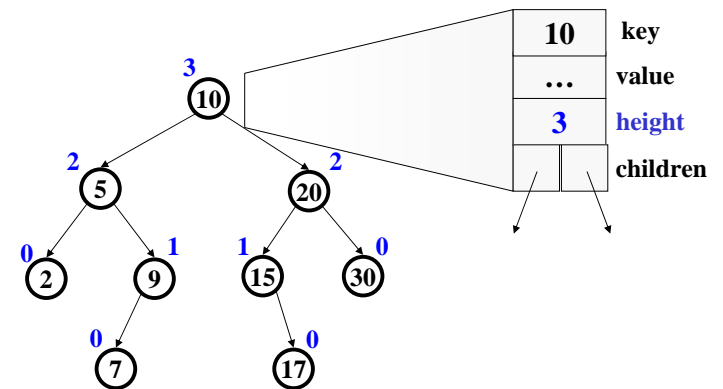
But as we insert and delete elements, we need to:

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 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, 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 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

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

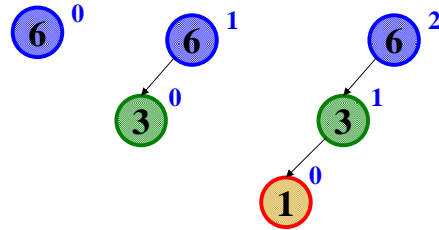
## Case #1: Example

Insert(6)  
Insert(3)  
Insert(1)

Third insertion violates  
balance property

- happens to be at  
the root

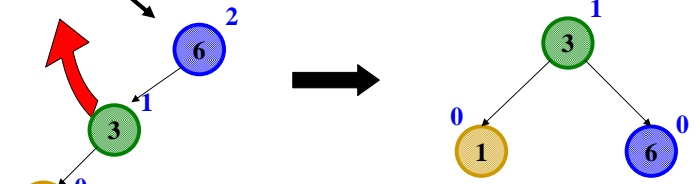
What is the only way to  
fix this?



## 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)

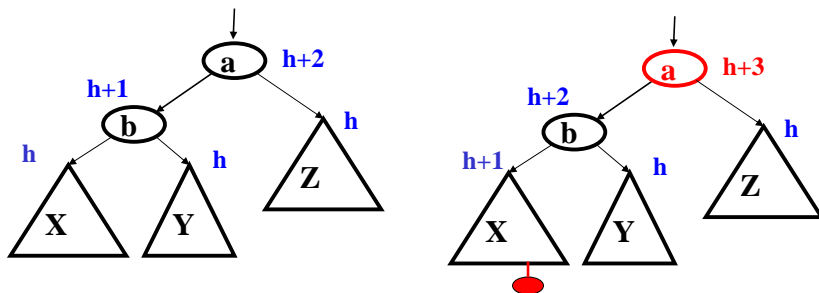
AVL Property violated  
here



**Intuition: 3 must become root**  
**new-parent-height = old-parent-height-before-insert**

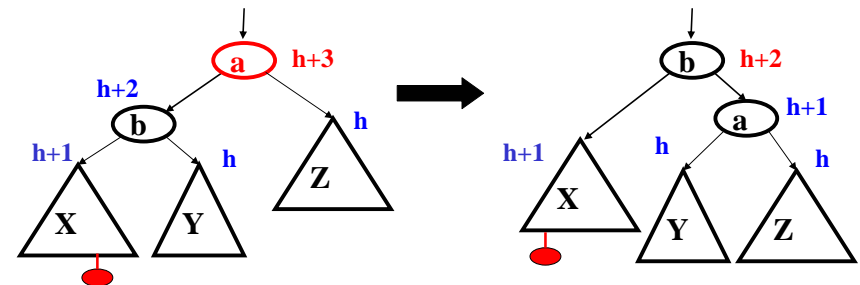
## The example generalized

- 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



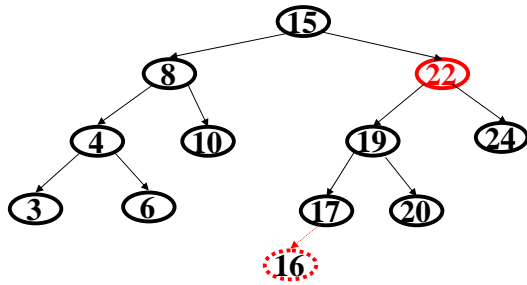
## The general left-left case

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

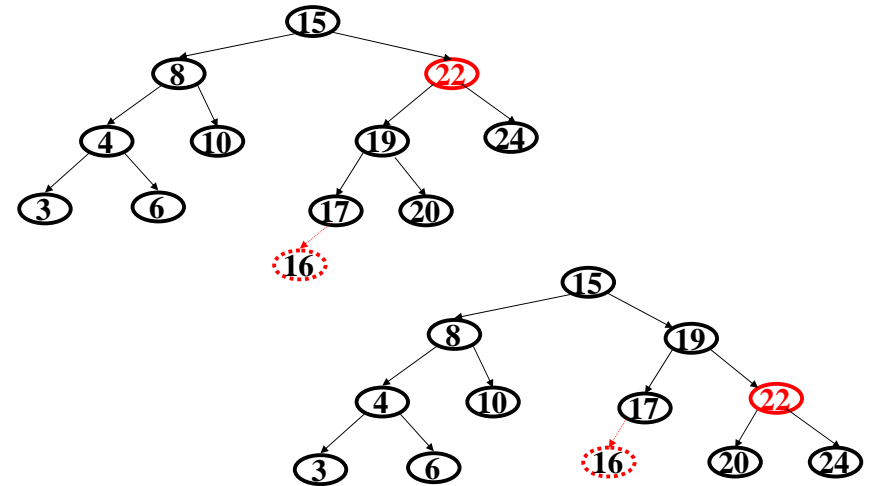


- A single rotation restores balance at the node
  - To same height as before insertion (so ancestors now balanced)

### Another example: insert(16)

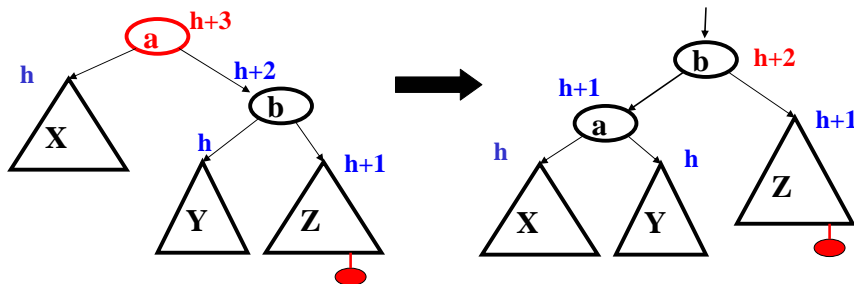


### Another example: insert(16)



### The general right-right case

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

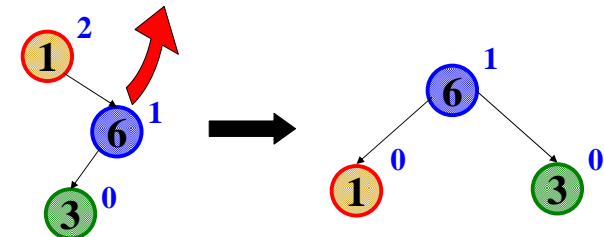


### Two cases to go

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

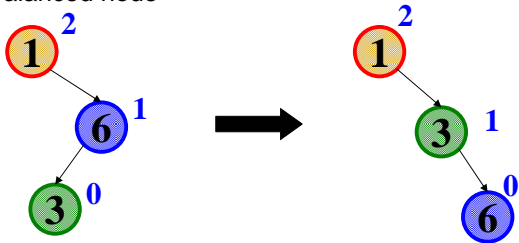


## Two cases to go

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

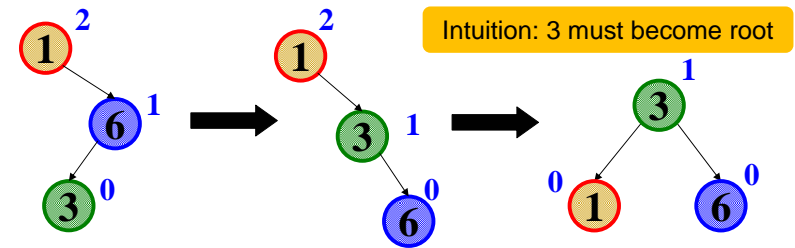
Simple example: `insert(1)`, `insert(6)`, `insert(3)`

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

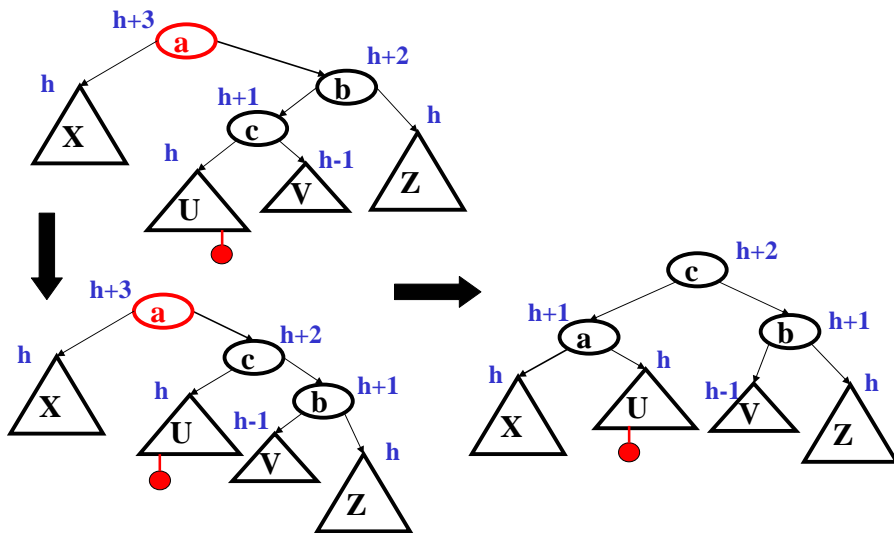


## Sometimes two wrongs make a right ☺

- 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

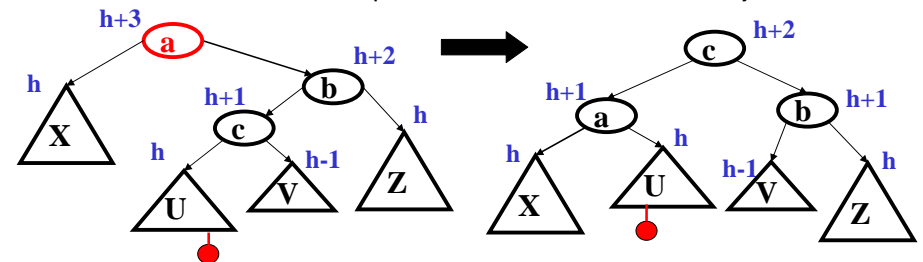


## The general right-left case



## 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:

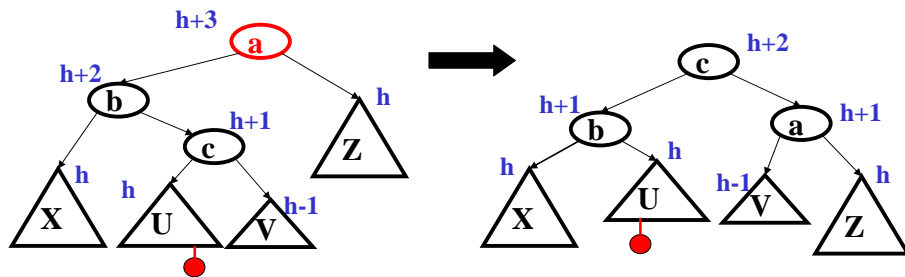


Easier to remember than you may think:

Move `c` to grandparent's position and then put `a`, `b`, `X`, `U`, `V`, and `Z` in the right places to get a legal 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

## Now efficiency

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

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

Will take some more rotation action to handle **delete**...