BST & AVL Trees

CSE 332 Summer 2020

Instructor: Richard Jiang

Teaching Assistants:

Hamsa Shankar Kristin Li Winston Jodjana

Maggie Jiang Hans Zhang Michael Duan

Jeffery Tian Annie Mao

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-07-08A

Announcements

- P2 is out! Once again, partner survey due Wednesday 6pm
- Quiz 2 out tonight, due 3am Saturday

pollev.com/332summer :: tinyurl.com/332-07-08A

Lecture Outline

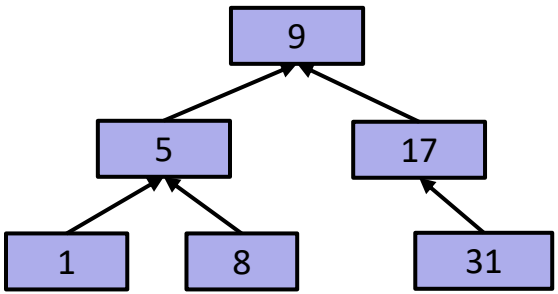
- Binary Search Trees
  - **Find/Add**
  - Remove
- AVL Tree
  - AVL Properties
  - Proving the Height Bound
  - Find
  - Add
  - Remove

pollev.com/332summer :: tinyurl.com/332-07-08A

Binary Search Trees: Find/Contains

- Unsurprisingly, this looks a lot like binary search
- Can you implement contains by putting the following statements in the correct order?
  - Hint: remember BST's invariants
- What is find's worst-case runtime?

```
boolean contains(BSTNode n,
                 Key k) {
   ABCD or ABDC
}
```



A

```
if (n == null)
  return false;
```

B

```
if (k.equals(n.key))
  return true;
```

C

```
if (k < n.k) {
  return contains(
      n.left, k);
}
```

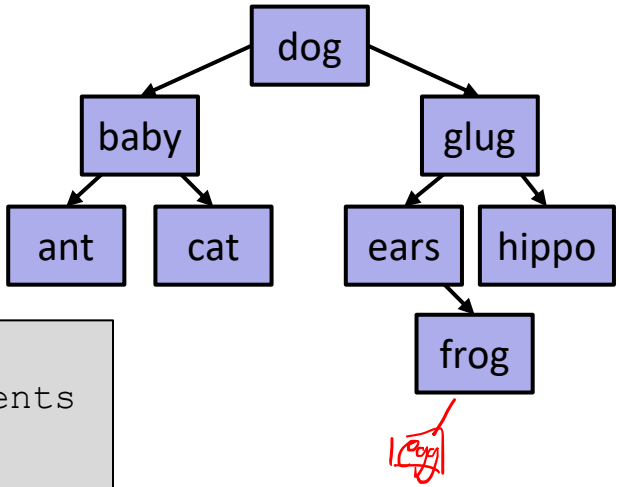
D

```
if (k >= n.k) {
  return contains(
      n.right, k);
}
```

Binary Search Trees: Add

- Where does the new item belong?
- How do we use BST invariants to ensure the leaf is added correctly?

must be leaf




```
BSTNode add(BSTNode t, Item i) {
  // Implement by putting statements
  // in the correct order

  D B C A

}
```

A	B	C	D
`return t;`	`if (k < i.key)) { t.left = add(t.left, i); }`	`if (k > i.key) { t.right = add(t.right, i); }`	`if (t == null){ return new BSTNode(i); }`

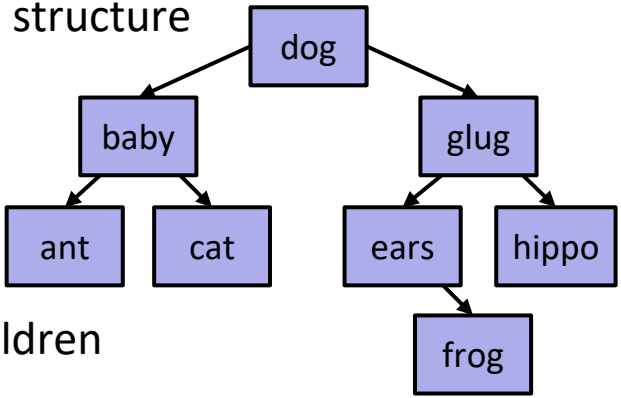
Lecture Outline

- Binary Search Trees
  - Find/Add
  - **Remove**
- AVL Tree
  - AVL Properties
  - Proving the Height Bound
  - Find
  - Add
  - Remove

pollev.com/332summer :: tinyurl.com/332-07-08A

Binary Search Trees: Remove

- Removing an item disrupts the tree structure
  - `find` the node to be removed
  - Remove it
  - "Fix" the tree so that it is still a BST



- 3 cases based on the number of children
  1. Node has no children
  2. Node has one child
  3. Node has two children

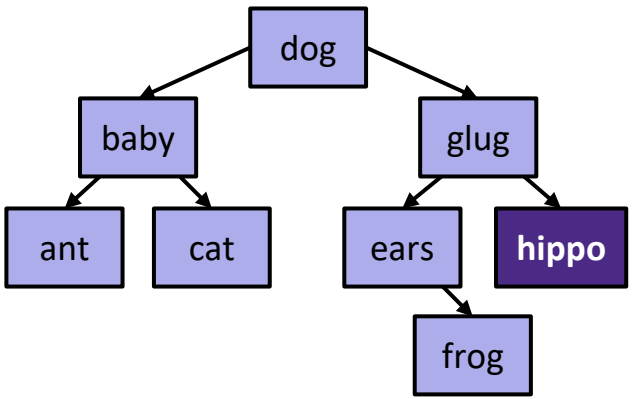
Reminder: a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

- In each case, we must maintain the **BST Ordering**!

BST Remove: Case #1: Leaf

- Remove the node with the key **hippo**

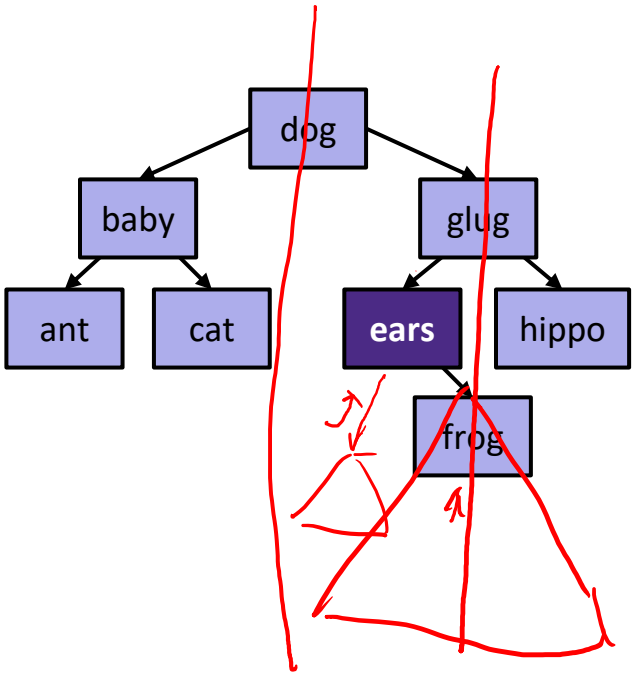
```
BSTNode remove(BSTNode n) {

    find n

    remove n



}
```



BST Remove: Case #2: One Child

- Remove the node with the key **ears**
  - What does the BST invariant say about the descendant's keys?

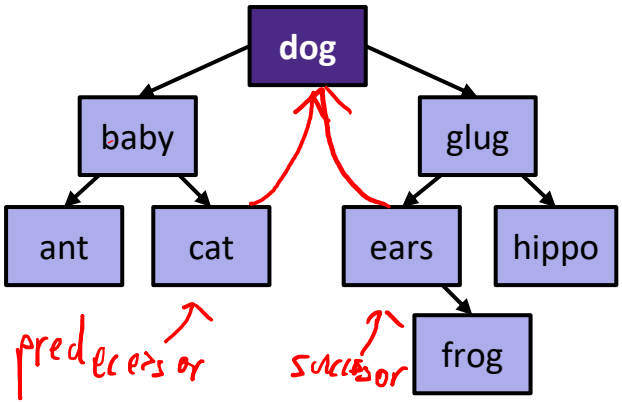
```
BSTNode remove(BSTNode n) {
    Find n
    remove n
    put child in n's spot

}
```



BST Remove: Case #3: Two Children

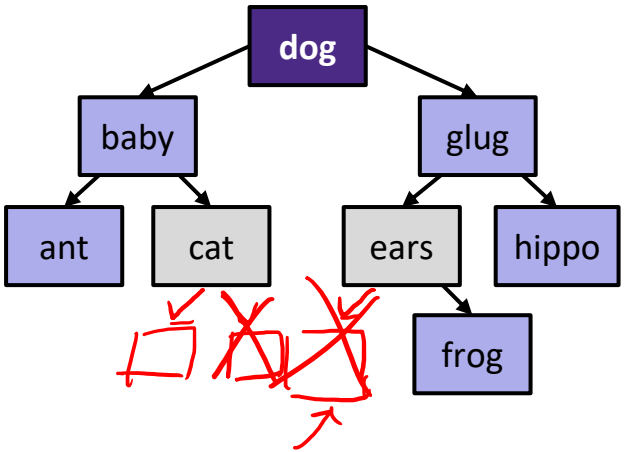
- Remove the node with the key **dog**

- The replacement node's key:
  - Must be $\succ$ than all keys in left subtree
  - Must be $\prec$ than all keys in right subtree



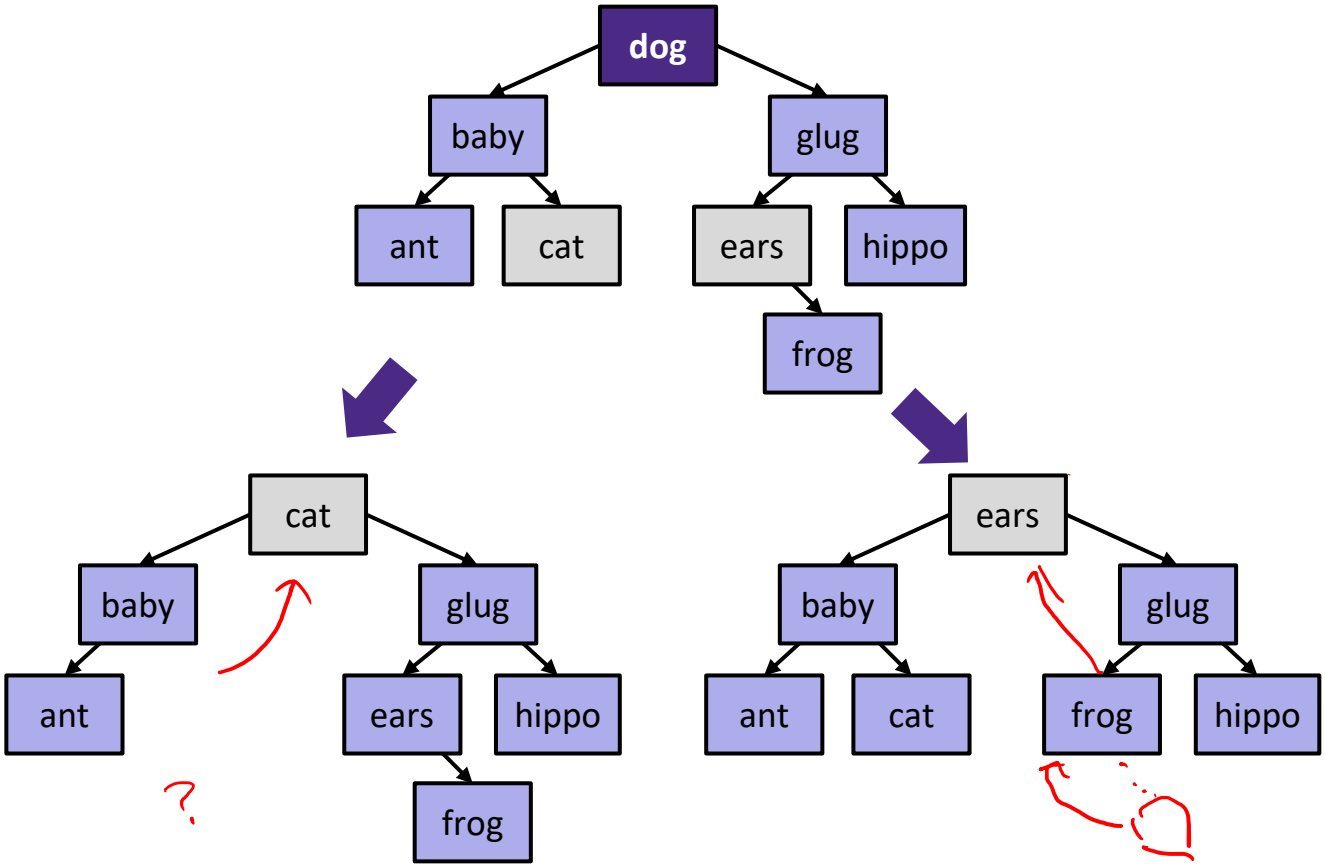
BST Remove: Case #3: Two Children

- Remove the node with the key **dog**



- The replacement node's key:
  - Must be $\succ$ than all keys in left subtree: **predecessor** (**cat**)
  - Must be $\prec$ than all keys in right subtree: **successor** (**ears**)
- The predecessor or successor have either 0 or 1 children

BST Remove: Case #3: Two Children

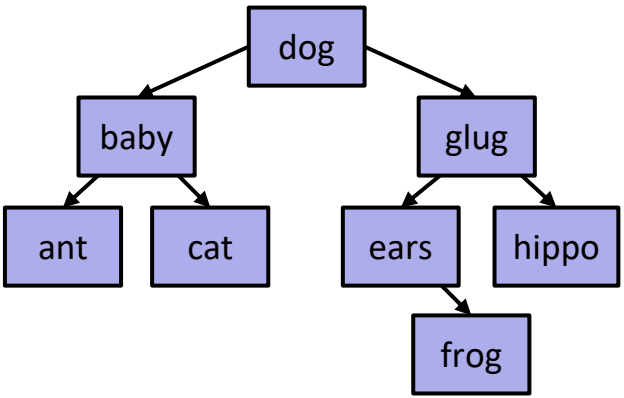


Aside: Finding the largest (or smallest) node

- The predecessor is the largest item in the left subtree
- The successor is the smallest item in the right subtree

- How do you find the largest (and smallest) item in a tree?
  - Remember that subtrees are trees too

```
BSTNode largest(BSTNode n) {
  while (n.right != null) {
    n = n.right;
  }
  return n;
}
```



BST Summary

- Binary Search Trees implement both the Set and Dictionary ADTs

- Binary Search Trees are *recursively defined*

	LinkedList Dictionary, Worst Case	BST Dictionary, Average Case	BST Dictionary, Worst Case
		Randomized	
Find	Θ(N)	Θ(h) aka Θ(log N)	Θ(h) aka Θ(N)
Add	Θ(N)	Θ(h) aka Θ(log N)	Θ(h) aka Θ(N)
Remove	Θ(N)	Θ(h) aka Θ(log N)	Θ(h) aka Θ(N)

Lecture Outline

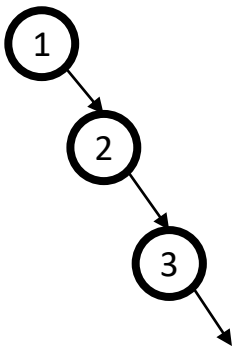
- Binary Search Trees
  - Find/Add
  - Remove
- AVL Tree
  - **AVL Properties**
  - Proving the Height Bound
  - Find
  - Add
  - Remove

pollev.com/332summer :: tinyurl.com/332-07-08A

buildTree for Binary Search Trees

- We had **`buildHeap`**, so let's consider **`buildTree`**

- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
  - If inserted in given order, what is the resultant tree?
  - What is the big-O runtime for sorted input?
  - Is inserting in the reverse order any better?



buildTree Runtime: $O(n^2)$

(not a happy place)

Balancing a BST

Observation

- BST: the shallower the better!
- For a BST with *n* items 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 order lead to worst-case!

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

- ensures depth is always $O(\log n)$
- is easy to maintain

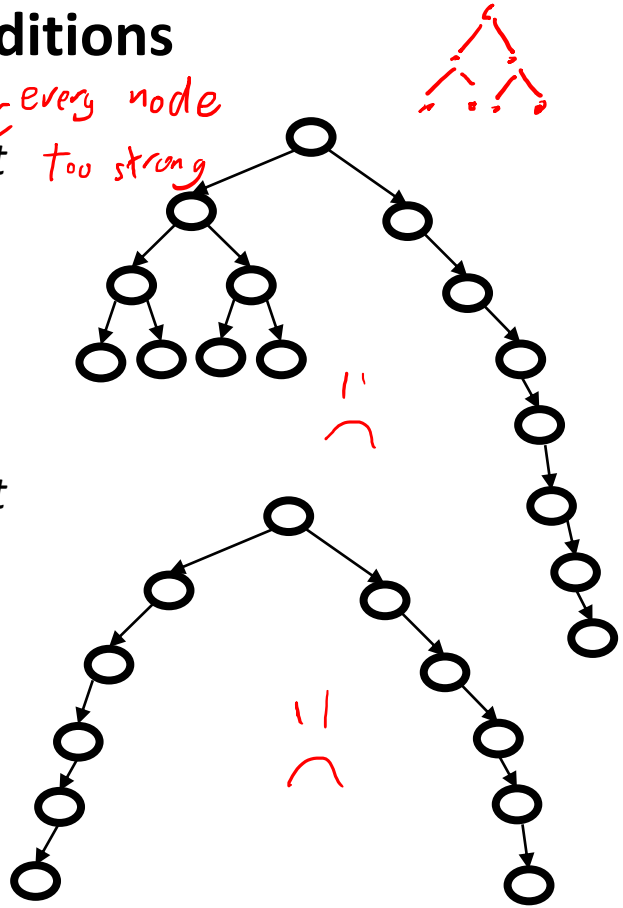
Potential BST Balance Conditions

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

  Too weak!
  Height mismatch example:

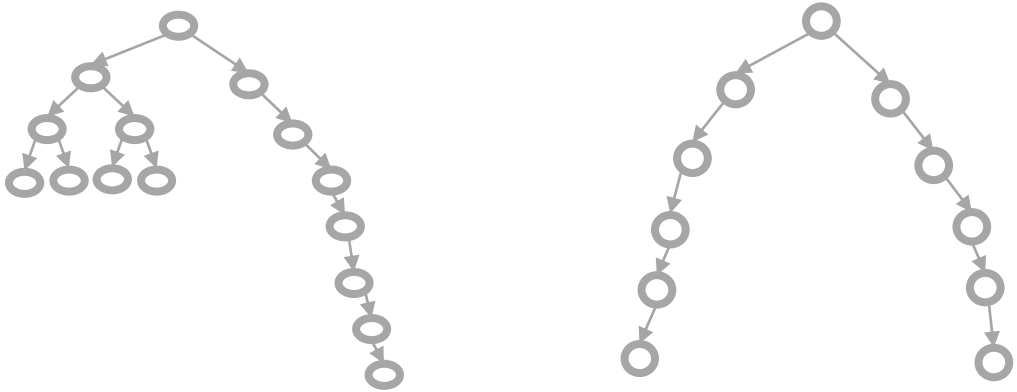
- Left and right subtrees of the *root* have equal *height*

  Too weak!
  Double chain example:



The AVL Balance Condition (1 of 2)

- Left and right subtrees of the *root* have equal number of nodes
- Left and right subtrees of the *root* have equal *height*



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

The AVL Balance Condition (2 of 2)

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

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

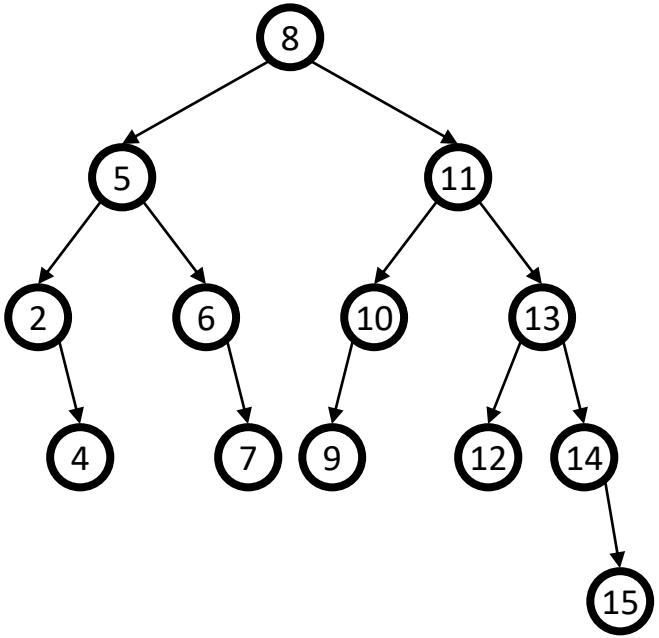
AVL property: **for every node *x*, $-1 \leq$ balance(*x*) ≤ 1**

Results:

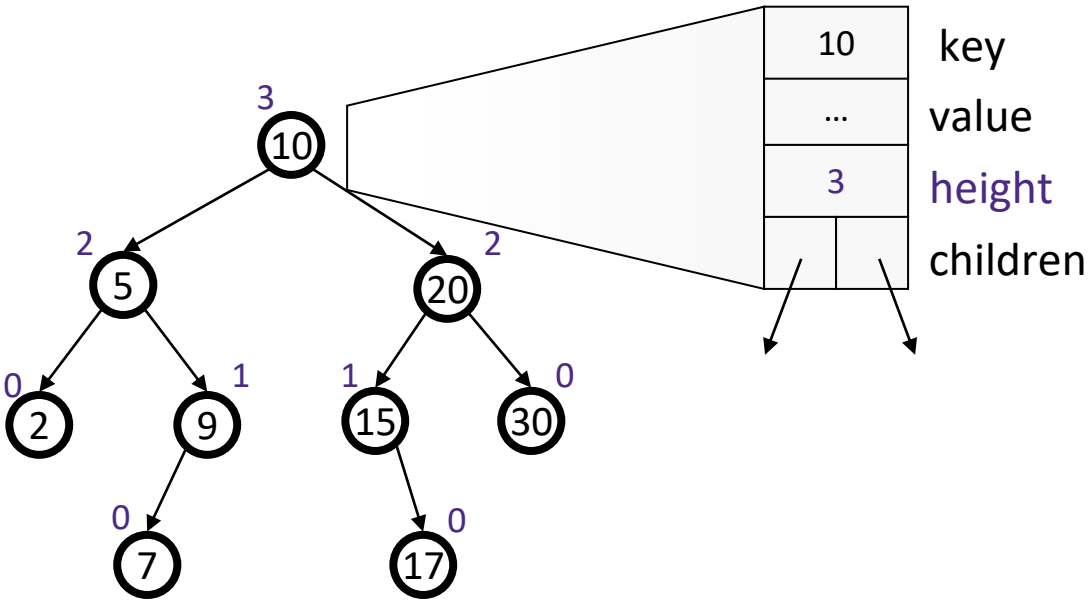
- Ensures shallow depth: h ∈ O(log n)
  - Will prove this by showing that an AVL tree of height *h* must have a number of nodes *exponential* in *h*
- Efficient to maintain using rotations

The AVL Tree Data Structure (1 of 2)

- Structural properties
  - Binary tree property (0, 1, or 2 children)
  - Heights of left and right subtrees for every node differ by at most 1
- Ordering property
  - Same as for BST



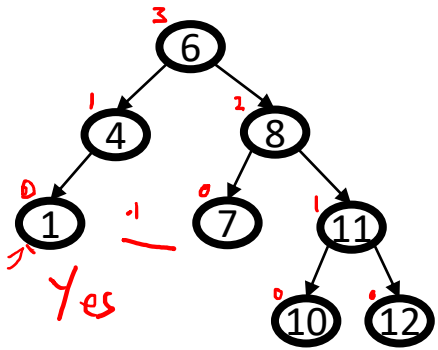
The AVL Tree Data Structure (2 of 2)

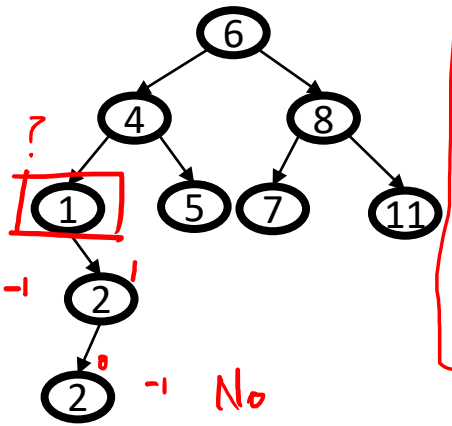


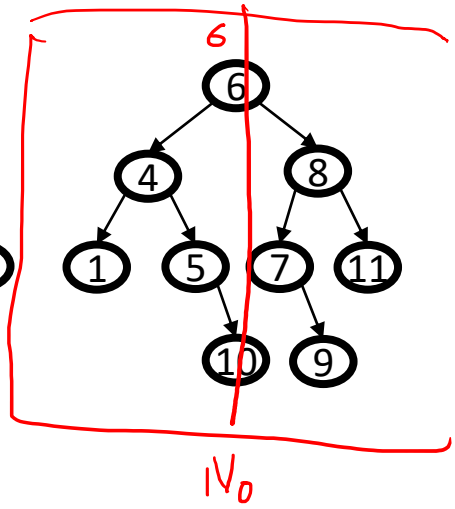
Poll Everywhere

pollev.com/332summer

- Are the following trees AVL trees?







A. No / No / No
B. Yes / No / No
C. Yes / Yes / No
D. Yes / Yes / Yes
E. Yes / No / Yes
F. I'm not sure …

Height of an AVL Tree? (1 of 2)

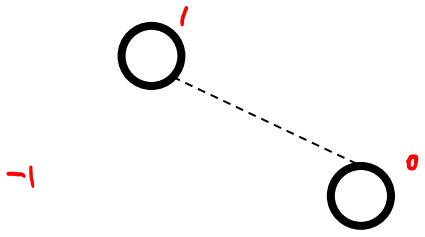
- The "best case" AVL tree is a perfect tree

- What does the "worst case" AVL tree look like?

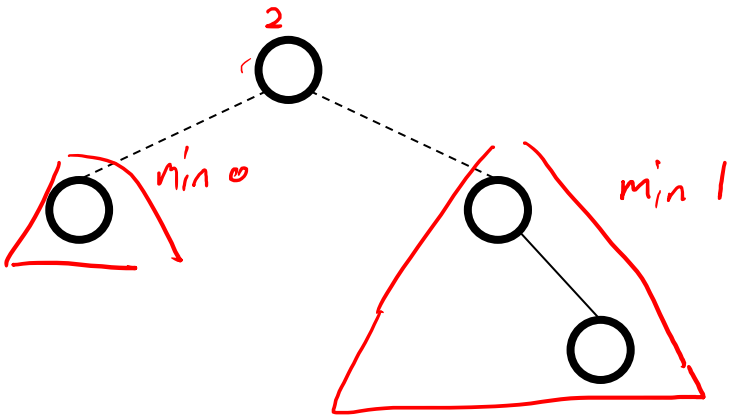
- Let S(h) be the minimum # of nodes in a tree of height h:
  - Where S(-1) = 0 and S(0) = 1

Minimal AVL Tree (height = 0)

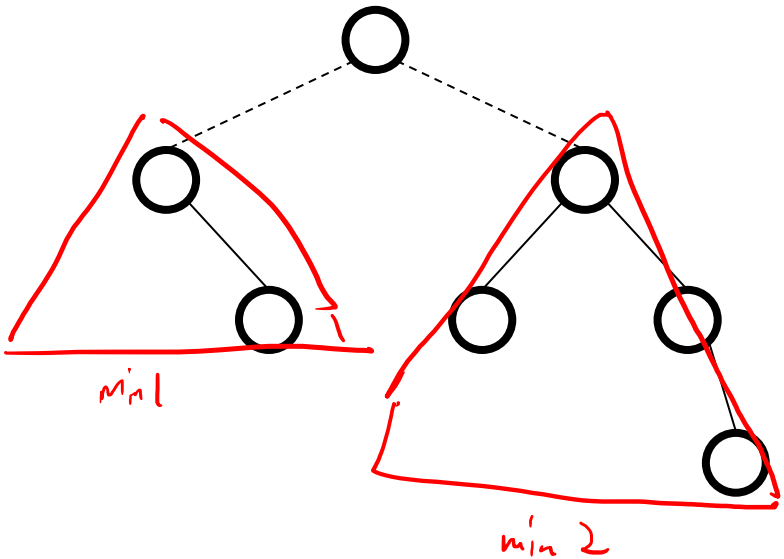
Minimal AVL Tree (height = 1)



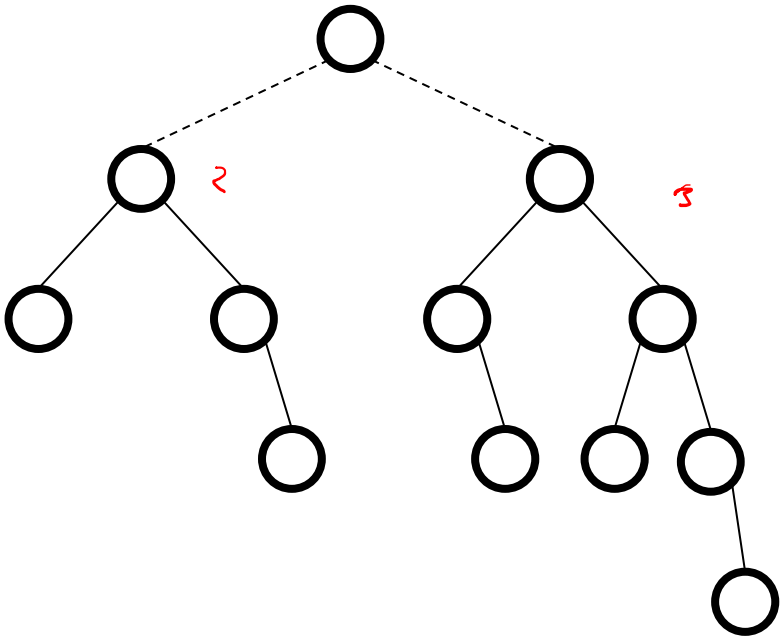
Minimal AVL Tree (height = 2)



Minimal AVL Tree (height = 3)



Minimal AVL Tree (height = 4)



Height of an AVL Tree? (2 of 2)

- 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 a tree of height *h*

$$\mathbf{S}(h) = \mathbf{S}(h-1) + \mathbf{S}(h-2) + 1 \quad \text{root node}$$

where $\mathbf{S}(-1) = 0$ and $\mathbf{S}(0) = 1$

- Solution of Recurrence: $\mathbf{S}(h) \approx 1.62^h$

$n = 1.62^h$

$\log_{1.62} n = h$ ↓ $h \rightarrow \Theta(\log n)$

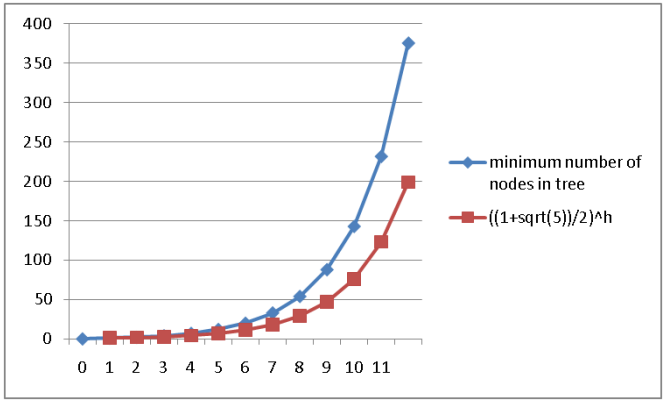
Lecture Outline

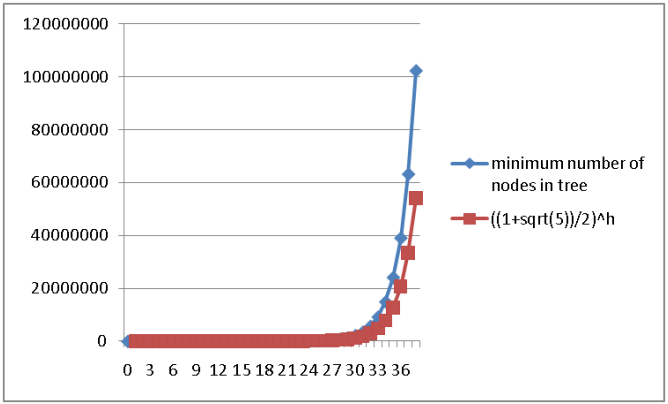
- Binary Search Trees
  - Find/Add
  - Remove
- AVL Tree
  - AVL Properties
  - **Proving the Height Bound**
  - Find
  - Add
  - Remove

pollev.com/332summer :: tinyurl.com/332-07-08A

Before We Prove It

- Good intuition from plots comparing:
  - $S(h)$ computed directly from the definition
  - $((1+\sqrt{5})/2)^h$ ← 1.62
- $S(h)$ is always bigger, up to trees with huge numbers of nodes
  - Graphs aren't proofs, so let's prove it

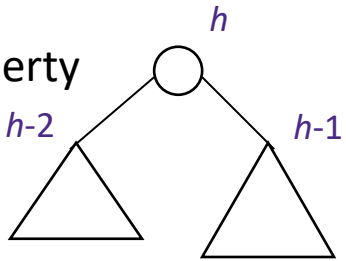




The Proof Outline

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

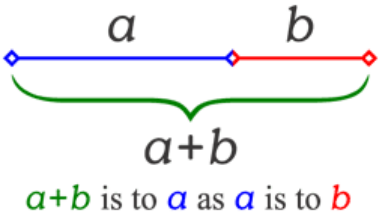
- If we can prove that $S(h)$ grows exponentially in h, then a tree with n nodes has a logarithmic height



- Step 1: Define $S(h)$ inductively using AVL property
  - $S(-1)=0$, $S(0)=1$, $S(1)=2$
  - *For $h \geq 1$, $S(h) = 1+S(h-1)+S(h-2)$*
- Step 2: Show this recurrence grows really fast
  - Similar to Fibonacci numbers
  - Can prove for all h, $S(h) > \phi^h - 1$ where ϕ is the golden ratio, $(1+\sqrt{5})/2$, about 1.62
  - Growing faster than 1.6^h is "plenty exponential"

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 = ϕb`

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

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 ($k > 1$):

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*
	$= \phi^k + \phi^{k-1} - 1$	*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)*

Lecture Outline

- Binary Search Trees
  - Find/Add
  - Remove
- AVL Tree
  - AVL Properties
  - Proving the Height Bound
  - **Find**
  - Add
  - Remove

pollev.com/332summer :: tinyurl.com/332-07-08A

AVL Find

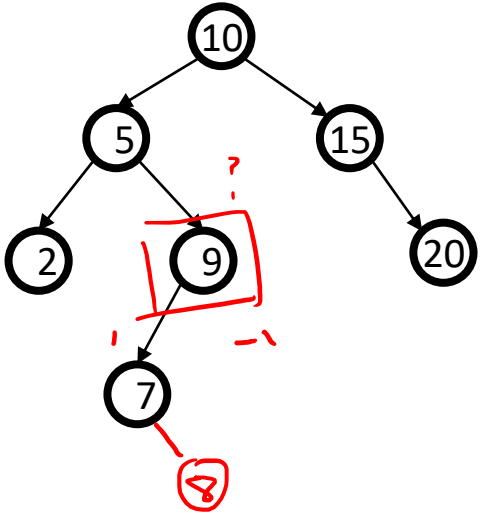
- Surprise! You already know this one, it's the same as BST Find

And Now for Some Bad News …

- find() is O(log n)!

- But as we add() and remove elements(), we need to:
  - Track heights
  - Detect imbalance
  - Restore balance

Is this tree AVL-balanced?
How about after insert(8)?

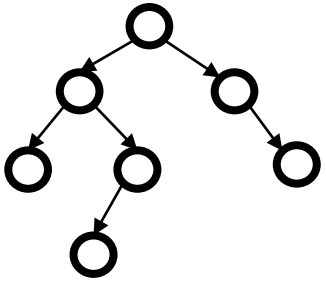


Lecture Outline

- Binary Search Trees
  - Find/Add
  - Remove
- AVL Tree
  - AVL Properties
  - Proving the Height Bound
  - Find
  - **Add**
  - Remove

pollev.com/332summer :: tinyurl.com/332-07-08A

AVL add(): Overall Approach



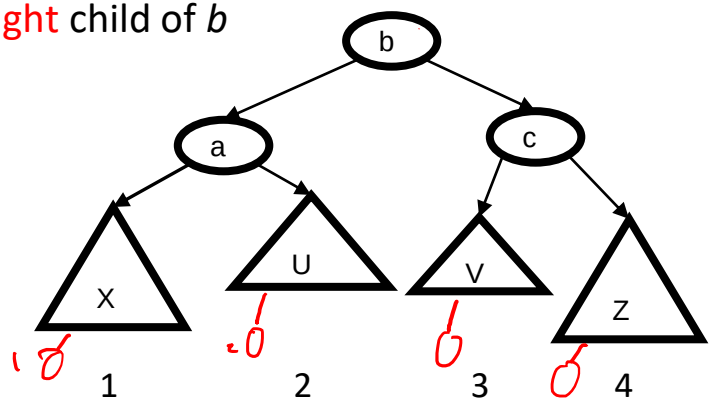
- Our overall algorithm looks like:
  1. Insert the new node as in a BST (a new leaf)
  2. For each node on the path from the root to the new leaf:
     - The insertion may (or may not) have changed the node's height
     - Detect height imbalance and perform a *rotation* to restore balance

- Fact that makes it a bit easier:
  - Imbalances only occur along the path from the new leaf to the root
  - There must be a deepest element that is unbalanced
  - After rebalancing this deepest node, every node above it is also rebalanced
  - Therefore, *at most one need needs to be rebalanced*

AVL add(): Cases

- Let *b* be the node where an imbalance occurs

- There are four cases to consider. The insertion is in the:
  1. left subtree of the left child of *b*
  2. right subtree of the left child of *b*
  3. left subtree of the right child of *b*
  4. right subtree of the right child of *b*



Case #1: Example

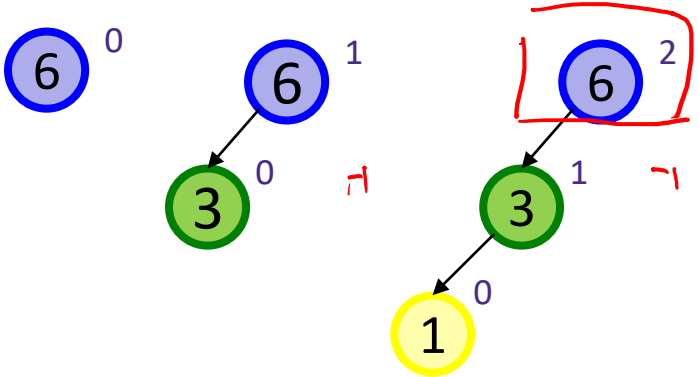
The insertion is in the:

1. left subtree of the left child of *b*
2. right subtree of the left child of *b*
3. left subtree of the right child of *b*
4. right subtree of the right child of *b*

add(6)

add(3)

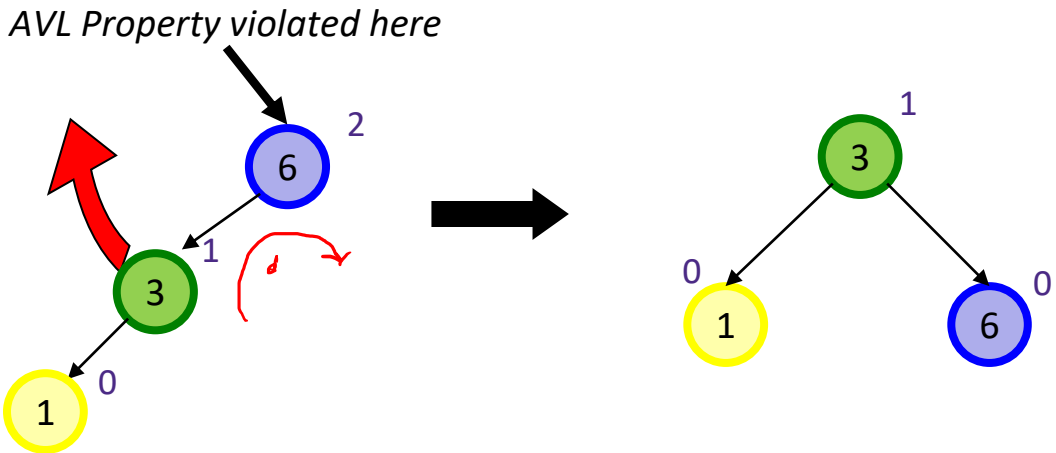
add(1)



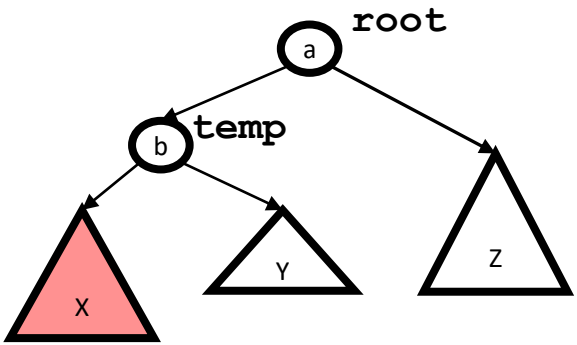
- Last add() violates balance property
- What is the only way to fix this?

Case #1 Fix: Apply "Single Rotation"

- ***Single rotation:***
  - Move child of unbalanced node into parent position
  - Parent becomes the "other" child



Case #1: Pseudocode



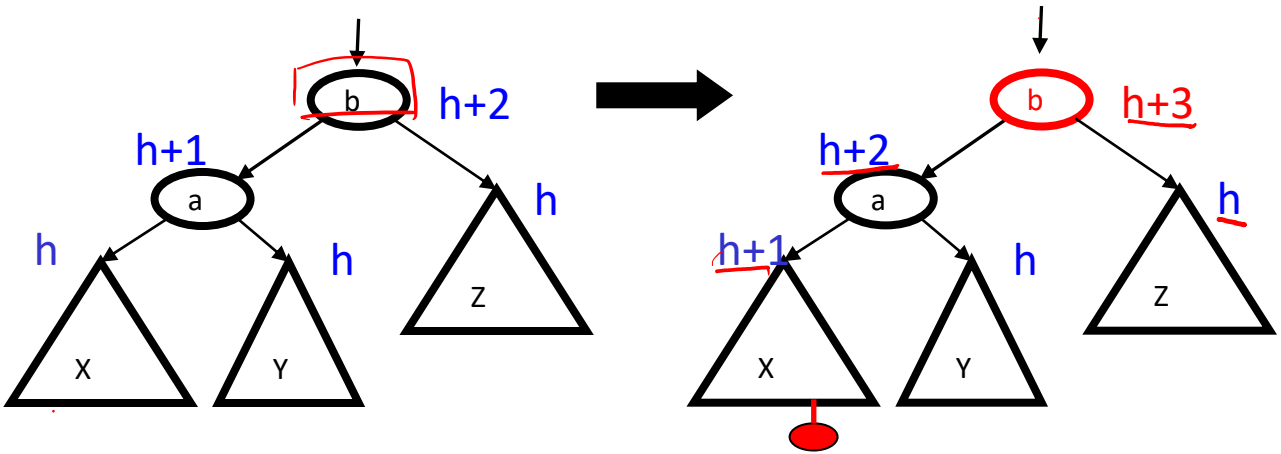

```
void RotateWithLeftChild(Node root) {
  Node temp = root.left
  root.left = temp.right
  temp.right = root
  root.height = max(root.right.height(),
                    root.left.height()) + 1
  temp.height = max(temp.right.height(),
                    temp.left.height()) + 1
  root = temp
}
```

RotateWithLeftChild rotates the tree clockwise

Case #1: Why It Works (1 of 2)

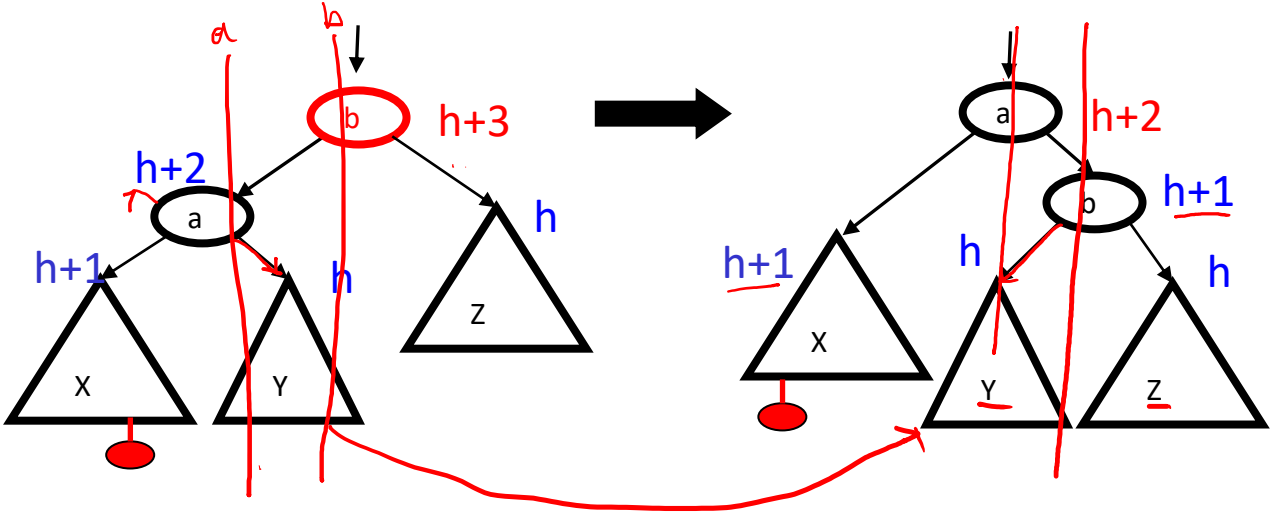
Oval: a node in the tree
Triangle: a subtree

- Node is imbalanced due to insertion *somewhere* in **left-left grandchild**
- First we did the insertion, which would make ***b*** imbalanced



Case #1: Why It Works (2 of 2)

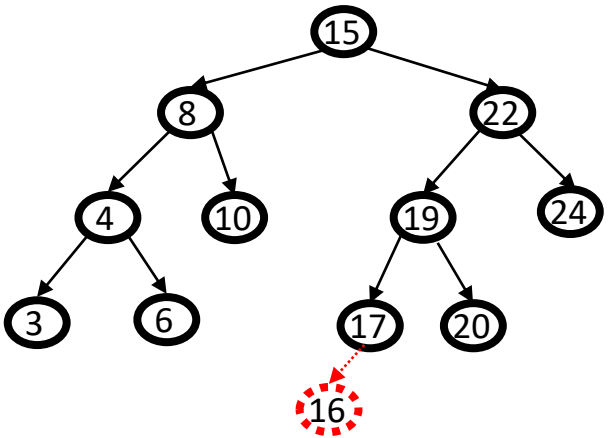
- So we rotate at b, maintaining BST order: X < a < Y < b < Z
- Result:
  - A single rotation restores balance at the formerly-imbalanced node
  - Height is same as before insertion, so ancestors now balanced



Case #1: Another Example: add(16)

The insertion is in the:

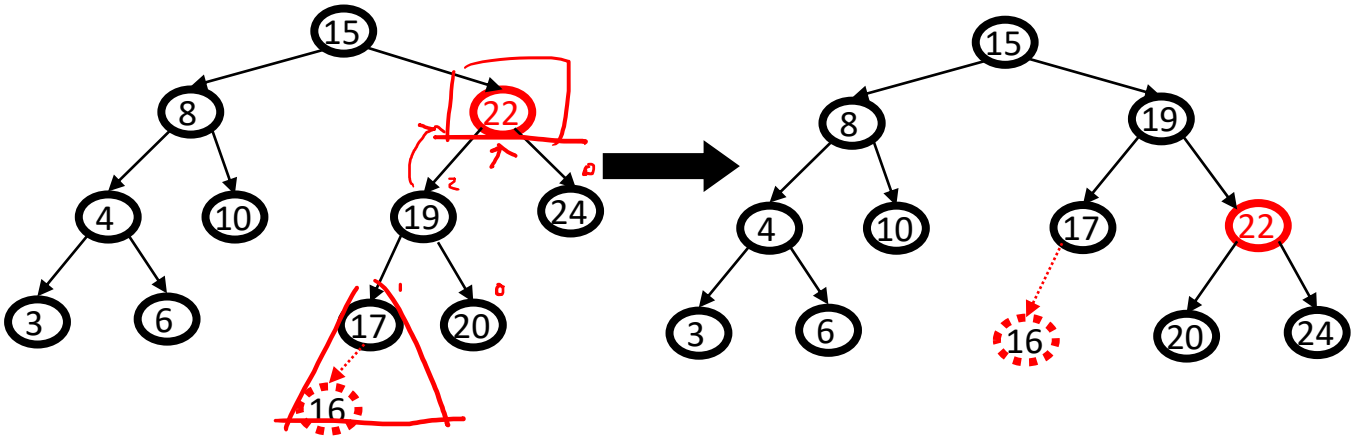
1. left subtree of the left child of *b*
2. right subtree of the left child of *b*
3. left subtree of the right child of *b*
4. right subtree of the right child of *b*



Case #1: Another Example: add(16)

The insertion is in the:

1. left subtree of the left child of *b*
2. right subtree of the left child of *b*
3. left subtree of the right child of *b*
4. right subtree of the right child of *b*

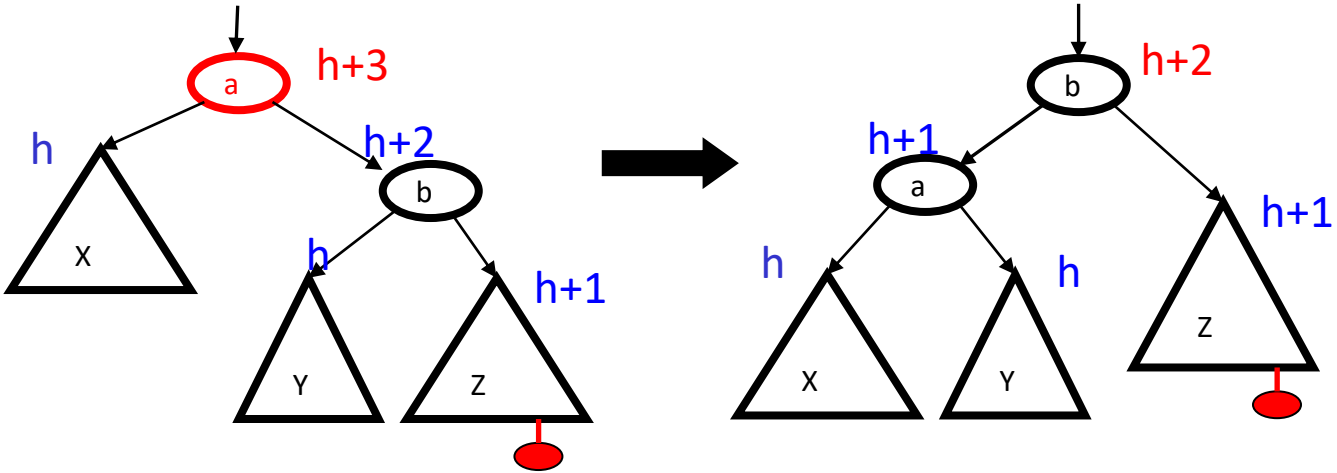


Case #1 ≈ Case #4

The insertion is in the:

1. left subtree of the left child of *b*
2. right subtree of the left child of *b*
3. left subtree of the right child of *b*
4. right subtree of the right child of *b*

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



RotateWithRightChild rotates the tree counter-clockwise

Case #3: Example

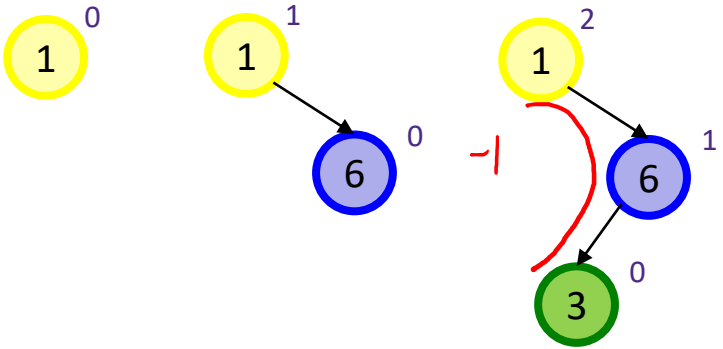
Insert(1)

Insert(6)

Insert(3)

The insertion is in the:

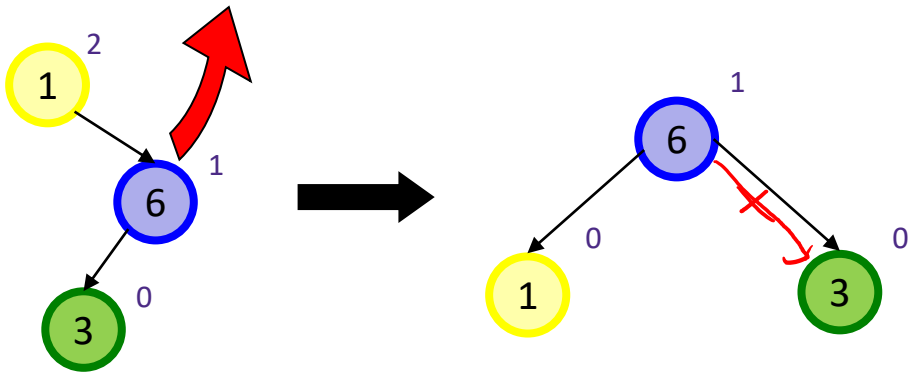
1. left subtree of the left child of *b*
2. right subtree of the left child of *b*
3. left subtree of the right child of *b*
4. right subtree of the right child of *b*



- Single rotations are not enough for insertions into the left-right subtree (or the right-left subtree; ie, case #2)

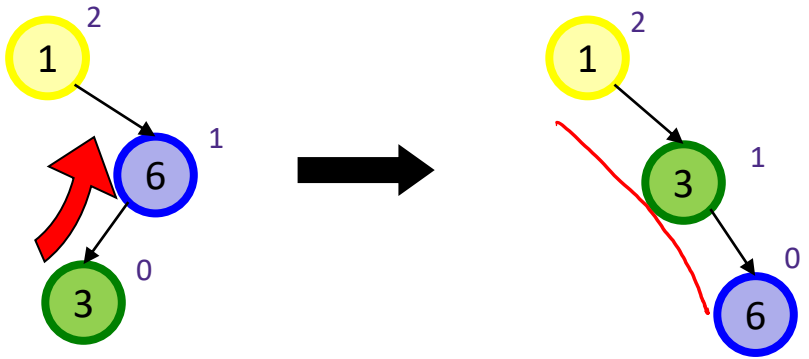
Case #3: Wrong Fix #1

- **First wrong idea**: single rotation like we did for left-left
  - Violates BST ordering property!



Case #3: Wrong Fix #2

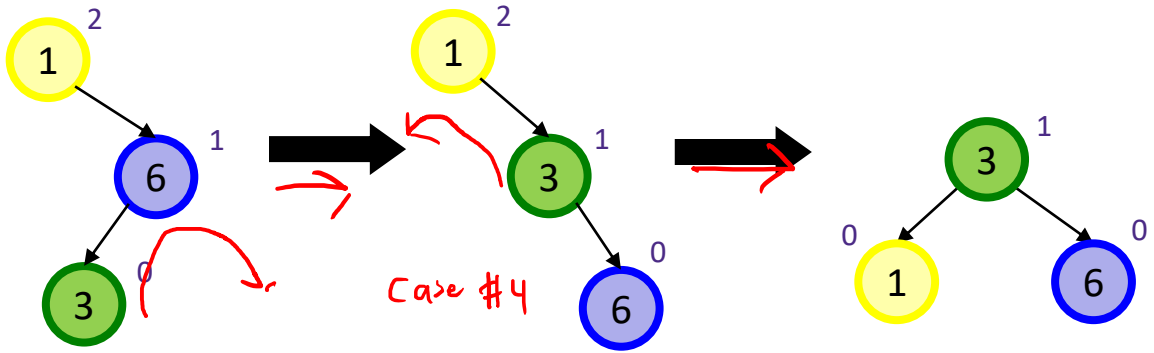
- **Second wrong idea**: single rotation on the child of the unbalanced node
  - Doesn't actually fix anything!



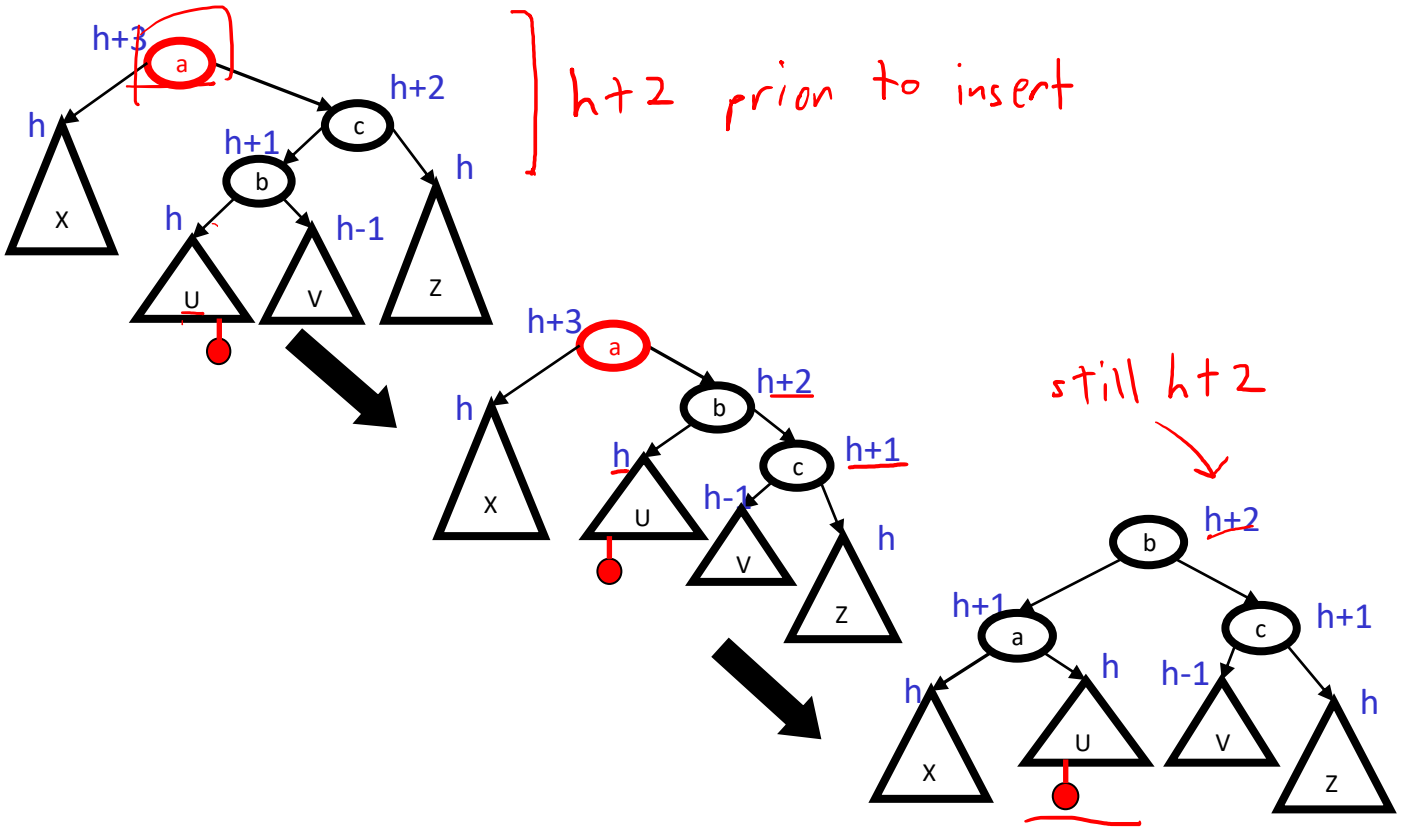
Case #3: Sometimes Two Wrongs Make a Right ☺

- First idea violated the BST ordering
- Second idea didn't fix balance
- … but if we do both single rotations, starting with the second, it works!

DoubleRotation:
1. Rotate problematic child and grandchild
2. Then rotate between self and new child

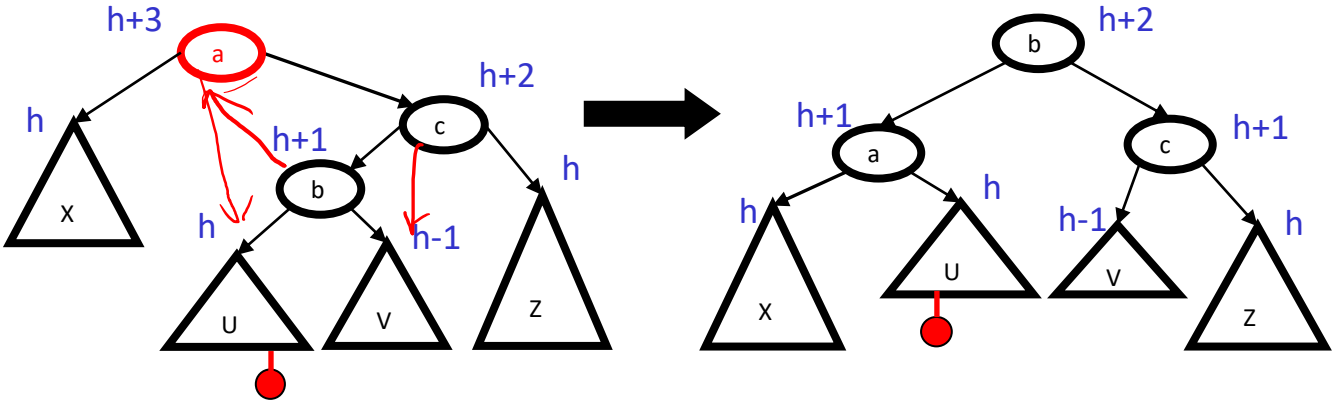


Case #3: Why It Works



Case #3: Comments

- Height of subtree after rebalancing is the same as before insert
  - So, no ancestor in the tree will need rebalancing
- Doesn't have to be two rotations; can just move b to grandparent's position and put a, c, X, U, V, and Z in the only legal positions for a BST



Case #3: Pseudocode

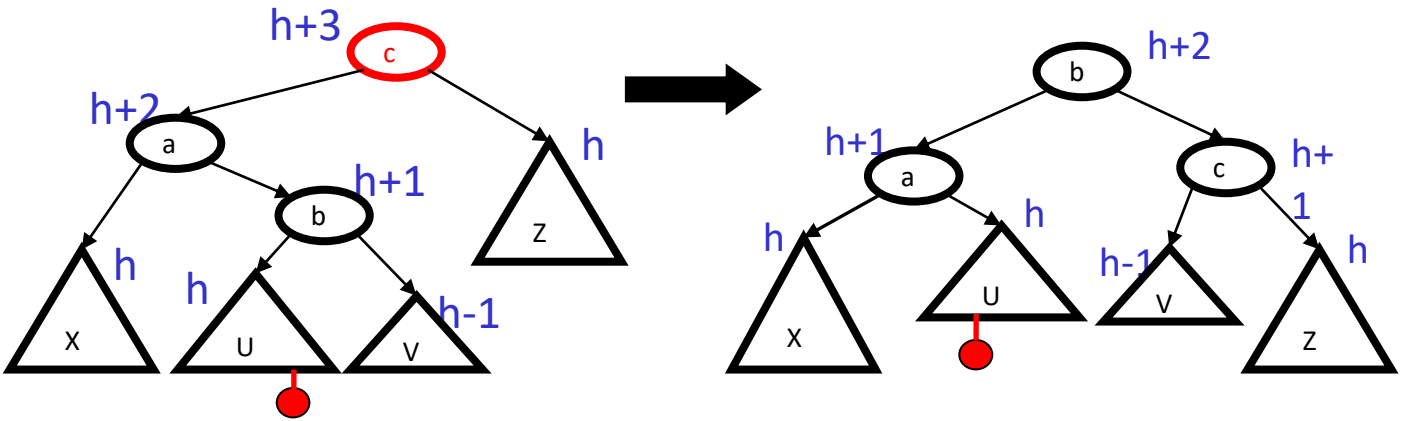
```
void DoubleRotateWithRightChild(Node root) {
  RotateWithLeftChild(root.right)
  RotateWithRightChild(root)
}
```

Case #3 ≈ Case #2

The insertion is in the:

1. left subtree of the left child of *b*
2. right subtree of the left child of *b*
3. left subtree of the right child of *b*
4. right subtree of the right child of *b*

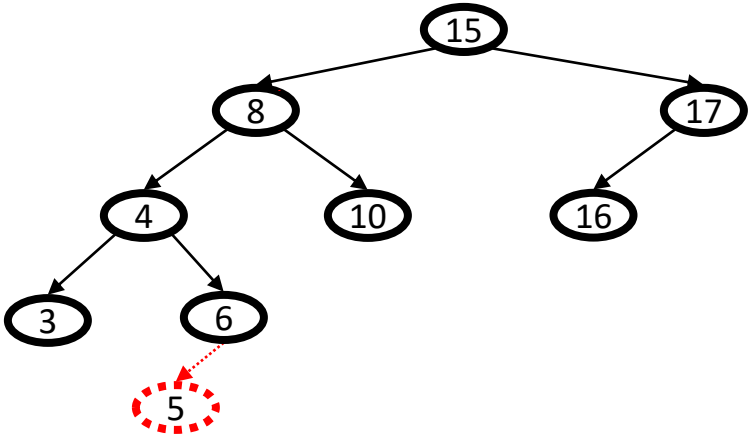
- Mirror image of right-left
  - Again, no new concepts, only new code to write



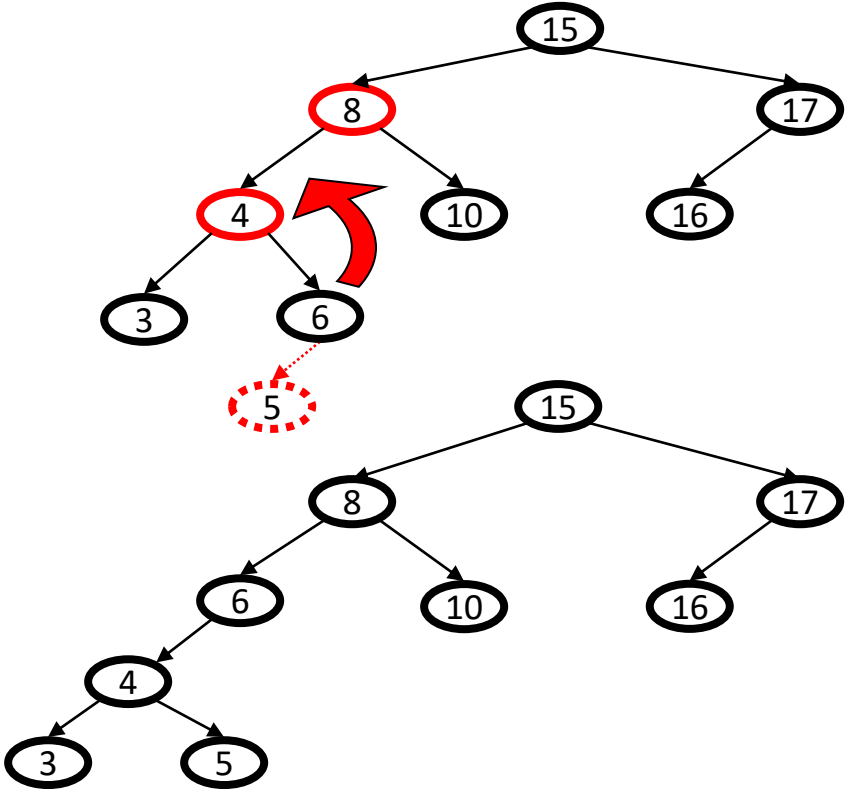
AVL add(): Summary

- Insert as if 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 insertion
  - So all ancestors are now balanced

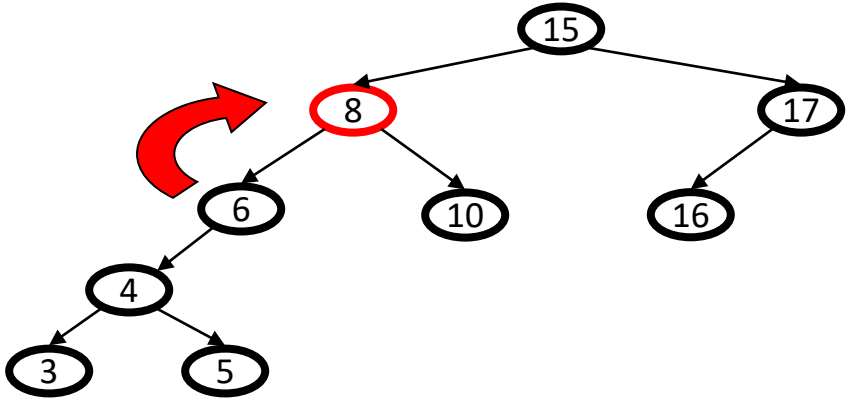
Double Rotation: Example (1 of 3)

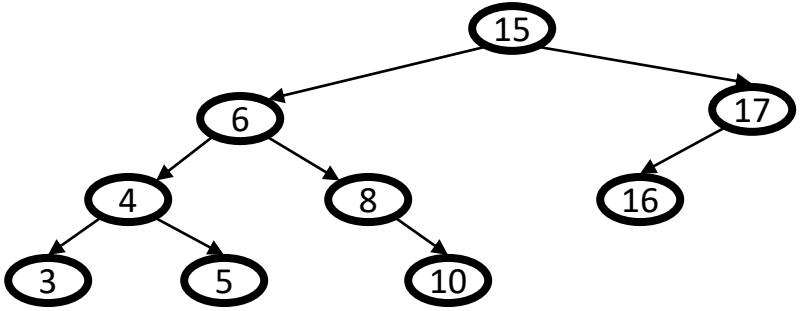


Double Rotation: Example (2 of 3)



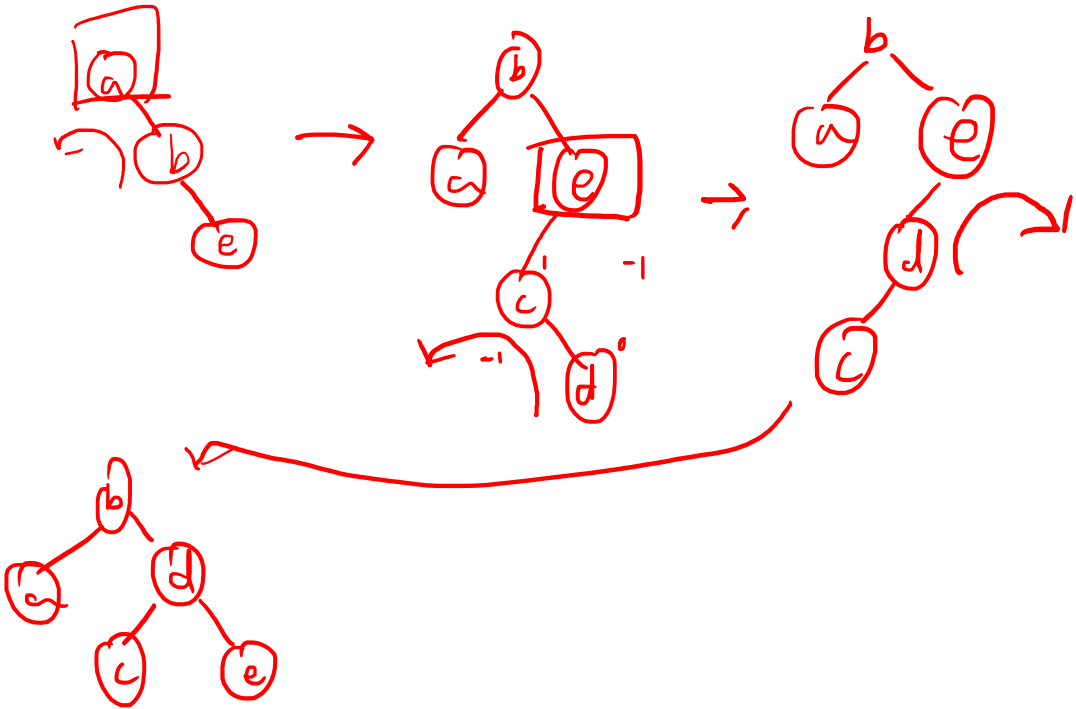
Double Rotation: Example (3 of 3)





Student Activity #1: add() into an AVL tree

- add(a)
- add(b)
- add(e)
- add(c)
- add(d)



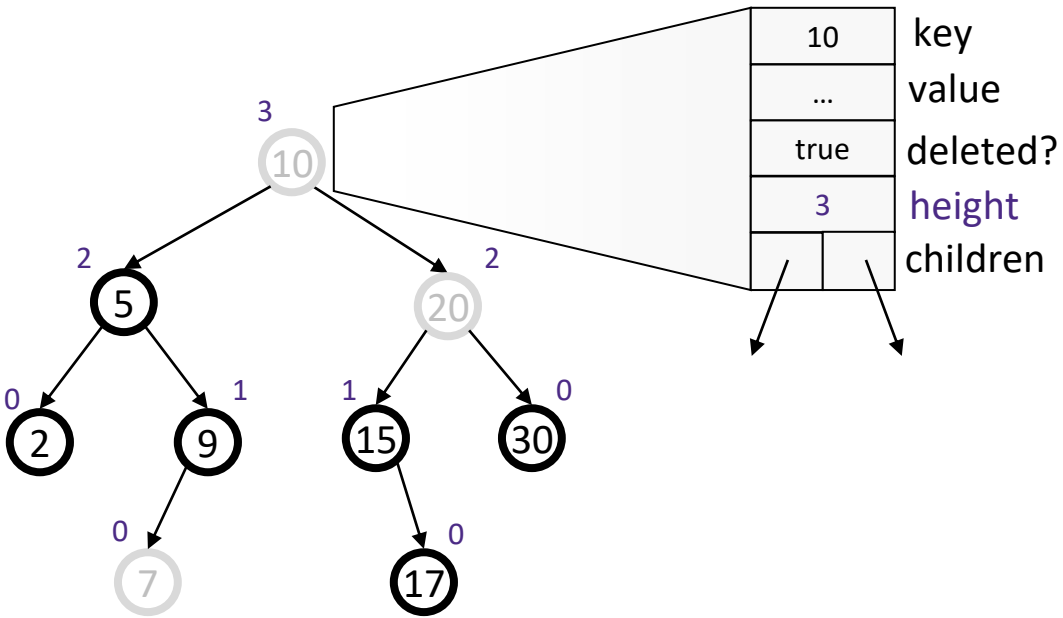
Lecture Outline

- Binary Search Trees
  - Find/Add
  - Remove
- AVL Tree
  - AVL Properties
  - Proving the Height Bound
  - Find
  - Add
  - **Remove**

pollev.com/332summer :: tinyurl.com/332-07-08A

AVL Remove

- The "easy way" is lazy deletion
  - Otherwise, we have several imbalance cases
  - See Weiss, 3rd ed. for more details



Pros and Cons of AVL Trees

- Arguments for AVL trees:
  - All operations are logarithmic worst-case because trees are always balanced
  - Height rebalancing adds no more than a constant factor to the speed of add and remove

- Arguments against AVL trees:
  - Difficult to program and debug
  - Additional space for the height and deleted? fields
  - Asymptotically faster, but rebalancing takes time
  - Most large data sets require database-like systems on disk, and thus use other structures (e.g., B-trees, our next data structure)