AVL Trees

CSE 332 Spring 2021

Instructor: Hannah C. Tang

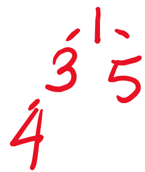
Teaching Assistants:

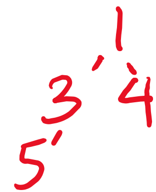
Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

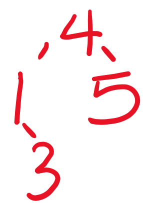
gradescope.com/courses/256241

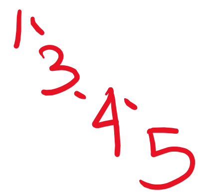
- In a binary **min heap**, repeatedly call `add()` on the following sequence of elements. Do not use buildHeap()
  - {4, 1, 5, 3}
  - {1, 3, 4, 5}





- In a binary **search tree**, repeatedly call `add()` on the following sequence of elements.
  - {4, 1, 5, 3}
  - {1, 3, 4, 5}





- What impact, if any, does the order of elements have on the resultant trees' **structure** and **ordering**?

Announcements

- Projects are due at 11:59pm
  - P1 had an extra day added; due *tonight*

- Fill out P2 partner survey tonight!

- Quiz 1’s question 3 (the one about spell prefixes)

Lecture Outline

- AVL Tree
  - **Bounding a BST's height**
  - *(Proving the AVL tree's height bound)*
  - Find
  - Add
    - *(Add Exercises)*
  - Remove
  - Wrapup

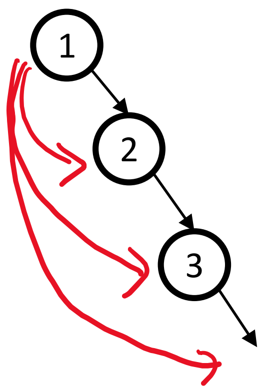
Why does BST height matter? (1 of 2)

	BST, Randomized	BST, Worst
Find	Θ(h) aka Θ(log N)	Θ(h) aka Θ(N)
Add	Θ(h) aka Θ(log N)	Θ(h) aka Θ(N)
Remove	Θ(h) aka Θ(log N)	Θ(h) aka Θ(N)

- For a BST with *n* items:
  - Randomized height is Θ(log n) – see text for proof
  - Worst case height is Θ(n)
- Simple cases, such as inserting in order, lead to worst case structure!

Why does BST height matter? (2 of 2)

- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
  - The resultant tree is a "linked list"
  - What is the big-Oh *aggregate* runtime for n add()s of sorted input?



Aggregate Runtime for n adds: O(n^2)

(not a happy place)

$$1+2+3+\ldots+n = \frac{n(n+1)}{2} \in O(n^2)$$

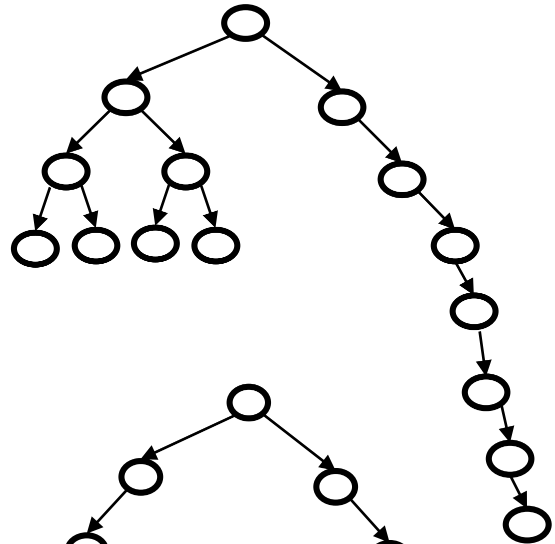
Balancing a BST

- *Solution*: Require a **Balance Condition** that:
  1. Ensures height is always O(log n)
  2. Is easy to maintain

Potential BST Balance Conditions

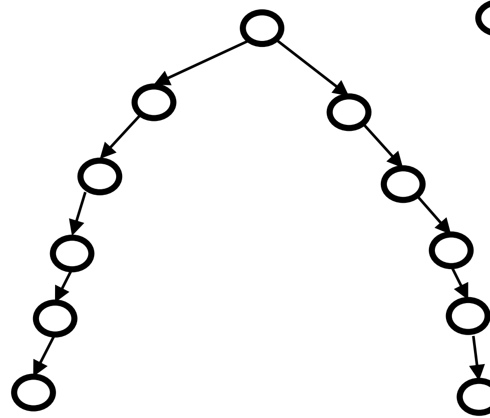
- Left and right subtrees of the *root* have equal number of nodes

Too weak!
Height mismatch example:



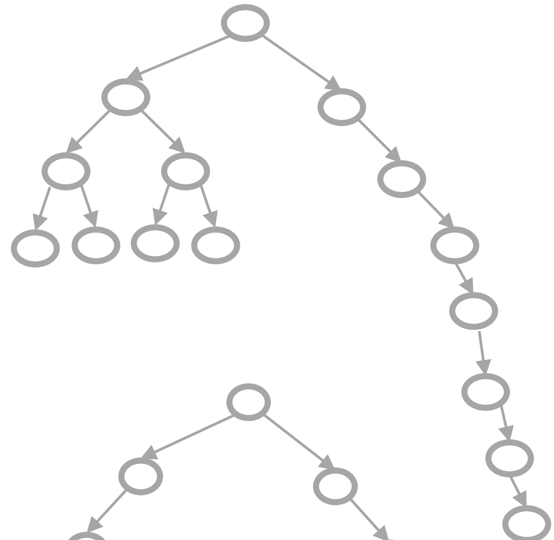
- Left and right subtrees of the *root* have equal *height*

Too weak!
Double chain example:

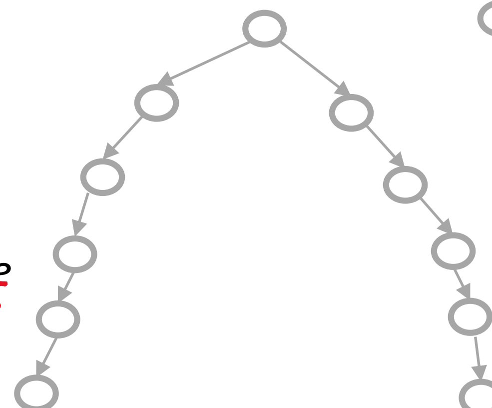


The AVL Balance Condition (1 of 2)

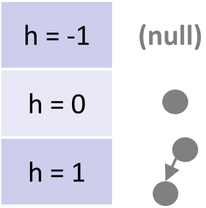
- Left and right subtrees of the *root* have equal number of nodes



- 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)

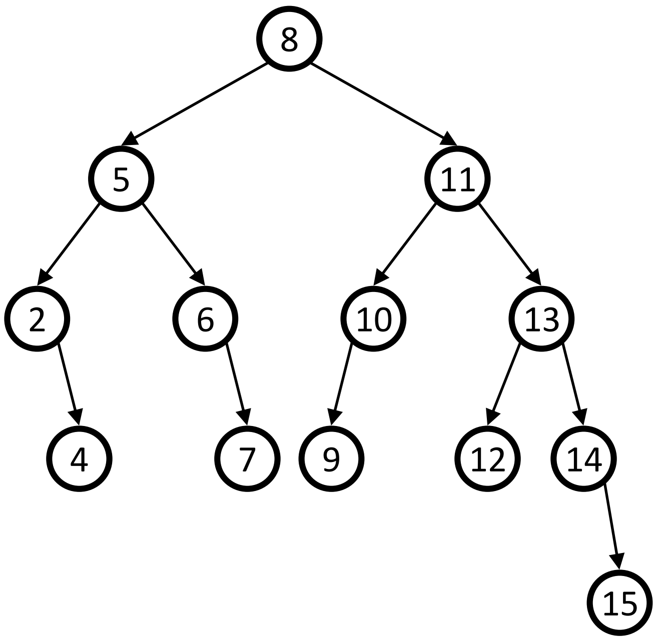
AVL property: **for every node *x*, –1 ≤ balance(*x*) ≤ 1**

Results:

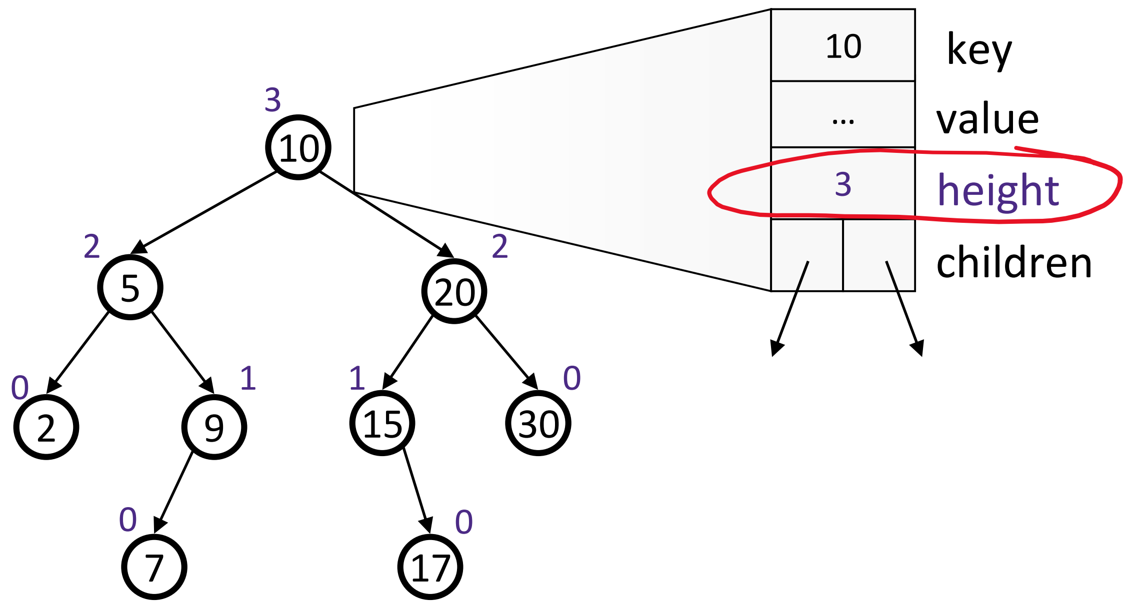
- Ensures shallow depth: h ∈ Θ(log n)
  - Will prove this by showing that an AVL tree of height *h* must have a number of nodes *exponential* in *h*
- Efficient to maintain using rotations

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

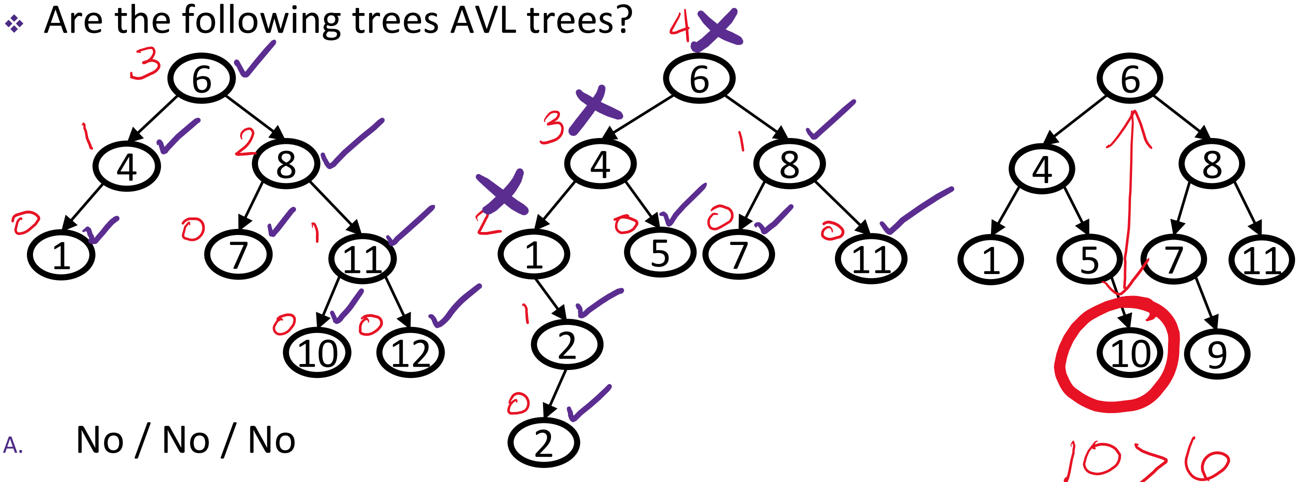


The AVL Tree Data Structure (2 of 2)



gradescope gradescope.com/courses/256241

- Are the following trees AVL trees?



A. No / No / No

B. Yes / No / No

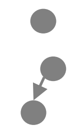
C. Yes / Yes / No

D. Yes / Yes / Yes

E. Yes / No / Yes

Height of an AVL Tree? (1 of 2)

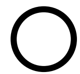
h = -1	(null)
h = 0	
h = 1	



- The "best case" AVL tree is a perfect tree
- What does the "worst case" AVL tree look like?
- Let $S(h)$ = minimum # of nodes in an AVL tree of height h
  - And also $S(-1) = 0$, $S(0) = 1$
  - … so what is the expression for $S(h)$?

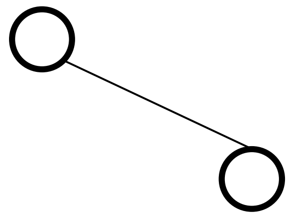
h = -1	(null)
h = 0	
h = 1	

Minimal AVL Tree (height = 0)

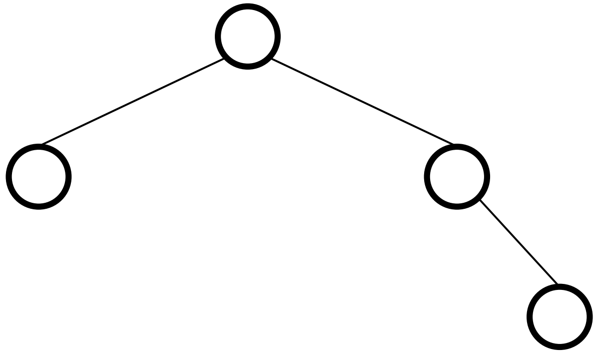


Minimal AVL Tree (height = 1)

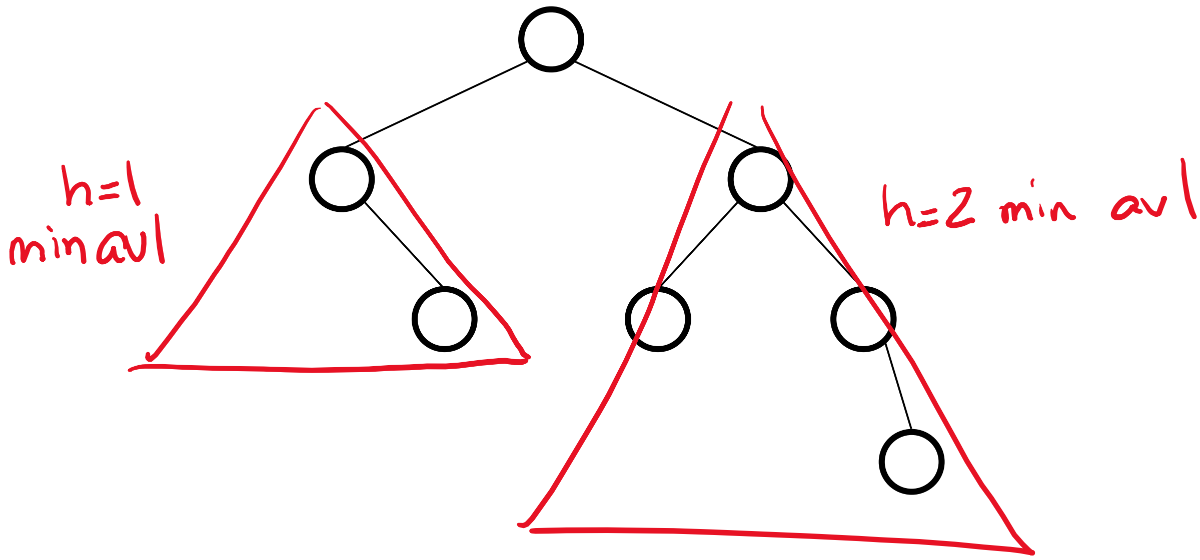
h = -1	(null)
h = 0	
h = 1	



Minimal AVL Tree (height = 2)

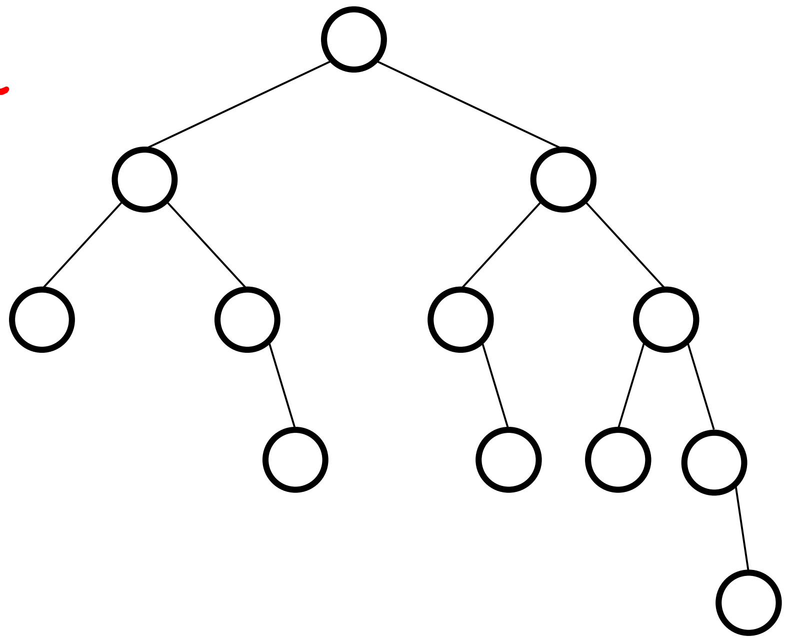


Minimal AVL Tree (height = 3)



Minimal AVL Tree (height = 4)

N	h
1	1
4	2
7	3
12	4



Height of an AVL Tree? (2 of 2)

- Let $S(h)$ = minimum # of nodes in an AVL tree of height h
  - And also $S(-1) = 0$, $S(0) = 1$
  - … what is the expression for $S(h)$?
  - $S(h) = S(h-1) + S(h-2) + 1$ the root

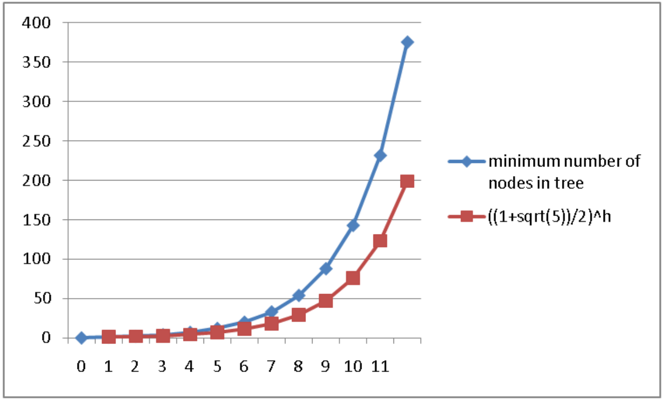
- Solution of Recurrence: $S(h) \approx 1.62^h$

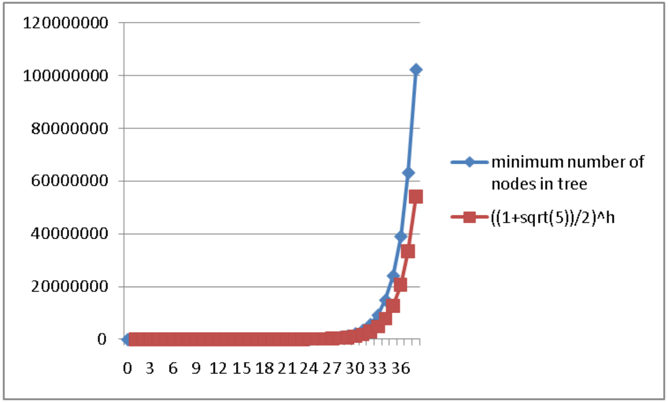
Lecture Outline

- AVL Tree
  - Bounding a BST's height
  - ***(Proving the AVL tree's height bound)***
  - Find
  - Add
    - *(Add Exercises)*
  - Remove
  - Wrapup

Before We Prove It

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



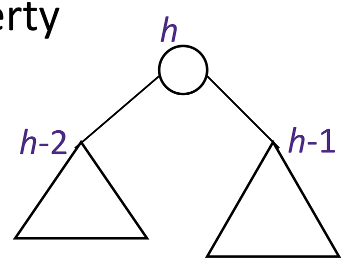


The Proof Outline

Let $S(h)$ = the min # 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

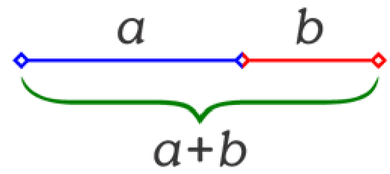
- Step 1: Define $S(h)$ inductively using AVL property
  - $S(-1)=0,\ S(0)=1,\ S(1)=2$
  - $S(h) = 1 + S(h-1) + S(h-2)$ for $h \geq 1$



- 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, $(1+\sqrt{5})/2 \approx 1.62$
  - Growing faster than 1.62^h is "plenty exponential"

Interlude: The Golden Ratio

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



a+b is to a as a is to b

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 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 (1 of 2)

$S(-1)=0, \quad S(0)=1, \quad S(1)=2$
$S(h)=1 + S(h-1) + S(h-2) \quad \text{for } h \geq 1$

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

Proof: By induction on h

Base cases:

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

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

The Proof (2 of 2)

```
S(-1)=0,   S(0)=1,   S(1)=2
S(h)=1 + S(h-1) + S(h-2)   for h≥ 1
```

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

Proof: By induction on h

Inductive case ($k > 1$):

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

- AVL Tree
  - Bounding a BST's height
  - *(Proving the AVL tree's height bound)*
  - **Find**
  - Add
    - *(Add Exercises)*
  - Remove
  - Wrapup

AVL Find

- Surprise! You already know this one

AVLs are BSTs!

Lecture Outline

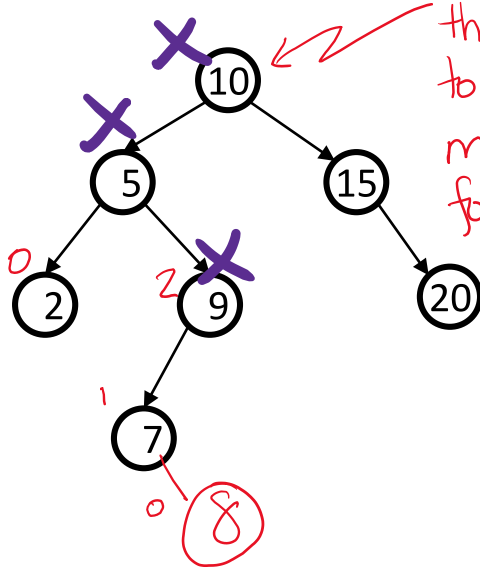
- AVL Tree
  - Bounding a BST's height
  - *(Proving the AVL tree's height bound)*
  - Find
  - **Add**
    - *(Add Exercises)*
  - Remove
  - Wrapup

And Now for Some Bad News …

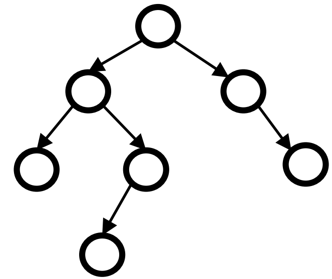
- 🎉 🎉 🎉 find() is O(log n)! 🎉 🎉 🎉
- But as we add() and remove elements(), we need to:
  - 👎 Track heights
  - 👎 Detect imbalance
  - 👎 Restore balance

Note how this happens to be our min AVL for h=3

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



AVL add(): Overall Approach



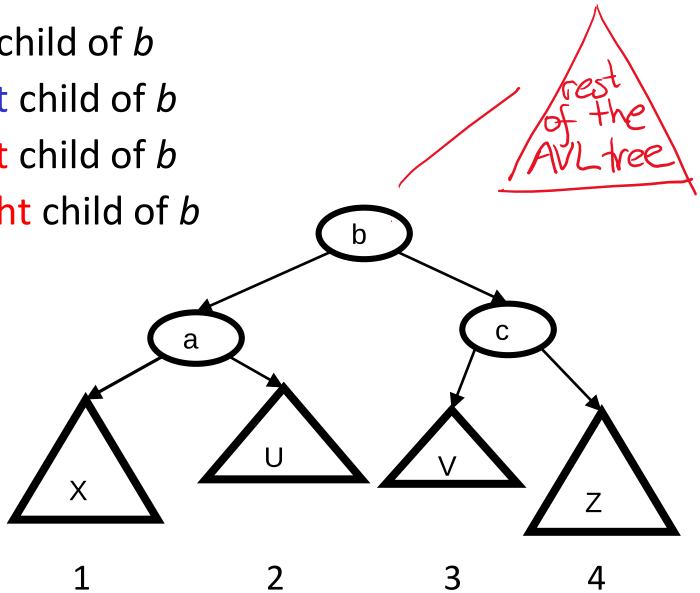
- Our overall algorithm looks like:
  1. Insert the new node as in a BST (a new leaf)
  2. For each node on the path from the root to the new leaf:
     - The insertion may (or may not) have changed the node’s height
     - Detect height imbalance and perform a *rotation* to restore balance

- 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 node needs to be rebalanced*

AVL add(): Cases

- Let *b* be the deepest 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*



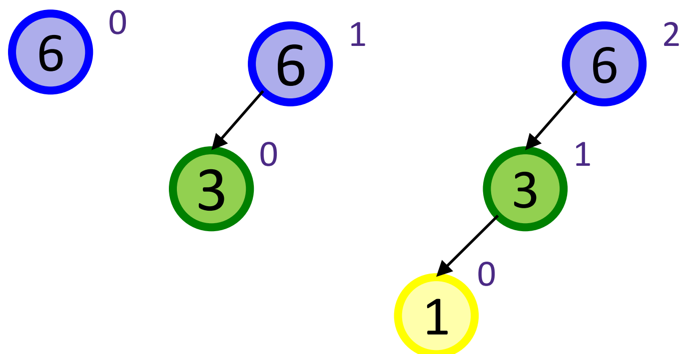
Case #1: Example

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)

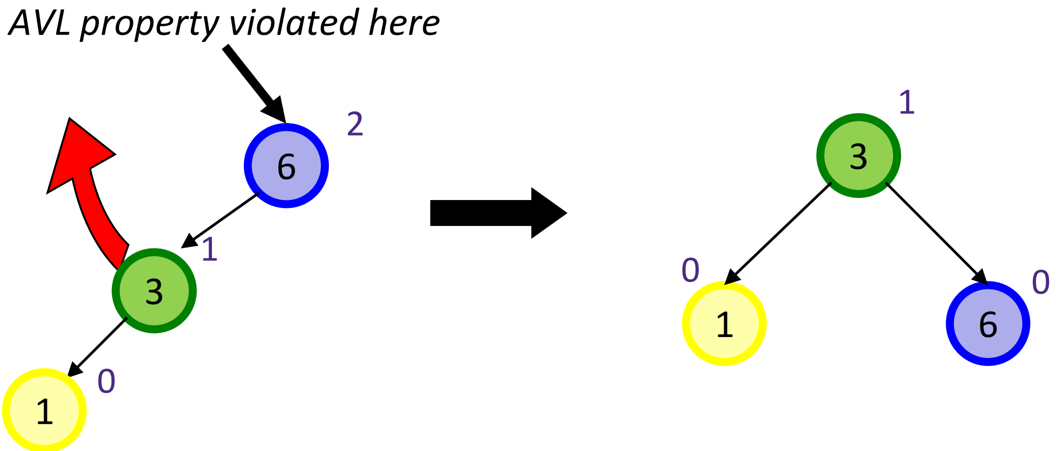
add(1)



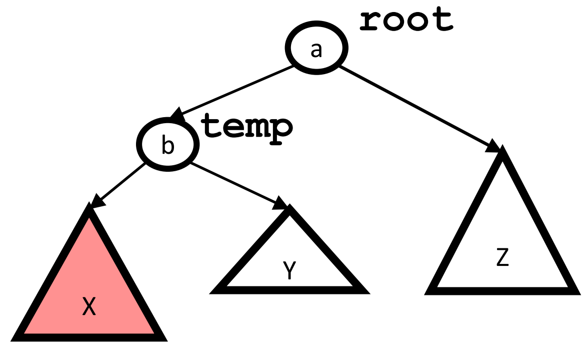
- Last add() violates balance property
- What is the only way to fix this?

Case #1 Fix: Apply "Single Rotation"

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



Case #1: Pseudocode



RotateRight

```
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
}
```

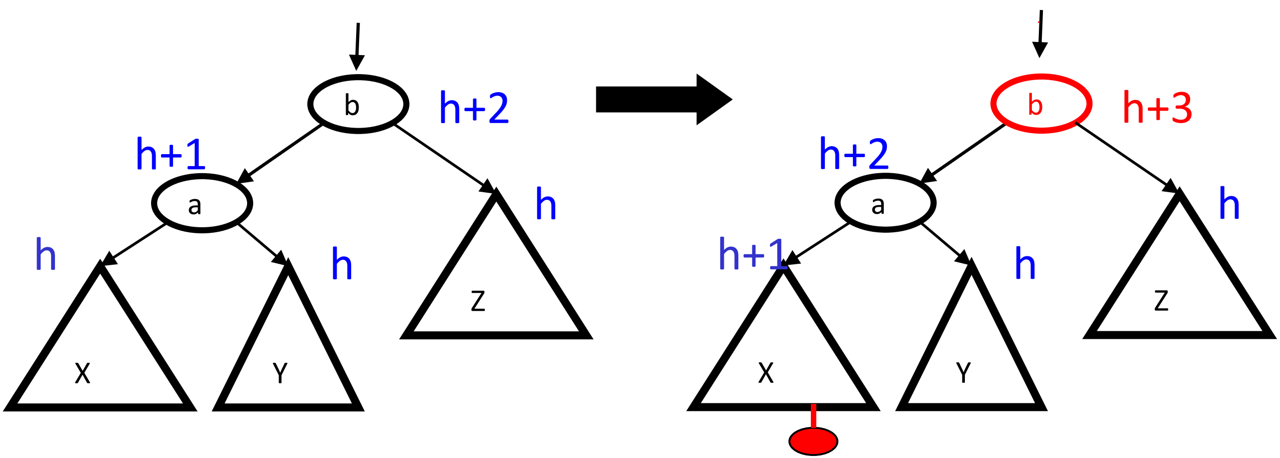
RotateWithLeftChild rotates the tree clockwise

Oval: a node in the tree
Triangle: a subtree

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

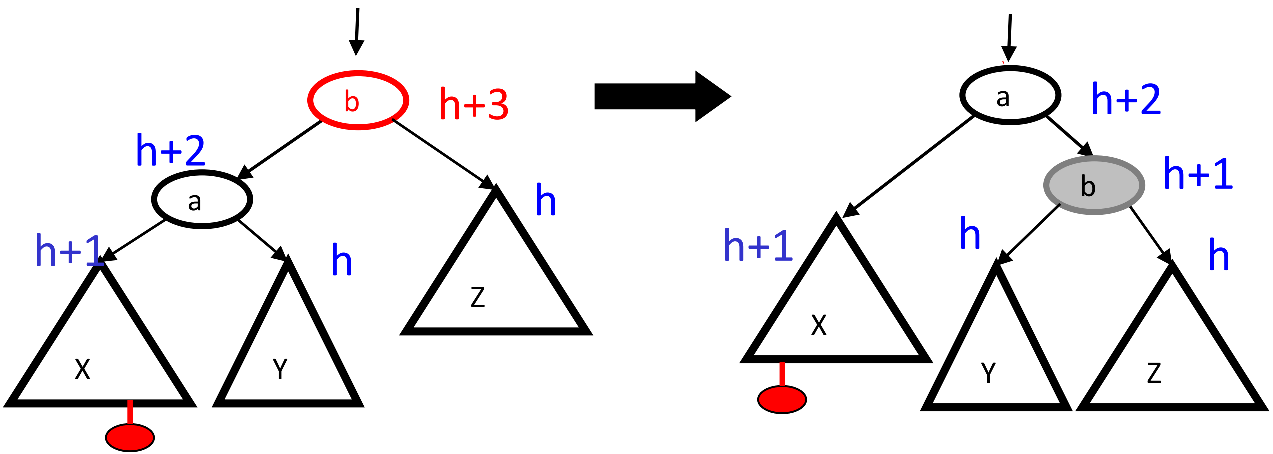
- Node is imbalanced due to insertion *somewhere* in **left-left grandchild**

- First we did the insertion, which would make ***b*** imbalanced



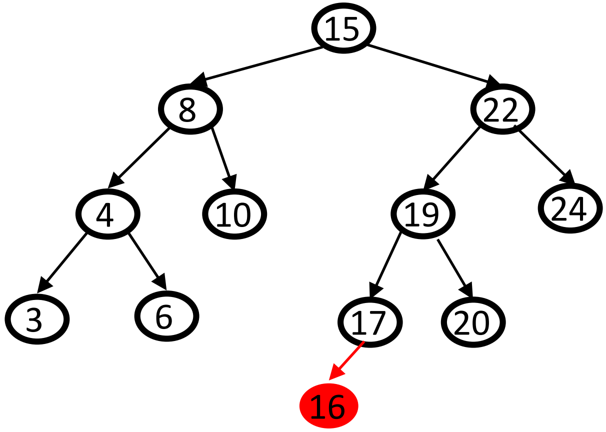
Case #1: Why It Works (2 of 2)

- 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)

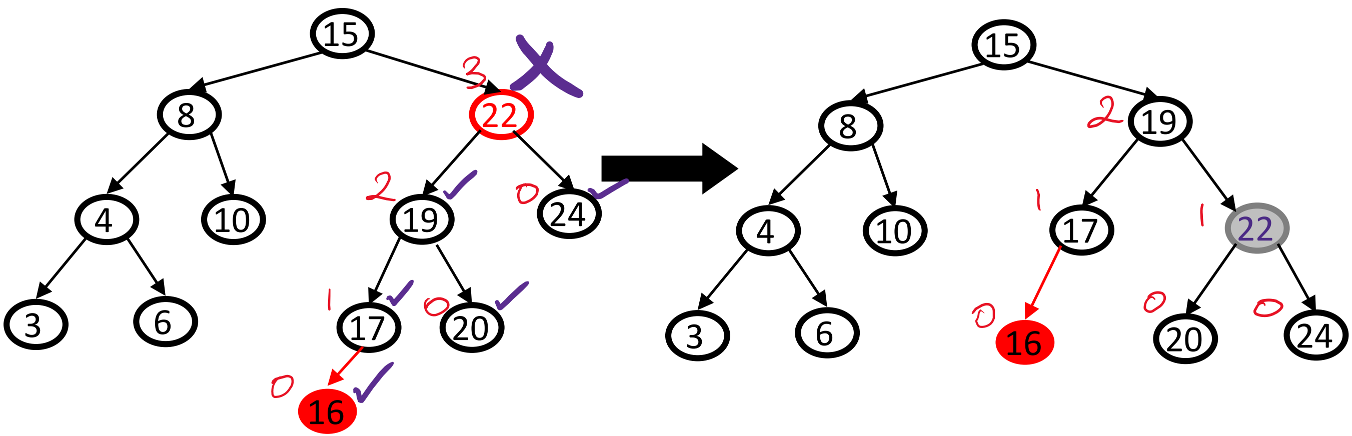
The insertion is in the:
1. left subtree of the left child of *b*
2. right subtree of the left child of *b*
3. left subtree of the right child of *b*
4. right subtree of the right child of *b*



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*

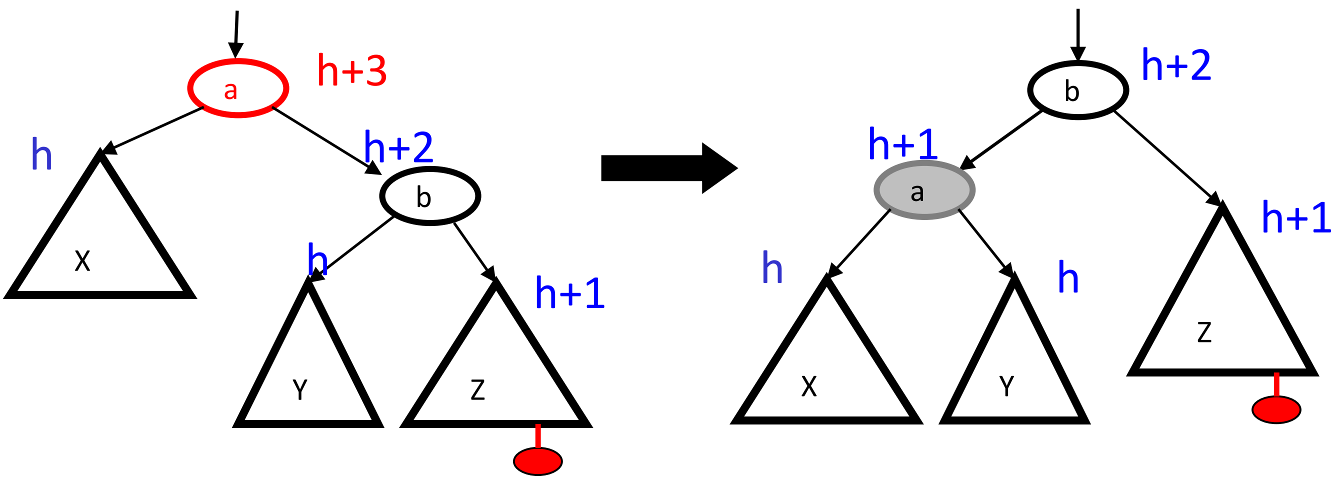


Case #1 ≈ Case #4

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

aka RotateLeft

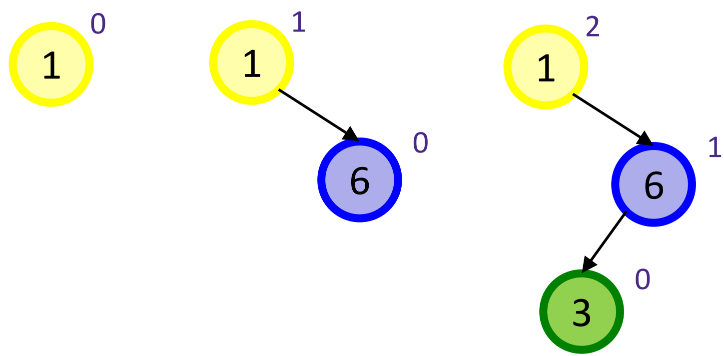
Case #3: Example

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*

Insert(1)

Insert(6)

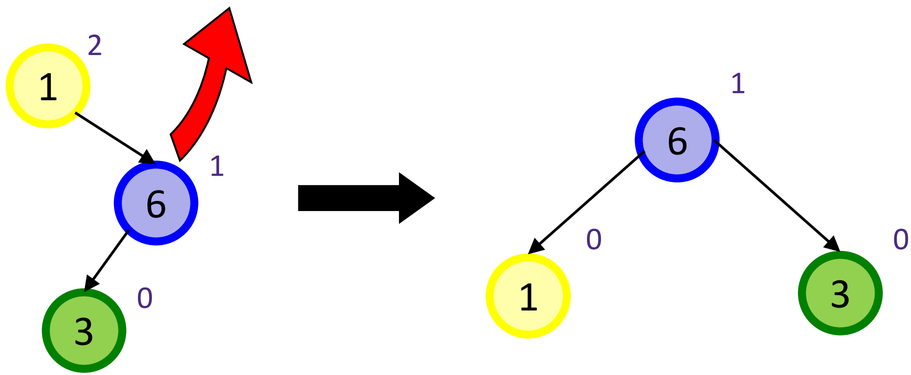
Insert(3)



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

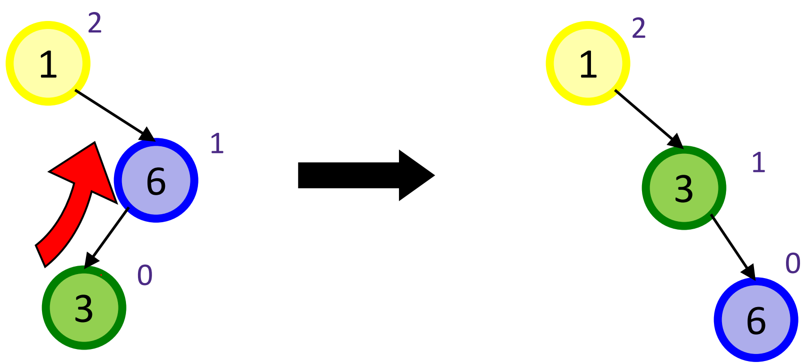
Case #3: Wrong Fix #1

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



Case #3: Wrong Fix #2

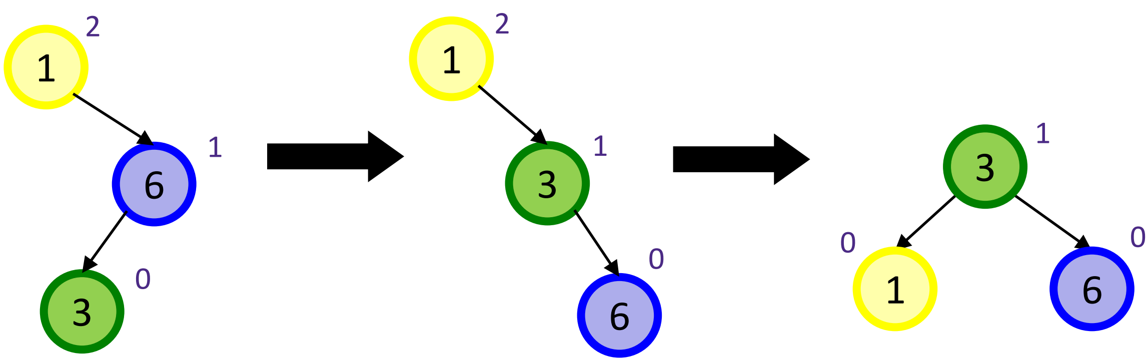
- **Second wrong idea**: single rotation on the child of the unbalanced node
  - Doesn't actually fix anything!



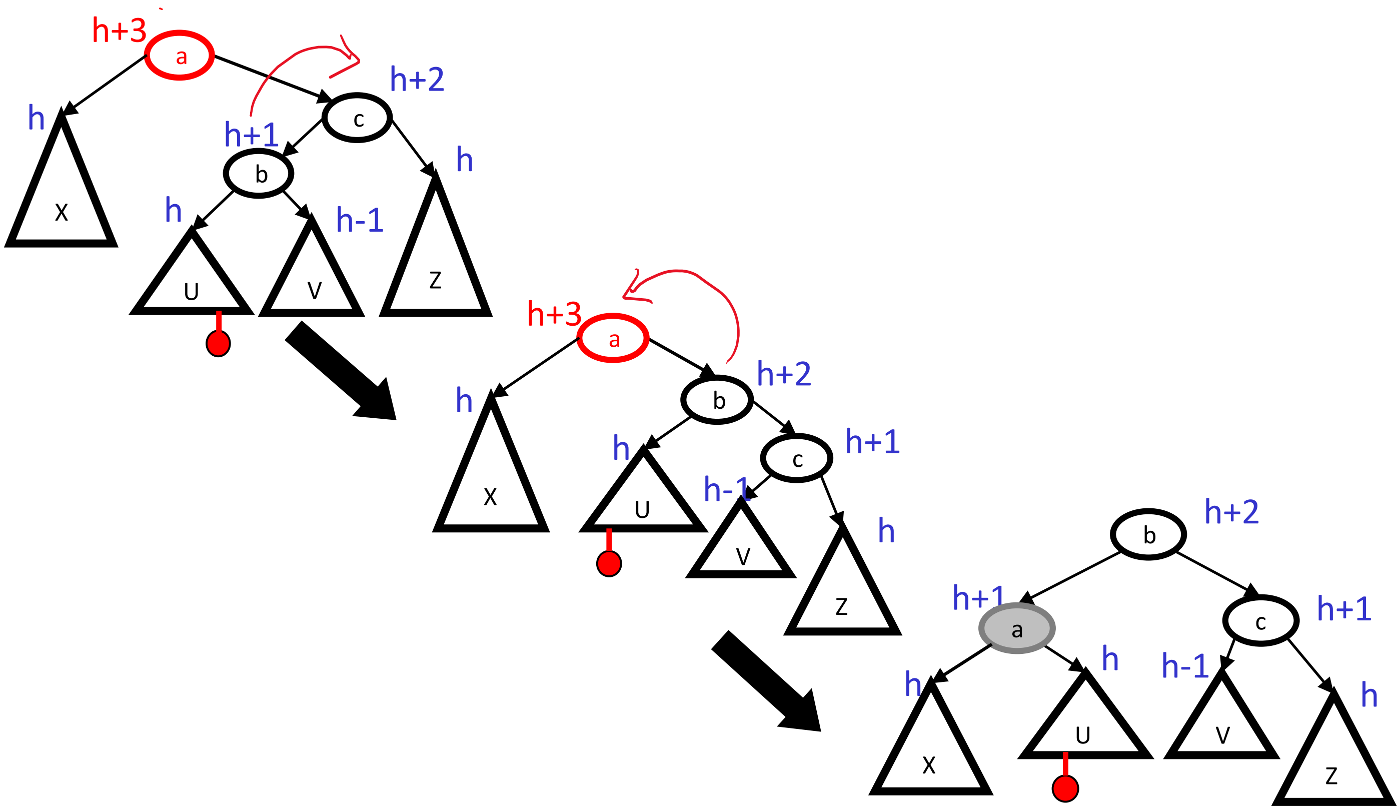
Case #3: Sometimes Two Wrongs Make a Right ☺

- 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

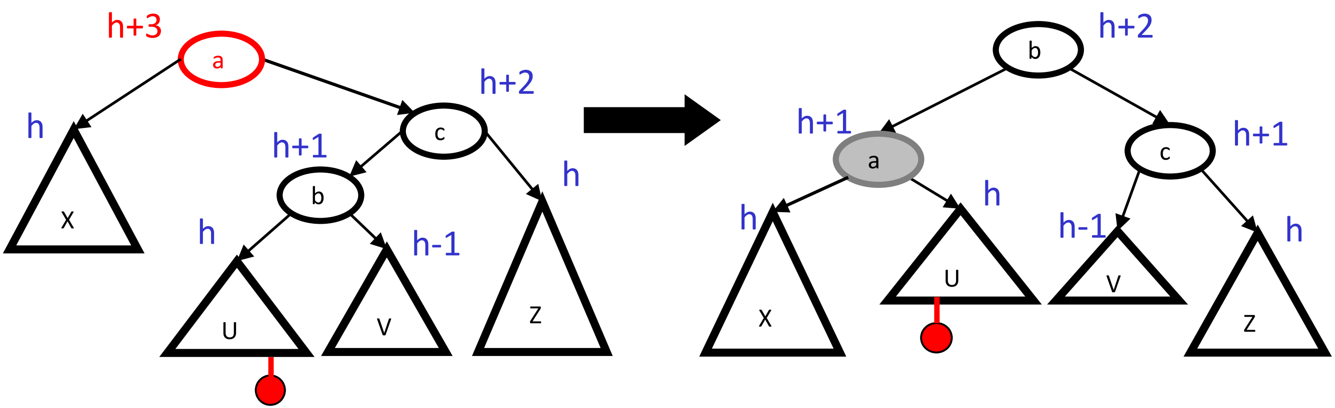


Case #3: Why It Works



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

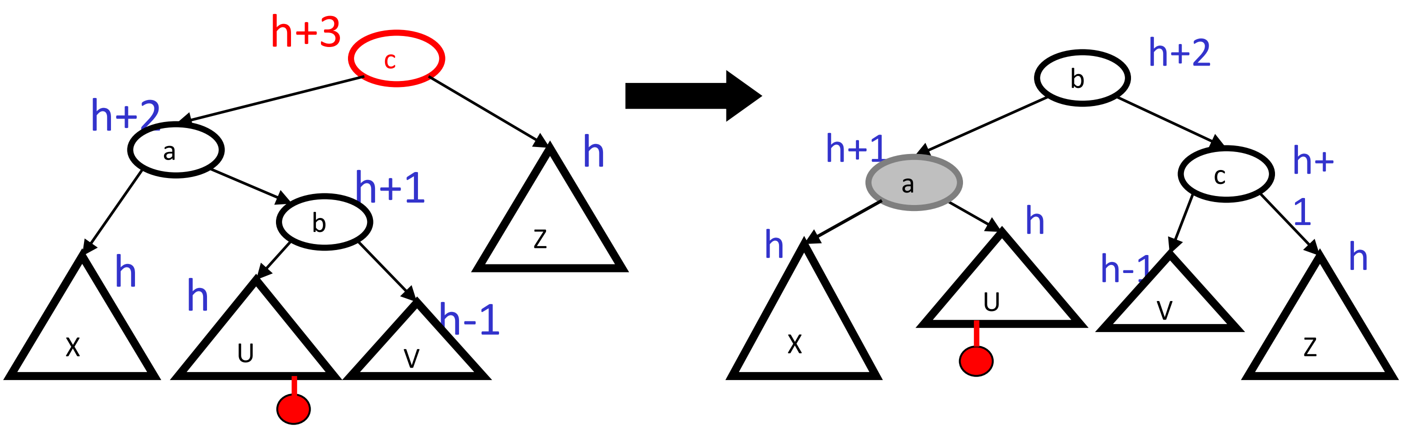
```
void DoubleRotateWithRightChild(Node root) {
  RotateWithLeftChild(root.right)
  RotateWithRightChild(root)
}
```

can also just update the pointers directly

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



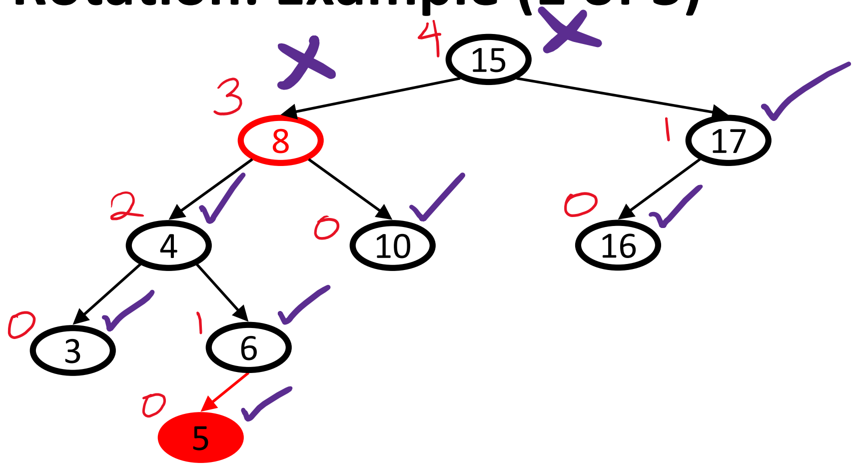
AVL add(): Summary

- 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 rotation, the smallest-unbalanced subtree has the same height as before insertion
  - So all ancestors are now balanced

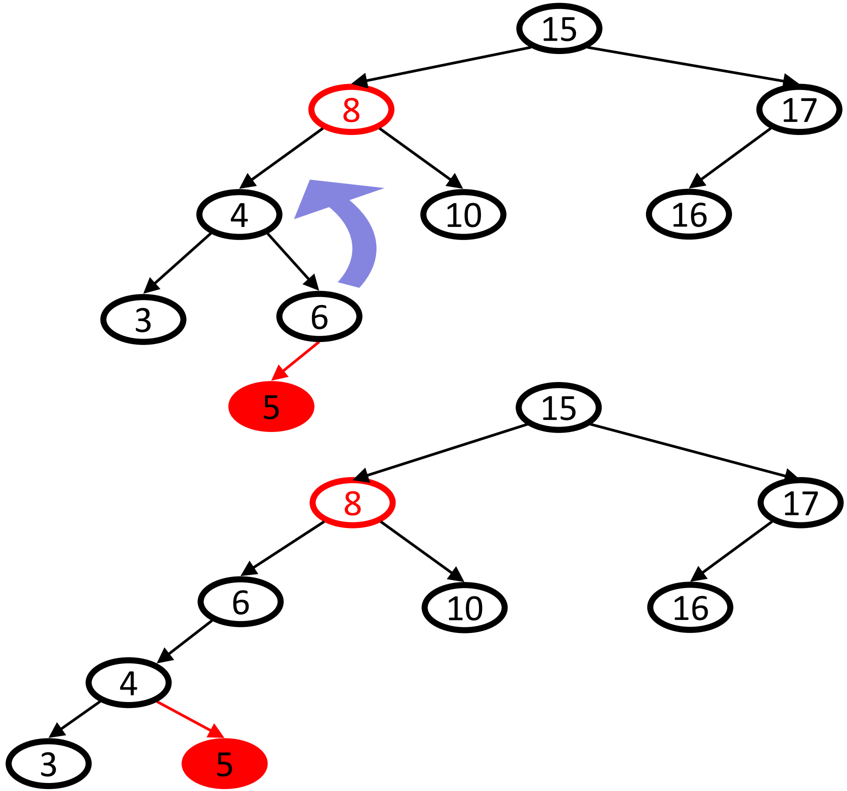
Lecture Outline

- AVL Tree
  - Bounding a BST's height
  - *(Proving the AVL tree's height bound)*
  - Find
  - Add
    - ***(Add Exercises)***
  - Remove
  - Wrapup

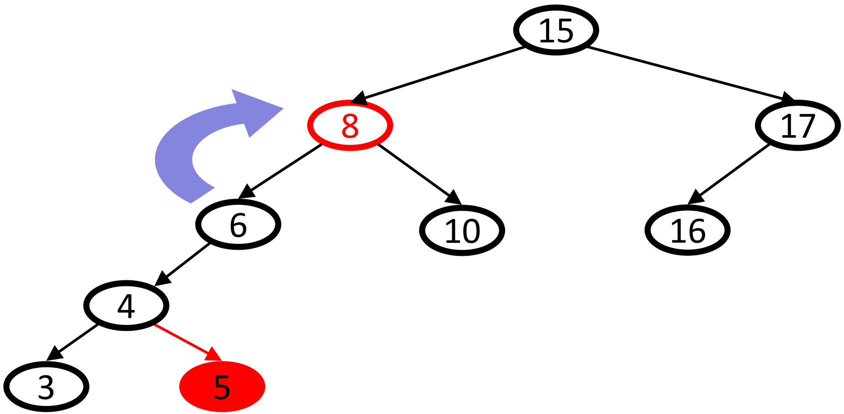
Double Rotation: Example (1 of 3)

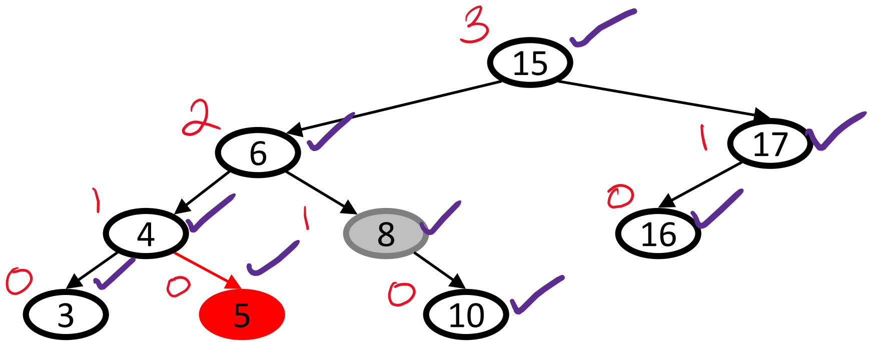


Double Rotation: Example (2 of 3)



Double Rotation: Example (3 of 3)



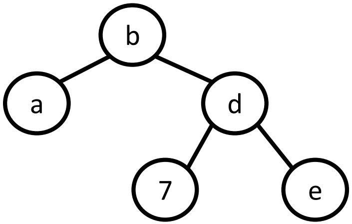


Student Activity #1: add() into an AVL tree

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

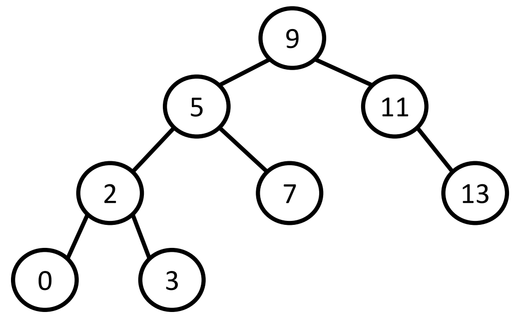
Student Activity #1: Answer

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



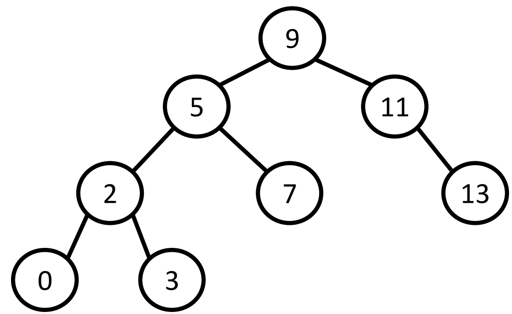
Student Activity #2: Single and Double Rotations

- Inserting which integer values would cause this tree to need a:
  - Single Rotation?
  - Double Rotation?
  - No Rotation?



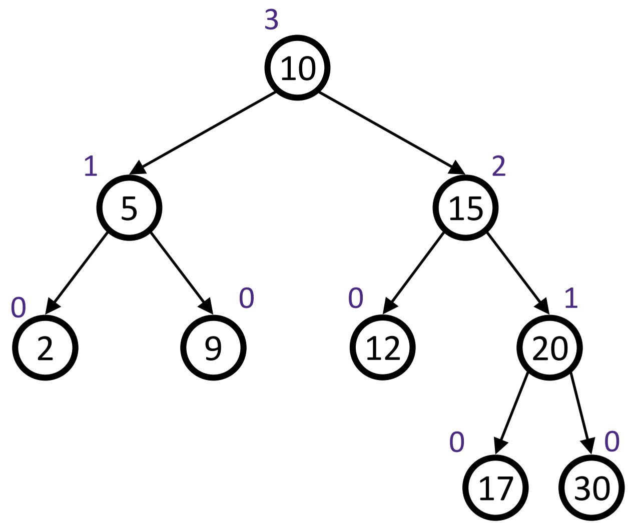
Student Activity #2: Answer

- Inserting which integer values would cause this tree to need a:
  - Single Rotation? 1, 14
  - Double Rotation? 4, 12
  - No Rotation? 6, 8, 10,



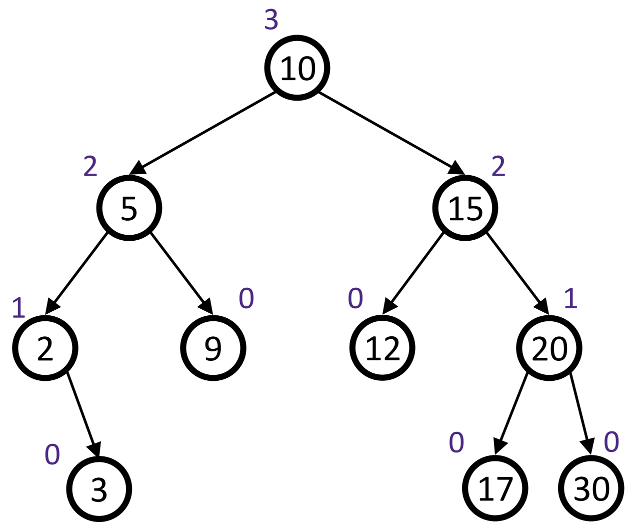
Student Activity #3: Add Sequence (1 of 2)

- add(3)
  - Is the resultant tree balanced?
  - If not, how would you fix it?



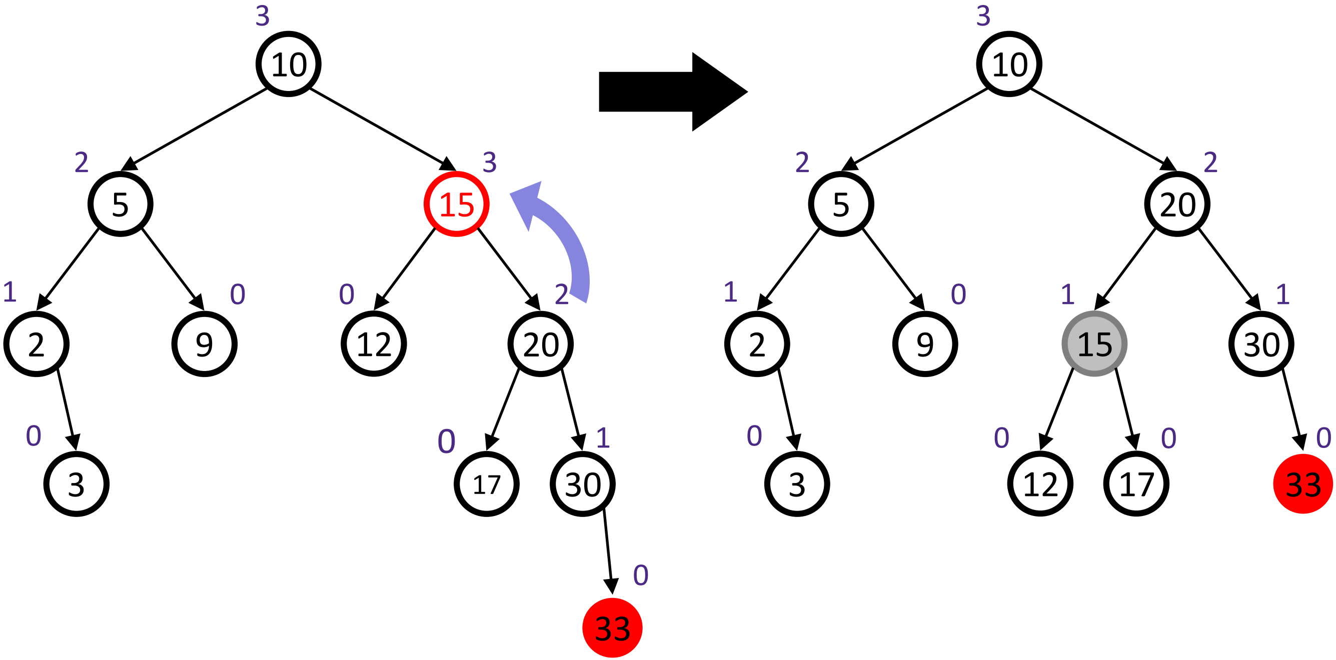
Student Activity #3: Add Sequence (2 of 2)

- Next, add(33)
  - Is the resultant tree balanced?
  - If not, how would you fix it?



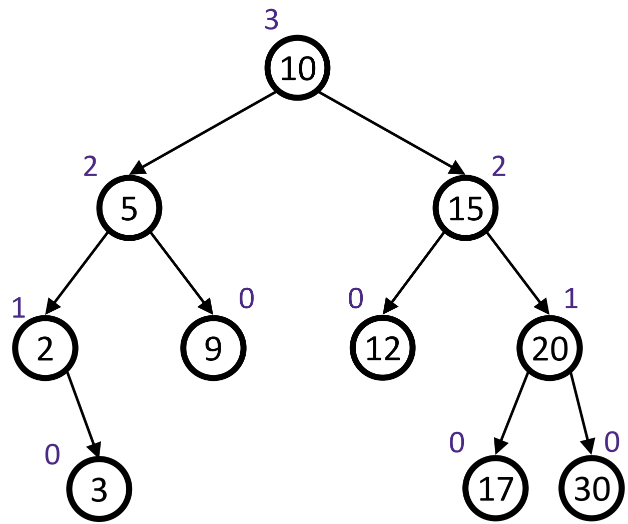
Student Activity #3: Answer

- Single rotation to the rescue!



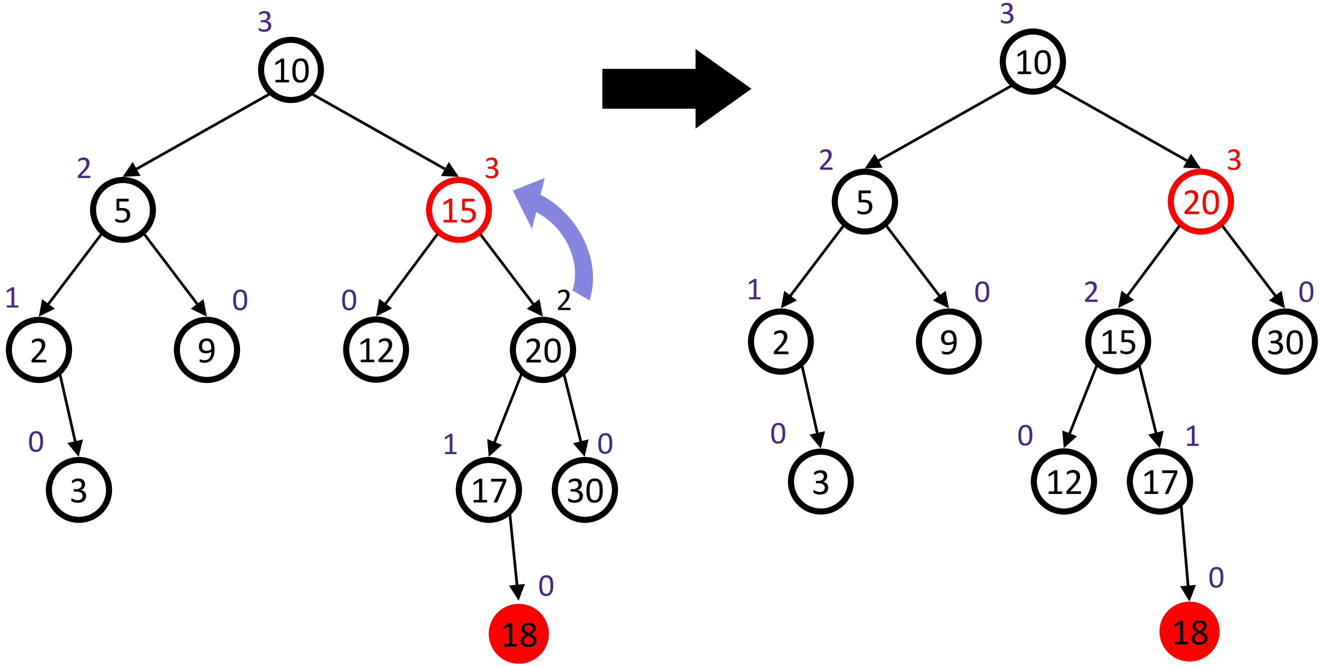
Student Activity #4: Harder Add Sequence (1 of 2)

- add(18)
  - Is the resultant tree balanced?
  - If not, how would you fix it?



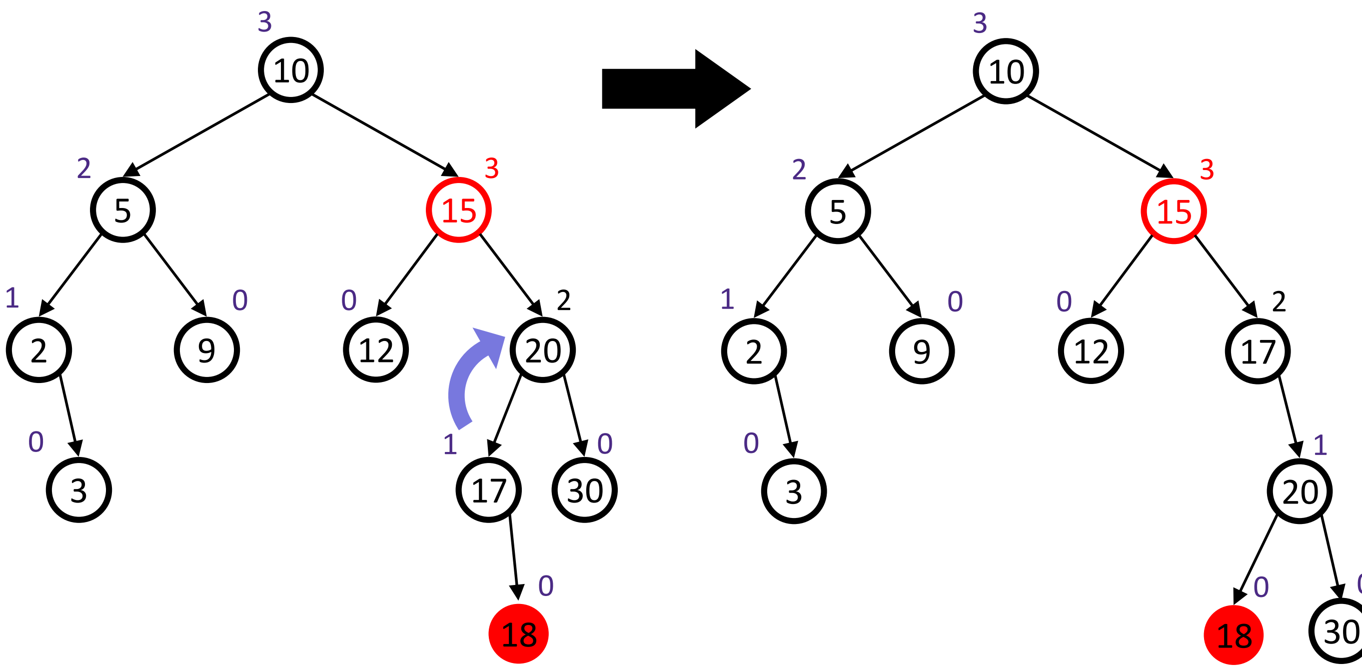
Student Activity #4: Harder Add Sequence (2 of 2)

- Single Rotation doesn't work



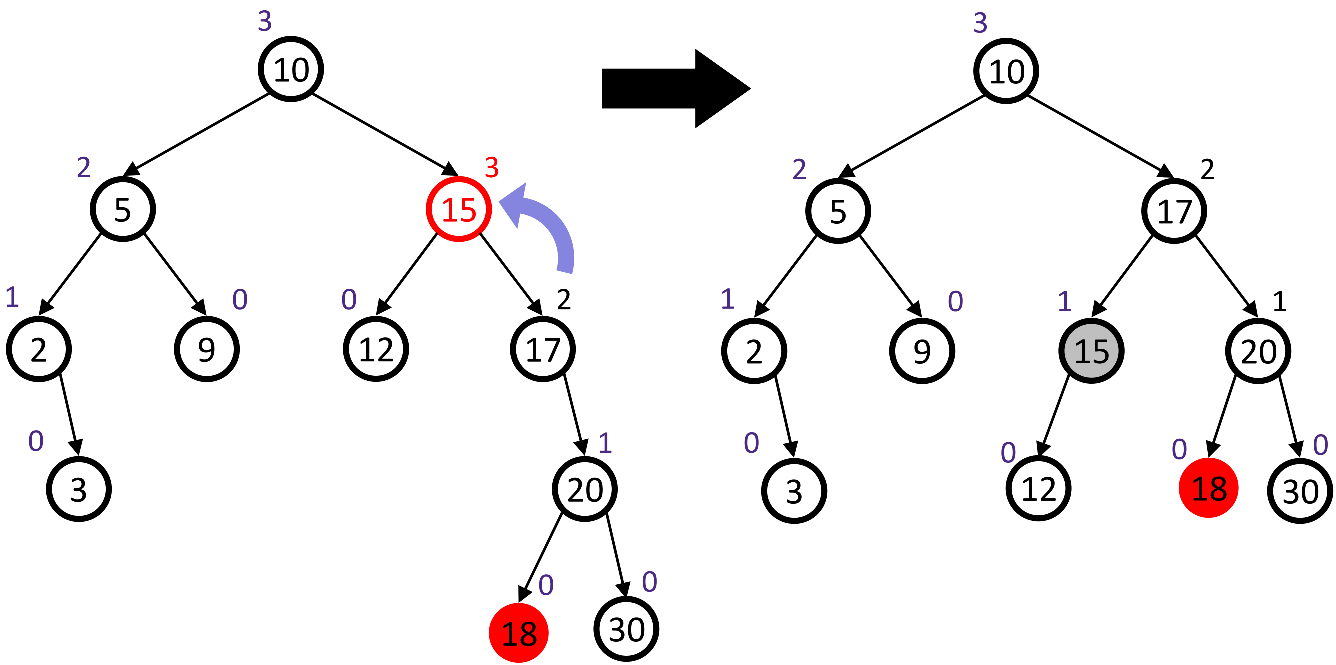
Student Activity #4: Answer (1 of 2)

- Double rotation, part 1



Student Activity #4: Answer (2 of 2)

- Double rotation, part 2

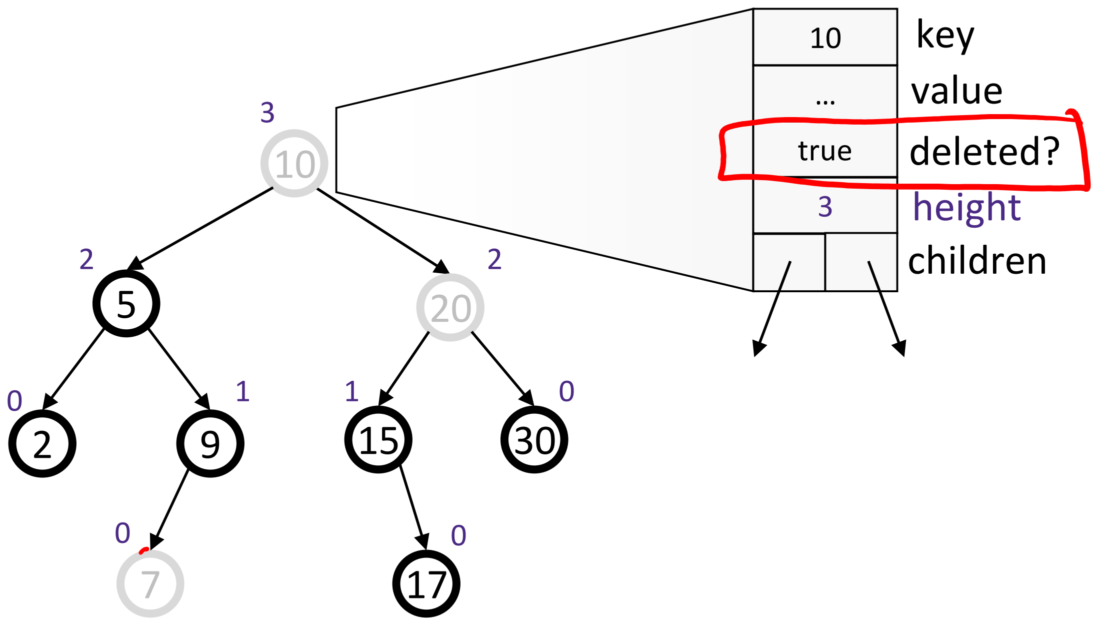


Lecture Outline

- AVL Tree
  - Bounding a BST's height
  - *(Proving the AVL tree's height bound)*
  - Find
  - Add
    - *(Add Exercises)*
  - **Remove**
  - Wrapup

AVL Remove: The Easy Way

- The "easy way" is lazy deletion



AVL Remove: The Hard Way

- We have several imbalance cases
  - See Weiss, 3rd ed. for more details

Lecture Outline

- AVL Tree
  - Bounding a BST's height
  - *(Proving the AVL tree's height bound)*
  - Find
  - Add
    - *(Add Exercises)*
  - Remove
  - **Wrapup**

AVL Tree Operations (1 of 2)

- AVL find:
  - Same as BST find $\Theta(h)$ . . .
  - Worst-case complexity:
    - Tree is balanced! So can also say $\Theta(\log n)$
- AVL add:
  - First BST add, then check balance and potentially “fix” the AVL tree
  - Four different imbalance cases
  - Worst-case complexity:
    - Tree starts and ends balanced
    - A rotation is O(1) and there’s an O(log n) path to root

AVL Tree Operations (2 of 2)

- AVL remove
  - We suggest lazy deletion
    - Worst-case complexity: $\Theta(\log n)$
  - Deletion requires more rotations than insert; but worst-case complexity still O(log n)

CHAPTER 4/TREES

Deletion in AVL trees is somewhat more complicated than insertion, and is left as an exercise. Lazy deletion is probably the best strategy if deletions are relatively infrequent.

4.5. Splay Trees

We now describe a relatively simple data structure, known as a *splay tree*, that guarantees

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
  - Compared to other balanced BSTs (eg, Red-Black trees), the constants aren't great
  - Most large data sets require database-like systems on disk, and thus use other structures (e.g., B-trees, our next data structure)