

CSE 332 Summer 2026

Lecture 7: AVL Trees

Michael Whitmeyer

<http://www.cs.uw.edu/332>

Reminder: Dictionary (Map) ADT

- Contents:
 - Sets of key+value pairs
 - Keys must be comparable
- Operations:
 - **insert**(key, value)
 - Adds the (key,value) pair into the dictionary
 - If the key already has a value, overwrite the old value
 - Consequence: Keys cannot be repeated
 - **find**(key)
 - Returns the value associated with the given key
 - **delete**(key)
 - Remove the key (and its associated value)

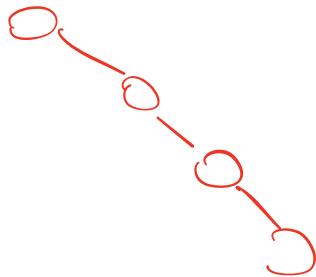
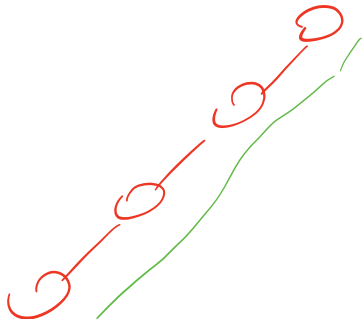
Naïve attempts

Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Heap	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree (worst)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
<u>AVL Tree</u>	<u>$\Theta(\log n)$</u>	<u>$\Theta(\log n)$</u>	<u>$\Theta(\log n)$</u>

Worst Case Analysis

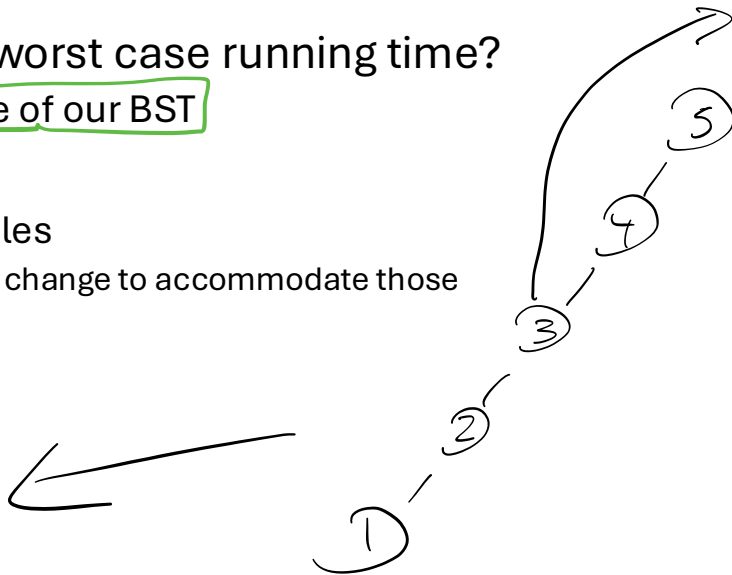
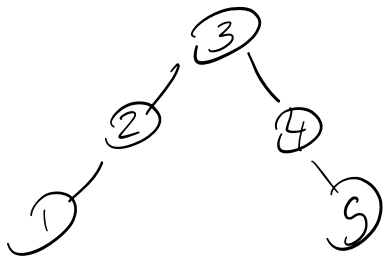
- For each of Find, insert, Delete:
 - Worst case running time matches height of the tree
- What is the maximum height of a BST with n nodes?

- $\Theta(n)$



Improving the worst case

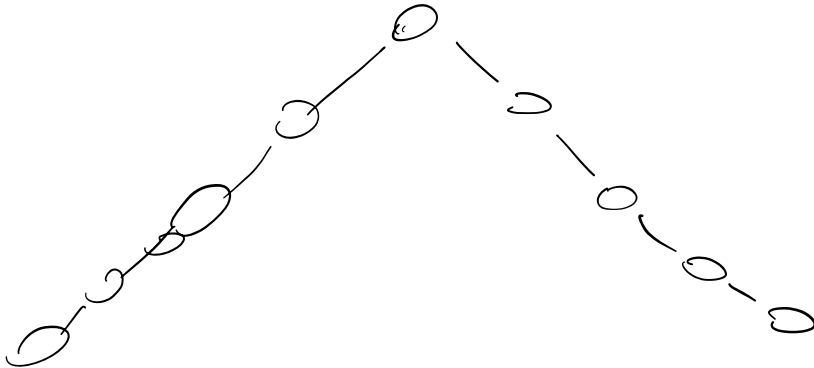
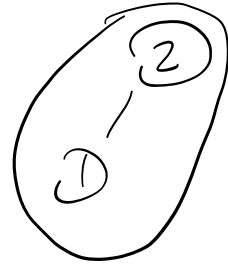
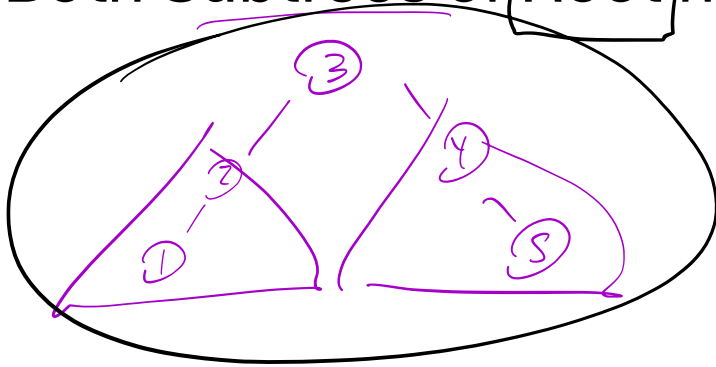
- How can we get a better worst case running time?
 - Add rules about the shape of our BST
- AVL Tree
 - A BST with some shape rules
 - Find/Insert/Delete need to change to accommodate those



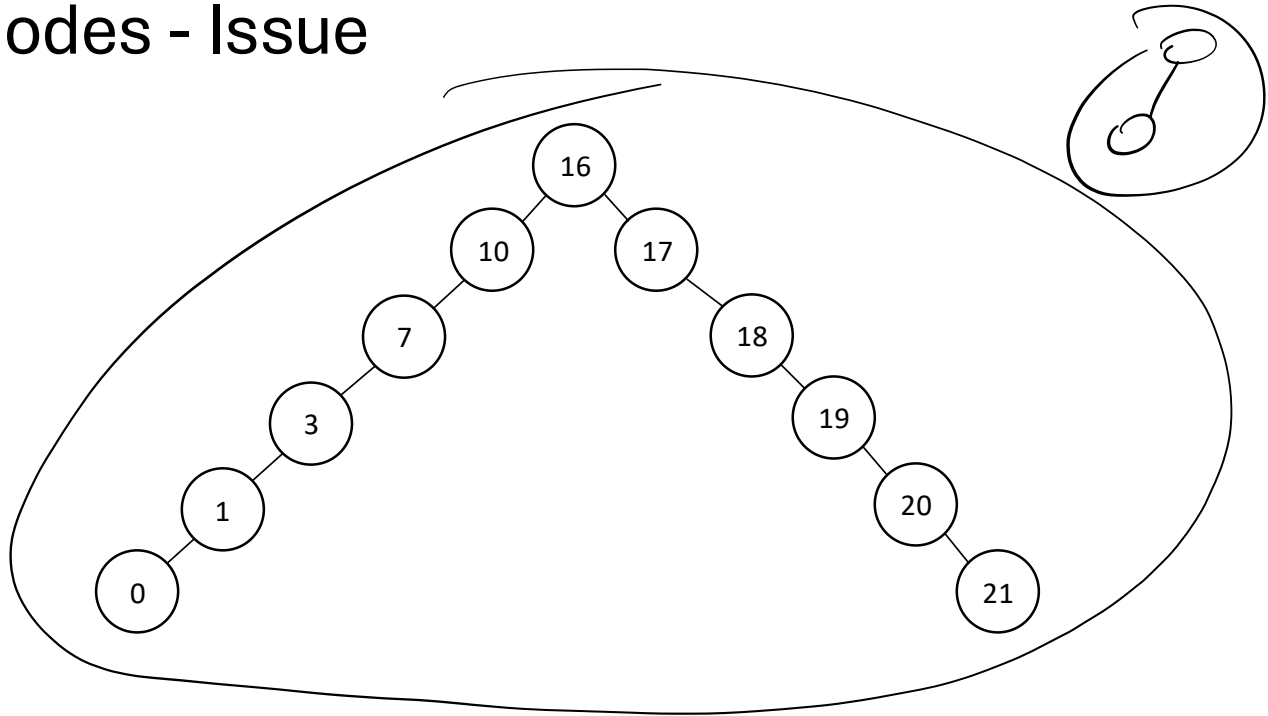
“Balanced” Binary Search Trees

- We get better running times by having “shorter” trees
- Trees get tall due to them being “sparse” (many one-child nodes)
- Idea: modify how we insert/delete to keep the tree more “full”
 - Encourage Branches!

Idea 1: Both Subtrees of Root have same # Nodes

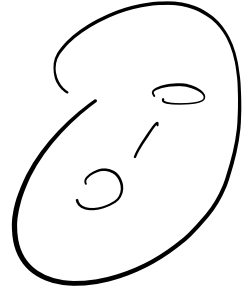
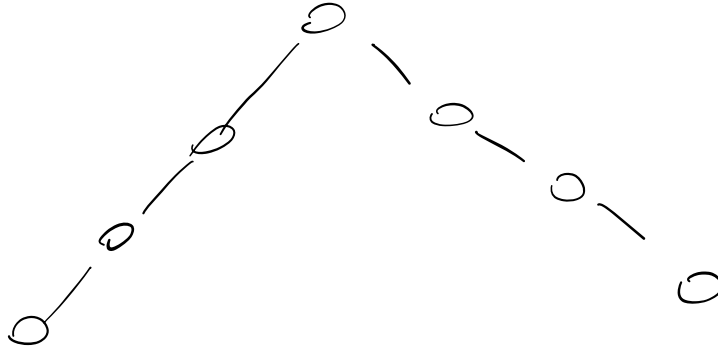


Idea 1: Both Subtrees of Root have same # Nodes - Issue



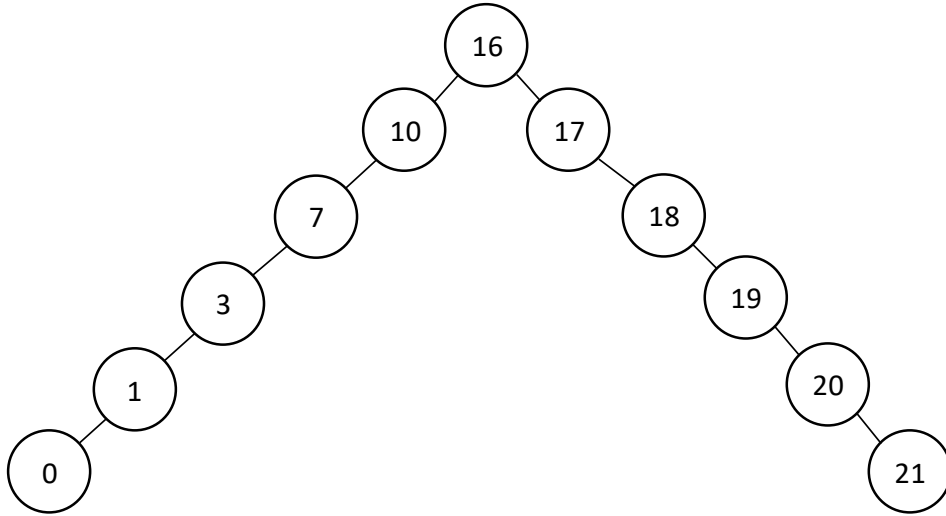
Idea 2: Both Subtrees of ~~Root~~ have same height

every node



idea

Idea 2: Both Subtrees of Root have same height - Issue



Idea 3: Both Subtrees of every Node have
same # Nodes



Idea 3: Both Subtrees of every Node have same # Nodes - Issue

Not all tree sizes are possible!

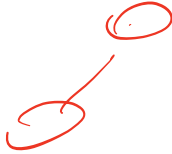
For example, cannot have a tree of size 2.

Idea 4: Both Subtrees of every Node have
same height

Idea 4: Both Subtrees of every Node have same height - Issue

Not all tree sizes are possible!

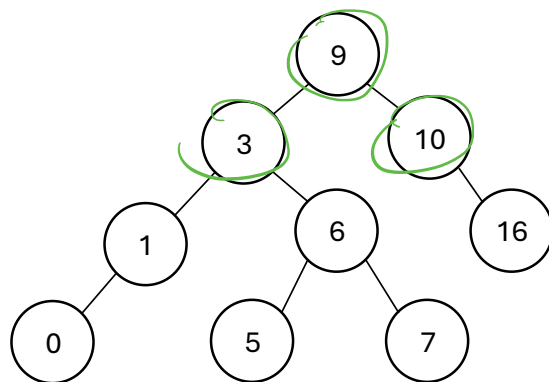
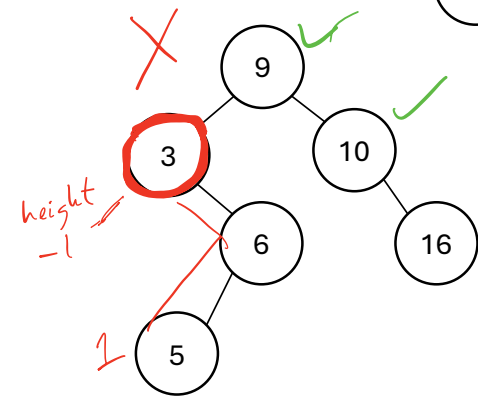
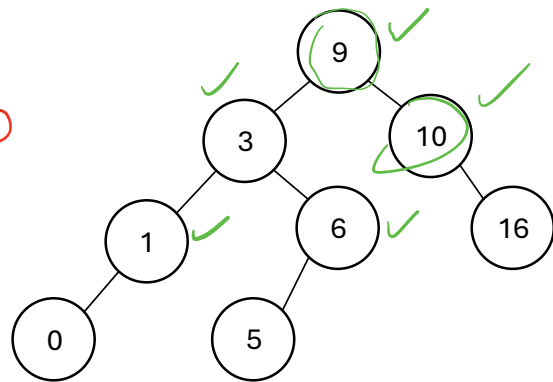
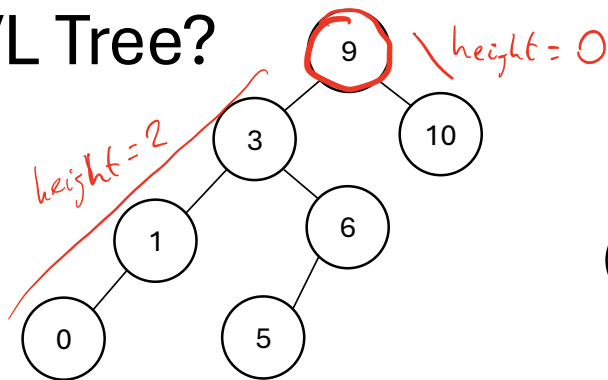
For example, cannot have a tree of size 2.



AVL Tree

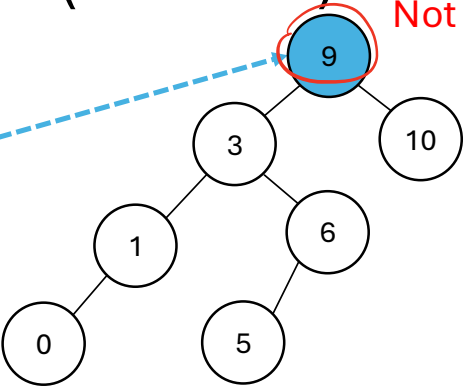
- A Binary Search tree that maintains that the left and right subtrees of every node have heights that differ by at most one.
 - height of left subtree and height of right subtree off by at most 1
 - Not too weak (ensures trees are short)
 - Not too strong (works for any number of nodes)
- Idea of AVL Tree:
 - When you insert/delete nodes, if tree is “out of balance” then modify the tree
 - Modification = “rotation”

Is it an AVL Tree?

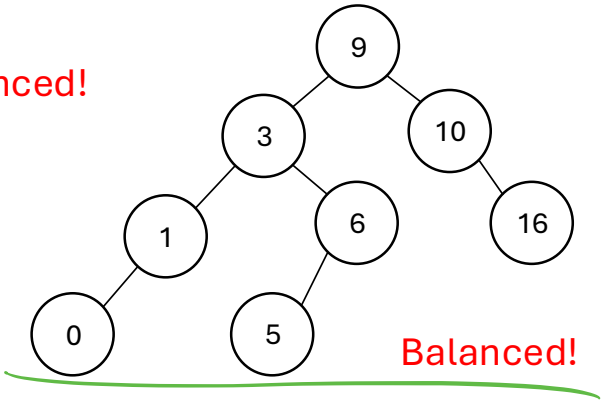


Is it an AVL Tree? (Answers)

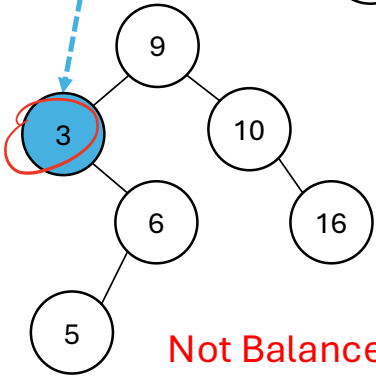
“Problem” Node
Its children’s heights differ by more than 1



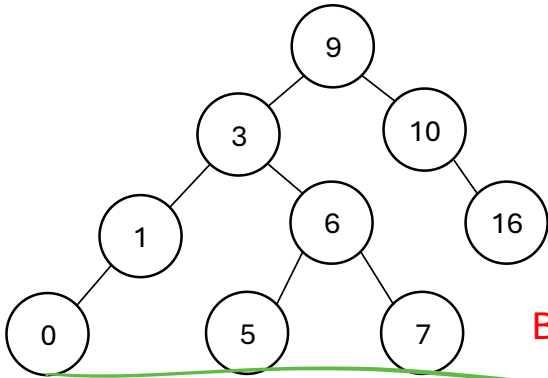
Not Balanced!



Balanced!



Not Balanced!

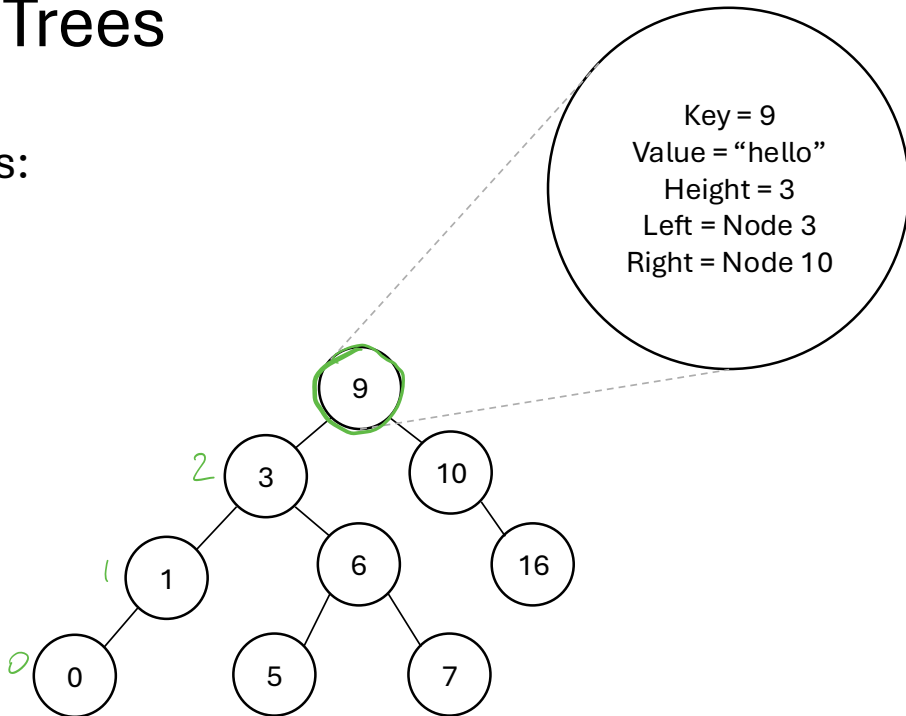


Balanced!

Using AVL Trees

- Each node has:

- Key
- Value
- Height
- Left child
- Right child

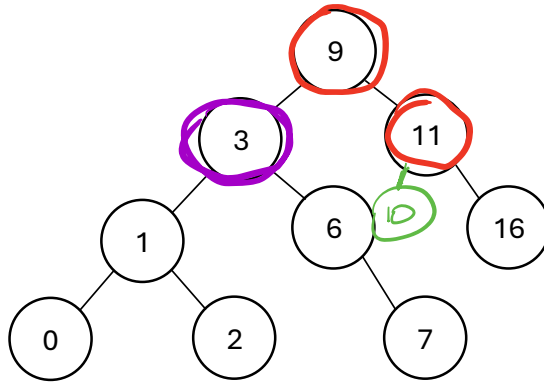


Inserting into an AVL Tree

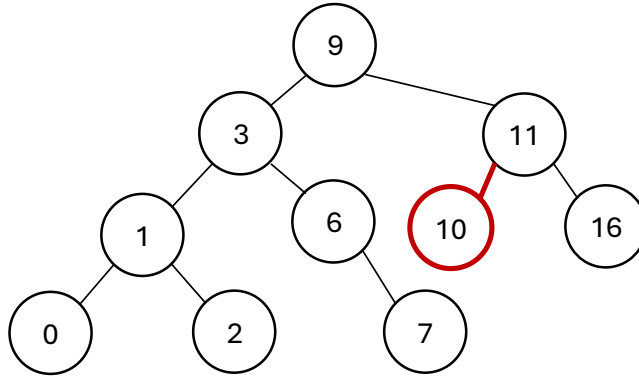


- Starts out the same way as BST:
 - “Find” where the new node should go
 - Put it in the right place (it will be a leaf)
 - Next check the balance
 - If the tree is still balanced, you’re done!
 - Otherwise we need to do rotations
- 
- 
- 

Insert Example (Insert 10)



Insert Example (Insert 10) - After

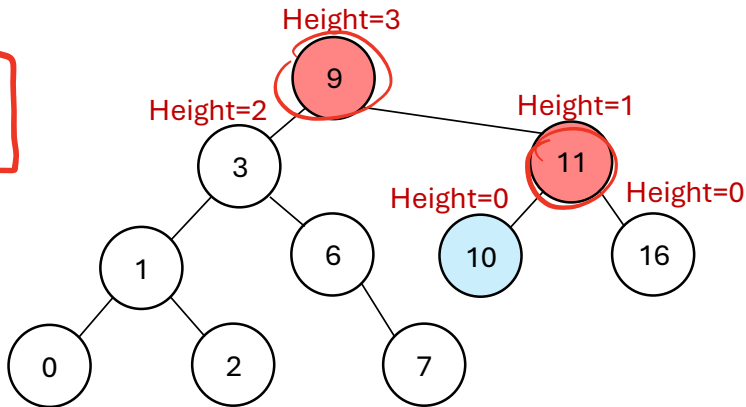


Insert 10 (After)

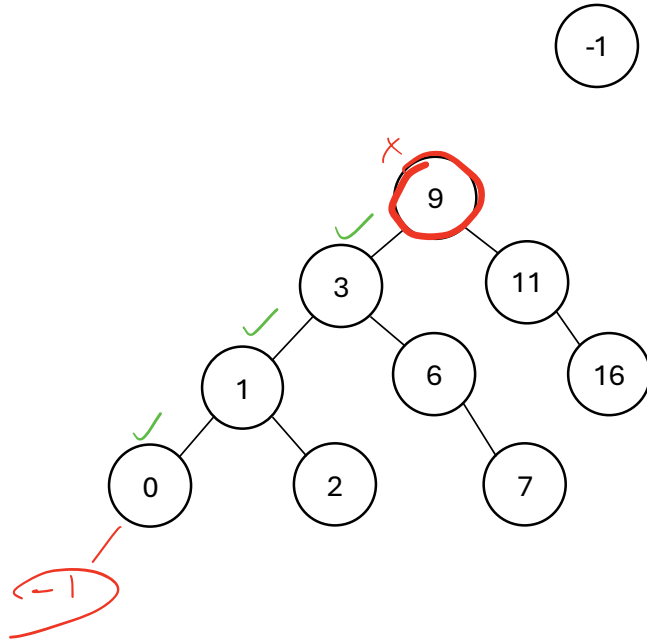
Is the tree still balanced?

Only need to check nodes in the path from root to the new node

Why? Tree was balanced before the insert, so unchanged subtrees cannot be unbalanced.



Insert Example (Insert -1)



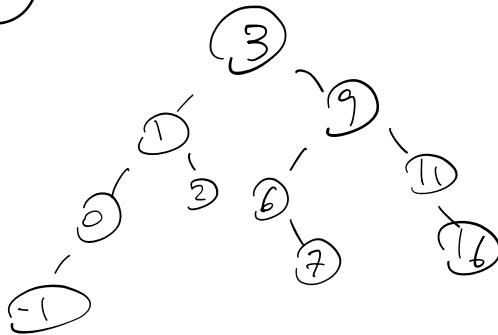
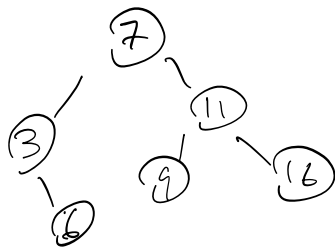
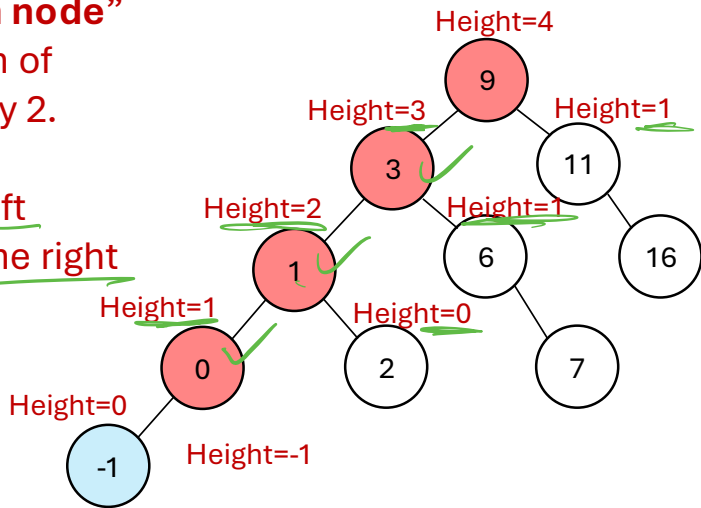
Insert -1, BST Insert Step

Not Balanced!

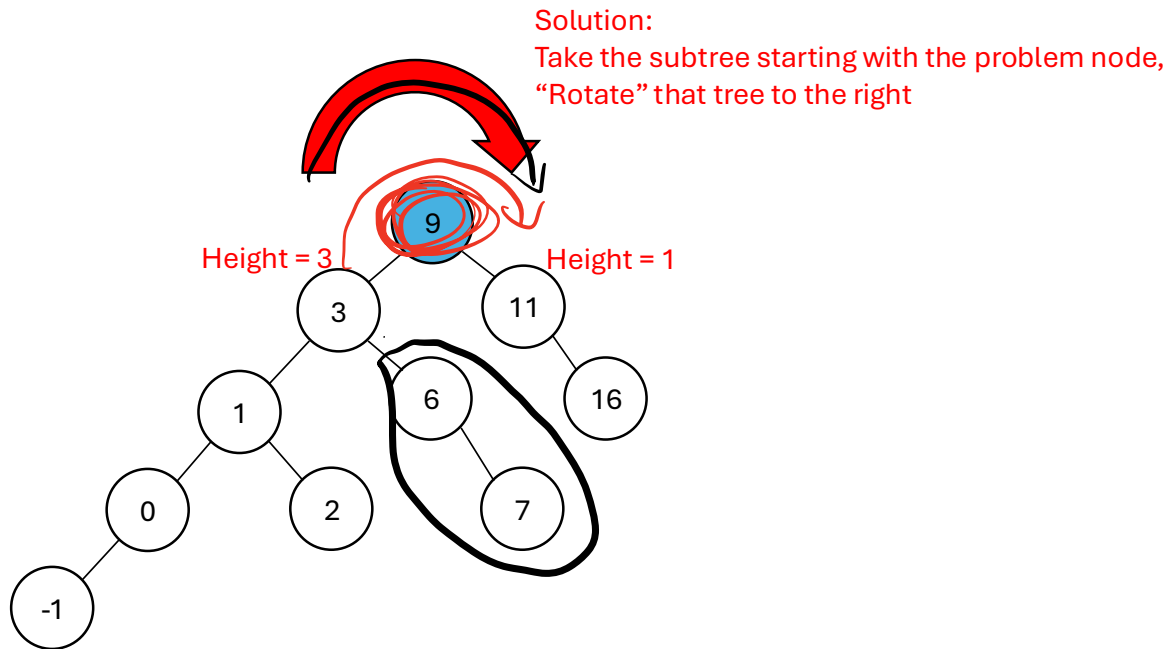
Node 9 is a **“problem node”**

Left and right children of node 9 are different by 2.

Idea: “shorten” the left subtree, “lengthen” the right

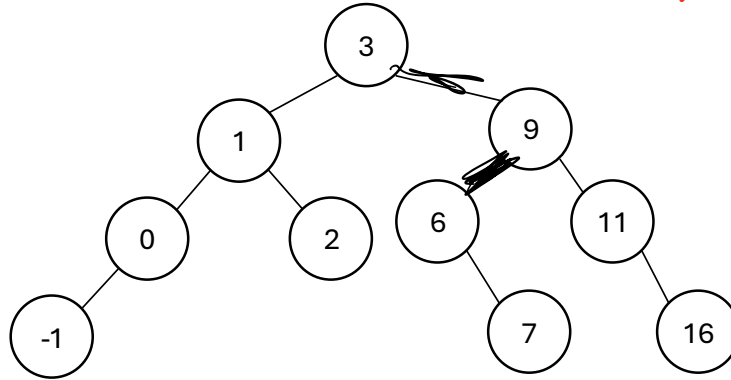


Not Balanced after Inserting -1



Final Result of Inserting -1

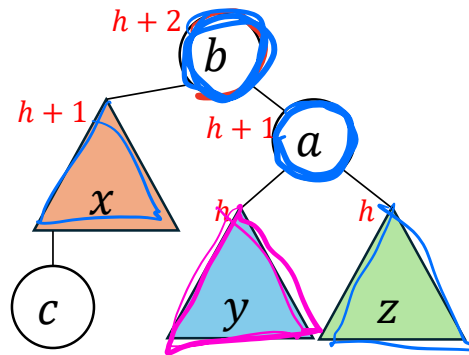
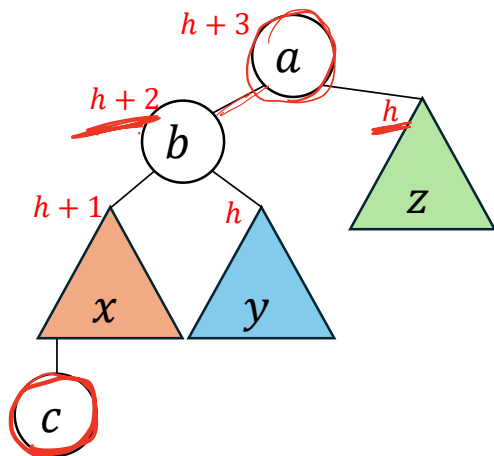
"right rotation at 9"



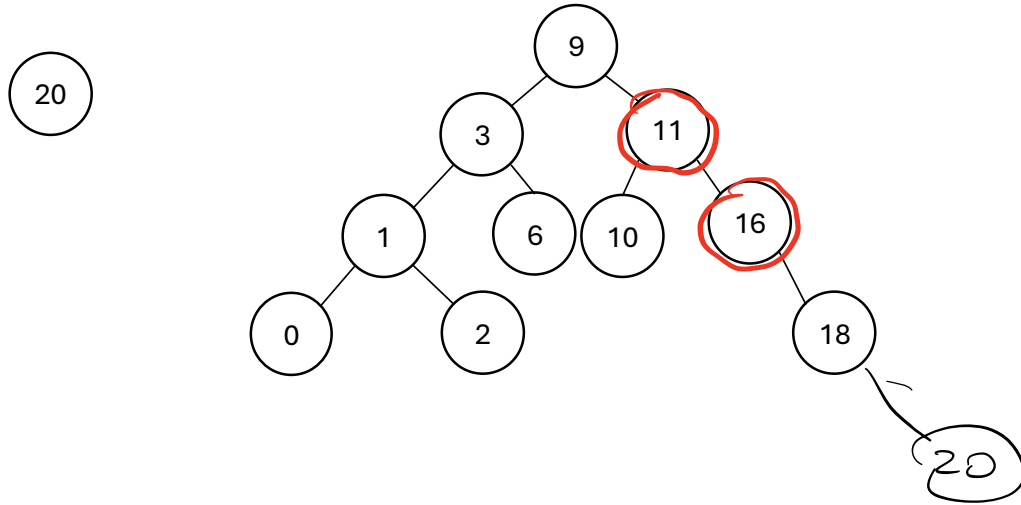
Right Rotation

- We just inserted c , node a is the deepest “problem” node
- Make the left child the new root
- Make the old root the right child of the new
- Make the new root’s right subtree the old root’s left subtree

$a.\text{left} = b.\text{right}$
 $b.\text{right} = a$



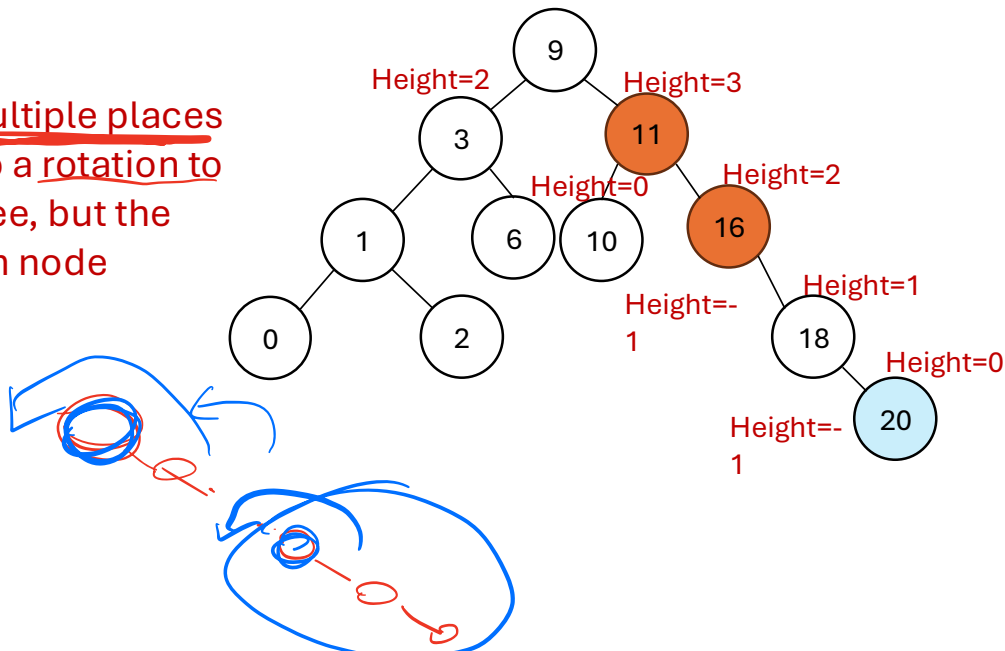
Insert Example (Insert 20)



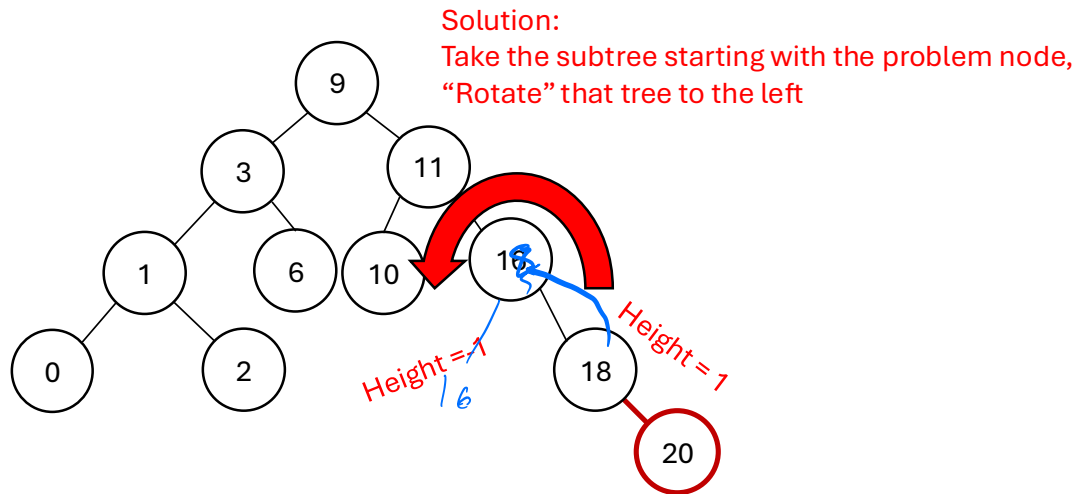
Not Balanced! Multiple Problem Nodes

Here, nodes 11 and 16 are problem nodes.

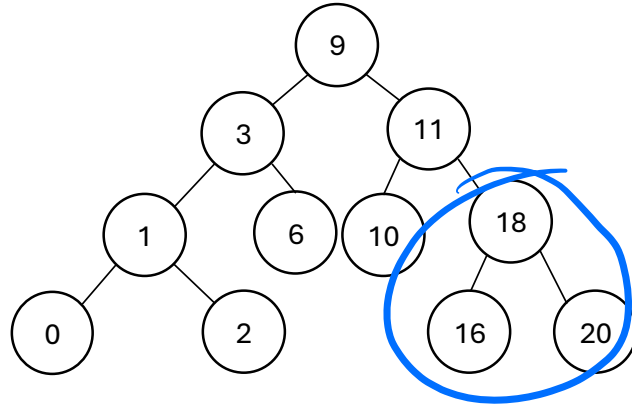
There may be multiple places where we can do a rotation to rebalance the tree, but the deepest problem node always works!



Not Balanced After inserting 20

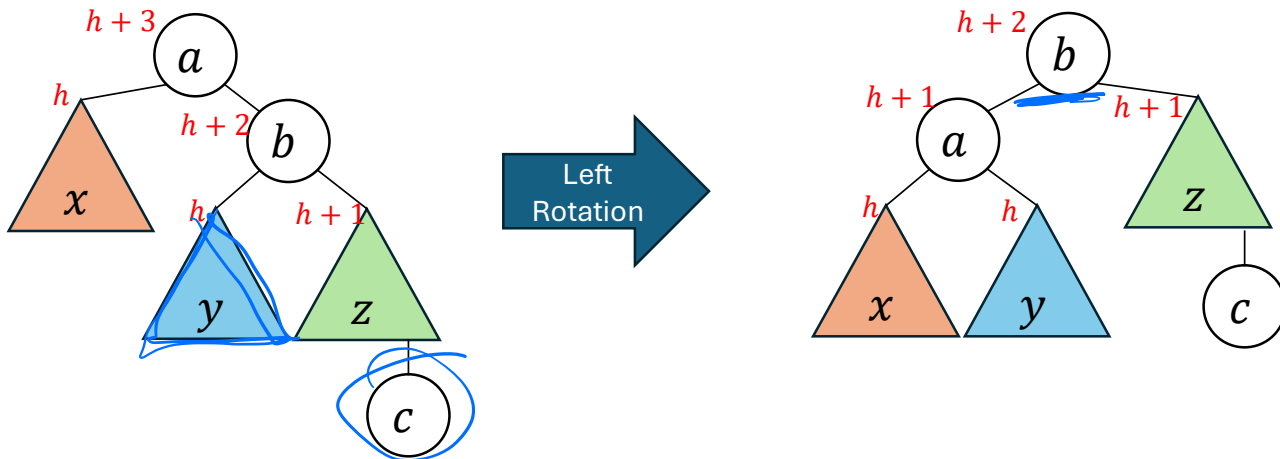


Final Result of Inserting 20



Left Rotation

- We just inserted c , node a is the deepest “problem” node
- Make the right child the new root
- Make the old root the left child of the new
- Make the new root’s left subtree the old root’s right subtree

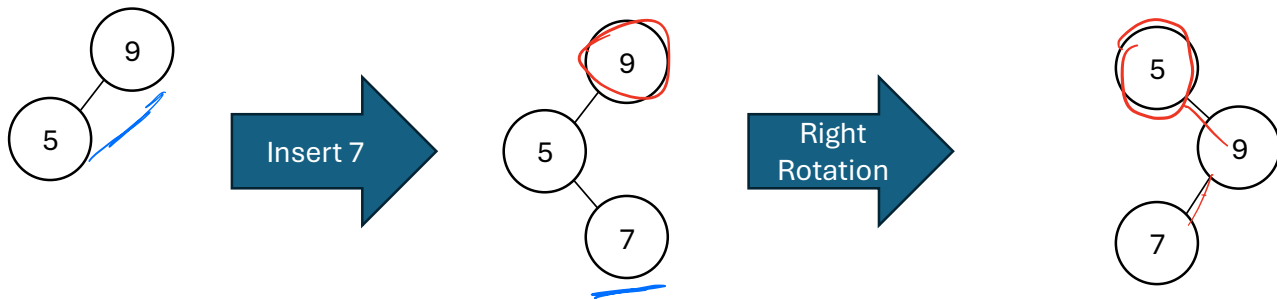


Insertion Story So Far

- After insertion, update the heights of the node's ancestors
- Check for unbalance
- If unbalanced then at the deepest unbalanced root:
 - If the left subtree was deeper then rotate right
 - If the right subtree was deeper then rotate left



This is incomplete!
There are some cases
where this doesn't work!



Insertion Cases

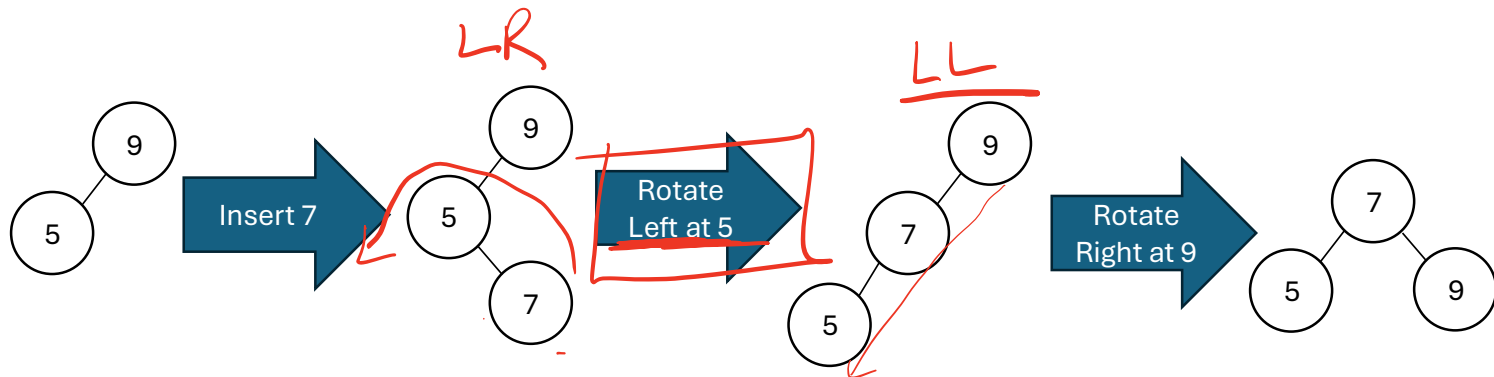
- After insertion, update the heights of the node's ancestors
- Check for unbalance
- If unbalanced then at the deepest unbalanced root:
 - Case LL: If we inserted in the **left** subtree of the **left** child then rotate right
 - Case RR: If we inserted in the **right** subtree of the **right** child then rotate left
 - Case LR: If we inserted into the **right** subtree of the **left** child then ???
 - Case RL: If we inserted into the **left** subtree of the **right** child then ???

Cases LR and RL require 2 rotations!

Case LR

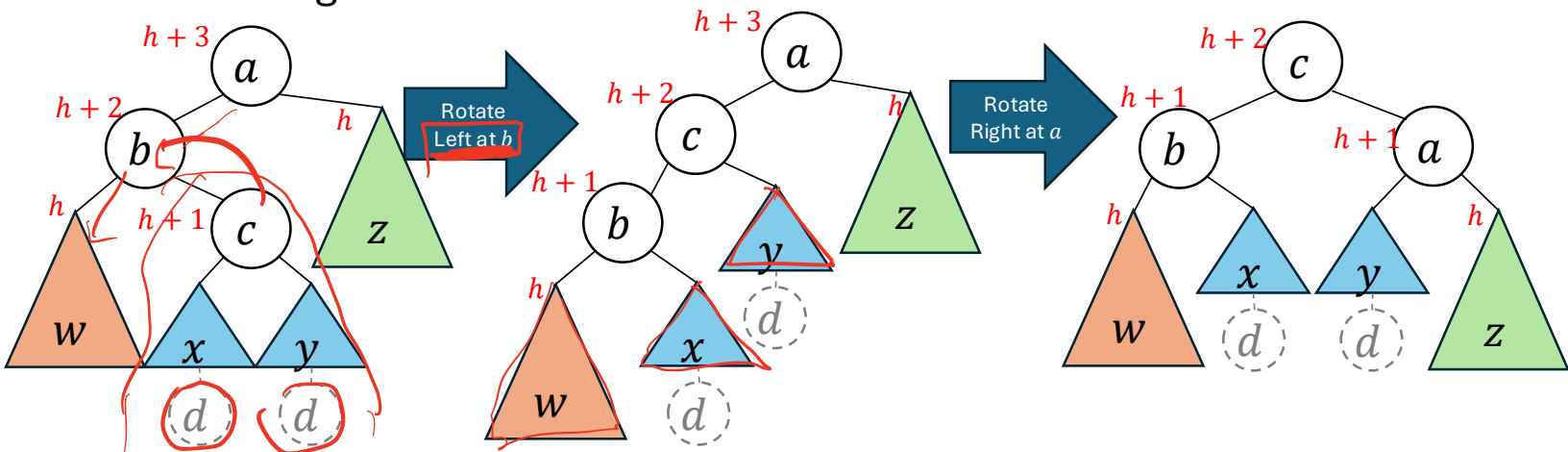
- From deepest problem node:
 - Rotate left at the left child
 - Rotate right at the problem node

inserted at right subtree of left child.



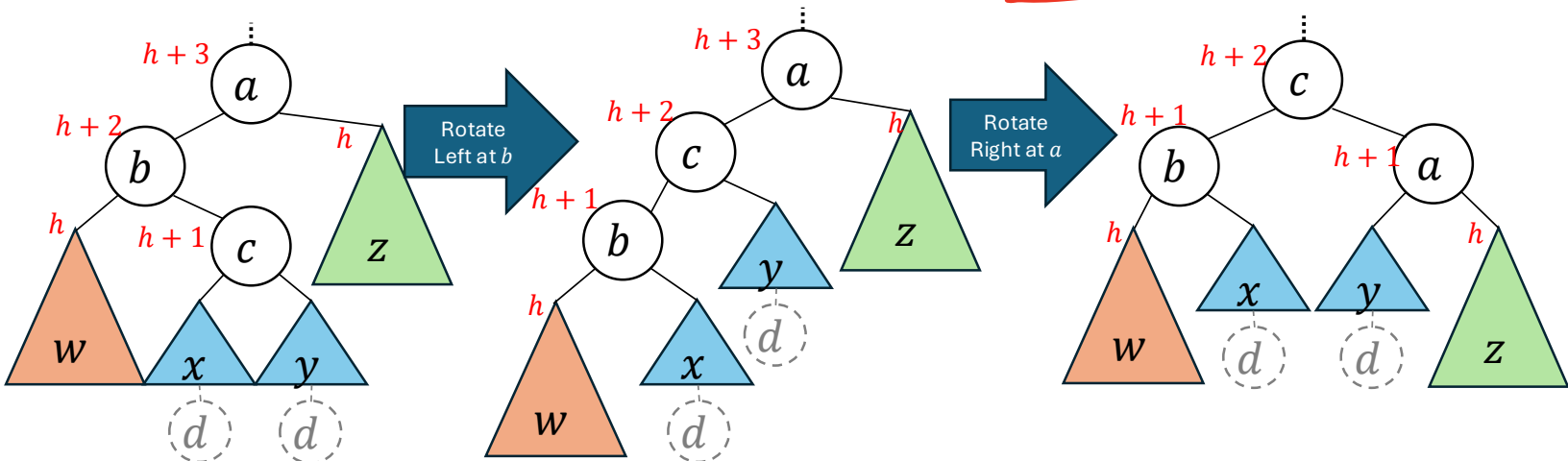
Case LR in General

- We just inserted d , node a is the deepest “problem” node
- Imbalance caused by inserting in the left child’s right subtree
- Rotate left at the left child
- Rotate right at the unbalanced node



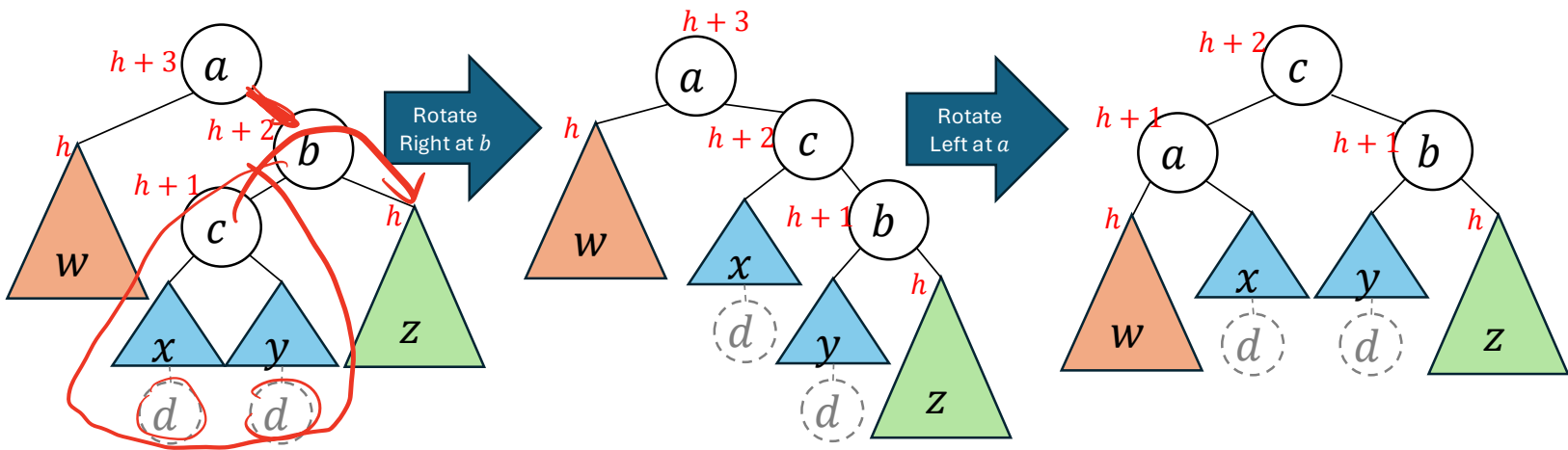
Case LR Implementation

```
b=a.left  
c=b.right  
b.right=c.left  
a.left=c.right  
c.right=a  
c.left=b  
return c
```



Case RL in General

- We just inserted d , node a is the deepest “problem” node
- Imbalance caused by inserting in the right child’s left subtree
- Rotate right at the right child
- Rotate left at the unbalanced node



Insert Summary

- After a BST insertion, update the heights of the node's ancestors
- From leaf to root, check if each node is balanced
- If a node is unbalanced then at the deepest unbalanced node:
 - Case LL: If we inserted in the **left** subtree of the **left** child then: rotate right
 - Case RR: If we inserted in the **right** subtree of the **right** child then: rotate left
 - Case LR: If we inserted into the **right** subtree of the **left** child then: rotate left at the left child and then rotate right at the root
 - Case RL: If we inserted into the **left** subtree of the **right** child then: rotate right at the right child and then rotate left at the root
- Done after either reaching the root or applying **one** of the above cases

Delete Summary

- TLDR: same cases, reverse direction of rotation, may need to repeat with ancestors
- After a BST deletion, update the heights of the node's ancestors
- From leaf to root, check if each node is unbalanced
- If a node is unbalanced then at the deepest unbalanced node:
 - Case LL: If we deleted in the **left** subtree of the **left** child then: rotate left
 - Case RR: If we deleted in the **right** subtree of the **right** child then: rotate right
 - Case LR: If we deleted into the **right** subtree of the **left** child then: rotate right at the left child and then rotate left at the root
 - Case RL: If we deleted into the **left** subtree of the **right** child then: rotate left at the right child and then rotate right at the root
- Continue checking until reach the root

Why is this $\Theta(\log n)$ time?

- We get poor running times when $height \approx n$
- Let $M(h)$ be the minimum count of nodes in an AVL tree of height h
- An AVL tree of height h must have **one subtree of height $h - 1$**
- This means the **other subtree has height at least $h - 2$**
- $M(h) = M(h - 1) + M(h - 2) + 1$

Comparing to Fibonacci Sequence

- $M(h) = M(h-1) + M(h-2) + 1$
- $F(n) = F(n-1) + F(n-2)$
 - Fibonacci Sequence
- So $M(h) > F(h)$
- For large values of h , $F(h) \approx \phi^h$
 - ϕ being the golden ratio. $\phi > 1.6$
- This means that an AVL tree of height h has at least ϕ^h nodes
 - And not more than 2^{h+1} nodes
- So a tree of n nodes has height at most $\log_\phi(n)$
 - We need $n \geq \phi^h$, so $\log_\phi n \geq h$
- All operations run in time $O(\log n)$
 - The maximum number of nodes for height h is $2^{h+1} - 1$, so they are also $\Omega(\log n)$