CSE 373: Data Structures and Algorithms

Lecture 10: AVL Trees

Instructor: Lilian de Greef
Quarter: Summer 2017

Today

- Announcements
- BSTs continued (th~~is time, bringing~~
  - buildTree
  - Balance Conditions
  - AVL Trees
  - Tree rotations

Announcements

- Reminder: homework 3 due Friday
- Homework 2 grades should come out today or tomorrow!
- Section
  - Will especially go over material from today (it's especially tricky)
  - TAs can go over some of the tougher hw2 questions in section if you want/ask

Back to Binary Search Trees

`buildTree` for BST

Let's consider `buildTree` (insert values starting from an empty tree)

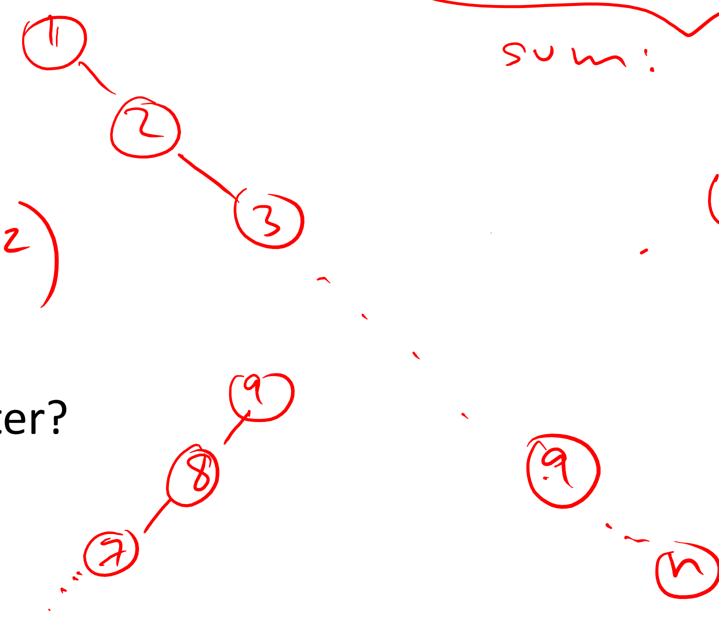
Insert values 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST

$\underbrace{1, 2, 3, \ldots, 9 \ldots n}$

sum: $\frac{n(n+1)}{2}$

$O(n^2)$

- If inserted in given order, what is the tree?
- What big-O runtime for buildTree on this sorted input? $O(n^2)$
- Is inserting in the reverse order any better?

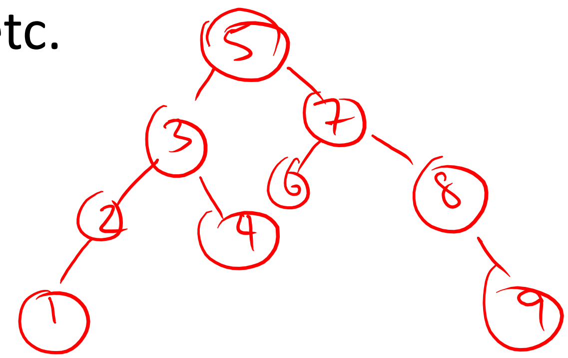


`buildTree` for BST

Insert values 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST

What we if could somehow re-arrange them

- median first, then left median, right median, etc.
  5, 3, 7, 2, 1, 4, 8, 6, 9



- What tree does that give us?

- What big-O runtime? $O(n \log n)$

insert $O(\log n)$

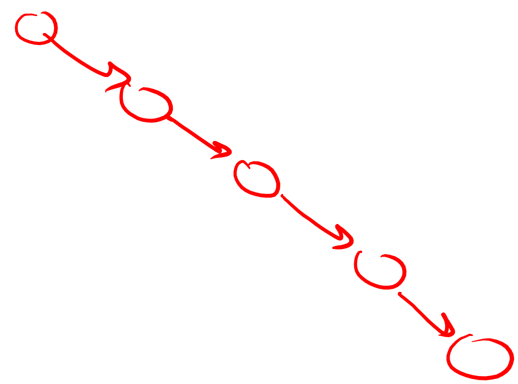
Balancing Binary Search Trees

BST: Efficiency of Operations?

Problem: operations may be inefficient if BST is unbalanced

Worst-case running time:

worst case:



- `find`, `insert`, `delete` $O(n)$
- `buildTree` $O(n^2)$

How can we make a BST efficient?

Observation

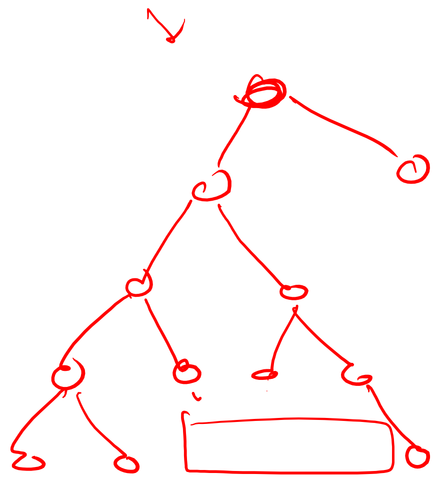
BST : the shallow, the better

Solution: Require a **Balance Condition** that

1. Ensures depth is always $O(\log n)$
2. Is efficient to maintain

- When we build the tree, make sure it's balanced.
- BUT…Balancing a tree only at build time is insufficient.
- We also need to also keep the tree balanced as we perform operations.

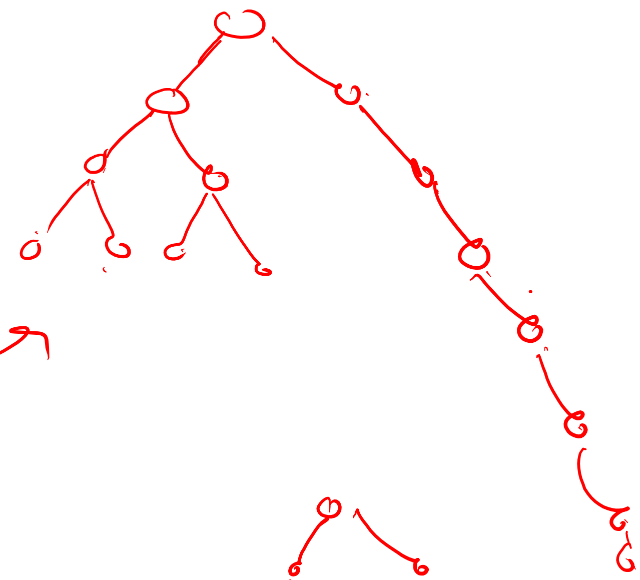
Ideas for Balance Conditions?



Potential Balance Conditions

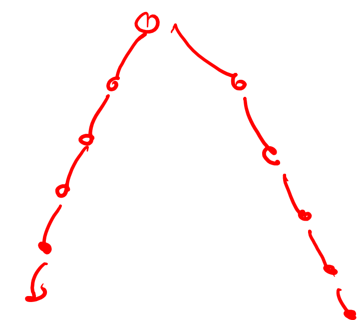
- Left and right subtrees of the root have equal # of nodes

  (Not strong enough! eg. height mismatch)

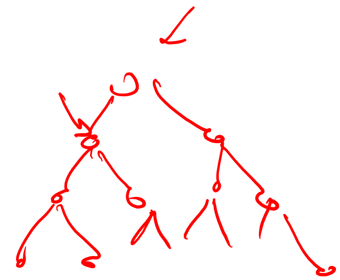


- Left and right subtrees of the root also have equal height

  (still not enough! eg. Double chain)



Potential Balance Conditions



- Left and right subtrees of every node have equal # nodes.

  (Too strong! Only perfect trees ($2^h - 1$ nodes))

- Left and right subtrees of every node have equal height

  (too strong! still only perfect trees)

Potential Balance Conditions

AVL Tree!

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

If we define

$$\text{balance}(\text{node}) = \text{height}(\text{node.left}) - \text{height}(\text{node.right})$$

Maintain property for every node

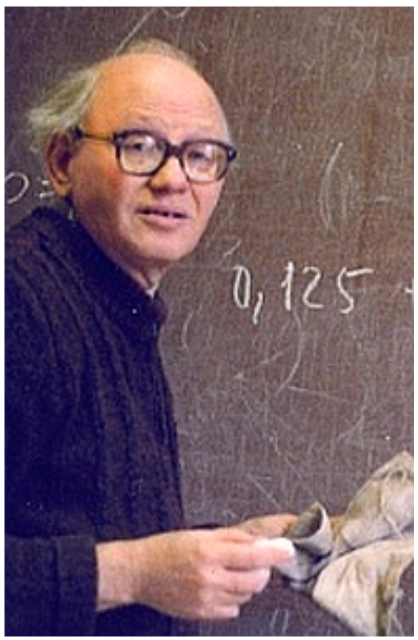
$$-1 \le \text{balance}(\text{node}) \le 1$$

AVL Tree

A kind of self-balancing binary search tree!

AVL Tree (Bonus material: etymology)

Invented by Georgy **A**delson-**V**elsky and Evgenii **L**andis in 1962



The AVL Tree Data Structure

An **AVL tree** is a *self-balancing* binary search tree.

Structural properties

1. Binary tree property (same as BST)
2. Order property (same as for BST)

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

   where **balance**(*node*) = height(*node*.left) – height(*node*.right)

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

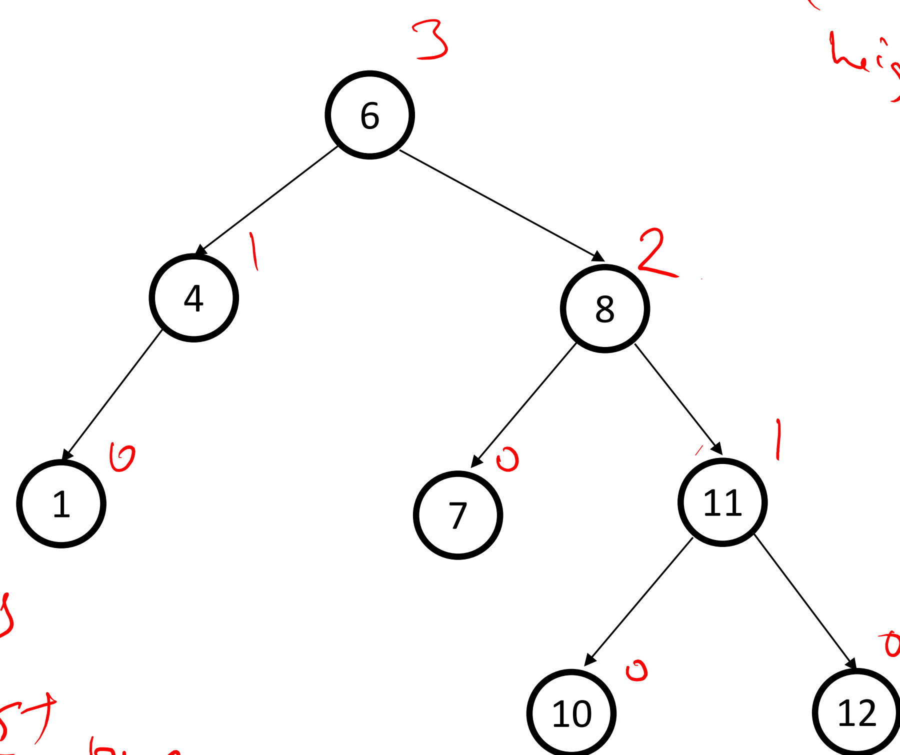
$\log n + 1$

Example #1: Is this an AVL Tree?

height(node) = 1+ max(height left, height right)

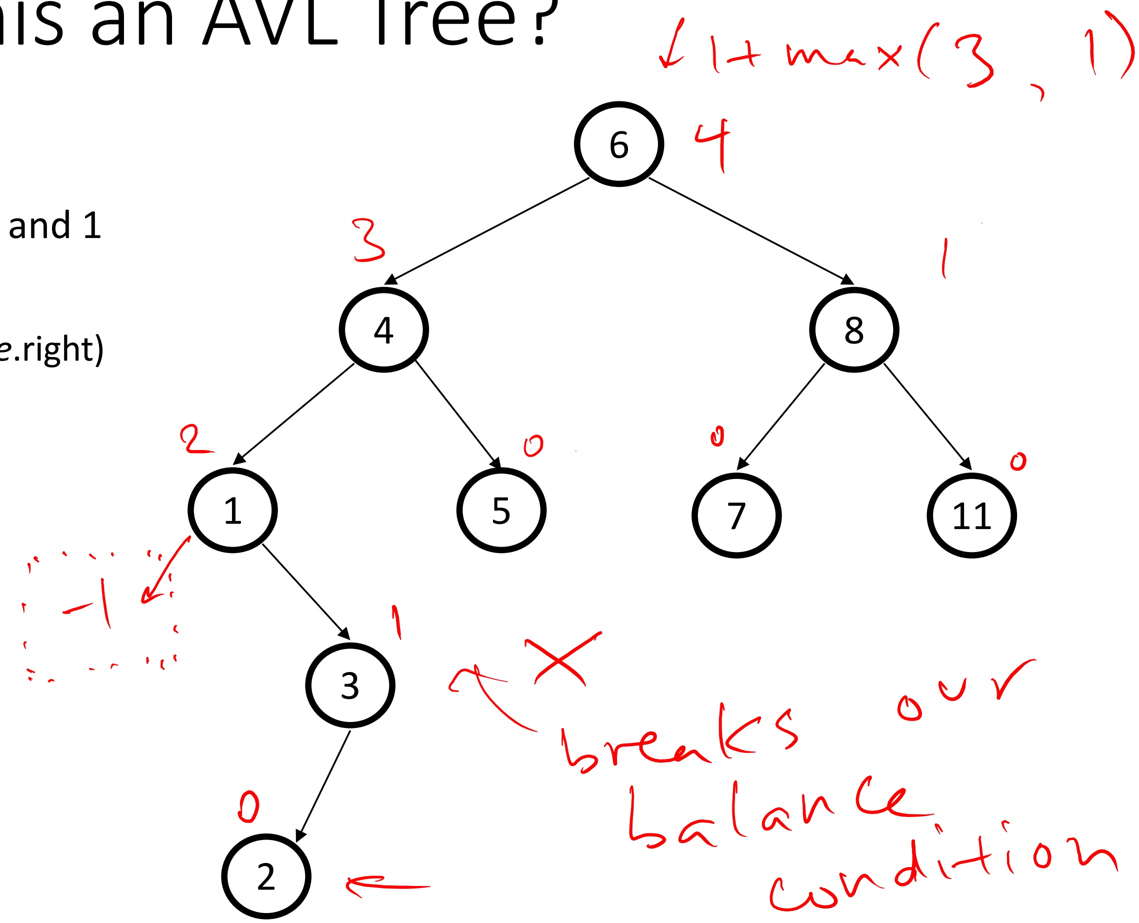
Balance Condition:
balance of every node is between -1 and 1

where balance(*node*) =
height(*node*.left) – height(*node*.right)



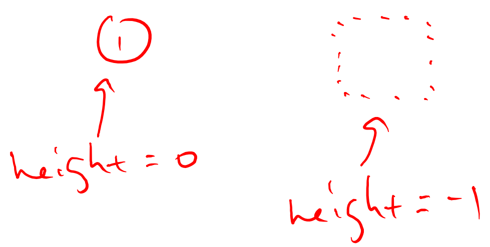
Yes! Because the left and right subtree of every node have heights differing by at most one

Example #2: Is this an AVL Tree?



Balance Condition:
balance of every node is between -1 and 1

where balance(*node*) =
height(*node*.left) – height(*node*.right)



AVL Trees

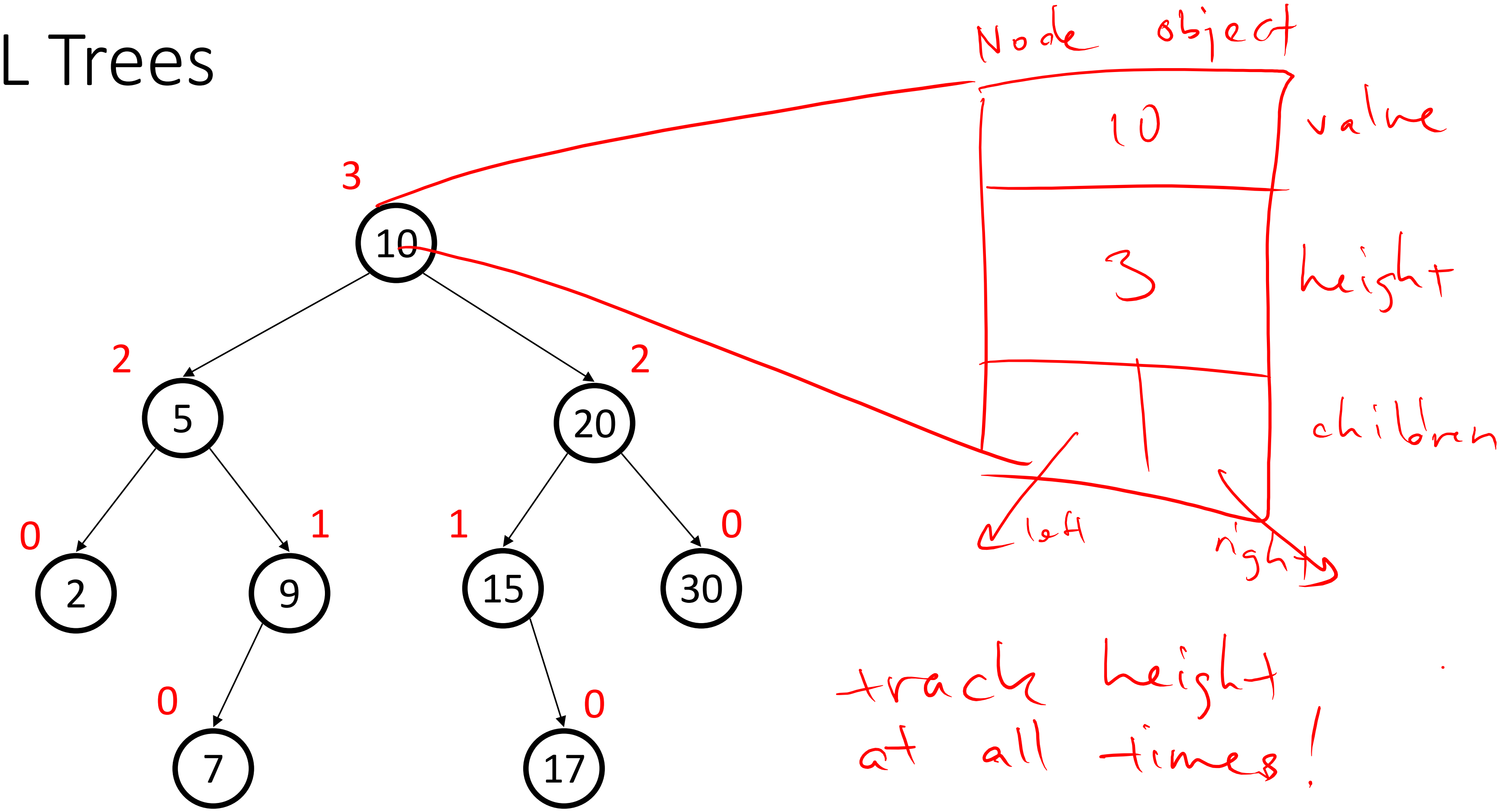
Good News:

Because height of AVL tree is O(log(n)), then `find` $O(\log n)$

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

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

AVL Trees



track height
at all times!

AVL tree operations

- AVL `find`:
  - Same as usual BST `find`
- AVL `insert`:

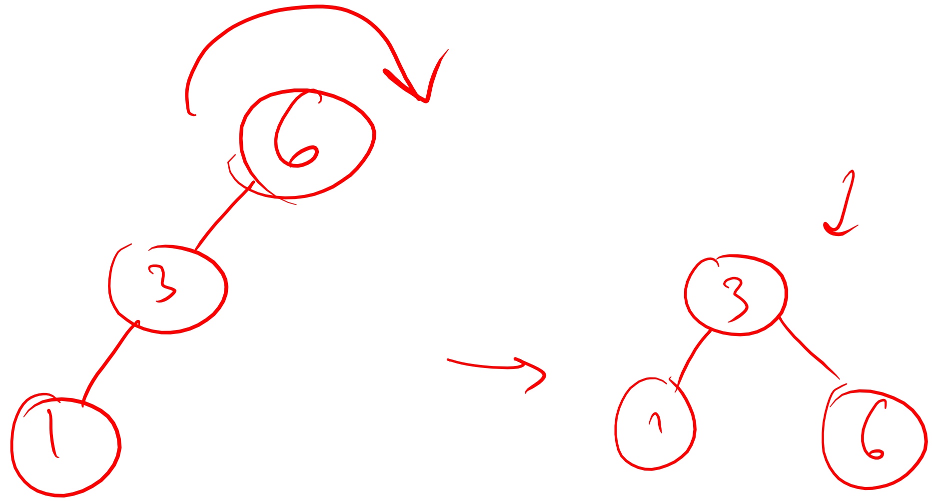
  - First, usual generic BST insert ← ①
    then check balance condition ← ②
    and potentially "fix" the tree ← ③
  - Four different imbalance case
- AVL `delete`:
  - The "easy way" is lazy deletion
  - Otherwise, do the deletion and then check for several imbalance cases (we will skip this)

First `insert` example

Insert(6)

Insert(3)

Insert(1)



Third insertion

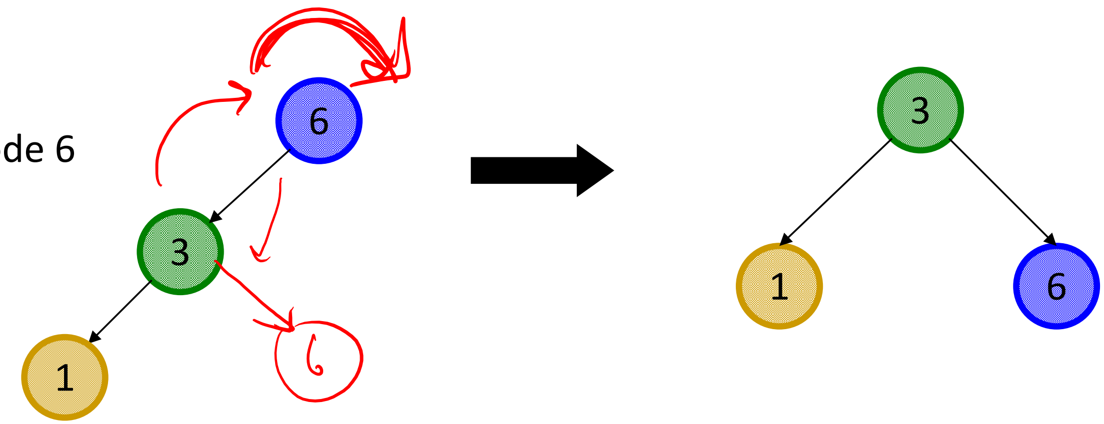
violates balance condition
(happens to be at the root in this example)

What's the only way to fix it?

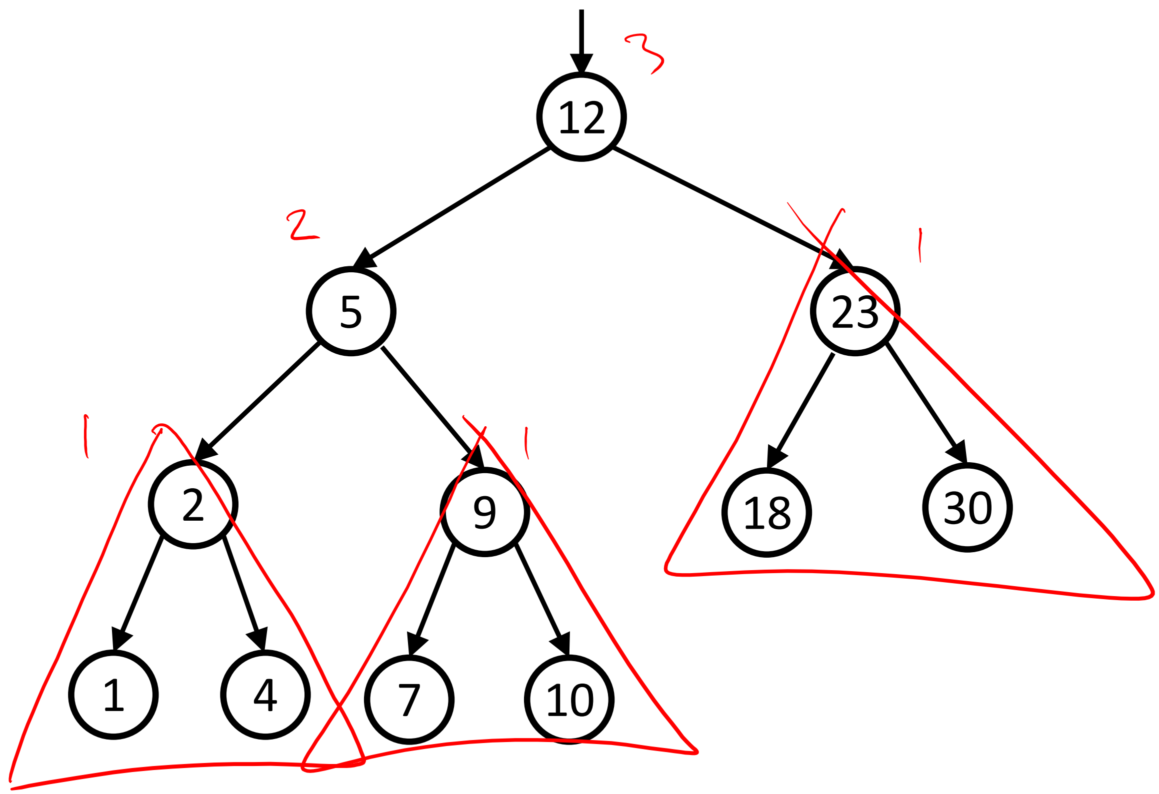
Fix: Apply “Single Rotation”

- ***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 (we’ll see in generalized example)

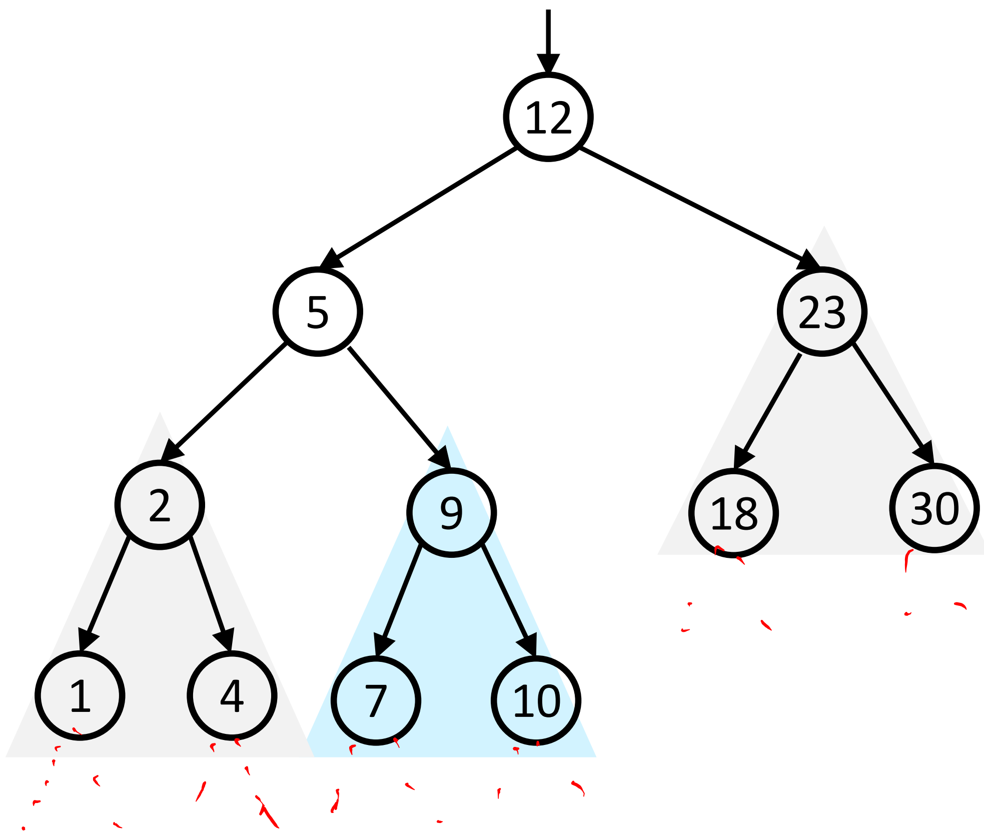
AVL Property
violated at node 6



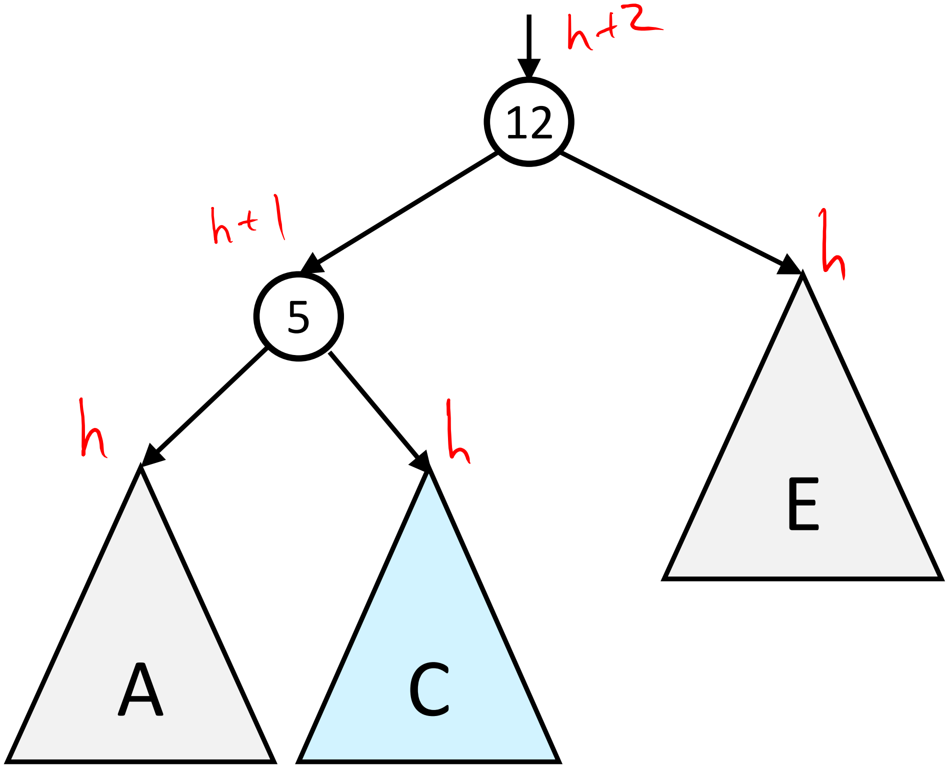
Generalizing our examples...



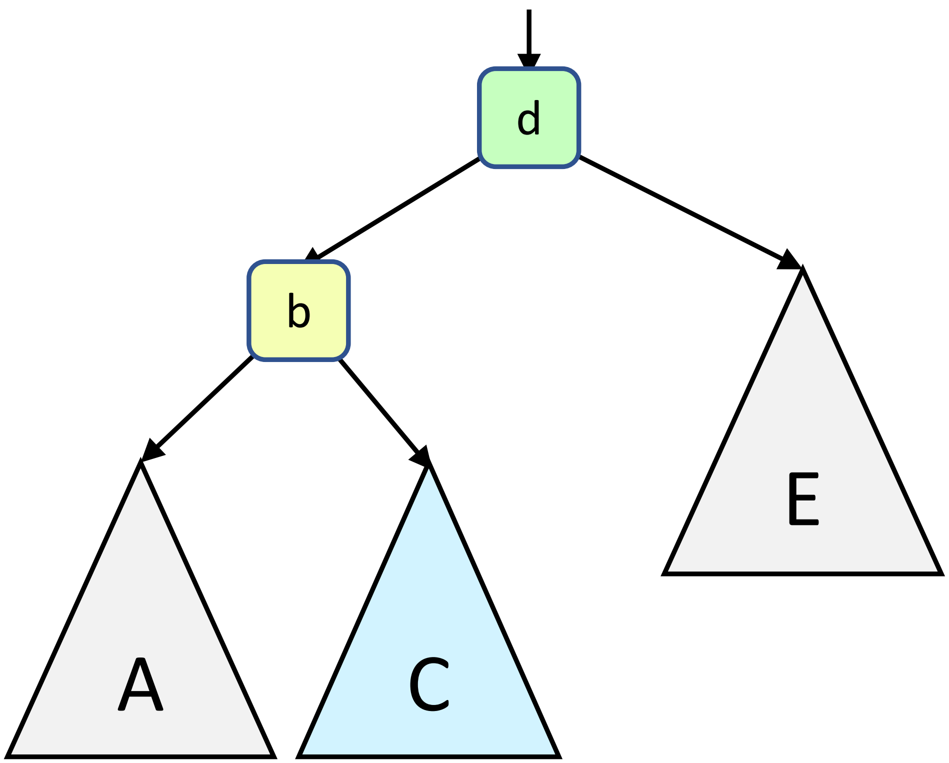
Generalizing our examples...



Generalizing our examples...

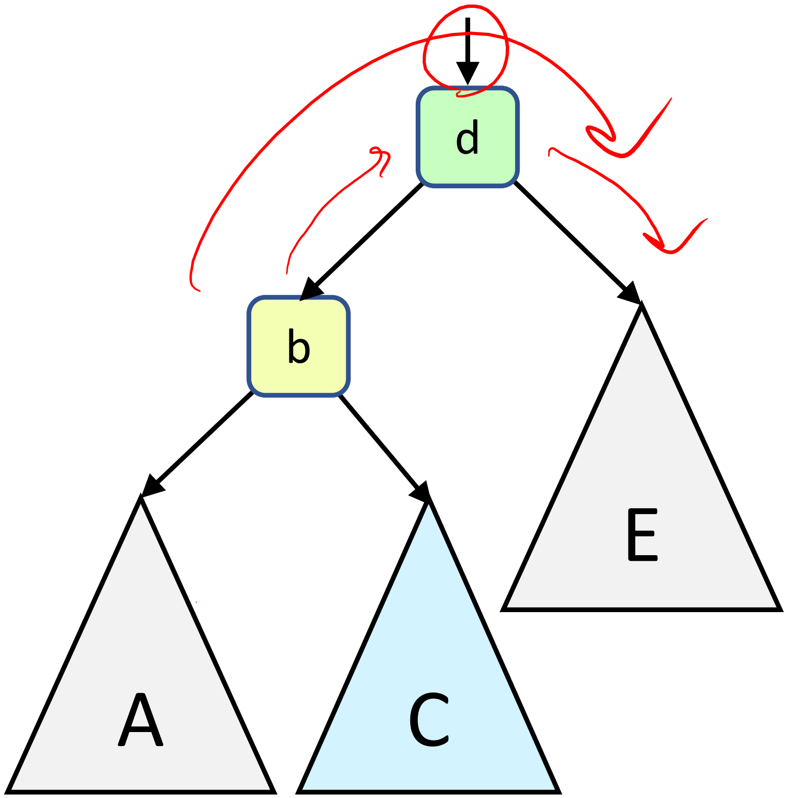


Generalizing our examples...

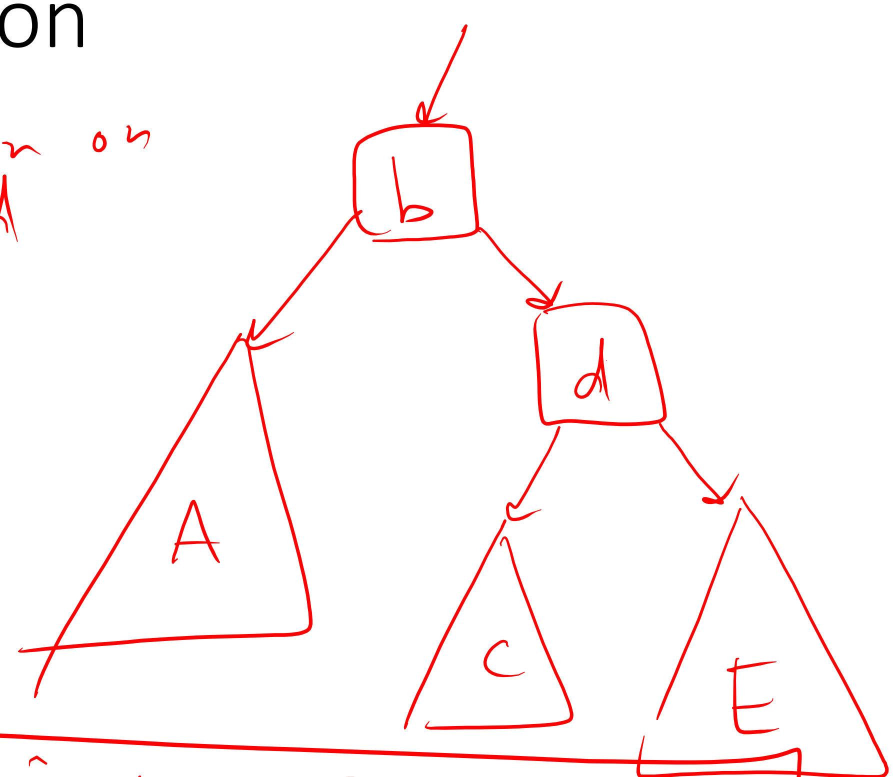


AVL tree

Generalized Single Rotation



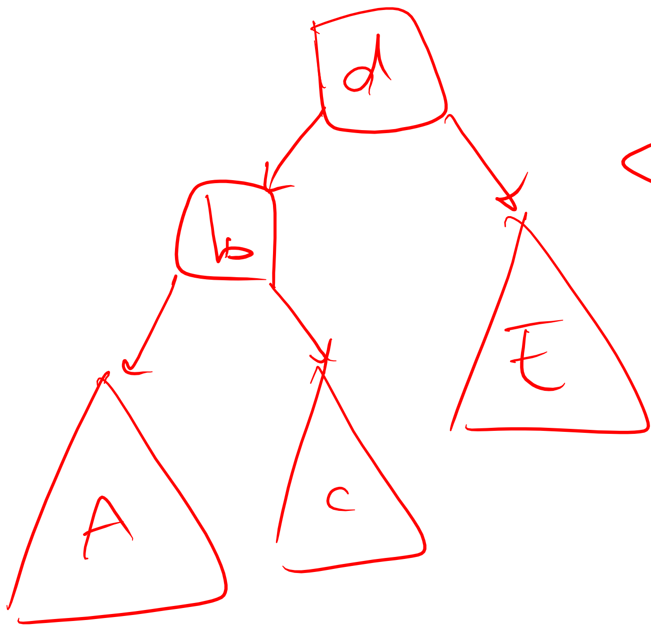
rotation on d

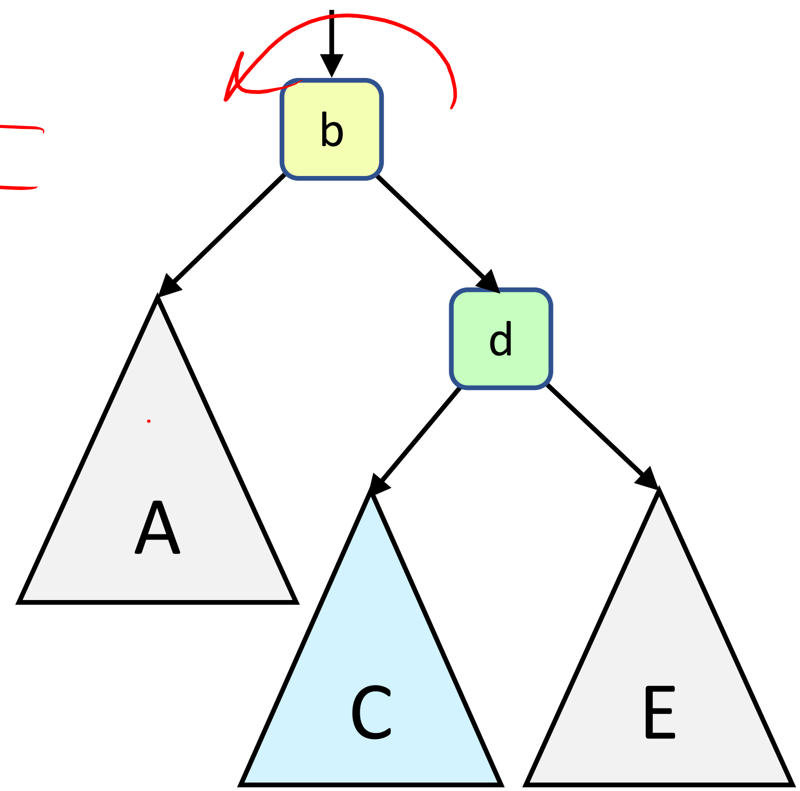


Right Rotation

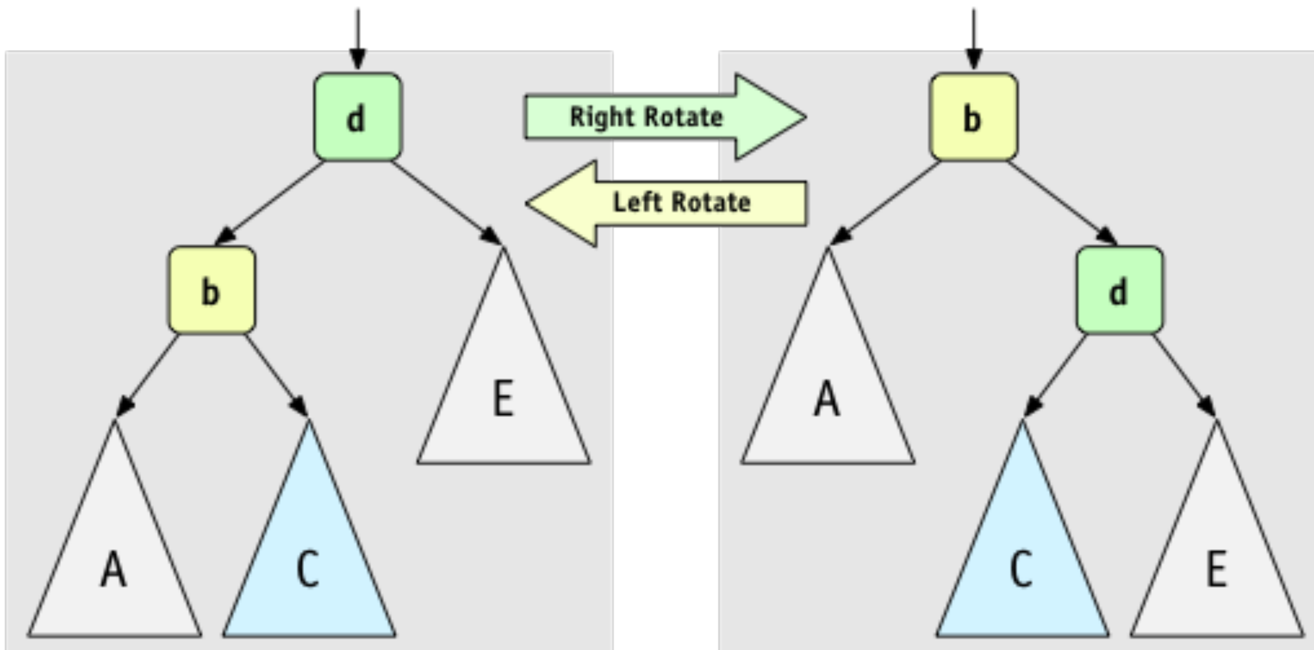
Generalized Single Rotation

Left Rotation





Single Rotations



(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

AVL Tree `insert` (more specific):

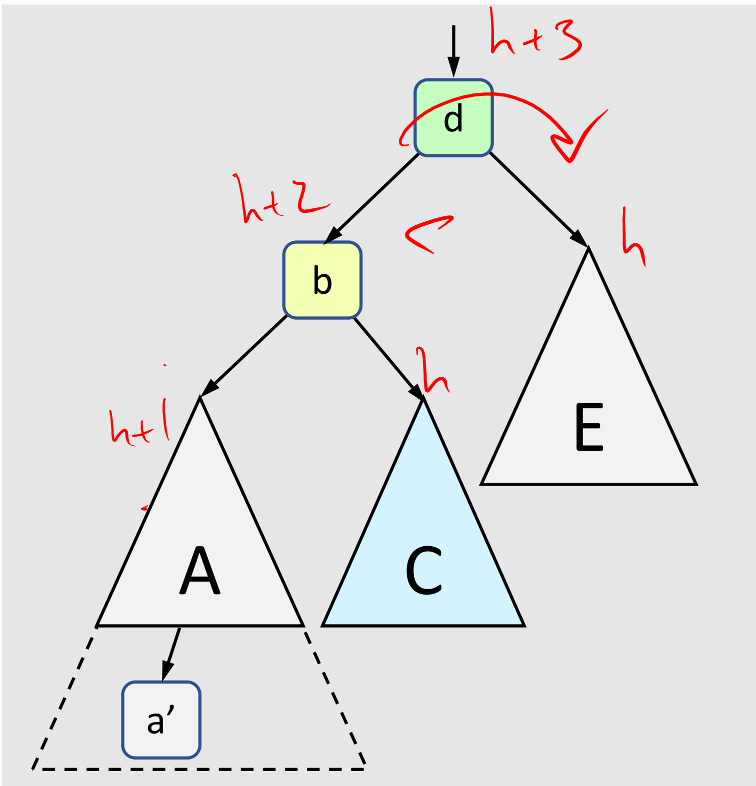
1. Insert the new node as in our generic BST (a new leaf)

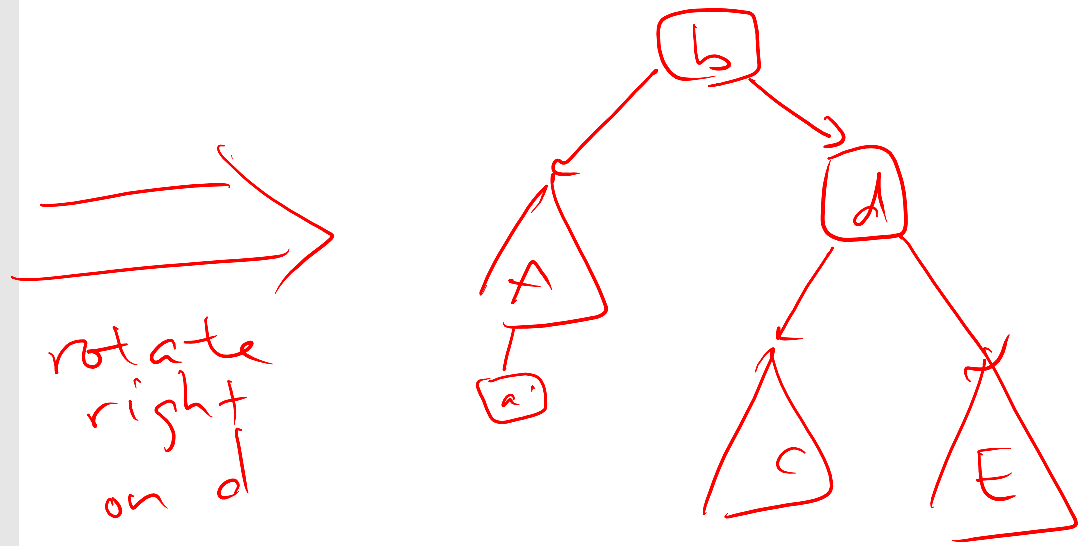
2. For each node on the path from the root to the new leaf, the insertion may (or may not) have changed the node's height

3. So after insertion in a subtree, detect height imbalance and perform rotation to restore balance at that node

4. Always look for the deepest node that is unbalanced

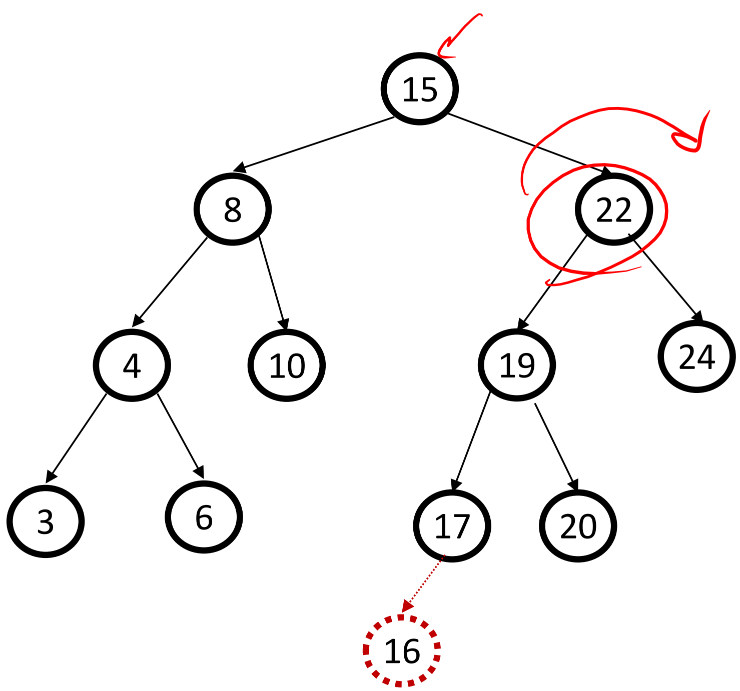
Case #1: Left-Left Case

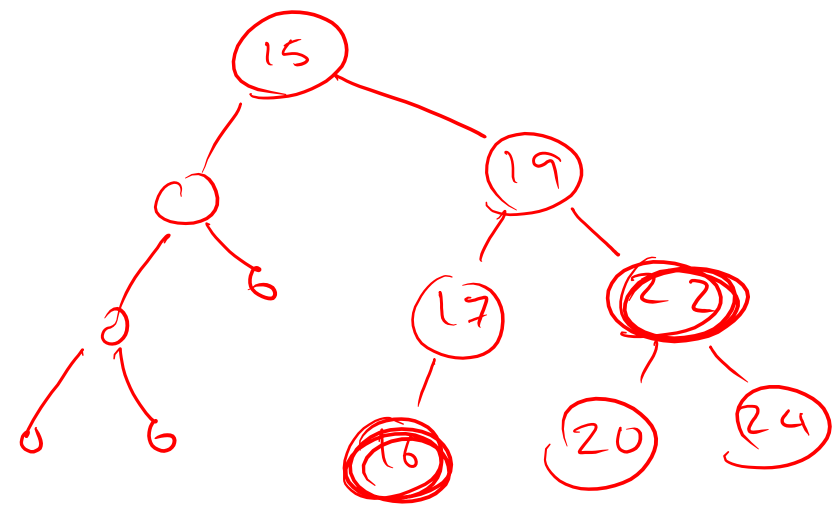




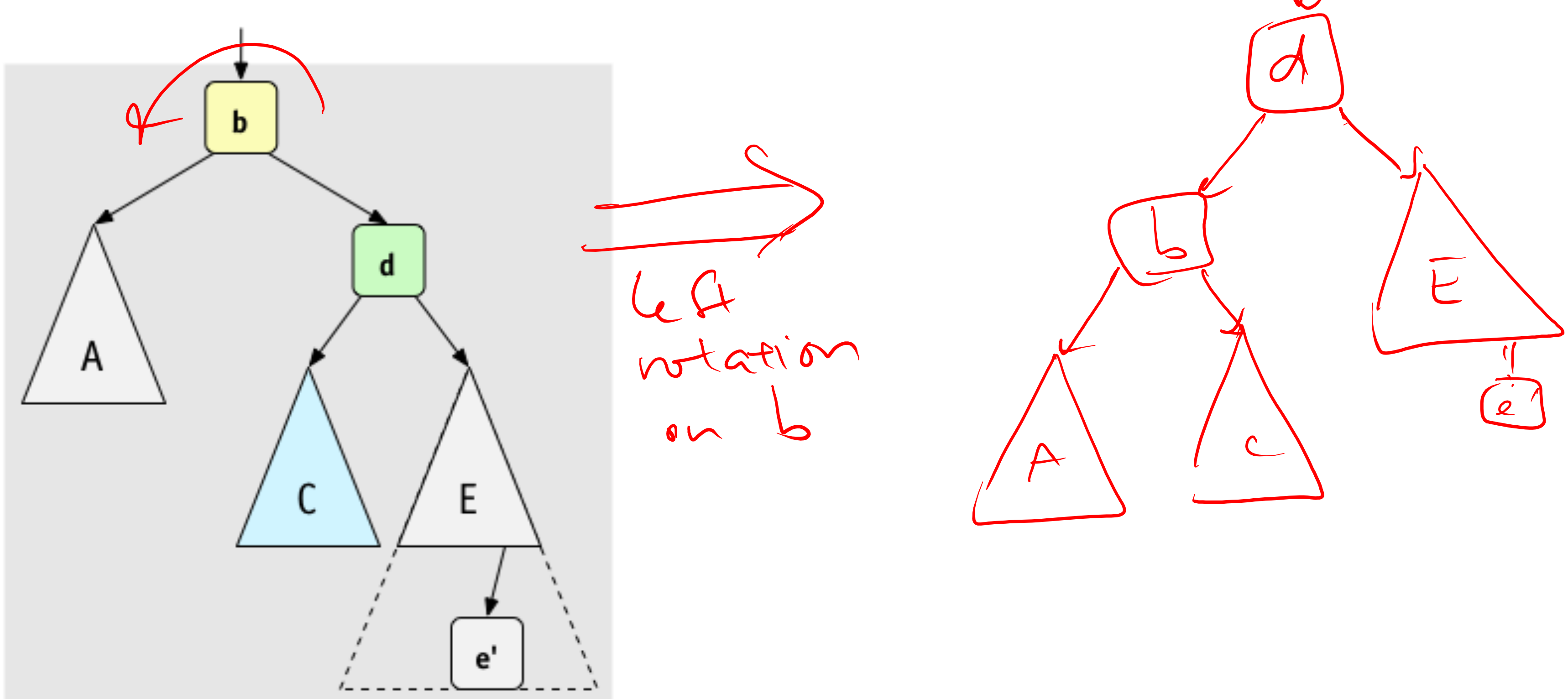
(Figures by Melissa O’Neill, reprinted with her permission to Lilian)

Example #2 for left-left case: `insert(16)`



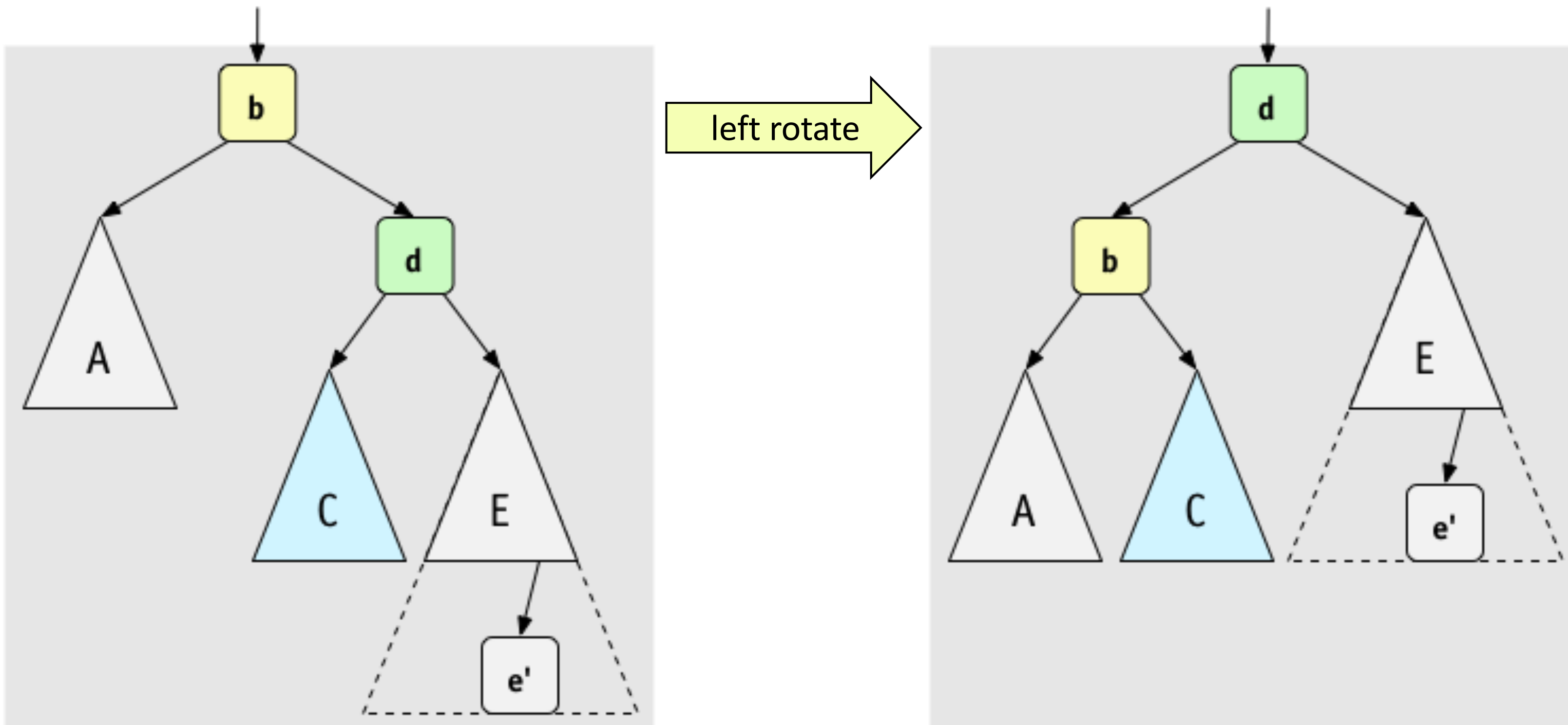


Case #2: Right-Right Case



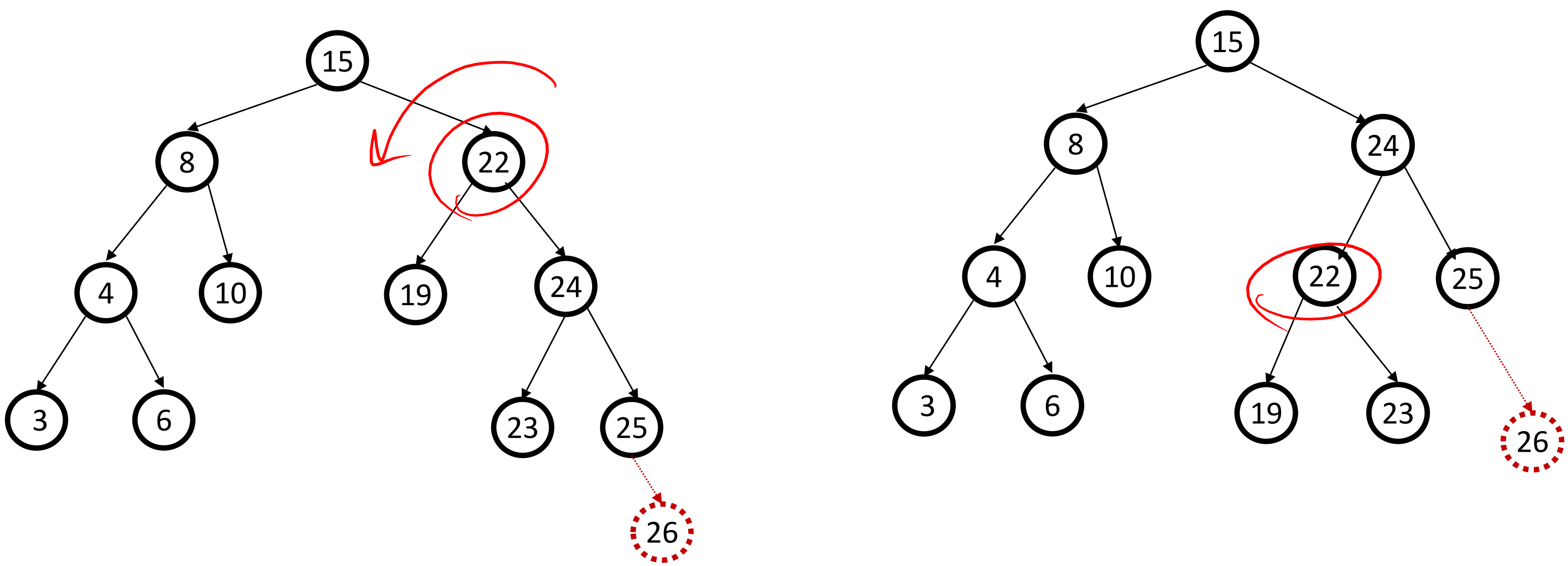
(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Case #2: Left-Left Case

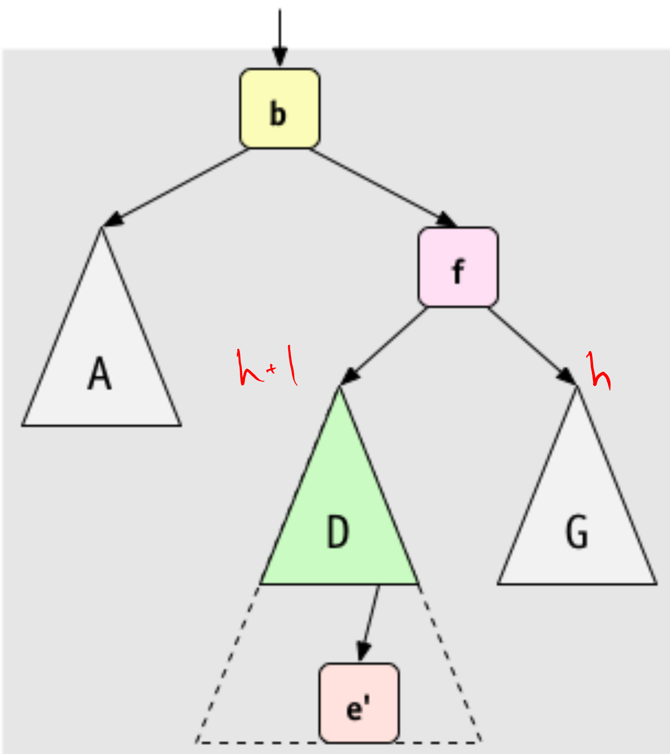


(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Example for right-right case: `insert(26)`

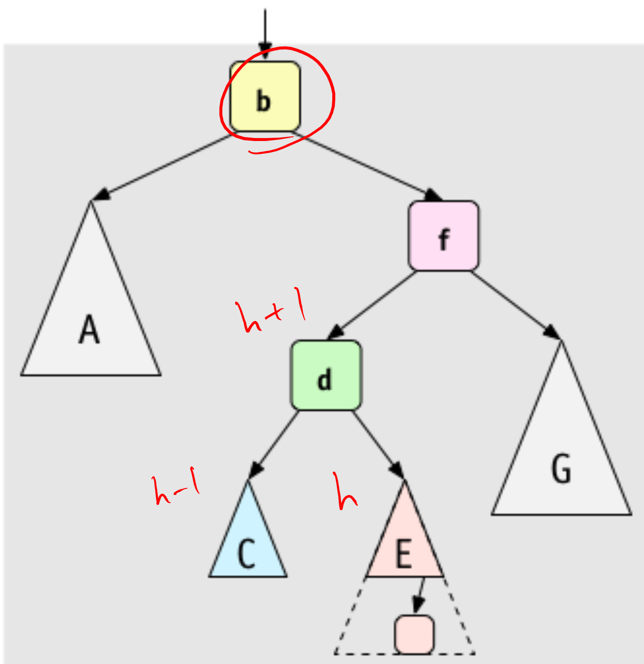


Case #3: Right - Left Case

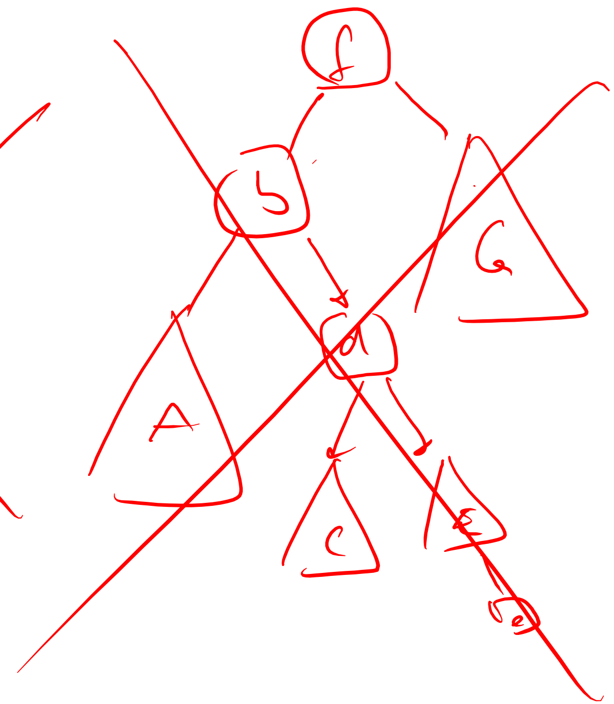


(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

A Better Look at Case #3:

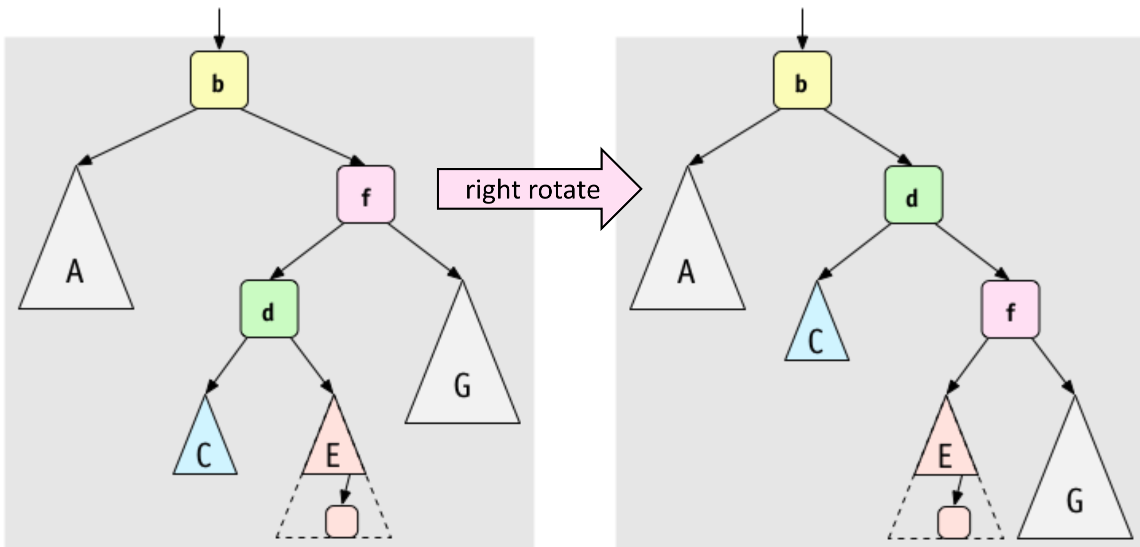


~~try left rotation~~



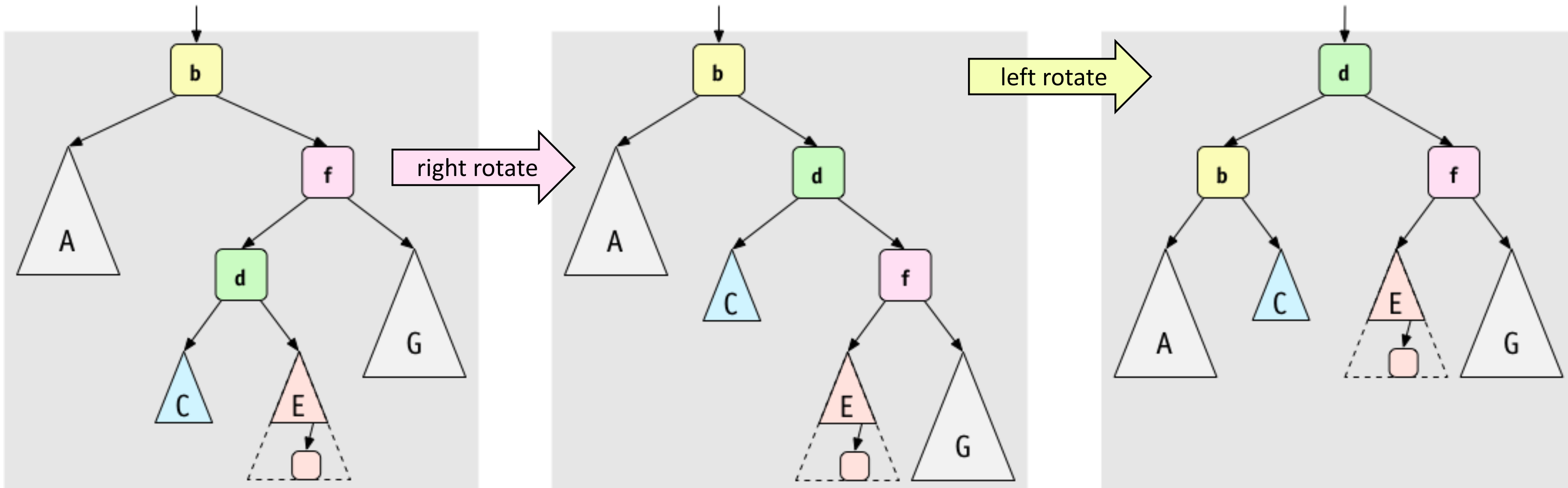
(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Case #3: Right-Left Case (after one rotation)



(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Case #3: Right-Left Case (after two rotations)



A way to remember it:
Move d to grandparent's position. Put everything else in their only legal positions for a BST.

(Figures by Melissa O'Neill, reprinted with her permission to Lilian)