CSE 373: Data Structures and Algorithms

Lecture 10: AVL Trees

Instructor: Lilian de Greef
Quarter: Summer 2017

Today

- Announcements
- BSTs continued (this time, bringing
  - buildTree
  - Balance Conditions
  - AVL Trees
  - Tree rotations

Announcements

- Reminder: homework 3 due Friday
- Homework 2 grades should come out today
- 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

- If inserted in given order,
  what is the tree?
- What big-O runtime for
  buildTree on this sorted input?
- 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?

Balancing Binary Search Trees

BST: Efficiency of Operations?

Problem:

Worst-case running time:

- `find`, `insert`, `delete`

- `buildTree`

How can we make a BST efficient?

Observation

Solution: Require a **Balance Condition** that

- 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.

Potential Balance Conditions

- Left and right subtrees

- Left and right subtrees

Potential Balance Conditions

- Left and right subtrees

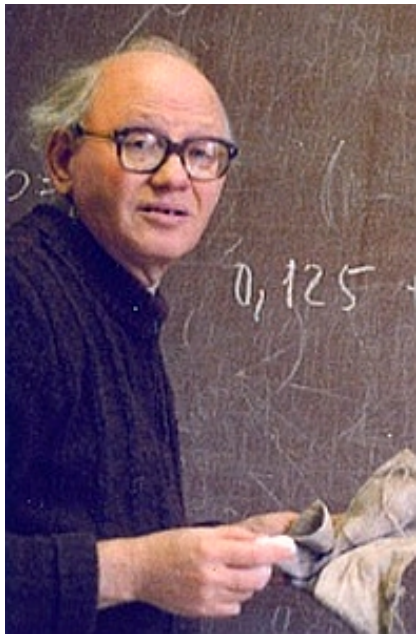
- Left and right subtrees

Potential Balance Conditions

Left and right subtrees

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

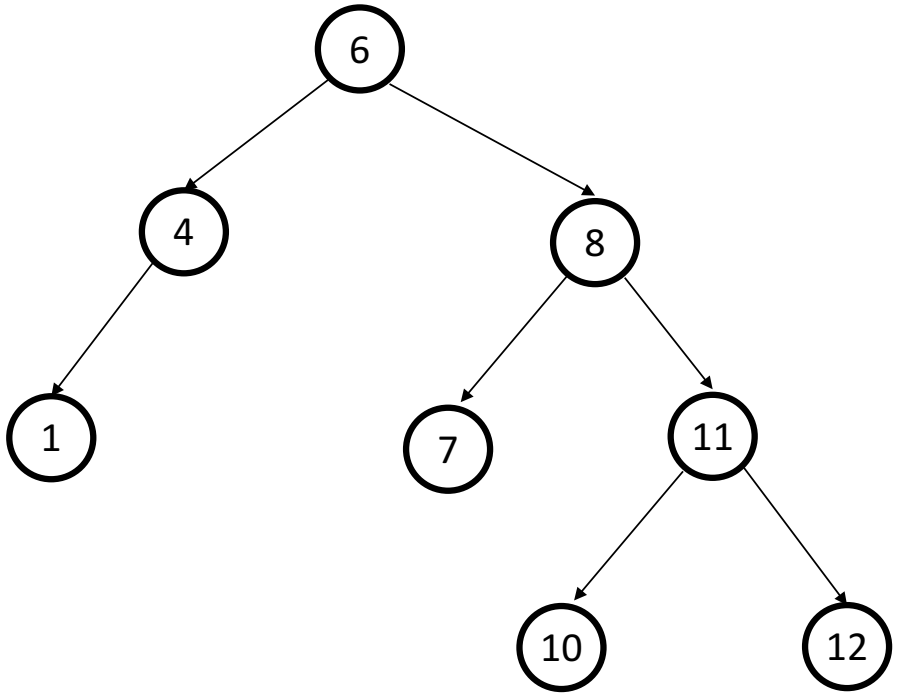
   where **balance**(*node*) = height(*node*.left) – height(*node*.right)

Result: **Worst-case** depth is

Example #1: Is this an AVL Tree?

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

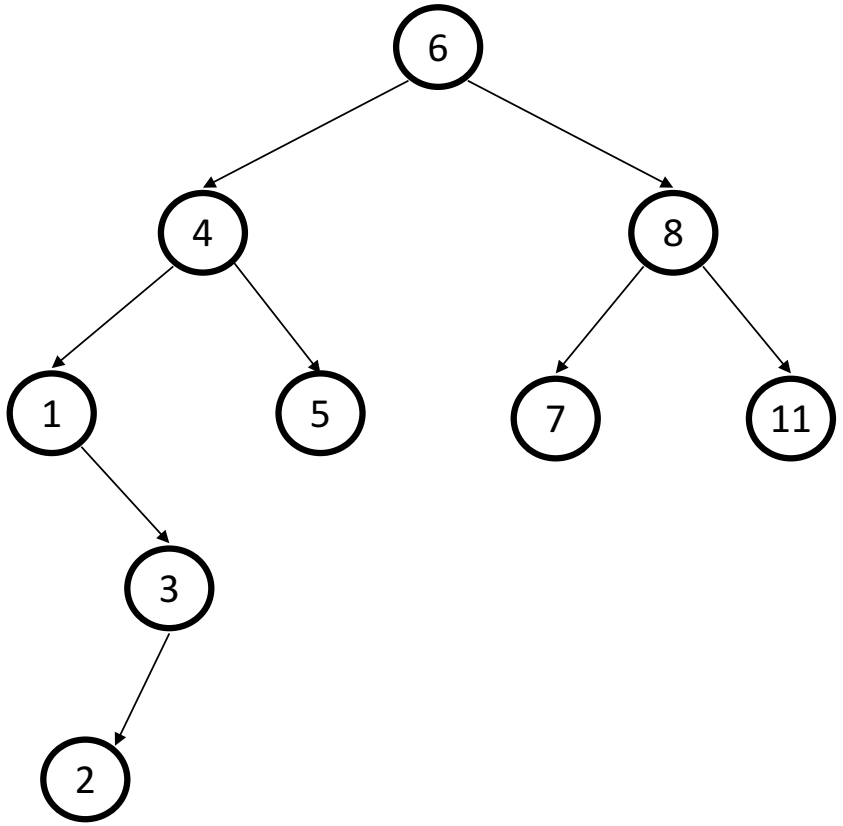
where balance(*node*) =
height(*node*.left) – height(*node*.right)



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)



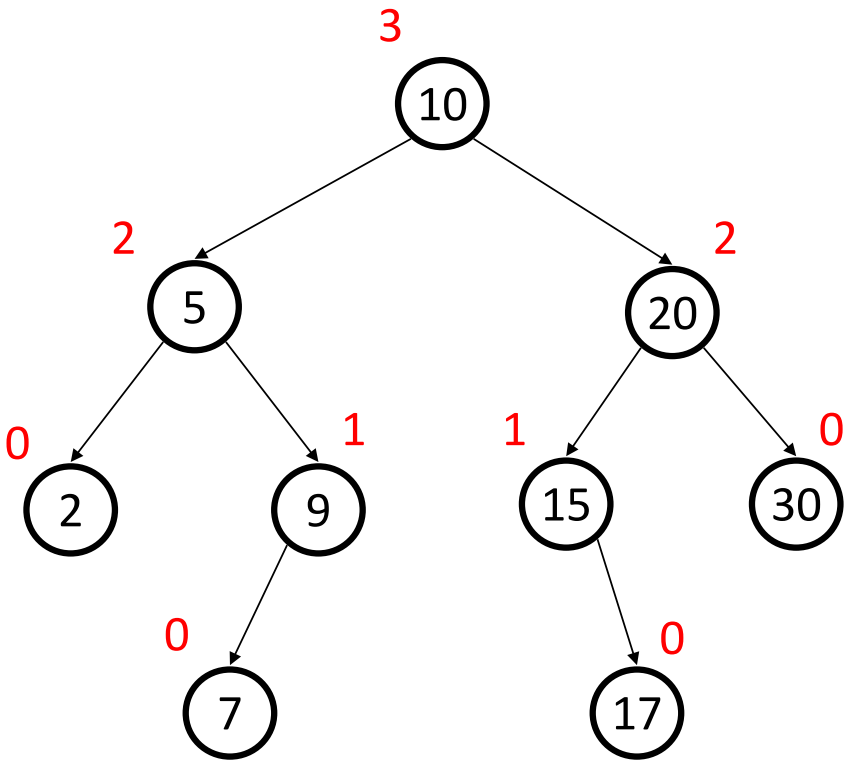
AVL Trees

Good News:

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

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

AVL Trees



AVL tree operations

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

- AVL `insert`:

- 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)

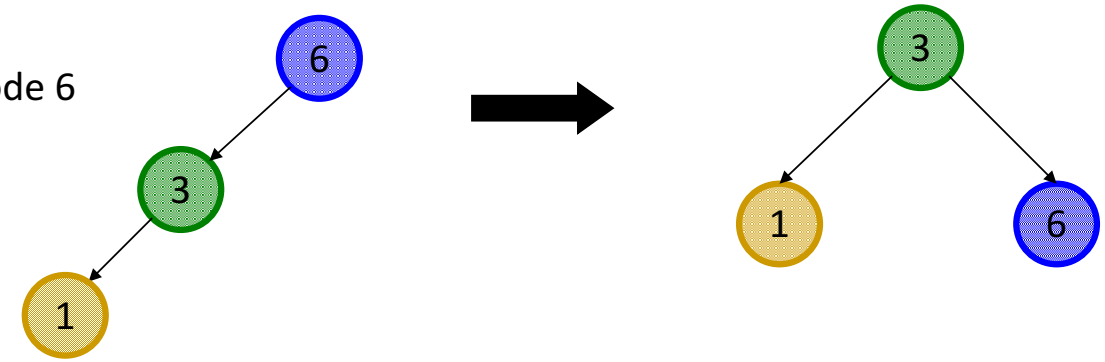
Third insertion

What's the only way to fix it?

Fix: Apply “Single Rotation”

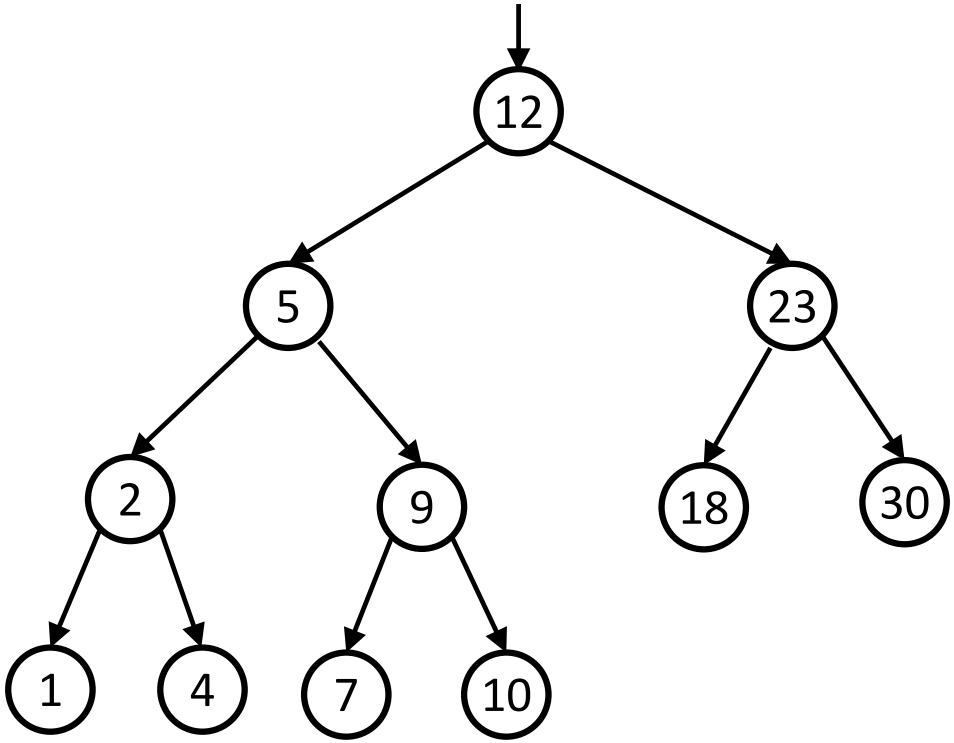
- ***Single rotation:*** The basic operation we’ll use to rebalance
  - Move child of unbalanced node into parent position
  - Parent becomes the “other” child (always okay in a BST!)
  - Other subtrees move in only way BST allows (we’ll see in generalized example)

AVL Property
violated at node 6

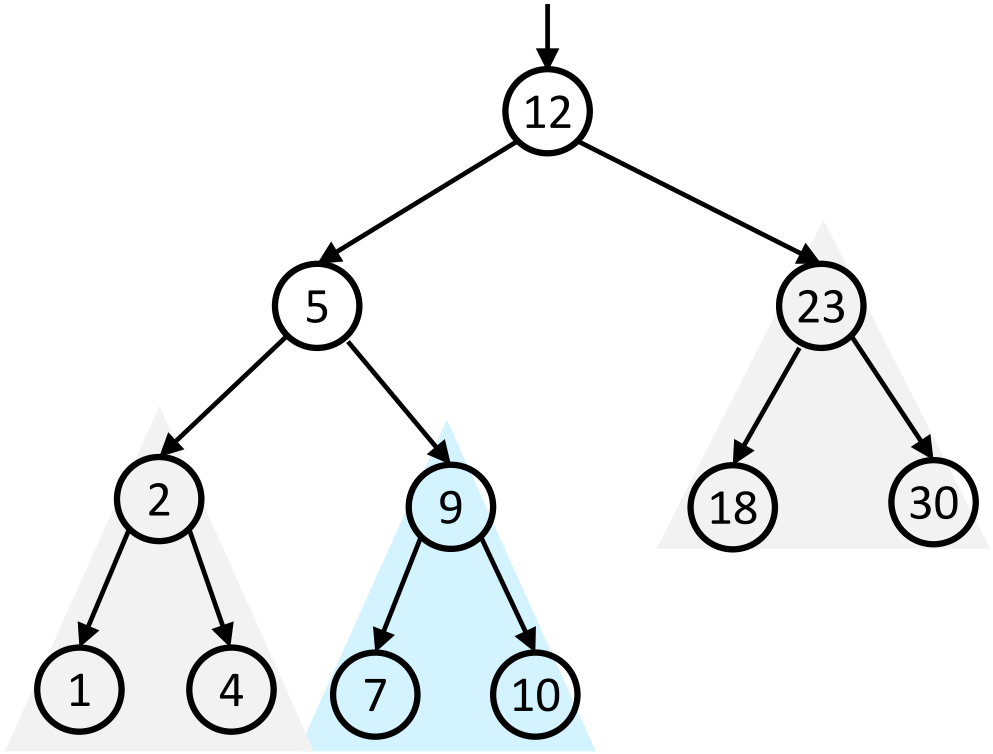


Tree Rotations: Generalized

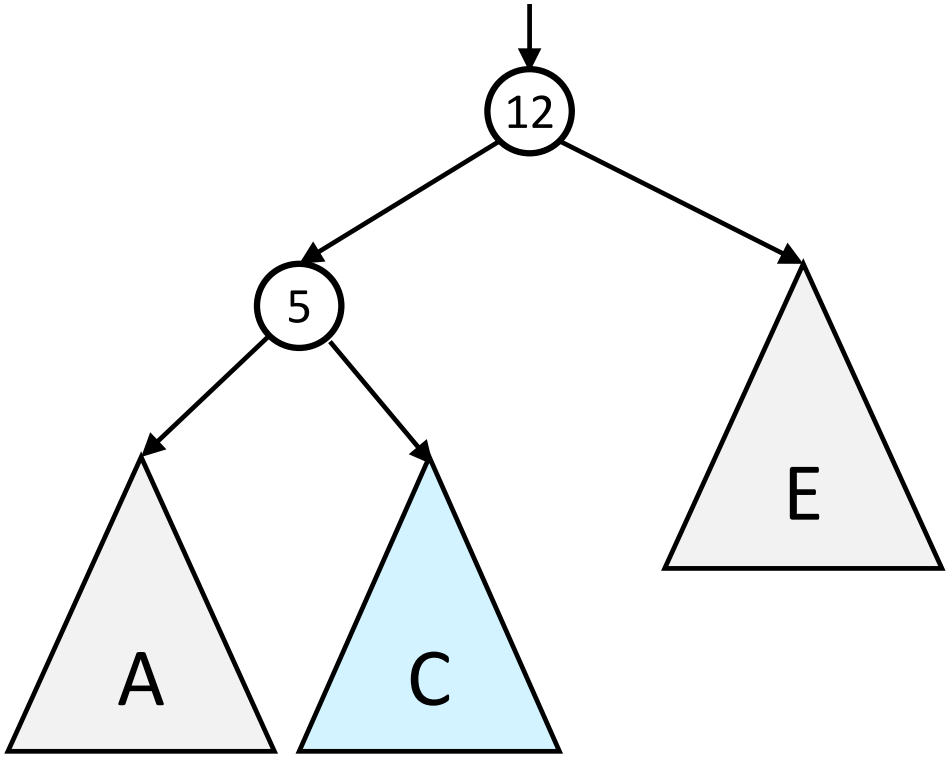
Generalizing our examples…



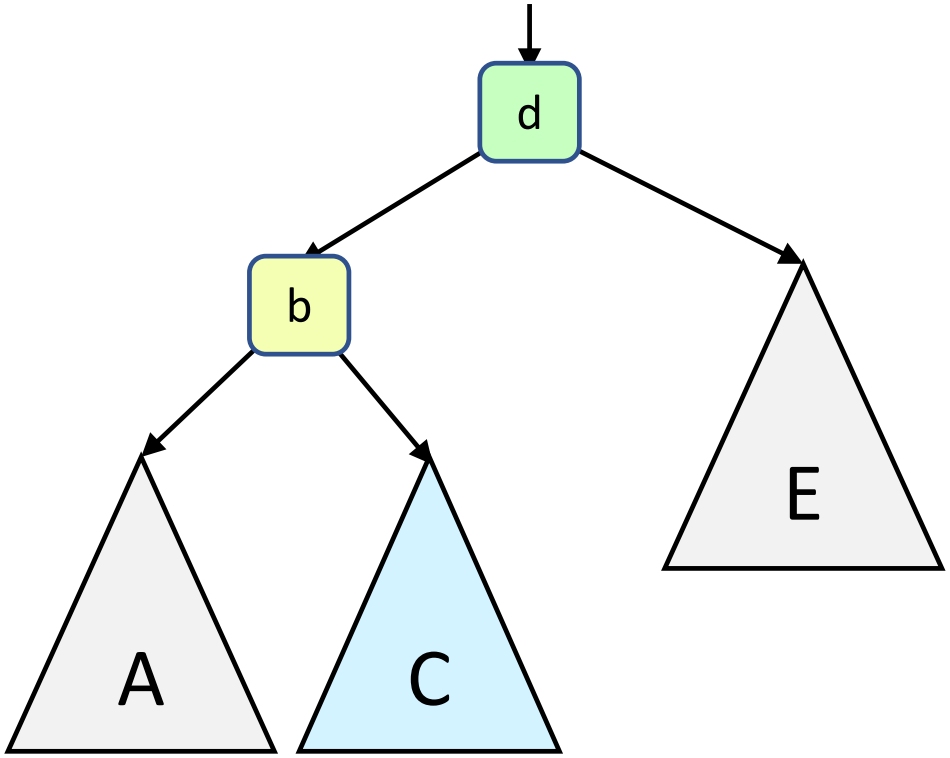
Generalizing our examples…



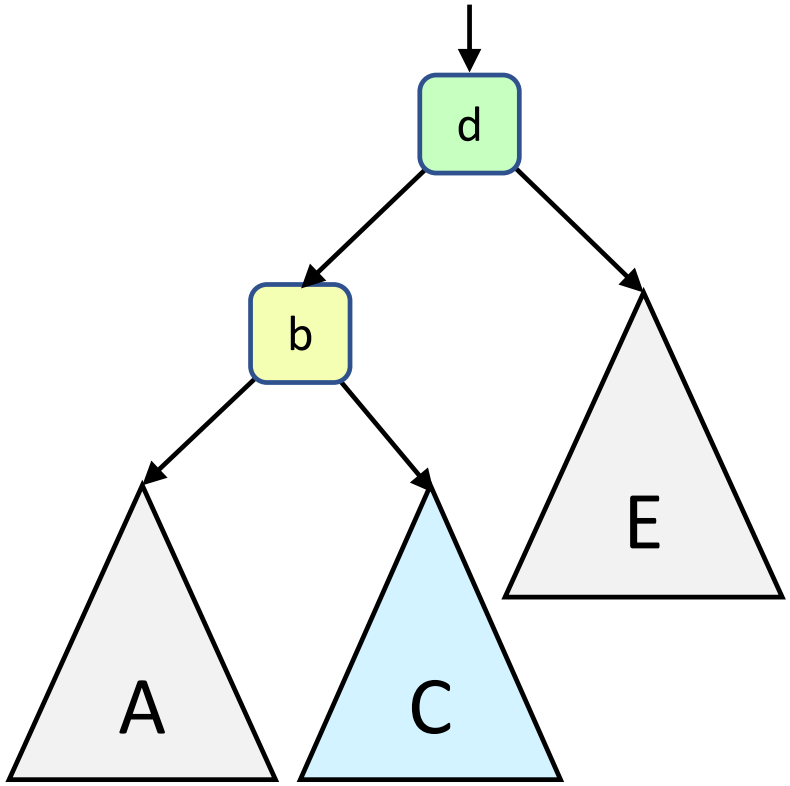
Generalizing our examples…



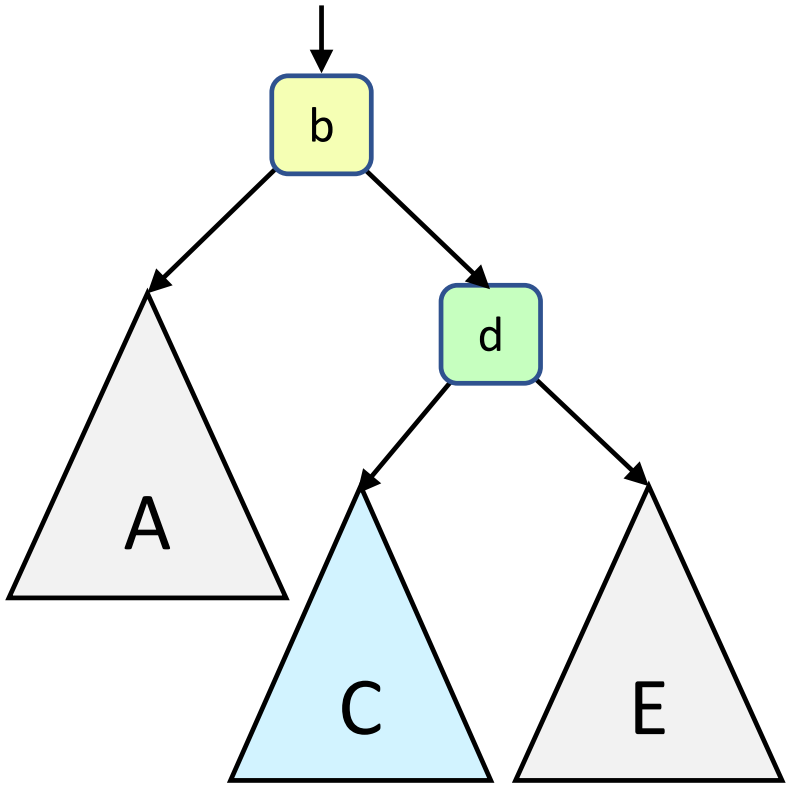
Generalizing our examples…



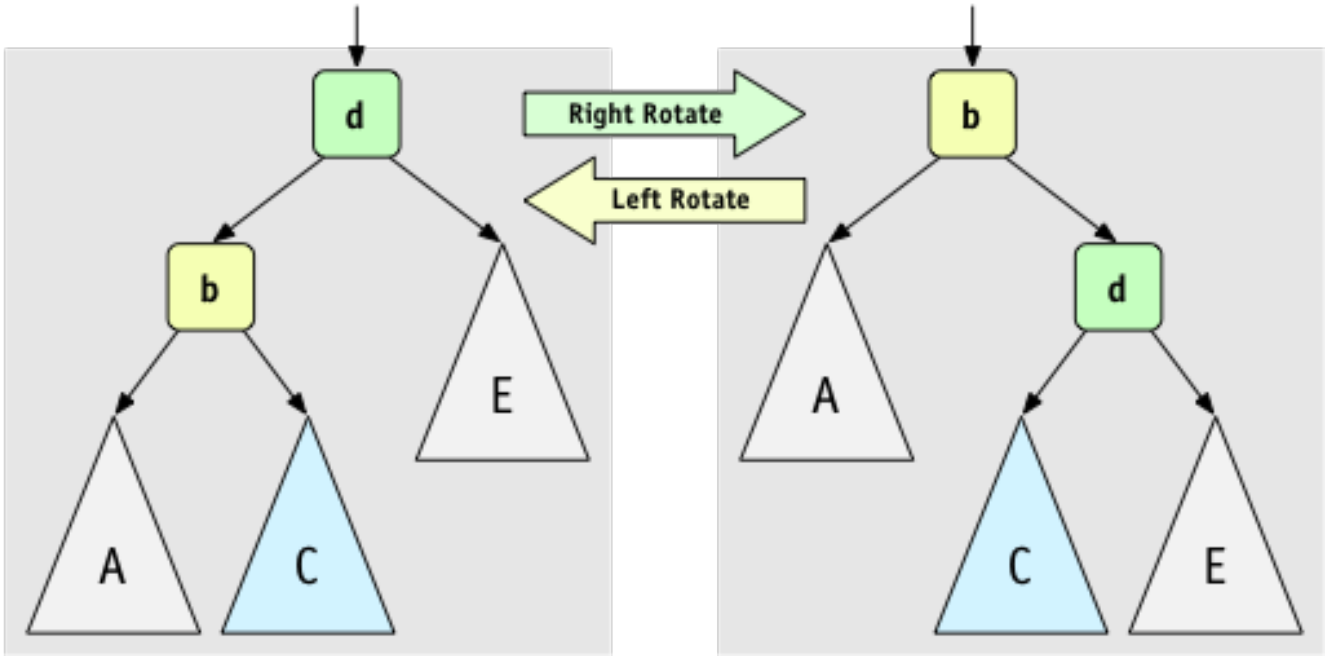
Generalized Single Rotation



Generalized Single Rotation



Single Rotations



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

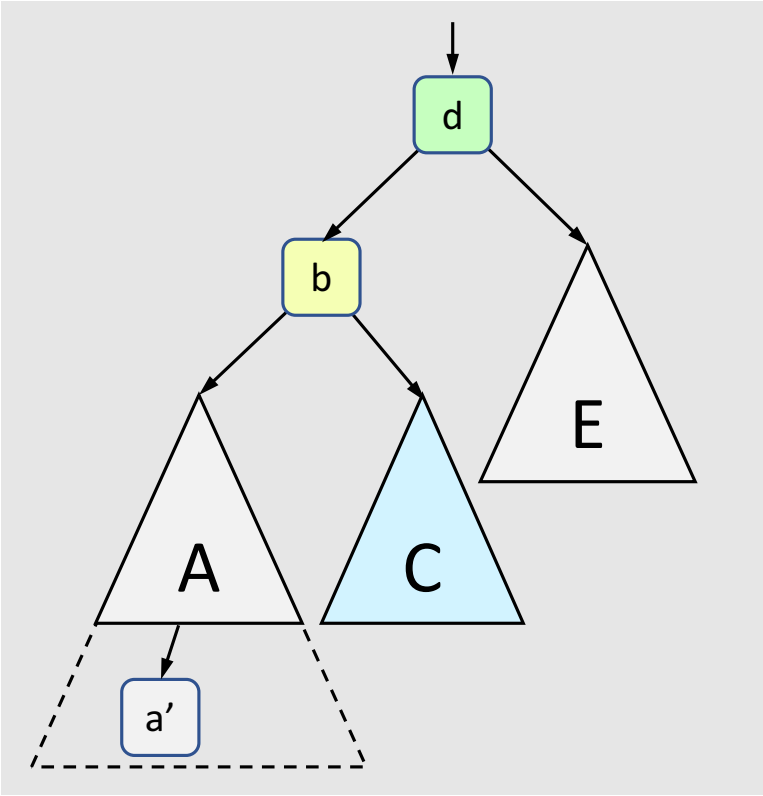
AVL Tree `insert` (more specific):

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

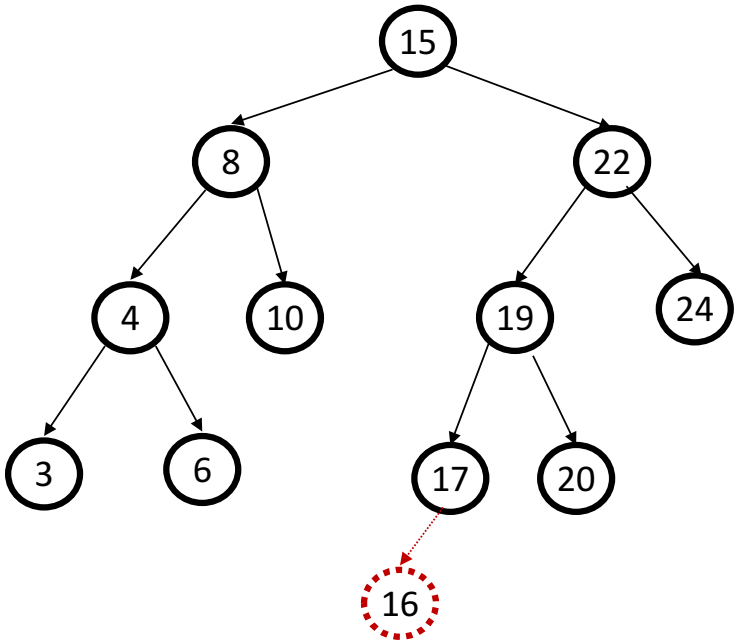
3. So after insertion in a subtree,

Case #1:

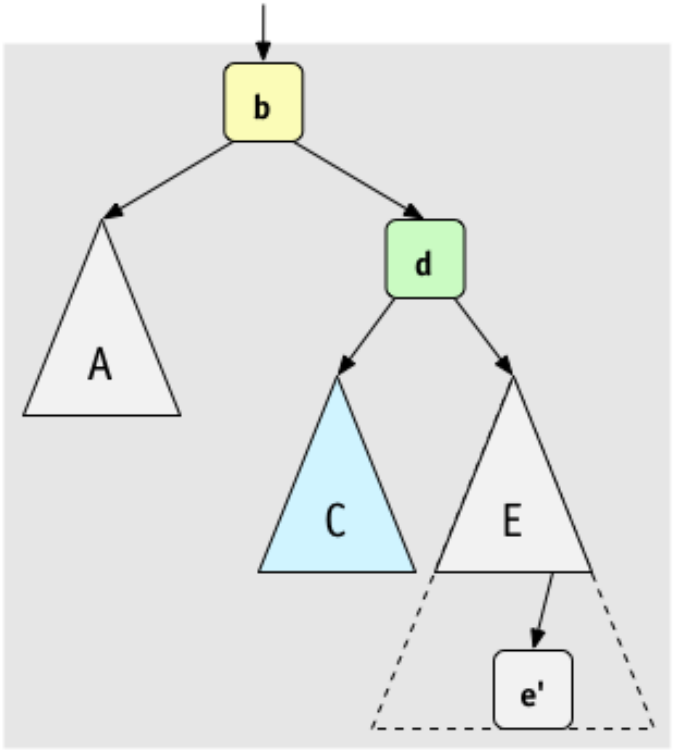


(Figures by Melissa O’Neill, reprinted with her permission to Lilian)

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

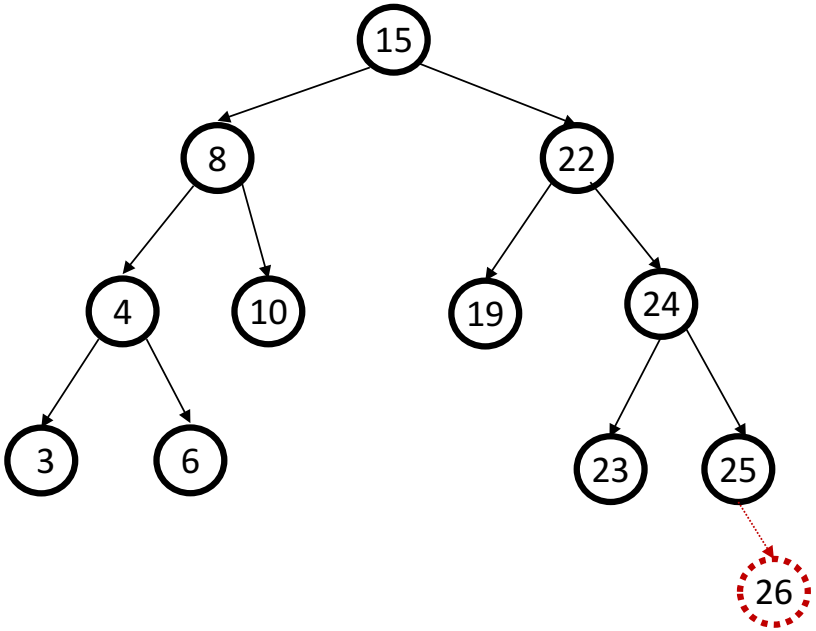


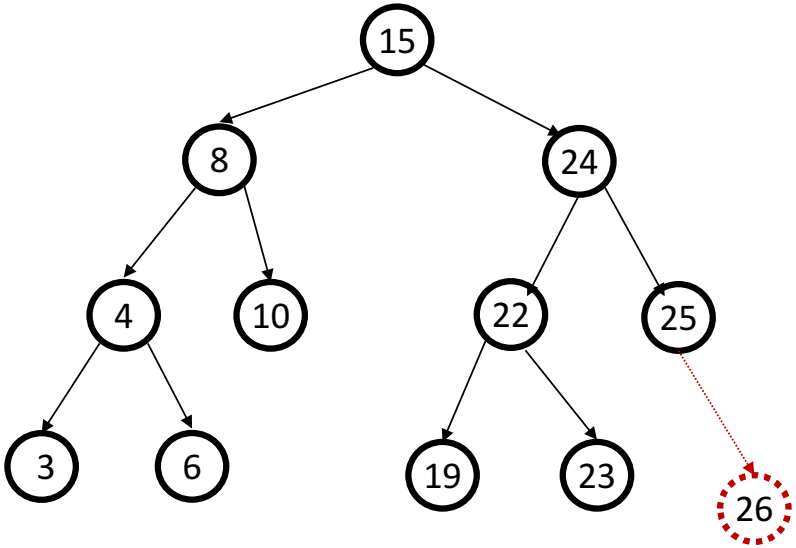
Case #2:



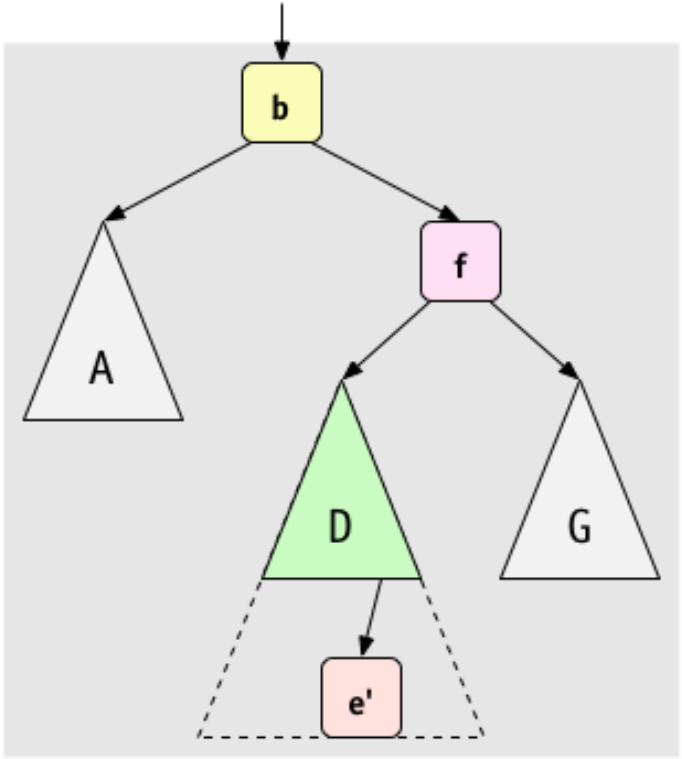
(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

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



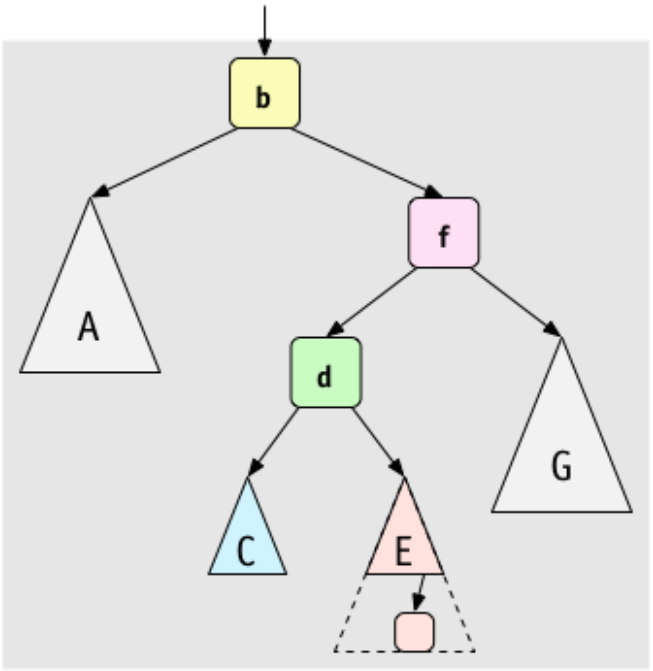


Case #3:



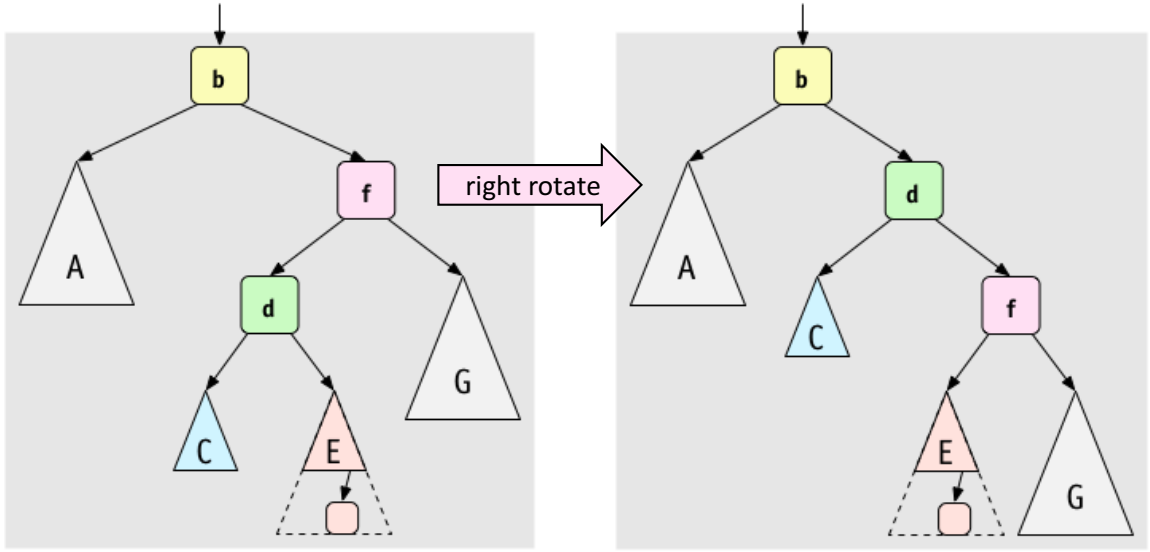
(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

A Better Look at Case #3:



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

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

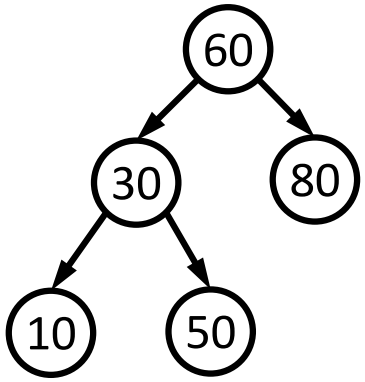


A way to remember it:
Move d to grandparent's position. Put everything else in their only legal positions for a BST.

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

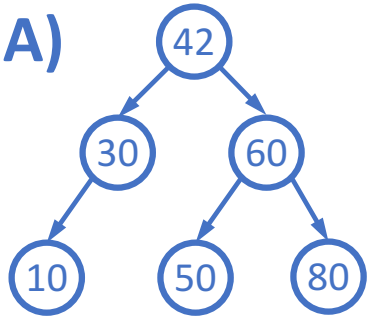
Practice time! Example of Case #4

Starting with this AVL tree:

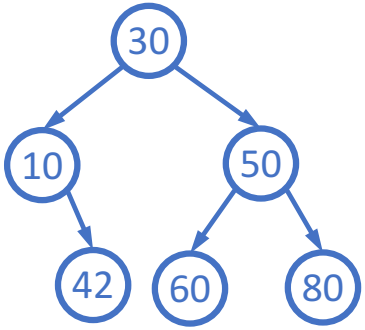


Which of the following is the updated AVL tree after inserting 42?

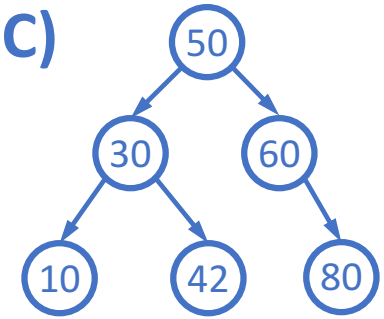
A)



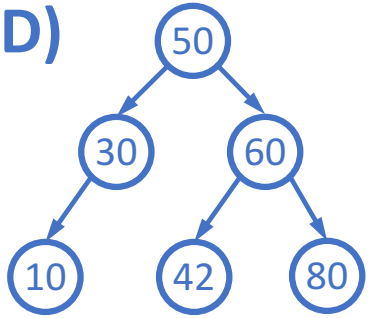
B)



C)



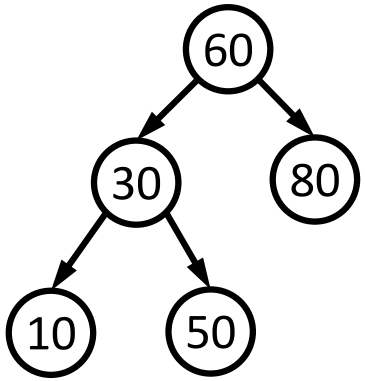
D)



(Extra space for your scratch-work)

Practice time! Example of Case #4

Starting with this AVL tree:



Which of the following is the updated AVL tree after inserting 42?

What's the name of this case?

What rotations did we do?

Insert, summarized

- Insert as in our generic 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 occurs because
- 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

AVL Tree Efficiency

- Worst-case complexity of `find`:

- Worst-case complexity of `insert`:

- Worst-case complexity of `buildTree`:

Takes some more rotation action to handle `delete`…

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced
2. Height balancing adds no more than a constant factor to the speed of `insert` and `delete`

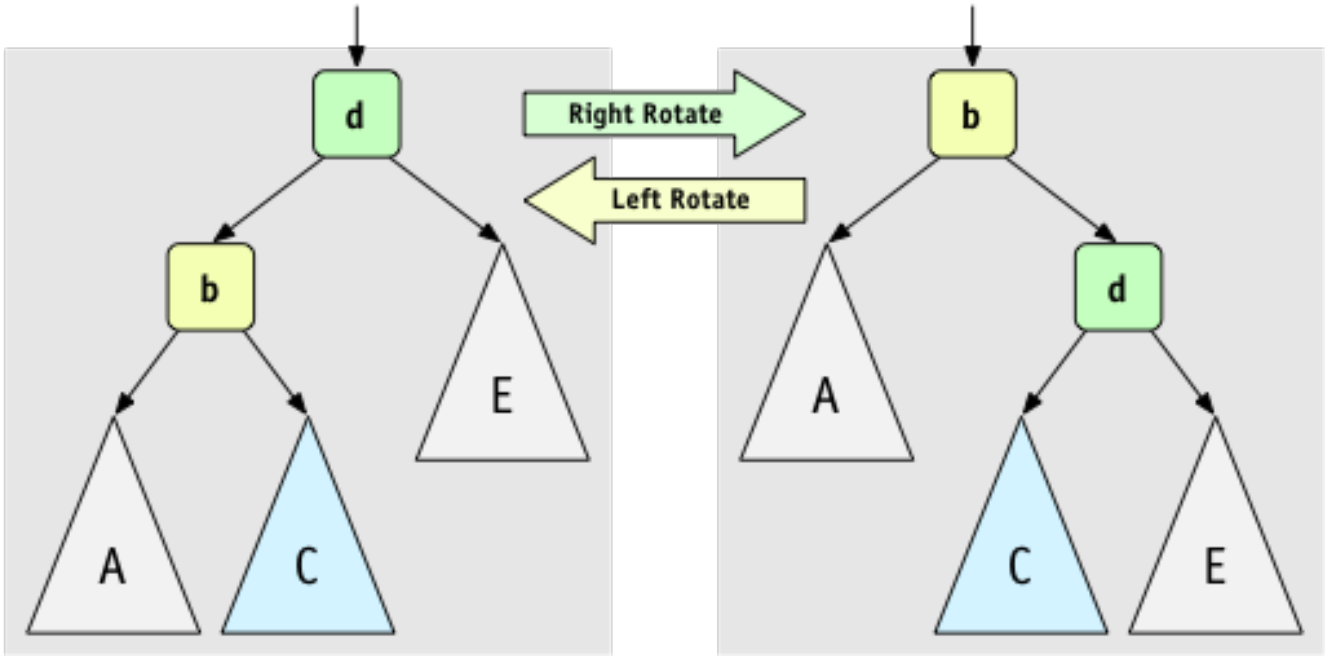
Arguments against AVL trees:

1. Difficult to program & debug [but done once in a library!]
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. If *amortized* logarithmic time is enough, use splay trees (also in the text, not covered in this class)

AVL Tree Rotation Cheat-Sheet

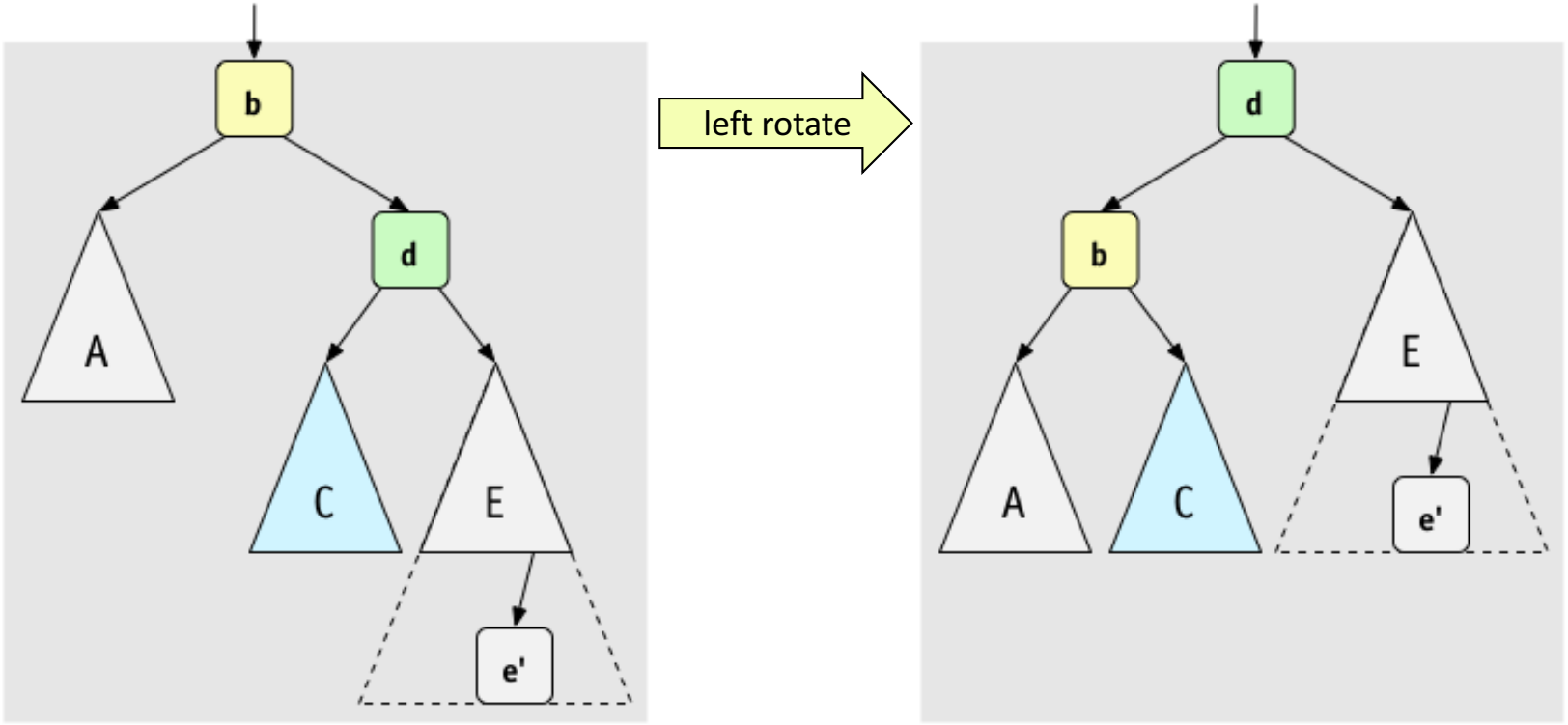
(Just two of the four cases)

Single Rotations



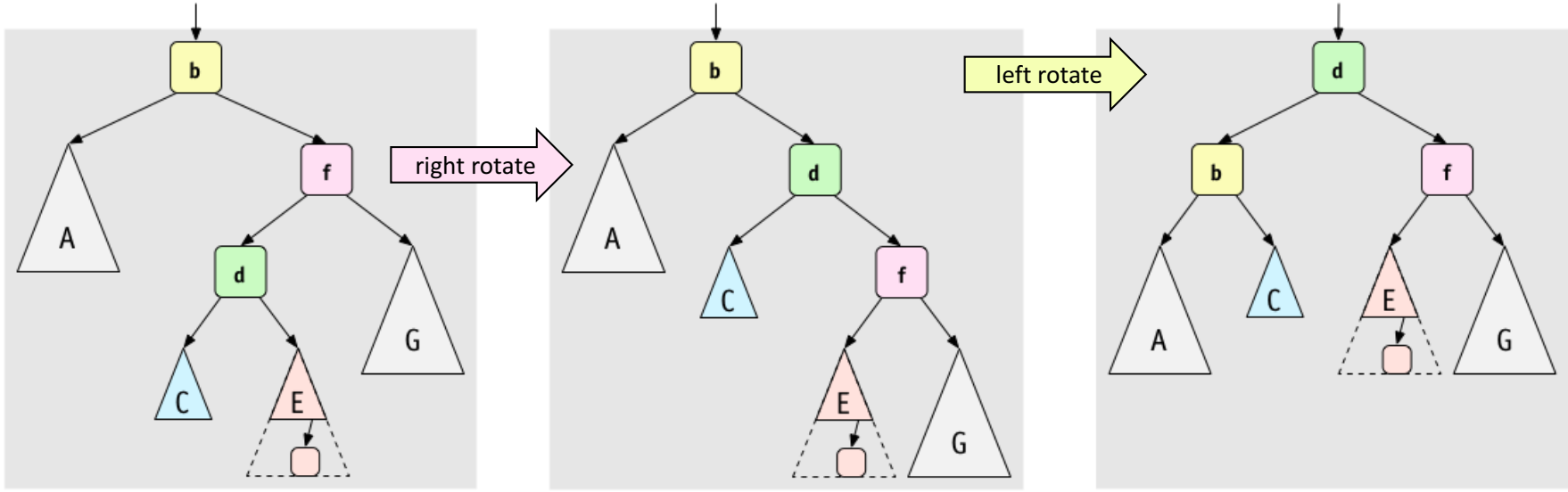
(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Case #2: Left-Left Case



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

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)