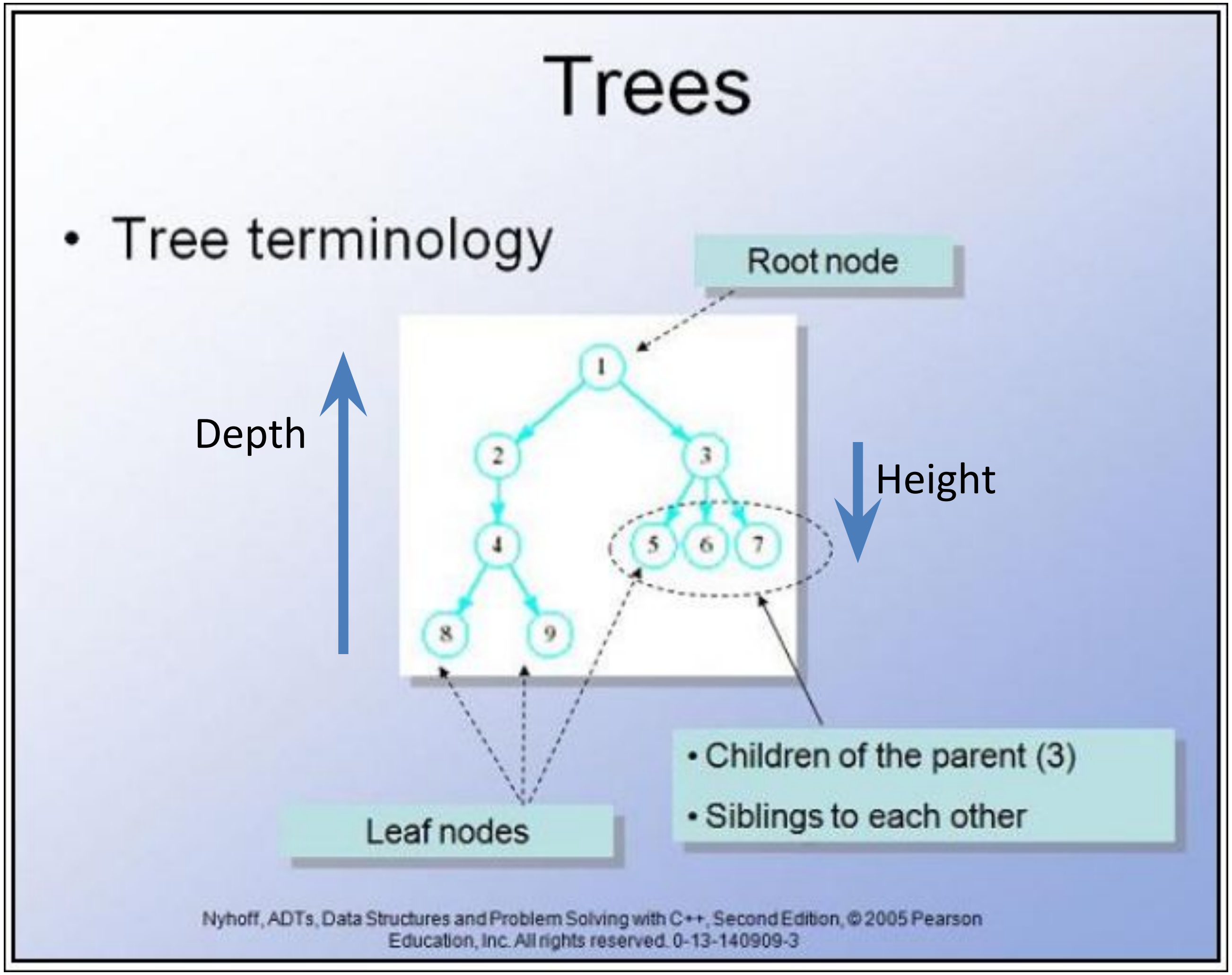
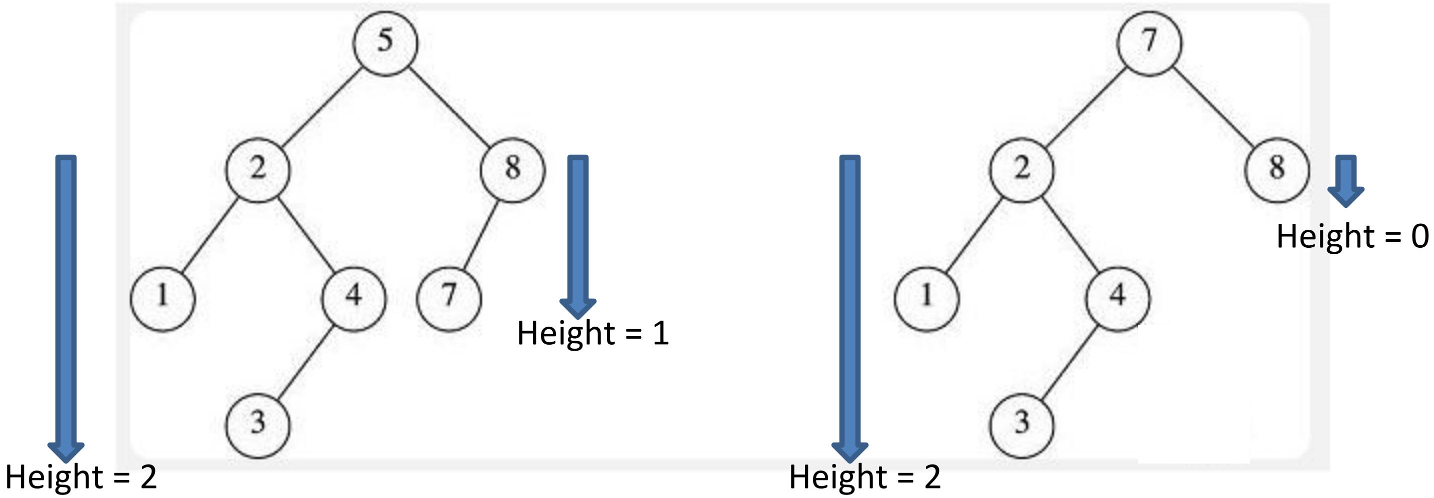
AVL Balanced Trees, Rotations & Splay Trees

Paul Hayter



- ***Depth*** – how many levels to the (sub)root is a node (looking up)
- ***Height*** – how many levels from a (sub)root to deepest child (looking down)
- ***Height*** is independent of the ***subtree*** we are considering, because ***height*** is measured *downward*, not upward. Height is a key feature of AVL trees.

Balanced and Unbalanced Trees

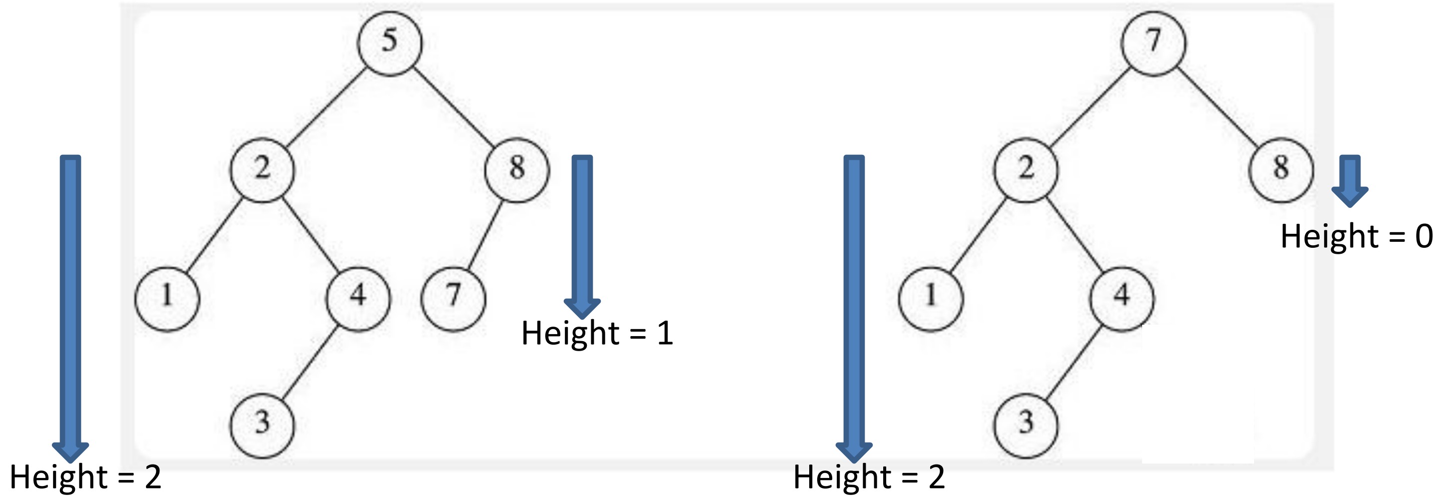


For both trees, what is the height of the two subtrees of the root?

An ***AVL balanced tree*** is when subtree heights differ by no more than 1.
Which tree is balanced? Why?

Balance Factor = Height(Right subtree) – Height(Left subtree)
(though some texts write this the other way)

AVL Balanced Trees

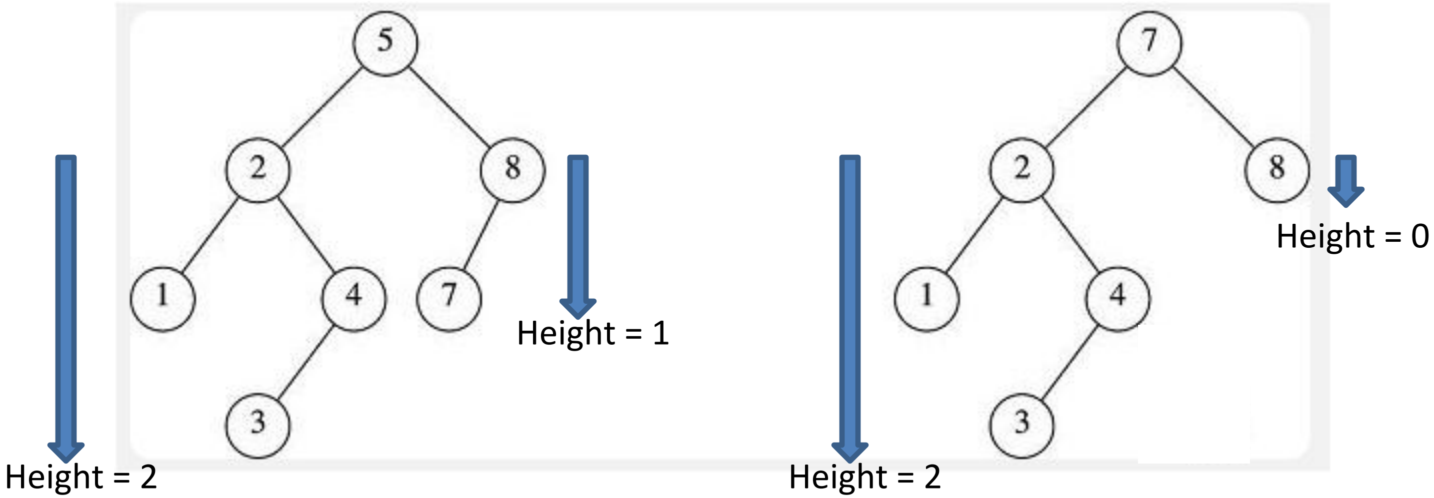


This tree meets ***AVL condition*** at the root **(5)**, since its left subtree **(2)** has height 2, and its right subtree **(8)** has height 1.
Balance Factor = 1 – 2 = -1

This tree violates ***AVL condition*** at the root node **(7)** because its left subtree **(2)** has height 2, while its right subtree **(8)** has height 0.
Balance Factor = 0 – 2 = -2.

Except for the root, *both* trees have ***AVL*** balanced nodes, so in this case, it is only the root node on the right example that is not ***AVL*** balanced.

Computing Height



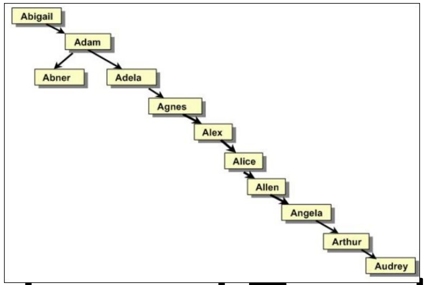
Height = max(leftChildHeight, rightChildHeight) + 1

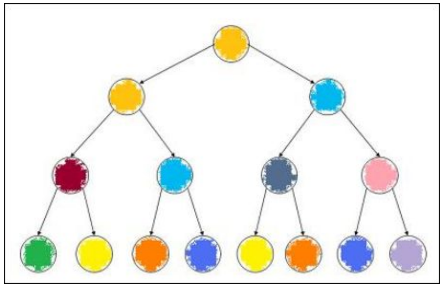
Discussion:

- What are the heights of the root nodes of these trees?
- What is the height of a null node (i.e., a child of a leaf node)?
  (Hint: consider the children of Node 8 of the unbalanced tree on right.)

Balanced Trees

- Discussion:
  - Is an AVL tree a Binary Search Tree? Explain.
  - What is the balance condition for an AVL Tree?
- Worked Exercise
  - Draw an Unbalanced Tree
  - Draw a Balanced Tree





- What advantage does a Balanced Tree have over an unbalanced Tree?
- What is the Big-O of searching a Balanced Tree?

Rotations

Discussion:

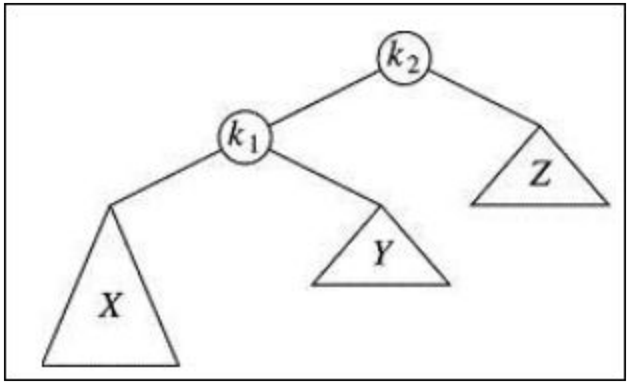
- What is a rotation and when is it used?
- Weiss' text notation (which is common) is:
  - Right rotation when nodes move to the right
  - Left rotation when nodes move to the left
  - Note: Loceff modules have the ***opposite*** meaning

Note: Be aware that double rotation terminology in the Weiss text is different from the rotation language used in the Loceff modules.

Loceff Module	Weiss' Text
Double Left	Right-Left
Double Right	Left-Right

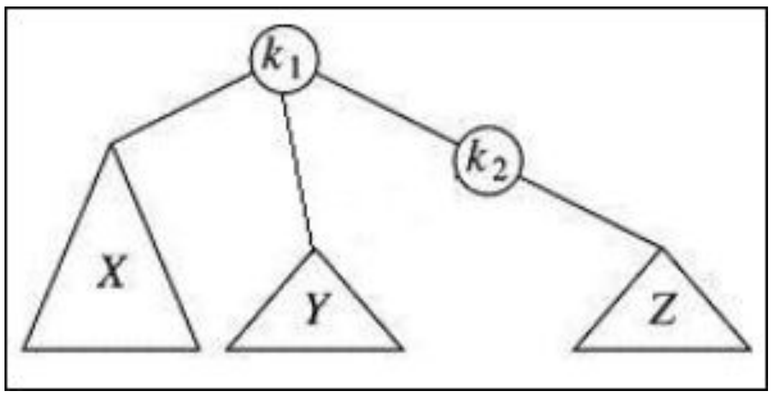
Rotations: Right Rotation Example

Initial unbalanced tree



Weiss Notation (opposite of Loceff) for ‘left’ & ‘right’

Discussion: Why is this balance operation legal for a Binary Search Tree (BST)?



Illegal tree: intermediate state

For visualization only!!!

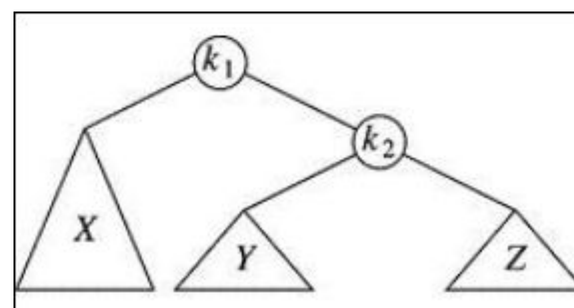
Note:

k_1 → X unchanged

k_2 → Z unchanged

Note that Y is still less than k2 and greater than k1

Final balanced tree

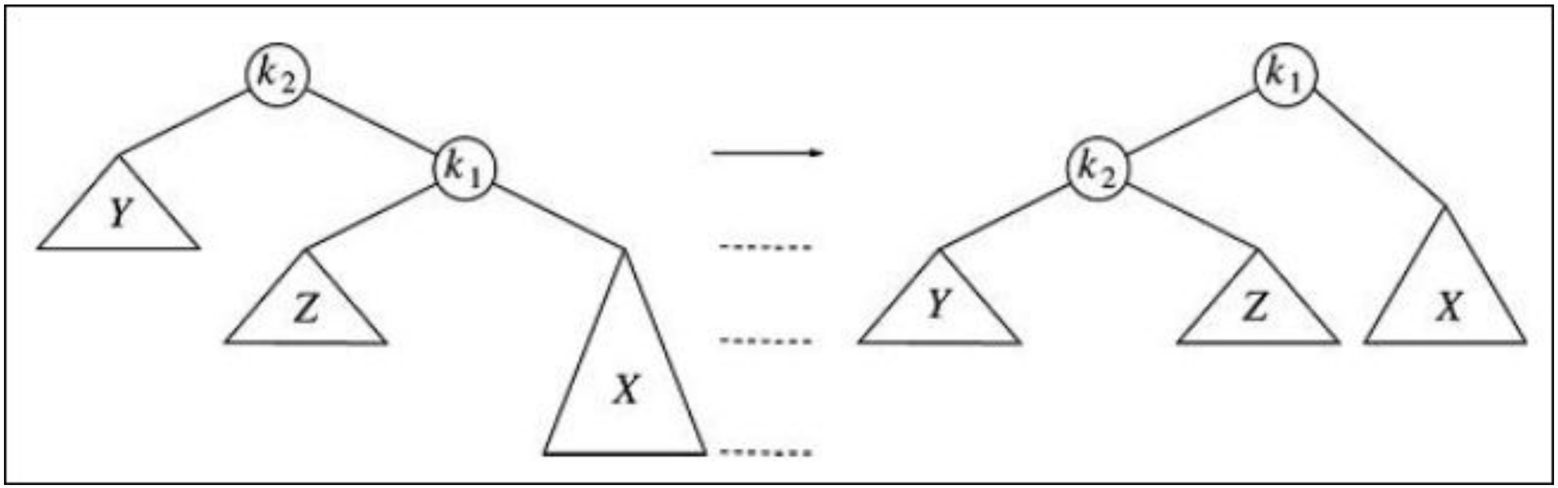


Rotations: Left Rotation Example

Weiss Notation (opposite of Loceff) for ‘left’ & ‘right’

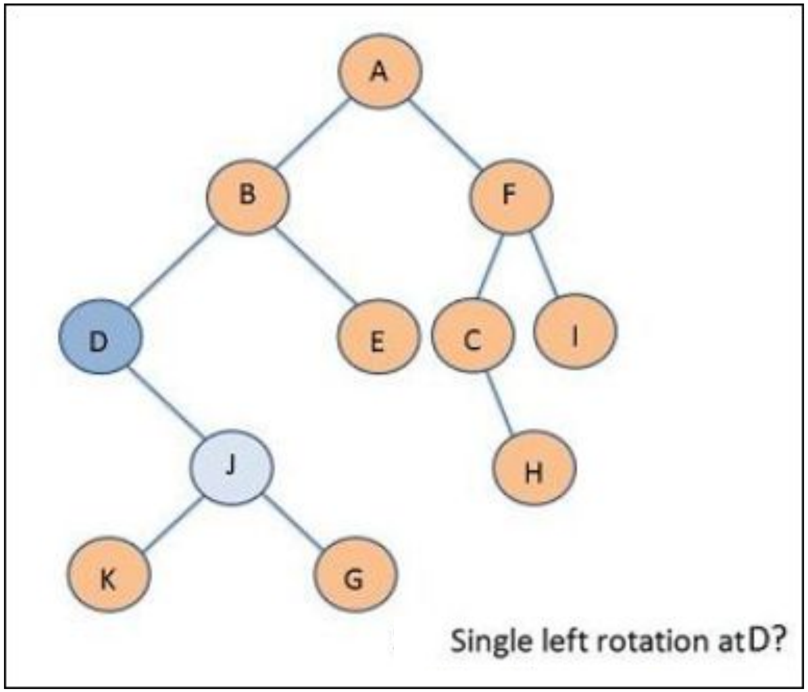
Initial unbalanced tree

Final balanced tree

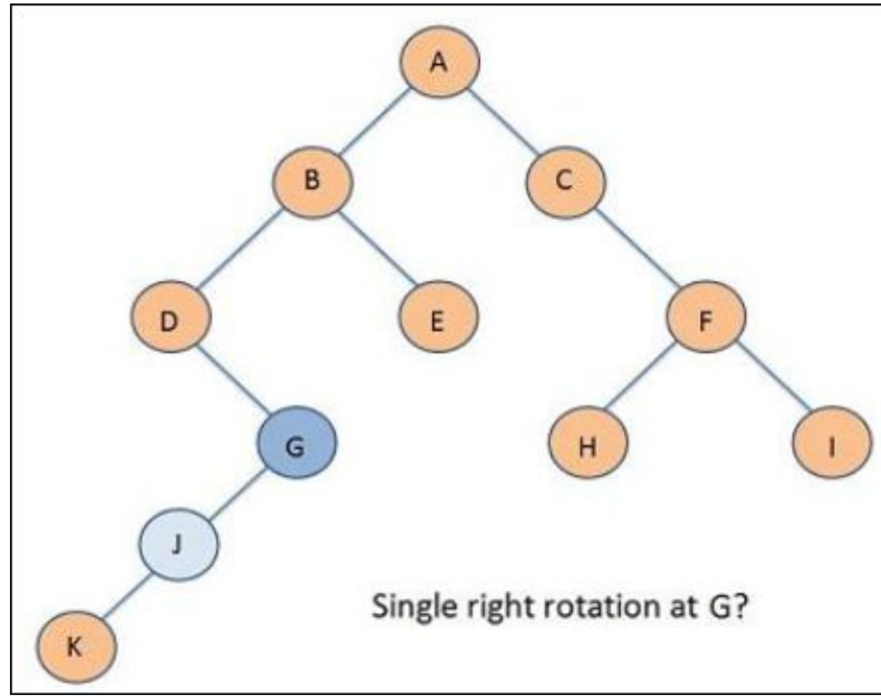


Rotation Exercises

Weiss Notation (opposite of Loceff) for ‘left’ & ‘right’



Problem 1

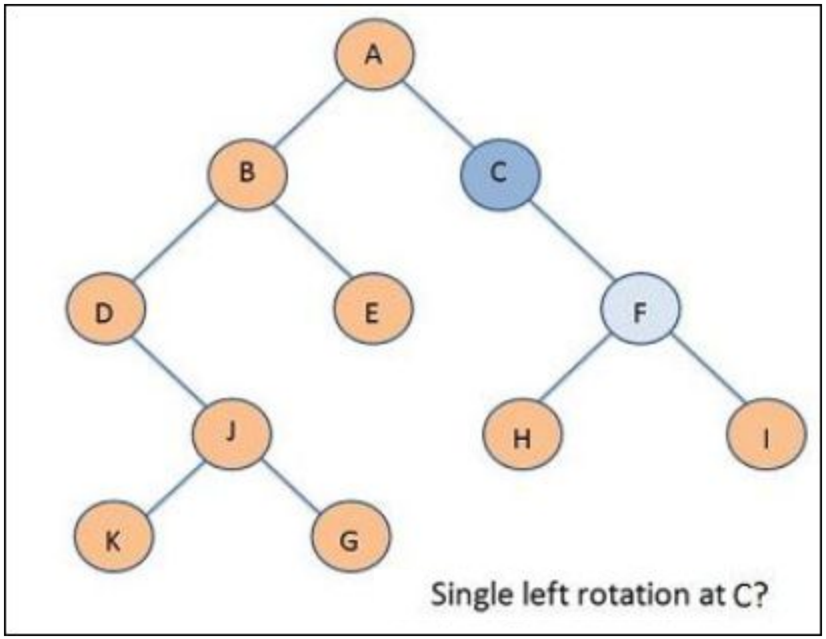


Problem 2

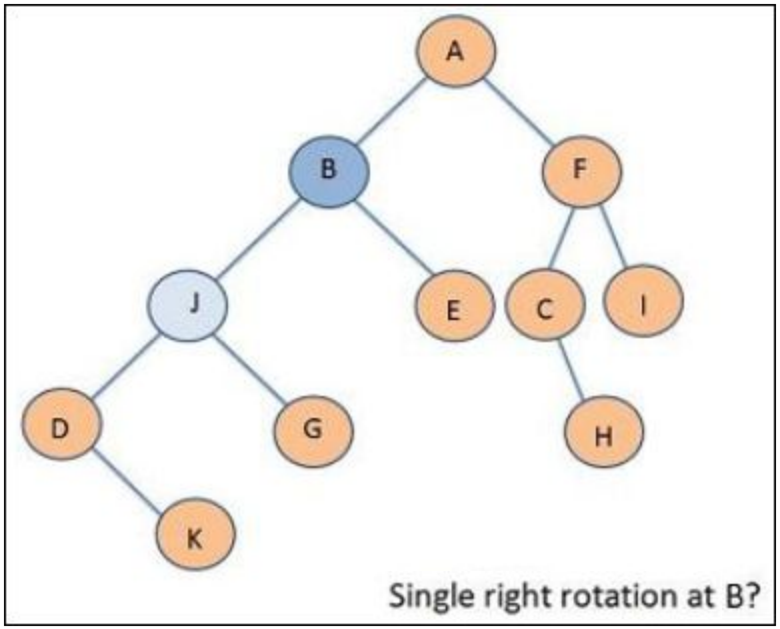
Note: For practice only – these rotations won’t balance the trees.
Also, Node letters are for ID only and are not keys.

Rotation Exercises

Weiss Notation (opposite of Loceff) for ‘left’ & ‘right’



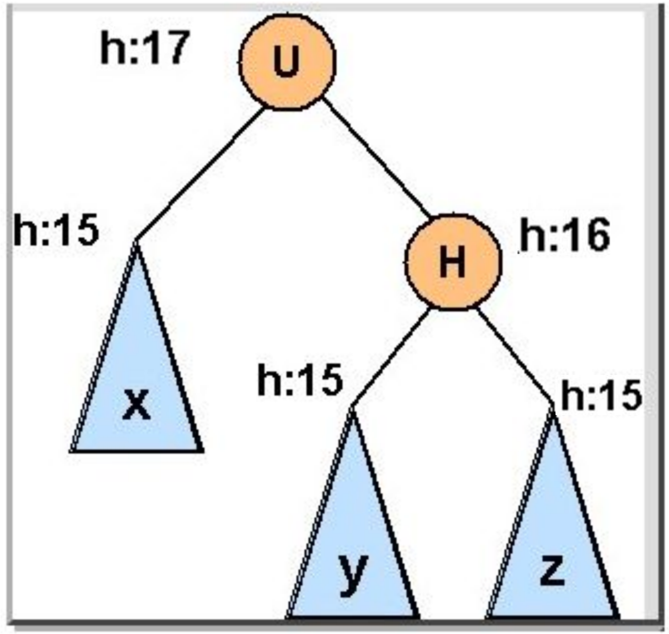
Problem 3



Problem 4

Note: For practice only – these rotations won’t balance the trees.
Also, Node letters are for ID only and are not keys.

Rotation Exercises



Discuss:

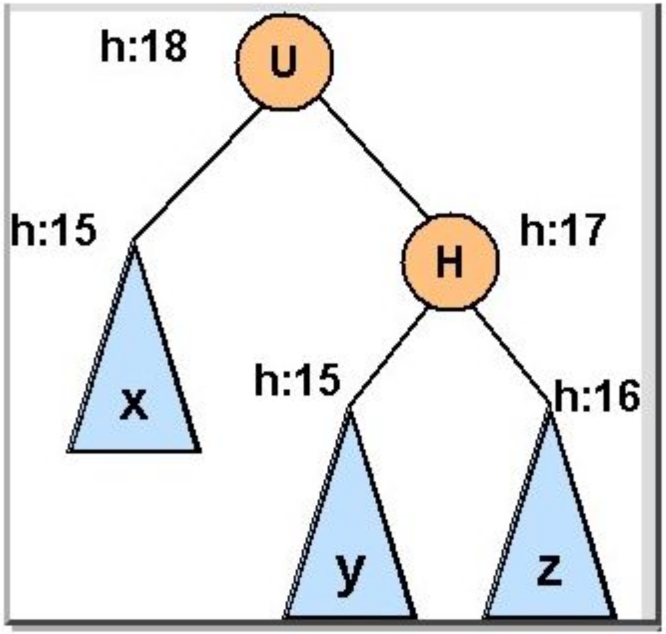
- Is this tree balanced?
- Is it balanced after insertion into Y-subtree with height increase?
- Is it balanced after insertion into Z-subtree with height increase?

Remember: Height = max(left_child_height, right_child_height) + 1

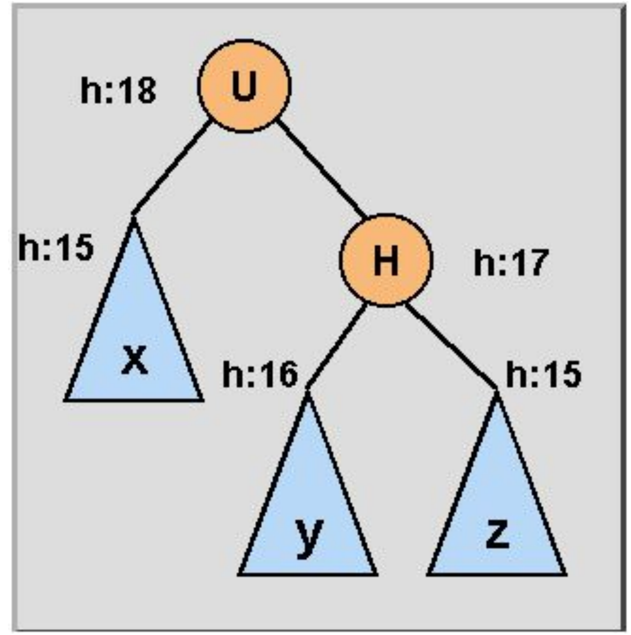
Based on image from: secweb.cs.odu.edu/~zeil/cs361/web/website/Lectures/avl/pages/doublerot.html

Rotation Exercises

Weiss Notation (opposite of Loceff) for ‘left’ & ‘right’



White Tree



Grey Tree

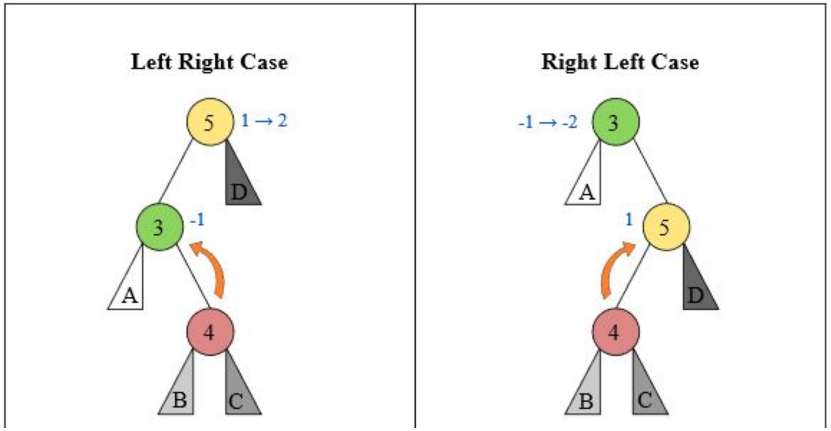
1. Rotate Left both trees and annotate changes in height (the h values)
2. After rotation, which tree(s) are balanced?
3. How could a balanced outcome be predicted?
4. Is this prediction the same for a Rotate Right?

Remember: Height = max(leftChildHeight, rightChildHeight) + 1

Images: https://secweb.cs.odu.edu/~zeil/cs361/web/website/Lectures/avl/pages/doublerot.html

Double Rotation

In Left-Right Case rotate into a Left-Left Case to set up for a balanced rotation
Left-Right Case → Left-Left Case → Balance

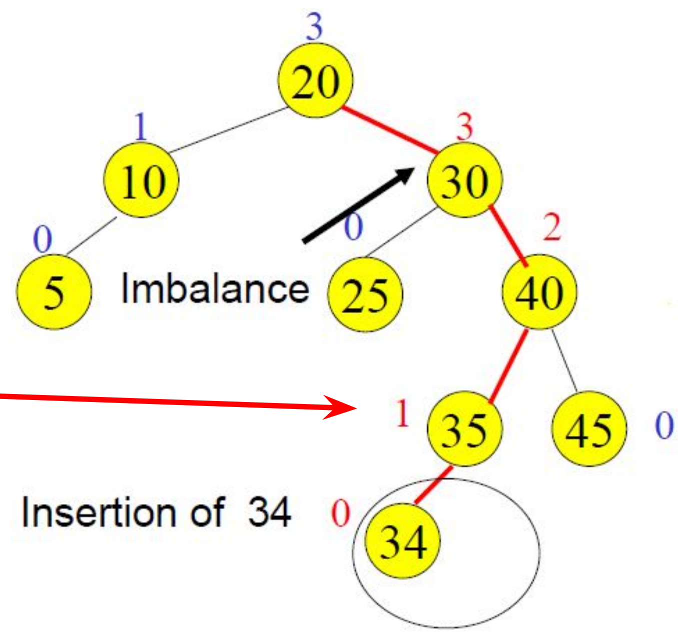


Note how this is only necessary when the tree is 'heavy' on the ***inside***, that is, of greater height on the tree's inside

Note the symmetry in applying the 'heavy inside' rule in both cases

Image: Mikel McDaniel

Double Rotation Exercise



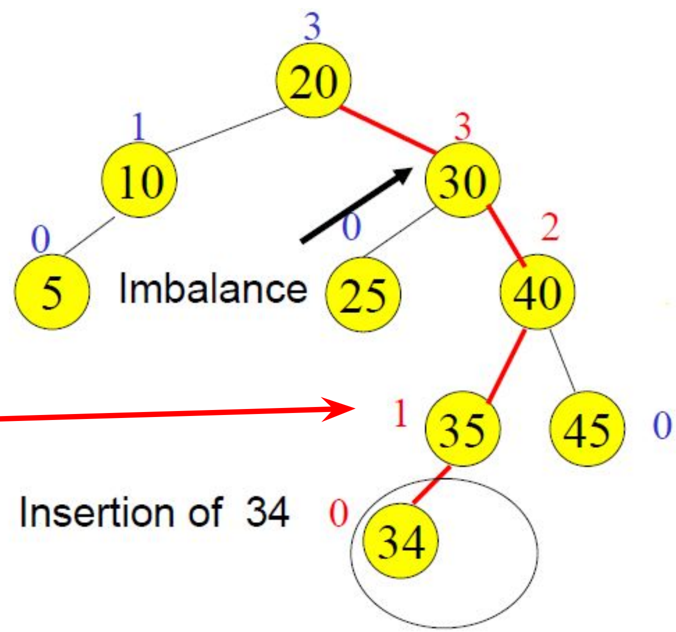
Updated heights in red

History of this tree

1) Inserted Node 34
2) Updated Node heights going up from node 34 *one parent at a time* (updated heights shown in red)
3) Checked Balance Factor after each height update

Based on Image from: *courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture12.pdf*

Double Rotation Exercise

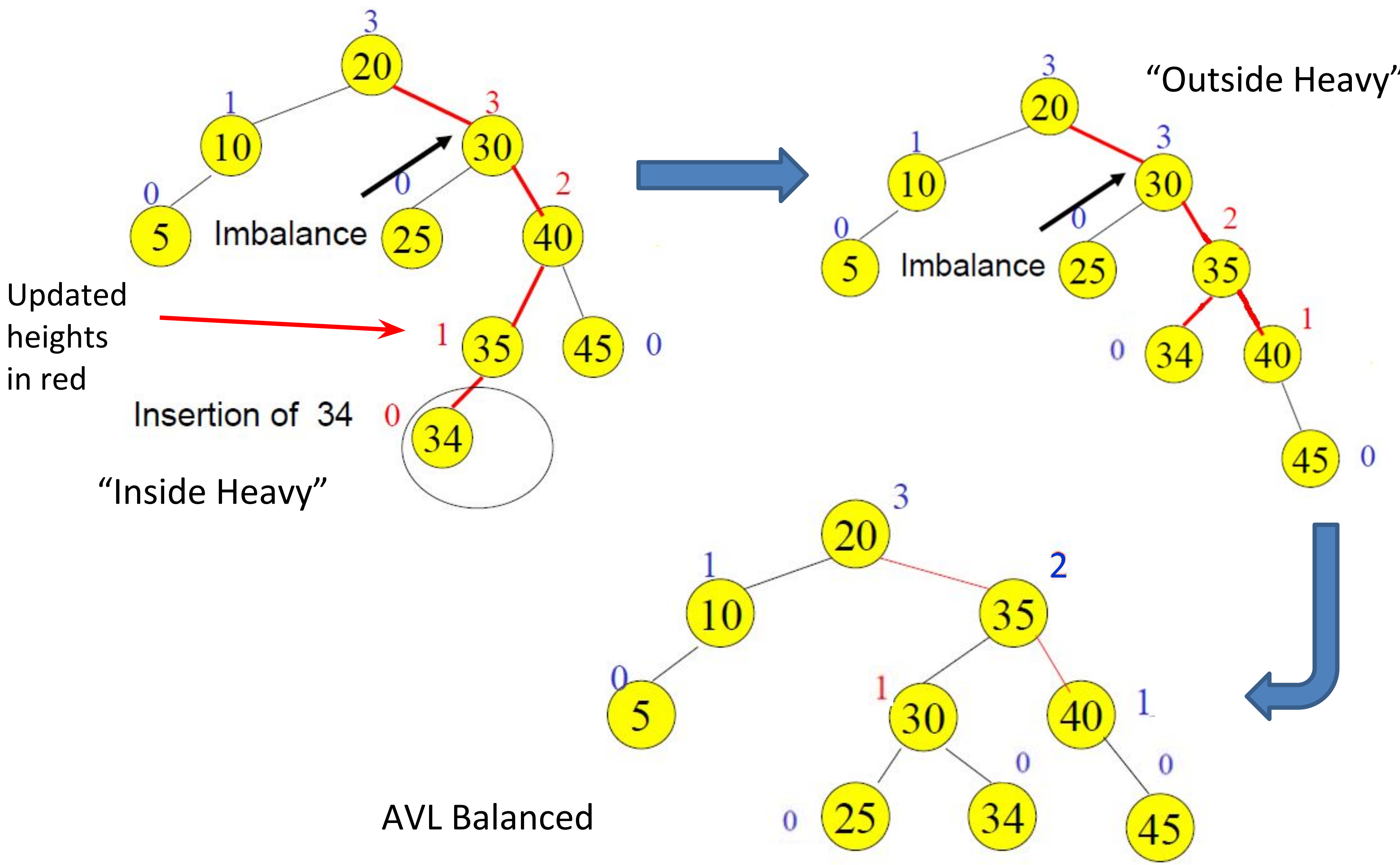


Updated heights in red

1) Why imbalance at Node 30?
2) Is tree 'heavy' on the inside?
3) What rotations should be done and at which nodes?
4) What is the result of these rotations?

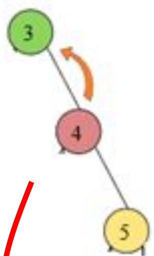
Images:
https://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture12.pdf

Double Rotation Exercise

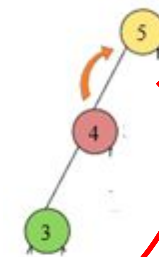


Images: https://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture12.pdf

AVL Rotation Summary



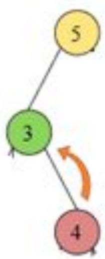
Let the node that needs rebalancing be α.



There are 4 cases:

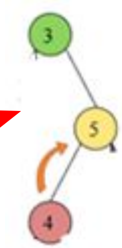
Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.



Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

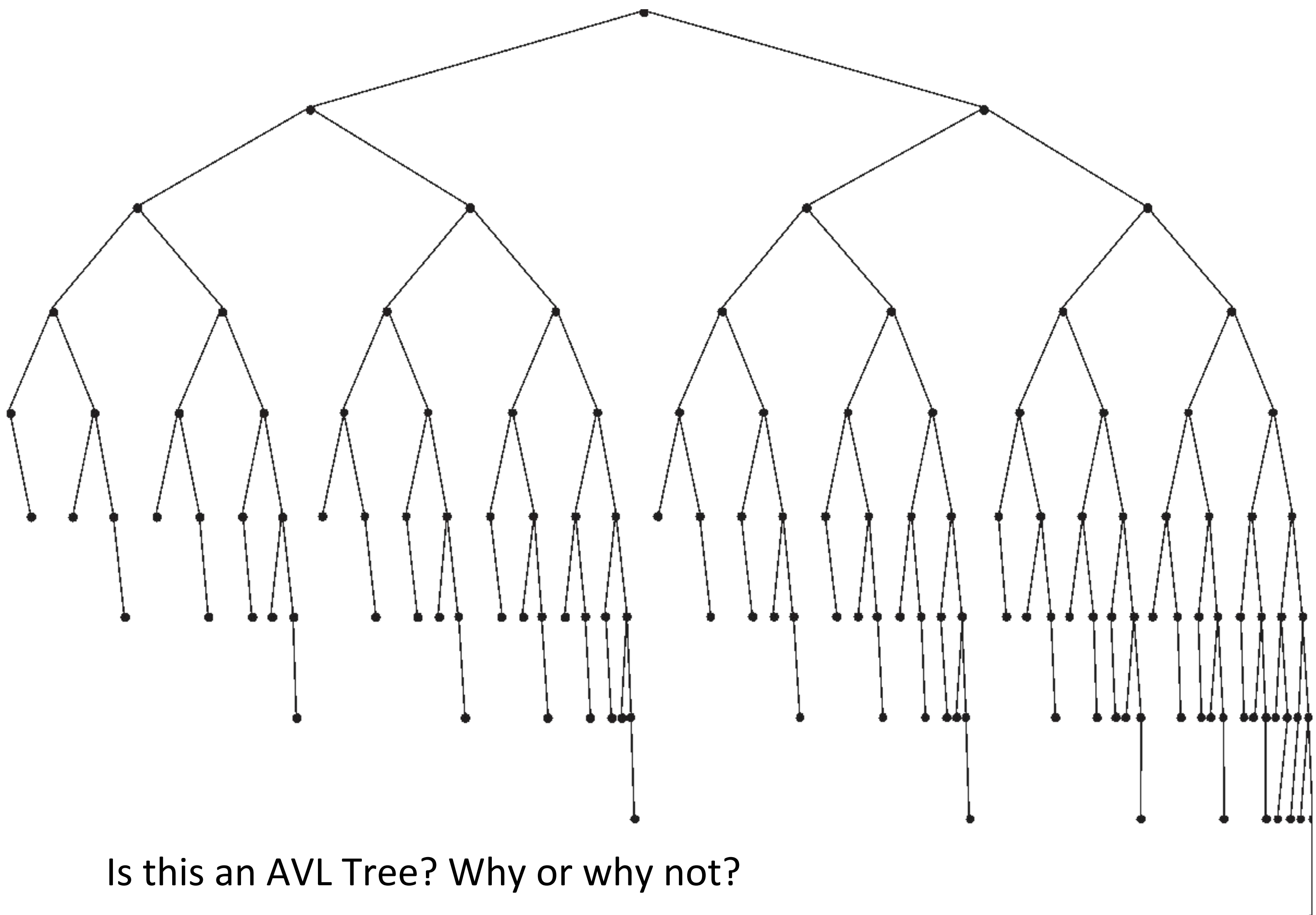


The rebalancing is performed through four separate rotation algorithms.

Note: Arrows show first (or only) rotation.

Summary: *https://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture12.pdf*

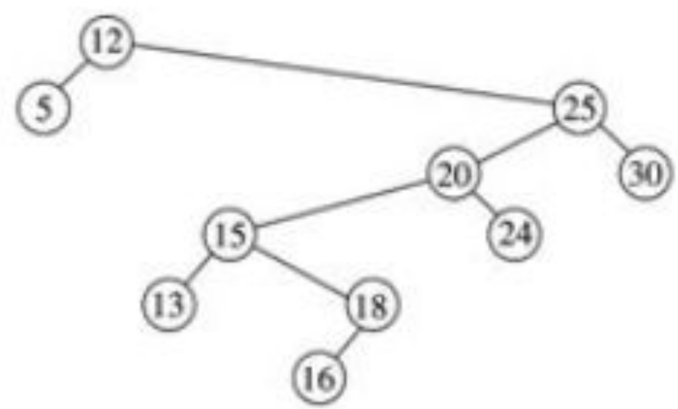
AVL Tree?



Is this an AVL Tree? Why or why not?

Splay Trees

- Discussion: Is a Splay Tree a Balanced Tree? Explain.
- What are the advantages of a Splay Tree?
- How might those advantages affect when this choice of tree is used?



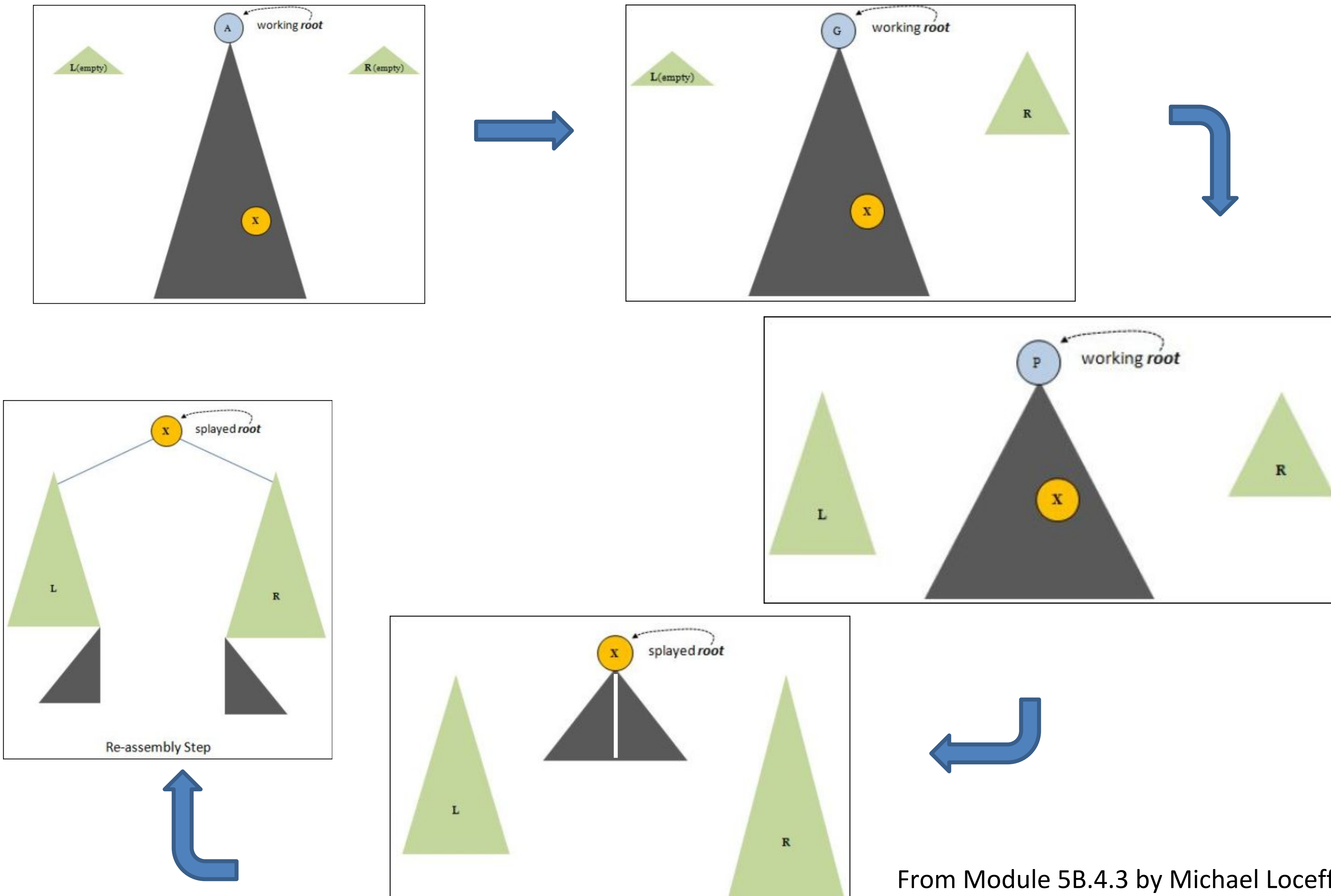
Example Splay Tree from Weiss Text

Splay Algorithm Explained

- First Pass Explanation
  - Top-level and general description
  - Rough sketches missing details
- Second Pass Explanation with navigation details
  - More detailed description
  - More detailed sketches

Splay Tree Algorithm Concept

Search for X

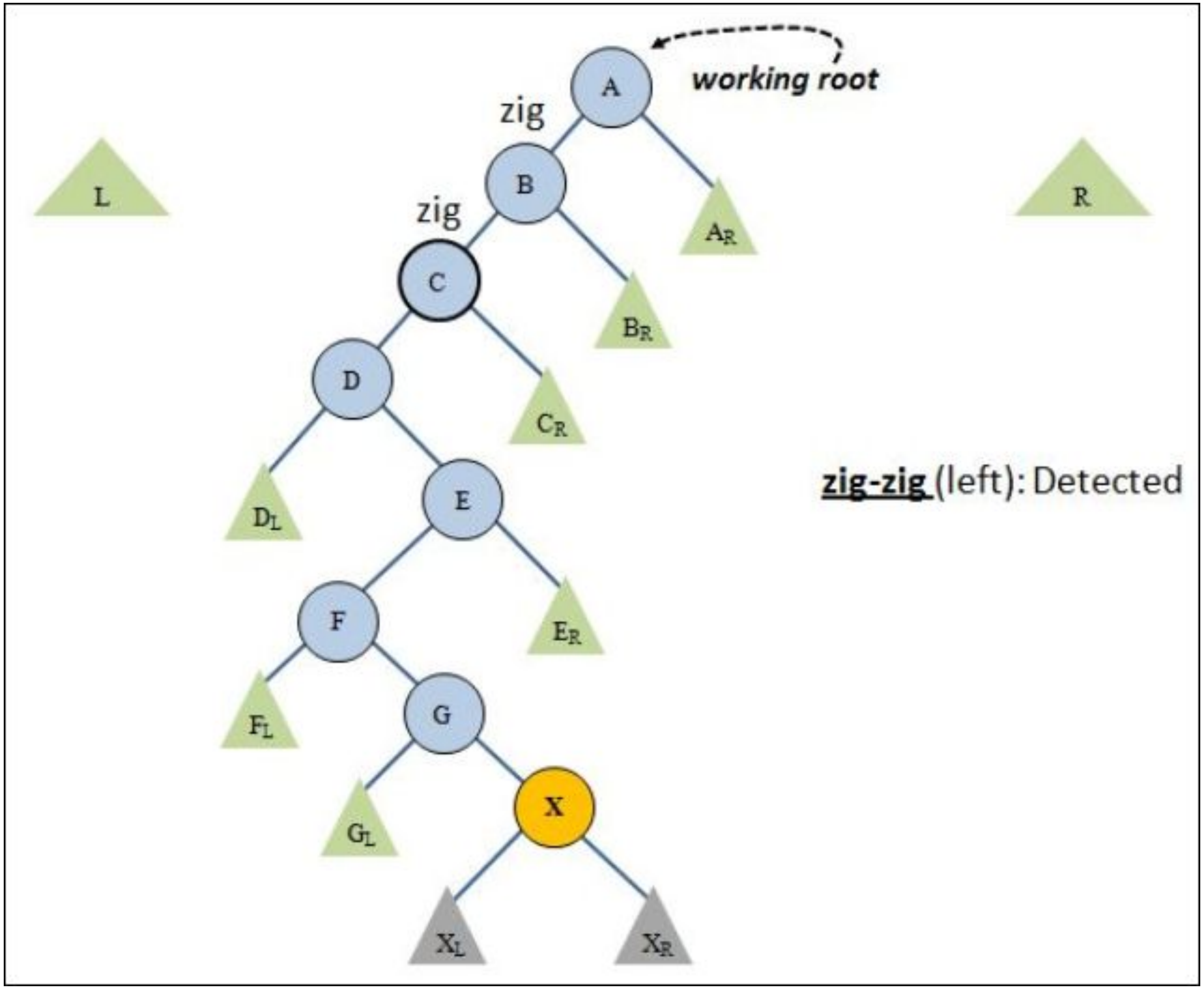


From Module 5B.4.3 by Michael Loceff

Splay Tree Navigation (Detail)

- As we search down the tree looking for X, the following can happen:
  - We move twice to the Right or Left (through those children) – called a *zig-zig*
  - We move Right-Left or Left-Right (through those children – called a *zig-zag*
- Need to know how to disposition those nodes that we move through

Splay Tree Navigation



A *zig-zig* is TWO sequential data "checks" in the same direction when looking for X. Left-Left or Right-Right

Now we know:

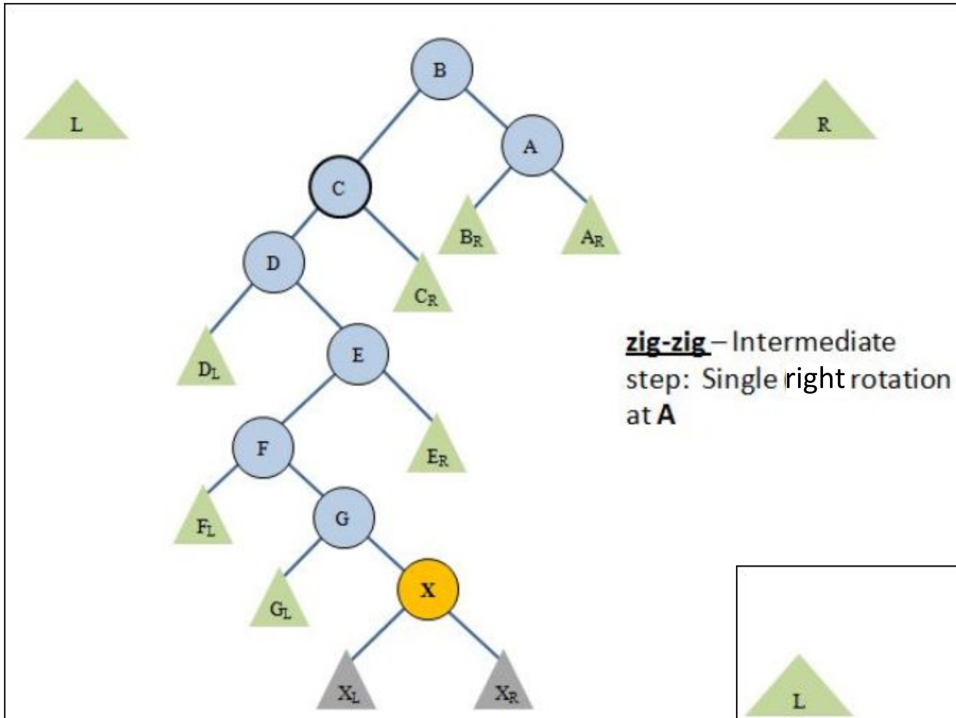
1) X is not A or B
2) Since Left-Left, then X < A & B

Next Steps:

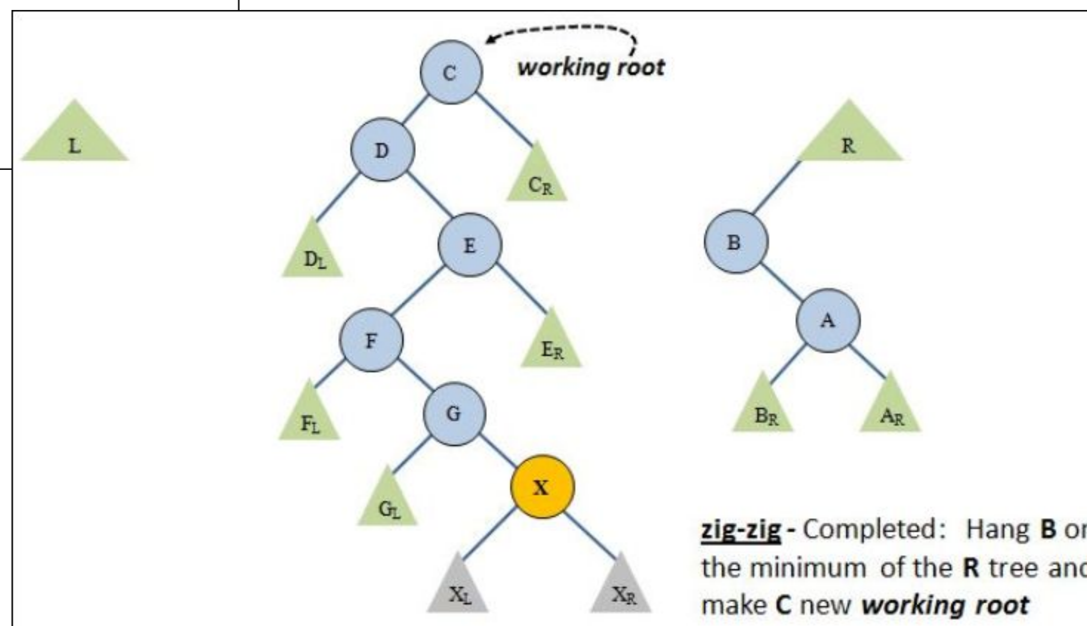
1) Single rotation Right at A
2) Remove A & B to minimum of R-Tree
3) Appoint C as new root

Note: this image starts at an *intermediate* phase of splaying so **L** and **R** have content

Splay Tree Navigation: zig-zig



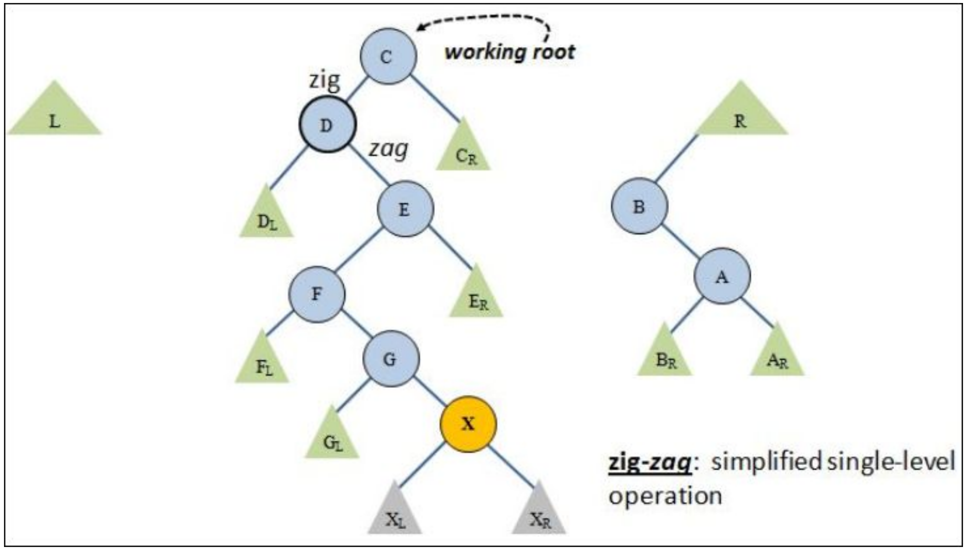
Remember:
Moved to R-Tree since X < A & B



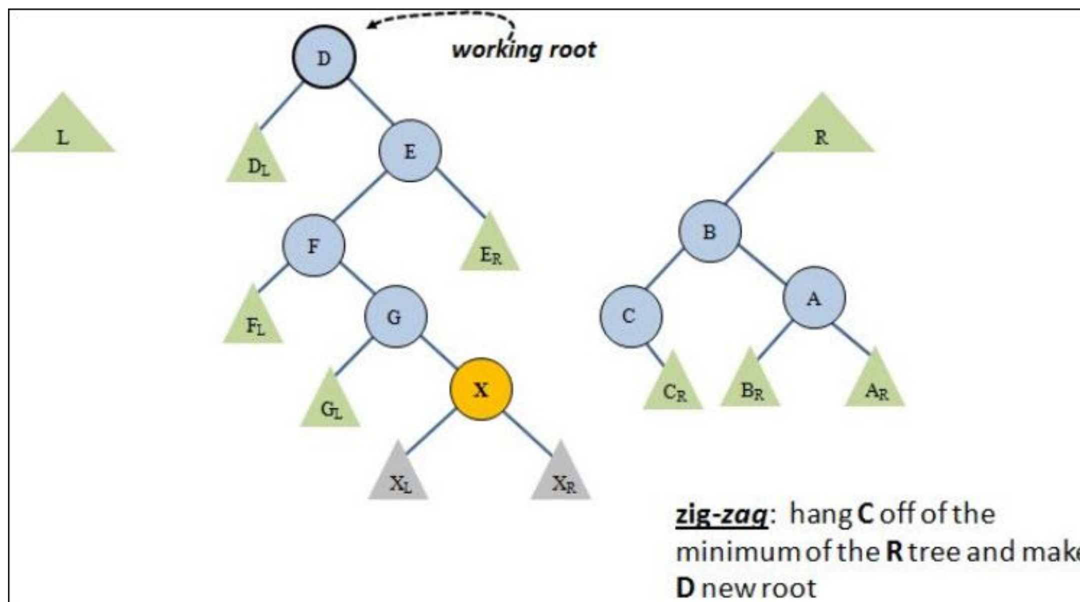
If *zig-zig* had been Right-Right, these operations would be reversed since X > A & B:

1) Single Left rotation
2) Move to L-Tree maximum

Splay Tree Navigation: zig-zag



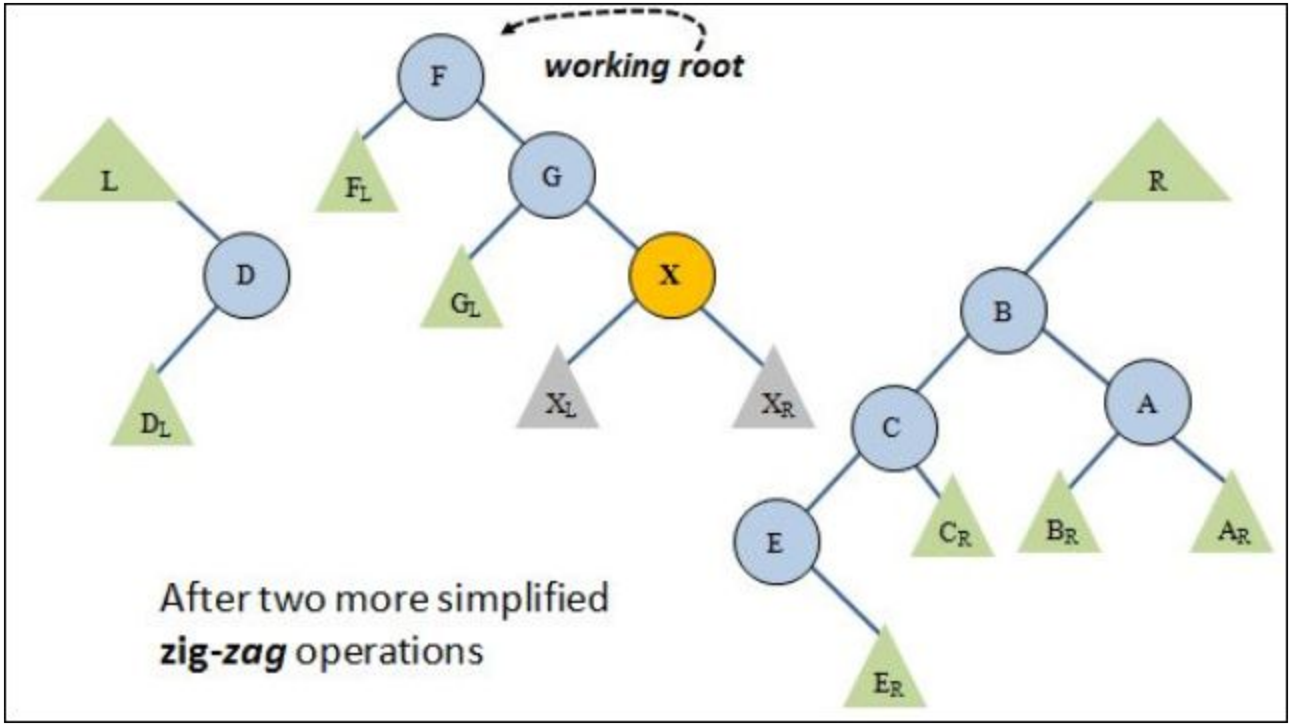
Remember:
Moved to R-Tree since X < C



If *zig-zag* had been Right-Left, these operations would be reversed since X > C:

1) Move to L-Tree maximum
2) (No rotation in either)

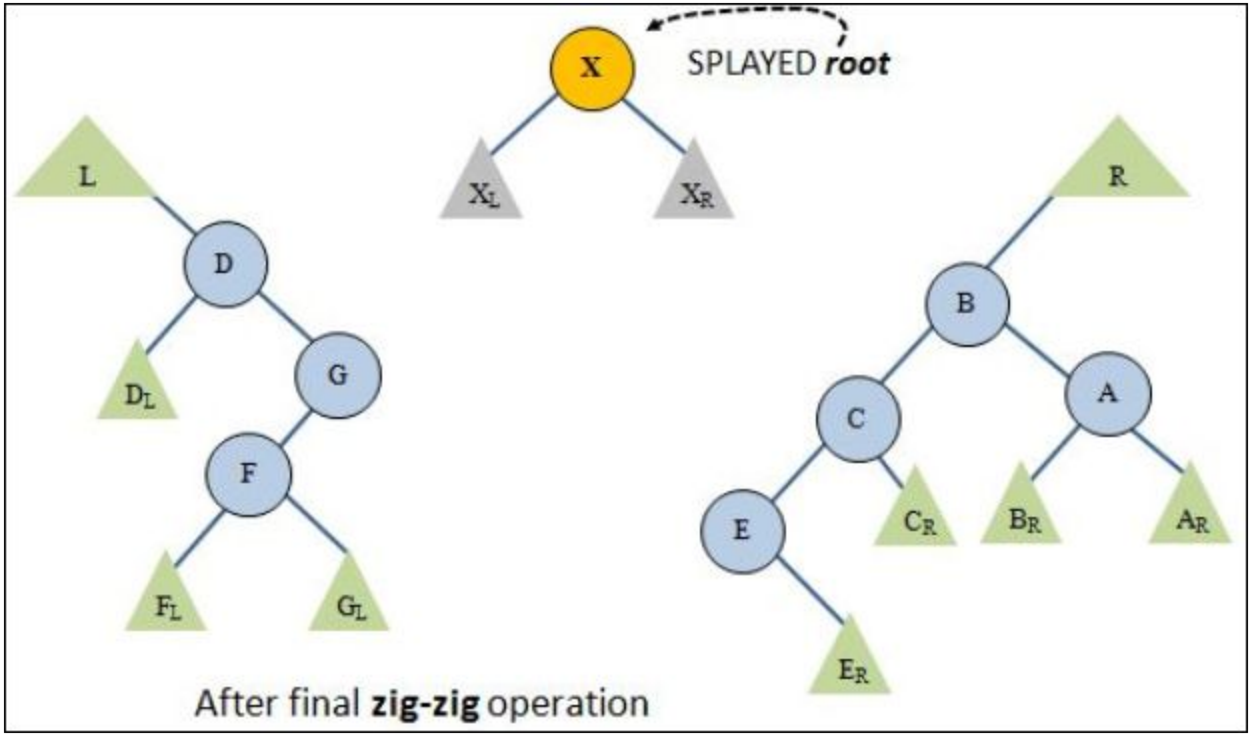
Splay Tree Navigation: Discussion



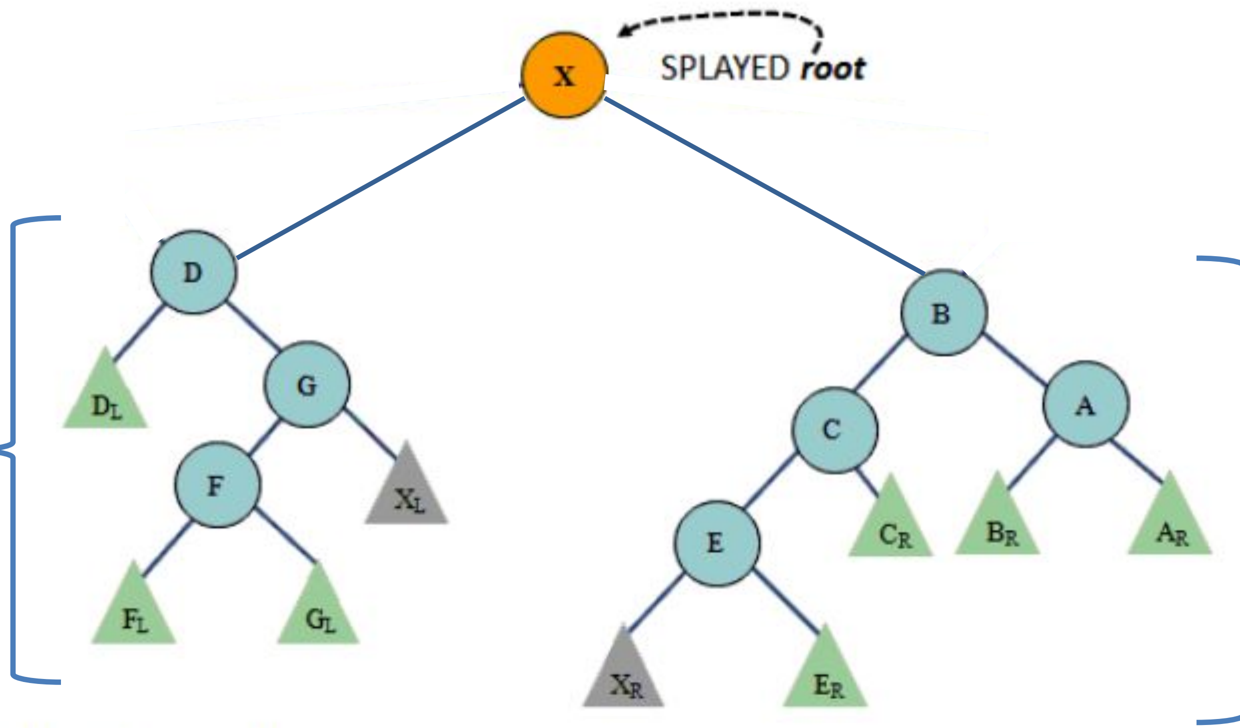
Discussion: What operations should be done now?

1) Rotate Left at F
2) Move F & G to L tree to the maximum position (since F & G < X)

Splay Tree Navigation: X Found



Splay Tree Navigation: Re-assembly



Left Tree and X_L

Right Tree and X_R

After re-assembly

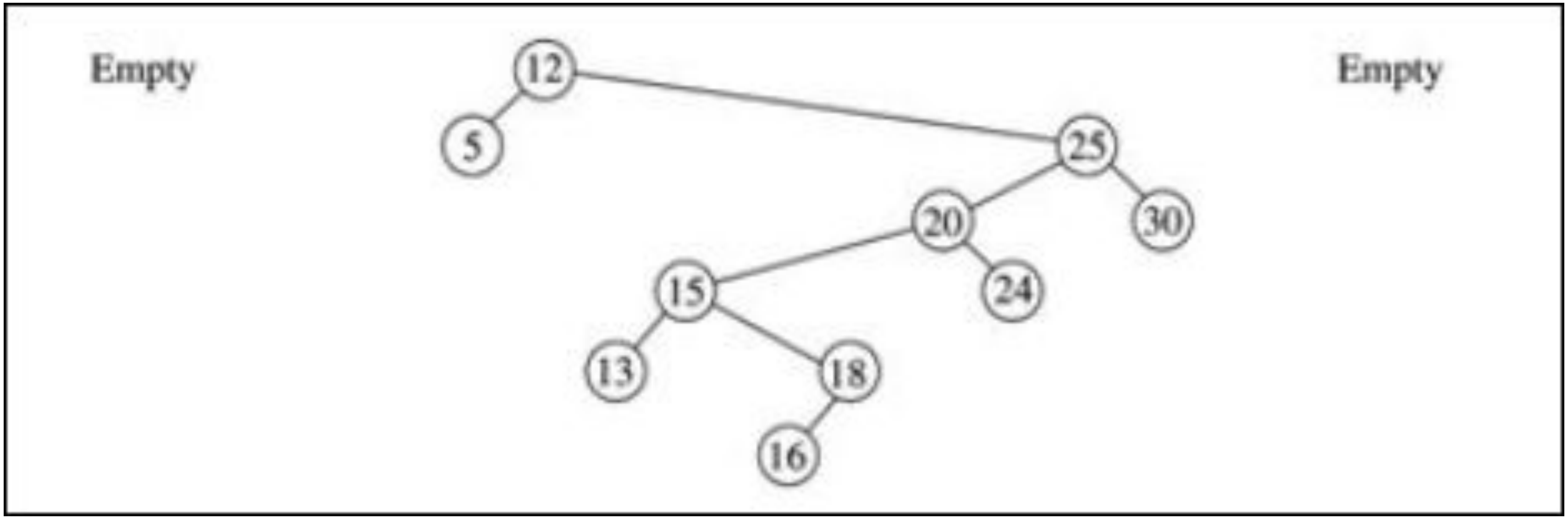
Note location of X_L and X_R
Why does that move make sense?

Discussion: Splay Tree Observations

- Is the resultant Splay Tree balanced?
- How does the structure of the resultant Splay Tree compare with how it started?
- If there is a search for X again, how long will it take? What is the Big-O in that case?

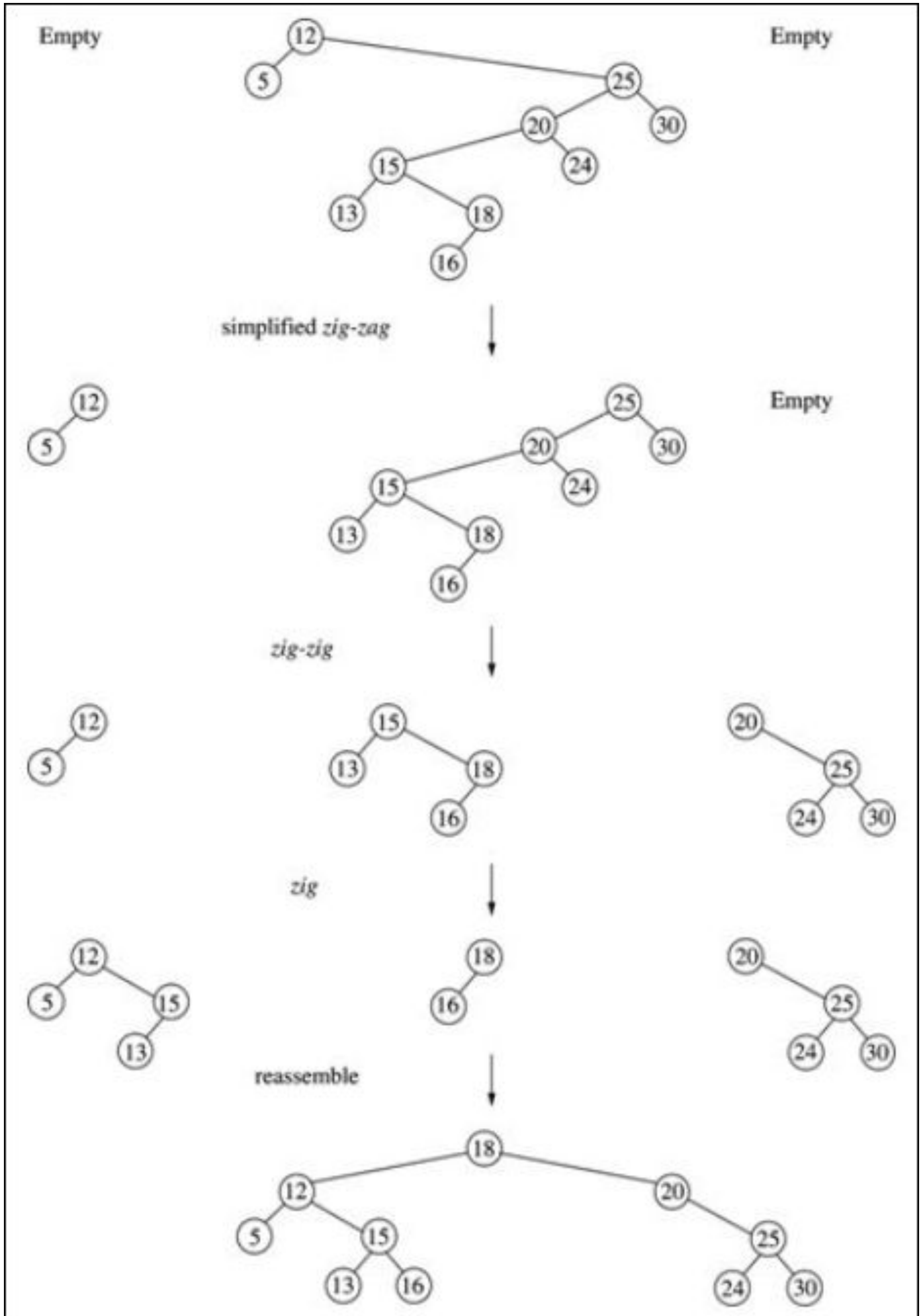
Worked Exercise

Splay for 19 in this tree



Discussion:

- Knowing that 19 is not in this tree, how useful will the resultant Splay tree be if we first searched for 19, then wanted to insert 19?
- What would be the time complexity of the insertion after the search for 19?



Solution for Splay for 19
From Weiss' text

Note: Weiss algorithm is slightly different than the modules algorithm and has a "simplified zig-zag" in addition to "zig-zag".

Homework Advice

- Follow the pseudo-code in modules closely (sections 5B.6.1 through 5B.6.3)
  - Like the coding the Subset Sum algorithm, not all details for efficient implementation are given in the pseudo-code.
  - Be careful in maintaining BST structure in the right-tree, left-tree and main-tree.
- Use the rotate methods from the modules (make the small necessary modifications)

Caution: Due to slight differences in the Module's algorithm from the text by Mark Weiss, the text's Splay tree examples may differ from your 'play computer' results.

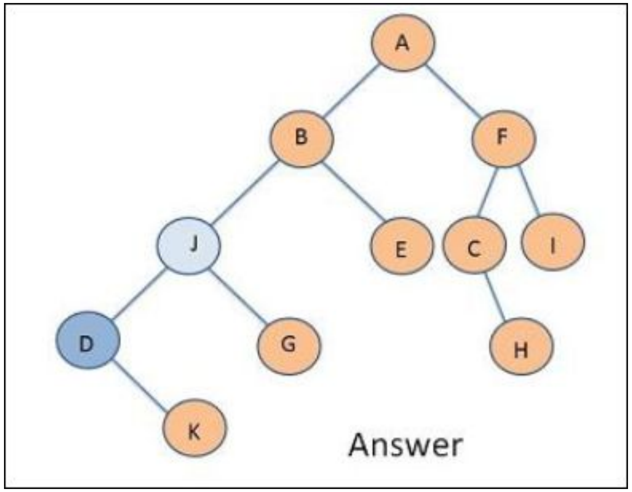
Backup

Splay Algorithm

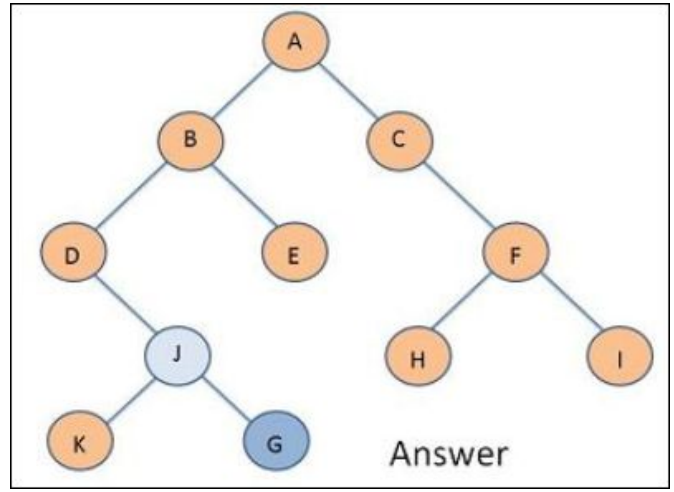
- It is an iterative (loop) process that always has three trees
  - The ***main tree*** that you are searching (which you expect contains ***x***)
  - A ***left tree,*** **L**, which contains nodes ***less than x*** that you have already examined and can be safely moved "out of the way" of the main tree, and
  - A ***right tree, R***, which contains nodes ***greater than x*** that you have already examined and can be safely moved "out of the way" of the main tree
- Initially the main tree is the entire tree, ***T***, that you are searching and **L** and **R** are empty trees.
- As the splay progresses, you will be moving left and right down the path towards **x**, much like we have done in the past in such algorithms. As you do, you will be removing nodes you encounter from the ***main tree*** and placing them into either **L** or **R**, depending on whether they are less than or greater than ***x***. Thus, the ***main tree*** will get smaller, and the **L** and **R** trees will grow.
- If you find ***x*** (or hit a **null**), you are done with the splay. You can then **re-assemble** the three trees by adjusting a few links and return.

 From Module 5B.4.3 by Michael Loceff

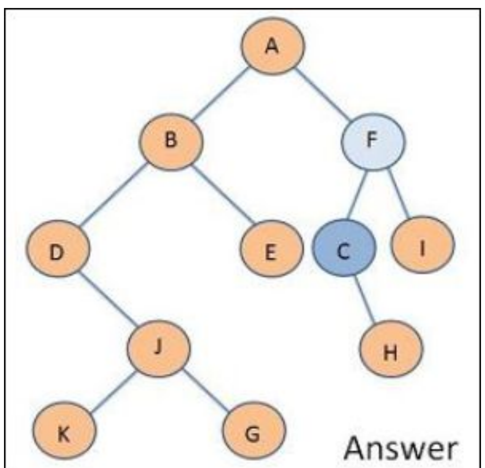
Rotation Exercise Answers



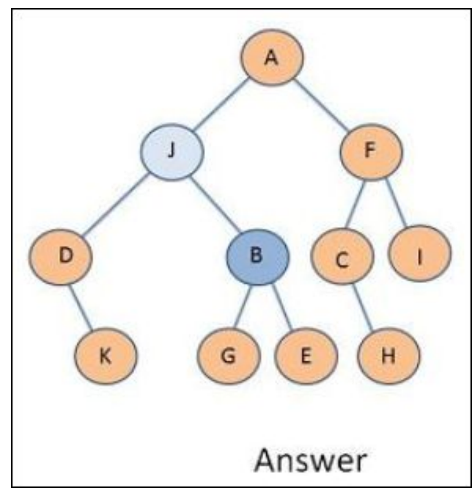
Solution to #1



Solution to #2



Solution to #3



Solution to #4

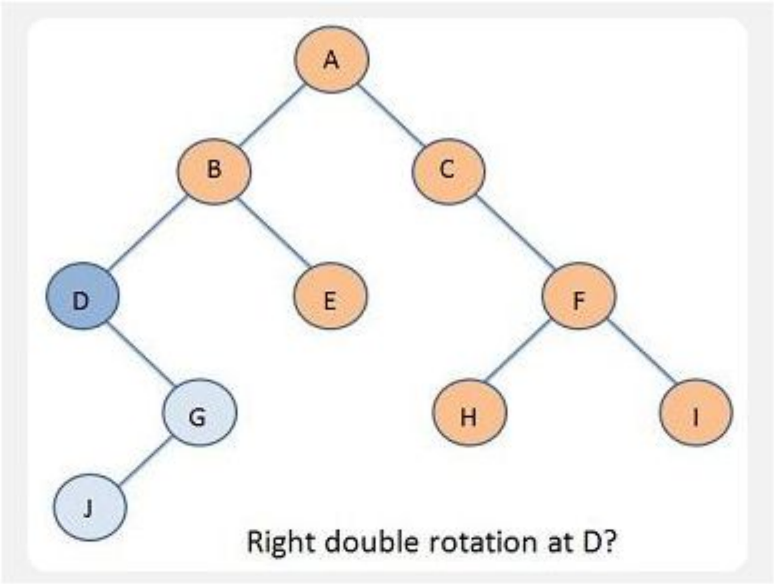
Height of NULL

```
      Zebra
       /
      /
     /
Tomato
```

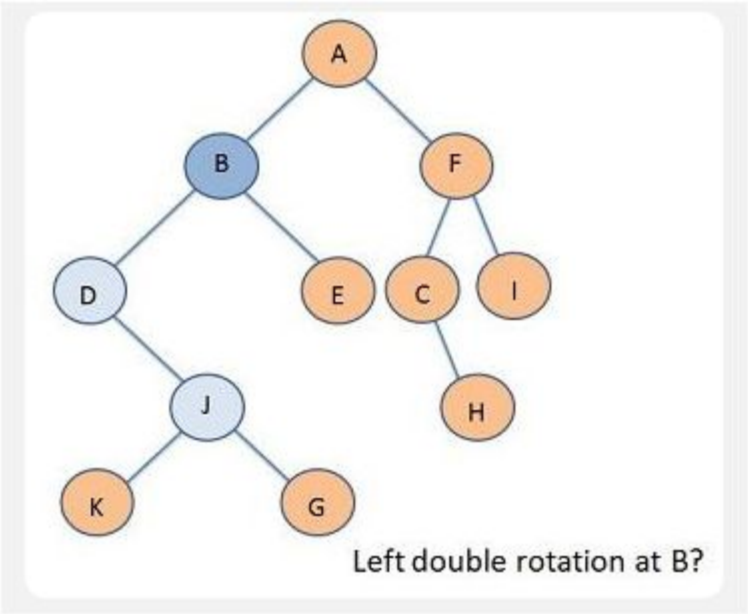
In this tree:

- Zebra's left-child subtree height is 0
- Zebra's right-child subtree height is -1
- the difference in subtree height between these two children is 1, therefore this is a balanced tree

Double Rotation

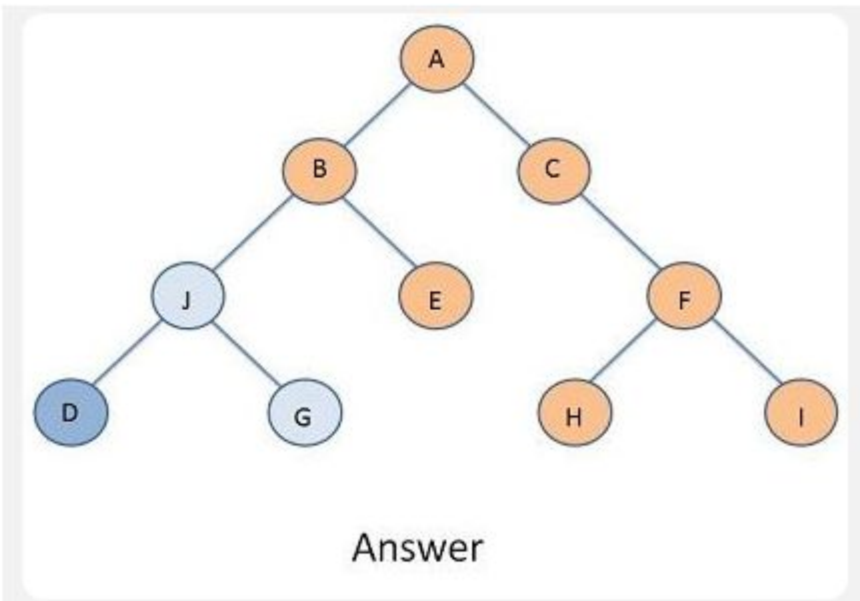


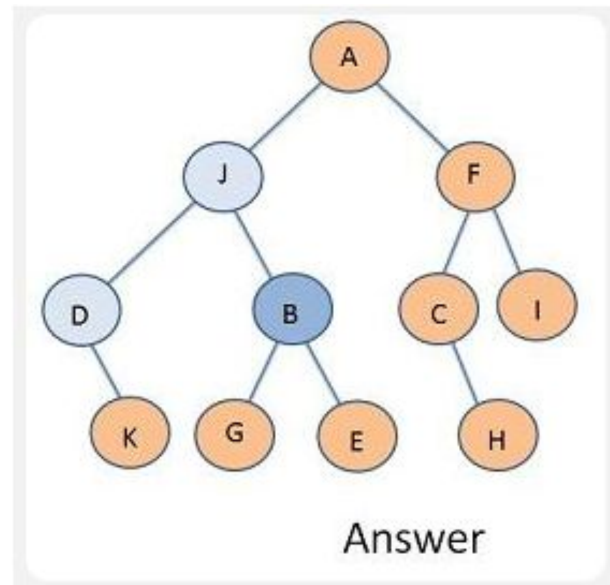
Problem 1



Problem 2

Double Rotation Solutions

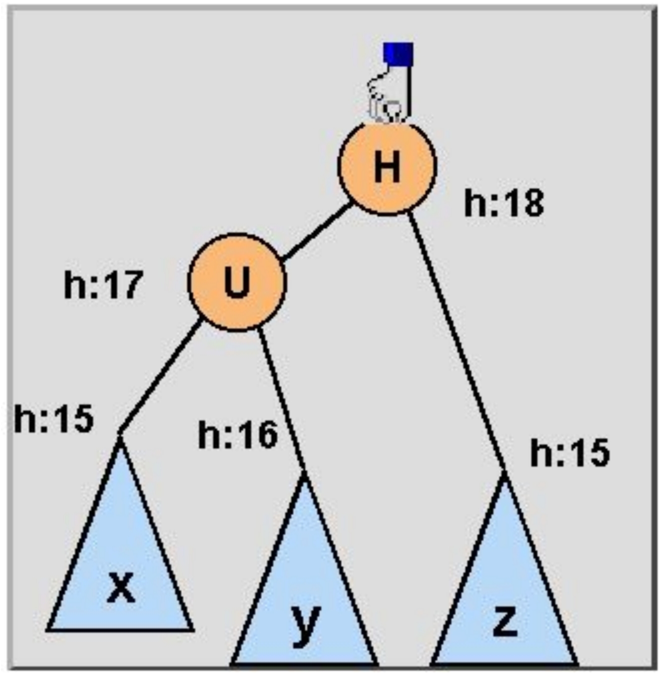




Problem 1

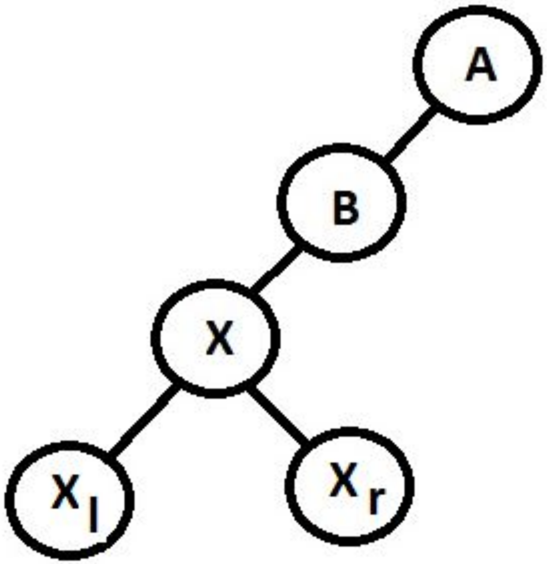
Problem 2

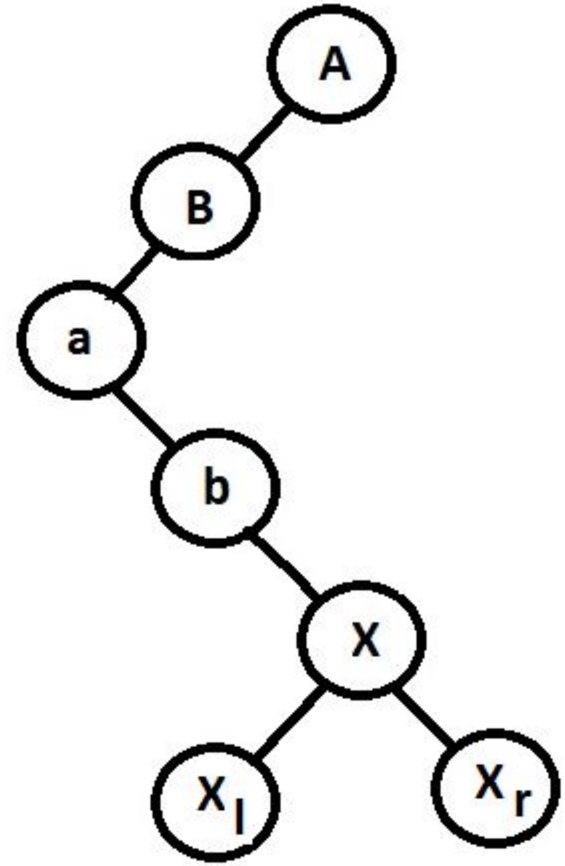
White Tree/Grey Tree Solution



Tree is the same for both trees except for “White Tree” the height of Y is 15 and the height of Z is 16 (unchanged from initial condition).

X_r Goes to Right Tree in Re-assembly

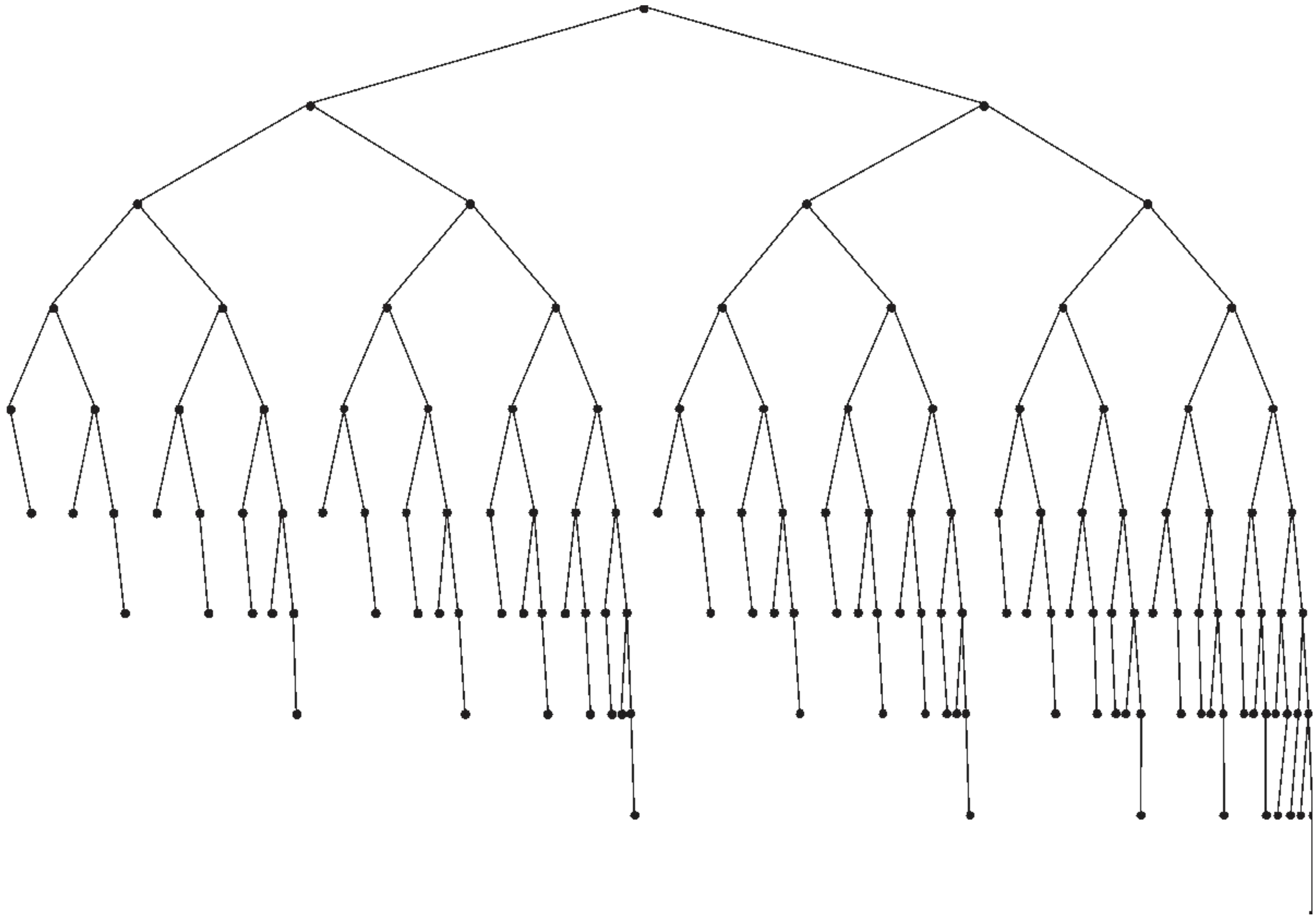




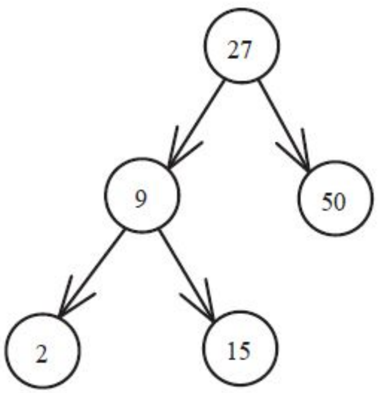
Note in both cases, X_r is greater than X and also less than A & B, therefore at re-assembly X_r goes to Left-Child of Right Tree (which has A & B above it).
A & B are above since a zig-zig left causes their rotation and move to Right Tree.

Backup & Extra

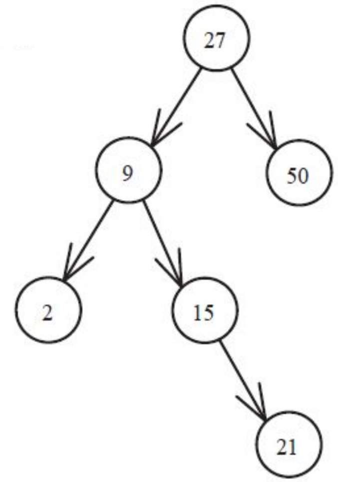
AVL Tree of Maximum Height



AVL Insert & Balance Exercise

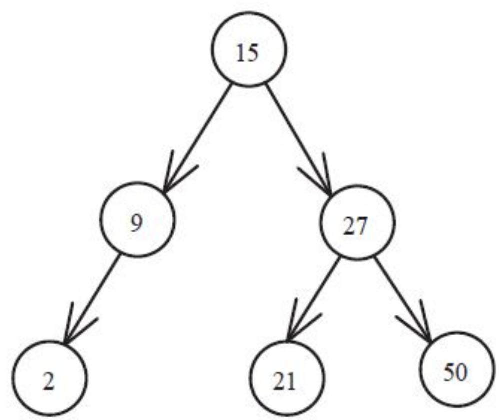


Insert 21



After insert is tree AVL misbalanced? If so, where?

What rotations needed to restore balance? Draw on whiteboard.



Images: Mikel McDaniel

Splay Pseudo-code

- initialize the **rightTree**, **leftTree**, **rightTreeMin**, and **leftTreeMax** to **NULL**
- loop while **root != NULL** (**root** should not *become* **NULL**, but this protects against **NULL** parameter)
    - if **x < root**
        - check for **root**'s left child **NULL**. If so, **x** not in tree, break loop.
        - if x < **root**'s left child we have **zig zig** (left) so do a single rotate (left) at **root**
            - check for (new) root's left child **NULL**. If so, **x** not in tree, break loop.
        - add **root** to **rightTree** at its minimum node - update the **rightTreeMin** to point to this node
        - update the new working **root**: set **root** to **root**'s left child
    - otherwise, **if x > root**
        - check for **root**'s right child **NULL**. If so, **x** not in tree, break loop.
        - if x > **root**'s right child we have **zig zig** (right) so do a single rotate (right) at **root**
            - check for (new) **root**'s right child **null**. If so, **x** not in tree, break loop.
        - add **root** to **leftTree** at its maximum node - update the **leftTreeMax** to point to this node
        - update the new working **root**: set **root** to **root**'s right child
    - otherwise we have found **x** at **root**. break.
- reassemble
    - if the left tree is not **NULL**, hang **root**'s left child onto its maximum and set **root**'s left child = the left tree.
    - if the right tree is not **NULL**, hang **root**'s right child onto its minimum and set **root**'s right child = the right tree.