Lecture 10

Christopher Godley

CSCI 2270 Data Structures

July 2nd, 2018

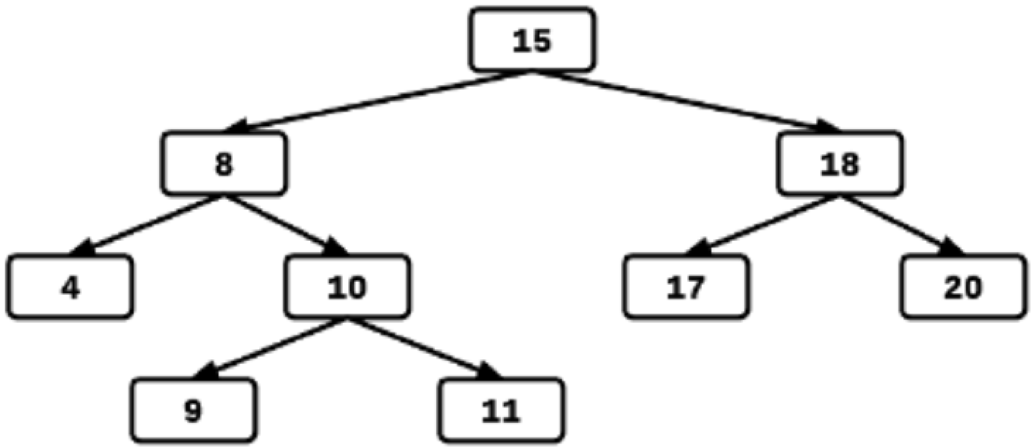
*slides from Dr. Wayne Cochran, WSUV

Binary Search Trees: Balanced Trees

- *"A balanced binary search tree has the minimum possible maximum height. For each node x, the heights of the left and right sub-trees of x differ by at most 1."* - section 9.3 in textbook
- Two parts:
  - Minimum possible maximum height
  - For each node x, the heights of its left and right sub-trees differ by at most one

Binary Search Trees: Balanced Trees

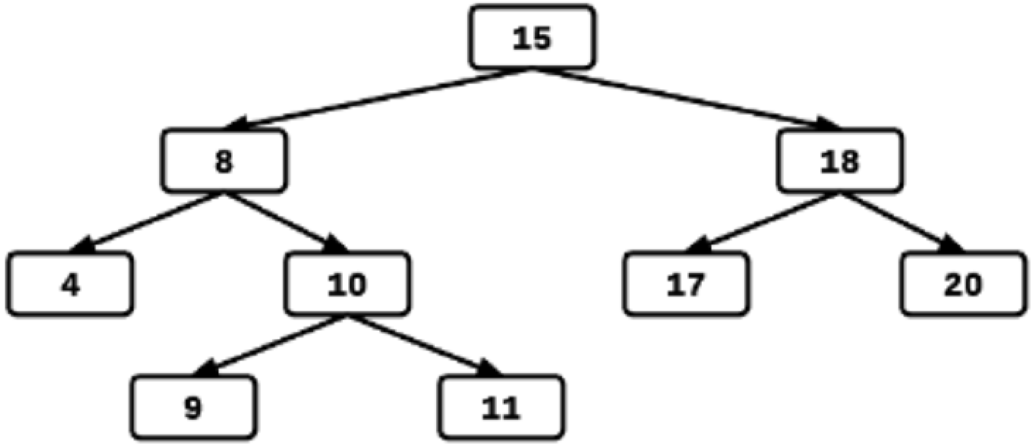
- Minimum Maximum Height
  - *h* is the height of a tree
  - Insert, delete, and search run in O(*h)* time when balanced
  - if *n = h,* insert, delete, and search run in O(*n*) time
  - The minimum maximum height should be the floor of *log_2(n)*



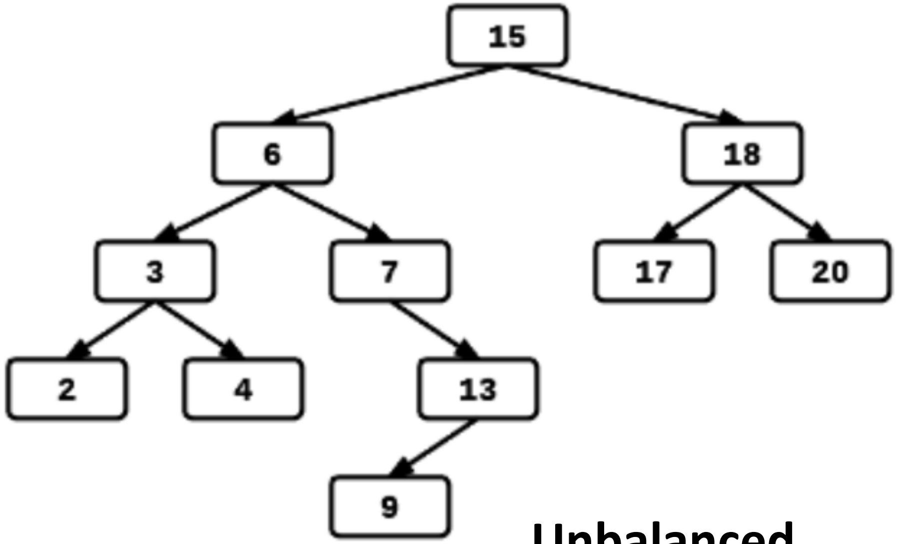
log_2(9) = 3.16
floor(3.16) = 3

Binary Search Trees: Balanced Trees

- For each node x, the heights of its left and right sub-trees differ by at most one
  - Assess the height of each left and right sub-tree at every node in the tree
  - The heights of these sub-trees must differ by at most one



Balanced



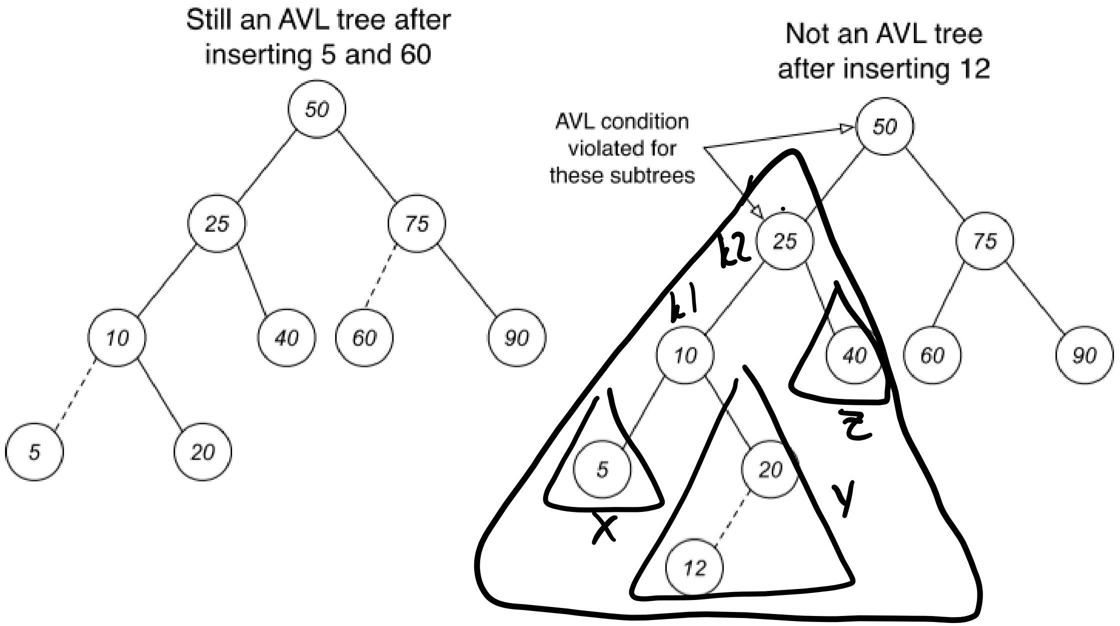
Unbalanced

AVL Trees

- An AVL (Adelson-Velki-Landis 1962) tree is a binary search tree with the following *balance condition:*
  - The height of the left and right subtrees for **every** subtree differ by at most 1
- Guarantees the depth of the tree with N nodes is O(logN)
- Height of tree with one node is 0
- Height of empty tree is -1

AVL Trees

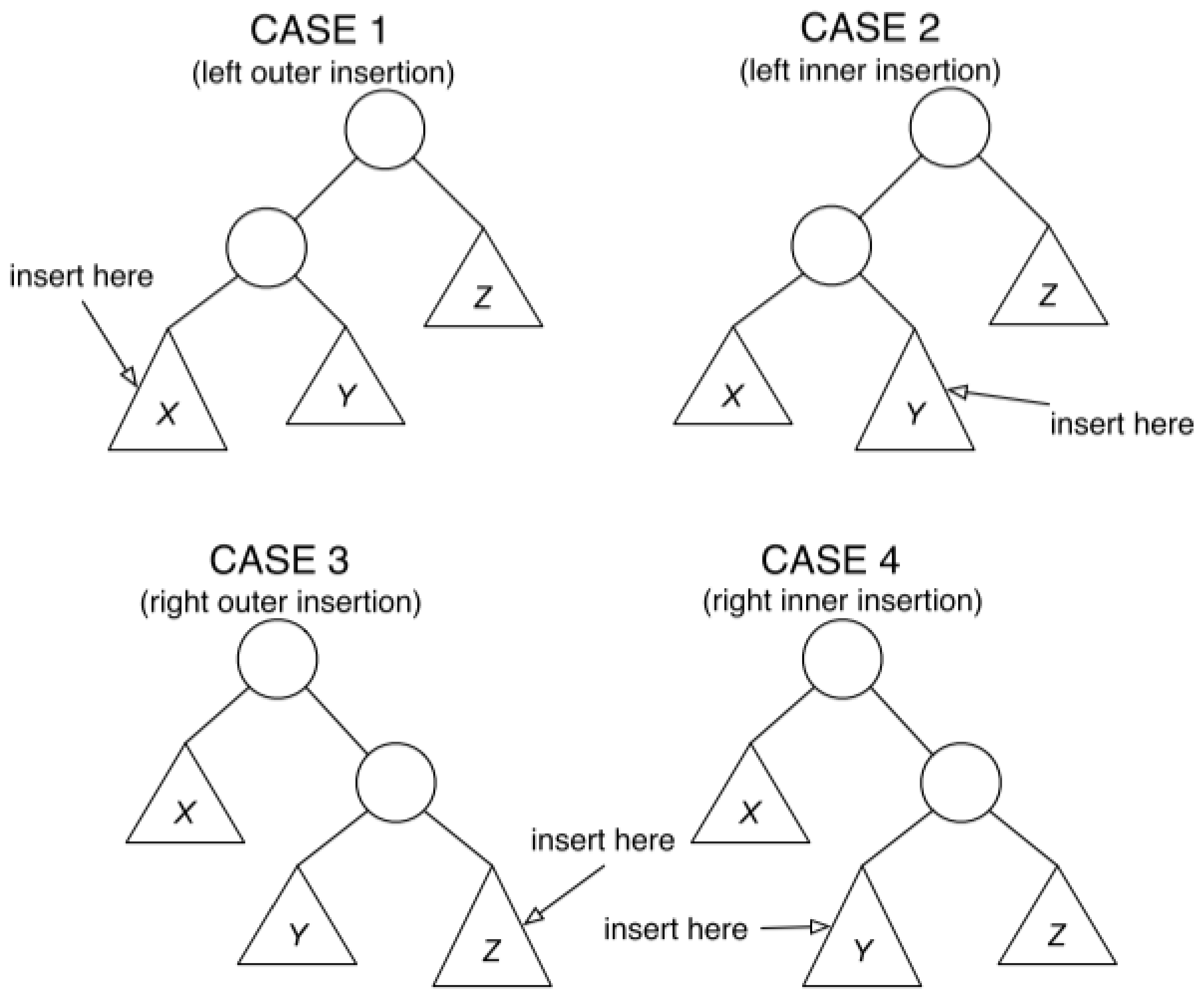
- Balance Condition:
  - Modifying an AVL tree (by inserting or deleting for example) can break the AVL balance condition
  - We need to restore this condition before completing the operation
  - The modifications we need are called **rotations**



AVL Trees: Rotations

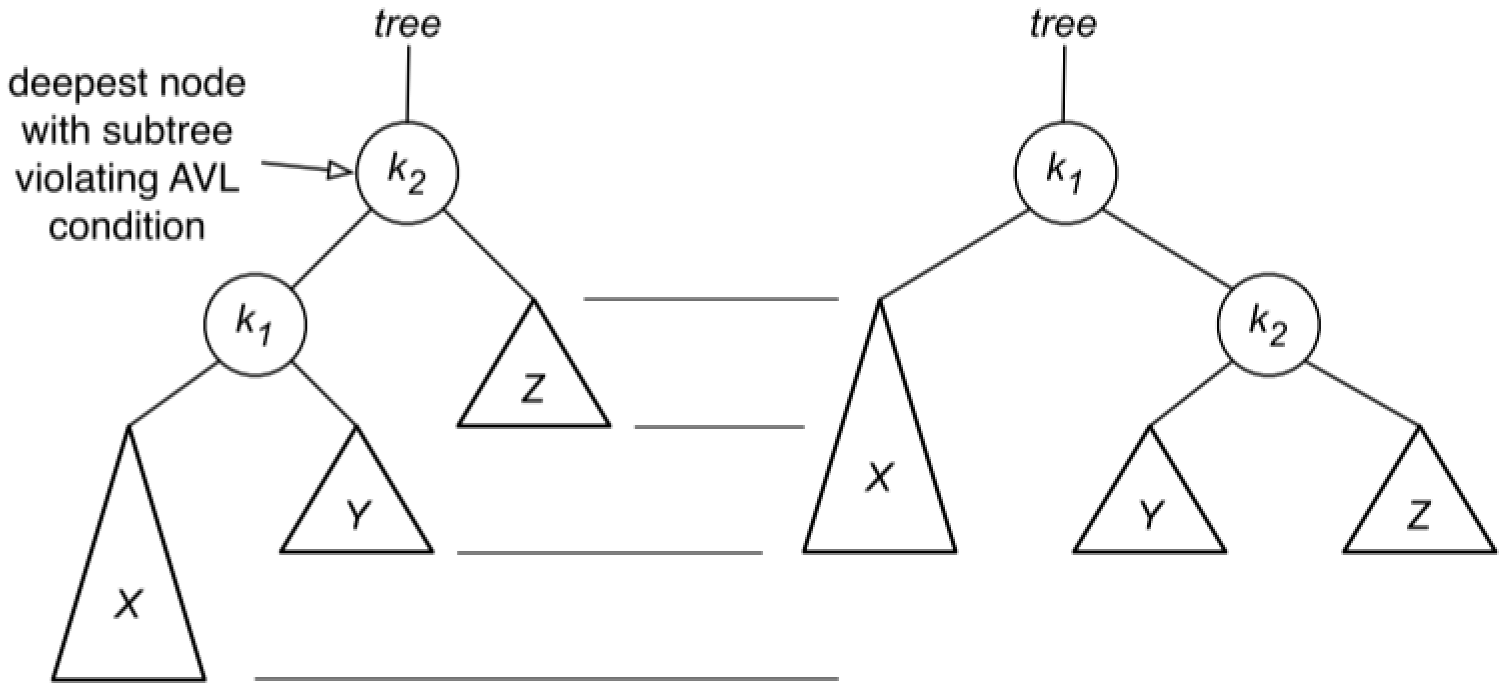
- Only four possible cases after a single insertion into a valid AVL tree
- Rebalancing the deepest violating subtree rebalances the entire tree
  - True for insertion
  - Not for delete
- Of the four cases,
  - Two “outer” cases are symmetric
  - Two “inner” cases are symmetric

AVL Trees: Rotations

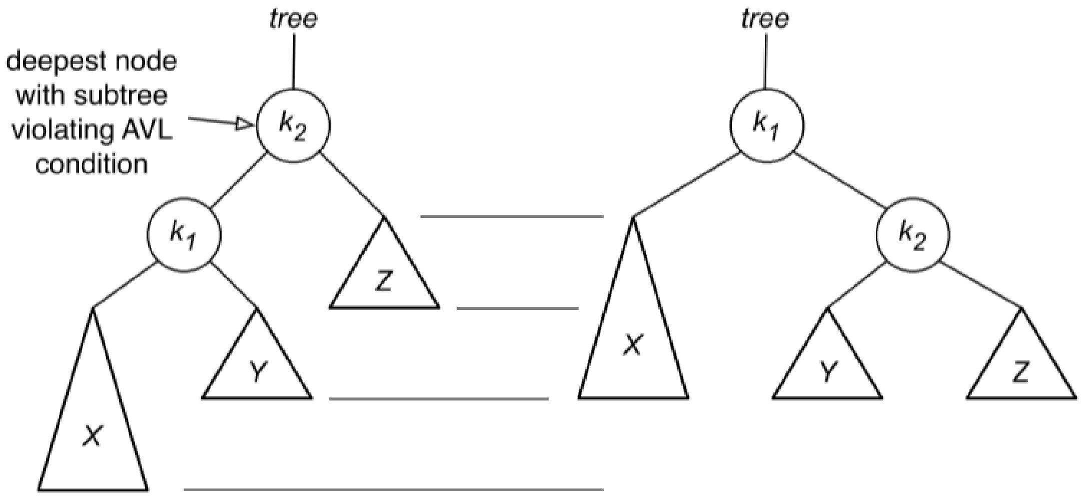


AVL Trees: Example

- Right Rotate:



AVL Trees: Example

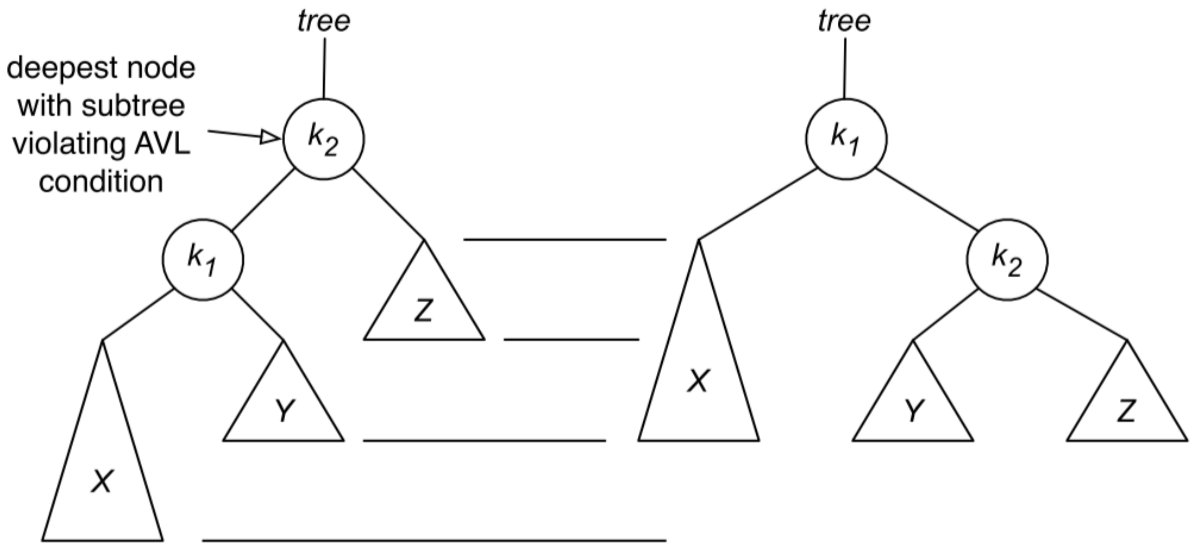


- Right Rotate:

- The tree on the left was modified, likely via inserts, so that it no longer satisfies the AVL condition at node *k2.*
  - *k2'*s left subtree is 2 deeper than its right subtree
  - The depth of the "outer" tree *X* has increased (case 1)
  - Since *k2* is the deepest node that violates the AVL condition, the subtree at *k1* does NOT violate the AVL condition, so *Y* must be 1 deeper than *Z*

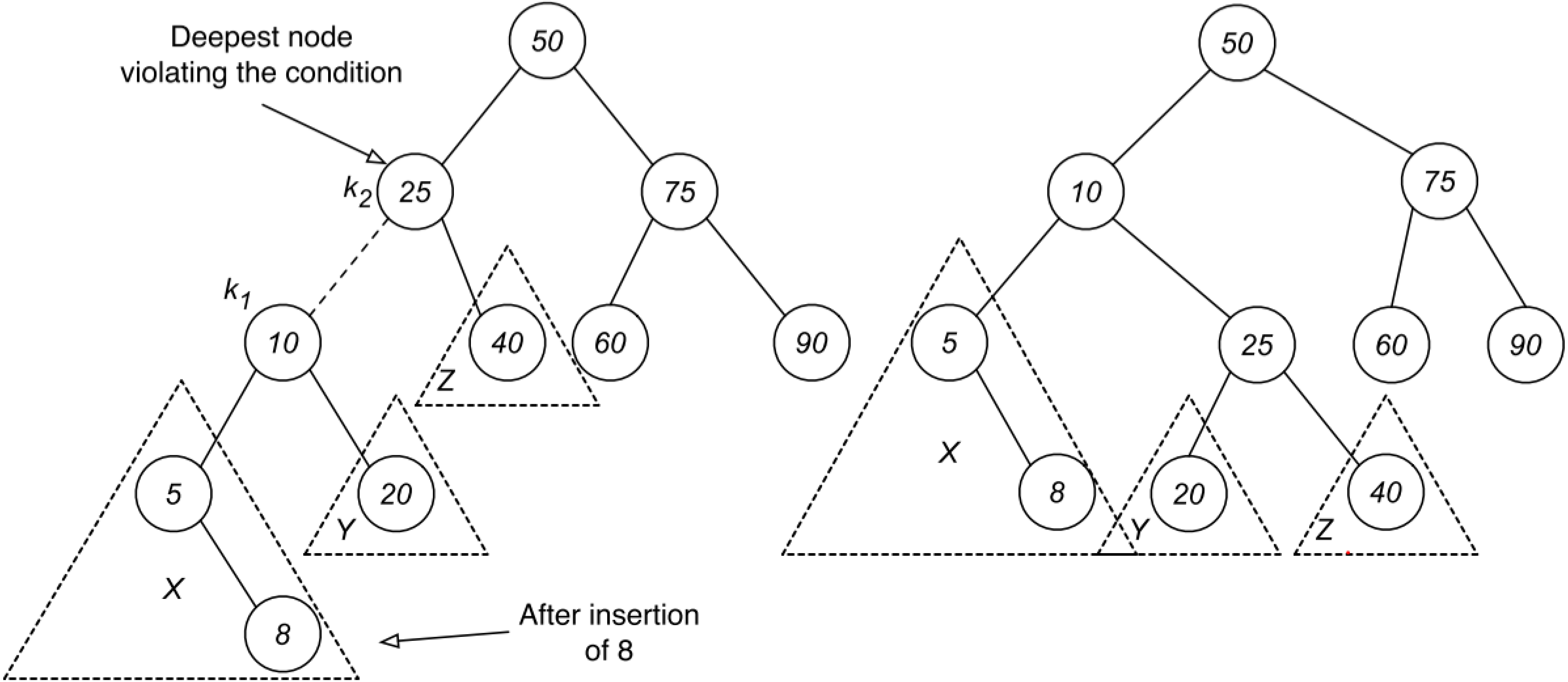
- Perform a **right rotate** at k2-k1 to obtain the repaired tree (right)

AVL Trees: Right Rotate



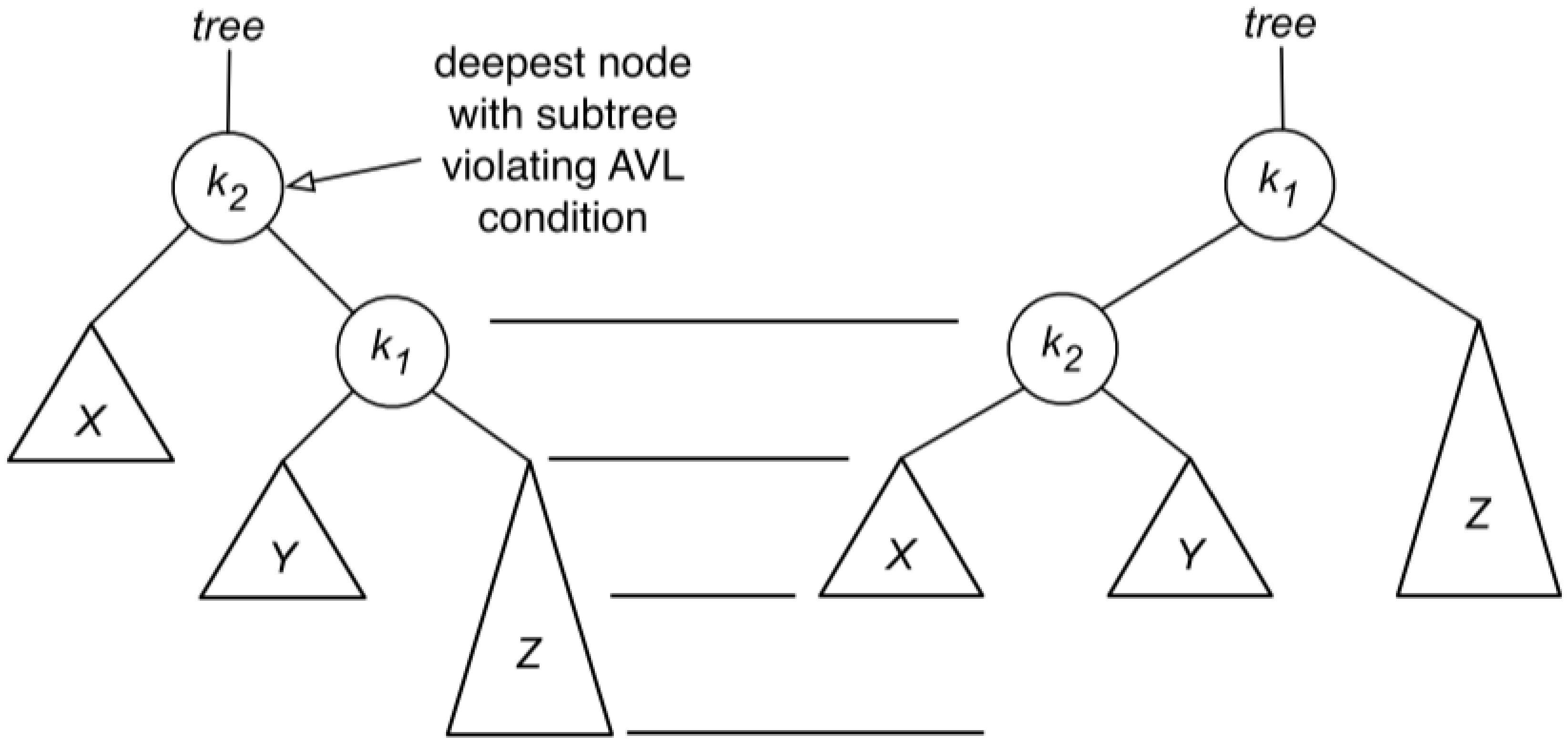

```
Y = k1.right;
k1.right = k2;
k2.left = Y;
tree = k1;
```

AVL Trees: Right Rotate

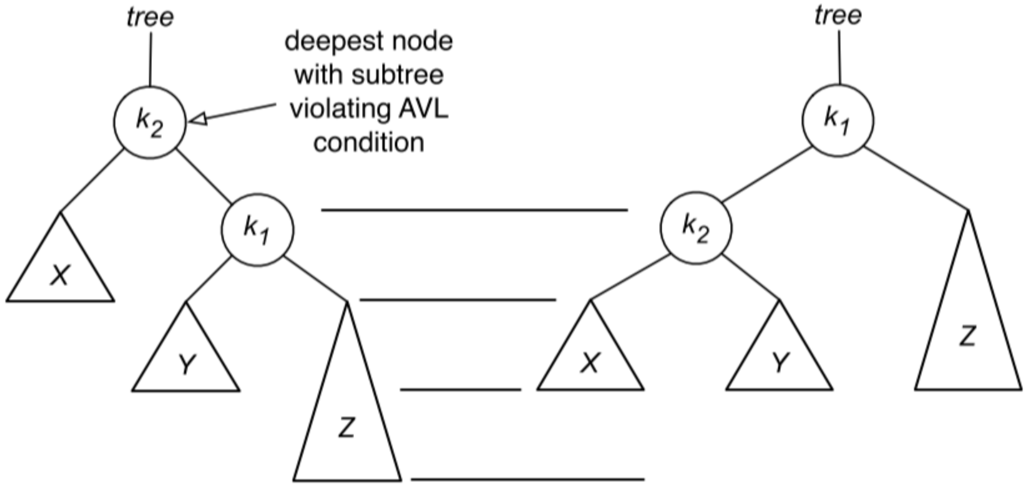


AVL Trees: Example

- Left Rotate



AVL Trees: Example

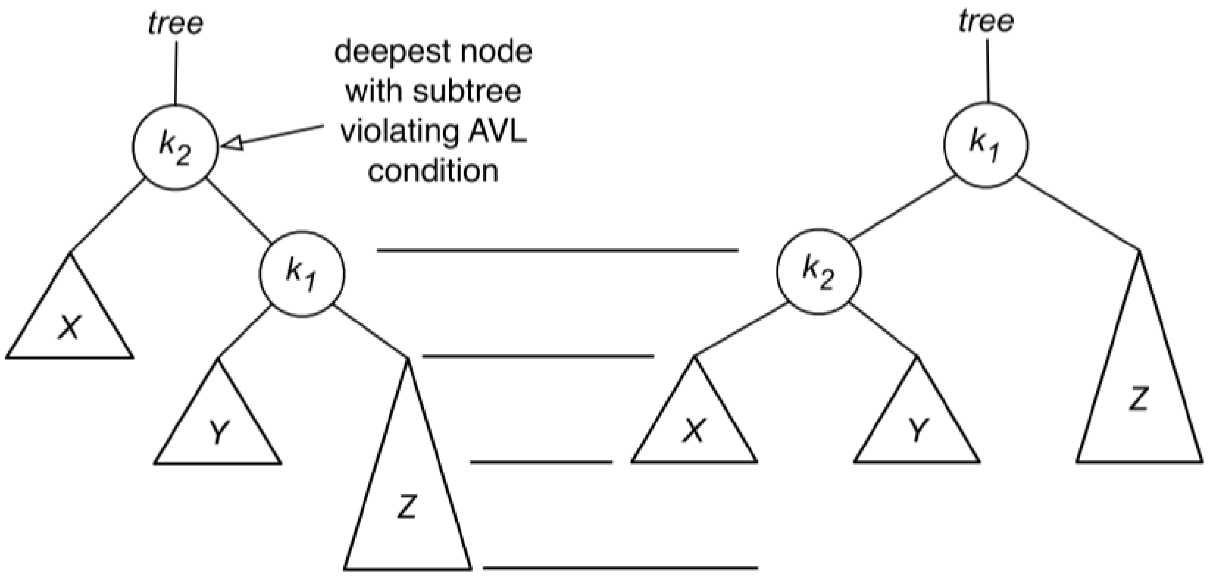


- Left Rotate

- This AVL tree (left) has been modified (insert/delete) so that it no longer satisfies the AVL condition at *k2*
  - *k2*'s right subtree is 2 deeper than its left
  - The depth of the "outer" tree Z has increased (case 4)
  - Since *k2* is the deepest node that violates the AVL condition, the subtree at *k1* does not violate the AVL condition, so Z must be 1 deeper than Y

- Perform a **left rotate** at *k2 - k1* to obtain the repaired tree (right)

AVL Trees: Left Rotate



Y = k1.**left**;

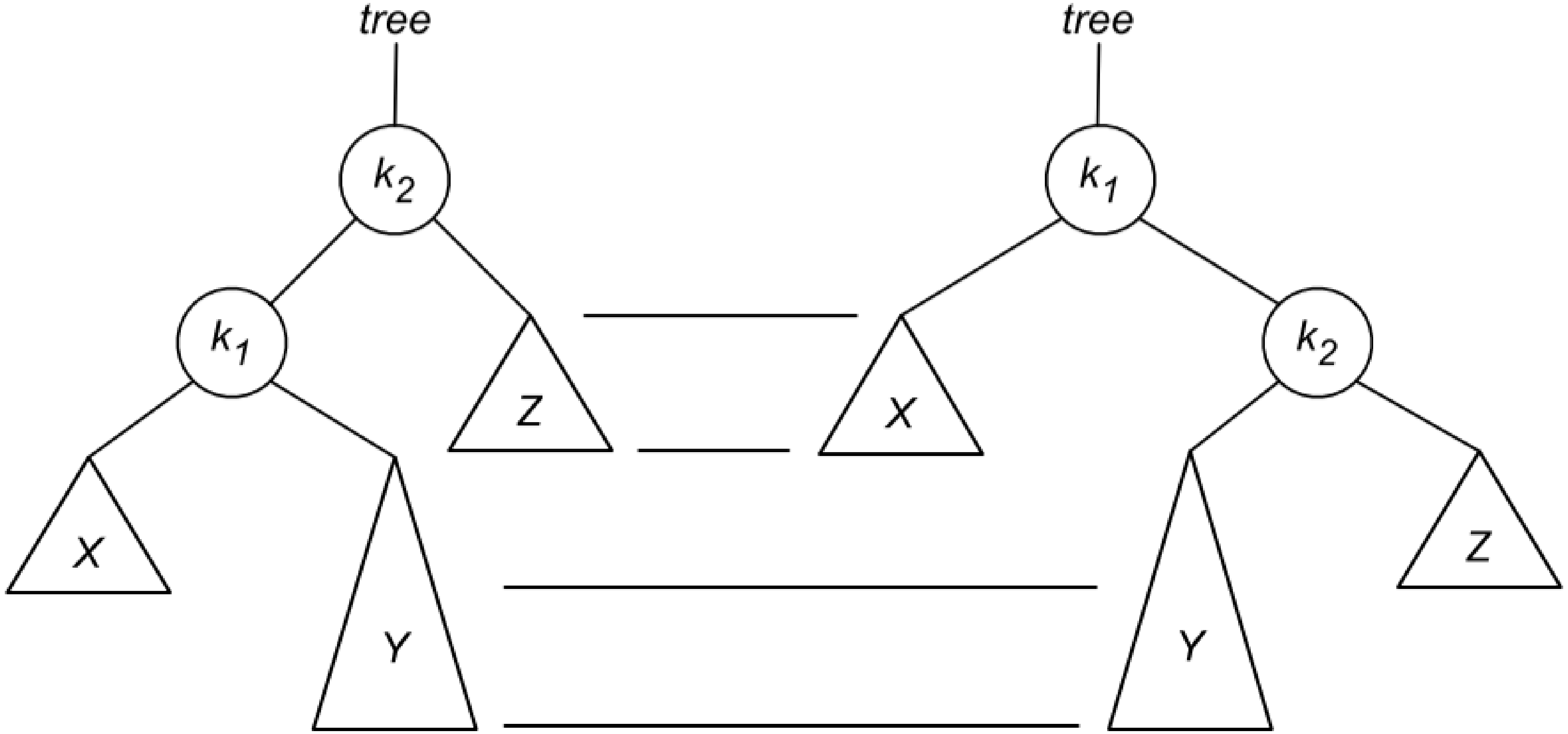
k1.**left** = k2;

k2.**right** = Y;

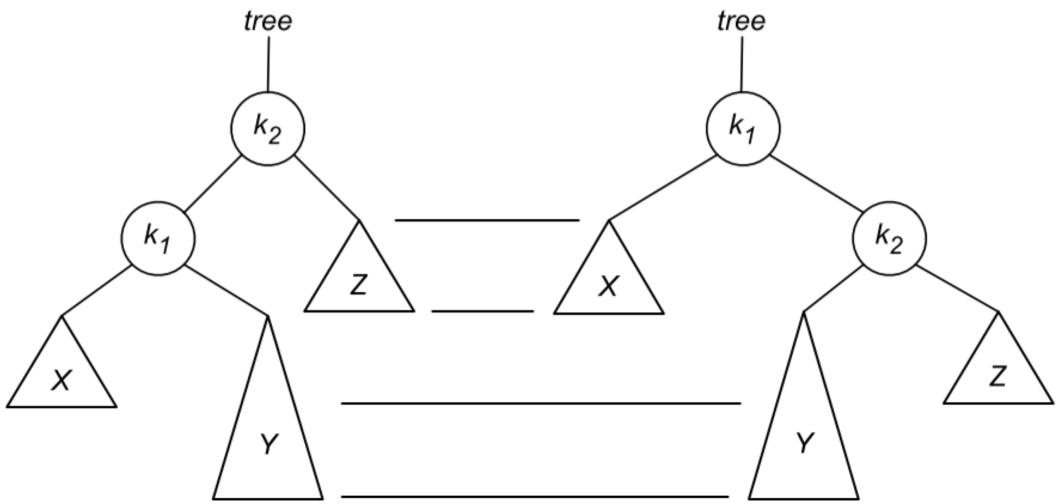
tree = k1;

AVL Trees

- Sometimes rotations fail:
  - Ex)



AVL Trees

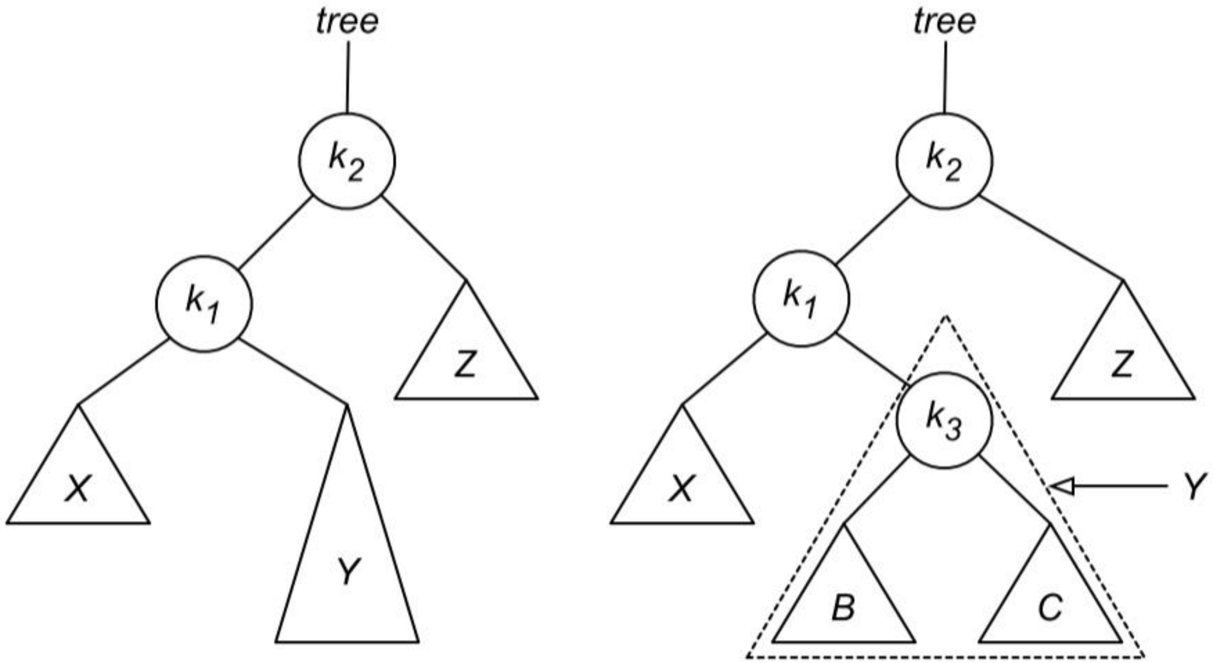


- The AVL tree (left) has been modified so that it no longer satisfies the AVL condition at *k2.*
  - *k2*'s left subtree is 2 deeper than its right
  - The depth of the "inner" tree Y has increased (case 2)
  - Since *k2* is the deepest node that violates the AVL condition, the subtree at *k1* does not violate the AVL condition, so Y must be 1 deeper than Z

- A **right rotate** at *k2 - k1* FAILS to repair the tree (right)

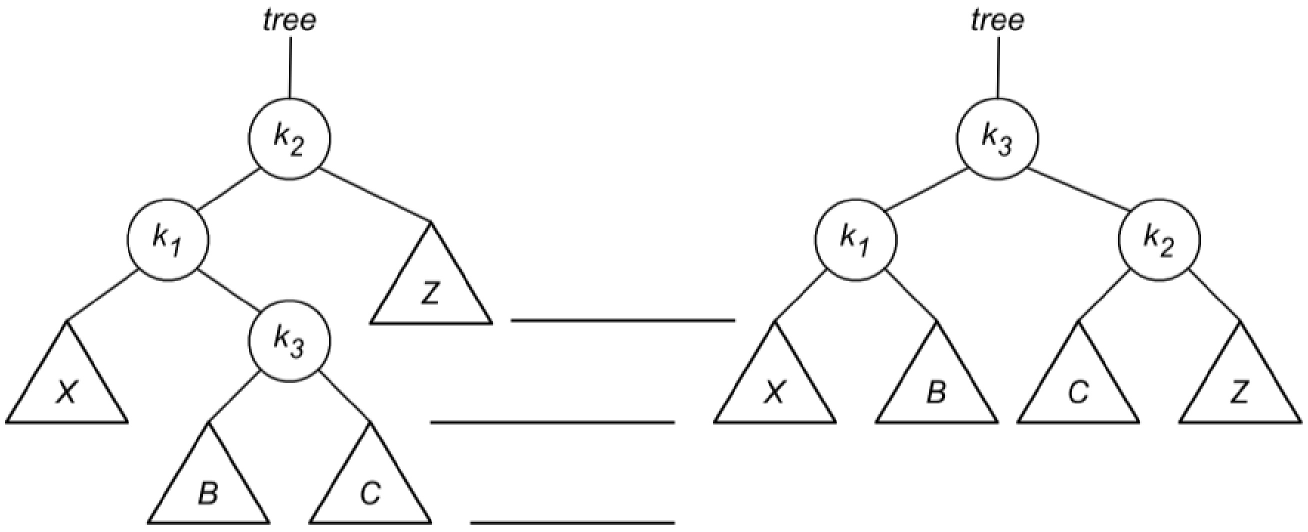
AVL Trees

- Since the subtree Y is 2 deeper than Z, it is non-empty with at least one child.
- We redraw Y with its two children (either B or C must be non-empty)
- Exactly one of the two children B or C is 2 levels deeper than Z



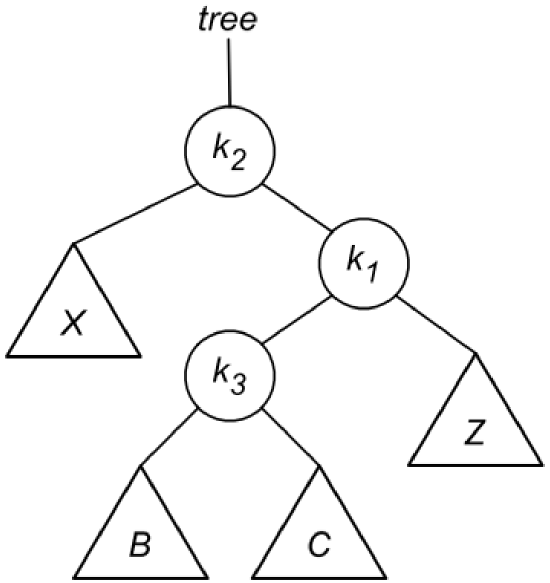
AVL Trees: Double Rotation

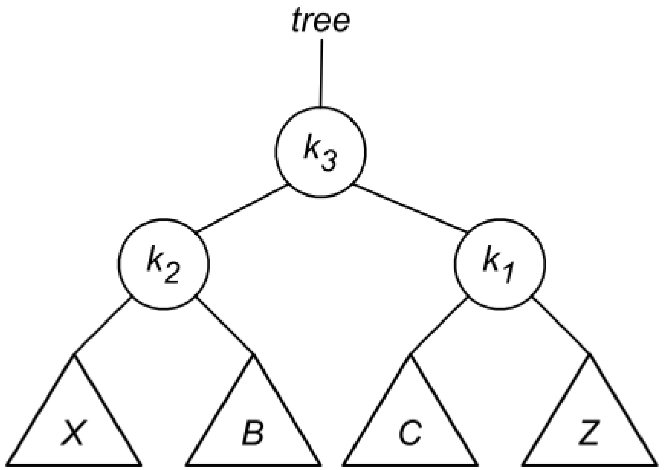
- Right Double Rotation
  - We cannot satisfy the AVL balance condition by leaving *k2* at the root nor by rotating *k1* to the root
  - We are forced to place *k3* at the root, which uniquely determines the locations of the four subtrees (right)
- Fixing the tree with a right **double rotation**
  - Left rotate between *k1 - k3*
  - Right rotate *k2 - k3*



AVL Trees: Double Rotation

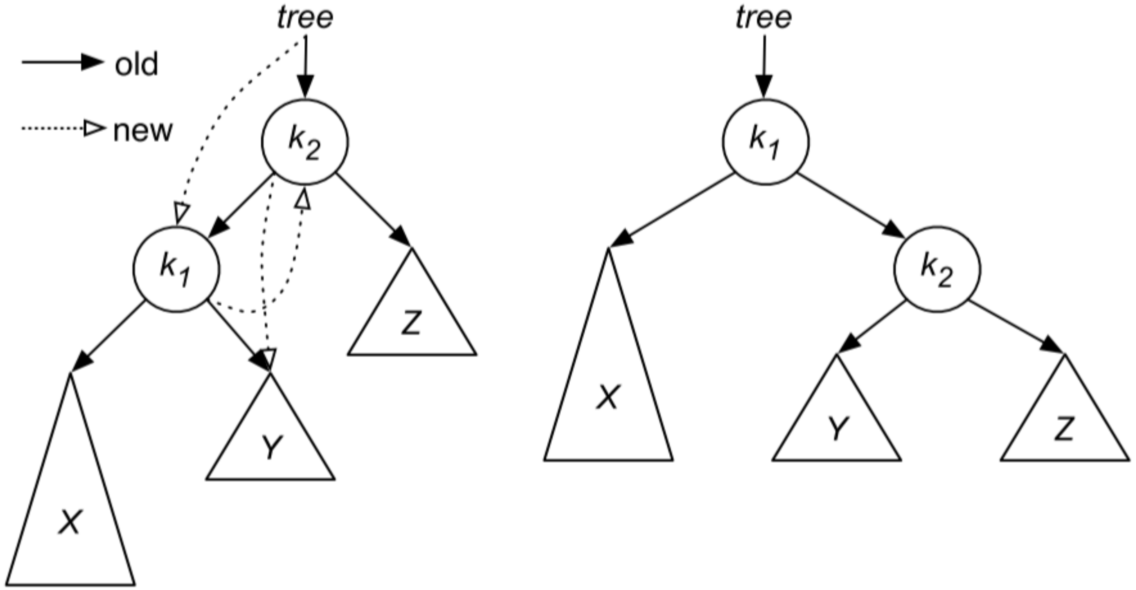
- Symmetric to previous case
- Fixing the tree with a left **double rotation**
  - Right rotate between *k1 - k3*
  - Left rotate *k2 - k3*





AVL Trees: Algorithms

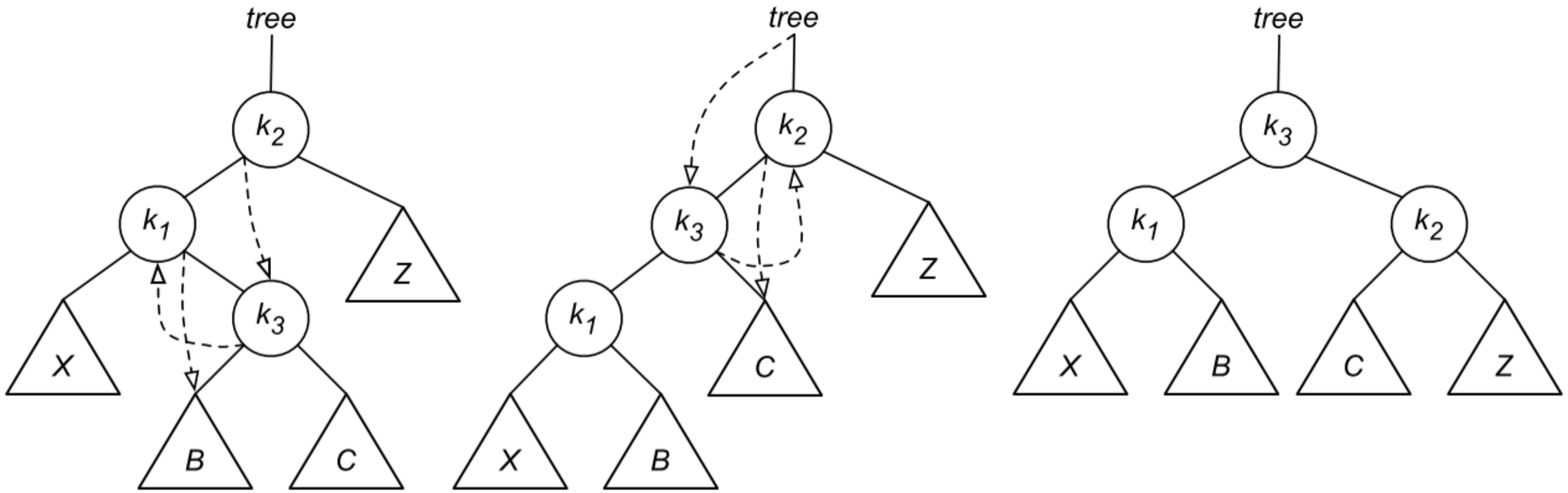
O(1)




```
AVLTree rightRotate(AVLTree tree) {
  AVLTree k2 = tree;
  AVLTree k1 = tree.left;
  k2.left = k1.right;
  k1.right = k2;
  k2.height = 1 + max(height(k2.left), height(k2.right));
  k1.height = 1 + max(height(k1.left), k2.height);
  return k1; // return new root
}
```

AVL Trees: Algorithms

O(1)




```
tree.left = leftRotate(tree.left);
tree = rightRotate(tree);
```