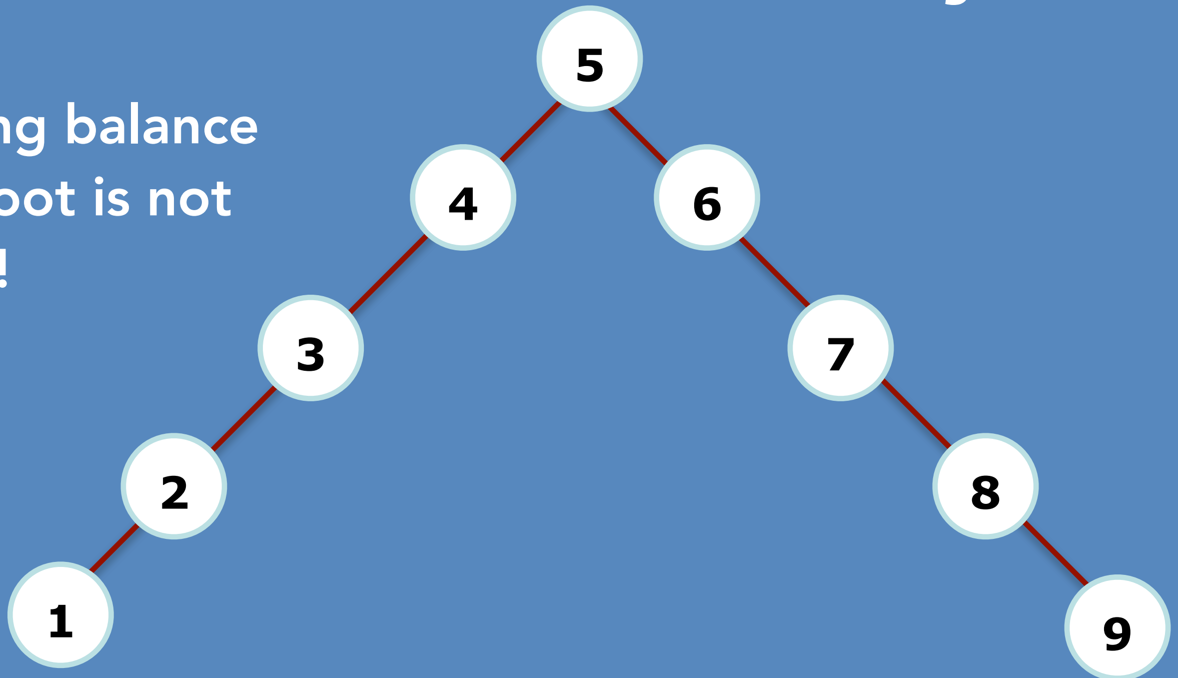
COMP 15
Data Structures
Balanced Trees — AVL Trees

Balanced Trees

- A "balanced" binary search tree is one with a condition that is (1) easy to maintain, and (2) ensures that the depth of the tree is on the order of log ***N***.
- A simple idea is to require that the left and right subtrees have the same height…

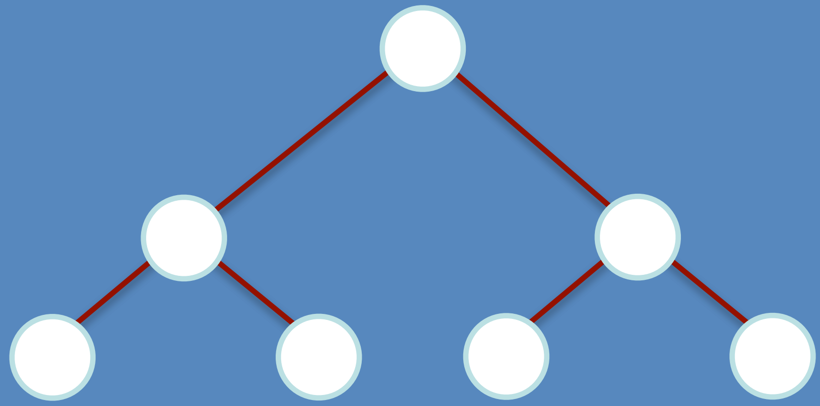
Balanced Trees: a bad binary tree

Requiring balance at the root is not enough!



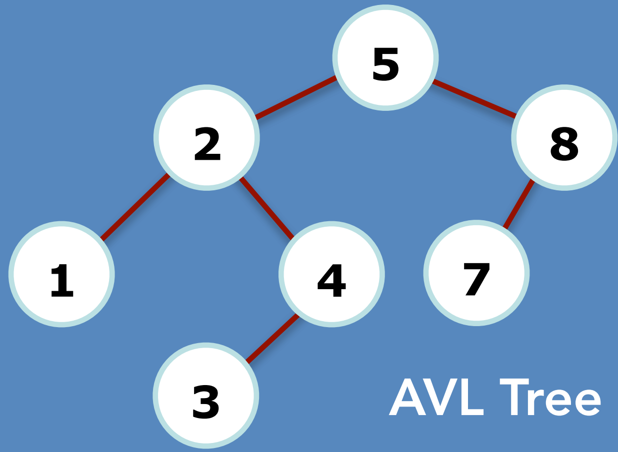
Balanced Trees

- Another balance condition could be to insist that every node must have left and right subtrees of the same height: too rigid to be useful: only perfectly balanced trees with 2^k-1 nodes would satisfy the condition (even with the guarantee of small depth).

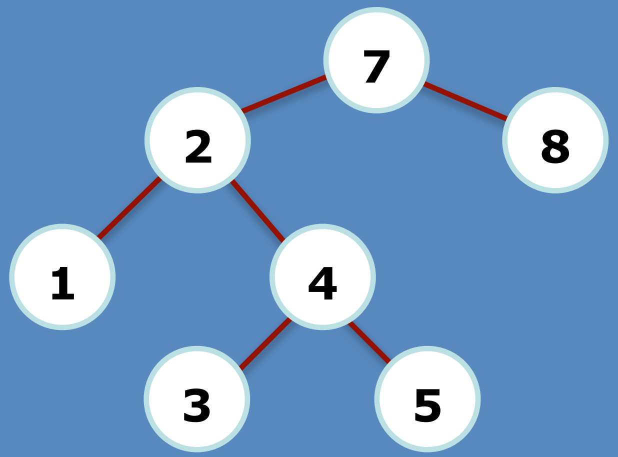


AVL Trees

- An ***AVL*** tree (Adelson-Velskii and Landis) is a compromise. It is the same as a binary search tree, but with ***added invariant:*** for every node, the height of the left and right subtrees can differ only by 1 (and an empty tree has a height of -1).



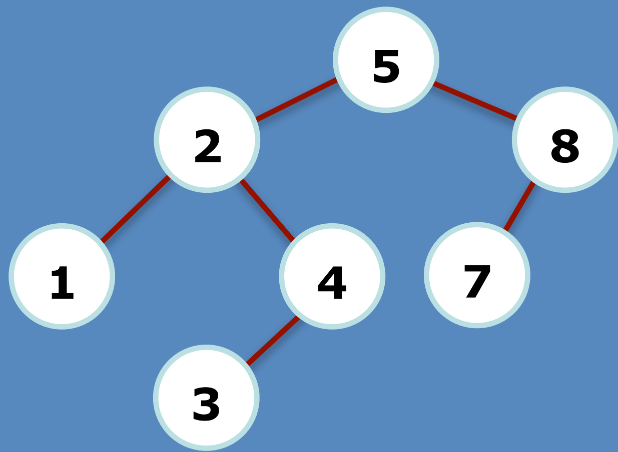
AVL Tree



Not an AVL Tree

AVL Trees

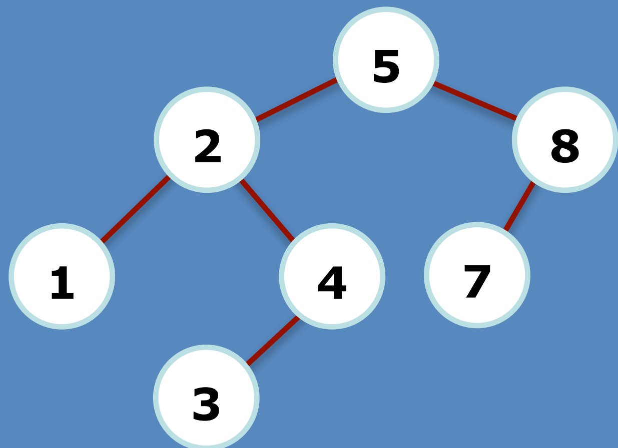
- Height information is kept for each node, and the height is *almost* log N in practice.



- When we insert into an AVL tree, we have to update the balancing information back up the tree
- We also have to maintain the AVL property — tricky! Think about inserting 6 into the tree: this would upset the balance at node 8.

AVL Trees

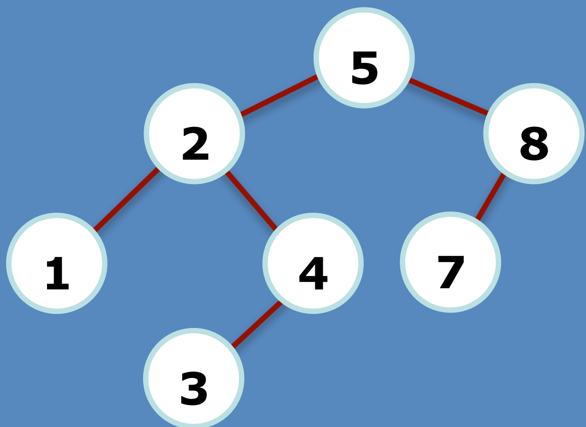
- As it turns out, a simple modification of the tree, called ***rotation***, can restore the AVL property.



- After insertion, only nodes on the path from the insertion might have their balance altered, because only those nodes had their subtrees altered.
- We will re-balance as we follow the path up to the root updating balancing information.

AVL Trees

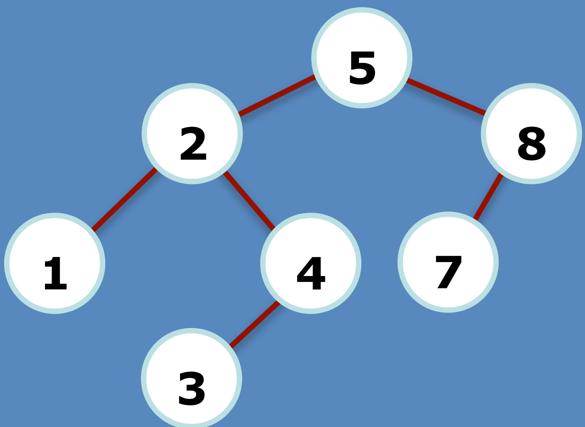
- We will call the node to be balanced, α



- Because any node has at most two children, and a height imbalance requires that α's two subtrees' heights differ by two, there can be four violation cases:

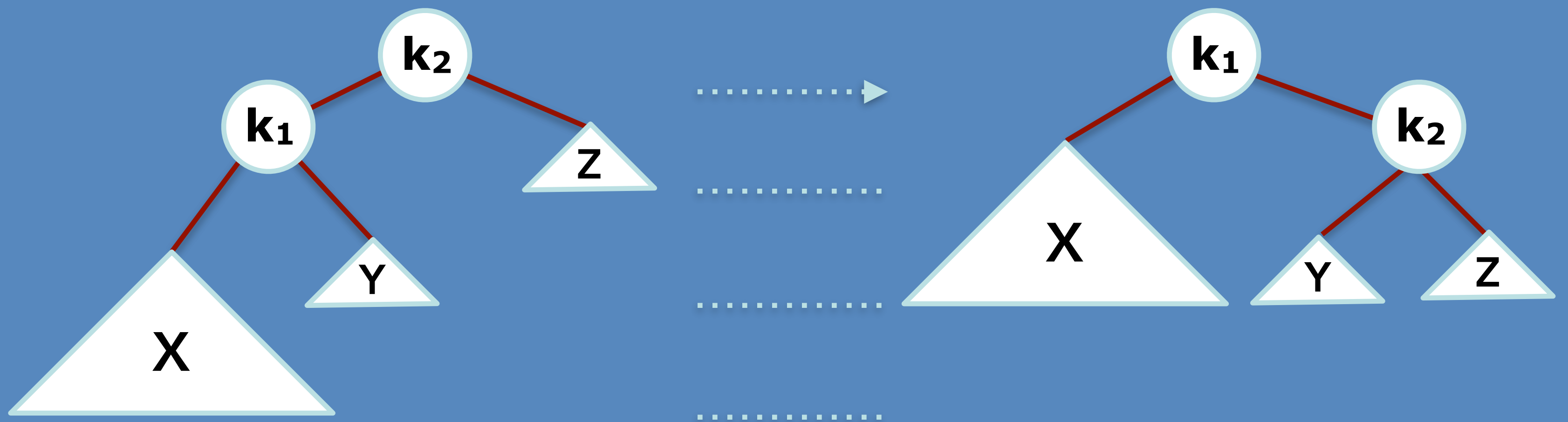
1. An insertion into the left subtree of the left child of α.
2. An insertion into the right subtree of the left child of α.
3. An insertion into the left subtree of the right child of α.
4. An insertion into the right subtree of the right child of α.

AVL Trees: rotations



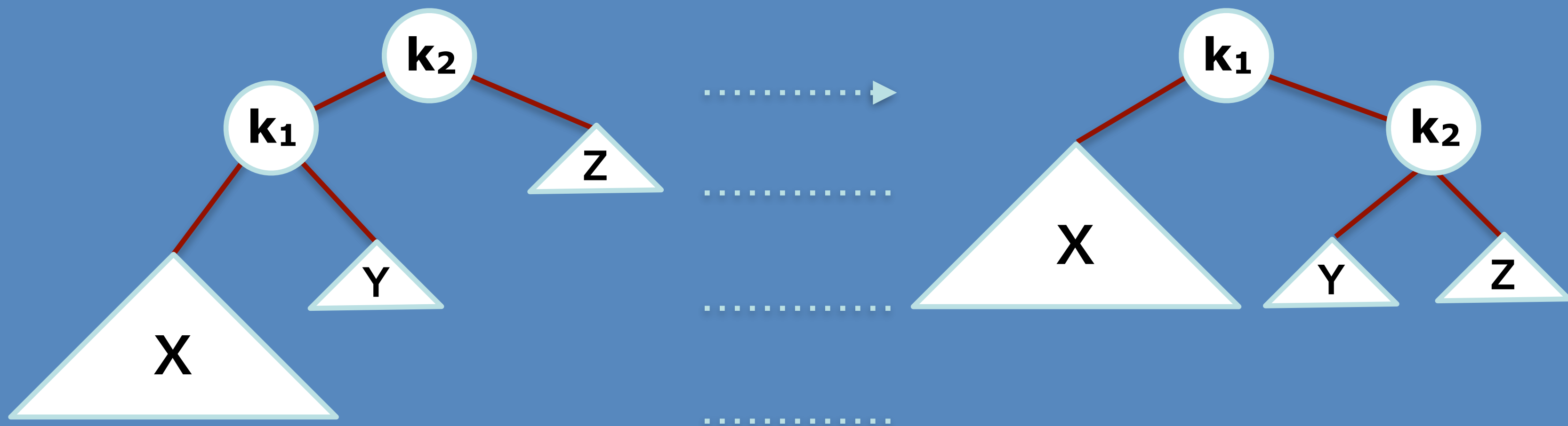
- For "outside" cases (left-left, right-right), we can do a "single rotation"
- For "inside" cases (left-right, right, left), we have to do a more complex "double rotation."

AVL Trees: Single Rotation



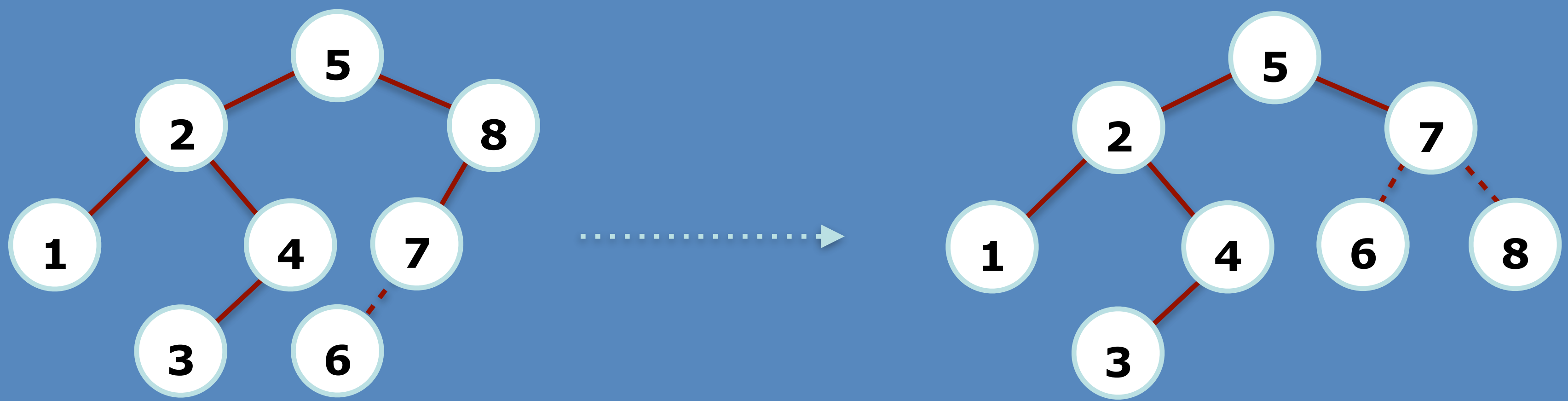
k_2 violates the AVL property, as X has grown to be 2 levels deeper than Z. Y cannot be at the same level as X because k_2 would have been out of balance before the insertion. We would like to move X up a level and Z down a level (fine, but not strictly necessary).

AVL Trees: Single Rotation



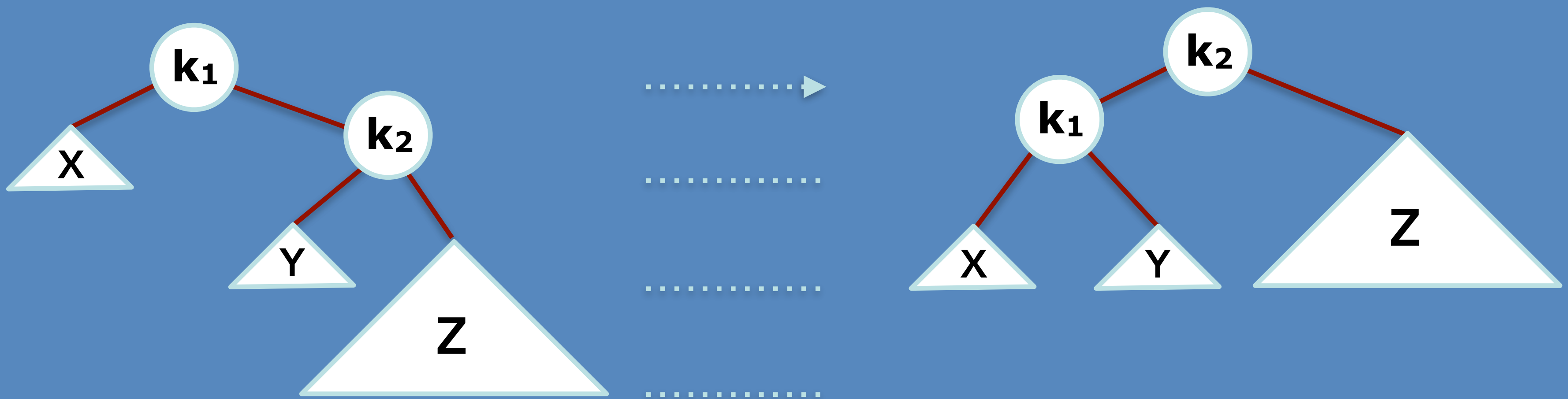
Visualization: Grab k_1 and shake, letting gravity take hold. k_1 is now the new root. In the original, $k_2 > k_1$, so k_2 becomes the right child of k_1. X and Z remain as the left and right children of k_1 and k_2, respectively. Y can be placed as k_2's left child and satisfies all ordering requirements.

AVL Trees: Single Rotation



Insertion of 6 breaks AVL property at 8 (not 5!), but is fixed with a single rotation (we "rotate 8 right" by grabbing 7 and hoisting it up)

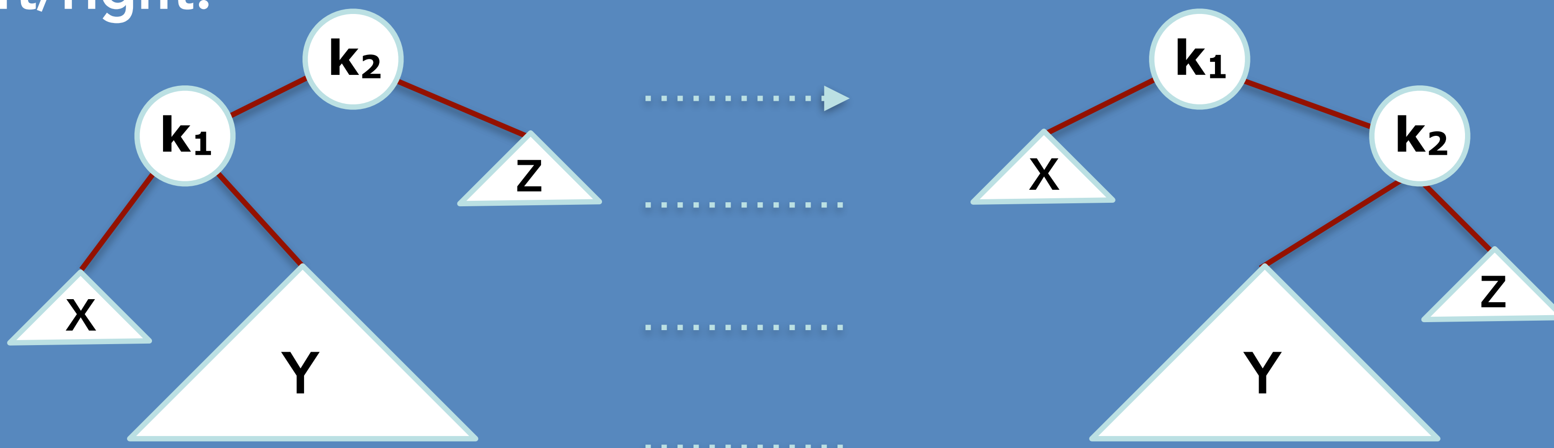
AVL Trees: Single Rotation



It is a symmetric case for the right-subtree of the right child. k_1 is unbalanced, so we "rotate k1 left" by hoisting k2) http://www.cs.usfca.edu/~galles/visualization/AVLtree.html

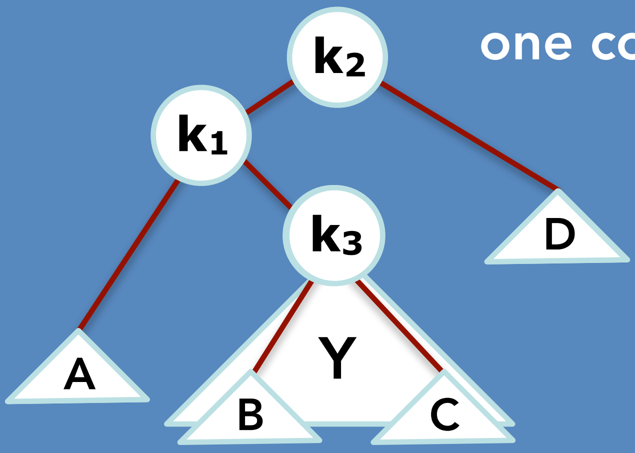
Insert 3, 2, 1, 4, 5, 6, 7

AVL Trees: Single Rotation doesn't work for right/left, left/right!



Subtree Y is too deep (unbalanced at k_2), and the single rotation does not make it any less deep.

AVL Trees: Double Rotation (can be thought of as one complex rotation or two simple single rotations)

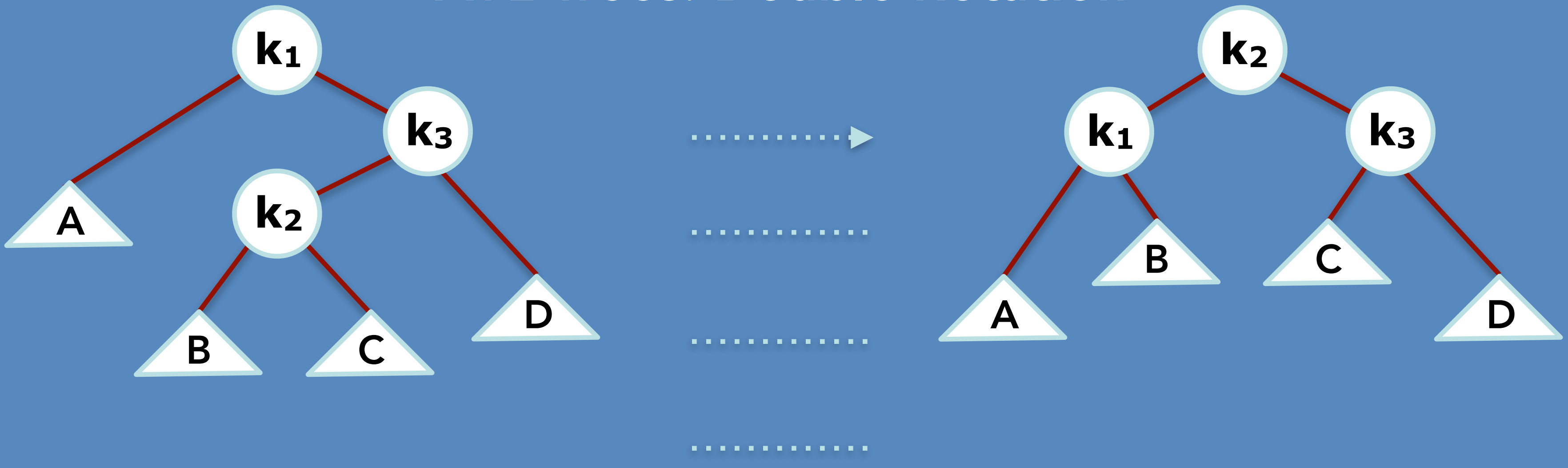


Instead of three subtrees, we can view the tree as four subtrees, connected by three nodes.

AVL Trees: Double Rotation

We can't leave k_2 as root, nor can we make k_1 root (as shown before). So, k_3 must become the root.

AVL Trees: Double Rotation



Double rotation also fixes an insertion into the left subtree of the right child (k_1 is unbalanced, so we first rotate k_3 right, then we rotate k_1 left) http://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Before: Insert 3, 2, 1, 4, 5, 6, 7

Continuing: Insert 16, 15, 14, 13, 12, 11, 10, 8, 9

AVL Trees: How to Code

- Coding up AVL tree rotation is straightforward, but can be tricky.
- A recursive solution is easiest, but not too fast. However, clarity generally wins out in this case.
- To insert a new node into an AVL tree:
  1. Follow normal BST insertion.
  2. If the height of a subtree does not change, stop.
  3. If the height does change, do an appropriate single or double rotation, and update heights up the tree.
  4. One rotation will always suffice.

- Example code can be found here: http://www.sanfoundry.com/cpp-program-implement-avl-trees/

Other Balanced Tree Data Structures

- 2-3 tree
- AA tree
- AVL tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap

Advanced Reading

Wikipedia article on self-balancing trees (be sure to look at all the implementations):
http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

Red Black Trees:
https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html

YouTube AVL Trees: http://www.youtube.com/watch?v=YKt1kquKScY

References

Wikipedia article on AVL Trees: http://en.wikipedia.org/wiki/AVL_tree

Really amazing lecture on AVL Trees: https://www.youtube.com/watch?v=FNeL18KsWPc

Assigned reading:
See Mon, 2/23/2015: http://www.cs.tufts.edu/comp/15/schedule/