CSE 373

AVL trees, continued

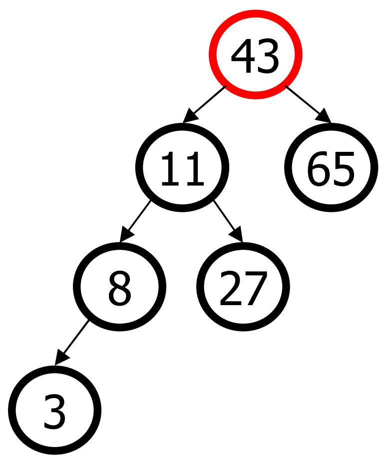
read: Weiss Ch. 4, section 4.1 - 4.4

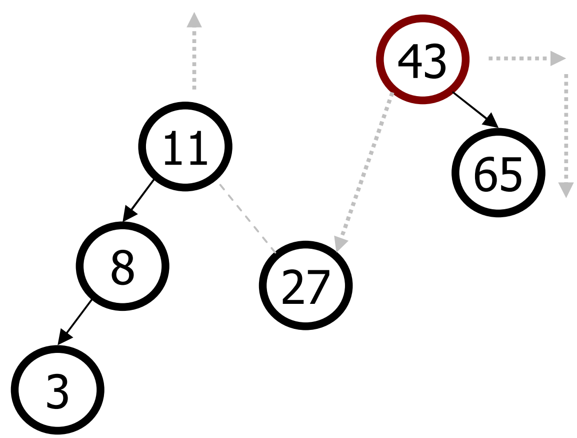
slides created by Marty Stepp

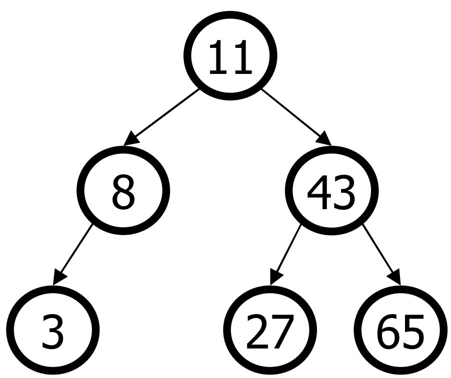
http://www.cs.washington.edu/373/

Right rotation

1. Detach left child (11)'s right subtree (27) *(don't lose it!)*
2. Consider left child (11) be the new parent.
3. Attach old parent (43) onto right of new parent (11).
4. Attach new parent (11)'s old right subtree (27) as left subtree of old parent (43).

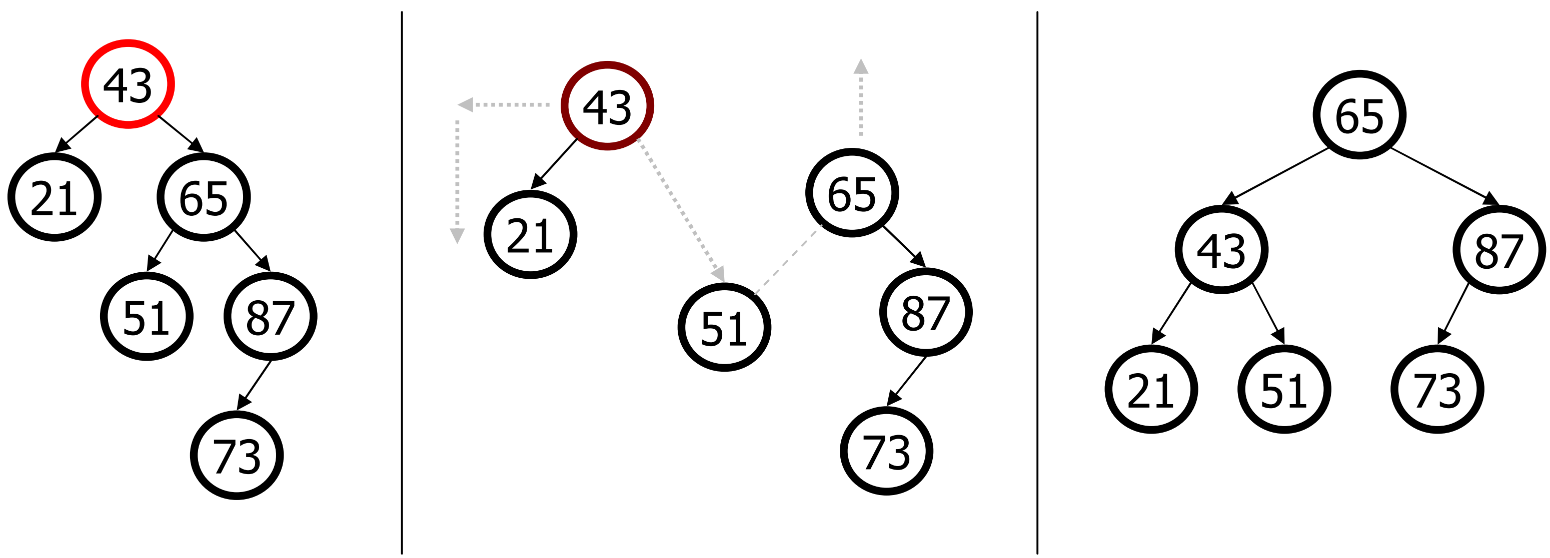






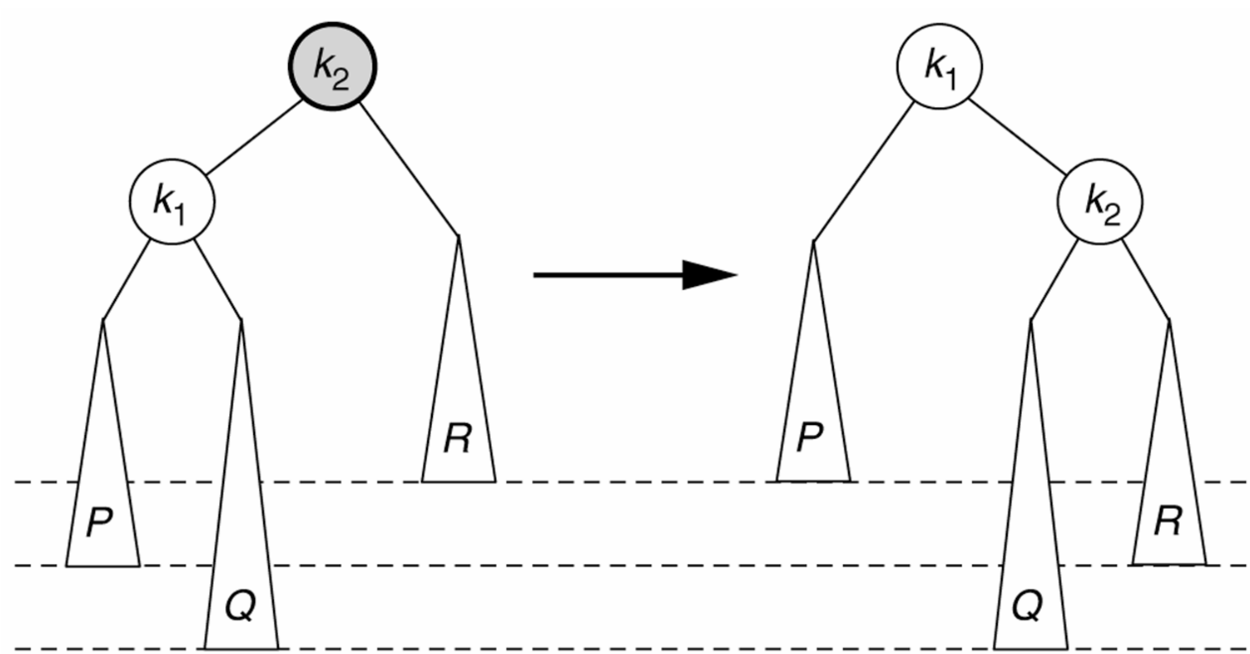
Left rotation

1. Detach right child (65)'s left subtree (51) *(don't lose it!)*
2. Consider right child (65) be the new parent.
3. Attach old parent (43) onto left of new parent (65).
4. Attach new parent (65)'s old left subtree (51)
   as right subtree of old parent (43).



Problem cases

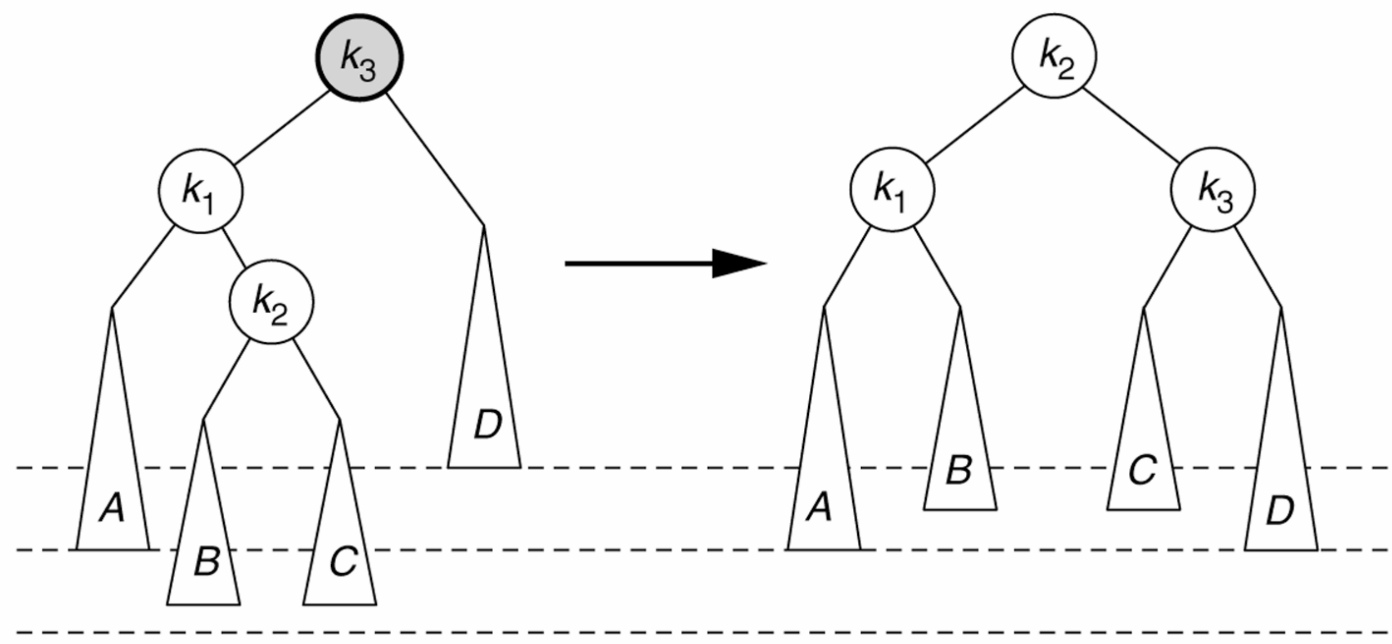
- A single right rotation does not fix Case 2 (LR).



  - (Similarly, a single left rotation does not fix Case 3 (RL).)

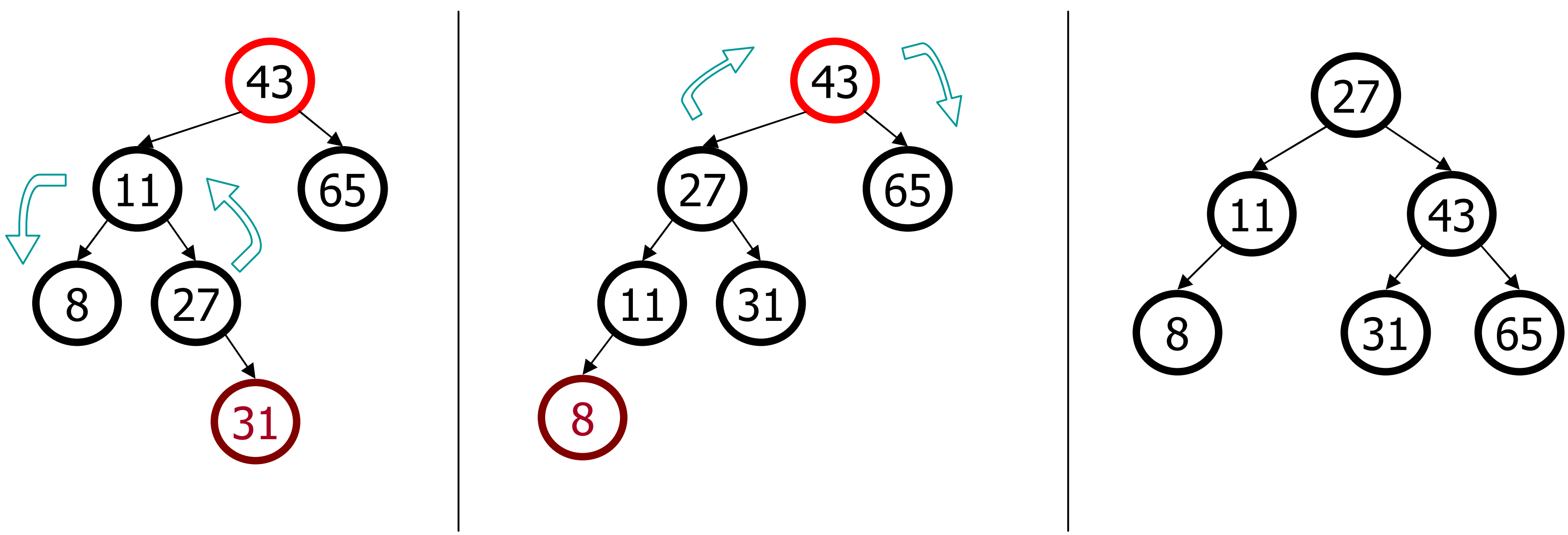
Left-right double rotation

- **left-right double rotation**: *(fixes Case 2 (LR))*
  - 1) left-rotate k_3's left child ... reduces Case 2 into Case 1
  - 2) right-rotate k_3 to fix Case 1



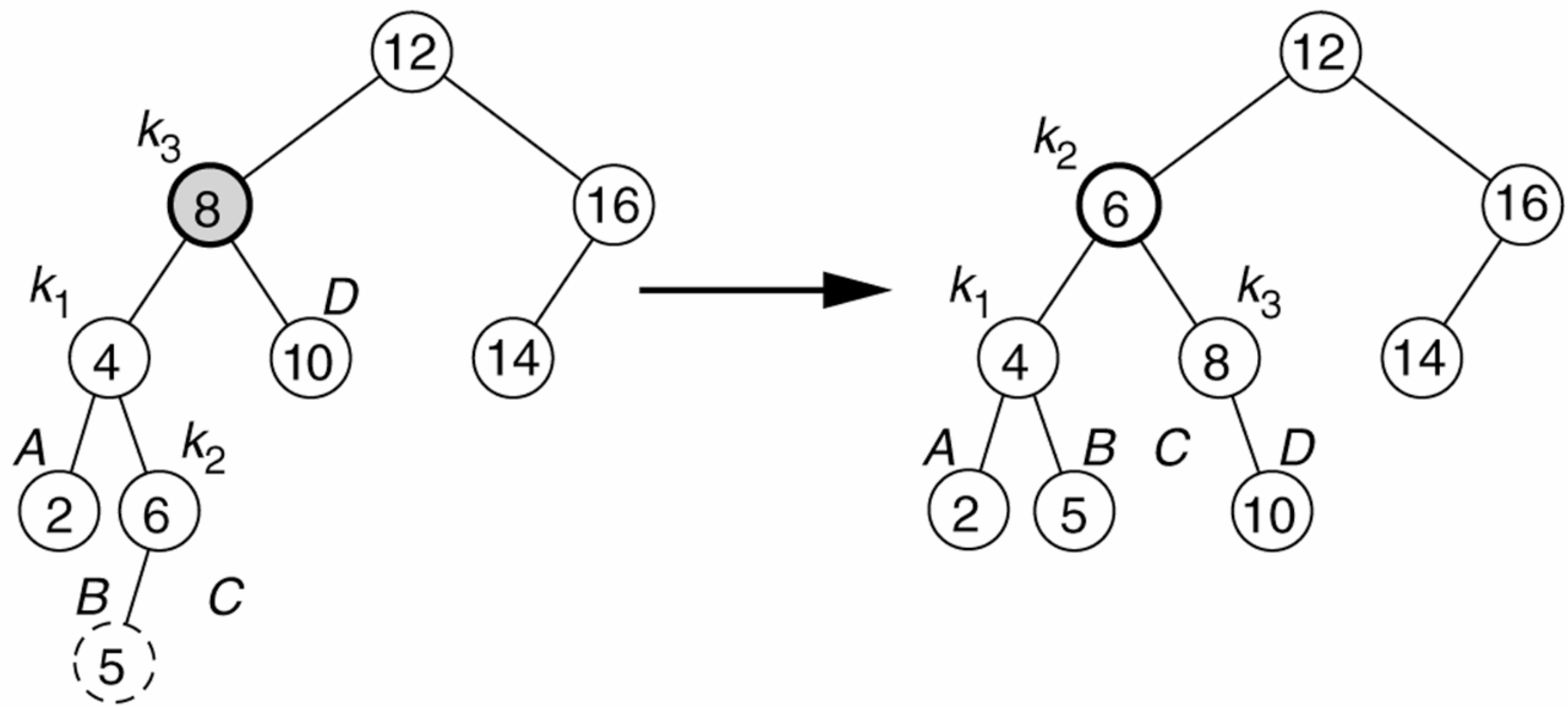
Left-right rotation steps

1. Left-rotate the overall parent's left child (11).
   - This reduces Case 2 (LR) to Case 1 (LL).
2. Right-rotate the overall parent (43).
   - This repairs Case 1 to be balanced.



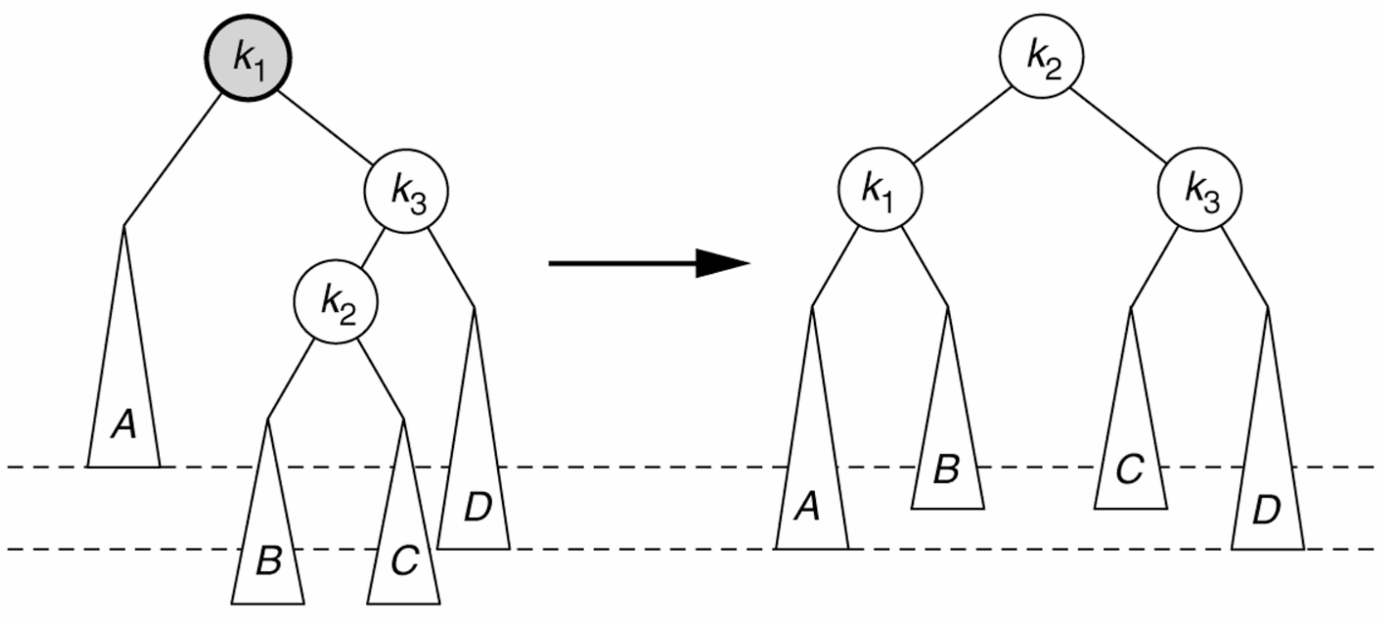
Left-right rotation example

- What is the balance factor of k_1, k_2, k_3 before and after rotating?



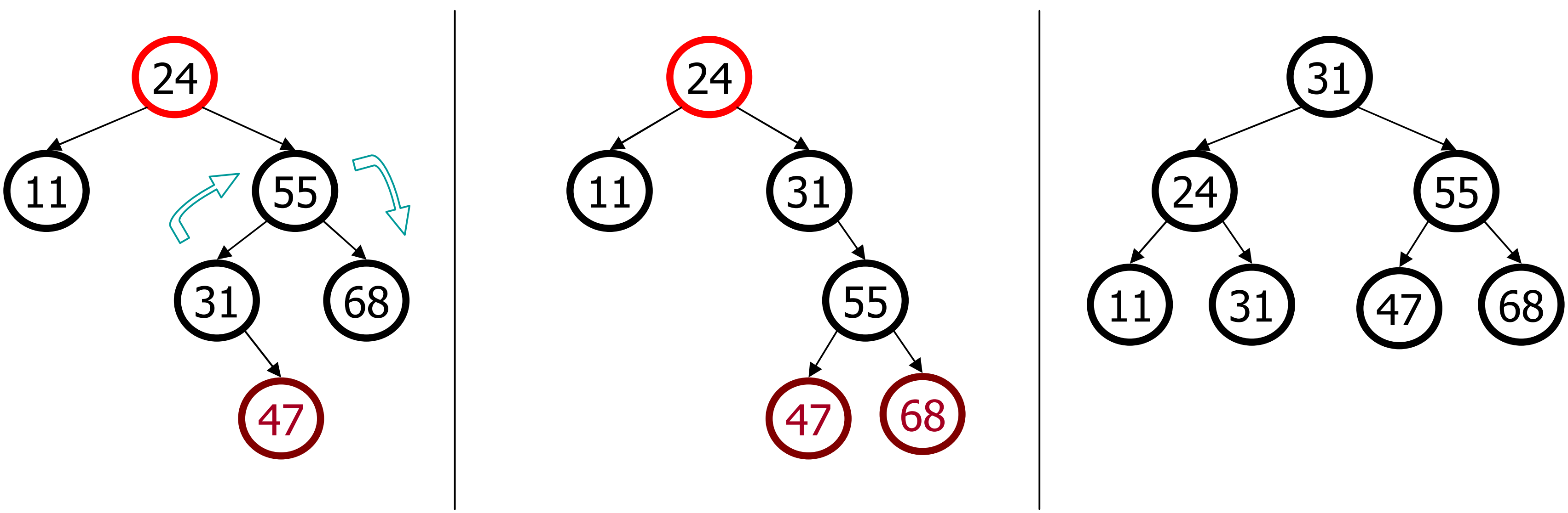
Right-left double rotation

- **right-left double rotation**: *(fixes Case 3 (RL))*
  - 1) right-rotate k_1's right child ... reduces Case 3 into Case 4
  - 2) left-rotate k_1 to fix Case 4



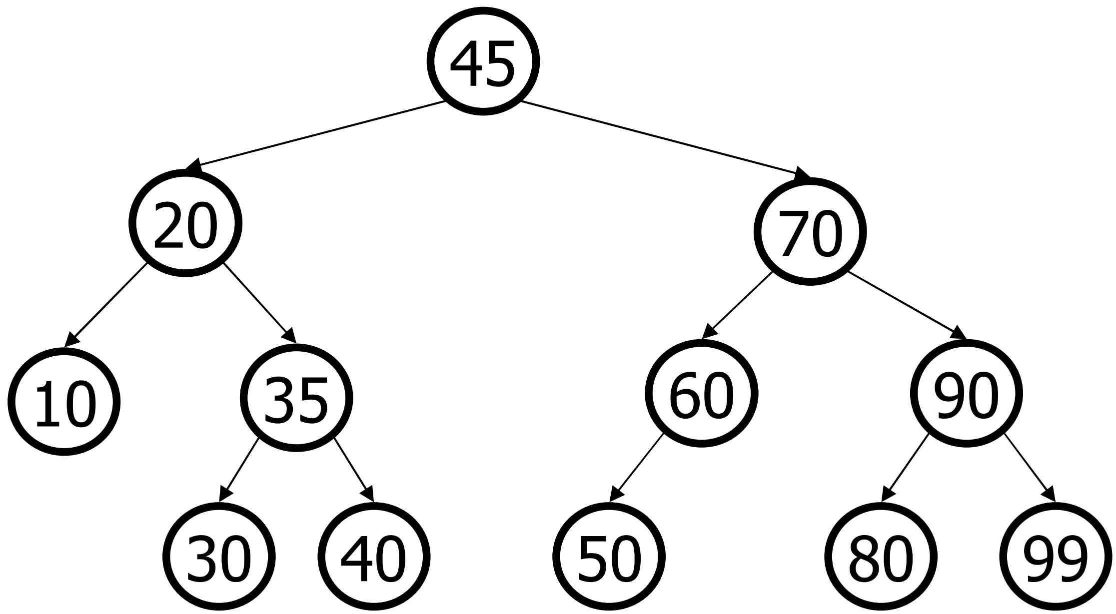
Right-left rotation steps

1. Right-rotate the overall parent's right child (11).
   - This reduces Case 3 (RL) to Case 4 (RR).
2. Right-rotate the overall parent (43).
   - This repairs Case 4 to be balanced.



AVL add example

- Draw the AVL tree that would result if the following numbers were added in this order to an initially empty tree:
  - 20, 45, 90, 70, 10, 40, 35, 30, 99, 60, 50, 80



Implementing add

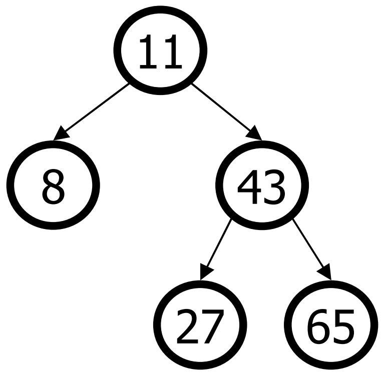
- Perform normal BST add. But as recursive calls return, update each node's height from new leaf back up to root.
  - If a node's balance factor becomes +/- 2, rotate to rebalance it.

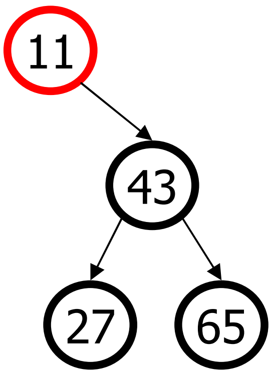
- How do you know which of the four Cases you are in?
  - Current node BF < -1 $\rightarrow$ Case 1 (LL) or 2 (LR).
    - look at current node's left child BF.
      - left child BF < 0 $\rightarrow$ Case 1 *(fix with R rotation)*
      - left child BF > 0 $\rightarrow$ Case 2 *(fix with LR rotations)*
  - Current node BF > 1 $\rightarrow$ Case 3 (RL) or 4 (RR).
    - look at current node's right child BF.
      - right child BF < 0 $\rightarrow$ Case 3 *(fix with RL rotations)*
      - right child BF > 0 $\rightarrow$ Case 4 *(fix with L rotation)*

AVL remove

- Removing from an AVL tree can also unbalance the tree.
  - Similar cases as with adding: LL, LR, RL, RR
  - Can be handled with the same remedies: rotate R, LR, RL, L

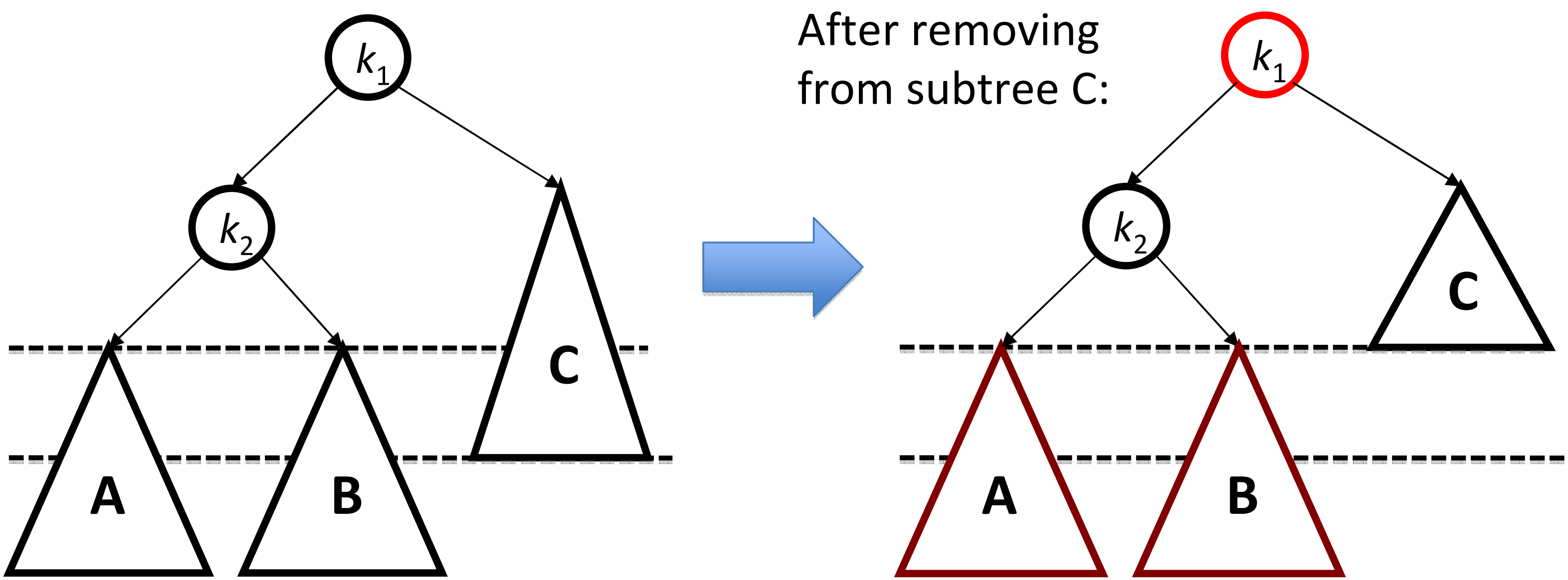
```
set.remove(8);
```





Remove extra cases

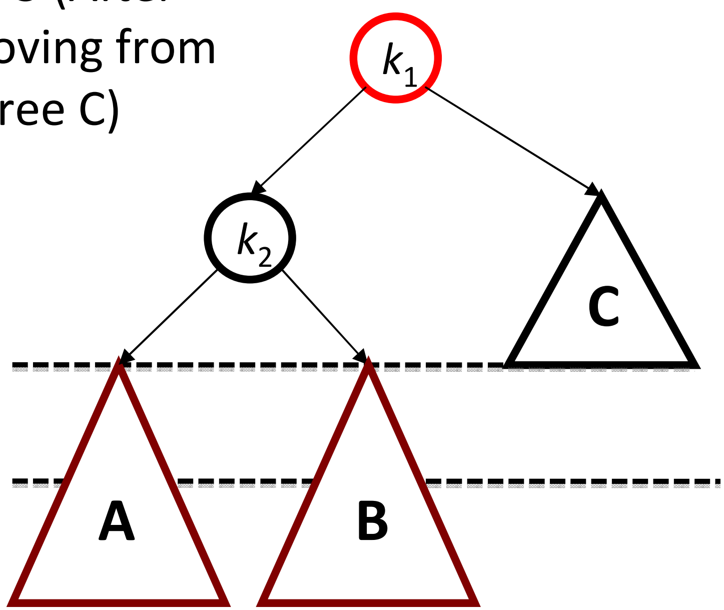
- AVL remove has 2 more cases beyond the 4 from adding:
  - In these cases, the offending subtree has a balance factor of 0. (The cause of imbalance is *both* LL *and* LR relative to k_1 below.)
  - When an add makes a node imbalanced, the child on the imbalanced side always has a balance factor of either -1 or 1.



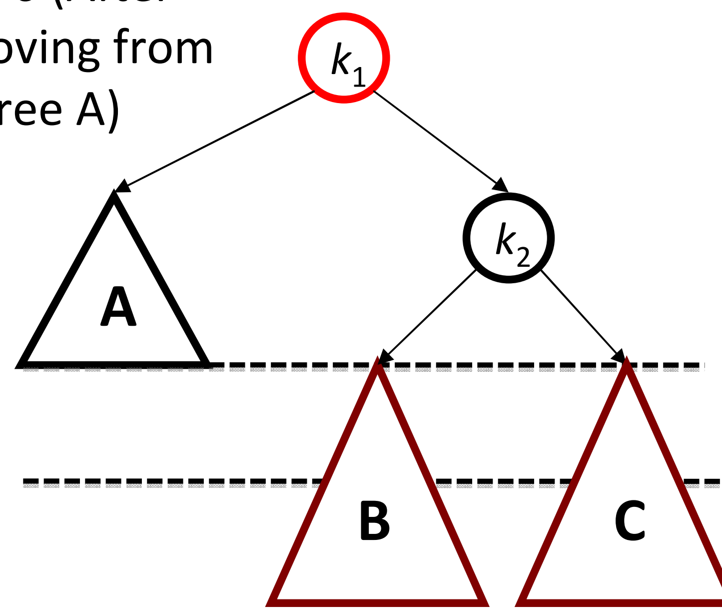
Labeling the extra cases

- Let's label these two new cases of remove imbalance:
  - *Case 5:* Problem is in both the LL and LR subtrees of the parent.
  - *Case 6:* Problem is in both the RL and RR subtrees of the parent.

Case 5 (After removing from subtree C)



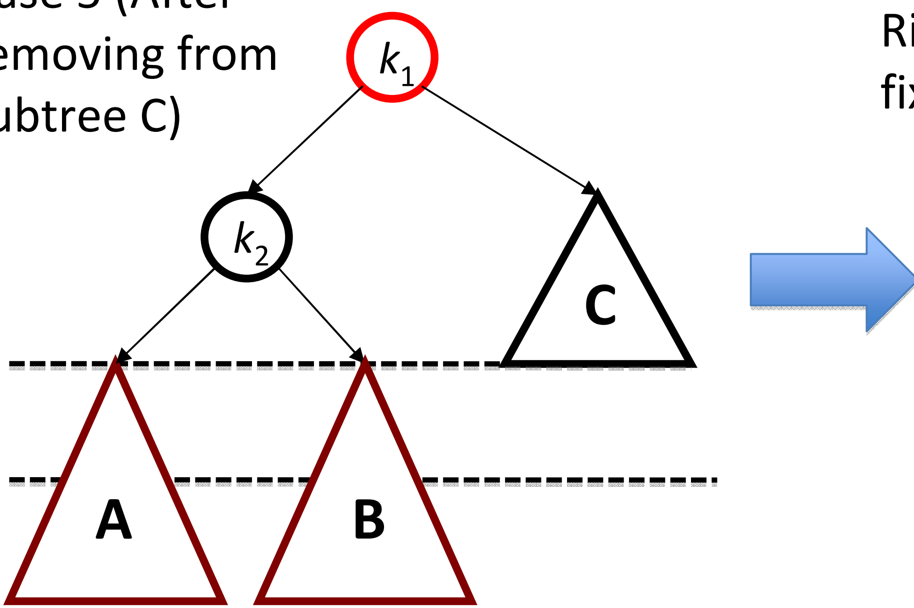
Case 6 (After removing from subtree A)



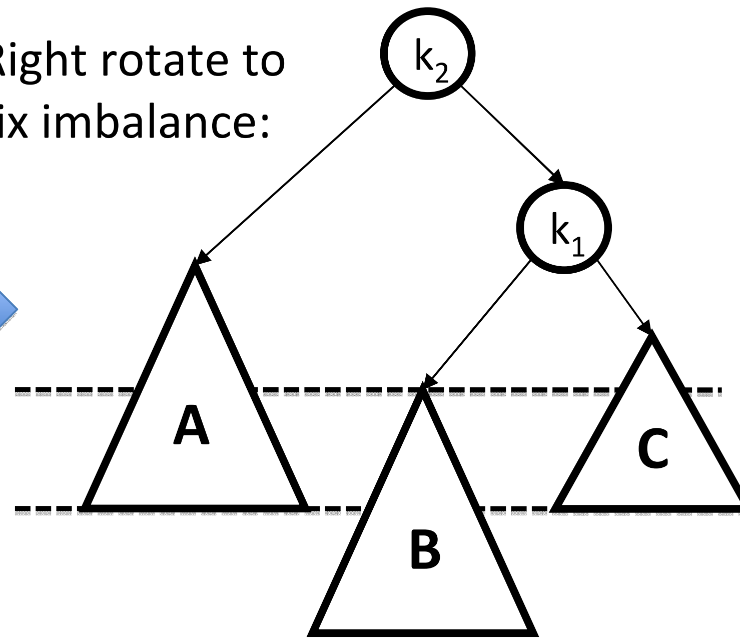
Fixing remove cases

- Each of these new cases can be fixed through a single rotation:
  - To fix Case 5, we right rotate *(shown below)*
  - To fix Case 6, we left rotate *(symmetric case)*

Case 5 (After removing from subtree C)



Right rotate to fix imbalance:

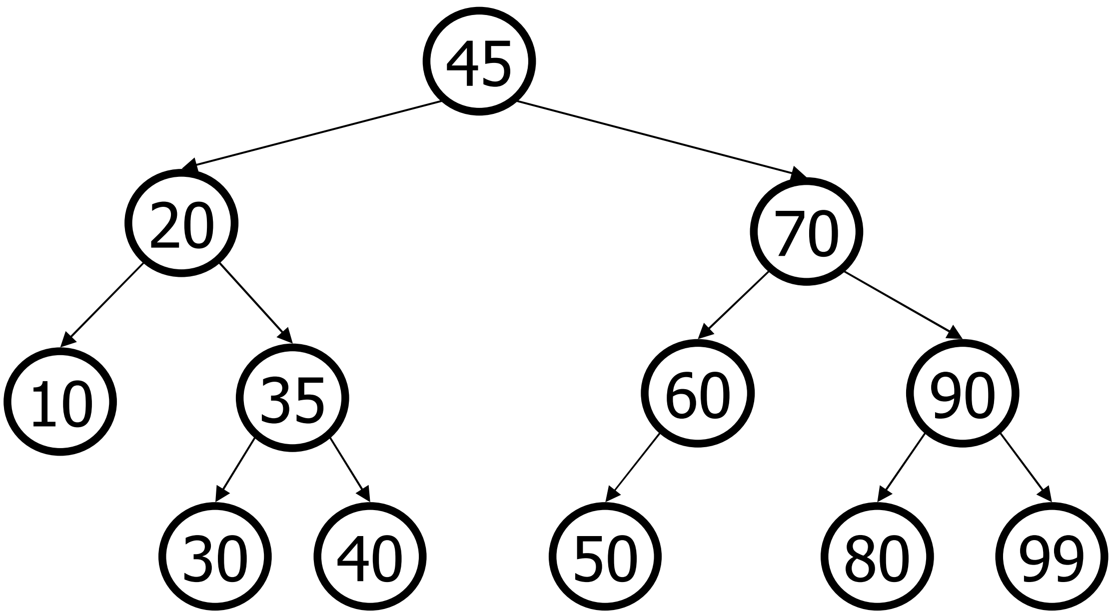


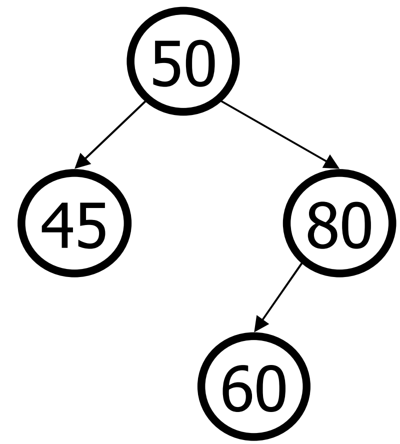
Implementing remove

- Perform normal BST remove. But as recursive calls return, update each node's height from new leaf back up to root.
  - If a node's balance factor becomes +/- 2, rotate to rebalance it.
  - Current node BF < -1 → Case 1 (LL) or 2 (LR) or 5 (L-both).
    - look at current node's left child BF.
      - left child BF < 0 → Case 1 *(fix with R rotation)*
      - left child BF > 0 → Case 2 *(fix with LR rotations)*
      - left child BF = 0 → Case 5 *(fix with R rotation)*
  - Current node BF > 1 → Case 3 (RL) or 4 (RR) or 6 (R-both).
    - look at current node's right child BF.
      - right child BF < 0 → Case 3 *(fix with RL rotations)*
      - right child BF > 0 → Case 4 *(fix with L rotation)*
      - right child BF = 0 → Case 6 *(fix with L rotation)*

AVL remove example

- Suppose we start with the AVL tree below.
  - Draw the AVL tree that would result if the following numbers were removed in this order from the tree:
    - 10, 30, 40, 35, 70, 90, 99





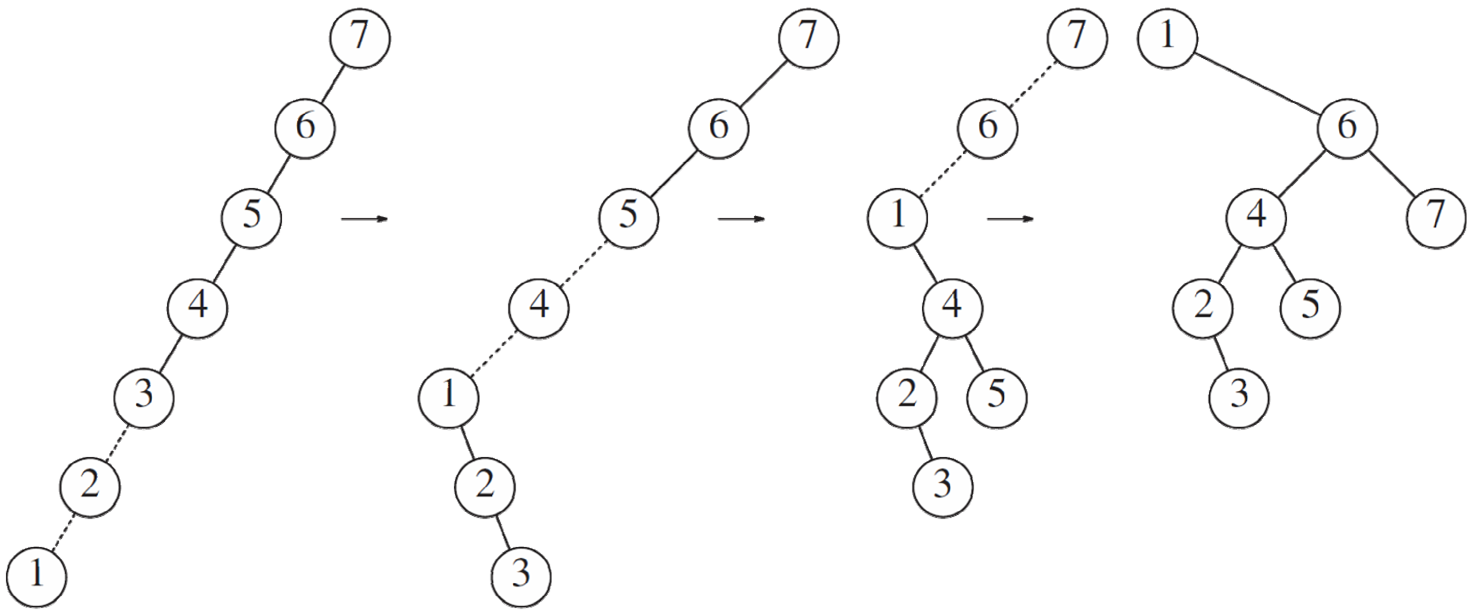
How efficient is AVL?

- An AVL tree has the same general operations as a BST, with rebalancing added to the add/remove operations.
  - How much time does it take to rebalance the tree?
    - How much time does it take to rebalance one node?
    - How many nodes at most will we need to rebalance per add/remove?

  - **add:** O(log N) to walk down the tree and add the element;
    O(log N) number of nodes' heights to update;
    O(1) single node may need to be rotated. O(log *N*) overall.
  - **contains:** Unmodified. O(log *N*).
  - **remove:** O(log *N*) to walk down the tree and remove the element;
    O(log *N*) number of nodes' heights to update;
    O(1) single node may need to be rotated. O(log *N*) overall.

Other interesting trees

- **splay tree:** Rotates each element you access to the top/root
  - very efficient when that element is accessed again (happens a lot)
  - easy to implement and does not need height field in each node



Other interesting trees

- **red-black tree:** Gives each node a "color" of red or black.
  - Root is black. Root's direct children are red.
  - If a node is red, its children must all be black.
  - Every path downward from a node to the bottom must contain the same number of "black" nodes.