BLG 223E - Recitation 5
Balanced Trees

M. Alpaslan Tavukçu (tavukcu22@itu.edu.tr)
Batuhan Can (canb18@itu.edu.tr)

Outline

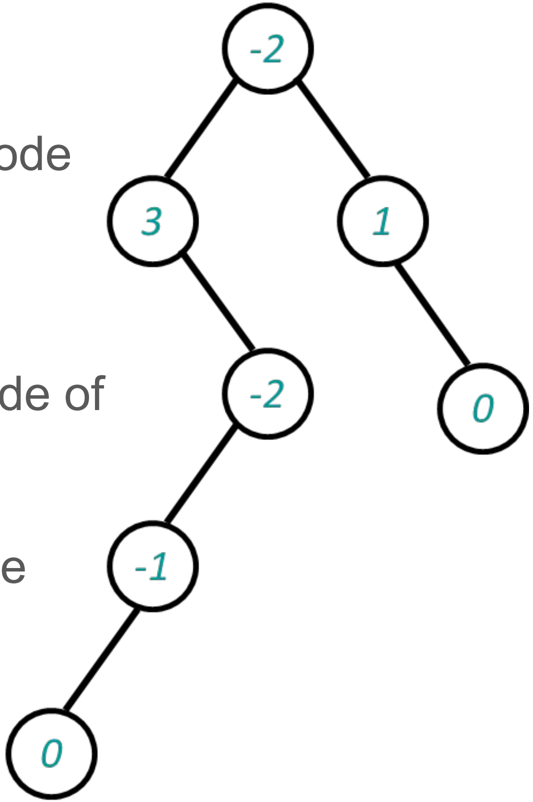
- AVL Trees
- AVL Tree Implementation
- Red-Black Trees
- Red-Black Tree Implementation

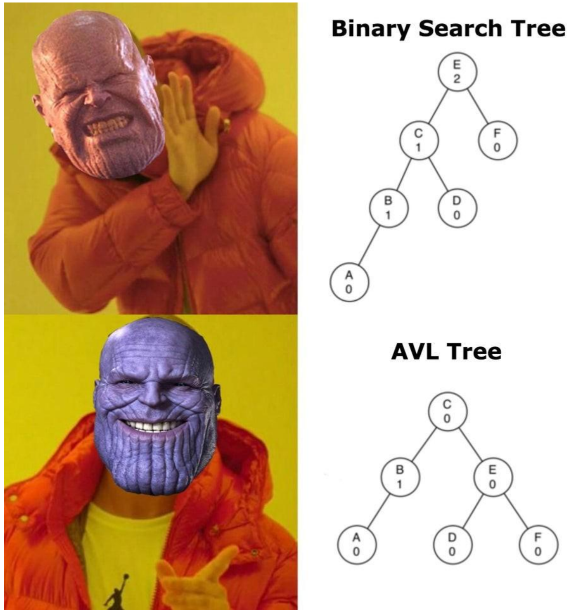
Balanced Trees

- We want to benefit from the logarithmic characteristic of binary trees.
- BST may suffer in some cases, lose its balance, and act like it is an almost-LinkedList.
- Using a self-balancing tree is the choice, if we want to maintain the rapidness of tree operations.
- AVL Trees, Red-black Trees, etc.

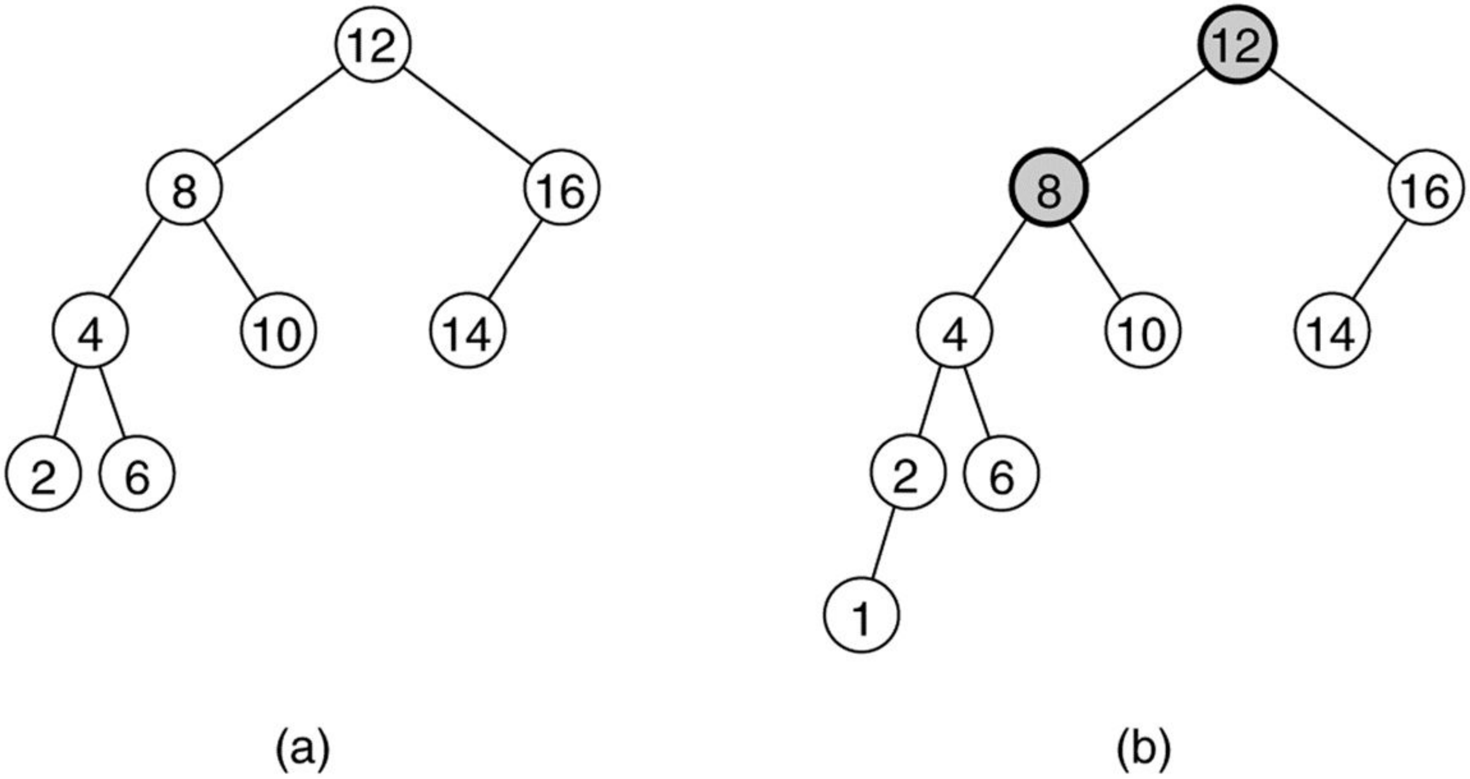
AVL Trees

- Balance factor, for a tree node
  - BF(T) = h(T.Right) - h(T.Left)
- An AVL Tree maintains a “balance factor” in each node of 0, 1, or -1
- Example shows the balance factor of each node





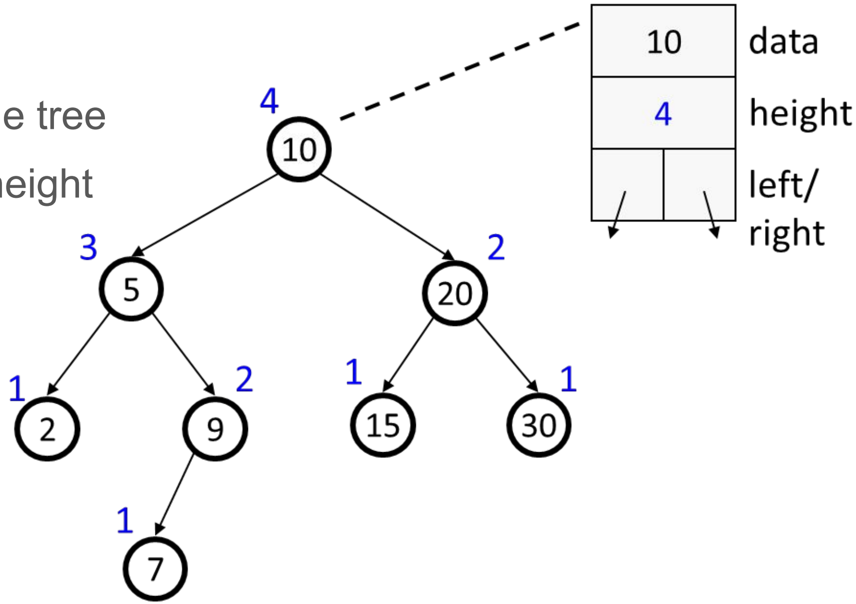
AVL Trees : Examples



AVL Trees : Subtree Height

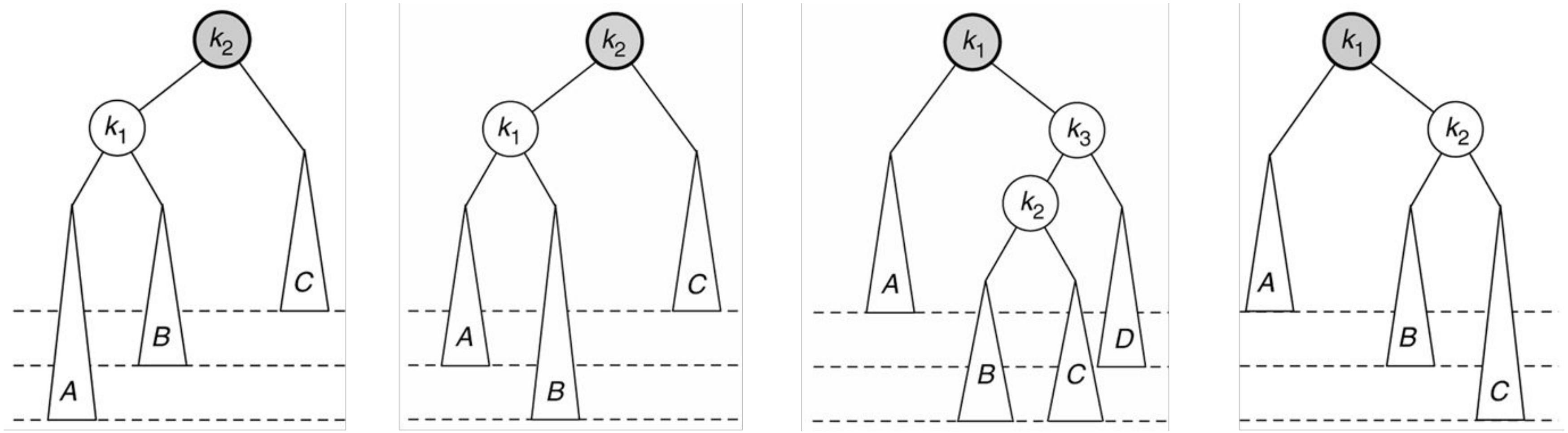
- AVL tree operations depend on height
- It can be computed recursively by walking the tree
- Or each node can keep track of its subtree height as a field

```
private class TreeNode {
    private E data;
    private int height;
    private TreeNode left;
    private TreeNode right;
}
```



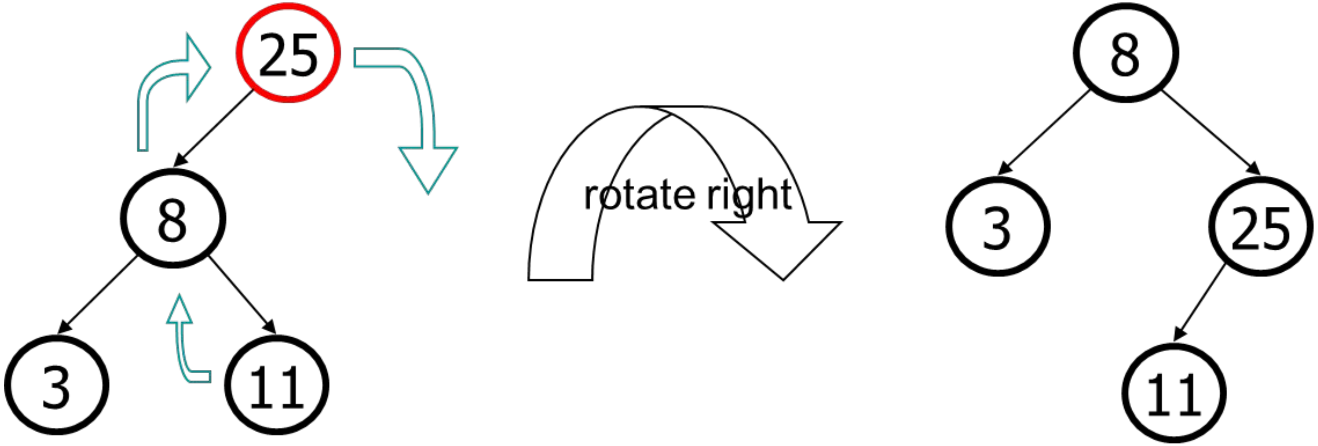
AVL : Insert Cases

- Consider the lowest node k_2 that has now become unbalanced.
- The new node could be in one of the four following grandchild subtrees relative to k_2. Left-Left, Left-Right, Right-Left, Right-Right



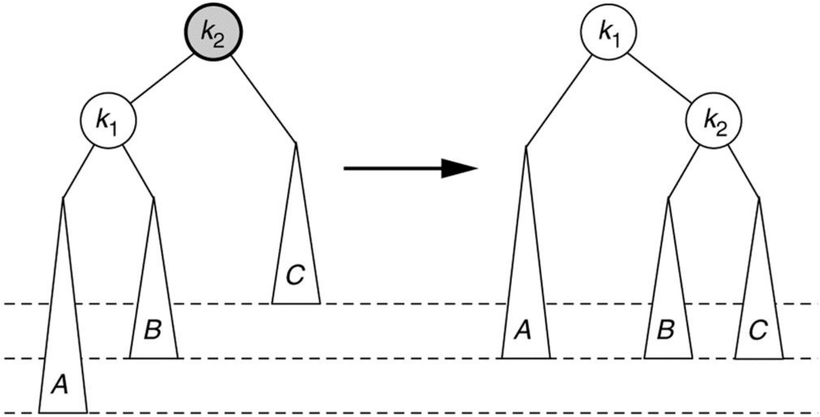
AVL Trees : Rotation

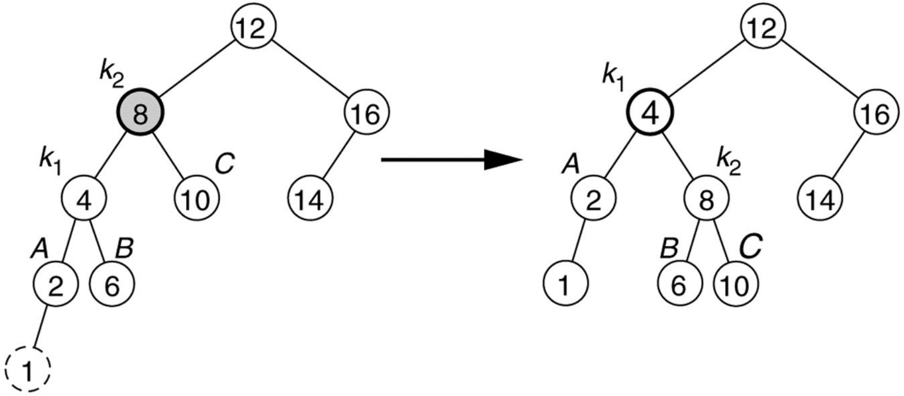
- If a node has become out of balanced in a given direction, rotate it in the opposite direction
- *rotation* : A swap between parent and left or right child, maintaining BST ordering



AVL Trees : Right Rotation

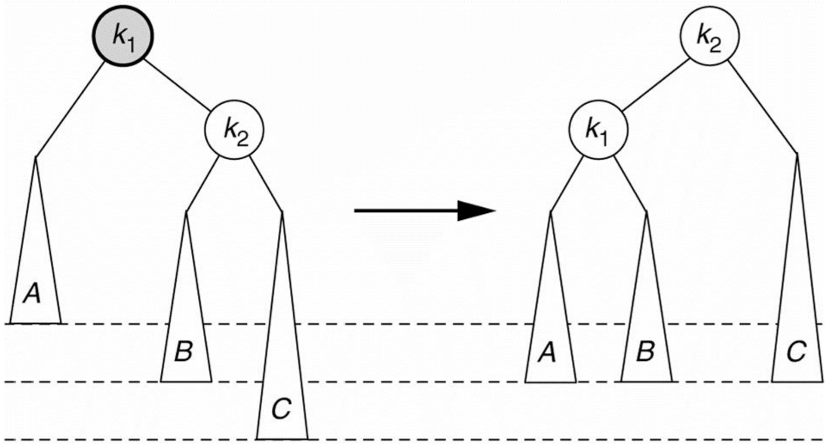
- Left child k_1 becomes parent
- Original parent k_2 demoted to right
- k_1’s original right subtree B (if any) is attached to k_2 as left subtree
- **It fixes Case 1 (Left-Left)**

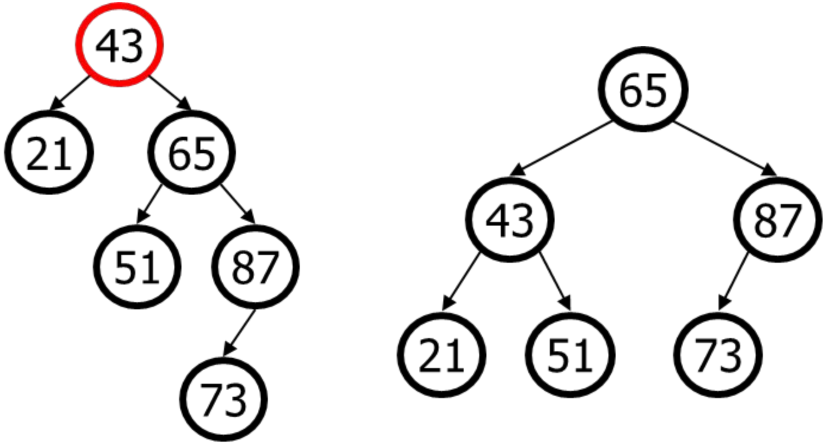




AVL Trees : Left Rotation

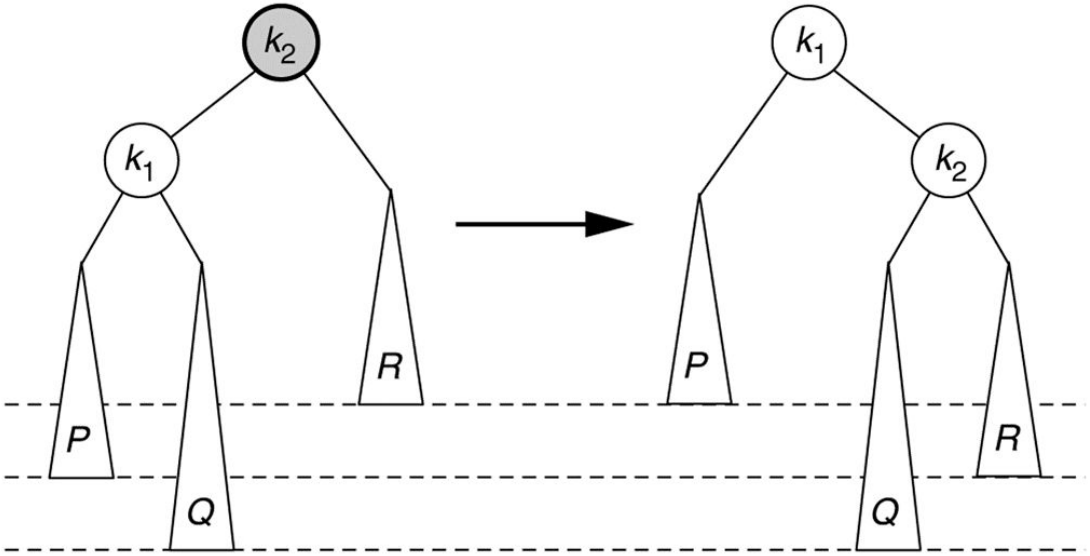
- Right child k_2 becomes parent
- Original parent k_1 demoted to left
- k_2’s original right subtree B (if any) is attached to k_1 as left subtree
- **It fixes Case 4 (Right-Right)**





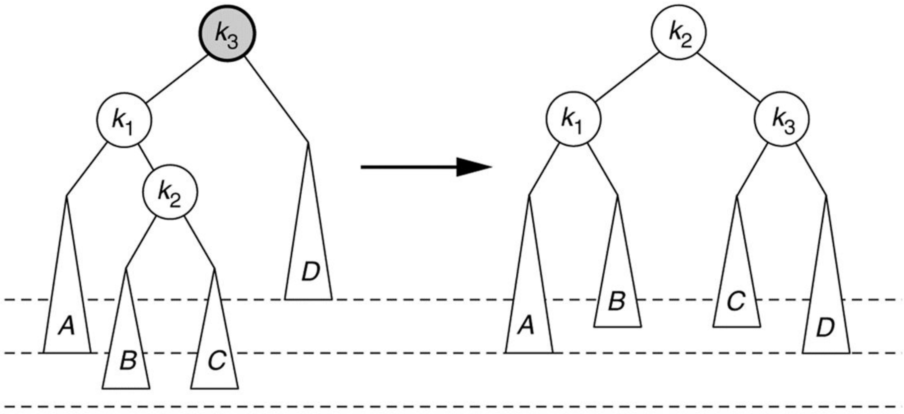
AVL Trees : Problem Cases

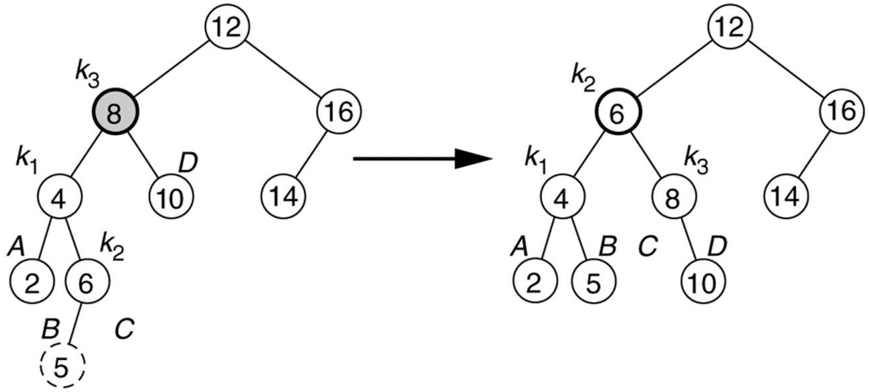
- A single right rotation does not fix Case 2 (Left-Right)
- A single left rotation does not fix Case 3 (Right-Left)



AVL Trees : Left-Right Double Rotation

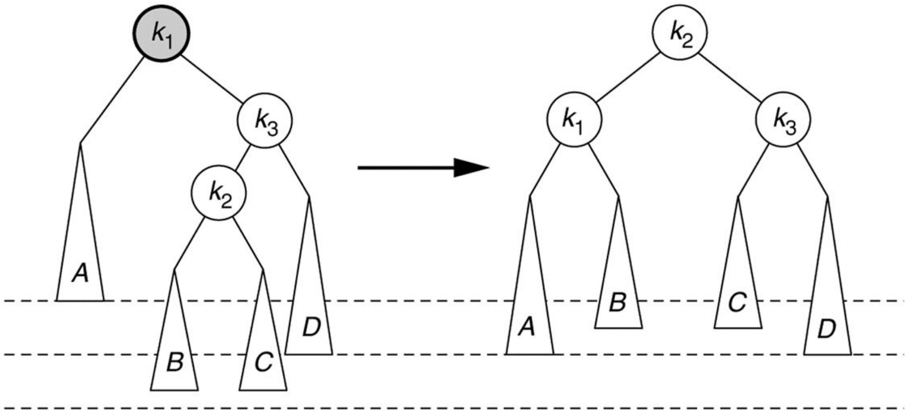
- Left-rotate k_3's left child
- Right-rotate k_3
- It fixes Case 2 (Left-Right)





AVL Trees : Right-Left Double Rotation

- Right-rotate k_1's right child
- Left-rotate k_1
- It fixes Case 3 (Right-Left)

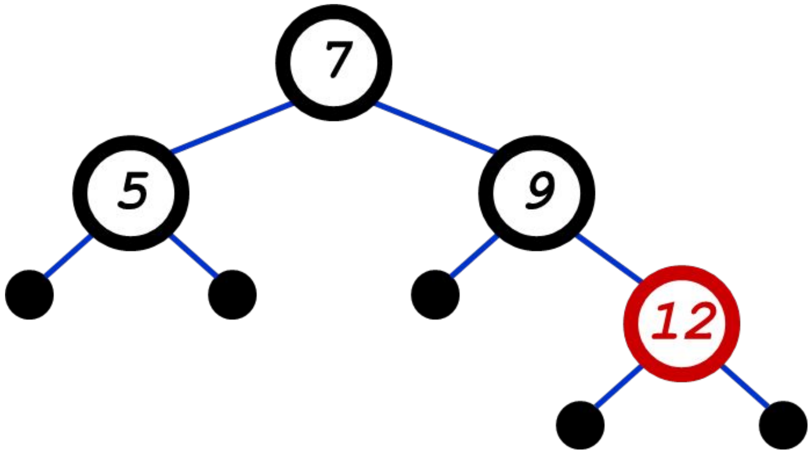


Red-Black Trees

- Some rules to follow :
  1. Every node is either red or black
  2. Every leaf (null pointer) is black
  3. If a node is red, both children are black
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black
- Black-height : number of black nodes on path to leaf

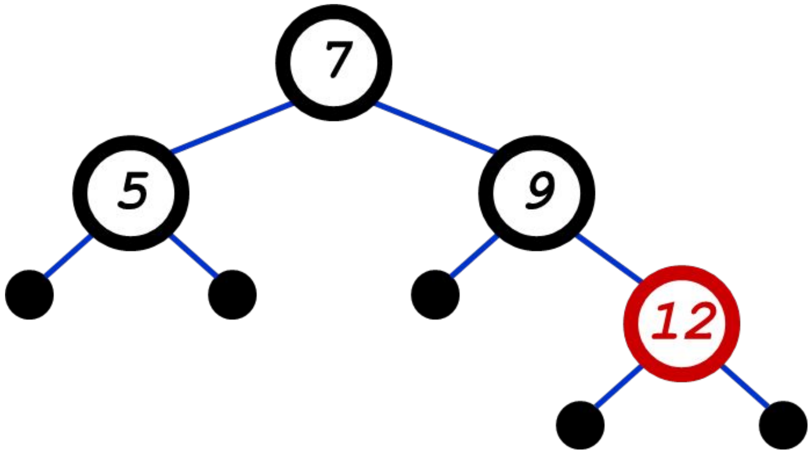
Red-Black Trees : An Example : Coloring

- Some rules to follow :
  1. Every node is either red or black
  2. Every leaf (null pointer) is black
  3. If a node is red, both children are black
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black



Red-Black Trees : The Problem With Insertion

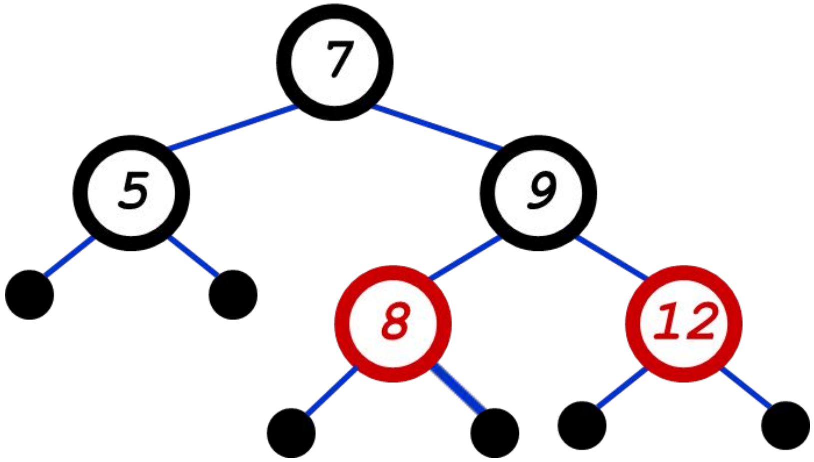
Let's try to insert 8, and see what happens



- Some rules to follow :
    1. Every node is either red or black
    2. Every leaf (null pointer) is black
    3. If a node is red, both children are black
    4. Every path from node to descendent leaf contains the same number of black nodes
    5. The root is always black

Red-Black Trees : The Problem With Insertion

Let's try to insert 11, and see what happens

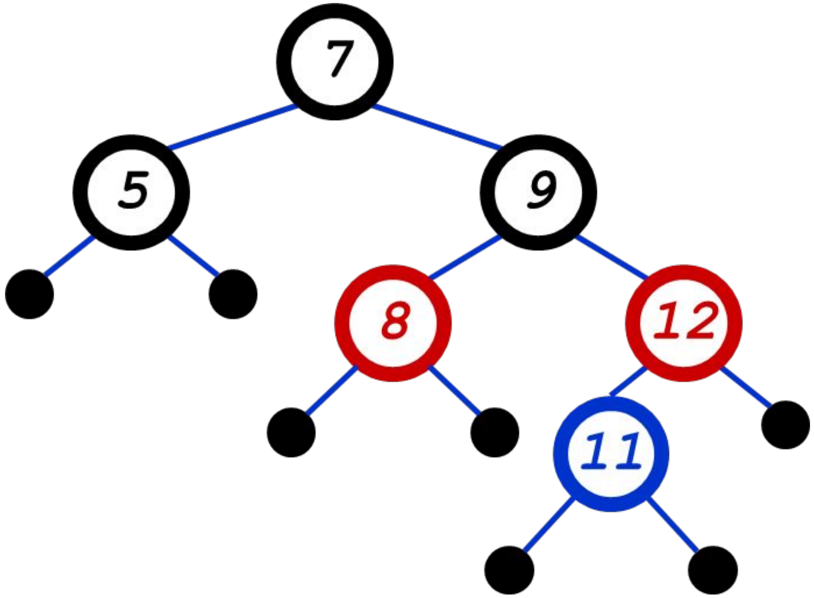


- Some rules to follow :
  1. Every node is either red or black
  2. Every leaf (null pointer) is black
  3. If a node is red, both children are black
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

Red-Black Trees : The Problem With Insertion

Let's try to insert 11, and see what happens

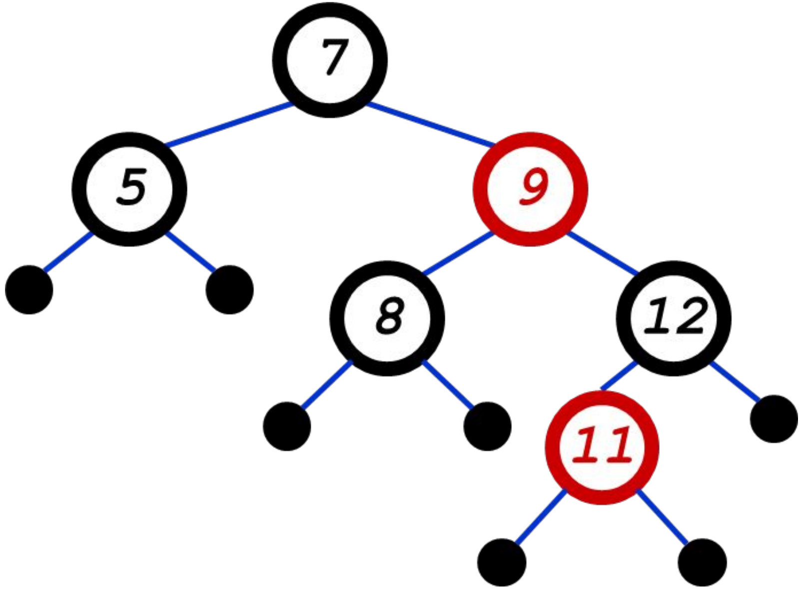
We need to RECOLOR



- Some rules to follow :
  1. Every node is either red or black
  2. Every leaf (null pointer) is black
  3. If a node is red, both children are black
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

Red-Black Trees : The Problem With Insertion

Let's try to insert 10, and see what happens



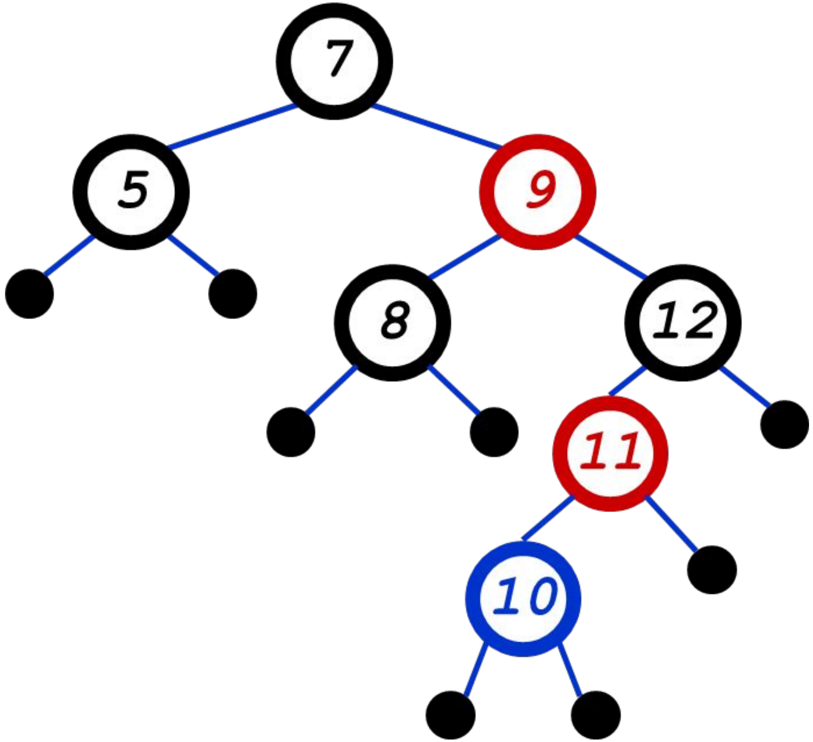
- Some rules to follow :
  1. Every node is either red or black
  2. Every leaf (null pointer) is black
  3. If a node is red, both children are black
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

Red-Black Trees : The Problem With Insertion

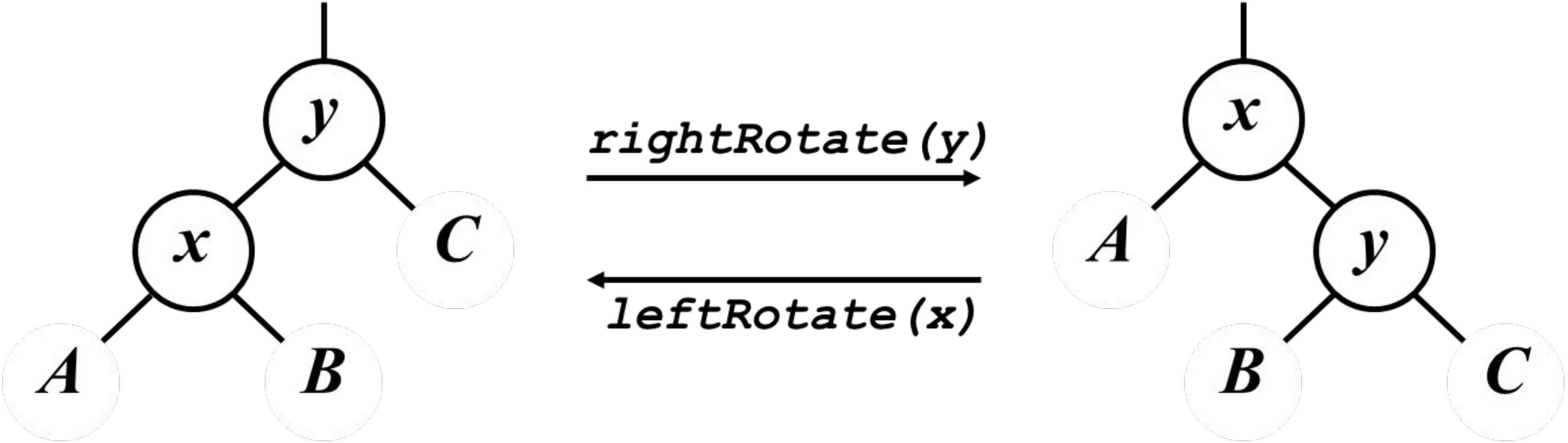
Let's try to insert 10, and see what happens

Recoloring does not work, we need to ROTATE

- Some rules to follow :
    1. Every node is either red or black
    2. Every leaf (null pointer) is black
    3. If a node is red, both children are black
    4. Every path from node to descendent leaf contains the same number of black nodes
    5. The root is always black

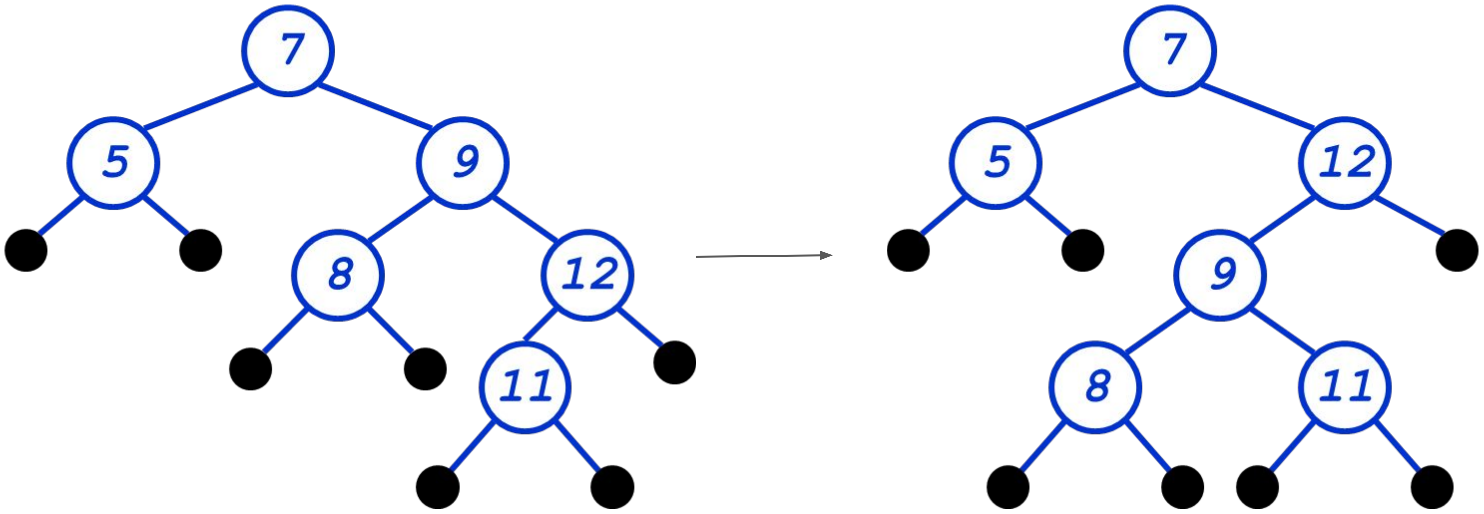


Red-Black Trees : Rotation



Is in-order key ordering preserved after rotation ?

Red-Black Trees : Rotation Example



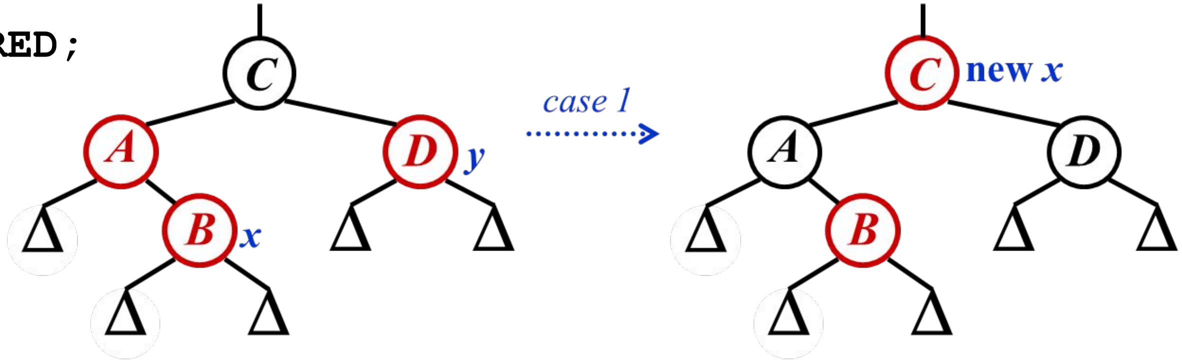
Red-Black Trees : Insertion

The basic idea is :

- Insert x into tree, color x red
- Only r-b property 3 might be violated
- If so, move violation up tree until a place is found where it can be fixed

Red-Black Trees : Insert Case #1

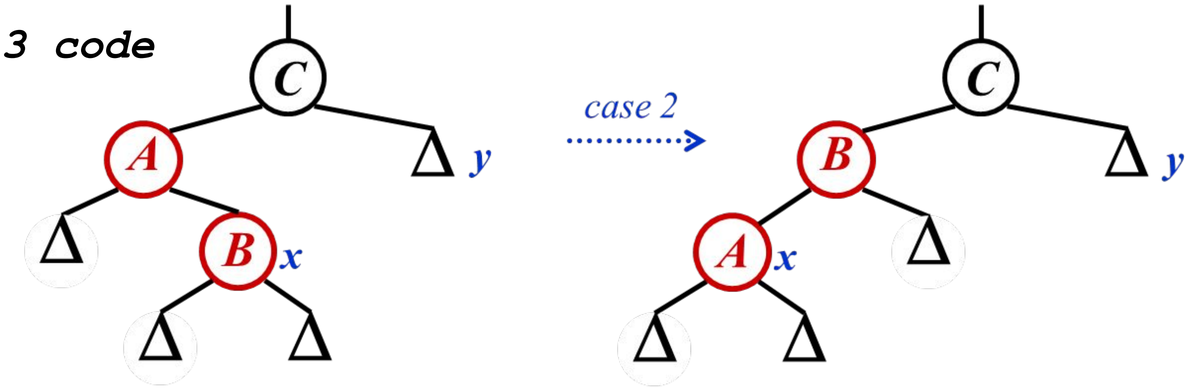
```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```



Change colors of some nodes, preserving #4: all downward paths have equal b.h.
The while loop now continues with x's grandparent as the new x

Red-Black Trees : Insert Case #2

```
if (x == x->p->right)
    x = x->p;
    leftRotate(x);
// continue with case 3 code
```

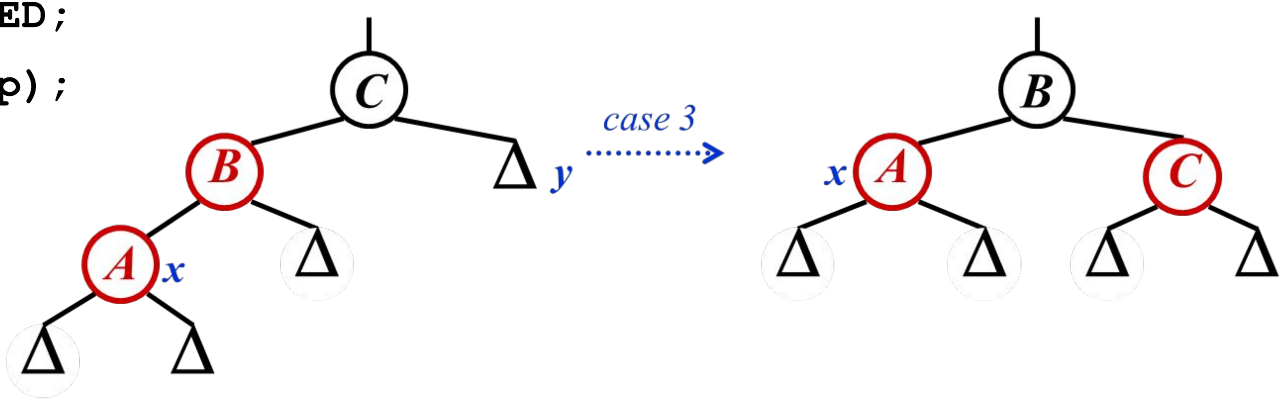


Transform case 2 into case 3 (x is left child) with a left rotation

This preserves property 4: all downward paths contain same number of black nodes

Red-Black Trees : Insert Case #3

```
x->p->color = BLACK;
x->p->p->color = RED;
rightRotate(x->p->p);
```



Perform some color changes and do a right rotation

Again, preserves property 4: all downward paths contain same number of black nodes

Red-Black Trees : A Quick Note

- Cases #1, #2, and #3 hold if x’s parent is a left child
- If x’s parent is a right child, cases #4, #5, and #6 are symmetric