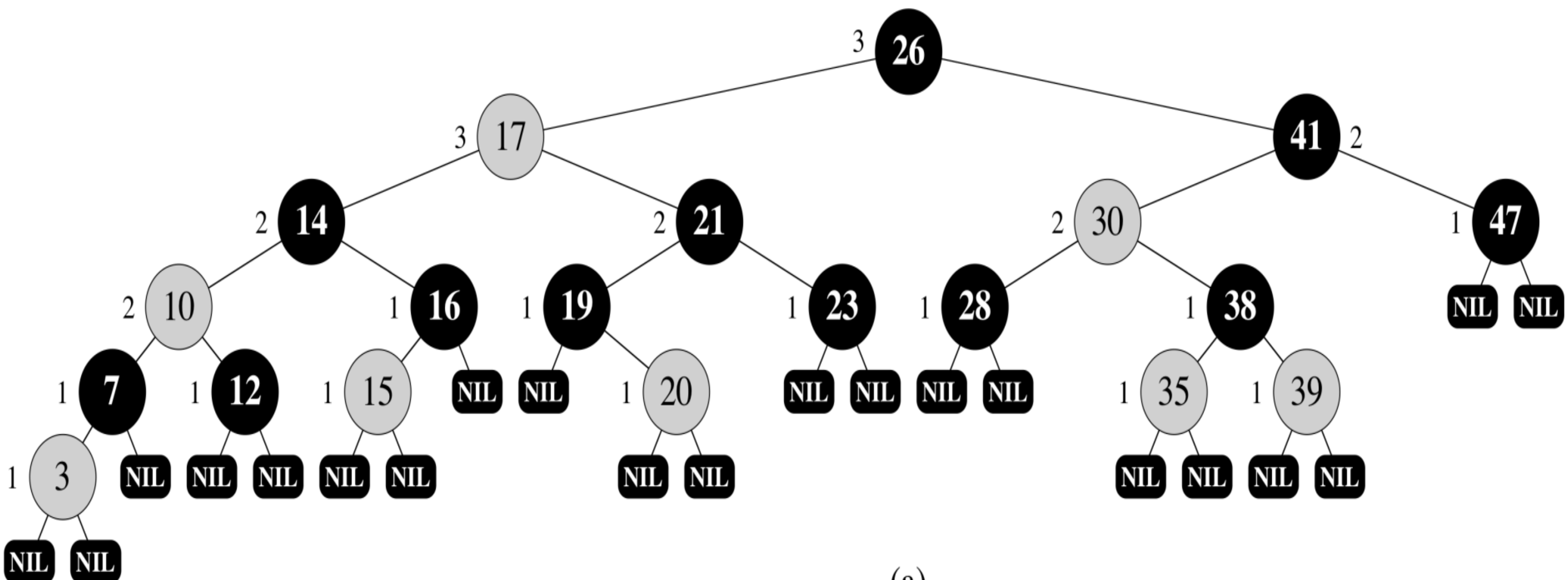
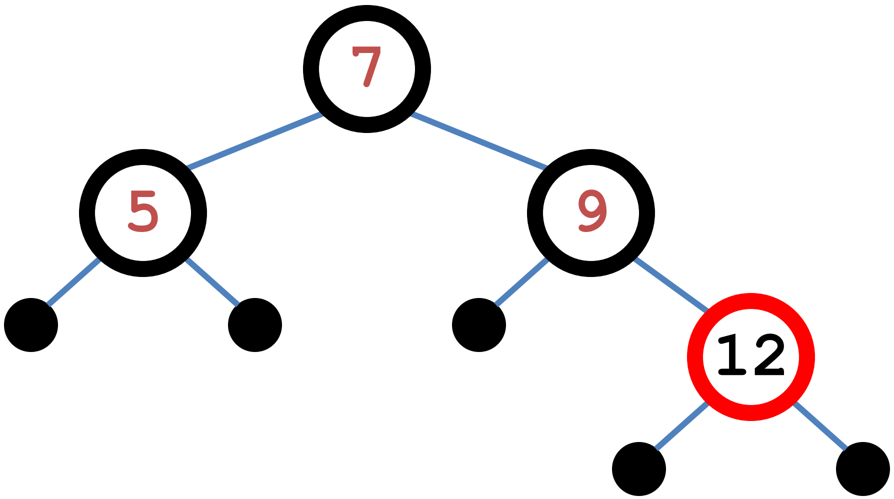
Example: Red Black Tree



(a)

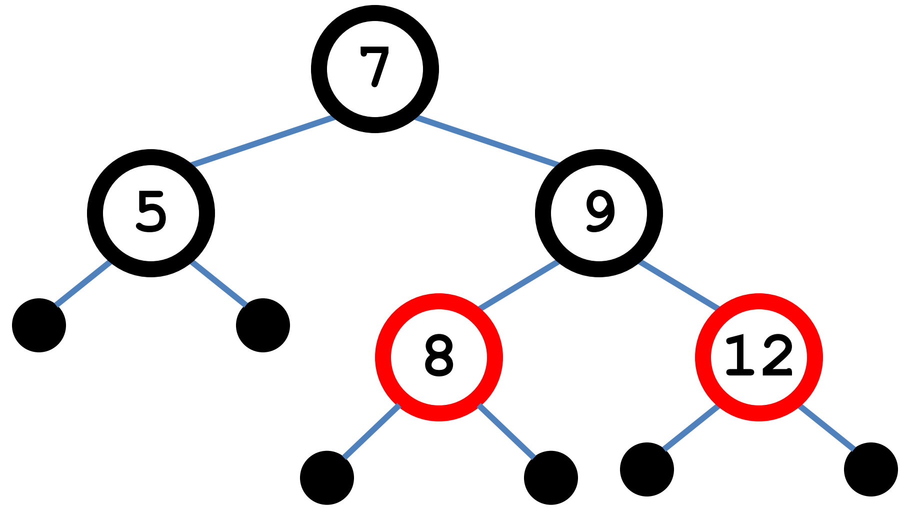
Red-Black Trees: An Example

- *Color this tree:*



Red-Black Trees: The Problem With Insertion

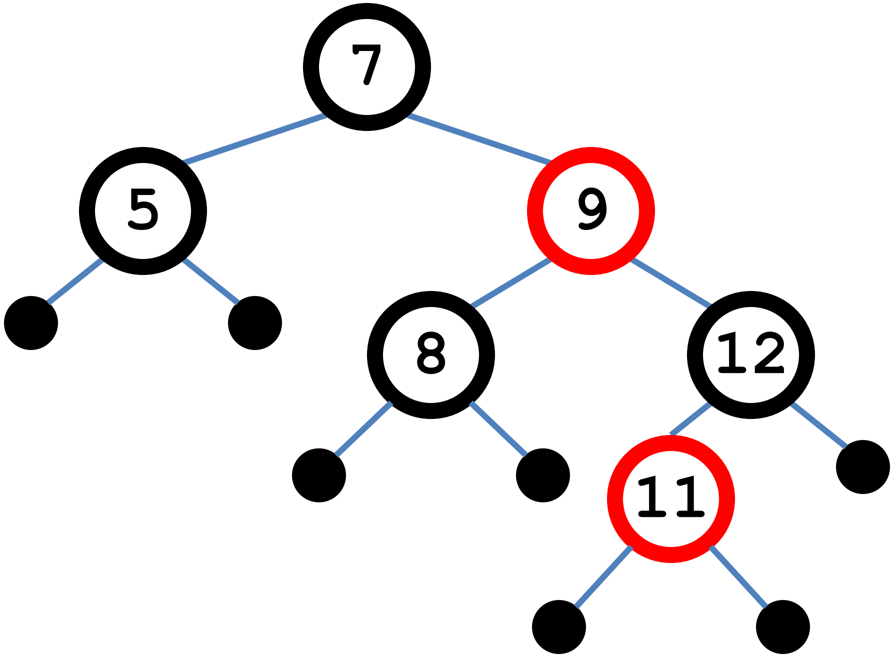
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



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

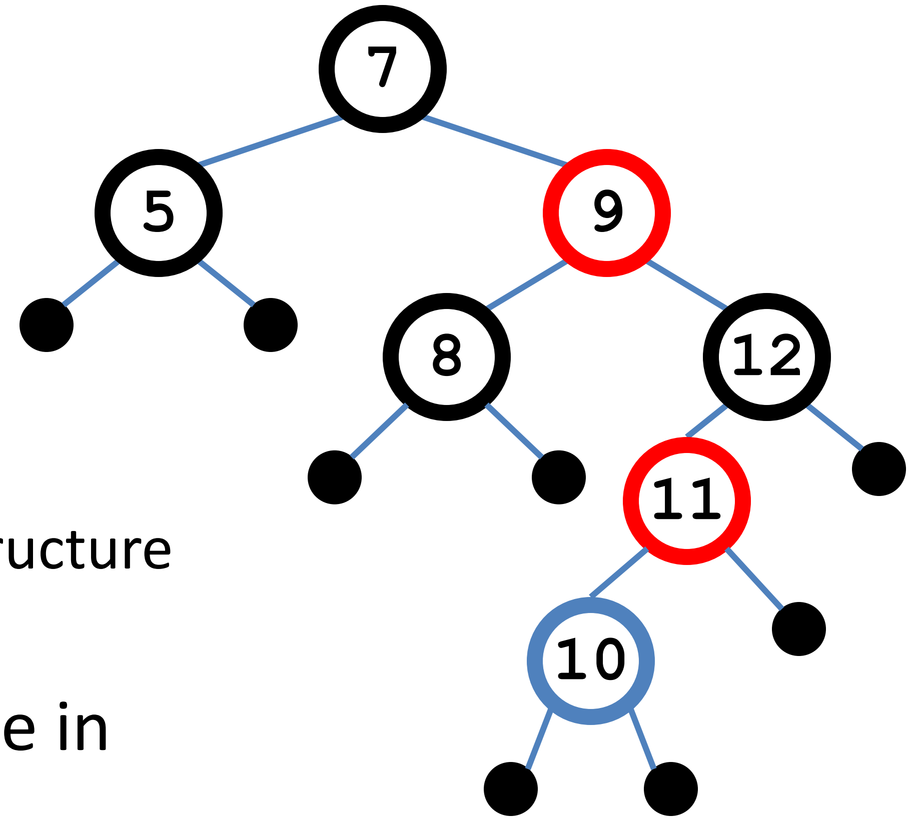
Red-Black Trees:
The Problem With Insertion

- Insert 11
  - *Where does it go?*
  - *What color?*
    - Solution:
      recolor the tree



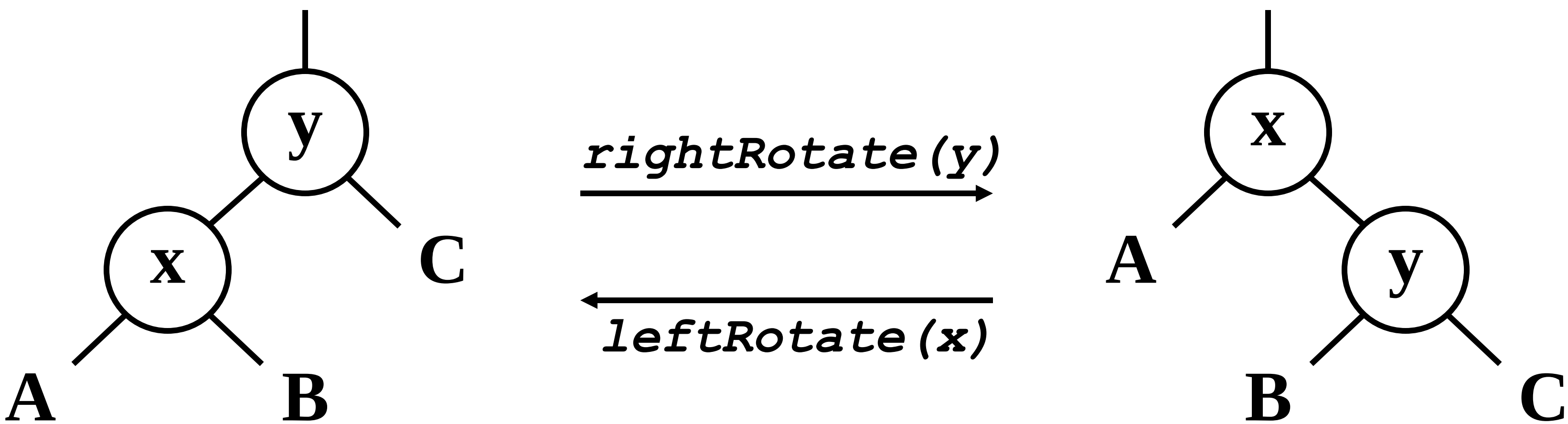
Red-Black Trees: The Problem With Insertion

- Insert 10
  - *Where does it go?*
  - *What color?*
    - A: no color! Tree is too imbalanced
    - Must change tree structure to allow recoloring
  - Goal: restructure tree in O(lg *n*) time



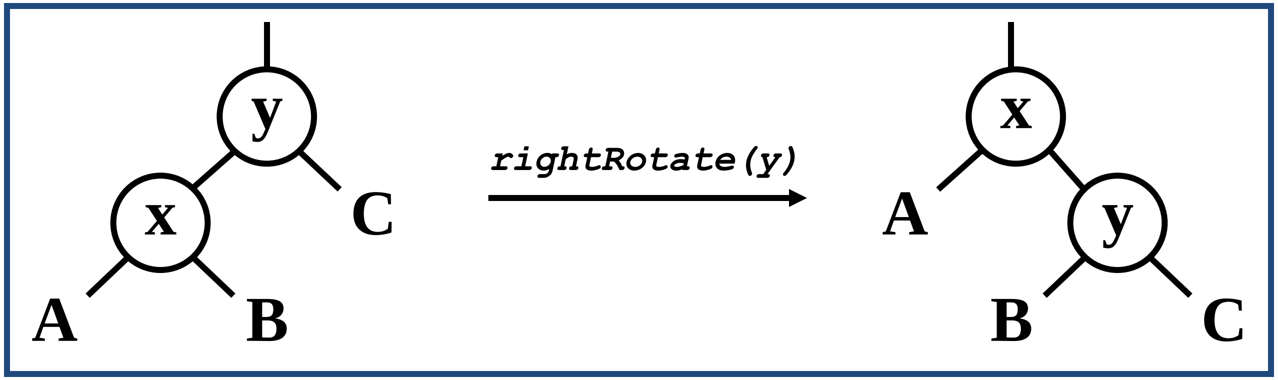
RB Trees: Rotation

- basic operation that changes tree structure is called *rotation*:



- *Rotation preserves inorder key ordering*

RB Trees: Rotation



- A lot of pointer manipulation
  - *x* keeps its left child
  - *y* keeps its right child
  - *x*'s right child becomes *y*'s left child
  - *x*'s and *y*'s parents change

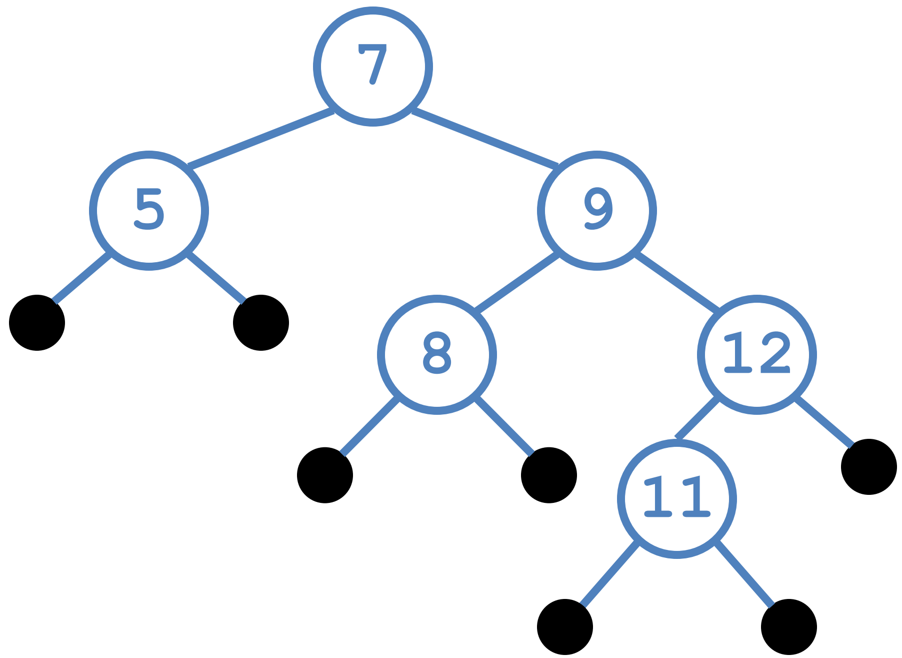
Left Rotate

LEFT-ROTATE(T, x)

```
y = x.right                      // set y
x.right = y.left                 // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                        // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                       // put x on y's left
x.p = y
```

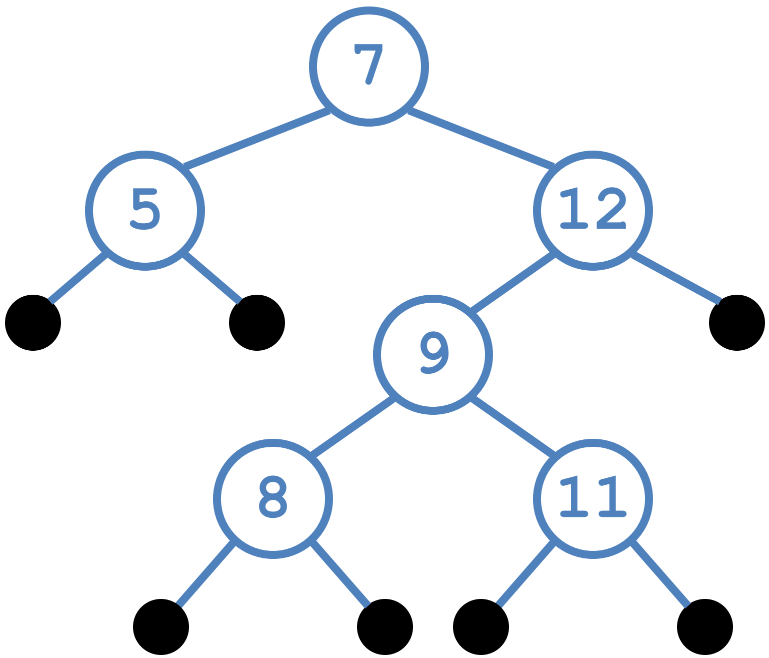
Rotation Example

- Rotate left about 9:



Rotation Example

- Rotate left about 9:



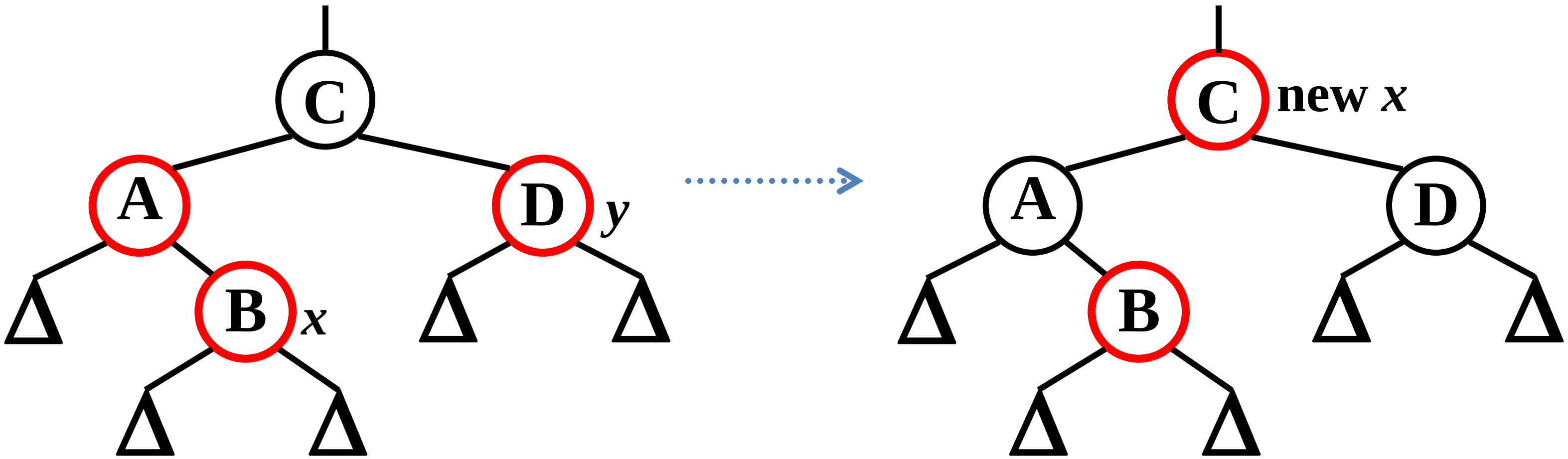
Red-Black Trees: Insertion

- Insertion: the basic idea
  - Insert *x* into tree, color *x* red
  - Only r-b property #3 might be violated (if x.p red)
    - If so, move violation up tree until a place is found where it can be fixed

RB Insert: Case 3II

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

- Case 1: “aunt” is red
- In figures below, all Δ’s are equal-black-height subtrees

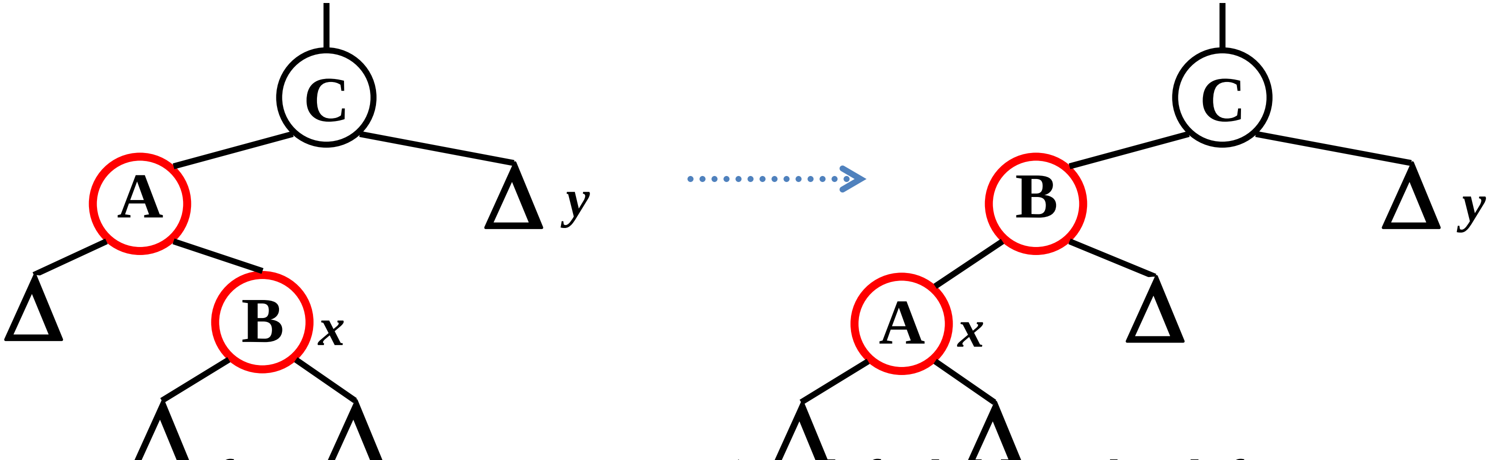


*Change colors of some nodes, preserving #4: all downward paths have equal **bh**.*
The while loop now continues with x’s grandparent as the new x

RB Insert: Case 3l

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

- Case 3lA:
  - “Aunt” is black
  - Node *x* is a right child
- Do a left-rotation

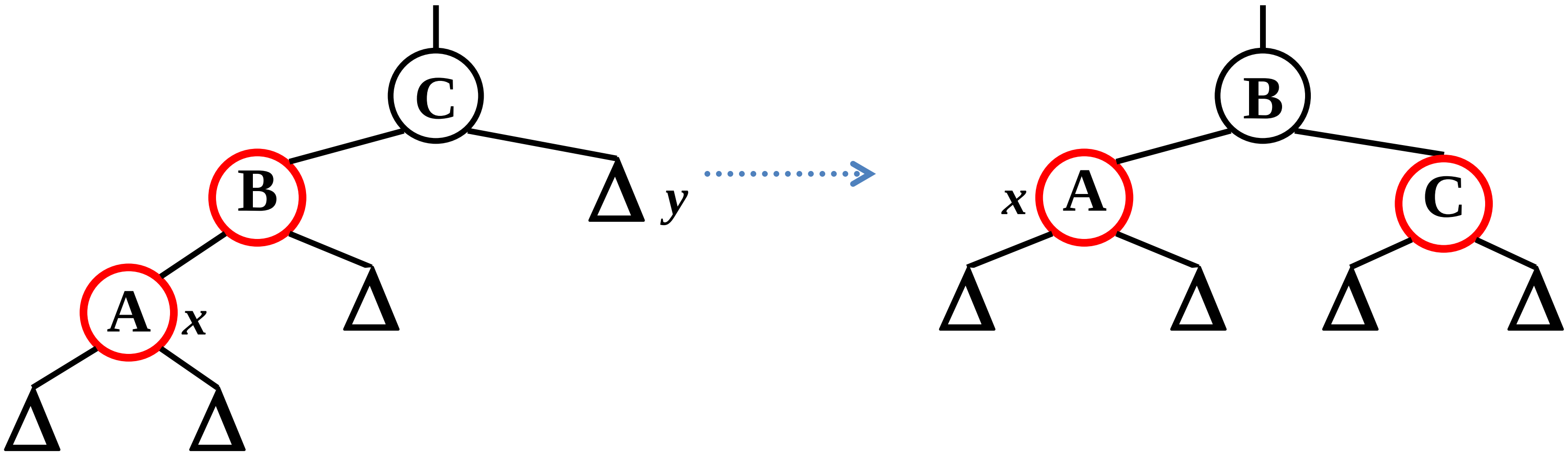


Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property# 4: all downward paths contain same number of black nodes

RB Insert: Case 3IC

```
x.p.color = BLACK;
x.p.p.color = RED;
rightRotate(x.p.p);
```

- Case 3IC:
  - "Aunt" is black
  - Node *x* is a left child
- Change colors; rotate right



Perform some color changes and do a right rotation

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

RB Insert: Rest of cases

- Previous cases hold if x's parent is a left child
- If x's parent is a right child, the cases are symmetric (swap left for right)