



# Red Black Tree

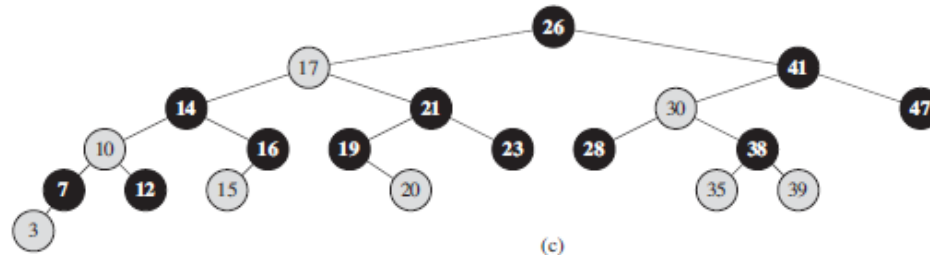
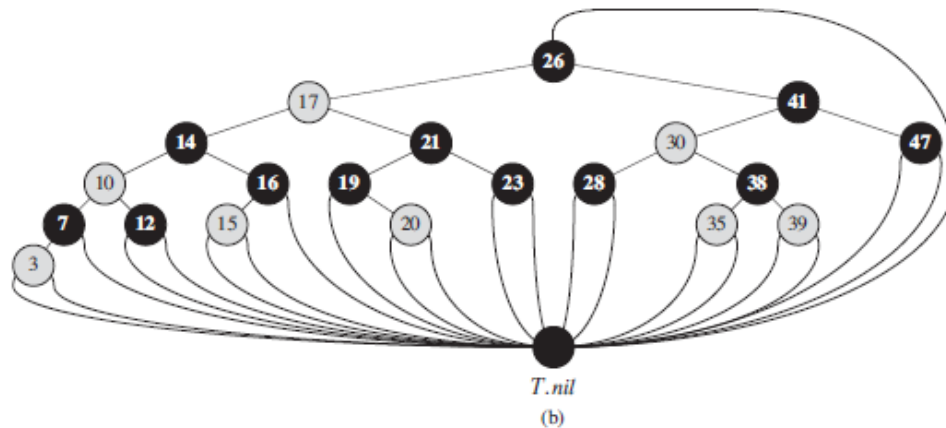
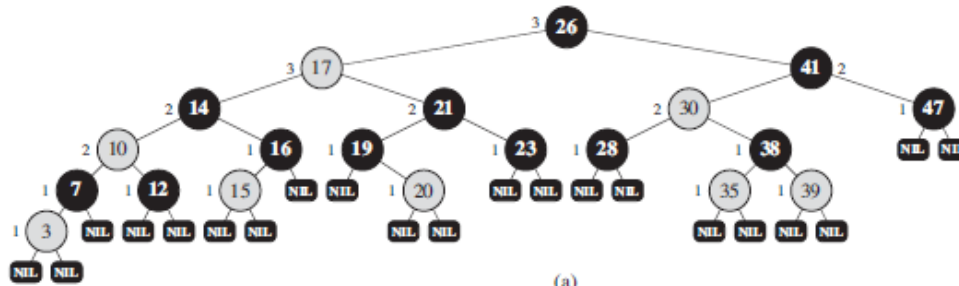
Chapter 13 from Textbook

# Red-Black Trees

---

- *Red-black trees:*
  - Binary search trees augmented with node color
  - Operations designed to guarantee that the height  $h = O(\lg n)$
- First: describe the properties of red-black trees
- Then: prove that these guarantee  $h = O(\lg n)$
- Finally: describe operations on red-black trees

# Red-Black Trees



# Red-Black Properties

- The *red-black properties*:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
    - Note: this means every “real” node has 2 children
  3. If a node is red, both children are black
    - Note: can't have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

# Red-Black Trees

- Put example on board and verify properties:
  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 of a node  $x$* : # black nodes on path to leaf not including  $x$ 
  - Label example with  $h$  and  $bh$  values

# Height of Red-Black Trees

- *What is the minimum black-height of a node with height  $h$ ?*
- A: a height- $h$  node has black-height  $\geq h/2$
- Theorem: A red-black tree with  $n$  internal nodes has height  $h \leq 2 \lg(n + 1)$
- *How do you suppose we'll prove this?*

# RB Trees: Proving Height Bound

- Prove:  $n$ -node RB tree has height  $h \leq 2 \lg(n+1)$
- Claim: A subtree rooted at a node  $x$  contains at least  $2^{bh(x)} - 1$  internal nodes
  - Proof by induction on height  $h$
  - Base step:  $x$  has height 0 (i.e., NULL leaf node)
    - *What is  $bh(x)$ ?*

# RB Trees: Proving Height Bound

- Prove:  $n$ -node RB tree has height  $h \leq 2 \lg(n+1)$
- Claim: A subtree rooted at a node  $x$  contains at least  $2^{bh(x)} - 1$  internal nodes
  - Proof by induction on height  $h$
  - Base step:  $x$  has height 0 (i.e., NULL leaf node)
    - *What is  $bh(x)$ ?*
    - A: 0
    - So...subtree contains  $2^{bh(x)} - 1$   
 $= 2^0 - 1$   
 $= 0$  internal nodes (TRUE)



# RB Trees: Proving Height Bound

- Inductive proof that subtree at node  $x$  contains at least  $2^{\text{bh}(x)} - 1$  internal nodes
  - Inductive step:  $x$  has positive height and 2 children
    - Each child has black-height of  $\text{bh}(x)$  or  $\text{bh}(x)-1$  (*Why?*)
    - The height of a child = (height of  $x$ ) - 1
    - So the subtrees rooted at each child contain at least  $2^{\text{bh}(x)-1} - 1$  internal nodes
    - Thus subtree at  $x$  contains
$$(2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1$$
$$= 2 \cdot 2^{\text{bh}(x)-1} - 1 = 2^{\text{bh}(x)} - 1 \text{ nodes}$$

# RB Trees: Proving Height Bound

- Thus at the root of the red-black tree:

$$n \geq 2^{\text{bh}(\text{root})} - 1 \quad (\text{Why?})$$

$$n \geq 2^{h/2} - 1 \quad (\text{Why?})$$

$$\lg(n+1) \geq h/2 \quad (\text{Why?})$$

$$h \leq 2 \lg(n+1) \quad (\text{Why?})$$

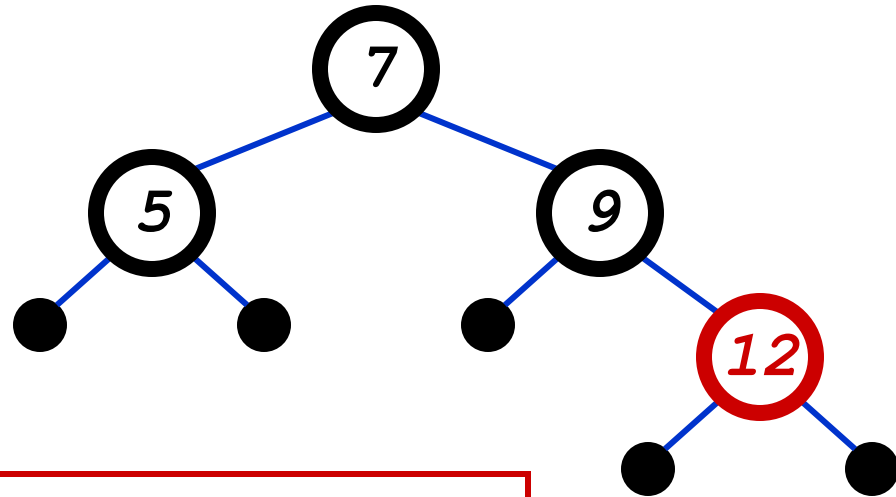
Thus  $h = O(\lg n)$

# RB Trees: Worst-Case Time

- So we've proved that a red-black tree has  $O(\lg n)$  height
- Corollary: These operations take  $O(\lg n)$  time:
  - Minimum(), Maximum()
  - Successor(), Predecessor()
  - Search()
- Insert() and Delete():
  - Will also take  $O(\lg n)$  time
  - But will need special care since they modify tree

# Red-Black Trees: An Example

- *Color this tree:*

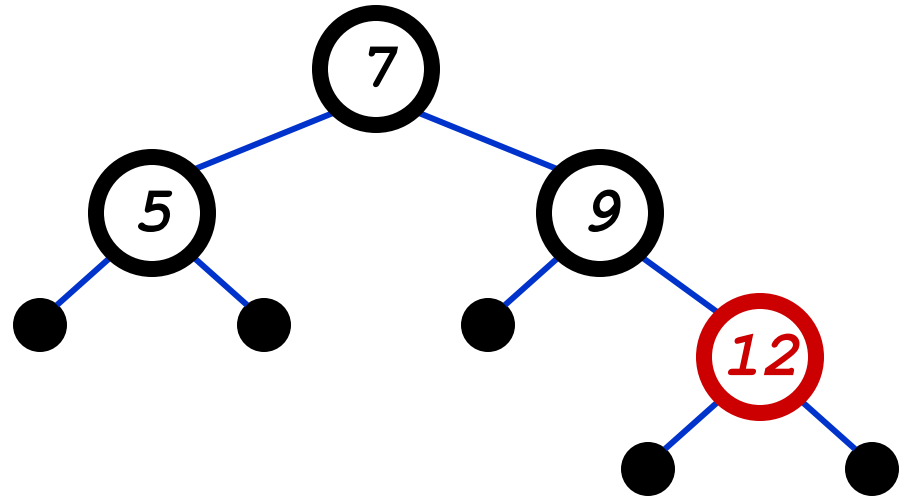


Red-black properties:

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

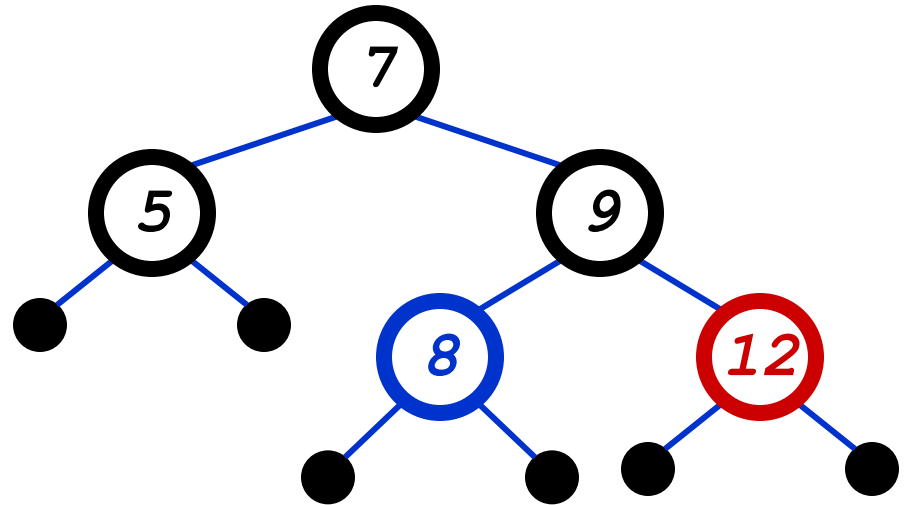
- Insert 8
  - *Where does it go?*



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

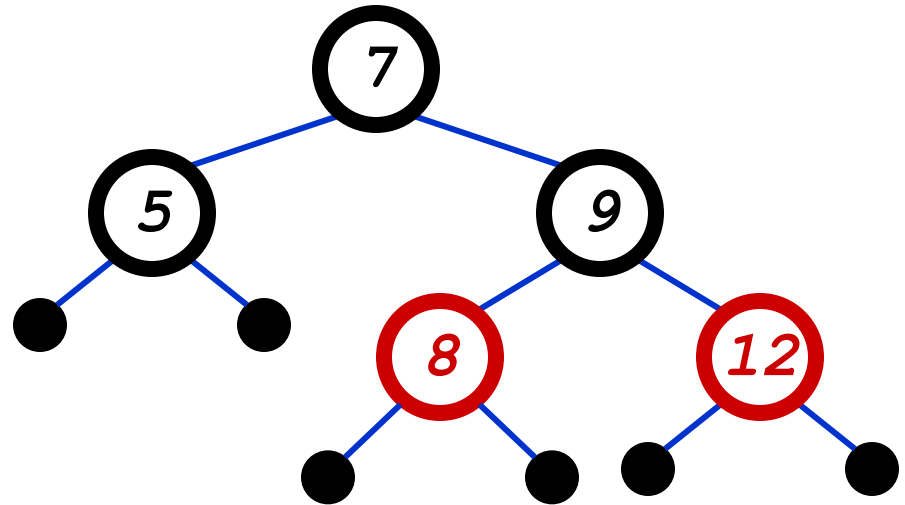
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



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

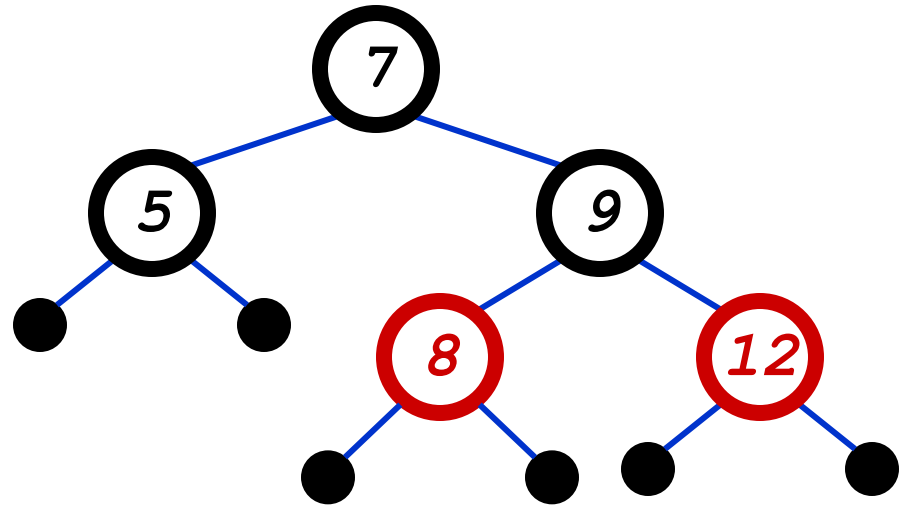
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



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?*

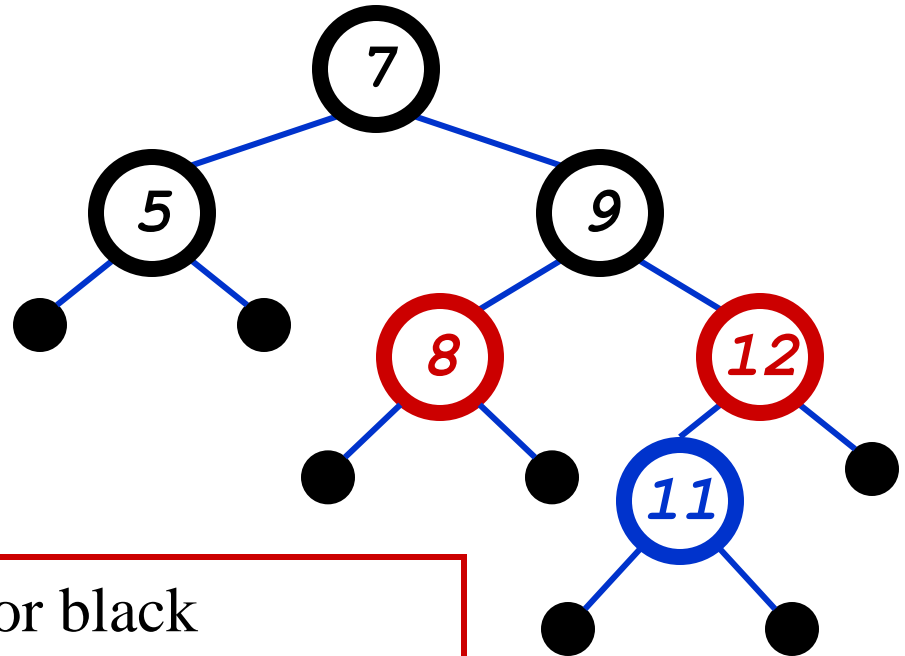


1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black



# Red-Black Trees: The Problem With Insertion

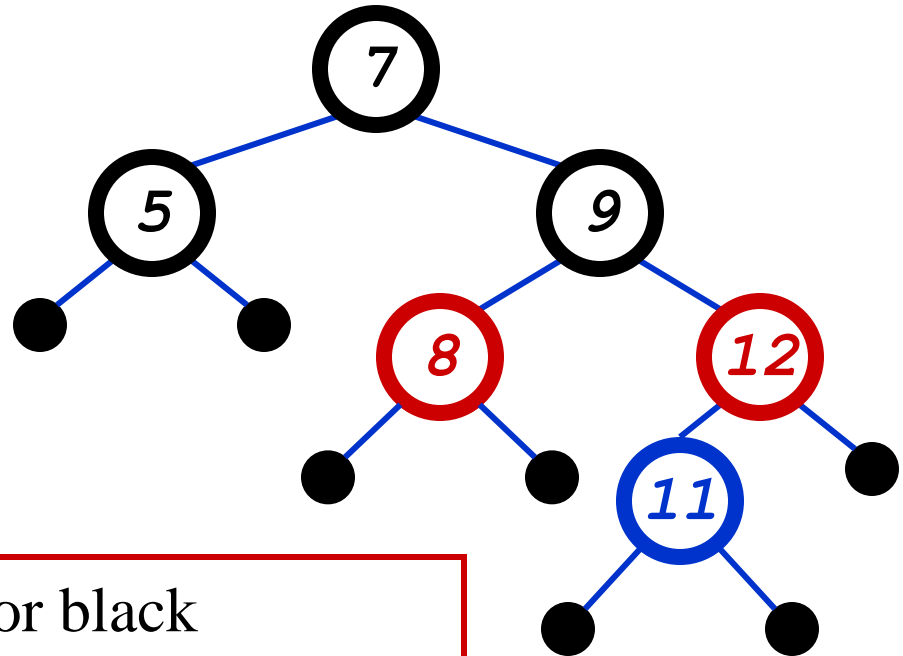
- Insert 11
  - *Where does it go?*
  - *What color?*



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

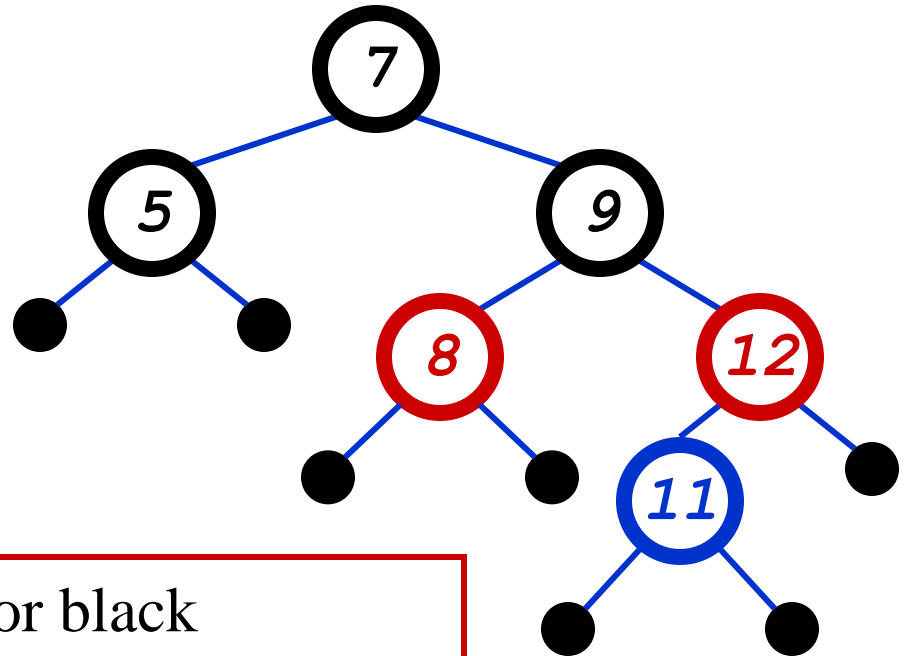
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)



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

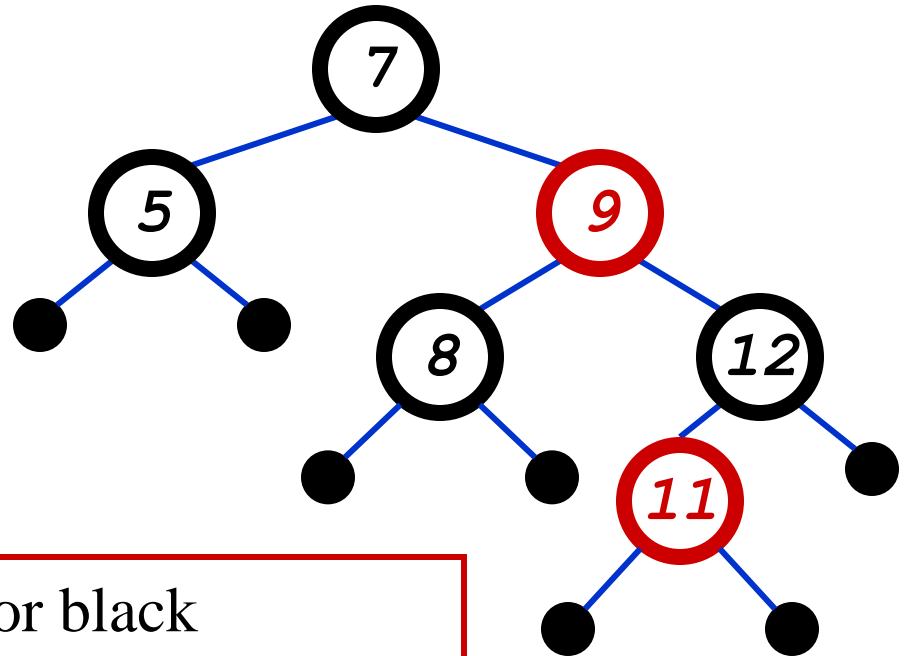
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)
    - Can't be black! (#4)



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

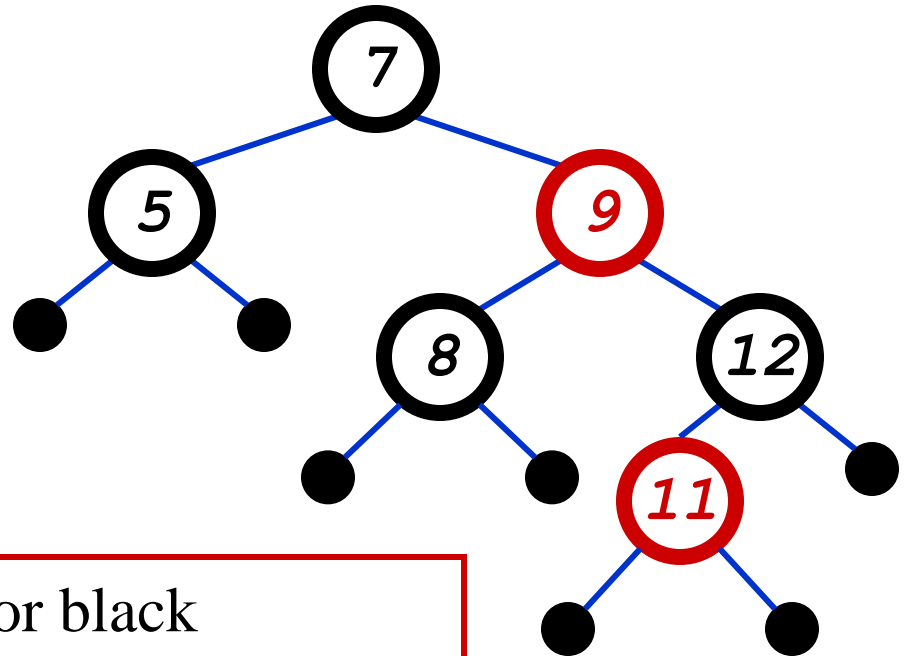
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Solution:  
recolor the tree



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

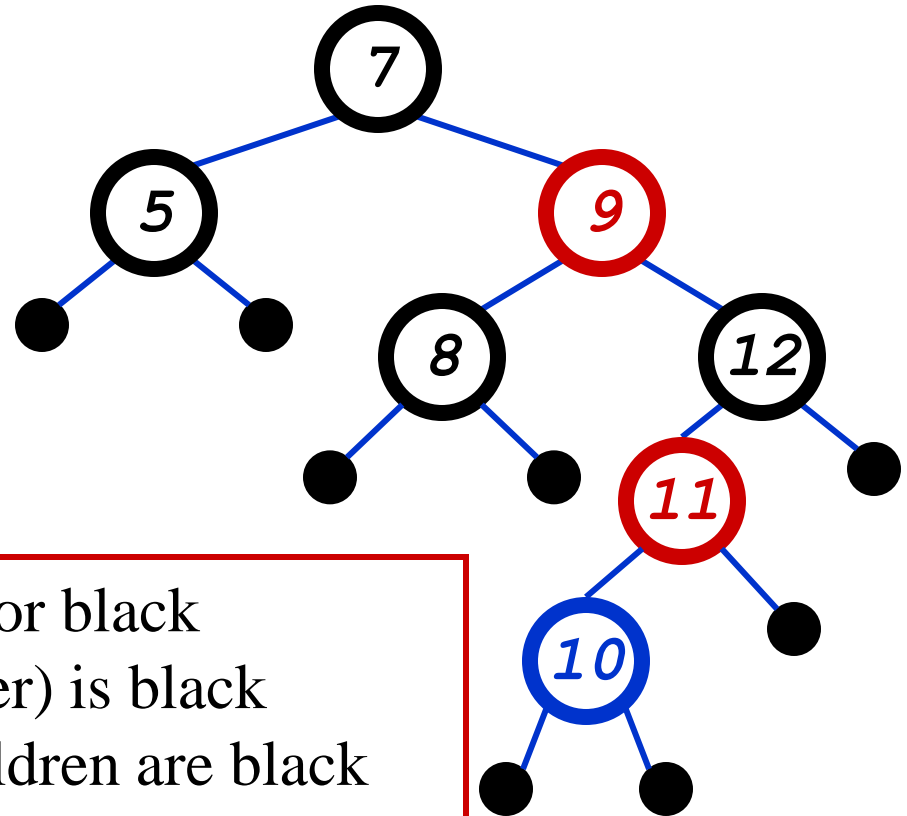
- Insert 10
  - *Where does it go?*



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

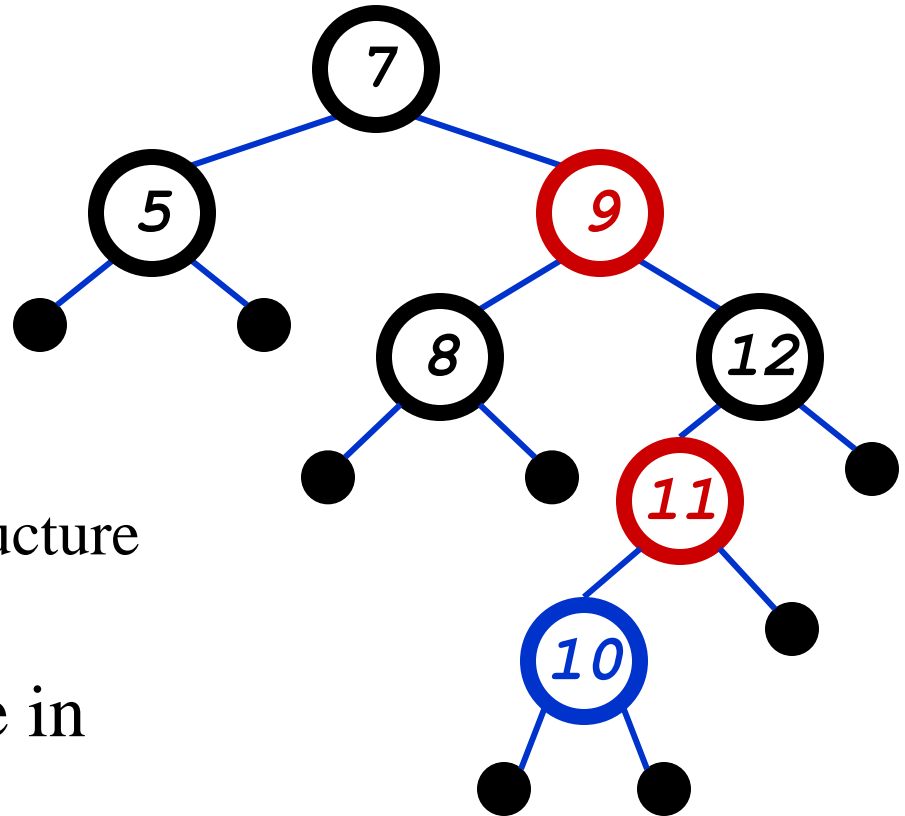
- Insert 10
  - *Where does it go?*
  - *What color?*



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

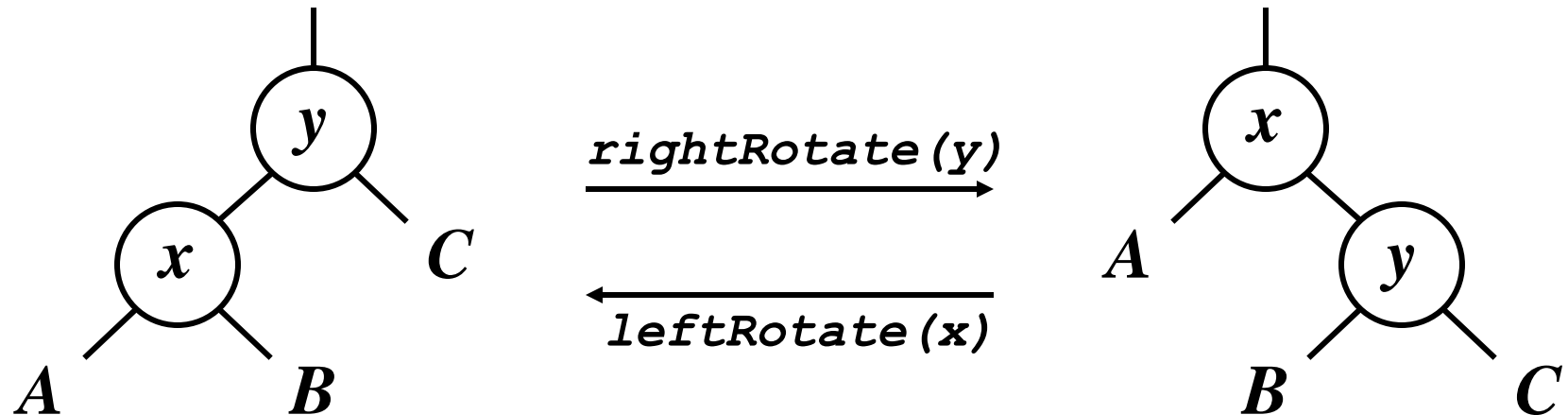
# Red-Black Trees: The Problem With Insertion

- Insert 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

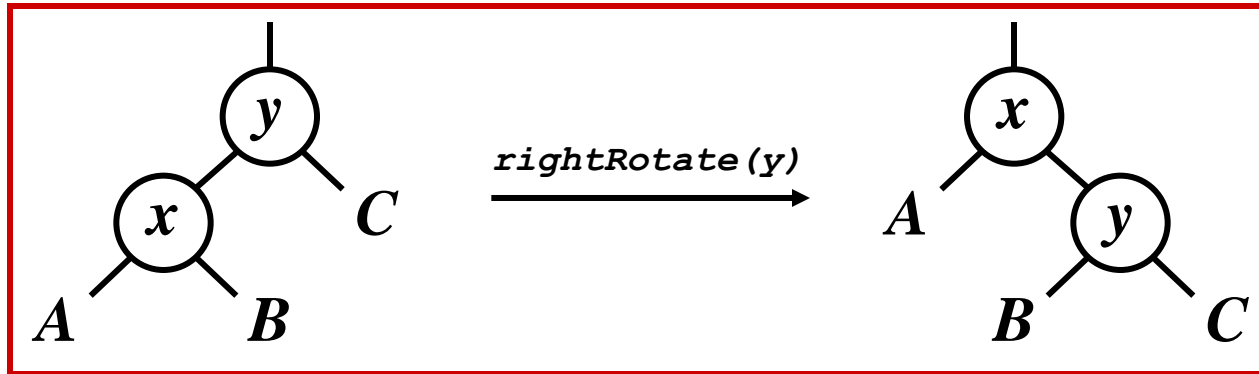
- Our basic operation for changing tree structure is called *rotation*:



- *Does rotation preserve inorder key ordering?*
- *What would the code for **`rightRotate()`** actually do?*



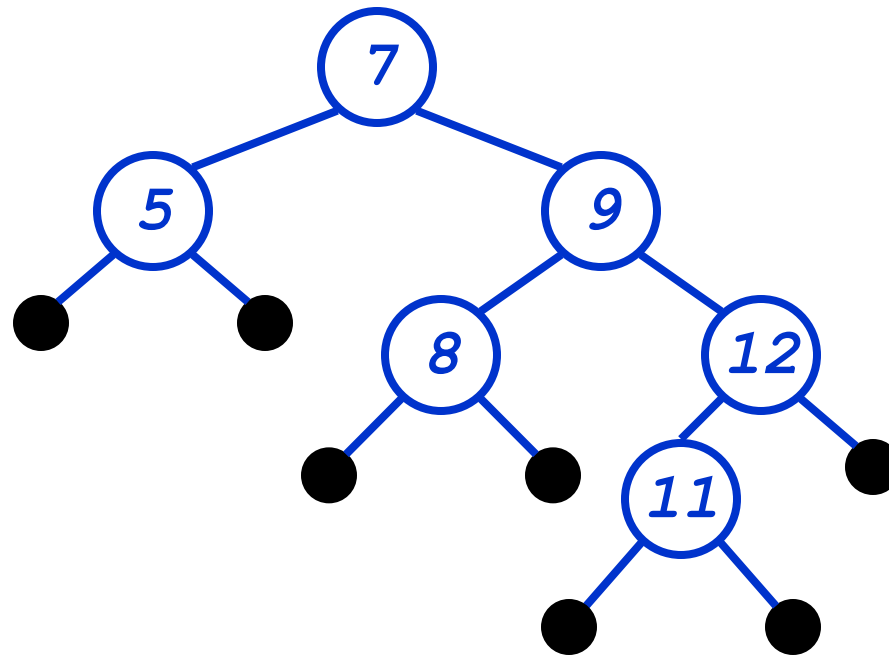
# RB Trees: Rotation



- Answer: 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
- What is the running time?
  - $O(1)$  (only pointer manipulation)

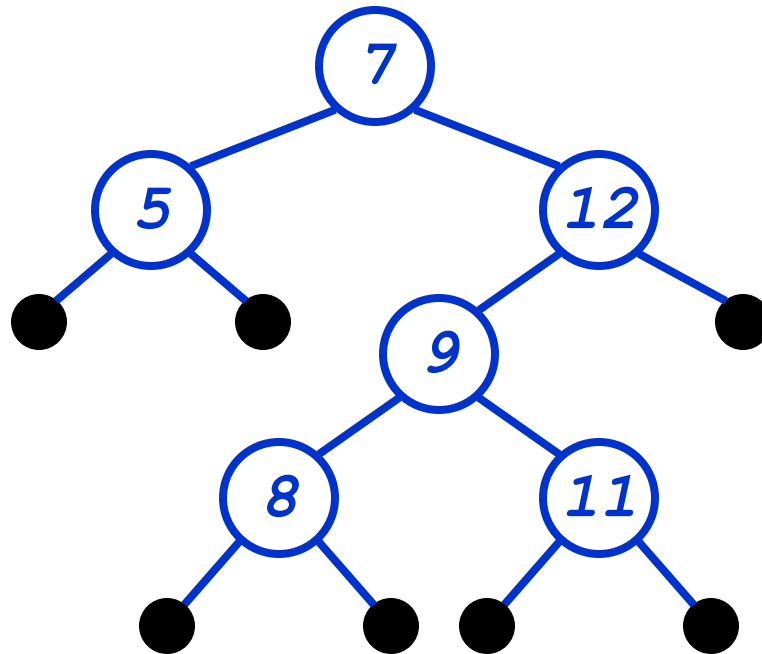
# Rotation Example

- 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  $p[x]$  red)
    - If so, move violation up tree until a place is found where it can be fixed
  - Total time will be  $O(\lg n)$

## rbInsert(x)

```
treeInsert(x);
```

```
x->color = RED;
```

```
// Move violation of #3 up tree, maintaining #4 as invariant:
```

```
while (x!=root && x->p->color == RED)
```

```
if (x->p == x->p->p->left)
```

```
    y = x->p->p->right;
```

```
    if (y->color == RED)
```

```
        x->p->color = BLACK;
```

```
        y->color = BLACK;
```

```
        x->p->p->color = RED;
```

```
        x = x->p->p;
```

} Case 1

```
else // y->color == BLACK
```

```
    if (x == x->p->right)
```

```
        x = x->p;
```

```
        leftRotate(x);
```

```
        x->p->color = BLACK;
```

```
        x->p->p->color = RED;
```

```
        rightRotate(x->p->p);
```

} Case 2

} Case 3

```
else // x->p == x->p->p->right
```

```
    (same as above, but with
```

```
    "right" & "left" exchanged)
```

## rbInsert(x)

```
treeInsert(x);
```

```
x->color = RED;
```

```
// Move violation of #3 up tree, maintaining #4 as invariant:
```

```
while (x!=root && x->p->color == RED)
```

```
if (x->p == x->p->p->left)
```

```
    y = x->p->p->right;
```

```
    if (y->color == RED)
```

```
        x->p->color = BLACK;
```

```
        y->color = BLACK;
```

```
        x->p->p->color = RED;
```

```
        x = x->p->p;
```

} Case 1: uncle is RED

```
else // y->color == BLACK
```

```
    if (x == x->p->right)
```

```
        x = x->p;
```

```
        leftRotate(x);
```

```
        x->p->color = BLACK;
```

```
        x->p->p->color = RED;
```

```
        rightRotate(x->p->p);
```

} Case 2

} Case 3

```
else // x->p == x->p->p->right
```

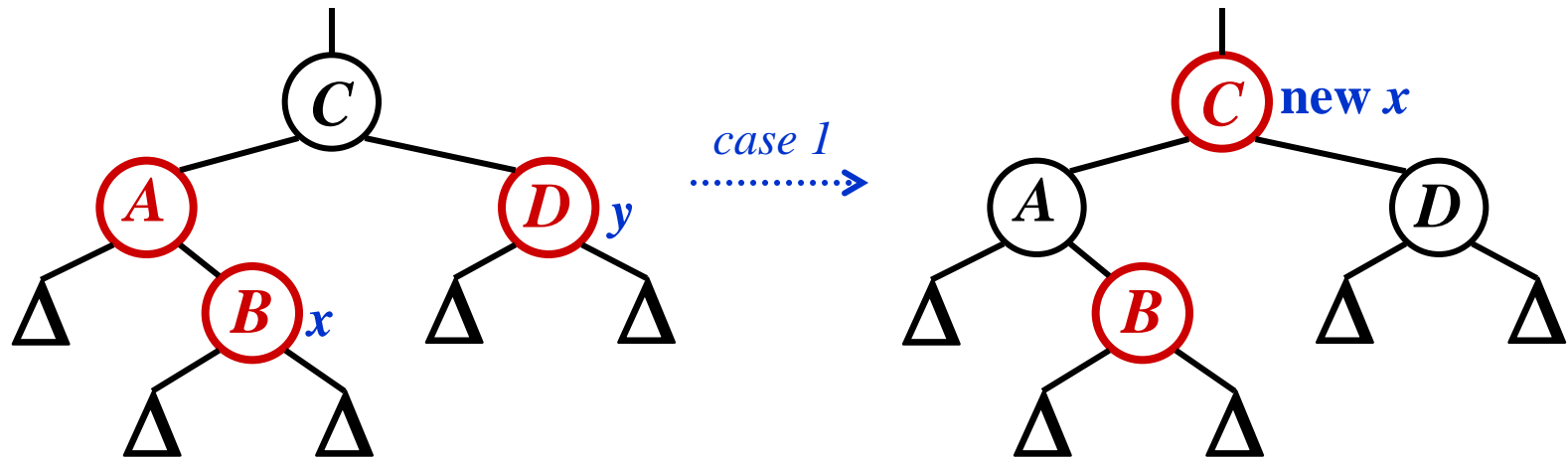
```
    (same as above, but with
```

```
    "right" & "left" exchanged)
```

# RB Insert: Case 1

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

- Case 1: “uncle” is red
- In figures below, all  $\Delta$ 's are equal-black-height subtrees

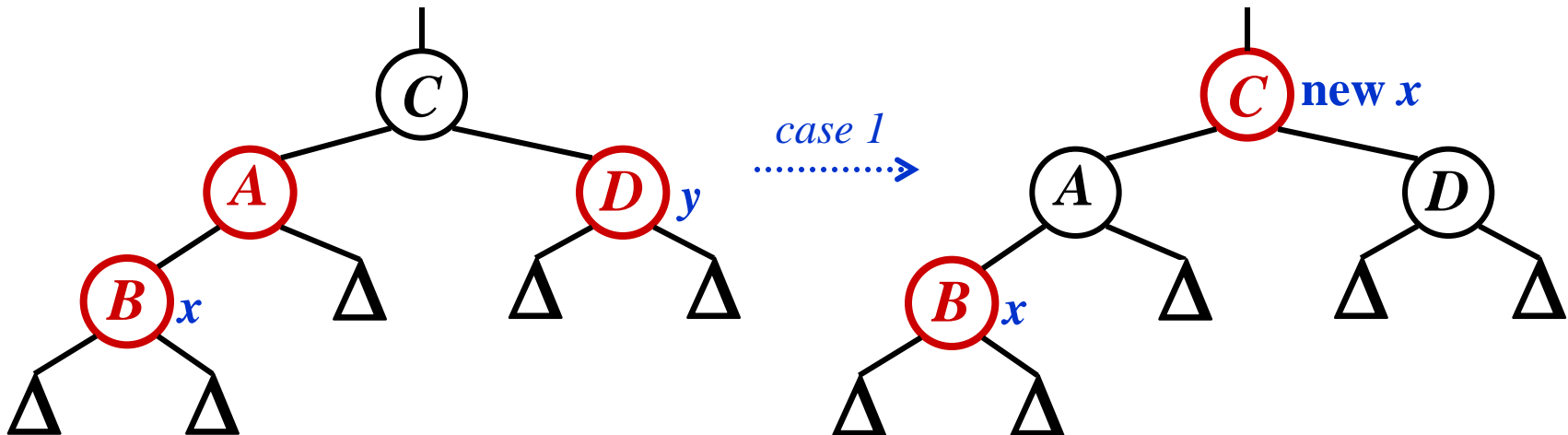


*Change colors of some nodes, preserving #4: all downward paths have equal b.h.  
The while loop now continues with  $x$ 's grandparent as the new  $x$*

# RB Insert: Case 1

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

- Case 1: “uncle” is red
- In figures below, all  $\Delta$ 's are equal-black-height subtrees



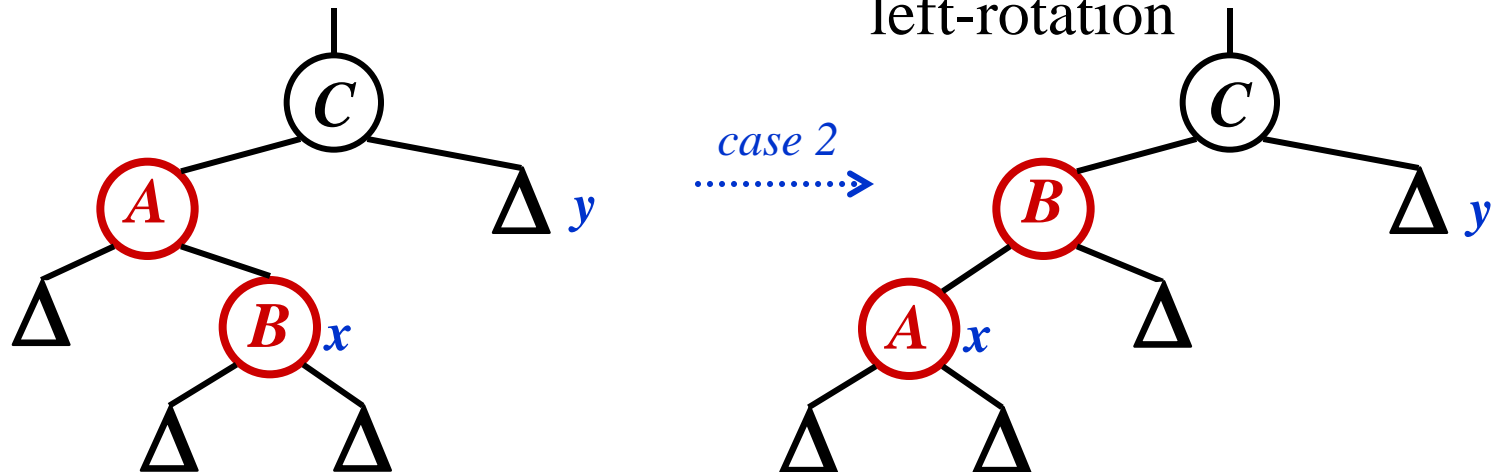
*Same action whether  $x$  is a left or a right child*



# RB Insert: Case 2

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

- Case 2:
  - “Uncle” is black
  - Node  $x$  is a right child
- Transform to case 3 via a left-rotation

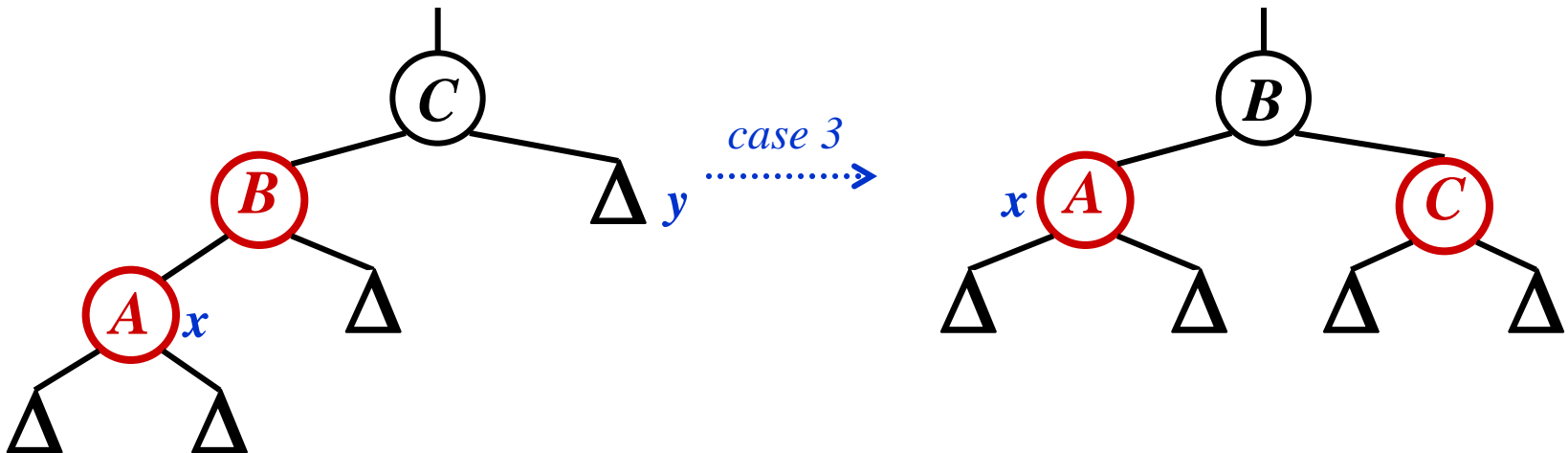


*Transform case 2 into case 3 ( $x$  is left child) with a left rotation  
This preserves property 4: all downward paths contain same number of black nodes*

# RB Insert: Case 3

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

- Case 3:
  - “Uncle” 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: Cases 4-6

---

- Cases 1-3 hold if  $x$ 's parent is a left child
- If  $x$ 's parent is a right child, cases 4-6 are symmetric (swap left for right)

# Red-Black Trees: Deletion

---

- And you thought insertion was tricky...
- We will not cover RB delete in class
  - You should read section 13.4 on your own
  - Read for the overall picture, not the details