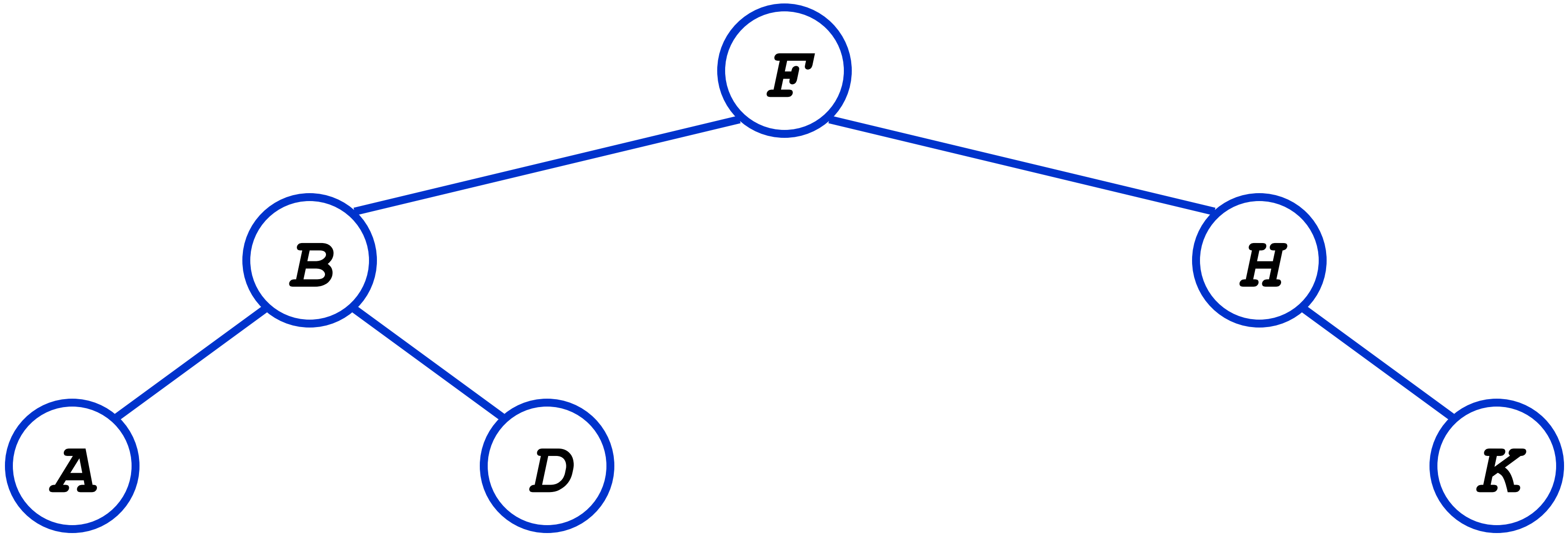
Red-Black Trees

Review: Binary Search Trees

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, elements have:
  - *key*: an identifying field inducing a total ordering
  - *left*: pointer to a left child (may be NULL)
  - *right*: pointer to a right child (may be NULL)
  - *p*: pointer to a parent node (NULL for root)

Review: Binary Search Trees

- BST property:
  $key[left(x)] \leq key[x] \leq key[right(x)]$
- Example:



Review: Inorder Tree Walk

- An *inorder walk* prints the set in sorted order:

```
TreeWalk(x)
    TreeWalk(left[x]);
    print(x);
    TreeWalk(right[x]);
```

  - Easy to show by induction on the BST property
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root

BST Search

```
TreeSearch(x, k)
    if (x = NULL  or  k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x], k);
    else
        return TreeSearch(right[x], k);
```

BST Search (Iterative)

```
IterativeTreeSearch(x, k)
    while (x != NULL  and  k != key[x])
        if (k < key[x])
            x = left[x];
        else
            x = right[x];
    return x;
```

BST Insert

- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a "trailing pointer" to keep track of where you came from (like inserting into singly linked list)
- Like search, takes time O(h), h = tree height

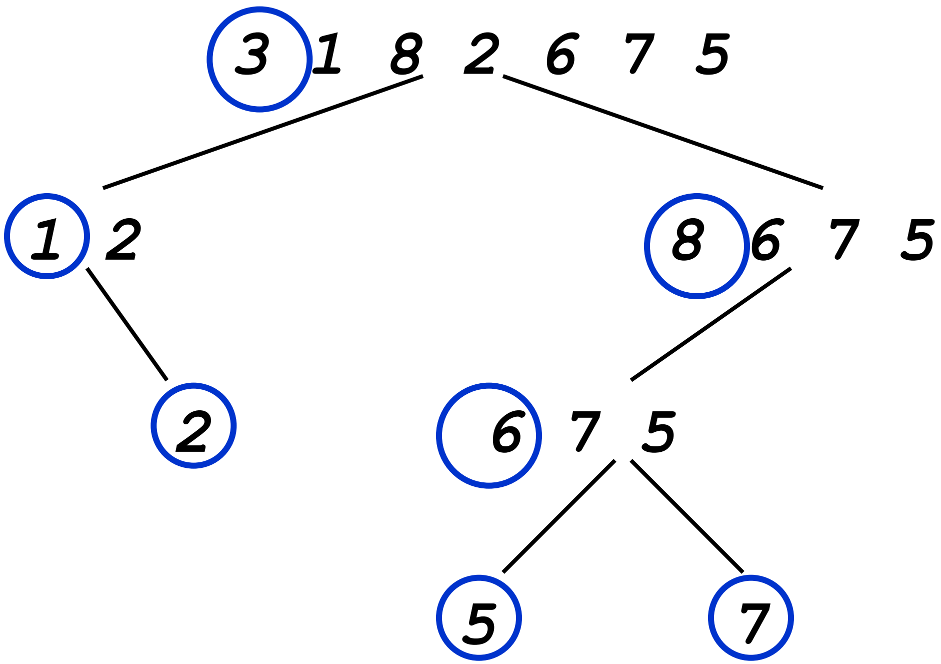
Sorting With BSTs

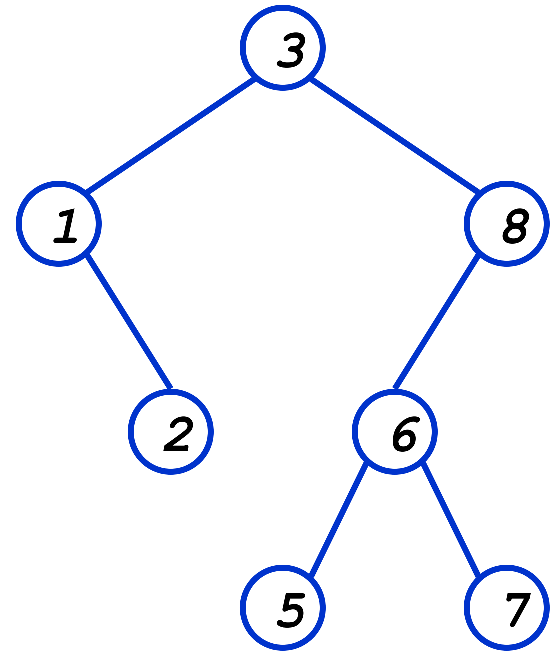
- Basic algorithm:
  - Insert elements of unsorted array from 1..n
  - Do an inorder tree walk to print in sorted order
- Running time:
  - Worst case: ?
  - Average case: ? (it's a quicksort!)

Sorting With BSTs

- Average case analysis
  - It's a form of quicksort!

```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```





Search, Minimum and Maximum

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ key[x]
     do if k < key[x]
           then x ← left[x]
           else x ← right[x]
return x
```

```
TREE-MINIMUM(x)
while left[x] ≠ NIL
     do x ← left[x]
return x
```

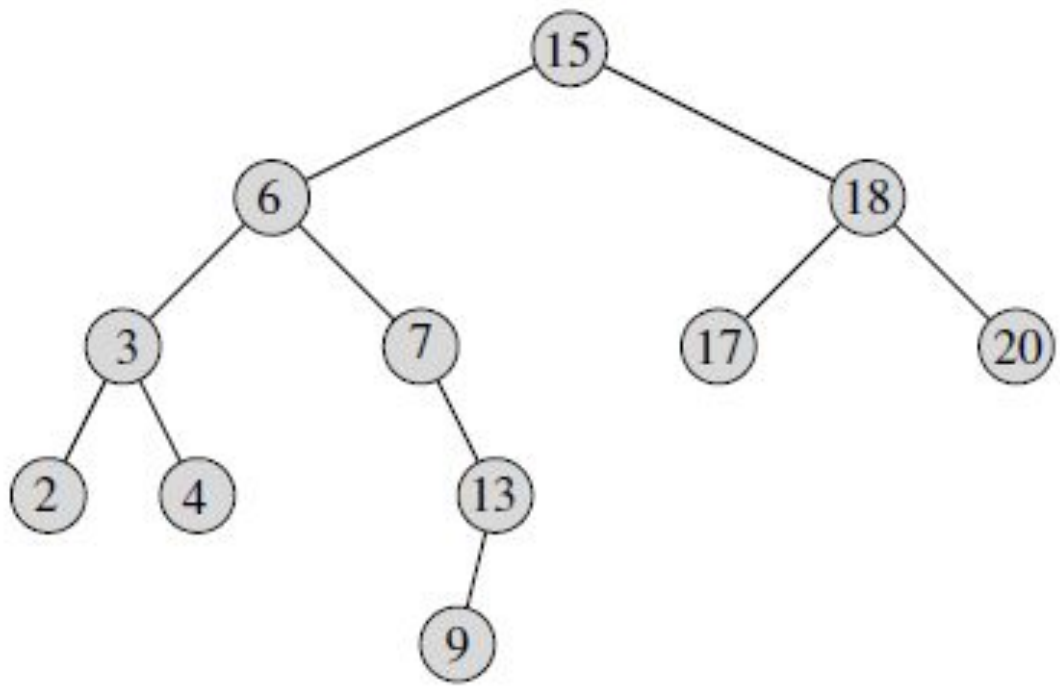
```
TREE-MAXIMUM(x)
while right[x] ≠ NIL
     do x ← right[x]
return x
```

More BST Operations

- Minimum:
  - Find leftmost node in tree
- Successor:
  - x has a right subtree: successor is minimum node in right subtree
  - x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    - Intuition: As long as you move to the left up the tree, you’re visiting smaller nodes.
- Predecessor: similar to successor

More BST Operations

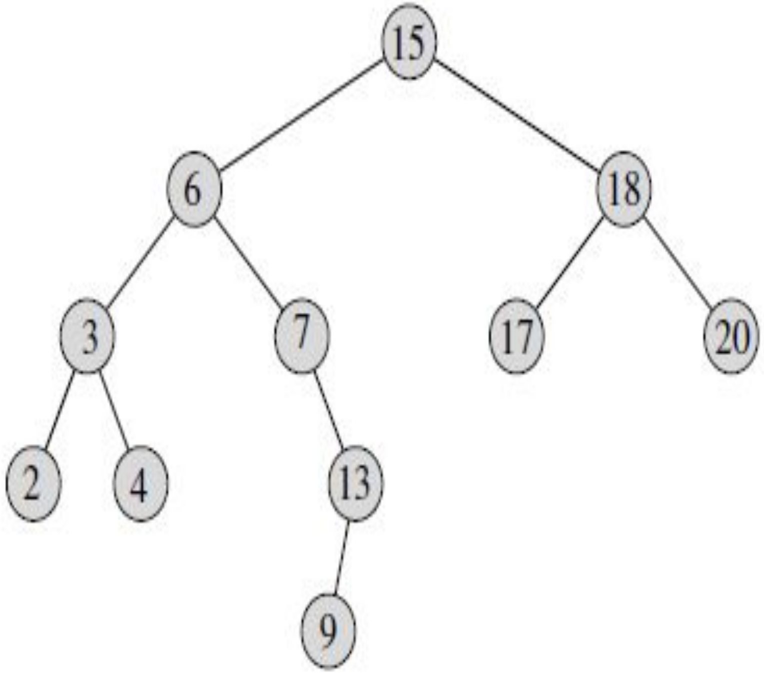
- x has a right subtree: successor is minimum node in right subtree
- x has no right subtree: successor is closest ancestor of x whose left child is also ancestor (or self) of x



Successor

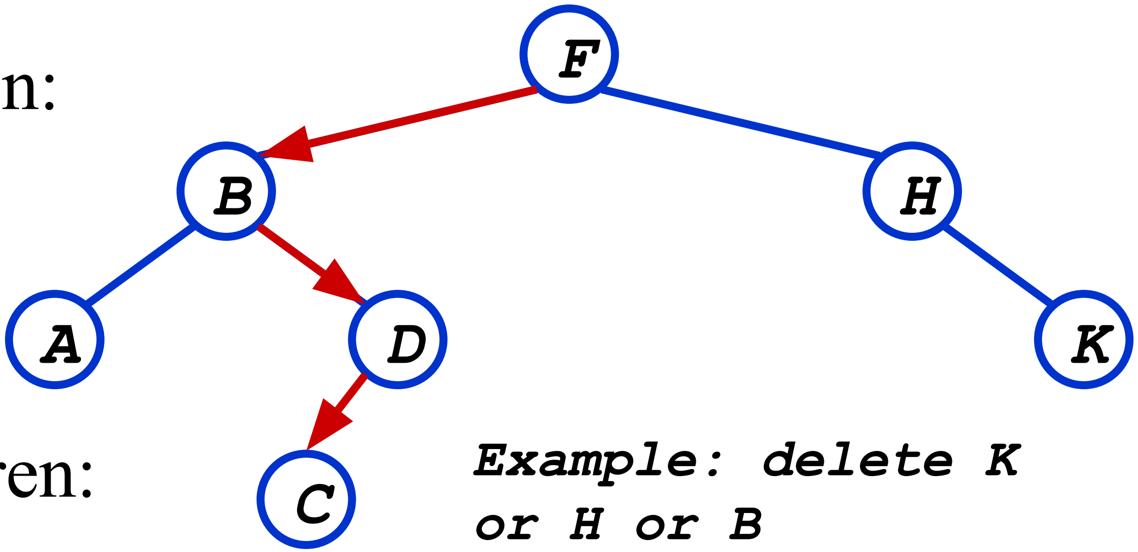
- x has a right subtree: successor is minimum node in right subtree
- x has no right subtree: successor is closest ancestor of x whose left child is also ancestor (or self) of x

```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
    then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
    do x ← y
       y ← p[y]
return y
```



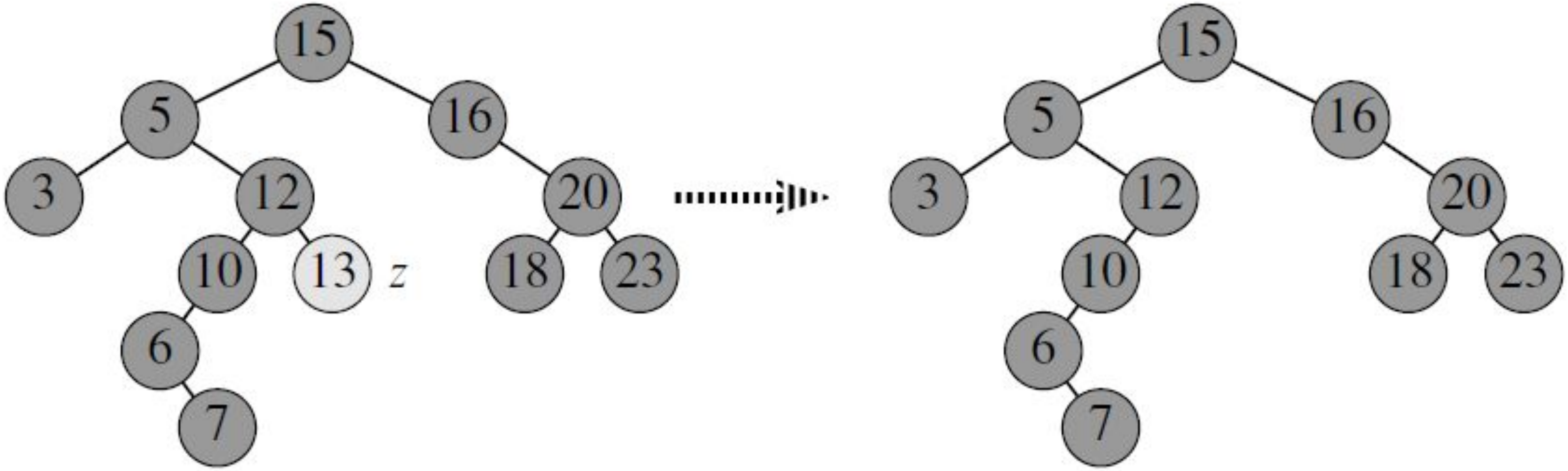
More BST Operations

- Delete:
  - x has no children:
    - Remove x
  - x has one child:
    - Splice out x
  - x has two children:
    - Swap x with successor
    - Perform case 1 or 2 to delete it

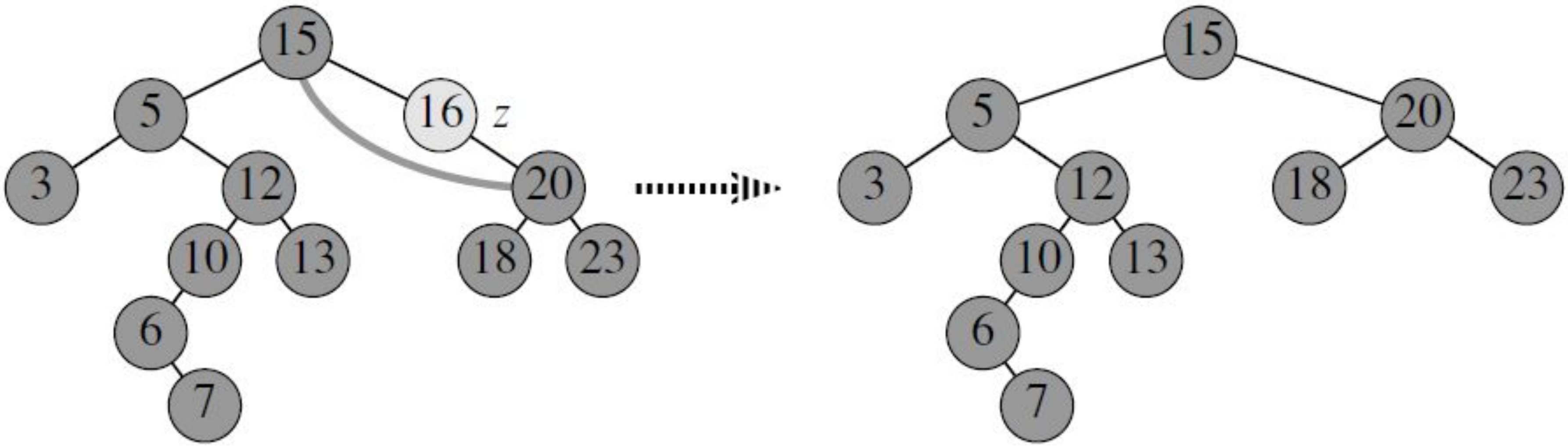


Example: delete K or H or B

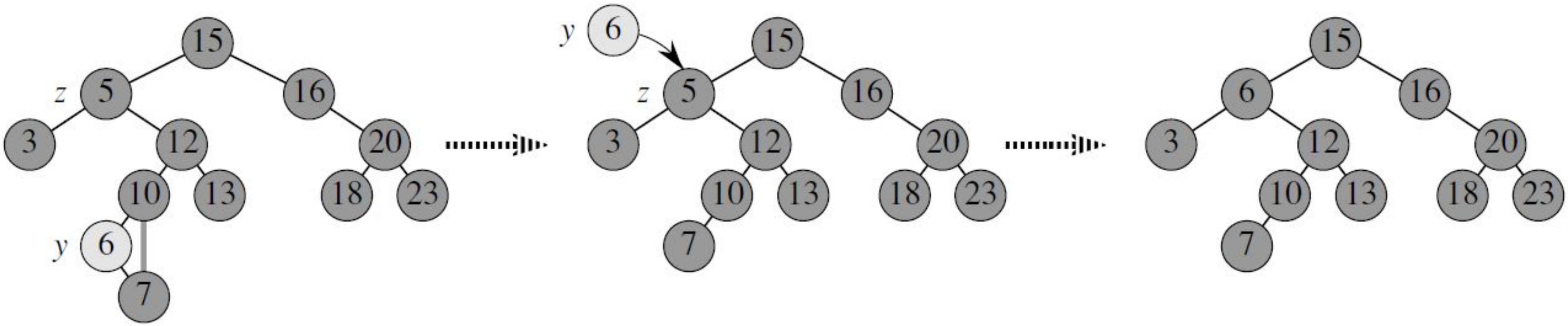
Delete



Delete



Delete


15
z
5
16
3
12
20
10
13
18
23
y
6
7
y
6
15
z
5
16
3
12
20
10
13
18
23
7
15
6
16
3
12
20
10
13
18
23
7

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

Each node of the tree now contains the fields *color, key, left, right*, and *p*

Red-Black Properties

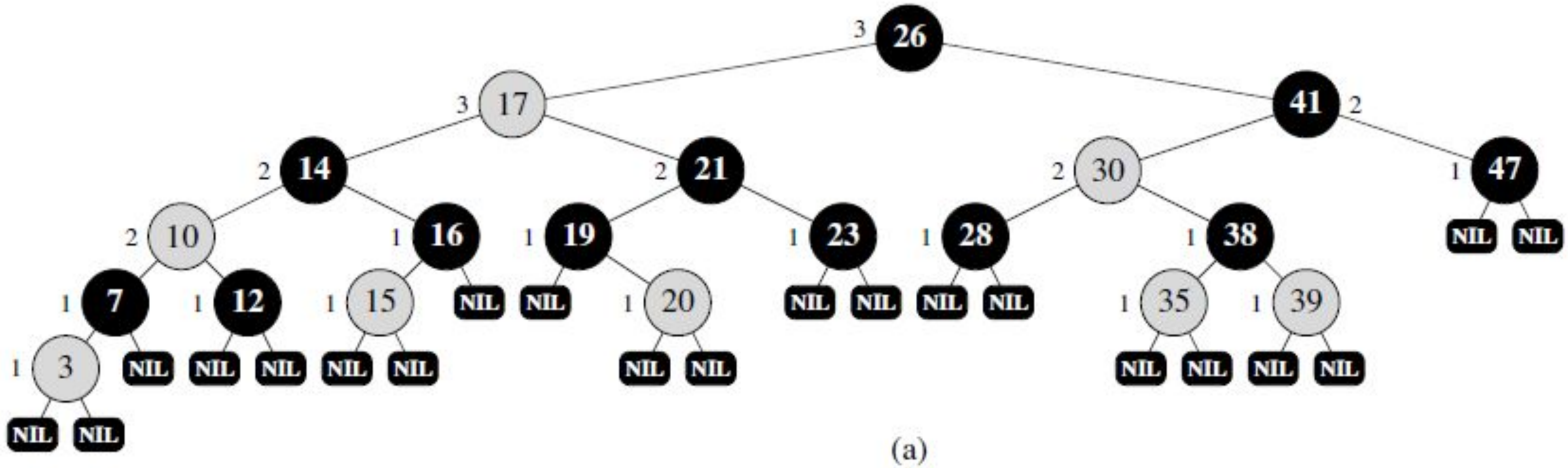
- The *red-black properties*:
  1. Every node is either red or black
  2. The root is always black
  3. Every leaf (orig. NULL) is black
     - Note: this means every "real" node has 2 children
  4. If a node is red, both children are black
     - Note: can't have 2 consecutive reds on a path
  5. Every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

Red-Black Trees

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

- *black-height:* # black nodes on path to leaf
  - Label example with *h* and bh values

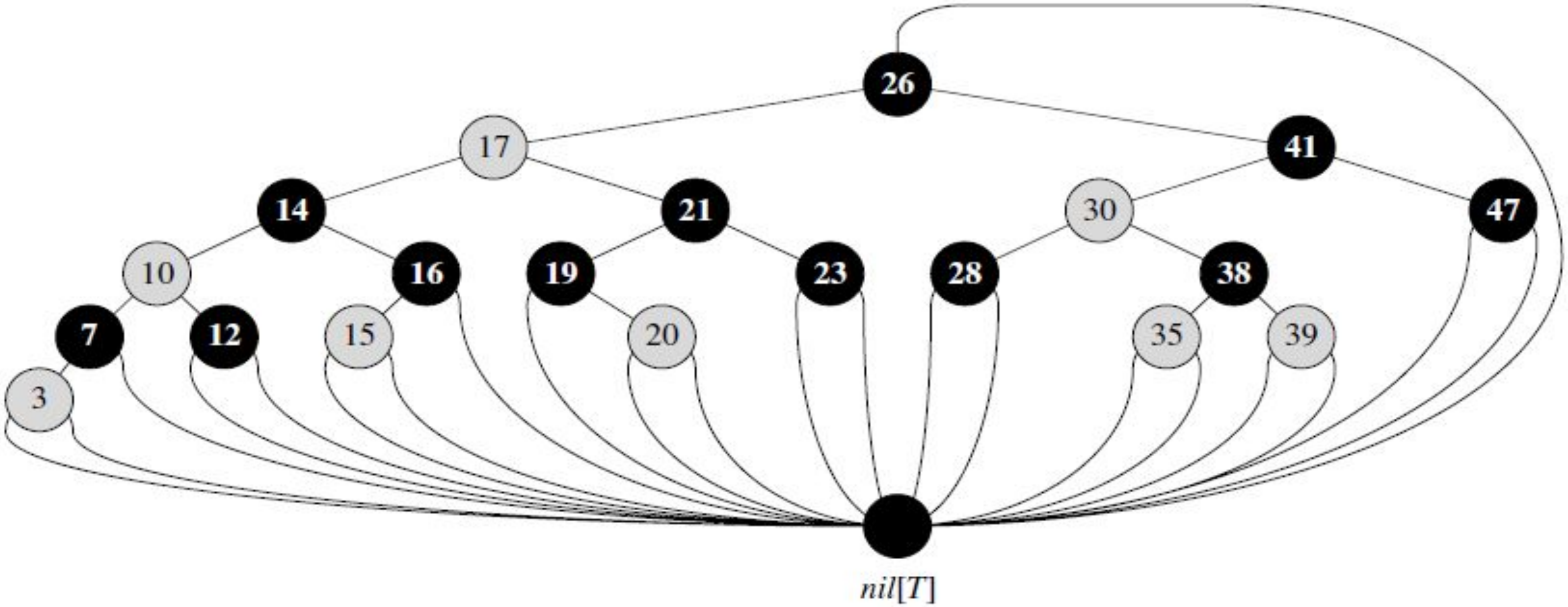
Example red-black tree



(a)

Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL's have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *nil*[*T*], which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

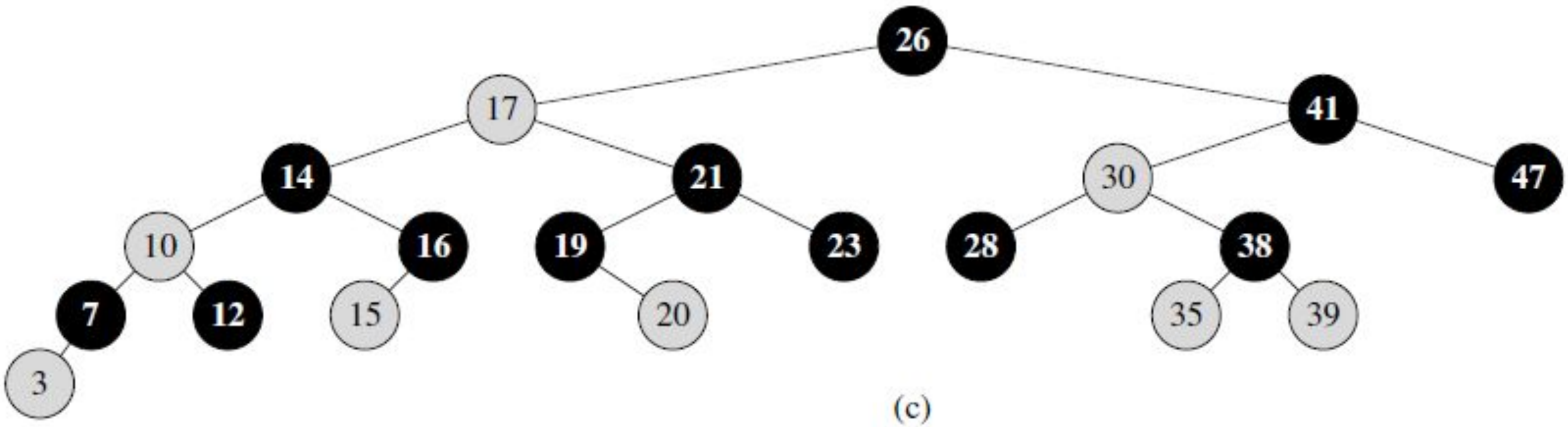
Example red-black tree



(b)

Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL's have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *nil*[*T*], which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

Example red-black tree



(c)

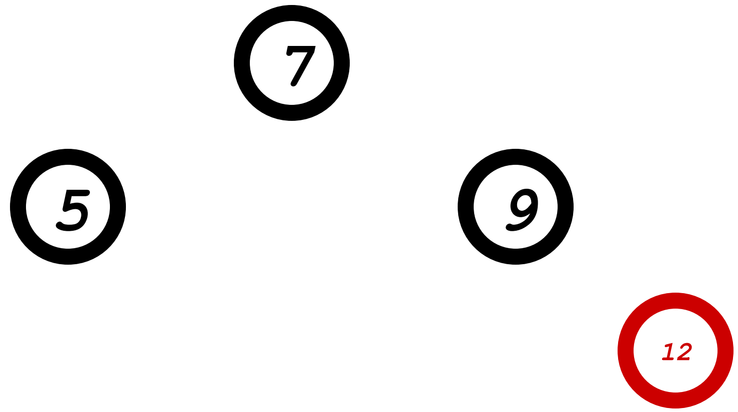
Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL's have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *nil*[T], which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

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

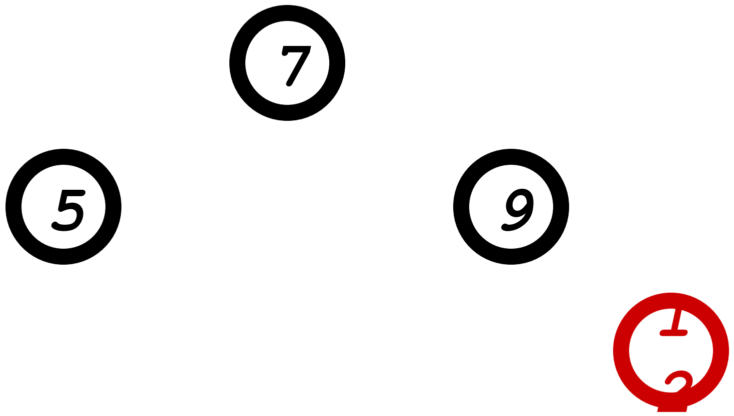
- *Color this tree:*



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

Red-Black Trees: The Problem With Insertion

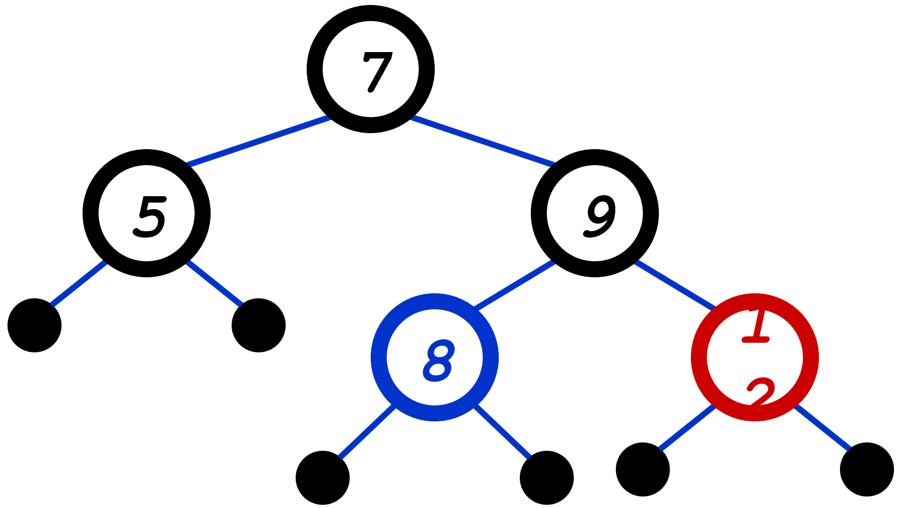
- Insert 8
  - *Where does it go?*



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

Red-Black Trees: The Problem With Insertion

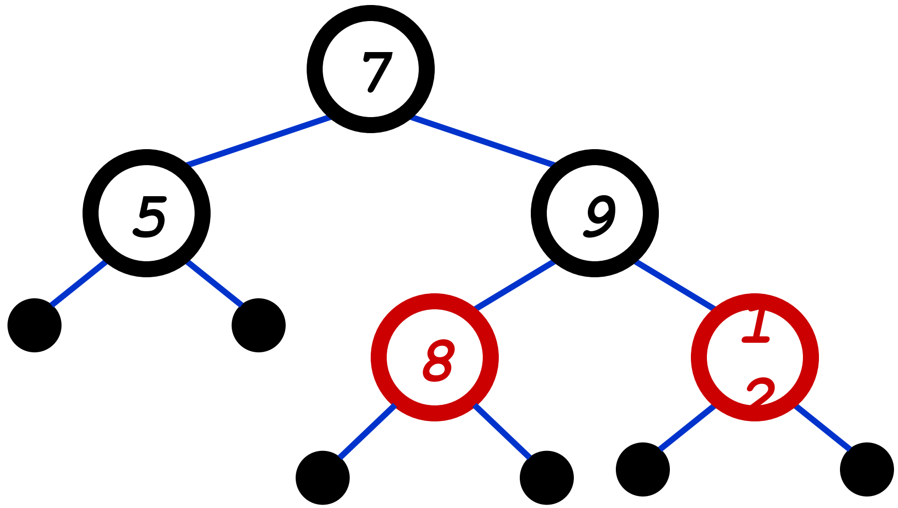
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



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

Red-Black Trees: The Problem With Insertion

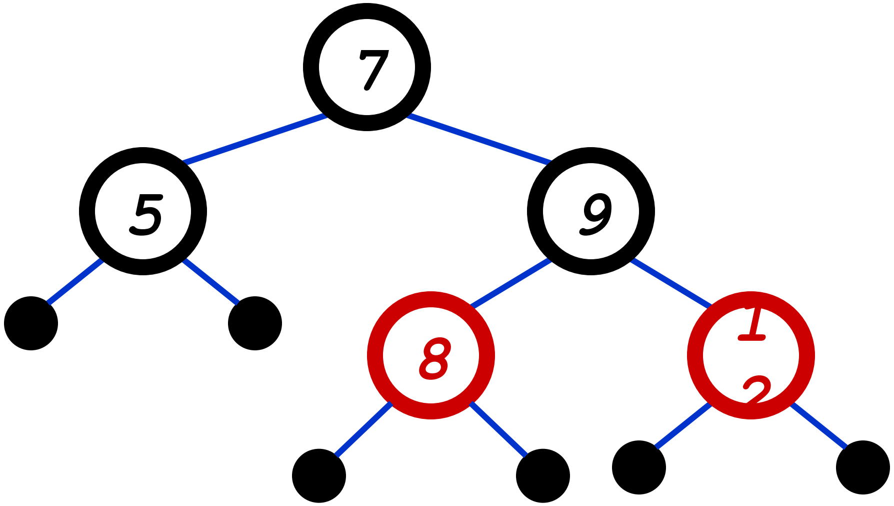
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



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

Red-Black Trees: The Problem With Insertion

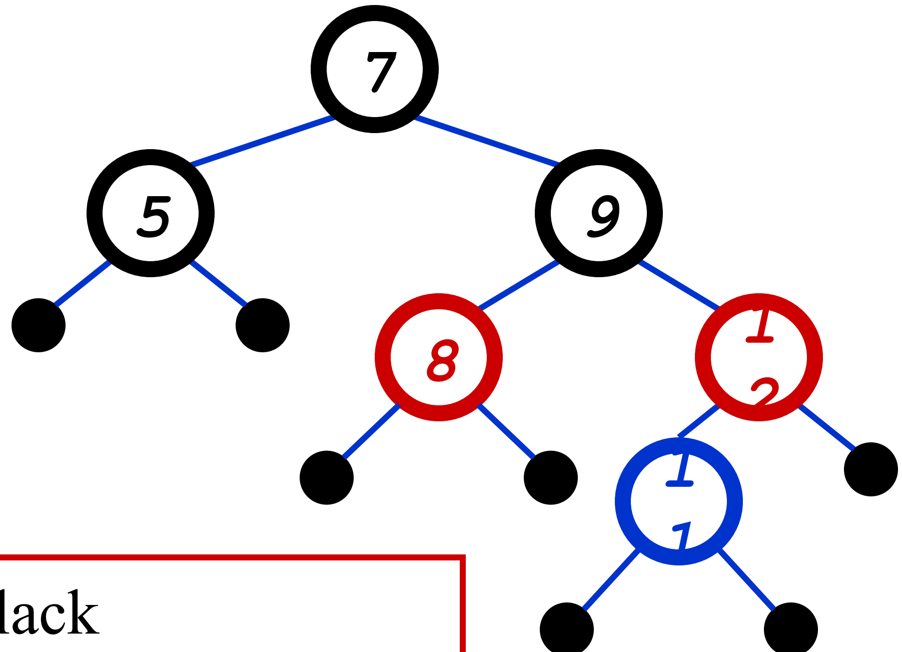
- Insert 11
  - *Where does it go?*



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

Red-Black Trees: The Problem With Insertion

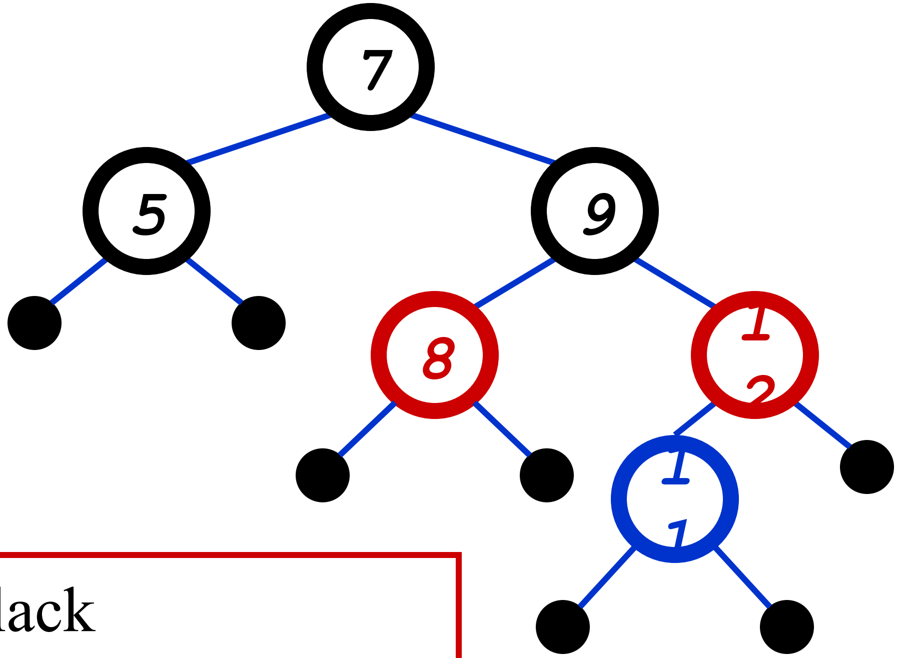
- Insert 11
  - *Where does it go?*
  - *What color?*



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

Red-Black Trees: The Problem With Insertion

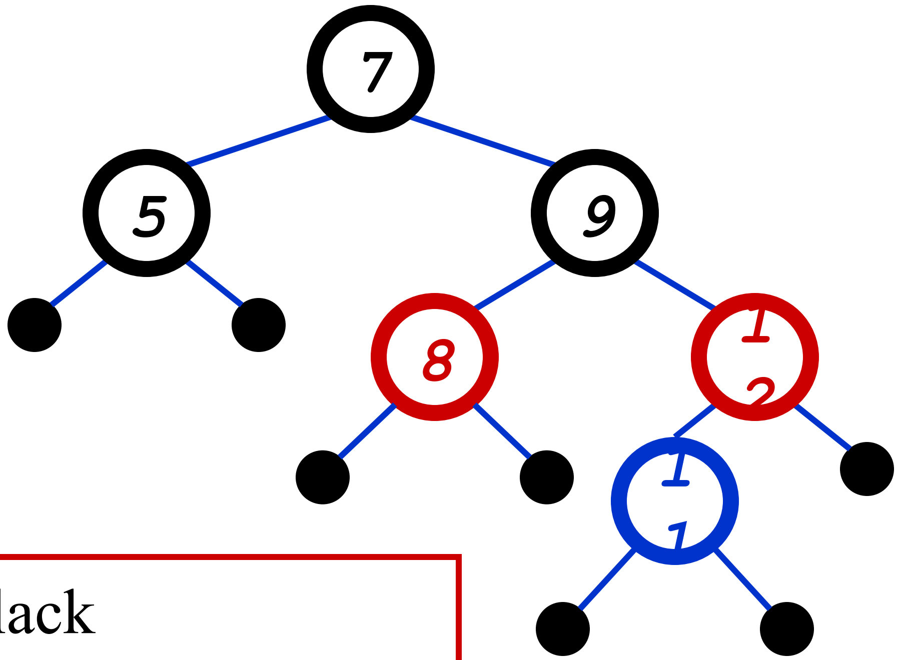
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can’t be red! (#4)



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

Red-Black Trees: The Problem With Insertion

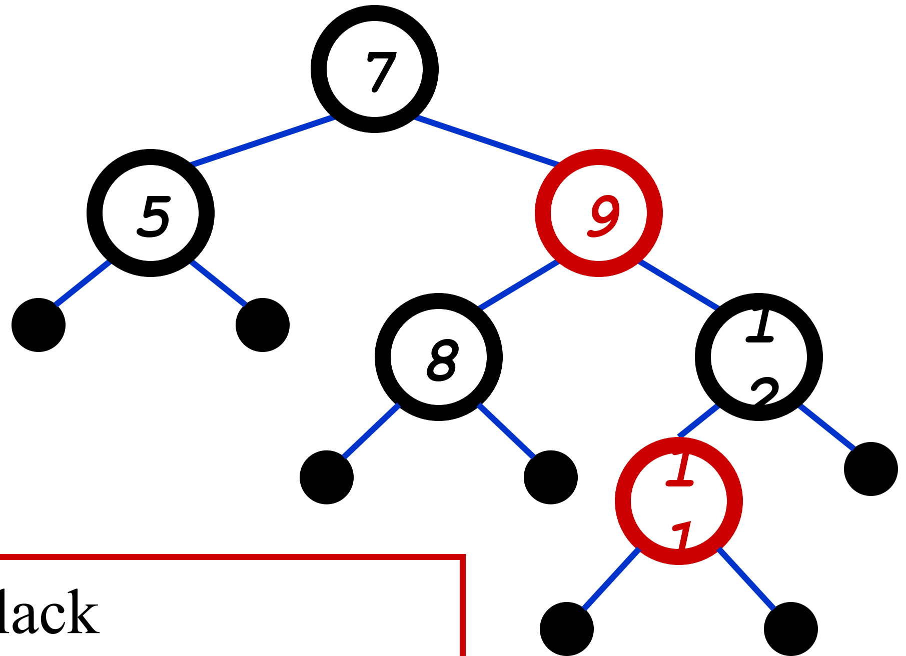
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can’t be red! (#4)
    - Can’t be black! (#5)



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

Red-Black Trees: The Problem With Insertion

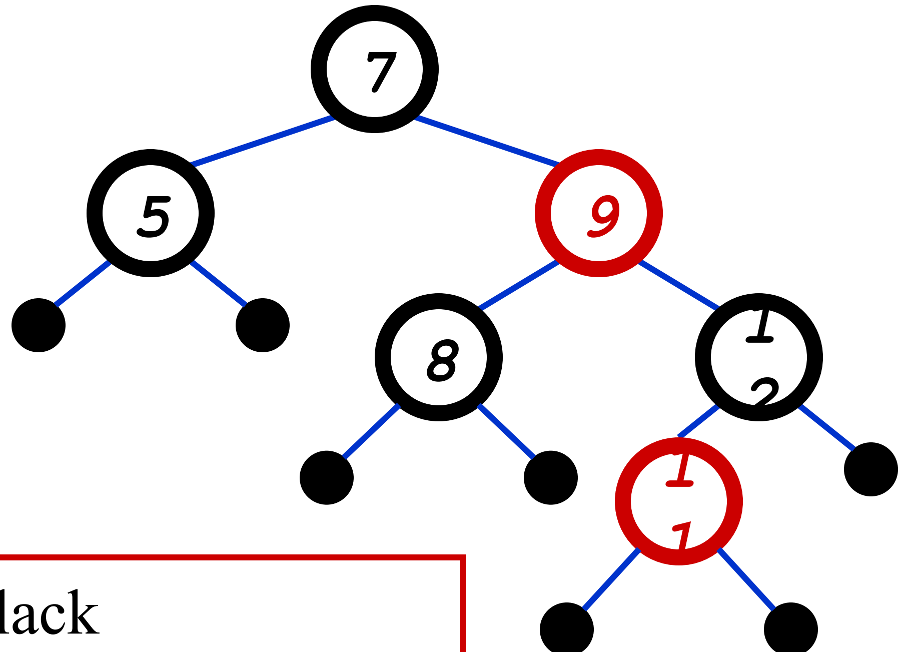
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Solution: recolor the tree



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

Red-Black Trees: The Problem With Insertion

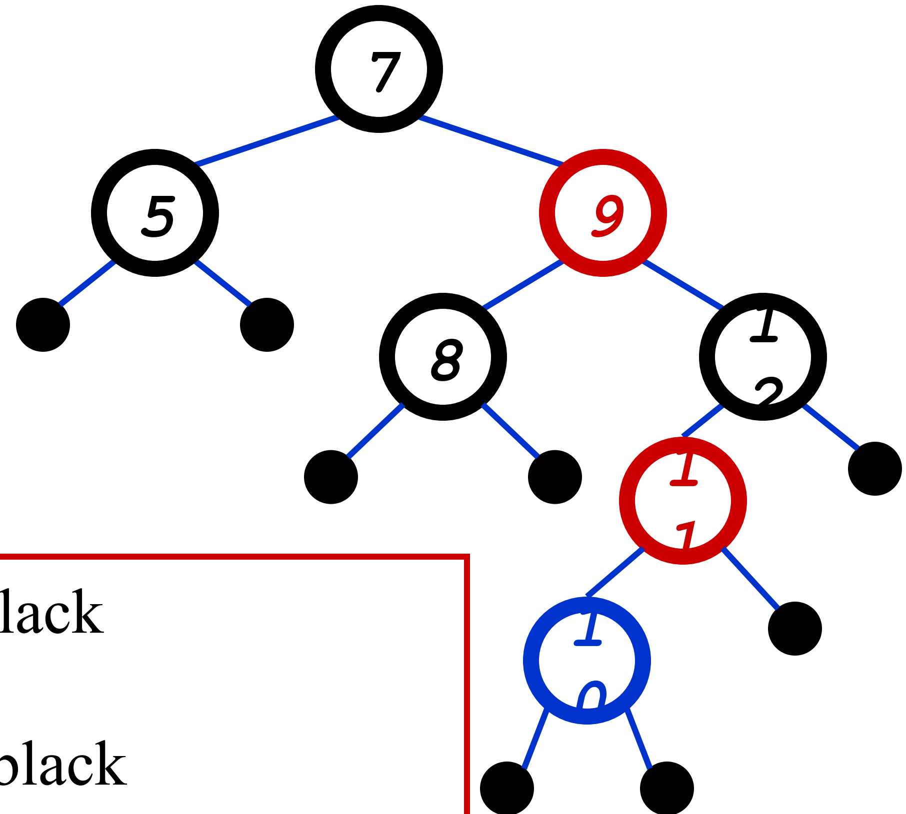
- Insert 10
  - *Where does it go?*



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

Red-Black Trees: The Problem With Insertion

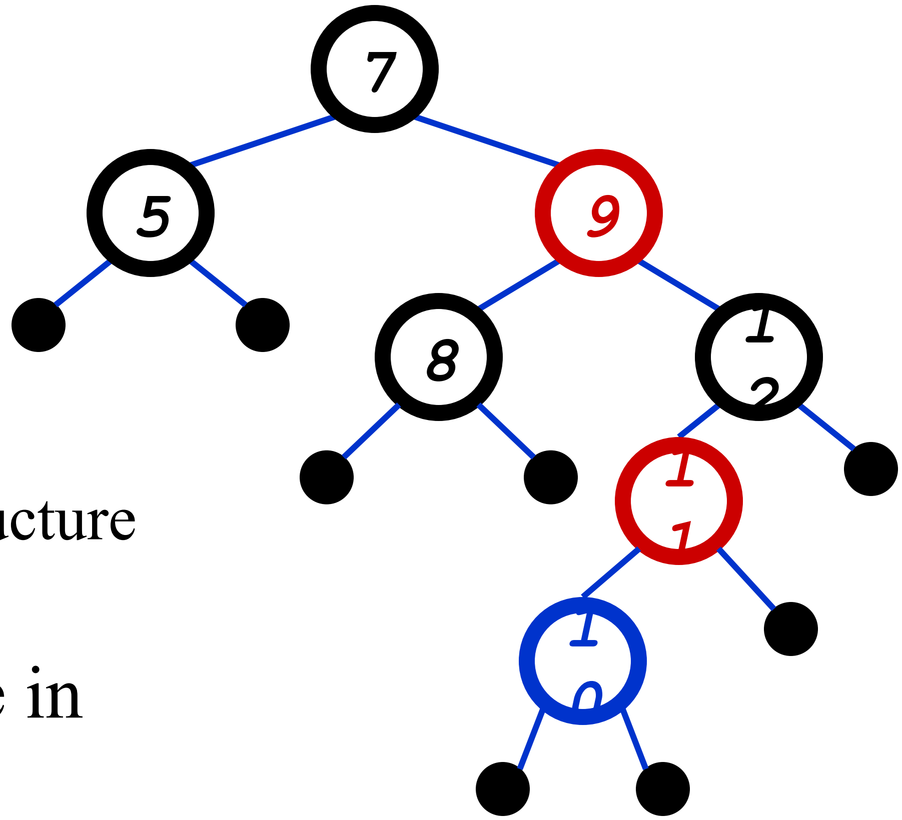
- Insert 10
  - *Where does it go?*
  - *What color?*



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

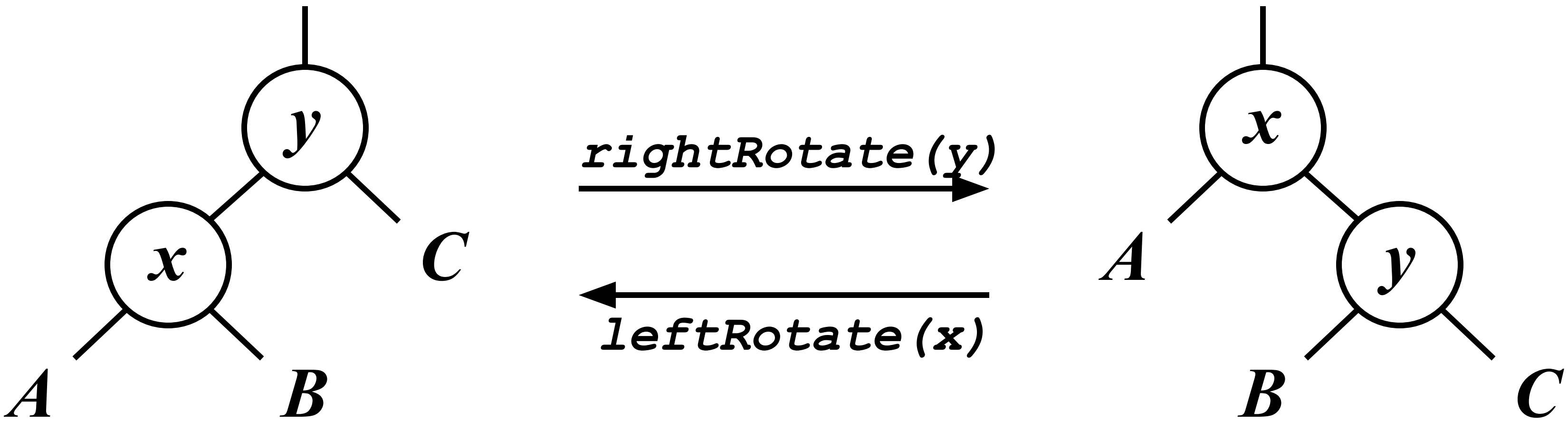
Red-Black Trees: The Problem With Insertion

- 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



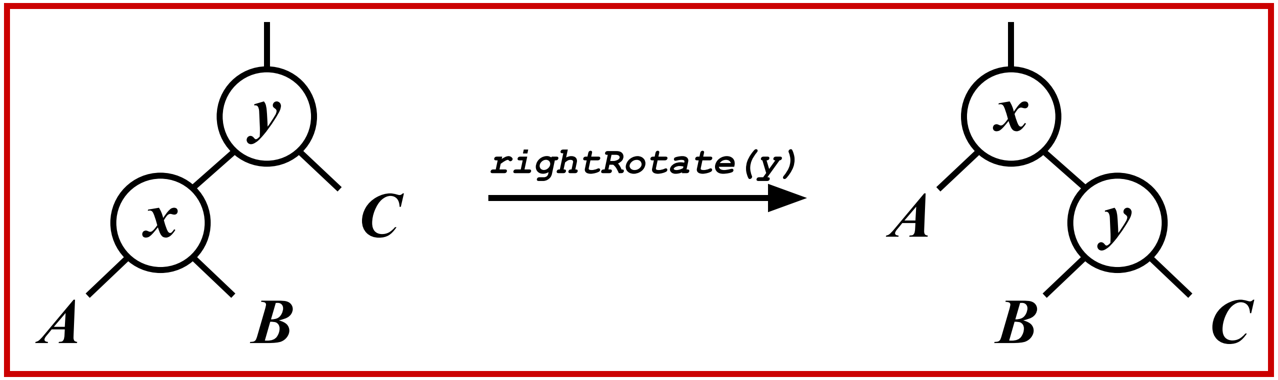
RB Trees: Rotation

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



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

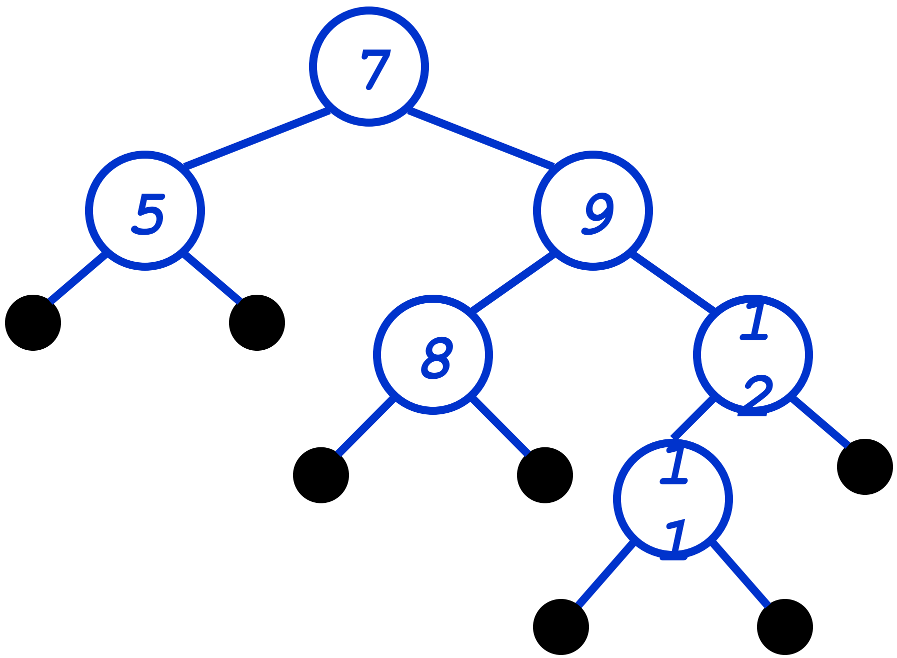
RB Trees: Rotation



- 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?*

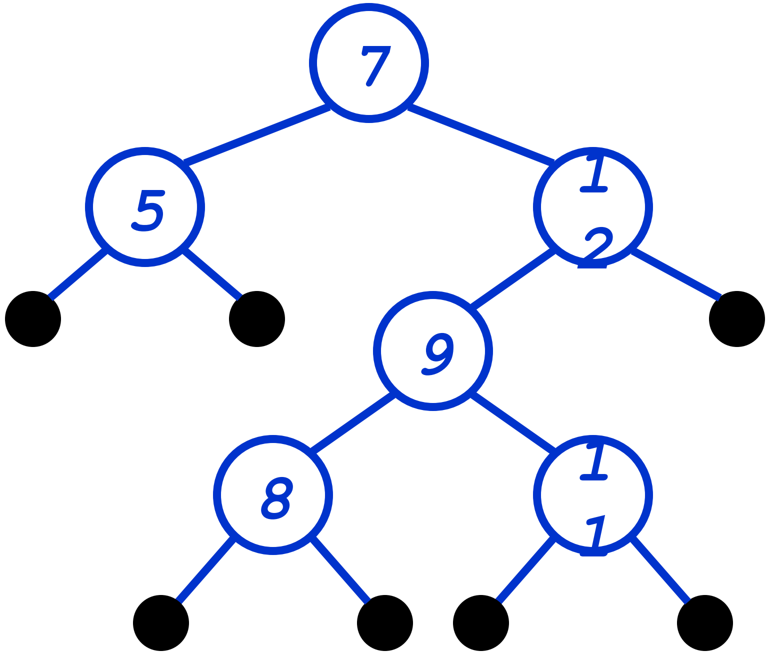
Rotation Example

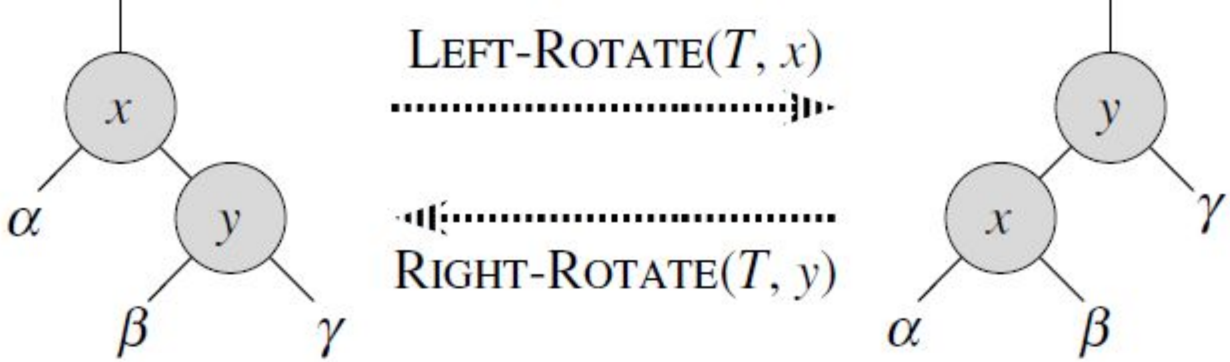
- Rotate left about 9:



Rotation Example

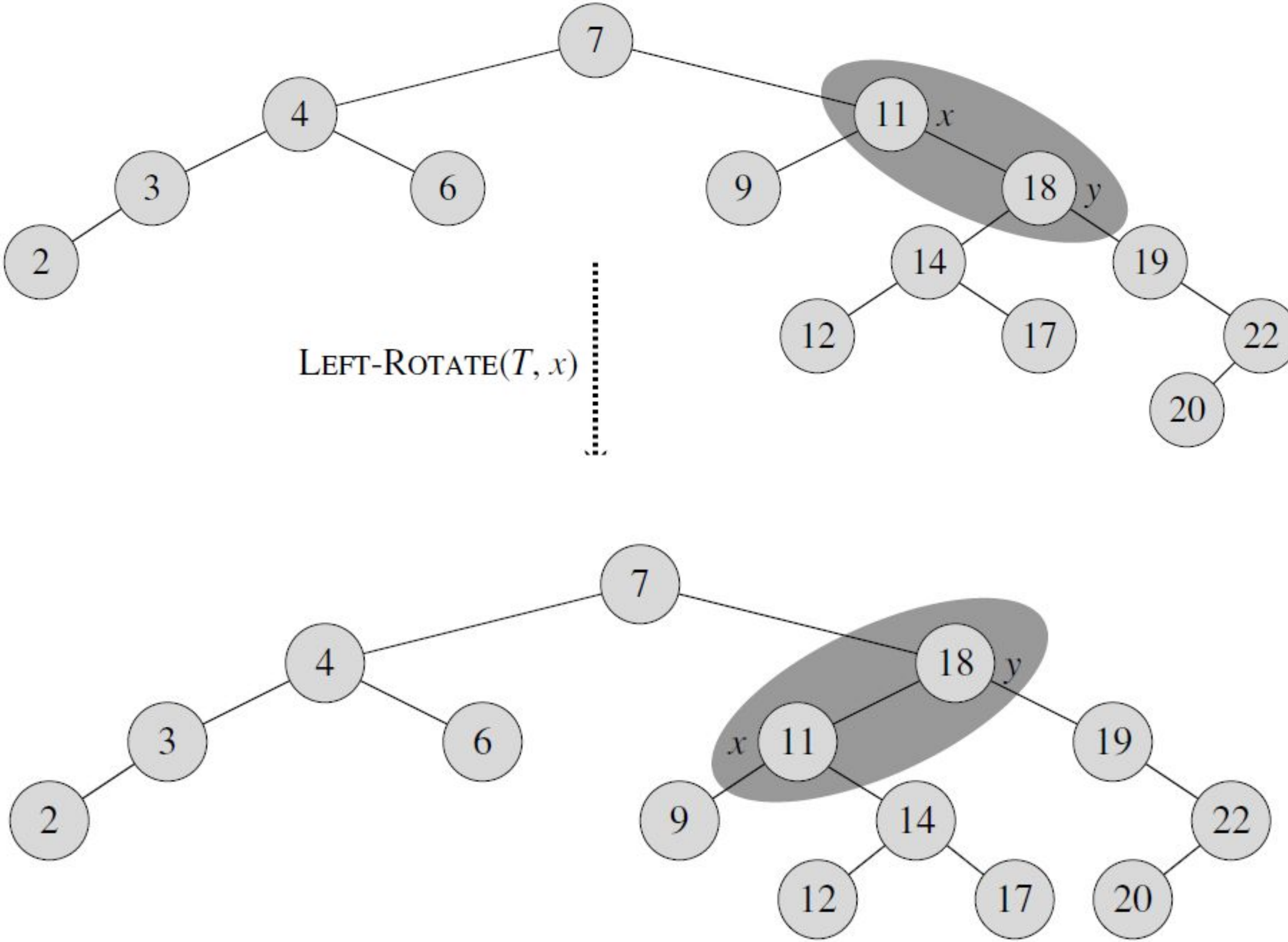
- Rotate left about 9:





LEFT-ROTATE(T, x)

```
y ← right[x]                 ▷ Set y.
right[x] ← left[y]           ▷ Turn y's left subtree into x's right subtree.
if left[y] ≠ nil[T]
   then p[left[y]] ← x
p[y] ← p[x]                  ▷ Link x's parent to y.
if p[x] = nil[T]
   then root[T] ← y
   else  if x = left[p[x]]
            then left[p[x]] ← y
            else  right[p[x]] ← y
left[y] ← x                  ▷ Put x on y's left.
p[x] ← y
```


LEFT-ROTATE(T, x)
7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20

Insertion

- Insert node z into the tree T as if it were an ordinary binary search tree, and then we color z red.
- To guarantee that the red-black properties are preserved, call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations.

RB-INSERT-FIXUP

Which red-black properties can be violated?

- #2: which requires the root to be black
- #4, which says that a red node cannot have a red child
- Property 2 is violated if z is the root, and property 4 is violated if z's parent is red.

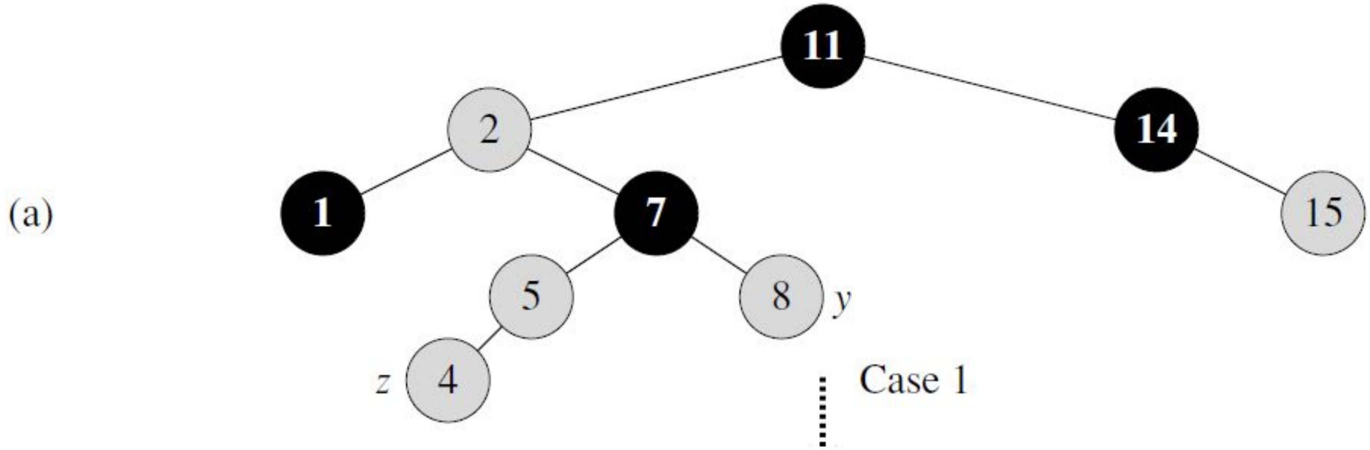


Figure 13.5 Case 1 of the procedure RB-INSERT. Property 4 is violated, since z and its parent $p[z]$ are both red. The same action is taken whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $p[p[z]]$ as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.

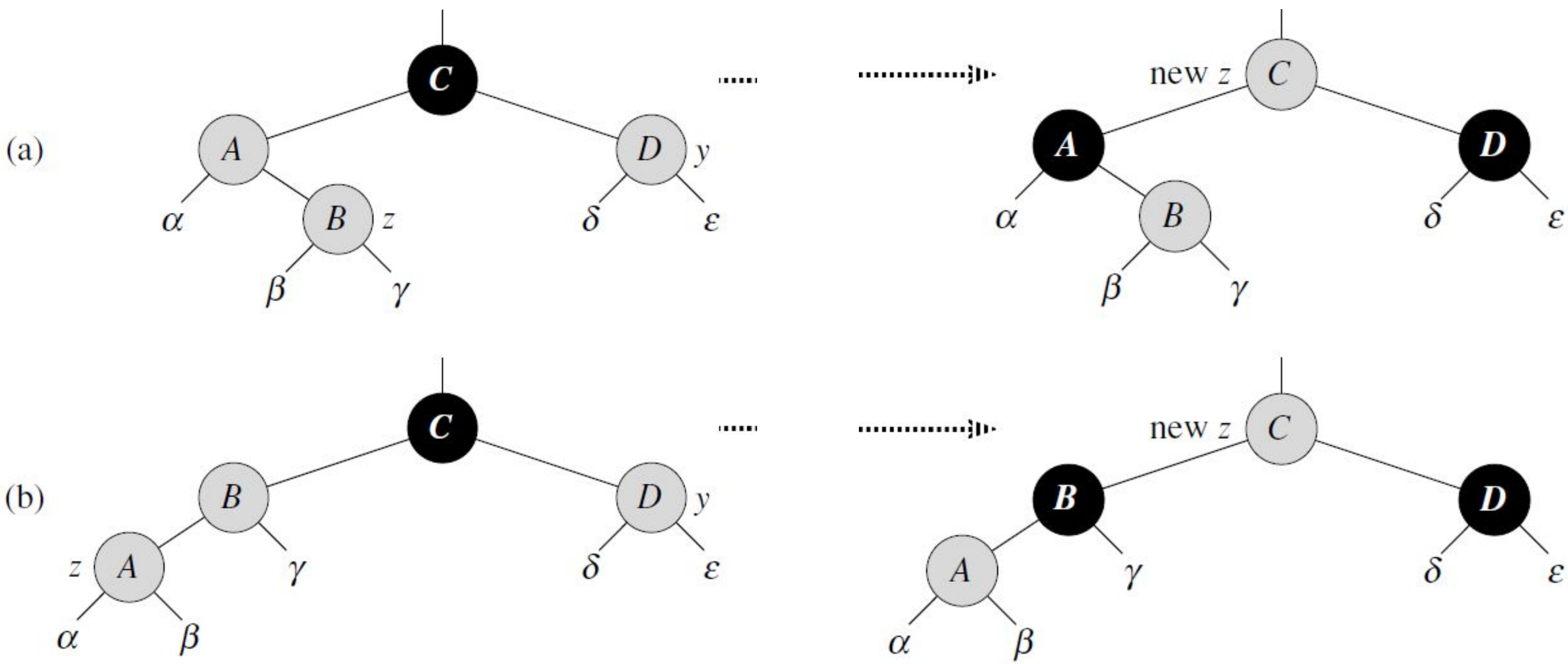
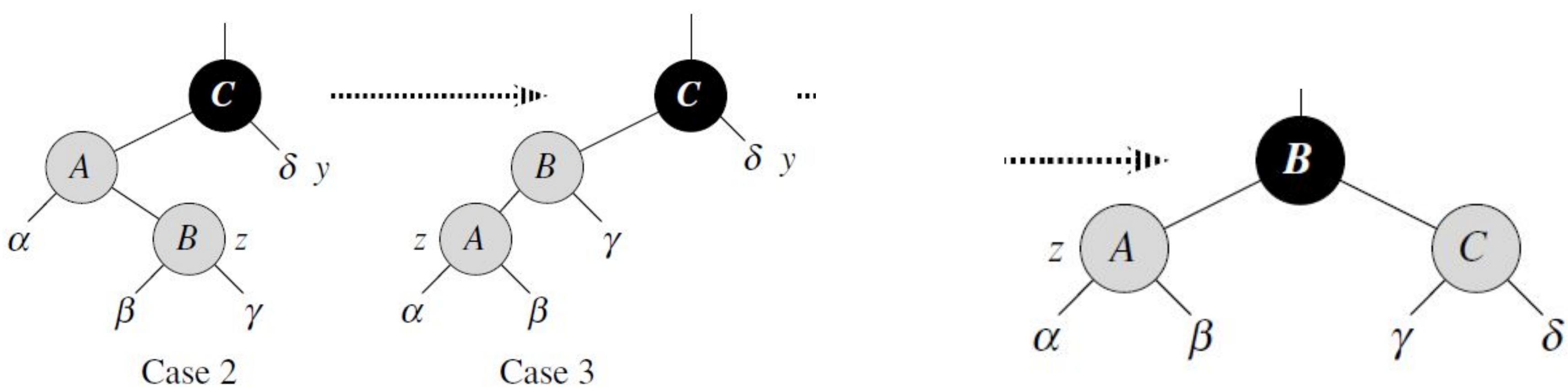


Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $p[z]$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 5: all downward paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.



Red-Black Trees: Insertion

- Insertion: the basic idea
  - Insert x into tree, color x red
  - Only r-b property 4 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;
      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);
  else    // x->p == x->p->p->right
      (same as above, but with
       "right" & "left" exchanged)
```

Case 1

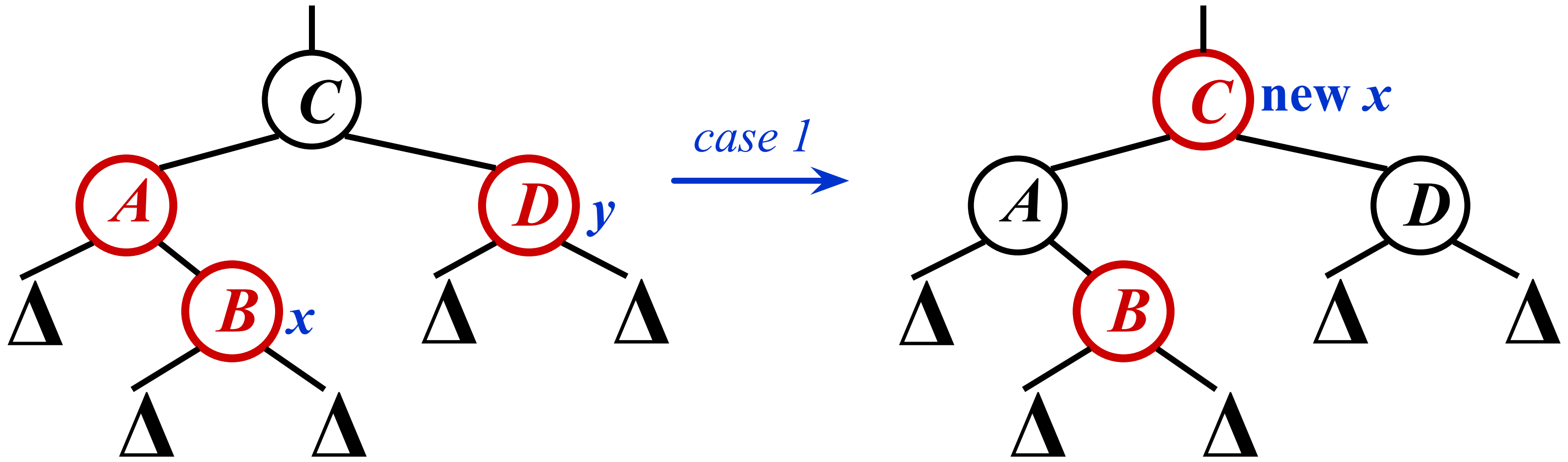
Case 2

```
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;          Case 1: uncle is RED
          x = x->p->p;
      else   // y->color == BLACK
          if (x == x->p->right)
              x = x->p;                  Case
              leftRotate(x);             2
          x->p->color = BLACK;
          x->p->p->color = RED;
          rightRotate(x->p->p);
  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 Δ’s are equal-black-height subtrees

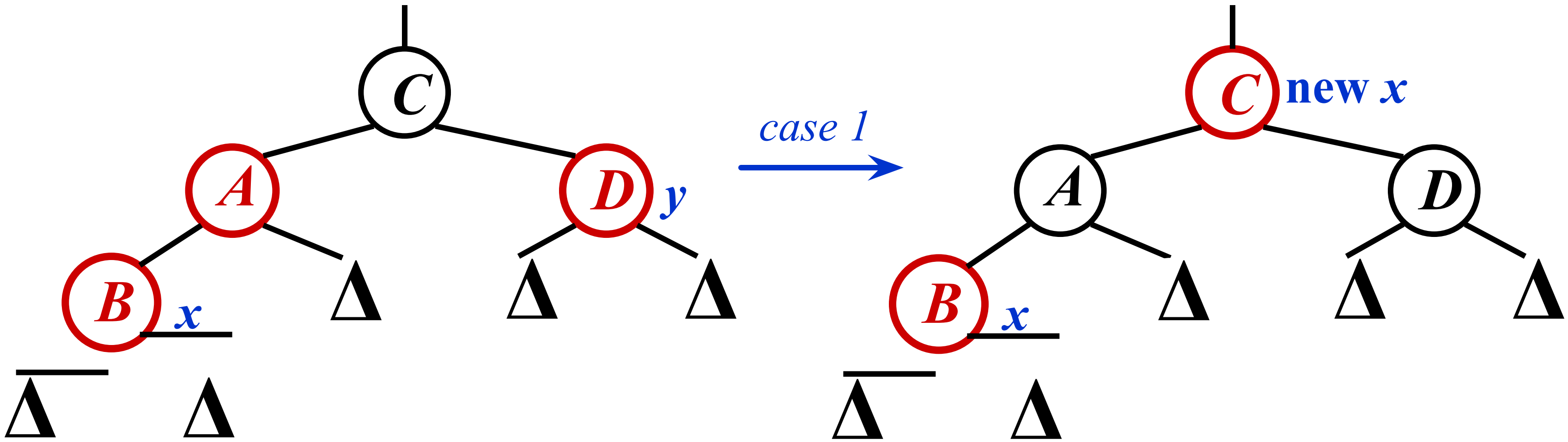


Change colors of some nodes, preserving #4: all downward paths have equal b.h.
The while loop now continues with x’s grandparent as the new x

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 Δ's are equal-black-height subtrees

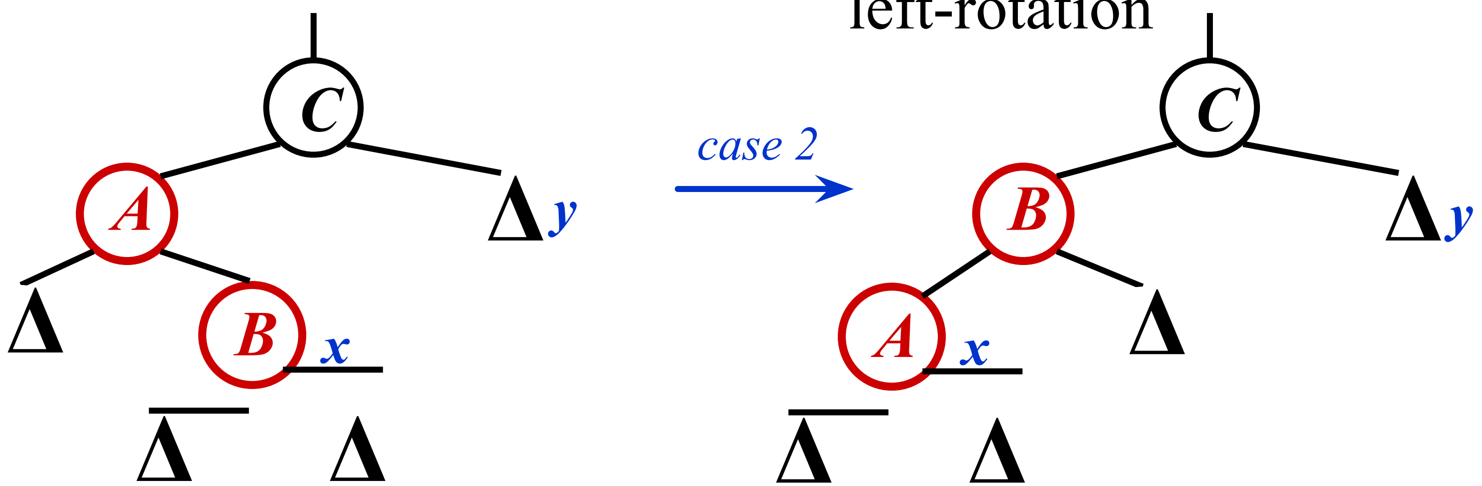


Same action whether x is a left or a right child

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

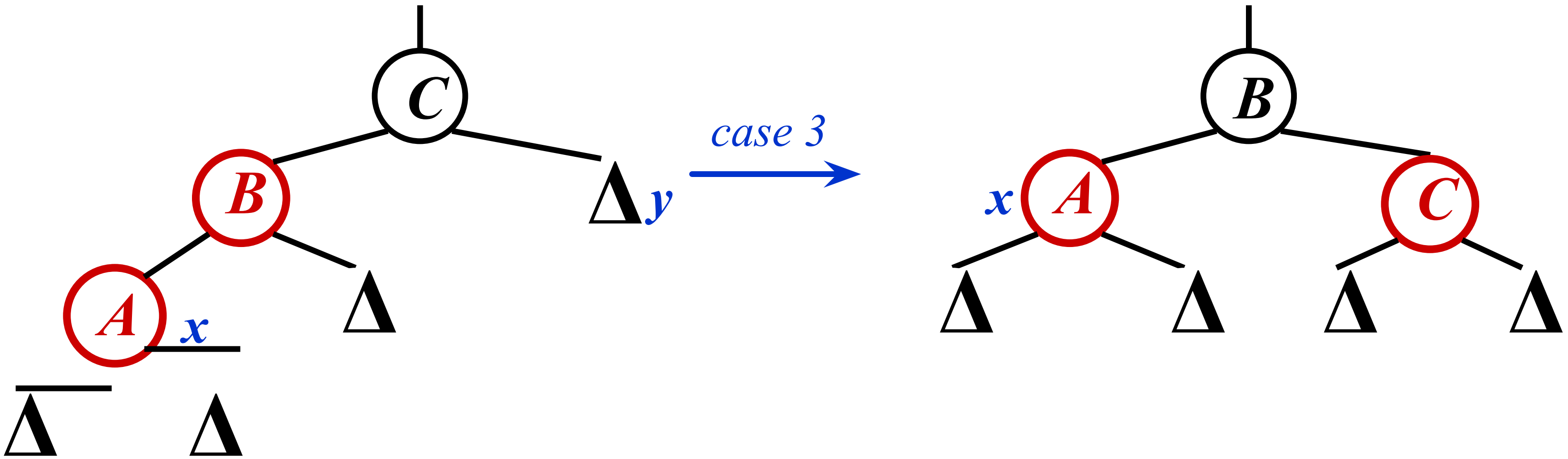


Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property 4: all downward paths contain same number of black nodes

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
  - Read for the overall picture, not the details

The End