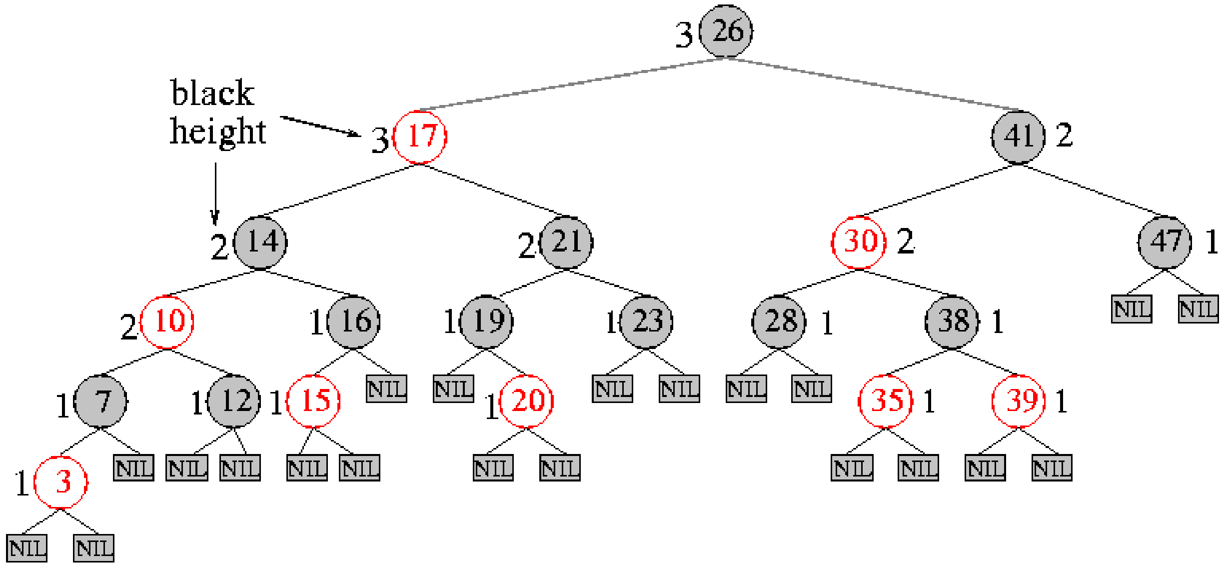
Unit 3: Data Structures on Trees

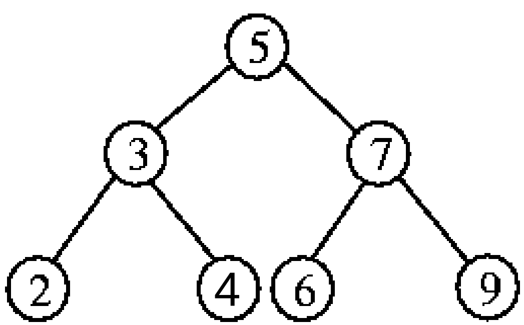
- Course contents:
  - Binary search trees
  - Red-black trees
- Readings:
  - Chapters 10 (self reading), 12, 13, and 14 (self reading for Chapter 14)



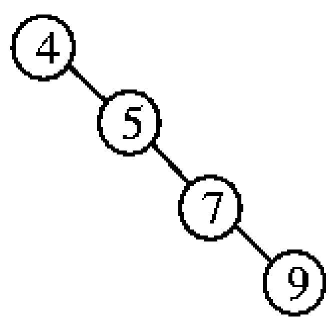
Binary Search Trees

Binary Search Tree

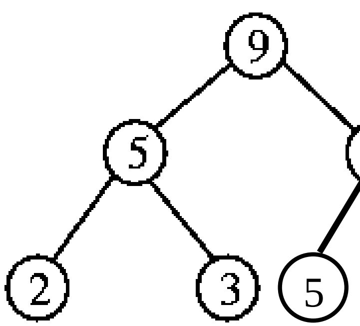
- **Binary-search-tree (BST) property:** Let x be a node in a BST.
  - If y is a node in the **left** subtree of x, then $y.key \leq x.key$.
  - If y is a node in the **right** subtree of x, then $x.key \leq y.key$.
- Tree construction: Worst case: $O(n^2)$; average case: $O(n \lg n)$, where n is the # of nodes.



complete binary search tree:
h = O(lg n)



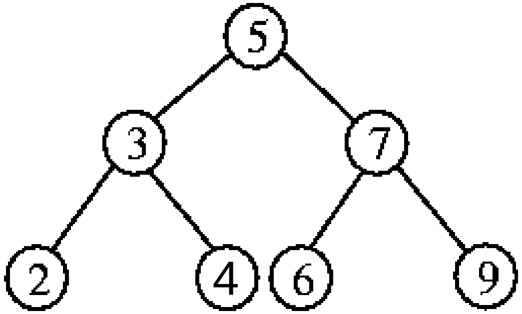
skewed binary search tree:
h = O(n)



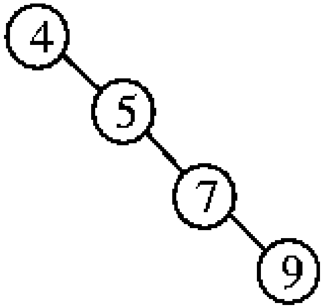
a heap, but not a search tree

Binary Search Tree (cont'd)

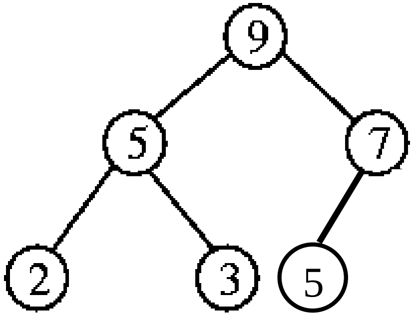
- Operations **Search, Minimum, Maximum, Predecessor, Successor, Insert, Delete** can be performed in $O(h)$ time, where h is the height of the tree.
- Worst case: $h = \theta(n)$; balanced BST: $h = \theta(\lg n)$.
- Can we guarantee $h = \theta(\lg n)$? **Balance search trees!!**



complete binary search tree:
h = O(lg n)



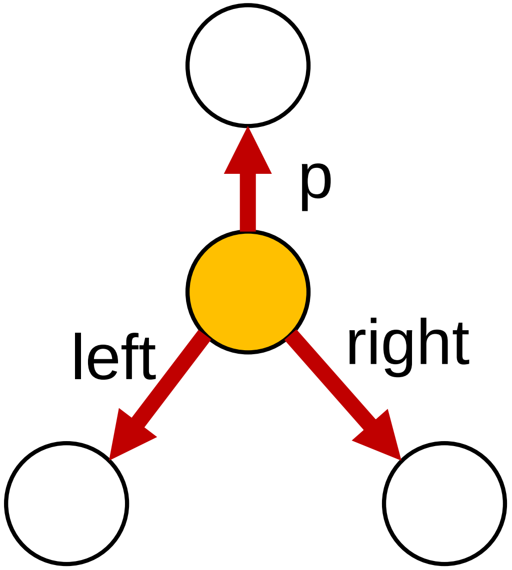
skewed binary search tree:
h = O(n)



a heap, but not a search tree

Implementation of a Tree Node

```
struct node
{
    int key_value;
    node *parent;
    node *left;
    node *right;
}
```

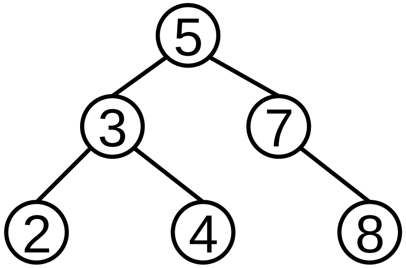


Tree Traversal

- Print out all the keys in sorted order in $\Theta(n)$ time
  - Print the root between the values in its left subtree and those in its right subtree

```
Inorder-Tree-Walk(x)
1.  if x ≠ NIL
2.      Inorder-Tree-Walk(x.left)
3.      Print x.key
4.      Inorder-Tree-Walk(x.right)
```

- Preorder/postorder
  - Print the root before/after the values in either subtree
- Example:
  - Infix: 2, 3, 4, 5, 7, 8
  - Prefix: 5, 3, 2, 4, 7, 8
  - Postfix: 2, 4, 3, 8, 7, 5



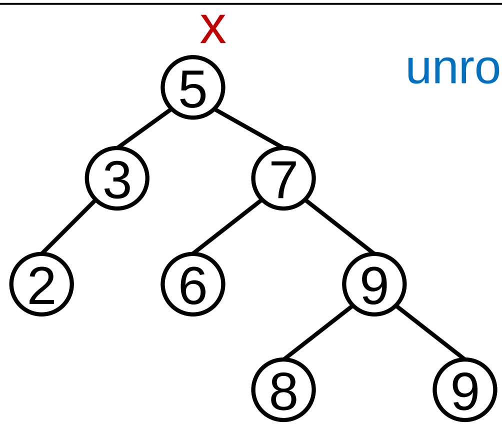
Tree Search

- Operations **Search** can be performed in $O(h)$ time, where h is the height of the tree.

```
Tree-Search(x, k)
   if x = NIL or k = x.key
       return x
   if k < x.key
       return Tree-Search(x.left, k)
   else  return Tree-Search(x.right, k)
```

```
Iterative-Tree-Search(x, k)
   while x ≠ NIL and k ≠ x.key
       if k < x.key
           x = x.left
       else x = x.right
   return x
```

unrolling the recursion into a while loop



Search for the key 8:
$5 \longrightarrow 7 \longrightarrow 9 \longrightarrow 8$

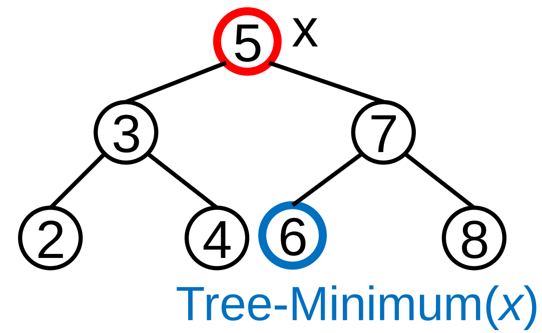
Tree Successor

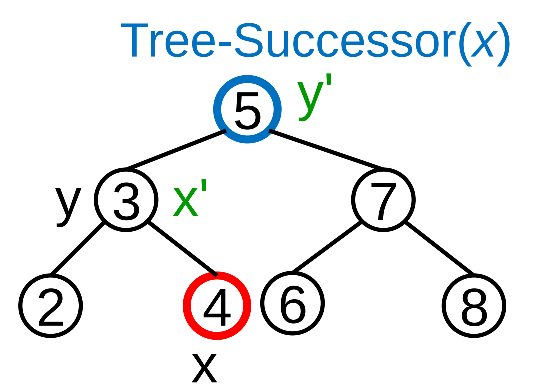
- **Successor** of a node *x*: a node y with the smallest key such that *key*[*y*] ≥ *key*[*x*]
- **Ancestor** of a node *x* is any node on the unique path from root to *x*, including *x*

```
Tree-Successor(x)
    if x.right ≠ NIL
        return Tree-Minimum(x.right)
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
    return y
```

```
Tree-Minimum(x)
    while x.left ≠ NIL
        x = x.left
    return x
```

y is the lowest ancestor of *x*, whose left child is also an ancestor of *x*





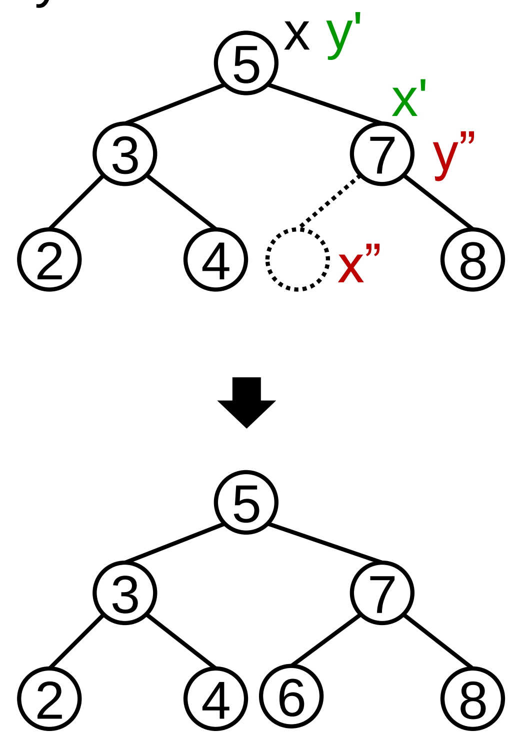
Tree Insertion

- Insert *z* into tree *T*.

```
Tree-Insert(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z  // T is empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

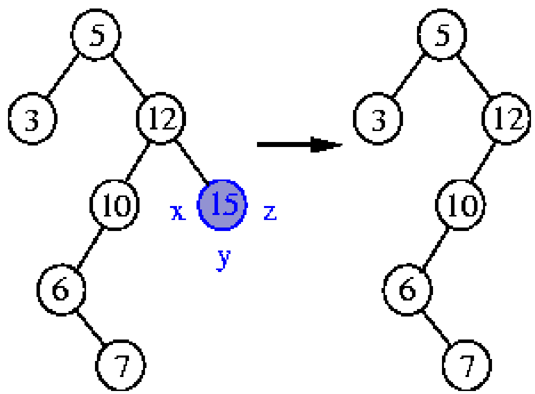
Insert “6” to the tree

y = NIL

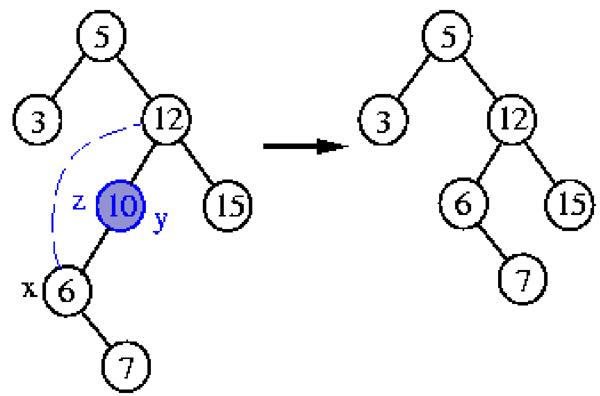


Deletion in Binary Search Trees

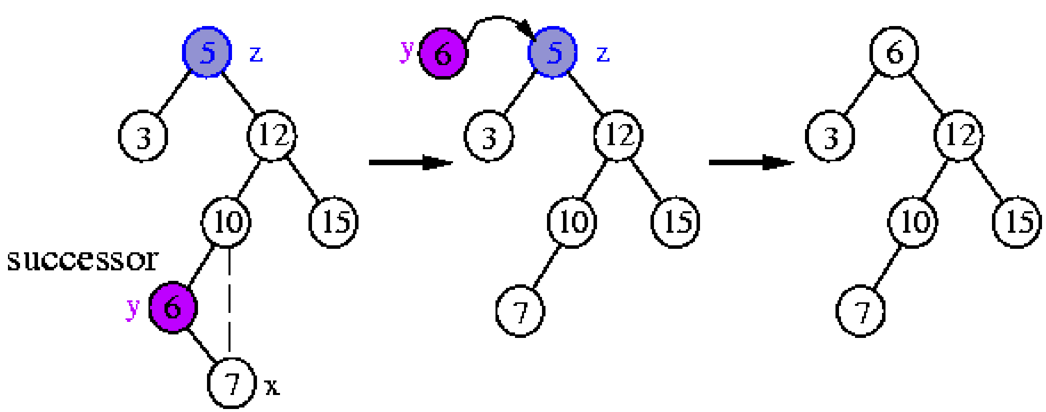
- **Case 1:** The node to be deleted (z) has **no children** (i.e., a leaf).
- **Case 2:** The node to be deleted (z) has only **one child**.
- **Case 3:** The node to be deleted (z) has **two children**.



case 1: z has no children.



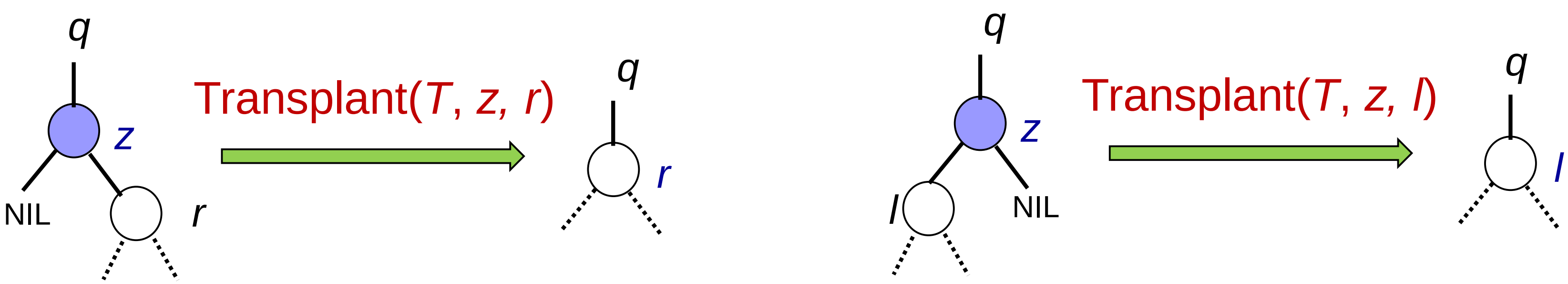
case 2: z has one child.



case 3: z has two children.

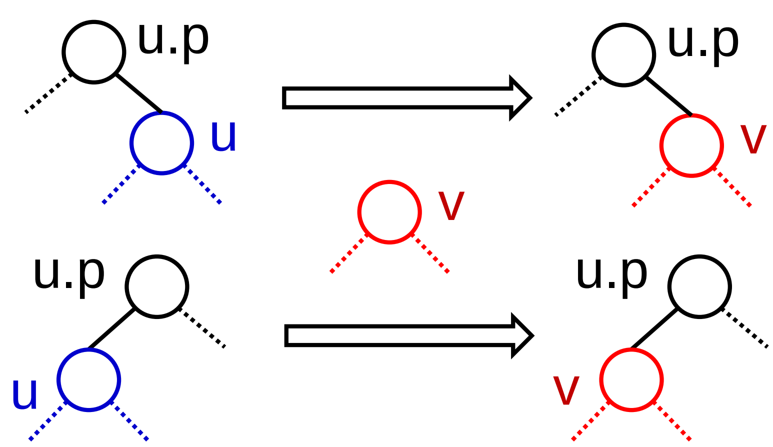
Deleting z in a Binary Search Tree

- If z has one child



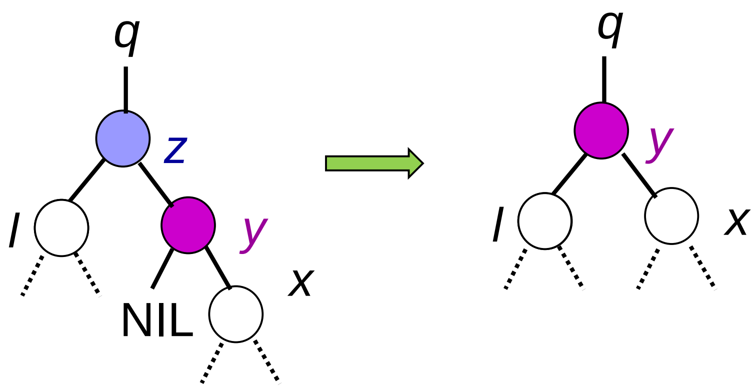

```
Transplant(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

Transplant(*T*, *u*, *v*) replace subtree rooted at u with subtree rooted at v

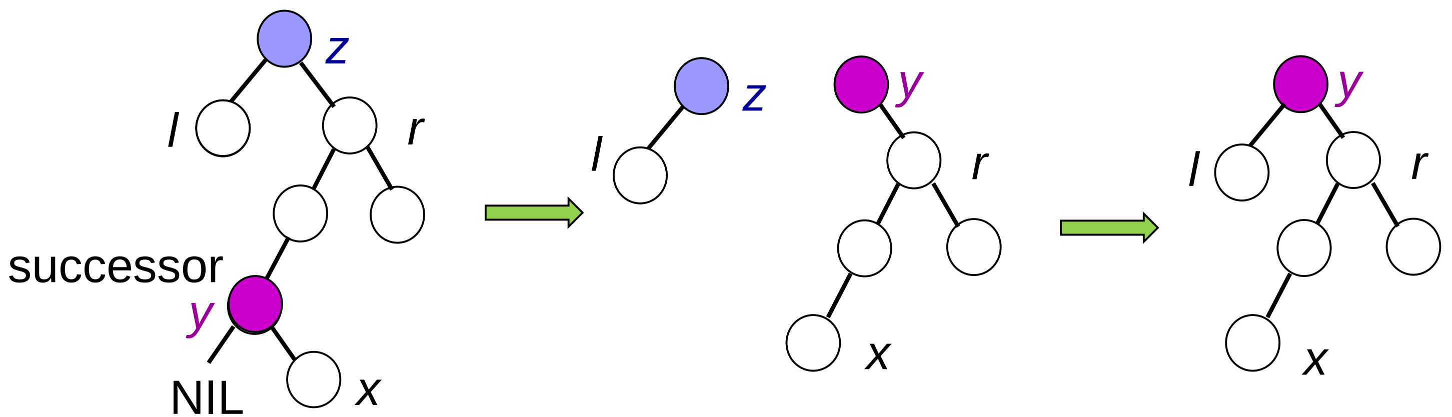


Deleting z in a Binary Search Tree (cont'd)

- If z has two children
  - If z's successor is z's right child

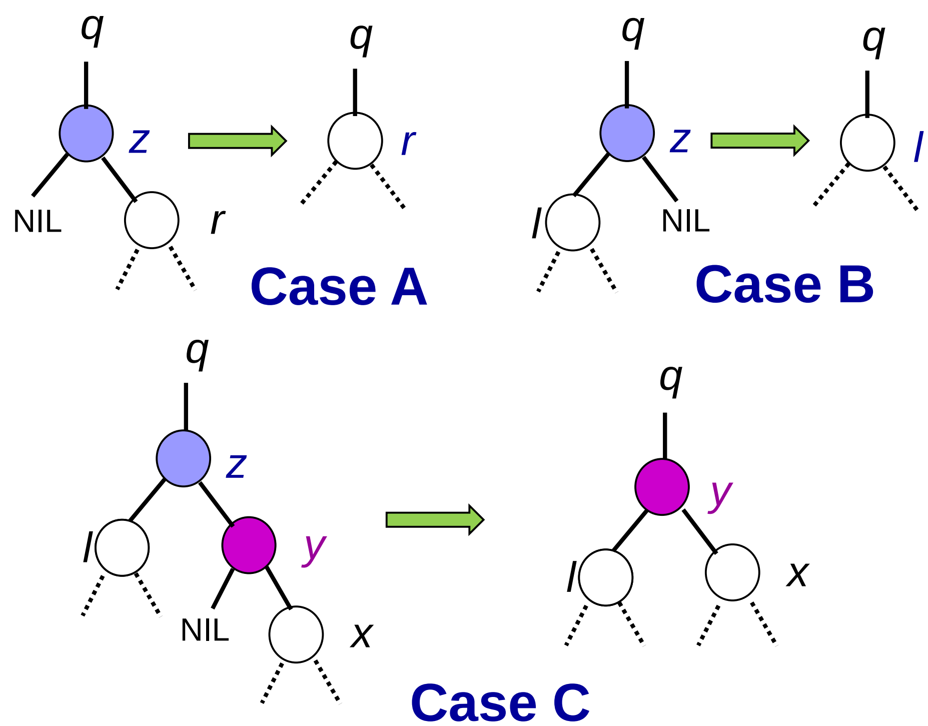


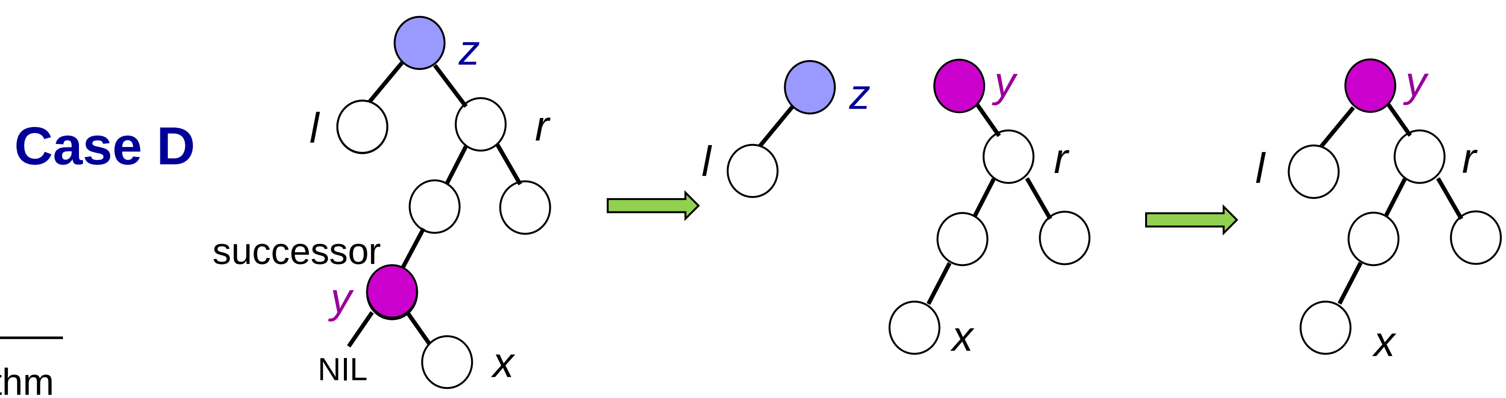
  - Otherwise



Tree Deletion

```
Tree-Delete(T, z)
if z.left == NIL    // case A
    Transplant(T, z, z.right)
elseif z.right == NIL    // case B
    Transplant(T, z, z.left)
else y = Tree-Minimum(z.right)
    if y.p ≠ z    // case D
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transplant(T, z, y) // case C.D
    y.left = z.left
    y.left.p = y
```



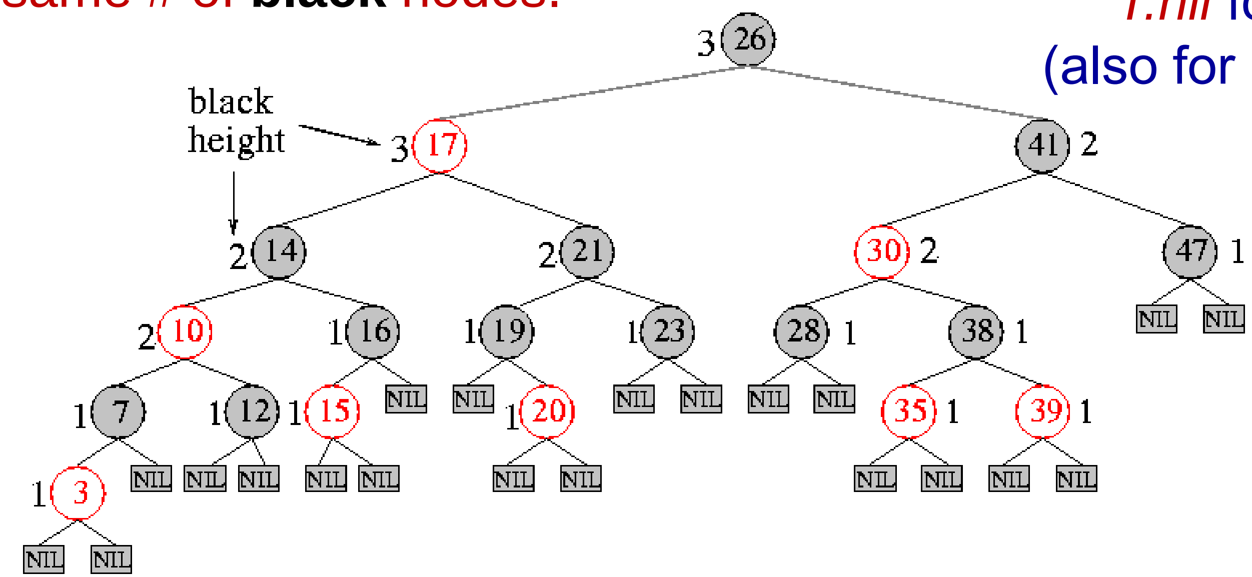


Red-Black Trees

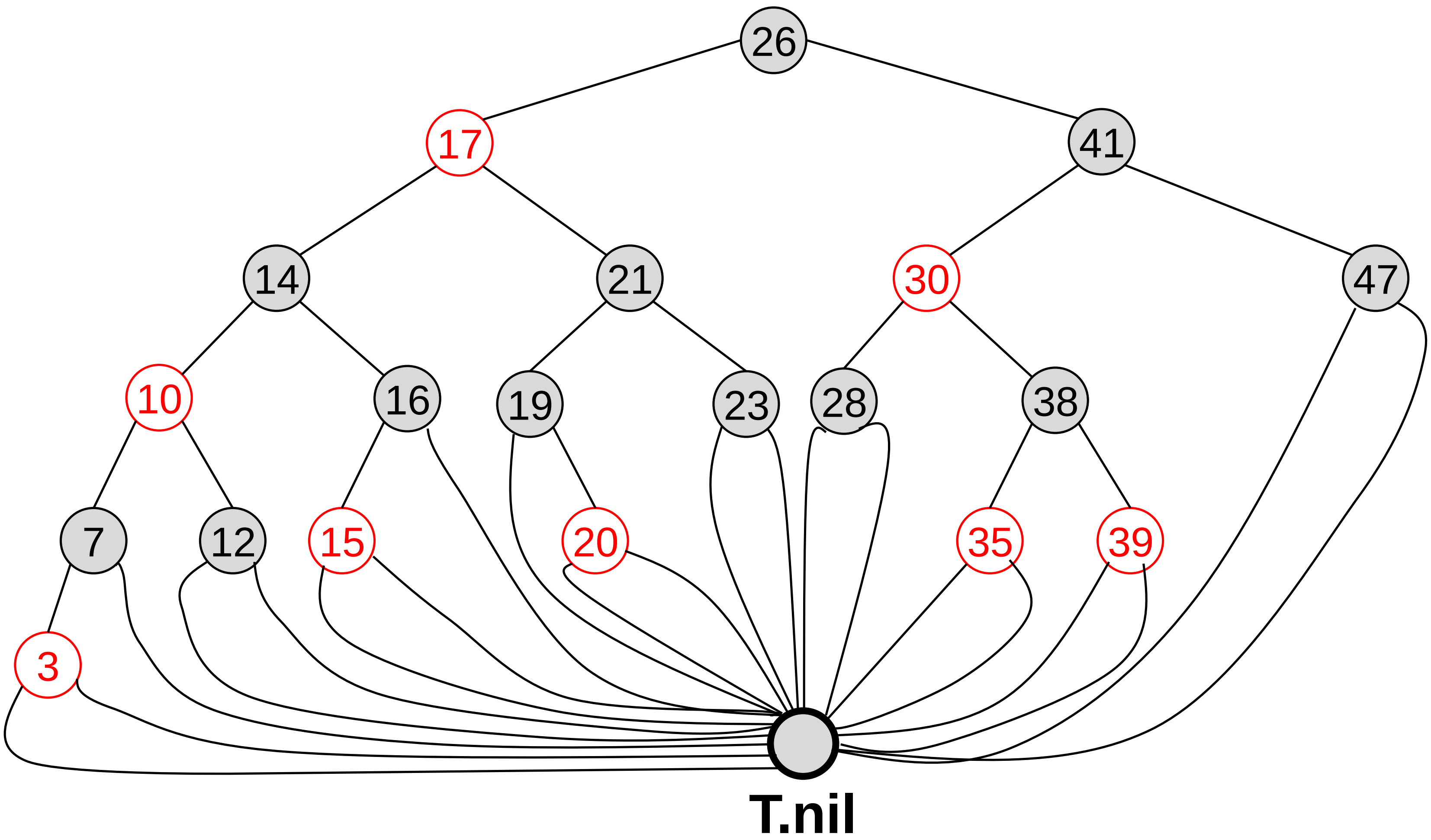
Balanced Search Trees: Red-Black Trees

- Add **color** field to nodes of binary trees.
- **Red-black tree properties:**
  1. Every node is either **red** or **black**.
  2. The root is **black**.
  3. Every leaf (NIL) is **black**.
  4. If a node is **red**, both its children are **black** (i.e., no two consecutive reds on a simple path).
  5. Every simple path from a node to a descendent leaf contains the same # of **black** nodes.

T.nil for all NIL's (also for root's parent)



The Actual Data Structure



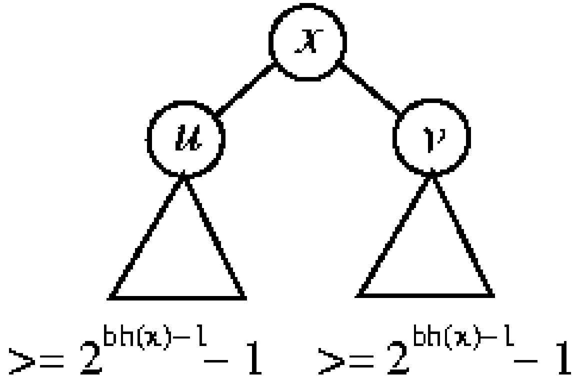
Black Height

- **Black height** of node *x*, $bh(x)$: # of blacks on path to leaf, **not counting *x*.**
- **Theorem:** A red-black tree with *n* internal nodes has height at most $2\lg(n+1)$.
  - Strategy: First bound the # of nodes in any subtree, then bound the height of any subtree.
  - **Claim 1:** Any subtree rooted at *x* has $\geq 2^{bh(x)} - 1$ internal nodes.
  - **Claim 2:** $bh(root) \geq h/2$ (*h*: height of the red-black tree), i.e., at least half the nodes on any single path from the root to leaf must be black.
  - At root, $n \geq 2^{bh(root)} - 1 \geq 2^{h/2} - 1 \Rightarrow h \leq 2\lg(n+1)$.

Black Height (cont'd)

- **Claim 1:** Any subtree rooted at x has $\geq 2^{bh(x)} - 1$ internal nodes.
- Prove by induction
  - $bh(x) = 0 \rightarrow x$ is a NIL (leaf) node.
  - Assume the claim is true for all trees with black height $< bh(x)$.
    - If x is red, both subtrees have black height $bh(x) - 1$.
    - If x is black, both subtrees have black height at least $bh(x)-1$.
  - Thus, # of internal nodes in any subtree rooted at x is

    $n_x \geq (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 \geq 2^{bh(x)} - 1$.

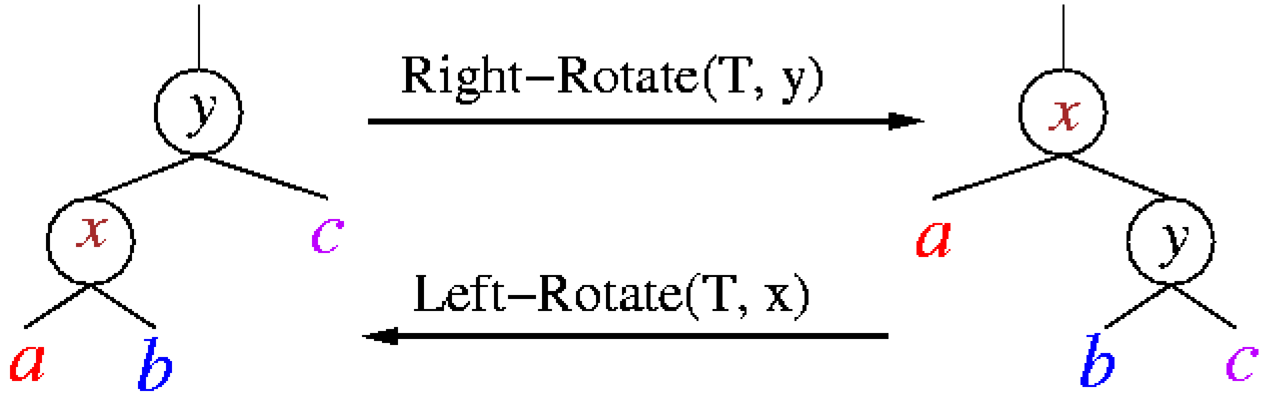


Black Height (cont'd)

- **Claim 2:** $bh(root) \geq h/2$, (h: height of the red-black tree)
  - Claim 2 can be proved by Property 4: no two consecutive reds on a simple path.
- By Claim 1 and Claim 2, a red-black tree with n internal nodes has height at most $2\lg(n+1)$.
  - Thus, red-black trees are **balanced**. (Height is at most twice optimal.)
- **Corollary:** Search, Minimum, Maximum, Predecessor, Successor take $O(\lg n)$ time.
- How about **Delete** and **Insert**?
  - Need to maintain the red-black tree properties!

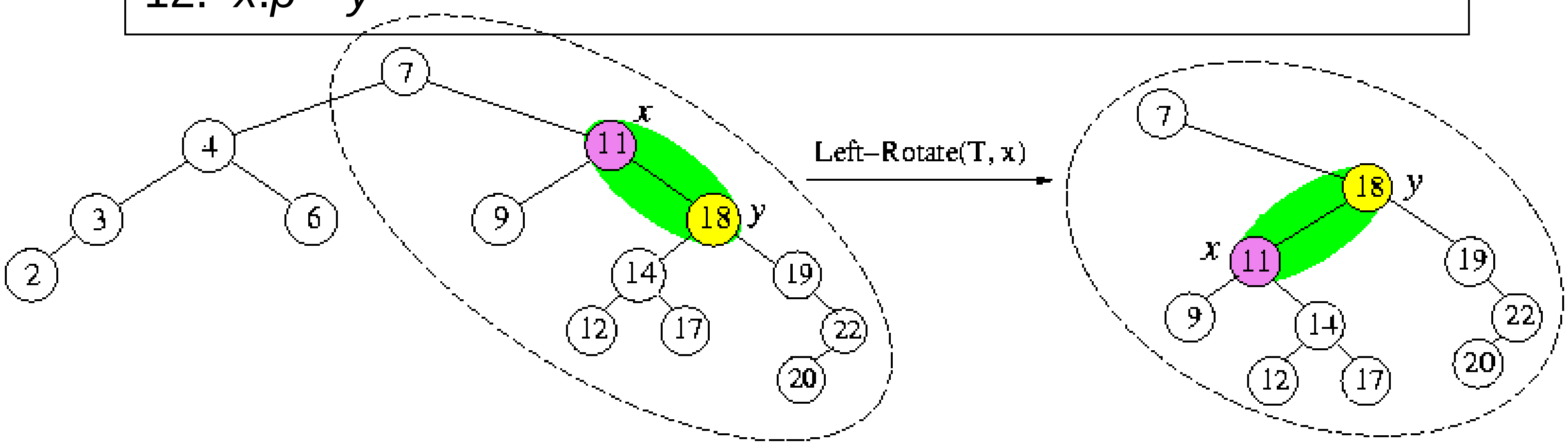
Rotations

- **Left/right rotations:** The basic restructuring step for binary search trees.
- Rotation is a local operation changing $O(1)$ pointers.
- **In-order property preservation:** An in-order search tree before a rotation stays an in-order one.
  - In-order: <*a*, *x*, *b*, *y*, *c*>
  - **The property of a binary search tree still holds!!**



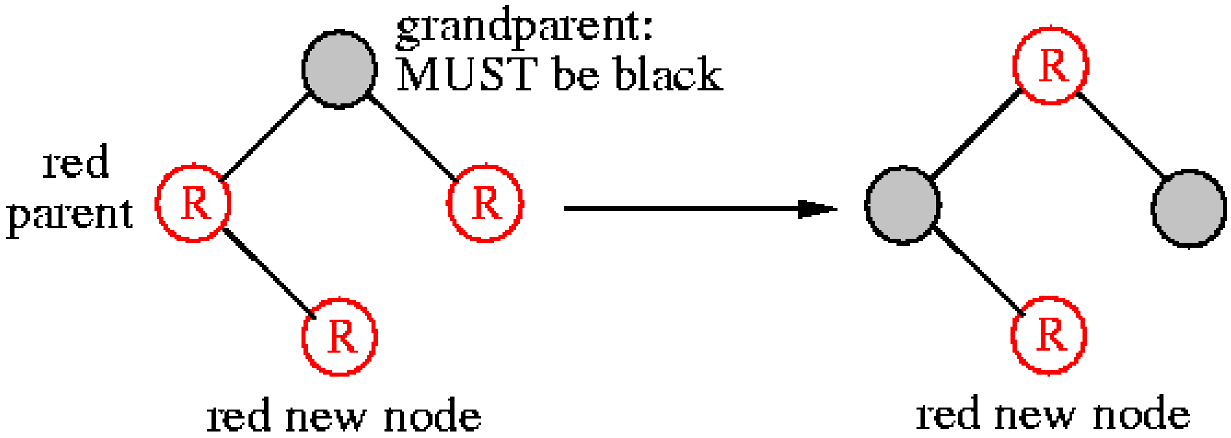
Left Rotation

```
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 is a left child
        x.p.left = y
    else x.p.right = y
    y.left = x         // Put x on y's left
    x.p = y
```



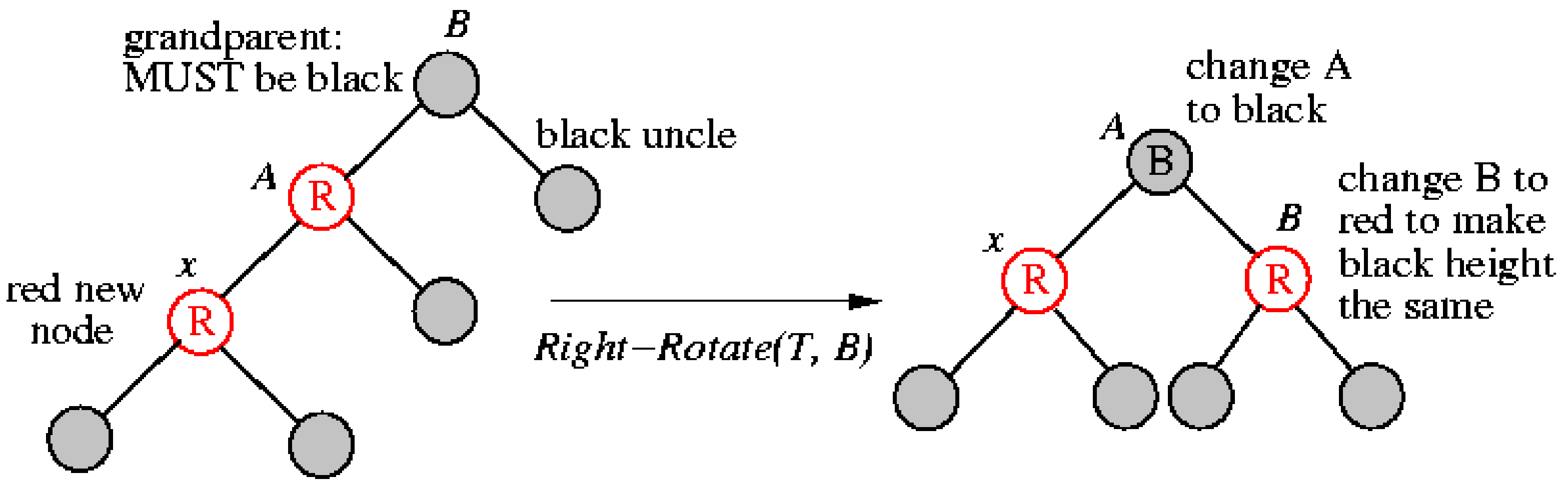
Insertion: Red Uncle

- Every insertion takes place at a leaf; this changes a black NIL pointer to a node with two black NIL pointers.
- To preserve the black height of the tree, the new node is set to **red**.
  - If the new parent is **black**, then we are done; otherwise, the tree must be restructured (Property 4 violation)!!
- How to fix two reds in a row? Check **uncle's color!**
  - If the uncle is **red**, reversing the relatives' colors either solves the problem or pushes it one-level higher.



Insertion: Black Uncle

- How to fix two reds in a row? Check **uncle's color!**
  - If the uncle is **black** (all nodes around the new node and its parent must be black), rotate right about the grandparent.
  - Change some nodes' colors to make the black height the same as before.

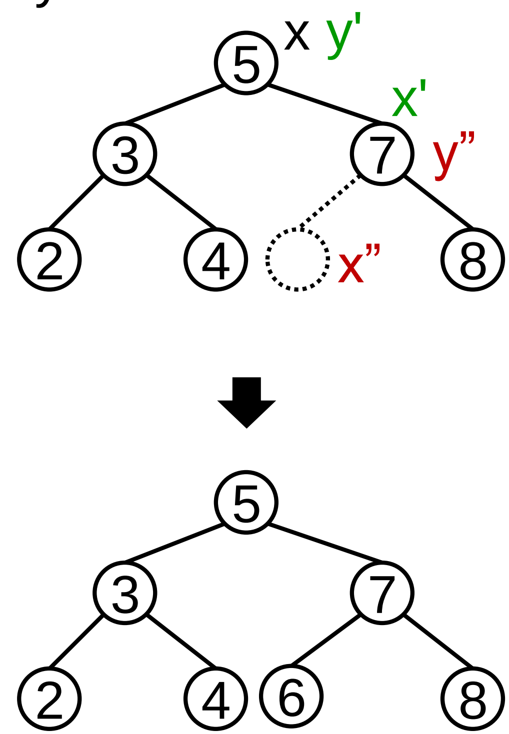


RB-Tree Insertion

```
RB-Insert(T, z)   // insert z into T
    y = T.nil
    x = T.root
    while x ≠ T.nil
        y = x
        if z.key < x.key
            x = x.left
        else x = x.right
    z.p = y
    if y == T.nil
        T.root = z   // T is empty
    elseif z.key < y.key
        y.left = z
    else y.right = z
    z.left = T.nil
    z.right = T.nil
    z.color = RED
    RB-Insert-Fixup(T,z)
```

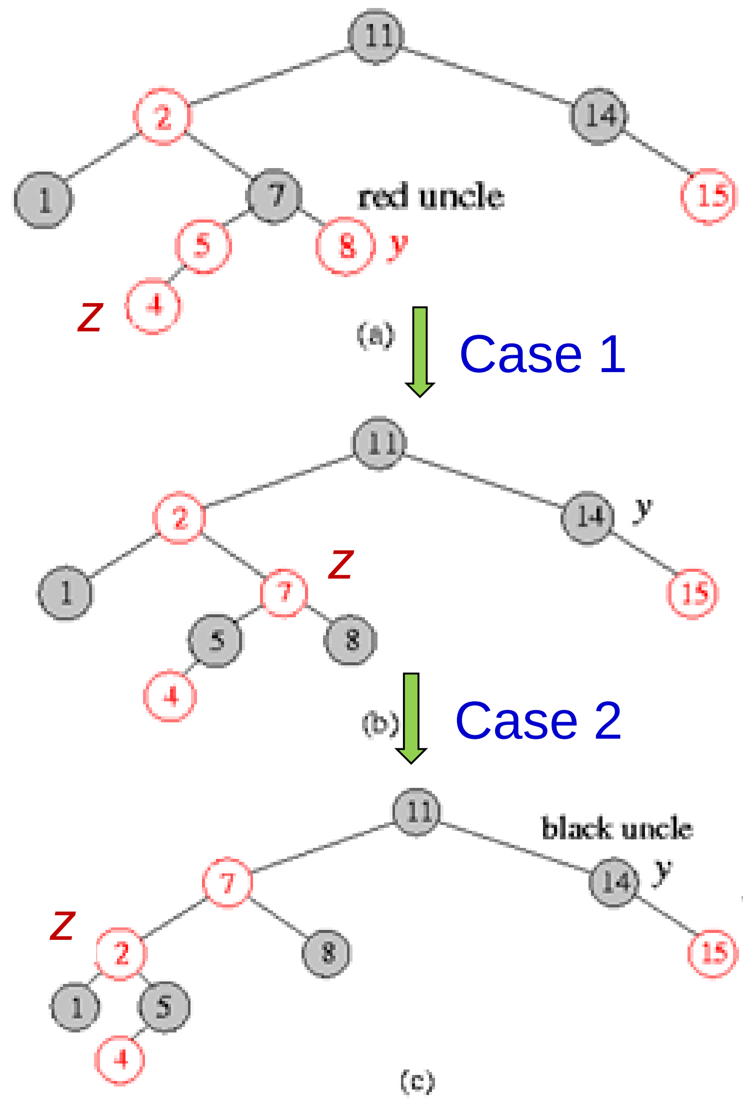
Insert “6” to the tree

y = NIL



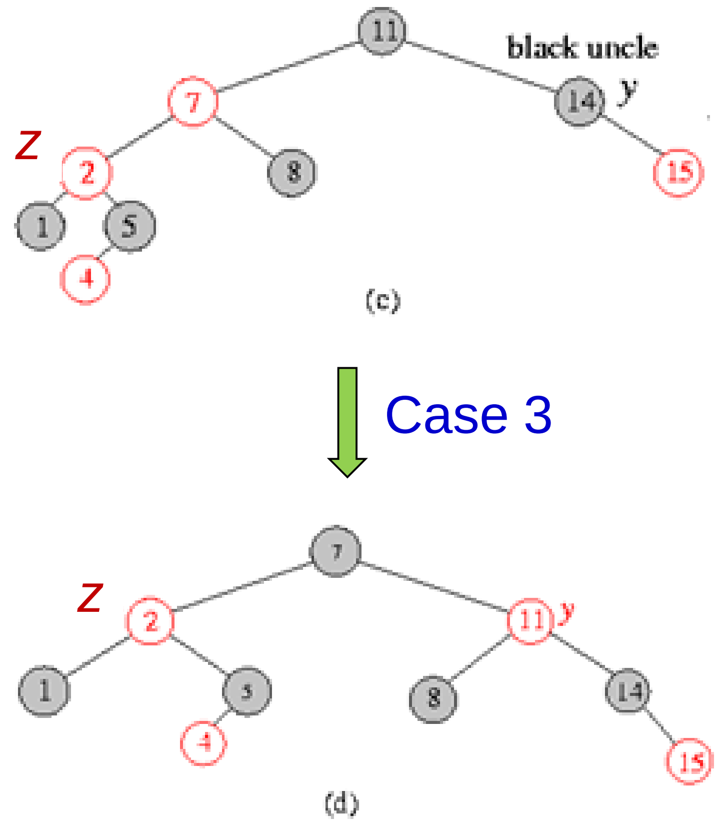
RB-Tree Insertion Fixup

```
RB-Insert-Fixup(T, z)
   while z.p.color == RED
      if z.p == z.p.p.left
         y = z.p.p.right
         if y.color == RED
            z.p.color = BLACK     // case 1
            y.color = BLACK       // case 1
            z.p.p.color = RED     // case 1
            z = z.p.p             // case 1
         else
            if z == z.p.right
               z = z.p               // case 2
               Left-Rotate(T, z)     // case 2
            z.p.color = BLACK       // case 3
            z.p.p.color = RED       // case 3
            Right-Rotate(T, z.p.p)  // case 3
      else (same as then clause with
         "right" and "left" exchanged)
   T.root.color = BLACK
```



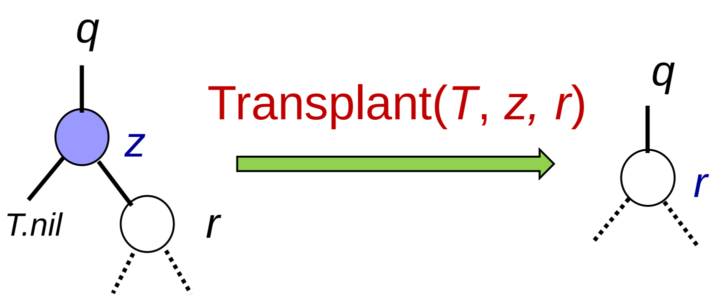
RB-Tree Insertion Fixup (cont'd)

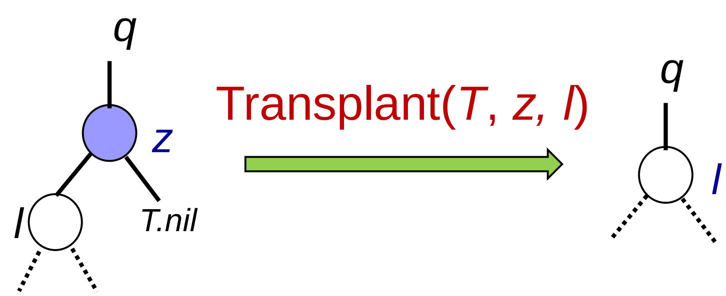
```
RB-Insert-Fixup(T, z)
  while z.p.color == RED
      if z.p == z.p.p.left
          y = z.p.p.right
          if y.color == RED
              z.p.color = BLACK        // case 1
              y.color = BLACK          // case 1
              z.p.p.color = RED        // case 1
              z = z.p.p                // case 1
          else
              if z == z.p.right
                  z = z.p              // case 2
                  Left-Rotate(T, z)    // case 2
              z.p.color = BLACK        // case 3
              z.p.p.color = RED        // case 3
              Right-Rotate(T, z.p.p)   // case 3
      else (same as then clause with
          "right" and "left" exchanged)
  T.root.color = BLACK
```



Deleting z in an RB Tree

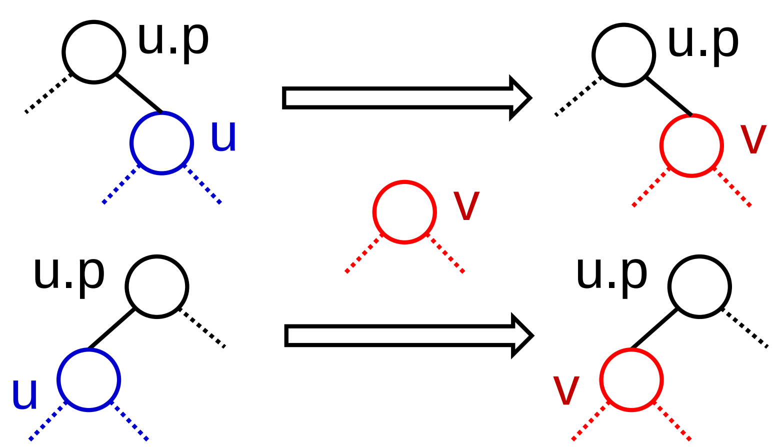
- If z has one child





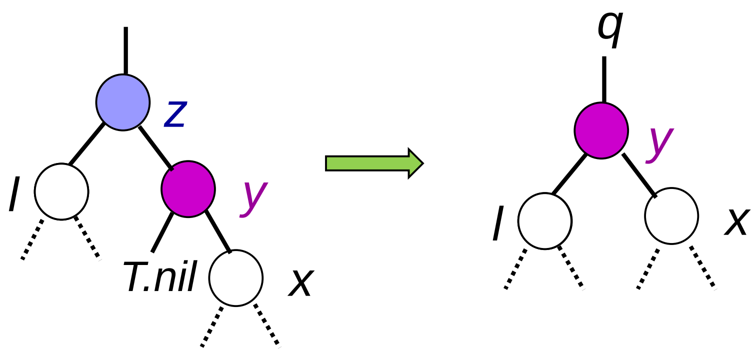

```
RB-Transplant(T, u, v)
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

Transplant(*T*, *u*, *v*) replace subtree rooted at u with subtree rooted at v

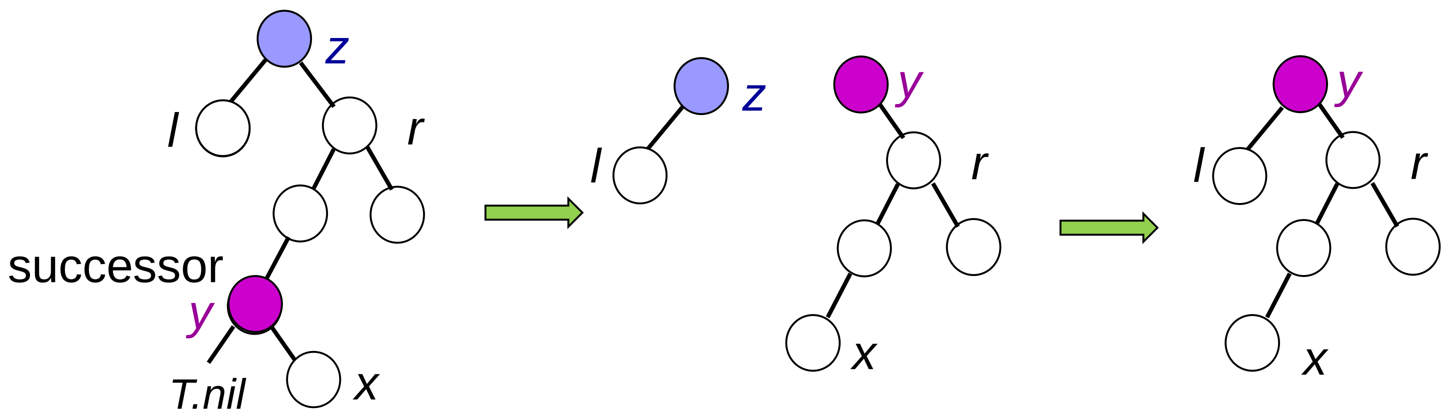


Deleting z in a Binary Search Tree (cont'd)

- If z has two children
  - If z's successor is z's right child

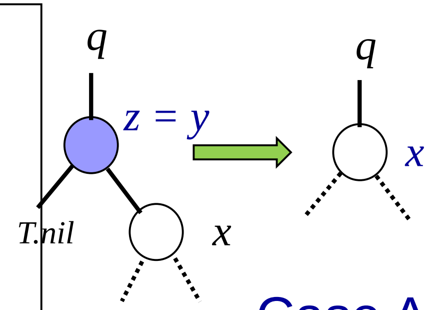


  - Otherwise



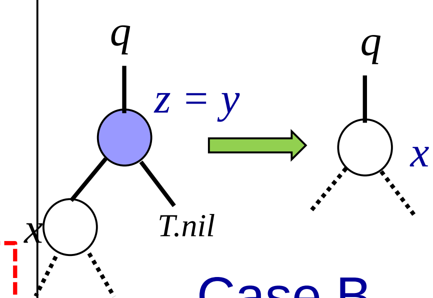
RB-Tree Deletion

```
RB-Delete(T, z)
1.  y = z
2.  y-original-color = y.color
3.  if z.left == T.nil              // case A
4.      x = z.right
5.      RB-Transplant(T, z, z.right)
6.  elseif z.right == T.nil         // case B
7.      x = z.left
8.      RB-Transplant(T, z, z.left)
9.  else y = Tree-Minimum(z.right)
10.     y-original-color = y.color
11.     x = y.right
12.     if y.p == z                 // case C
13.        x.p = y
14.     else RB-Transplant(T, y, y.right)
15.        y.right = z.right    // case D
16.        y.right.p = y
17.      RB-Transplant(T, z, y)  // cases C, D
18.      y.left = z.left
19.      y.left.p = y
20.      y.color = z.color
21. if y-original-color == BLACK
22.      RB-Delete-Fixup(T, x)
```

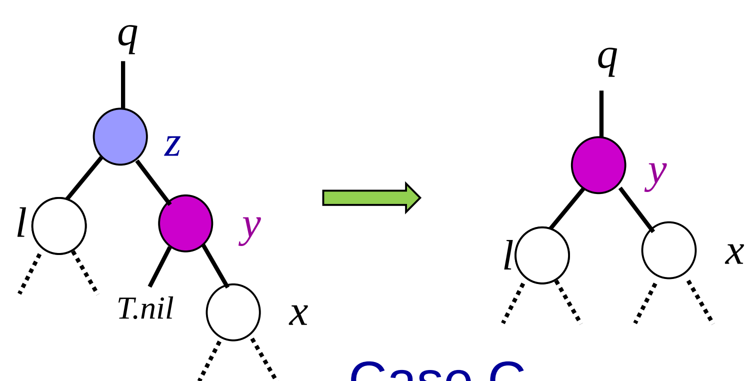


Case A

- If *y* is black, call RB-Delete-Fixup (line 22) to fix the black height
- *z*: the deleted node
- *x*: *y*'s sole child



Case B

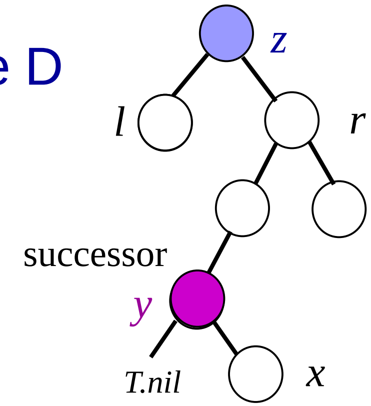


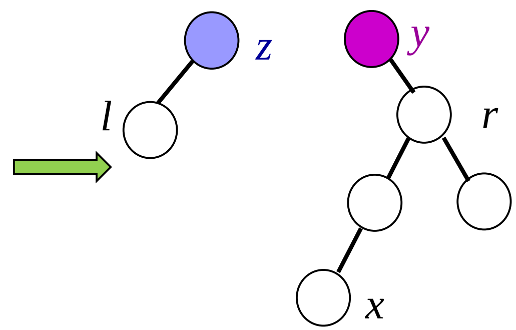
Case C

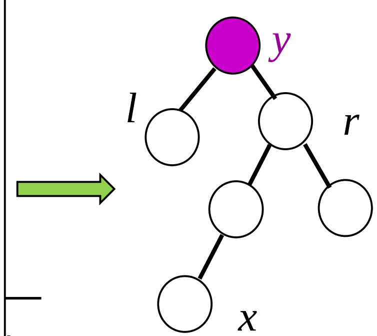
RB-Tree Deletion (cont'd)

```
RB-Delete(T, z)
1.  y = z
2.  y-original-color = y.color
3.  if z.left == T.nil                  // case A
4.      x = z.right
5.      RB-Transplant(T, z, z.right)
6.  elseif z.right == T.nil             // case B
7.      x = z.left
8.      RB-Transplant(T, z, z.left)
9.  else y = Tree-Minimum(z.right)
10.     y-original-color = y.color
11.     x = y.right
12.     if y.p == z                     // case C
13.         x.p = y
14.     else RB-Transplant(T, y, y.right)
15.         y.right = z.right     // case D
16.         y.right.p = y
17.      RB-Transplant(T, z, y)  // cases C, D
18.      y.left = z.left
19.      y.left.p = y
20.      y.color = z.color
21. if y-original-color == BLACK
22.      RB-Delete-Fixup(T, x)
```

Case D







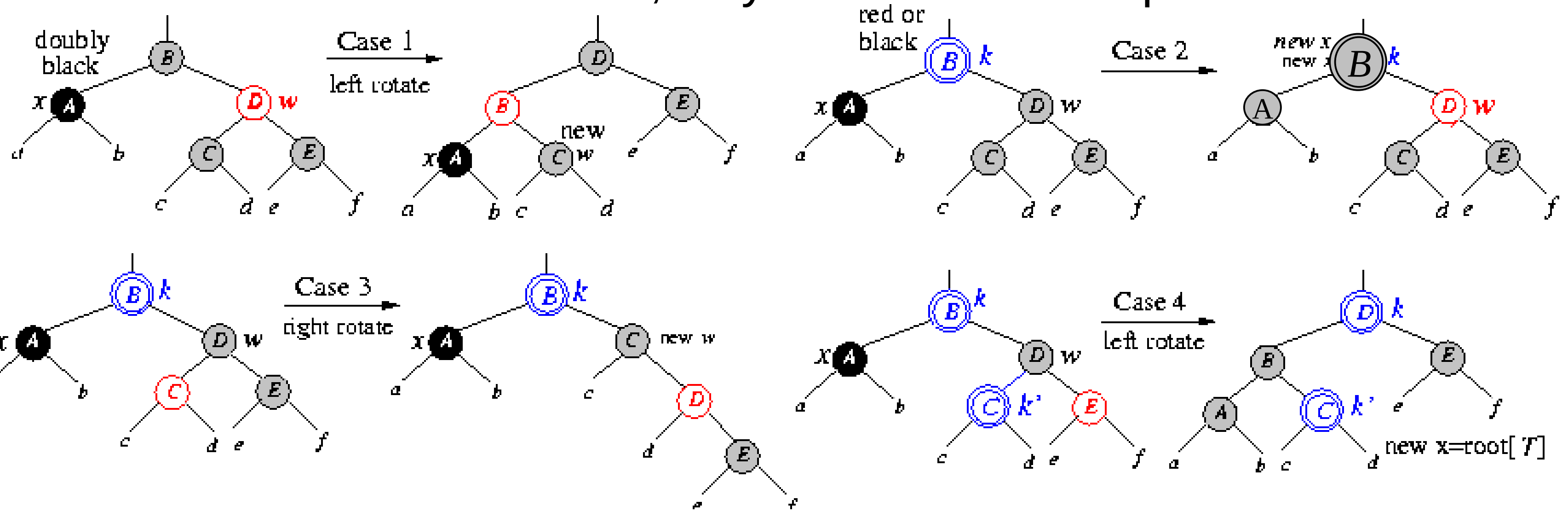
- If *y* is black, call RB-Delete-Fixup (line 22) to fix the black height
- *z*: the deleted node
- *x*: *y*'s sole child

Deletion Color Fixup

- If *y* is black, we must give each of its descendants another black ancestor ⇒ **push *y*'s blackness onto its child *x*.**
- If an appropriate node is red, simply color it black; must restructure, otherwise.
  - Black NIL becomes **doubly black**.
  - Red becomes black.
  - Black becomes doubly black.
- **Goal:** Recolor and restructure the tree so as to get rid of **doubly black**.
- **Key:** Move the extra black up the tree until
  - *x* points to a red node, simply color it black.
  - *x* points to the root, the extra black can simply be removed.
  - Suitable rotations and recolorings can be performed.

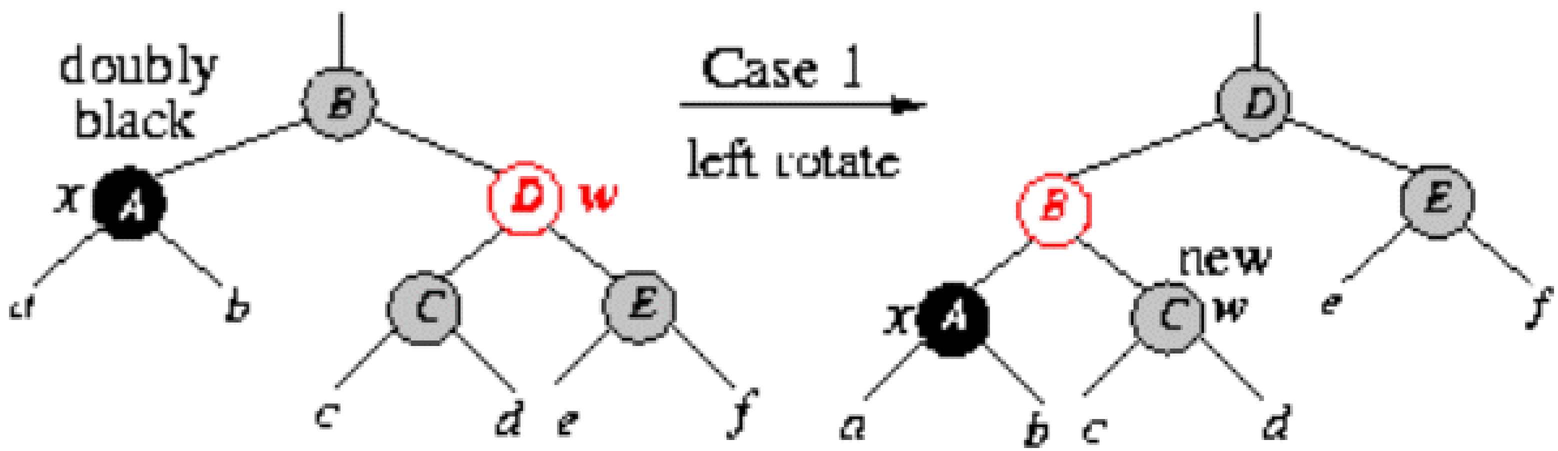
Four Cases for Color Fixup

- **Case 1:** The doubly black node *x* has a red sibling *w*.
- **Case 2:** *x* has a black sibling and two black nephews.
- **Case 3:** *x* has a black sibling, and its left nephew is red and its right nephew is black.
- **Case 4:** *x* has a black sibling, and its right nephew is red (left nephew can be any color).
- The # of black nodes in each path is preserved.
- At most 3 rotations are done; only case 2 can be repeated.



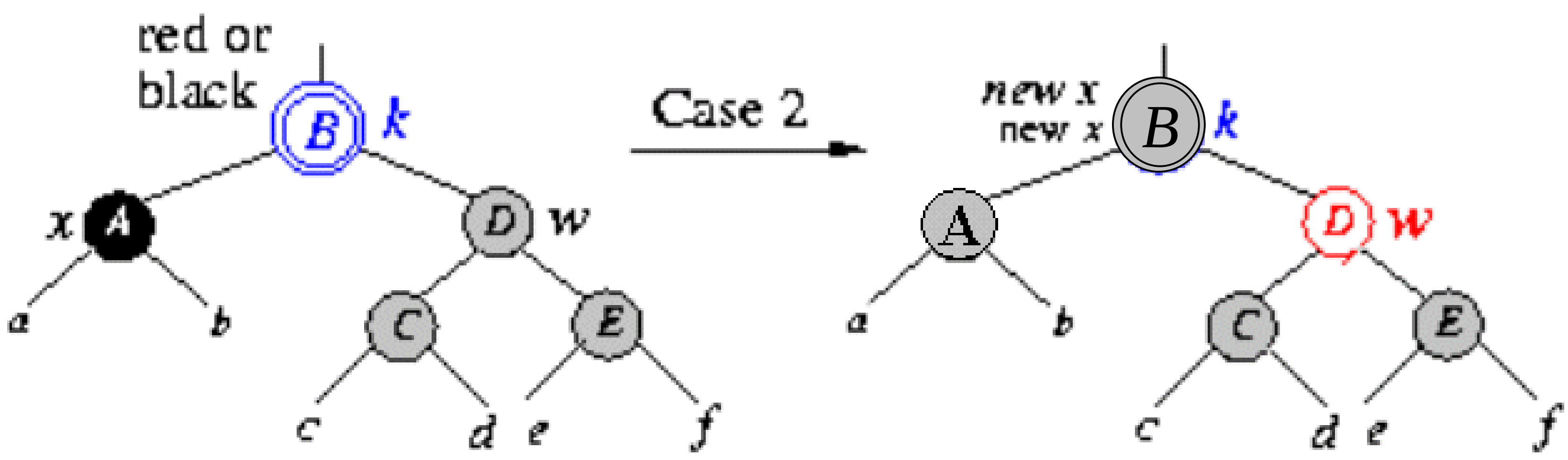
Case 1 for Color Fixup

- **Case 1:** The doubly black node *x* has a red sibling *w*.
- One left rotation around *x*.*p* and constant # of color changes are done.
- The # of black nodes in each path is preserved.
- Converts into case 2, 3, or 4.



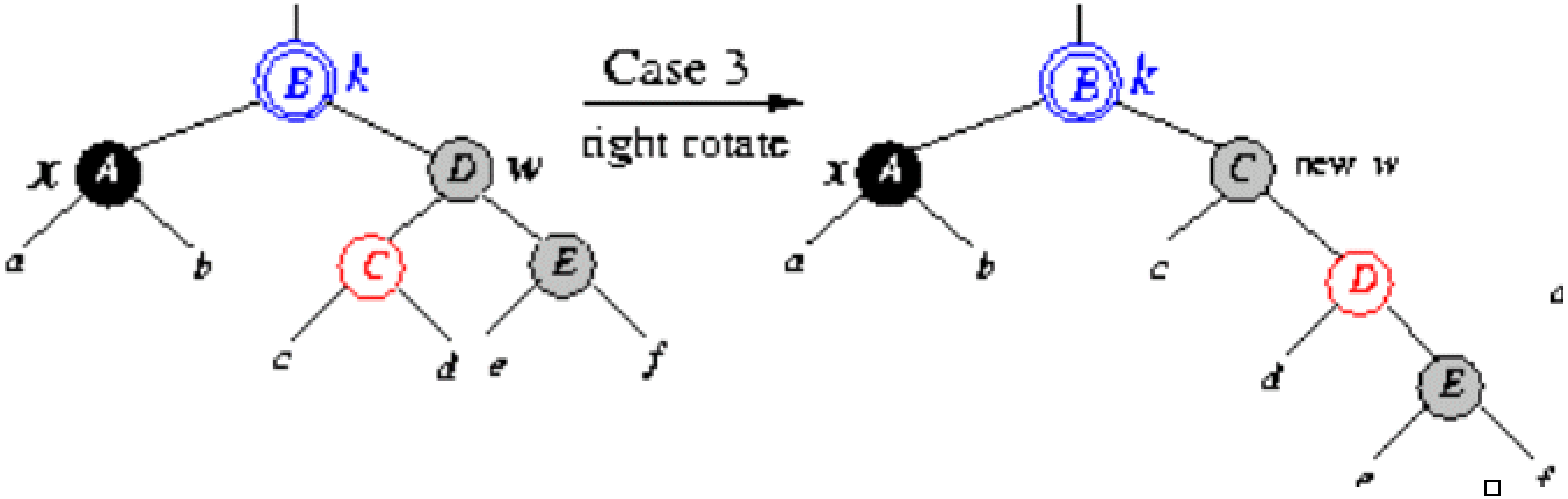
Case 2 for Color Fixup

- **Case 2:** *x* has a black sibling and two black nephews.
- Take off one black from *x* and its sibling, push one black up to *x.p*, and repeat the while loop with *x.p* as the new node *x*.
- The # of black nodes in each path is preserved.
- Perform constant # of color changes and repeat at most *O*(*h*) times. (No rotation!!)



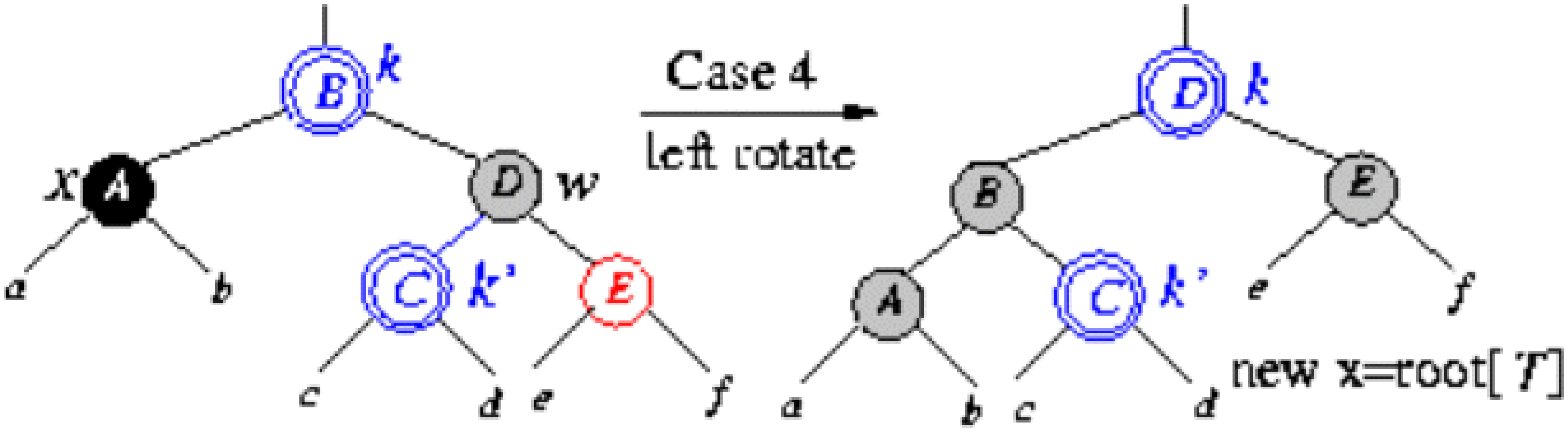
Case 3 for Color Fixup

- **Case 3:** *x* has a black sibling, and its left nephew is red and its right nephew is black.
- Perform a right rotation on *x*'s sibling and constant # of color changes.
- The # of black nodes in each path is preserved.
- Convert into case 4.



Case 4 for Color Fixup

- **Case 4:** *x* has a black sibling, and its right nephew is red (left nephew can be any color).
- Performs a left rotation on *x.p* and constant # of color changes => remove the doubly black on *x*.
- The # of black nodes in each path is preserved.
- Make *x* as the root and terminate the while loop.



Color Fixup for Deletion

```
RB-Delete-Fixup(T,x)
while x ≠ T.root  and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color ==  RED
            w.color = BLACK    // Case 1
            x.p.color = RED    // Case 1
            Left-Rotate(T, x.p)  // Case 1
            w = x.p.right        // Case 1
        If w.left.color == BLACK and w.right.color == BLACK
            w.color = RED      // Case 2
            x = x.p            // Case 2
        else if w.right.color == BLACK
                w.left.color = BLACK    // Case 3
                w.color = RED           // Case 3
                Right-Rotate(T, w)      // Case 3
                w = x.p.right           // Case 3
            w.color = x.p.color      // Case 4
            x.p.color =  BLACK       // Case 4
            w.right.color = BLACK    // Case 4
            Left-Rotate(T, x.p)      // Case 4
            x = T.root
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

```
RB-Delete-Fixup(T,x)
while x ≠ T.root  and x.color == BLACK
      if x == x.p.left
            w = x.p.right
            if w.color ==  RED
                  w.color = BLACK    // Case 1
                  x.p.color = RED    // Case 1
                  Left-Rotate(T, x.p) // Case 1
                  w = x.p.right      // Case 1
            If w.left.color == BLACK and w.right.color == BLACK
                  w.color = RED      // Case 2
                  x = x.p            // Case 2
            else
            {
                  if w.right.color == BLACK
                  {
                        w.left.color = BLACK     // Case 3
                        w.color = RED            // Case 3
                        Right-Rotate(T, w)       // Case 3
                        w = x.p.right            // Case 3
                  }
                  w.color = x.p.color      // Case 4
                  x.p.color =  BLACK       // Case 4
                  w.right.color = BLACK    // Case 4
                  Left-Rotate(T, x.p)      // Case 4
                  x = T.root
            }
      else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Conclusion: Red-Black Trees

- Red-black trees are balanced binary search trees.
- All dictionary operations (Minimum, Maximum, Search, Successor, Predecessor, Insert, Delete) can be performed in $O(\lg n)$ time.
- At most 3 rotations are done to rebalance.

- Visualization tool for the red-black tree (different deletion operation from our lecture): https://www.cs.usfca.edu/~galles/visualization/RedBlack.html