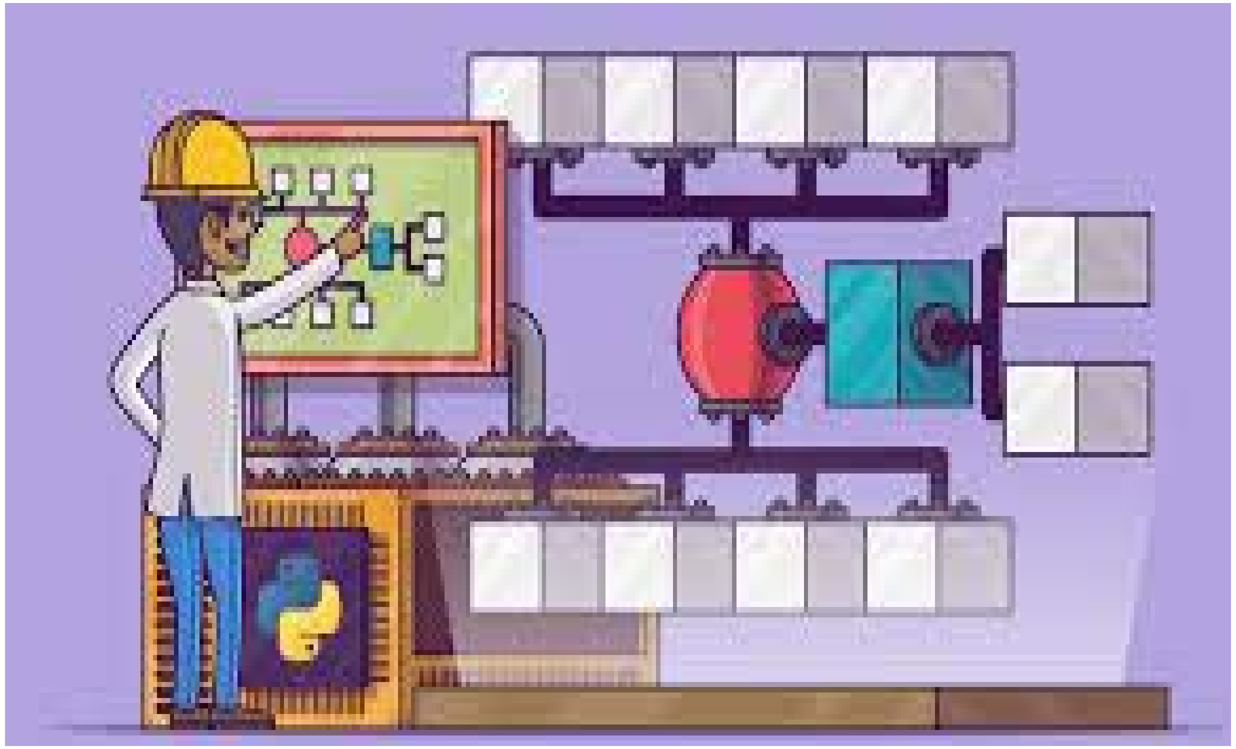
ساختمان داده ها



مدرس:
سمانه حسینی سمنانی

دانشگاه صنعتی اصفهان- دانشکده برق و کامپیوتر

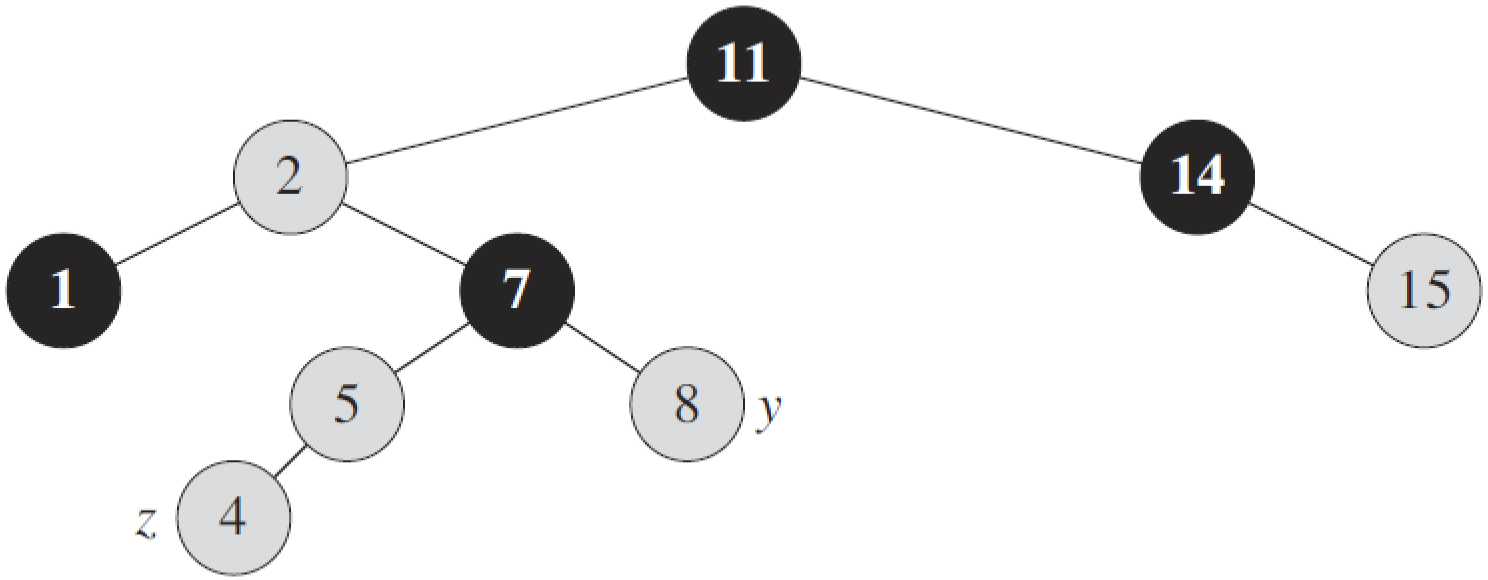
درخت ها

- مفاهیم اولیه
- پیمایش درخت
- درخت دودویی معادل
- پیاده سازی درخت
- درخت جستجوی دودویی
- درخت عبارت
- Heap tree (هرم بیشینه)

درخت ها

- Red-black tree

Insertion in Red-black tree



Insertion of node Z in binary search tree and then color Z red. Why red?

To guarantee that the red-black properties are preserved,
we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations.

Insertion in Red-black tree

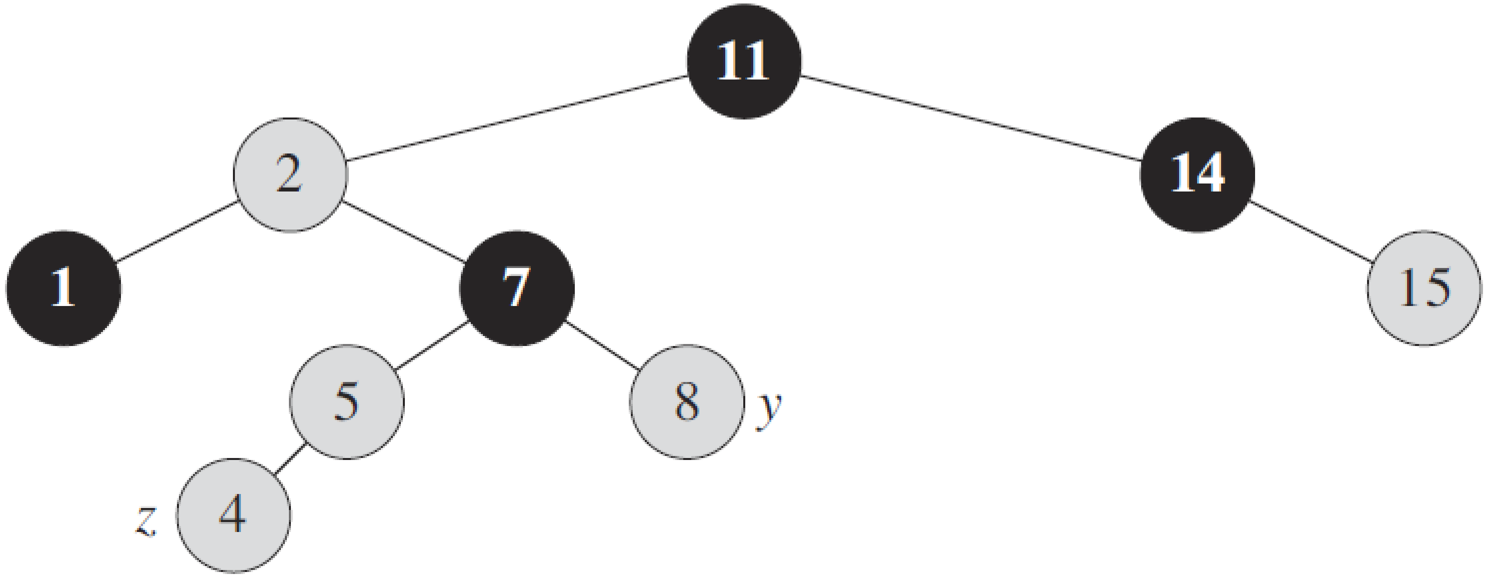
RB-INSERT(T, z)

```
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
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)
```

almost the same as Insert in binary tree

```
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
```

Insertion in Red-black tree



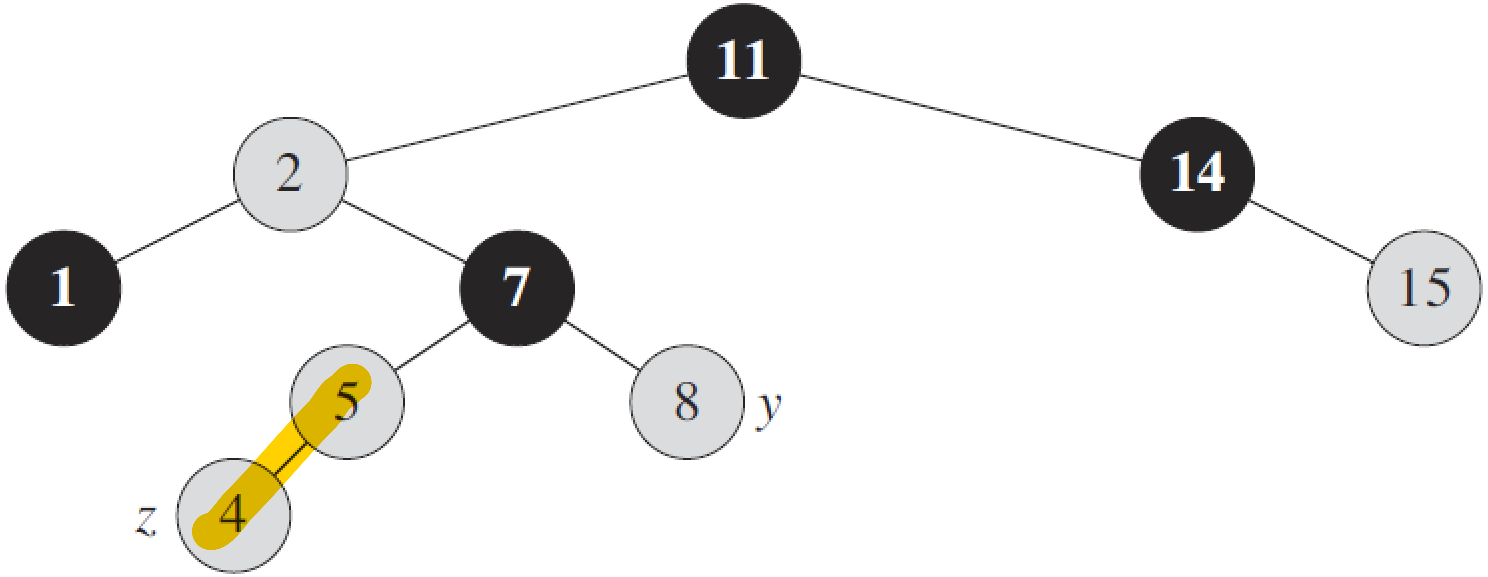
Because both z and its parent z.*p* are red, a violation of property 4 occurs

Red-black tree

- A red-black tree is a binary tree that satisfies the following red-black 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, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Insertion in Red-black tree

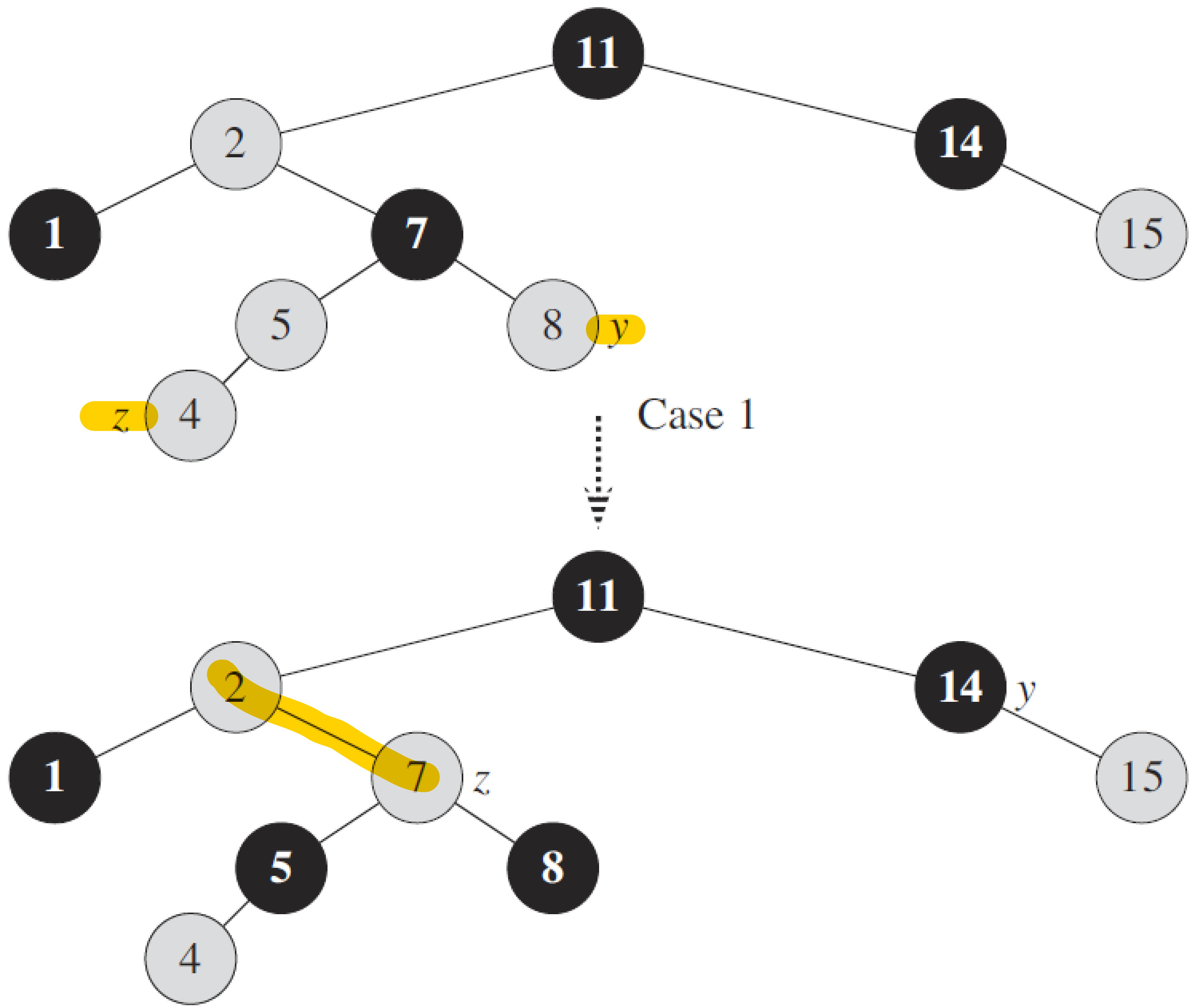


Because both z and its parent z.*p* are red, a violation of property 4 occurs

Since z’s uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**.

RB-INSERT-FIXUP(T, z)

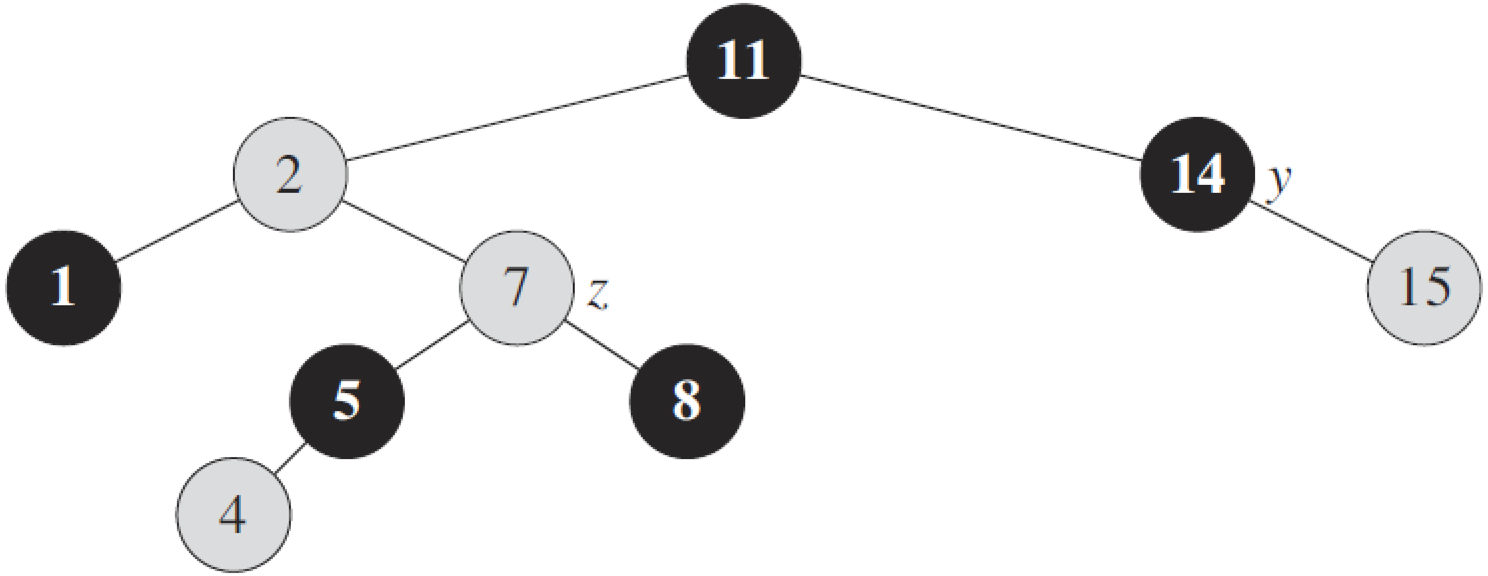
```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                                   // case 1
            y.color = BLACK                                     // case 1
            z.p.p.color = RED                                   // 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
```


11
2
14
1
7
15
5
8
y
z
4
Case 1
11
2
14
y
1
7
z
15
5
8
4

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
```

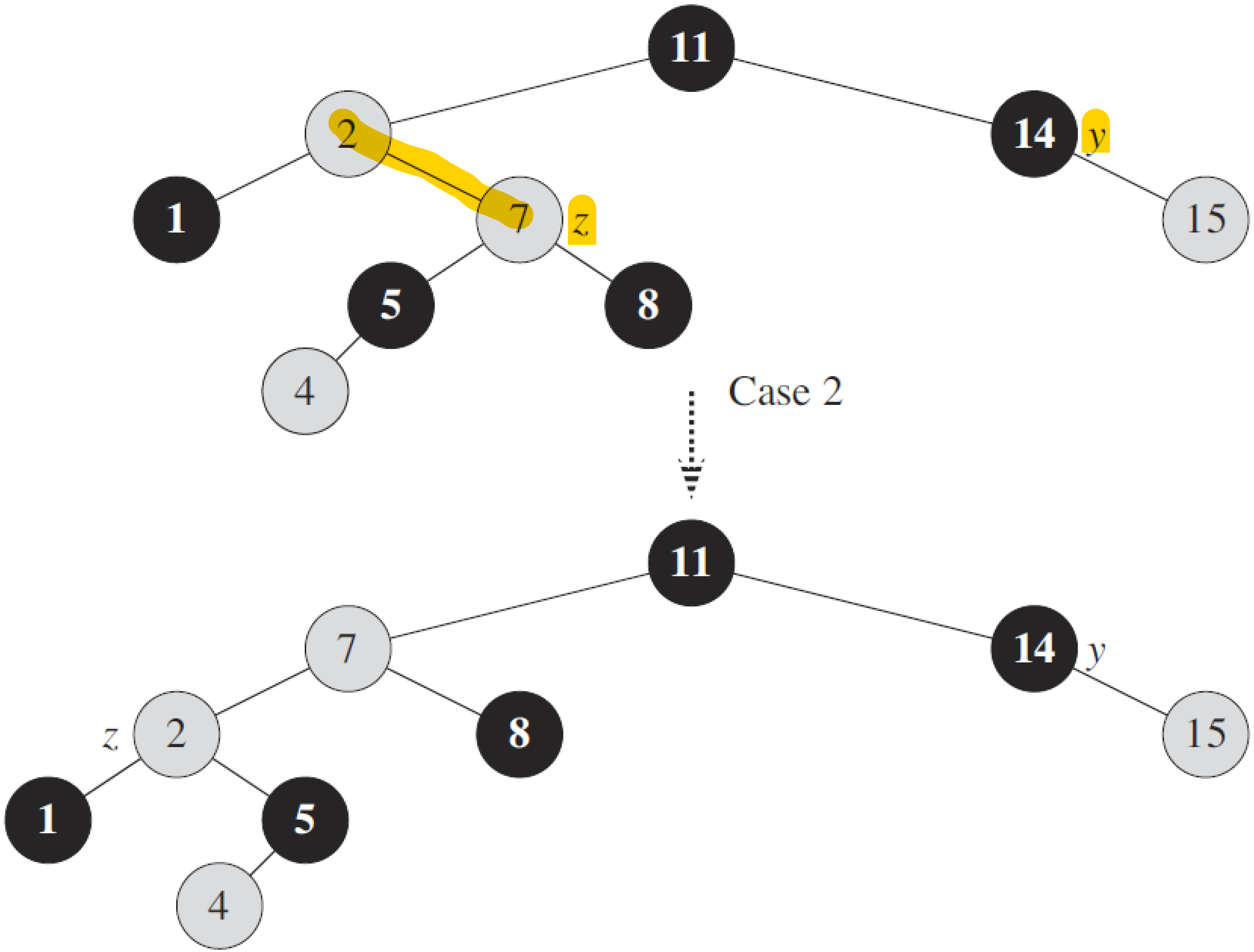
Insertion in Red-black tree



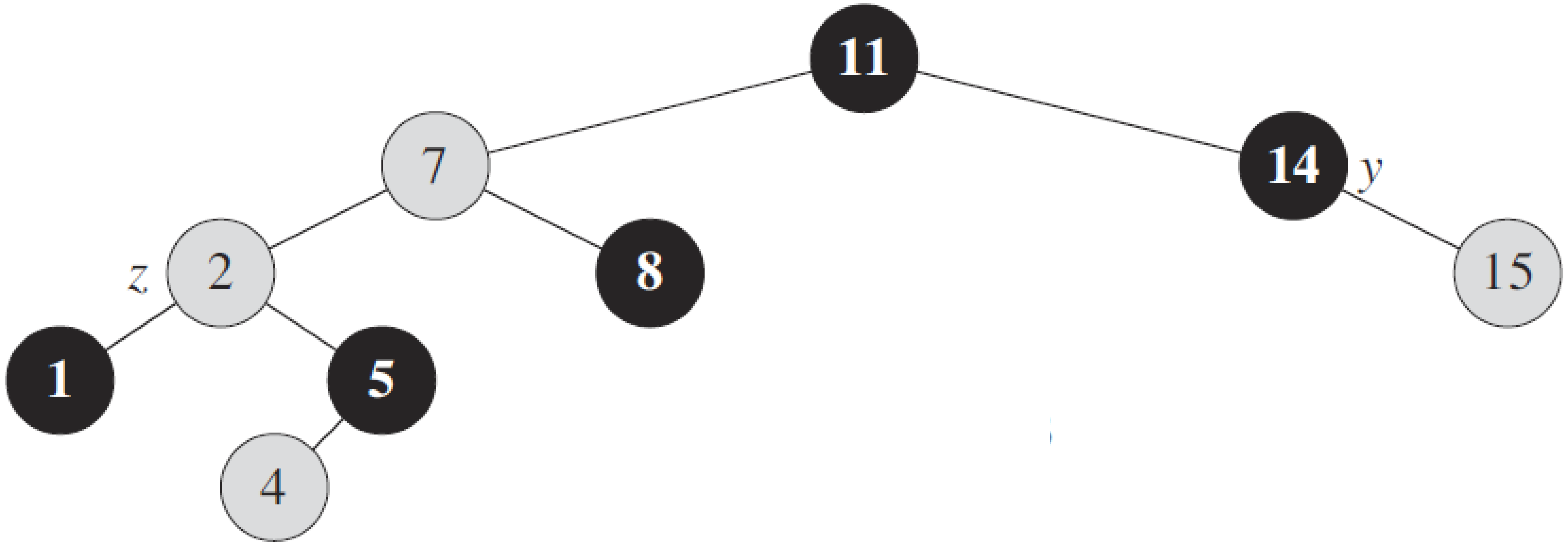
z and its parent are both red, but z's uncle y is black.
Since z is the right child of z.*p*, case 2 applies.
We perform a left rotation, and the tree that results is shown in **(c)**.

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
```

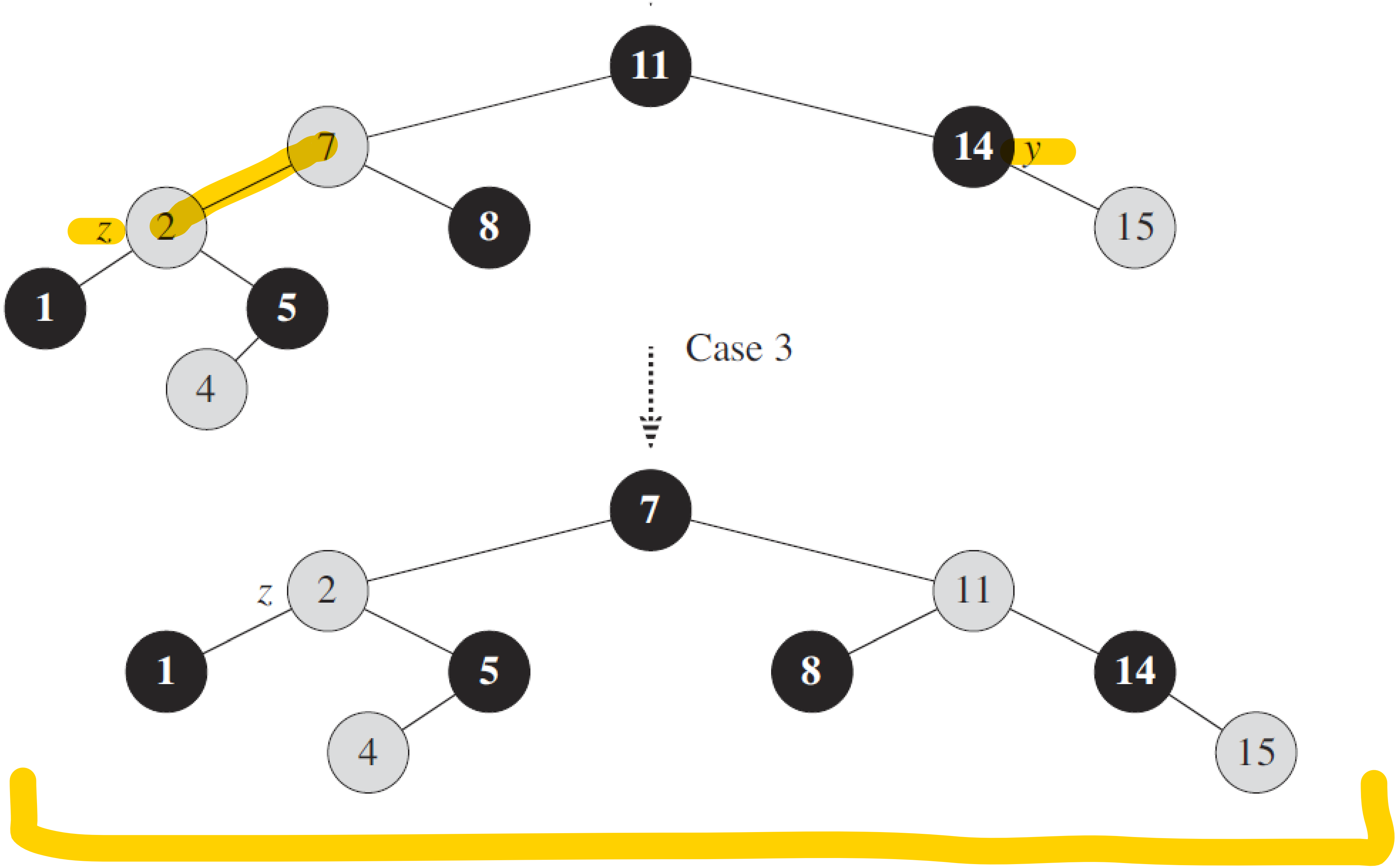

11
2
14
y
1
7
z
15
5
8
4
Case 2
11
7
14
y
z
2
8
15
1
5
4

```
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
```



Now, z is the left child of its parent, and case 3 applies.
Recoloring and right rotation yield the tree in **(d)**,
which is a legal red-black tree.

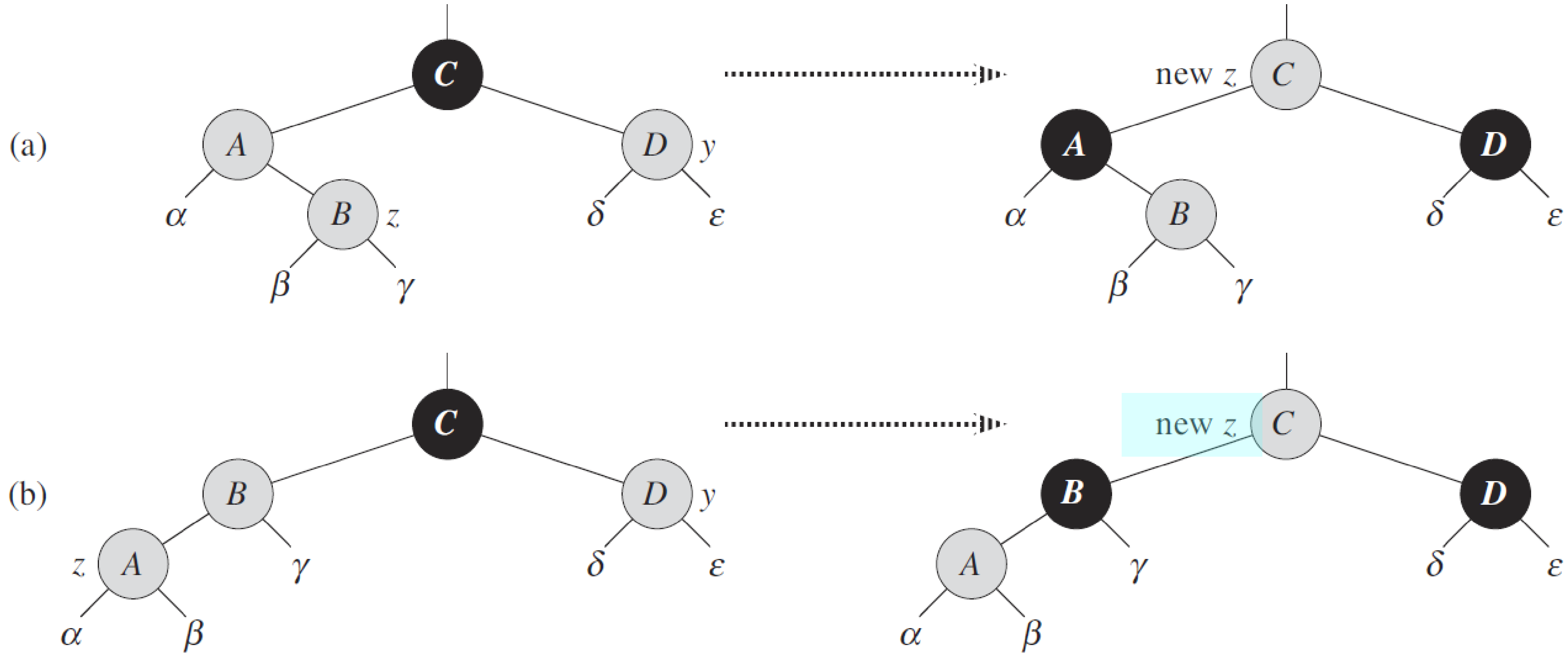
```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                    // case 1
            y.color = BLACK                      // case 1
            z.p.p.color = RED                    // case 1
            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
```



11
7
14
y
z
2
8
15
1
5
4
Case 3
7
z
2
11
1
5
8
14
4
15

```
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
```

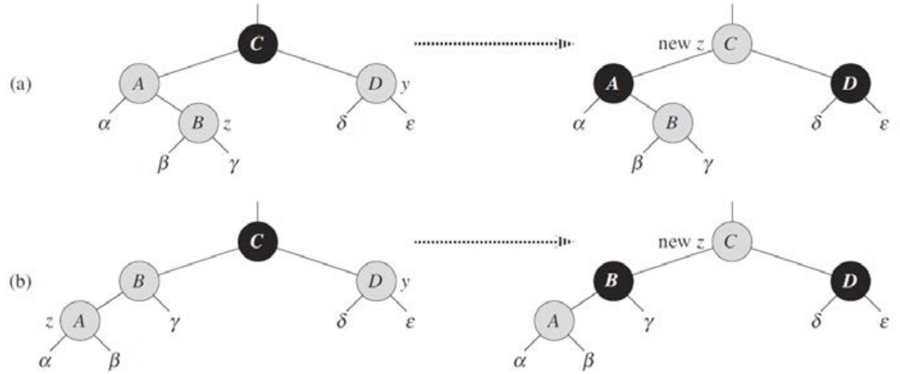
Case 1: z's uncle y is red



سمانه حسینی سمنانی
هیات علمی دانشکده برق و کامپیوتر- دانشگاه صنعتی اصفهان

Case 1: z’s uncle y is red

$$z' = z.p.p$$



a. Because this iteration colors $z.p.p$ red, node z' is red at the start of the next iteration.

b. The node $z'.p$ is $z.p.p.p$ in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.

c. We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.

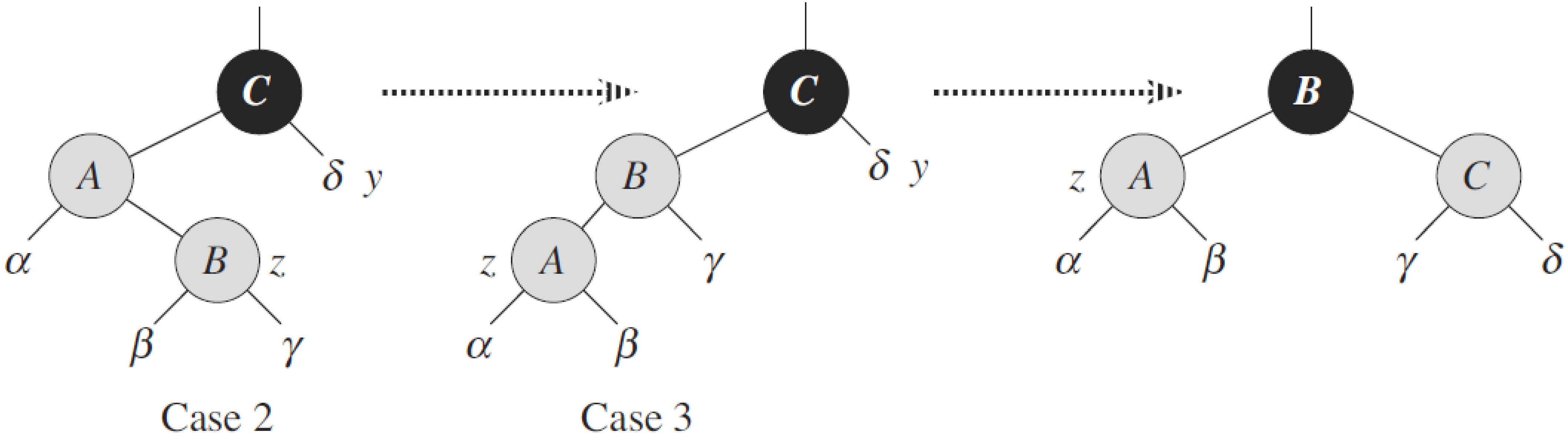
Red-black tree

- A red-black tree is a binary tree that satisfies the following red-black properties:

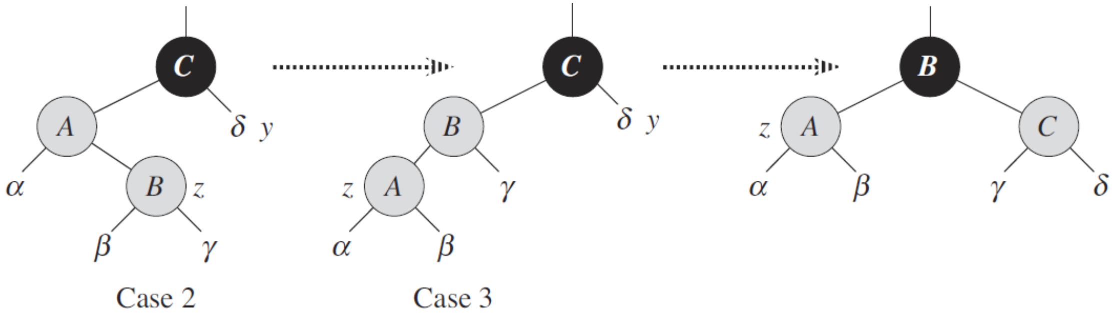
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Case 2 and Case 3

- Case 2: z's uncle y is black and z is a right child
- Case 3: z's uncle y is black and z is a left child.



Case 1: z’s uncle y is red



a. Case 2 makes z point to $z.p$, which is red. No further change to z or its color occurs in cases 2 and 3.

b. Case 3 makes $z.p$ black, so that if $z.p$ is the root at the start of the next iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

Since node z is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

What is the running time of RB-INSERT?

RB-INSERT(T, z)

```
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
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)
```

$O(\lg n)$?

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
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
                z = z.p
                LEFT-ROTATE(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

the **while** loop repeats only if case 1 occurs,
and then the pointer z moves two levels up the tree.

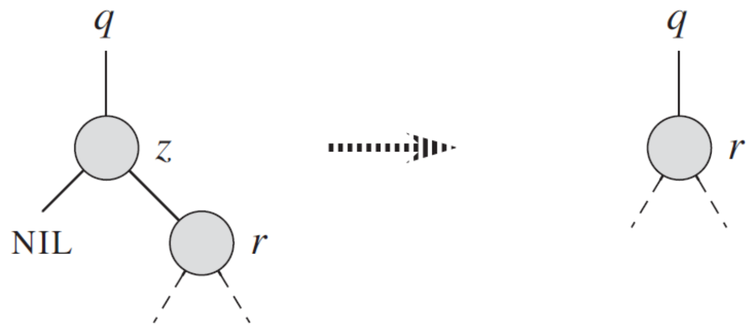
$O(\lg n)$

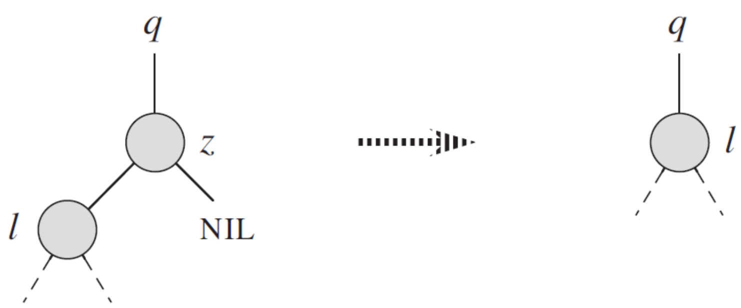
Deletion in Binary Tree

```
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
```

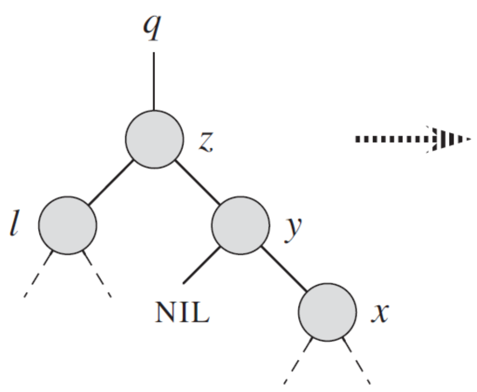
```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

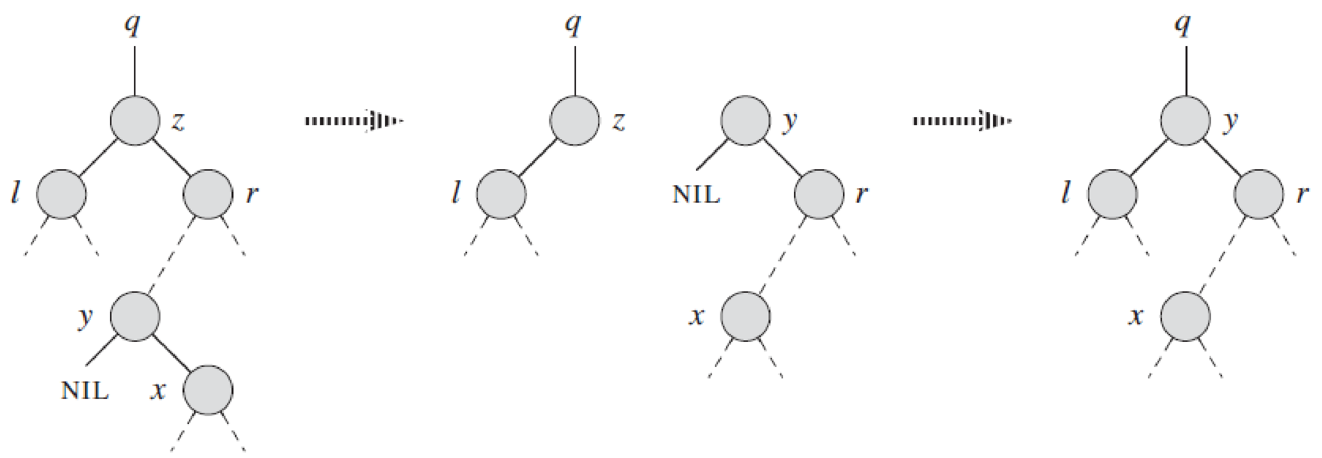
Deletion in Binary Tree





Deletion in Binary Tree





Deletion in Red-Black Tree

```
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
```

```
RB-DELETE(T, z)

y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

Deletion in Red-Black Tree

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

Deletion in Red-Black Tree

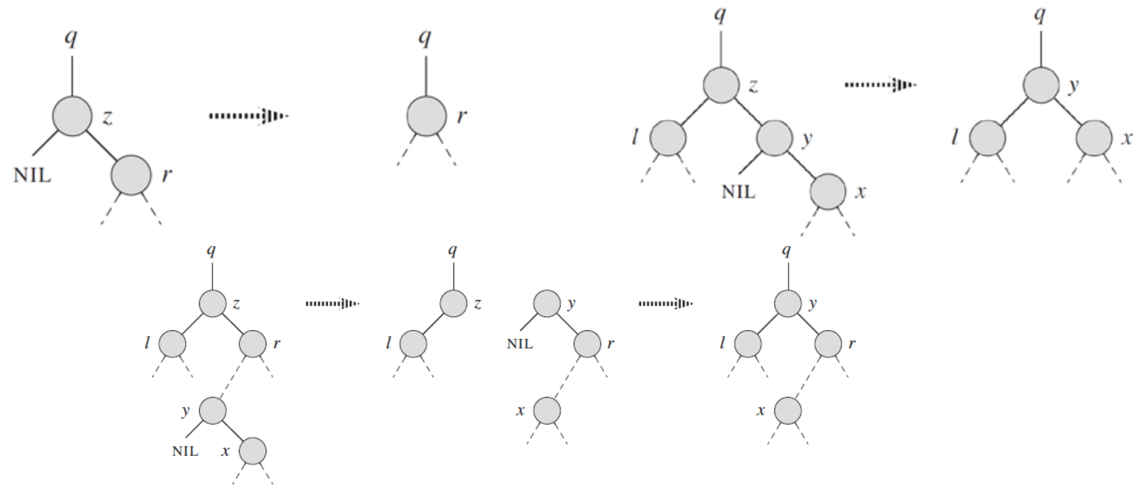
RB-DELETE(T, z)

```
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

We maintain node y as the node either removed from the tree or moved within the tree.

Because node y's color might change, the variable *y-original-color* stores y's color before any changes occur

we keep track of the node x that moves into node y's original position.

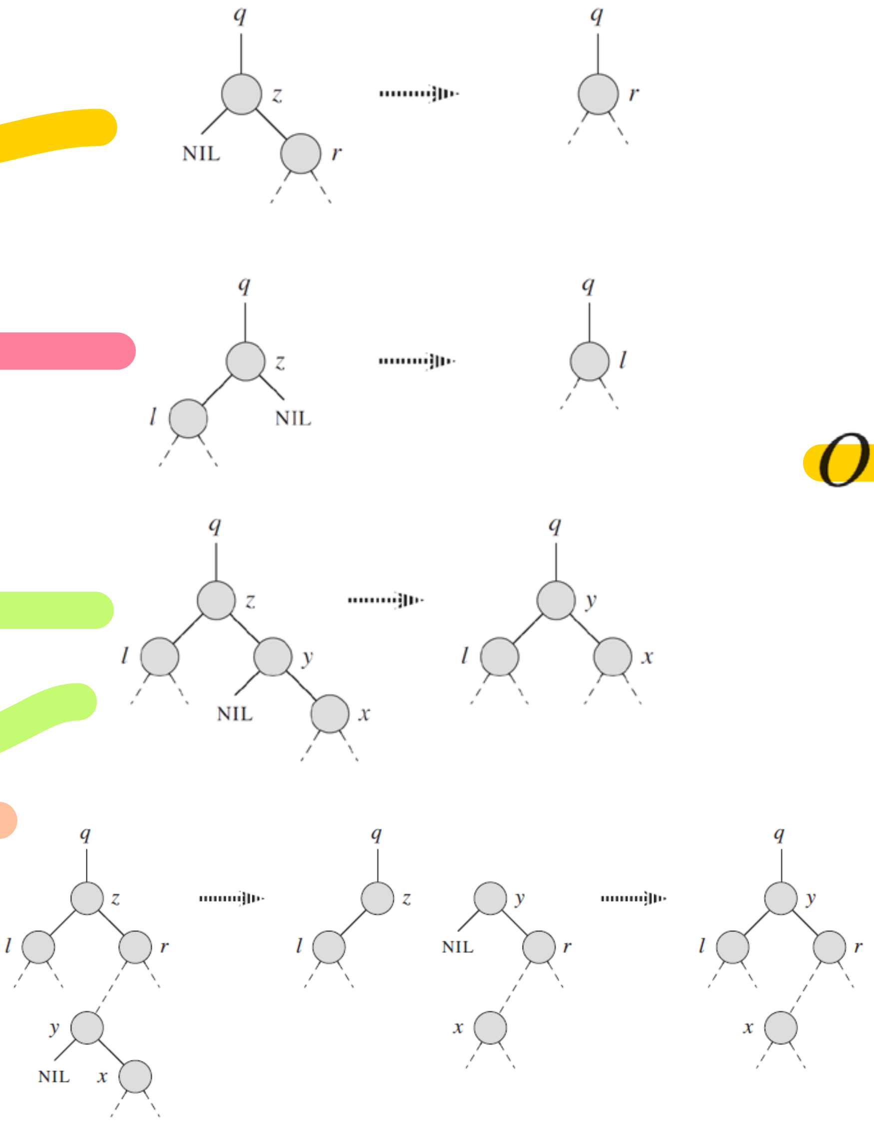


node y moves into node z's original position in the red-black tree

if node y was black, we might have introduced one or more violations of the red-black properties, and so we call RB-DELETE-FIXUP to restore the red-black properties.

Deletion in Red-Black Tree

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```



$O(\lg n)$

Deletion in Red-Black Tree

- If y was red, the red-black properties still hold when y is removed or moved, for the following reasons:

1. No black-heights in the tree have changed.
2. No red nodes have been made adjacent. Because y takes z's place in the tree, along with z's color, we cannot have two adjacent red nodes at y's new position in the tree. In addition, if y was not z's right child, then y's original right child x replaces y in the tree. If y is red, then x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
3. Since y could not have been the root if it was red, the root remains black.

Deletion in Red-Black Tree

- If node y was black, three problems may arise:

1. If y had been the root and a red child of y becomes the new root, we have violated property 2.
2. If both x and x.p are red, then we have violated property 4.
3. moving y within the tree causes any simple path that previously contained y to have one fewer black node. Thus, property 5 is now violated by any ancestor of y in the tree.

RB-DELETE-FIXUP

```
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                               // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```