

Red-Black Trees

Properties of Red-Black Trees

☞ Red-black tree

- A binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK.
- One of many search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worse case.
 - By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than *twice* as long as any other.

Properties of Red-Black Trees

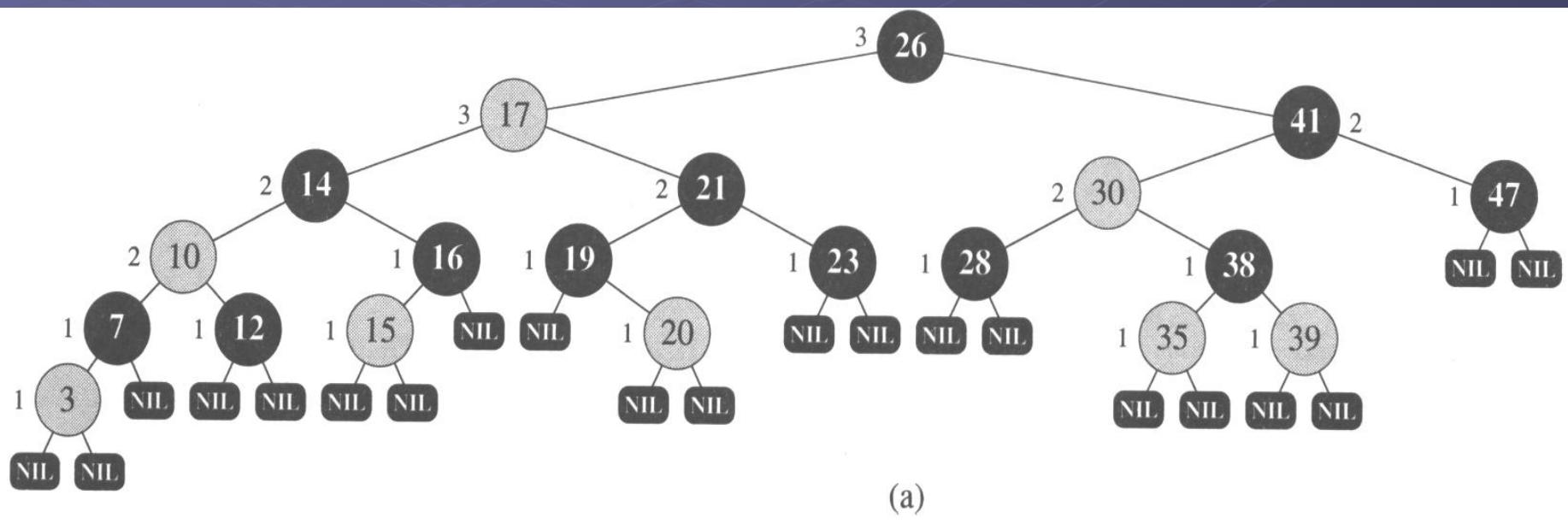
☞ Red-black properties:

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

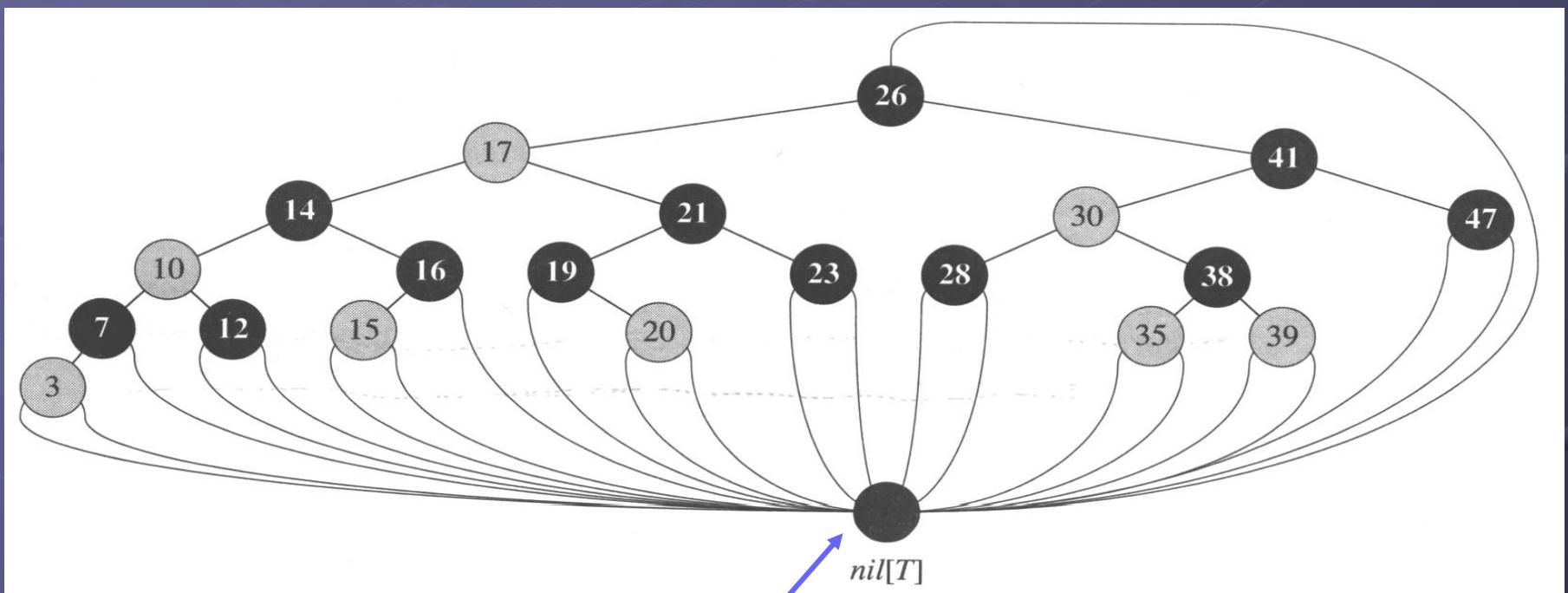
☞ *Black-height* of a node, $bh(x)$

- The number of black nodes on any path from, but not including, a node x down to a leaf.

Properties of Red-Black Trees

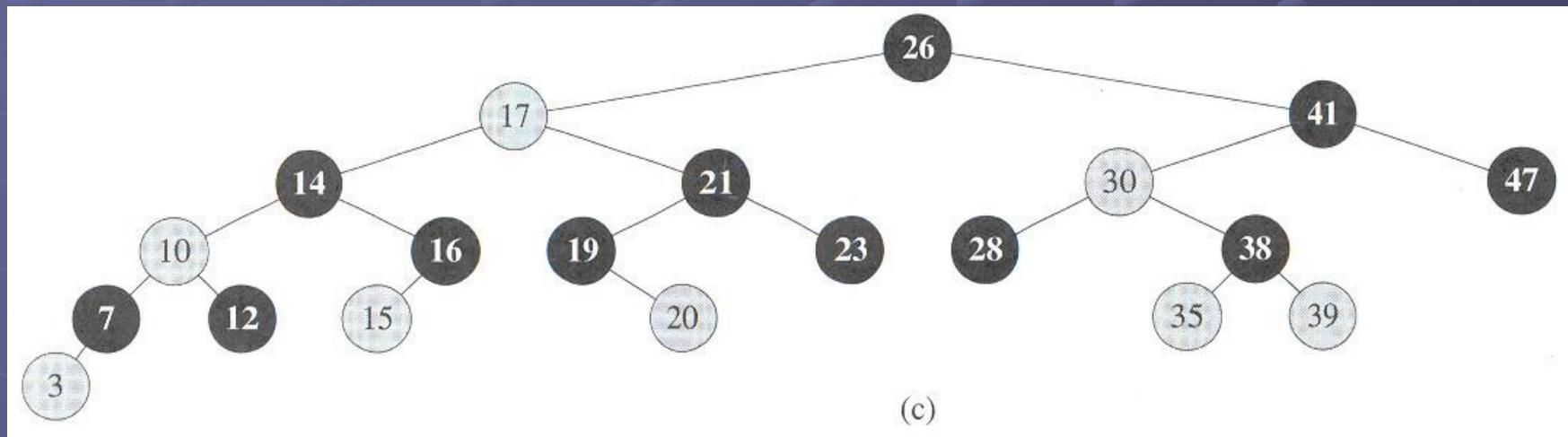


Properties of Red-Black Trees



Sentinel

Properties of Red-Black Trees



Properties of Red-Black Trees

☞ The black-height of a red-black tree

- To be the black-height of its root.

☞ Lemma 13.1

- A red-black tree with n internal nodes has height at most $2\lg(n + 1)$.

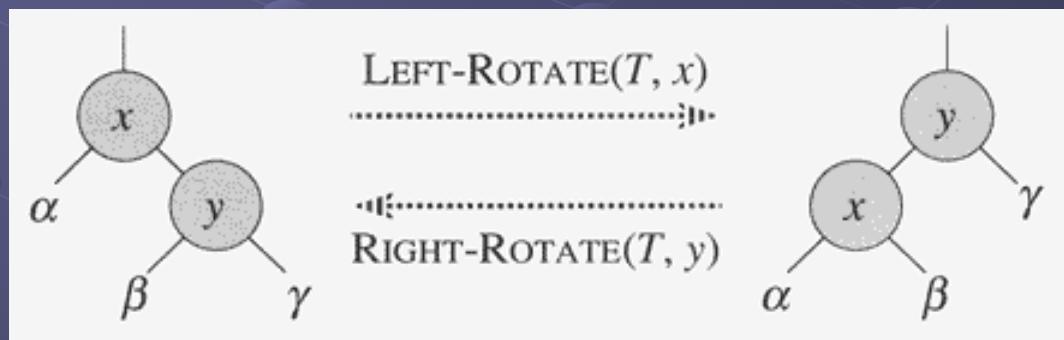
■ Proof:

- First showing the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
- By induction on the height of x .
- The black-height of the root must be at least $h/2$; thus, $n \geq 2^{h/2} - 1$.

Rotations

☞ Rotation

- A local operation preserves the binary-search-tree property.
- Change the pointer structure.
- *Left rotations*
- *Right rotations*

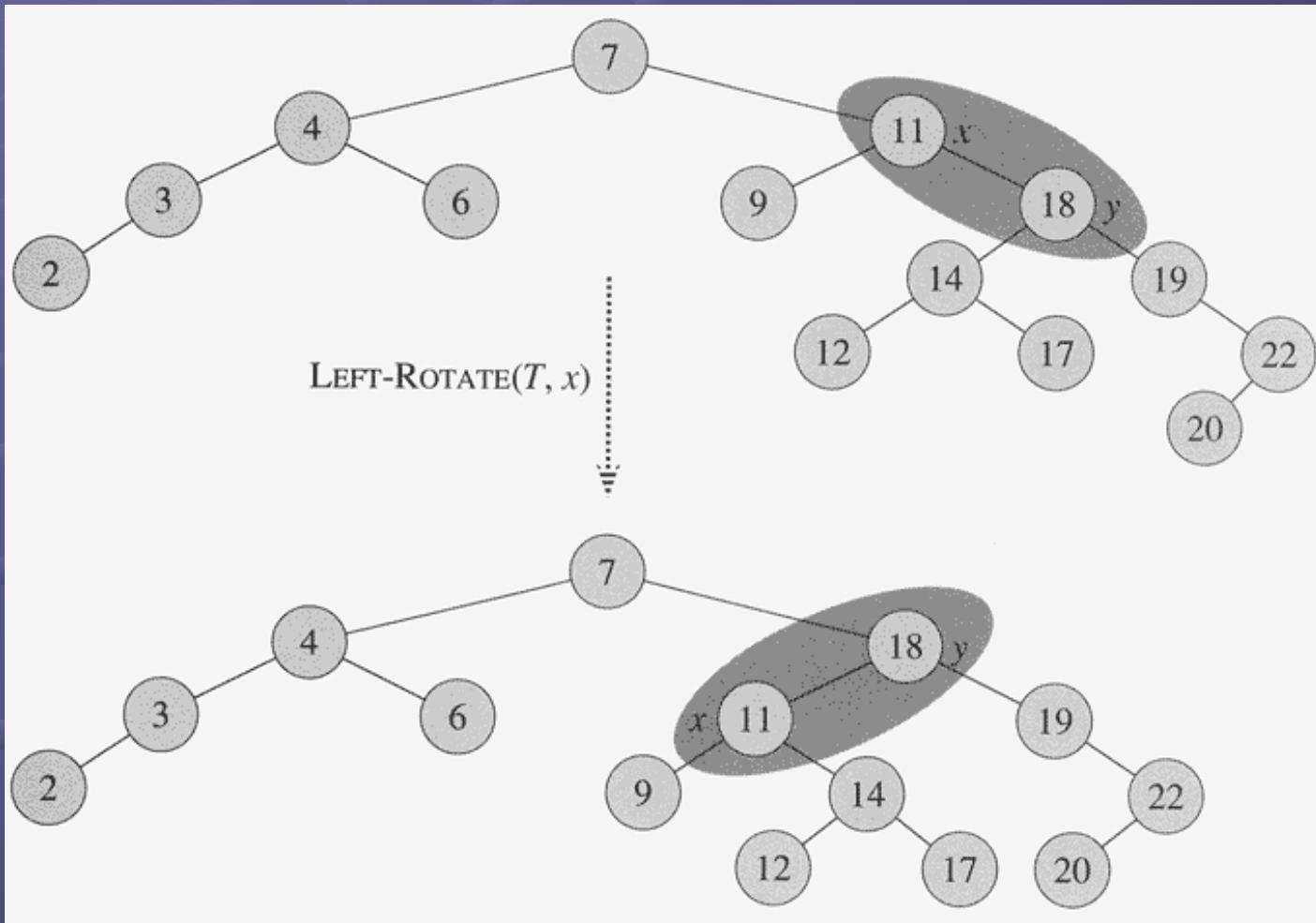


Rotations

LEFT-ROTATE(T, x)

```
1   $y = x.right$                                 // set  $y$ 
2   $x.right = y.left$                             // turn  $y$ 's left subtree into  $x$ 's right subtree
3  if  $y.left \neq T.nil$ 
4       $y.left.p = x$ 
5   $y.p = x.p$                                     // link  $x$ 's parent to  $y$ 
6  if  $x.p == T.nil$ 
7       $T.root = y$ 
8  elseif  $x == x.p.left$ 
9       $x.p.left = y$ 
10 else  $x.p.right = y$ 
11  $y.left = x$                                     // put  $x$  on  $y$ 's left
12  $x.p = y$ 
```

Rotations



Insertion

☞ RB-INSERT procedure

- Insert node z into the tree T as if it were an ordinary binary search tree, and then color z red.

```

1    $y = T.nil$ 
2    $x = T.root$ 
3   while  $x \neq T.nil$ 
4        $y = x$ 
5       if  $z.key < x.key$ 
6            $x = x.left$ 
7       else  $x = x.right$ 
8        $z.p = y$ 
9       if  $y == T.nil$ 

```

```

10       $T.root = z$ 
11  elseif  $z.key < y.key$ 
12       $y.left = z$ 
13  else  $y.right = z$ 
14       $z.left = T.nil$ 
15       $z.right = T.nil$ 
16       $z.color = RED$ 
17      RB-INSERT-FIXUP( $T, z$ )

```

Insertion

☞ RB-INSERT-FIXUP procedure

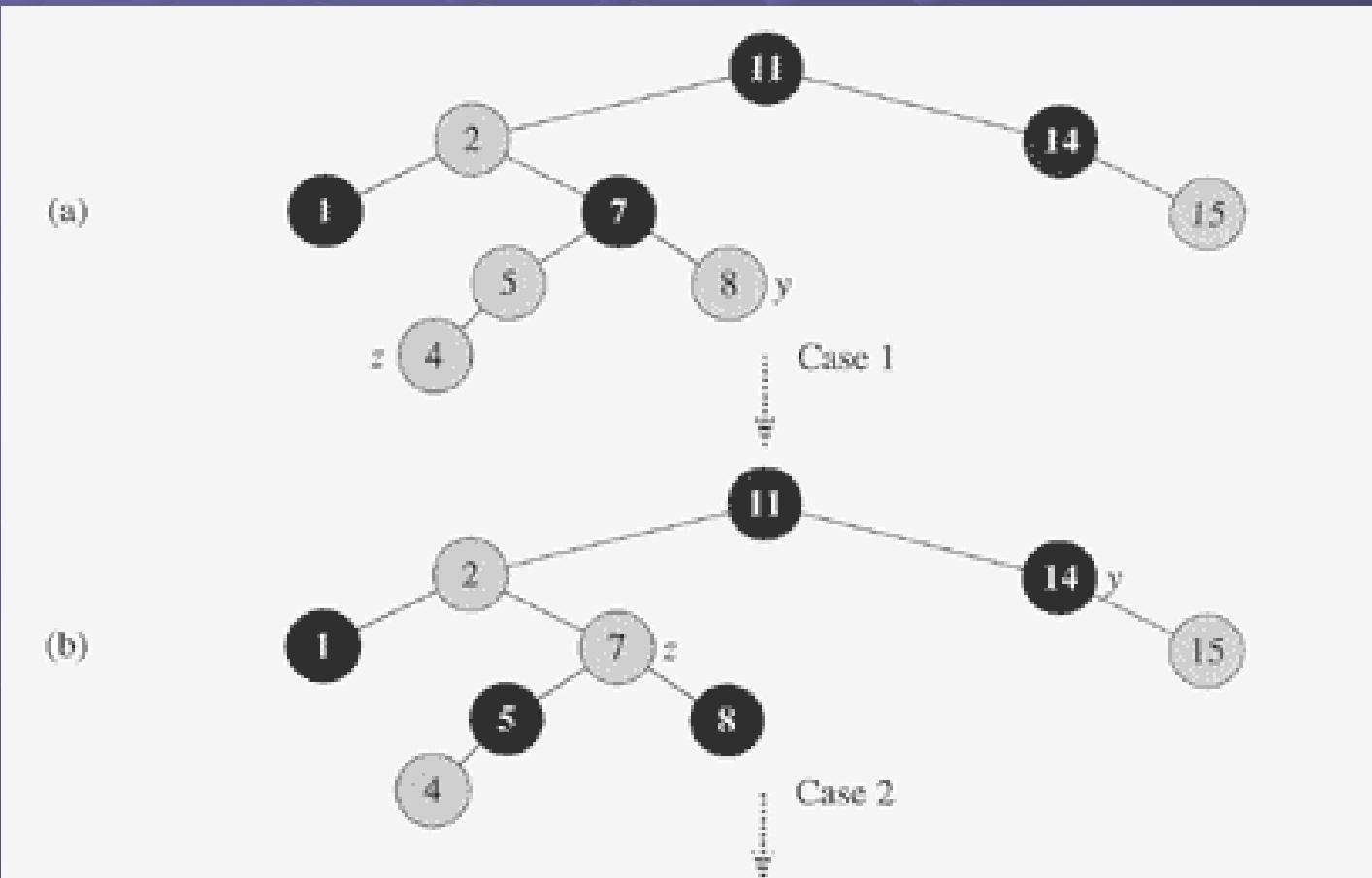
```
RB-INSERT-FIXUP( $T, z$ )
```

```
1  while  $z.p.color == \text{RED}$ 
2      if  $z.p == z.p.p.left$ 
3           $y = z.p.p.right$ 
4          if  $y.color == \text{RED}$ 
5               $z.p.color = \text{BLACK}$            // case 1
6               $y.color = \text{BLACK}$            // case 1
7               $z.p.p.color = \text{RED}$         // case 1
8               $z = z.p.p$                  // case 1
```

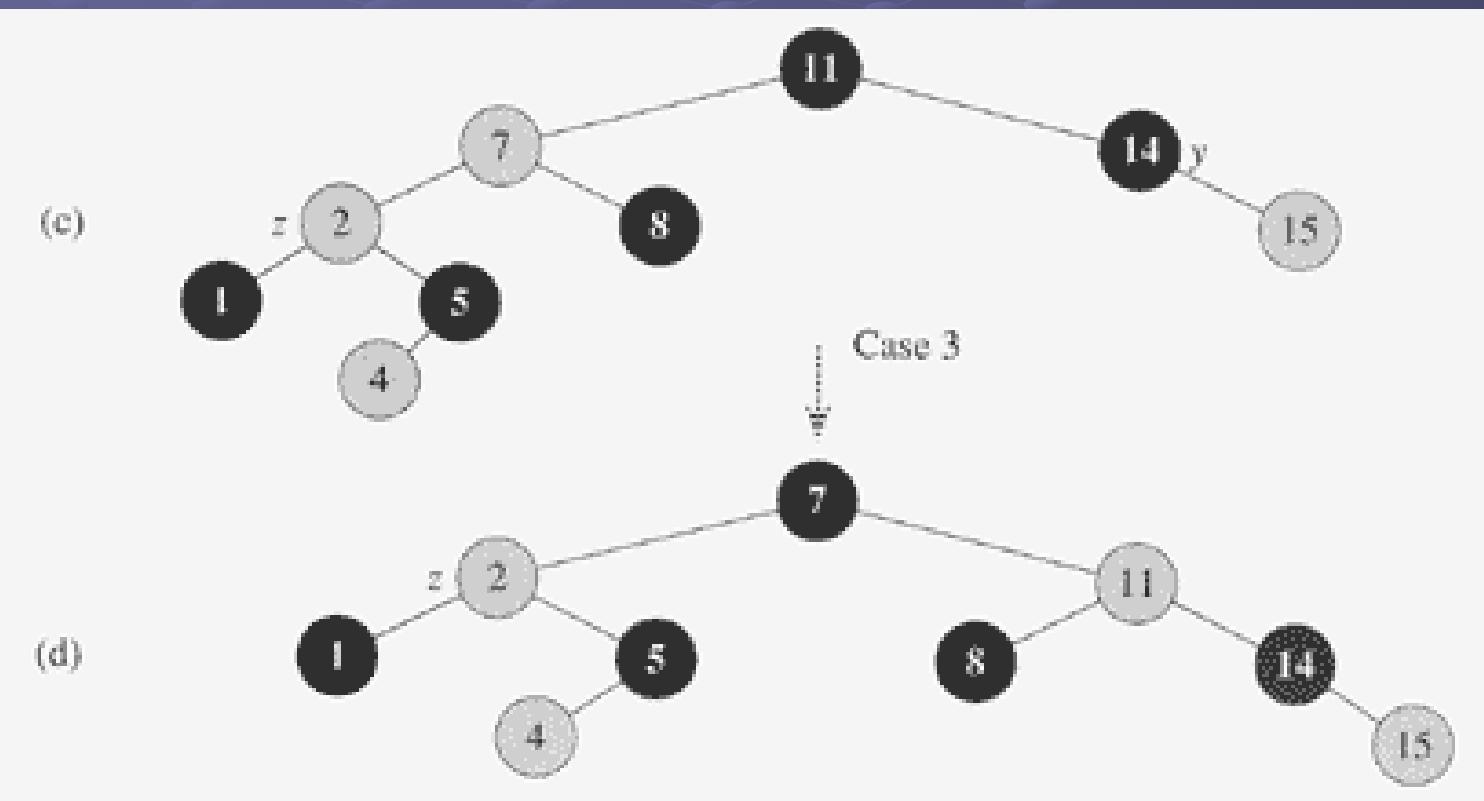
Insertion

```
9     else if  $z == z.p.right$ 
10             $z = z.p$                                 // case 2
11            LEFT-ROTATE( $T, z$ )                // case 2
12             $z.p.color = \text{BLACK}$               // case 3
13             $z.p.p.color = \text{RED}$              // case 3
14            RIGHT-ROTATE( $T, z.p.p$ )        // case 3
15     else (same as then clause
           with “right” and “left” exchanged)
16      $T.root.color = \text{BLACK}$ 
```

Insertion



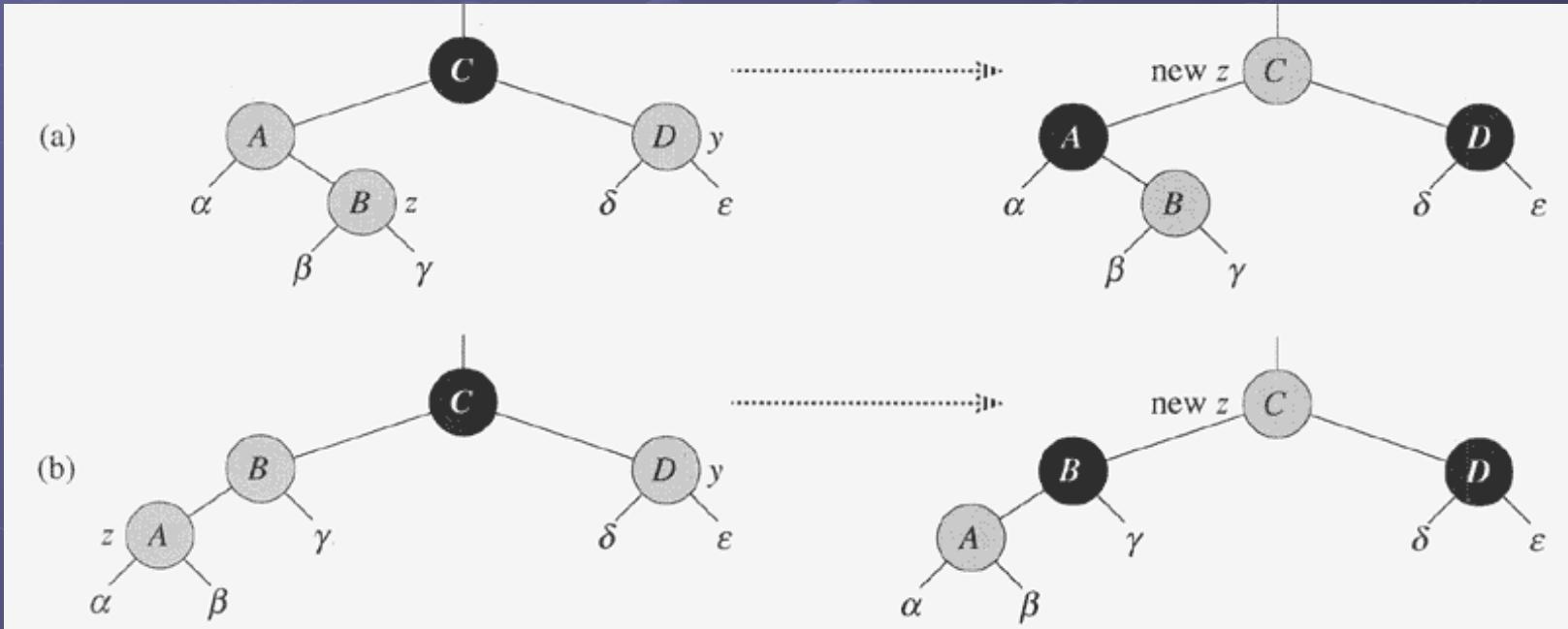
Insertion



Insertion

Case 1:

■ z 's uncle y is red.



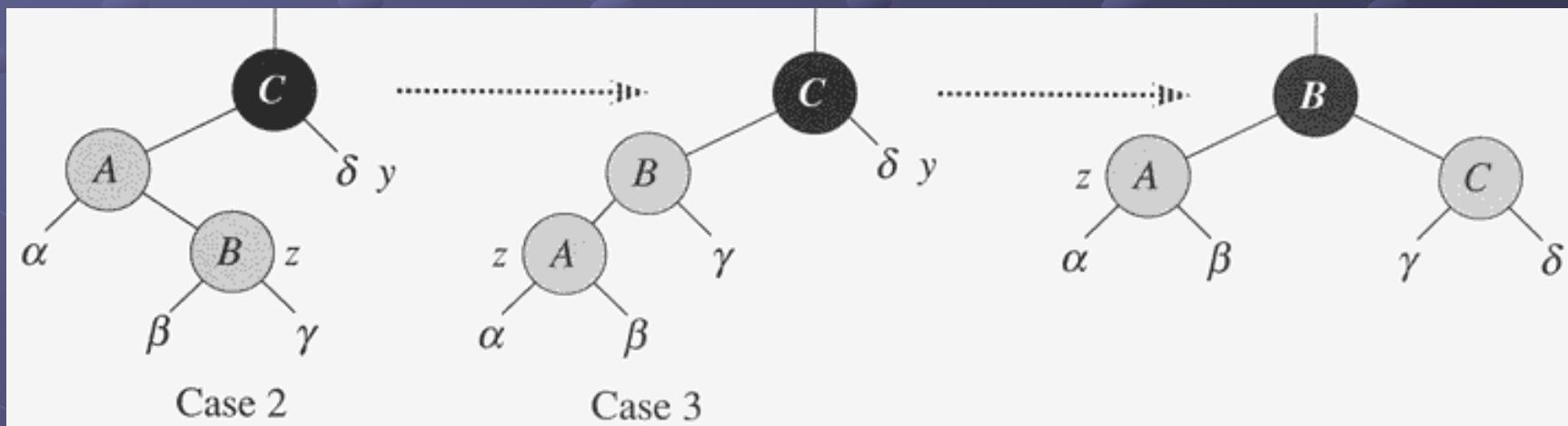
Insertion

☞ 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.



Deletion

☞ RB Transplant

```
RB-TRANSPLANT( $T, u, v$ )
```

```
1  if  $u.p == T.nil$ 
2       $T.root = v$ 
3  elseif  $u == u.p.left$ 
4       $u.p.left = v$ 
5  else  $u.p.right = v$ 
6   $v.p = u.p$ 
```

Deletion

☞ RB-DELETE procedure

■ Takes time $O(\lg n)$.

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4     $x = z.\text{right}$ 
5    RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7     $x = z.\text{left}$ 
8    RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10    $y\text{-original-color} = y.\text{color}$ 
```

Deletion

```
11       $x = y.right$ 
12      if  $y.p == z$ 
13           $x.p = y$ 
14      else RB-TRANSPLANT( $T, y, y.right$ )
15           $y.right = z.right$ 
16           $y.right.p = y$ 
17      RB-TRANSPLANT( $T, z, y$ )
18       $y.left = z.left$ 
19       $y.left.p = y$ 
20       $y.color = z.color$ 
21      if  $y-original-color == \text{BLACK}$ 
22          RB-DELETE-FIXUP( $T, x$ )
```

Deletion

- ☞ If the spliced-out node y in RB-DELETE is *black*, three problems may arise:
 - If y had been the root and a red child of y becomes the new root. (violate property 2)
 - If both x and $x.p$ were red. (violate property 4)
 - y 's removal cause any path that previously contained y to have one fewer black node. (violate property 5)

Deletion

☞ RB-DELETE-FIXUP procedure

- Restores the RB-tree properties when $y.\text{color}$ is *black*.
- ☞ If y is *red*, the red-black properties still hold when y is spliced out:
 - No black-heights in the tree have changed.
 - No red nodes have been made adjacent, and
 - Since y could not have been the root if it was red, the root remains black.
- ☞ The goal of the **while** loop
 - Move the extra black up the tree.

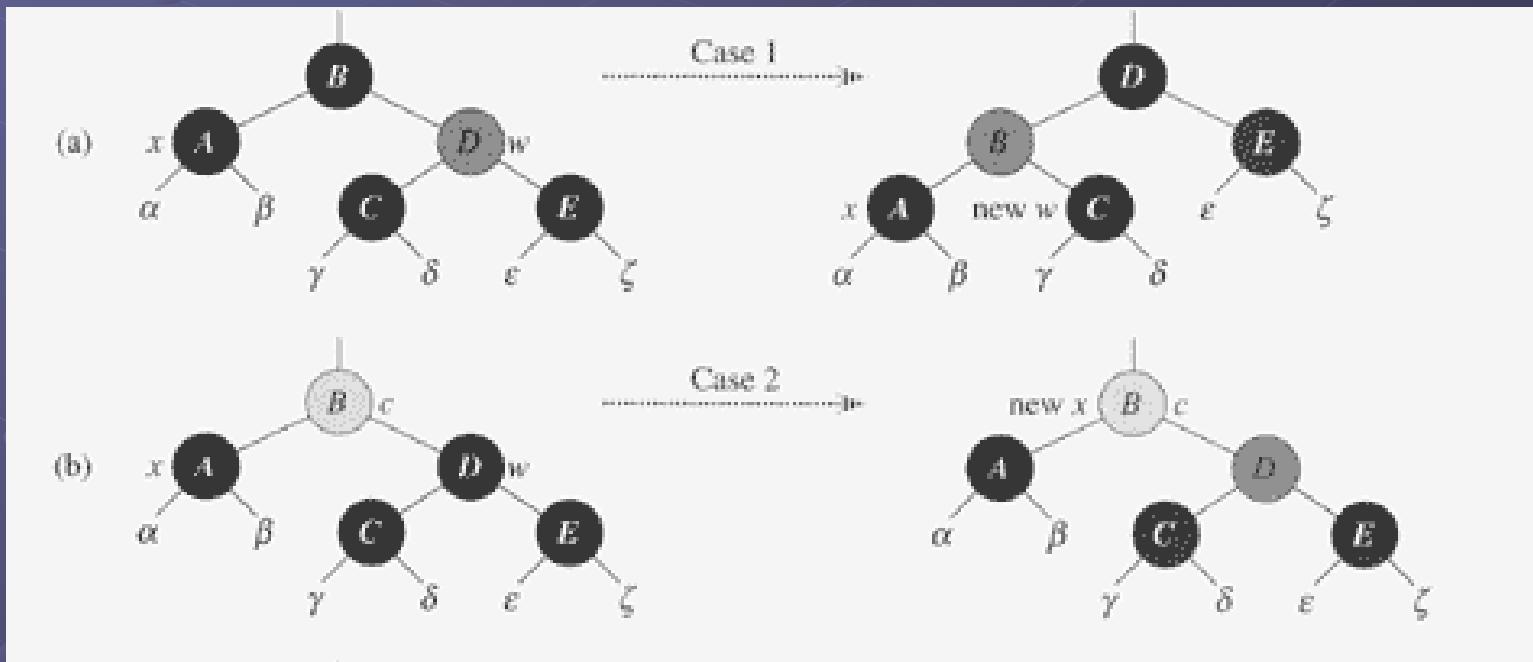
Deletion

RB-DELETE-FIXUP(T, x)

```
1  while  $x \neq T.root$  and  $x.color == \text{BLACK}$ 
2      if  $x == x.p.left$ 
3           $w = x.p.right$ 
4          if  $w.color == \text{RED}$ 
5               $w.color = \text{BLACK}$                                 // case 1
6               $x.p.color = \text{RED}$                             // case 1
7              LEFT-ROTATE( $T, x.p$ )                      // case 1
8               $w = x.p.right$                                 // case 1
9          if  $w.left.color == \text{BLACK}$  and  $w.right.color == \text{BLACK}$ 
10              $w.color = \text{RED}$                                 // case 2
11              $x = x.p$                                     // case 2
```

Deletion

- ☞ Case 1: x 's sibling w is red.
- ☞ Case 2: x 's sibling w is black, and both of w 's children are black.



Deletion

```

12    else if  $w.right.color == \text{BLACK}$ 
13         $w.left.color = \text{BLACK}$                                 // case 3
14         $w.color = \text{RED}$                                  // case 3
15         $\text{RIGHT-ROTATE}(T, w)$                          // case 3
16         $w = x.p.right$                                // case 3
17         $w.color = x.p.color$                            // case 4
18         $x.p.color = \text{BLACK}$                           // case 4
19         $w.right.color = \text{BLACK}$                       // case 4
20         $\text{LEFT-ROTATE}(T, x.p)$                          // case 4
21         $x = T.root$                                   // case 4
22    else (same as then clause with “right” and “left” exchanged)
23     $x.color = \text{BLACK}$ 

```

Deletion

- ☞ Case 3: x 's sibling w is black, and w 's left child is red, and w 's right child is black.
- ☞ Case 4: x 's sibling w is black, and w 's right child is red.

