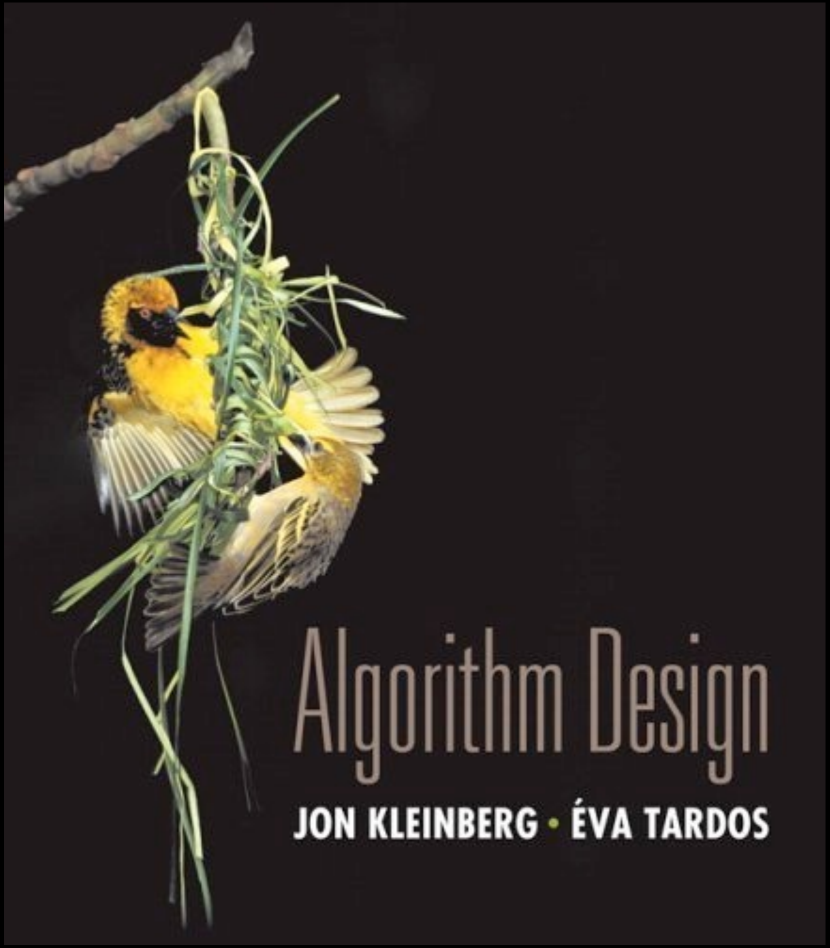
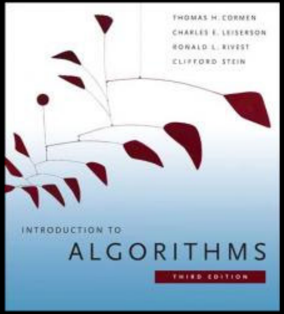


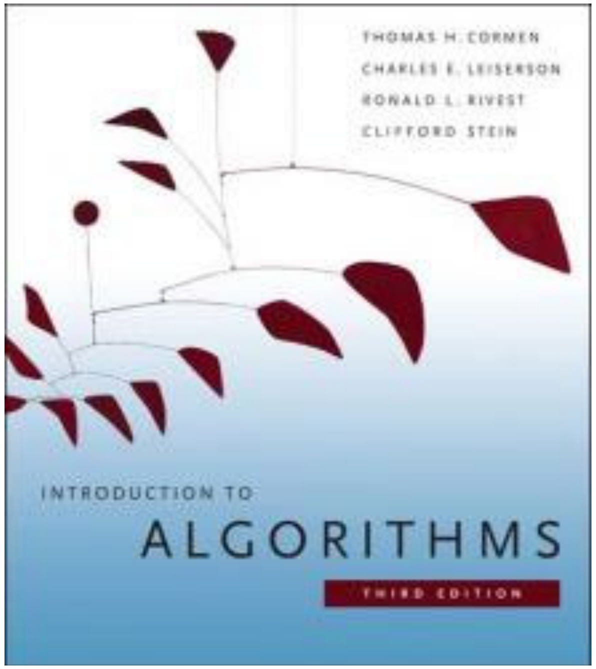
Chapter 3

Graphs

CLRS 12-13



CLRS 12 Binary Search Trees


THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
INTRODUCTION TO
ALGORITHMS
THIRD EDITION

Binary Search Trees

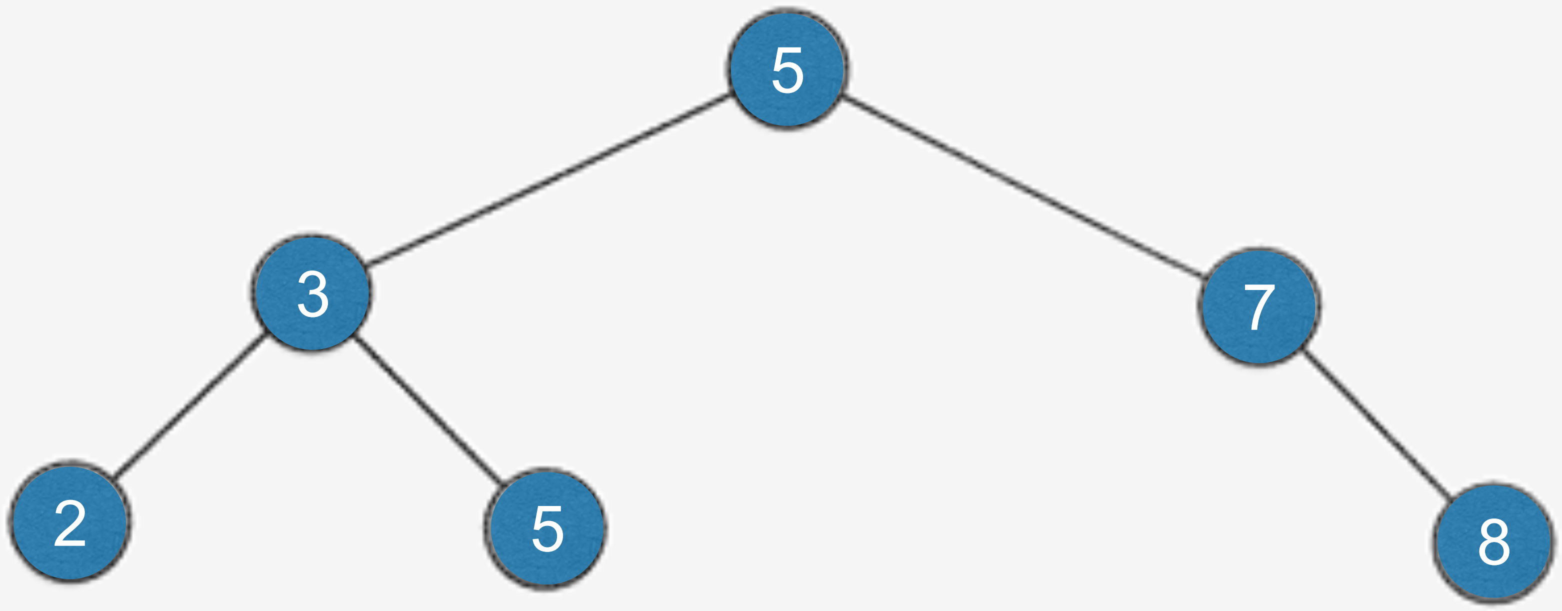
The search tree data structure supports many dynamic-set operations, including

SEARCH,
MINIMUM, **MAXIMUM**,
PREDECESSOR, **SUCCESSOR**,
INSERT, and **DELETE**.

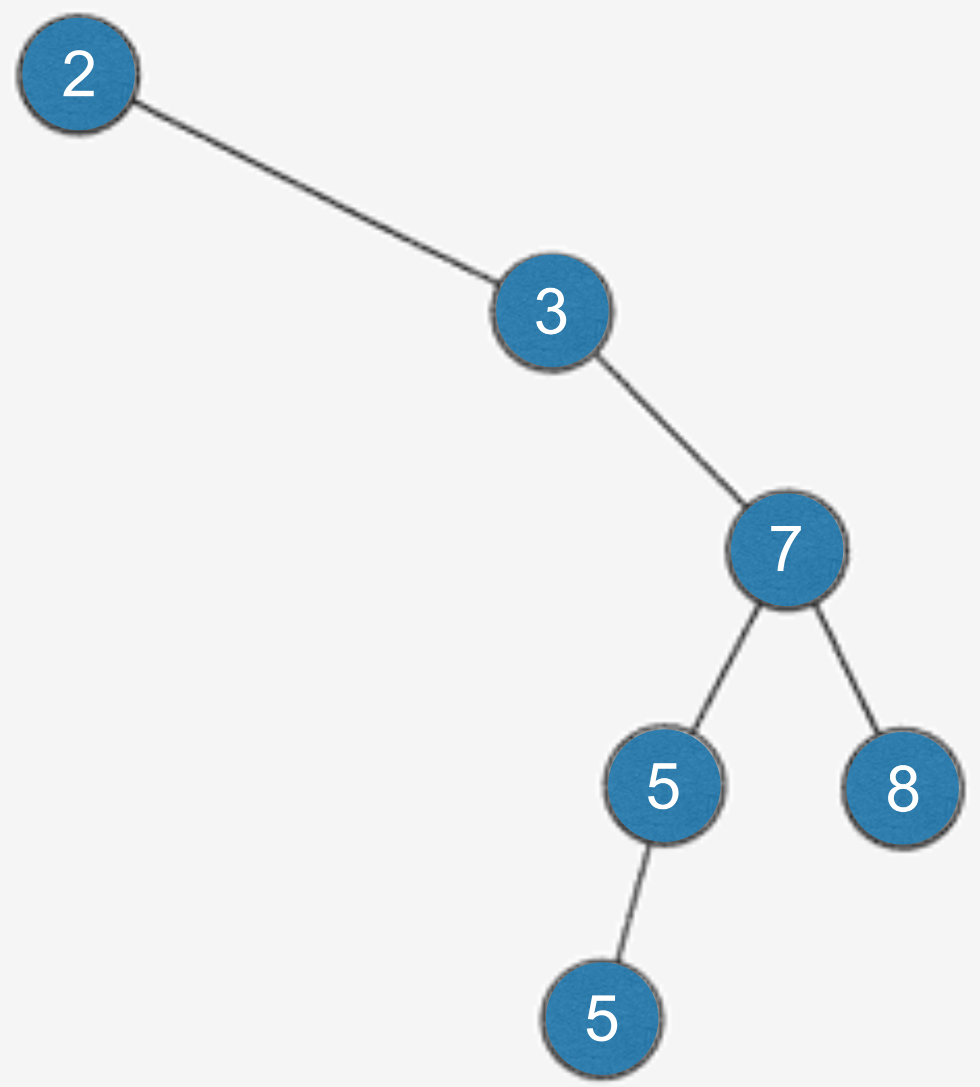
Thus, we can use a search tree both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time.

In practice, we can design variations of binary search trees with good guaranteed worst-case performance on basic operations. Chapter 13 presents one such variation, **RED-BLACK** trees, which have height $O(\lg n)$.



(a)



(b)

Binary Search Trees

We can represent a binary search tree (BST) by a linked data structure in which each node is an object.

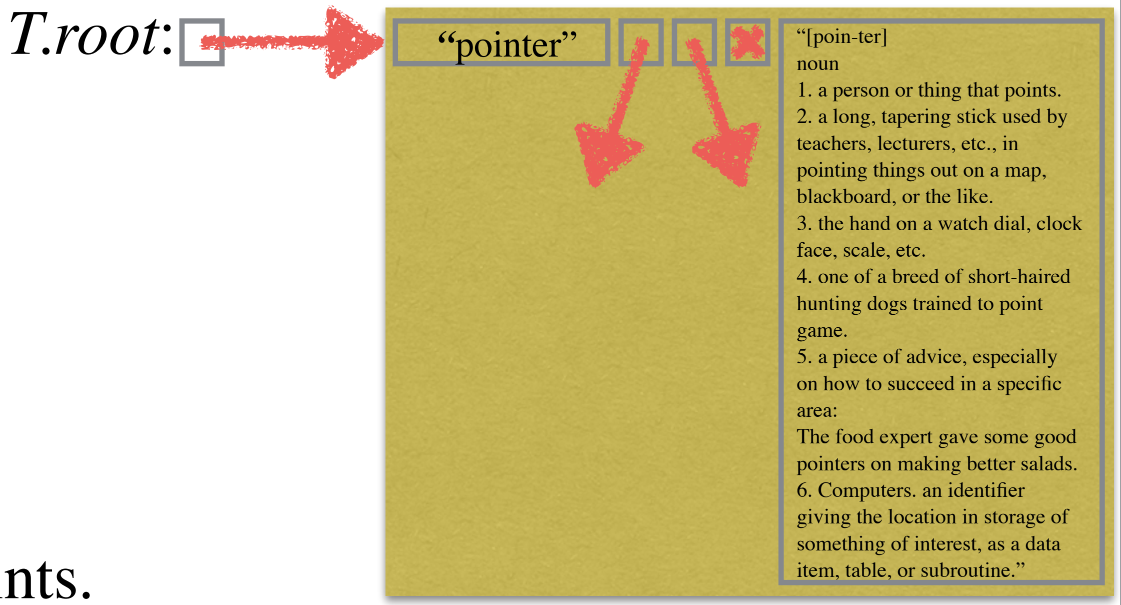
In addition to a *key* and satellite *data*, each node contains attributes *left*, *right*, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value **NIL**. The root node is the only node in the tree whose parent is **NIL**.

binary-search-tree property:
Let x be a node in a binary search tree.
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 $y.key \geq x.key$.

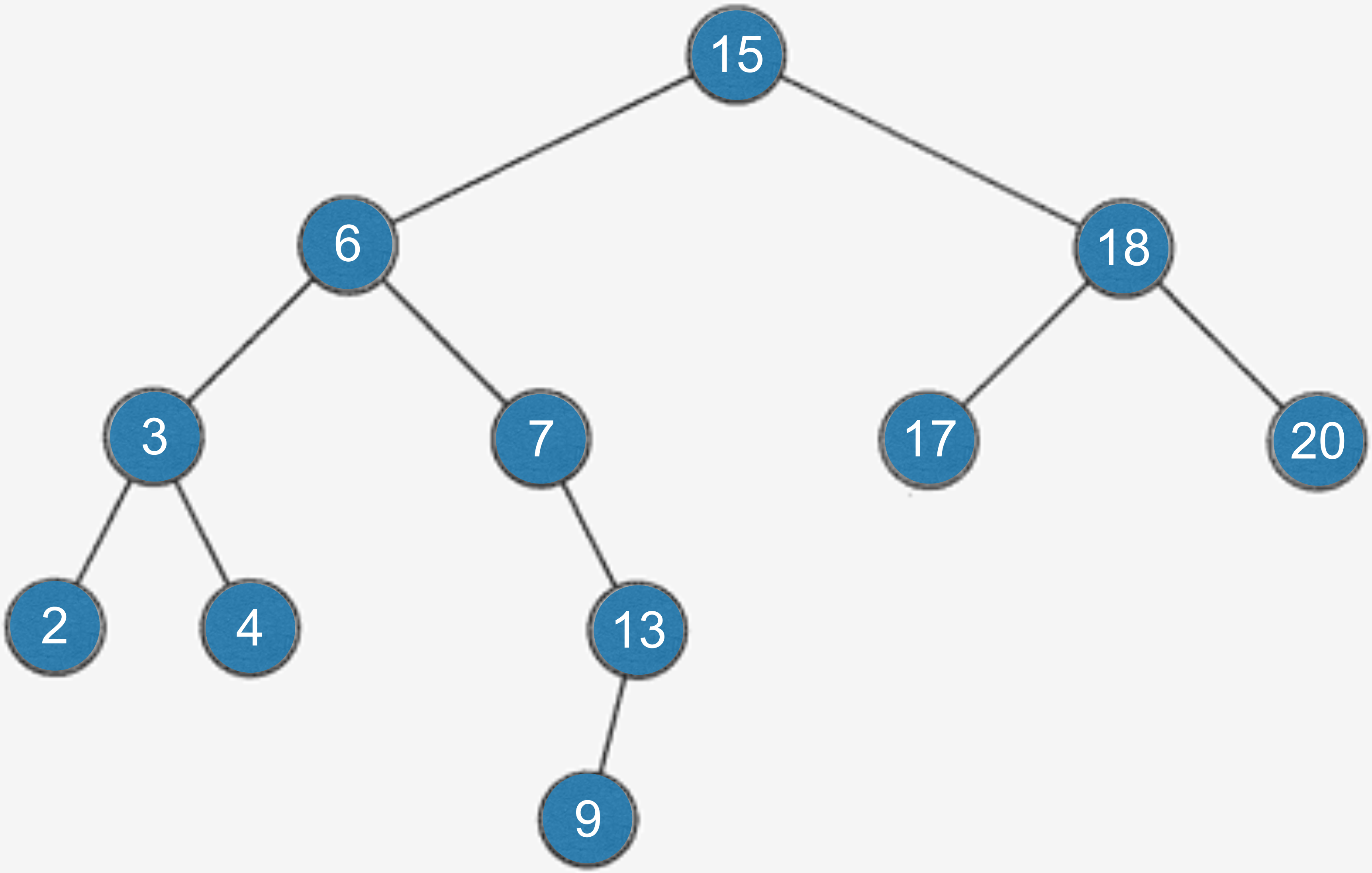
BST objects

T.root:

key: "pointer"
left: *y*
right: *z*
p: NIL
satellite data:
"[poin-ter]
noun
1. a person or thing that points.
2. a long, tapering stick used by teachers, lecturers, etc., in pointing things out on a map, blackboard, or the like.
3. the hand on a watch dial, clock face, scale, etc.
4. one of a breed of short-haired hunting dogs trained to point game.
5. a piece of advice, especially on how to succeed in a specific area:
The food expert gave some good pointers on making better salads.
6. Computers. an identifier giving the location in storage of something of interest, as a data item, table, or subroutine."


T.root:
"pointer"
"[poin-ter]
noun
1. a person or thing that points.
2. a long, tapering stick used by teachers, lecturers, etc., in pointing things out on a map, blackboard, or the like.
3. the hand on a watch dial, clock face, scale, etc.
4. one of a breed of short-haired hunting dogs trained to point game.
5. a piece of advice, especially on how to succeed in a specific area:
The food expert gave some good pointers on making better salads.
6. Computers. an identifier giving the location in storage of something of interest, as a data item, table, or subroutine."

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```


15
6
18
3
7
17
20
2
4
13
9

```
TREE-SEARCH($x$, $k$)
1  if $x$ == NIL or $k$ == $x.key$
2      return $x$
3  if $k < x.key$
4      return TREE-SEARCH($x.left$, $k$)
5  else return TREE-SEARCH($x.right$, $k$)
```

```
ITERATIVE-TREE-SEARCH($x$, $k$)
1  while $x \neq$ NIL and $k \neq x.key$
2      if $k < x.key$
3          $x = x.left$
4      else $x = x.right$
5  return $x$
```

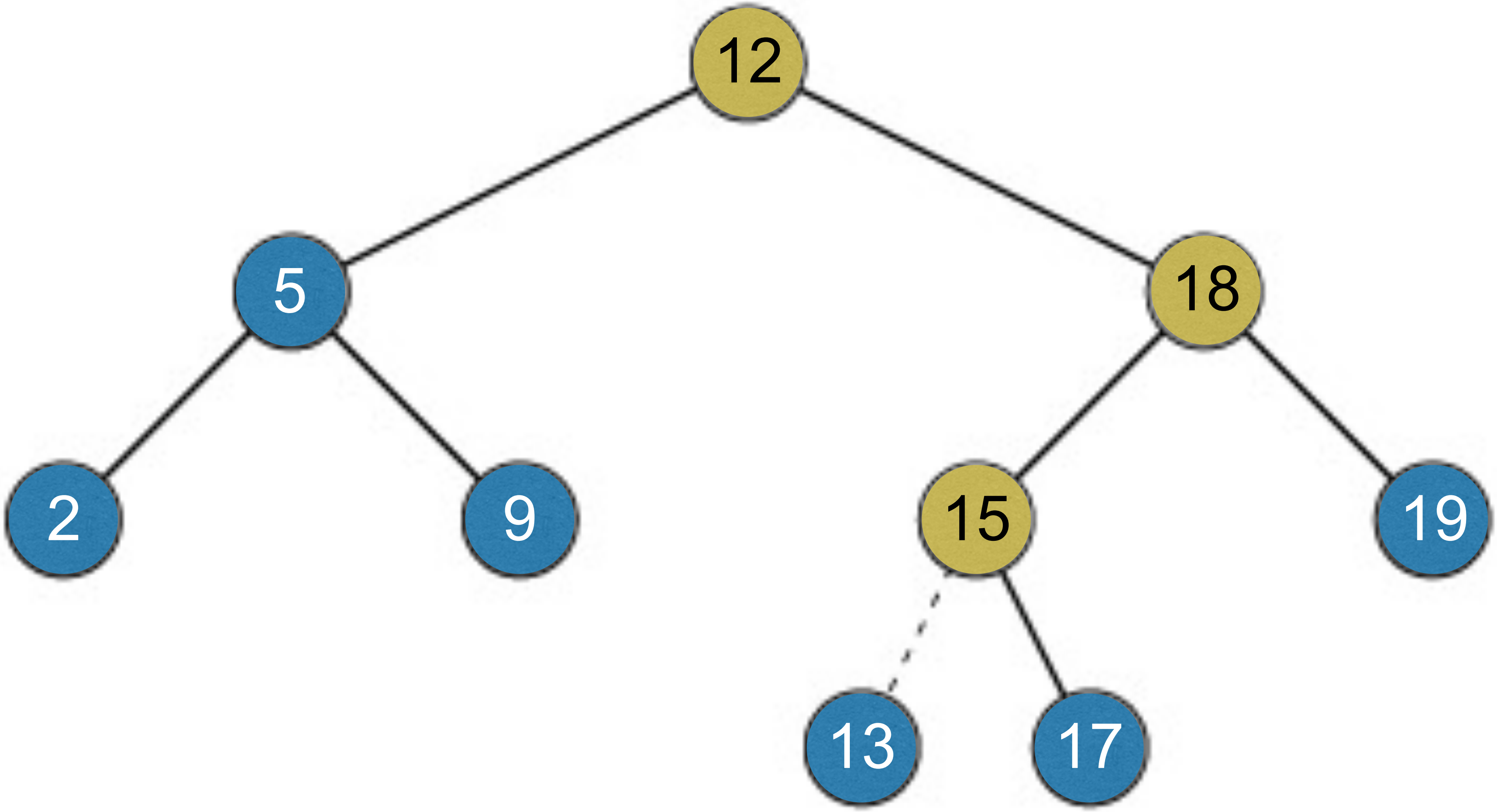
```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

```
TREE-SUCCESSOR($x$)
1  if $x.right \neq$ NIL
2      return TREE-MINIMUM($x.right$)
3  $y = x.p$
4  while $y \neq$ NIL and $x == y.right$
5      $x = y$
6      $y = y.p$
7  return $y$
```

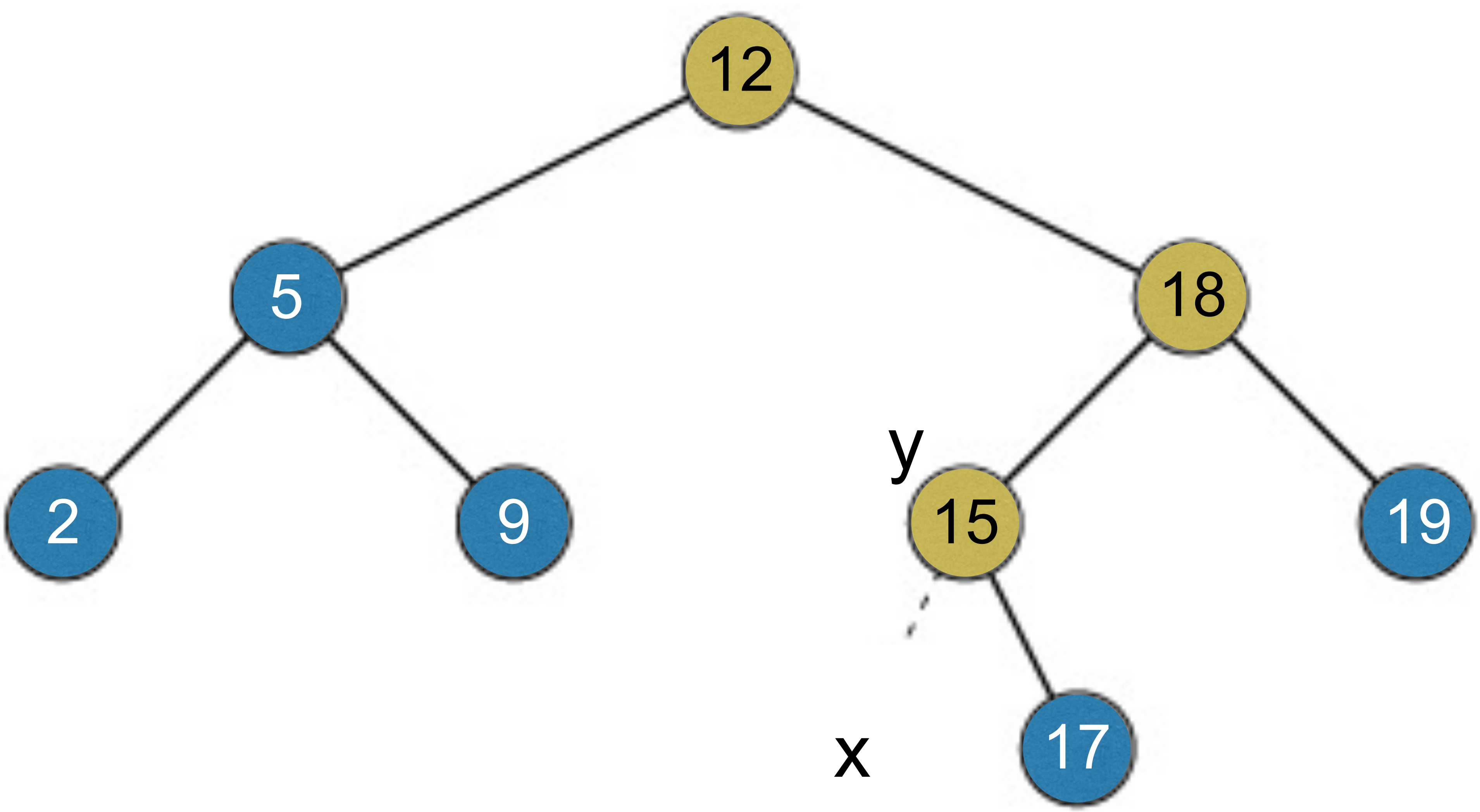
```
TREE-PREDECESSOR($x$)
1  if $x.left \neq$ NIL
2      return TREE-MAXIMUM($x.left$)
3  $y = x.p$
4  while $y \neq$ NIL and $x == y.left$
5      $x = y$
6      $y = y.p$
7  return $y$
```

BST insertion



```
TREE-INSERT(T, z)
1  y = NIL
2  x = T.root
3  while x ≠ NIL                  // finds y the parent
4      y = x                      // of the new node z
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == NIL
10     T.root = z                 // tree T was empty
11 elseif z.key < y.key
12         y.left = z             // assigns z as the
13     else y.right = z           // correct child of y.
```

BST insertion



BST deletion

Figure 12.4: Deleting a node *z* from a binary search tree. Node *z* may be the root, a left child of node *q*, or a right child of *q*.

(a) Node *z* has no left child. We replace *z* by its right child *r*, which may or may not be **NIL**.

(b) Node *z* has a left child *l* but no right child. We replace *z* by *l*.

(c) Node *z* has two children; its left child is node *l*, its right child is its successor *y*, and *y*'s right child is node *x*. We replace *z* by *y*, updating *y*'s left child to become *l*, but leaving *x* as *y*'s right child.

(d) Node *z* has two children (left child *l* and right child *r*), and its successor $y \neq r$ lies within the subtree rooted at *r*. We replace *y* by its own right child *x*, and we set *y* to be *r*'s parent. Then, we set *y* to be *q*'s child and the parent of *l*.

BST deletion

(a)

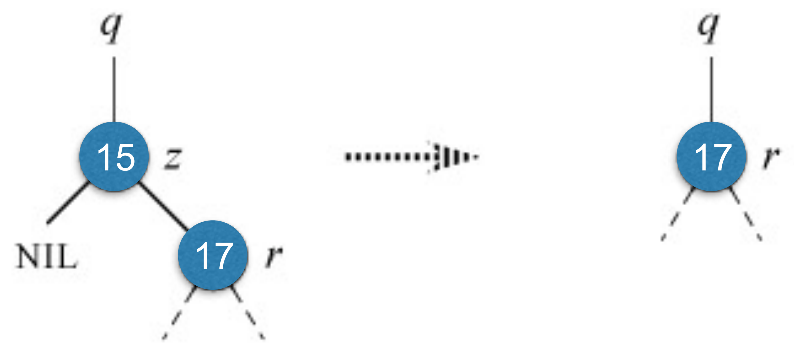


Figure 12.4: Deleting a node z from a binary search tree. Node z may be the root, a left child of node q, or a right child of q.

(a) Node z has no left child. We replace z by its right child r, which may or may not be **NIL**.

BST deletion

(b)

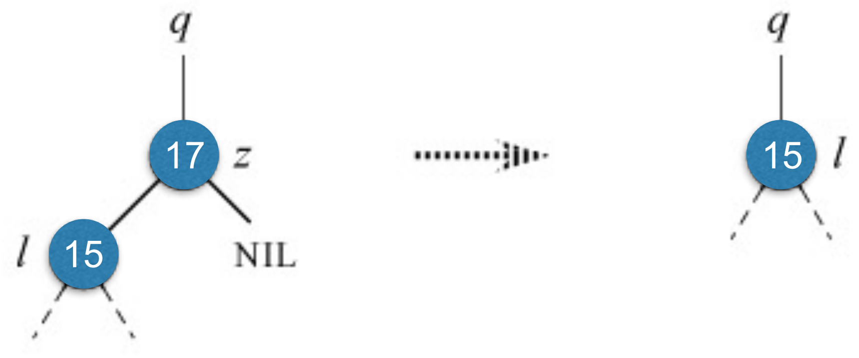


Figure 12.4: Deleting a node *z* from a binary search tree. Node *z* may be the root, a left child of node *q*, or a right child of *q*.

(b) Node *z* has a left child *l* but no right child. We replace *z* by *l*.

BST deletion

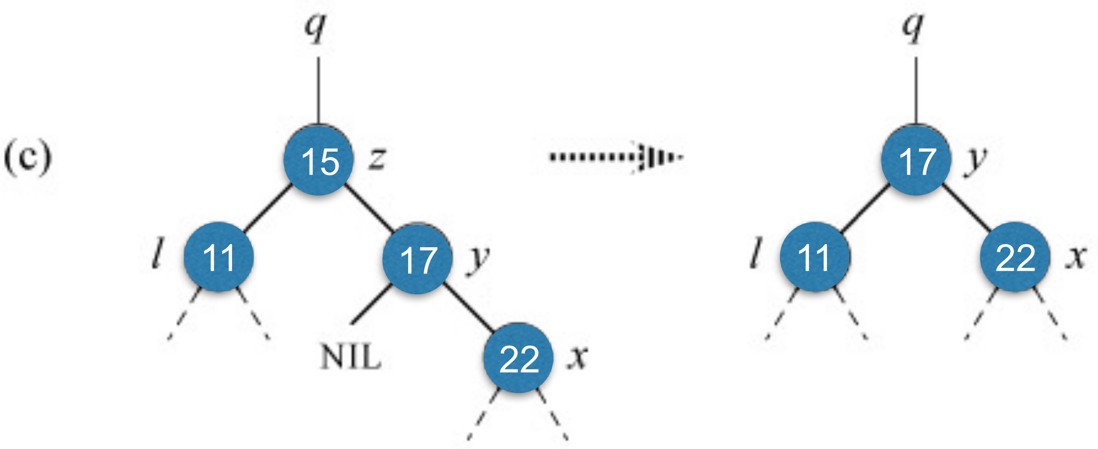


Figure 12.4: Deleting a node z from a binary search tree. Node z may be the root, a left child of node q, or a right child of q.

(c) Node z has two children; its left child is node l, its right child is its successor y, and y's right child is node x. We replace z by y, updating y's left child to become l, but leaving x as y's right child.

BST deletion

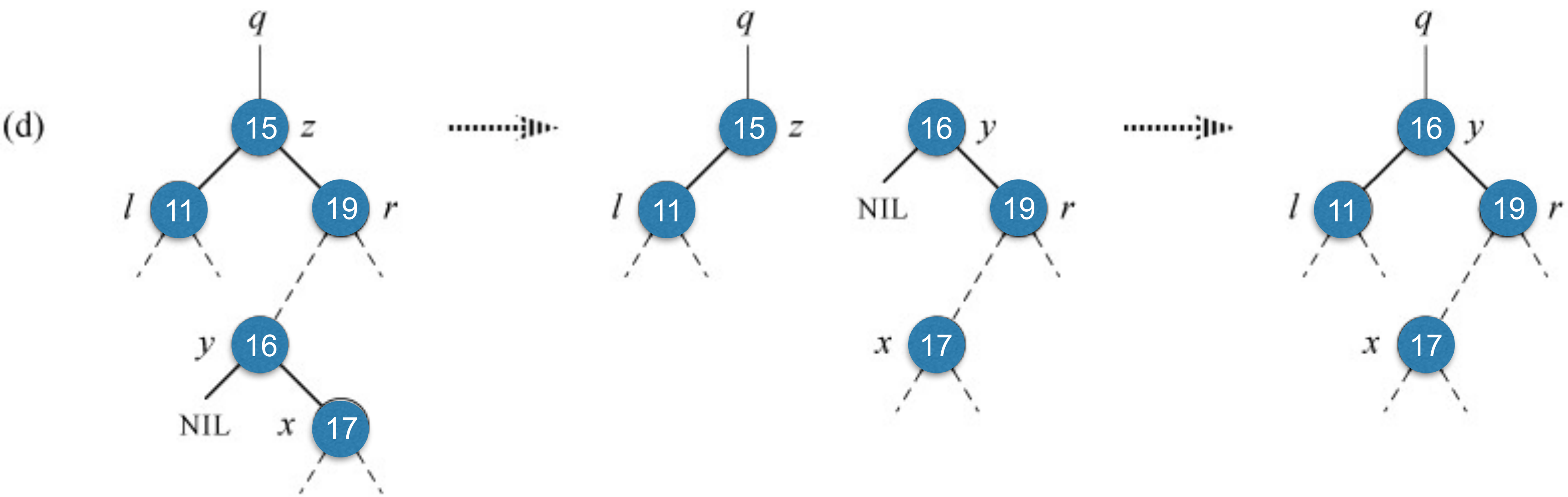


Figure 12.4: Deleting a node z from a binary search tree. Node z may be the root, a left child of node q, or a right child of q.

(d) Node z has two children (left child l and right child r), and its successor $y \neq r$ lies within the subtree rooted at r. We replace y by its own right child x, and we set r to be y's right child. Then, we set y to be q's child and we set l to be y's left child.

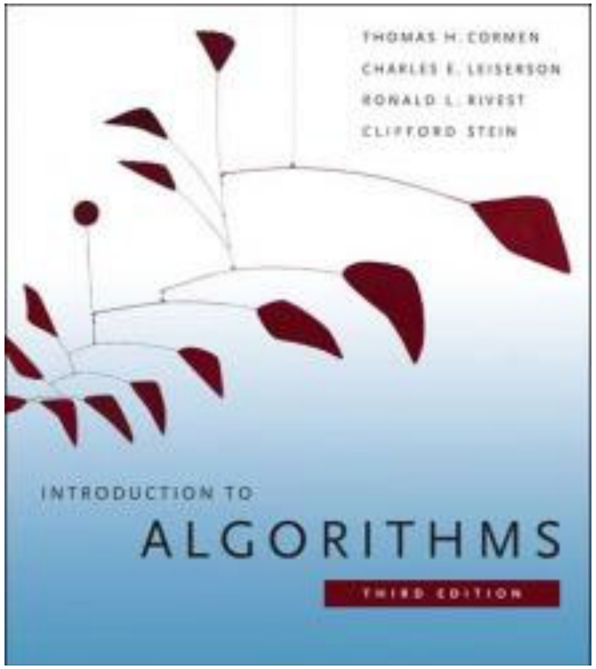
BST deletion

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4          u.p.left = v
5      else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

Replaces *u* by *v*.

```
TREE-DELETE(T, z)
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)
3  elseif z.right == NIL
4          TRANSPLANT(T, z, z.left)
5      else y = TREE-MINIMUM(z.right)
6          if y.p ≠ z
7              TRANSPLANT(T, y, y.right)
8              y.right = z.right
9              y.right.p = y
10         TRANSPLANT(T, z, y)
11         y.left = z.left
12         y.left.p = y
```

CLRS 13 Red-Black Trees


THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
INTRODUCTION TO
ALGORITHMS
THIRD EDITION

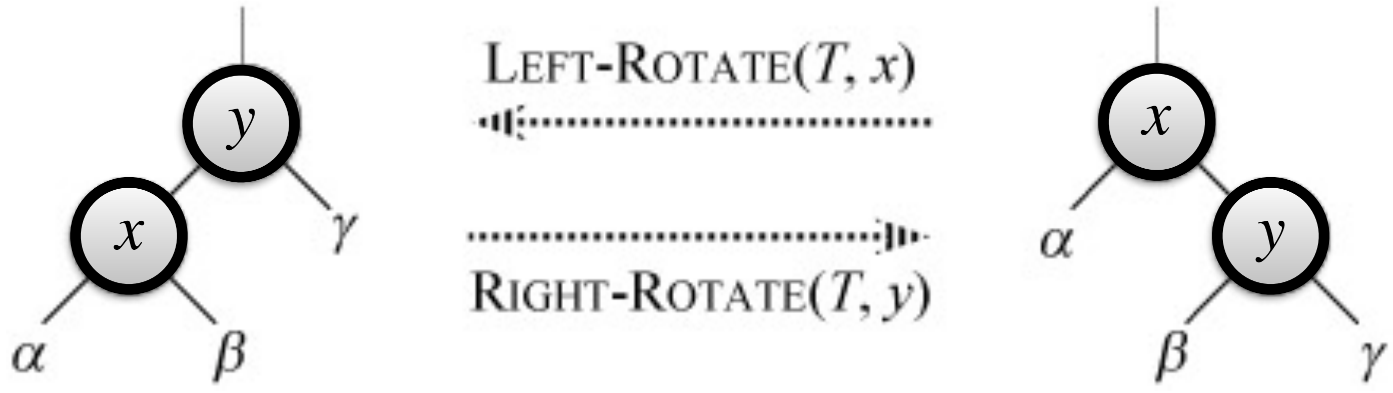
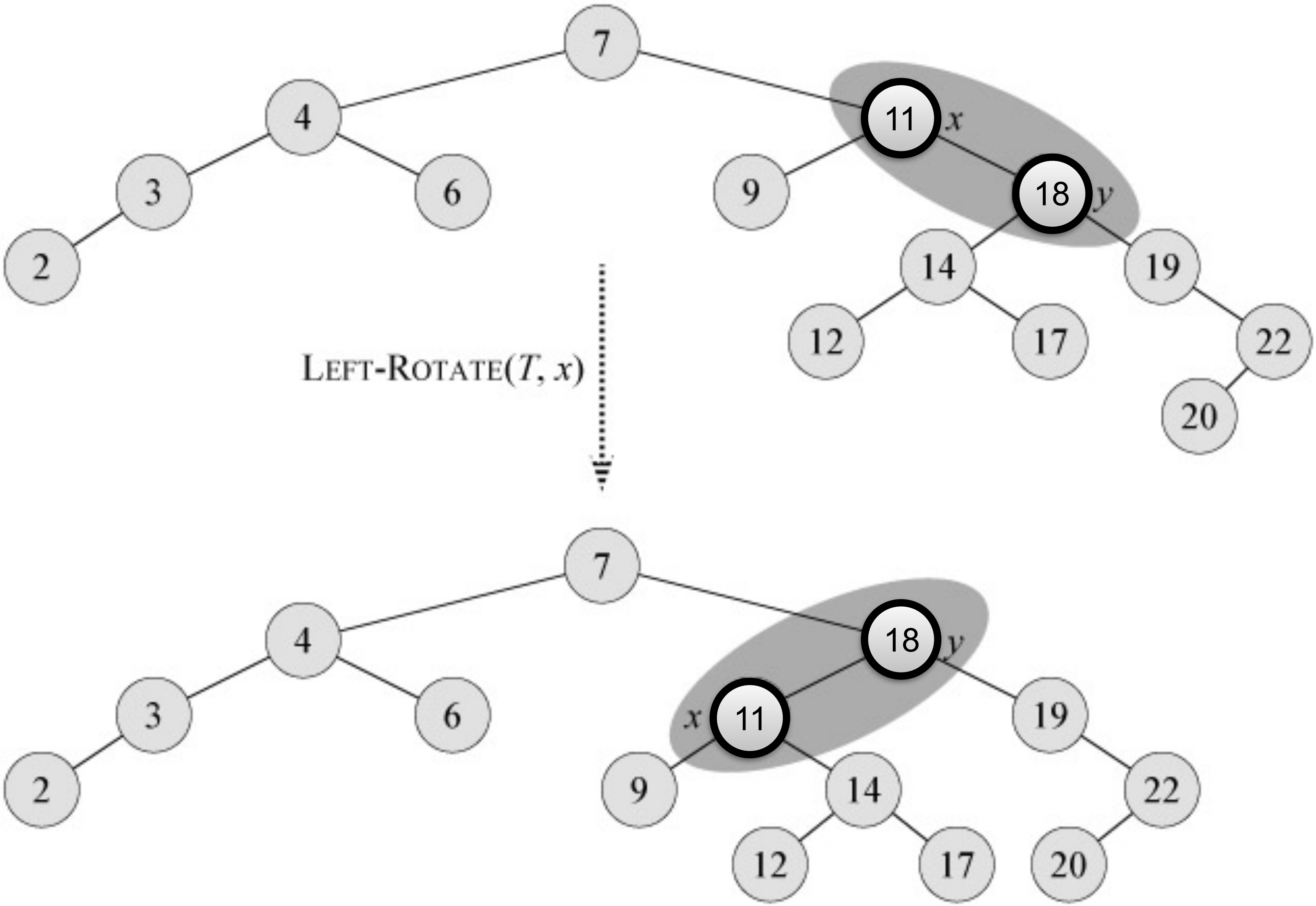


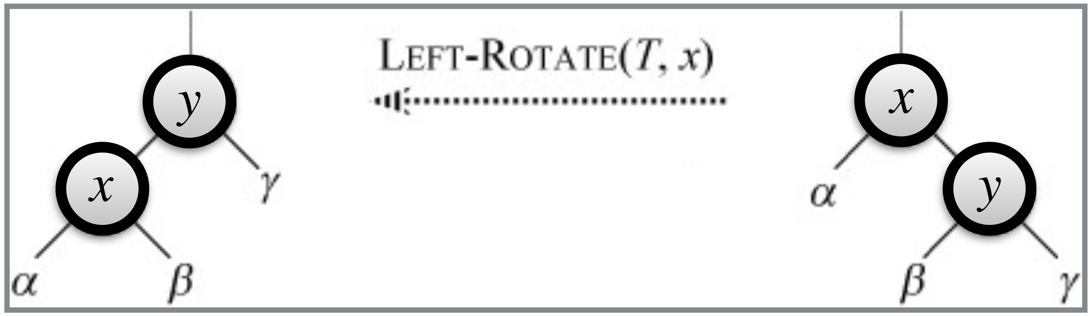
Figure 13.2: The rotation operations on a binary search tree. The operation **LEFT-ROTATE(*T*,*x*)** transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation **RIGHT-ROTATE(*T*,*y*)** transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property—the keys in α precede *x.key*, which precedes the keys in β, which precede *y.key*, which precedes the keys in γ.


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

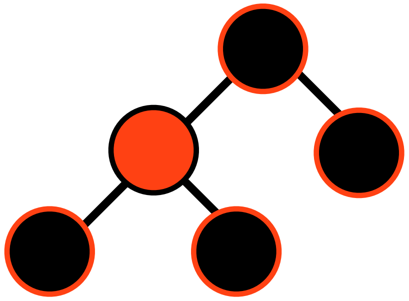
The pseudocode for **LEFT-ROTATE** assumes that $x.right \neq T.nil$.

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 $y.p == T.nil$
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 $x.p = y$
```



RED-BLACK Tree properties



1. Every node is either red or black.
2. The root is black.
3. Every leaf (*T.nil*) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

RED-BLACK Trees

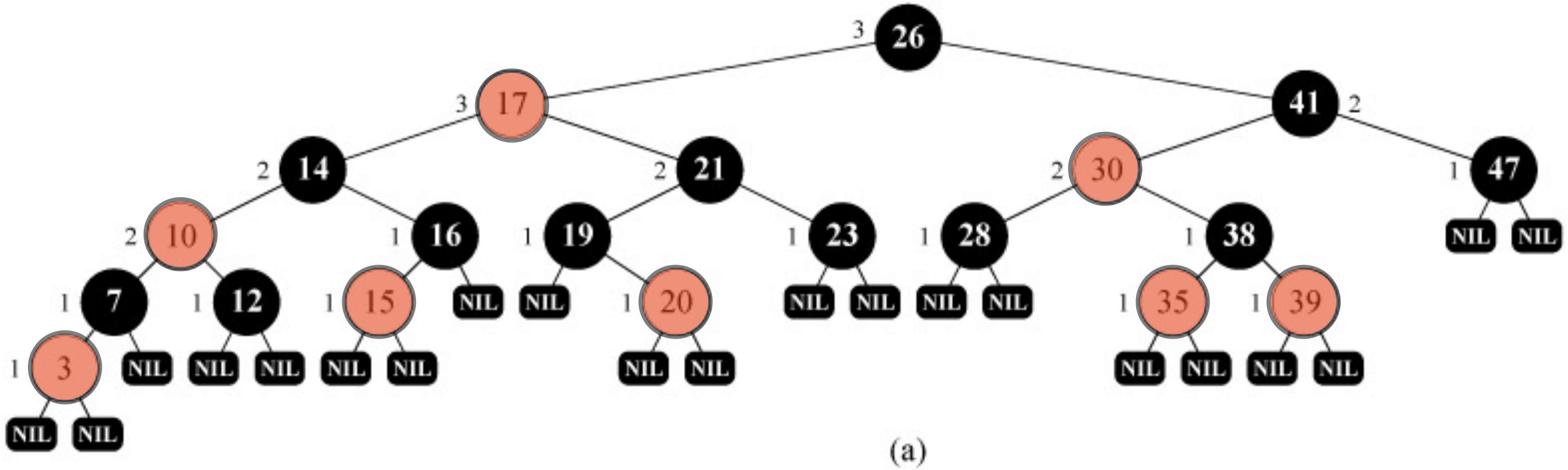


Figure 13.1: A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, 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.

RED-BLACK Trees

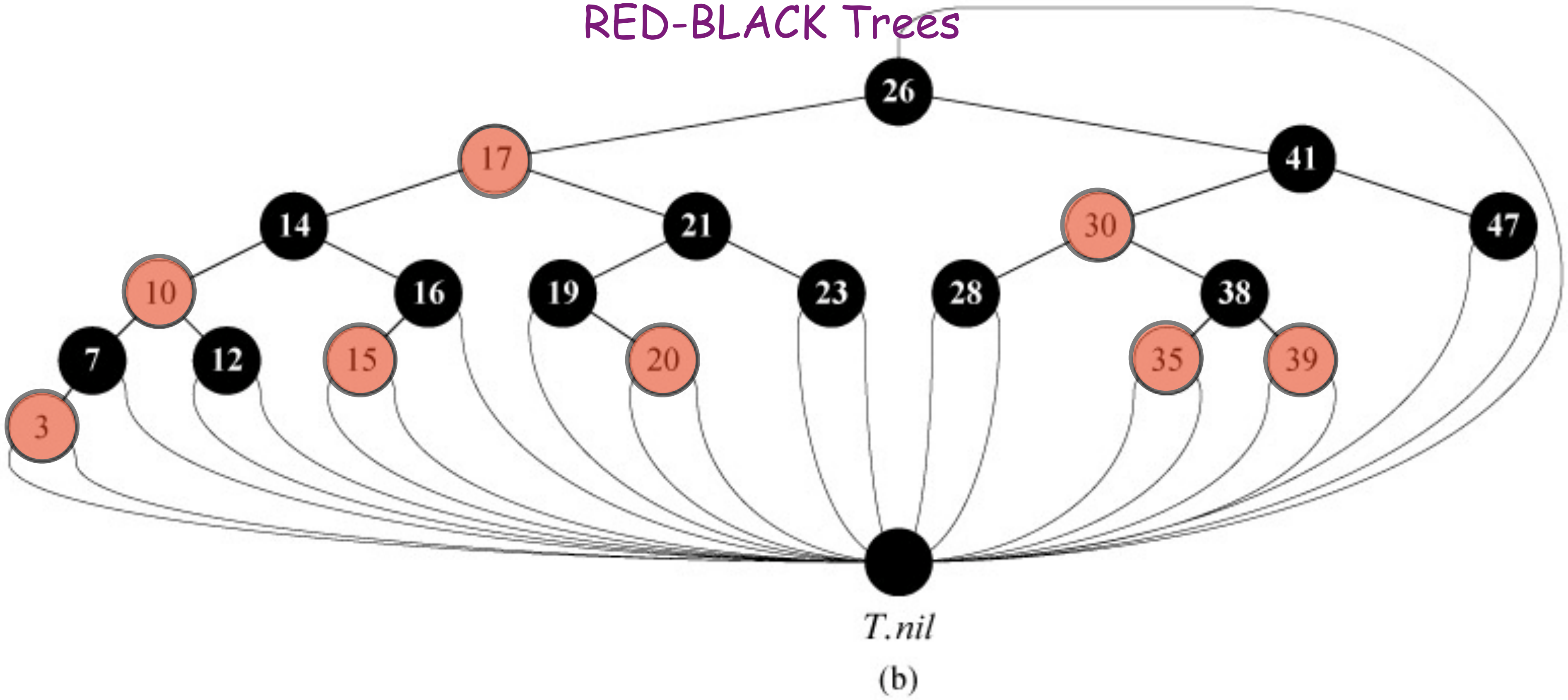


Figure 13.1: A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. (a) Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, 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.

RED-BLACK Trees

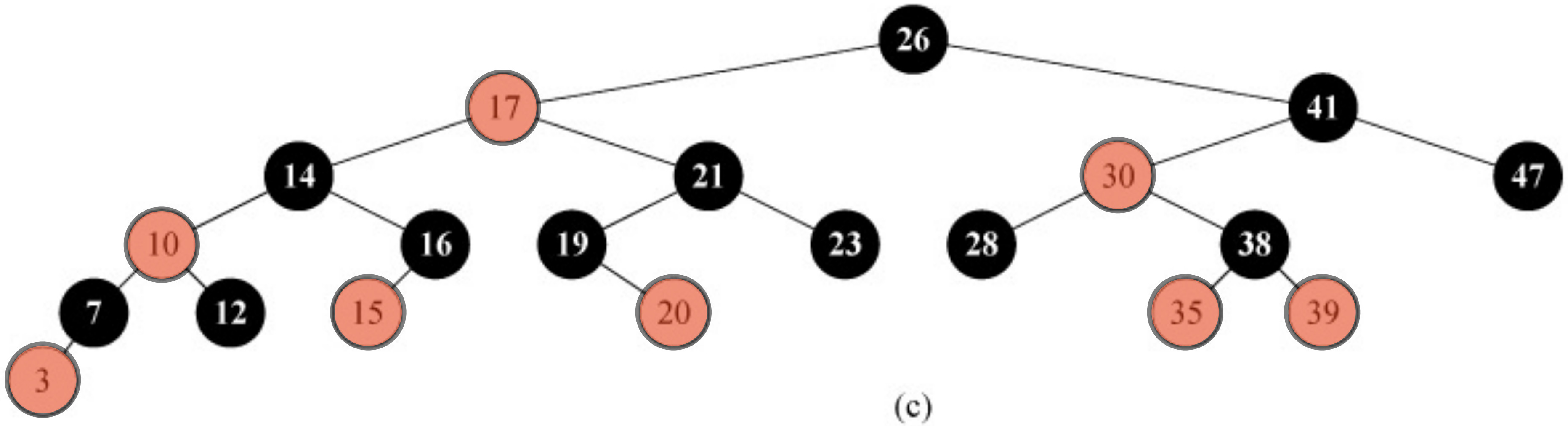


Figure 13.1: A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, 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.

RED-BLACK Trees

Height of a red-black tree

- ***Height of a node*** is the number of edges in a longest path to a leaf.
- ***Black-height*** of a node x: bh(x) is the number of black nodes (including *nil*$[T]$) on the path from x to leaf, not counting x. By property 5, black-height is well defined.

Claim

Any node with height h has black-height $\geq h/2$.

Proof By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ are black. ■ (claim)

RED-BLACK Trees

Claim

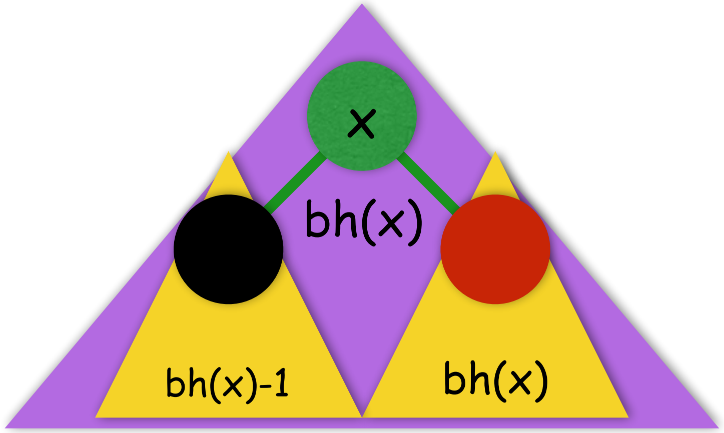
The subtree rooted at any node x contains at least $2^{\mathbf{bh}(x)} - 1$ internal nodes.

Proof: We prove this claim by induction on the height of x.

If the height of x is 0, then x must be a leaf ($T.nil$), and the subtree rooted at x indeed contains at least $2^{\mathbf{bh}(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

For the inductive step, consider a node x that has positive height $\boldsymbol{h}(x)>0$ and is an internal node with two children. Each child has a black-height of either $\mathbf{bh}(x)$ or $\mathbf{bh}(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{\mathbf{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{\mathbf{bh}(x)-1} - 1) + (2^{\mathbf{bh}(x)-1} - 1) + 1 = 2^{\mathbf{bh}(x)} - 1$ internal nodes, which proves the claim.

RED-BLACK Trees



For the inductive step, consider a node x that has positive height $\boldsymbol{h}(x)>0$ and is an internal node with two children. Each child has a black-height of either $\mathbf{bh}(x)$ or $\mathbf{bh}(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{\mathbf{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{\mathbf{bh}(x)-1} - 1) + (2^{\mathbf{bh}(x)-1} - 1) + 1 = 2^{\mathbf{bh}(x)} - 1$ internal nodes, which proves the claim.

RED-BLACK Trees

Lemma

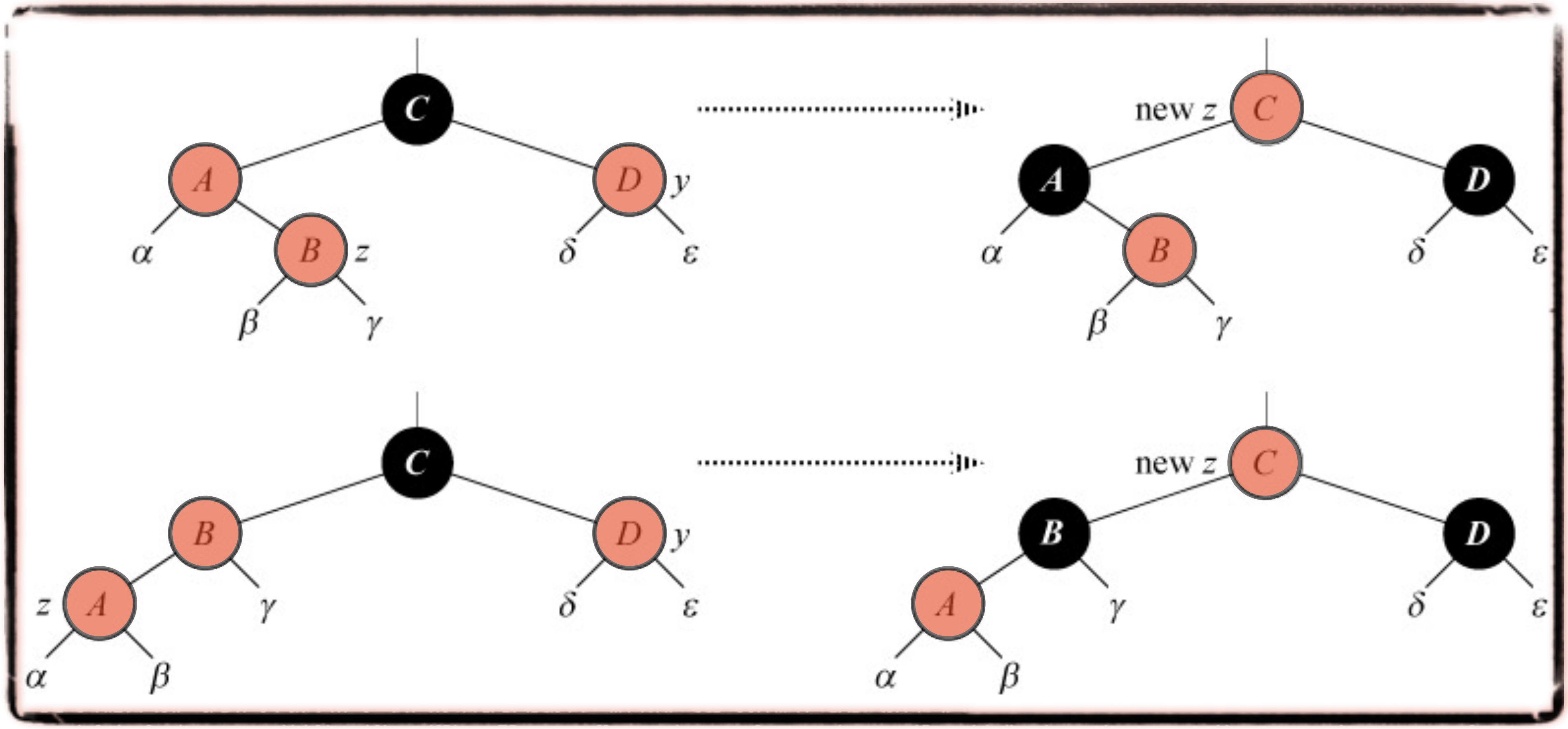
A red-black tree with n internal nodes has height at most $2 \lg(n + 1)$.

Proof

Let h be the height of the tree. According to **property 4**, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least $h/2$. Given that the subtree rooted at the root r contains at least $2^{\mathbf{bh}(r)} - 1$ internal nodes, we have $n \geq 2^{h/2} - 1$. Moving the 1 to the left-hand side and taking logarithms on both sides yields

$$\lg(n + 1) \geq h/2, \text{ or } h \leq 2 \lg(n + 1).$$

RED-BLACK Tree INSERTION



RED-BLACK Trees insertion

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)
```

RED-BLACK Trees FIXUP

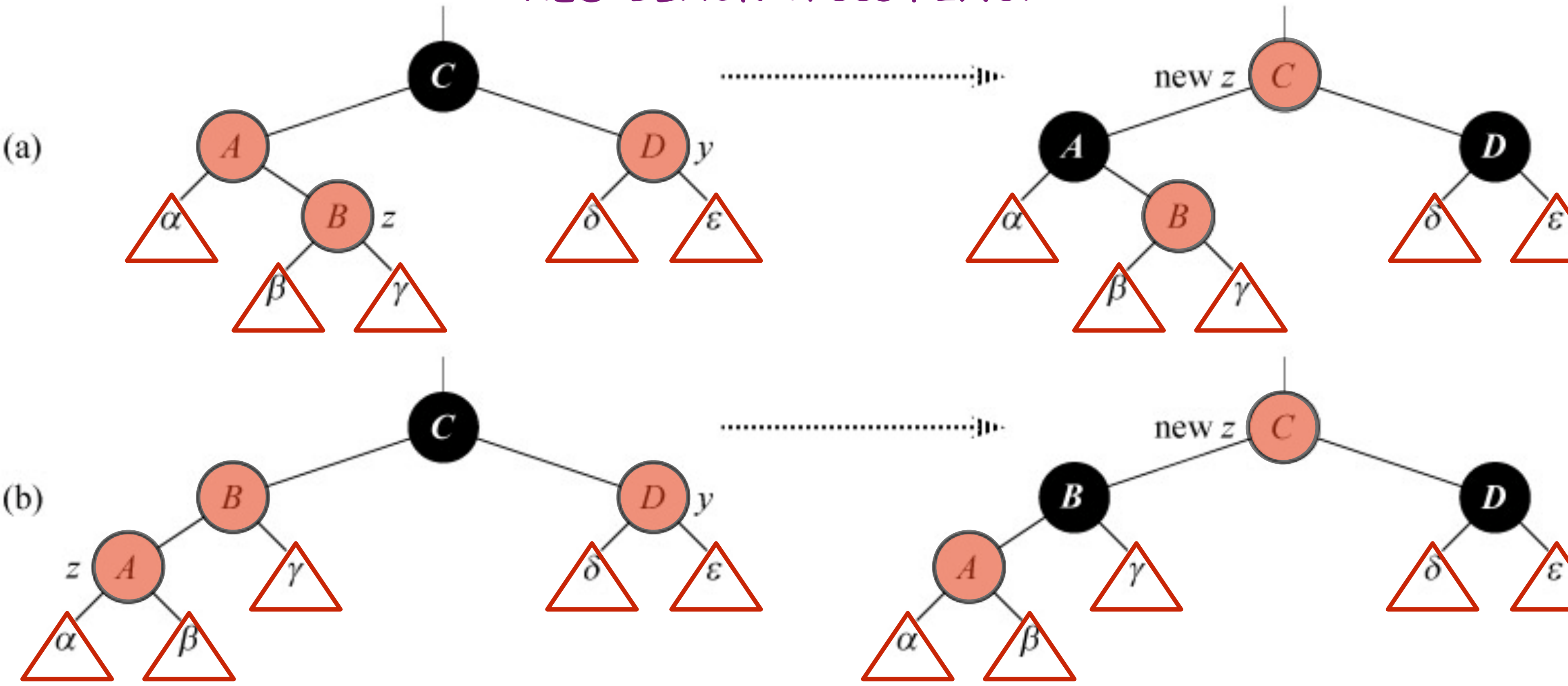


Figure 13.5: Case 1 of the procedure **RB-INSERT-FIXUP**. **Property 4** is violated, since *z* and its parent *z.p* are both red. We take the same action whether **(a)** *z* is a right child or **(b)** *z* is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for **case 1** changes the colors of some nodes, preserving **property 5**—all downward simple paths from a node to a leaf have the same number of blacks. The while loop continues with node *z*'s grandparent *z.p.p* 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.

RED-BLACK Trees FIXUP

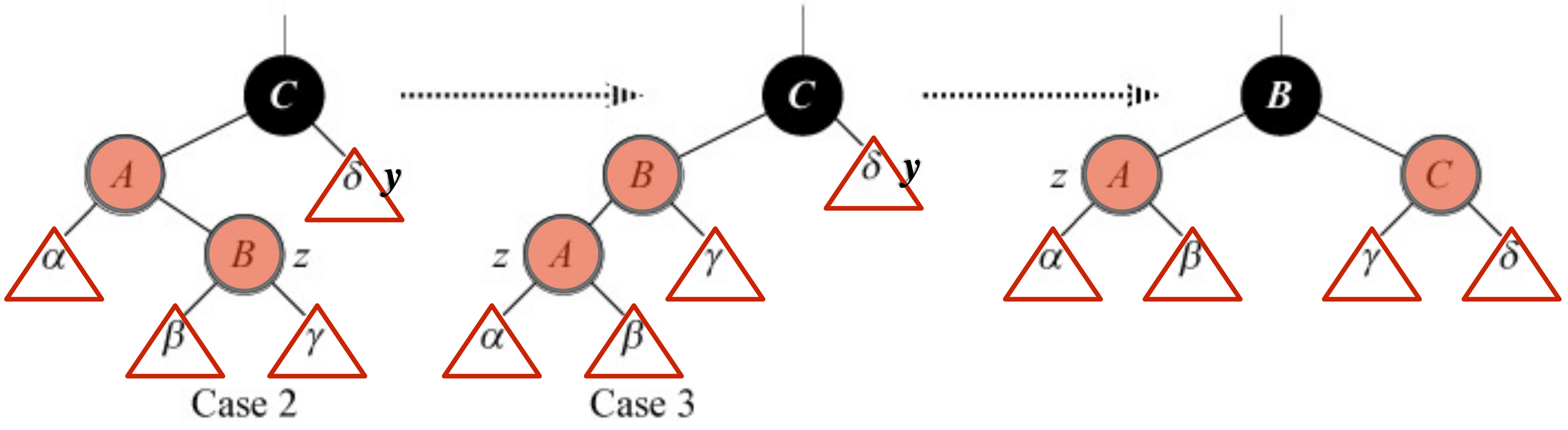
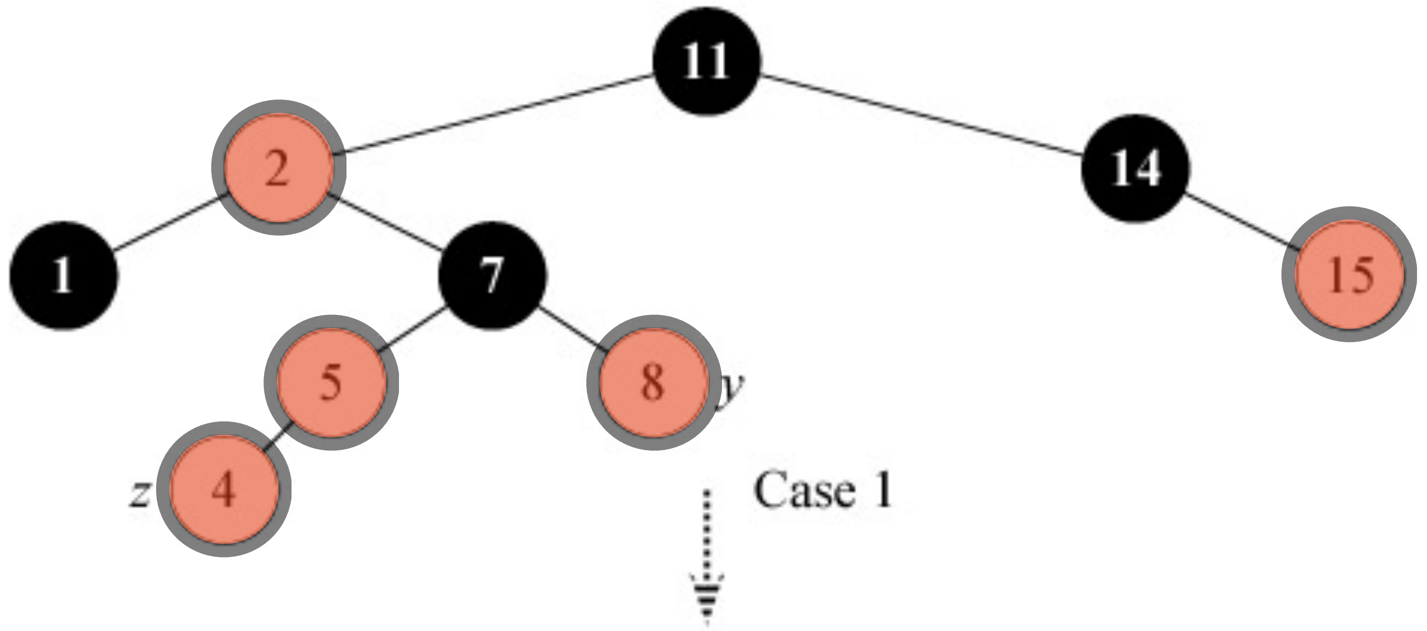


Figure 13.6: Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in **case 1**, **property 4** is violated in either **case 2** or **case 3** because *z* and its parent *z.p* are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from **property 4**, and δ because otherwise we would be in **case 1**), and each has the same black-height. We transform **case 2** into **case 3** by a left rotation, which preserves **property 5**—all downward simple 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.

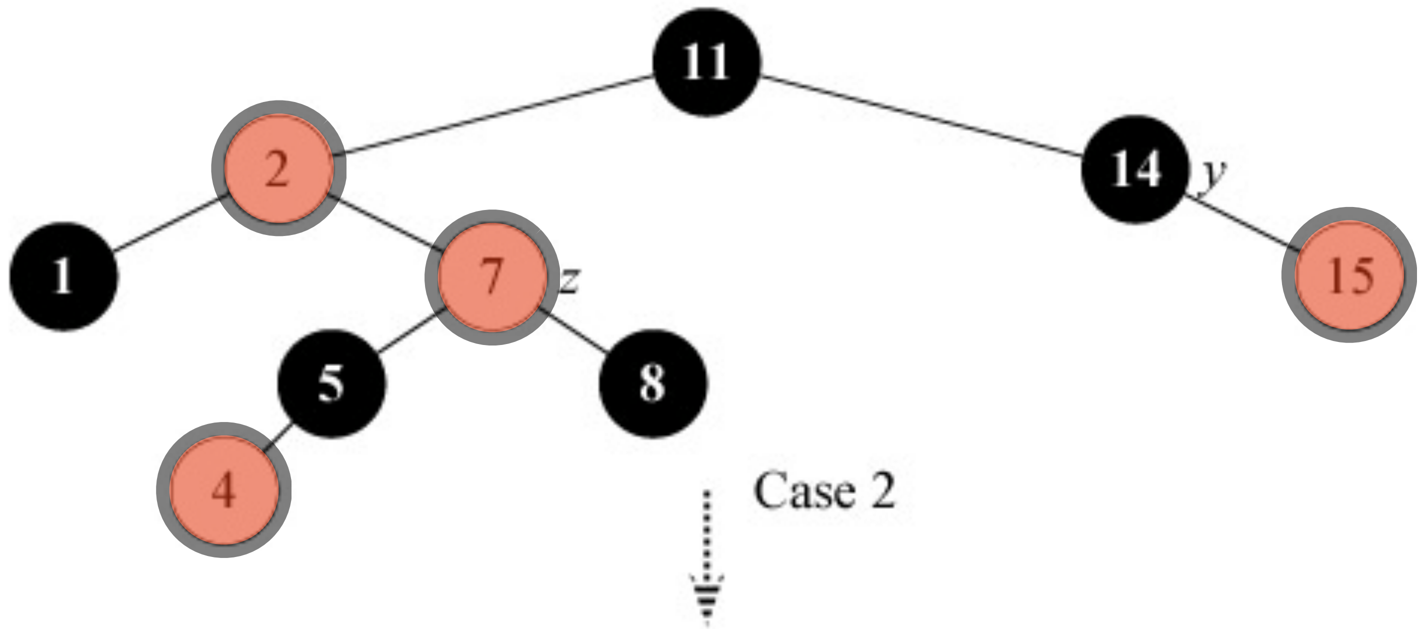
RB-INSERT-FIXUP (*T*, *z*)

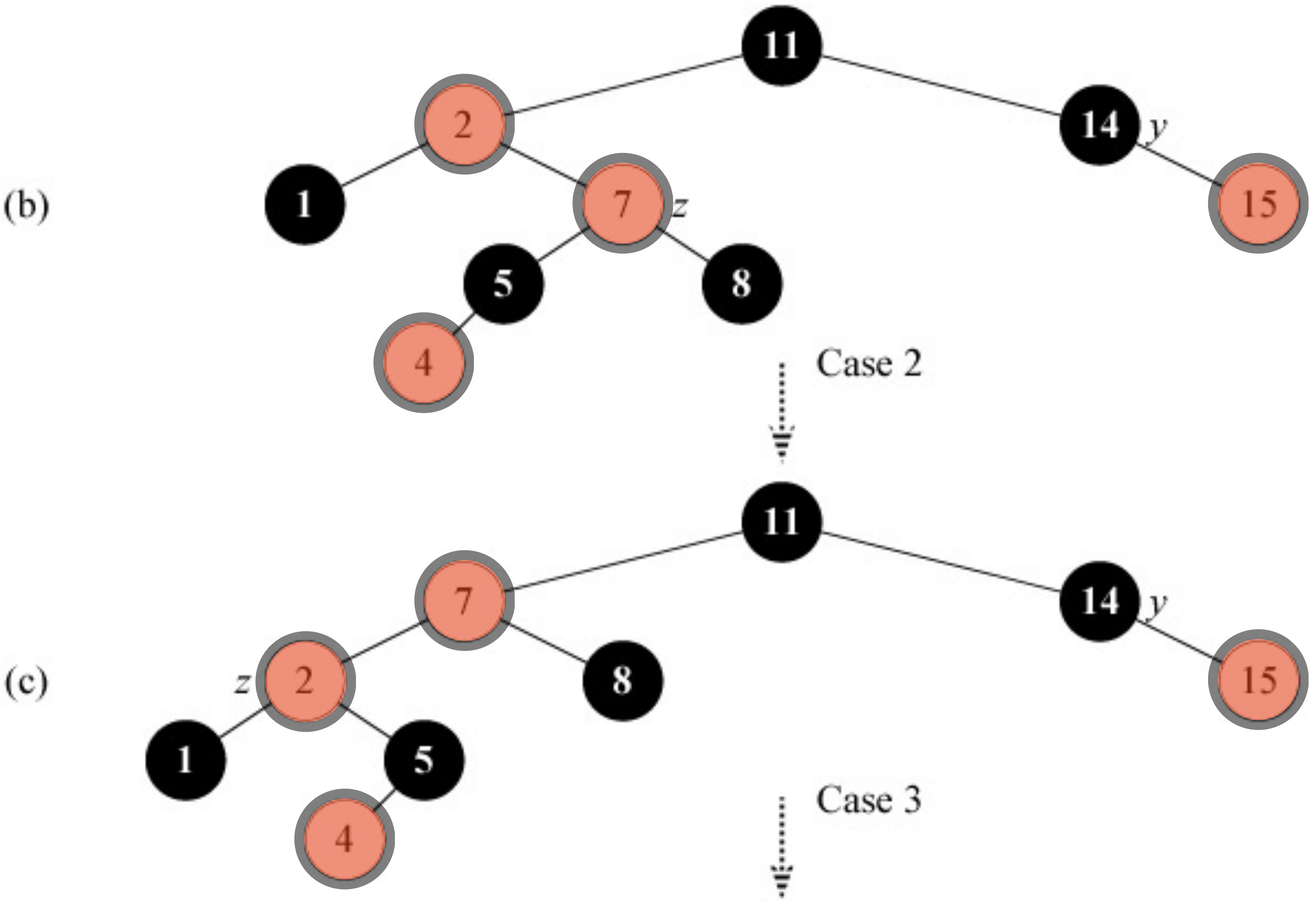
```
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.color = BLACK                    // case 1
6              y.color = BLACK                      // case 1
7              z.p.p.color = RED                    // case 1
8              z = z.p.p                            // case 1
9          elseif z == z.p.right
10                 z = z.p                          // case 2
11                 LEFT-ROTATE (T, z)               // case 2
12             z.p.color = BLACK                    // case 3
13             z.p.p.color = RED                    // case 3
14             RIGHT-ROTATE(T, z.p.p)               // case 3
15     else (same as 3—14 with "right" and "left" exchanged)
16 T.root.color = BLACK
```

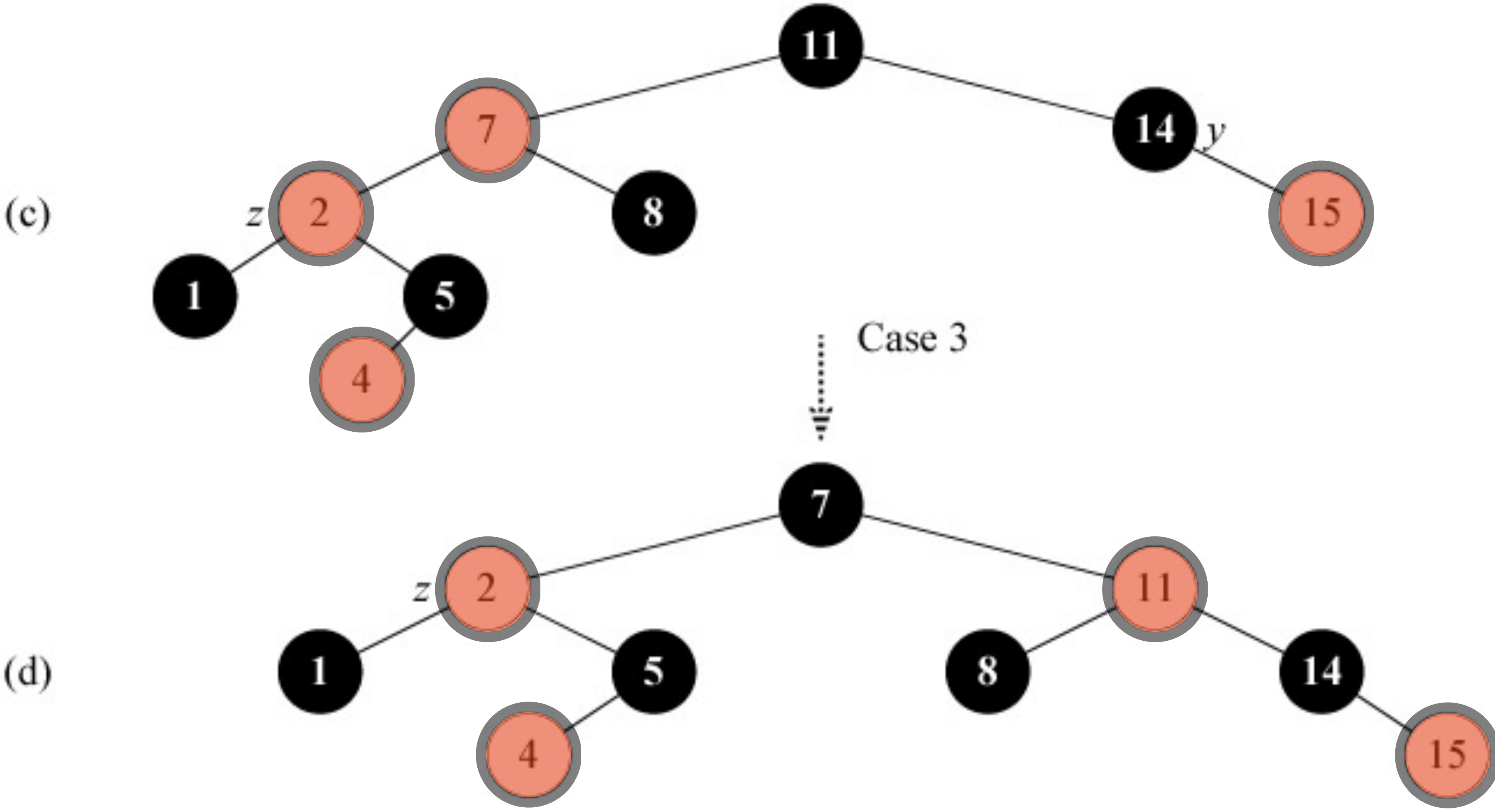
(a)



(b)




(b)
11
2
14
y
1
7
z
15
5
8
4
Case 2
(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3


(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3
(d)
7
z
2
11
1
5
8
14
4
15

Application of Red-Black Trees

First Repetition. Given n numbers $a_1, \ldots, a_n$, find the smallest i such that there exists $1 \le j < i$ such that $a_j = a_i$.

22	31	44	7	12	19	22	35	3	40	27

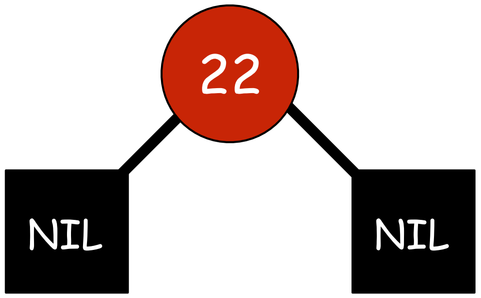
Solution:
Build a Red-Black Tree with the values encountered so far. Whenever you consider a new value in the list, binary search for it in the RB tree. If found, you have solved your problem, and if absent from the RB tree, insert the new value in the Red-Black tree where it belongs.

This is $O(i \log i)$ where i is the location of the first repetition. Most natural solutions will be $\Theta(i^2)$.

First Repetition from RED-BLACK Trees

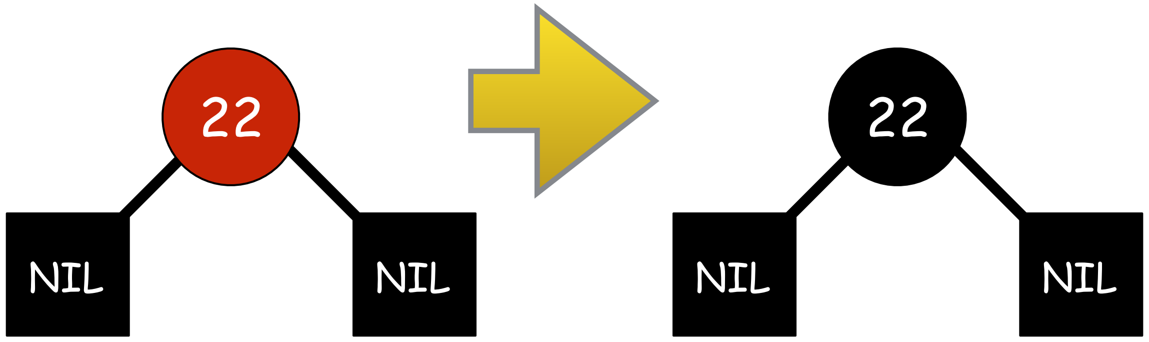
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



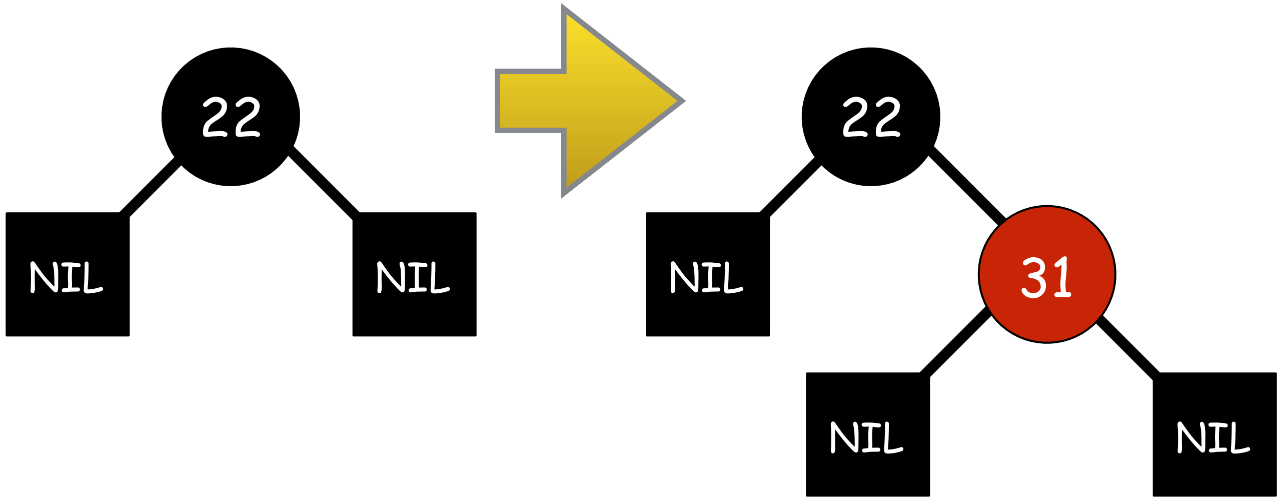
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



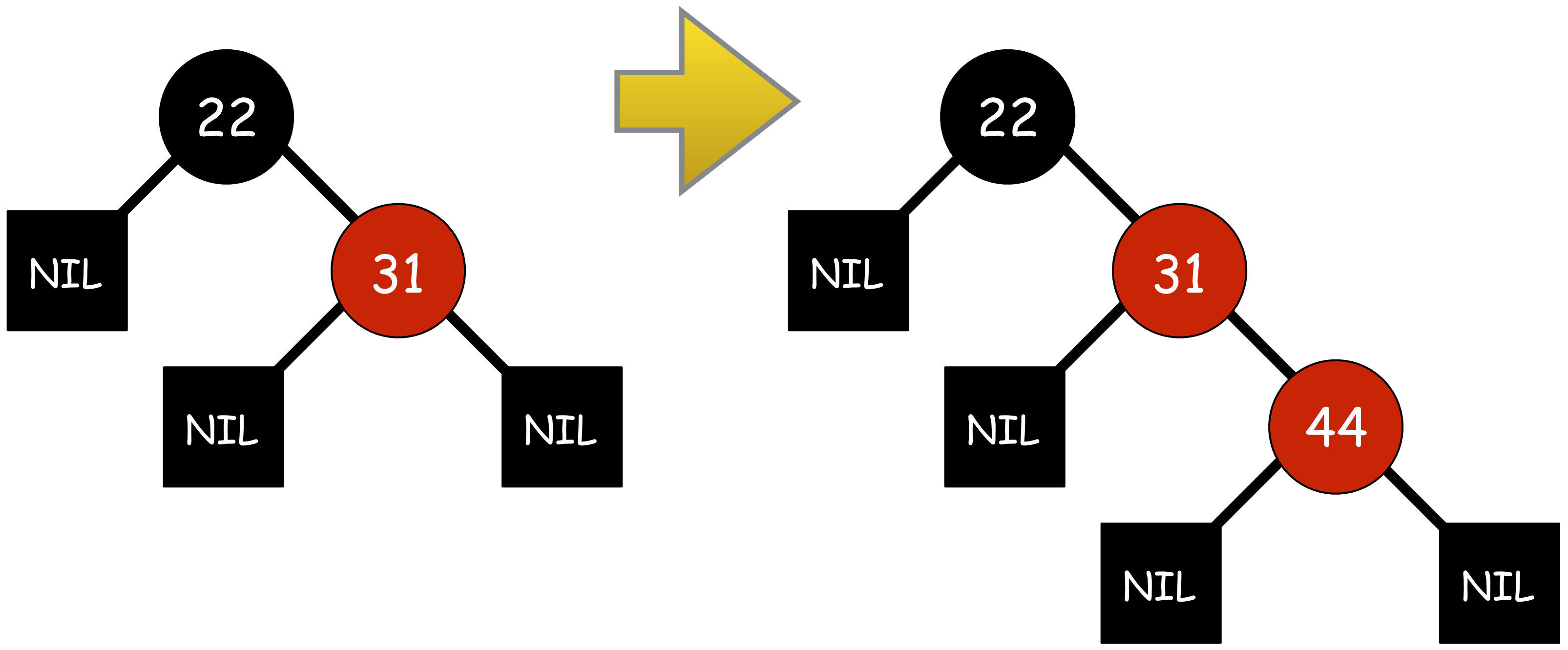
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



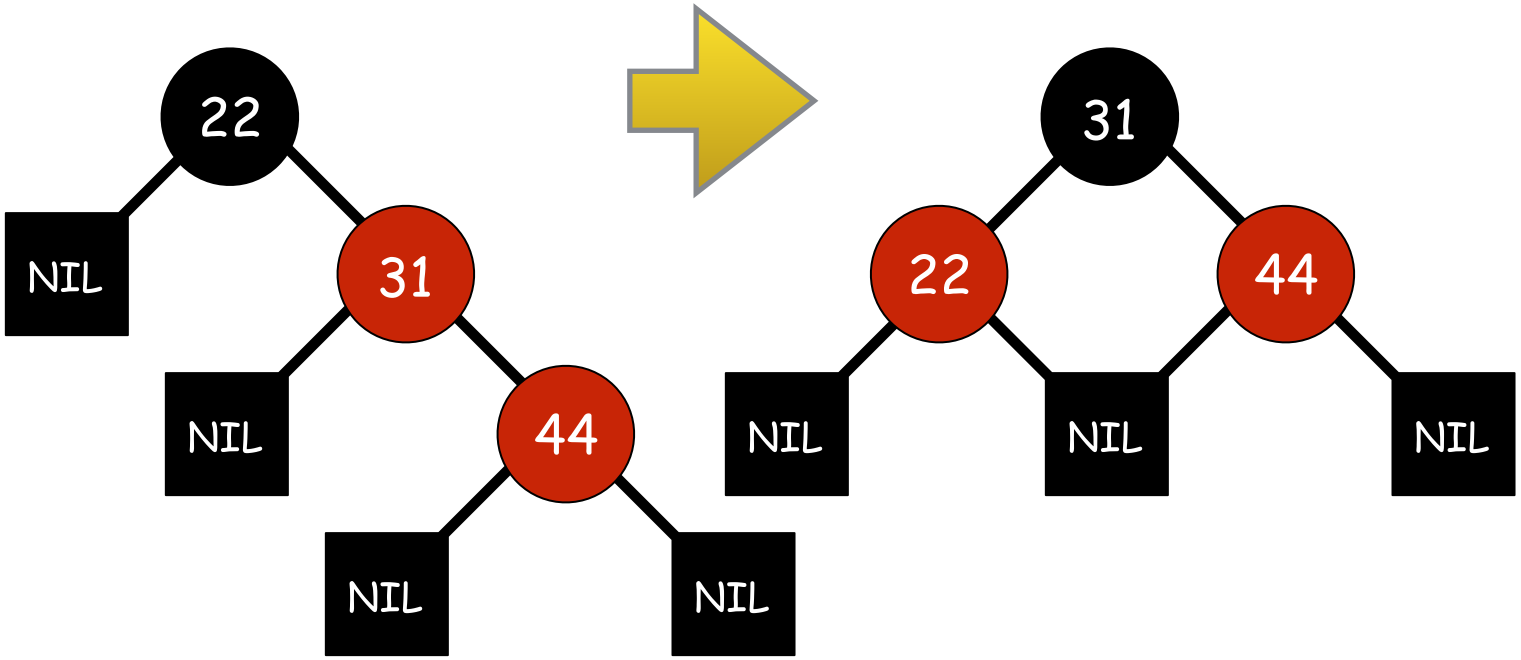
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



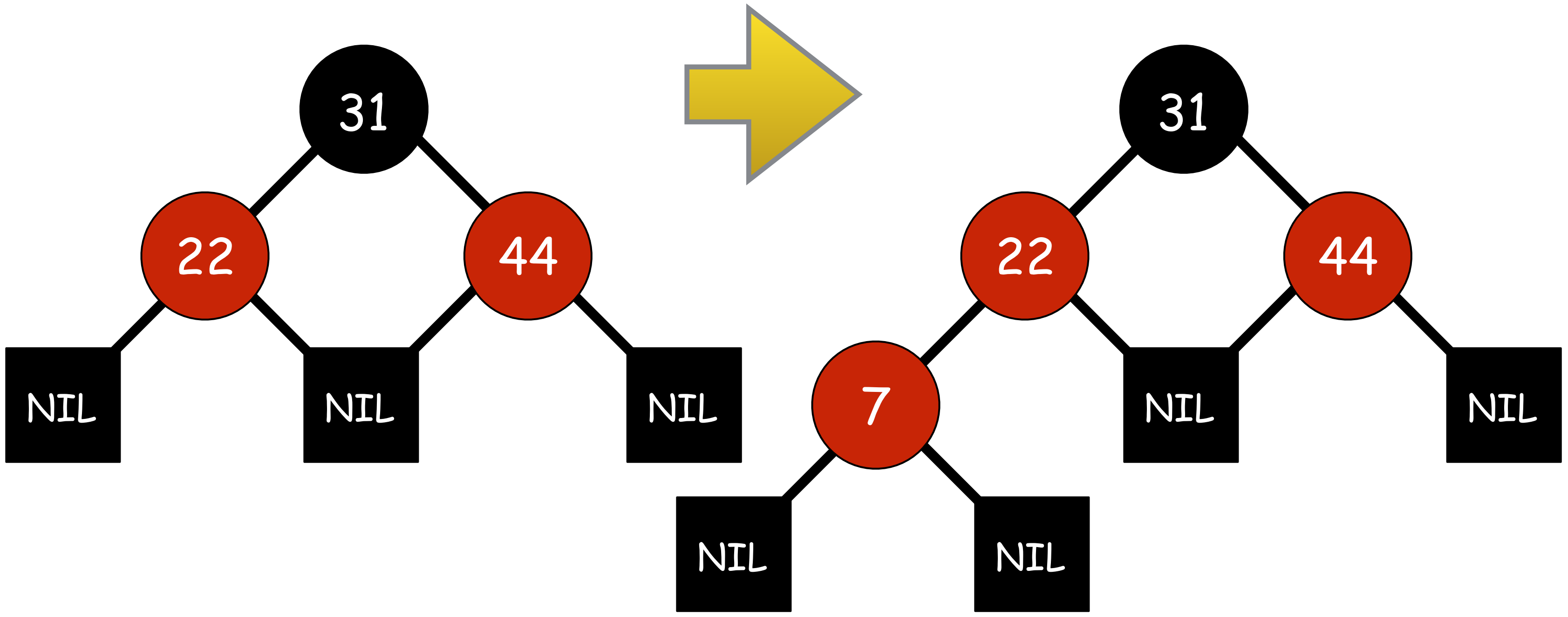
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



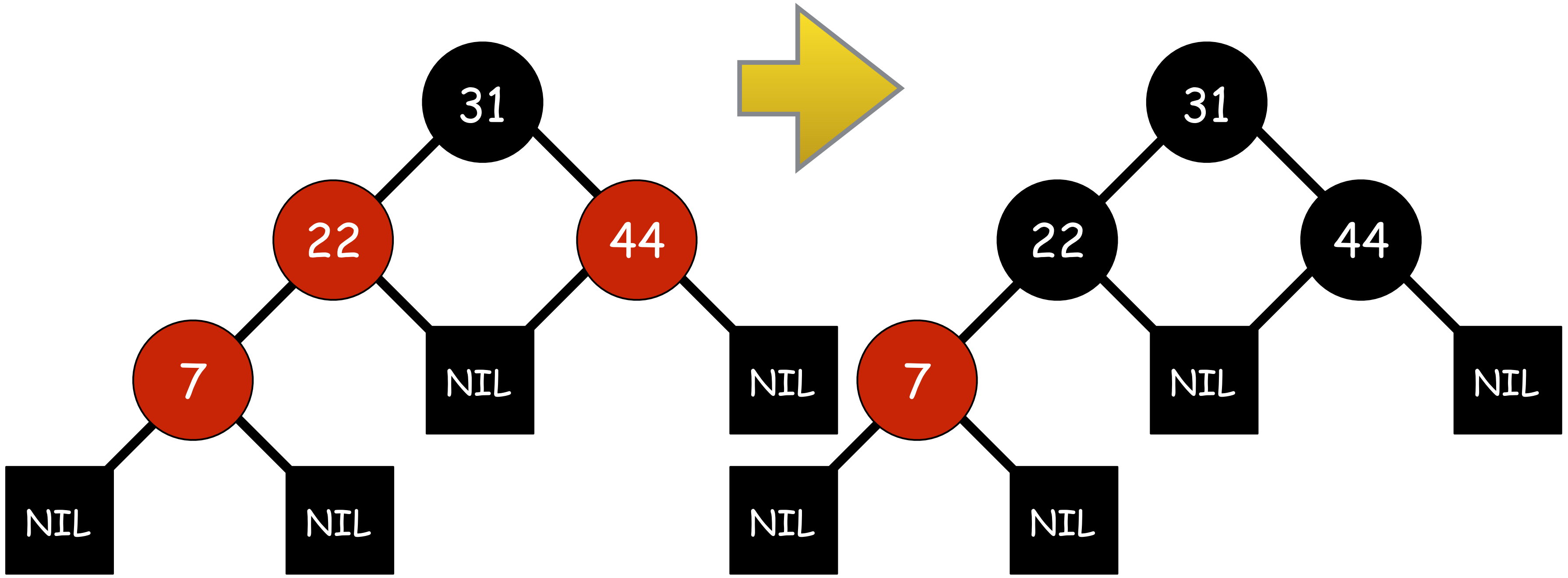
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



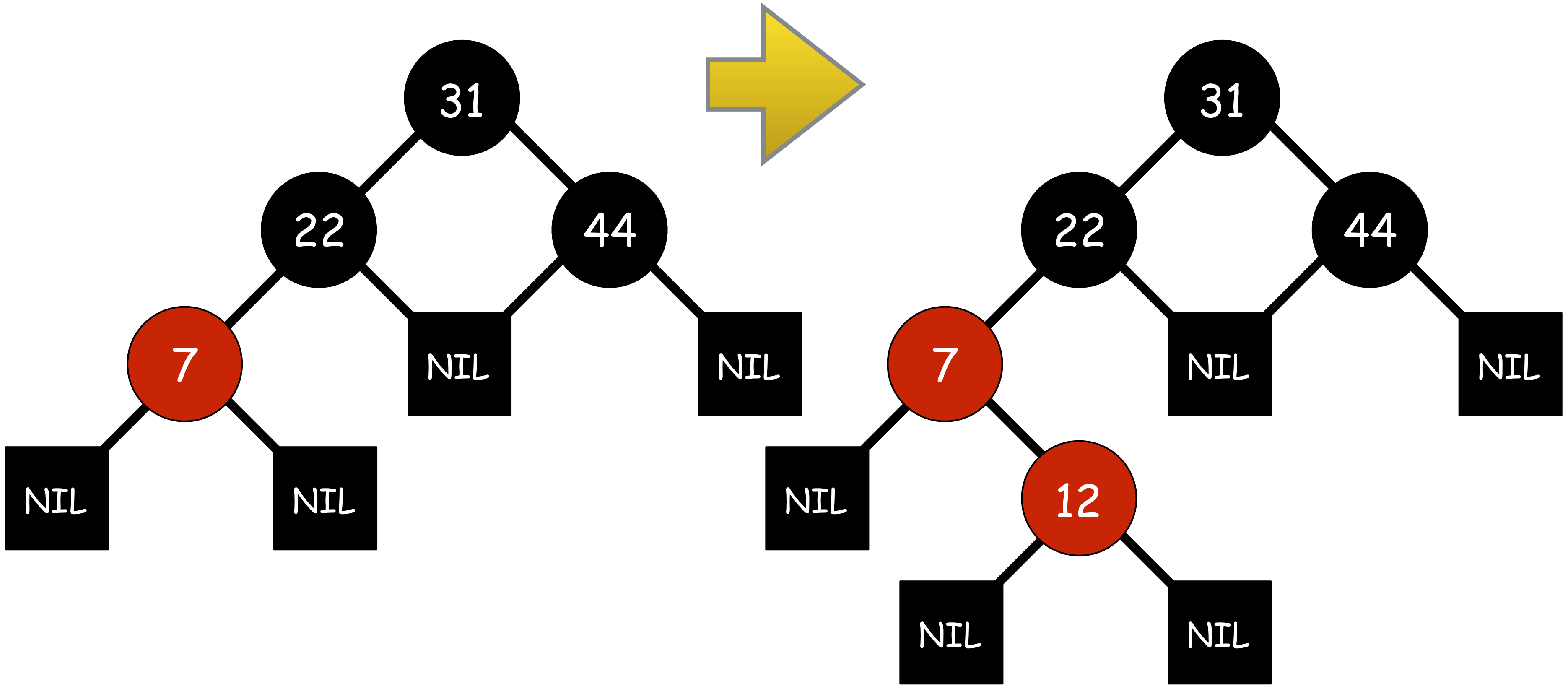
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



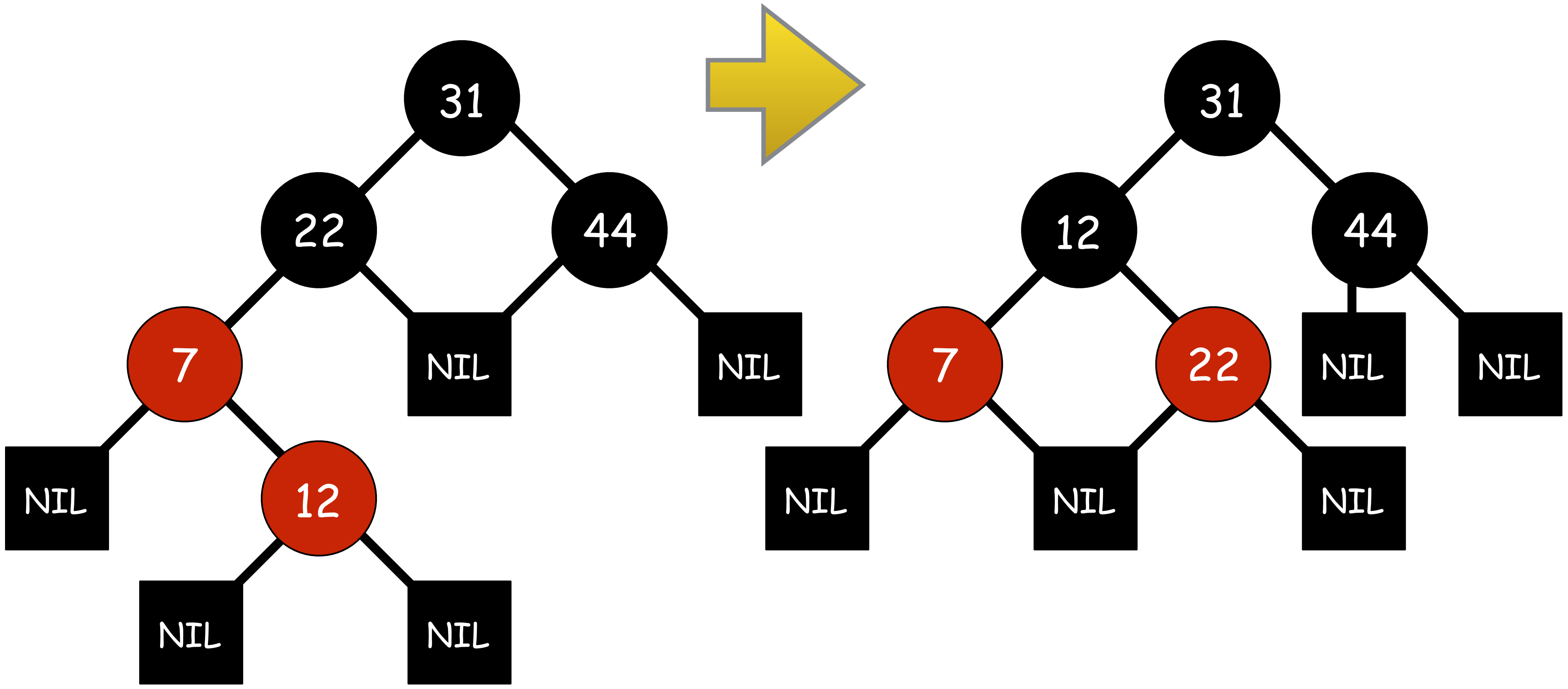
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



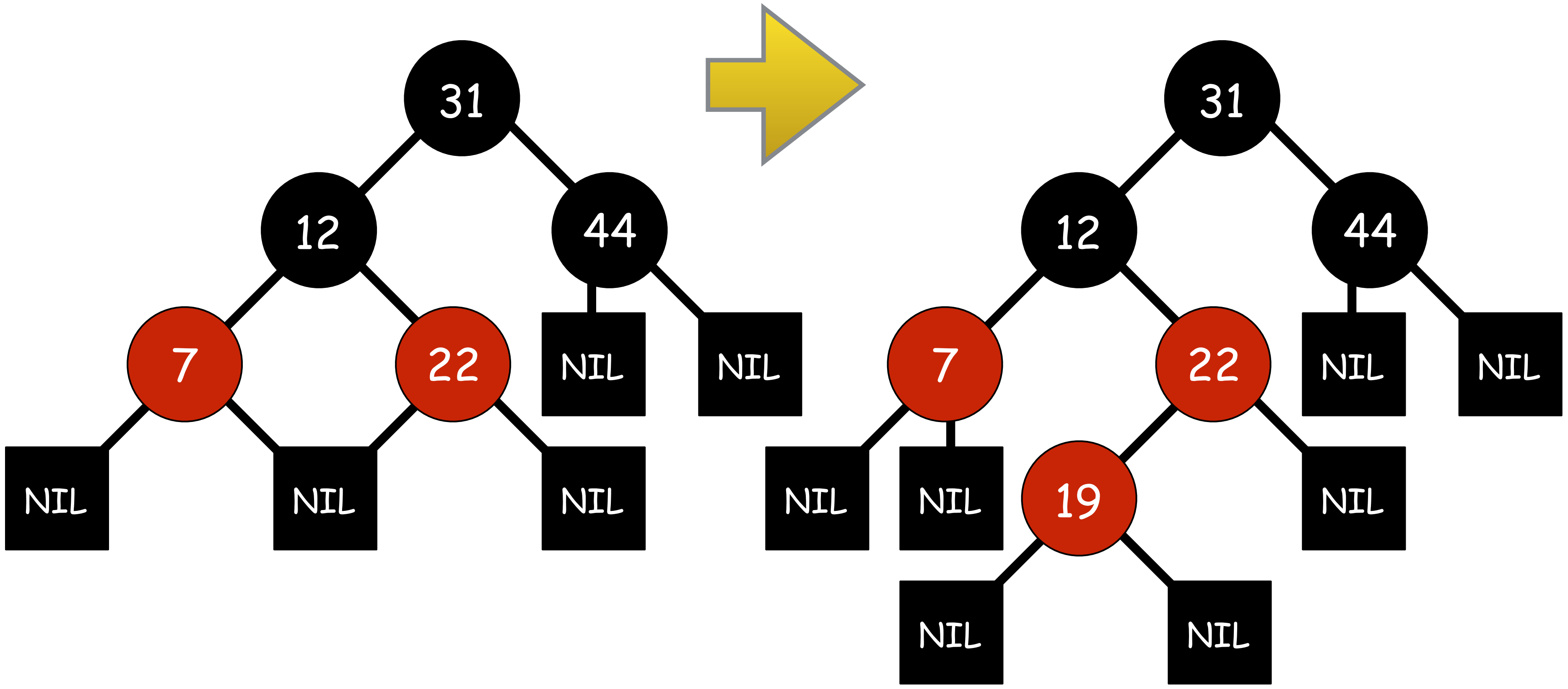
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



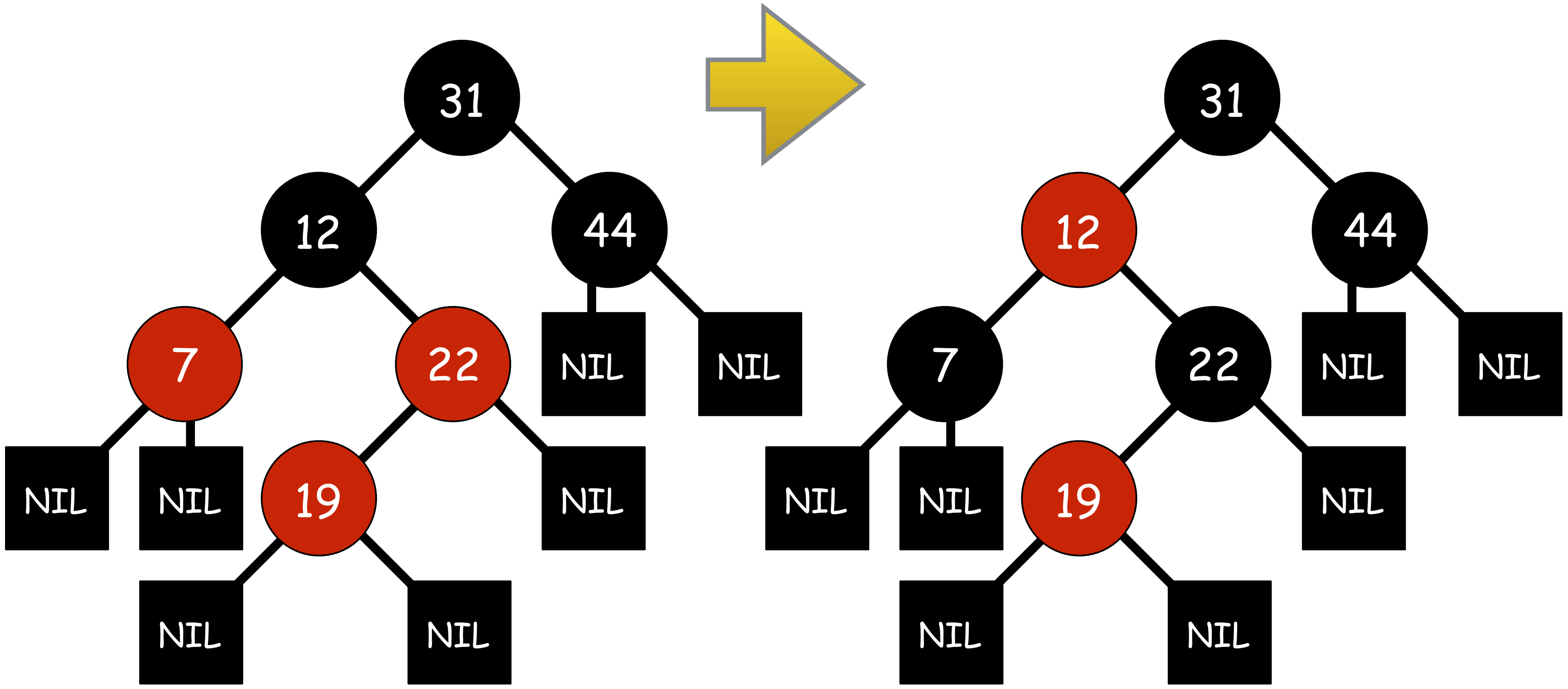
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



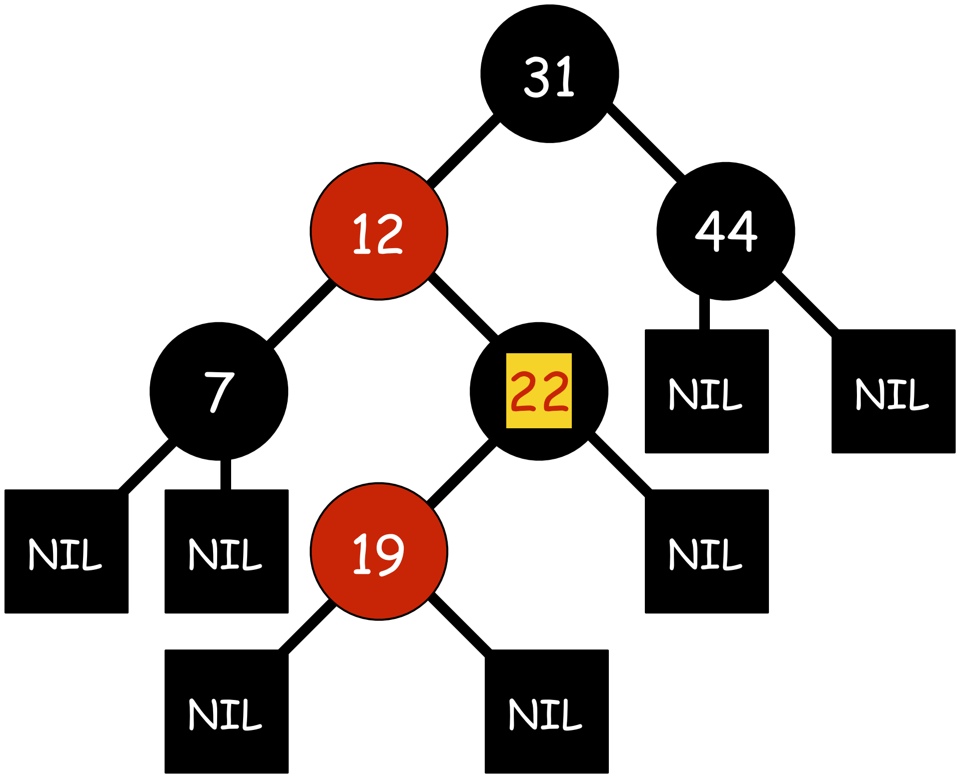
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees



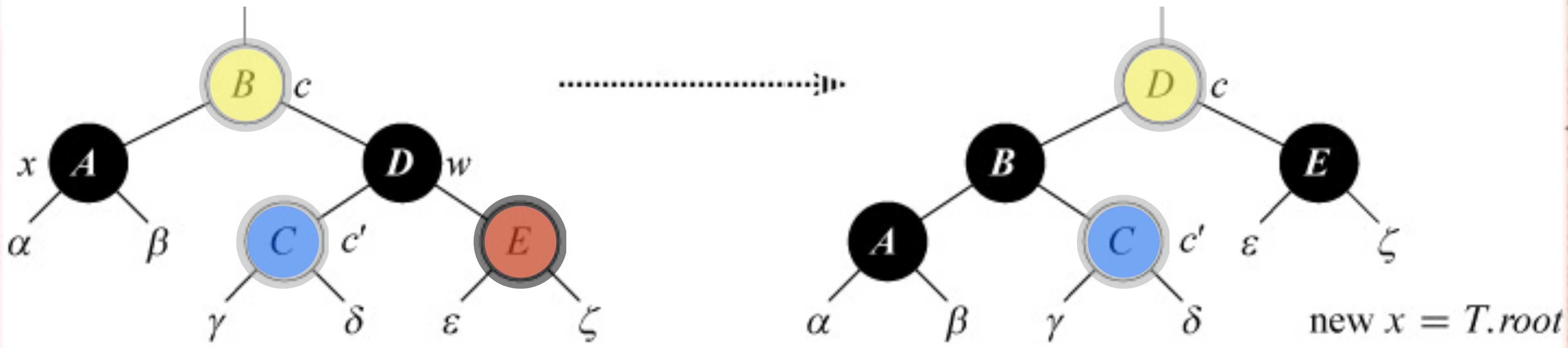
22	31	44	7	12	19	22	35	3	40	27

First Repetition from RED-BLACK Trees

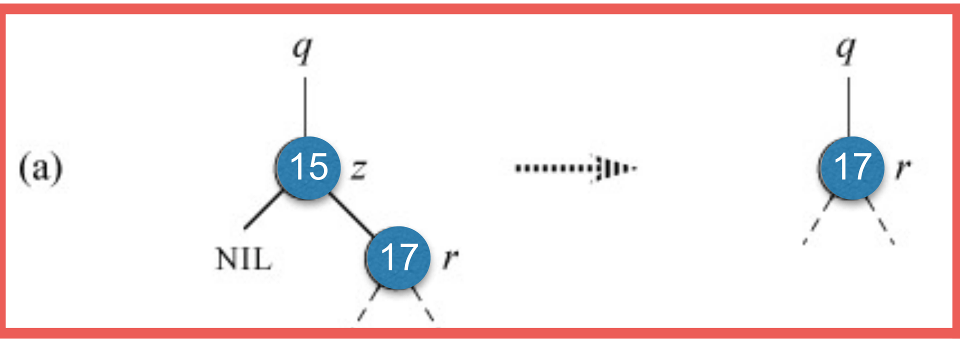


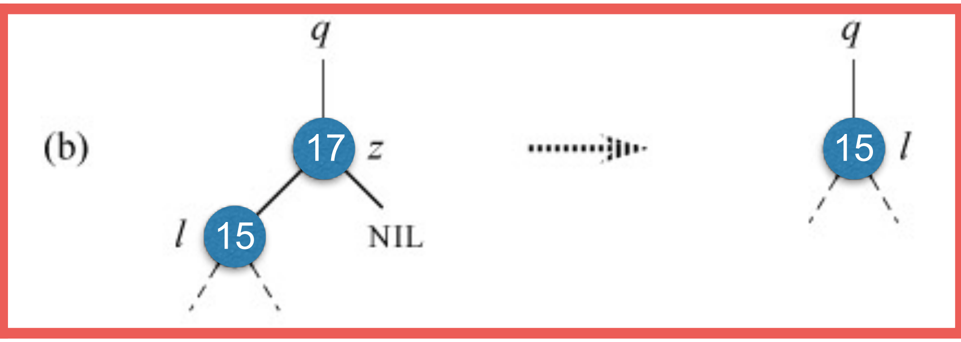
22	31	44	7	12	19	22	35	3	40	27

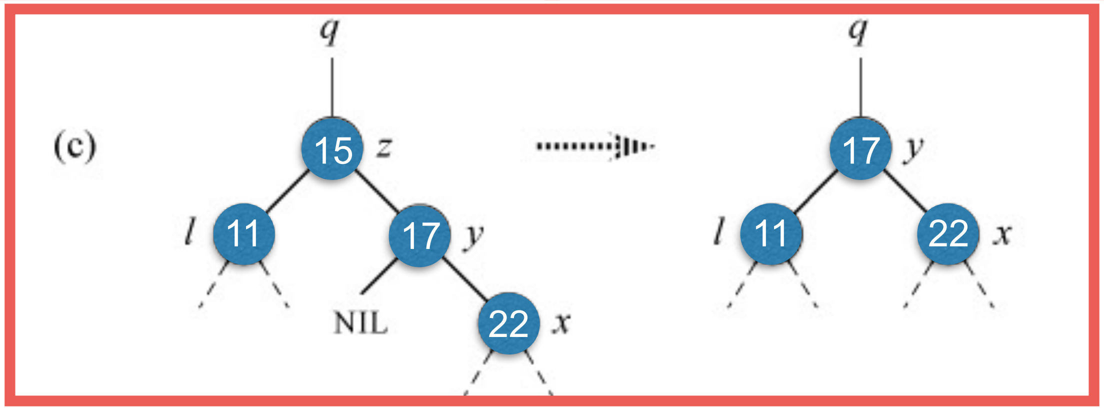
RED-BLACK Tree DELETION

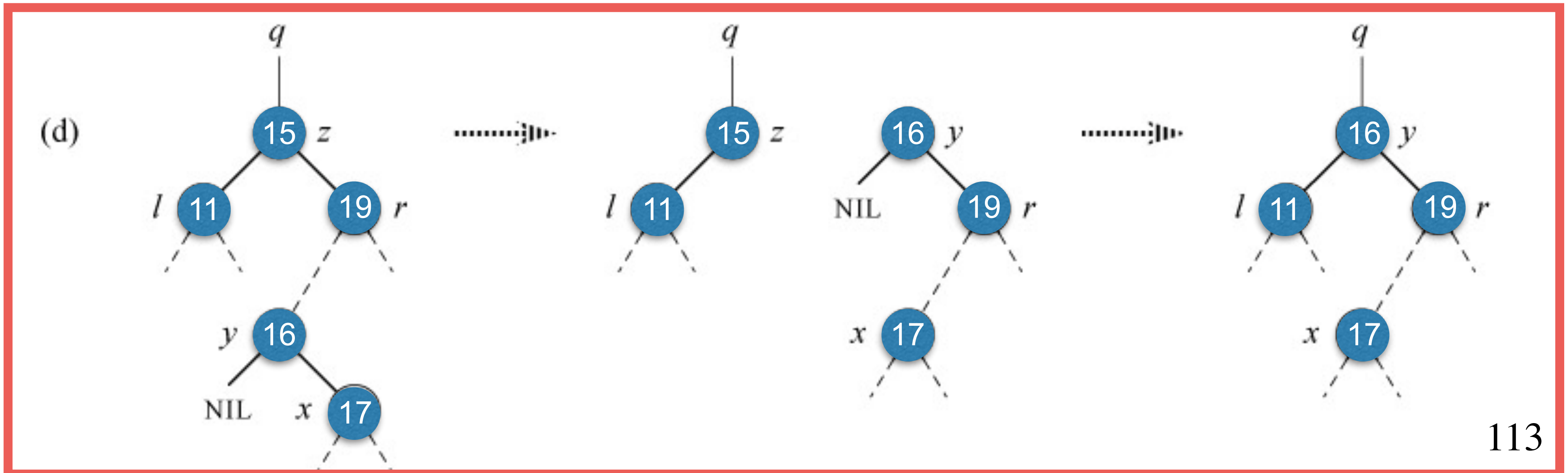


RED-BLACK Trees deletion









RED-BLACK Trees deletion

```
RB-DELETE(T, z)
 1 y = z
 2 y-original-color = y.color
 3 if z.left == T.nil                                   (a)
 4     x = z.right
 5     RB-TRANSPLANT(T, z, z.right)
 6 elseif z.right == T.nil                              (b)
 7         x = z.left
 8         RB-TRANSPLANT(T, z, z.left)
 9     else y = TREE-MINIMUM(z.right)                   (c) & (d)
10         y-original-color = y.color
11         x = y.right
12         if y.p == z                                  (d) only
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)                       (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)
```

RED-BLACK Trees deletion

If node *y* was **BLACK**, three problems may arise, which the call of **RB-DELETEFIXUP** will remedy.

1. if ***y*** had been the root and a **RED** child of ***y*** becomes the new root, **property 2** is violated.

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 violated by any ancestor of *y* in the tree.

   …

RED-BLACK Trees deletion

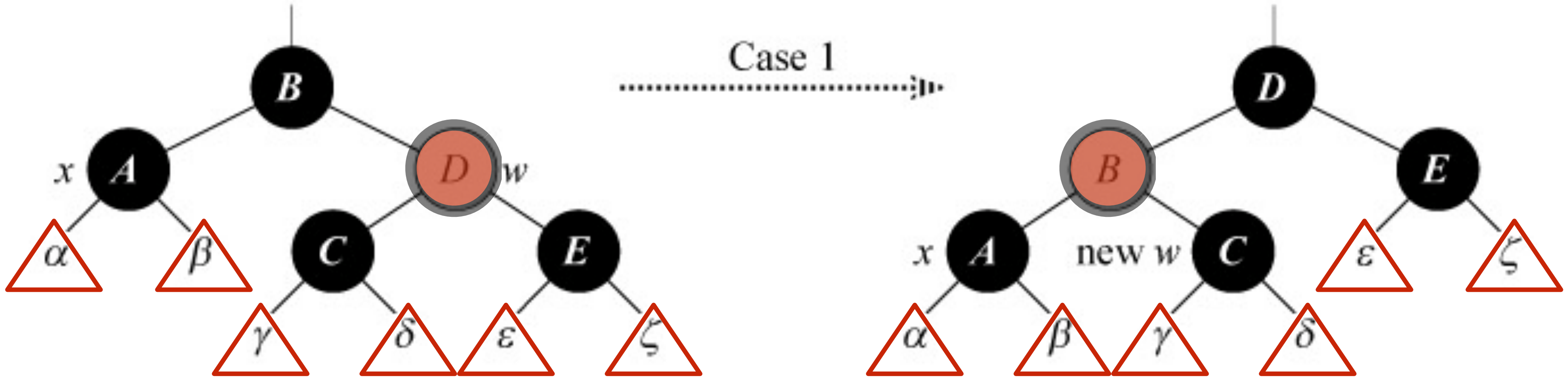
3. Moving ***y*** within the tree causes any simple path that previously contained ***y*** to have one fewer **BLACK** node. Thus, **property 5** is violated by any ancestor of *y* in the tree.

   We can restore **property 5** by saying that node ***x***, now occupying ***y***'s original position, has an "extra" **BLACK**. That is, if we add 1 to the count of **BLACK** nodes on any simple path that contains ***x***, then under this interpretation, **property 5** holds.

   When we remove or move the **BLACK** node ***y***, we "push" its **BLACK**ness onto ***x***. The problem is that now node ***x*** is either "doubly **BLACK**" or "**RED**-and-**BLACK**", and it contributes either 2 or 1, respectively, to the count of **BLACK** nodes on simple paths containing ***x***.

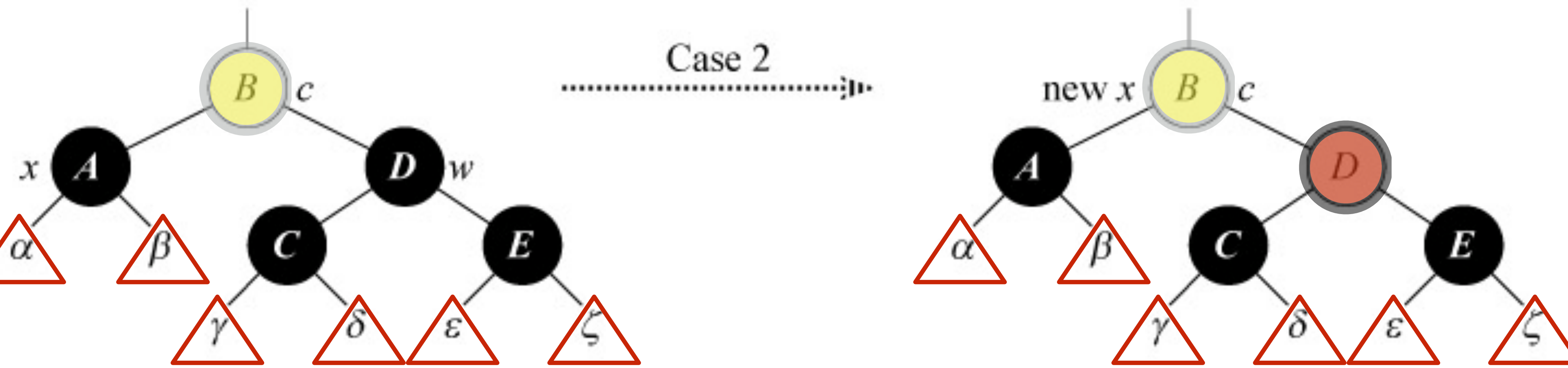
   The color attribute of ***x*** will remain either **RED** (if **RED**-and-**BLACK**) or **BLACK** (if doubly **BLACK**) and the extra **BLACK** on the node is simply indicated by *x* pointing to that node (rather than in the color attribute).

Case 1: x's sibling w is RED (lines 5–8 of **RB-DELETE-FIXUP**)



- w must have black children.
- Make w black and $x.p$ red.
- Then left rotate on $x.p$.
- New sibling of x was a child of w before rotation $\Rightarrow$ must be black.
- Go immediately to case 2, 3, or 4.

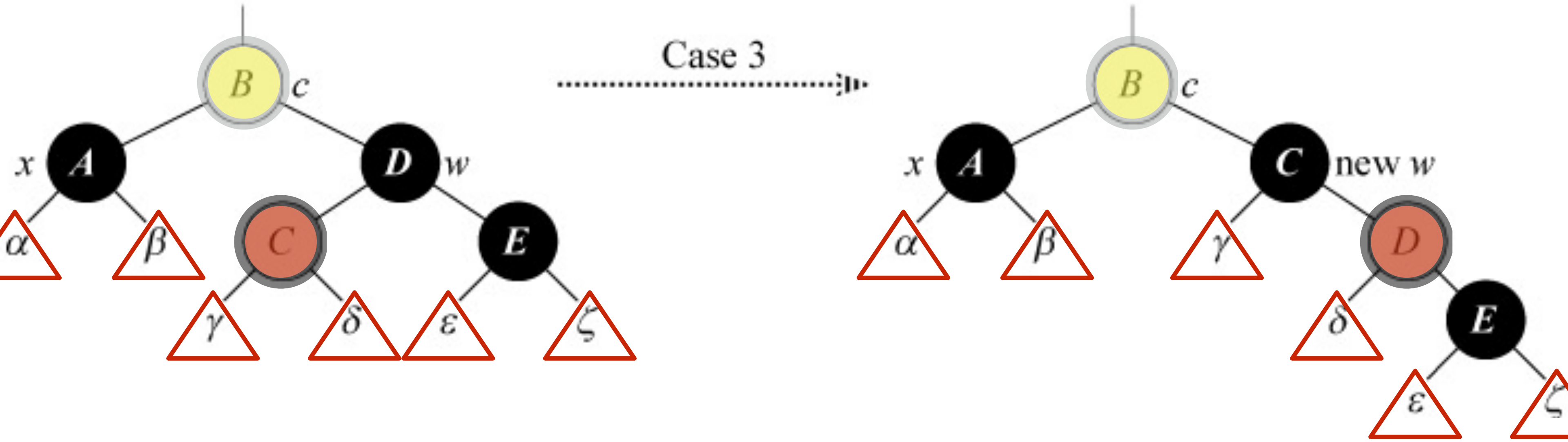
Case 2: *x*'s sibling *w* is BLACK, and both of *w*'s Children are BLACK (lines 10–11 of **RB-DELETE-FIXUP**)



[Node with gray outline is of unknown color, denoted by c.]

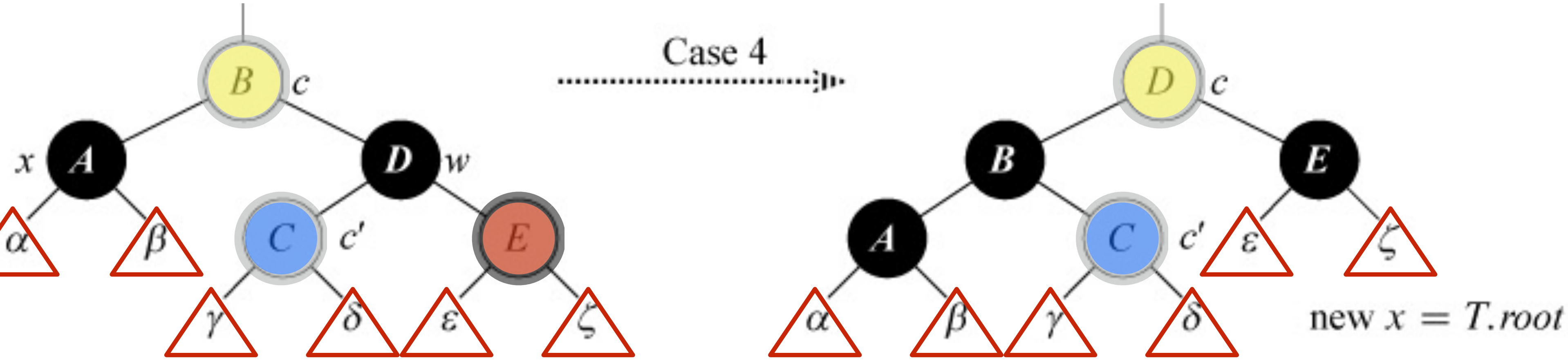
- Take 1 black off x ($\Rightarrow$ singly black) and off w ($\Rightarrow$ red).
- Move that black to $x.p$.
- Do the next iteration with $x.p$ as the new x.
- If entered this case from case 1, then $x.p$ was red $\Rightarrow$ new x is red & black $\Rightarrow$ color attribute of new x is RED $\Rightarrow$ loop terminates. Then new x is made black in the last line.

Case 3: *w* is BLACK, *w*'s left child is RED, and *w*'s right child is BLACK (lines 13–16 of **RB-DELETE-FIXUP**)



- Make w red and w's left child black.
- Then right rotate on w.
- New sibling w of x is black with a red right child $\Rightarrow$ case 4.

Case 4: *x*'s sibling *w* is BLACK, and w's right child is RED (lines 17–21 of **RB-DELETE-FIXUP**)

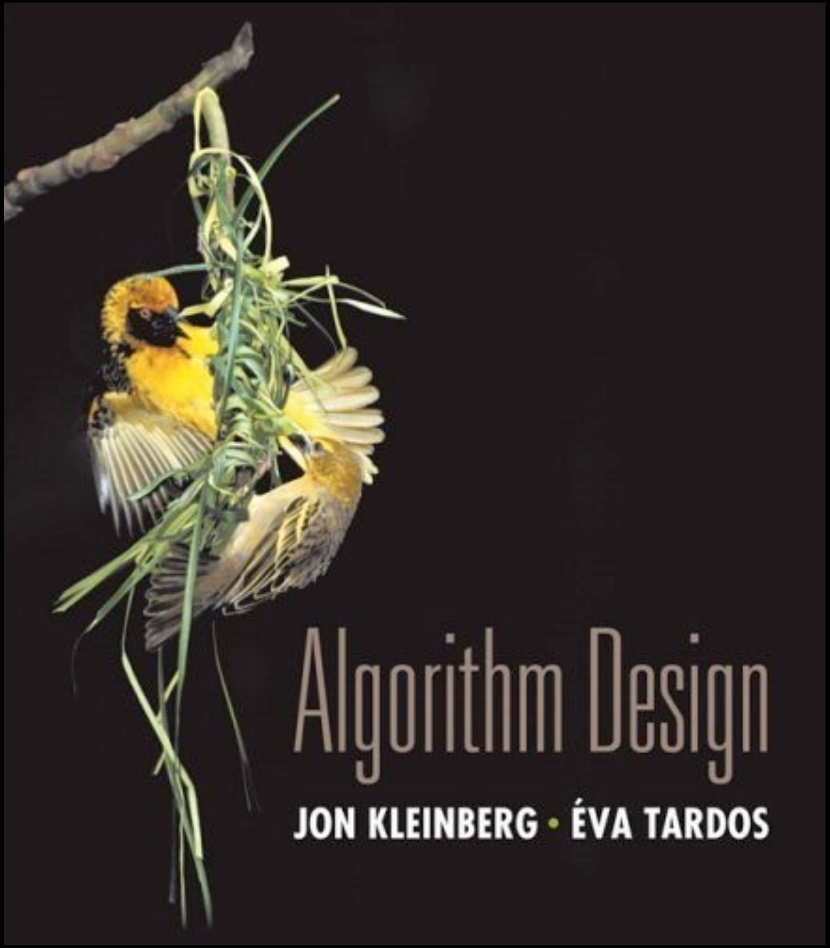


[Now there are two nodes of unknown colors, denoted by c and c′.]

- Make w be $x.p$'s color (c).
- Make $x.p$ black and w's right child black.
- Then left rotate on $x.p$.
- Remove the extra black on x ($\Rightarrow$ x is now singly black) without violating any red-black properties.
- All done. Setting x to root causes the loop to terminate.

RB-DELETE-FIXUP (*T*, *x*)

```
 1 while $x \neq T.root$ and $x.color$ == BLACK
 2     if $x$ == $x.p.left$
 3         $w = x.p.right$
 4         if $w.color$ == RED
 5             $w.color$ = BLACK                          // case 1
 6             $x.p.color$ = RED                          // case 1
 7             LEFT-ROTATE($T$, $x.p$)                    // case 1
 8             $w = x.p.right$                            // case 1
 9         if $w.left.color$ == BLACK and $w.right.color$ == BLACK
10             $w.color$ = RED                            // case 2
11             $x = x.p$                                  // case 2
12         else if $w.right.color$ == BLACK
13                 $w.left.color$ = BLACK                 // case 3
14                 $w.color$ = RED                        // case 3
15                 RIGHT-ROTATE($T$, $w$)                 // case 3
16                 $w = x.p.right$                        // case 3
17             $w.color = x.p.color$                      // case 4
18             $x.p.color$ = BLACK                        // case 4
19             $w.right.color$ = BLACK                    // case 4
20             LEFT-ROTATE($T$, $x.p$)                    // case 4
21             $x = T.root$                               // case 4
22     else (same as 3–21 with “right” and “left” exchanged)
23 $x.color$ = BLACK
```



Chapter 3

Graphs

CLRS 12-13