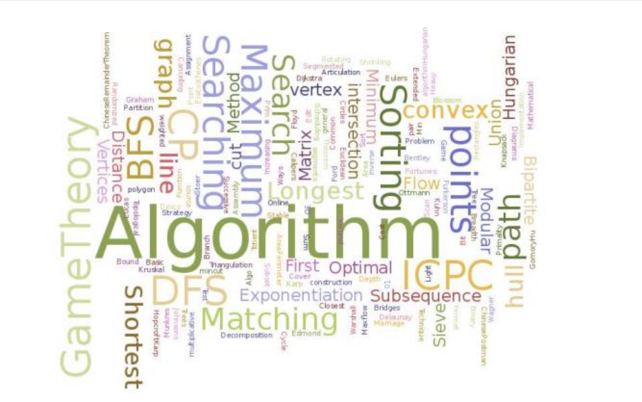
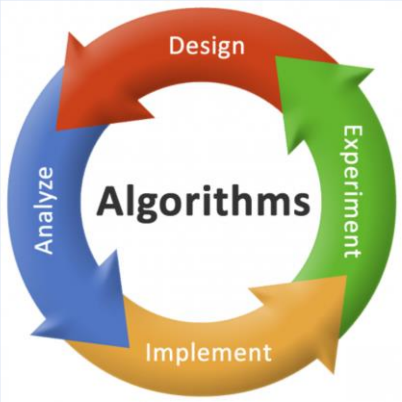
Red-Black Trees

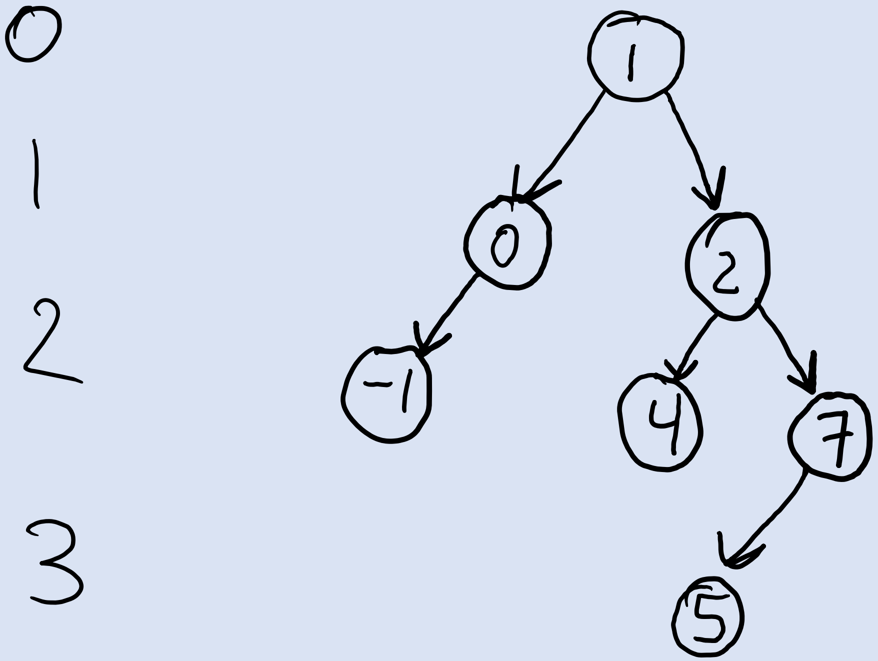
COP 3503
Fall 2021
Department of Computer Science
University of Central Florida
Dr. Steinberg

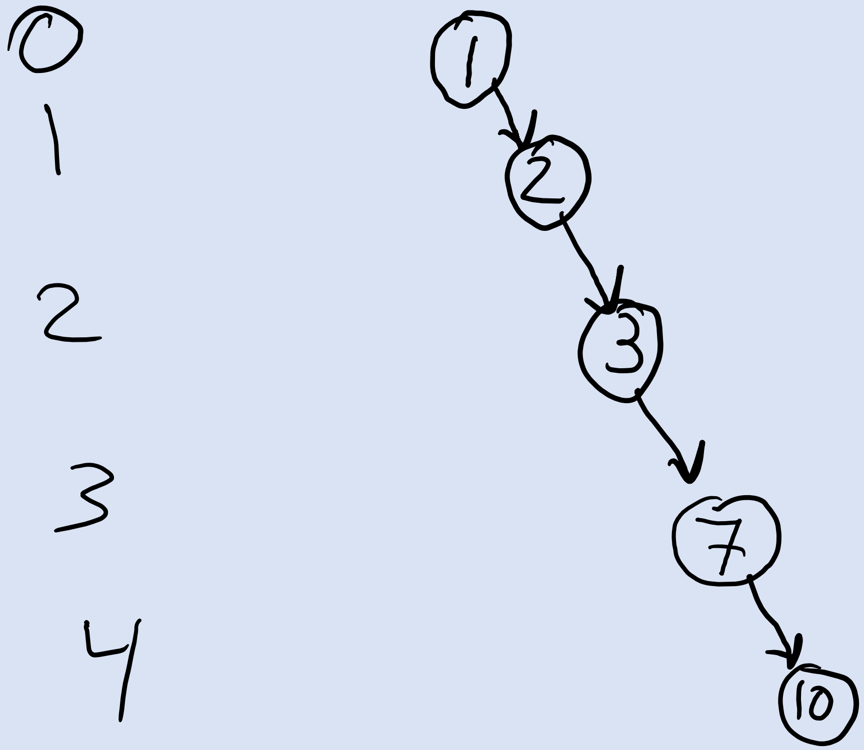




Introduction

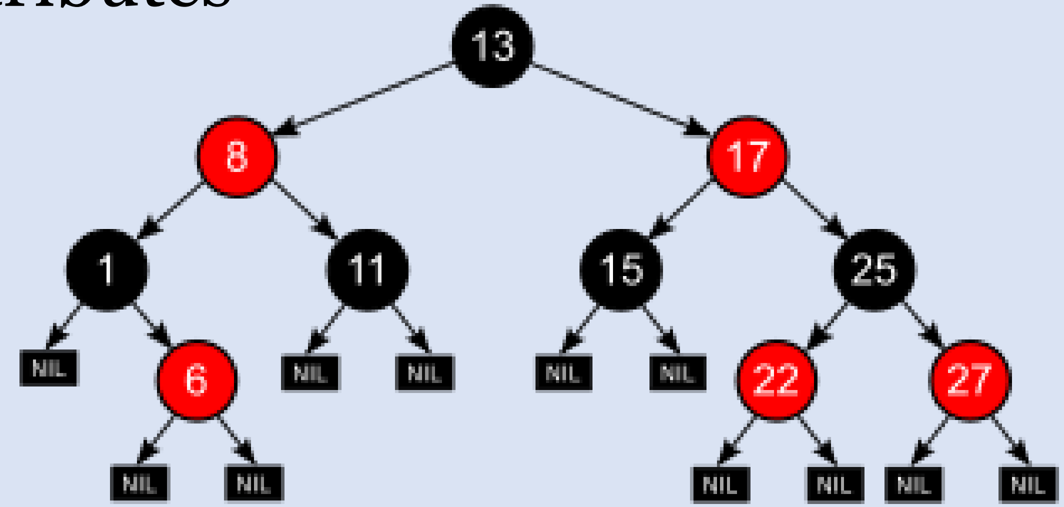
- Binary Search Trees – Trees where a node has only at most two children
- Examples:





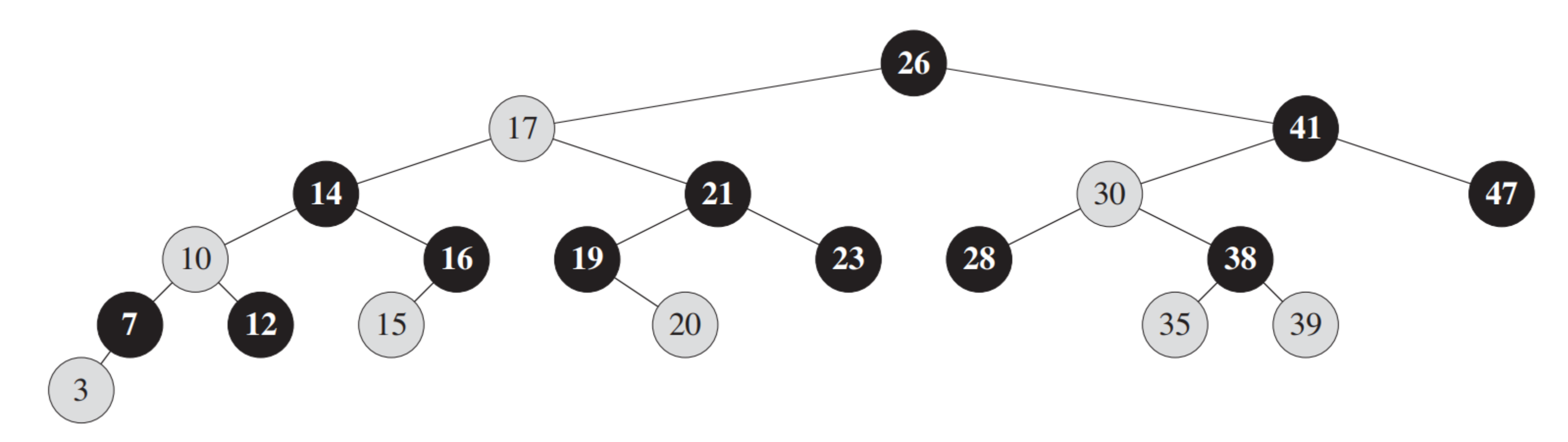
Red-Black Trees

- Red-Black Trees
- Special BST with one extra bit of storage
  - Color RED or BLACK
- Utilizing colors helps ensure the R-B Tree is balanced.
- Each node contains the following attributes
  - color
  - key
  - left
  - right
  - p



Red-Black Tree Properties

1. Every node is either red or black
2. The root is black
3. Every leaf 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 contains the same number of black nodes.



Height Property of R-B Tree and Proof

- A red-black tree with n internal nodes has a height at most $2\lg(n+1)$
- Subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
- If height of node x is 0, then that means it's a leaf. $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes
- Using an inductive step, lets consider node x a positive height and is also an internal node with two children. Each child has a bh(x) or bh(x) – 1 depending on its color (red or black)
- Using inductive hypothesis, each child will have at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = 2^{bh(x)} - 1$ internal nodes.
- Let h be height of tree. Based on definition, half the nodes on any simple path from root to leaf (not including root) must be black. Black height must be $\frac{h}{2}$ which leads to

$$n \geq 2^{\frac{h}{2}} - 1$$

$$n + 1 \geq 2^{\frac{h}{2}}$$

$$\lg(n+1) \geq \frac{h}{2}$$

$$2\lg(n+1) \geq h$$

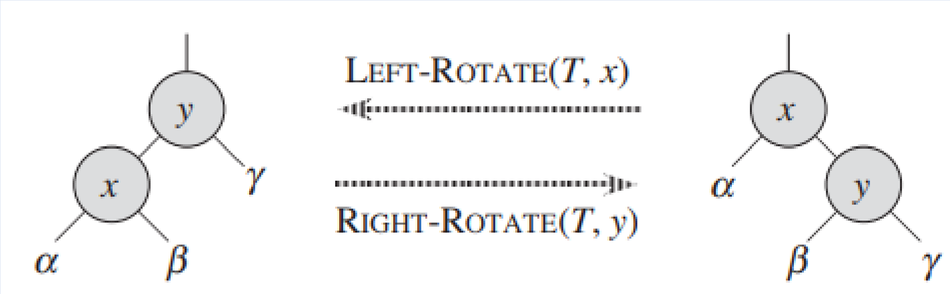
From this, our RT on algorithms involving R-B Trees run in log time!

Insertion in R-B Trees

- When inserting a new key into the tree, we have to be careful in maintaining the properties of the R-B trees. This means we may have to modify it a bit to maintain its properties, so everything stays in log time!

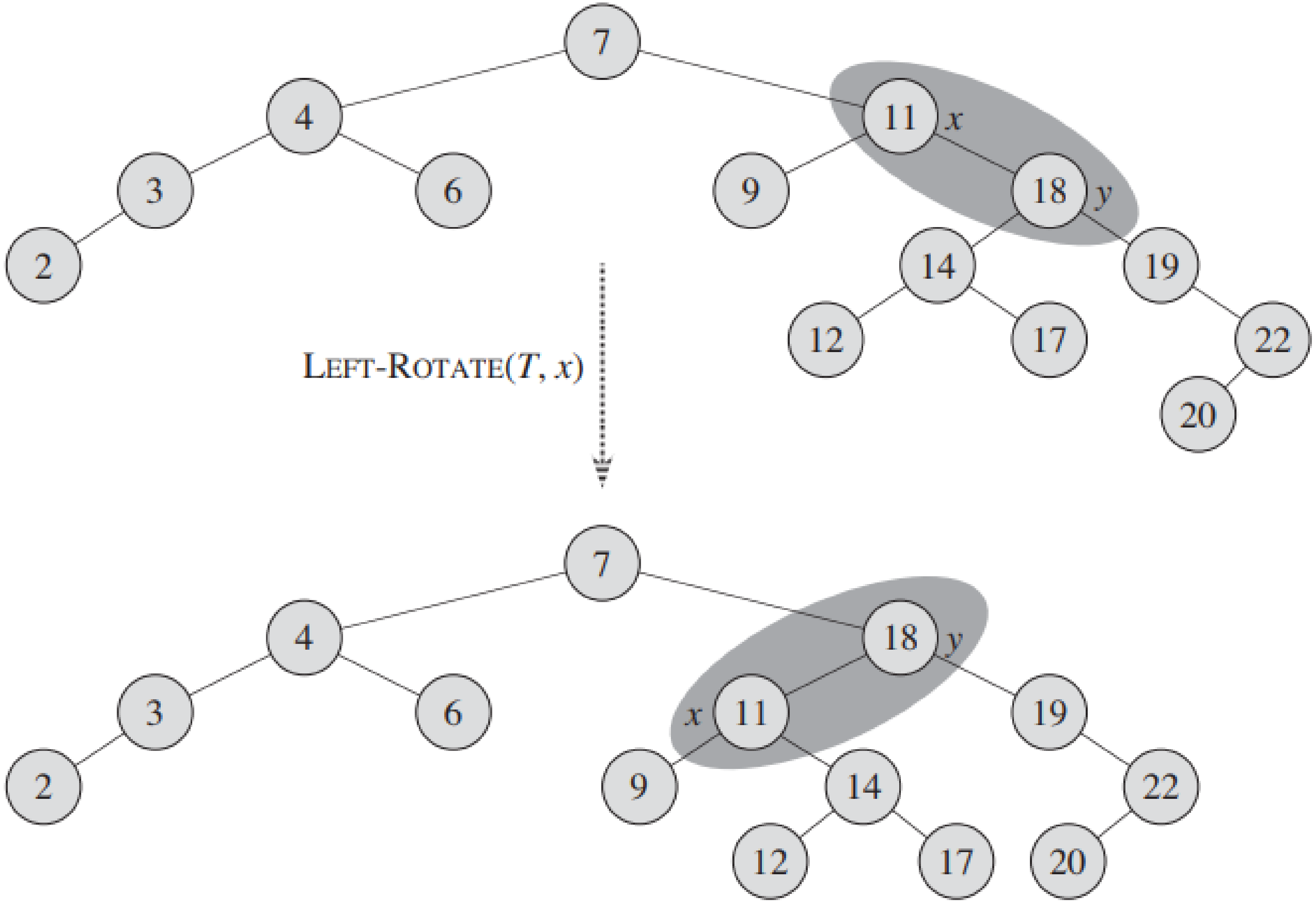
Rotations

- In order to keep the log running times, R-B trees will have to perform a rotation operation, which changes the structure in balancing itself.



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.p.left = y
else x.p.right = y
y.left = x                     // put x on y's left
x.p = y
```



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

Insertion Operation in R-B Trees

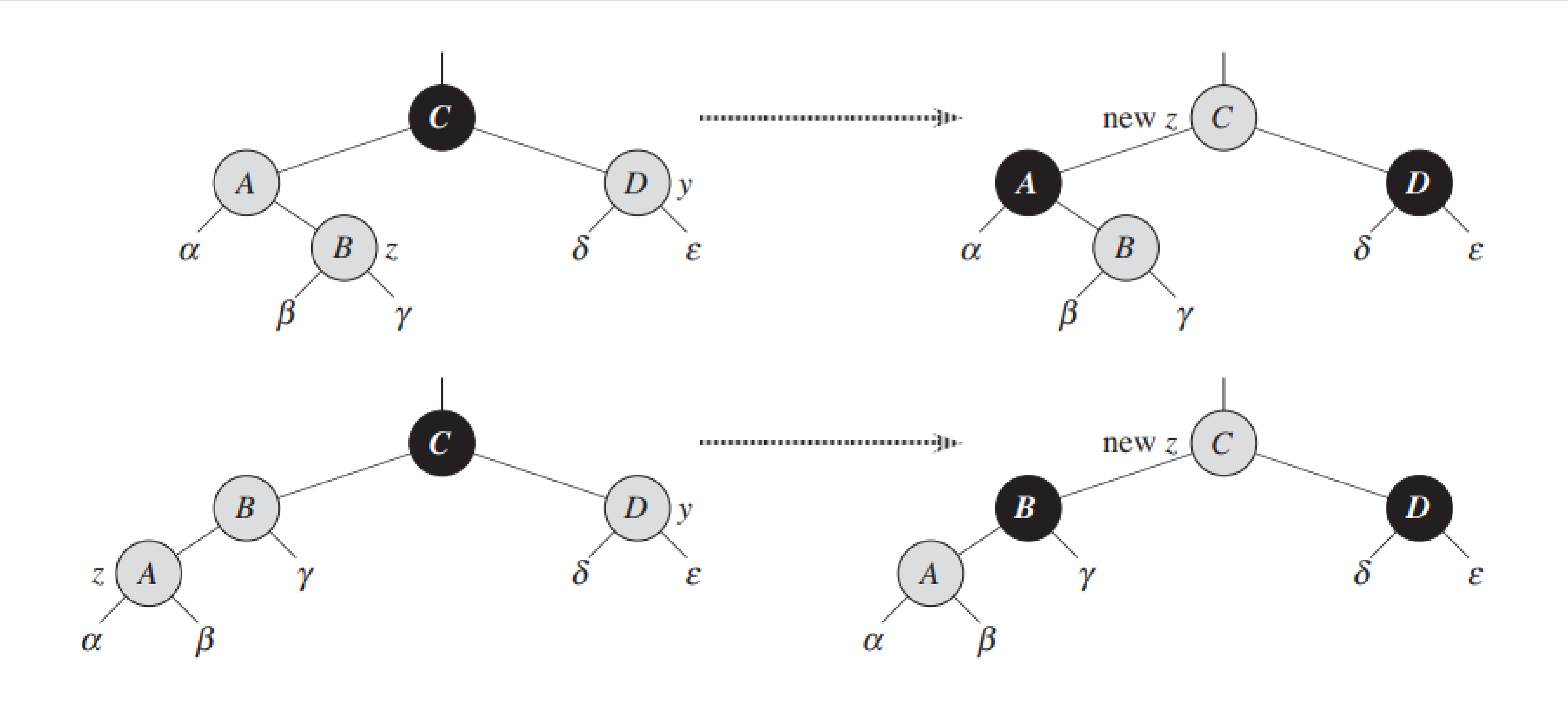
```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

```
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
```

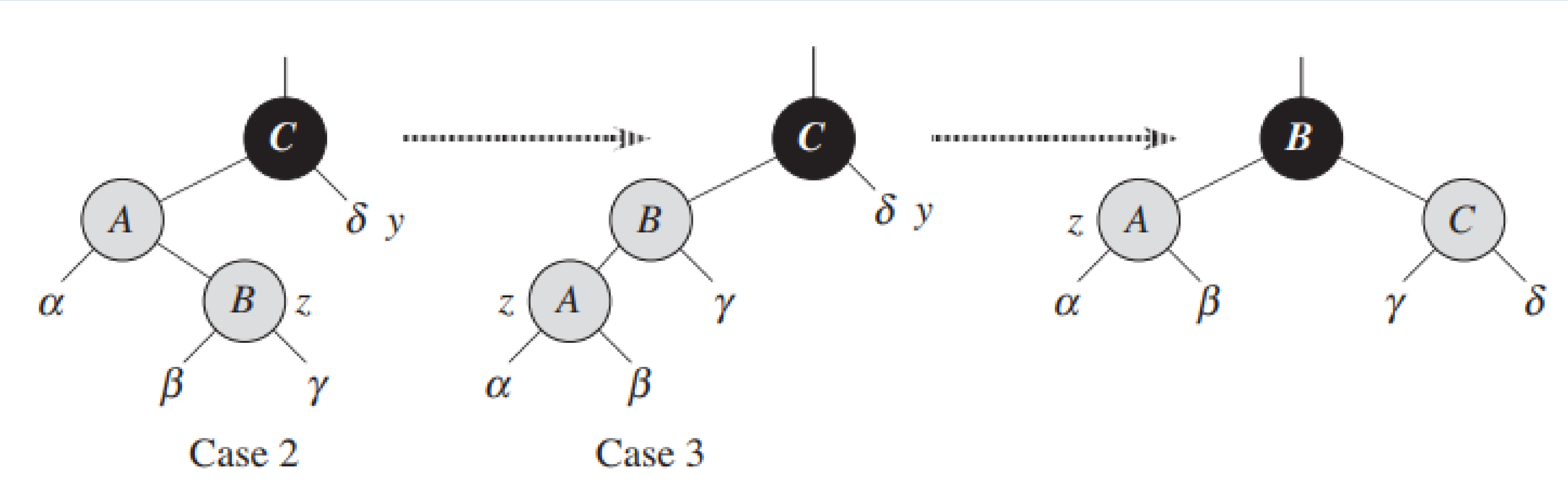
Maintaining the R-B Tree properties

- 3 cases that we must observe after insertion

Case 1 z's uncle y is red



Cases 2 & 3 z’s uncle y is black and z is a right/left child



Insertion Examples

Delete Operation

```
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
```

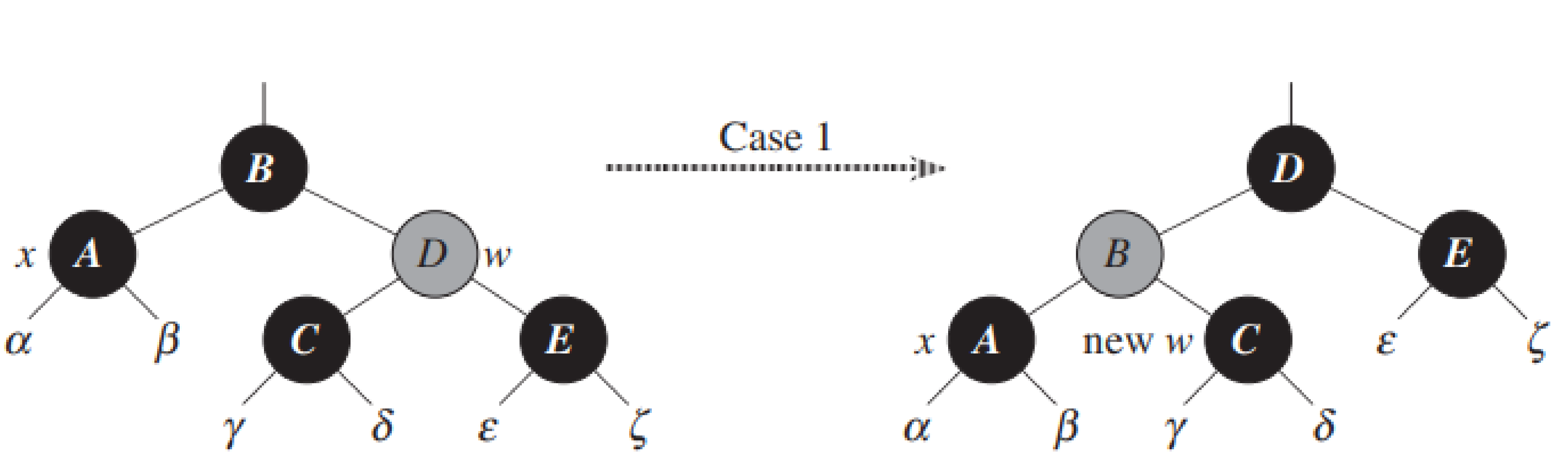
```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y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

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

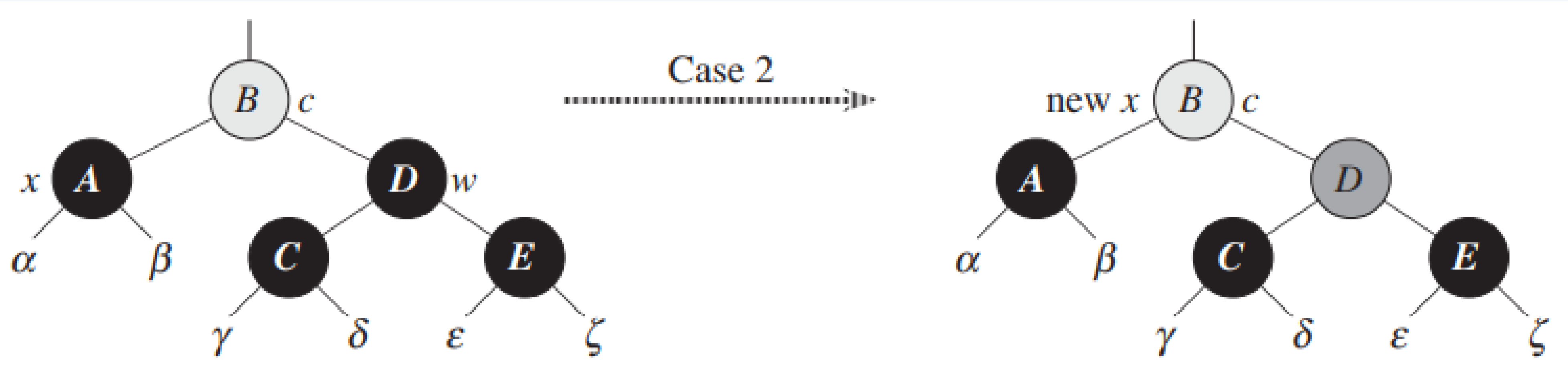
Maintaining the R-B Tree properties

- 4 cases that we must observe after insertion

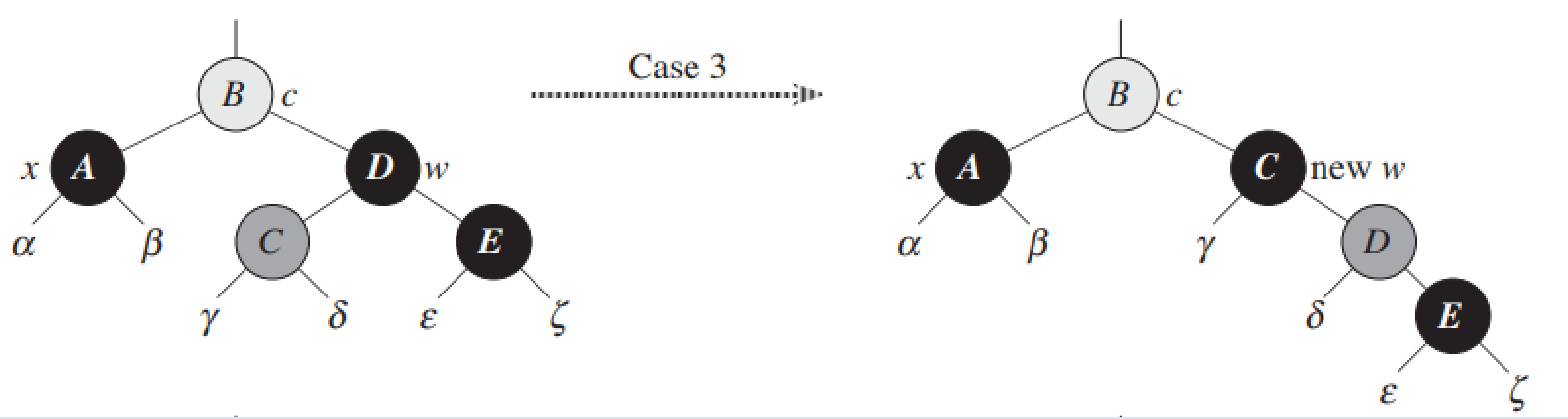
Case 1: x's sibling w is red



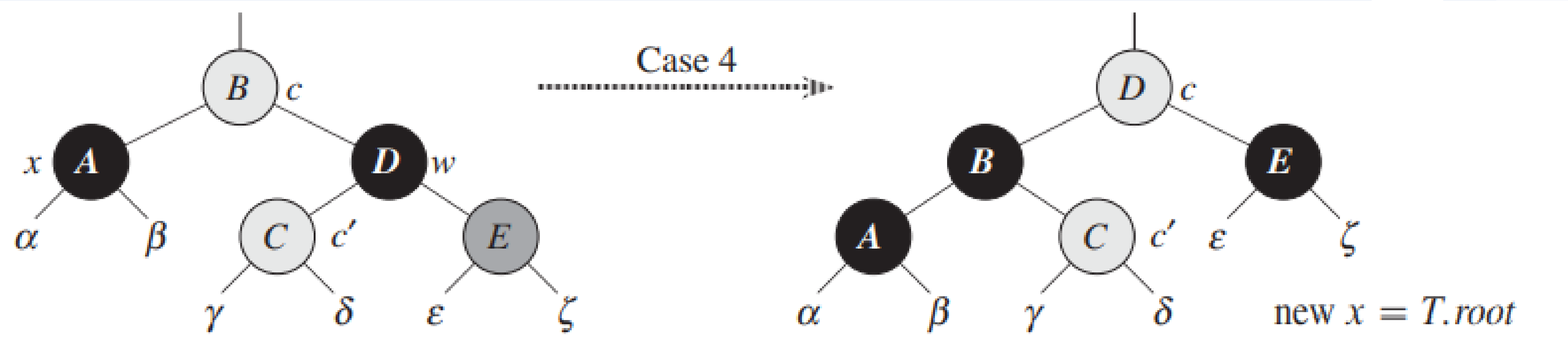
Case 2: x's sibling w is black, and both of w's children are black



Case 3: x's sibling w is black, 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



Delete Examples