CH08-320201
Algorithms and Data Structures

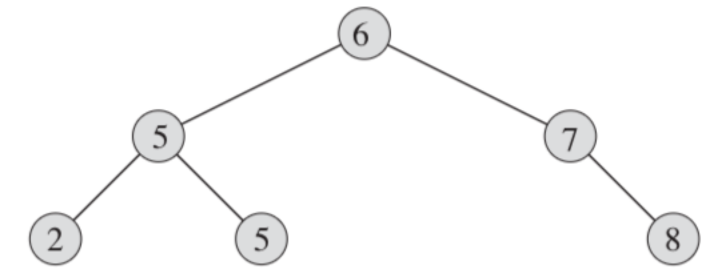
Lecture 15/16 — 10 Apr 2018

Prof. Dr. Michael Sedlmair

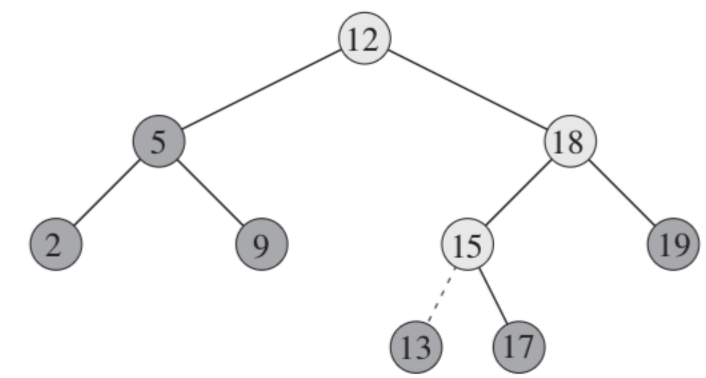
Jacobs University

Spring 2018

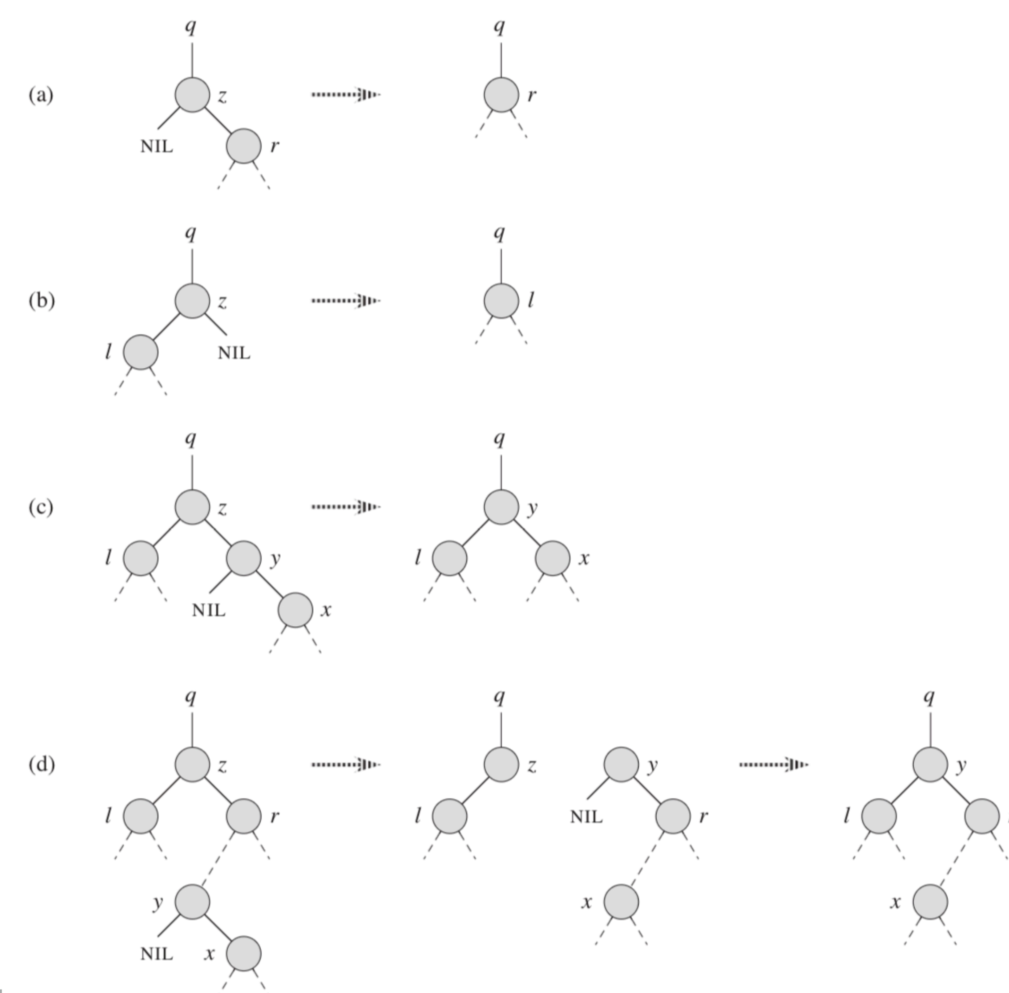
Last time



- **Binary Search Trees (BST)**
  - Let x be a node of the BST.
    - y in the left subtree of x ⇒ y.key ≤ x.key
    - y in the right subtree of x ⇒ y.key ≥ x.key



- Querying
  - Search
  - Minimum & Maximum
  - Successor & Predecessor
- Modifying
  - Tree-Insert
  - Tree-Delete
- Running time: O(h)
- Problems?



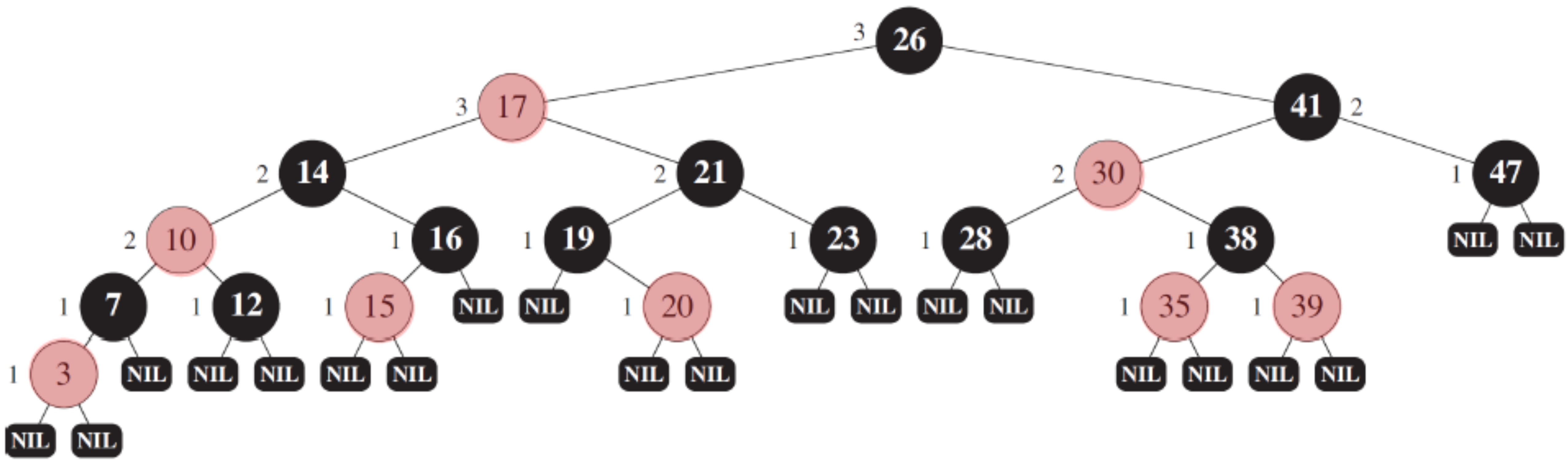
3.5 Red-black Trees

Concept

- A red-black tree is a **BST** that besides the attributes about parent, left child, right child, and key holds the attribute of a color (**red** or **black**), which is encoded in one additional bit.
- Special convention: All leaves have NIL key.
- The node colors are used to impose constraints on the nodes such that no path from the root to a leaf is more than twice as long as any other path.
- Hence, the tree is **approximately balanced.**

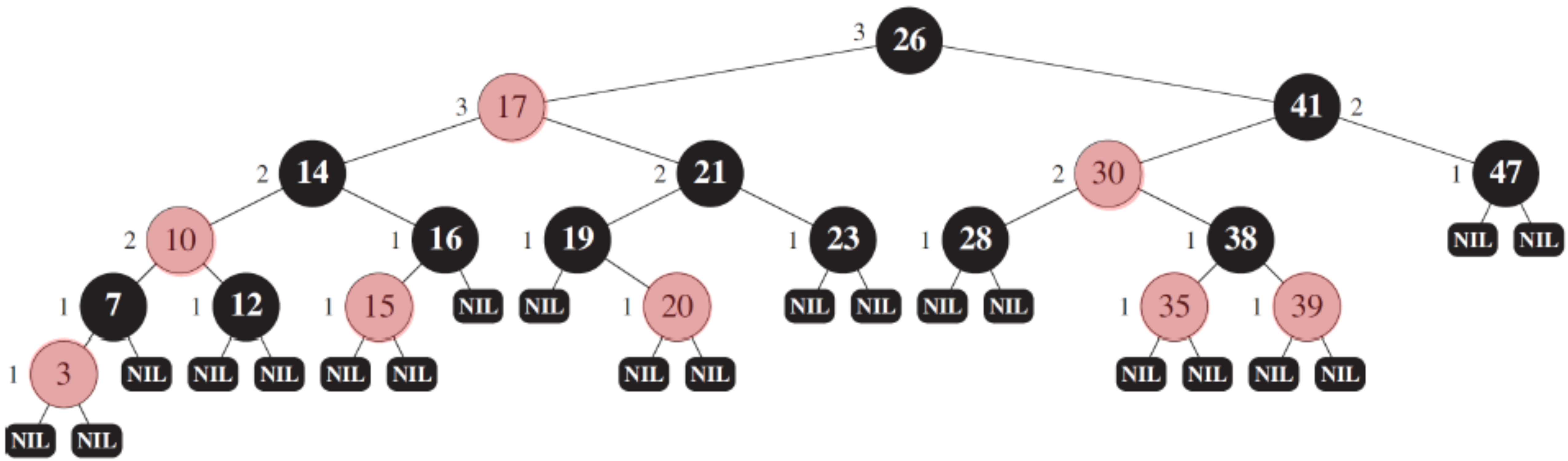
Property 1 (Duh property)

- Every node is either red or black.



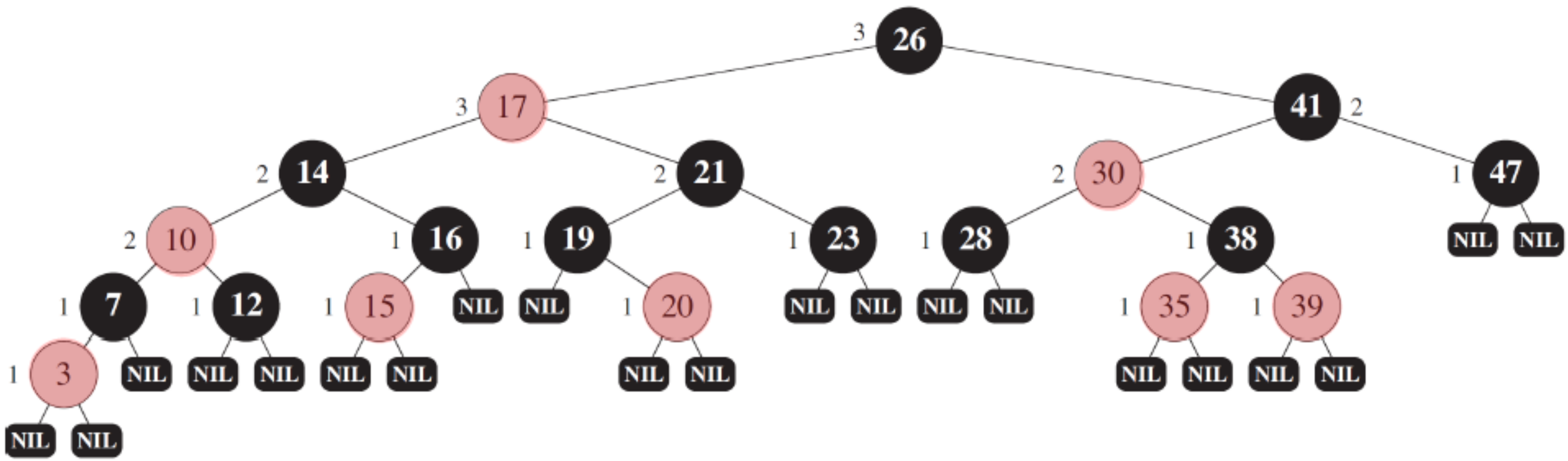
Property 2 (RooB property)

- The root is black



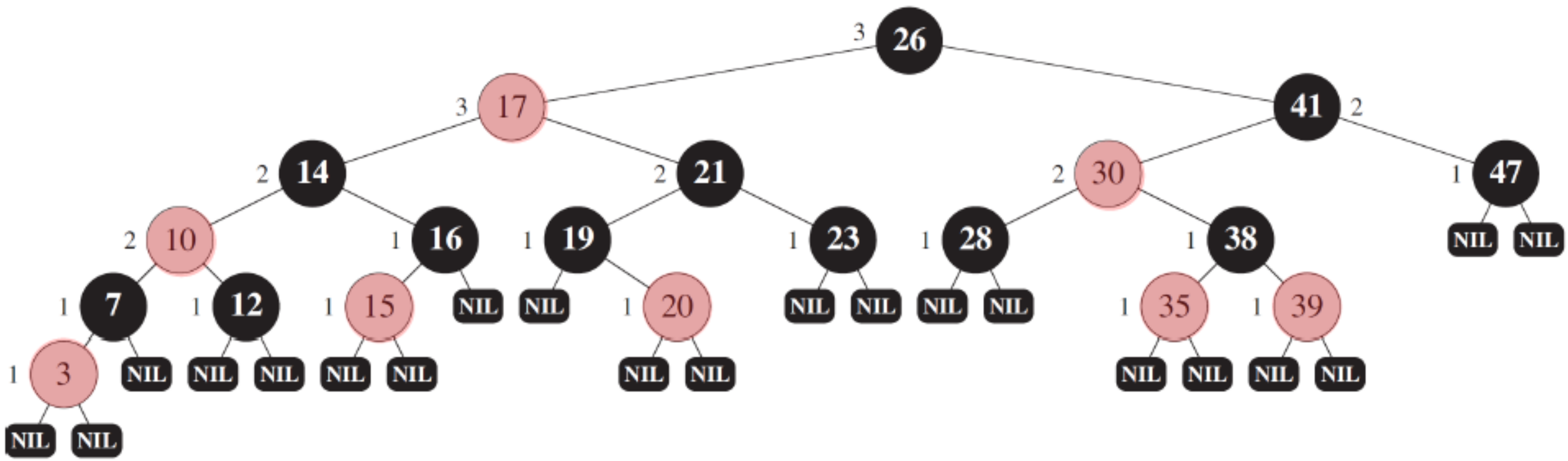
Property 3 (LeaB property)

- All leaves (NIL) are black



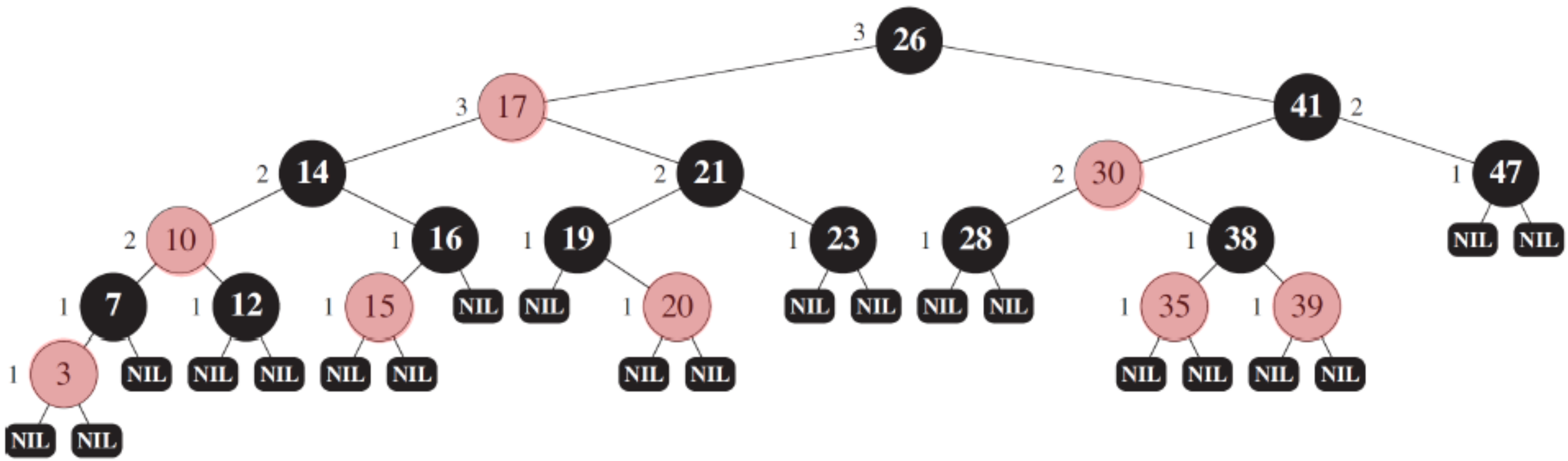
Property 4 (BredB property)

- If a node is red, then both children are black.



Property 5 (BH property)

- For each node all paths from the node to a leaf have the same number of black nodes.

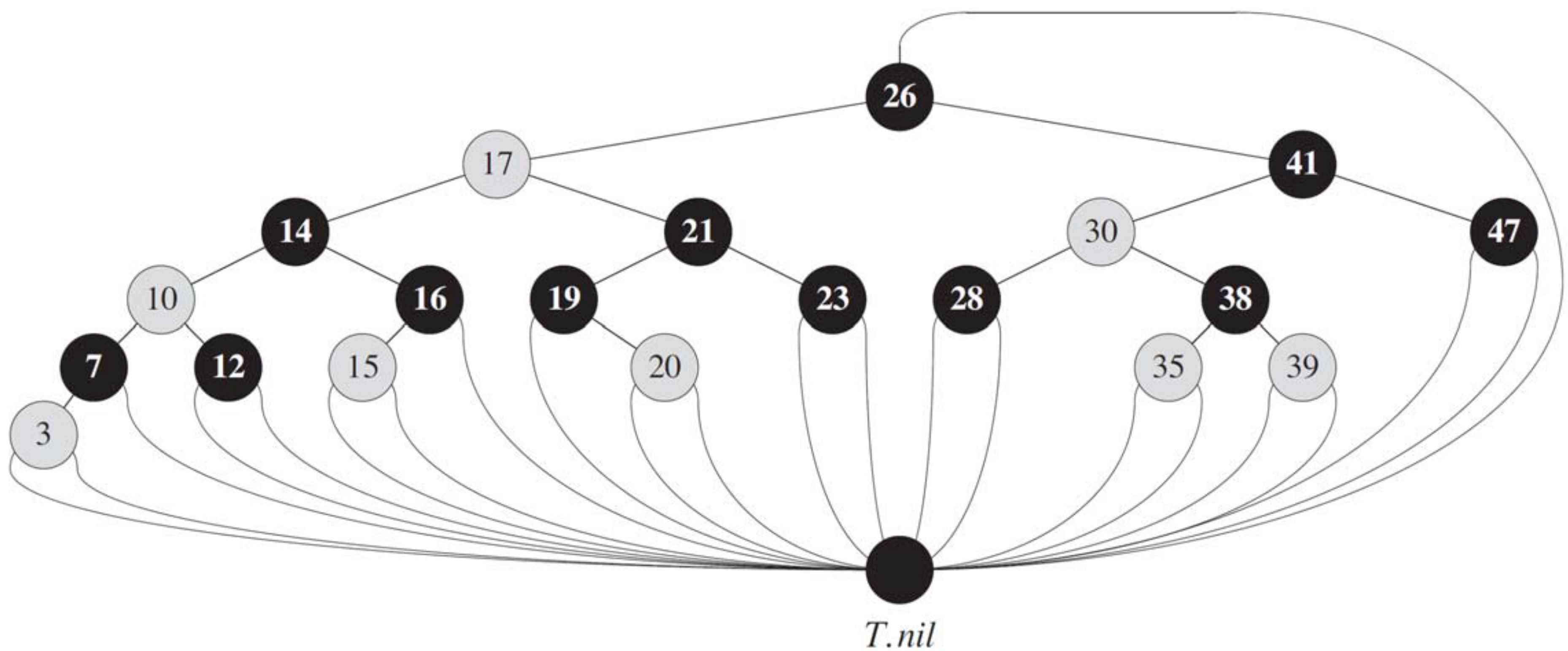


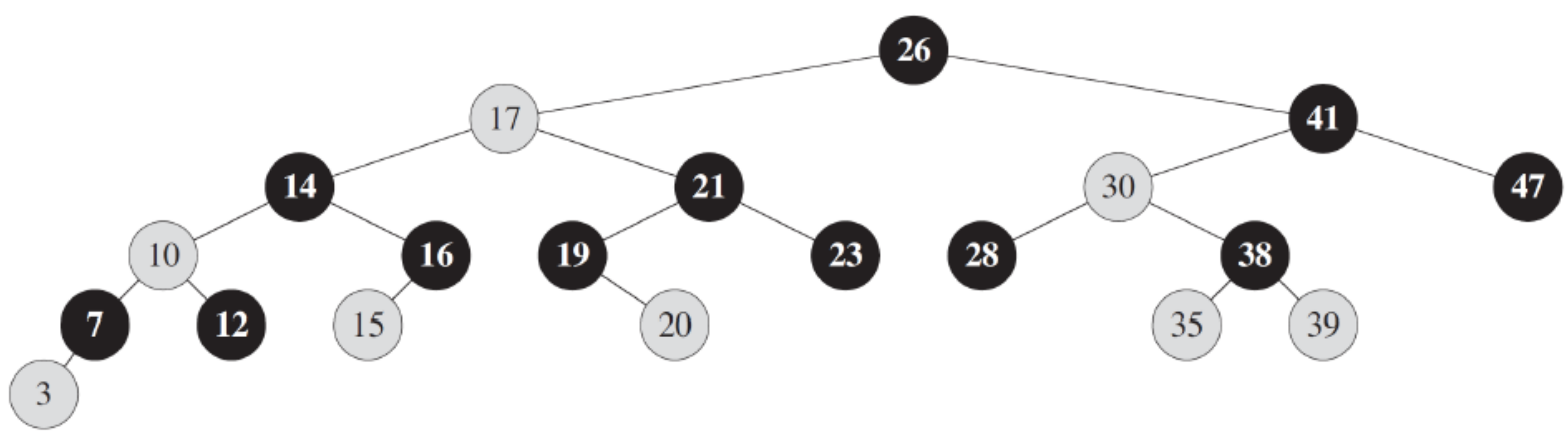
- For each node x, we can define a unique black height bh(x).

Properties

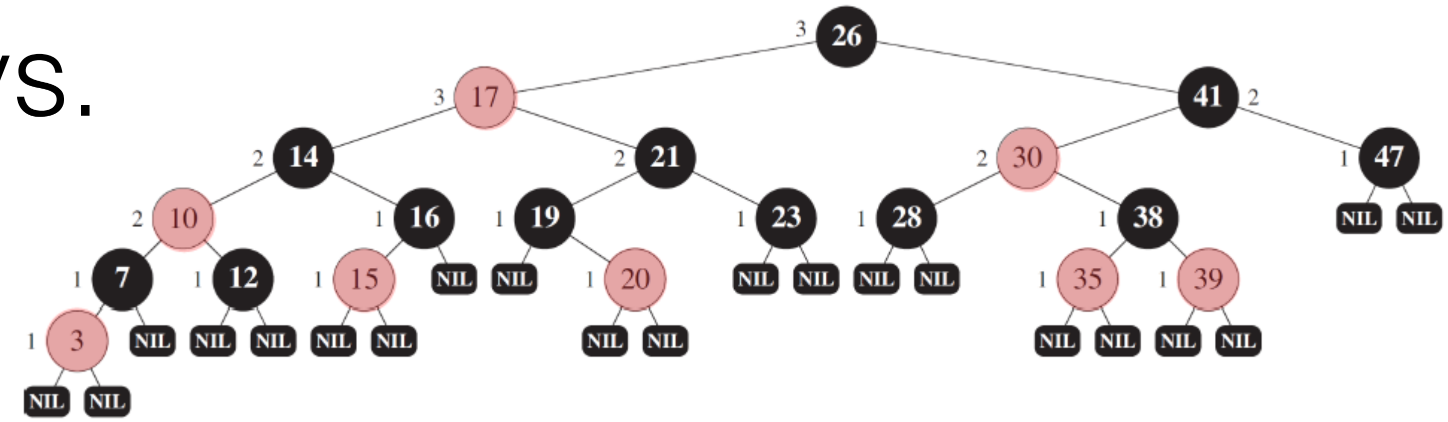
1. Every node is either red or black (**Duh**)
2. The root is black (**RooB**)
3. All leaves are black (**LeaB**)
4. If a node is red, then both children are black (**BredB**)
5. For each node all paths from the node to a leaf have the same number of black nodes (**BH**)

NIL Sentinel





Number of nodes vs. black-height



Lemma 1:

Let $n(x)$ be the number of non-leaf nodes of a red-black subtree rooted at x.

Then, $n(x) \geq 2^{bh(x)} - 1$.

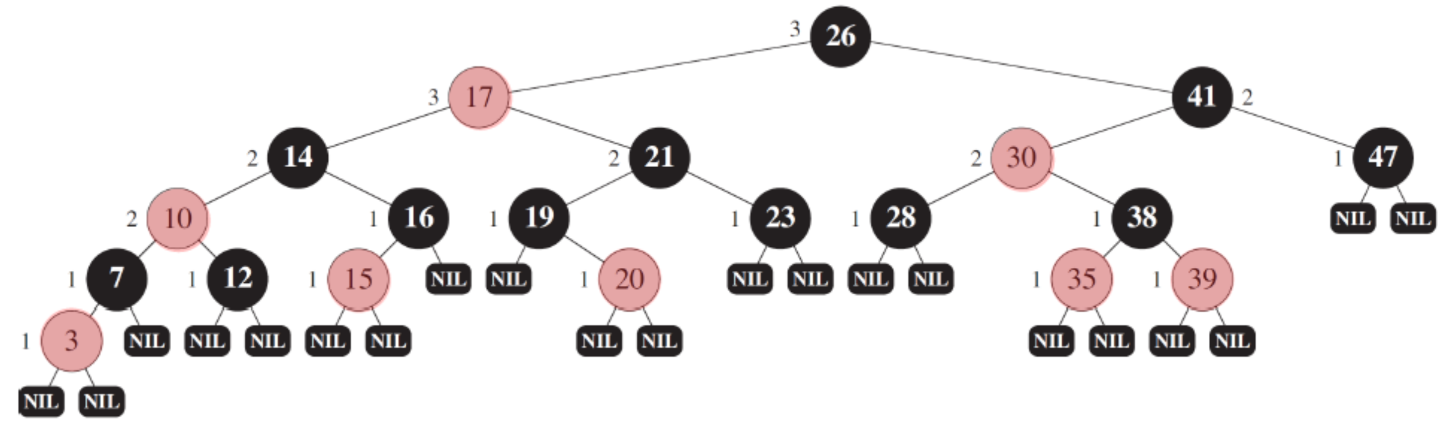
Proof (by induction on height $h(x)$ of node x):

- $h(x) = 0$: x is a leaf. $bh(x)=0$. $2^{bh(x)} - 1 = 0$. $n(x) \geq 0$. True.
- $h(x) > 0$: x is a non-leaf node. It has two children c_1 and c_2. If c_i is red, then $bh(c_i) = bh(x)$, else $bh(c_i) = bh(x)-1$.

  (use assumption) Since $h(c_i) < h(x)$, $n(c_i) \geq 2^{bh(c_i)} - 1 \geq 2^{bh(x)-1} - 1$.

  Thus, $n(x) = n(c_1)+n(c_2)+1 \geq 2(2^{bh(x)-1} - 1)+1 = 2^{bh(x)} - 1$.

Height vs. black-height



Lemma 2:

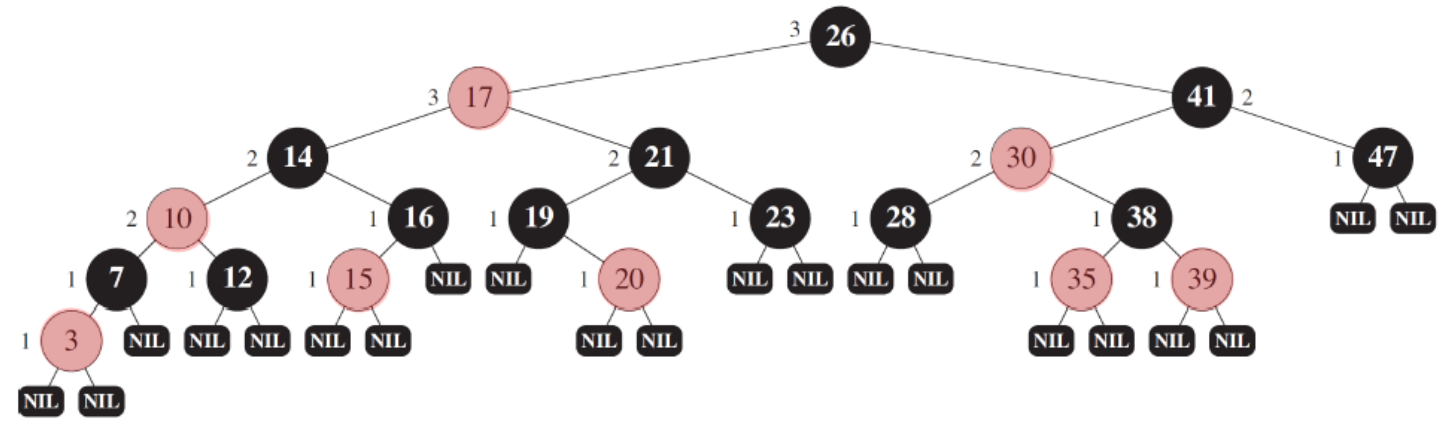
Let h be the height of a red-black tree with root r.

Then, $bh(r) \geq h/2$.

Proof:

- Let $r, v_1, v_2, \ldots, v_h$ be the longest path in the tree.
- The number of black nodes in the path is $bh(r)$.
- Thus, the number of red nodes is $h - bh(r)$.
- Since v_h is black (LeaB property) and every red node in the path must be followed by a black one (BredB property), we have $h - bh(r) \leq bh(r)$.
- Hence, $bh(r) \geq h/2$.

Height of a red-black tree



Theorem:

A red-black tree with n non-leaf nodes has height $h \leq 2 \lg(n+1)$.

Proof:

- Lemma 1: $n \geq 2^{bh(r)} - 1$ (r being the root).
- Lemma 2: $bh(r) \geq h/2$.
- Thus, $n \geq 2^{h/2} - 1$.
- So, $h \leq 2 \lg(n+1)$.

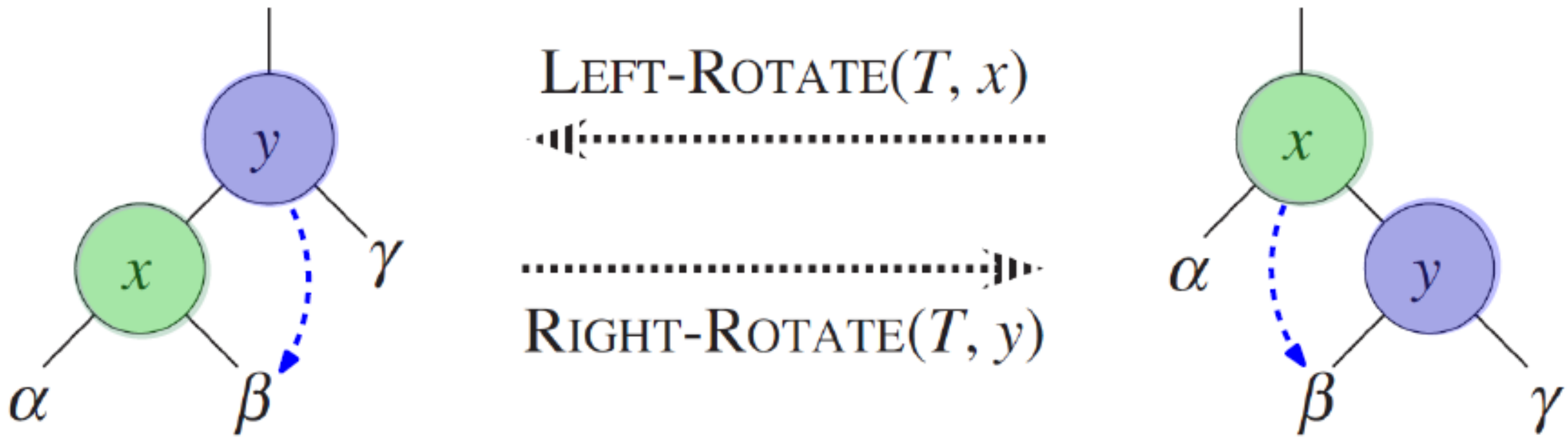
Corollary:

- The height of a red-black tree is $O(\lg n)$.
- All dynamic set operations can be performed in $O(\lg n)$, if we maintain the red-black tree properties.

Operations

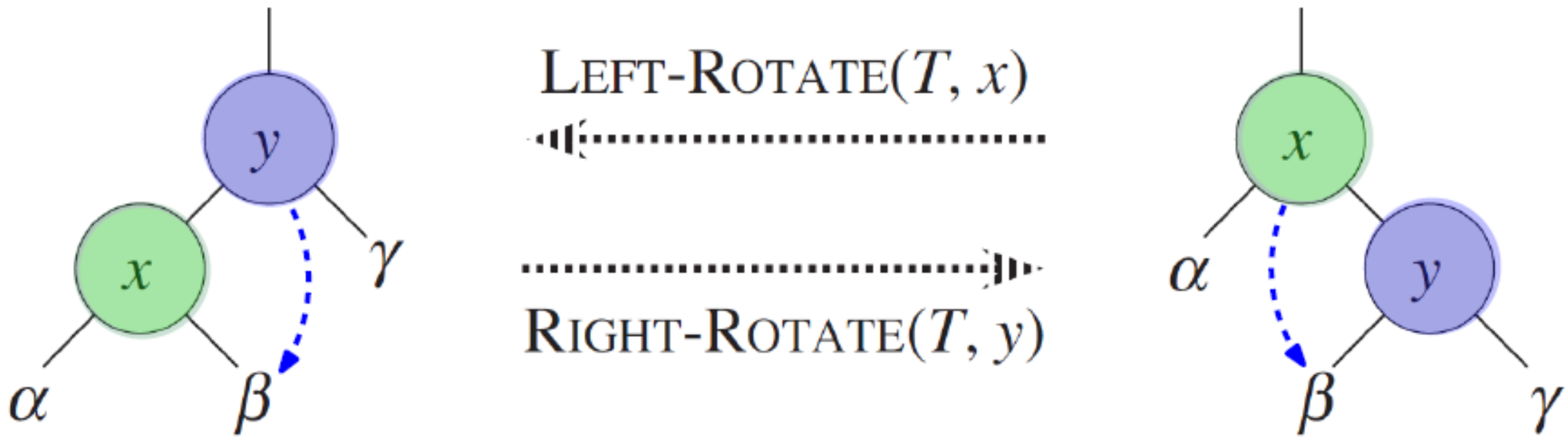
- Querying
  - SEARCH / MINIMUM & MAXIMUM / SUCCESSOR & PREDECESSOR
  - Just as in normal BST (see last lecture)
  - O(lg n)
- Modifying
  - TREE-INSERT / TREE-DELETE —> O(lg n)
  - BUT: Need to guarantee red-black tree properties
    - must change color of some nodes
    - change pointer structure through rotation

Rotations



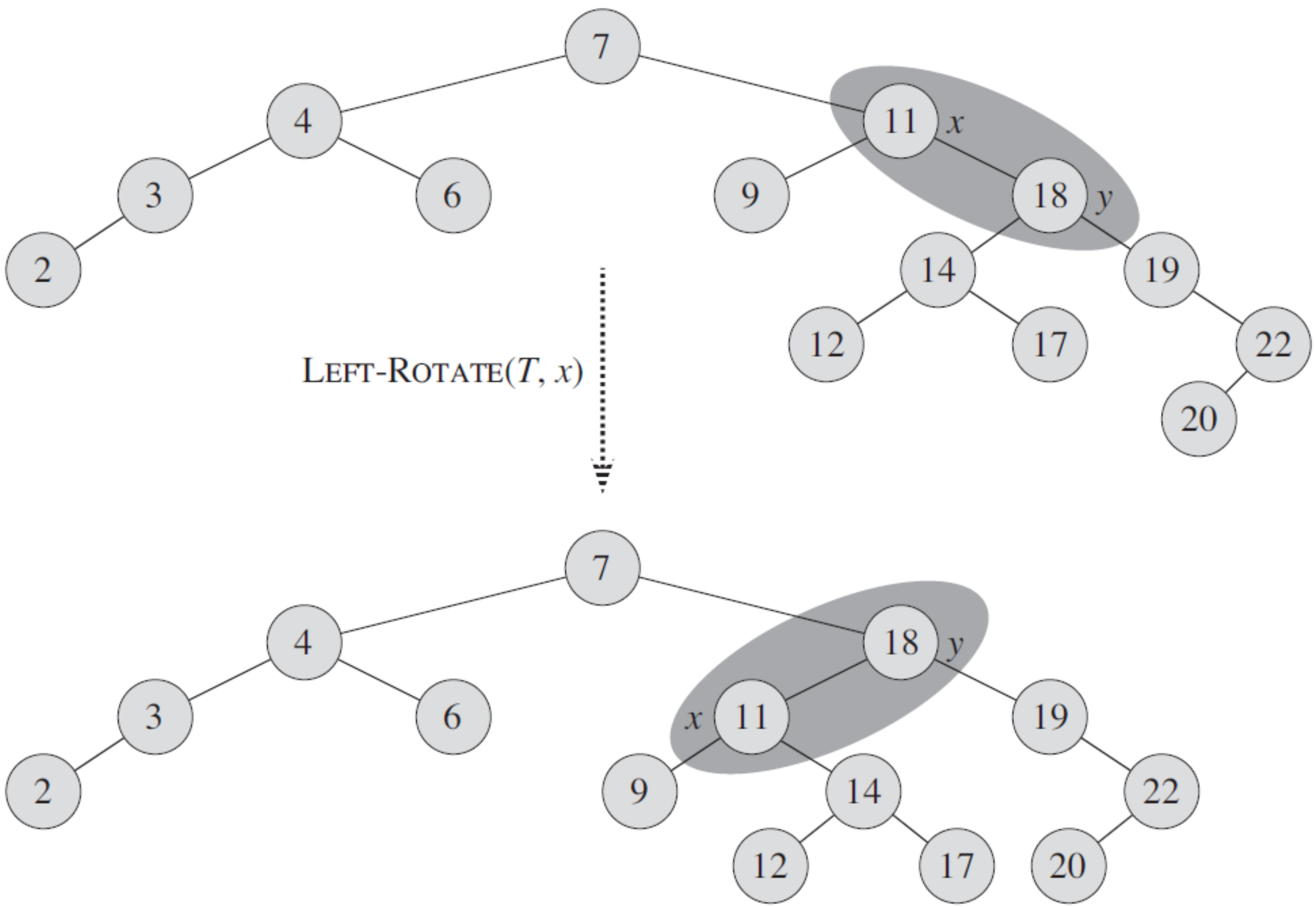
- Right-Rotate (T,y):
  - node y becomes right child of its left child x.
  - new left child of y is former right child of x.
- Left-Rotate (T,x):
  - node x becomes left child of its right child y.
  - new right child of x is former left child of y.

Rotations

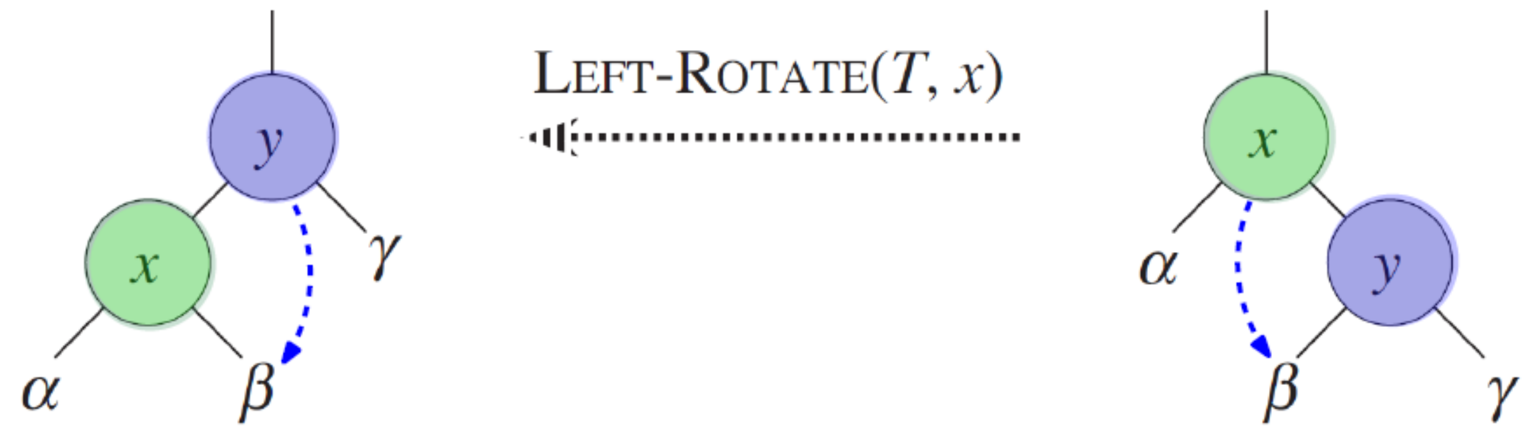


- BST property is preserved:
  - (left): key(α) ≤ x.key ≤ key(β) ≤ y.key ≤ key(γ)
  - (right): key(α) ≤ x.key ≤ key(β) ≤ y.key ≤ key(γ)

Rotation: Example



Rotation



LEFT-ROTATE(T, x)

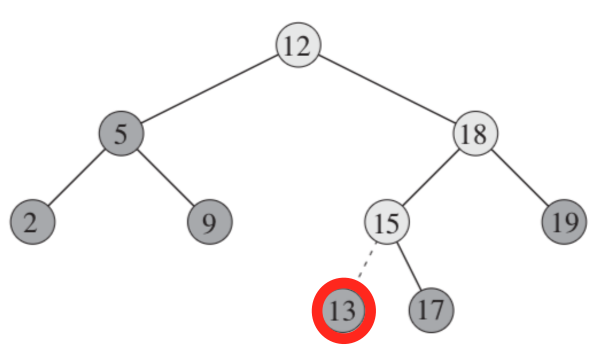
```
y = x.right                      // set y
x.right = y.left                 // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                        // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                       // put x on y's left
x.p = y
```

Running time: O(1).

Insertion

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
elseif z.key < y.key
    y.left = z
else y.right = z
```



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)
```

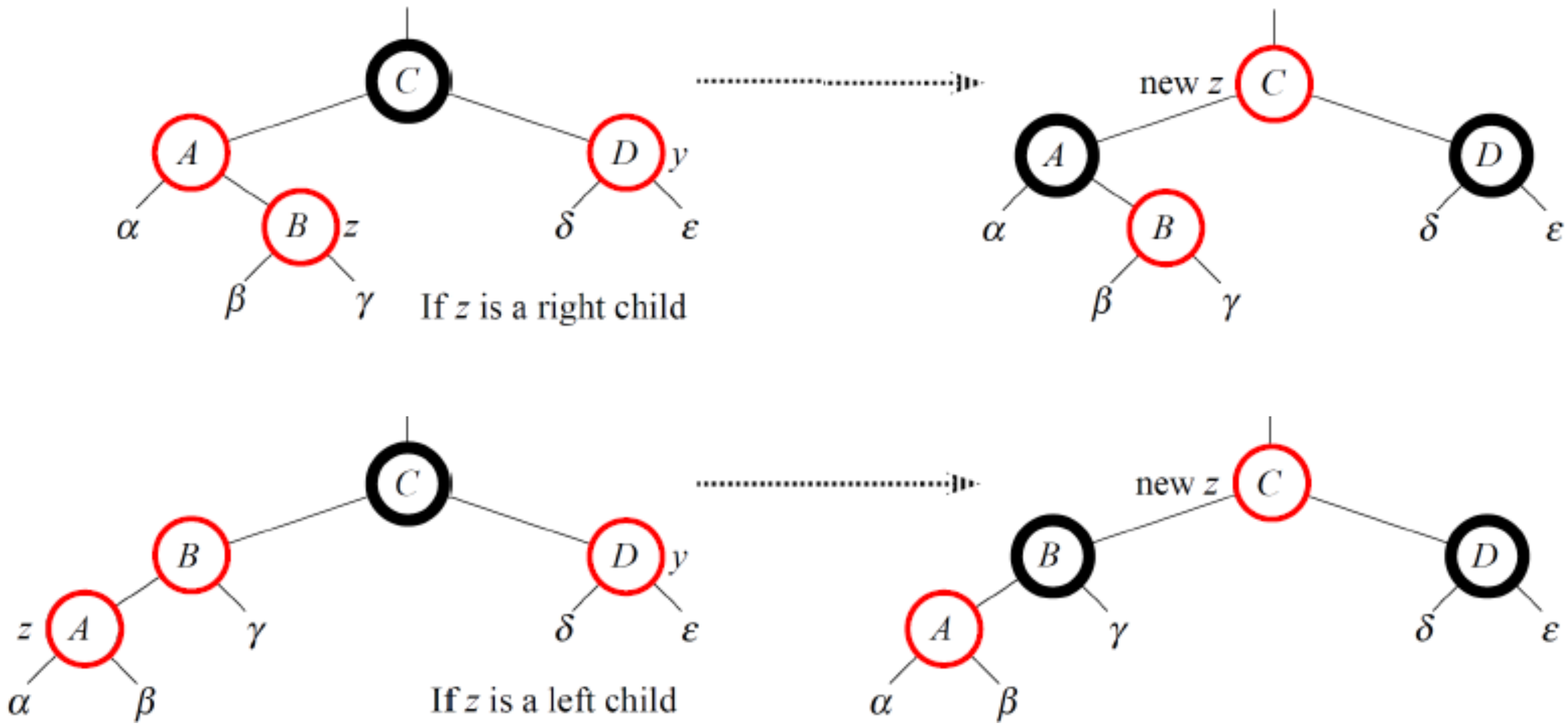
Fixing red-black tree properties

- We are inserting a **red** node to a valid red-black tree.
- Which properties may be violated?
  1. Duh: Cannot be violated. ✔
  2. RooB: Violated, if inserted node is root. ✘
  3. LeaB: Inserted node is not a leaf, i.e., no violation. ✔
  4. BredB: Violated, if parent of inserted node is red. ✘
  5. BH: Not affected by red nodes, i.e., no violation. ✔

Fixing BredB

- BredB for node z is violated, if z.p is red.
- Then, z.p.p is black. (BredB property!)
- We need to consider different cases depending on the **uncle y of z,** i.e., the child of z.p.p that is not z.p.
- There are 6 cases:
  - z.p is left child of z.p.p
    - y is red **(Case 1)**
    - y is black
      - z is right child of z.p **(Case 2)**
      - z is left child of z.p **(Case 3)**
  - z.p is right child of z.p.p
    - y is red **(symmetric to Case 1)**
    - y is black
      - z is right child of z.p **(symmetric to Case 2)**
      - z is left child of z.p **(symmetric to Case 3)**

Case 1 **(red uncle)**

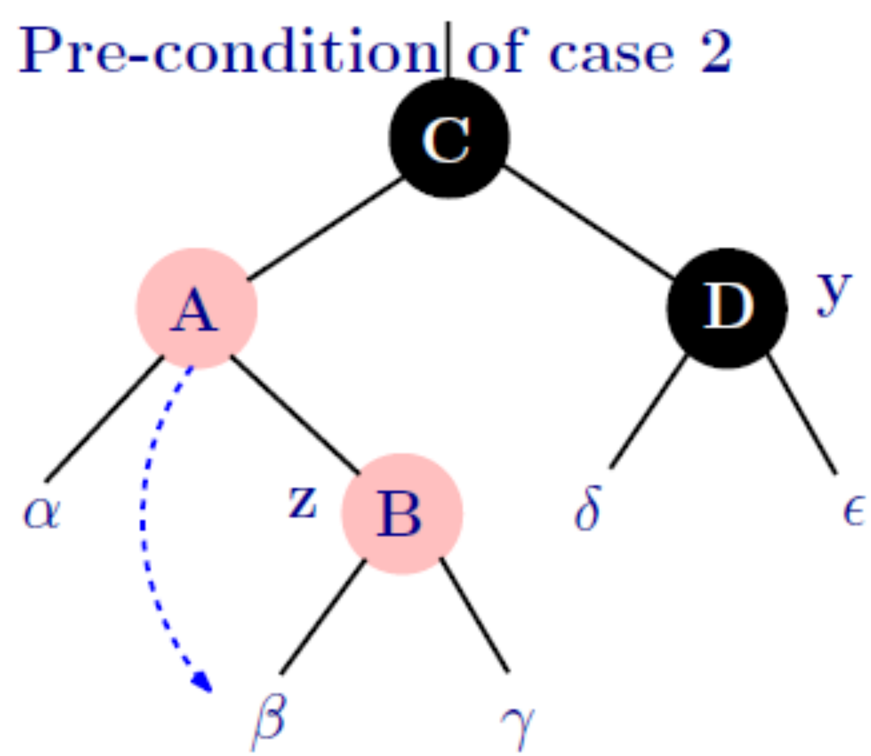


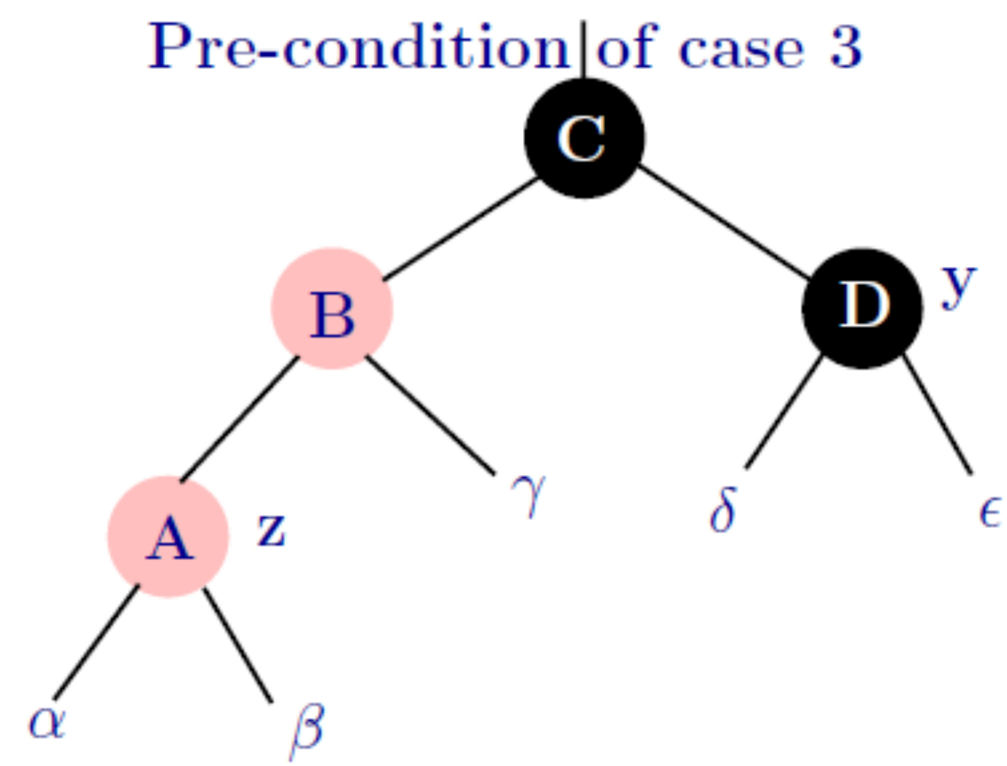

```
if z.p == z.p.p.left
    y = z.p.p.right
    if y.color == RED
        z.p.color = BLACK
        y.color = BLACK
        z.p.p.color = RED
        z = z.p.p
```

Case 1

There is a new problem, if z.p.p.p is red.
Algorithm needs to continue with z.p.p.

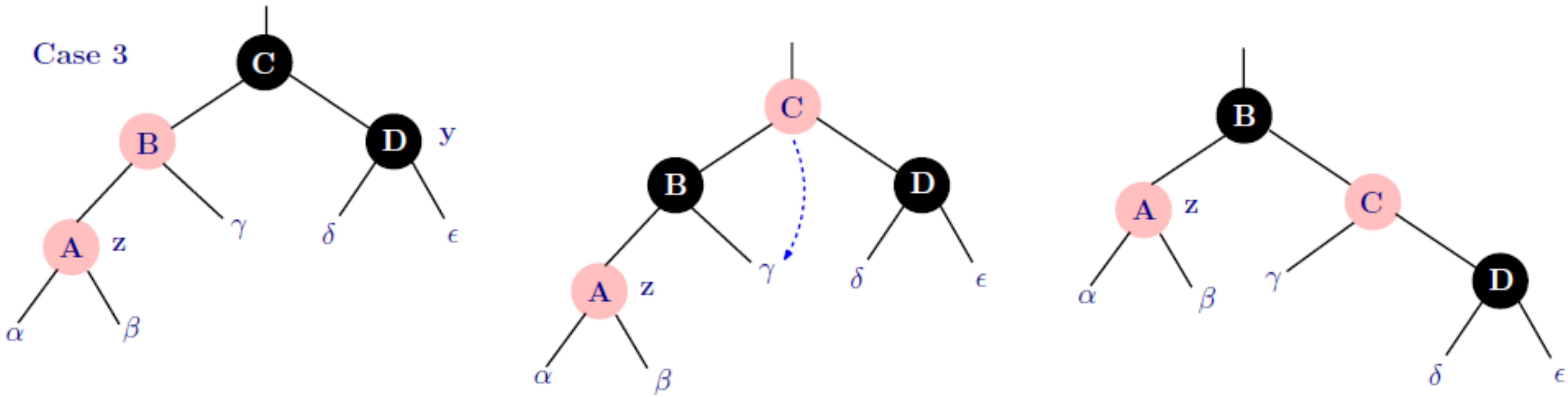
Case 2 **(black uncle, z right child)**






```
else if z == z.p.right
    z = z.p                  Case 2
    LEFT-ROTATE(T, z)
```

Case 3 **(black uncle, z left child)**



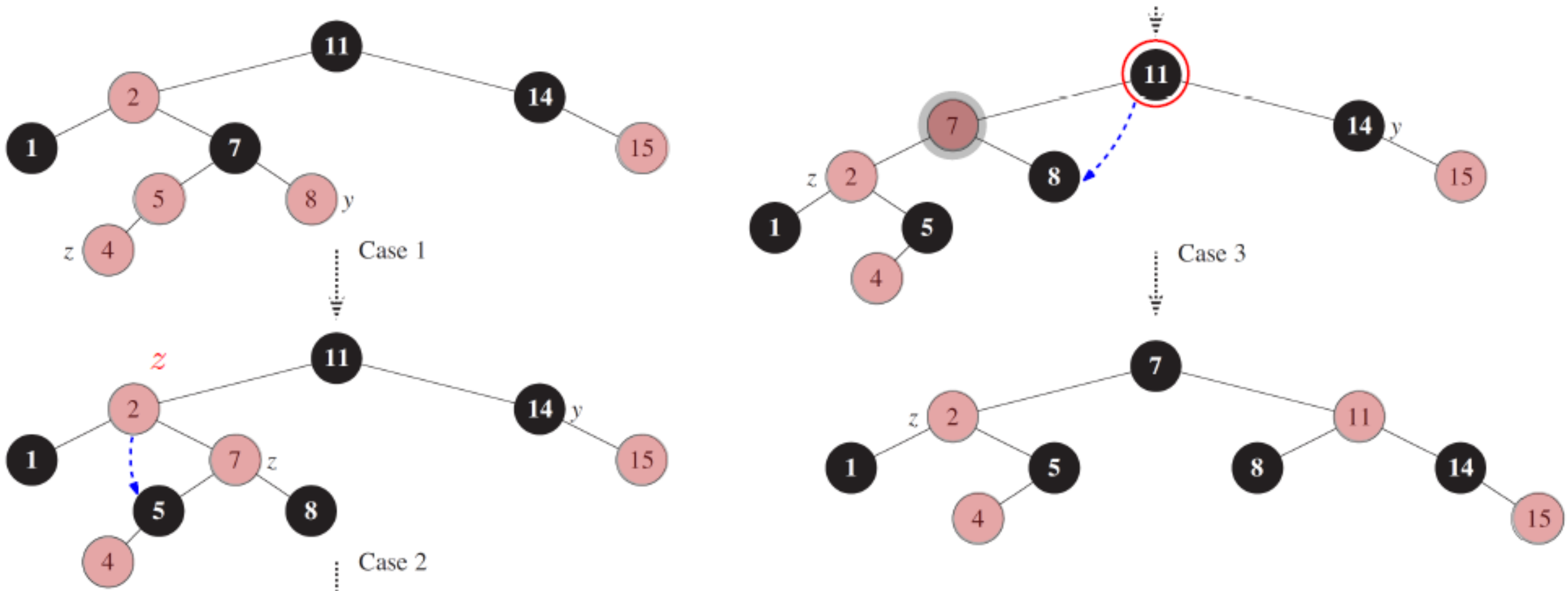
$z.p.color = $ BLACK $z.p.p.color = $ RED RIGHT-ROTATE$(T, z.p.p)$	Case 3

Putting it all together

- We need to put the 3 cases (and the 3 symmetric cases) together.
- Moreover, we need to propagate the considerations upwards (see Case 1).
- Finally, we have to fix RooB.

```
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
```

Example

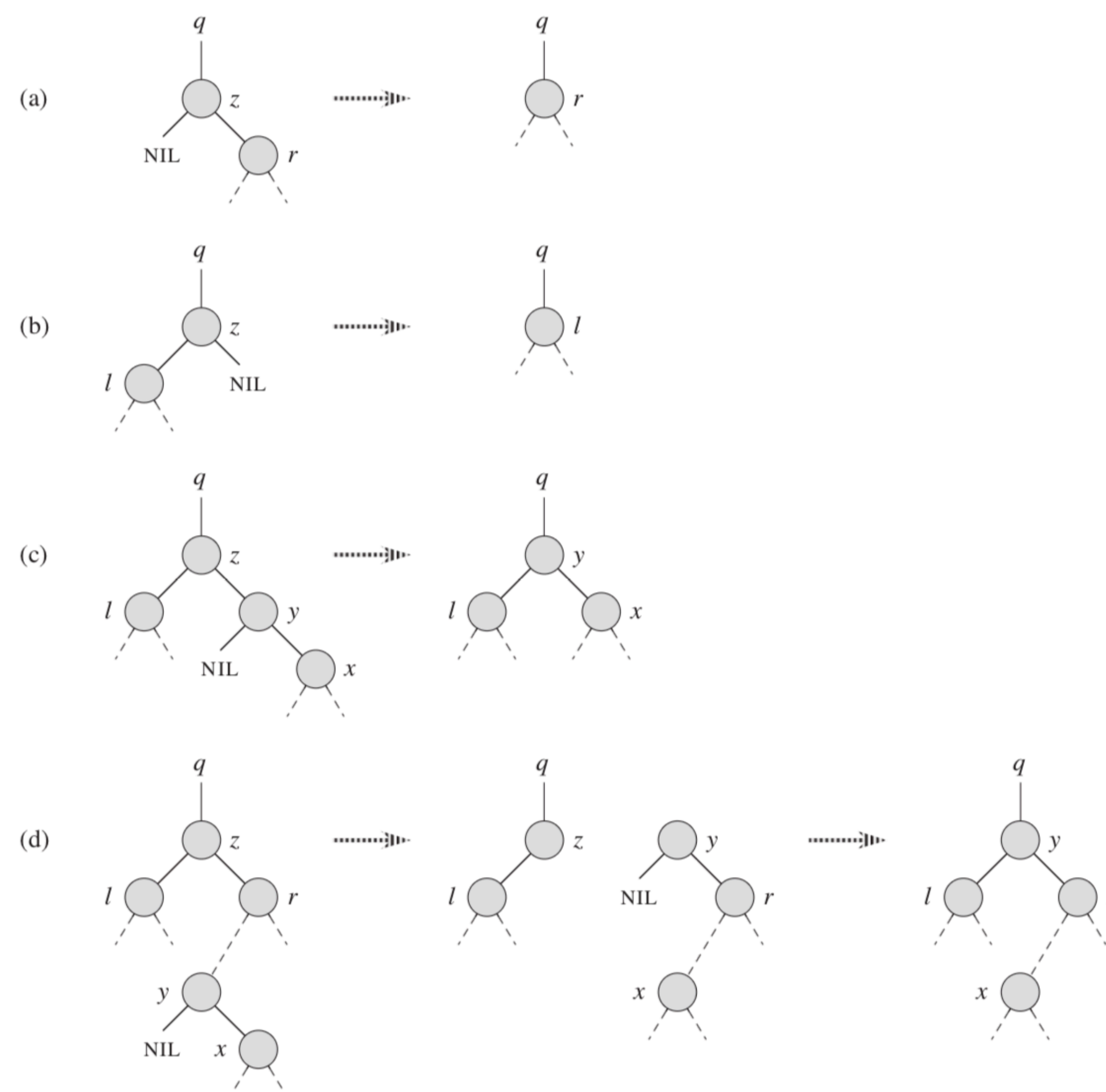


Running time

- In worst case, we have to go all the way from the leaf to the root along the longest path within the tree.
- Hence, running time is $O(h) = O(\lg n)$ for the fixing of the red-black tree properties.
- Overall, running time for insertion is $O(h) = O(\lg n)$.
- Example for building up a red-black tree by iterated node insertion: http://www.youtube.com/watch?v=vDHFF4wjWYU

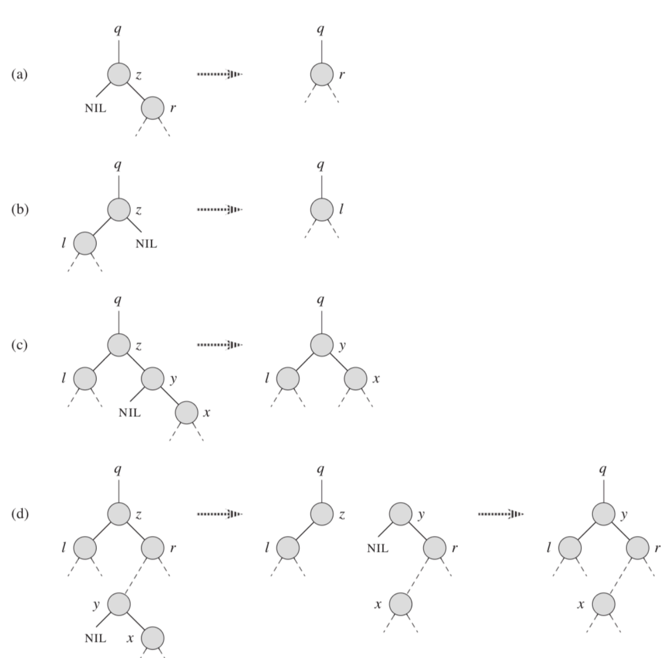
Deletion (recall BST)

```
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 (RB)

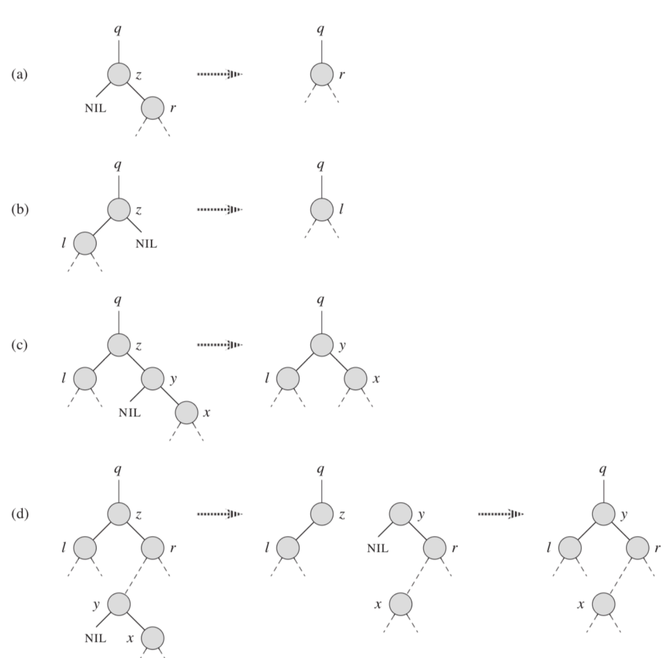
```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p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

- node y
  - either removed (a/b)
  - or moved in the tree (c/d)
  - y-original-color
- node x
  - the node that moves into y's original position
  - x.p points to y's original parent (since it moves into y's position, note special case in 12/13)



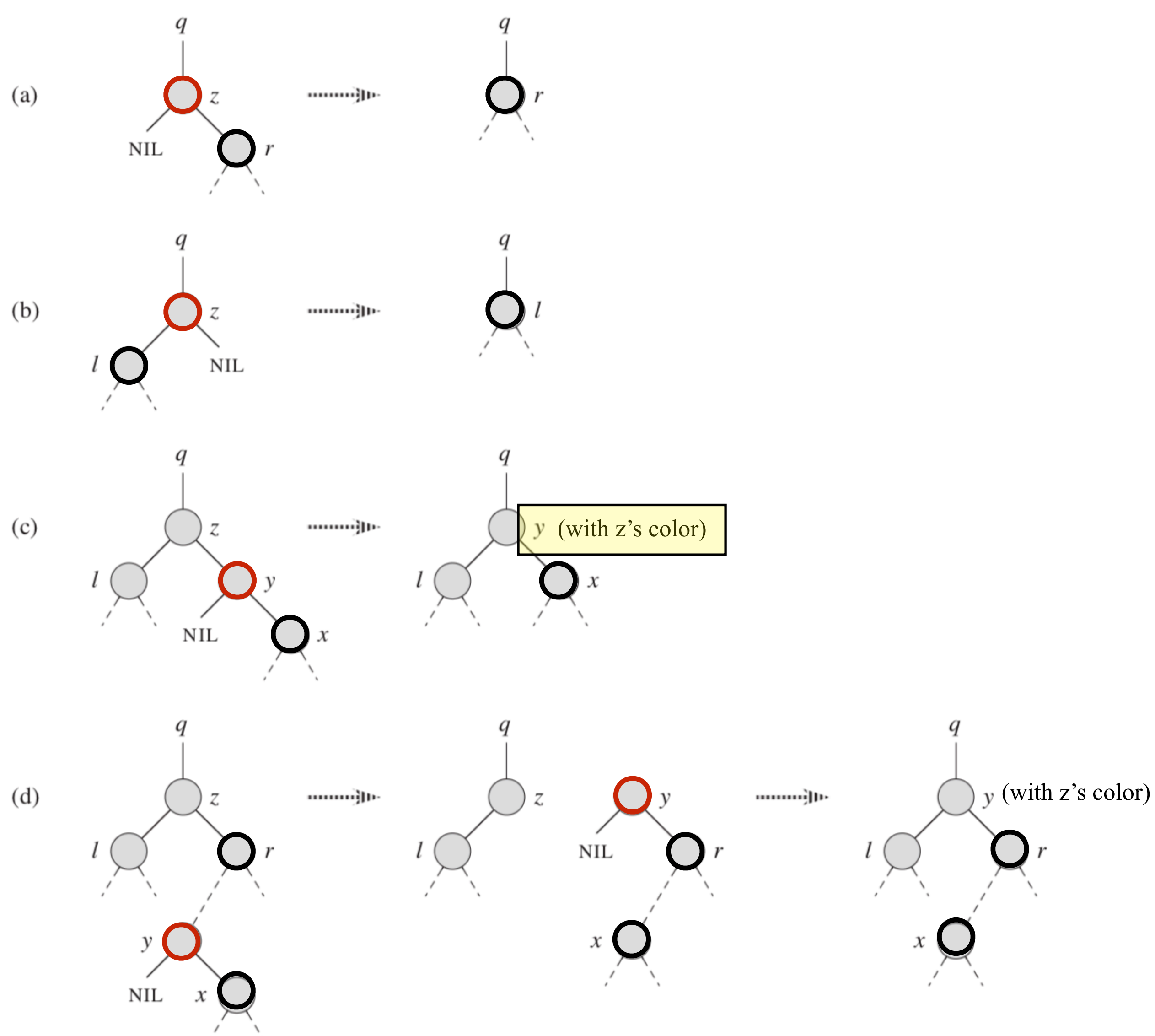
RB-DELETE(T, z)

```
1  y = z
2  y-original-color = y.color
3  if z.left == T.nil
4      x = z.right
5      RB-TRANSPLANT(T, z, z.right)
6  elseif z.right == T.nil
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
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 == BLACK
22     RB-DELETE-FIXUP(T, x)
```

1. Every node is either red or black (Duh)
2. The root is black (RooB)
3. All leaves are black (LeaB)
4. If a node is red, then both children are black (BredB)
5. For each node all paths from the node to a leaf have the same number of black nodes (BH)

Deletion

- y-original-color == red



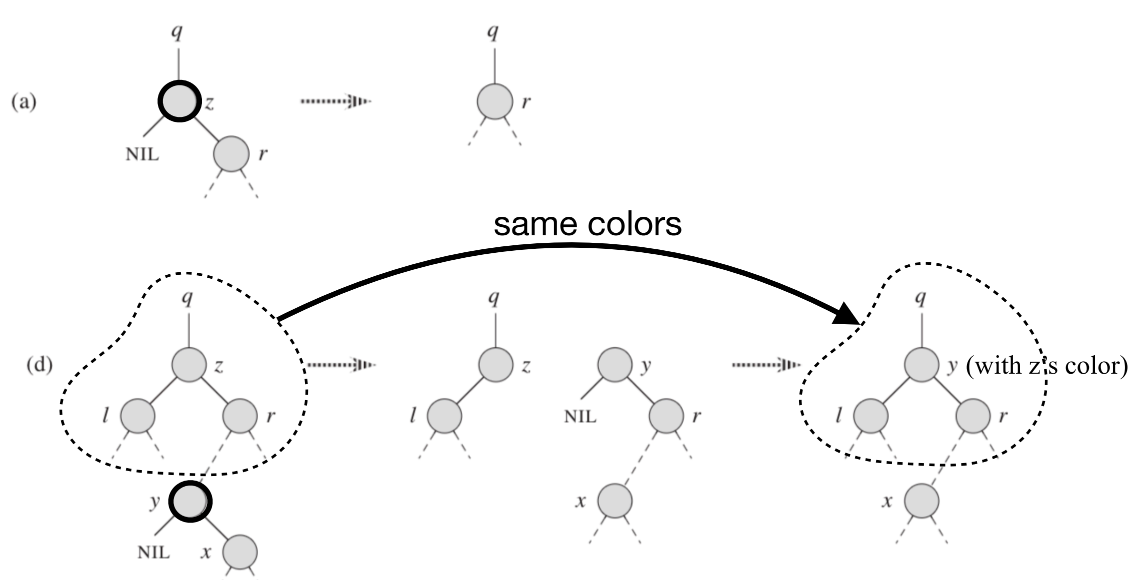
RB-DELETE(T, z)

```
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)
```

1. Every node is either red or black (Duh)
2. The root is black (RooB)
3. All leaves are black (LeaB)
4. If a node is red, then both children are black (BredB)
5. For each node all paths from the node to a leaf have the same number of black nodes (BH)

Deletion

- y-original-color == red
  - no problem
- y-original-color == black
  - violations might occur (2,4,5)
  - main idea to fix
    - x gets an “**extra black**” & needs to get rid of it
  - 4 cases



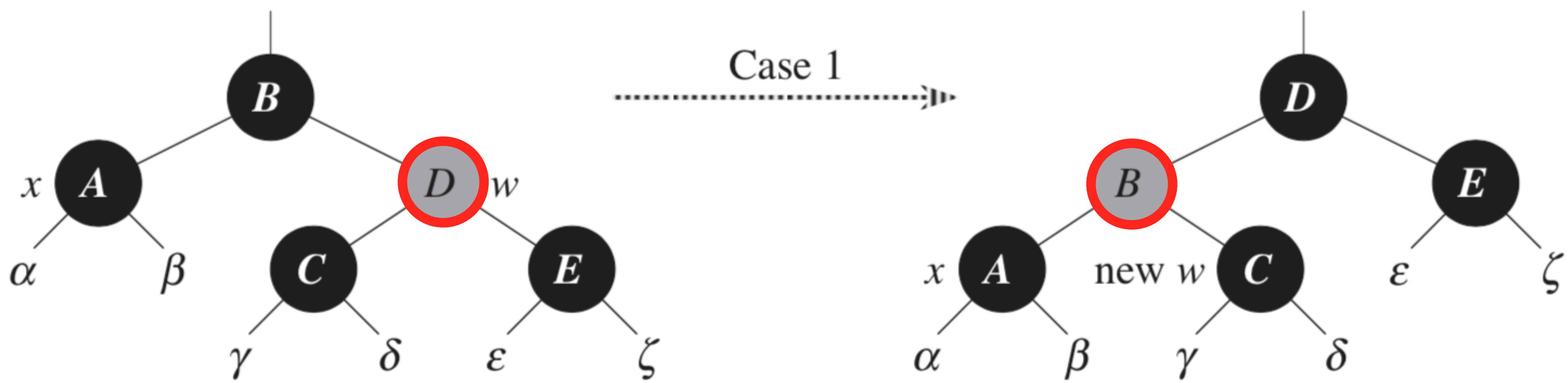
RB-DELETE(T, z)

```
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)
```

Fixing red-black tree properties

- **Case 1:** x's sibling w is red.

  Transform to Case 2, 3, or 4 by left rotation and changing colors of nodes B and D.



x = node with extra black

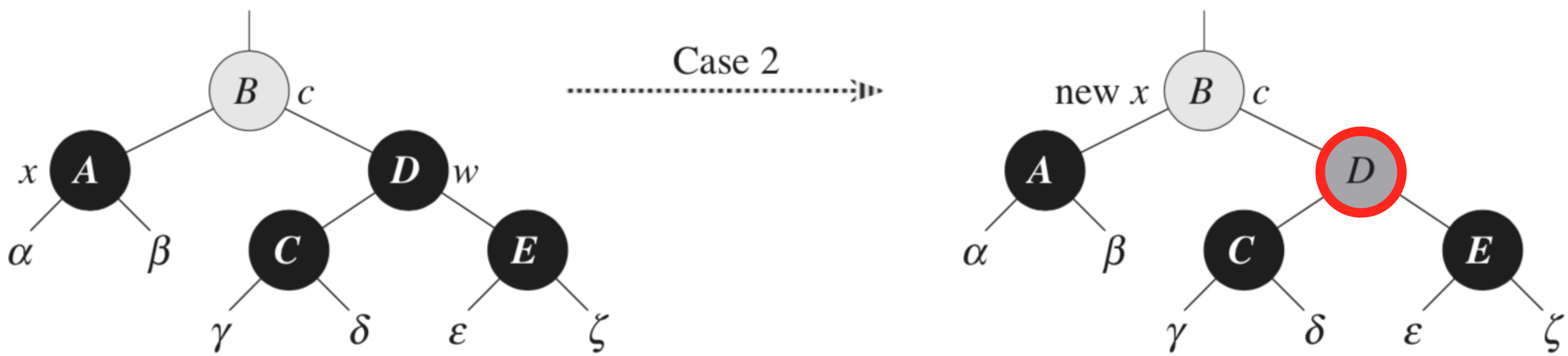
w = x's sibling

if $w.color$ == RED
    $w.color = $ BLACK
    $x.p.color = $ RED
    LEFT-ROTATE$(T, x.p)$
    $w = x.p.right$

Fixing red-black tree properties

- **Case 2:** x‘s sibling w is black and the children of w are black.

  Set color of w to red and propagate upwards.



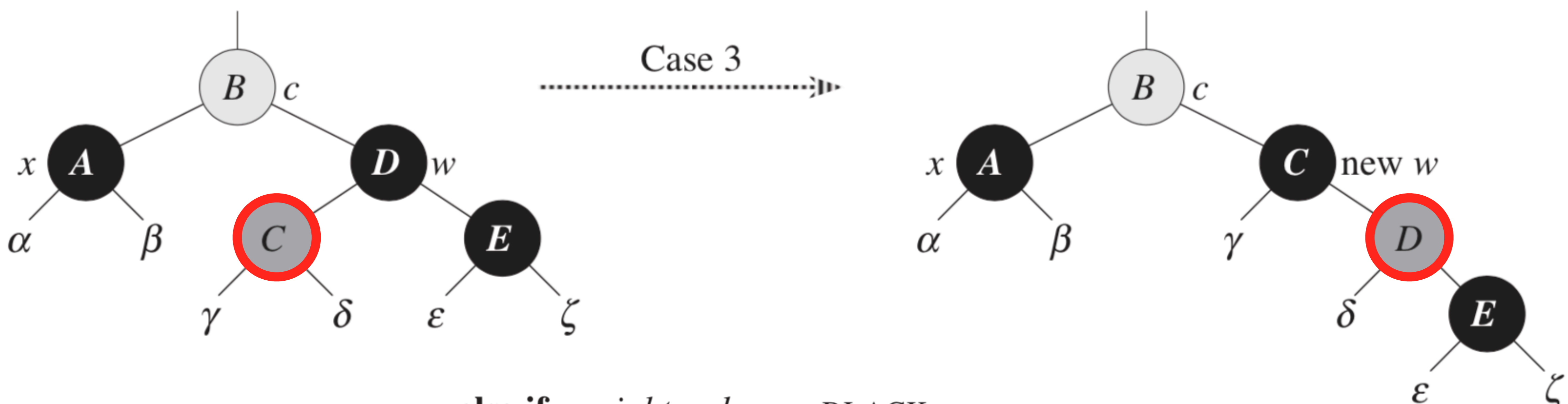
x = node with extra black
w = x’s sibling
c = color of the node

if $w.left.color$ == BLACK and $w.right.color$ == BLACK
    $w.color = $ RED
    $x = x.p$

Fixing red-black tree properties

- **Case 3:** x's sibling w is black and the left child of w is red, while the right child of w is black.

  Transform to Case 4 by right rotation and changing colors of nodes C and D.

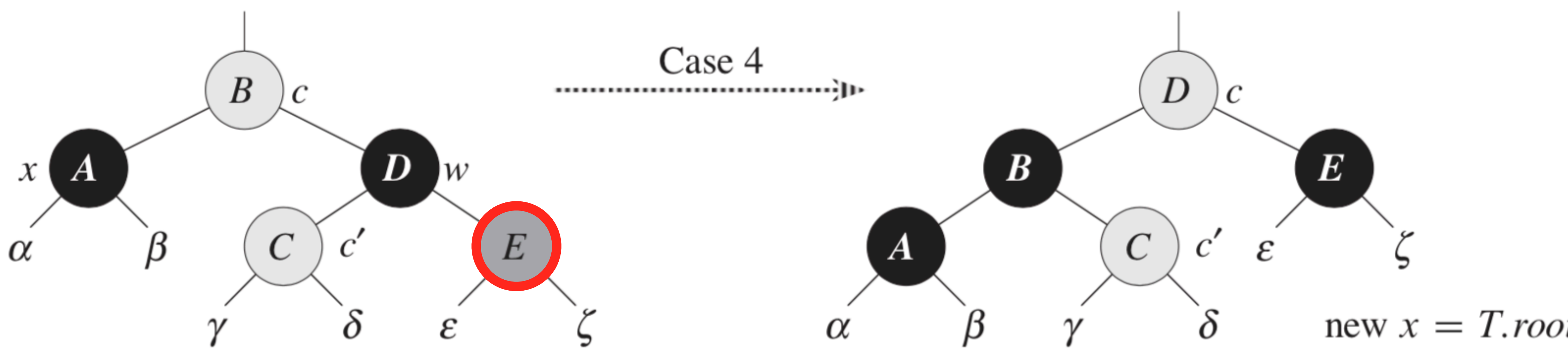



```
else if w.right.color == BLACK
        w.left.color = BLACK
        w.color = RED
        RIGHT-ROTATE(T, w)
        w = x.p.right
```

Fixing red-black tree properties

- **Case 4:** x's sibling w is black and the right child of w is red.

  Perform a left-rotate and change colors of B, D, and E. Then, the loop terminates.



$w.color = x.p.color$
$x.p.color = \text{BLACK}$
$w.right.color = \text{BLACK}$
$\text{LEFT-ROTATE}(T, x.p)$

Fixing red-black tree properties

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
```

The running time is O(h) = O(lg n).

Conclusion

- Modifying operations on red-black trees can be executed in O(lg n) time.