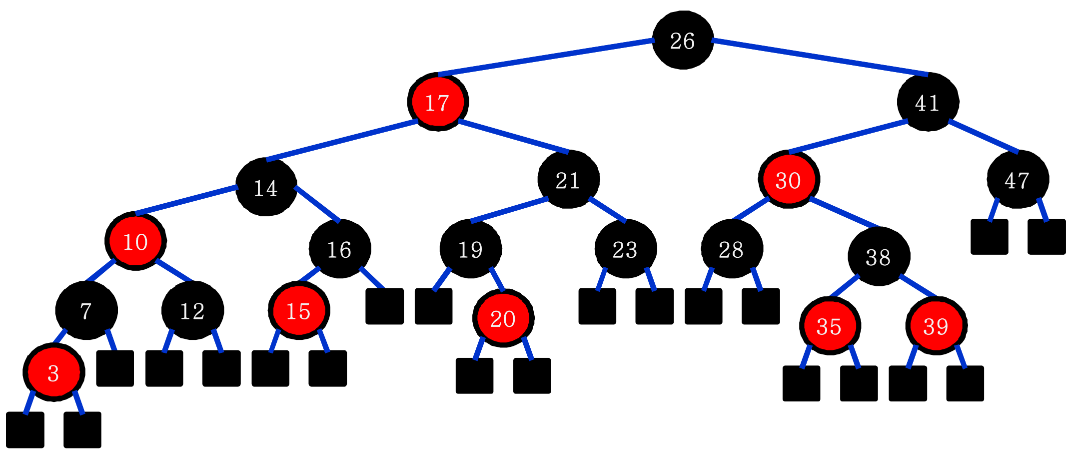
Red-Black Trees



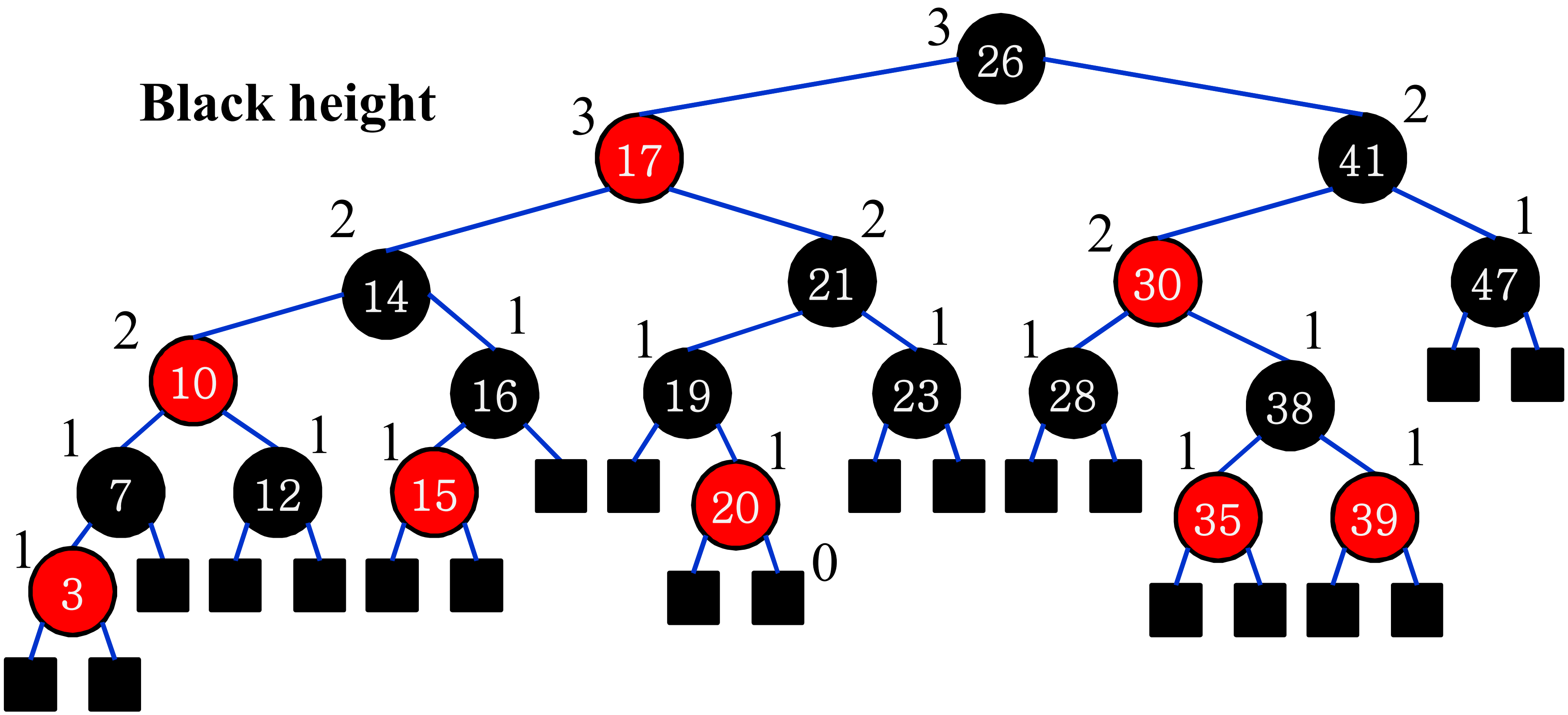
Dr. Md. Abul Kashem Mia, Professor, CSE Dept, BUET

Red-Black Trees

- A *red-black tree* is a binary search tree with one extra attribute for each node: the *color*, which is either red or black.

- A red-black tree is a binary search tree which has the following *red-black properties*:
  - Color Property: Every node is either red or black
  - Root Property: The root is black
  - External Property: Every external node is black
  - Internal Property: Both children of a red node are black
  - Depth Property: For each node, all simple paths from the node to descendant leaves contain the same number of black nodes

Red-Black Trees



Black height: It is the number of black nodes on any simple path from a node *x* (not including it) to a leaf. Black height of any node *x* is bh(*x*).

The number of black nodes from a node to any leaf is the same.

Red-Black Trees

- **Proposition**: The height of a red-black tree storing n items is $O(\log n)$.

Justification:

We first show that, the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

If bh(x) = 0 then x is a leaf, so the subtree rooted at x contains

$$2^0 - 1 = 0 \text{ internal nodes.}$$

Let a node x has a positive height and is an internal node which has two children.

Each child of x has a black height of either bh(x) or bh(x)-1.

So we can use the recursive hypothesis to conclude that the subtree rooted at x contains at least $(2^{bh(x)-1}-1) + (2^{bh(x)-1}-1) + 1$

$$= (2 \cdot 2^{bh(x)-1}-1) = (2^{bh(x)}-1) \text{ internal nodes}$$

Red-Black Trees

Let h be the height of the tree.

According to the internal property, the black-height of the root must be at least $h/2$; thus

$$n \geq 2^{h/2}-1.$$

$$\log(n+1) \geq h/2$$

$$h \leq 2\log(n+1).$$

Thus $h = O(\log n)$.

The main Idea:

Since red nodes cannot have red children, in the worst case, the number of nodes on a path must alternate red/black.

Thus, that path can be only twice as long as the black depth of the tree.

Therefore, the worst case height of the tree is $O(2\log n_b)$.

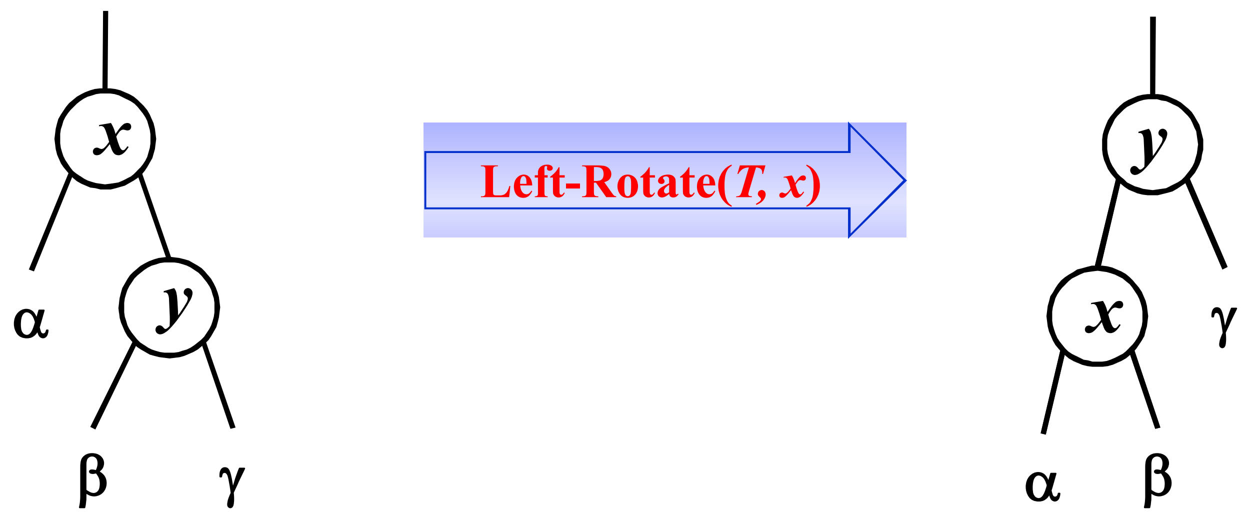
Therefore, the height of a red-black tree is $O(\log n)$.

Red-Black Trees

- **Rotations:** Rotations maintain the inorder ordering of keys:

  $\alpha \le x \le \beta \le y \le \gamma$.

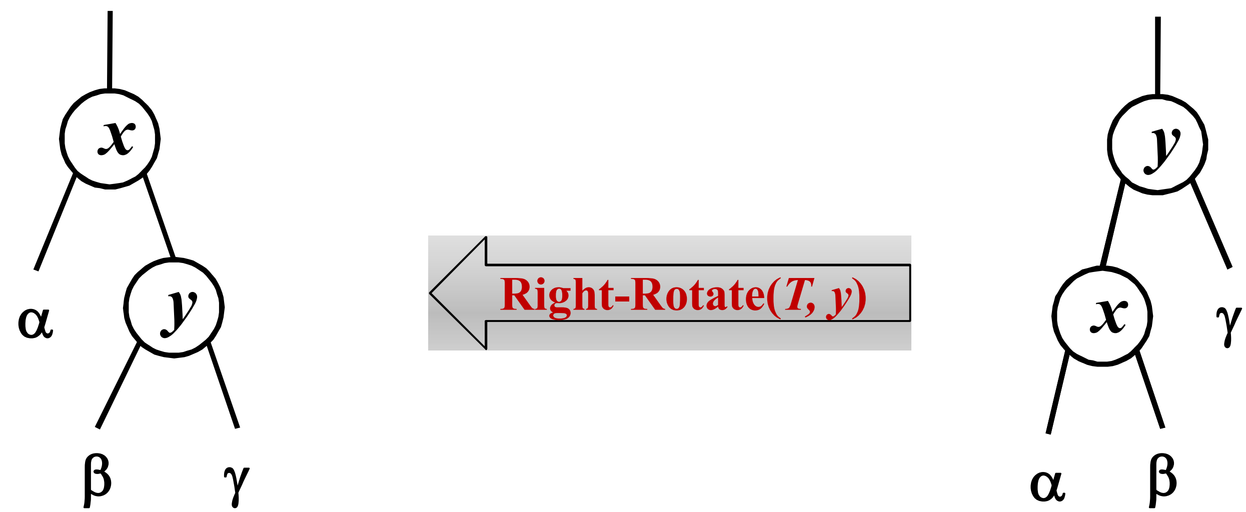
A rotation can be performed in $O(1)$ time, since a constant number of pointers need to be modified.



Red-Black Trees

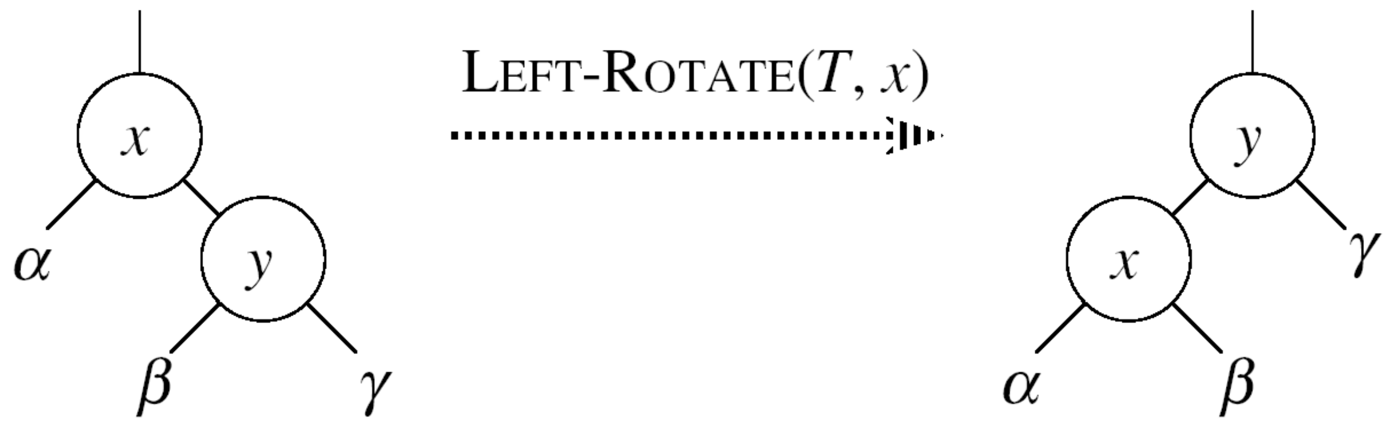
- **Rotations:** Rotations maintain the inorder ordering of keys:
  $\alpha \le x \le \beta \le y \le \gamma$.

A rotation can be performed in $O(1)$ time, since a constant number of pointers need to be modified.



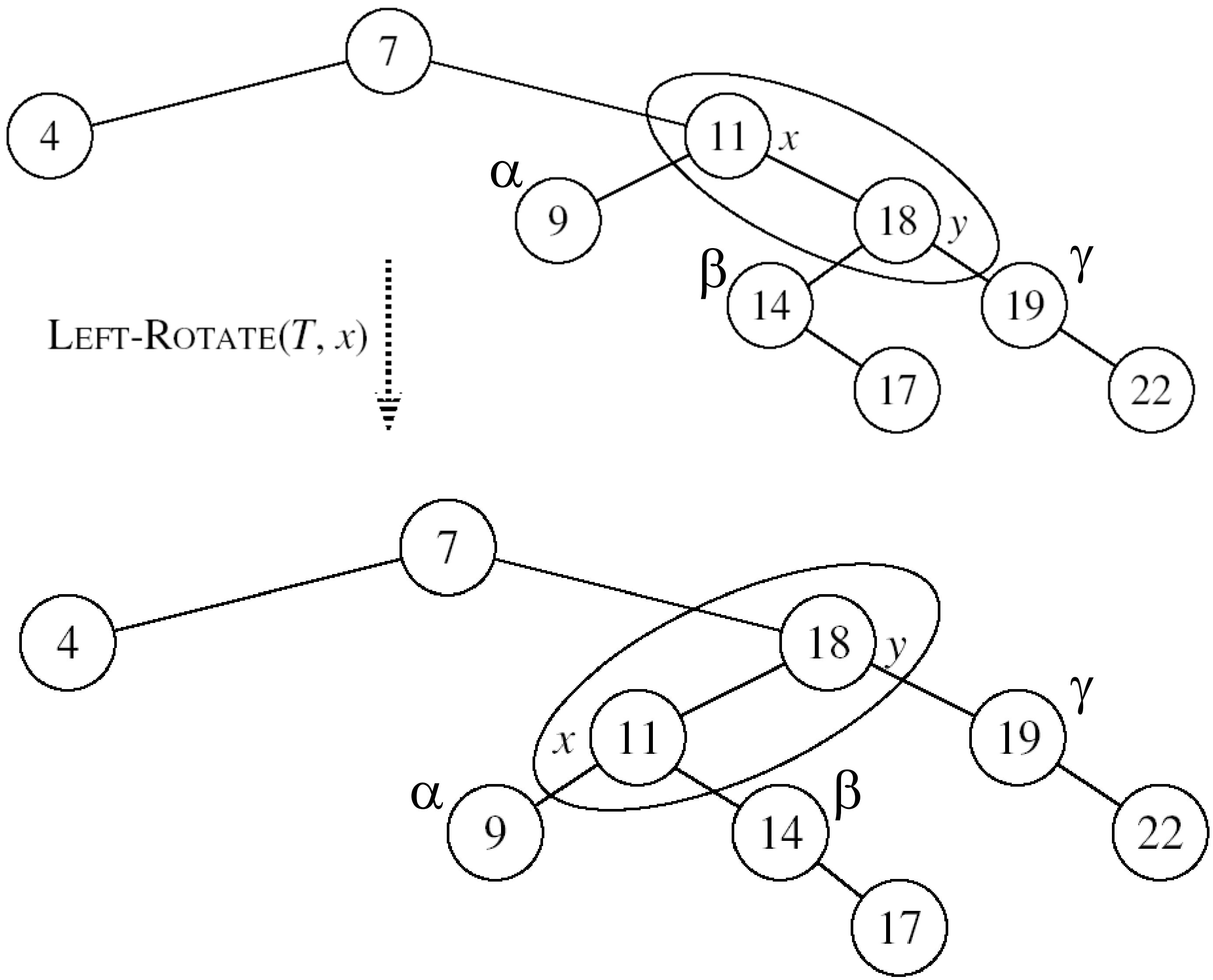
Left Rotations

- Assumptions for a left rotation on a node *x*:
  - The right child of *x* (that is, *y*) is not NIL



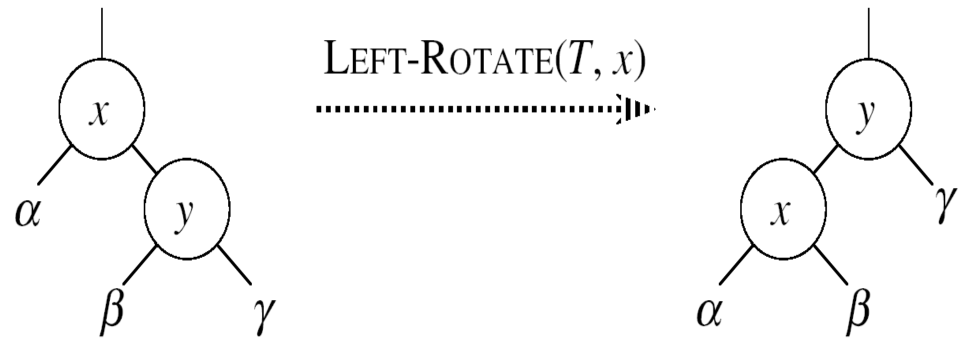
- Idea:
  - Pivots around the link from *x* to *y*
  - Makes *y* the new root of the subtree
  - *x* becomes *y*'s left child
  - *y*'s left child becomes *x*'s right child

Example: Left-Rotate



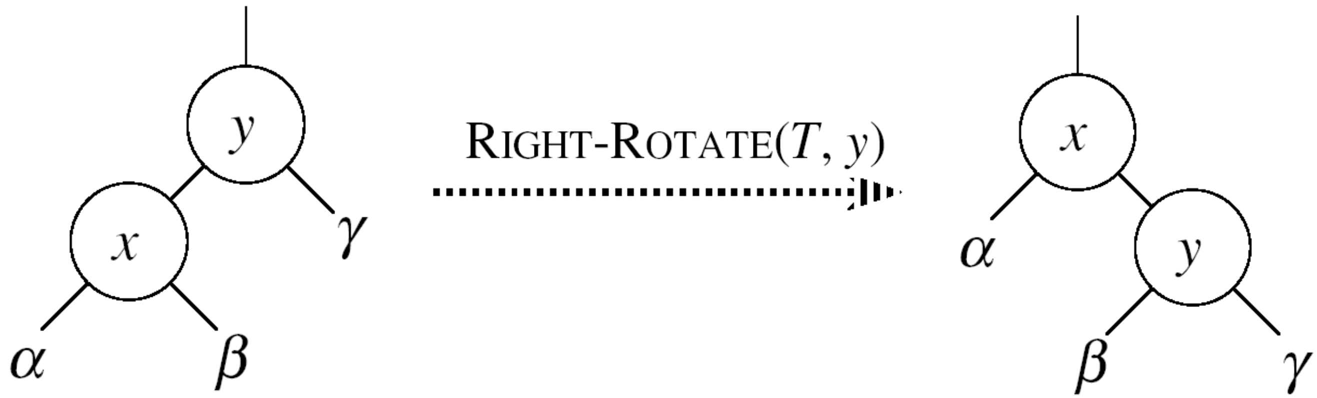
Left-Rotate(*T, x*)

```
y ← right[x]        \\ set y
right[x] ← left[y]
p[left[y]] ← x
p[y] ← p[x]
if p[x] = nil then          \\ x is the root
    root[T] ← y
else if x = left[p[x]] then  \\ check whether x is the left child of p[x]
    left[p[x]] ← y
else
    right [p[x]] ← y
left[y] ← x
p[x] ← y
```



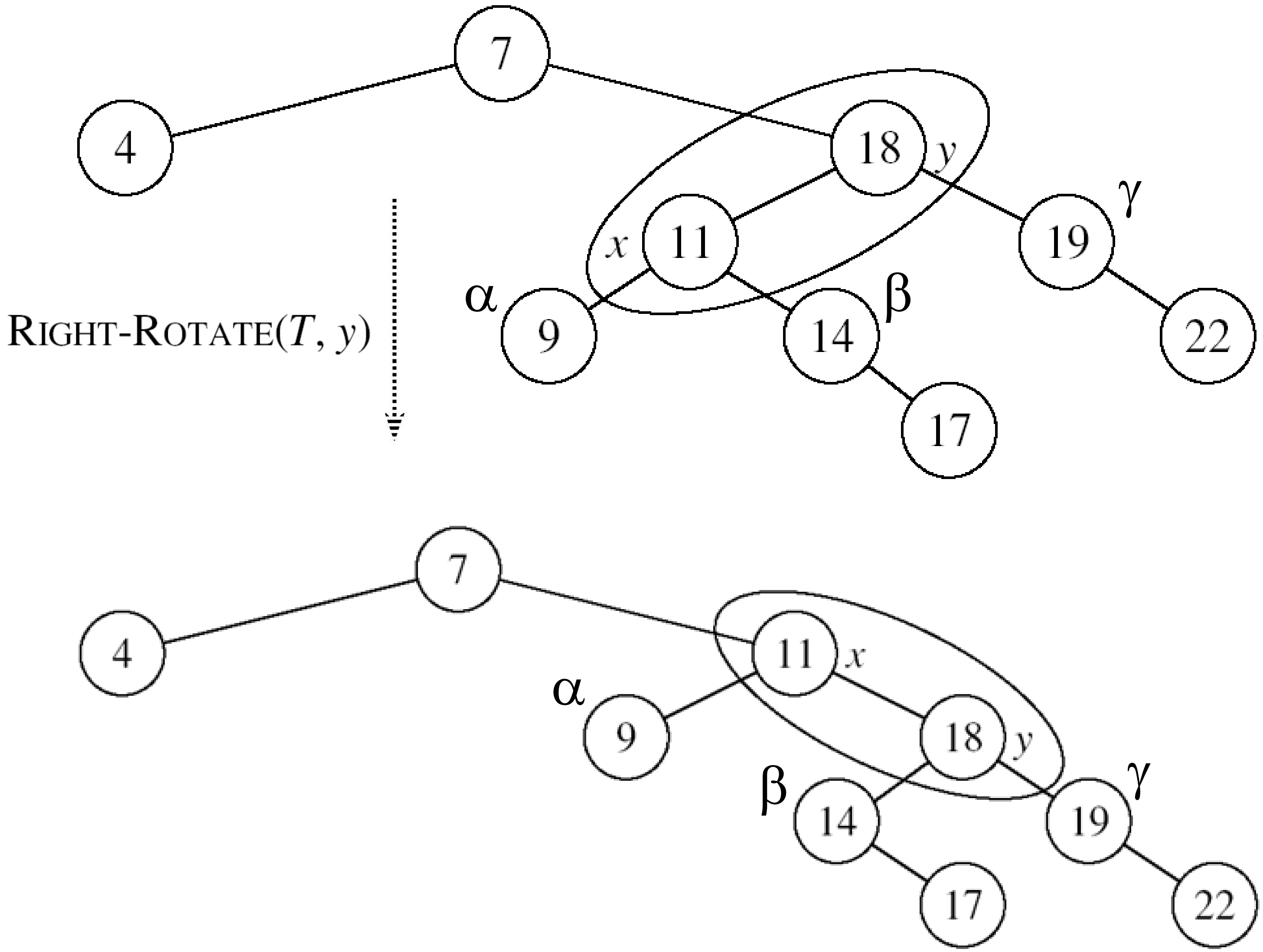
Right Rotations

- Assumptions for a right rotation on a node **x**:
  - The left child of y (that is, x) is not NIL



- Idea:
  - Pivots around the link from y to x
  - Makes x the new root of the subtree
  - y becomes x's right child
  - x's right child becomes y's left child

Example: Right-Rotate



Red-Black Trees: Insertion

- Goal:
  - Insert a new node *z* into a red-black tree.
- Idea:
  - Insert node *z* into the tree as for an ordinary binary search tree.
    - Procedure RB-Insert(*T*, *z*)
  - Color the node *z* **red.**
  - Fix the modified tree by re-coloring nodes and performing rotation to preserve red-black tree property.
    - Use an auxiliary procedure RB-Insert-Fixup(*T*, *z*)

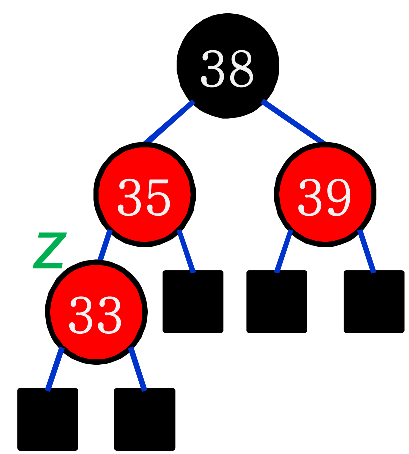
Red-Black Properties Affected by Insert

1. Every node is either **red** or **black** OK!
2. The root is **black** If z is the root ⇒ not OK
3. Every leaf (NIL) is **black** OK!
4. If a node is red, then both its children are black

If p(z) is red ⇒ not OK
z and p(z) are both red

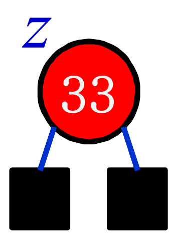
OK!

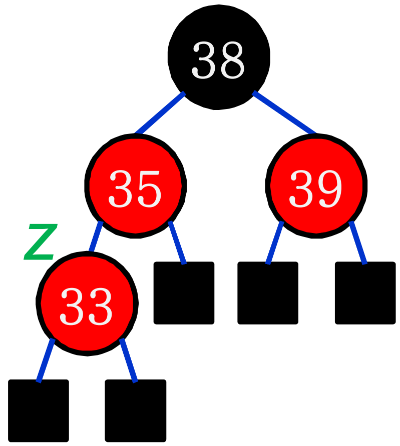
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes



Red-Black Properties Affected by Insert

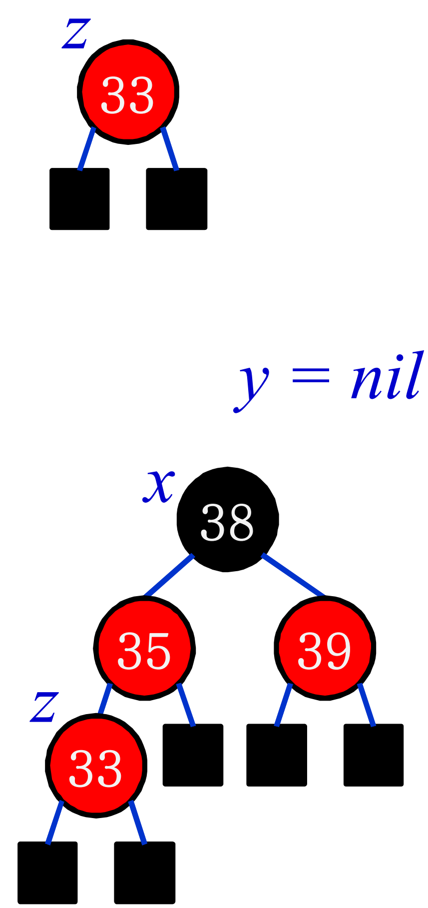
- The RB-Insert(T, z) can violate two properties
  - Root property
    - If z is the root.
  - Internal Property
    - If $p[z]$ is red.
- After each insert there is at most one violation



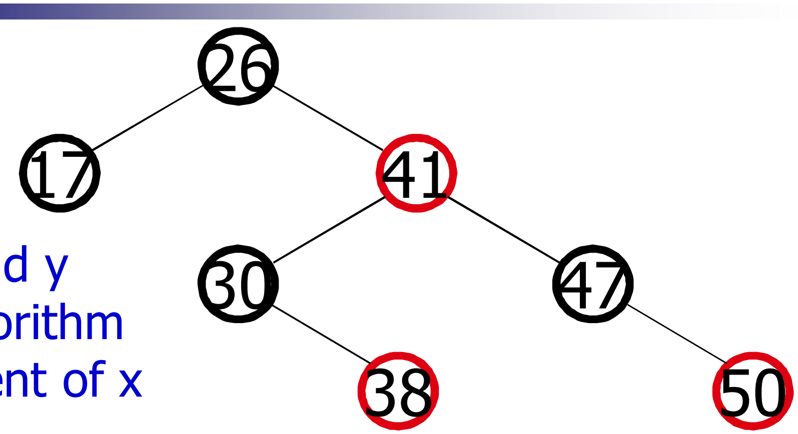


Red-Black Trees: Insertion

```
RB-Insert(T, z)
    y ← nil
    x ← root[T]
    while x ≠ nil do
        y ← x
        if key[z] < key[x] then x ← left[x]
         else x ← right[x]
    p[z] ← y
    if y = nil then root[T] ← z
    else if key[z] < key[y] then left[y] ← z
        else right[y] ← z
    left[z] ← right[z] ← nil
    color[z] ← red
    RB-Insert-Fixup(T, z)
```



Red-Black Trees: Insertion



1. $y \leftarrow NIL$
2. $x \leftarrow root[T]$

- Initialize nodes x and y
- Throughout the algorithm y points to the parent of x

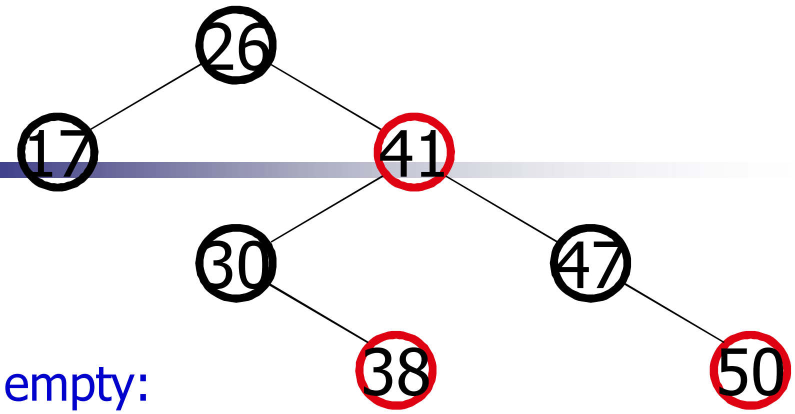
3. **while** $x \neq NIL$
4. **do** $y \leftarrow x$
5. **if** $key[z] < key[x]$
6. **then** $x \leftarrow left[x]$
7. **else** $x \leftarrow right[x]$

- Go down the tree until reaching a leaf
- At that point y is the parent of the node to be inserted

8. $p[z] \leftarrow y$

- Sets the parent of z to be y

RB-Insert(T, z)




```
if y = NIL
   then root[T] ← z
   else if key[z] < key[y]
           then left[y] ← z
           else right[y] ← z
left[z] ← NIL
right[z] ← NIL
color[z] ← RED
RB-INSERT-FIXUP(T, z)
```

The tree was empty: set the new node to be the root

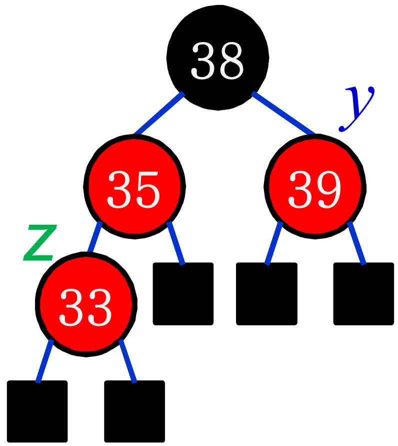
Otherwise, set z to be the left or right child of y, depending on whether the inserted node is smaller or larger than y's key

Set the fields of the newly added node

Fix any inconsistencies that could have been introduced by adding this new red node

Red-Black Trees: Insert-Fixup

- Problem: We may have one pair of consecutive reds where we did the insertion [Internal Property violation].
- Solution: rotate and move it up.
  - 6 cases have to be handled, 3 of which are symmetric to the other 3.
  - We consider the 3 cases in which $p[z]$ is a left child.
  - The other 3 cases in which $p[z]$ is a right child can be handled similarly.



Let y be z's uncle ($p[z]$'s sibling).

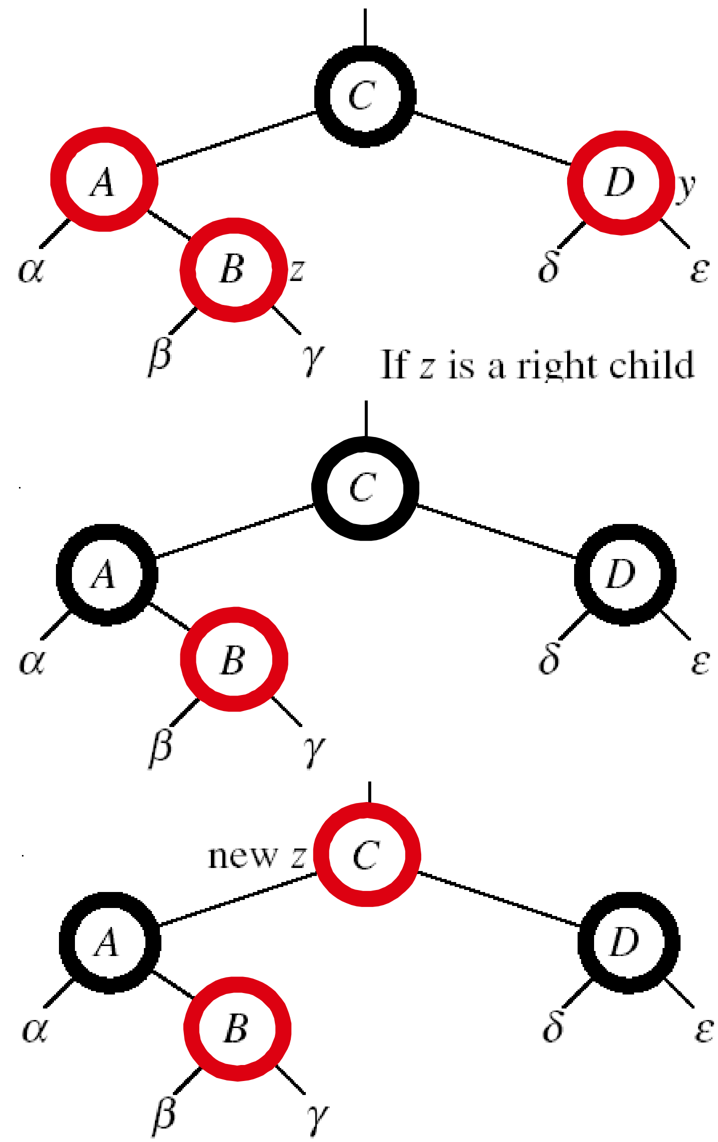
Red-Black Trees: Insert-Fixup (Case 1)

Case 1:

z's "uncle" (y) is **red**

Idea: (z is a right child)

- p[p[z]] (z's grandparent) must be black: z and p[z] are both red
  - Color p[z] **black**
  - Color y **black**
  - Color p[p[z]] **red**
  - z = p[p[z]]
  - Push the **"red"** violation up the tree



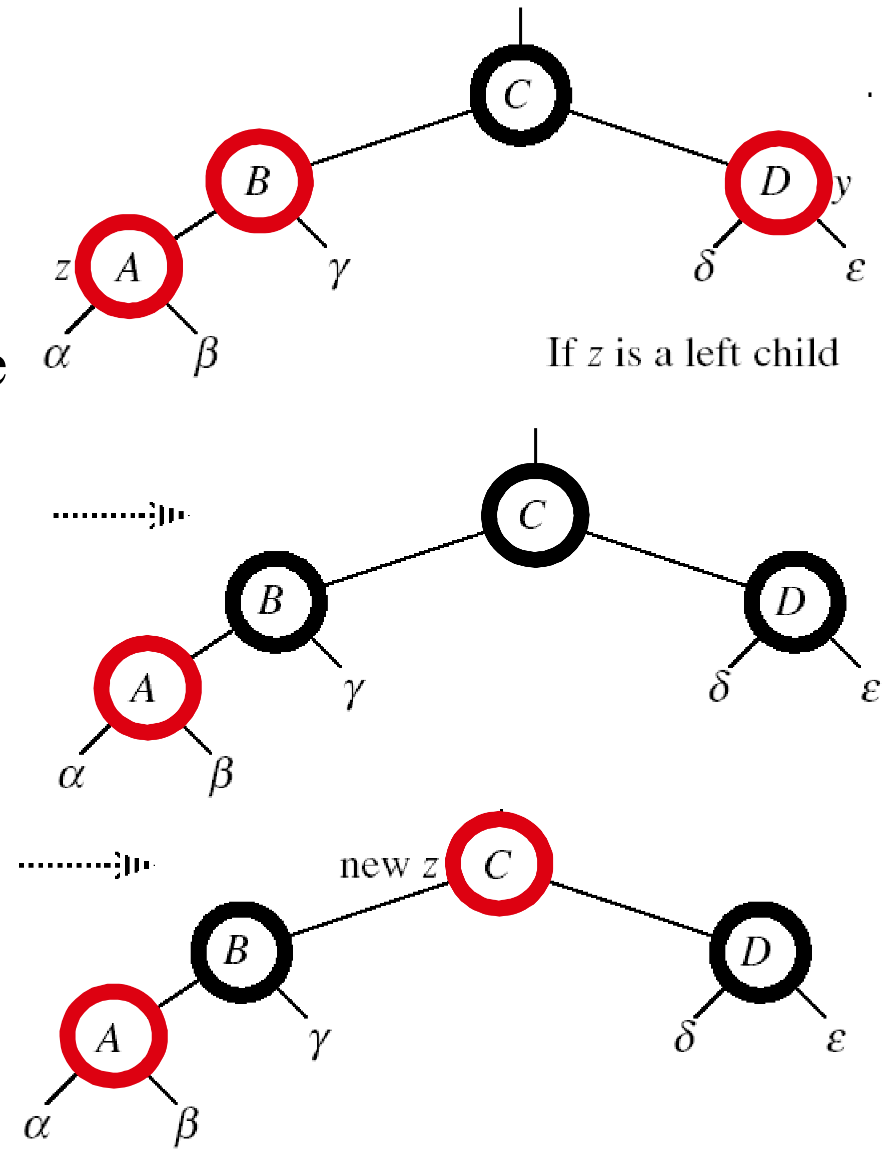
Red-Black Trees: Insert-Fixup (Case 1)

Case 1:

z's "uncle" (y) is **red**

Idea: (z is a left child)

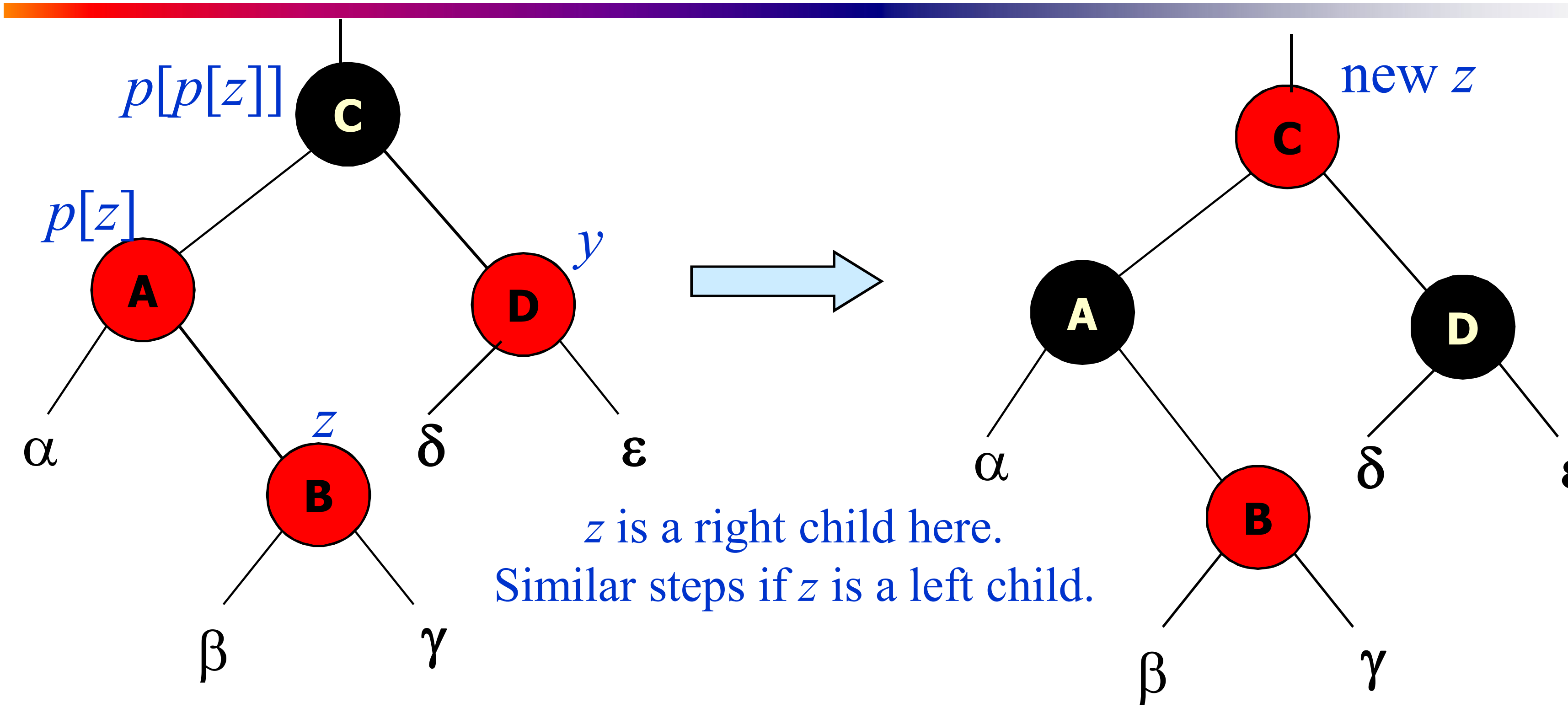
- p[p[z]] (z's grandparent) must be black: z and p[z] are both red



- color p[z] ← **black**
- color y ← **black**
- color p[p[z]] ← **red**
- z = p[p[z]]

- Push the **"red"** violation up the tree

Red-Black Trees: Insert-Fixup (Case 1)



z is a right child here.
Similar steps if *z* is a left child.

- *p*[*p*[*z*]] (*z*'s grandparent) must be black, since *z* and *p*[*z*] are both red and there are no other violations of property 4.
- Make *p*[*z*] and *y* black $\Longrightarrow$ now *z* and *p*[*z*] are not both red. But property 5 might now be violated.
- Make *p*[*p*[*z*]] red $\Longrightarrow$ restores property 5.
- The next iteration has *p*[*p*[*z*]] as the new *z* (i.e., *z* moves up 2 levels).

Red-Black Trees: Insert-Fixup (Case 2)

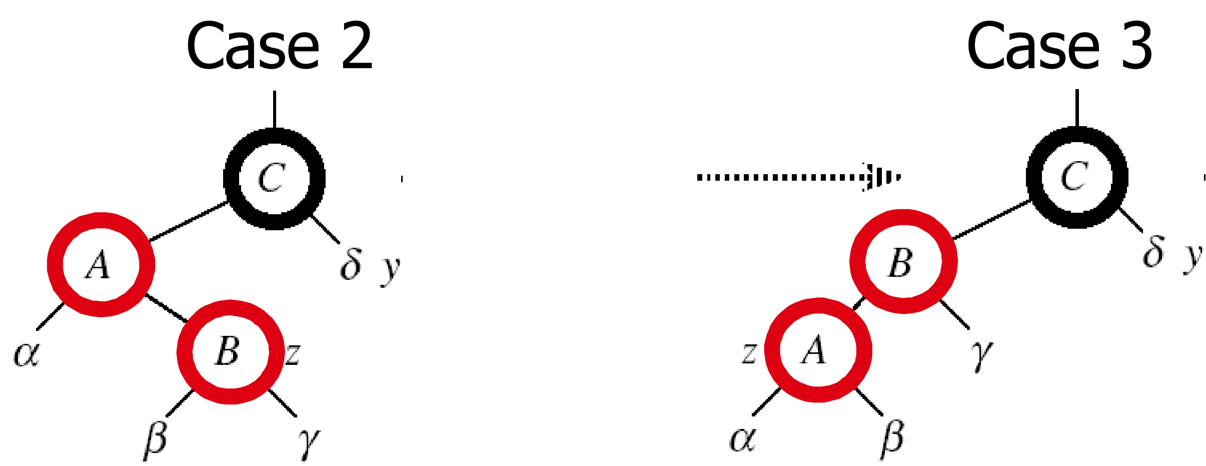
Case 2:

- z's "uncle" (y) is **black**
- z is a right child

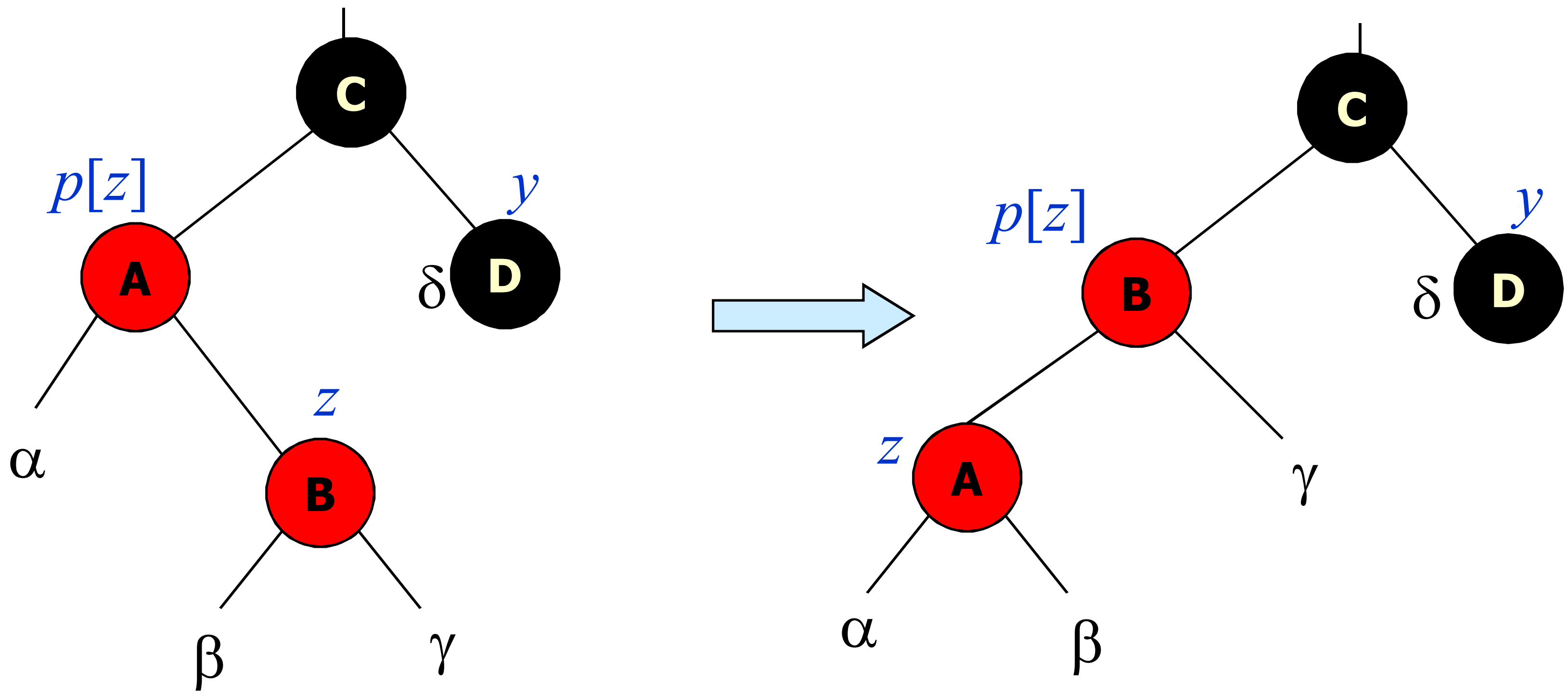
Idea:

- `z ← p[z]`
- LEFT-ROTATE(`T, z`)

⇒ now z is a left child, and both z and `p[z]` are red ⇒ case 3



Red-Black Trees: Insert-Fixup (Case 2)



- Left rotate around $p[z]$.
- $p[z]$ and z switch roles $\Rightarrow$ now z is a left child, and both z and $p[z]$ are red.
- Takes us immediately to case 3.

Red-Black Trees: Insert-Fixup (Case 3)

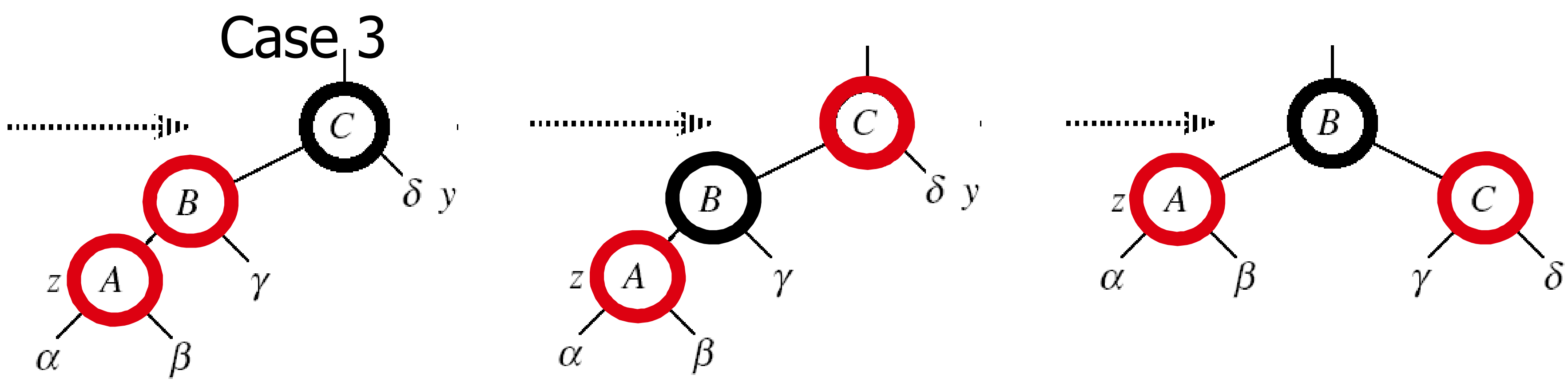
Case 3:

- z's "uncle" (y) is **black**
- z is a left child

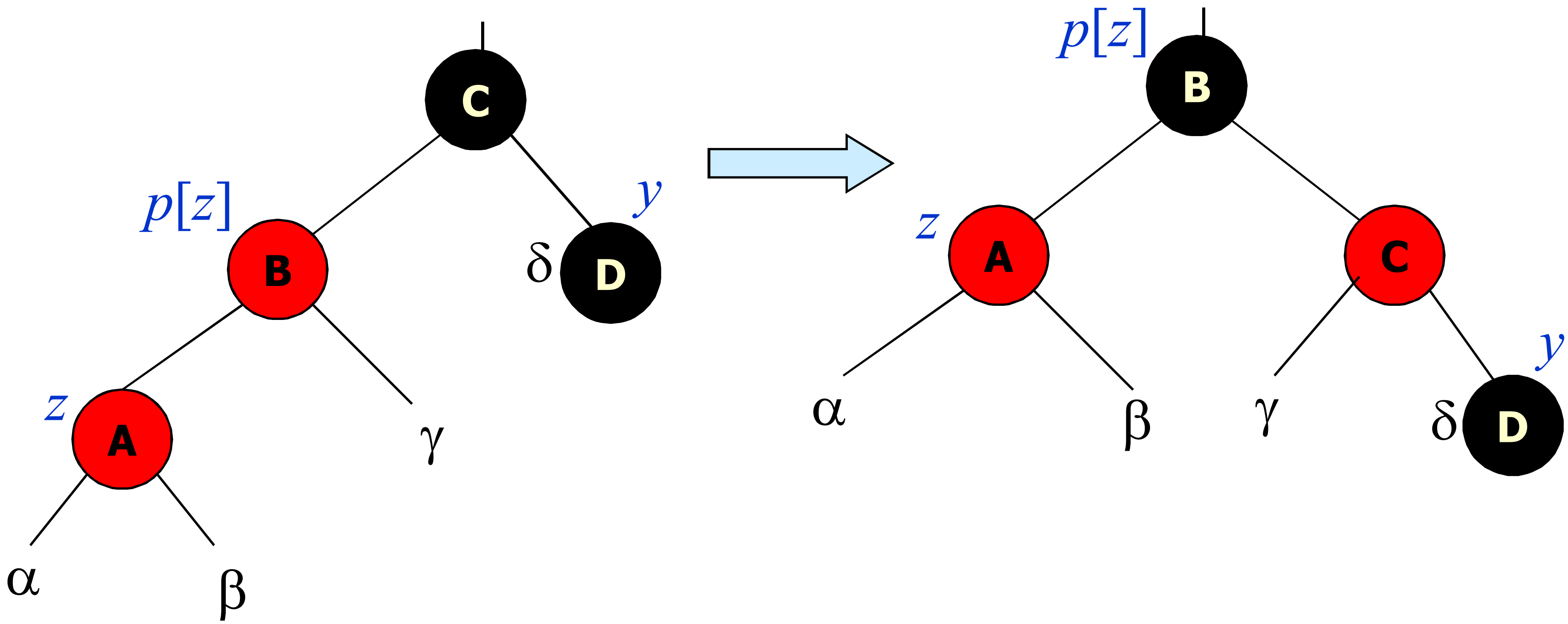
Idea:

- color p[z] ← **black**
- color p[p[z]] ← **red**
- RIGHT-ROTATE(T, p[p[z]])

- No longer have 2 reds in a row
- p[z] is now black



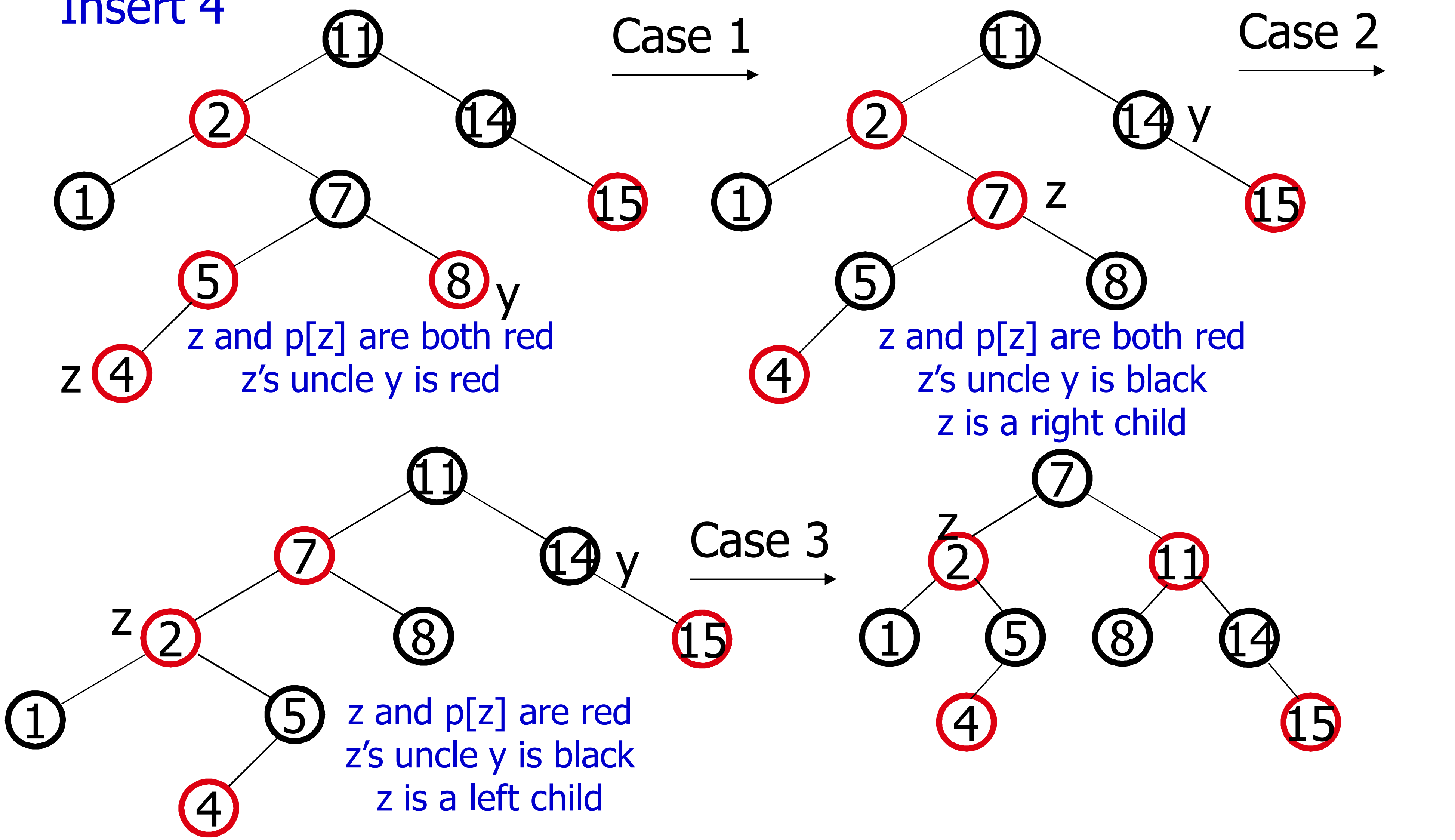
Red-Black Trees: Insert-Fixup (Case 3)



- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate on $p[p[z]]$. Ensures property 4 is maintained.
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.

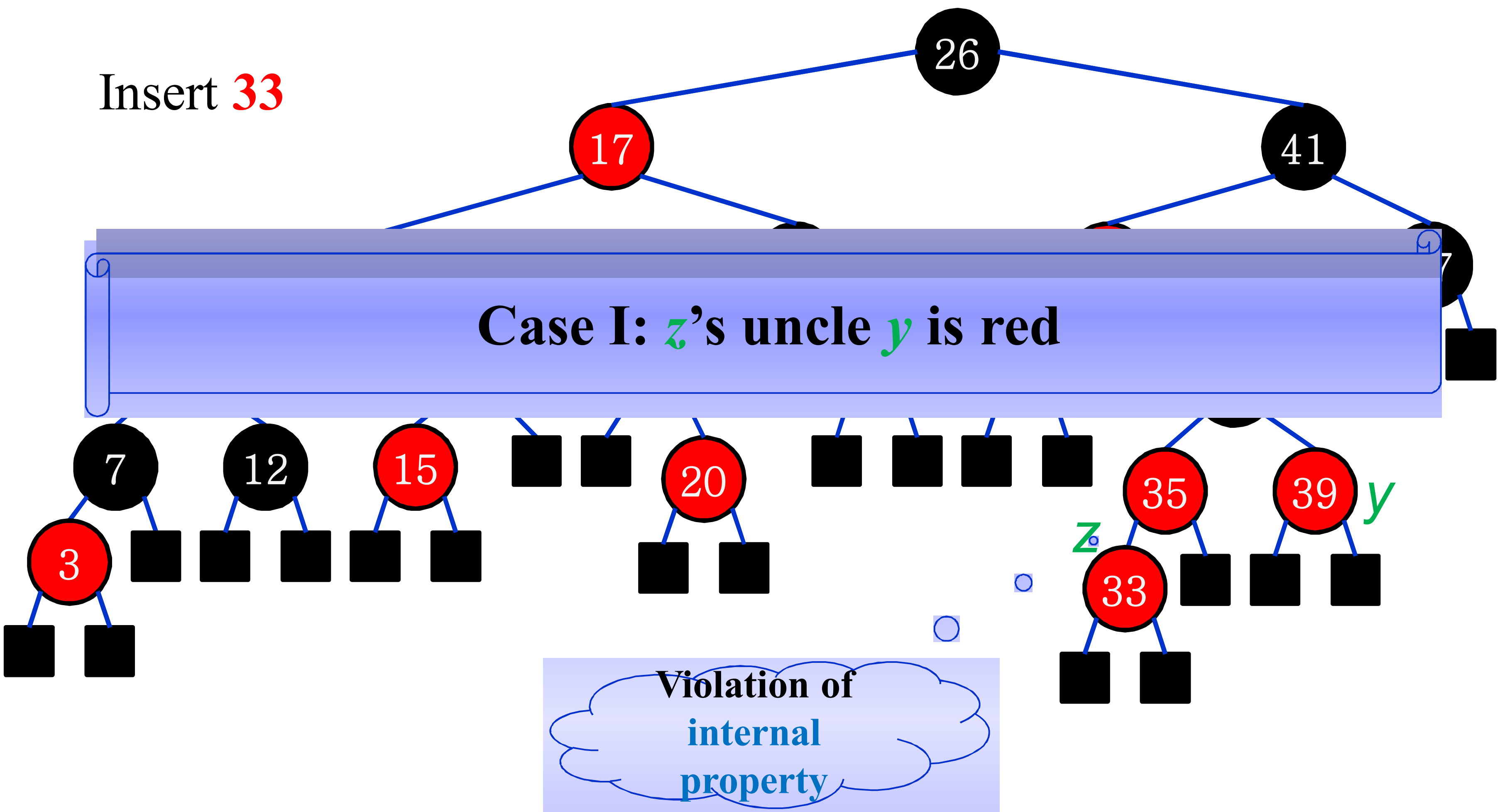
Red-Black Trees: Insert-Fixup (Example)

Insert 4



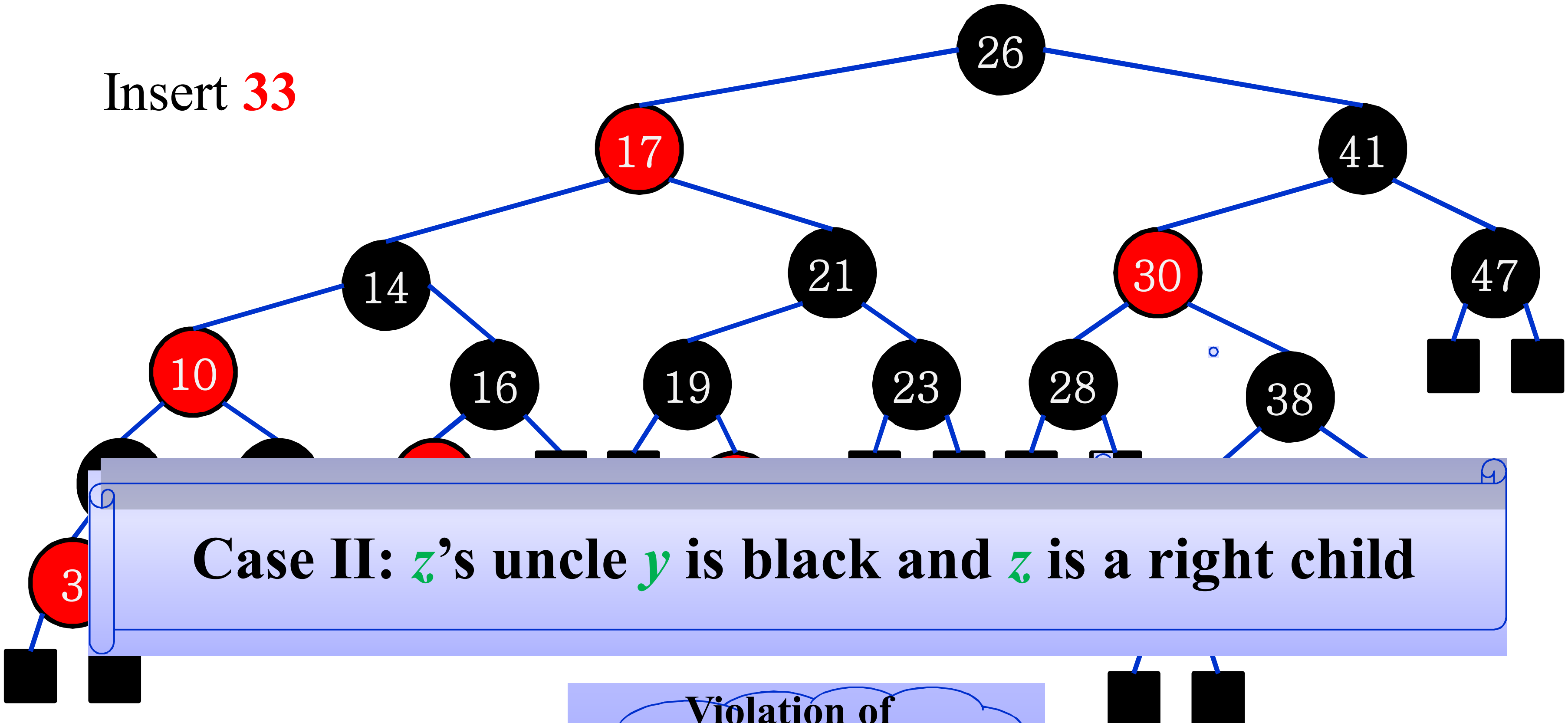
Red-Black Trees: Insert-Fixup

Insert **33**



Red-Black Trees: Insert-Fixup

Insert **33**

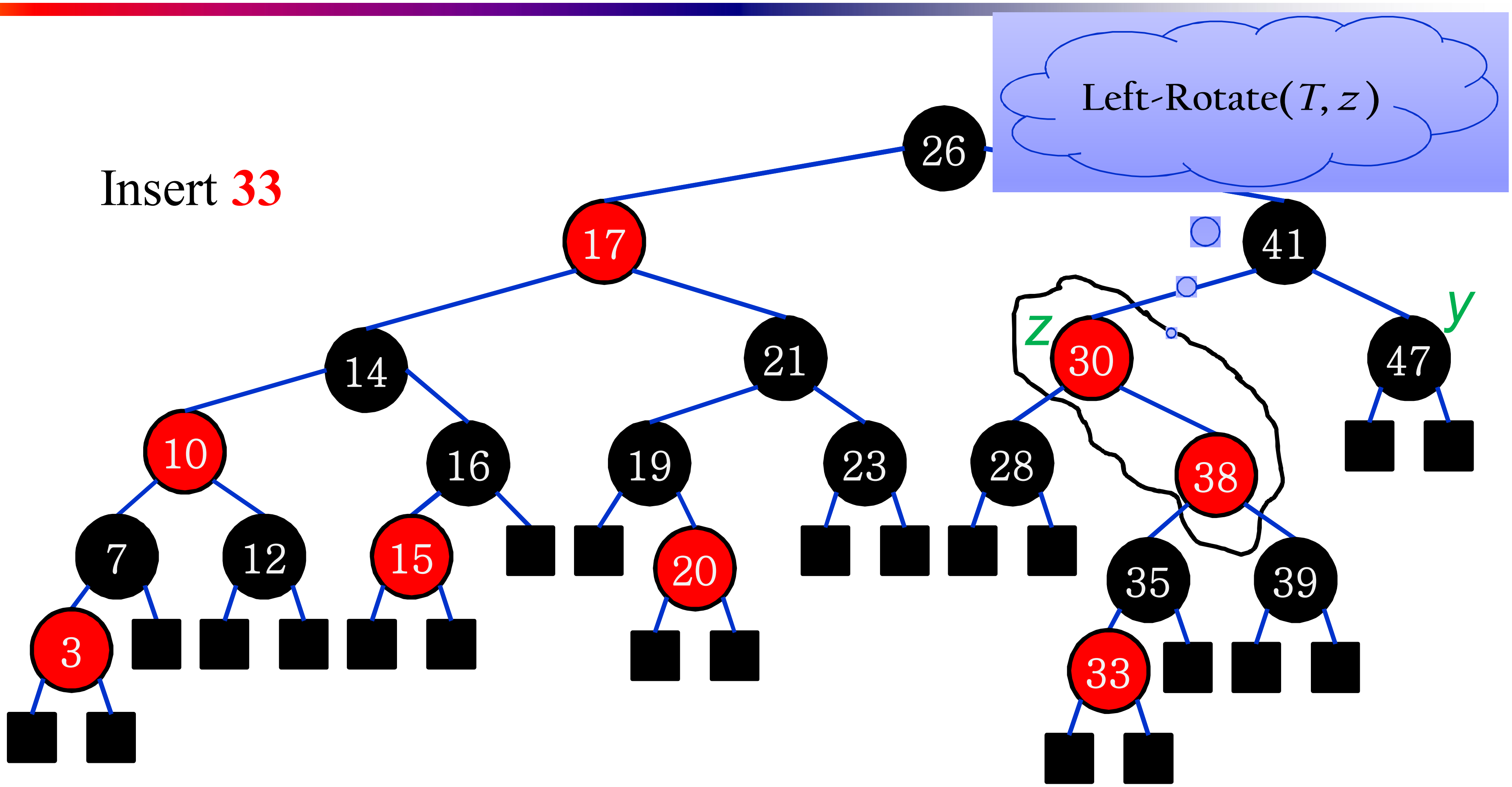


Case II: *z*'s uncle *y* is black and *z* is a right child

Violation of
internal
property

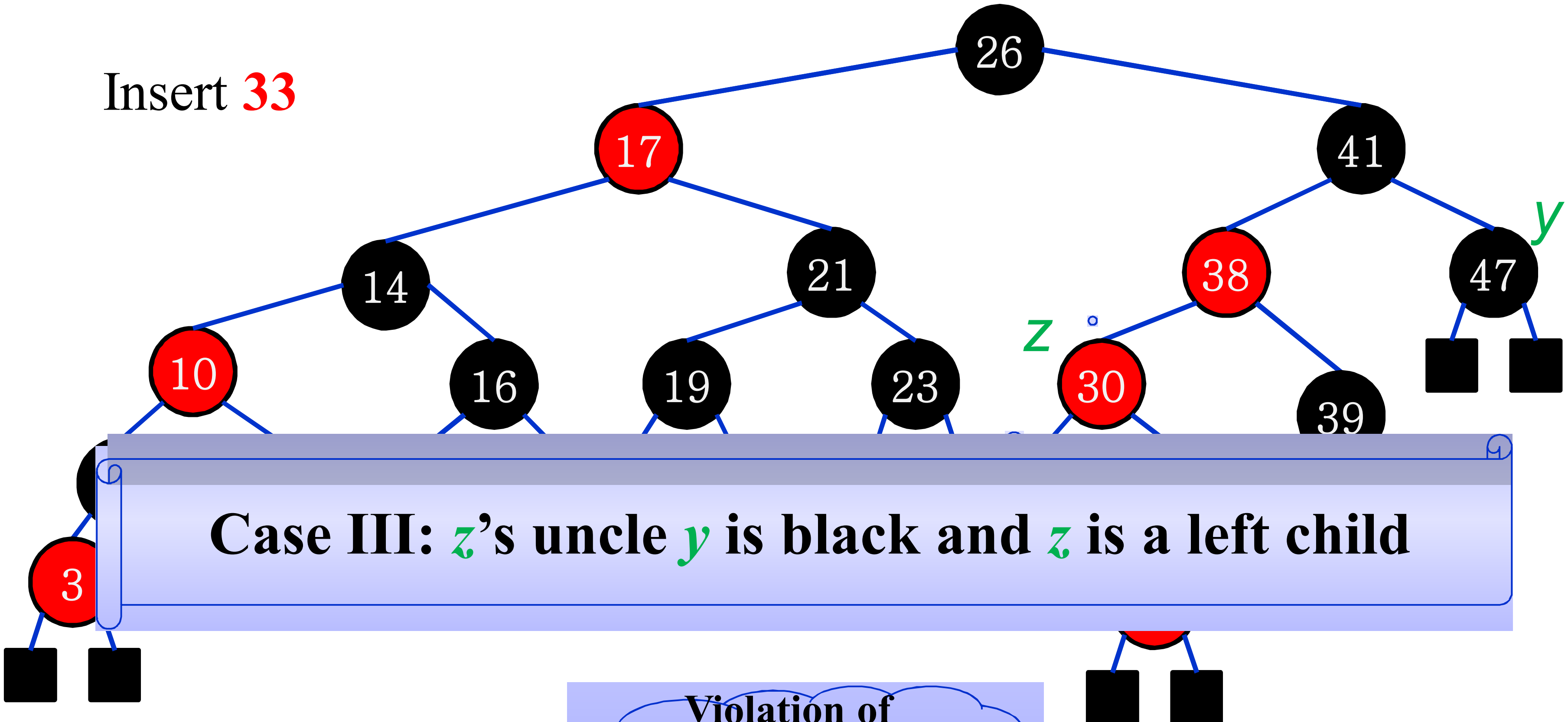
Red-Black Trees: Insert-Fixup

Insert **33**



Red-Black Trees: Insert-Fixup

Insert **33**

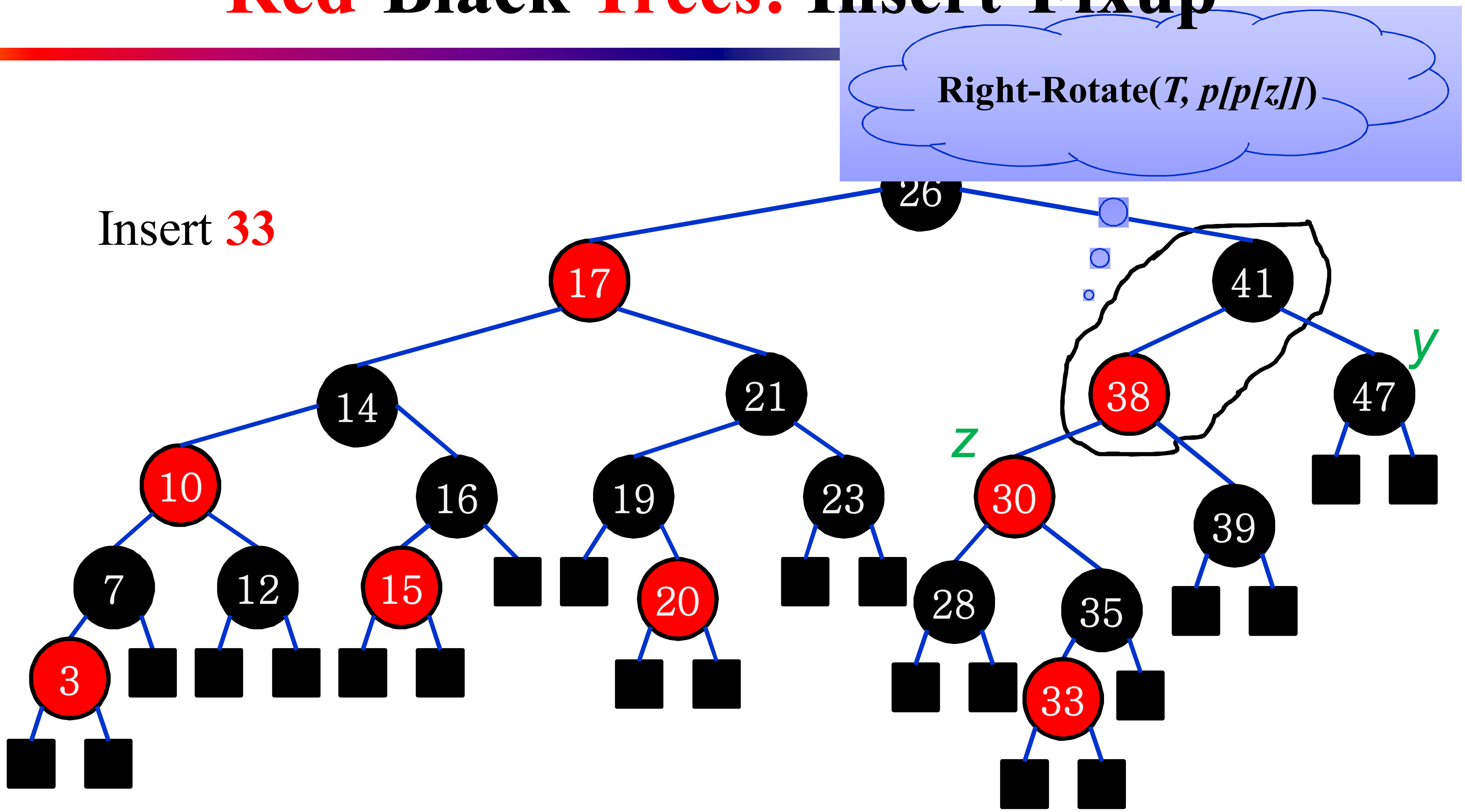


Case III: *z*'s uncle *y* is black and *z* is a left child

Violation of internal property

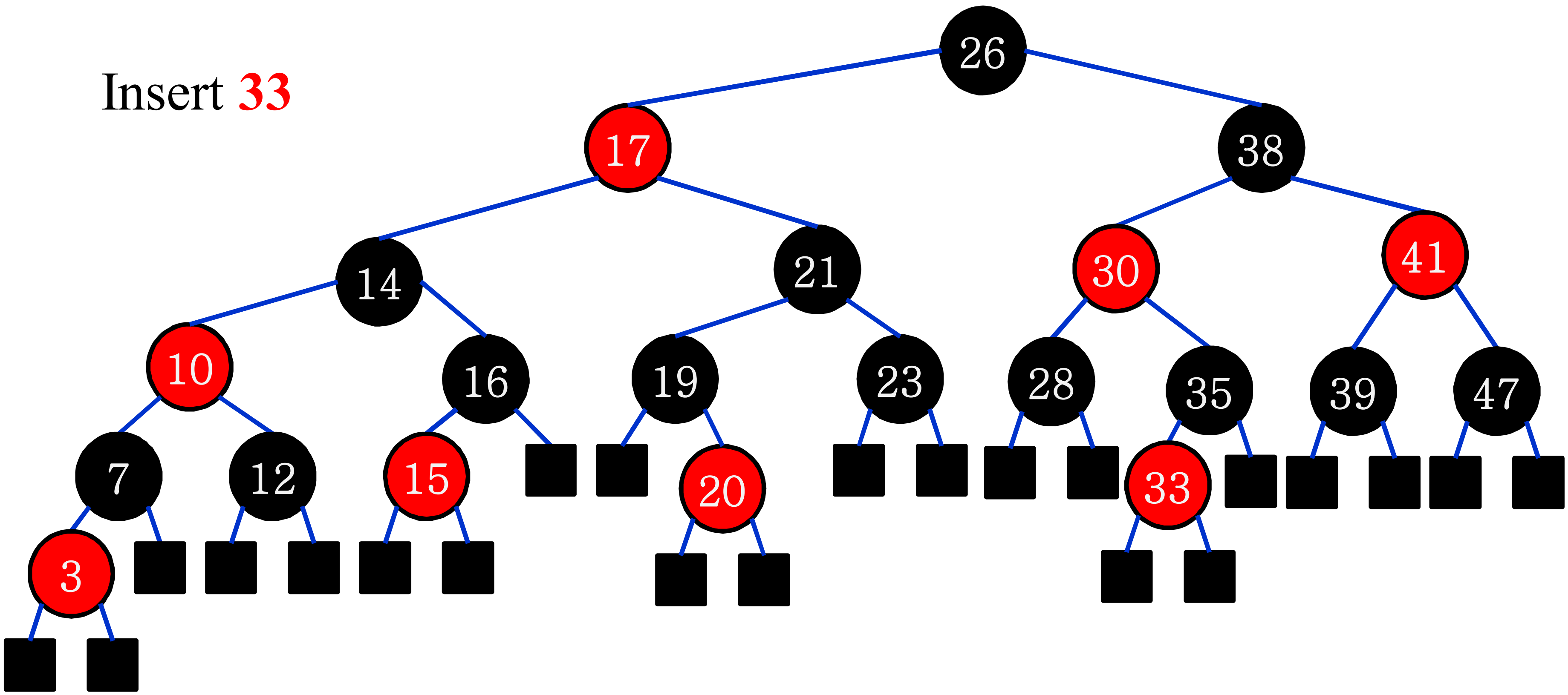
Red-Black Trees: Insert-Fixup

Insert **33**



Red-Black Trees: Insert-Fixup

Insert **33**



Red-Black Trees: Insert-Fixup

```
RB-Insert-Fixup(T, z)
1     while color[p[z]] = = red do
2         if p[z] = = left[p[p[z]]] then
3             y = right[p[p[z]]]          \\ set y as z's uncle
4             if color[y] = = red  then      \\ case I
5                 color[p[z]] ← black
6                 color[y] ←  black
7                 color[p[p[z]]] ← red
8                 z = p[p[z]]
9             else
10                if z = = right[p[z]] then    \\ case II
11                    z ← p[z]
12                    Left-Rotate(T, z)
13                color[p[z]] ← black     \\ case III
14                color[p[p[z]]] ← red
15                Right-Rotate(T, p[p[z]])
16        else same as then clause of line 2 with left and right exchanged.
17    color[root[T]] ←  black          \\ for root property
```

Red-Black Trees: Correctness of Insertion

Loop invariant:

- At the start of each iteration of the **while** loop in RB-Insert-Fixup(T, z),
  - z is red.
  - If z is the root, then $p[z]$ is black [*nil*].
  - There is at most one red-black violation:
    - Property 2: z is a red root, or
    - Property 4: z and $p[z]$ are both red.

Red-Black Trees: Correctness of Insertion

- **Initialization:** √
- **Termination:**
  - The loop terminates only if $p[z]$ is black. Hence, Property 4 is OK.
  - The last line (Line 17) ensures Property 2 always holds.
- **Maintenance:**
  - Violation of Property 2: We drop out when z is the root (since then $p[z]$ is *nil*, which is black).
  - Violation of Property 4: When we start the loop body, the only violation is of Property 4.

Red-Black Trees: Analysis of Insertion

- $O(\log n)$ time to get through RB-Insert up to the call of RB-Insert-Fixup.
- Within RB-Insert-Fixup:
  - Each iteration takes $O(1)$ time.
  - Each iteration but the last moves z up 2 levels.
  - $O(\log n)$ levels $\Rightarrow$ $O(\log n)$ time.
- Thus, insertion in a red-black tree takes $O(\log n)$ time.

Red-Black Trees: Deletion

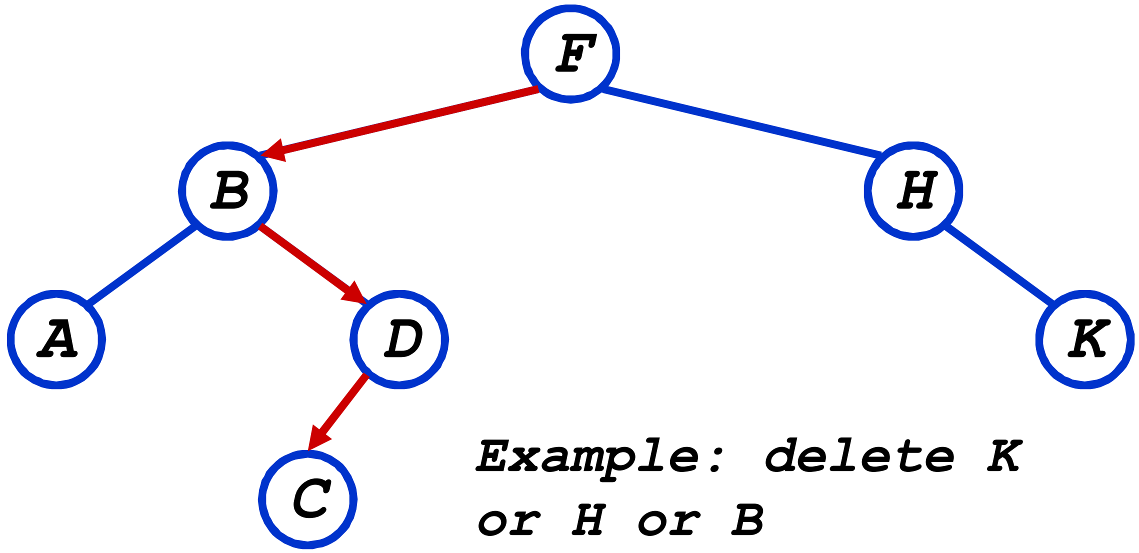
- Deletion, like insertion, should preserve all the Red-Black properties.

- The properties that may be violated depends on the color of the deleted node.
  - Red – OK. Why?
  - Black ???

- Steps:
  - Do regular BST deletion.
  - Fix any violations of Red-Black properties that may result.

Operations of BSTs: Delete

- Delete node *z*
- 3 cases:
  - *z* has no children:
    - Remove *z*
  - *z* has one child:
    - Splice out *z*
  - *z* has two children:
    - Find its inorder successor *y*
    - Replace *z* with *y*
    - Delete *y*



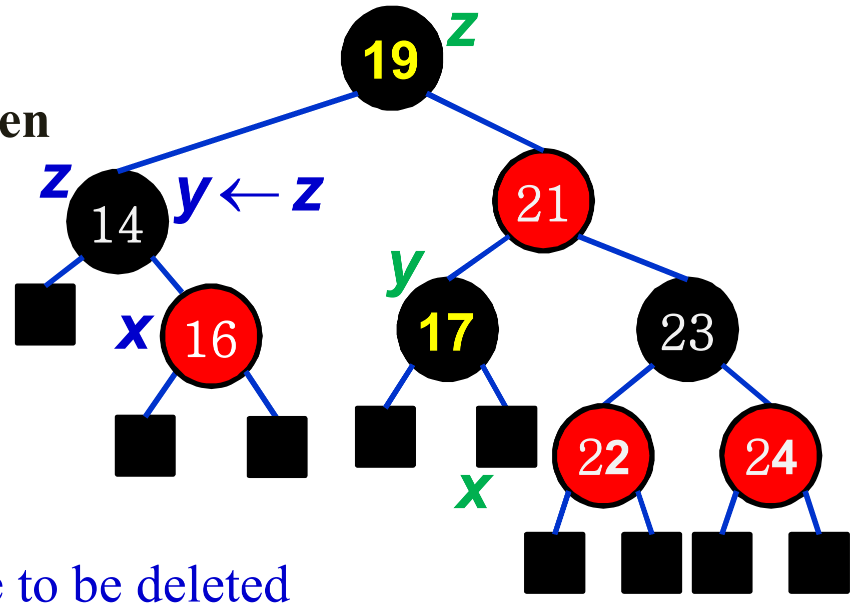
Example: delete K or H or B

[redacted]

Red-Black Trees: Deletion

RB-Delete(T, z)

```
if left[z] = nil or right[z] = nil then
        y ← z
else y ← TREE_SUCCESSOR(z)
if left[y] ≠ nil then
        x ← left[y]
else x ← right[y]
p[x] ← p[y]
if p[y] = nil then
        root[T] ← x
else if y = left[p[y]] then
        left[p[y]] ← x
else right[p[y]] ← x
if y ≠ z then
        key[z] ← key[y]
if color[y] = black then
        RB-Delete-Fixup(T, x)
return y
```



z: node to be deleted

y: points to z when z has < 2 children and is deleted,

y: otherwise, points to z's successor and will move to z's position

x: node that moves to y's original position

Red-Black Properties Affected by Delete

- If *y* is black, we could have violations of red-black properties:

1. Every node is either **red** or **black** OK!
2. The root is **black**

   If y is the root and x is red, then the root has become red.
   ⇒ not OK
3. Every leaf (NIL) is **black** OK!
4. If a node is red, then both its children are black

   If p[y] and x are both red
   ⇒ not OK
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

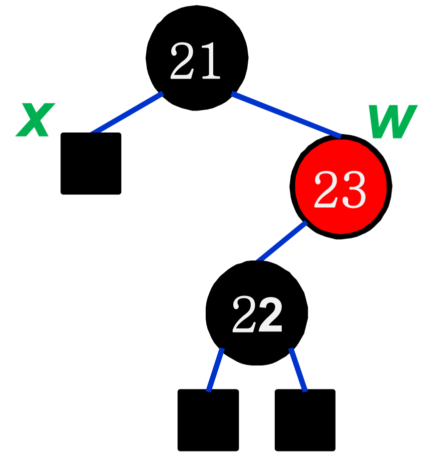
   Any path containing y now has 1 fewer black node
   ⇒ not OK

Red-Black Properties Affected by Delete

- Violation of Prop. 5: Any path containing *y* now has 1 fewer black node.
  - Correct by giving *x* an "extra black."
  - Add 1 to count of black nodes on paths containing *x*.
  - Now property 5 is OK, but property 1 is not.
  - *x* is either ***doubly black*** (if *color*[*x*] = BLACK) or ***red & black*** (if *color*[*x*] = RED).
  - The attribute *color*[*x*] is still either RED or BLACK. No new values for *color* attribute.
  - In other words, the extra blackness on a node is by virtue of *x* pointing to the node.
- Remove the violations by calling RB-Delete-Fixup.

Red-Black Trees: Delete-Fixup

- **Idea:** Move the extra black up the tree until x points to a red & black node $\Rightarrow$ turn it into a black node,
- x points to the root $\Rightarrow$ just remove the extra black, or
- We can do certain rotations and recoloring and finish.
- Within the **while** loop:
  - x always points to a nonroot ***doubly black*** node.
  - w is x's sibling.
  - w cannot be *nil*, since that would violate Property 5 at $p[x]$.

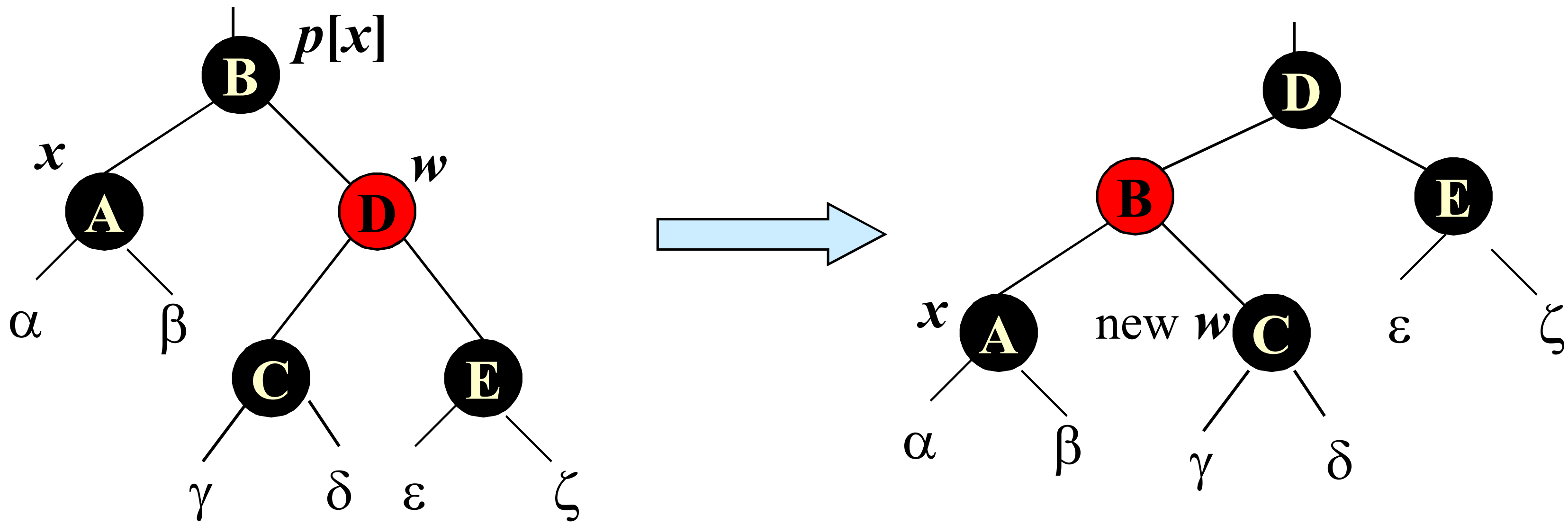


- 8 cases in all, 4 of which are symmetric to the other.

RB-Delete-Fixup: Case 1 – *w* is red

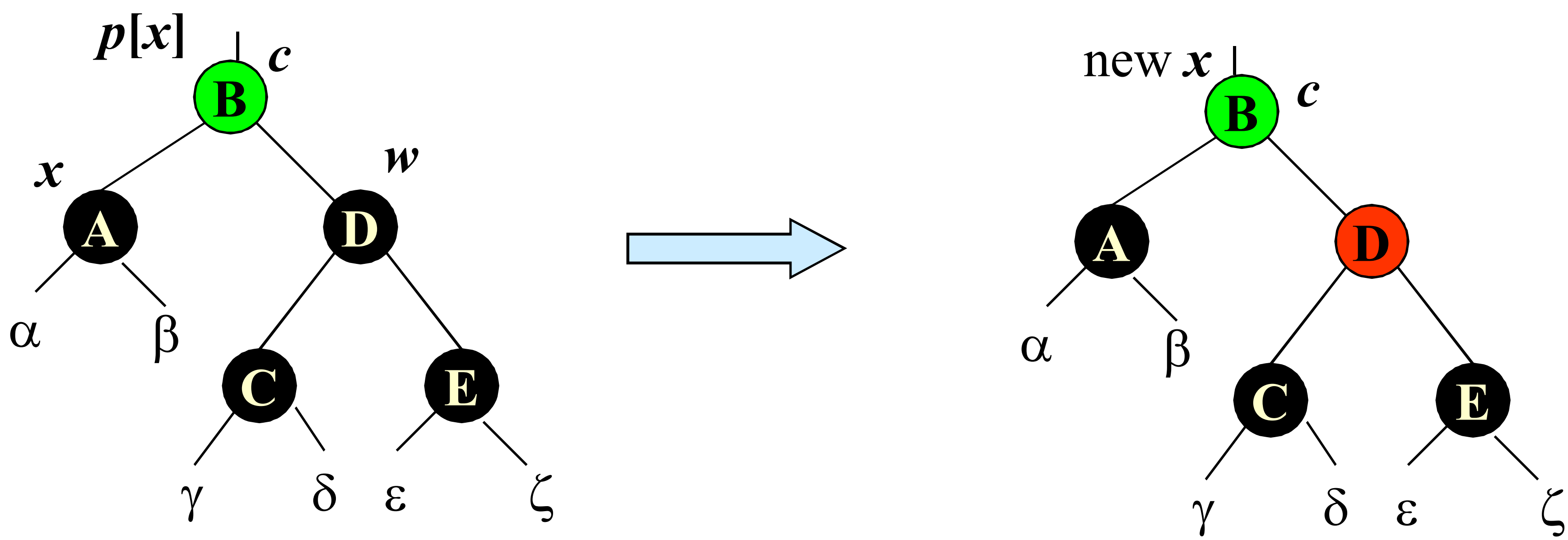
After removing or moving the black node *y*, we push its blackness onto *x*

x: now a nonroot doubly black node, *w*: sibling of *x*



- *w* must have black children.
- Make *w* black and *p*[*x*] red (because *w* is red *p*[*x*] couldn't have been red).
- Then left rotate on *p*[*x*].
- New sibling of *x* was a child of *w* before rotation ⇒ must be black.
- Go immediately to case 2, 3, or 4.

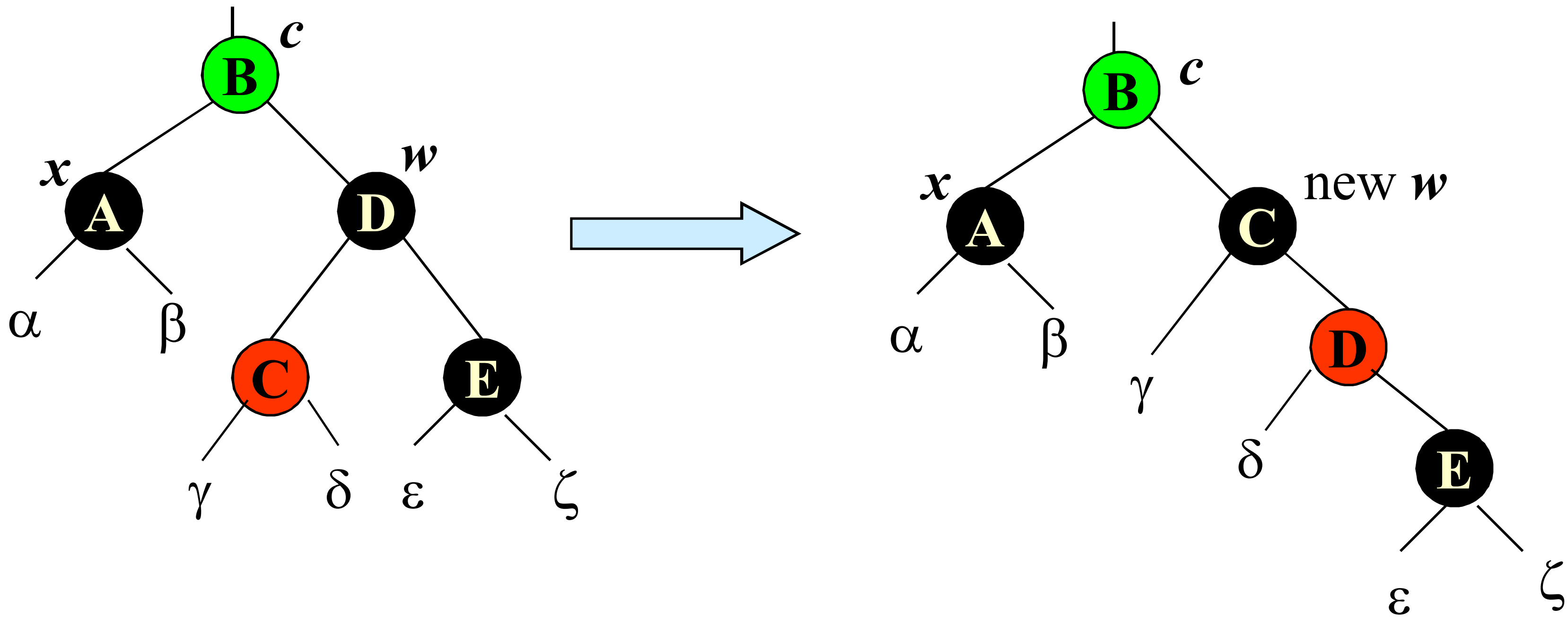
RB-Delete-Fixup: Case 2 – *w* is black, both *w*'s children are black



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

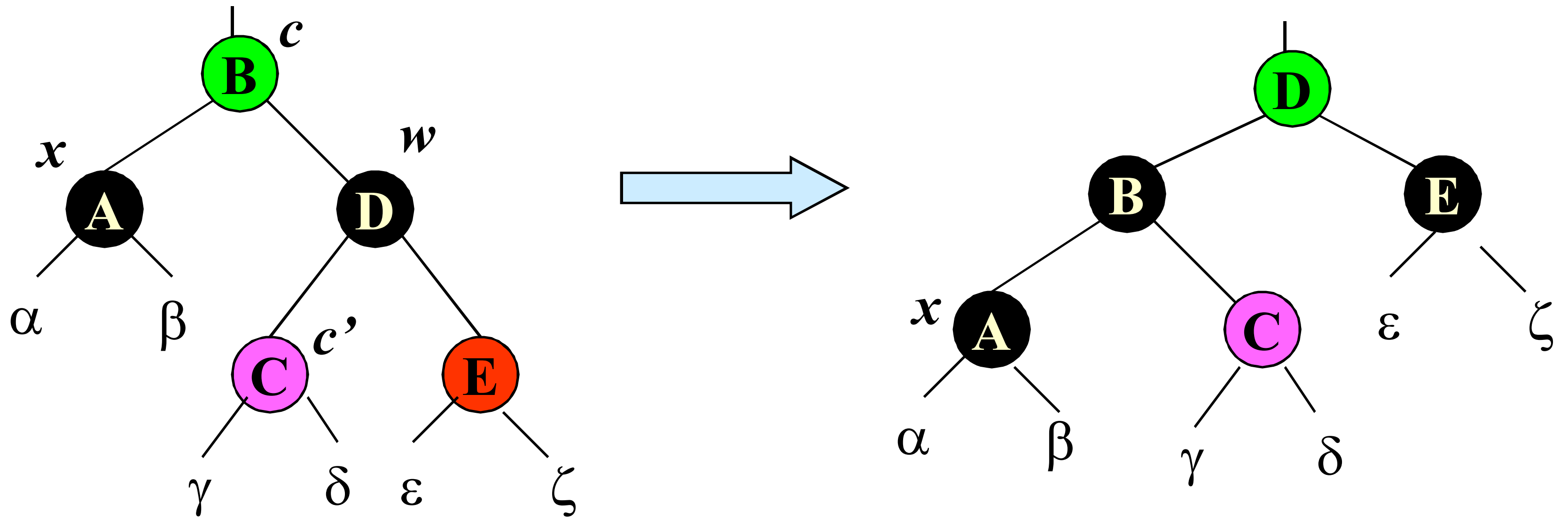
Then new x is made black in the last line.

RB-Delete-Fixup: Case 3 – *w* is black, *w*'s left child is red, *w*'s right child is black



- 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 ⇒ Case 4.

RB-Delete-Fixup: Case 4 – *w* is black, *w*'s right child is red



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

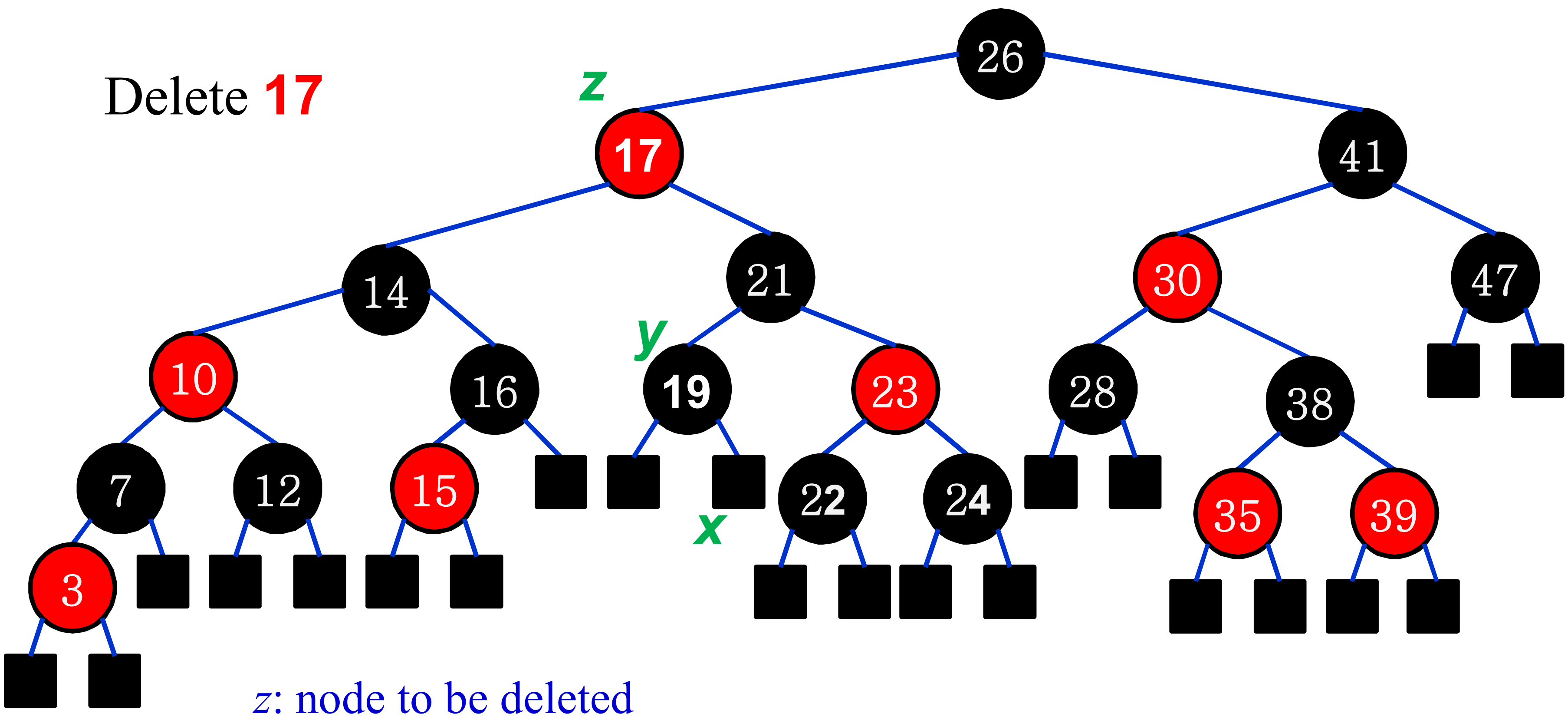
Red-Black Trees: Analysis of Deletion

- $O(\log n)$ time to get through RB-Delete up to the call of RB-Delete-Fixup.
- Within RB-Delete-Fixup:
  - Case 2 is the only case in which more iterations occur.
    - x moves up 1 level.
    - Hence, $O(\log n)$ iterations.
  - Each of cases 1, 3, and 4 has 1 rotation

    $\Rightarrow \leq 3$ rotations in all.
  - Hence, $O(\log n)$ time.

Red-Black Trees: Delete-Fixup

Delete **17**



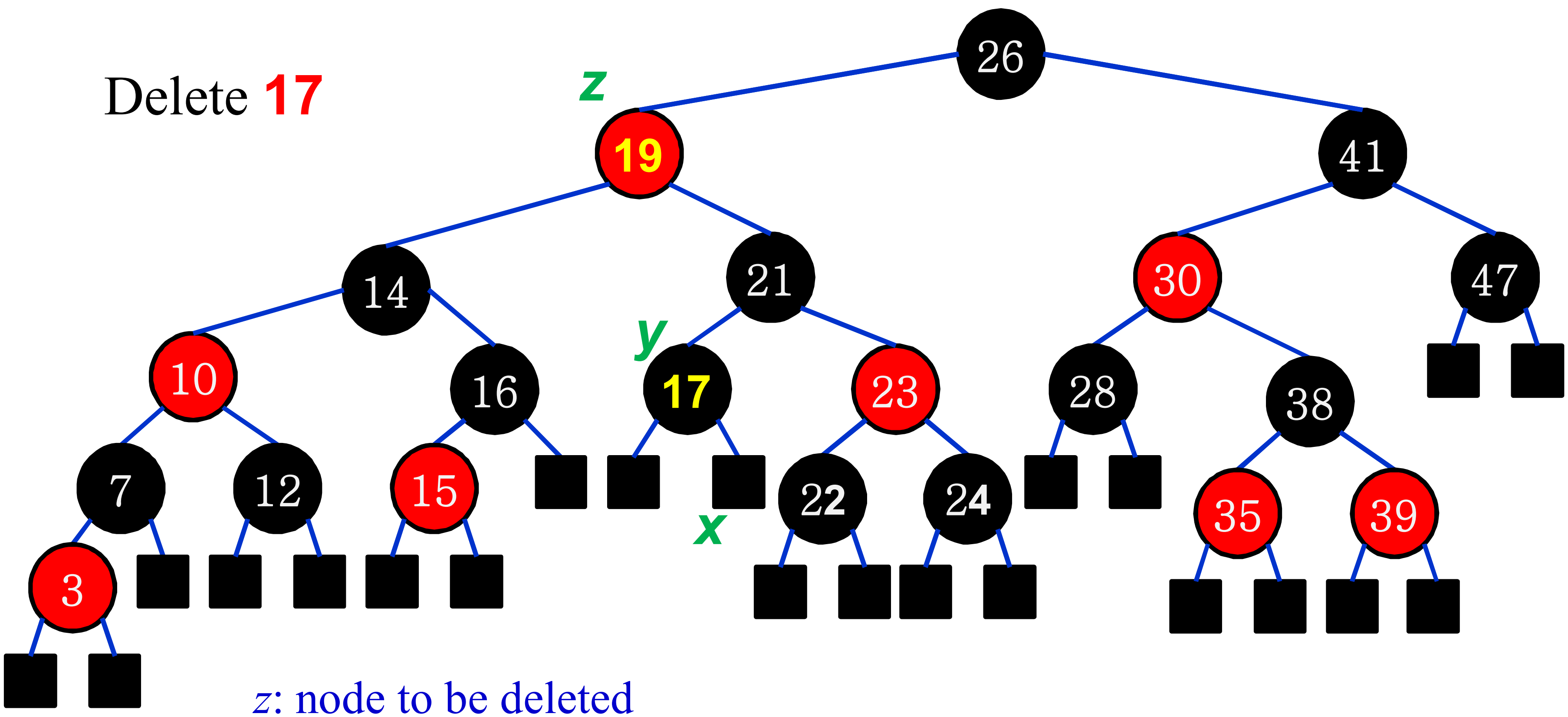
z: node to be deleted

y: points to *z* when *z* has < 2 children and is deleted,
otherwise, points to *z*'s successor and will move to *z*'s position

x: node that moves to *y*'s original position

Red-Black Trees: Delete-Fixup

Delete **17**



z: node to be deleted

y: points to *z* when *z* has < 2 children and is deleted,
otherwise, points to *z*'s successor and will move to *z*'s position

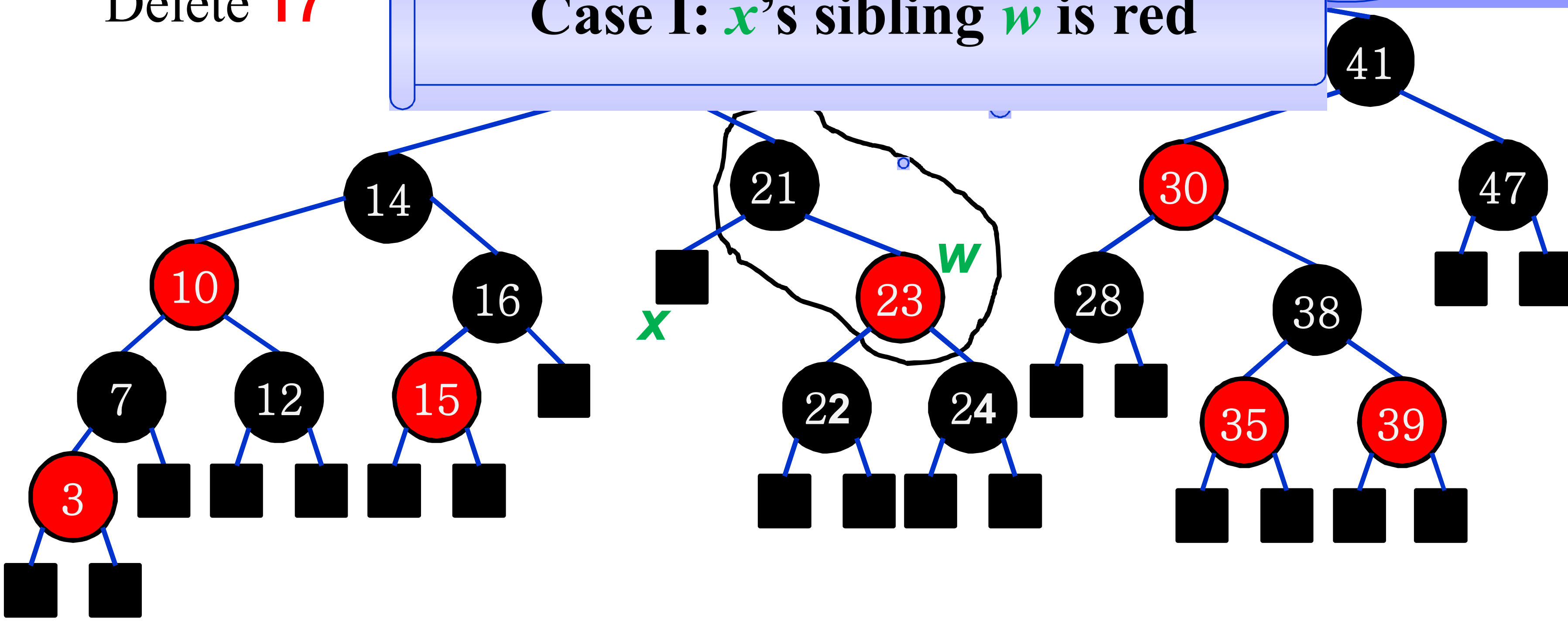
x: node that moves to *y*'s original position

Red-Black Trees: Delete-Fixup

Left-Rotate($T, p[x]$)

Delete **17**

Case I: x's sibling w is red



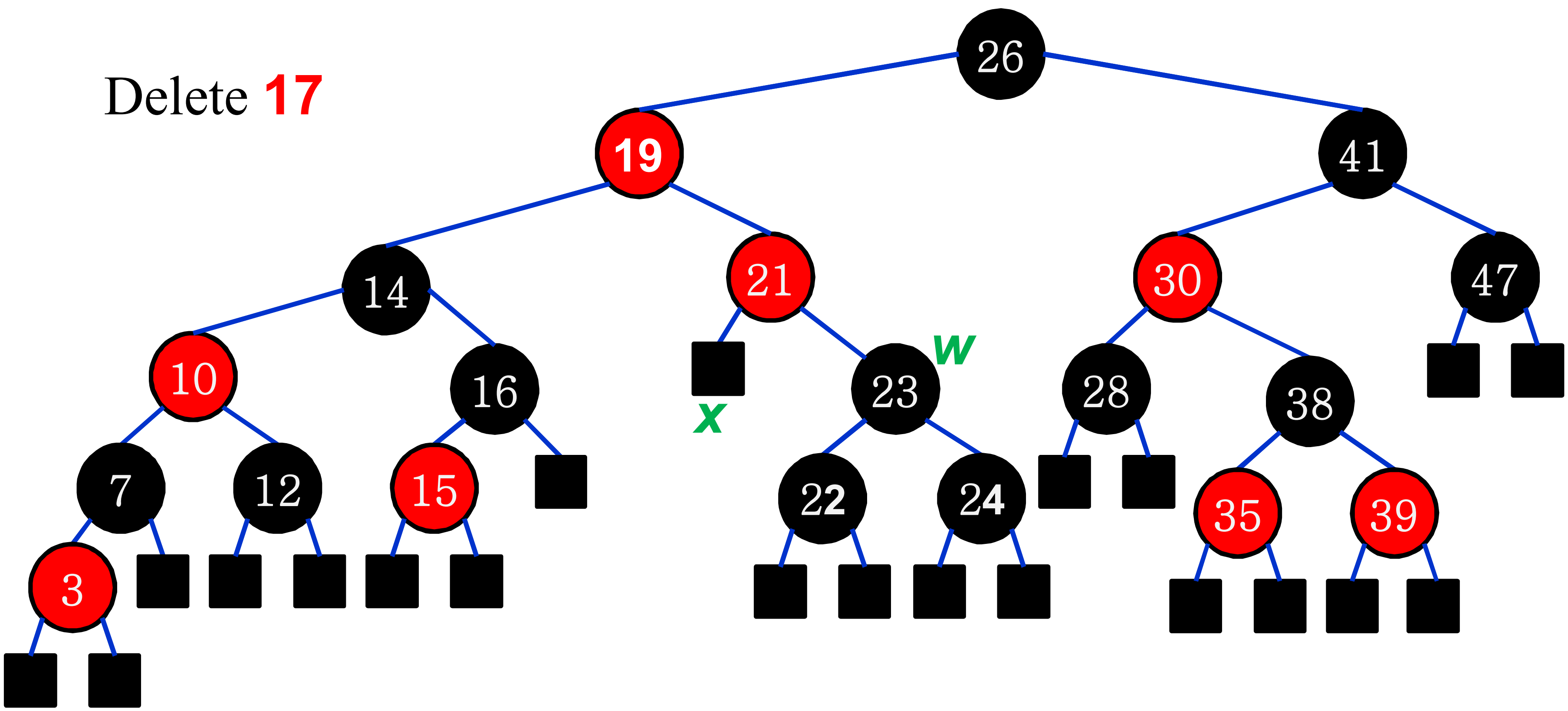
After removing or moving the black node y, we push its blackness onto x

x: now a nonroot doubly black node

w: sibling of x and w cannot be *nil* since x is doubly black

Red-Black Trees: Delete-Fixup

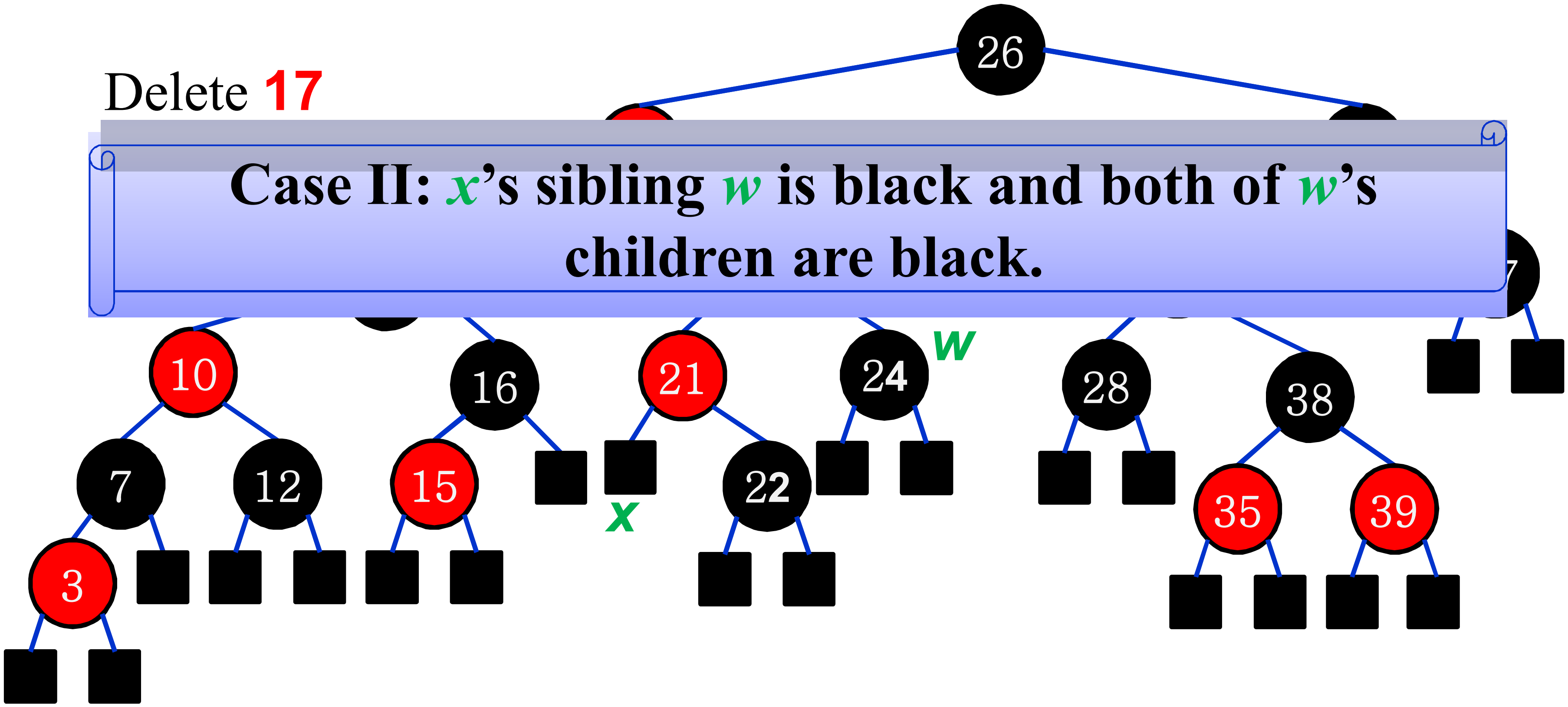
Delete **17**



Red-Black Trees: Delete-Fixup

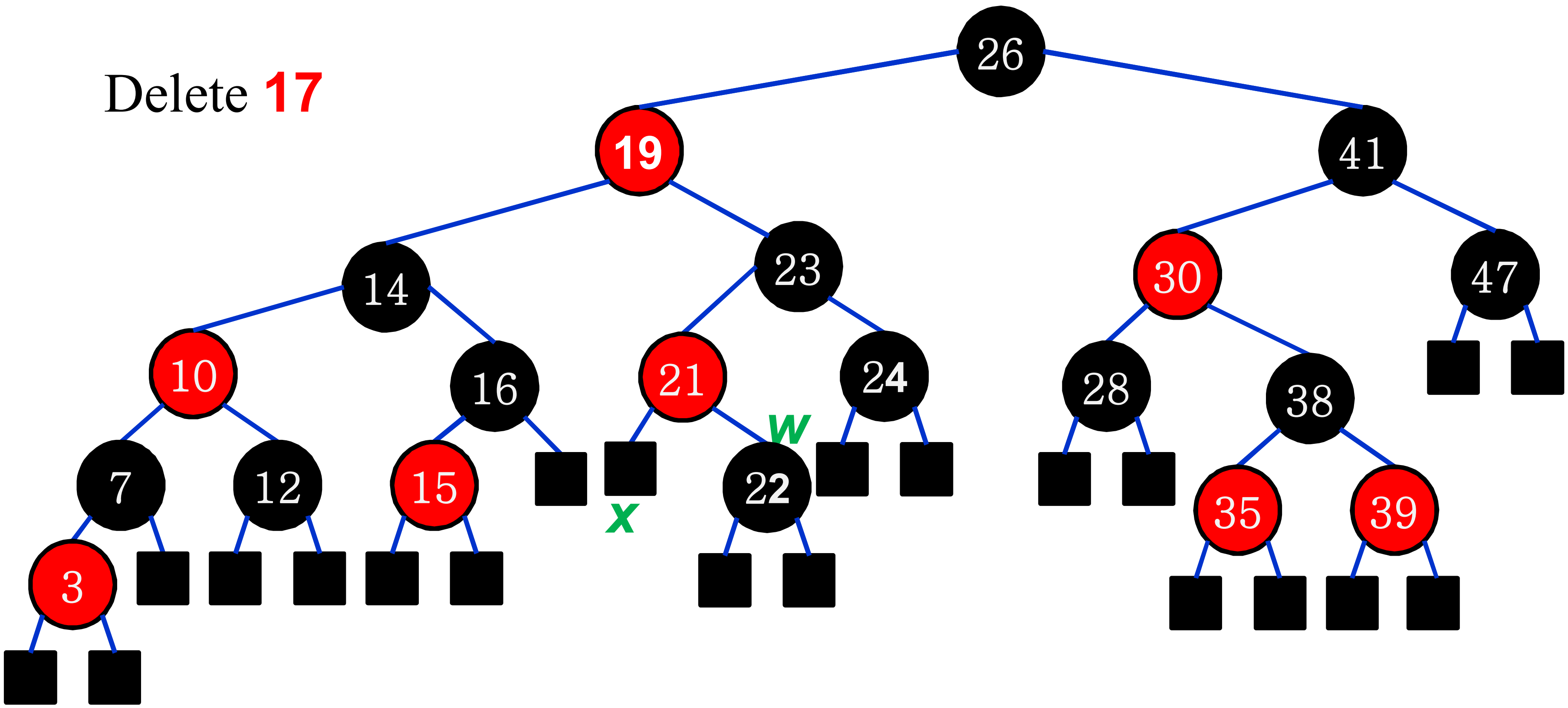
Delete **17**

Case II: *x*'s sibling *w* is black and both of *w*'s children are black.



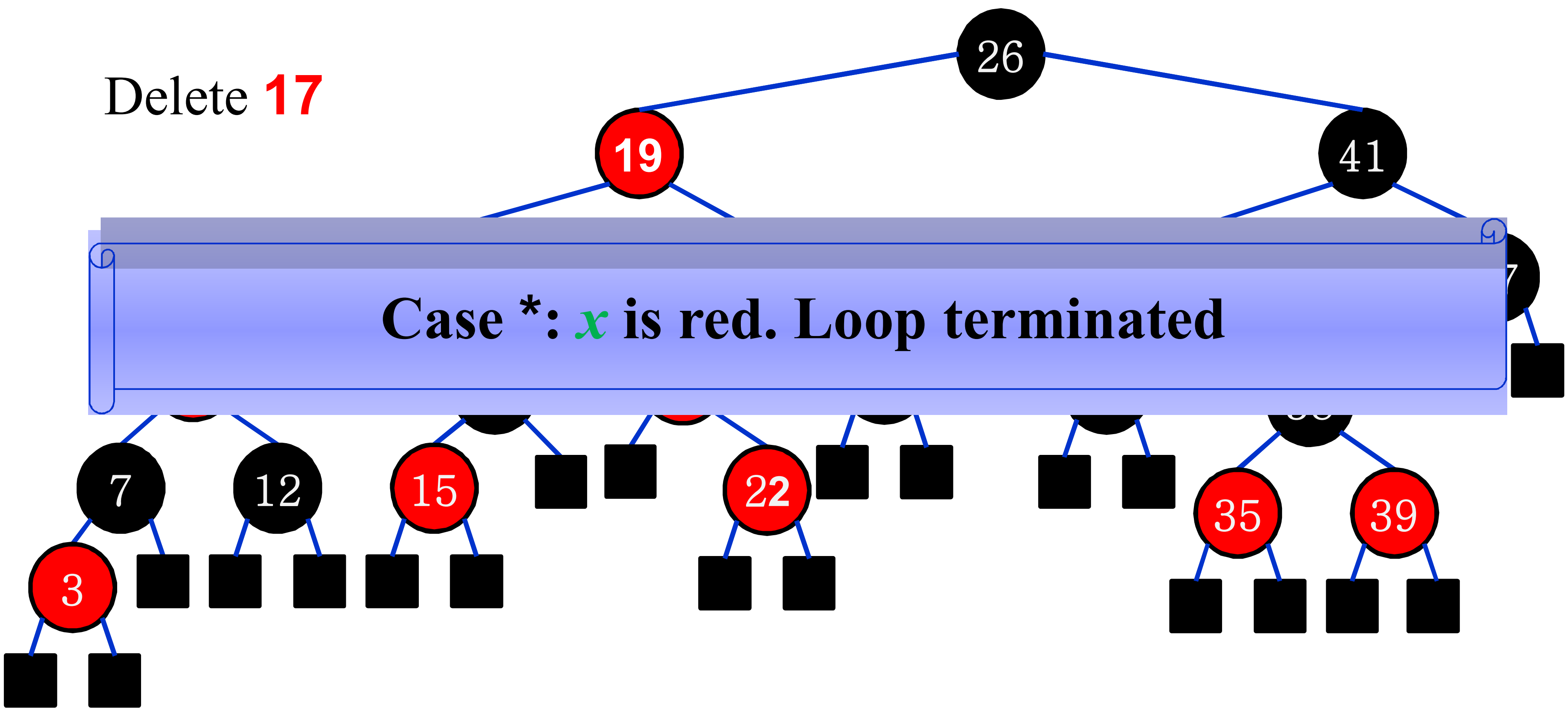
Red-Black Trees: Delete-Fixup

Delete **17**



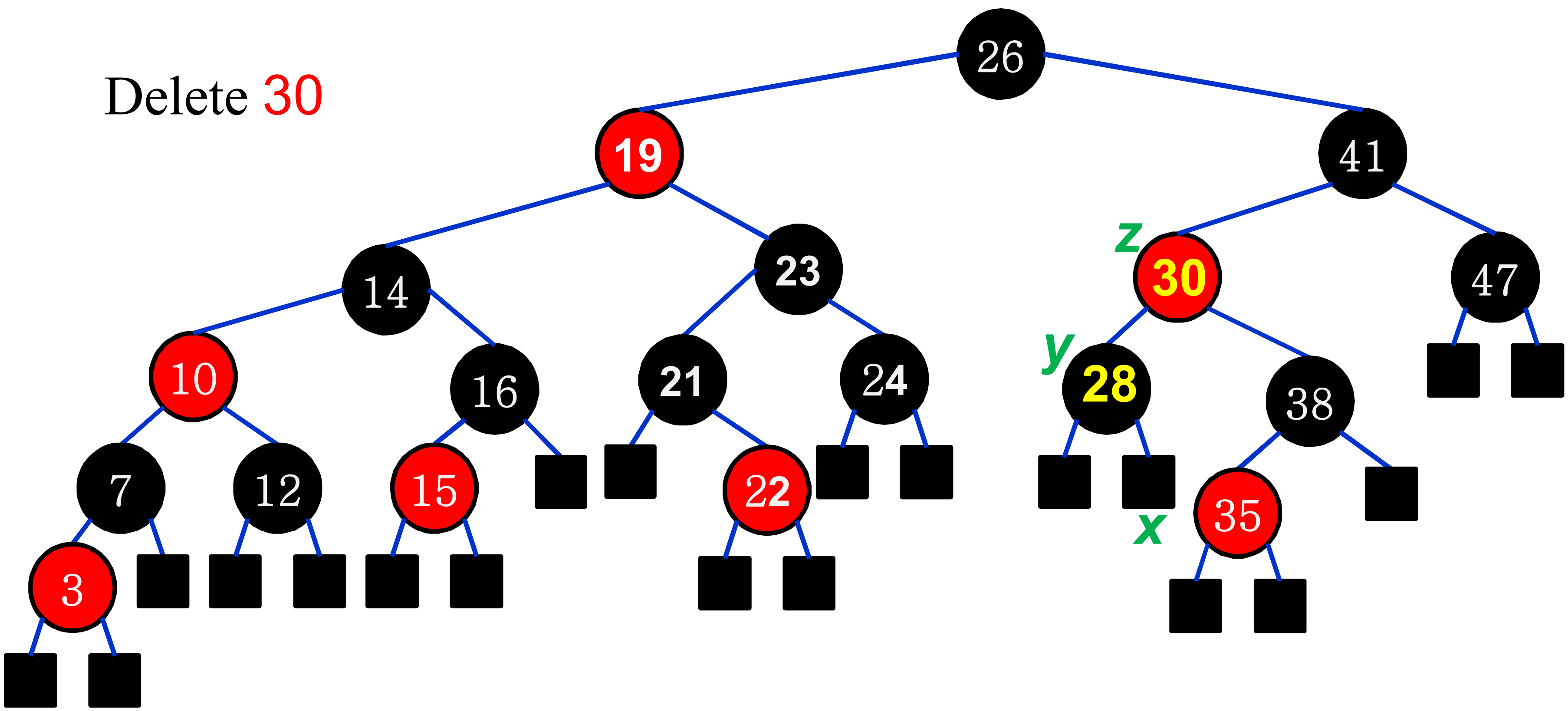
Red-Black Trees: Delete-Fixup

Delete **17**



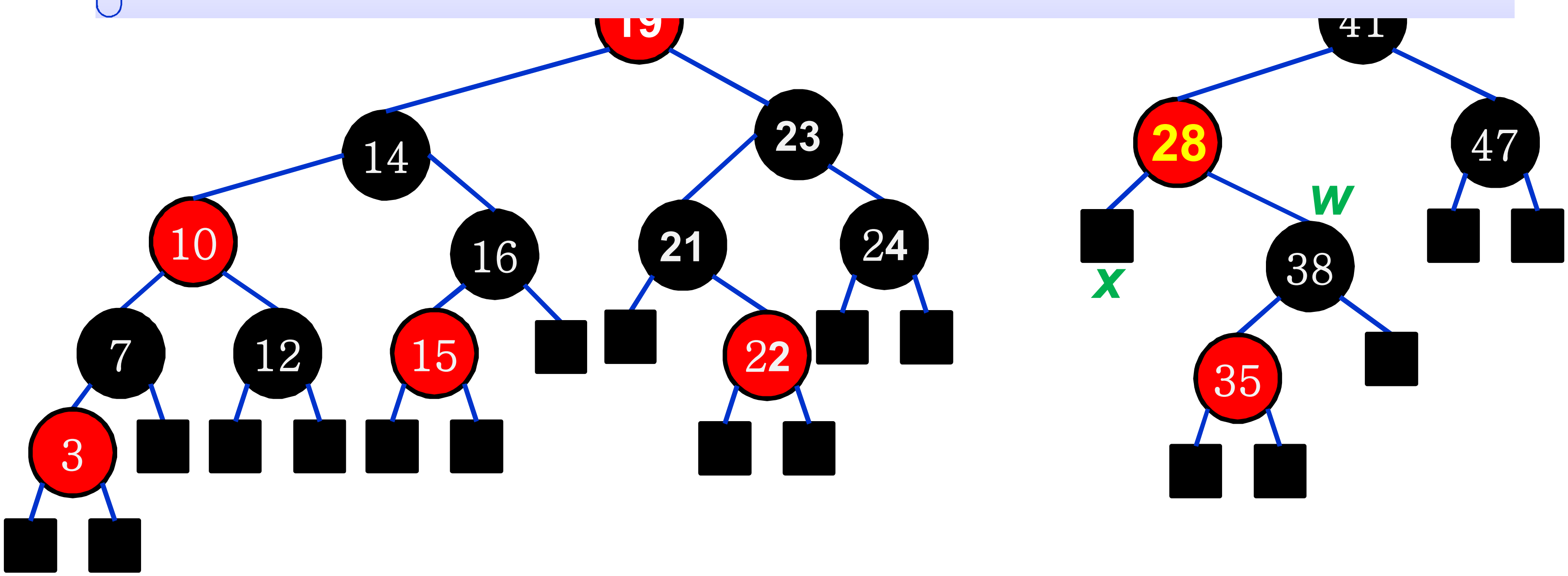
Red-Black Trees: Delete-Fixup

Delete 30



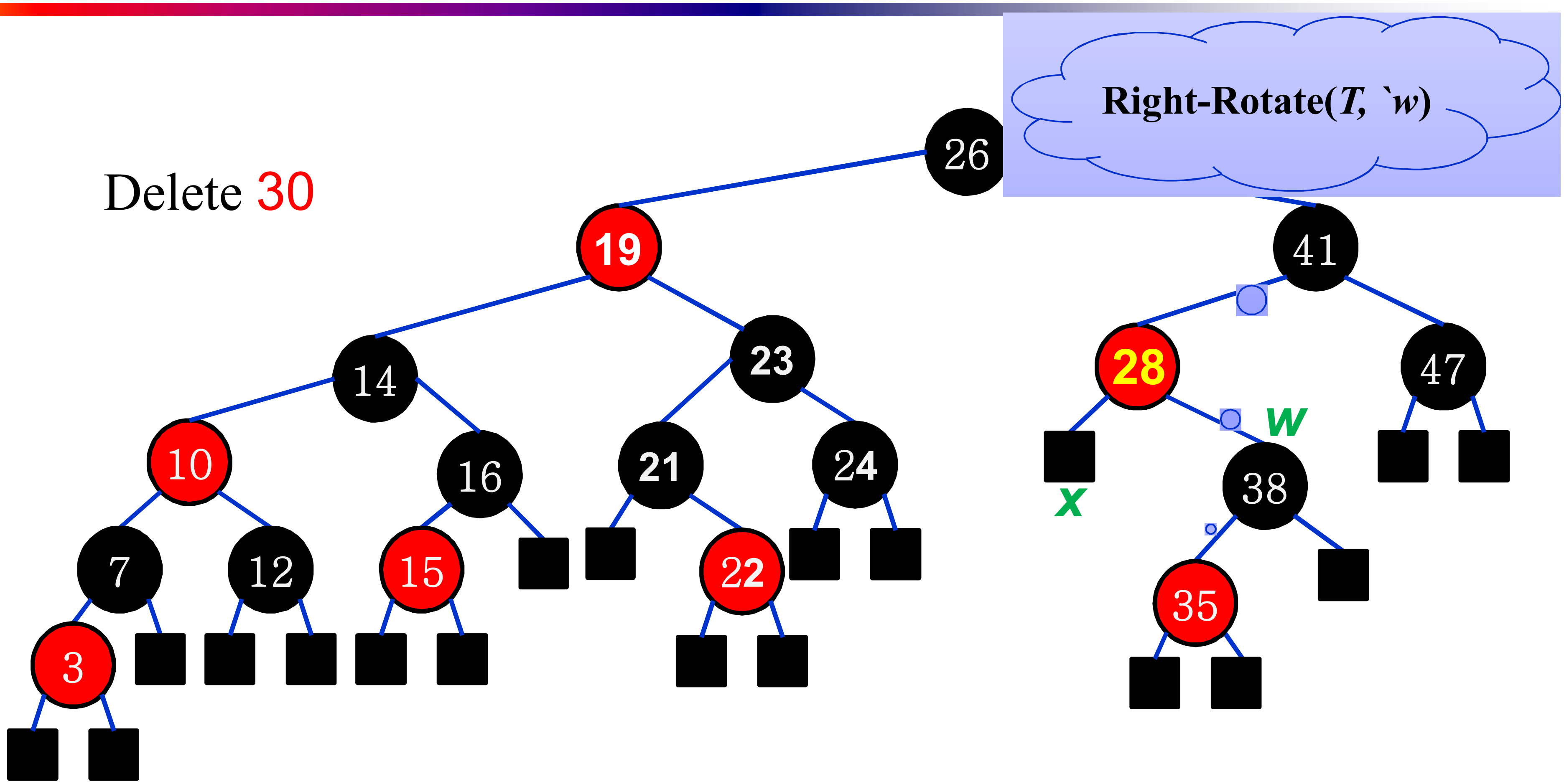
Red-Black Trees: Delete-Fixup

Case III: *x*'s sibling *w* is black and left child of *w* is red and right child of *w* is black



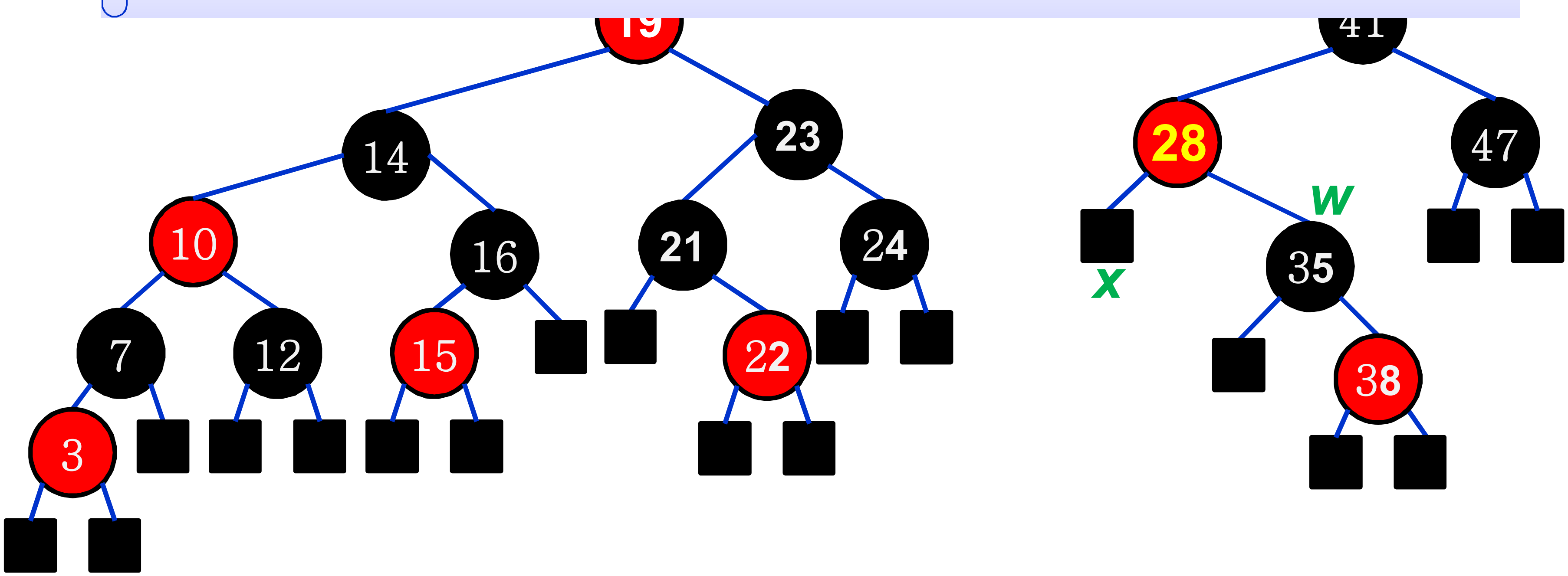
Red-Black Trees: Delete-Fixup

Delete 30



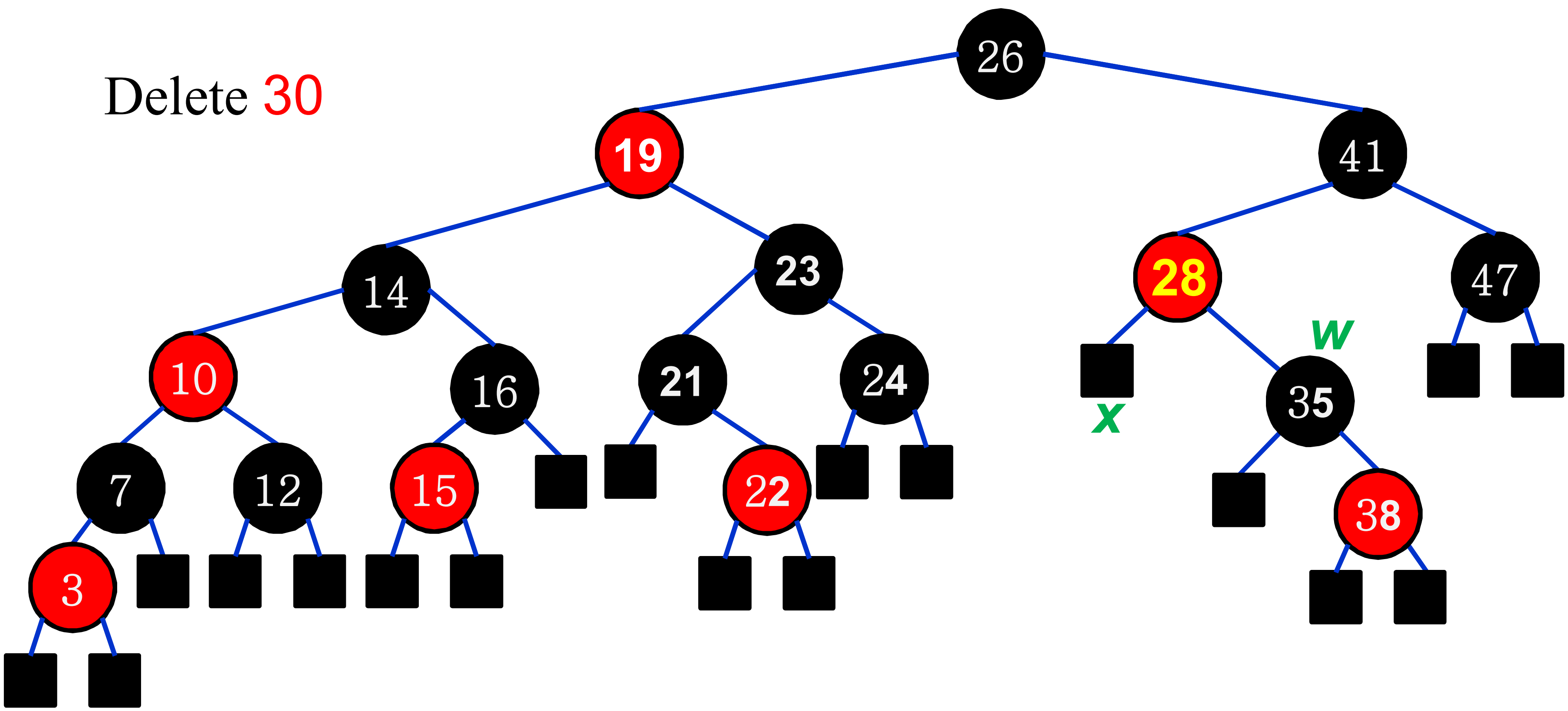
Red-Black Trees: Delete-Fixup

Case IV: *x*'s sibling *w* is black and right child of *w* is red



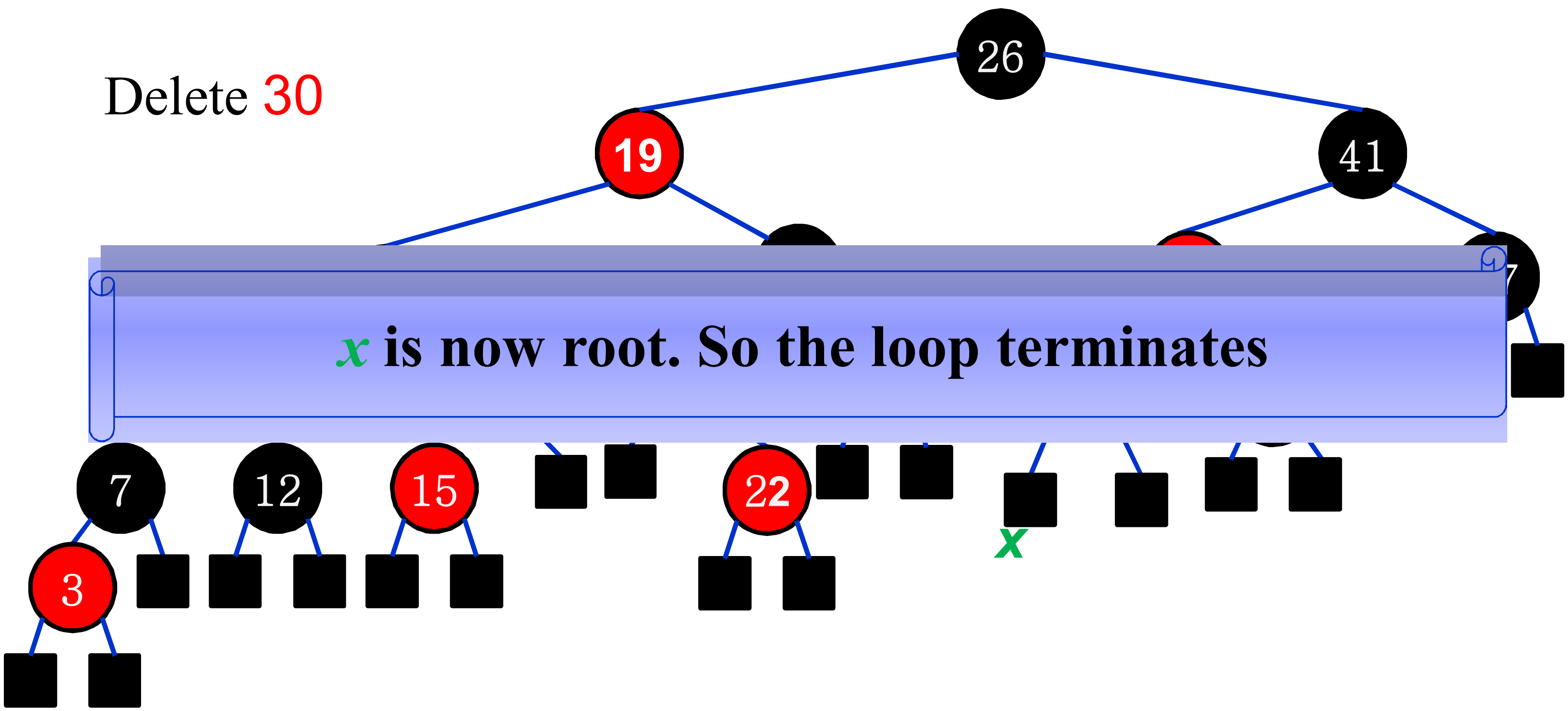
Red-Black Trees: Delete-Fixup

Delete 30



Red-Black Trees: Delete-Fixup

Delete 30

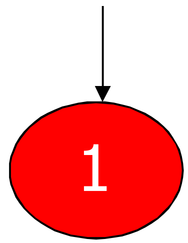


Red-Black Trees: Delete-Fixup

```
RB-Delete-Fixup(T, x)
1  while color[x] = = black and x ≠ root[T] do
2      if x = = left[p[x]] then
3          w = right[p[x]]                          \\ set w as x's sibling
4          if color[w] = = red then                 \\ case I
5              color[w] ← black
6              color[p[x]] ← red
7              Left-Rotate(T, P[x])
8              w← right[p[x]]
9          if color[left[w]]= = black and color[right[w]]= black then     \\ case II
10             color[w]← red
11             x ← p[x]
12         else
13             if color[right[w]] = = black then    \\ case III
14                 color[left[w]] ← black
15                 color[w] ← red
16                 Right-Rotate(T, w)
17                 w ← right[p[x]]
18             color[w]← color[p[x]]                \\ case IV
19             color[p[x]]←black
20             color[right[w]] ←black
21             Left-Rotate(T, P[x])
22             x← root[T]
23     else same as then clause of line 2 with left and right exchanged.
24 color[x] ← black
```

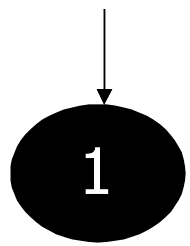
Example of Inserting Sorted Numbers

- 1 2 3 4 5 6 7 8 9

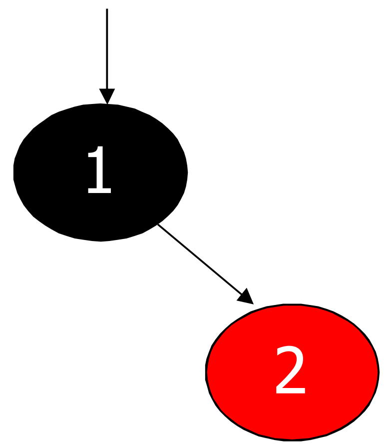


Insert 1:

- A leaf, so red.
- Realize it is root, so recolor to black.



Insert 2



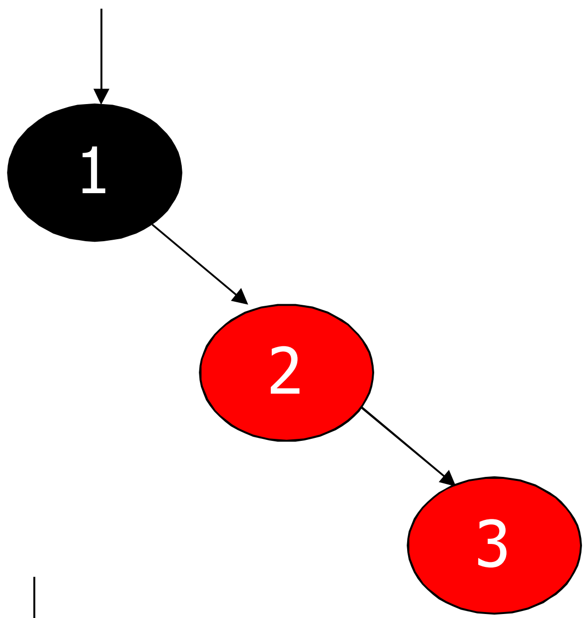
Insert 2:

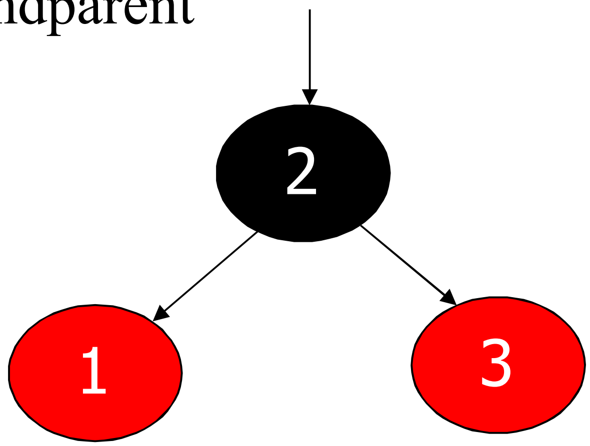
- Make 2 red.
- Parent is black so done.

Insert 3

Insert 3:

- Parent is red.
- Parent's sibling is black (*nil*).
- So it is Case 3:
  - Color 2 by Black, and 1 by Red
  - Left Rotate parent and grandparent



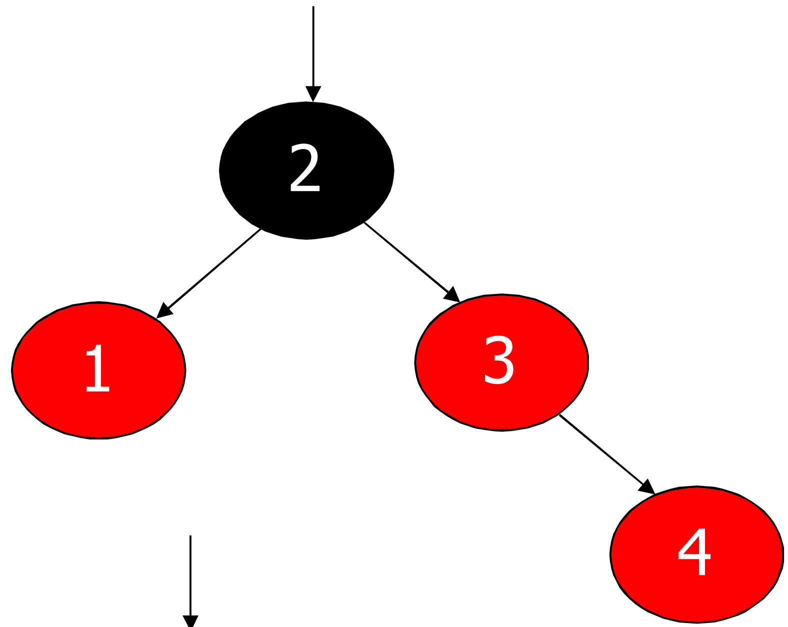


Insert 4

Insert 4:

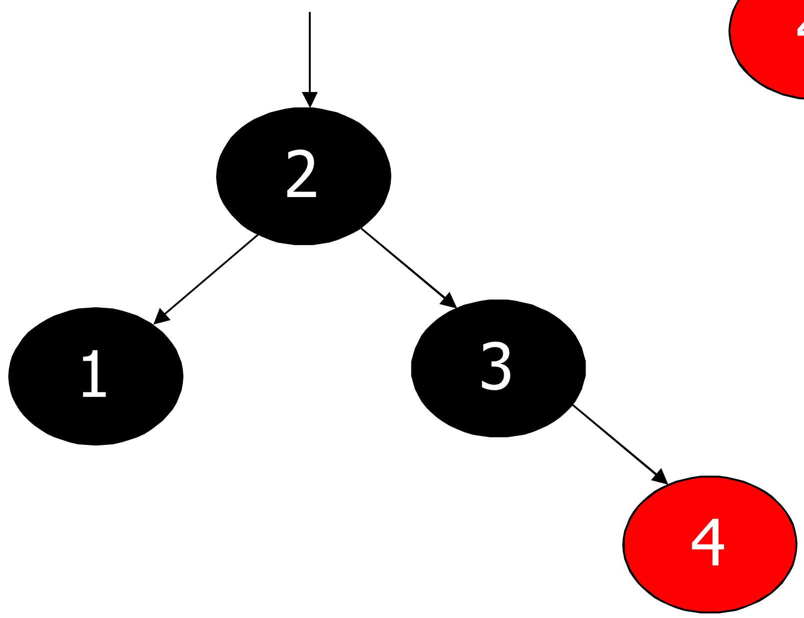
4's uncle 1 is red. So it is Case 1.

- Recolor 2 red and both of its children black.
- Realize 2 is root, so color back to black.

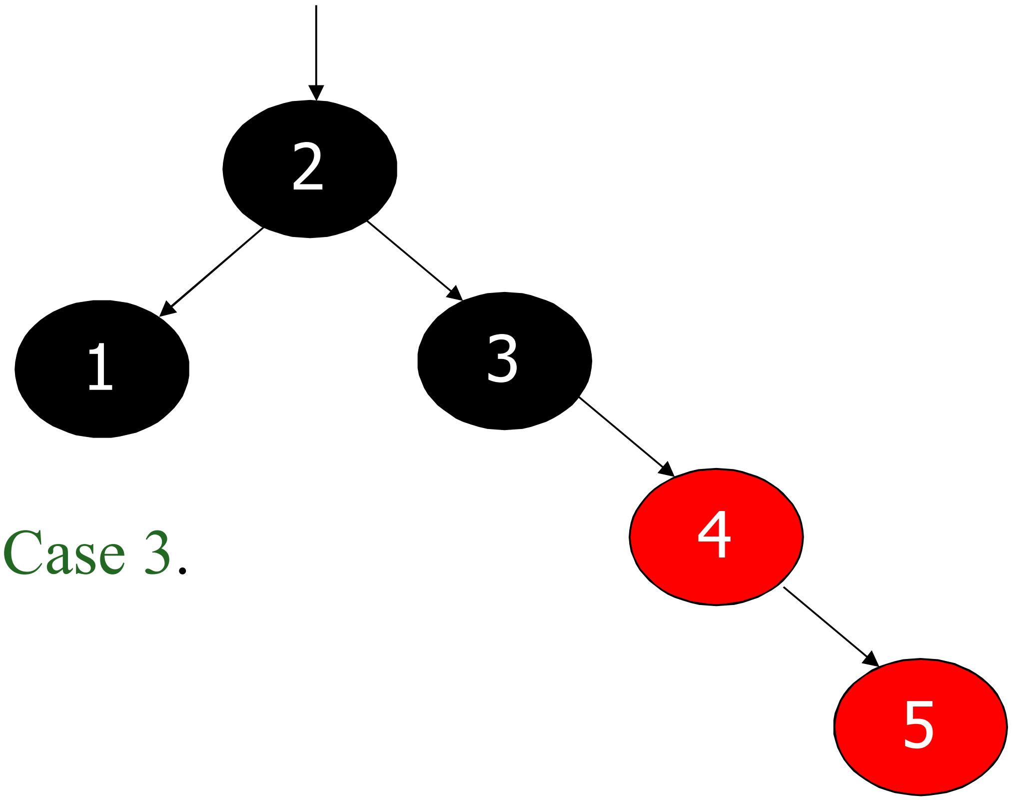


Now parent of 4 is black,

so done.



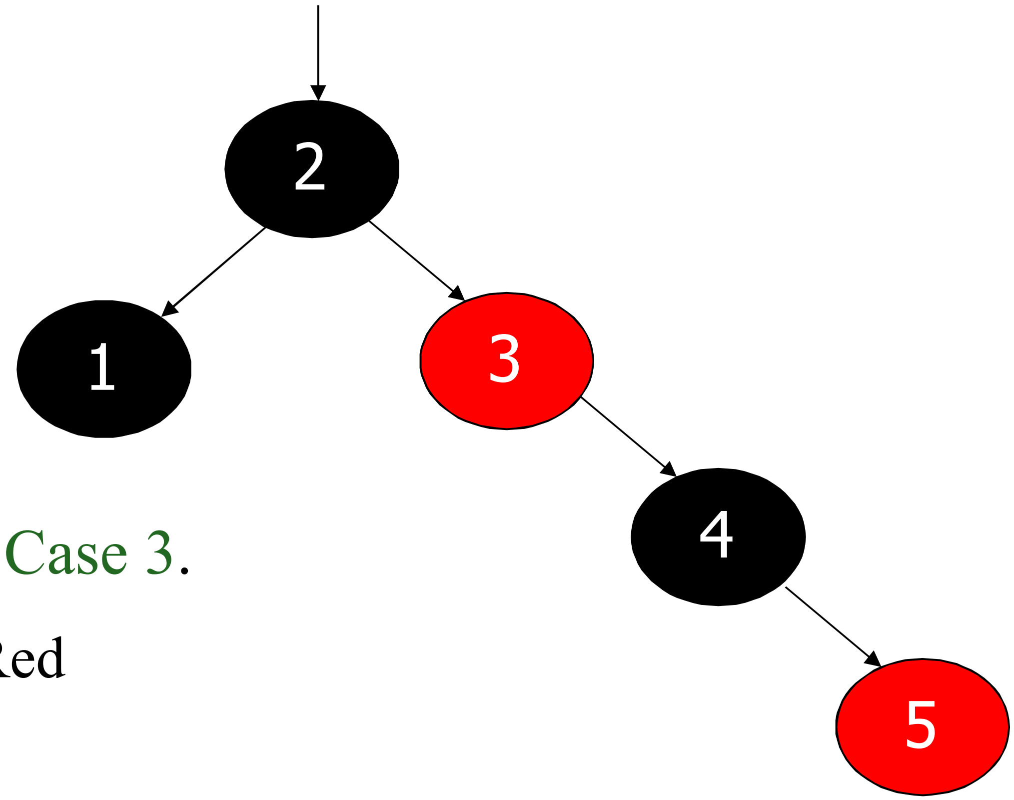
Insert 5



Insert 5:

5's uncle(*nil*) is black. So it is Case 3.

Insert 5

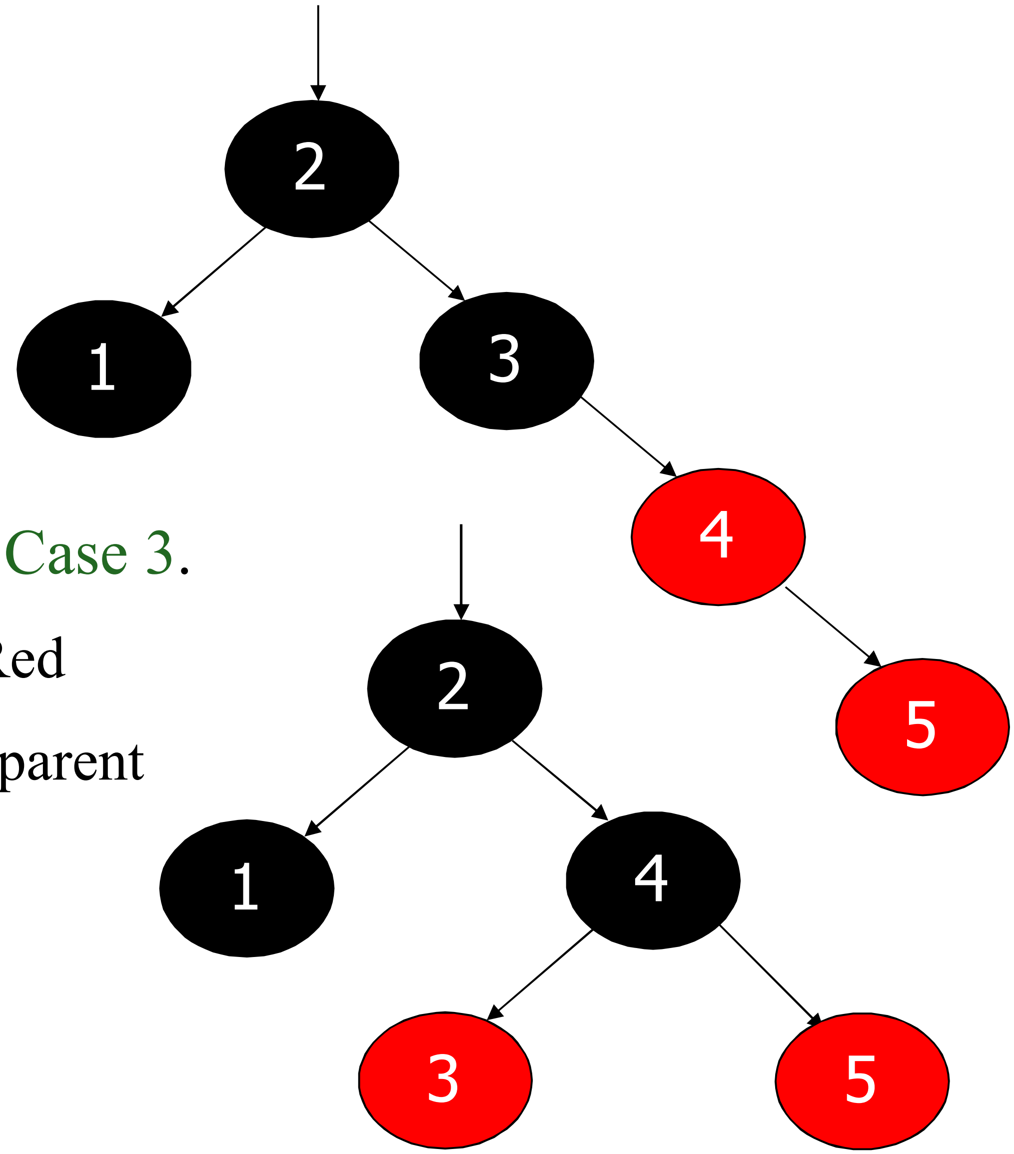


Insert 5:

5’s uncle(*nil*) is black. So it is Case 3.

- Color 4 by Black, and 3 by Red

Insert 5



Insert 5:

5's uncle(*nil*) is black. So it is Case 3.

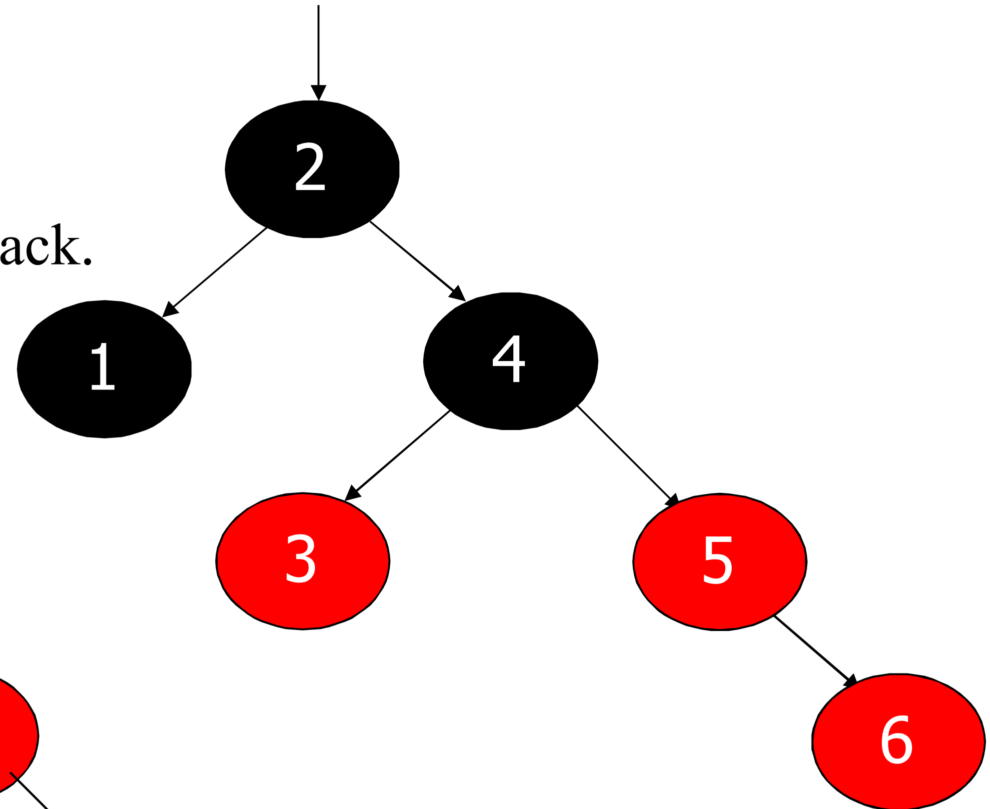
- Color 4 by Black, and 3 by Red
- Left Rotate parent and grandparent

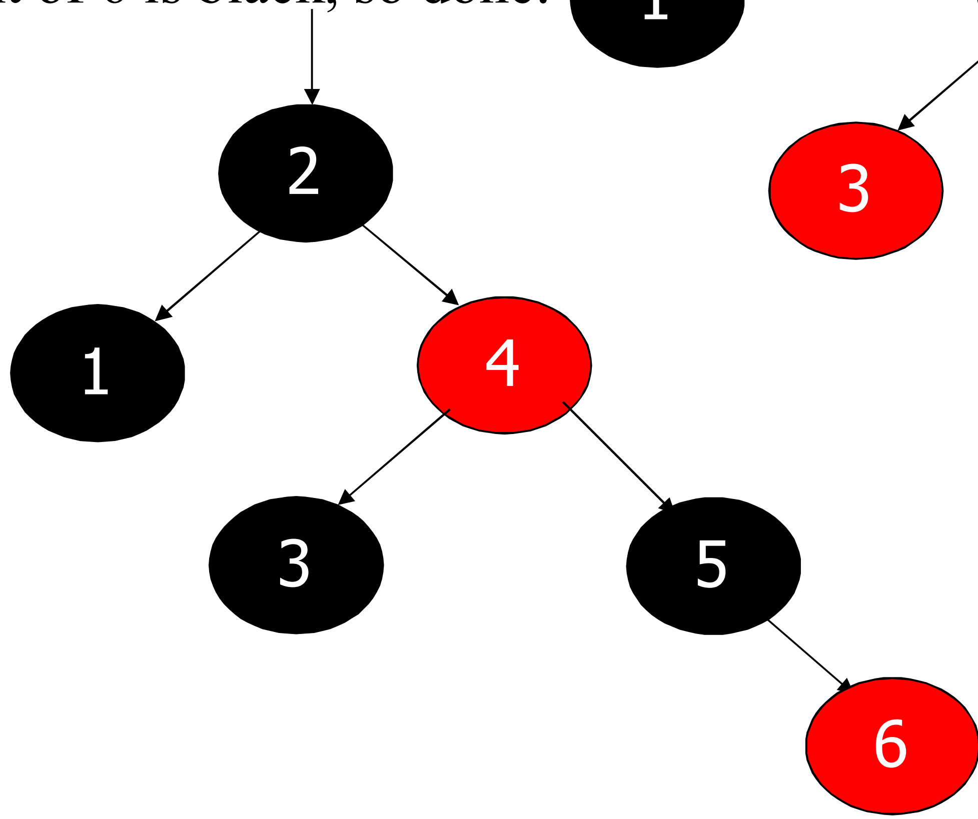
Insert 6

Insert 6:

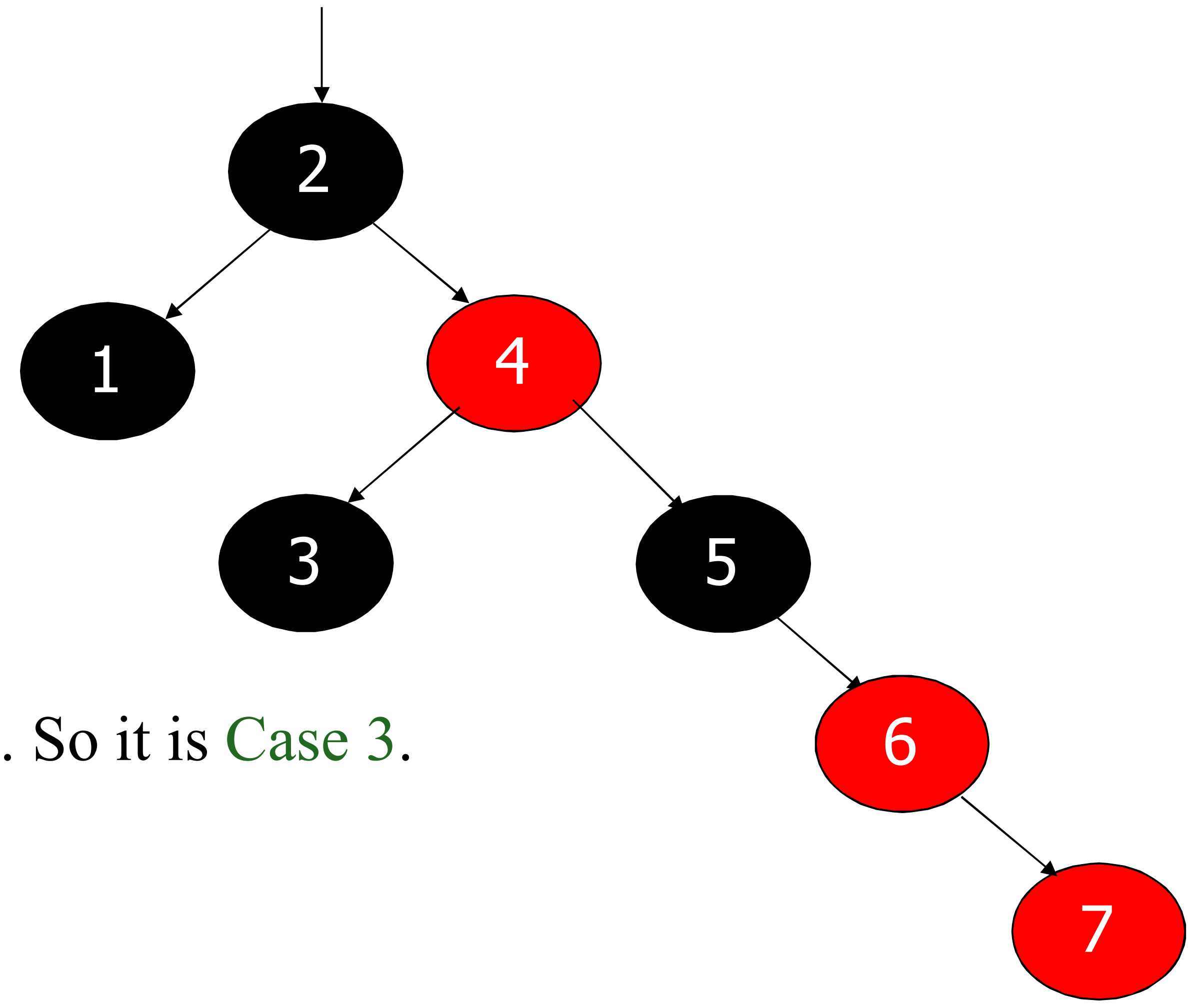
6's uncle 3 is red. So it is Case 1.

- Recolor 4 red and both children black.
- Now parent of 6 is black, so done.





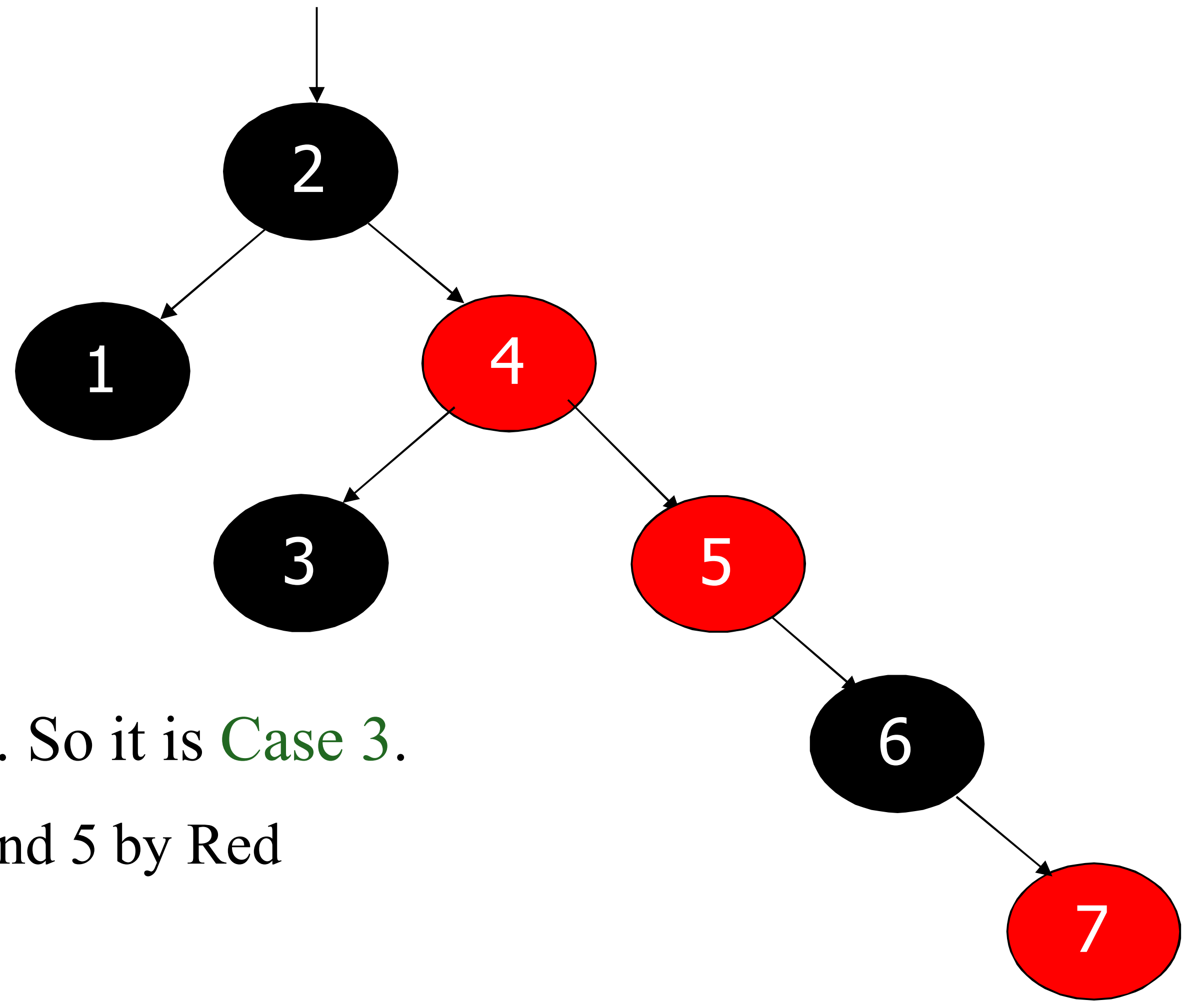
Insert 7



Insert 7:

7's uncle(*nil*) is black. So it is Case 3.

Insert 7

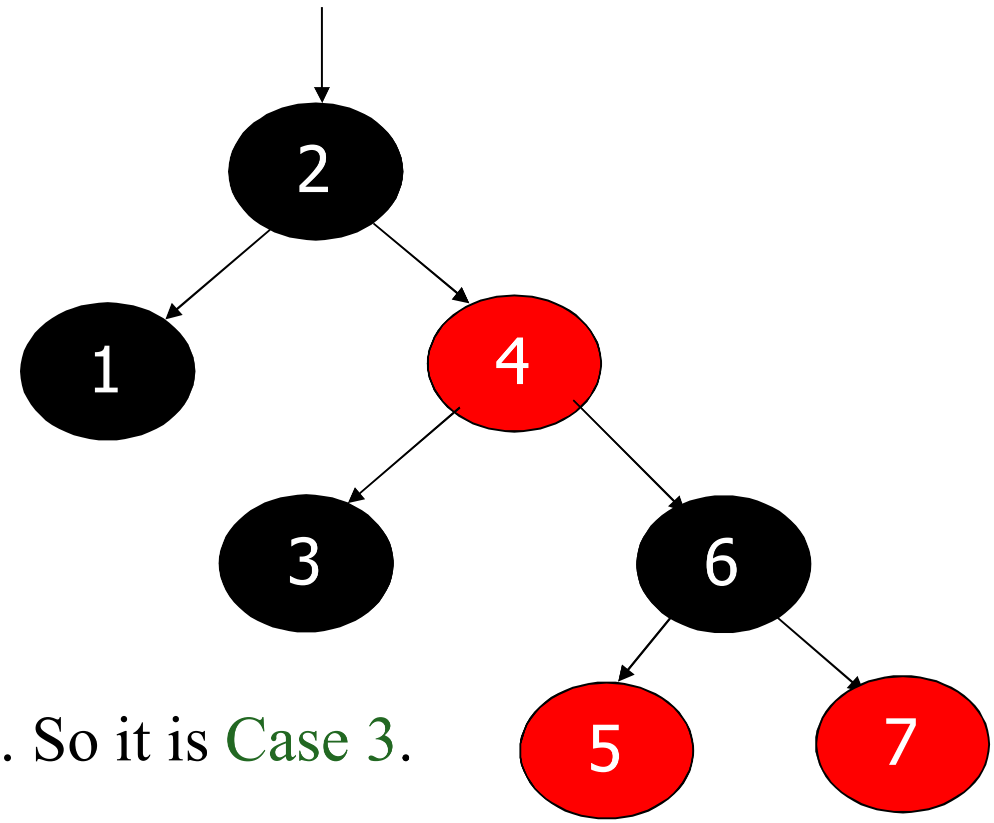


Insert 7:

7's uncle(*nil*) is black. So it is Case 3.

- Color 6 by Black, and 5 by Red

Insert 7

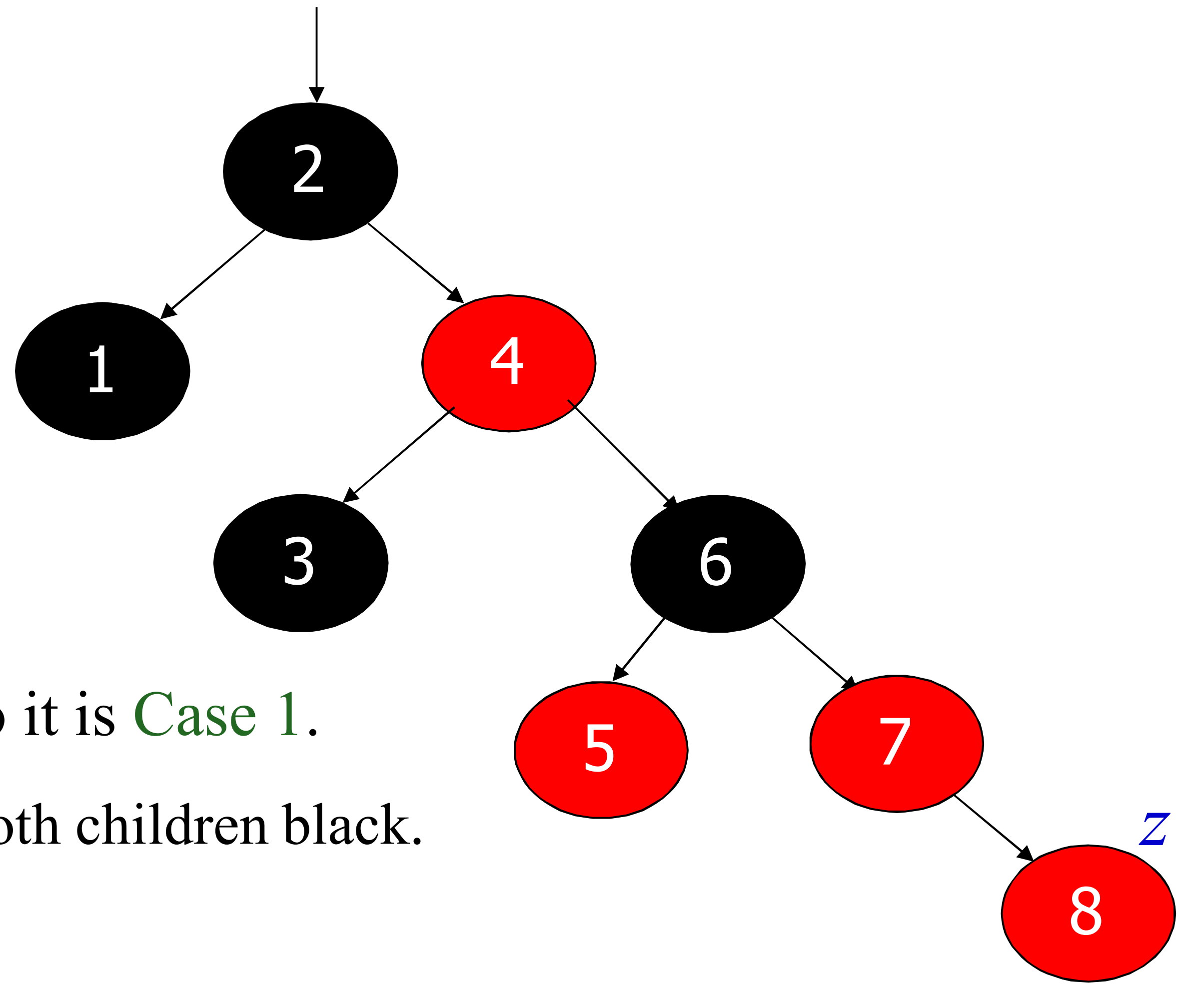


Insert 7:

7's uncle(*nil*) is black. So it is Case 3.

- Color 6 by Black, and 5 by Red
- Left Rotate parent and grandparent

Insert 8

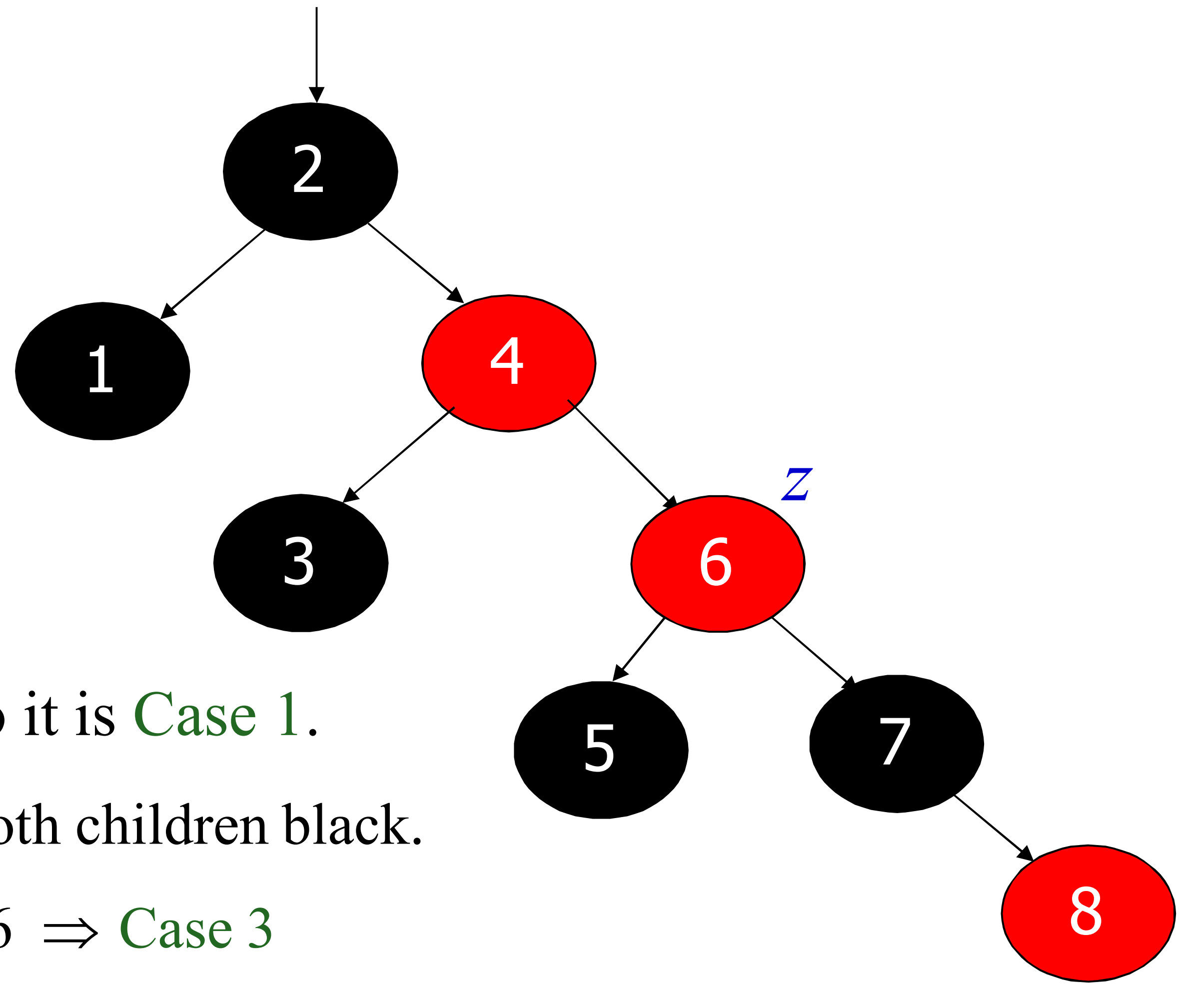


Insert 8:

8’s uncle 5 is red. So it is Case 1.

- Recolor 6 red and both children black.

Insert 8

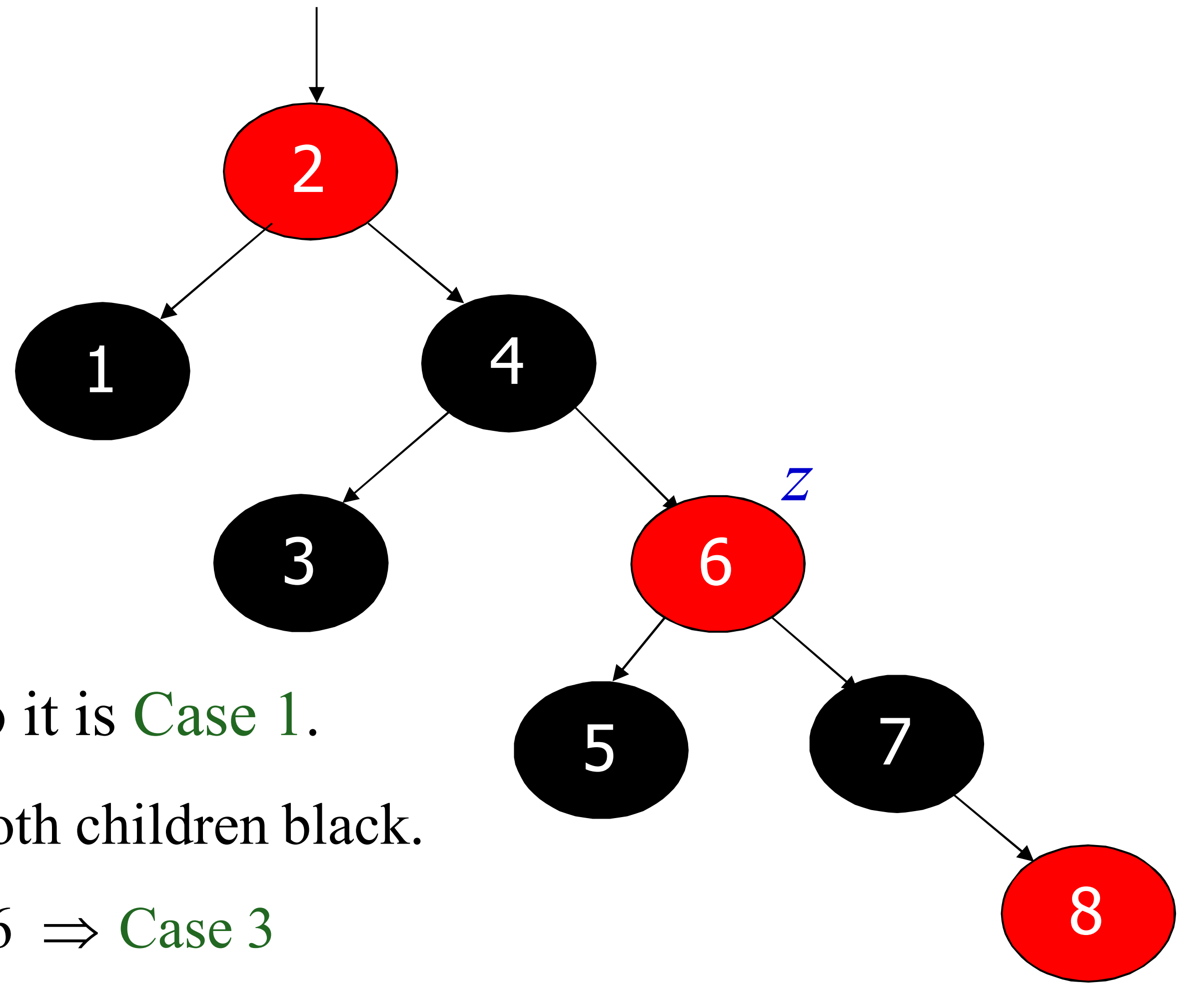


Insert 8:

8's uncle 5 is red. So it is Case 1.

- Recolor 6 red and both children black.
- Now new z is node 6 $\Rightarrow$ Case 3

Insert 8

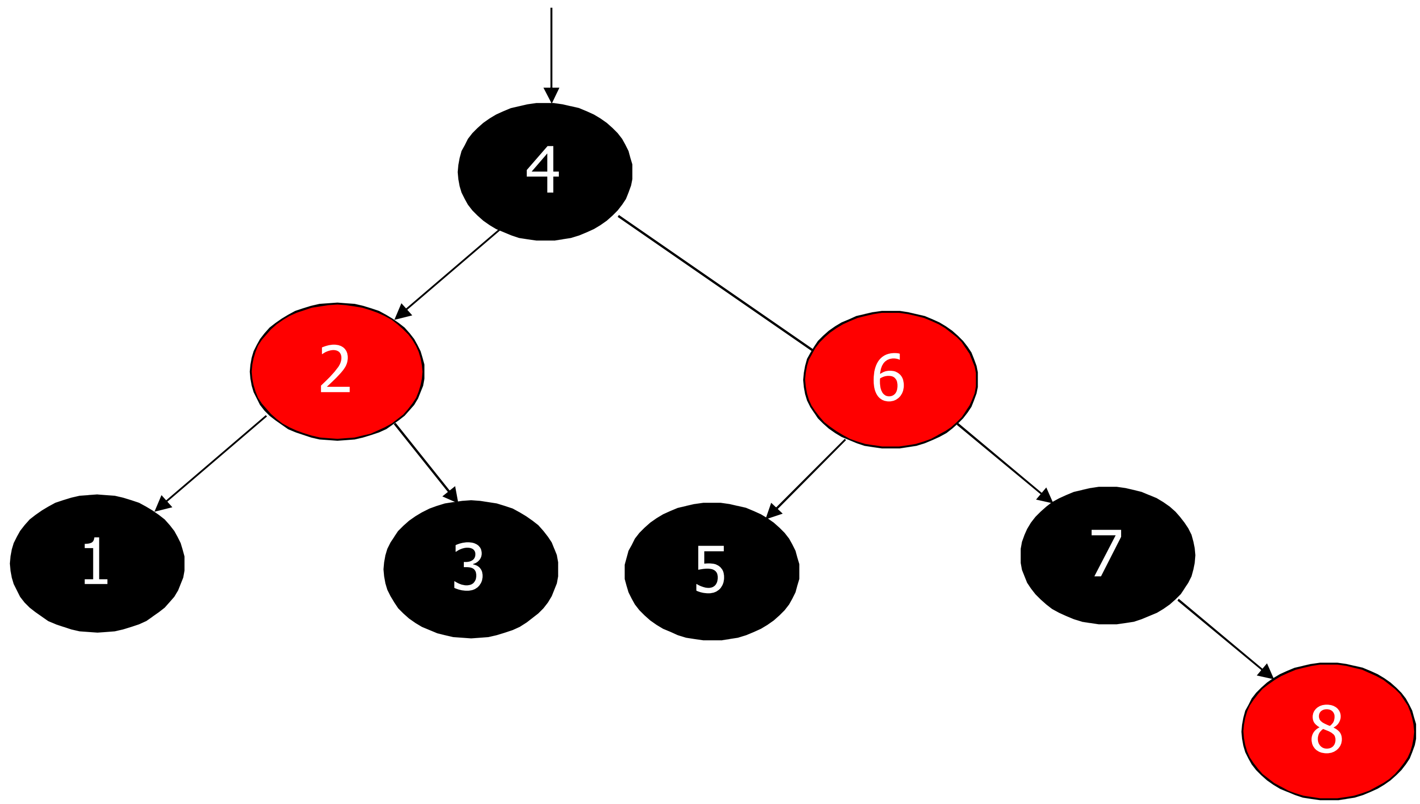


Insert 8:

8's uncle 5 is red. So it is Case 1.

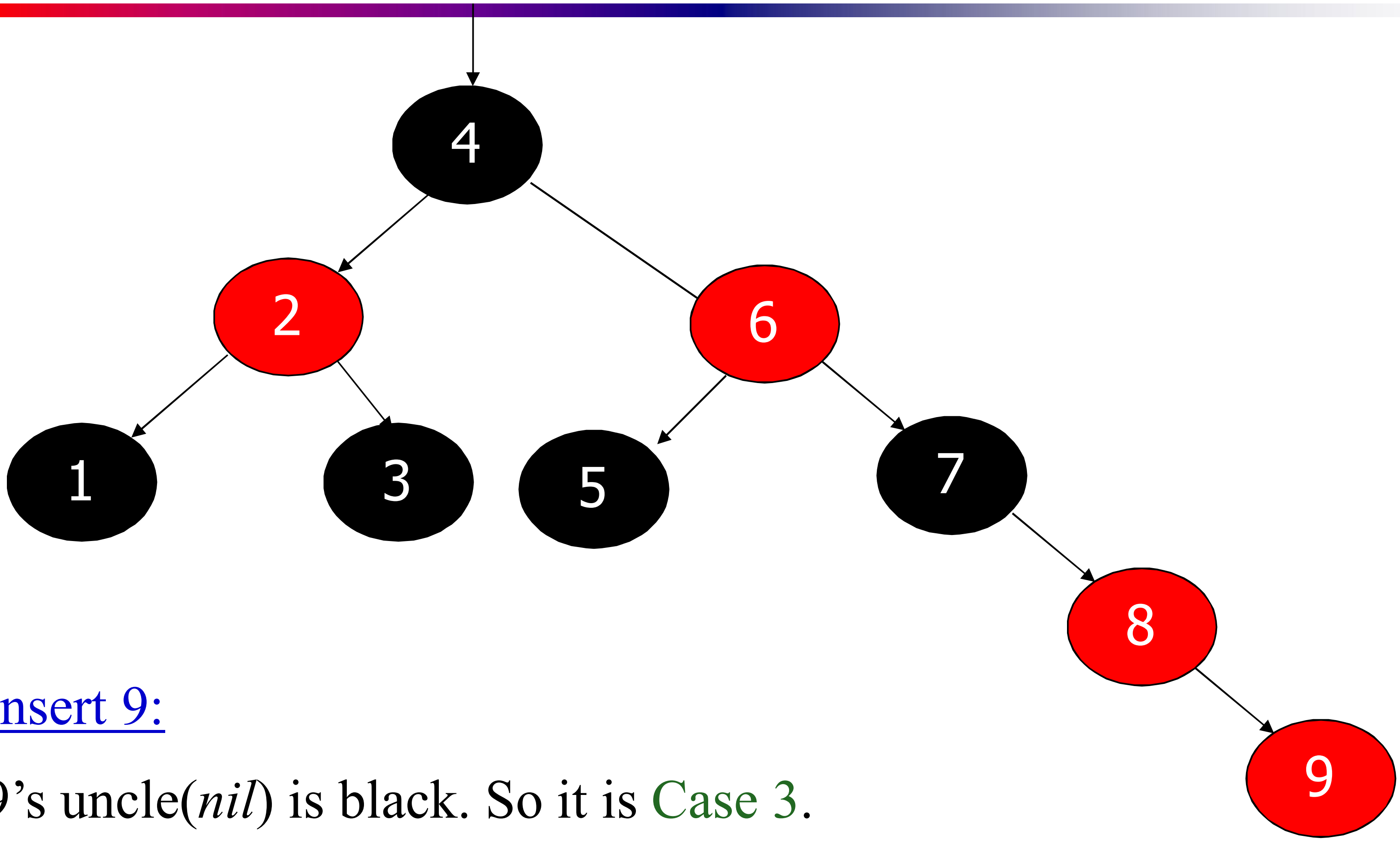
- Recolor 6 red and both children black.
- Now new z is node 6 $\Rightarrow$ Case 3
  - Color 4 by Black, and 2 by Red

Insert 8



- Now new z is node 6 $\Rightarrow$ Case 3
  - Color 4 by Black, and 2 by Red
  - Left Rotate parent and grandparent

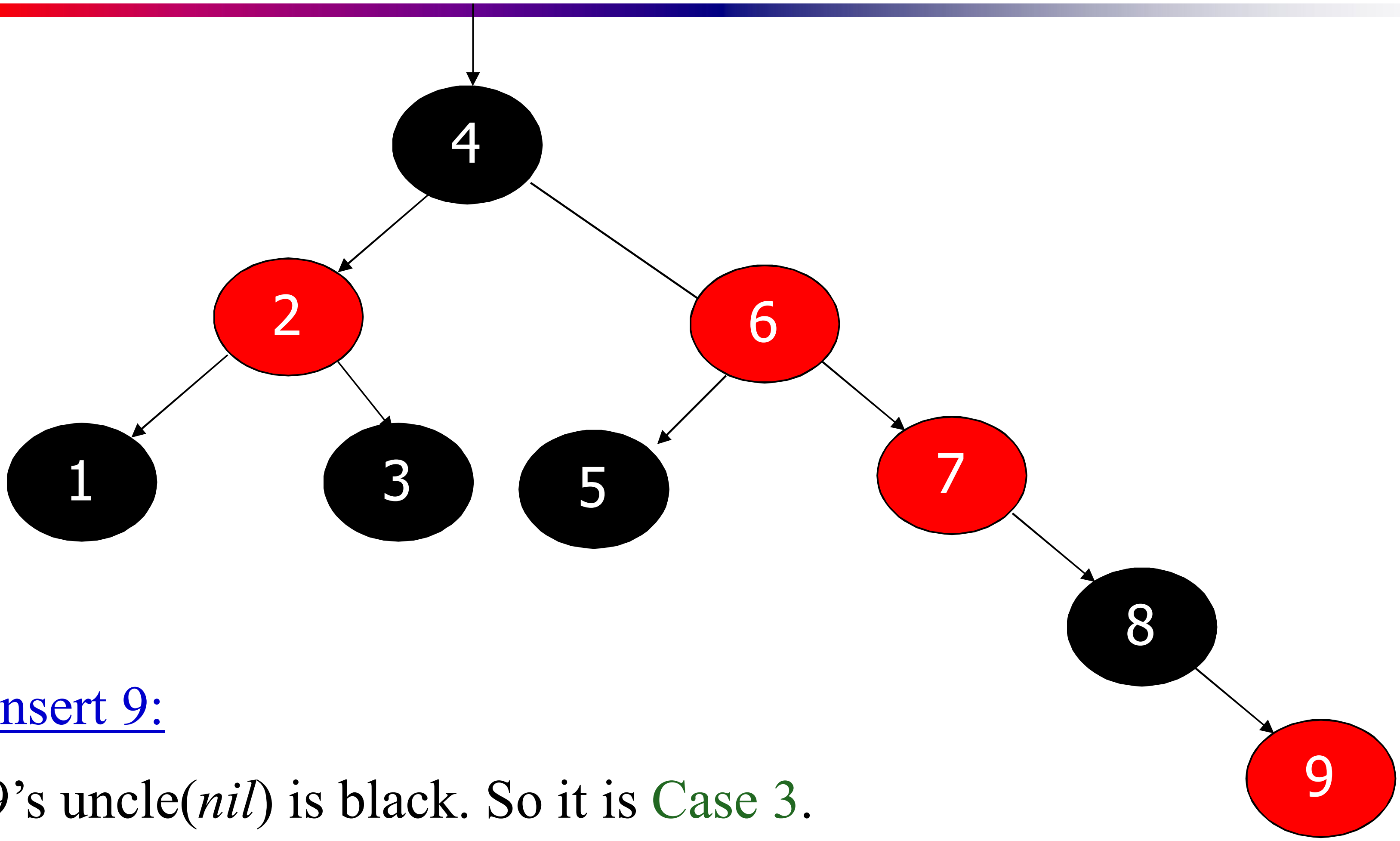
Insert 9



Insert 9:

9’s uncle(*nil*) is black. So it is Case 3.

Insert 9

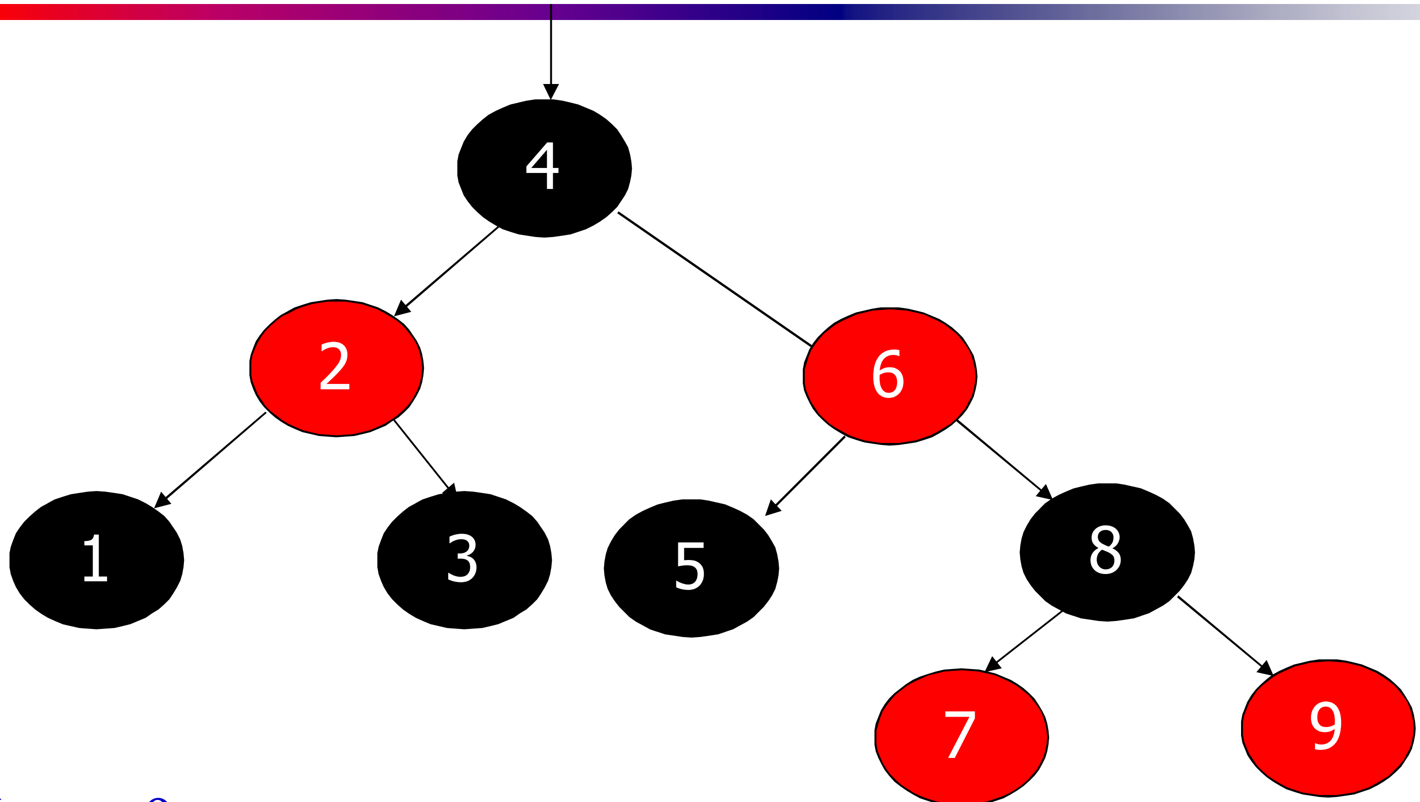


Insert 9:

9's uncle(*nil*) is black. So it is Case 3.

- Color 8 by Black, and 7 by Red

Insert 9



Insert 9:

9's uncle(*nil*) is black. So it is Case 3.

- Color 8 by Black, and 7 by Red
- Left Rotate parent and grandparent

Problems

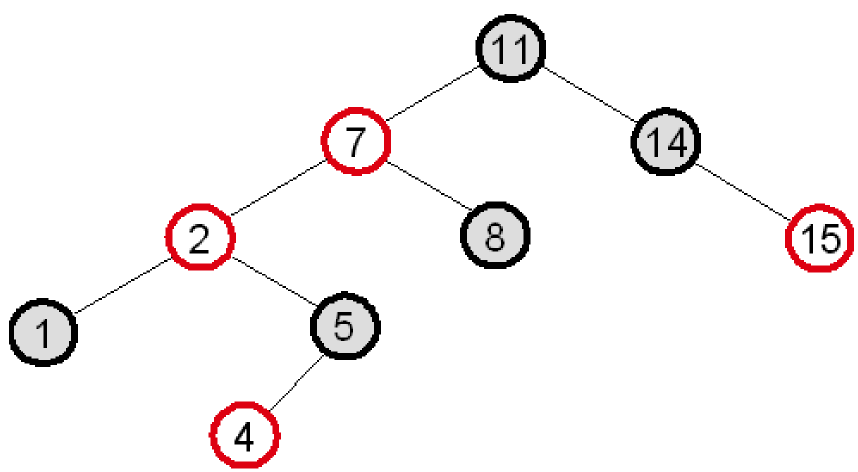
- What is the ratio between the longest path and the shortest path in a red-black tree?

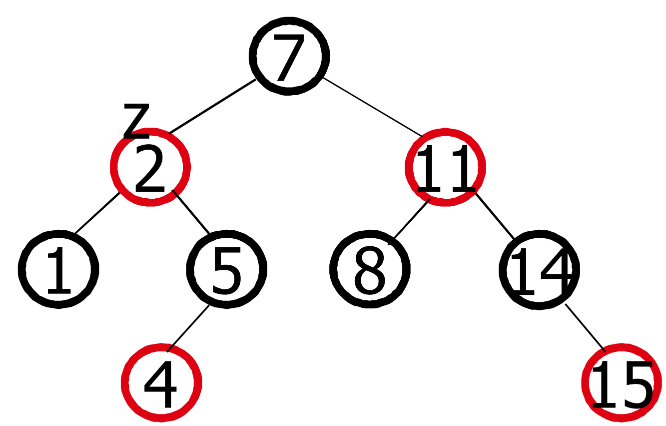
  - The shortest path is at least bh(root)
  - The longest path is equal to h(root)
  - We know that $h(root) \leq 2bh(root)$

  - Therefore, the ratio is ≤ 2

Problems

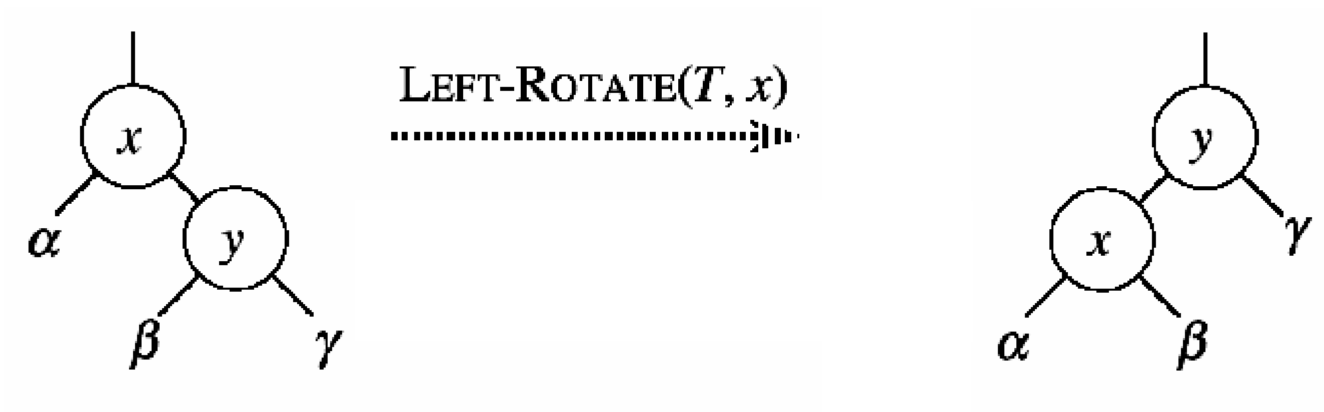
- What red-black tree property is violated in the tree below? How would you restore the red-black tree property in this case?
  - Property violated: if a node is red, both its children are black
  - Fixup: color 7 black, 11 red, then right-rotate around 11





Problems

- Let a, b, c be arbitrary nodes in subtrees α, β, γ in the tree below. How do the depths of a, b, c change when a left rotation is performed on node x?
  - a: increases by 1
  - b: stays the same
  - c: decreases by 1



Problems

- When we insert a node into a red-black tree, we initially set the color of the new node to red. Why didn't we choose to set the color to black?

- **(Exercise 13.4-7, page 294)** Would inserting a new node to a red-black tree and then immediately deleting it, change the tree?