



UMC 201: Data Structures & Algorithms

Lecture 11: Red-Black trees: Deletion

Indian Institute of Science

Recap

- In the last lecture, we discussed Red-Black (RB) trees and showed how the dynamic-set operations **Search**, **Minimum**, **Maximum**, **Successor**, **Predecessor**, and **Insert** can be implemented so that each one runs in $O(\log n)$ time on a RB tree with n nodes.
- We saw that the red-black properties imply that the height of a RB tree is $O(\log n)$.

Recap

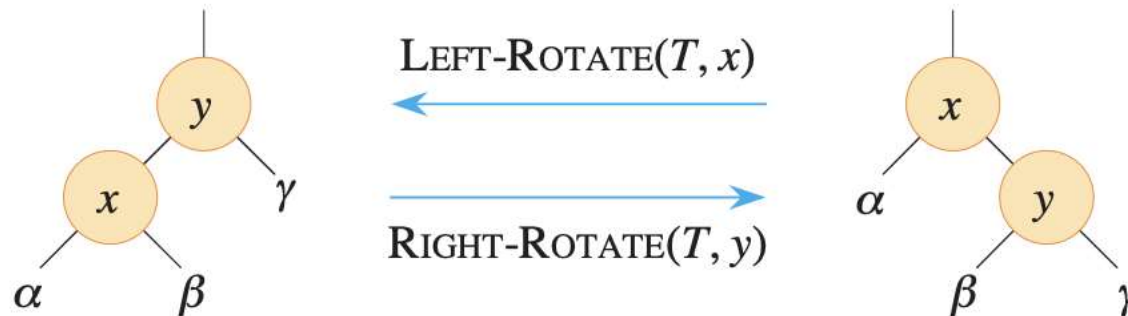
- In the last lecture, we discussed Red-Black (RB) trees and showed how the dynamic-set operations **Search**, **Minimum**, **Maximum**, **Successor**, **Predecessor**, and **Insert** can be implemented so that each one runs in $O(\log n)$ time on a RB tree with n nodes.
- We saw that the red-black properties imply that the height of a RB tree is $O(\log n)$.
- In today's lecture, we'll see how to do **Deletion** in a RB tree also in $O(\log n)$ time.

Recap: Red-Black properties

- A **red-black tree** is a BST that satisfies the following five red-black properties:
 1. Every node is either **red** or **black**.
 2. The root is **black**.
 3. Every leaf (NIL) 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 contain the same number of **black** nodes.

Recap: Rotations

- The pointer structure changes through **rotation**, which is a local operation in a BST that preserves the binary-search-tree property. There are two kinds of rotations: **left rotation** and **right rotation**.



- For left rotation, assume that $x.\text{right}$ is not $T.\text{nil}$. Similarly, for right rotation, assume $y.\text{left}$ is not $T.\text{nil}$.

Recap: Transplant

- The procedure below replaces the subtree rooted at node u with the subtree rooted at node v .

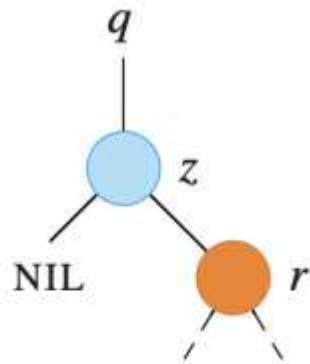
```
TRANSPLANT( $T, u, v$ )  
1  if  $u.p == \text{NIL}$   
2       $T.root = v$   
3  elseif  $u == u.p.left$   
4       $u.p.left = v$   
5  else  $u.p.right = v$   
6  if  $v \neq \text{NIL}$   
7       $v.p = u.p$ 
```

- u 's parent ends up having v as the appropriate child.
- Node u 's parent becomes node v 's parent.
- Time complexity is $O(1)$.

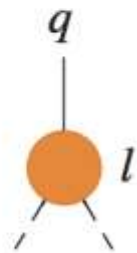
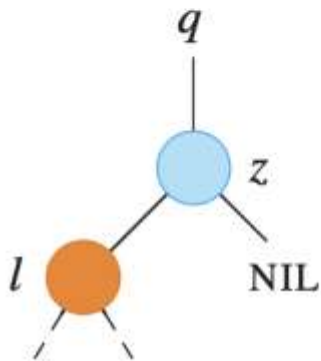
Recap: BST Deletion

- The overall strategy for deleting a node z from a BST has two cases:
 - Case 1. z has at most one child.
 - Case 2. z has two children.
 - ❖ Case 2a. The successor of z is **not** z 's right child.
 - ❖ Case 2b. The successor of z is z 's right child.
- Intuitively, in Case 2, we should “replace” z by the successor (or the predecessor) of z in the tree.

Recap: BST Deletion Case I



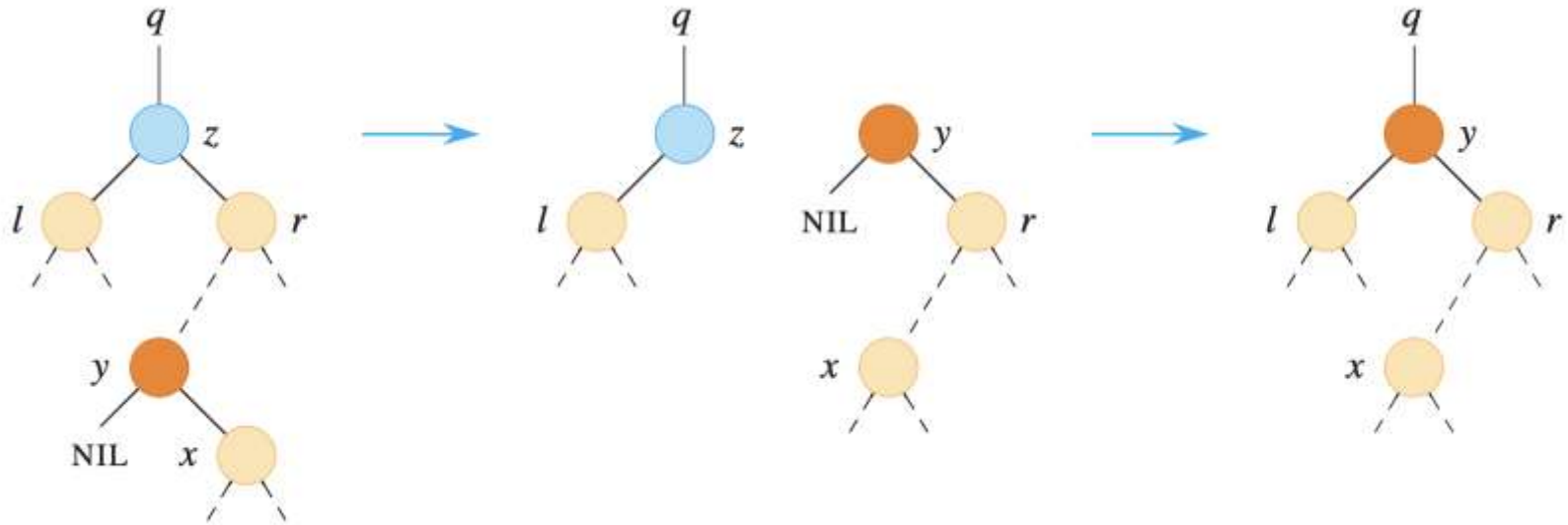
$\text{Transplant}(z, z.\text{right})^*$



$\text{Transplant}(z, z.\text{left})$

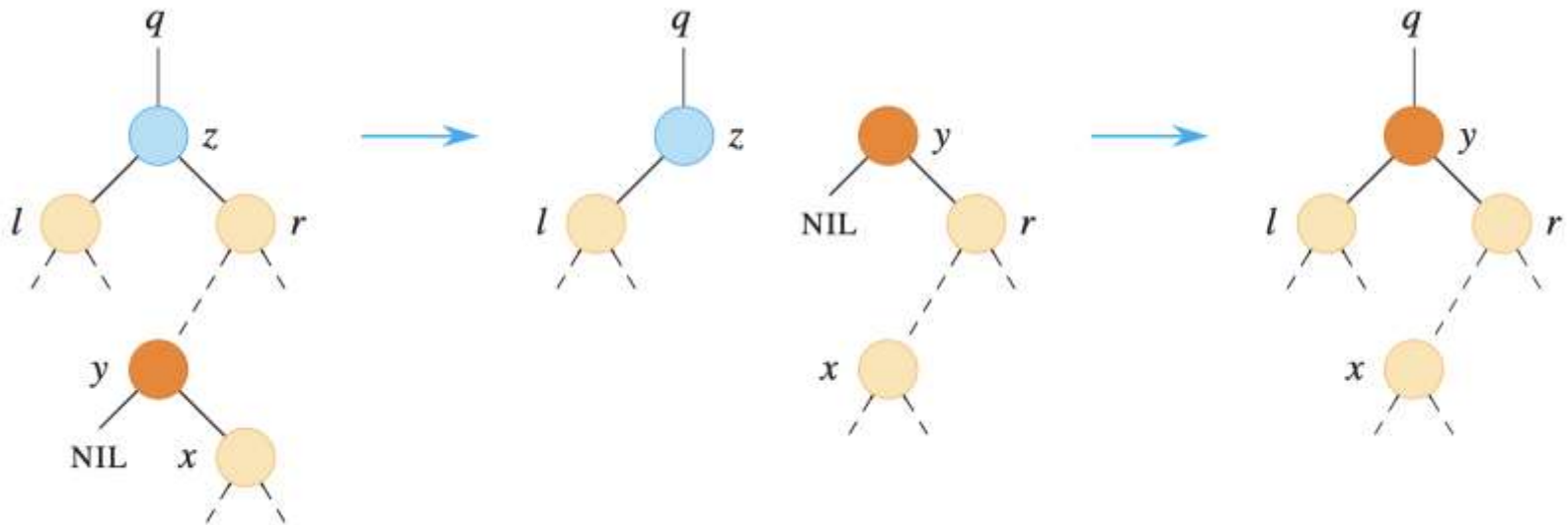
*skipping the argument T in Transplant for simplicity

Recap: BST Deletion Case 2a



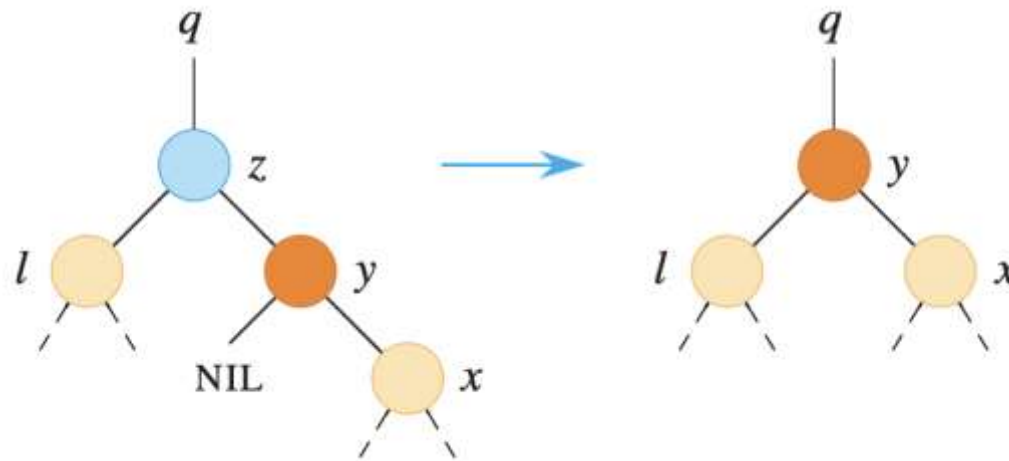
- In this case y , the successor of z , is **not** z 's right child.

Recap: BST Deletion Case 2a



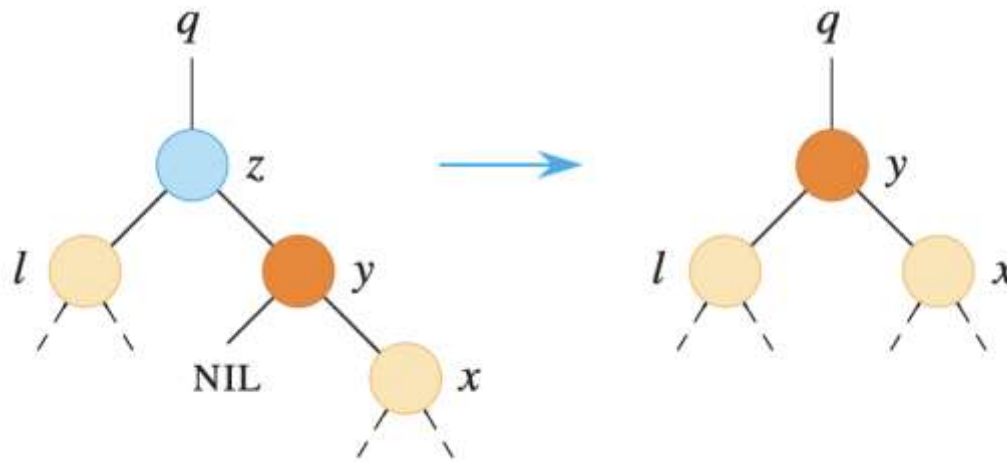
- First, we prepare a subtree rooted at y as follows:
 - Replace y by its own right child x .
 - Set y to be r 's parent.
- Then, set y to be q 's child and the parent of l .

Recap: BST Deletion Case 2b



- In this case y , the successor of z , is z 's right child.

Recap: BST Deletion Case 2b



- Replace z by y , updating y 's left child to become l .

Recap: BST Deletion Pseudocode

TREE-DELETE(T, z)

```
1  if  $z.left == \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else  $y = \text{TREE-MINIMUM}(z.right)$ 
6      if  $y \neq z.right$ 
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      $y.left.p = y$ 
```

Note. TREE-MINIMUM($z.right$)
returns the successor of z .

Handling Case 1

Handling Case 2a

Handling Case 2b

Red-Black tree: Deletion

RB Transplant

- The procedure for deleting a node from a RB tree is based on the **TREE-DELETE** procedure for BSTs.
- Recall that the **TREE-DELETE** procedure uses the **TRANSPLANT** procedure.

RB Transplant

- The procedure for deleting a node from a RB tree is based on the **TREE-DELETE** procedure for BSTs.
- Recall that the **TREE-DELETE** procedure uses the **TRANSPLANT** procedure.
- First, we need to customize the **TRANSPLANT** procedure a bit so that it applies to a RB tree.
- We'll do this next in the **RB-TRANSPLANT** procedure.

TRANSPLANT vs. RB-TRANSPLANT

TRANSPLANT(T, u, v)

```
1  if  $u.p == \text{NIL}$ 
2       $T.root = v$ 
3  elseif  $u == u.p.left$ 
4       $u.p.left = v$ 
5  else  $u.p.right = v$ 
6  if  $v \neq \text{NIL}$ 
7       $v.p = u.p$ 
```

RB-TRANSPLANT(T, u, v)

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

- The main difference between the two procedures is that in Steps 6-7, even if $v = T.nil$, we assign $v.p = u.p$. We'll take advantage of this later in the code.

TRANSPLANT vs. RB-TRANSPLANT

TRANSPLANT(T, u, v)

```
1  if  $u.p == \text{NIL}$ 
2       $T.root = v$ 
3  elseif  $u == u.p.left$ 
4       $u.p.left = v$ 
5  else  $u.p.right = v$ 
6  if  $v \neq \text{NIL}$ 
7       $v.p = u.p$ 
```

RB-TRANSPLANT(T, u, v)

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

- **Note.** The colors of the nodes u and v are not changed by RB-TRANSPLANT.

RB Deletion: The Idea

- The strategy for deleting a node *z* is the same as in the **TREE-DELETE** procedure. We handle two cases:
 - Case 1. *z* has at most one child.
 - Case 2. *z* has two children.
 - ❖ Case 2a. The successor of *z* is **not** *z*'s right child.
 - ❖ Case 2b. The successor of *z* is *z*'s right child.

RB Deletion: The Idea

- The strategy for deleting a node z is the same as in the **TREE-DELETE** procedure. We handle two cases:
 - **Case 1.** z has at most one child.
 - **Case 2.** z has two children.
 - ❖ **Case 2a.** The successor of z is not z 's right child.
 - ❖ **Case 2b.** The successor of z is z 's right child.
- Idea. In **Case 1**, “replace” z by its only child, and in **Case 2**, “replace” z by the successor of z .

RB Deletion: The Idea

- The strategy for deleting a node z is the same as in the **TREE-DELETE** procedure. We handle two cases:
 - **Case 1.** z has at most one child.
 - **Case 2.** z has two children.
 - ❖ **Case 2a.** The successor of z is **not** z 's right child.
 - ❖ **Case 2b.** The successor of z is z 's right child.
- **Notations.** In **Case 1**, we'll let $y := z$, and in **Case 2**, $y := \text{succ}(z)$. Observe that y has at most one child. Let x be the only child of y . Note, x could be $T.\text{nil}$.

RB Deletion: The Idea

- The strategy for deleting a node z is the same as in the **TREE-DELETE** procedure. We handle two cases:
 - **Case 1.** z has at most one child.
 - **Case 2.** z has two children.
 - ❖ **Case 2a.** The successor of z is not z 's right child.
 - ❖ **Case 2b.** The successor of z is z 's right child.
- **Notations.** Since, in **Case 2**, $y := \text{succ}(z)$ replaces z , we'll set $y.\text{color} = z.\text{color}$. But, we'll remember y 's original color in the variable $y\text{-original-color}$.

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = z$ and storing its original color.

Handling Case I

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = z$ and storing its original color.

Handling Case I

x is the only child of y .

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = z$ and storing its original color.

Handling **Case I**

x is the only child of y .

Obs I. Owing to the customization of **RB-TRANSPLANT**, in **Case I**, $x.p = y.p$ even if $x = T.\text{nil}$. Recall, node x is the only child of $y = z$.

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = z$ and storing its original color.

Handling **Case I**

x is the only child of y .

Obs I. Owing to the customization of **RB-TRANSPLANT**, in **Case I**, $x.p = y.p$ even if $x = T.\text{nil}$. Recall, node x is the only child of $y = z$.

$y = z$'s parent before it gets replaced by its only child.

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = \text{succ}(z)$ and storing its original color.

Handling Case 2

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = \text{succ}(z)$ and storing its original color.

x is the only child of y .

Handling Case 2

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = \text{succ}(z)$ and storing its original color.

x is the only child of y .

Handling Case 2a

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = \text{succ}(z)$ and storing its original color.

x is the only child of y .

Handling Case 2a

Obs 2a. In Case 2a, the RB-TRANSPLANT call in Step 13 sets $x.p = y.p$ even if $x = T.\text{nil}$. This is just as in Case 1.

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21     if  $y\text{-original-color} == \text{BLACK}$ 
22         RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = \text{succ}(z)$ and storing its original color.

x is the only child of y .

Handling Case 2a

Obs 2a. In Case 2a, the RB-TRANSPLANT call in Step 13 sets $x.p = y.p$ even if $x = T.\text{nil}$. This is just as in Case 1.

y 's parent before it moves up and replaces z

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = \text{succ}(z)$ and storing its original color.

x is the only child of y .

Handling Case 2b

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = \text{succ}(z)$ and storing its original color.

x is the only child of y .

Handling Case 2b

Obs 2b. In Case 2b, Step 16 sets $x.p = y$ even if $x = T.\text{nil}$.

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = \text{succ}(z)$ and storing its original color.

x is the only child of y .

Handling Case 2b

Obs 2b. In Case 2b, Step 16 sets $x.p = y$ even if $x = T.\text{nil}$.

Note. Isn't $x.p$ already y ? Well, no if $x = T.\text{nil}$.

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Setting $y = \text{succ}(z)$ and storing its original color.

x is the only child of y .

Handling Case 2

At the end of Case 2, set $y.\text{color} = z.\text{color}$.

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Claim 1. If the original color of y is **red**, then none of the five RB properties are violated.

We'll prove the claim in the next slide.

RB Deletion: Pseudocode

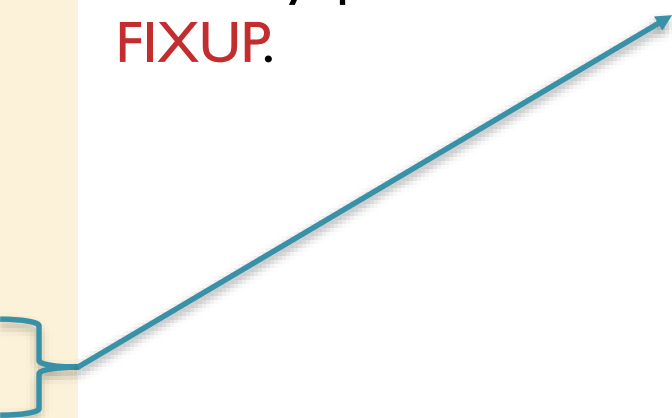
RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Claim 1. If the original color of y is **red**, then none of the five RB properties are violated.

We'll prove the claim in the next slide.

So, we need a fixup only if the original color of y is **black**. This is done in Steps 21-22 by the auxiliary procedure **RB-DELETE-FIXUP**.



RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Claim 1. If the original color of y is **red**, then none of the five RB properties are violated.

We'll prove the claim in the next slide.

So, we need a fixup only if the original color of y is **black**. This is done in Steps 21-22 by the auxiliary procedure **RB-DELETE-FIXUP**.

Note. **RB-DELETE-FIXUP** takes only T and x as arguments but not y and z .

RB Deletion: Pseudocode

RB-DELETE(T, z)

```
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y \neq z.\text{right}$ 
13         RB-TRANSPLANT( $T, y, y.\text{right}$ )
14          $y.\text{right} = z.\text{right}$ 
15          $y.\text{right}.p = y$ 
16     else  $x.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )
```

Claim 1. If the original color of y is **red**, then none of the five RB properties are violated.

We'll prove the claim in the next slide.

So, we need a fixup only if the original color of y is **black**. This is done in Steps 21-22 by the auxiliary procedure **RB-DELETE-FIXUP**.

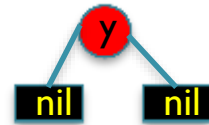
Note. If the time complexity of **RB-DELETE-FIXUP** is $O(\log n)$, then so is the time complexity of **RB-DELETE**.

RB Deletion: Analysis

- **Claim 1.** If the original color of y is **red**, then none of the five RB properties are violated.
- **Proof.** Of course, the tree continues to remain a BST – this was shown in the analysis of the **TREE-DELETE** procedure. So, we only need to worry about the five RB properties.

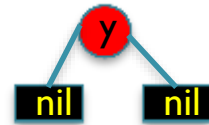
RB Deletion: Analysis

- **Claim 1.** If the original color of y is **red**, then none of the five RB properties are violated.
- **Proof.** The proof follows from the observations below:
- **Obs.** If the original color of y is **red**, then both the children of y are *T.nil*.



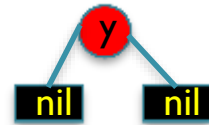
RB Deletion: Analysis

- **Claim 1.** If the original color of y is **red**, then none of the five RB properties are violated.
- **Proof.** The proof follows from the observations below:
- **Obs.** If the original color of y is **red**, then both the children of y are *T.nil*.
- This is because, in both **Case 1** and **2**, y has at most one (non-NIL) child. In a RB tree, a **red** node either has no NIL children, or both children are *T.nil* (otherwise, either Property 4 or 5 is violated).



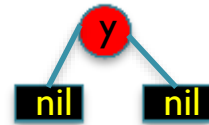
RB Deletion: Analysis

- **Claim 1.** If the original color of y is **red**, then none of the five RB properties are violated.
- **Proof.** The proof follows from the observations below:
- **Obs.** If the original color of y is **red**, then both the children of y are *T.nil*.
- **Obs.** If a **red** node with NIL children is replaced with *T.nil*, then none of the RB properties are violated.



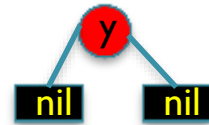
RB Deletion: Analysis

- **Claim 1.** If the original color of y is **red**, then none of the five RB properties are violated.
- **Proof.** The proof follows from the observations below:
- **Obs.** If the original color of y is **red**, then both the children of y are $T.nil$.
- **Obs.** If a **red** node with NIL children is replaced with $T.nil$, then none of the RB properties are violated.
- In **Case 1**, $y = z$ gets replaced by $T.nil$, if y is **red**. This doesn't violate any of the RB properties.



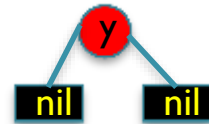
RB Deletion: Analysis

- **Claim 1.** If the original color of y is **red**, then none of the five RB properties are violated.
- **Proof.** The proof follows from the observations below:
- **Obs.** If the original color of y is **red**, then both the children of y are $T.nil$.
- **Obs.** If a **red** node with NIL children is replaced with $T.nil$, then none of the RB properties are violated.
- In **Case 2a**, y gets replaced by $T.nil$ first, if y is **red**. Then, y replaces z and takes the color of z . This doesn't violate any of the RB properties.



RB Deletion: Analysis

- **Claim 1.** If the original color of y is **red**, then none of the five RB properties are violated.
- **Proof.** The proof follows from the observations below:
- **Obs.** If the original color of y is **red**, then both the children of y are ***T.nil***.
- **Obs.** If a **red** node with NIL children is replaced with ***T.nil***, then none of the RB properties are violated.
- In **Case 2b**, y replaces z and takes the color of z . Again, this doesn't violate any of the RB properties.



RB Deletion: Analysis

- **Claim 1.** If the original color of y is **red**, then none of the five RB properties are violated.
- Henceforth, we will assume that the original color of y is **black**.
- We need to understand which RB properties may get violated and how to fix them.

RB Deletion: Analysis

- **Claim 1.** If the original color of y is **red**, then none of the five RB properties are violated.
- Henceforth, we will assume that the original color of y is **black**.
- We need to understand which RB properties may get violated and how to fix them.
- We'll do so by examining **Case 1**, **2a**, and **2b** closely.

RB Deletion: Violations and fixes

- Case I. z has at most one child.

Situation I:

(x is **red**)



- In this situation, x was **red** before. We turn x **black** after it replaces $y = z$.

RB Deletion: Violations and fixes

- Case I. z has at most one child.

Situation I:

(x is **red**)



- Property 1 (every node is **red** or **black**), Property 2 (root is **black**), Property 3 (a leaf is **black**), and Property 4 (no **red-red** edge) can't be violated.

RB Deletion: Violations and fixes

- Case I. z has at most one child.

Situation I:

(x is red)



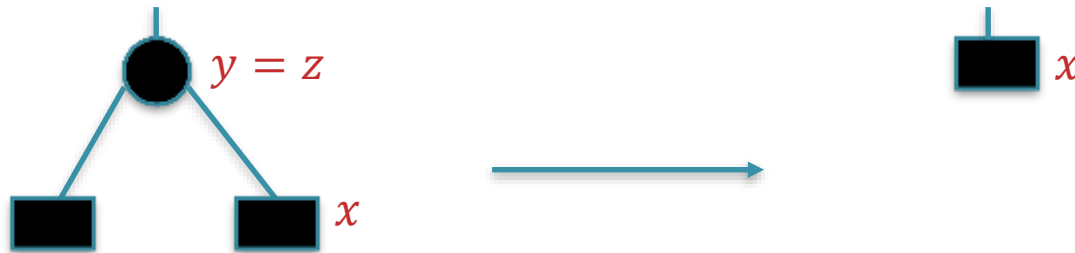
- Also, Property 5 (black-height is well-defined) is not violated as $bh(x) = 1$, which is the same as $bh(y)$ before the deletion of $y = z$, and x is turned **black**.

RB Deletion: Violations and fixes

- Case 1. z has at most one child.

Situation 2:

(x is **black**)



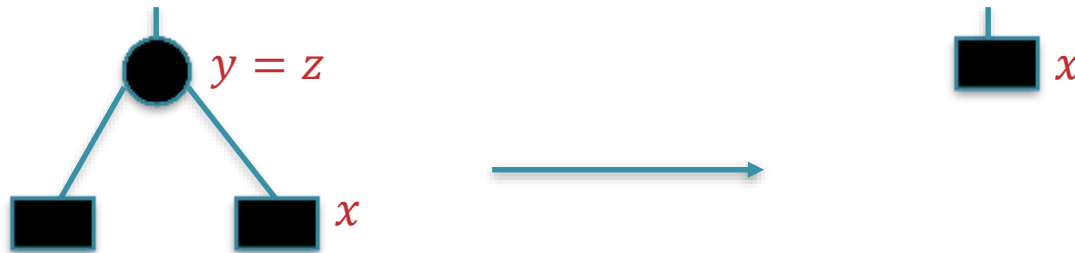
- Property 1 (every node is **red** or **black**), Property 2 (root is **black**), Property 3 (a leaf is **black**), and Property 4 (no **red-red** edge) can't be violated.

RB Deletion: Violations and fixes

- **Case 1.** z has at most one child.

Situation 2:

(x is **black**)



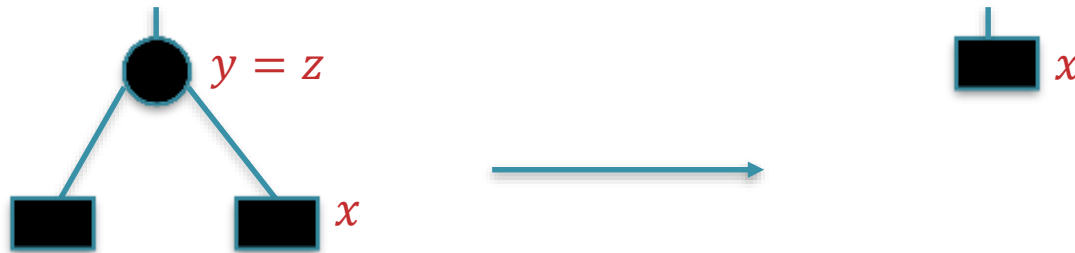
- But Property 5 (black-height is well-defined) may be violated as paths which were passing through y before will now have one less black-height.
- We'll take care of this violation a bit weirdly. $\square$

RB Deletion: Violations and fixes

- **Case 1.** z has at most one child.

Situation 2:

(x is **black**)



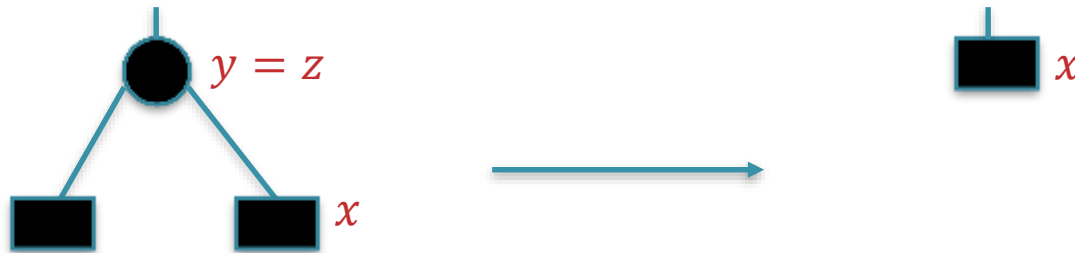
- **A weird way of thinking.** Whenever $x.color = \text{black}$ beforehand, we pretend that it is **doubly-black** afterward, so that Property 5 is not violated, but Property 1 (a node is **red** or **black**) is “violated”.
- **Note.** We’re not actually changing $x.color$ to **doubly-black**, just pretending to be so for the sake of analysis.

RB Deletion: Violations and fixes

- **Case 1.** z has at most one child.

Situation 2:

(x is **black**)



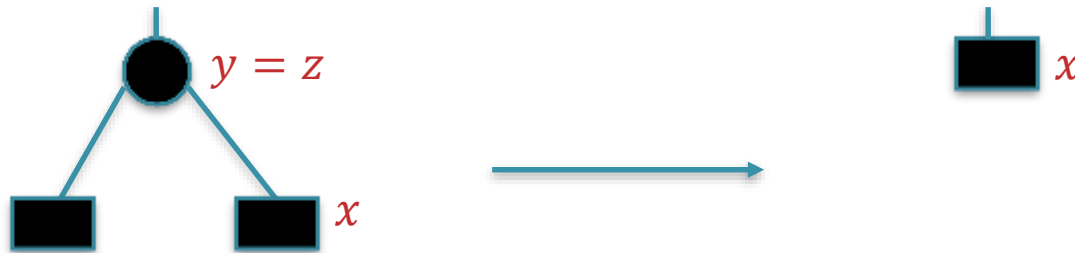
- **A weird way of thinking.** Whenever $x.color = \text{black}$ beforehand, we pretend that it is **doubly-black** afterward, so that Property 5 is not violated, but Property 1 (a node is **red** or **black**) is “violated”.
- The algorithm keeps pushing the **doubly-black** x “upward” until we no longer need to pretend.

RB Deletion: Violations and fixes

- **Case 1.** z has at most one child.

Situation 2:

(x is **black**)



- **A weird way of thinking.** Whenever $x.color = \text{black}$ beforehand, we pretend that it is **doubly-black** afterward, so that Property 5 is not violated, but Property 1 (a node is **red** or **black**) is “violated”.
- The algorithm keeps pushing the **doubly-black** x “upward” until we no longer need to pretend.

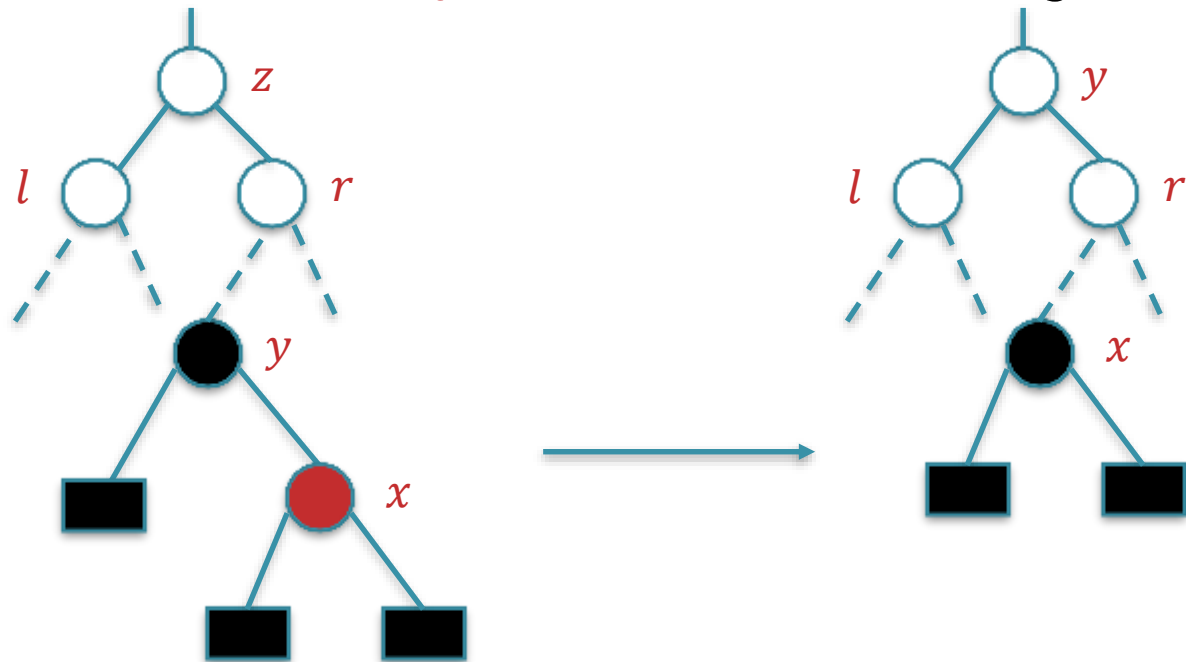
By pointing x to another node up in the tree.

RB Deletion: Violations and fixes

- **Case 2a.** z has two children, $y = \text{succ}(z)$ isn't z 's right child.

Situation 1:

(x is **red**)



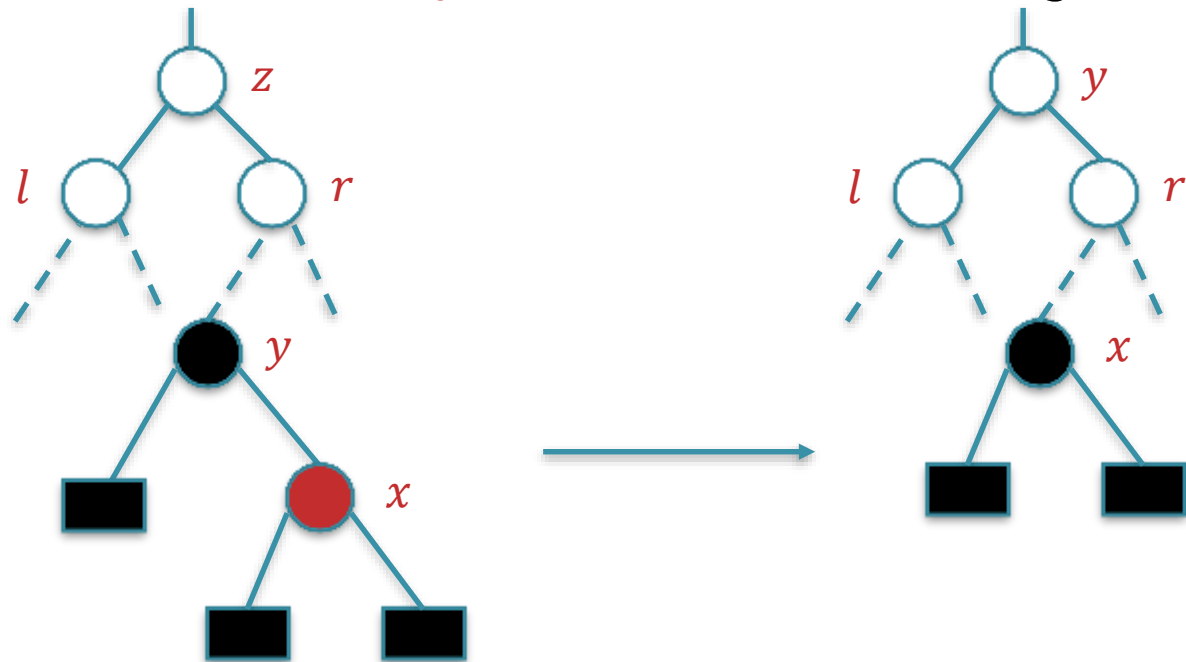
- In this situation, x was **red** before. We turn x **black** after it replaces y ; y replaces z and takes z 's color.

RB Deletion: Violations and fixes

- **Case 2a.** z has two children, $y = \text{succ}(z)$ isn't z 's right child.

Situation 1:

(x is **red**)



- Property 1 (every node is **red** or **black**), Property 2 (root is **black**), Property 3 (a leaf is **black**), and Property 4 (no **red-red** edge) can't be violated.

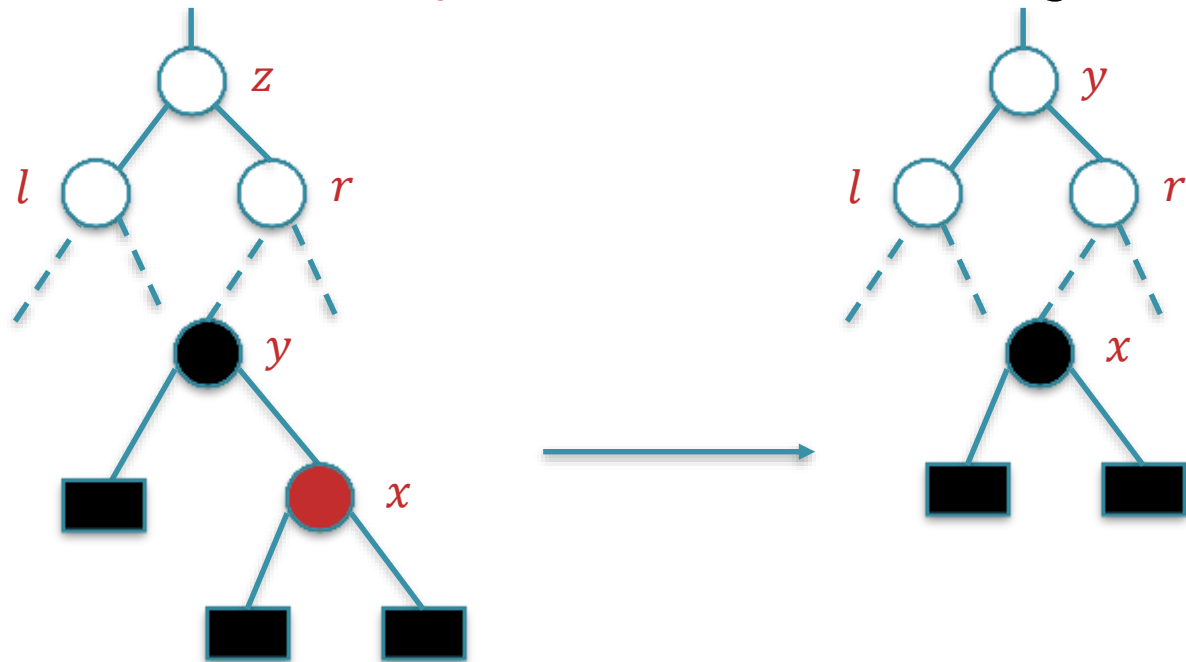
$y.\text{color} = z.\text{color}$

RB Deletion: Violations and fixes

- **Case 2a.** z has two children, $y = \text{succ}(z)$ isn't z 's right child.

Situation 1:

(x is **red**)



- Also, Property 5 (black-height is well-defined) is not violated as $bh(x) = 1$, which is the same as $bh(y)$ before the shifting of y , and x is turned **black**.

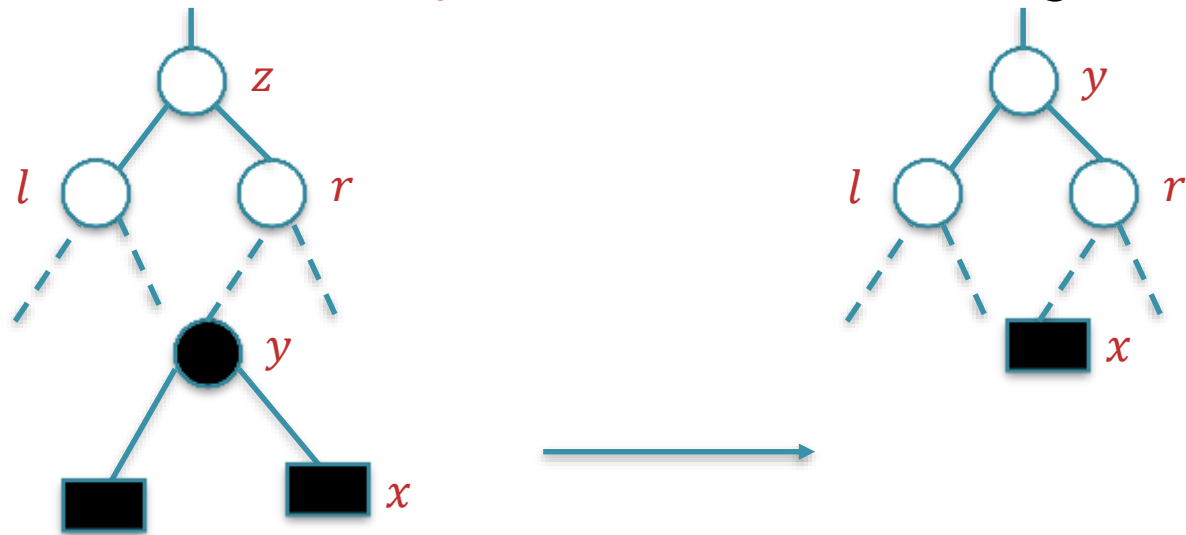
$y.\text{color} = z.\text{color}$

RB Deletion: Violations and fixes

- **Case 2a.** z has two children, $y = \text{succ}(z)$ isn't z 's right child.

Situation 2:

(x is **black**)



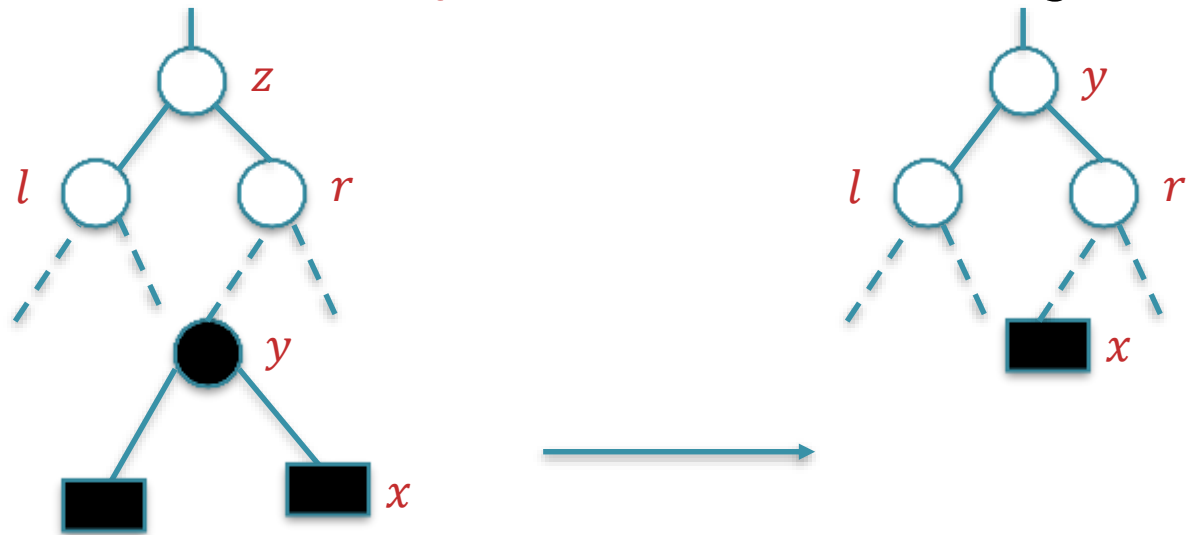
- Property 1 (every node is **red** or **black**), Property 2 (root is **black**), Property 3 (a leaf is **black**), and Property 4 (no **red-red** edge) can't be violated.

RB Deletion: Violations and fixes

- **Case 2a.** z has two children, $y = \text{succ}(z)$ isn't z 's right child.

Situation 2:

(x is **black**)



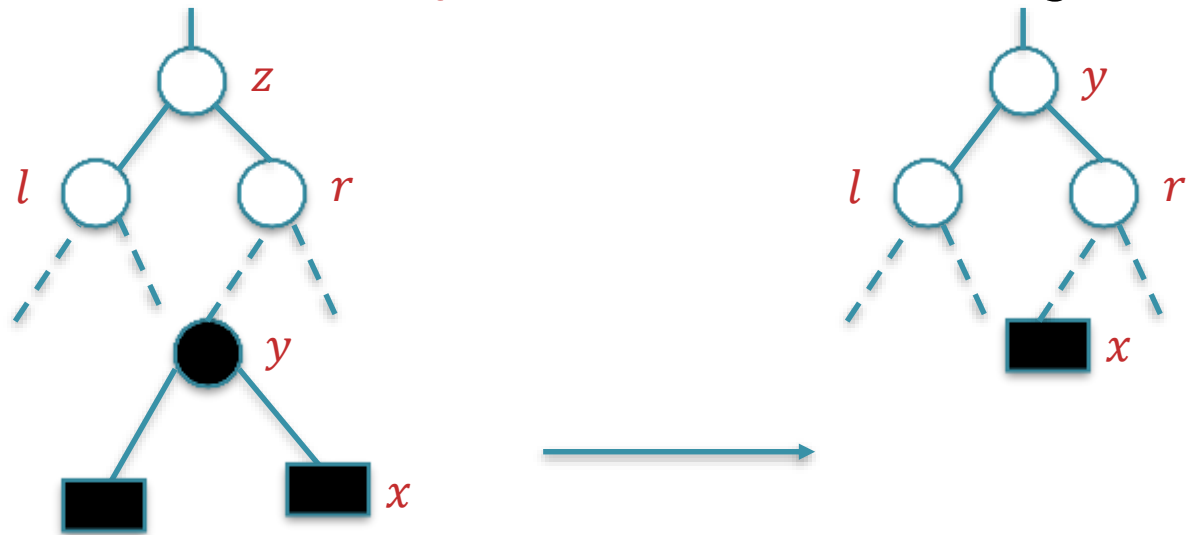
- But Property 5 (black-height is well-defined) may be violated as paths which were passing through y before will now have one less black-height.
- We'll take care of this violation a bit weirdly as before.

RB Deletion: Violations and fixes

- **Case 2a.** z has two children, $y = \text{succ}(z)$ isn't z 's right child.

Situation 2:

(x is **black**)



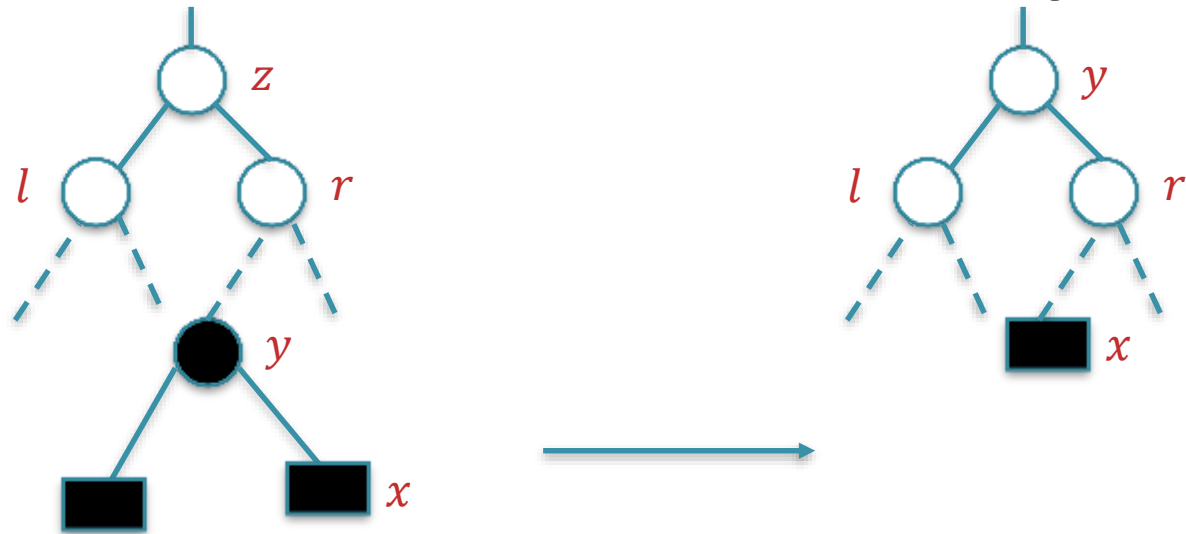
- **Weird thinking** (again). Whenever $x.\text{color} = \text{black}$ beforehand, we pretend that it is **doubly-black** afterward, so that Property 5 is not violated, but Property 1 (a node is **red** or **black**) is “violated”.

RB Deletion: Violations and fixes

- **Case 2a.** z has two children, $y = \text{succ}(z)$ isn't z 's right child.

Situation 2:

(x is **black**)



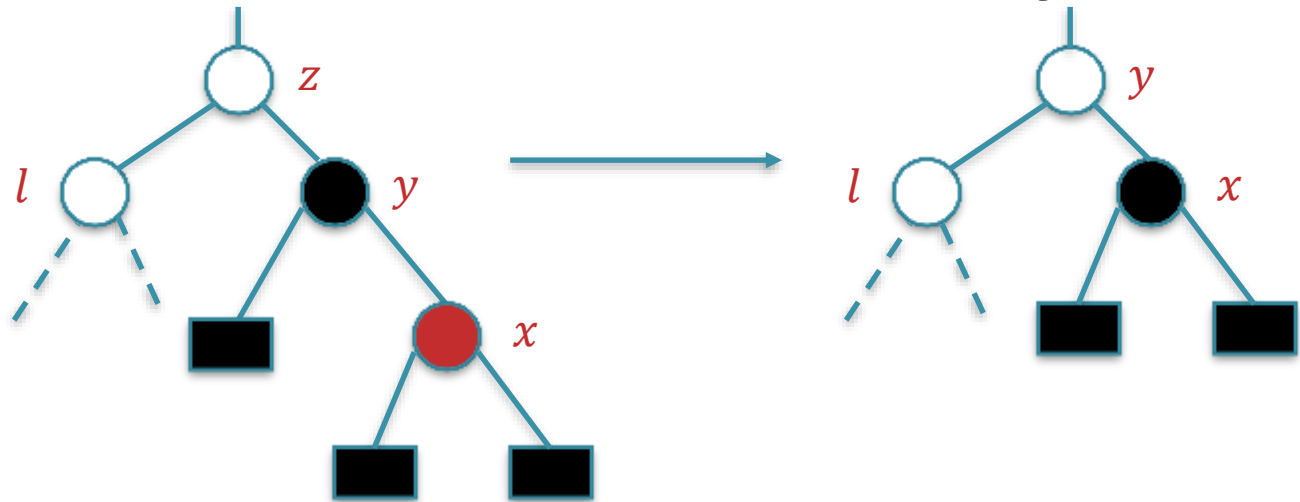
- The algorithm keeps pushing the **doubly-black** x “upward” until we no longer need to pretend.

RB Deletion: Violations and fixes

- Case 2b. z has two children, $y = \text{succ}(z)$ is z 's right child.

Situation 1:

(x is **red**)



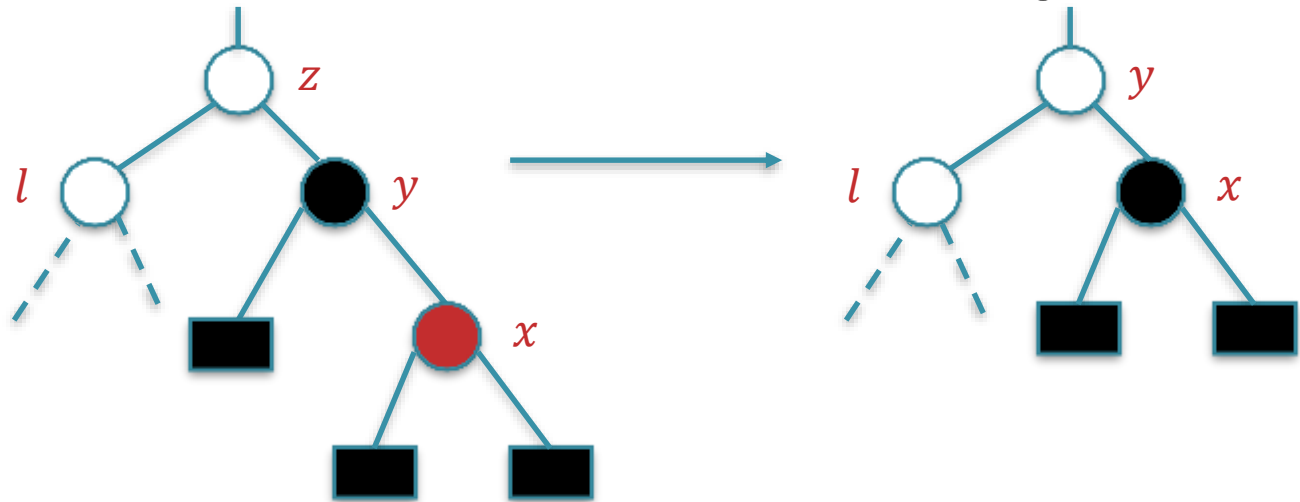
- In this situation, x was **red** before. We turn x **black** after it replaces y ; y replaces z and takes z 's color.

RB Deletion: Violations and fixes

- **Case 2b.** z has two children, $y = \text{succ}(z)$ is z 's right child.

Situation 1:

(x is **red**)



- Property 1 (every node is **red** or **black**), Property 2 (root is **black**), Property 3 (a leaf is **black**), and Property 4 (no **red-red** edge) can't be violated.

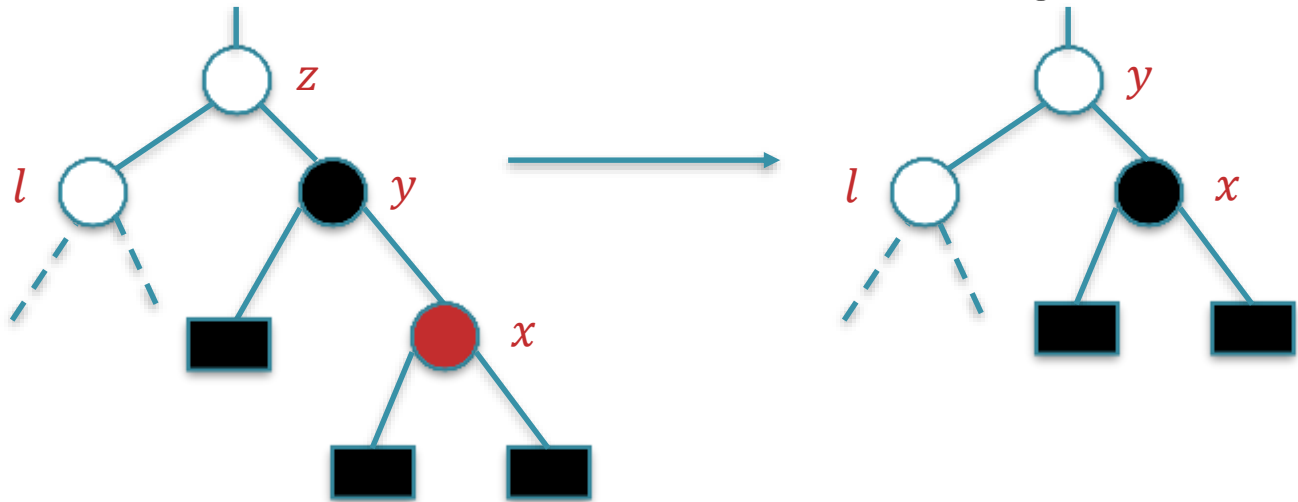
$y.\text{color} = z.\text{color}$

RB Deletion: Violations and fixes

- **Case 2b.** z has two children, $y = \text{succ}(z)$ is z 's right child.

Situation 1:

(x is **red**)



- Also, Property 5 (black-height is well-defined) is not violated as $bh(x) = 1$, which is the same as $bh(y)$ before the shifting of y , and x is turned **black**.

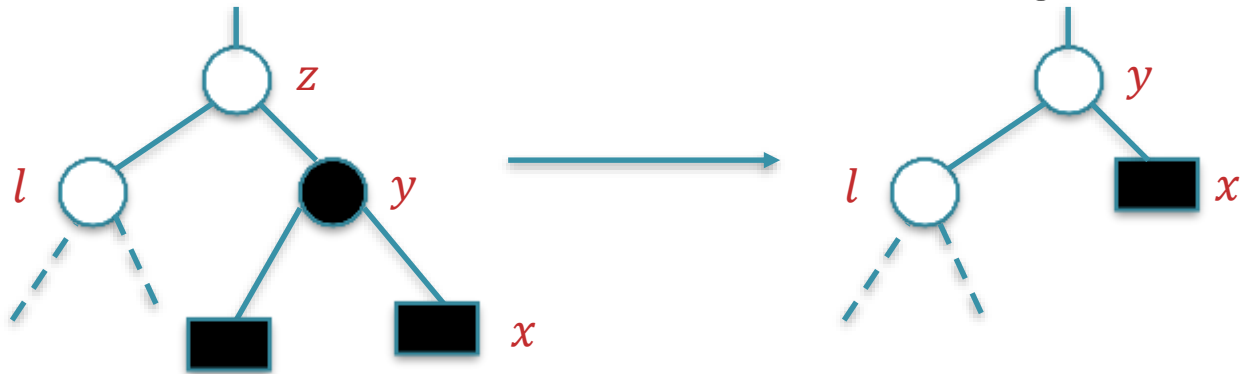
$y.\text{color} = z.\text{color}$

RB Deletion: Violations and fixes

- **Case 2b.** z has two children, $y = \text{succ}(z)$ is z 's right child.

Situation 2:

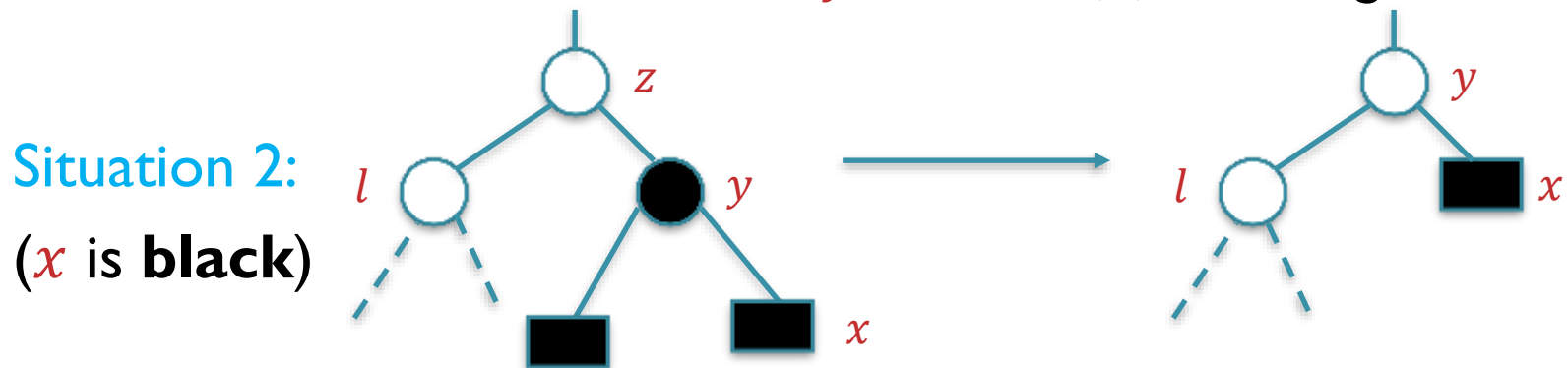
(x is **black**)



- Property 1 (every node is **red** or **black**), Property 2 (root is **black**), Property 3 (a leaf is **black**), and Property 4 (no **red-red** edge) can't be violated.

RB Deletion: Violations and fixes

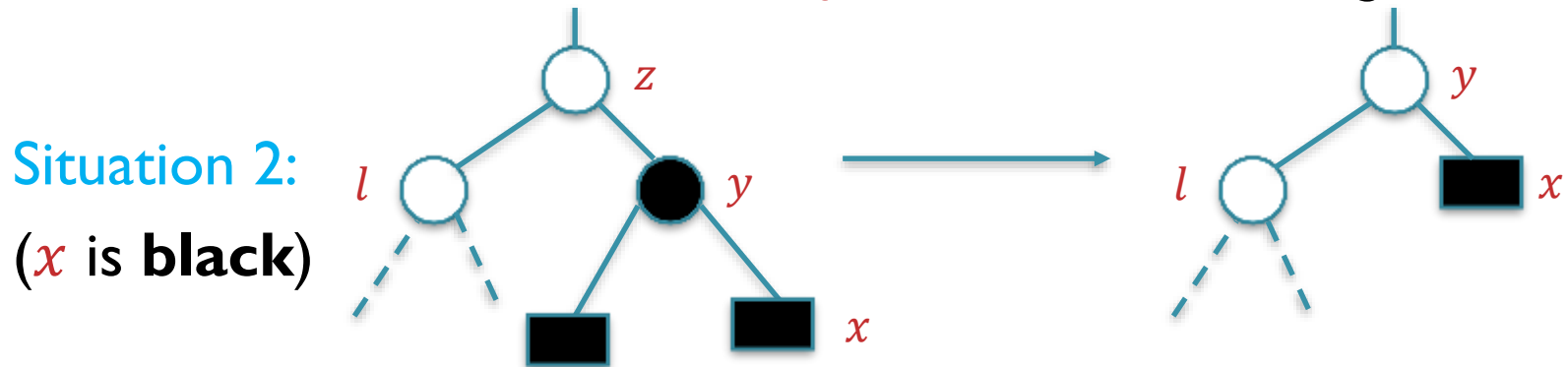
- **Case 2b.** z has two children, $y = \text{succ}(z)$ is z 's right child.



- But Property 5 (black-height is well-defined) may be violated as paths which were passing through y before will now have one less black-height.
- We'll take care of this violation a bit weirdly as before.

RB Deletion: Violations and fixes

- **Case 2b.** z has two children, $y = \text{succ}(z)$ is z 's right child.



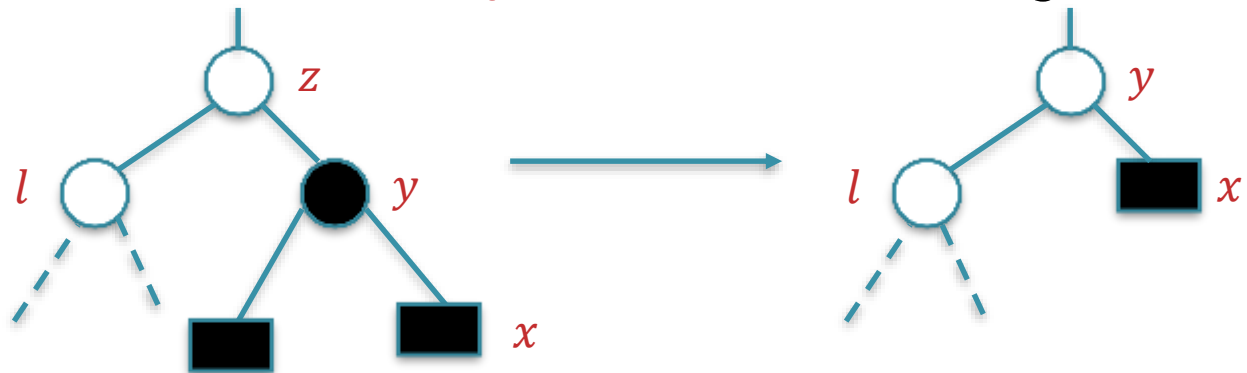
- **Weird thinking** (again). Whenever $x.\text{color} = \text{black}$ beforehand, we pretend that it is **doubly-black** afterward, so that Property 5 is not violated, but Property 1 (a node is **red** or **black**) is “violated”.

RB Deletion: Violations and fixes

- **Case 2b.** z has two children, $y = \text{succ}(z)$ is z 's right child.

Situation 2:

(x is **black**)



- **Weird thinking** (again). Whenever $x.\text{color} = \text{black}$ beforehand, we pretend that it is **doubly-black** afterward, so that Property 5 is not violated, but Property 1 (a node is **red** or **black**) is “violated”.
- The algorithm keeps pushing the **doubly-black** x “upward” until we no longer need to pretend.

RB Deletion: Violations and fixes

- **Obs.** Thus, in each of the three cases:
 - If **Situation 1** ($x.color = \text{red}$) arises, then we simply color x **black** afterward and stop.
 - If **Situation 2** ($x.color = \text{black}$) arises, then we pretend that x is **doubly-black** afterward and push x “upward” until we no longer need to pretend.

RB Deletion: Violations and fixes

- **Obs.** Thus, in each of the three cases:
 - If **Situation 1** ($x.color = \text{red}$) arises, then we simply color x **black** afterward and stop.
 - If **Situation 2** ($x.color = \text{black}$) arises, then we pretend that x is **doubly-black** afterward and push x “upward” until we no longer need to pretend.
- **Obs.** If we manage to push x all the way up and make it $T.root$, then we no longer need to pretend that it is **doubly-black**.

RB-Delete-Fixup: Moving x up the tree

- We move x up the tree in a **while** loop, which runs as long as $x \neq T.root$ and $x.color = \text{black}$.
- Within the loop, we have two symmetric situations: x is a left child, and x is a right child.

RB-Delete-Fixup: Moving x up the tree

- We move x up the tree in a **while** loop, which runs as long as $x \neq T.root$ and $x.color = \text{black}$.
- Within the loop, we have two symmetric situations: x is a left child, and x is a right child.

```
RB-DELETE-FIXUP( $T, x$ )
```

```
1  while  $x \neq T.root$  and  $x.color == \text{BLACK}$   
2      if  $x == x.p.left$            // is  $x$  a left child?
```

Symmetric Situation 1

```
23      else // same as lines 3–22, but with “right” and “left” exchanged
```

Symmetric Situation 2

```
44   $x.color = \text{BLACK}$ 
```

RB-Delete-Fixup: Moving x up the tree

- We move x up the tree in a **while** loop, which runs as long as $x \neq T.root$ and $x.color = \text{black}$.
- Within the loop, we have two symmetric situations: x is a left child, and x is a right child.

```
RB-DELETE-FIXUP( $T, x$ )
```

```
1  while  $x \neq T.root$  and  $x.color == \text{BLACK}$   
2      if  $x == x.p.left$            // is  $x$  a left child?
```

Symmetric Situation 1

```
23      else // same as lines 3–22, but with “right” and “left” exchanged
```

Symmetric Situation 2

```
44   $x.color = \text{BLACK}$ 
```

If $x.color = \text{red}$ before, then **RB-DELETE-FIXUP** turns x **black** and stops.

RB-Delete-Fixup: Moving x up the tree

- We move x up the tree in a **while** loop, which runs as long as $x \neq T.root$ and $x.color = \text{black}$.
- Within the loop, we have two symmetric situations: x is a left child, and x is a right child.

```
RB-DELETE-FIXUP( $T, x$ )
```

```
1  while  $x \neq T.root$  and  $x.color == \text{BLACK}$   
2      if  $x == x.p.left$            // is  $x$  a left child?
```

Symmetric Situation 1

```
23      else // same as lines 3–22, but with “right” and “left” exchanged
```

Symmetric Situation 2

```
44   $x.color = \text{BLACK}$ 
```

We'll focus on Symmetric Situation 1.

RB-Delete-Fixup: Moving x up the tree

- We move x up the tree in a **while** loop, which runs as long as $x \neq T.root$ and $x.color = \text{black}$.
- Within the loop, we have two symmetric situations: x is a left child, and x is a right child.
- **Note.** To handle the two symmetric situations, we need access to $x.p$, even if x is $T.nil$. That's why $x.p$ is set appropriately in the **RB-DELETE** procedure.

RB-Delete-Fixup: Moving x up the tree

- We move x up the tree in a **while** loop, which runs as long as $x \neq T.root$ and $x.color = \text{black}$.
- Within the loop, we have two symmetric situations: x is a left child, and x is a right child.
- **Note.** To handle the two symmetric situations, we need access to $x.p$, even if x is $T.nil$. That's why $x.p$ is set appropriately in the **RB-DELETE** procedure.
- **Notation.** The **sibling** of x is denoted by w .
- Since x is **doubly-black**, w cannot be $T.nil$.

RB-Delete-Fixup: Moving x up the tree

- We handle **Symmetric Situation I** in four cases:
 - Case 1: w is **red**.
 - Case 2: w is **black**; its children are also **black**.
 - Case 3: w is **black**; its left child is **red**, right child is **black**.
 - Case 4: w is **black**; its right child is **red**.
- **Note.** The four cases are exhaustive.

RB-Delete-Fixup: Moving x up the tree

- We handle **Symmetric Situation I** in four cases:
 - **Case 1**: w is **red**.
 - **Case 2**: w is **black**; its children are also **black**.
 - **Case 3**: w is **black**; its left child is **red**, right child is **black**.
 - **Case 4**: w is **black**; its right child is **red**.
- **Note**. The four cases are exhaustive.
- **Caution**. Do not confuse these cases with **Case 1**, **Case 2a** and **Case 2b** that we used in the **RB-DELETE** procedure to delete the node z .

RB-Delete-Fixup: Moving x up the tree

- We handle **Symmetric Situation I** in four cases:
 - **Case 1:** w is **red**.
 - **Case 2:** w is **black**; its children are also **black**.
 - **Case 3:** w is **black**; its left child is **red**, right child is **black**.
 - **Case 4:** w is **black**; its right child is **red**.
- Handling each of the cases will require a constant number of rotations and color change.

RB-Delete-Fixup: Moving x up the tree

- We handle **Symmetric Situation I** in four cases:
 - Case 1: w is **red**.
 - Case 2: w is **black**; its children are also **black**.
 - Case 3: w is **black**; its left child is **red**, right child is **black**.
 - Case 4: w is **black**; its right child is **red**.
- Case 4 will make x the root and exit the loop.

RB-Delete-Fixup: Moving x up the tree

- We handle **Symmetric Situation I** in four cases:
 - **Case 1**: w is **red**.
 - **Case 2**: w is **black**; its children are also **black**.
 - **Case 3**: w is **black**; its left child is **red**, right child is **black**.
 - **Case 4**: w is **black**; its right child is **red**.
- **Case 4** will make x the root and exit the loop.
- **Case 3** will reduce to **Case 4**.

RB-Delete-Fixup: Moving x up the tree

- We handle **Symmetric Situation I** in four cases:
 - **Case 1**: w is **red**.
 - **Case 2**: w is **black**; its children are also **black**.
 - **Case 3**: w is **black**; its left child is **red**, right child is **black**.
 - **Case 4**: w is **black**; its right child is **red**.
- **Case 4** will make x the root and exit the loop.
- **Case 3** will reduce to **Case 4**.
- **Case 1** will reduce to one of **Case 2**, **3**, or **4**.

RB-Delete-Fixup: Moving x up the tree

- We handle **Symmetric Situation I** in four cases:
 - **Case 1**: w is **red**.
 - **Case 2**: w is **black**; its children are also **black**.
 - **Case 3**: w is **black**; its left child is **red**, right child is **black**.
 - **Case 4**: w is **black**; its right child is **red**.
- **Case 4** will make x the root and exit the loop.
- **Case 3** will reduce to **Case 4**.
- **Case 1** will reduce to one of **Case 2**, **3**, or **4**.
- If **Case 2** enters through **Case 1** (or $x.p.color = \text{red}$), then $x.color$ becomes **red** and so the loop exits.

RB-Delete-Fixup: Moving x up the tree

- We handle **Symmetric Situation I** in four cases:
 - **Case 1**: w is **red**.
 - **Case 2**: w is **black**; its children are also **black**.
 - **Case 3**: w is **black**; its left child is **red**, right child is **black**.
 - **Case 4**: w is **black**; its right child is **red**.
- **Case 4** will make x the root and exit the loop.
- **Case 3** will reduce to **Case 4**.
- **Case 1** will reduce to one of **Case 2**, **3**, or **4**.
- Otherwise, **Case 2** pushes the node x up the tree.

RB-Delete-Fixup: Moving x up the tree

- We handle **Symmetric Situation I** in four cases:
 - **Case 1**: w is **red**.
 - **Case 2**: w is **black**; its children are also **black**.
 - **Case 3**: w is **black**; its left child is **red**, right child is **black**.
 - **Case 4**: w is **black**; its right child is **red**.
- In summary, each of **Case 1**, **3** and **4** lead to termination after performing a constant number of rotations and color change.

RB-Delete-Fixup: Moving x up the tree

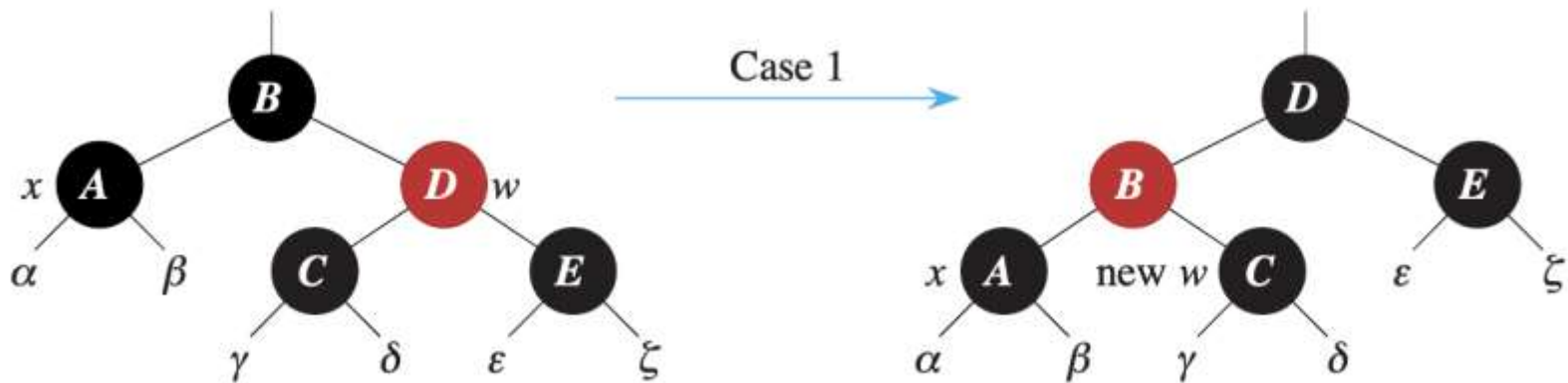
- We handle **Symmetric Situation I** in four cases:
 - **Case 1**: w is **red**.
 - **Case 2**: w is **black**; its children are also **black**.
 - **Case 3**: w is **black**; its left child is **red**, right child is **black**.
 - **Case 4**: w is **black**; its right child is **red**.
- In summary, each of **Case 1**, **3** and **4** lead to termination after performing a constant number of rotations and color change. **Case 2** is the only case in which the loop can be repeated, but then the pointer x moves up the tree.

RB-Delete-Fixup: Moving x up the tree

- We handle **Symmetric Situation I** in four cases:
 - **Case 1:** w is **red**.
 - **Case 2:** w is **black**; its children are also **black**.
 - **Case 3:** w is **black**; its left child is **red**, right child is **black**.
 - **Case 4:** w is **black**; its right child is **red**.
- Thus, the time complexity of **RB-DELETE-FIXUP** is $O(\log n)$ and so is the time complexity of **RB-DELETE**.

RB-Delete-Fixup: Handling Case I

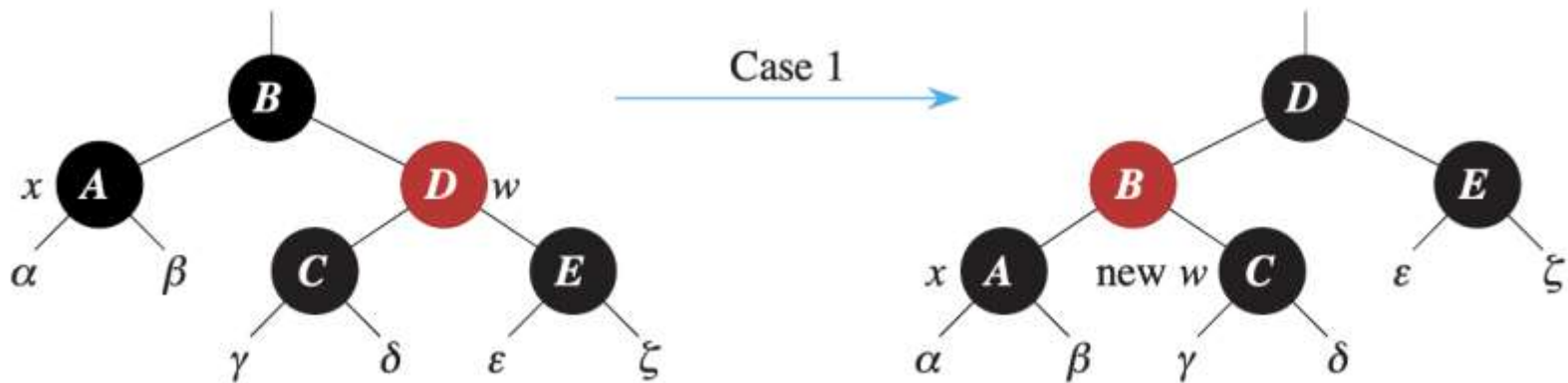
- Case I: w is red.



- Lemma 1.** Assuming that x is **doubly-black**, none of the five RB properties are violated after handling Case I as above. (*Proof. Practice problem*)

RB-Delete-Fixup: Handling Case I

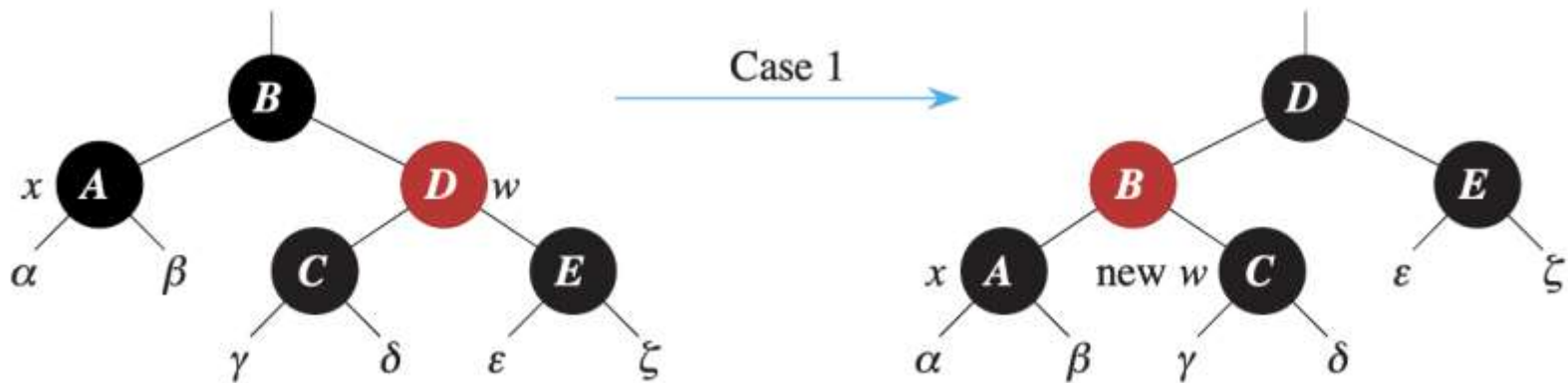
- Case I: w is red.



- Lemma 1.** Assuming that x is **doubly-black**, none of the five RB properties are violated after handling **Case I** as above. (*Proof. Practice problem*)
- Note that the new w is **black**, so it reduces to **Case 2, 3, or 4**. Also, note that the new $x.p$ is **red**.

RB-Delete-Fixup: Handling Case I

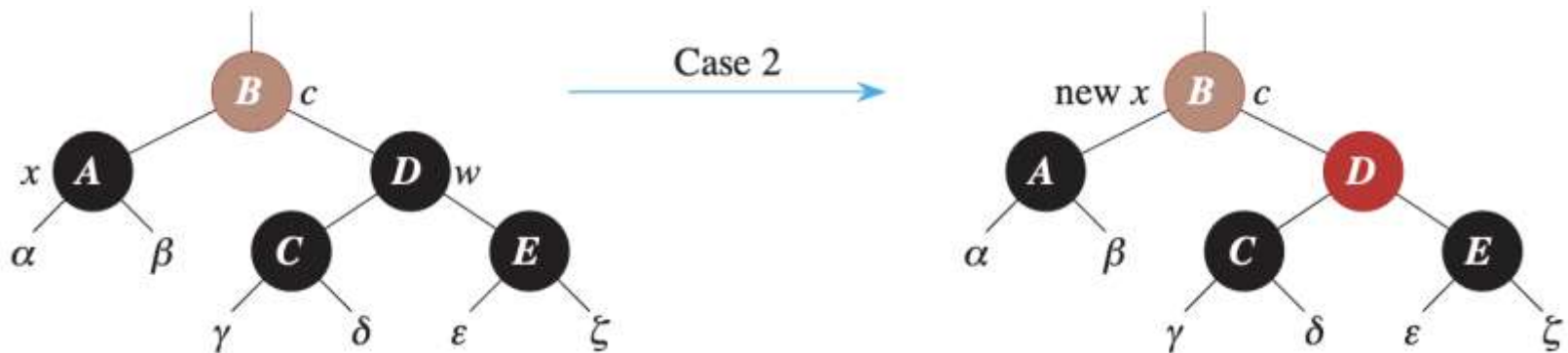
- Case I: w is red.



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

RB-Delete-Fixup: Handling Case 2

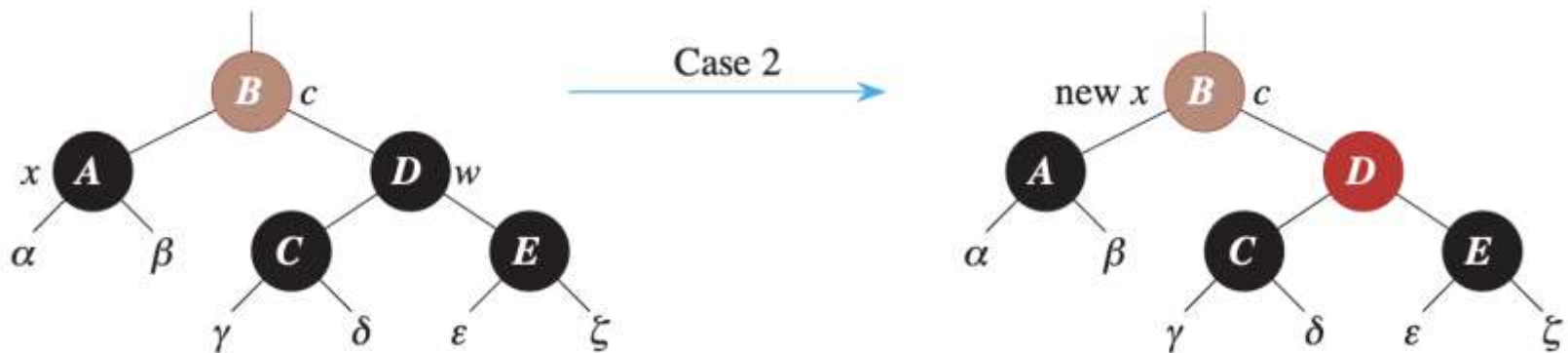
- Case 2: w is **black**; its children are also **black**.



- **Notation.** The color of the brown node $x.p$ is c , which is either **red** or **black**.
- Notice that the pointer x moves up in this case.

RB-Delete-Fixup: Handling Case 2

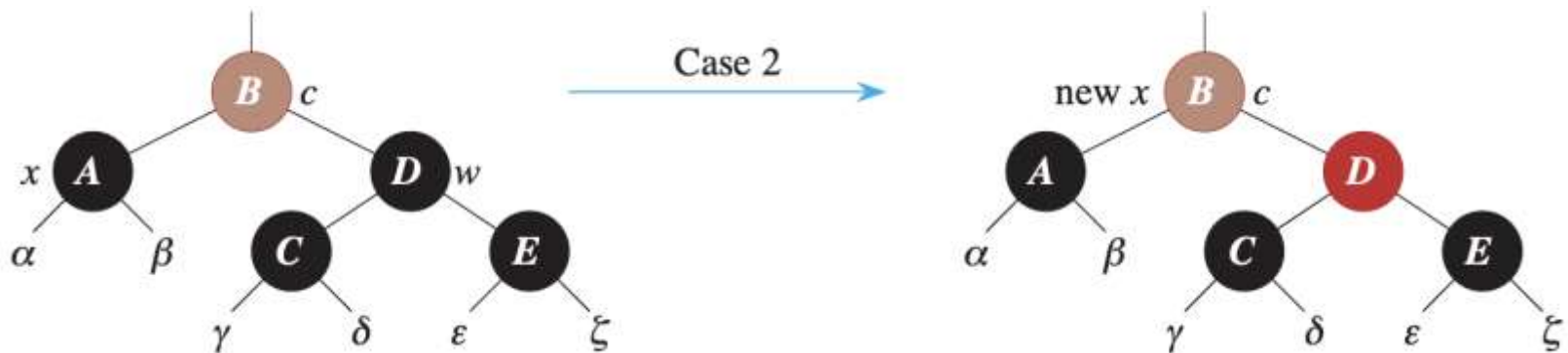
- **Case 2:** w is **black**; its children are also **black**.



- **Lemma 2.** After handling **Case 2** as above, if the color of the new x is **red**, then turning it to **black** satisfies all the five RB properties.
- Recall, after handling **Case 1**, $x.p$ becomes **red**. So, if **Case 2** enters through **Case 1**, then we exit the loop.

RB-Delete-Fixup: Handling Case 2

- **Case 2:** w is **black**; its children are also **black**.

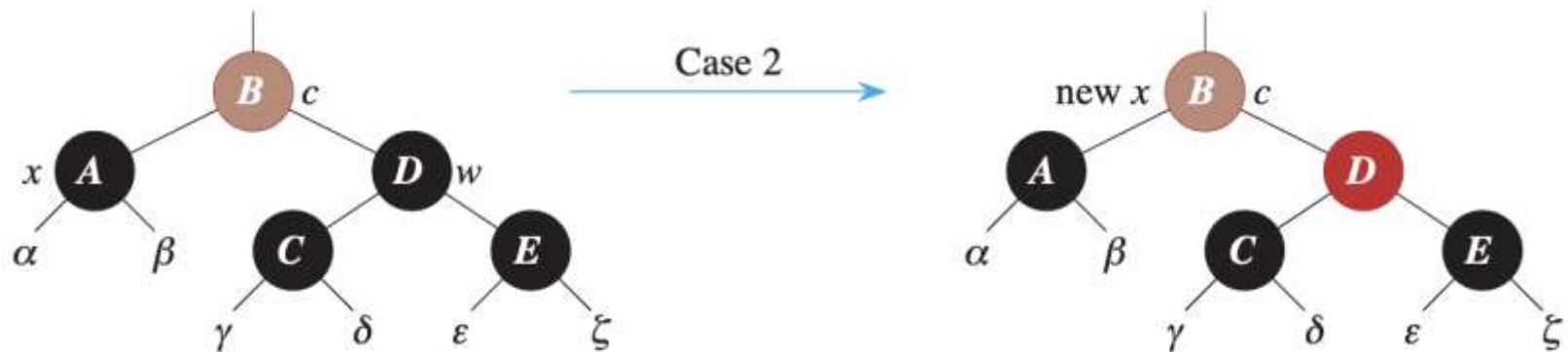


- **Lemma 2.** After handling **Case 2** as above, if the color of the new x is **red**, then turning it to **black** satisfies all the five RB properties. Otherwise, if the color of the new x is **black**, then by treating x as **doubly-black** none except Property 1 is violated.

Proof. Practice problem

RB-Delete-Fixup: Handling Case 2

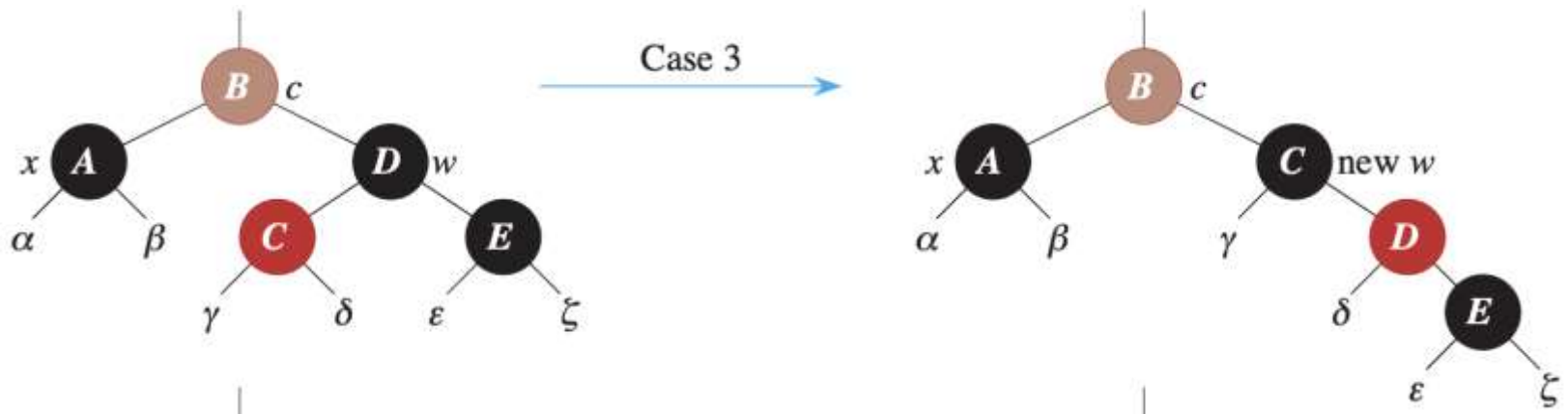
- Case 2: w is **black**; its children are also **black**.



$w.color = \text{RED}$
 $x = x.p$

RB-Delete-Fixup: Handling Case 3

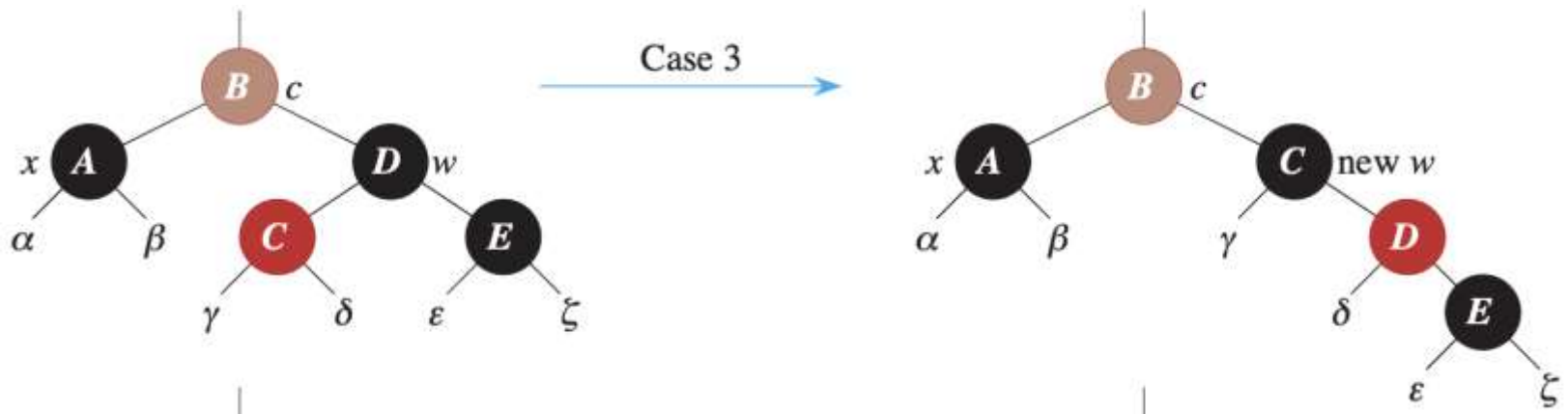
- **Case 3:** w is **black**; its left child is **red**, right child is **black**.



- **Notation.** The color of the brown node $x.p$ is c , which is either **red** or **black**.
- Notice that the new w 's right child is **red**, and so, **Case 3** reduces to **Case 4**.

RB-Delete-Fixup: Handling Case 3

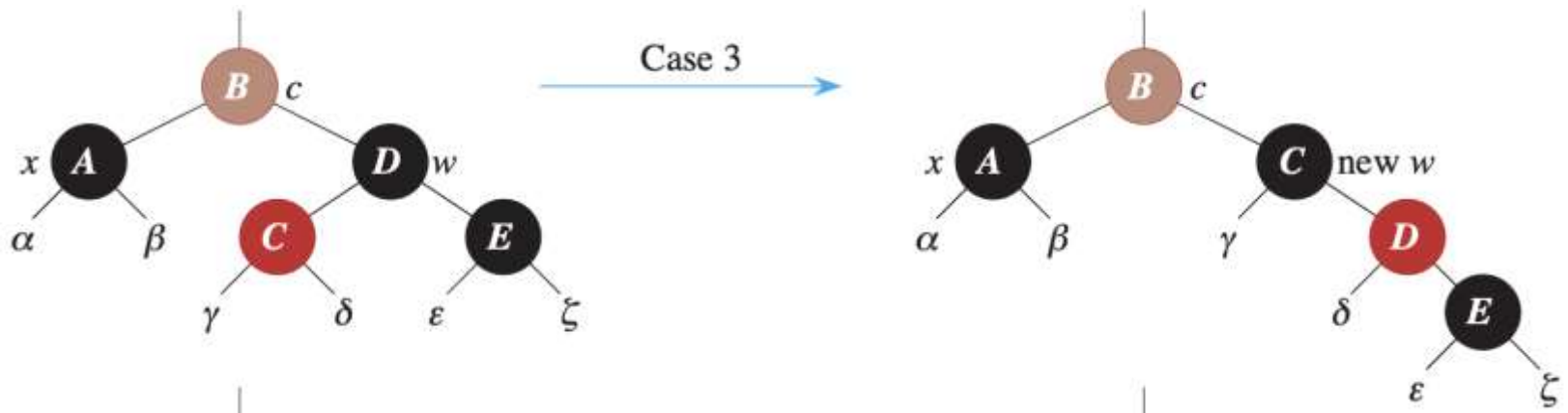
- **Case 3:** w is **black**; its left child is **red**, right child is **black**.



- **Lemma 3.** Assuming that x is **doubly-black**, none except Property 1 is violated after handling **Case 3** as above. (*Proof. Practice problem*)

RB-Delete-Fixup: Handling Case 3

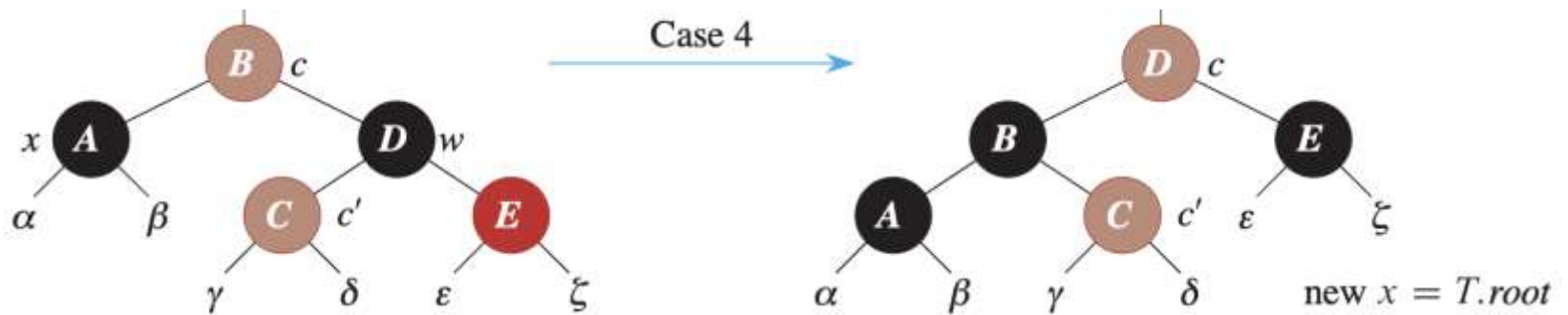
- **Case 3:** w is **black**; its left child is **red**, right child is **black**.



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

RB-Delete-Fixup: Handling Case 4

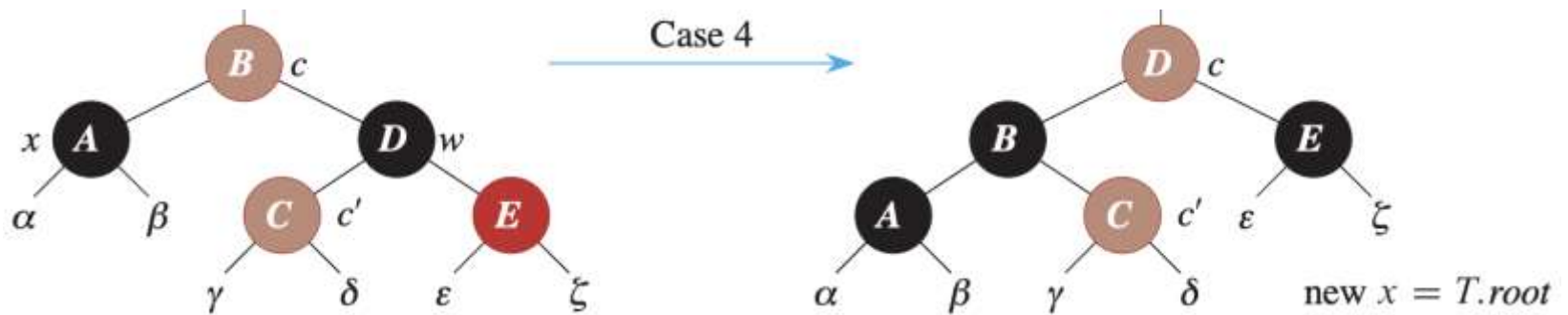
- Case 4: w is **black**; its right child is **red**.



- Notation.** The color of the brown nodes $x.p$ and $w.left$ are c and c' , respectively.
- Notice that in this case, we set $x = T.root$.

RB-Delete-Fixup: Handling Case 4

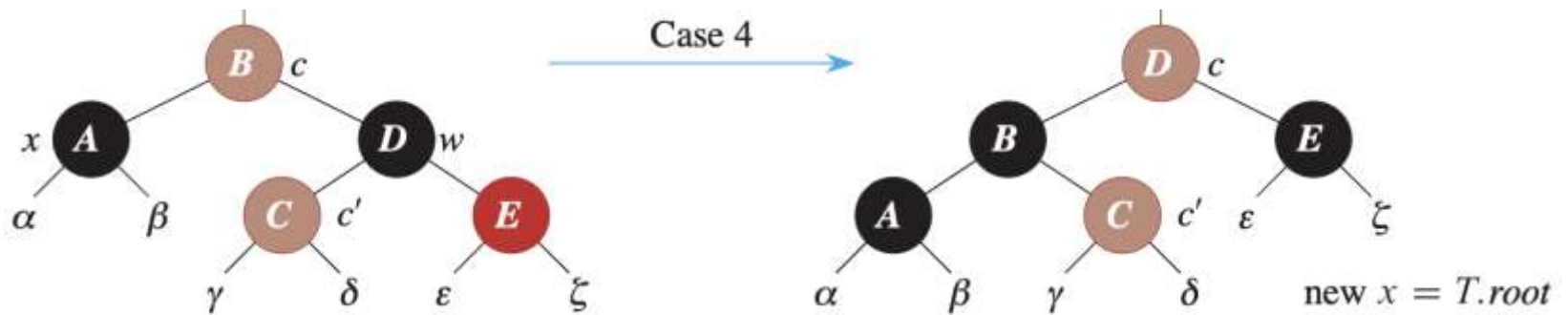
- **Case 4:** w is **black**; its right child is **red**.



- **Lemma 4.** After handling **Case 4** as above, if we set the color of $x = T.root$ to **black**, then none of the five RB properties are violated. (*Proof. Practice problem*)

RB-Delete-Fixup: Handling Case 4

- Case 4: w is **black**; its right child is **red**.



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

RB-Delete-Fixup: Pseudocode

RB-DELETE-FIXUP(T, x)

```
1  while  $x \neq T.root$  and  $x.color == BLACK$ 
2      if  $x == x.p.left$            // is  $x$  a left child?
3           $w = x.p.right$          //  $w$  is  $x$ 's sibling
4          if  $w.color == RED$ 
5               $w.color = BLACK$ 
6               $x.p.color = RED$ 
7              LEFT-ROTATE( $T, x.p$ )
8               $w = x.p.right$ 
9          if  $w.left.color == BLACK$  and  $w.right.color == BLACK$ 
10              $w.color = RED$ 
11              $x = x.p$ 
12         else
13             if  $w.right.color == BLACK$ 
14                  $w.left.color = BLACK$ 
15                  $w.color = RED$ 
16                 RIGHT-ROTATE( $T, w$ )
17                  $w = x.p.right$ 
18              $w.color = x.p.color$ 
19              $x.p.color = BLACK$ 
20              $w.right.color = BLACK$ 
21             LEFT-ROTATE( $T, x.p$ )
22              $x = T.root$ 
23     else // same as lines 3–22, but with “right” and “left” exchanged
```

Symmetric Situation 1

Symmetric Situation 2

Time complexity:
 $O(\log n)$

44 $x.color = BLACK$

Practice problems

Practice Problem I

Prove Lemma 1, 2, 3 and 4.

Practice Problem 2

Consider the operation $\text{RB-ENUMERATE}(T, r, a, b)$, which outputs all the keys k such that $a \leq k \leq b$ in a subtree rooted at node r in an n -node red-black tree T . Describe how to implement RB-ENUMERATE in $\Theta(m + \lg n)$ time, where m is the number of keys that are output. Assume that the keys in T are unique and that the values a and b appear as keys in T . How does your solution change if a and b might not appear in T ?

Practice Problem 3

~~13-2~~ *Join operation on red-black trees*

The *join* operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \leq x.key \leq x_2.key$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

- a.* Suppose that you store the black-height of a red-black tree T as the new attribute $T.bh$. Argue that RB-INSERT and RB-DELETE can maintain the bh attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show how to determine the black-height of each node visited while descending through T , using $O(1)$ time per node visited.

Practice Problem 3 (contd.)

Let T_1 and T_2 be red-black trees and x be a key value such that for any nodes x_1 in T_1 and x_2 in T_2 , we have $x_1.key \leq x.key \leq x_2.key$. You will show how to implement the operation $\text{RB-JOIN}(T_1, x, T_2)$, which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2 .

- b.** Assume that $T_1.bh \geq T_2.bh$. Describe an $O(\lg n)$ -time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $T_2.bh$.
- c.** Let T_y be the subtree rooted at y . Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.

Practice Problem 3 (contd.)

- d.* What color should you make x so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(\lg n)$ time.
- e.* Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \leq T_2.bh$.
- f.* Argue that the running time of RB-JOIN is $O(\lg n)$.