

# Red-Black Trees (2)

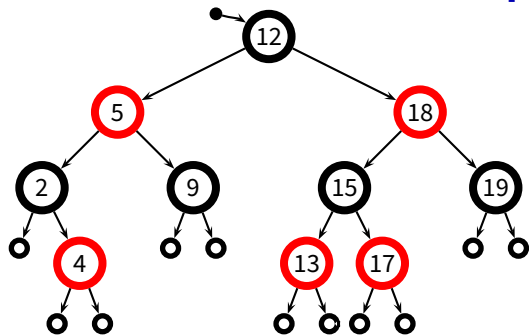
Antonio Carzaniga

Faculty of Informatics  
Università della Svizzera italiana

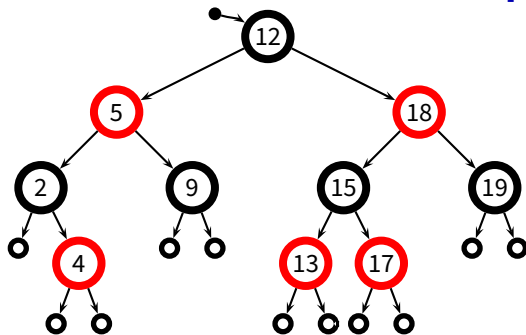
April 13, 2016

# Recap on Red-Black Trees

## Recap on Red-Black Trees

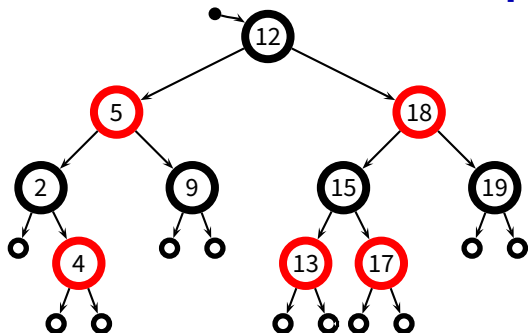


## Recap on Red-Black Trees



- *Red-black-tree property*

## Recap on Red-Black Trees



### ■ Red-black-tree property

1. every node is either **red** or **black**
2. the root is **black**
3. every (NIL) leaf is **black**
4. if a node is **red**, then both its children are **black**
5. for every node  $x$ , each path from  $x$  to its descendant leaves has the same number of **black** nodes  $bh(x)$  (the *black-height* of  $x$ )

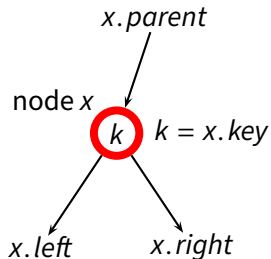
## Recap on Red-Black Trees (2)

### ■ Implementation

- ▶  $T$  represents the tree, which consists of a set of *nodes*
- ▶  $T.root$  is the root node of tree  $T$
- ▶  $T.nil$  is the “sentinel” node of tree  $T$

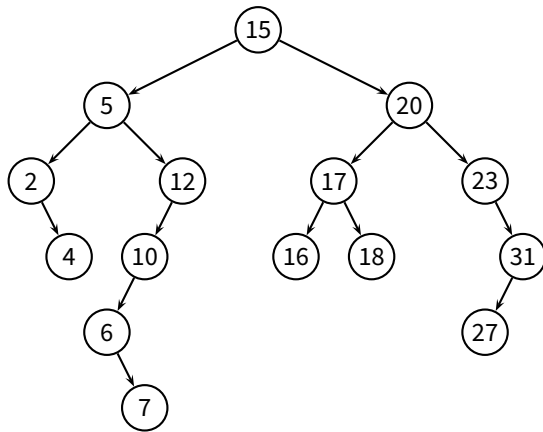
### Nodes

- ▶  $x.parent$  is the parent of node  $x$
- ▶  $x.key$  is the key stored in node  $x$
- ▶  $x.left$  is the left child of node  $x$
- ▶  $x.right$  is the right child of node  $x$
- ▶  $x.color \in \{\text{RED}, \text{BLACK}\}$  is the color of node  $x$



# Recap on Deletion in Binary Trees

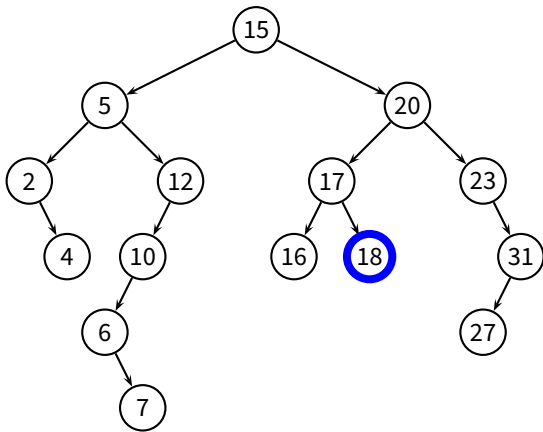
## Recap on Deletion in Binary Trees



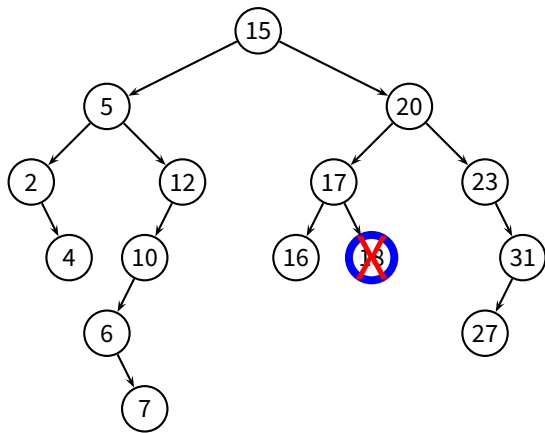


## Recap on Deletion in Binary Trees

1. z has no children



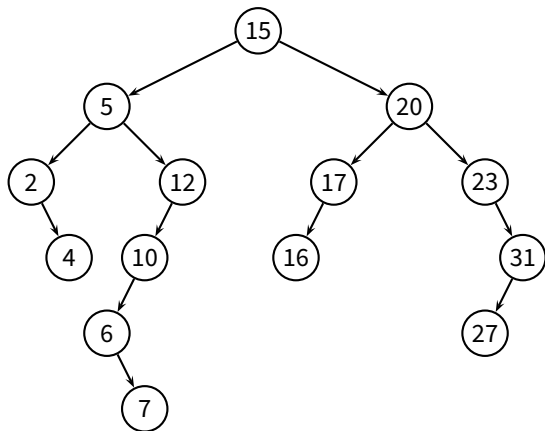
## Recap on Deletion in Binary Trees



1. z has no children

▶ simply remove z

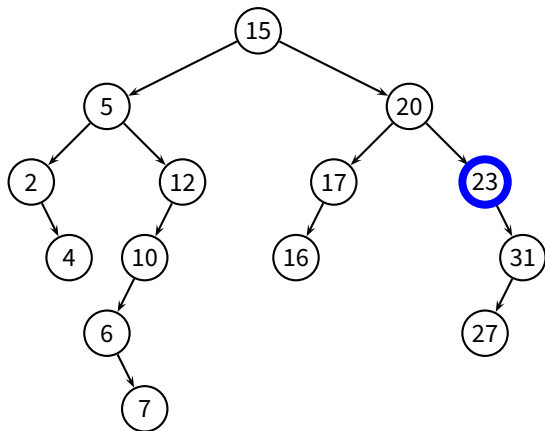
## Recap on Deletion in Binary Trees



1. z has no children

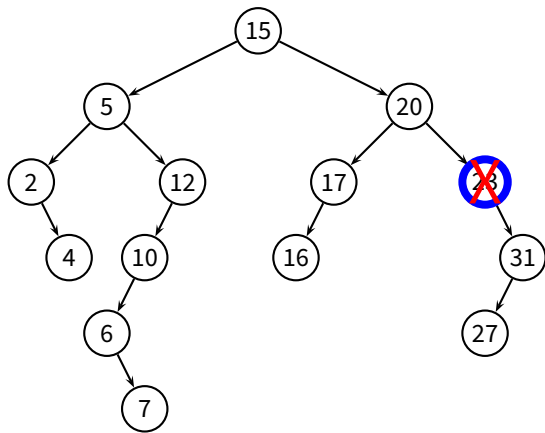
- ▶ simply remove z

## Recap on Deletion in Binary Trees



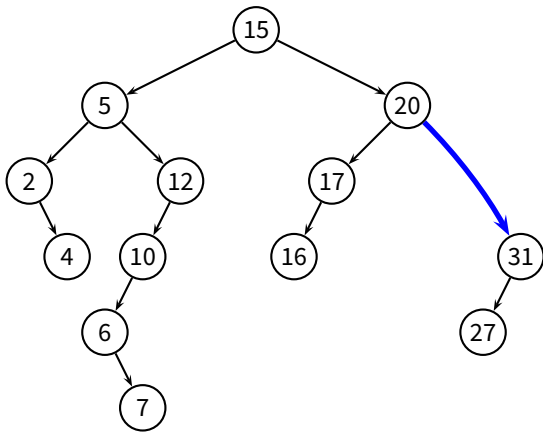
1. z has no children
  - ▶ simply remove z
2. z has one child

## Recap on Deletion in Binary Trees



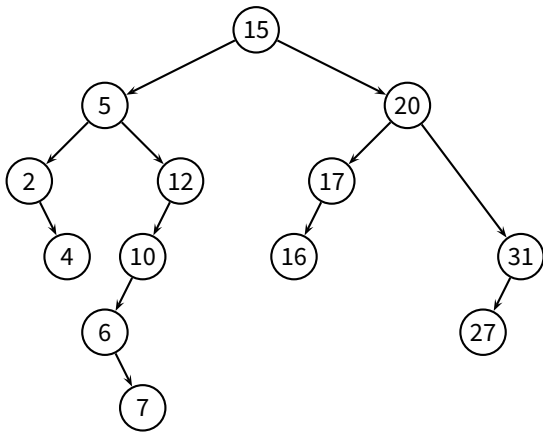
1. z has no children
  - ▶ simply remove z
2. z has one child
  - ▶ remove z

## Recap on Deletion in Binary Trees



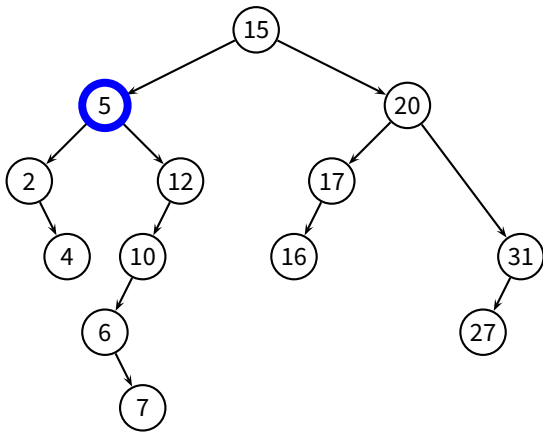
1.  $z$  has no children
  - ▶ simply remove  $z$
2.  $z$  has one child
  - ▶ remove  $z$
  - ▶ connect  $z.parent$  to  $z.right$

## Recap on Deletion in Binary Trees



1.  $z$  has no children
  - ▶ simply remove  $z$
2.  $z$  has one child
  - ▶ remove  $z$
  - ▶ connect  $z.parent$  to  $z.right$

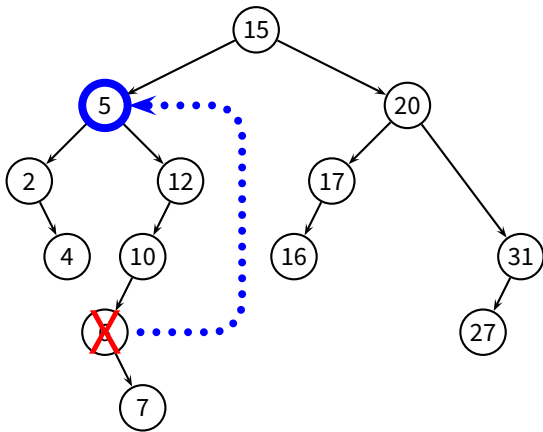
## Recap on Deletion in Binary Trees



1.  $z$  has no children
  - ▶ simply remove  $z$
2.  $z$  has one child
  - ▶ remove  $z$
  - ▶ connect  $z.parent$  to  $z.right$
3.  $z$  has two children

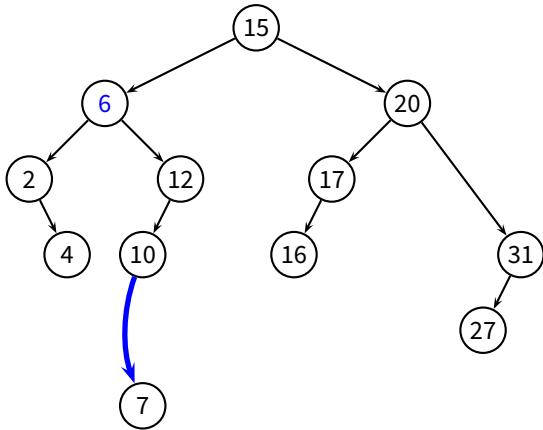


# Recap on Deletion in Binary Trees



1.  $z$  has no children
  - ▶ simply remove  $z$
2.  $z$  has one child
  - ▶ remove  $z$
  - ▶ connect  $z.parent$  to  $z.right$
3.  $z$  has two children
  - ▶ replace  $z$  with  $y = \mathbf{TREE-SUCCESSOR}(z)$
  - ▶ remove  $y$  (1 child!)

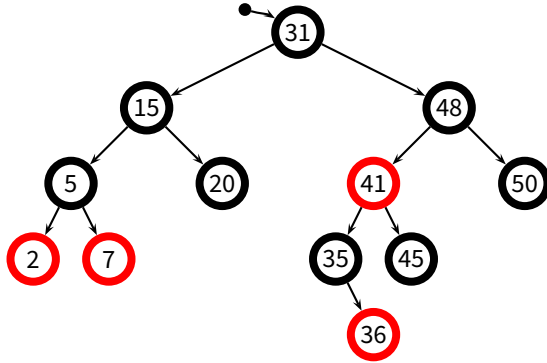
# Recap on Deletion in Binary Trees



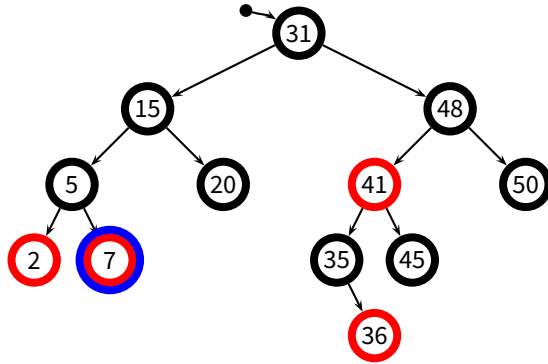
1.  $z$  has no children
  - ▶ simply remove  $z$
2.  $z$  has one child
  - ▶ remove  $z$
  - ▶ connect  $z.parent$  to  $z.right$
3.  $z$  has two children
  - ▶ replace  $z$  with  $y = \mathbf{TREE-SUCCESSOR}(z)$
  - ▶ remove  $y$  (1 child!)
  - ▶ connect  $y.parent$  to  $y.right$



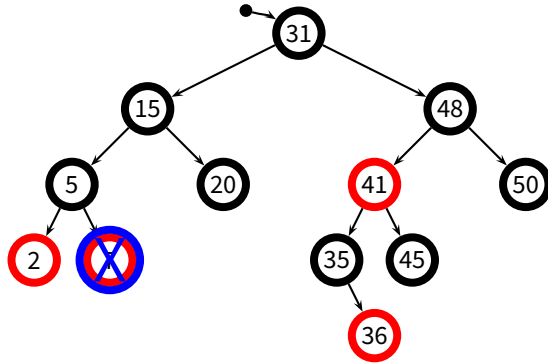
# Red-Black Deletion



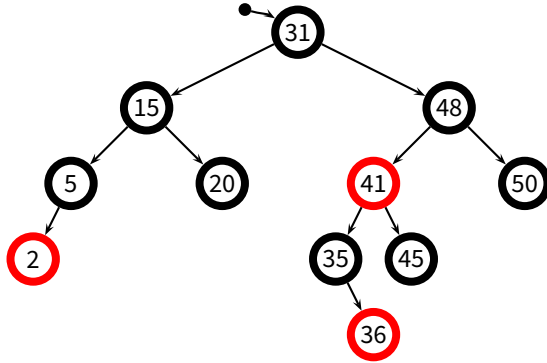
# Red-Black Deletion

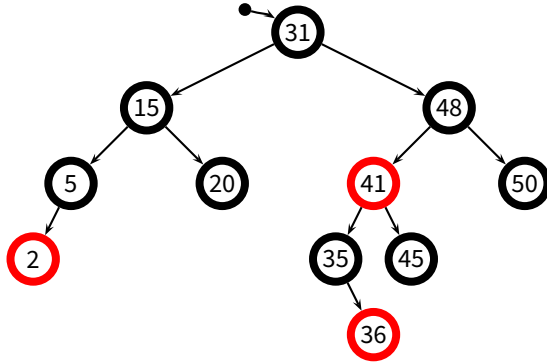


# Red-Black Deletion



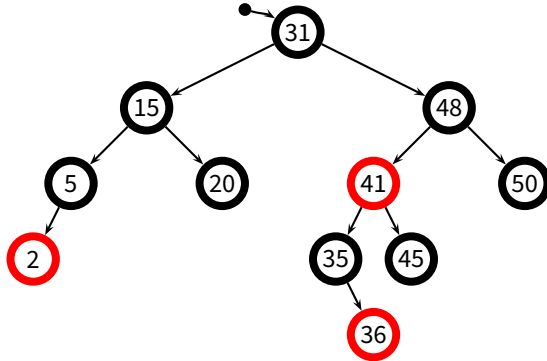
# Red-Black Deletion



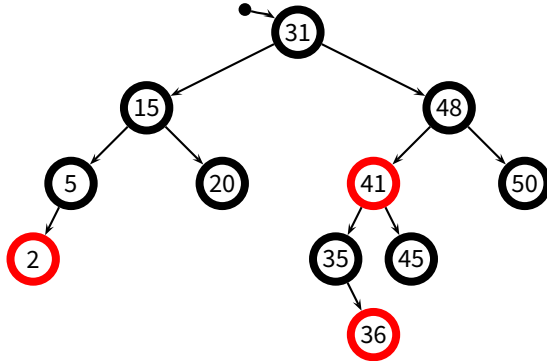


- A deleting a **red leaf** does not require any adjustment



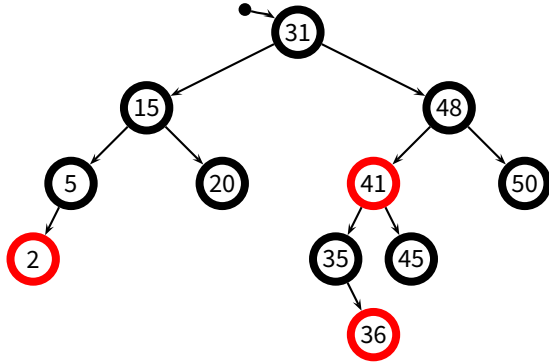


- A deleting a **red leaf** does not require any adjustment
  - ▶ the deletion does not affect the black height (property 5)

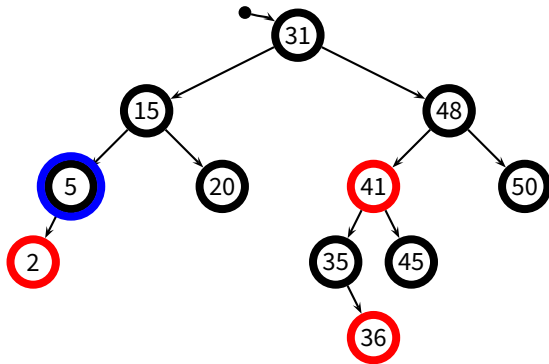


- A deleting a **red leaf** does not require any adjustment
  - ▶ the deletion does not affect the black height (property 5)
  - ▶ no two red nodes become adjacent (property 4)

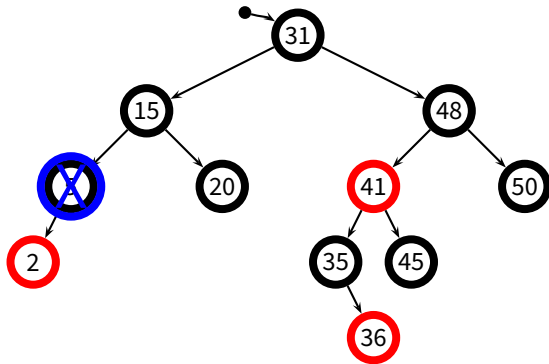
## Red-Black Deletion (2)



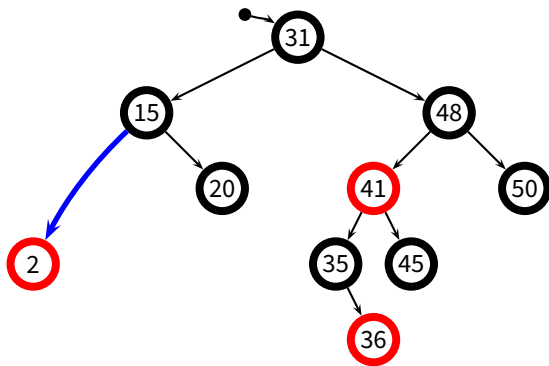
## Red-Black Deletion (2)



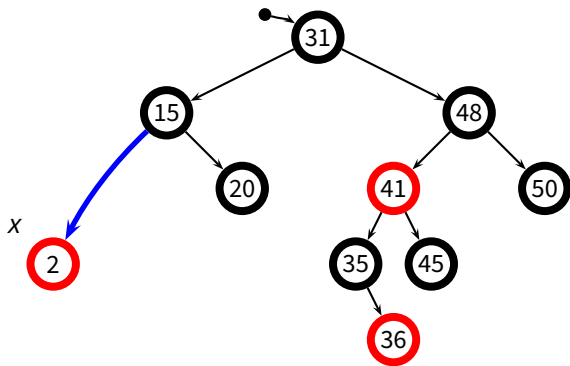
## Red-Black Deletion (2)



## Red-Black Deletion (2)

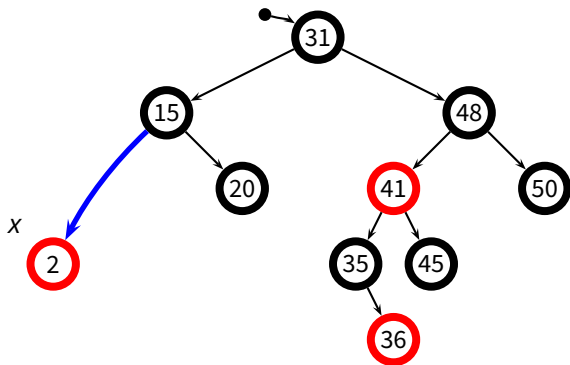


## Red-Black Deletion (2)



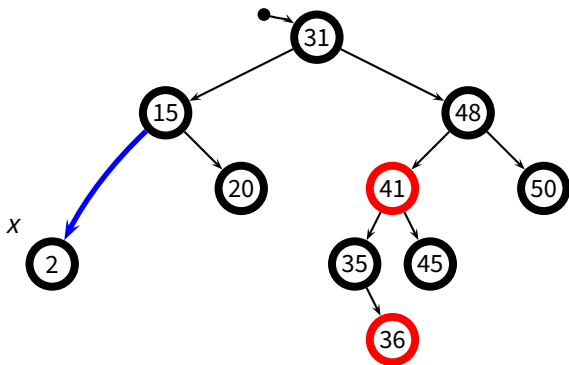
- Deleting a **black** node changes the balance of black-height in a subtree  $x$

## Red-Black Deletion (2)



- Deleting a **black** node changes the balance of black-height in a subtree  $x$ 
  - ▶ **RB-DELETE-FIXUP**( $T, x$ ) fixes the tree after a deletion





- Deleting a **black** node changes the balance of black-height in a subtree  $x$ 
  - ▶ **RB-DELETE-FIXUP**( $T, x$ ) fixes the tree after a deletion
  - ▶ in this simple case:  $x.color = \text{BLACK}$



- $y$  is the spliced node ( $y = z$  if  $z$  has zero or one child)
  - ▶ if  $y$  is **red**, then no fixup is necessary
  - ▶ so, here we assume that  $y$  is **black**

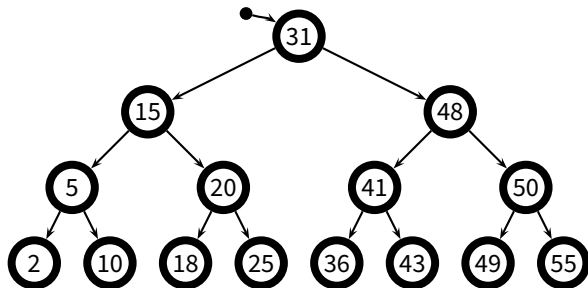
- $y$  is the spliced node ( $y = z$  if  $z$  has zero or one child)
  - ▶ if  $y$  is **red**, then no fixup is necessary
  - ▶ so, here we assume that  $y$  is **black**
- $x$  is either  $y$ 's only child or  $T.nil$ 
  - ▶  $y$  was spliced out, so  $y$  can not have two children
  - ▶  $x = T.nil$  iff  $y$  has no (key-bearing) children

- $y$  is the spliced node ( $y = z$  if  $z$  has zero or one child)
  - ▶ if  $y$  is **red**, then no fixup is necessary
  - ▶ so, here we assume that  $y$  is **black**
- $x$  is either  $y$ 's only child or  $T.nil$ 
  - ▶  $y$  was spliced out, so  $y$  can not have two children
  - ▶  $x = T.nil$  iff  $y$  has no (key-bearing) children
- **Problem 1:**  $y = T.root$  and  $x$  is **red**
  - ▶ violates red-black property 2 ( $root$  must be **black**)

- $y$  is the spliced node ( $y = z$  if  $z$  has zero or one child)
  - ▶ if  $y$  is **red**, then no fixup is necessary
  - ▶ so, here we assume that  $y$  is **black**
- $x$  is either  $y$ 's only child or  $T.nil$ 
  - ▶  $y$  was spliced out, so  $y$  can not have two children
  - ▶  $x = T.nil$  iff  $y$  has no (key-bearing) children
- **Problem 1:**  $y = T.root$  and  $x$  is **red**
  - ▶ violates red-black property 2 ( $root$  must be **black**)
- **Problem 2:** both  $x$  and  $y.parent$  are **red**
  - ▶ violates red-black property 4 (no adjacent red nodes)

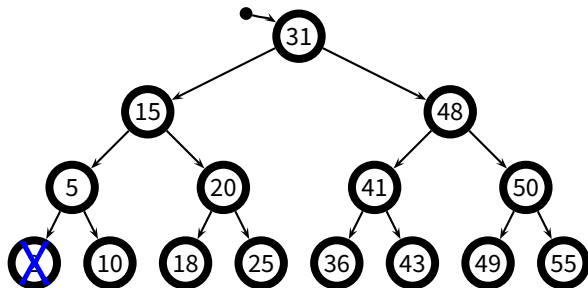
- $y$  is the spliced node ( $y = z$  if  $z$  has zero or one child)
  - ▶ if  $y$  is **red**, then no fixup is necessary
  - ▶ so, here we assume that  $y$  is **black**
- $x$  is either  $y$ 's only child or  $T.nil$ 
  - ▶  $y$  was spliced out, so  $y$  can not have two children
  - ▶  $x = T.nil$  iff  $y$  has no (key-bearing) children
- **Problem 1:**  $y = T.root$  and  $x$  is **red**
  - ▶ violates red-black property 2 ( $root$  must be **black**)
- **Problem 2:** both  $x$  and  $y.parent$  are **red**
  - ▶ violates red-black property 4 (no adjacent red nodes)
- **Problem 3:** we are removing  $y$ , which is black
  - ▶ violates red-black property 5 (same *black height* for all paths)

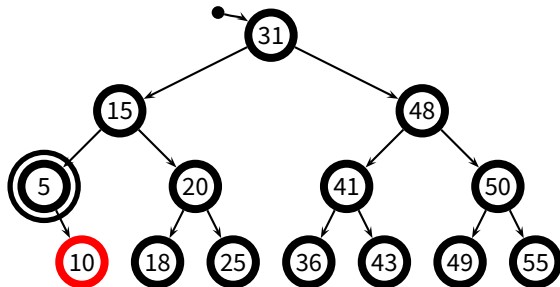
## Red-Black Deletion (3)



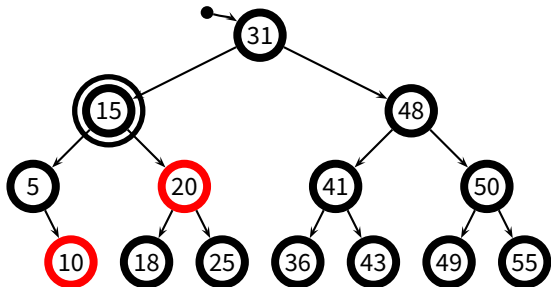


## Red-Black Deletion (3)

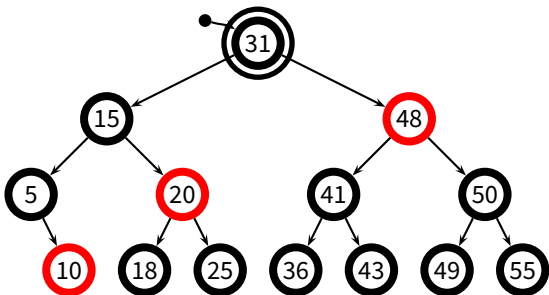




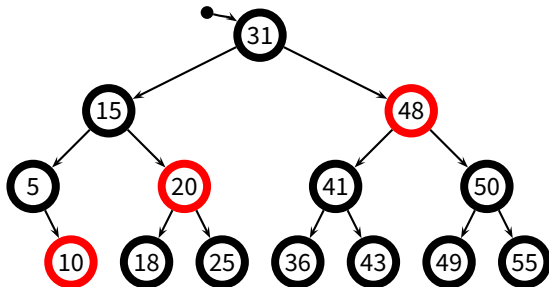
- $x$  carries an *additional black weight*
  - ▶ the fixup algorithm pushes it up towards to root



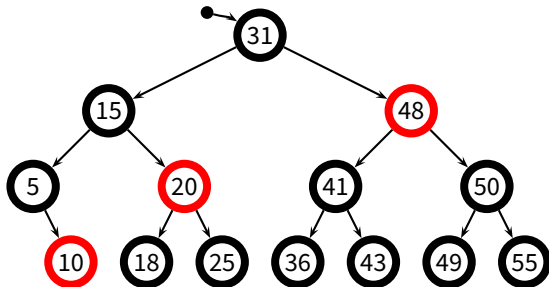
- $x$  carries an *additional black weight*
  - ▶ the fixup algorithm pushes it up towards to root



- $x$  carries an *additional black weight*
  - ▶ the fixup algorithm pushes it up towards to root

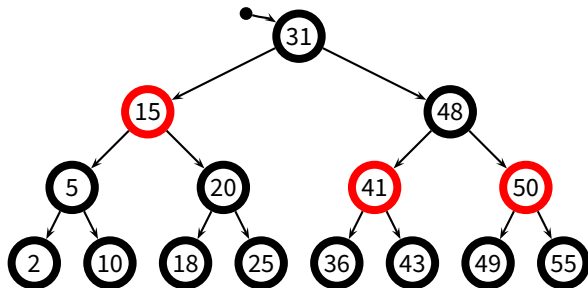


- $x$  carries an *additional black weight*
  - ▶ the fixup algorithm pushes it up towards to root

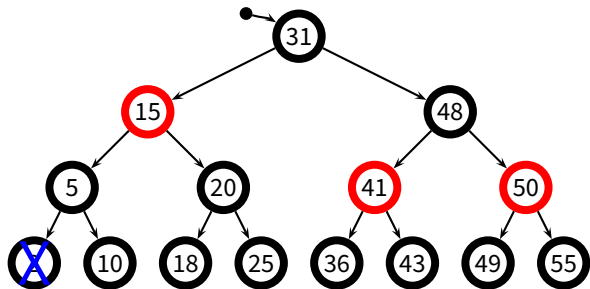


- $x$  carries an *additional **black** weight*
  - ▶ the fixup algorithm pushes it up towards to root
- The *additional **black** weight* can be discarded if it reaches the *root*, otherwise...

## Red-Black Deletion (4)

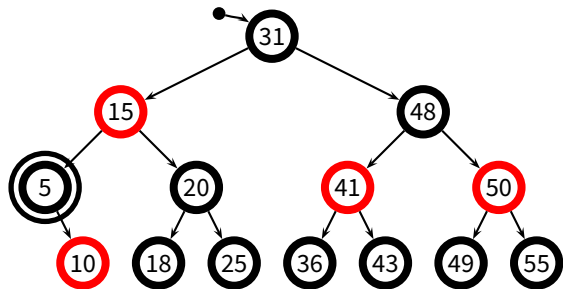


## Red-Black Deletion (4)

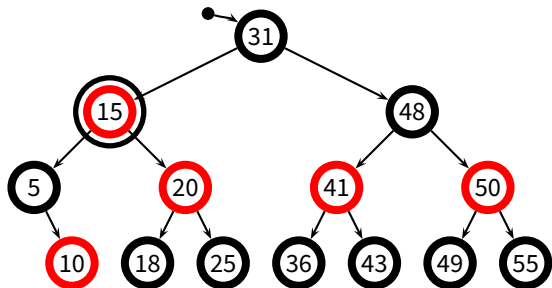




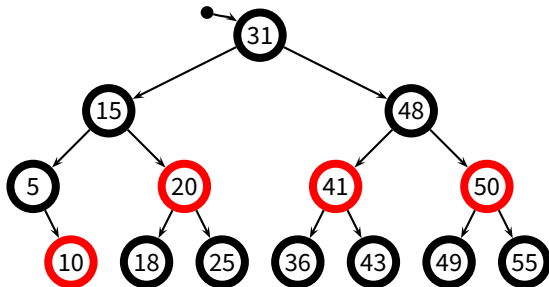
## Red-Black Deletion (4)



## Red-Black Deletion (4)

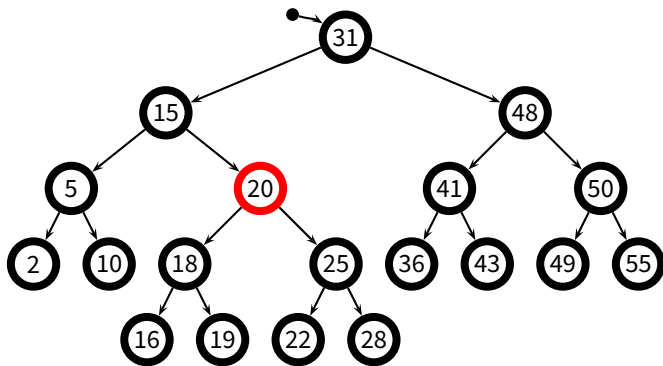


## Red-Black Deletion (4)

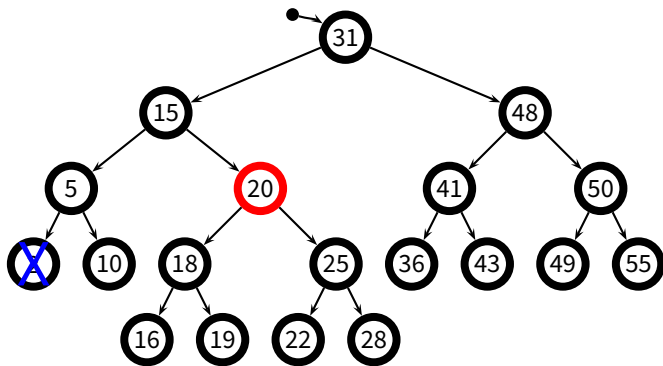


- The *additional black weight* can also stop as soon as it reaches a **red** node, which will absorb the extra **black** color

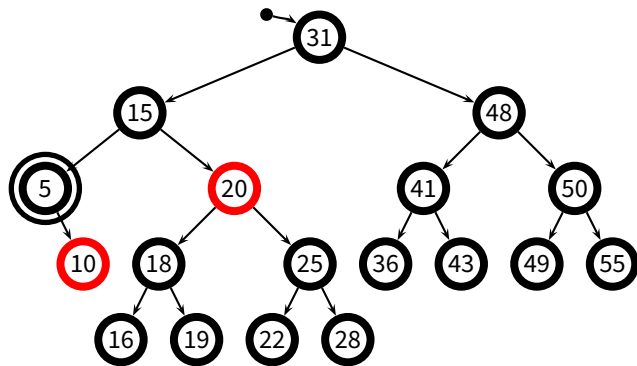
## Red-Black Deletion (5)



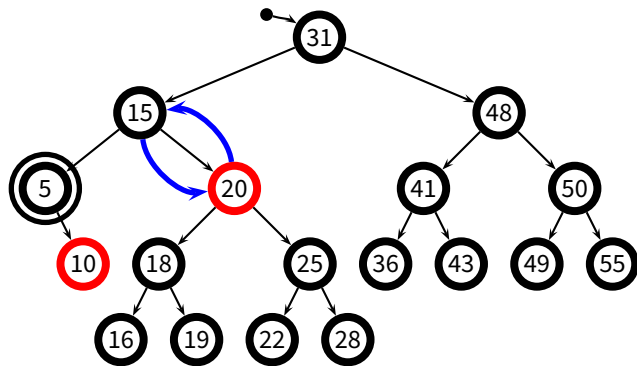
# Red-Black Deletion (5)



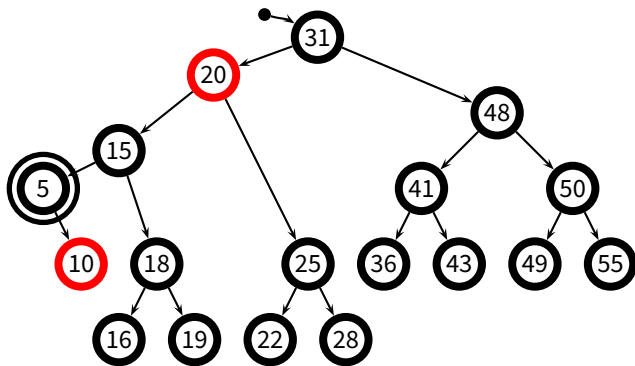
## Red-Black Deletion (5)



## Red-Black Deletion (5)

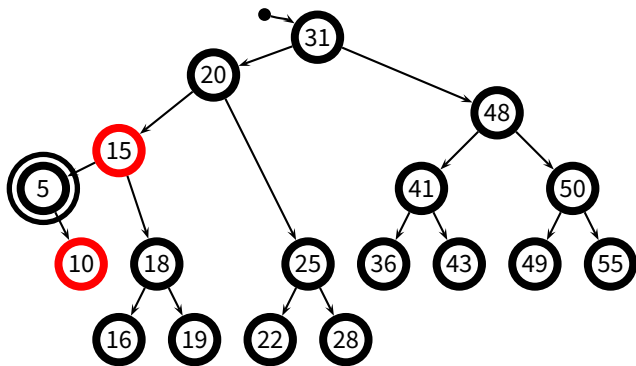


## Red-Black Deletion (5)

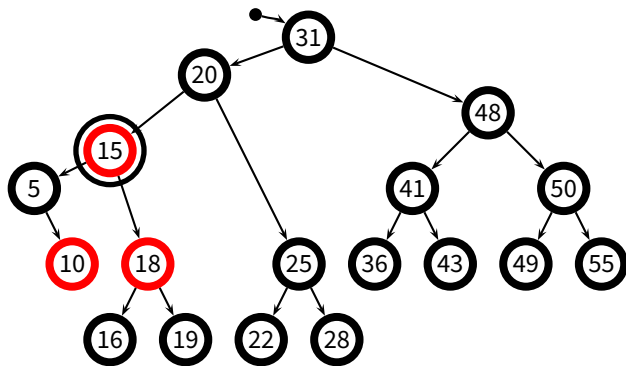




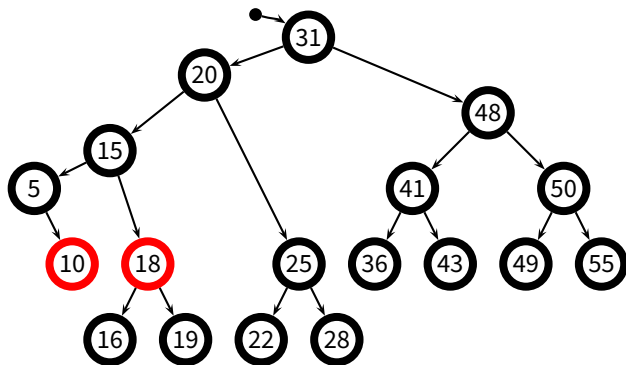
## Red-Black Deletion (5)



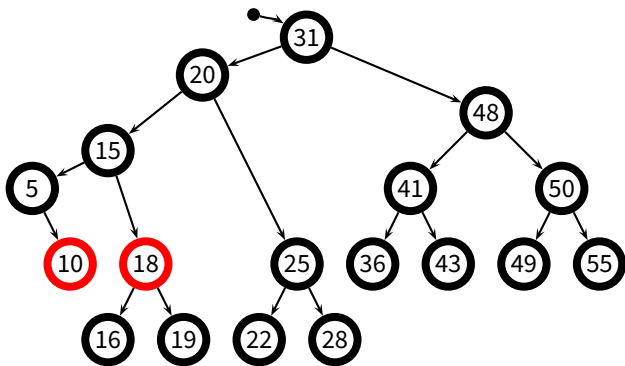
## Red-Black Deletion (5)



## Red-Black Deletion (5)



## Red-Black Deletion (5)



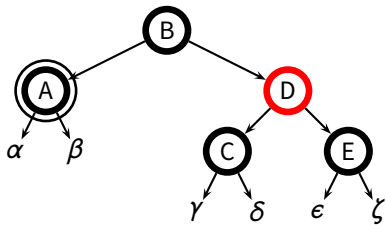
- In other cases where we can not push the additional black color up, we can apply appropriate rotations and color transfers that preserve all other red-black properties

# Basic Fixup Iteration (1)

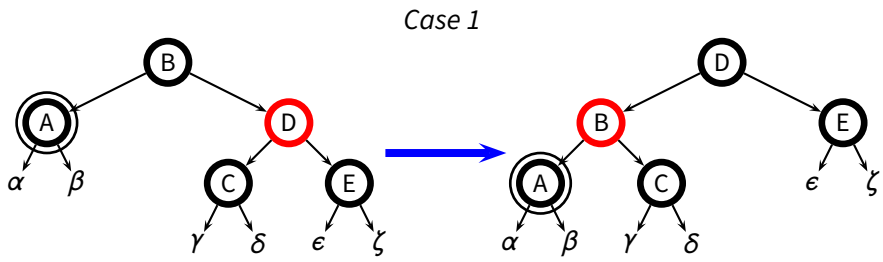
*Case 1*

# Basic Fixup Iteration (1)

Case 1

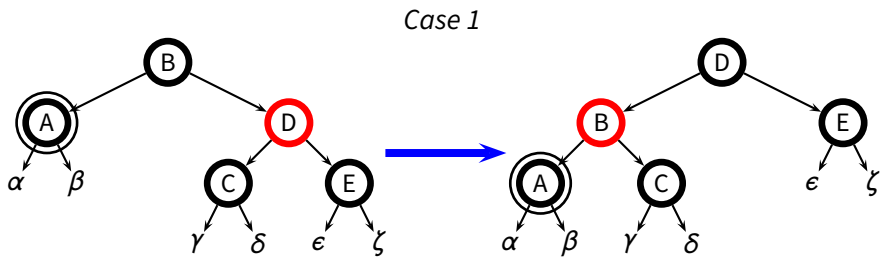


# Basic Fixup Iteration (1)



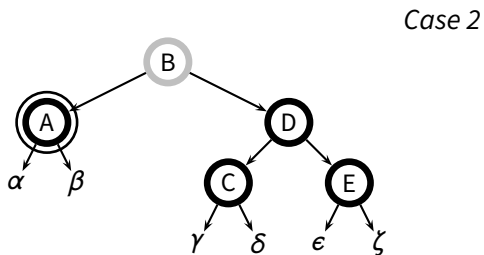
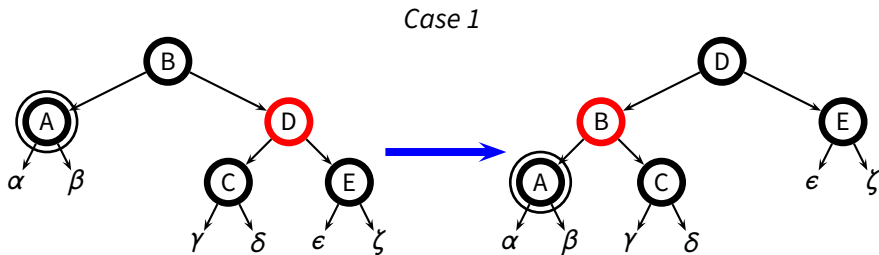


# Basic Fixup Iteration (1)

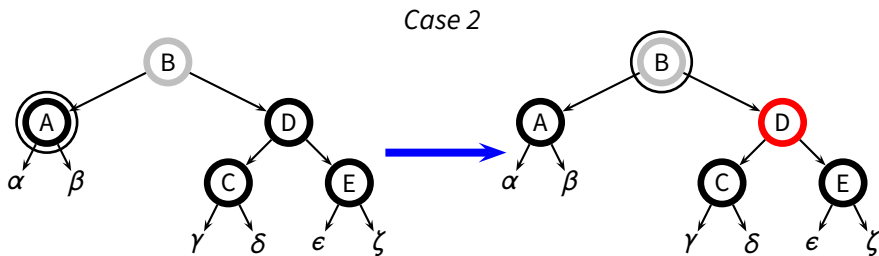
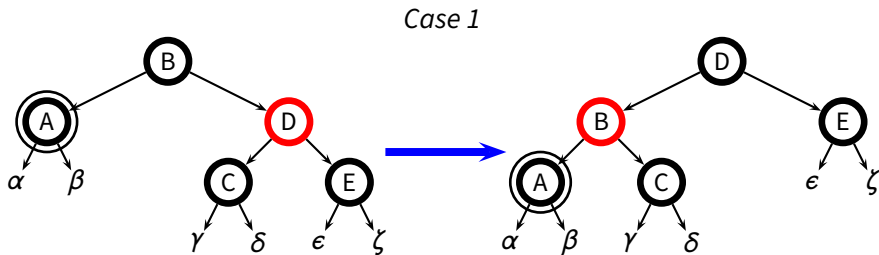


Case 2

# Basic Fixup Iteration (1)



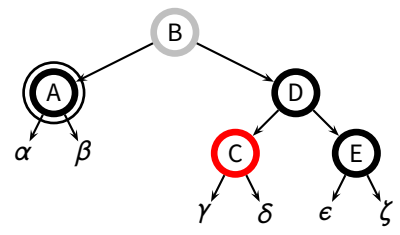
# Basic Fixup Iteration (1)



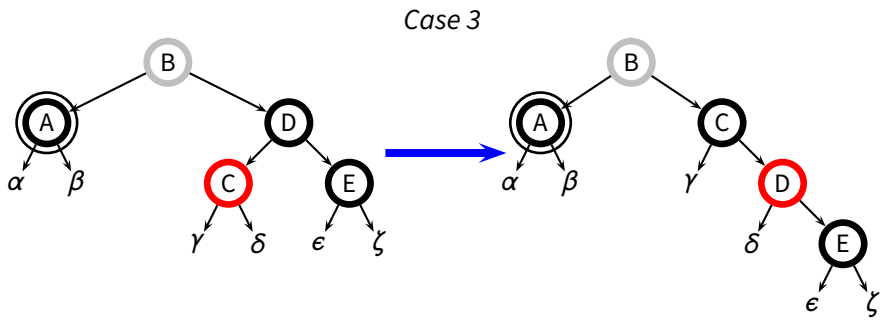
*Case 3*

# Basic Fixup Iteration (2)

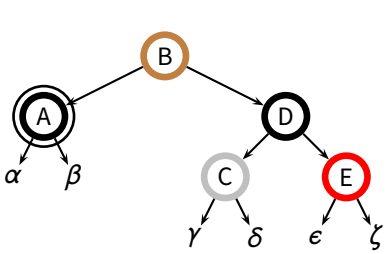
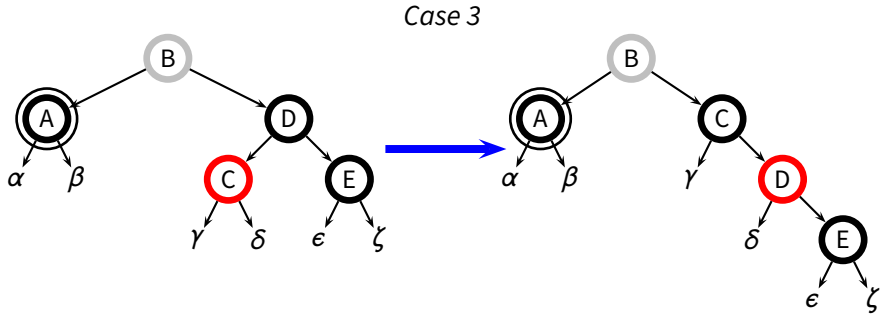
Case 3



# Basic Fixup Iteration (2)

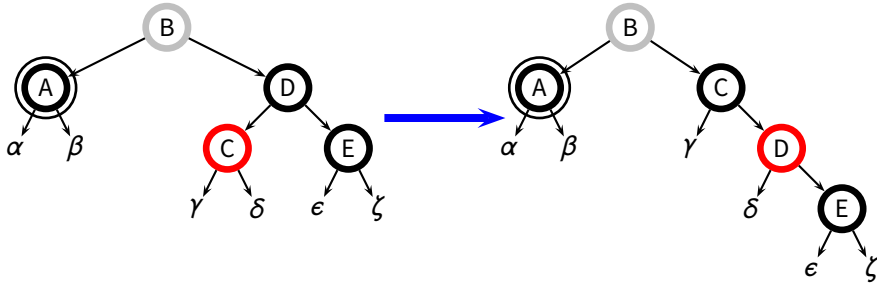


# Basic Fixup Iteration (2)

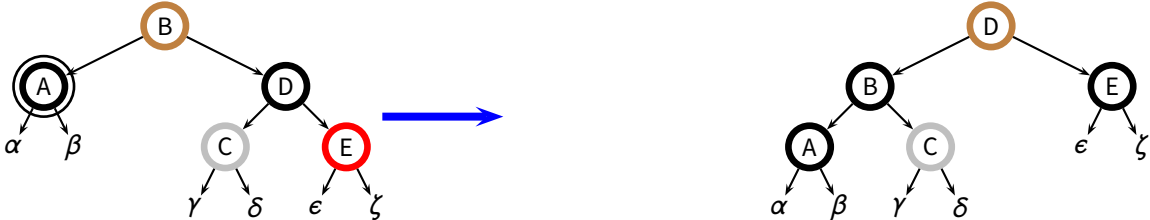


# Basic Fixup Iteration (2)

Case 3



Case 4





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

```
1  while  $x \neq T.root \wedge x.color = BLACK$ 
2      if  $x == x.parent.left$ 
3           $w = x.parent.right$ 
4          if  $w.color == RED$ 
5              case 1...
6          if  $w.left.color == BLACK \wedge w.right.color = BLACK$ 
7               $w.color = RED$  // case 2
8               $x = x.parent$ 
9          else if  $w.right.color == BLACK$ 
10             case 3...
11             case 4...
12         else same as above, exchanging right and left
13      $x.color = BLACK$ 
```