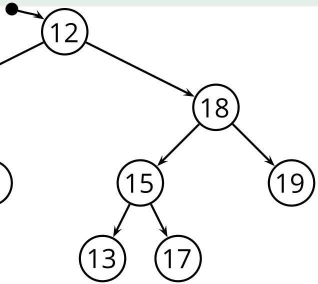
Algorithms and Data Structures (II)

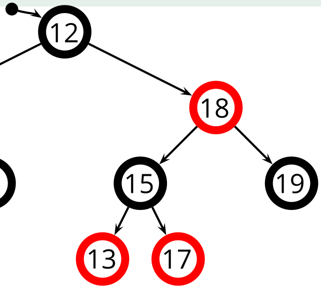
Gabriel Istrate

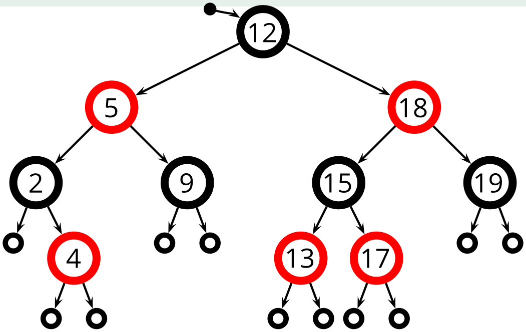
April 1, 2020

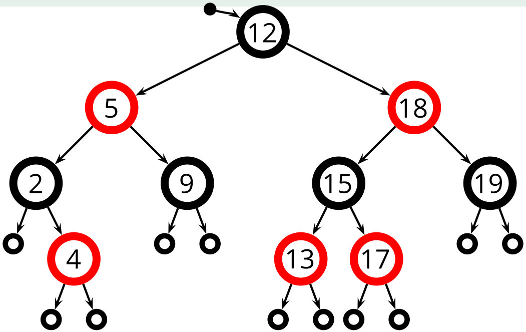
- Want: data structure to support INSERT, DELETE, SEARCH in $O(\log n)$ time.
- Binary search trees: insert, delete, search.
- But complexity bound not met unless trees balanced.

Today: wrap-up red-black-trees.

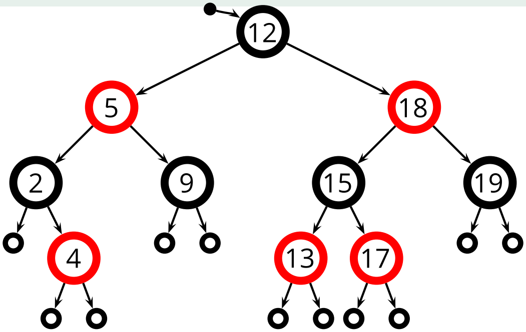

12
5
18
2
9
15
19
4
13
17


12
5
18
2
9
15
19
4
13
17

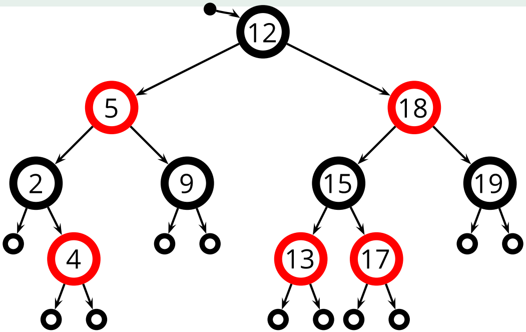

12
5
18
2
9
15
19
4
13
17



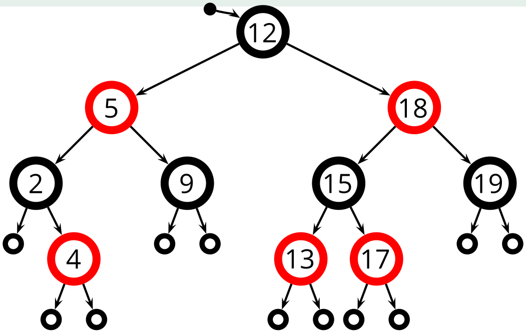
- *Red-black-tree property*



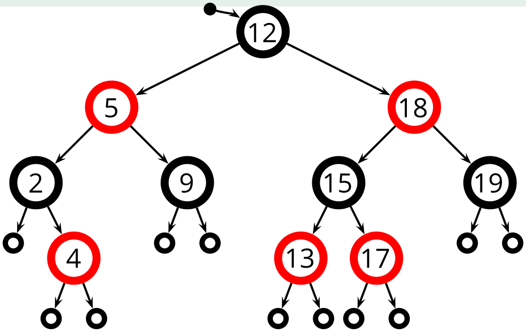
- *Red-black-tree property*
  1. every node is either **red** or **black**



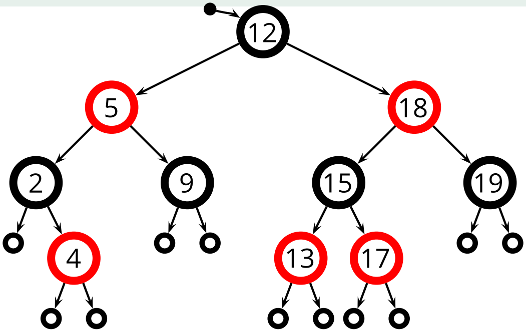
- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**



- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**

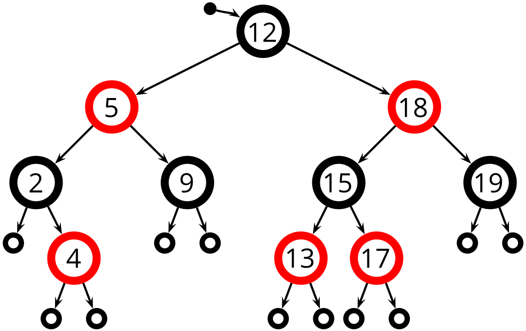


- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**

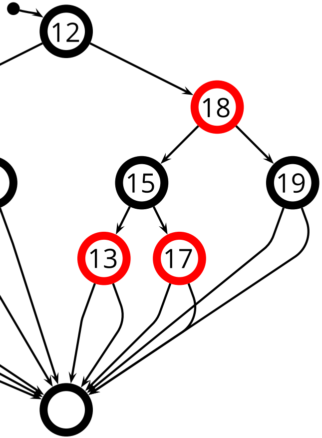


- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**
  5. for every node x, each path from x to its descendant leaves has the same number of **black** nodes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

- *Implementation*

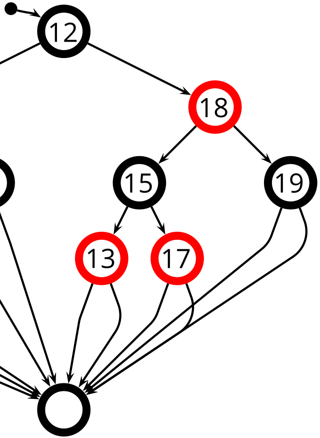


- *Implementation*



  - we use a common “sentinel” node to represent leaf nodes

- *Implementation*



- we use a common “sentinel” node to represent leaf nodes
- the sentinel is also the parent of the root node

- *Implementation*
  - T represents the tree, which consists of a set of *nodes*

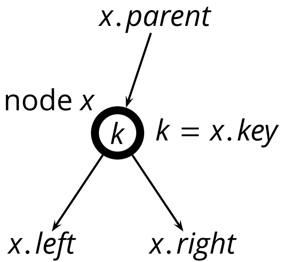
- *Implementation*
  - T represents the tree, which consists of a set of *nodes*
  - $T.root$ is the root node of tree T

- *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

- *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

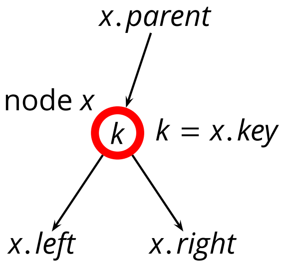
- $x.parent$ is the parent of node x
- $x.key$ is the key stored in node x
- $x.left$ is the left child of node x
- $x.right$ is the right child of node x



- *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



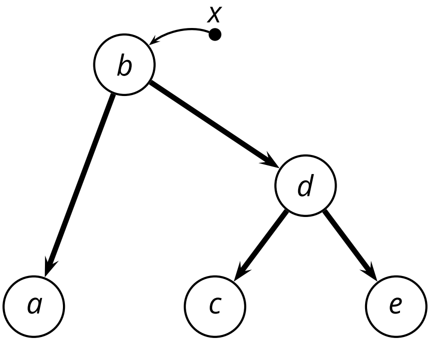
Lemma: the height $h(x)$ of a red-black tree with $n = size(x)$ internal nodes is at most $2\log(n+1)$.

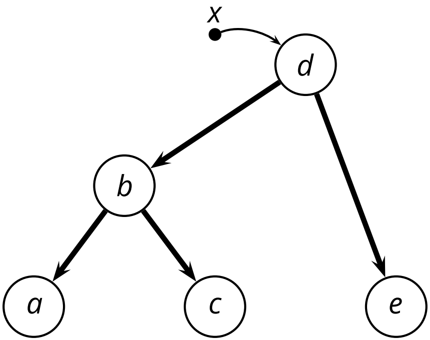
Lemma: the height $h(x)$ of a red-black tree with $n = size(x)$ internal nodes is at most $2\log(n+1)$.

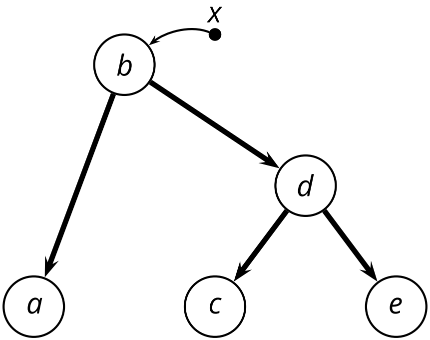
- A red-black tree works as a binary search tree for search, etc.
- So, the complexity of those operations is $T(n) = O(h)$, that is

$$\boxed{T(n) = O(\log n)}$$

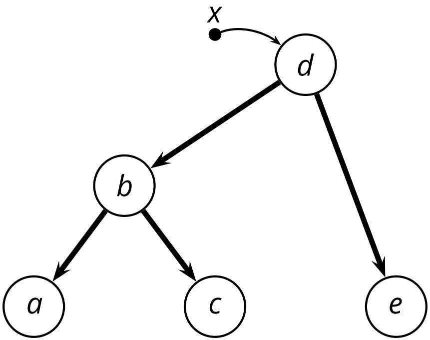
  - which is also the *worst-case* complexity


x
b
d
a
c
e


x
d
b
a
c
e



- $x = \textsc{Right-Rotate}(x)$



- $x = \textsc{Right-Rotate}(x)$
- $x = \textsc{Left-Rotate}(x)$

Reminder: rotate

```
LEFT-ROTATE(T, x)
y ← right[x]                      ▷ Set y.
right[x] ← left[y]                ▷ Turn y's left subtree into x's right subtree.
p[left[y]] ← x
p[y] ← p[x]                       ▷ Link x's parent to y.
if p[x] = nil[T]
   then root[T] ← y
   else  if x = left[p[x]]
            then left[p[x]] ← y
            else  right[p[x]] ← y
left[y] ← x                       ▷ Put x on y's left.
p[x] ← y
```

- **RB-INSERT**(T, z) works as in a binary search tree

- **RB-INSERT**(T, z) works as in a binary search tree
- Except that it must preserve the *red-black-tree property*

- **RB-INSERT**(T, z) works as in a binary search tree
- Except that it must preserve the *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*)

- **RB-INSERT**(T, z) works as in a binary search tree
- Except that it must preserve the *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)
- *General strategy*

- **RB-INSERT**(T, z) works as in a binary search tree
- Except that it must preserve the *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*)
- *General strategy*
  1. insert *z* as in a binary search tree
  2. color *z* **red** so as to preserve property 5
  3. *fix the tree* to correct possible violations of property 4

- Inserting z might violate red-black properties.

- Properties 1,3,5 hold (because z replaces black sentinel).
- Property 2 violated if z is the root.$\longrightarrow$ recolor root black
- Property 4 violated if z's parent is red.

- Will have to take into account the color of the uncle node
- Sibling of the parent node.
- Invariant: At the start of each iteration of the loop
    1. Node z is red.
    2. If $p[z]$ is the root then $p[z]$ is black.
    3. If there is a violation of R-B then there is at most one violation, of Property 2 or 4.
    4. If violation of property 2: because z is the root and is red.
    5. If violation of property 4: because both z and $p[z]$ red.

- Initialization: was red-black tree with no violations, inserted node z. Easy to see that invariant fixed.
- Termination: when loop terminates, $p[z]$ is black. Thus there is no violation of property 4 at loop termination.
- Line 16 restores property 2 too.
- Maintenance: six cases, symmetric. Three cases.
- Case 1: z's uncle is red.
- Case 2: z's uncle y is black and z is in-line.
- Case 3: z's uncle y is black and z is in zig-zag.

Violation: example and repair

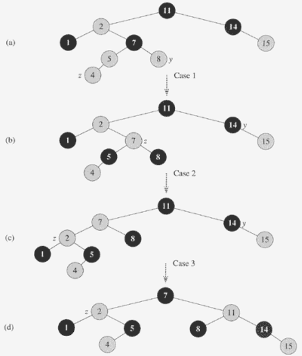


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Since z and its parent $p[z]$ are both red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code can be applied. Nodes are recolored and the pointer z is moved up the tree, resulting in the tree shown in **(b)**. Once again, z and its parent are both red, but z's uncle y is black. Since z is the right child of $p[z]$, case 2 can be applied. A left rotation is performed, and the tree that results is shown in **(c)**. Now z is the left child of its parent, and case 3 can be applied. A right rotation yields the tree in **(d)**, which is a legal red-black tree.

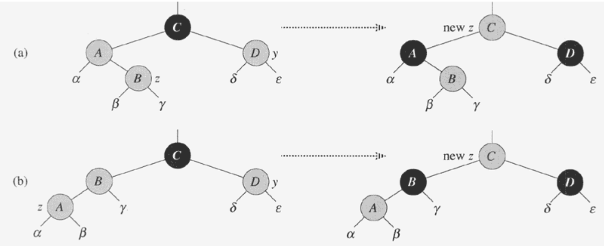


Figure 13.5 Case 1 of the procedure RB-INSERT. Property 4 is violated, since z and its parent $p[z]$ are both red. The same action is taken whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $p[p[z]]$ as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.

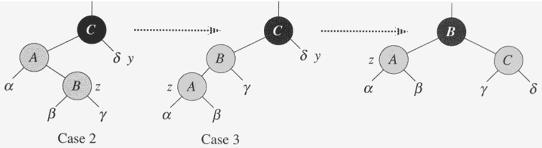


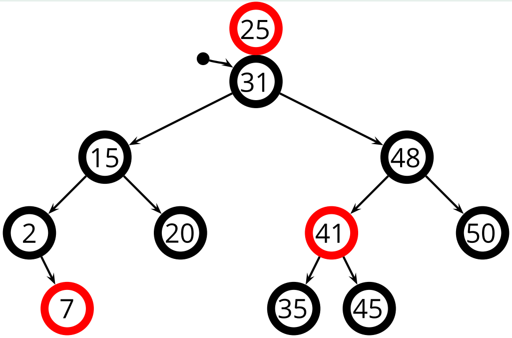
Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $p[z]$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 5: all downward paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

RB-Insert

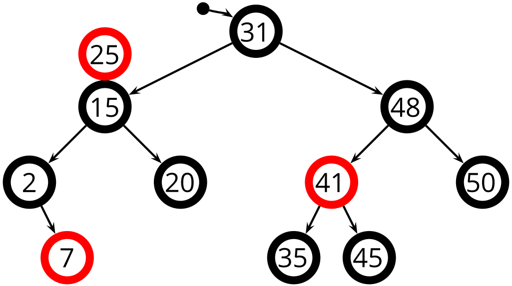
RB-Insert(T, z)

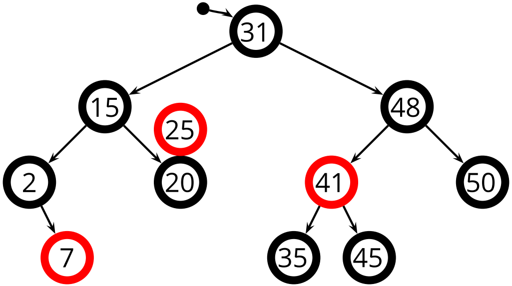
```
y = T.nil
x = T.root
while x ≠ T.nil
        y = x
        if z.key < x.key
                x = x.left
        else x = x.right
z.parent = y
if y == T.nil
        T.root = z
else if z.key < y.key
                y.left = z
        else y.right = z
z.left = z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T,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

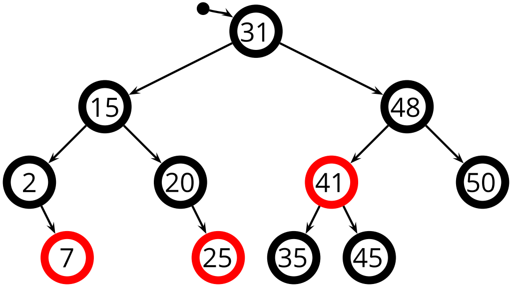
Red-Black Insertion (2)

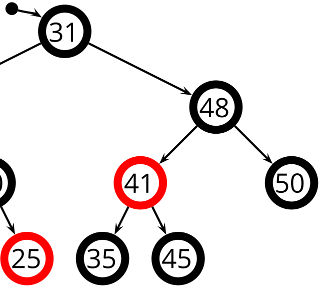


Red-Black Insertion (2)



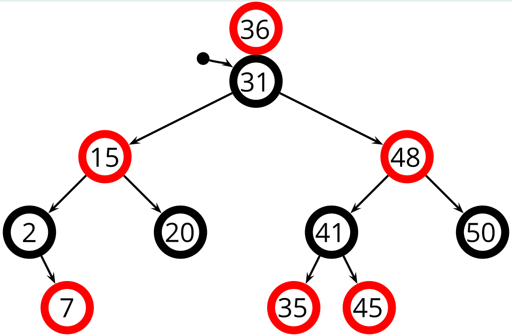

31
15
25
48
2
20
41
50
7
35
45

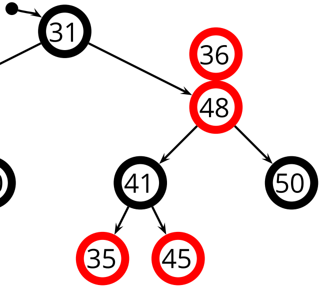

31
15
48
2
20
41
50
7
25
35
45

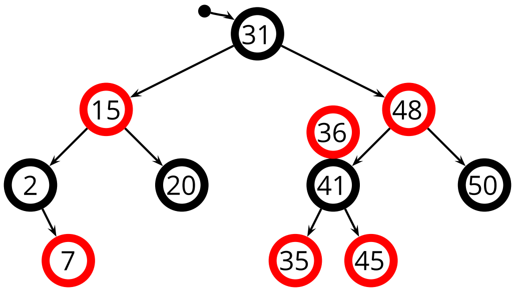


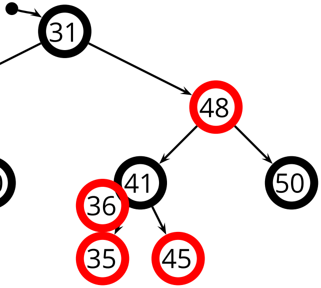
- z's father is **black**, so no fixup needed

Red-Black Insertion (3)

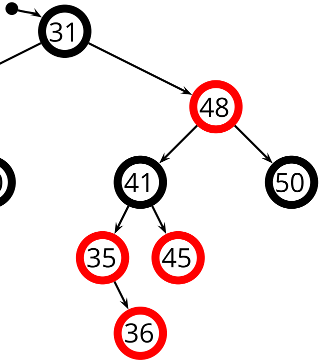

36
31
15
48
2
20
41
50
7
35
45

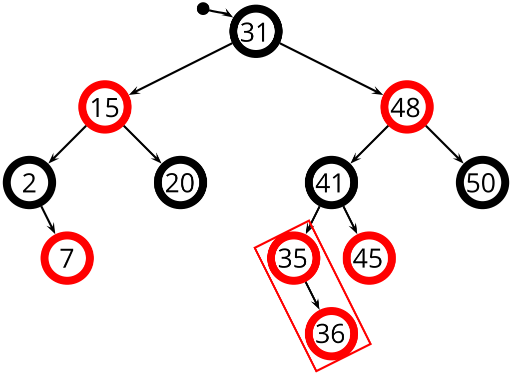

31
36
15
48
2
20
41
50
7
35
45

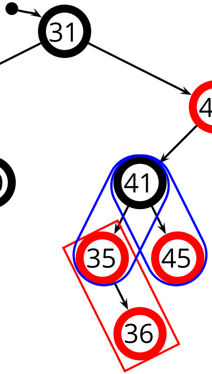

31
15
48
36
2
20
41
50
7
35
45


31
15
48
2
20
41
50
36
7
35
45

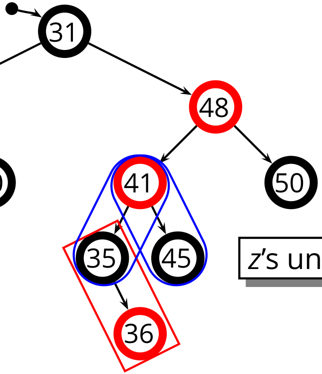
Red-Black Insertion (3)



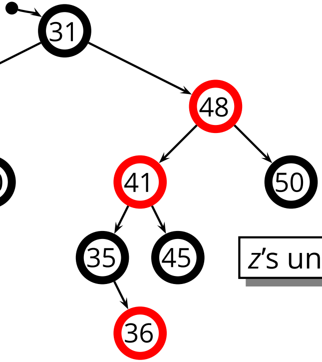

31
15
48
2
20
41
50
7
35
45
36



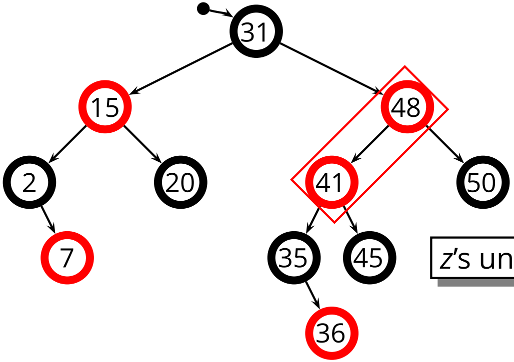
z's uncle is **red**: Case 1!



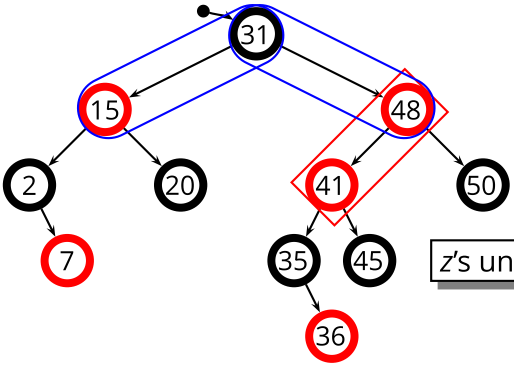
z's uncle is **red**: Case 1!



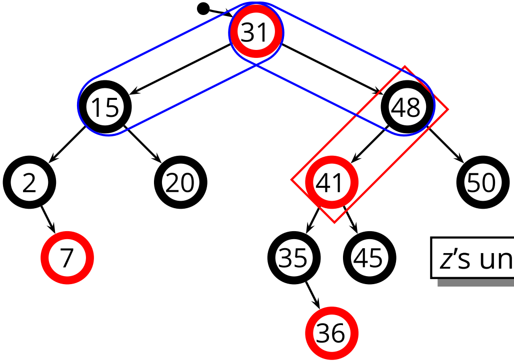
z's uncle is **red**: Case 1!



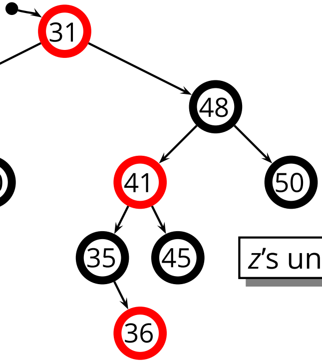
z's uncle is **red**: Case 1!



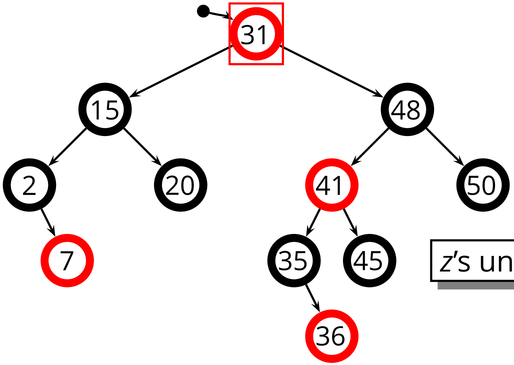
z's uncle is **red**: Case 1!



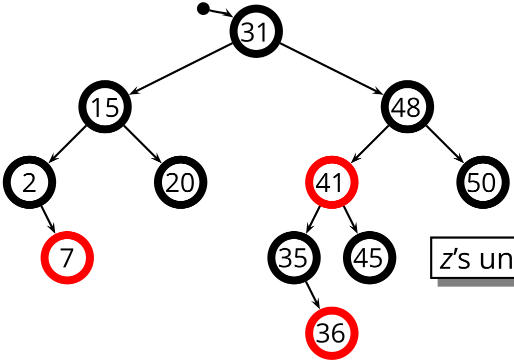
z's uncle is **red**: Case 1!



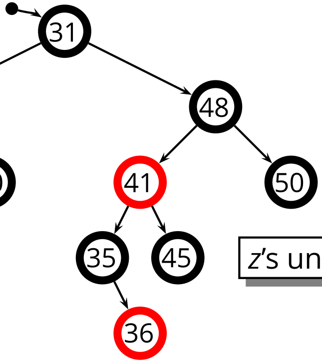
z's uncle is **red**: Case 1!



z's uncle is **red**: Case 1!

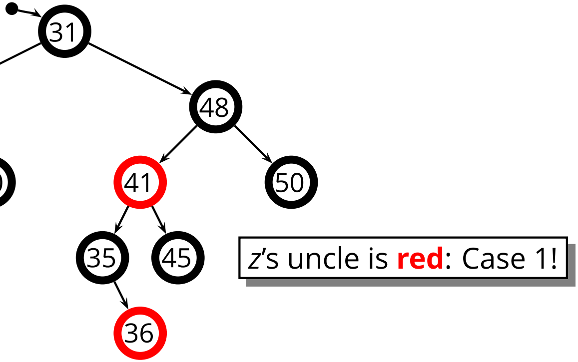


z's uncle is **red**: Case 1!

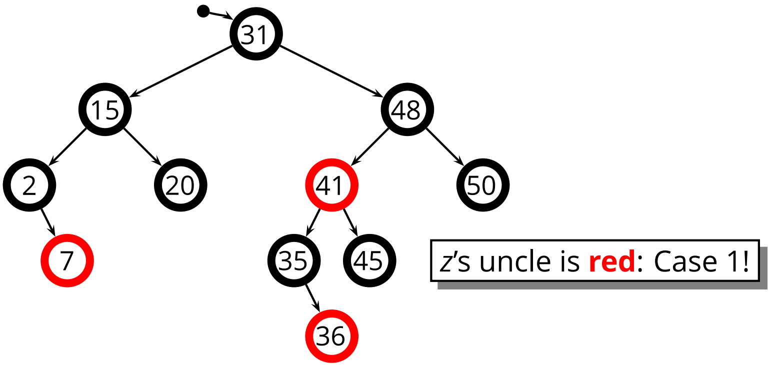


z's uncle is **red**: Case 1!

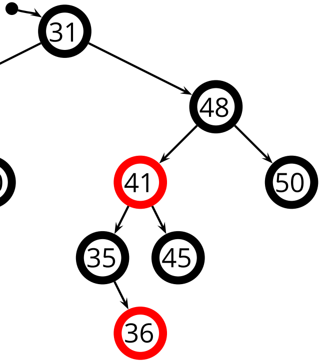
- A **black** node can become **red** and transfer its **black** color to its two children

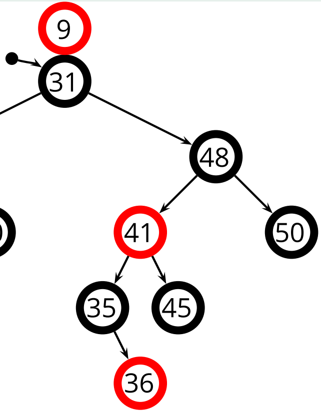


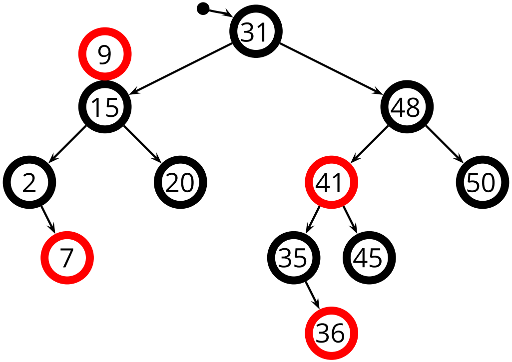
- A **black** node can become **red** and transfer its **black** color to its two children
- This may cause other **red–red** conflicts, so we iterate...



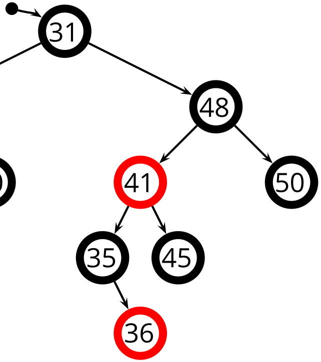
- A **black** node can become **red** and transfer its **black** color to its two children
- This may cause other **red–red** conflicts, so we iterate. . .
- The root can change to **black** without causing conflicts

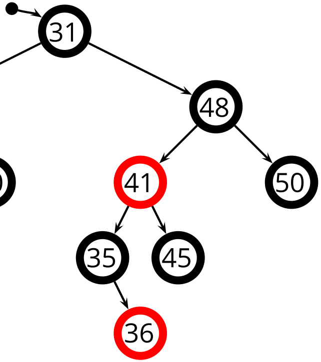

31
15
48
2
20
41
50
7
35
45
36

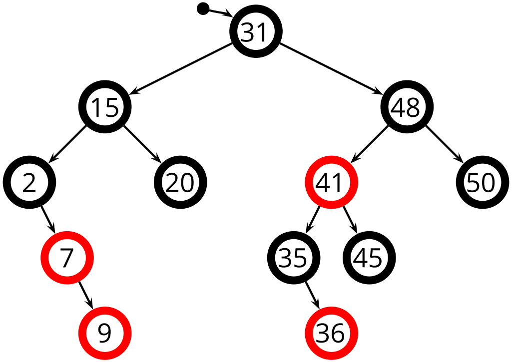

9
31
15
48
2
20
41
50
7
35
45
36

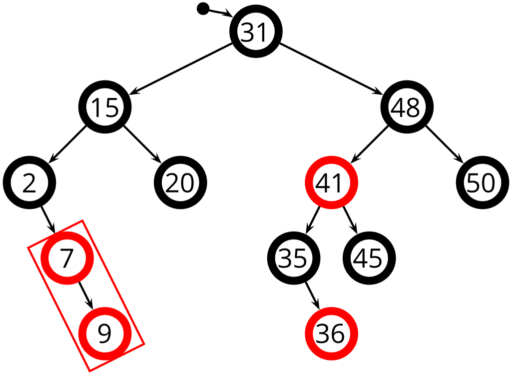

31
9
15
48
2
20
41
50
7
35
45
36

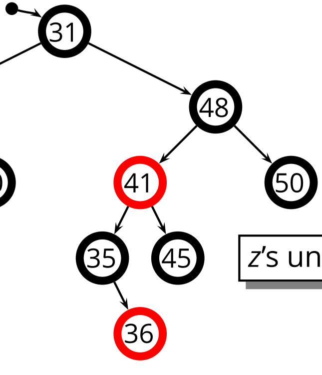
Red-Black Insertion (4)



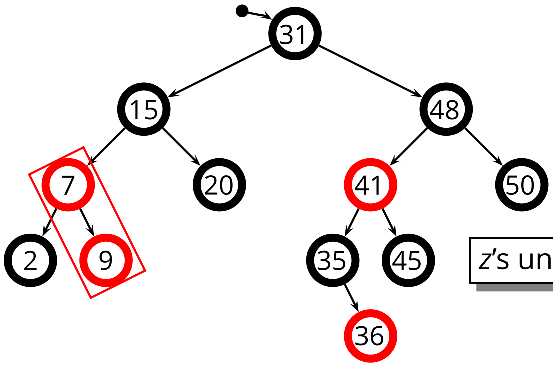

31
15
48
2
9
20
41
50
7
35
45
36


31
15
48
2
20
41
50
7
35
45
9
36

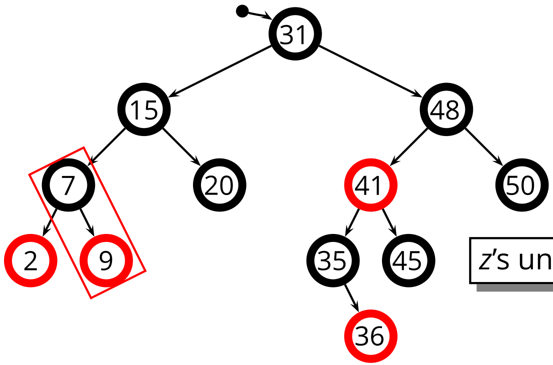

31
15
48
2
20
41
50
7
35
45
9
36



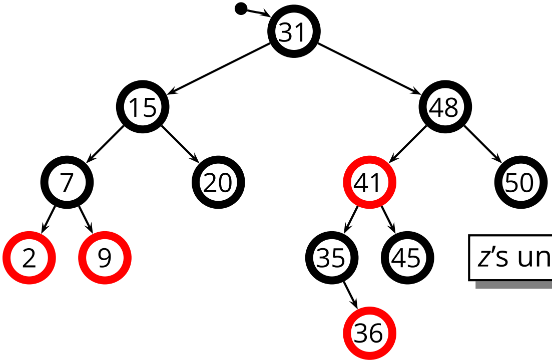
z's uncle is **black**. In-line ! (C.3)



z's uncle is **black**. In-line ! (C.3)



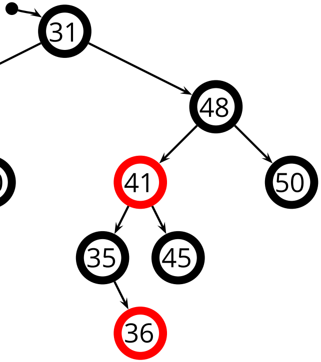
z's uncle is **black**. In-line ! (C.3)

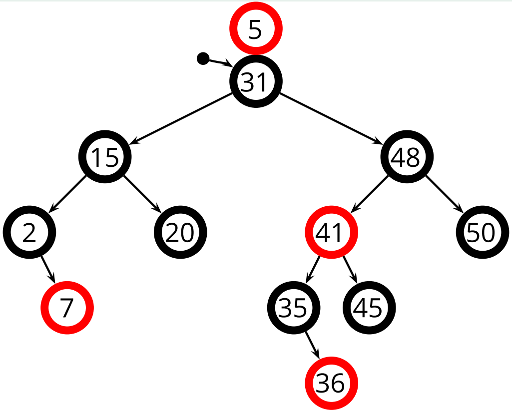


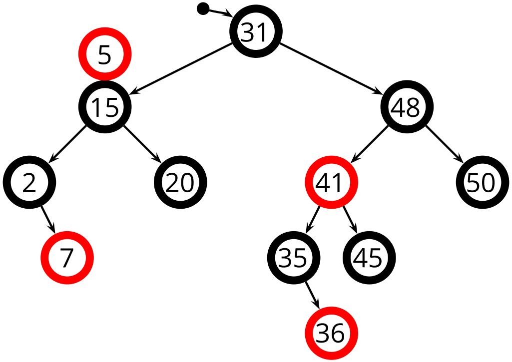
z's uncle is **black**. In-line ! (C.3)

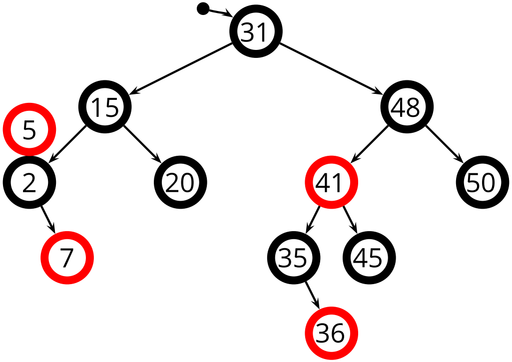
- An *in-line* ***red***–**red** conflicts can be resolved with a rotation plus a color switch

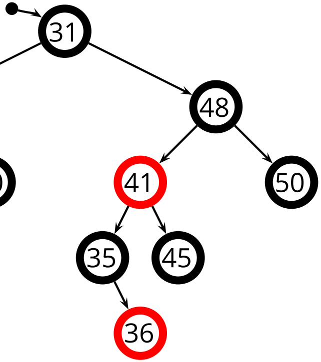
Red-Black Insertion (5)

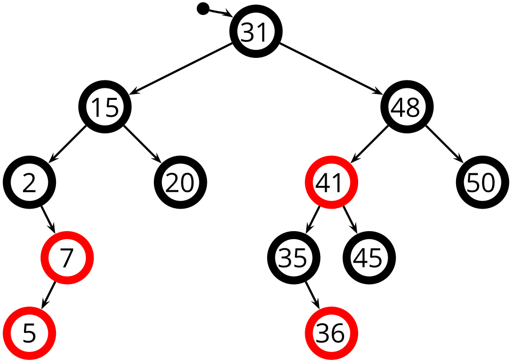


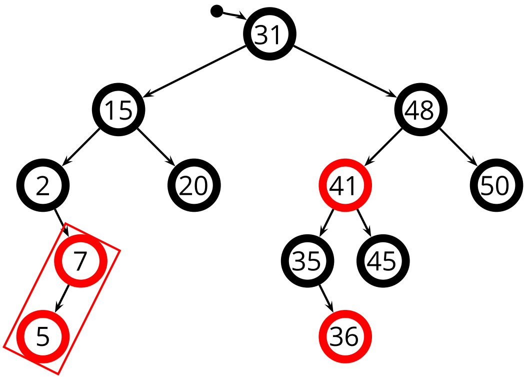

5
31
15
48
2
20
41
50
7
35
45
36

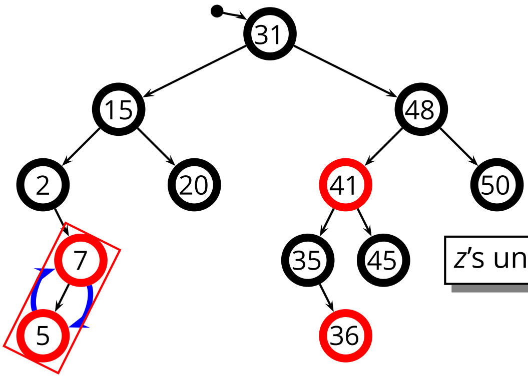

31
5
15
48
2
20
41
50
7
35
45
36


31
15
48
5
2
20
41
50
7
35
45
36

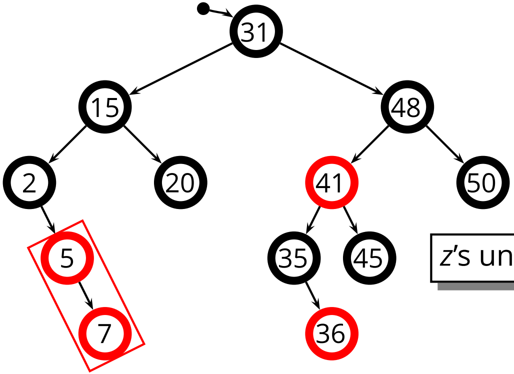

31
15
48
2
5
20
41
50
7
35
45
36


31
15
48
2
20
41
50
7
35
45
5
36

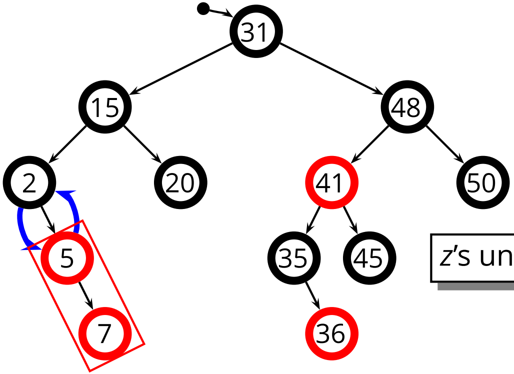

31
15
48
2
20
41
50
7
35
45
5
36



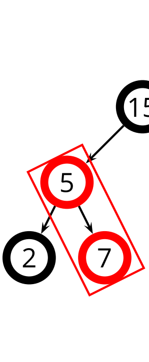
z's uncle is **black**. Zig-Zag (C.2)



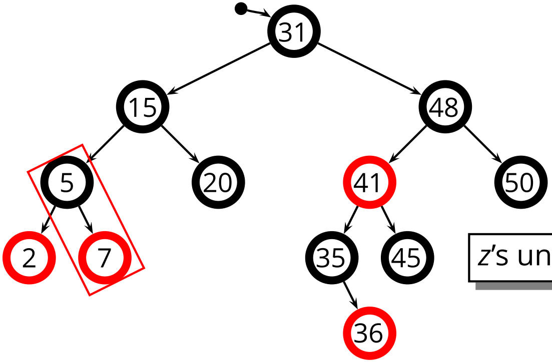
z's uncle is **black**. Zig-Zag (C.2)



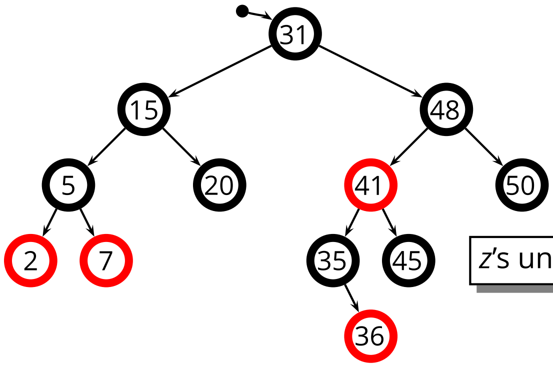
z's uncle is **black**. Zig-Zag (C.2)



z's uncle is **black**. Zig-Zag (C.2)

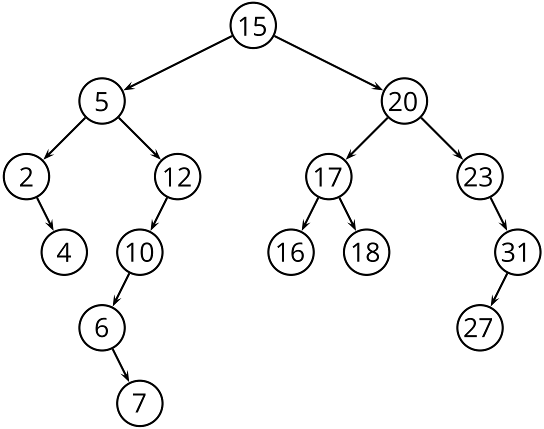


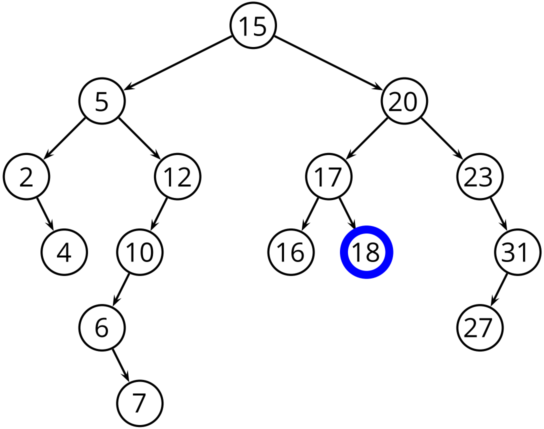
z's uncle is **black**. Zig-Zag (C.2)



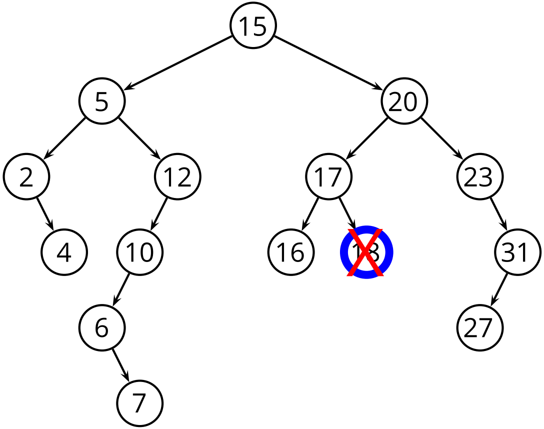
z's uncle is **black**. Zig-Zag (C.2)

- A *zig-zag* **red**–**red** conflict can be resolved with a rotation to turn it into an *in-line* conflict, and then a rotation plus a color switch

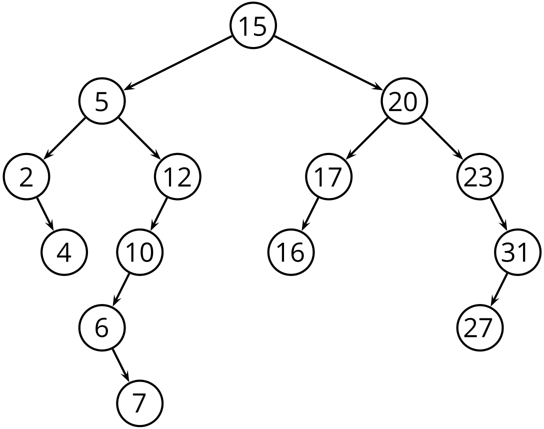

15
5
20
2
12
17
23
4
10
16
18
31
6
27
7



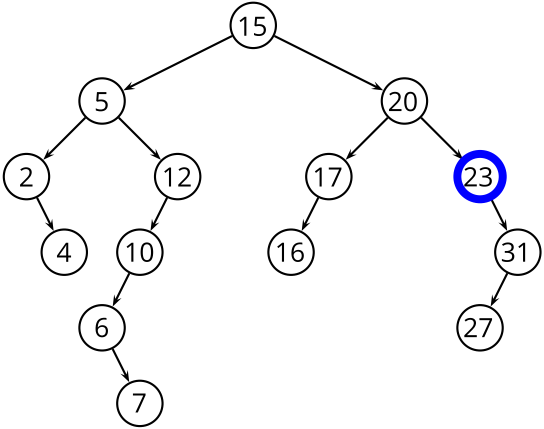
1. z has no children



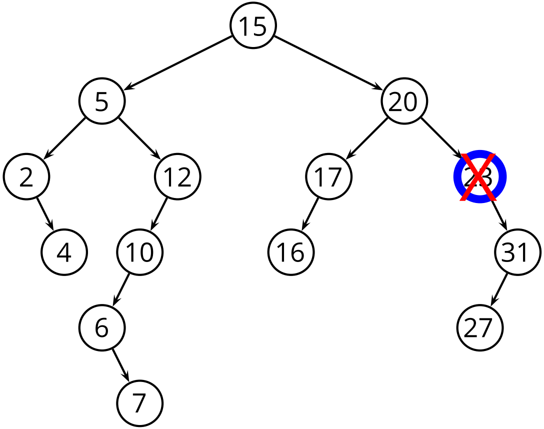
1. *z* has no children
   - simply remove *z*



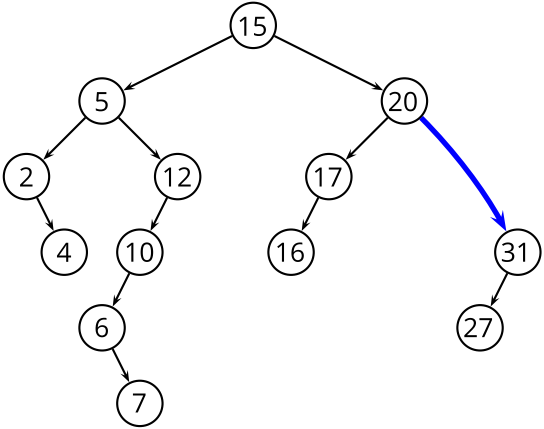
1. z has no children
   - simply remove z



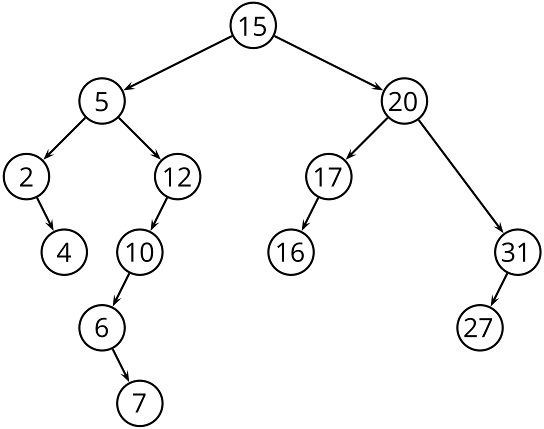
1. z has no children
   - simply remove z
2. z has one child



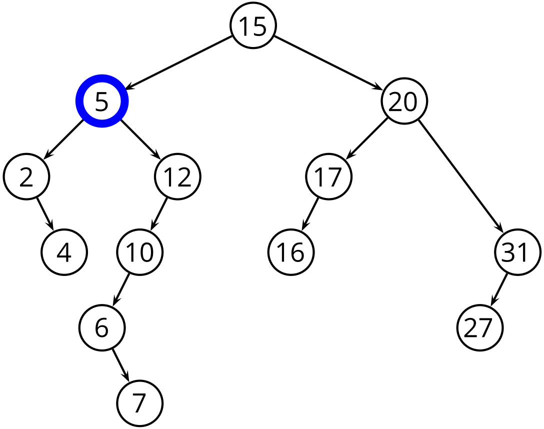
1. *z* has no children
   - simply remove *z*
2. *z* has one child
   - remove *z*



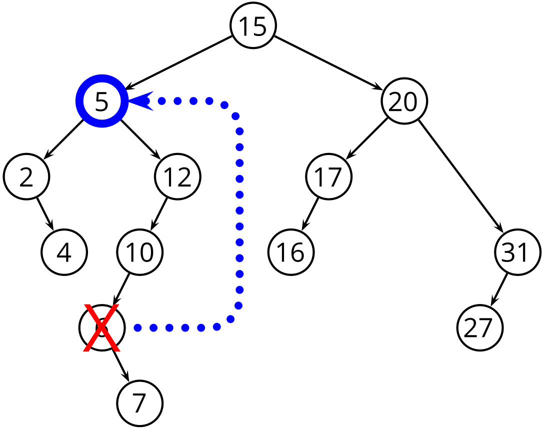
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$



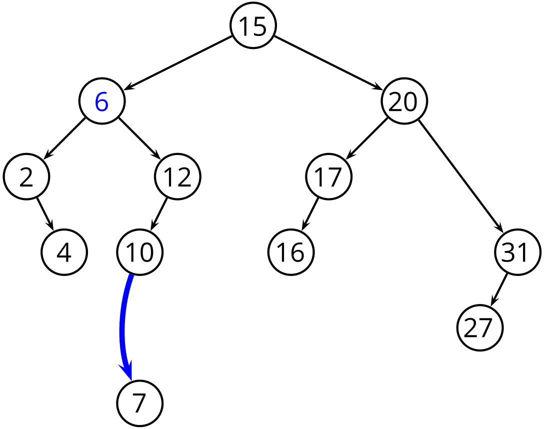
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$



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



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 $y = \textsc{Tree-Successor}(z)$
   - remove y (1 child!)



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 $y = \textsc{Tree-Successor}(z)$
   - remove y (1 child!)
   - connect $y.parent$ to $y.right$

Simple case

Removed node y red - no violations.

Removed node y was black - three problems:

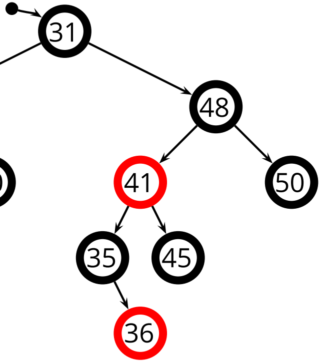
- y was the root: a red child of y becomes the root. Property 2 violated.
- both x and $p[y]$ were red: Property 4 violated.
- y's removal causes some path that contained y to contain one fewer black node: Property 5 violated by any ancestor of y in the tree.

Solution

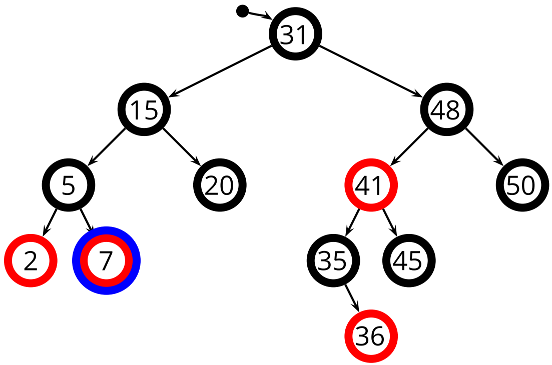
Move the extra black up the tree until:

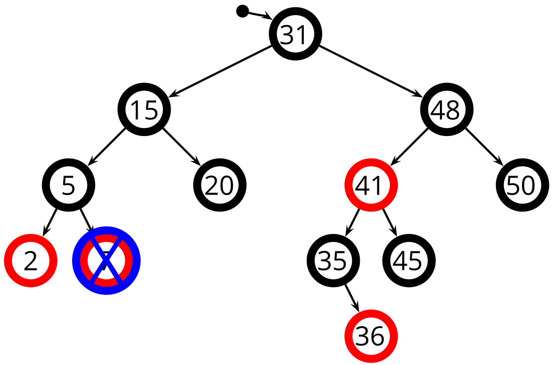
- x points to a red-and-black node, in which case we color x black.
- x points to the root, in which case the extra black can be removed
- suitable rotations and recolorings can be performed.

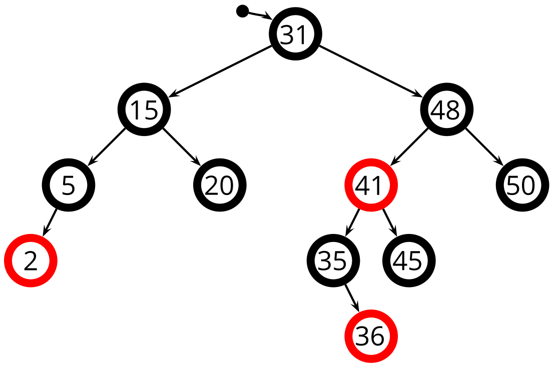
- Case 1: x's sibling is red.
- Case 2: x's sibling w is black, and both w's children are black.
- Case 3: x's sibling w is black, w's left child is red, w's right child is black.
- Case 4: x's sibling w is black, and w's right child is red.

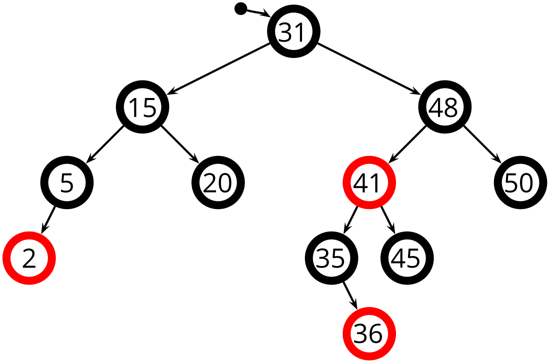


31
15
48
5
20
41
50
2
7
35
45
36

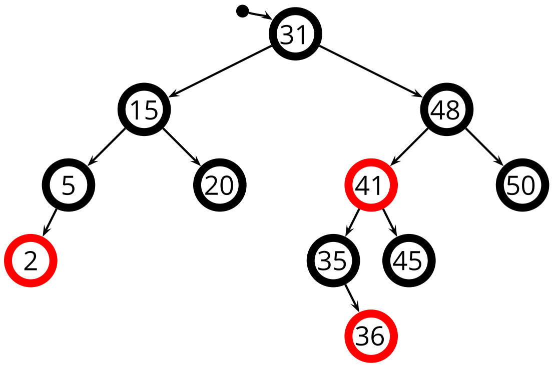

31
15
48
5
20
41
50
2
7
35
45
36


31
15
48
5
20
41
50
2
35
45
36

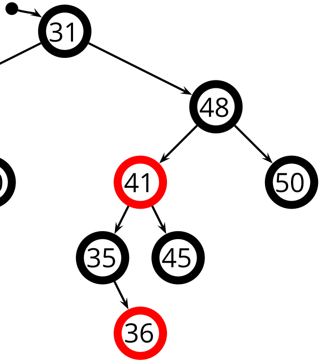

31
15
48
5
20
41
50
2
35
45
36



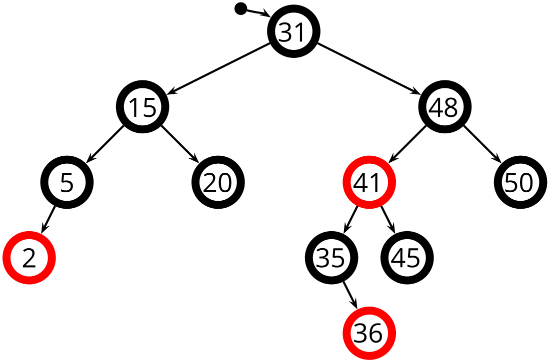
- Deleting a **red** *leaf* does not require any adjustment



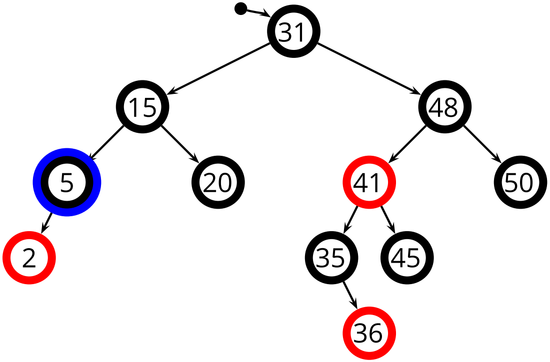
- Deleting a **red** *leaf* does not require any adjustment
  - the deletion does not affect the black height (property 5)

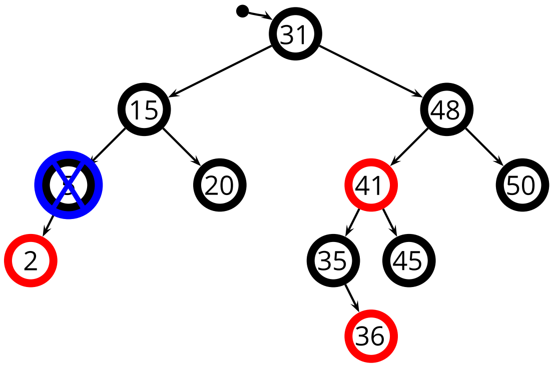


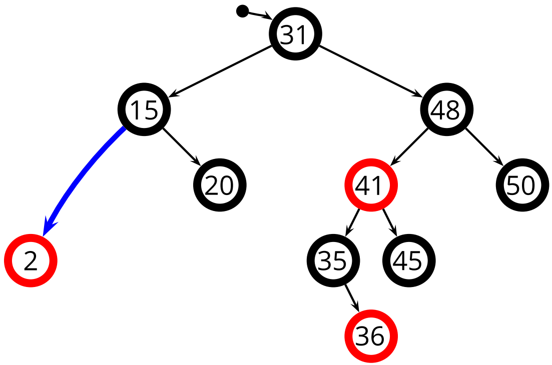
- Deleting a **red** *leaf* does not require any adjustment
  - the deletion does not affect the black height (property 5)
  - no two red nodes become adjacent (property 4)

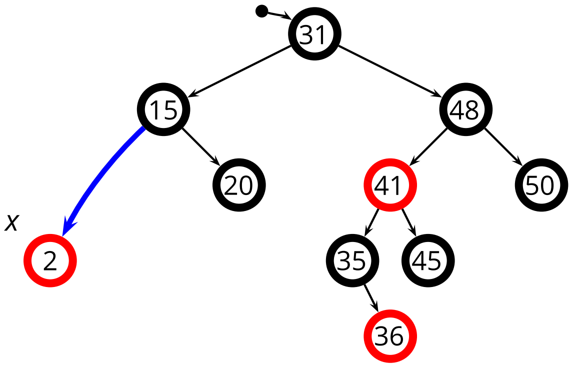

31
15
48
5
20
41
50
2
35
45
36

Red-Black Deletion (2)

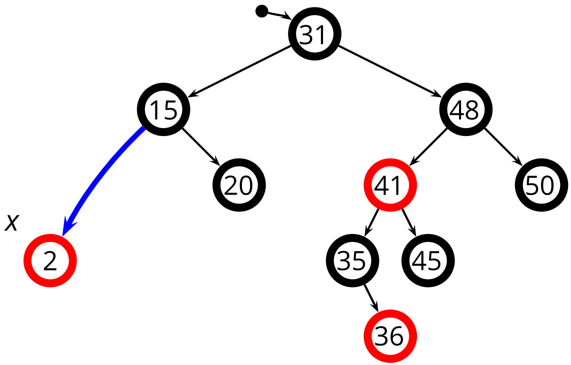



31
15
48
20
41
50
2
35
45
36

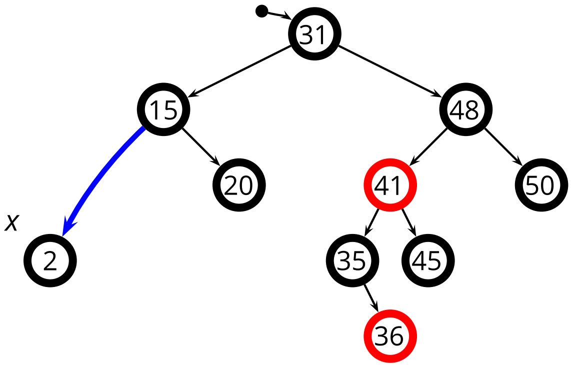

31
15
48
2
20
41
50
35
45
36



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



- 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: $x.color$ = BLACK

Fixup Conditions

- 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**

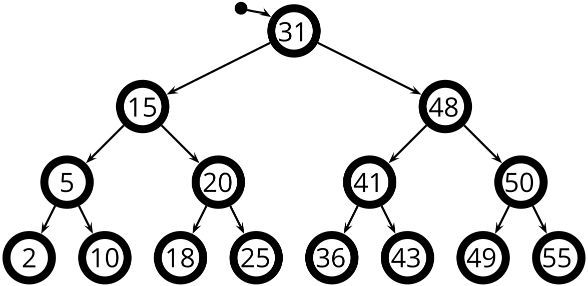
- y is the spliced node ($y = z$ if z has zero or one child)
  - if y is **red**, then no fixup is necessary
  - so, here we assume that y is **black**
- x is either y's only child or $T.nil$
  - y was spliced out, so y can not have two children
  - $x = T.nil$ iff y has no (key-bearing) children

- 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 **??** (*root* must be **black**)

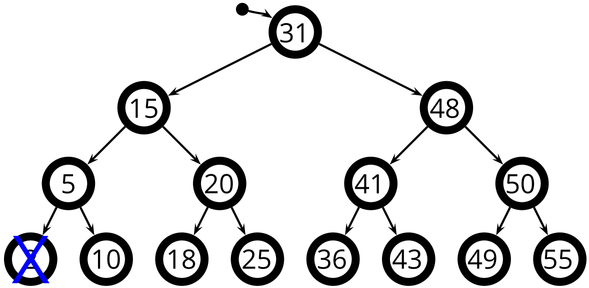
- y is the spliced node ($y = z$ if z has zero or one child)
  - if y is **red**, then no fixup is necessary
  - so, here we assume that y is **black**
- x is either y's only child or $T.nil$
  - y was spliced out, so y can not have two children
  - $x = T.nil$ iff y has no (key-bearing) children
- ***Problem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

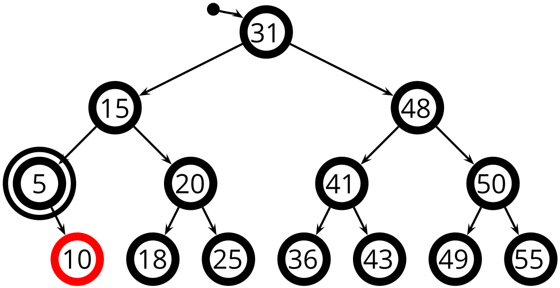
- y is the spliced node ($y = z$ if z has zero or one child)
  - if y is **red**, then no fixup is necessary
  - so, here we assume that y is **black**
- x is either y's only child or $T.nil$
  - y was spliced out, so y can not have two children
  - $x = T.nil$ iff y has no (key-bearing) children
- ***Problem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

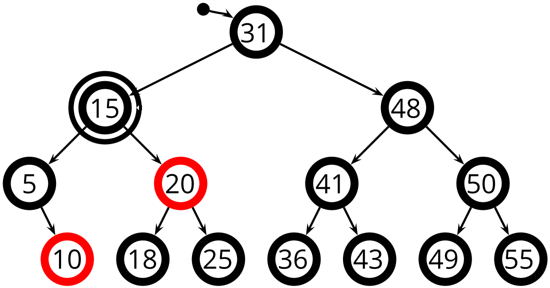


Red-Black Deletion (3)

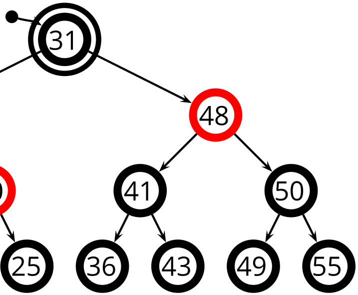




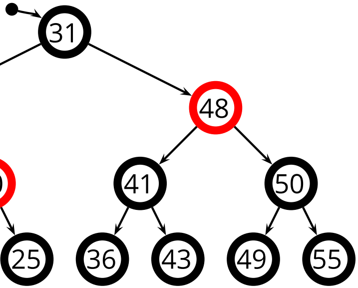
- x carries an *additional **black** weight*
  - the fixup algorithm pushes it up towards to root



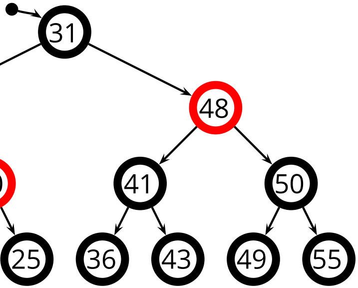
- x carries an *additional* ***black*** *weight*
  - the fixup algorithm pushes it up towards to root



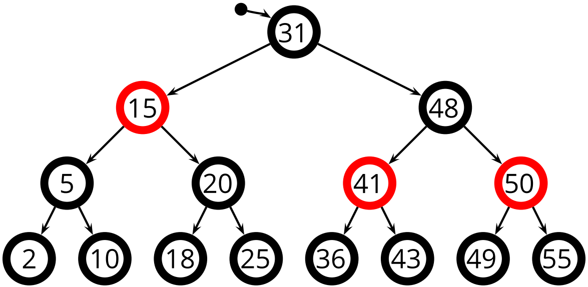
- x carries an *additional **black** weight*
  - the fixup algorithm pushes it up towards to root

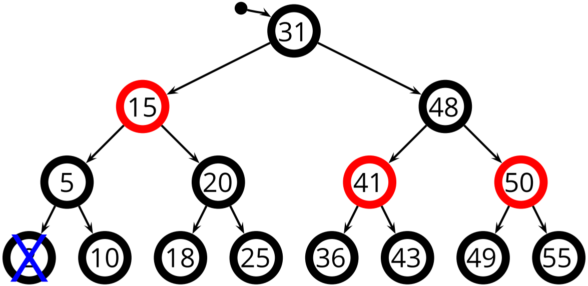


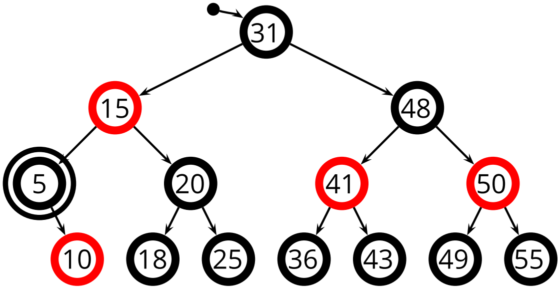
- *x* carries an *additional **black** weight*
  - the fixup algorithm pushes it up towards to root



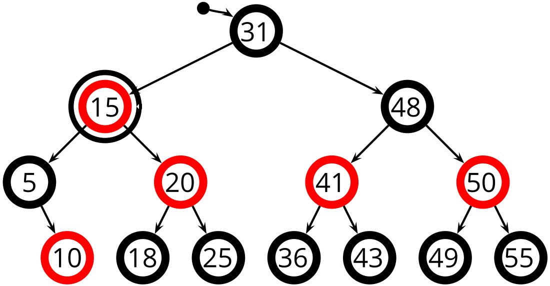
- x carries an *additional **black** weight*
  - the fixup algorithm pushes it up towards to root
- The *additional **black** weight* can be discarded if it reaches the *root*, otherwise...

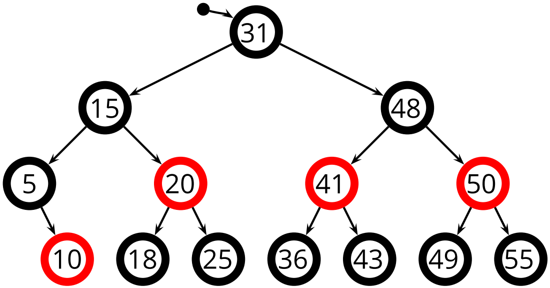

31
15
48
5
20
41
50
2
10
18
25
36
43
49
55


31
15
48
5
20
41
50
10
18
25
36
43
49
55


31
15
48
5
20
41
50
10
18
25
36
43
49
55

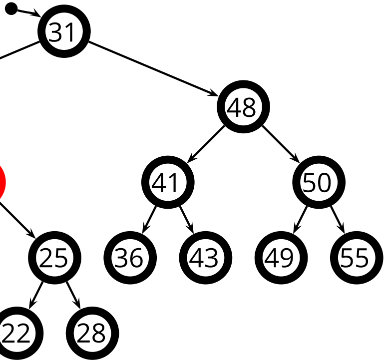
Red-Black Deletion (4)



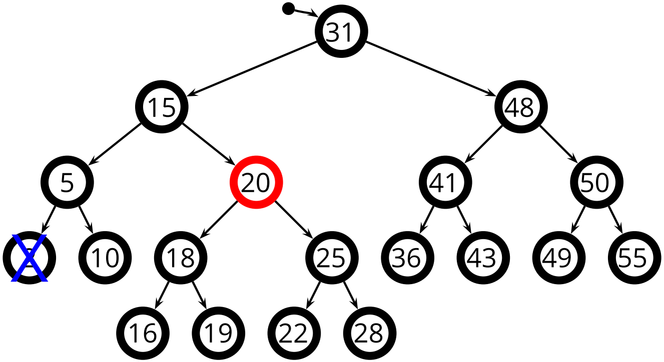


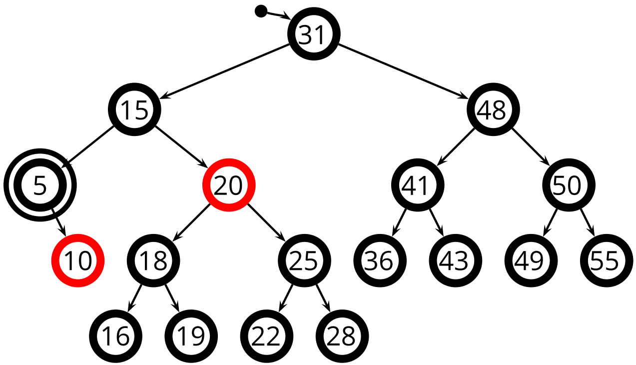
- The *additional **black** weight* can also stop as soon as it reaches a **red** node, which will absorb the extra **black** color

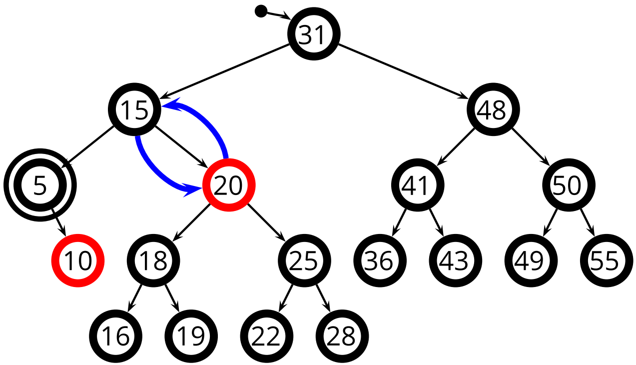
Red-Black Deletion (5)

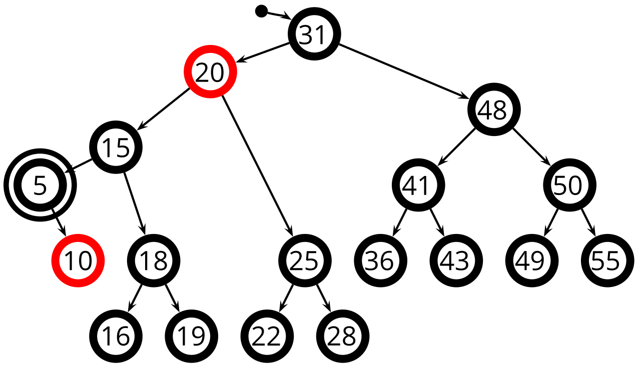


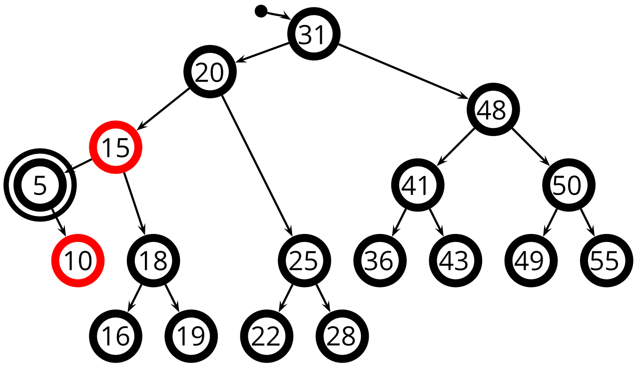
Red-Black Deletion (5)



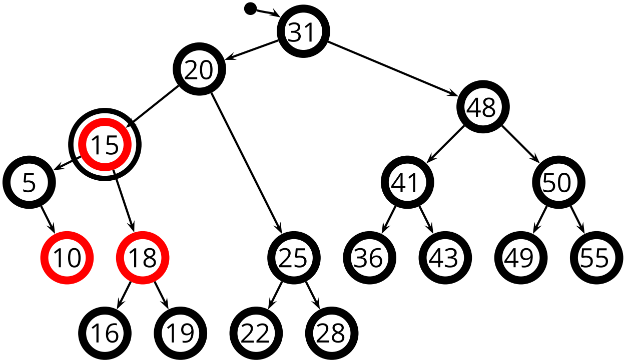

31
15
48
5
20
41
50
10
18
25
36
43
49
55
16
19
22
28


31
15
48
5
20
41
50
10
18
25
36
43
49
55
16
19
22
28

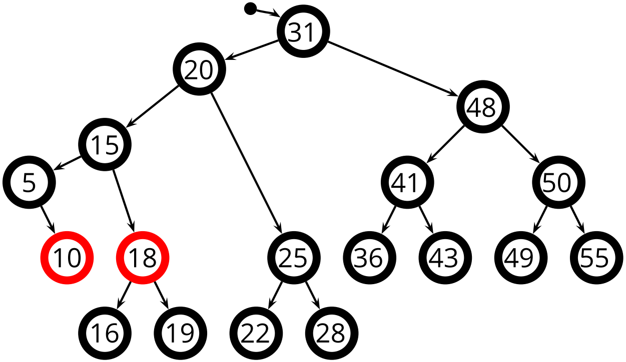

31
20
48
15
5
41
50
10
18
25
36
43
49
55
16
19
22
28

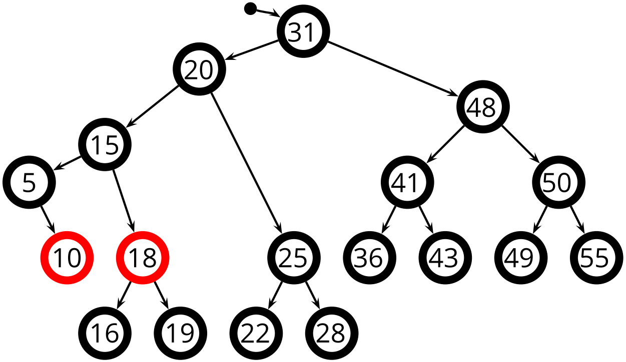

31
20
48
15
5
41
50
10
18
25
36
43
49
55
16
19
22
28

Red-Black Deletion (5)



Red-Black Deletion (5)



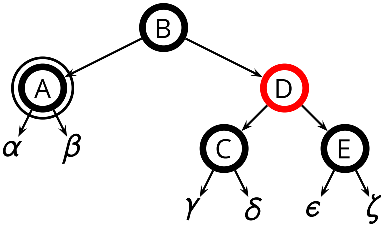


- In other cases where we can not push the additional black color up, we can apply appropriate rotations and color transfers that preserve all other red-black properties

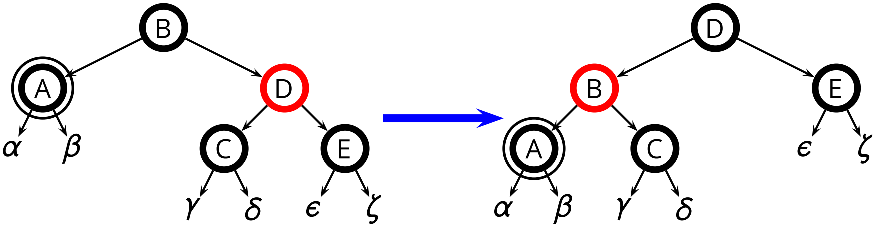
Basic Fixup Iteration (1)

Case 1

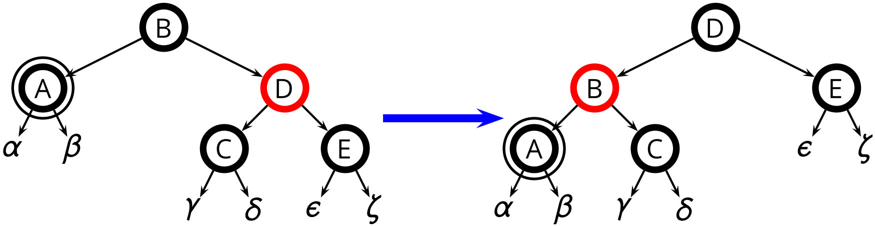
Case 1



Case 1

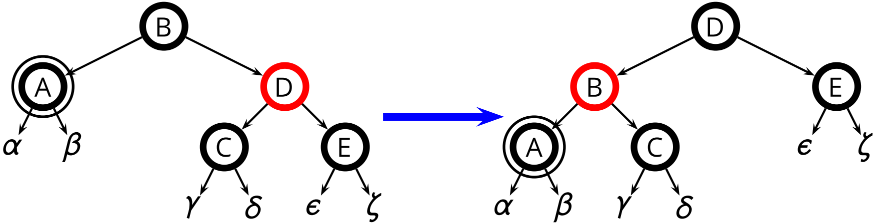

B
A
D
α
β
C
E
γ
δ
ϵ
ζ
D
B
E
A
C
ϵ
ζ
α
β
γ
δ

Case 1

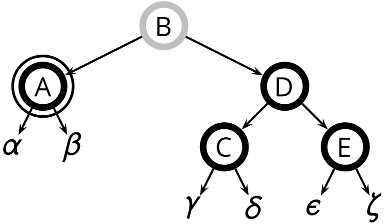


Case 2

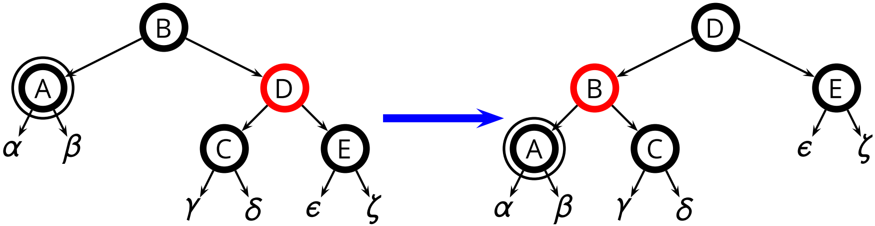
Case 1



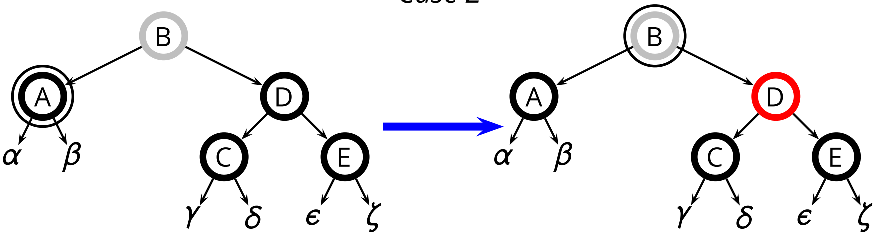
Case 2



Case 1

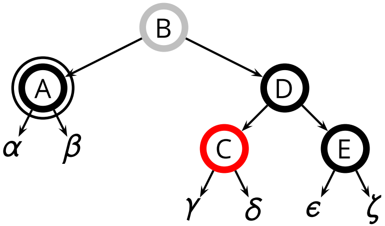


Case 2

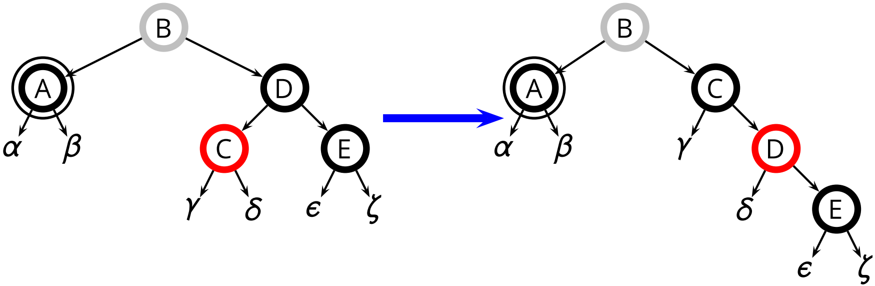


Case 3

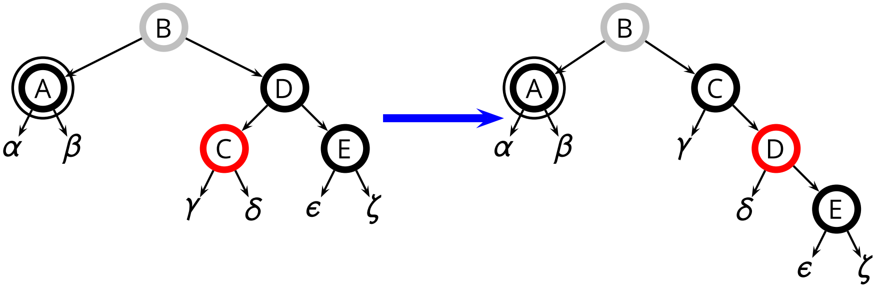
Case 3



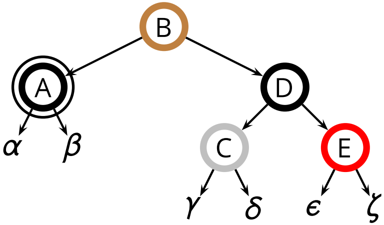
Case 3


B
A
D
C
E
α
β
γ
δ
ϵ
ζ
B
A
C
D
E
α
β
γ
δ
ϵ
ζ

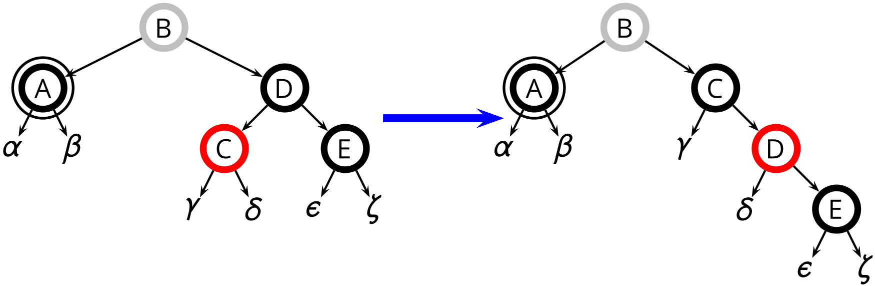
Case 3



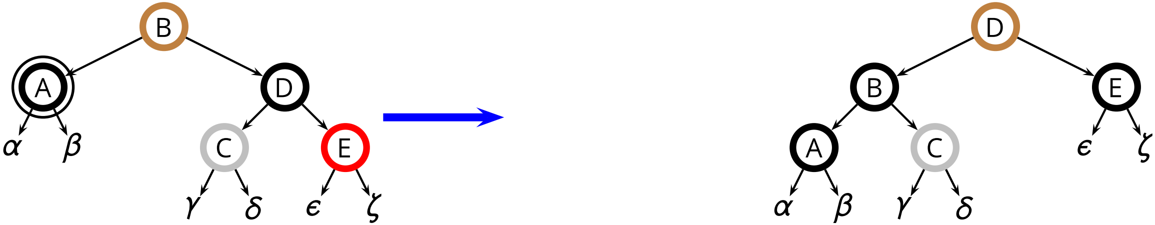
Case 4



Case 3



Case 4



Red-Black Delete Fixup

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root ∧ x.color = BLACK
    if x == x.parent.left
        w = x.parent.right
        if w.color == RED
            case 1...
        if w.left.color == BLACK ∧ w.right.color = BLACK
            w.color = RED                        // case 2
            x = x.parent
        else if w.right.color == BLACK
                case 3...
            case 4...
    else same as above, exchanging right and left
x.color = BLACK
```

- Search, insert, delete in dictionary: $O(\log n)$.
- Red-black trees important in functional programming: persistent data structures.
- Approximate balance maintained via colors, and invariants on coloring.
- Restoring these invariants after insertions/deletions performed using rotations.
- ...
- Finally done with binary search trees !

- We studied data structures to design/improve algorithms
- Let's see some examples of algorithms that these complicated BST's improve !

  Next time !