Red-Black Trees

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

April 28, 2020

Outline

- Red-black trees

Summary on Binary Search Trees

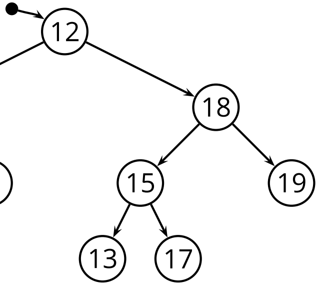
- Binary search trees
  - embody the ***divide-and-conquer*** search strategy
  - **SEARCH**, **INSERT**, **MIN**, and **MAX** are $O(h)$, where h is the ***height of the tree***
  - in general, $h(n) = \Omega(\log n)$ and $h(n) = O(n)$
  - ***randomization*** can make the worst-case scenario $h(n) = n$ highly unlikely

Summary on Binary Search Trees

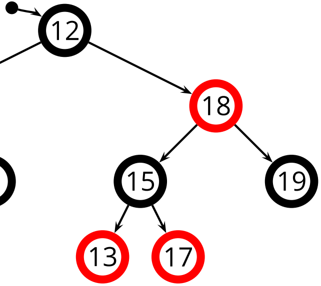
- Binary search trees
  - embody the ***divide-and-conquer*** search strategy
  - **Search**, **Insert**, **Min**, and **Max** are $O(h)$, where h is the ***height of the tree***
  - in general, $h(n) = \Omega(\log n)$ and $h(n) = O(n)$
  - ***randomization*** can make the worst-case scenario $h(n) = n$ highly unlikely
- Problem
  - worst-case scenario is unlikely but still possible
  - simply bad cases are even more probable

Red-Black Tree

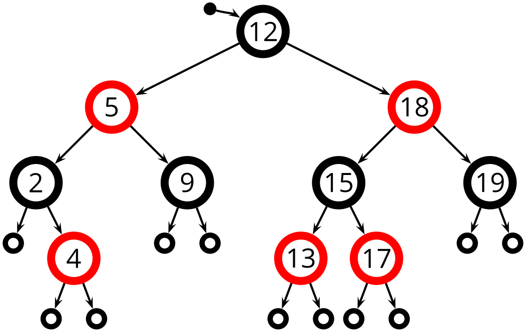
Red-Black Tree



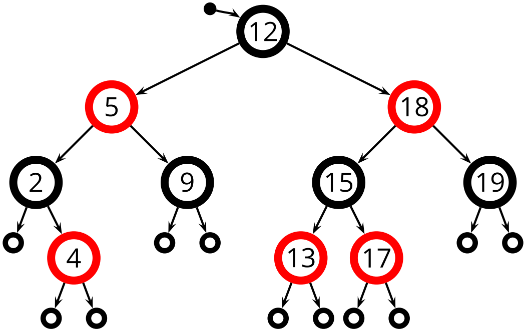
Red-Black Tree



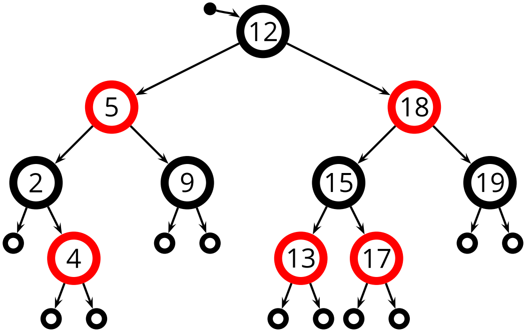
Red-Black Tree



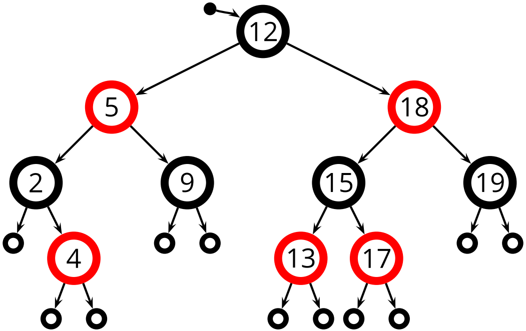
Red-Black Tree



- *Red-black-tree property*



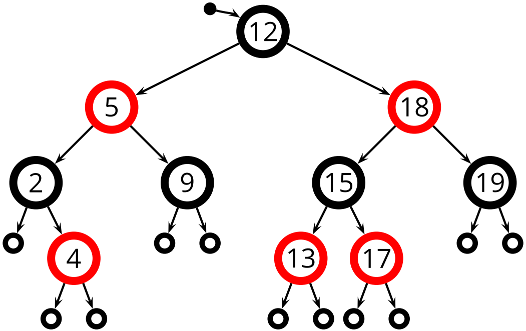
- *Red-black-tree property*
  1. every node is either **red** or **black**



- *Red-black-tree property*

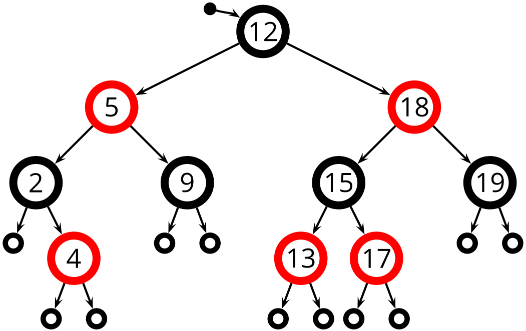
  1. every node is either **red** or **black**
  2. the root is **black**

Red-Black Tree



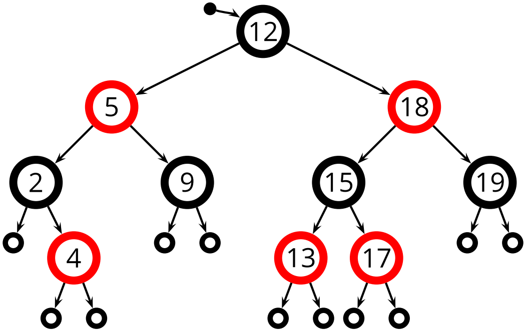
- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**

Red-Black Tree



- *Red-black-tree property*

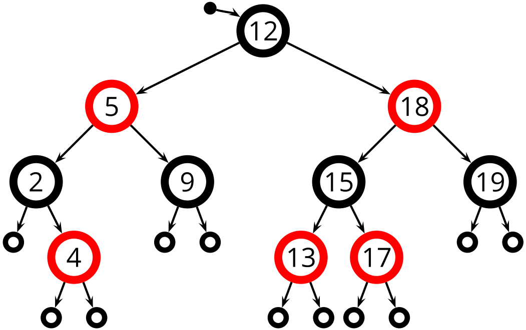
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**



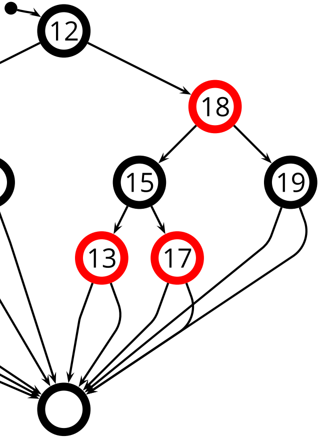
- *Red-black-tree property*

  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*

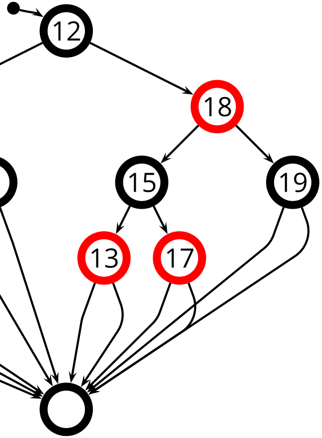


- *Implementation*



  - 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*

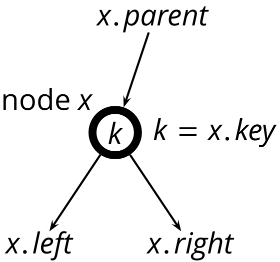
- *Implementation*
  - T represents the tree, which consists of a set of *nodes*
  - $T.root$ is the root node of tree T

- *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

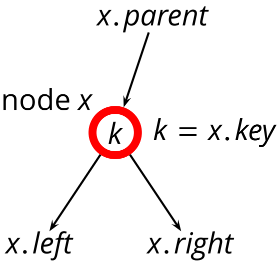
Nodes

- $x.parent$ 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

Height of a Red-Black Tree

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 $2\log(n+1)$.

Proof:

1. prove that $\forall x : size(x) \geq 2^{bh(x)} - 1$ by induction:
   1.1 ***base case:*** x is a leaf, so $size(x) = 0$ and $bh(x) = 0$

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

Proof:

1. prove that $\forall x : size(x) \geq 2^{bh(x)} - 1$ by induction:

   1.1 ***base case:*** x is a leaf, so $size(x) = 0$ and $bh(x) = 0$

   1.2 ***induction step:*** consider y_1, y_2, and x such that $y_1.parent = y_2.parent = x$, and assume (induction) that $size(y_1) \geq 2^{bh(y_1)} - 1$ and $size(y_2) \geq 2^{bh(y_2)} - 1$;
   prove that $size(x) \geq 2^{bh(x)} - 1$

Height of a Red-Black Tree

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

Proof:

1. prove that $\forall x : size(x) \geq 2^{bh(x)} - 1$ by induction:

    1.1 ***base case:*** x is a leaf, so $size(x) = 0$ and $bh(x) = 0$

    1.2 ***induction step:*** consider y_1, y_2, and x such that $y_1.parent = y_2.parent = x$, and assume (induction) that $size(y_1) \geq 2^{bh(y_1)} - 1$ and $size(y_2) \geq 2^{bh(y_2)} - 1$;
    prove that $size(x) \geq 2^{bh(x)} - 1$
    proof:
    $size(x) = size(y_1) + size(y_2) + 1 \geq (2^{bh(y_1)} - 1) + (2^{bh(y_2)} - 1) + 1$

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

Proof:

1. prove that $\forall x : size(x) \geq 2^{bh(x)} - 1$ by induction:

   1.1 ***base case:*** x is a leaf, so $size(x) = 0$ and $bh(x) = 0$

   1.2 ***induction step:*** consider y_1, y_2, and x such that $y_1.parent = y_2.parent = x$, and assume (induction) that $size(y_1) \geq 2^{bh(y_1)} - 1$ and $size(y_2) \geq 2^{bh(y_2)} - 1$;
   prove that $size(x) \geq 2^{bh(x)} - 1$
   proof:
   $size(x) = size(y_1) + size(y_2) + 1 \geq (2^{bh(y_1)} - 1) + (2^{bh(y_2)} - 1) + 1$
   since

$$bh(x) = \begin{cases} bh(y) & \text{if } color(y) = \text{RED} \\ bh(y) + 1 & \text{if } color(y) = \text{BLACK} \end{cases}$$

Height of a Red-Black Tree

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

Proof:

1. prove that $\forall x : size(x) \geq 2^{bh(x)} - 1$ by induction:

   1.1 ***base case:*** x is a leaf, so $size(x) = 0$ and $bh(x) = 0$

   1.2 ***induction step:*** consider y_1, y_2, and x such that $y_1.parent = y_2.parent = x$, and assume (induction) that $size(y_1) \geq 2^{bh(y_1)} - 1$ and $size(y_2) \geq 2^{bh(y_2)} - 1$;
   prove that $size(x) \geq 2^{bh(x)} - 1$

   proof:

   $size(x) = size(y_1) + size(y_2) + 1 \geq (2^{bh(y_1)} - 1) + (2^{bh(y_2)} - 1) + 1$

   since

   $$bh(x) = \begin{cases} bh(y) & \text{if } color(y) = \text{RED} \\ bh(y) + 1 & \text{if } color(y) = \text{BLACK} \end{cases}$$

   $size(x) \geq (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$

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

Proof:

1. prove that $\forall x : size(x) \geq 2^{bh(x)} - 1$ by induction:

   1.1 ***base case:*** x is a leaf, so $size(x) = 0$ and $bh(x) = 0$

   1.2 ***induction step:*** consider y_1, y_2, and x such that $y_1.parent = y_2.parent = x$, and assume (induction) that $size(y_1) \geq 2^{bh(y_1)} - 1$ and $size(y_2) \geq 2^{bh(y_2)} - 1$;
   prove that $size(x) \geq 2^{bh(x)} - 1$
   proof:
   $size(x) = size(y_1) + size(y_2) + 1 \geq (2^{bh(y_1)} - 1) + (2^{bh(y_2)} - 1) + 1$
   since

$$bh(x) = \begin{cases} bh(y) & \text{if } color(y) = \text{RED} \\ bh(y) + 1 & \text{if } color(y) = \text{BLACK} \end{cases}$$

   $size(x) \geq (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$

Height of a Red-Black Tree (2)

1. $size(x) \geq 2^{bh(x)} - 1$

Height of a Red-Black Tree (2)

1. $size(x) \geq 2^{bh(x)} - 1$

2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black

Height of a Red-Black Tree (2)

1. $size(x) \geq 2^{bh(x)} - 1$

2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black, thus $bh(x) \geq h(x)/2$

Height of a Red-Black Tree (2)

1. $size(x) \geq 2^{bh(x)} - 1$

2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black, thus $bh(x) \geq h(x)/2$

3. From steps 1 and 2, $n = size(x) \geq 2^{h(x)/2} - 1$

Height of a Red-Black Tree (2)

1. $size(x) \geq 2^{bh(x)} - 1$

2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black, thus $bh(x) \geq h(x)/2$

3. From steps 1 and 2, $n = size(x) \geq 2^{h(x)/2} - 1$, therefore

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

Height of a Red-Black Tree (2)

1. $size(x) \geq 2^{bh(x)} - 1$
2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black, thus $bh(x) \geq h(x)/2$
3. From steps 1 and 2, $n = size(x) \geq 2^{h(x)/2} - 1$, therefore

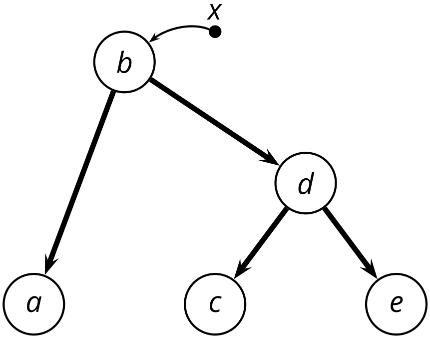
$$h \leq 2\log(n+1)$$

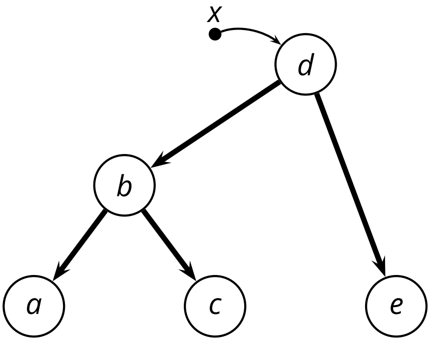
- A red-black tree works as a binary search tree for search, etc.
- So, the complexity of those operations is $T(n) = O(h)$, that is

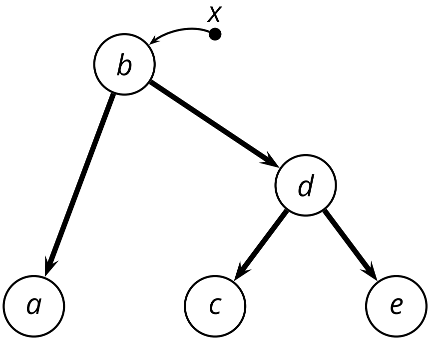
$$\boxed{T(n) = O(\log n)}$$

  - which is also the *worst-case* complexity

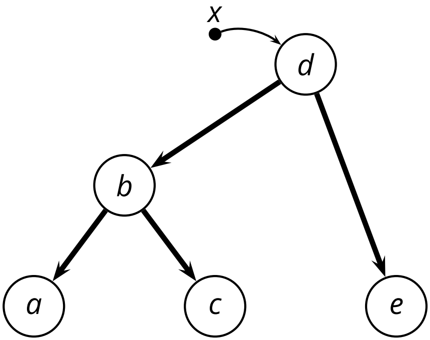
Rotation




x
d
b
a
c
e



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



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

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

Red-Black Insertion

- **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*

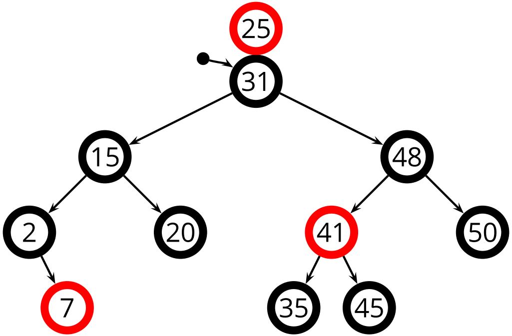
- **RB-INSERT**(T, z) works as in a binary search tree
- Except that it must preserve the *red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**
  5. for every 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

RB-INSERT

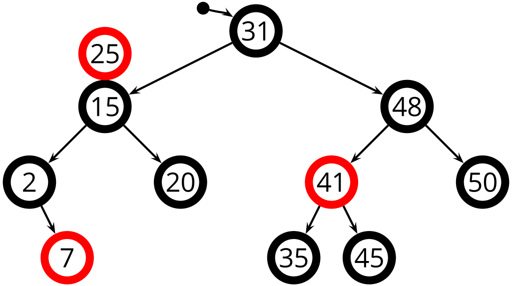
RB-INSERT(T, z)

```
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)
```

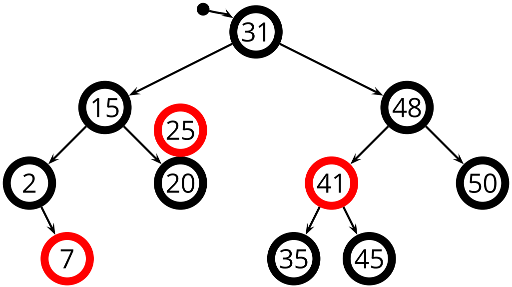
Red-Black Insertion (2)



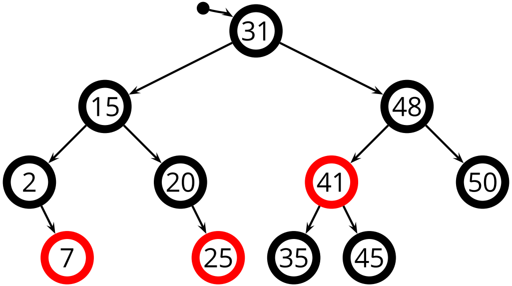
Red-Black Insertion (2)

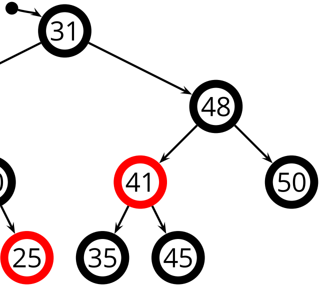


Red-Black Insertion (2)



Red-Black Insertion (2)

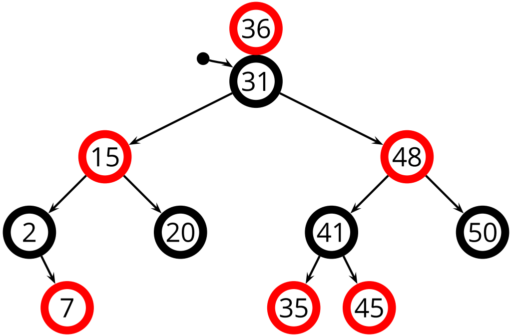




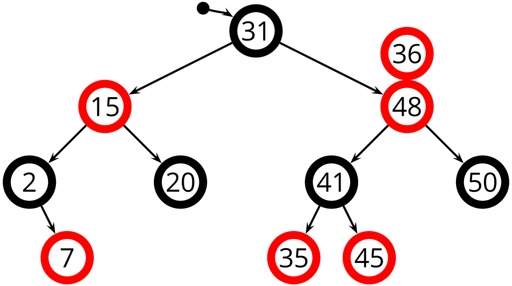
- *z*'s parent is **black**, so no fixup needed

Red-Black Insertion (3)

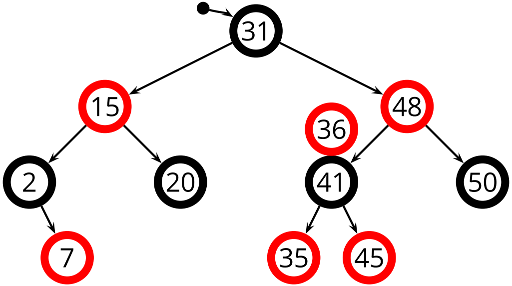
Red-Black Insertion (3)



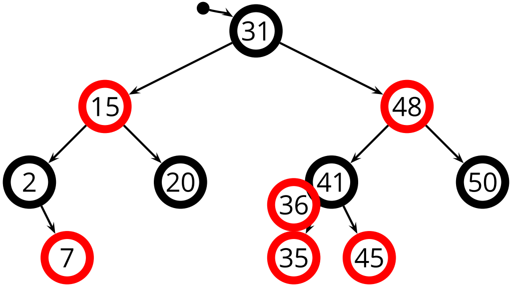
Red-Black Insertion (3)



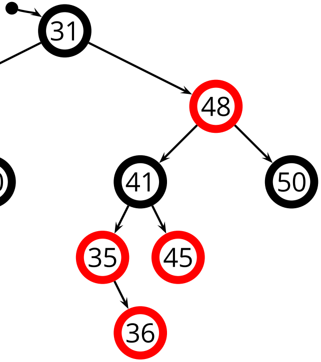
Red-Black Insertion (3)



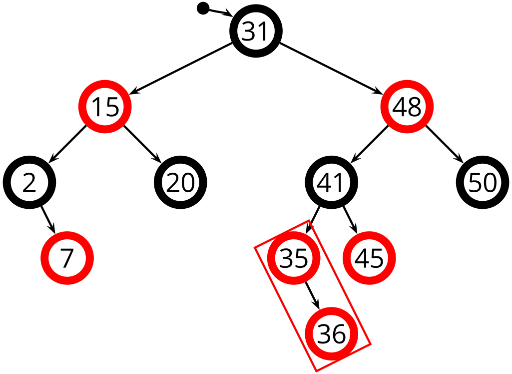
Red-Black Insertion (3)



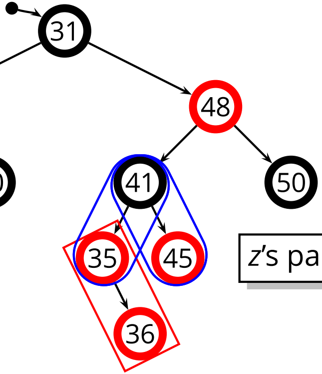
Red-Black Insertion (3)



Red-Black Insertion (3)

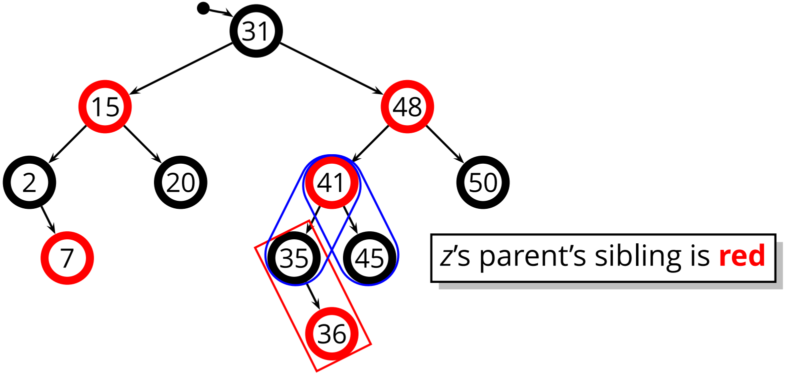


Red-Black Insertion (3)

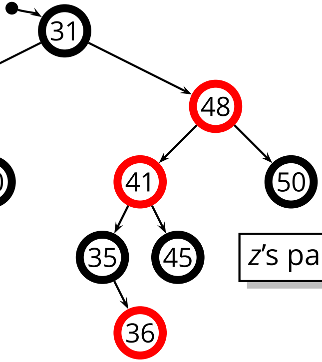


z's parent's sibling is **red**

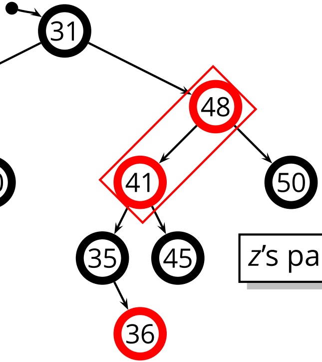
Red-Black Insertion (3)



Red-Black Insertion (3)

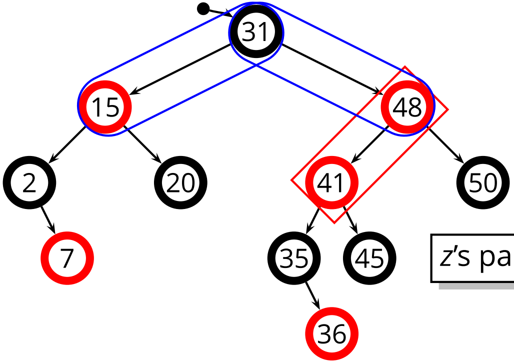


z's parent's sibling is **red**



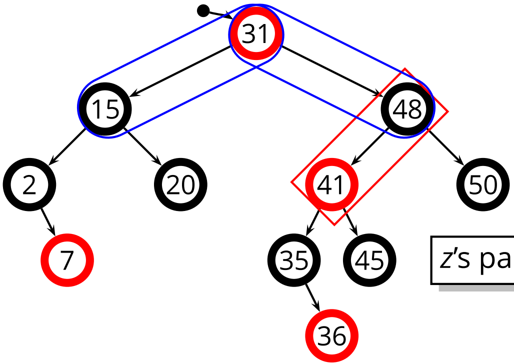
z's parent's sibling is **red**

Red-Black Insertion (3)



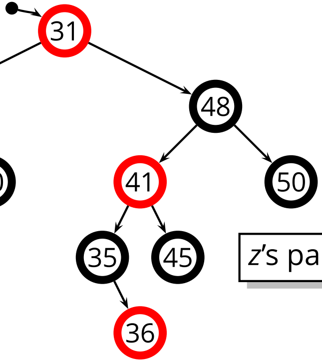
z's parent's sibling is **red**

Red-Black Insertion (3)



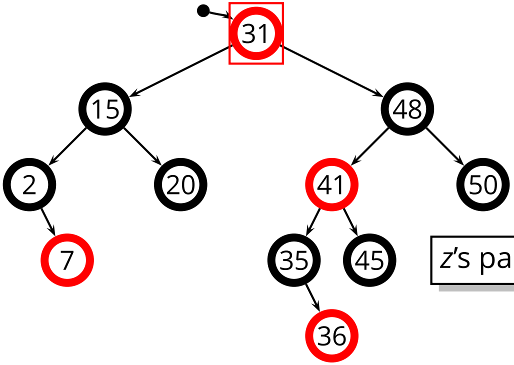
z's parent's sibling is **red**

Red-Black Insertion (3)



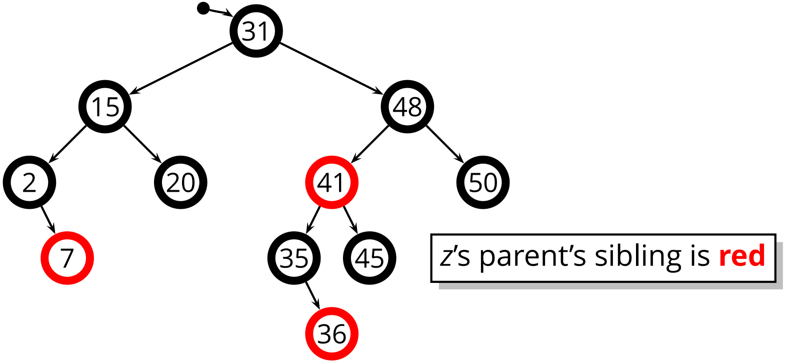
z's parent's sibling is **red**

Red-Black Insertion (3)

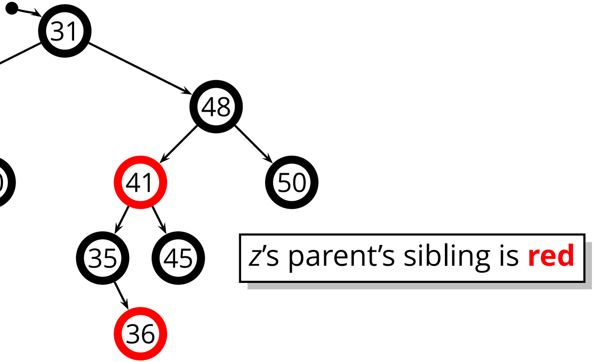


z's parent's sibling is **red**

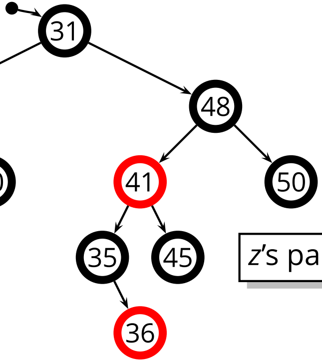
Red-Black Insertion (3)



Red-Black Insertion (3)



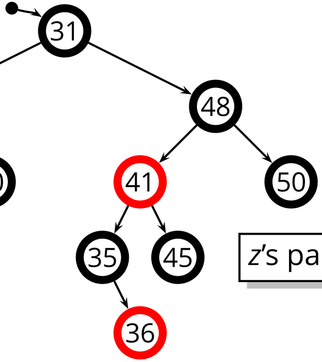
- A **black** node can become **red** and transfer its **black** color to its two children



z's parent's sibling is **red**

- 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…

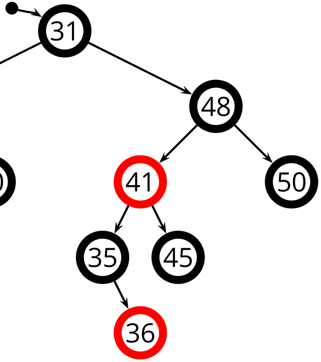
Red-Black Insertion (3)



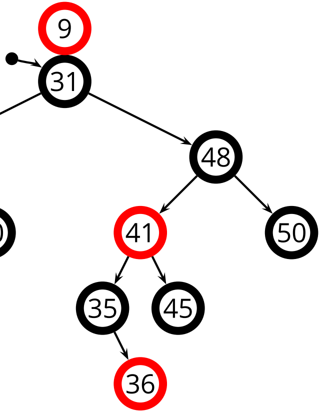
z's parent's sibling is **red**

- 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

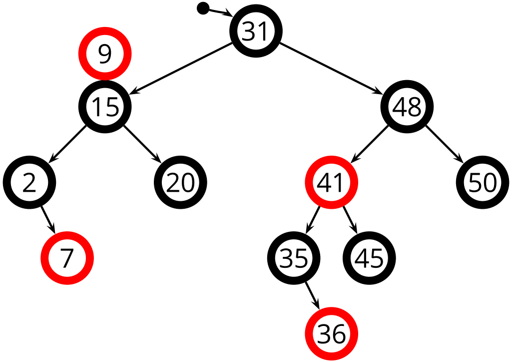
Red-Black Insertion (4)



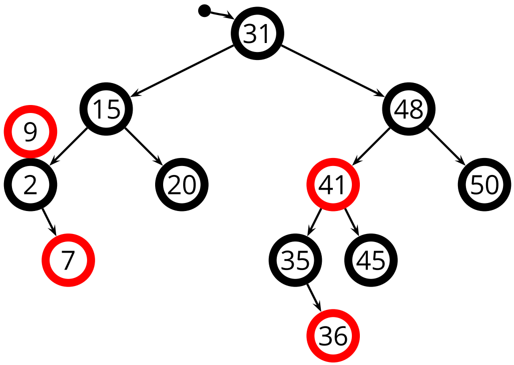
Red-Black Insertion (4)



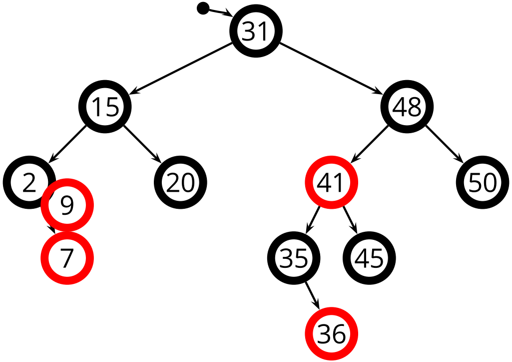
Red-Black Insertion (4)



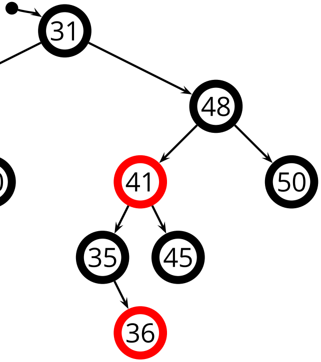
Red-Black Insertion (4)



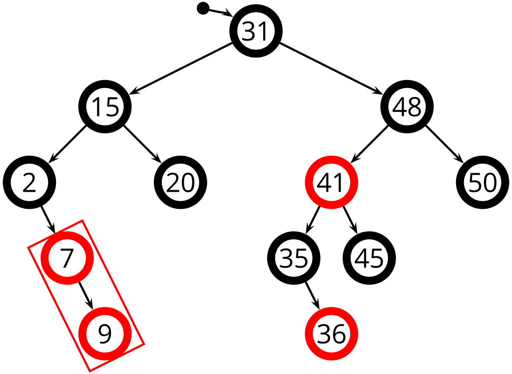
Red-Black Insertion (4)



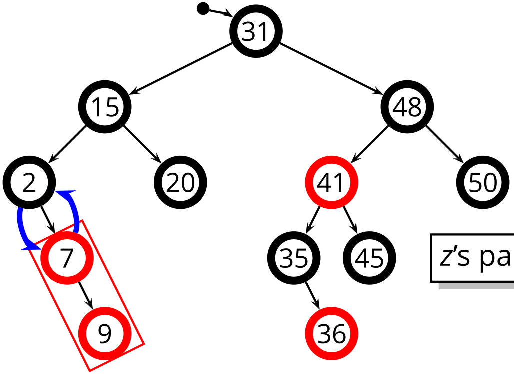
Red-Black Insertion (4)



Red-Black Insertion (4)

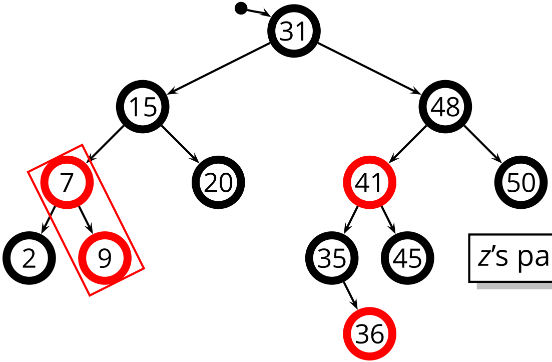


Red-Black Insertion (4)



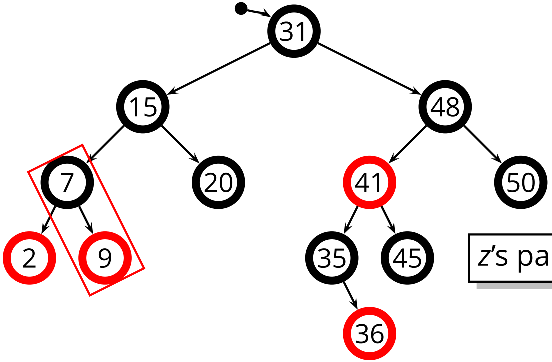
z's parent's sibling is **black**

Red-Black Insertion (4)



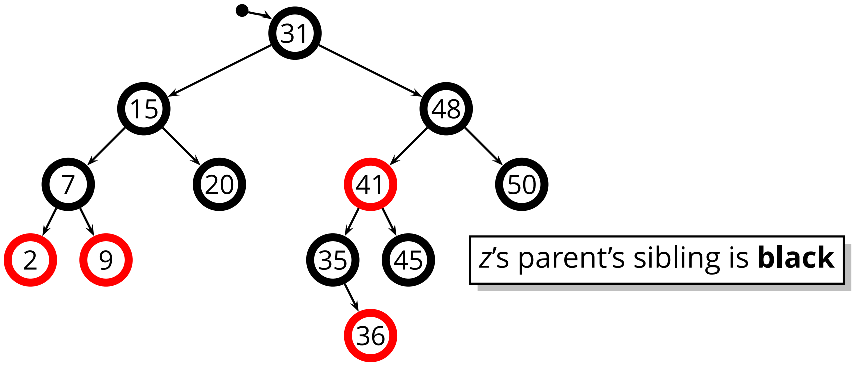
z's parent's sibling is **black**

Red-Black Insertion (4)



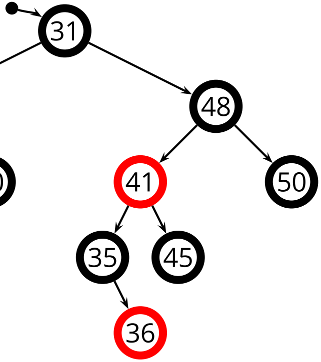
z's parent's sibling is **black**

Red-Black Insertion (4)

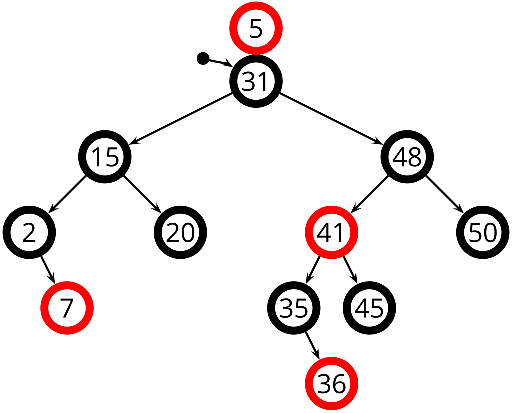


- An *in-line* ***red*–red** conflicts can be resolved with a rotation plus a color switch

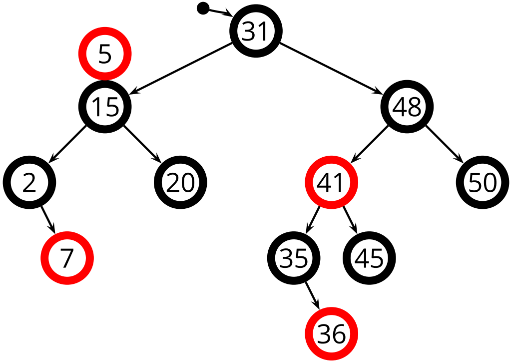
Red-Black Insertion (5)



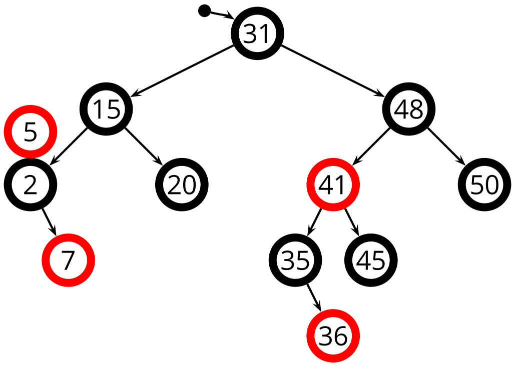
Red-Black Insertion (5)



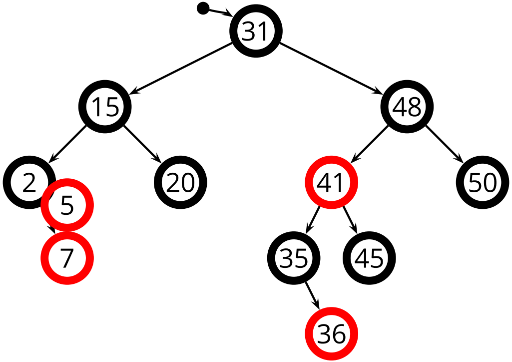
Red-Black Insertion (5)



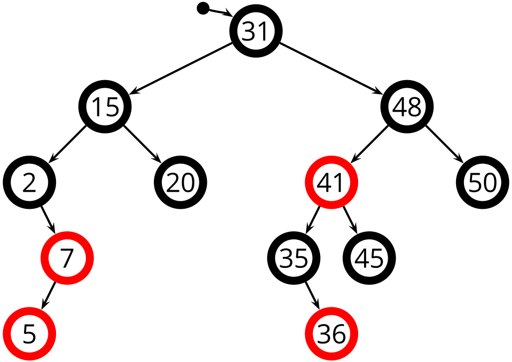
Red-Black Insertion (5)



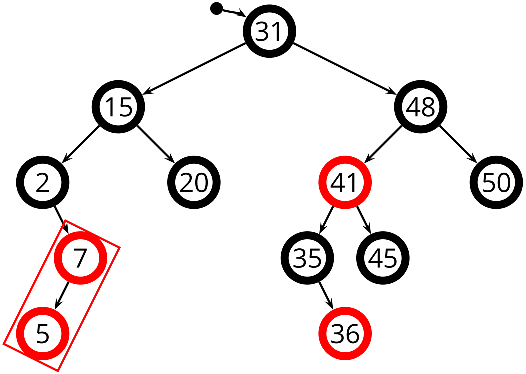
Red-Black Insertion (5)



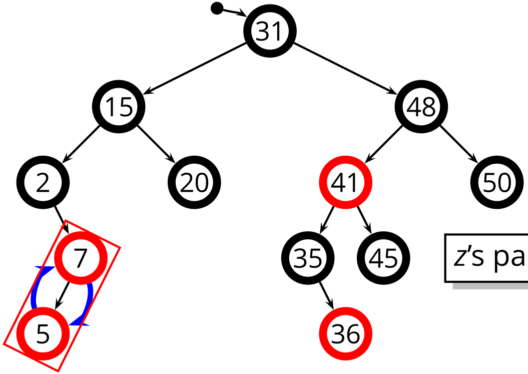
Red-Black Insertion (5)



Red-Black Insertion (5)

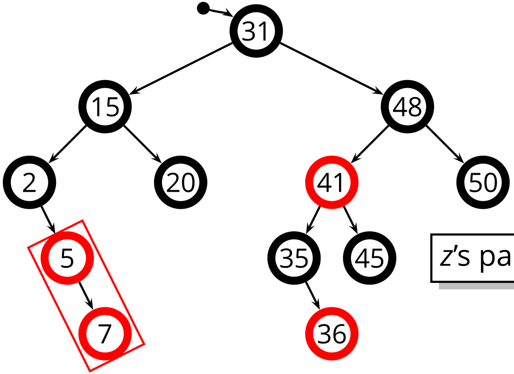


Red-Black Insertion (5)



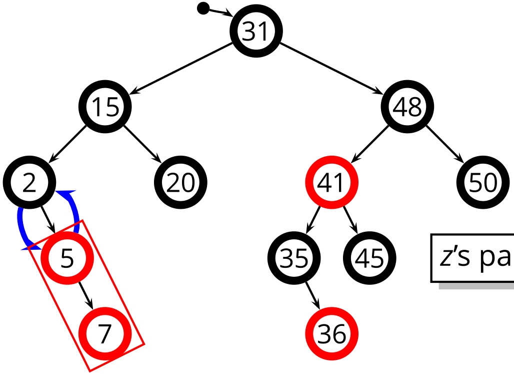
z's parent's sibling is **black**

Red-Black Insertion (5)



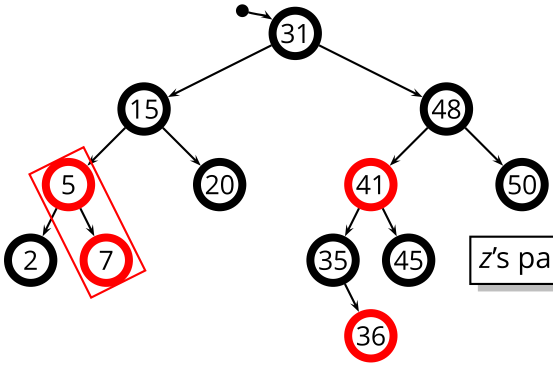
z's parent's sibling is **black**

Red-Black Insertion (5)



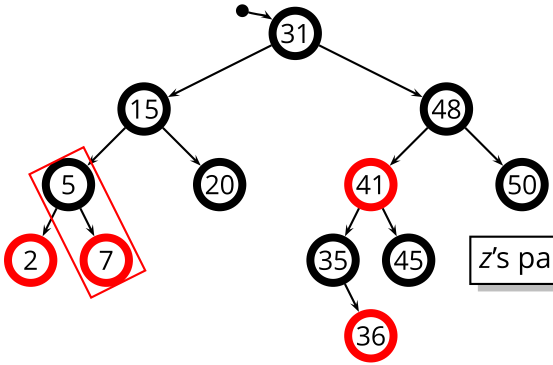
z's parent's sibling is **black**

Red-Black Insertion (5)



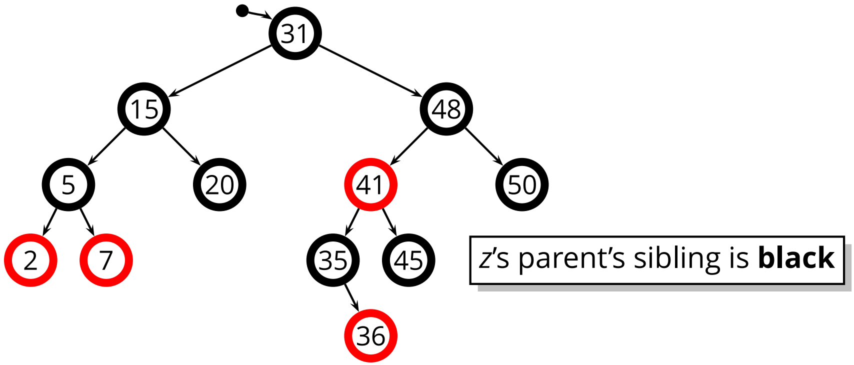
z's parent's sibling is **black**

Red-Black Insertion (5)



z's parent's sibling is **black**

Red-Black Insertion (5)



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