Algorithms and Data Structures (II)

Gabriel Istrate

April 1, 2020

Outline

- AVL trees.
- Splay trees
- Red-black trees

Where we are

- 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.
- Can rebalance.

Today: three self-adjusting trees, finally meet the $O(\log n)$ bound: AVL trees, splay trees, 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 be used to 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 be used to 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

Step back

- Why study all this stuff ?
- Linked list: search linear.
- Balanced binary trees: search logarithmic.
- For frequent searches it pays.
- Advantage: as long as trees “approximately balanced”.
- But: operations (inserts/deletes) can destroy balance.

Self-balancing trees

If insertion/deletion unbalances the tree, rebalance it.

Why three of them ?

- AVL trees: more strictly balanced than R-B trees. Better for lookup intensive programs.
- For an insert intensive tasks, use a Red-Black tree.
- Simplicity of implementation: splay trees > red-black trees > AVL trees.
- Splay trees: only $O(\log n)$ amortized.
- Splay trees: suitable for cases where there are large number of nodes but only few of them are accessed frequently.
- Splay trees: more *memory-efficient* than AVL trees, because they do not need to store balance information in the nodes.
- AVL trees: more useful in multithreaded environments with lots of lookups, because lookups in an AVL tree can be done in parallel.
- Benchmarking: AVL trees more than 20% faster than R-B trees in "real-life" benchmarkis

Why three of them ?

- treaps
- T-trees
- tango trees

... many other ! (But this is an introduction)

You can invent your own trees ...

Tango trees: A type of binary search tree proposed by Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pãtraşcu in 2004.

They work by partitioning a BST into a set of preferred paths, which are themselves stored in auxiliary trees (so the tango tree is represented as a tree of trees)

In practice

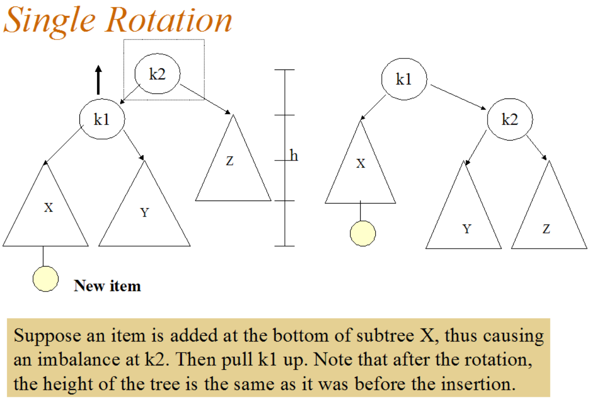
- red-black trees:
  1. Java: java.util.TreeMap , java.util.TreeSet .
  2. C++ STL: map, multimap, multiset.
  3. Linux kernel: completely fair scheduler, linux/rbtree.h
- Splay trees: typically used in the implementation of caches, memory allocators, routers, garbage collectors, data compression, etc.
- **Implementations** of AVL trees, RB-trees, splay trees: *not standardized.* STL provides only minimal set of containers.

AVL trees

- Balancedness condition #1: left and right subtrees of the root have the same height. too weak.
- Balancedness condition #2: left and right subtrees of **every node** have the same height. too strong.

- AVL (Adelson-Velskii and Landis) trees: binary search trees that verify the following balancedness condition: for every node *v* the left and right subtrees of *v* have height differing by at most one.
- When a tree violates rule #3 a repair is done.
- The repair is done during insertions, as soon as rule #3 is violated.
- The repair is accomplished via "single" and "double" rotations.

Single rotations



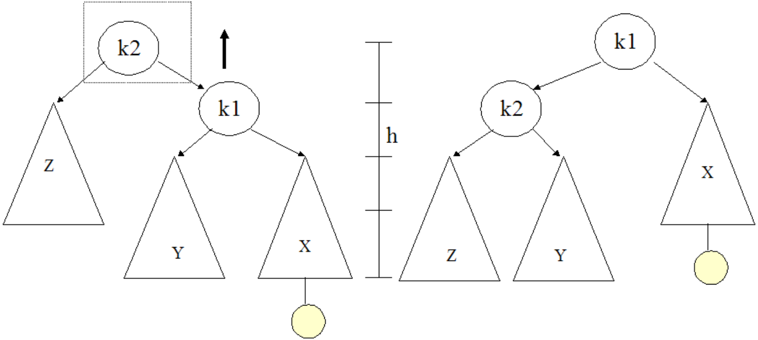
Single Rotation

Suppose an item is added at the bottom of subtree X, thus causing an imbalance at k2. Then pull k1 up. Note that after the rotation, the height of the tree is the same as it was before the insertion.

Single rotations

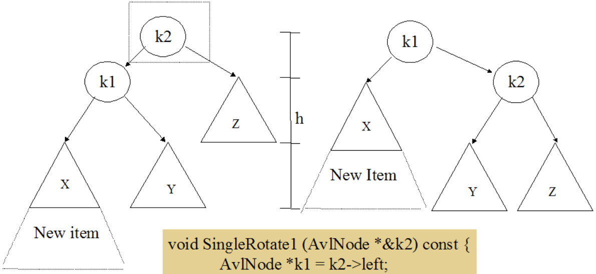
- Balance factor of a node: difference of heights between left and right subtrees.
- AVL trees: each node balance factor 0 or ±1.
- After single rotations, the new height of the entire subtree is exactly the same as the height of the original subtree prior to the insertion of the new data item that caused X to grow.
- Thus no further updating of heights on the path to the root is needed, and consequently no further rotations are needed.

Single rotations: another example



Suppose an item is added at the bottom of subtree X, thus causing an imbalance at k2. Then pull k1 up. Note that after the rotation, the height of the tree is the same as it was before the insertion.

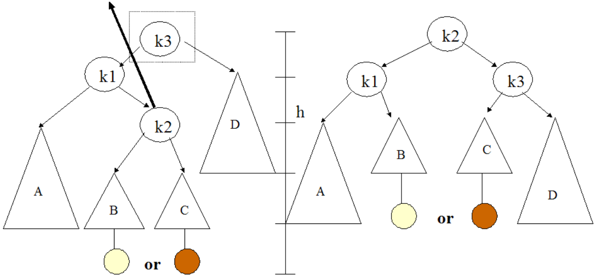
Single rotations: C++



```
void SingleRotate1 (AvlNode *&k2) const {
        AvlNode *k1 = k2->left;
        k2->left = k1->right;
        k1->right = k2;
        k2 = k1;
 }
```

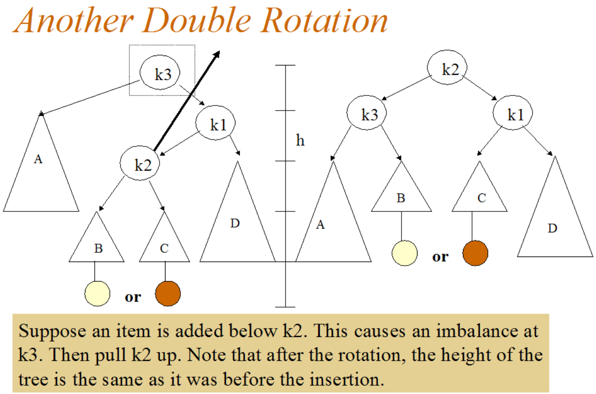
Double rotations

Double Rotation



Suppose an item is added below k2. This causes an imbalance at k3. Then pull k2 up. Note that after the rotation, the height of the tree is the same as it was before the insertion.

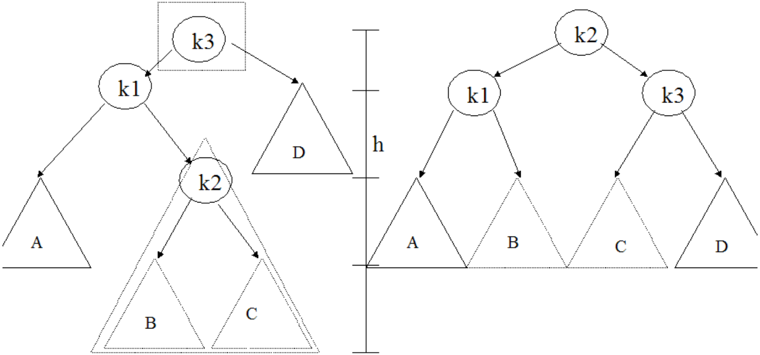
Double rotations (II)



Another Double Rotation

Suppose an item is added below k2. This causes an imbalance at k3. Then pull k2 up. Note that after the rotation, the height of the tree is the same as it was before the insertion.

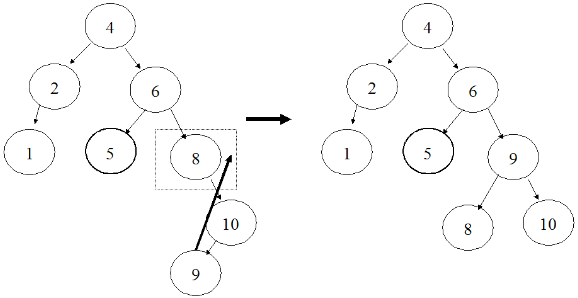
Double rotations: C++



```
void DoubleRotate1 (AvlNode *&k3) const {
        SingleRotate1( k3->left );
        SingleRotate1( k3 );
 }
```

Using double rotations in practice

An Example



Imbalance at node 8 solved with double rotation.

Which rotations to use ?

- Recognizing which rotation you have to use is the hardest part.
    1. Find the imbalanced node.
    2. Go down two nodes towards the newly inserted node.
    3. If the path is straight, use single rotation.
    4. If the path zig-zags, use double rotation.

Deleting a node

- Use deleteByCopying() to delete a node. This allows reducing the problem of deleting a node with two descendants to deleting a node with at most one descendant.
- After a node has been deleted, balance factors updated from the parent of the deleted node to the root.
- For each node whose balance becomes ± 2, a single or double rotation has to be performed to restore balance of the tree.
- Deletion: at most $O(\log n)$ rotations.
- Deletion might improve balance factor of its parent.
- It may also worsen the balance factor of its grandparent.

Wrapup

- As with the single rotations, double rotations restore the height of the subtree to what it was before the insertion.
- This guarantees that all rebalancing and height updating is complete.
- AVL trees maintain balance of binary search trees while they are being created via insertions of data.
- An alternative approach is to have trees that readjust themselves when data is accessed, making often accessed data items move to the top of the tree (splay trees).

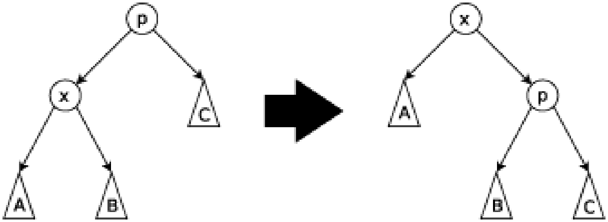
Splay trees

- Invented by Sleator and Tarjan (1985).
- to splay $\sim$ to spread out.
- Self-balancing binary trees, simpler to implement than AVL, red-black trees.
- Additional property: recently accessed elements quick to access.
- Insertion, lookup, removal: $O(\log n)$ amortized time.
- That roughly means that the average price per operation in a long sequence of operations is $O(\log n)$.
- Fundamental operation: splaying. Rearranging the tree such that certain elemen brought at the top of the tree.

Splaying

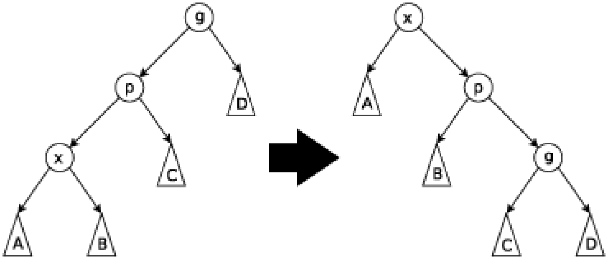
- When a node x is accessed, a splaying operation performed to bring it to the top.
- Composed of a sequence of splaying steps.
- Each splaying step brings x closer to the root.
- Steps depend on:
- Whether z is left or right child of its parent p.
- Whether p is root or not, and
- Whether p is left or right child of **its** parent g.
- Three types of splaying steps.

First case: *p* is the root



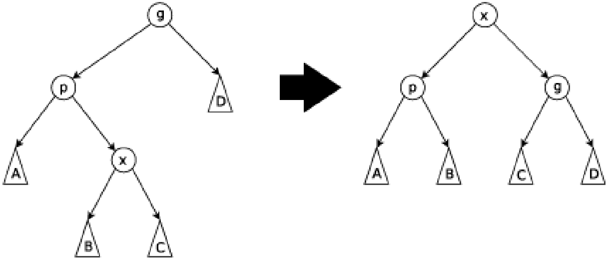
"zig": basically rotation.

Second case: *p* not the root, *x*, *p* both left or both right children



"Zigzig"

Third case: p, x alternate sides



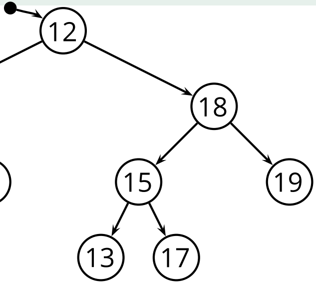
"Zigzag"

Splaying operations

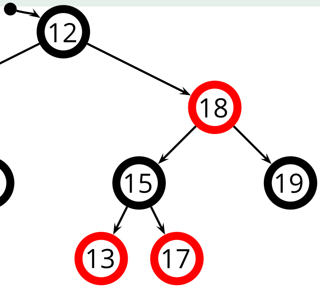
- First case: rotation.
- All cases: actually two mirror-image cases (only one shown in picture).
- Advantages: more accessed nodes closer to root. Useful for implementing caches, garbage collection.
- Disadvantages: random access worse than for other balanced BST.
- Particularly bad: access elements in sorted order.

Red-Black Trees

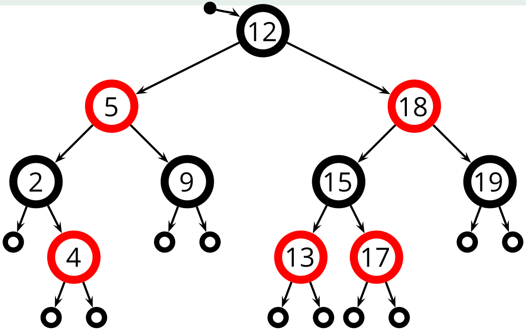
Red-Black Trees



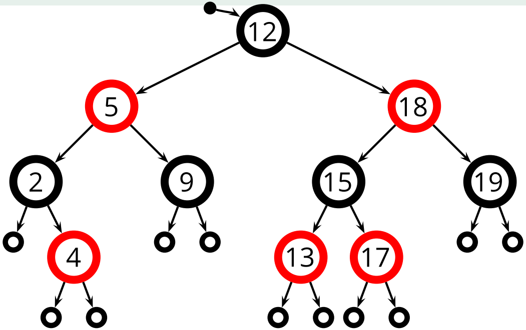
Red-Black Trees



Red-Black Trees

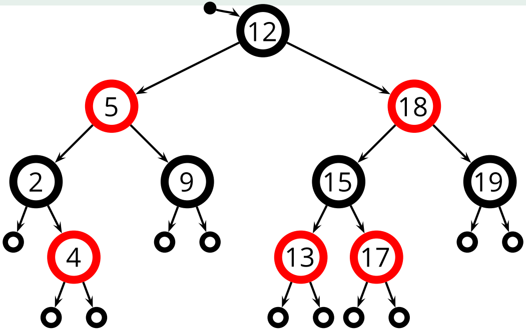


Red-Black Trees



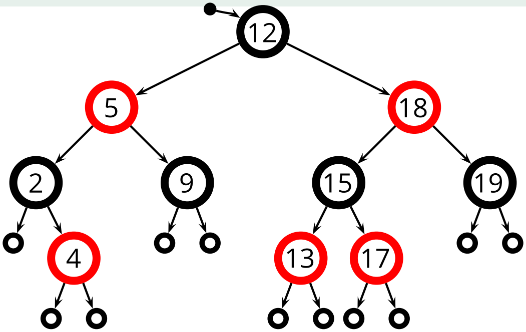
- *Red-black-tree property*

Red-Black Trees



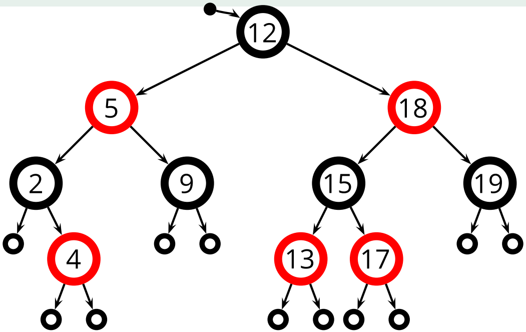
- *Red-black-tree property*
  1. every node is either **red** or **black**

Red-Black Trees



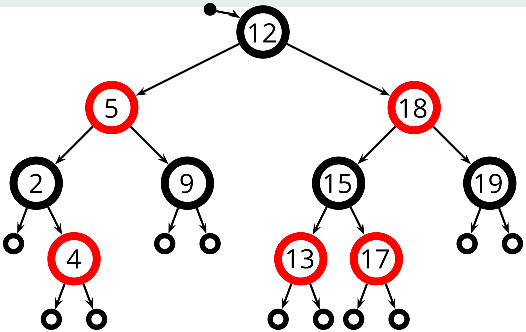
- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**

Red-Black Trees

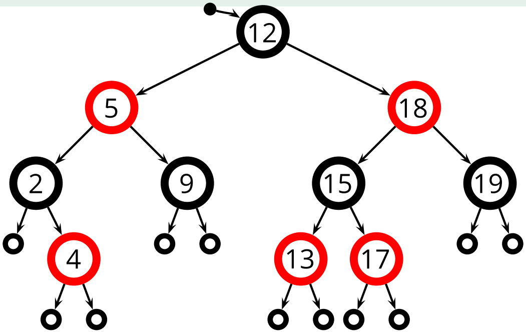


- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**

Red-Black Trees



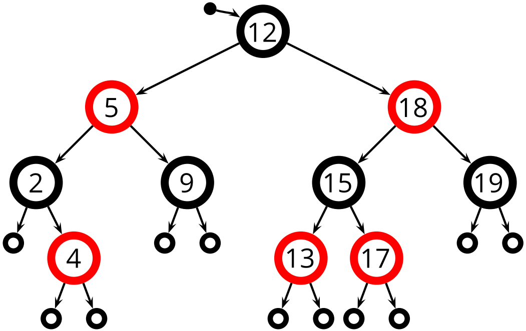
- *Red-black-tree property*
  1. every node is either **red** or **black**
  2. the root is **black**
  3. every (NIL) leaf is **black**
  4. if a node is **red**, then both its children are **black**



- *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)

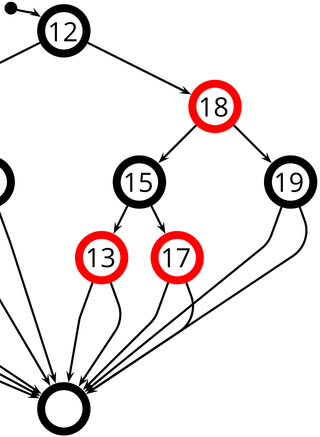
Red-Black Trees (2)

- *Implementation*



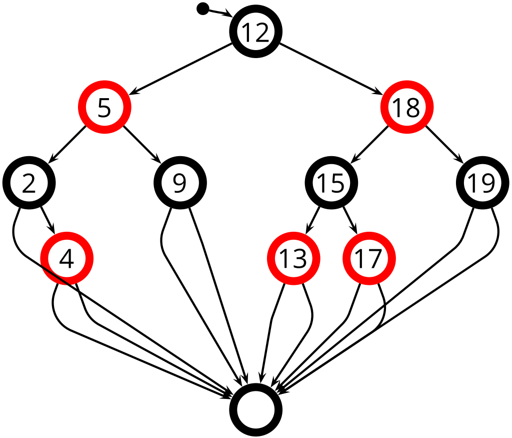
Red-Black Trees (2)

- *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

Red-Black Trees (3)

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

Red-Black Trees (3)

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

Red-Black Trees (3)

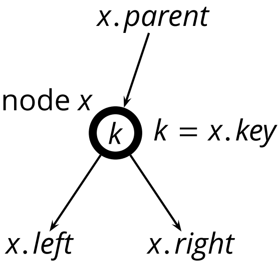
- *Implementation*
  - T represents the tree, which consists of a set of *nodes*
  - $T.root$ is the root node of tree T
  - $T.nil$ is the “sentinel” node of tree T

Red-Black Trees (3)

- *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

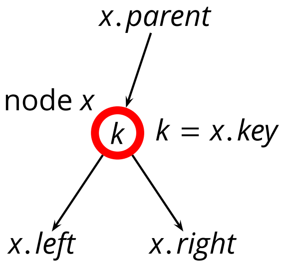


Red-Black Trees (3)

- *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

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 $2\log(n+1)$.

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

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. ***base case:*** x is a leaf, so $size(x) = 0$ and $bh(x) = 0$
    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. ***base case:*** x is a leaf, so $size(x) = 0$ and $bh(x) = 0$
   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:
      $size(x) = size(y_1) + size(y_2) + 1 \geq (2^{bh(y_1)} - 1) + (2^{bh(y_2)} - 1) + 1$

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

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

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

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

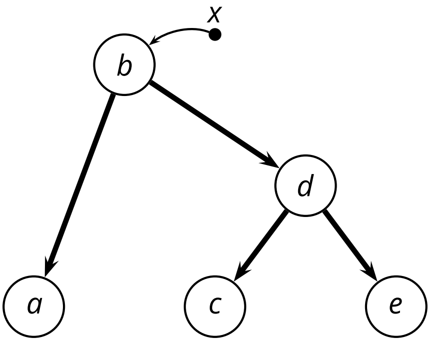
$$h \leq 2\log(n+1)$$

- 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

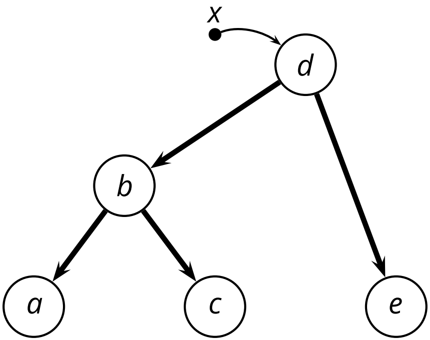
$$\boxed{T(n) = O(\log n)}$$

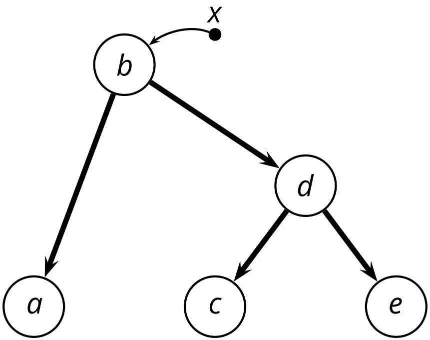
  - which is also the *worst-case* complexity

Rotation



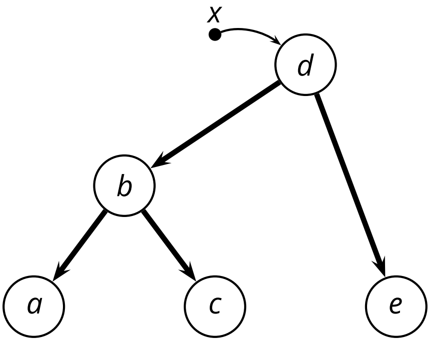
Rotation





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

Rotation



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

Red-Black Insertion

- **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*

Red-Black Insertion

- **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*)

Red-Black Insertion

- **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*

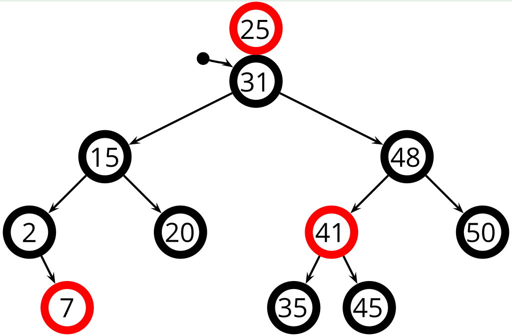
Red-Black Insertion

- **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

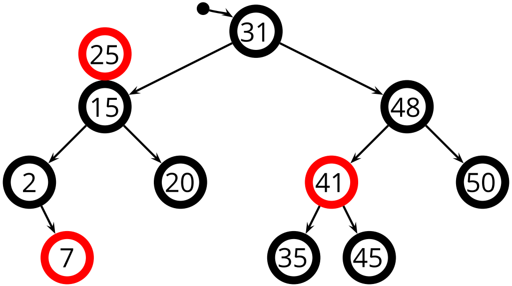
RB-INSERT

```
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)
```

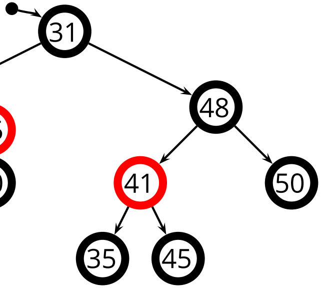
Red-Black Insertion (2)



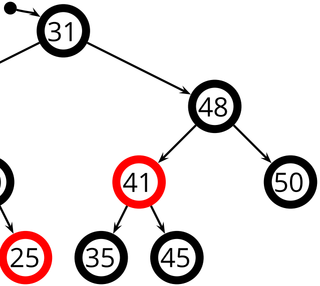
Red-Black Insertion (2)



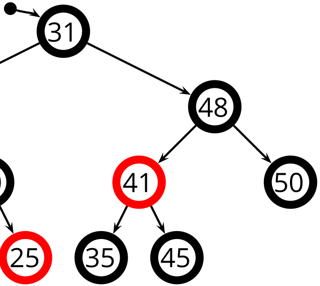
Red-Black Insertion (2)



Red-Black Insertion (2)



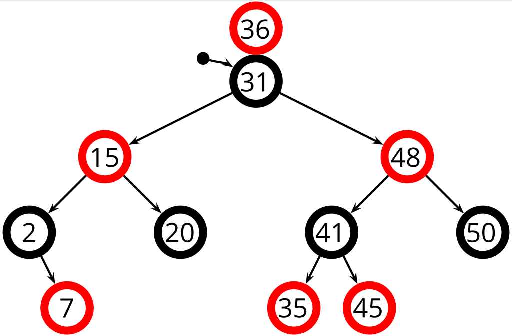
Red-Black Insertion (2)



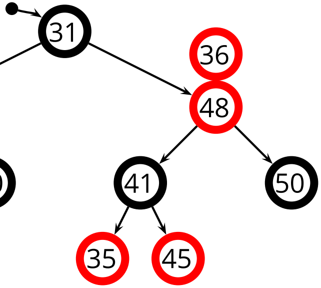
- *z*'s father is **black**, so no fixup needed

Red-Black Insertion (3)

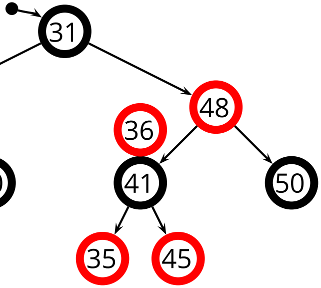
Red-Black Insertion (3)



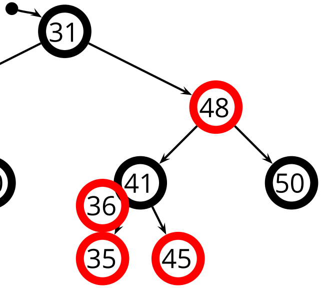
Red-Black Insertion (3)



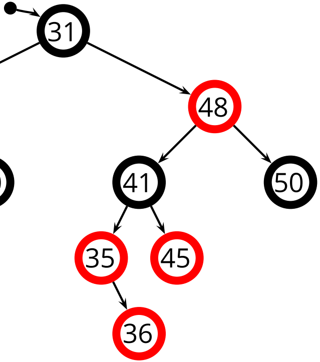
Red-Black Insertion (3)



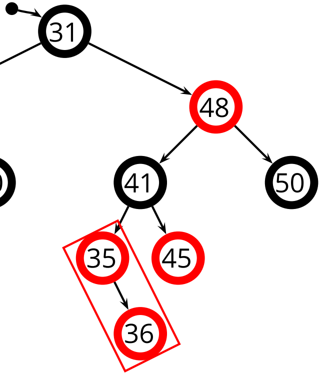
Red-Black Insertion (3)



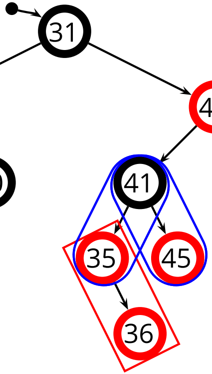
Red-Black Insertion (3)



Red-Black Insertion (3)

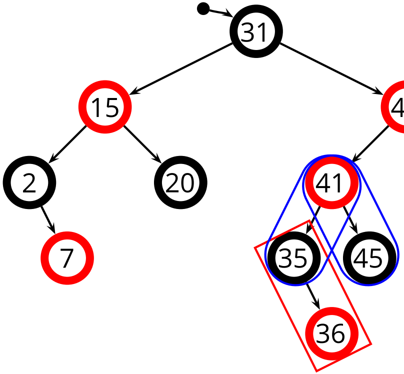


Red-Black Insertion (3)



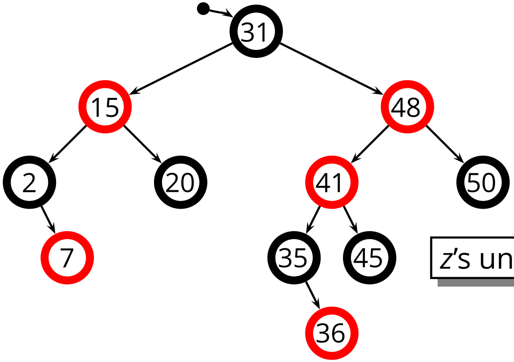
z's uncle is **red**

Red-Black Insertion (3)



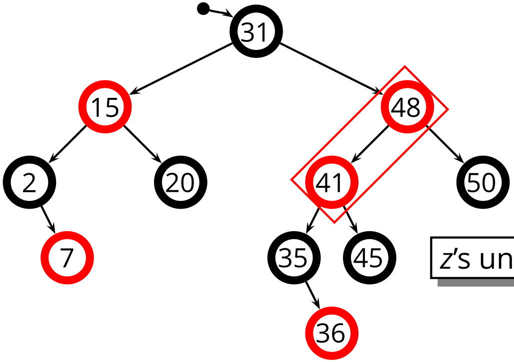
z's uncle is **red**

Red-Black Insertion (3)



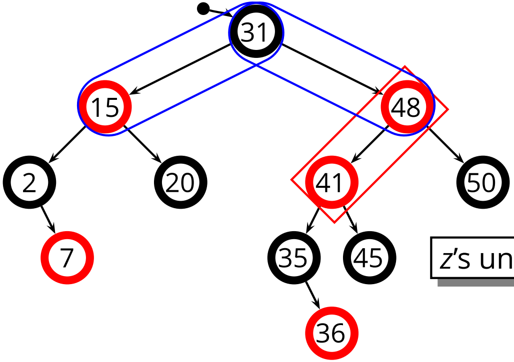
z's uncle is **red**

Red-Black Insertion (3)



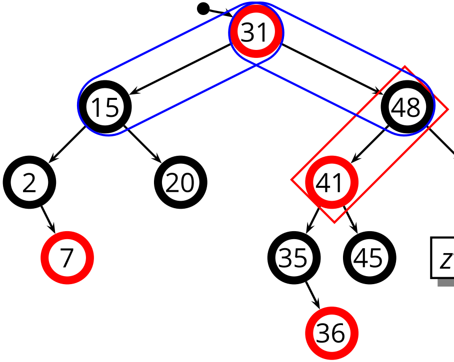
z's uncle is **red**

Red-Black Insertion (3)



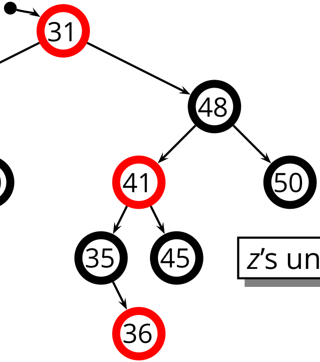
z's uncle is **red**

Red-Black Insertion (3)



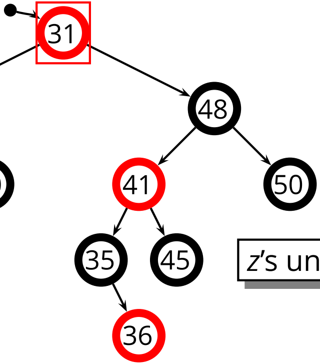
z's uncle is **red**

Red-Black Insertion (3)



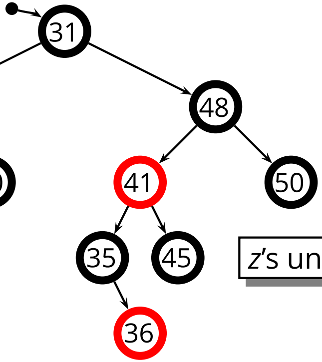
z's uncle is **red**

Red-Black Insertion (3)



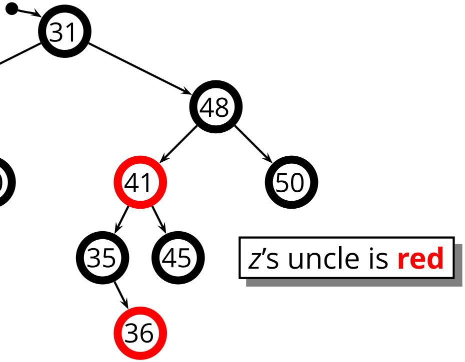
z's uncle is **red**

Red-Black Insertion (3)



z's uncle is **red**

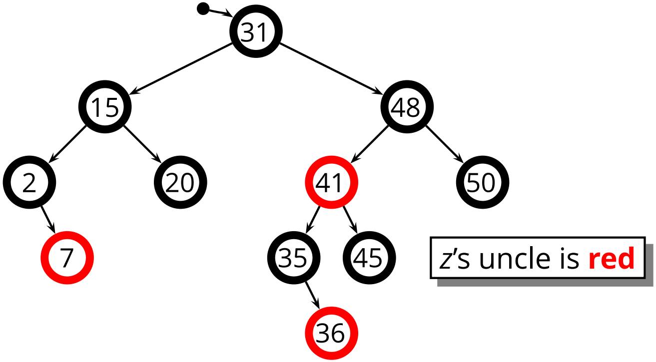
Red-Black Insertion (3)



z's uncle is **red**

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

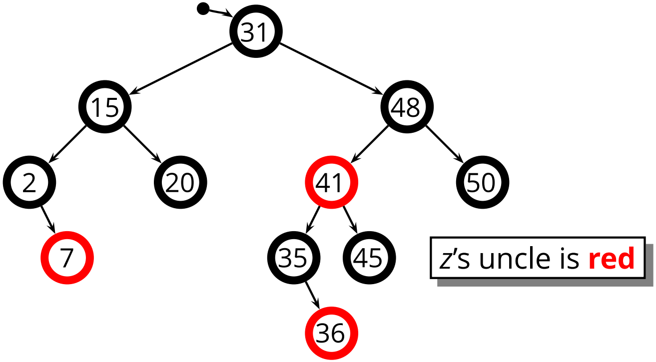
Red-Black Insertion (3)



z’s uncle 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. . .

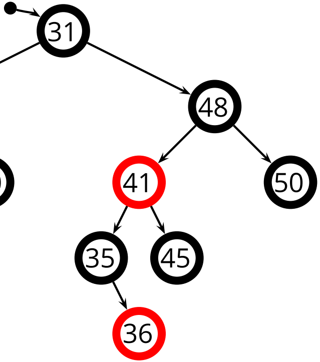
Red-Black Insertion (3)



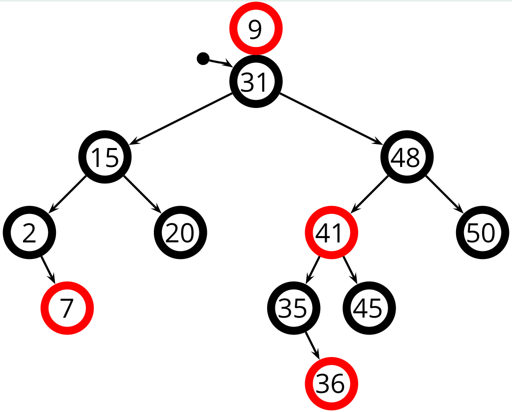
z’s uncle is **red**

- 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

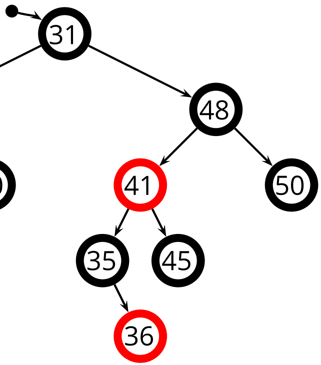
Red-Black Insertion (4)



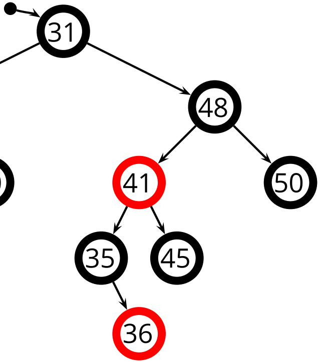
Red-Black Insertion (4)



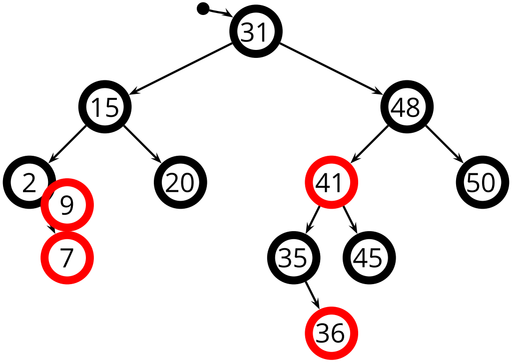
Red-Black Insertion (4)



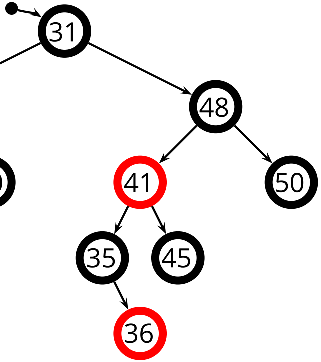
Red-Black Insertion (4)



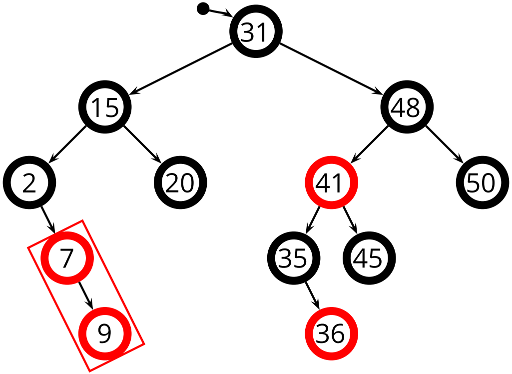
Red-Black Insertion (4)



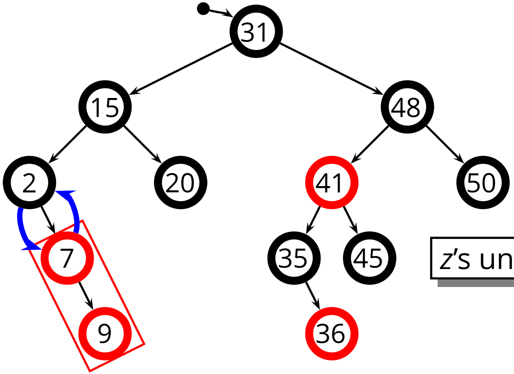
Red-Black Insertion (4)



Red-Black Insertion (4)

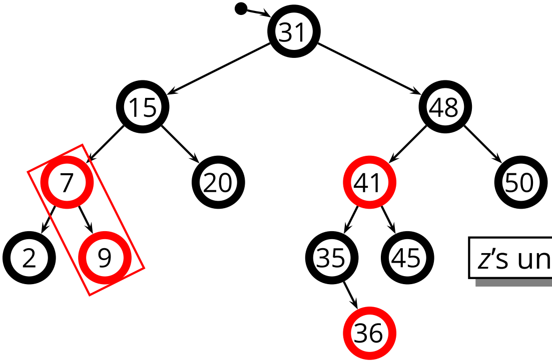


Red-Black Insertion (4)



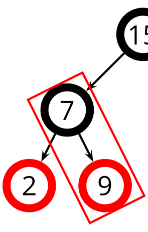
z's uncle is **black**

Red-Black Insertion (4)



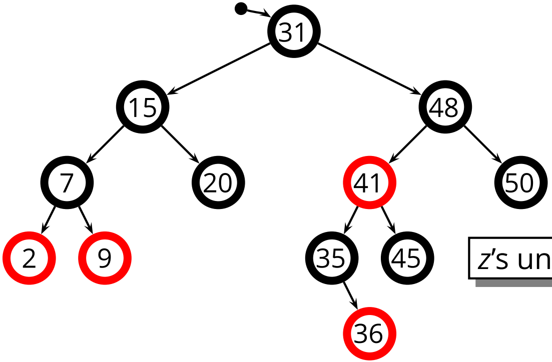
z's uncle is **black**

Red-Black Insertion (4)



z's uncle is **black**

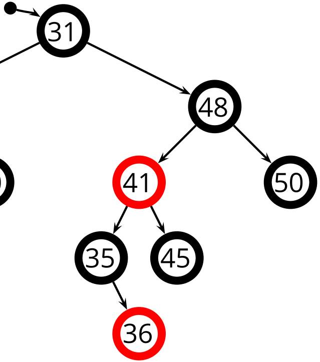
Red-Black Insertion (4)



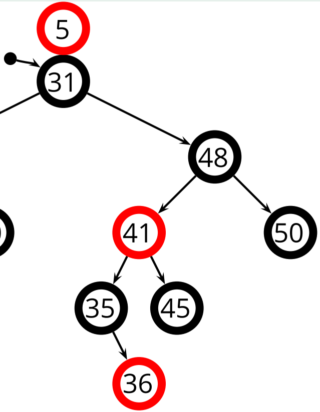
z's uncle is **black**

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

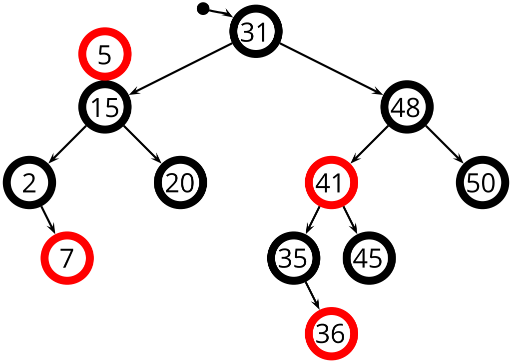
Red-Black Insertion (5)



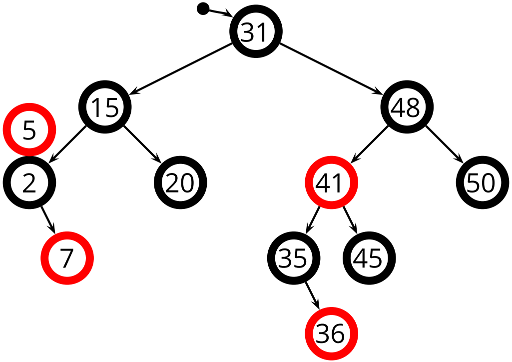
Red-Black Insertion (5)



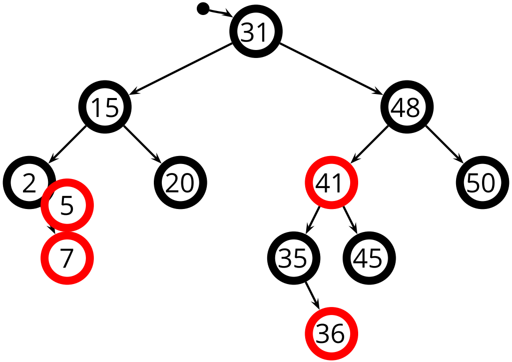
Red-Black Insertion (5)



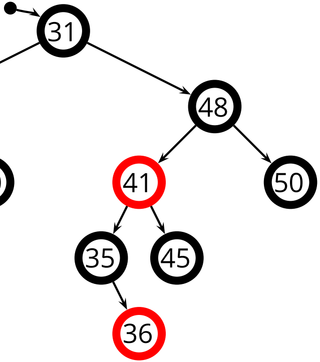
Red-Black Insertion (5)



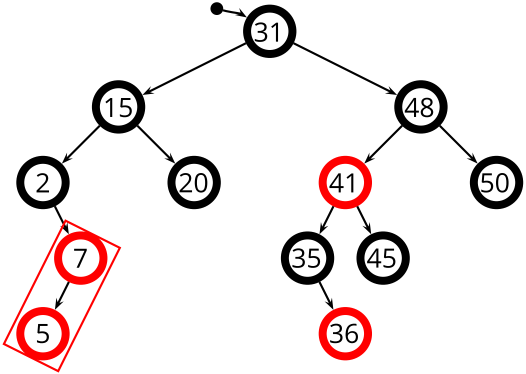
Red-Black Insertion (5)



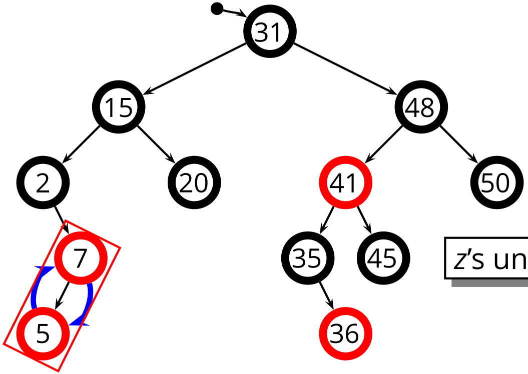
Red-Black Insertion (5)



Red-Black Insertion (5)

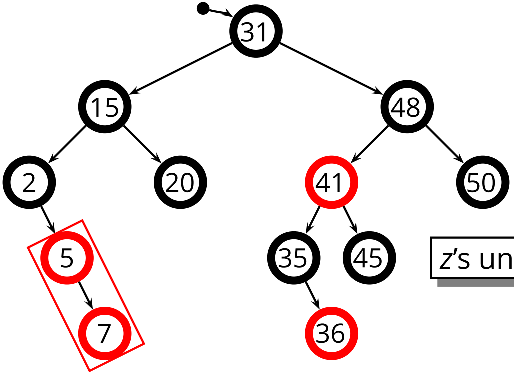


Red-Black Insertion (5)



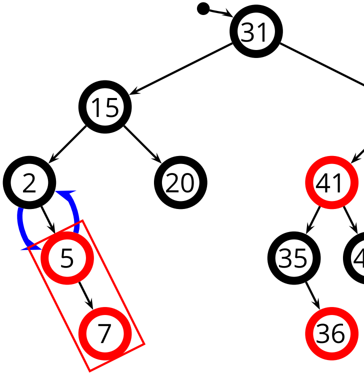
z's uncle is **black**

Red-Black Insertion (5)



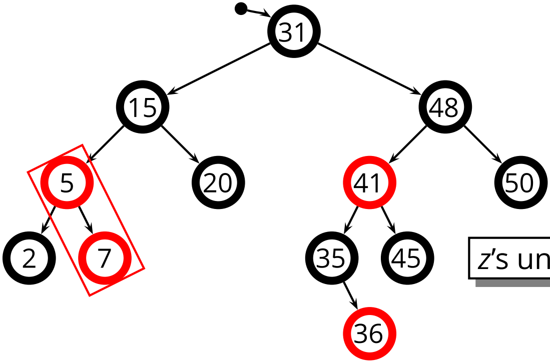
z's uncle is **black**

Red-Black Insertion (5)



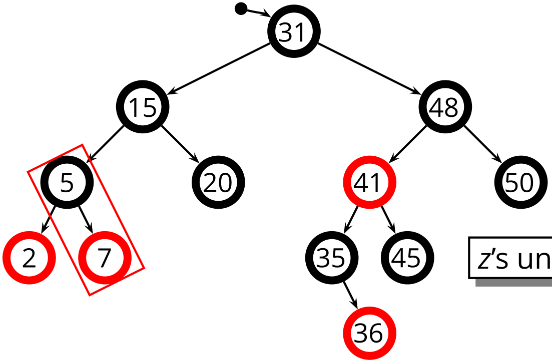
z's uncle is **black**

Red-Black Insertion (5)



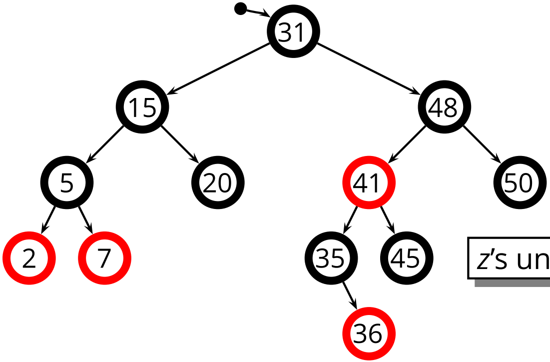
z's uncle is **black**

Red-Black Insertion (5)



z's uncle is **black**

Red-Black Insertion (5)

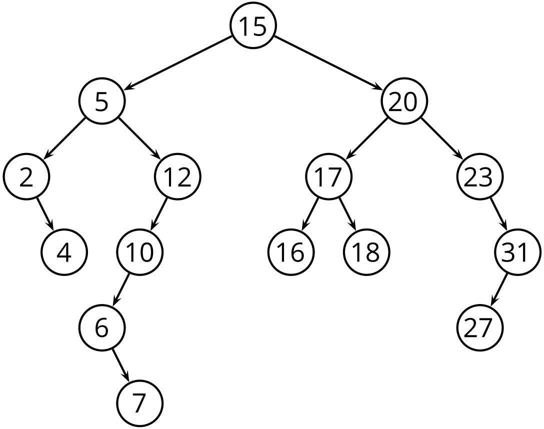


z's uncle is **black**

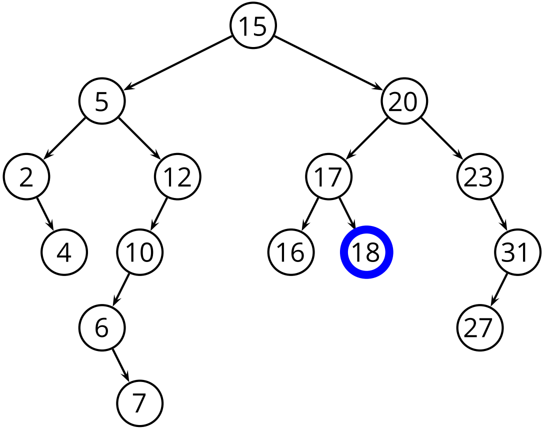
- A *zig-zag* **red–red** conflicts can be resolved with a rotation to turn it into an *in-line* conflict, and then a rotation plus a color switch

Recap on Deletion in Binary Trees

Recap on Deletion in Binary Trees

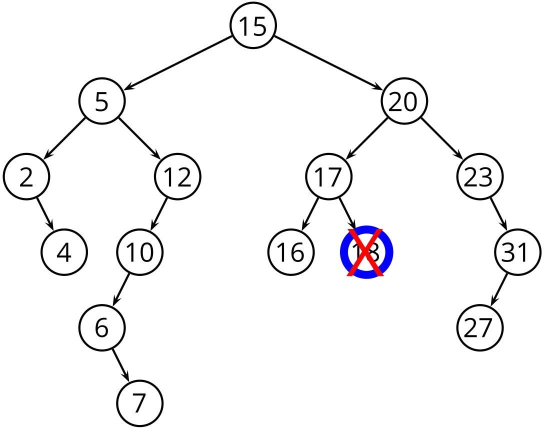


Recap on Deletion in Binary Trees



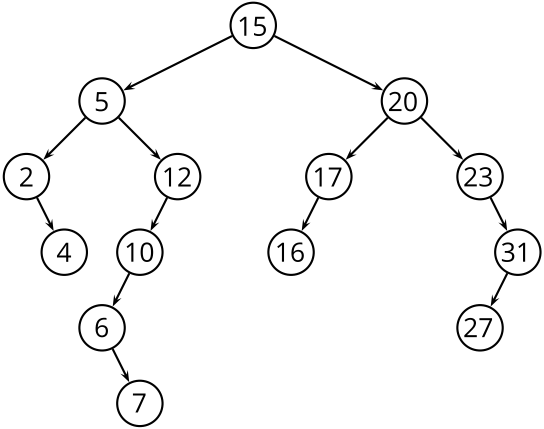
1. *z* has no children

Recap on Deletion in Binary Trees



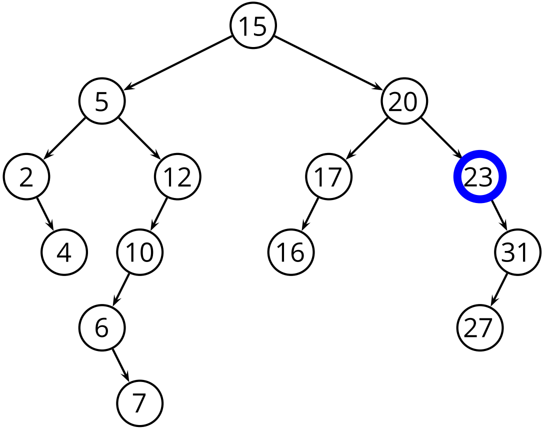
1. *z* has no children
   - simply remove *z*

Recap on Deletion in Binary Trees



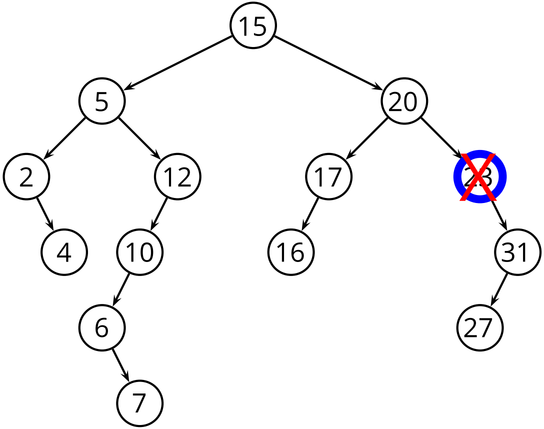
1. *z* has no children
    - simply remove *z*

Recap on Deletion in Binary Trees



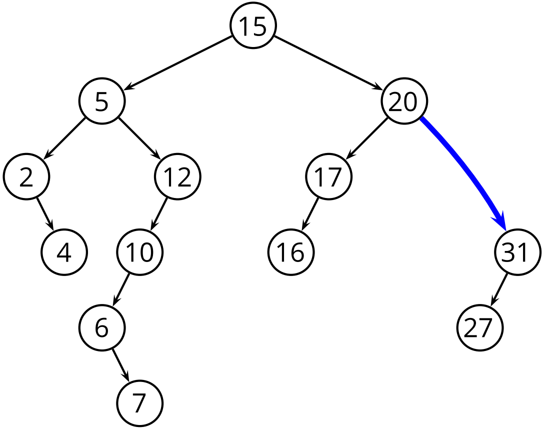
1. *z* has no children
    - simply remove *z*
2. *z* has one child

Recap on Deletion in Binary Trees



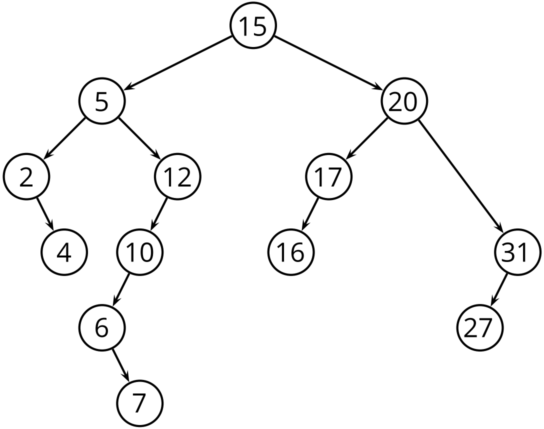
1. *z* has no children
   - simply remove *z*
2. *z* has one child
   - remove *z*

Recap on Deletion in Binary Trees



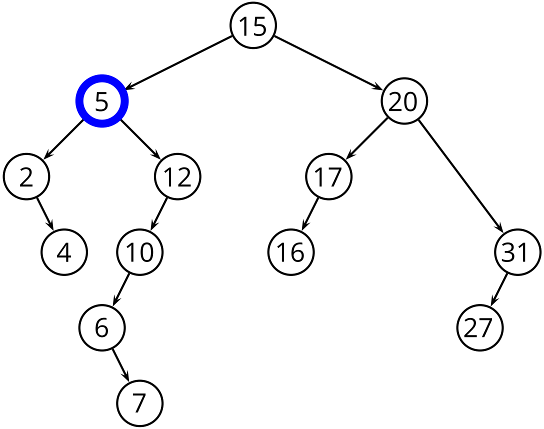
1. *z* has no children
    - simply remove *z*
2. *z* has one child
    - remove *z*
    - connect $z.parent$ to $z.right$

Recap on Deletion in Binary Trees



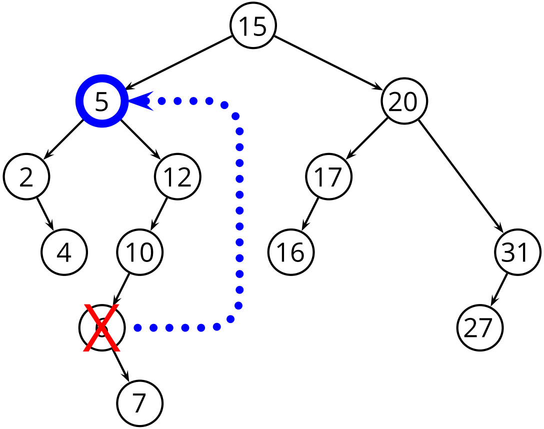
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect $z.parent$ to $z.right$

Recap on Deletion in Binary Trees



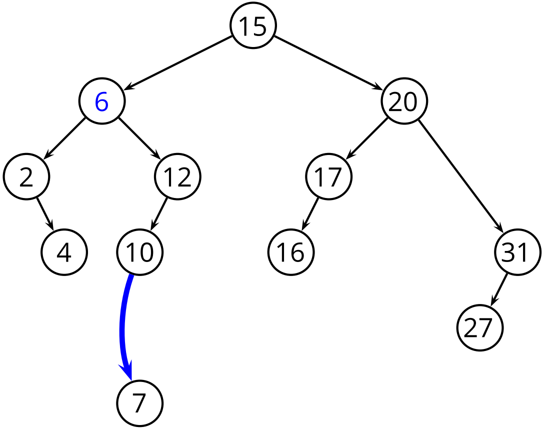
1. *z* has no children
    - simply remove *z*
2. *z* has one child
    - remove *z*
    - connect $z.parent$ to $z.right$
3. *z* has two children

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 $y =$ **Tree-Successor**(z)
   - remove y (1 child!)

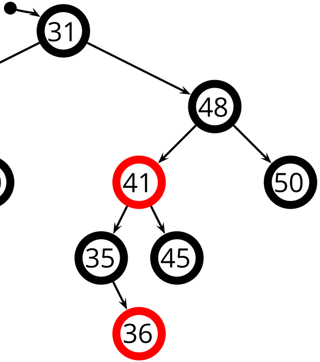
Recap on Deletion in Binary Trees



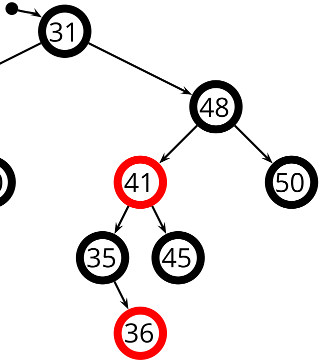
1. z has no children
    - simply remove z
2. z has one child
    - remove z
    - connect $z.parent$ to $z.right$
3. z has two children
    - replace z with $y = \textbf{Tree-Successor}(z)$
    - remove y (1 child!)
    - connect $y.parent$ to $y.right$

Red-Black Deletion

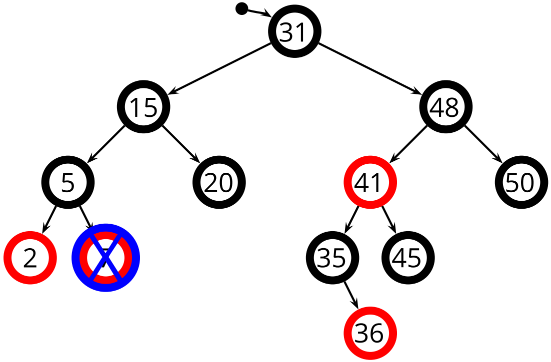
Red-Black Deletion



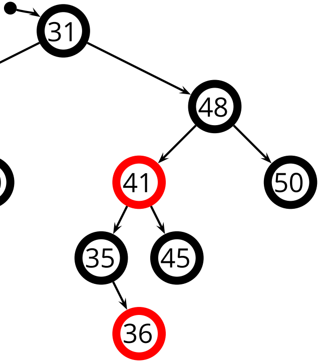
Red-Black Deletion



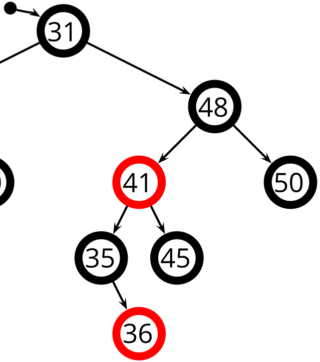
Red-Black Deletion



Red-Black Deletion

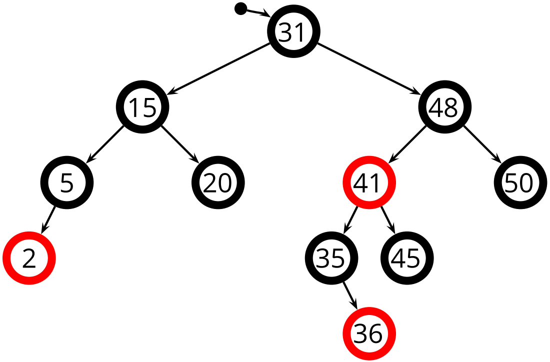


Red-Black Deletion



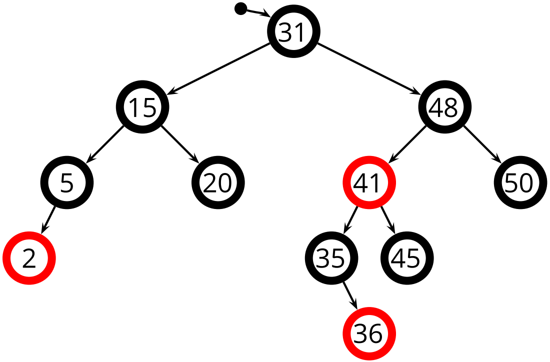
- A deleting a **red** *leaf* does not require any adjustment

Red-Black Deletion



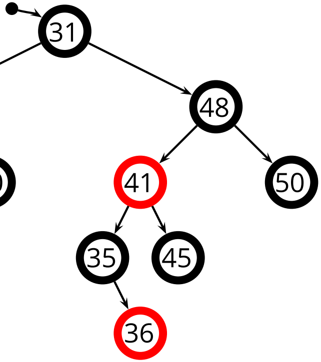
- A deleting a **red** *leaf* does not require any adjustment
  - the deletion does not affect the black height (property 5)

Red-Black Deletion

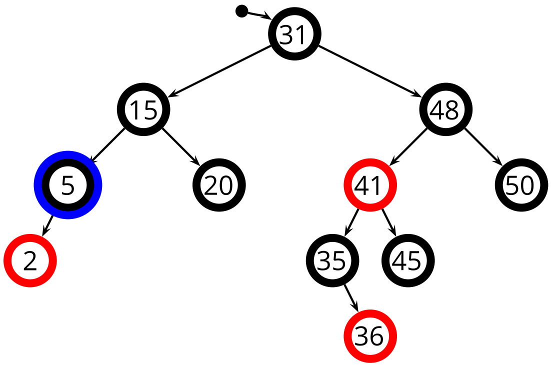


- A deleting a **red** *leaf* does not require any adjustment
    - the deletion does not affect the black height (property 5)
    - no two red nodes become adjacent (property 4)

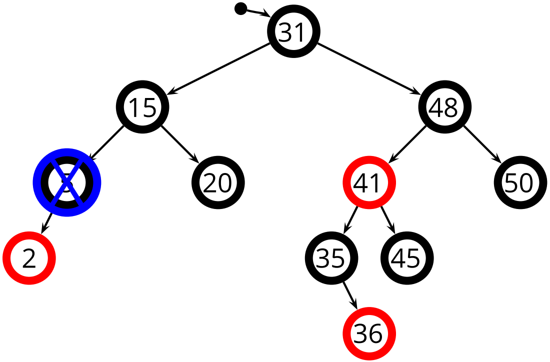
Red-Black Deletion (2)



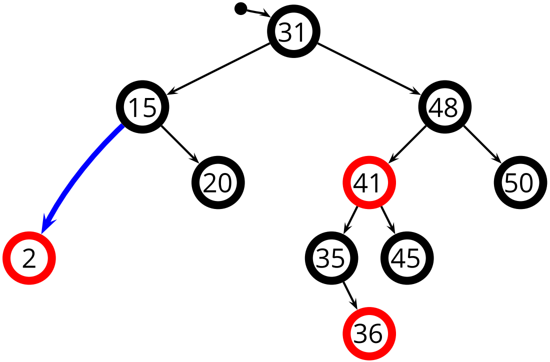
Red-Black Deletion (2)



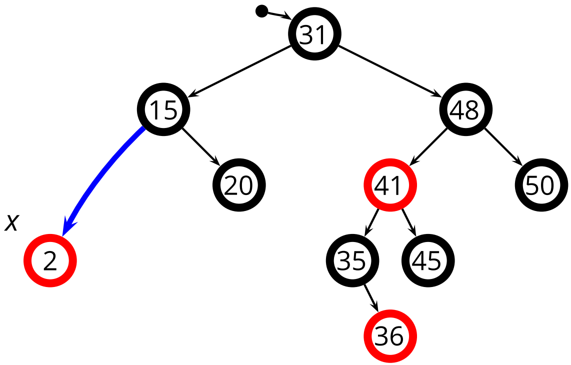
Red-Black Deletion (2)



Red-Black Deletion (2)

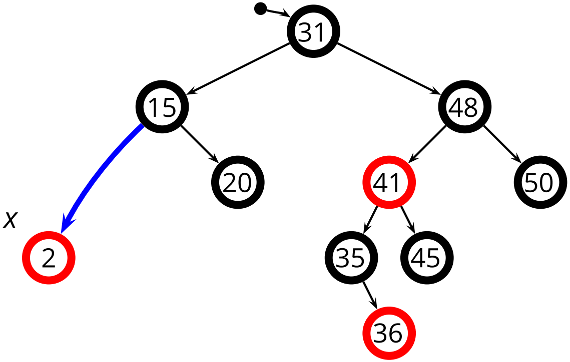


Red-Black Deletion (2)



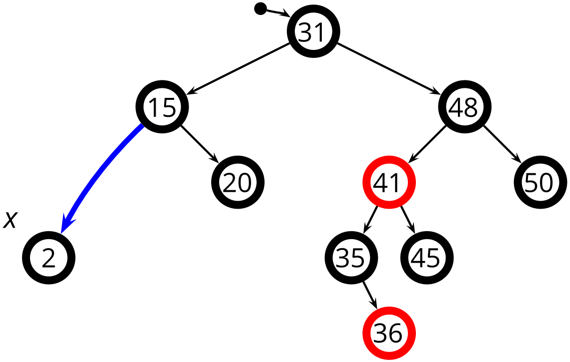
- Deleting a ***black*** *node* changes the balance of black-height in a subtree *x*

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

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
  - in this simple case: $x.color$ = BLACK

Fixup Conditions

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**

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**
- 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

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**
- 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**)

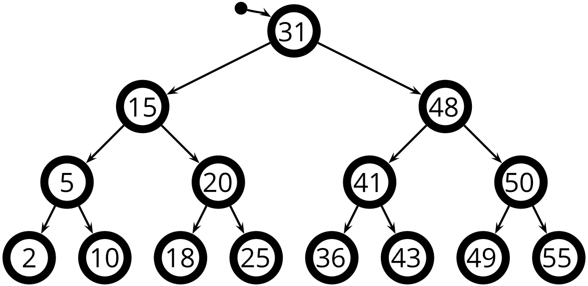
Fixup Conditions

- 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**
- 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)

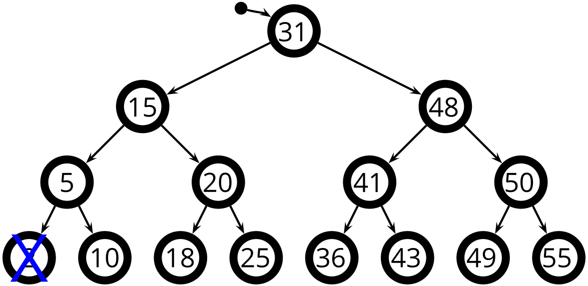
Fixup Conditions

- 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**
- 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)

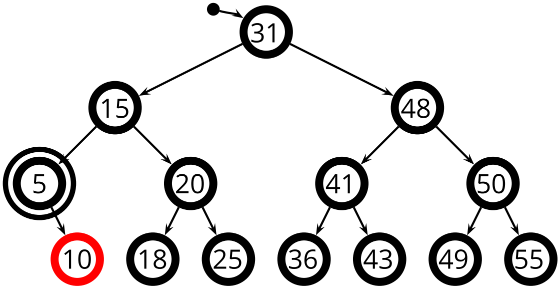
Red-Black Deletion (3)



Red-Black Deletion (3)

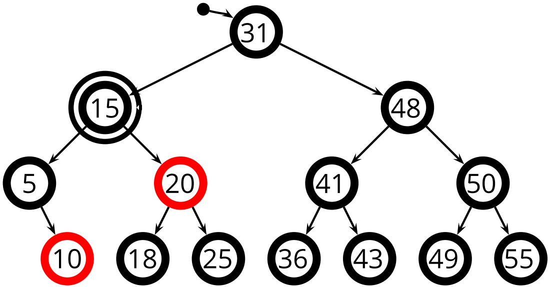


Red-Black Deletion (3)

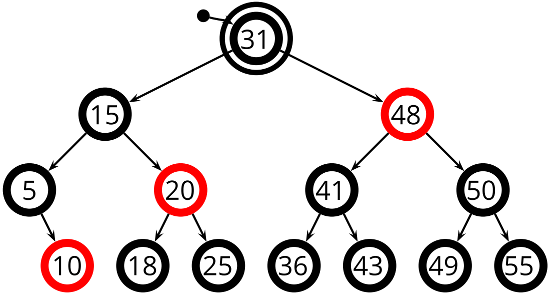


- *x* carries an *additional* ***black*** *weight*
    - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)

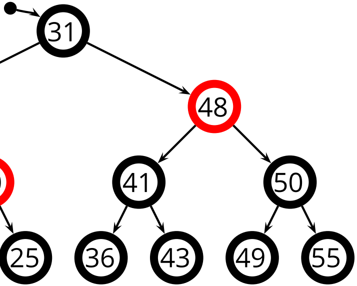


- *x* carries an *additional **black** weight*
    - the fixup algorithm pushes it up towards to root



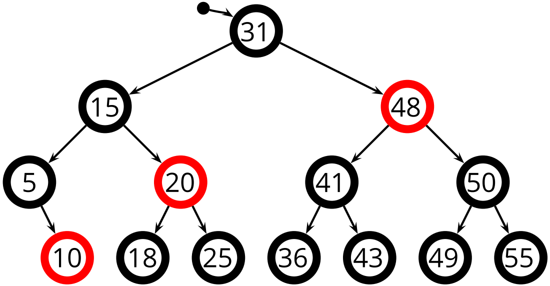
- *x* carries an *additional **black** weight*
    - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)



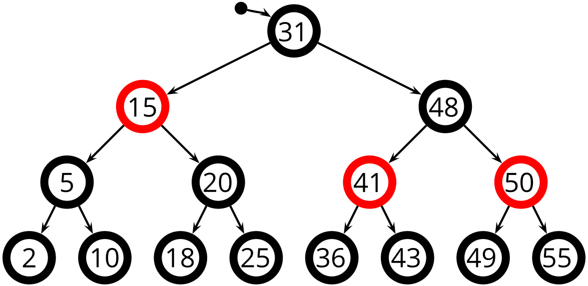
- *x* carries an *additional **black** weight*
  - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)

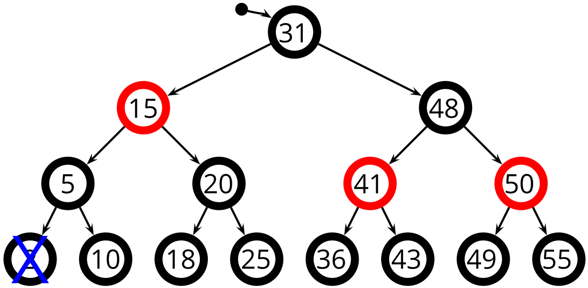


- *x* carries an *additional **black** weight*
    - the fixup algorithm pushes it up towards to root
- The *additional **black** weight* can be discarded if it reaches the *root*, otherwise. . .

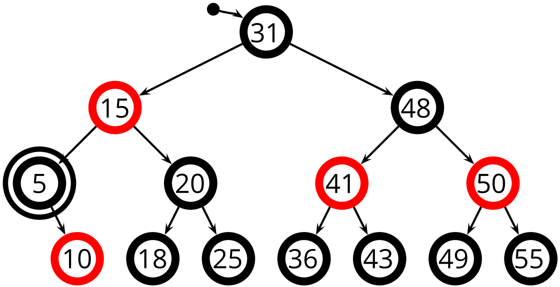
Red-Black Deletion (4)



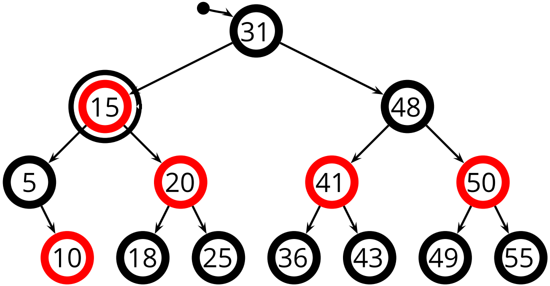
Red-Black Deletion (4)



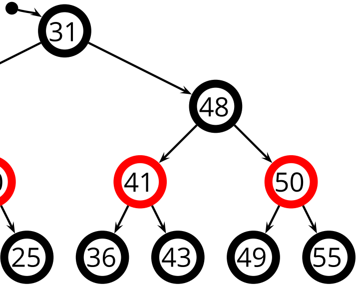
Red-Black Deletion (4)



Red-Black Deletion (4)

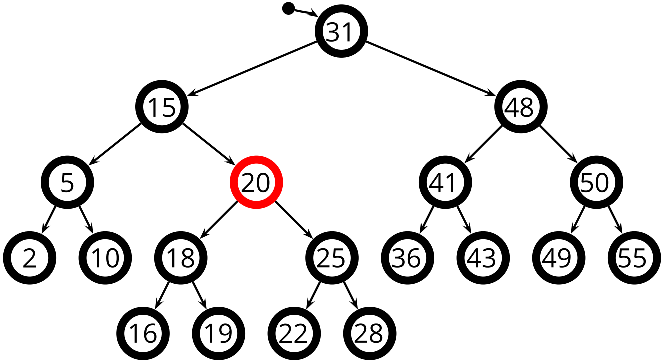


Red-Black Deletion (4)

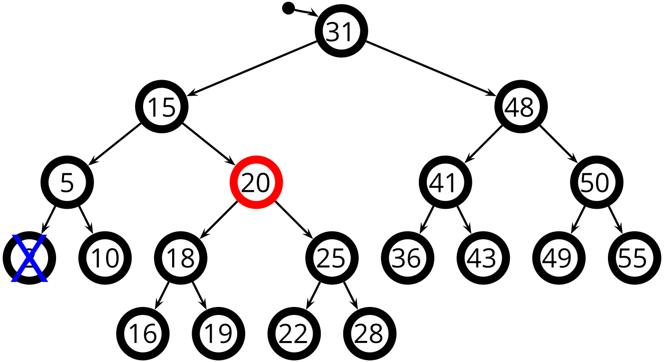


- The *additional **black** weight* can also stop as soon as it reaches a **red** node, which will absorb the extra **black** color

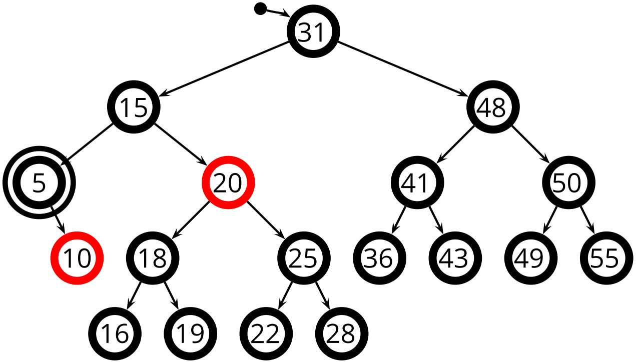
Red-Black Deletion (5)



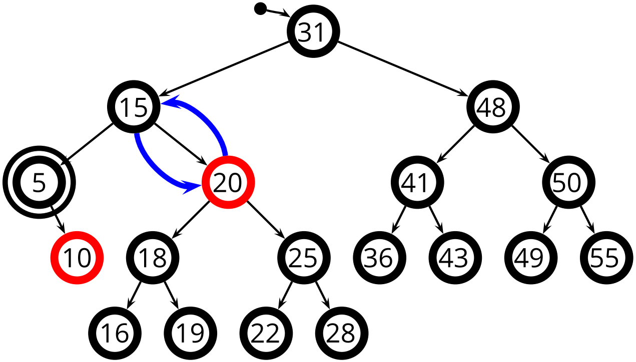
Red-Black Deletion (5)



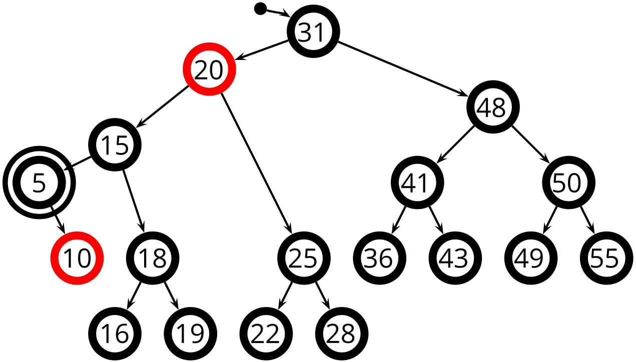
Red-Black Deletion (5)



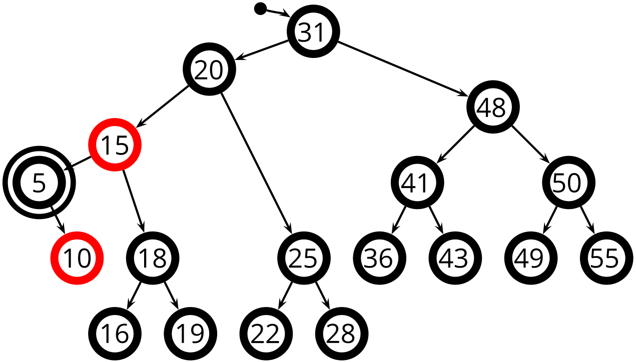
Red-Black Deletion (5)



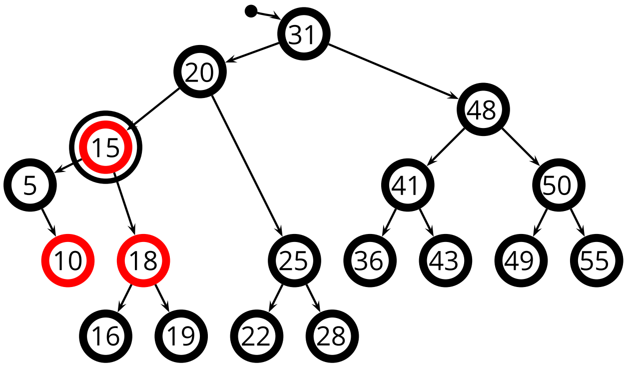
Red-Black Deletion (5)



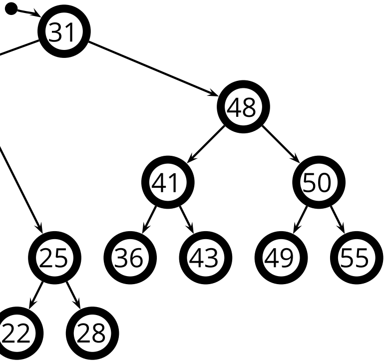
Red-Black Deletion (5)



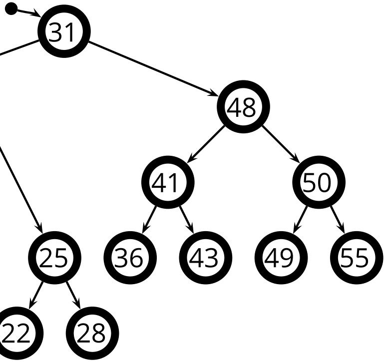
Red-Black Deletion (5)



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

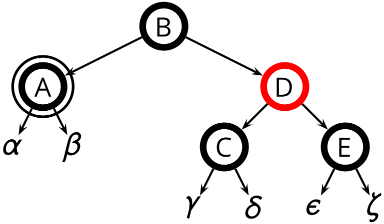
Basic Fixup Iteration (1)

Basic Fixup Iteration (1)

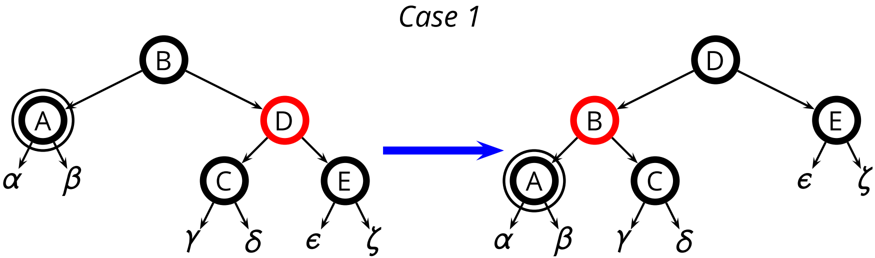
Case 1

Basic Fixup Iteration (1)

Case 1

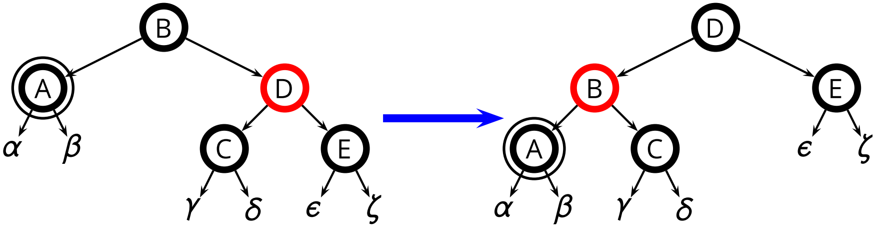


Basic Fixup Iteration (1)



Basic Fixup Iteration (1)

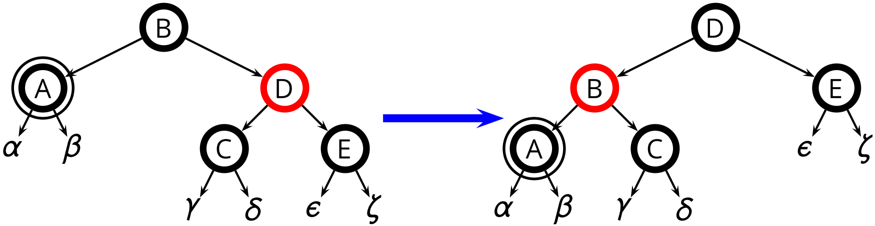
Case 1



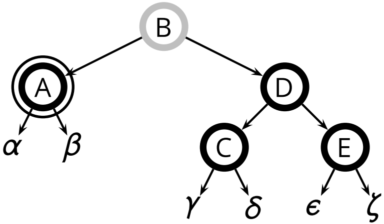
Case 2

Basic Fixup Iteration (1)

Case 1

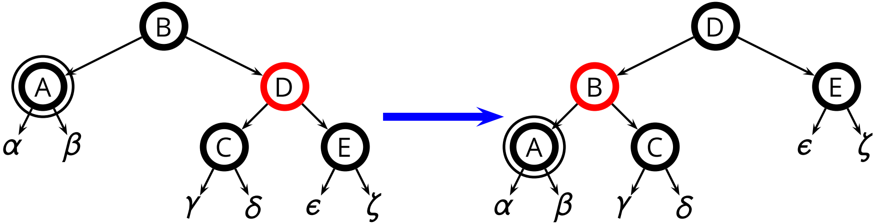


Case 2

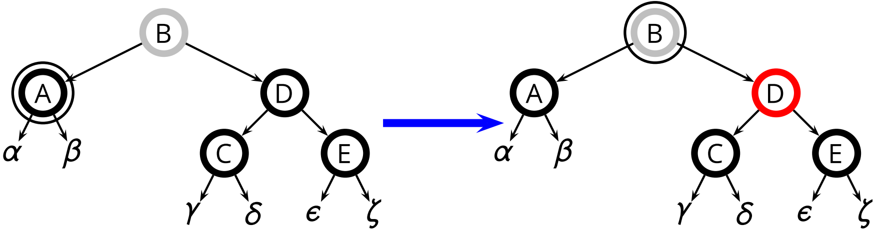


Basic Fixup Iteration (1)

Case 1



Case 2

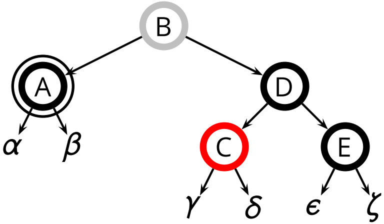


Basic Fixup Iteration (2)

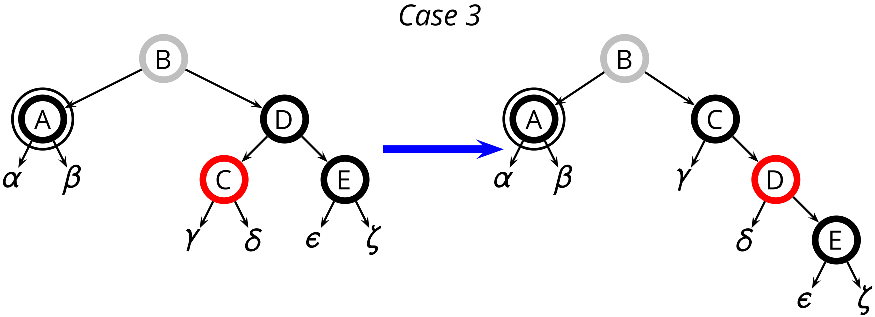
Case 3

Basic Fixup Iteration (2)

Case 3

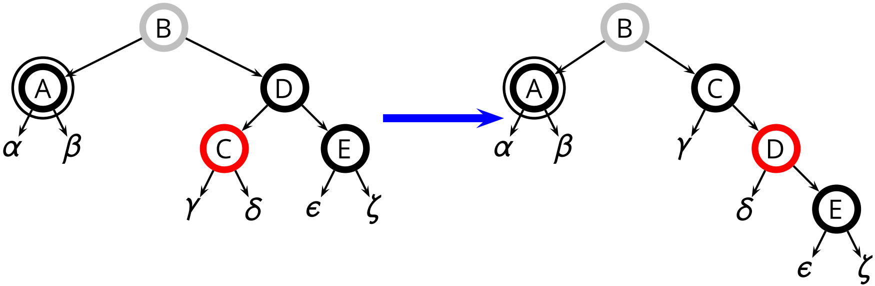


Basic Fixup Iteration (2)

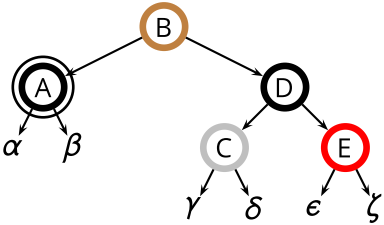


Basic Fixup Iteration (2)

Case 3

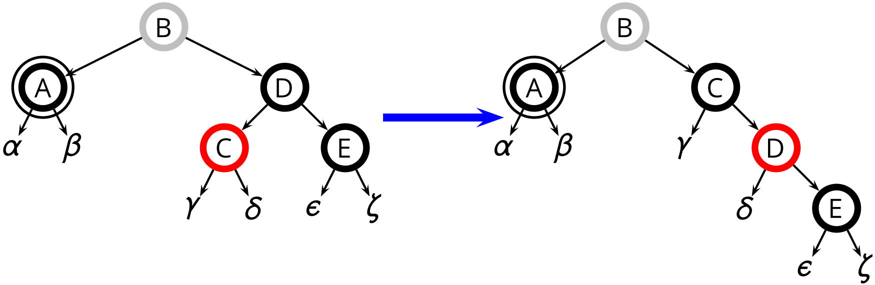


Case 4

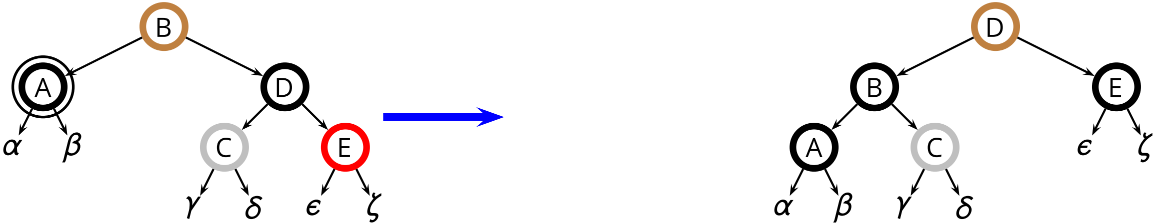


Basic Fixup Iteration (2)

Case 3



Case 4



Red-Black Delete Fixup

```
RB-DELETE-FIXUP($T, x$)
while $x \neq T.root \wedge x.color = \text{BLACK}$
    if $x == x.parent.left$
        $w = x.parent.right$
        if $w.color == \text{RED}$
            case 1...
        if $w.left.color == \text{BLACK} \wedge w.right.color = \text{BLACK}$
            $w.color = \text{RED}$                    // case 2
            $x = x.parent$
        else if $w.right.color == \text{BLACK}$
                case 3...
            case 4...
    else same as above, exchanging right and left
$x.color = \text{BLACK}$
```