Red-Black Trees

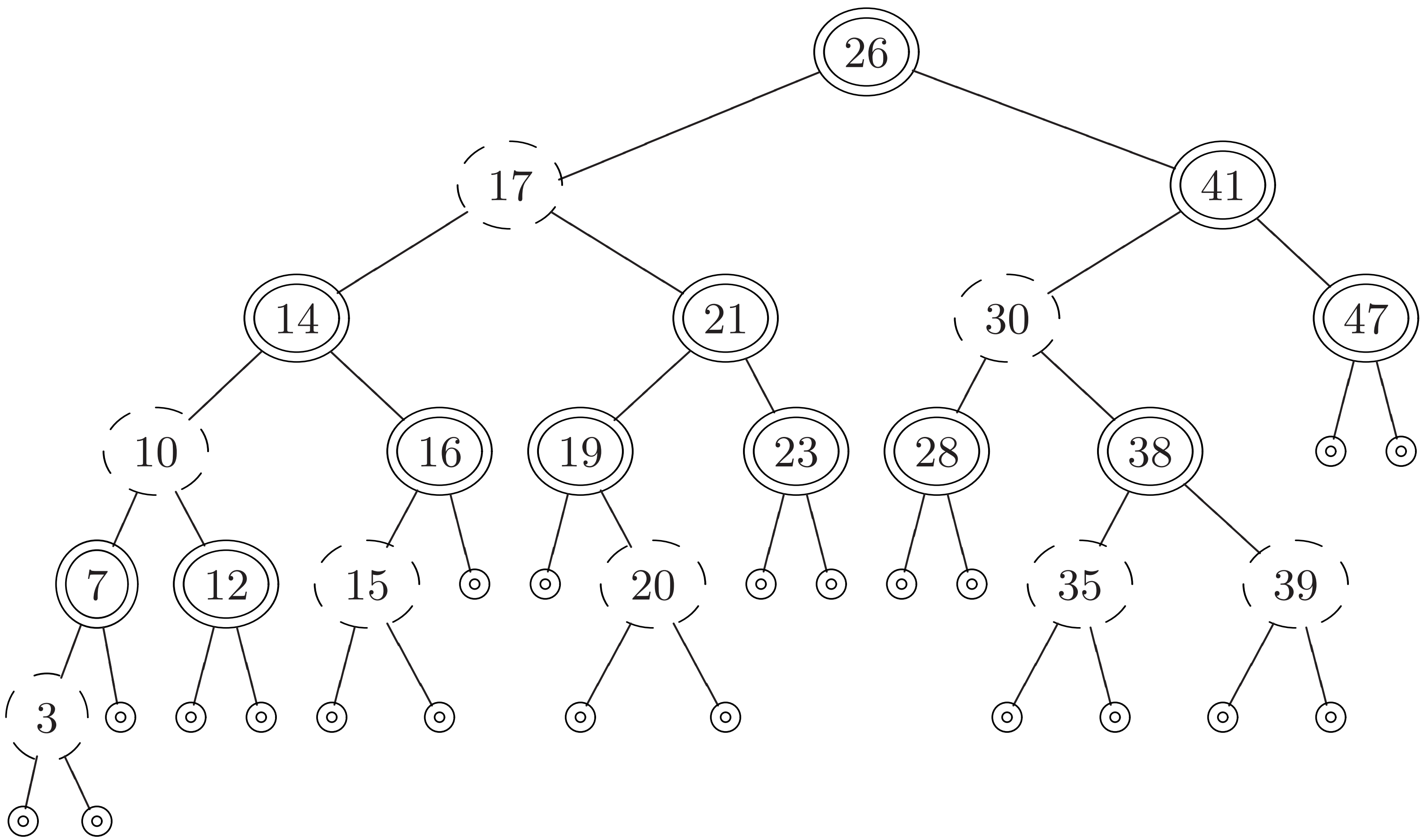
Red-Black Trees

Definition. A **red-black tree** is a binary search tree with the following properties:

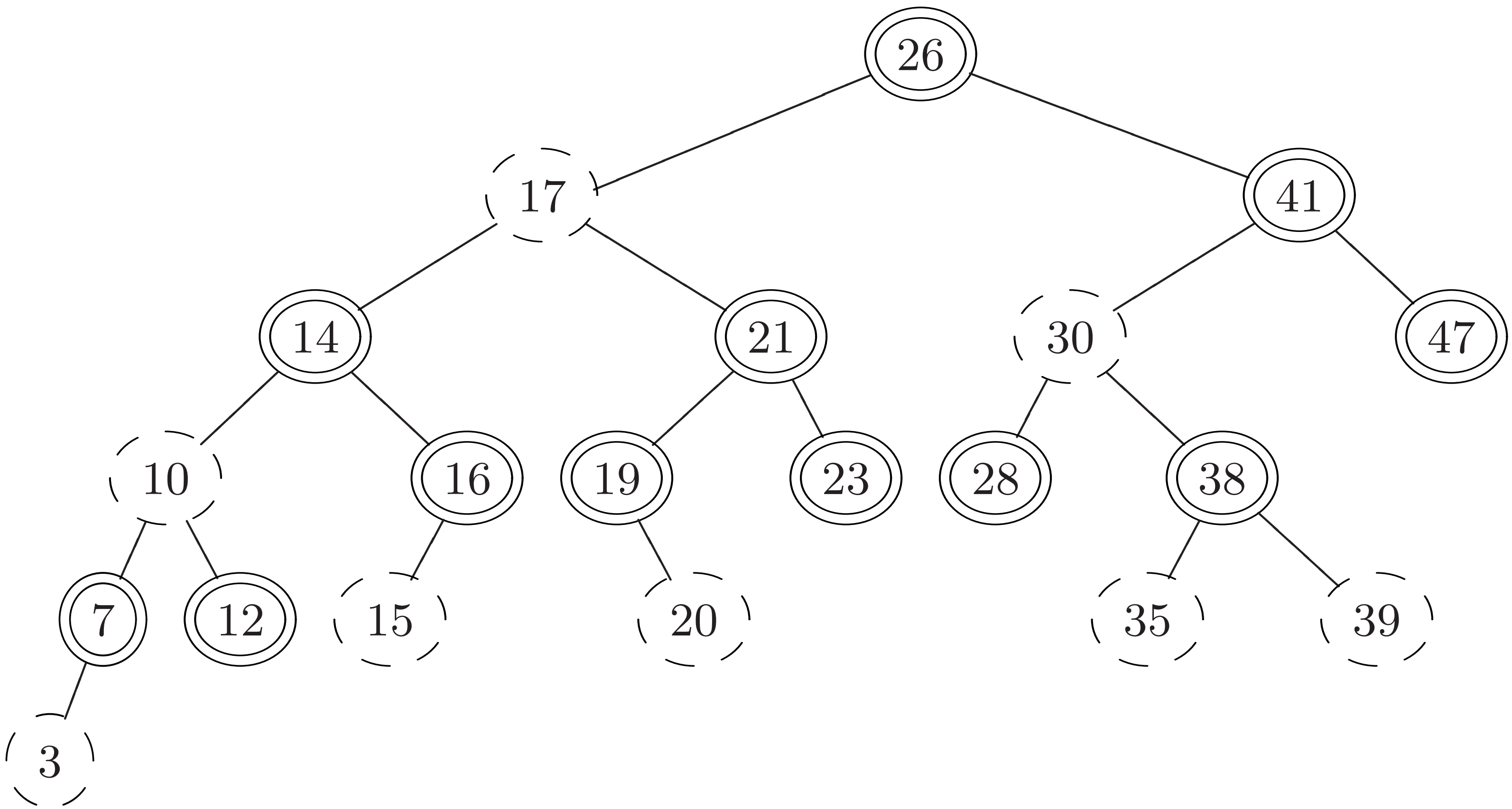
- Every node is either red or black;
- The root is black;
- Every leaf is **NIL** and is black;
- If a node is red, then both its children are black;
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

(Note: Every node in a binary tree is either a leaf or has BOTH a left AND right child.)

Red-Black Tree: Example

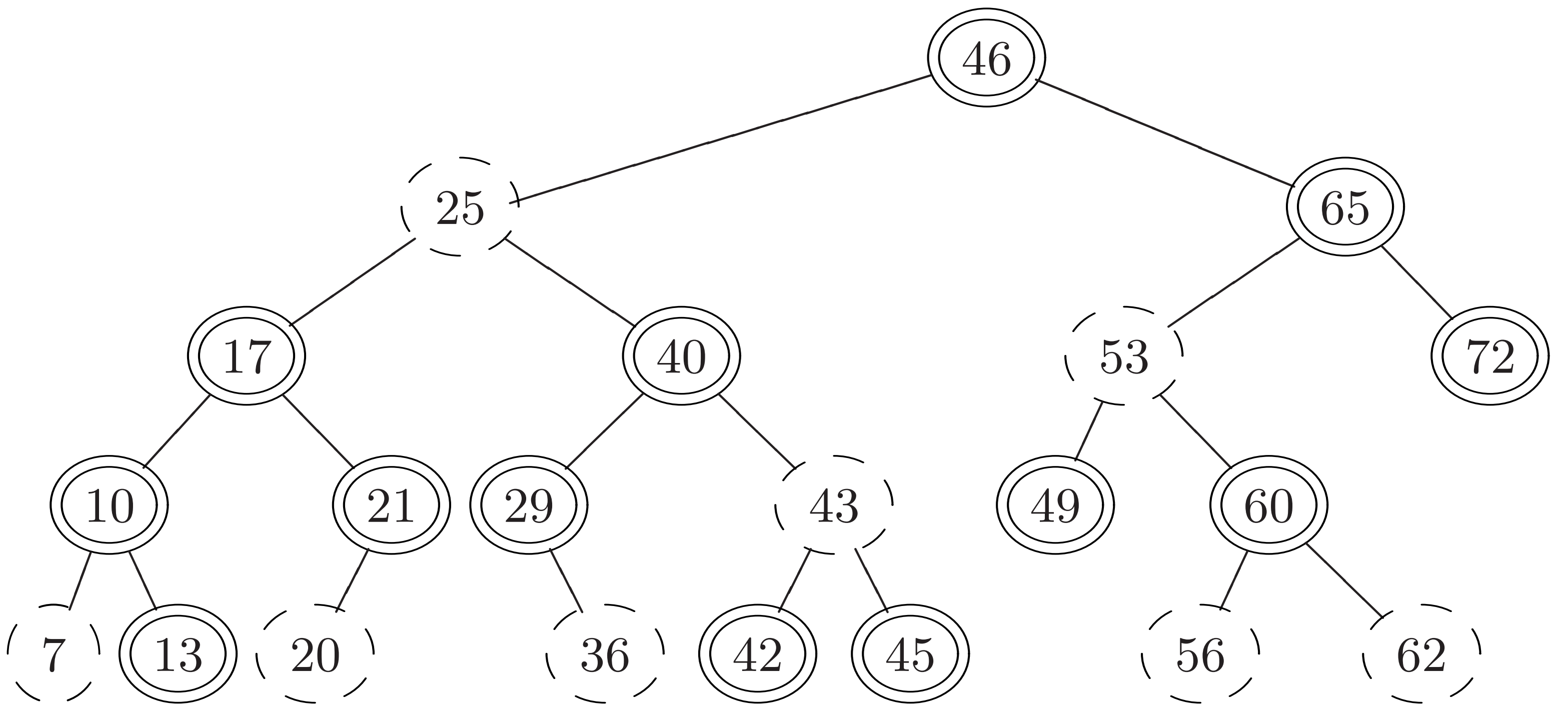


Red-Black Tree: Example



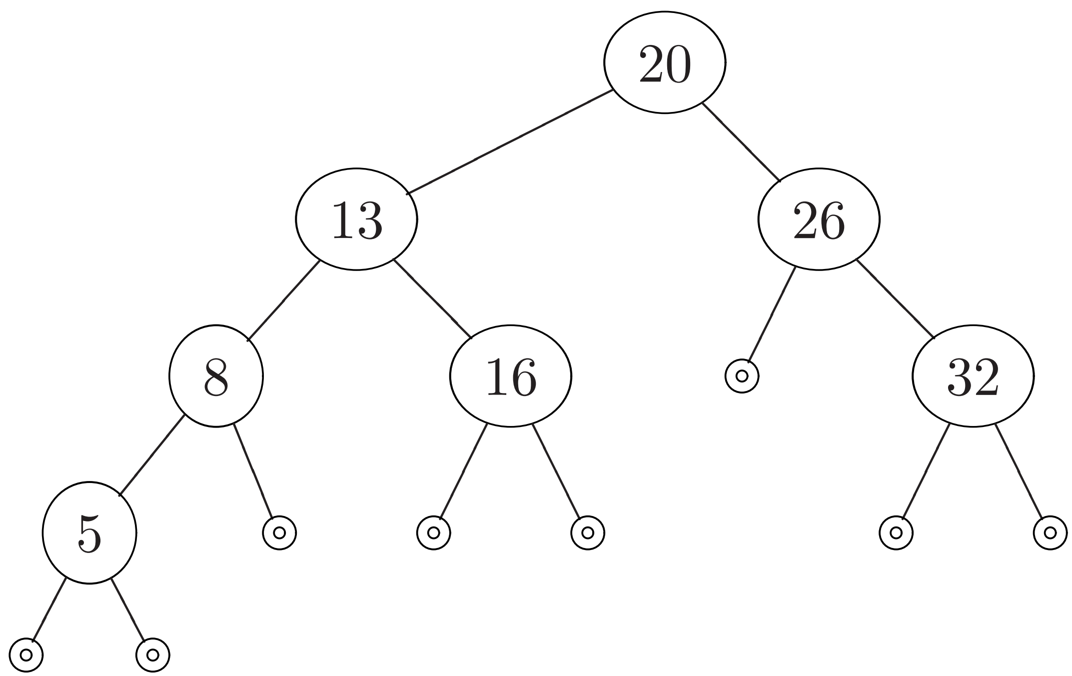
<u>NOT</u> a Red-Black Tree

This tree is <u>NOT</u> a red-black tree. Why not?



Red-Black Tree Exercise

Color the following tree so that it is a red-black tree:



Red-Black Tree Height

Definition. A **red-black tree** satisfies the following properties:

- Every node is either red or black;
- The root is black;
- Every leaf is **NIL** and is black;
- If a node is red, then both its children are black;
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Theorem. A red-black tree with n internal nodes has height at most $2\log_2(n+1)$.

Complete Binary Tree Size

Theorem 1. A complete binary search tree of height h has $2^{h+1} - 1$ nodes.

Proof. Level i has 2^i nodes ($i = 0, 1, \ldots, h$).

$1 + 2 + 2^2 + 2^3 + \ldots + 2^h = 2^{h+1} - 1$. □

Red-Black Tree Size

Theorem 2. A red-black tree of height h has at least $2^{\lceil h/2 \rceil} - 1$ internal nodes.

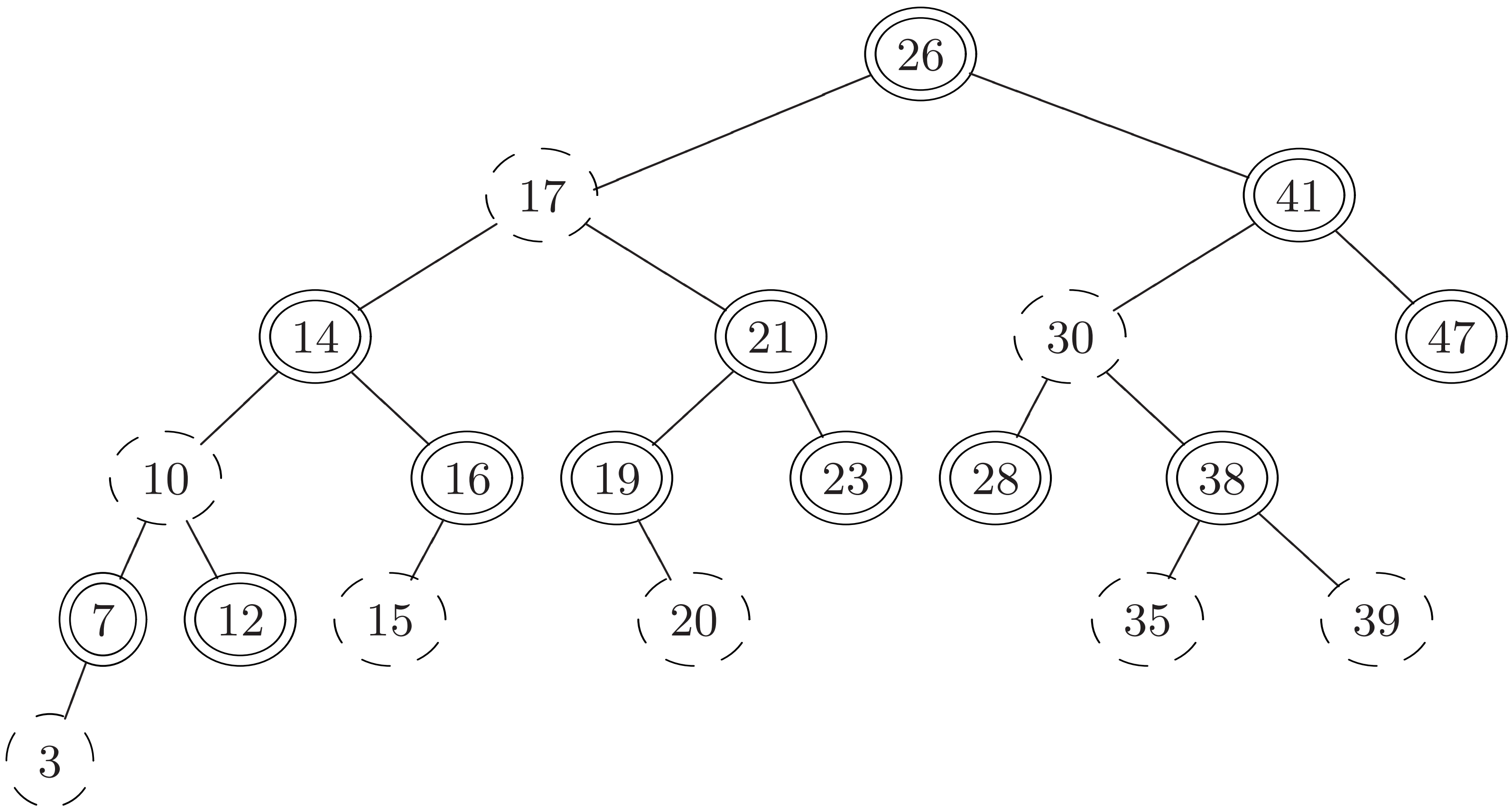
Proof. (By Dr. Y. Wang.)
Let T be a red-black tree of height h.
Remove the leaves of T forming a tree T' of height $h - 1$.
Let r be the root of T'.
Since no child of a red node is red and r is black, the longest path from r to a leaf has at least $\lceil h/2 \rceil$ black nodes.
Since every path from r to a leaf has the same number of black nodes, every path from r to a leaf has at least $\lceil h/2 \rceil$ black nodes.
Thus, T' contains a complete binary tree of height at least $\lceil h/2 \rceil - 1$.
(Note: height = Number of EDGES on longest path from root to leaf.)
By Theorem 1, tree T' has at least $2^{\lceil h/2 \rceil - 1 + 1} - 1$ nodes so tree T has at least $2^{\lceil h/2 \rceil} - 1$ internal nodes. □

Red-Black Tree Height

Theorem. A red-black tree with n internal nodes has height at most $2\log_2(n+1)$.

Proof. Let h be the height of a red-black tree with n nodes.
By Theorem 2, $n \geq 2^{\lceil h/2 \rceil} - 1$.
Thus, $\log_2(n+1) \geq \lceil h/2 \rceil \geq h/2$ so $h \leq 2\log_2(n+1)$. $\square$

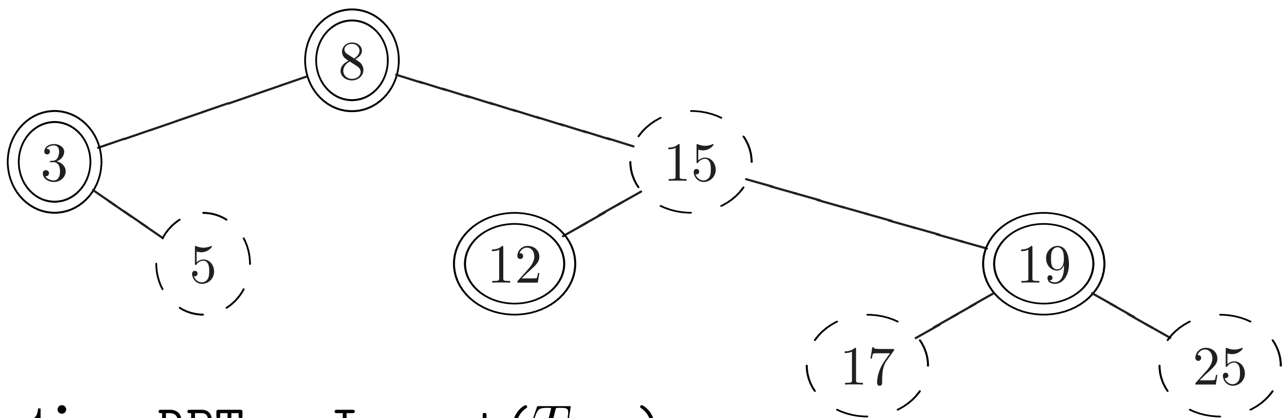
Red-Black Tree: Insert



Red-Black Tree Insert

```
function RBLocateParent(T, z)
/* Return future parent of z in tree */
y ← NIL;
x ← T.root;
while (x is not a leaf) do
    y ← x;
    if (z.key < x.key) then x ← x.left;
    else x ← x.right;
end
return (y);
```

Red-Black Tree Insert



function `RBTreeInsert`(T, z)

$y \leftarrow$ `RBLocateParent`(T, z);

z.parent $\leftarrow y$;

if $(y = \textbf{NIL})$ **then** T.root $\leftarrow z$; /* *tree T was empty* */

else if (z.key $<$ y.key) **then** y.left $\leftarrow z$;

else y.right $\leftarrow z$;

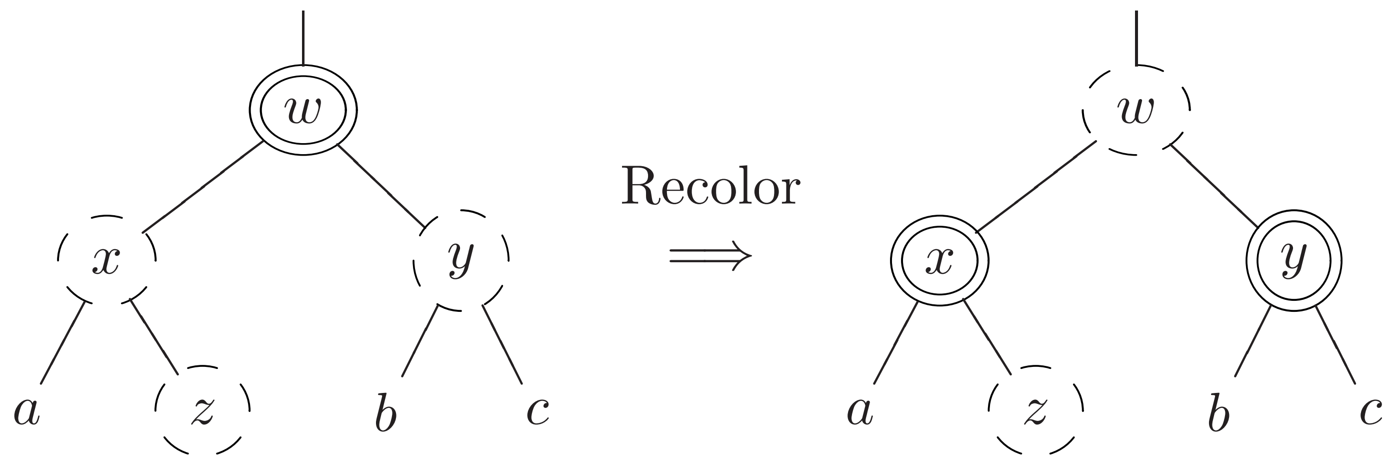
z.left $\leftarrow$ leaf;

z.right $\leftarrow$ leaf;

z.color $\leftarrow$ Red;

`RBInsertFixup`(T, z);

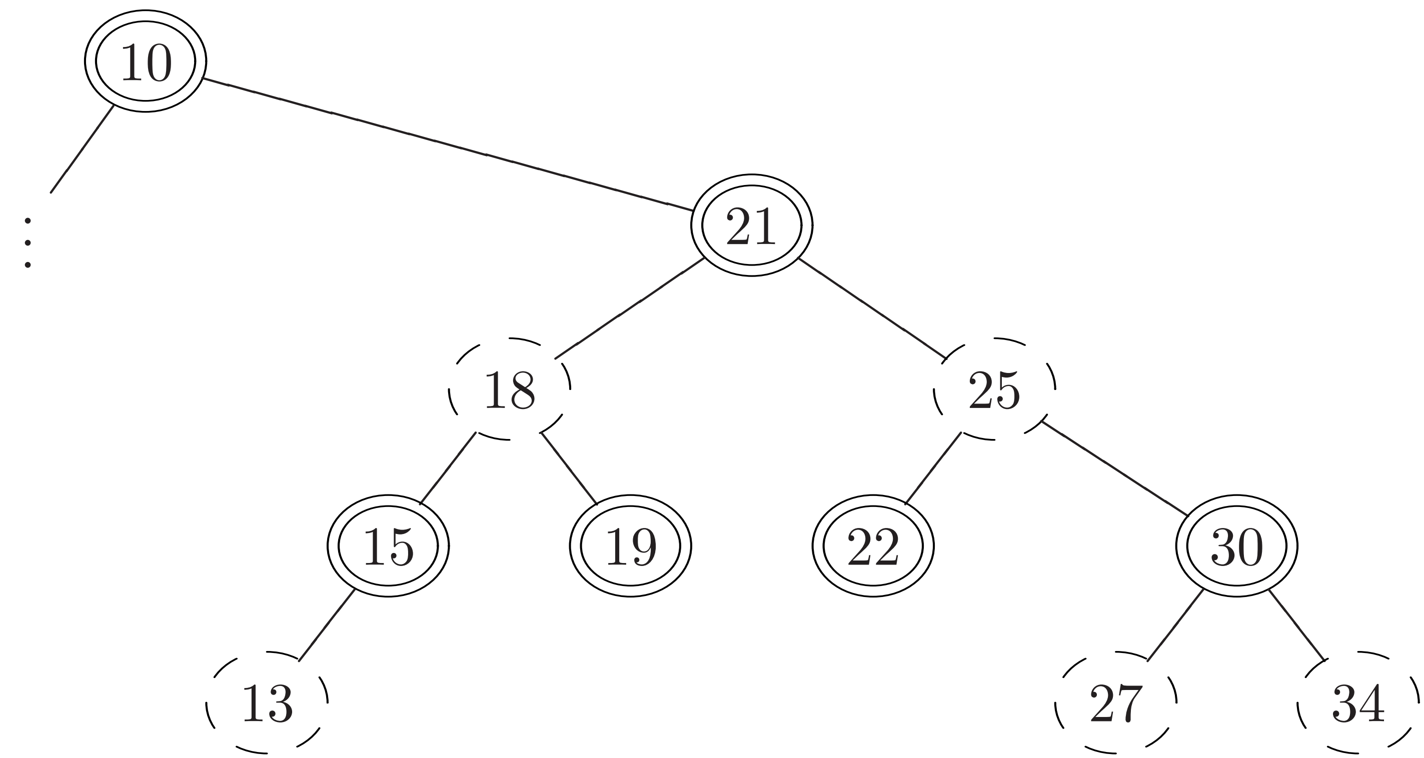
Insert Fixup: Case I



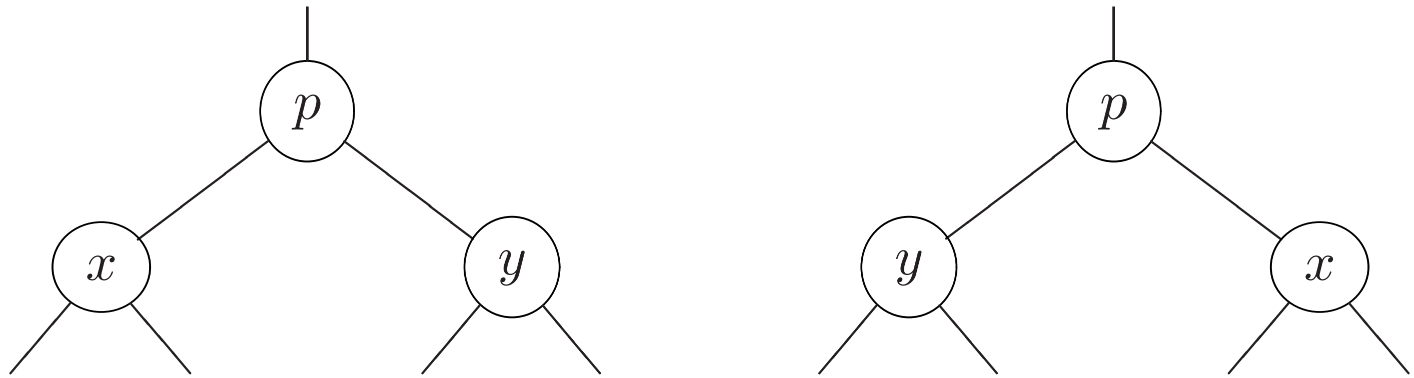
If the parent and "uncle" of z are Red:

- Color the parent of z Black;
- Color the uncle of z Black;
- Color the grandparent of z Red;
- Repeat on the grandparent of z.

Red-Black Tree Insert Fixup: Case I



Sibling



function `Sibling`(x)

/ Return sibling of x */*

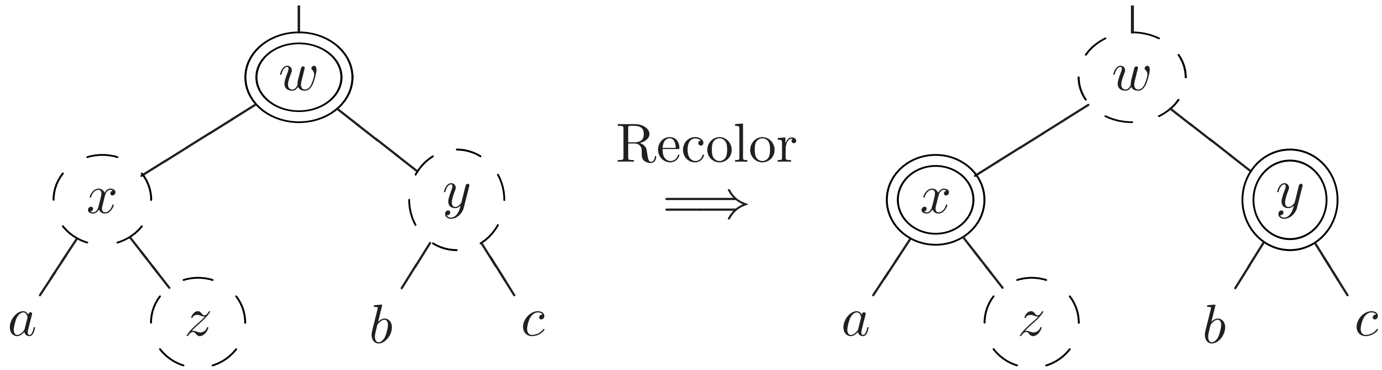
if (x.parent = **NIL**) **then** `error` "Root has no siblings.";

$p \leftarrow x$.parent;

if (p.left $= x$) **then return** (p.right);

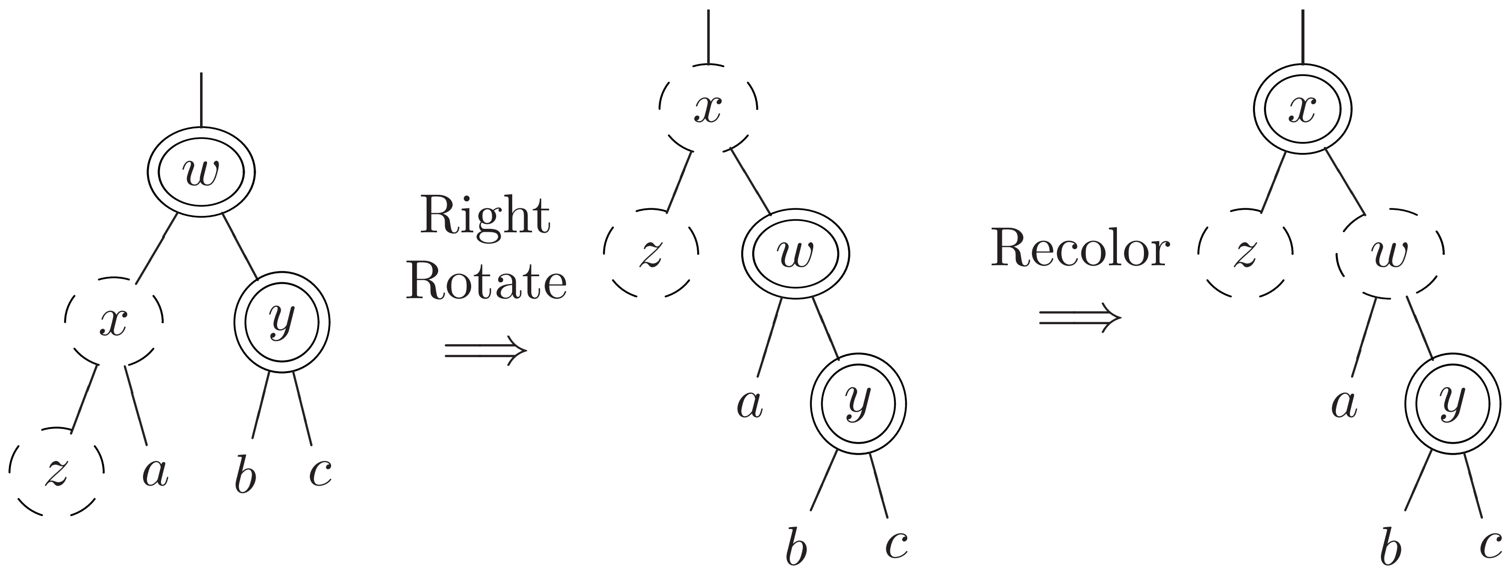
else return (p.left);

Red-Black Tree Insert Fixup: Case I




```
function RBInsertFixupA(T, alters z)
while (z ≠ T.root) and (z.parent.color ≠ Black) do
    y ← Sibling(z.parent);
    if (y.color = Black) then return;
    z.parent.color ← Black;
    y.color ← Black;
    z ← z.parent.parent;
    z.color ← Red;
end
```

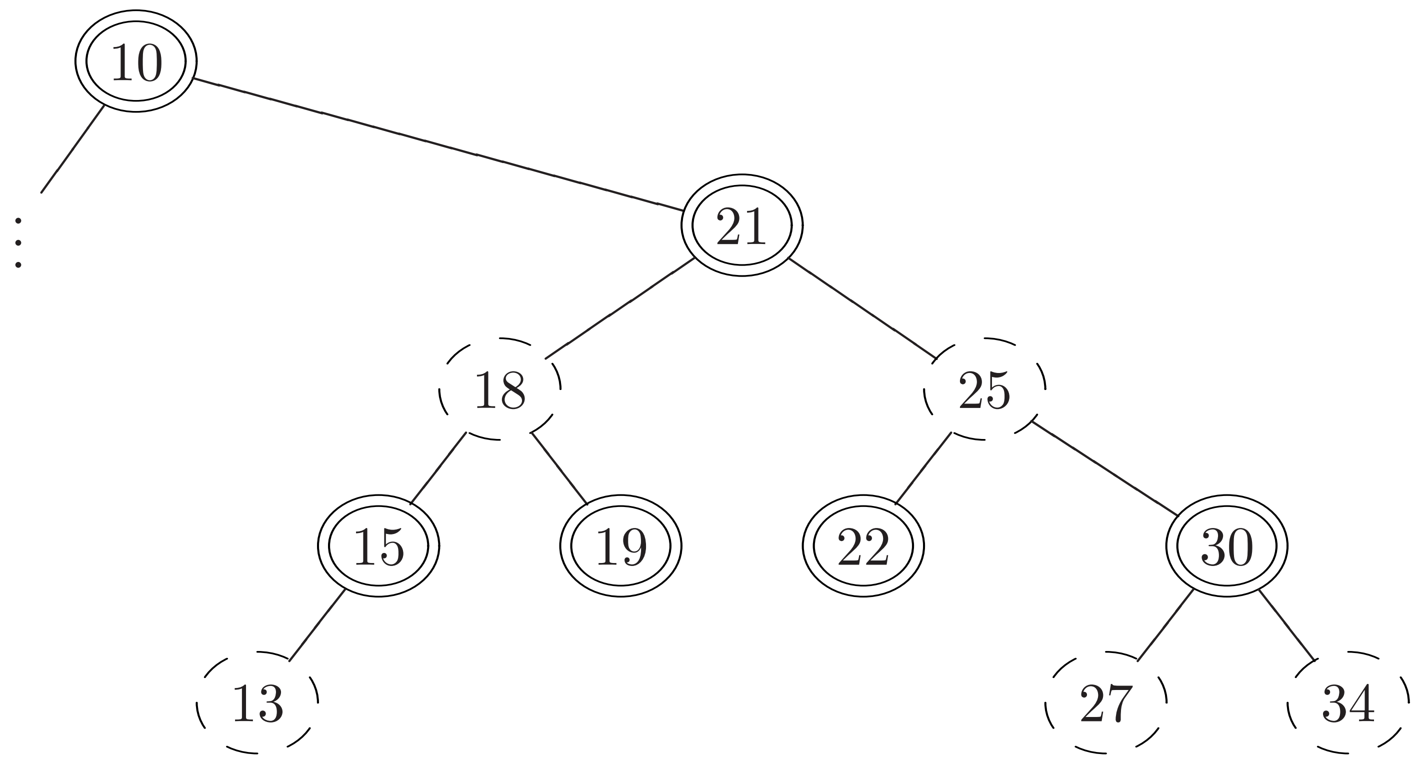
Insert Fixup: Case III



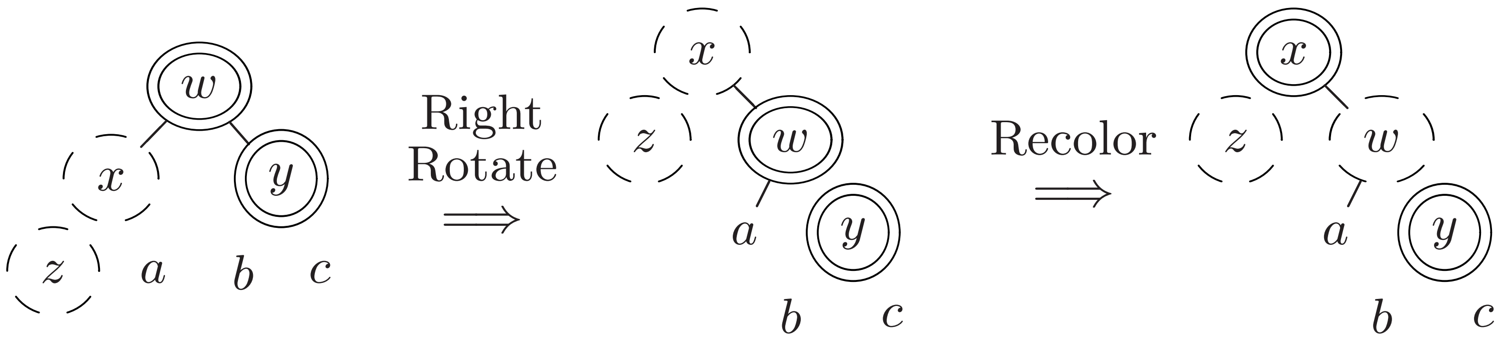
If the parent of z is red and its "uncle" is black:
If z is a left child and its parent is a left child:

- Right Rotate on the grandparent of z;
- Color the parent of z Black;
- Color the sibling of z red.

Red-Black Tree Insert Fixup: Case III



Red-Black Tree Insert Fixup: Case III



function `RBInsertFixupC(`T, **alters** z`)`

if $(z = T.\mathsf{root})$ **or** $(z.\mathsf{parent.color} = \text{Black})$ **then return**;

$x \leftarrow z.\mathsf{parent}$;

$w \leftarrow x.\mathsf{parent}$;

if $(z = x.\mathsf{left})$ **and** $(x = w.\mathsf{left})$ **then**

  `RightRotate(`T,w`);`

  $x.\mathsf{color} \leftarrow \text{Black}$;

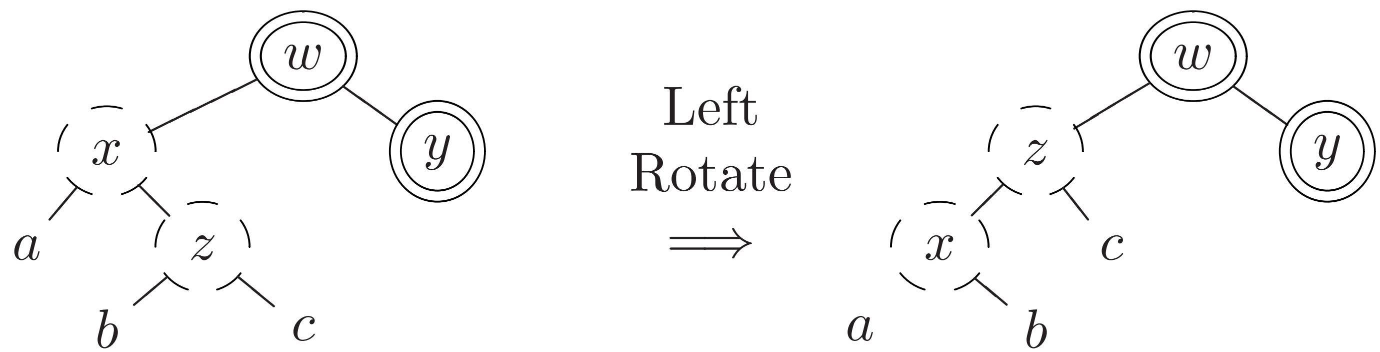
  $w.\mathsf{color} \leftarrow \text{Red}$;

else if $(z = x.\mathsf{right})$ **and** $(x = w.\mathsf{right})$ **then**

  Handle same as above with “right” and “left” exchanged

...

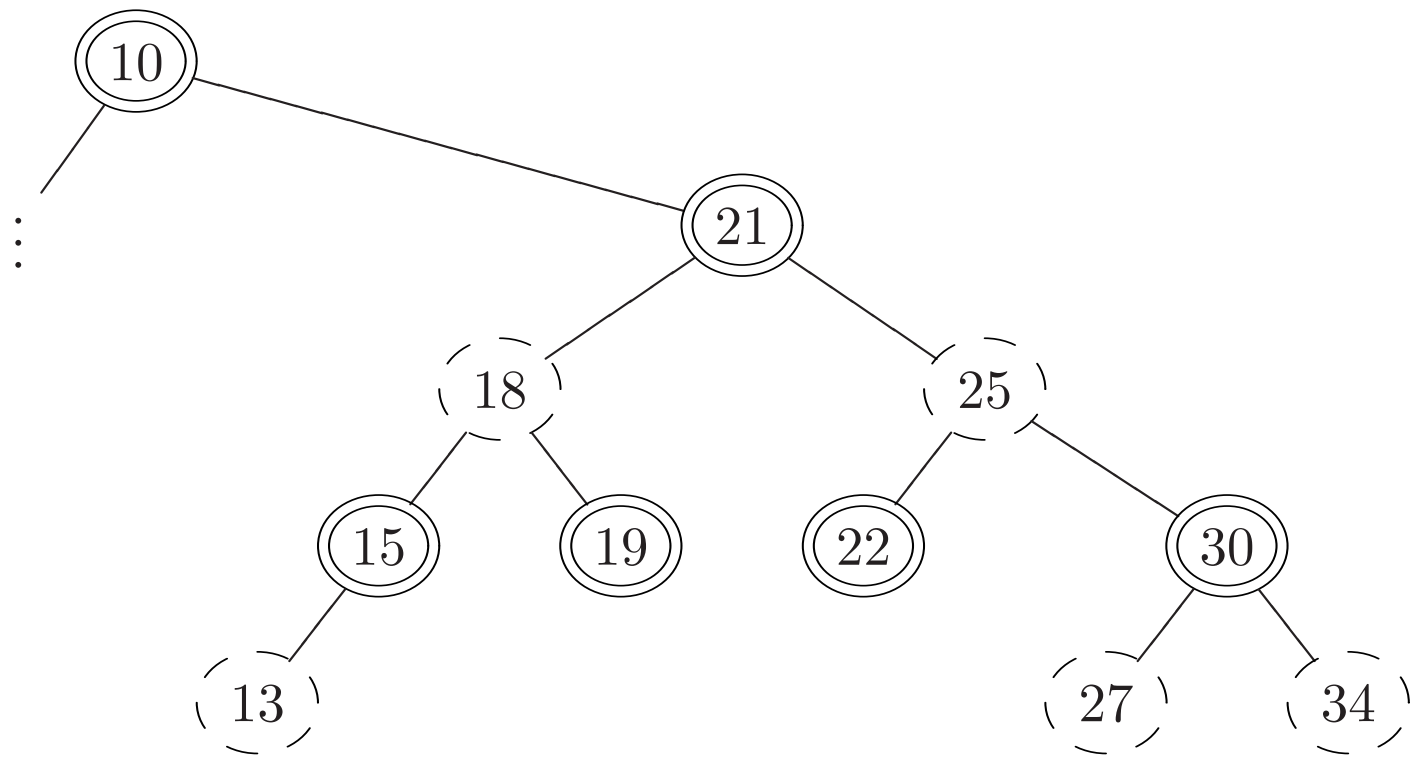
Insert Fixup: Case II



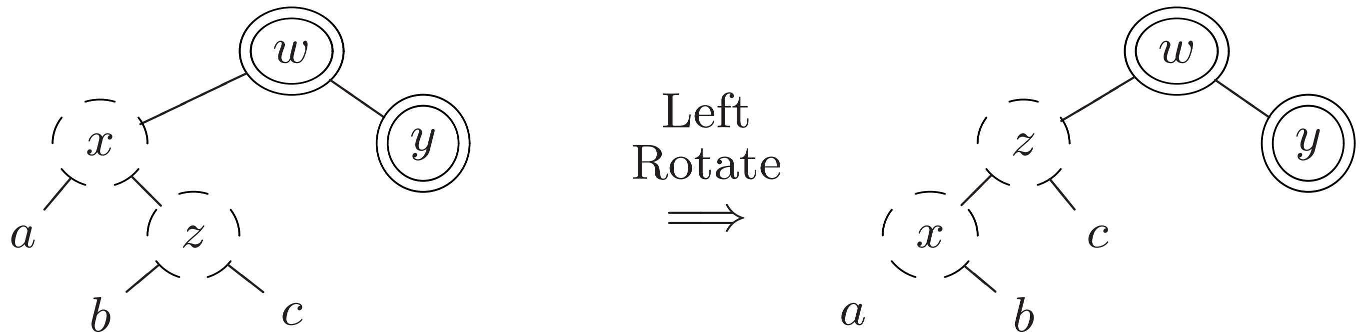
If the parent x of z is red and its "uncle" is black:
If z is a right child and its parent x is a left child:

- $z \leftarrow x$
- Left Rotate on x;
- Apply algorithm for Case III.

Red-Black Tree Insert Fixup: Case II



Red-Black Tree Insert Fixup: Case II



function `RBInsertFixupB`(T, **alters** z)

if ($z = T$.root) **or** (z.parent.color = Black) **then return**;

$x \leftarrow z$.parent;

$w \leftarrow x$.parent;

if ($z = x$.right) **and** ($x = w$.left) **then**

    $z \leftarrow x$;

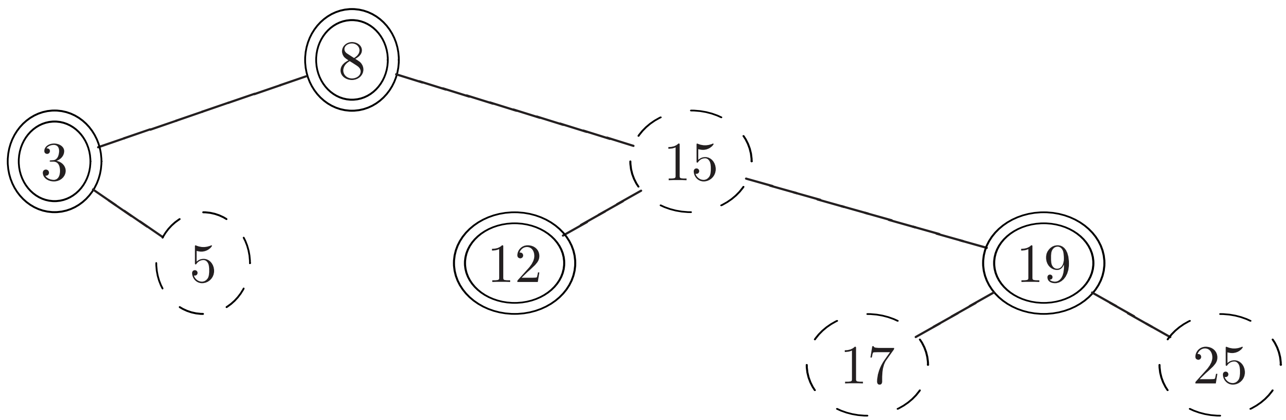
    `LeftRotate`(T,x);

else if ($z = x$.left) **and** ($x = w$.right) **then**

    Handle same as above with "right" and "left" exchanged

. . .

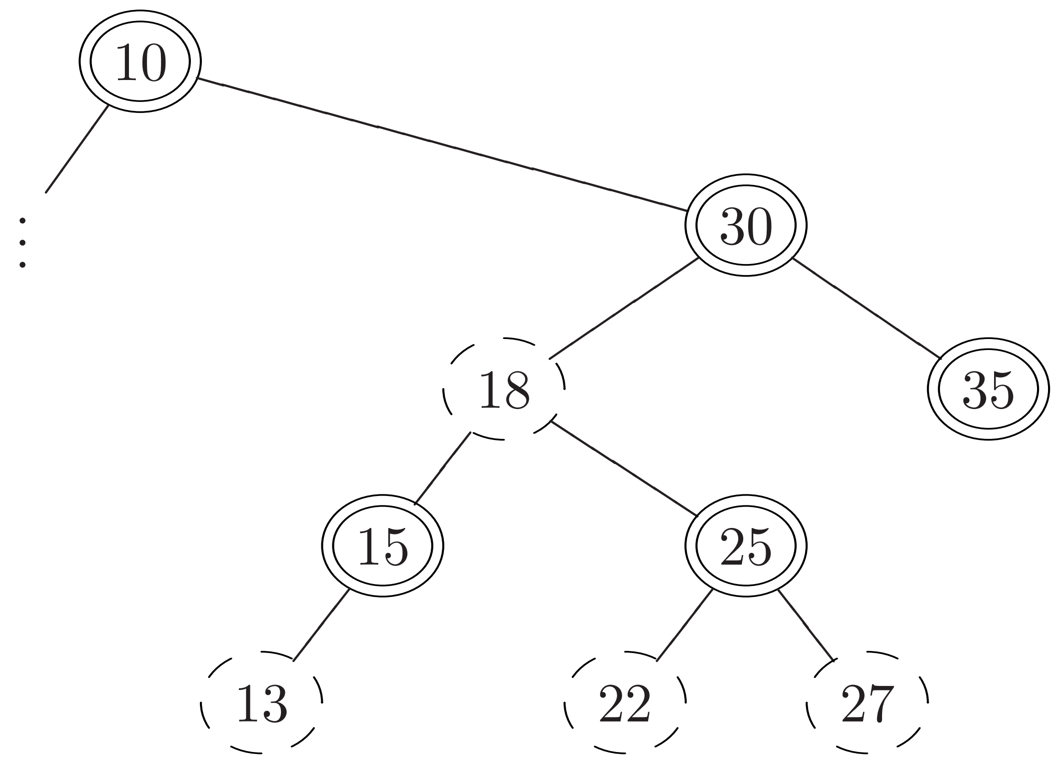
Red-Black Tree Insert Fixup



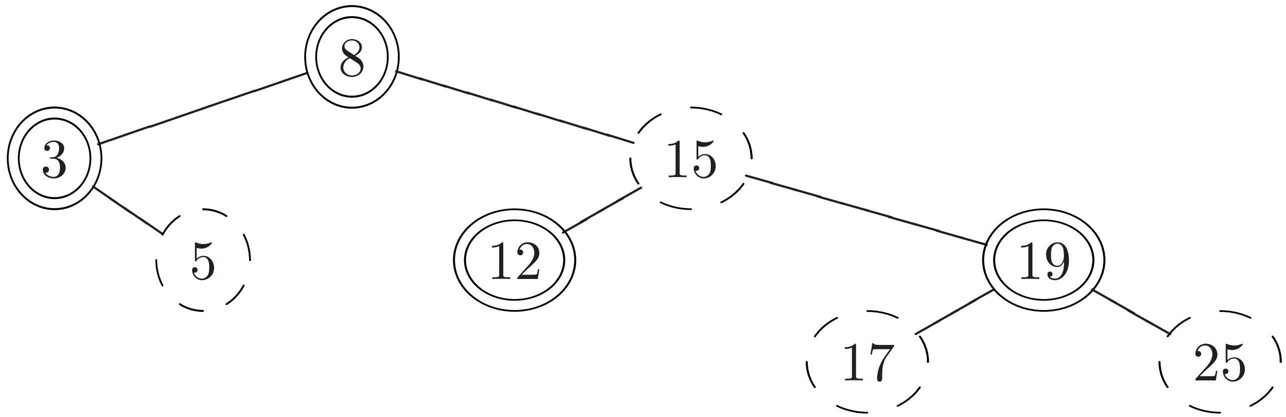
function `RBInsertFixup`(T, z)

```
RBInsertFixupA (T,z);
RBInsertFixupB (T,z);
RBInsertFixupC (T,z);
T.root.color ← Black;
```

Red-Black Tree Insert Fixup



Running Time Analysis: `RBInsertFixup`



function `RBInsertFixup`(T, z)

1 `RBInsertFixupA` (T,z);

2 `RBInsertFixupB` (T,z);

3 `RBInsertFixupC` (T,z);

4 T.root.color $\leftarrow$ Black;