



UMC 201: Data Structures & Algorithms

Lecture 10: Red-Black trees

Indian Institute of Science

Recap

- In the last lecture, we discussed binary search trees (BST) and showed how the dynamic-set operations **Search**, **Minimum**, **Maximum**, **Successor**, **Predecessor**, **Insert**, and **Delete** can be implemented so that each one runs in $O(h)$ time on a binary tree of height h .

Recap

- In the last lecture, we discussed binary search trees (BST) and showed how the dynamic-set operations **Search**, **Minimum**, **Maximum**, **Successor**, **Predecessor**, **Insert**, and **Delete** can be implemented so that each one runs in $O(h)$ time on a binary tree of height h .
- **Question.** Can we ensure that the height of a BST is low?

Recap

- In the last lecture, we discussed binary search trees (BST) and showed how the dynamic-set operations **Search**, **Minimum**, **Maximum**, **Successor**, **Predecessor**, **Insert**, and **Delete** can be implemented so that each one runs in $O(h)$ time on a binary tree of height h .
- **Question**. Can we ensure that the height of a BST is low?
- Today, we'll discuss a BST scheme, namely **red-black trees**, which maintain height $O(\log n)$ for trees with n keys, thereby ensuring that each of the above operations takes $O(\log n)$ time in the worst case.

RB trees vs. other data structures

	Insert	Delete	Search	Min/Max
Unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted array	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
Doubly Linked list	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Heaps	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(1)$
Hash tables	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$ (expected)	$\Theta(n)$
Red-Black trees	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

RB trees vs. other data structures

	Insert	Delete	Search	Min/Max
Unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted array	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
Doubly Linked list	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Heaps	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(1)$
Hash tables	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$ (expected)	$\Theta(n)$
Red-Black trees	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

- RB trees (1978) are widely used in practice to build sorted data structures, memory managers, and OS schedulers.

RB trees vs. other data structures

	Insert	Delete	Search	Min/Max
Unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted array	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
Doubly Linked list	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Heaps	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(1)$
Hash tables	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$ (expected)	$\Theta(n)$
Red-Black trees	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

- The Linux kernel uses a RB tree in its Scheduler to pick the next task for the CPU. It also uses RB trees to manage virtual memory.

Red-Black trees

- A **red-black tree** is a BST with one extra bit of storage per node: its **colour**, which is either **red** or **black**.
- By constraining the node colours on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately **balanced**.

Red-Black trees

- A **red-black tree** is a BST with one extra bit of storage per node: its **colour**, which is either **red** or **black**.
- By constraining the node colours on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately **balanced**.
- Indeed, we'll prove that the height of a red-black (RB) tree with n keys (nodes) is at most $2 \log(n + 1) = \Theta(\log n)$.

Red-Black trees

- Each node of a RB tree contains the attributes: *colour*, *key*, *left*, *right*, and *p*.

Red-Black trees

- Each node of a RB tree contains the attributes: *colour*, *key*, *left*, *right*, and *p*.
- If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the values NIL.
- We'll think of these NILs as pointers to leaves (external nodes) of the RB tree and the normal, key-bearing nodes as internal nodes of the tree.

Red-Black trees: Definition

- A **red-black tree** is a BST that satisfies the following five red-black properties:
 1. Every node is either **red** or **black**.
 2. The root is **black**.
 3. Every leaf (NIL) is **black**.

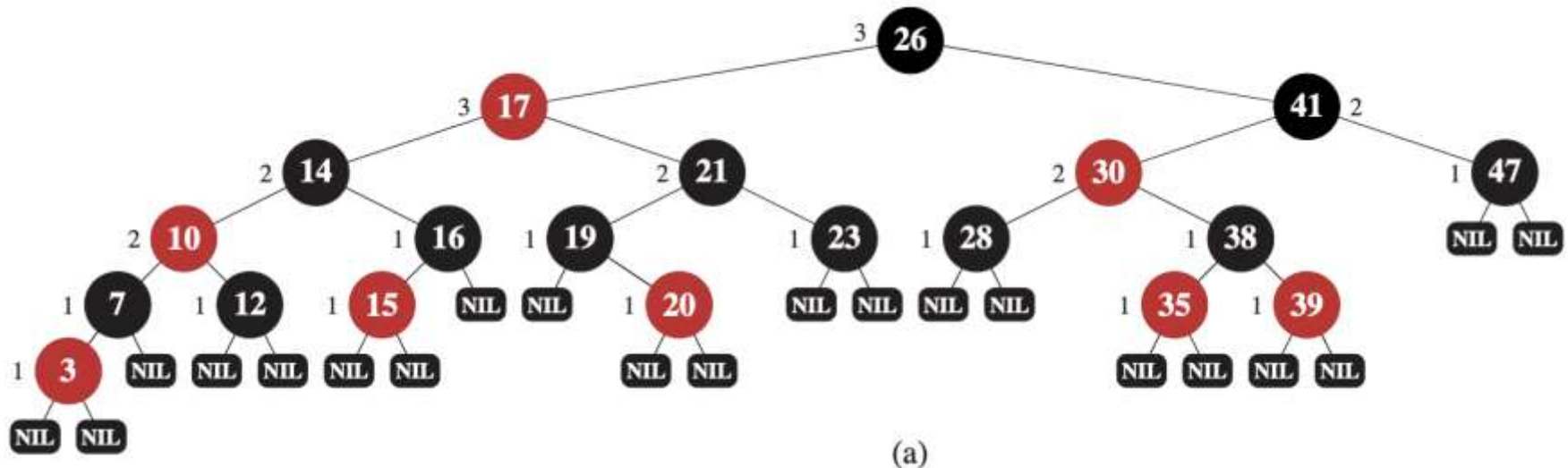
Red-Black trees: Definition

- A **red-black tree** is a BST that satisfies the following five red-black properties:
 1. Every node is either **red** or **black**.
 2. The root is **black**.
 3. Every leaf (NIL) is **black**.
 4. If a node is **red**, then both its children are **black**.

Red-Black trees: Definition

- A **red-black tree** is a BST that satisfies the following five red-black properties:
 1. Every node is either **red** or **black**.
 2. The root is **black**.
 3. Every leaf (NIL) is **black**.
 4. If a node is **red**, then both its children are **black**.
 5. For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.

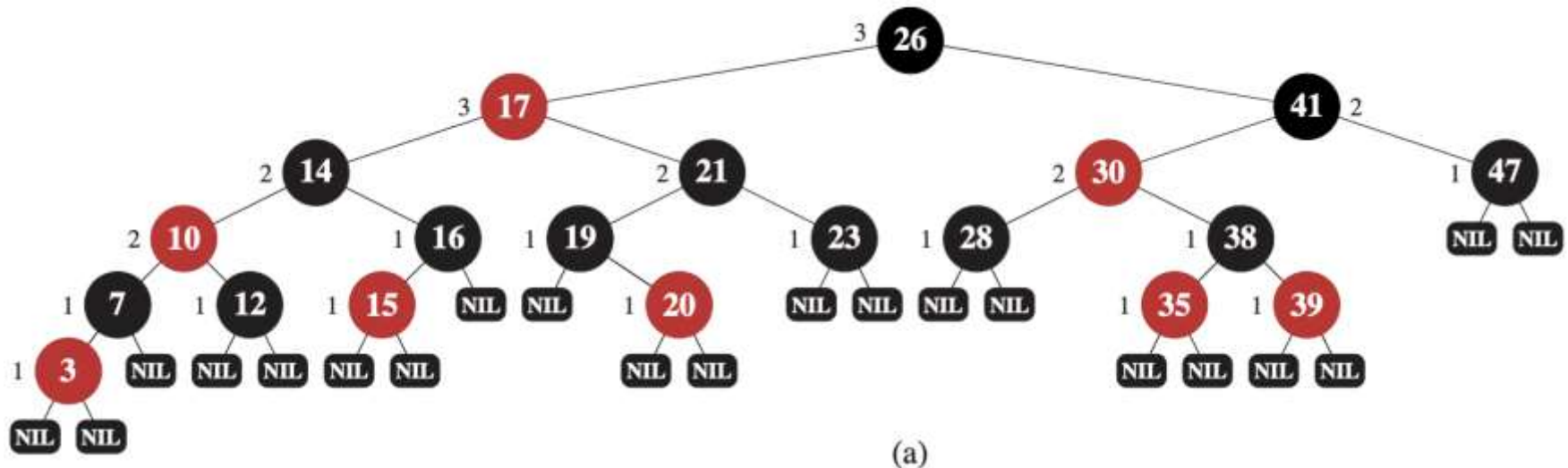
Red-Black trees: Example



- We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the **black-height** of the node, denoted $bh(x)$.

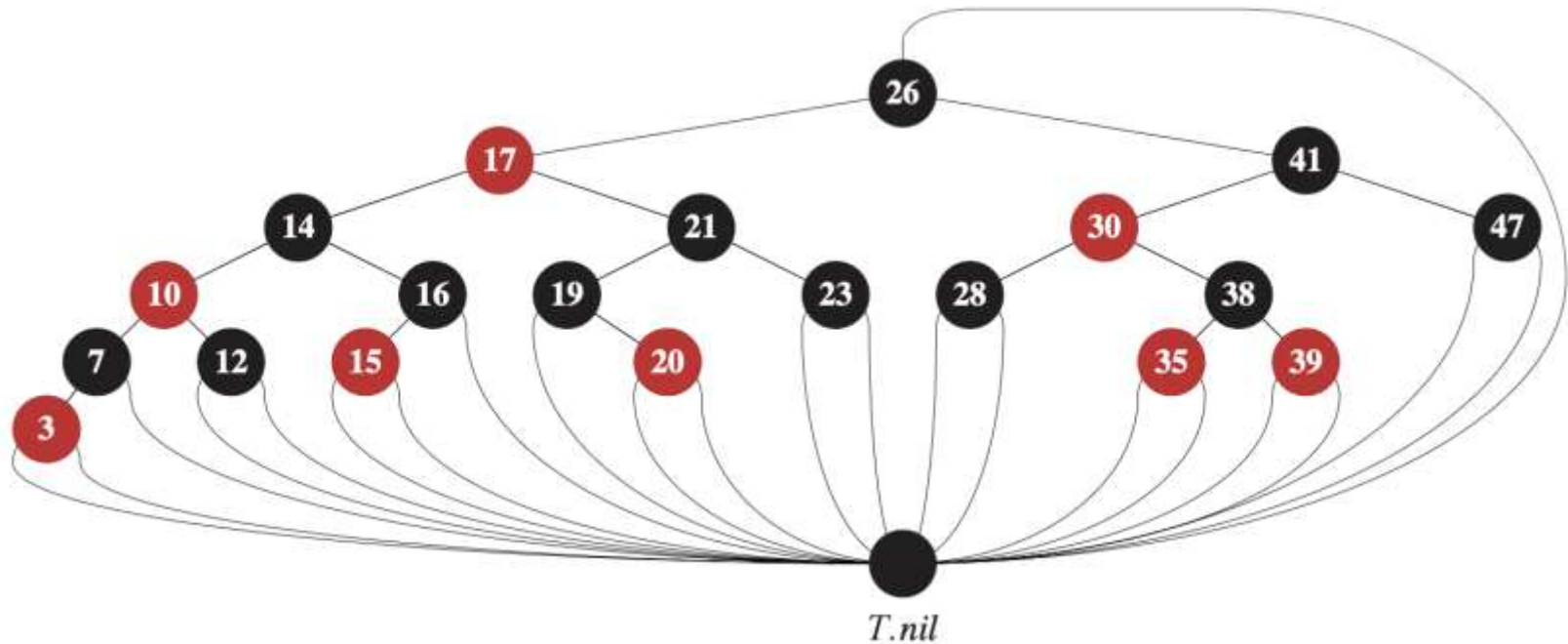
Owing to RB Property 5, black-height is well-defined.

Red-Black trees: Example



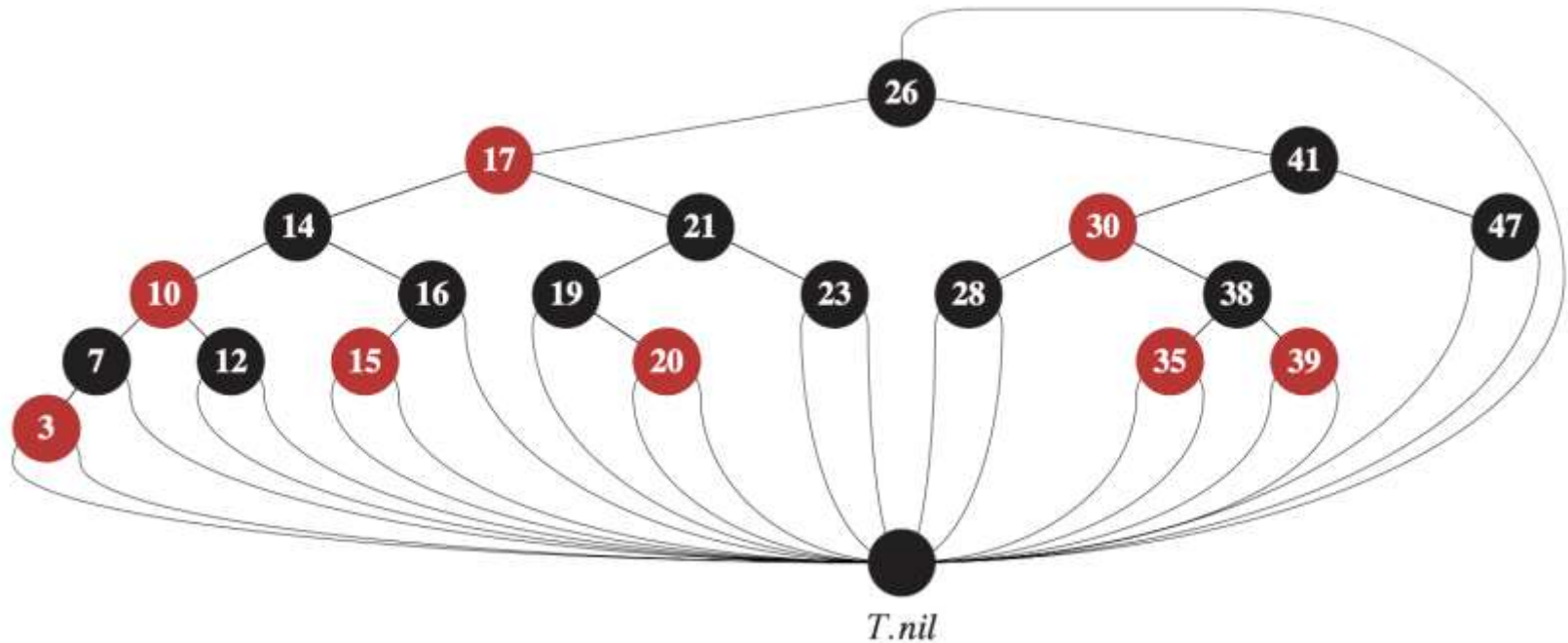
- We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the **black-height** of the node, denoted $bh(x)$. In the above figure, each non-NIL node is marked with its black-height, where NILs have black-height 0.

Red-Black trees: Example



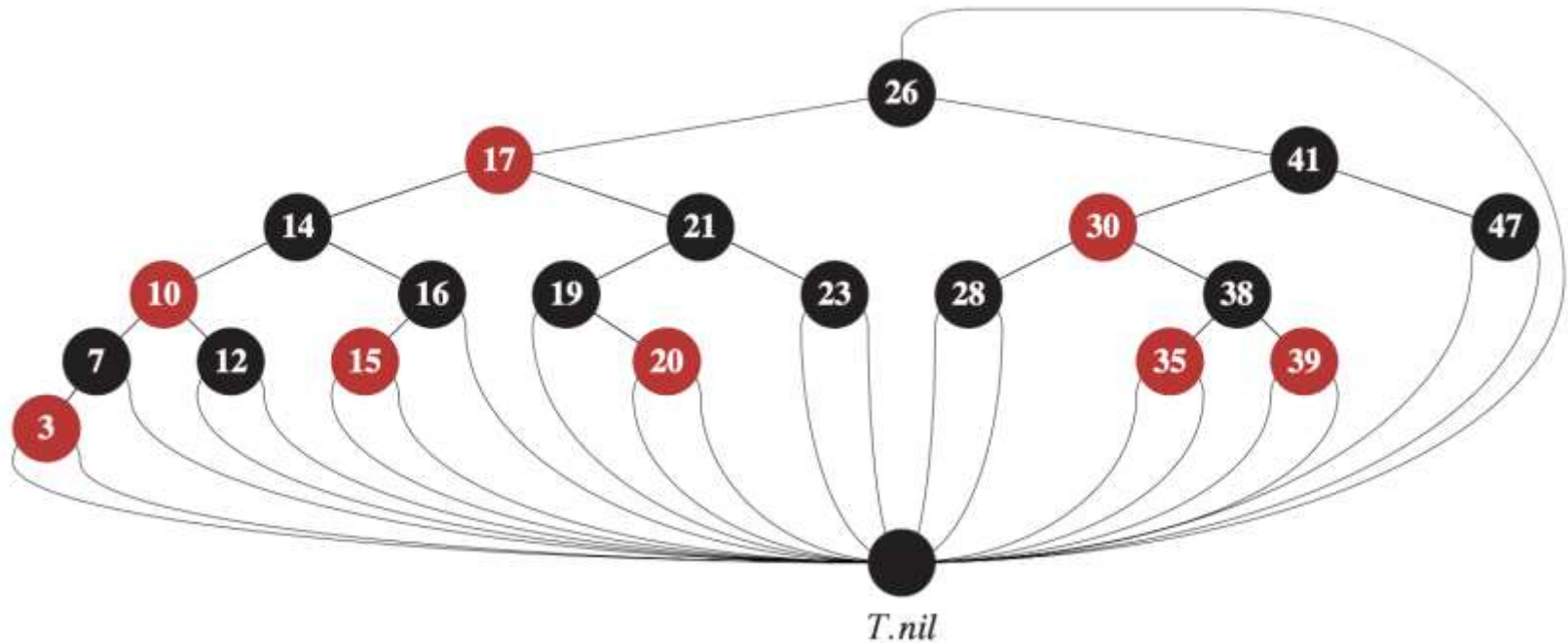
- For convenience in dealing with boundary conditions in RB tree codes, we use a single sentinel to represent NIL.

Red-Black trees: Example



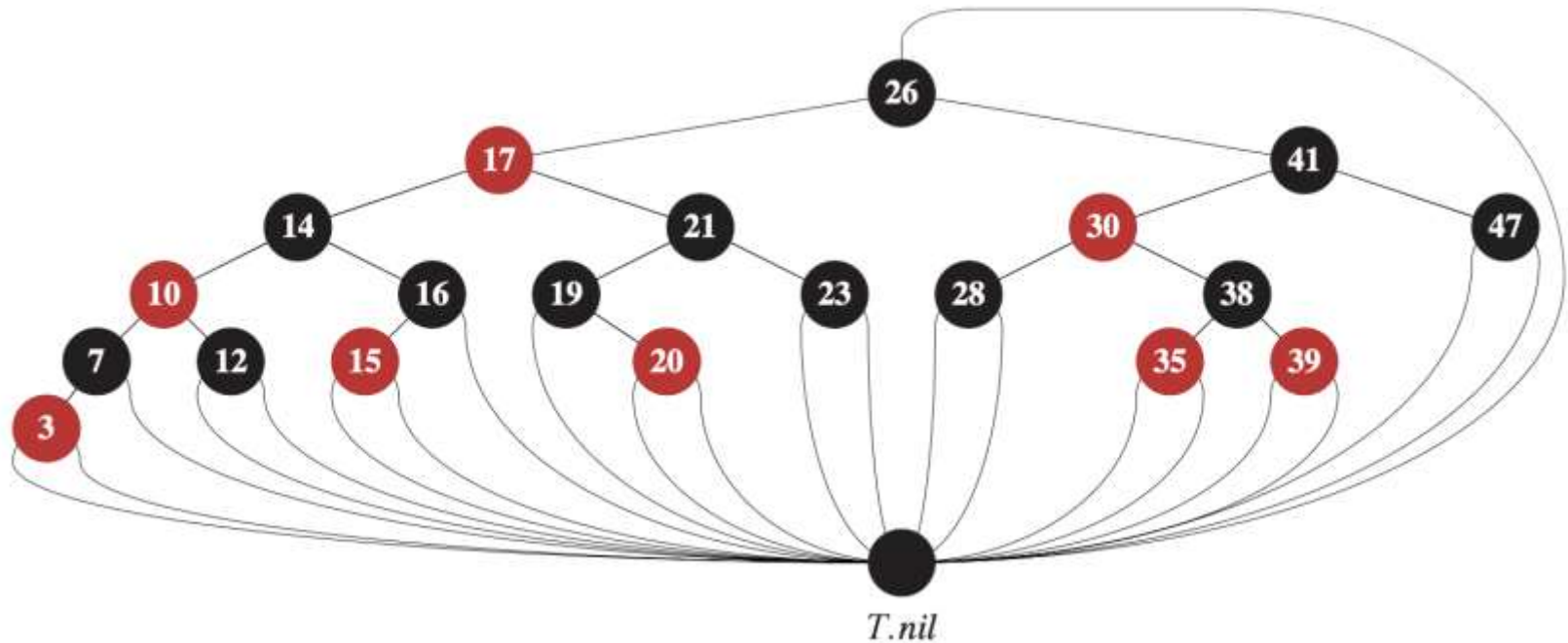
- For convenience in dealing with boundary conditions in RB tree codes, we use a single sentinel to represent NIL. In the above figure, all pointers to NIL are replaced by pointers to the sentinel *T.nil*.

Red-Black trees: Example



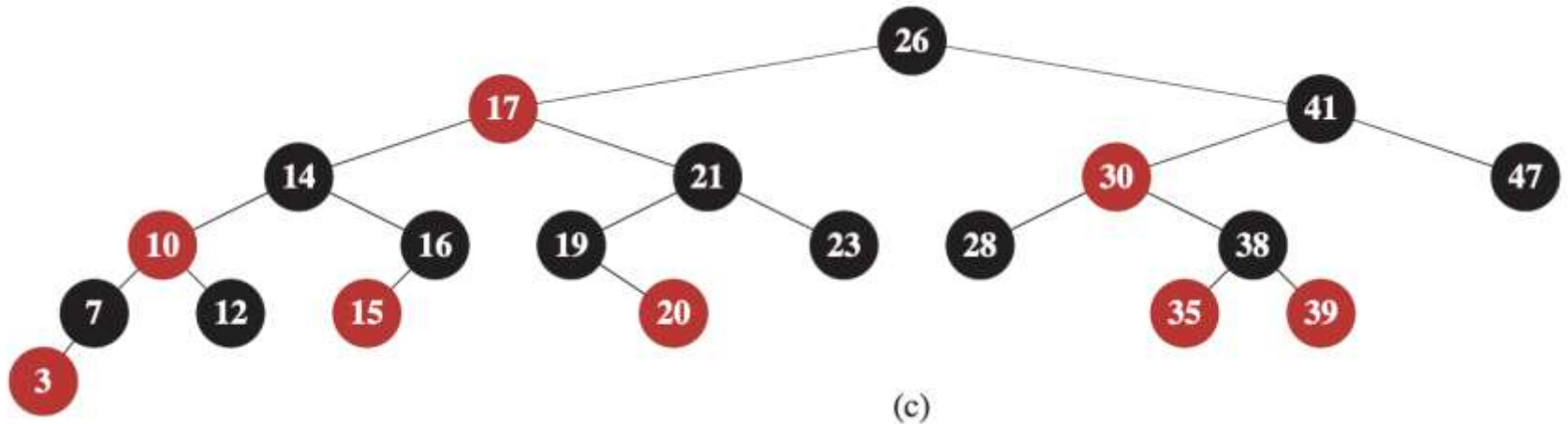
- For a RB tree T , the sentinel $T.nil$ is a node with the same attributes as an ordinary node in T . Its *colour* is **black**, and the other attributes – p , *left*, *right*, and *key* – can take on arbitrary values to yield simpler code (otherwise, the other attributes are immaterial).

Red-Black trees: Example



- We use the sentinel in order to save space. Just the one sentinel *T.nil* represents all the NILs – all leaves and the root's parent.

Red-Black trees: Example



- We generally confine our interest to the internal nodes of a RB tree, since they hold the key values. Henceforth, we'll omit the leaves in drawings of RB trees (as in the above figure).

Height of a Red-Black tree

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

Height of a Red-Black tree

- **Lemma.** A red-black tree with n internal nodes has height at most $2 \log(n + 1)$.
- We'll prove the lemma by proving the following claim.
- **Claim.** The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

Height of a Red-Black tree

- **Lemma.** A red-black tree with n internal nodes has height at most $2 \log(n + 1)$.
- We'll prove the lemma by proving the following claim.
- **Claim.** The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
- Applying the claim on the root node, we get $n \geq 2^{bh(root)} - 1$, implying, $bh(root) \leq \log(n + 1)$.

Height of a Red-Black tree

- **Lemma.** A red-black tree with n internal nodes has height at most $2 \log(n + 1)$.
- We'll prove the lemma by proving the following claim.
- **Claim.** The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
- Applying the claim on the root node, we get $n \geq 2^{bh(root)} - 1$, implying, $bh(root) \leq \log(n + 1)$.
- As both the children of a **red** node are **black**, height of the root (which is the height of the tree) is at most $2 bh(root) \leq 2 \log(n + 1)$.

Height of a Red-Black tree

- **Lemma.** A red-black tree with n internal nodes has height at most $2 \log(n + 1)$.
- We'll prove the lemma by proving the following claim.
- **Claim.** The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
- **Proof.** We prove by induction on the height of x . If the height of x is 0, then x is a leaf ($T.nil$), and hence the subtree rooted at x contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

Height of a Red-Black tree

- **Lemma.** A red-black tree with n internal nodes has height at most $2 \log(n + 1)$.
- We'll prove the lemma by proving the following claim.
- **Claim.** The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
- **Proof.** Now consider a node x that has height > 0 . Each child of x has black-height of either $bh(x) - 1$ (if it's **black**) or $bh(x)$ (if it's **red**).

Height of a Red-Black tree

- **Lemma.** A red-black tree with n internal nodes has height at most $2 \log(n + 1)$.
- We'll prove the lemma by proving the following claim.
- **Claim.** The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
- **Proof.** Now consider a node x that has height > 0 . Each child of x has black-height of either $bh(x) - 1$ (if it's **black**) or $bh(x)$ (if it's **red**). Since the height of a child is $<$ the height of x , by induction, the subtree rooted at each child has $\geq 2^{bh(x)-1} - 1$ nodes.

Height of a Red-Black tree

- **Lemma.** A red-black tree with n internal nodes has height at most $2 \log(n + 1)$.
- We'll prove the lemma by proving the following claim.
- **Claim.** The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
- **Proof.** Now consider a node x that has height > 0 . Each child of x has black-height of either $bh(x) - 1$ (if it's **black**) or $bh(x)$ (if it's **red**). So, the subtree rooted x has $\geq 2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nodes. (The $+1$ is due to x itself.)



Efficiency of the operations

- As an immediate consequence of the lemma, each of the operations **Search**, **Minimum**, **Maximum**, **Successor**, and **Predecessor** runs in $O(\log n)$ time on a RB tree.

Efficiency of the operations

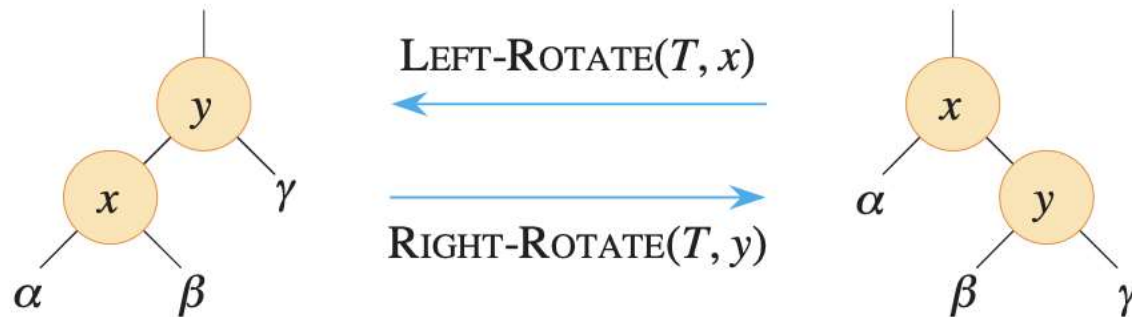
- As an immediate consequence of the lemma, each of the operations **Search**, **Minimum**, **Maximum**, **Successor**, and **Predecessor** runs in $O(\log n)$ time on a RB tree.
- Although the BST insert and delete procedures discussed in the last lecture run in $O(\log n)$ time when given an RB tree as an input, we cannot just use them to implement the operations **Insert** and **Delete**. Those procedures do not necessarily maintain the red-black properties.

Efficiency of the operations

- As an immediate consequence of the lemma, each of the operations **Search**, **Minimum**, **Maximum**, **Successor**, and **Predecessor** runs in $O(\log n)$ time on a RB tree.
- Although the BST insert and delete procedures discussed in the last lecture run in $O(\log n)$ time when given an RB tree as an input, we cannot just use them to implement the operations **Insert** and **Delete**. Those procedures do not necessarily maintain the red-black properties. To restore these properties, pointers and colours within nodes need to change.

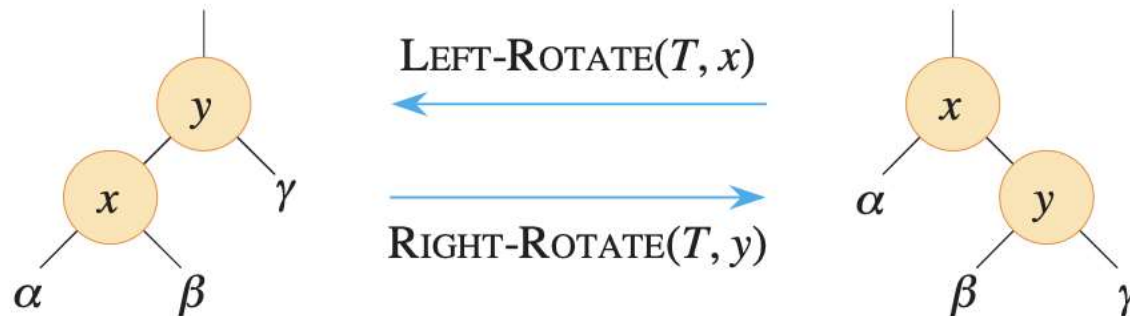
Rotations

- The pointer structure changes through **rotation**, which is a local operation in a BST that preserves the binary-search-tree property. There are two kinds of rotations: **left rotation** and **right rotation**.



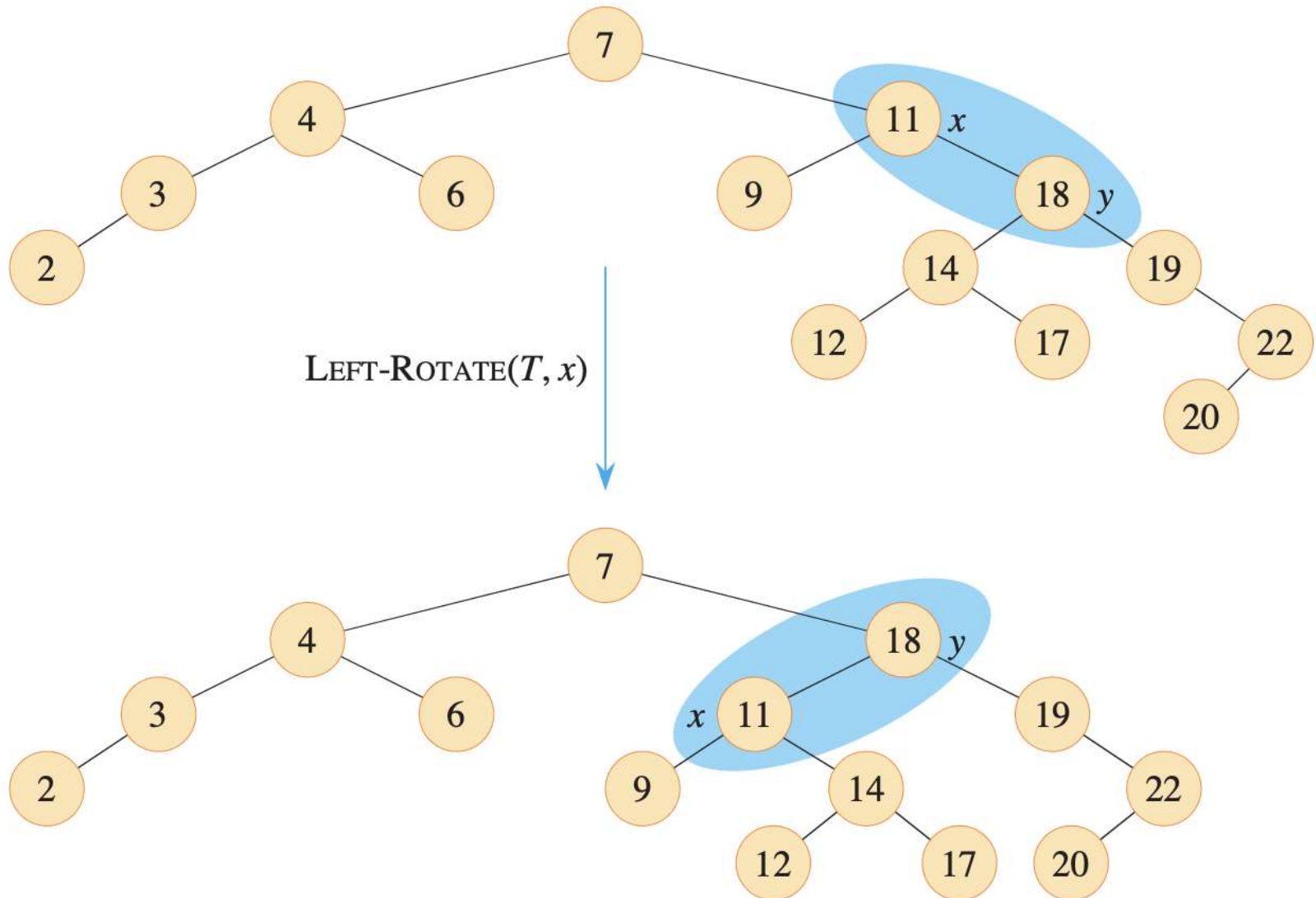
Rotations

- The pointer structure changes through **rotation**, which is a local operation in a BST that preserves the binary-search-tree property. There are two kinds of rotations: **left rotation** and **right rotation**.



- For left rotation, assume that $x.\text{right}$ is not $T.\text{nil}$. Similarly, for right rotation, assume $y.\text{left}$ is not $T.\text{nil}$.

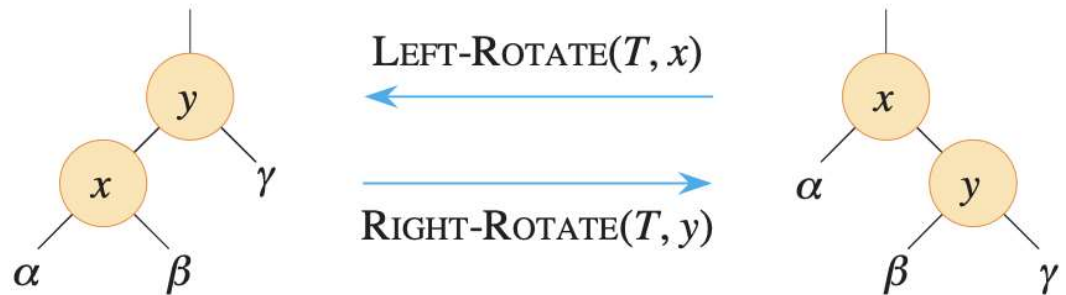
Left rotation: Example



Left rotation: Pseudocode

LEFT-ROTATE(T, x)

```
1   $y = x.right$ 
2   $x.right = y.left$ 
3  if  $y.left \neq T.nil$ 
4       $y.left.p = x$ 
5   $y.p = x.p$ 
6  if  $x.p == T.nil$ 
7       $T.root = y$ 
8  elseif  $x == x.p.left$ 
9       $x.p.left = y$ 
10 else  $x.p.right = y$ 
11  $y.left = x$ 
12  $x.p = y$ 
```

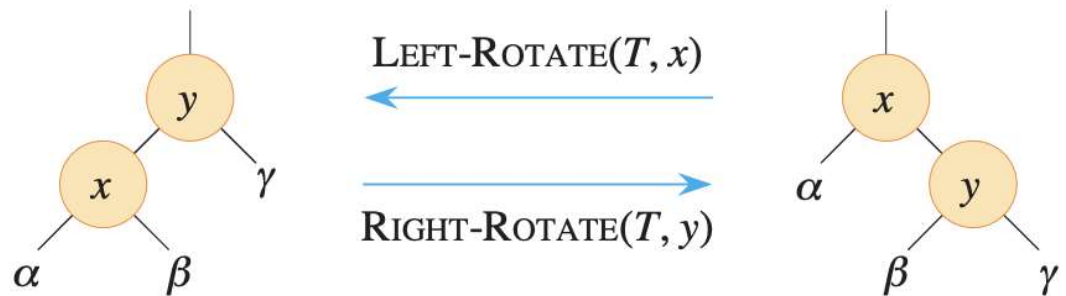


- Only pointers are changed by a rotation, all other attributes in a node remain the same.

Left rotation: Pseudocode

LEFT-ROTATE(T, x)

```
1   $y = x.right$ 
2   $x.right = y.left$ 
3  if  $y.left \neq T.nil$ 
4       $y.left.p = x$ 
5   $y.p = x.p$ 
6  if  $x.p == T.nil$ 
7       $T.root = y$ 
8  elseif  $x == x.p.left$ 
9       $x.p.left = y$ 
10 else  $x.p.right = y$ 
11  $y.left = x$ 
12  $x.p = y$ 
```



- Only pointers are changed by a rotation, all other attributes in a node remain the same.
- The code of **RIGHT-ROTATE** is symmetric.
- Both **LEFT-ROTATE** and **RIGHT-ROTATE** run in $O(1)$ time.

Red-Black tree: Insertion

Insertion: The Idea

- In order to insert a new node into a RB tree with n internal nodes in $O(\log n)$ time, we'll modify the **TREE-INSERT** procedure for binary search trees provided in the previous lecture.
- The new **RB-INSERT** procedure starts by inserting the new node z into the RB tree T as if it were an ordinary BST.

Insertion: The Idea

- In order to insert a new node into a RB tree with n internal nodes in $O(\log n)$ time, we'll modify the **TREE-INSERT** procedure for binary search trees provided in the previous lecture.
- The new **RB-INSERT** procedure starts by inserting the new node z into the RB tree T as if it were an ordinary BST.
- It then colours z **red**. Finally, to guarantee that the red-black properties are preserved, an auxiliary procedure **RB-INSERT-FIXUP** is called.

Insertion: Pseudocode

RB-INSERT(T, z)

```
1   $x = T.root$ 
2   $y = T.nil$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
14  $z.left = T.nil$ 
15  $z.right = T.nil$ 
16  $z.color = RED$ 
17 RB-INSERT-FIXUP( $T, z$ )
```



Steps 1-8 find a suitable parent for z by traversing the tree just as in the **TREE-INSERT** procedure for BSTs.

Insertion: Pseudocode

RB-INSERT(T, z)

```
1   $x = T.root$ 
2   $y = T.nil$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11  elseif  $z.key < y.key$ 
12      $y.left = z$ 
13  else  $y.right = z$ 
14   $z.left = T.nil$ 
15   $z.right = T.nil$ 
16   $z.color = RED$ 
17  RB-INSERT-FIXUP( $T, z$ )
```

Steps 9-13 decide if z should be the left/right child of its parent just as in the **TREE-INSERT** procedure for BSTs.

Insertion: Pseudocode

RB-INSERT(T, z)

```
1   $x = T.root$ 
2   $y = T.nil$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
14  $z.left = T.nil$ 
15  $z.right = T.nil$ 
16  $z.color = RED$ 
17 RB-INSERT-FIXUP( $T, z$ )
```

Steps 14-15 set $T.nil$ as the children of the newly inserted node z .

Insertion: Pseudocode

RB-INSERT(T, z)

```
1   $x = T.root$ 
2   $y = T.nil$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
14  $z.left = T.nil$ 
15  $z.right = T.nil$ 
16  $z.color = RED$ 
17 RB-INSERT-FIXUP( $T, z$ )
```

Step 16 sets the colour of z **red**. This is a “reasonable” as $T.nil$ is **black**.

Insertion: Pseudocode

RB-INSERT(T, z)

```
1   $x = T.root$ 
2   $y = T.nil$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
14  $z.left = T.nil$ 
15  $z.right = T.nil$ 
16  $z.color = RED$ 
17 RB-INSERT-FIXUP( $T, z$ )
```

Step 17 restores the red-black properties.

Insertion: Pseudocode

RB-INSERT(T, z)

```
1   $x = T.root$ 
2   $y = T.nil$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
14  $z.left = T.nil$ 
15  $z.right = T.nil$ 
16  $z.color = RED$ 
17 RB-INSERT-FIXUP( $T, z$ )
```

Step 17 restores the red-black properties.

But how? We shall see...

Insertion: Pseudocode

RB-INSERT(T, z)

```
1   $x = T.root$ 
2   $y = T.nil$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11  elseif  $z.key < y.key$ 
12      $y.left = z$ 
13  else  $y.right = z$ 
14   $z.left = T.nil$ 
15   $z.right = T.nil$ 
16   $z.color = RED$ 
17  RB-INSERT-FIXUP( $T, z$ )
```

Obs. If **RB-INSERT-FIXUP** runs in $O(\log n)$ time, then so does **RB-INSERT**.

Insertion: Violation of RB properties

- **Question.** What violations of the red-black properties might occur upon the call to **RB-INSERT-FIXUP** ?

Insertion: Violation of RB properties

- **Question.** What violations of the red-black properties might occur upon the call to **RB-INSERT-FIXUP** ?
- Property 1 (i.e., every node is either **red** or **black**) continues to hold.
- Property 3 (i.e., every leaf is **black**) also hold.

Insertion: Violation of RB properties

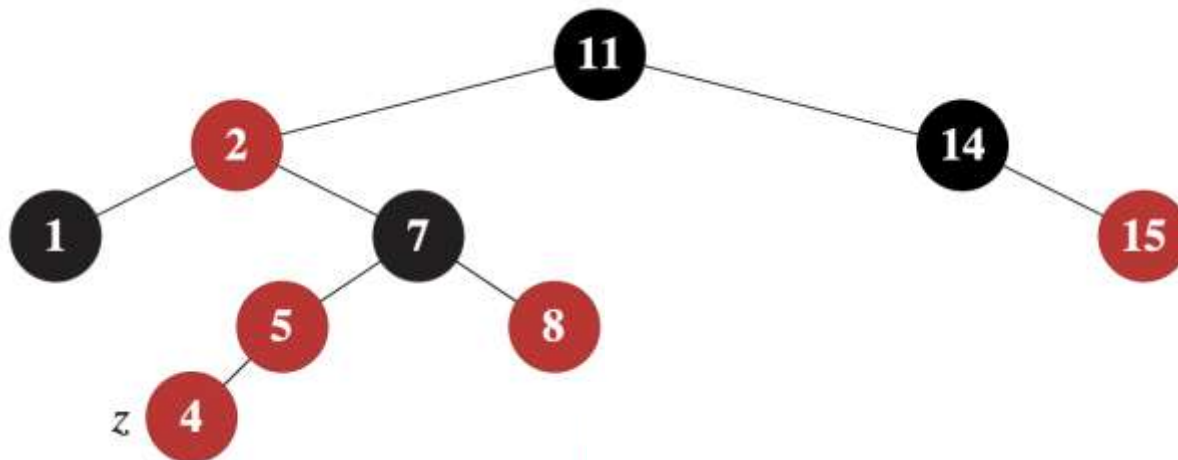
- **Question.** What violations of the red-black properties might occur upon the call to **RB-INSERT-FIXUP** ?
- Property 1 (i.e., every node is either **red** or **black**) continues to hold.
- Property 3 (i.e., every leaf is **black**) also hold.
- Property 5 (i.e., the number of black nodes is the same on every simple path to a leaf from a given node) is satisfied as well because **z** is **red** with sentinel children.

Insertion: Violation of RB properties

- **Question.** What violations of the red-black properties might occur upon the call to **RB-INSERT-FIXUP** ?
- Property 2 (i.e., the root is **black**) is violated if the new node **z** becomes the root (as **z** is coloured **red**).

Insertion: Violation of RB properties

- **Question.** What violations of the red-black properties might occur upon the call to **RB-INSERT-FIXUP** ?
- Property 2 (i.e., the root is **black**) is violated if the new node **z** becomes the root (as **z** is coloured **red**).
- Property 4 (i.e., a **red** node cannot have a **red** child) is violated if **z.p** is **red** as in the example below.



Insertion: Violation of RB properties

- **Question.** What violations of the red-black properties might occur upon the call to **RB-INSERT-FIXUP** ?
- Property 2 (i.e., the root is **black**) is violated if the new node **z** becomes the root (as **z** is coloured **red**).
- Property 4 (i.e., a **red** node cannot have a **red** child) is violated if **z.p** is **red** as in the example below.
- **Important note.** Observe that when **RB-INSERT-FIXUP** is called at most one of Property 2 and Property 4 is violated, but not both.

Insert Fixup: The loop

- The **RB-INSERT-FIXUP** procedure runs a **while** loop to fix the violated RB properties of the tree ***T***.
- Inside the loop, the node ***z*** (which is the newly inserted node to begin with) changes with iterations.

```
RB-INSERT-FIXUP(T, z)
```

```
1  while z.p.color == RED
```

```
    {  
      .  
      .  
      .  
    }
```

```
30 T.root.color = BLACK
```

Insert Fixup: The loop

- The **RB-INSERT-FIXUP** procedure runs a **while** loop to fix the violated RB properties of the tree *T*.
- Inside the loop, the node *z* (which is the newly inserted node to begin with) changes with iterations.

```
RB-INSERT-FIXUP(T, z)
```

```
1  while z.p.color == RED
```

```
{
```

```
·
```

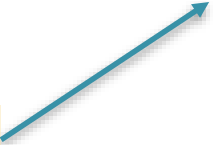
```
·
```

```
·
```

```
}
```

```
30 T.root.color = BLACK
```

Root is colored
black in the end.



Insert Fixup: The loop

- The **RB-INSERT-FIXUP** procedure runs a **while** loop to fix the violated RB properties of the tree ***T***.
- Inside the loop, the node ***z*** (which is the newly inserted node to begin with) changes with iterations.
- The loop, which runs as long as ***z.p.colour*** is **red**, has two symmetric possibilities:
 - **Possibility 1**: ***z.p*** is the left child of ***z.p.p***.
 - **Possibility 2**: ***z.p*** is the right child of ***z.p.p***.

Insert Fixup: The loop

- The **RB-INSERT-FIXUP** procedure runs a **while** loop to fix the violated RB properties of the tree ***T***.
- Inside the loop, the node ***z*** (which is the newly inserted node to begin with) changes with iterations.
- The loop, which runs as long as ***z.p.colour*** is **red**, has two symmetric possibilities:
 - **Possibility 1**: ***z.p*** is the left child of ***z.p.p***.
 - **Possibility 2**: ***z.p*** is the right child of ***z.p.p***.

(We'll see that the loop "invariant" ensures that ***z.p*** & ***z.p.p*** exist.)

Insert Fixup: The loop

- The **RB-INSERT-FIXUP** procedure runs a **while** loop to fix the violated RB properties of the tree ***T***.
- Inside the loop, the node ***z*** (which is the newly inserted node to begin with) changes with iterations.

```
RB-INSERT-FIXUP(T, z)
```

```
1  while z.p.color == RED
```

```
    {
```

```
        Code for Possibility 1.
```

```
        Code for Possibility 2.
```

```
    }
```

```
30 T.root.color = BLACK
```

Insert Fixup: The loop

- The **RB-INSERT-FIXUP** procedure runs a **while** loop to fix the violated RB properties of the tree ***T***.
- Inside the loop, the node ***z*** (which is the newly inserted node to begin with) changes with iterations.

```
RB-INSERT-FIXUP(T, z)
```

```
1  while z.p.color == RED
```

```
{
```

```
    Code for Possibility 1.
```

```
    Code for Possibility 2.
```

```
}
```

```
30 T.root.color = BLACK
```

We'll focus on Possibility 1;
Possibility 2 is symmetric.

Insert Fixup: Loop invariant

- We'll show that the **while** loop maintains the following three-part invariant at the start of every iteration:
 - a. Node z (which changes with iterations) is **red**.
 - b. If $z.p$ is the root, then $z.p$ is **black**.

Insert Fixup: Loop invariant

- We'll show that the **while** loop maintains the following three-part invariant at the start of every iteration:
 - a. Node z (which changes with iterations) is **red**.
 - b. If $z.p$ is the root, then $z.p$ is **black**.
- As the loop runs as long as $z.p.colour$ is **red**, Part **b** will help us show that $z.p.p$ exists. So, one of **Possibility 1** & **Possibility 2** holds true within the loop.

Insert Fixup: Loop invariant

- We'll show that the **while** loop maintains the following three-part invariant at the start of every iteration:
 - a. Node z (which changes with iterations) is **red**.
 - b. If $z.p$ is the root, then $z.p$ is **black**.
 - c. If T violates any of the RB properties, then it violates at most one of them, and the violation is either Property 2 (the root is **black**) or Property 4 (a **red** node cannot have a **red** child), but not both.

Insert Fixup: Loop invariant

- We'll show that the **while** loop maintains the following three-part invariant at the start of every iteration:
 - a. Node z (which changes with iterations) is **red**.
 - b. If $z.p$ is the root, then $z.p$ is **black**.
 - c. If T violates any of the RB properties, then it violates at most one of them, and the violation is either Property 2 (the root is **black**) or Property 4 (a **red** node cannot have a **red** child), but not both. If T violates Property 2, it is because z is the root and it is red.

Insert Fixup: Loop invariant

- We'll show that the **while** loop maintains the following three-part invariant at the start of every iteration:
 - a. Node z (which changes with iterations) is **red**.
 - b. If $z.p$ is the root, then $z.p$ is **black**.
 - c. If T violates any of the RB properties, then it violates at most one of them, and the violation is either Property 2 (the root is **black**) or Property 4 (a **red** node cannot have a **red** child), but not both. If T violates Property 2, it is because z is the root and it is red. (This means, if the loop is entered in an iteration, then Property 2 must hold at the start of the iteration.)

Insert Fixup: Loop invariant

- We'll show that the **while** loop maintains the following three-part invariant at the start of every iteration:
 - a. Node z (which changes with iterations) is **red**.
 - b. If $z.p$ is the root, then $z.p$ is **black**.
 - c. If T violates any of the RB properties, then it violates at most one of them, and the violation is either Property 2 (the root is **black**) or Property 4 (a **red** node cannot have a **red** child), but not both. If T violates Property 2, it is because z is the root and it is red. If T violates Property 4, it is because both z and $z.p$ are red.

Insert Fixup: Loop invariant

- We'll show that the **while** loop maintains the following three-part invariant at the start of every iteration:
 - a. Node z (which changes with iterations) is **red**.
 - b. If $z.p$ is the root, then $z.p$ is **black**.
 - c. If T violates any of the RB properties, then it violates at most one of them, and the violation is either Property 2 (the root is **black**) or Property 4 (a **red** node cannot have a **red** child), but not both. If T violates Property 2, it is because z is the root and it is red. If T violates Property 4, it is because both z and $z.p$ are red.

Essentially, the loop fixes violation of Property 4.

Insert Fixup: Loop invariant

- **Obs.** It follows from the discussion before (Slides 48-53) that the loop invariant is true upon entering the first iteration of the **while** loop.

Insert Fixup: Loop invariant

- **Obs.** It follows from the discussion before (Slides 48-53) that the loop invariant is true upon entering the first iteration of the **while** loop.
- We need to show that each iteration of the loop maintains the invariant, that the loop terminates, & that the invariant gives a useful property at loop termination.

Insert Fixup: Loop invariant

- **Obs.** It follows from the discussion before (Slides 48-53) that the loop invariant is true upon entering the first iteration of the **while** loop.
- We need to show that each iteration of the loop maintains the invariant, that the loop terminates, & that the invariant gives a useful property at loop termination.
- We'll see that each iteration of the loop has **two** possible outcomes: either the pointer z moves up T , or some rotations occur and then the loop terminates. This will help us argue that the complexity of **RB-INSERT-FIXUP** is $O(\log n)$.

Maintaining the invariant

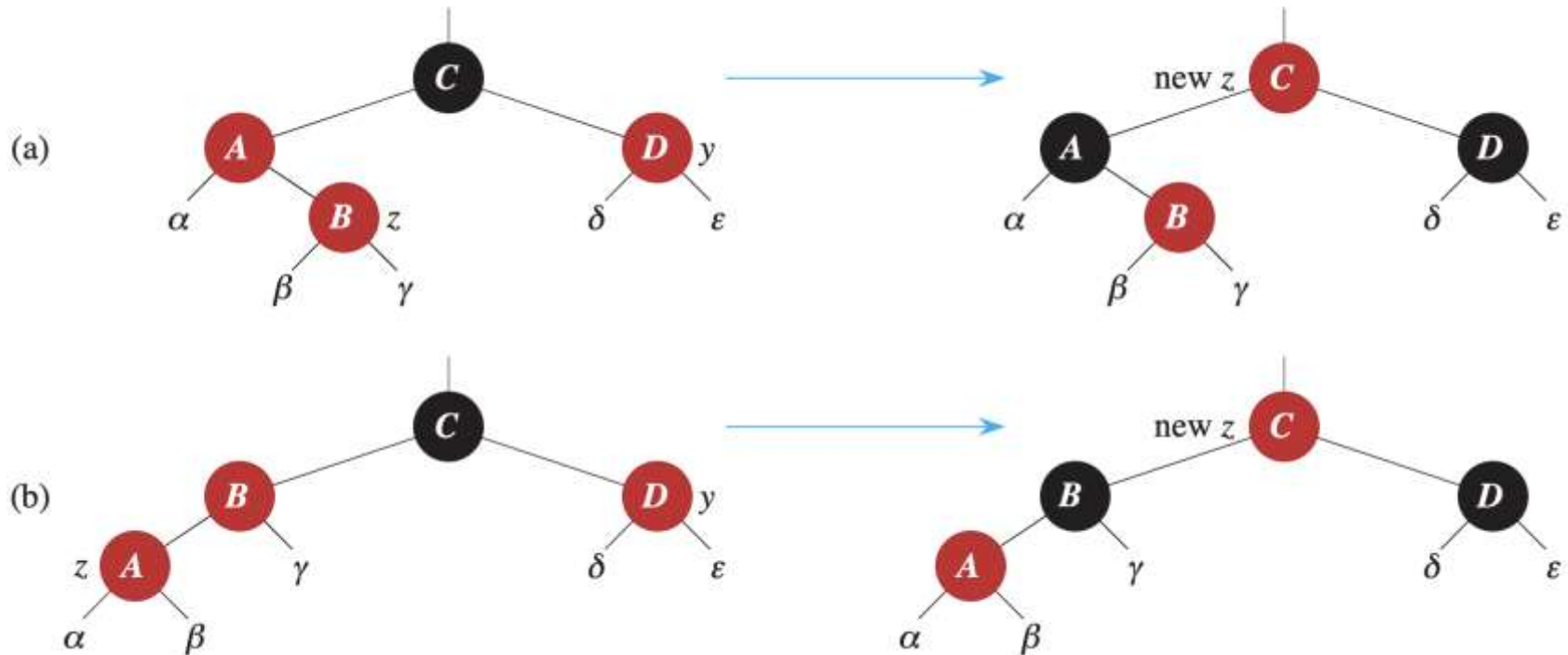
- We'll see how the loop invariant is maintained in **Possibility 1** ($z.p$ is the left child of $z.p.p$) by examining three cases: **Case 1**, **2**, and **3**.
- **Definition.** The sibling of a node's parent will be referred to as the **uncle** of the node.

Maintaining the invariant

- We'll see how the loop invariant is maintained in **Possibility 1** ($z.p$ is the left child of $z.p.p$) by examining three cases: **Case 1**, **2**, and **3**.
- **Definition.** The sibling of a node's parent will be referred to as the **uncle** of the node.
 - **Case 1:** z 's uncle y is **red**.
 - **Case 2:** z 's uncle y is **black** & z is a right child.
 - **Case 3:** z 's uncle y is **black** & z is a left child.

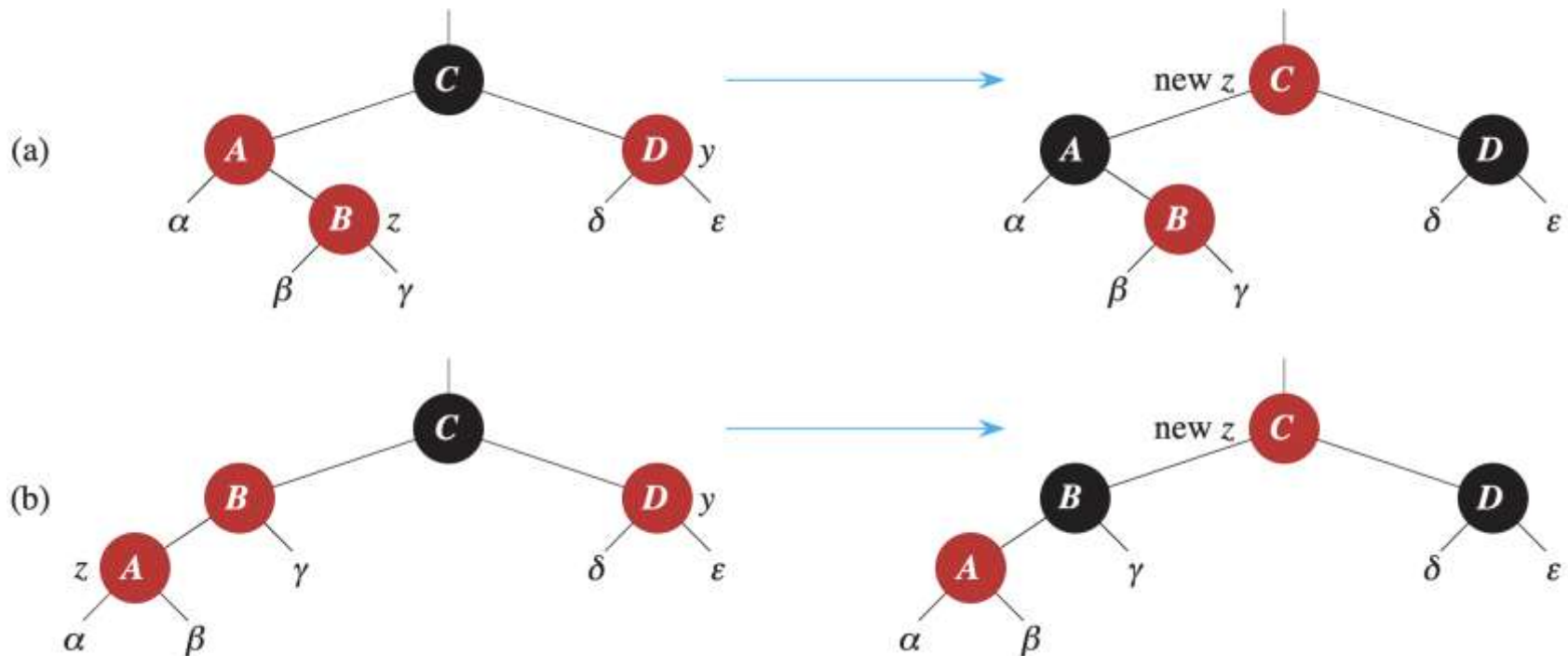
Maintaining the invariant: Case I

- Case I: z 's uncle y is red.



Maintaining the invariant: Case I

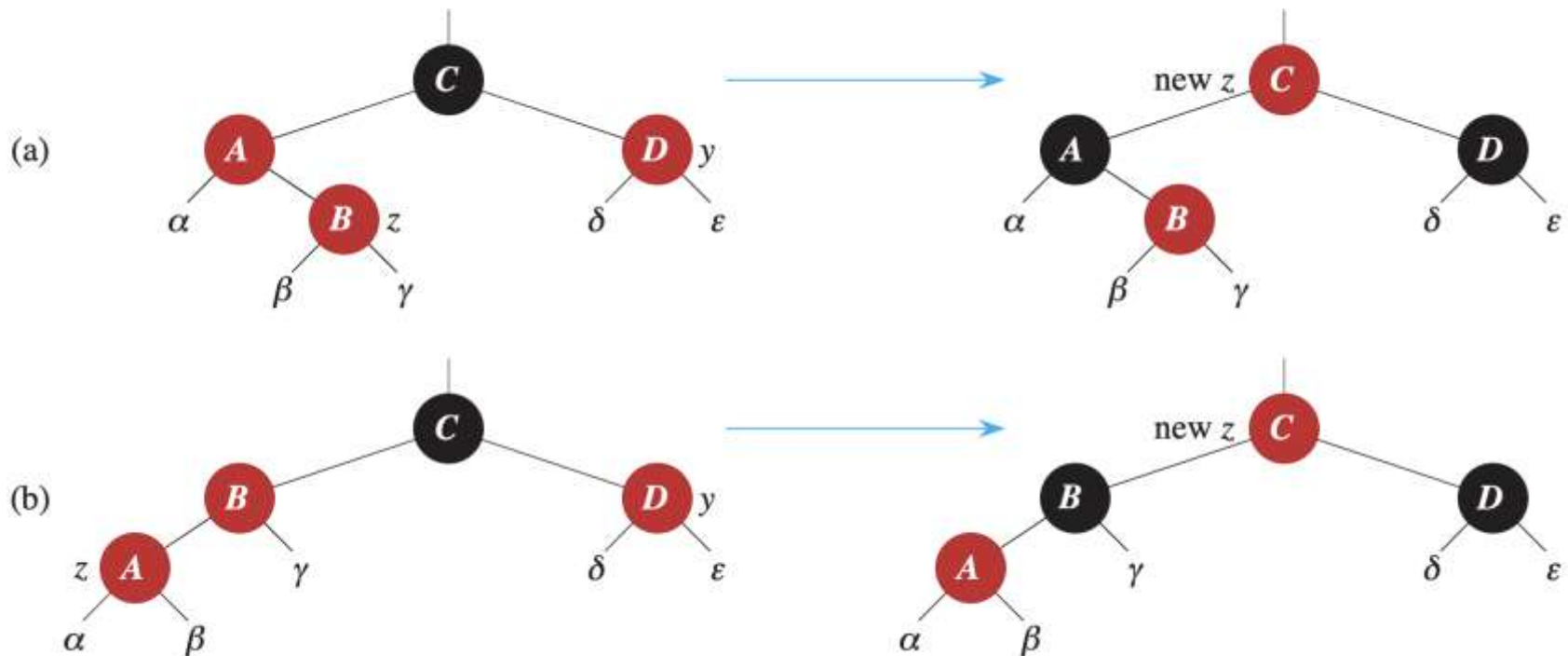
- Case I: z 's uncle y is red.



- Invariant a (z is red) is maintained.

Maintaining the invariant: Case I

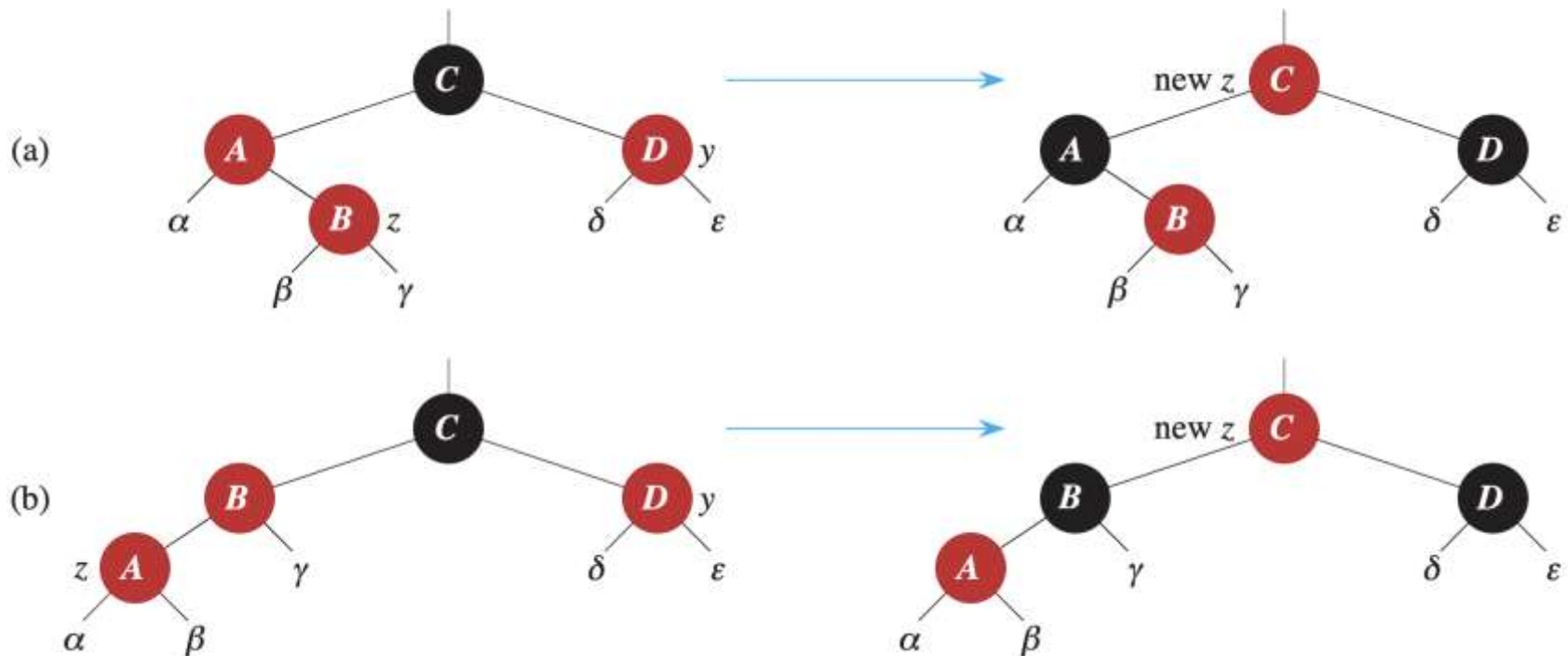
- Case I: z 's uncle y is red.



- Invariant **b** ($z.p$ is the root $\Rightarrow z.p$ in **black**) is maintained. (As RB Property 2 holds at the start of the iteration.)

Maintaining the invariant: Case I

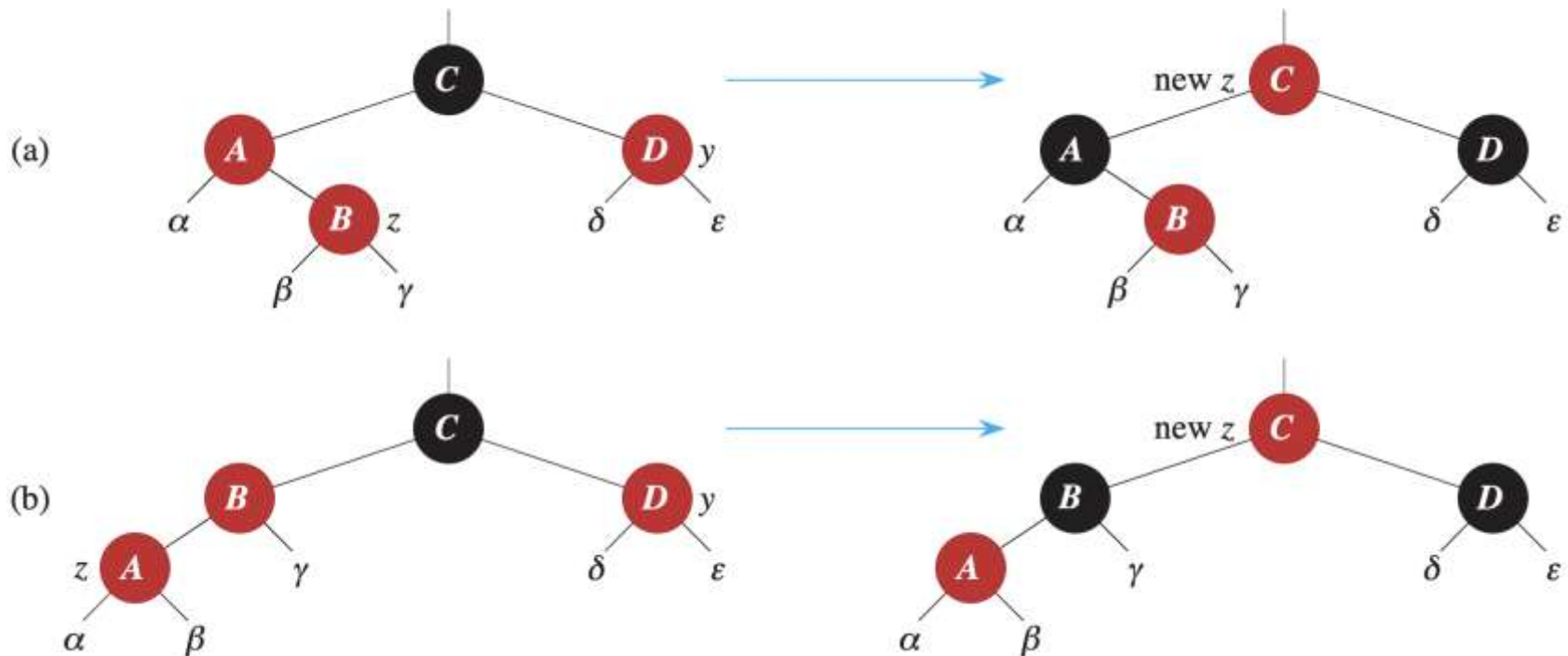
- Case I: z 's uncle y is red.



- Property 1 (a node is red or black), 3 (leaf is black), and 5 (black-height is well-defined) are not violated.

Maintaining the invariant: Case I

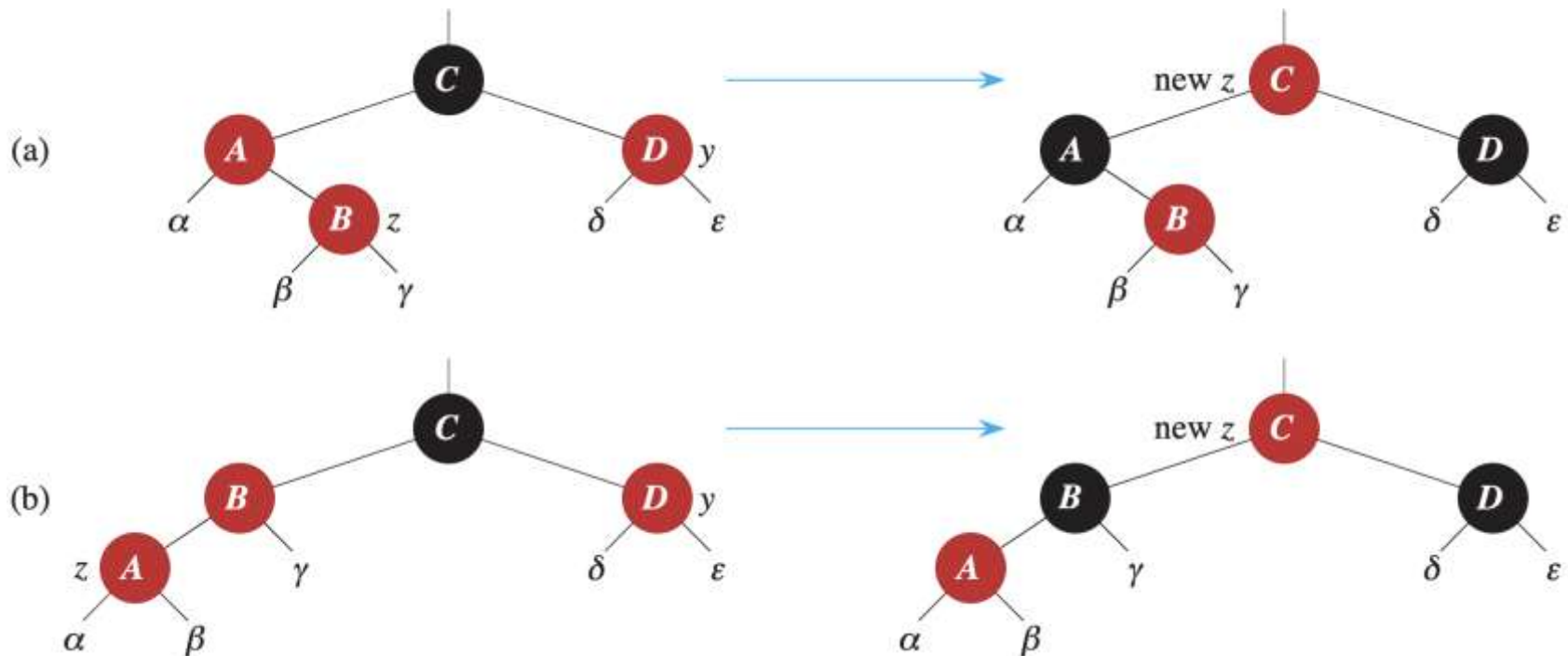
- Case I: z 's uncle y is red.



- If Property 2 (root is **black**) is violated, then it must be that the new z is the root. (Next time, loop is not entered.)

Maintaining the invariant: Case I

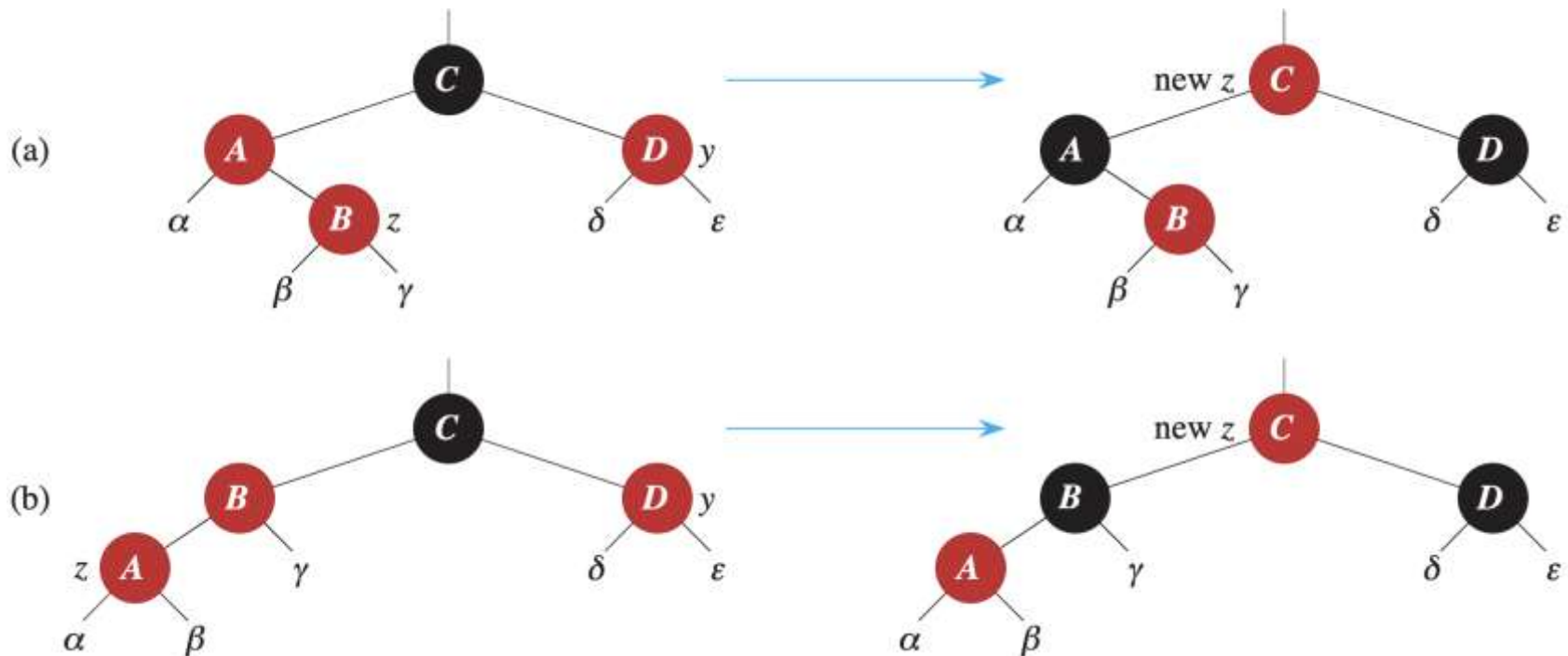
- Case I: z 's uncle y is red.



- If Property 4 (red node has no red child) is violated, then it must be that the new z and its parent are red.

Maintaining the invariant: Case I

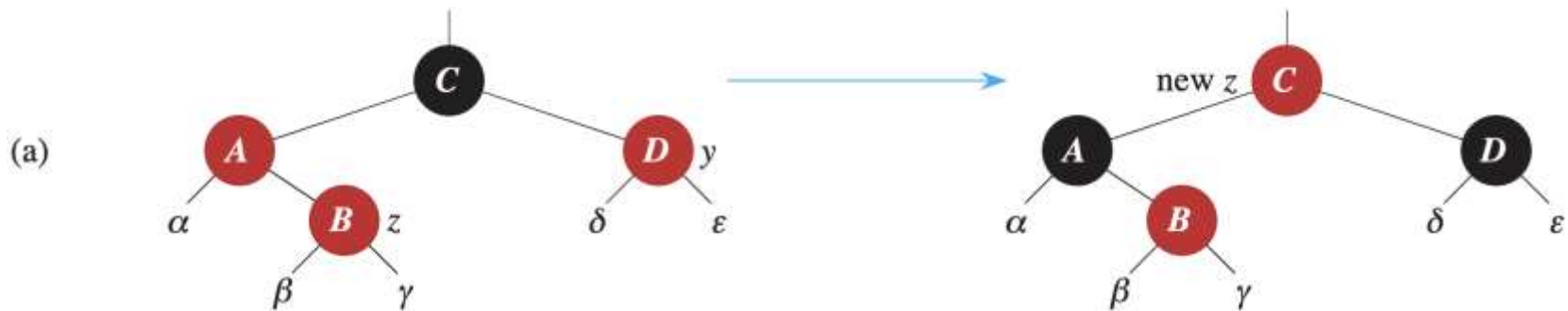
- Case I: z 's uncle y is red.



- So, Invariant c is also maintained.

Maintaining the invariant: Case I

- Case I: z 's uncle y is red.



if $y.color == \text{RED}$

$z.p.color = \text{BLACK}$

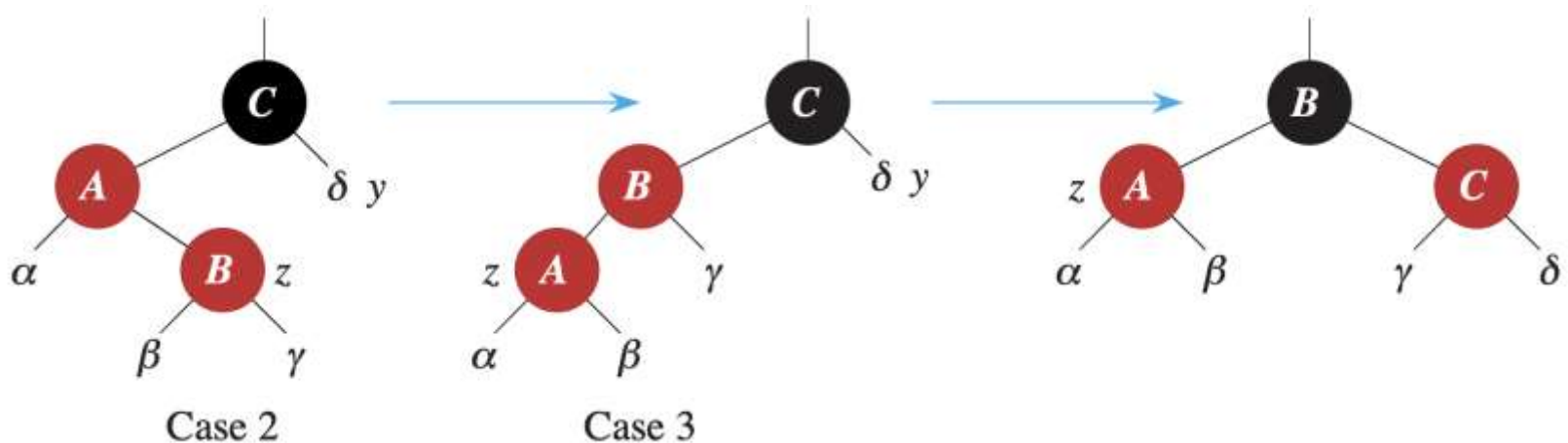
$y.color = \text{BLACK}$

$z.p.p.color = \text{RED}$

$z = z.p.p$

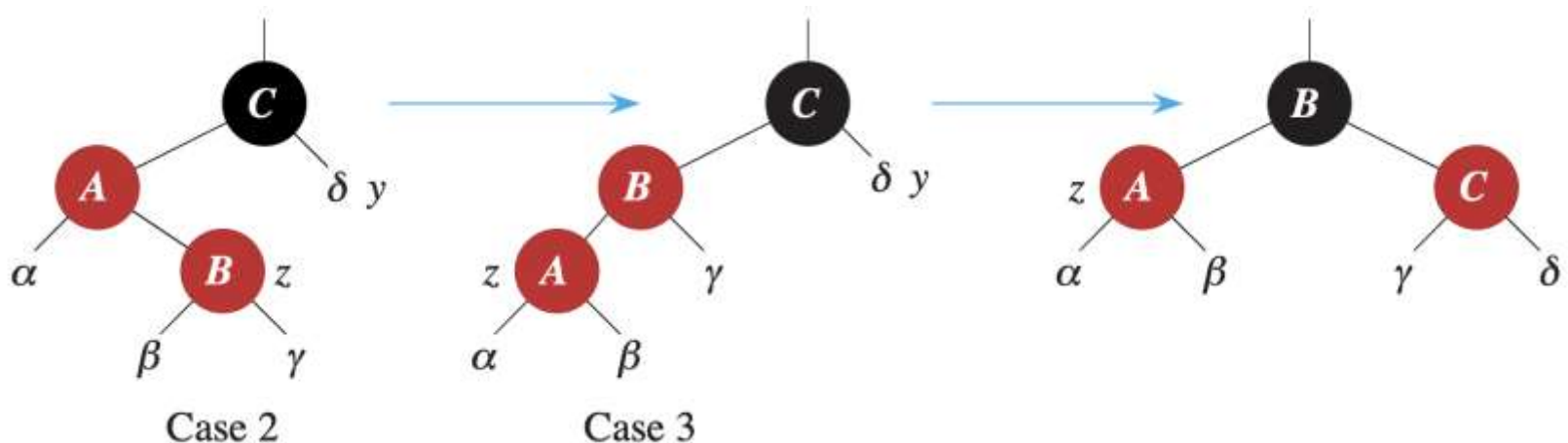
Maintaining the invariant: Case 2 & 3

- Case 2: z 's uncle y is **black** & z is a right child.
- Case 3: z 's uncle y is **black** & z is a left child.



Maintaining the invariant: Case 2 & 3

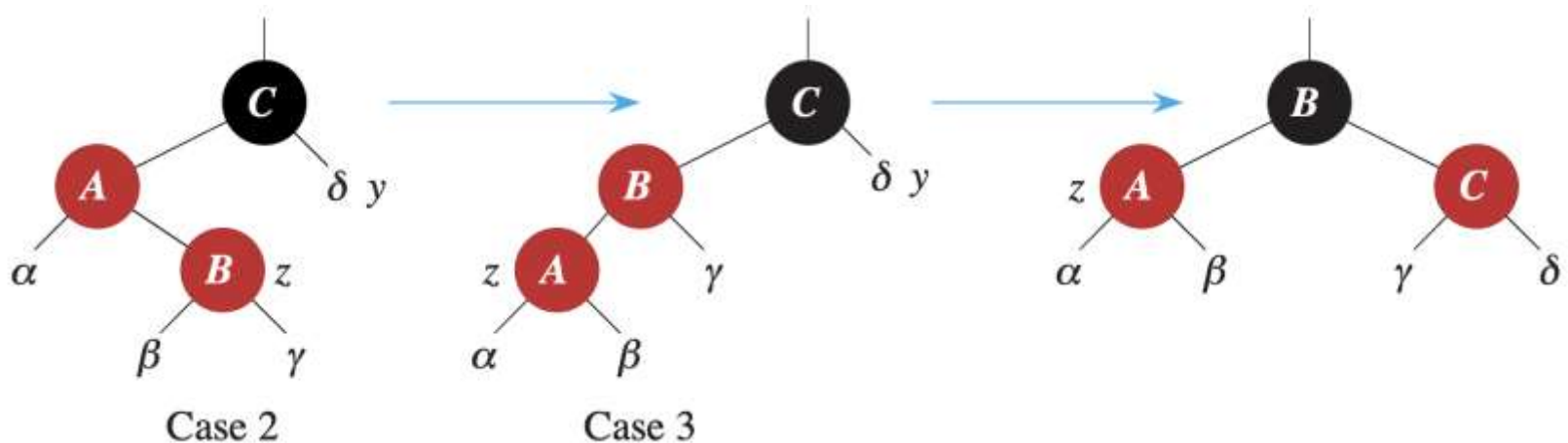
- **Case 2:** z 's uncle y is **black** & z is a right child.
- **Case 3:** z 's uncle y is **black** & z is a left child.



- **Case 2** is reduced to **Case 3** first using a left rotation, and then **Case 3** is handled using a right rotation and two colour changes.

Maintaining the invariant: Case 2 & 3

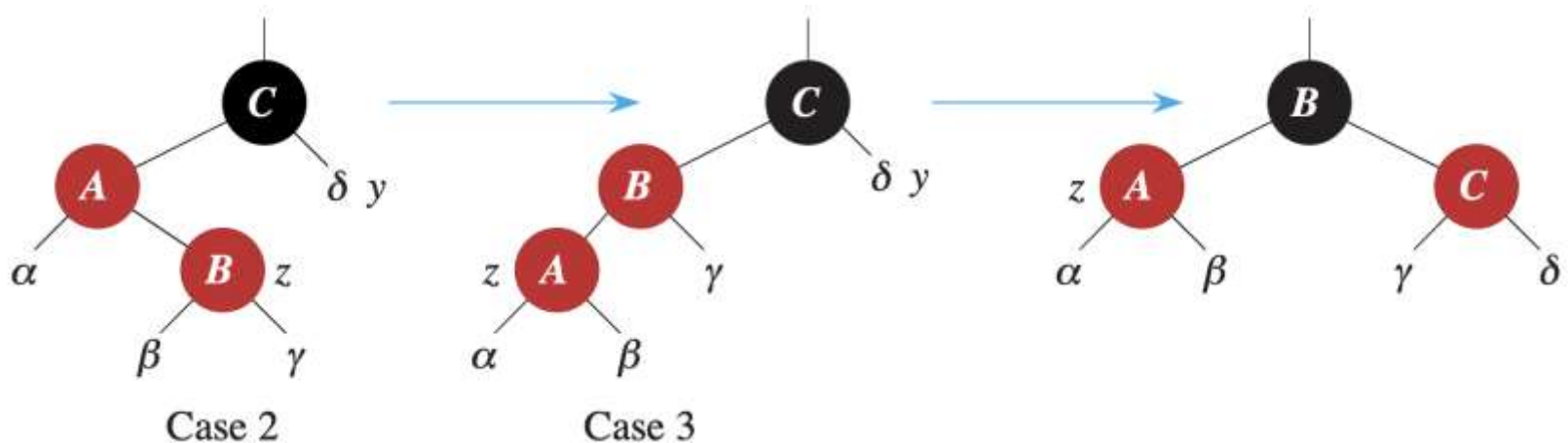
- Case 2: z 's uncle y is **black** & z is a right child.
- Case 3: z 's uncle y is **black** & z is a left child.



- Invariant a (z is **red**) is maintained.

Maintaining the invariant: Case 2 & 3

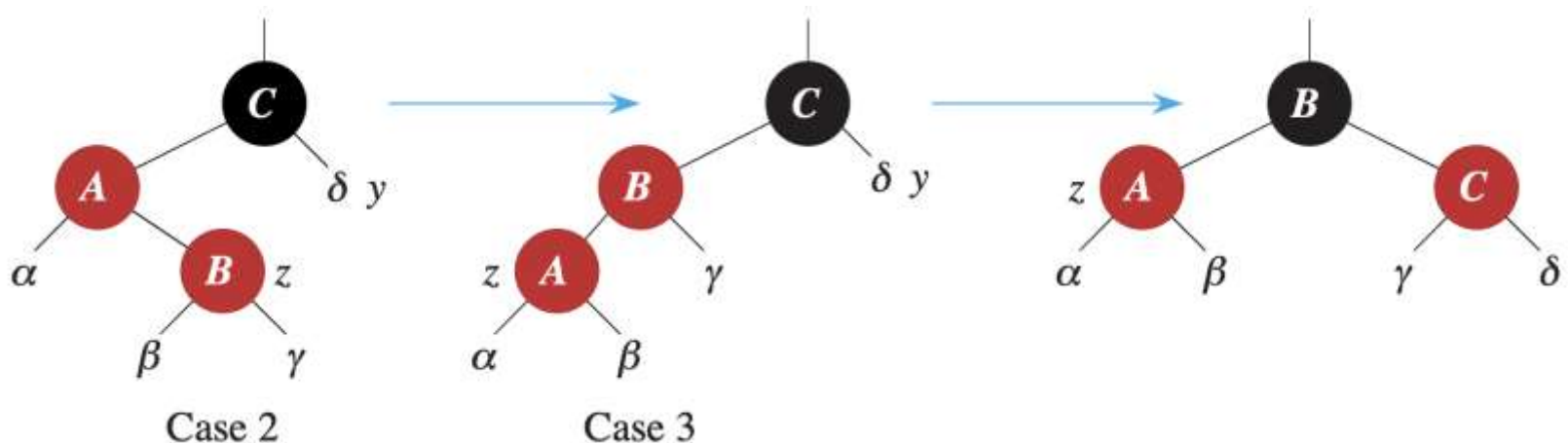
- Case 2: z 's uncle y is **black** & z is a right child.
- Case 3: z 's uncle y is **black** & z is a left child.



- Invariant b ($z.p$ is the root $\Rightarrow z.p$ in **black**) is maintained.

Maintaining the invariant: Case 2 & 3

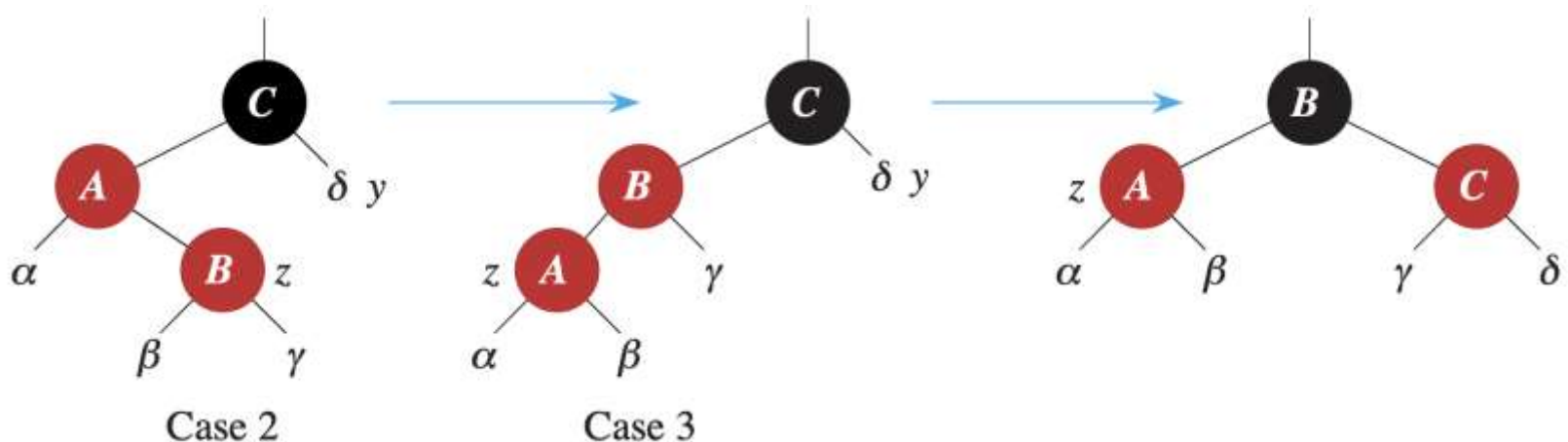
- Case 2: z 's uncle y is **black** & z is a right child.
- Case 3: z 's uncle y is **black** & z is a left child.



- Property 1 (a node is **red** or **black**) and 3 (leaf is **black**) are not violated.

Maintaining the invariant: Case 2 & 3

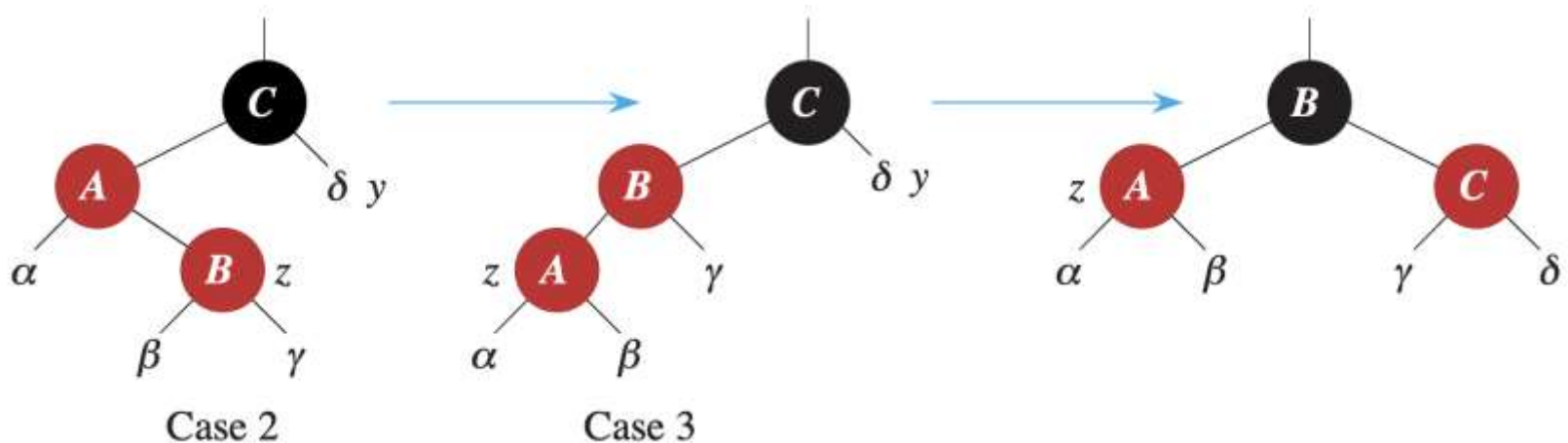
- Case 2: z 's uncle y is **black** & z is a right child.
- Case 3: z 's uncle y is **black** & z is a left child.



- Property 5 (black-height is well-defined) is also not violated as α , β , γ , and δ have the same black-height.

Maintaining the invariant: Case 2 & 3

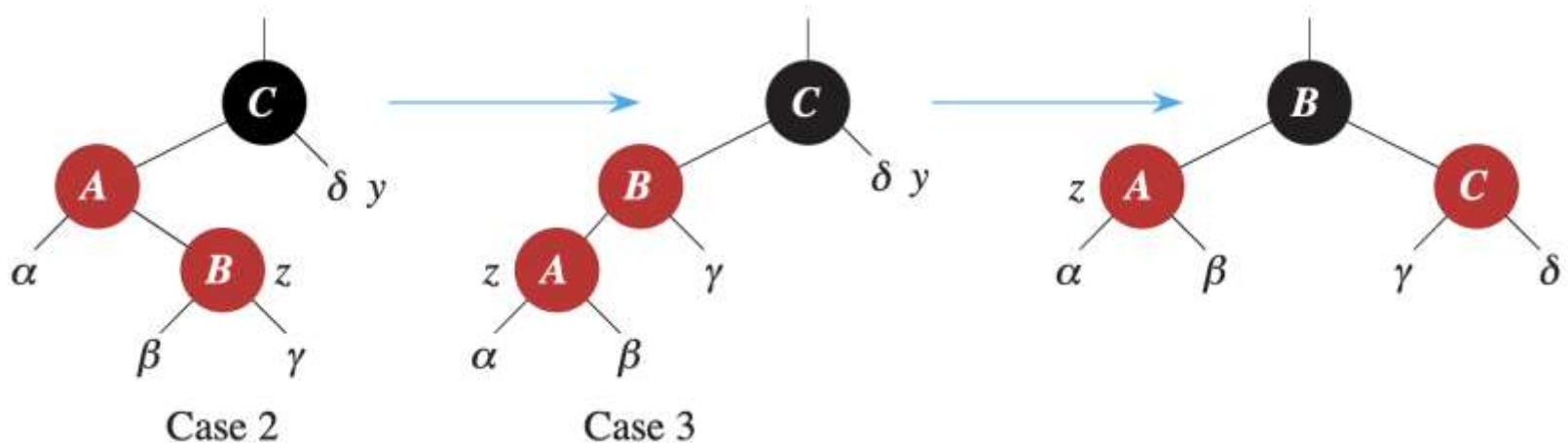
- Case 2: z 's uncle y is **black** & z is a right child.
- Case 3: z 's uncle y is **black** & z is a left child.



- Property 2 (root is **black**) is not violated.

Maintaining the invariant: Case 2 & 3

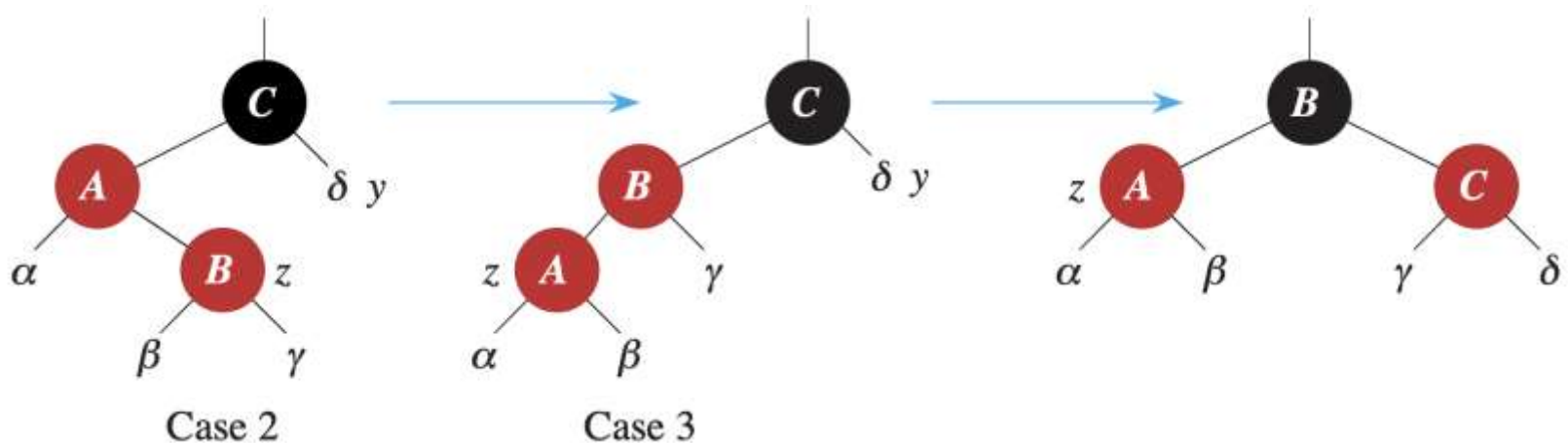
- Case 2: z 's uncle y is **black** & z is a right child.
- Case 3: z 's uncle y is **black** & z is a left child.



- Property 4 (**red** node has no **red** child) is also not violated.

Maintaining the invariant: Case 2 & 3

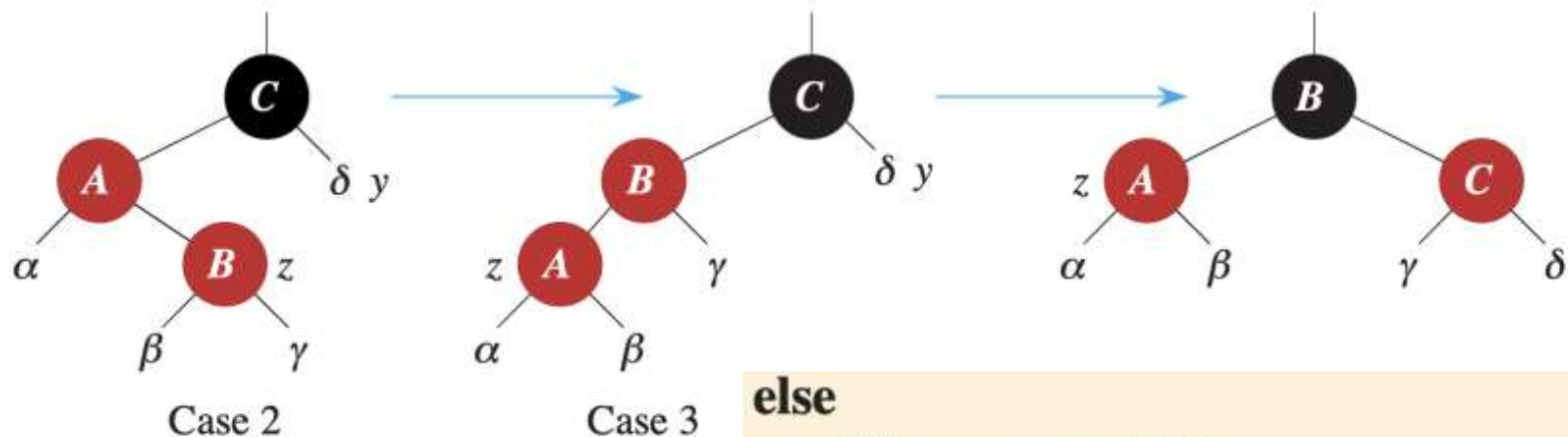
- **Case 2:** z 's uncle y is **black** & z is a right child.
- **Case 3:** z 's uncle y is **black** & z is a left child.



- As none of the five properties are violated, Invariant c is maintained. Observe that the **while** loop won't be entered next time as the colour of $z.p$ isn't **red**.

Maintaining the invariant: Case 2 & 3

- **Case 2:** z 's uncle y is **black** & z is a right child.
- **Case 3:** z 's uncle y is **black** & z is a left child.



else

if $z == z.p.right$

$z = z.p$

LEFT-ROTATE(T, z)

$z.p.color = \text{BLACK}$

$z.p.p.color = \text{RED}$

RIGHT-ROTATE($T, z.p.p$)

} case 2

} case 3

Insert Fixup: Pseudocode

RB-INSERT-FIXUP(T, z)

```
1  while  $z.p.color == RED$ 
2      if  $z.p == z.p.p.left$                 // is  $z$ 's parent a left child?
3           $y = z.p.p.right$                 //  $y$  is  $z$ 's uncle
4          if  $y.color == RED$                 // are  $z$ 's parent and uncle both red?
5               $z.p.color = BLACK$ 
6               $y.color = BLACK$ 
7               $z.p.p.color = RED$ 
8               $z = z.p.p$ 
9          else
10             if  $z == z.p.right$ 
11                  $z = z.p$ 
12                 LEFT-ROTATE( $T, z$ )
13                  $z.p.color = BLACK$ 
14                  $z.p.p.color = RED$ 
15                 RIGHT-ROTATE( $T, z.p.p$ )
16     else // same as lines 3–15, but with “right” and “left” exchanged
```

case 1

case 2

case 3

Code for Possibility 1.

Code for Possibility 2.

30 $T.root.color = BLACK$

Insert: Time complexity

- **Obs.** In **RB-INSERT-FIXUP**, the **while** loop repeats only if **Case 1** occurs, and then the pointer **z** moves two levels up the tree.
- So, the total number of times the **while** loop can be executed is $O(\log n)$.
- Thus, **RB-INSERT** takes a total of $O(\log n)$ time.

Practice problems

Practice Problem I

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Argue that in a red-black tree, a red node cannot have exactly one non-NIL child.

Practice Problem 2

Write pseudocode for RIGHT-ROTATE.

Show that any arbitrary n -node binary search tree can be transformed into any other arbitrary n -node binary search tree using $O(n)$ rotations. (*Hint:* First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain.)

Practice Problem 3

Line 16 of RB-INSERT sets the color of the newly inserted node z to red. If instead z 's color were set to black, then property 4 of a red-black tree would not be violated. Why not set z 's color to black?

Practice Problem 4

Professor Teach is concerned that RB-INSERT-FIXUP might set $T.nil.color$ to RED, in which case the test in line 1 would not cause the loop to terminate when z is the root. Show that the professor's concern is unfounded by arguing that RB-INSERT-FIXUP never sets $T.nil.color$ to RED.

Practice Problem 5

Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.