Red Black Tree

Red-black trees: Overview

- Red-black trees are a variation of binary search trees to ensure that the tree is ***balanced***.
  - » Height is $O(\lg n)$, where n is the number of nodes.
- Operations take $O(\lg n)$ time in the worst case.

Applications

Red Black trees are used in

- Completely Fair Scheduler in Linux Kernel: completely fair scheduler, linux/rbtree.h
- Computational Geometry Data structures
- Java: java.util.TreeMap , java.util.TreeSet .
- C++ STL: map, multimap, multiset.

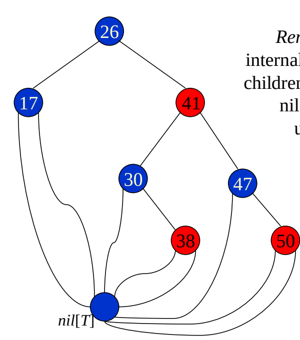
Red-black Tree

- Binary search tree + 1 bit per node: the attribute *color*, which is either **red** or **black**.
- All other attributes of BSTs are inherited:
  - » *key*, *left*, *right*, and *p*.
- All empty trees (leaves) are colored black.
  - » We use a single sentinel, *nil*, for all the leaves of red-black tree T, with *color*[*nil*] = black.
  - » The root's parent is also *nil*[T].

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 paths from the node to descendant leaves contain the same number of black nodes.

Red-black Tree – Example



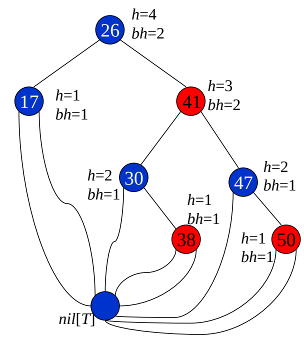
Remember: every internal node has two children, even though nil leaves are not usually shown.

Height of a Red-black Tree

- Height of a node:
  - » $h(x)$ = number of edges in a longest path to a leaf.
- Black-height of a node x, $bh(x)$:
  - » $bh(x)$ = number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x.
- Black-height of a red-black tree is the black-height of its root.
  - » By Property 5(For each node, all paths from the node to descendant leaves contain the same number of black nodes),
  - » black height is well defined.

Height of a Red-black Tree

- Example:
- Height of a node:
  - $h(x)$ = # of edges in a longest path to a leaf.
- Black-height of a node $bh(x)$ = # of black nodes on path from x to leaf, not counting x.
- How are they related?
  - » $bh(x) \le h(x) \le 2\, bh(x)$

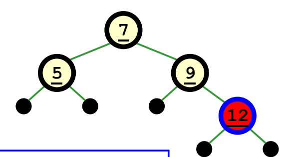


Operations on RB Trees

- All operations can be performed in ***O*(lg *n*)** time.
- The query operations, which don't modify the tree, are performed in exactly the same way as they are in BSTs.
- Insertion and Deletion are not straightforward.

Red-Black Trees: An Example

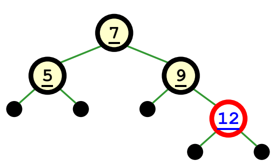
- *Color this tree:*



Red-black properties:
1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

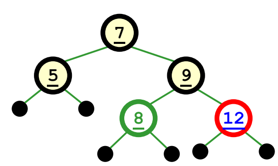
- Insert 8
  - » *Where does it go?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

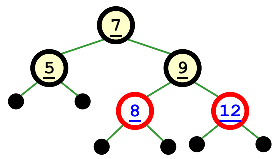
- Insert 8
  - » *Where does it go?*
  - » *What color should it be?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

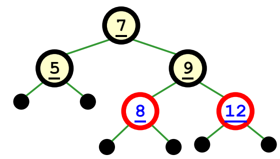
- Insert 8
  - » *Where does it go?*
  - » *What color should it be?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

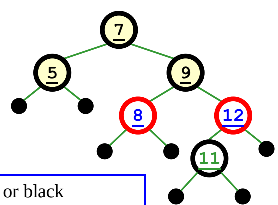
- Insert 11
  - » *Where does it go?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

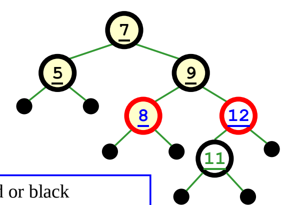
- Insert 11
  - » *Where does it go?*
  - » *What color?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

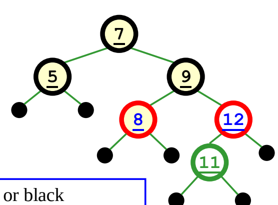
- Insert 11
  - » *Where does it go?*
  - » *What color?*
    - Can’t be red! (#3)



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

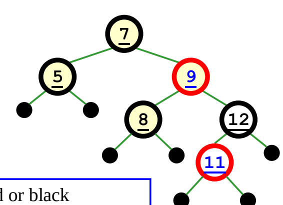
- Insert 11
  - » *Where does it go?*
  - » *What color?*
    - Can’t be red! (#3)
    - Can’t be black! (#4)



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

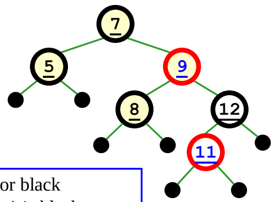
- Insert 11
  - » *Where does it go?*
  - » *What color?*
    - Solution: recolor the tree



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

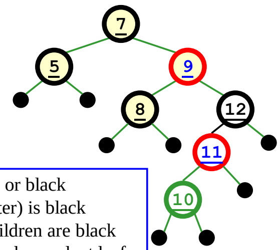
- Insert 10
  - » *Where does it go?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

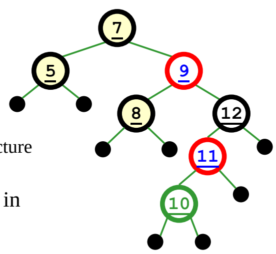
- Insert 10
  - » *Where does it go?*
  - » *What color?*



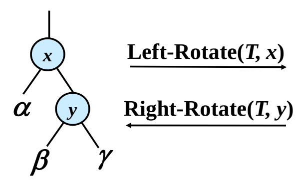
1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

- Insert 10
  - » *Where does it go?*
  - » *What color?*
    - A: no color! Tree is too imbalanced
    - Must change tree structure to allow recoloring
  - » Goal: restructure tree in O(lg n) time

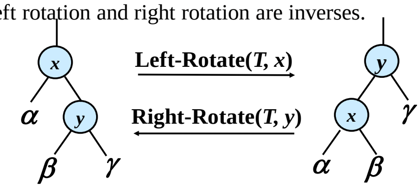


Rotations



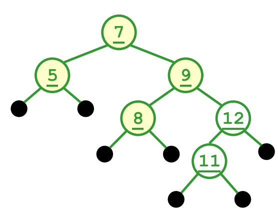
Rotations

- Rotations are the basic tree-restructuring operation for almost all *balanced* search trees.
- Rotation takes a red-black-tree and a node,
- Changes pointers to change the local structure, and
- Won't violate the binary-search-tree property.
- Left rotation and right rotation are inverses.



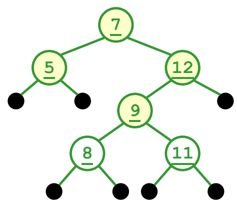
Rotation Example

- Rotate left about 9:



Rotation Example

- Rotate left about 9:



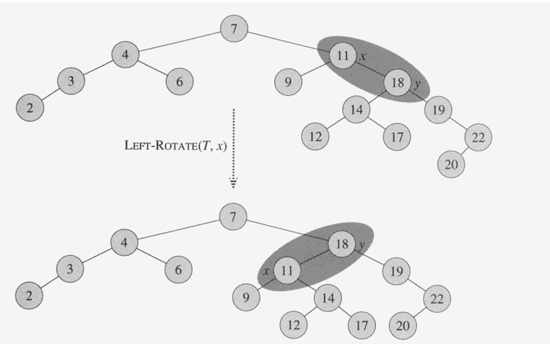
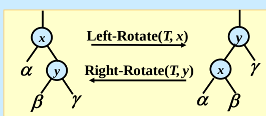


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

Left Rotation – Pseudo-code

Left-Rotate (*T, x*)

```
y ← right[x]   // Set y.
right[x] ← left[y]   //Turn y's left subtree into x's right subtree.
if left[y] ≠ nil[T ]
    then p[left[y]] ← x
p[y] ← p[x]          // Link x's parent to y.
if p[x] = nil[T ]
    then root[T ] ← y
    else if x = left[p[x]]
        then left[p[x]] ← y
        else right[p[x]] ← y
left[y] ← x          // Put x on y's left.
p[x] ← y
```



Rotation

- The pseudo-code for Left-Rotate assumes that
  - » *right*[x] ≠ *nil*[T], and
  - » root's parent is *nil*[T].
- Left Rotation on x, makes x the left child of y, and the left subtree of y into the right subtree of x.
- Pseudocode for Right-Rotate is symmetric: exchange *left* and *right* everywhere.
- ***Time:*** $O(1)$ for both Left-Rotate and Right-Rotate, since a constant number of pointers are modified.

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 paths from the node to descendant leaves contain the same number of black nodes.

Insertion in RB Trees

- Insertion must preserve all red-black properties.
- Should an inserted node be colored **Red**? **Black**?
- Basic steps:
  - » Use Tree-Insert from BST (slightly modified) to insert a node x into T.
    - Procedure **RB-Insert(*x*)**.
  - » **Color** the **node *x* red.**
  - » Fix the modified tree by **re-coloring nodes** and **performing rotation** to **preserve RB tree property**.
    - Procedure **RB-Insert-Fixup**.

Insertion

RB-Insert(*T*, *z*)

```
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
   do y ← x
      if key[z] < key[x]
         then x ← left[x]
         else x ← right[x]
p[z] ← y
if y = nil[T]
   then root[T] ← z
   else if key[z] < key[y]
      then left[y] ← z
      else right[y] ← z
```

RB-Insert(*T*, *z*) Contd.

```
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
RB-Insert-Fixup (T, z)
```

Which of the RB properties might be violated?

Fix the violations by calling RB-Insert-Fixup.

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 paths from the node to descendant leaves contain the same number of black nodes.

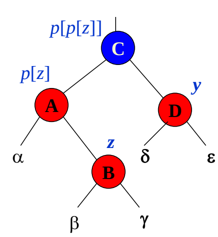
Properties violations

- **Property 1** (each node black or red): hold
- **Property 3** (each leaf is black sentinel): hold.
- **Property 5**: same number of blacks: hold
- **Property 2**: (root is black), not, if z is root (and colored red).
- **Property 4: (the child of a red node must be black), not, if z's parent is red.**

Insertion – Fixup

- Problem: **We may have one pair of consecutive reds where we did the insertion.**
- Solution: **Rotate the tree and then Three cases have to be handled**

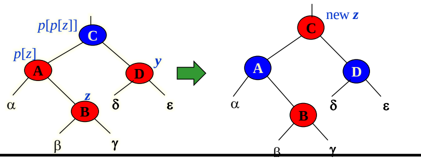
- **Case 1, 2, 3**: p[z] is the left child of p[p[z]].
- Correspondingly, **there are 3 other cases**,
- In which p[z] is the right child of p[p[z]]



Insertion – Fixup

RB-Insert-Fixup (T, z)

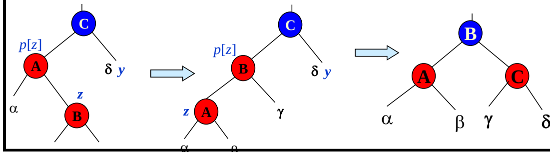
```
while color[p[z]] = RED
   do if p[z] = left[p[p[z]]]
        then y ← right[p[p[z]]]
             if color[y] = RED
                then color[p[z]] ← BLACK   // Case 1
                     color[y] ← BLACK      // Case 1
                     color[p[p[z]]] ← RED  // Case 1
                     z ← p[p[z]]           // Case 1
```



Insertion – Fixup

RB-Insert-Fixup(T, z) (Contd.)

```
             else if z = right[p[z]]  // color[y] ≠ RED
                   then z ← p[z]                    // Case 2
                        LEFT-ROTATE(T, z)           // Case 2
                   color[p[z]] ← BLACK              // Case 3
                   color[p[p[z]]] ← RED             // Case 3
                   RIGHT-ROTATE(T, p[p[z]])         // Case 3
   else (if p[z] = right[p[p[z]]])(same as then clause
        with "right" and "left" exchanged)
color[root[T]] ← BLACK
```



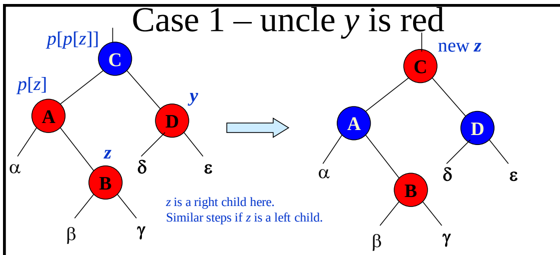
Insertion – Fixup

Termination

The loop terminates only if $p[z]$ is black. Hence, property 4 is OK.
The last line ensures property 2 always holds.

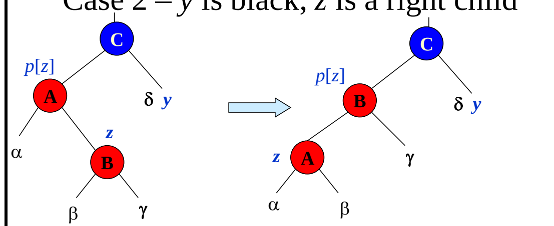

```
RB-INSERT-FIXUP(T, z)
 while color[p[z]] = RED
     do if p[z] = left[p[p[z]]]
           then y ← right[p[p[z]]]
                if color[y] = RED
                   then color[p[z]] ← BLACK                 ▷ Case 1
                        color[y] ← BLACK                    ▷ Case 1
                        color[p[p[z]]] ← RED                ▷ Case 1
                        z ← p[p[z]]                         ▷ Case 1
                   else if z = right[p[z]]
                           then z ← p[z]                    ▷ Case 2
                                LEFT-ROTATE(T, z)           ▷ Case 2
                        color[p[z]] ← BLACK                 ▷ Case 3
                        color[p[p[z]]] ← RED                ▷ Case 3
                        RIGHT-ROTATE(T, p[p[z]])            ▷ Case 3
           else (same as then clause
                        with "right" and "left" exchanged)
 color[root[T]] ← BLACK
```

Case 1 – uncle *y* is red



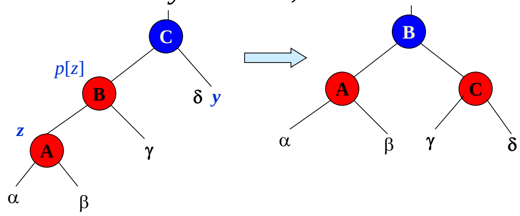
- $p[p[z]]$ (z's grandparent) must be black, since **z and $p[z]$ are both red** and there are no other violations of property 4.
- Make $p[z]$ and y black ⇒ now z and $p[z]$ are not both red. **But property 5 might now be violated.**
- Make $p[p[z]]$ red ⇒ restores property 5.
- The next iteration has $p[p[z]]$ as the new z (i.e., z moves up 2 levels).

Case 2 – *y* is black, *z* is a right child



- Left rotate around $p[z]$, $p[z]$ and z switch roles ⇒ **now z is a left child, and both z and $p[z]$ are red.**
- Takes us immediately to case 3.

Case 3 – *y* is black, *z* is a left child



- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate on $p[p[z]]$. Ensures **property 4 is maintained**.
- No longer have 2 reds in a row.
- ***p*[*z*] is now black** $\Rightarrow$ no more iterations.

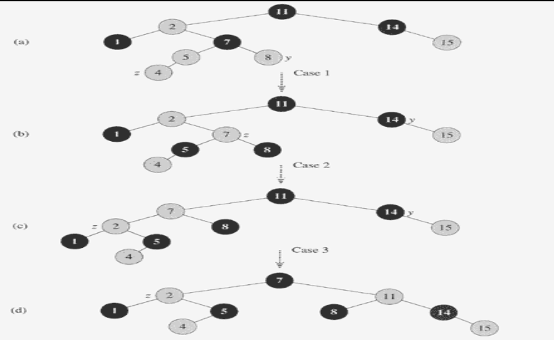


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Since z and its parent $p[z]$ are both red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code can be applied. Nodes are recolored and the pointer z is moved up the tree, resulting in the tree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

Algorithm Analysis

- $O(\lg n)$ time to get through RB-Insert up to the call of RB-Insert-Fixup.
- Within RB-Insert-Fixup:
  - » Each iteration takes $O(1)$ time.
  - » Each iteration but the last **moves *z* up 2 levels**.
  - » $O(\lg n)$ levels $\Rightarrow O(\lg n)$ time.
  - » Thus, insertion in a red-black tree takes $O(\lg n)$ time.
  - » There are at most 2 rotations overall.

Deletion

- Deletion, like insertion, should preserve all the RB properties.
- The properties that may be violated depends on the color of the deleted node.
  - » **Red – OK. <u>Why?</u>**
  - » Black?
- Steps:
  - » Do regular BST deletion.
  - » **Fix any violations of RB properties** that may result.

Deletion

<u>RB-Delete(*T*, *z*)</u>
1. **if** *left*[*z*] = *nil*[*T*] or *right*[*z*] = *nil*[*T*]
2. **then** *y* ← *z*
3. **else** *y* ← TREE-SUCCESSOR*(z)*
4. **if** *left*[*y*] = *nil*[*T*]
5. **then** *x* ← *left*[*y*]
6. **else** *x* ← *right*[*y*]
7. *p*[*x*] ← *p*[*y*] // Do this, even if *x* is *nil*[*T*]

Deletion

<u>RB-Delete (*T*, *z*) (Contd.)</u>

8. **if** *p*[*y*] = *nil*[*T*]
9. **then** *root*[*T*] ← *x*
10. **else if** *y* = *left*[*p*[*y*]]
11. **then** *left*[*p*[*y*]] ← *x*
12. **else** *right*[*p*[*y*]] ← *x*
13. **if** *y* = *z*
14. **then** *key*[*z*] ← *key*[*y*]
15. copy *y*'s satellite data into *z*
16. **if** *color*[*y*] = BLACK
17. **then** RB-Delete-Fixup(*T*, *x*)
18. **return** *y*

The node passed to the fixup routine is the lone child of the spliced up node, or the sentinel.

RB Properties Violation

- If y is black, we could have violations of red-black properties:
  - Prop. 1. OK.
  - Prop. 2. If y is the root and x is red, then the root has become red.
  - Prop. 3. OK.
  - Prop. 4. Violation if $p[y]$ and x are both red.
  - Prop. 5. Any path containing y now has 1 fewer black node.

RB Properties Violation

- Prop. 5. Any path containing y now has 1 fewer black node.
  - Correct by giving x an "extra black."
  - Add 1 to count of black nodes on paths containing x.
  - Now property 5 is OK, but property 1 is not.
  - x is either ***doubly black*** (if $color[x]$ = BLACK) or ***red & black*** (if $color[x]$ = RED).
  - The attribute $color[x]$ is still either RED or BLACK. No new values for *color* attribute.
  - In other words, the extra blackness on a node is by virtue of x pointing to the node.
- Remove the violations by calling RB-Delete-Fixup.

Deletion – Fixup

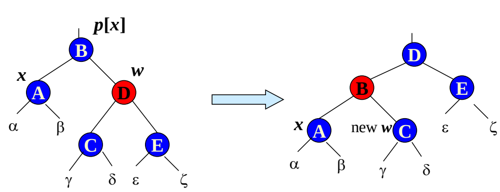
```
RB-Delete-Fixup(T, x)
while x ≠ root[T ] and color[x] = BLACK
    do if x = left[p[x]]
        then w ← right[p[x]]
            if color[w] = RED
                then color[w] ← BLACK          // Case 1
                    color[p[x]] ← RED          // Case 1
                    LEFT-ROTATE(T, p[x])       // Case 1
                    w ← right[p[x]]            // Case 1
```

```
RB-Delete-Fixup(T, x) (Contd.)
    /* x is still left[p[x]] */
    if color[left[w]] = BLACK and color[right[w]] = BLACK
        then color[w] ← RED                    // Case 2
            x ← p[x]                           // Case 2
        else if color[right[w]] = BLACK
                then color[left[w]] ← BLACK    // Case 3
                    color[w] ← RED             // Case 3
                    RIGHT-ROTATE(T,w)          // Case 3
                    w ← right[p[x]]            // Case 3
            color[w] ← color[p[x]]             // Case 4
            color[p[x]] ← BLACK                // Case 4
            color[right[w]] ← BLACK            // Case 4
            LEFT-ROTATE(T, p[x])               // Case 4
            x ← root[T ]                       // Case 4
    else (same as then clause with "right" and "left" exchanged)
color[x] ← BLACK
```

Deletion – Fixup

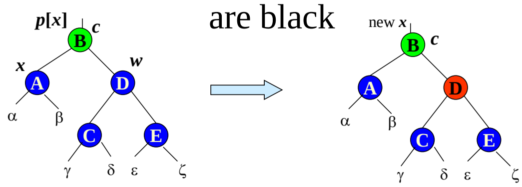
- ***Idea:*** Move the extra black up the tree until x points to a red & black node ⇒ turn it into a black node,
- x points to the root ⇒ just remove the extra black, or
- We can do certain rotations and recolorings and finish.
- Within the **while** loop:
  - x always points to a nonroot doubly black node.
  - w is x's sibling.
  - w cannot be $nil[T]$, since that would violate property 5 at $p[x]$.
- 8 cases in all, 4 of which are symmetric to the other.

Case 1 – *w* is red



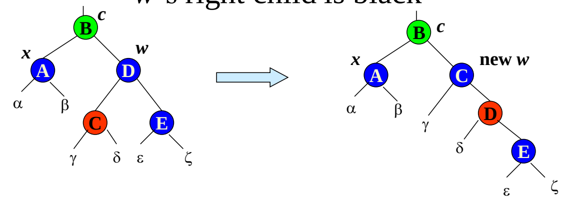
- w must have black children.
- Make w black and $p[x]$ red (because w is red $p[x]$ couldn't have been red).
- Then left rotate on $p[x]$.
- New sibling of x was a child of w before rotation ⇒ must be black.
- Go immediately to case 2, 3, or 4.

Case 2 – *w* is black, both *w*’s children are black



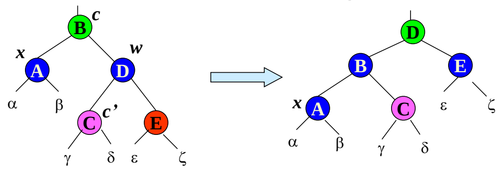
- Take 1 black off x ($\Rightarrow$ singly black) and off w ($\Rightarrow$ red).
- Move that black to $p[x]$.
- Do the next iteration with $p[x]$ as the new x.
- If entered this case from case 1, then $p[x]$ was red $\Rightarrow$ new x is red & black $\Rightarrow$ color attribute of new x is RED $\Rightarrow$ loop terminates. Then new x is made black in the last line.

Case 3 – *w* is black, *w*’s left child is red, *w*’s right child is black



- Make w red and w’s left child black.
- Then right rotate on w.
- New sibling w of x is black with a red right child $\Rightarrow$ case 4.

Case 4 – *w* is black, *w*’s right child is red



- Make w be $p[x]$’s color (c).
- Make $p[x]$ black and w’s right child black.
- Then left rotate on $p[x]$.
- Remove the extra black on x ($\Rightarrow$ x is now singly black) without violating any red-black properties.
- All done. Setting x to root causes the loop to terminate.

Analysis

- $O(\lg n)$ time to get through RB-Delete up to the call of RB-Delete-Fixup.
- Within RB-Delete-Fixup:
  - » Case 2 is the only case in which more iterations occur.
    - x moves up 1 level.
    - Hence, $O(\lg n)$ iterations.
  - » Each of cases 1, 3, and 4 has 1 rotation $\Rightarrow \leq 3$ rotations in all.
  - » Hence, $O(\lg n)$ time.