Design and Analysis of Algorithms

Red-Black Trees
Lecture

Instructor: Dr. G P Gupta

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.

Red-black Tree

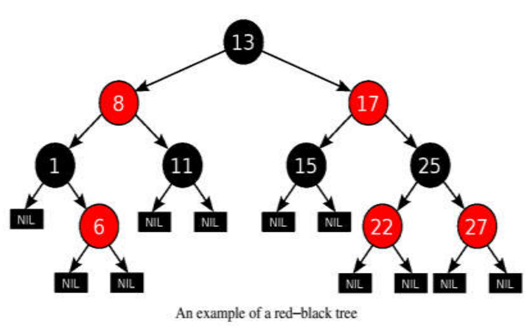
- A **red-black** tree is a ***binary search tree with one extra bit of storage*** per node: its color, which can be either **RED** or **BLACK**.
- All other attributes of BSTs are inherited:
  - » *key*, *left*, *right*, and *p*.
- **All empty trees (leaves) are colored black**.
  - » We use a single 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

A red-black tree is a binary tree that satisfies the following 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 any node ***x*** to a descendant leaf have the same number of black nodes = **black-height(*x*)**.

Red-black Tree – Example

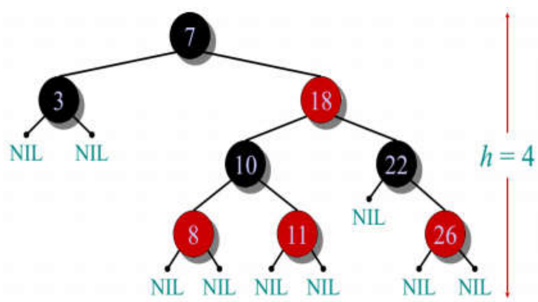


An example of a red–black tree

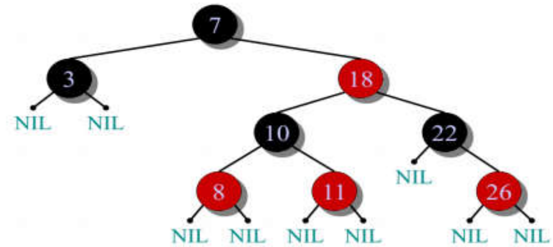
Height of a Red-black Tree

- **Height of a node:**
  - » $h(x)$ = number of edges in a longest path to a leaf.
- **Black-height of a node x, $bh(x)$:**
  - » $\boldsymbol{bh(x)}$ = 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, black height is well defined.

Example of a red-black tree

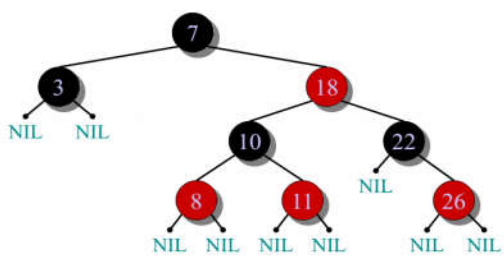


Example of a red-black tree



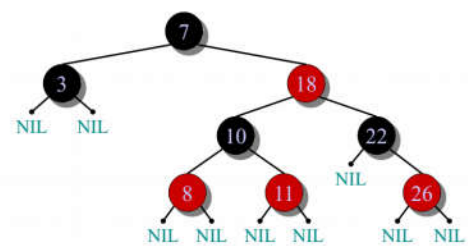
1. Every node is either red or black.

Example of a red-black tree



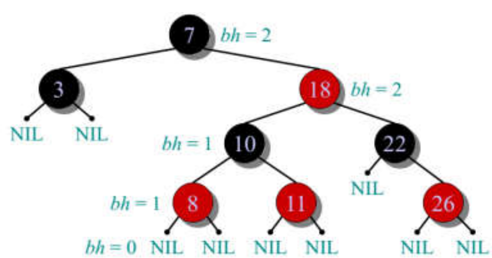
2. The root and leaves (NIL's) are black.

Example of a red-black tree



3. If a node is red, then its parent is black.

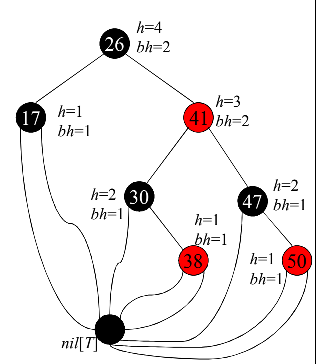
Example of a red-black tree



4. All simple paths from any node x to a descendant leaf have the same number of black nodes = *black-height*(x).

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?**
  » $bh(x) \le h(x) \le 2\, bh(x)$

red-black tree

- **Lemma:** ***The subtree rooted at any node x has ≥ $2^{bh(x)}$–1 internal nodes.***
- **Proof:** By induction on height of x.
  - » **Base Case:** Height $h(x) = 0 \Rightarrow x$ is a leaf $\Rightarrow$ bh(x) = 0.
    Subtree has 2^0–1 = 0 nodes. √
  - » **Induction Step:** Height $h(x) = h > 0$ and $bh(x) = b$.
    - Each child of x has height $h - 1$ and black-height either b (child is red) or $b - 1$ (child is black).
    - **By Induction hypothesis**, each child has ≥ $2^{bh(x)-1}$–1 internal nodes.
    - **Subtree rooted at x has** ≥ 2 ($2^{bh(x)-1} - 1$)+ 1 = $2^{bh(x)} - 1$ internal nodes. (**The + 1 is for x itself.**)

Height of a red-black tree

Theorem. A red-black tree with n keys has height $h \le 2\lg(n + 1)$.

Proof:

• **Lemma:** The subtree rooted at any node x has ≥ $2^{bh(x)}$–1 internal nodes.

• By the above lemma, $n \ge 2^{bh} - 1$,

and since $bh \ge h/2$, we have $n \ge 2^{h/2} - 1$.

$\Rightarrow h \le 2\lg(n + 1)$.

Lemma "RB Height"

Lemma 1: Consider a node x in an RB tree: The longest descending path from x to a leaf has length $h(x)$, which is at most twice the length of the shortest descending path from x to a leaf.

Proof:

black nodes on any path from $x = bh(x)$ (prop 5)

≤ # nodes on shortest path from x, $s(x)$. (prop 1)

But, there are no consecutive red (prop 4),

and we end with black (prop 3), so $h(x) \le 2\, bh(x)$.

Thus, $h(x) \le 2\, s(x)$.

Operations on RB Trees

- **Insertion and Deletion**
- **The queries :** ***SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR***
  - all run in $O(\lg n)$ time on a red-black tree with n nodes.
- All operations can be performed in ***O*(lg *n*)** time.

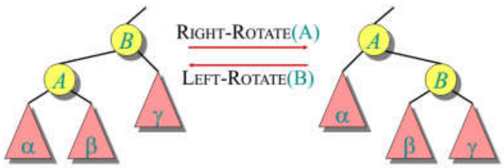
Operations on RB Trees

- Insertion and Deletion ***are not straightforward***. Why?

Operations on RB Trees

- The operations INSERT and DELETE cause modifications to the red-black tree:
  - ***color changes,***
  - ***restructuring the links of the tree: "rotations".***

Rotations



Rotations maintain the inorder ordering of keys:

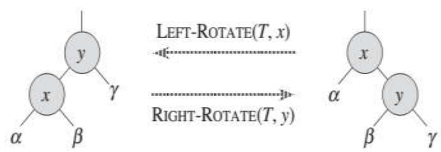
- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c$.

A rotation can be performed in $O(1)$ time.

Rotation

- The pseudo-code for Left-Rotate assumes that
  - » $right[x] \neq nil[T]$, 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.

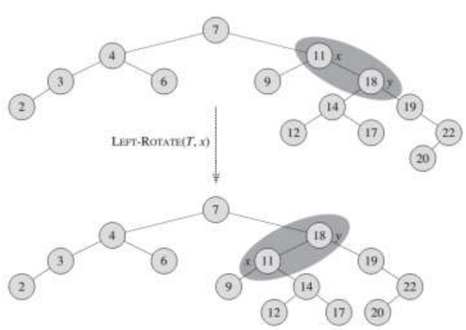
LEFT-ROTATE



LEFT-ROTATE

```
LEFT-ROTATE(T, x)
y = x.right              // set y
x.right = y.left         // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x               // put x on y's left
x.p = y
```

LEFT-ROTATE



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 z into T.
    - Procedure **RB-Insert(T,z)**.
  - » Color the node z red.
  - » Fix the modified tree ***by re-coloring nodes*** and performing **rotation** to preserve RB tree property.
    - Procedure **RB-Insert-Fixup**.

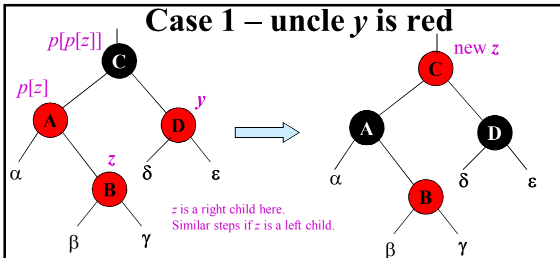
```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

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

Insertion into a red-black tree

- There are **three Case:**

Case 1 – uncle *y* is red

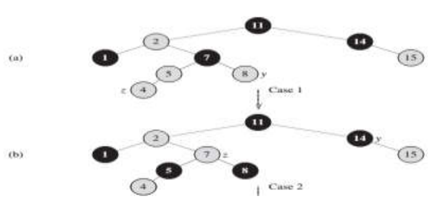


- $p[p[z]]$ (z's grandparent) must be black, since z and $p[z]$ are both red and there are no other violations of property 4.
- Make $p[z]$ and y black $\Longrightarrow$ now z and $p[z]$ are not both red. But property 5 might now be violated.
- Make $p[p[z]]$ red $\Longrightarrow$ restores property 5.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

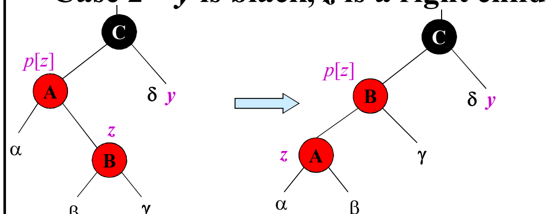
Insertion into a red-black tree

- **Case 1**

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                  // case 1
            y.color = BLACK                    // case 1
            z.p.p.color = RED                  // case 1
            z = z.p.p                          // case 1
```



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

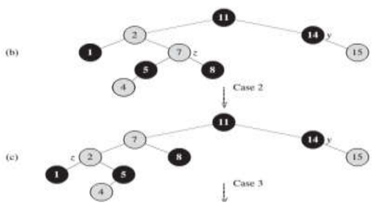


- **Left rotate around $p[z]$,** $p[z]$ and z switch roles $\Longrightarrow$ now z is a left child, and both z and $p[z]$ are red.
- Takes us immediately to case 3.

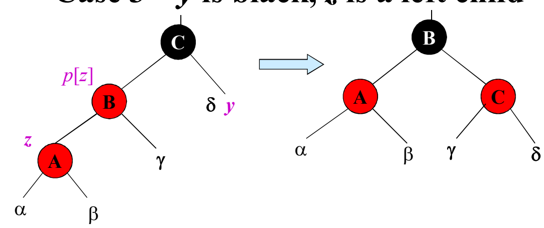
Insertion into a red-black tree

- **Case 2**

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK              // case 1
            y.color = BLACK                // case 1
            z.p.p.color = RED              // case 1
            z = z.p.p                      // case 1
        else if z == z.p.right
                z = z.p                    // case 2
                LEFT-ROTATE(T, z)          // case 2
```



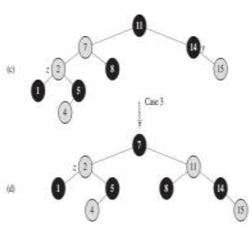
Case 3 – *y* is black, *z* is a left child



- 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.

Insertion into a red-black tree

- **Case 3**




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

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.*
  - » Note: ***there are at most 2 rotations overall.***

<u>Exercise</u>

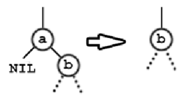
- Show the red-black trees that result after successively inserting the keys 41; 38; 31; 12; 19; 8 into an initially empty red-black tree.
-

red-black tree :Deletion

- Deleting a node from a red-black tree is a bit more complicated than inserting a node.

red-black tree :Deletion

- Use usual BST deletion algorithm.
- Let deleting node **a** (disregard colors, fix later)
- **Case 1: a has no left child**

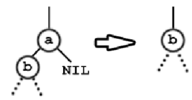


- Remove a and put its right child (b) instead
- **Note:**
  - **if the red rule is now broken b and its new father** (originally a's father),
  - we can color node **b** in black; keeping the black height balance , since a was definitely black (as its father is red)

red-black tree :Deletion cont..

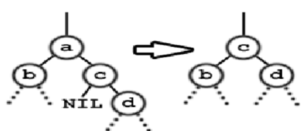
- **Case 2: a has no right child**:
- Remove **a** and put its left child b in its place

Note: same as case 1



red-black tree :Deletion cont..

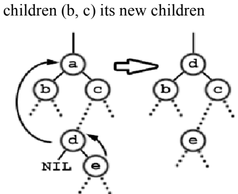
- **Case 3: a has two children, a's successor (c) is its right child**:
  - » Remove a and put its successor (c) in its place
  - » Make a's left child (b) the successor's (c) left child



- **Note:**
  - successor node always has no left child
  - Moving the successor node , we color it in a's color.
  - **If the successor was black**, the child that replaced it (d) is colored in **"extra" black**, **making it red-black or black-black**. This is fixed in the correction.

red-black tree :Deletion cont..

- **Case 4 : a has two children, a's successor (d) is not its child**:
  - » Put the successor's (**d**) left child (e) instead of it
  - » Remove **a** and put its successor (**d**) instead of it,
  - » making a's children (b, c) its new children



Notes: same as case 3

red-black tree :Deletion

- When we want to delete node z and **z has fewer than two children**, then z is removed from the tree, and then assign z to the y.
- When **z has two children**, then y should be z's successor, and y moves into z's position in the tree.
  - » remember y's color before it is removed from or moved within the tree, and
  - » keep track of the node x that moves into y's original position in the tree, because node x might also cause violations of the red-black properties.
- After deleting node z , RB-DELETE calls an auxiliary procedure **RB-DELETE-FIXUP**, which changes colors and performs rotations to restore the red-black properties.

red-black tree :Deletion

```
RB-DELETE(T, z)
  y = z
  y-original-color = y.color
  if z.left == T.nil
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
  elseif z.right == T.nil
      x = z.left
      RB-TRANSPLANT(T, z, z.left)
  else y = TREE-MINIMUM(z.right)
      y-original-color = y.color
      x = y.right
      if y.p == z
          x.p = y
      else RB-TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      RB-TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
      y.color = z.color
  if y-original-color == BLACK
      RB-DELETE-FIXUP(T, x)
```

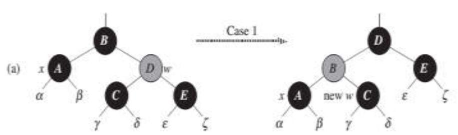
Correction after Deletion in RB-Tree

RB-DELETE-FIXUP

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                   // case 1
            x.p.color = RED                                   // case 1
            LEFT-ROTATE(T, x.p)                               // case 1
            w = x.p.right                                     // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                     // case 2
            x = x.p                                           // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                          // case 3
                w.color = RED                                 // case 3
                RIGHT-ROTATE(T, w)                            // case 3
                w = x.p.right                                 // case 3
            w.color = x.p.color                               // case 4
            x.p.color = BLACK                                 // case 4
            w.right.color = BLACK                             // case 4
            LEFT-ROTATE(T, x.p)                               // case 4
            x = T.root                                        // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

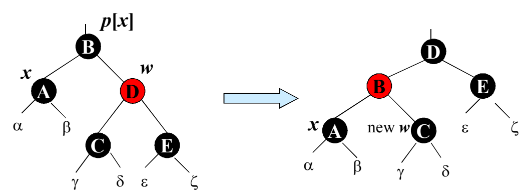
RB-DELETE-FIXUP : <u>case-1</u>

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                   // case 1
            x.p.color = RED                                   // case 1
            LEFT-ROTATE(T, x.p)                               // case 1
            w = x.p.right                                     // case 1
```



Case 1: x's sibling w is red

Case 1 – *w* is red

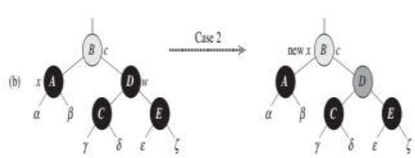


- *w* must have black children.
- Make *w* black and *p*[*x*] red (because *w* is red *p*[*x*] couldn't have been red).
- Then left rotate on *p*[*x*].
- New sibling of *x* was a child of *w* before rotation ⇒ must be black.
- **Go immediately to** case 2, 3, or 4.

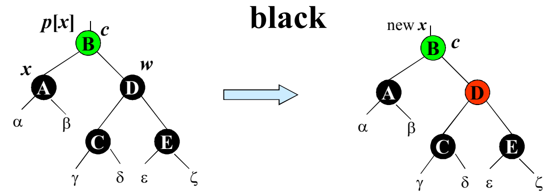
RB-DELETE-FIXUP : case-2

Case 2: x's sibling w is black, and both of w's children are black

```
if w.left.color == BLACK and w.right.color == BLACK
    w.color = RED                                             // case 2
    x = x.p                                                   // case 2
```



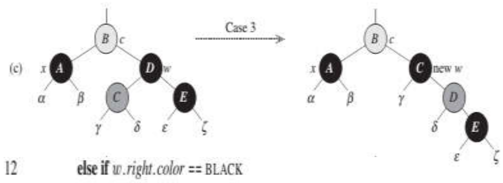
Case 2 – *w* is black, both *w*'s children are black



- Take 1 black off *x* (⇒ singly black) and off *w* (⇒ 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 ⇒ new *x* is red & black ⇒ color attribute of new *x* is RED ⇒ loop terminates. Then new *x* is made black in the last line.

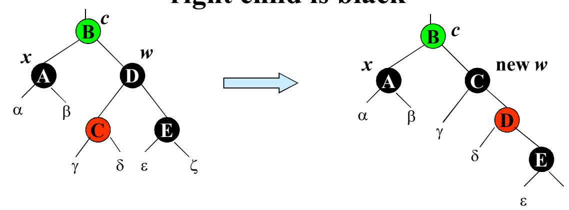
RB-DELETE-FIXUP : case-3

- **Case 3:** x's sibling w is black, w's left child is red, and w's right child is black




```
else if w.right.color == BLACK
        w.left.color = BLACK                // case 3
        w.color = RED                       // case 3
        RIGHT-ROTATE(T, w)                  // case 3
        w = x.p.right                       // case 3
```

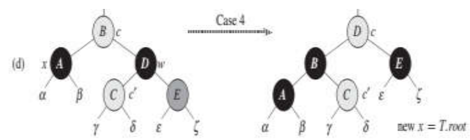
Case 3 – *w* is black, *w*'s left child is red, *w*'s right child is black



- 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 ⇒ case 4.

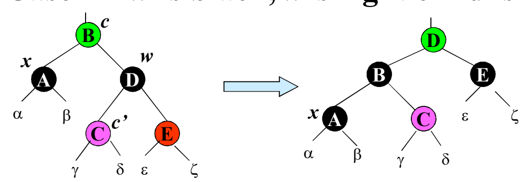
RB-DELETE-FIXUP : case-4

- **Case 4:** x's sibling w is black, and w's right child is red




```
        w.color = x.p.color                 // case 4
        x.p.color = BLACK                   // case 4
        w.right.color = BLACK               // case 4
        LEFT-ROTATE(T, x.p)                 // case 4
        x = T.root                          // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

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* (⇒ *x* is now singly black) without violating any red-black properties.
- All done. Setting *x* to root causes the loop to terminate.

Analysis

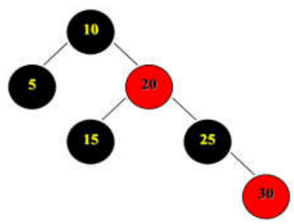
- $O(\lg n)$ time to get through RB-Delete up to the call of RB-Delete-Fixup.
- Within RB-Delete-Fixup:
  - » Case 2 is the only case in which more iterations occur.
    - *x* moves up 1 level.
    - Hence, $O(\lg n)$ iterations.
  - » Each of cases 1, 3, and 4 has 1 rotation ⇒ ≤ 3 rotations in all.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

Exercises-1

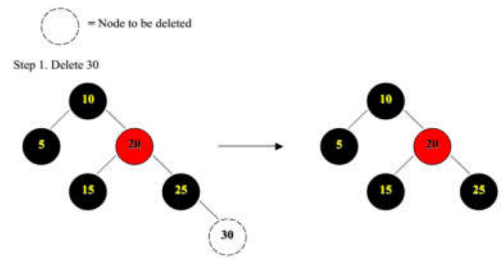
- Suppose that a node x is inserted into a red-black tree with RB-INSERT an then immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

Exercises-2

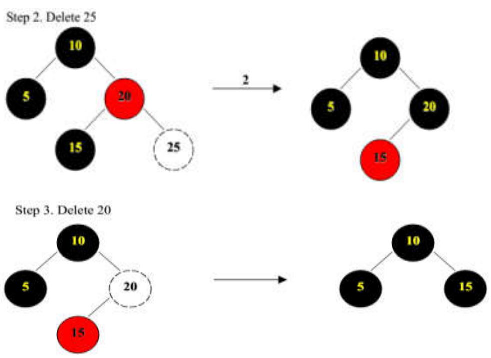
- Show the red-black tree that results from successively deleting the keys 30, 25, 20, 15, 10, and 5 from following RB-Tree.



Exercises-2: solution



Exercises-2: solution



Exercises-3: solution