Red-Black Trees

Balanced Insertions and Deletions
Operations in O(lg n)

Red-Black Trees

- Def.: Red-Black Tree

A ***red-black tree*** is a binary search tree + 1 bit per node: an attribute $color$, which is either red or black.

All leaves are empty (nil) and colored black.

- We use a single sentinel, $nil[T]$, for all the leaves of red-black tree T.
- $color[nil[T]]$ is black.
- The root's parent is also $nil[T]$.

All other attributes of binary search trees are inherited by red-black trees (key, $left$, $right$, and p). We don't care about the key in $nil[T]$.

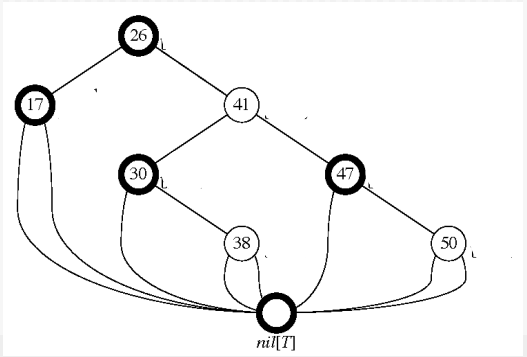
Red-Black Trees

Red-black properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf ($nil[T]$) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Red-Black Trees

- Example:

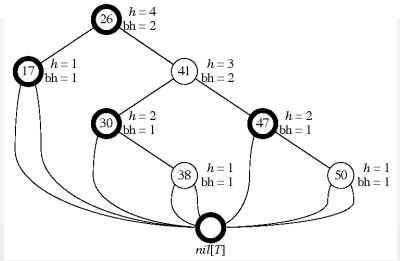


Red-Black Trees

- Height

Height of a red-black tree

- ***Height of a node*** is the number of edges in a longest path to a leaf.
- ***Black-height*** of a node x: $\mathrm{bh}(x)$ is the number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x. By property 5, black-height is well defined.



Red-Black Trees

- Two sub-lemmata

Claim

Any node with height h has black-height $\geq h/2$.

Proof By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ are black. ■ (claim)

Claim

The subtree rooted at any node x contains $\geq 2^{\mathrm{bh}(x)} - 1$ internal nodes.

Proof By induction on height of x.

Basis: Height of $x = 0 \Rightarrow x$ is a leaf $\Rightarrow \mathrm{bh}(x) = 0$. The subtree rooted at x has 0 internal nodes. $2^0 - 1 = 0$.

Inductive step: Let the height of x be h and $\mathrm{bh}(x) = b$. Any child of x has height $h - 1$ and black-height either b (if the child is red) or $b - 1$ (if the child is black). By the inductive hypothesis, each child has $\geq 2^{\mathrm{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains $\geq 2 \cdot (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ internal nodes. (The $+1$ is for x itself.) ■ (claim)

Red-Black Trees

- Another one

Lemma

A red-black tree with n internal nodes has height $\leq 2\lg(n+1)$.

Proof Let h and b be the height and black-height of the root, respectively. By the above two claims,

$$n \geq 2^b - 1 \geq 2^{h/2} - 1 .$$

Adding 1 to both sides and then taking logs gives $\lg(n+1) \geq h/2$, which implies that $h \leq 2\lg(n+1)$. ■ (theorem)

Red-Black Trees

- Insertion & Deletion

If we insert, what color to make the new node?

- Red? Might violate property 4.
- Black? Might violate property 5.

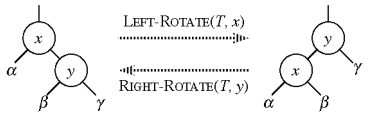
If we delete, thus removing a node, what color was the node that was removed?

- Red? OK, since we won't have changed any black-heights, nor will we have created two red nodes in a row. Also, cannot cause a violation of property 2, since if the removed node was red, it could not have been the root.
- Black? Could cause there to be two reds in a row (violating property 4), and can also cause a violation of property 5. Could also cause a violation of property 2, if the removed node was the root and its child—which becomes the new root—was red.

Red-Black Trees

- Rotations

- Won't upset the binary-search-tree property.
- Have both left rotation and right rotation. They are inverses of each other.
- A rotation takes a red-black-tree and a node within the tree.



Red-Black Trees

- The Rotation Algorithm in 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
```

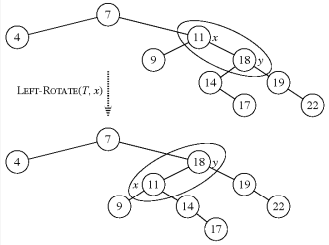
The pseudocode for LEFT-ROTATE assumes that

- $right[x] \neq nil[T]$, and
- root's parent is $nil[T]$.

Pseudocode for RIGHT-ROTATE is symmetric: exchange *left* and *right* everywhere.

Red-Black Trees

- Rotation Example:



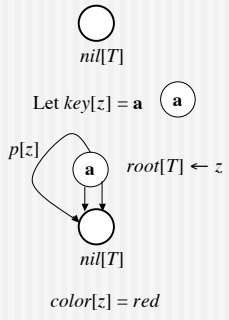
- Before rotation: keys of x's left subtree $\leq 11 \leq$ keys of y's left subtree $\leq 18 \leq$ keys of y's right subtree.
- Rotation makes y's left subtree into x's right subtree.
- After rotation: keys of x's left subtree $\leq 11 \leq$ keys of x's right subtree $\leq 18 \leq$ keys of y's right subtree.

Time: $O(1)$ for both LEFT-ROTATE and RIGHT-ROTATE, since a constant number of pointers are modified.

Red-Black Trees

- Insertion Pseudo-Code

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



Red-Black Trees

- Insertion

- RB-INSERT ends by coloring the new node z red.
- Then it calls RB-INSERT-FIXUP because we could have violated a red-black property.

Which property might be violated?

1. OK.
2. If z is the root, then there's a violation. Otherwise, OK.
3. OK.
4. If $p[z]$ is red, there's a violation: both z and $p[z]$ are red.
5. OK.

Remove the violation by calling RB-INSERT-FIXUP:

Red-Black Trees

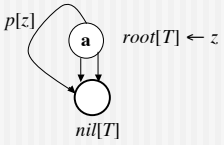
- Fix the Problems

```
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
```

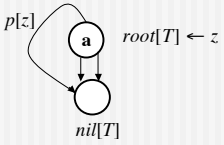

Red-Black Trees

- Fix the Problems

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



$color[z] = red$

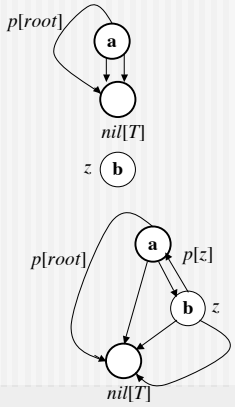


$color[z] = black$

Red-Black Trees

- Insert another

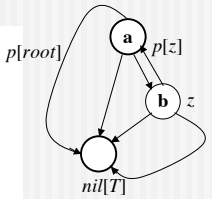
```
RB-INSERT(T, z)
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
    do y ← x
       if key[z] < key[x]
          then x ← left[x]
          else x ← right[x]
p[z] ← y
if y = nil[T]
   then root[T] ← z
   else if key[z] < key[y]
           then left[y] ← z
           else right[y] ← z
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
RB-INSERT-FIXUP(T, z)
```



Red-Back Trees

- Fix it

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

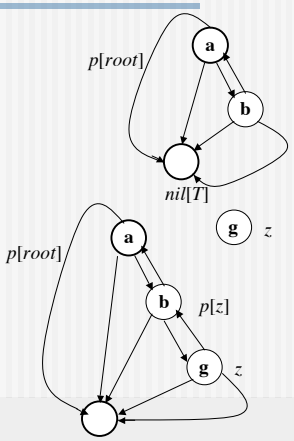


Nothing to do - root already black

Red-Black Trees

- Insert another

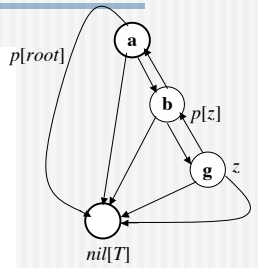
```
RB-INSERT(T, z)
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
    do y ← x
       if key[z] < key[x]
          then x ← left[x]
          else x ← right[x]
p[z] ← y
if y = nil[T]
   then root[T] ← z
   else if key[z] < key[y]
          then left[y] ← z
          else right[y] ← z
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
RB-INSERT-FIXUP(T, z)
```



Red-Black Trees

- And fix it

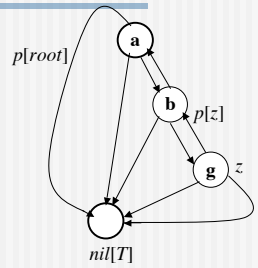
```
RB-INSERT-FIXUP(T, z)
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
          then y ← right[p[p[z]]]
               if color[y] = RED
                  then color[p[z]] ← BLACK
                       color[y] ← BLACK
                       color[p[p[z]]] ← RED
                       z ←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



Red-Black Trees

- And fix it

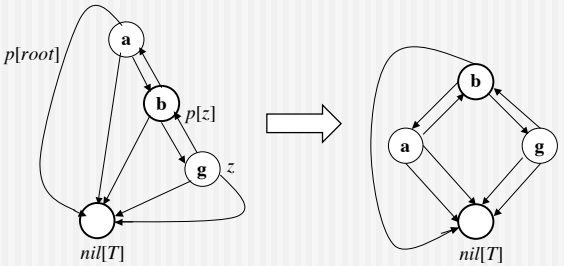
```
else /* if p[z] = right[p[p[z]] then */
     y ← left[p[p[z]]]
     if color[y] = RED
       then color[p[z]] ← BLACK
            color[y] ← BLACK
            color[p[p[z]]] ← RED
            z ← p[p[z]]
       else if z = left[p[z]]
              then z ← p[z]
                   Right-Rotate(T, z)
            color[p[z]] ← BLACK
            color[p[p[z]] ← RED
            Left-Rotate(T, p[p[z]])

color[root[T]] ← BLACK
```



Note: y is BLACK and z is NOT a left child of its parent, so we color $p[z]$ BLACK, $p[p[z]]$ RED and Left-Rotate on $p[p[z]]$ and finish by coloring $p[z]$ (the new root) BLACK.

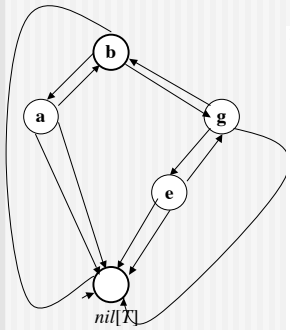
Red-Black Trees

- Here is the sequence: color $p[z]$ BLACK, $p[p[z]]$ RED and Left-Rotate on $p[p[z]]$ and finish by coloring $p[z]$ (the new root) BLACK.



Red-Black Trees

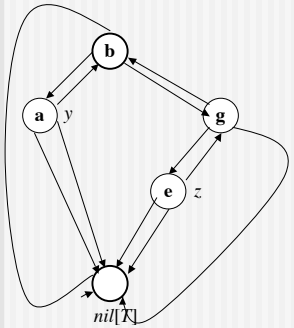
- Insert e. The problem, at this point, is that the number of black nodes along each path must change. Look back at the code: does this apply?




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


Red-Black Trees

- Look back at the code: does this apply?

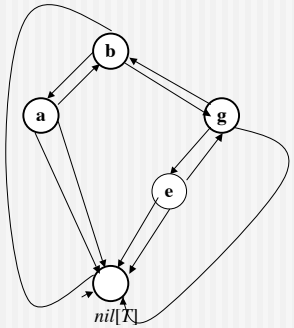



```
else /* if p[z] = right[p[p[z]] then */
     y ← left[p[p[z]]]
     if color[y] = RED
       then color[p[z]] ← BLACK
            color[y] ← BLACK
            color[p[p[z]]] ← RED
            z ← p[p[z]]
       else if z = left[p[z]]
              then z ← p[z]
                   Right-Rotate(T, z)
            color[p[z]] ← BLACK
            color[p[p[z]] ← RED
            Left-Rotate(T, p[p[z]])
color[root[T]] ← BLACK
```

Since y is RED, set $color[p[z]]$ to BLACK, $color[y]$ to BLACK, $color\ p[p[z]]$ to RED, z to $p[p[z]]$; since now $color[p[z]] = color[p[root[T]] =$ BLACK, the while loop ends. Set $z = root[T]$ to BLACK.

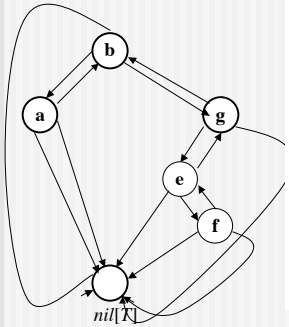
Red-Black Trees

- And the tree looks like:



Red-Black Trees

- Add f: Do we use the left or right code?We start using the left code.




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


Red-Black Trees

- Why does this work?

Loop invariant:

At the start of each iteration of the **while** loop,

a. z is red.

b. There is at most one red-black violation:

- Property 2: z is a red root, or
- Property 4: z and $p[z]$ are both red.

Red-Black Trees

- The Induction

Initialization: We've already seen why the loop invariant holds initially.

Termination: The loop terminates because $p[z]$ is black. Hence, property 4 is OK. Only property 2 might be violated, and the last line fixes it.

Maintenance: We drop out when z is the root (since then $p[z]$ is the sentinel $nil[T]$, which is black). When we start the loop body, the only violation is of property 4.

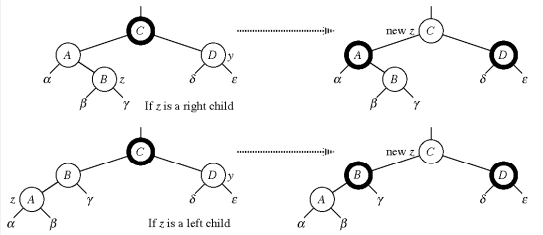
There are 6 cases, 3 of which are symmetric to the other 3. The cases are not mutually exclusive. We'll consider cases in which $p[z]$ is a left child.

Let y be z's uncle ($p[z]$'s sibling).

Red-Black Trees

- Case 1

Case 1: y is red

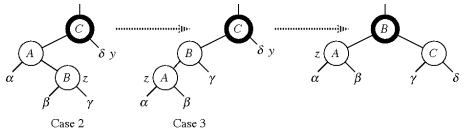


- $p[p[z]]$ (z's grandparent) must be black, since z and $p[z]$ are both red and there are no other violations of property 4.
- Make $p[z]$ and y black ⇒ now z and $p[z]$ are not both red. But property 5 might now be violated.
- Make $p[p[z]]$ red ⇒ restores property 5.
- The next iteration has $p[p[z]]$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

Red-Black Trees

- Case 2

Case 2: y is black, z is a right child



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

Red-Black Trees

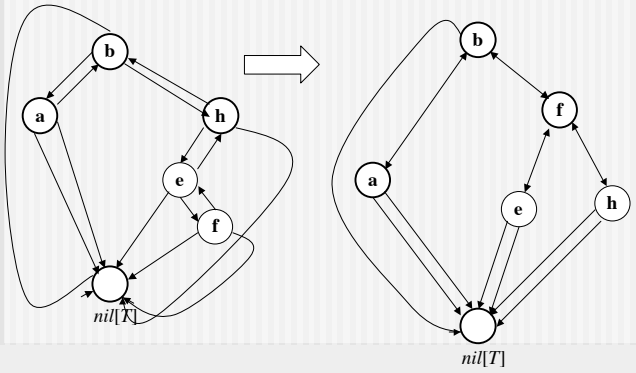
- Case 3

Case 3: y is black, z is a left child

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

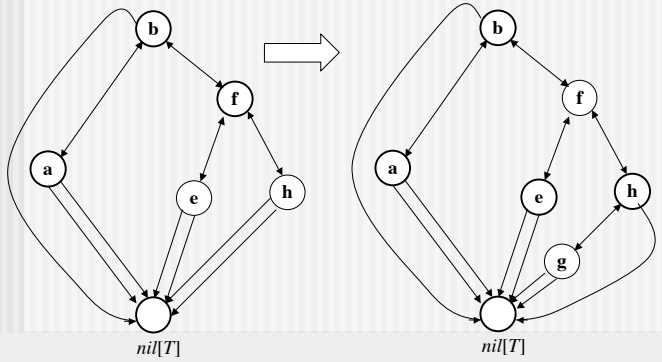
Red-Black Trees

- Finish the last insertion (modified).



Red-Black Trees

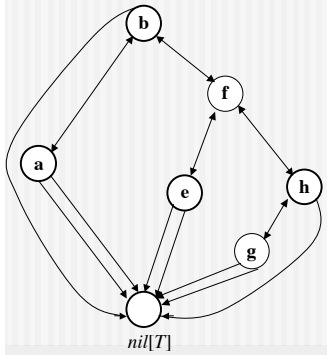
- Now add **g**:



Red-Black Trees

- Now for the deletion: delete **g**.

```
RB-DELETE(T, z)
if left[z] = nil[T] or right[z] = nil[T]
   then y ← z
   else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T]
   then x ← left[y]
   else x ← right[y]
p[x] ← p[y]
if p[y] = nil[T]
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
if color[y] = BLACK
   then RB-DELETE-FIXUP(T, x)
return y
```



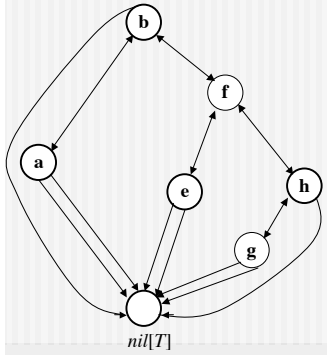
Red-Black Trees

- Chasing the code with the picture (or vice-versa, your pick), we end up just removing the node labeled **g**. Nothing else changes, other than the left pointer out of the node labeled **h**, which goes to *nil*[T]. **g** just disappears without triggering any adjustments: it was a RED node, so the number of BLACK nodes along that path did not change. Notice that no labels or other data are copied, since the node we are deleting is the very last in a chain - the next nodes are the sentinel leaves.
- Deleting any other node will trigger more complicated readjustments.

Red-Black Trees

- Remove **f**:

```
RB-DELETE(T, z)
if left[z] = nil[T] or right[z] = nil[T]
   then y ← z
   else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T]
   then x ← left[y]
   else x ← right[y]
p[x] ← p[y]
if p[y] = nil[T]
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
if color[y] = BLACK
   then RB-DELETE-FIXUP(T, x)
return y
```

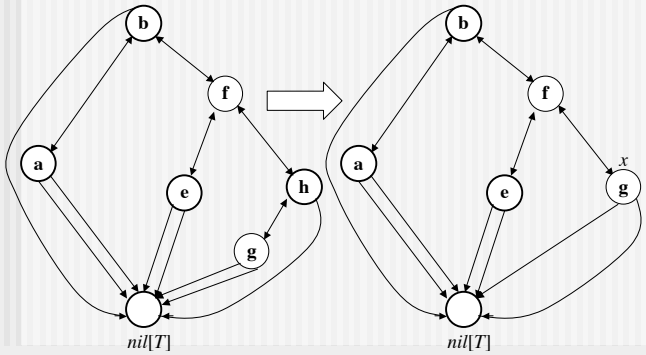


Red-Black Trees

- This is also easy, since removing **f** involves finding its successor (**g**), re-attaching the parent of **g** to a sentinel (rather than **g**), and copying the contents of **g** into **f**. Since the node actually removed (**g**) is RED, nothing needs to be done.
- We now try to remove h - this is a BLACK node and, because it has only one child, it will actually be removed. This will finally trigger RB-Delete-Fixup(T, *x*).

Red-Black Trees

- Before the call to RB-Delete-Fixup(T, x) We have:



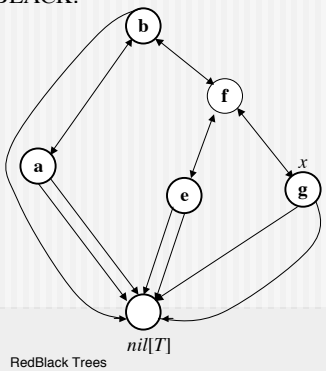
Red-Black Trees

- Here is the pseudo-code

```
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
               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
```


Red-Black Trees

- Since x is neither the root, nor is it BLACK (it took the place of a BLACK node that was removed), the code tells us to just color it BLACK.

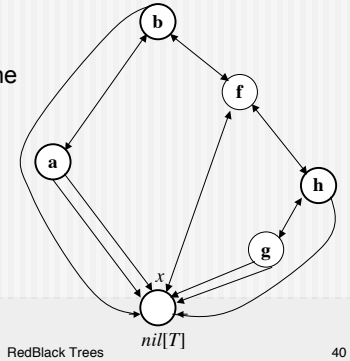


Red-Black Trees

- How about removing **e** from the original tree? Since the deletion itself does not worry about color, we just remove **e**, and x is the nil[T] sentinel.

Note that the parent of the sentinel is now **f**.

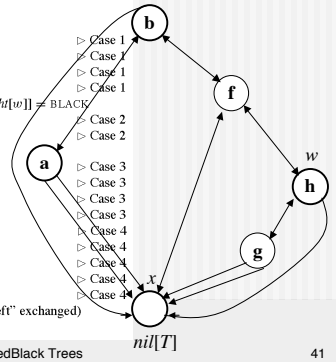
Furthermore, x is not the root, and its color is BLACK.



Red-Black Trees

- Compare the code and the tree:

```
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
            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
```



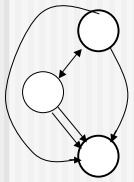
Red-Black Trees

- We have a number of cases to take care of - 8 (precisely but not mutually exclusive). We will look at 4 of them, leaving the 4 symmetric ones as exercises.
- We observe that splicing out ANY RED node requires no fix-up: it does not alter the black height of any node; it does not introduce any pair of parent-child RED nodes; and it does not change the root.
- The only case where fixing up will be needed is when the spliced out node is BLACK: that will alter the black height of its ancestors, and may violate the requirement that there be no pair of parent-child RED nodes.

Red-Black Trees

How can the splicing out of a black node y effect the result?

1. y had been the root and a red child of y becomes (physically) the new root, violating property 2. This can occur only if we have the configuration on the left, or its symmetric counterpart.



The fix is to just color the new root BLACK, but we will examine it in the context of RB-Delete-Fixup.

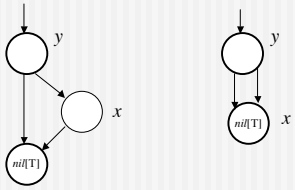
Red-Black Trees

2. Both x and $p[x] = p[y]$ are RED. Property 4 is violated (red has only black children).
3. If y was BLACK, its removal from any path will cause any path that contained it to have one fewer BLACK nodes. Property 5 (all paths from node down have same number of black nodes) is violated by any ancestor of y in the tree.

How do we solve the problem? pretend that the node x has an extra BLACK. Then all is well (properties 4 & 5), and we have to figure out where to unload this extra black…

Red-Black Trees

- Note that, in RB-Delete-Fixup(T, x), with y the node actually deleted, x is
  a. y's sole non-sentinel child before y was spliced out
  b. the sentinel itself, if y had no children.



Also, after deletion, $p[x] = p[y]$

Red-Black Trees

With y black, we could violate the properties:

1. Every node is either red or black. NO
2. The root is black: YES, if y is the root and x is red.
3. Every leaf is black. NO
4. If a node is red then both of its children are black: YES, if $p[y]$ and x are both red (from the left hand example).
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes: YES, any path that contained y now has one fewer black nodes.

We can fix 5 by giving x "an extra black" from its **deleted parent** - then the count of black nodes if "fixed": we will push the extra black around until we can safely unload it…

Red-Black Trees

Note: we have violated Property 1, since we now have nodes that are neither red nor black.: x is doubly-black if x was black; it is red&black if it was red. Note that $color[x]$ is still just RED or BLACK - the extra black comes from **pointing** to it (which means we have to unload its extra blackness **before** we stop pointing to it).

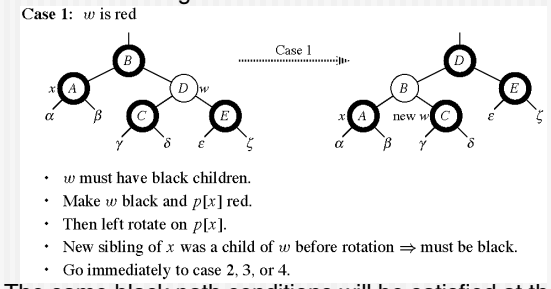
IDEA: move the extra black up the tree until:

- x points to a red&black node --> turn it into a red one
- x points to the root --> just remove the extra black
- perform rotations and recolorings

The first two points tell you when it's safe to unload the extra black; the last one tells you how to move it up. That's where we go now.

Red-Black Trees

We start by assuming x is the left child of its parent and that w is its sibling:



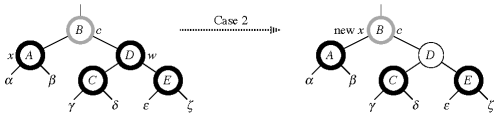
- w must have black children.
- Make w black and $p[x]$ red.
- Then left rotate on $p[x]$.
- New sibling of x was a child of w before rotation ⇒ must be black.
- Go immediately to case 2, 3, or 4.

The same black path conditions will be satisfied at the end of the recoloring and rotation as at the beginning.

Red-Black Trees

The tree rooted at A has one more black node than the trees rooted at B and C. We don't know the color of the children of C - assume they are both black (other cases later): A, B, D are, respectively, the old A, B, C.

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



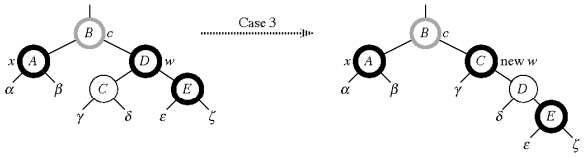
[Node with gray outline is of unknown color, denoted by c.]

- 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.

Red-Black Trees

Next case: doesn't quite end, since we can't yet get rid of the extra black on A. You just move to the case where the colors of the children of w are swapped (γ is black)

Case 3: w is black, w's left child is red, and w's right child is black

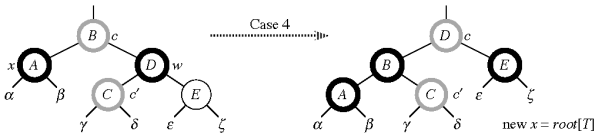


- Make w red and w's left child black.
- Then right rotate on w.
- New sibling w of x is black with a red right child $\Rightarrow$ case 4.

Red-Black Trees

Finally:

Case 4: w is black, w's left child is black, and w's right child is red



[Now there are two nodes of unknown colors, denoted by c and c'.]

- 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.

Red-Black Trees

Red-Black Trees