Red-Black Trees

Balanced Insertions and Deletions
Operations in $O(\lg n)$

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

- All leaves are empty (*nil*) and colored *BLACK*.
- We use a single sentinel, $T.nil$, for all the leaves of a red-black tree $T$.
- $T.nil.color$ is black.
- The root’s parent is also $T.nil$.

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 $T.nil$. 

Red-Black Trees

Red-Black Properties

1. Every node is either red or black
2. The root is black
3. Every leaf (T.nil) 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

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

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. $\blacksquare$

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

Proof By induction on height of $x$.

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

Inductive step: Let the height of $x$ be $h$ and $bh(x)$ = $b$. 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 $\geq 2^{bh(x)} - 1$ internal nodes. Thus, the subtree rooted at $x$ contains $\geq 2 \cdot (2^{bh(x)} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes. (The +1 is for $x$ itself.) $\blacksquare$
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,
\[ n \geq 2^h - 1 \geq 2^{h/2} - 1. \]
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?
- 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: managing change
  - 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.

These will be the techniques used to handle “tree surgery” required to maintain all the properties after an insertion or deletion.

The Rotation Algorithm in Pseudo-Code

```
\text{LEFT-ROTATE}(T, x)
\begin{align*}
& y = x.\text{right} & \text{// set } y \\
& x.\text{right} = y.\text{left} & \text{// turn } y \text{’s left subtree into } x \text{’s right subtree} \\
& \text{if } y.\text{left} \neq T.\text{nil} & \\
& y.\text{left}.p = x & \text{// link } x \text{’s parent to } y \\
& y.p = x.p & \\
& \text{if } x.p = T.\text{nil} & \\
& T.root = y & \text{// link } x \text{’s parent to } y \\
& \text{else if } x = x.p.\text{left} & \\
& x.p.\text{left} = y & \text{// put } x \text{ on } y \text{’s left} \\
& \text{else } x.p.\text{right} = y & \\
& y.\text{left} = x & \text{// put } x \text{ on } y \text{’s left} \\
& x.p = y &
\end{align*}
\text{RIGHT-ROTATE}(T, x)
```

The pseudocode for \text{LEFT-ROTATE} assumes
- $x.\text{right} \neq T.\text{nil}$
- root’s parent is $T.\text{nil}$

\text{RIGHT-ROTATE} is symmetric: exchange left and right.
Red-Black Trees

Rotation Example:

Before rotation: keys of z’s left subtree ≤ 11 ≤ keys of y’s left subtree ≤ 18 ≤ keys of y’s right subtree.

Rotation makes y’s left subtree into z’s right subtree.

After rotation: keys of z’s left subtree ≤ 11 ≤ keys of z’s right subtree ≤ 18 ≤ keys of y’s right subtree.

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

Insertion

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

What property may be violated?

1. Every node is either red or black: OK
2. The root is black: if z is the root, then there is a violation. Otherwise, OK. – NOT OK.
3. Every leaf (T.nil) is black: OK
4. If a node is red then both its children are black: If z is red, there is a violation: are both z and z.p red? No: OK
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes: OK

Remove the violation by calling RB-INSERT-FIXUP.

```java
Red-Black Trees

Fix the Problems

RB-INSERT-FIXUP(T, z)
1    while z.p.color == RED
2        if z.p == z.p.p.left
3            y = z.p.p.right
4            if y.color == RED
5                z.p.color = BLACK  // case 1
6                y.color = BLACK  // case 1
7                z.p.p.color = RED  // case 1
8            z = z.p
9        else if z == z.p.right
10                z = z.p
11            z.p.color = BLACK  // case 2
12            LEFT-ROTATE(T, z)  // case 2
13        else (same as then clause
14            with “right” and “left” exchanged)
15            z.p.color = RED  // case 3
16            RIGHT-ROTATE(T, z.p.p)  // case 3
17    RIGHT-ROTATE(T, z.p)
18            z.p.color = BLACK
```
Red-Black Trees

- **Fix the Problems**

  ```
  RB-INSERT-FIXUP(T, z)
  1 while z.p.color == RED
  2     if z.p == z.p.p.left
  3         y = z.p.p.right
  4         if y.color == RED
  5             z.p.color = BLACK
  6             y.color = BLACK
  7             z.p.p.color = RED
  8             z = z.p.p
  9         else if z == z.p.right
 10             z = z.p
 11             LEFT-ROTATE(T, z)
 12             z.p.color = BLACK
 13             z.p.p.color = RED
 14             RIGHT-ROTATE(T, z, z.p.p)
 15         else (same as then clause
 16             with “right” and “left” exchanged)
 17     z.root.color = BLACK
  ```

Red-Black Trees

- **Insert another**

  ```
  RB-INSERT(T, z)
  1 y = T.nil
  2 x = T.root
  3 while x ≠ 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)
  ```
Red-Back Trees

- **Fix it**

  ```java
  RB-INSERT-FIXUP(T, z)
  while z.color == RED
    if z.p == z.p.p.left
      y = z.p.p.right
      if y.color == RED
        z.p.color = BLACK
      y.color = BLACK
      z.p.p.color = RED
      z = z.p.p
    else if z == z.p.p.right
      z = z.p
    LEFT-ROTATE(T, z)
  RIGHT-ROTATE(T, z.p.p)
  else (same as then clause
    with “right” and “left” exchanged)
  T.root.color = BLACK
  Nothing to do - root already black
  ```

Red-Black Trees

- **Insert another**

  ```java
  RB-INSERT(T, z)
  y = T.nil
  x = T.root
  while x != T.nil
    if x.key < y.key
      x = x.left
    else
      y = x
      x = x.right
  y.p = y
  if y == T.nil
    T.root = z
  else if z.key < y.key
    y.left = z
    z.p = y
  else
    y.right = z
    z.p = y
  z.left = T.nil
  z.right = T.nil
  z.color = RED
  RB-INSERT-FIXUP(T, z)
  ```
Red-Black Trees

And fix it

```
RB-INSERT-FIXUP(T, z)
1   while z.p.color == RED
2     y = z.p.right
3       if y.color == RED
4         z.p.color = BLACK
5         y.color = RED
6         z.p.p.color = RED
7         z = z.p.p
8     else if z == z.p.left
9         z = z.p
10        RIGHT-ROTATE(T, z)
11        z.p.color = BLACK
12        z.p.p.color = RED
13        LEFT-ROTATE(T, z.p.p)
14        T.root.color = BLACK
```

Note: y is BLACK and z is NOT a left child of its parent, so we color z.p BLACK, z.p.p RED and LEFT-ROTATE on z.p.p and finish by coloring z.p (the new root) BLACK.
Red-Black Trees

- Here is the sequence: color \( z.p \) BLACK, \( z.p.p \) RED and LEFT-ROTATE on \( z.p.p \) and finish by coloring \( z.p \) (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?

```python
def RB_INSERT_FIXUP(T, z):
    while z.p.color == RED:
        if y == z.p.right:
            if y.color == RED:
                z.p.color = BLACK
                y.color = RED
                z.p.p.color = RED
            else:
                if z == z.p.right:
                    z = z.p
                else:
                    z = z.p
                    LEFT-ROTATE(T, z)
                z.p.color = BLACK
                T.root.color = BLACK
        T.nil
```
Red-Black Trees

- Look back at the code: does this apply?

```java
else if (y == z.p.right)
    if (y.color == RED)
        z.p.color = BLACK
        y.color = BLACK
        z.p.p.color = RED
        z = z.p.p
    else if (z.p.right == T.nil)
        z.p.right = z
        RIGHT-ROTATE(T, z)
    z.p.color = BLACK
    z.p.p.color = RED
    LEFT-ROTATE(T, z.p.p)
    Troot.color = BLACK
```

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

And the tree looks like:
Add f: Do we use the left or right code? We start using the left code.

Red-Black Trees

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

Since z.p == z.p.p.left, y.color is black and z == z.p.right, we move z up and call LEFT-ROTATE.
Red-Black Trees

Now we color and then **RIGHT-ROTATE**

```plaintext
RB-INSERT-FIXUP(T, z)
1   while z.p.color == RED
2      y = z.p.right
3      if y.color == RED
4         z.p.color = BLACK
5         y.color = BLACK
6         z.p.p.color = RED
7         z = z.p
8         else if z == z.p.right
9               else if z == z.p.p
10                  T.nil
11         LEFT-ROTATE(T, z)
12         z.p.color = BLACK
13         z.p.p.color = RED
14         RIGHT-ROTATE(T, z.p.p)
15         else (same as then clause
16                 with "right" and "left" exchanged)
17         T.root.color = BLACK
```

Red-Black Trees

**RIGHT-ROTATE** and Stop, since z.p is NOT red.
Why does this work?

We have a loop invariant: at the start of each iteration of the while loop,

- \( z \) is red
- If \( z.p \) is the root, then \( z.p \) is black
- There is at most one red-black violation:
  1. Property 2: \( z \) is a red root, or
  2. Property 4: \( z \) and \( z.p \) are both red

Initialization

We start with a red-black tree with no violations. We add a red node \( z \). We show each part of the invariant holds when \text{RB-INSERT-FIXUP} is called.

- When \text{RB-INSERT-FIXUP} is called, \( z \) is the red node that was added.
- If \( z.p \) is the root, it started out black and did not change prior to the call to \text{RB-INSERT-FIXUP}.
- Properties 1, 3, 5 hold prior to the call to \text{RB-INSERT-FIXUP}.
  - If the tree violates Prop. 2, then the red root is the newly added node \( z \), which must be the only internal node in the tree. Because the parent and both children of \( z \) are the sentinel (black), the tree does not violate Prop. 4.
  - If the tree violates Prop. 4, since the children of \( z \) are black sentinels and the tree has no other violations prior to \( z \) being added, the violation must be because both \( z \) and \( p.z \) are red. No other red-black properties are violated.
Termination
The loop terminates because $z.p$ is black. If $z$ is the root, then $z.p$ is the sentinel (black). The tree does not violate Prop. 4 (If a node is red then both its children are black.) at loop termination. By the loop invariant, the only property that might fail to hold is Prop. 2 (The root is black). Line 16 ($t.root.color = \text{BLACK}$) restores this property so that at termination of RB-INSERT-FIXUP, all red-black properties hold.

Maintenance
We need to consider 6 cases in the while loop, but 3 are symmetric to the other 3, depending on whether $z.p$ is a left or right child of $z.p.p$.
- $z.p.p$ exists since part b) of the loop invariant (slide 29) states that if $z.p$ is the root it must be black.
- We enter the loop only if $z.p$ is red.
- Line 3 sets $y = z.p.p.right$
- Line 4 tests the color of $y$: if red, we go to Case 1; if black we go to Cases 2 and 3.
- In all case $z$’s grandparent $z.p.p$ is black since $z.p$ is red and Prop.4 is violated only between $z$ and $z.p$. 
Red-Black Trees

- $z.p.p$ must be black ($z$ and $z.p$ are red - no other violations of P. 4).
- Make $z.p$ and $y$ black: $z$ and $z.p$ not both red. Prop. 5 (For each node, all paths from the node to descendant leaves contain the same number of black nodes) may be violated.
- Make $z.p.p$ red. Prop. 5 now holds.
- $z.p.p$ is the new $z$. Test the while.

Note that all the indicated subtrees must have black roots.
- **LEFT-ROTATE** around $z.p$. Now $z$ is a left-child and both $z$ and $z.p$ are red.
- We go immediately to Case 3. **RIGHT-ROTATE** around $z.p.p$, resetting two colors, and you are done.
Red-Black Trees

Case 3: specifically, $y$ is black and $z$ is a left child:

- Make $z.p$ black and $z.p.p$ red
- RIGHT-ROTATE on $z.p.p$
- We no longer have 2 reds in a row
- $z.p$ is now black – stop the while.

Red-Black Trees

- Finish the last insertion (modified).
Now add $g$: 

The analysis should be rather obvious: $O(\lg n)$ for insertion, since the height of the tree is $O(\lg n)$.

Next Step: deletion.
We start working with deletion – since these are binary search trees, we have to deal with the same problems and introduce an auxiliary "transplant" procedure:

```plaintext
RB-TRANSPLANT(T, u, v)
1   if u.p == T.nil
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else u.p.right = v
6       v.p = u.p
```

This references $T.nil$ instead of $NIL$, and the assignment on the last line is unconditional: we can assign to $v.p$ even if $p$ points to the sentinel.

Now for the deletion: try deleting $g$.

```plaintext
RB-DELETE(T, z)
1   y = z
2   y-original-color = y.color
3   if z.left == T.nil
4       x = z.right
5       RB-TRANSPLANT(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-TRANSPLANT(T, z, z.left)
9   else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      elseif RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17       RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21     if y-original-color == BLACK
22     RB-DELETE-FIXUP(T, x)
```
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 \( Tnil \). \( g \) 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.

Remove \( f \):

```c
RB-DELETE(T, z)
1 y = z
2 y.original-color = y.color
3 if z.left == T.nil
4 x = z.right
5 RB-TRANSPLANT(T, z, z.right)
6 else
7 z = z.right
8 RB-TRANSPLANT(T, z, z.left)
9 else
10 x = TREE-MINIMUM(z.right)
11 y.original-color = x.color
12 if y.p == z
13 x.p = y
14 else
15 x = RB-TRANSPLANT(T, y, y.right)
16 y.right = x.right
17 y.right.p = y
18 RB-TRANSPLANT(T, z, y)
19 y.left = z.left
20 y.left.p = y
21 if y.original-color == BLACK
22 RB-DELETE-FIXUP(T, x)
```
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(\( T, x \)).

Before the call to RB-DELETE-FIXUP(\( T, x \)) We have:
Red-Black Trees

- Here is the pseudo-code

```plaintext
RB-DELETE-FIXUP(T, x)
1 while x != T.root and x.color == BLACK
2   if x == x.p.left
3     w = x.p.right
4       if w.color == RED
5         w.color = BLACK
6         x.p.color = RED
7         LEFT-ROTATE(T, x)
8       else
9         w = x.p.right
10        if w.left.color == BLACK and w.right.color == BLACK
11           w.color = RED
12           x.p.color = BLACK
13           if x == x.p.left
14              w.right.color = RED
15              RIGHT-ROTATE(T, w)
16           else
17              w.left.color = RED
18              LEFT-ROTATE(T, x)
19              w.color = RED
20       end
21     end
22   end
23 x = T.root // same as then clause with "right" and "left" exchanged
24 x.color = BLACK
```

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.
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 $T.nil$ sentinel. Note that the parent of the sentinel is now $f$.

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

Compare the code and the tree:

```cpp
RB-DELETE-FIXUP(T, x)
1 while x != T.root and x.color == BLACK
2 if x == x.p.left
3 w = x.p.right
4 if w.color == RED
5 w.color = BLACK
6 x.p.color = RED
7 LEFT-ROTATE(T, x.p)
8 w = x.p.right
9 if w.left.color == BLACK and w.right.color == BLACK
10 w.color = RED
11 x = x.p
12 else if w.right.color == BLACK
13 w.left.color = BLACK
14 w.color = RED
15 RIGHT-ROTATE(T, w)
16 w = x.p.right
17 w.color = x.p.color
18 x.p.color = BLACK
19 w.right.color = BLACK
20 LEFT-ROTATE(T, x.p)
21 x = T.root
22 else (same as then clause with “right” and “left” exchanged)
23 x.color = BLACK
```
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.

How can the splicing out of a black node $y$ affect 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 $x.p == y.p$ 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...

Note that, in RB-DELETE-FIXUP($T, x$), with $y$ the node actually deleted, $x$ is

- $y$'s sole non-sentinel child before $y$ was spliced out
- the sentinel itself, if $y$ had no children.

Also, after deletion, $x.p == y.p$
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 \( y.p \) 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…

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 \( x.color \) 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

$x$ is the left child of its parent and $w$ is its sibling:

Case 1: $w$ is red

- $w$ must have black children
- Make $w$ black and $x.p$ red
- then left-rotate on $x.p$
- new sibling of $x$ was a child of $w$ before the rotation $\Rightarrow$ 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

- Take 1 black off $x$ (now singly black) and $w$ (now red).
- Move that black to $x.p$.
- Do the next iteration with $x.p$ as the new $x$.
- If entered this Case from Case 1, the $x.p$ was red $\Rightarrow$ new $x$ is red&black $\Rightarrow$ color attribute of new $x$ is RED $\Rightarrow$ loop terminates. The new $x$ is made black in the last line.
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 ⇒ case 4.

Finally:

- Make w be x.p’s color (c)
- Make x.p black and w’s right child black
- Then left rotate on x.p
- 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.