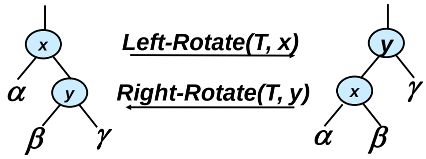
RB Trees: Rotation

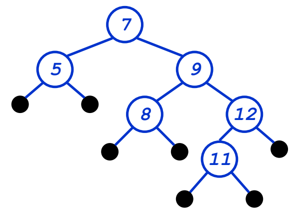


- Answer: A lot of pointer manipulation
  - *x* keeps its left child
  - *y* keeps its right child
  - *x*'s right child becomes *y*'s left child
  - *x*'s and *y*'s parents change
- *What is the running time?*

L13.1

Rotation Example

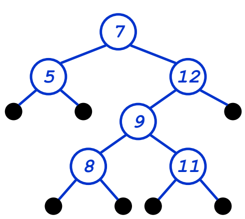
- Rotate left about 9:



L13.2

Rotation Example

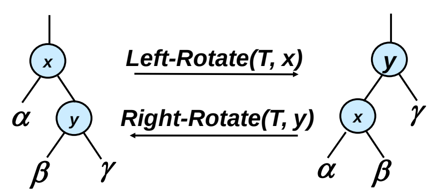
- Rotate left about 9:



L13.3

RB Trees: Rotation

- 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.

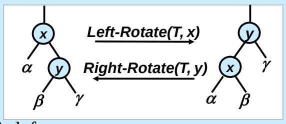


L13.4

Left Rotation – Pseudo-code

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
    then y.left.p = x
y.p = x.p          // Link x's parent to y.
if x.p == T.nil
    then T.root = y
else if x == x.p.left
        then x.p.left = y
else x.p.right = y
y.left = x         // Put x on y's left.
x.p = y
```



L13.5

Rotation

- The pseudo-code for Left-Rotate assumes that
  - *X.right* ≠ *T.nil*, and
  - root's parent is *T.nil*.
- 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.

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 = 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
else if 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)
```

How does it differ from the Tree-Insert procedure of BSTs?

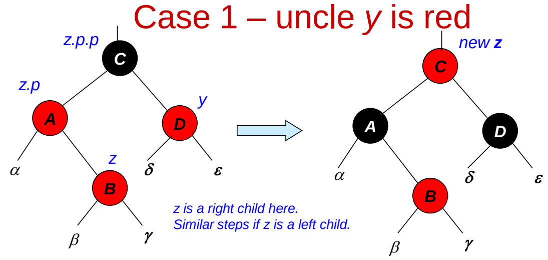
Which of the RB properties might be violated?

Fix the violations by calling RB-Insert-Fixup.

Insertion – Fixup

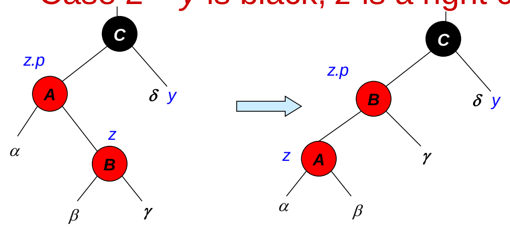
```
RB-Insert-Fixup (T, z)
1.    while z.p.color == RED
2.      if z.p == z.p.p.left
3.          y = z.p.p.right                  // y now is z's uncle
4.          if y.color == RED
5.              z.p.color = BLACK            // Case 1
6.              y.color = BLACK              // Case 1
7.              z.p.p.color = RED            // Case 1
8.              z = z.p.p                    // Case 1
9.          else if z == z.p.right     // y.color ≠ RED
10.                 z = z.p                  // Case 2
11.                 LEFT-ROTATE(T, z)        // Case 2
12.             z.p.color = BLACK            // Case 3
13.             z.p.p.color = RED            // Case 3
14.             RIGHT-ROTATE(T, z.p.p)       // Case 3
15.     else (if z.p == z.p.p.right)(same as 3-14
16.             with "right" and "left" exchanged)
17.   T.root.color = BLACK
```

Case 1 – uncle *y* is red



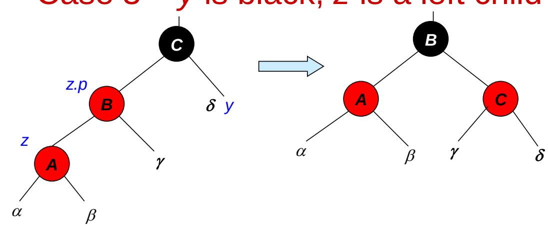
- *z.p.p* (*z*'s grandparent) must be black, since *z* and *z.p* are both red and there are no other violations of property 4.
- Make *z.p* and *y* black ⟹ now *z* and *z.p* are not both red. But property 5 might now be violated.
- Make *z.p.p* red ⟹ restores property 5.
- The next iteration has *z.p.p*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

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



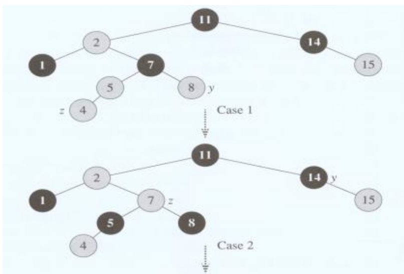
- Left rotate around *z.p*, *z.p* and *z* switch roles ⟹ now *z* is a left child, and both *z* and *z.p* are red.
- Takes us immediately to case 3.

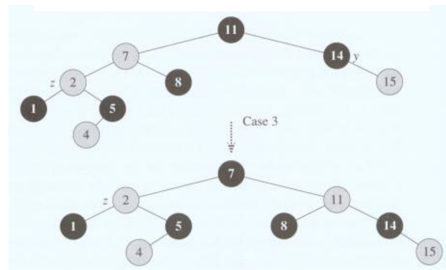
Case 3 – *y* is black, *z* is a left child



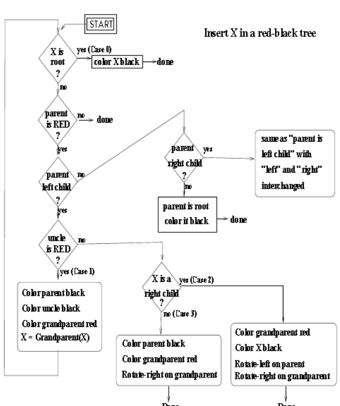
- Make *z.p* black and *z.p.p* red.
- Then right rotate on *z.p.p*. Ensures property 4 is maintained.
- No longer have 2 reds in a row.
- *z.p* is now black ⟹ no more iterations.

The operation of RB-INSERT-FIXUP





Flow Chart for Insertion in RB Tress



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 $\mathbf{O(\lg n)}$ time.
  - Note: there are at most 2 rotations overall.
    - It never performs more than two rotations, since the while loop terminates if case 2 or case 3 executed.

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. Why?
  - Black?
- Steps:
  - Do regular BST deletion.
  - Fix any violations of RB properties that may result.

Deletion - Transplant

```
Transplant(T, u, v)
/* Handle u is root of T */
1. if u.p == NIL
2.     T.root = v
/* if u is a left child */
3. else if u == u.p.left
4.     u.p.left = v
/* if u is a right child */
5. else u.p.right = v
/* update v.p if v is non-NIL */
6. if v ≠ NIL
7.     v.p = u.p
```

```
RB-Transplant(T, u, v)
/* Handle u is root of T */
1. if u.p == T.nil
2.     T.root = v
/* if u is a left child */
3. else if u == u.p.left
4.     u.p.left = v
/* if u is a right child */
5. else u.p.right = v
6. v.p = u.p
```

L12.18

Deletion – BST v.s. RB Tree

```
Tree-Delete(T, z)
   /* (a) z has no left child */
1  if z.left == NIL
2.     Transplant(T, z, z.right)
   /* (b) z has a left child, but no right child */
3  else if z.right == NIL
4      Transplant(T, z, z.left)
   /* (c) z has two children */
5  else y = Tree-Minimum(z.right)  /* find z's successor */
6      if y.p ≠ z
7          Transplant(T, y, y.right)
8          y.right = z.right
9          y.right.p = y
   /* (d) if y is z's right child */
10     Transplant(T, z, y)
11     y.left = z.left    /* replace y's left child by z's left
       child */
12     y.left.p = y
```

```
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 if 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.     else 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)
```

RB-Deletion

```
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 if z.right == T.nil
7.      x = z.left
8.      RB-Transplant(T, z, z.left)
9.  else y = Tree-Minimum(z.right)   // z has two children
10.     y-original–color = y.color
11.     x = y.right
12.     if y.p == z
13.         x.p = y
14.     else 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)
```

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

- 1~2: Maintain y either removed or moved.
- 3 ~ 8: if z has fewer than two children, z is removed.
- 9: z has two children, y points to z's successor. After deleting z, y will point to z's position
- 10: y may change color. keep y's color. We cannot remove or move y if it's black (line 21).
- 11: Keep track of x that moves into y's original position like BST.
- 12: if z's successor y is z's child
- 13: set x.p to y causes x.p to point to the original position of y's parent, even if x = T.nil.
- 14~20: Remove z and set y to z's position
- 21: if y is Black, we might have introduced one or more violations of properties. Use RB-Delete-Fixup to restore properties.

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 *y.p* and *x* are both red.
  - Prop. 5. Any path containing *y* now has 1 fewer black node.

RB Properties Violation

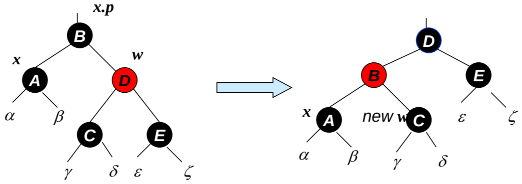
- Prop. 5. Any path containing *y* now has 1 fewer black node.
  - Correct by giving *x* an "extra black."
  - Add 1 to count of black nodes on paths containing *x*.
  - Now property 5 is OK, but property 1 is not.
  - *x* is either ***doubly black*** (if *x.color* = BLACK) or ***red & black*** (if *x.color* = RED).
  - The attribute *x.color* 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.

```
RB-Delete-Fixup(T, x)        // to restores properties 1, 2, 4.
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                 // Case 1
6.               x.p.color = RED                 // Case 1
7.               LEFT-ROTATE(T, x.p)             // Case 1
8.               w = x.p.right                   // Case 1
9.           if w.left.color == BLACK and w.right.color == BLACK     // x is still x.p.left
10.              w.color = RED                   // Case 2
11.              x = x.p                         // Case 2
12.          else if w.right.color == BLACK
13.                  w.left.color = BLACK        // Case 3
14.                  w.color = RED               // Case 3
15.                  RIGHT-ROTATE(T,w)           // Case 3
16.                  w = x.p.right               // Case 3
17.              w.color = x.p.color             // Case 4
18.              x.p.color = BLACK               // Case 4
19.              w.right.color = BLACK           // Case 4
20.              LEFT-ROTATE(T, x.p)             // Case 4
21.              x = T.root                      // Case 4
22.      else (same as then clause with "right" and "left" exchanged)
23.  x.color = BLACK
```

Deletion – Fixup

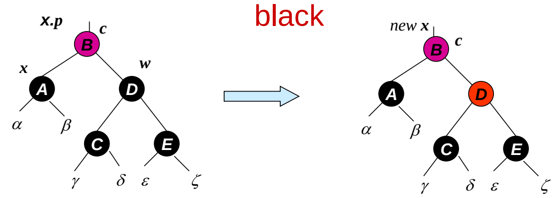
- ***Idea:*** Move the extra black up the tree until *x* points to a red-&-black node ⇒ turn it into a black node (line 23),
- *x* points to the root ⇒ just remove the extra black, or
- Having performed suitable rotations and recolorings and exit the loop.
- Within the **while** loop:
  - *x* always points to a nonroot doubly black node.
  - *w* is *x*'s sibling.
  - *w* cannot be *T.nil*, since that would violate property 5 at *x.p*.
- 8 cases in all, 4 of which are symmetric to the other.

Case 1 – x's sibling *w* is red



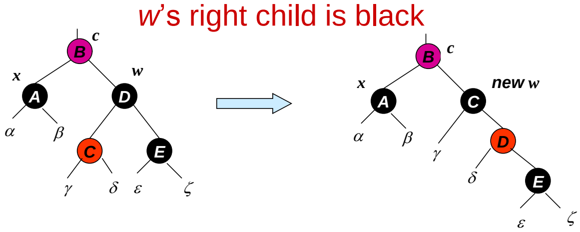
- *w* must have black children.
- Make *w* black and *x.p* red (because *w* is red *x.p* couldn't have been red).
- Then left rotate on *x.p*.
- New sibling of *x* was a child of *w* before rotation ⇒ must be black.
- Go immediately to case 2, 3, or 4.

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



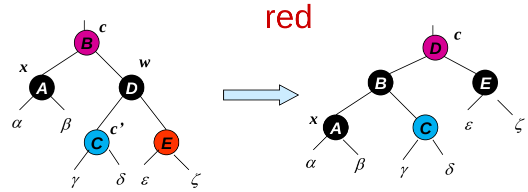
- Take 1 black off *x* (⇒ singly black) and off *w* (⇒ red).
- Move that black to *x.p*.
- Do the next iteration with *x.p* as the new *x*.
- If entered this case from case 1, then *x.p* was red ⇒ new *x* is red & black ⇒ color attribute of new *x* is RED ⇒ loop terminates. Then new *x* is made black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

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.

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



- 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.

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.

RB Tree Visualization

- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html