Red-Black Tree

Red-black trees: Overview

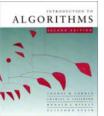
- Red-black trees are a variation of binary search trees to ensure that the tree is **balanced**.
 - Balanced search tree: A search-tree data structure for which a height of O(lg n) is guaranteed when implementing a dynamic set of n items.
- Operations take $O(\lg n)$ time in the worst case.

Red-Black Tree

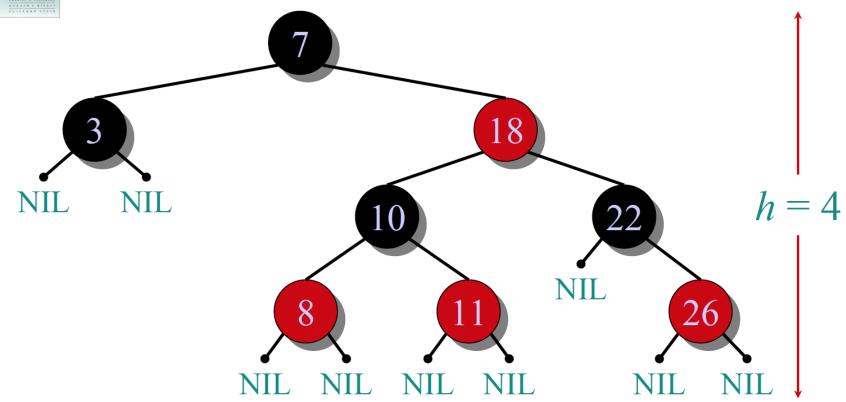
- Binary search tree + 1 bit per node:
 - the attribute *color*, which is either **red** or **black**.
- All other attributes of BST's node are inherited:
 - *key*, *left*, *right*, and *p*.
- All empty trees (leaves) are colored black.

Red-Black properties:

- 1. Every node is either red or black.
- The root is black.
- 3. Every leaf (*nil*) is black.
- 4. If a node is red, then both its children are black.
- 5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.



Example of a red-black tree



Red-Black Tree Height

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) \leq # 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) \leq 2 bh(x). Thus, h(x) \leq 2 s(x). QED
```

Red-Black Tree Operations

Corollary.

- The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR of Binary Search Tree all run in $O(\lg n)$ time on a red-black tree with n nodes.
- Cause no violation to the red-black properties.
- The operations INSERT and DELETE require modifications to the redblack tree

Insertion into A Red-Black Tree

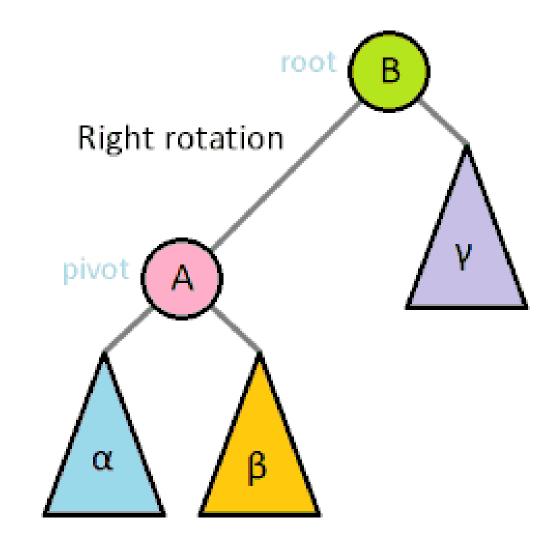
BST-insert x

Color x red

Property 4 may be violated

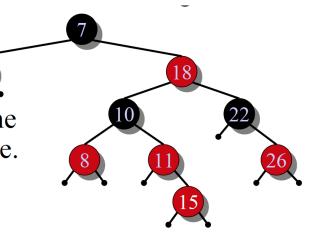
Idea: Move the violation up the tree

Rotation



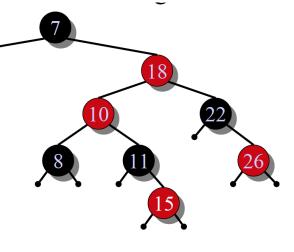
Example:

- Insert x = 15.
- Recolor, moving the violation up the tree.



Example:

- Insert x = 15.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).

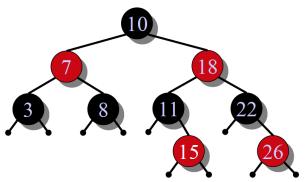


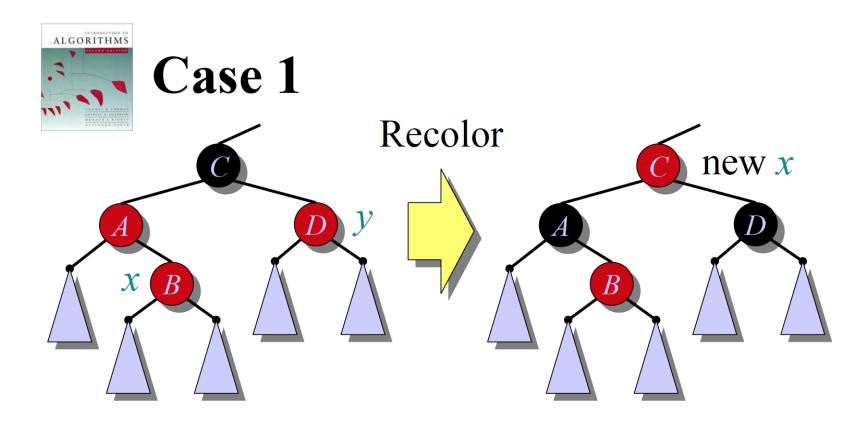
Example:

- Insert x = 15.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- Left-Rotate(7) and recolor.



- Insert x = 15.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- Left-Rotate(7) and recolor.



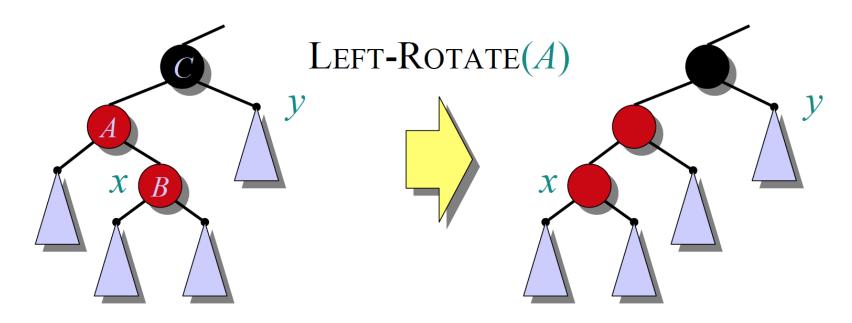


(Or, children of *A* are swapped.)

Push C's black onto A and D, and recurse, since C's parent may be red.



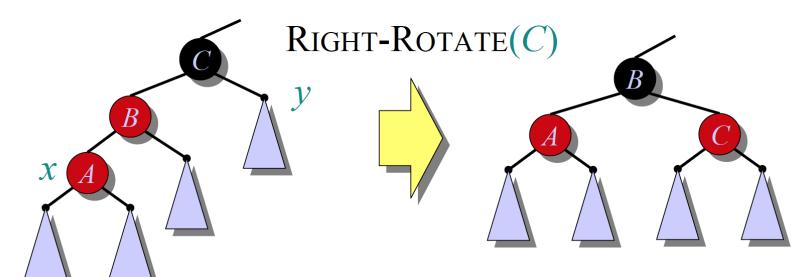
Case 2



Transform to Case 3.



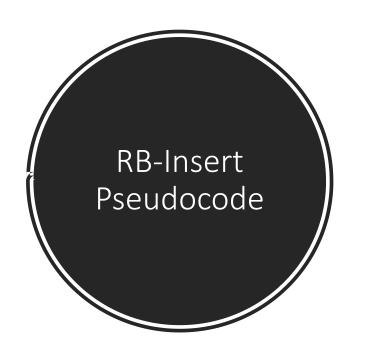
Case 3



Done! No more violations of RB property 3 are possible.

Runing time

- Go up the tree performing Case 1, which only recolors nodes.
 - $O(h) = O(\lg n)$
- If Case 2 or Case 3 occurs
 - perform 1 or 2 rotations, and terminate.
 - O(1)
- ightharpoonup Running time: $O(\lg n)$ with O(1) rotations.



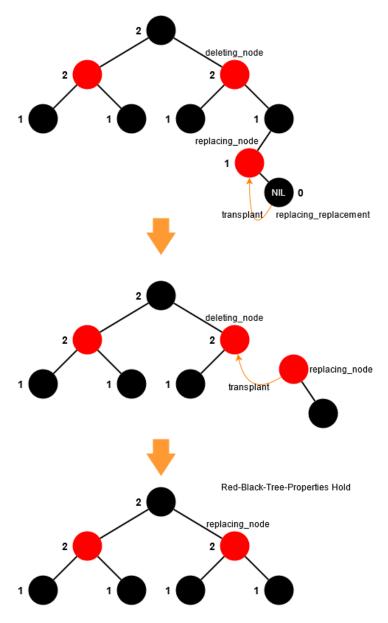
```
RB-INSERT(T, x)
    TREE-INSERT(T, x)
    color[x] \leftarrow RED \triangleright only RB property 3 can be violated
    while x \neq root[T] and color[p[x]] = RED
        do if p[x] = left[p[p[x]]]
             then y \leftarrow right[p[p[x]]] > y = aunt/uncle of x
                    if color[y] = RED
                     then \langle Case 1 \rangle
                     else if x = right[p[x]]
                            then \langle Case 2 \rangle \triangleright Case 2 falls into Case 3
                           \langle Case 3 \rangle
             else ("then" clause with "left" and "right" swapped)
    color[root[T]] \leftarrow BLACK
```

Deletion

- a bit more complicated than inserting a node.
- based on the BST's Tree-Delete
- need to customize the TRANSPLANT so that it applies to a red-black tree

```
RB-TRANSPLANT(T, u, v)
   if u.p == (T.nil)
       T.root = v
   elseif u == u.p.left
       u.p.left = v
  else u.p.right = v
   v.p = u.p
```

Deletion: Transplant

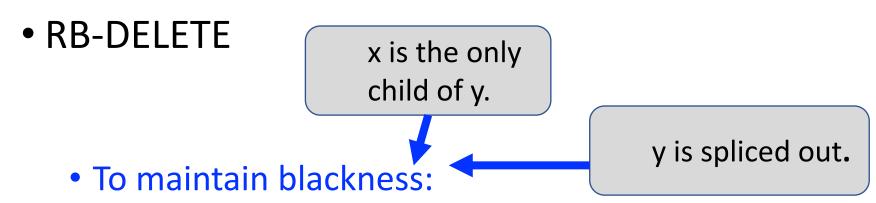


Not over yet!

```
RB-DELETE(T, z)
    v = z
 2 \quad 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 = \text{Tree-Minimum}(z.right)
10
        y-original-color = y.color
11
        x = y.right
        if y.p == z
12
13
             x.p = y
        else RB-TRANSPLANT(T, y, y.right)
14
             y.right = z.right
15
16
             y.right.p = y
        RB-TRANSPLANT (T, z, y)
17
        y.left = z.left
18
        y.left.p = y
19
        y.color = z.color
20
    if y-original-color == BLACK
        RB-DELETE-FIXUP(T, x)
```

Why RB-DELETE-FIXUP?

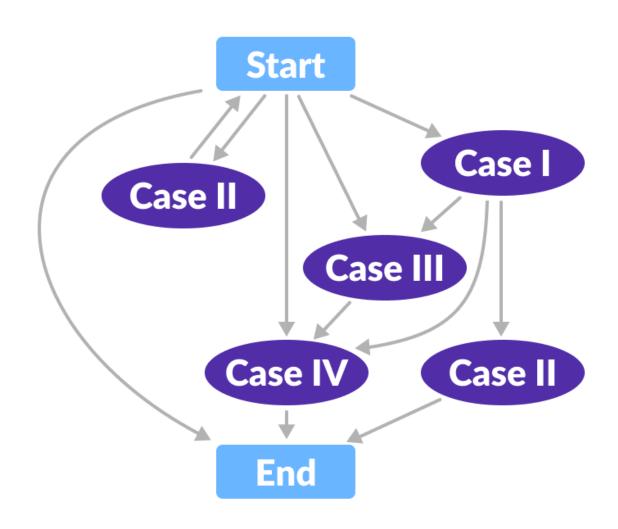
- If the deleted node is red?
 - Not a problem no RB properties violated



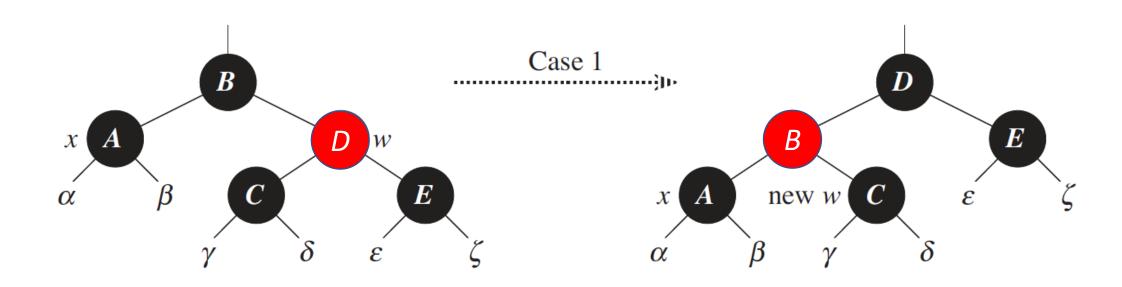
- x takes blackness of y.
- If x was black, now it is **doubly black** (needs to be resolved)

```
RB-DELETE-FIXUP (T, x)
    while x \neq T.root and x.color == BLACK
        if x == x.p.left
            w = x.p.right
            if w.color == RED
                 w.color = BLACK
                                                                   // case 1
                                                                   // case 1
 6
                 x.p.color = RED
                 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
10
                 w.color = RED
                                                                   // case 2
                                                                   // case 2
11
                 x = x.p
            else if w.right.color == BLACK
12
13
                     w.left.color = BLACK
                                                                   // case 3
                                                                   // case 3
14
                     w.color = RED
15
                     RIGHT-ROTATE (T, w)
                                                                   // case 3
16
                     w = x.p.right
                                                                   // case 3
                                                                   // case 4
17
                 w.color = x.p.color
                                                                   // case 4
18
                 x.p.color = BLACK
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)
    x.color = BLACK
```

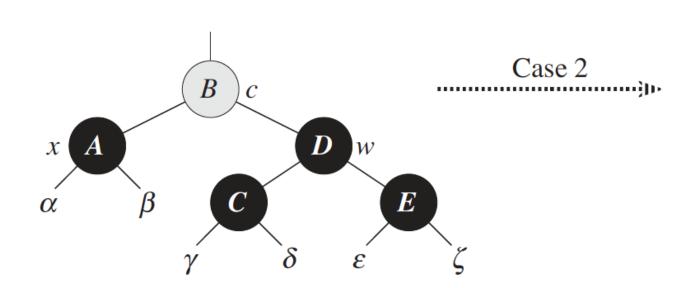
RB-DELETE-FIXUP



Case 1: x's sibling w is red



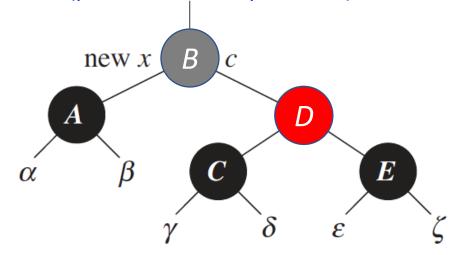
Case 2: x's sibling w is black, and both of w's children are black



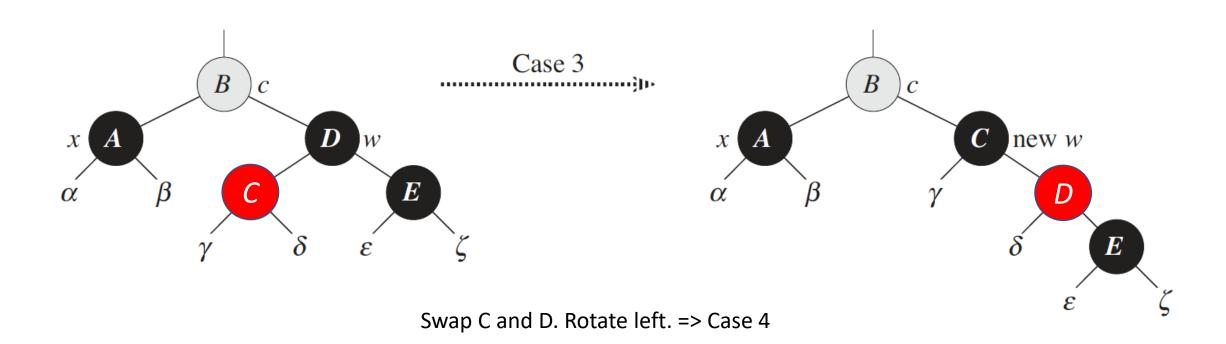
If B was red, it becomes black, solved!

If B was black, it becomes doubly black

(problem moved up the tree)

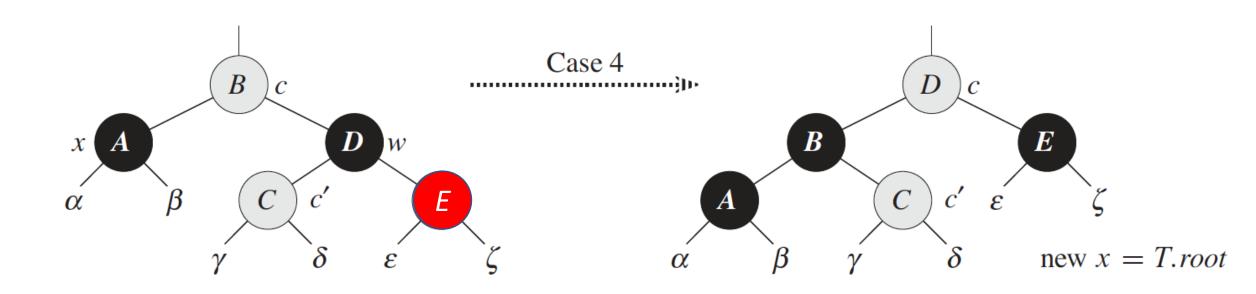


Case 3: x's sibling w is black, w's left child is red, and w's right child is black



The idea is to align the opposite side of x with redness in between blacknesses.

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



As B rotates down to take extra-blackness, D rises to take its place. Then, E's redness allows it to come up to maintain blackness of D.