Red-Black Trees
Deletion

ESO207

Indian Institute of Technology, Kanpur

Summary

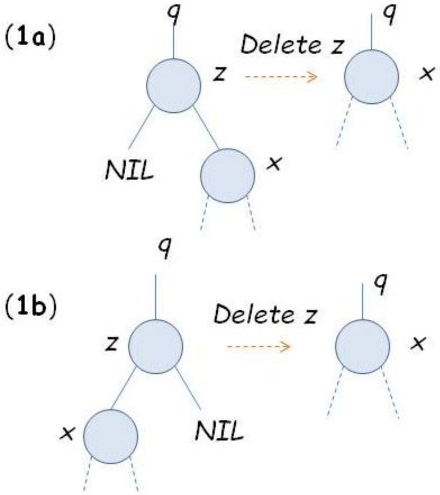
- We now look at the operation of deleting a given node from a red-black tree.
- An algorithm is presented that takes $O(\log n)$ time.
- Deleting a node is a bit more complicated than inserting a node.
- It is based upon, and extends the deletion operation for binary search trees.

Review of Deletion in Binary Search Trees

- Review: Consider DELETE(T, z) where
  1. T is a binary search tree, and,
  2. z is the node to be deleted.
- Split into three cases:
- **Case 1:** z has only one child x.

  Action: Transplant the subtree rooted at z by the sub-tree rooted at x. (x can be NIL).

Case 1



Case 1: z has only one child.

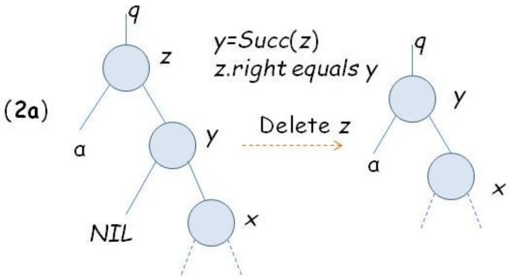
Review of Deletion in Binary Search Trees

- **Case 2:** z has both children and the successor y of z is the right child of z.

  Action 1: Transplant the subtree rooted at z by the subtree rooted at y.

  Action 1: Make the left child of z as the left child of y. (Note: the left child of y is NIL).

Case 2



Deletion in BST: Case 3

- **Case 3:** z has both children, and the successor y of z is not the right child of z.
- So, the successor y of z is in the left subtree of the right child of z.
- Note: y's left child is NIL.

  a. Transplant subtree rooted at y by the subtree rooted at $y.right$. Now replace z by y, that is,
  b. Make y's right child same as z's right child (which is non-NIL) and change its parent pointer to point to y (Set $y.right.p = y$).

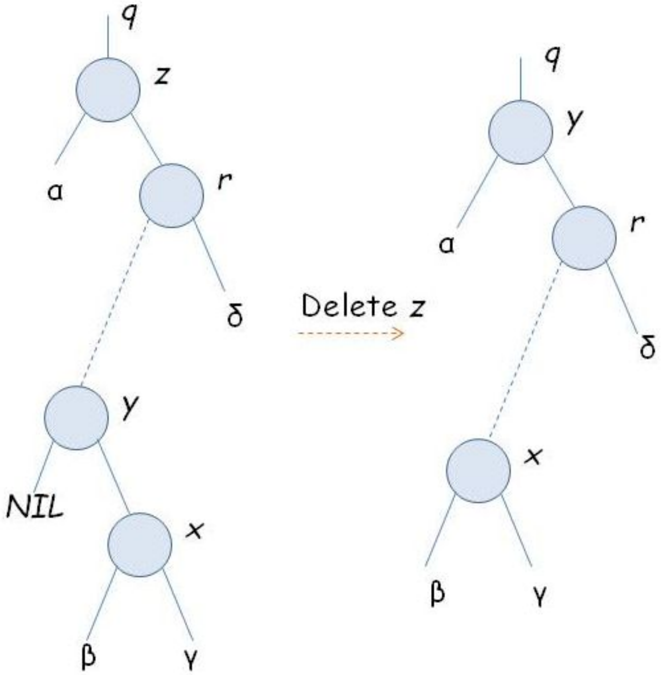
Case 3: Deletion in BST

c. Transplant subtree rooted at z by the sub-tree rooted at y. Note now that y's left child is NIL and y's right child is same as z's right child.

d. The transplant operation makes y's parent same as z's parent, and y becomes the left(respectively, right) child of its parent as z.

e. Make the left child of z to be the left child of y, and change its parent pointer to point to y. (Set $y.left = z.left$ and set $y.left.p = y$).

(3)



RB-Transplant

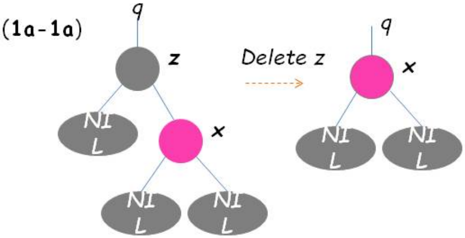
- For red-black trees, we use procedure RB-TRANSPLANT
- Small modification of TRANSPLANT procedure of BST. Uses T.NIL instead of NIL.
- Replaces subtree rooted at u by the subtree rooted at v.
- Color of v (and other nodes) unchanged. u is deleted from tree.

RB-TRANSPLANT(T, u, v)

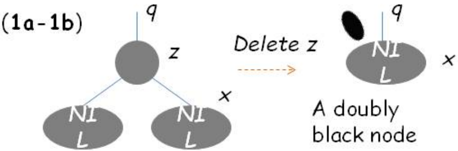
if $u.p == T$.NIL // u is the root of T
    $T.root = v$
elseif $u == u.p.left$ // u is the left child of its parent
    $u.p.left = v$ // make v the left child of u's parent
else $u.p.right = v$ // otherwise make it the right child
$v.p = u.p$

RB Deletion: Case 1

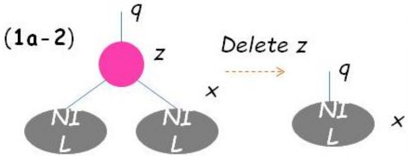
```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.NIL
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.NIL
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
// Now we may have an RB-Tree problem if
// y-original-color == BLACK
```



x replaces ***z***. *Case* 1 before fixup.



x is *NIL* and replaces ***z*** *which is black*. To preserve the black-height property , we think that z sheds its blackness onto x making it ***doubly black***.



x is *NIL* and replaces *z*. Black heights are preserved

If Node z is colored red then in Case 1 (i.e., one of the children of z is NIL) then transplanting z by its non-NIL child x preserves red-black properties.

RB Deletion: Case 1

- Call RB-DELETE-FIXUP if deleted node z is black.

```
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)
if y-original-color = BLACK
    RB-DELETE-FIXUP(T, x)
```

Cases 2 and 3

```
    // z has two children: Case 2
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)
```

- In Cases 2 and 3, y is z's successor and y will move into z's position, as in the case of BST deletion.
- Node y will take z's color (line 20). So, the original color of y is remembered in *y-original-color*.

Cases 2 and 3

```
    // z has two children: Case 2
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)
```

- x is the node that moves into y's position. x points to y's only child, or, if y has no children, to T.NIL.
- $x.p$ is set to point to the original position in the tree of y's parent, even if x is T.NIL.
- But if z is y's original parent, then, y remains x's parent. (Line 13).

Cases 2 and 3

```
    // z has two children: Case 2
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)
```

- If y was black originally, then by removing it and placing it elsewhere, with z's color—may have introduced one or more violations of the red-black properties.
- This is restored by RB-DELETE-FIXUP in line 22.
- If y was originally colored red, then no such violations result, whether y is removed or moved.

Cases 2 and 3

// z has two children: Case 2
else $y = \text{Tree-Minimum}(z.right)$
    $y\text{-}original\text{-}color = y.color$
    $x = y.right$
    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 $y\text{-}original\text{-}color$ == BLACK
    RB-DELETE-FIXUP(T, x)

If y was originally colored red, then no such violations result, whether y is removed or moved. Because,

1. No black-heights in the tree have changed.
2. No red nodes have been made adjacent–because:

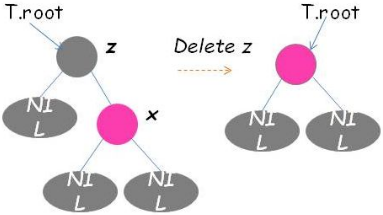
Cases 2 and 3

If y was originally colored red, no red-nodes are made adjacent, because,

1. y replaces z with z's color. So there cannot be two adjacent red nodes in y's new position
2. If y was not z's right child, then y's original right child x replaces y. If y is red, x must be black and therefore, cannot cause two red nodes to be adjacent.
3. y cannot be the root since the root is black

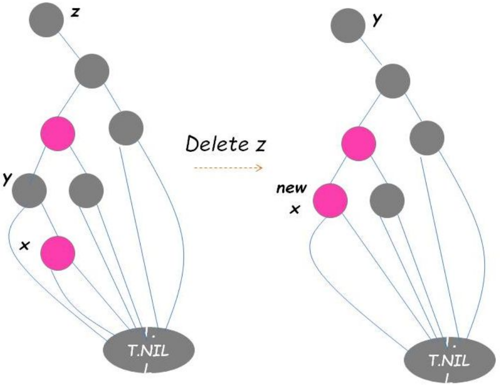
Problems that RB-DELETE-FIXUP must solve

- RB-DELETE-FIXUP is called when the original color of y is black.
- If y had been the root (Case 1: with z equal to y and z had only one child x which was red)



Problem 2 for RB-DELETE-FIXUP

- In Case 3, y is successor of z and x is right child of y.
- It can happen that x and x's new parent are both red.



Problem 3 for RB-DELETE-FIXUP

- Since y's color was black, moving y around the tree causes any simple path that previously contained y to have one fewer black node. Thus black-height property may be violated.
- This is **virtually** corrected by saying that the node x now occupying y's place has an extra black.
- We will allow x (and only x) to have an additional count of blackness.

Doubly black or red + black node

- So if x is colored red, it means it is colored both red and black.
- If x is colored black, it means it is colored doubly black.
- That is, black count of a node could be 0 (red), 1 (singly black) or 2 (doubly black).
- The tree is no longer a red-black tree (because of red + black or doubly black).

Color transference principle

- The node that moves in the tree, namely y, takes the color of the node that is deleted, namely z, and transfers its original color to its only child x.
- x takes the place of y and y takes the place of z.

RB-Delete-Fixup

- Main loop structure is shown below.
- x refers to the child of y that moves into the position of y and inherits the blackness of y.

RB-DELETE-FIXUP (T, x) {
1. **while** ($x \neq T.root$ and $x.color$ == BLACK) {
   ...
   }
   $x.color =$ BLACK
}

- In the code, x points to the "problem" node (i.e., the one that is doubly black or red$+$ black). This is the only multi-colored node in the tree.
- By $x.color$, we mean its original color. It will be assumed that there is an **additional black transferred to it.**

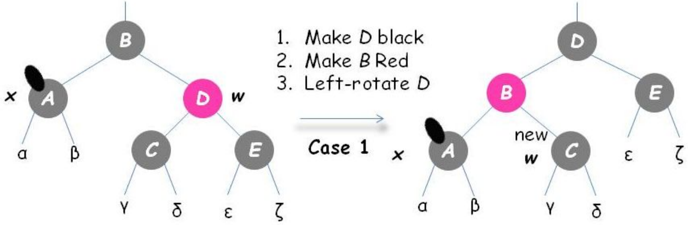
RB-Delete-Fixup

```
RB-DELETE-FIXUP (T, x) {
1.    while (x ≠ T.root and x.color == BLACK) {
              ...
      }
      x.color = BLACK
}
```

- Note that loop terminates when *x*'s color is red and color of *x* is set to black.
- In other words, a red + black is converted to black.

RB-Delete-Fixup: Case 1 of 4

- **Case 1: x's sibling w is red.**
- x is doubly black. w has black children. Make w black and parent of x red.
- Rotate left.
- Goal is to convert to one of cases 2, 3 and 4.



Code for sibling

```
SIBLING(T, x)
    if x ≠ NIL and x.p ≠ NIL
        if x == x.p.left
            return x.p.right
        else
            return x.p.left
    else return NIL
```

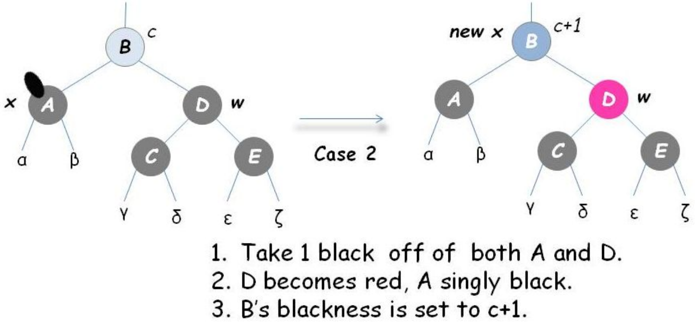
Pseudo-code Partial

RB-DELETE-FIXUP(T, x)

1. **while** $x \neq T.root$ and $x.color$ == BLACK
2. $\quad$ $w = sibling(x)$
3. $\quad$ **if** $w.color$ == RED $\qquad$ // Case 1
4. $\qquad$ $w.color = $ BLACK
5. $\qquad$ $x.p.color = $ RED
6. $\qquad$ LEFT-ROTATE$(T, x.p)$
7. $\qquad$ $w = sibling(x)$

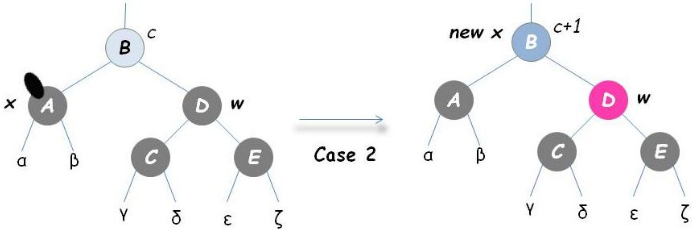
RB-Delete-Fixup: Case 2 of 4

- Cases 2,3 and 4 occur when node *w*, the sibling of *x* is black.
- These cases are distinguished by the color of *w*'s children.
- **Case 2: *x*'s sibling *w* is black, and both of *w*'s children are black.**



RB-Delete-Fixup: Case 2 of 4

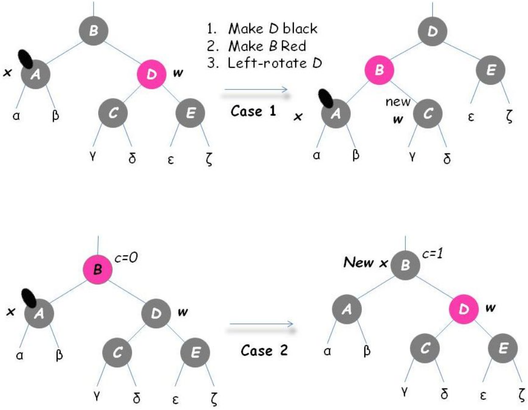
- **Case 2:** ***x*'s sibling *w* is black, and both of *w*'s children are black.**
- Take one black off both x and w.
- This leaves x singly black and w red.
- The additional black is transferred to x's parent.
- x now points to $x.p$.



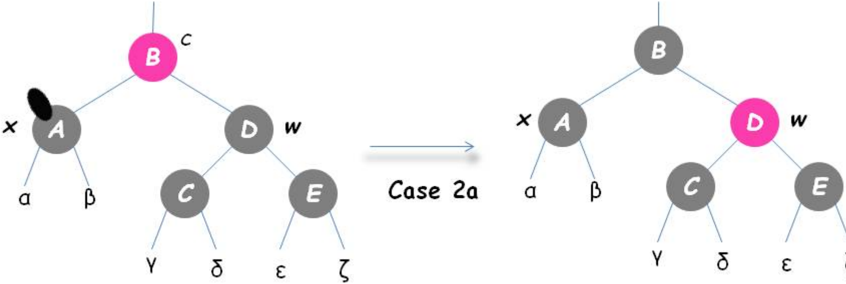
1. Take 1 black off of both A and D.
2. D becomes red, A singly black.
3. B's blackness is set to c+1.

Case1 → Case 2

- If we enter Case 2 from Case 1, then we terminate.

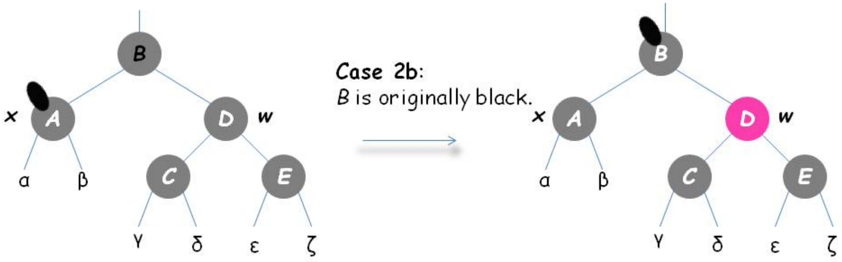


Case 2: termination



Case 2a: If *B* was originally red, then, the new *B* becomes black.
Fixup terminates.

Case 2: Violation moves up the tree



Case 2b: If *B* was originally black, then, the new *B* becomes doubly black.
Violation moves up the tree.

Pseudocode Partial

RB-DELETE-FIXUP(T, x)

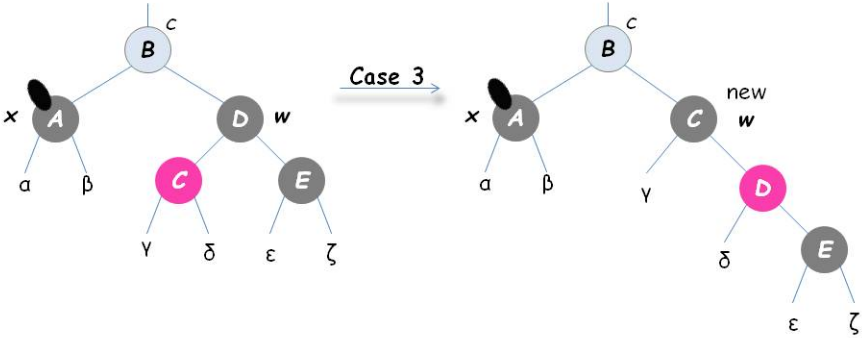
    if $w.left.color$ == BLACK and $w.right.color$ == BLACK // Case 2

        $w.color = $ RED

        $x = x.p$

RB-Delete-Fixup: Case 3 of 4

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



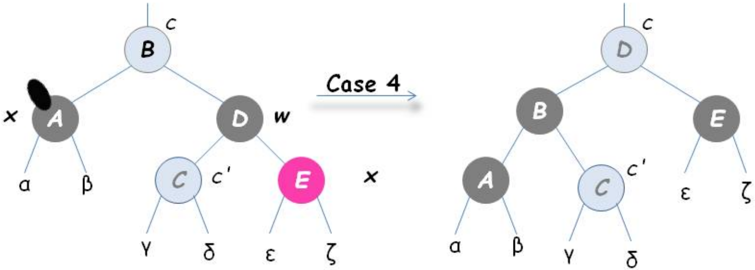
1. Interchange colors of C and D.
2. Right-rotate about D.
3. Purpose is to make the right child of D red, which is case 4.

Pseudo-code Partial

```
RB-DELETE-FIXUP(T, x)
        elseif  w.right.color == BLACK     // Case 3
            w.left.color = BLACK
            w.color = RED
            RIGHT-ROTATE(T, w)
            w = SIBLING(x)
```

RB-Delete-Fixup: Case 4

Case 4: ***x*'s sibling *w* is black, *w′* right child is red.**



Note:

1. B has color c, *C* has color c′. c,c′ could be red or black.
2. Rotate left around B.
3. B transfers its color c to D and takes up the additional blackness from A.
4. E is colored black (this is why E was needed to be red).

Pseudocode Partial

RB-DELETE-FIXUP(T, x)

$w.color = x.p.color$
$x.p.color =$ BLACK
$w.right.color =$ BLACK
LEFT-ROTATE$(T, x.p)$
break // break from **while** loop
// this is a termination condition