Deletions from Red-Black Trees

Michael T. Gastner (28 March 2023)

Disclaimer: These slides are based on and occasionally quote from ‘Introduction to Algorithms’ (3rd ed.) by Cormen et al. (2009), MIT Press.

Deleting a Node z from a General Binary Search Tree

Reminder

Before we study the special case of red-black trees, let us review how to delete a node z from a general binary search tree. There are four different cases:

1. z has no left child.
2. z has a left child but not a right child.
3. z has two children and the successor of z (i.e., the node with the next larger key) is z's right child.
4. z has two children and the successor of z is not z's right child.

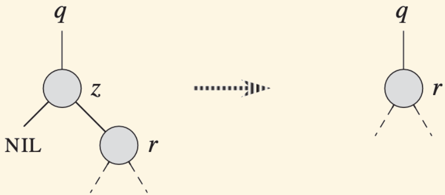
The next 4 pages illustrate how to delete a node in each of these cases.

2 / 24

Deleting a Node z from a General Binary Search Tree

Case 1: z Has No Left Child

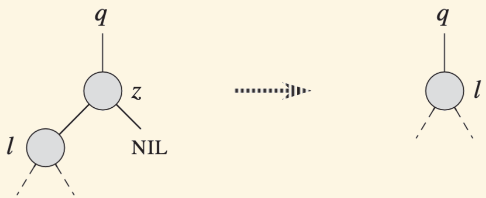
If z has no left child, then we replace z with its right child r, which may or may not be NIL.



Deleting a Node z from a General Binary Search Tree

Case 2: z Has a Left Child but No Right Child

If z has a left child l but no right child, then we replace z with l.

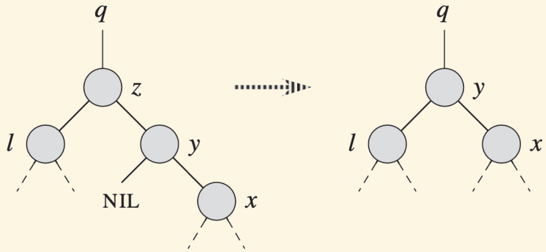


Deleting a Node z from a General Binary Search Tree

Case 3: z Has Two Children and the Successor of z Is z's Right Child

If z has two children and its left child l is its successor y, then

1. we replace z with y.
2. we update y's left child to be l.
3. y keeps its right child x.

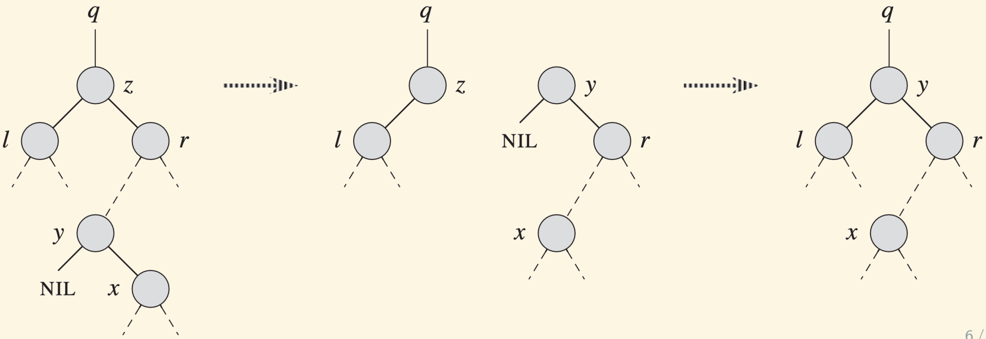


Deleting a Node z from a General Binary Search Tree

Case 4: z Has Two Children and the Successor of z Is Not z's Right Child

If z has a left child l and a right child r and if z's successor y is not r, then

1. we replace y with y's right child x.
2. we set y to be r's parent.
3. we set y's parent to be z's former parent q, which may or may not be NIL.
4. we set l's parent to be y.



6 / 24

Deleting a Node z from a General Binary Search Tree

Tree-Delete, Transplant, and Tree-Minimum

In the pseudocode on the next page, the procedure Tree-Delete(T, z) deletes a node z from a binary search tree T. Tree-Delete relies on two other procedures:

- Transplant(T, u, v) replaces the subtree rooted at node u with the subtree rooted at node v.
- Tree-Minimum(x) returns a pointer to the minimum element in the subtree rooted at node x.

The pseudocode of Transplant and Tree-Minimum is included on the next page alongside the pseudocode of Tree-Delete.

7 / 24

Deleting a Node z from a General Binary Search Tree

Pseudocode

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
TREE-MINIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```


8 / 24

Running Time of a Deletion from a General Binary Search Tree
$O(h)$

Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREE-MINIMUM in line 5.

TREE-MINIMUM need $O(h)$ time on a tree of height h because the sequence of nodes encountered forms a simple path downward from the root.

9 / 24

Deleting a Node z from a Red-Black Tree

Keeping Track of Moving Nodes

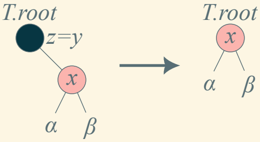
When we delete a node z from a red-black tree, we must ensure that the resulting tree satisfies all five red-black properties. There are two important notes, y and x that play an important role for maintaining the red-black properties:

- Node y:
  - If z has fewer than two children, then we set $y = z$.
  - If z has two children, then y is z's successor, which is the node that moves into the position vacated by z.
- Node x is the node that moves into y's original position.

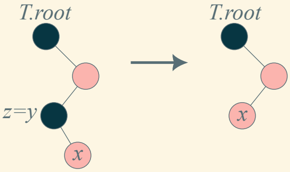
10 / 24

We Cannot Directly Apply Tree-Delete to Red-Black Trees

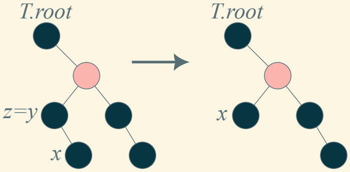
Scenarios that Violate Red-Black Properties



The root may become red.



Red parents may have red children.



Simple downward paths to different leaf nodes may contain a different number of black nodes.

Changes to TREE-DELETE

New procedures for repairing red-black properties and transplanting subtrees

To prevent violations of red-black properties, the procedure RB-DELETE will call a procedure named RB-DELETE-FIXUP.

RB-DELETE will also call a procedure RB-TRANSPLANT which differs in detail from TRANSPLANT. Here are the pseudocodes side by side with changes highlighted:

```
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
```

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```


12 / 24

RB-DELETE

Pseudocode

```
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)
else y = TREE-MINIMUM(z.right)
    y-original-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-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```


13 / 24

Comment on RB-DELETE

Why Do We Not Need to Fix Up the Tree If *y-original-color* Is Red?

If *y-original-color* is red, the red-black properties still hold after line 20, for the following reasons:

- If y is red, then removing or moving it does not change any black-height in the tree.
- If y is red, then removing $y = z$ in the **if** or **elseif** clause cannot cause red nodes to become adjacent. If, instead, y is moved to z's position in the **else** clause starting on line 9, then y takes z's previous color on line 20, and, again, the action taken on y does not cause a red parent to have a red child.
- If y was not z's right child, then y's original right child x replaces y in the tree. If y is red, then x must be black; hence, replacing y by x cannot cause two red nodes to become adjacent.
- Because y could not have been the root if it was red, the root remains black.

14 / 24

How Do We Fix Up the Tree If *y-original-color* Is Black?

Case Distinctions

If x is red, we simply turn it black. This action permits x to be the root or have a red parent. Because the black node y is replaced by a black node x, no black-heights are changed.

If x is black, we must apply different actions depending on the colours of x and its relatives:

- Case 1: x's sibling w is red.
- Case 2: x's sibling w is black, and both of w's children are black.
- Case 3: x's sibling w is black, w's left child is red, and w's right child is black.
- Case 4: x's sibling w is black, and w's right child is red.

The actions may need to be taken iteratively. By the end, we ensure that x is black. The pseudocode is on the next two pages. Afterwards, I illustrate each case.

15 / 24

RB-DELETE-FIXUP

Pseudocode (Part 1)

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                  // case 1: x's sibling w is red.
            x.p.color = RED                  // case 1
            LEFT-ROTATE(T, x.p)              // case 1
            w = x.p.right                    // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                    // case 2: x's sibling w is black, and ...
            x = x.p                          // case 2: ...both of w's children ...
                                             // case 2: ...are black.
```

Continued on next page.

16 / 24

RB-DELETE-FIXUP

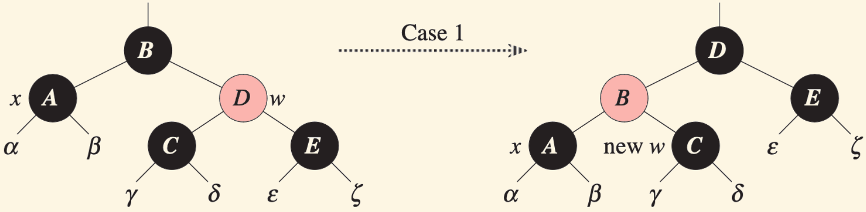
Pseudocode (Part 2)

```
        else if w.right.color == BLACK
                w.left.color = BLACK          // case 3: x's sibling w is black, ...
                w.color = RED                 // case 3: ...w's left child is red, ...
                RIGHT-ROTATE(T, w)            // case 3: ...and w's right child ...
                w = x.p.right                 // case 3: ...is black.
            w.color = x.p.color               // case 4: x's sibling w is black, and ...
            x.p.color = BLACK                 // case 4: ...w's right child is red.
            w.right.color = BLACK             // case 4
            LEFT-ROTATE(T, x.p)               // case 4
            x = T.root                        // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```


17 / 24

Case 1

x's Sibling w Is Red



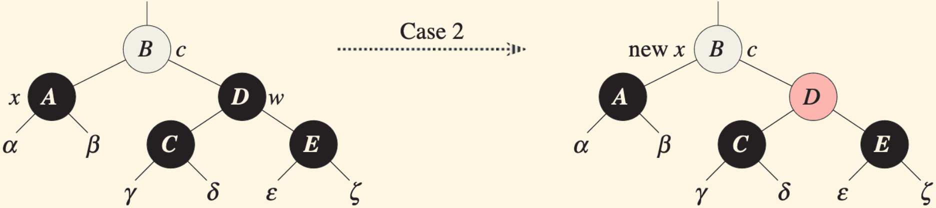
Since w must have black children, we can switch the colors of w and $x.p$ and then perform a left-rotation on $x.p$ without violating any of the red-black properties. The new sibling of x, which is one of w's children prior to the rotation, is now black, and thus we have converted case 1 into case 2, 3, or 4.

18 / 24

Case 2

x's Sibling w Is Black, and Both of w's Children Are Black

Both light grey nodes labeled B in the diagrams below have the same color attribute c, which may be either RED or BLACK.

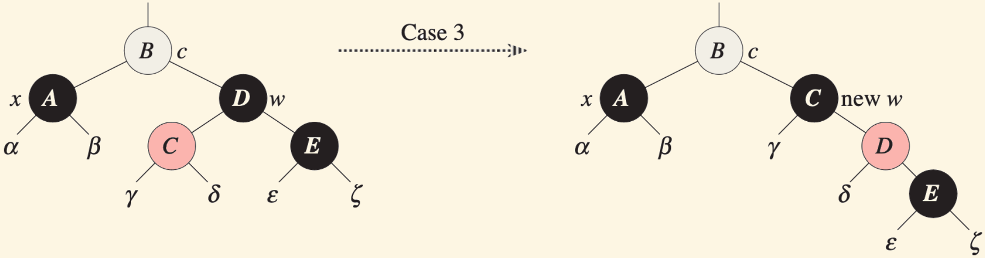


When we arrive at case 2, the black-height of w equals the black-height of x plus 1. We reduce the black-height of w by making it red. Afterwards, x moves up to its former parent.

If we arrive at case 2 through case 1, the new node x is red. Hence, the **while** loop in RB-DELETE-FIXUP terminates and x changes its color to black.

Case 3

x's Sibling w Is Black, w's Left Child Is Red, and w's Right Child Is Black



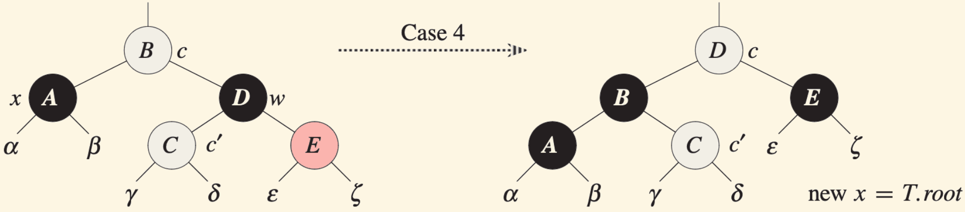
We can switch the colors of w and its left child $w.left$ and then perform a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child. Therefore, we have transformed case 3 into case 4.

20 / 24

Case 4

x's Sibling w Is Nlack, and w's Right Child Is Red

The color attributes c and c' in the diagram below may be either RED or BLACK.



Whe we arrive at case 4, the black-height of w equals the black-height of x plus 1. By making some color changes and performing a left rotation on $x.p$, we balance the black-heights and also satisfy all other red-black properties.

Setting x to be the root causes the **while** loop to terminate.

Running Time of RB-DELETE

$O(\log n)$

Since the height of a red-black tree of n nodes is $O(\log n)$, the total cost of the procedure without the call to RB-DELETE-FIXUP takes $O(\log n)$ time.

Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the **while** loop can be repeated, and then the pointer x moves up the tree at most $O(\log n)$ times, performing no rotations.

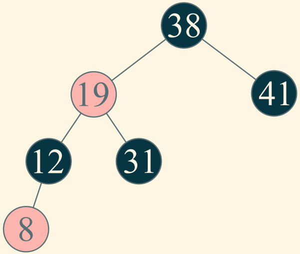
Therefore, the procedure RB-DELETE-FIXUP takes $O(\log n)$ time and performs at most three rotations, and the overall time for RB-DELETE is also $O(\log n)$.

22 / 24

Exercise

Deleting Nodes from a Red-Black Tree

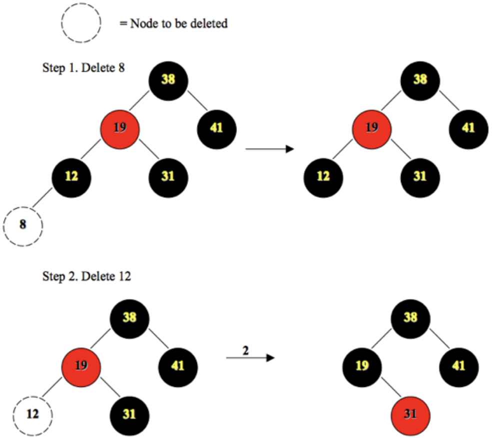
Consider the initial red-black tree shown below. Show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.

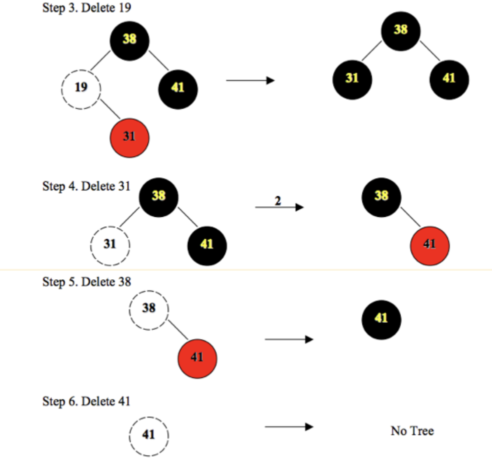


23 / 24

Solution to Exercise

https://github.com/gzc/CLRS/blob/master/C13-Red-Black-Trees/13.4.md





24 / 24