Red-Black Trees

The lecture notes are mostly based on Chapter 14 of Cormen, Leiserson, Rivest, and Stein. Introduction to Algorithms. 3rd Ed. 2009. MIT Press. Cambridge, Massachusetts.

Contents

Motivation:

Red-black trees are one of many search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take O(lg n) time in the worst case.

1 What is red-black tree

A red-black tree is a binary search tree

- With one extra bit of storage per node: Color
- Color can be either red or black
- Ensures that there is no such path, which is more than twice as long as any other
- Balanced

A node in a red-black tree contains:

- key
- left: pointer to left child.

  left = NIL if no left child.
- right: pointer to right child.

  right = NIL if no right child.
- p: pointer to parent.

  p=NIL for the root node.
- **color: color of the node**

Lemma 1

A red-black tree with n internal nodes has height at most 2 lg(n + 1)

2 Properties of a red-black tree

- Every node is either **RED** or **BLACK**.
- The root is black.
- Every leaf (NIL) is black.
- If a node is red, both its children are black.
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

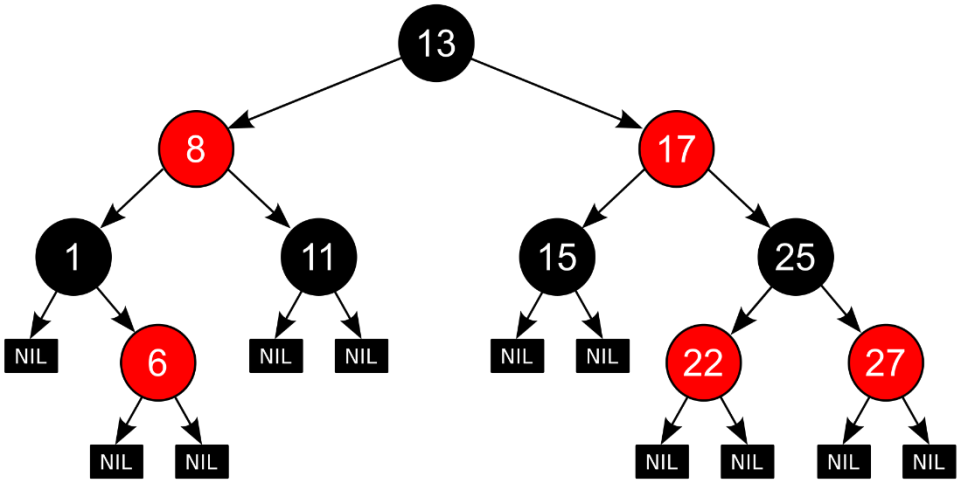


Figure 1: Example of a red-black tree.

3 Rotation of nodes in a red-black tree

```
LEFT-ROTATE(T, x)
 1  y = x.right              // set y
 2  x.right = y.left         // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4       y.left.p = x
 5  y.p = x.p                // link x's parent to y
 6  if x.p == T.nil
 7       T.root = y
 8  elseif x == x.p.left
 9       x.p.left = y
10  else x.p.right = y
11  y.left = x               // put x on y's left
12  x.p = y
```

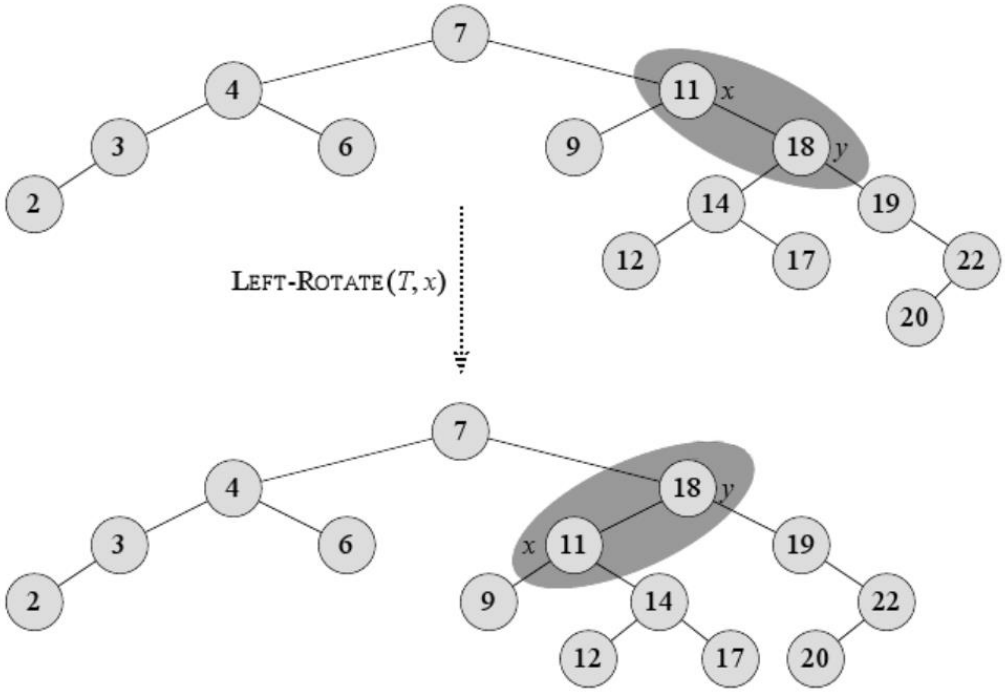


Figure 2: Left rotation in a red-black tree

```
RIGHT-ROTATE(T,x)
 1  y = x.left                    // set y
 2  x.left = y.right              // turn y's right subtree into x's left subtree
 3  if y.right ≠ T.nil
 4      t.right.p = x
 5  y.p = x.p                     // link x's parent to y
 6  if x.p == T.nil
 7      T.root = y
 8  else if x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.right =x                    // put x on y's right
12  x.p =y
```

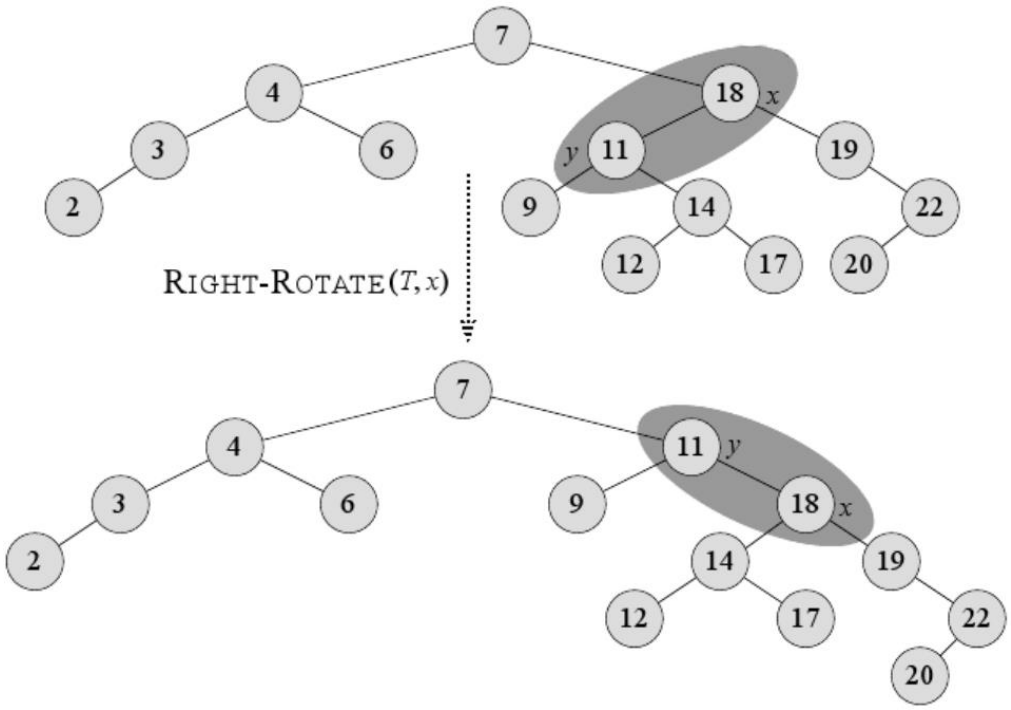


Figure 3: Right rotation in a red-black tree.

4 Insertion of a node into a red-black tree

```
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)
```

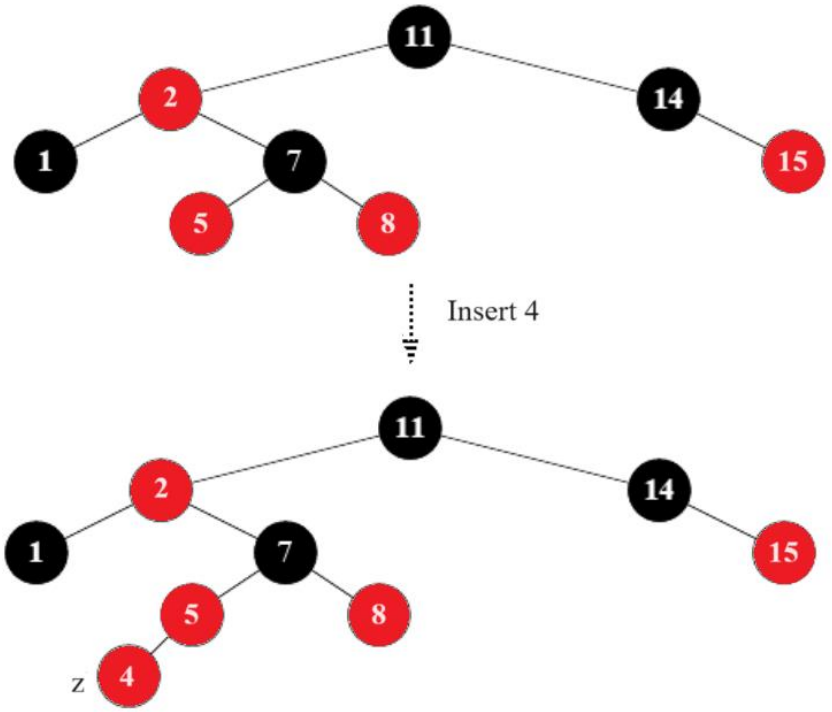


Figure 4: Insertion in a red-black tree before fixup.

- RB-INSERT-FIXUP restores the red-black properties to the tree

```
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.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

// if z's parent is a left child
// y is z's uncle
// are z's parent and uncle both red?

Case 1

Case 2

Case 3

- Uncle node: Parent node's sibling (Child from same parent)

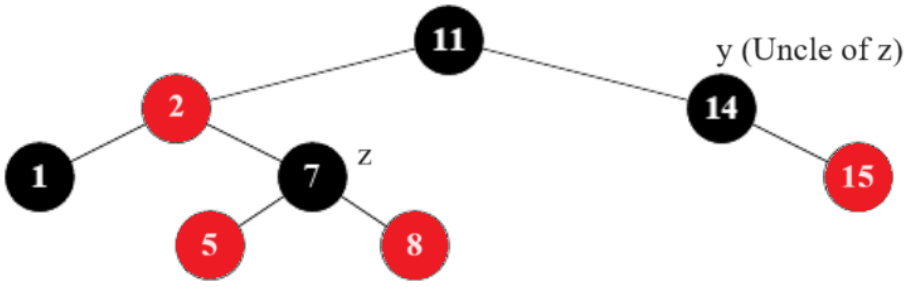


Figure 5: Uncle node.

- The cases to check to initiate insertion-fixup:

  Case 1. z's uncle y is red
  - Color y and z's parent black,
  - Color z's grandparent red.
  - Update z = z's grandparent.

Case 2. z's uncle y is black and z is a right child
- Update z= z's parent.
- Left rotate z.

Case 3. z's uncle y is black and z is a left child
- Color z's parent black
- Color z's grandparent red.
- Right rotate z's grandparent.

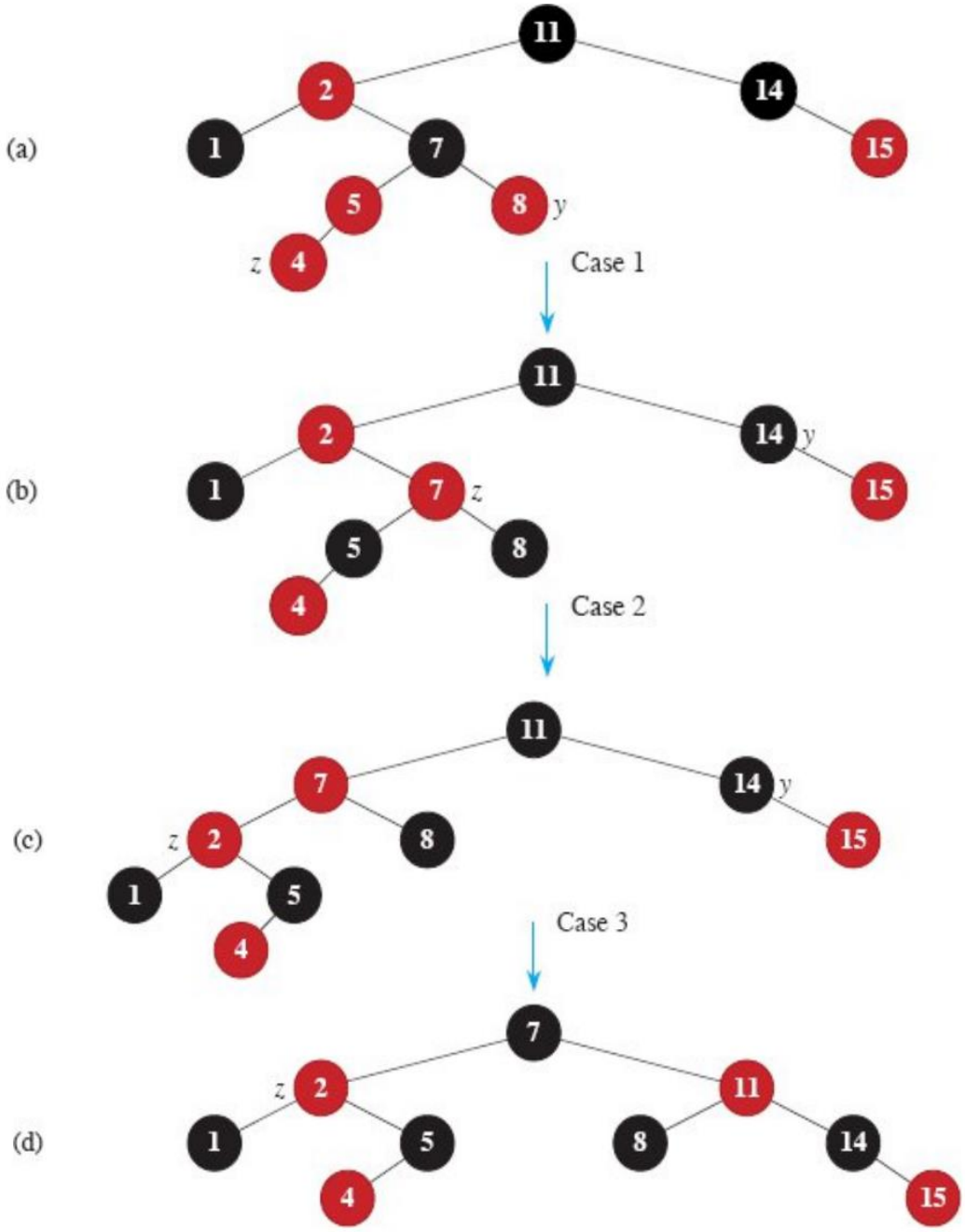


Figure 6: RB-Insertion-Fixup cases.

5 Deletion of a node from a red-black tree

- RB-DELETE deletes a node from the tree
- RB-TRANSPLANT helps to move subtrees within the tree

```
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       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-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
```

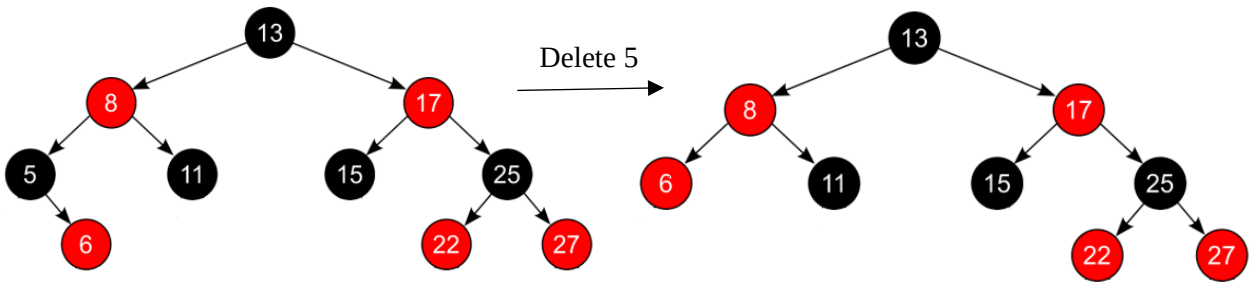


Figure 7: Deletion in a red-black tree.

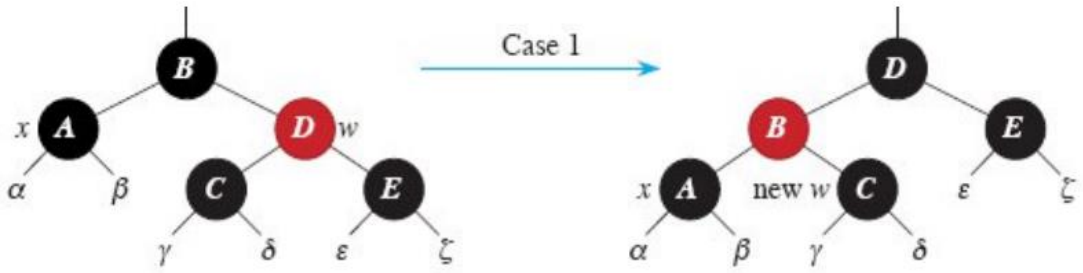
- RB-DELETE-FIXUP restores the red-black properties to the tree

```
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                                    }  Case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                                    }
11              x = x.p                                          }  Case 2
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                                }  Case 3
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                                       }  Case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

- The cases to check to initiate delete-fixup:

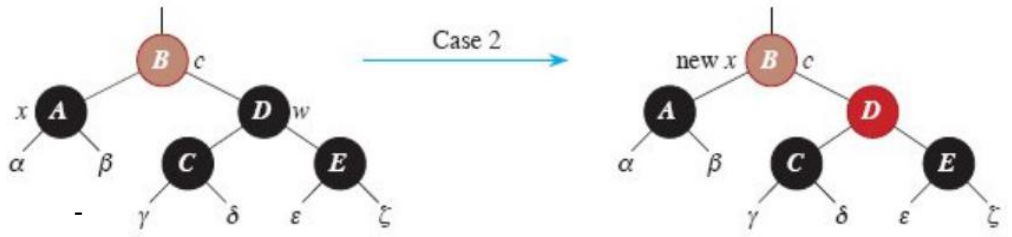
Case 1. x's sibling w is **red**

- Color w black
- Color x's parent red
- Left rotate x's parent
- Update w = right child of x's parent



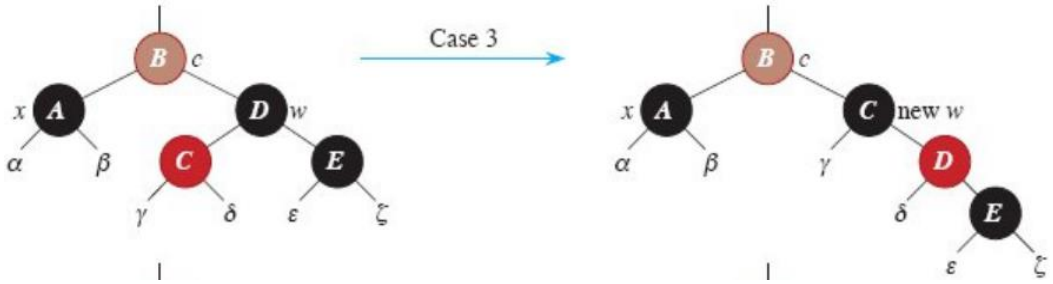
Case 2. x's sibling w is **black**, and both of w's children are **black**

- Color w red
- Update x = parent of x



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

- Color w's left child black
- Color w red
- Right rotate w
- Update w = right child of x's parent



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

- Color w as x's parent
- Color w's right child black
- Left rotate parent of x
- Update x = root of the tree

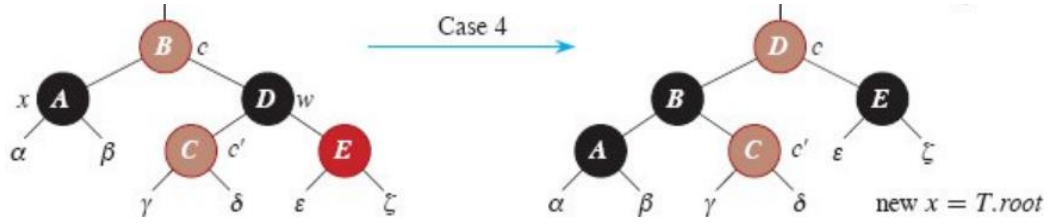


Figure 8: Deletion-fixup cases.

- Finally, color x to black.