CSE - 5311 Advanced Algorithms

Instructor : Nomaan Mufti

Submitted by

Raja Rajeshwari Anugula & Srujana Tiruveedhi

Contents

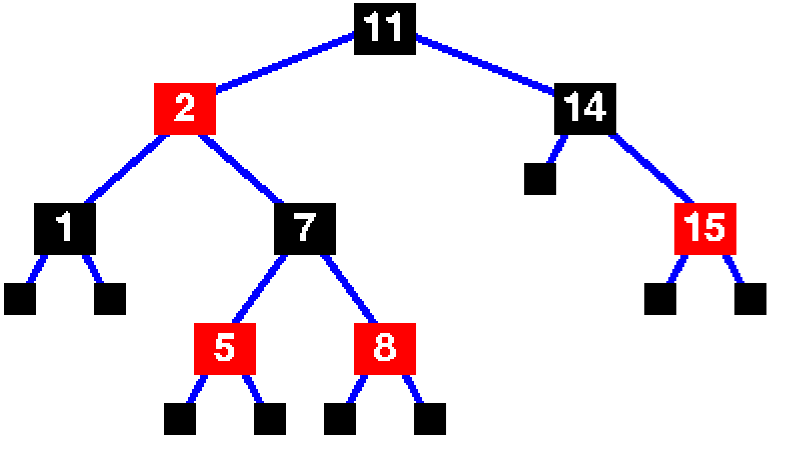
- RED – BLACK Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion

RED BLACK TREES

- A special class of binary trees that avoids the worst-case behavior of $O(n)$ that we can see in “plain” binary search trees
- Balanced: height is $O(\lg n)$, where n is the number of nodes
- Operations will take $O(\lg n)$ time in the worst case.
- Consist of one extra bit of storage per node; its color, which can be either RED or BLACK.
- Each node of the tree contains fields: color, key, left, right, parent.
- Red-Black Trees ensure that longest path is no more than twice as long as any other path

RB Tree Properties

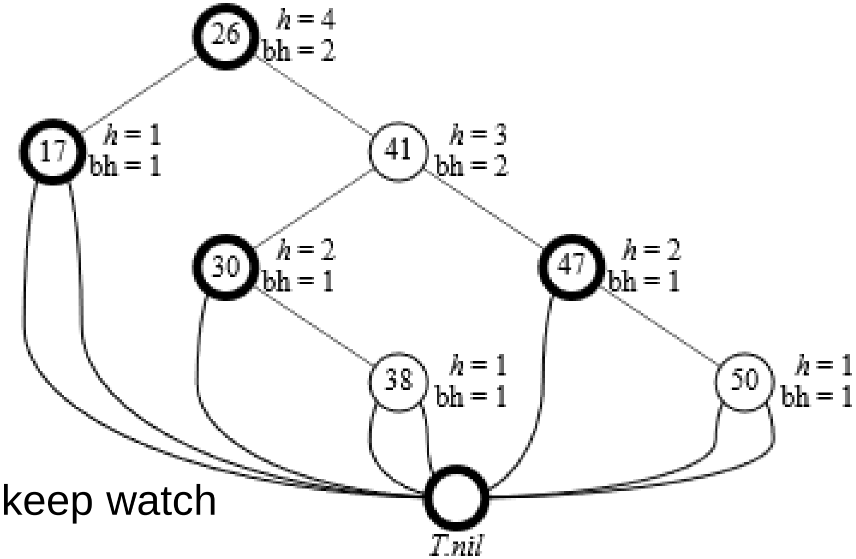
1. Every node is either red or black.
2. The root is black.
3. Every leaf (*T.nil*) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.



RB Trees

Points to remember:

- All leaves are empty (nil) and colored black.
- We may use a single sentinel, *T.nil*, for all the leaves of red-black tree T
- *T.nil.color* is black
- Root's parent is also *nil*
- All other attributes of binary search trees are inherited by red-black trees (*key*, *left*, *right*, and *p*)
- *key* of *T.nil* is not utilized



Sentinel - A soldier or guard whose job is to stand and keep watch

A node with a special value

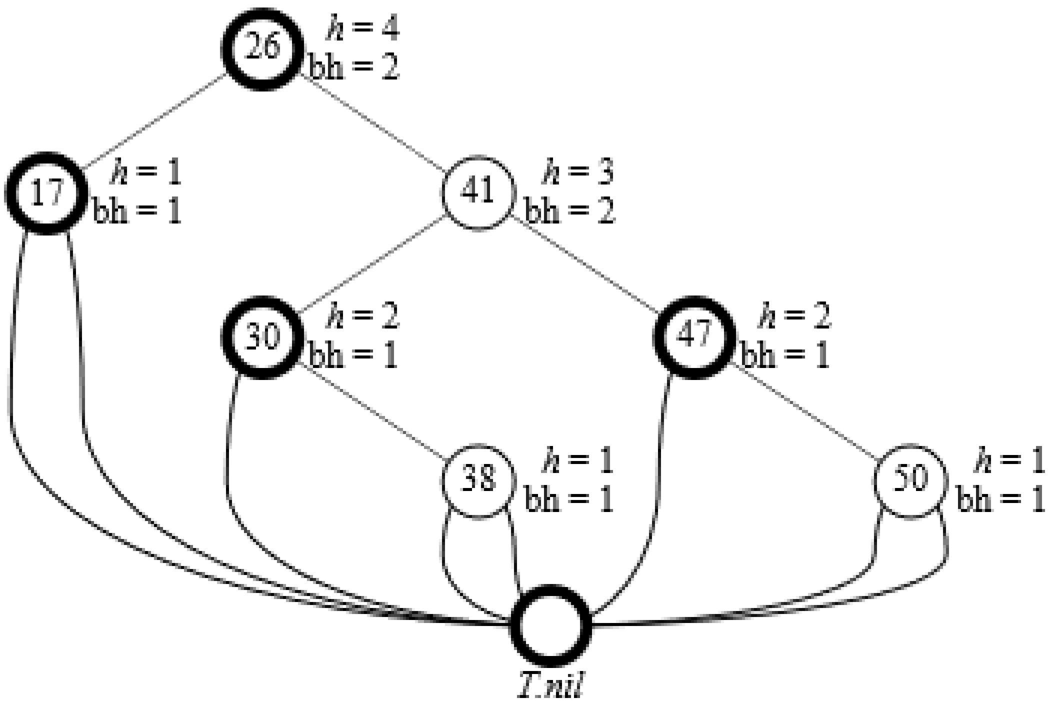
RB Trees

More points to remember:

- A RB tree with n internal nodes has a height of almost $2\lg(n+1)$
- Maximum path length is $O(\lg n)$
- Finding an element is real quick in RB trees, i.e., it takes $O(\lg n)$ time
- Insertion and Deletion take $O(\lg n)$ time

RB Height

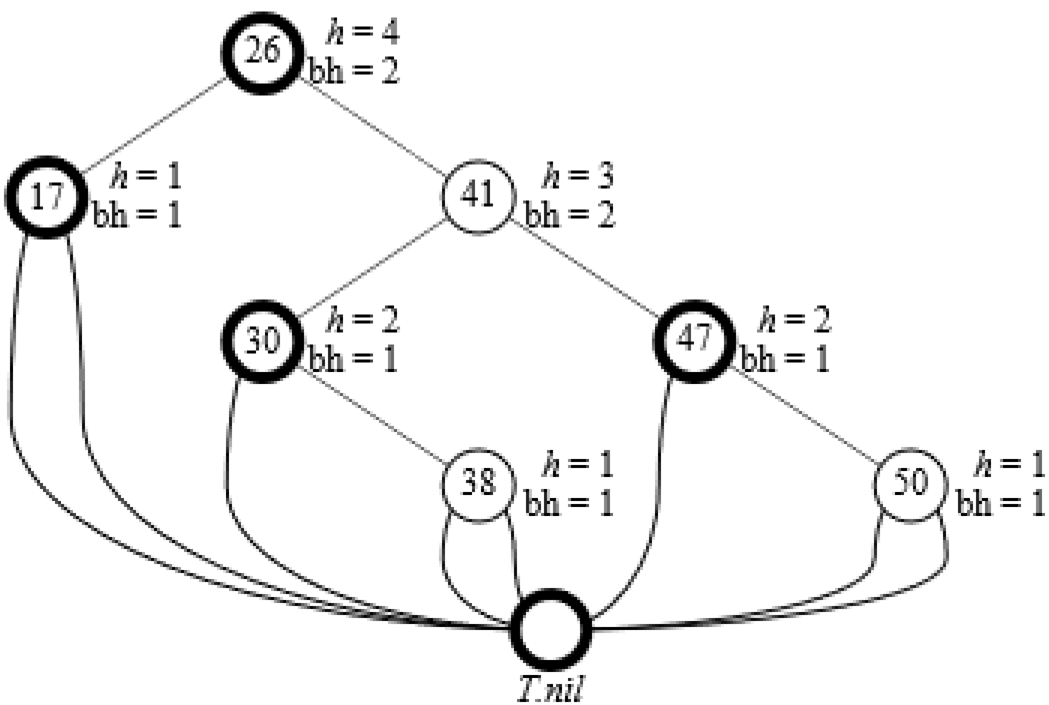
- **Height of a node** is the number of edges in a longest path to a leaf
- **Black-height** of a node x, $\mathrm{bh}(x)$, is the number of black nodes (including $T.nil$) on the path from x to leaf, not counting x
  (see also Property 5 above)



RB Height Claim 1

Any node with height h has black-height $\geq h/2$

Proof: By Property 4, from the node to a leaf $\leq h/2$ nodes are red
Hence more than $h/2$ are black nodes to the leaf



RB Height Claim 2

A subtree rooted at any node x contains $2^{\mathrm{bh}(x)} - 1$ internal nodes

Proof: By induction on the height of x

- Height of $x = 0$ is a leaf $\Longrightarrow \mathrm{bh}(x) = 0$. The subtree rooted at x has zero internal nodes $\Rightarrow 2^0 - 1 = 0$
- Let the height of x by h and $\mathrm{bh}(x) = b$. Any child of x has height $h - 1$ and black height is b (if the child is red) or $b - 1$ (if the child is black)

RB Height Claim 2

A subtree rooted at any node x contains $\geq 2^{\mathrm{bh}(x)} - 1$ internal nodes

Proof: By induction on the height of x (continues)

- By the inductive hypothesis, each child has $\geq 2^{b-1} - 1$ internal nodes
- Thus, the subtree rooted at x contains $\geq 2\left(2^{b-1} - 1\right) + 1 = 2^{b} - 1$ internal nodes. (two times for children and $+1$ for x itself)
- Thus, x contains $\geq 2^{\mathrm{bh}(x)} - 1$

RB Height, Lemma

A RB tree with n internal nodes has height

$$h \leq 2\lg(n+1)$$

Proof: Let h and b be the height and black-height of the root, respectively. By the above two claims

$$n \geq 2^b - 1 \geq 2^{h/2} - 1$$

Adding 1 to RHS and LHS and then taking logs we get $\lg(n+1) \geq h/2$, which provides the proof

Operations on RB Trees

The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(\text{height})$ time. Thus, they take $O(\lg n)$ time on red-black trees.

Insertion and deletion are not so easy.

If we insert, what color to make the new node?

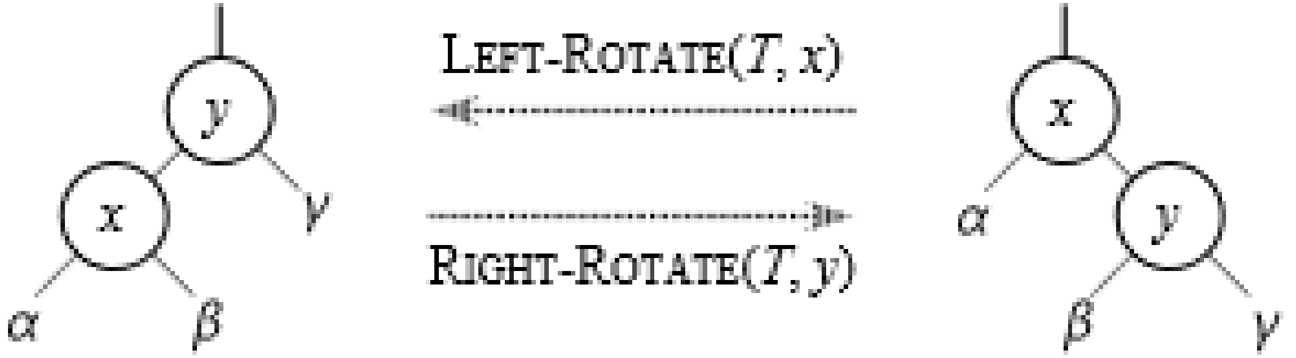
- Red? Might violate property 4.
- Black? Might violate property 5.

If we delete, thus removing a node, what color was the node that was removed?

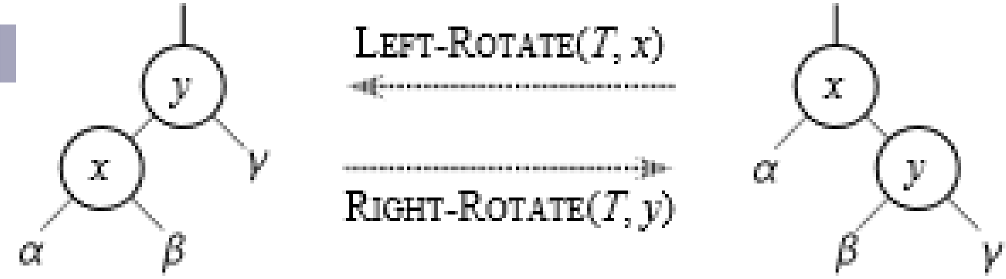
- Red? OK, since we won't have changed any black-heights, nor will we have created two red nodes in a row. Also, cannot cause a violation of property 2, since if the removed node was red, it could not have been the root.
- Black? Could cause there to be two reds in a row (violating property 4), and can also cause a violation of property 5. Could also cause a violation of property 2, if the removed node was the root and its child—which becomes the new root—was red.

Rotations

- **Insertion and Deletion modify the tree, the result may violate the properties of red black trees. To restore these properties rotations are conducted**
- **We can have either LEFT rotation or RIGHT rotation by which we must change colors of some of the nodes in the tree and also change the pointer structure**.



Rotations



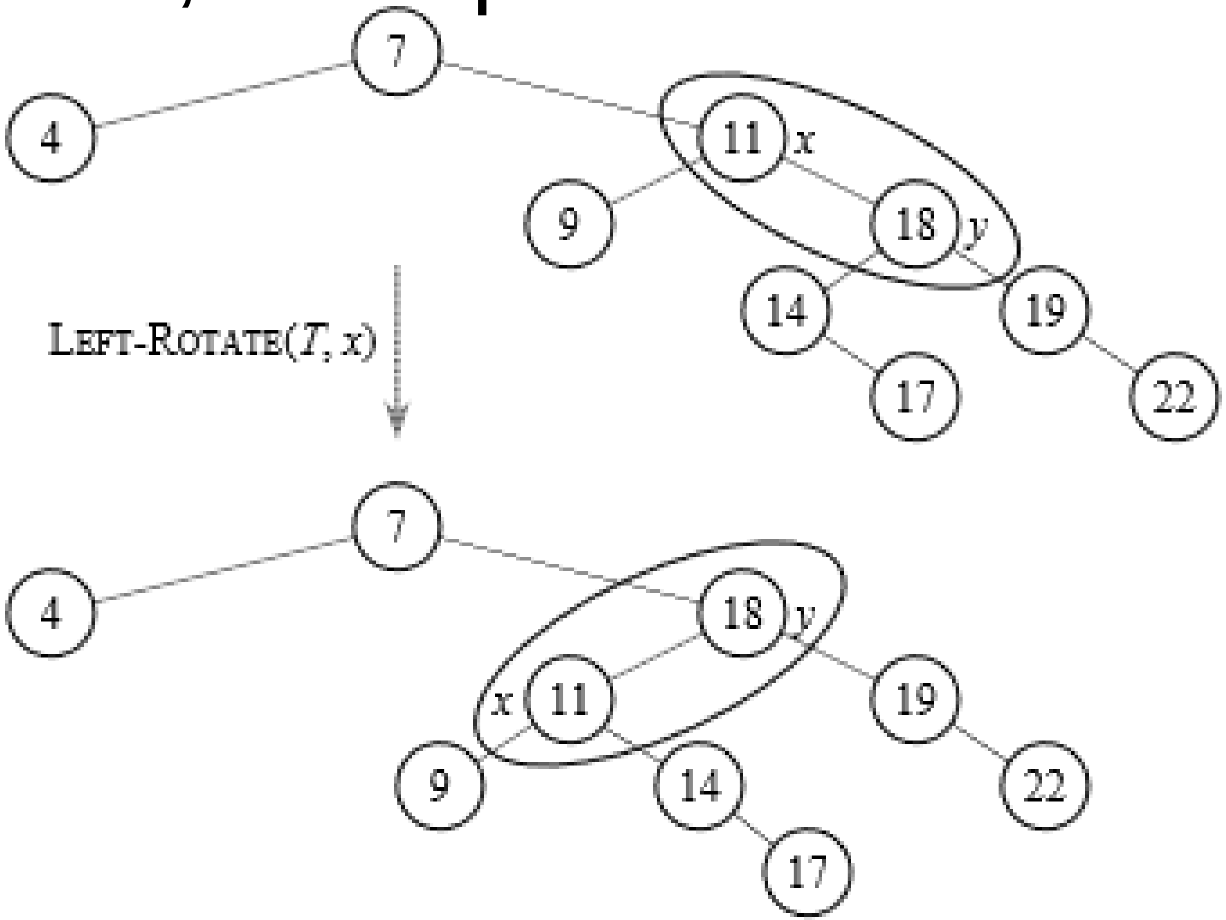
LEFT-ROTATE(T, x)

```
y = x.right                    // set y
x.right = y.left               // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                      // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                     // put x on y's left
x.p = y
```

The pseudocode for LEFT-ROTATE assumes that

- $x.right \neq T.nil$, and
- root's parent is $T.nil$.

Rotations, Example

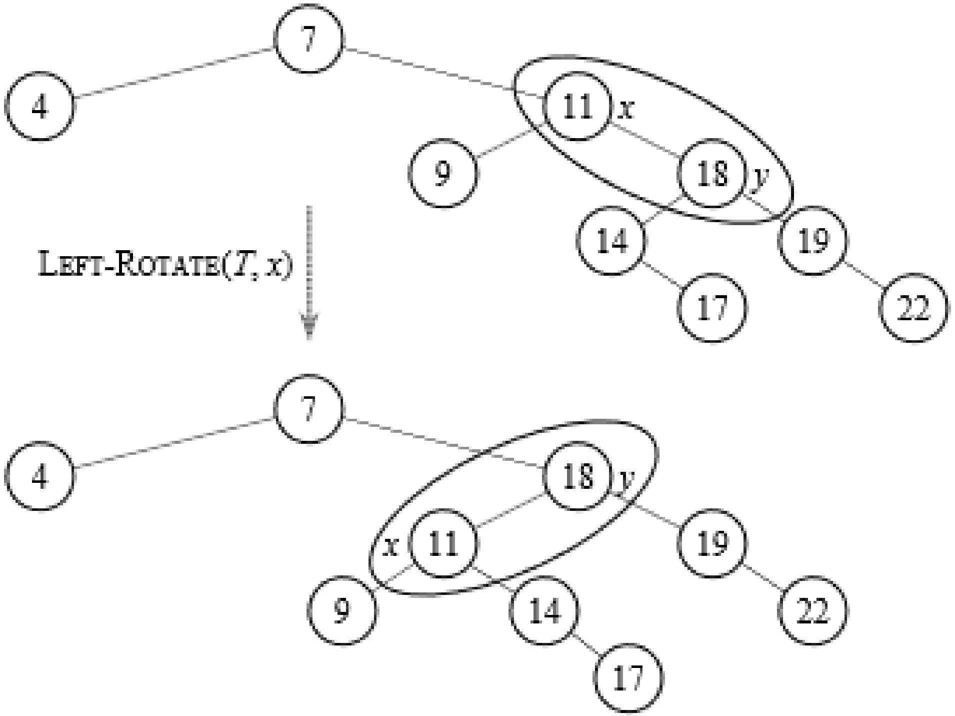


- Before rotation: keys of x's left subtree $\leq 11 \leq$ keys of y's left subtree $\leq 18 \leq$ keys of y's right subtree.
- Rotation makes y's left subtree into x's right subtree.
- After rotation: keys of x's left subtree $\leq 11 \leq$ keys of x's right subtree $\leq 18 \leq$ keys of y's right subtree.

Rotations

- **When we do a LEFT rotation on a node x we assume that its right child y is not nil, i.e x may be any node in the tree whose right child is not Nil**

- **It makes y the new root of the sub tree, with x as y's left child and y's left child as x's right child**



Time

$O(1)$ for both LEFT-ROTATE and RIGHT-ROTATE, since a constant number of pointers are modified.

Insertion

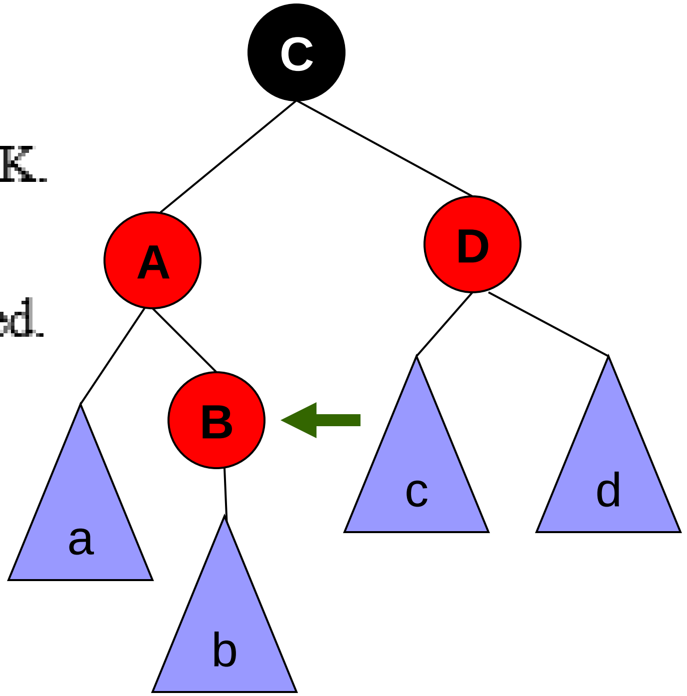
```
RB-INSERT(T, z)
  y = T.nil
  x = T.root
  while x ≠ T.nil
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == T.nil
      T.root = z
  elseif z.key < y.key
      y.left = z
  else y.right = z
  z.left = T.nil
  z.right = T.nil
  z.color = RED
  RB-INSERT-FIXUP(T, z)
```

- RB-INSERT ends by coloring the new node z red.
- Then it calls RB-INSERT-FIXUP because we could have violated a red-black property

1. Every node is either red or black.
2. The root is black.
3. Every leaf ($T.nil$) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Which Properties are Violated?

1. OK.
2. If z is the root, then there's a violation. Otherwise, OK.
3. OK.
4. If $z.p$ is red, there's a violation: both z and $z.p$ are red.
5. OK.



Removing the Violations

```
RB-INSERT-FIXUP(T, z)

while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                        // case 1
            y.color = BLACK                          // case 1
            z.p.p.color = RED                        // case 1
            z = z.p.p                                // case 1
        else if z == z.p.right
                z = z.p                              // case 2
                LEFT-ROTATE(T, z)                    // case 2
            z.p.color = BLACK                        // case 3
            z.p.p.color = RED                        // case 3
            RIGHT-ROTATE(T, z.p.p)                   // case 3
    else (same as then clause with "right" and "left" exchanged)
T.root.color = BLACK
```

Insertion, Discussion

At the start of each iteration of the **while** loop,

a. z is red.

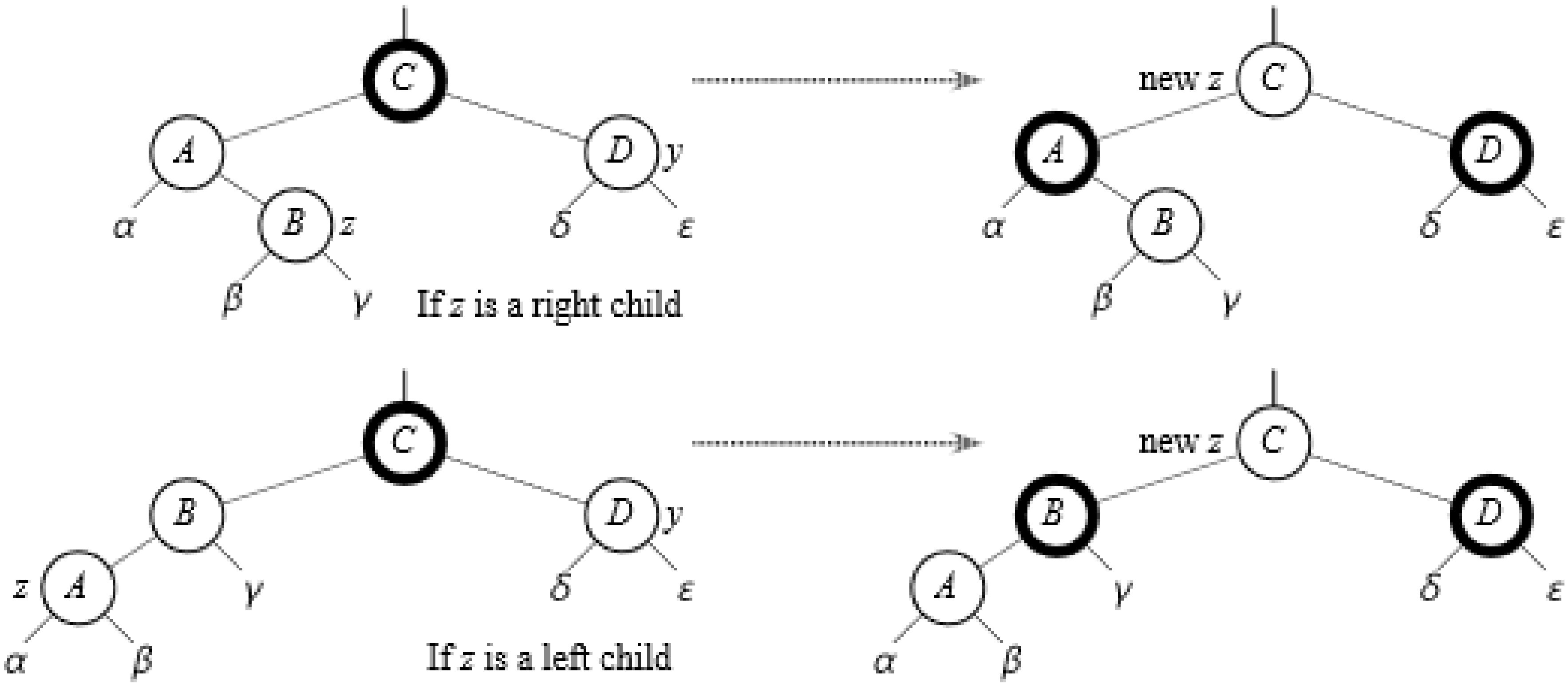
b. There is at most one red-black violation:

- Property 2: z is a red root, or
- Property 4: z and $z.p$ are both red.

There are 6 cases, 3 of which are symmetric to the other 3. The cases are not mutually exclusive. We'll consider cases in which $z.p$ is a left child.

Let y be z's uncle ($z.p$'s sibling).

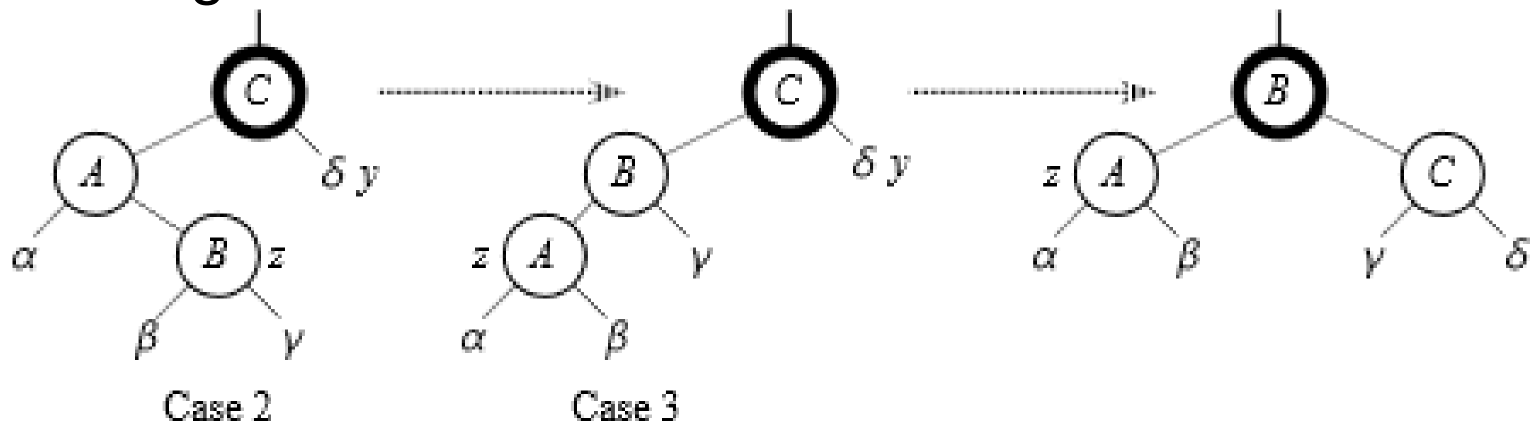
Insertion, Case 1: y is red



- $z.p.p$ (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 $\Rightarrow$ now z and $z.p$ are not both red. But property 5 might now be violated.
- Make $z.p.p$ red $\Rightarrow$ restores property 5.
- The next iteration has $z.p.p$ as the new z (i.e., z moves up 2 levels).

Insertion, Case 2 & 3: y is black

- z is a right child



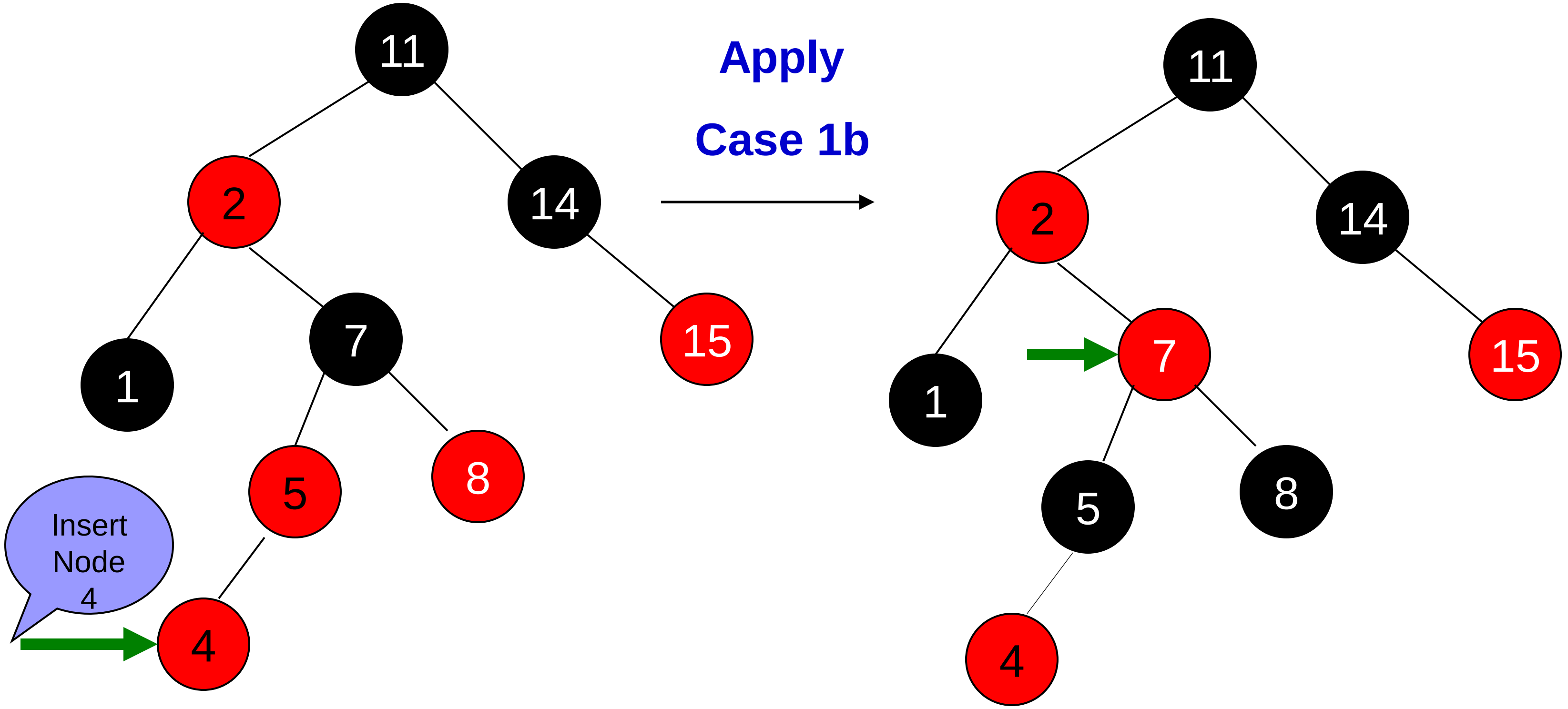
  - Left rotate around $z.p \Rightarrow$ now z is a left child, and both z and $z.p$ are red.
  - Takes us immediately to case 3.

- z is a left child

  - Make $z.p$ black and $z.p.p$ red.
  - Then right rotate on $z.p.p$.
  - No longer have 2 reds in a row.
  - $z.p$ is now black $\Rightarrow$ no more iterations.

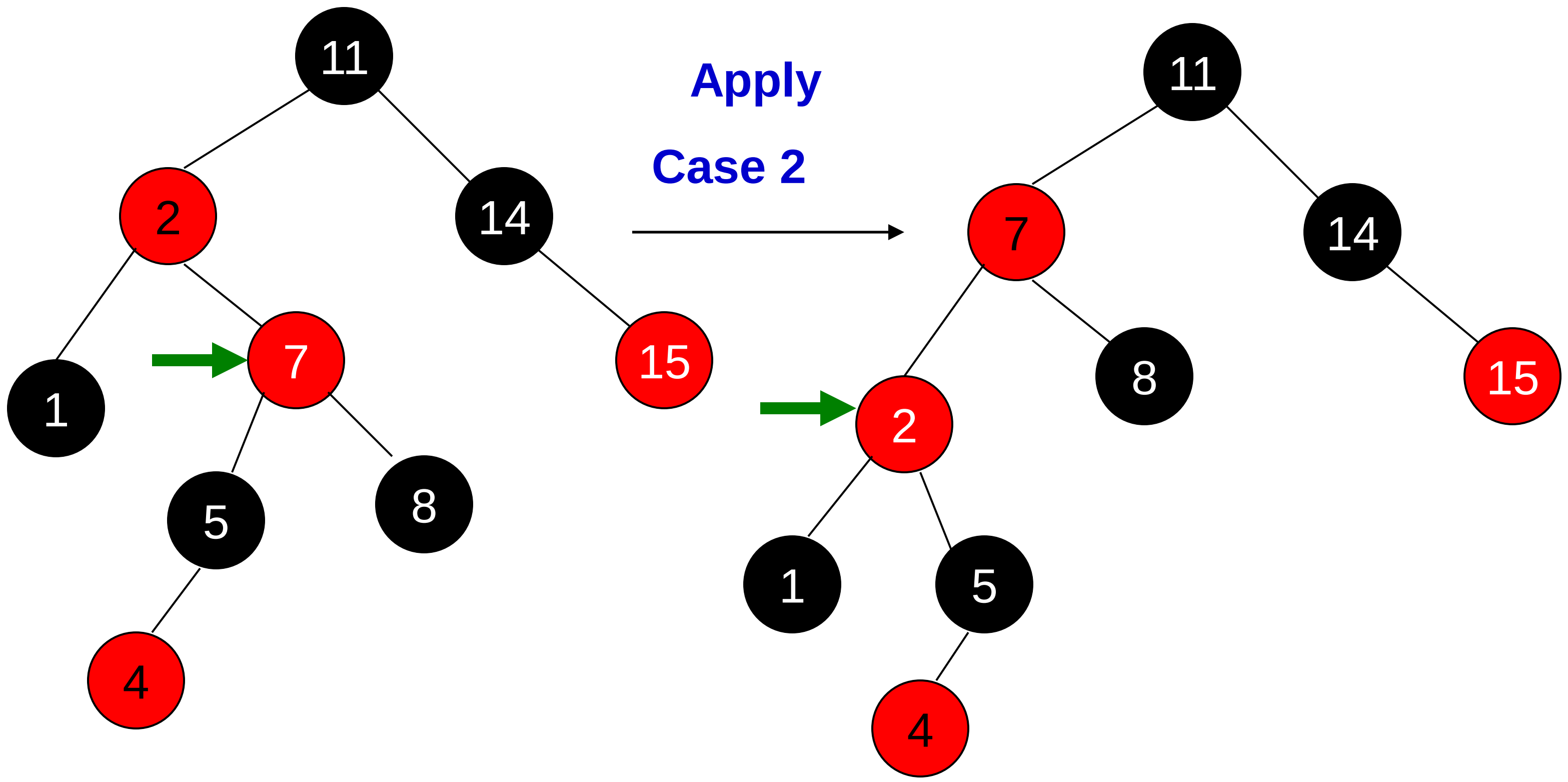
RED BLACK TREES

Example:



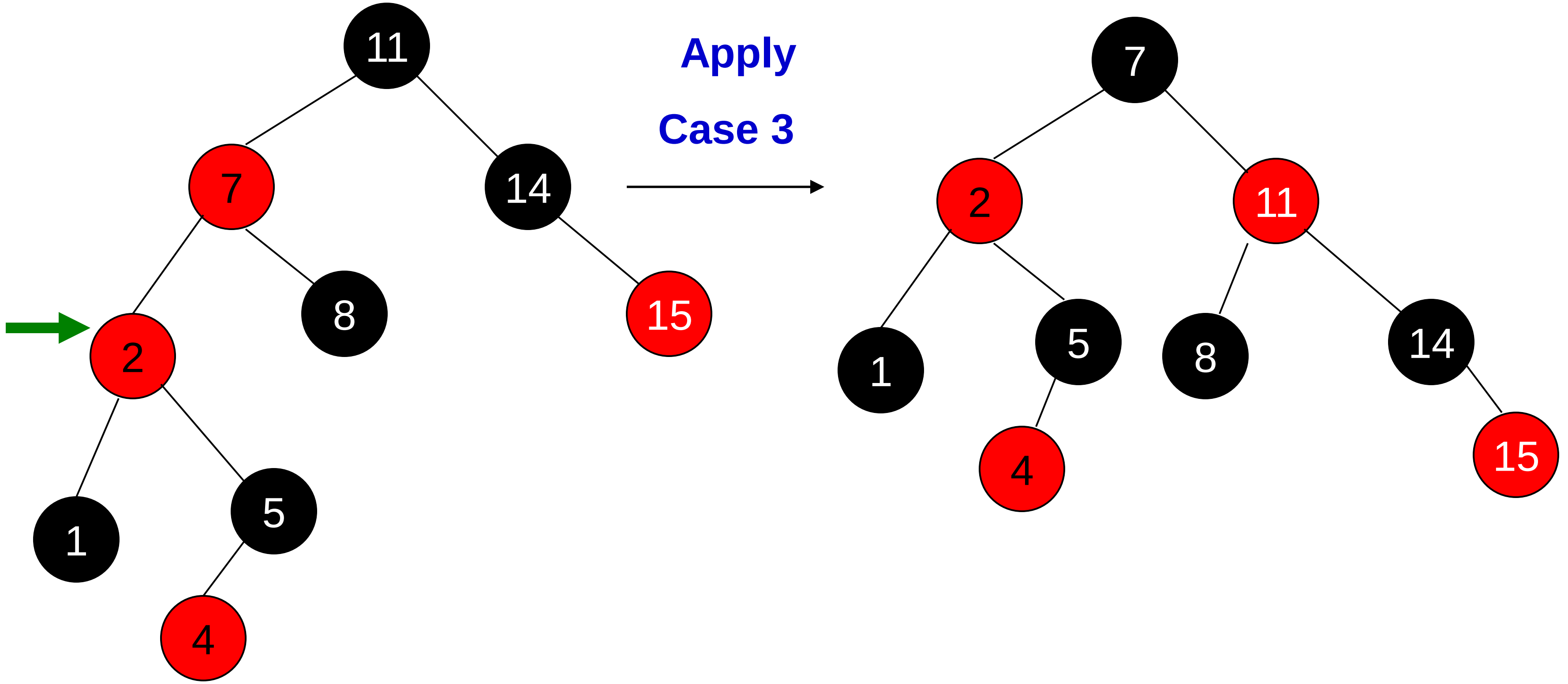
RED BLACK TREES

Example:



RED BLACK TREES

Example:



Insertion, 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 is either the last one or it moves z up 2 levels.
- $O(\lg n)$ levels $\Rightarrow O(\lg n)$ time.
- Also note that there are at most 2 rotations overall.

Thus, insertion into a red-black tree takes $O(\lg n)$ time.

Deletion

- RB-DELETE calls a RB-TRANSPLANT procedure.
- Then it calls RB-DELTE-FIXUP because we could have violated a red-black property
- TREE-MINIMUM returns the minimum key (the minimum key of a binary search tree is located at the leftmost node)
- y or x is the successor of z

```
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)
```

RB-DELETE: Discussion

- y is the node either removed from the tree (when z has fewer than 2 children) or moved within the tree (when z has 2 children).
- x is the node that moves into y's original position. It's either y's only child, or $T.nil$ if y has no children.
- If y's original color was black, the changes to the tree structure might cause red-black properties to be violated, and we call RB-DELETE-FIXUP at the end to resolve the violations.

RB-Transplant

RB-TRANSPLANT(T, u, v)

if $u.p == T.nil$
  $T.root = v$
elseif $u == u.p.left$
  $u.p.left = v$
else $u.p.right = v$
$v.p = u.p$

RB-TRANSPLANT replaces the subtree rooted at u by the subtree rooted at v:

- Makes u's parent become v's parent (unless u is the root, in which case it makes the root)
- u's parent gets v as either its left or right child, depending on whether u was a left or right child
- Doesn't update $v.left$ or $v.right$

RB-DELETE: Violations

If y was originally black, what violations of red-black properties could arise?

1. No violation.
2. If y is the root and x is red, then the root has become red.
3. No violation.
4. Violation if $x.p$ and x are both red.
5. Any simple 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.

Remove the violations by calling RB-DELETE-FIXUP:

RB-DELETE-FIXUP

```
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.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 = x.p                                          // case 2
          else if w.right.color == BLACK
                  w.left.color = BLACK                         // case 3
                  w.color = RED                                // case 3
                  RIGHT-ROTATE(T, w)                           // case 3
                  w = x.p.right                                // case 3
              w.color = x.p.color                              // case 4
              x.p.color = BLACK                                // case 4
              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
```

RB-DELETE-FIXUP, The Idea

Move the extra black up the tree until

- x points to a red & black node $\Rightarrow$ turn it into a black node,
- x points to the root $\Rightarrow$ just remove the extra black, or
- we can do certain rotations and recolorings and finish.

Within the **while** loop:

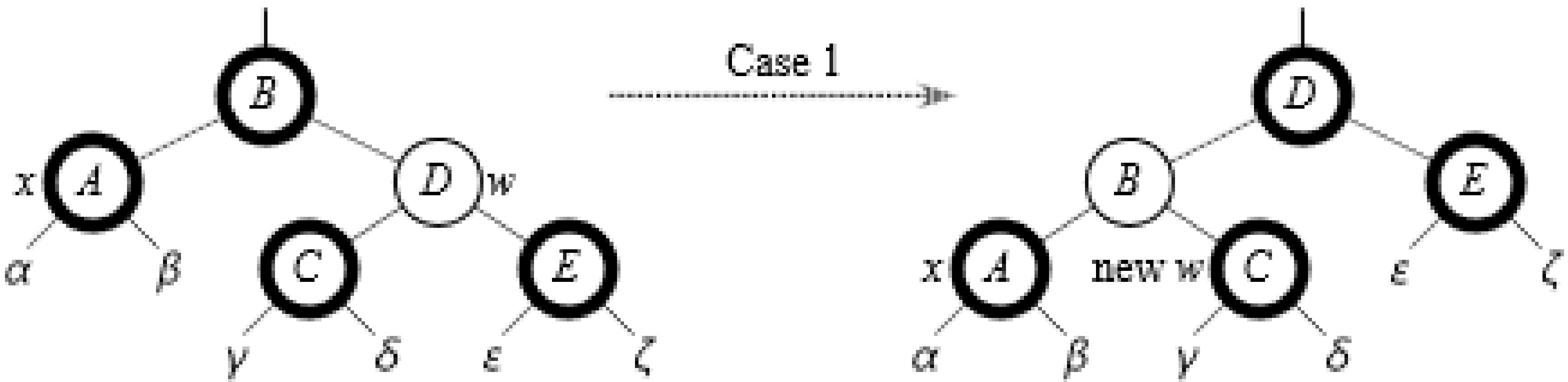
- x always points to a nonroot doubly black node.
- w is x's sibling.
- w cannot be $T.nil$, since that would violate property 5 at $x.p$.

There are 8 cases, 4 of which are symmetric to the other 4. As with insertion, the cases are not mutually exclusive. We'll look at cases in which x is a left child.

Note:
When a black node is deleted and replaced by a black child, the child is marked as doubly black
When a black node is deleted and replaced by a red child (or vice verse), the child is marked as red & black node

RB-DELETE-FIXUP, Case 1

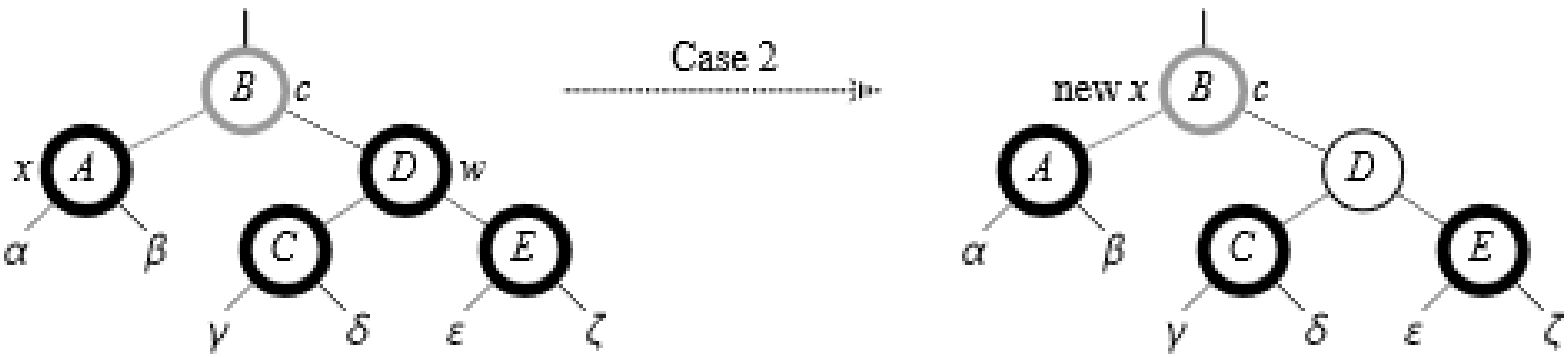
Case 1: w is red



- w must have black children.
- Make w black and $x.p$ red.
- Then left rotate on $x.p$.
- New sibling of x was a child of w before rotation $\Rightarrow$ must be black.
- Go immediately to case 2, 3, or 4.

RB-DELETE-FIXUP, Case 2

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

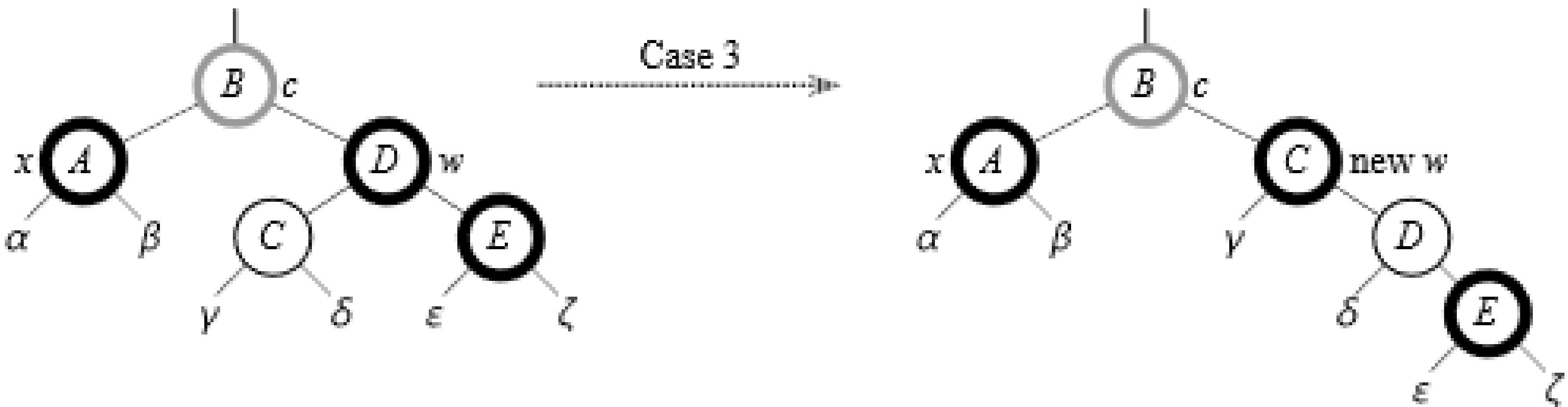


[Node with gray outline is of unknown color, denoted by c.]

- Take 1 black off x ($\Rightarrow$ singly black) and off w ($\Rightarrow$ 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 $\Rightarrow$ new x is red & black $\Rightarrow$ color attribute of new x is RED $\Rightarrow$ loop terminates. Then new x is made black in the last line.

RB-DELETE-FIXUP, Case 3

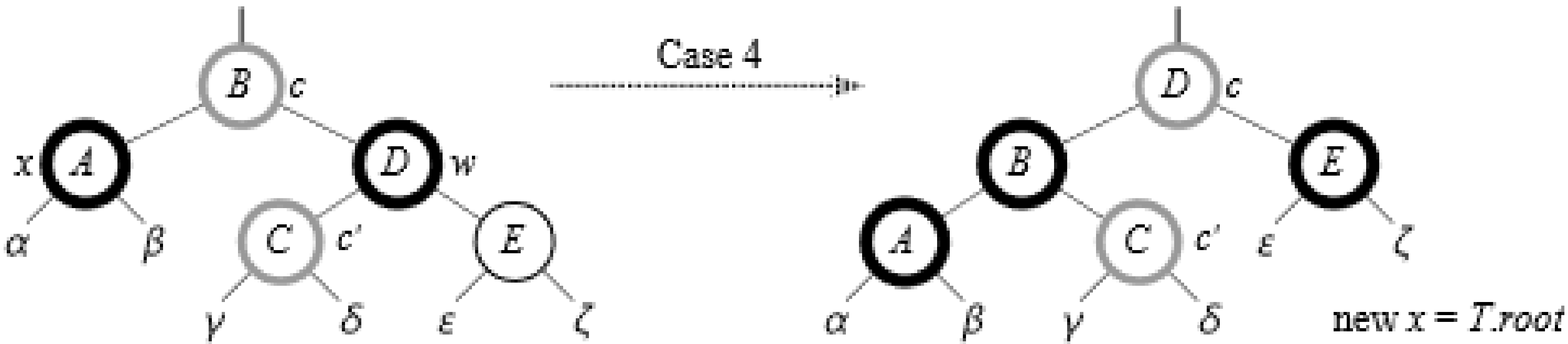
Case 3: w is black, w's left child is red, and w's right child is black



- 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 $\Rightarrow$ case 4.

RB-DELETE-FIXUP, Case 4

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



[Now there are two nodes of unknown colors, denoted by c and c'.]

- 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 ($\Rightarrow$ x is now singly black) without violating any red-black properties.
- All done. Setting x to root causes the loop to terminate.

RB-DELETE, 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.

HW

Exercises

- 13.1-3, 13.1-4
- 13.2-4
- 13.3-3, 13.3-4
- 13.4-6

Backup Slides