Lecture 13
Red-Black Trees

Sultan ALPAR
associate professor, IITU
s.alpar@iitu.edu.kz

Red-black trees

- A variation of 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.

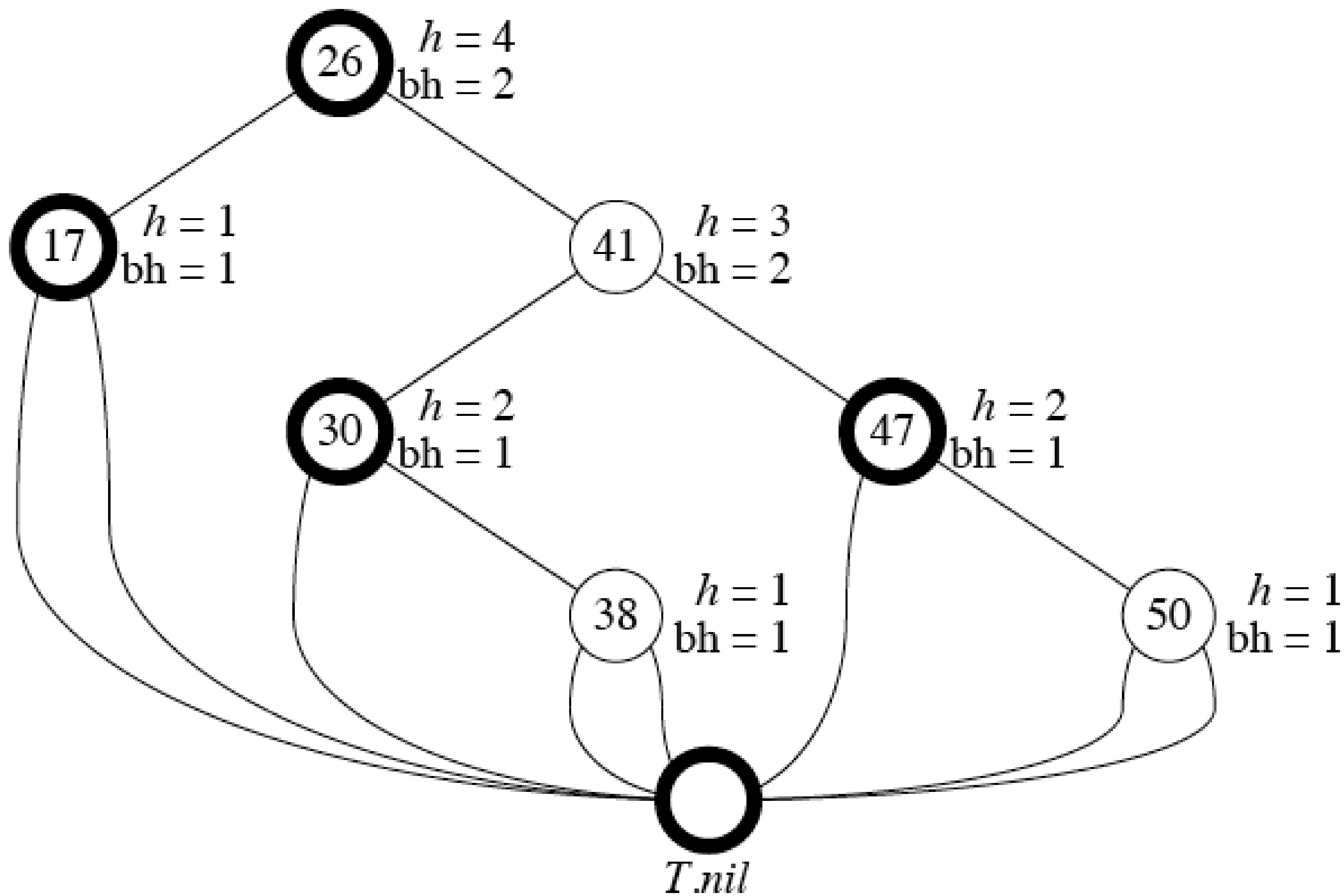
Red-black trees

- A ***red-black tree*** is a binary search tree + 1 bit per node: an attribute color, which is either red or black.
- All leaves are empty (nil) and colored black.
- We use a single sentinel, *T.nil,* for all the leaves of red-black tree *T* .
- *T.nil.color* is black.
- The root's parent is also *T.nil.*
- All other attributes of binary search trees are inherited by red-black trees (*key, left,right,* and *p).* We don't care about the *key* in *T.nil.*

Red-black 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.

Example



26
h = 4
bh = 2
17
h = 1
bh = 1
41
h = 3
bh = 2
30
h = 2
bh = 1
47
h = 2
bh = 1
38
h = 1
bh = 1
50
h = 1
bh = 1
T.nil

Height of a red-black tree

- ***Height of a node*** is the number of edges in a longest path to a leaf***.***
- ***Black-height*** of a node *x*: bh(*x*) is the number of black nodes (including *T.nil*) on the path from *x* to leaf, not counting *x*. By property 5, black-height is well defined.

Height of a red-black tree

- ***Claim***

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

- ***Proof*** By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red.

  Hence $\geq h/2$ are black.

Height of a red-black tree

- ***Claim***

  The subtree rooted at any node x contains $\geq 2^{bh(x)-1}$ internal nodes.
- ***Proof*** By induction on height of x.
- **Basis:** Height of $x = 0 \Rightarrow x$ is a leaf $\Rightarrow bh(x) = 0$. The subtree rooted at x has 0 internal nodes. $2^0 - 1 = 0$.
- **Inductive step:** Let the height of x be h and $bh(x) = b$. Any child of x has height h - 1 and black-height either b (if the child is red) or b -1 (if the child is black). By the inductive hypothesis, each child has $\geq 2^{bh(x)-1} - 1$ internal nodes.

  Thus, the subtree rooted at x contains $\geq 2 \cdot (2^{bh(x)-1} - 1) + 1$ internal nodes. (The +1 is for x itself.)

Height of a red-black tree

- ***Lemma***
- A red-black tree with n internal nodes has height $\leq 2 \lg(n + 1)$.
- ***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 both sides and then taking logs gives $\lg(n + 1) \geq h/2$, which implies that $h \leq 2 \lg(n + 1)$.

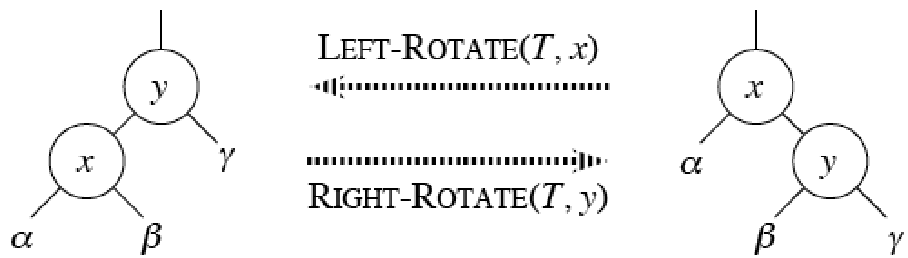
Operations on red-black trees

- The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in O(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.

Rotations

- The basic tree-restructuring operation.
- Needed to maintain red-black trees as balanced binary search trees.
- Changes the local pointer structure. (Only pointers are changed.)
- Won’t upset the binary-search-tree property.
- Have both left rotation and right rotation. They are inverses of each other.
- A rotation takes a red-black-tree and a node within the tree

Rotations

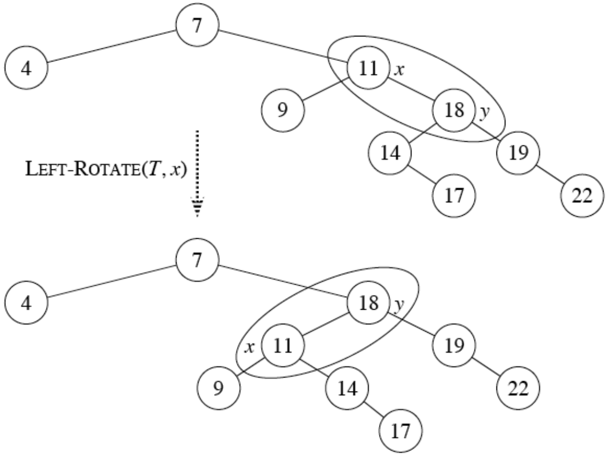



```
LEFT-ROTATE(T, x)
y = x.right                    // set y
x.right = y.left               // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                      // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else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

Rotations

- The pseudocode for LEFT-ROTATE assumes that
  - *x.right* ≠ *T.nil,* and
  - root's parent is *T.nil.*
- Pseudocode for RIGHT-ROTATE is symmetric: exchange *left* and *right* everywhere.

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.

Time

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

Insertion

- 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.
- Which property might be 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.

Insertion

- **Loop invariant:**
- At the start of each iteration of the **while loop,**

a. z is red.

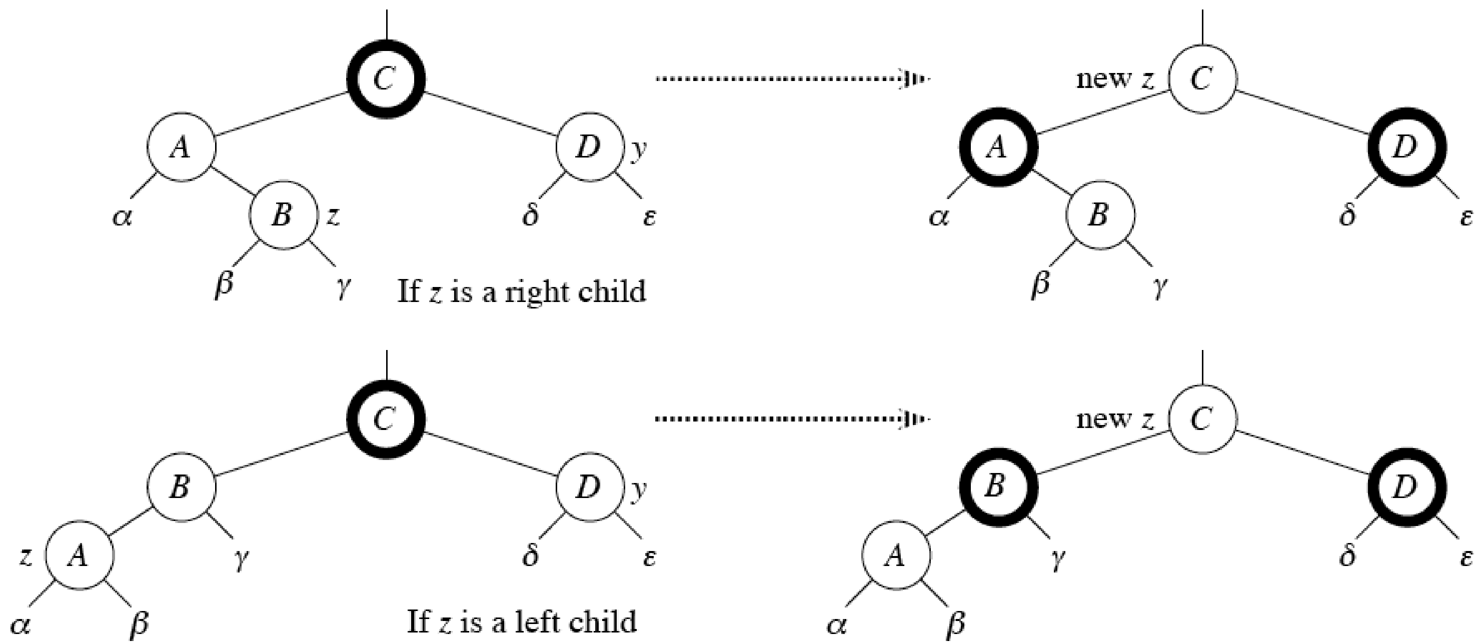
b. There is at most one red-black violation:
   - Property 2: z is a red root, or
   - Property 4: z and $z.p$ are both red.

Insertion

- **Initialization:** We've already seen why the loop invariant holds initially.
- **Termination:** The loop terminates because *z.p* is black. Hence, property 4 is OK**.** Only property 2 might be violated, and the last line fixes it.
- **Maintenance:** We drop out when *z* is the root (since then *z.p* is the sentinel *T.nil,* which is black). When we start the loop body, the only violation is of property 4.
- 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).

Loop invariant

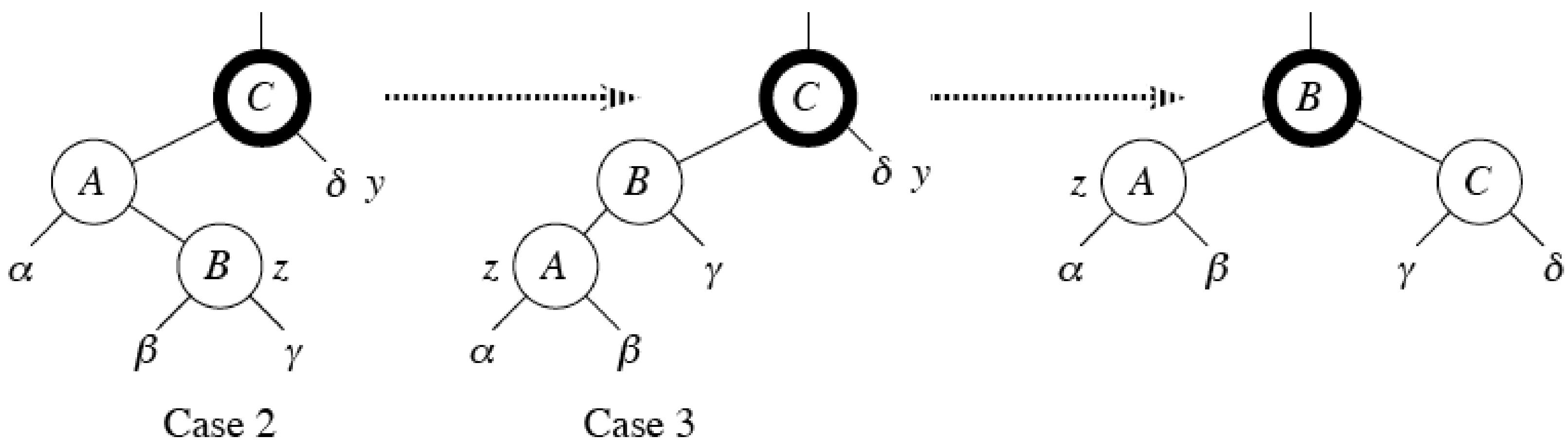
Case 1: y is red



- $z.p.p$ (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).

Loop invariant

Case 2: 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.

Loop invariant

Case 3: y is black, 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.

Analysis

$O(\lg n)$ time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.

Analysis

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.

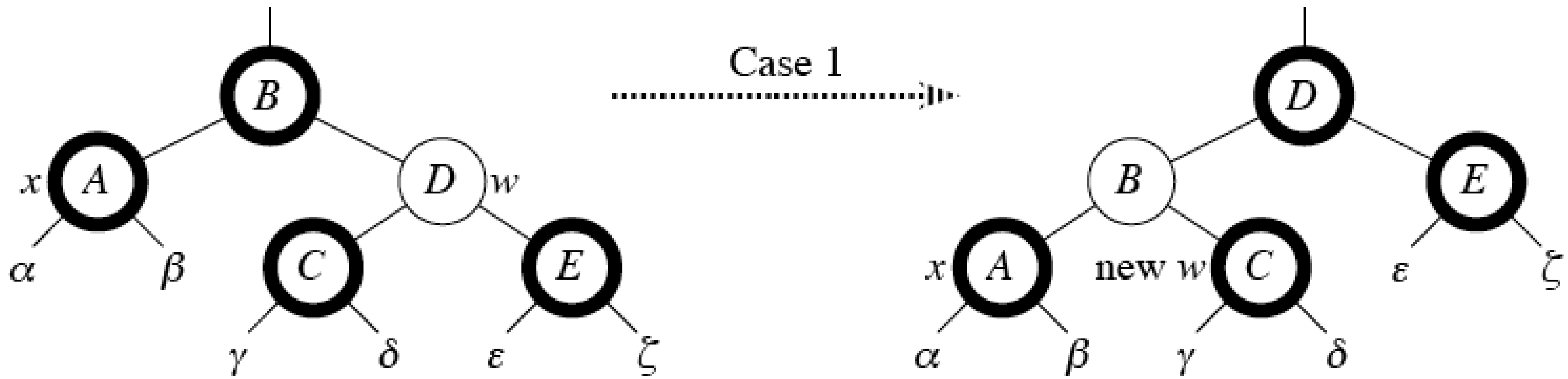
Thus, insertion into a red-black tree takes $O(\lg n)$ time.

Idea

- Move the extra black up the tree until
  - *x* points to a red & black node ⇒ turn it into a black node,
  - *x* points to the root ⇒ 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.

Case 1

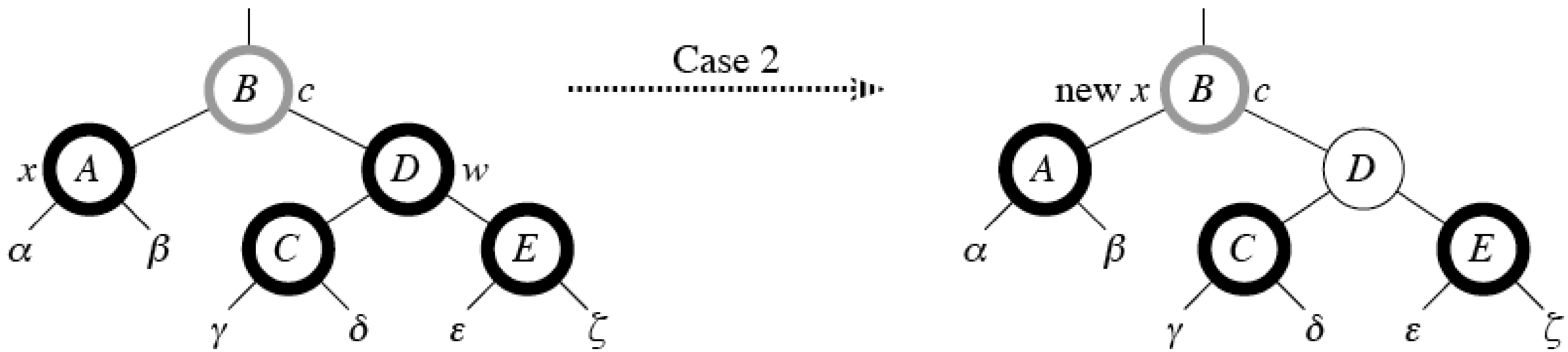
Case 1: w is red



- 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.

Case 2

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

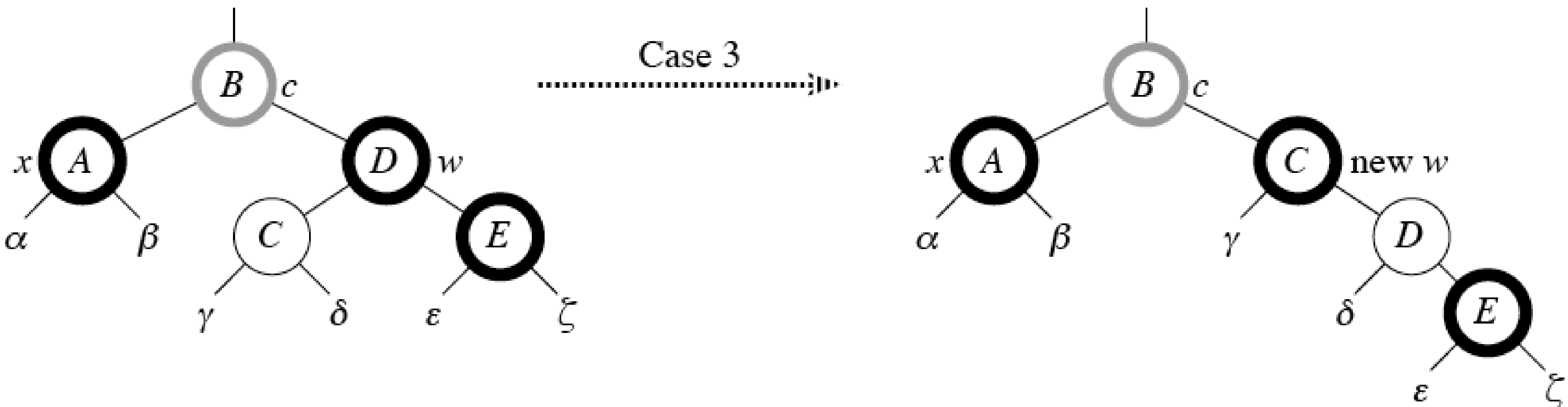


[Node with gray outline is of unknown color, denoted by c *.]*

- 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.

Case 3

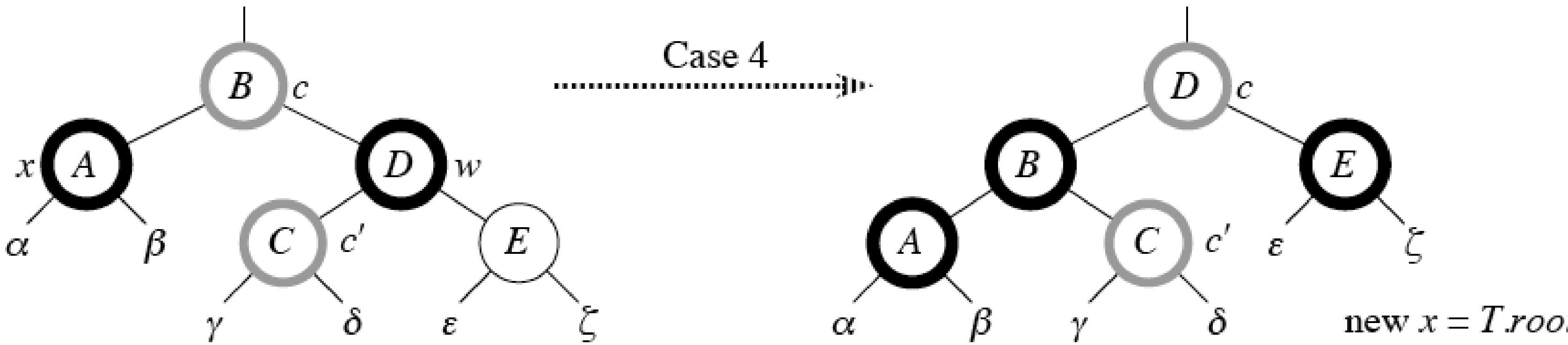
Case 3: w is black, w's left child is red, and w's right child is black



- 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 ⇒ case 4.

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.

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 ⇒ ≤ 3 rotations in all.
  - Hence, O(lg n) time.