CS 590 – Algorithms

Lecture 7 – Red-Black Trees

Outlines

7. Red-black trees (RBTs)
7.1. Characteristics and Properties
7.2. Black Heights
7.3. Operations
    7.3.1. Rotations
    7.3.2. Insertions
    7.3.3. Deletions

7. Red-black trees

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.

7.1. Properties

Red-black trees:

- A ***red-black tree*** is a self-balancing binary search tree with one extra (+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.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*).

7.1. Properties

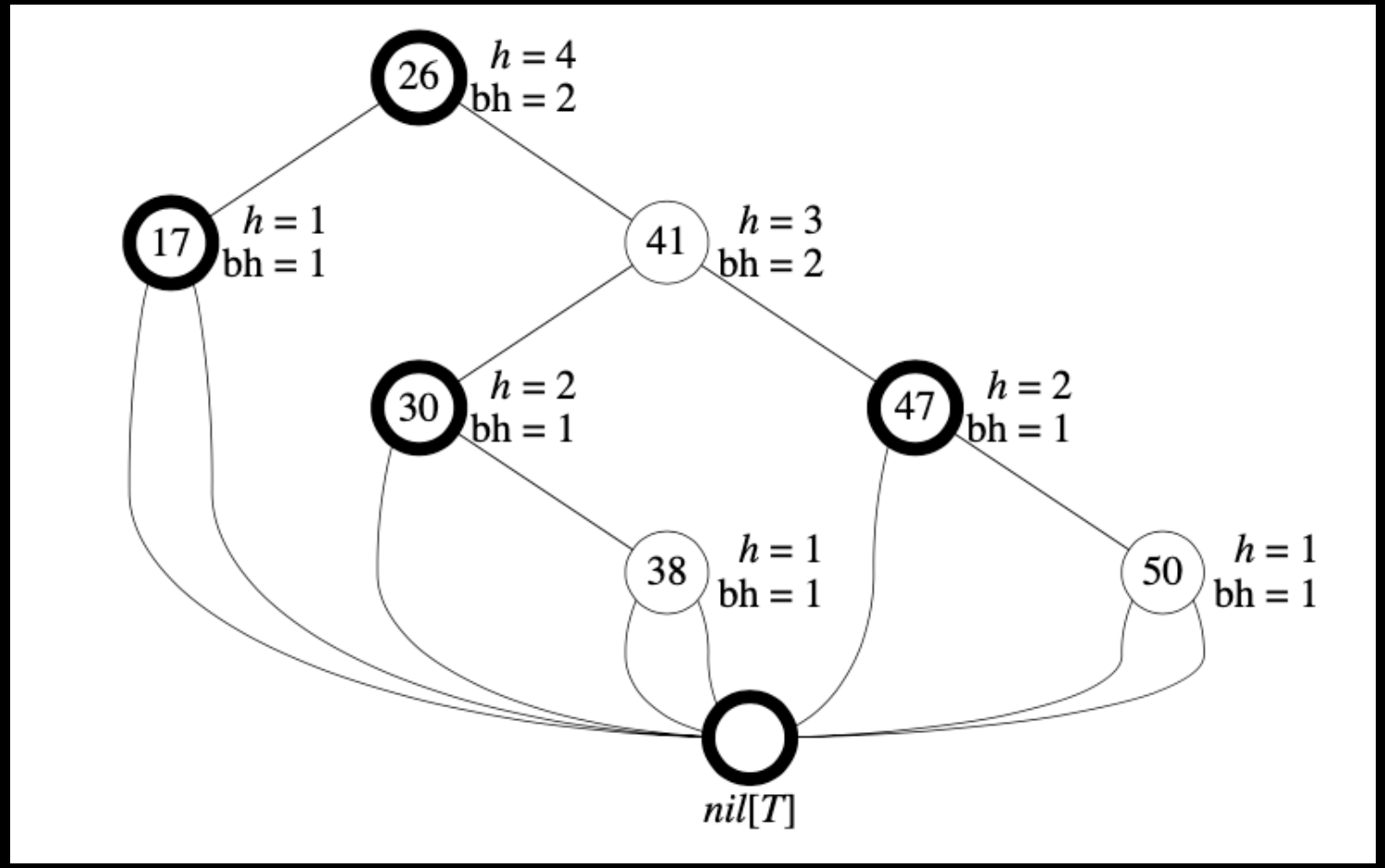
Red-black trees properties:

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.

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.

7.2. Black Heights



7.2. Black Heights

Claim

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

Claim

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

Lemma

A red-black tree with n internal nodes has height $\leq 2 \lg(n + 1)$.

7.3. Operations

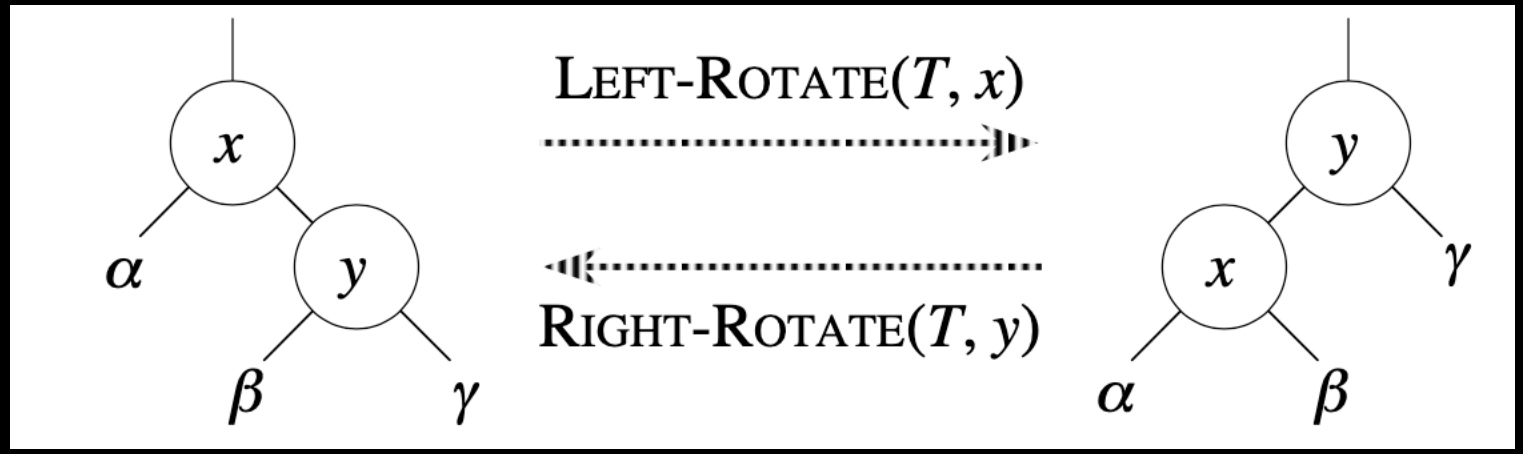
Operations on red-black trees

- The non-modifying binary-search-tree operations are unchanged.
- MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(h) = O(\lg n)$ 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. Does not violate property 4.
      - Cannot violate property 2 – root is black.
    - **Black**: Could cause two reds in a row (violating property 4), and
      - can cause a violation of property 5.
      - Possibly violate property 2, if the removed node was the root and its child was red.

7.3.1. Rotations

Rotations:

- The basic tree-restructuring operation.
- Used 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.



7.3.1. Rotations

```
Algorithm (LEFT-ROTATE(T,x))
(1)     y = x.right                  //set y
(2)     x.right = y.left             //turn left subtree of y into right of x
(3)     if (y.left ≠ T.nil) then
(4)         y.left.p = x
(5)     y.p = x.p                    //link parent of x to y
(6)     if (x.p = T.nil) then
(7)         T.root = y
(8)     else
(9)         if (x = x.p.left) then
(10)            x.p.left = y
(11)        else
(12)            x.p.right = y
(13)    y.left = x                   //put x of left of y
(14)    x.p = y
```

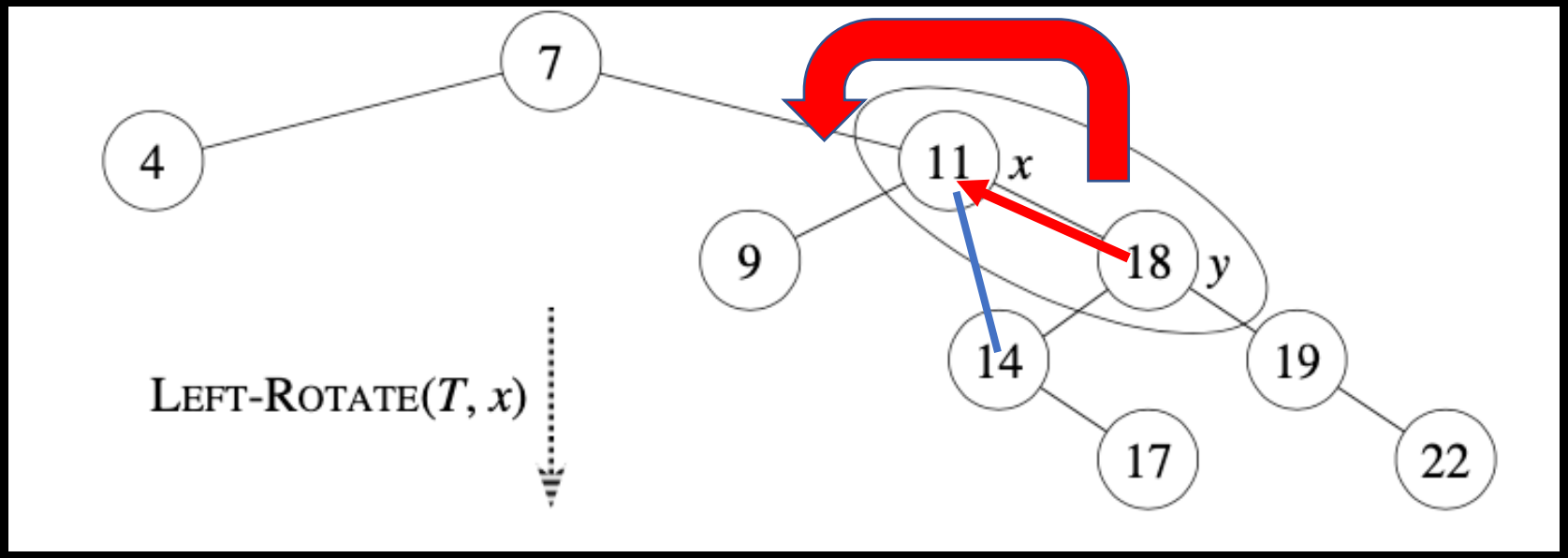
7.3.1. Rotations

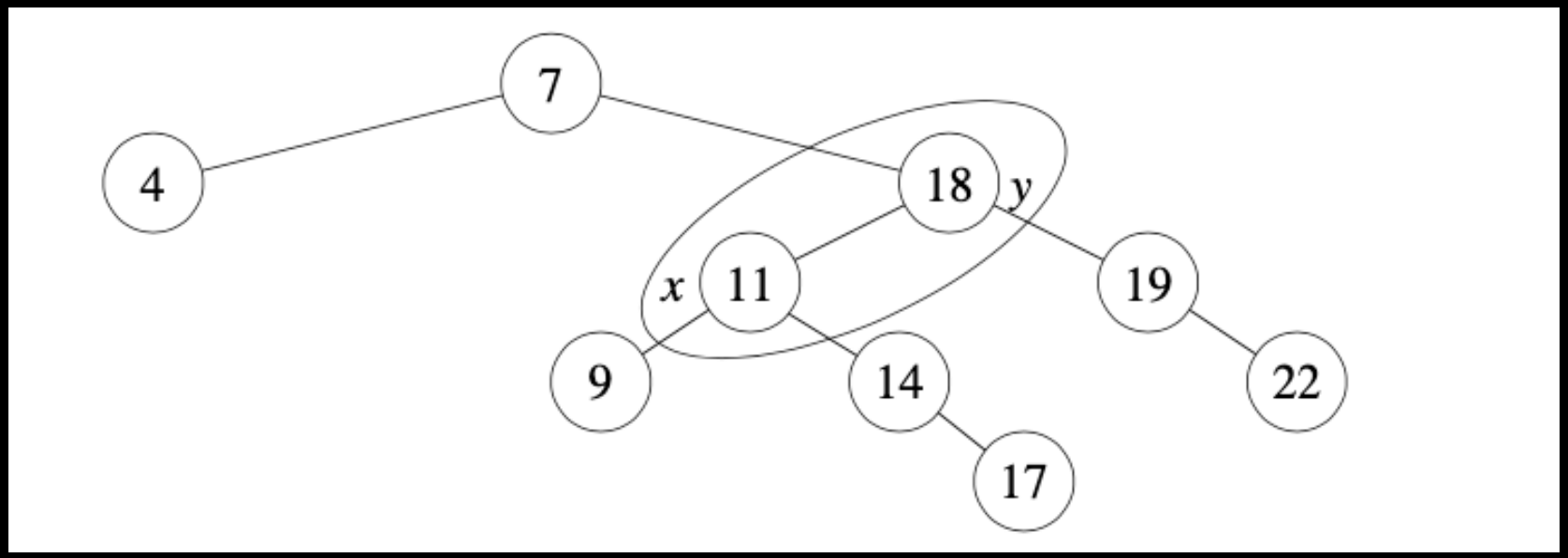
The pseudocode for LEFT-ROTATE assumes that

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

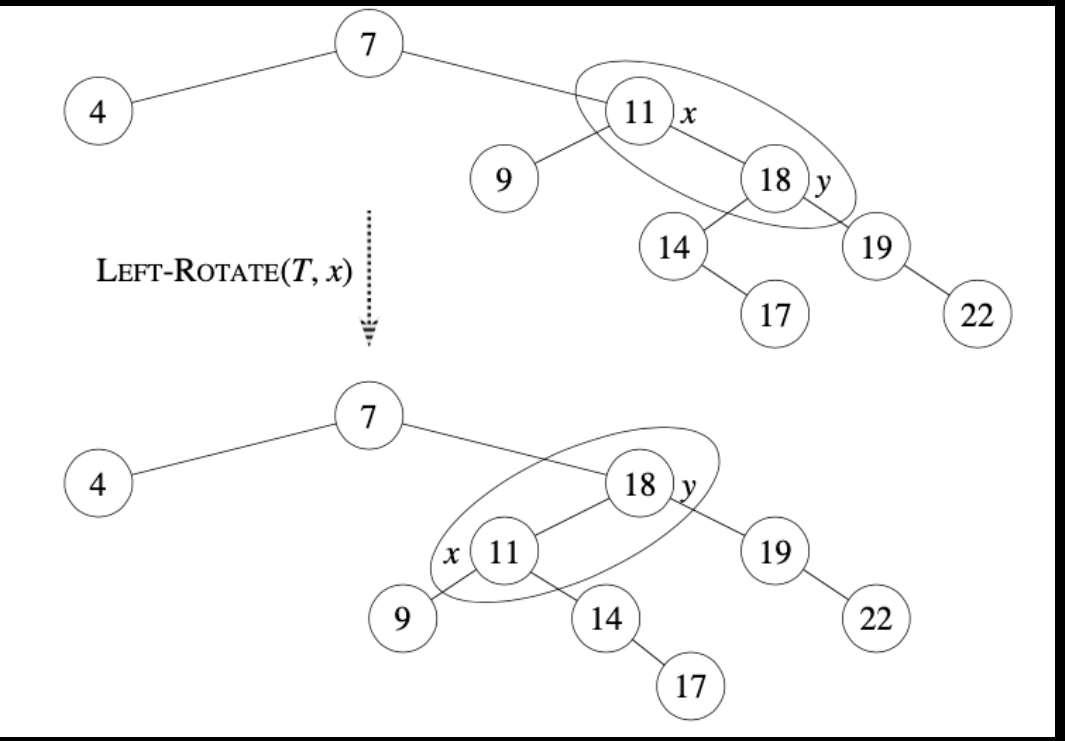
Pseudocode for RIGHT-ROTATE is symmetric: exchange *left* and *right* everywhere.

7.3.1. Rotations





7.3.1. Rotations



- 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.

Notes:

- Rotation is a very basic operation, also used in Adelson-Velskii, Landis (AVL – BST whose h of right and left subtrees are differ by 1) trees and splay trees (BST w/ no explicit balance condition).
- Some books talk of rotating on an edge rather than on a node.

7.3.2. Insertions

Algorithm (RB-**INSERT(T,x)**)

```
(1)     y = T.nil, x = T.root
(2)     while (x ≠ T.nil) do
(3)       y = x
(4)       if (z.key < x.key) then
(5)         x = x.left
(6)       else x = x.right
(7)     z.p = y
(8)     if (y = NIL) then
(9)       T.root = z
(10)    else if (z.key < y.key) then
(11)      y.left = z
(12)    else y.right = z
(13)    z.left = T.nil
(14)    z.right = T.nil
(15)    z.color = RED
(16)    RB-INSERT-FIXUP(T,z)
```

7.3.2. Insertions

Algorithm (**TREE-INSERT(T,x)**)

```
(1)    y = NIL, x = T.root
(2)    while (x ≠ NIL) do
(3)      y = x
(4)      if (z.key < x.key) then
(5)        x = x.left
(6)      else x = x.right
(7)    z.p = y
(8)    if (y = NIL) then
(9)      T.root = z
(10)   else if (z.key < y.key) then
(11)     y.left = z
(12)   else y.right = z
```

Algorithm (RB-**INSERT(T,x)**)

```
(1)    y = T.nil, x = T.root
(2)    while (x ≠ T.nil) do
(3)      y = x
(4)      if (z.key < x.key) then
(5)        x = x.left
(6)      else x = x.right
(7)    z.p = y
(8)    if (y = NIL) then
(9)      T.root = z
(10)   else if (z.key < y.key) then
(11)     y.left = z
(12)   else y.right = z
(13)   z.left = T.nil
(14)   z.right = T.nil
(15)   z.color = RED
(16)   RB-INSERT-FIXUP(T,z)
```

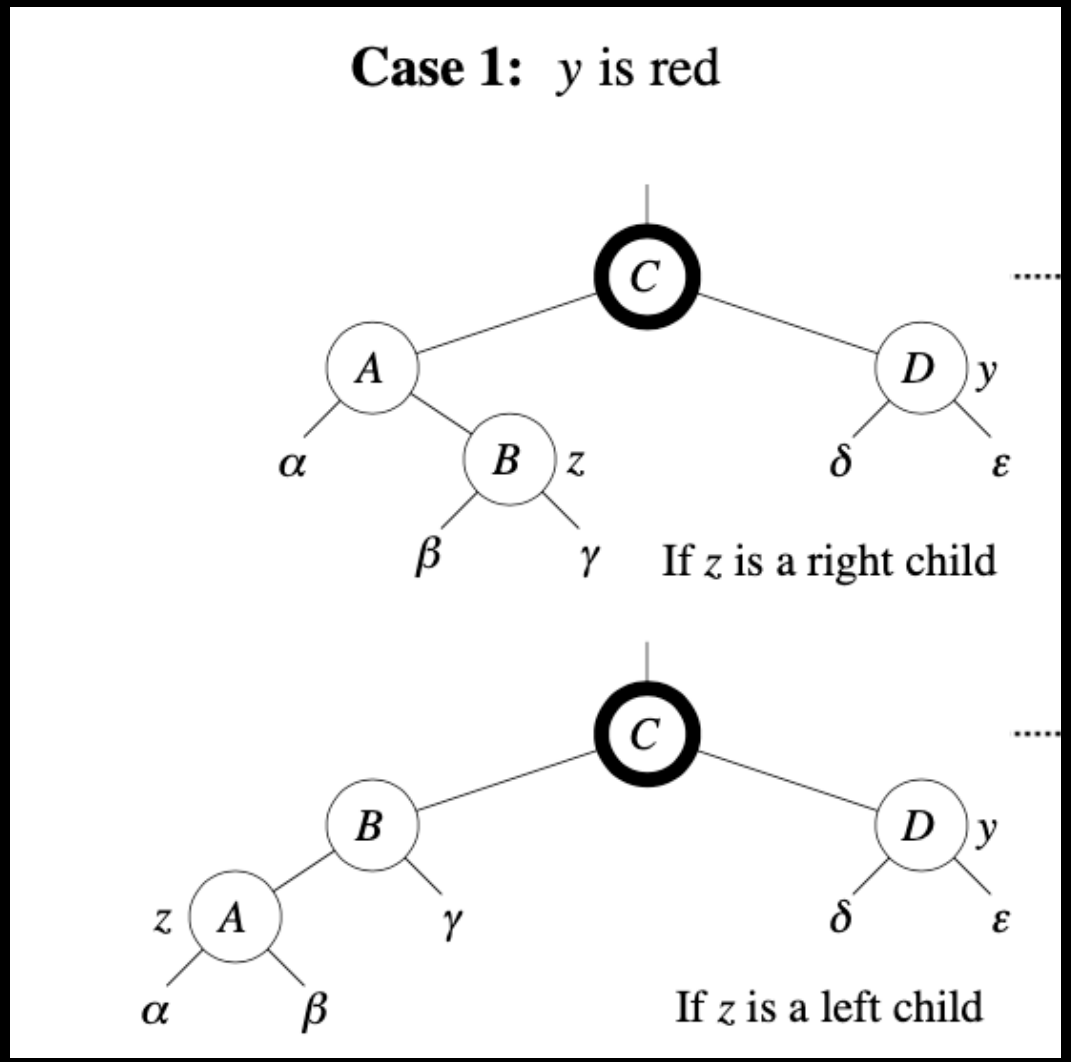
7.3.2. Insertions

- 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.

⇒ Remove the violation by calling RB-INSERT-FIXUP:

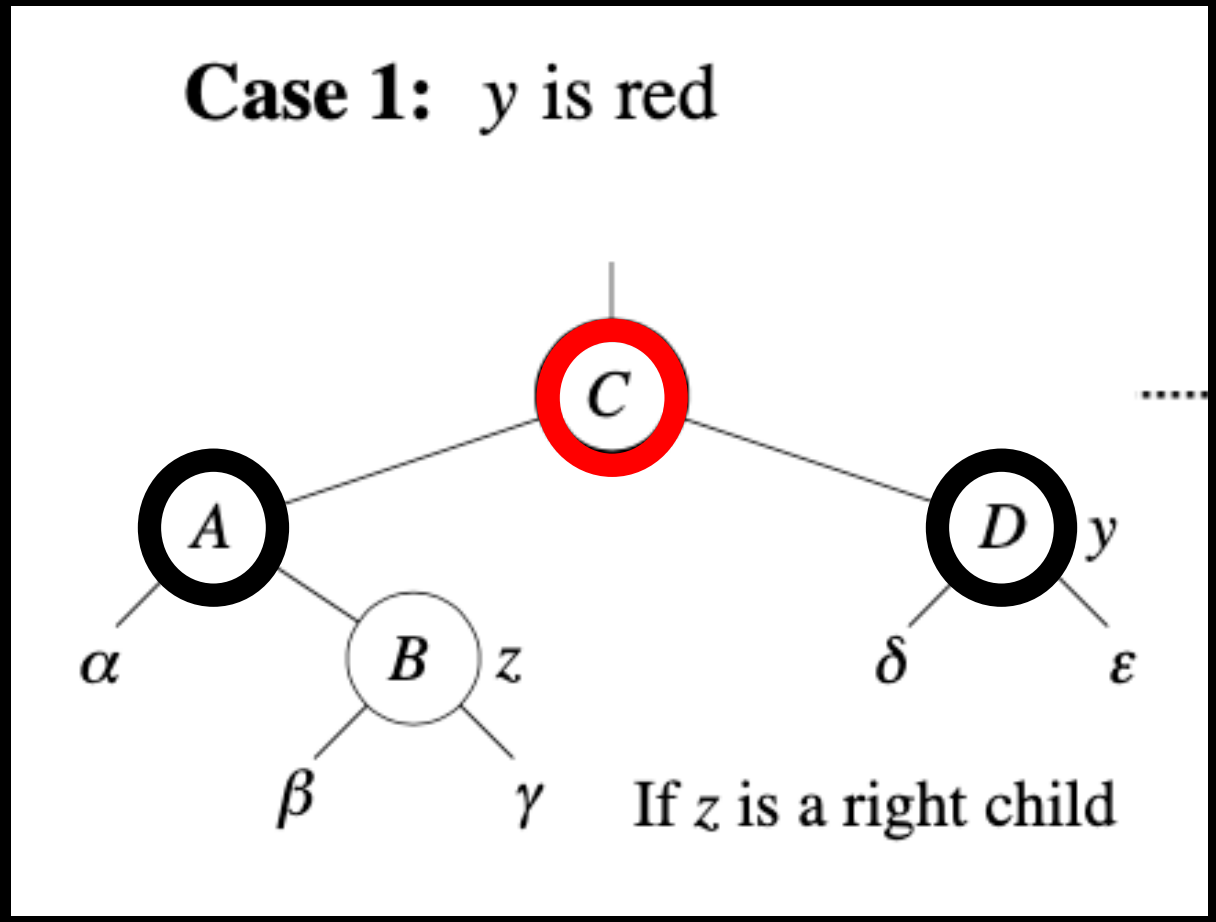
7.3.2. Insertions

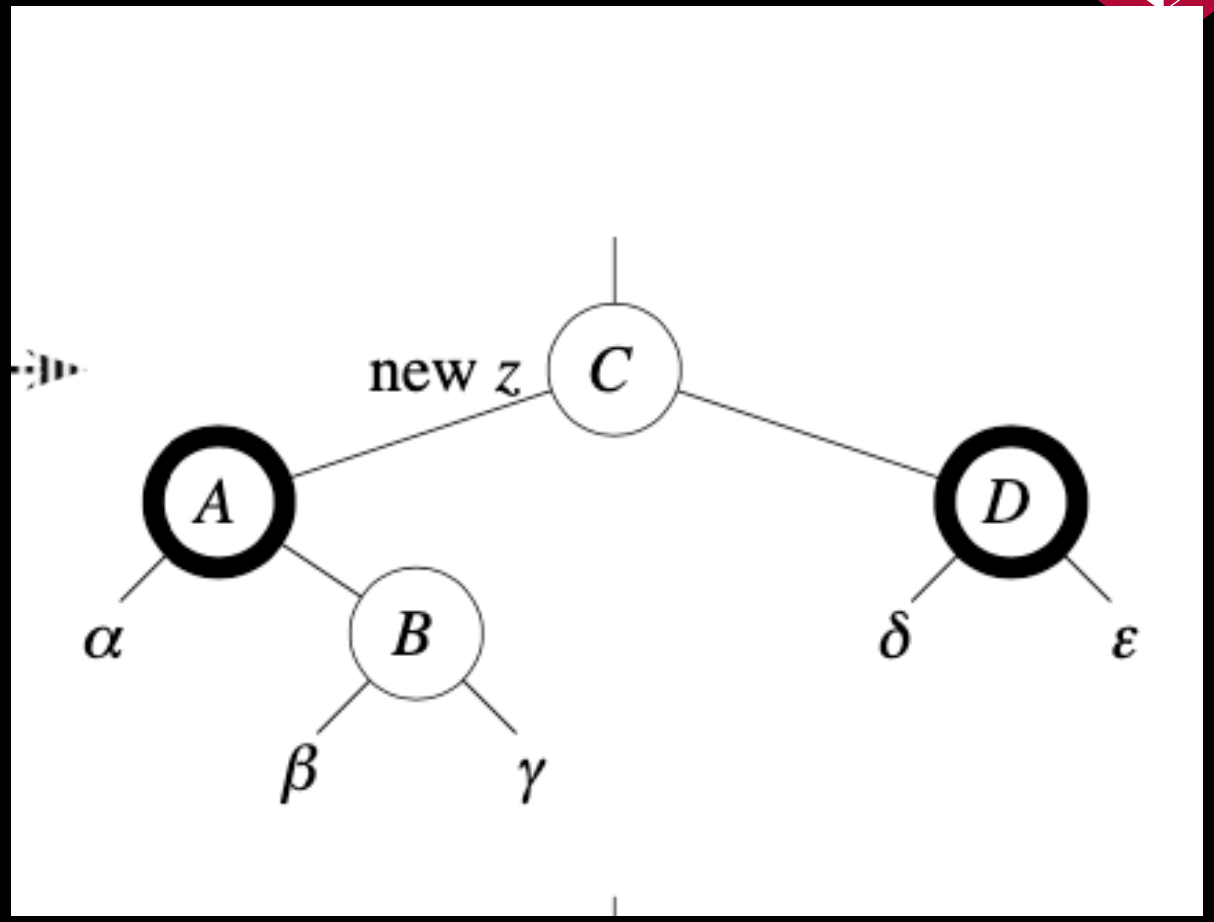
when both z.p and y are red



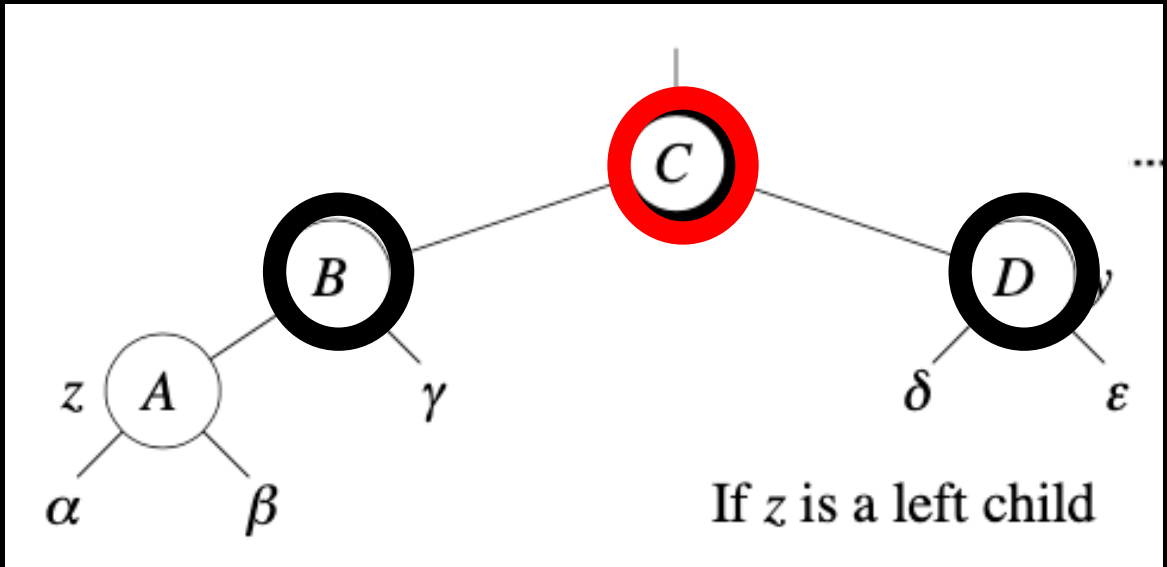
- ***z.p.p* (*z*'s grandparent) must be black**, since *z* and *z.p* are both red and there are no other violations of property 4.
- **Make *z.p* and *y* black** ⇒ now *z* and *z.p* are not both red.
- But property 5 might now be violated.
- **Make *z.p.p* red** ⇒ restores property 5.
- The next iteration has ***z.p.p* as the new *z*** (i.e., *z* moves up 2 levels).

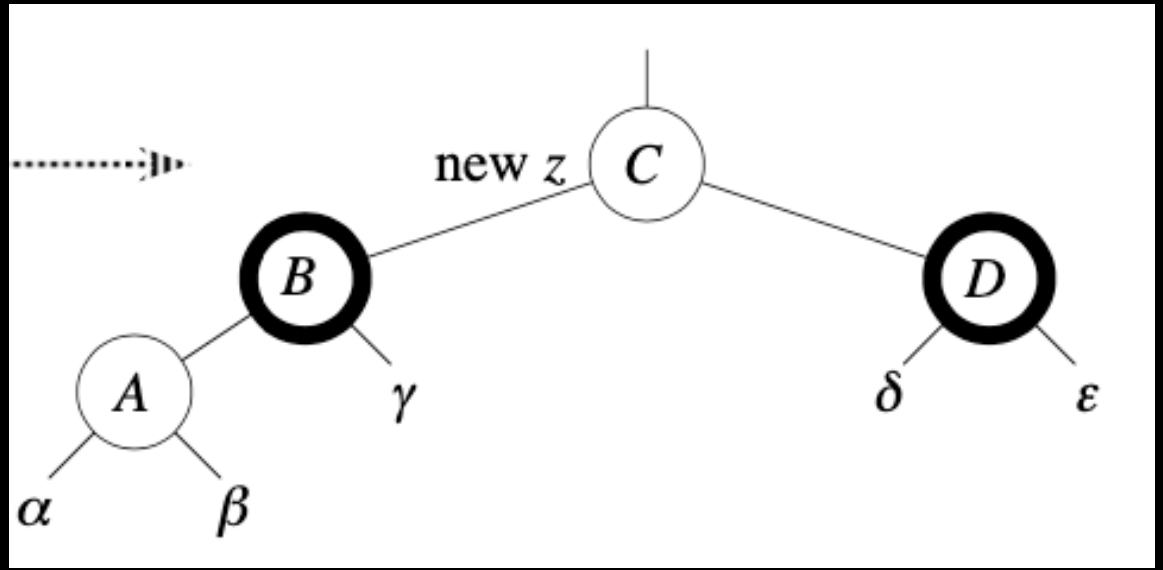
7.3.2. Insertions





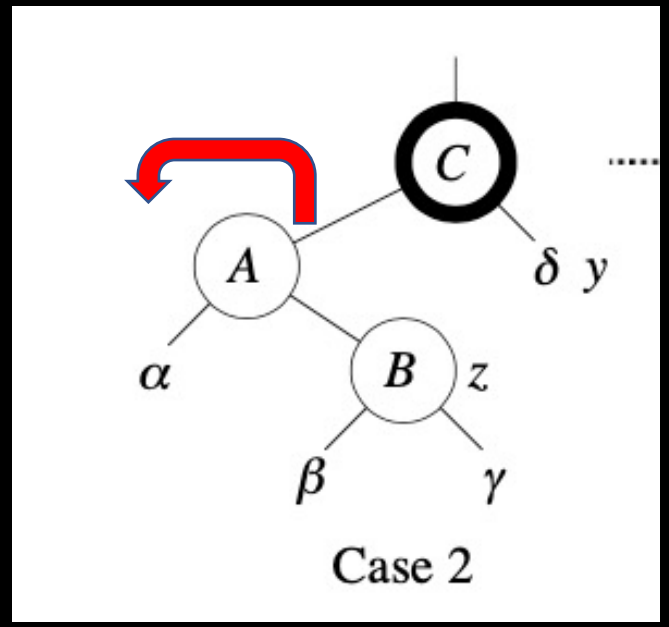
7.3.2. Insertions





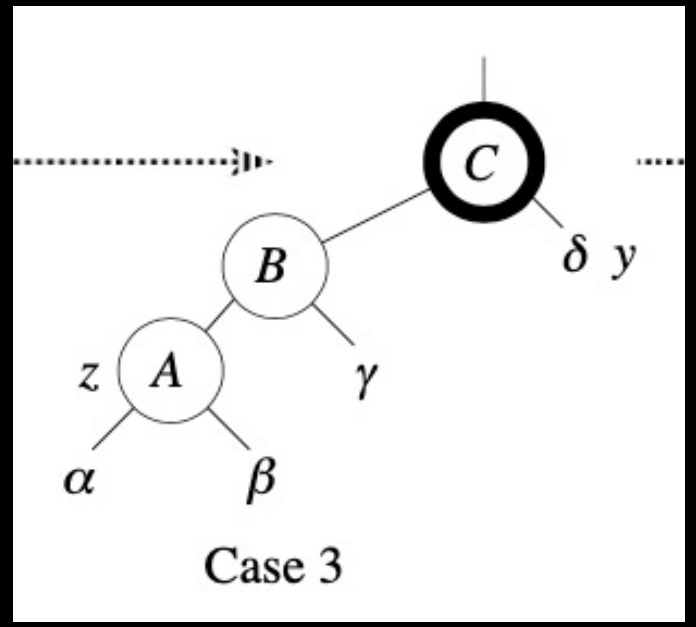
7.3.2. Insertions

Case 2: y is black, z is a right child



Case 2

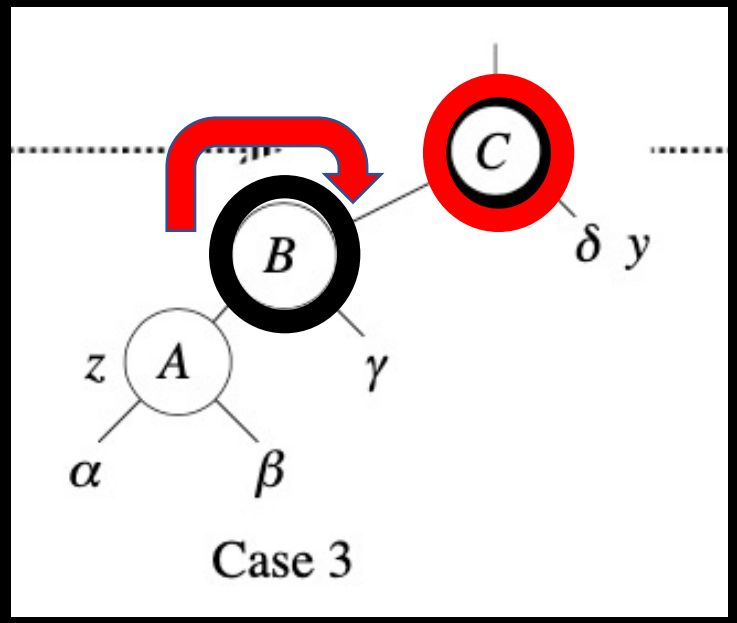
- **Left rotate around $z.p$** ⇒ now z is a left child, and both z and $z.p$ are red.
- Takes us immediately to case 3.



Case 3

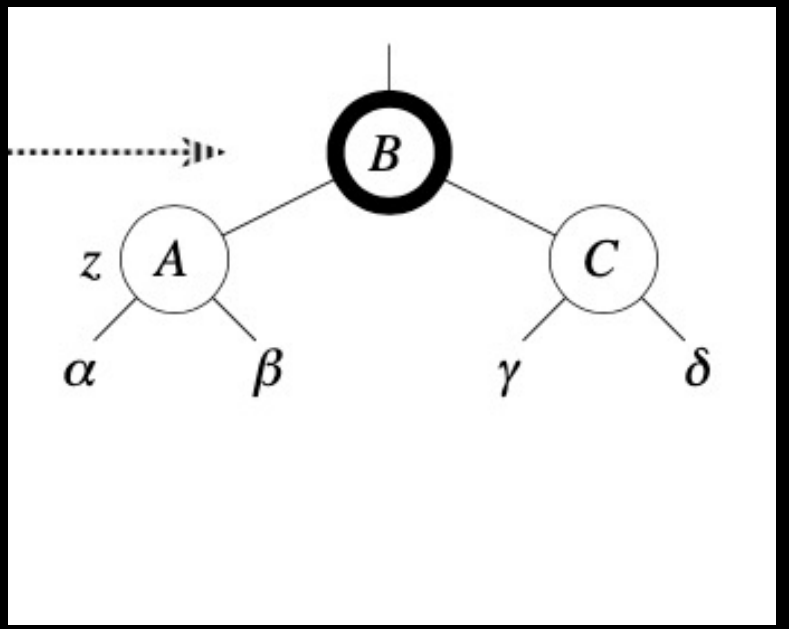
7.3.2. Insertions

Case 3: y is black, z is a left child



Case 3

- **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** ⇒ no more iterations.



7.3.2. Insertions

```
Algorithm (RB-INSERT-FIXUP(T,z))
(1)     while (z.p.color = RED) do
(2)       if (z.p = z.p.p.left) then
(3)         y = z.p.p.left
(4)         if (y.color = RED) then
(5)           z.p.color = BLACK, y.color = BLACK    //case 1
(6)           z.p.p.color = RED
(7)           z = z.p.p
(8)         else
(9)           if (z = z.p.right) then
(10)            z = z.p                             //case 2
(11)            LEFT-ROTATE(T,z)
(12)          z.p.color = BLACK, z.p.p.color = RED  //case 3
(13)          RIGHT-ROTATE(T, z.p.p)
(14)      else
(15)        (do same with right and left exchange)
(16)    T.root.color = BLACK
```

7.3.2. Insertions

Algorithm (RB-**INSERT-FIXUP(T,z)**)

```
(1)     while (z.p.color = RED) do
(2)       if (z.p = z.p.p.left) then
(3)          y = z.p.p.left
(4)          if (y.color = RED) then
(5)             z.p.color = BLACK, y.color = BLACK
(6)             z.p.p.color = RED
(7)             z = z.p.p
(8)          else
(9)            if (z = z.p.right) then
(10)              z = z.p                                  //case 2
(11)              LEFT-ROTATE(T,z)
(12)           z.p.color = BLACK, z.p.p.color = RED       //case 3
(13)           RIGHT-ROTATE(T, z.p.p)
(14)      else
(15)         (do same with right and left exchange)
(16)   T.root.color = BLACK
```

Loop invariant:

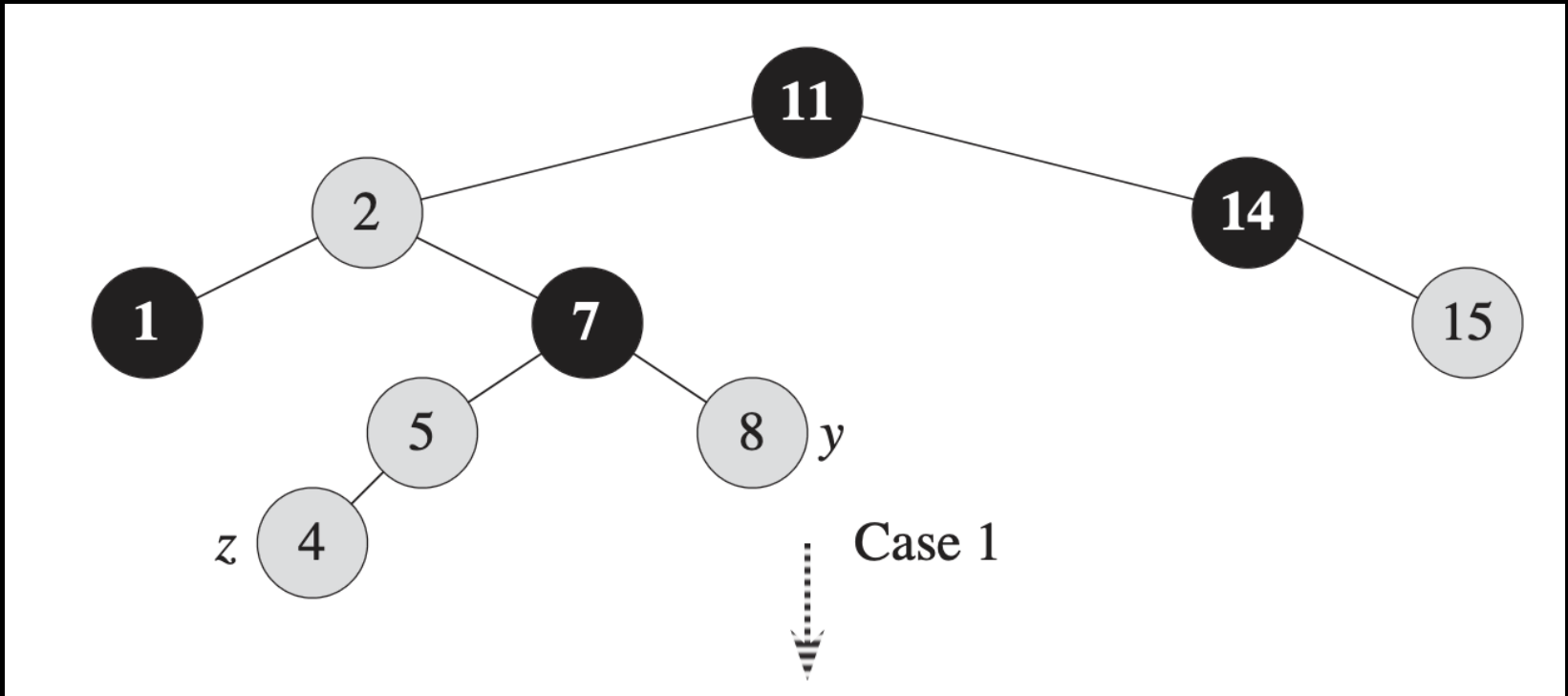
At the start of each iteration of the **while** loop,

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

7.3.2. Insertions

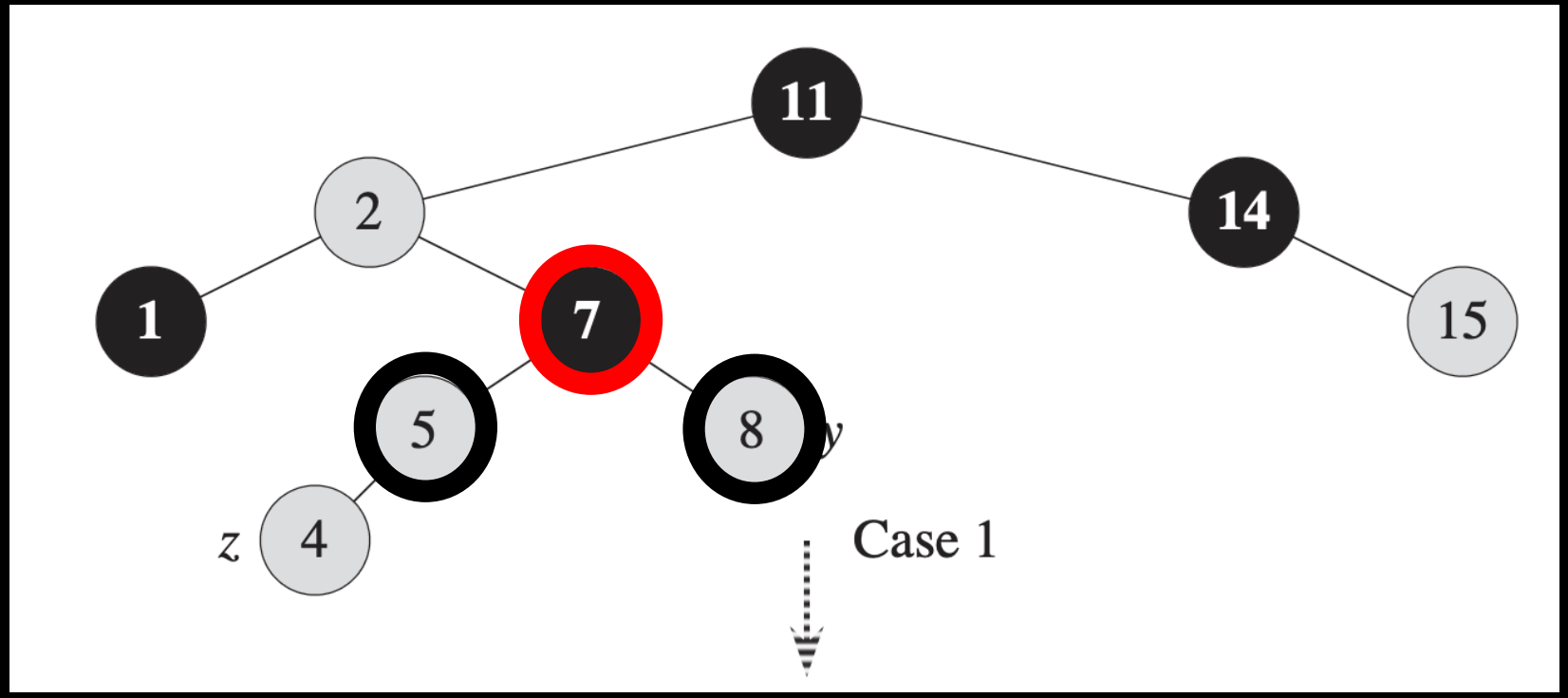
- **Initialization:** We've already seen why the loop invariant holds initially.
- **Termination:** The loop terminates because $z.p$ is black. Hence, property 4 is OK. Only property 2 might be violated, and the last line fixes it.
- **Maintenance:** We drop out when z is the root (since then $z.p$ is 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).

7.3.2. Insertions

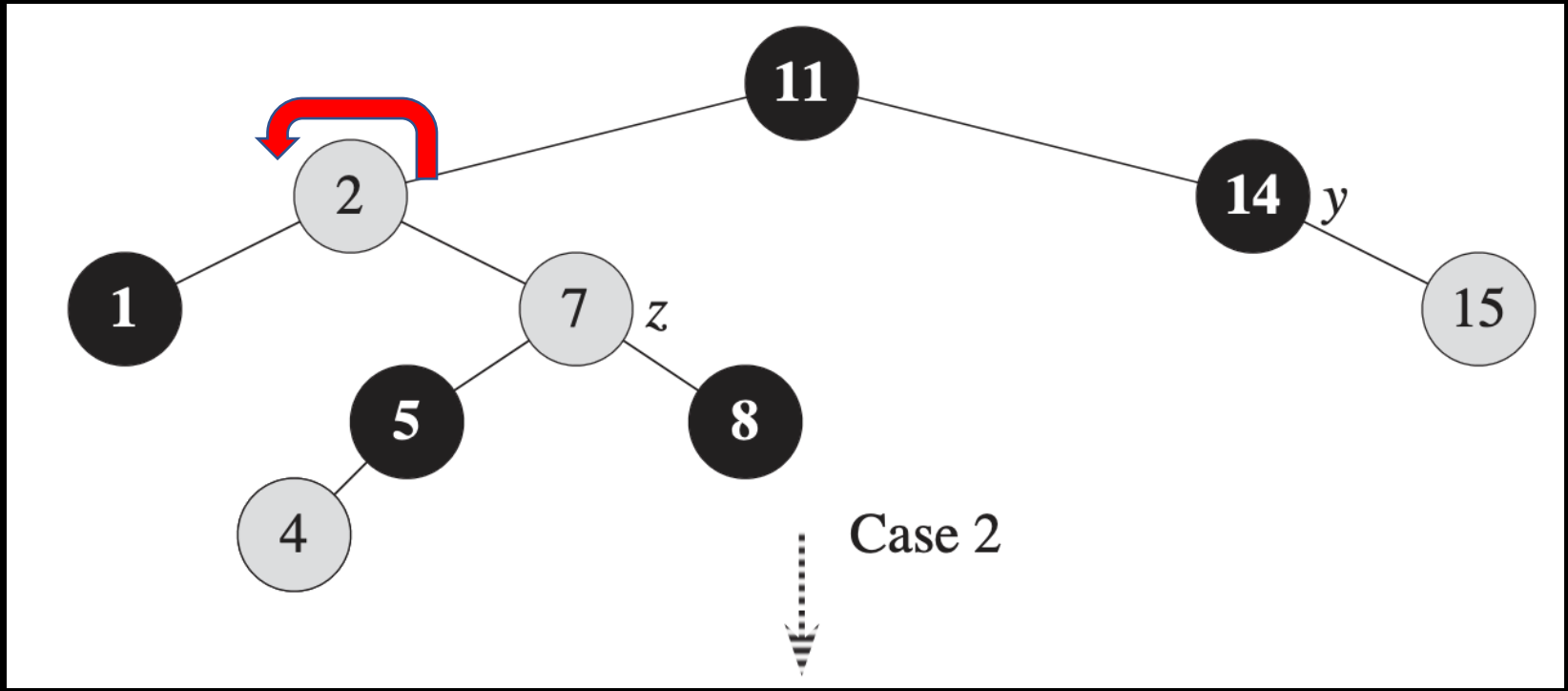


- Which of the red-black properties might be violated upon the call to RB-INSERT-FIXUP?
- Property 1, 3, 5 are satisfied.
- Property 2 and property 4 possible violations are due to z being colored red.
- Property 2 is violated if z is the root, and property 4 is violated if z's parent is red.

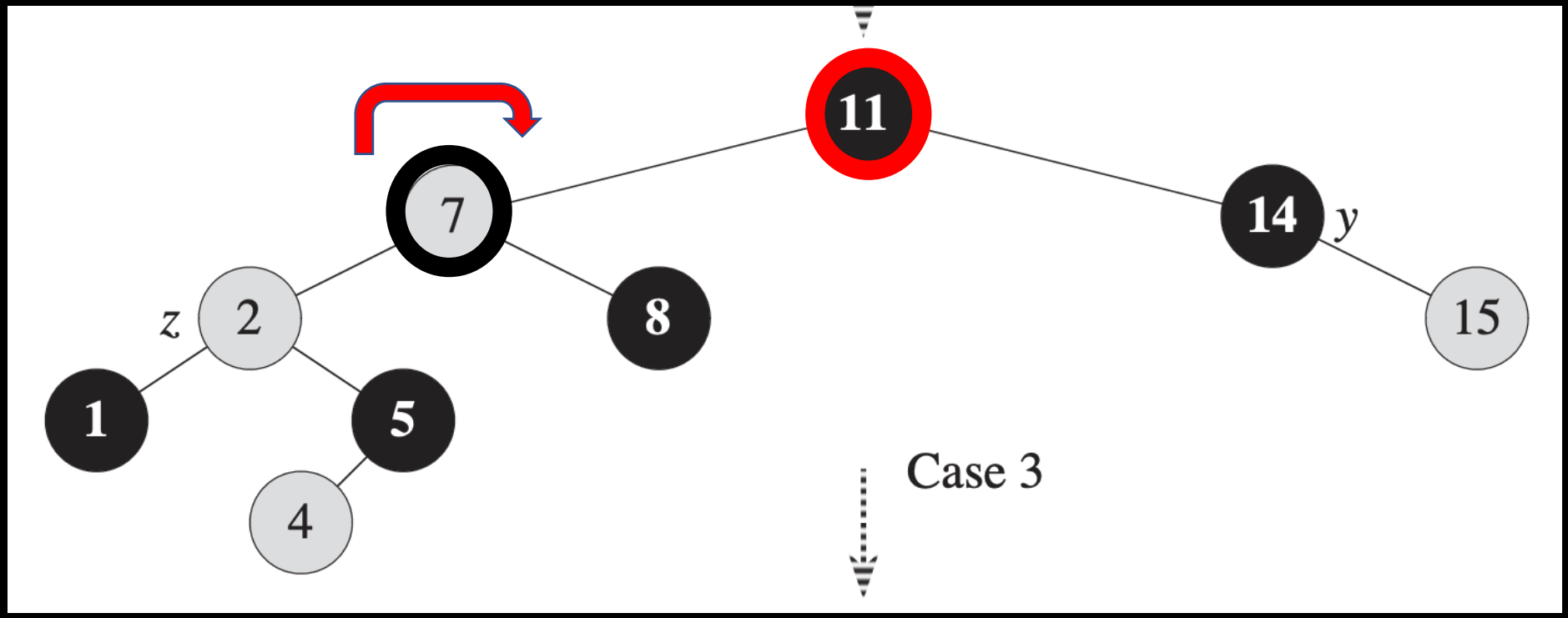
7.3.2. Insertions



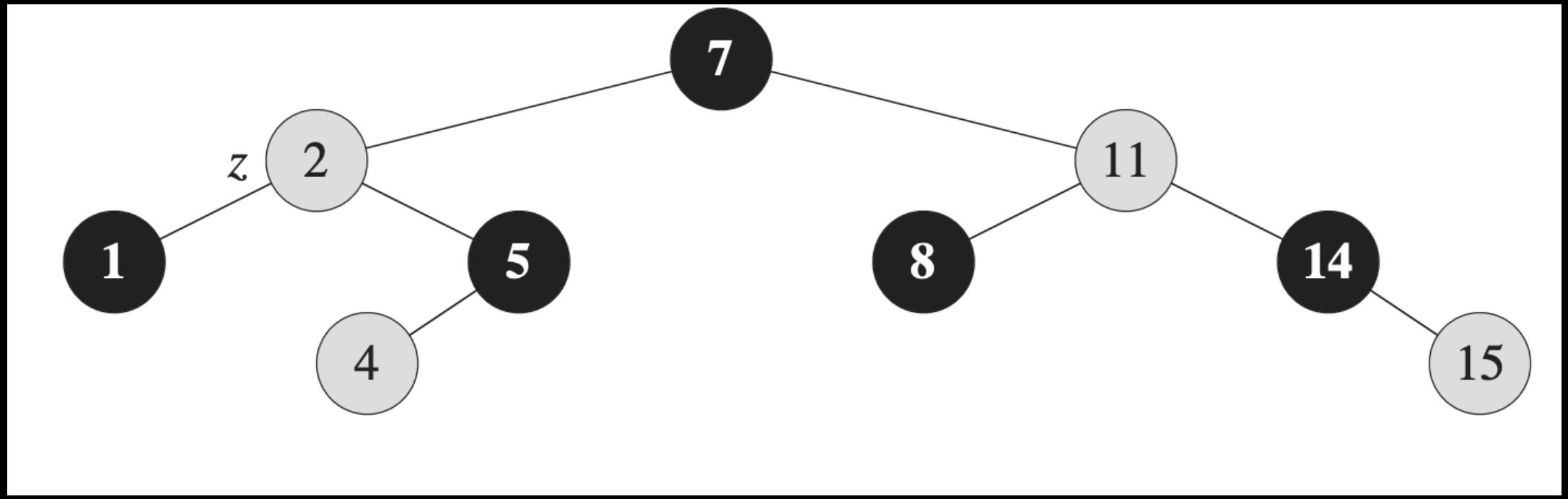
7.3.2. Insertions



7.3.2. Insertions



7.3.2. Insertions



7.3.2. Insertions

Analysis

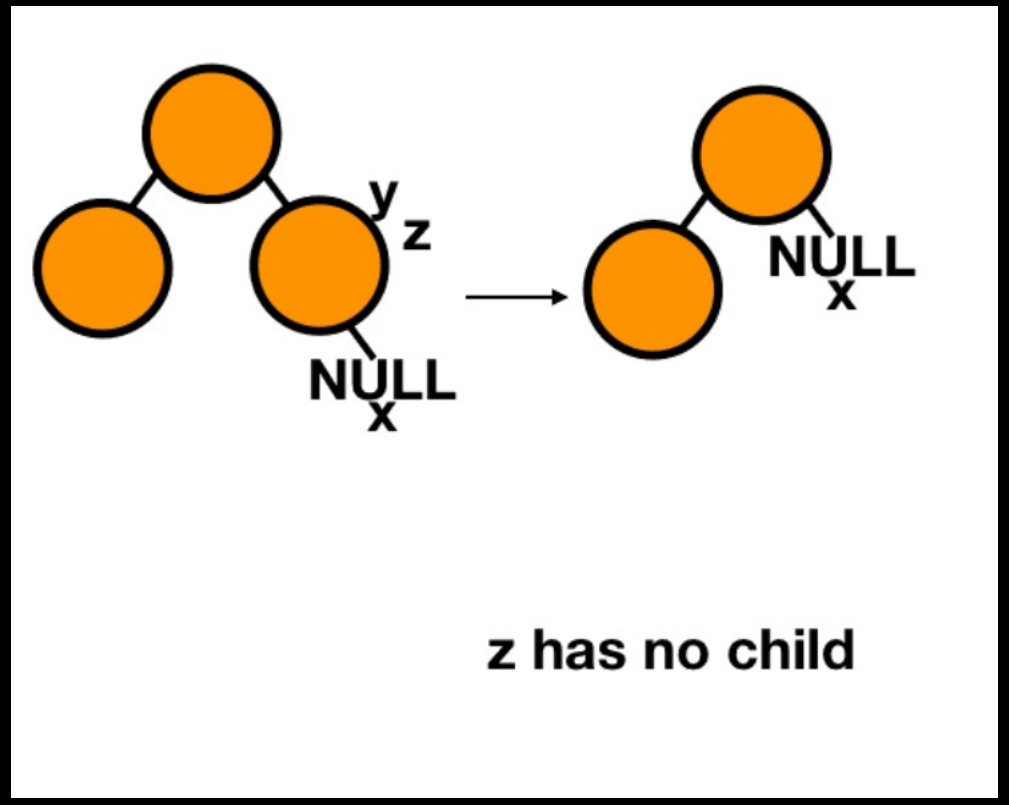
$O(\lg n)$ time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.

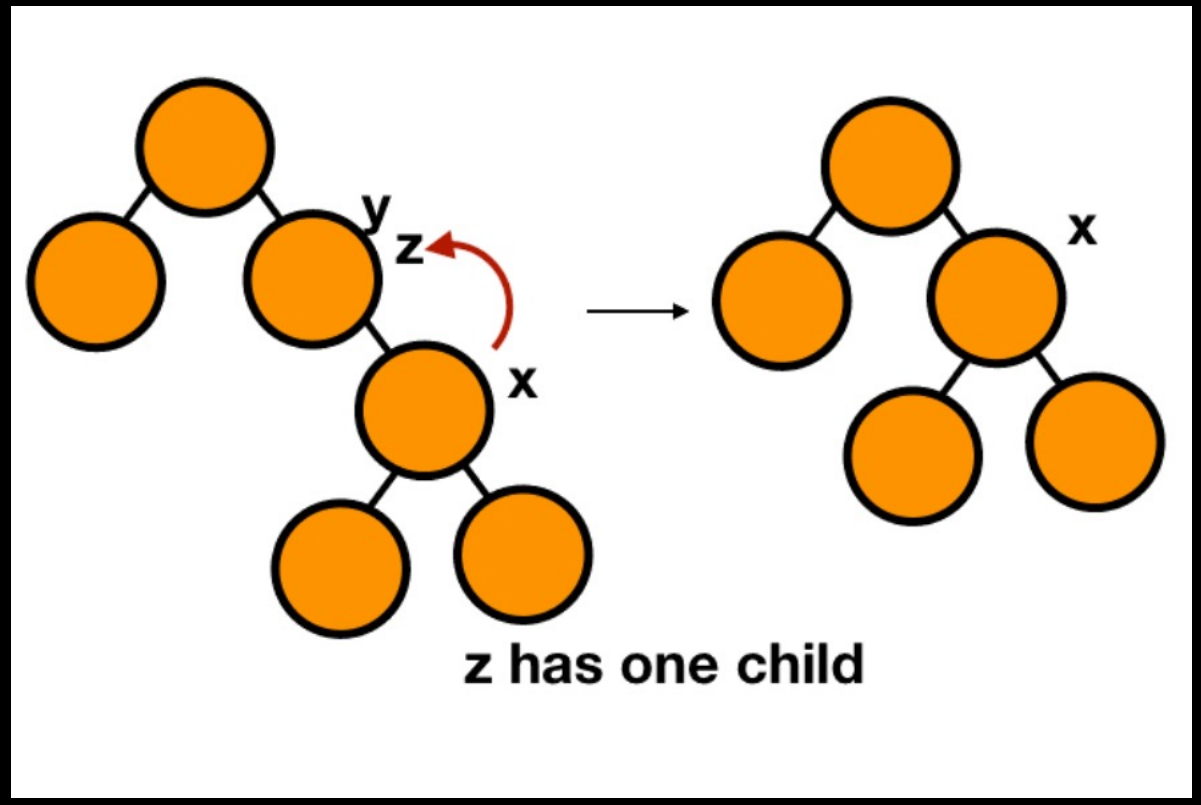
Within RB-INSERT-FIXUP:

- 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.

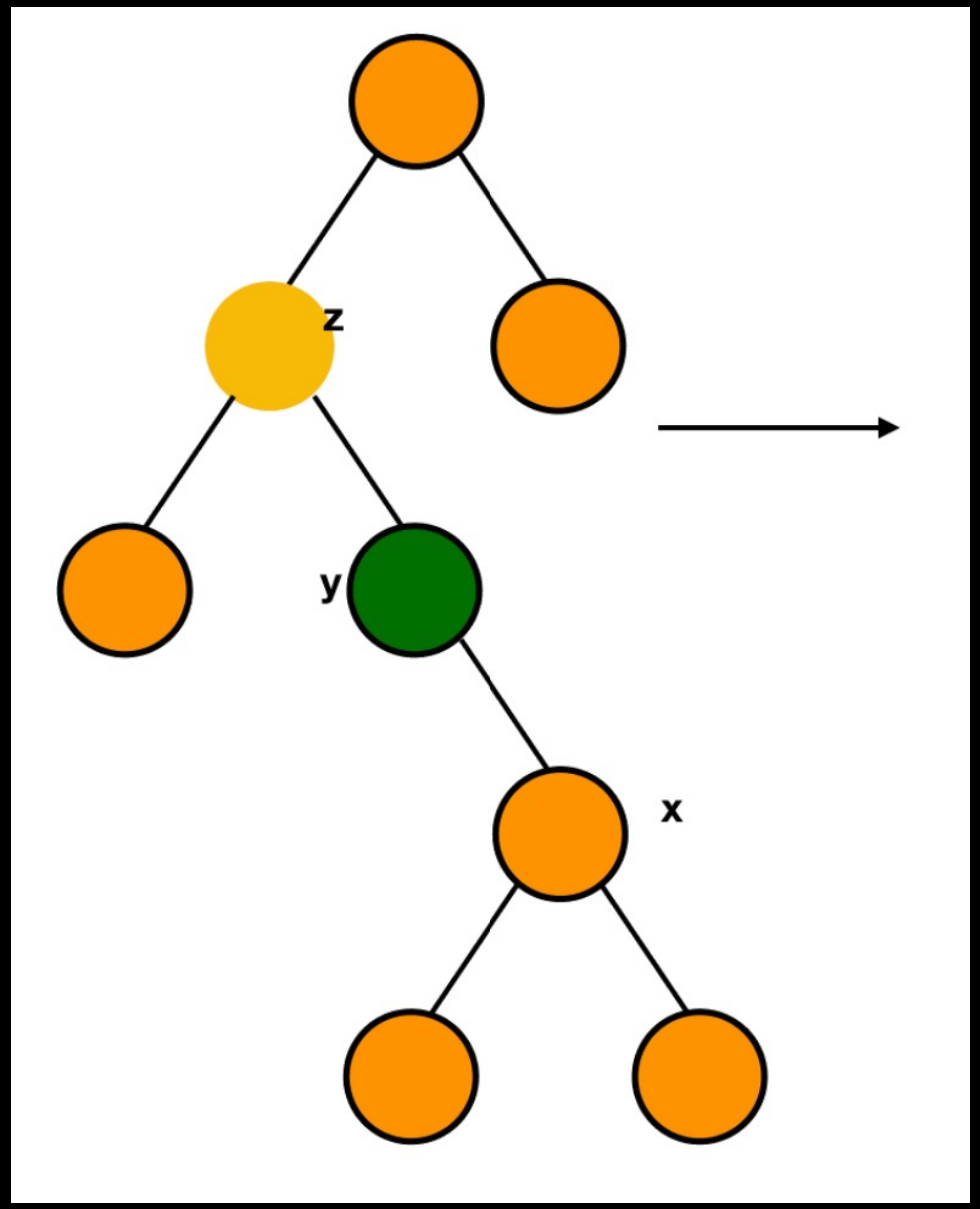
Thus, insertion into a red-black tree takes $O(\lg n)$ time.

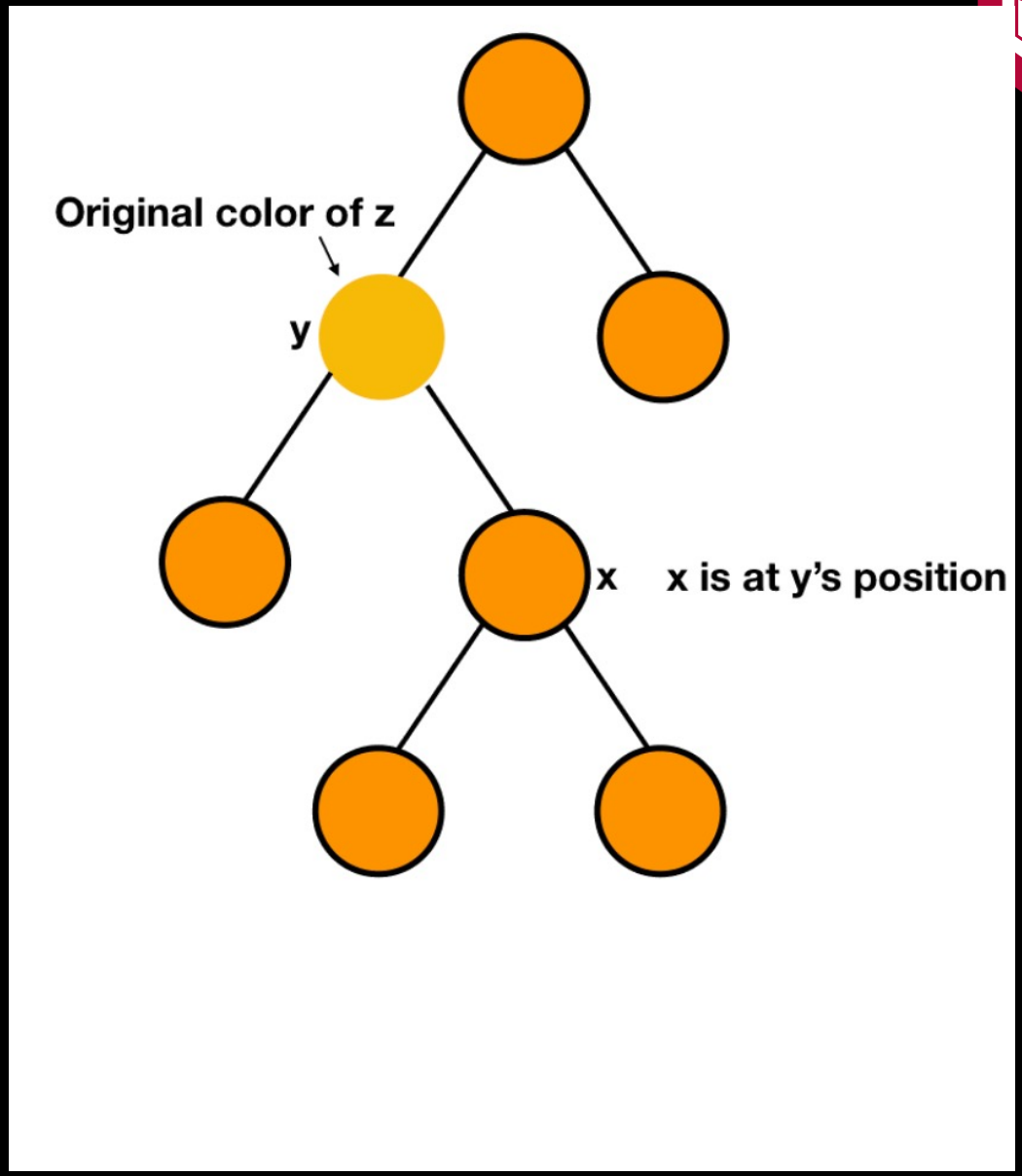
Recap – BST Deletion



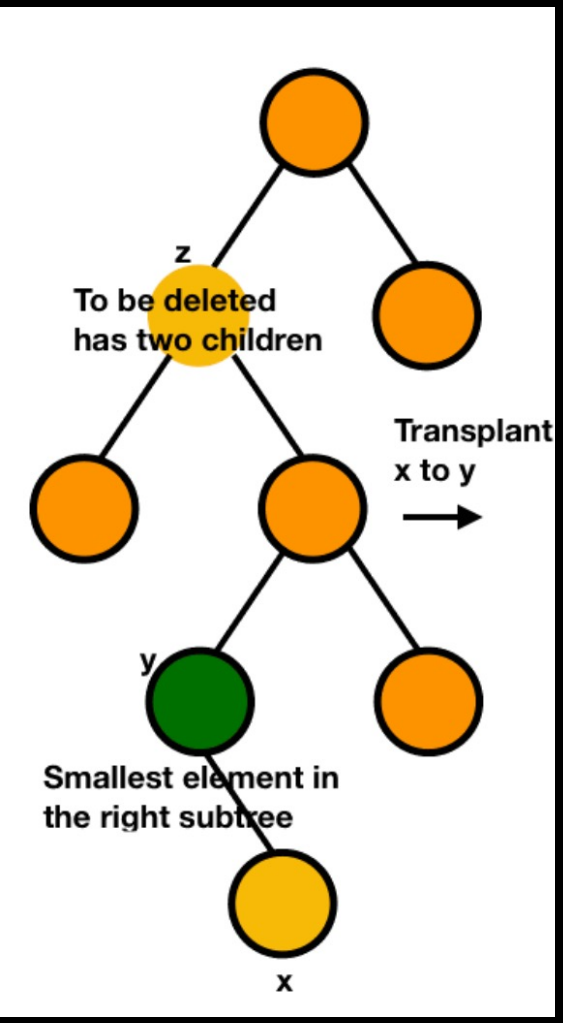


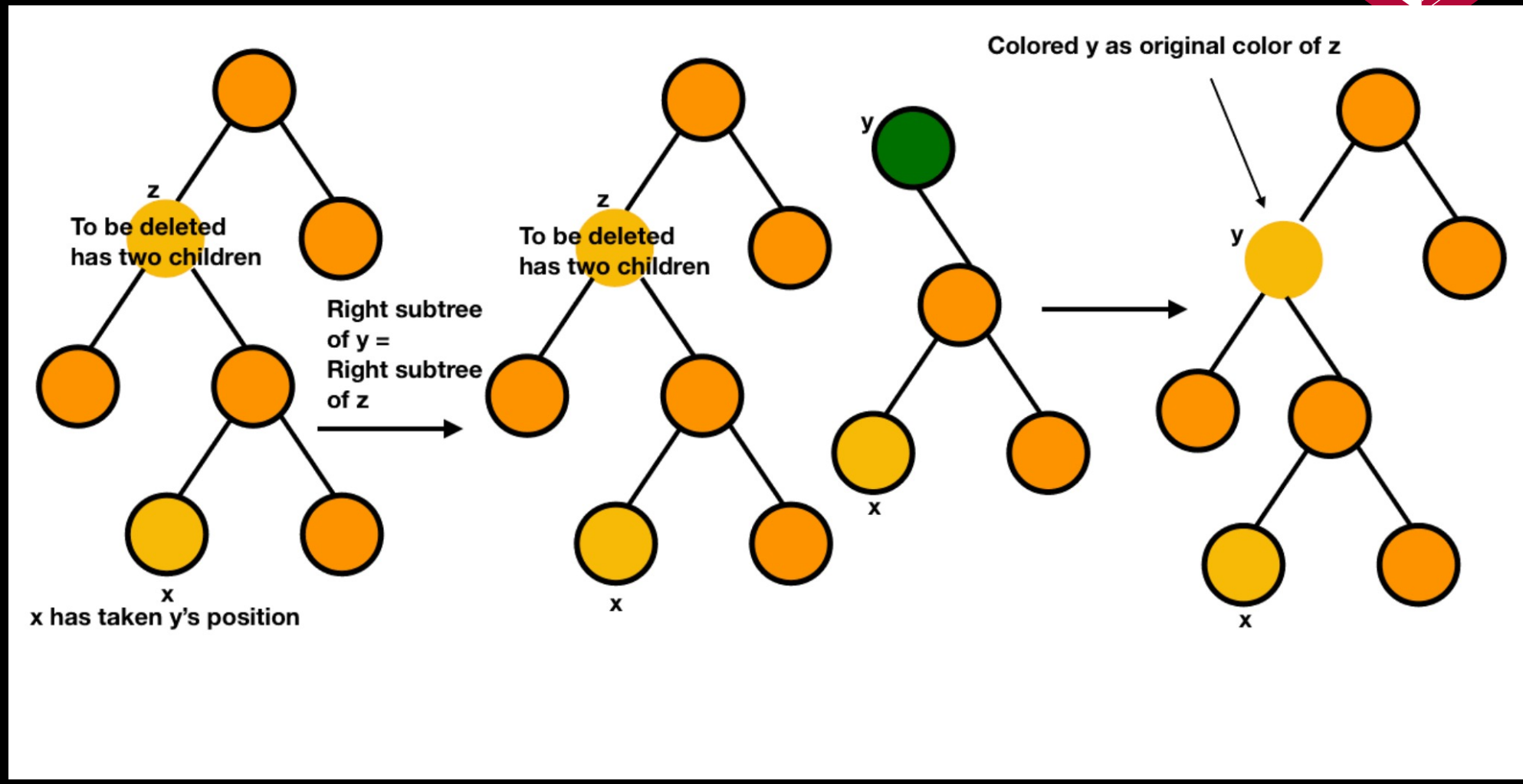
Recap – BST Deletion



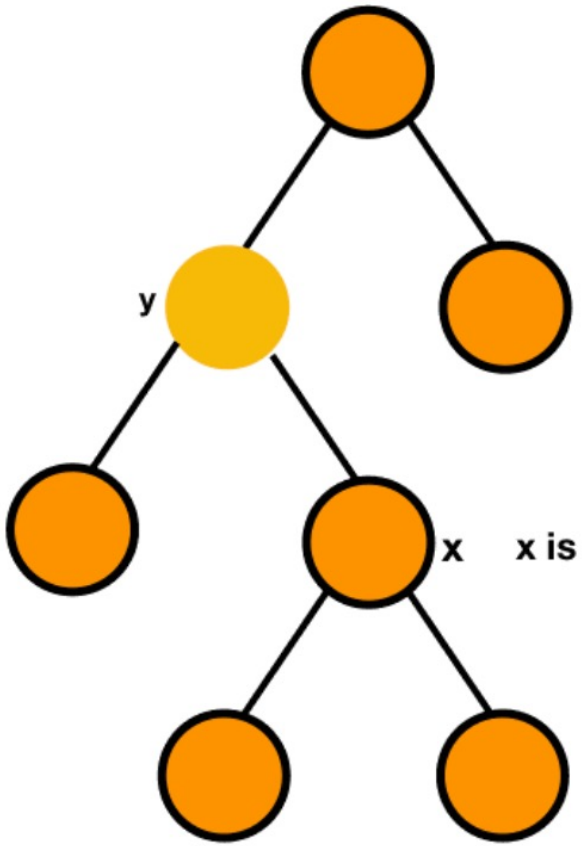


Recap – BST Deletion

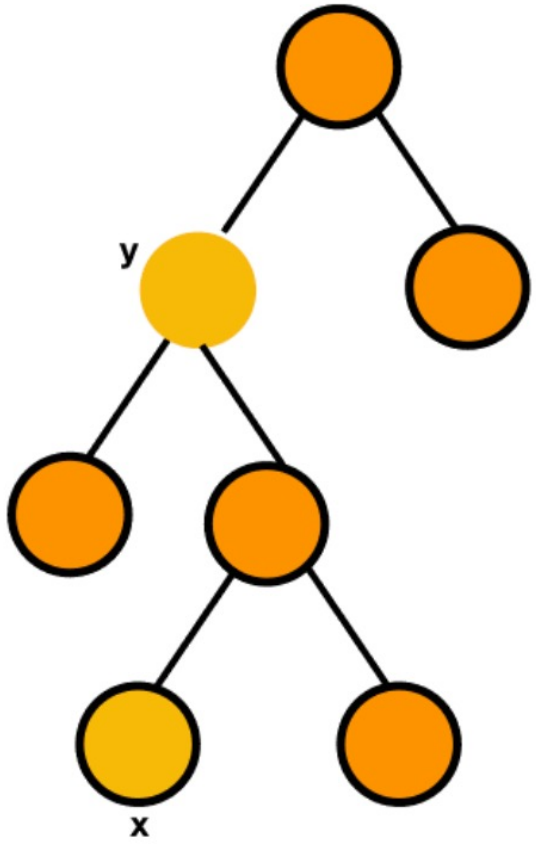


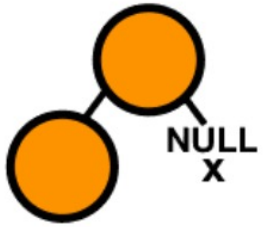


7.3.3. Deletions



x is at y's position





x has taken y's original position.
Red-Black property might have violated.

7.3.3. Deletions

```
(1)  RB-TRANSPLANT(T, u, v)
(2)  if u.p = T.nil
(3)      T.root ← v
(4)  elseif u = u.p.left
(5)      u.p.left ← v
(6)  else u.p.right ← v
(7)  v.p ← u.p
```

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right                          // z has no left child
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left                           // just left child
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)           //two children, y is successor of z
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)    //y in subtree of z subtree
        y.right = z.right                //and not root of subtree
        y.right.p = y
    RB-TRANSPLANT(T, z, y)               // replace z by y
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

- Deleting a node from a red-black tree is a bit more complicated than inserting a node.
- The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure.

7.3.3. Deletions

Difference between RB-DELETE and TREE-DELETE:

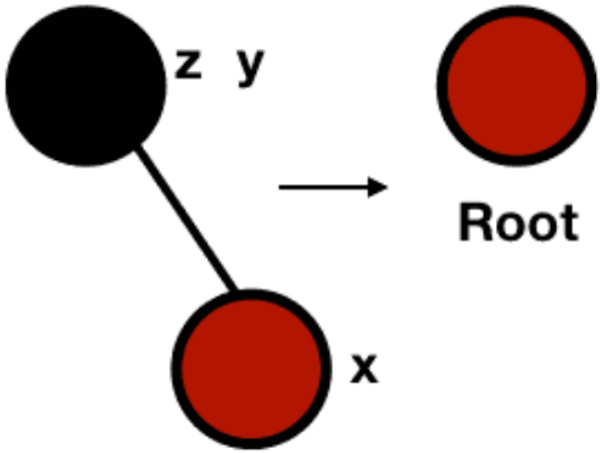
- We maintain node y as the node either removed from the tree (when z has fewer than 2 children) or moved within the tree (when z has 2 children).
- Need to save original color of y. It is tested at the end, because if it's black, then removing or moving y could cause red-black properties to be violated.
- x is the node that moves into original position of y. It is either only child of y, or T.nil if y has no child.
- Set x.p to point to the original position of y's parent, even if x=T.nil. x.p is set into one of two ways:
  - If z not original parent of y, then x.p is set in last line of RB-TRANSPLANT.
  - Otherwise, y will move up to take z's position in the tree. Assignment x.p=y makes x.p point to original poision of y's parent , even if x is T.nil.
- If original color of y was black, the changes to the tree structure might cause a violation of the red-black properties. Call RB-DELETE-FIX to resolve violations.

7.3.3. Deletions

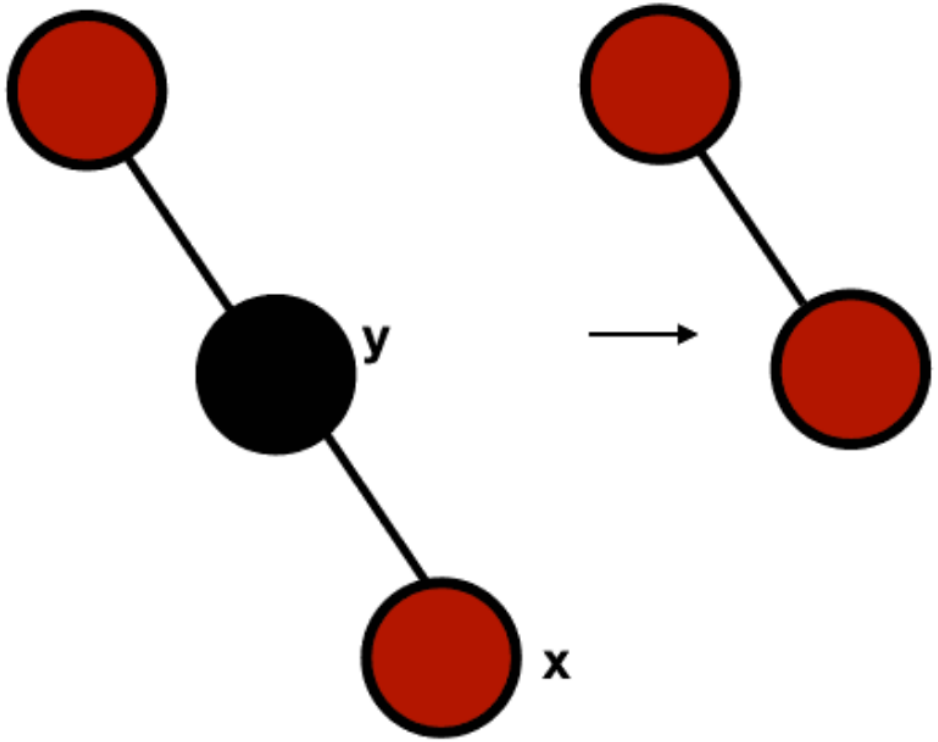
Possible violations of red-black properties:

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 in paths containing x.
   - x is either doubly black (if x.color=BLACK) or red & black (if x.color=RED).
   - Property 5 satisfied, but property 1 violated.
   - The attribute x.color is still either RED or BLACK. No new values for the color attribute.
   - Extra blackness on a node is by virtue of x pointing to the node.

7.3.3. Deletions



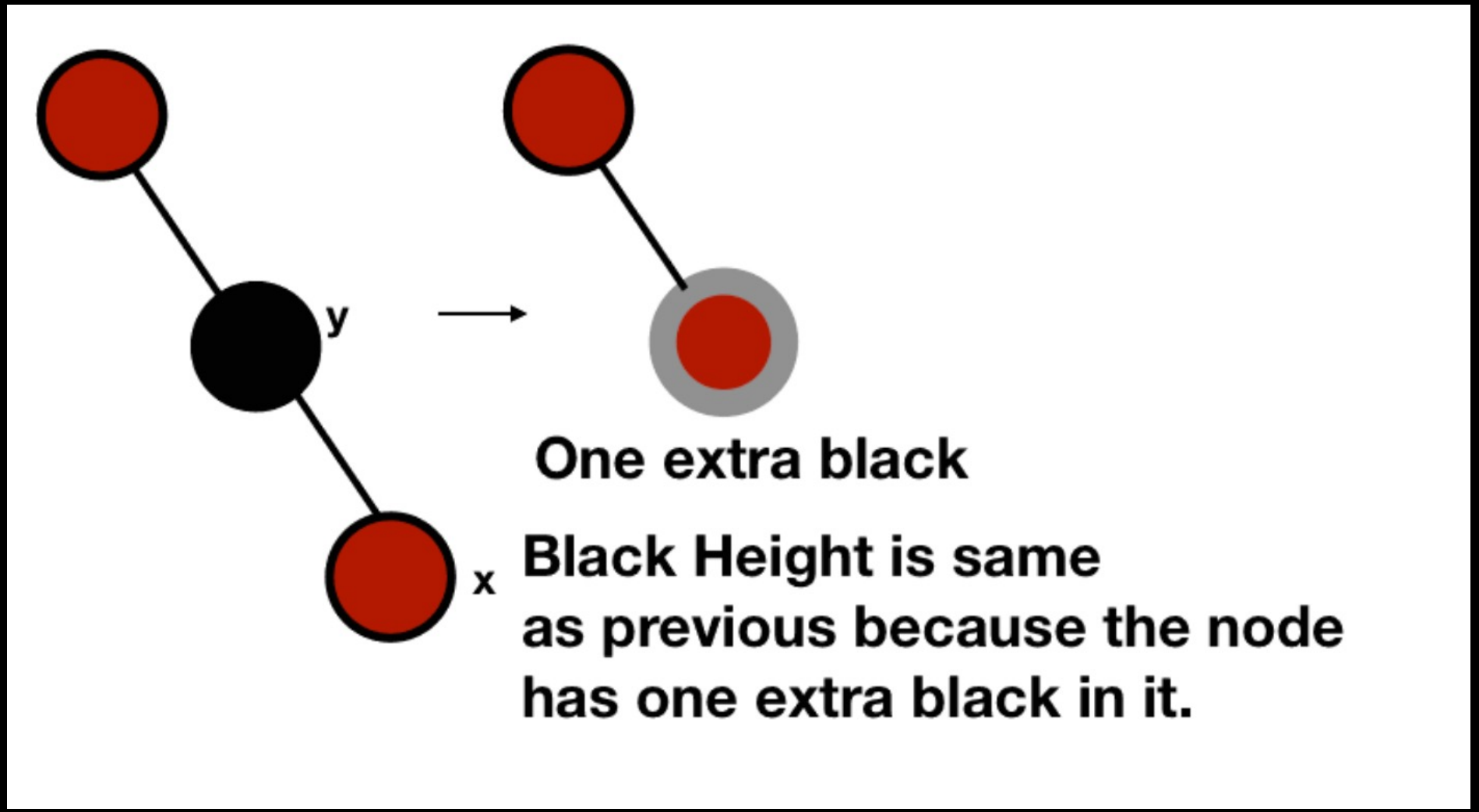
Violation of prop. 2



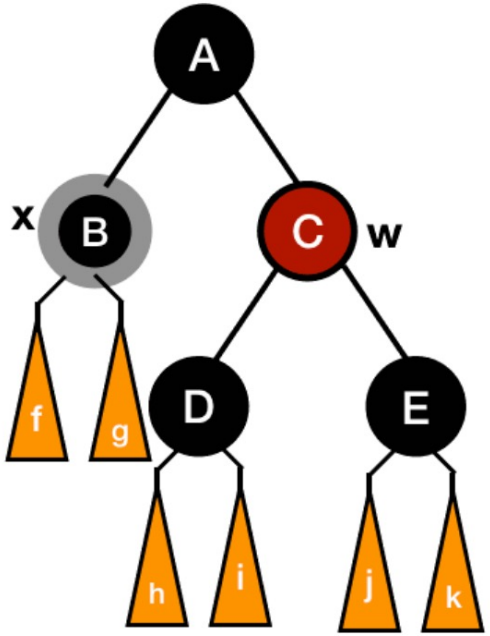
Violation of prop. 4

Violation of prop. 5,
black height affected

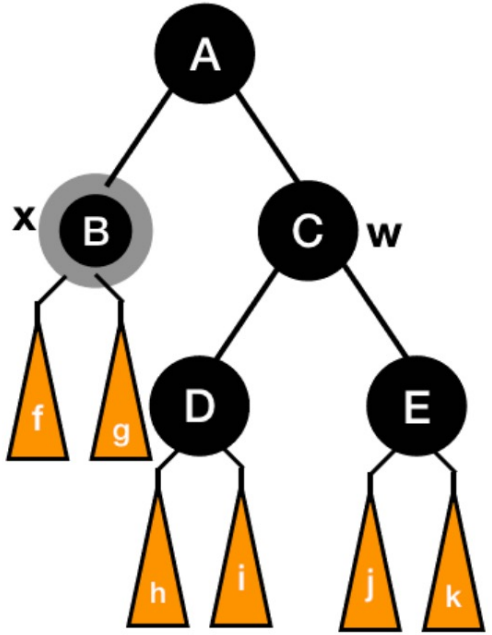
7.3.3. Deletions



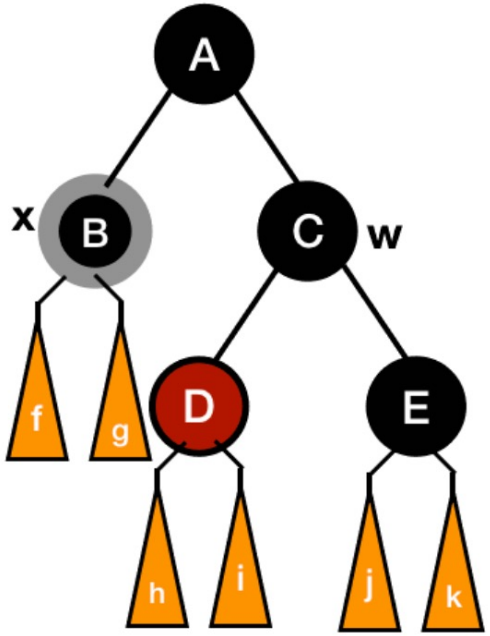
7.3.3. Deletions



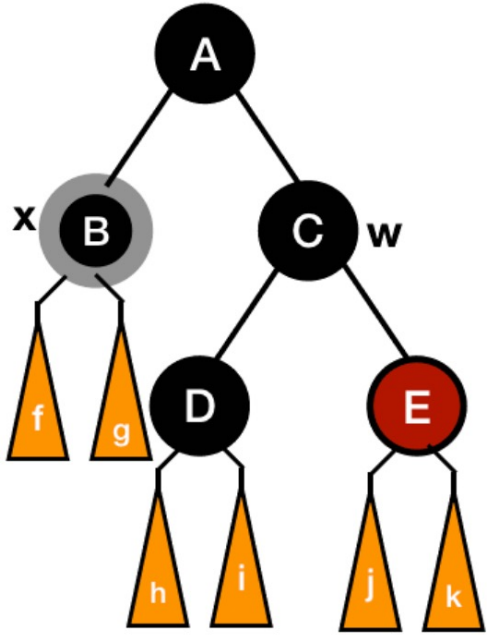
Case 1



Case 2

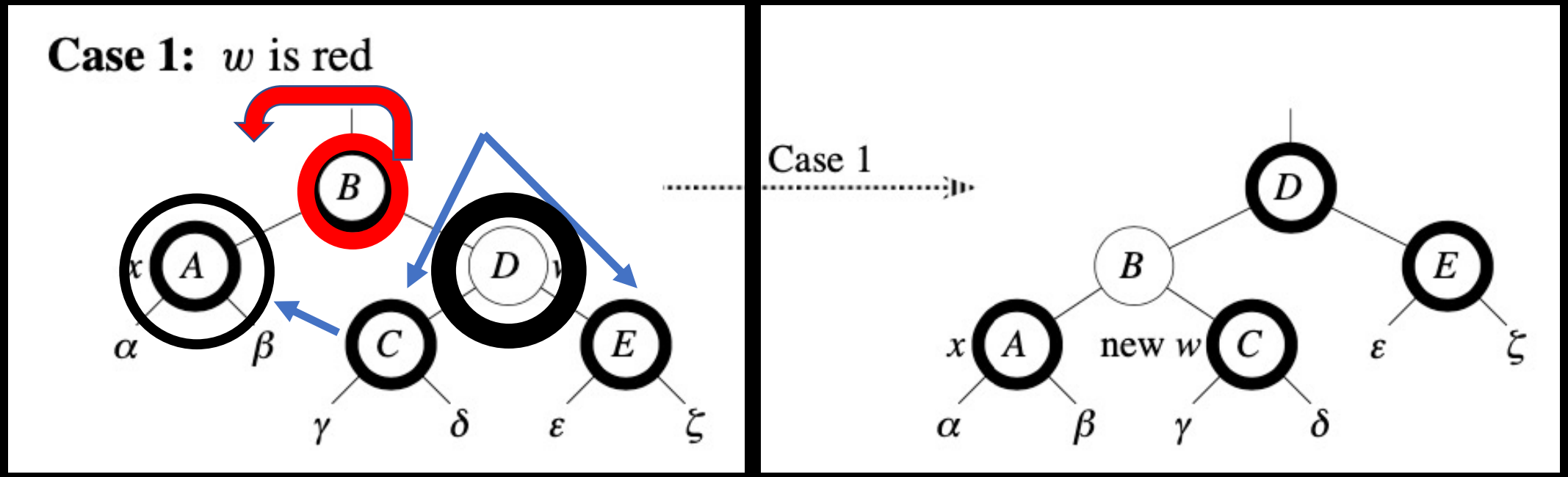


Case 3



Case 4

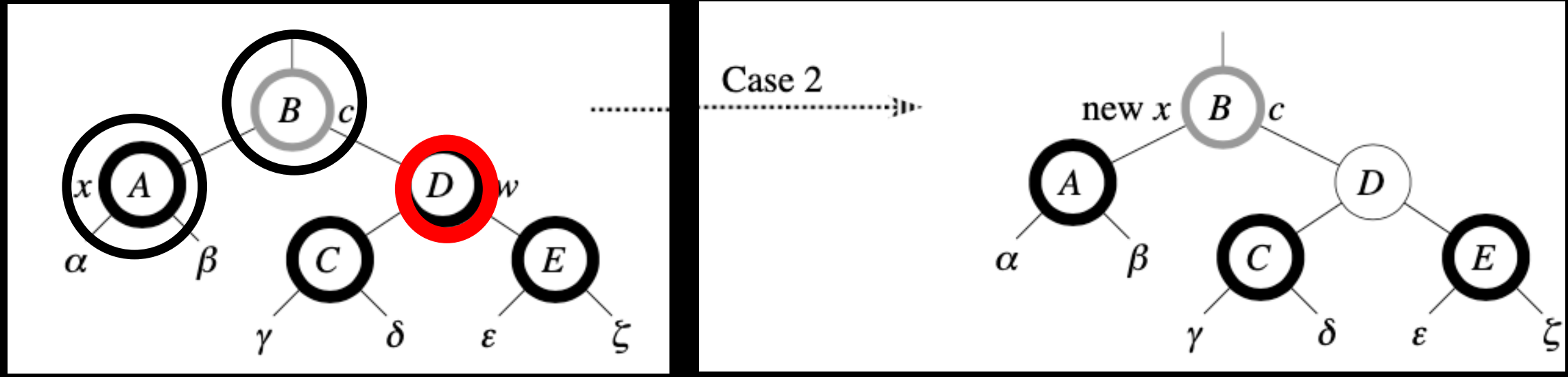
7.3.3. Deletions



- **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** ⇒ must be black.
- Go immediately to case 2, 3, or 4.

7.3.3. Deletions

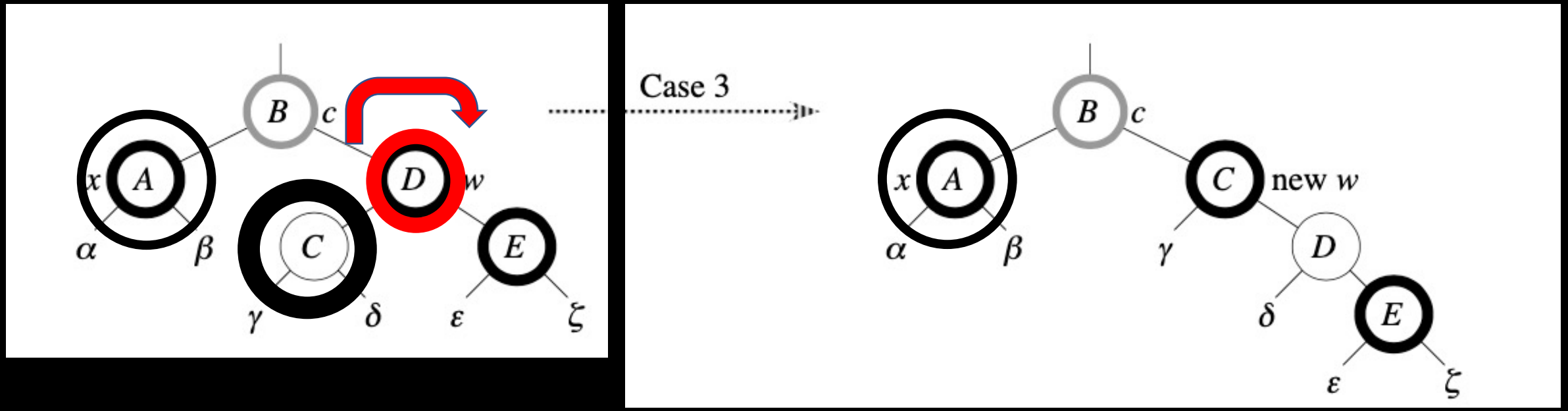
Case 2: w is black and both of w's children are black



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

7.3.3. Deletions

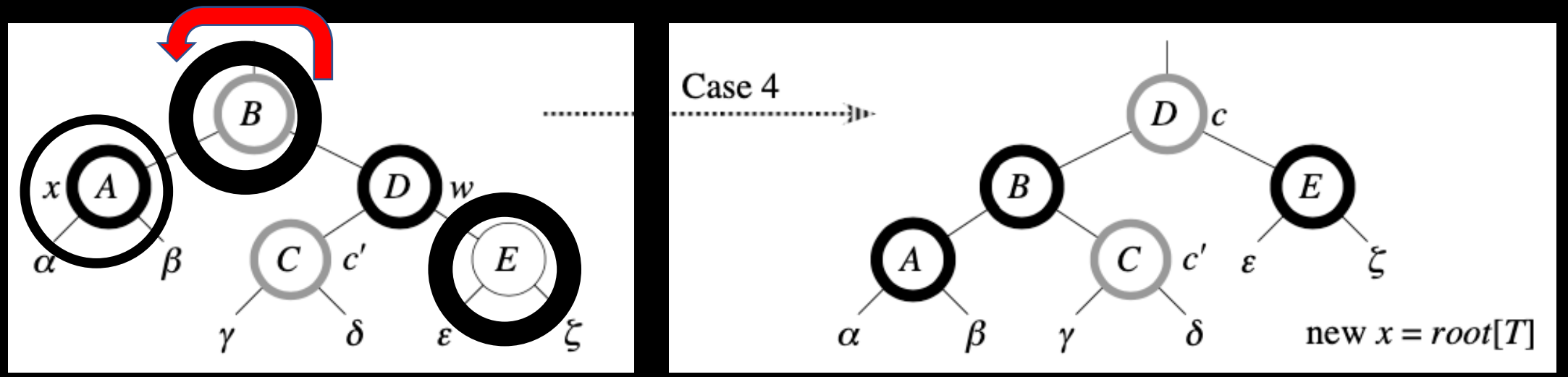
Case 3: w is black, w's left child is red, and w's right child is black



- **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.

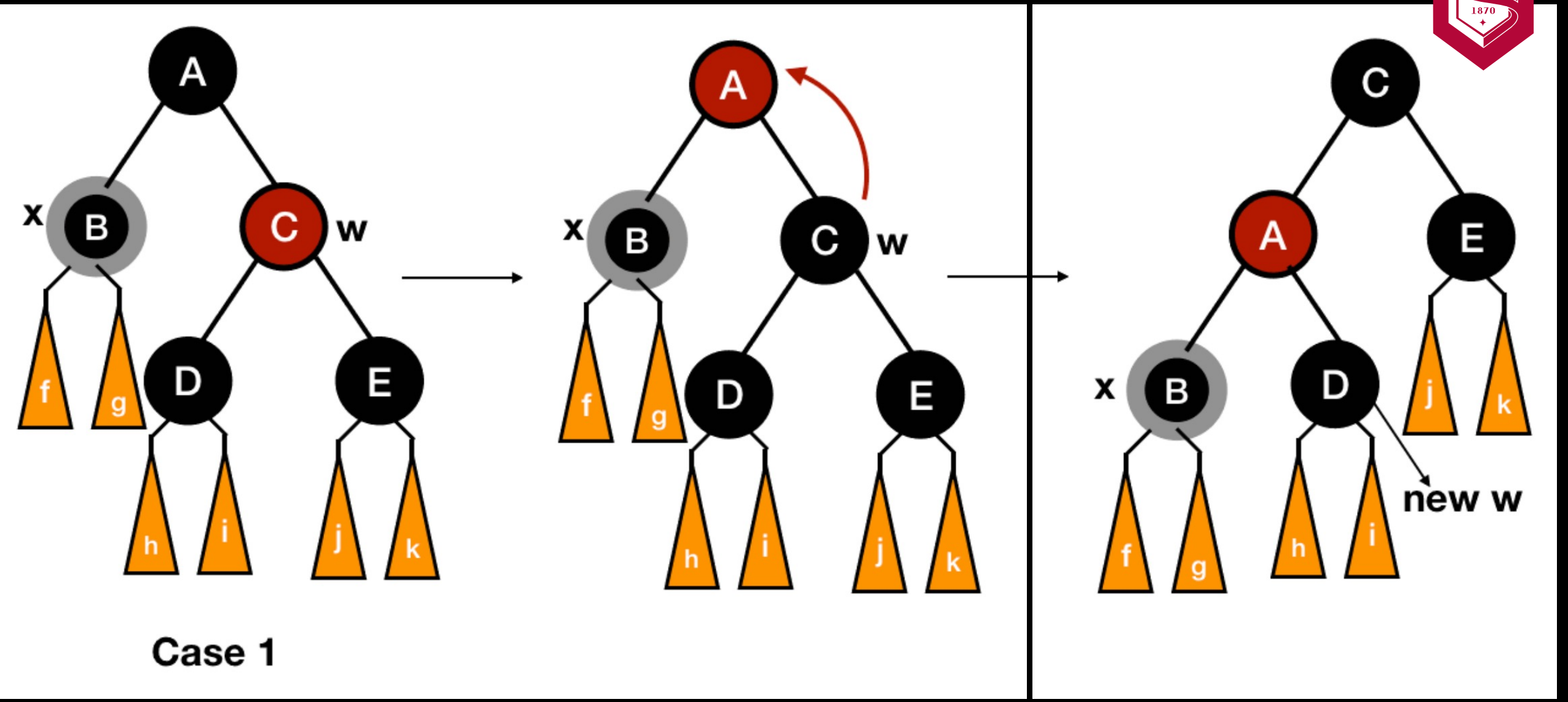
7.3.3. Deletions

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

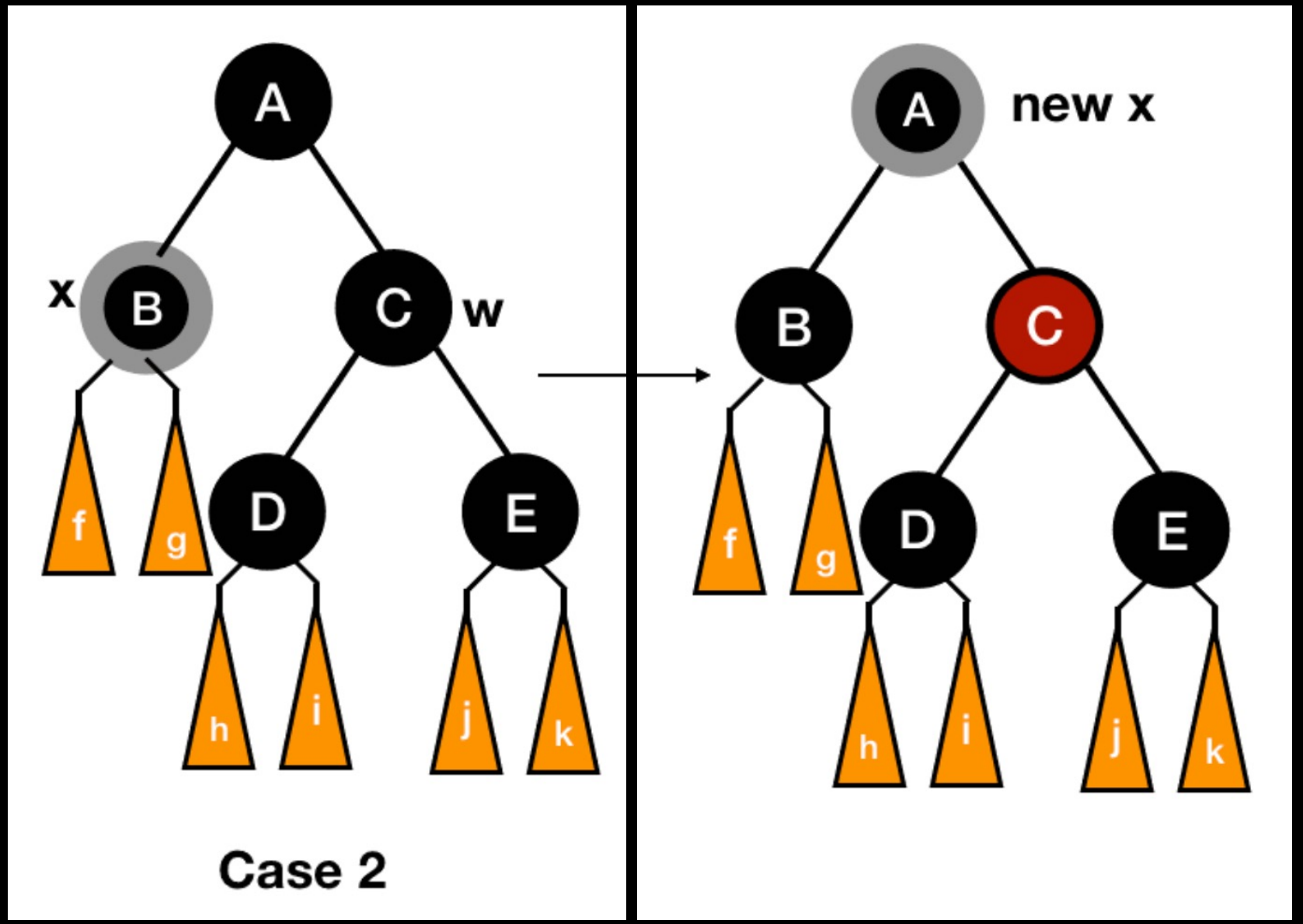


- Make w be $x.p$'s color (c).
- Make $x.p$ black and w's right child black.
- Then left rotate on $x.p$.
- Remove the extra black on x ($\Rightarrow$ x is now singly black) without violating any red-black properties.
- All done. Setting x to root causes the loop to terminate.

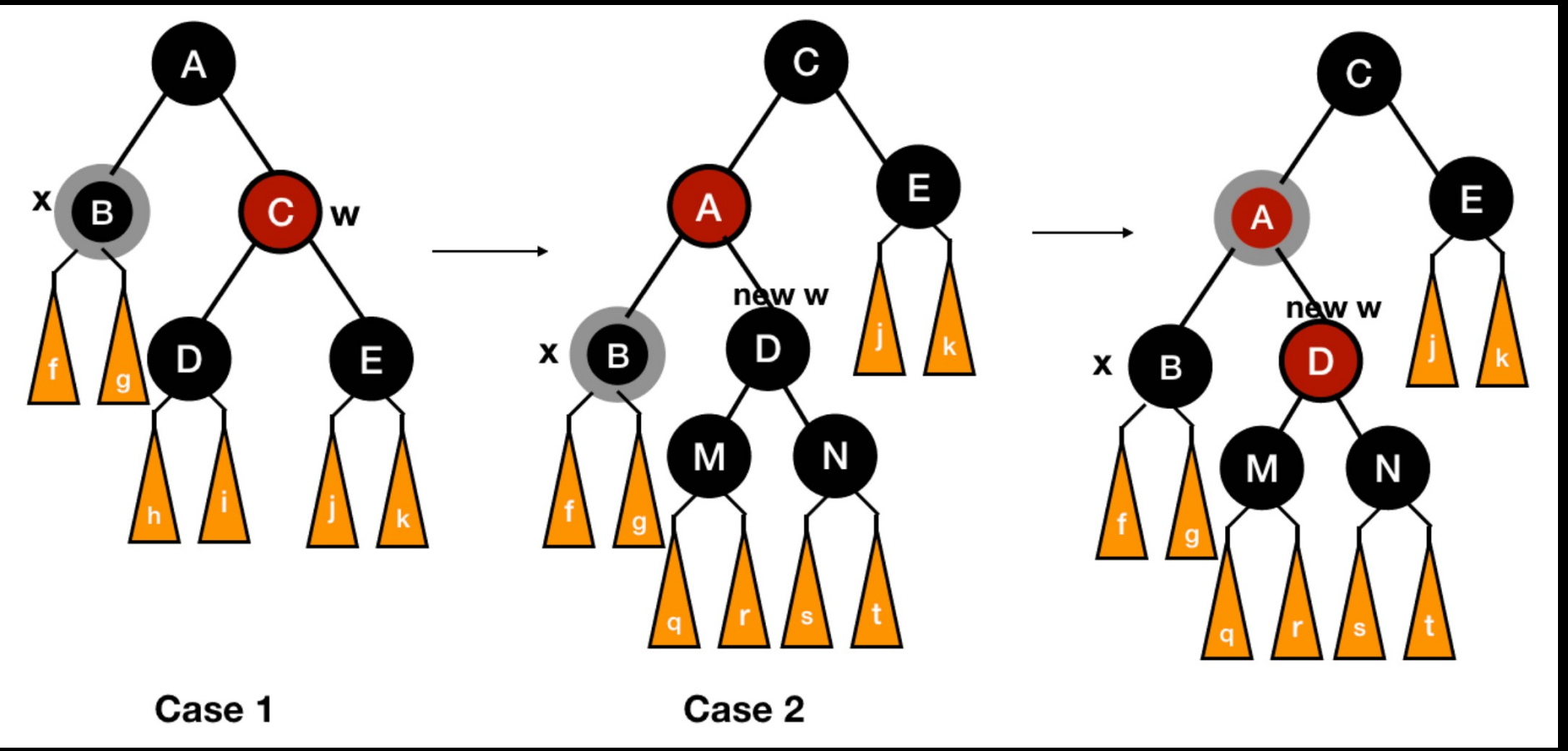
7.3.3. Deletions

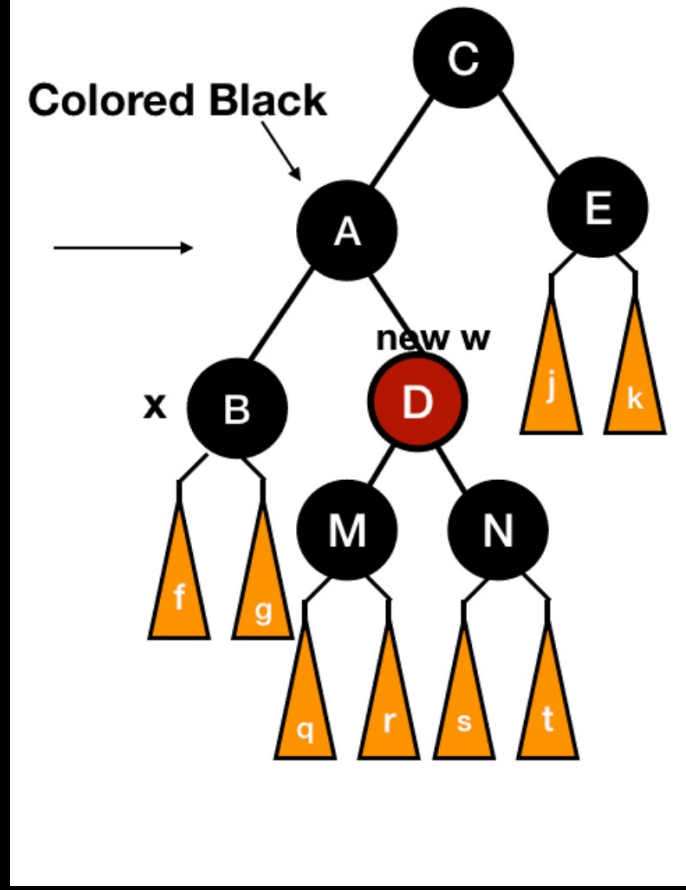


7.3.3. Deletions

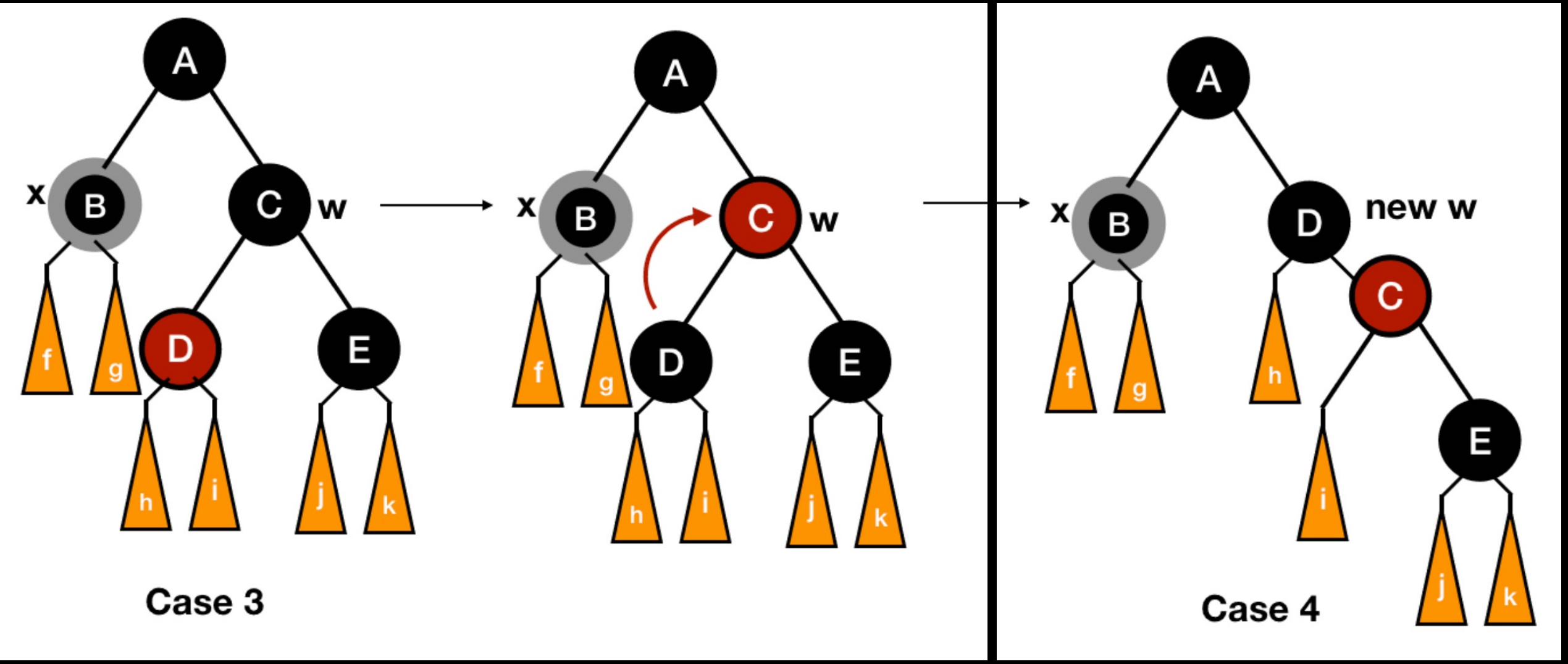


7.3.3. Deletions

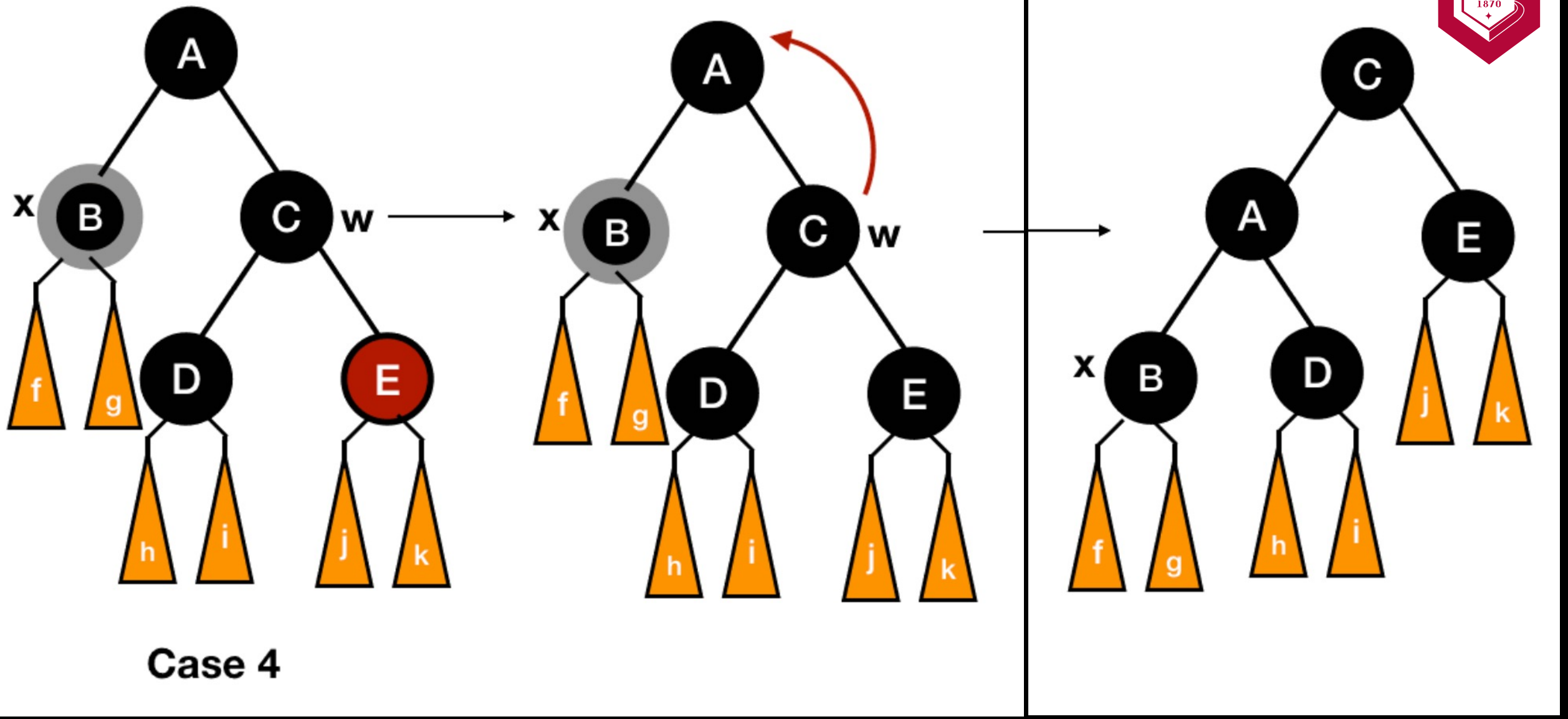




7.3.3. Deletions



7.3.3. Deletions



Case 4

7.3.3. Deletions

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 re-colorings and finish.

Within the **while** loop:

- x always points to a non-root 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.

```
     RB-DELETE-FIXUP(T,x)
(1)  while (x ≠ T.root and x.color == BLACK) do
(2)      if (x == x.p.left) then
(3)        w = x.p.right
(4)        if (w.color == RED) then
(5)              w.color == BLACK                          //case 1
(6)              x.p.color = RED
(7)              LEFT-ROTATE(T, x.p)
(8)              w = x.p.right
(9)        if (w.left.color == BLACK and w.right.color == BLACK) then
(10)             w.color = RED                             //case 2
(11)             x = x.p
(12)       else if (w.right.color == BLACK) then
(13)                w.left.color = BLACK                   //case 3
(14)                w.color = RED
(15)               RIGHT-ROTATE(T,w)
(16)               w = x.p.right
(17)           w.color = x.p.color                        //case 4
(18)           x.p.color = BLACK
(19)           w.right.color = BLACK
(20)           LEFT-ROTATE(T, x.p)
(21)           x = T.root
(22)     else (same as then clause with "right" and "left" exchanged)
(23) x.color = BLACK
```

7.3.3. Deletions

Analysis

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

Within RB-DELETE-FIXUP:

1. Case 2 is the only case in which more iterations occur.
   - x moves up 1 level.
   - Hence, $O(\lg n)$ iterations.
2. Each of cases 1, 3, and 4 has 1 rotation $\Rightarrow \leq 3$ rotations in all.
3. Hence, $O(\lg n)$ time.