CS 590 – Algorithms

Lecture 7 – Red-Black Trees

Outlines

7. Red-black trees

Red-black trees

- A variation of binary search trees.
- Self-balancing BST:
  - ***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:

- All attributes of BST will be inherited in ***red-black tree (RBT)***
- But RBT will have one extra (+1) bit per node.
  - New attribution: x.color - either **red** or **black**.
- RBT uses a single sentinel, T.nil, for all the leaves.
- T.nil.color = black
- T.root.p = T.nil

7.1. Properties

RBT Color Properties:

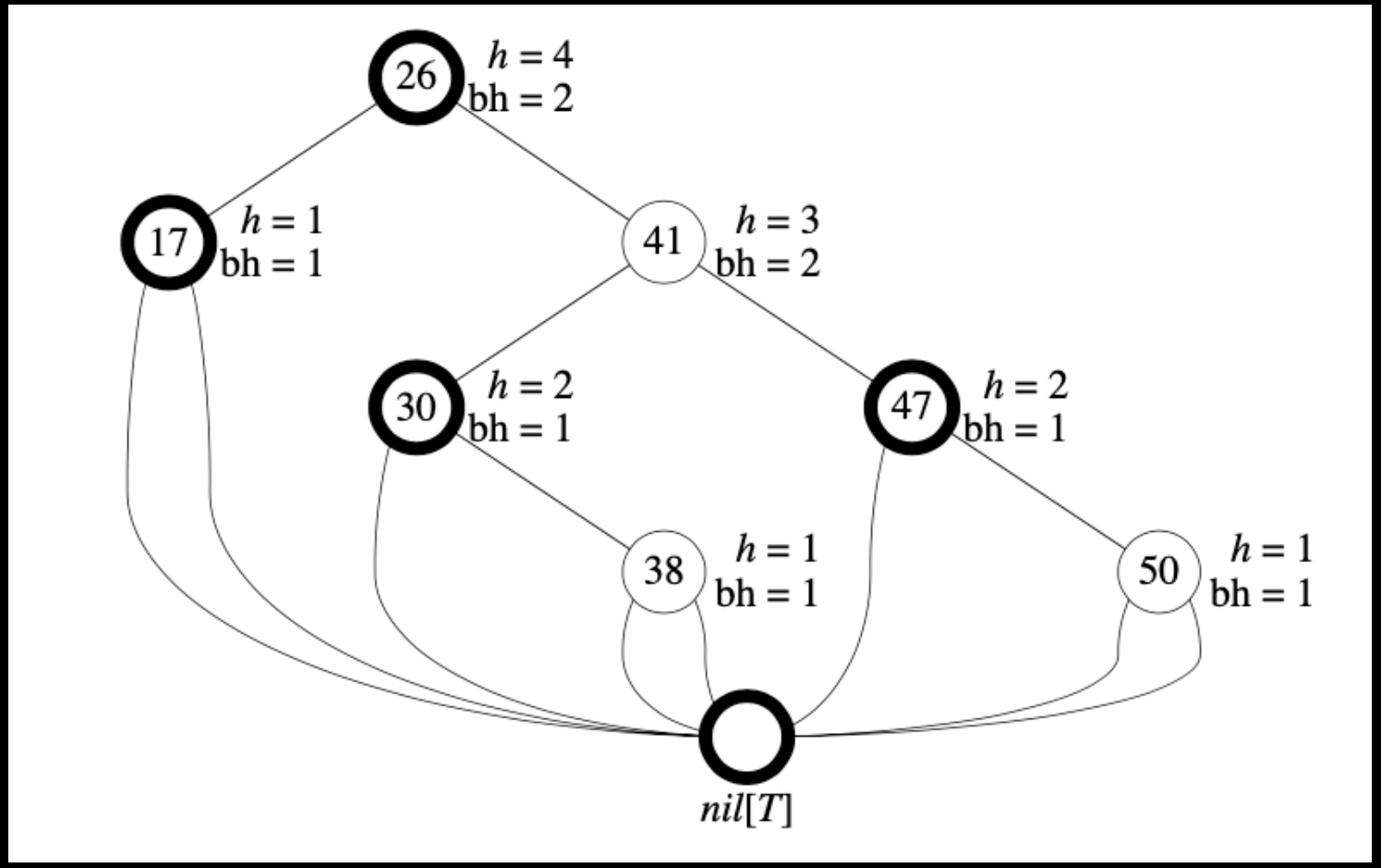
1. Every node, x.color, is either red or black.
2. T.root.color = black.
3. T.nil.color = black.
4. If x.color = red, x.left.color != red and x.right.color != red.
5. All paths from x to its descent will have the **same number of black nodes** (see black heights, bh(x)).

7.1. Properties

Black Height (bh) Properties:

- The height of the tree, h(T), is the number of edges in the longest path to a leaf.
- The black height, bh(x), is the number of black nodes from a node x to a leaf.
  - x.color is exclusive.
  - T.nil is inclusive.

7.2. Black Heights



7.2. Black Heights

- **Claim 1:**
  - If h(x) = h, bh(x) is ≥ h/2 for all x in T.
- **Claim 2:**
  - A subtree of x contains >= $2^{bh(x)} - 1$ internal nodes.
- **Lemma 1:**
  - An RBT with n-many internal nodes has h <= 2*lg(n+1).

7.3. Operations

Operations on red-black trees

- The non-modifying BST operations will be carried over.
  - Inorder-Tree-Walk, Minimum, Successor…
  - Except Insertion and Deletion.
- All searching algorithms will have the running time in O(h).

7.3.1. Rotations

Rotation Operation:

- is the basic tree-restructuring operation taking a node x within T.
- is to maintain RBTs as self-balanced BSTs.
- will use pointers to change the local pointer structure.
- must not upset the BST's property.
- operates in two directions, left and right.
- separate implementation is easier.

7.3.1. Rotations

LEFT-ROTATE(T, x) algorithm implementation.

1. Set y as x.right:
2. Turn the left subtree of y into the right of x:
3. What happens if y.left is not nil?
    - Link point x as y.left.p
4. Link x.p. to y.p
5. What if x.p = T.nil?
    - What becomes the root of the tree?
    - What if x is the left child?
    - What if x is the right child?

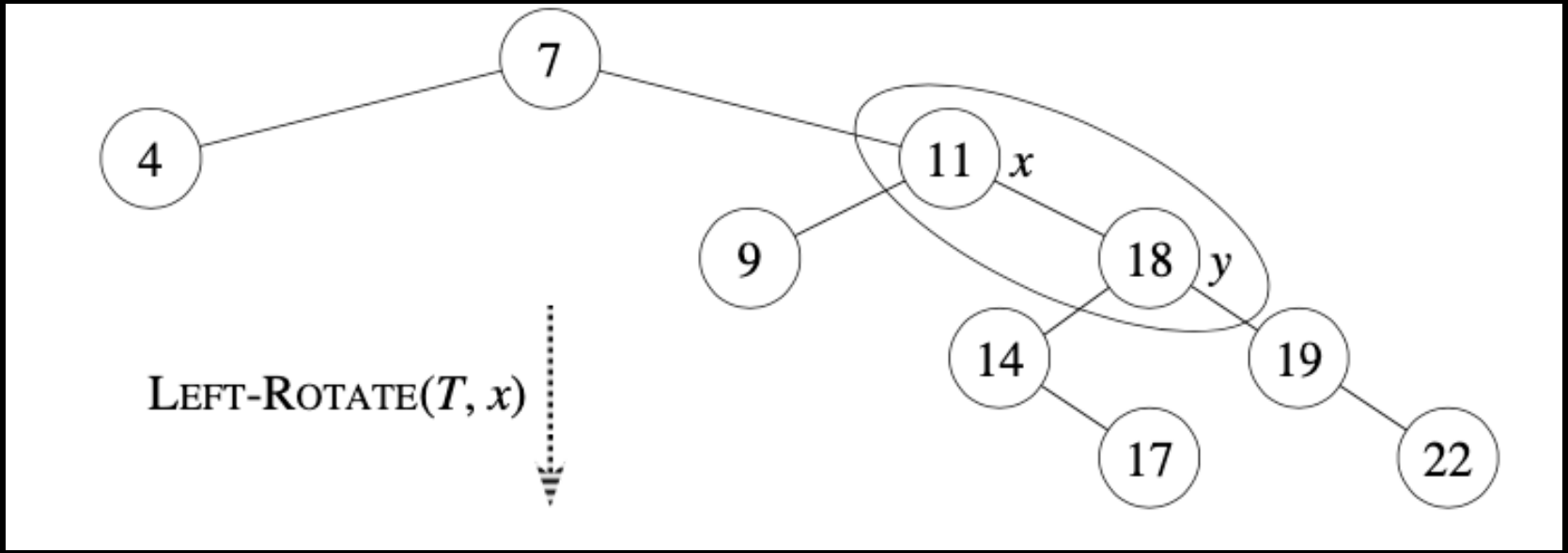
7.3.1. Rotations

```
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 != T.nil
  - root's parent is T.nil.
- The running time is T(n) = O(1).
- The RIGHT-ROTATE algorithm is symmetric:
  - exchange left and right everywhere.

7.3.1. Rotations



7.3.2. Insertion

Insertion Consideration:

- Suppose a node z is inserted into RBT.
- If z.color = Red?
  - Property 1:
  - Property 2:
  - Property 3:
  - Property 4:
  - Property 5:

7.3. Operations

Insertion Consideration:

- Suppose a node z is inserted into RBT.
- If z.color = Black?
  - Property 1:
  - Property 2:
  - Property 3:
  - Property 4:
  - Property 5:

7.3.2. Insertion

- Recall BST insertion:
  - Two pointers, x and y, were used to find the location.
- A new node z will be inserted as BST insertion. Why?
- z.color?
- We need an additional algorithm to fix to maintain RBT color properties.

7.3.2. Insertion

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
```

7.3.2. Insertion

- The color properties are violated when z.color = z.p.color = red.
- The new balancing considers the following cases depending on the color of z's uncle, y.color.
  1. z.p = z.p.p.left & y.color = red (case 1)
     1. z = z.p.right & y.color = black (case 2)
     2. z = z.p.left & y.color = black (case 3)
  2. Symmetric cases when z.p = z.p.p.right.

7.3.2. Insertion

Case 1: z.p = z.p.p.left and y.color = red

- Assume z.p.p.color = black.
- **Make z.p.color = y.color = black.**
  - Property 4:
  - Property 5:
- **Make z.p.p.color = red**.
- Let new z be z.p.p.

7.3.2. Insertion

Case 2: z = z.p.right and y.color = black

- **Let z = z.p then**
- **LEFT-ROTATE(T, z)**
  - Property 4:
  - Property 5:
- Transforms to case 3: z = z.p.left

7.3.2. Insertion

Case 3: z = z.p.left and y.color = black

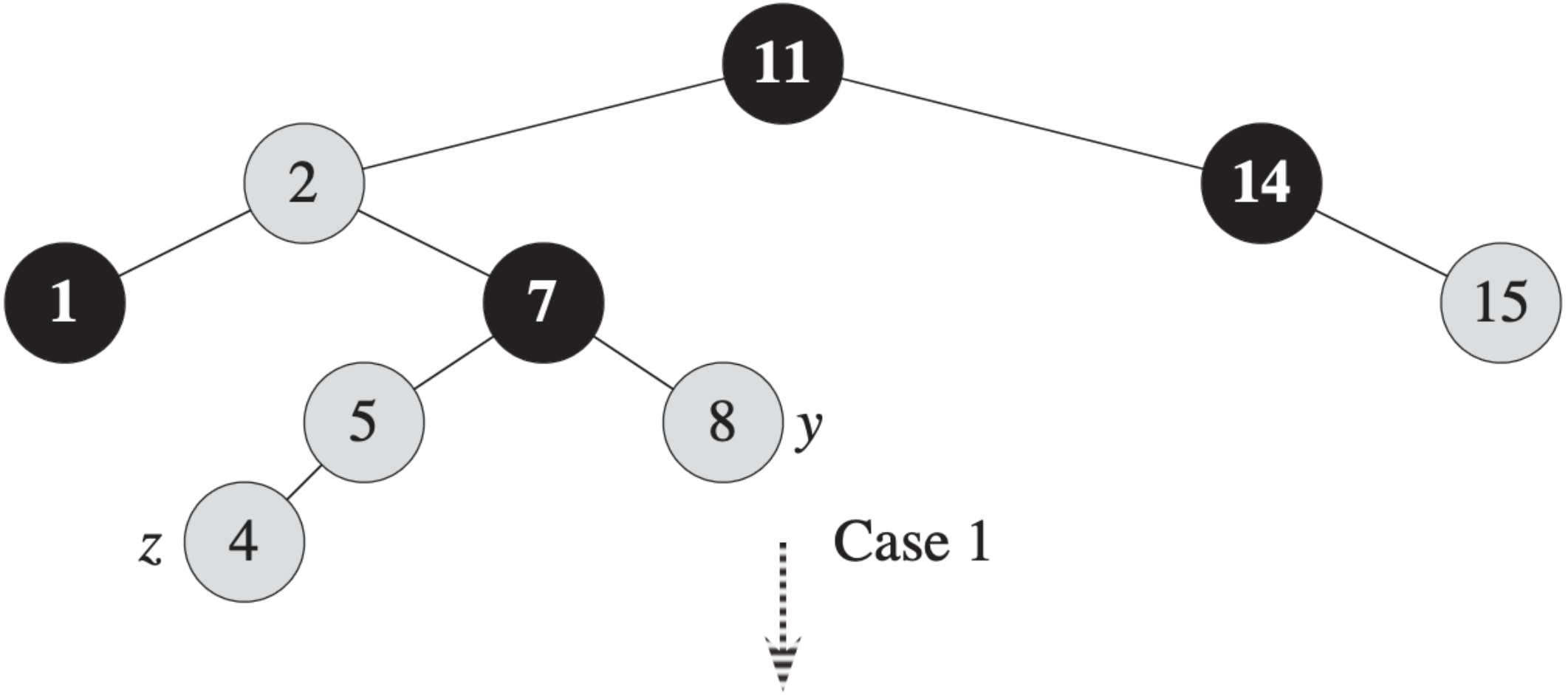
- **Make z.p.color = red and z.p.p.color = balck**
- **RIGHT-ROTATE(T, z.p.p)**
  - Property 4:
  - Property 5:
- Transforms to case 3: z = z.p.left

7.3.2. Insertion

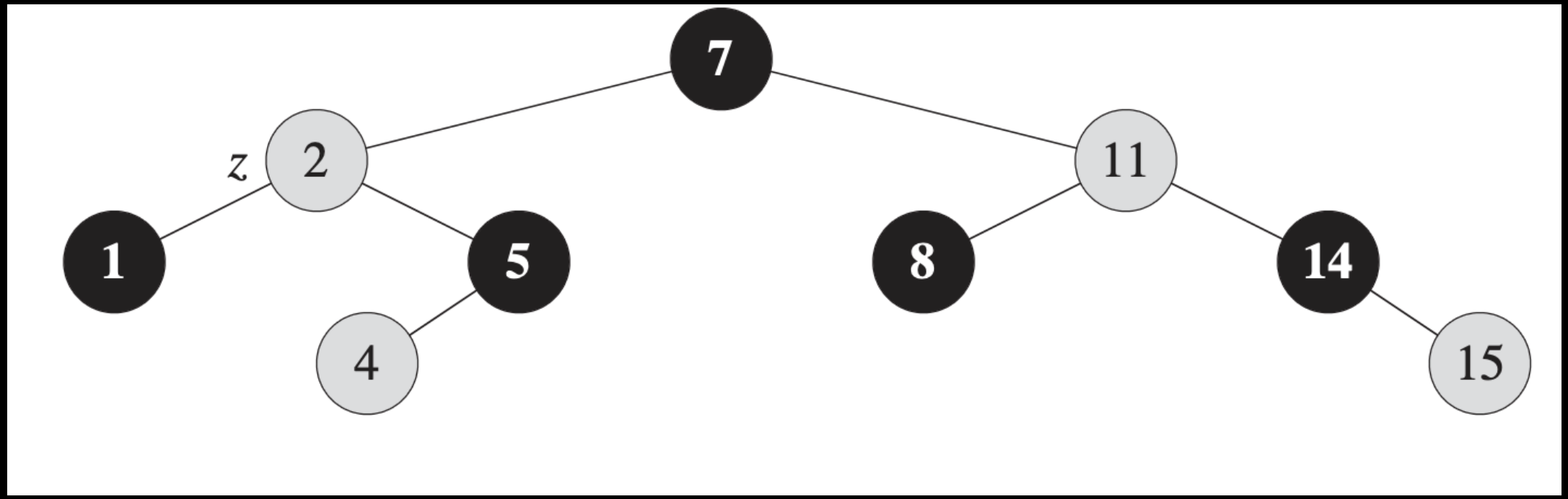
Algorithm (RB-**INSERT-FIXUP(T,z)**)

```
(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, 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. Insertion



7.3.2. Insertion



7.3.2. 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.

7.3. Operations

Deletion Consideration:

- Suppose a node z is removed.
  - If z.color = red?
    - Property 1:
    - Property 2:
    - Property 3:
    - Property 4:
    - Property 5:

7.3. Operations

Deletion Consideration:

- Suppose a node z is removed.
  - If z.color = black?
    - Property 1:
    - Property 2:
    - Property 3:
    - Property 4:
    - Property 5:

7.3.3. Deletion

- Recall BST deletion operations:
  - z having a single child:
  - z with two children:
- In RBT deletion,
  - Similar to BST deletion, the transplant algorithm will be used.
  - Case operations from BST deletion will be carried over.

7.3.3. Deletion

- These are the differences from the BST deletion:

1. A node y will be used as a removing node.
2. y.color must be identified at the beginning and saved.
    1. y's original color will be saved as y-original-color.
3. x will replace y (BST deletion replaces z with y)
4. Call RB_DELETE_FIXUP(T, x) if y-original-color = black.

7.3.3. Deletion

```
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
```

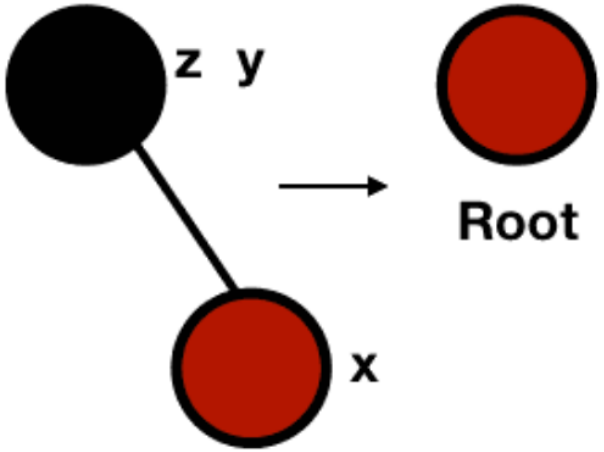
```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
     x = z.right                          // z has no left child
     RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
     x = z.left                           // just left child
     RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)            //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.

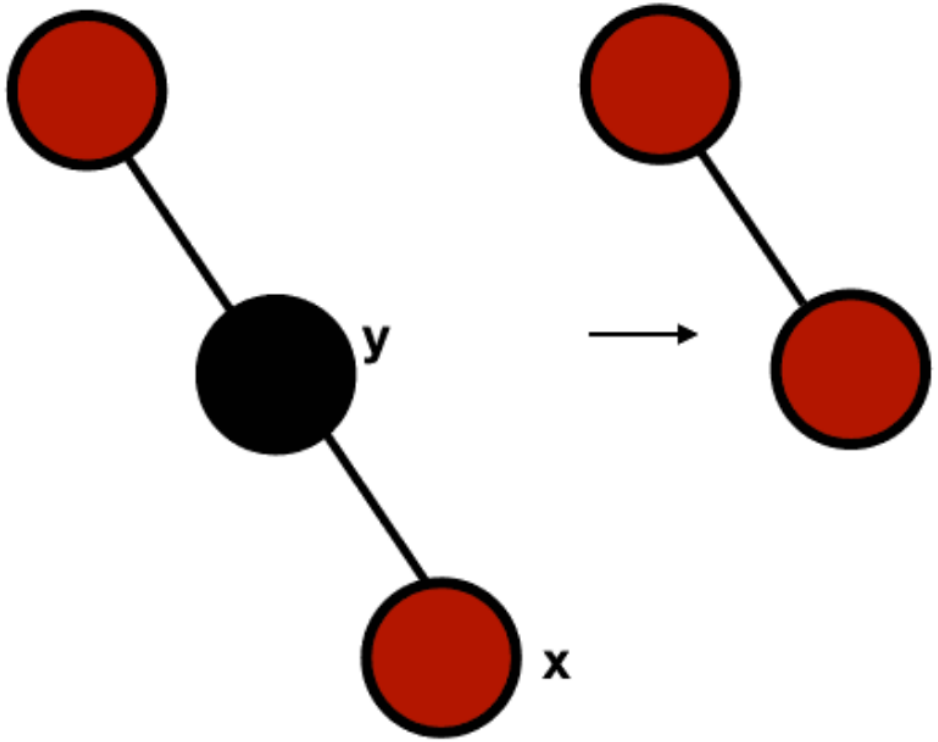
7.3.3. Deletion

- When the RBT 5th property is violated,
  - Give x an “extra black”.
  - 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. Deletion



Violation of prop. 2

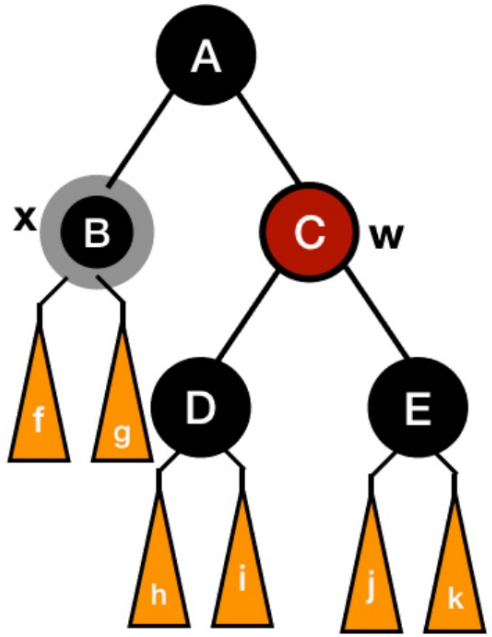


Violation of prop. 4

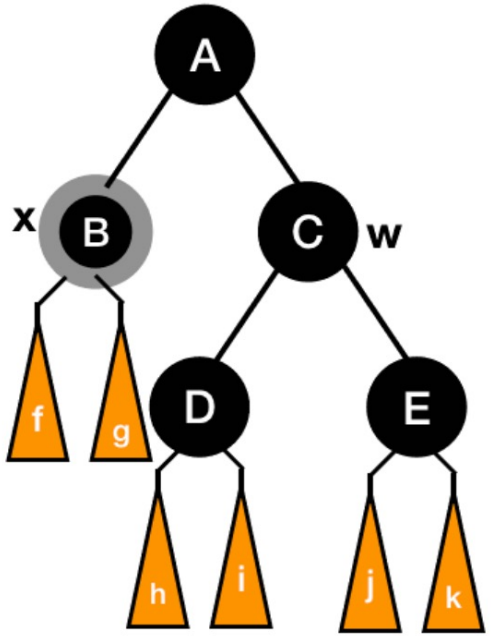
Violation of prop. 5,
black height affected

7.3.3. Deletion

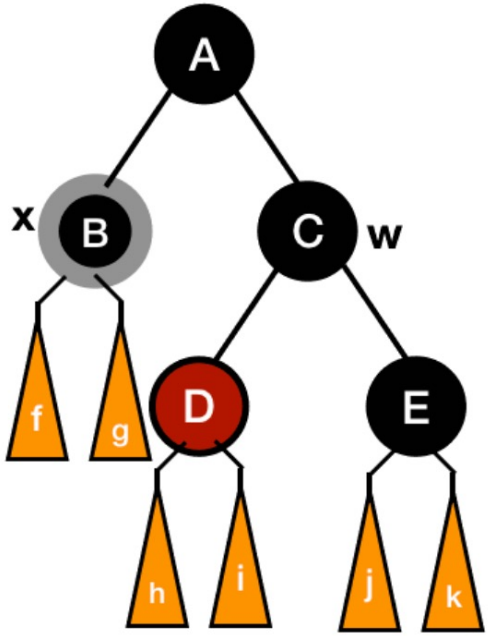
- There are four cases to consider after the deletion when x = x.p.left



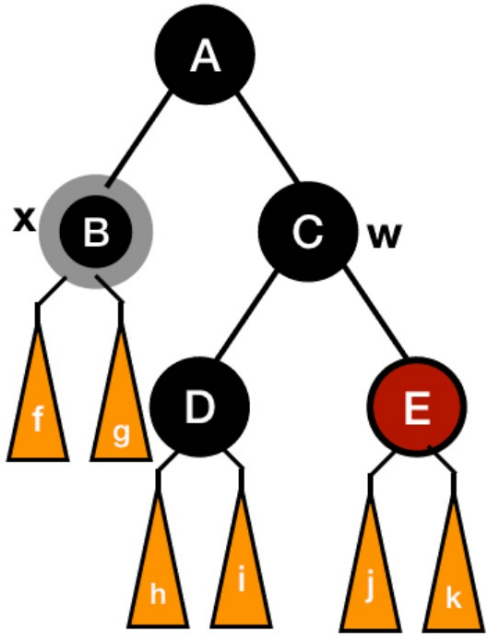
Case 1



Case 2

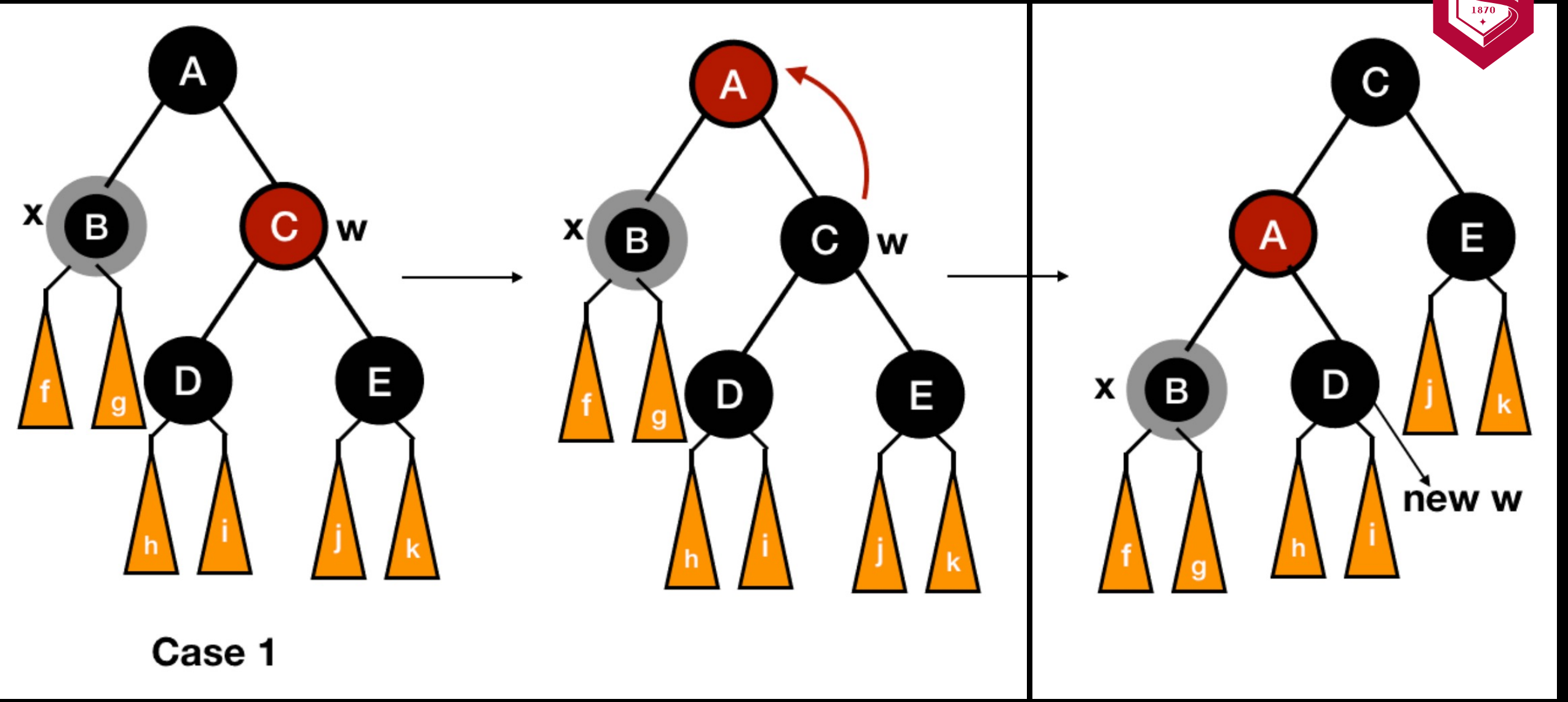


Case 3



Case 4

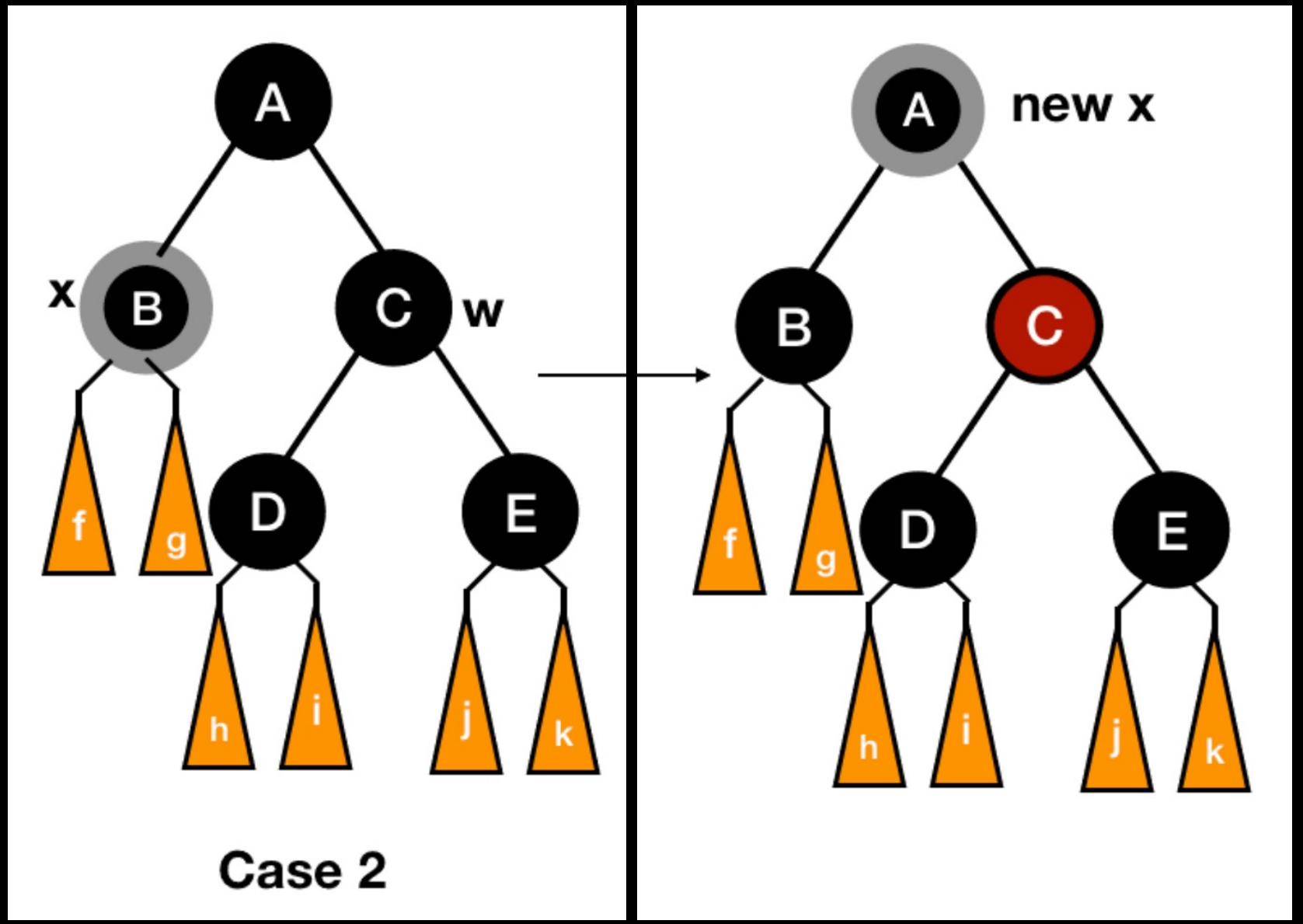
7.3.3. Deletion



7.3.3. Deletion

- w.left.color = w.right.color = black and w.color = red.
- Make w.color = black and x.p.color = red.
- Then LEFT_ROTATE(T, x.p).
- w’s child is a new sibling of x
- Make a new w = x.p.right
- Go immediately to cases 2, 3, or 4.

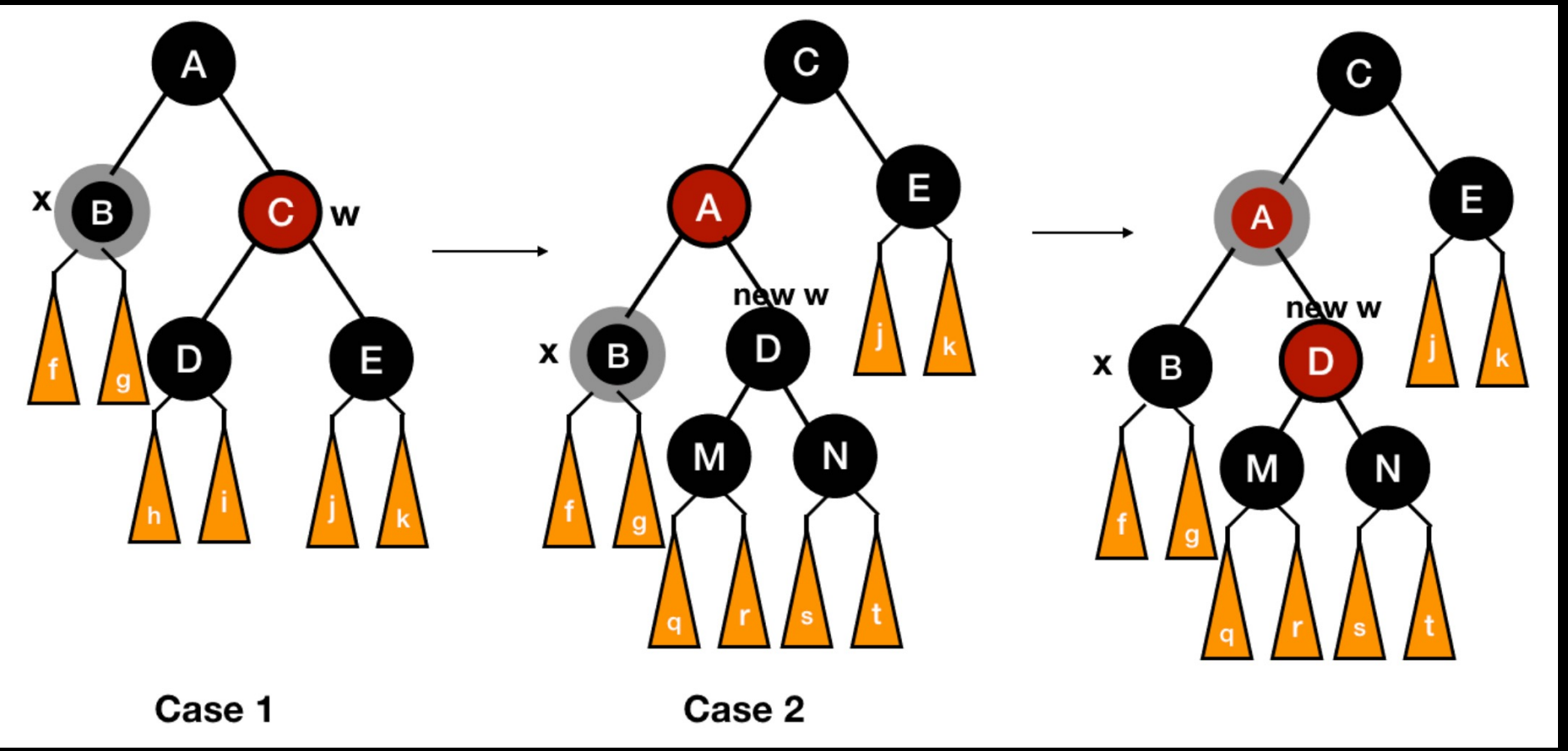
7.3.3. Deletion

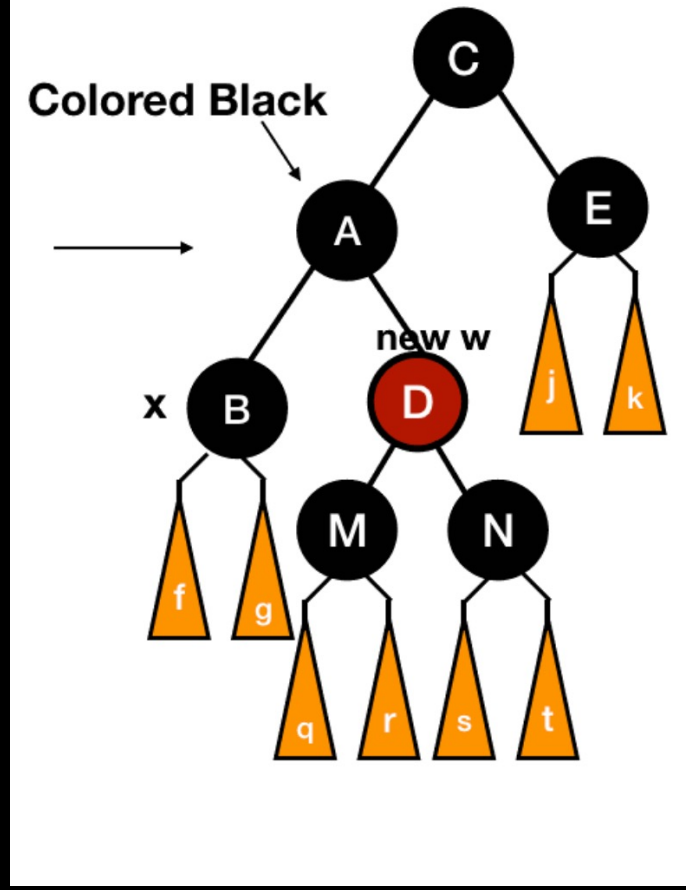


7.3.3. Deletion

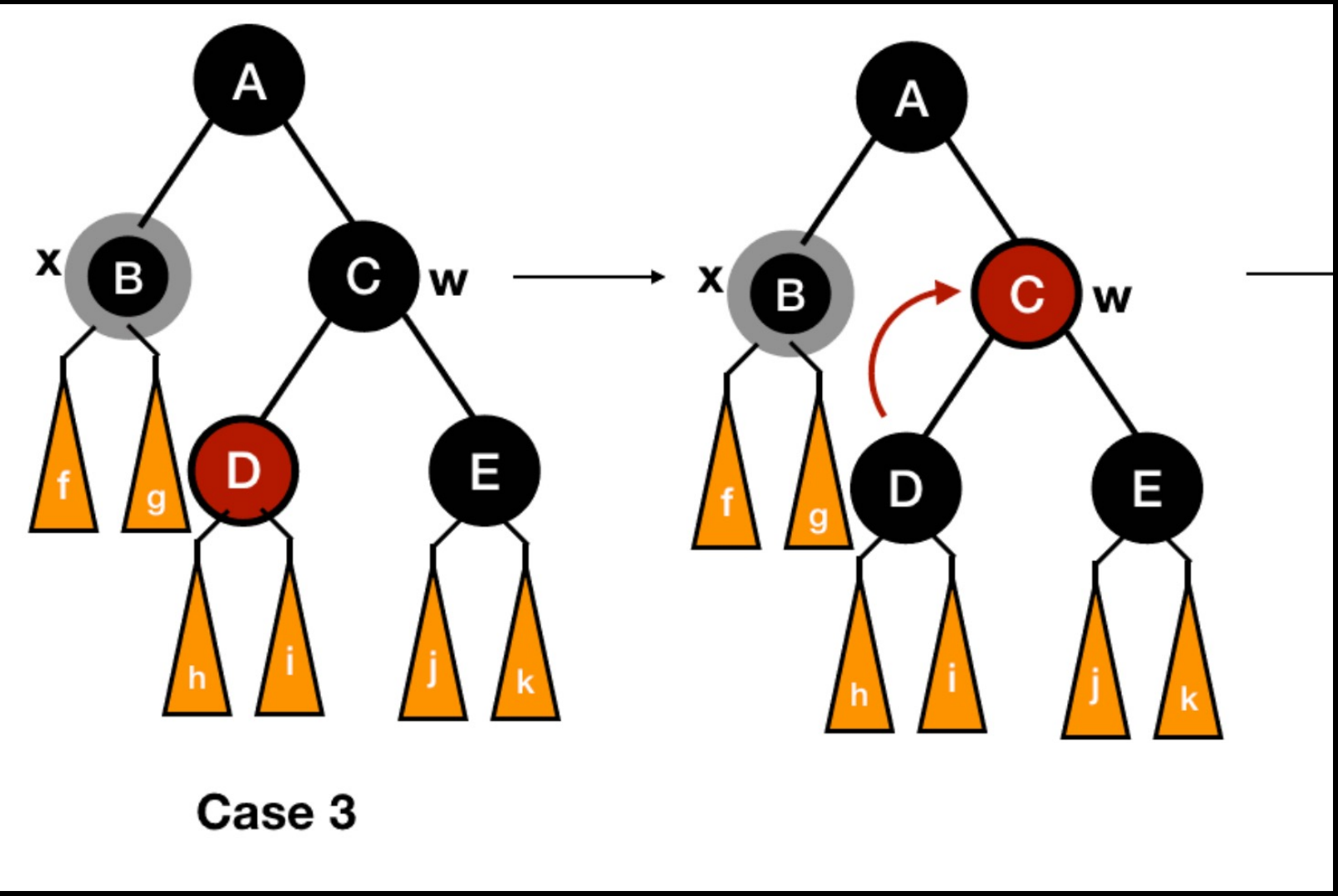
- Case 2: if w.left.color = w.right.color = w.color = black
  - Make w.color = red
  - Move up to have x = x.p

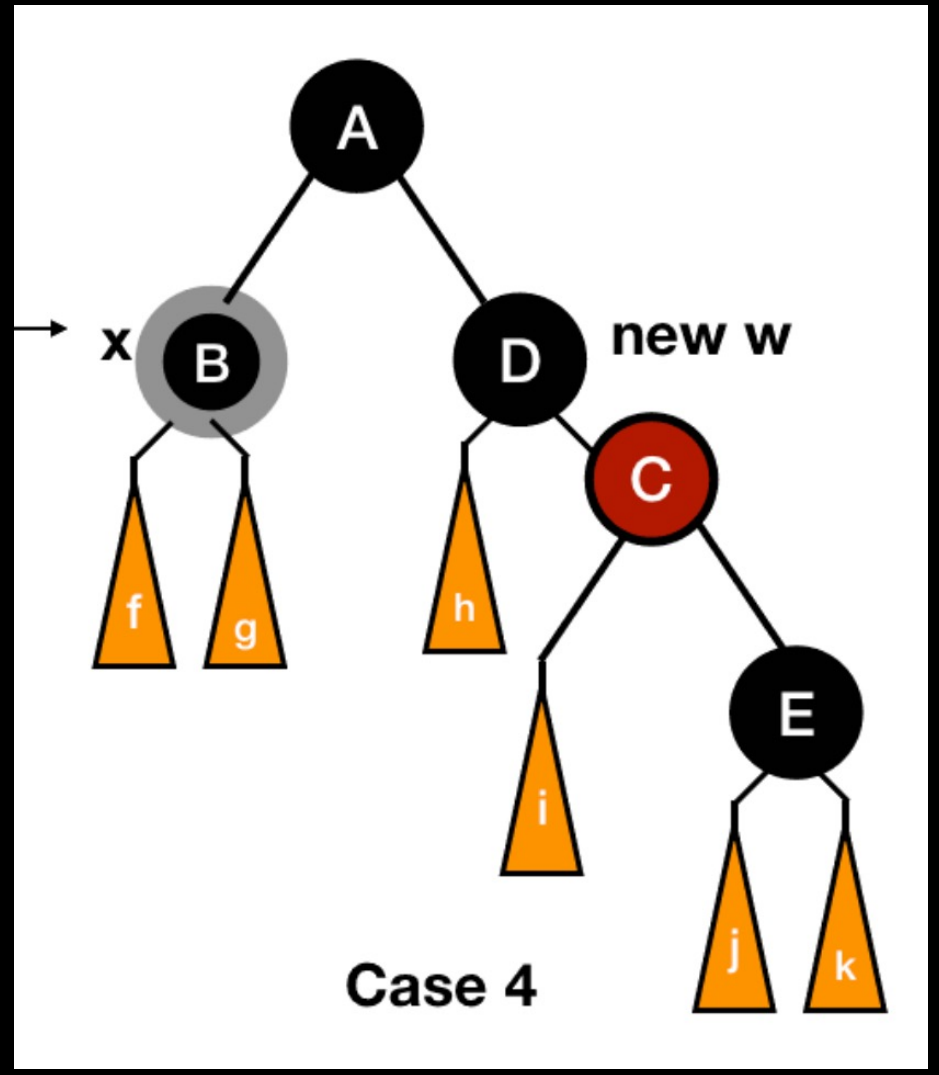
7.3.3. Deletion





7.3.3. Deletion

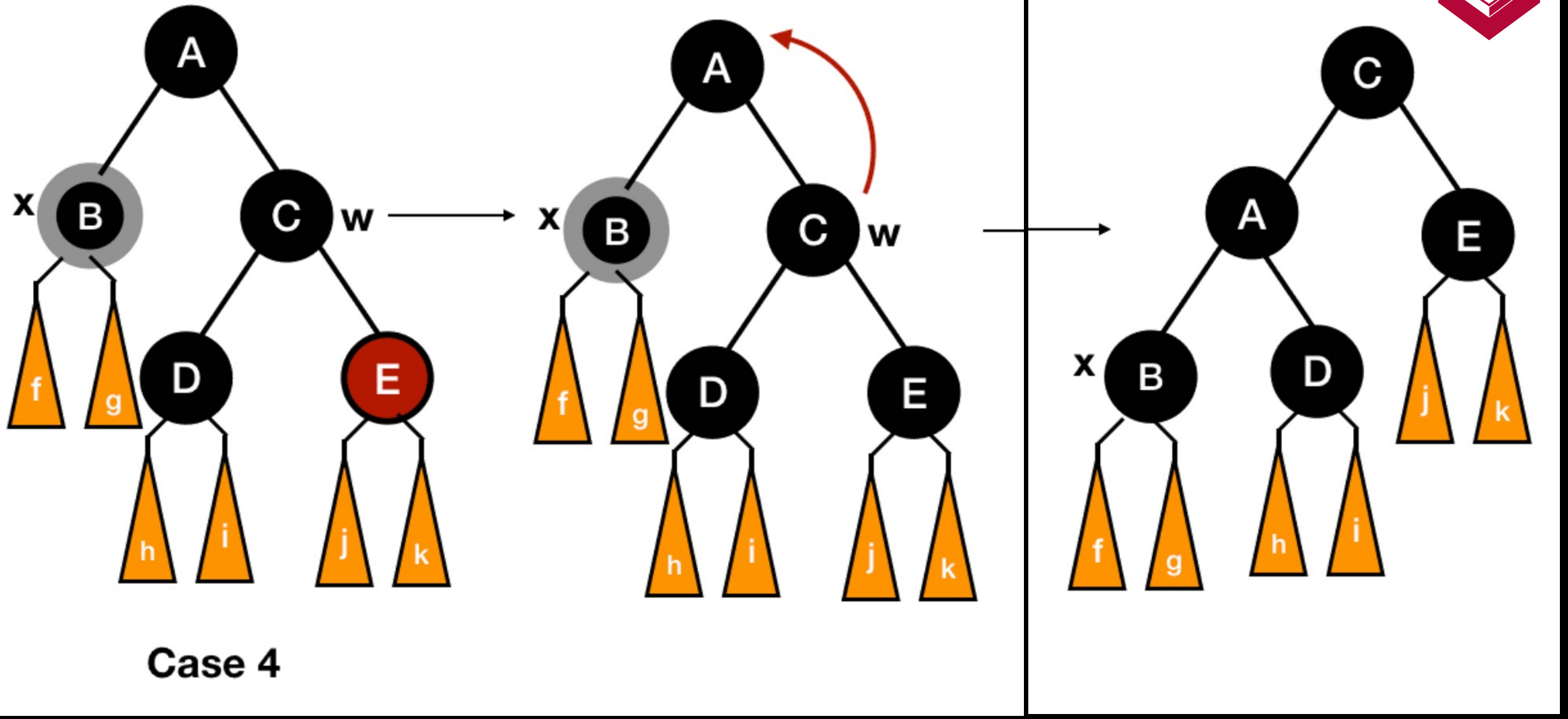




7.3.3. Deletion

- Case 3: w.color = w.right.color = black and w.left.color = red
- Make w.color =red and w.left.color = black
- Then, RIGHT_ROTATE(T, w)
- x's new sibling w.color = black and w.right.color = red

7.3.3. Deletion



Case 4

7.3.3. Deletion

- Case 4: w.color = w.left.color = black and w.right.color = red
- Make w.color = x.p.color, x.p.color = balck, w.right.color=black
- Then, LEFT_ROTATE(T, x.p).
- Remove extra black on x.
- Set x = T.root
- All done.

7.3.3. Deletion

```
     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. Deletion

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 ⇒ 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.

7.3.3. Deletion

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 case 1, 3, and 4 have 1 rotation $\Rightarrow \leq 3$ rotations.
3. Hence, $O(\lg n)$ time.