7.3. Operations

BXJU2

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

BXJU2

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

BXJU2

- 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

BXJU2

- 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)
```

BST Deletion

BXJU2

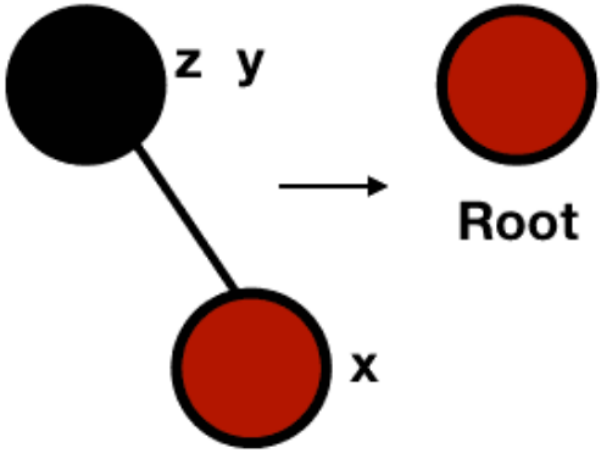
- Deleting a node from a red-black tree is a bit more complicated than inserting a node.
- The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure.

7.3.3. Deletion

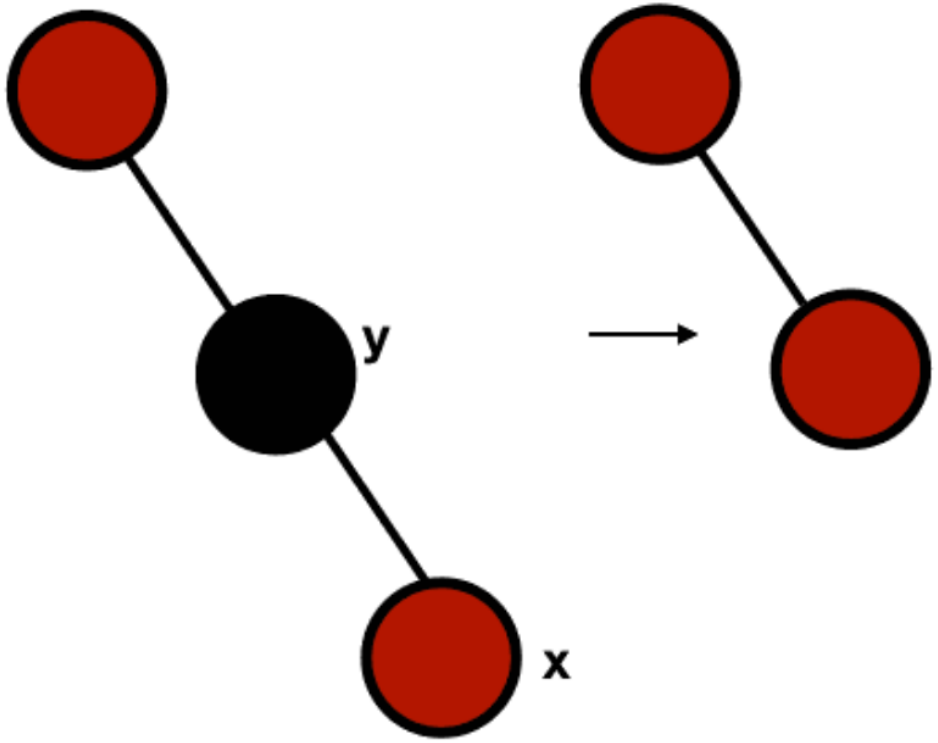
BXJU2

- 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



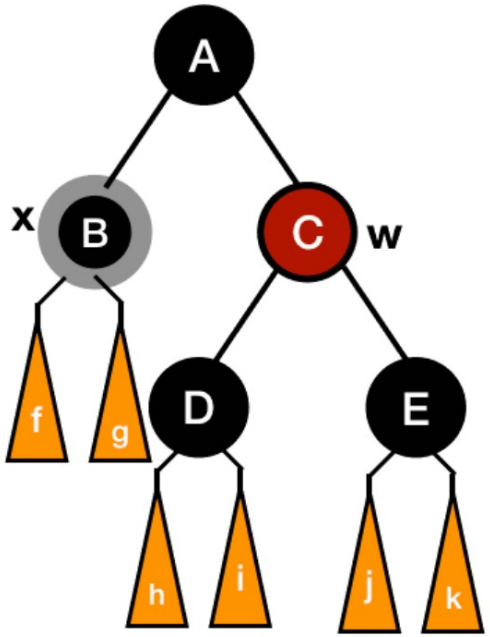
Violation of prop. 4

Violation of prop. 5,
black height affected

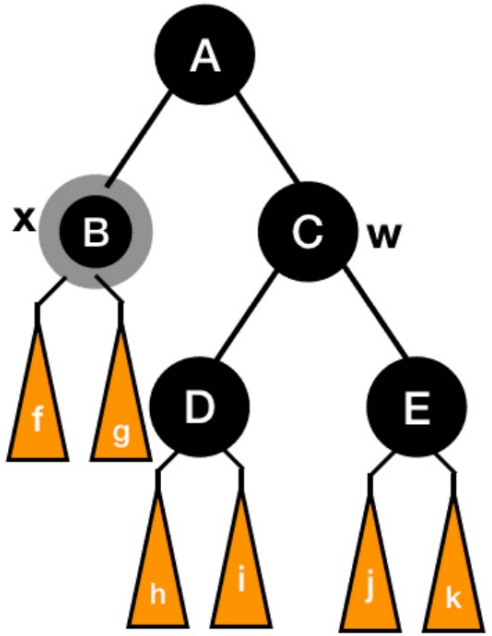
7.3.3. Deletion

BXJU2

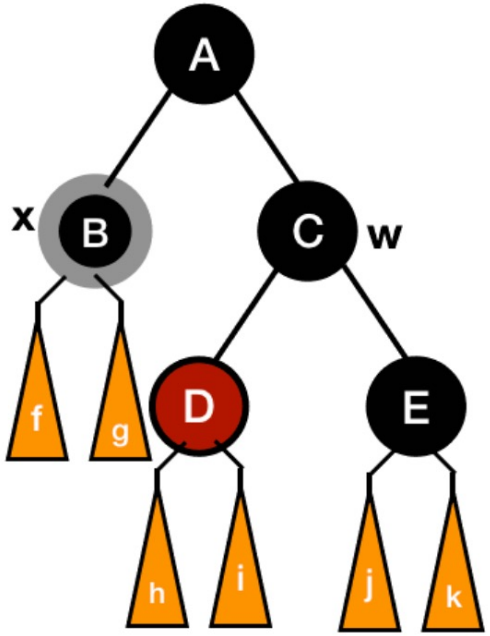
- There are four cases to consider after the deletion when x = x.p.left



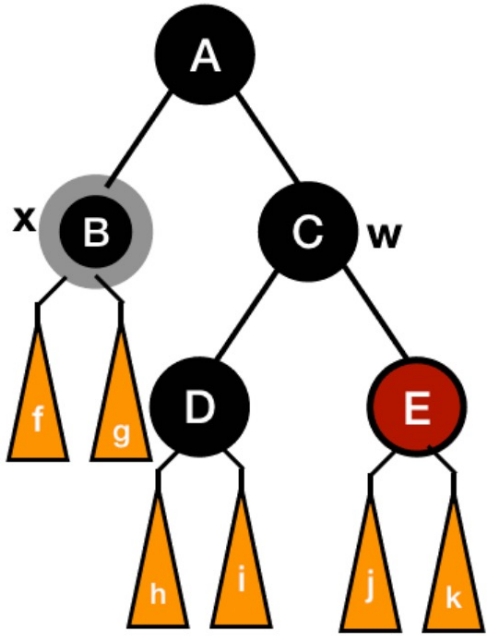
Case 1



Case 2



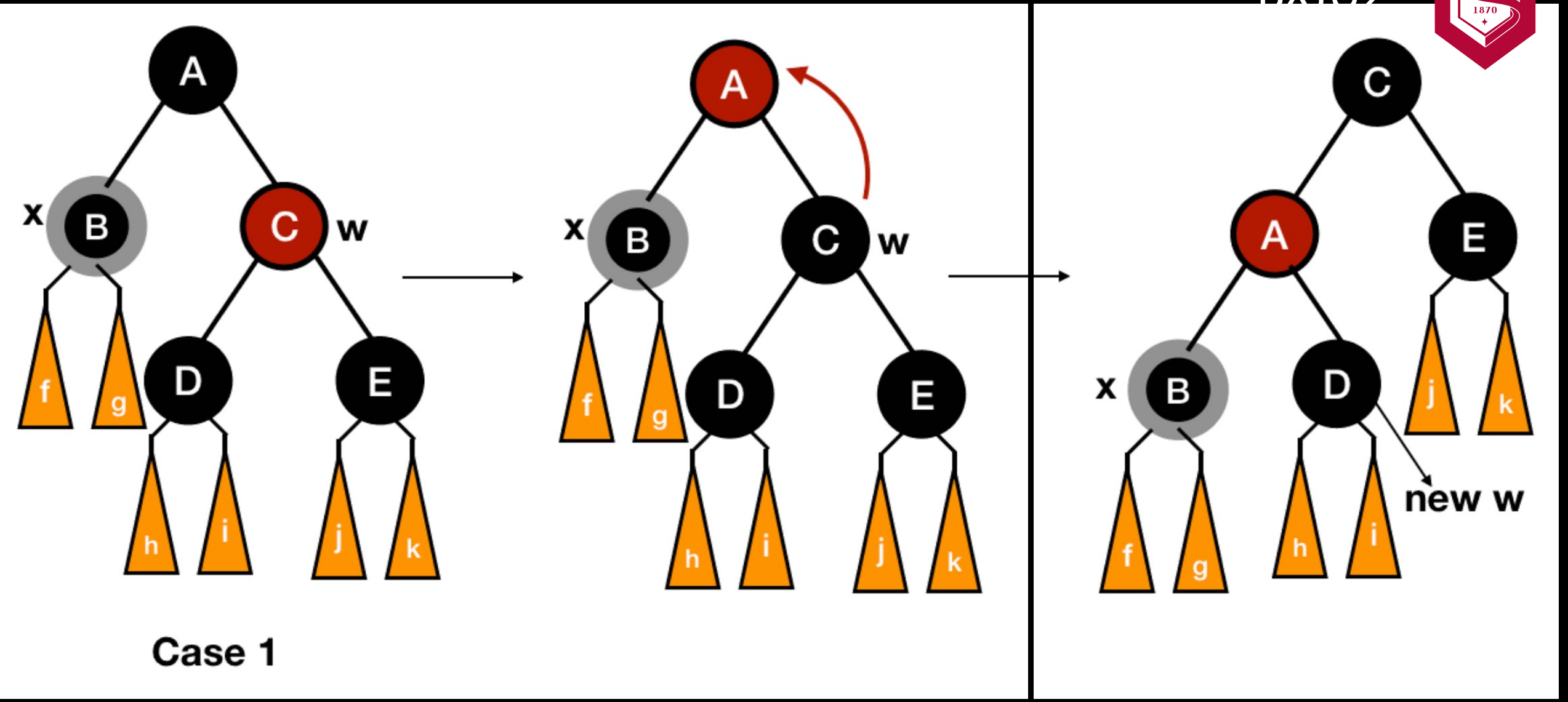
Case 3



Case 4

7.3.3. Deletion

BXJU2



7.3.3. Deletion

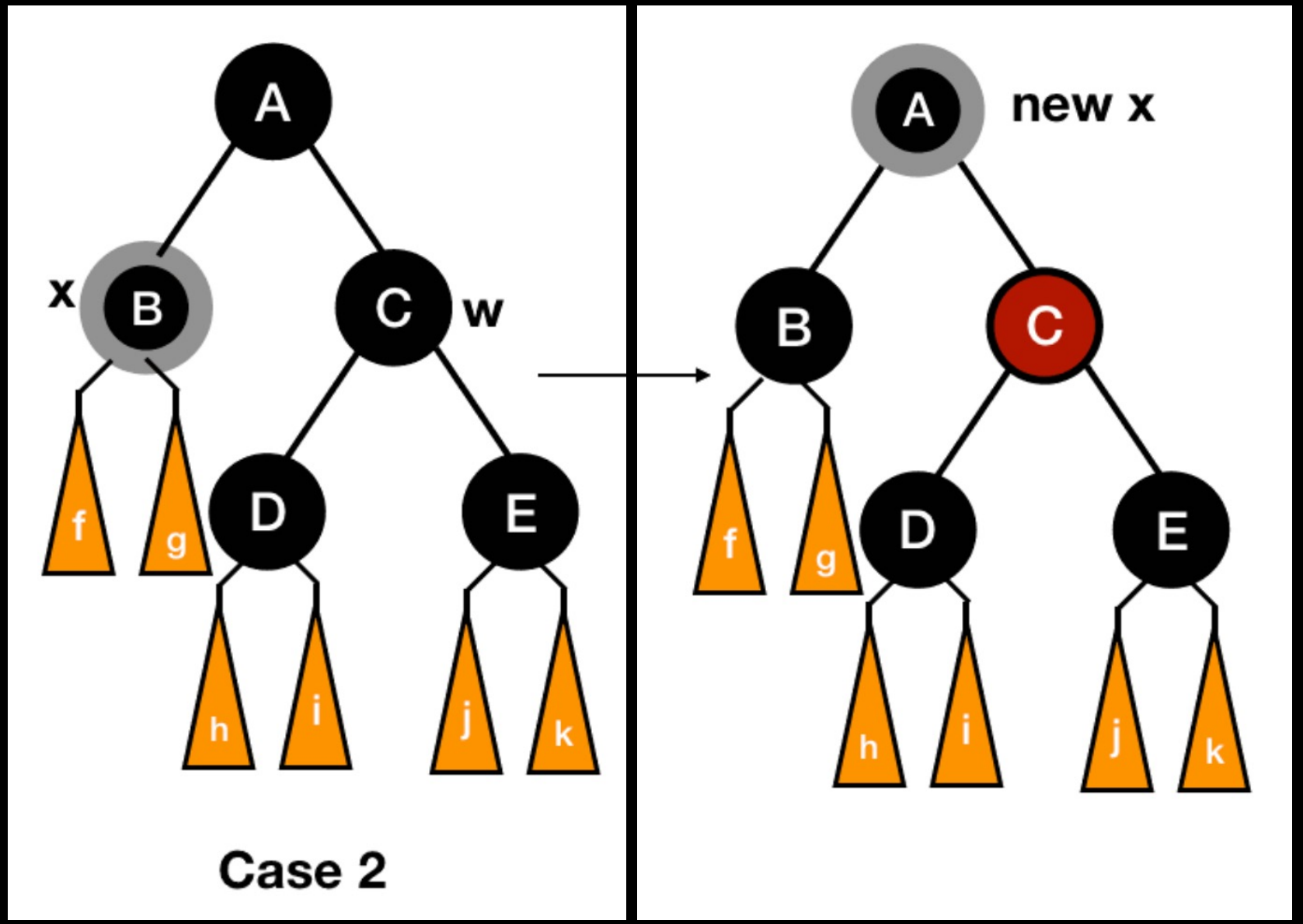
BXJU2

- 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.

w.left = x.p.right

7.3.3. Deletion

BXJU2



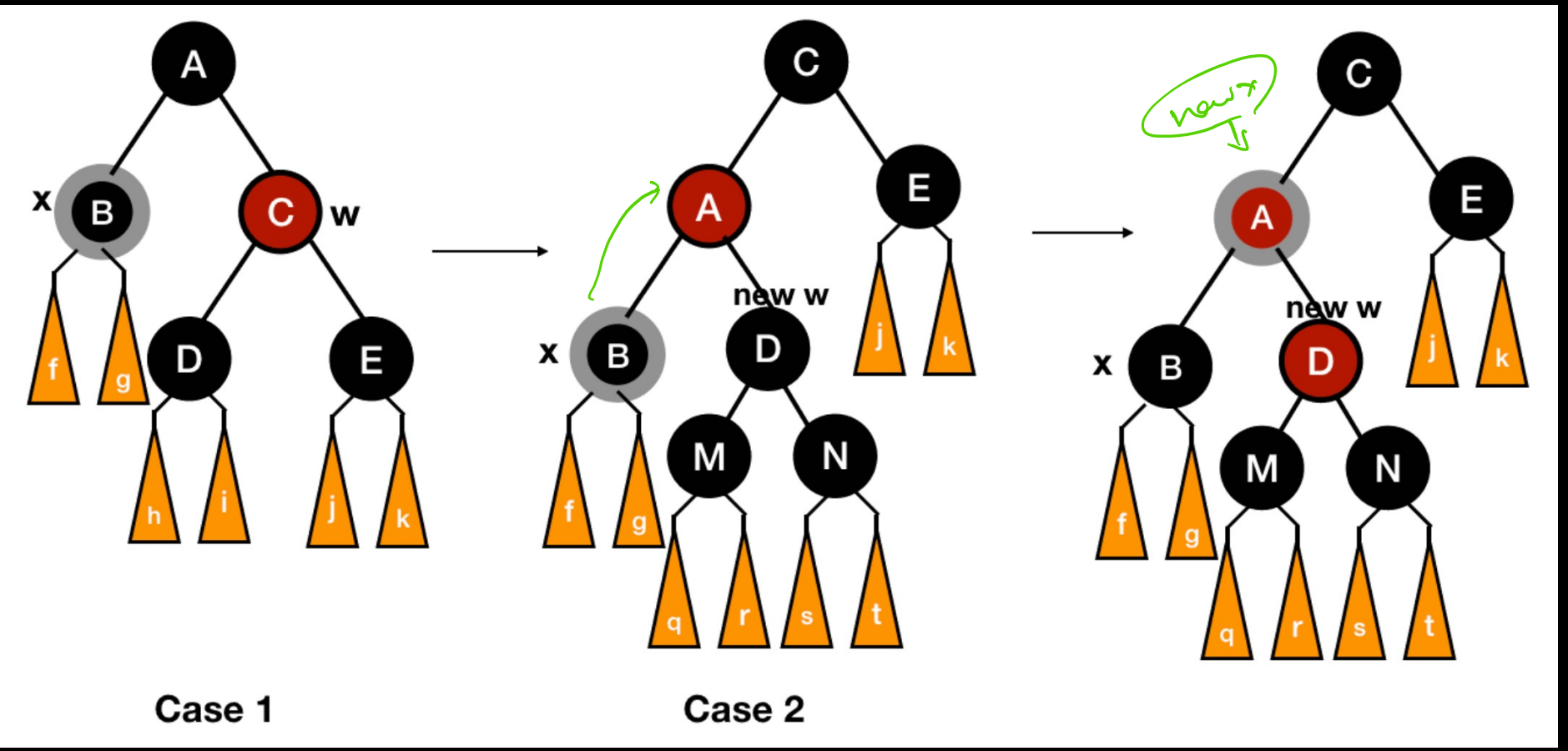
7.3.3. Deletion

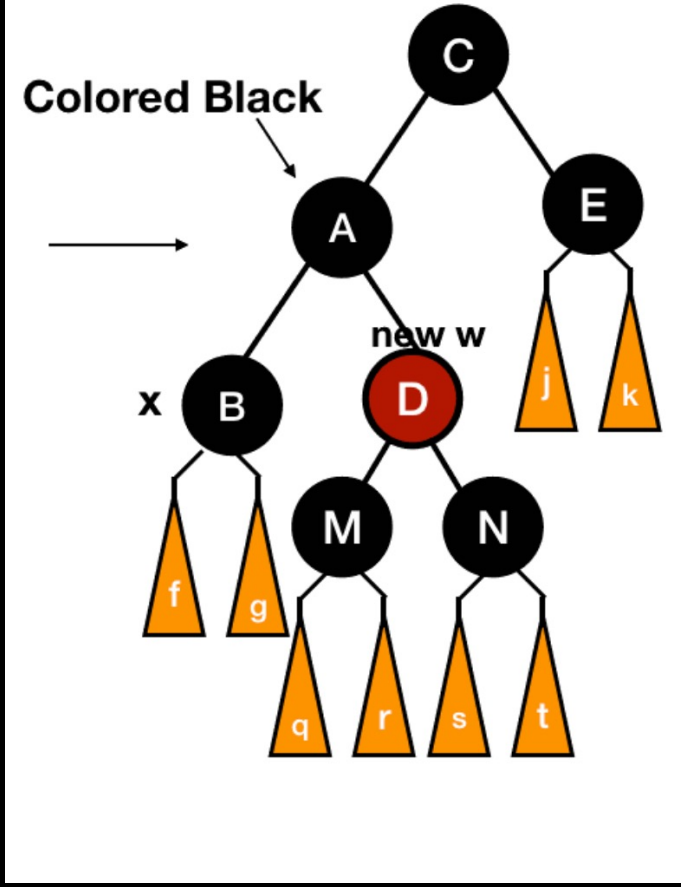
BXJU2

- 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

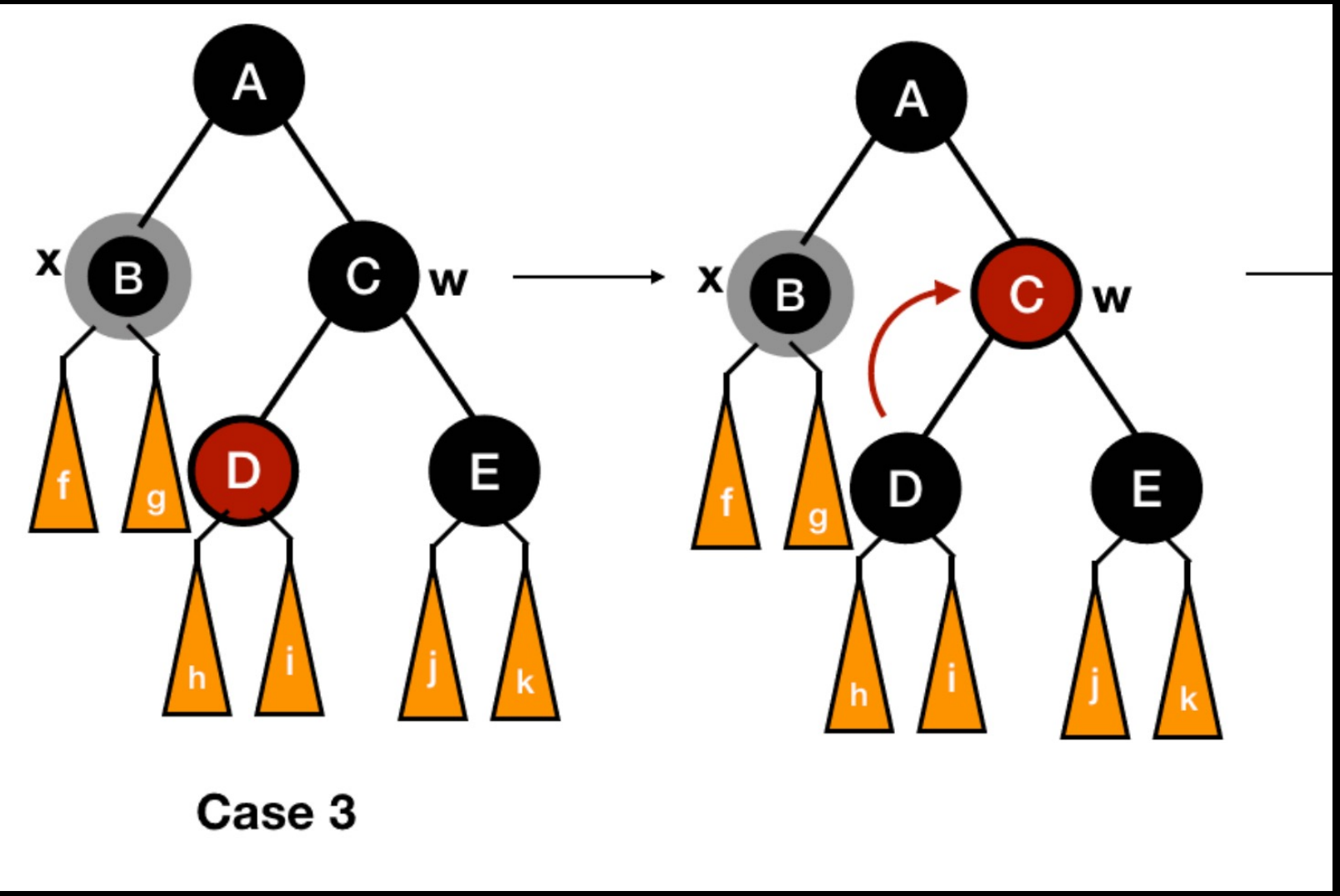
BXJU2

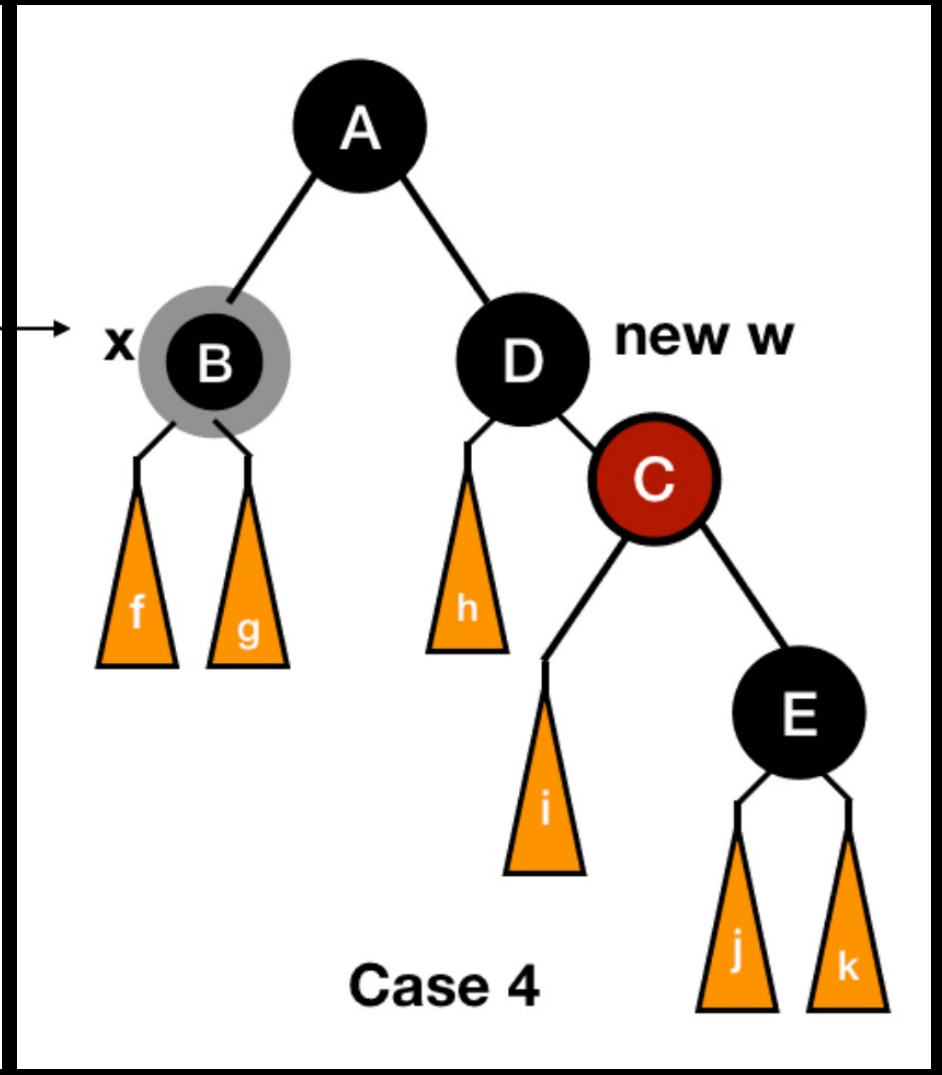




7.3.3. Deletion

BXJU2





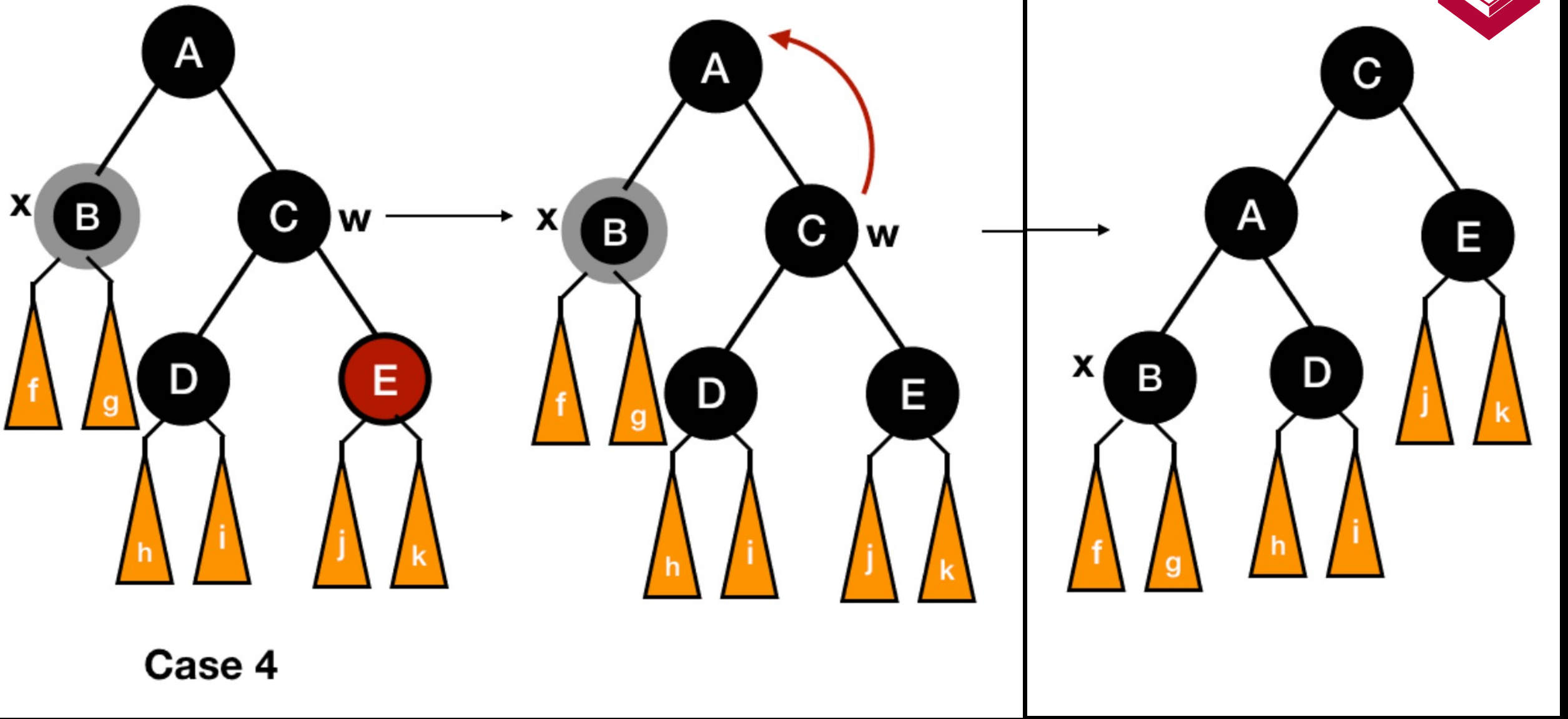
7.3.3. Deletion

BXJU2

- 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

BXJU2



Case 4

7.3.3. Deletion

BXJU2

- 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

BXJU2


```
     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

BXJU2

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

BXJU2

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.