Data Structure & Algorithms

Red Black Trees Deletion

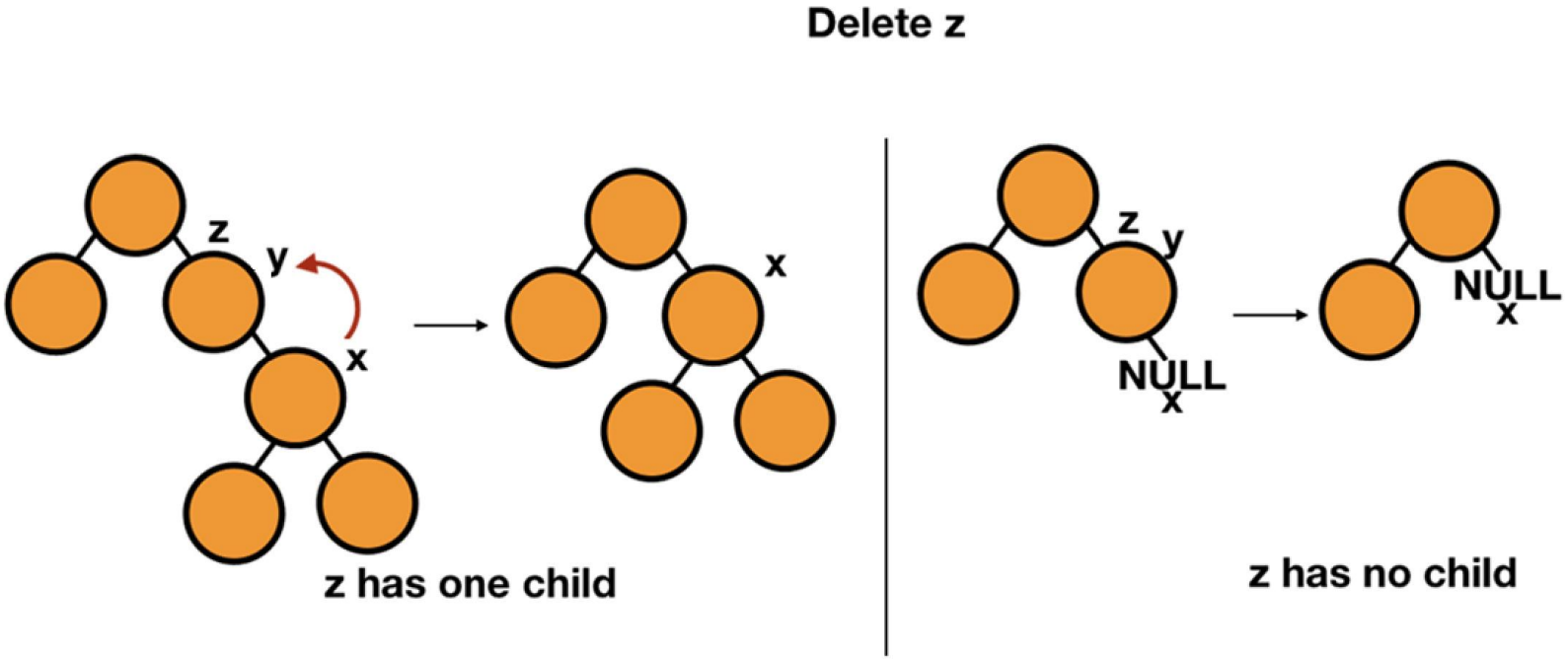
Deletion

- The deletion process in a red-black tree is also similar to the deletion process of a normal binary search tree. Similar to the insertion process, we will make a separate function to fix any violations of the properties of the **red-black tree**.
- Just go through the $\underline{Delete}$ function of binary search trees because we are going to develop the code for deletion on the basis of that only. After that, we will develop the code to fix the violations.
- On the basis of the $\underline{Transplant}$ function of a normal binary search tree, we can develop the code for the transplant process for red-black trees as shown in the next slide.

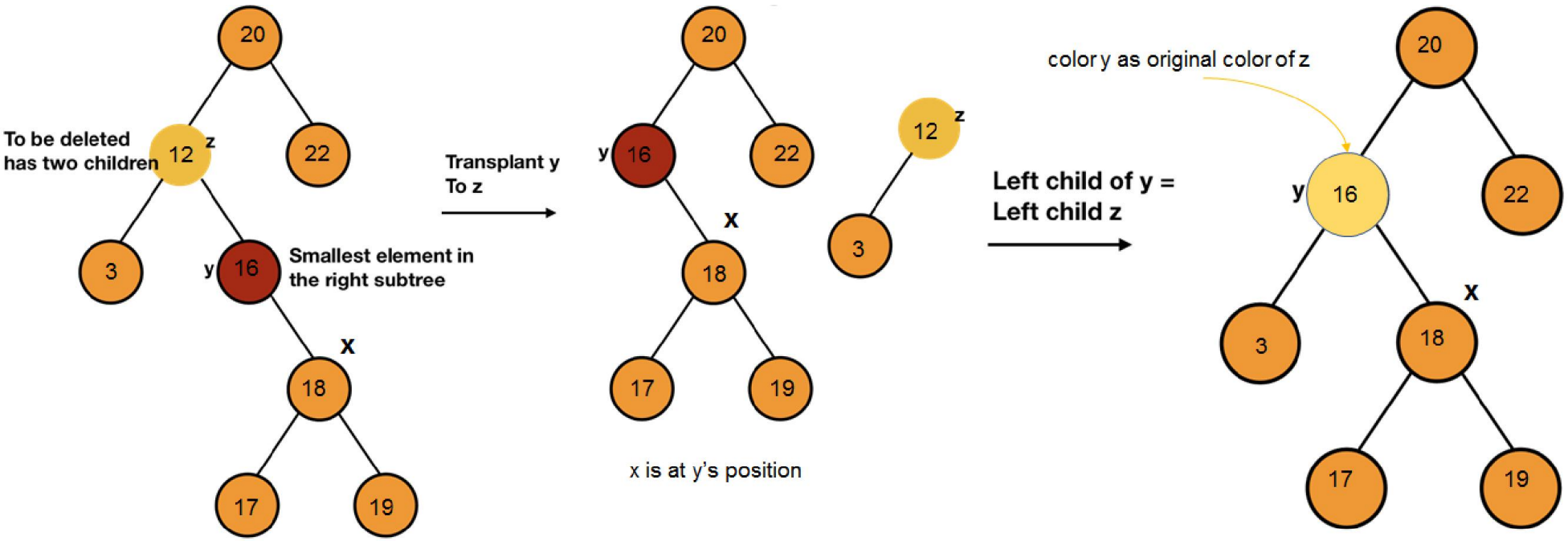
RB-TRANSPLANT

```
RB-TRANSPLANT(T, u, v)
  if u.parent == T.NIL // u is root
      T.root = v
  elseif u == u.parent.left //u is left child
      u.parent.left = v
  else // u is right child
      u.parent.right = v
  v.parent = u.parent
```

BST Delete – Review (Case 1 & 2)



BST Delete – Review (Case 3)



Delete Overview

- In the first two cases (1&2) when z has less than two children, x is taking the position of y/z.
- In the next case (3), since y is the minimum of the right subtree of z; we replace the node z with y and recolor it to the original color of z. Also the node x is again taking y's position.
- Any violations of the **red-black** properties, can only happen when x is taking the position of y. So, we will store the original color of y along the process and use it to see if any property is violated during the process or not.

Properties of Red-Black Trees – Review

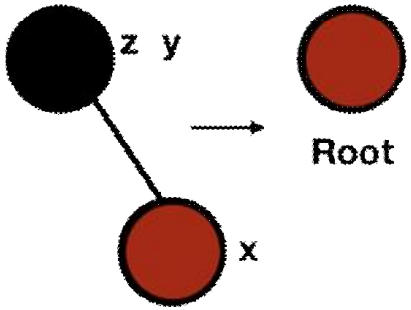
1. Every node is colored either **red** or **black**.
2. Root of the tree is **black**.
3. All leaves are **black**.
4. Both children of a red node are black i.e., there can't be consecutive **red** nodes.
5. All the simple paths from a node to descendant leaves contain the same number of **black** nodes.

Violations

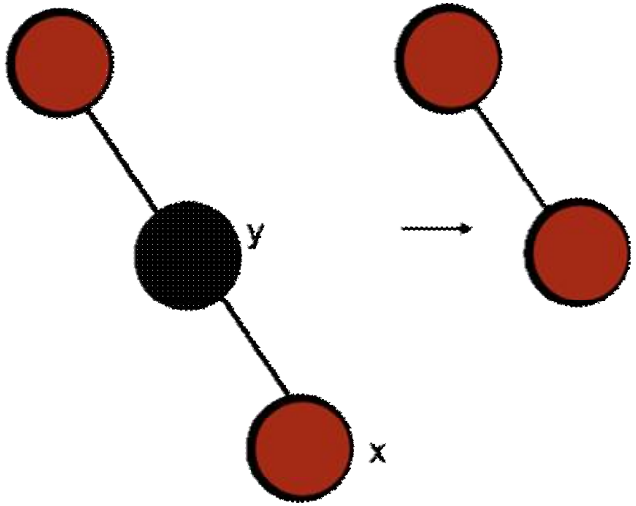
When x taking the position of y:

1. Property 1 can not be violated.
2. Property 2 can be violated if y is root and x taking its position is **red**. In this case, the original color of y will be **black**.
3. Property 3 is not going to be violated.
4. Property 4 can be violated only if the parent and $child(x)$ of y are **red**. In this case also, the original color of y will be **black** and after removing it, there will be two consecutive **reds**.
5. Property 5 can also be violated only if y is **black** because removing it will affect the black height of the nodes.

Violations (cont.)



Violation of prop. 2



Violation of prop. 4

Violation of prop. 5,
black height affected

Violations (cont.)

Note that any violation is going to happen if the original color of y was **black** and we will use this as the condition to run the code for fixing the violation.

The above claim can also be justified as if y was **red**:

- removing it is not going to affect the black height
- if it was **red**, then it cannot be the root, so root is still **black**
- no **red** nodes are made consecutive

Let's transform the above picture of deletion in code and then write the code to fix the violation of properties.

RBT Deletion Code

- The deletion code is similar to that of the deletion of a normal binary search tree. You can take a look at that before moving forward for better understanding.
- As stated above, we are going to store the original color of y in our process. Initially, we will mark z as y and if we deal with the case where the node z has two children, we will change the node y (We will discuss this later).

```
RB-DELETE(T, z)
  y = z
  y_orignal_color = y.color
```

RBT Deletion Code (cont.)

- After this, we will handle the case when the node z has only one child similar to what we did in the delete procedure of a normal binary search tree.

```
RB-DELETE(T, z)
  ...
  if z.left == T.NIL //no children or only right
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
  else if z.right == T.NIL // only left child
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
```

RBT Deletion Code (cont.)

- If the node z has two children, then we will mark the smallest element in the z's right subtree as y.

```
RB-DELETE(T, z)
  ...
  if z.left == T.NIL //no children or only right
    ...
  else if z.right == T.NIL // only left child
    ..
  else // both children
    y = MINIMUM(z.right)
    y_orignal_color = y.color
    x = y.right
```

RBT Deletion Code (cont.)

- The minimum element of the right subtree of z can either be a direct child (right child) of z or it can be a left child of some other node.
- We have marked x as the right child of y (i.e. $x = y.right$), but it might also be possible that x is $T.NIL$.
  - In that case, the parent of x will be pointing to any arbitrary node and not y. But we need to know the exact parent of x to run the code for fixup. So, we will take caution of pointing the parent of x to y , i.e. $x.parent = y$.

RBT Deletion Code (cont.)

- If y is the direct child of z, we will make y as the parent of x right away. In the other case, it will be done further in the process.

```
RB-DELETE(T, z)
  ...
  if z.left == T.NIL //no children or only right
    ...
  else if z.right == T.NIL // only left child
    ..
  else // both children
    ...
    if y.parent == z // y is direct child of z
      x.parent = y
```

RBT Deletion Code (cont.)

- If y is not the direct child of z, we will first transplant the right subtree of y to y → $RB-TRANSPLANT(T, y, y.right)$.
- After this, we will change the right of y to right of z.
- After this we can transplant y to z in both cases (whether y is direct child or not).

```
RB-DELETE(T, z)
  ...
    ...
    if y.parent == z // y is direct child of z
      x.parent = y
    else
      RB-TRANSPLANT(T, y, y.right)
      y.right = z.right
      y.right.parent = y
    RB-TRANSPLANT(T, z, y)
```

RBT Deletion Code (cont.)

- Next, we will put the left of z to the left of y and color y as z.

```
RB-DELETE(T, z)
  ...
    ...
    if y.parent == z // y is direct child of z
      x.parent = y
    else
      ...
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.parent = y
    y.color = z.color
```

RBT Deletion Code (cont.)

- At last, we will call the function to fix the violation if the original color of y was **black** (as discussed above).

```
RB-DELETE(T, z)
  ...
  if y_orignal_color == black
    RB-DELETE-FIXUP(T, x)
```

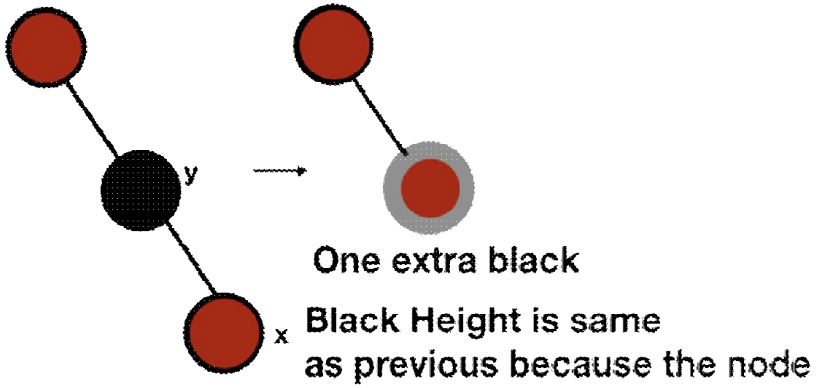
RB-DELETE (All Together)

```
RB-DELETE(T, z)
  y = z
  y_orignal_color = y.color
  if z.left == T.NIL //no children or only right
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
  else if z.right == T.NIL // only left child
      x = z.left
      RB-TRANSPLANT(T, z, z.left)
  else // two children
      y = MINIMUM(z.right)
      y_orignal_color = y.color
      x = y.right
      if y.parent == z // y is a direct child of z
          x.parent = y
      else             // y is an indirect child of z
          RB-TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.parent = y
      RB-TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.parent = y
      y.color = z.color
  if y_orignal_color == black
      RB-DELETE-FIXUP(T, x)
```

Fixing Violation of RB Tree in Deletion

- The violation of the property 5 (change in black height) is our main concern. This happens when y was **black**. We are going to deal it differently.
- We are going to say that the property 5 has not been violated and the node x which is now occupying y's original position has an "extra **black**" in it. In this way, the property of black height is not violated but the property 1 i.e., every node should be either **black** or **red** is violated because the node x is now either "**black**-**black**" or "**red**-**black**.

Fixing Violation of RB Tree in Deletion (cont.)



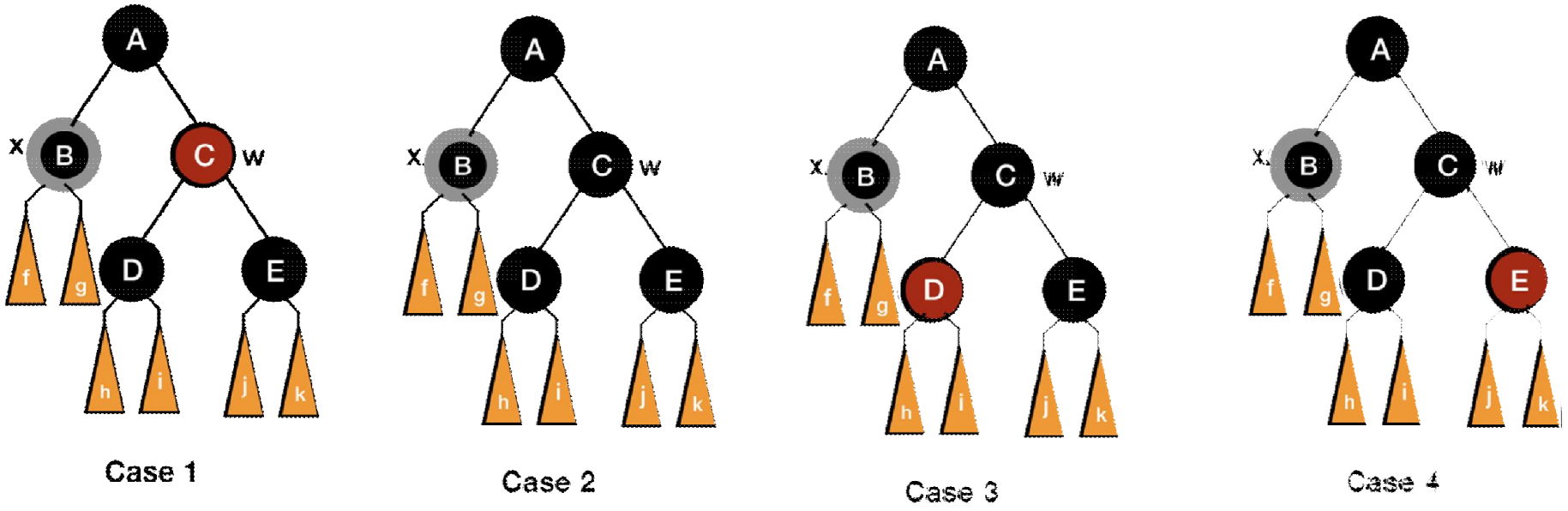
Fixing Violation of RB Tree in Deletion (cont.)

- That is being said, we can say that either property 1, 2 or 4 can be violated.
- If x is **red**-**black**, we can simply recolor it black and this will fix the violation of the property 1 without causing any other violation. This will also solve the violation of the property 4.
- If x is the root and is **black**-**back**, we can simply remove the one extra **black** and thus, fixing the violation of the property 1.
- With the above two mentioned cases, violations of properties 2 and 4 are completely solved and now we can focus only on solving the violation of property 1.

Fixing Violation of RB Tree in Deletion (cont.)

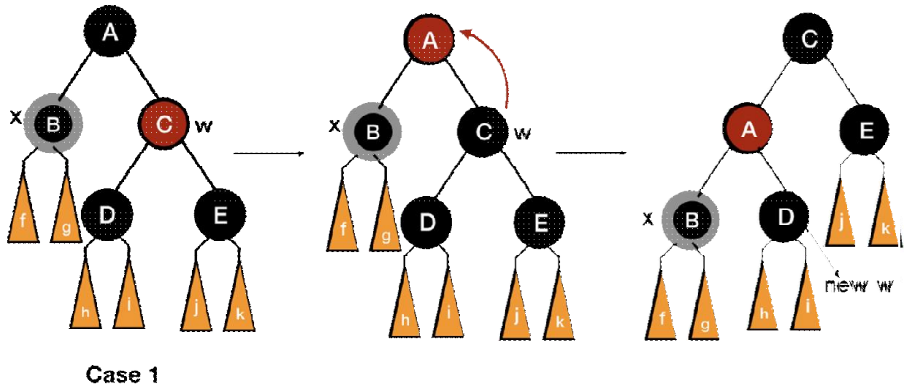
- There can be 4 cases (w is x's sibling):
  1. w is **red**.
  2. w is **black** and its both children are **black**.
  3. w is **black** and its right child is black and left child is **red**.
  4. w is **black** and its right child is **red**.

Fixing Violation of RB Tree in Deletion (cont.)



Fixing Violation of RB Tree in Deletion (cont.)

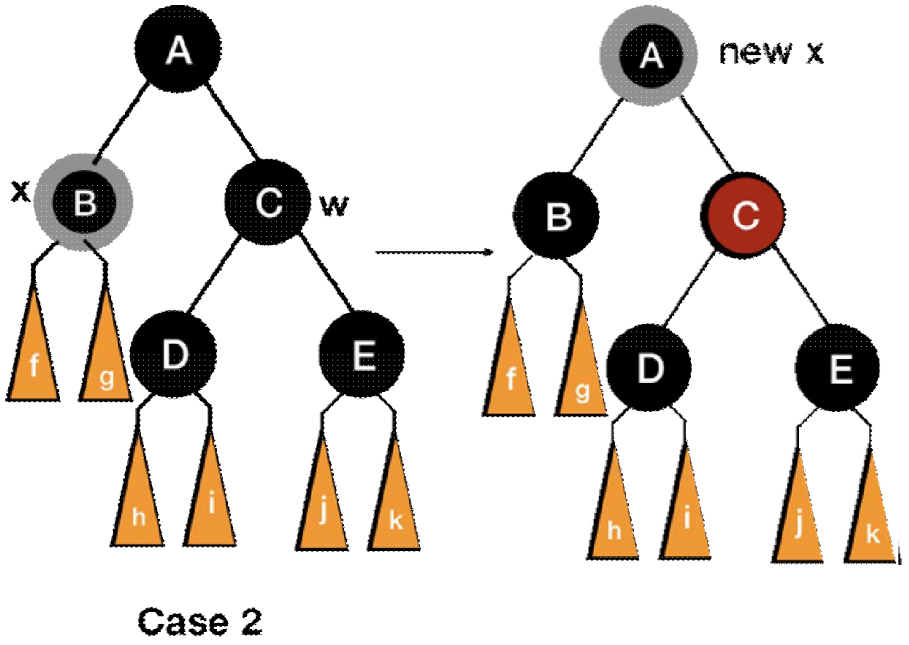
- For the first case, we can switch the colors of w and its parent and then left rotate the parent of x. In this way, we will enter either case 2, 3 or 4.



Case 1

Fixing Violation of RB Tree in Deletion (cont.)

- For the second case, we will take out one **black** from x and w both. This will leave x **black** and w **red**.
- For the compensation, we will add one extra **black** to the parent of x and mark it as the new x and repeat the entire process of fixing the violations for the new x.

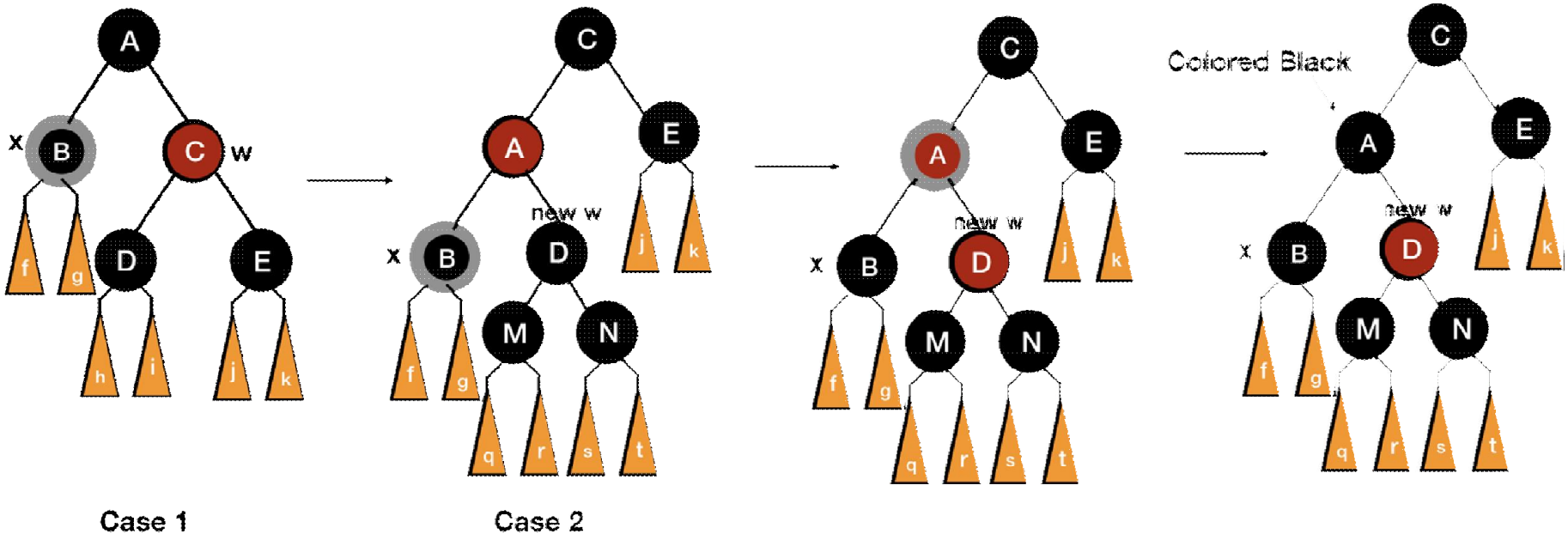


Case 2

Fixing Violation of RB Tree in Deletion (cont.)

- Also if we have entered case 2 from case 1, the parent must be **red** and now **red** and **black**, so it will be simply fixed by coloring it **black**.

Fixing Violation of RB Tree in Deletion (cont.)

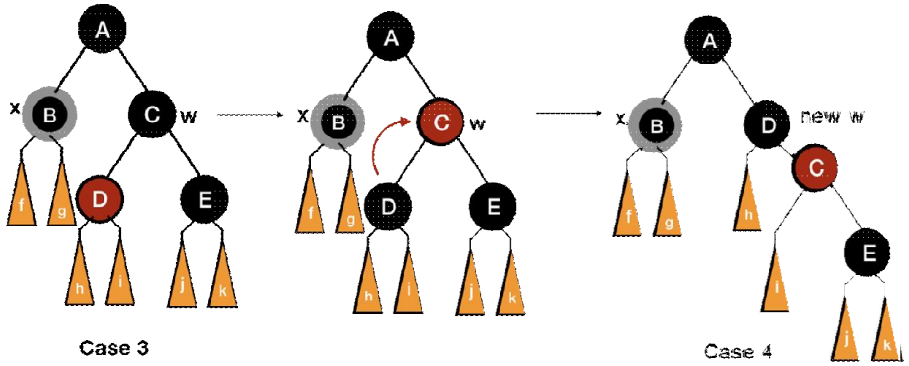


Case 1

Case 2

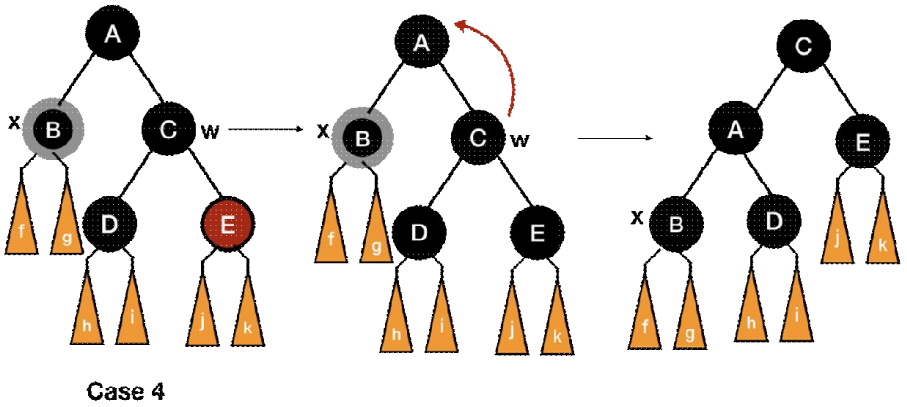
Fixing Violation of RB Tree in Deletion (cont.)

- We will transform case 3 to case 4 by switching the colors of w and its left child and then rotating right w.



Fixing Violation of RB Tree in Deletion (cont.)

- For case 4, we first colored w same as the parent of x and then colored the parent of x **black**. After this, we colored the right child of w **black** and then left rotated the parent of x. At last, we removed extra **black** from x without violating any properties.
- By now, you must be clear with the way of fixing properties. Let's write the code to do so.



RB-DELETE-FIXUP

```
RB-DELETE-FIXUP(T, x)
  while x!= T.root and x.color == black
      if x == x.parent.left
          w = x.parent.right
          if w.color == red //case 1
              w.color = black
              x.parent.color = red
              LEFT-ROTATE(T, x.parent)
              w = x.parent.right
          if w.left.color == black and w.right.color == black //case 2
              w.color = red
              x = x.parent
```

RB-DELETE-FIXUP (cont.)

```
        else //case 3 or 4
            if w.right.color == black //case 3
                w.left.color = black
                w.color = red
                RIGHT-ROTATE(T, w)
                w = x.parent.right
            //case 4
            w.color = x.parent.color
            x.parent.color = black
            w.right.color = black
            LEFT-ROTATE(T, x.parent)
            x = T.root
    else
        code will be symmetric
x.color = black
```

RB-DELETE-FIXUP (cont.)

- As we have already discussed, if x is the root or **red** and **black**, we will simply color it **black**. So, the loop can only be executed in those conditions, otherwise, we are doing $x.color = black$ at the last of the code.
- The above code is for x being the left child, the code when x is the right child will be symmetric.

RB-Delete Code in C

```
void rb_transplant(red_black_tree *t, tree_node *u, tree_node *v) {
    if(u->parent == t->NIL)
        t->root = v;
    else if(u == u->parent->left)
        u->parent->left = v;
    else
        u->parent->right = v;
    v->parent = u->parent;
}

tree_node* minimum(red_black_tree *t, tree_node *x) {
    while(x->left != t->NIL)
        x = x->left;
    return x;
}
```

RB-Delete Code in C (cont.)

```
void rb_delete(red_black_tree *t, tree_node *z) {
    tree_node *y = z;
    tree_node *x;
    enum COLOR y_orignal_color = y->color;
    if(z->left == t->NIL) {
        x = z->right;
        rb_transplant(t, z, z->right);
    } else if(z->right == t->NIL) {
        x = z->left;
        rb_transplant(t, z, z->left);
    } else {
        y = minimum(t, z->right);
        y_orignal_color = y->color;
        x = y->right;
        if(y->parent == z) {
            x->parent = z;
        } else {
            rb_transplant(t, y, y->right);
            y->right = z->right;
            y->right->parent = y;
        }
        rb_transplant(t, z, y);
        y->left = z->left;
        y->left->parent = y;
        y->color = z->color;
    }
    if(y_orignal_color == Black) rb_delete_fixup(t, x);
}
```

RB-Delete Code in C (cont.)

```
void rb_delete_fixup(red_black_tree *t, tree_node *x) {
    while(x != t->root && x->color == Black) {
        if(x == x->parent->left) {
            tree_node *w = x->parent->right;
            if(w->color == Red) {
                w->color = Black;
                x->parent->color = Red;
                left_rotate(t, x->parent);
                w = x->parent->right;
            }
            if(w->left->color == Black && w->right->color == Black) {
                w->color = Red;
                x = x->parent;
            } else {
                if(w->right->color == Black) {
                    w->left->color = Black;
                    w->color = Red;
                    right_rotate(t, w);
                    w = x->parent->right;
                }
                w->color = x->parent->color;
                x->parent->color = Black;
                w->right->color = Black;
                left_rotate(t, x->parent);
                x = t->root;
            }
```

RB-Delete Code in C (cont.)

```
        } else {
            tree_node *w = x->parent->left;
            if(w->color == Red) {
                w->color = Black;
                x->parent->color = Red;
                right_rotate(t, x->parent);
                w = x->parent->left;
            }
            if(w->right->color == Black && w->left->color == Black) {
                w->color = Red;
                x = x->parent;
            } else {
                if(w->left->color == Black) {
                    w->right->color = Black;
                    w->color = Red;
                    left_rotate(t, w);
                    w = x->parent->left;
                }
                w->color = x->parent->color;
                x->parent->color = Black;
                w->left->color = Black;
                right_rotate(t, x->parent);
                x = t->root;
            }
        }
    }
    x->color = Black;
}
```