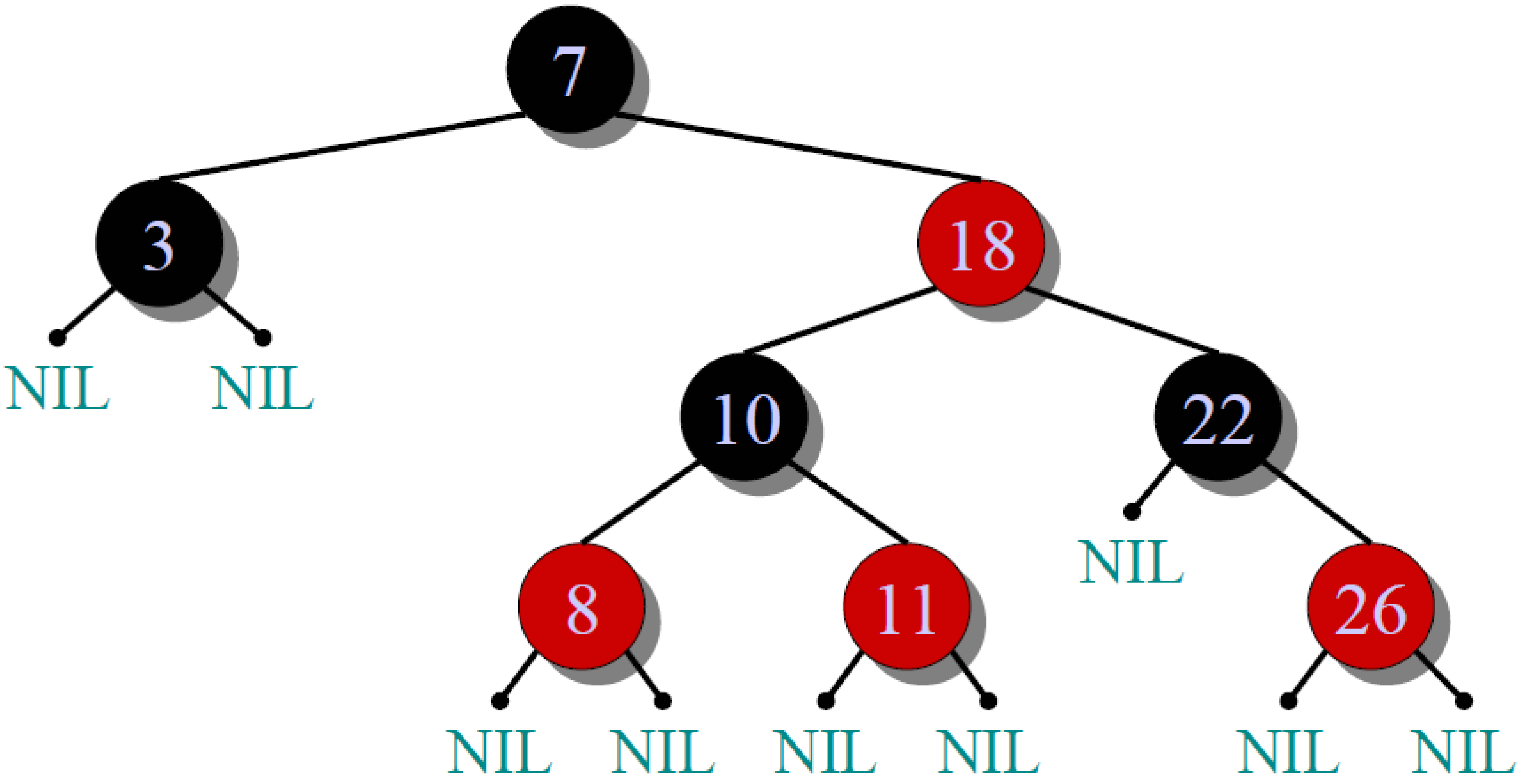
Red-Black Tree

SWE2016-44

Example of a Red-Black Tree



Red-Black Tree

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

1) **Every node has a color either red or black**

Red-Black Tree

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

1) **Every node has a color either red or black**
2) **Root of tree is always black**

Red-Black Tree

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

1) **Every node has a color either red or black**
2) **Root of tree is always black**
3) **There are no two adjacent red nodes (A red node cannot have a red parent or red child)**

Red-Black Tree

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules:

1) **Every node has a color either red or black**
2) **Root of tree is always black**
3) **There are no two adjacent red nodes (A red node cannot have a red parent or red child)**
4) **Every path from root to a NULL node has same number of black nodes**

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST.

→ O(n) for a skewed Binary tree.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST.

→ O(n) for a skewed Binary tree.

→ Since a Red-Black Tree ensures almost balanced, the height of the tree remains O(Log n) after every insertion and deletion.

→ Red-Black Tree is not always a balanced tree.

How does a Red-Black Tree ensure balance?

We can try any combination of colors and see all of them violate Red-Black tree property.

```
A chain of 3 nodes is nodes is not possible in Red-Black Trees.
Following are NOT Red-Black Trees
          30              30               30
         / \             /  \             /  \
       20   NIL        20    NIL        20    NIL
      / \             / \              / \
    10   NIL        10   NIL         10   NIL
Violates          Violates          Violates
Property 4.      Property 4        Property 3
```

How does a Red-Black Tree ensure balance?

We can try any combination of colors and see all of them violate Red-Black tree property.

```
Following are different possible Red-Black Trees with above 3 keys
         20                             20
       /    \                         /    \
     10      30                     10      30
    /  \    /  \                   /  \    /  \
 NIL   NIL NIL NIL              NIL   NIL NIL NIL
```

Black Height of a Red-Black Tree

Black height is number of black nodes on a path from root to a leaf. Leaf nodes are also counted black nodes.

Black Height of a Red-Black Tree

Black height is number of black nodes on a path from root to a leaf. Leaf nodes are also counted black nodes.

From properties 3 (no two adjacent red nodes) and 4 (same number of black nodes),

Black-height >= h/2.

Height of a Red-Black Tree

Every Red Black Tree with n nodes has $Height \le 2Log_2(n+1)$.

Height of a Red-Black Tree

Every Red Black Tree with n nodes has *Height* <= *$2Log_2(n+1)$*.

Proof)

1) **For a general Binary Tree, let k be the minimum number of nodes on all root to NULL paths, then n >= 2k – 1 (Ex. If k is 3, then n is at least 7). That is, k <= Log_2 (n+1).**

Height of a Red-Black Tree

Every Red Black Tree with n nodes has *Height* <= *2Log₂(n+1)*.

Proof)

1) **For a general Binary Tree, let k be the minimum number of nodes on all root to NULL paths, then n >= 2k – 1 (Ex. If k is 3, then n is at least 7). That is, k <= Log_2 (n+1).**
2) **From 1) and property 4, there is a root to leaf path with at most Log_2 (n+1) black nodes: h' <= Log_2 (n+1)**

Height of a Red-Black Tree

Every Red Black Tree with n nodes has *Height* <= *2Log$_2$(n+1)*.

Proof)

1) **For a general Binary Tree, let k be the minimum number of nodes on all root to NULL paths, then n >= 2k – 1 (Ex. If k is 3, then n is at least 7). That is, k <= Log$_2$ (n+1).**
2) **From 1) and property 4, there is a root to leaf path with at most Log$_2$ (n+1) black nodes: h' <= Log$_2$ (n+1)**
3) **Black-height is at least h/2: h' >= h/2**

Height of a Red-Black Tree

Every Red Black Tree with n nodes has *Height* <= $2Log_2(n+1)$.

Proof)

1) **For a general Binary Tree, let k be the minimum number of nodes on all root to NULL paths, then n >= $2^k – 1$ (Ex. If k is 3, then n is at least 7). That is, k <= $Log_2 (n+1)$.**
2) **From 1) and property 4, there is a root to leaf path with at most $Log_2 (n+1)$ black nodes: h' <= $Log_2 (n+1)$**
3) **Black-height is at least h/2: h' >= h/2**
4) **From 2) and 3), h <= $2Log_2 (n+1)$**

Insertion

The goal of the insert operation is to insert key K into tree T, maintaining T's red-black tree properties

Insertion

If T is a non-empty tree, then we do the following:

1. **Use the BST insert algorithm to add K to the tree**

2. **Color the node containing K red**

3. **Restore red-black tree properties (if necessary)**

Insertion

To restore the violated property, we use:

1. Recoloring

2. Rotation (Left, Right, Double)

We try recoloring first, if recoloring doesn’t work, then we go for rotation.

Insertion

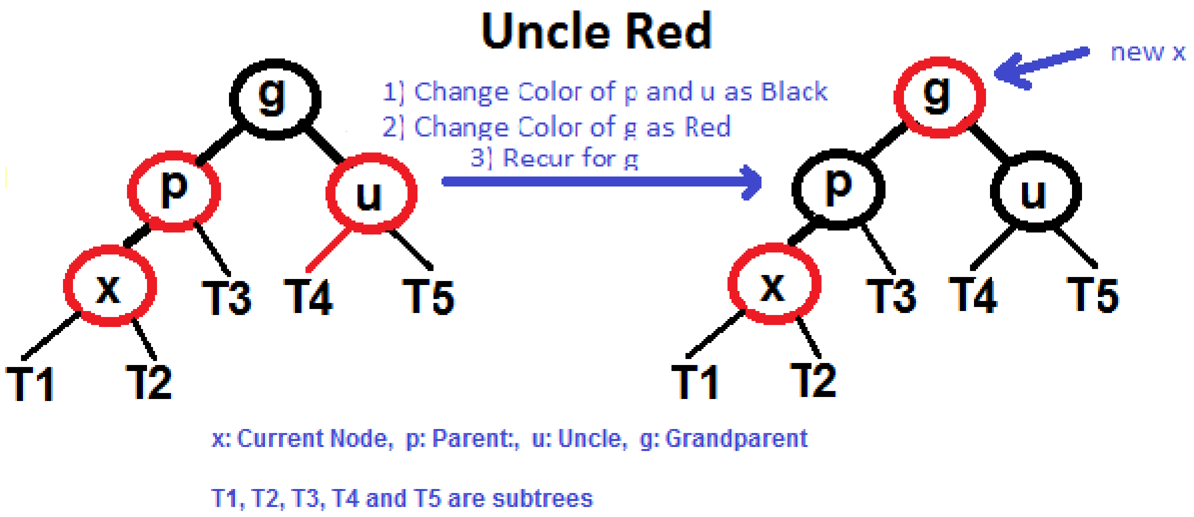
1) Perform standard BST insertion and make the color of newly inserted nodes as RED.

2) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).

3) Do following if color of x’s parent is not BLACK and x is not root.

Insertion

a. If x's uncle is RED (Grand parent must have been black from property 4)
   I. Change color of parent and uncle as BLACK.
   II. color of grand parent as RED.
   III. Change x = x's grandparent, repeat steps 2 and 3 for new x.



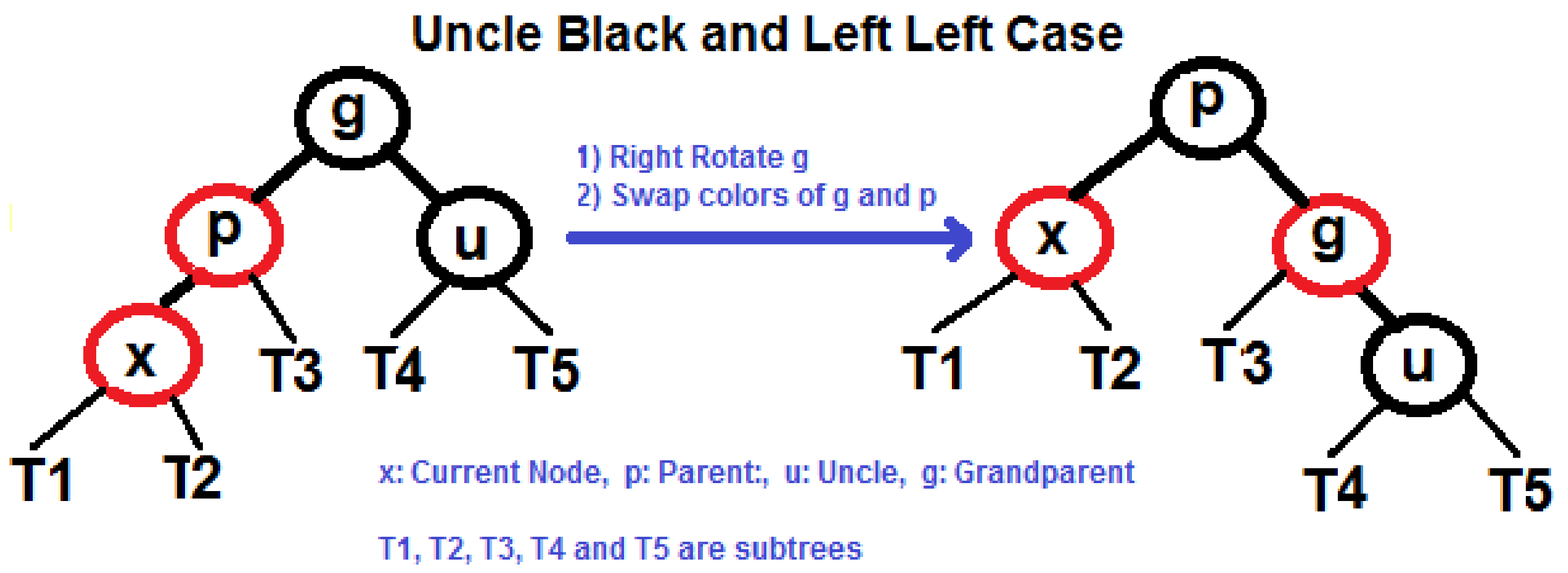
Insertion

b. If x’s uncle is BLACK, then there can be four configurations for x, x’s parent (p) and x’s grandparent (g)
   I. Left Left Case (p is left child of g and x is left child of p)
   II. Left Right Case (p is left child of g and x is right child of p)
   III. Right Right Case (Mirror of case i)
   IV. Right Left Case (Mirror of case ii)

Insertion

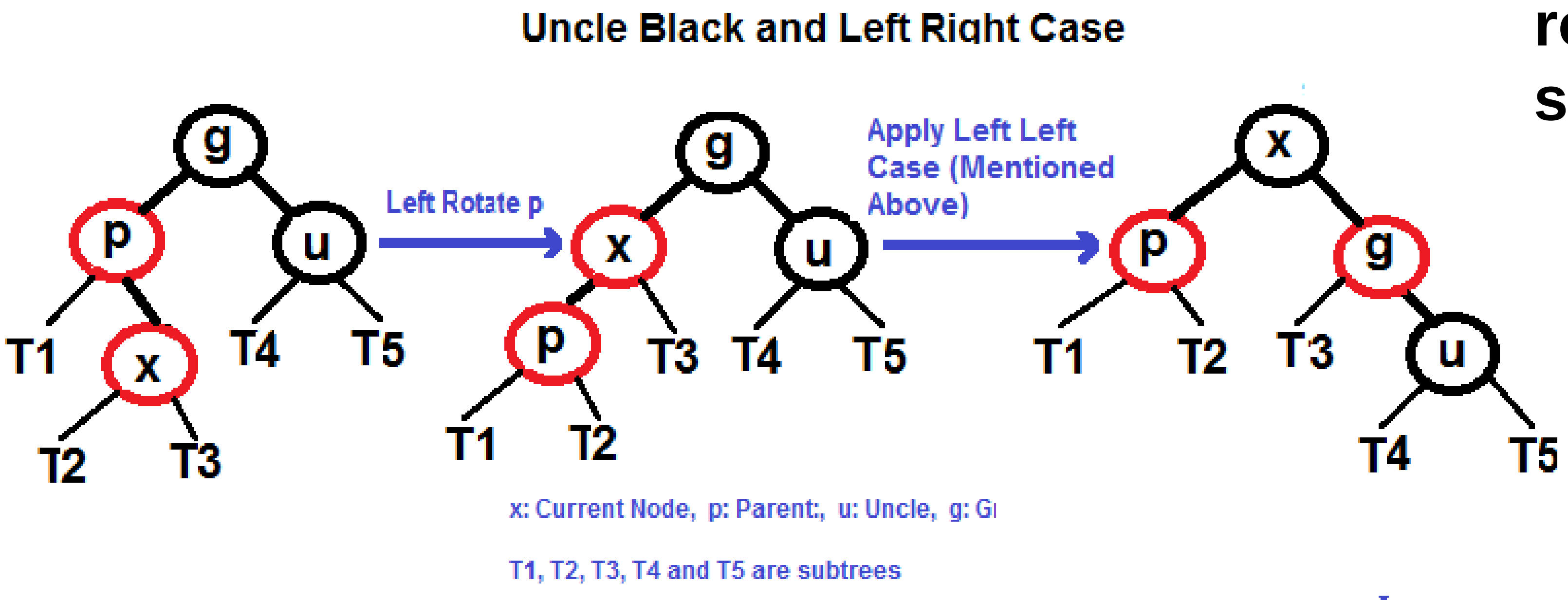
I. Left Left Case (p is left child of g and x is left child of p)

rotateRight(root, g)
swap(p→color, g→color)



Insertion

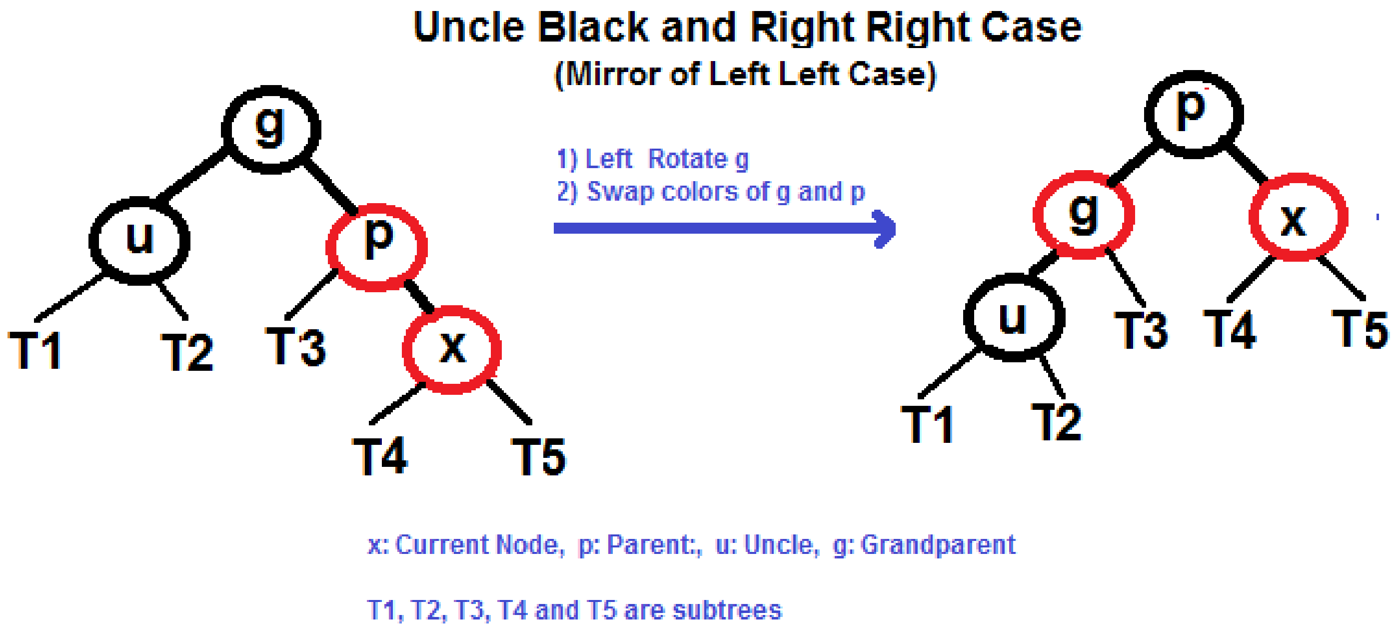
II. Left Right Case (p is left child of g and x is right child of p)



rotateLeft(root, p)
rotateRight(root, g)
swap(x→color, g→color)

Insertion

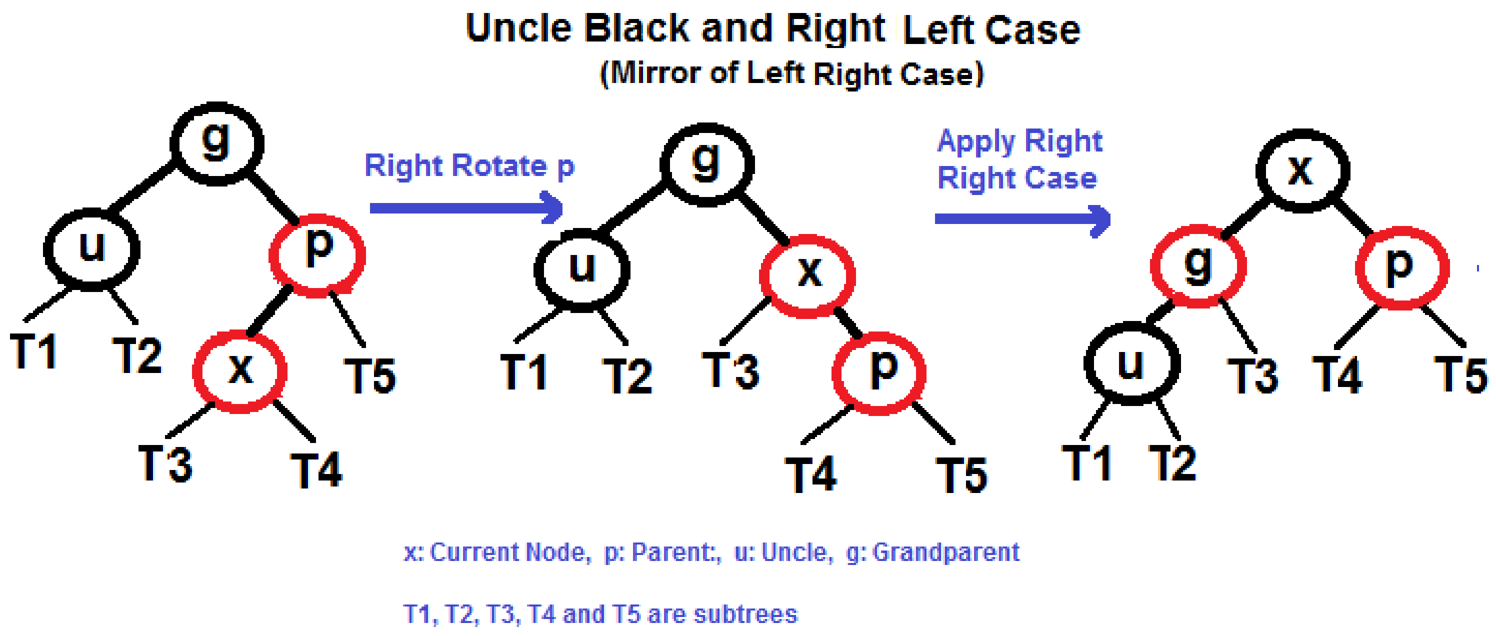
III. Right Right Case (Mirror of case i)



rotateLeft(root, g)
swap(p→color, g→color)

Insertion

IV. Right Left Case (Mirror of case ii)



rotateRight(root, p)
rotateLeft(root, g)
swap(x→color, g→color)

Insertion Analysis

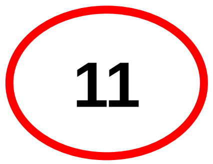
- **Go up the tree performing Case 3-a), which only recolors nodes.**
- **If Case 3-b) occurs, perform 1 or 2 rotations, and terminate.**

→ Running time: O(log n) with O(1) rotations.

Example of Insertion

11
1
14
2
7
15

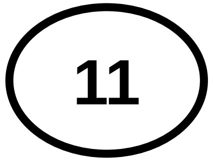
Example of Insertion



Insert 11

11
1
14
2
7
15

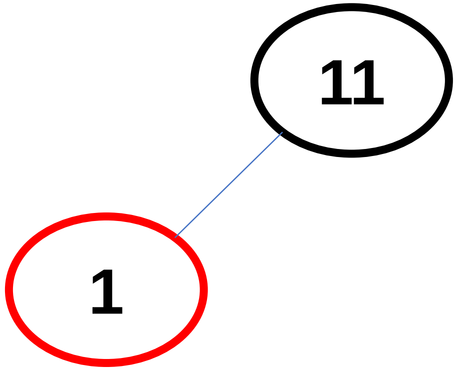
Example of Insertion



Insert 11

11
1
14
2
7
15

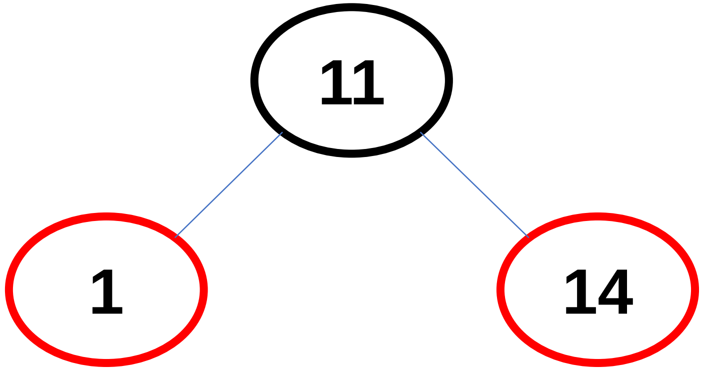
Example of Insertion



Insert 1

11
1
14
2
7
15

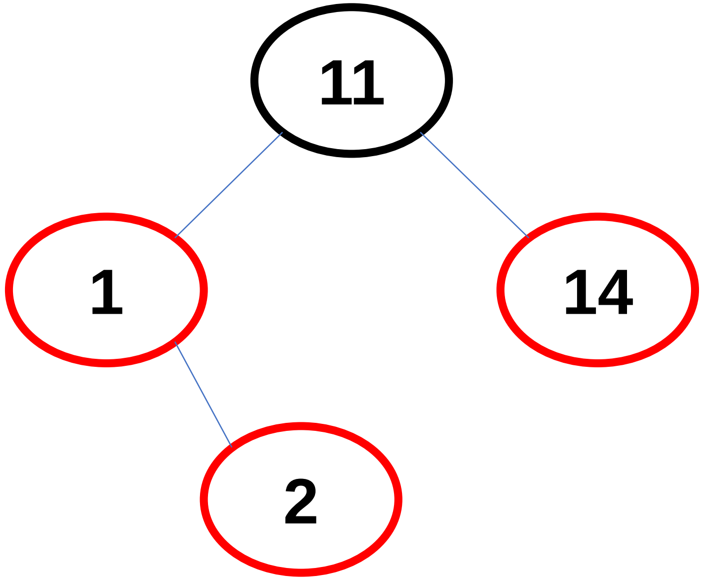
Example of Insertion



Insert 14

11
1
14
2
7
15

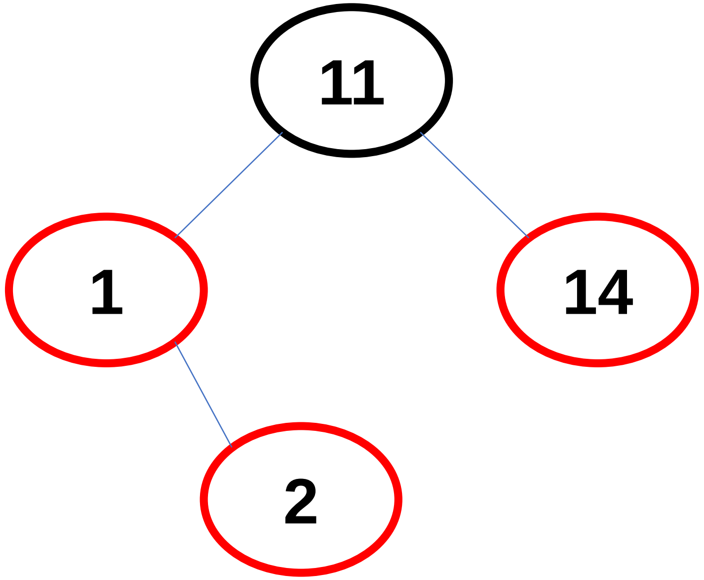
Example of Insertion



Insert 2

11
1
14
2
7
15

Example of Insertion

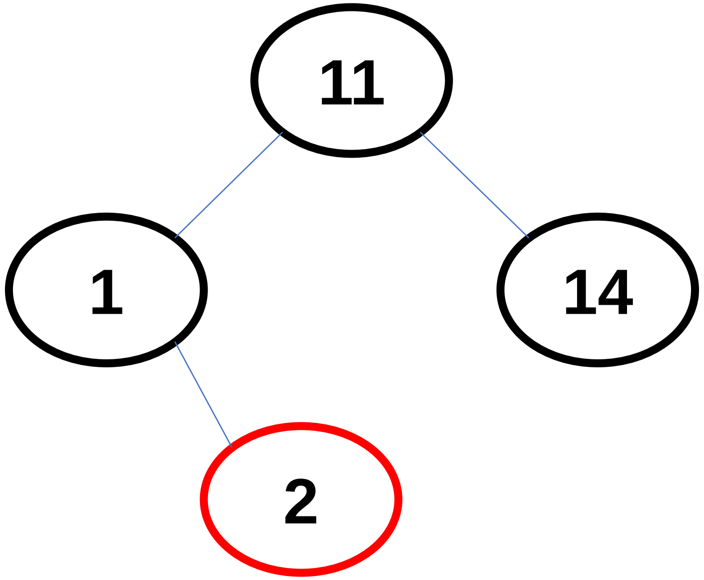


Insert 2

11
1
14
2
7
15

If X’s uncle is RED and X’s parent is not BLACK, change color of parent and uncle as BLACK

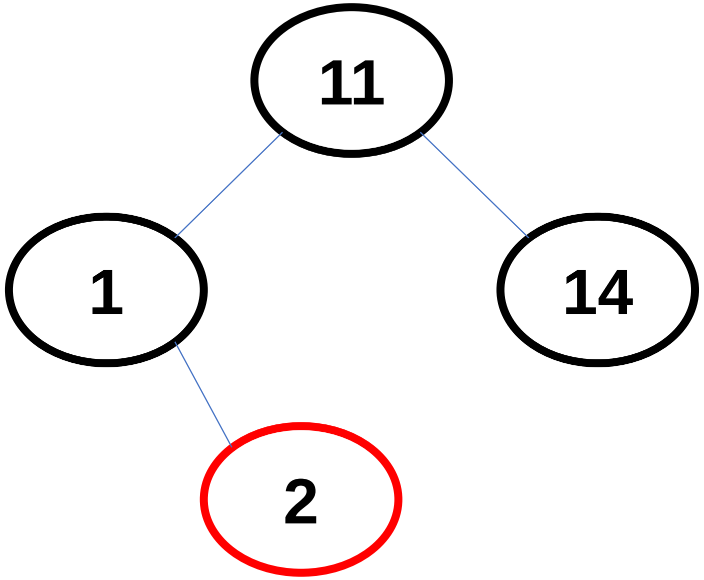
Example of Insertion



Insert 2

11
1
14
2
7
15

Example of Insertion

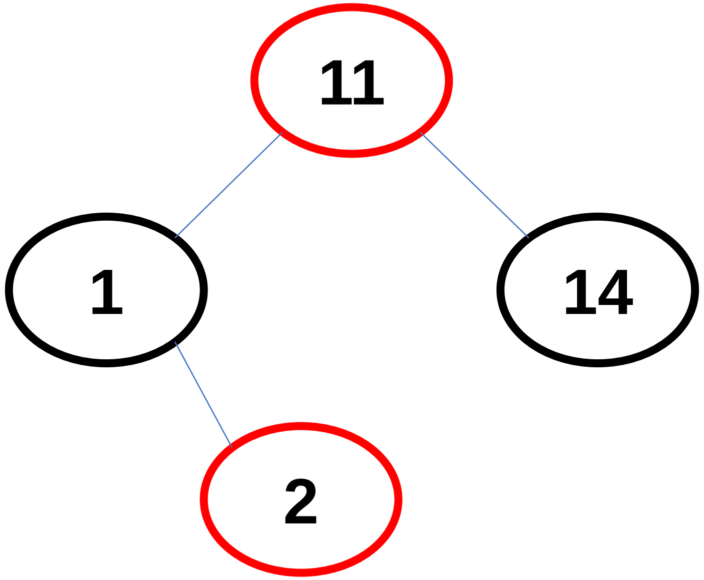


Insert 2

11
1
14
2
7
15

Color of Grand parent as RED

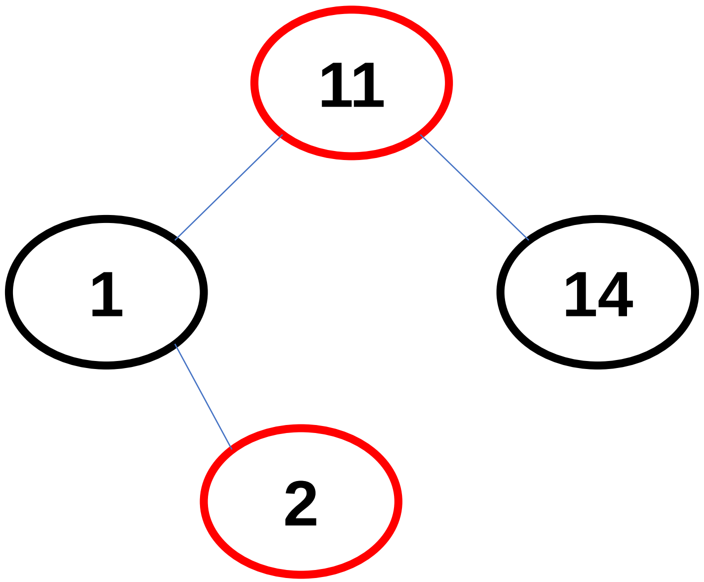
Example of Insertion



Insert 2

11
1
14
2
7
15

Example of Insertion

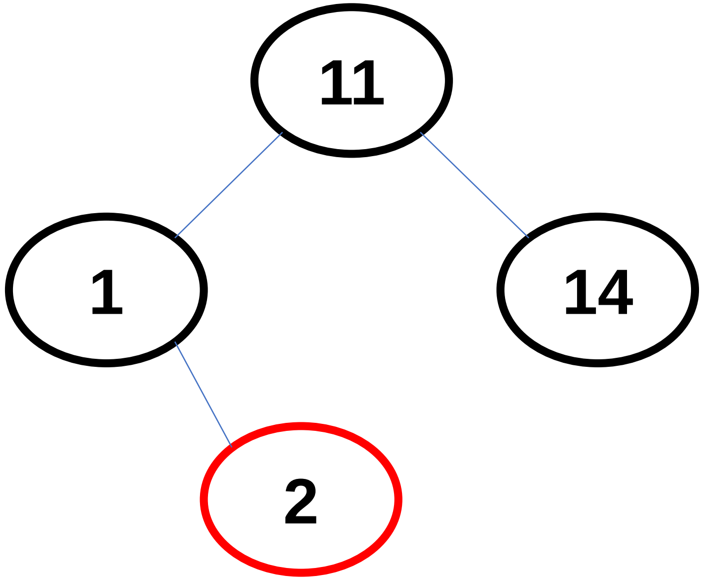


Insert 2

11
1
14
2
7
15

As 11 is a root node, change its color to black

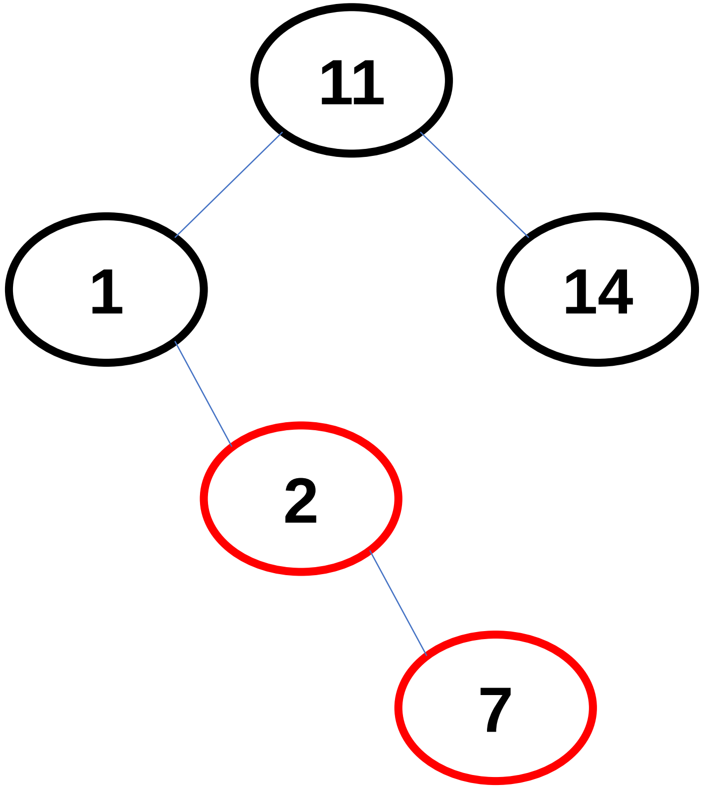
Example of Insertion



Insert 2

11
1
14
2
7
15

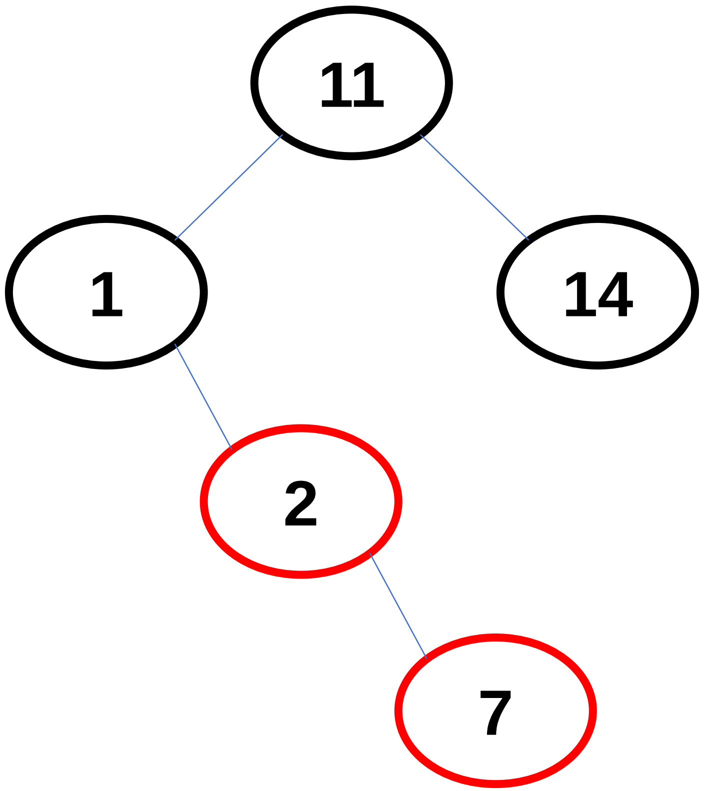
Example of Insertion



Insert 7

11
1
14
2
7
15

Example of Insertion

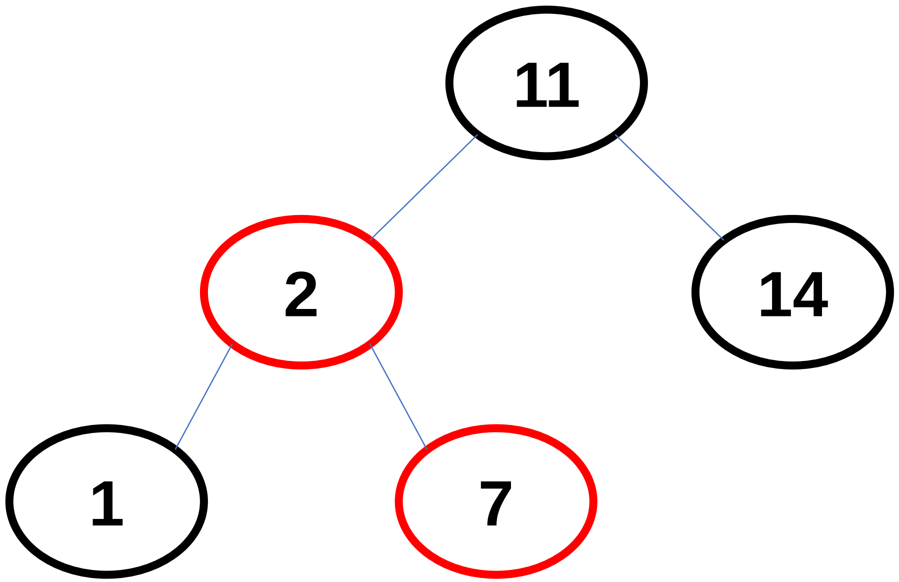


Left rotate(2) and recolor nodes

Insert 7

11
1
14
2
7
15

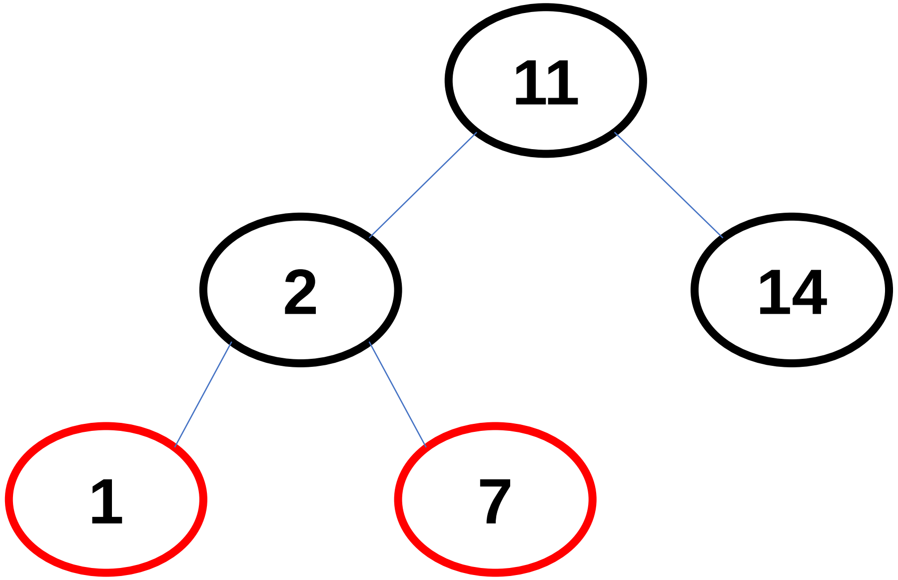
Example of Insertion



Insert 7

11
1
14
2
7
15

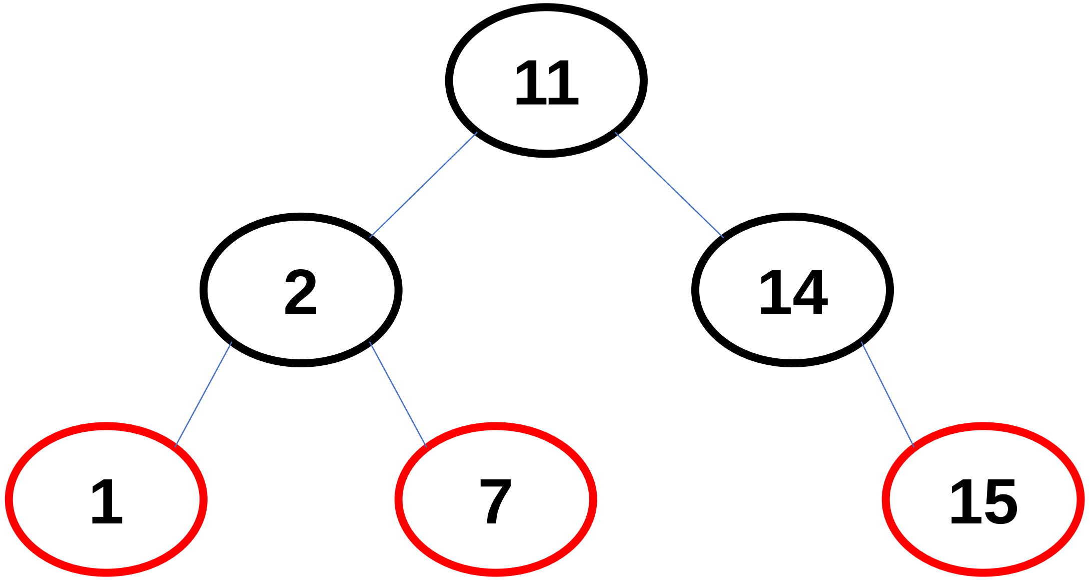
Example of Insertion



Insert 7

11
1
14
2
7
15

Example of Insertion



Insert 15

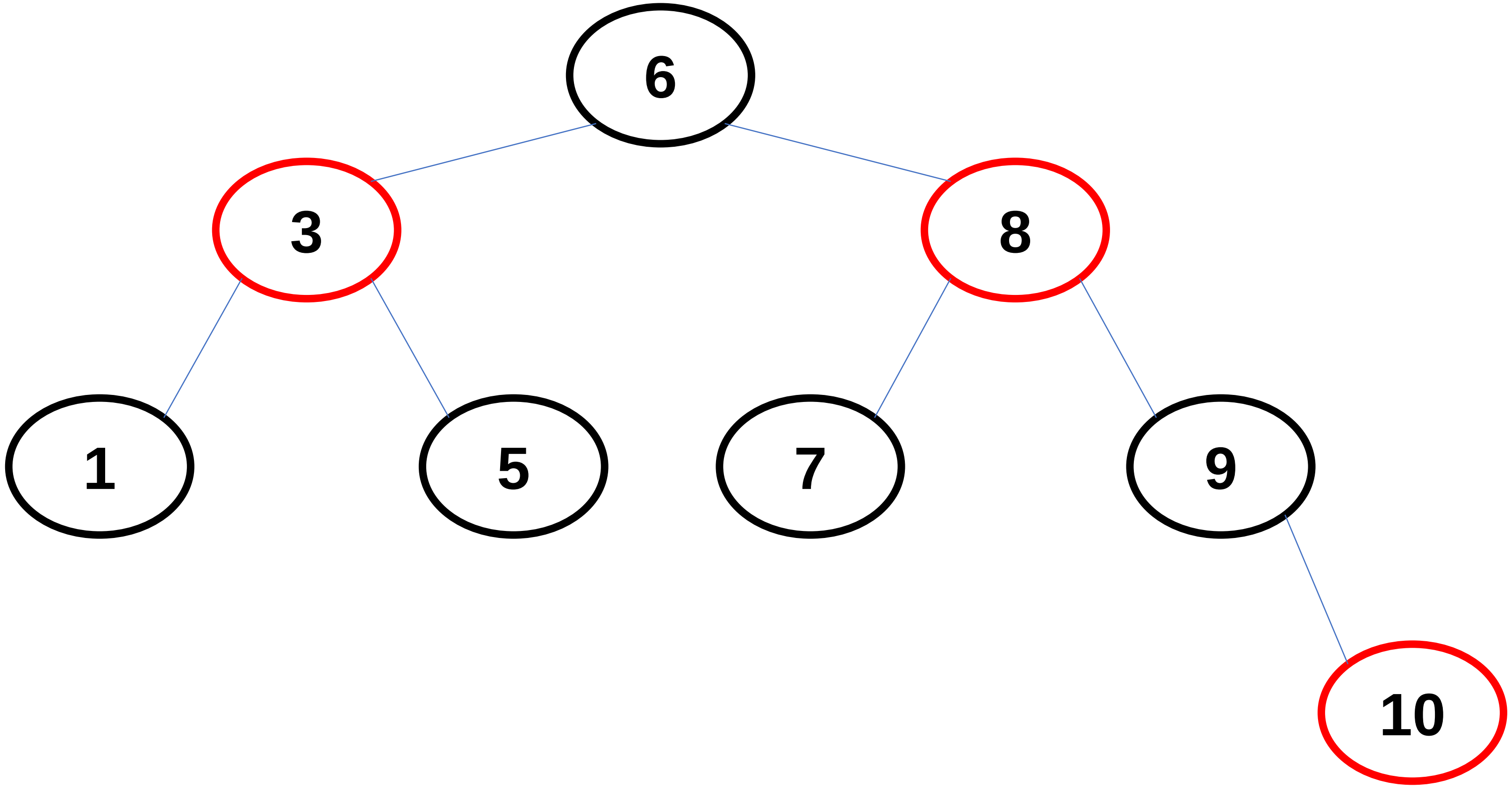
11
1
14
2
7
15

Exercise

Insert 3, 1, 5, 7, 6, 8, 9, 10

Exercise

Insert 3, 1, 5, 7, 6, 8, 9, 10



Deletion

Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.

In delete operation, we check color of sibling to decide the appropriate case.

Deletion

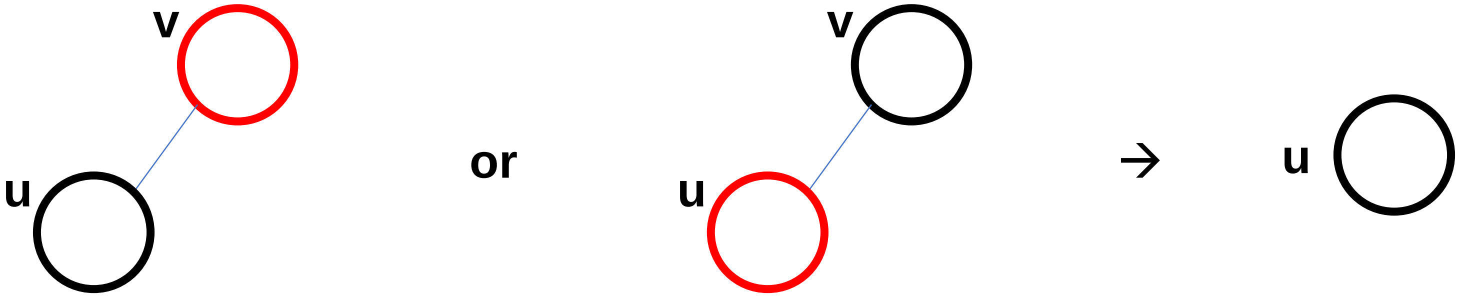
- Delete as we delete from BST.
  - →End up deleting the node which is either a leaf or has one child

- We delete an internal node from a BST simply by replacing it by its inorder successor and then we recursively call delete operation on inorder susccessor node.

- v: deleted node, u: the child that replaces v

Deletion

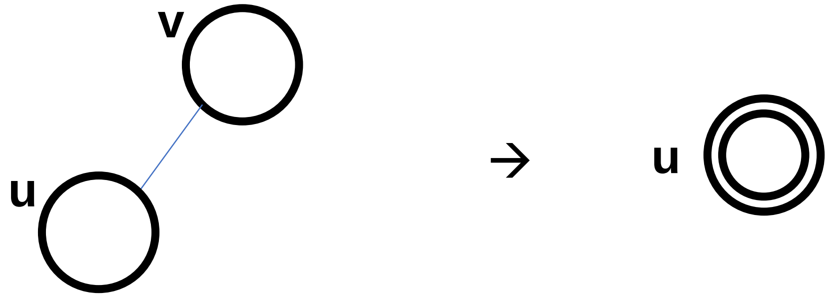
Either u or v is RED



→ Replace v by u, u will be a black node

Deletion

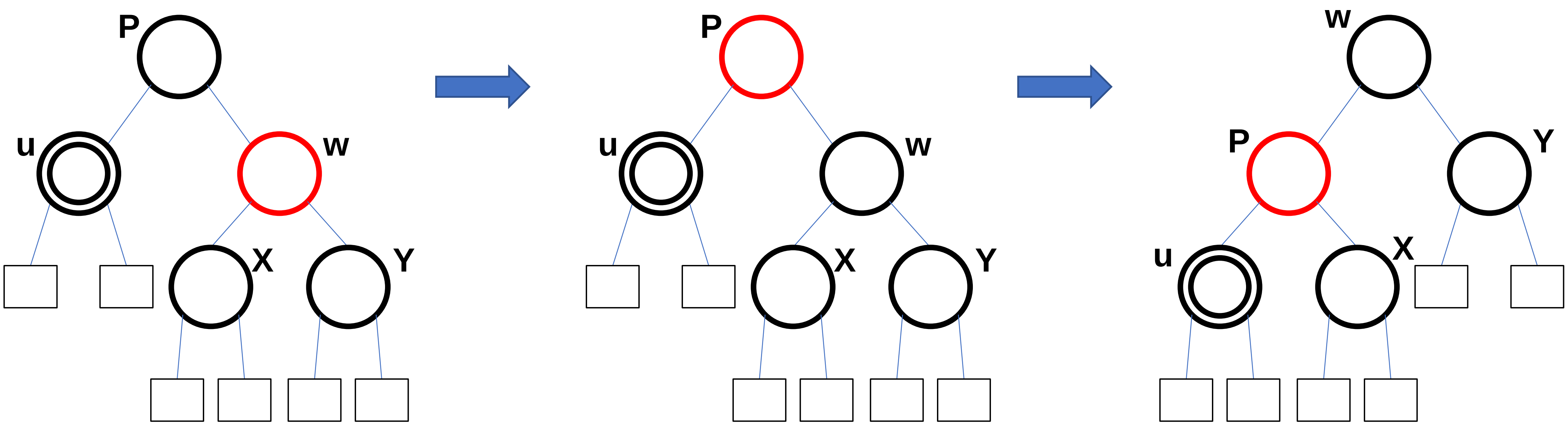
Both u and v are BLACK



→ If node u is root, make it single black

Deletion (Case 1)

Node u's sibling w is RED

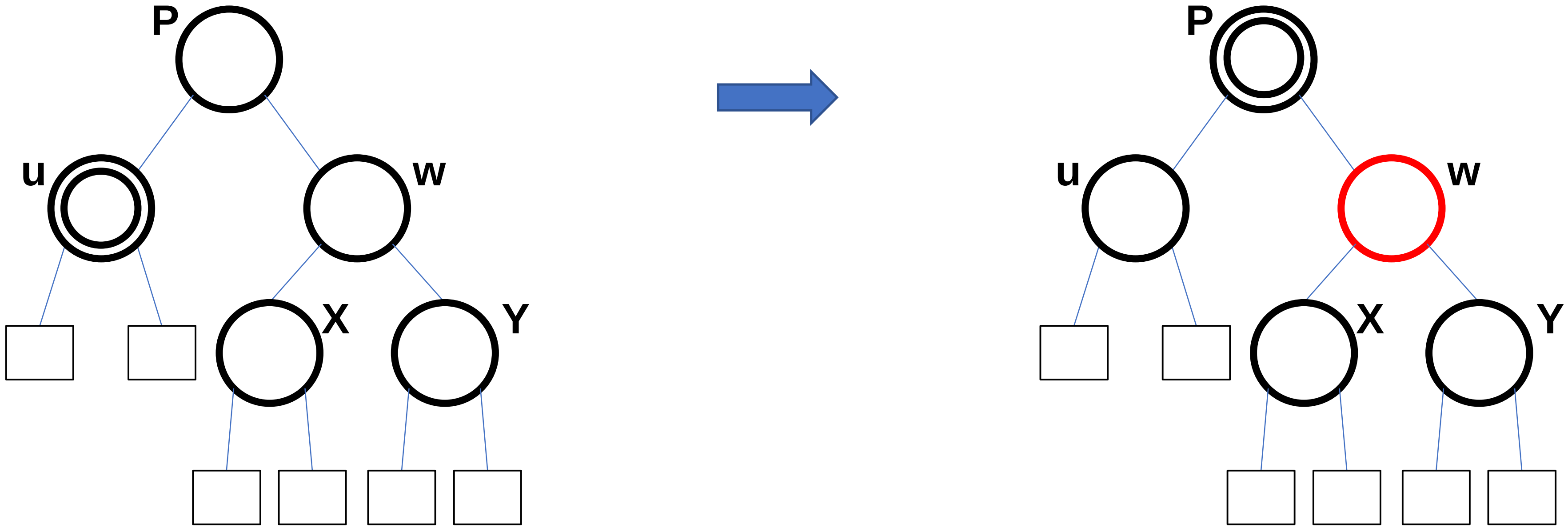


Deletion (Case 2)

Node u's sibling w is BLACK, and both of w's children are BLACK

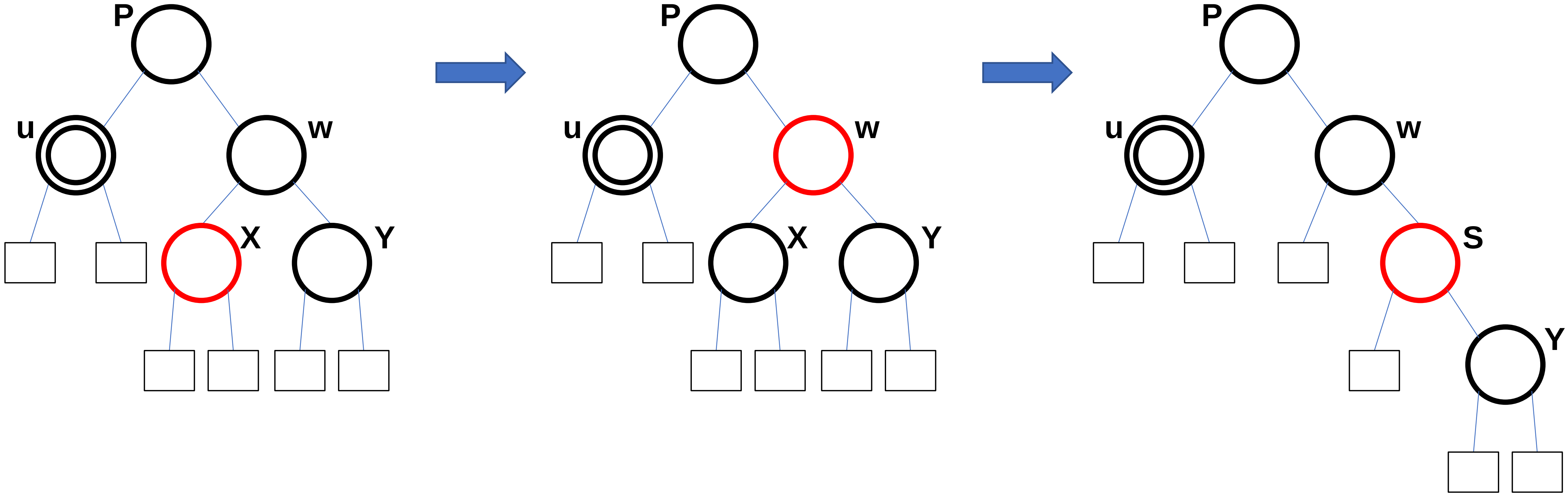
Deletion (Case 2)

Node u's sibling w is BLACK, and both of w's children are BLACK



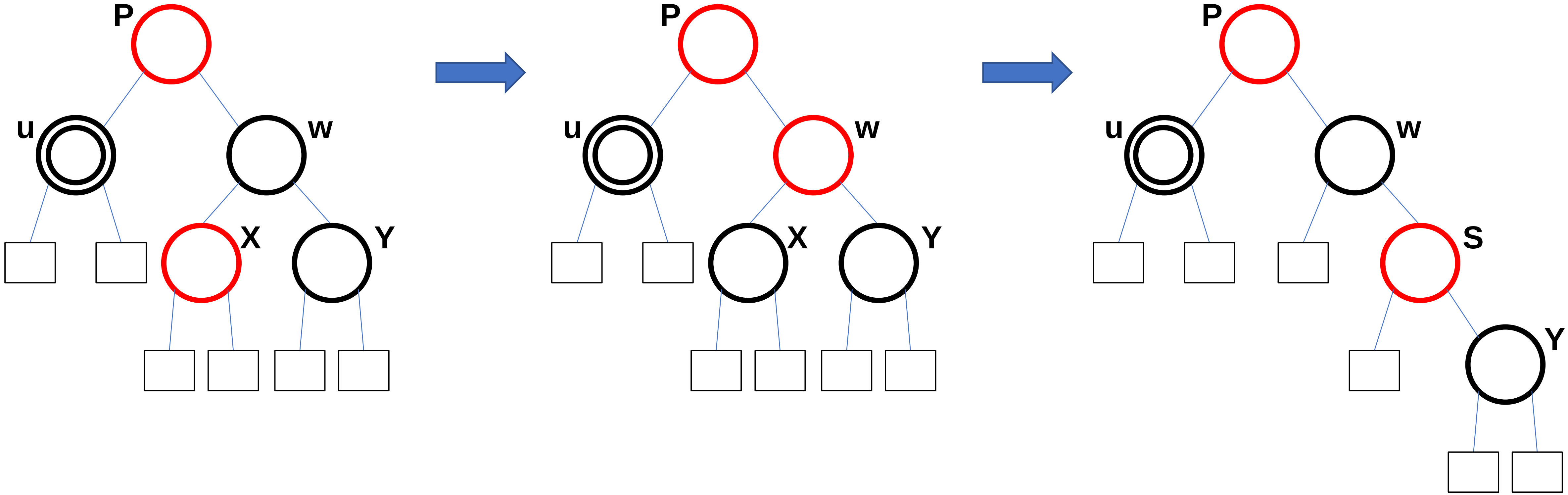
Deletion (Case 3)

Node u's sibling w is BLACK, and w's left child is RED and w's right child is BLACK



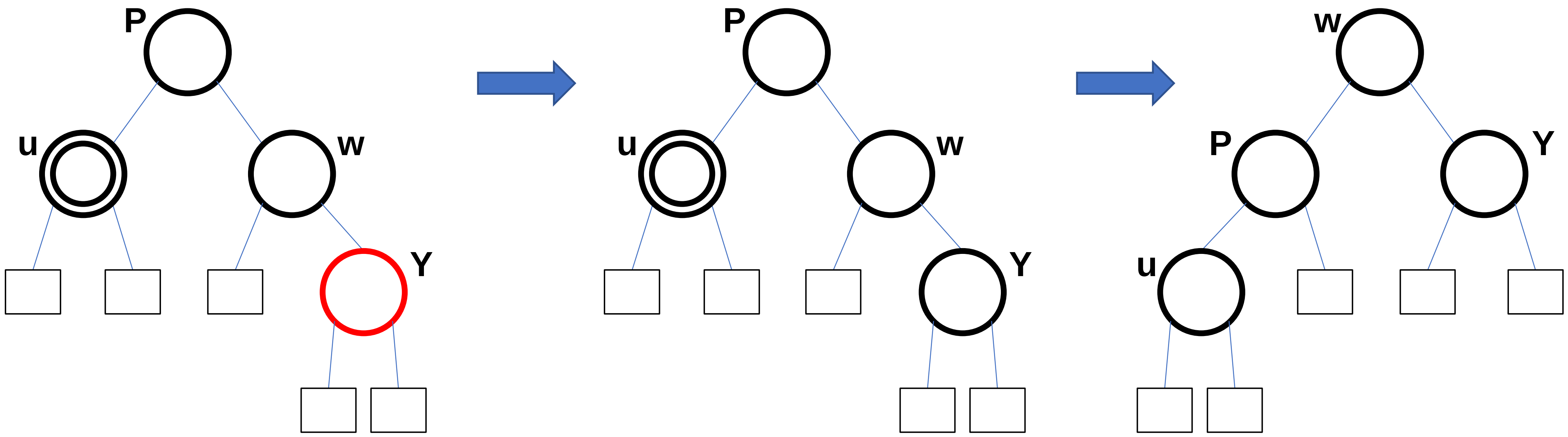
Deletion (Case 3)

Node u's sibling w is BLACK, and w's left child is RED and w's right child is BLACK



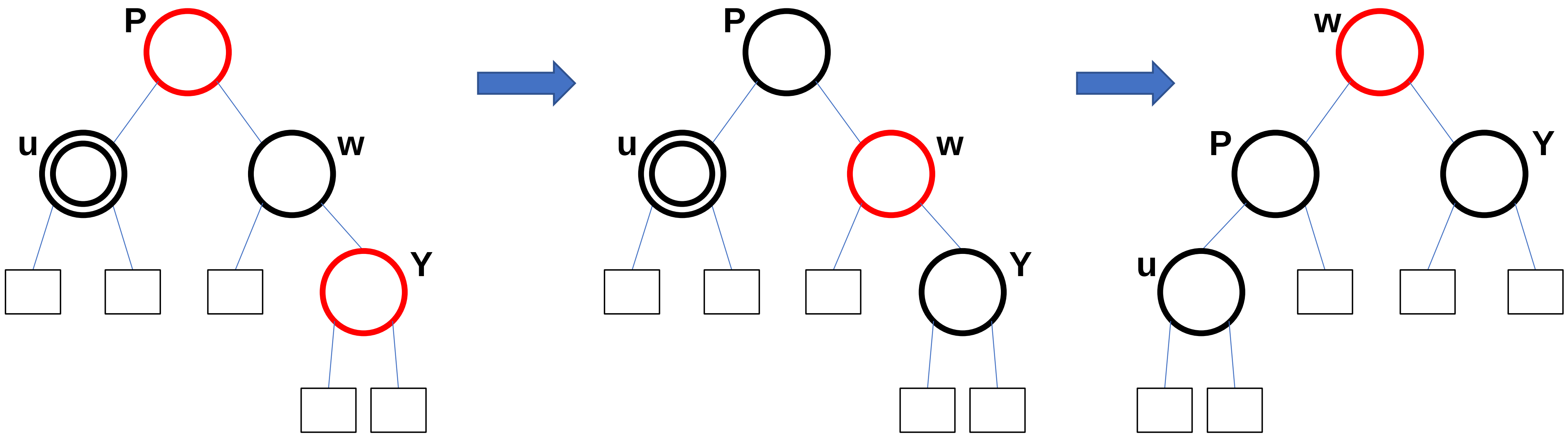
Deletion (Case 4)

Node u's sibling w is BLACK, and w's right child is RED



Deletion (Case 4)

Node u's sibling w is BLACK, and w's right child is RED



Deletion Analysis

- Case 2 is the only case in which more iterations occur.

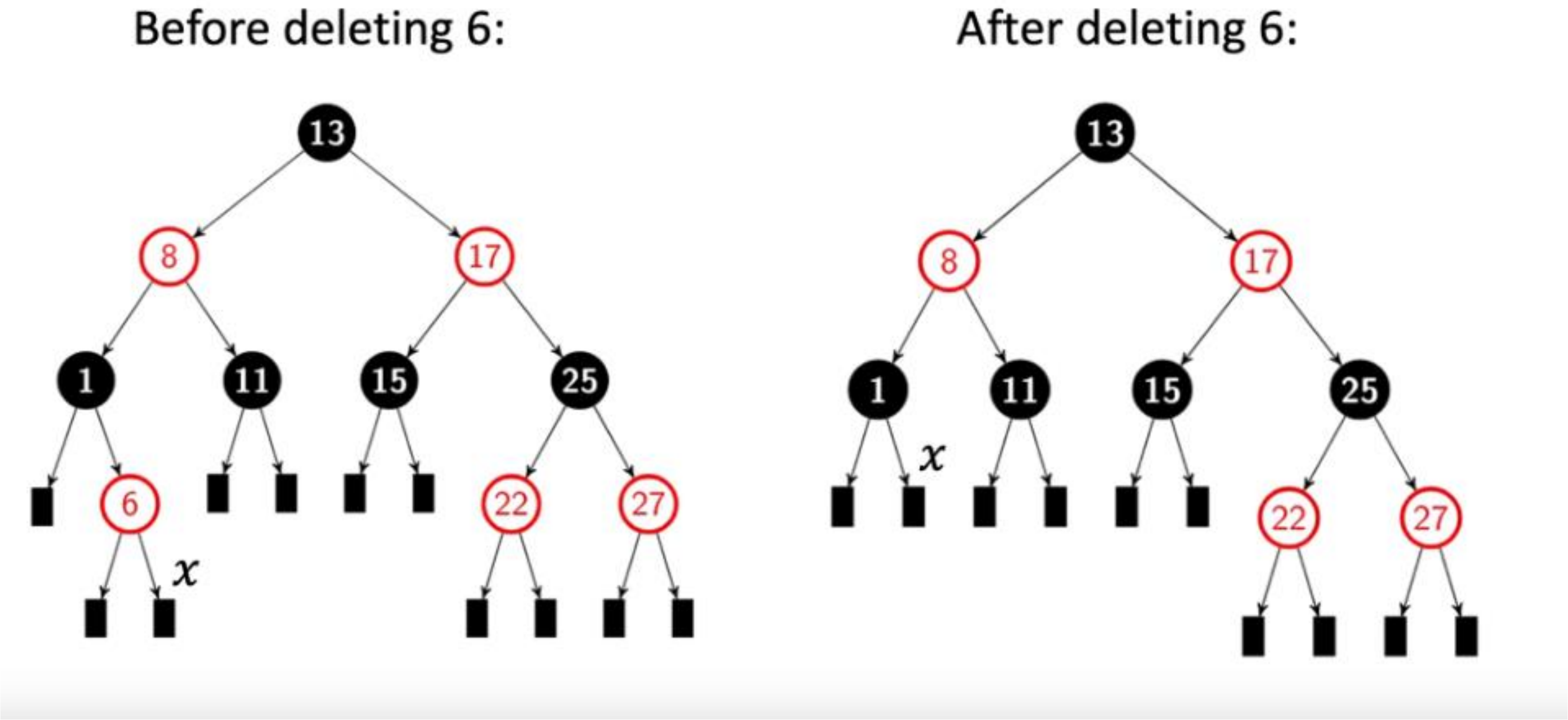
→ u moves up 1 level. Hence, O(log n) iterations.

- Each of cases 1, 3, and 4 has 1 rotation

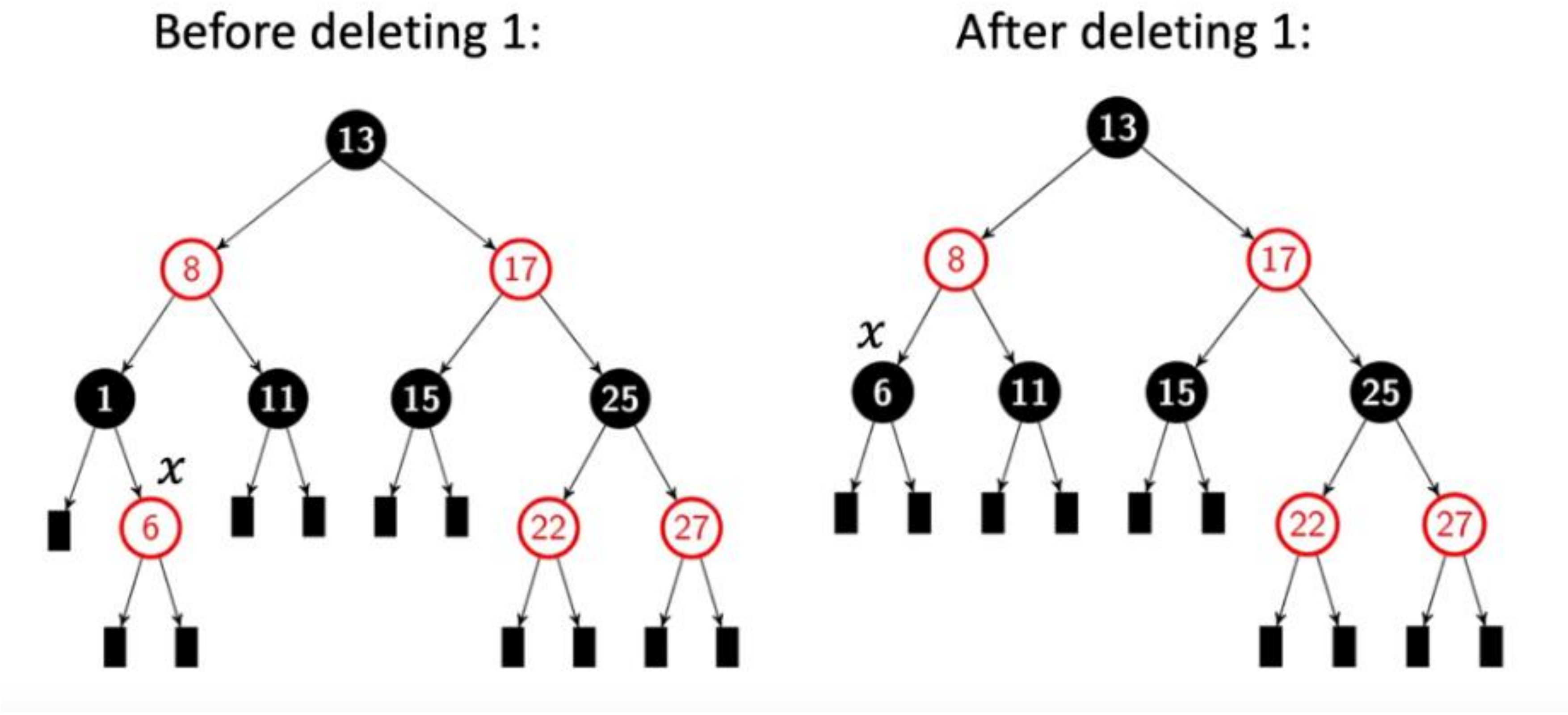
→ ≤ 3 rotations in all

→ Running time: O(log n)

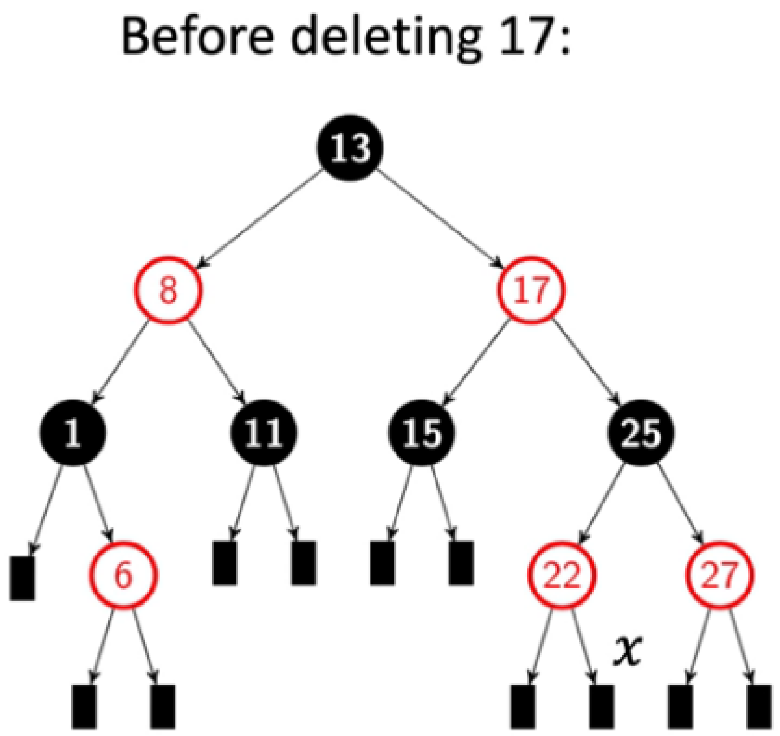
Example of Deletion (1)

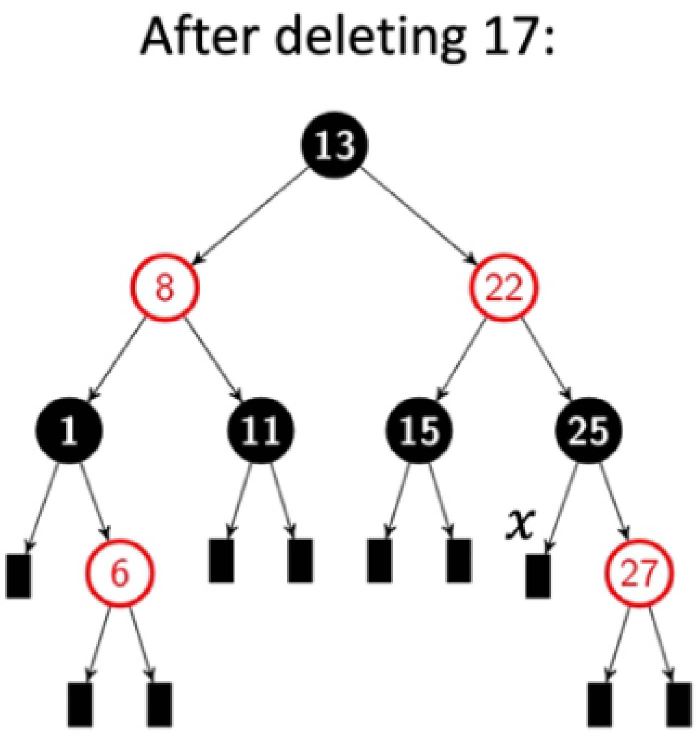


Example of Deletion (2)

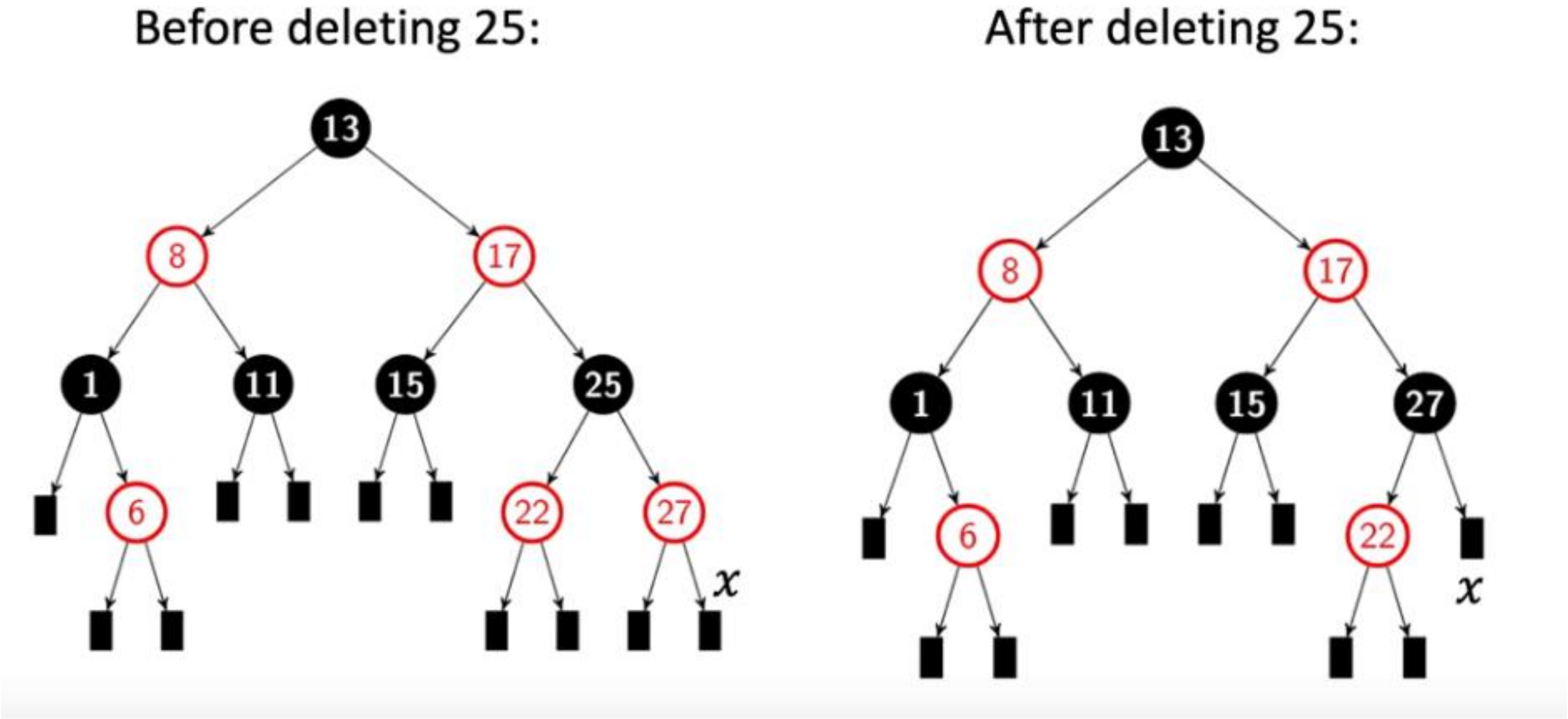


Example of Deletion (3)

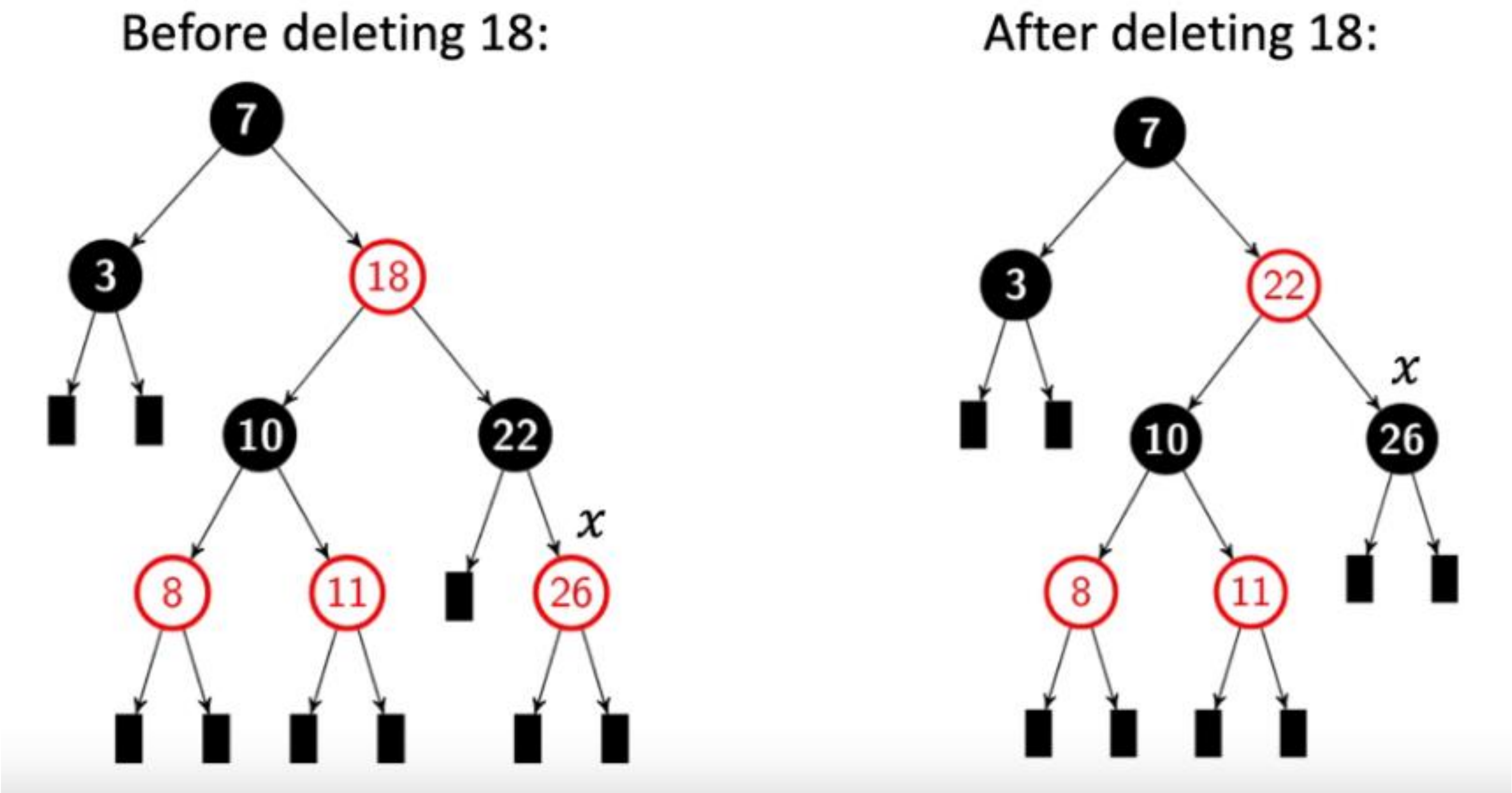




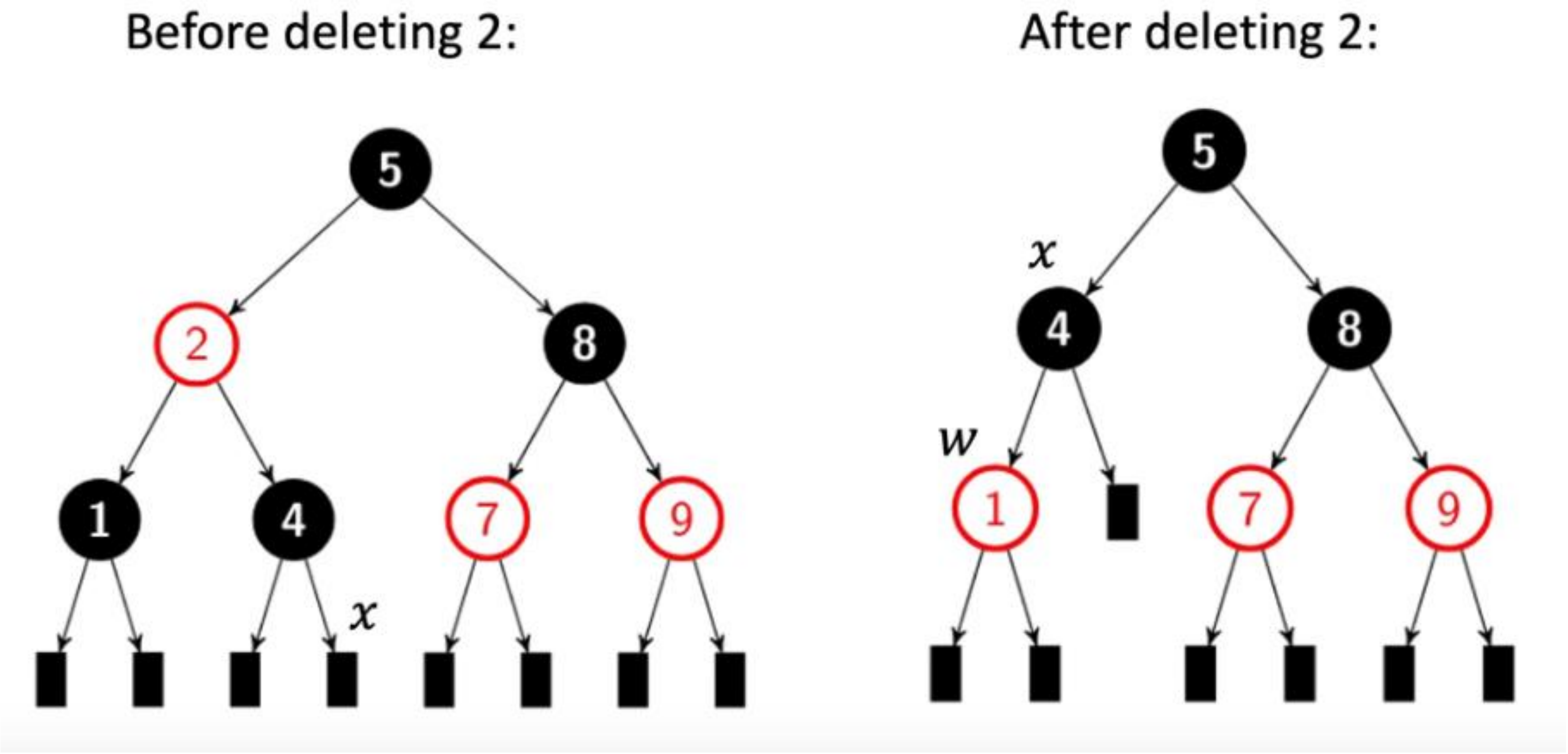
Example of Deletion (4)



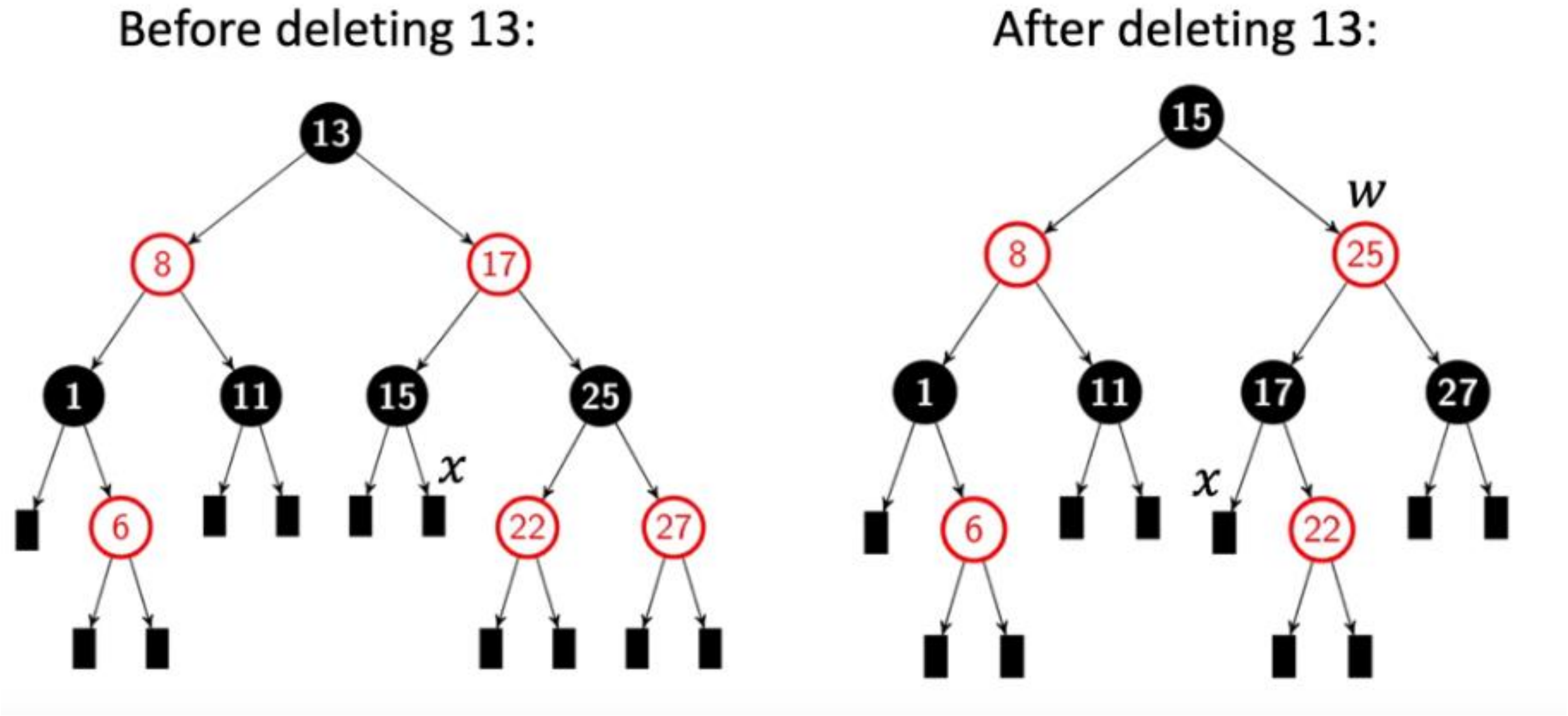
Example of Deletion (5)



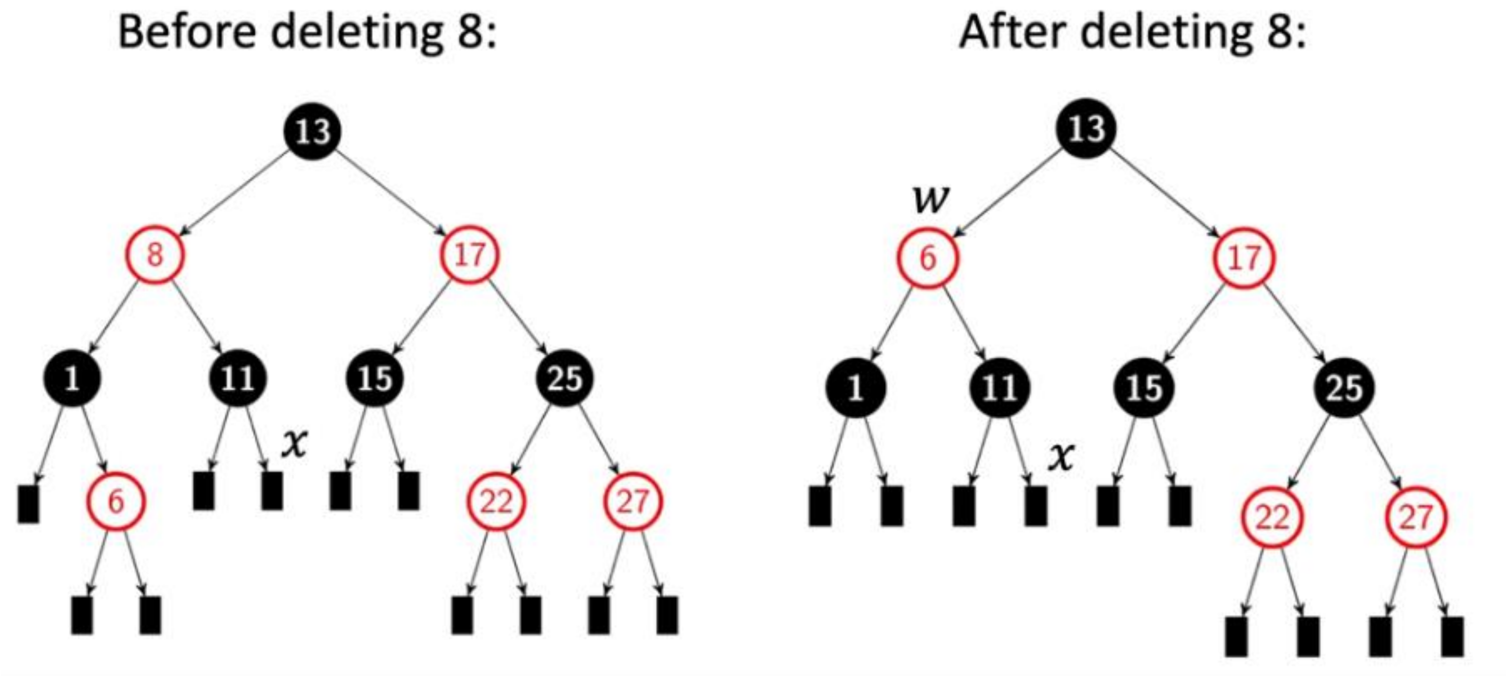
Example of Deletion (6)



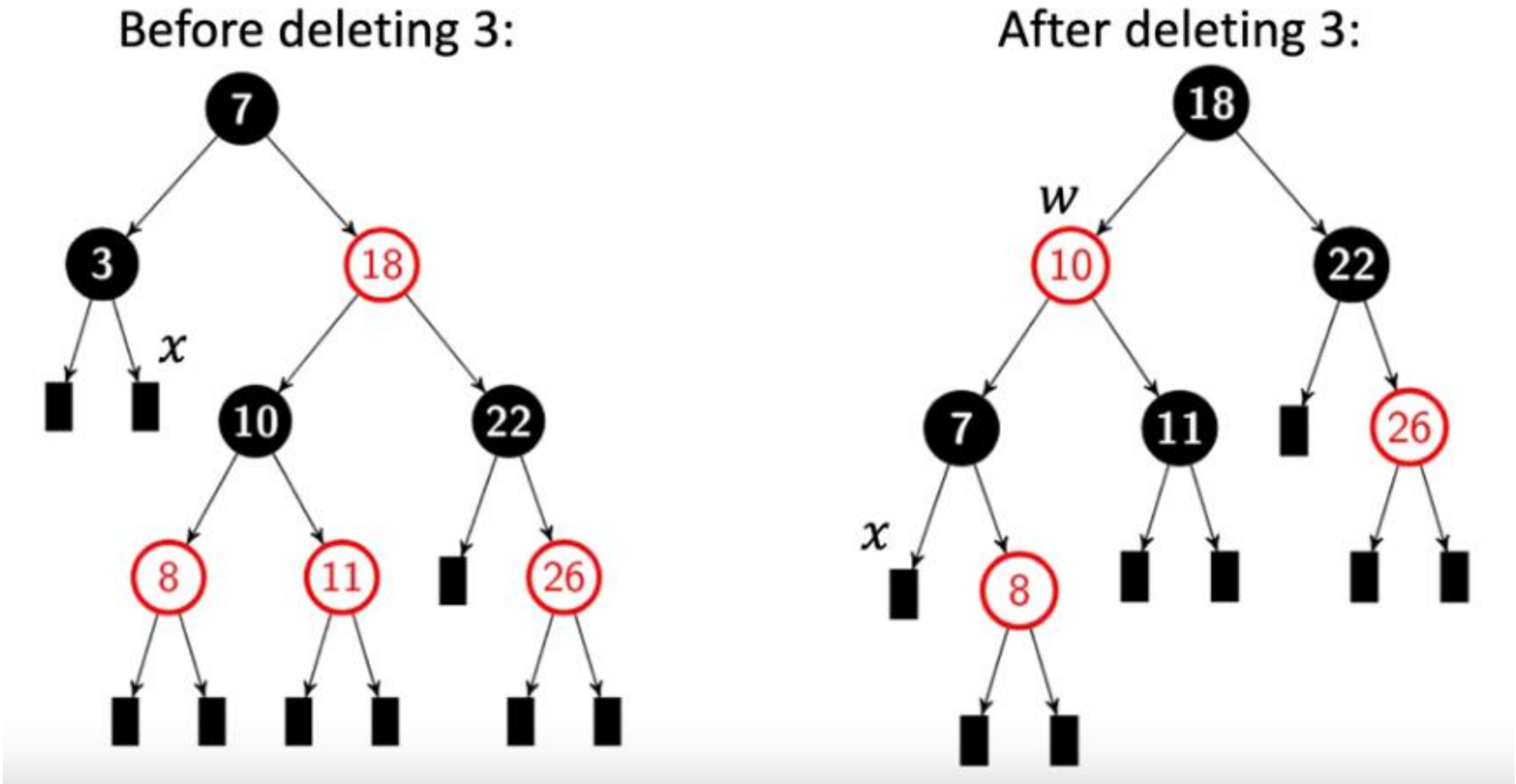
Example of Deletion (7)



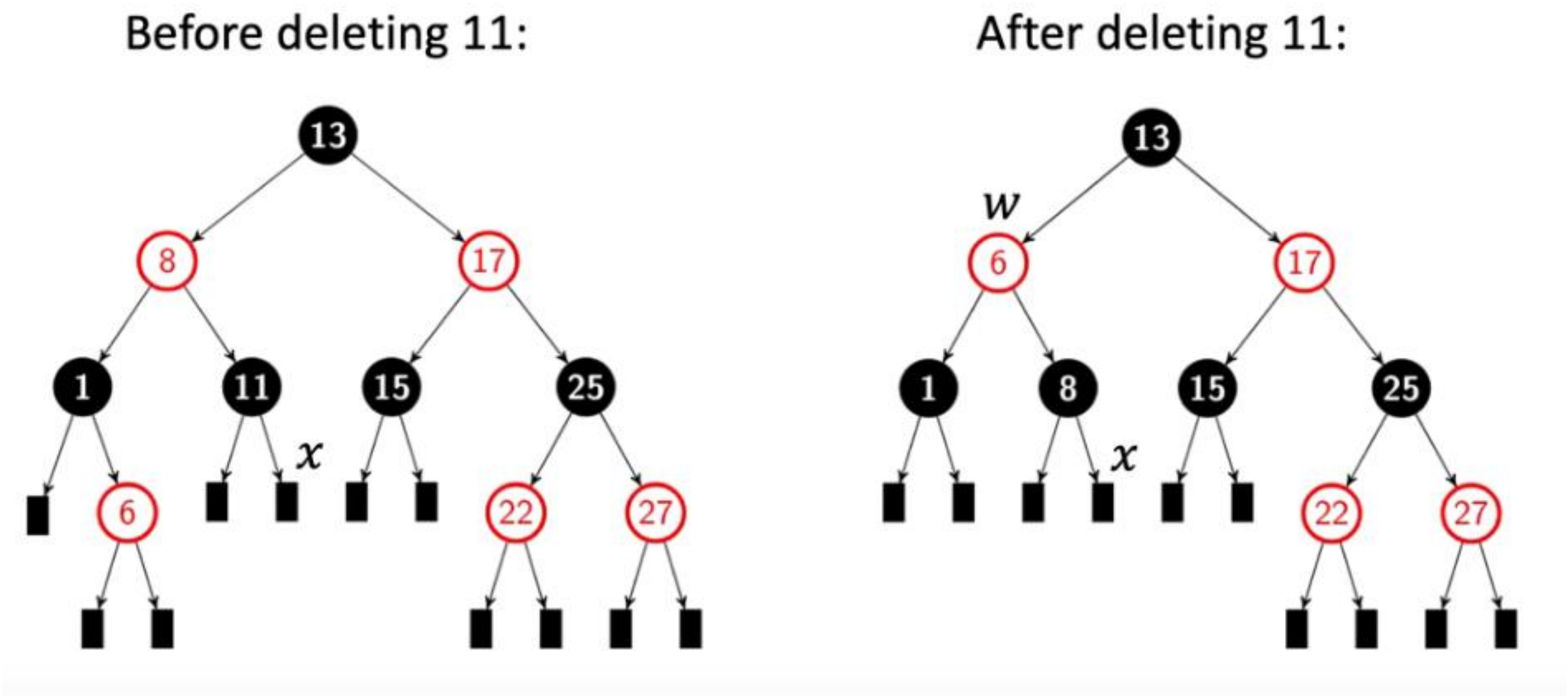
Example of Deletion (8)



Example of Deletion (9)



Example of Deletion (10)



Reference

- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms*”, September 29, 2004

- https://www.geeksforgeeks.org