Design and Analysis of Algorithms

Unit-2

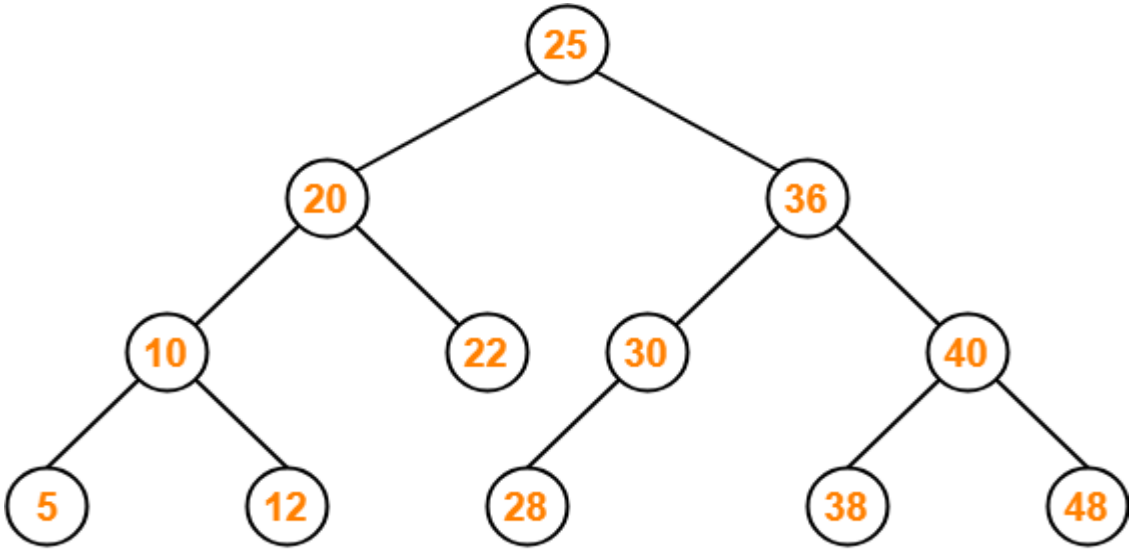
Red-Black Tree

Red-Black Tree

Binary Search Tree(BST)

A binary tree is said to be binary search tree if the value at the left child is less than value at the parent node and value at the right child is greater or equal than value at the parent node.

Example:



Binary Search Tree

Red-Black Tree

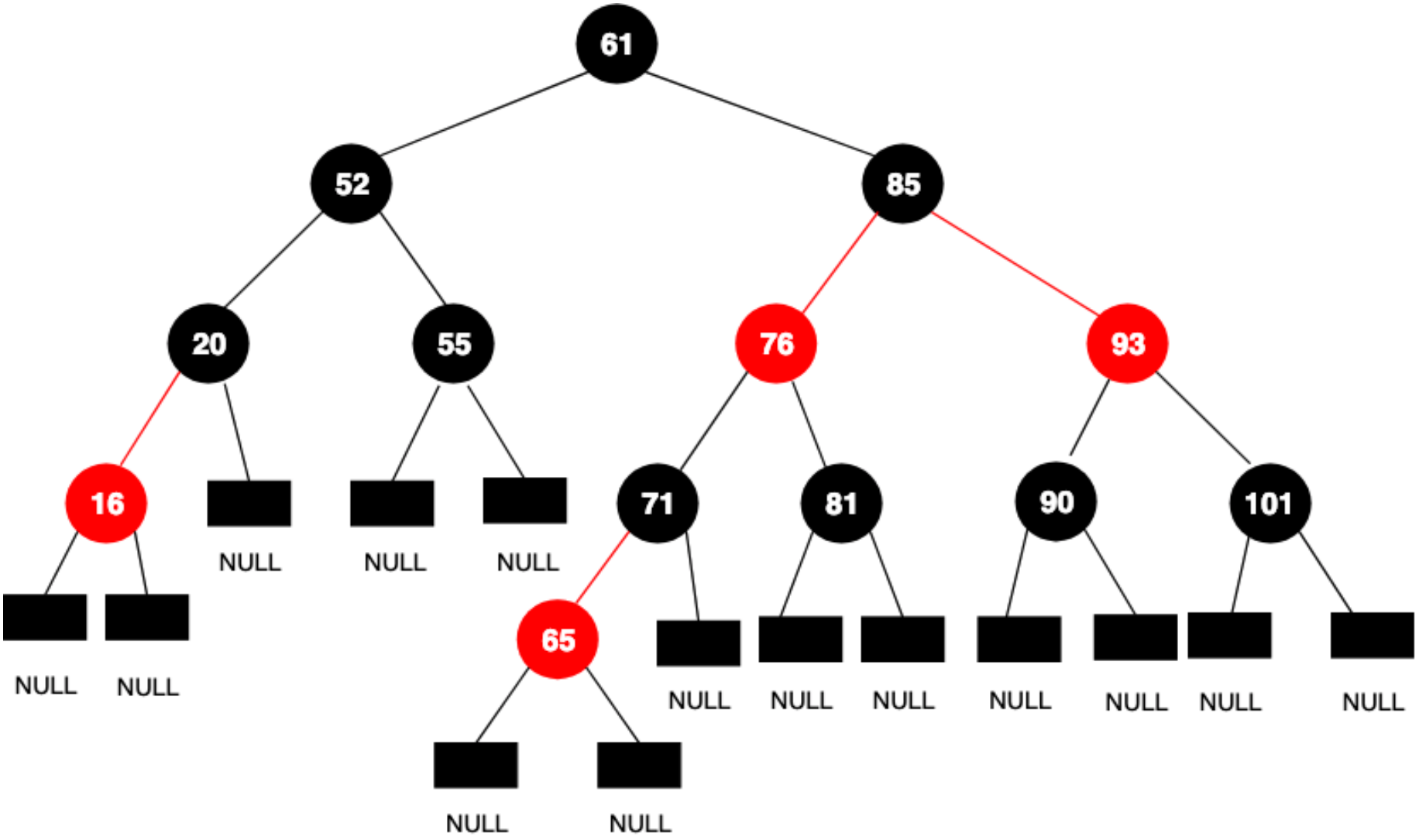
Definition

A red-black tree is a binary search tree that satisfies the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. The color of every leaf node is black and its value is always NIL.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Red-Black Tree

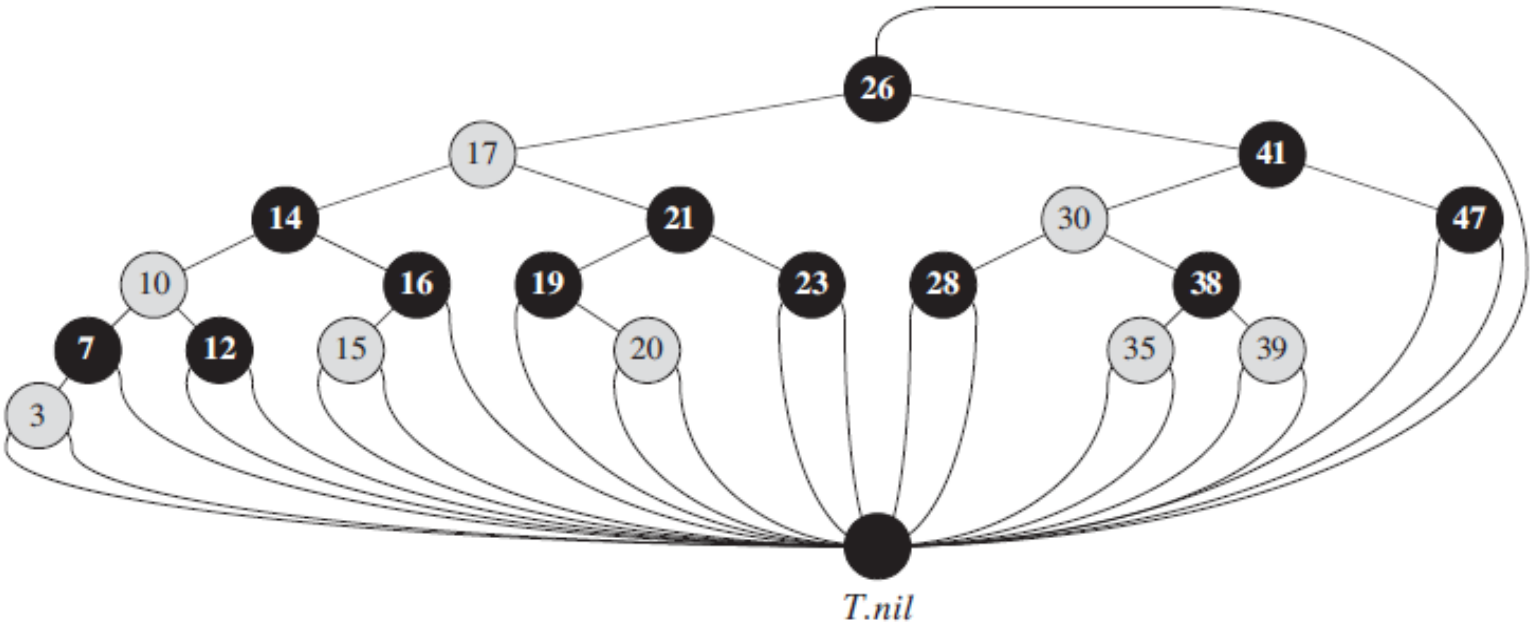
Example:



Red-Black Tree

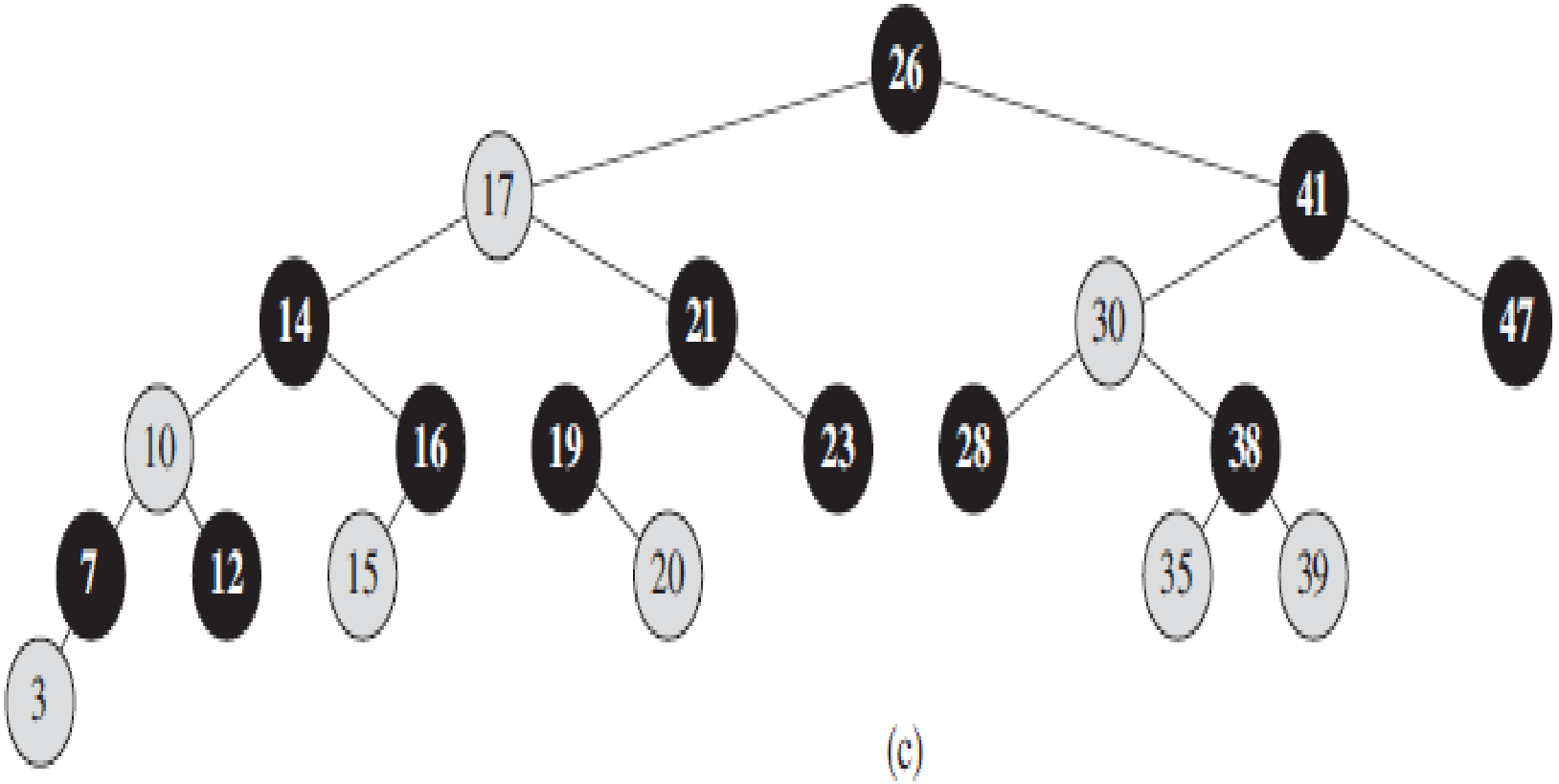
Sentinel

For a red-black tree T, the sentinel is an object with the same attributes as an ordinary node in the tree. Its color is black and its value is nil. It is represented by T.nil or nil[T].



Red-Black Tree

Red-Black tree without leaf node



(c)

Red-Black Tree

Black height of a node

Black height of a node x in the red-black tree is the number of black nodes on any downward path from node x to a leaf node but not including node x. It is denoted by bh(x).

Black height of a Red-Black tree

Black height of a red-black tree is equal to the black height of the root node.

Red-Black Tree

Theorem: A red-black tree with n internal nodes has height at most 2lg(n+1).

Proof: Before proving the theorem, first we will prove the following statement.

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

We will prove the statement (1) using induction method. We will use induction parameter as the height of the red-black tree.

For height, h = 0.

Red-black tree of height 0 is only single node i.e. it will be following :-

Clearly, minimum number of internal nodes in this tree = 0

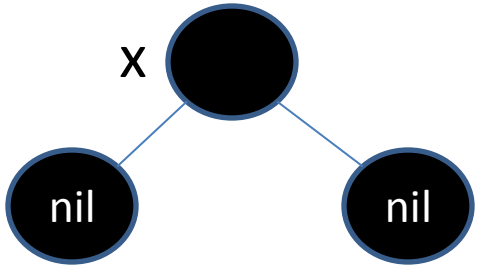
and $2^{bh(x)}-1 = 2^{0}-1 = 1-1 = 0$

Therefore, statement (1) is true for height h=0.

Red-Black Tree

For height, h = 1.

Red-black tree of height h = 1 will be the following:-



Clearly, minimum number of internal nodes in this tree = 1

And $2^{bh(x)}-1 = 2^{1}-1 = 2-1 = 1$

Therefore, statement (1) is also true for height h=1.

Now, we assume statement (1) is true for children of node x.
We will prove the statement for node x.

Red-Black Tree

Now,

The minimum number of internal nodes in the subtree rooted at node x

= Minimum number of internal nodes in the subtree rooted at left child of node x + Minimum number of internal nodes in the subtree rooted at right child of node x + 1

Since black height of each child of node x has either bh(x) or bh(x)-1, depending on the color of child. If the color of the child is red then black height of child is bh(x) but if the color of the child is black then black height of child is bh(x) -1. Therefore,

The minimum number of internal nodes in the subtree rooted at node x

$$= 2^{(bh(x)-1)} - 1 + 2^{(bh(x)-1)} - 1 + 1$$

$$= 2 \cdot 2^{(bh(x)-1)} - 1$$

$$= 2^{bh(x)} - 1$$

Therefore, statement (1) is also proved for node x of any height.

Red-Black Tree

Now, we will prove the given theorem using statement (1).

Let h is the height of the red-black tree.

Using property (4) of the red-black tree, minimum number of black nodes on any path from root node to the leaf node will be h/2. Therefore, minimum black height of red-black tree of height h will be h/2.

Now, the minimum number of internal nodes in red-black tree of height h $= 2^{h/2} - 1$

Since the gien number of internal nodes in red-black tree is n, therefore

$$2^{h/2} - 1 \leq n \Rightarrow 2^{h/2} \leq n + 1$$

$$\Rightarrow h/2 \leq \lg(n+1) \Rightarrow h \leq 2\lg(n+1)$$

Therefore, the maximum height of red-black tree will be $2\lg(n+1)$.

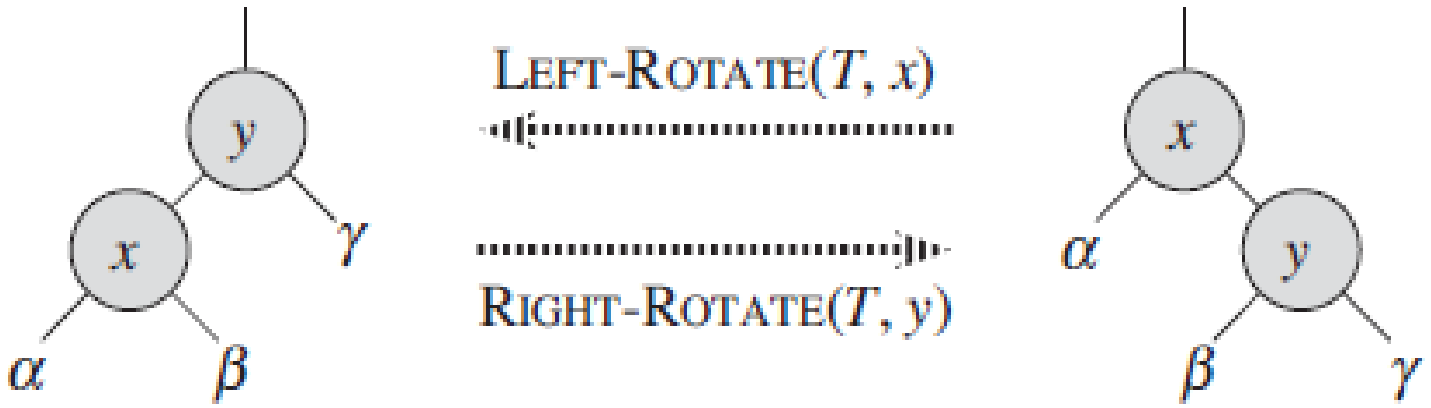
Now, it is proved.

Rotation operation

We use two types of rotations in the insertion and deletion of a node in the red-black tree.

(1) Left rotation

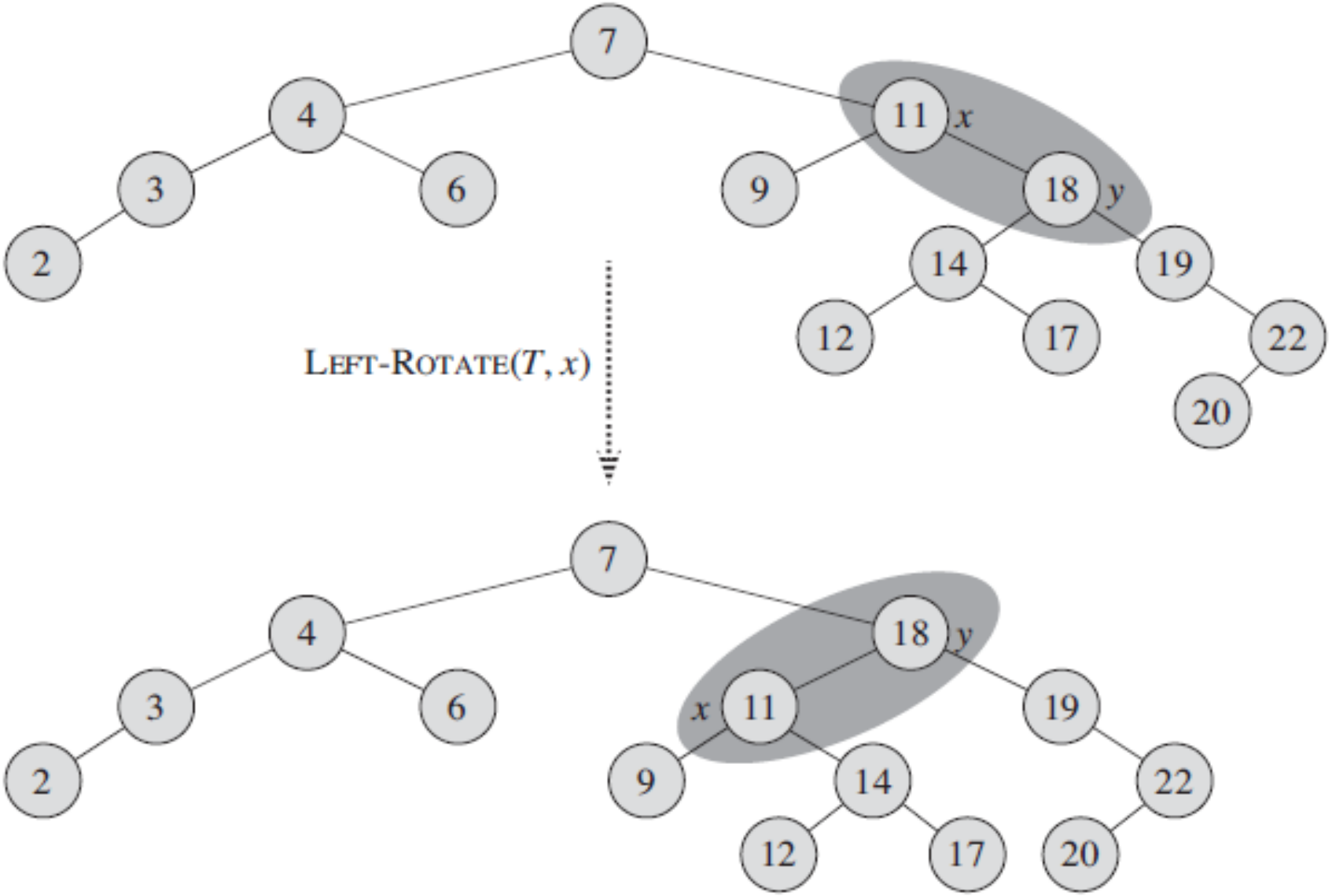
(2) Right rotation



Left rotation algorithm

```
LEFT-ROTATE(T, x)
 1  y = x.right                 // set y
 2  x.right = y.left            // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4       y.left.p = x
 5  y.p = x.p                   // link x's parent to y
 6  if x.p == T.nil
 7       T.root = y
 8  elseif x == x.p.left
 9       x.p.left = y
10  else x.p.right = y
11  y.left = x                  // put x on y's left
12  x.p = y
```

Left rotation algorithm



Right rotation algorithm

```
RIGHT-ROTATION(T, y)
1.      x = y.left
2.      y.left = x.right
3.      if x.right  ≠ T.nil
4.              x.right.p = y
5.      x.p = y.p
6.      if y.p = = T.nil
7.              T.root = x
8.      else if  y = = y.p.right
9.              y.p.right = x
10.     else  y.p.left = x
11.     x.right = y
12.     y.p = x
```

Insertion operation

Suppose we want to insert a node z into red-black tree T. We use the following steps for this purpose:-

1. Insert node z into red-black tree using binary search tree insertion process.
2. Make the color of new node z to be **red**.
3. If the color of parent of node z is **black** then we make the color of root node to be **black** and stop the process.
4. Otherwise we maintain the properties of red-black tree using the following procedure.

    (4-a) We start a loop and continue until color of parent of node z turns **black**.

    (4-b) If parent of node z is the left child of its parent then we do the following actions:-

    (4b-i) Find the sibling of parent of node z i.e. uncle of z. Let it is denoted by node y.

Insertion operation

(4b-ii) Now, there will be three cases.

Case-1: If color of y is **red** then we do the following actions:-

(1) z.p.color = **black**

(2) y.color = **black**

(3) z.p.p.color = **red**

(4) z = z.p.p

Case-2: If color of y is **black** and z is right child then we do following actions:-

(1) z = z.p

(2) Perform left rotation at node z.

Case-3: If color of y is **black** and z is left child then we do following actions:-

(1) z.p.color = **black**

(2) z.p.p.color = **red**

(3) Perform right rotation at node z.p.p i.e. at grandparent of z

Insertion operation

(4-c) If parent of node z is the right child of its parent then we do the following actions:-

(4c-i) Find the sibling of parent of node z i.e. uncle of z. Let it is denoted by node y.

(4c-ii) Now, there will be three cases.

Case-1: If color of y is **red** then we do the following actions:-

(1) z.p.color = **black**

(2) y.color = **black**

(3) z.p.p.color = **red**

(4) z = z.p.p

Insertion operation

Case-2: If color of y is **black** and z is left child then we do following actions:-

(1) z = z.p

(2) Perform right rotation at node z.

Case-3: If color of y is **black** and z is right child then we do following actions:-

(1) z.p.color = **black**

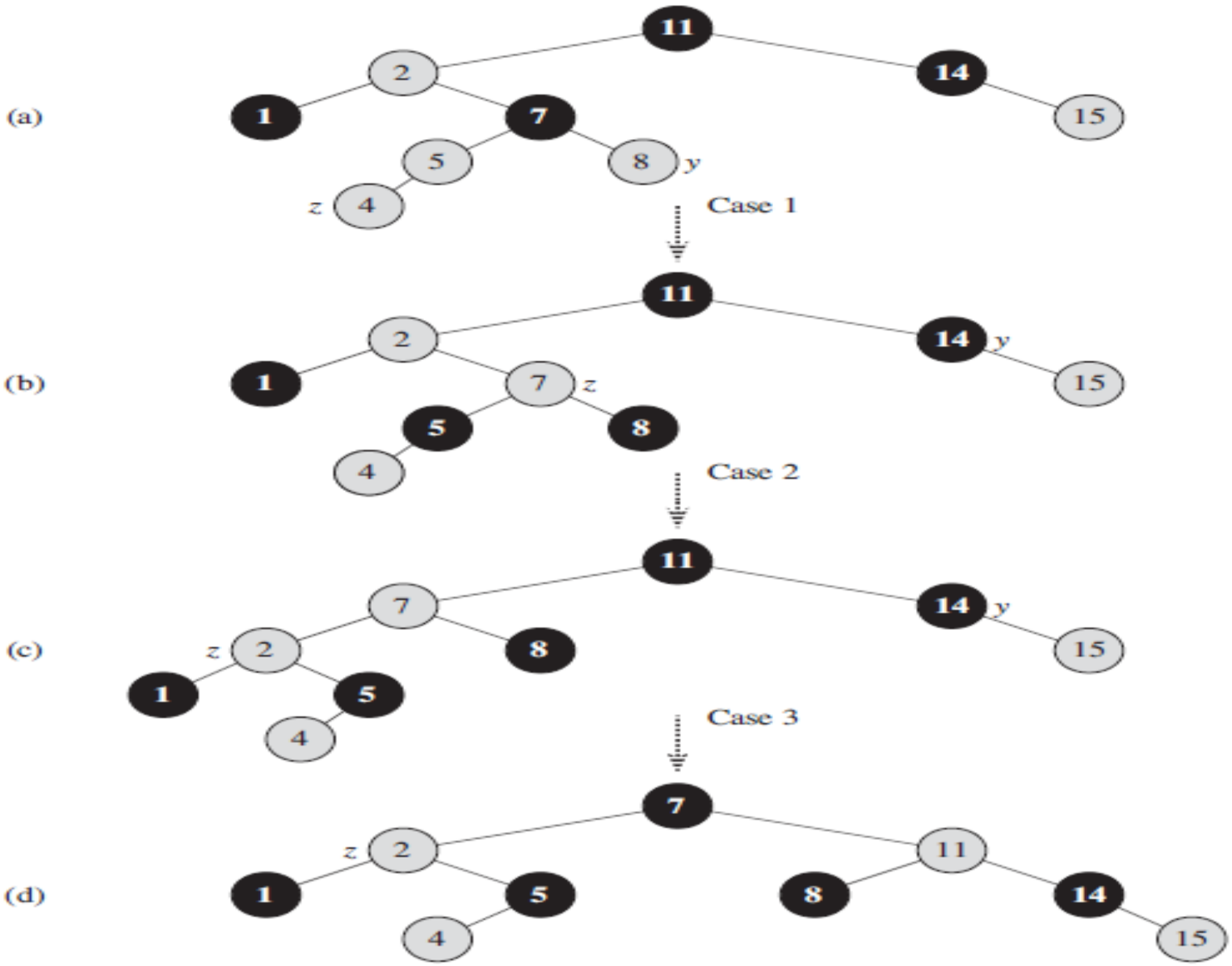
(2) z.p.p.color = **red**

(3) Perform left rotation at node z.p.p i.e. at grandparent of z

(4-d) After exit from the loop, we make the color of root node to be **black** i.e.

T.root.color = **black**

Insertion operation

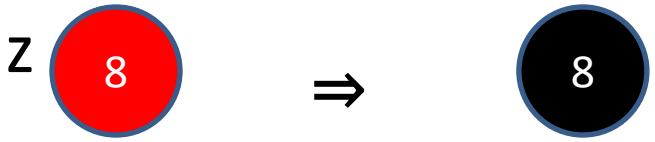


Insertion operation

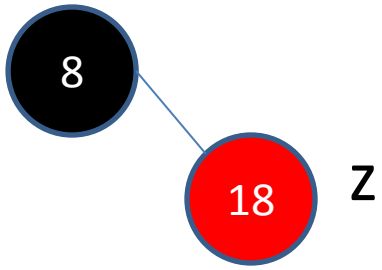
Example: Create the red-black tree by inserting following sequence of numbers:- 8, 18, 5, 15, 17, 25, 40 and 80.

Initially, red-black tree is empty.

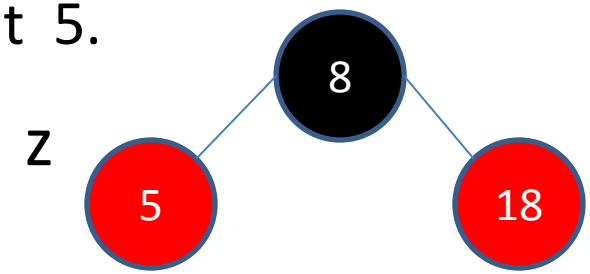
Solution: Initially tree is empty. First insert element 8.



Now, Insert next element 18.

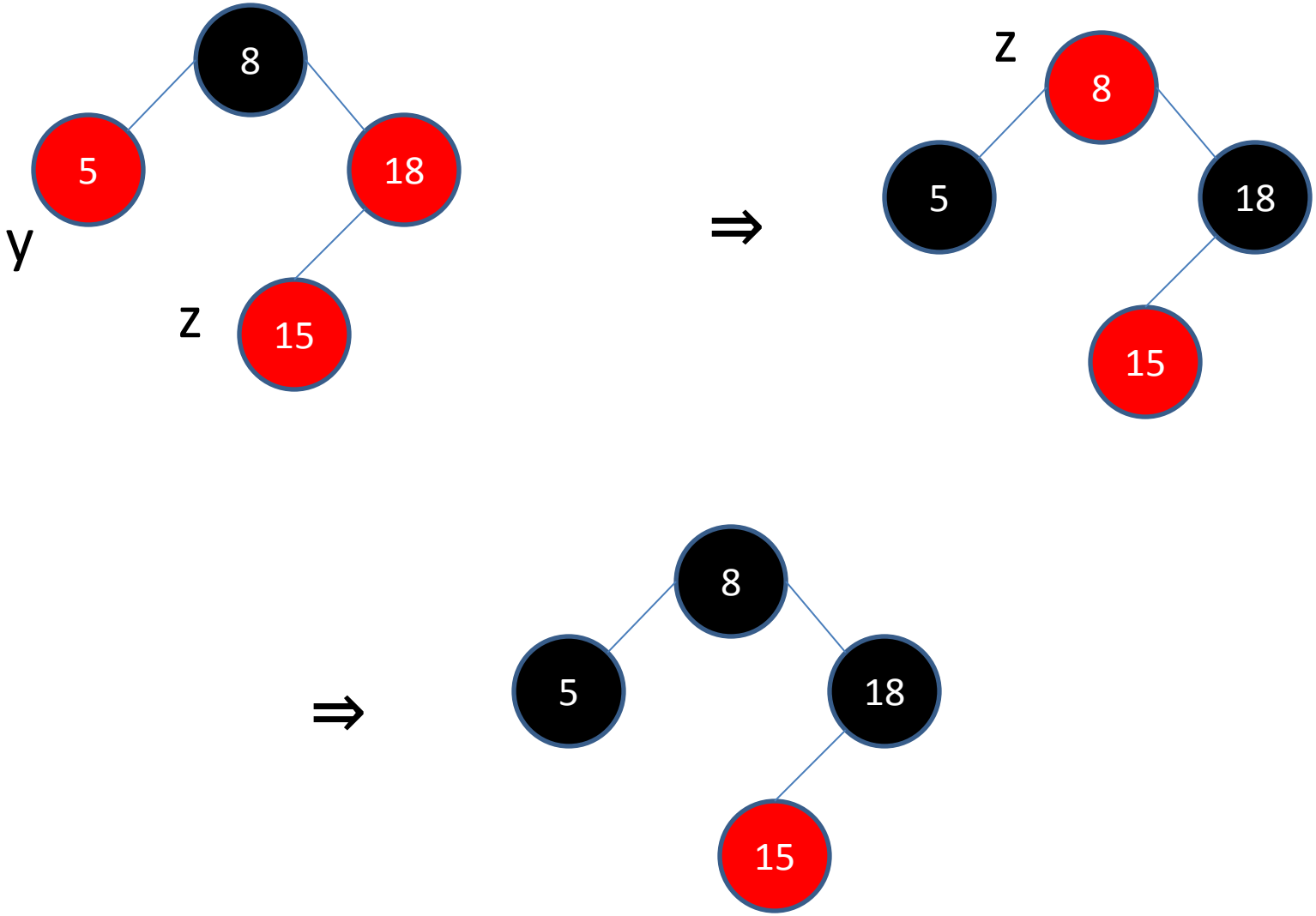


Now, Insert next element 5.



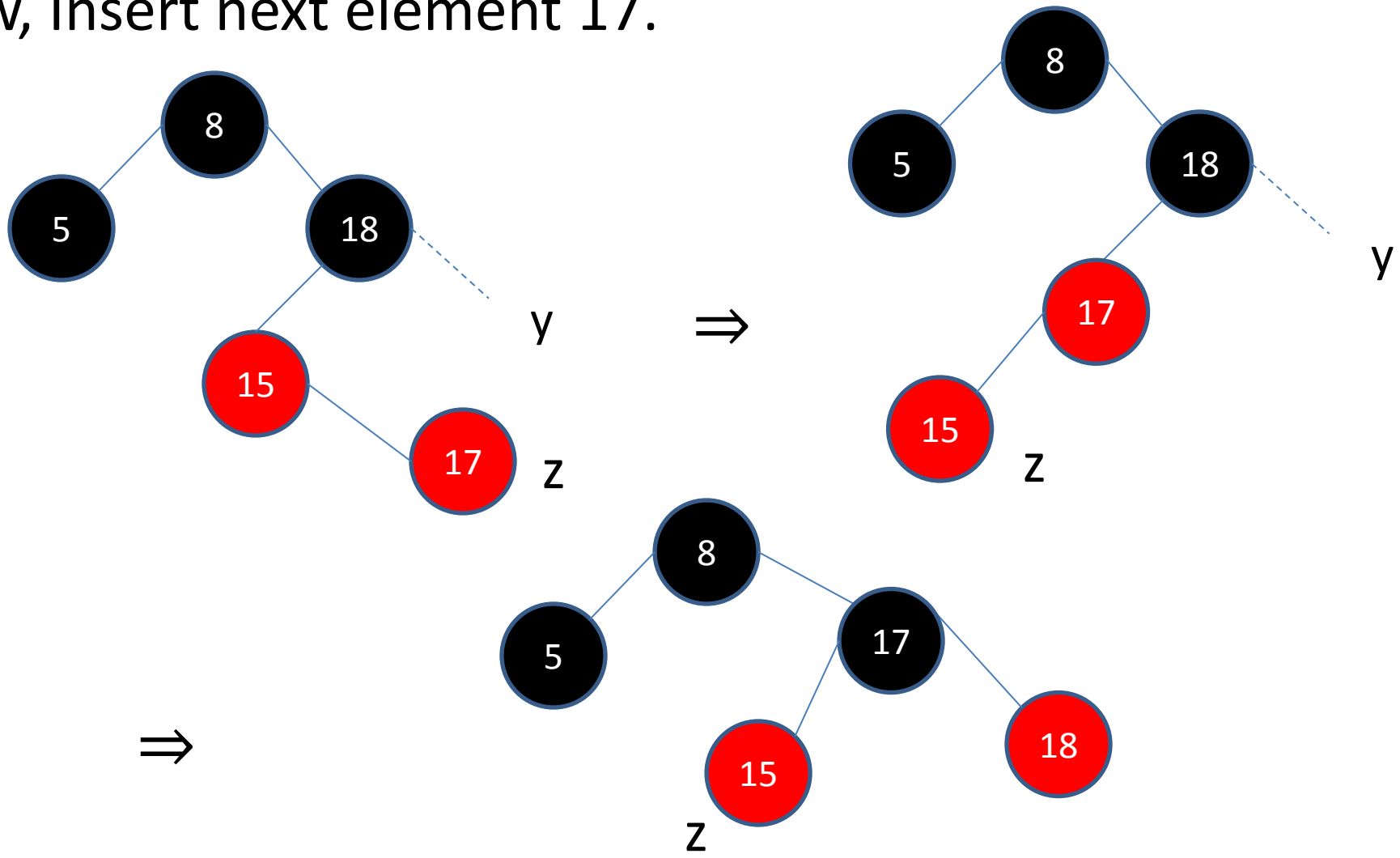
Insertion operation

Now, Insert next element 15.



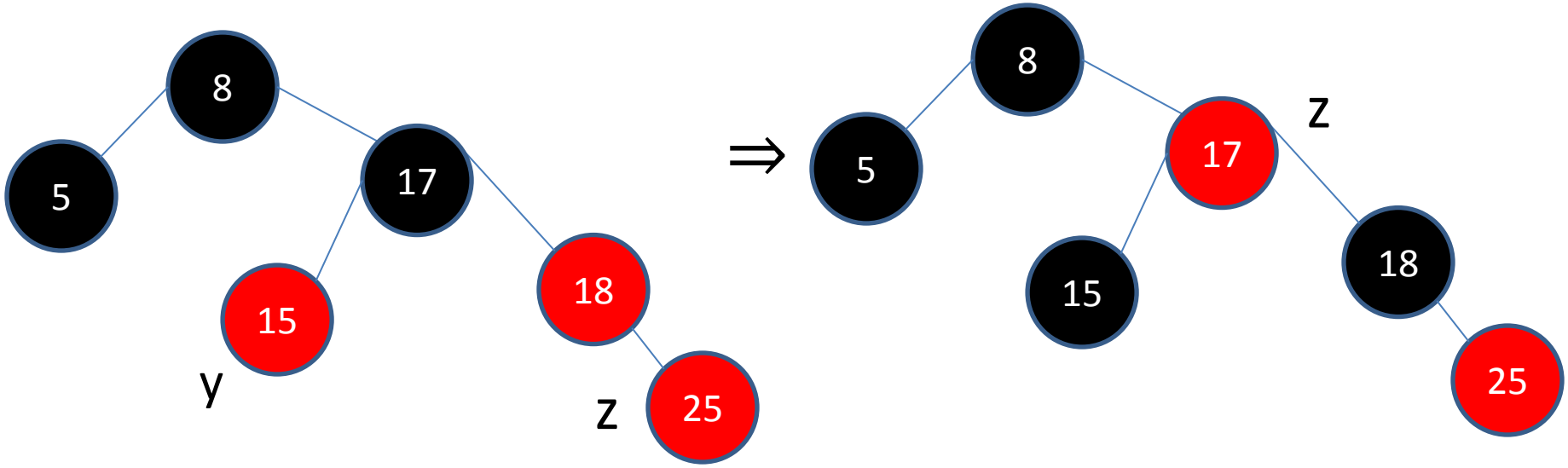
Insertion operation

Now, Insert next element 17.



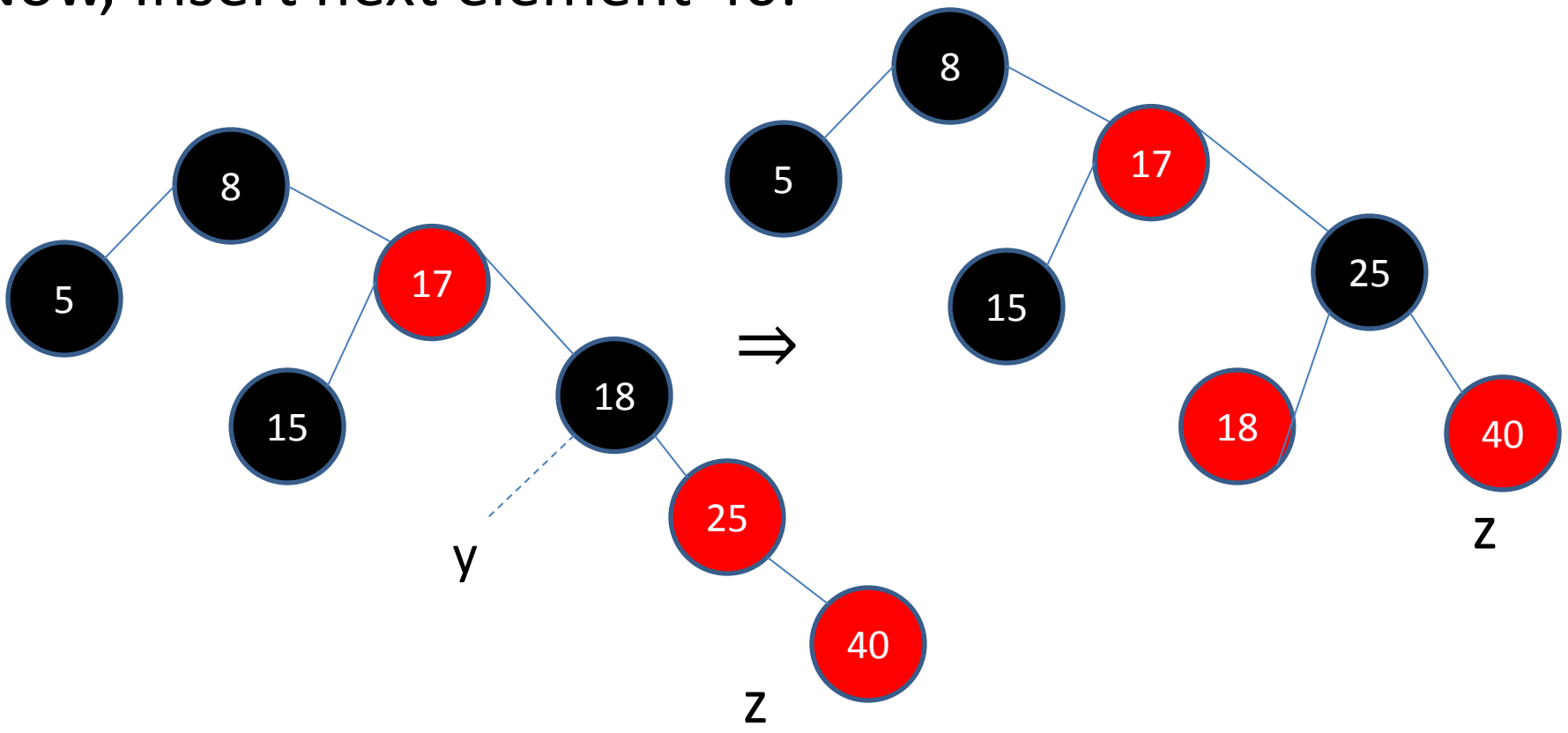
Insertion operation

Now, Insert next element 25.



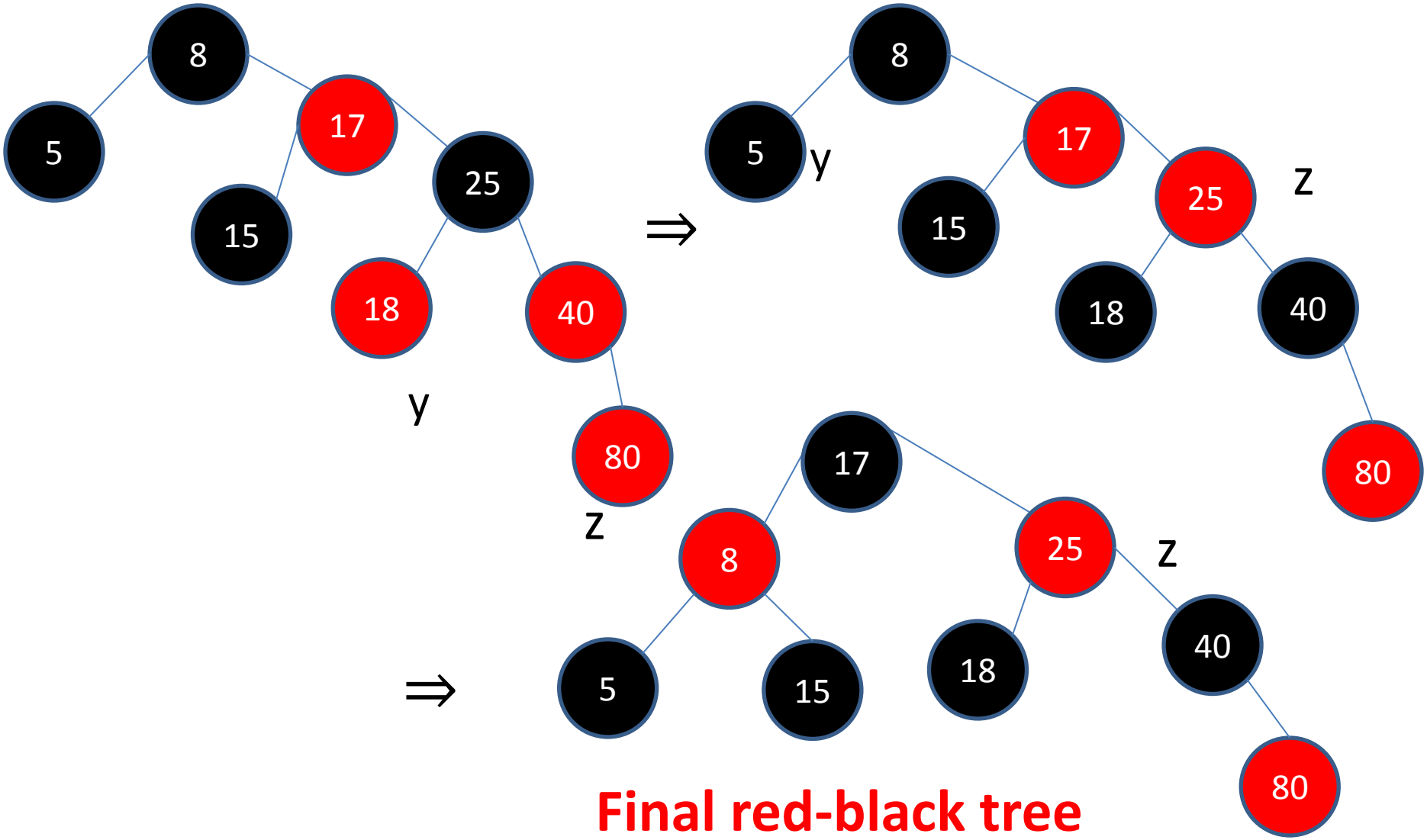
Insertion operation

Now, Insert next element 40.



Insertion operation

Now, Insert next element 80.



Final red-black tree

Insertion Algorithm

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

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
 6              y.color = BLACK
 7              z.p.p.color = RED
 8              z = z.p.p
 9          else if z == z.p.right
10                  z = z.p
11                  LEFT-ROTATE(T, z)
12              z.p.color = BLACK
13              z.p.p.color = RED
14              RIGHT-ROTATE(T, z.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

Time complexity of insertion algorithm is $\theta(\log n)$.

Deletion operation

Suppose we want to delete a node z from a red-black tree T.
We use the following steps for this purpose:-

1. First we delete node z using binary search tree deletion process.
2. Find node y in the following way:-
   - ❖ If node z has two children then y will be successor of z otherwise y will be z.
3. After finding y, we find x in the following way:-
   - ❖ If node y has left child then x will be left child of y otherwise x will be right child of y.
4. If color of y is **red**, then we terminate the process.
5. If color of y is **black**, then we maintain the properties of red-black tree in the following way:-

Deletion operation

(5-a) We start and continue a loop if x is not root node and color of x is **black**.

(5-b) if x is the left child then we do the following actions:-

(i) Find sibling of x. Let it is denoted by w.

(ii) There will be four cases:-

Case-1: If color of w is **red**, then we do the following actions:-

(1) w.color = **black**

(2) x.p.color = **red**

(3) Apply left rotation at parent of node x.

Case-2: If color of w is **black** and color of its both children is also **black**, then we do the following actions:-

(1) w.color = **red**

(2) x = x.p

Deletion operation

Case-3: If color of w is **black** and color of its left child is **red** and color of its right child is **black**, then we do the following actions:-

(1) w.left.color = **black**

(2) w.color = **red**

(3) Apply right rotation at node w.

Case-4: If color of w is **black** and color of its right child is **red**, then we do the following actions:-

(1) w.right.color = **black**

(2) w.color = x.p.color

(3) x.p.color = **black**

(4) Apply left rotation at parent of node x.

(5) x = T.root

Deletion operation

(5-c) if x is the right child then we do the following actions:-

(i) Find sibling of x. Let it is denoted by w.

(ii) There will be four cases:-

Case-1: If color of w is **red**, then we do the following actions:-

(1) w.color = **black**

(2) x.p.color = **red**

(3) Apply right rotation at parent of node x.

Case-2: If color of w is **black** and color of its both children is also **black**, then we do the following actions:-

(1) w.color = **red**

(2) x = x.p

Deletion operation

Case-3: If color of w is **black** and color of its right child is **red** and color of its left child is **black**, then we do the following actions:-

(1) w.right.color = **black**

(2) w.color = **red**

(3) Apply left rotation at node w.

Case-4: If color of w is **black** and color of its left child is **red,** then we do the following actions:-

(1) w.left.color = **black**

(2) w.color = x.p.color

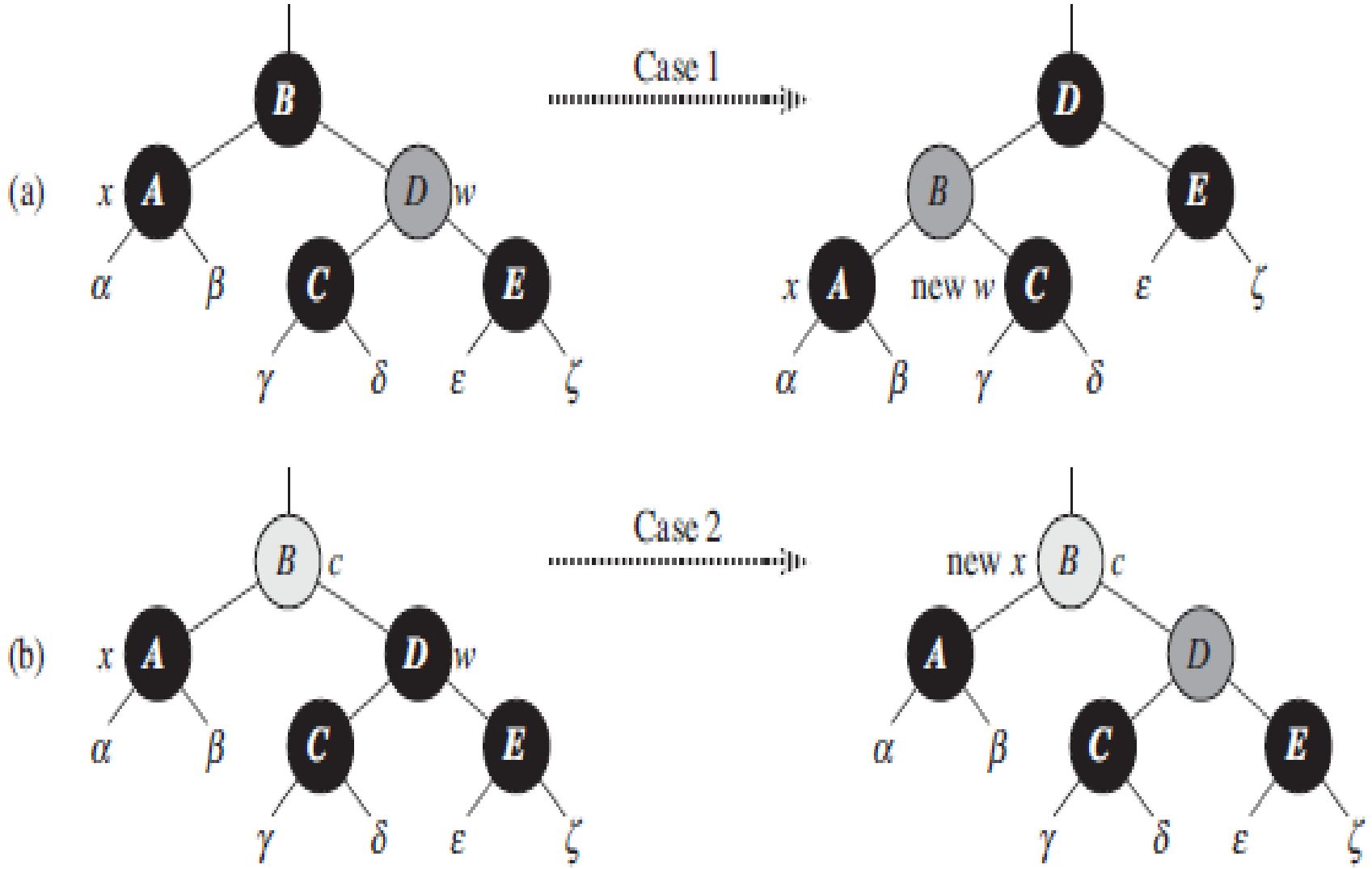
(3) x.p.color = **black**

(4) Apply right rotation at parent of node x.

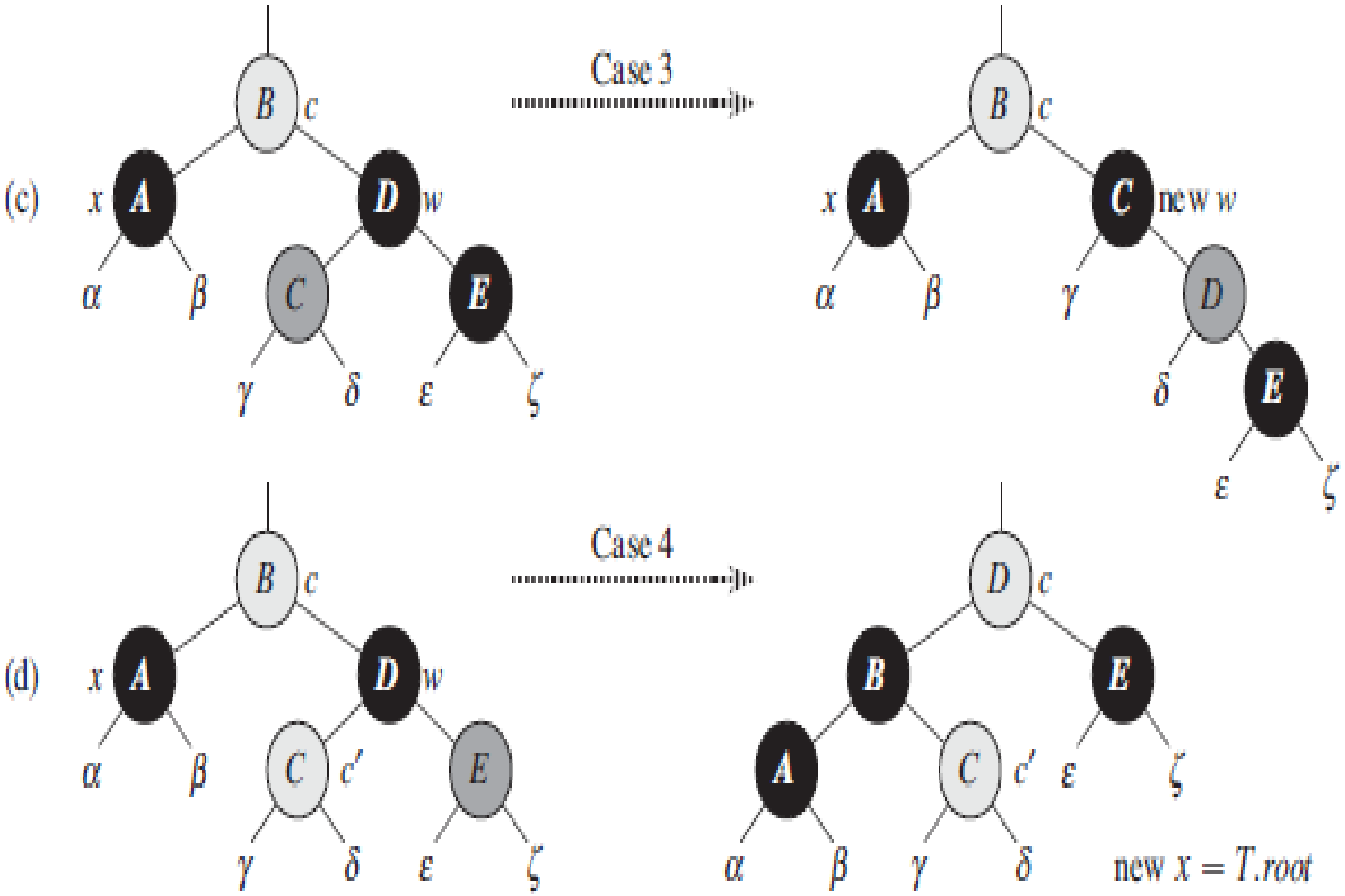
(5) x = T.root

(5-d) After exit from the loop, we make the color of node x to the **black** i.e. x.color = **black**

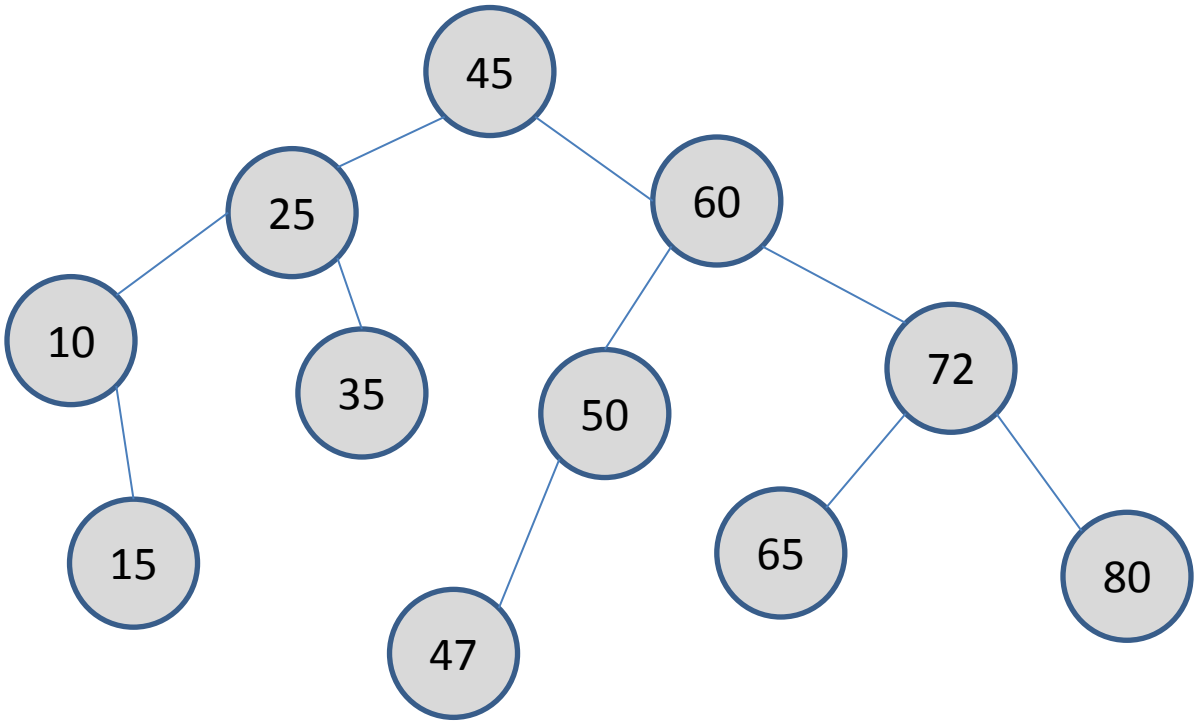
Deletion operation



Deletion operation

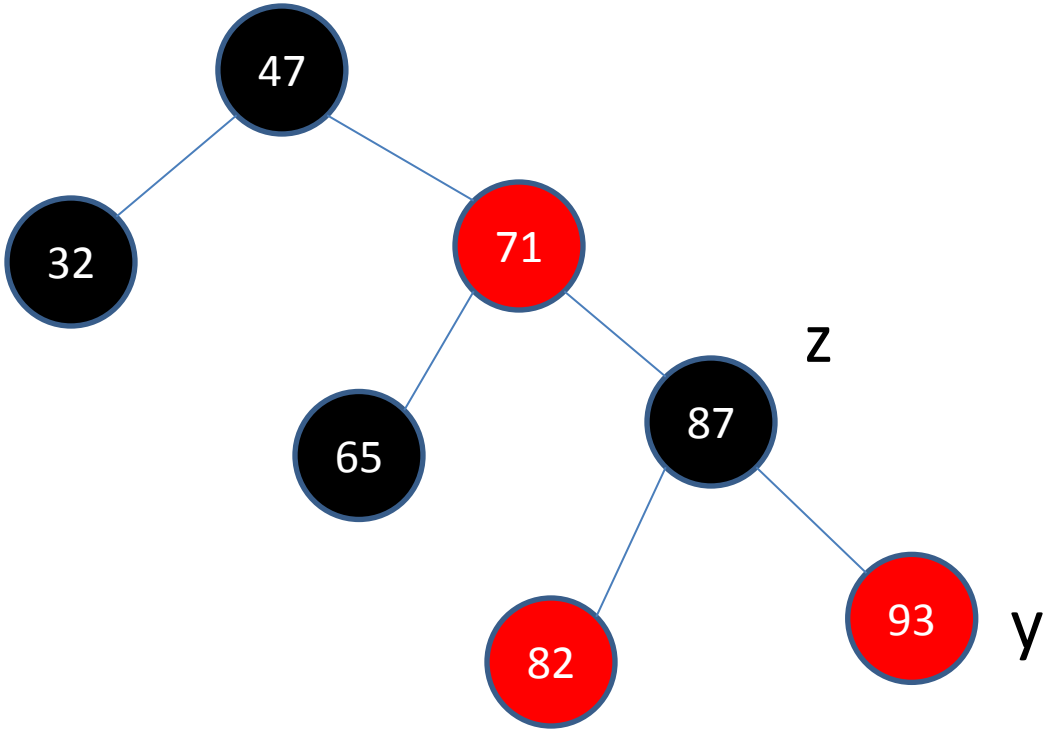


Binary search tree deletion



Deletion operation

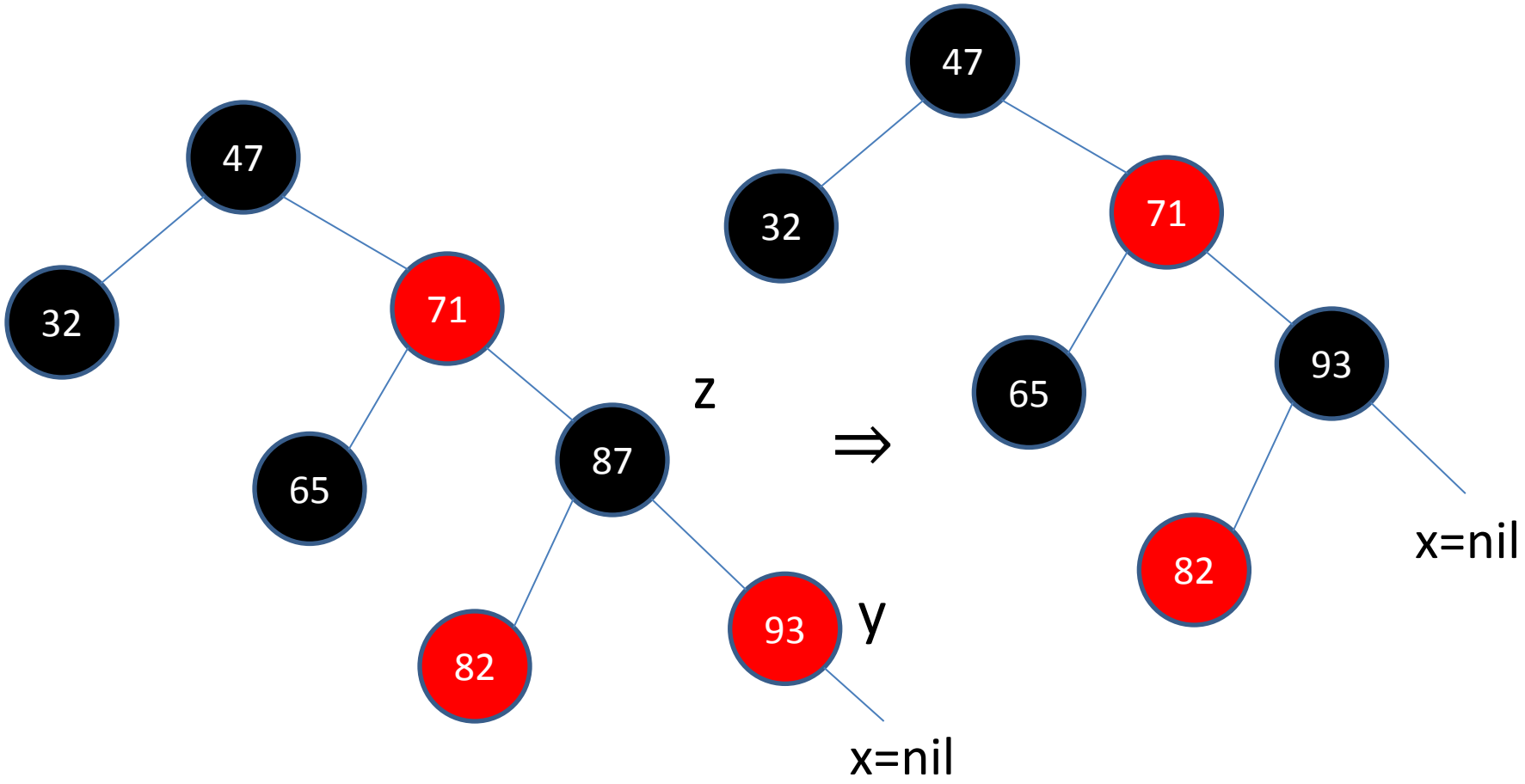
Example: Consider the following red-black tree



Delete the element 87, 32 and 71 in order.

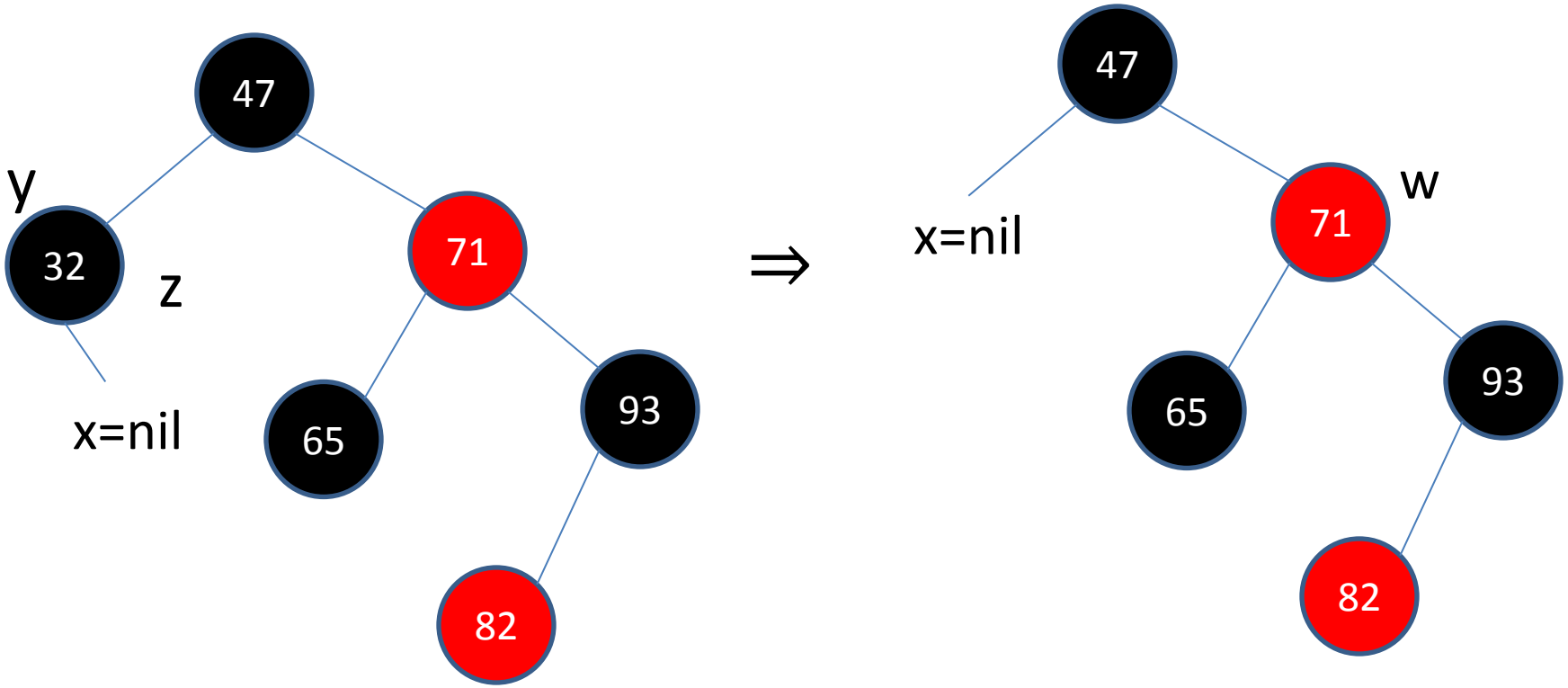
Deletion operation

Deletion of 87



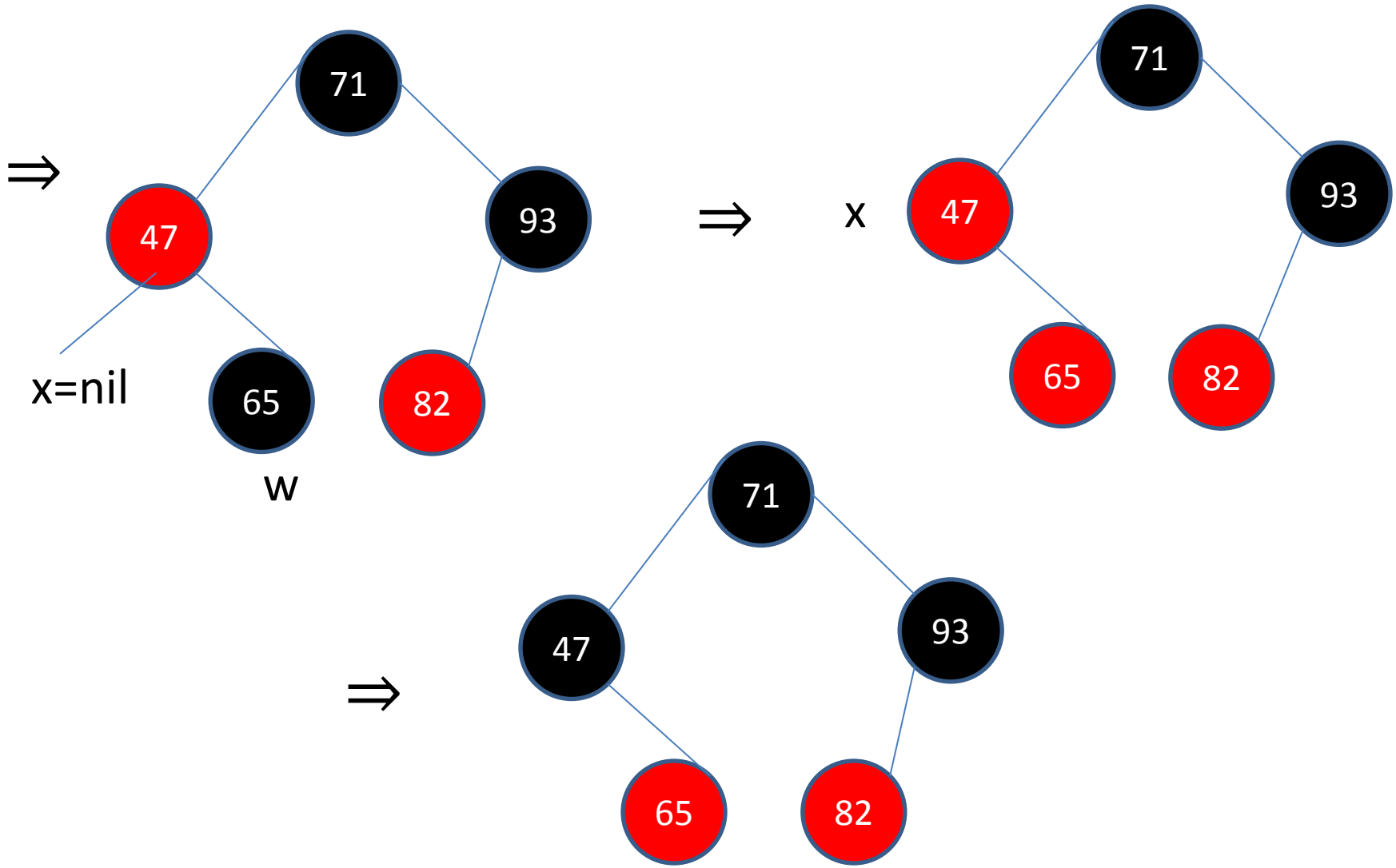
Deletion operation

Deletion of 32



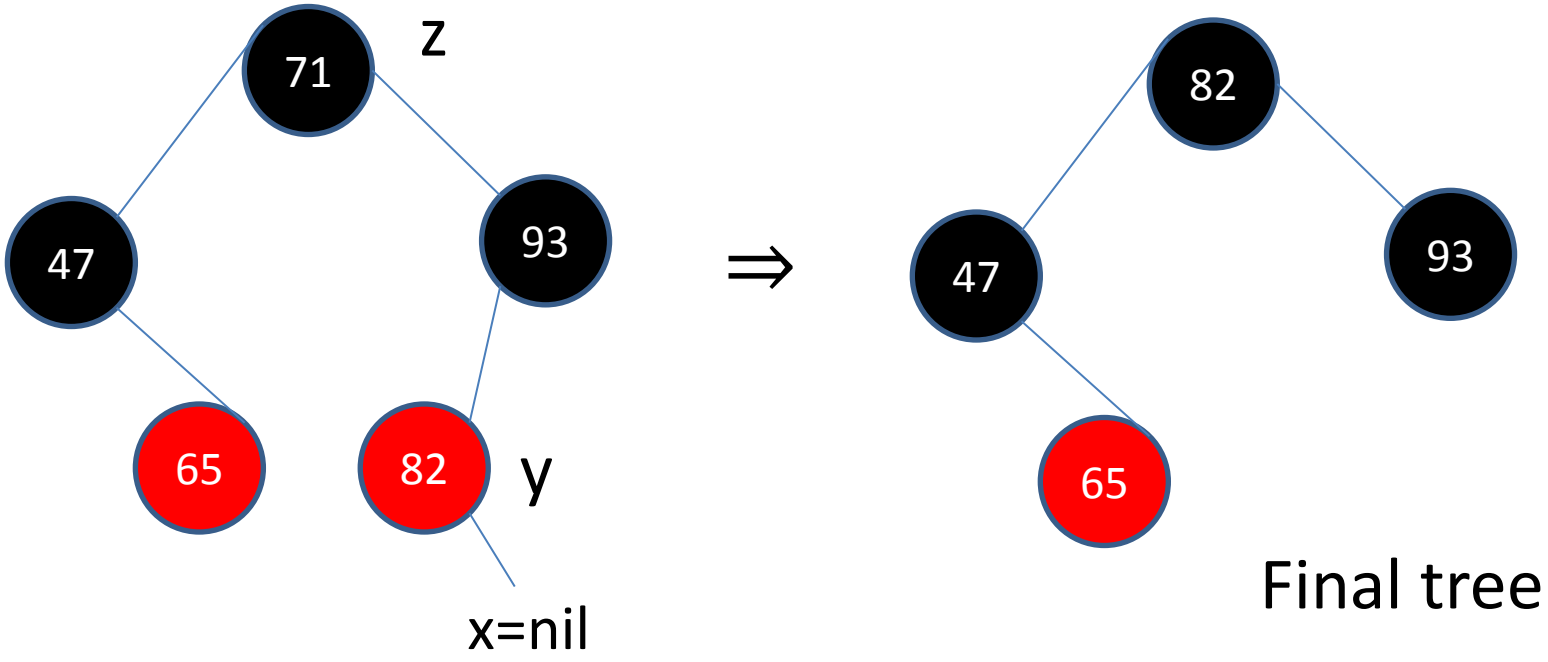
Deletion operation

Deletion of 32(continue)



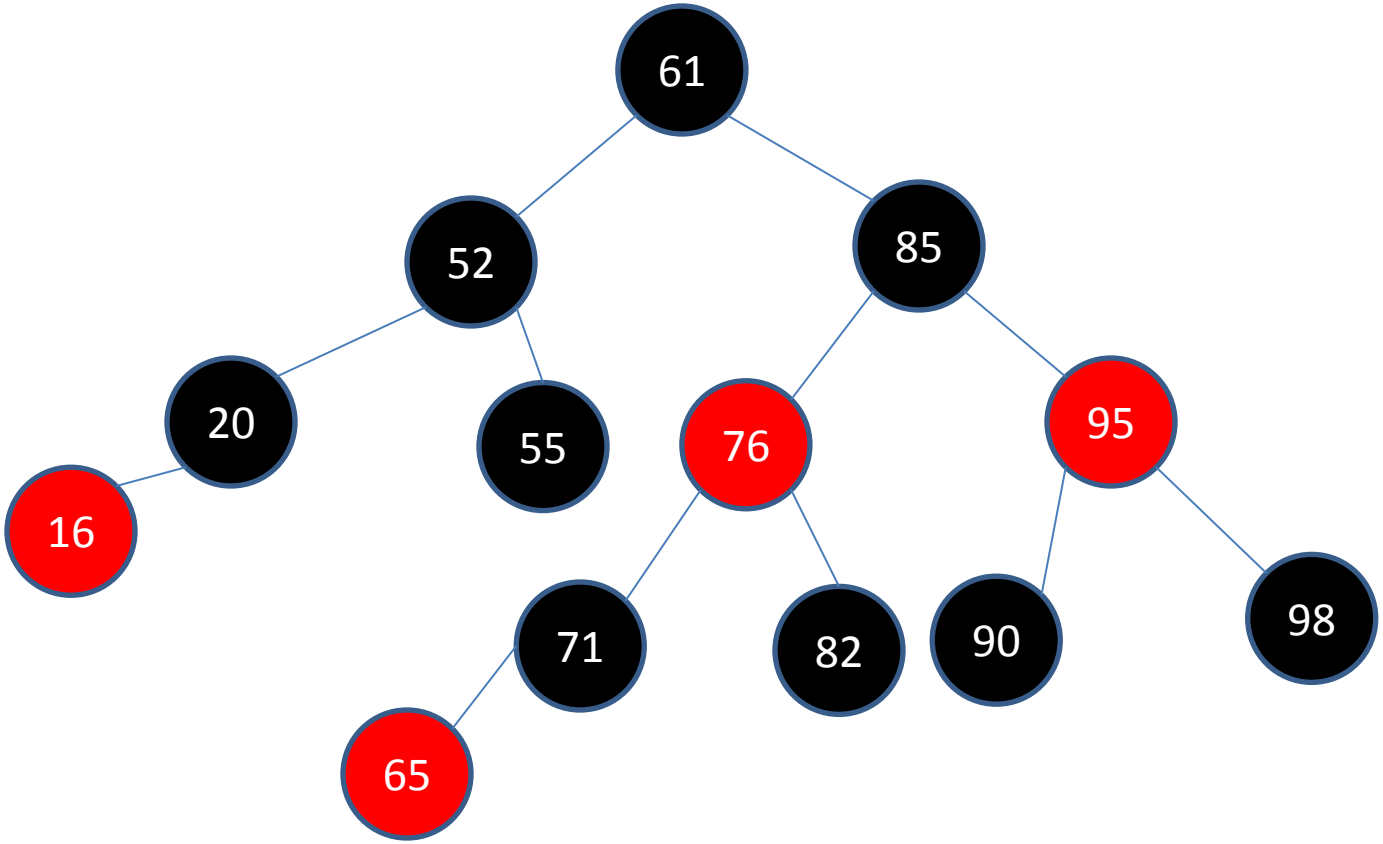
Deletion operation

Deletion of 71



Deletion operation

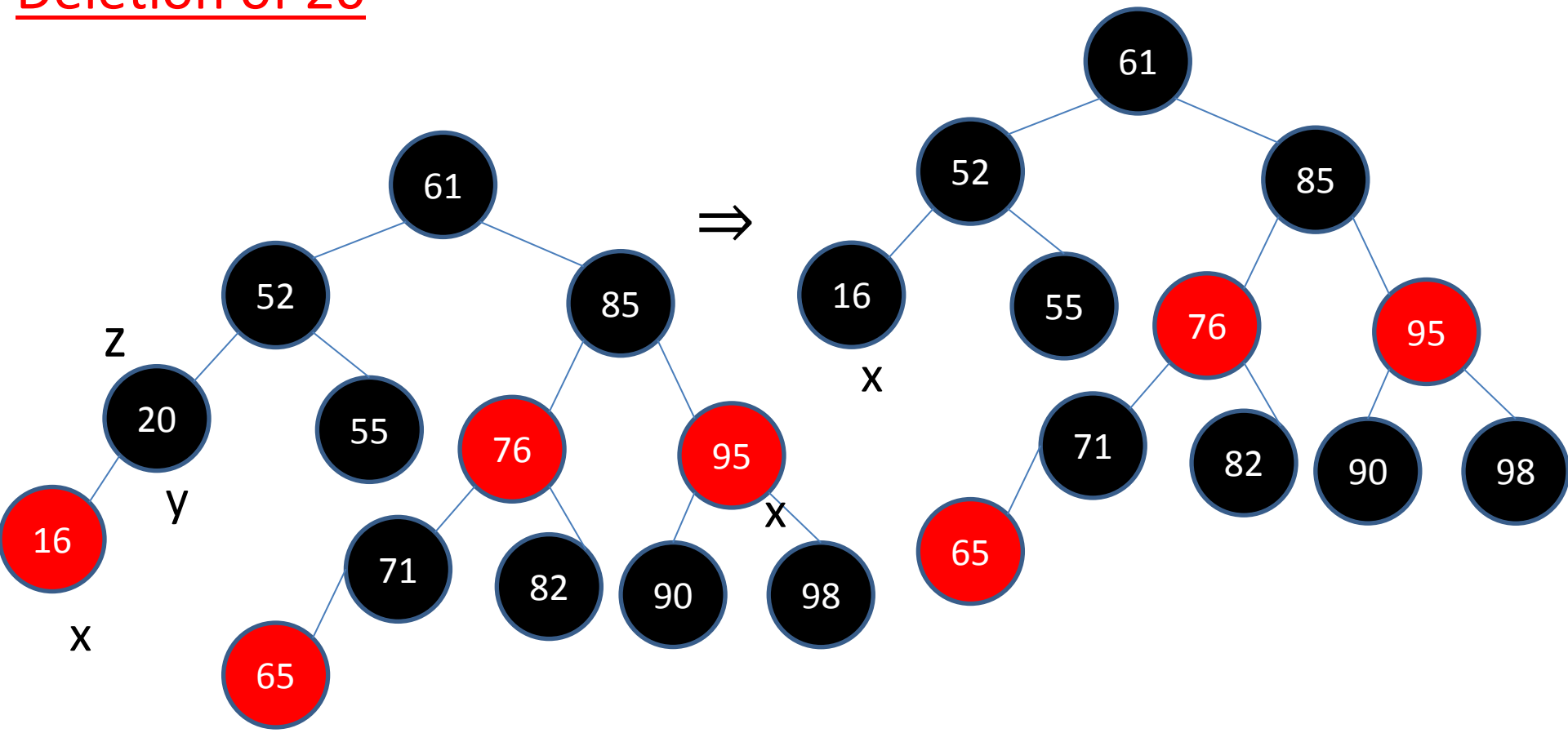
Example: Consider the following red-black tree



Delete the element 20, 85, 95 and 98 in order.

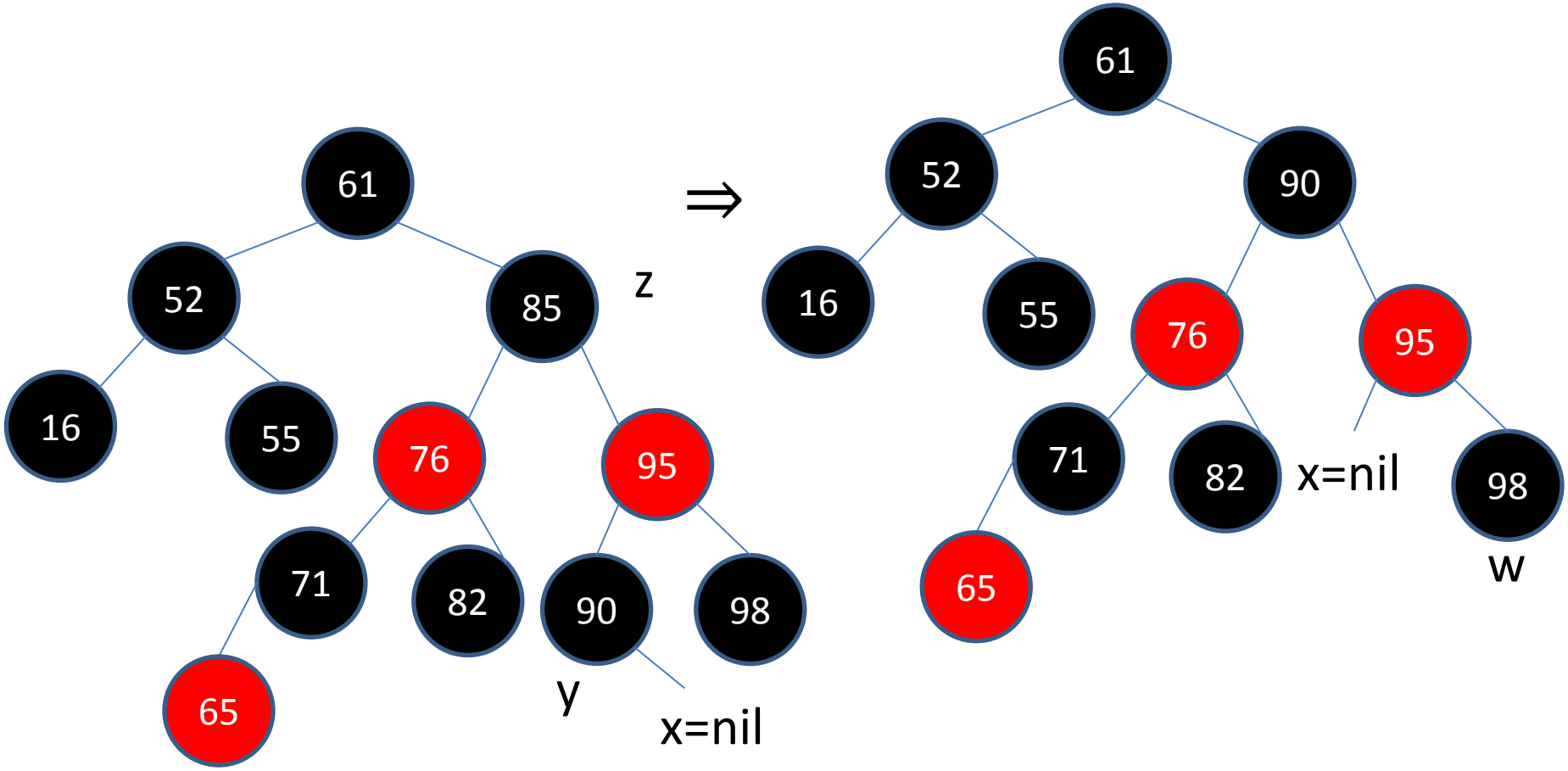
Deletion operation

Deletion of 20



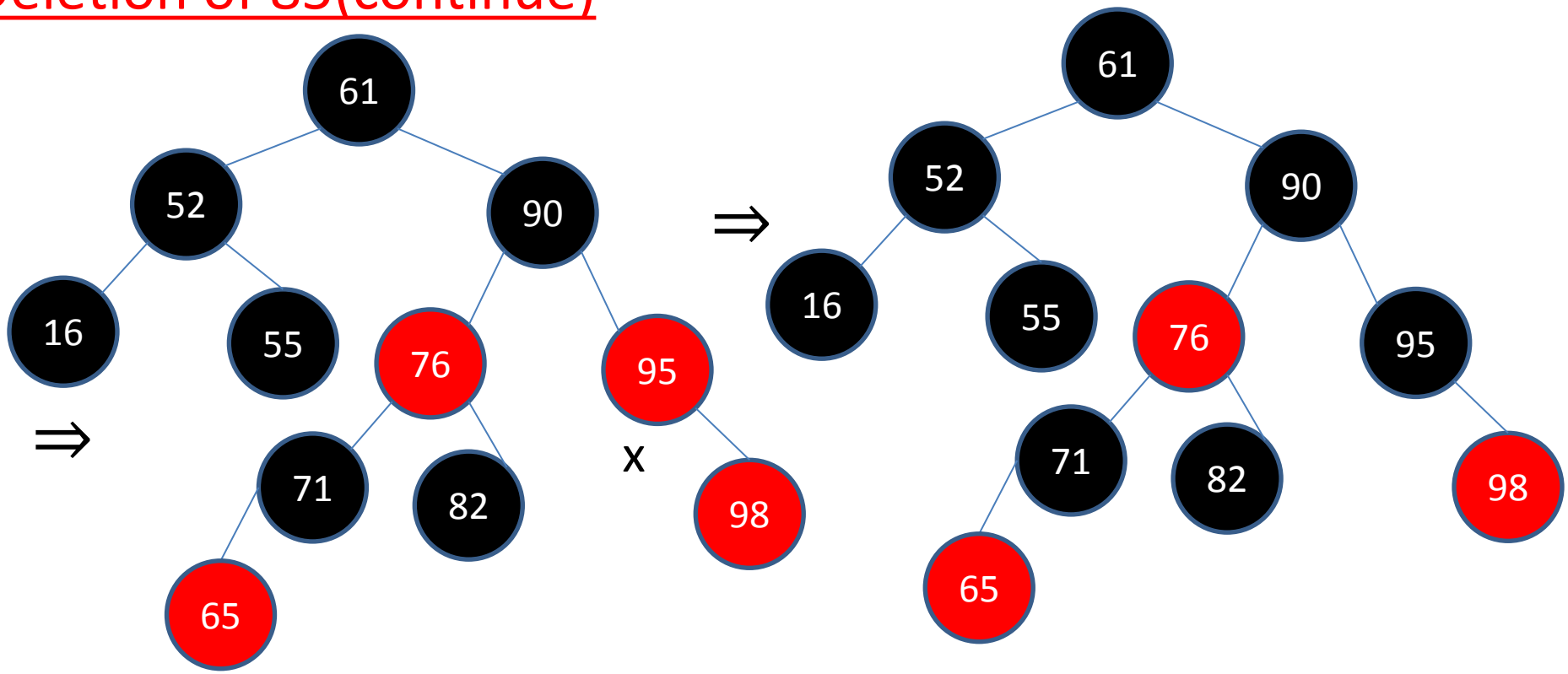
Deletion operation

Deletion of 85



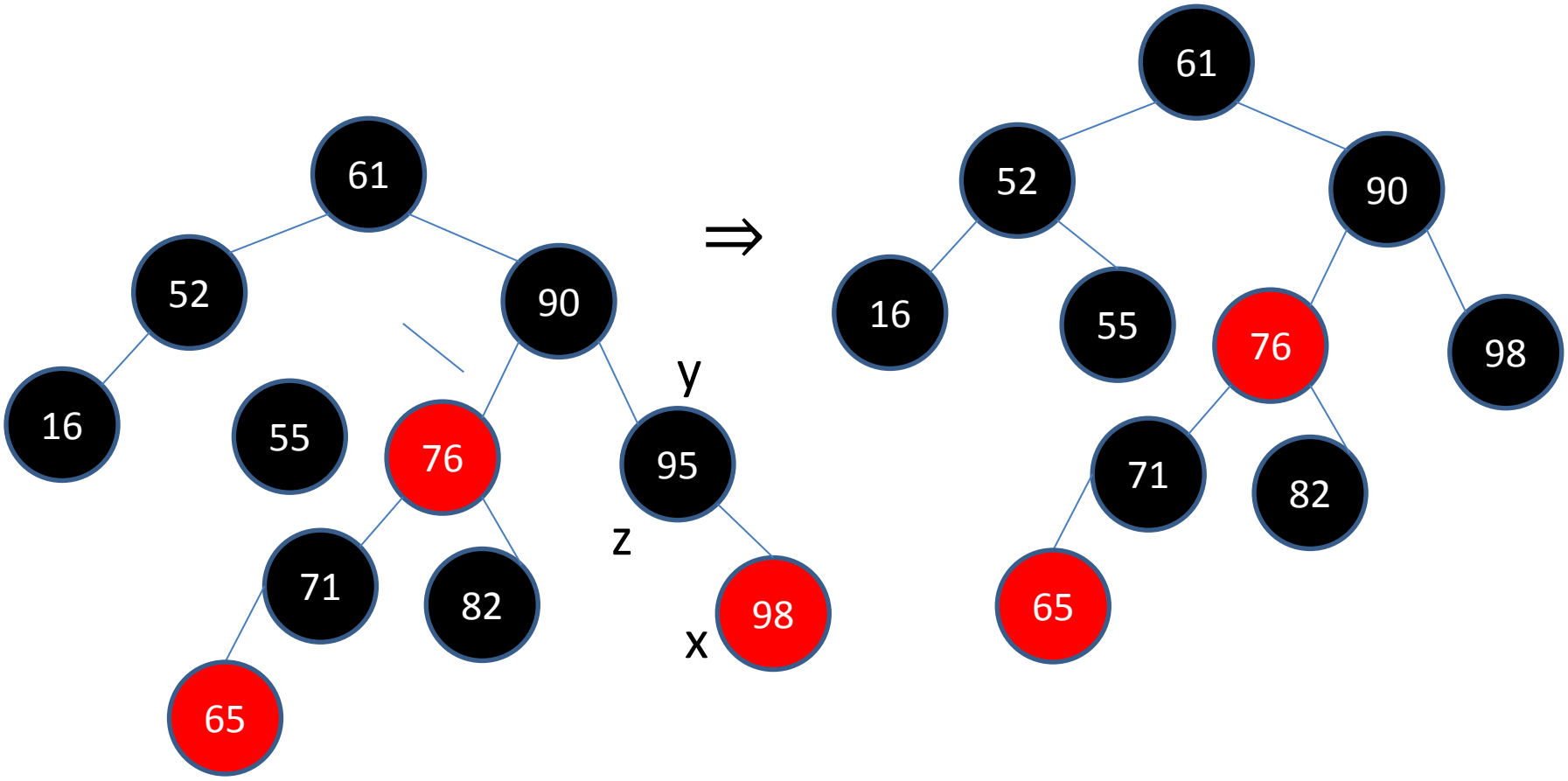
Deletion operation

Deletion of 85(continue)



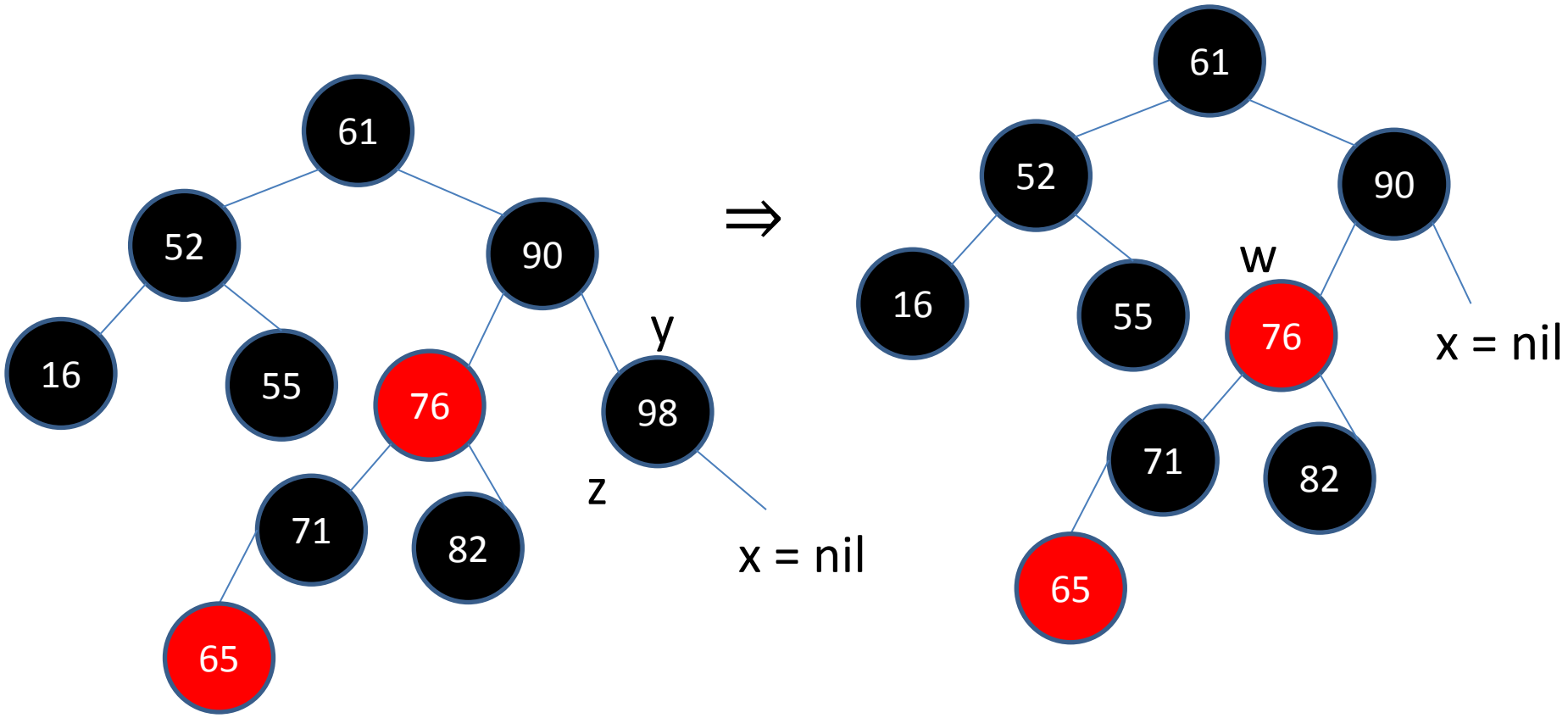
Deletion operation

Deletion of 95



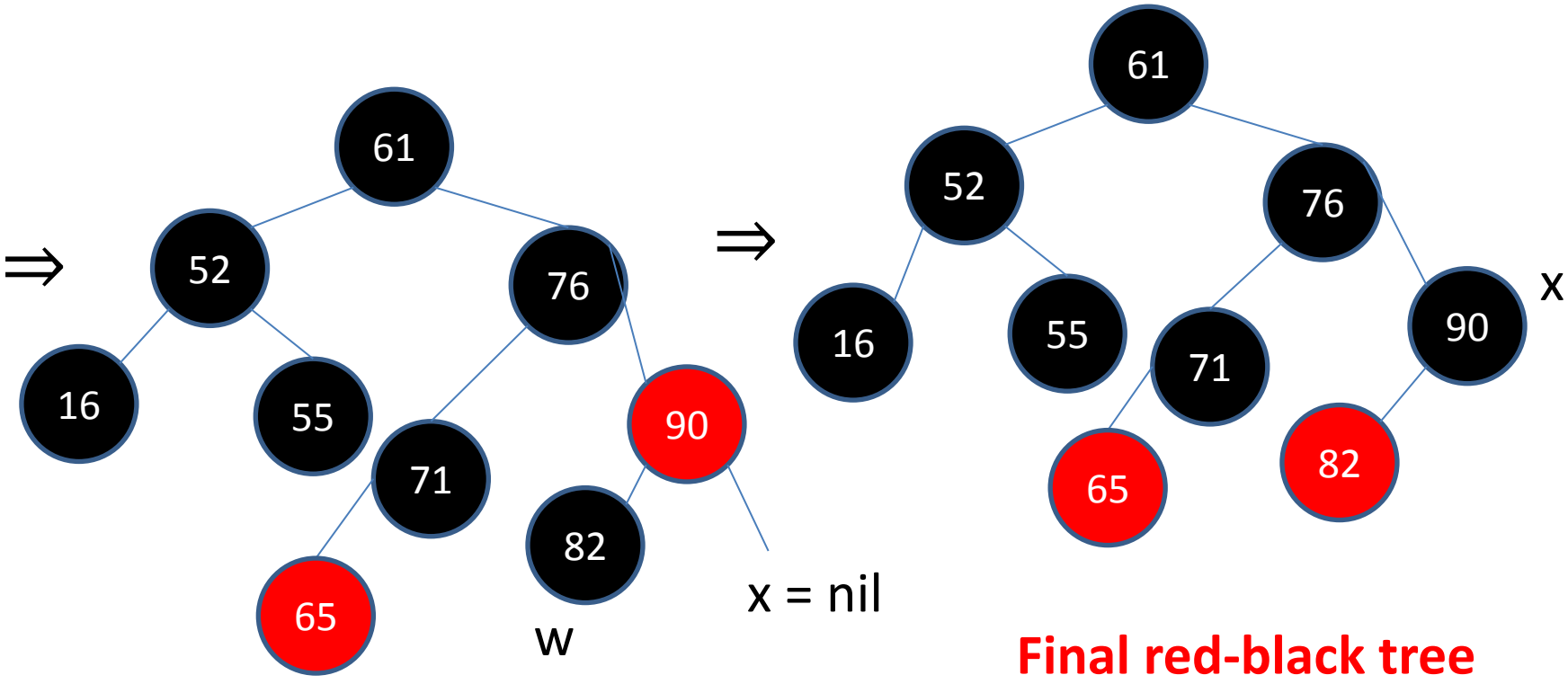
Deletion operation

Deletion of 98



Deletion operation

Deletion of 98(continue)



Final red-black tree

AKTU exam. questions

1. Insert the elements 8, 20, 11, 14, 9, 4, 12 in a Red-Black Tree and delete 12, 4, 9, 14 respectively.
2. What is Red-Black tree? Write an algorithm to insert a node in an empty red-black tree explain with suitable example.
3. Insert the following element in an initially empty RB-Tree. 12, 9, 81, 76, 23, 43, 65, 88, 76, 32, 54. Now Delete 23 and 81.
4. Write the properties of Red-Black Tree. Illustrate with an example, how the keys are inserted in an empty red-black tree.

Thank You.