Red Black Trees

Data Structures

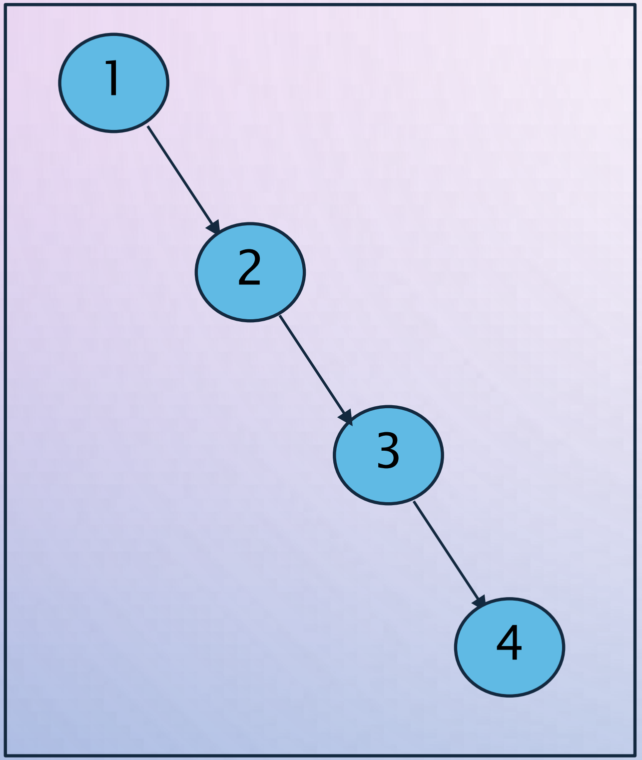
Some Properties of Binary Search Tree

- The common properties of binary search trees are as follows:
  - The left sub tree of a node contains only nodes with keys less than the node's key.
  - The right sub tree of a node contains only nodes with keys greater than the node's key.
  - The left and right sub tree each must also be a binary search tree.
  - There must be no duplicate nodes.
  - A unique path exists from the root to every other node.

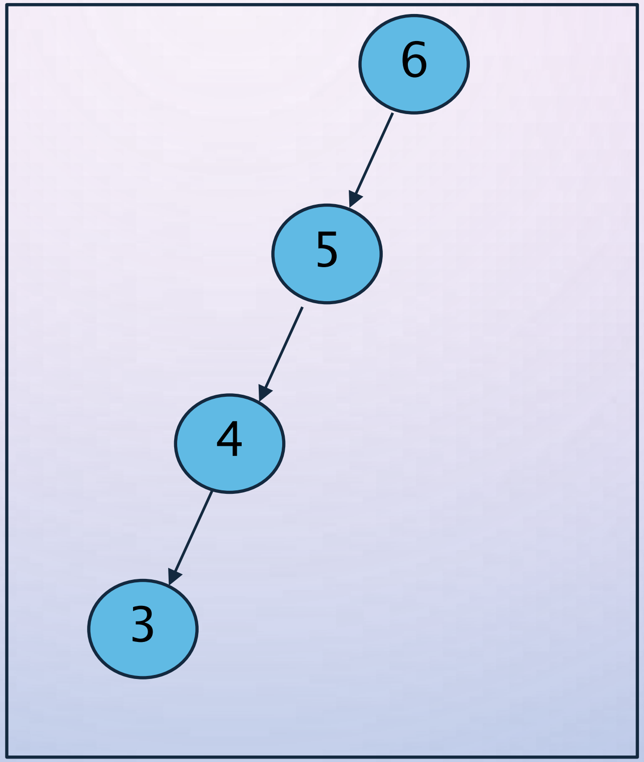
Problem with Binary Search Tree

- Binary Search Tree is fast in insertion and deletion etc. when balanced.
- Time Complexity of performing operations (e.g. searching , inserting , deletion etc) on binary search tree is O(logn) in best case i.e when tree is balanced.
- And on the other hand performance degrades from O(logn) to O(n) when tree is not balanced.
- Basic binary search trees have three very nasty degenerate cases where the structure stops being logarithmic and becomes a glorified linked list.
- The two most common of these degenerate cases is ascending or descending sorted order (the third is outside-in alternating order).

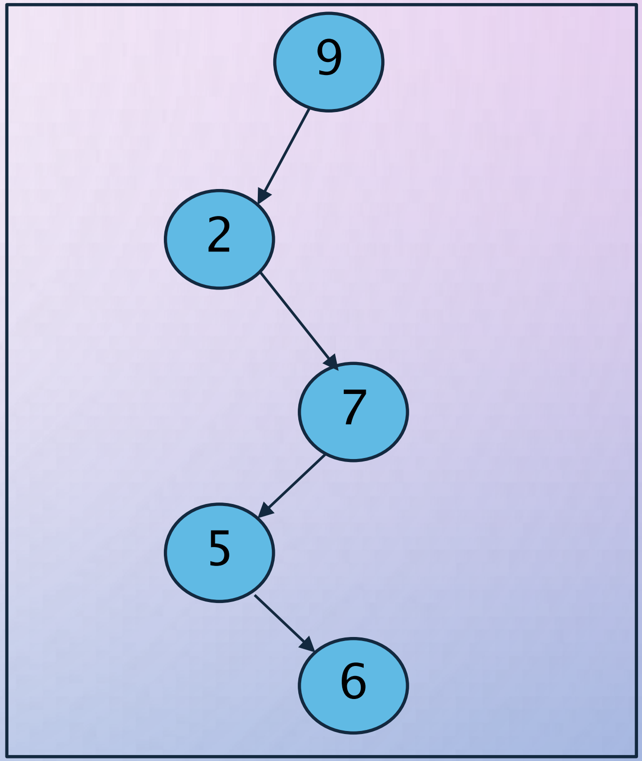
Problem with Binary Search Tree



Ascending order



Descending Order



Alternating Order

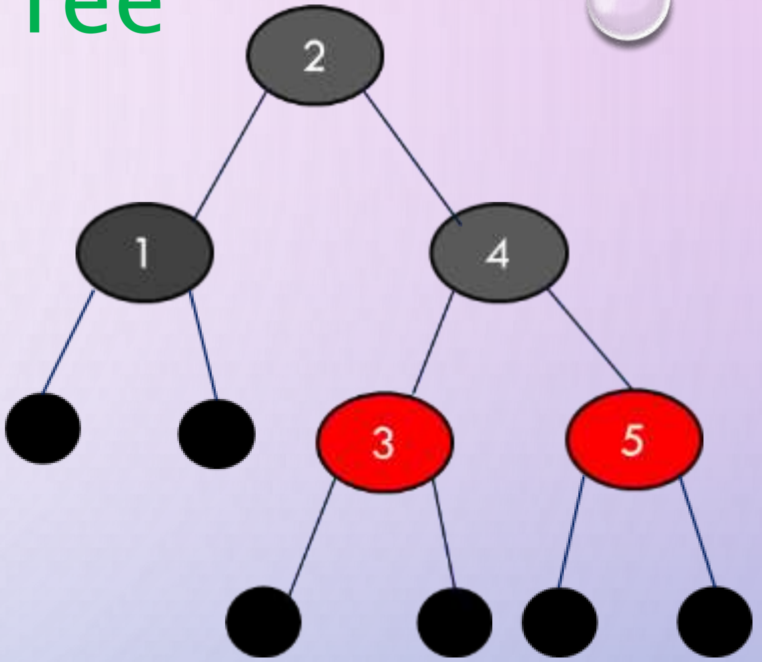
Red Black Tree

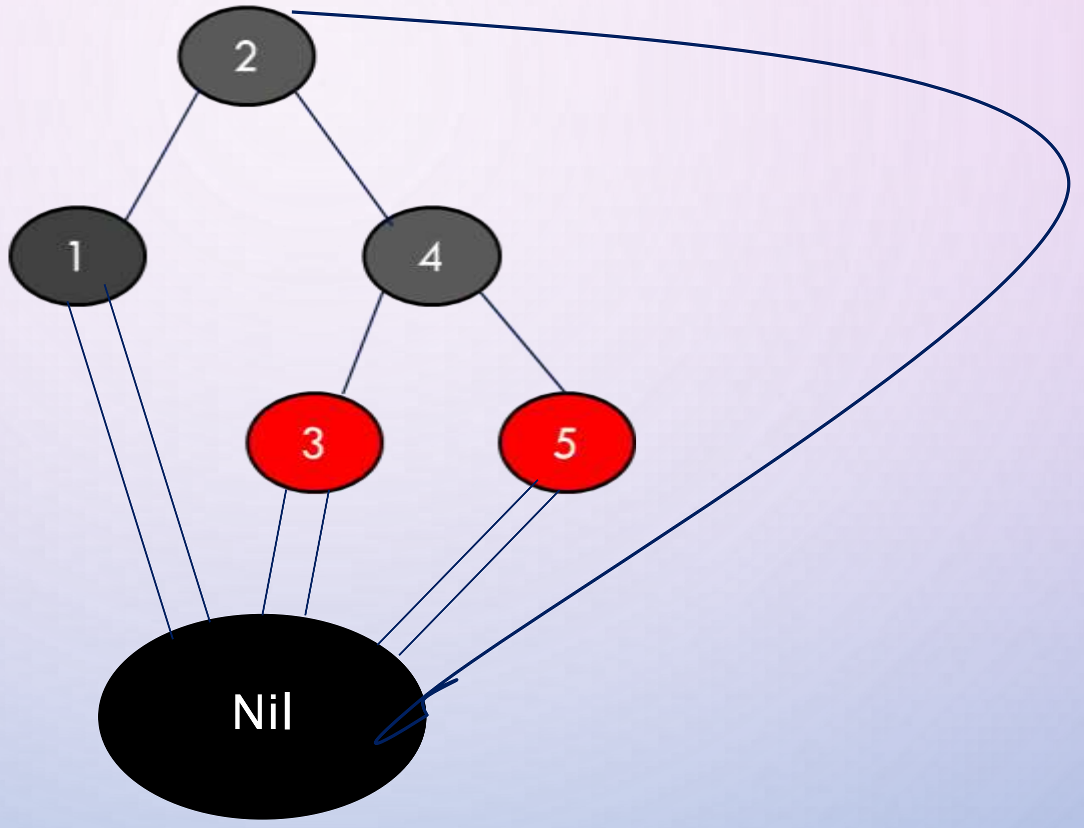
- A red-black tree is a balanced binary search tree with one extra bit of storage per node: its color, which can be either **Red** or **Black**.
- By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.
- Each node of the tree now contains the attributes color, key, left, right, and p.
- If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL.

Properties of Red Black Tree

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

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
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.




2
1
4
3
5
Nil

Red Black Tree (Rotations)

- The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red black tree with n keys, take O(log n)time. Because they modify the tree, the result may violate the red-black properties.

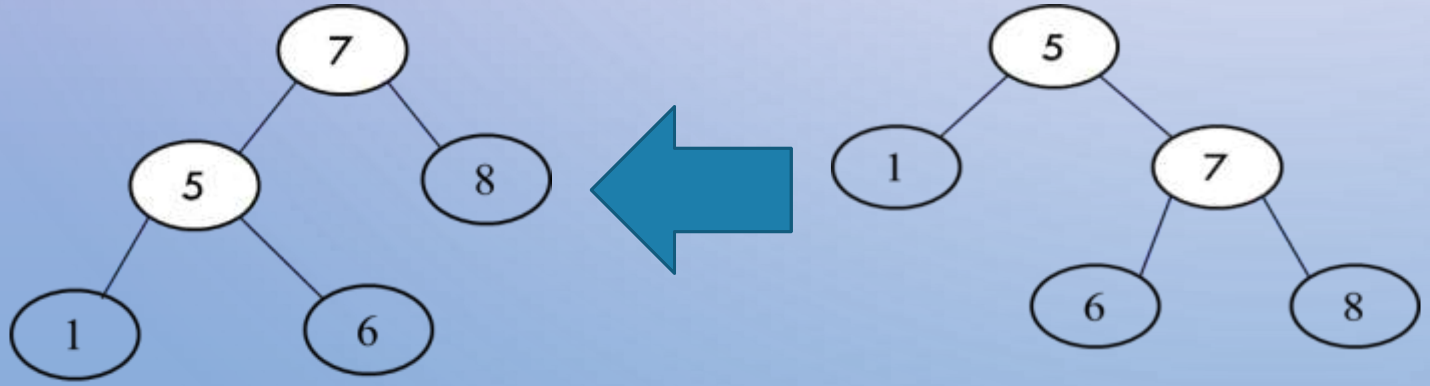
- To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

- We change the pointer structure through rotation, which is a local operation in a search tree that preserves the binary-search-tree property.

- There are two kinds of rotations : left rotations and right rotations.

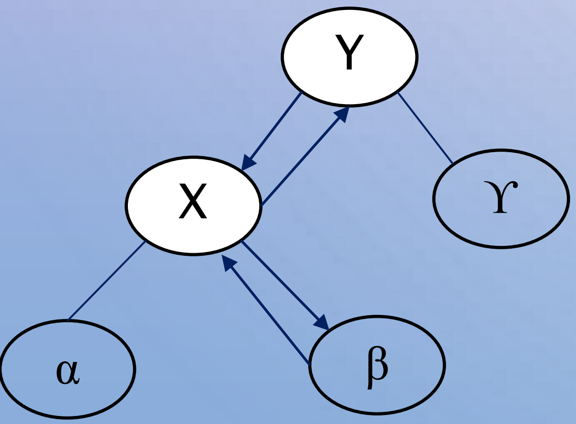
Left Rotation

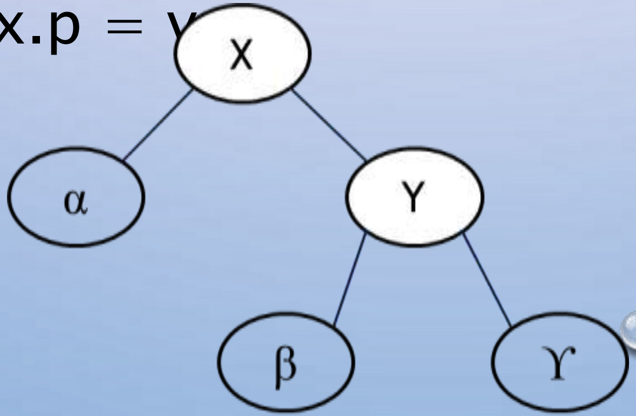
- When we do a left rotation on a node x, we assume that its right child y is not null ;x may be any node in the tree whose right child is not null.



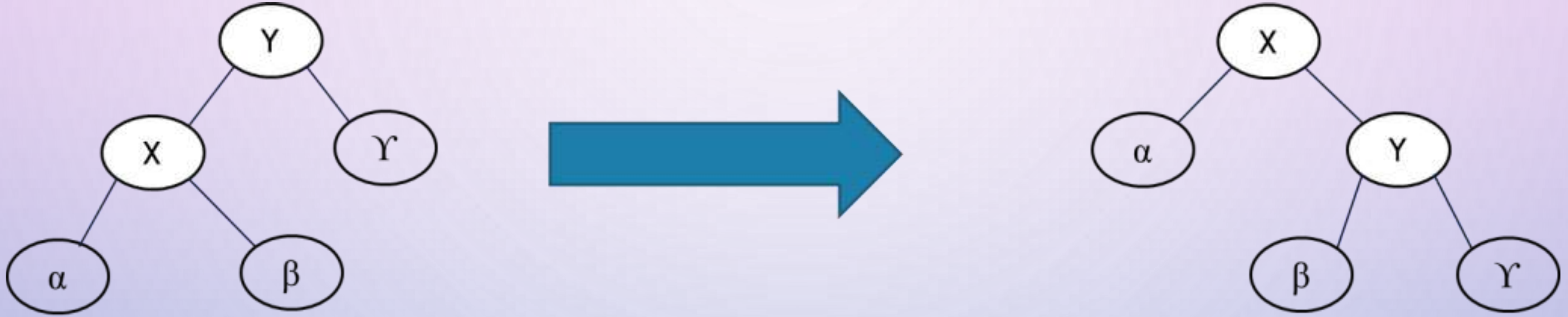
LEFT-ROTATE (T , x)

```
y = x.right                // set y new temporary node
x.right =  y.left          // turn y's left subtree into x's right subtree
if y.left != T.nil
      y.left.p = x
y.p= x.p                   // link x's parent to y
if x.p == T. nil
      T.root = y
else if x == x.p.left
      x.p.left = y
else x.p.right = y
y.left = x                 // put x on y's left
x.p = y
```

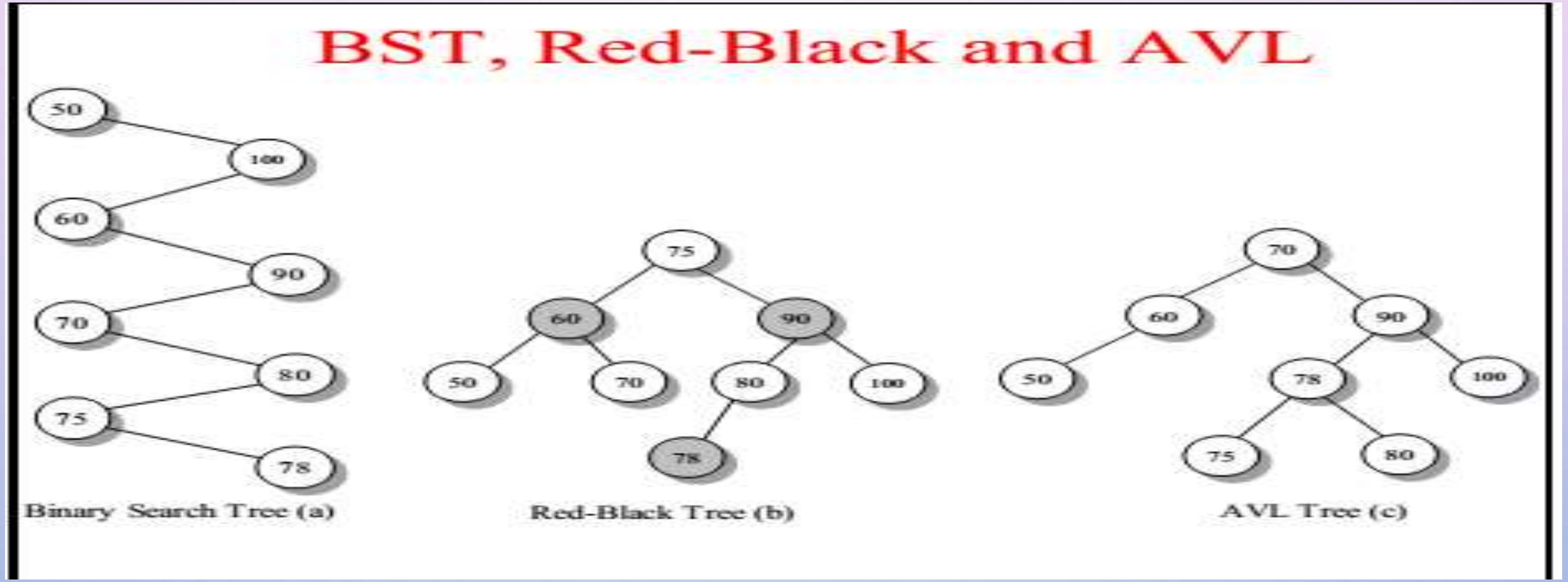




Right Rotation



Comparison


BST, Red-Black and AVL
50
100
60
90
70
80
75
78
Binary Search Tree (a)
75
60
90
50
70
80
100
78
Red-Black Tree (b)
70
60
90
50
78
100
75
80
AVL Tree (c)

Insertion

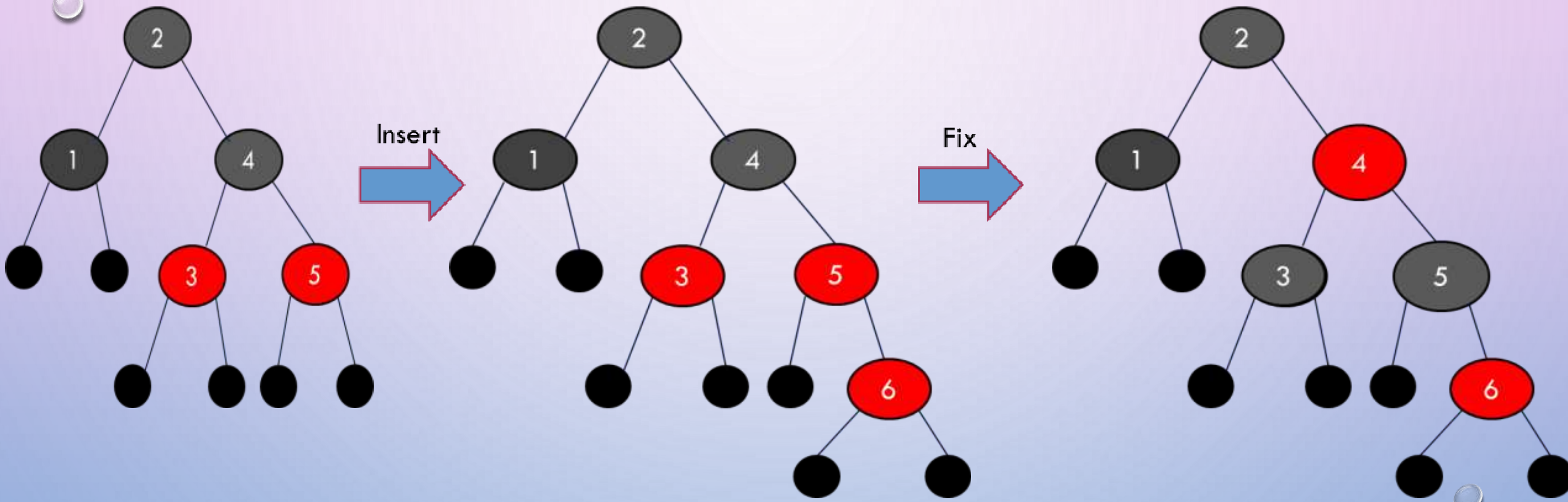
- We can insert a node into an n-node red-black tree in O(logn) time.

- To do so, we use a slightly modified version of the TREE-INSERT procedure to insert node into the tree T as if it were an ordinary binary search tree.

- Then we color it to red.

- To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations.

Insertion



Insertion

RB-INSERT(T, z)

```
y ← T.nil
x ← T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        then x ← x.left
    else x ← x.right
z.p = y
if y = T.nil
    then T.root ← z
else if z.key< y.key
    then y.left ← z
else y.right ← z
    z.left ← T.nil
    z.right ← T.nil
    z.color← RED
RB-INSERT-FIXUP(T, z)
```

Insertion

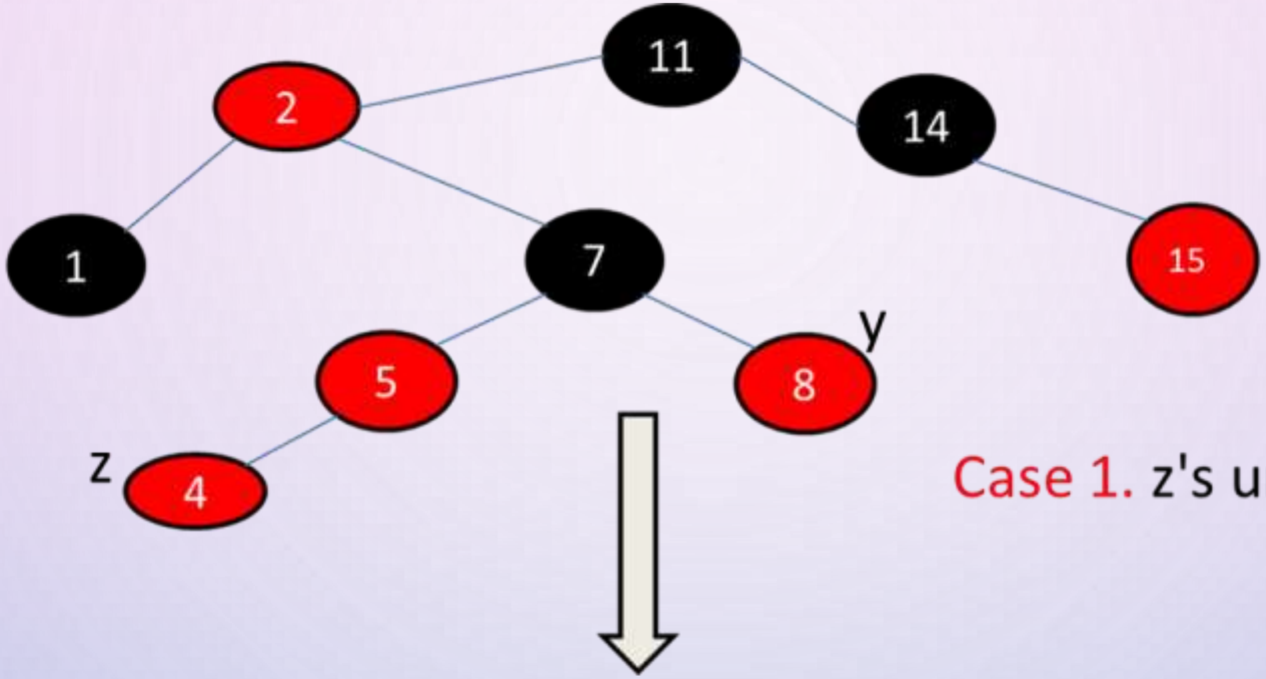
The procedures TREE-INSERT and RB-INSERT differ in four ways.

- First, all instances of NIL in TREE-INSERT are replaced by T.nil.
- Second, we set z.left and z.right to T. nil in lines 14–15 of RB-INSERT, in order to maintain the proper tree structure.
- Third, we color z red in line 16.
- Fourth, because coloring z.red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T,z) in line 17 of RB-INSERT to restore the red-black properties

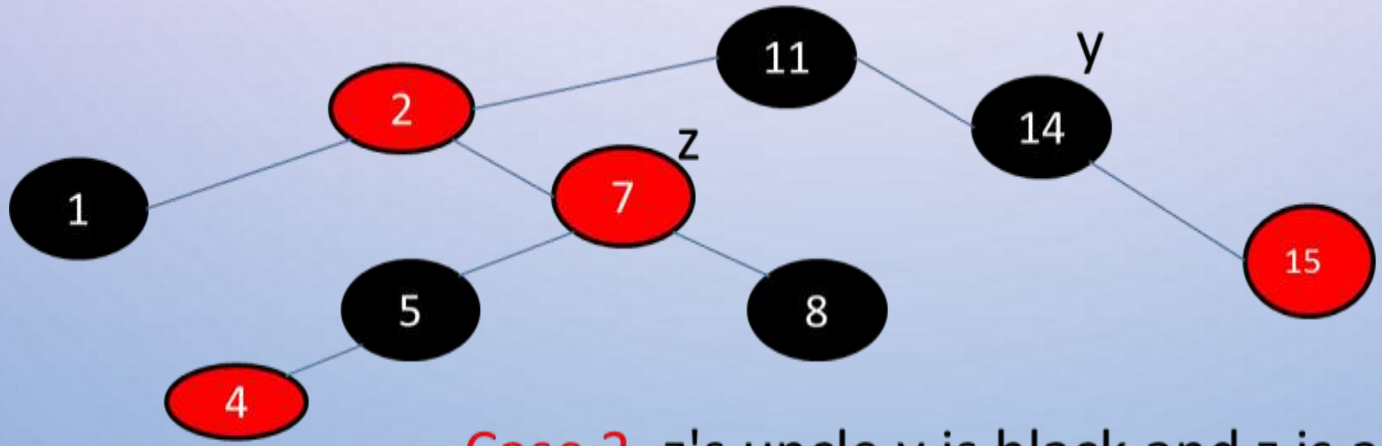
Insertion

- Case 1. z's uncle y is red
- Case 2. z's uncle y is black and z is a right child
- Case 3. z's uncle y is black and z is a left child

Insertion

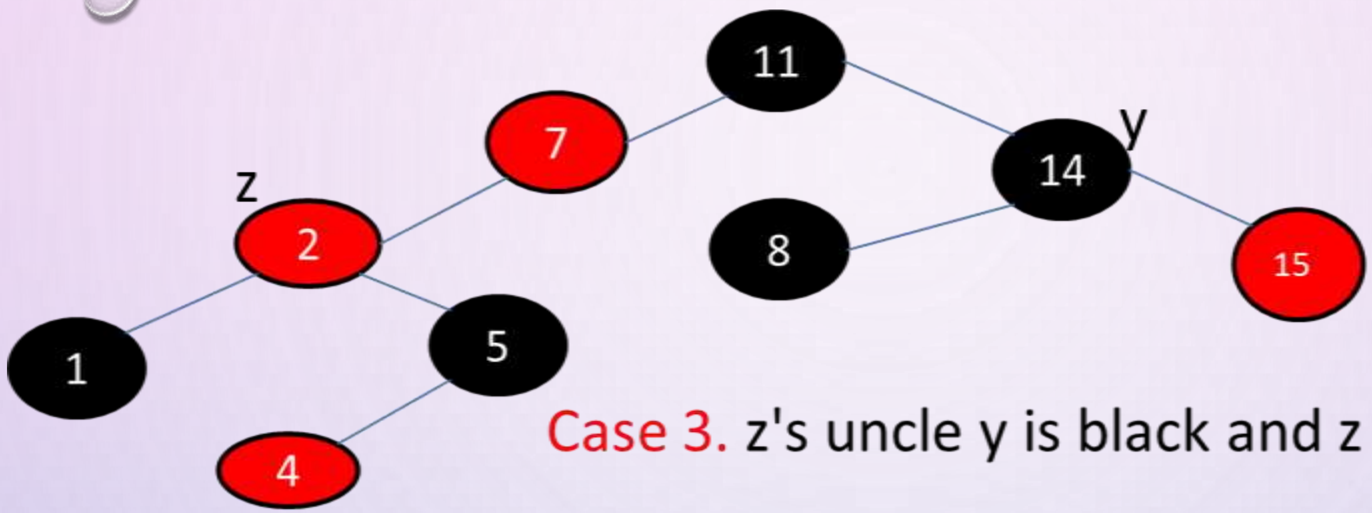


Case 1. z's uncle y is red

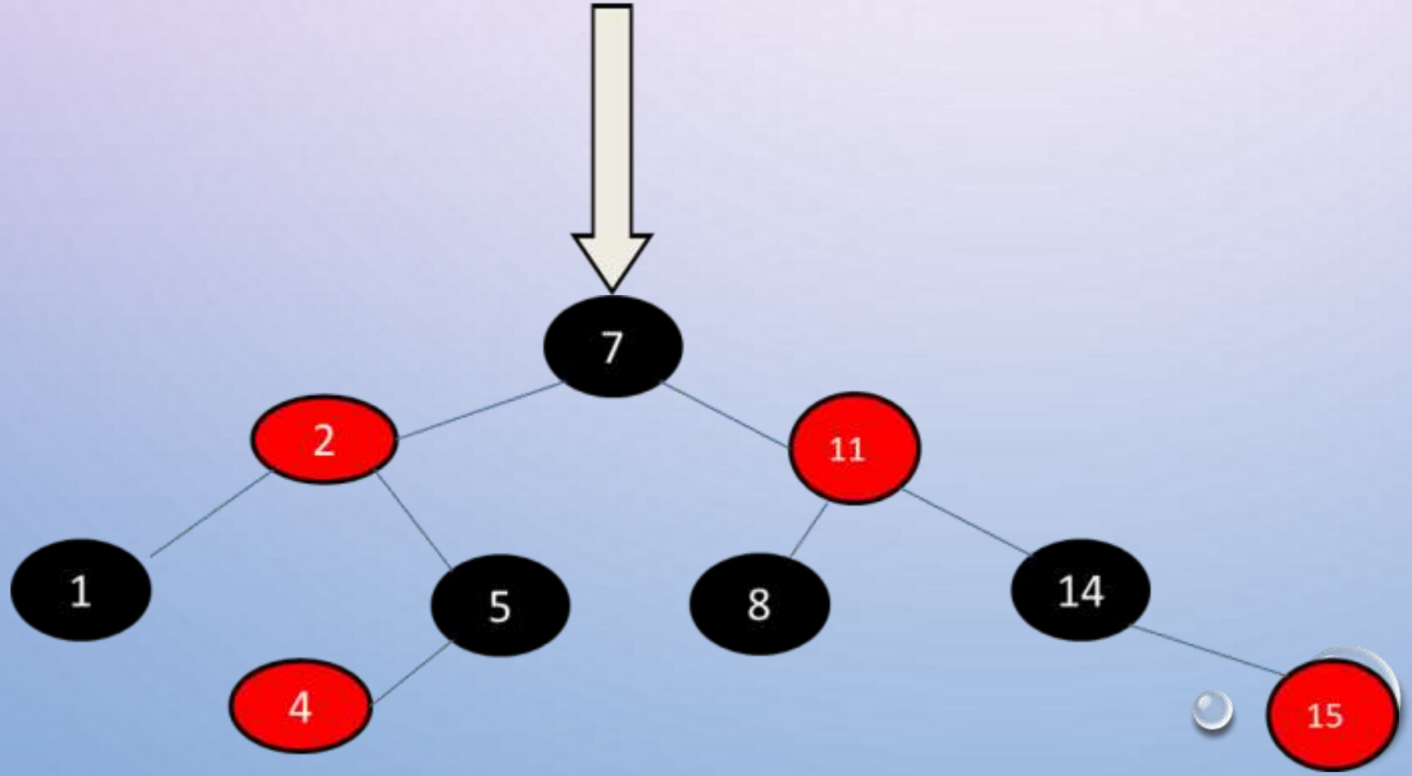


Case 2. z's uncle y is black and z is a right child

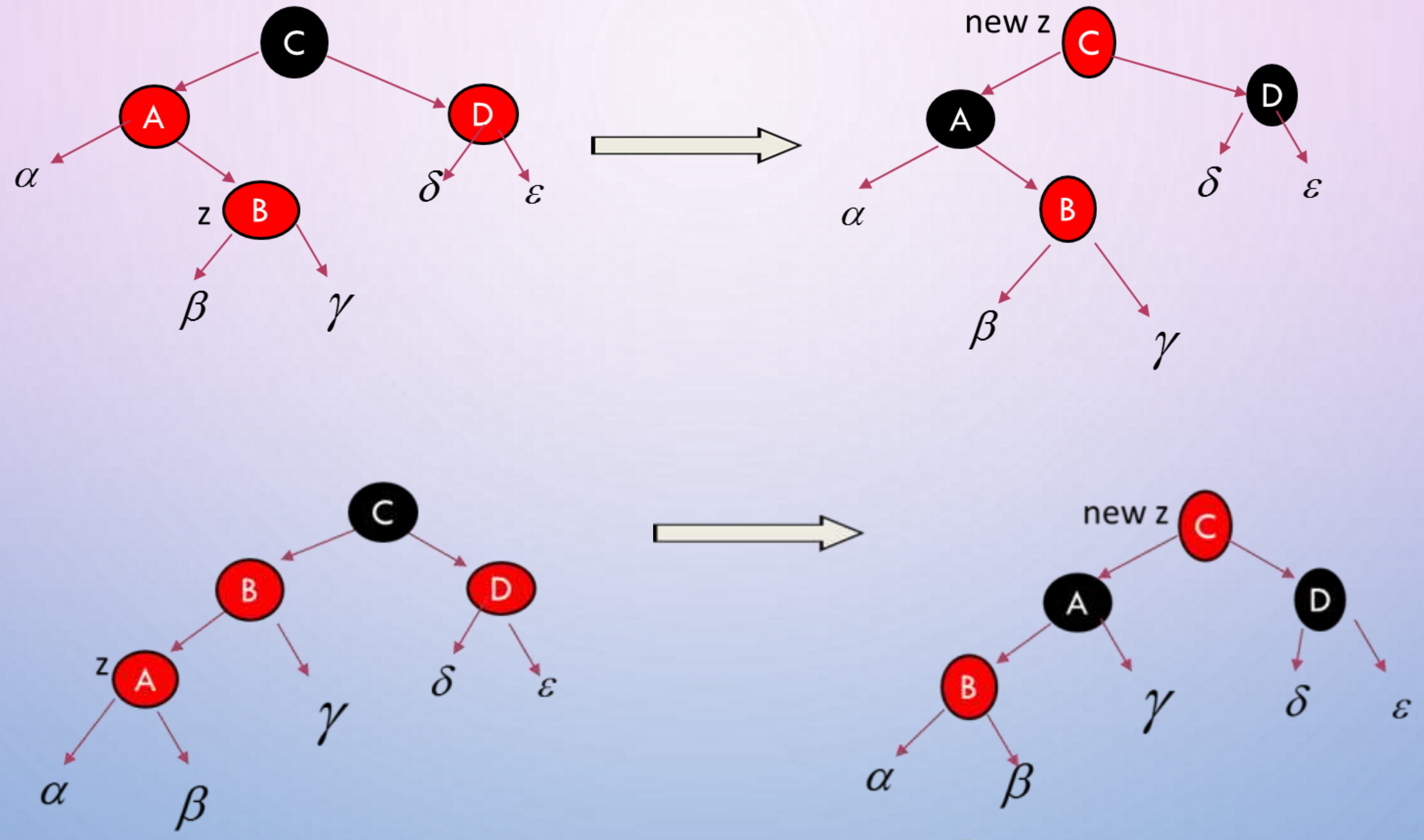
Insertion



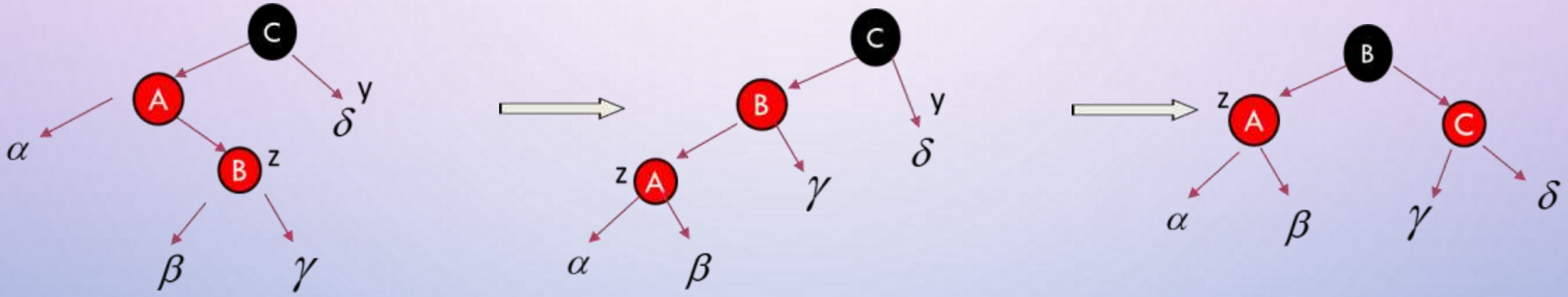
Case 3. z's uncle y is black and z is a left child



Insertion



Insertion



Case2: z's uncle y is black and z is a right child

Case3: z's uncle y is black and z is a left child

Insertion

RB-INSERT-FIXUP(T, z)

```
while z.p.color == RED
        if z.p == z.p.p.left
                then y ← z.p.p.right
                if y.color== RED
                        then z.p.color← BLACK            //Case 1
                        y.color ← BLACK                  //Case 1
                        z.p.p.color← RED                 //Case 1
                        z ← z.p.p                        // Case 1
                else if z = z.p.right
                        then z ← z.p                     //Case 2
                        LEFT-ROTATE(T, z)                //Case 2
                z.p .color← BLACK                        //Case 3
                z.p .p.color← RED                        //Case 3
                RIGHT-ROTATE(T, z.p .p)                  // Case 3
        else .same as then clause with "right" an= "left" exchange=)
T.root.color← BLACK
```

Insertion

- To understand how RB-INSERT-FIXUP works, we shall break our examination of the code into three major steps.

- First, we shall determine what violations of the red-black properties are introduced in RB-INSERT when node z is inserted and colored red.

- Second, we shall examine the overall goal of the while loop in lines 1–15.

- Finally, we shall explore each of the three cases within the while loop's body and see how they accomplish the goal.

Insertion

- The while loop in lines 1–15 maintains the following three-part invariant.
- At the start of each iteration of the loop,
  a) Node z is red.

  b) If z.p is the root, then z.p is black.

  c) If there is a violation of the red-black properties , there is at most one violation, and it is of either property 2 or property 4. If there is a violation of property 2, it occurs because z is the root and is red. If there is a violation of property 4, it occurs because both z and z.p are red.

Deletion

- Like the other basic operations on an Red Black tree, deletion of a node takes time O(logn)
- 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 RB-DELETE procedure
- First, we need to customize the TRANSPLANT subroutine that RB-DELETE calls so that it applies to a red-black tree.

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

Deletion

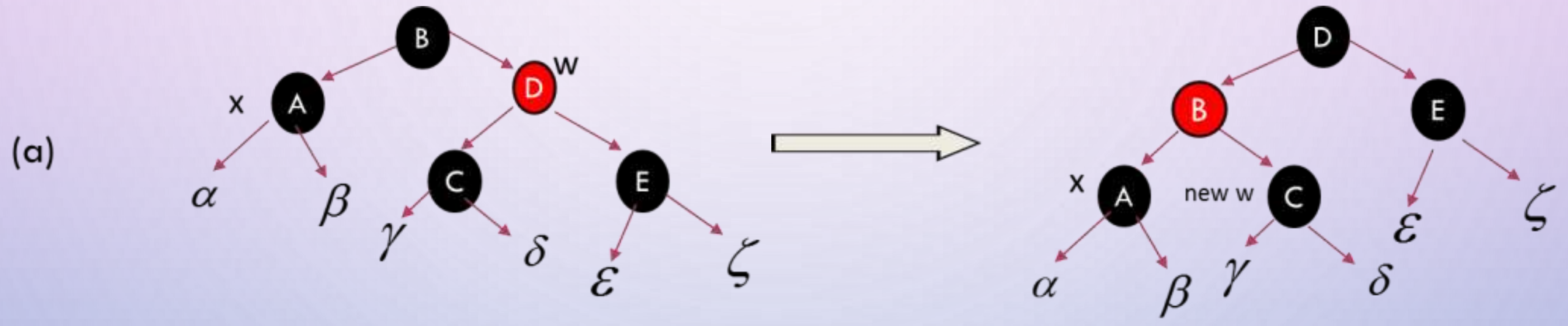
RB-DELETE (T,z)

```
y = z
y-original-color =  y.color
if z.left = = T. nil
        x = z. right
        RB-TRANSPLANT (T, z, z. right)
elseif z. right = = T. nil
        x = z. left
        RB-TRANSPLANT(T, z, z. left)
else y = TREE-MINIMUM(z. right)
        y-original-color = y. color
        x = y. right
        if y. p = = z
                x. p = y
```

Deletion

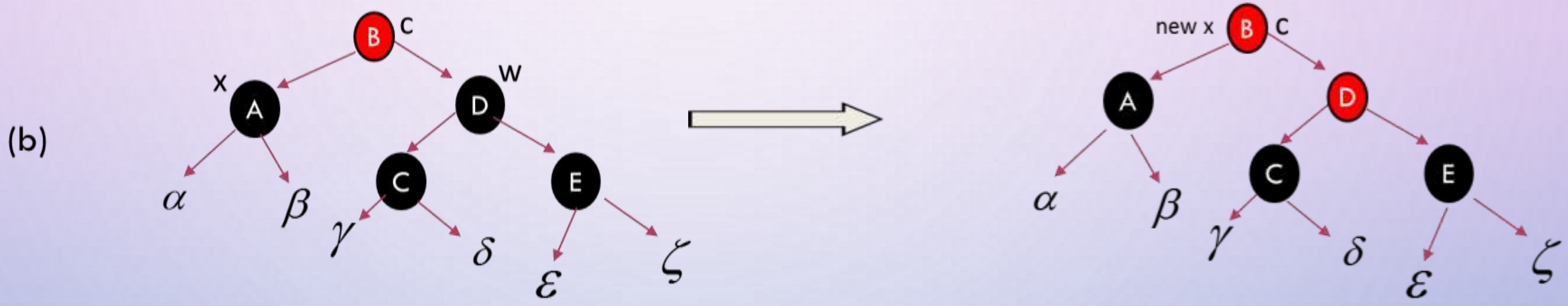
```
else RB-TRANSPLANT(T, y, y. right)
                y. right = z. right
                y. right. p = y
        RB-TRANSPLANT(T, z, y)
        y. left = z. left
        y. left. p = y
        y. color = z. color
if y-original-color == BLACK
        RB-DELETE-FIXUP(T, x)
```

Deletion



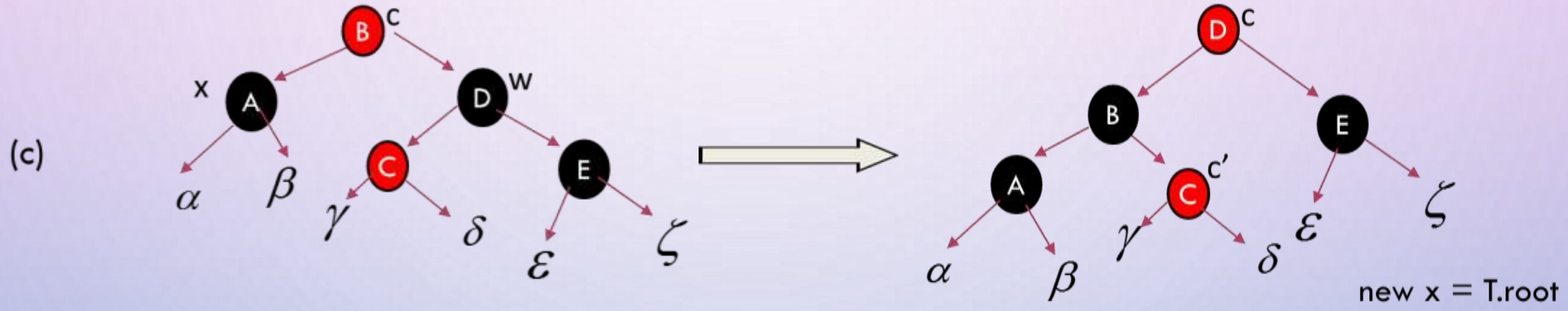
Case 1: x's sibling w is red

Deletion



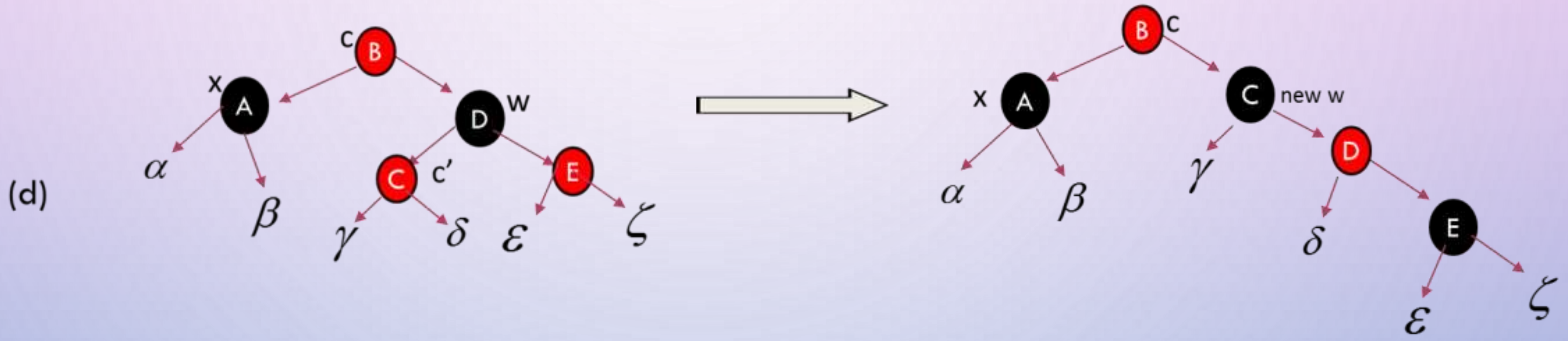
Case 2: x's sibling w is black, and both of w's children are black

Deletion



Case 3: x's sibling w is black, w's left child is red, and w's right child is black

Deletion



Case 4: x's sibling w is black, and w's right child is red

Deletion

RB-DELETE-FIXUP(T, x)

```
while x !=T.root an= x. color == BLACK
        if x == x. p. left
                w = x. p. right
                if w. color = = RED
                        w. color = BLACK                //case 1
                        x. p. color = RED               //case 1
                        LEFT-ROTATE(T, x. p)            //case 1
                        w = x. p. right                 //case 1
                if w. left. color == BLACK and w. right. color = = BLACK
                        w. color = RED          //case 2
                        x = x. p        //case 2
```

Deletion

```
        else if w. right. color == BLACK
                w. left. color = BLACK              //case 3
                w. color = RED                      //case 3
                RIGHT-ROTATE(T, w)                  //case 3
                w = x. p. right                     //case 3
            w. color = x. p. color                  //case 4
            x. p. color = BLACK                     //case 4
            w. right. color = BLACK                 //case 4
            LEFT-ROTATE(T, x. p)                    //case 4
            x = T. root                             //case 4
    else (same as then clause with “right” and “left” exchanged)
x. color = BLACK
```