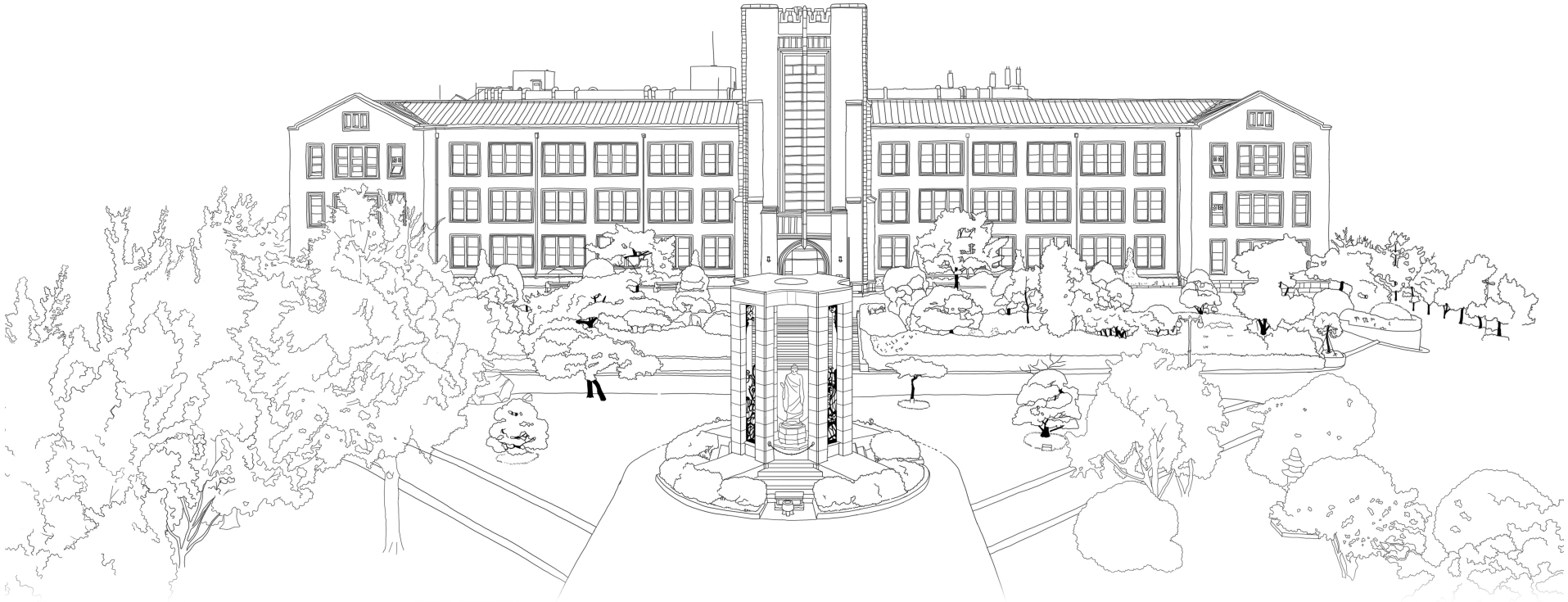


Introduction to Algorithms

DongYeon Kim
Department of Multimedia Engineering
Dongguk University

BigDataLab
Dept. of Multimedia Engineering at Dongguk University

dongguk UNIVERSITY

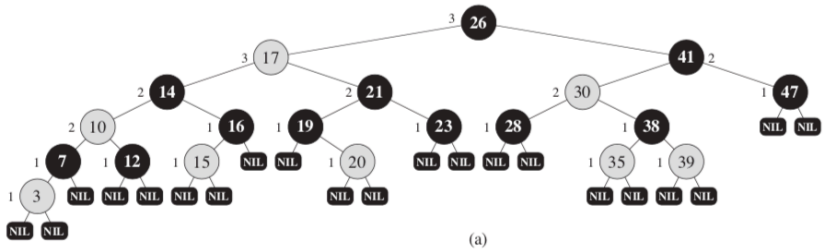
Contents

❖ Chapter.3 Data Structures

❖ Properties of Red-Black trees

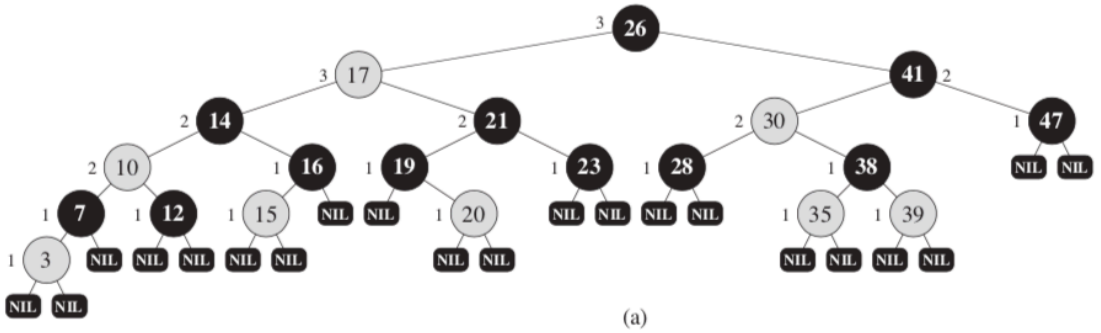
- **Introduction**
  - ➢ BST of height h supports any of the basic operations in $O(h)$ time
    - ✓ The operations are fast if the height of the tree is small
    - ✓ If the height is high, operations may tun no faster than linked list
  - ➢ Red-black trees are balanced search tree
    - ✓ Guarantee that basic operations take $O(\lg n)$ time in worst-case

- **Definition of Red-Black tree**
  - ➢ BST with one extra bit of storage per node : its color (Red or Black)
  - ➢ Constrain the node colors on any simple path from the root to a leaf
  - ➢ Ensure that no such path is more than twice as long as any other
    - ✓ The tree is approximately balanced



❖ Properties of Red-Black trees

- Definition of Red-Black tree



- Red-black tree properties

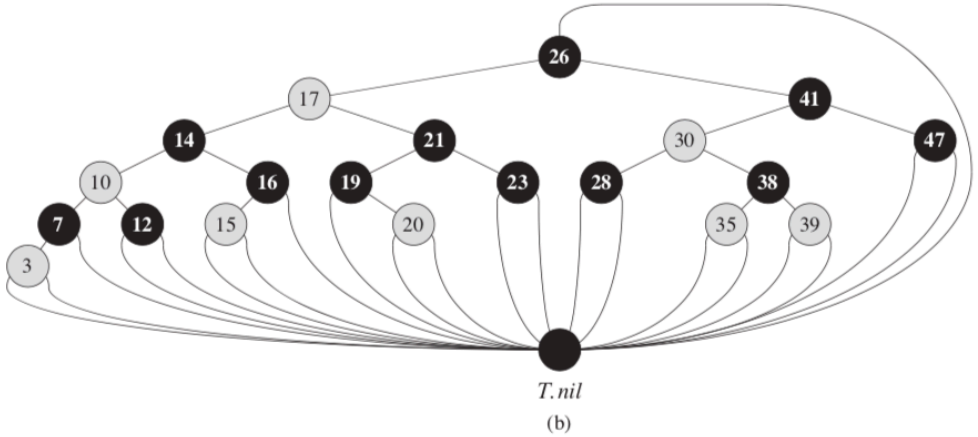
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

➢ Each node contains the attributes : $color$, key, $left, right$ and p

➢ Regard NIL as being pointers to leaves of the BST

❖ Properties of Red-Black trees

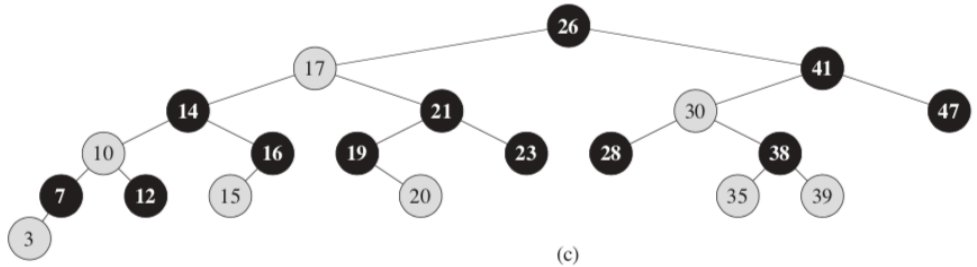
- Single sentinel Red-Black tree



(b)

- Use a single sentinel to represent NIL
- For a RB tree T, the sentinel $T.nil$ is an object
  - ✓ Represents same attributes as an ordinary node
  - ✓ Color is black
  - ✓ All pointers to NIL are replaced by $T.nil$
- Save the wasted space

- Simple Red-black tree

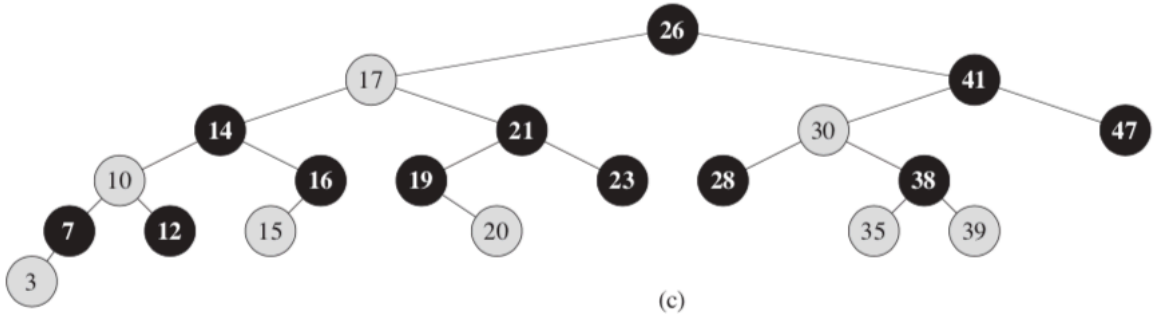


(c)

- Confine interest to the internal nodes of the tree
- leaves and the root's parent omitted entirely ($T.nil$)

❖ Properties of Red-Black trees

- Black-height : bh(x)



(c)

➢ The number of black nodes on any simple path from, but not including a node x (down to a leaf)

➢ All descending simple paths from the node have the same number of black nodes

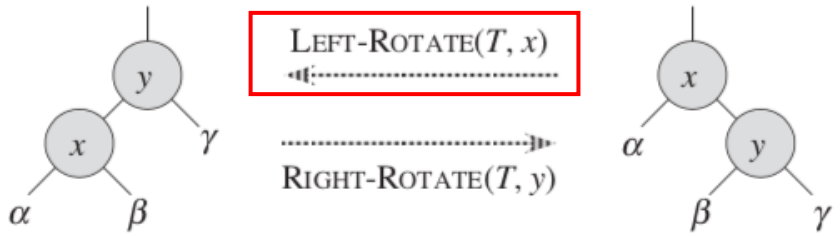
✓ Red-black tree property 5

➢ The black-height of a RB tree : the black height of the tree's root

❖ Rotation

- Introduction
  - ➢ The operation $\boldsymbol{TREE-INSERT}$, $\boldsymbol{TREE-DELETE}$ take $O(\lg n)$ time
    - ✓ modify the tree structure → the result may violate the RB tree properties
    - ✓ To restore these properties, must change the colors of nodes or change the pointer structure
  - ➢ Change the pointer structure through $\boldsymbol{rotation}$

- Definition
  - ➢ A local operation in a search tree to preserve the BST property

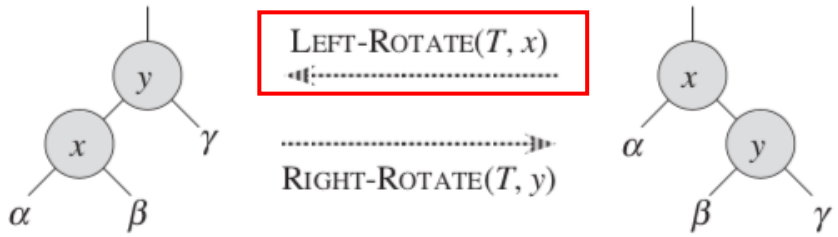


- ➢ Left rotation on a node x
  - ✓ Assume that right child y is not $T.nil$
  - ✓ Makes y the new root of the subtree, with x as y's left child and y's left child as x's right child

❖ Rotation

- Pseudocode
  - ➢ Assume that $x.right \neq T.nil$ and the root's parent is $T.nil$

```
LEFT-ROTATE(T, x)
1   y = x.right
2   x.right = y.left
3   if y.left ≠ T.nil
4       y.left.p = x
5   y.p = x.p
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
12  x.p = y
```

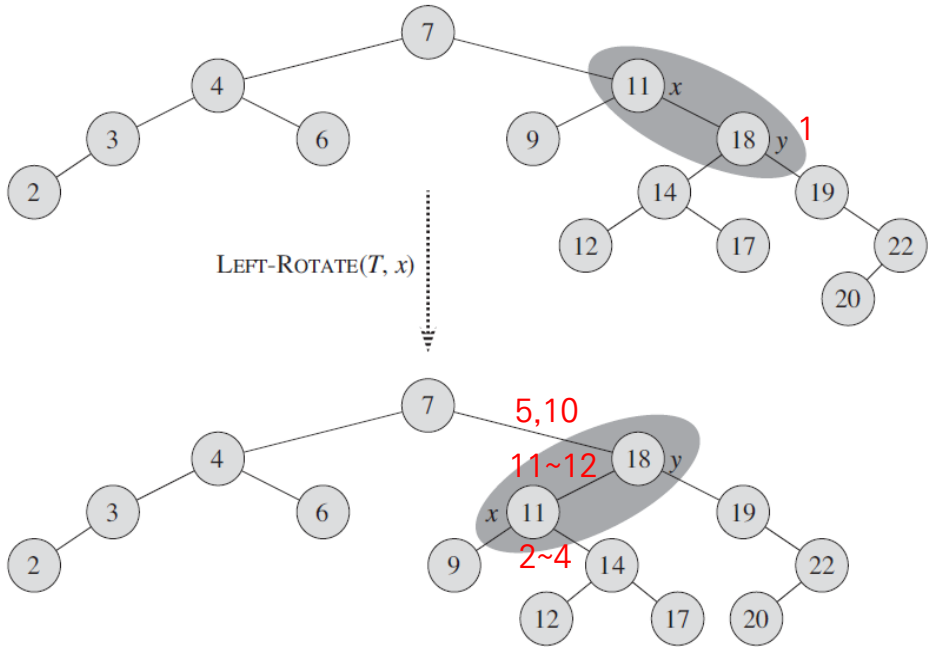


- ➢ line 1: set y as x.right
- ➢ line 2 : turn y's left subtree into x's right subtree $(x \rightarrow \beta)$
- ➢ line 3~4 : set β's parent as node x
- ➢ line 5 : set y's parent as x's parent
- ➢ line 6~7 : in the case that x is root node
- ➢ line 8~10 : set $x.p.left$ or $x.p.right$ as y
- ➢ line 11~12 : connect y and x

❖ Rotation

- Example

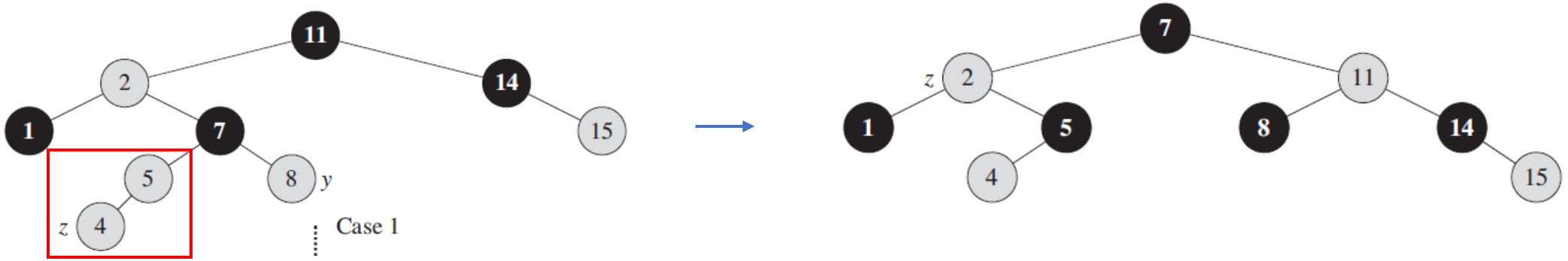
```
LEFT-ROTATE(T, x)
 1   y = x.right
 2   x.right = y.left
 3   if y.left ≠ T.nil
 4       y.left.p = x
 5   y.p = x.p
 6   if x.p == T.nil
 7       T.root = y
 8   elseif x == x.p.left
 9       x.p.left = y
10   else x.p.right = y
11   y.left = x
12   x.p = y
```



- Running time
  - ➢ Only parameters are changed by a rotation → $O(1)$

❖ Insertion

- $\boldsymbol{RB-INSERT}$
  - ➢ Can insert a node into an n-node red-black tree in $O(\lg n)$ time
  - ➢ Insert node z into the tree T as if it were an BST
    - ✓ after inserting, color the node z to red
    - ✓ Use a modified $\boldsymbol{TREE-INSERT}$ procedure
  - ➢ To guarantee that the RB tree properties are preserved, call auxiliary procedure $\boldsymbol{RB-INSERT-FIXUP}$ to recolor nodes and perform rotations



$\boldsymbol{RB-INSERT}$ $\boldsymbol{RB-INSERT-FIXUP}$

❖ Insertion

- ***RB − INSERT*** − Pseudocode

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ 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 == NIL
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

→

```
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)
```

➢ TREE-INSERT & RB-INSERT differ in four ways

1) All instances of NIL in $TREE - INSERT$ are replaced by $T.nil$
2) Set $z.left$ and $z.right$ to $T.nil$
   - ✓ line 14~15
   - ✓ maintain proper tree structure
3) Color z red
   - ✓ line 16
4) Coloring z red may cause a violation of the RB tree properties
   - ✓ call ***RB − INSERT − FIXUP*** (line 17)
   - ✓ Restore the red-block properties

❖ Insertion

- ***RB − INSERT − FIXUP***
  - ➢ To understand how ***RB − INSERT − FIXUP*** works, break examination of the code into three major steps

```
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              // case 1
 6              y.color = BLACK                // case 1
 7              z.p.p.color = RED              // case 1
 8              z = z.p.p                      // case 1
 9          else if z == z.p.right
10                  z = z.p                    // case 2
11                  LEFT-ROTATE(T, z)          // case 2
12              z.p.color = BLACK              // case 3
13              z.p.p.color = RED              // case 3
14              RIGHT-ROTATE(T, z.p.p)         // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

1) Determine what violations of the RB tree properties are introduced
2) Examine the overall goal of the while loop in line 1~15
   - ✓ Loop invariant
3) Explore each of three cases within the while loop's body and how they accomplish the goal

❖ Insertion

- ***RB − INSERT − FIXUP***

1) Determine what violations of the RB tree properties are introduced

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.

➢ Property 1 & 3

✓ Certainly continues to hold

✓ Both children of the newly inserted red node are the sentinel $T.nil$

➢ Property 5

✓ newly inserted node is red node

✓ replace black sentinel($T.nil$) but have two black sentinel children

The only properties that might be violated are property 2 and 4 → due to z being colored red

❖ Insertion

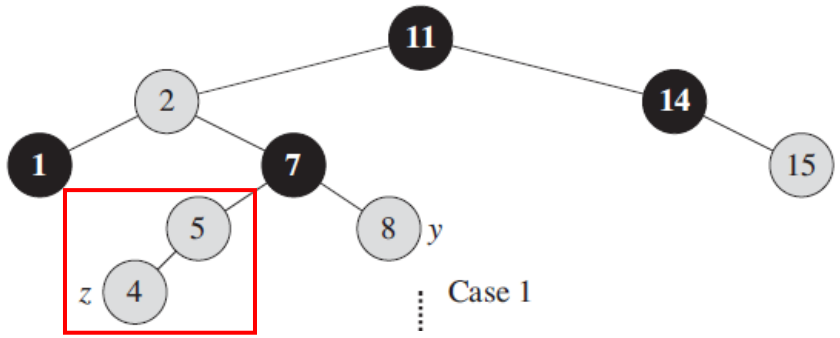
- ***RB − INSERT − FIXUP***

1) Determine what violations of the RB tree properties are introduced

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.

➢ Property 2
  ✓ Violated if z is the root node

➢ Property 4
  ✓ Violated if z's parent is red
  ✓ Red node has red child



❖ Insertion

- $RB - INSERT - FIXUP$

2) Examine the overall goal of the while loop in line 1~15 (Loop invariant)

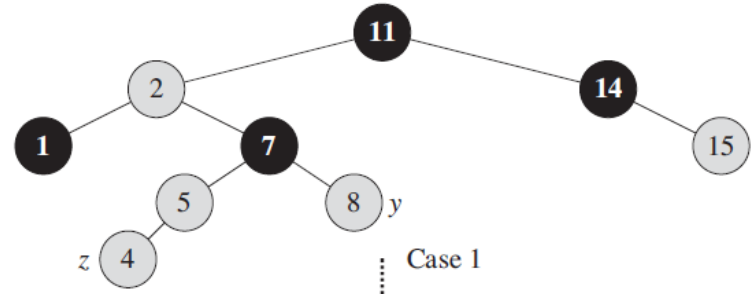
➢ The while loop in line 1~15 maintains the three-part invariant at the start of each iteration

a. Node z is red.

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

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

About violations



➢ Initialization

✓ Add red node z (a)

✓ The root did not changed prior to the call of $RB - INSERT - FIXUP$ (b)

✓ Prior to the first iteration, we started with RB-tree with no violations (c)

❖ Insertion

- ***RB − INSERT − FIXUP***

2) Examine the overall goal of the while loop in line 1~15 (Loop invariant)

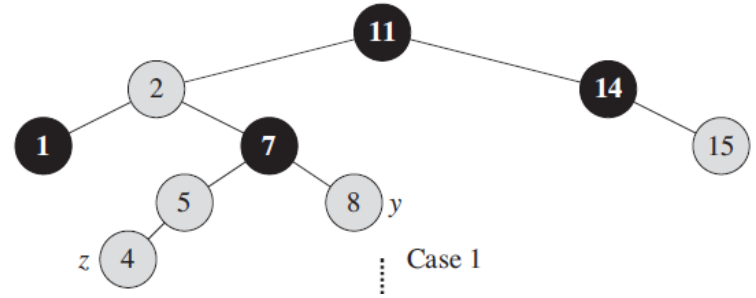
➢ The while loop in line 1~15 maintains the three-part invariant at the start of each iteration

a. Node z is red.

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

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

About violations



➢ Termination

✓ The loop is terminated when $z.p$ is black → tree does not violate property 4 (a) (b) (c)

✓ Line 16 (the last line) restore property 2 → 16 $T.root.color = \text{BLACK}$

❖ Insertion

- ***RB − INSERT − FIXUP***

2) Examine the overall goal of the while loop in line 1~15 (Loop invariant)

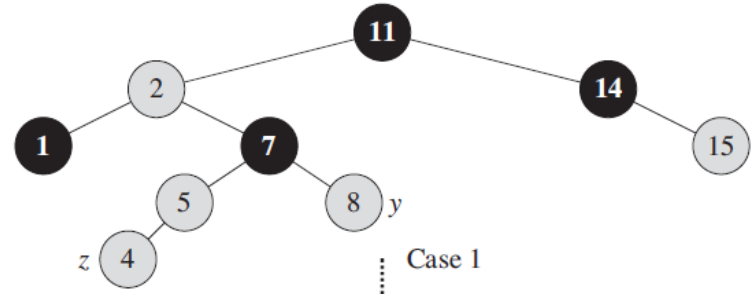
➢ The while loop in line 1~15 maintains the three-part invariant at the start of each iteration

a. Node z is red.

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

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

About violations



➢ Maintenance

✓ We need to consider six cases in the while loop, but three of them are symmetric

✓ Can enter a loop iteration only if $z.p$ is red → $z.p$ cannot be the root (a) (b)

✓ In step 3, we can find violation in property 4 in three cases (c)

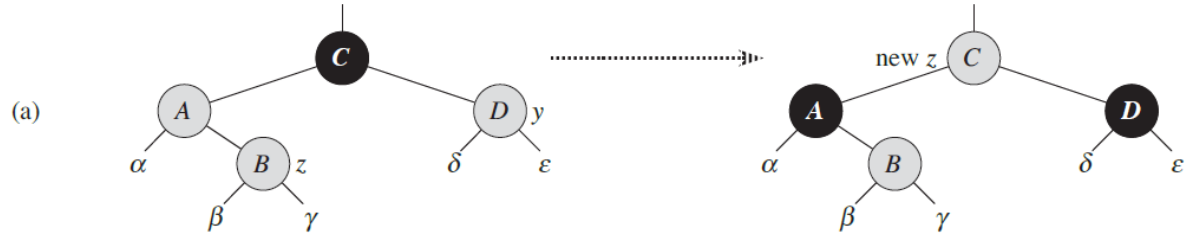
❖ Insertion

- ***RB − INSERT − FIXUP***

3) Explore each of three cases within the while loop's body and how they accomplish the goal

➢ Case 1 – z's uncle y is red

```
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                ▷ case 1
 6              y.color = BLACK                  ▷ case 1
 7              z.p.p.color = RED                ▷ case 1
 8              z = z.p.p                        ▷ case 1
 9          else if z == z.p.right
10                  z = z.p                      ▷ case 2
11                  LEFT-ROTATE(T, z)            ▷ case 2
12              z.p.color = BLACK                ▷ case 3
13              z.p.p.color = RED                ▷ case 3
14              RIGHT-ROTATE(T, z.p.p)           ▷ case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```



- ✓ line 3 : set y as z's uncle
- ✓ Both $z.p$ and y are red
  - i. We can know that $z.p.p$ is black (property 4)
  - ii. Can color $z.p$ and y are black
  - iii. Can fix the problem of z and $z.p$ both being red
- ✓ But we need to color $z.p.p$ red → maintaining property 5
- ✓ Repeat the while loop with $z.p.p$ as the new node z
  - i. Pointer z moves up two levels in the tree

❖ Insertion

- ***RB − INSERT − FIXUP***

3) Explore each of three cases within the while loop's body and how they accomplish the goal

➢ Case 1 – z's uncle y is red – **loop invariant**

```
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              // case 1
 6              y.color = BLACK                // case 1
 7              z.p.p.color = RED              // case 1
 8              z = z.p.p                      // case 1
 9          else if z == z.p.right
10                  z = z.p                    // case 2
11                  LEFT-ROTATE(T, z)          // case 2
12              z.p.color = BLACK              // case 3
13              z.p.p.color = RED              // case 3
14              RIGHT-ROTATE(T, z.p.p)         // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

a. Node z is red.

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

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

✓ z denotes node z, z^s denotes $z.p.p$

a. color $z.p.p$ red(7), z^s becomes z at the start of next iteration(8)

b. $z^s.p = z.p.p.p$,

   i. if $z^s.p$ is root, it was black prior to this iteration

   ii. The color of this node doesn't change

c. If node z^s is the root at the start of the next iteration, violate property 2

   i. Since z^s is red but it is the root of the tree

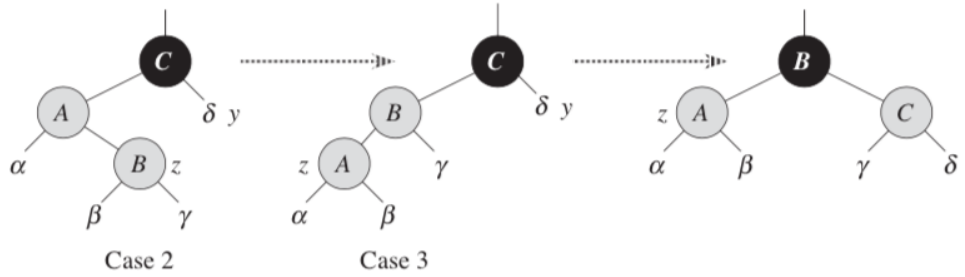
❖ Insertion

- ***RB − INSERT − FIXUP***

3) Explore each of three cases within the while loop's body and how they accomplish the goal

➢ Case 2, 3 – z's uncle y is black and z is a right child(2) or left child(3)

```
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                ▷ case 1
 6              y.color = BLACK                  ▷ case 1
 7              z.p.p.color = RED                ▷ case 1
 8              z = z.p.p                        ▷ case 1
 9          else if z == z.p.right
10              z = z.p                          ▷ case 2
11              LEFT-ROTATE(T, z)                ▷ case 2
12              z.p.color = BLACK                ▷ case 3
13              z.p.p.color = RED                ▷ case 3
14              RIGHT-ROTATE(T, z.p.p)           ▷ case 3
15      else (same as then clause
              with "right" and "left" exchanged)
16  T.root.color = BLACK
```



✓ Distinguish the two cases according to whether z is left or right child of $z.p$

✓ line 10~11

i. constitute case 2

ii. using $LEFT - ROTATE$, transform the situation into case 3

✓ In case 3, both z and $z.p$ are red

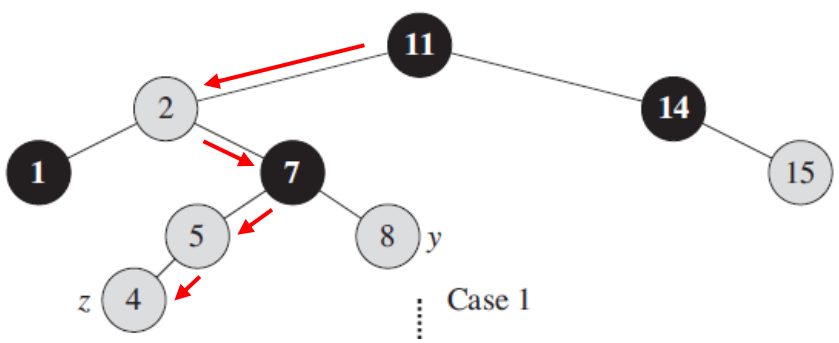
Black-height

i. If we use $RIGHT - ROTATE$, violates properties 5

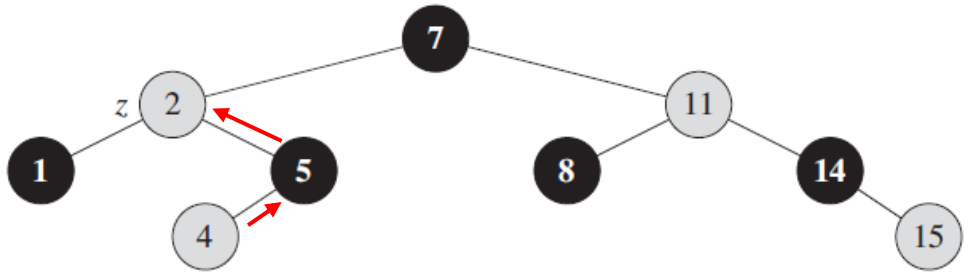
ii. Color $z.p$ black and $z.p.p$ red → use $RIGHT - ROTATE$

❖ Insertion

- The running time of $\boldsymbol{INSERT}$ procedure
  - ➢ The running time of $\boldsymbol{RB-INSERT}$ and $\boldsymbol{RB-INSERT-FIXUP}$ take $O(\lg n)$ time



$\boldsymbol{RB-INSERT}$



$\boldsymbol{RB-INSERT-FIXUP}$

- ✓ Moves from root to leaf
- ✓ Takes $O(h) = O(\lg n)$

- ✓ Moves from leaf to root
- ✓ Just change pointer and color attribute
- ✓ Takes $O(h) = O(\lg n)$

❖ Deletion

- Introduction
  - ➢ Deletion of a node takes $O(\lg n)$ time
  - ➢ Deleting a node from a RB tree is based on the $\boldsymbol{TREE-DELETE}$ procedure
  - ➢ Use $\boldsymbol{TRANSPLANT}$ subroutine that $\boldsymbol{TREE-DELETE}$ calls so that it applies to a RB tree

- $\boldsymbol{RB-TRANSPLANT}$

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

→

```
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
```

➢ $\boldsymbol{RB-TRANSPLANT}$ & $\boldsymbol{TRANSPLANT}$ differ in two ways

1) All instances of NIL in $\boldsymbol{TRANSPLANT}$ are replaced by sentinel $T.nil$
2) The assignment to $v.p$ occurs unconditionally
   - ✓ Can assign to $v.p$ even if v points to the sentinel

❖ Deletion

- $\boldsymbol{RB-DELETE}$
  - ➢ $\boldsymbol{RB-DELETE}$ is like the $\boldsymbol{TREE-DELETE}$ with additional lines of pseudocode
  - ➢ Whole process
    - ✓ Keep track of a node y that might cause violations of the RB tree properties
    - ✓ When we want to delete the node z
      - i. If z has fewer than two children → z is directly removed from tree → node y to be z
      - ii. If z has two children → y should be z's successor → y moves to z's position
    - ✓ Must store y's color before it is removed(i) from or moved(ii) within the tree
    - ✓ After deleting node z, $\boldsymbol{RB-DELETE}$ calls an auxiliary procedure $\boldsymbol{RB-DELETE-FIXUP}$
      - i. maintain RB tree properties

❖ Deletion

- $RB - DELETE$ – pseudocode

```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

→

```
RB-DELETE(T, z)
1   y = z
2   y-original-color = y.color
3   if z.left == T.nil
4       x = z.right
5       RB-TRANSPLANT(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-TRANSPLANT(T, z, z.left)
9   else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

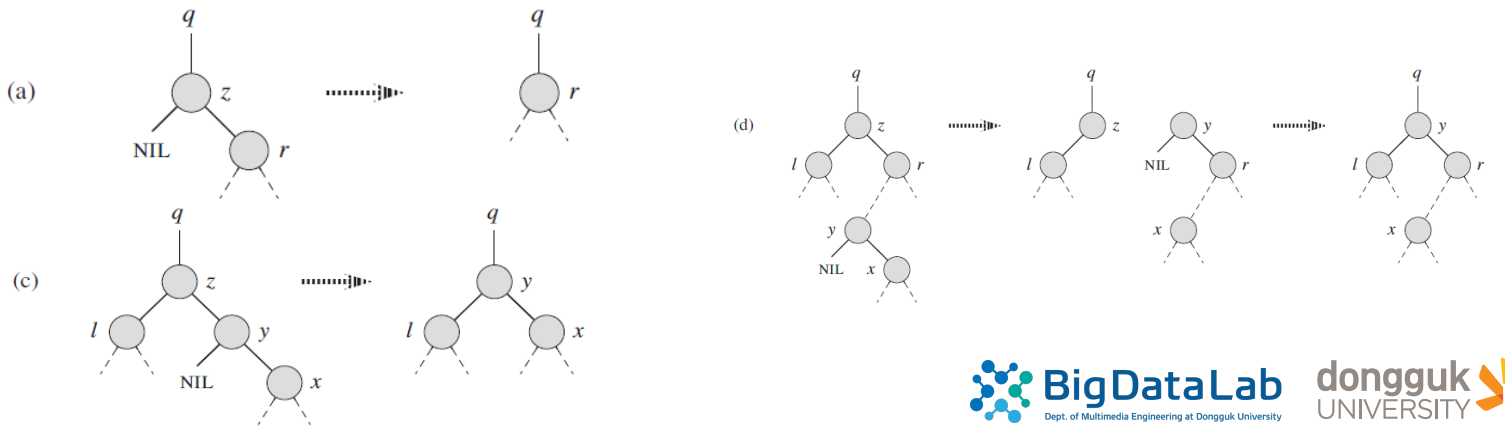
- Have the same basic-structure
  - ✓ $NIL \rightarrow T.nil$
  - ✓ $TRANSPLANT \rightarrow RB - TRANSPLANT$
- Maintain node y
  - ✓ It is removed or moved within tree
  - ✓ Set y as z (1)
- node y's color might change
  - ✓ $y - original - color$ stores y's color before any changes occur (2, 10)

❖ Deletion

- $RB - DELETE$ – pseudocode

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

- Keep track of the node x that moves into node y's original position
  - ✓ If node z has only one child node → save y's original position (4,7)
  - ✓ Directly delete node z using $RB - TRANSPLANT$
- When z has two children
  - ✓ Node y moves into node z's position
  - ✓ line 13~20
- If $y - original - color$ is black, it could violate the RB tree properties 5
  - ✓ Call $RB - DELETE - FIXUP$ using value x



Thank You!

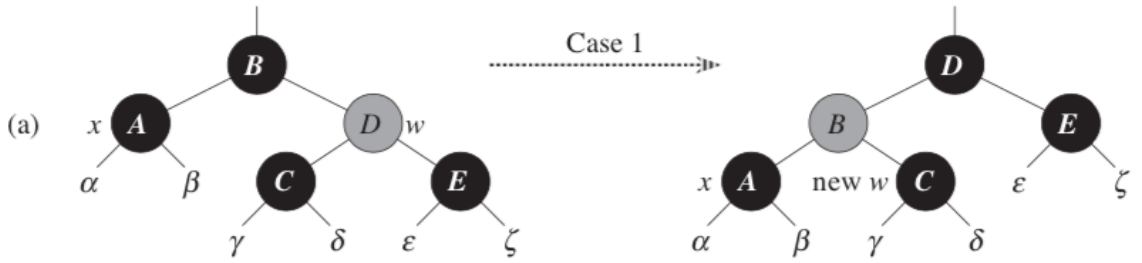
❖ Deletion

- ***RB − DELETE − FIXUP*** − pseudocode

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                          // case 1
 6              x.p.color = RED                          // case 1
 7              LEFT-ROTATE(T, x.p)                      // case 1
 8              w = x.p.right                            // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                            // case 2
11              x = x.p                                  // case 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK                 // case 3
14                  w.color = RED                        // case 3
15                  RIGHT-ROTATE(T, w)                   // case 3
16                  w = x.p.right                        // case 3
17              w.color = x.p.color                      // case 4
18              x.p.color = BLACK                        // case 4
19              w.right.color = BLACK                    // case 4
20              LEFT-ROTATE(T, x.p)                      // case 4
21              x = T.root                               // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

➢ The pseudocode fixup 4 cases violations

1) Case 1 : x's sibling w is red



✓ Switch the colors of w and $x.p$

✓ Perform a left rotation on $x.p$

✓ Move pointer w to $x.p.right$

✓ Transform case 1 to case 2,3,or4

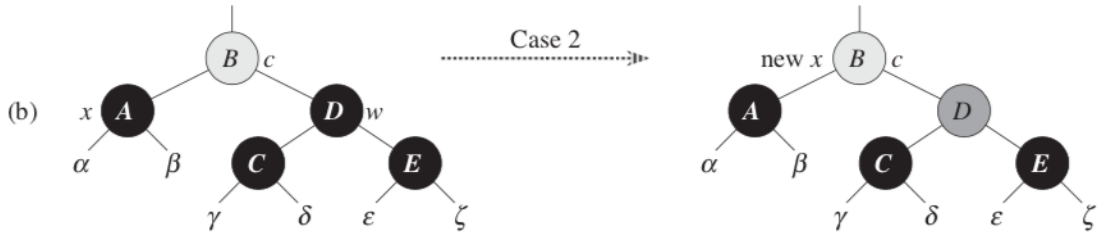
❖ Deletion

- ***RB − DELETE − FIXUP*** − pseudocode

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                          // case 1
 6              x.p.color = RED                          // case 1
 7              LEFT-ROTATE(T, x.p)                      // case 1
 8              w = x.p.right                            // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                            // case 2
11              x = x.p                                  // case 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK                 // case 3
14                  w.color = RED                        // case 3
15                  RIGHT-ROTATE(T, w)                   // case 3
16                  w = x.p.right                        // case 3
17              w.color = x.p.color                      // case 4
18              x.p.color = BLACK                        // case 4
19              w.right.color = BLACK                    // case 4
20              LEFT-ROTATE(T, x.p)                      // case 4
21              x = T.root                               // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

➢ The pseudocode fixup 4 cases violations

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



- ✓ Change w color as red (10)
- ✓ Move pointer x to $x.p$ (11)
- ✓ If new x is black → examine case 1
- ✓ If new x is red → terminates loop and change to black(23)

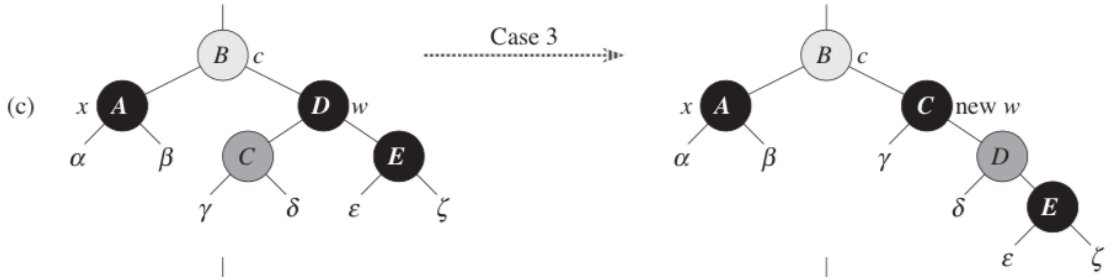
❖ Deletion

- ***RB − DELETE − FIXUP*** − pseudocode

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                          // case 1
 6              x.p.color = RED                          // case 1
 7              LEFT-ROTATE(T, x.p)                      // case 1
 8              w = x.p.right                            // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                            // case 2
11              x = x.p                                  // case 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK                 // case 3
14                  w.color = RED                        // case 3
15                  RIGHT-ROTATE(T, w)                   // case 3
16                  w = x.p.right                        // case 3
17              w.color = x.p.color                      // case 4
18              x.p.color = BLACK                        // case 4
19              w.right.color = BLACK                    // case 4
20              LEFT-ROTATE(T, x.p)                      // case 4
21              x = T.root                               // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

➢ The pseudocode fixup 4 cases violations

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



- ✓ Switch color of w and $w.left$ (13~14)
- ✓ Perform a right rotation on w (15)
- ✓ Move pointer w to $x.p.right$ (16)
- ✓ Transform case 3 to case 4

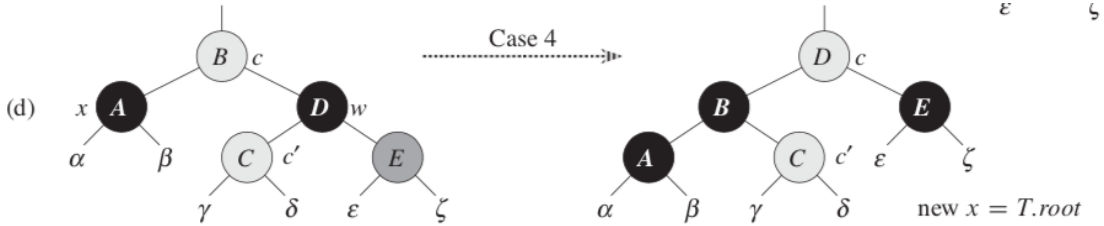
❖ Deletion

- $RB-DELETE-FIXUP$ – pseudocode

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                          // case 1
 6              x.p.color = RED                          // case 1
 7              LEFT-ROTATE(T, x.p)                      // case 1
 8              w = x.p.right                            // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                            // case 2
11              x = x.p                                  // case 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK                 // case 3
14                  w.color = RED                        // case 3
15                  RIGHT-ROTATE(T, w)                   // case 3
16                  w = x.p.right                        // case 3
17              w.color = x.p.color                      // case 4
18              x.p.color = BLACK                        // case 4
19              w.right.color = BLACK                    // case 4
20              LEFT-ROTATE(T, x.p)                      // case 4
21              x = T.root                               // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

- The pseudocode fixup 4 cases violations

4) Case 4 : x's sibling w is black, w's left child is red



- ✓ By making some color changes (17~19)
- ✓ Perform a left rotation on $x.p$ (20)
- ✓ Set x to be the root to terminate loop condition (21)