CSE3081 Design and Analysis of Algorithms

Dept. of Computer Engineering,
Sogang University

Chapter 13. Red-Black Trees

Binary Search Tree

- A binary search tree of height h can support any of the basic dynamic-set operations such as insert, delete, search in $O(h)$ time.
  - The height of a node in a tree is the number of edges on the longest simple downward path from the node to a leaf.
  - The height of a tree is the height of its root.

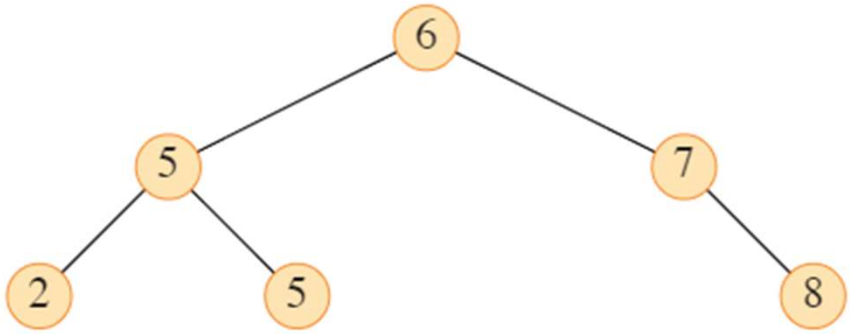
- These operations are fast if the height of the binary search tree is small.
  - $O(\log n)$ when n is the number of elements in the tree.

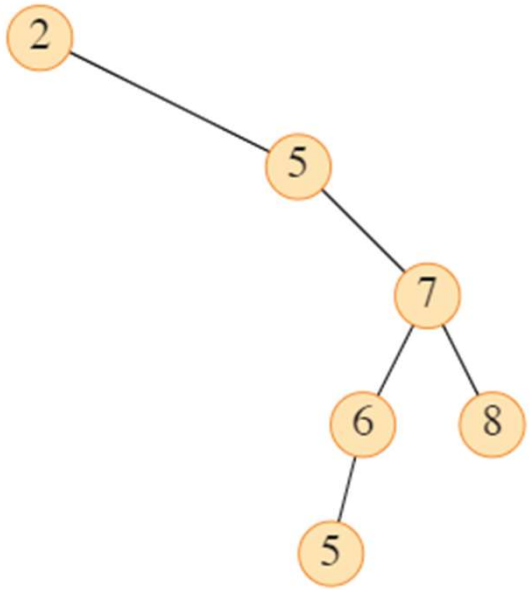
- However, if its height is large, the operations take longer time.
  - $O(n)$ when n is the number of elements in the tree.

- Height of a binary search tree is small when the tree is "balanced".

Binary Search Tree: Examples

- balanced vs. unbalanced binary search tree





Red-Black Tree

- One of the many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.

13.1 Properties of Red-Black Trees

Red-Black Tree

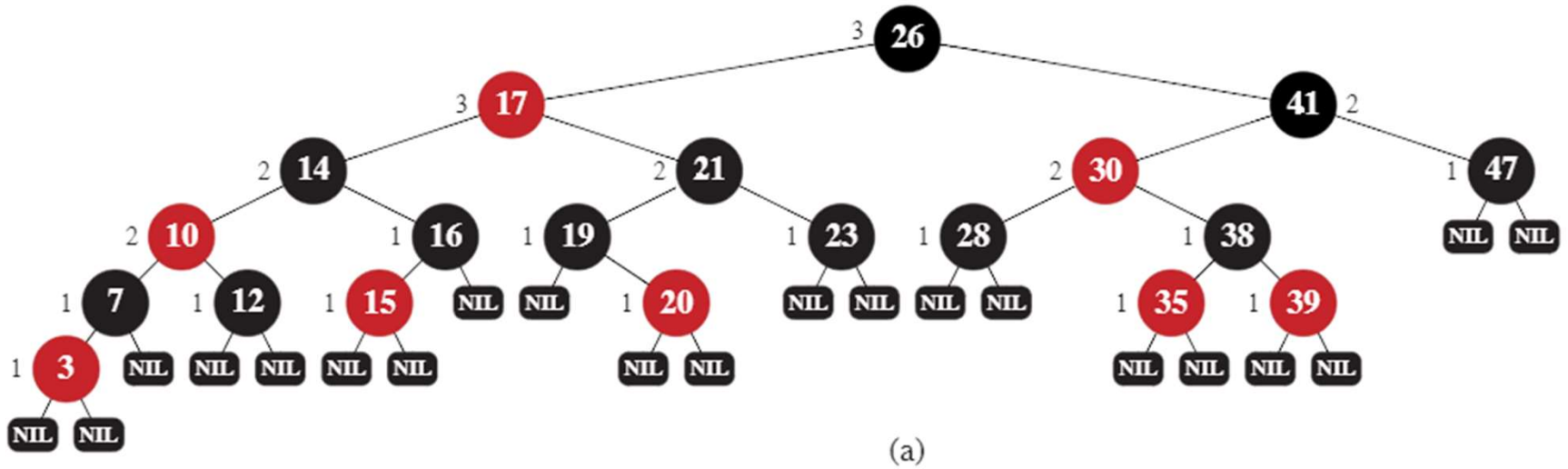
- A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK.

- 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.
  - The tree is approximately balanced.

- The height of a red-black tree with n keys is at most $2\log(n+1)$, which is $O(\log n)$.

Red-Black Tree: Red-Black Properties

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



(a)

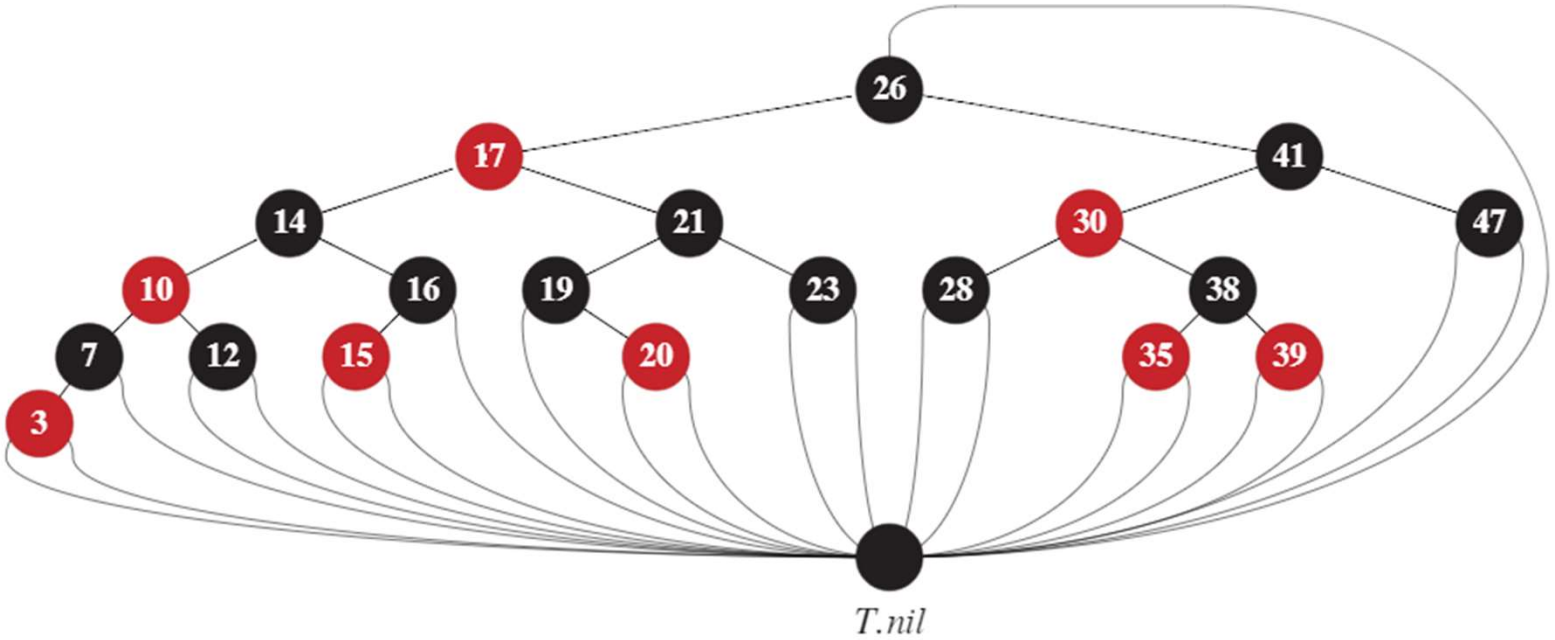
Red-Black Tree: Node Attributes and NIL

- Each node of the tree contains the following attributes
  - *color*: BLACK or RED
  - *key*: key representing the node (the number)
  - *left*: left child node
  - *right*: right child node
  - *p*: parent

- If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL.

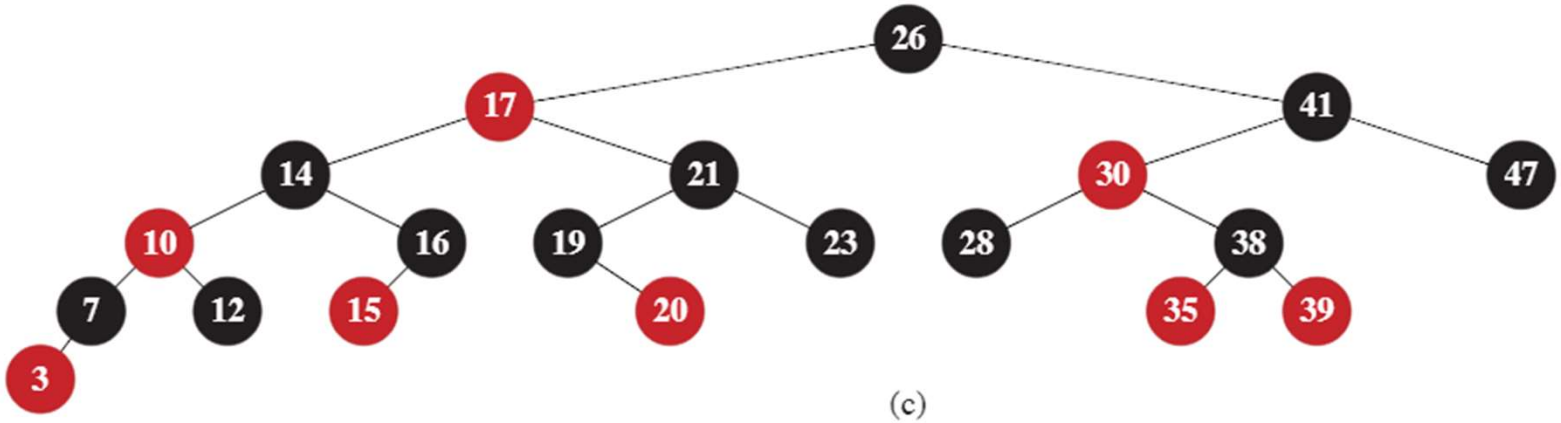
Red-Black Tree: NIL Representation

- For convenience in dealing with boundary conditions in red-black tree code, we may use a single sentinel to represent NIL.
- $T.nil$ represents all the NILS – all leaves and the root's parent.
- The values of the attributes *p*, *left*, *right*, and *key* of the sentinel are immaterial.



Red-Black Tree: Drawing Convention

- We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values.
- Thus, we often omit the leaves in drawings of red-black trees.



(c)

Red-Black Tree: Black-Height

- Black-height
  - The number of black nodes on any simple path from, but not including, a node x down to a leaf

- We denote the black-height of node x as $\mathrm{bh}(x)$.

- The notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes (Property 5).

- The black-height of a red-black tree is the black-height of its root.

Red-Black Tree: Why it makes a good search tree

- Lemma 13.1
  - A red-black tree with n internal nodes has height at most $2\log(n+1)$.

- We start by showing that the subtree rooted at any node x contains at least $2^{bh(x)}-1$ internal nodes.
  - We prove this claim by induction on the height of x.

- base step
  - If the height of x is 0, then x must be a leaf ($T.nil$), and the subtree rooted at x contains at least $2^{bh(x)}-1=2^0-1=0$ internal nodes.

- inductive step
  - Consider a node x that has positive height and is an internal node. Then node x has two children, either or both which may be a leaf.

Red-Black Tree: Why it makes a good search tree

- Inductive step (cont.)
  - If a child is black, it contributes 1 to x's black-height but not to its own.
  - If a child is red, then it contributes to neither x's black-height nor its own.
  - Therefore, each child has a black-height of either $bh(x) - 1$ (if it is black) or $bh(x)$ (if it is red).
  - Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes.
  - Then, the subtree rooted at x contains at least $\left(2^{bh(x)-1} - 1\right) + \left(2^{bh(x)-1} - 1\right) + 1 = 2^{bh(x)} - 1$ internal nodes, which proves the claim.

Red-Black Tree: Why it makes a good search tree

- Let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black.
  - Property 4: If the node is red, then both its children are black.

- Consequently, the black-height of the root must be at least $h/2$, and thus,
- $n \geq 2^{h/2} - 1$

- Moving the 1 to the left-hand side and taking logarithms on both sides yields $\log(n+1) \geq h/2$, or $h \leq 2\log(n+1)$.

- Thus, dynamic-set operations such as insert, search, and delete run in $O(\log n)$ time on a red-black tree.

13.2 Rotations

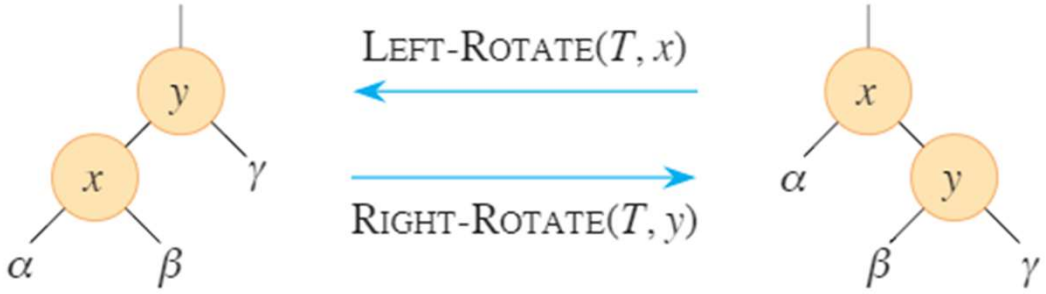
Insertion and Deletion on a Red-Black Tree

- When we apply the operations TREE-INSERT and TREE-DELETE, it modifies the tree.
- The result may violate the red-black properties.
- To restore these properties, colors and pointers within nodes need to change.

Rotation

- The pointer structure changes through rotation.
- Rotation is a local operation that preserves the binary-search-tree property.

- Left rotation
  - Node x has a right child y, which must not be NIL.
  - The left rotation changes the subtree originally rooted at x by twisting the link between x and y to the left.
  - The new root of the subtree is y.
  - x becomes y's left child.
  - y's original left child becomes x's right child.



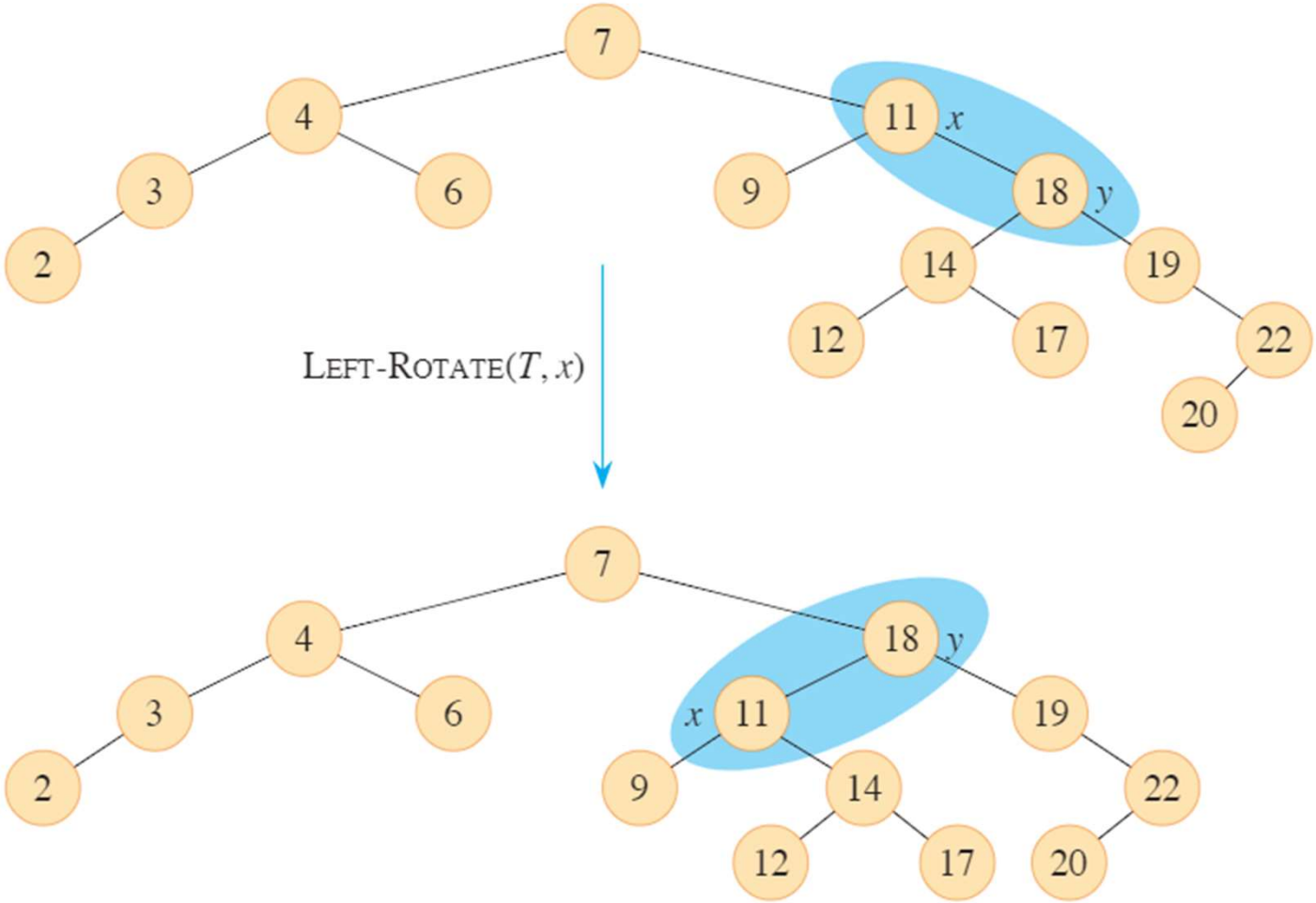
Rotation: Pseudocode

- Pseudocode for left rotation
  - The code assumes that $x.right$ is not NIL.

```
LEFT-ROTATE(T, x)
 1  y = x.right
 2  x.right = y.left        // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil       // if y's left subtree is not empty ...
 4       y.left.p = x       // ... then x becomes the parent of the subtree's root
 5  y.p = x.p               // x's parent becomes y's parent
 6  if x.p == T.nil         // if x was the root ...
 7       T.root = y         // ... then y becomes the root
 8  elseif x == x.p.left    // otherwise, if x was a left child ...
 9       x.p.left = y       // ... then y becomes a left child
10  else x.p.right = y      // otherwise, x was a right child, and now y is
11  y.left = x              // make x become y's left child
12  x.p = y
```

- The code for right rotation will be symmetric.
- Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time.

Rotation: Example



13.3 Insertion

Insertion into a Red-Black Tree

- In order to insert a node into a red-black tree with n internal nodes in $O(\log n)$ time and maintain the red-black properties, we need to modify the TREE-INSERT procedure.

- First, insert node z into the tree T as if it were an ordinary binary search tree, then z is given the color red.

- To guarantee that the red-black properties are preserved, an auxiliary procedure RB-INSERT-FIXUP recolors nodes and performs rotations.

Insertion: Procedure RB-INSERT

```
RB-INSERT(T, z)
 1  x = T.root                    // node being compared with z
 2  y = T.nil                     // y will be parent of z
 3  while x ≠ T.nil               // descend until reaching the sentinel
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y                       // found the location—insert z with parent y
 9  if y == T.nil
10      T.root = z                // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil                // both of z's children are the sentinel
15  z.right = T.nil
16  z.color = RED                 // the new node starts out red
17  RB-INSERT-FIXUP(T, z)         // correct any violations of red-black properties
```

Insertion: Procedure RB-INSERT-FIXUP (1)

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left                  // is z's parent a left child?
 3          y = z.p.p.right                   // y is z's uncle
 4          if y.color == RED                 // are z's parent and uncle both red?
 5              z.p.color = BLACK             }
 6              y.color = BLACK               }  case 1
 7              z.p.p.color = RED             }
 8              z = z.p.p                     }
 9          else
10              if z == z.p.right
11                  z = z.p                   }  case 2
12                  LEFT-ROTATE(T, z)         }
13              z.p.color = BLACK             }
14              z.p.p.color = RED             }  case 3
15              RIGHT-ROTATE(T, z.p.p)        }
```

Insertion: Procedure RB-INSERT-FIXUP (2)

```
16      else // same as lines 3–15, but with "right" and "left" exchanged
17           y = z.p.p.left
18           if y.color == RED
19                z.p.color = BLACK
20                y.color = BLACK
21                z.p.p.color = RED
22                z = z.p.p
23           else
24                if z == z.p.left
25                     z = z.p
26                     RIGHT-ROTATE(T, z)
27                z.p.color = BLACK
28                z.p.p.color = RED
29                LEFT-ROTATE(T, z.p.p)
30   T.root.color = BLACK
```

RB-INSERT-FIXUP: violation of red-black property

- After inserting a node and before calling RB-INSERT-FIXUP, which red-black property can be violated?

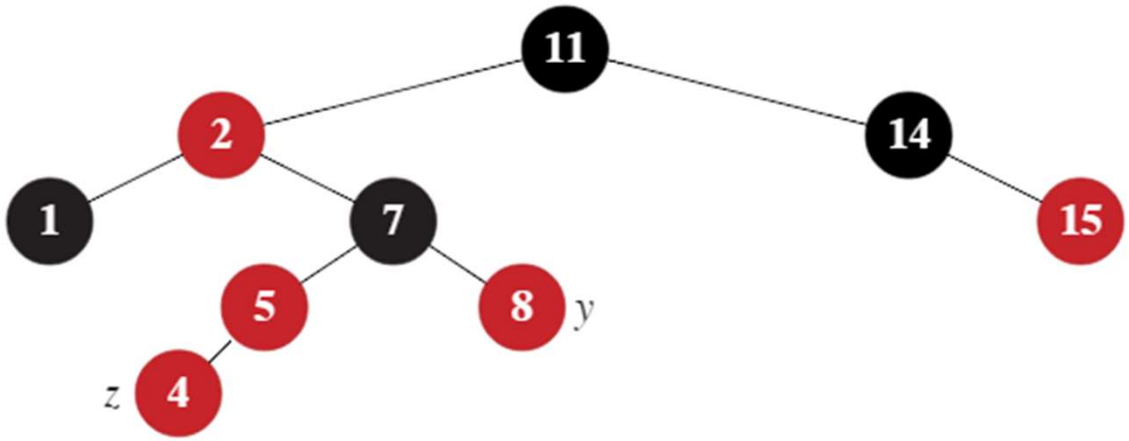
- Red-black property
  1. Every node is either red or black. → maintained
  2. The root is black.
  3. Every leaf (NIL) is black. → maintained
  4. If the node is red, then both its children are black.
  5. For each node, all simple paths from the node to descendant leaves contain the same number of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

- Property 5 is maintained because node z replaces the (black) sentinel (NIL), and node z is red with sentinel children.

RB-INSERT-FIXUP: violation of red-black property

- Property 2 is violated if z is the root.

- Property 4 is violated if z's parent is red.



RB-INSERT-FIXUP: restoring the red-black property

- The while loop of lines 1-29 has two symmetric possibilities.
  - lines 3-15 deal with the situation in which node z's parent $z.p$ is a left child of z's grandparent $z.p.p$.
  - lines 17-29 deal with the situation in which $z.p$ is a right child of $z.p.p$.
  - These code blocks are symmetric. So we focus on lines 3-15.

- The while loop maintains the following three-part invariant at the start of each iteration of the loop.
  1. Node z is red
  2. If $z.p$ is the root, then $z.p$ is black.
  3. 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, but not both.
     - 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.

RB-INSERT-FIXUP: Loop Invariant Initialization

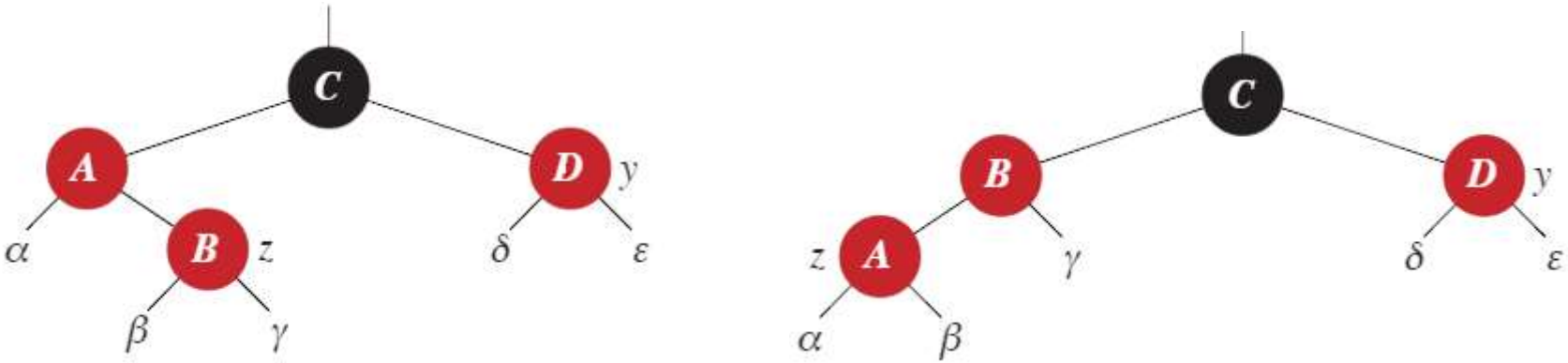
- Before RB-INSERT is called, the red-black tree has no violations.
- RB-INSERT adds a red node z and calls RB-INSERT-FIXUP.

- When RB-INSERT-FIXUP is called, z is the red node that was added.
- If $z.p$ is the root, then $z.p$ started out black (property 2) and did not change before the call of RB-INSERT-FIXUP.
- If the tree violates property 2 (the root must be black), then the red root must be the newly added node z, which is the only internal node in the tree. Because the parent and both children of z are the sentinel (NIL), which is black, the tree does not also violate property 4.
- If the tree violates property 4 (both children of a red node are black), then, because the children of node z are black sentinels and the tree had no other violations prior to z being added, the violation must be because both z and $z.p$ are red. Since $z.p$ is red, which means z is not the root, the tree does not violate property 2.

RB-INSERT-FIXUP: Loop Invariant Maintenance

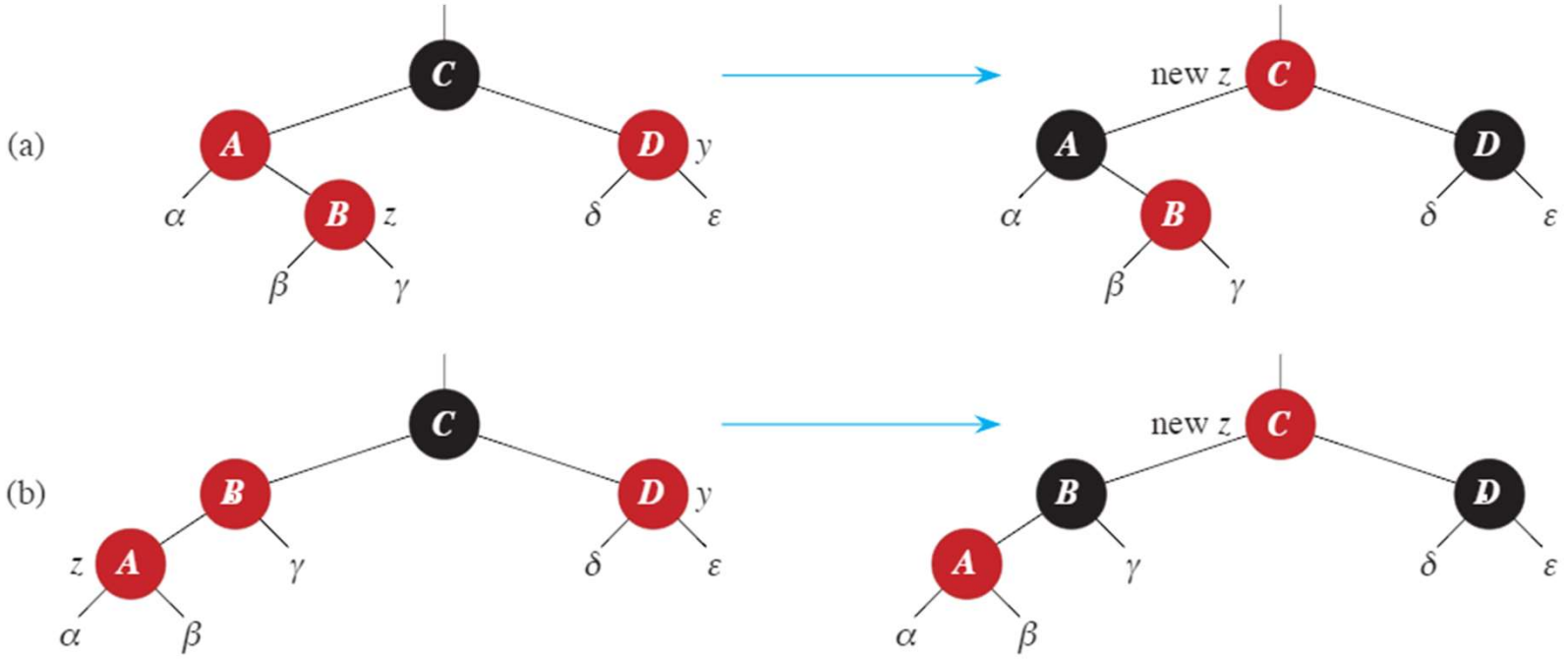
- There are three cases in lines 3-15.
- We only enter the while loop when $z.p$ is red. Thus, $z.p$ cannot be the root. Hence, $z.p.p$ exists.

- Case 1: z's uncle y is red (lines 4-8)
  - z's "uncle" is the sibling of z's parent.



RB-INSERT-FIXUP: Loop Invariant Maintenance

- We can change the color between $z.p.p$ (black) and $z.p$ (red). (lines 5-7)
  - Property 5 is preserved, because the blackness transferred down one level.



- Then, we make $z.p.p$ as the new z. (line 8)

RB-INSERT-FIXUP: Loop Invariant Maintenance

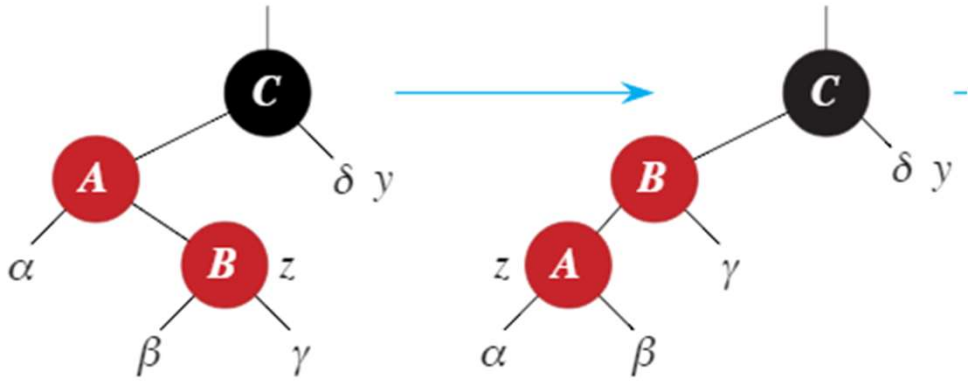
- Why does actions in case 1 maintain loop invariant?

- Because this iteration colors $z.p.p$ red, node z' is red at the start of the next iteration.
- The node $z'.p$ is $z.p.p.p$ in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.
- If node z' is the root at the start of next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since z' is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to z'.
- If node z' is not the root at the start of next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4. If $z'.p$ was black, there is no violation of property 4. If $z'.p$ was red, coloring z' red created one violation of property 4, between z' and $z'.p$.

RB-INSERT-FIXUP: Loop Invariant Maintenance

- 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

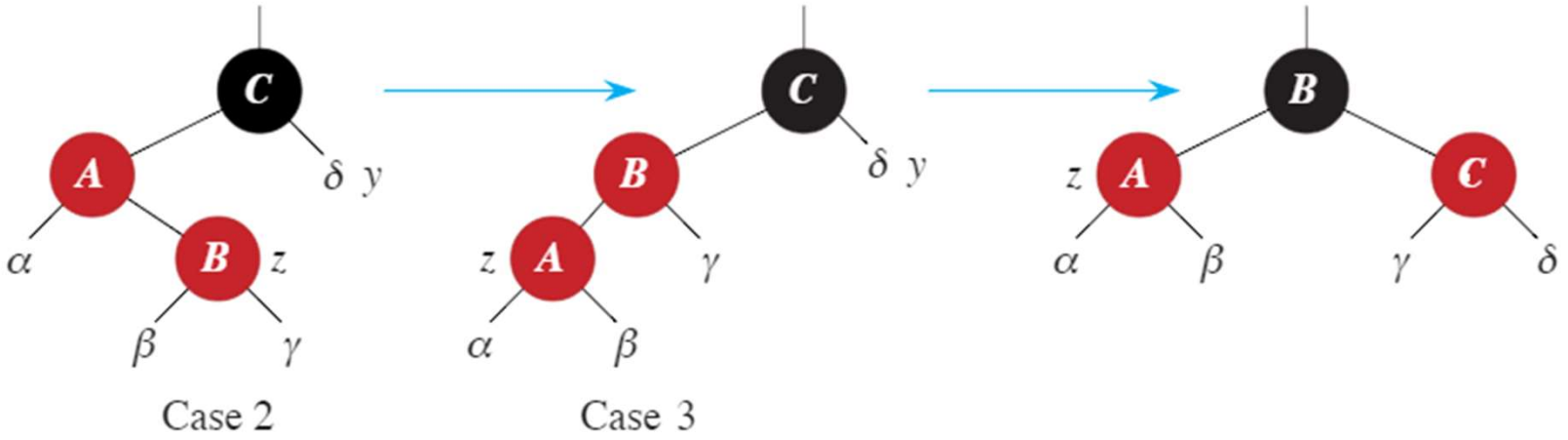
- In case 2 and 3, the color of z's uncle y is black.
- Lines 11-12 constitute case 2, where node z is a right child of its parent.
- A left rotation transforms the situation into case 3 (lines 13-15).
- z now points to what was $z.p$ in the iteration.
- Because both z and $z.p$ are red, the rotation affects the black-heights of nodes nor property 5.



Case 2 Case 3

RB-INSERT-FIXUP: Loop Invariant Maintenance

- Case 3 performs some color changes and right rotation.
  - $z.p$'s color becomes black, and $z.p.p$'s color becomes red.
  - Then we perform right rotation
- After case 3, there are no longer two red nodes in a row.
- The while loop terminates upon the next test in line 1, since $z.p$ is not black.



RB-INSERT-FIXUP: Loop Invariant Maintenance

- Why does actions in case 2 and 3 maintain loop invariant?

- Case 2 makes z point to $z.p$, which is red. No further changes to z or its color occurs in case 2 and 3.
- Case 3 makes $z.p$ black, so that if $z.p$ is the root at the start of the next iteration, it is black.
- As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.
- Since node z is not the root in cases 2 and 3, we know that there is no violation of property 2.
- Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.
- Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.

RB-INSERT-FIXUP: Loop Invariant Termination

- If case 1 occurs, the node pointer z moves toward the root in each iteration, so that eventually $z.p$ is black. (If z is the root, then $z.p$ is the sentinel $T.nil$, which is black.)
- If either case 2 or case 3 occurs, then the loop terminates.
- Since the loop terminates because $z.p$ is black, the tree does not violate property 4 at loop termination.
- By the loop invariant, the only property might fail to hold is property 2.
- Line 30 restores this property by coloring the root black.

- Thus, when RB-INSERT-FIXUP terminates, all the red-black properties hold.

RB-INSERT: running time

- Lines 1-16 in RB-INSERT (before calling RB-INSERT-FIXUP) take $O(\log n)$ time.
- In RB-INSERT-FIXUP, the while loop repeats only if case 1 occurs.
- If case 1 occurs, the pointer moves two levels up the tree.
- The total number of times the while loop can be executed is therefore $O(\log n)$.

- Thus, RB-INSERT takes a total of $O(\log n)$ time.

13.4 Deletion

Deletion from a Red-Black Tree

- To delete a node from a red-black tree in $O(\log n)$ time, we need to modify TREE-DELETE and TRANSPLANT procedures.

- First, we modify TRANSPLANT procedure to make RB-TRANSPLANT.
- Two differences with TRANSPLANT
  - Line 1 references the sentinel $T.nil$ instead of NIL.
  - The assignment to $v.p$ in line 6 occurs unconditionally: the procedure can assign to $v.p$ even if v points to the sentinel.

```
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 (1)

```
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)        // replace z by its right child
 6  elseif z.right == T.nil
 7       x = z.left
 8       RB-TRANSPLANT(T, z, z.left)         // replace z by its left child
```

Deletion: RB-DELETE (2)

```
 9  else y = TREE-MINIMUM(z.right)          // y is z's successor
10       y-original-color = y.color
11       x = y.right
12       if y ≠ z.right                     // is y farther down the tree?
13            RB-TRANSPLANT(T, y, y.right)  // replace y by its right child
14            y.right = z.right             // z's right child becomes
15            y.right.p = y                 //       y's right child
16       else x.p = y                       // in case x is T.nil
17       RB-TRANSPLANT(T, z, y)             // replace z by its successor y
18       y.left = z.left                    // and give z's left child to y,
19       y.left.p = y                       //       which had no left child
20       y.color = z.color
21  if y-original-color == BLACK            // if any red-black violations occurred,
22       RB-DELETE-FIXUP(T, x)              //       correct them
```

RB-DELETE: explanation

- RB-DELETE is like the TREE-DELETE procedure, with additional lines of code to deal with node x and y that may be involved in red-black property violations.
- node z: the node being deleted
- node y
  - When node z has at most one child, then y will be z.
  - When node z has two children, then y will be z's successor, which has no left child and moves into z's position in the tree.
  - Additionally, y takes on z's color.
- node x
  - Whether node z has at most one child or z has two children, node y will have at most one child. We call this child node x, which takes y's place in the tree.
  - (Node x will be the sentinel $T.nil$ if y has no children.)
- Since node y will be either removed from the tree or moved within the tree, the procedure needs to keep track of y's original color.
- If the red-black properties might be violated after deleting node z, an auxiliary procedure RB-DELETE-FIXUP is called to restore the properties.

Difference between TREE-DELETE and RB-DELETE

- Lines 1 and 9: set node y
  - When node z being deleted has at most one child, then y will be z. (line 1)
  - When z has two children, then, as in TREE-DELETE, y will be z's successor.

- Lines 2 and 10: record y's color
  - Because node y's color might change, the variable y-original-color stores y's color immediately after assignments to y.
  - When node z has two children, then nodes y and z are distinct. In this case, line 17 moves y into z's original position in the tree, and line 20 gives y the same color as z.
  - When node y was originally black, removing or moving it could cause violations of the red-black properties, which is corrected by RB-DELETE-FIXUP in line 22.

Difference between TREE-DELETE and RB-DELETE

- Lines 1 and 9: set node y
  - When node z being deleted has at most one child, then y will be z. (line 1)
  - When z has two children, then, as in TREE-DELETE, y will be z's successor.

- Lines 2 and 10: record y's color
  - Because node y's color might change, the variable y-original-color stores y's color immediately after assignments to y.
  - When node z has two children, then nodes y and z are distinct. In this case, line 17 moves y into z's original position in the tree, and line 20 gives y the same color as z.
  - When node y was originally black, removing or moving it could cause violations of the red-black properties, which is corrected by RB-DELETE-FIXUP in line 22.

- Line 4, 7, 11: keep track of node x
  - The procedure keeps track of node x that moves into node y's original position at the time of call.
  - Node x points to either y's only child or, if y has no children, the sentinel $T.nil$.

Difference between TREE-DELETE and RB-DELETE

- Lines 12 and 16
  - Since node x moves into node y's original position, the attribute $x.p$ must be set correctly. If node z has two children and y is z's right child, then y just moves into z's position, with x remaining a child of y. Line 12 checks for this case.
  - If y is z's right child, we still need to do one thing: set $x.p$ to y (line 16). This is because the call to RB-DELETE-FIXUP relies on $x.p$ being y even if x is $T.nil$.

- Lines 5, 8, and 13
  - Otherwise, node z is either the same as node y or it is a proper ancestor of y's original parent. In these cases, the calls to RB-TRANSPLANT set $x.p$ correctly in line 6 of RB-TRANSPLANT.

Difference between TREE-DELETE and RB-DELETE

- Lines 21-22
  - Finally, if node y was black, one or more violations of the red-black properties might arise. The call of RB-DELETE-FIXUP restores the red-black properties.

- If y was red, the red-black properties still hold when y is removed or moved, for the following reasons.
  - No black-heights in the tree have changed.
  - No red nodes have been made adjacent.
    - If z has at most one child, then y and z are the same node. That node is removed, with a child taking its place. If the removed node was red, then neither its parent nor its children can also be red, so moving a child to take its place cannot cause two red nodes to become adjacent.
    - If z has two children, then y takes z's place in the tree, along with z's color, so there cannot be two adjacent red nodes at y's new position in the tree. In addition, if y was not z's right child, then y's original right child x replaces y in the tree. Since y is red, x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
  - Because y could not have been the root if it was red, the root remains black.

RB-DELETE: Red-Black Property Violations

- If node y was black, three problems arise.
- First, if y was the root and a red child of y became the new root, property 2 is violated.
- Second, if both x and its new parent are red, then violation of property 4 occurs.
- Third, moving y within the tree causes any simple path that previously contained y to have one less black node. Thus, property 5 is now violated by any ancestor of y in the tree.

RB-DELETE: Restoring Property 5

- We can correct the violation of property 5 by saying that when the black node y is removed or moved, its blackness transfers to the node x that moves into y's original position, giving x an "extra" black.

- That is, if we add 1 to the count of black nodes on any simple path that contains x, then under this interpretation, property 5 holds.

- However, now node x is neither red nor black, violating property 1.
- Instead, node x is either "doubly black" or "red-and-black", and it contributes either 2 or 1 to the count of black nodes on simple paths containing x.
- If $x.color = RED$, it is red-and-black. If $x.color = BLACK$, it is doubly black.

RB-DELETE-FIXUP: Restoring Red-Black Properties

- RB-DELETE-FIXUP restores red-black properties 1, 2, 4.
- Here we focus on property 1. (Others are left as exercises.)

- The goal of the while loop in lines 1-43 is to move the extra black up the tree until:
  - x points to a red-and-black node, in which case line 44 colors (singly) black
  - x points to the root, in which case the extra black simply vanishes
  - having performed suitable rotations and recolorings, the loop exits.

- RB-DELETE-FIXUP handles two symmetric situations
  - Lines 3-22 for when node x is a left child.
  - Lines 24-43 for when node x is a right child.
  - We focus on lines 3-22.

RB-DELETE-FIXUP: Restoring Red-Black Properties

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left                    // is x a left child?
 3          w = x.p.right                   // w is x's sibling
 4          if w.color == RED
 5              w.color = BLACK
 6              x.p.color = RED
 7              LEFT-ROTATE(T, x.p)                          } case 1
 8              w = x.p.right
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                                } case 2
11              x = x.p
12          else
13              if w.right.color == BLACK
14                  w.left.color = BLACK
15                  w.color = RED
16                  RIGHT-ROTATE(T, w)                       } case 3
17                  w = x.p.right
18              w.color = x.p.color
19              x.p.color = BLACK
20              w.right.color = BLACK                        } case 4
21              LEFT-ROTATE(T, x.p)
22              x = T.root
```

RB-DELETE-FIXUP: Restoring Red-Black Properties

```
23      else // same as lines 3–22, but with "right" and "left" exchanged
24           w = x.p.left
25           if w.color == RED
26                w.color = BLACK
27                x.p.color = RED
28                RIGHT-ROTATE(T, x.p)
29                w = x.p.left
30           if w.right.color == BLACK and w.left.color == BLACK
31                w.color = RED
32                x = x.p
33           else
34                if w.left.color == BLACK
35                     w.right.color = BLACK
36                     w.color = RED
37                     LEFT-ROTATE(T, w)
38                     w = x.p.left
39                w.color = x.p.color
40                x.p.color = BLACK
41                w.left.color = BLACK
42                RIGHT-ROTATE(T, x.p)
43                x = T.root
44 x.color = BLACK
```

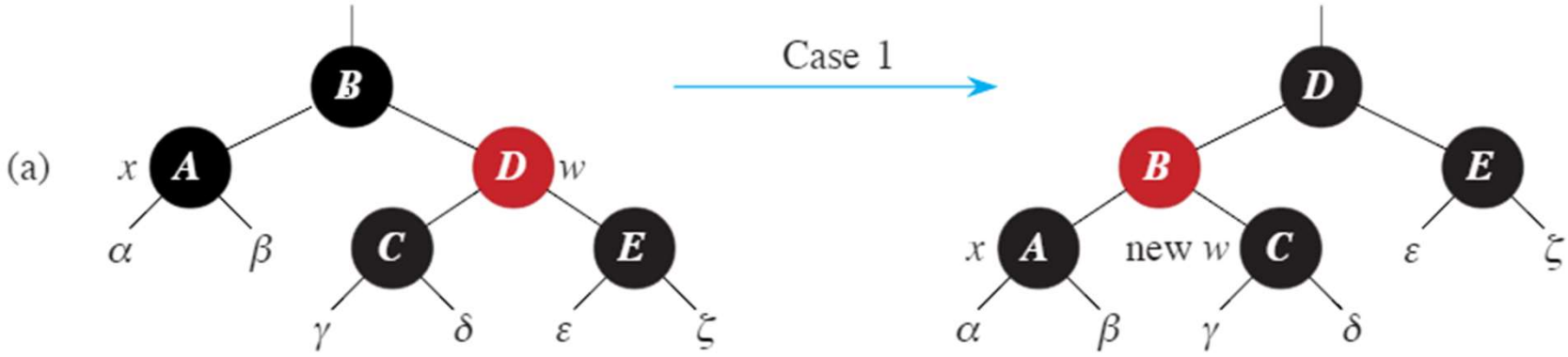
RB-DELETE-FIXUP: Restoring Red-Black Properties

- Line 1: within the while loop, x always points to a non-root doubly black node.
- Line 2: determines whether x is a left child or a right child of its parent $x.p$ so that either lines 3-22 or 24-43 will execute in a given iteration.
- Line 3: The sibling of x is always denoted by a pointer w.
  - Since node x is doubly black, node w cannot be $T.nil$, because otherwise, the number of blacks on the simple path from $x.p$ to the (singly black) leaf w would be smaller than the number on the simple path from $x.p$ to x.

- Now we meet the four cases.

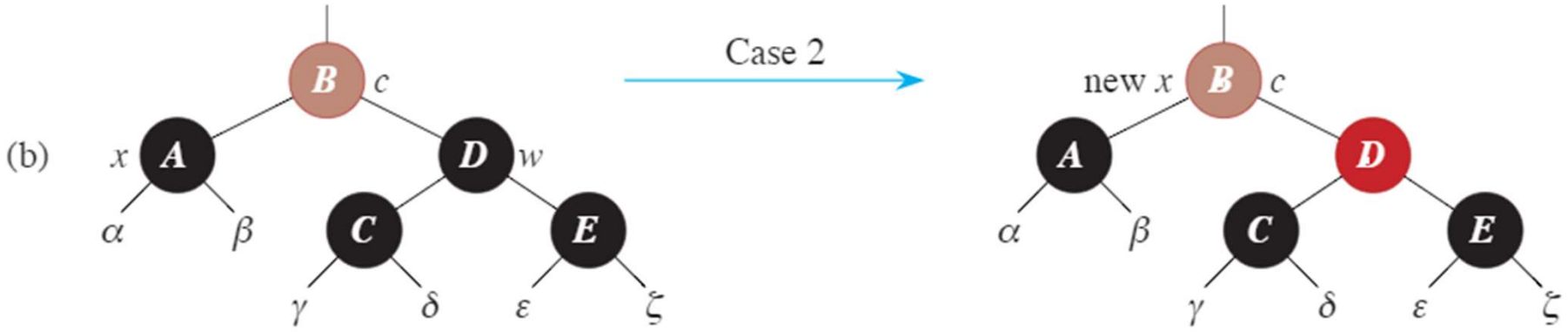
RB-DELETE-FIXUP: Restoring Red-Black Properties

- Case 1: x's sibling is red
- Because w is red, it must have black children. This case switches the colors of w and $x.p$ and then performs a left-rotation on $x.p$ without violating any of the red-black properties.
- The new sibling of x, which is one of w's children prior to the rotation, is now black, and thus case 1 converts into one of cases 2, 3, or 4.



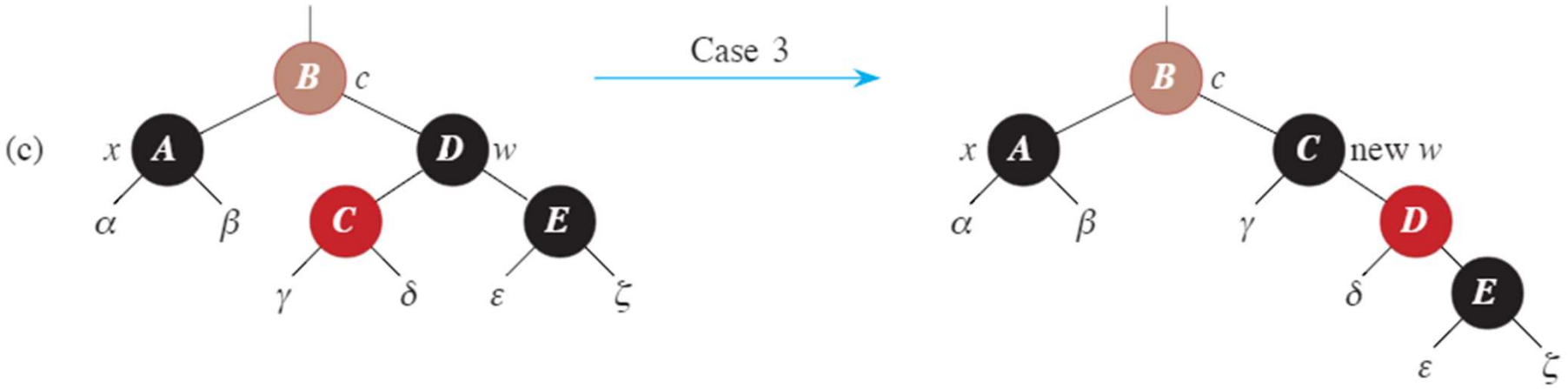
RB-DELETE-FIXUP: Restoring Red-Black Properties

- Case 2: x's sibling is black, and both of w's children are black
  - Actions in this case removes one black from both x and w, leaving x with only one black and leaving w red.
  - To compensate for x and w each losing one black, x's parent $x.p$ can take on an extra black.
  - Line 11 does so by moving up one level, so that the while loop repeats with $x.p$ as the new node x.
- If case 2 enters through case 1, the new node x is red-and-black, since the original $x.p$ was red. Hence, the color of the new node x is RED, and the loop terminates. Line 44 then colors the new node x (singly) black.



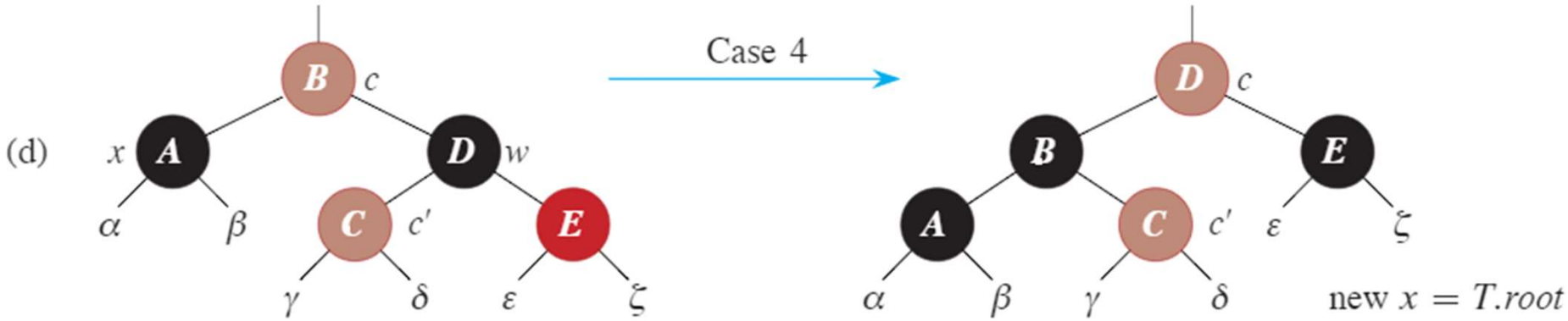
RB-DELETE-FIXUP: Restoring Red-Black Properties

- Case 3: x's sibling is black, w's left child is red, and w's right child is black
  - Actions in this case switches the colors of w and its left child $w.left$ and then performs a right rotation on w without violating any of the red-black properties.
  - The new sibling w of x is now a black node with a red right child, and thus case 3 falls into case 4.



RB-DELETE-FIXUP: Restoring Red-Black Properties

- Case 4: x's sibling is black, w's right child is red
  - Some color changes and a left rotation on $x.p$ allows the extra black on x to vanish, making it singly black, without violating any of the red-black properties.
  - set w's color as $x.p$'s color
  - set $x.p$'s color BLACK
  - set w's right child's color BLACK
  - perform left rotation on $x.p$
  - Line 22 sets x to be the root, and the while loop terminates.



RB-DELETE: running time

- The total cost of RB-DELETE without the call to RB-DELETE-FIXUP takes $O(\log n)$ time.
- Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations.
- Case 2 is the only case in which the while loop can be repeated, and then the pointer x moves up the tree at most $O(\log n)$ times, performing no rotations.
- Thus, the procedure RB-DELETE-FIXUP takes $O(\log n)$ time.
- The overall running time for RB-DELETE is therefore $O(\log n)$.

End of Class

Questions?

Instructor office: AS-1013

Email: jso1@sogang.ac.kr