Lecture 10

Algorithm Analysis

Arne Kutzner
Hanyang University / Seoul Korea

Red-Black Trees

→ 이진탐색트리

→ Balanced Binary Search Tree (균형잡힌 트리!!)

↪ 높이는 무조건 logn → 시간복잡도 : $O(\log n)$

General Definition

- A ***red-black tree*** is a binary search tree + 1 bit per node: an attribute *color*, which is either red or black.
- All leaves are empty (nil) and colored black.
- We use a single sentinel, *nil*[*T*], for all the leaves of red-black tree *T* .
  - *color*[*nil*[*T*]] is black.
- The root's parent is also *nil*[*T*].

조건 ☆

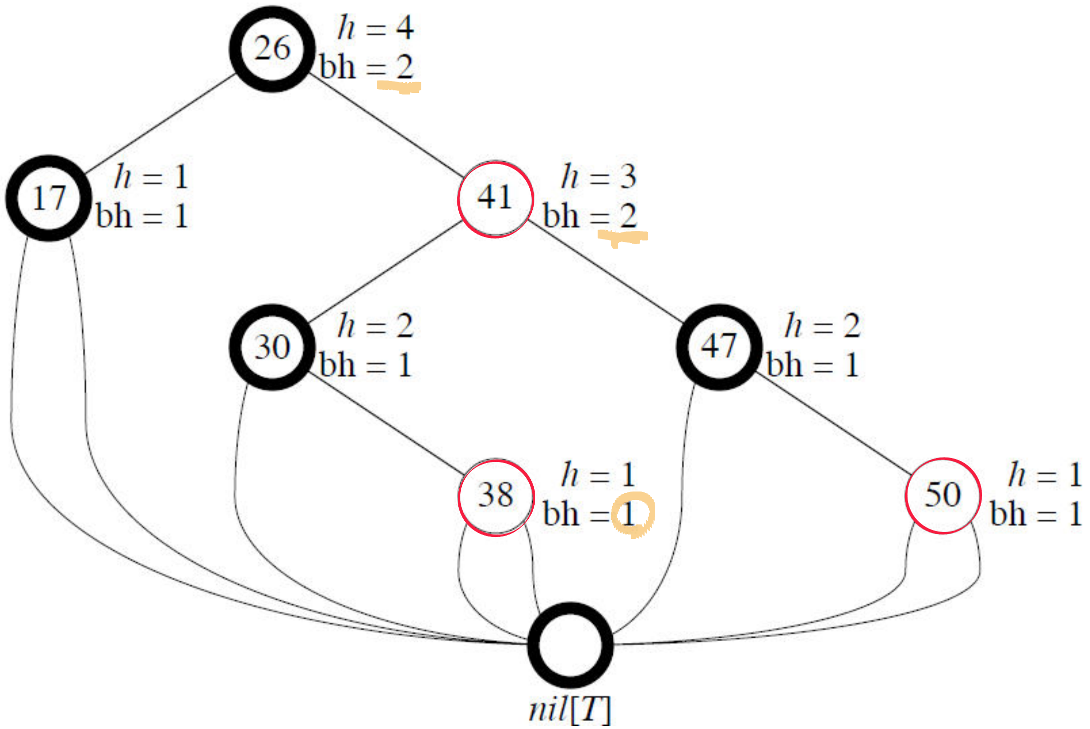
Red-Black Properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf (*nil*[*T*]) is black. 모든 external node는 검정색
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.) 빨강 노드의 자식은 검정. (No Double Red)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes. 모든 리프노드에서 Black Depth는 같다. (리프-루트 노드까지 가는 경로에서 만나는 블랙노드의 개수는 같다.)

Height

Black Height

Example



Algorithm Analysis L10.5

including nil[T]

not counting x

Height of a red-black tree

- ***Height of a node*** is the number of edges in a longest path to a leaf. log n
- ***Black-height*** of a node *x*: bh*(x)* is the number of black nodes (including *nil*[*T*]) on the path from *x* to leaf, not counting *x*. By property 5, black-height is well defined.

Upper Bound for Height

- ***Lemma 1:*** Any node with height *h* has $\boxed{\text{black-height} \geq h/2}$.
- ***Lemma 2:*** The subtree rooted at any node *x* contains $\boxed{\geq 2\cancel{bh(x)} - 1}$ internal nodes.

  $2^{bh(x)}$

  Proof: By induction on height of *x*.

Upper Bound for Height (cont.)

- ***Lemma*** A red-black tree with n internal nodes has height $\le 2 \lg(n + 1)$.

  Proof Let h and b be the height and black-height of the root, respectively. By the above two lemmas,

  $\color{red}{n \ge 2^b - 1 \ge 2^{h/2} - 1}$

  So we get $h \le 2 \lg(n + 1)$.

*

① $bh \ge h/2$

② node x contains $\ge 2^{bh(x)} - 1$ internal nodes

③ rb tree with n internal nodes has $h \le 2 \cdot \log(n+1)$

Operations on red-black trees

- The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in *O(*height*)* time. Thus, they take *O(*lg *n)* time on red-black trees.
- Insertion and deletion are not so easy.
- If we insert, what color to make the new node?

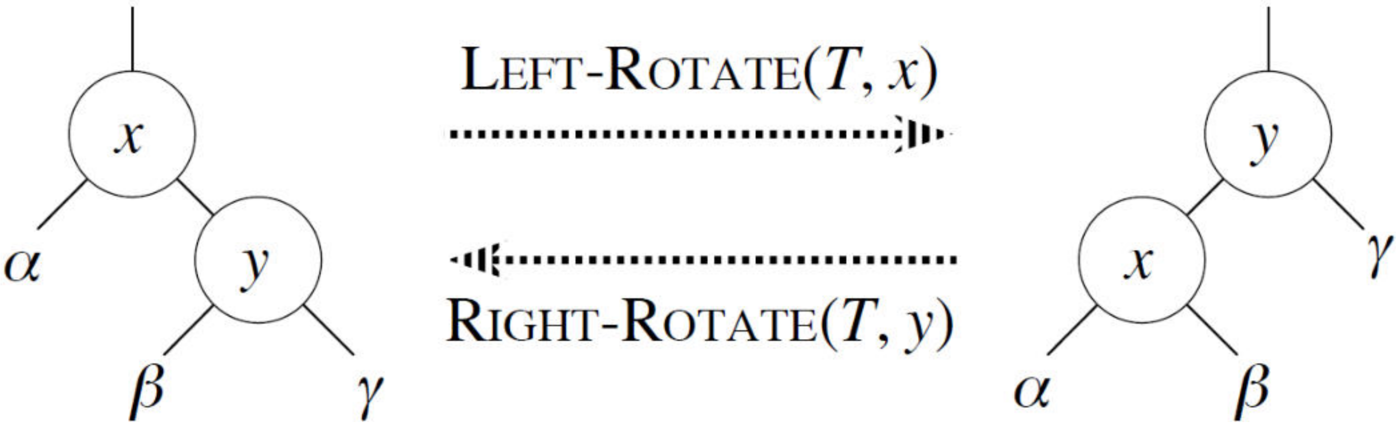
* successor :
대상 노드의 오른쪽 서브트리의 최솟값.

predecessor :
대상 노드의 왼쪽 서브트리의 최댓값.

Rotations : 한 노드를 중심으로 부분적으로 트리의 모양을 수정하는 연산

- The basic tree-restructuring operation.
- Needed to ~~maintain red-black trees~~ as balanced binary search trees.
- Changes the local pointer structure. (Only pointers are changed.)
- Won.t upset the binary-search-tree property.
- Have both left rotation and right rotation. They are inverses of each other.
- A rotation takes a red-black-tree and a node within the tree.

Rotations (cont.)



α, β, γ → 서브트리

Rotations (Pseudocode)

LEFT-ROTATE(T, x)

```
y ← right[x]                ▷ Set y.
right[x] ← left[y]          ▷ Turn y's left subtree into x's right subtree.
if left[y] ≠ nil[T]
   then p[left[y]] ← x
p[y] ← p[x]                 ▷ Link x's parent to y.
if p[x] = nil[T]
   then root[T] ← y
   else  if x = left[p[x]]
            then left[p[x]] ← y
            else  right[p[x]] ← y
left[y] ← x                 ▷ Put x on y's left.
p[x] ← y
```

Rotations Complexity

- ***Time:*** $O(1)$ for both LEFT-ROTATE and RIGHT-ROTATE, since a constant number of pointers are modified
- ***Notes:***
  Rotation is a very basic operation, also used in AVL trees and splay trees.

O(1)

Insertion

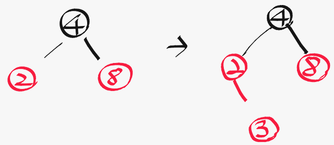
·

$\text{RB-INSERT}(T, z)$
$y \leftarrow nil[T]$
$x \leftarrow root[T]$
while $x \neq nil[T]$
  do $y \leftarrow x$
    if $key[z] < key[x]$
      then $x \leftarrow left[x]$
      else $x \leftarrow right[x]$
$p[z] \leftarrow y$
if $y = nil[T]$
  then $root[T] \leftarrow z$
  else **if** $key[z] < key[y]$
      then $left[y] \leftarrow z$
      else $right[y] \leftarrow z$
$left[z] \leftarrow nil[T]$
$right[z] \leftarrow nil[T]$
$color[z] \leftarrow \text{RED}$
$\boxed{\text{RB-INSERT-FIXUP}(T, z)}$

ex)

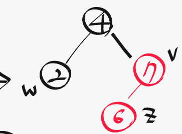
1) Root Property에 의해 루트 노드 색 → 검정 ④

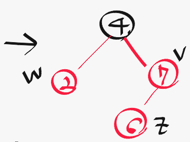
2) 값들 삽입 (삽입되는 노드의 색은 무조건 RED!! → Double Red 생김)



조건 위배!!

3) Double Red 문제해결 → 현재 insert 된 노드의 uncle node 색깔에 따라

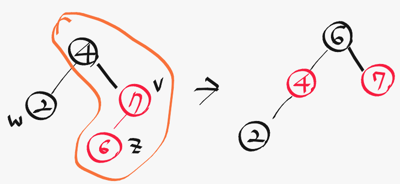
[1] Restructuring →  인 경우

[2] Recoloring →  인 경우.

[1] Restructuring : z와 v와 v의 부모를 오름차순으로 정렬

→ 무조건 가운데 있는 값을 부모로 만들고 나머지 둘을 자식으로

→ Black → Red

∴ 

O(logn)
한번에 끝남!!

[2] Recoloring : z의 부모와 그 형제 w를 검정으로 하고 부모의 부모를 Red로 한다.

→ 부모의 부모가 Root 가 아닐때 Double Red 다시 발생할수도 있다.

Root면 red → black

→ O(logn)

Properties of RB-INSERT

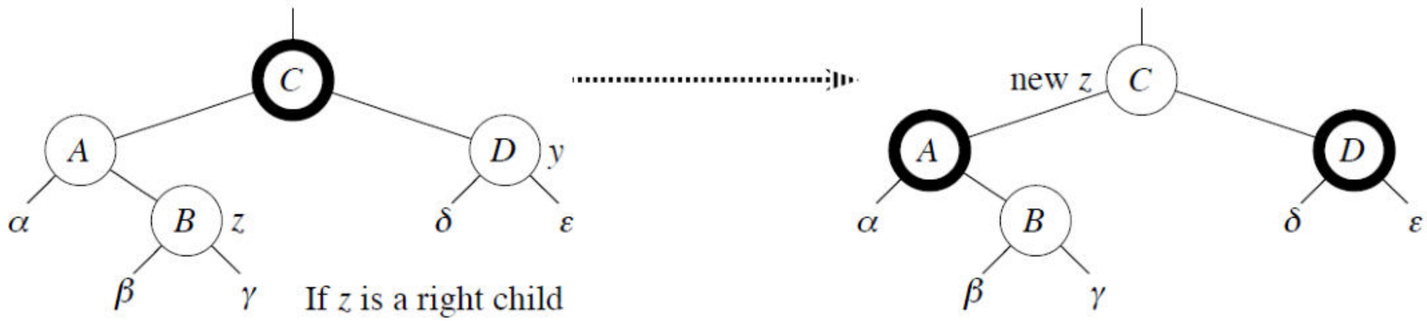
- RB-INSERT ends by coloring the new node *z* red.
- Then it calls RB-INSERT-FIXUP because we could have violated a red-black property.
- Which property might be violated?
  1. OK.
  2. If *z* is the root, then there.s a violation. Otherwise, OK.
  3. OK.
  4. **If *p*[*z*] is red, there is a violation: both *z* and *p*[*z*] are red.**
  5. OK.

Pseudocode FIXUP

```
RB-INSERT-FIXUP(T, z)
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
          then y ← right[p[p[z]]]
               if color[y] = RED
                 then color[p[z]] ← BLACK                    ▷ Case 1
                      color[y] ← BLACK                       ▷ Case 1
                      color[p[p[z]]] ← RED                   ▷ Case 1
                      z ← p[p[z]]                            ▷ Case 1
                 else  if z = right[p[z]]
                         then z ← p[z]                       ▷ Case 2
                              LEFT-ROTATE(T, z)              ▷ Case 2
                      color[p[z]] ← BLACK                    ▷ Case 3
                      color[p[p[z]]] ← RED                   ▷ Case 3
                      RIGHT-ROTATE(T, p[p[z]])               ▷ Case 3
          else  (same as then clause
                       with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

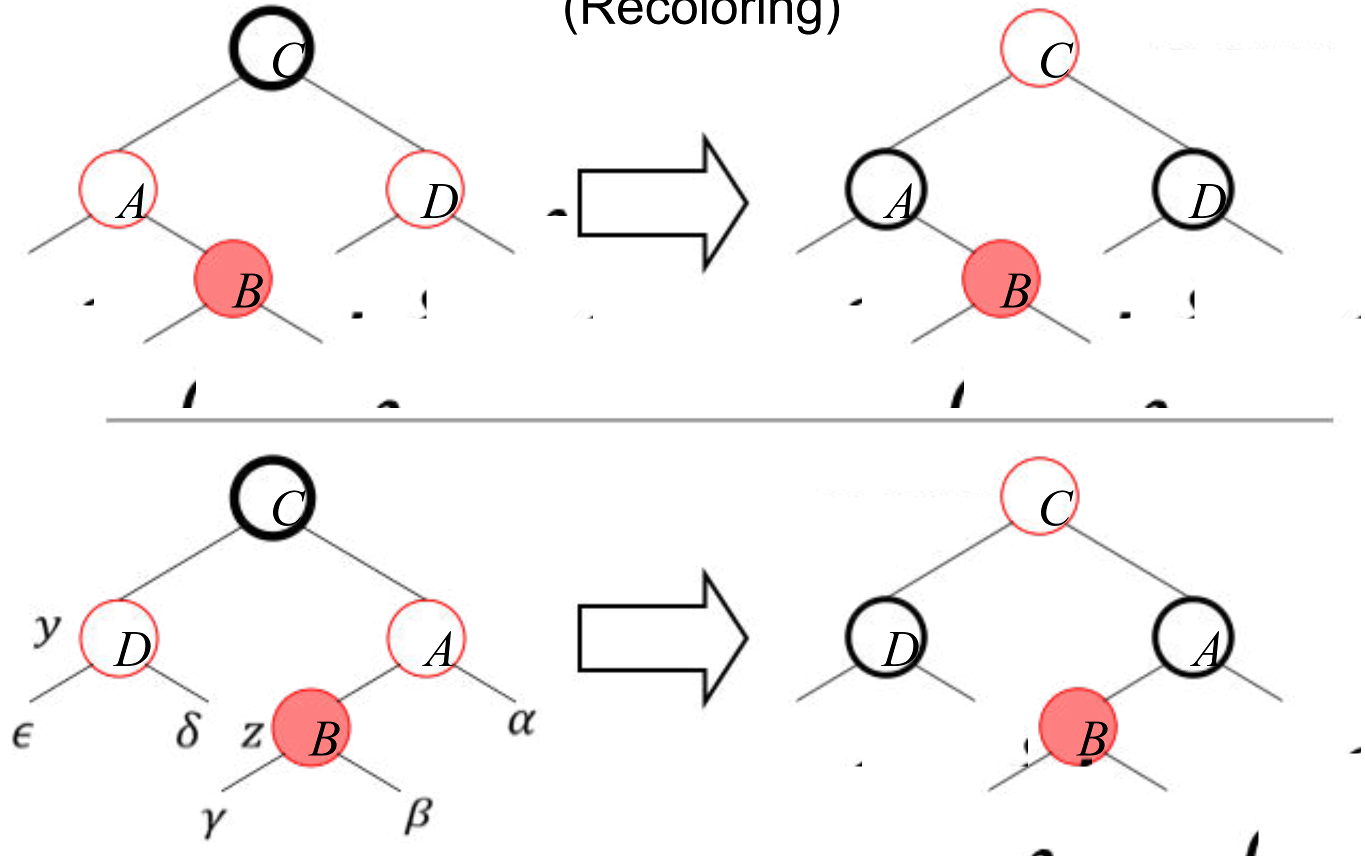
recoloring

Case 1: *y (z's uncle)* is red



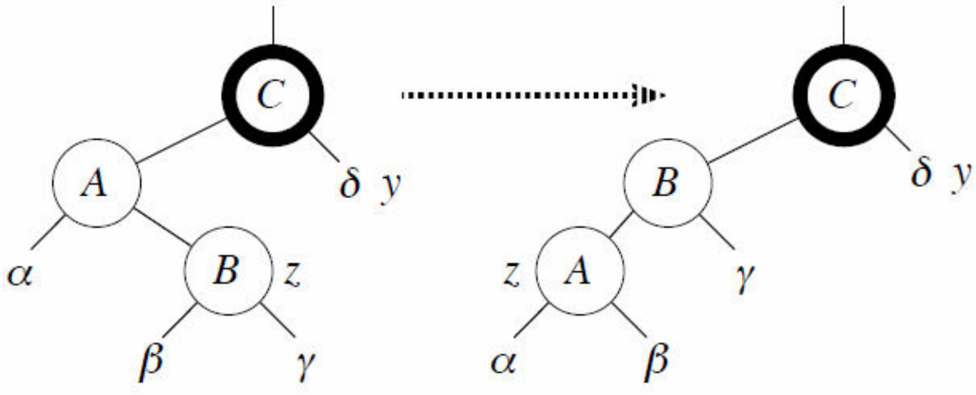
- *p*[*p*[*z*]] (*z*.s grandparent) must be black, since *z* and *p*[*z*] are both red and there are no other violations of property 4.
- Make *p*[*z*] and *y* black ⇒ now *z* and *p*[*z*] are not both red. But property 5 might now be violated.
- Make *p*[*p*[*z*]] red ⇒ restores property 5.
- The next iteration has *p*[*p*[*z*]] as the new *z* (i.e., *z* moves up 2 levels).
- There are 4 variants of case 1

Case 1:
(Recoloring)



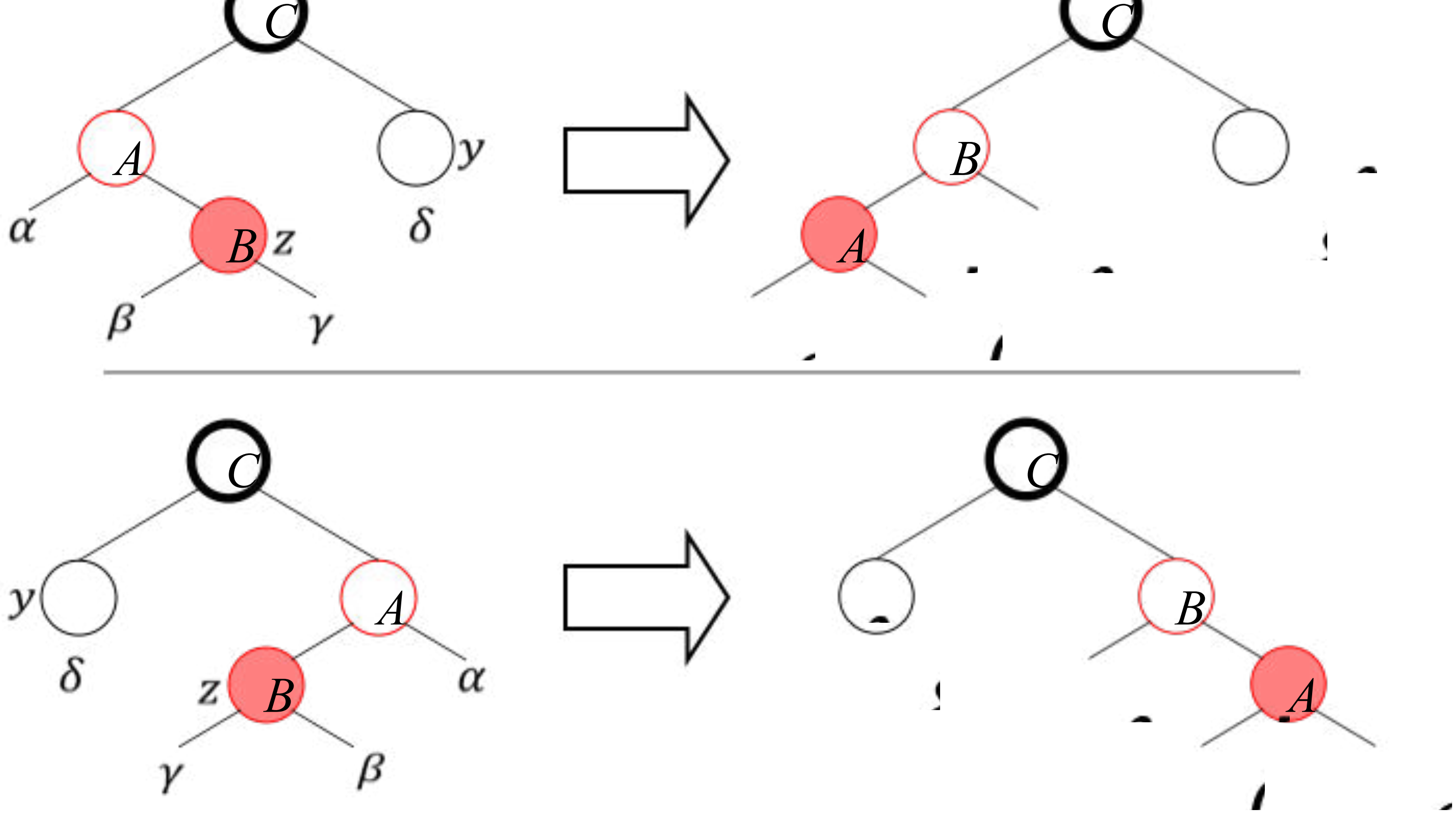
restructuring

Case 2: *y* is black, *z* is a right child

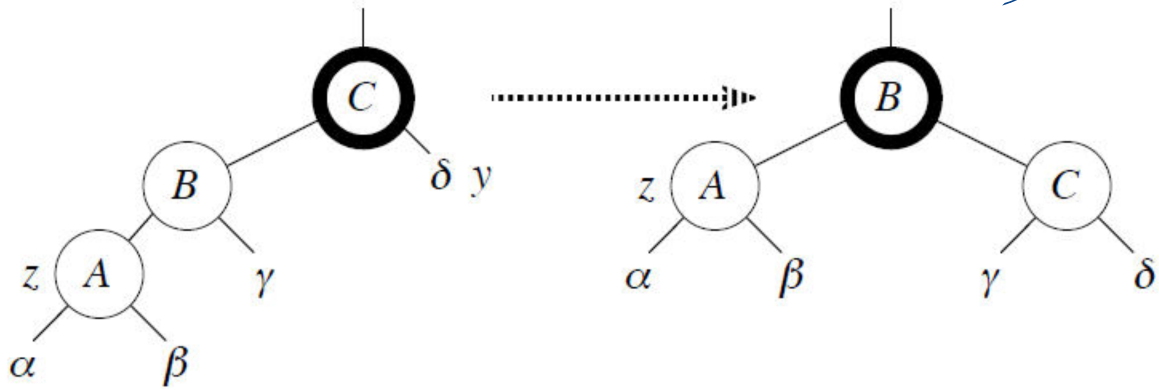


- Left rotate around $p[z] \Rightarrow$ now z is a left child, and both z and $p[z]$ are red.
- Takes us immediately to case 3.

Case 2:
(Rotation)

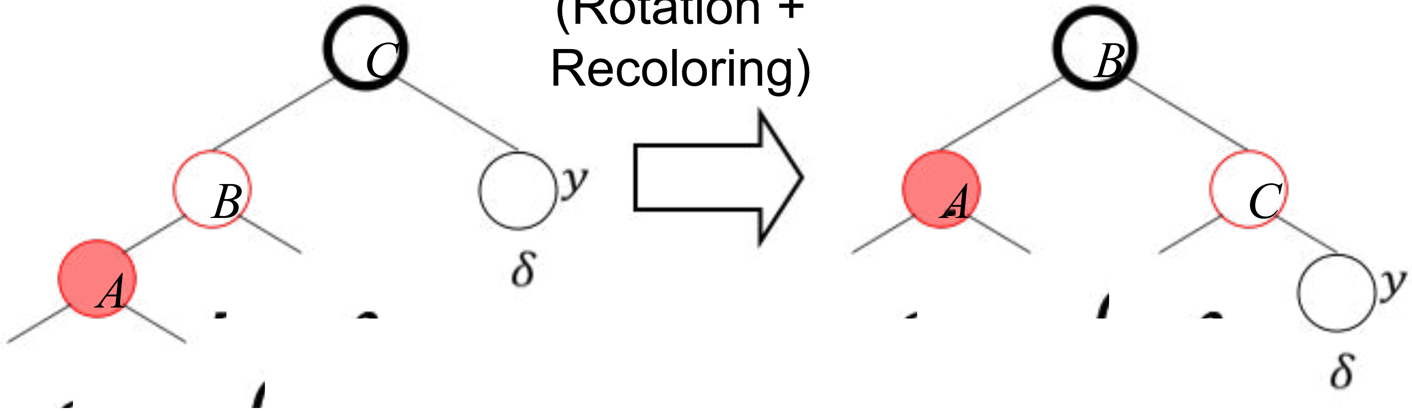


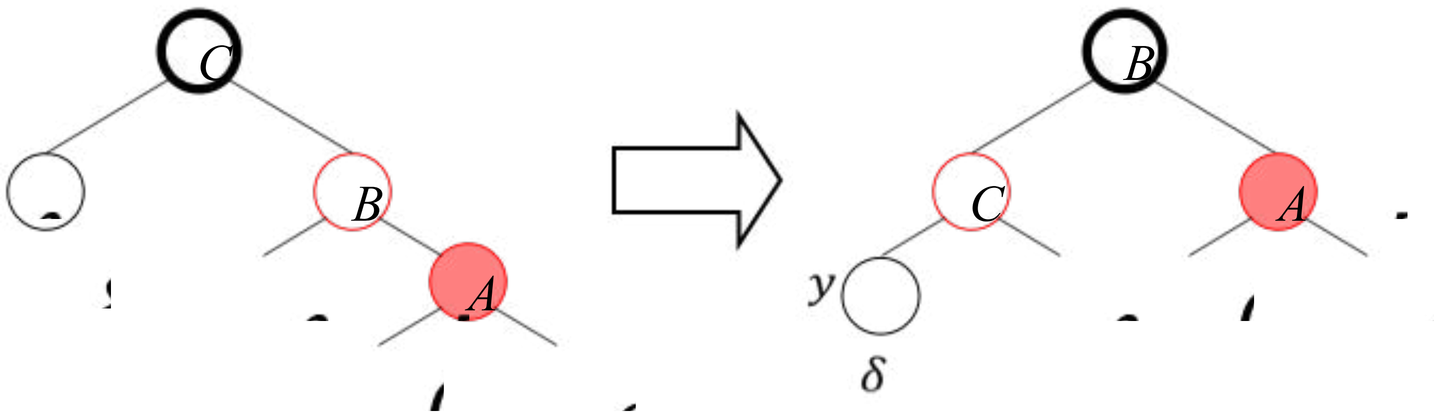
Case 3: *y* is black, *z* is a left child



- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate on $p[p[z]]$.
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.

Case 3:
(Rotation +
Recoloring)





Algorithm Analysis L10.22

1?

Analysis

- $O(\lg n)$ time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.
- Within RB-INSERT-FIXUP:
  - Each iteration takes $O(1)$ time.
  - Each iteration is either the last one or it moves z up 2 levels. z가 2레벨 상승
  - $O(\lg n)$ levels $\Rightarrow$ $O(\lg n)$ time.
  - Also note that there are at most 2 rotations overall.
- Thus, insertion into a red-black tree takes $O(\lg n)$ time.

★ 트리의 높이에 비례하는 시간복잡도.

Correctness

Loop invariant:

At the start of each iteration of the **while** loop,

1. *z* is red.
2. There is at most one red-black violation:
   - Property 2: *z* is a red root, or
   - Property 4: *z* and *p*[*z*] are both red.

- **Initialization:** loop invariant holds initially.
- **Termination:** The loop terminates because *p*[*z*] is black. Hence, property 4 is OK. Only property 2 might be violated, and the last line fixes it.
- **Maintenance:** We drop out when *z* is the root (since then *p*[*z*] is the sentinel *nil*[*T*], which is black). When we start the loop body, the only violation is of property 4.

Deletion

```
RB-DELETE(T, z)
if left[z] = nil[T] or right[z] = nil[T]
   then y ← z
   else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T]
   then x ← left[y]
   else x ← right[y]
p[x] ← p[y]
if p[y] = nil[T]
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
if color[y] = BLACK
   then RB-DELETE-FIXUP(T, x)
return y
```

Deletion - Remarks

- *y* is the node that was actually spliced out.
- *x* is
  - either y's sole non-sentinel child before *y* was spliced out,
  - or the sentinel, if *y* had no children.
  - In both cases, *p*[*x*] is now the node that was previously *y*'s parent.

y가 red → 그냥 DELETE

3 possible Problems

위반

- If *y* is $\boxed{\text{black}}$, we could have violations of red-black properties:

1. OK.
2. If *y* is the root and *x* is red, then the root has become red. (루트노드가 black 이라는 규칙위반)
3. OK.
4. Violation if *p*[*y*] and *x* are both red. (레드노드 불연속 규칙위반)
5. Any path containing *y* now has 1 fewer black node. > extra black 부여

룰: 모든 노드에대해서 그 노드로부터 자손인 리프노드에 이르는 모든 경로에 동일한 개수의 Black 노드 존재함.

→ 원래 y를 포함했던 모든 경로는 이제 Black 노드가 하나 부족.

⇒ 노드 x에 "extra black" 부여 → double black 혹은 red/black 이 됨.

How to Fix ? Idea:

- Add 1 to count of black nodes on paths containing *x*. //
- Now property 5 is OK, but property 1 is not (1: 각노드는 r or b)
- *x* is either ***doubly black*** (if *color*[*x*] = BLACK) or ***red & black*** (if *color*[*x*] = RED)
- The attribute *color*[*x*] is still either RED or BLACK. No new values for *color* attribute.

Pseudocode for Fixup

```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
          then w ← right[p[x]]
               if color[w] = RED
                 then color[w] ← BLACK                          ▷ Case 1
                      color[p[x]] ← RED                         ▷ Case 1
                      LEFT-ROTATE(T, p[x])                      ▷ Case 1
                      w ← right[p[x]]                           ▷ Case 1
               if color[left[w]] = BLACK and color[right[w]] = BLACK
                 then color[w] ← RED                            ▷ Case 2
                      x ← p[x]                                  ▷ Case 2
                 else if color[right[w]] = BLACK
                         then color[left[w]] ← BLACK            ▷ Case 3
                              color[w] ← RED                    ▷ Case 3
                              RIGHT-ROTATE(T, w)                ▷ Case 3
                              w ← right[p[x]]                   ▷ Case
                      color[w] ← color[p[x]]                    ▷ Case
                      color[p[x]] ← BLACK                       ▷ Case 4
                      color[right[w]] ← BLACK                   ▷ Case 4
                      LEFT-ROTATE(T, p[x])                      ▷ Case 4
                      x ← root[T]                               ▷ Case 4
          else (same as then clause with "right" and "left" exchanged)
color[x] ← BLACK
```

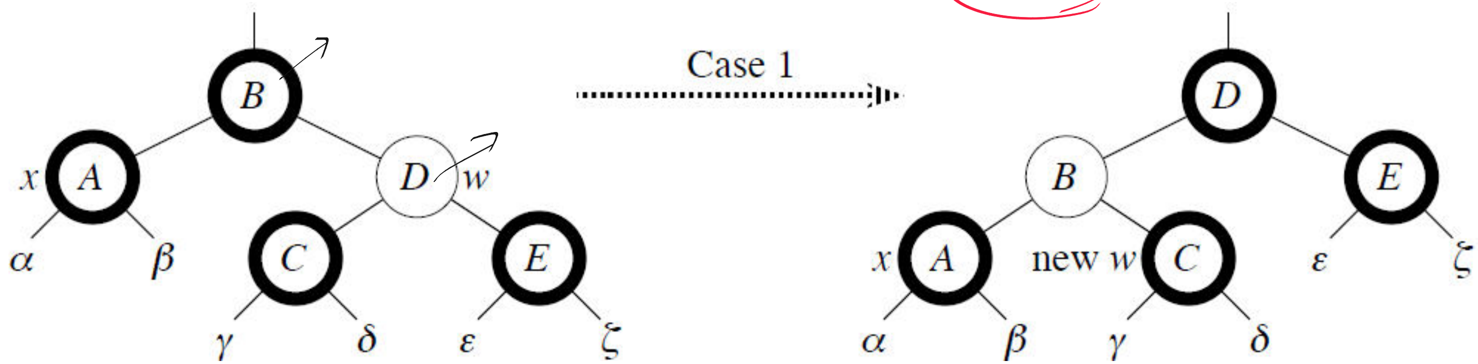
RB-DELETE-FIXUP Remarks

extra black을 트리의 위쪽으로 이동시킴

- ***Idea:*** Move the extra black up the tree until
  - *x* points to a red & black node ⇒ turn it into a black node, x가 r&b 되면 → x를 그냥 black노드로
  - *x* points to the root ⇒ just remove the extra black, or
- Within the **while** loop:
  - *x* always points to a nonroot doubly black node.
  - *w* is *x*'s sibling.
  - *w* cannot be *nil*[*T*], since that would violate property 5 at *p*[*x*].

x는 y의 자식.

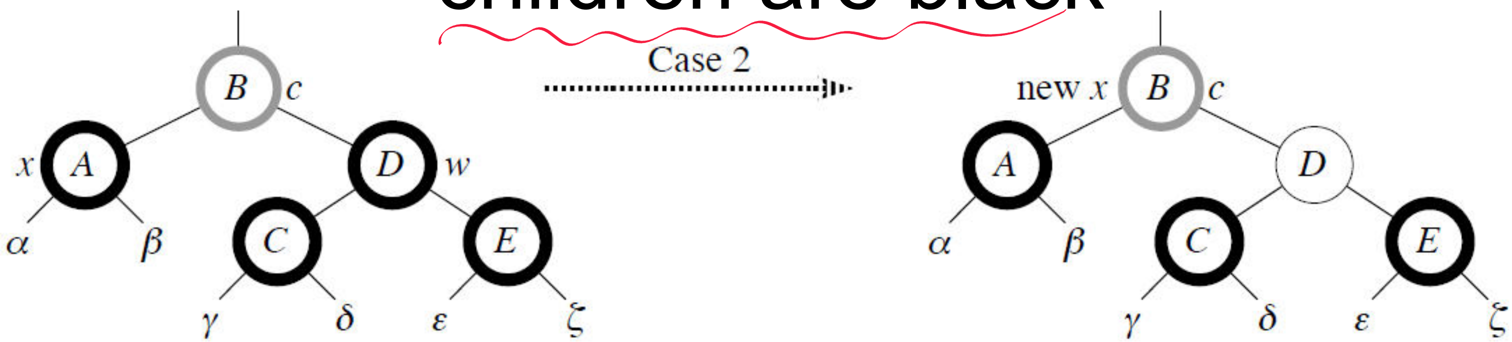
Case 1: *w* is red



- *w* must have black children.
- Make *w* black and *p*[*x*] red.
- Then left rotate on *p*[*x*].
- New sibling of *x* was a child of *w* before rotation ⇒ must be black.
- Go immediately to case 2, 3, or 4.

Case 1 : X의 부모노드에 대해서 left-rotation을 적용하면, 새로운 w 노드가 black이 되기 때문에 case 2,3,4로 넘어가게된다.

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

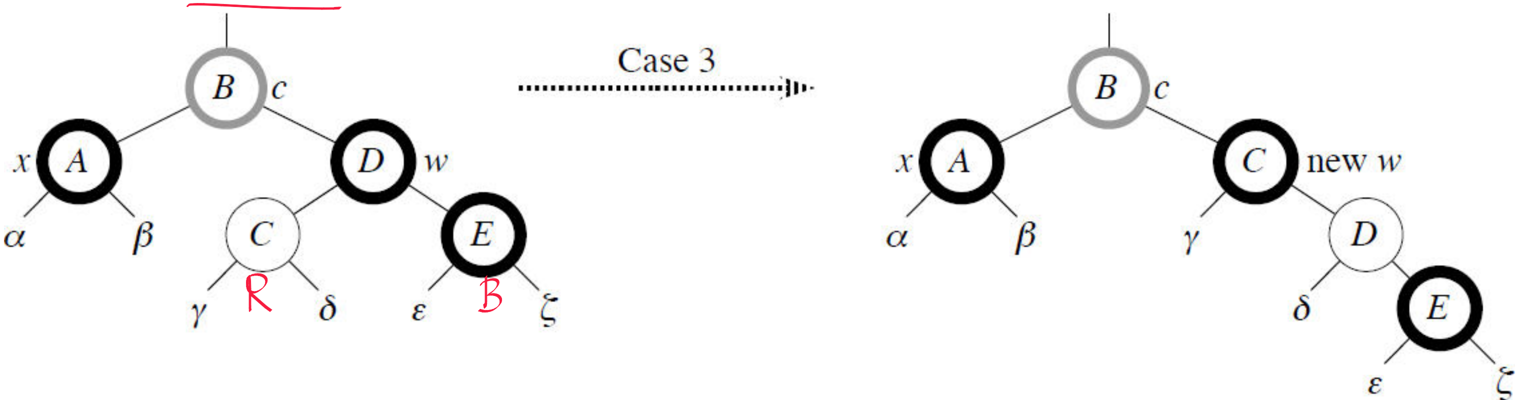


- Take 1 black off x (⇒singly black) and off w (⇒red). x와 w로부터 black 하나씩 빼어서 → p[x]
- Move that black to $p[x]$.
- Do the next iteration with $p[x]$ as the new x.
- If entered this case from case 1, then $p[x]$ was red
  ⇒ new x is red & black
  ⇒ color attribute of new x is RED ⇒ loop terminates. 중요
  Then new x is made black in the last line.

new x (p[x])가 red면 → new x = red&black 으로 black으로 변환후 종료.

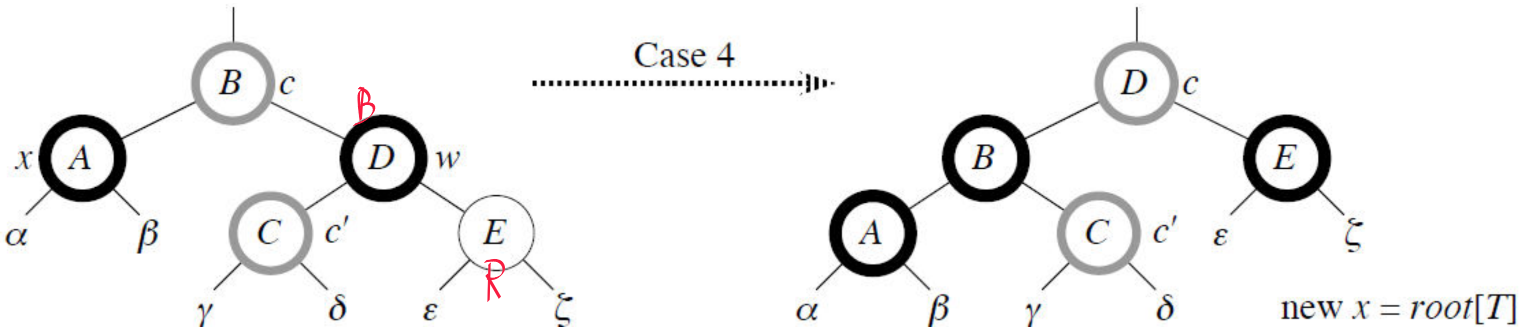
black이면 → new x = double black → loop

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



- Make *w* red and *w*'s left child black.
- Then right rotate on *w*.
- New sibling *w* of *x* is black with a red right child ⇒ case 4.

Case 4: *w* is black, *w*'s left child is black, and *w*'s right child is red



- Make w be $p[x]$'s color (c). W 색을 P[X]의 색으로
- Make $p[x]$ black and w's right child black.
- Then left rotate on $p[x]$.
- Remove the extra black on x ($\Rightarrow$ x is now singly black) without violating any red-black properties.
- All done. Setting x to root causes the loop to terminate.

Analysis - Deletion

- $O(\lg n)$ time to get through RB-DELETE up to the call of RB-DELETE-FIXUP
- Within RB-DELETE-FIXUP:
- Case 2 is the only case in which more iterations occur
  - x moves up 1 level
  - Hence, $O(\lg n)$ iterations
- Each of cases 1, 3, and 4 has 1 rotation $\Rightarrow \leq 3$ rotations in all
- Hence, $O(\lg n)$ time