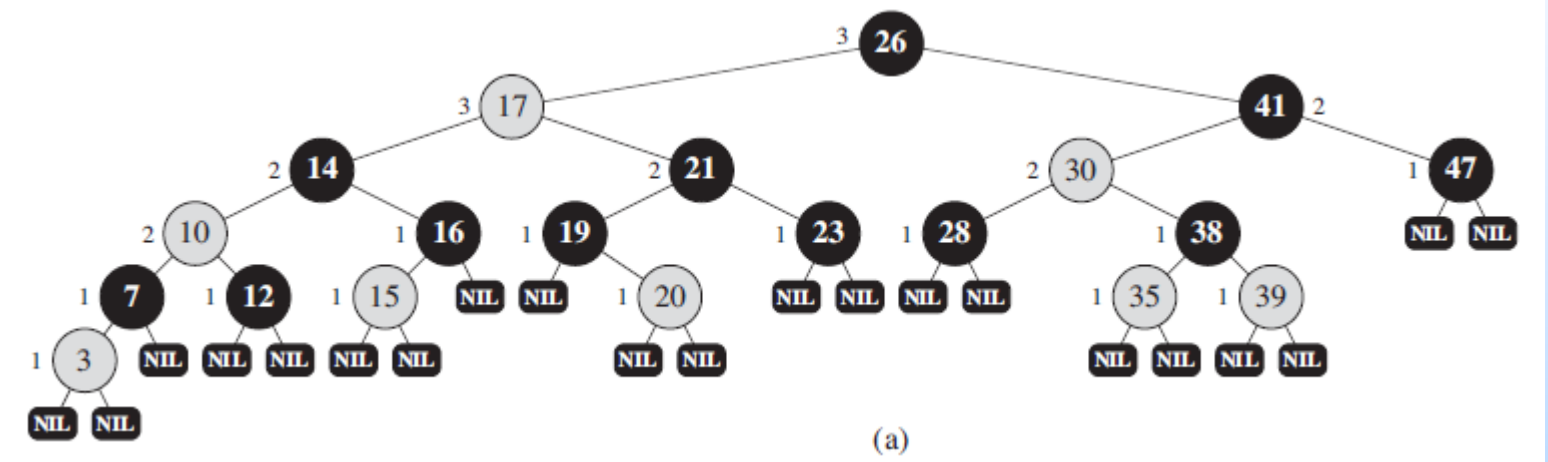
Red-Black Trees
(Chapter 13)

김동진
(NHN NEXT)

동일한 Black Height 확인 문제

- 상황
  - Tree의 각 노드의 색깔은 red 혹은 black이다.
  - Leaf node가 가리키는 NULL은 모든 black으로 간주한다.



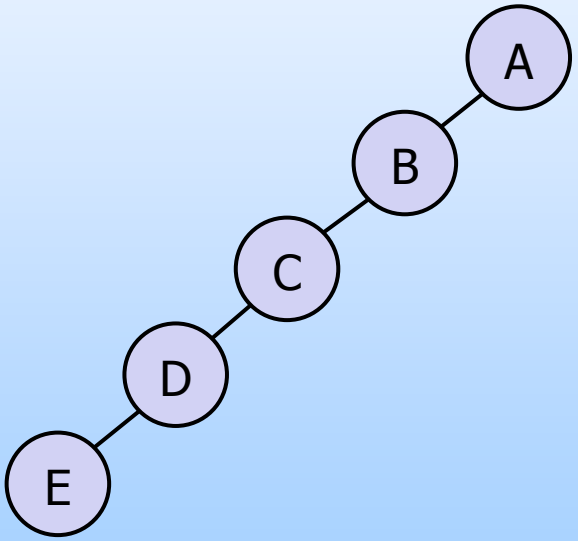
- 주어진 tree가 아래 조건을 만족하는지 검사하는 코드를 작성하시오.
  - 각 노드에 대해서 노드로부터 descendant leaves까지의 단순 경로는 모두 같은 수의 black nodes들을 포함하면 경로상의 black node 개수를 return하고 그렇지 않으면 -1을 return.
  - int checkBlackHeight(node_t *node);

목 차

- 소개
- Rotation
- Insertion
- Deletion

필요성

- Binary Search Tree의 약점 해결
  - Worst case에 O(n)의 search, insert, delete 연산
  - Skewed tree가 worst case에 해당됨

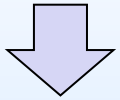


목 표

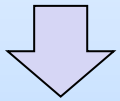
- Search
  - worst case에 O(lg n)
- Insert
  - worst case에 O(lg n)
- Delete
  - worst case에 O(lg n)

아이디어

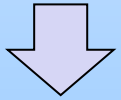
Binary Search Tree의 연산 시간은 tree의 depth에 비례한다.



동일한 노드 개수일 때 depth를 최소화하자.



가능한 최소 depth는 complete binary tree 구조일 때이며 $\lg n$이다



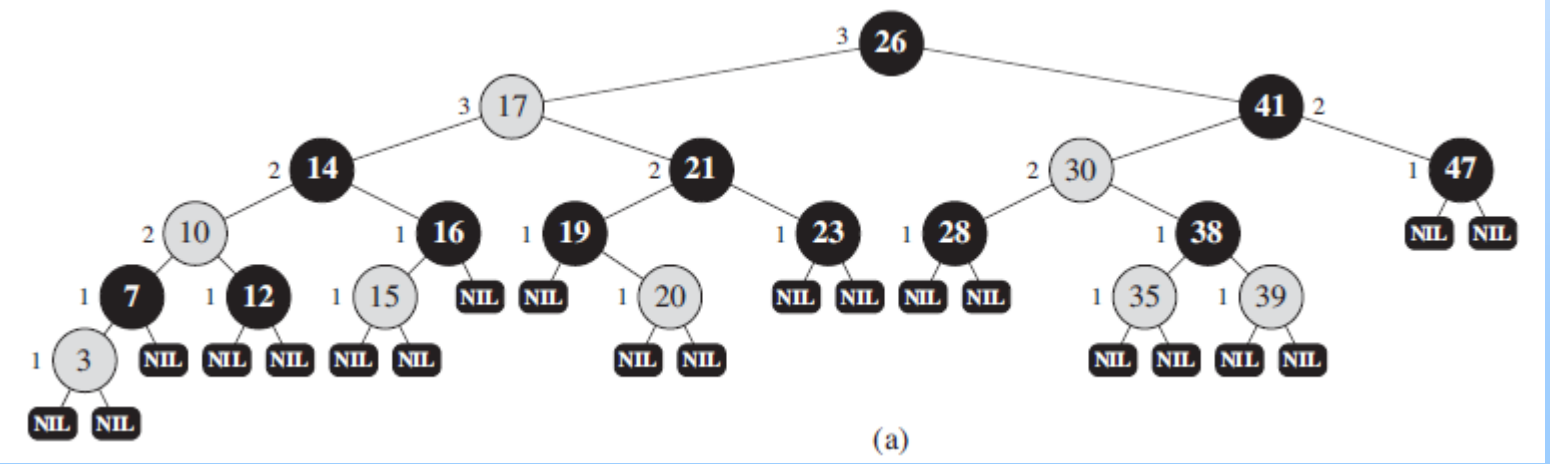
Root의 left subtree와 right subtree의 depth 비율을
상수 값 이내로 유지하자.

Red-black Tree의 특징

- Binary Search Tree이다.
- 각 노드는 Red 혹은 Black 색깔을 가진다.
- Root부터 leaf까지의 모든 경로 중 최소 경로와 최대 경로의 크기 비율은 2보다 크지 않다.
  - Balanced 상태
- 노드의 child가 없을 경우 child를 가리키는 포인터는 NIL 값을 저장한다.
  - 이러한 NIL들을 leaf node로 간주한다.

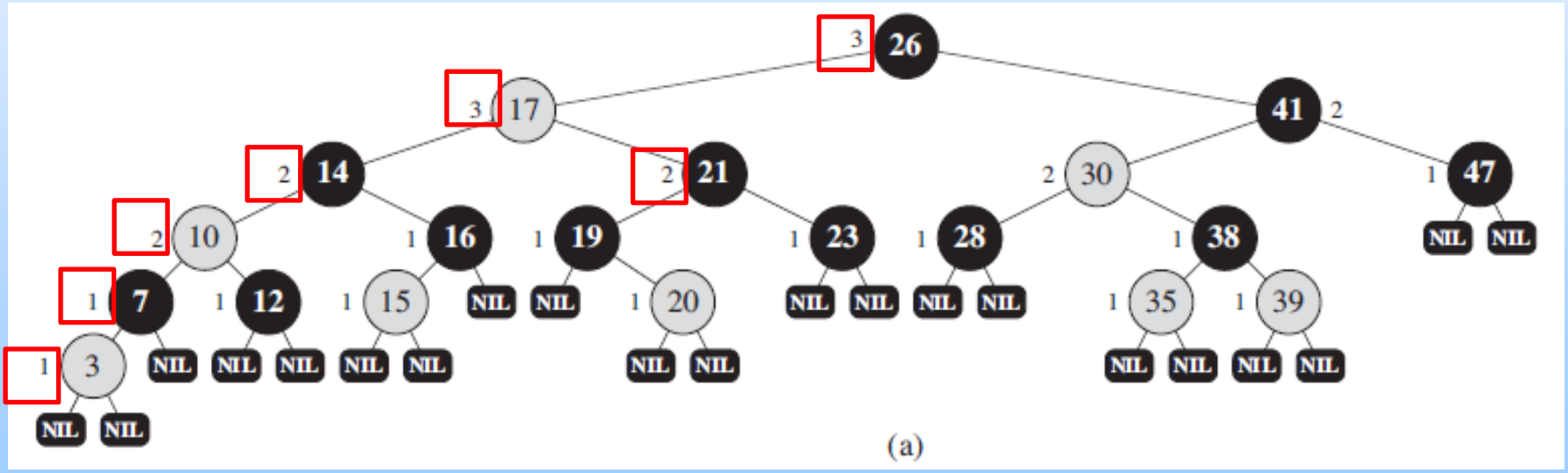
정 의

- Red-black tree는 다음의 성질을 만족하는 binary search tree이다.
  1. 각 노드는 red 혹은 black이다.
  2. Root는 black이다.
  3. 각 leaf(NIL)은 black이다.
  4. 노드가 red라면 두 개의 children은 모두 black이다.
  5. 각 노드에 대해서 노드로부터 descendant leaves까지의 단순 경로는 모두 같은 수의 black nodes들을 포함하고 있다.

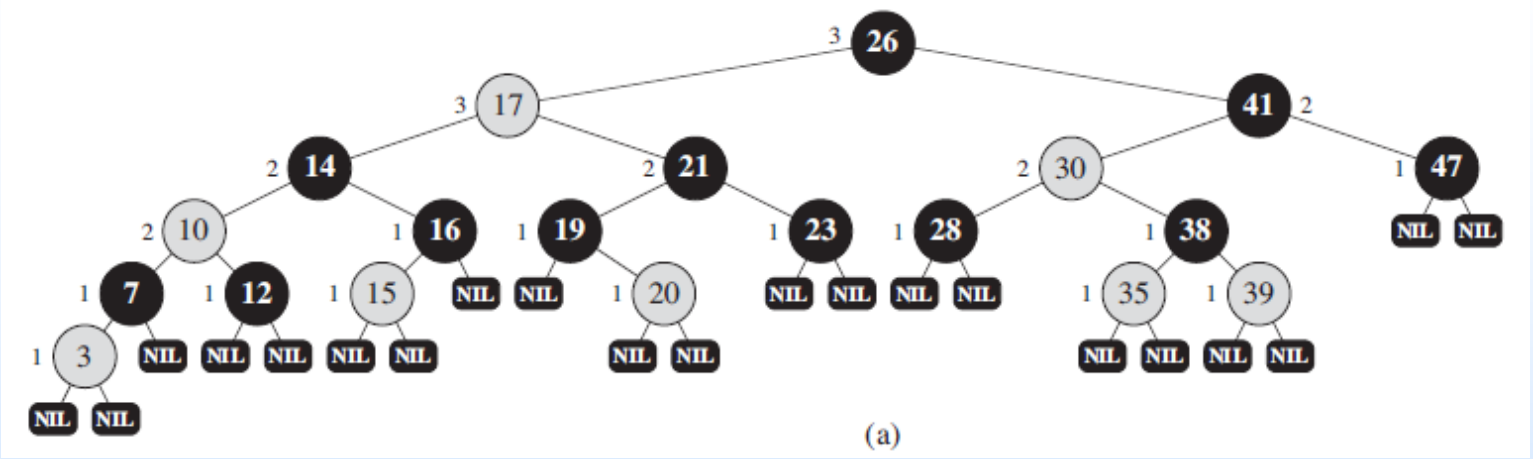


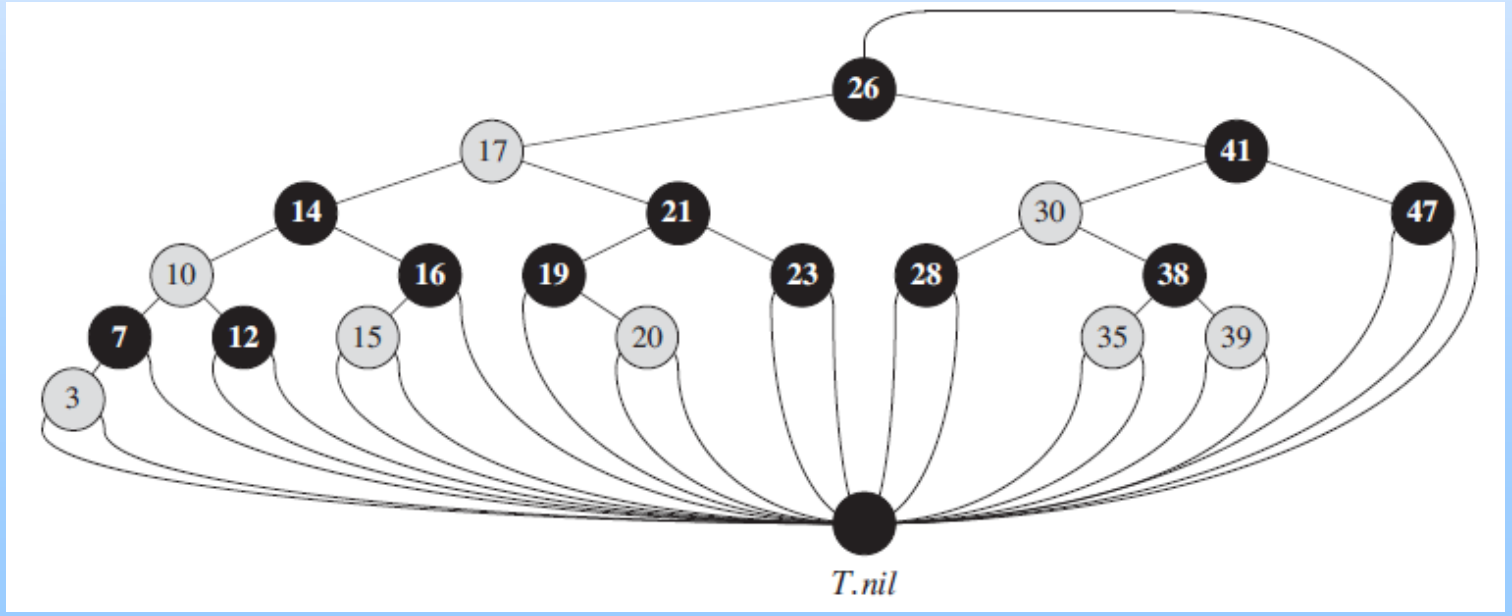
Black-Height

- 노드 x로부터 노드 x를 포함하지 않은 leaf node까지의 simple path 상에 있는 black nodes의 개수
  - 노드 옆에 적힌 숫자
- $bh(x)$로 표기

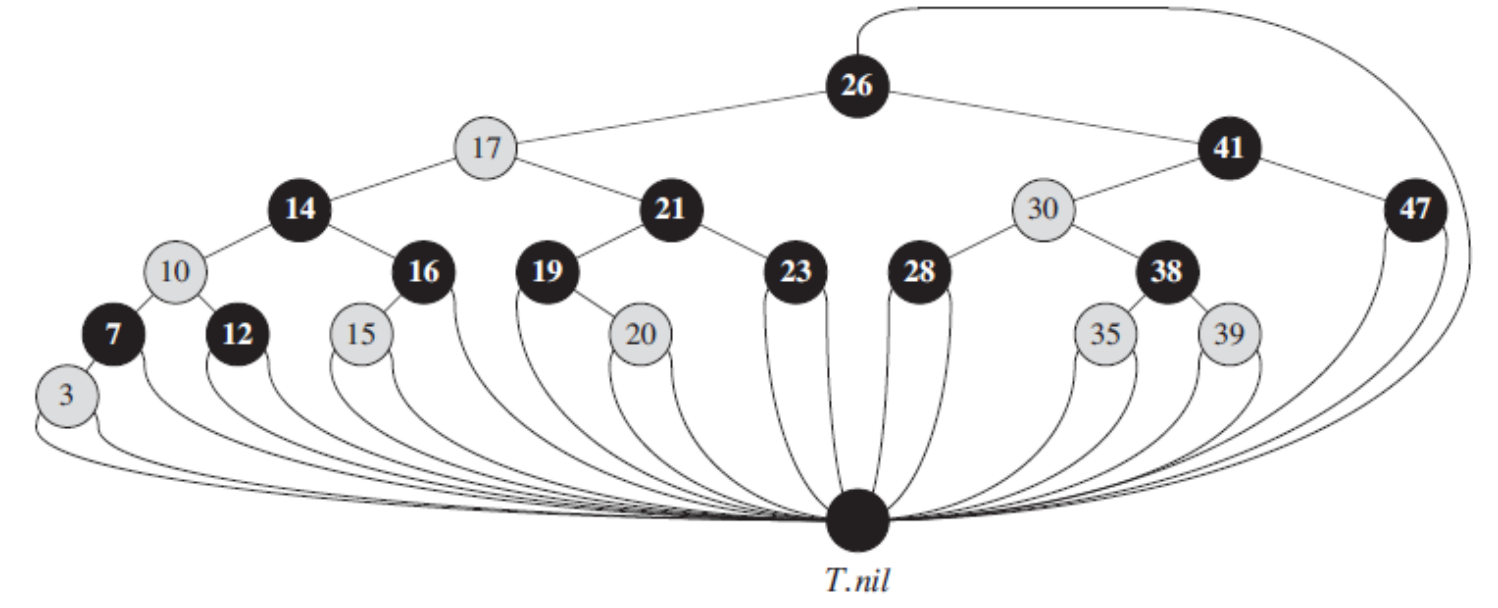


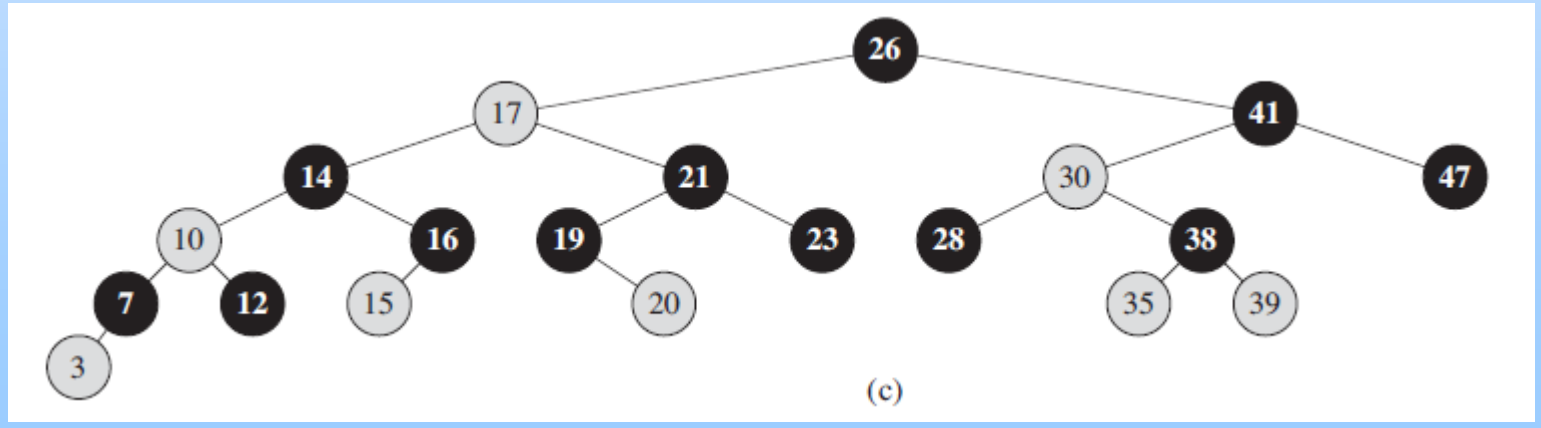
Sentinel 사용하는 표현 방법





Sentinel 표시하지 않은 방법

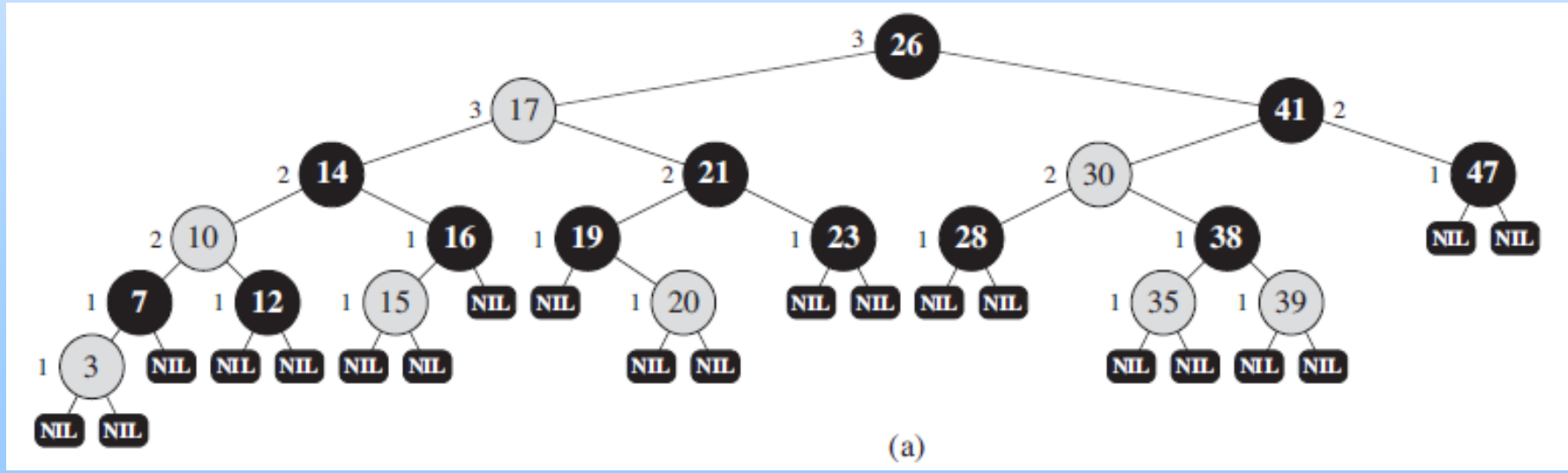




Red-Black Tree의 height (1)

- Lemma 13.1
  - n개의 internal node로 구성되는 red-black tree의 height h는 최대 $2\lg(n+1)$이다.

- 예제 트리
  - Internal 노드 개수: 20개
  - Height: 6 ($\leq 2\lg(21)$)



Red-Black Tree의 height (2)

- Lemma 13.1의 의미
  - N개의 node로 구성되는 tree의 height
    - $O(\lg n)$
  - Search/insert/delete 연산 시간
    - $O(\lg n)$

Red-Black Tree의 height (3)

- Lemma 13.1의 증명
  - 노드 x를 root로 하는 subtree의 internal node 개수 n는 최소 $2^{bh(x)} - 1$개인 것을 증명
    - 증명되었다고 가정하면

$$2^{bh(x)} - 1 \leq n \Rightarrow bh(x) \leq \lg(n+1)$$

  - Red-black tree의 property 4(red node의 child는 black)를 이용하여 height와 black height 비교

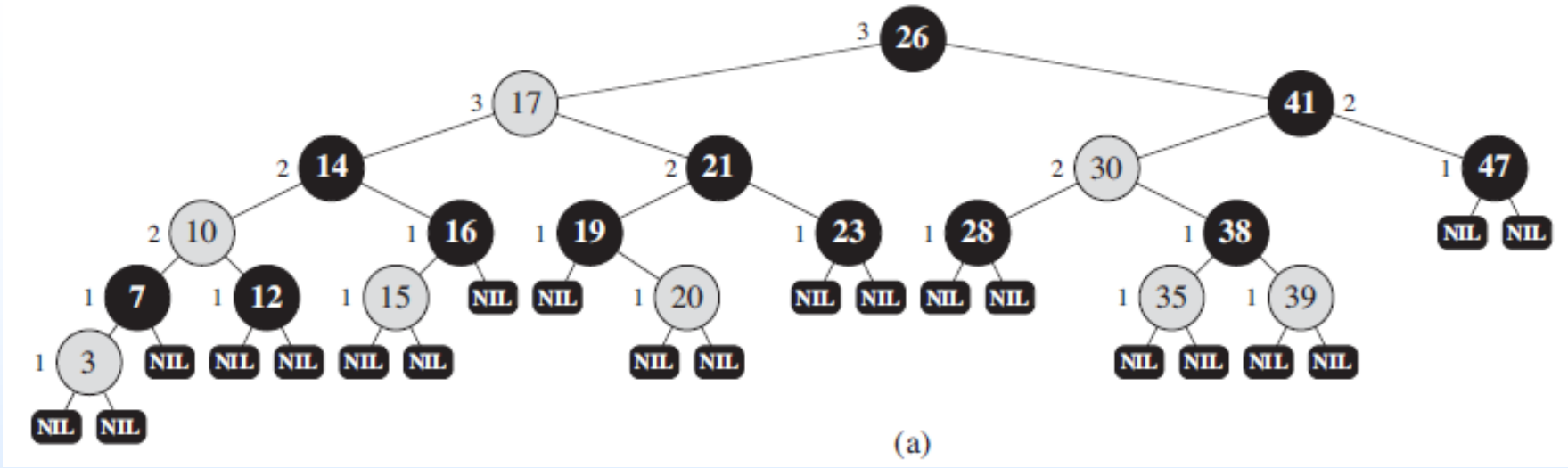
$$h(x) \leq 2bh(x)$$

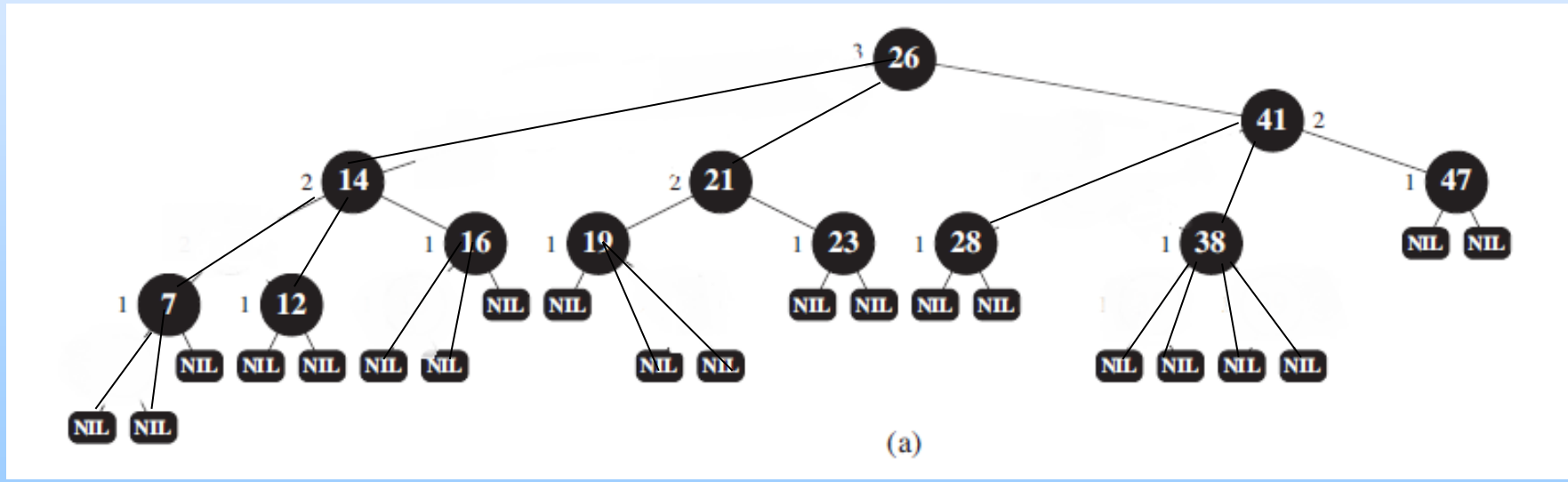
  - 결론

$$h(x) \leq 2\lg(n+1)$$

Red-Black Tree의 height (4)

- 노드 x 를 root로 하는 subtree의 internal node 개수 증명 (방법 1)
  - Black node만 남겼을 때의 노드 개수를 세는 방법으로 증명
  - Red-black tree에서 red node 삭제한 tree 생성
    - (다음 슬라이드 참조)
  - Red node가 삭제되면 property 5에 의해서 root부터 leaf까지의 모든 경로에는 동일한 개수의 black node가 있으므로 모든 leaf node들의 level은 동일하다.
  - 또한, node의 degree는 최소 2부터 최대 4가 됨
  - 따라서, internal node의 개수는 depth $bh(x)$ 인 full binary tree의 모든 internal node 개수보다 크거나 같다.
    - $bh(x)$에는 노드 x의 색깔이 포함되어 있지 않으므로 tree의 height는 $bh(x)$이다.
    - leaf node(NIL)를 제외한 internal node만 고려하므로 internal node만으로 구성된 tree의 height는 $bh(x) - 1$이다.
  - 총 internal node 개수는 최소 $1 + 2 + \cdots + 2^{bh(x)-1} = 2^{bh(x)} - 1$개보다 크거나 같다.


3 26
3 17
41 2
2 14
2 21
2 30
1 47
2 10
1 16
1 19
1 23
1 28
1 38
NIL NIL
1 7
1 12
1 15
NIL
NIL
1 20
NIL NIL
NIL NIL
1 35
1 39
1 3
NIL NIL
NIL NIL
NIL NIL
NIL NIL
NIL NIL
NIL NIL
NIL NIL
NIL NIL
NIL NIL
NIL NIL
NIL NIL
NIL NIL
NIL NIL
(a)


26
41 2
2 14
2 21
1 47
1 16
1 19
1 23
1 28
1 38
NIL NIL
1 7
1 12
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
(a)

Red-Black Tree의 height (5)

- 노드 x를 root로 하는 subtree의 internal node 개수 증명 (방법 2)
  - Height h에 대한 induction으로 증명
  - Basis step
    - $h = 0$인 경우: $bh(x) = 0$이므로 $2^{bh(x)} - 1 = 2^0 - 1 = 0$ 개의 internal node가 있음 (node x가 leaf, 즉 T.nil인 경우임)
  - Inductive step
    - $h = k$인 node x에 대해서 x를 root로 하는 subtree의 internal node 개수가 최소 $2^{bh(x)} - 1$개라고 가정하자.
    - $h = k + 1$인 node x의 black-height가 $bh(x)$이면 x의 child의 black-height는 $bh(x)$ 혹은 $bh(x) - 1$이다.
      - x의 child가 red이면 child의 black-height는 $bh(x)$
      - x의 child가 black이면 child의 black-height는 $bh(x) - 1$
    - h가 증가할 때 $bh(x)$는 같거나 증가하므로 "black-height가 $bh(x)$인 tree의 최소 internal node 개수"는 "left subtree internal node 최소 개수" + "right subtree internal node 최소 개수" + 1개이다.
    - 따라서, $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$

Red-Black Tree의 height (6)

- Red-black tree의 property 4를 이용하여 height $h \leq 2\lg(n+1)$ 임을 증명
  - Property 4: If a node is red, then both its children are black
    - Root로부터 leaf까지의 경로 중 root를 제외하고 red인 node가 있으면 반드시 black인 노드가 한 개 있으므로 black node의 개수는 최소 $h/2$ 이다.
    - 따라서, root의 black-height, $bh(x)$는 최소 $h/2$이어야 한다.

$$h(x) \leq 2bh(x)$$

  - Internal node 개수의 최소 값 증명 결과를 이용하면

$$2^{bh(x)} - 1 \leq n \quad \Rightarrow \quad bh(x) \leq \lg(n+1)$$

  - 식을 정리하면
    - $h \leq 2\ \lg(n+1)$

Red-Black Tree의 height (7)

- Lemma 13.1의 의미
  - Red-black tree의 internal node 개수가 n개이면 red-black tree의 height는 $2 \lg(n+1)$보다 작거나 같다.
  - 즉, $O(\lg n)$으로 search할 수 있다.

실습 주제(1)

- 문제 풀이
  - 알고리즘 교재 13.1-1
    - Figure 13.1(a)는 슬라이드 8페이지 그림임.
  - 알고리즘 교재 13.1-2

13.1-1

In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys $\{1, 2, \ldots, 15\}$. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

13.1-2

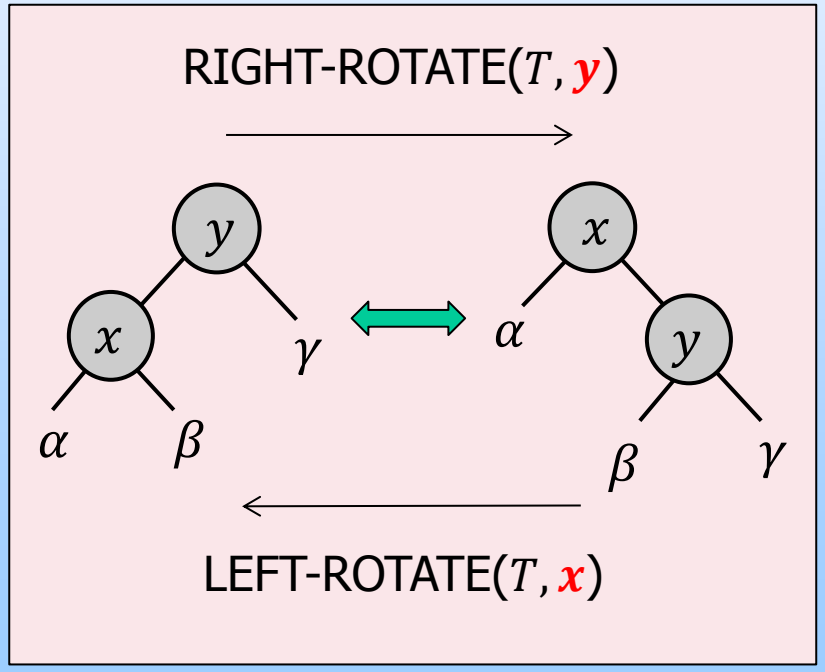
Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

목 차

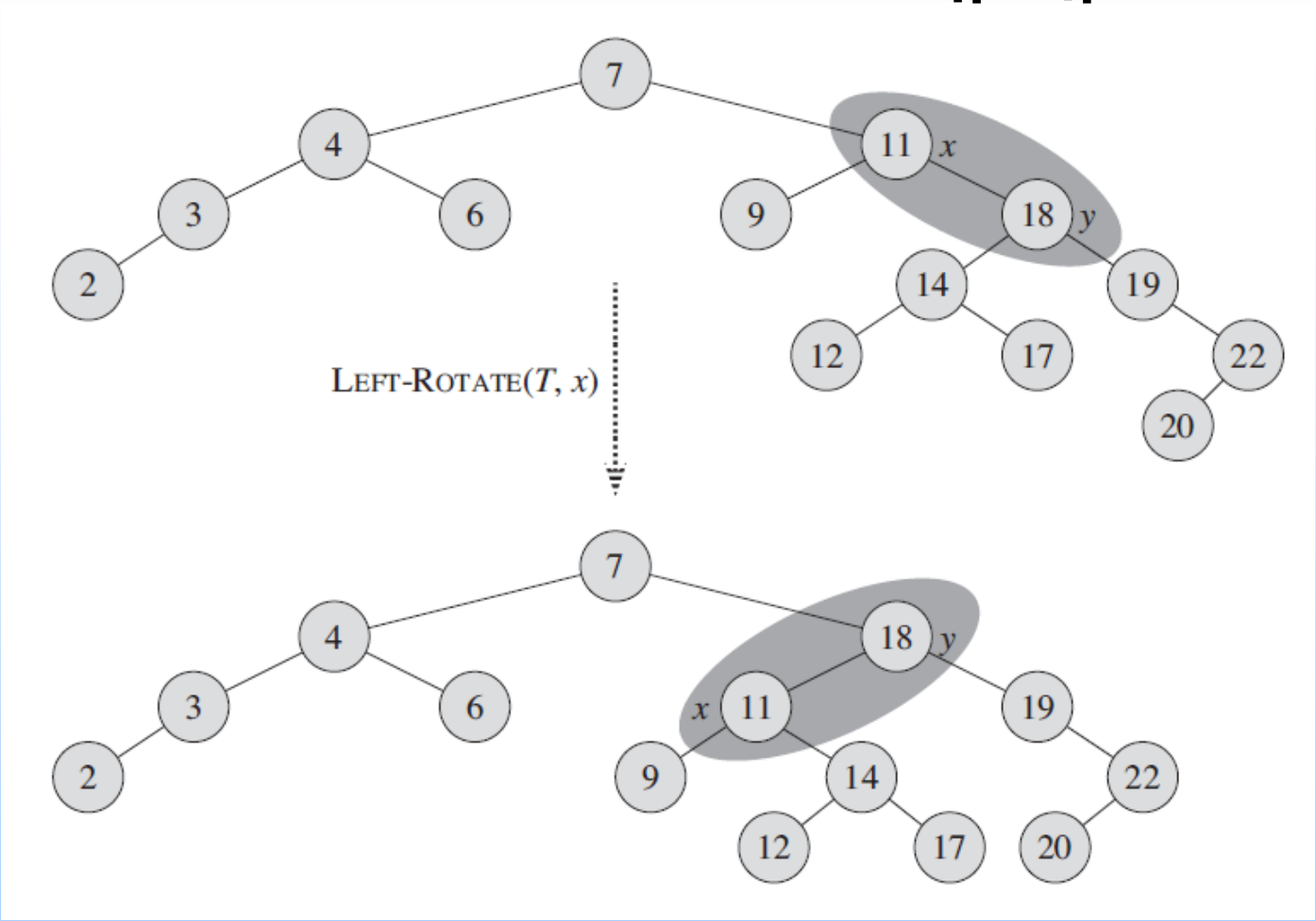
- 소개
- Rotation
- Insertion
- Deletion

Rotation 연산

- 주어진 노드를 축으로 왼쪽 혹은 오른쪽으로 회전
- Binary search tree의 특성을 유지하면서 회전
- 변경 전후에 inorder traversal 순서가 동일
  - $\alpha x \beta y \gamma \leftrightarrow \alpha x \beta y \gamma$



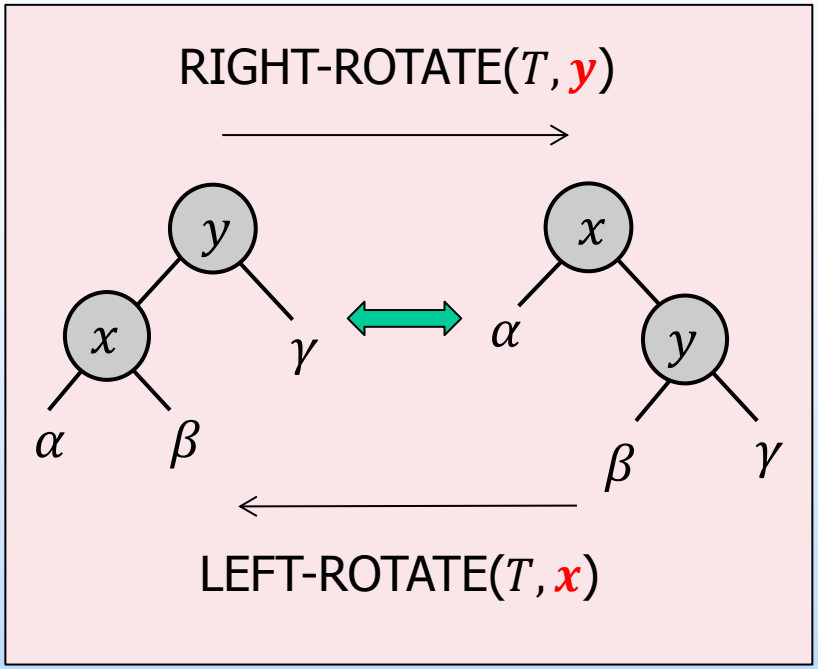
Left Rotation 예제



- 11이 18의 left child로 변경
- 18의 left child인 14가 11의 right child로 변경
- 7의 right subtree가 전체적으로 균형을 잡은 듯한 느낌…

Rotation 후 Level 변화

- Right rotation
  - α의 level은 1 감소
  - γ의 level은 1 증가
- Left rotation
  - α의 level은 1 증가
  - γ의 level은 1 감소

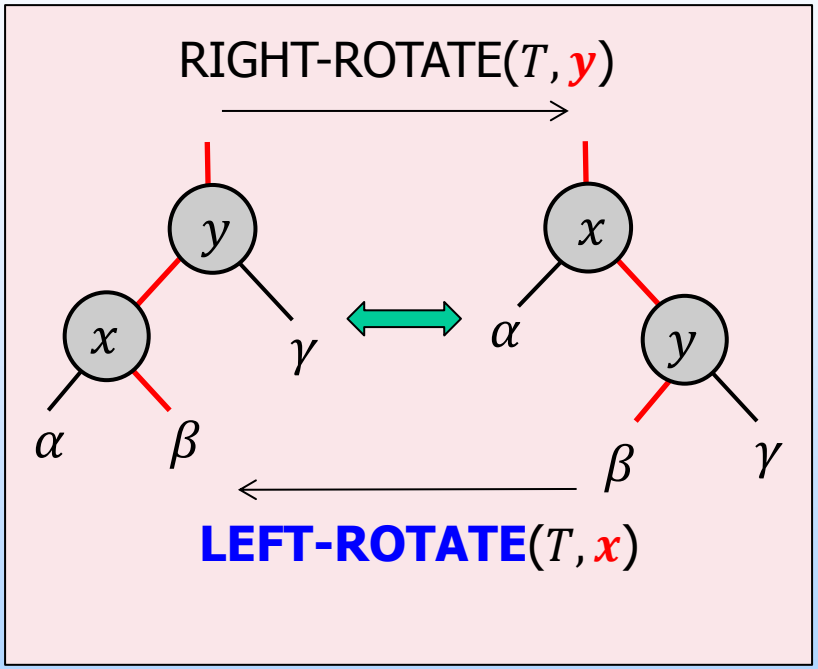


Rotation 연산의 용도

- Left subtree와 right subtree의 heights 차이 조정
- Insert/Delete 수행 시 balance 유지에 유용

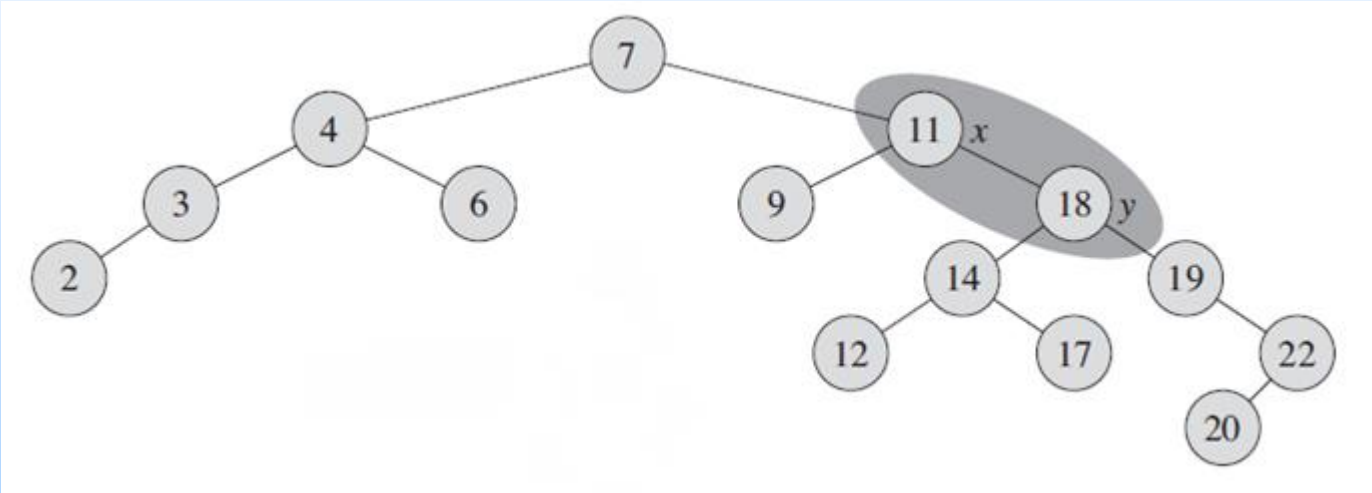
Rotation에 의해 변경되는 정보

- Left rotation 시 변경되는 연결 정보
  - β의 parent
    - Parent가 y에서 x로 변경
  - x의 parent
    - y의 parent가 됨
    - Child가 x에서 y로 변경
    - NIL이면 y가 root
  - x와 y의 관계
    - x가 y 의 left child가 됨
    - y가 x의 parent가 됨



실습 주제(2)

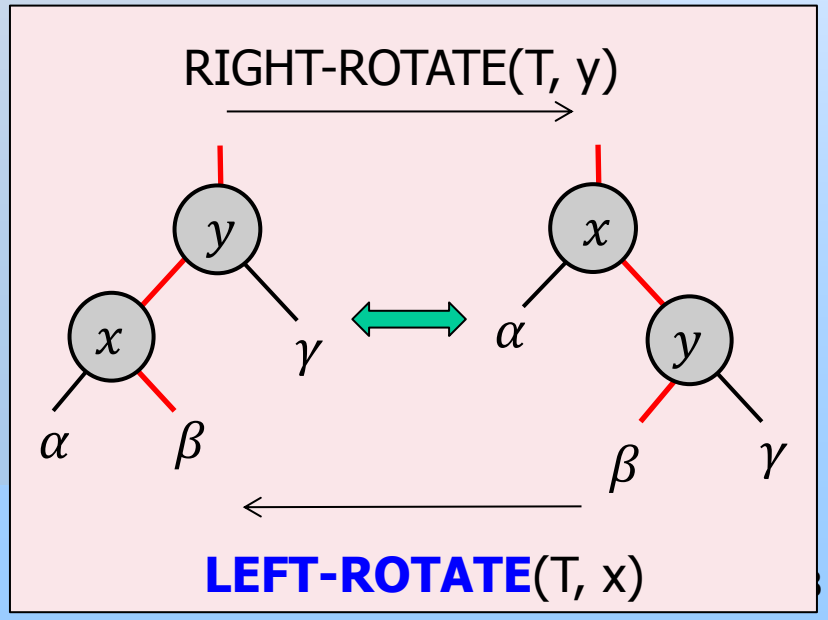
- 문제 풀이
  - 4를 중심으로 RIGHT-ROTATE
  - 11을 중심으로 LEFT-ROTATE
  - 19를 중심으로 LEFT-ROTATE



Rotation Pseudo Code

LEFT-ROTATE (T, x)

```
1    y = x.right              // set y
2    x.right = y.left         // subtree β 연결 조정
3    if y.left ≠ T.nil        // subtree β 연결 조정
4        y.left.p = x         // subtree β 연결 조정
5    y.p = x.p                // y의 parent 연결 조정
6    if x.p == T.nil          // x의 parent 연결 조정
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   // x와 y의 연결 조정
12   x.p = y      // x와 y의 연결 조정
```



실습 주제(3)

- Right-rotate code 구현
  - int rightRotate(tree_t *tree, node_t *node);

HW#7.C

- HW#7.C1
  - 실습주제 3 완료
    - Right-rotate code 구현

목 차

- 소개
- Rotation
- Insertion
- Deletion

전 략

Binary search tree의 특성을 유지하면서 insert

Insert한 노드의 색깔을 RED로 지정

Red-black tree 특성 위배 시 rotation 및 색깔 조정

RED 노드 삽입에 의한 특성 위배

- 분류
  - Parent의 색깔이 BLACK일 때 발생하는 경우
  - Parent의 색깔이 RED일 때 발생하는 경우

Parent의 색깔로 구분해서
다루는 것이 insertion 동작 이해의 첫 번째 실마리

Parent의 색깔이 BLACK인 경우 (1)

- Red 노드 삽입 시 RB-tree의 properties 위배 여부
  - Every node is either red or black. ← 만족
  - The root is black. ← 추가된 RED 노드가 root인 경우에 위배
  - Every leaf (NIL) is black. ← 만족
  - If a node is red, then both its children are black. ← 만족
  - 모든 경로에 같은 수의 black nodes. ← 만족

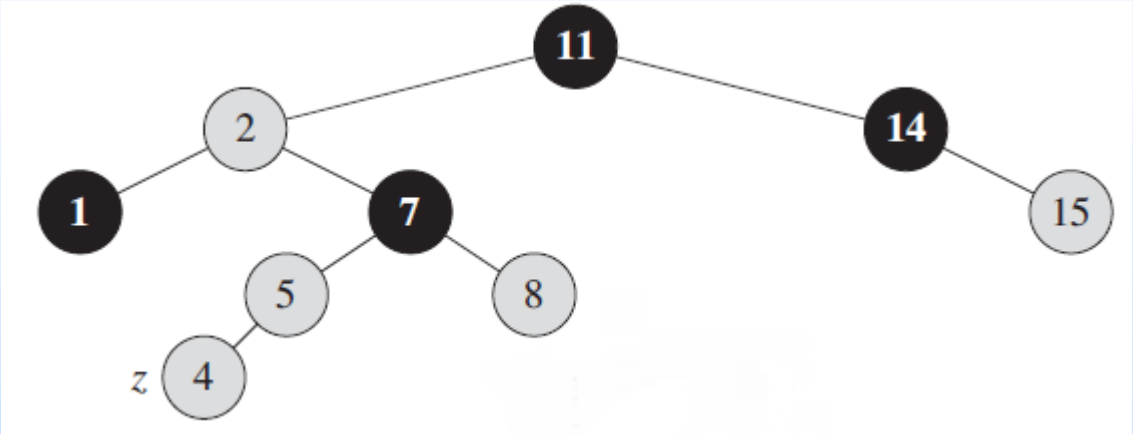
Parent가 BLACK이면
추가한 노드가 root일 때만 특성 위배.
(Root의 parent는 NIL로 지정되어 있고 NIL은 Black이다.)

Parent의 색깔이 BLACK인 경우 (2)

- 발생한 violation 해결 방안
  - 삽입한 노드의 색깔을 black으로 지정
  - 즉, root를 black으로 변경

Parent의 색깔이 RED인 경우 (1)

- Red 노드 삽입 시 RB-tree의 properties 위배 여부

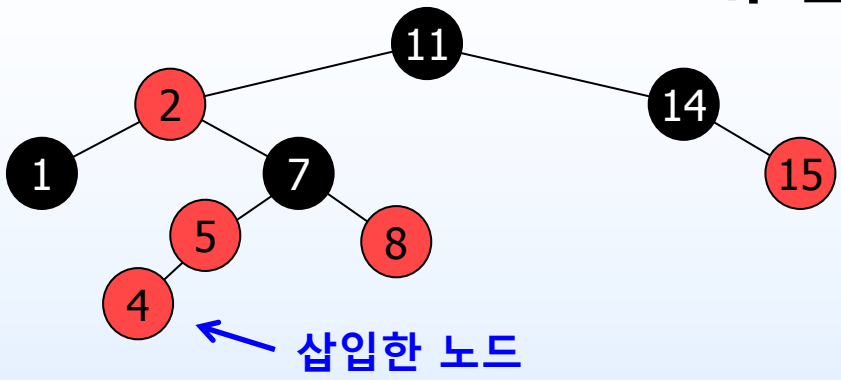


- Every node is either red or black. ← 만족
- The root is black. ← 만족
- Every leaf (NIL) is black. ← 만족
- If a node is red, then both its children are black. ← 위배
- 모든 경로에 같은 수의 black nodes. ← 만족

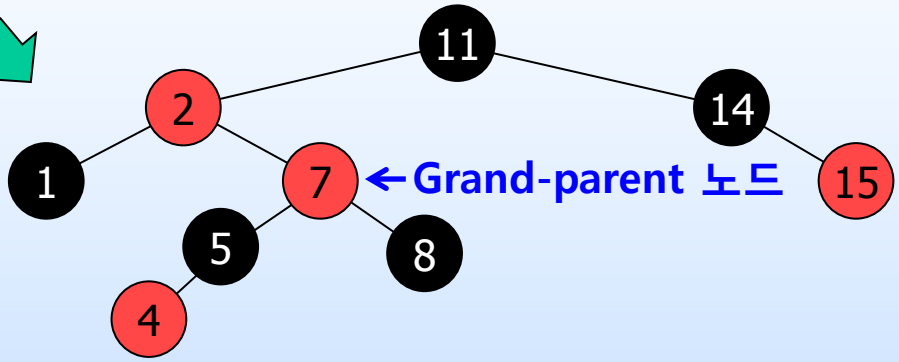
Parent의 색깔이 RED인 경우 (2)

- Properties violation 해결 방안
  1) Parent를 BLACK으로 변경
  2) Black height가 증가하지 않도록
     - Grand-parent를 RED로 변경 혹은 rotation
  3) RED로 바뀐 grand-parent가 root가 아닌 경우
     - Grand-parent의 parent가 RED인지 확인
     - RED이면 grand-parent를 기준으로 1번/2번 작업 반복하면서 root까지 거슬러 올라감
  4) RED로 바뀐 grand-parent가 root인 경우
     - Root를 BLACK으로 변경
     - 이 경우에만 tree의 black height가 1 증가함

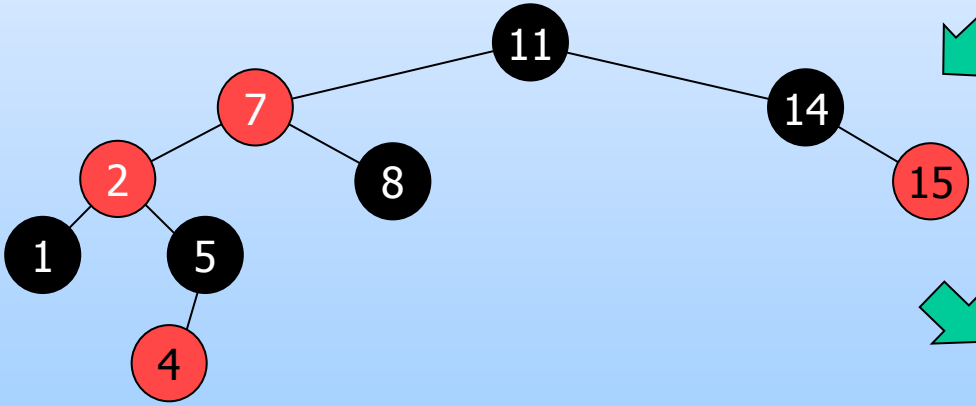
해결 예제



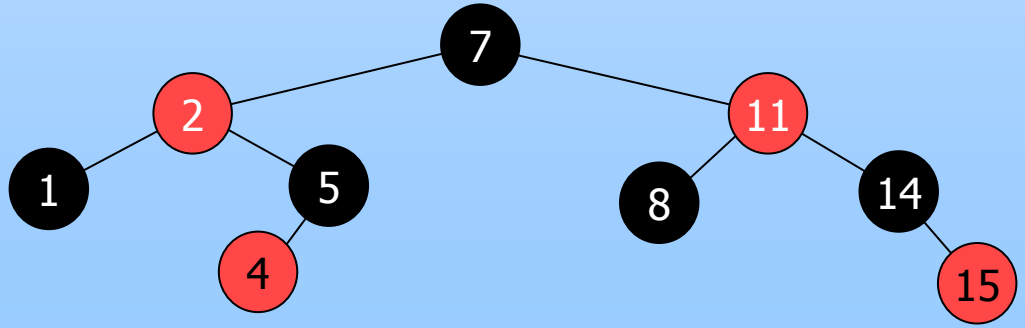
노드 5와 노드 8을 black으로 변경
노드 7을 red로 변경 → 일단 4에 의한 위배 해결



노드 2를 중심으로 left-rotate → 사전 작업



노드 7을 black으로 변경
노드 11을 red로 변경
노드 11을 중심으로 right-rotate → 완료



Parent의 색깔이 RED인 경우 (3)

- Violation 발생 경우 분류
  - RED 노드 삽입에 의한 발생
    - RED-Node 삽입 Violation (강의에서 사용하는 용어)
  - Grand-parent가 RED로 변경되어 발생하는 경우
    - RED-GP Violation (강의에서 사용하는 용어)

- Tip
  - 위의 두 가지 경우는 실재적으로는 처리하는 방법이 같다.
  - 같을 수 있는 이유는 red-black tree에서 새로 삽입되는 노드는 black NIL nodes를 child로 갖기 때문이다.
  - 발생 가능한 모든 경우를 분석하기 위해서 분리해서 설명함 → 나중에 합침.

Parent의 색깔이 RED인 경우 (4)

- RED 노드 삽입에 의한 violations 분류 기준
  - 삽입한 노드 위치: left child 혹은 right child
  - 삽입한 노드의 parent 위치: left child 혹은 right child
  - Uncle(parent의 sibling)의 색깔: red 혹은 black

Case	Parent 색깔	Parent 위치	Uncle 색깔	삽입 노드 위치
1	RED	Left Child	RED	Left Child
2				Right Child
3			BLACK	Left Child
4				Right Child
5		Right Child	RED	Left Child
6				Right Child
7			BLACK	Left Child
8				Right Child

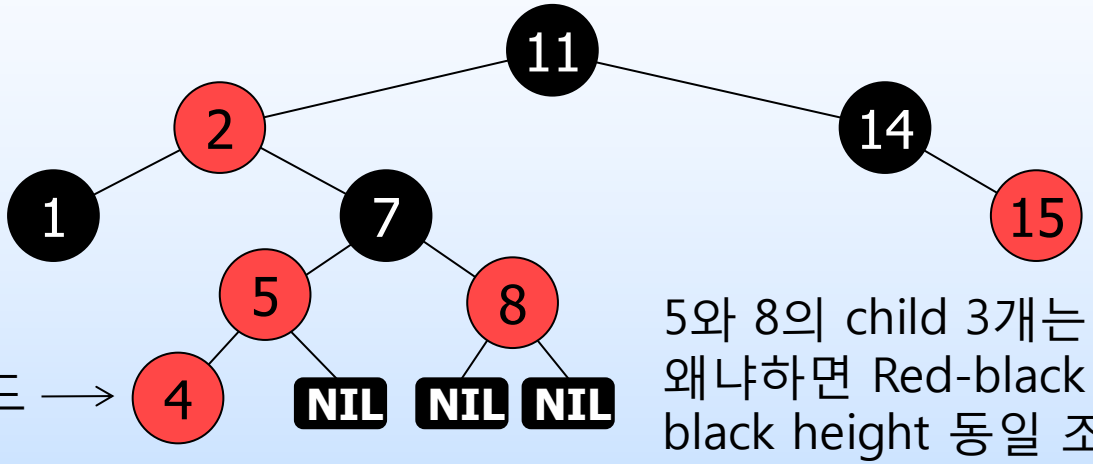
Parent의 색깔이 RED인 경우 (5)

- Grand-parent의 RED 변경에 의한 violations 분류 기준
  - Grand-parent 노드 위치: left child 혹은 right child
  - Grand-parent 노드의 parent 위치: left child 혹은 right child
  - Grand-parent 노드의 Uncle 색깔: red 혹은 black

Case	Parent 색깔	Parent 위치	Uncle 색깔	Grand-Parent 노드 위치
9	RED	Left Child	RED	Left Child
10				Right Child
11			BLACK	Left Child
12				Right Child
13		Right Child	RED	Left Child
14				Right Child
15			BLACK	Left Child
16				Right Child

RED 노드 삽입 Violations (1)

- Case 1

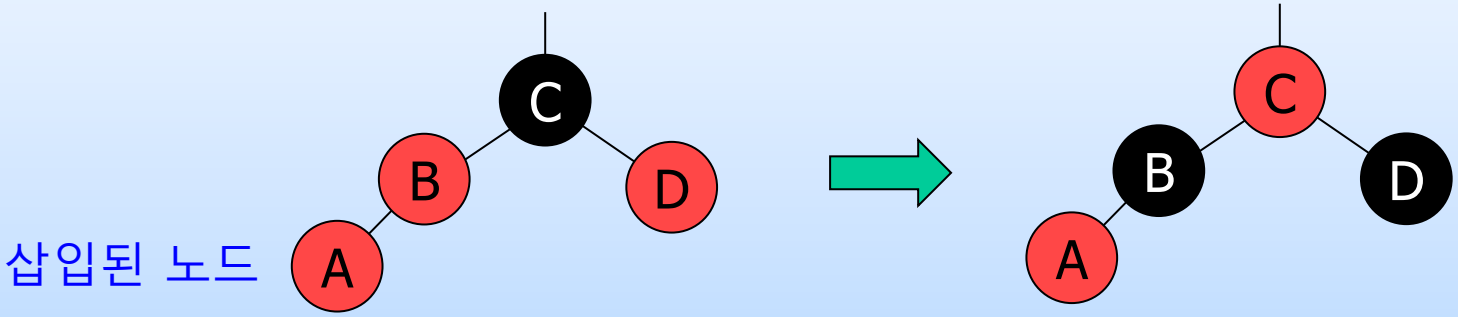


5와 8의 child 3개는 모두 NIL이어야만 함.
왜냐하면 Red-black tree의
black height 동일 조건 때문임.
다른 NIL들은 표시 생략함.

- Parent 색깔 : RED
- Parent의 위치 : left child
- Uncle 색깔 : RED
- 삽입 노드의 위치 : left child

RED 노드 삽입 Violations (2)

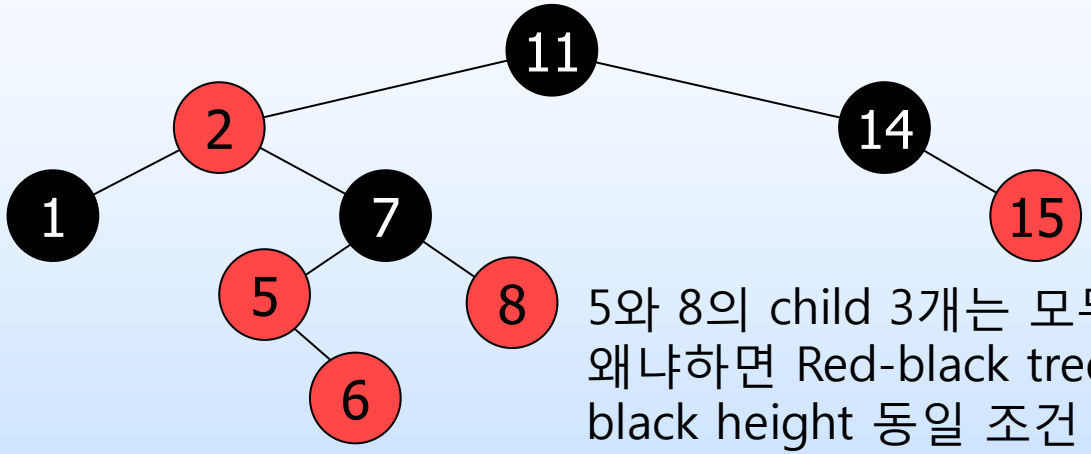
- Case 1 해결 방법
  - 삽입된 노드 A의 parent와 uncle의 색깔을 black으로 변경
  - 노드 A의 grand-parent를 red로 변경



Grand-parent C가 새로 violation을 발생시킨다면
RED-GP violation 처리 단계에서 해결

RED 노드 삽입 Violations (3)

- Case 2

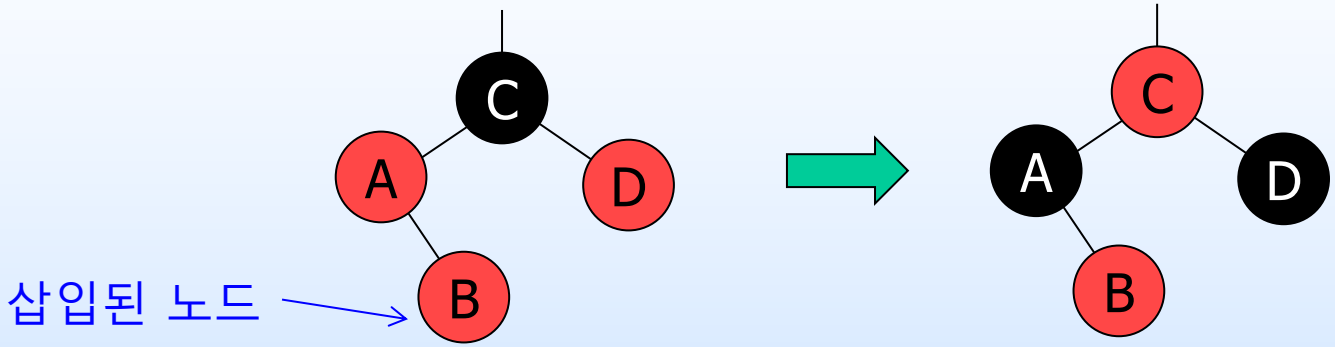


5와 8의 child 3개는 모두 NIL이어야만 함.
왜냐하면 Red-black tree의
black height 동일 조건 때문임.
NIL 표시 생략함.

- Parent 색깔 : red
- Parent의 위치 : left child
- Uncle 색깔 : red
- 삽입 노드의 위치 : right child

RED 노드 삽입 Violations (4)

- Case 2 해결 방법

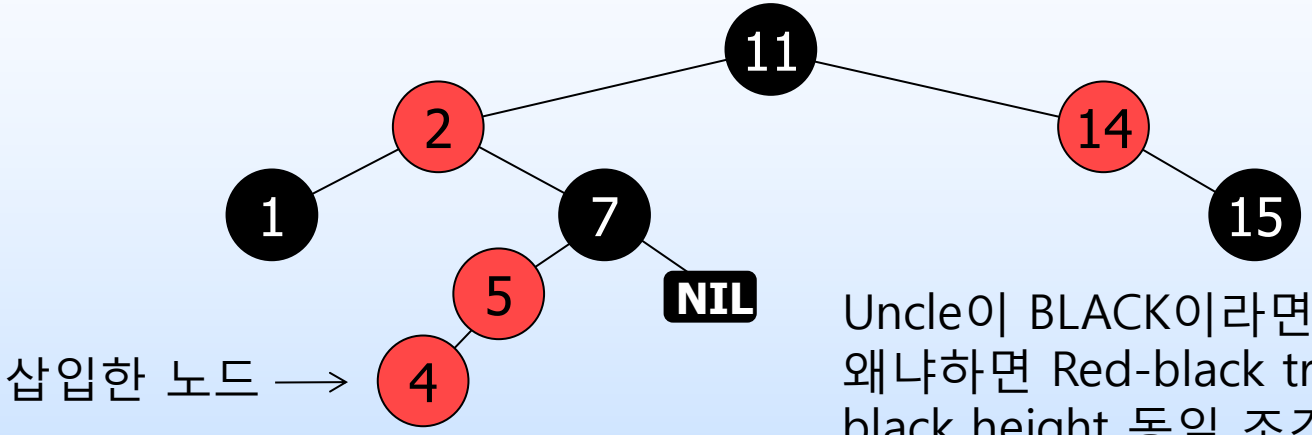


  - 삽입된 노드 B의 parent와 uncle의 색깔을 black으로 변경
  - 노드 B의 grand-parent를 red로 변경
  - Case 1과 처리 과정이 동일함
    - Parent와 uncle이 모두 RED인 경우 삽입한 노드가 left인지 right인지는 해결 방법에 영향을 주지 않음.

Grand-parent C가 새로 violation을 발생시킨다면
RED-GP violation 처리 단계에서 해결

RED 노드 삽입 Violations (5)

- Case 3

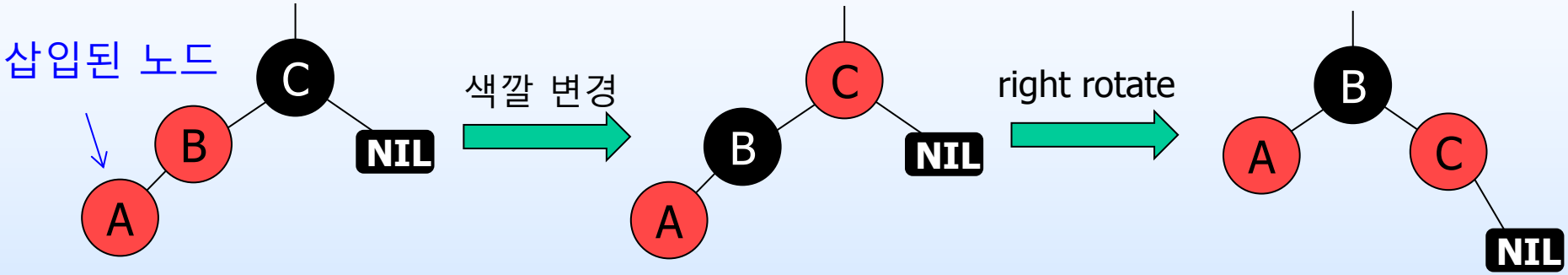


Uncle이 BLACK이라면 NIL이어야만 함.
왜냐하면 Red-black tree의
black height 동일 조건 때문임.
다른 NIL들은 표시 생략함.

- Parent 색깔 : red
- Parent의 위치 : left child
- Uncle 색깔 : black
- 삽입 노드의 위치 : left child

RED 노드 삽입 Violations (6)

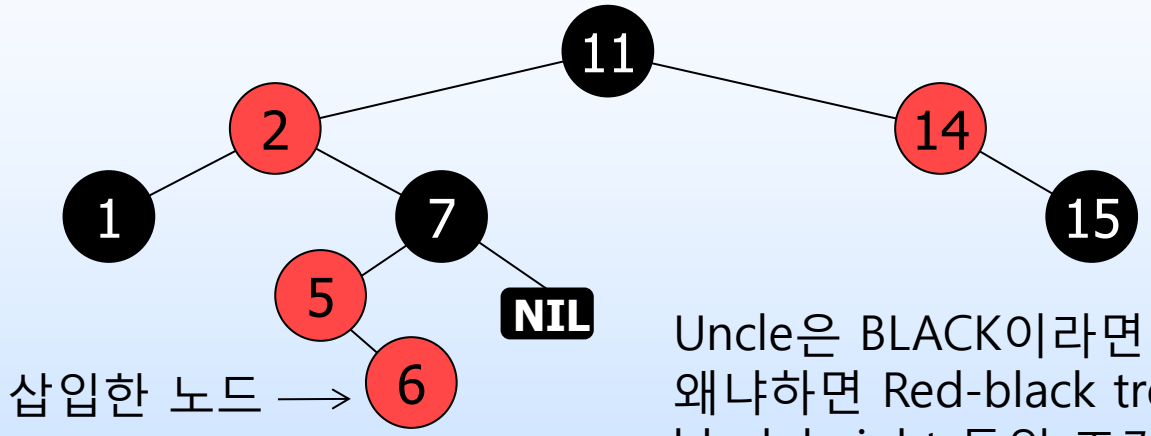
- Case 3 해결 방법



- 삽입된 노드 A의 Parent는 RED로 grand-parent는 BLACK으로 변경
- Right rotate 수행
- 모든 경로의 black height가 변하지 않았고 parent-child 모두 red인 경우가 없음
- 따라서, violation 발생하지 않으므로 insert 완료

RED 노드 삽입 Violations (7)

- Case 4

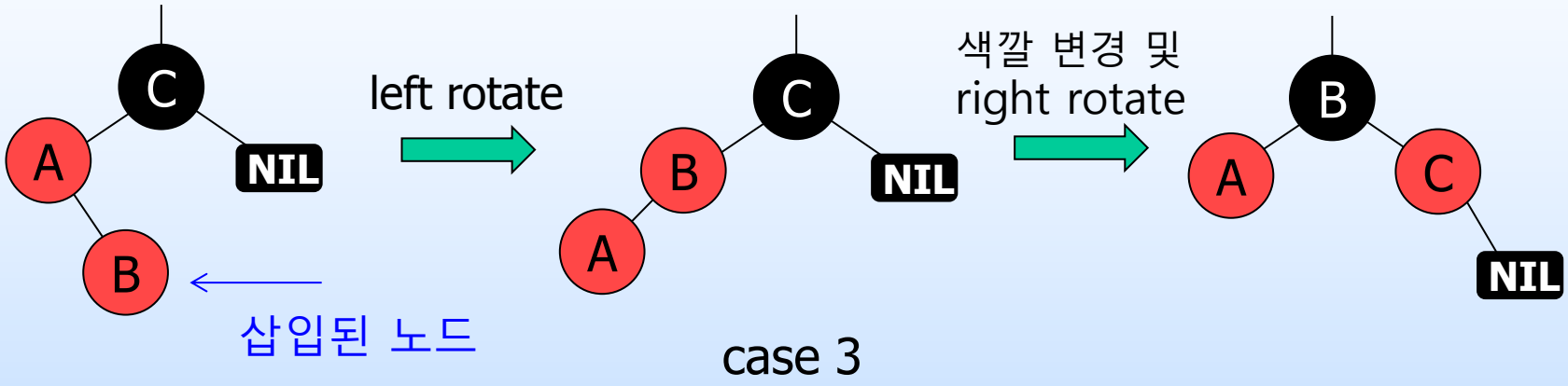


Uncle은 BLACK이라면 NIL이어야만 함.
왜냐하면 Red-black tree의
black height 동일 조건 때문임.
다른 NIL들은 표시 생략함.

- Parent 색깔: red
- Parent의 위치: left child
- Uncle 색깔 : black
- 삽입 노드의 위치 : right child

RED 노드 삽입 Violations (8)

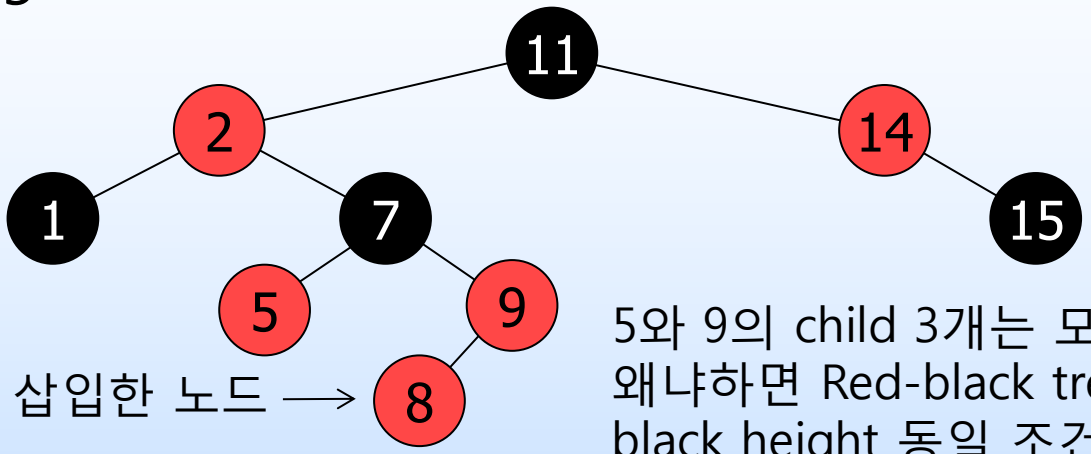
- Case 4 해결 방법



  - Left rotate 수행해서 case 3으로 변경
  - 노드 A의 parent는 BLACK으로 grand-parent는 RED로 변경
  - 다시 right rotate 수행
  - 모든 경로의 black height가 변하지 않았고 parent-child 모두 red인 경우가 없음
  - 따라서, violation 발생하지 않으므로 insert 완료

RED 노드 삽입 Violations (9)

- Case 5

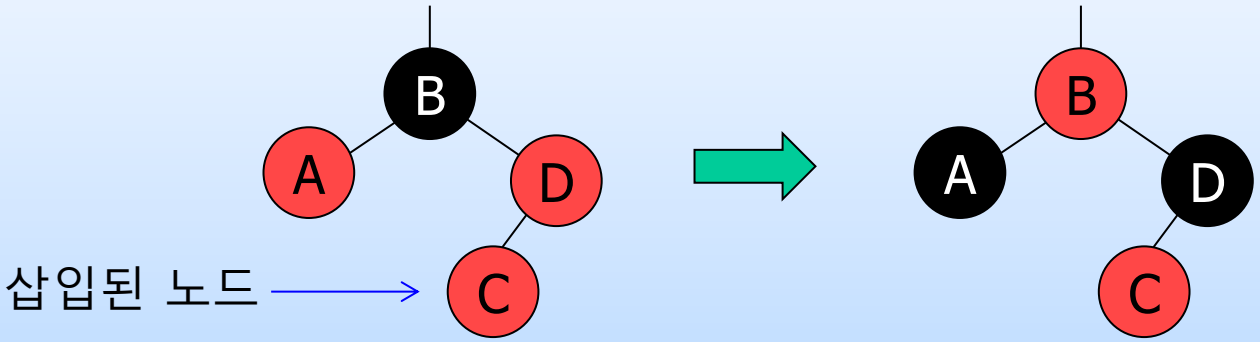


5와 9의 child 3개는 모두 NIL이어야만 함.
왜냐하면 Red-black tree의
black height 동일 조건 때문임.
NIL 표시 생략함.

- Parent 색깔 : red
- Parent의 위치 : right child
- Uncle 색깔 : red
- 삽입 노드의 위치 : left child

RED 노드 삽입 Violations (10)

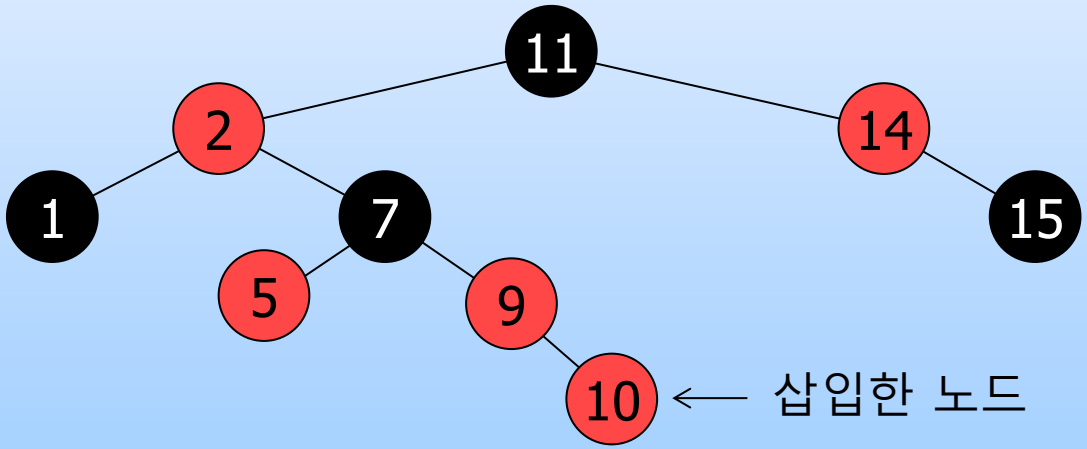
- Case 5 해결 방법
  - 삽입된 노드 C의 parent와 uncle의 색깔을 black으로 변경
  - 노드 C의 grand-parent B를 red로 변경



Grand-parent B가 새로 violation을 발생시킨다면
RED-GP violation 처리 단계에서 해결

RED 노드 삽입 Violations (11)

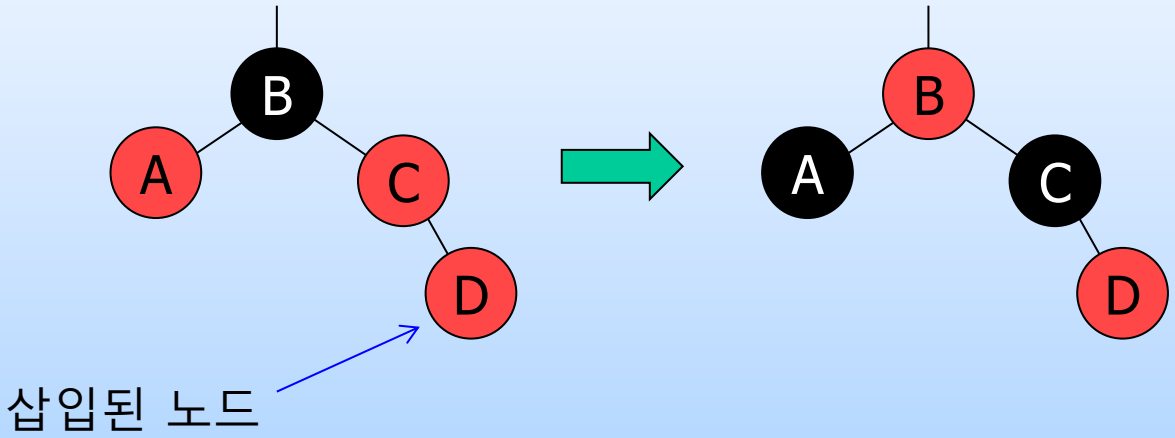
- Case 6
  - Parent 색깔 : red
  - Parent의 위치: right child
  - Uncle 색깔 : red
  - 삽입 노드의 위치 : right child



5와 9의 child 3개는 모두 NIL이어야만 함.
왜냐하면 Red-black tree의
black height 동일 조건 때문임.
NIL 표시 생략함.

RED 노드 삽입 Violations (12)

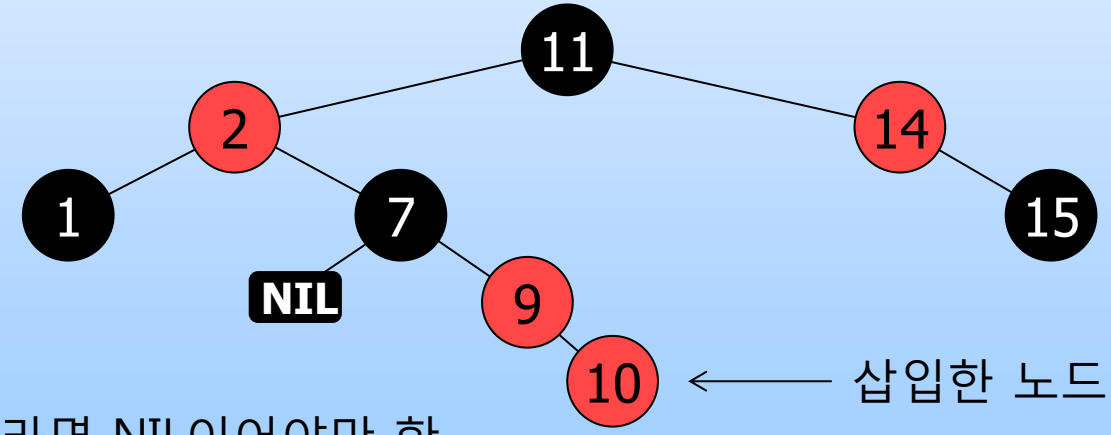
- Case 6 해결 방법
  - 삽입된 노드 D의 parent와 uncle의 색깔을 black으로 변경
  - 노드 D의 grand-parent B를 red로 변경



Grand-parent B가 새로 violation을 발생시킨다면
RED-GP violation 처리 단계에서 해결

RED 노드 삽입 Violations (13)

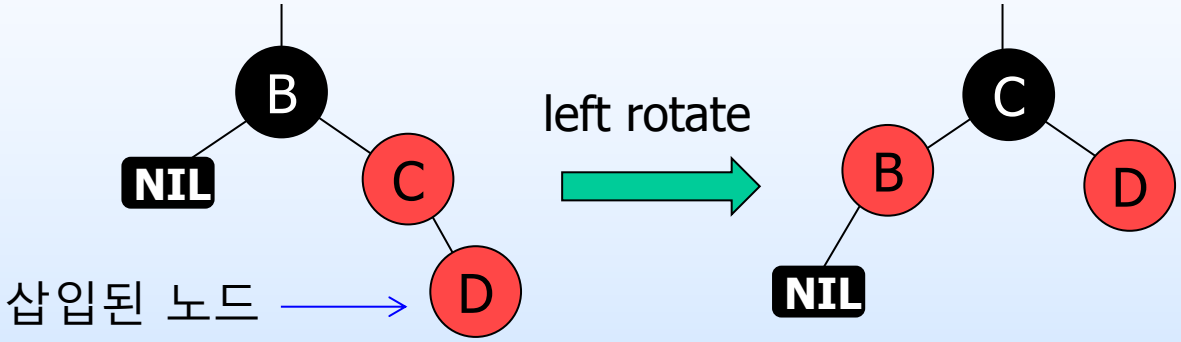
- Case 8
  - Parent 색깔 : red
  - Parent의 위치 : right child
  - Uncle 색깔 : black
  - 삽입 노드의 위치 : right child



Uncle이 BLACK이라면 NIL이어야만 함.
왜냐하면 Red-black tree의
black height 동일 조건 때문임.
다른 NIL들은 표시 생략함.

RED 노드 삽입 Violations (14)

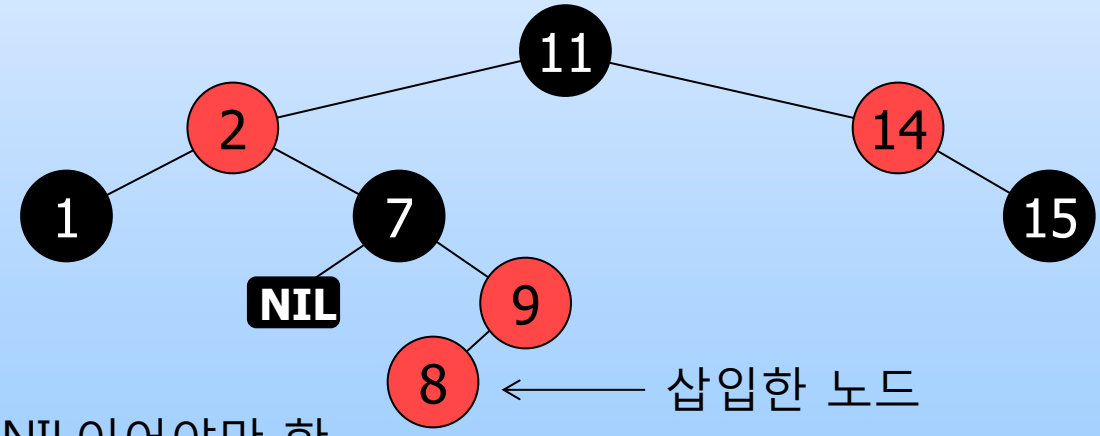
- Case 8 해결 방법



  - parent는 BLACK으로 grand-parent는 RED로 변경
  - Left rotate 수행
  - 모든 경로의 black height가 변하지 않았고 parent-child 모두 red인 경우가 없음
  - 따라서, violation 발생하지 않으므로 insert 완료

RED 노드 삽입 Violations (15)

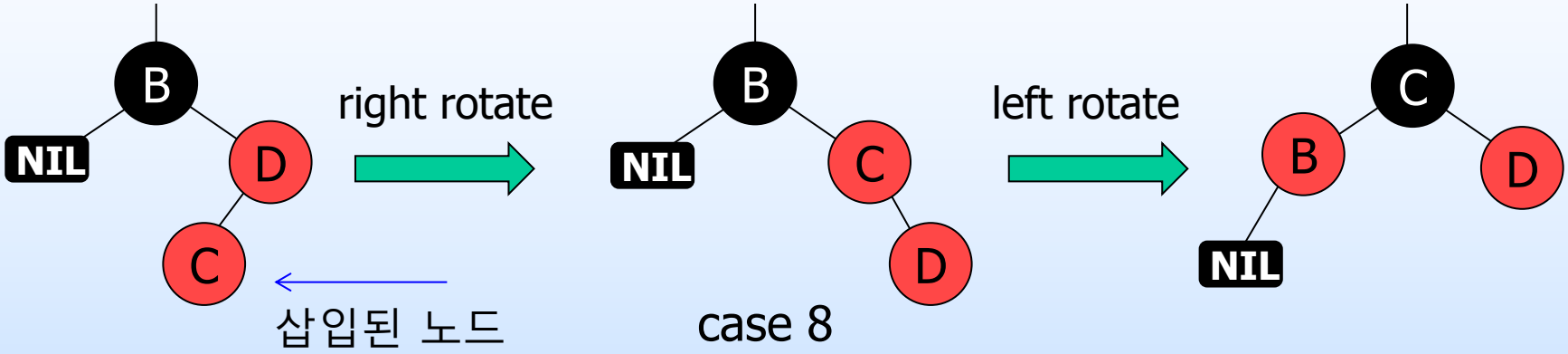
- Case 7
  - Parent 색깔 : red
  - Parent의 위치 : right child
  - Uncle 색깔 : black
  - 삽입 노드의 위치 : left child



Uncle이 BLACK이라면 NIL이어야만 함.
왜냐하면 Red-black tree의
black height 동일 조건 때문임.
다른 NIL들은 표시 생략함.

RED 노드 삽입 Violations (16)

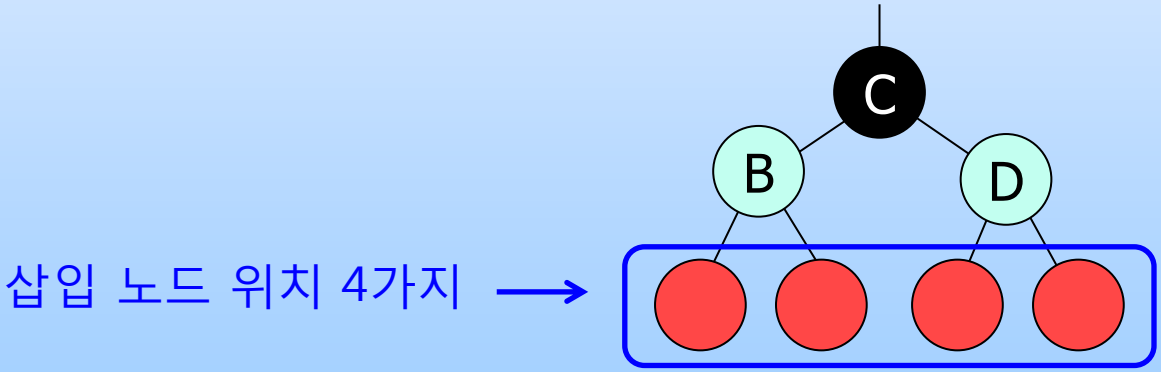
- Case 7 해결 방법



- Right rotate 수행해서 case 8로 변경
- 노드 D의 parent는 BLACK으로 grand-parent는 RED로 변경
- 다시 left rotate 수행
- 모든 경로의 black height가 변하지 않았고 parent-child 모두 red인 경우가 없음
- 따라서, violation 발생하지 않으므로 insert 완료

Cases 형식 요약

- 8가지 종류가 나오는 이유
  - 삽입되는 노드가 grand-child의 descendants로 들어가는 경우의 수 네 가지
  - Uncle 색깔 두 가지: red 혹은 black
  - Parent가 black인 경우는 고려 대상 아님
    - 왜냐하면 violation이 발생하지 않기 때문임.

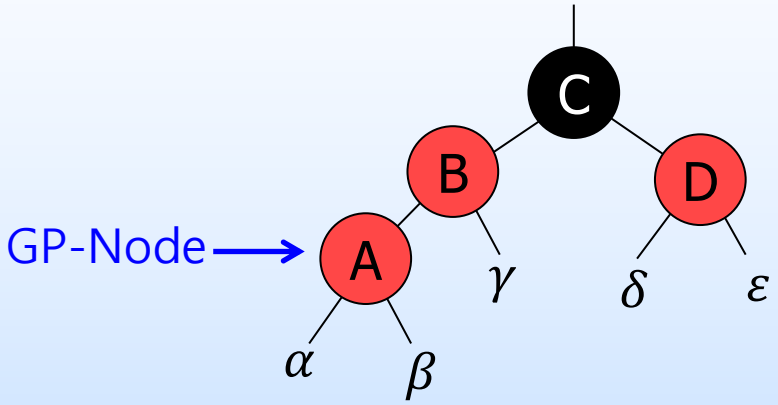


RED-GP Violations 특성

- RED 노드 삽입 violation 중 발생
  - RED 노드 삽입 후 grand-parent가 RED가 되는 경우
  - case 1, case 2, case 5, case 6에 발생 가능
- RED-GP violation이 발생한 노드는 NIL이 아닌 두 개의 children을 갖는다.
  - GP-Node로 명명
- 두 개의 children은 모두 BLACK이다.

RED-GP Violations (1)

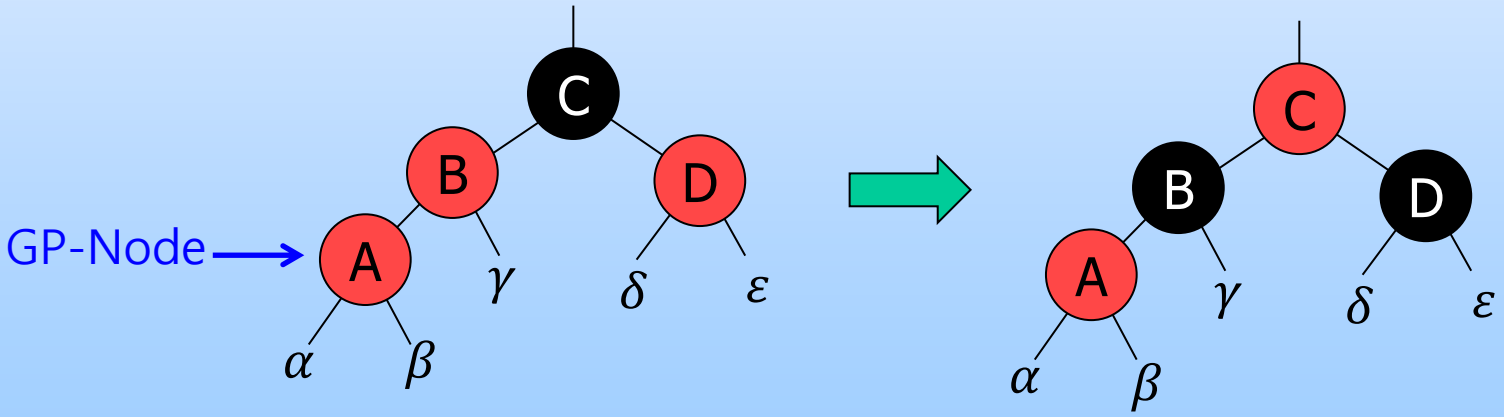
- Case 9



  - Parent 색깔 : RED
  - Parent의 위치 : left child
  - Uncle 색깔 : RED
  - GP-Node의 위치 : left child

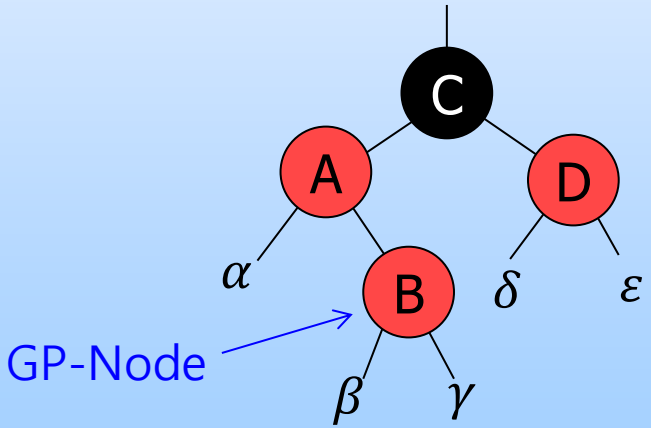
RED-GP Violations (2)

- Case 9 해결 방법
  - GP-Node A의 parent와 uncle의 색깔을 black으로 변경
  - GP-Node A의 grand-parent를 red로 변경
  - Grand-parent C가 다시 violation을 발생시킨다면
    - C가 root일 경우 black으로 변경하고 종료
    - C가 root가 아닌 경우 노드 C를 GP-Node로 지정하고 RED-GP violation 처리 다시 수행



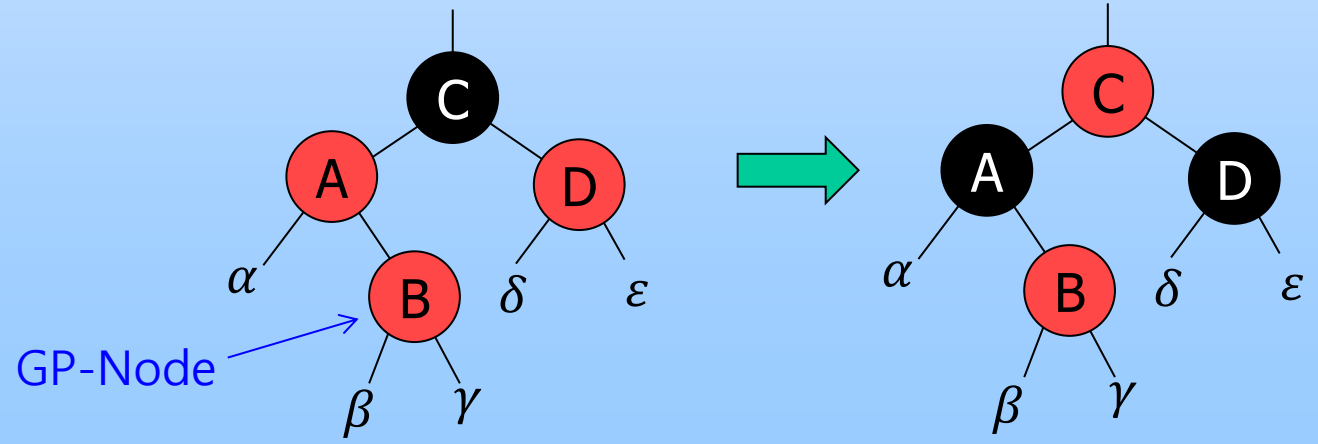
RED-GP Violations (3)

- Case 10
  - Parent 색깔 : red
  - Parent의 위치 : left child
  - Uncle 색깔 : red
  - GP-Node의 위치 : right child



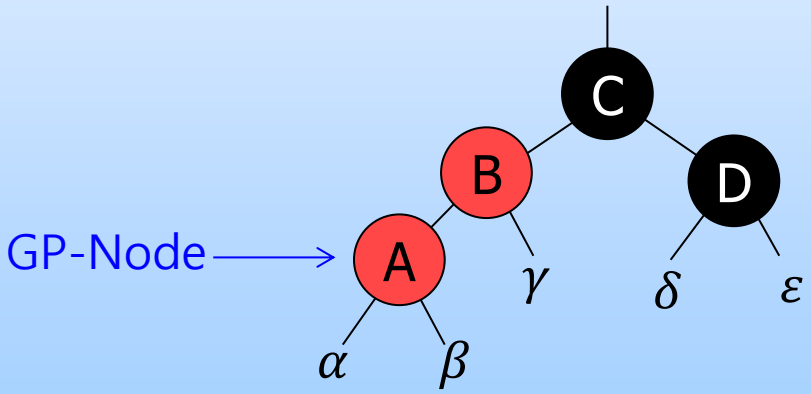
RED-GP Violations (4)

- Case 10 해결 방법
  - GP-Node B의 parent와 uncle의 색깔을 black으로 변경
  - 노드 B의 grand-parent를 red로 변경
  - Grand-parent C가 다시 violation을 발생시킨다면
    - C가 root일 경우 black으로 변경하고 종료
    - C가 root가 아닌 경우 노드 C를 GP-Node로 지정하고 RED-GP violation 처리 수행
  - Case 9와 처리 과정이 동일함
    - Parent와 uncle이 모두 RED인 경우 GP-Node가 left인지 right인지는 해결 방법에 영향을 주지 않음.



RED-GP Violations (5)

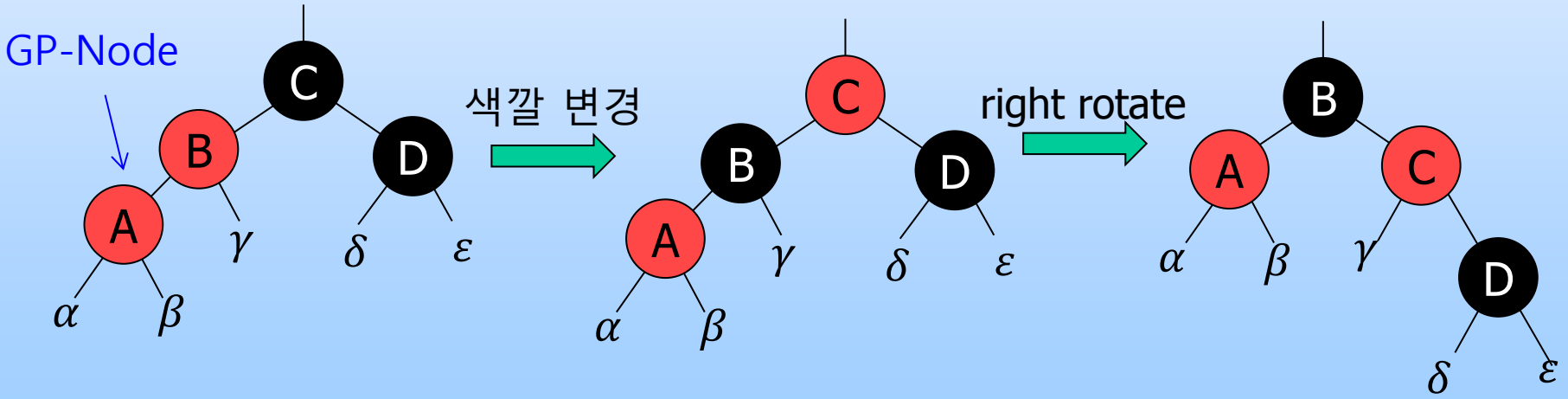
- Case 11
  - Parent 색깔 : red
  - Parent의 위치 : left child
  - Uncle 색깔 : black
  - GP-Node의 위치 : left child



case 3

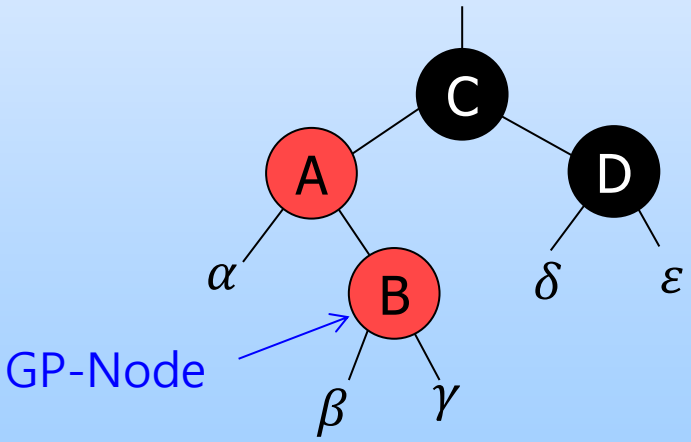
RED-GP Violations (6)

- Case 11 해결 방법
  - GP-Node A의 Parent는 BLACK으로 grand-parent는 RED로 변경
  - Right rotate 수행
  - 모든 경로의 black height가 변하지 않았고 parent-child 모두 red인 경우가 없음
  - 따라서, violation 발생하지 않으므로 insert 완료



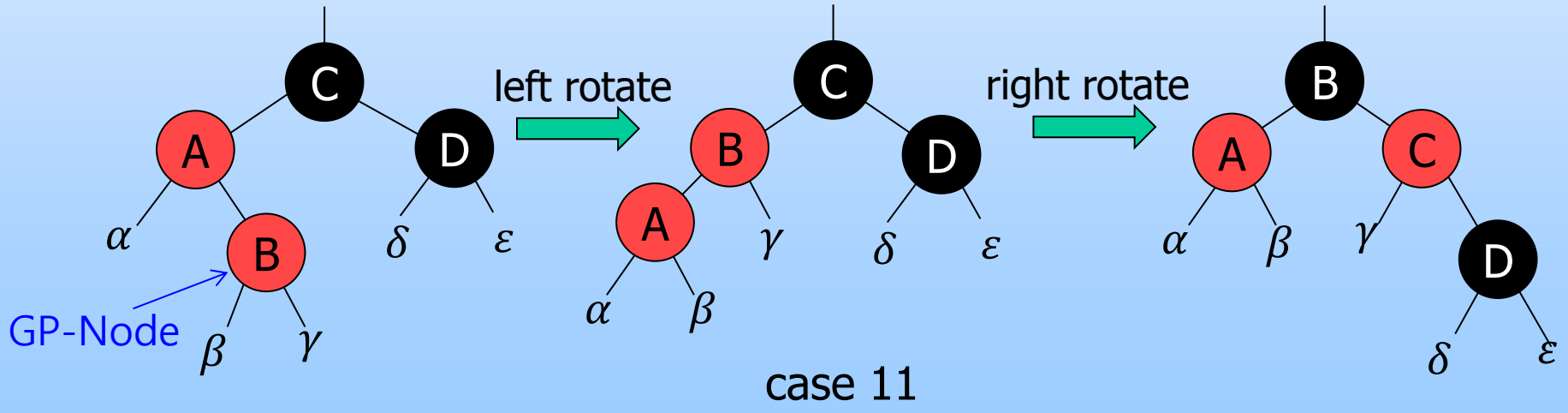
RED-GP Violations (7)

- Case 12
  - Parent 색깔: red
  - Parent의 위치: left child
  - Uncle 색깔 : black
  - GP-Node의 위치 : right child



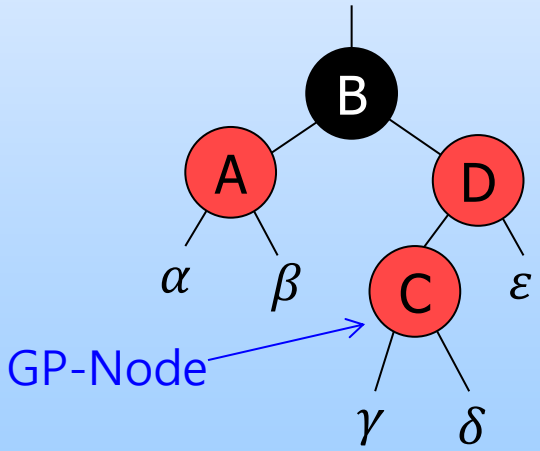
RED-GP Violations (8)

- Case 12 해결 방법
  - Left rotate 수행해서 case 11로 변경
  - 노드 A의 parent는 BLACK으로 grand-parent는 RED로 변경
  - 다시 right rotate 수행
  - 모든 경로의 black height가 변하지 않았고 parent-child 모두 red인 경우가 없음
  - 따라서, violation 발생하지 않으므로 insert 완료



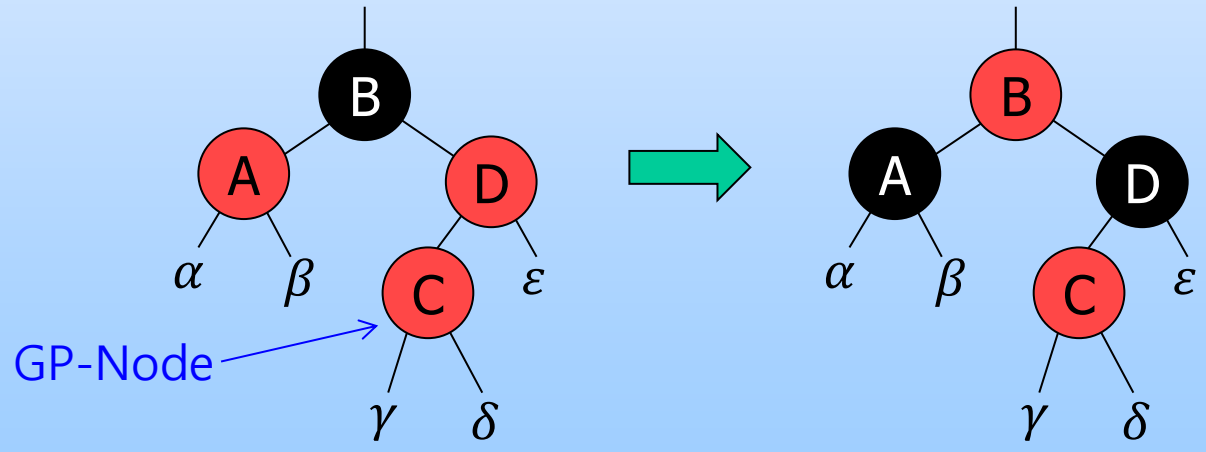
RED-GP Violations (9)

- Case 13
  - Parent 색깔 : red
  - Parent의 위치 : right child
  - Uncle 색깔 : red
  - GP-Node의 위치 : left child



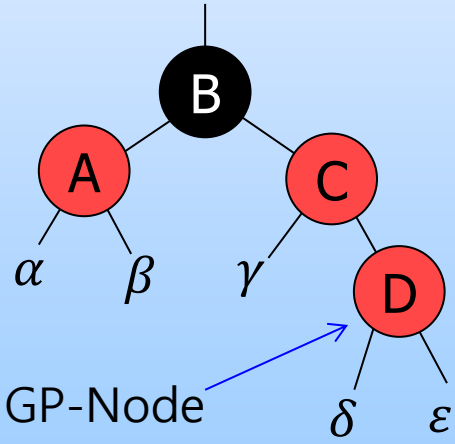
RED-GP Violations (10)

- Case 13 해결 방법
  - GP-Node C의 parent와 uncle의 색깔을 black으로 변경
  - GP-Node C의 grand-parent B를 red로 변경
  - Grand-parent B가 다시 violation을 발생시킨다면
    - B가 root일 경우 black으로 변경하고 종료
    - B가 root가 아닌 경우 노드 B를 GP-Node로 지정하고 RED-GP violation 처리 수행



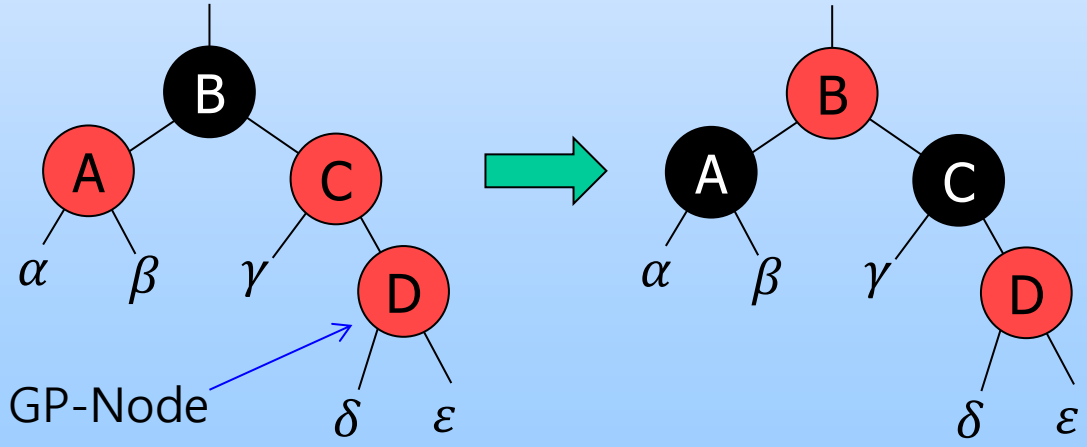
RED-GP Violations (11)

- Case 14
  - Parent 색깔 : red
  - Parent의 위치: right child
  - Uncle 색깔 : red
  - GP-Node의 위치 : right child



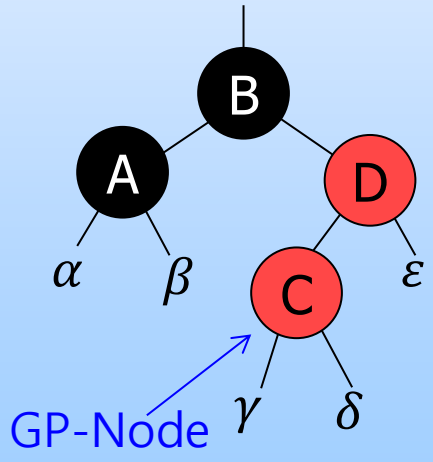
RED-GP Violations (12)

- Case 14 해결 방법
  - GP-Node D의 parent와 uncle의 색깔을 black으로 변경
  - 노드 D의 grand-parent B를 red로 변경
  - Grand-parent B가 다시 violation을 발생시킨다면
    - B가 root일 경우 black으로 변경하고 종료
    - B가 root가 아닌 경우 노드 B를 GP-Node로 지정하고 RED-GP violation 처리 수행



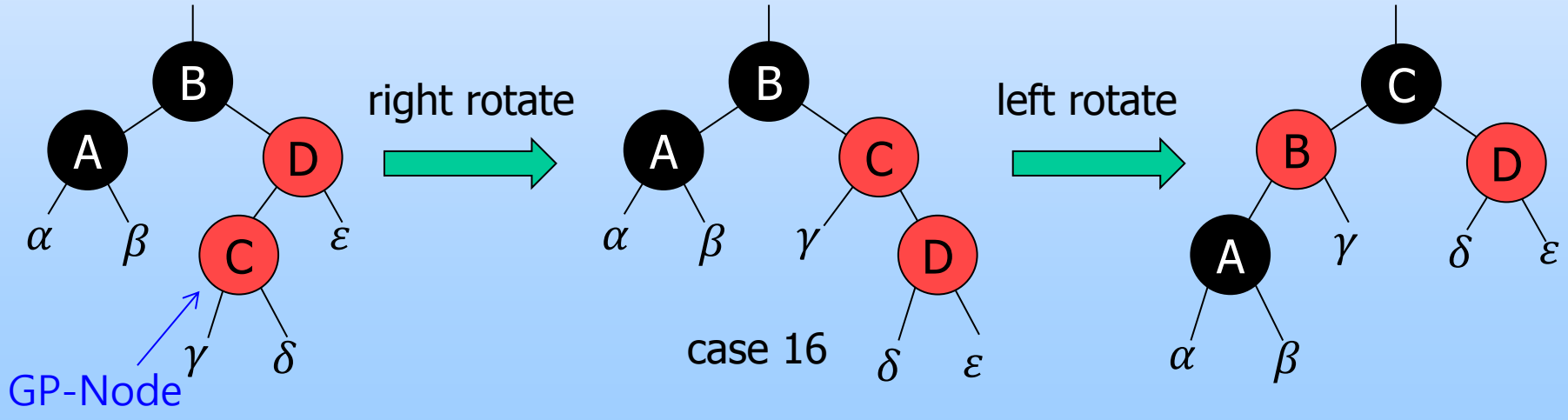
RED-GP Violations (13)

- Case 15
  - Parent 색깔 : red
  - Parent의 위치 : right child
  - Uncle 색깔 : black
  - GP-Node의 위치 : left child



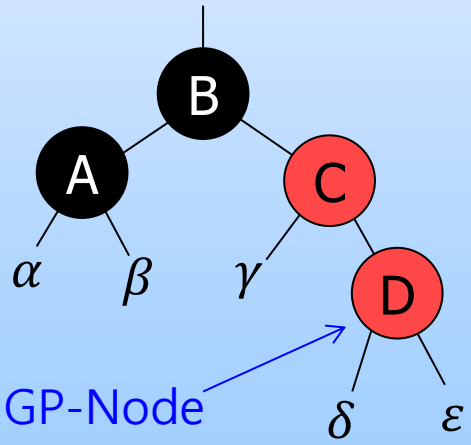
RED-GP Violations (14)

- Case 15 해결 방법
  - Right rotate 수행해서 case 16으로 변경
  - 노드 D의 parent는 BLACK으로 grand-parent는 RED로 변경
  - 다시 left rotate 수행
  - 모든 경로의 black height가 변하지 않았고 parent-child 모두 red인 경우가 없음
  - 따라서, violation 발생하지 않으므로 insert 완료



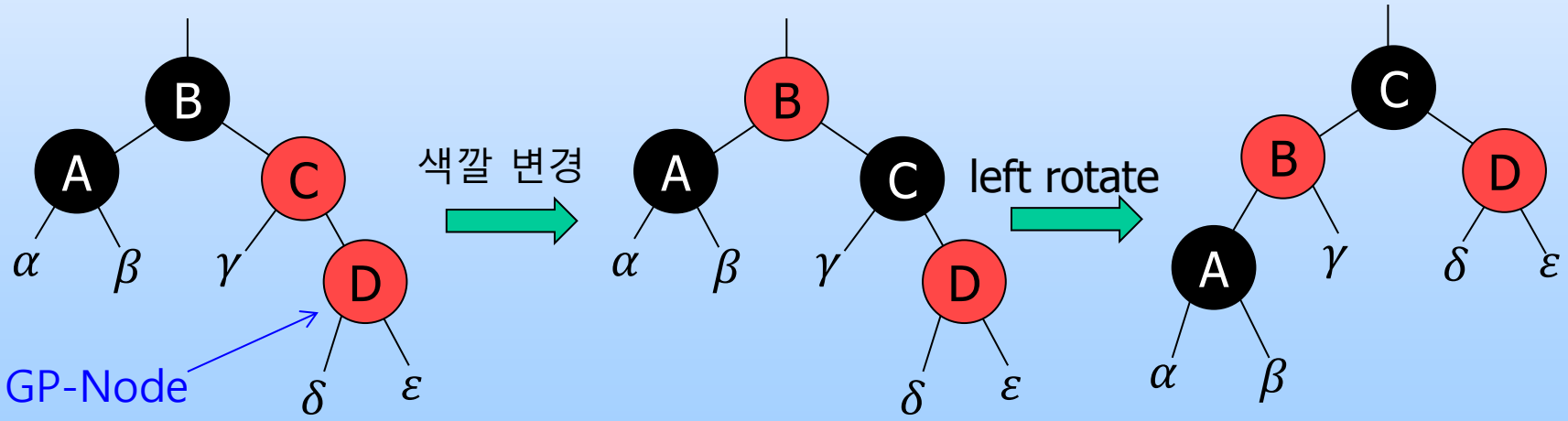
RED-GP Violations (15)

- Case 16
  - Parent 색깔 : red
  - Parent의 위치 : right child
  - Uncle 색깔 : black
  - GP-Node의 위치 : right child



RED-GP Violations (16)

- Case 16 해결 방법
  - parent는 BLACK으로 grand-parent는 RED로 변경
  - Left rotate 수행
  - 모든 경로의 black height가 변하지 않았고 parent-child 모두 red인 경우가 없음
  - 따라서, violation 발생하지 않으므로 insert 완료



중간 정리

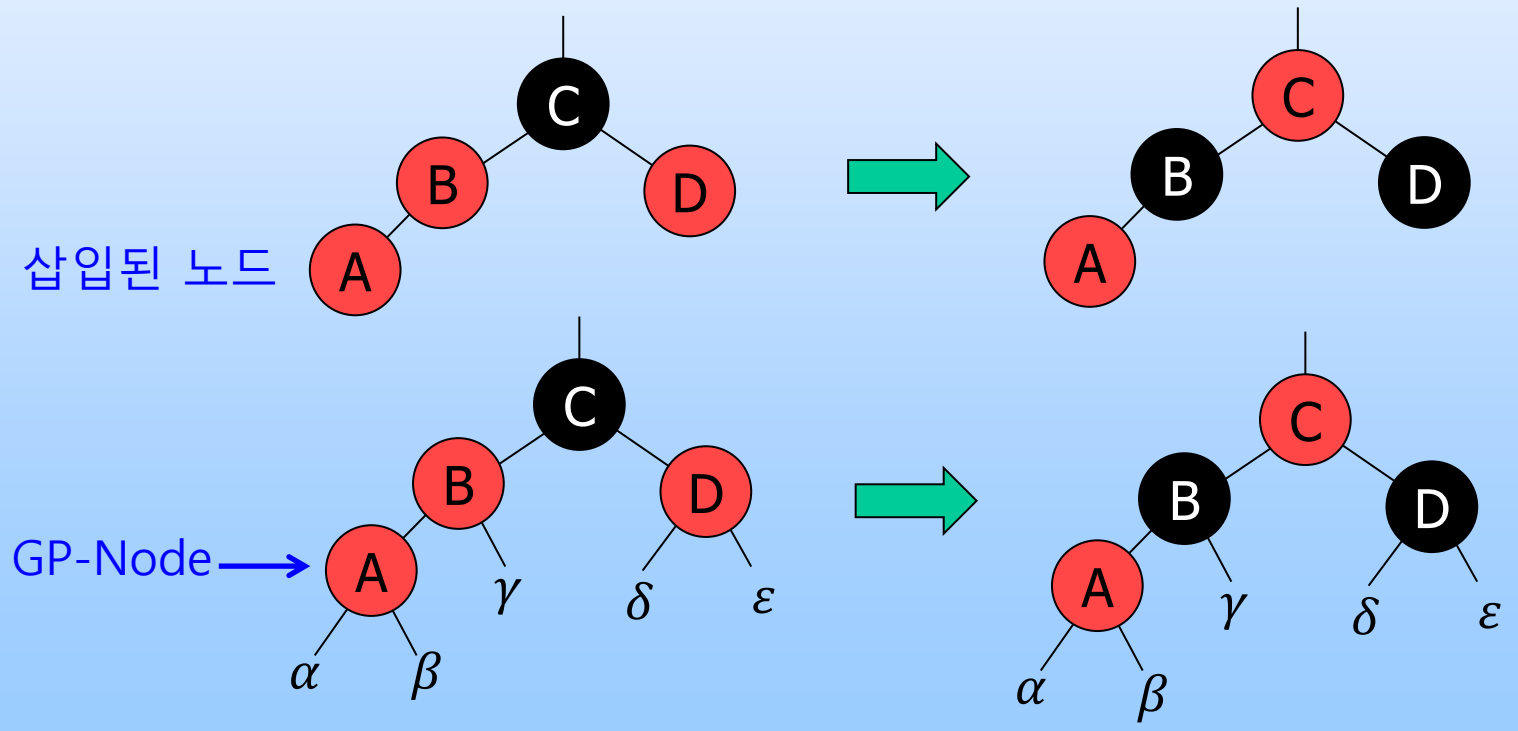
- Parent 노드 색깔이 RED일 때 violations 경우의 수
  - 총 16가지
  - RED node 삽입 violations 8가지
  - RED-GP violations 8가지

- 경우의 수 축소
  - RED node 삽입과 RED-GP 경우를 별도로 처리해야 하나?
  - Symmetric cases?
  - 동일하게 처리되는 경우는 없을까?

RED node 삽입과 RED-GP 비교 (1)

- Case 1 vs Case 9
  - 동일한 구조
  - RED-GP가 더 일반적
    - 예: NIL ∈ {트리 α}

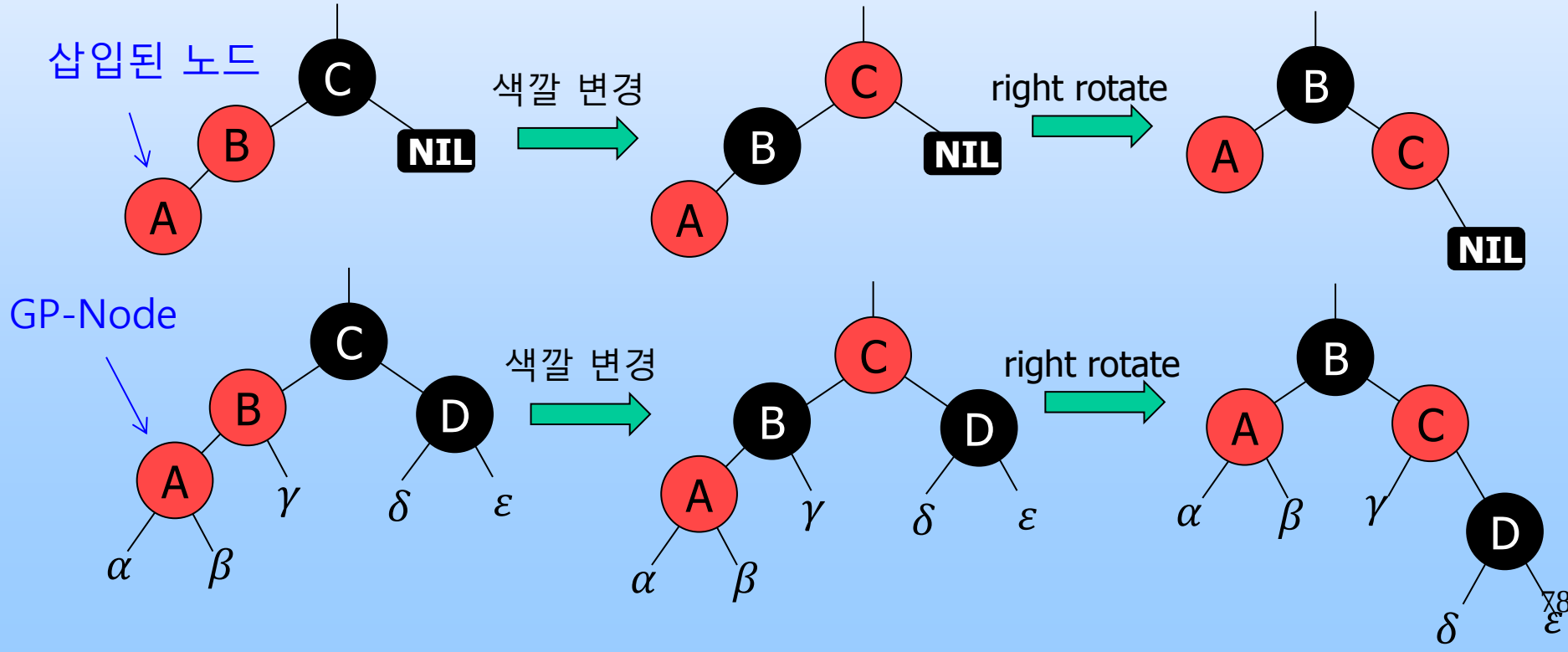
	Case 1	Case 9
Parent 색깔	RED	RED
Parent 위치	Left Child	Left Child
Uncle 색깔	RED	RED
처리할 노드 위치	Left Child	Left Child



RED node 삽입과 RED-GP 비교 (2)

- Case 3 vs Case 11
  - 동일한 구조
  - RED-GP가 더 일반적
    - 예: NIL ∈ {트리 D}

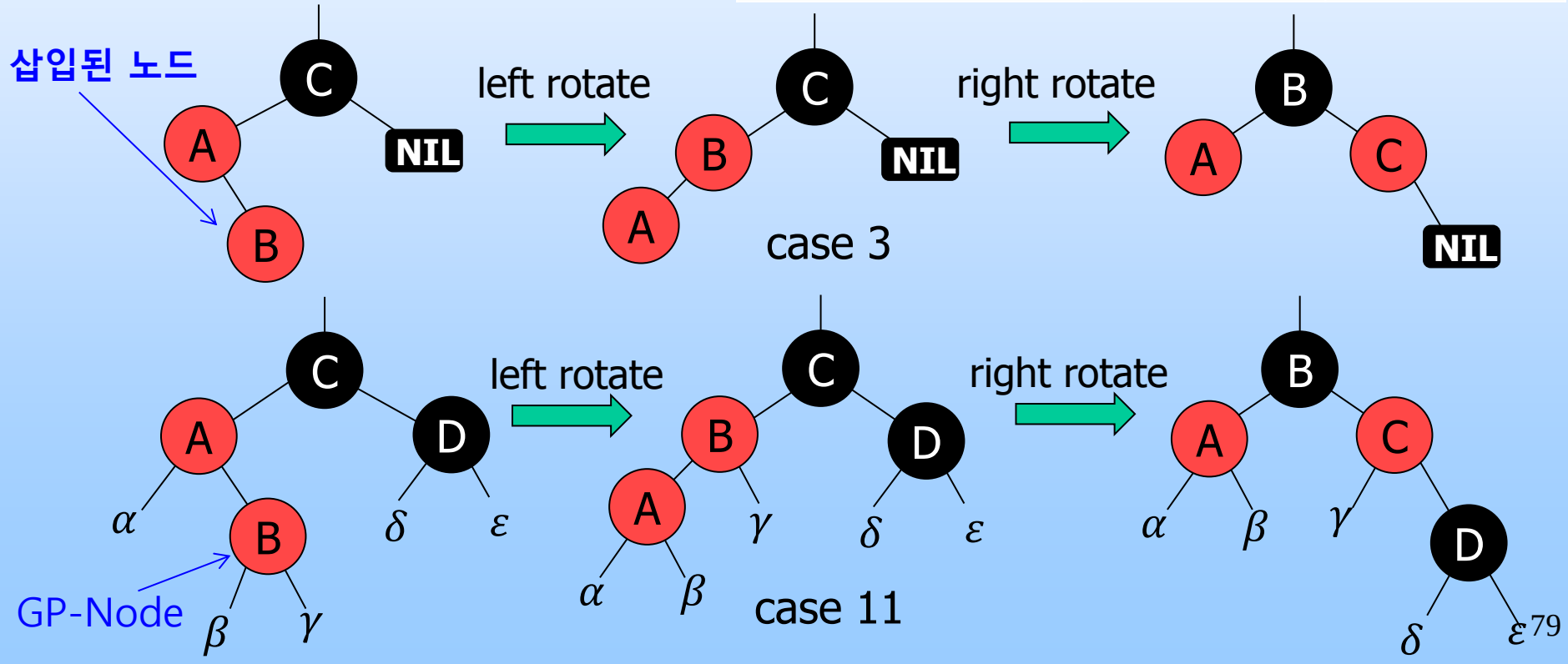
	Case 3	Case 11
Parent 색깔	RED	RED
Parent 위치	Left Child	Left Child
Uncle 색깔	BLACK	BLACK
처리할 노드 위치	Left Child	Left Child



RED node 삽입과 RED-GP 비교 (3)

- Case 4 vs Case 12
  - 동일한 구조
  - RED-GP가 더 일반적
    - NIL ∈ {트리 D}

	Case 4	Case 12
Parent 색깔	RED	RED
Parent 위치	Left Child	Left Child
Uncle 색깔	BLACK	BLACK
처리할 노드 위치	Right Child	Right Child



RED node 삽입과 RED-GP 비교 (4)

- 동일한 구조 Pairs
  - Case 2 vs Case 10
  - Case 5 vs Case 13
  - Case 6 vs Case 14
  - Case 7 vs Case 15
  - Case 8 vs Case 16

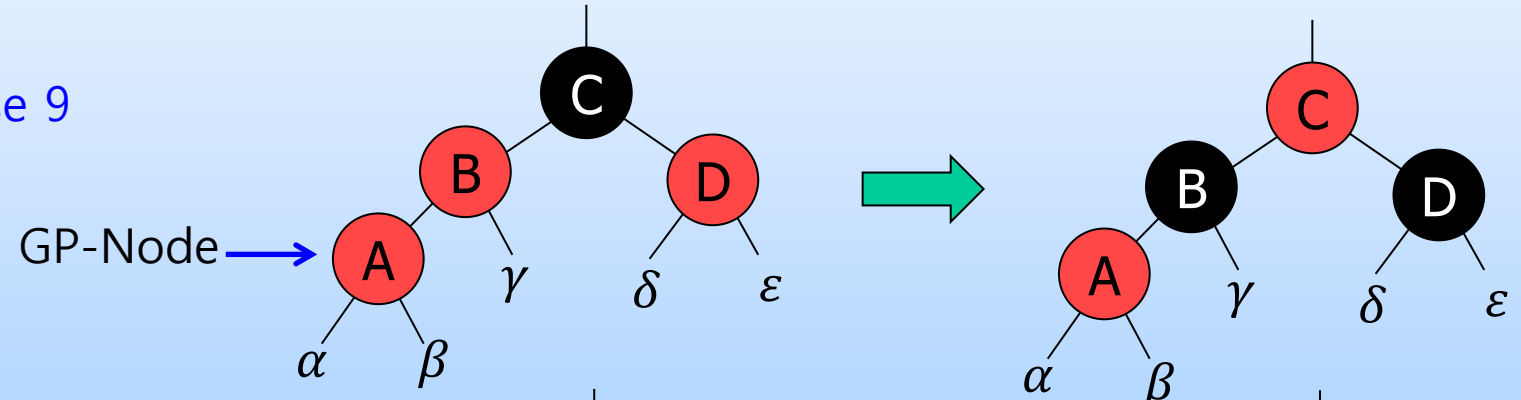
RED-GP violations을 처리하는
Case 9 ~ Case 16만 고려해도 모든 경우 처리가 가능함

Symmetric Cases (1)

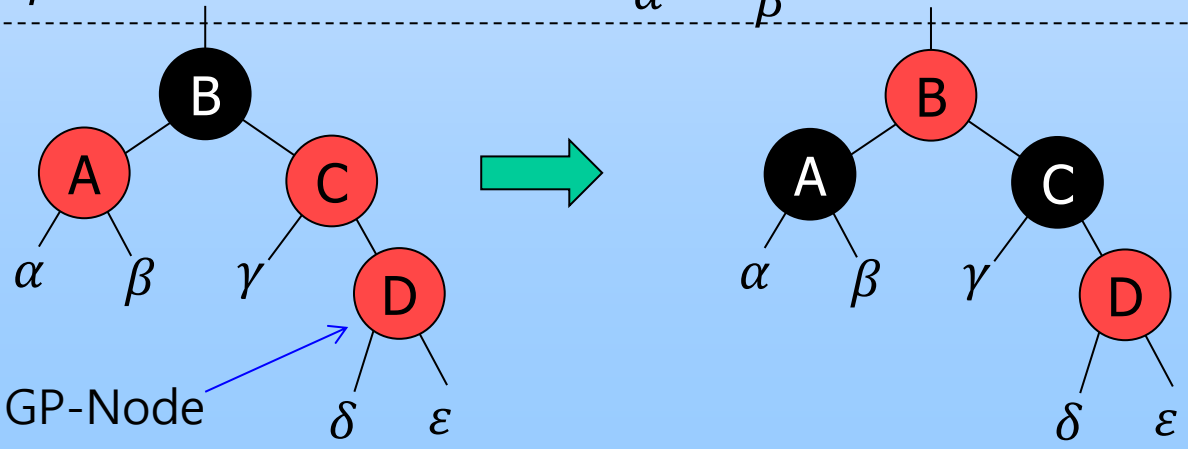
- Case 9 vs Case 14
  - **좌우 symmetric 구조**
  - 코드에서 Left와 Right만 바꾸면 됨

	Case 9	Case 14
Parent 색깔	RED	RED
Parent 위치	**Left** Child	**Right** Child
Uncle 색깔	RED	RED
처리할 노드 위치	**Left** Child	**Right** Child

Case 9



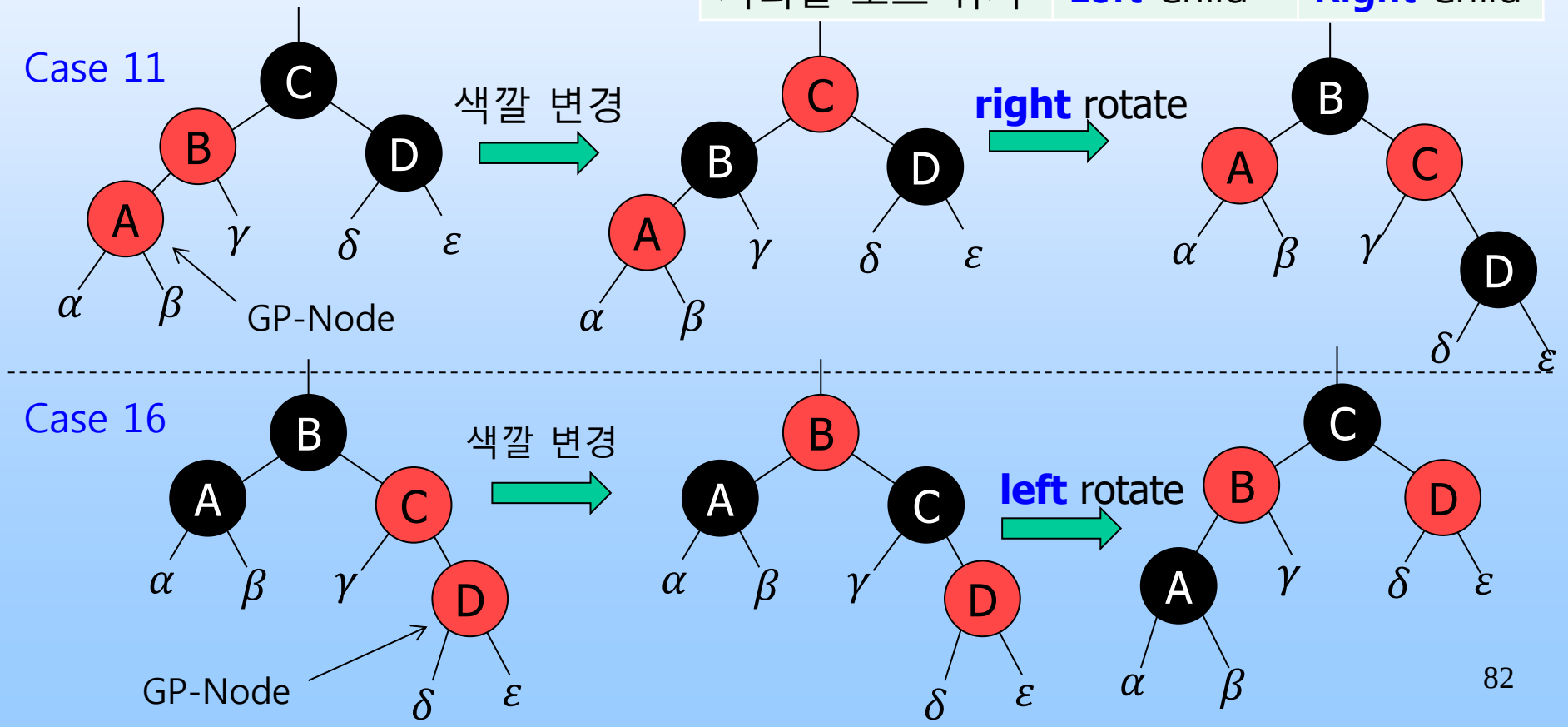
Case 14



Symmetric Cases (2)

- Case 11 vs Case 16
  - **좌우 symmetric 구조**
  - 코드에서 Left와 Right만 바꾸면 됨

	Case 11	Case 16
Parent 색깔	RED	RED
Parent 위치	**Left** Child	**Right** Child
Uncle 색깔	RED	RED
처리할 노드 위치	**Left** Child	**Right** Child



Symmetric Cases (3)

- 좌우 symmetric 대응 관계 정리

GP-Node가 Left Child	좌우 Symmetric	GP-Node가 Right Child
Case 9	↔	Case 14
Case 10	↔	Case 13
Case 11	↔	Case 16
Case 12	↔	Case 15

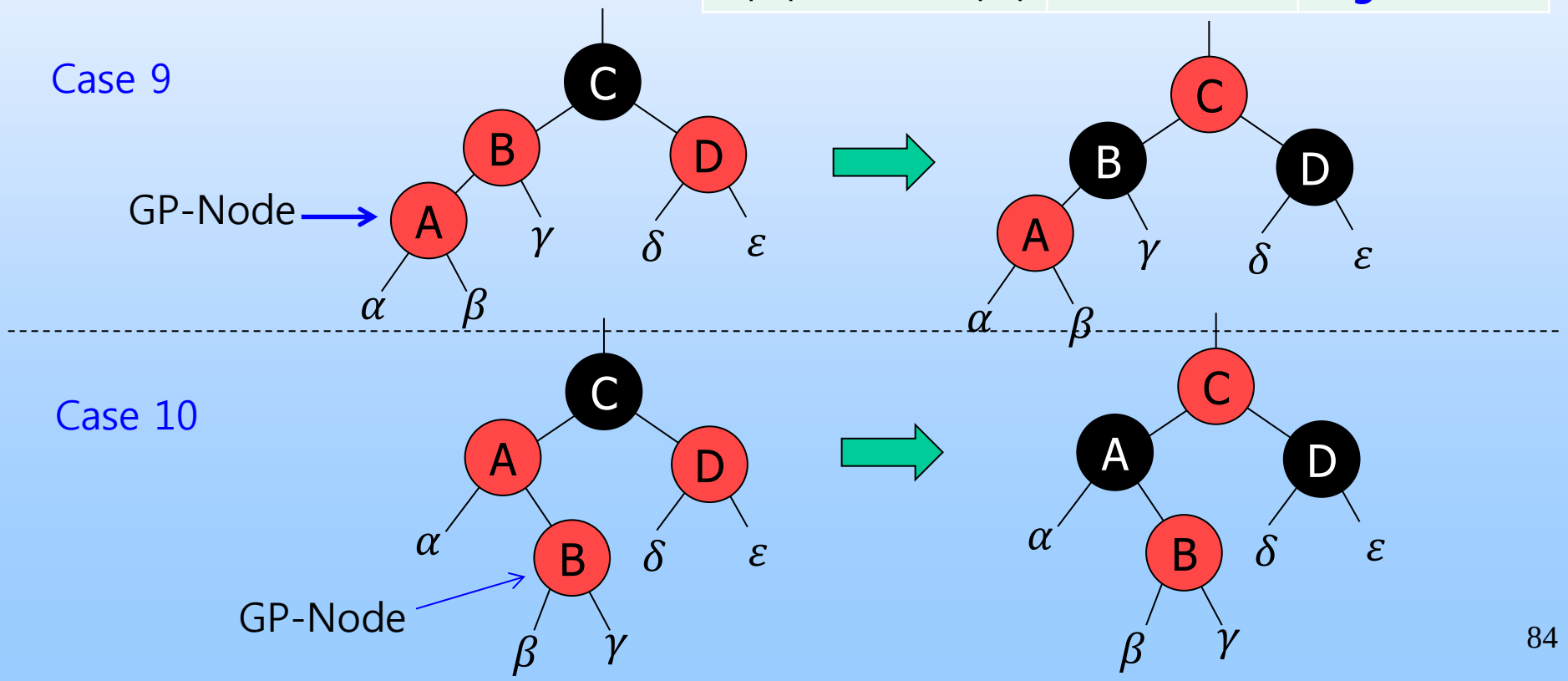
- 좌우 symmetric 대응 관계의 활용
  - 한쪽 코드만 구현한 후 구현된 결과에서 Left와 Right 이름만 바꾸면 대칭적인 다른 한쪽 코드가 완성된다.

Symmetric 확인 방법은 대칭성이 있는 알고리즘 구현할 때 코드 검증 및 안정적 코드 구현에 유용한 방법이다.

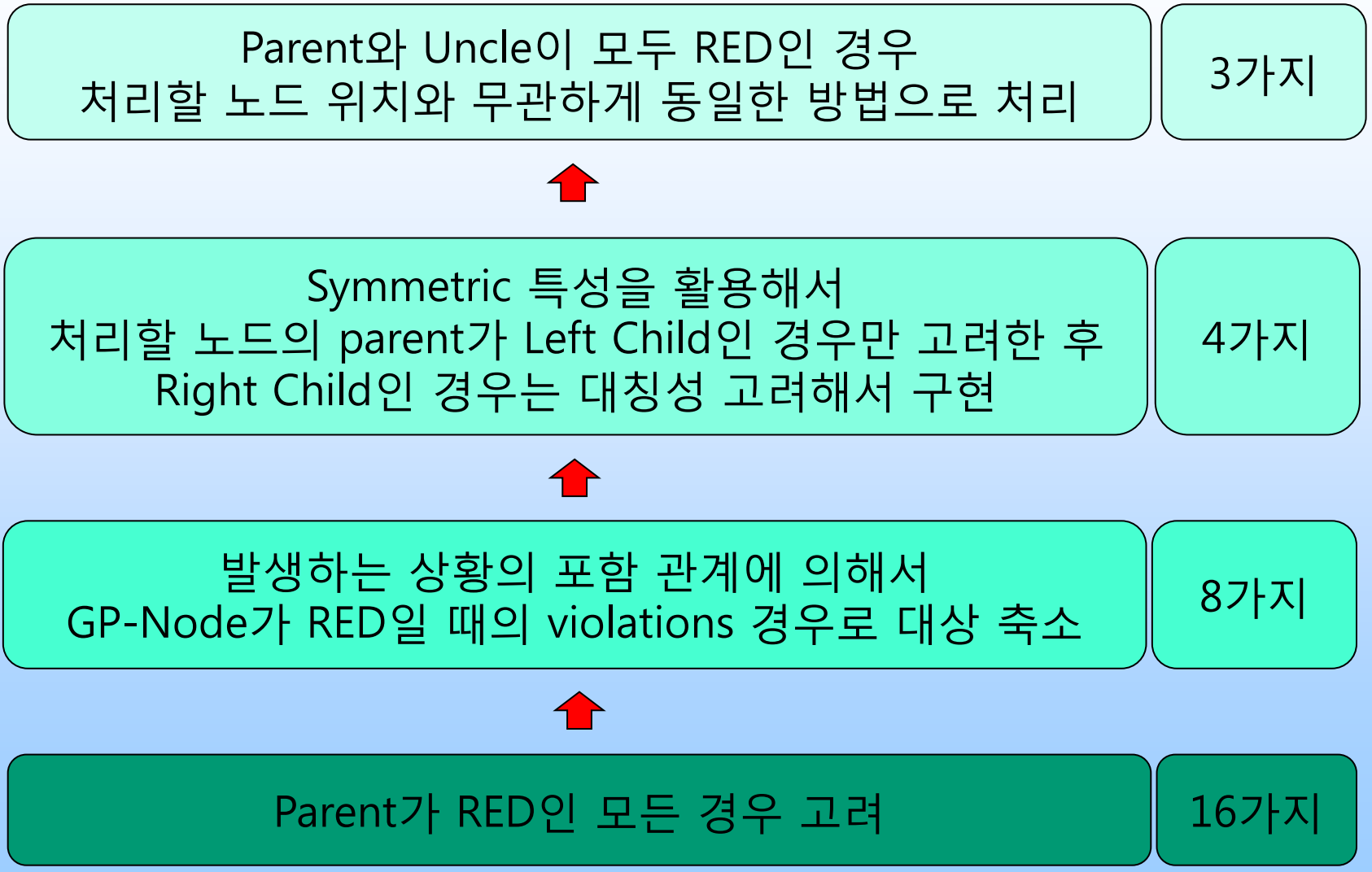
동일하게 처리되는 경우

- Case 9 vs Case 10
  - GP-Node의 위치를 고려할 필요 없음

	Case 9	Case 10
Parent 색깔	RED	RED
Parent 위치	Left Child	Left Child
Uncle 색깔	RED	RED
처리할 노드 위치	**Left** Child	**Right** Child

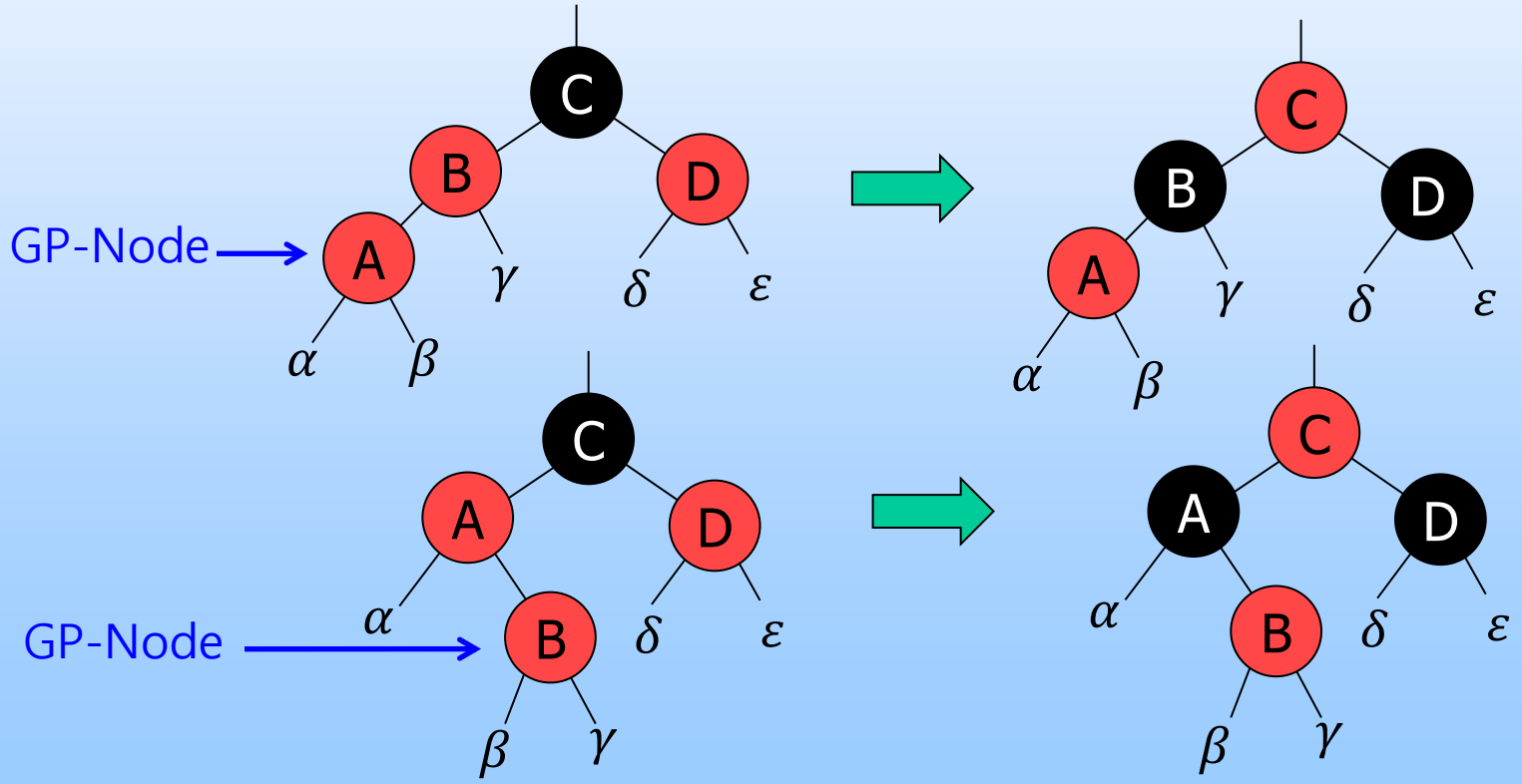


경우의 수 축소 결과


Parent와 Uncle이 모두 RED인 경우
처리할 노드 위치와 무관하게 동일한 방법으로 처리
3가지
Symmetric 특성을 활용해서
처리할 노드의 parent가 Left Child인 경우만 고려한 후
Right Child인 경우는 대칭성 고려해서 구현
4가지
발생하는 상황의 포함 관계에 의해서
GP-Node가 RED일 때의 violations 경우로 대상 축소
8가지
Parent가 RED인 모든 경우 고려
16가지

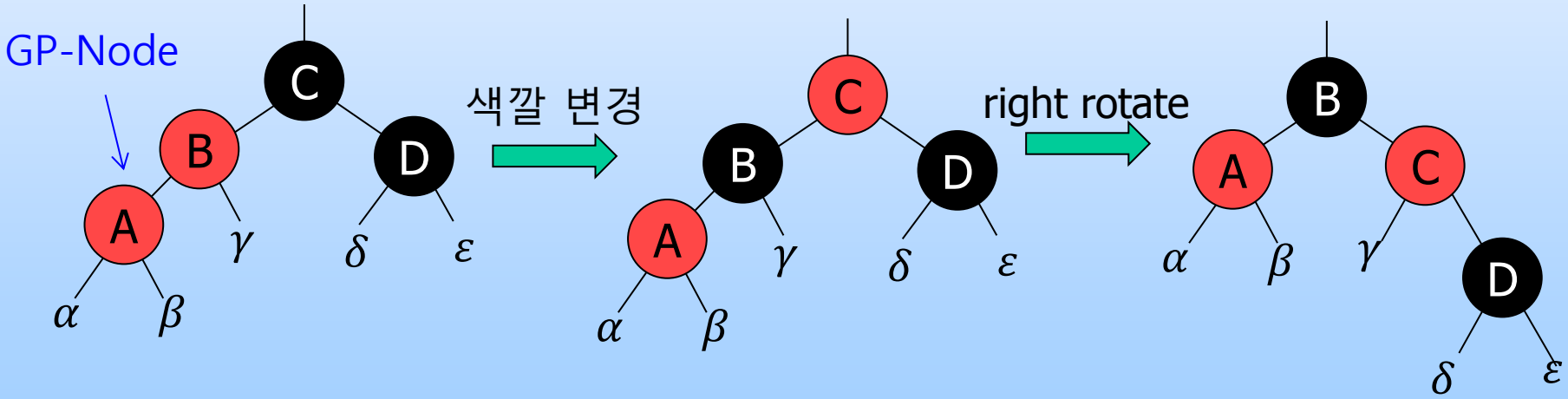
경우의 수 최종 정리 (1)

- 최종 Case 1 (앞에서 설명한 case 9와 case 10)
  - Parent 색깔 : RED
  - Parent의 위치 : left child
  - Uncle 색깔 : RED



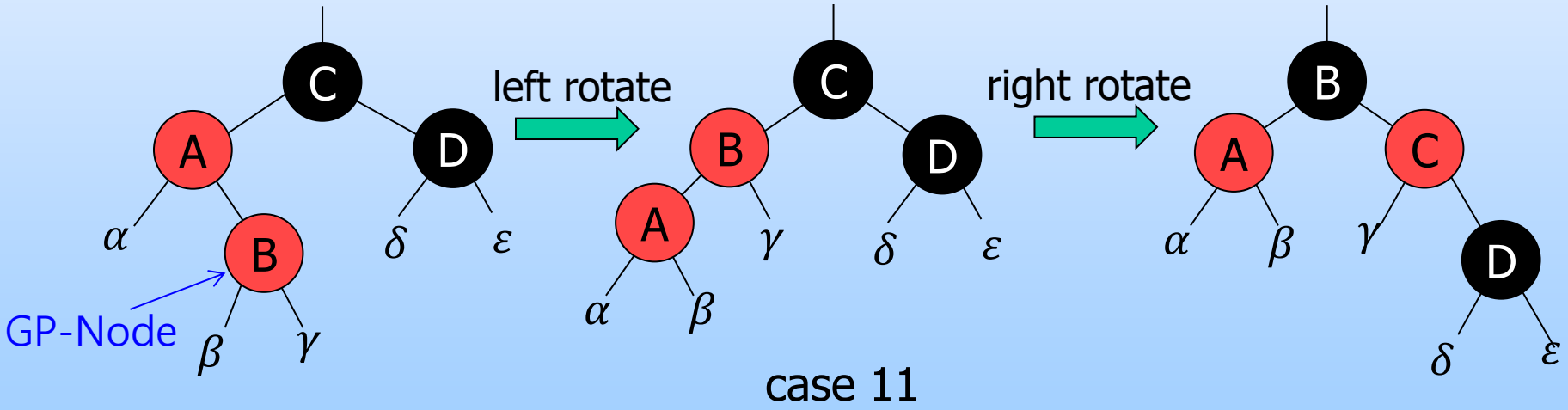
경우의 수 최종 정리 (2)

- 최종 Case 2 (앞에서 설명한 case 11)
  - Parent 색깔 : RED
  - Parent의 위치 : left child
  - Uncle 색깔 : BLACK
  - 처리할 노드의 위치: left child



경우의 수 최종 정리 (3)

- 최종 Case 3 (앞에서 설명한 case 12)
  - Parent 색깔 : RED
  - Parent의 위치 : Left Child
  - Uncle 색깔 : BLACK
  - 처리할 노드의 위치: Right Child



Parent의 색깔이 **BLACK**인 경우

- 직전에 정리한 경우의 수는 parent의 색깔이 RED인 경우였음

- 슬라이드 31과 32의 내용을 remind하면…
  - 삽입된 노드 혹은 현재 처리할 노드(Grand-Parent)가 root인 경우에 root를 BLACK으로 지정
    - 즉, 노드를 RED에서 BLACK로 변경

Pseudo Codes 기능

- RB-INSERT(T, newNode)
  - Node를 insert
  - Insert 후 RB-INSERT-FIXUP을 호출
  - Binary search tree의 insert와 거의 동일

- RB-INSERT-FIXUP(T, fixupNode)
  - fixupNode에 의해서 violation이 발생하는지 확인
  - Violation이 발생하면 rotation 및 색깔 보정
  - 최종적으로 T가 red-black tree가 되도록 함

```
RB-INSERT (T, newNode)
1    parentNode = T.nil                    // sentinel node로 초기화
2    curNode = T.root
3    while curNode ≠ T.nil                 // sentinel node 인지 확인
4         parentNode = curNode
5         if newNode.key < curNode.key
6              curNode = curNode.left
7         else  curNode = curNode.right
8     newNode.parent = parentNode
9     if parentNode == T.nil
10         T.root = newNode
11    elseif newNode.key < parentNode.key
12         parentNode.left = newNode
13    else   parentNode.right = newNode
14    newNode.left = T.nil                  // newNode child 초기화
15    newNode.right = T.nil
16    newNode.color = RED                   // newNode 색깔 초기화
17    RB-INSERT-FIXUP(T, newNode) // newNode에 의한 위배 처리
```

```
RB-INSERT-FIXUP (T, fixupNode)
1    while fixupNode.parent.color == RED // parent-child 모두 red → 위배
2        if fixupNode.parent == fixupNode.parent.parent.left
3            uncleNode = fixupNode.parent.parent.right
4            if uncleNode.color == RED
5                fixupNode.parent.color = BLACK                  // case 1
6                uncleNode.color = BLACK                         // case 1
7                fixupNode.parent.parent.color = RED             // case 1
8                fixupNode = fixupNode.parent.parent
9            else
10               if fixupNode == fixupNode.parent.right
11                   fixupNode = fixupNode.parent                // case 3
12                   LEFT-ROTATE(T, fixupNode)                   // case 3
13               fixupNode.parent.color = BLACK                  // case 2
14               fixupNode.parent.parent.color = RED             // case 2
15               RIGHT-ROTATE(T, fixupNode.parent.parent)        // case 2
16       else // if then 절에서 left와 right를 바꾼 경우와 같음.
17   T.root.color = BLACK // case 1과 parent가 black인 경우에 적용
```

Analysis

- RB-INSERT
  - Red-black tree의 height가 $O(\lg n)$이므로 $O(\lg n)$ time 소요
- RB-INSERT-FIXUP
  - 반복문은 case 1의 경우에만 수행
  - 매 반복문 수행마다 level은 2개씩 위로 움직임
  - 따라서, while loop 수행 횟수는 $O(\lg n)$
- 특징
  - Rotation은 최대 2번만 수행됨

실습 주제(4)

- 문제 풀이
  - 알고리즘 교재 13.3-2

13.3-2

Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

목 차

- 소개
- Rotation
- Insertion
- Deletion

전 략

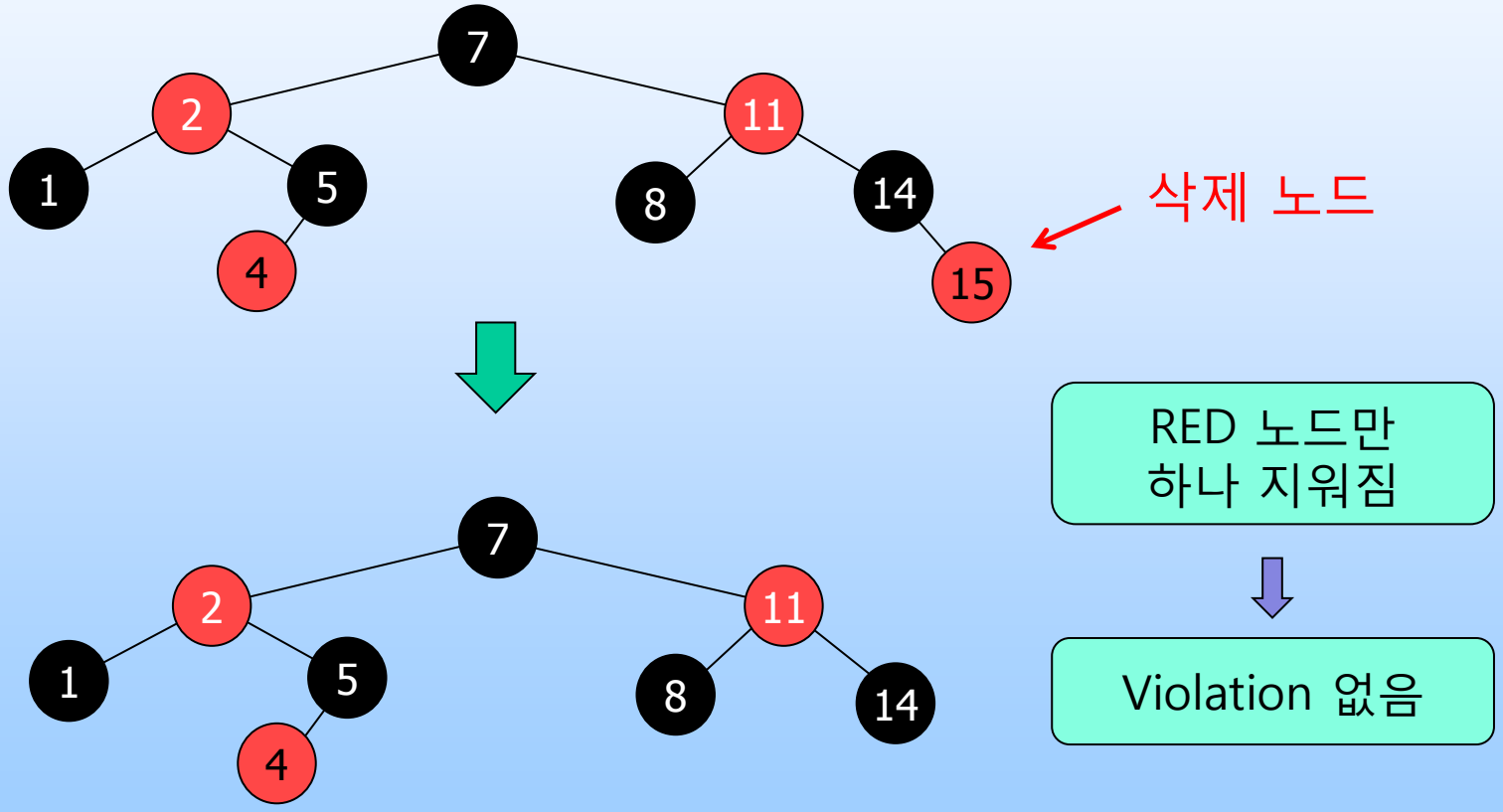
Binary search tree의 특성을 유지하면서 delete

지워진 색깔이 BLACK인지 확인

지워진 BLACK에 의해 black height가 1 감소한 경로에 black node가 1개 추가되도록 rotation 및 색깔 조정

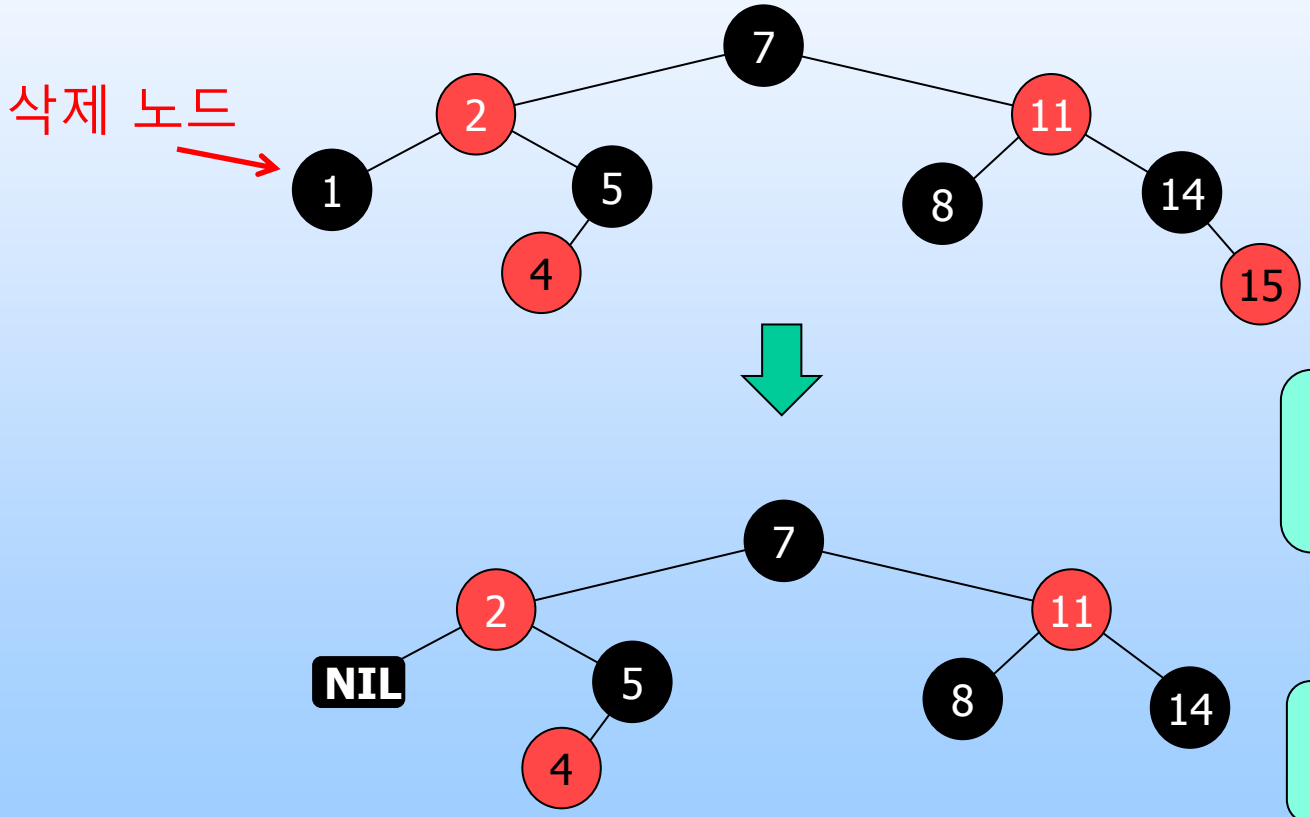
예제로 직관 구하기 (1)

- 삭제된 노드의 child 개수가 0개인 경우 (1)
  - 노드 15 삭제



예제로 직관 구하기 (2)

- 삭제된 노드의 child 개수가 0개인 경우 (2)
  - 노드 1 삭제

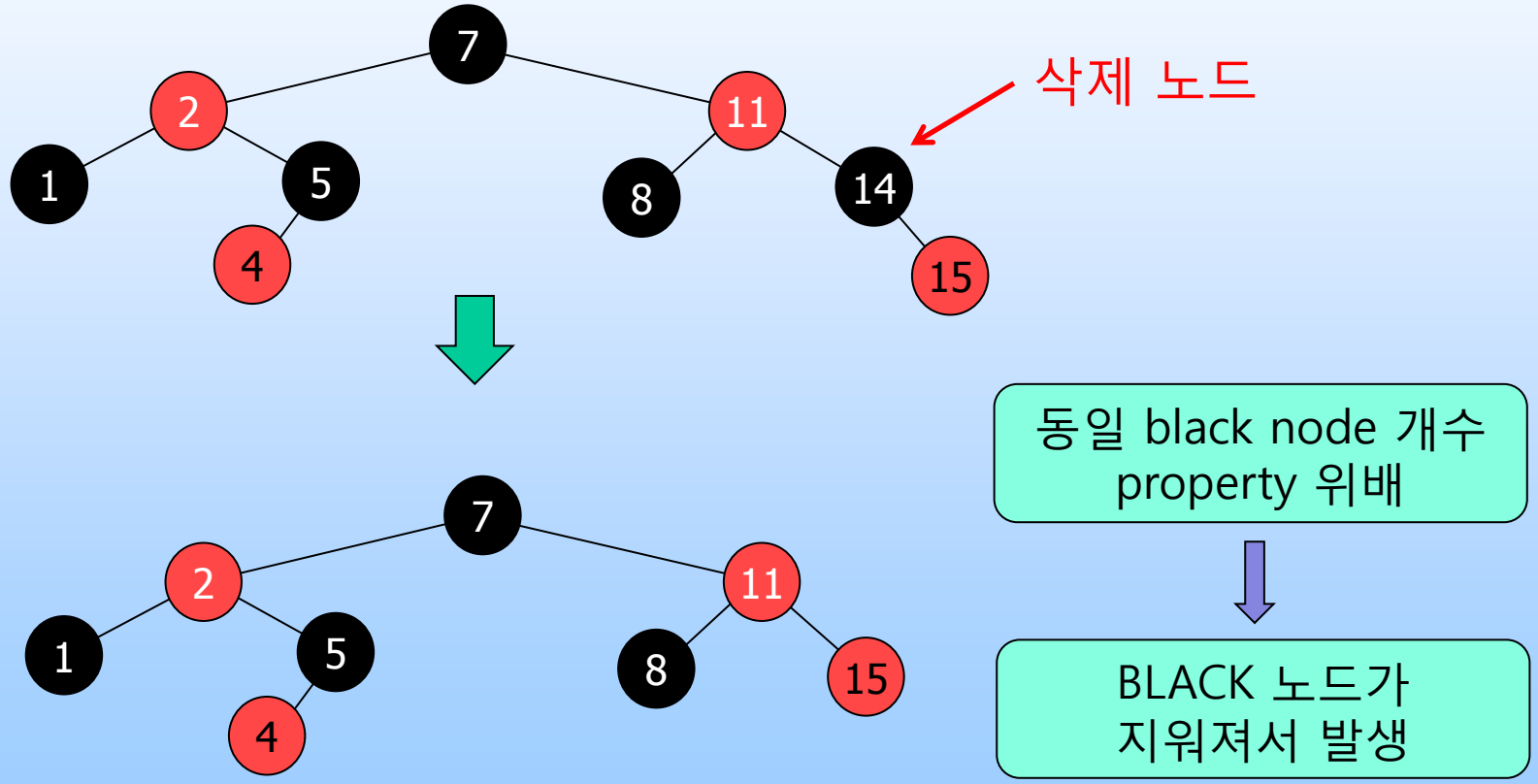


동일 black node 개수 property 위배
(NIL 표시 대부분 생략)

BLACK 노드가 지워져서 발생

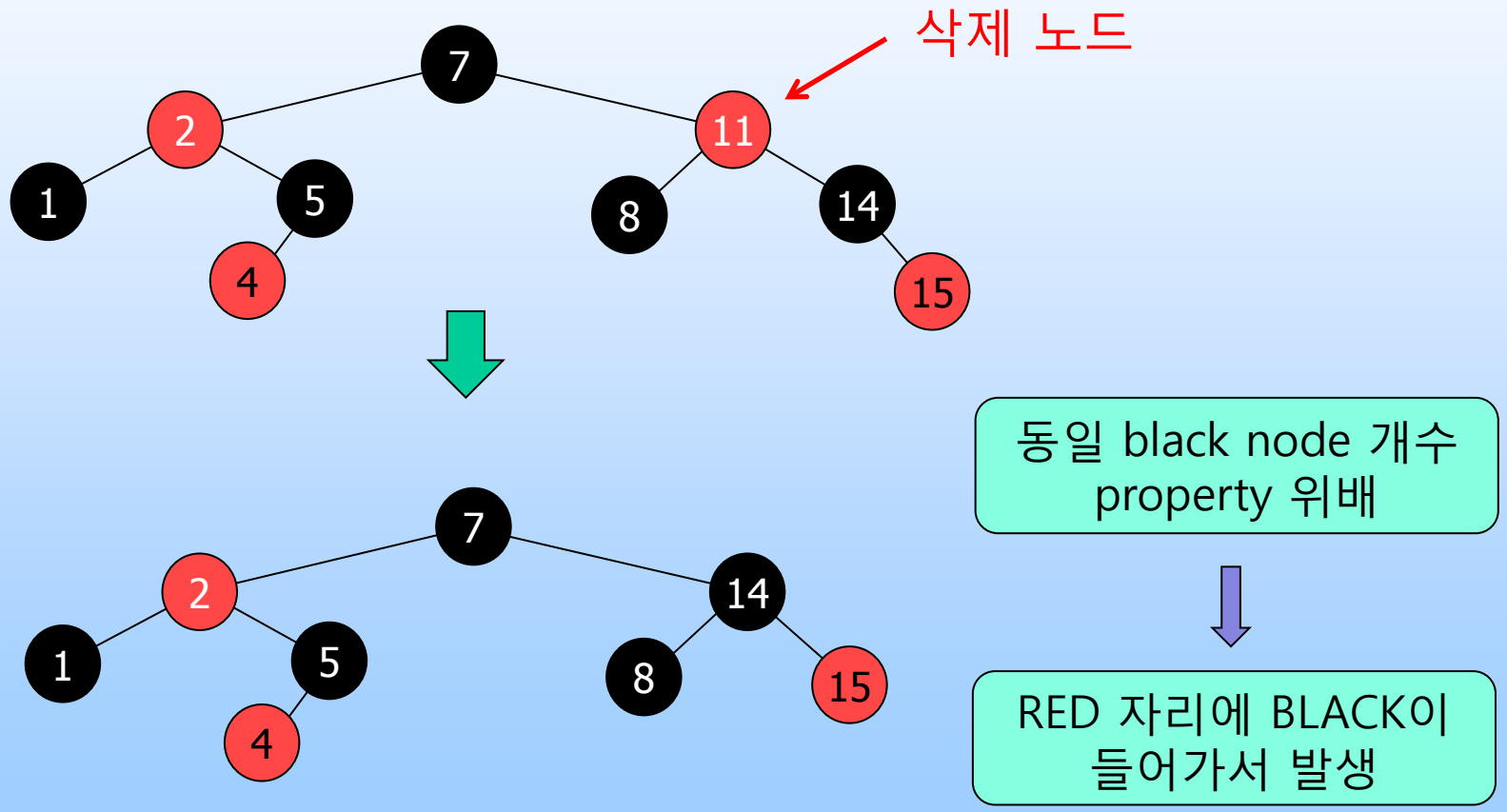
예제로 직관 구하기 (3)

- 삭제된 노드의 child 개수가 1개인 경우
  - 노드 14 삭제



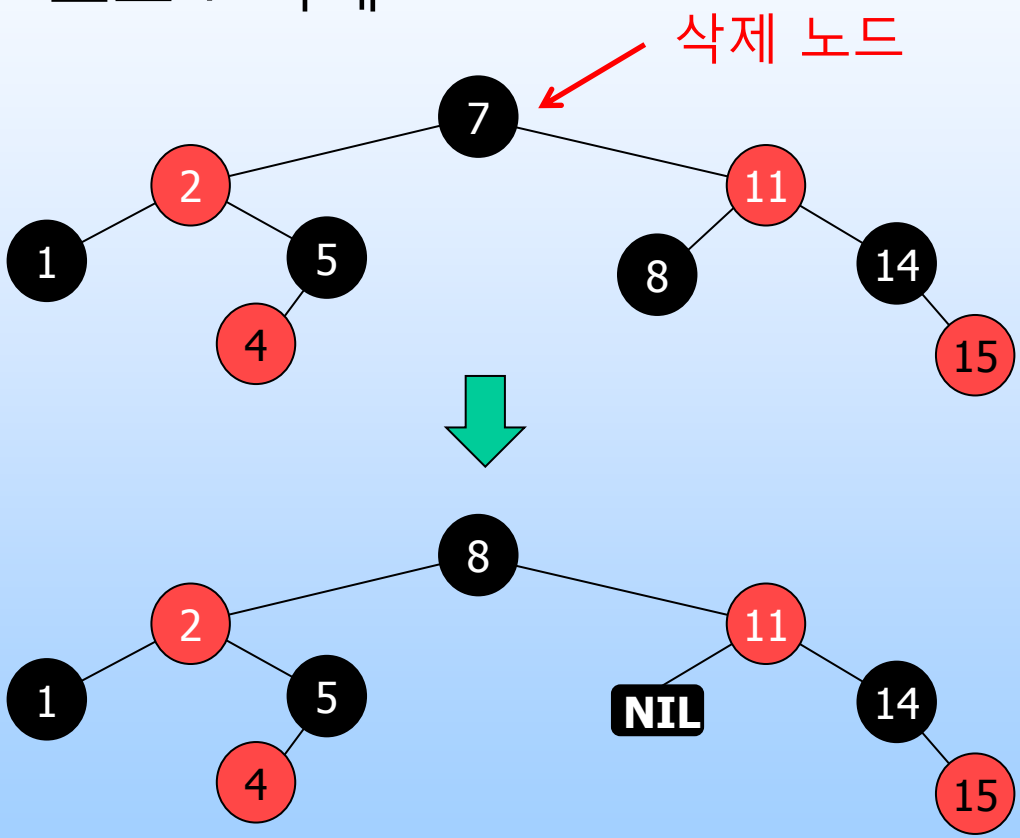
예제로 직관 구하기 (4)

- 삭제된 노드의 child 개수가 2개인 경우
  - 노드 11 삭제



예제로 직관 구하기 (5)

- 삭제된 노드의 child 개수가 2개인 경우
  - 노드 7 삭제



동일 black node 개수 property 위배

BLACK 노드가 삭제되어 발생

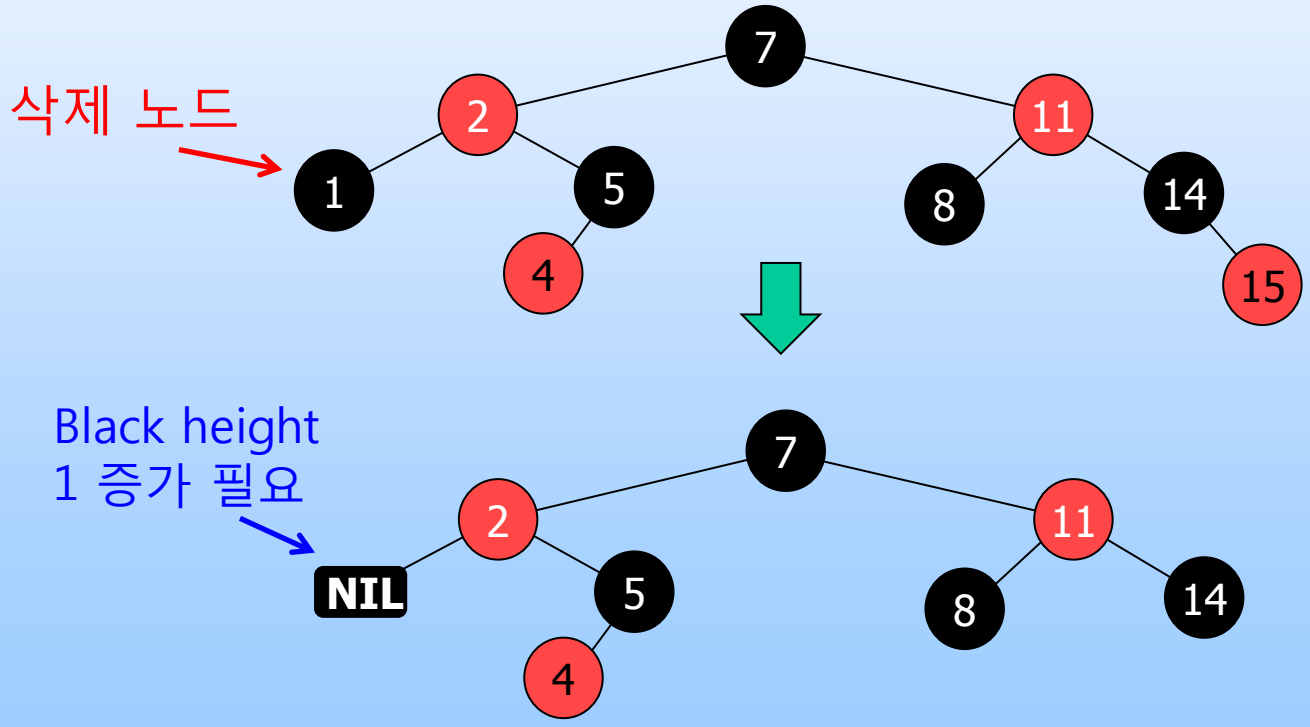
노드 삭제에 의한 특성 위배

- 분류
  - 삭제된 노드의 child가 0개인 경우
  - 삭제된 노드의 child가 1개인 경우
  - 삭제된 노드의 child가 2개인 경우

삭제된 노드의 child 개수로 구분해서
다루는 것이 delete 동작 이해의 첫 번째 실마리

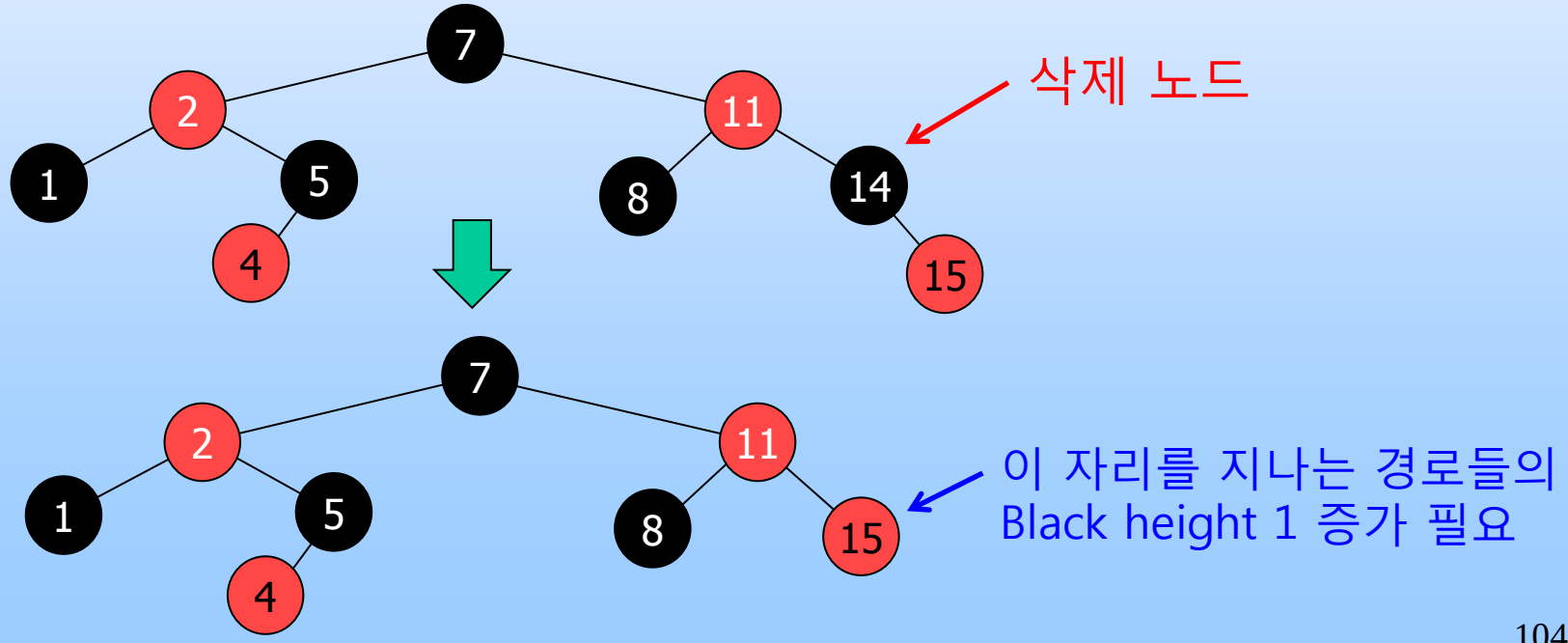
삭제 노드 Child 개수: 0개

- 삭제 노드가 RED인 경우
  - violation 없음
- 삭제 노드가 BLACK인 경우
  - 삭제된 노드 자리의 black height 1 증가 필요



삭제 노드 Child 개수: 1개

- 삭제된 노드가 red인 경우
  - Violation 없음.
  - 왜냐하면, 삭제된 노드의 child와 parent는 black이기 때문임.
- 삭제된 노드가 black인 경우
  - 삭제된 노드 자리를 지나는 경로의 black height 1 증가 필요

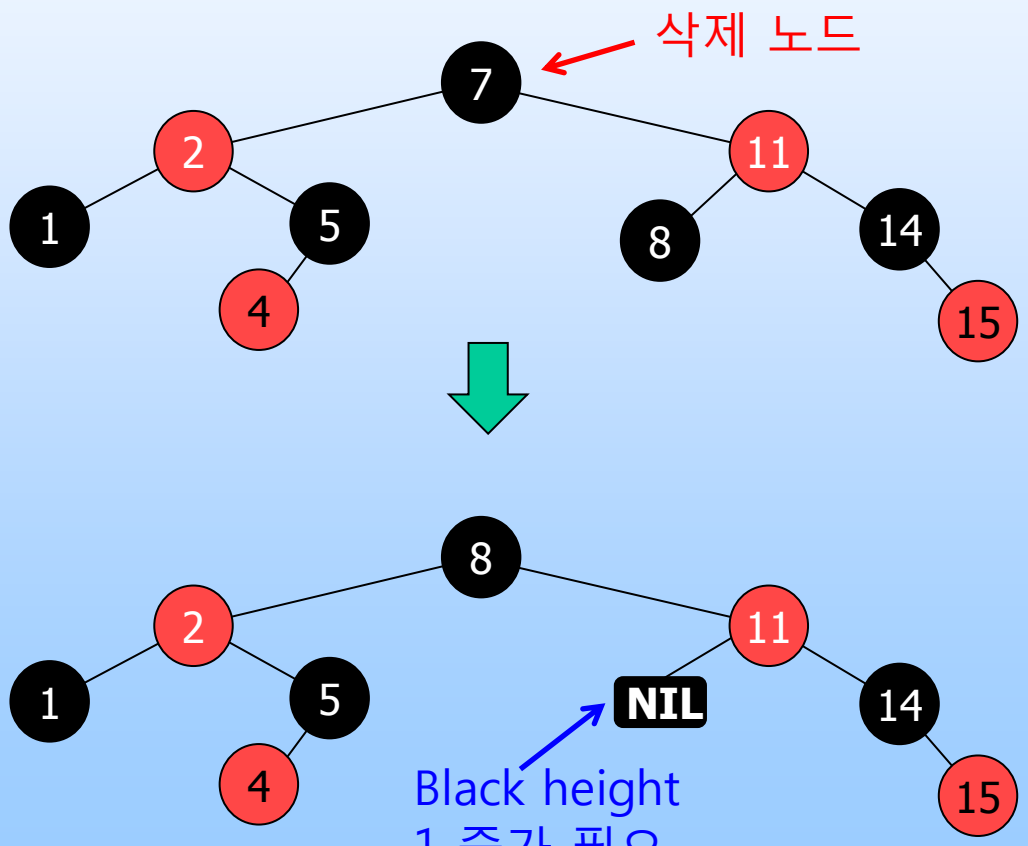


삭제 노드 Child 개수: 2개 (1)

- 아이디어
  - 삭제 노드 자리의 색깔 보존
    - successor를 삭제 노드 색깔로 변경한 후 삭제 노드 자리에 이식
  - Successor의 원래 색깔이 RED인 경우
    - violation 없음
  - Successor의 원래 색깔이 BLACK인 경우
    - Successor가 있던 자리에 오는 노드를 지나는 경로의 black height 1 증가 필요

삭제 노드 Child 개수: 2개 (2)

- 예제
  - Successor의 원래 색깔이 BLACK
  - Successor의 자리로 오는 노드가 BLACK



특성 위배 해결 방법 요약 (1)

- 삭제 노드의 child가 0개 혹은 1개일 때
  - 삭제 노드 색깔이 RED인 경우
    - Violation이 없으므로 delete 완료
  - 삭제 노드 색깔이 BLACK인 경우
    - 해당 자리를 지나는 경로의 black height를 1 증가
    - 증가 방법은 이후의 슬라이드에서 설명…

특성 위배 해결 방법 요약 (2)

- 삭제 노드의 child가 2개인 경우
  - Successor의 원래 색깔이 RED인 경우
    - violation이 없으므로 delete 완료
  - Successor가 원래 색깔이 BLACK인 경우
    - Successor가 있던 자리를 지나는 경로의 black height를 1 증가
    - 증가 방법은 이후의 슬라이드에서 설명…

특정 노드를 지나는 경로의 black height를
1 증가시키는 방법이 두 번째 실마리

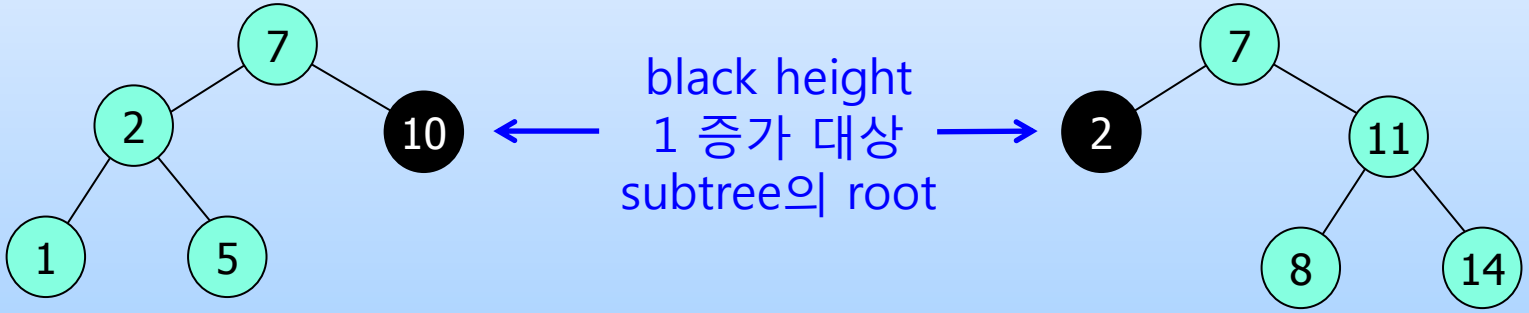
Black Height 1 증가 방법

- 목표
  - 노드 x를 지나는 모든 경로의 black height 1 증가

- 방법 요약
  - 노드 x가 red이면
    - 노드 x를 black으로 변경하고 종료
  - 노드 x가 black이면
    - Parent를 기준으로 rotation 및 색깔 변경으로 노드 x를 지나는 경로의 black height를 1 증가 혹은
    - Sibling을 지나는 모든 경로의 black height를 1 감소시킨 후 black height 증가 기준을 노드 x에서 노드 x의 parent로 변경
    - 위 작업을 반복하다가 parent가 root이면 root를 black으로 지정한 후 작업 종료

경우의 수 (1)

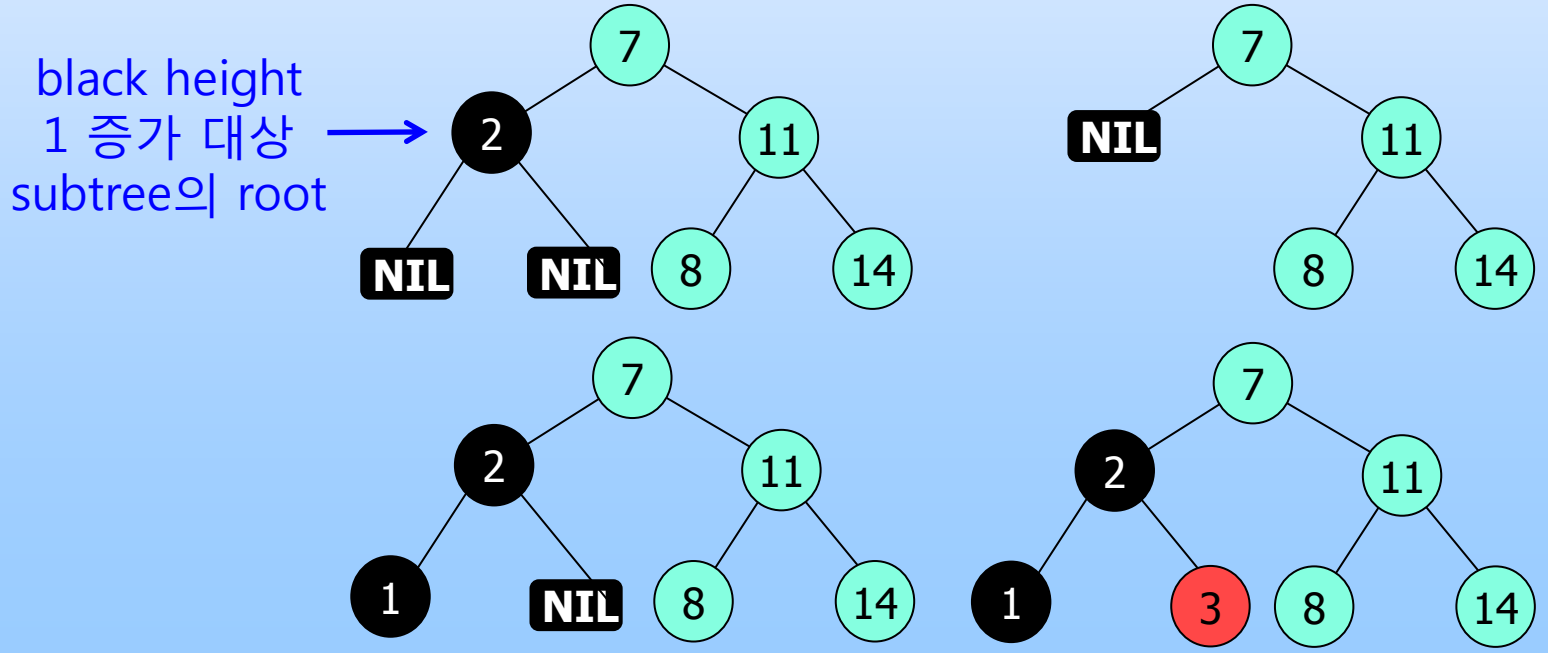
- 분류 기준
  - 노드의 위치 (2가지) : left child 혹은 right child
  - Parent의 색깔 (2가지) : red or black
  - Sibling의 색깔 (2가지) : red or black
  - Sibling의 children 색깔 (4가지)
    - black-black, black-red, red-black, red-red



- 총 32 종류
  - 종류 수 줄이는 방법은?
- Black height 1 증가 대상 노드가 NIL이거나 child 개수가 0, 1, 2인지에 대해서는 분리해서 다루지 않음.
  - Why? 설명은 다음 슬라이드에...

경우의 수 (2)

- Black height 1 증가 대상의 특성은 고려하지 않음. Why?
  - 특성
    - 노드가 NIL인지 여부
    - child 개수, 색깔 등
  - 각 경우를 분석해 보면 delete 처리 과정에서 해당 노드의 정보가 바뀌지 않는다는 것을 알 수 있음.
  - 따라서, NIL을 포함해서 모두 동일한 방법으로 처리될 수 있음.



경우의 수 (2)

Case	노드 위치	Parent	Sibling	Sibling의 child	가능한 조합
1	Left child	Black	Black	black – black	O
2				black – red	O
3				red – black	O
4				red – red	O
5			Red	black – black	O
6				black – red	X
7				red – black	X
8				red – red	X
9		red	Black	black – black	O
10				black – red	O
11				red – black	O
12				red – red	O
13			Red	black – black	X
14				black – red	X
15				red – black	X
16				red – red	X

경우의 수 (3)

◆ 가능한 조합의 경우의 수

Case	노드 위치	Parent	Sibling	Sibling의 child	가능한 조합
1	Left child	Black	Black	black – black	O
2				black – red	O
3				red – black	O
4				red – red	O
5			Red	black – black	O
9		red	Black	black – black	O
10				black – red	O
11				red – black	O
12				red – red	O

경우의 수 (4)

- Parent가 Black인 경우와 Red인 경우 동일한 방법으로 처리 가능
  - 각 경우 상세 분석 슬라이드 참고…

Case	노드 위치	Parent	Sibling	Sibling의 child	가능한 조합
1	Left child	Black / Red	Black	black – black	O
2				black – red	O
3				red – black	O
4				red – red	O
5		Black	Red	black – black	O

- Sibling의 children이 black-red, red-red인 경우 동일 처리 가능
  - 각 경우 상세 분석 슬라이드 참고…

Case	노드 위치	Parent	Sibling	Sibling의 child	가능한 조합
1	Left child	Black / Red	Black	black – black	O
2				Black/red – red	O
3				red – black	O
5		Black	Red	black – black	O

경우의 수 요약

◆ 노드 위치가 left child인 경우

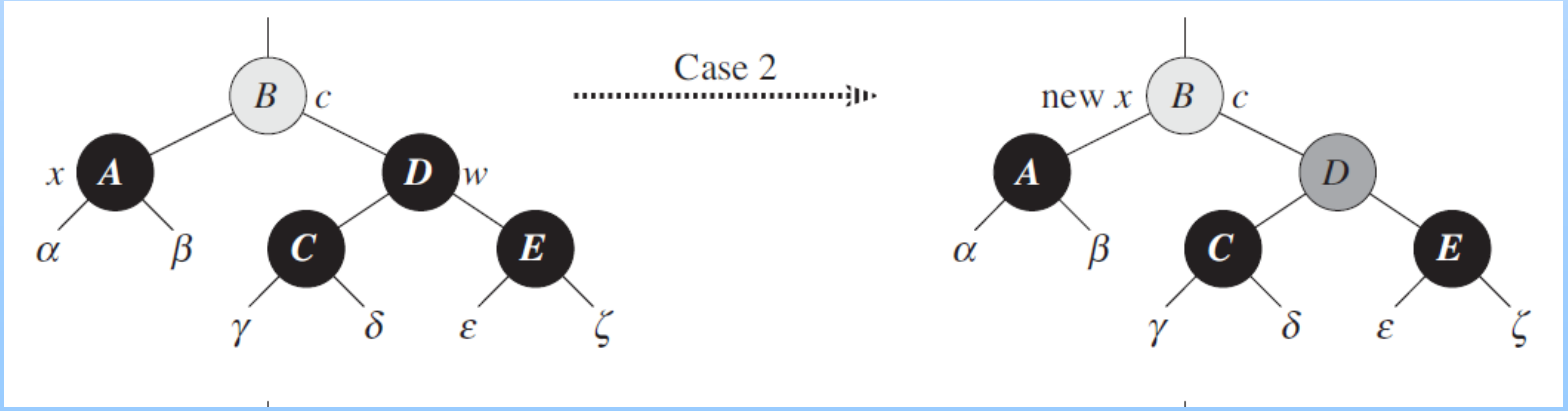
Case	노드 위치	Parent	Sibling	Sibling의 child	가능한 조합
1	Left child	Black / Red	Black	black – black	O
2				Black/red – red	O
3				red – black	O
5		Black	Red	black – black	O

◆ 노드 위치가 right child인 경우

Case	노드 위치	Parent	Sibling	Sibling의 child	가능한 조합
17	Right child	Black / Red	Black	black – black	O
18				Red – Black/red	O
19				black – red	O
24		Black	Red	black – black	O

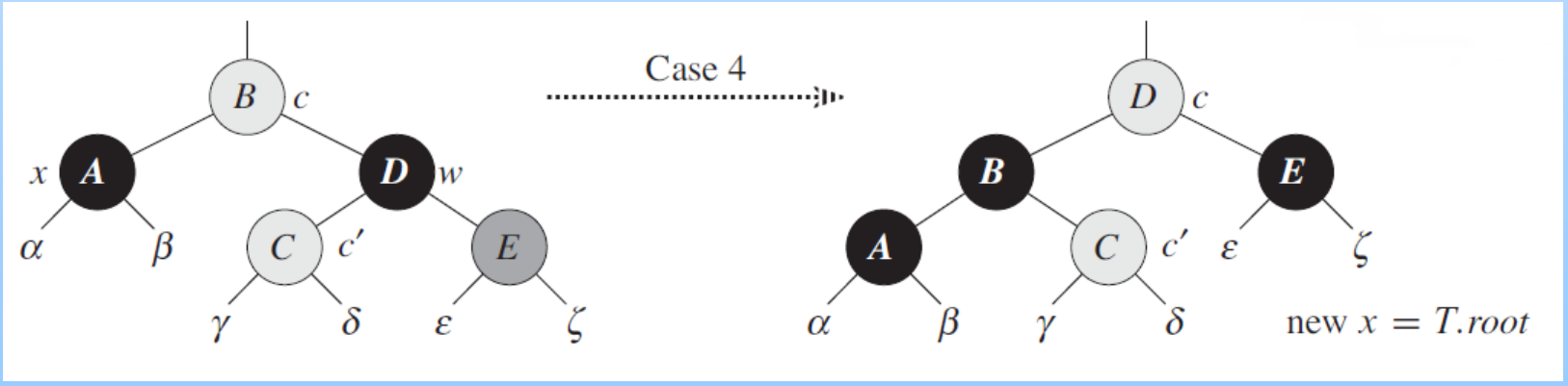
Cases 상세 분석 (1)

- Case 1 (교재 Case 2)
  - x : 처리할 노드
  - w : sibling
  - Sibling을 red로 변경
    - Sibling을 지나는 경로의 black height 1 감소
  - Parent가 red이면 black으로 지정한 후 delete 종료
    - Parent를 지나는 모든 경로의 black height 1 증가했으므로
  - Parent가 black이면 parent를 처리할 새로운 노드 x 로 지정
    - 노드 x의 black height 1 증가 작업 다시 수행
    - 노드 x가 root이면 delete 완료 ← tree의 black height 1 감소



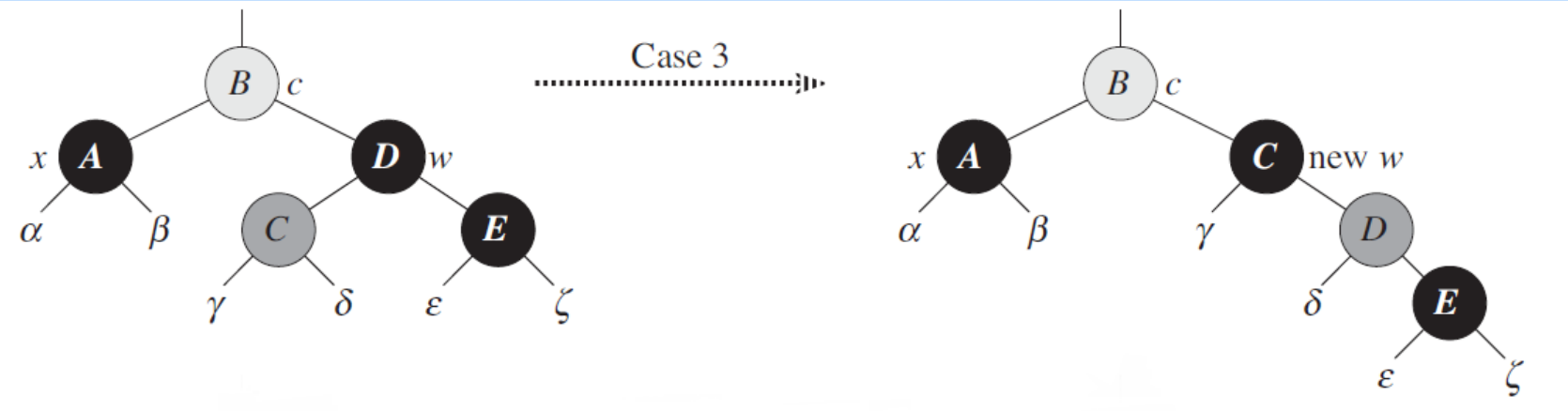
Cases 상세 분석 (2)

- Case 2 (교재 Case 4)
  - x : 처리할 노드
  - w : sibling
  - 수행 작업
    - Sibling을 parent의 색깔로 지정 (parent는 black 혹은 red)
    - Parent를 (sibling의 색깔) black으로 지정
    - Sibling의 rightChild를 red->black으로 변경
    - Left rotation
  - x를 지나는 모든 경로의 black height가 1증가 & 다른 경로의 black height 변경 없음. → delete 완료



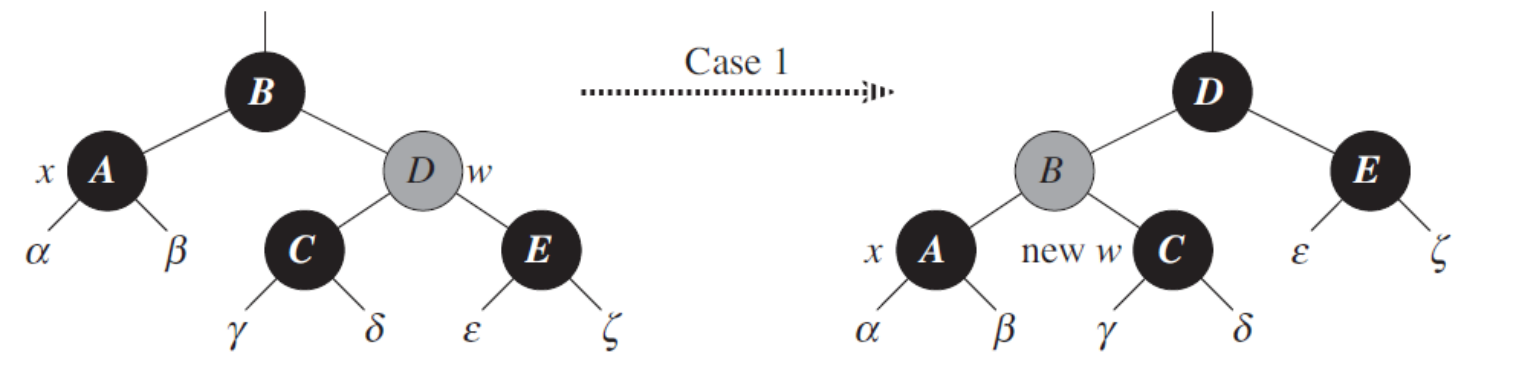
Cases 상세 분석 (3)

- Case 3 (교재 Case 3)
  - x : 처리할 노드
  - w : sibling
  - 수행 작업
    - Sibling의 left를 black으로 변경
    - Sibling을 red로 변경
    - Sibling을 중심으로 right rotation
  - Case 2 (교재 case 4)로 변경 → Case 2 다시 적용 → delete 완료



Cases 상세 분석 (4)

- Case 5 (교재 Case 1)
  - x : 처리할 노드
  - w : sibling
  - 수행 작업
    - 목표: siblilng이 black이 되도록 변경
    - parent를 red로 변경
    - Sibling을 black으로 변경
    - Parent를 중심으로 left rotation
  - 노드 C가 새로운 sibling이 됨 → sibling이 black인 경우로 변경
    - case 1 ~ case 4 중 하나로 변경



```
RB-TRANSPLANT (T, oldNode, plantNode)   // oldNode를 plantNode로 교체
1       if oldNode.parent == T.nil
2               T.root = plantNode
3       elseif oldNode == oldNode.parent.left
4               oldNode.parent.left = platNode
5       else  oldNode.parent.right = plantNode
6       plantNode.parent = oldNode.parent   // plantNode가 NIL일 때 예외 처리 없음
```

```
RB-DELETE (T, node)
1     // tmpNode = node                  // 교재의 pseudo code에 있지만 불필요한 코드
2     erasedColor = node.color
3     if node.left == T.nil               // left child가 없는 경우
4          fixupNode = node.right         // 보정 시작 노드 지정. NIL node 경우도 처리
5          RB-TRANSPLANT(T, node, node.right)
6     elseif node.right == T.nil
7          fixupNode = node.left          // 보정 시작 노드 지정
8          RB-TRANSPLANT(T, node, node.left)
9     else successor = TREE-MINIMUM(node.right)
10         erasedColor = successor.color // line 20에서 successor의 color 갱신
11         fixupNode = successor.right // 원래 successor 자리부터 fixup 시작
12         if successor.parent == deletedNode
13              fixupNode.parent = successor // NIL일 경우에는 parent 재지정
14         else RB-TRANSPLANT(T, successor, succssor.right)
15              successor.right = node.right
16              successor.right.parent = successor
17         RB-TRANSPLANT(T, node, successor)
18         successor.left = node.left
19        successor.left.parent = successor
20        successor.color = node.color
21     if erasedColor == BLACK
22          RB-DELETE-FIXUP(T, fixupNode)
```

```
RB-DELETE-FIXUP (T, fixupNode)
1     while  fixupNode ≠ T.root and fixupNode.color == BLACK
2         if fixupNode == fixupNode.parent.left
3             sibling = fixupNode.parent.right
4             if sibling.color == RED
5                 sibling.color = BLACK                    // case 5 (교재 case 1)
6                 fixupNode.parent.color = RED             // case 5 (교재 case 1)
7                 LEFT-ROTATE(T, fixupNode.parent)         // case 5 (교재 case 1)
8                  sibling = fixupNode.parent.right        // case 5 (교재 case 1)
9             if sibling.left.color == BLACK and sibling.right.color == BLACK
10                sibling.color = RED                      // case 1 (교재 case 2)
11                fixupNode = fixupNode.parent             // case 1 (교재 case 2)
12            else
13                 if sibling.right.color == BLACK
14                       sibling.left.color = BLACK        // case 3 (교재 case 3)
15                       sibling.color = RED               // case 3 (교재 case 3)
16                       RIGHT-ROTATE(T, sibling)          // case 3 (교재 case 3)
17                       sibling = fixupNode.parent.right  // case 3 (교재 case 3)
18                sibling.color = finxupNode.parent.color  // case 2 (교재 case 4)
19                fixupNode.parent.color = BLACK           // case 2 (교재 case 4)
20                sibling.right.color = BLACK              // case 2 (교재 case 4)
21                LEFT-ROTATE(T, fixupNode.parent)         // case 2 (교재 case 4)
22                fixupNode = T.root                       // 종료   // case 2 (교재 case 4)
23        else (then 코드에서 "right"와 "left"를 서로 바꾼 상태)
// 43  fixupNode.color = BLACK
```

```
RB-DELETE-FIXUP (T, fixupNode)
23        else                                  // fixupNode == fixupNode.parent.right 일 때
24             sibling = fixupNode.parent.left
25             if sibling.color == RED
26                  sibling.color = BLACK
27                  fixupNode.parent.color = RED
28                  RIGHT-ROTATE(T, fixupNode.parent)
29                  sibling = fixupNode.parent.left
30             if sibling.right.color == BLACK and sibling.left.color == BLACK
31                  sibling.color == RED
32                  fixupNode = fixupNode.parent
33             else if sibling.left.color == BLACK
34                        sibling.right.color = BLACK
35                         sibling.color = RED
36                         LEFT-ROTATE(T, sibling)
37                         sibling = fixupNode.parent.left
38                   sibling.color = finxupNode.parent.color
39                   fixupNode.parent.color = BLACK
40                   sibling.left.color = BLACK
41                   RIGHT-ROTATE(T, fixupNode.parent)
42                  fixupNode = T.root
43    fixupNode.color = BLACK
```

Delete Analysis

- Case 1 (교재 case 2)과 case 5가 반복적으로 수행될 수 있는 경우임
- Case 1 수행 횟수
  - Sibling을 지나는 경로의 black height를 1 낮추고 parent가 black인 경우
  - 최대 root까지 올라감
  - Tree의 depth는 $O(\lg n)$
- Case 5에서 처리할 노드의 depth가 1증가 현상 분석
  - Sibling과 parent의 sibling이 모두 black이라서 case 5가 다시 발생하기 전까지 처리할 노드의 depth가 최소 2 이상 감소하게 됨.
  - 따라서, 최악의 경우 $2\lg n$ 연산 수행하게 됨.
- RB-DELETE Time complexity
  - $O(\lg n)$

실습 주제(5)

- 문제 풀이
  - 알고리즘 교재 13.4-3

13.4-3

In Exercise 13.3-2, you found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.

HW7.C-Special

- Red-Black Tree 구현
  - 구현 항목
    - Insert, Delete
    - insert/delete 구동에 필요한 제반 함수들
    - 정상 동작 여부 test 함수
  - Due Date
    - 2014년 12월 22일
  - 제출 방법
    - Café에 비밀 글로 제출
  - 점수 비중
    - 100점(중간고사 비중의 40%)
  - 채점 기준
    - 정상 동작 여부
    - 코드 가독성
    - Case 분석을 잘 진행했음을 보여주는 주석