Lecture #11: Red Black Tree

School of Computer Science and Engineering
Kyungpook National University (KNU)

Woo-Jeoung Nam

Search trees

- 이진 검색 트리에서 search, insert, delete 등 동적연산 집합은 O(h) 수행시간이 필요
- 높이가 작은 경우 빠르게 수행 가능

- 높이가 크면 링크드 리스트와 별반 다르게 없다
  - 트리가 균형을 이루도록 구현을 하면 안정적이지 않을까?

Red-Black Tree

- 이진 검색트리(Binary Search Tree)의 일종으로, 균형 잡힌 트리(Balanced Tree)의 일종
- 이진 검색트리의 기본적인 특성을 유지하면서, 트리의 높이를 최소화하여 검색, 삽입, 삭제 등의 연산을 효율적으로 수행 가능

- 레드블랙 트리 특징
  - 각 노드는 레드 또는 블랙 색상을 가짐
  - 루트 노드와 모든 리프 노드는 블랙 색상을 가짐
  - 어떤 노드의 색상이 레드이면, 그 노드의 자식 노드들은 모두 블랙 색상을 가짐
  - 어떤 노드로부터 모든 리프 노드까지의 경로상에 있는 블랙 노드의 수는 모두 같음
- 이러한 규칙들이 만족되도록 레드-블랙 트리는 노드들을 회전하거나 색깔을 변경함으로써 균형을 조정

- 자바의 TreeMap과 TreeSet, C++의 STL set과 map, 리눅스 커널의 프로세스 스케줄러 등에서 레드-블랙 트리가 활용

Red-Black Tree

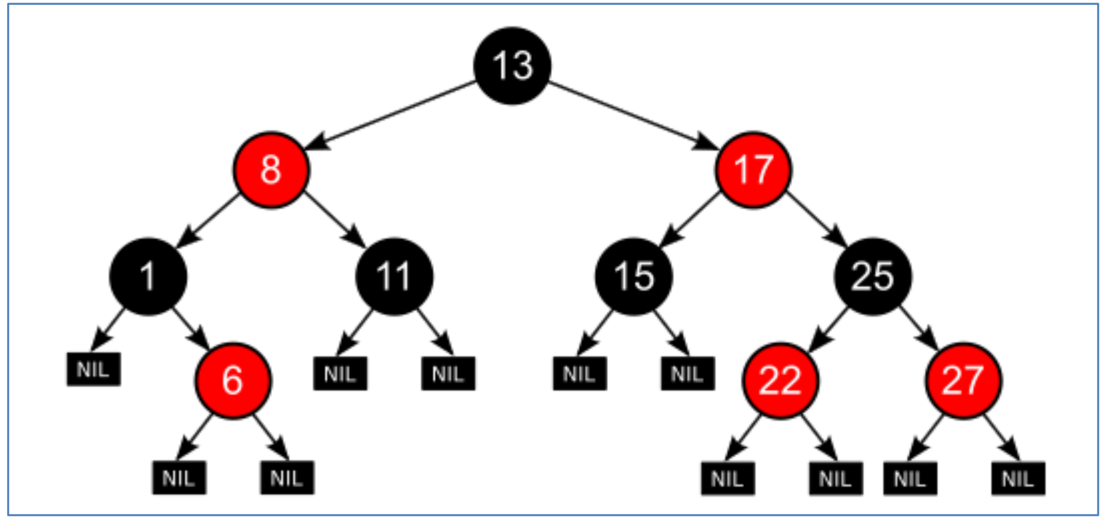
- 이진 검색트리(Binary Search Tree)의 일종으로, 균형 잡힌 트리(Balanced Tree)의 일종
- 이진 검색트리의 기본적인 특성을 유지하면서, 트리의 높이를 최소화하여 검색, 삽입, 삭제 등의 연산을 효율적으로 수행 가능

- 레드블랙 트리 특징
  - 각 노드는 레드 또는 블랙 색상을 가짐
  - 루트 노드와 모든 리프 노드는 블랙 색상을 가짐
  - 어떤 노드의 색상이 레드이면, 그 노드의 자식 노드들은 모두 블랙 색상을 가짐
  - 어떤 노드로부터 모든 리프 노드까지의 경로상에 있는 블랙 노드의 수는 모두 같음
- 이러한 규칙들이 만족되도록 레드-블랙 트리는 노드들을 회전하거나 색깔을 변경함으로써 균형을 조정

- 자바의 TreeMap과 TreeSet, C++의 STL set과 map, 리눅스 커널의 프로세스 스케줄러 등에서 레드-블랙 트리가 활용

Red-Black Tree

- 자가 균형 이진 탐색 트리



1. 모든 노드는 빨간색 혹은 검은색이다.

2. 루트 노드는 검은색이다.

3. 모든 리프 노드(NIL)들은 검은색이다. (NIL : null leaf, 자료를 갖지 않고 트리의 끝을 나타내는 노드)

4. 빨간색 노드의 자식은 검은색이다.

- ➢ **No Double Red(빨간색 노드가 연속으로 나올 수 없다)**

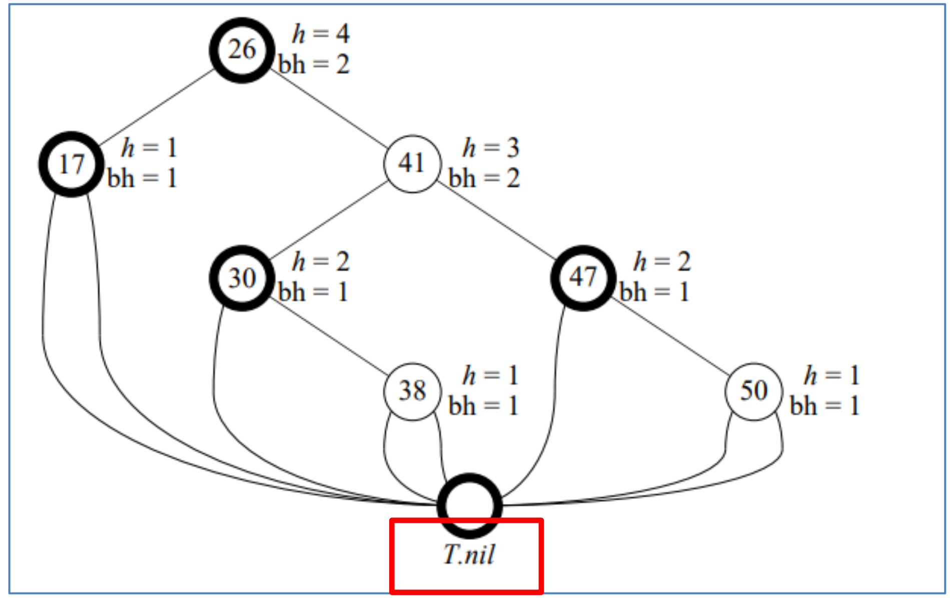
5. 모든 리프 노드에서 Black Depth는 같다.

- ➢ 리프노드에서 루트 노드까지 가는 경로에서 만나는 검은색 노드의 개수가 같다.

Red-Black Tree

- 자가 균형 이진 탐색 트리

- **Nodes with bold outline indicate black nodes**
- **Bh(black height): 한 노드 x에서 리프까지의 경로에 있는 모든 흑색 노드(x자신 제외)의 개수**



1. 모든 노드는 빨간색 혹은 검은색이다.

2. 루트 노드는 검은색이다.

3. 모든 리프 노드(NIL)들은 검은색이다. (NIL : null leaf, 자료를 갖지 않고 트리의 끝을 나타내는 노드)

4. 빨간색 노드의 자식은 검은색이다.

- **No Double Red(빨간색 노드가 연속으로 나올 수 없다)**

5. 모든 리프 노드에서 Black Depth는 같다.

- **리프노드에서 루트 노드까지 가는 경로에서 만나는 검은색 노드의 개수가 같다.**

Red-Black Tree

- H의 높이를 갖는 모든 노드는 흑색높이(black-height)가 h/2이상이다
- 특성 4. 빨간색 노드의 자식은 검은색이다.
  - No Double Red(빨간색 노드가 연속으로 나올 수 없다)
  - h/2 이하의 노드는 빨간색일 수밖에 없다

- N개의 내부 노드를 가지는 레드블랙 트리는 최대 2lg(n+1)이다.

Red-Black Tree

- N개의 내부 노드를 가지는 레드블랙 트리는 최대 2lg(n+1)이다.

- 수학적 귀납법을 통한 증명
- 먼저 루트가 노드 x인 서브트리는 적어도 $2^{bh(x)} - 1$개의 내부노드를 가짐을 증명
- X=0이면 x는 리프노드, bh(x) = 0, 2^0-1 = 0
- 노드 x가 양의 높이를 가지고 두 자식을 갖는 내부노드라고 가정

- 자식이 적색또는 흑색이냐에 따라 bh(x), bh(x)-1의 높이를 가짐
- 따라서 개수는 $2^{bh(x)} - 1$개의 내부노드를 가짐
- 양쪽 자식을 포함하는 서브트리는 적어도 $2^{bh(x)} - 1$+$2^{bh(x)} - 1$+1
- =$2^{bh(x)} - 1$개
- 앞전에 높이는 적어도 h/2
- $\boldsymbol{n} \geq 2^{\frac{h}{2}} - 1$
- 즉 $h \leq 2\lg(n+1)$

Red-Black Tree

Proof By induction on height of x.

Basis: Height of $x = 0 \Rightarrow x$ is a leaf $\Rightarrow \mathrm{bh}(x) = 0$. The subtree rooted at x has 0 internal nodes. $2^0 - 1 = 0$.

Inductive step: Let the height of x be h and $\mathrm{bh}(x) = b$. Any child of x has height $h - 1$ and black-height either b (if the child is red) or $b - 1$ (if the child is black). By the inductive hypothesis, each child has $\geq 2^{\mathrm{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains $\geq 2 \cdot (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ internal nodes. (The $+1$ is for x itself.) ■ (claim)

Lemma

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

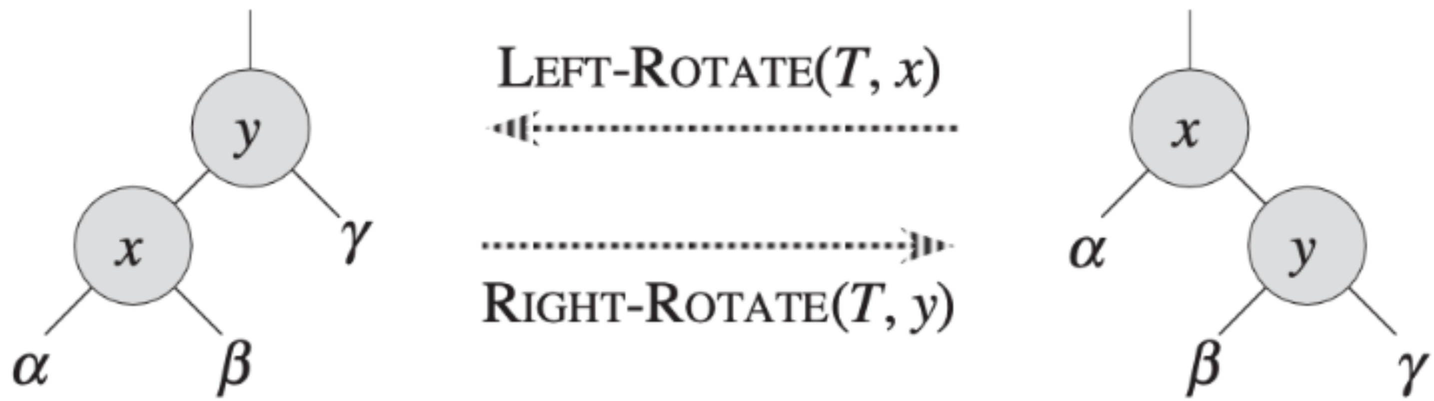
Proof Let h and b be the height and black-height of the root, respectively. By the above two claims,

$$n \geq 2^b - 1 \geq 2^{h/2} - 1 .$$

Adding 1 to both sides and then taking logs gives $\lg(n + 1) \geq h/2$, which implies that $h \leq 2 \lg(n + 1)$. ■ (theorem)

Red-Black Tree

- **Rotation**
  - 레드-블랙 트리의 삽입(insert), delete(삭제) 연산 과정에서 트리가 수정되기 때문에 레드-블랙 트리의 특성을 위반할 수 있다
  - 특성을 복구해주기 위해서 트리내의 일부 노드들의 색깔과 포인터를 변경해야 되고 이때 회전을 사용한다
    - 좌회전 전 (in-order traversal (중위 순회) - 오름차순)
    - α - x - β - y - γ
    - 좌회전 후 (in-order traversal (중위 순회) - 오름차순)
    - α - x - β - y - γ



Red-Black Tree

- **Rotation**
  - 가정: x의 오른쪽 자식이 리프노드가 아님(x.right != T.NIL)

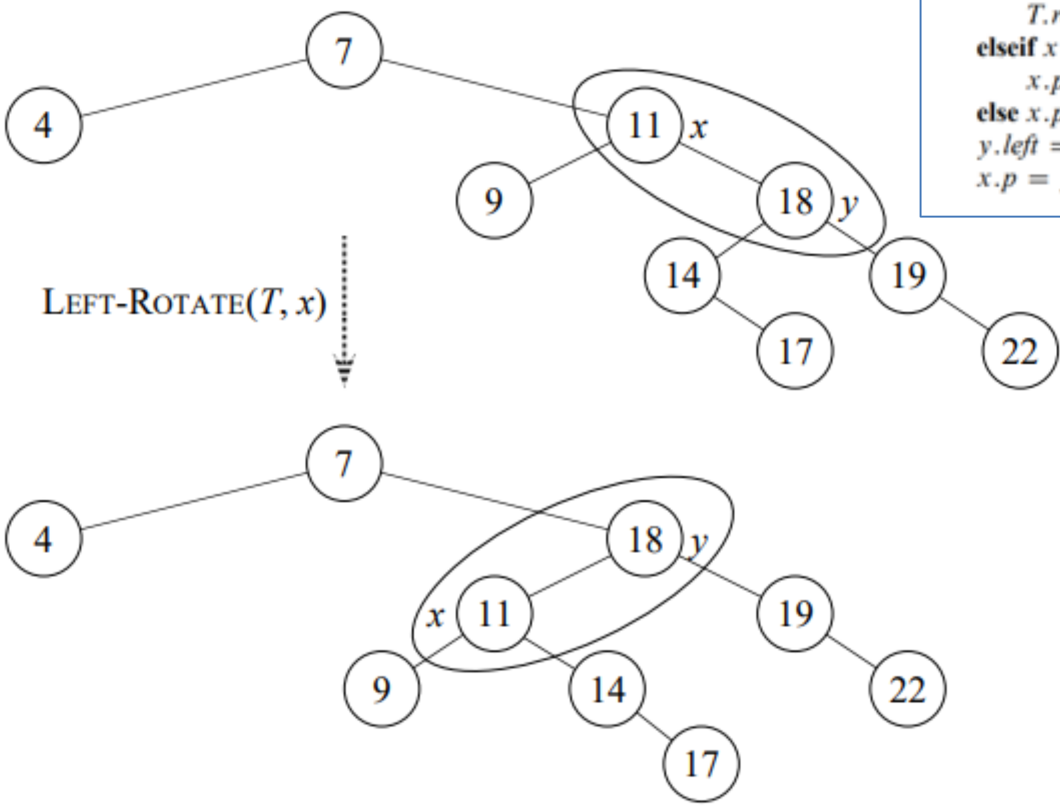
LEFT-ROTATE(T, x)

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

Red-Black Tree

- 예제로 이해

```
LEFT-ROTATE(T, x)
  y = x.right              // set y
  x.right = y.left         // turn y's left subtree into x's right subtree
  if y.left ≠ T.nil
      y.left.p = x
  y.p = x.p                // link x's parent to y
  if x.p == T.nil
      T.root = y
  elseif x == x.p.left
      x.p.left = y
  else x.p.right = y
  y.left = x               // put x on y's left
  x.p = y
```



Red-Black Tree - Insertion

- 레드-블랙 트리에서 새로운 노드를 삽입할 때, 새로운 노드는 항상 적색으로 입력한다.

- 만약 트리가 레드블랙 트리의 특성을 위반하면, 두가지 operation 을 통해 트리의 구조를 조정한다.
  - Recolor, Rotation

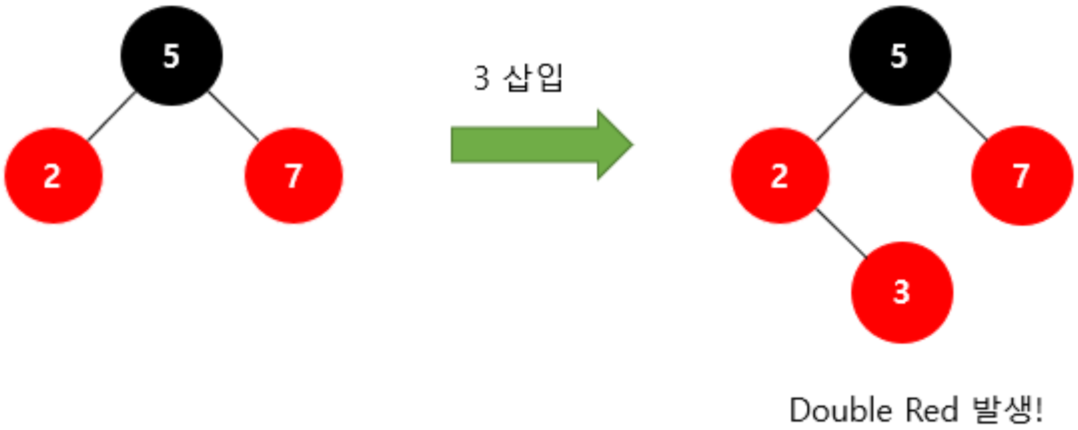
- 쉬운 예제로 이해해보자

출처: https://code-lab1.tistory.com/62, https://velog.io/@stthunderl/Red-Black-Tree-4-%EC%82%BD%EC%9E%85insert

Red-Black Tree - Insertion

- 레드-블랙 트리에 새로운 노드를 삽입할 때 새로운 노드는 항상 빨간색으로 삽입한다

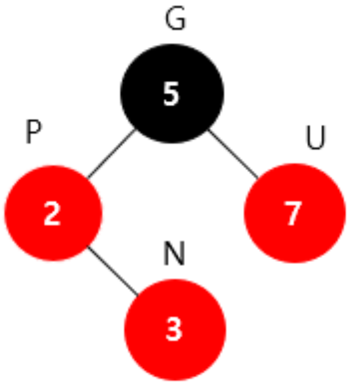
- 빨간 노드가 다음처럼 두번 연속 나타날 수 있다.
  - 특징 4 를 위배



Red-Black Tree - Insertion

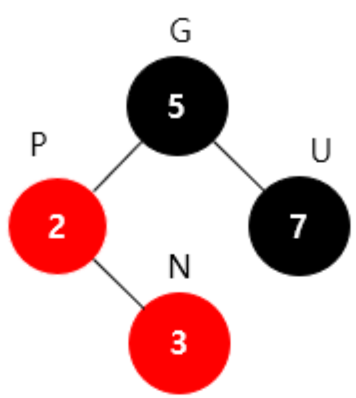
- 레드-블랙 트리에 새로운 노드를 삽입할 때 새로운 노드는 항상 빨간색으로 삽입한다

- 빘간 노드가 다음처럼 두번 연속 나타날 수 있다.
  - 특징 4 를 위배
- Notation을 정하자
  - 새로 삽입할 노드를 N(New), 부모 노드를 P(Parent), 조상 노드를 G(Grand Parent), 삼촌 노드를 U(Uncle)
  - 할아버지, 아버지, 삼촌 관계

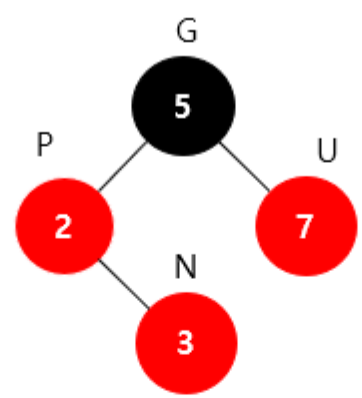


Red-Black Tree - Insertion

- **Notation을 정하자**
  - 새로 삽입할 노드를 N(New), 부모 노드를 P(Parent), 조상 노드를 G(Grand Parent), 삼촌 노드를 U(Uncle)
  - 할아버지, 아버지, 삼촌 관계
  - 삼촌노드가 검은색이면? Restructuring (Rotation)
  - 삼촌노드가 빨간색이면? Recoloring



U가 검은색 -> Restructuring

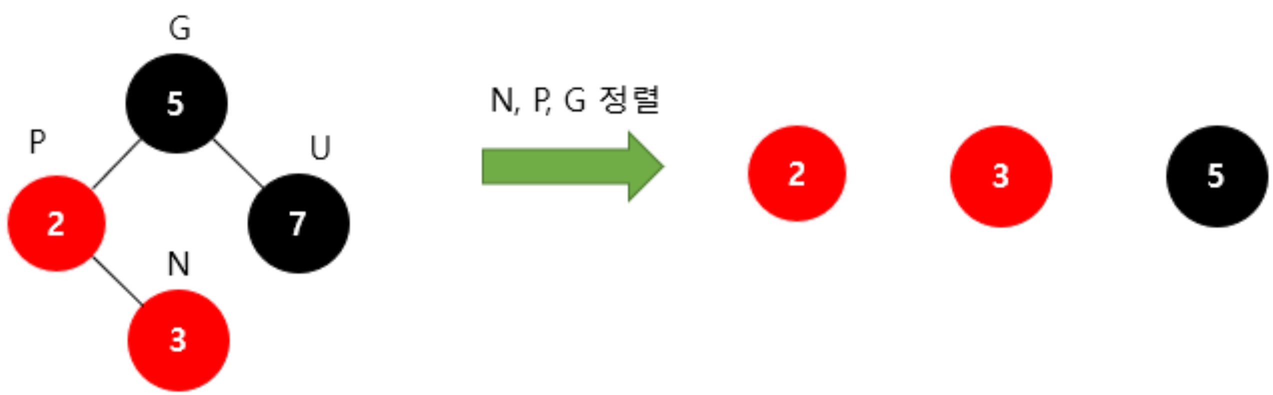


U가 빨간색 -> Recoloring

Red-Black Tree - Insertion

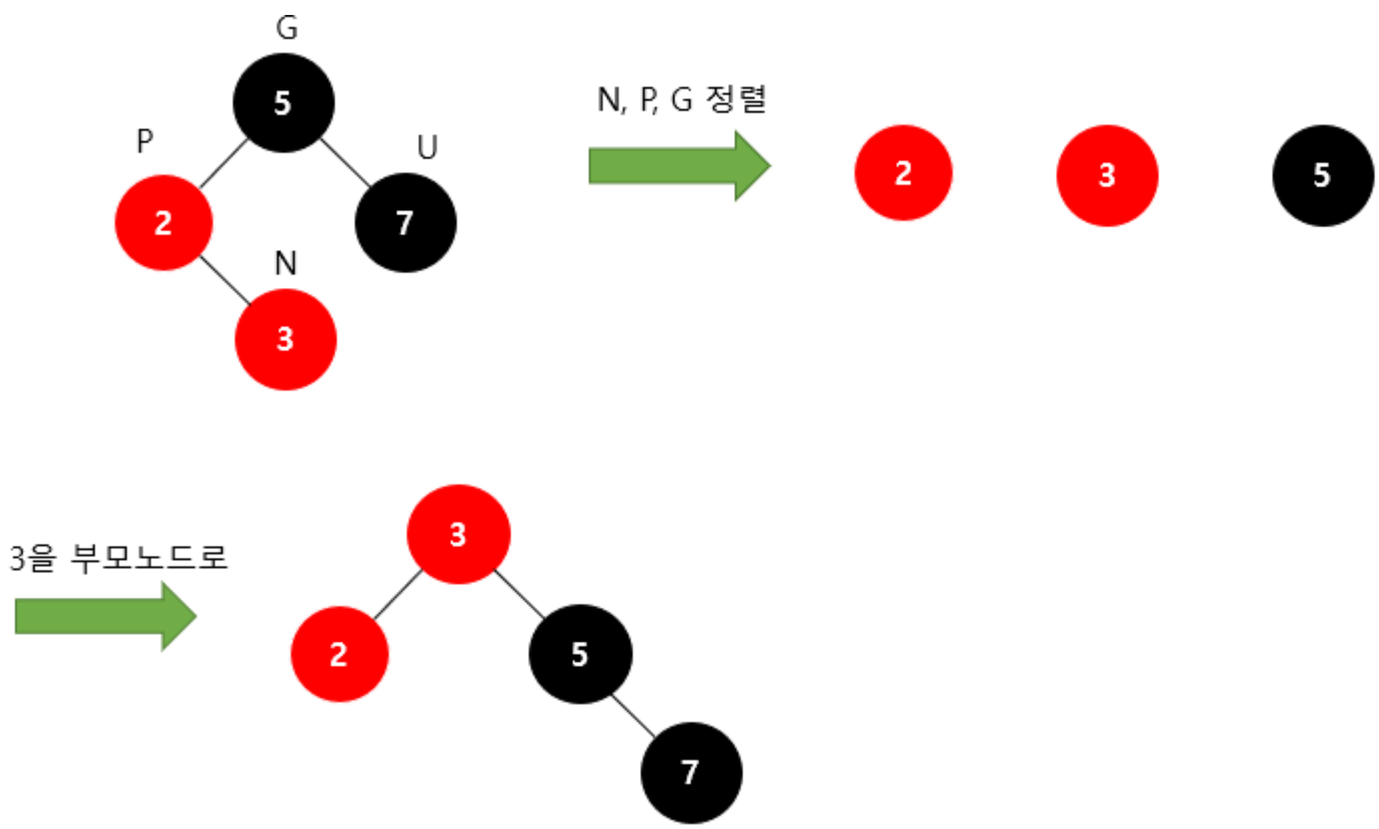
- **Restructing**
  - 1. 새로운 노드(N), 부모 노드(P), 조상 노드(G)를 오름차순으로 정렬한다.
  - 2. 셋 중 중간값을 부모로 만들고 나머지 둘을 자식으로 만든다.
  - 3. 새로 부모가 된 노드를 검은색으로 만들고 나머지 자식들을 빨간색으로 만든다

- **빨간색 중복인데 삼촌이 검은색 -> restructuring**



Red-Black Tree - Insertion

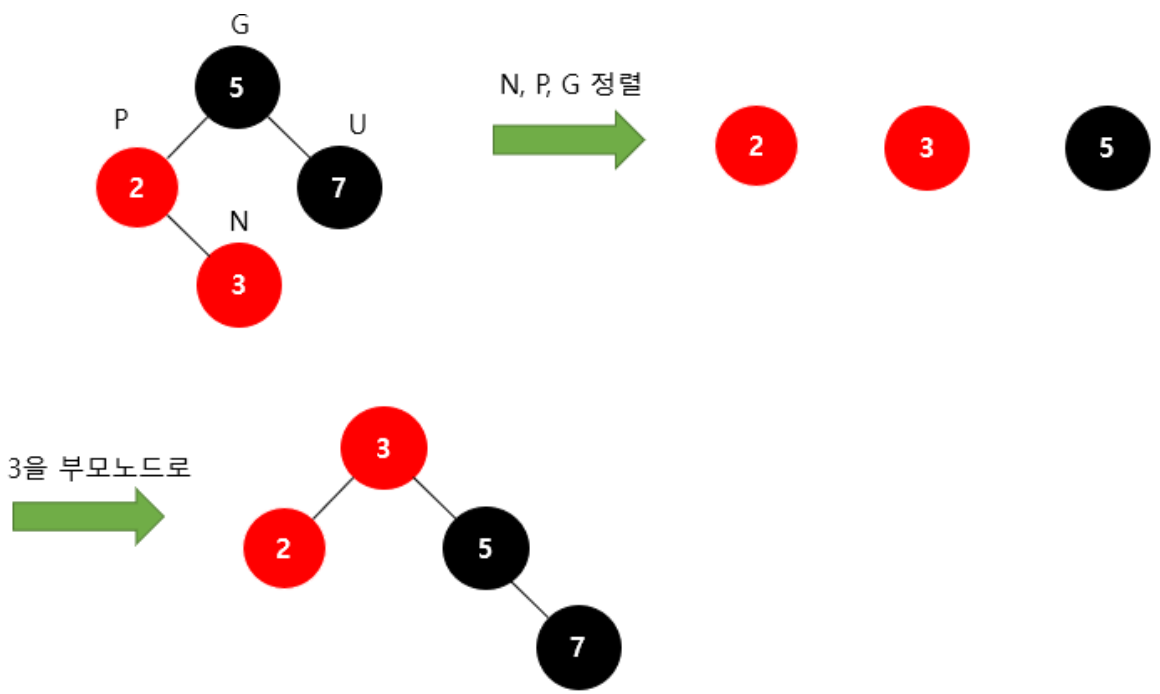
- **Restructing**
  - 1. 새로운 노드(N), 부모 노드(P), 조상 노드(G)를 오름차순으로 정렬한다.
  - 2. 셋 중 중간값을 부모로 만들고 나머지 둘을 자식으로 만든다.
  - 3. 새로 부모가 된 노드를 검은색으로 만들고 나머지 자식들을 빨간색으로 만든다



Red-Black Tree - Insertion

- **Restructing**
  - 1. 새로운 노드(N), 부모 노드(P), 조상 노드(G)를 오름차순으로 정렬한다.
  - 2. 셋 중 중간값을 부모로 만들고 나머지 둘을 자식으로 만든다.
  - 3. 새로 부모가 된 노드를 검은색으로 만들고 나머지 자식들을 빨간색으로 만든다

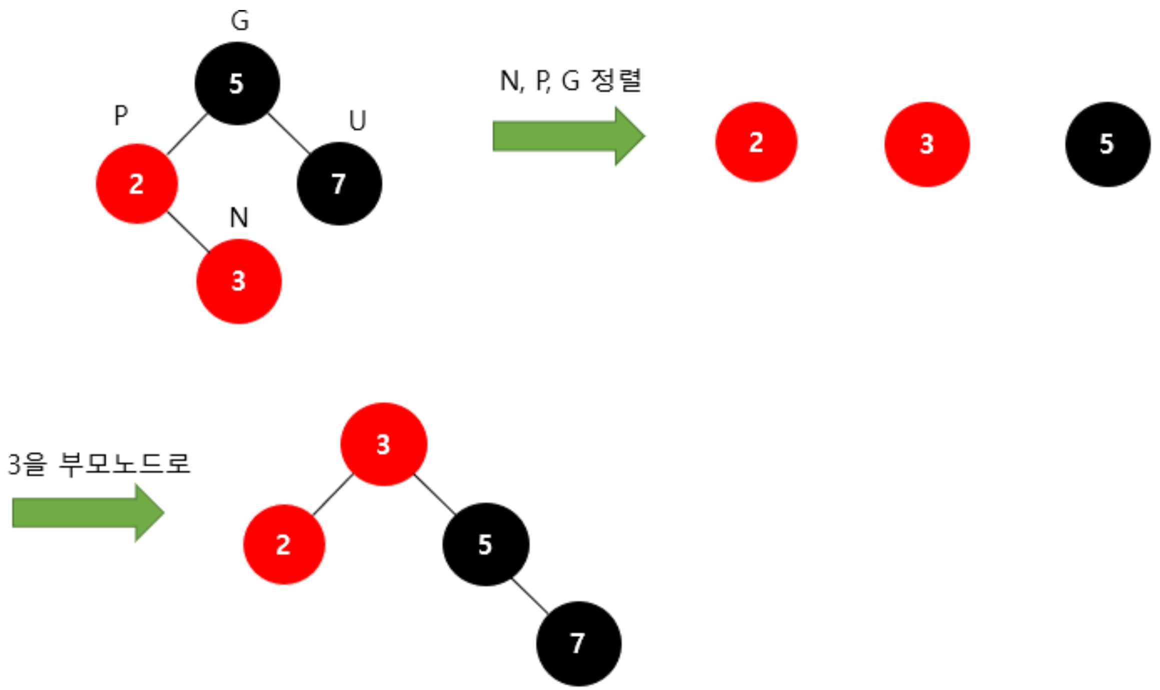
- 중간값인 3을 부모노드로 만듬



Red-Black Tree - Insertion

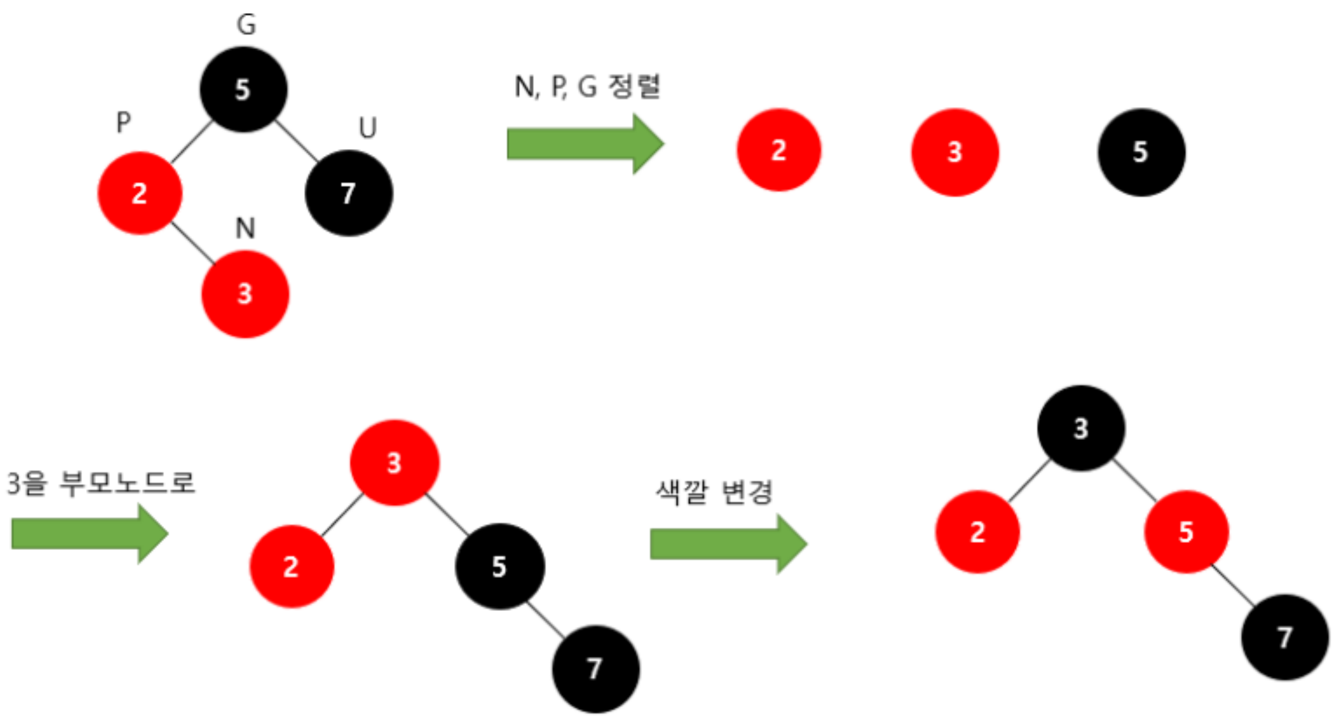
- **Restructing**
  - 1. 새로운 노드(N), 부모 노드(P), 조상 노드(G)를 오름차순으로 정렬한다.
  - 2. 셋 중 중간값을 부모로 만들고 나머지 둘을 자식으로 만든다.
  - 3. 새로 부모가 된 노드를 검은색으로 만들고 나머지 자식들을 빨간색으로 만든다

- 중간값인 3을 부모노드로 만듬, 2와 5를 자식노드로 바꾼다



Red-Black Tree - Insertion

- **Restructing**
  - 1. 새로운 노드(N), 부모 노드(P), 조상 노드(G)를 오름차순으로 정렬한다.
  - 2. 셋 중 중간값을 부모로 만들고 나머지 둘을 자식으로 만든다.
  - 3. 새로 부모가 된 노드를 검은색으로 만들고 나머지 자식들을 빨간색으로 만든다
- 새롭게 부모가 된 3을 검은색으로 바꾸고 나머지 자식 2,5를 빨간색으로 바꾼다
- Double Red 문제 해결(검정색 leaf노드는 그림상 생략됨)



Red-Black Tree - Insertion

- **Recoloring**

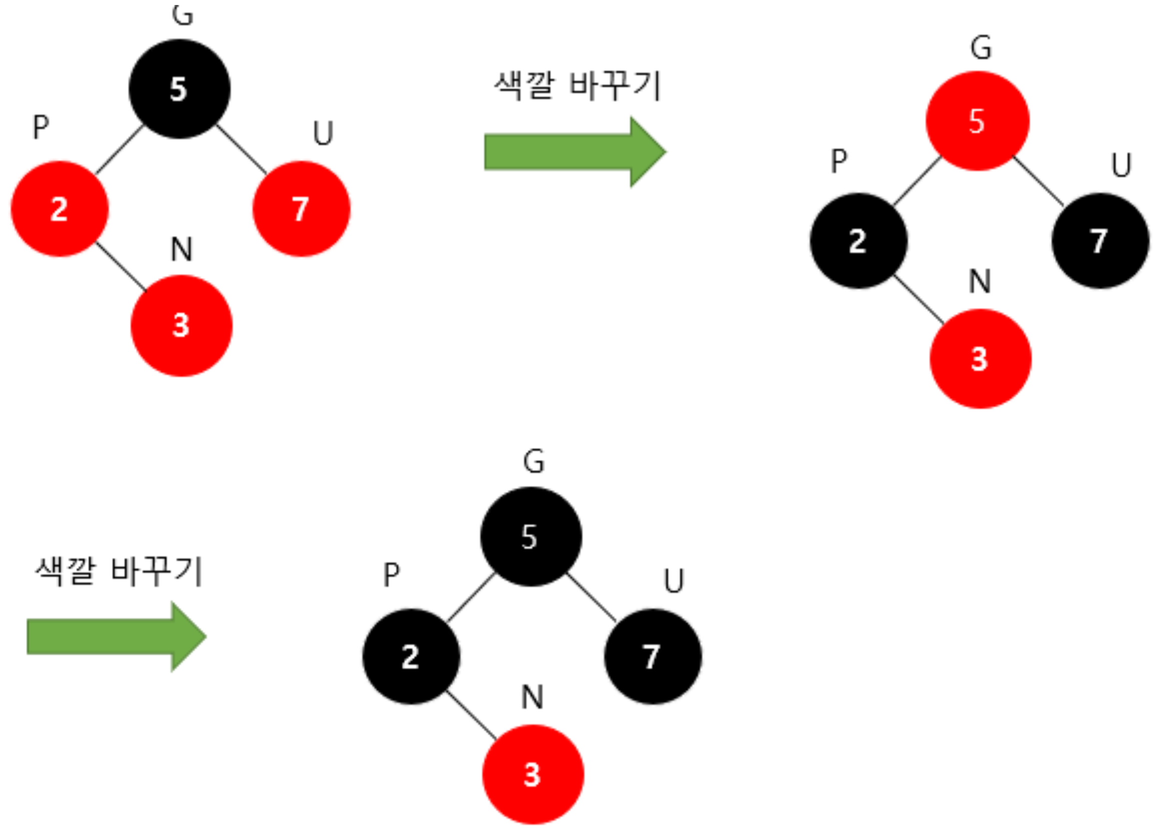
  1. 새로운 노드(N)의 부모(P)와 삼촌(U)을 검은색으로 바꾸고 조상(G)을 빨간색으로 바꾼다.

     1-1. 조상(G)이 루트 노드라면 검은색으로 바꾼다.

     1-2. 조상(G)을 빨간색으로 바꿨을 때 또다시 Double Red가 발생한다면 또다시 Restructuring 혹은 Recoloring을 진행해서 Double Red 문제가 발생하지 않을 때까지 반복한다.

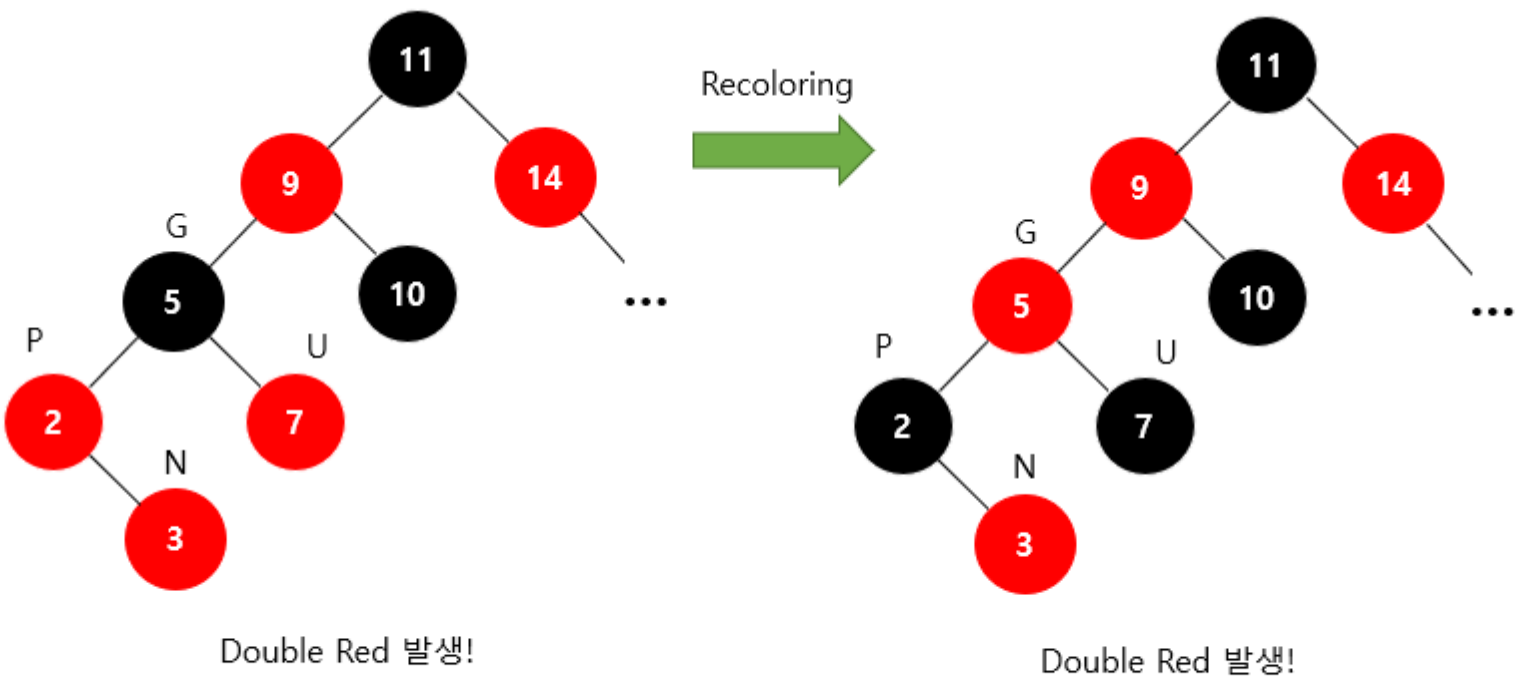
Red-Black Tree - Insertion

- **Recoloring**
  - 빨간색 두번나왔는데 삼촌이 빨간색이다->recoloring
  - 부모 p와 삼촌 u를 검은색으로 바꿈
  - 그리고 조상 G를 빨간색으로 바꾼다
  - 만약 조상 G가 루트노드라면? -> 검은색으로 바꾼다
  - 해결?



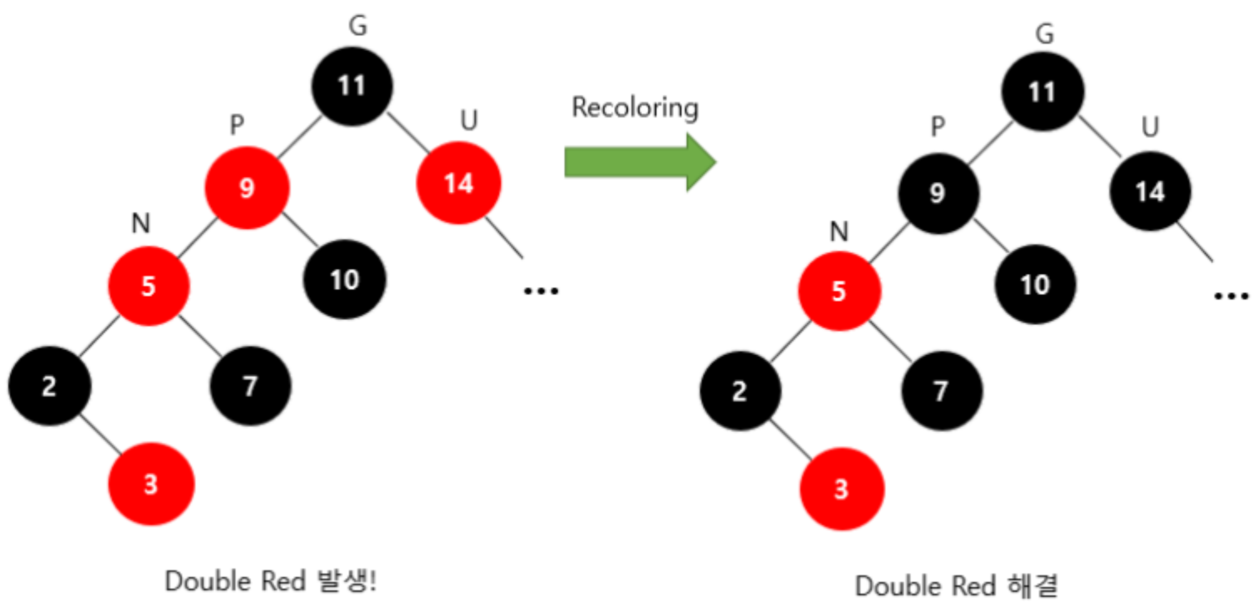
Red-Black Tree - Insertion

- **Recoloring**
  - 만약 조상 G가 루트노드가 아니라면?
  - 그 위의 조상노드의 조상이 빨간색일 수 있다 -> double red 발생



Red-Black Tree - Insertion

- **Recoloring**
  - 만약 조상 G가 루트노드가 아니라면?
  - 그 위의 조상노드의 조상이 빨간색일 수 있다 -> double red 발생
  - 해결될때까지 위로 타고가면서 진행
  - 기준이 다시 조상노드였던 5가 N이 되고 그 기준으로 진행
  - 만약 기준 노드의 삼촌이 검은색?-> Reconstructuring
  - 빨간색이라면? Recoloring



Red-Black Tree – Insertion (교재)

- **Pseudo code**
  - Tree-Insert와 무엇이 다른가?
  - 트리에서 삽입도 마지막 리프노드에 삽입
    - NIL 대신에 T.nil 로 대체 (검정색)
    - 삽입된 z의 left, right에 T.nil 삽입
    - 삽입된 z의 색은 적색
    - 적색으로 색칠시 위반될 수 있어서 RB-Insert_fixup으로 교정

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

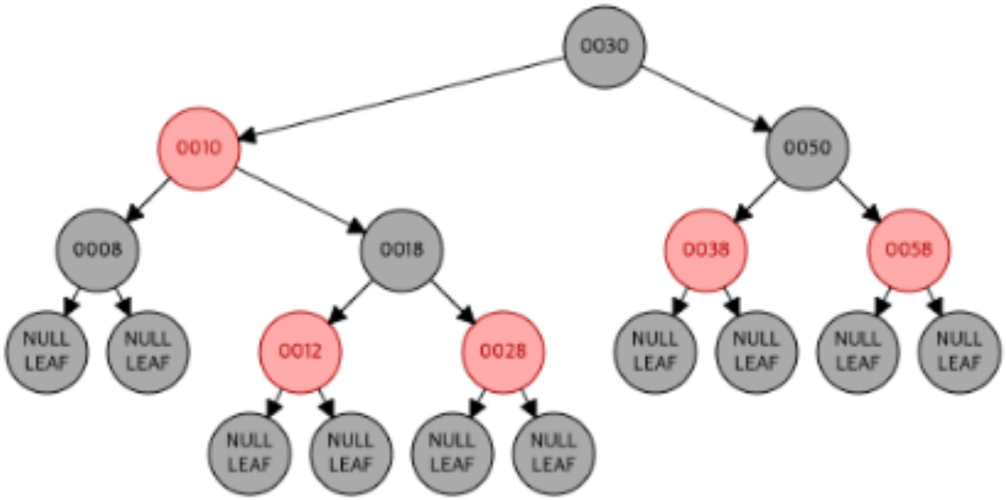
Red-Black Tree – Insertion (교재)

- **Pseudo code**
  - Case1: z의 삼촌 y가 적색인 경우 -> Recoloring
  - Case2: z의 삼촌 y가 흑색이며 z가 오른쪽 자식인 경우 -> Restructuring (left rotate)
  - Case3: z의 삼촌 y가 흑색이며 z가 왼쪽 자식인 경우 -> Restructuring (right rotate)

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

Red-Black Tree – Insertion (교재)

- 예제) 키값이 20인 노드를 다음 트리에 삽입
- Target Node를 20으로 설정

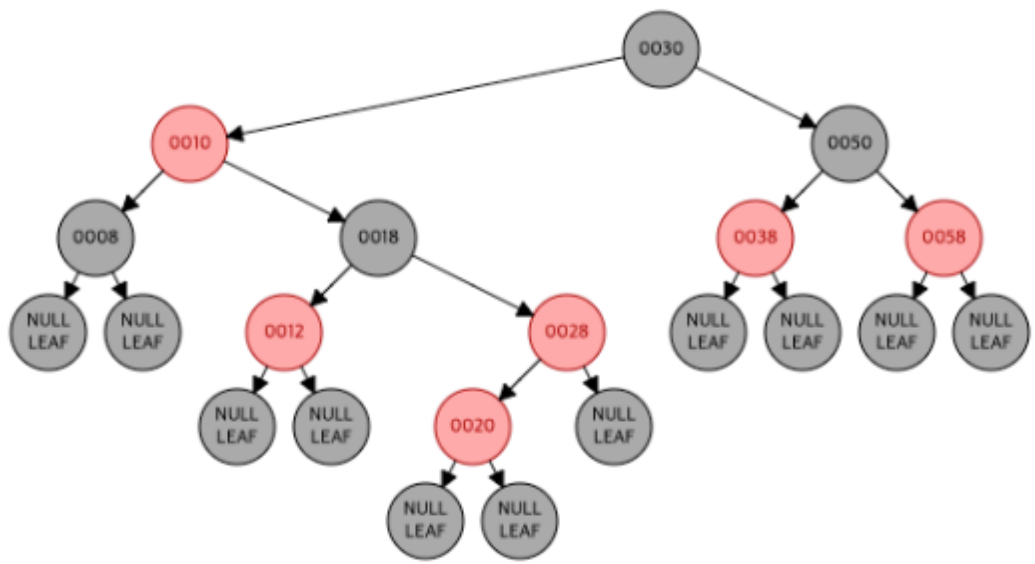


Red-Black Tree – Insertion (교재)

- 예제) 키값이 20인 노드를 다음 트리에 삽입
- Target Node를 20으로 설정, 들어갈 위치 탐색, Double Red!

  - key = 20 < 28 이고, 28의 왼쪽 노드가 nil.

Found null tree (or phantom leaf), inserting element



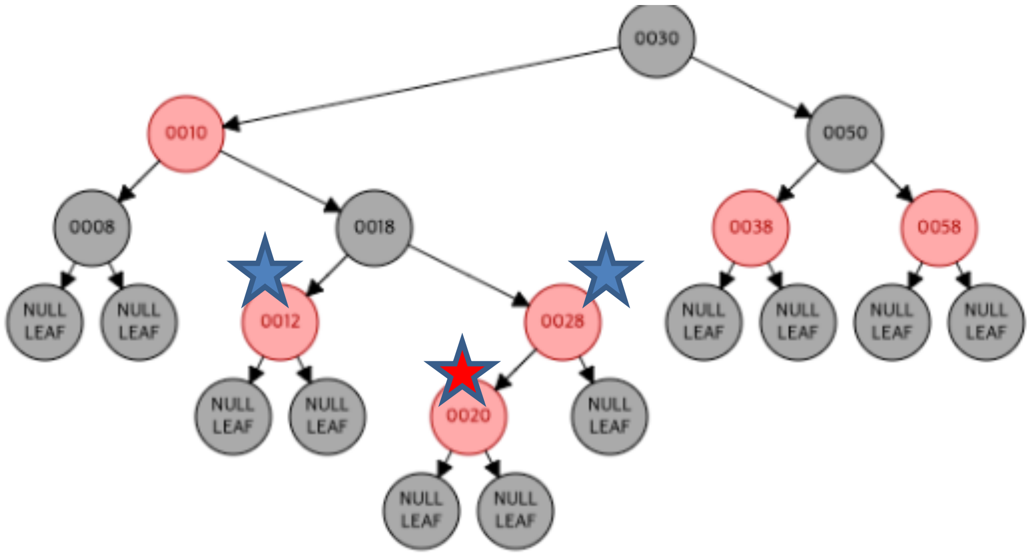
Red-Black Tree – Insertion (교재)

- **Insert-fixup**
  - while문 조건 : target의 부모가 적색.

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

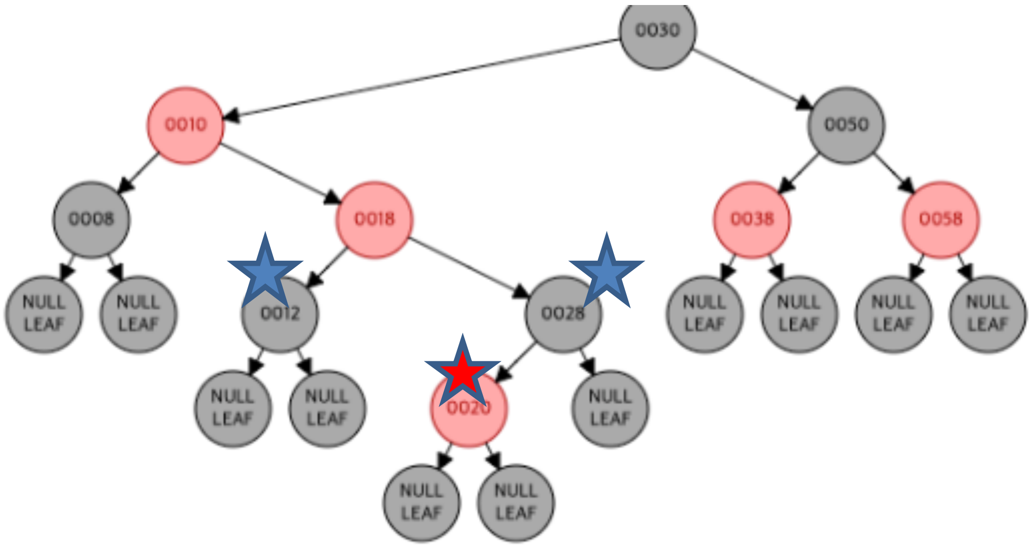
Red-Black Tree – Insertion (교재)

- **Insert-fixup**
  - case 1:부모 노드의 컬러가 적색 & 삼촌 노드의 컬러가 적색 ->recoloring



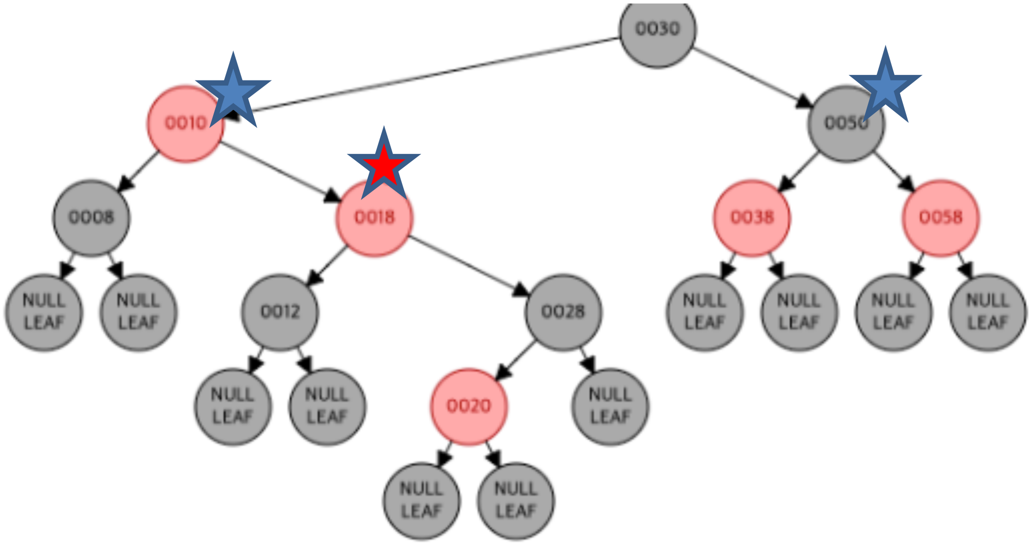
Red-Black Tree – Insertion (교재)

- **Insert-fixup**
  - case 1:부모 노드의 컬러가 적색 & 삼촌 노드의 컬러가 적색 ->recoloring
  - target node를 20의 할아버지 노드(18)로 설정



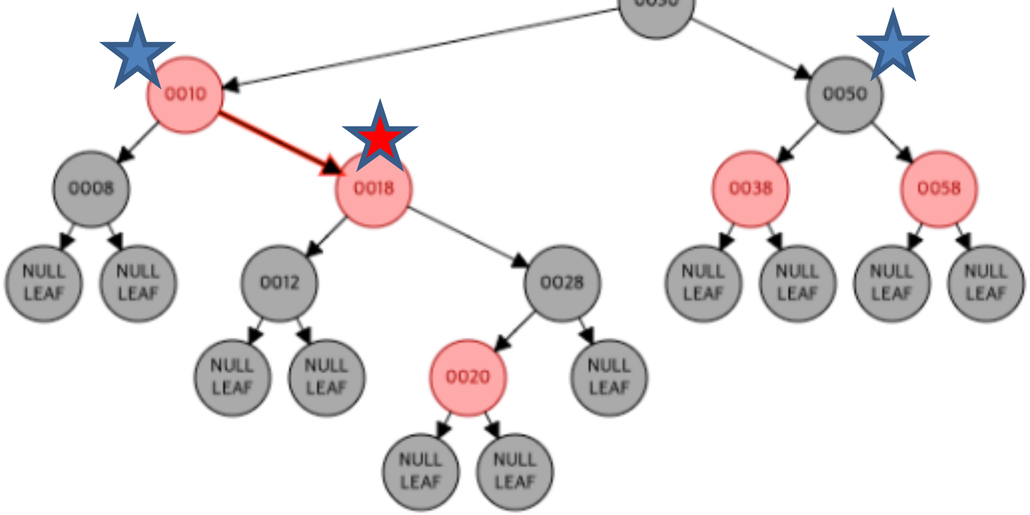
Red-Black Tree – Insertion (교재)

- **Insert-fixup**
  - 부모 노드의 컬러(10)가 적색 & 삼촌 노드의 컬러(50)가 흑색
  - 적색 다음에 또 적색 -> 특성 위반



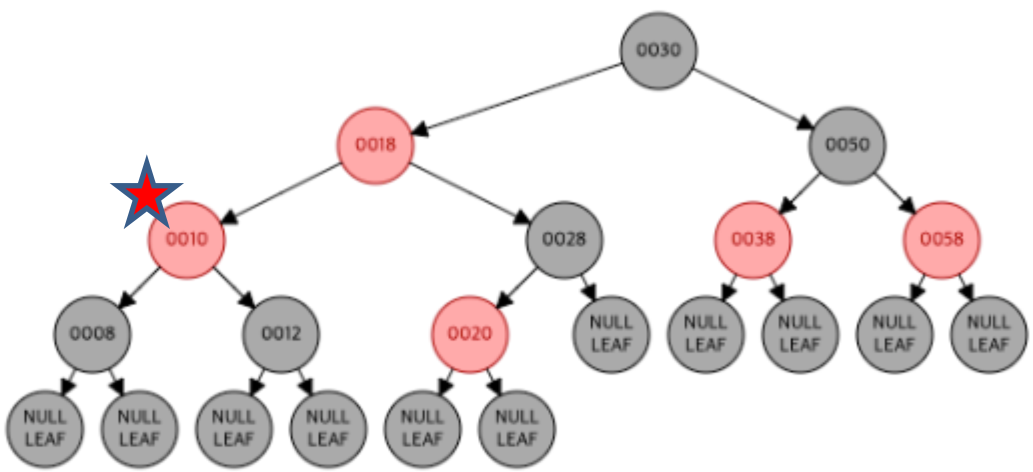
Red-Black Tree – Insertion (교재)

- **Insert-fixup**
  - Rotation- 부모노드가 왼쪽 자식이면 left rotation, 오른쪽 자식이면 right rotation
  - Target 노드를 부모노드로 설정후 left roatation (10과 18의 위치 교환)
  - 18->10



Red-Black Tree – Insertion (교재)

- **Insert-fixup**
  - 조정 후에 target node = 10, 부모노드와 여전히 적색적색 관계



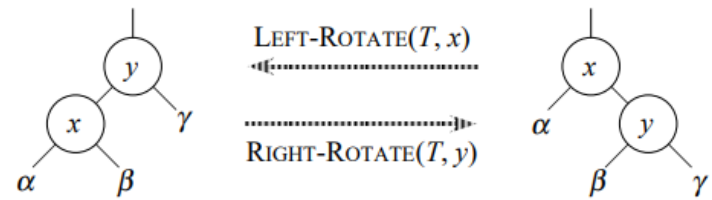
Red-Black Tree – Insertion (교재)

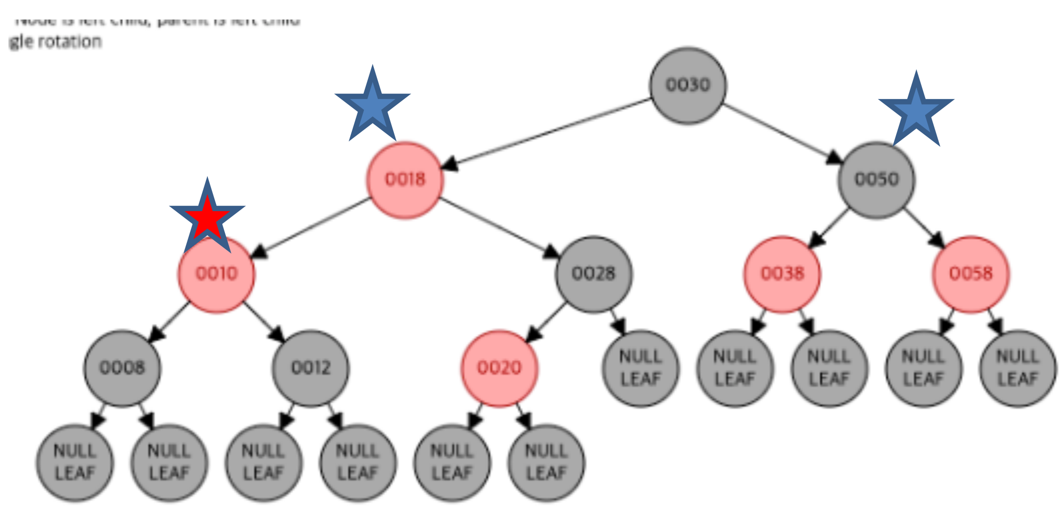
- Insert-fixup
  - 1. Fix-up 과정은 부모노드가 적색일 때 일어난다.
  - 2. case 1 은 부모노드와 삼촌 노드가 모두 적색일 때,
  - 3. case 2 (부모노드는 적색, 삼촌노드는 흑색)는 2가지로 나뉜다.
    - case 2 - (1) :
    - grandparent -> left -> right = target 또는 grapndparent -> right -> left = target.

    - case 2 - (2) :
    - grandparent -> left -> left = target 또는 grandparent -> right -> right – target

Red-Black Tree – Insertion (교재)

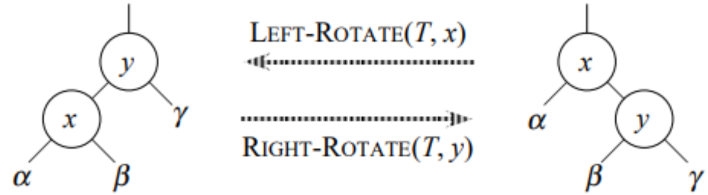
- **Case 2-2, target node = 10**
- **부모노드 적색, 삼촌노드 흑색**
- **Right-rotation 진행**

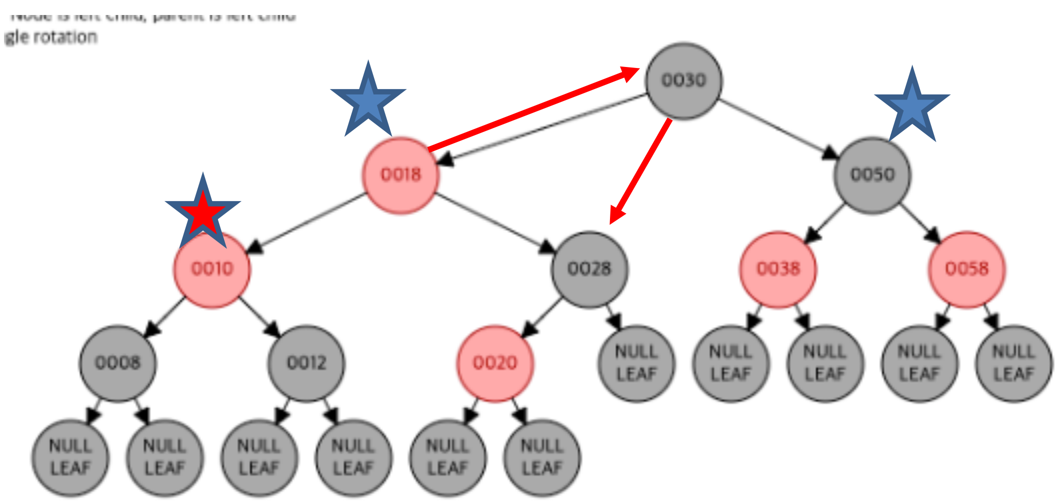




Red-Black Tree – Insertion (교재)

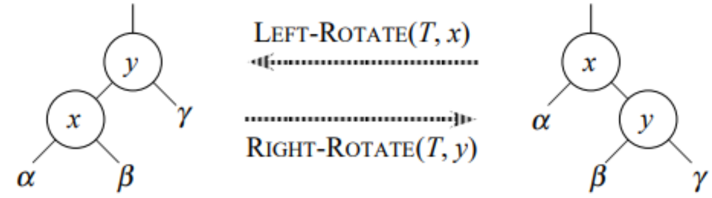
- Case 2-2, target node = 10
- 부모노드 적색, 삼촌노드 흑색
- Right-rotation 진행

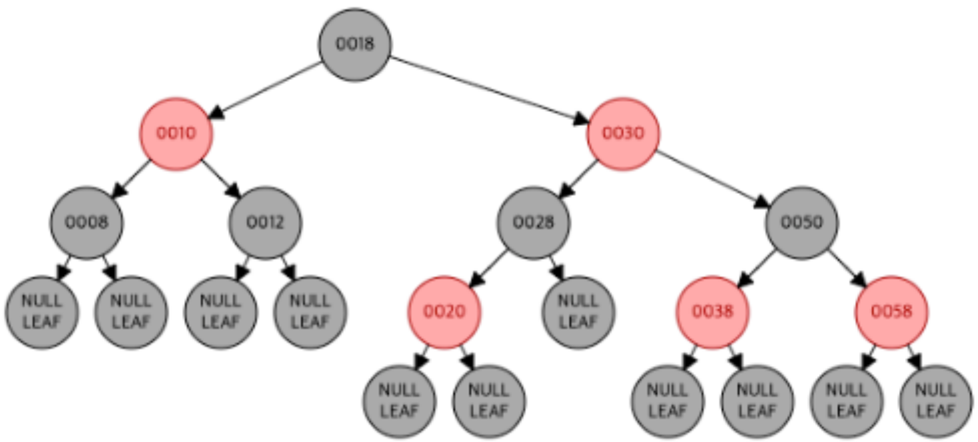




Red-Black Tree – Insertion (교재)

- Case 2-2, target node = 10
- 부모노드 적색, 삼촌노드 흑색
- Right-rotation 진행 후 recoloring
- 루트노드 흑색, 30을 흑색으로 바꾼다



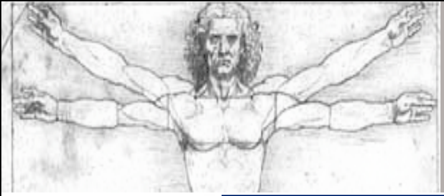


Red-Black Tree – Delete

- 이진 검색트리의 특성을 만족하면서 노드를 삭제
- 이후 delete-fixup과정에서 RB트리의 특성을 만족시키기 위해 트리 구조 수정

- 이진 검색트리에서의 Tree-delete를 기반으로 함
- Transplant함수를 RB에 맞게 수정

- 시간복잡도는 O(log n) 을 만족한다



Red-Black Tree – Delete

- 삭제의 과정
- 1. 노드 탐색: 삭제할 노드 x를 찾아야 합니다.
  - 이진 탐색 트리에서 삭제하려는 노드를 찾는 것과 동일한 방법으로 이루어집니다.
- 2. 노드 제거: 삭제할 노드 x를 발견하면, 3가지 경우를 고려해야 합니다.
  - 경우 1: 노드 x가 자식 노드를 가지지 않는 경우
    - 노드 x를 직접 삭제합니다.
  - 경우 2: 노드 x가 하나의 자식 노드를 가지는 경우
    - 노드 x를 삭제하고, 자식 노드를 x의 부모 노드와 연결합니다.
  - 경우 3: 노드 x가 두 개의 자식 노드를 가지는 경우
    - 노드 x의 후속자 노드(오른쪽 하위 트리에서 가장 작은 값)를 찾아 x의 위치로 이동시키고, 후속자 노드를 삭제합니다.
- 3. 레드-블랙 트리 속성 복원: 삭제 과정에서 레드-블랙 트리의 속성이 위반되었을 수 있습니다. 속성을 복원하기 위해 다음 작업을 수행합니다.
  - 속성 1: 각 노드는 빨간색 또는 검은색입니다. 유지 시키는 것이 핵심이다.
  - 속성 2: 루트 노드는 검은색입니다.
  - 속성 3: 모든 잎 노드는 검은색입니다.
  - 속성 4: 빨간색 노드의 자식 노드는 모두 검은색입니다. (즉, 빨간색 노드는 연속되지 않습니다.)
  - 속성 5: 각 노드에서 null 노드까지의 검은색 노드 수는 모두 같습니다.

Red-Black Tree – Delete

- 속성이 깨지는 경우가 발생하면, 회전 및 색상 변경을 통해 트리를 다시 균형잡힌 상태로 만듭니다.
  - 1. 색상 변경(Color Flip): 특정 노드의 색상을 변경합니다. 부모 노드와 자식 노드의 색상을 바꾸어 트리의 속성을 유지합니다.
  - 2. 회전(Rotation): 트리의 구조를 변경하여 균형을 유지하는 방법입니다.
    - 왼쪽 회전(Left Rotation): 부모 노드를 왼쪽으로 밀어내고, 오른쪽 자식 노드를 부모 노드로 만듭니다.
    - 오른쪽 회전(Right Rotation): 부모 노드를 오른쪽으로 밀어내고, 왼쪽 자식 노드를 부모 노드로 만듭니다.

Red-Black Tree – Delete

- **RB-Transplant**
  - 각각의 노드들은 parent, left, right, key, color 필드를 갖고있다.
  - transplant 과정에서는, target 노드와 그 노드의 parent 노드의 연결관계만을 처리하는 과정
- **target 노드를 대체할 노드를 구했다고 가정하고, target의 parent 입장에서 왼쪽 자식이었는지, 오른쪽 자식이었는지 파악한 후에 그 자식을 대체할 노드로 지정하고 대체할 노드 또한 target의 parent를 부모노드로 지정하여 두 노드를 연결**

v를 u의 sub tree로.

교체

```
TRANSPLANT(T, u, v)
  if u.p == NIL
      T.root = v
  elseif u == u.p.left
      u.p.left = v
  else u.p.right = v
  if v ≠ NIL
      v.p = u.p
```

#루트 처리

#일반적인 bst에서의 transplant.

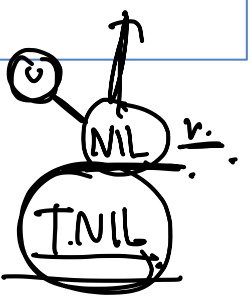
```
RB-TRANSPLANT(T, u, v)
  if u.p == T.nil
      T.root = v
  elseif u == u.p.left
      u.p.left = v
  else u.p.right = v
  v.p = u.p
```

Red-Black Tree – Delete

- **RB-Transplant**
  - 각각의 노드들은 parent, left, right, key, color 필드를 갖고있다.
  - transplant 과정에서는, target 노드와 그 노드의 parent 노드의 연결관계만을 처리하는 과정
- target 노드를 대체할 노드를 구했다고 가정하고, target의 parent 입장에서 왼쪽 자식이었는지, 오른쪽 자식이었는지 파악한 후에 그 자식을 대체할 노드로 지정하고 대체할 노드 또한 target의 parent를 부모노드로 지정하여 두 노드를 연결

RB-TRANSPLANT(T, u, v)
  if $u.p == T.nil$ #루트 일때
    $T.root = v$
  elseif $u == u.p.left$
    $u.p.left = v$
  else $u.p.right = v$
  $v.p = u.p$

if(v≠NIL)
조건 필요 X



- line 1 : if 삭제할 노드(u)의 부모노드가 nil 일 때,
  - 삭제할 노드가 루트노드라면,
- line 2 : 트리의 루트노드를 대체할 노드(v)로 설정.
- line 3 : else if 삭제할 노드(u)가 부모노드의 왼쪽 자식일 때,
- line 4 : 삭제할 노드(u)의 왼쪽 자식을 대체할 노드(v)로 설정.
- line 5 : else (삭제할 노드(u)가 부모노드의 오른쪽 자식일 때)
  - 삭제할 노드(u)의 오른쪽 자식을 대체할 노드(v)로 설정.
- line 6 : 삭제할 노드(u)의 부모노드를 대체할 노드(v)의 부모노드로 설정.

Red-Black Tree – Delete

- **RB-Delete**

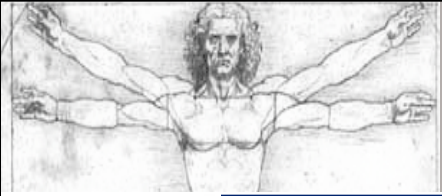
```
TREE-DELETE(T, z)
  if z.left == NIL
      TRANSPLANT(T, z, z.right)          // z has no left child
  elseif z.right == NIL
      TRANSPLANT(T, z, z.left)           // z has just a left child
  else // z has two children.
      y = TREE-MINIMUM(z.right)          // y is z's successor
      if y.p ≠ z
          // y lies within z's right subtree but is not the root of this subtree.
          TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      // Replace z by y.
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```

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

Red-Black Tree – Delete

- **Difference between RB-delete and Tree-delete**

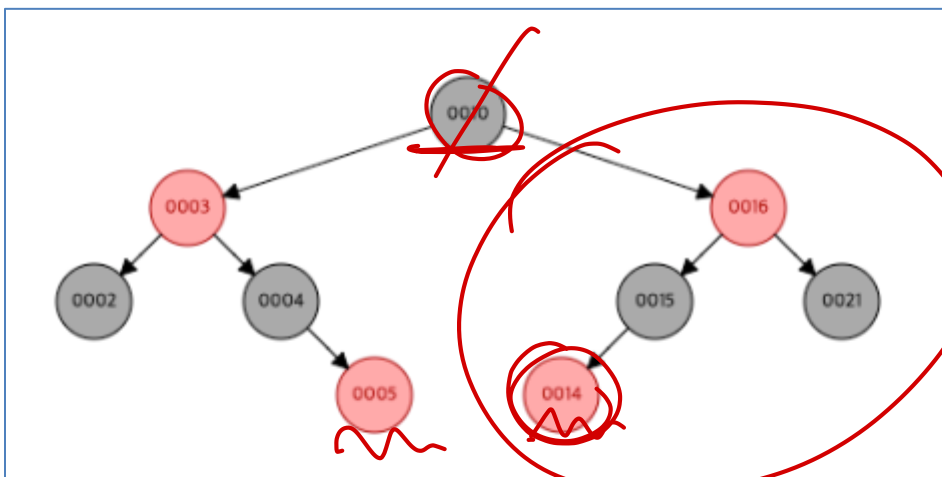
  - y is the node either removed from the tree (when z has fewer than 2 children) or moved within the tree (when z has 2 children).
  - Need to save y's original color (in *y-original-color*) to test it at the end, because if it's black, then removing or moving y could cause red-black properties to be violated.
  - x is the node that moves into y's original position. It's either y's only child, or $T.nil$ if y has no children.
  - Sets $x.p$ to point to the original position of y's parent, even if $x = T.nil$. $x.p$ is set in one of two ways:
    - If z is not y's original parent, $x.p$ is set in the last line of RB-TRANSPLANT.
    - If z is y's original parent, then y will move up to take z's position in the tree. The assignment $x.p = y$ makes $x.p$ point to the original position of y's parent, even if x is $T.nil$.
  - If y's original color was black, the changes to the tree structure might cause red-black properties to be violated, and we call RB-DELETE-FIXUP at the end to resolve the violations.



Red-Black Tree – Delete

- **최소값의 노드 산출(tree_minimum)**
  - 키 값 노드를 삭제할 때, 이진트리 특성 유지하면서 10 대처할 수있는 노드는 5,14 이다
  - 오른쪽 서브트리에서 가장 작은값 찾기 최소 찾자.
  - 왼쪽 서브트리에서 가장 큰 값 찾기

```
TREE-MINIMUM(x)
  while x.left ≠ NIL
      x = x.left
  return x
```



minimum을 찾자.

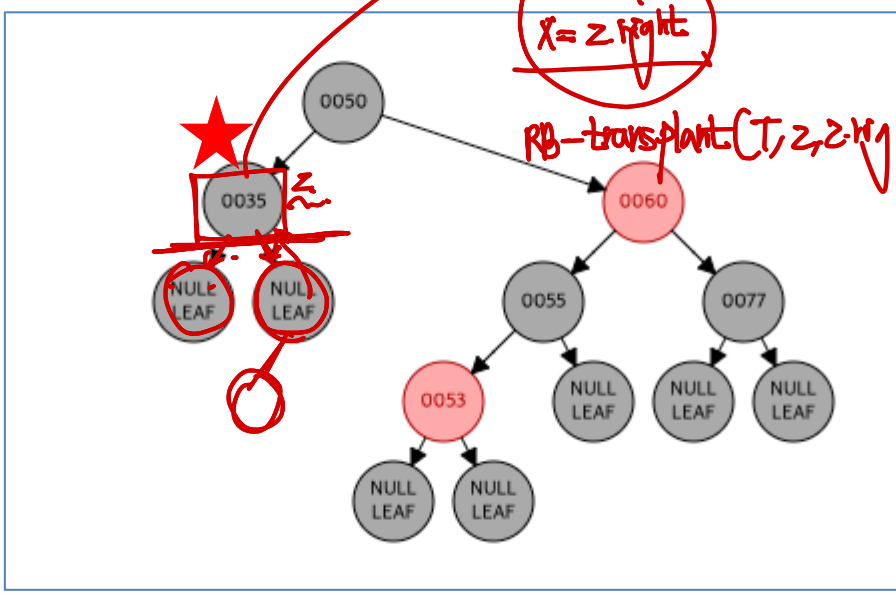
Red-Black Tree – Delete

- 예제 - target 확인, target color 따로 저장
  - Target = 35노드
  - target-original-color = target -> color
    - 지워질 노드의 색을 변수에 저장

검은색이면 나중에 fix-up

x= z.right

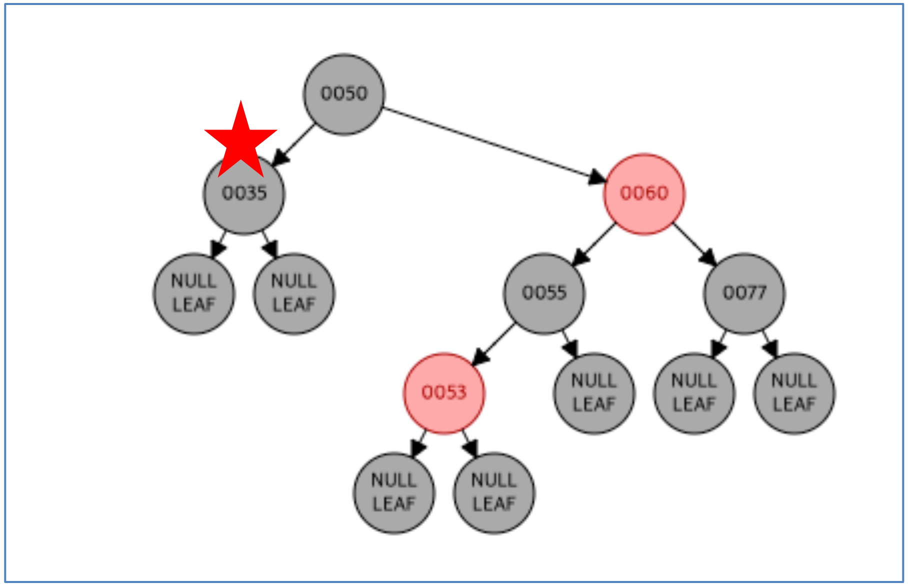
RB-transplant(T,z,z.right)




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

Red-Black Tree – Delete

- 예제 - target 자식 정보 확인
  - 삭제할 노드를 y에 복사하여 포인터처럼 활용하면서 쓸 것
  - 대체할 노드를 x에 따로 저장해놓음(정보저장용)

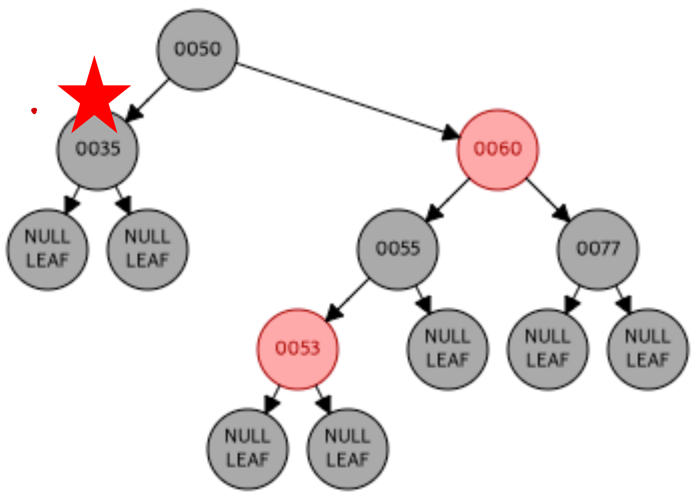



```
RB-DELETE(T, z)
  y = z  # y에 저장
  y-original-color = y.color
  if z.left == T.nil
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
  elseif z.right == T.nil
      x = z.left
      RB-TRANSPLANT(T, z, z.left)
  else y = TREE-MINIMUM(z.right)
      y-original-color = y.color
      x = y.right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

Red-Black Tree – Delete

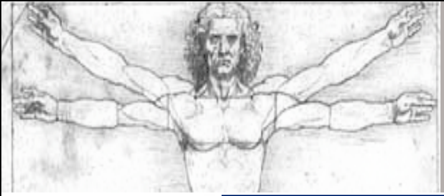
- 예제 – case 1:왼쪽 자식이 없을때
  - target의 오른쪽 자식을 x 로 설정.
  - 오른쪽 자식을 target 노드에 이식 (transplant)
  - 자식이 둘다 없을 때도 해당

[illegible]



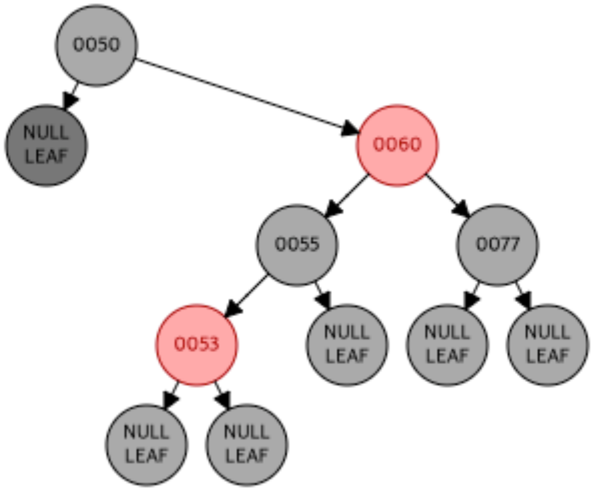
RB-DELETE(T, z)

$y = z$
y-*original-color* $= y.color$
if $z.left == T.nil$
  $x = z.right$
  RB-TRANSPLANT$(T, z, z.right)$
elseif $z.right == T.nil$
  $x = z.left$
  RB-TRANSPLANT$(T, z, z.left)$
else $y =$ TREE-MINIMUM$(z.right)$
  y-*original-color* $= y.color$
  $x = y.right$
  if $y.p == z$
    $x.p = y$
  else RB-TRANSPLANT$(T, y, y.right)$
    $y.right = z.right$
    $y.right.p = y$
  RB-TRANSPLANT(T, z, y)
  $y.left = z.left$
  $y.left.p = y$
  $y.color = z.color$
if y-*original-color* $==$ BLACK
  RB-DELETE-FIXUP(T, x)



Red-Black Tree – Delete

- 예제 – case 1:왼쪽 자식이 없을때
  - target의 오른쪽 자식을 x 로 설정.
  - 오른쪽 자식을 target 노드에 이식 (transplant)
  - 자식이 둘다 없을 때도 해당

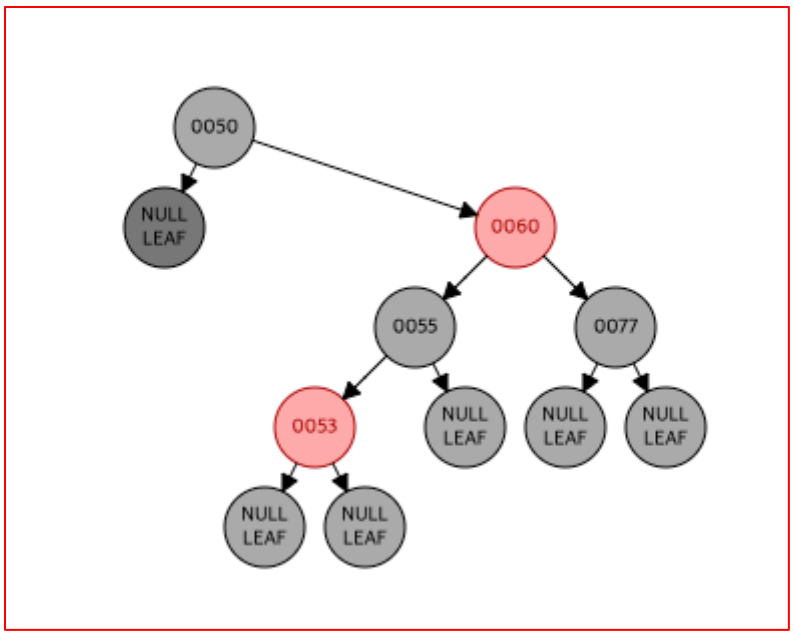



```
RB-DELETE(T, z)
  y = z
  y-original-color = y.color
  if z.left == T.nil
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
  elseif z.right == T.nil
      x = z.left
      RB-TRANSPLANT(T, z, z.left)
  else y = TREE-MINIMUM(z.right)
      y-original-color = y.color
      x = y.right
      if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

Red-Black Tree – Delete

- 예제 – case 2:오른쪽 자식이 없을 때
  - target의 왼쪽 자식을 x 로 설정.
  - 왼쪽 자식을 target 노드에 이식(transplant)

있는 자식 해결




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

Red-Black Tree – Delete

- 예제 – case 3:자식이 둘다 있을때
  - 오른쪽 서브트리에 최소 노드(Successor) 검색 후 RB-transplant
- 만약 삭제하는 node의 색이 적색일때?
  - 그냥 삭제한다
  - 레드블랙 특성이 유지된다
    - 흑색 높이는 그대로 유지
    - 적색노드가 인접하지 않는다
    - 루트는 원래 흑색이다
- 만약 삭제 노드가 블랙이라면?
  - Leaf노드까지의 블랙노드의 개수는 모두 일정해야 한다 ->깨짐
  - 또는 연속으로 레드가 나오는 케이즈 등장
  - RB-Delete-Fixup으로 고치자

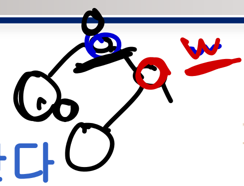
y=z
y-original-color
rb-transplant(T, y, y.right)
y.right = z.right
y.right.p = y
rb-transplant(T, z, y)
y.left = z.left
y.left.p = y
y.color = z.color

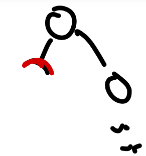
```
RB-DELETE(T, z)
  y = z
  y-original-color = y.color
  if z.left == T.nil
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
  elseif z.right == T.nil
      x = z.left
      RB-TRANSPLANT(T, z, z.left)
  else y = TREE-MINIMUM(z.right)
      y-original-color = y.color
      x = y.right
      if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

$$\begin{tikzcd} & T(n) \arrow[dl] \arrow[d] \\ T(n-1) & \underline{T(1)} \end{tikzcd}$$

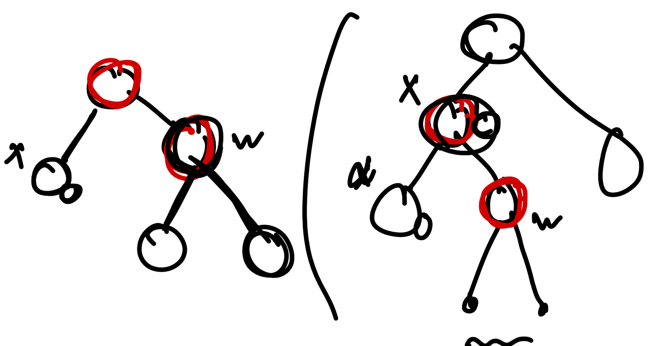
Red-Black Tree – Delete 4가지 case

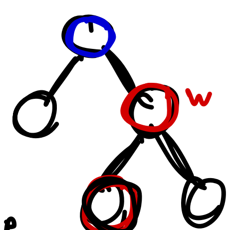
- 삭제 후 fixup
  - 특성 1,2,4를 복구한다
  - Fixup과정에서 중심은 x노드이다
  - Case1,2: x는 대체되는 노드 case3: x는 대체되는 노드의 오른쪽 자식

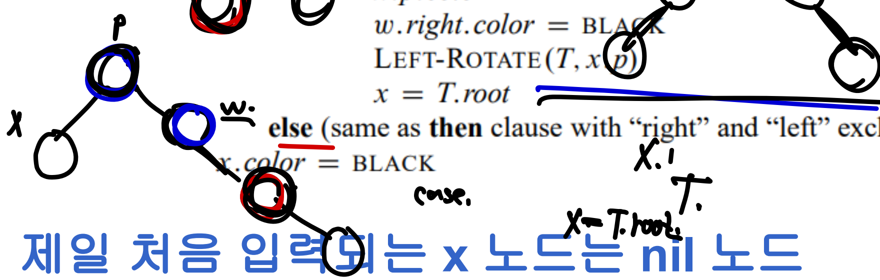




```
RB-DELETE-FIXUP(T, x)
  while x ≠ T.root and x.color == BLACK
      if x == x.p.left
          w = x.p.right
          if w.color == RED
              w.color = BLACK                                  // case 1
              x.p.color = RED                                  // case 1
             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







case.

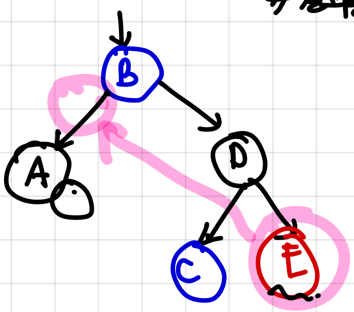
X.1

T.

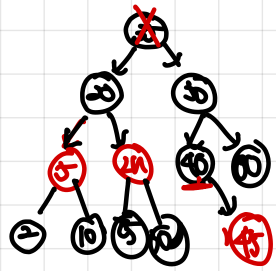
X=T.root

- delete_fix_up 과정에 제일 처음 입력되는 x 노드는 nil 노드
  - Successor, predecessor관계를 잘 따져보자

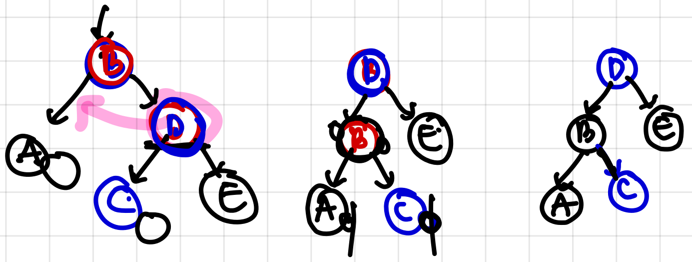
결과론적으로 D를 루트 B E를 검도사

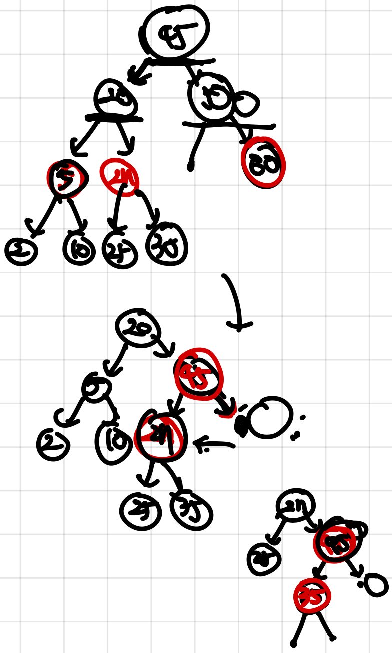


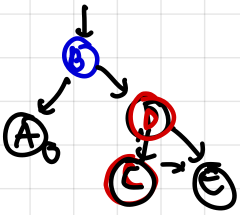
". 오른쪽 형제 black
→ 자녀가 red,



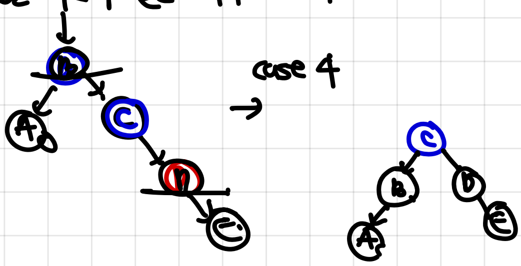
Red는 Doubly-black 위로 옮기기,

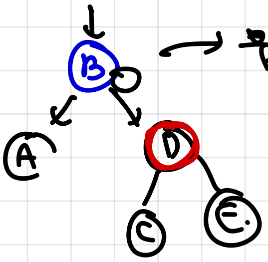






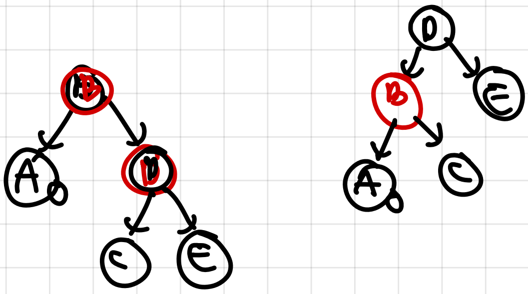
C를 이전과 같은 위치로 옮기자.

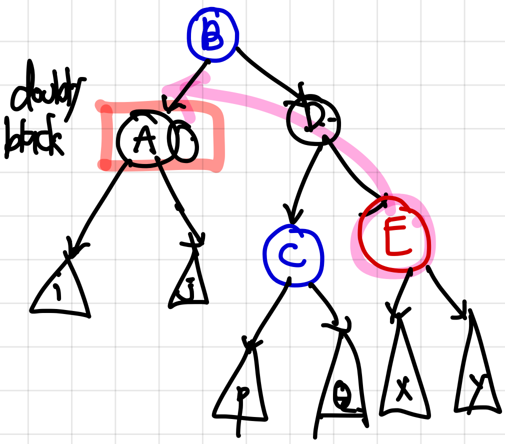




→ 부모가 레드면 black

올렸으면 → 오차시 키 무관 반복





i) doubly black의 오른쪽 형제가 black

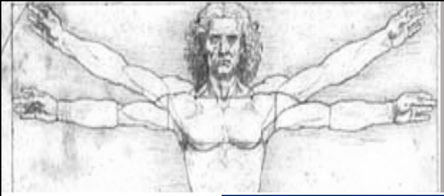
(& 그 형제의 오른쪽 자녀가 red)

→ red를 doubly black 위로 옮기고

옮긴 red로 extra-black

= red-and-black

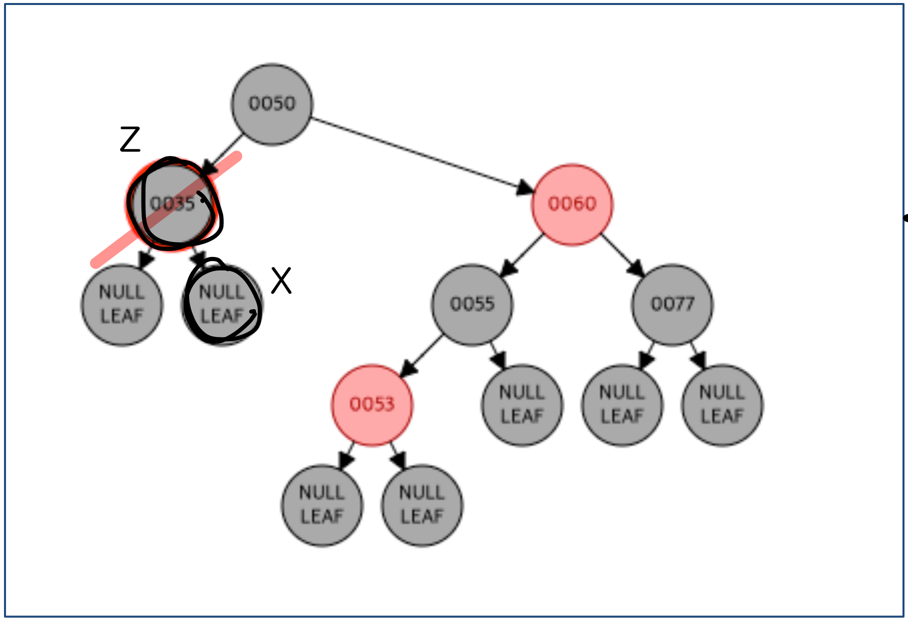
→ black



Red-Black Tree – Delete

- **Delete-fixup**
  - Target: 35, 이걸 삭제한다고 가정
  - Y.org.color = black, x = y.right

```
RB-DELETE(T, z)
  y = z
  y-original-color = y.color
  if z.left == T.nil
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
```



y = z

x = z.right

RB-TRANSPLANT(T, z, z.right)

50

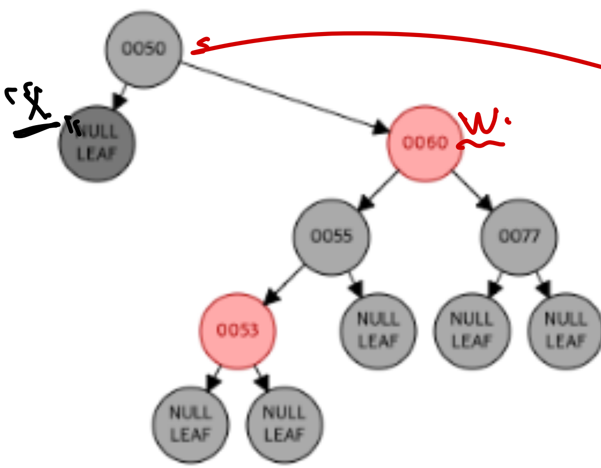
NIL

Red-Black Tree – Delete

- **Delete-fixup**
  - RB-Transplant 이후
  - X의 사촌을 w(60)에 저장

RB-DELETE(T, z)

$y = z$
y-*original-color* $= y.color$
if $z.left == T.nil$
    $x = z.right$
    RB-TRANSPLANT$(T, z, z.right)$



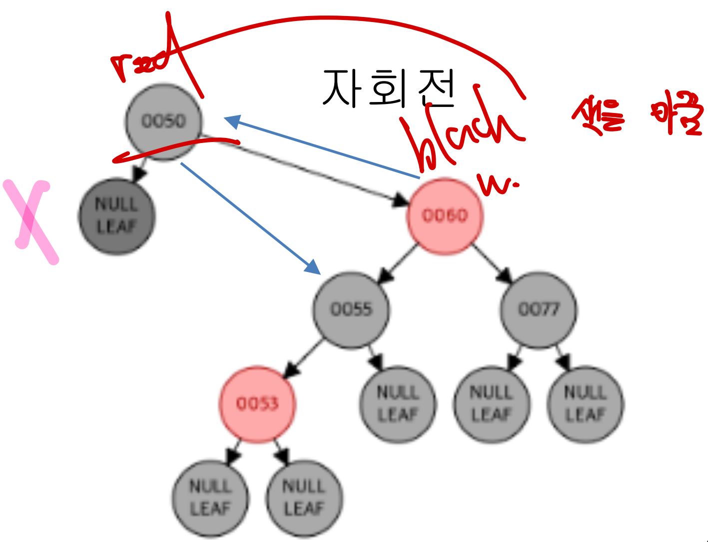
RB-DELETE-FIXUP(T, x)
while $x \neq T.root$ and $x.color$ == BLACK
    if $x == x.p.left$
        $w = x.p.right$

Red-Black Tree – Delete

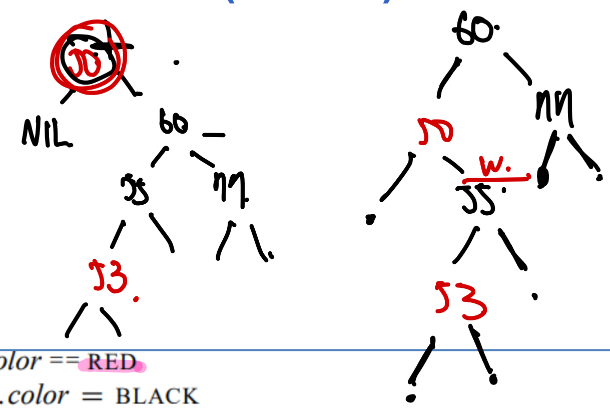
- **Delete-fixup**
  - Case1: w(60)이 적색노드일때?
  - (1) w 의 색깔과 x(nil)-> parent(50)의 색깔을 바꿔준다.
  - (2) x의 parent기준으로 좌회전!
  - (3) w를 x 의 parent의 오른쪽 자식으로 설정한다. (w = 55)

w.color : BLACK

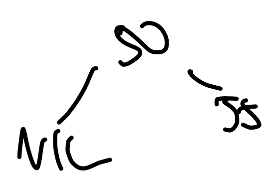
x.p.color = red



60

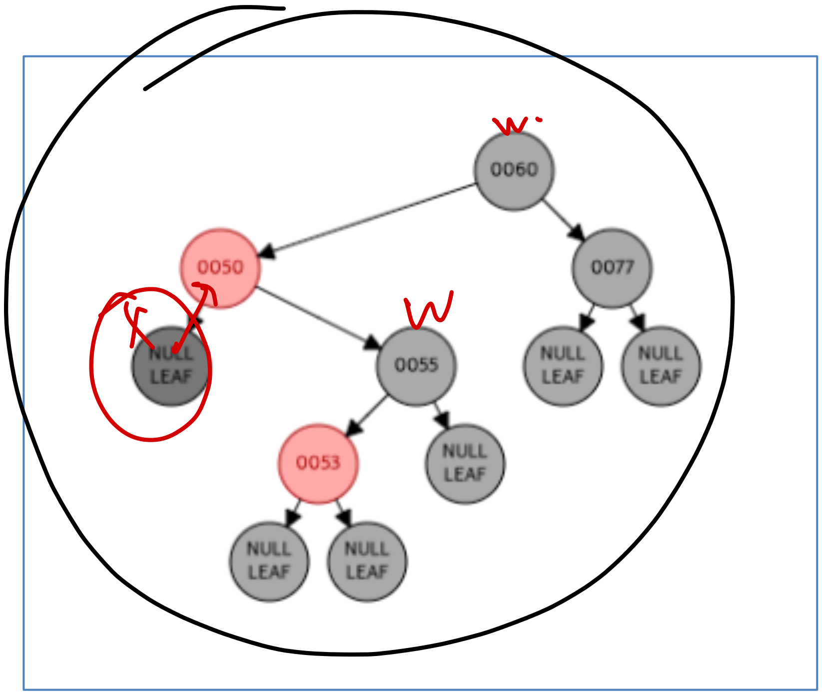


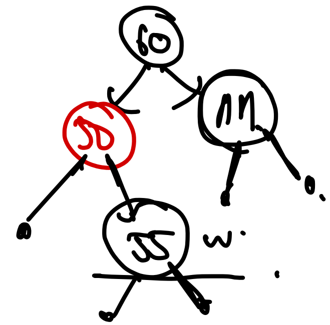
if $w.color$ == RED	
$\quad w.color$ = BLACK	// case 1
$\quad x.p.color$ = RED	// case 1
$\quad$ LEFT-ROTATE$(T, x.p)$	// case 1
$\quad w = x.p.right$	// case 1



Red-Black Tree – Delete

- **Delete-fixup**
  - Case1: w(60)이 적색노드일때?
  - (1) w 의 색깔과 x(nil)-> parent(50)의 색깔을 바꿔준다.
  - (2) x의 parent기준으로 좌회전!
  - (3) w를 x 의 parent의 오른쪽 자식으로 설정한다. (w = 55)






```
if w.color == RED
    w.color = BLACK                  // case 1
    x.p.color = RED                  // case 1
    LEFT-ROTATE(T, x.p)              // case 1
    w = x.p.right                    // case 1
```

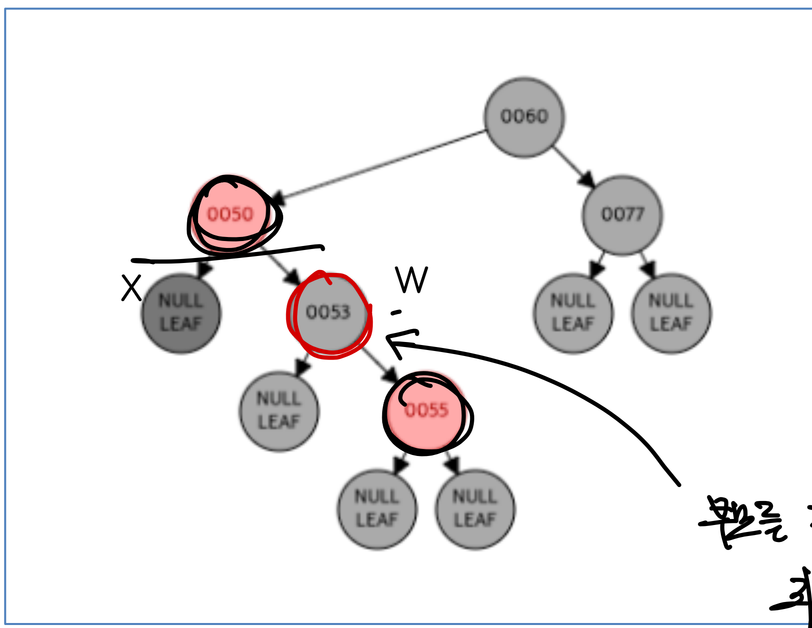
Red-Black Tree – Delete

- Delete-fixup
  - Case2: w 의 왼쪽 자식과 오른쪽 자식이 모두 black일 때
  - (1) w색을 레드로 칠한다.
  - (2) x를 x의 parent 로 설정한다.
    - (RB균형을 맞추도록 하면서 루트까지 진행.)

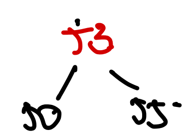
```
if w.color == RED
    w.color = BLACK                                   // case 1
    x.p.color = RED                                   // case 1
    LEFT-ROTATE(T, x.p)                               // case 1
    w = x.p.right                                     // case 1
if w.left.color == BLACK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

Red-Black Tree – Delete

- **Delete-fixup**
  - Case3: w의 왼쪽 자식이 red, w 의 오른쪽 자식이 black 일 때
  - (1) w의 왼쪽자식과 w의 색을 바꾼다.
  - (2) w에 대해서 우회전
  - (3) x 의 parent의 오른쪽 자식으로 w를 설정

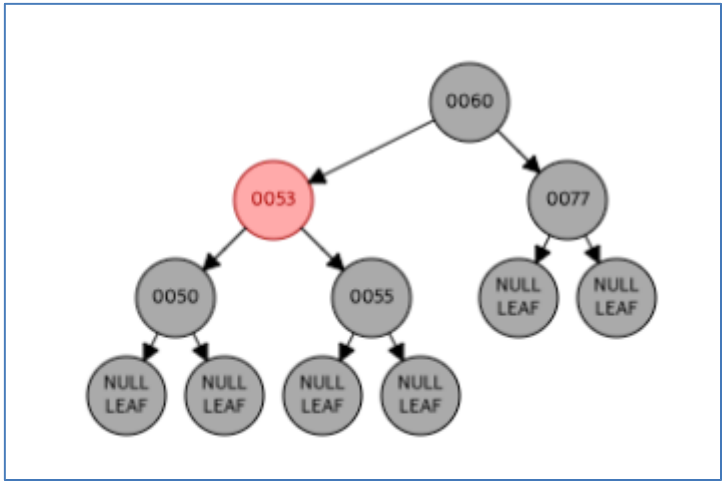



```
if w.color == RED
    w.color = BLACK                                   // case 1
    x.p.color = RED                                   // case 1
    LEFT-ROTATE(T, x.p)                               // case 1
    w = x.p.right                                     // case 1
if w.left.color == BLACK and w.right.color == BLACK
    w.color = RED                                     // case 2
    x = x.p                                           // case 2
else if w.right.color == BLACK
        w.left.color = BLACK                          // case 3
        w.color = RED                                 // case 3
        RIGHT-ROTATE(T, w)                            // case 3
        w = x.p.right                                 // case 3
```



Red-Black Tree – Delete

- **Delete-fixup**
  - case 3를 거치면 case 4의 형태로 바뀐다
  - case 4를 거치면 fix_up 과정이 종료
  - (1) w(53) 의 색깔과 x(nil)-> parent(50)의 색깔을 바꿔준다.
  - (2) w의 오른쪽 자식의 색을 검정색으로 바꾼다.
  - (3) x의 parent 기준으로 좌회전!
  - (4) x를 root 로 설정(while문 탈출)
- **최종적으로 레드블랙 트리의 속성을 만족!**

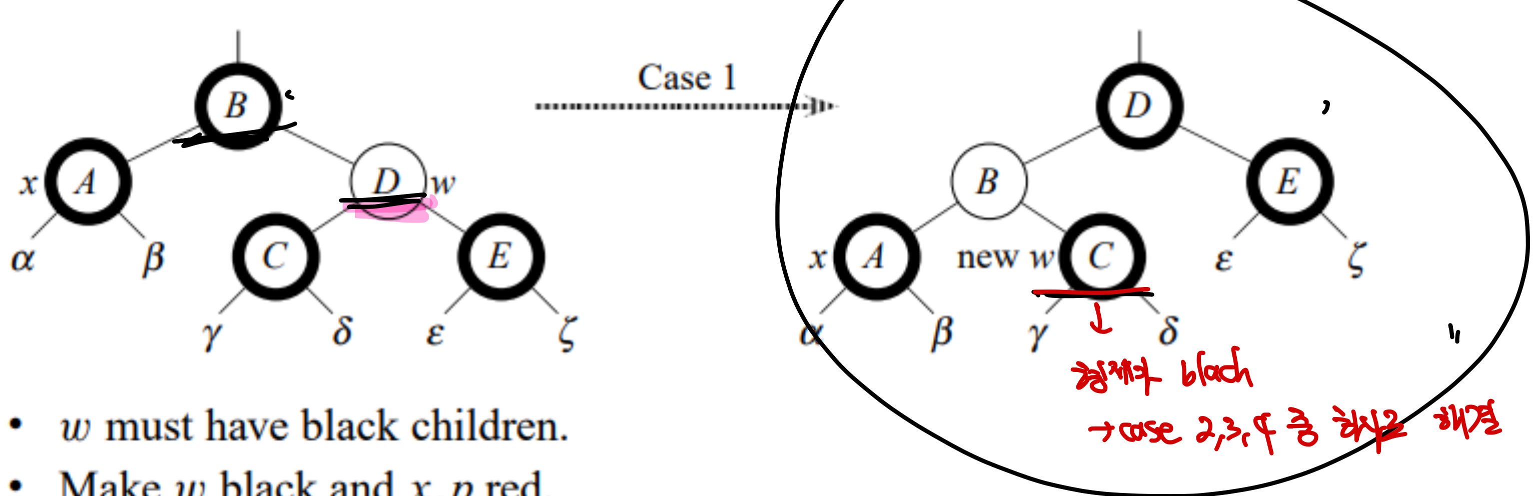



```
if w.left.color == BLACK and w.right.color == BLACK
     w.color = RED                                   // case 2
     x = x.p                                         // case 2
else if w.right.color == BLACK
          w.left.color = BLACK                       // case 3
          w.color = RED                              // case 3
          RIGHT-ROTATE(T, w)                         // case 3
          w = x.p.right                              // case 3
     w.color = x.p.color                             // case 4
     x.p.color = BLACK                               // case 4
     w.right.color = BLACK                           // case 4
     LEFT-ROTATE(T, x.p)                             // case 4
     x = T.root                                      // case 4
```

Red-Black Tree – Delete

- **Delete-fixup(교재)**

Case 1: w is red

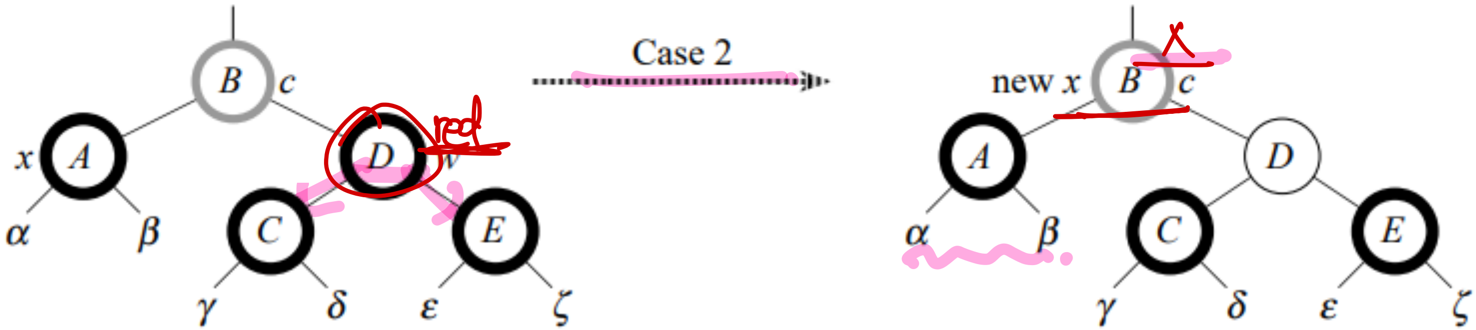


- w must have black children.
- Make w black and $x.p$ red.
- Then left rotate on $x.p$.
- New sibling of x was a child of w before rotation $\Rightarrow$ must be black.
- Go immediately to case 2, 3, or 4.

Red-Black Tree – Delete

- **Delete-fixup(교재)**

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



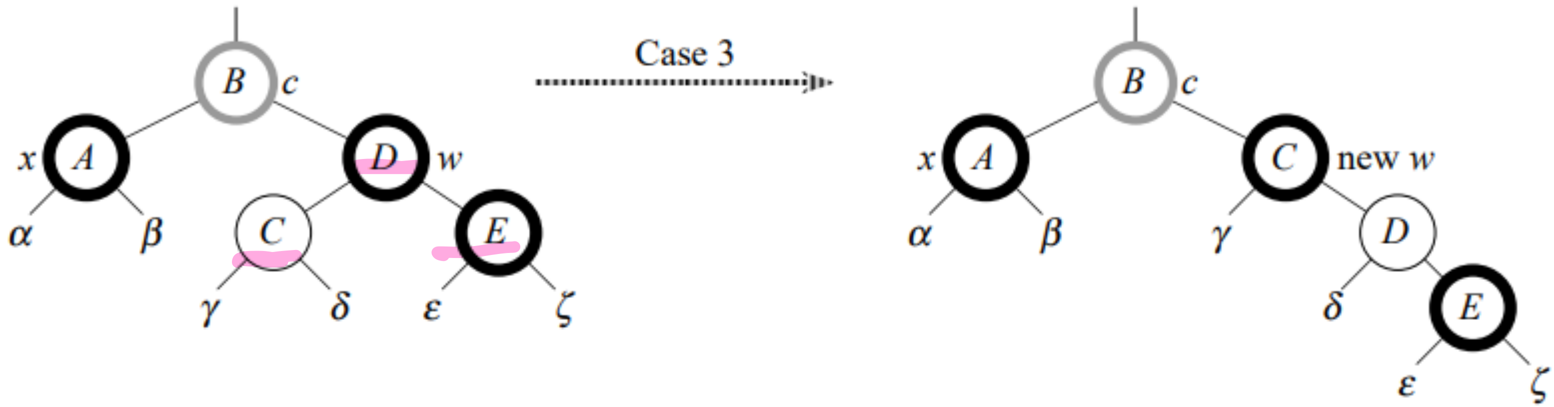
[Node with gray outline is of unknown color, denoted by c.]

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

Red-Black Tree – Delete

- **Delete-fixup(교재)**

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

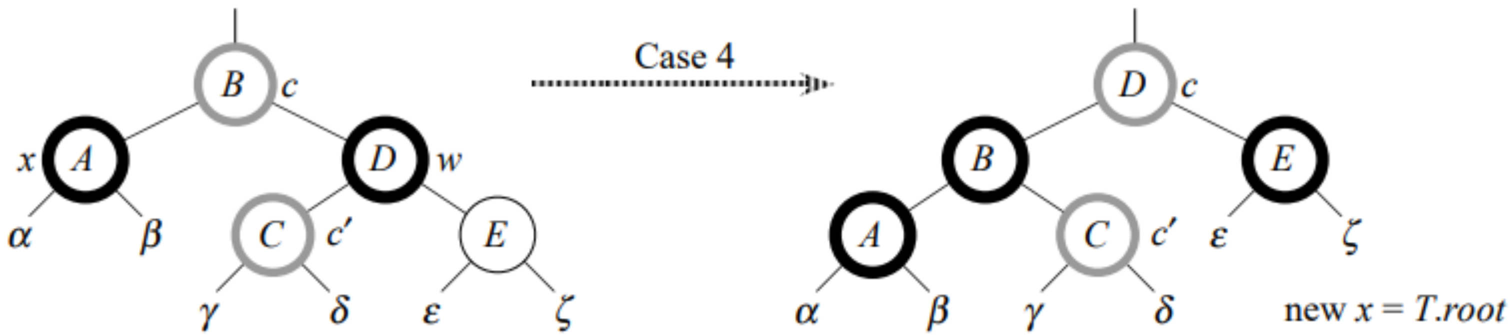


- Make w red and w's left child black.
- Then right rotate on w.
- New sibling w of x is black with a red right child $\Rightarrow$ case 4.

Red-Black Tree – Delete

- **Delete-fixup(교재)**

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



[Now there are two nodes of unknown colors, denoted by c and c'.]

- Make w be $x.p$'s color (c).
- Make $x.p$ black and w's right child black.
- Then left rotate on $x.p$.
- 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.

Red-Black Tree – Delete

- **RB-delete – analysis**
  - 노드 n개인 레드블랙 트리 높이는 O(lgn)
  - RB-delete-fixup빼고는 O(lgn)이다
  - 경우 1,2,3,4 나누어서 생각
  - 색깔변경, 회전은 최대 3번 상수 c
  - 경우 2에는 while문 루프가 반복된다 x->x.p
  - 최대 O(lgn)번 움직이고 회전은 수행 x
  - 따라서 최대 O(lgn)시간이 소요된다

pseudo-code가 핵심

$O(2\lg n + C)$

$\therefore O(\lg n)$

lgn

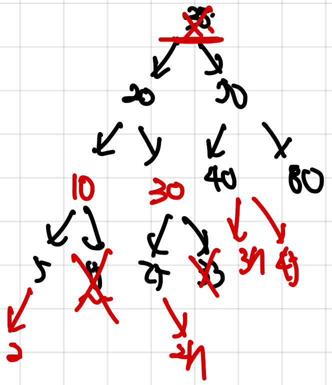
정리

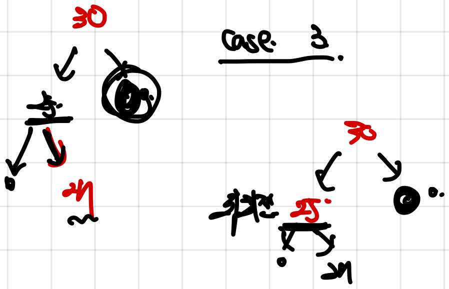
· 삭제되는 색이 black

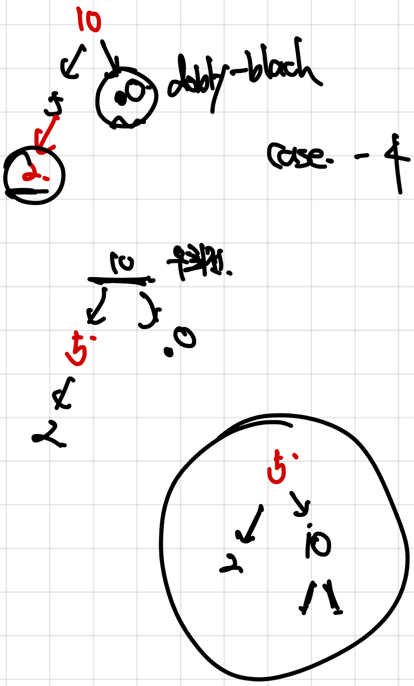
삭제되는 색이 있던 위치를 대체한 노드에 extra black을 부여.

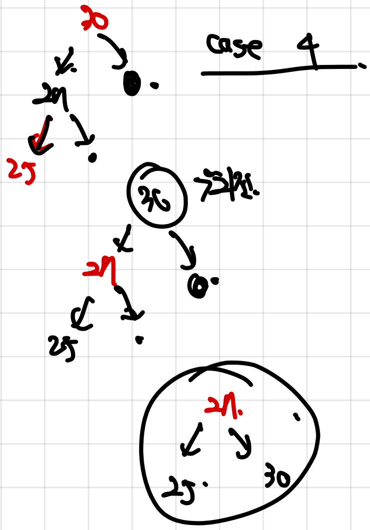
대체한 노드가 red-and-black = black

doubly-black → case 1,2,3,4.

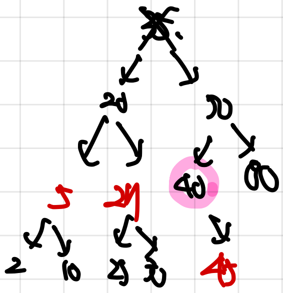




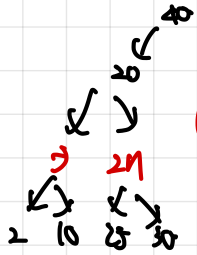




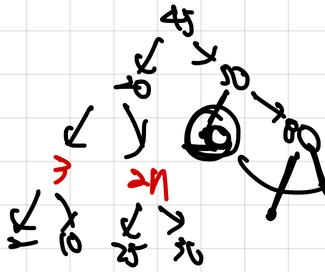
predecessor



→ 45가 40을 대체



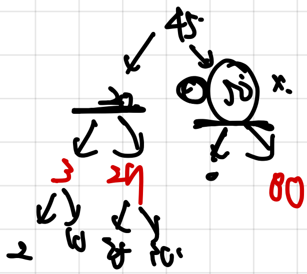
→ 40 삭제



case. 2

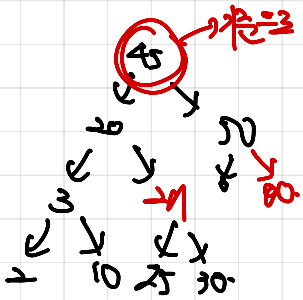
nil node + black

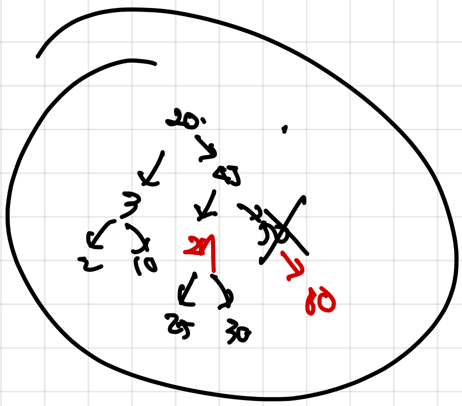
"double-black"

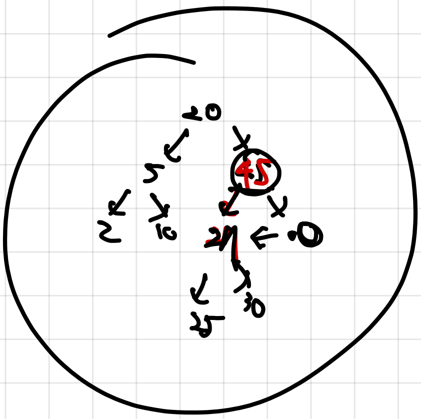


case. 4

부모를 right rotate.







case - 1