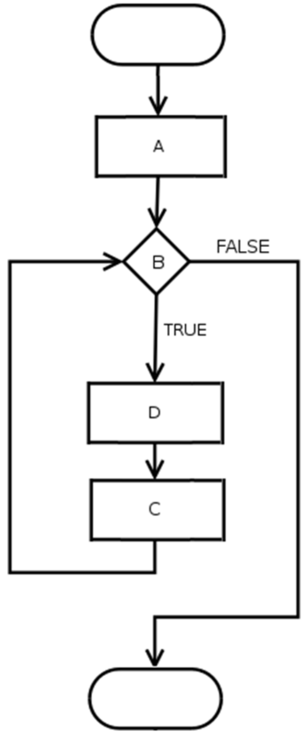
자료구조 & 알고리즘

```
for(A;B;C)
  D;
```

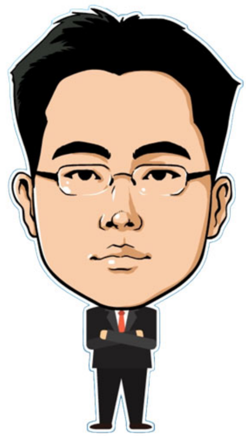


탐색 트리
(Search Tree)

Seo, Doo-Ok

Clickseo.com

clickseo@gmail.com



목 차

- 이진 탐색 트리

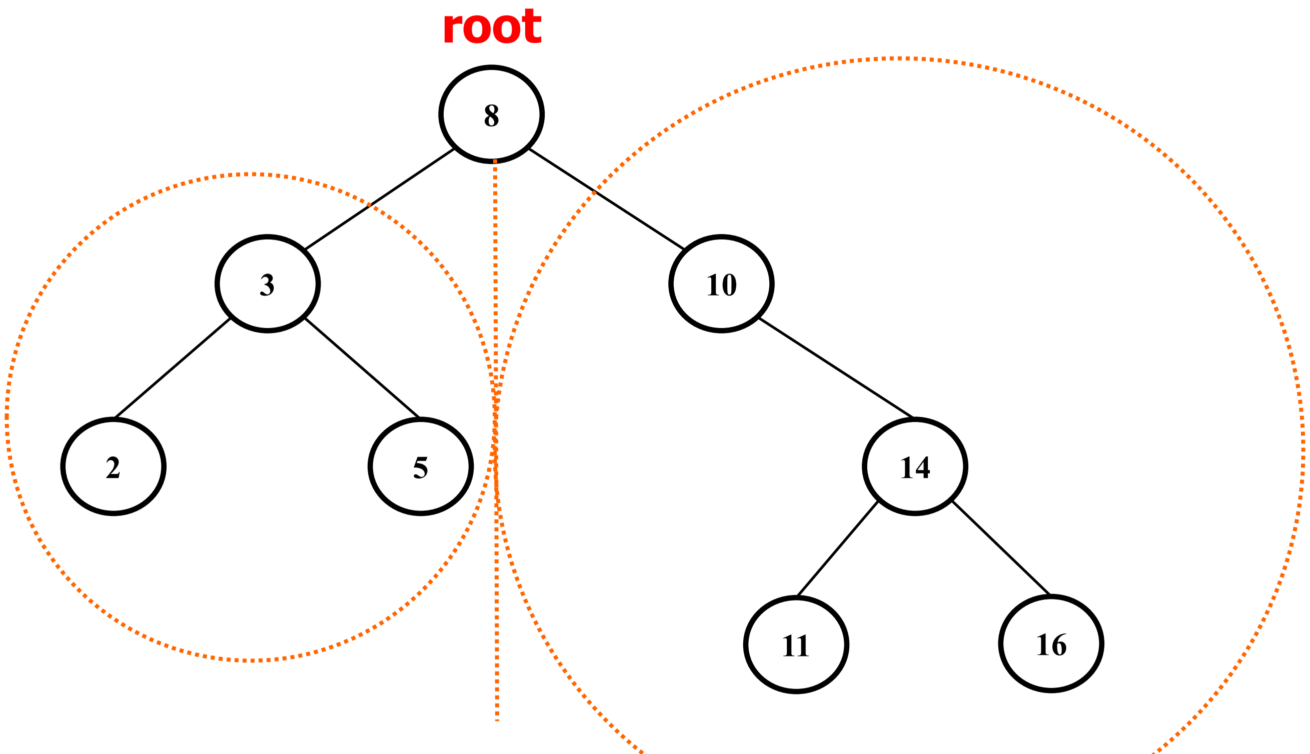
- 균형 탐색 트리

이진 탐색 트리

- 이진 탐색 트리
    - 이진 탐색 트리 연산
- 균형 탐색 트리

이진 탐색 트리 (1/3)

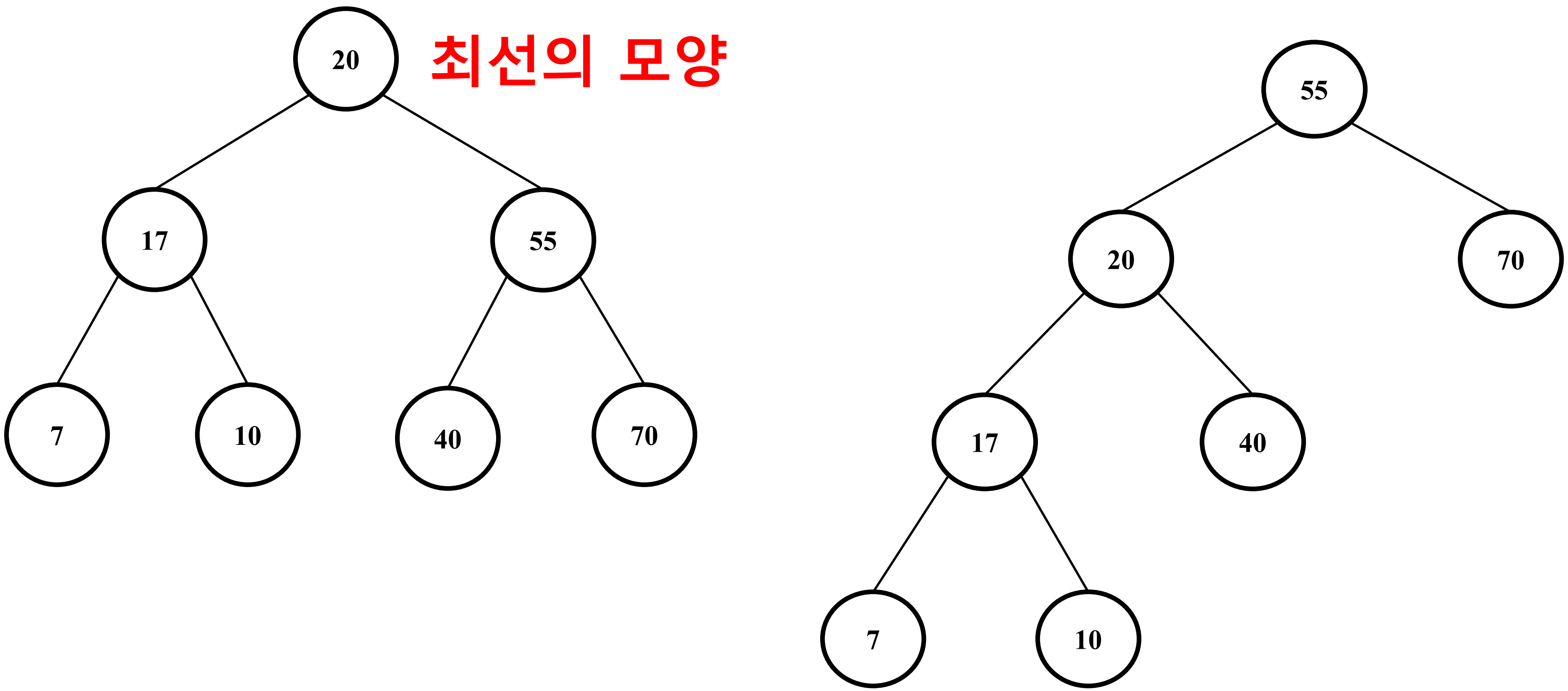
- **이진 탐색 트리**(Binary Search Tree)
  - 모든 노드는 서로 다른 키를 갖는다(유일한 키 값).
    - 각 노드는 최대 2개의 자식을 갖는다.



왼쪽 서브 트리의 키 값 < 루트의 키 값 < 오른쪽 서브 트리의 키 값

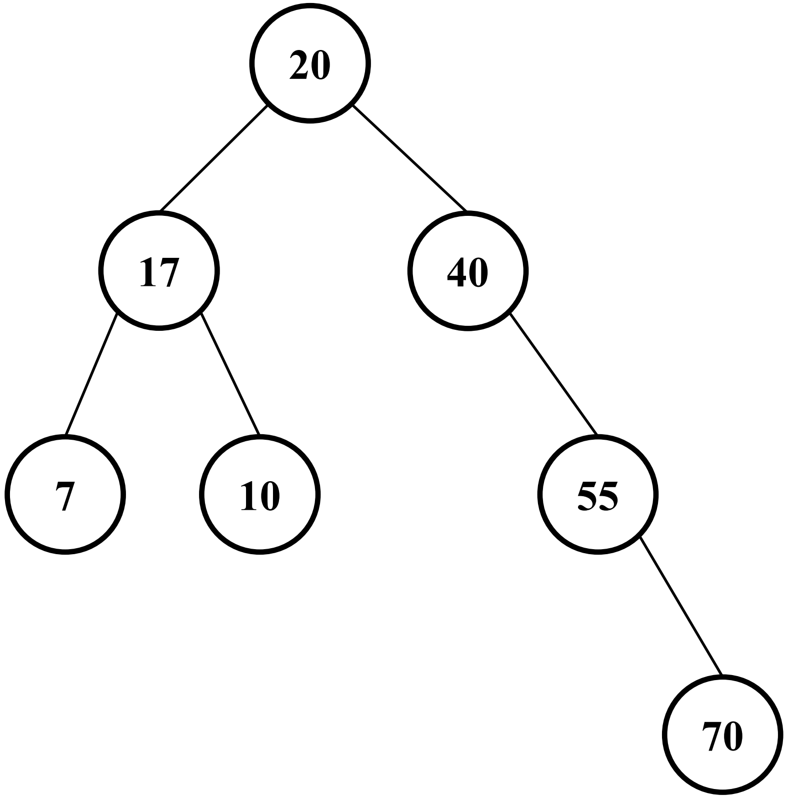
이진 탐색 트리 (2/3)

- **이진 탐색 트리**
  - 같은 데이터와 다른 이진 탐색 트리 #1

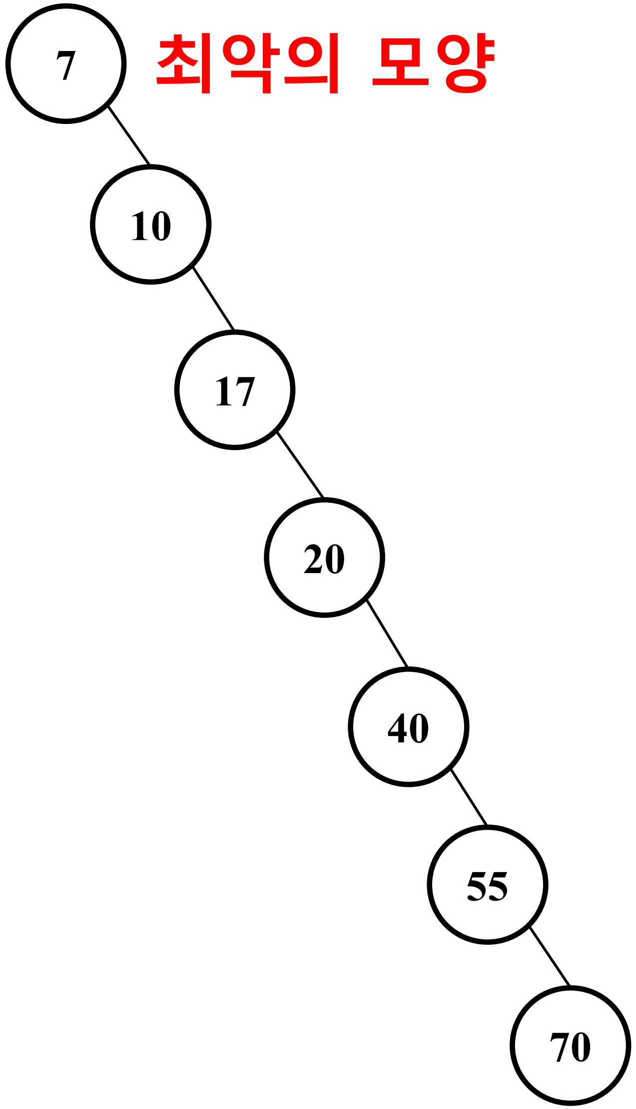


이진 탐색 트리 (3/3)

- 이진 탐색 트리
  - 같은 데이터와 다른 이진 탐색 트리 #2



최악의 모양

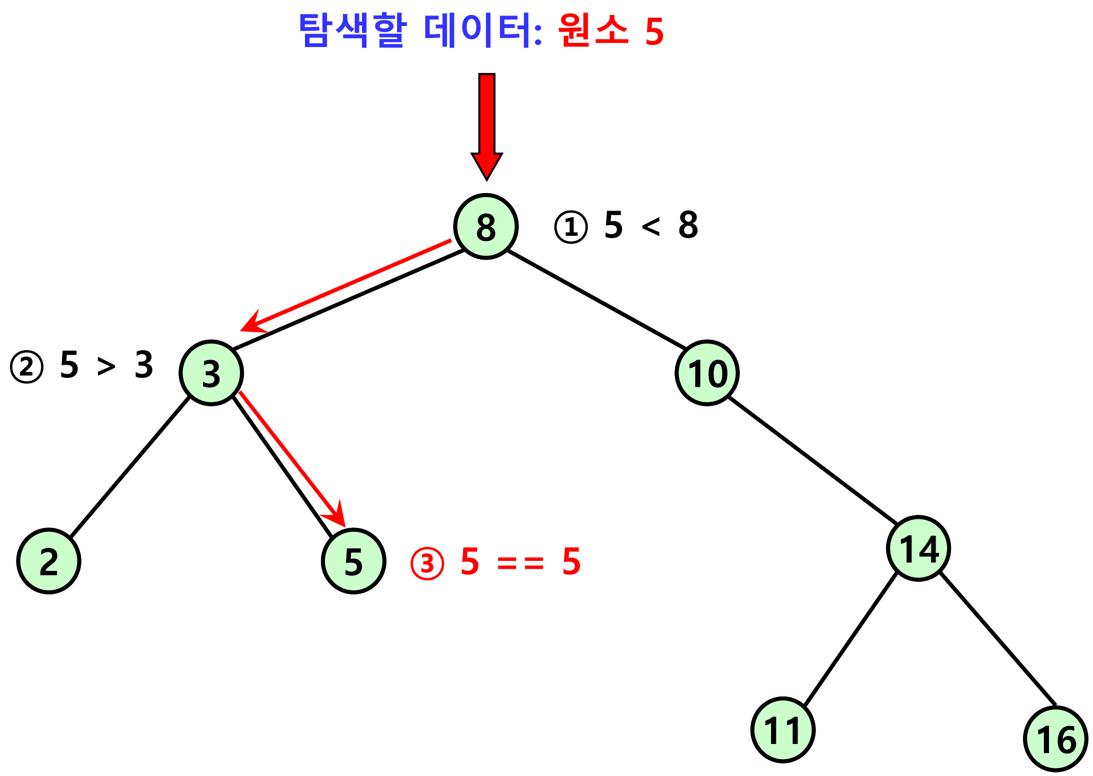


이진 탐색 트리

이진 탐색 트리 연산: 탐색, 삽입, 삭제

이진 탐색 트리 (1/7)

- **이진 탐색 트리:** 탐색
  - 탐색 과정

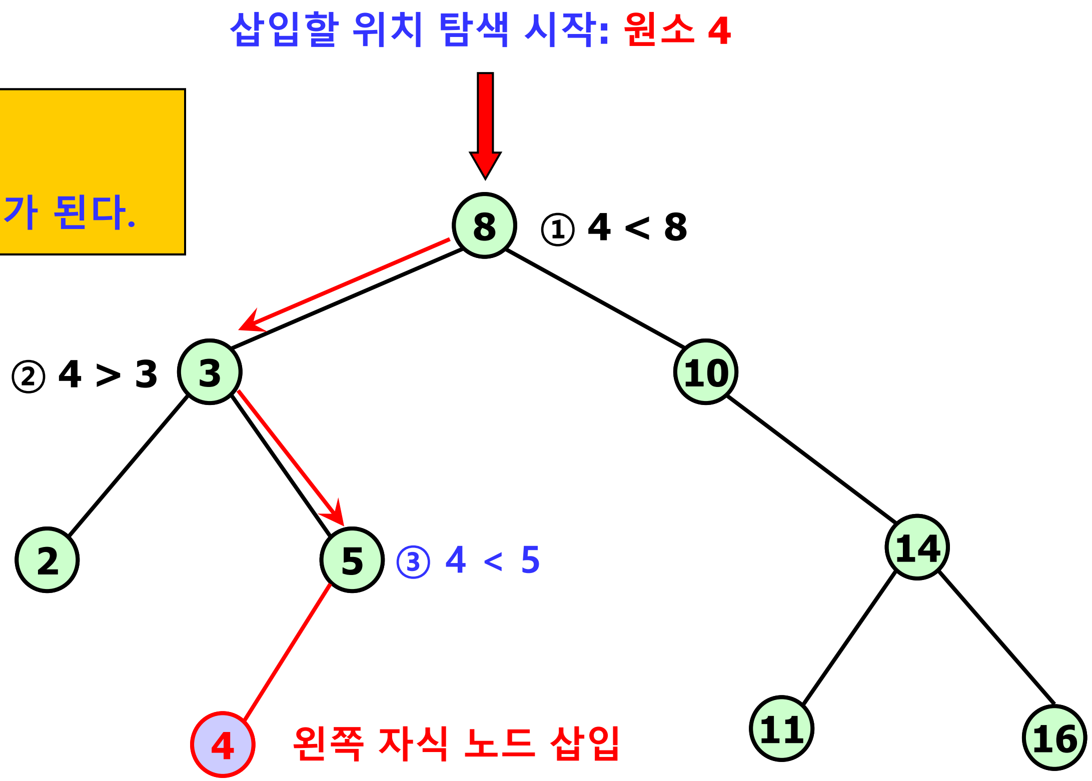


이진 탐색 트리 (2/7)

- **이진 탐색 트리: 삽입**
  - 삽입 과정
    1. 삽입할 노드의 위치(부모 노드의 주소) 탐색
    2. 노드 삽입

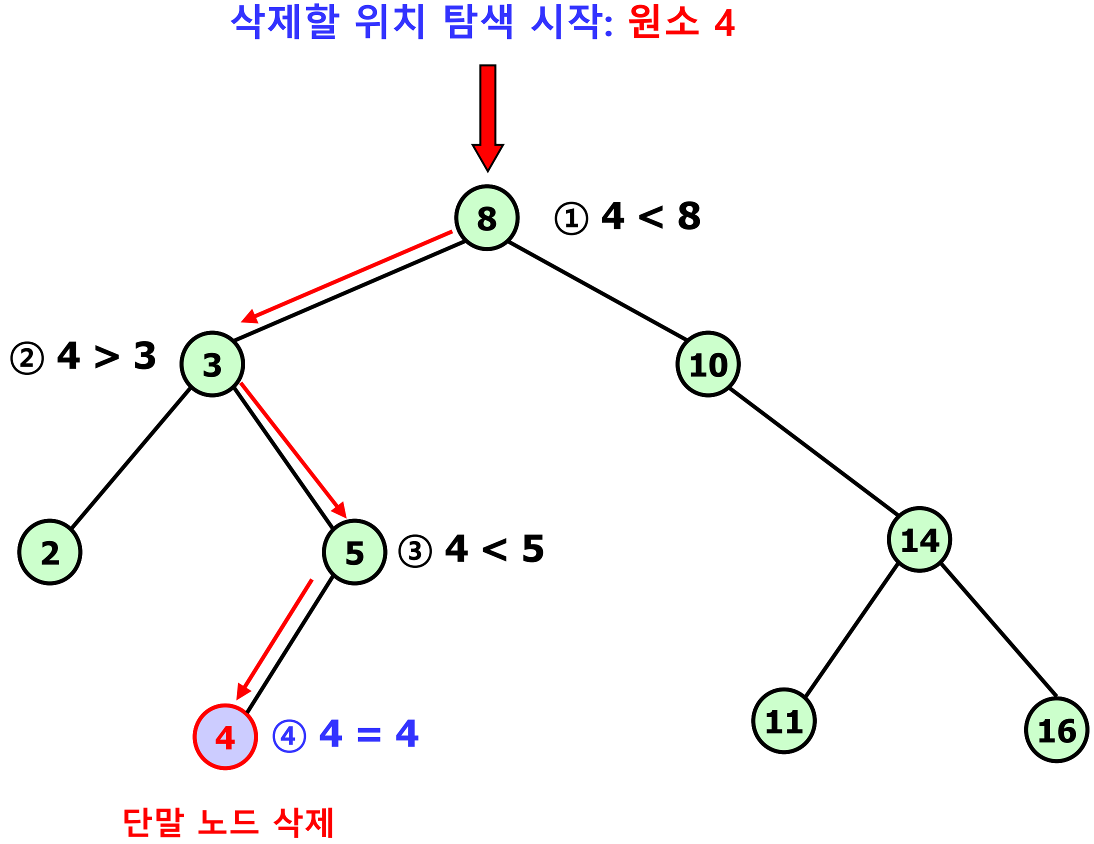
"탐색 실패가 결정 된 위치 "

즉, 왼쪽 자식 노드의 위치가 삽입 할 자리가 된다.



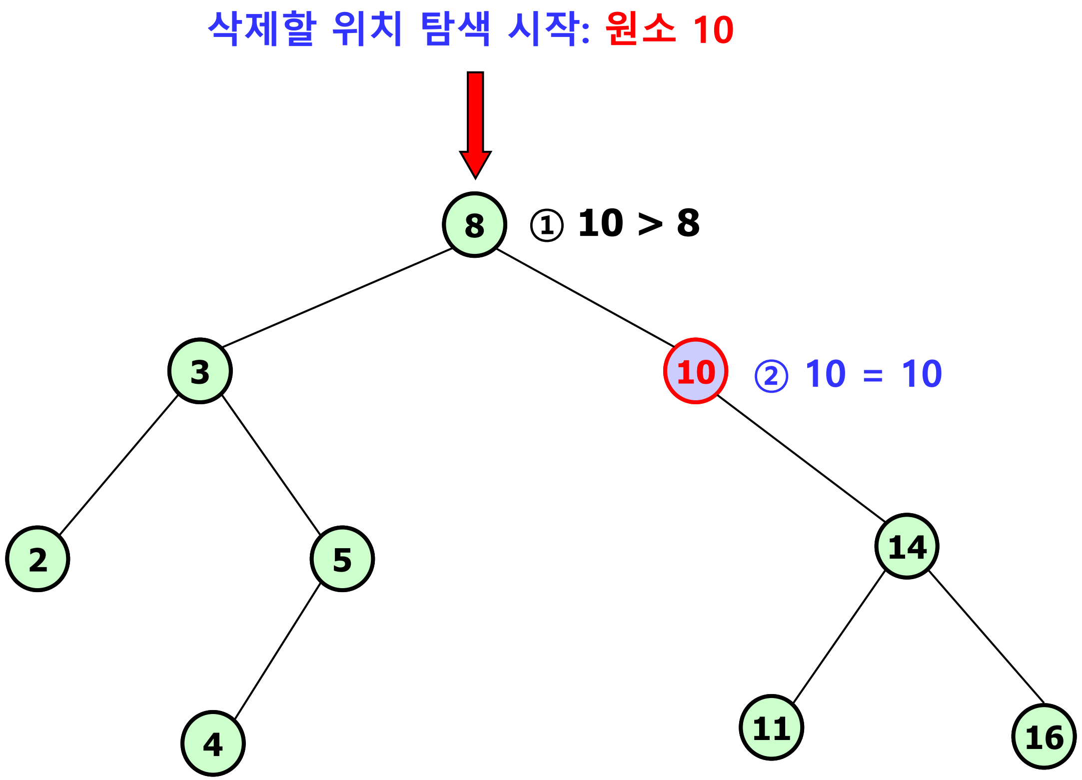
이진 탐색 트리 (3/7)

- **이진 탐색 트리:** 삭제 #1
  - 삭제 과정: 단말 노드



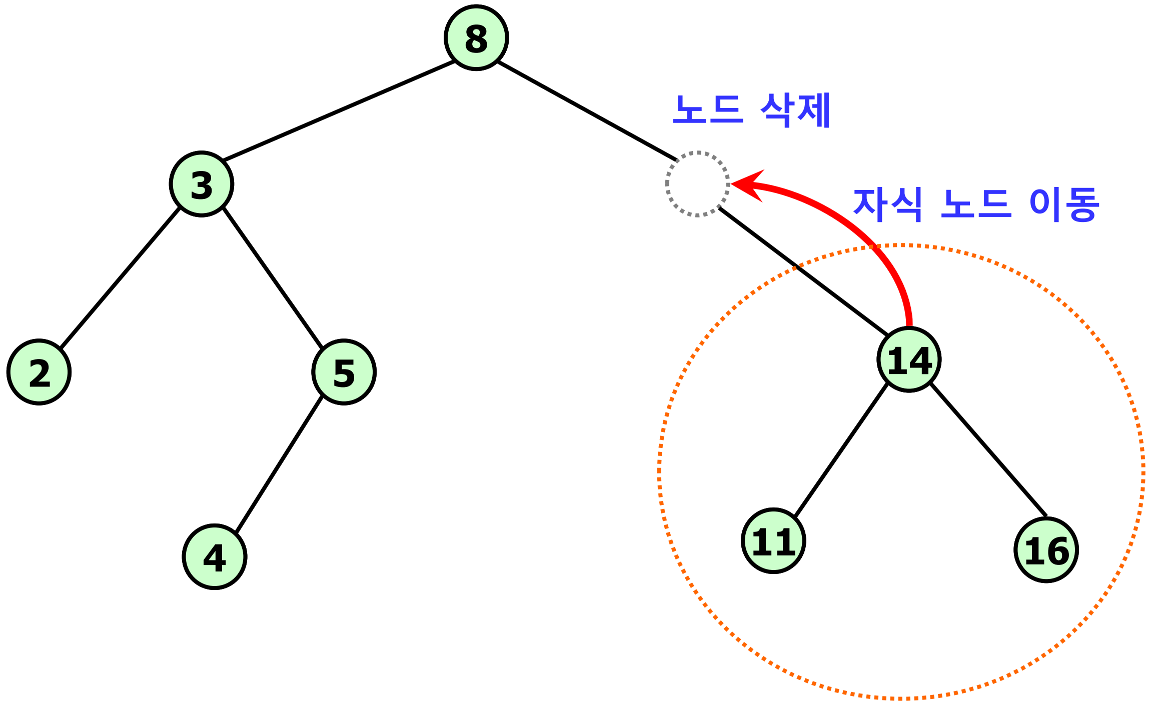
이진 탐색 트리 (4/7)

- **이진 탐색 트리:** 삭제 #2
  - 삭제 과정: 하나의 자식 노드만 존재
    1. 삭제할 노드의 탐색



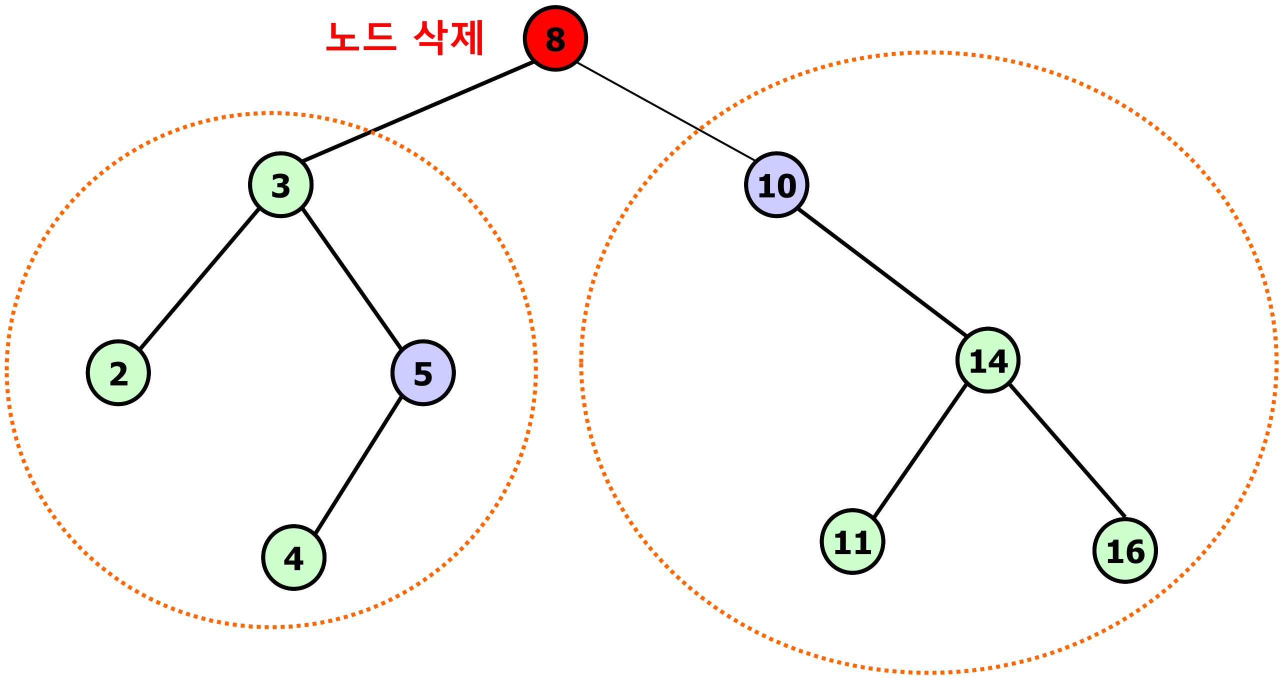
이진 탐색 트리 (5/7)

- **이진 탐색 트리:** 삭제 #2
  - 삭제 과정: 하나의 자식 노드만 존재
    2. 삭제할 노드의 삭제 및 위치 조정



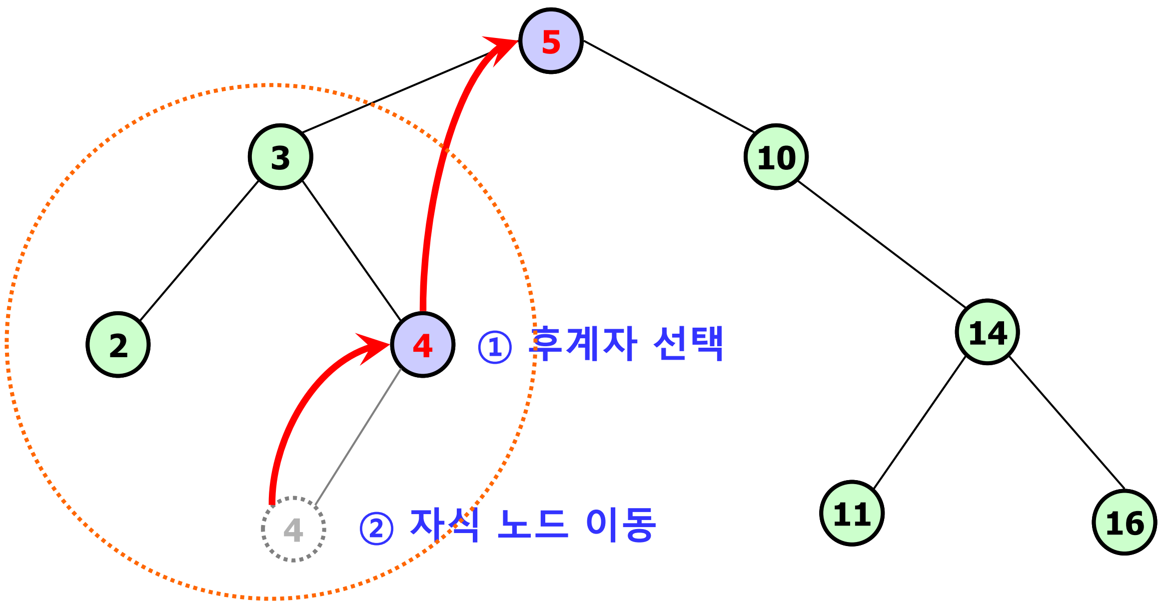
이진 탐색 트리 (6/7)

- **이진 탐색 트리: 삭제 #3**
  - 삭제 과정: 두 개의 자식 노드가 존재
    1. 삭제할 노드의 탐색 및 후계자 노드 선정
       - 왼쪽 서브 트리에서 가장 큰 키 값을 가진 노드
       - 오른쪽 서브 트리에서 가장 작은 키 값을 가진 노드



이진 탐색 트리 (7/7)

- **이진 탐색 트리:** 삭제 #3
  - 삭제 과정: 두 개의 자식 노드가 존재
    - **2. 트리 재구성:** 데이터 5를 가진 노드를 후계자로 선택한 경우



이진 탐색 트리

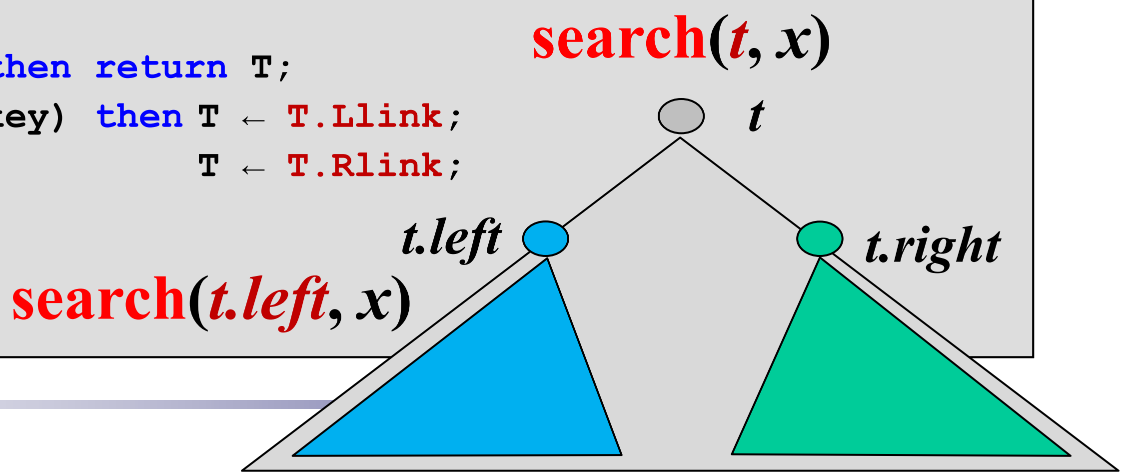
이진 탐색 트리 연산: 알고리즘

이진 탐색 트리 연산: 알고리즘 (1/3)

● 이진 탐색 트리 연산: 알고리즘(탐색)

```
// 재귀적 용법
searchBST(T, data)
    if (T = NULL) then                  return NULL;
    else if (data = T.key) then         return T;
    else if (data < T.key) then         return searchBST(T.Llink, data);
    else                                return searchBST(T.Rlink, data);
end searchBST()
```

```
// 비재귀적 용법
searchBST(T, data)
    while (T ≠ NULL) do
    {
        if (data = T.key) then return T;
        else if (data < T.key) then T ← T.Llink;
        else                        T ← T.Rlink;
    }
    return NULL;
end searchBST()
```

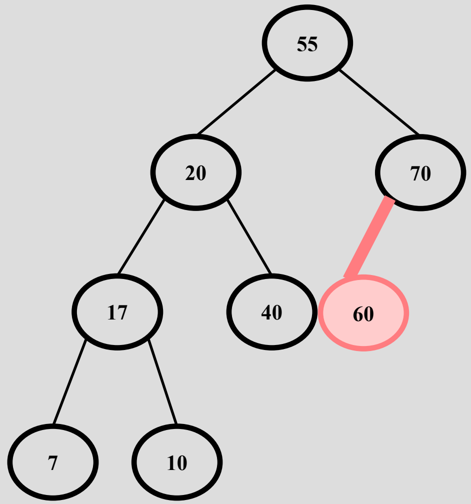


이진 탐색 트리 연산: 알고리즘 (2/3)

● 이진 탐색 트리 연산: 알고리즘(삽입)

```
// 재귀적 용법
insertBST(T, data)
    if (T = NULL) then T ← newDNode;
    else if (data < T.key) then      T.Llink = insertBST(T.Llink, data);
    else                             T.Rlink = insertBST(T.Rlink, data);
end insertBST()
```

```
// 비재귀적 용법
insertBST(T, data)
    while (T ≠ NULL) do
    {
          if (data = T.key) then
                    return error;
          parent ← T;
          if (data < T.key) then          T ← T.Llink;
          else                            T ← T.Rlink;
    }
    newDNode ← makeDNode(data);
    if (T = NULL) then                    T ← newDNode;
    else if (data < parent.key) then parent.Llink ← newDNode;
    else                                  parent.Rlink ← newDNode;
end insertBST()
```



이진 탐색 트리 연산: 알고리즘 (3/3)

● 이진 탐색 트리 연산: 알고리즘(삭제)

```
deleteBST(T, data)
    del ← 삭제할 노드;
    parent ← 삭제할 노드의 부모 노드;
    if (del = NULL) then return;
    if (del.Llink = NULL and del.Rlink = NULL) then {          // 단말 노드
          if (parent.Llink = del) then parent.Llink ← NULL;
          else parent.Rlink ← NULL;
    }
    else if (del.Llink = NULL or del.Rlink = NULL) then {      // 하나의 자식 노드
          if (del.Llink ≠ NULL) then {
                  if (parent.Llink = del) then parent.Llink ← del.Llink;
                  else parent.Rlink ← del.Llink;
          }
          else {
                  if (parent.Llink = del) then parent.Llink ← del.Rlink;
                  else parent.Rlink ← del.Rlink;
          }
    }
    else if (del.Llink ≠ NULL and del.Rlink ≠ NULL) {          // 두 개의 자식 노드
          max ← maxNode(del.Llink);         // min ← minNode(del.Rlink);
          del.key ← max.key;                // del.key ← min.key;
          deleteBST(del.Llink, del.key);    // deleteBST(del.Rlink, del.key);

    }
    end deleteBST()
```

균형 탐색 트리

- 이진 탐색 트리

- **균형 탐색 트리**
  - AVL 트리
  - 레드-블랙 트리
  - B 트리

균형 탐색 트리

- **균형 탐색 트리(Binary Search Tree)**
  - **이진 탐색 트리: AVL 트리, 레드-블랙 트리**

[이미지 출처: "IT CookBook, 쉽게 배우는 자료구조 with 파이썬", 문병로, 한빛아카데미, 2022.]

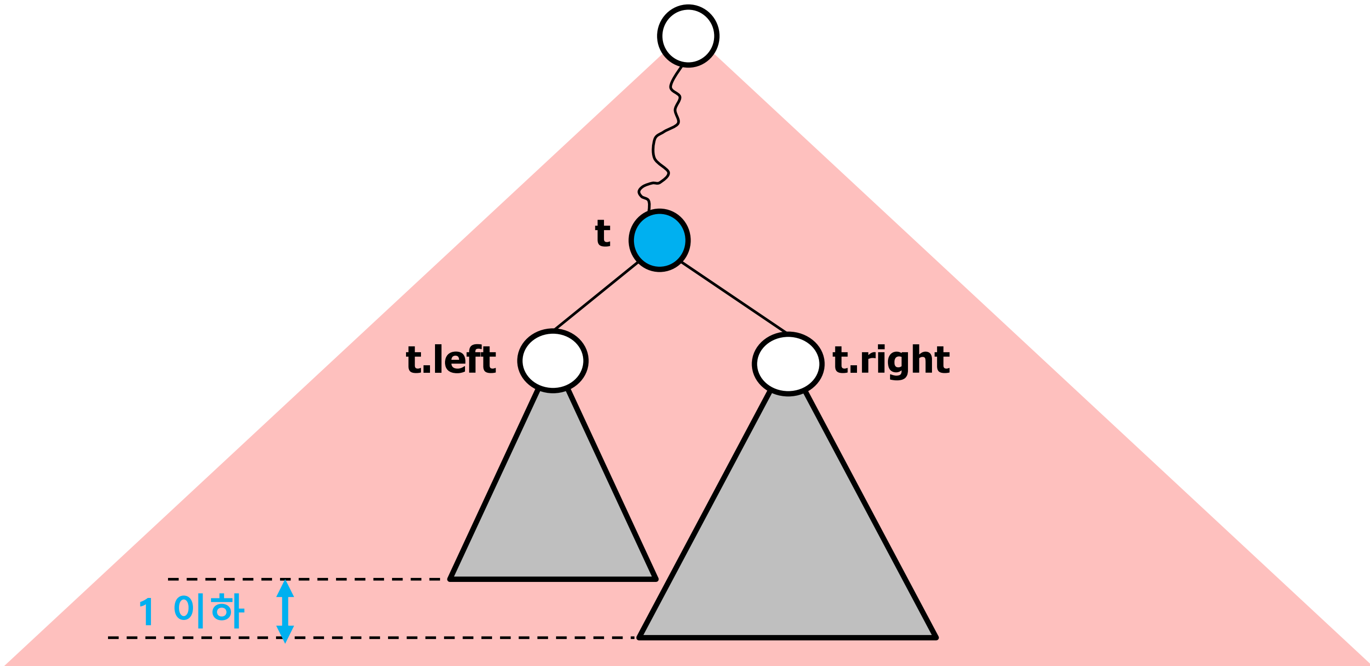
균형 탐색 트리

AVL 트리

AVL 트리 (1/9)

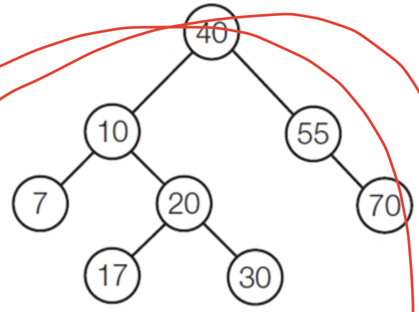
● AVL 트리

- 전체 트리의 구조가 균형이 맞도록 하는 트리
  - 모든 노드에 대해 좌 서브 트리의 높이(깊이)와 우 서브 트리의 높이의 차가 1을 넘지 않는다(즉, 트리 구조가 한쪽으로 쏠리는 것을 막을 수 있다).

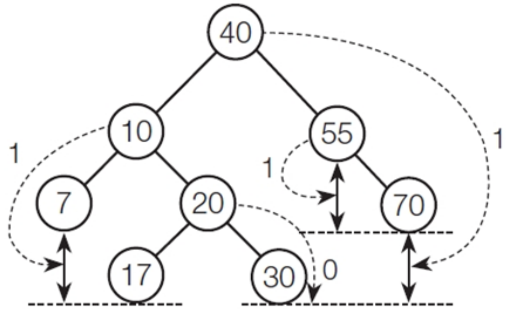


AVL 트리 (2/9)

- **AVL 트리**
  - **AVL 트리의 예**

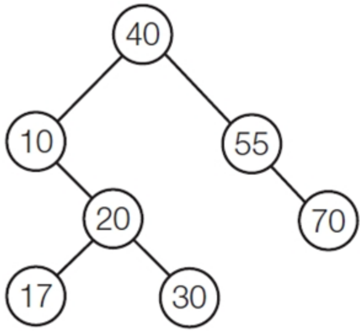


(a) AVL 트리의 예

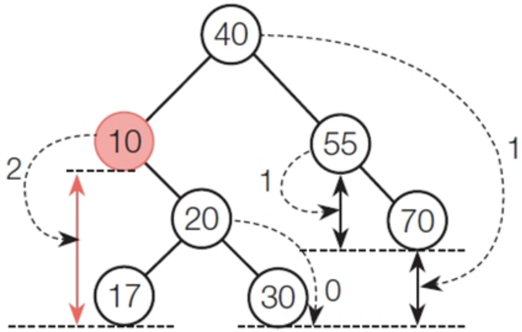


(b) 서브 트리의 높이 차

  - **AVL 트리가 아닌 예 #1**



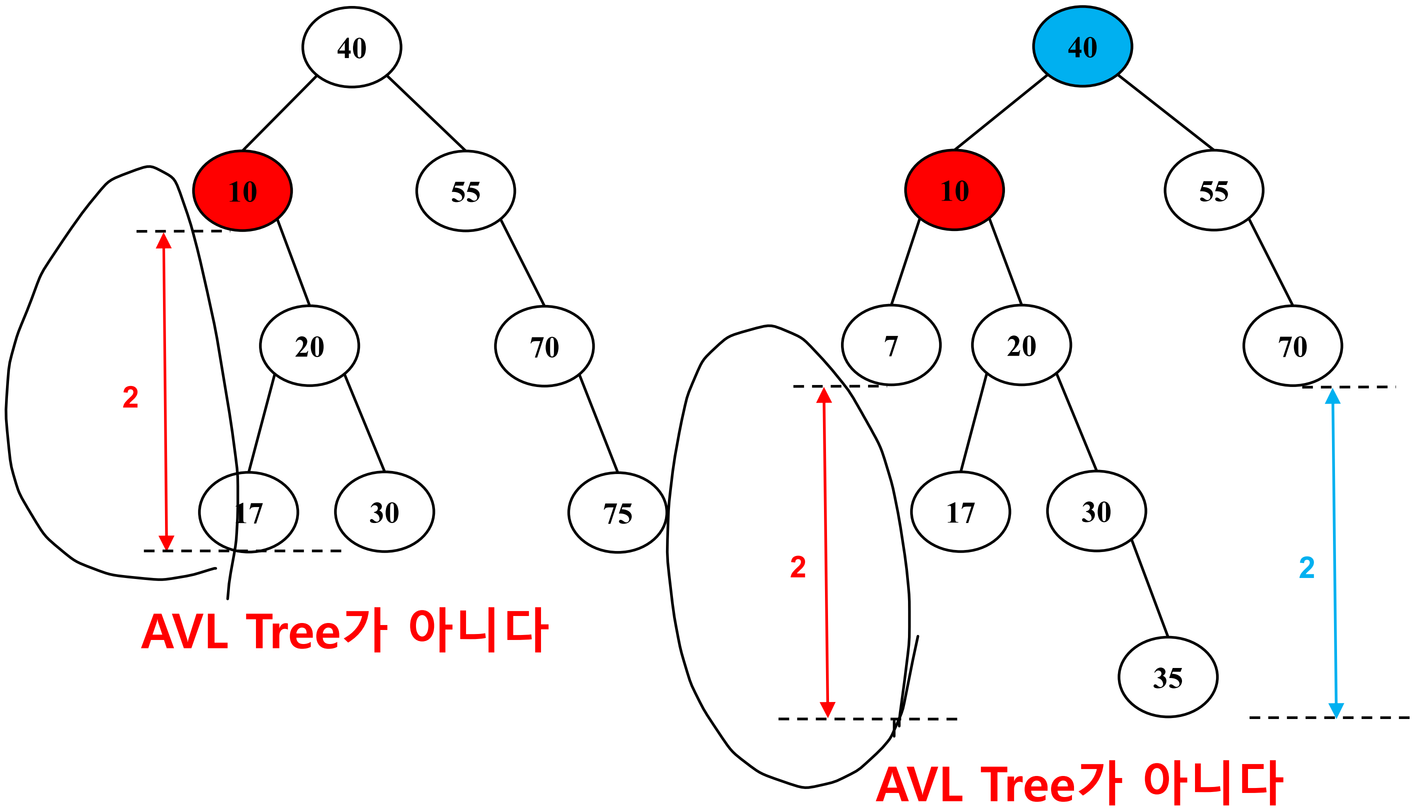
(a) AVL 트리가 아닌 예



(b) AVL 트리가 아닌 이유: 높이 차 2

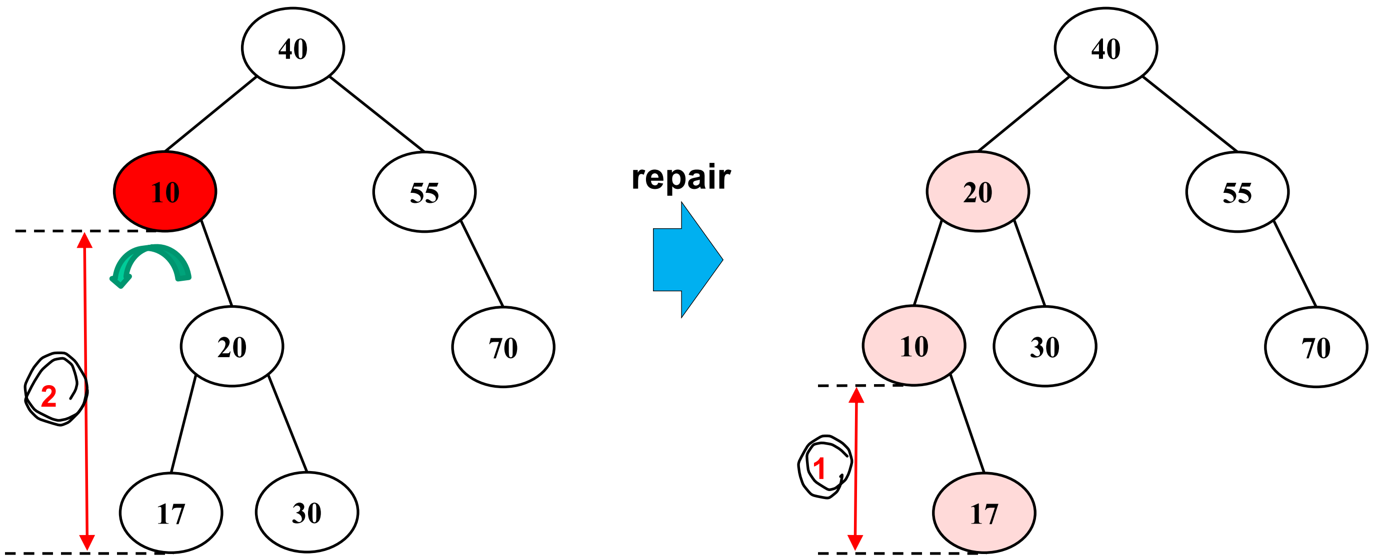
AVL 트리 (3/9)

- **AVL 트리**
  - AVL 트리가 아닌 예 #2



AVL 트리 (4/9)

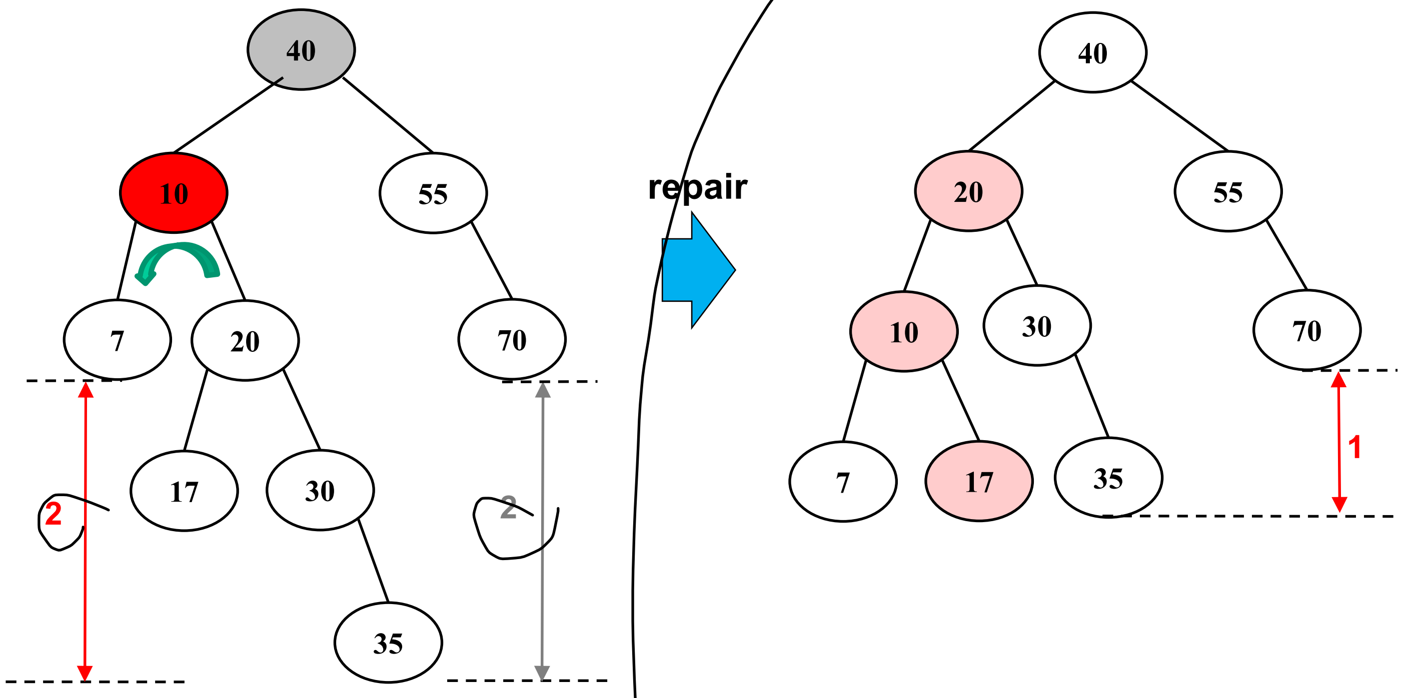
- **AVL 트리: 균형 맞추기**
  - **균형 맞추기: 좌회전**
    - 좌회전으로 불균형 해결



AVL Tree가 아닌 예 — 좌회전해서 AVL Tree로 수선됨

AVL 트리 (5/9)

- **AVL 트리:** 균형 맞추기
  - **균형 맞추기: 좌회전**
    - 좌회전으로 두 곳의 불균형 해결



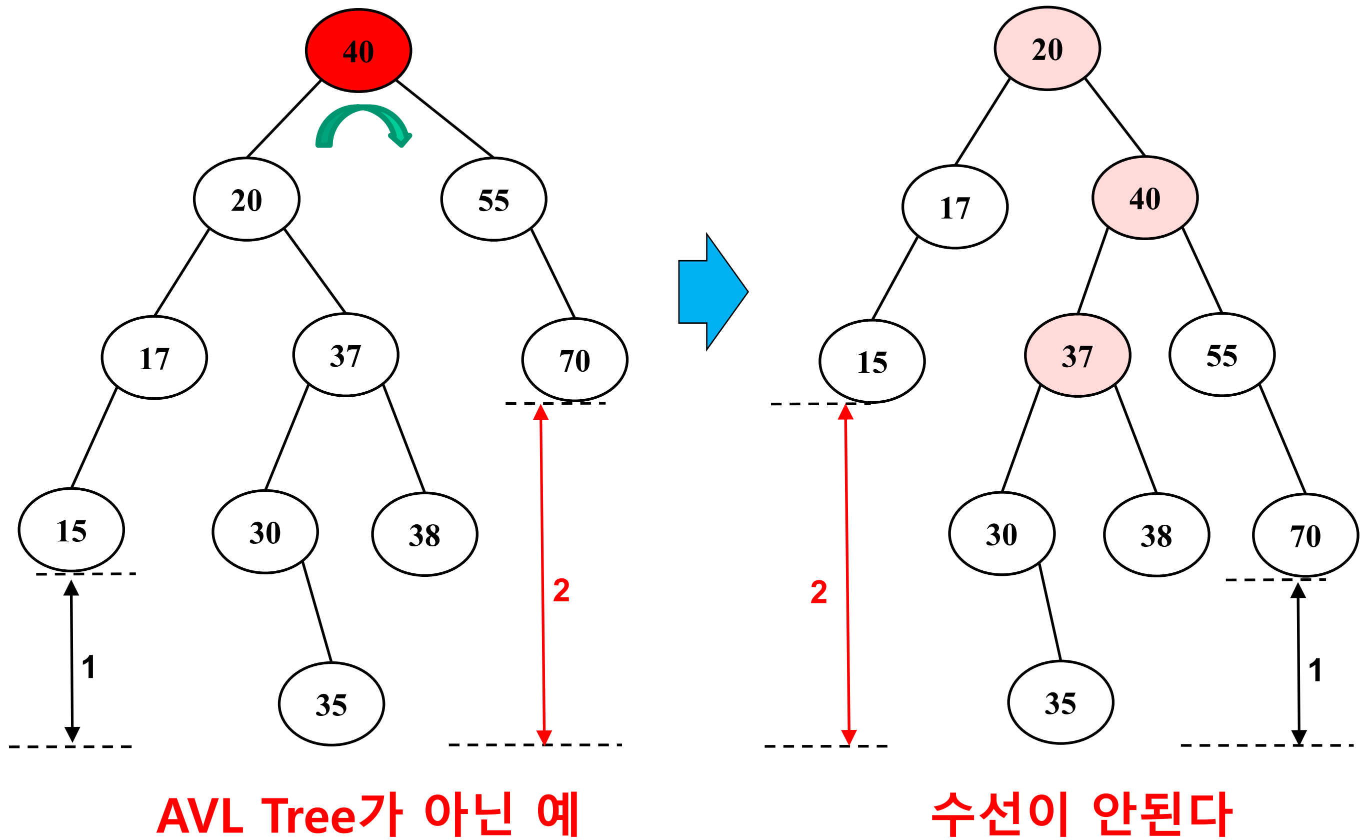
AVL Tree가 아닌 예

좌회전해서 AVL Tree로 수선됨

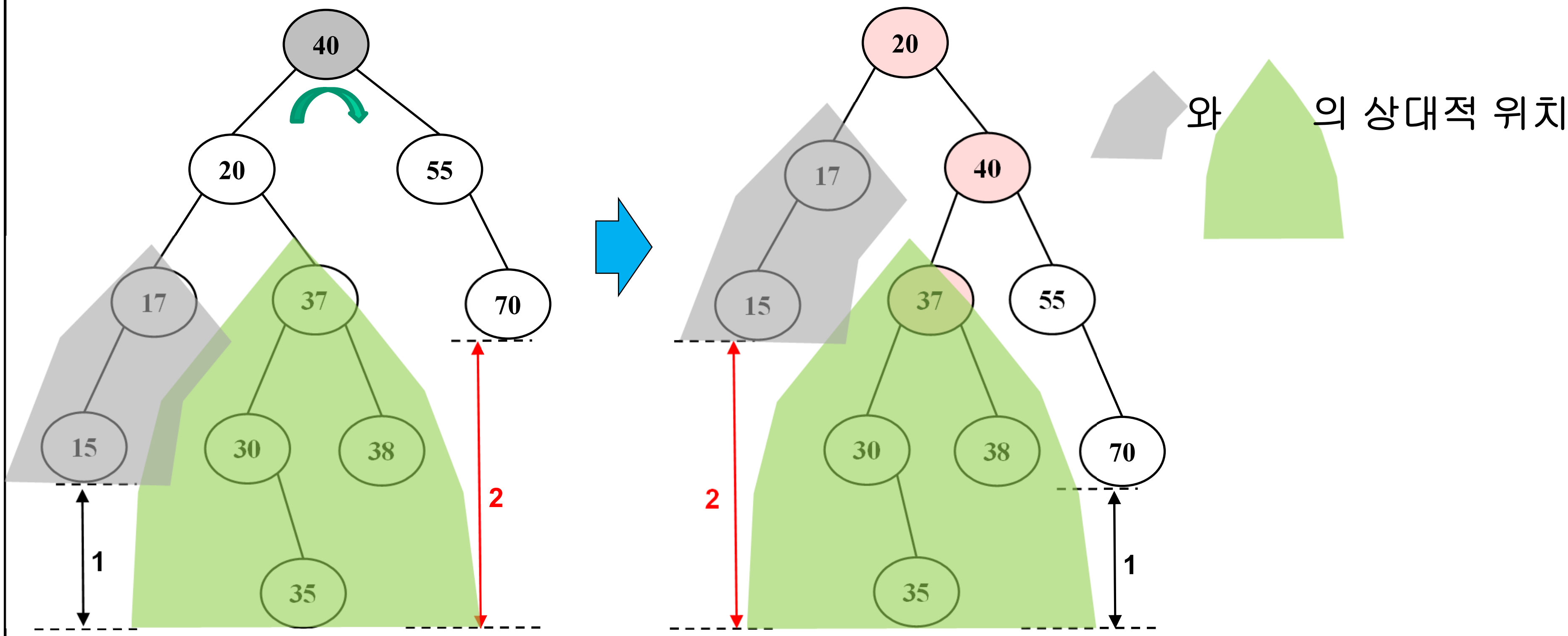
[이미지 출처: "IT CookBook, 쉽게 배우는 자료구조 with 파이썬", 문병로, 한빛아카데미, 2022.]

AVL 트리 (6/9)

- **AVL 트리: 균형 맞추기**
  - 균형 맞추기
    - 단순한 회전으로 해결이 안되는 경우



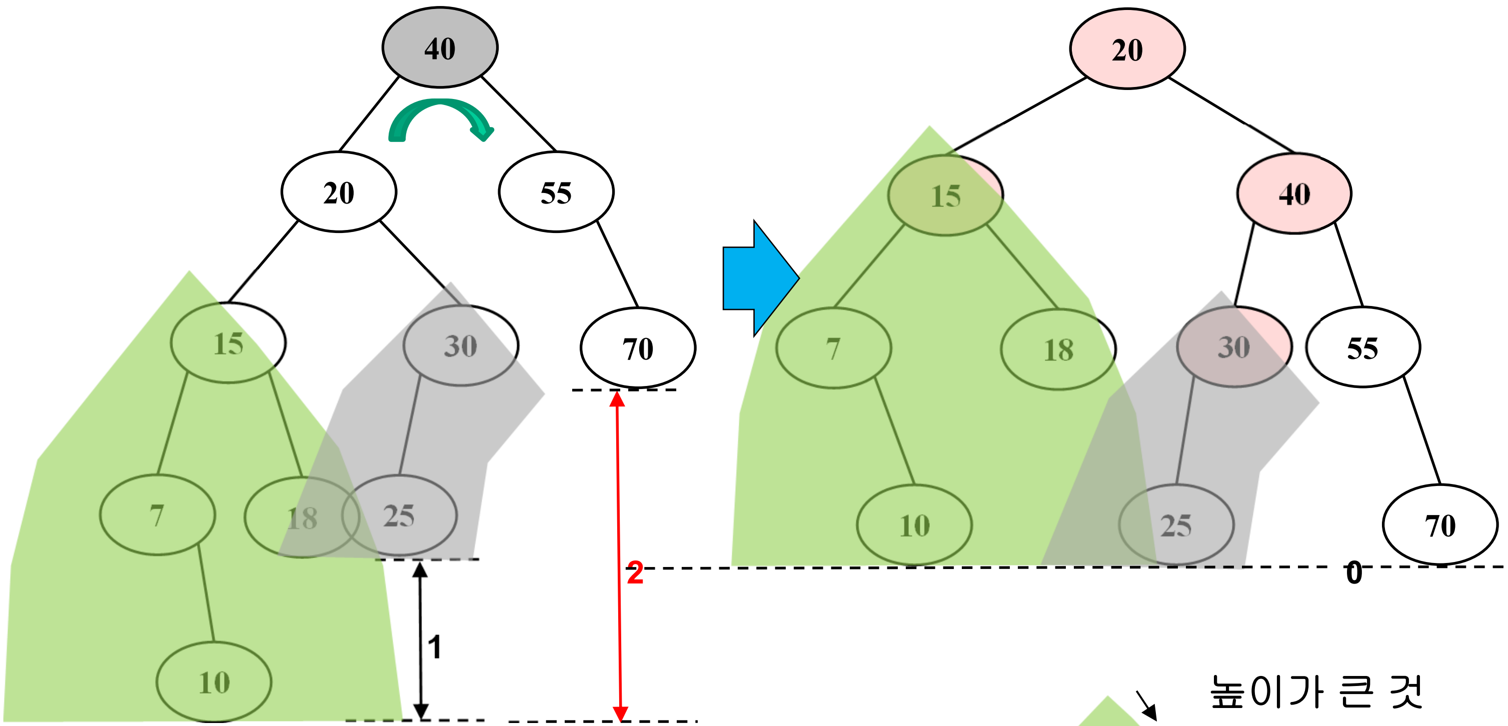
수선이 안 되는 이유



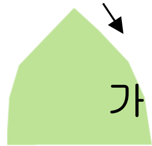
40
20
55
17
37
70
15
30
38
35
1
2
20
17
40
15
37
55
30
38
70
35
2
1
와
의 상대적 위치

수선의 예

한 번의 회전으로 균형이 해결되는 예



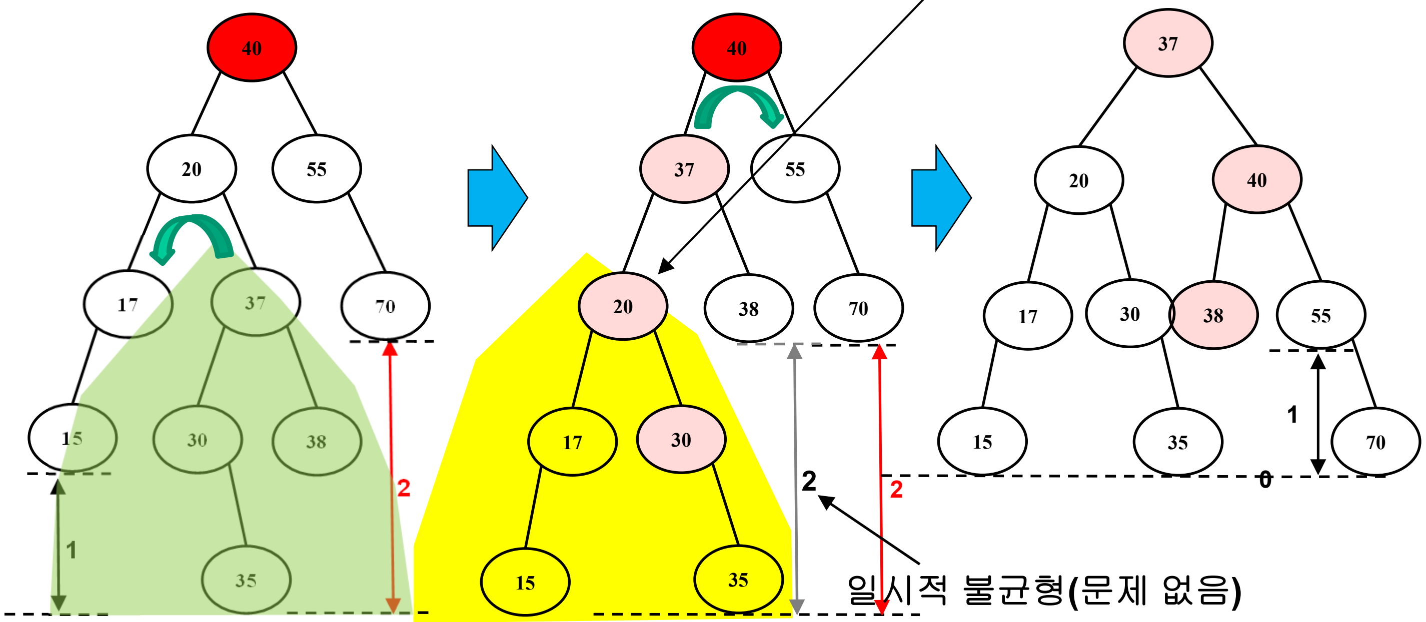
높이가 큰 것

앞 경우와의 차이:  가 왼쪽에 있다

표준화

좌회전 후 우회전으로 균형이 해결되는 예

좌회전 해서 서브트리의 <u>왼쪽</u>이 더 높게 만든 다음 우회전



(a) LR 타입이므로 좌회전 **(b) LL** 타입이므로 우회전 **(c)** 수선 완료

AVL 트리: 구성 및 구현 (1/6)

- **AVL 트리:** 구성 및 구현
  - 4가지 유형
    - LL : T.left.left 가 가장 깊음
    - LR : T.left.right 가 가장 깊음
    - RR : T.right.right 가 가장 깊음
    - RL : T.right.left 가 가장 깊음
  - 균형 맞추기
    - 좌회전(Left Rotation)
    - 우회전(Right Rotation)

알고리즘 11-3 AVL 트리 균형 잡기(스케치)

```
balanceAVL(t, type):
    switch type:
        case LL: 우회전(t)
        case LR: 좌회전(t.left)
                 balanceAVL(t, LL)
        case RR: 좌회전(t)
        case RL: 우회전(t.right)
                 balanceAVL(t, RR)
```

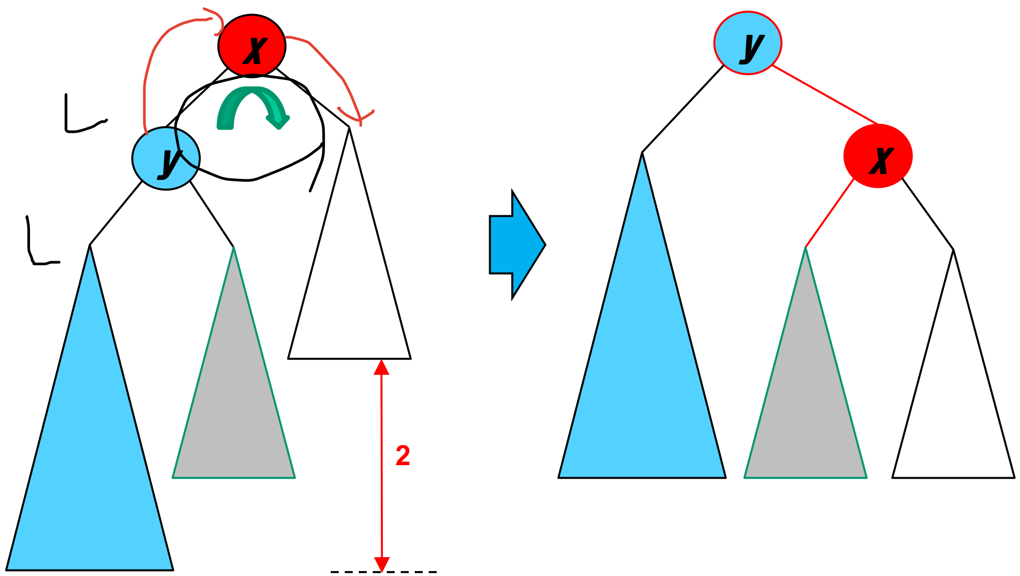
AVL 트리: 구성 및 구현 (2/6)

- **AVL 트리: 구성 및 구현**
  - 균형 맞추기: LL 유형

알고리즘 11-2 AVL 트리의 우회전

```
우회전(t): ◀ t: 회전의 중심 노드
    LChild ← t.left;
    LRChild ← LChild.right;
    LChild.right ← t;
    t.left ← LRChild;
    LChild.height ← max(LChild.right.height, LChild.left.height) + 1;
    t.height ← max(t.right.height, t.left.height) + 1;
```

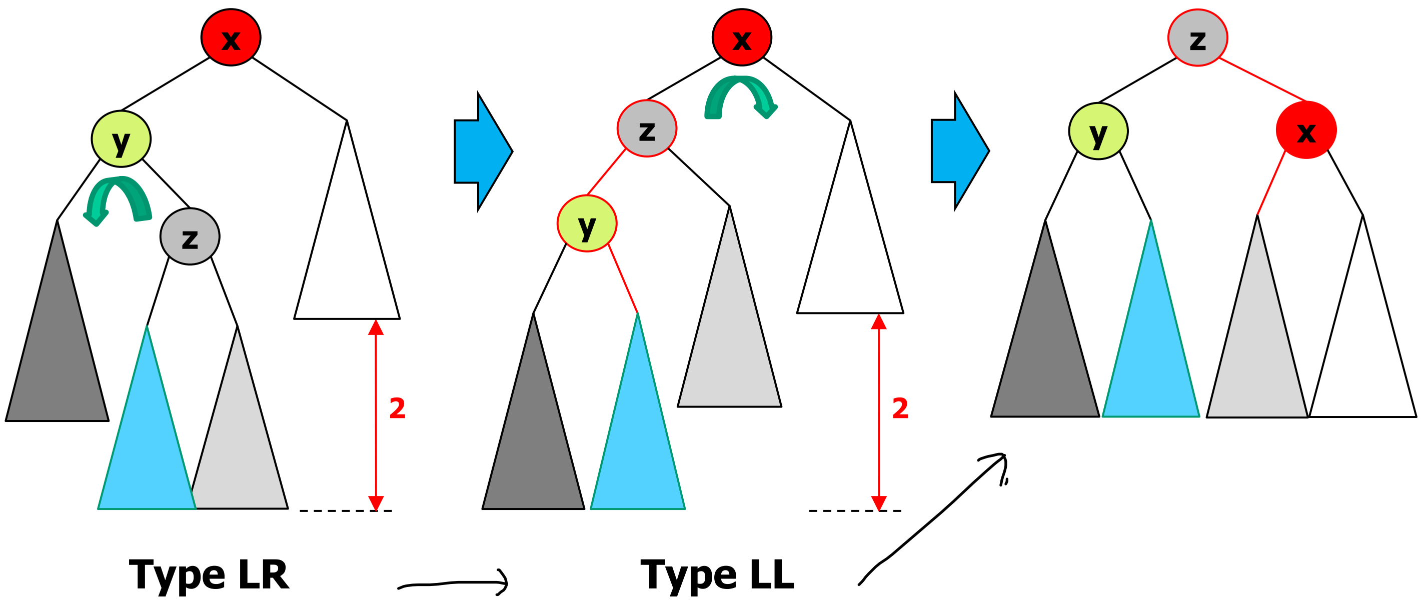
우회전(Right Rotation)



AVL 트리: 구성 및 구현 (3/6)

- **AVL 트리:** 구성 및 구현
  - 균형 맞추기: LR 유형

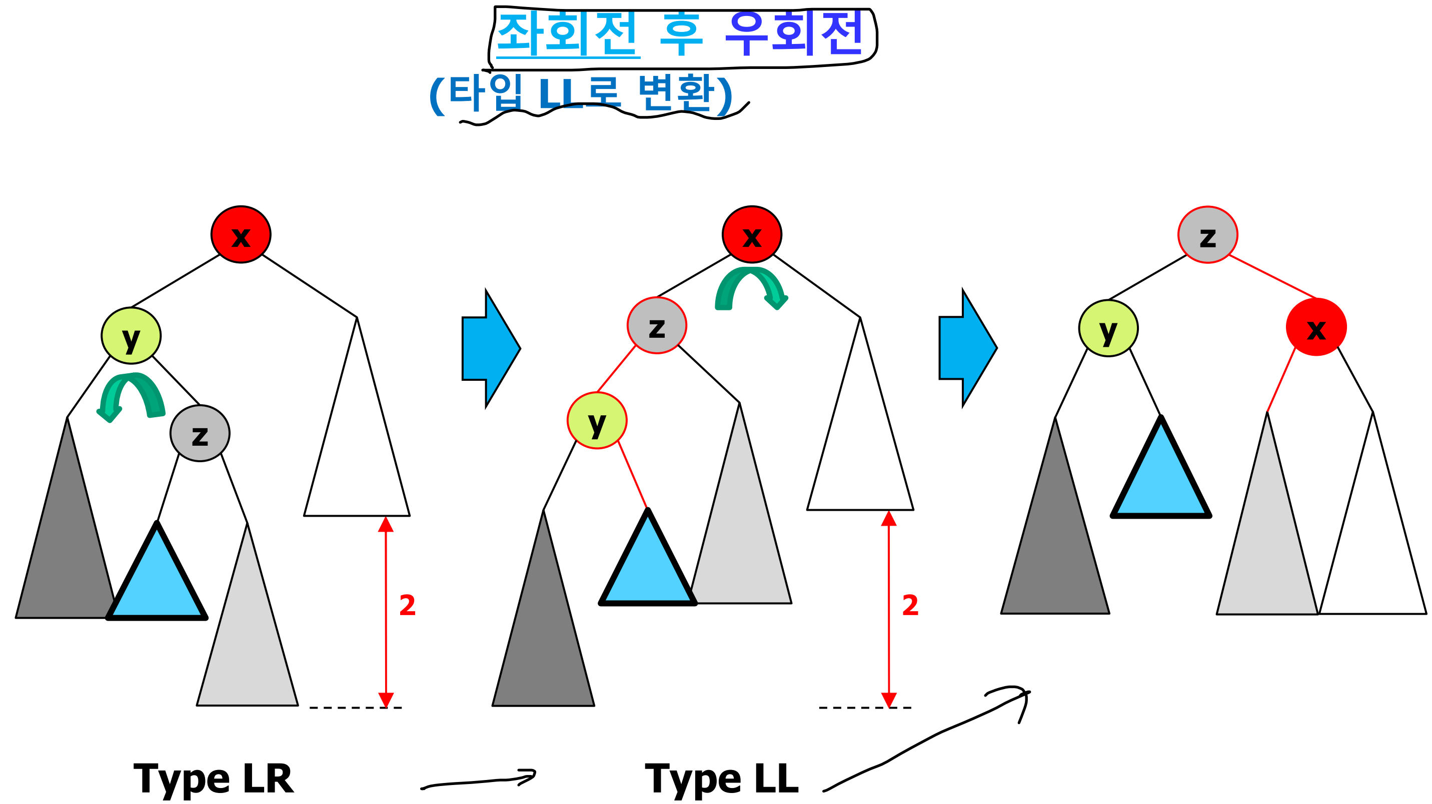
좌회전 후 우회전
(타입 LL로 변환)



[이미지 출처: "IT CookBook, 쉽게 배우는 자료구조 with 파이썬", 문병로, 한빛아카데미, 2022.]

AVL 트리: 구성 및 구현 (4/6)

- **AVL 트리: 구성 및 구현**
  - 균형 맞추기: LR 유형



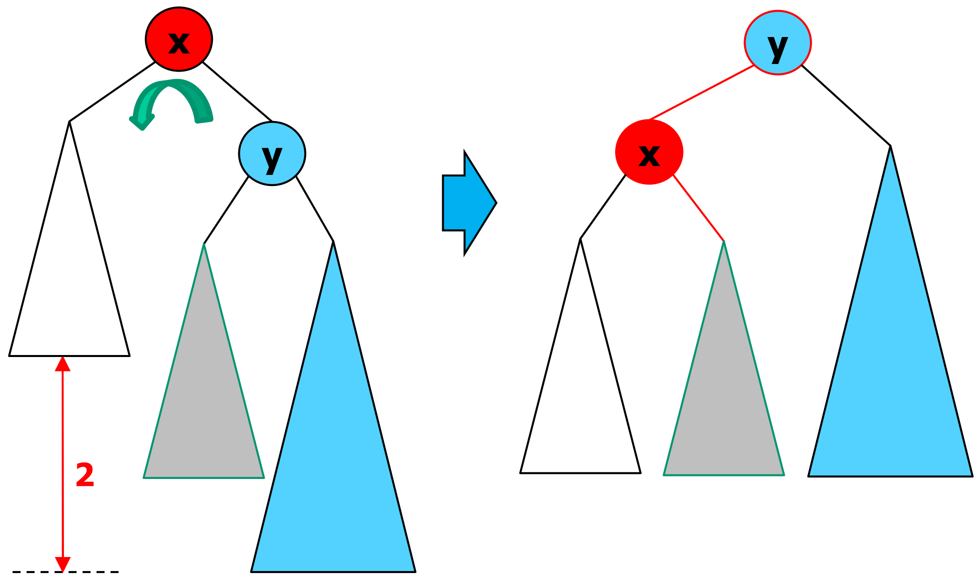
AVL 트리: 구성 및 구현 (5/6)

- **AVL 트리: 구성 및 구현**
  - 균형 맞추기: RR 유형

알고리즘 11-1 AVL 트리의 좌회전

```
좌회전(t): ◀ t: 회전의 중심 노드
    RChild ← t.right
    RLChild ← RChild.left
    RChild.left ← t
    t.right ← RLChild
    RChild.height ← max(RChild.right.height, RChild.left.height) + 1
    t.height ← max(t.right.height, t.left.height) + 1
```

좌회전(Left Rotation)



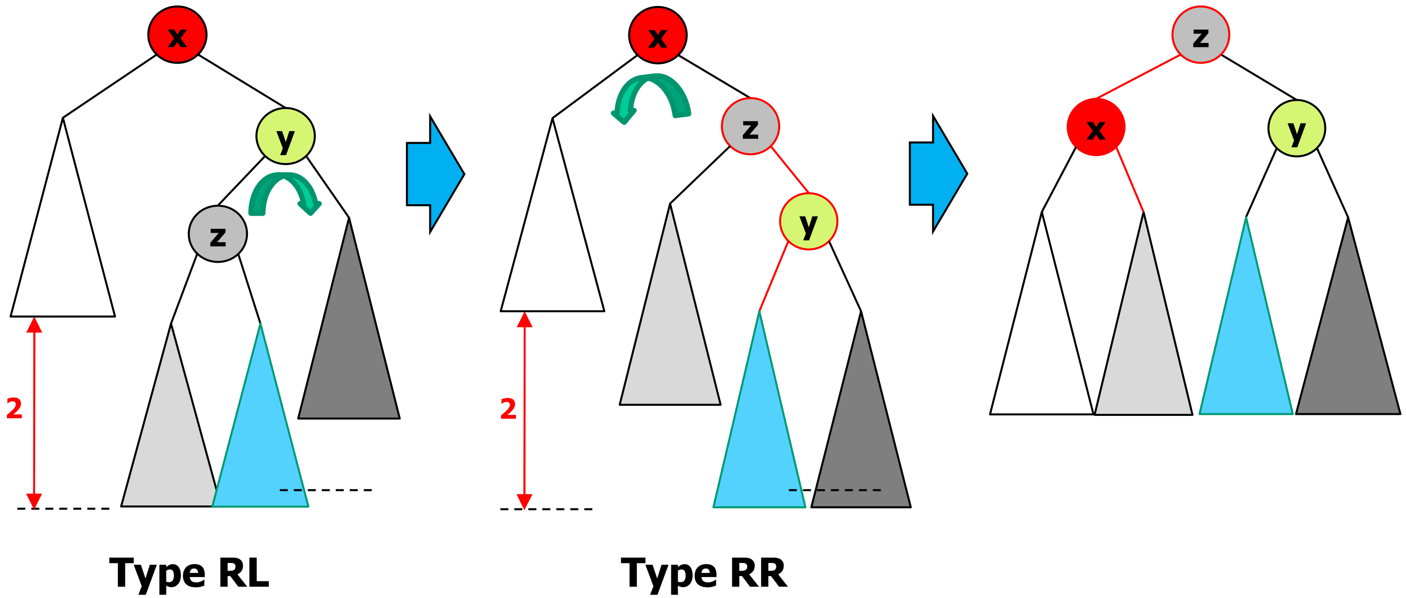
AVL 트리: 구성 및 구현 (6/6)

- **AVL 트리: 구성 및 구현**

삽입 삭제시 매번 정렬

  - 균형 맞추기: **RL 유형**

우회전 후 좌회전
(타입 RR로 변환)



참고

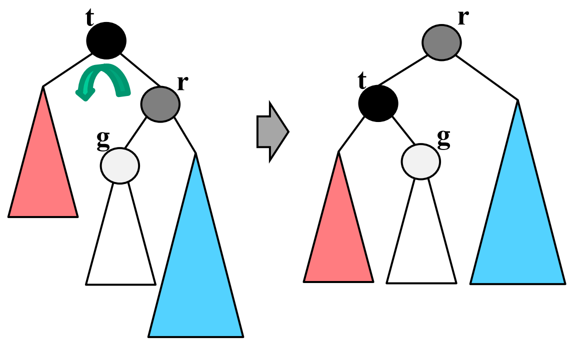
좌회전 Left Rotation

```
r ← t.right
g ← r.left
r.left ← t
t.right ← g
t.height ← max(t.right.height, t.left.height)
+ 1
r.height ← max(r.right.height, r.left.height)
+ 1
```

이 코드는 문제를 일으킬 수 있다

t.right, **t.left**, **r.right**가 **null**이면 에러 발생

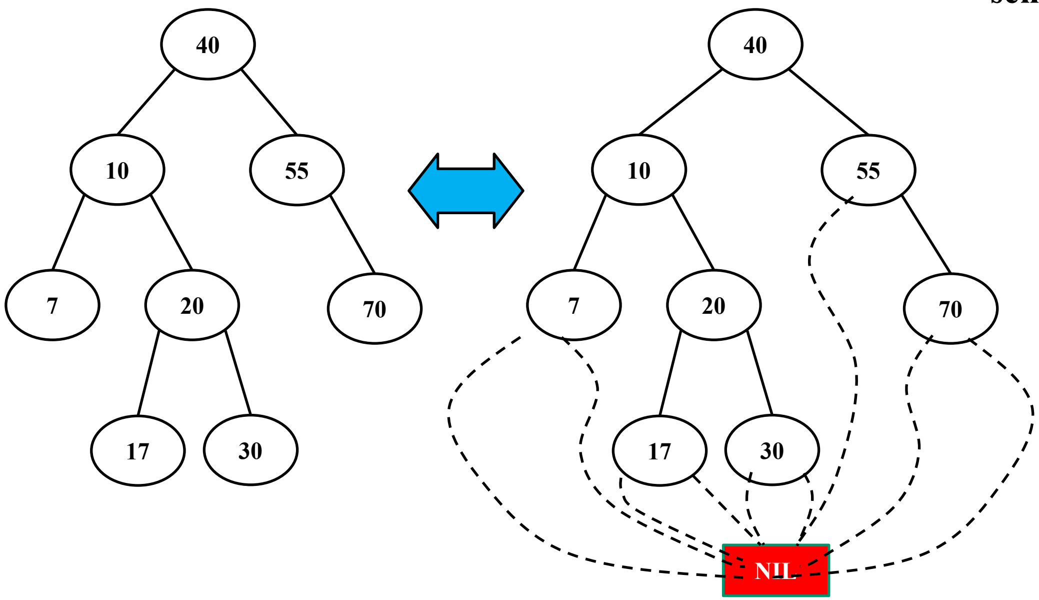
통상적인 방법으로 해결하려면
코드가 지저분해진다



유용한 수단: 경계 노드Sentinel

… AVLNode NIL = new AVLNode(null, null, null, 0); // sentinel

height 0



NIL이면 에러 안난다

…
t.height ← max(t.right.height, t.left.height) + 1
r.height ← max(r.right.height, r.left.height) + 1

null 대신 NIL 노드를 사용
NIL 노드는 하나만 만들면 된다
null도 정상 노드처럼 취급할 수 있다
→ 앞 페이지의 코드를 그대로 쓸 수 있다

수선 예

수선이 끝까지 올라갈 수도 있다

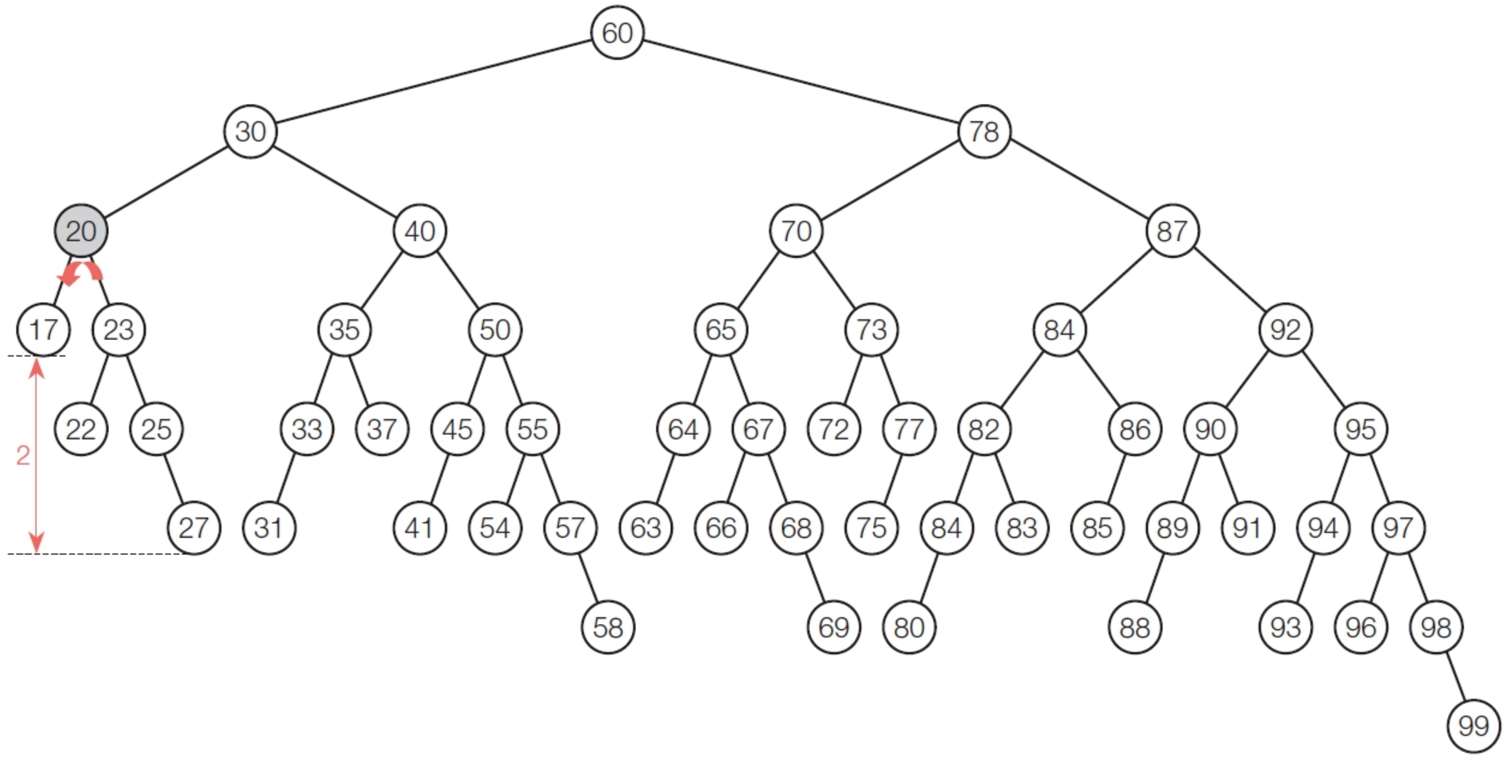


그림 11-18 RR 타입으로 균형이 깨진 예

수선 예

수선이 끝까지 올라갈 수도 있다

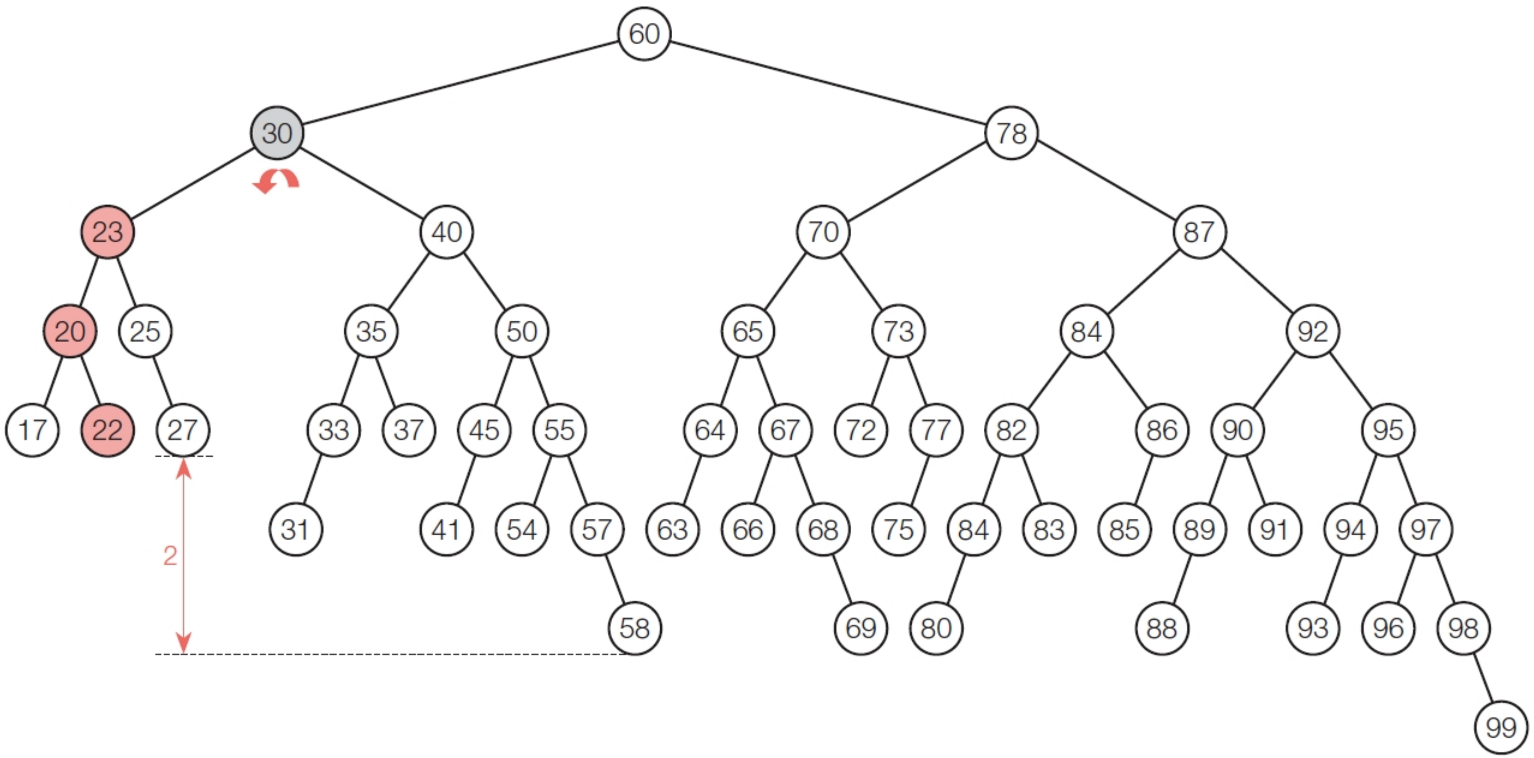


그림 11-19 한 번의 좌회전 후 상위 서브 트리에서 새롭게 균형이 깨진 상태

수선 예

수선이 끝까지 올라갈 수도 있다

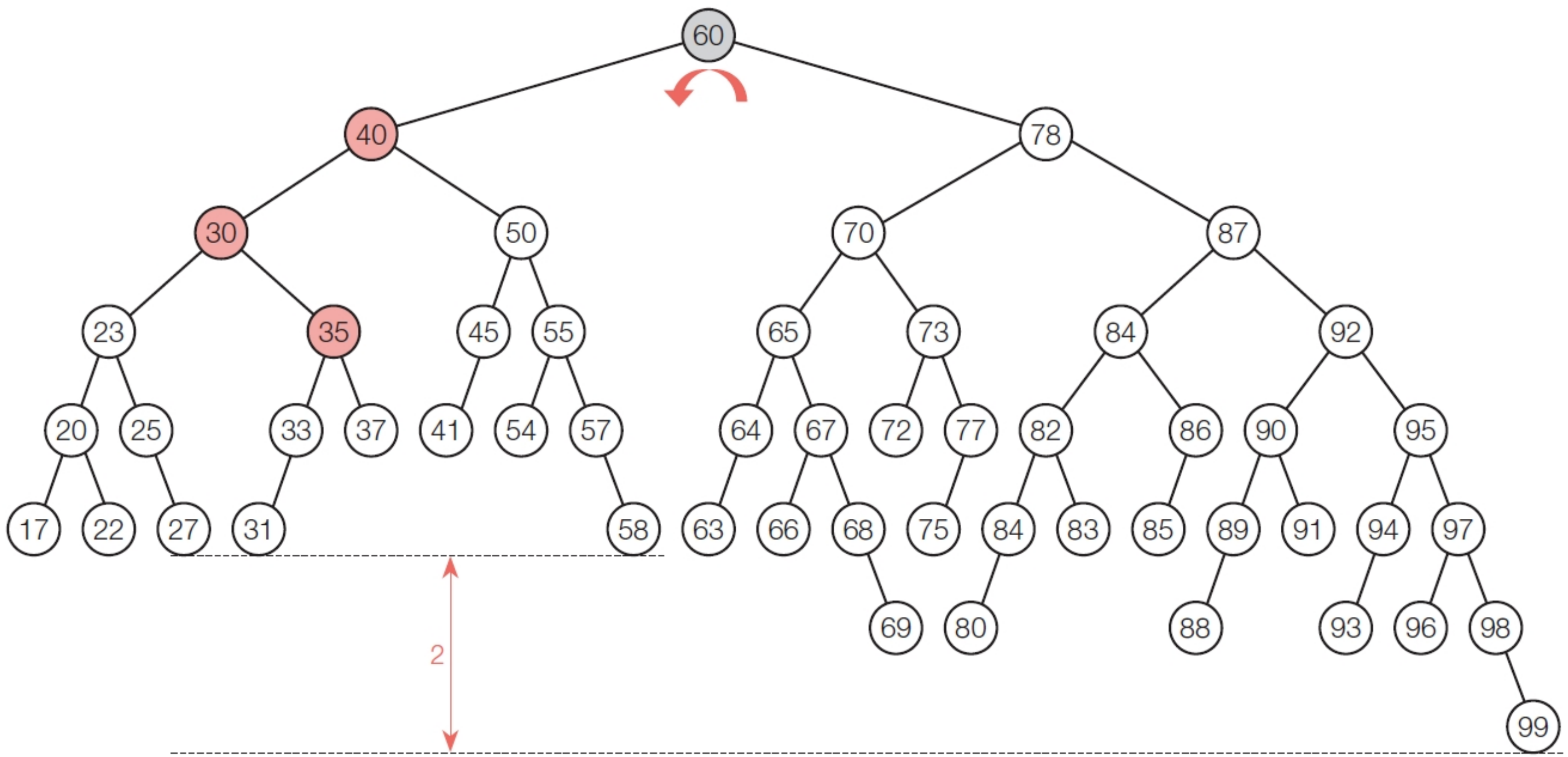


그림 11-20 두 번의 좌회전 후 상위 서브 트리에서 새롭게 균형이 깨진 상태

수선 예

수선이 끝까지 올라갈 수도 있다

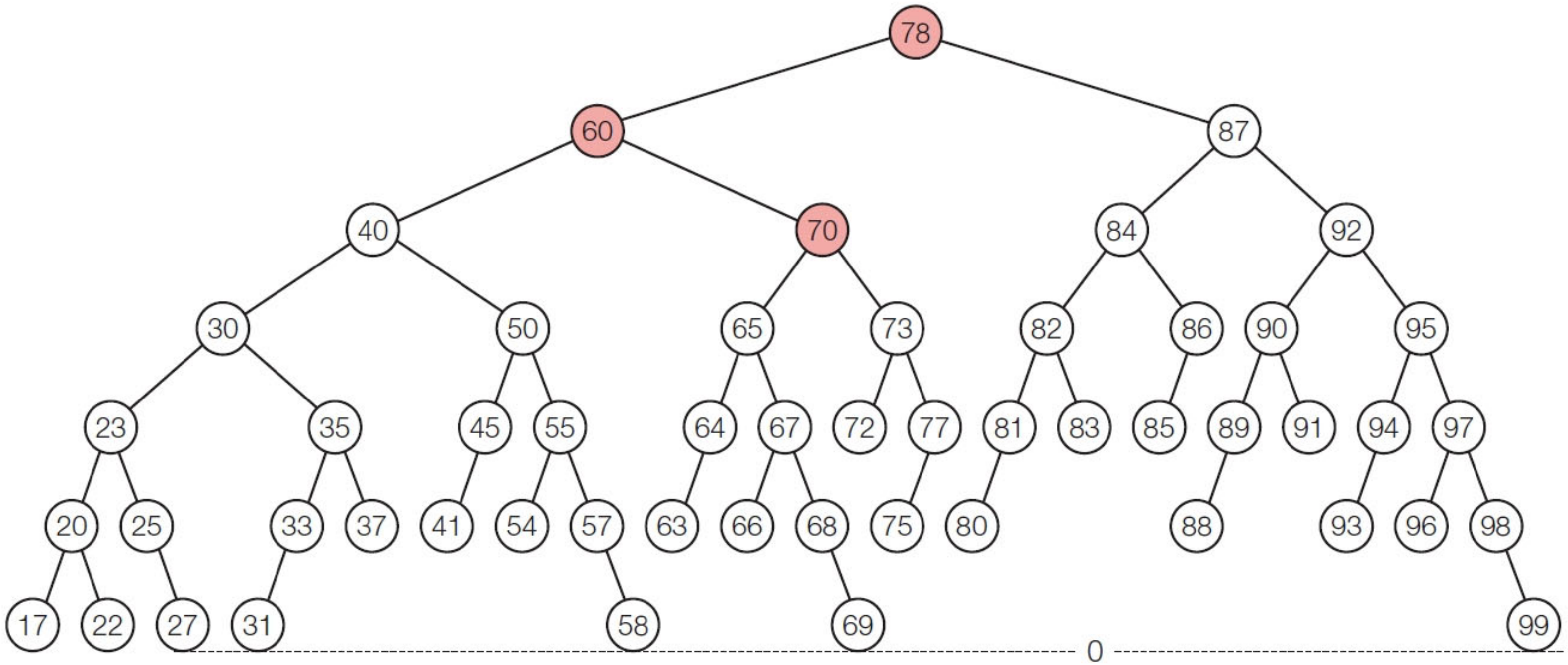


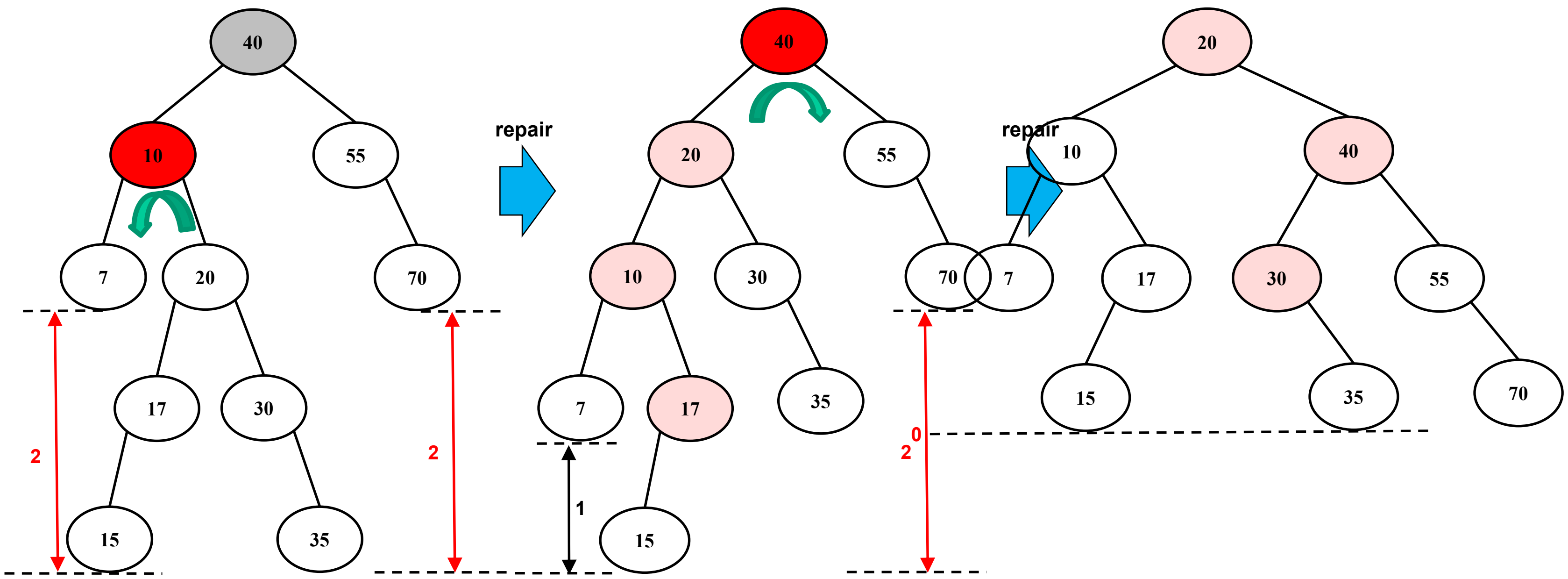
그림 11-21 세 번의 좌회전 후 균형이 해결된 상태

생각해보기

Example

AVL Tree에서
이런 상황이 일어날 수 있을까?

답: **AVL Tree**에서 왼쪽과 같은 상황은 존재할 수 없다.
이유를 생각해봄으로써 **AVL Tree**의 메커니즘에 대한 **insight**를 강화한다.

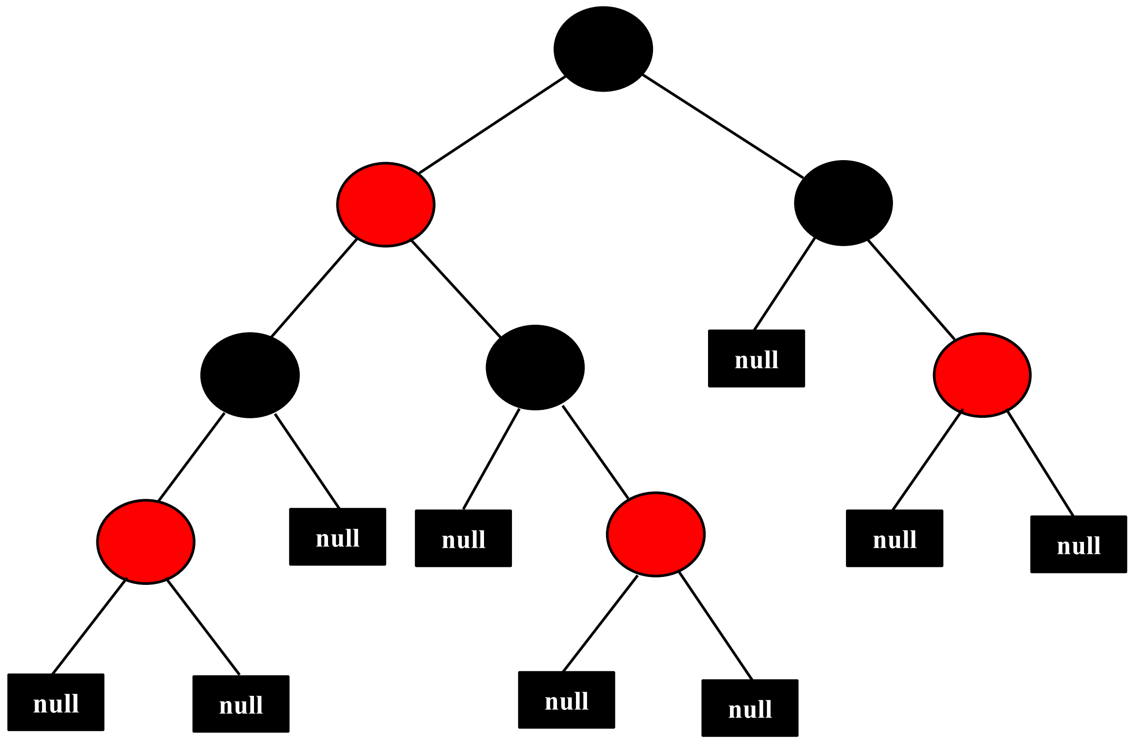


균형 탐색 트리

레드-블랙 트리

레드-블랙 트리

- **레드-블랙 트리**(Red-Black Tree, RB Tree)



레드-블랙 트리Red-Black Tree, RB Tree

레드-블랙 트리

- 모든 null 자리에 리프 노드를 둔다
- RB-Tree에서 리프 노드는 이 null 리프를 말한다
- 모든 노드는 레드 또는 블랙의 색을 갖는다

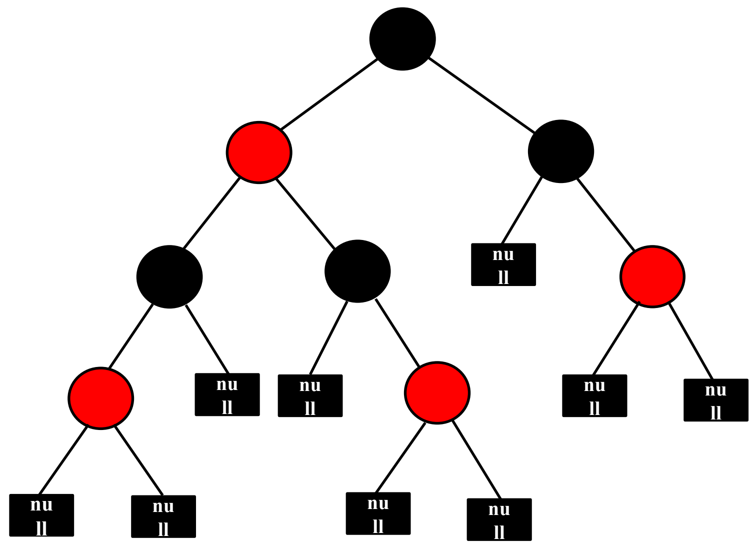
레드-블랙 트리 특성

① 루트는 블랙이다.

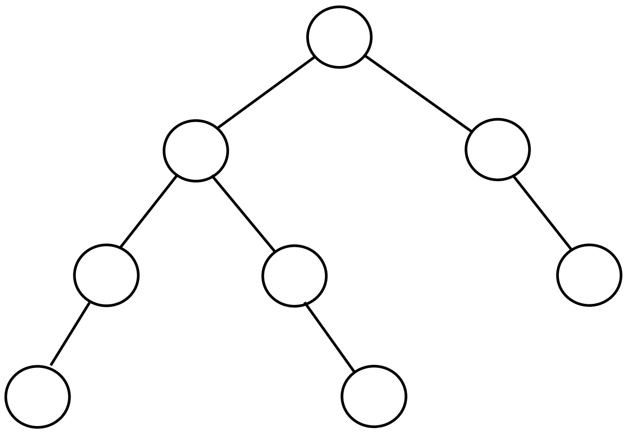
② 모든 리프 노드는 블랙이다.

③ 루트로부터 임의의 리프 노드에 이르는 경로 상에 레드 노드 두 개가 연속으로 출현하지 못한다.

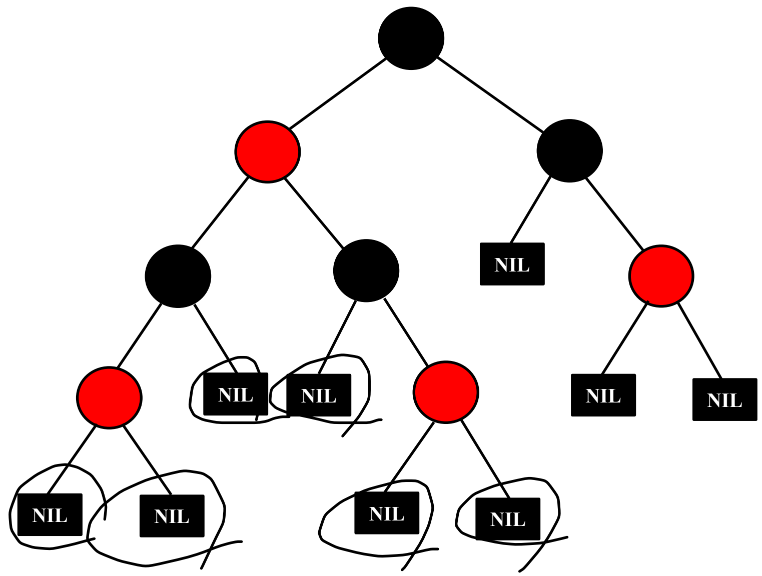
④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수(black height)는 모두 같다.



RB 트리의 예



(a) 이진 검색 트리의 예



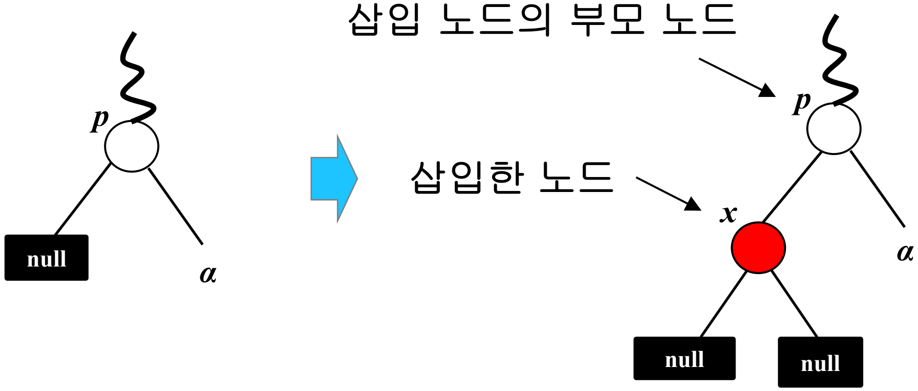
(b) (a)를 RB-트리로 만든 예

구현 시에

null 은 AVL-트리에서 처럼 sentinel NIL을 레퍼런스 하면 효과적이다

RB 트리의 삽입

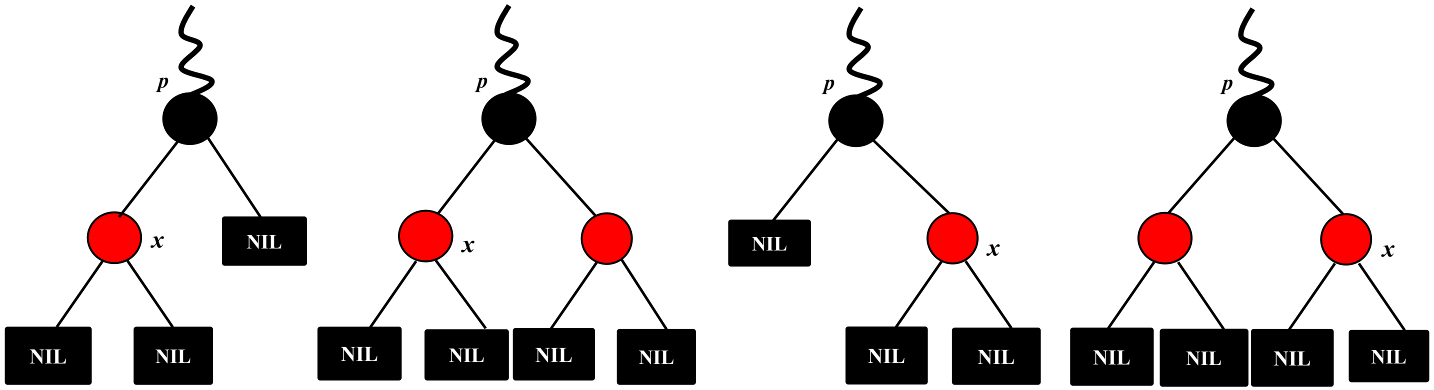
- 일반적인 BST의 삽입 작업 후,
  삽입 노드에 **레드**를 칠하고,
  삽입한 노드의 좌우에 null 리프를 달아준다



삽입 직후의 상황:
p가 블랙 또는 **레드**다

RB 트리의 삽입

- 삽입 직후의 상황: x의 부모 p가 **블랙** 또는 **레드**다
- **If** p가 **블랙**
  - RB 특성 다 만족. 완료!

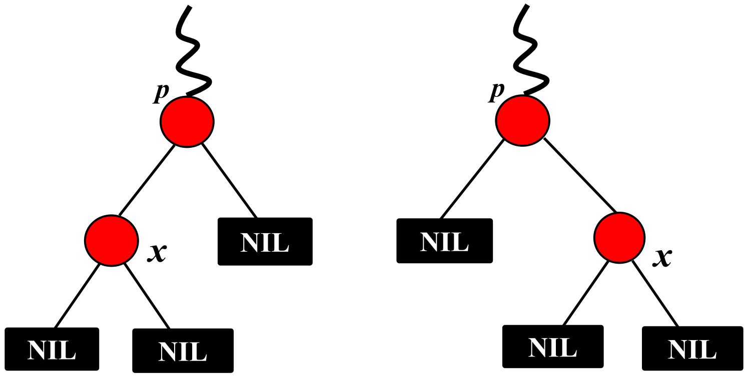


가능한 모양은 이 4가지 뿐이다

RB 트리의 삽입

p가 레드

- If p가 레드
  - RB 특성 ③이 깨졌다 → 수선 (다음 페이지 이후)

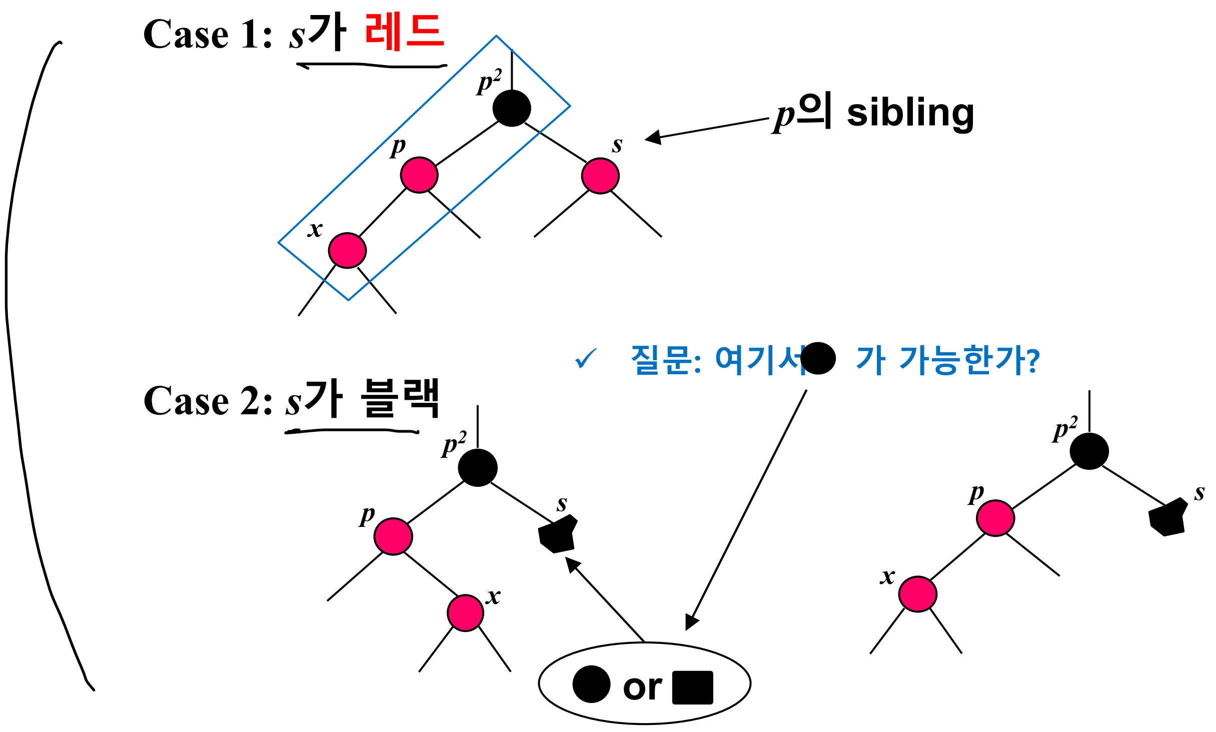


가능한 모양은 이 2가지 뿐이다

RB 트리의 삽입

p가 레드(수선)

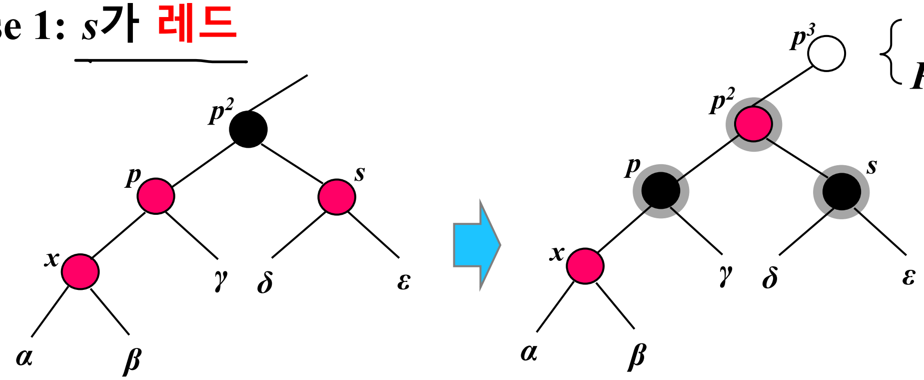
- If p가 레드
  - p의 형제sibling 노드 s에 따라 두 가지로 나눈다



RB 트리의 삽입

p가 레드(수선)

Case 1: s가 레드



P^3가 블랙이면 완료

P^3가 레드이면 p^2가 새 x : Recursive!

: 색이 바뀐 노드

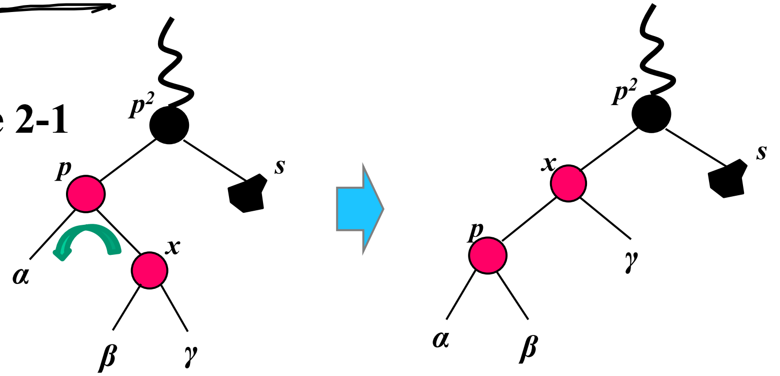
(p와 s를 블랙으로 바꾸고,
p^2을 레드로 바꾼다.)

RB 트리의 삽입

p가 레드(수선)

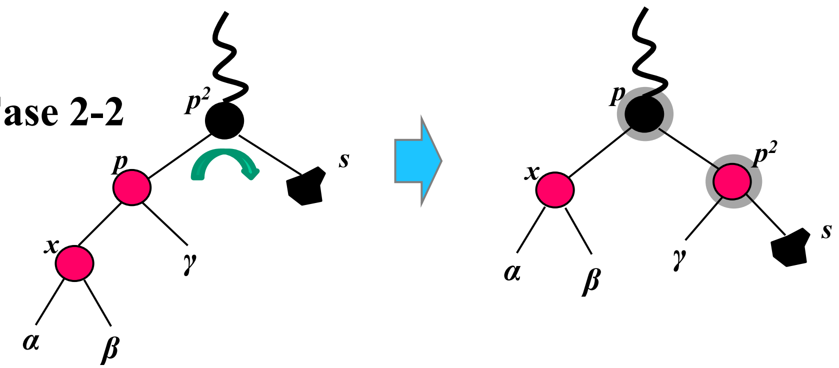
Case 2: s가 블랙

Case 2-1



Case 2-2로 변환

Case 2-2

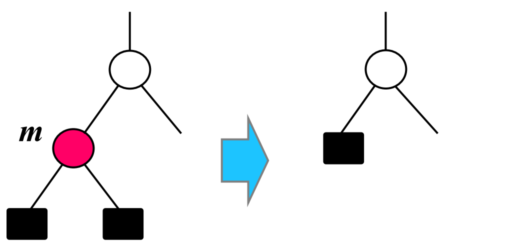


우회전하고, p와 p^2의 색을 바꾼다.
수선 끝.

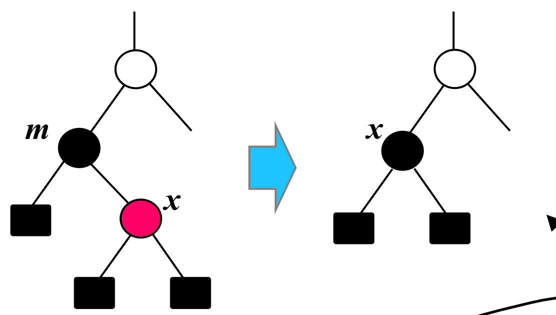
RB 트리의 삭제

- BST의 삭제 작업 중 Case 1과 2만 고려하면 됨

  ✓ 질문: Case 3은 왜 고려하지 않아도 되는가?



삭제 노드(m)가 레드이면 문제없다
(m은 반드시 자식이 없다)

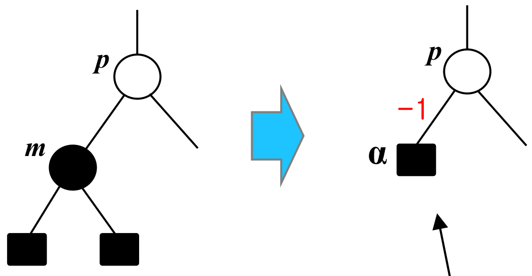


m 삭제 후
x의 색을 블랙으로 바꾸어준다

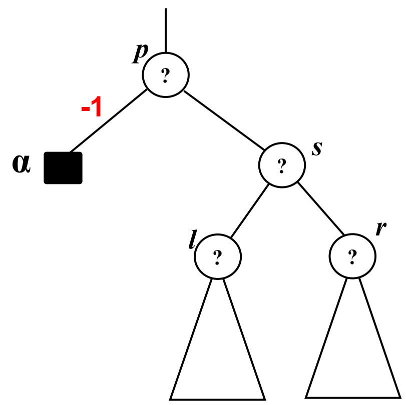
삭제 노드가 블랙이라도
m이 자식을 가지면(유일하고 반드시 레드다) 문제없다

RB 트리의 삭제

삭제 노드(m)가 블랙이고
자식이 없을 때만 문제 발생



RB 특성 ④가 깨진다



x의 주변 상황에 따라 처리 방법이 달라진다

이 강좌에서는 여기까지만.

시간 복잡도

AVL 트리와 RB 트리 모두
검색, 삽입, 삭제에 $O(\log n)$ 시간이 보장된다

✓ 생각해 보기: 위의 성질이 왜 만족되는지 생각해보자

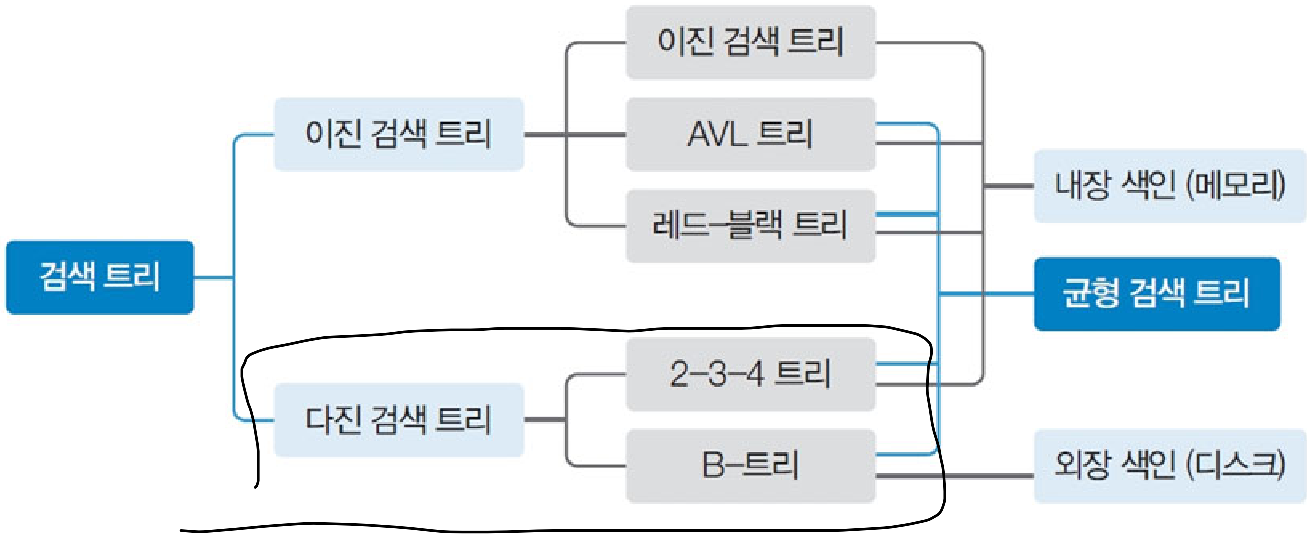
직관적으로 생각할 것

균형 탐색 트리

B 트리

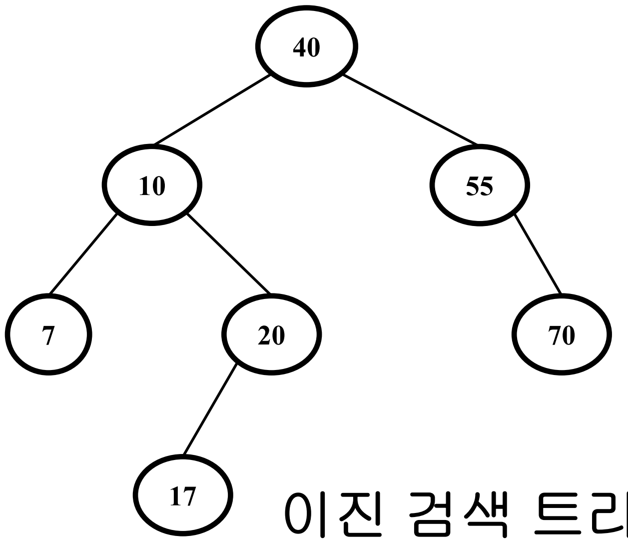
균형 탐색 트리

- **균형 탐색 트리(Binary Search Tree)**
  - **다진 탐색 트리: 2-3-4트리, B-트리**



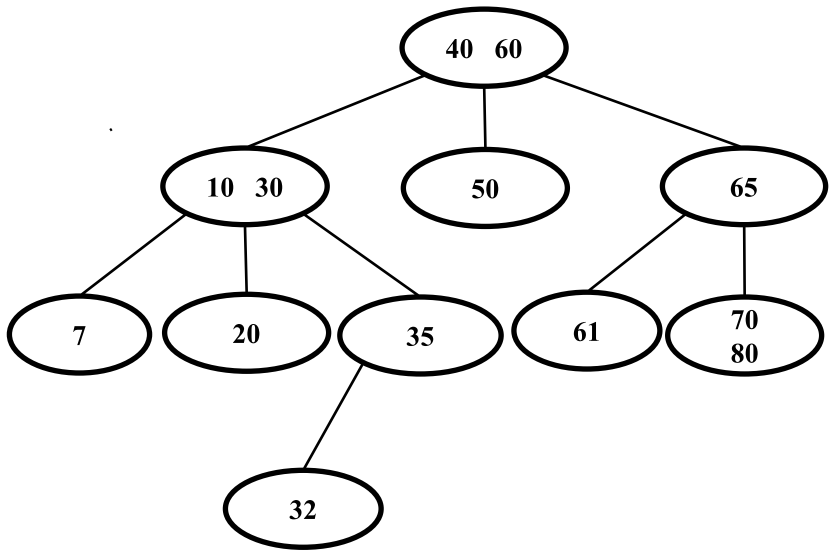
[이미지 출처: "IT CookBook, 쉽게 배우는 자료구조 with 파이썬", 문병로, 한빛아카데미, 2022.]

K-진 검색 트리



이진 검색 트리

K=2, 최대 2개 분기



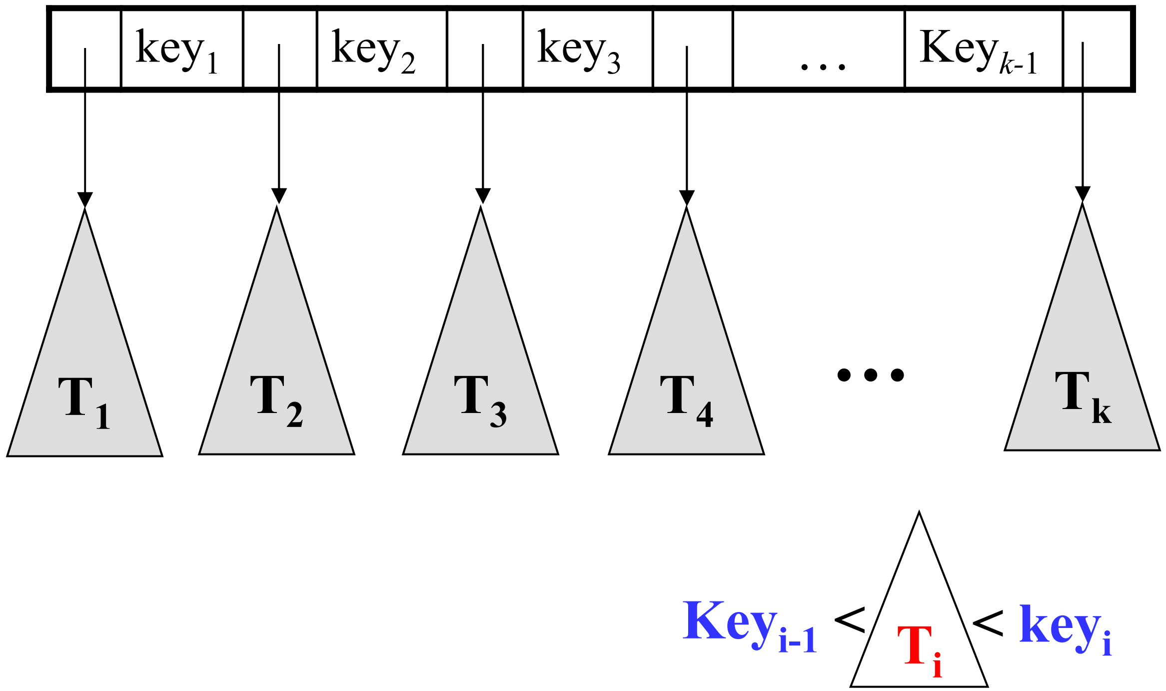
삼진 검색 트리

K=3, 최대 3개 분기

B-Tree의 환경

- 디스크의 접근 단위는 블록(페이지)
- 디스크에 한 번 접근하는 시간은 수십만 명령어의 처리 시간과 맞먹는다
- 검색 트리가 디스크에 저장되어 있다면
  트리의 높이를 최소화하는 것이 유리하다.
- B-트리는 K-진 검색 트리가 균형을 유지하도록 하여
  최악의 경우 디스크 접근 횟수를 줄인 것이다

K-진 검색 트리

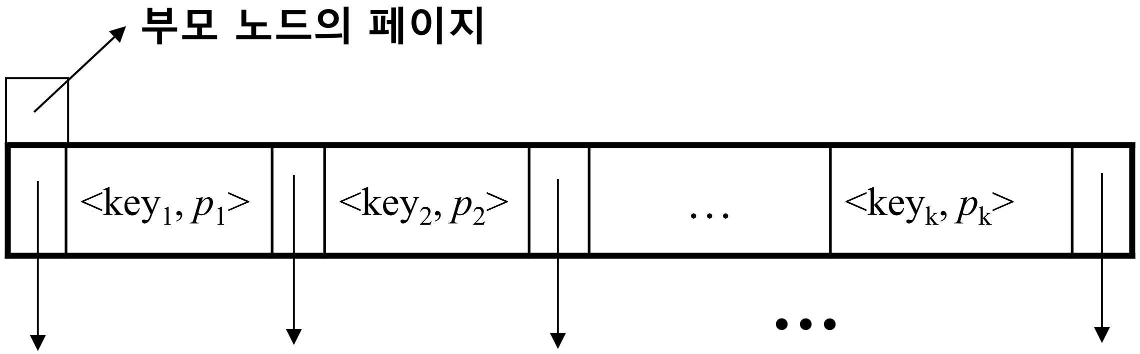


B-Tree의 성질

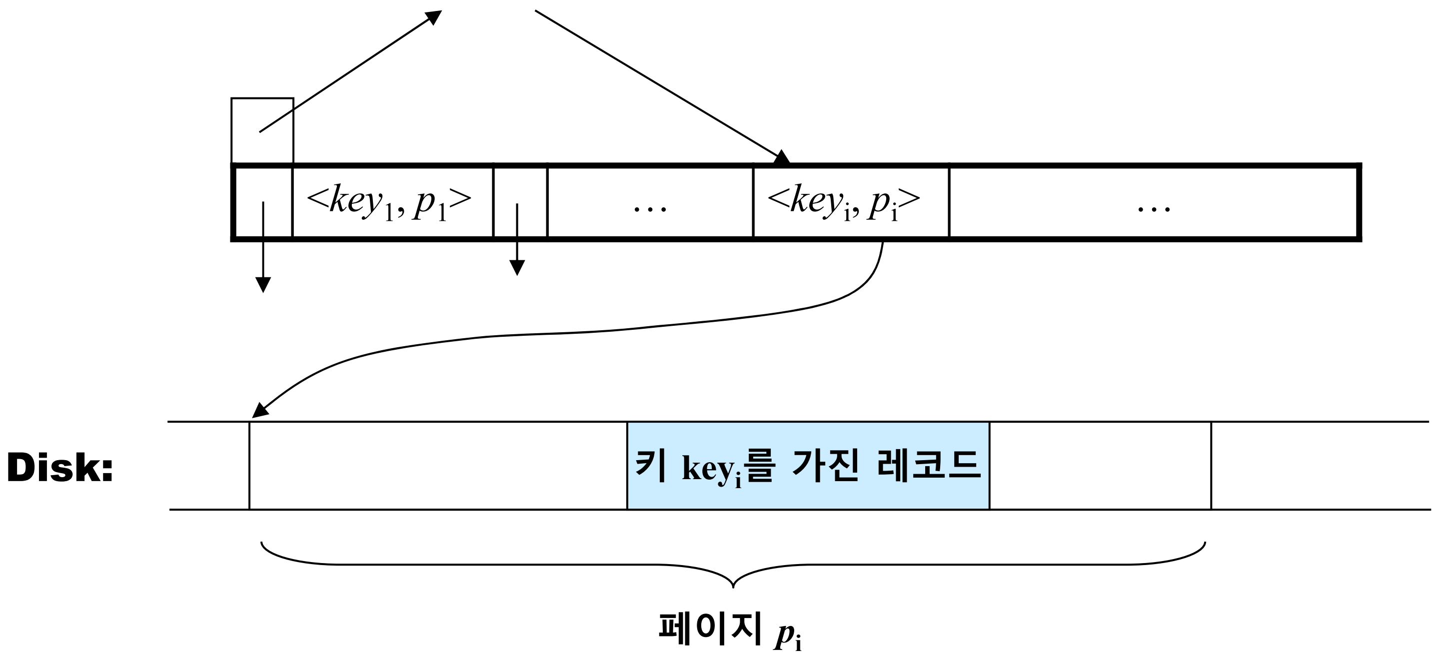
① 루트를 제외한 모든 노드는 $\lceil K/2 \rceil \sim K$ 개의 키를 갖는다.

② 모든 리프 노드는 같은 깊이를 가진다

B-Tree의 노드 구조



B-Tree를 통해 레코드에 접근하는 과정



B-Tree의 삽입

알고리즘 11-4 B-트리의 삽입(스케치)

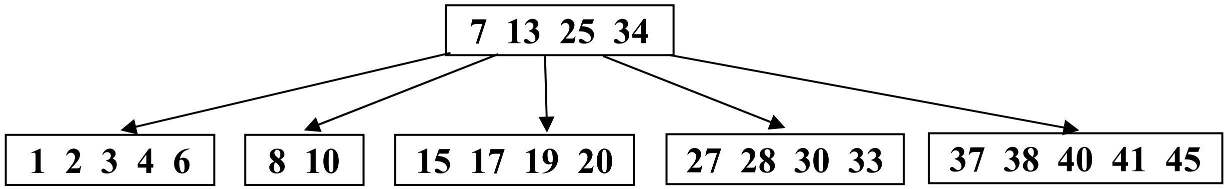
```
BTreeInsert(t, x):
◀ t: 트리의 루트 노드, x: 삽입하고자 하는 키
    x를 삽입할 리프 노드 r을 찾는다
    x를 r에 삽입 시도한다
    if (r에 오버플로우 발생) clearOverflow(r)

clearOverflow(r):
    if (r의 형제 노드 중 여유가 있는 노드가 있음)
        r의 남는 키를 넘긴다
   else
        r을 둘로 분할하고 가운데 키를 부모 노드로 넘긴다
        if (부모 노드 x에 오버플로우 발생) clearOverflow(x)
```

B-Tree의 삽입

K = 5 인 경우 예

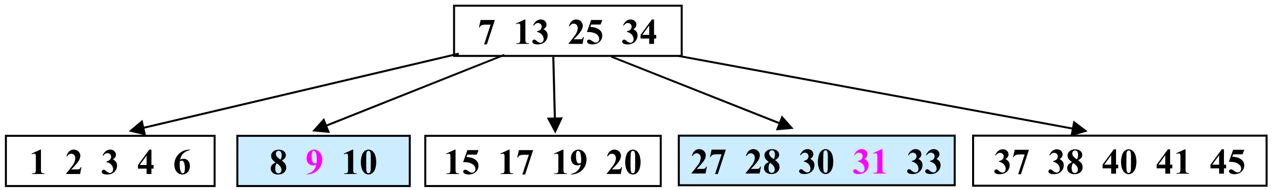
min=2 , max=5

1



9, 31 삽입

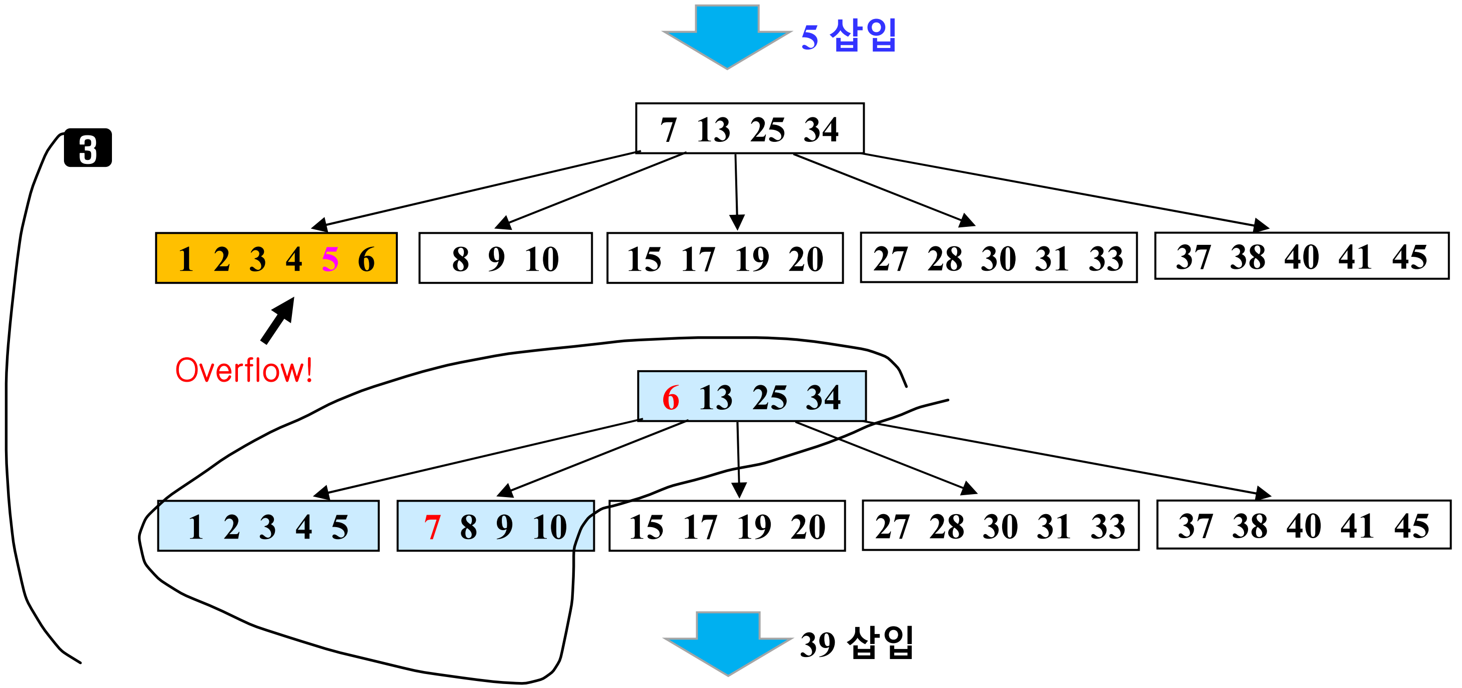
2



5 삽입

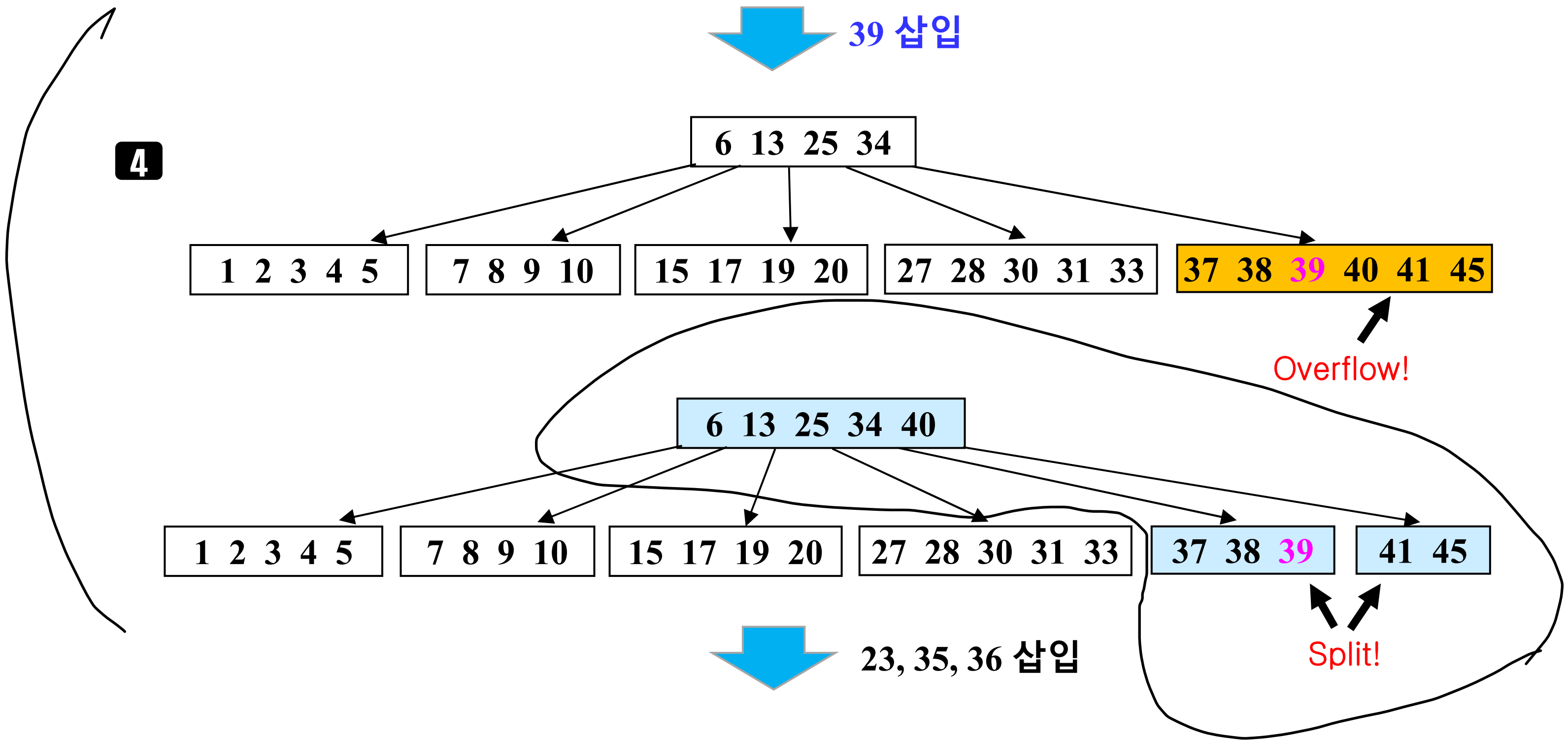
B-Tree의 삽입 예

K = 5인 경우



B-Tree의 삽입

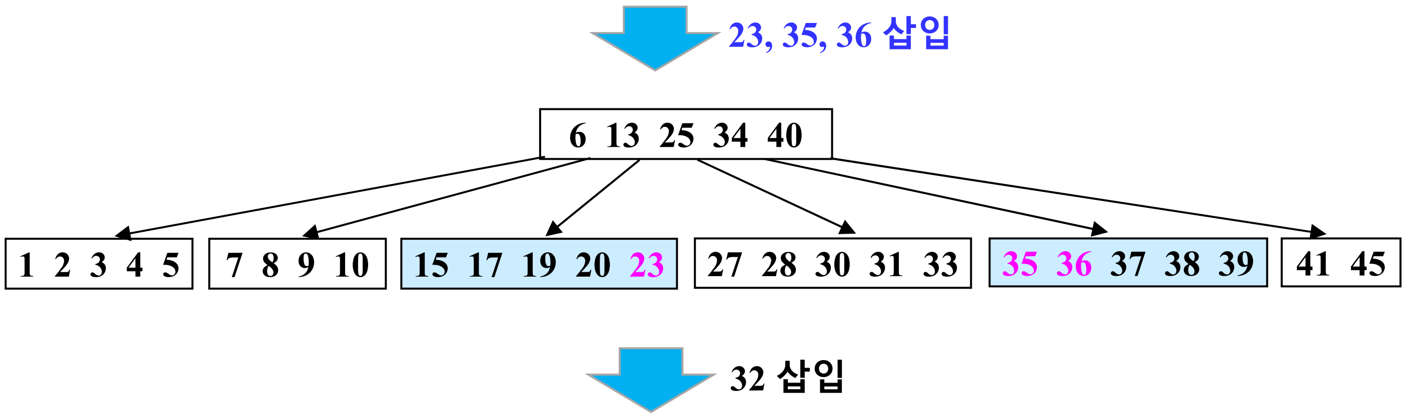
K = 5인 경우 예



B-Tree의 삽입 예

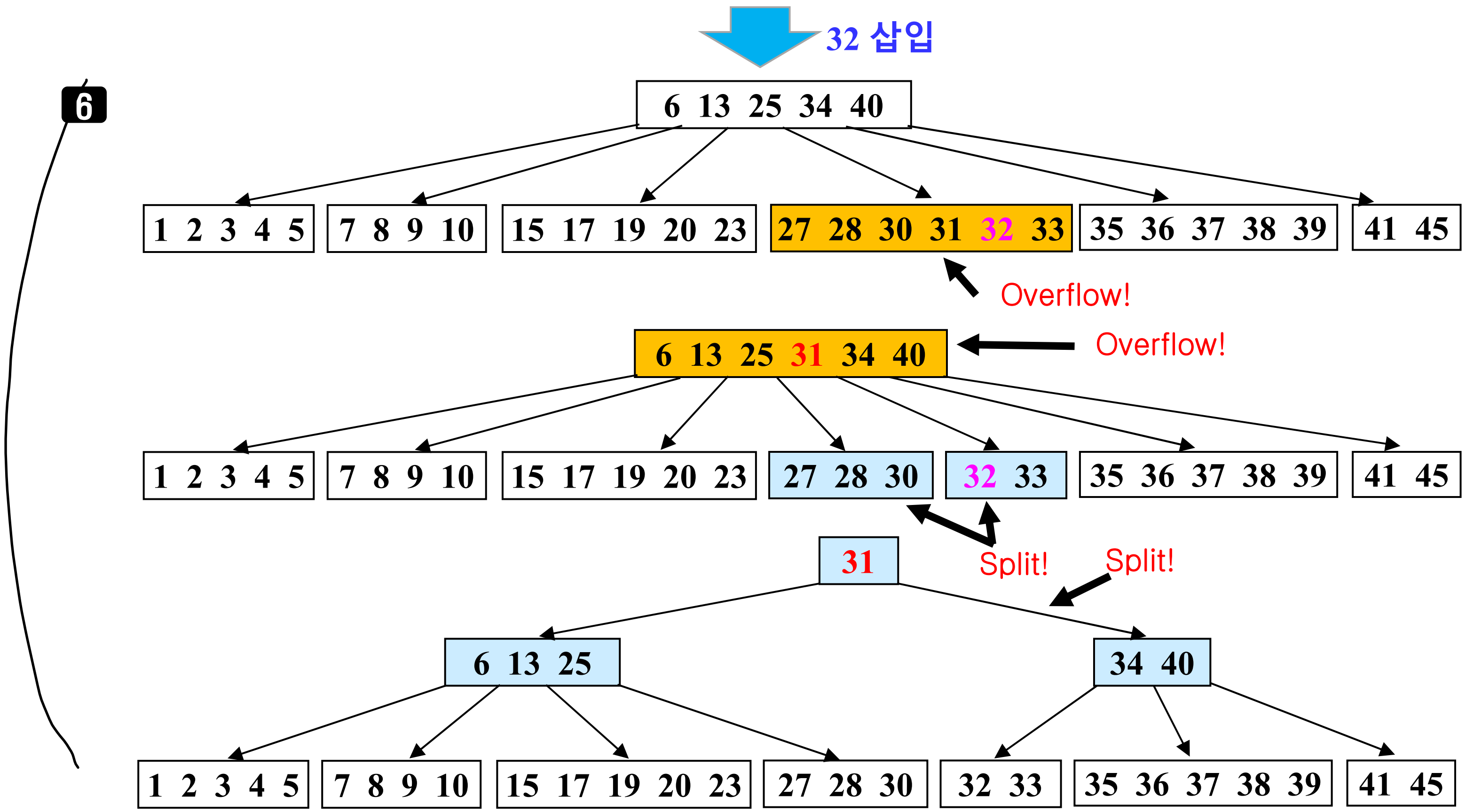
K = 5인 경우

5



B-Tree의 삽입

K = 5인 경우 예



B-Tree의 삭제

❶ x를 키로 갖고 있는 노드를 찾는다.

❷ 이 노드가 리프 노드가 아니면 x의 직후 원소 y를 가진 리프 노드 r을 찾아 x와 y를 맞바꾼다.
(직후 원소 y는 반드시 리프 노드에 있다.)

❸ 리프 노드 r에서 x를 제거한다.

❹ x를 제거한 후 노드에 언더플로우가 발생하면 적절히 해소한다

B-Tree의 삭제

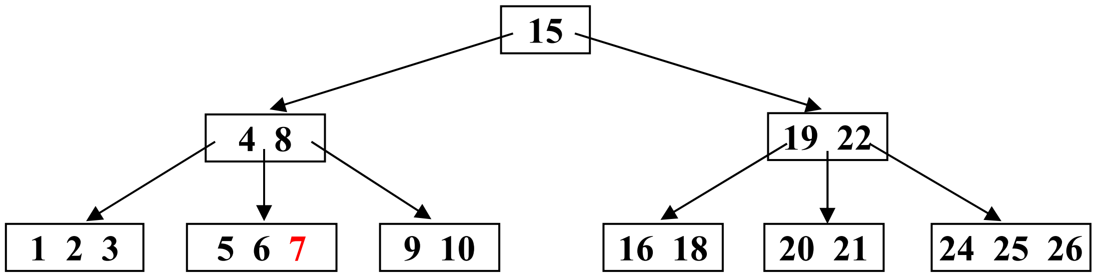
알고리즘 11-5 B-트리의 삭제(스케치)

```
BTreeDelete(t, x, r):
◀ t: 트리의 루트 노드, x: 삭제하고자 하는 키, r: x를 갖고 있는 노드
    if (x가 리프 노드 아님)
        x의 직후 원소 y를 가진 리프 노드를 찾는다
        x와 y를 맞바꾼다
    리프 노드에서 x를 제거하고 이 리프 노드를 r이라 한다
    if (r에서 언더플로우 발생) clearUnderflow(r)

clearUnderflow(r):
    if (r의 형제 노드 중 키를 하나 내놓을 수 있는 여분을 가진 노드가 있음)
        r이 키를 넘겨받는다
    else
        r의 형제 노드와 r을 병합하고 부모의 키 하나를 넘겨받는다
        if (부모 노드 p에 언더플로우 발생) clearUnderflow(p)
```

B-Tree의 삭제

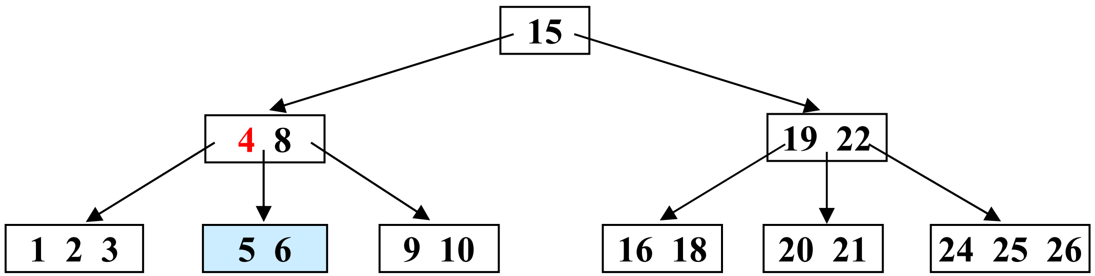
삭제 예

1



7 삭제

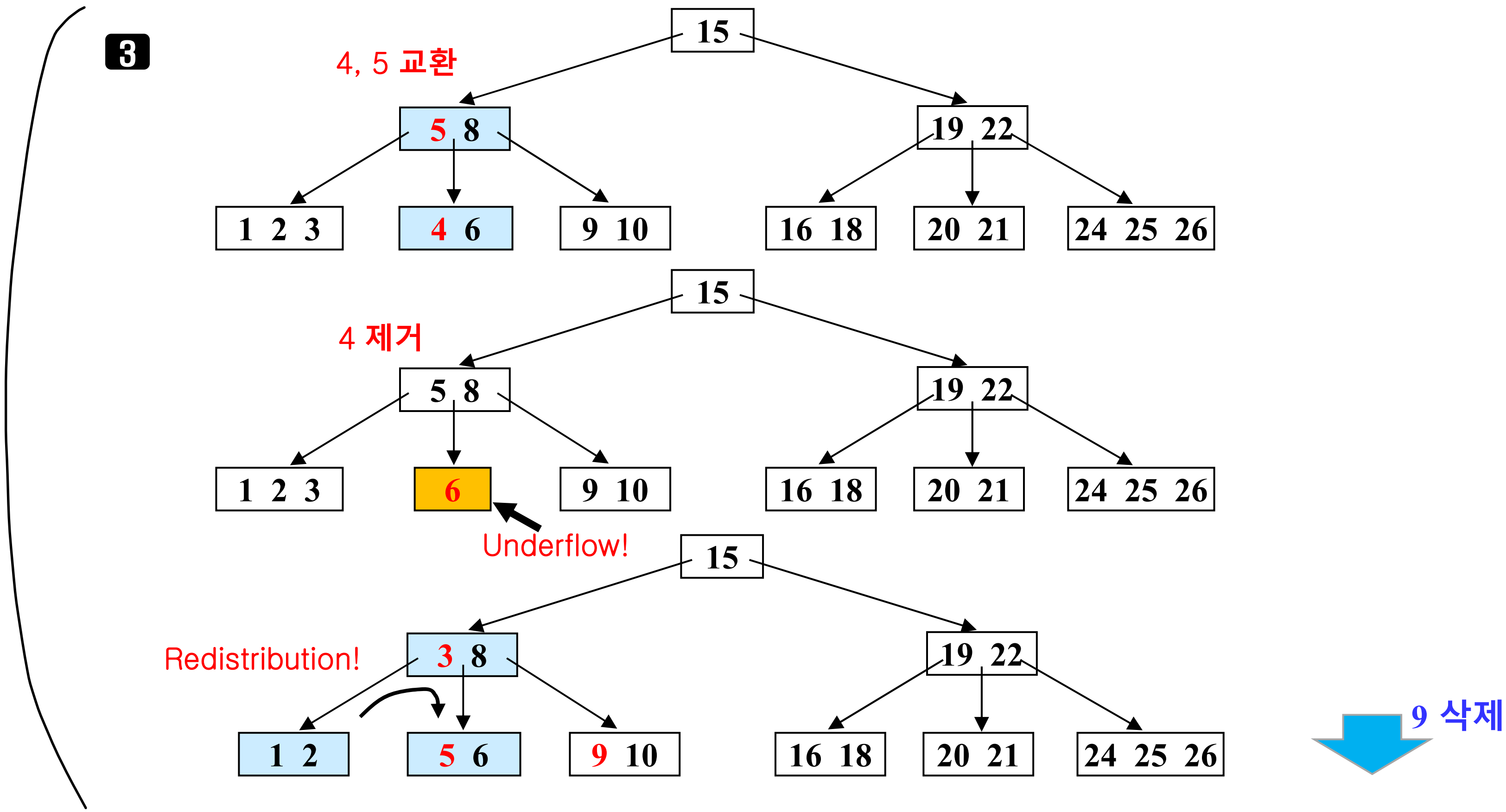
2



4 삭제

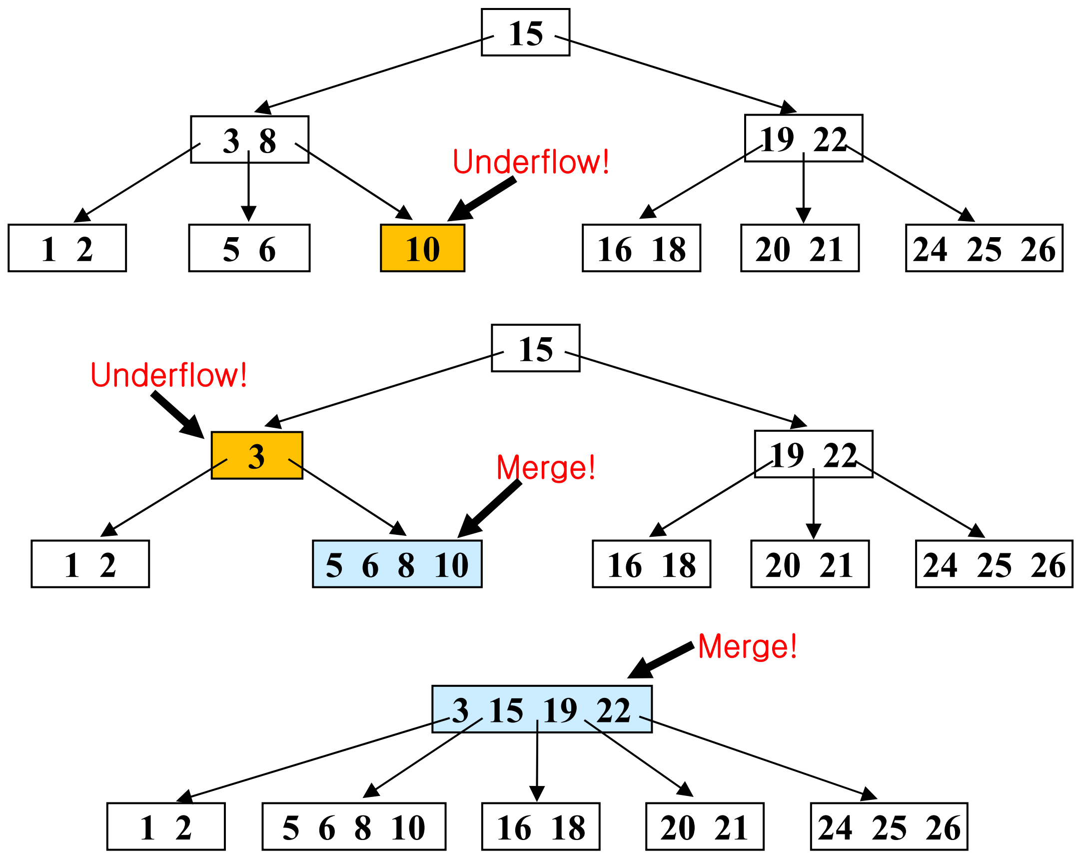
B-Tree의 삭제

삭제 예



B-Tree의 삭제

삭제 예



참고문헌

[1] Michael T. Goodrich 외 2인 지음, 김유성 외 2인 옮김, “C++로 구현하는 자료구조와 알고리즘”, 한티에듀, 2020.

[2] 주우석, “IT CookBook, C・C++ 로 배우는 자료구조론”, 한빛아카데미, 2019.

[3] 이지영, “C 로 배우는 쉬운 자료구조”, 한빛아카데미, 2022.

[4] “IT CookBook, 쉽게 배우는 자료구조 with 파이썬”, 문병로, 한빛아카데미, 2022.

[5] “프로그래밍 대회 공략을 위한 알고리즘과 자료 구조 입문”, 와타노베 유타카 저, 윤인성 역, 인사이트, 2021.

[6] “이것이 취업을 위한 코딩 테스트다 with 파이썬”, 나동빈, 한빛미디어, 2020.

[7] 문병로, “IT CookBook, 쉽게 배우는 알고리즘: 관계 중심의 사고법”(개정판), 개정판, 한빛아카데미, 2018.

[8] Richard E. Neapolitan, 도경구 역, “알고리즘 기초”, 도서출판 홍릉, 2017.