



# Search Tree

Jin Hyun Kim  
Fall, 2019

# 이진탐색트리

## (Binary Search Tree)

# 탐색트리

- 저장된 데이터에 대해 탐색, 삽입, 삭제, 갱신 등의 연산을 수행할 수 있는 자료구조
- 1차원 리스트나 연결리스트는 각 연산을 수행하는데  $O(N)$  시간이 소요
- 스택이나 큐는 특정 작업에 적합한 자료구조.
- 리스트 자료구조의 수행시간을 향상시키기 위한 트리 형태의 다양한 사전 자료구조들을 소개
  - 이진탐색트리, AVL트리, 2-3트리, 레드블랙트리, B-트리

# 이진탐색

- 정렬된 데이터의 중간에 위치한 항목을 기준으로 데이터를 두 부분으로 나누어 가며 특정 항목을 찾는 탐색방법

```
binary_search(left, right, t):
```

```
[1] if left > right: return None # 탐색 실패 (즉, t가 리스트에 없음)
```

```
[2] mid = (left + right) // 2 # 중간 항목의 인덱스 계산
```

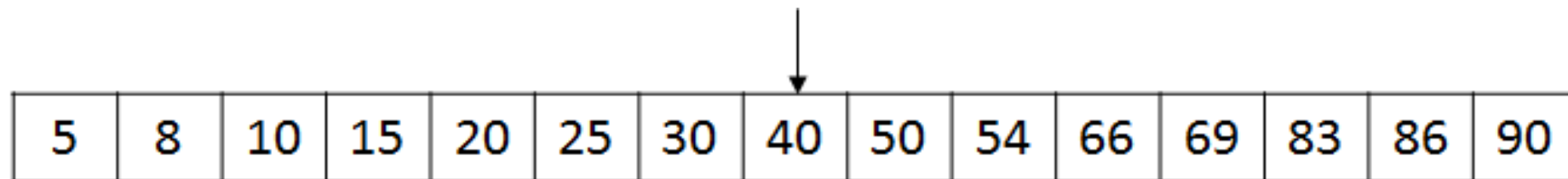
```
[3] if a[mid] == t: return mid # 탐색 성공
```

```
[4] if a[mid] > t: binary_search(left, mid-1, t) # 앞부분 탐색
```

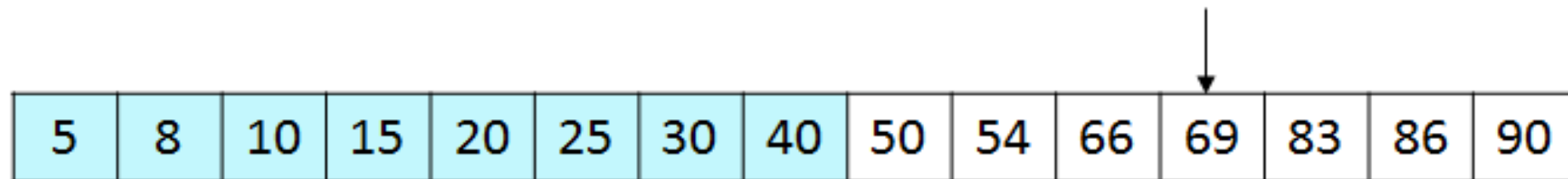
```
[5] else: binary_search(mid+1, right, t) # 뒷부분 탐색
```

# 이진탐색의 예

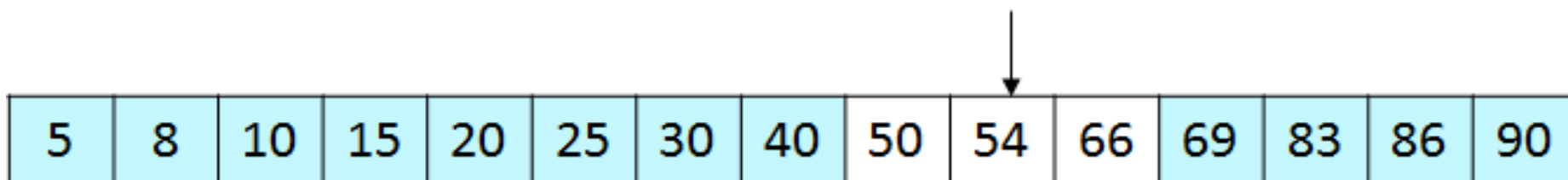
- 66을 찾아 가는 과정



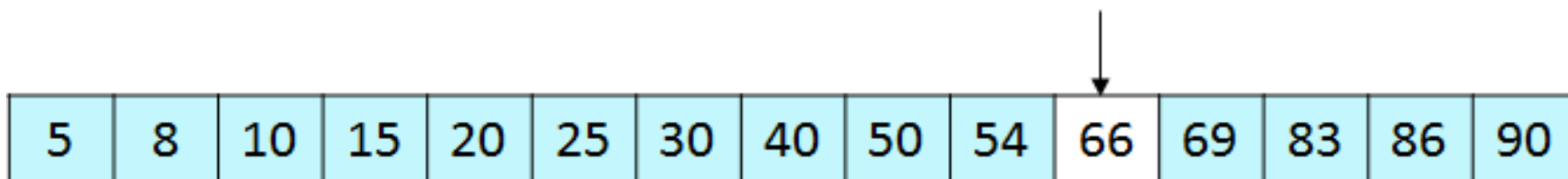
5	8	10	15	20	25	30	40	50	54	66	69	83	86	90
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----



5	8	10	15	20	25	30	40	50	54	66	69	83	86	90
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----



5	8	10	15	20	25	30	40	50	54	66	69	83	86	90
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----



5	8	10	15	20	25	30	40	50	54	66	69	83	86	90
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

# 수행시간

- $T(N)$  = 입력 크기  $N$ 인 정렬된 리스트에서 이진탐색을 하는데 수행되는 키 비교 횟수
- $T(N)$ 은 1번의 비교 후에 리스트의  $1/2$ , 즉, 앞부분이나 뒷부분을 재귀호출하므로

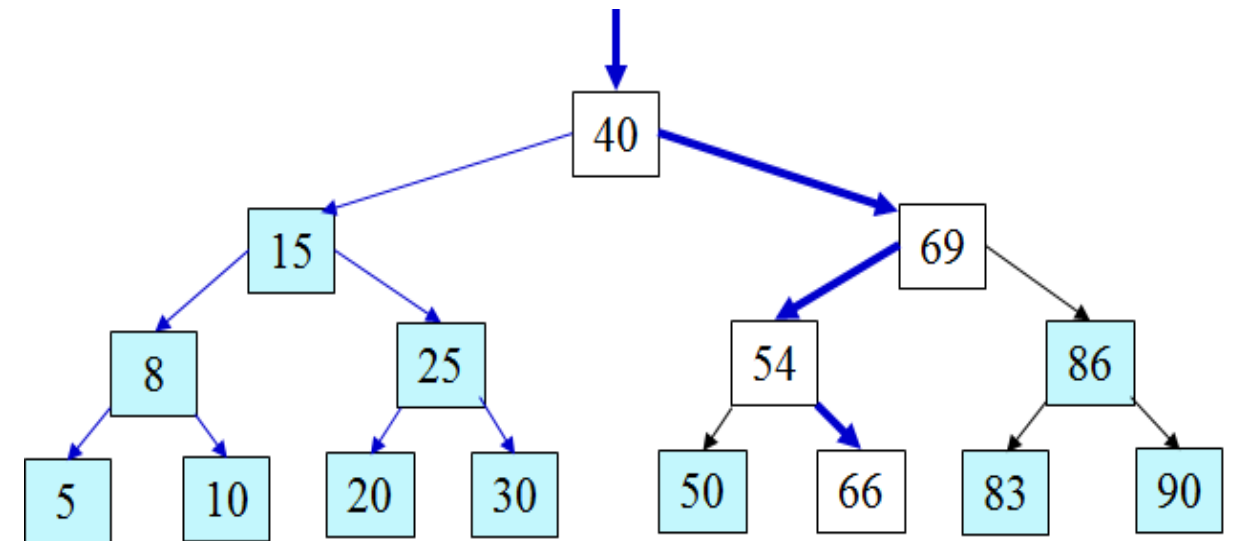
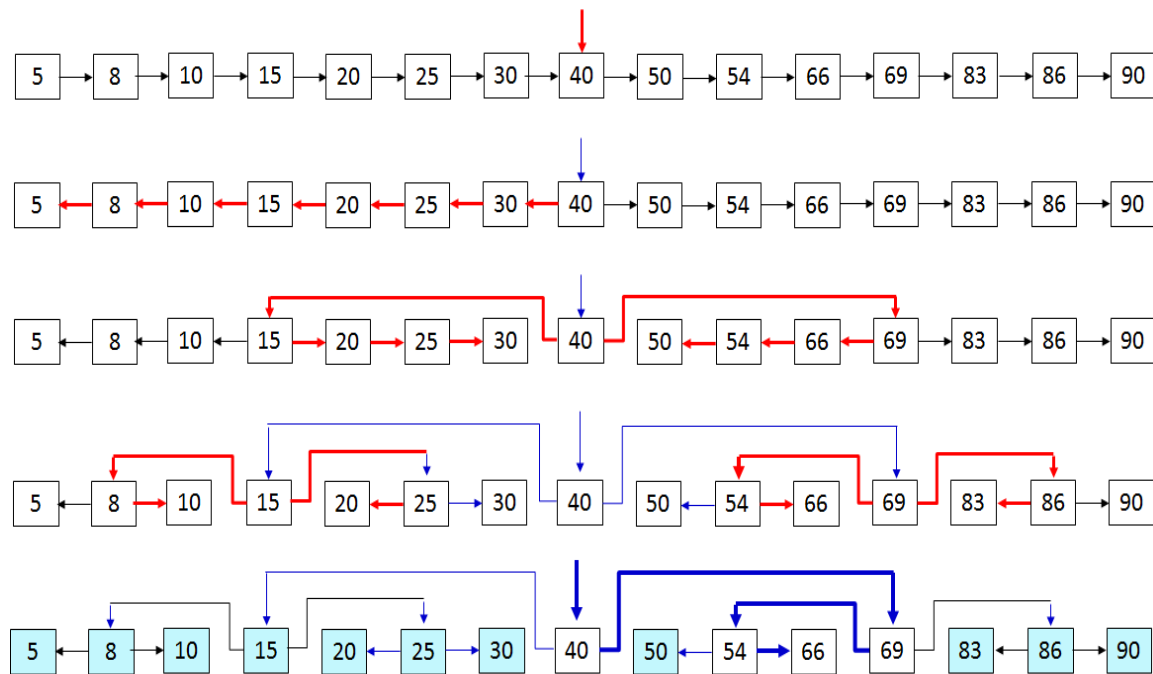
$$\begin{aligned} T(N) &= T(N/2) + 1 & T(N) &= T(N/2) + 1 \\ T(1) &= 1 & &= [T((N/2)/2) + 1] + 1 = T(N/2^2) + 2 \\ & & &= [T((N/2)/2^2) + 1] + 2 = T(N/2^3) + 3 \\ & & &= L = T(N/2^k) + k \\ & & &= T(1) + k, \text{ if } N = 2^k, k = \log_2 N \\ & & &= 1 + \log_2 N = O(\log N) \end{aligned}$$

# 이진탐색트리

## Binary Search Tree

- 이진탐색(Binary Search)의 개념을 트리 형태의 구조에 접목한 자료구조

# 이진탐색과 이진트리





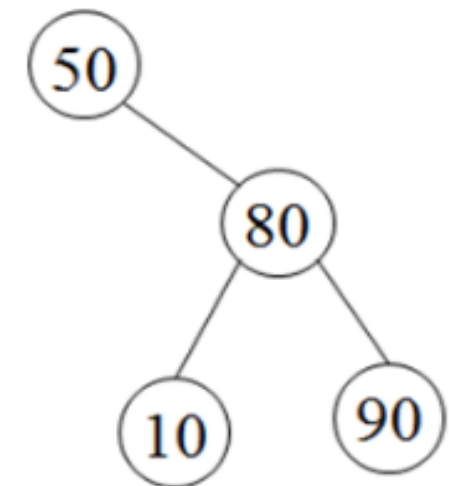
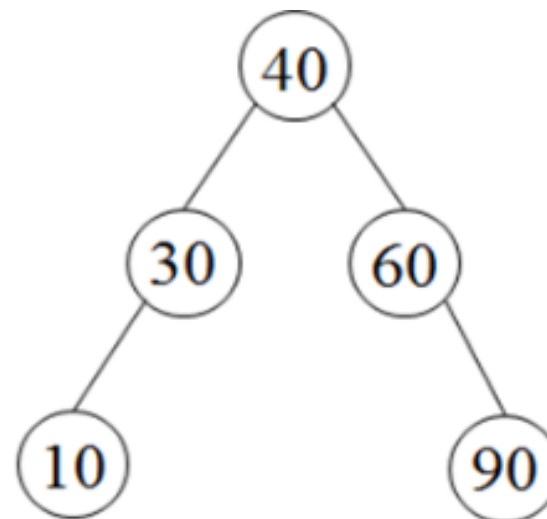
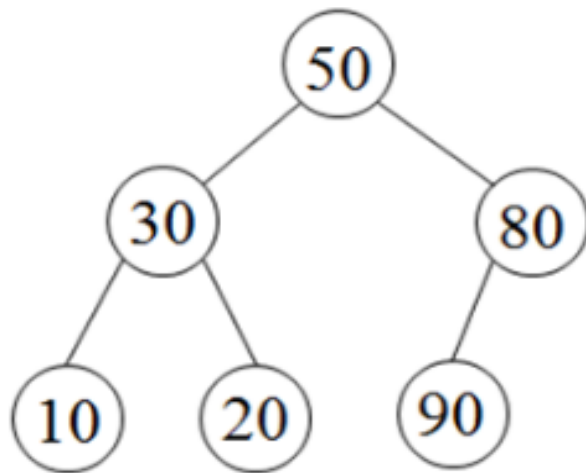
# 이진탐색트리

## Binary Search Tree

- 이진탐색(Binary Search)의 개념을 트리 형태의 구조에 접목한 자료구조
- 이진탐색트리는 이진트리로서 각 노드가 다음과 같은 조건을 만족한다.
- 각 노드  $n$ 의 키가  $n$ 의 왼쪽 서브트리에 있는 키들보다 (같거나) 크고,  $n$ 의 오른쪽 서브트리에 있는 키들보다 작다. **[이진탐색트리 조건]**

# 이진탐색트리

- 다음 중 이진탐색트리는?



# 이진탐색 클래스

```
01 class Node:
02     def __init__(self, key, value, left=None, right=None):
03         self.key    = key
04         self.value   = value
05         self.left    = left
06         self.right   = right
07
08 class BST:
09     def __init__(self): # 트리 생성자
10         self.root = None
11
12     def get(self, key): # 탐색 연산
13
14     def put(self, key, value): # 삽입 연산
15
16     def min(self): # 최솟값 가진 노드 찾기
17
18     def deletemin(self): # 최솟값 삭제
19
20     def delete(self, key): # 삭제 연산
```

노드 생성자  
키, 항목과 왼쪽, 오른쪽자식 레퍼런스

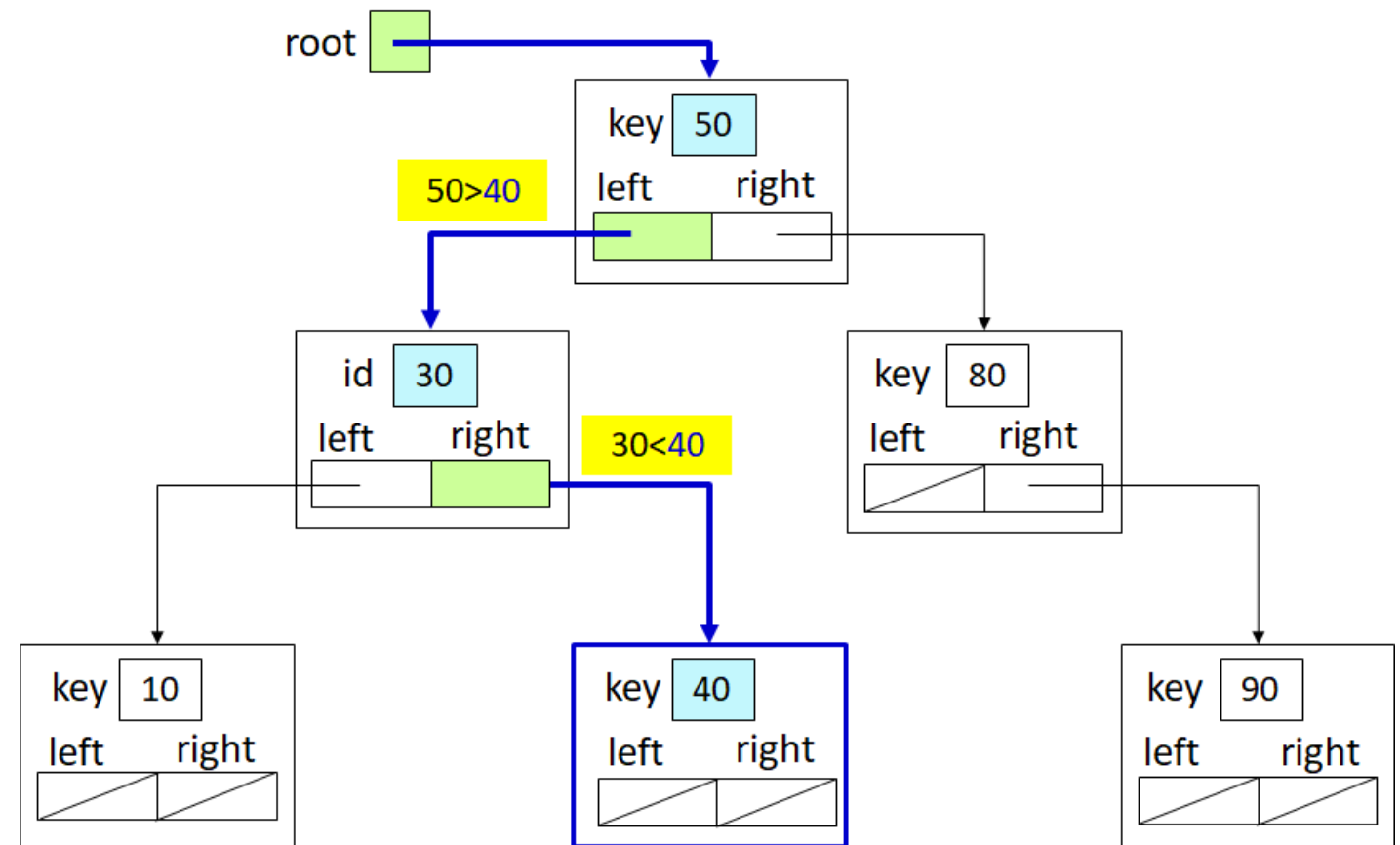
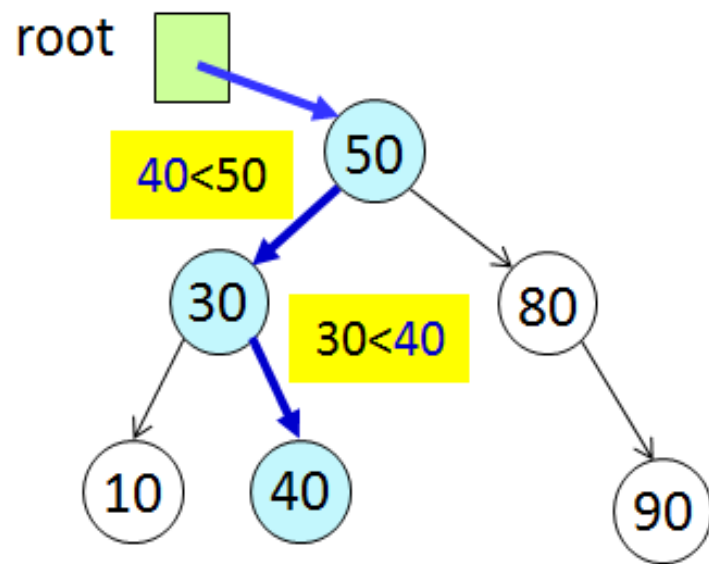
트리 루트

탐색, 삽입, 삭제 연산  
min()과 delete\_min()은  
삭제 연산에서 사용됨

# 탐색연산: `get(key)`

- 탐색하고자 하는 키가  $k$ 라면, 루트의 키와  $k$ 를 비교하는 것으로 탐색을 시작
- $k$ 가 루트의 키가  $k$  보다 작으면, 루트의 왼쪽 서브트리에서  $k$ 를 찾고, 크면 루트의 오른쪽 서브트리에서  $k$ 를 찾으며, 같으면 탐색 성공
- 왼쪽이나 오른쪽 서브트리에서  $k$ 를 탐색은 루트에서의 탐색과 동일

# 탐색연산: get(key)



# 탐색연산: get(key)

```
def get(self, k): # 탐색 연산
    return self.get_item(self.root, k)
```

```
def get_item(self, n, k):
    if n == None:
        return None
    if n.key > k:
        return self.get_item(n.left, k)
    elif n.key < k:
        return self.get_item(n.right, k)
    else:
        return n.value
```

탐색 실패

k가 노드의 key보다 작으면  
왼쪽 서브트리 탐색

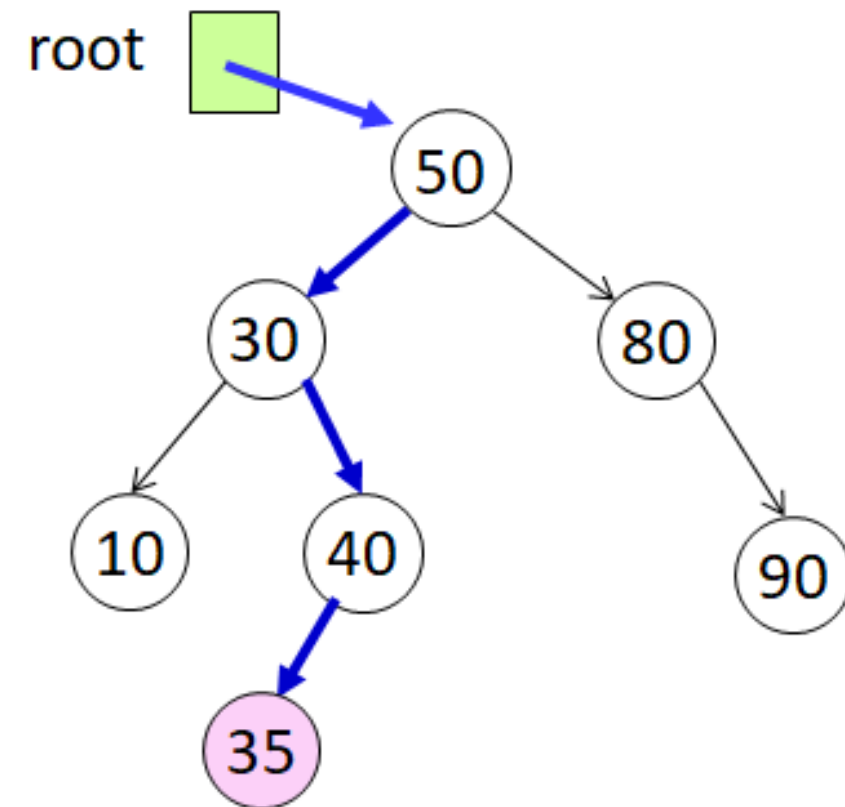
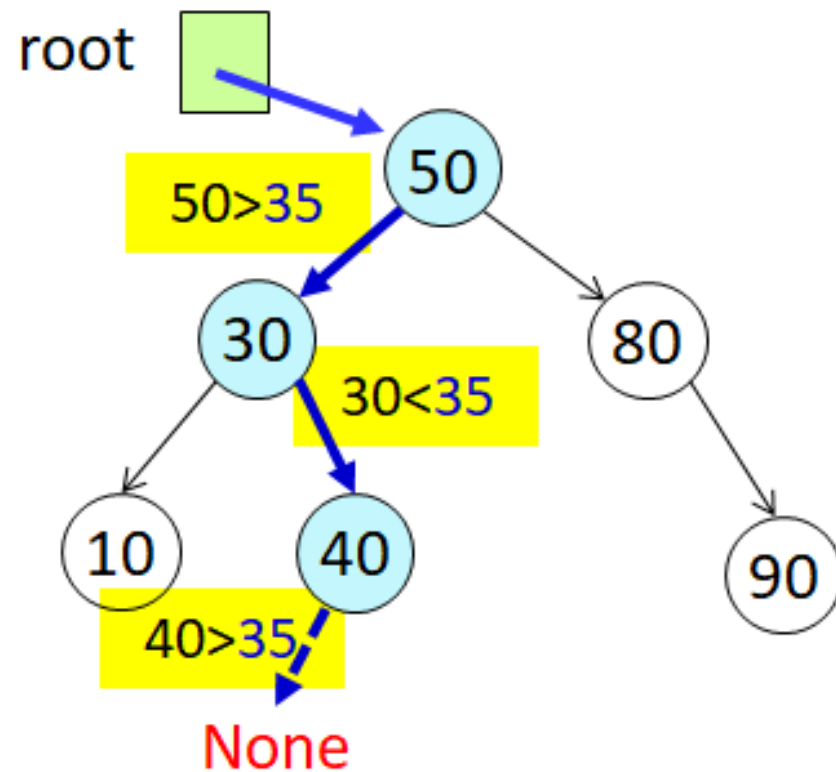
k가 노드의 key보다 크면  
오른쪽 서브트리 탐색

탐색 성공

# 삽입연산: `put(key, value)`

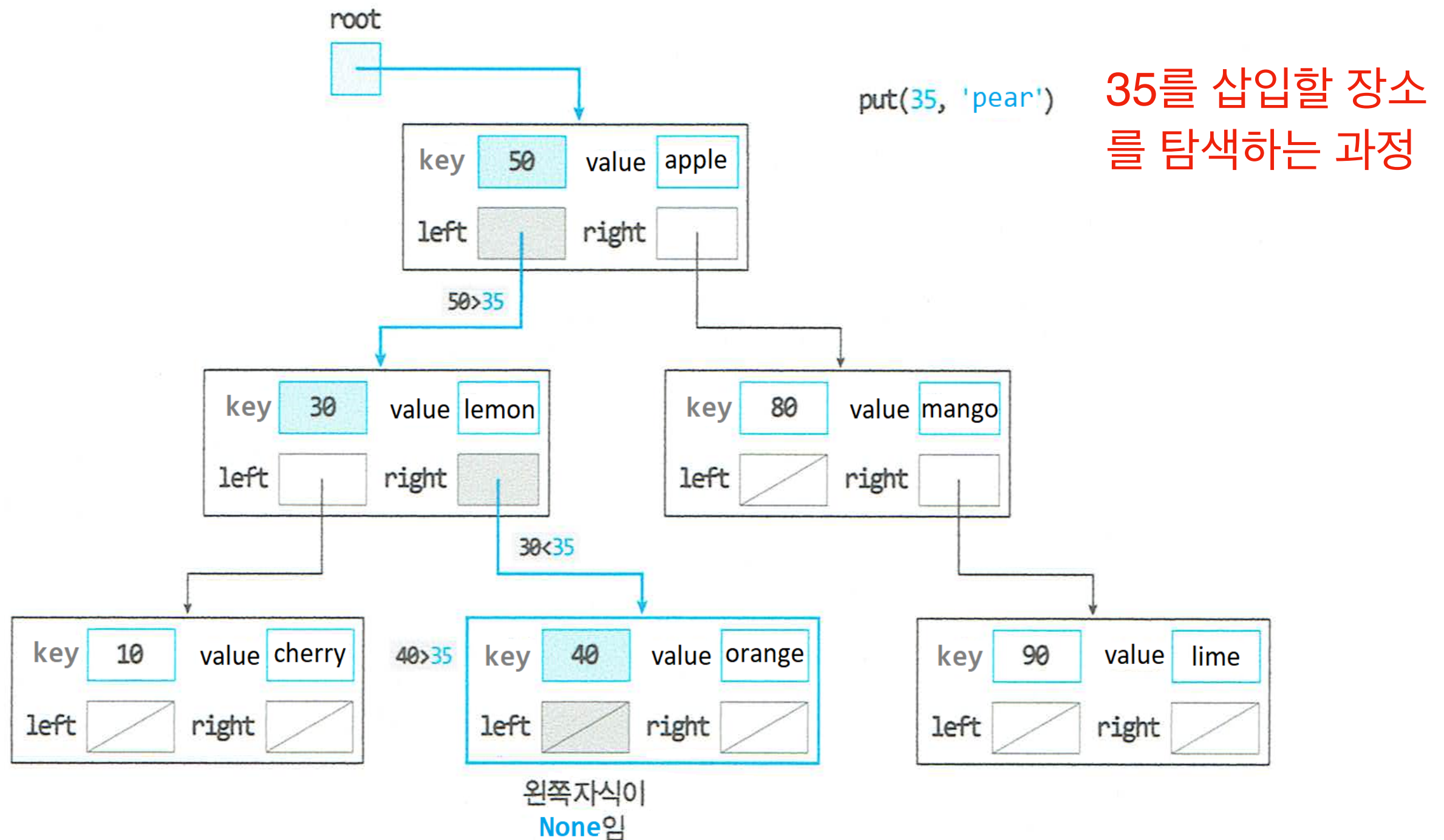
- 삽입은 탐색 연산과 거의 동일
- 탐색 중 `None`을 만나면 새 노드를 생성하여 부모노드와 연결
- 단, 이미 트리에 존재하는 키를 삽입한 경우, `value`만 갱신

# 삽입연산: put(key, value)



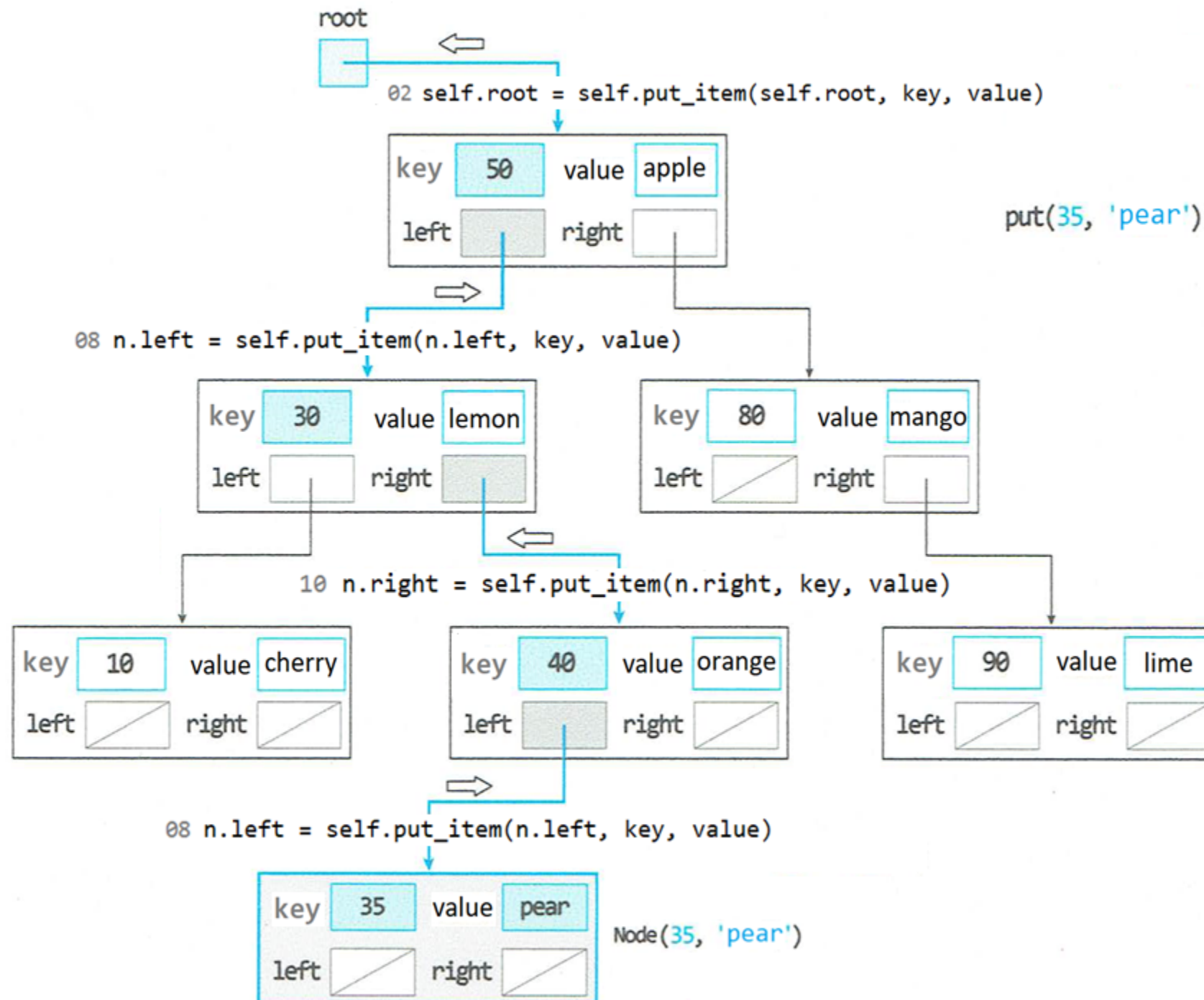


# 삽입연산: put(key, value)



# 삽입연산: put(key, value)

새 노드 삽입 후 루트  
로 거슬러 올라가며  
재 연결하는 과정



# 삽입연산: put(key, value)

```
01 def put(self, key, value): # 삽입 연산
02     self.root = self.put_item(self.root, key, value)
03
04 def put_item(self, n, key, value):
05     if n == None:
06         return Node(key, value)
07     if n.key > key:
08         n.left = self.put_item(n.left, key, value)
09     elif n.key < key:
10         n.right = self.put_item(n.right, key, value)
11     else:
12         n.vlaue = value
13     return n
```

루트와 put\_item()이 리턴하는 노드를 재 연결

새 노드 생성

n의 왼쪽자식과 put\_item()이 리턴하는 노드를 재 연결

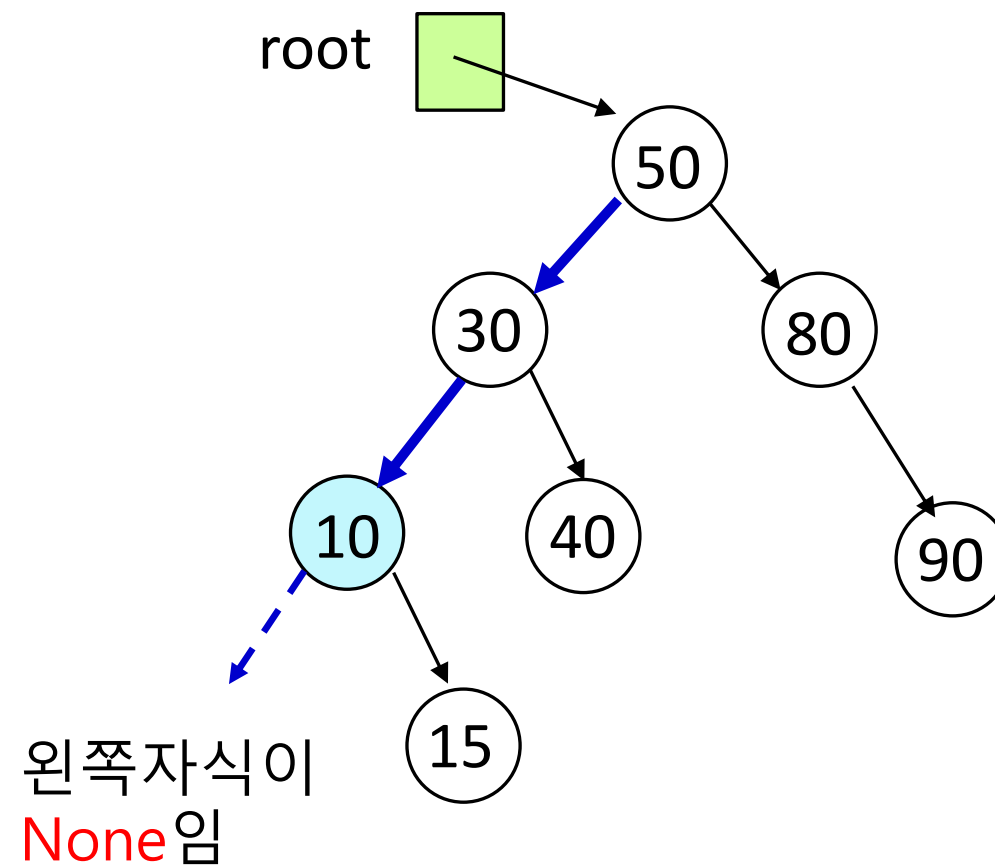
n의 오른쪽자식과 put\_item()이 리턴하는 노드를 재 연결

key가 이미 있으므로 value만 갱신

부모노드와 연결하기 위해 노드 n을 리턴

# 최소값 찾기

- 최소값은 루트노드로부터 왼쪽 자식을 따라 내려가며, None을 만났을 때 None의 부모가 가진 value



# 최소값 찾기

- 최솟값은 루트노드로부터 왼쪽 자식을 따라 내려가며, None을 만났을 때 None의 부모가 가진 value

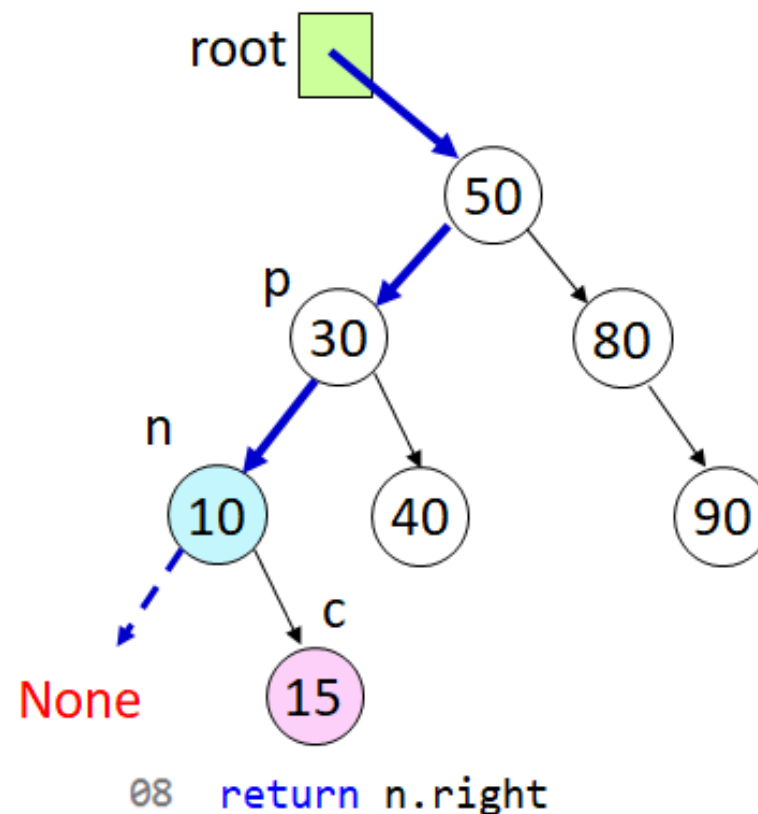
```
01 def min(self): # 최솟값 가진 노드 찾기
02     if self.root == None:
03         return None
04     return self.minimum(self.root)
05
06 def minimum(self, n):
07     if n.left == None:
08         return n
09     return self.minimum(n.left)
```

왼쪽자식이 None인  
노드(최솟값을 가진)  
를 리턴

왼쪽자식으로 재귀호출  
하며 최솟값 가진 노드  
를 리턴

# 최소값 삭제

- 최솟값을 가진 노드를 삭제하는 것은 최솟값을 가진 노드  $n$ 을 찾아낸 뒤,  $n$ 의 부모  $p$ 와  $n$ 의 오른쪽 자식  $c$ 를 연결
- 이 때  $c$  가 None이더라도 자식으로 연결



# 최소값 삭제

- 최솟값을 가진 노드를 삭제하는 것은 최솟값을 가진 노드  $n$ 을 찾아낸 뒤,  $n$ 의 부모  $p$ 와  $n$ 의 오른쪽 자식  $c$ 를 연결
- 이 때  $c$ 가  $\text{None}$ 이더라도 자식으로 연결

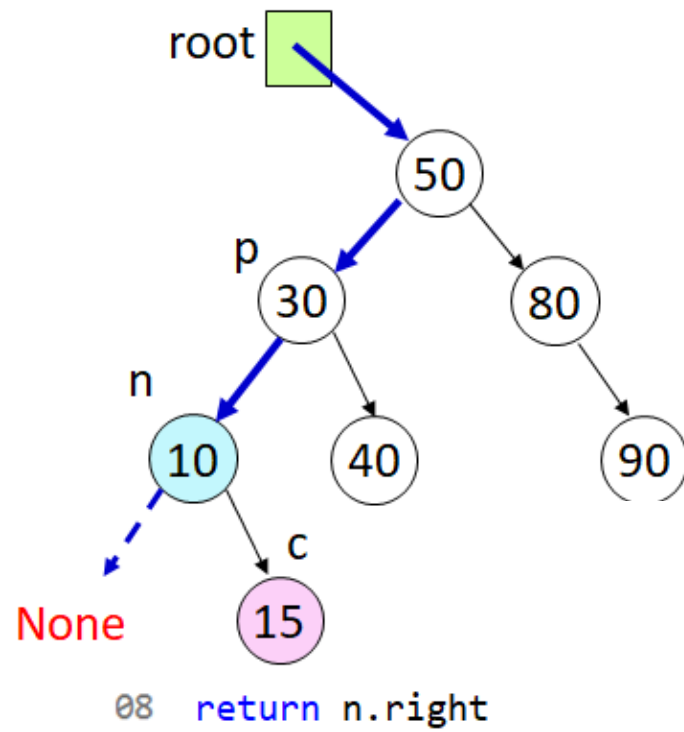
```
01 def delete_min(self): # 최솟값 삭제
02     if self.root == None:
03         print('트리가 비어 있음')
04     self.root = self.del_min(self.root)
05
06 def del_min(self, n):
07     if n.left == None:
08         return n.right
09     n.left = self.del_min(n.left)
10     return n
```

루트와 `del_min()`이 리턴하는 노드를 재 연결

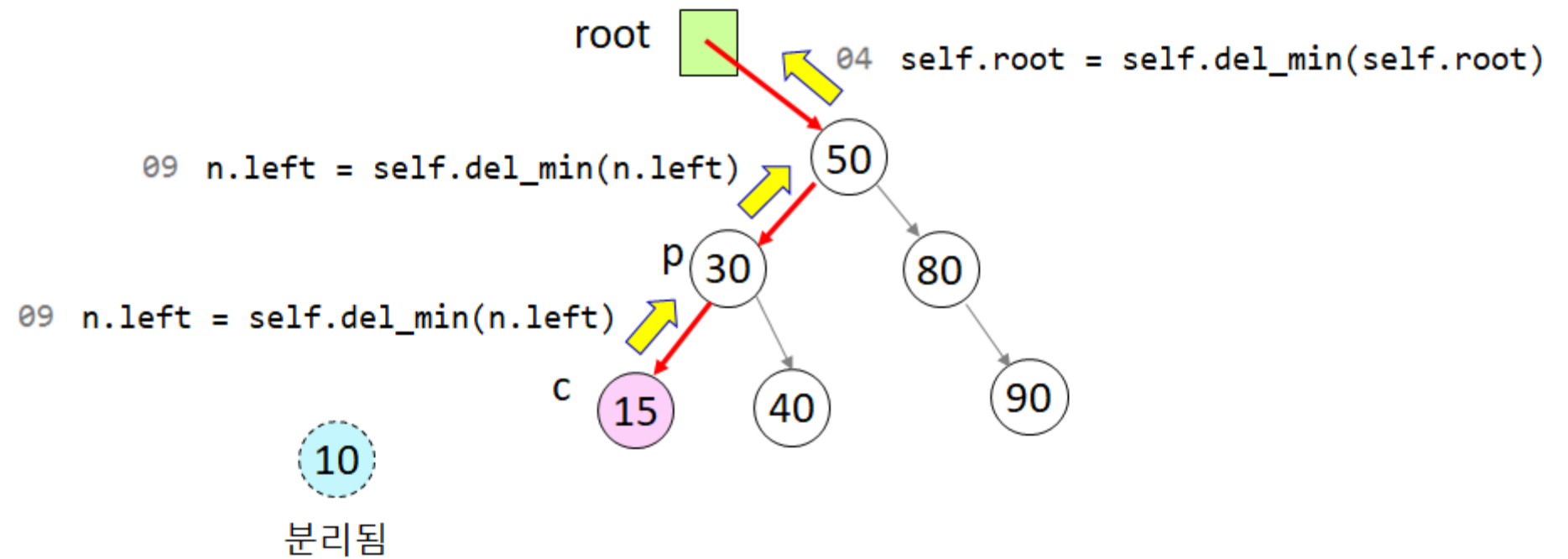
최솟값 가진 노드의 오른쪽 자식을 리턴

$n$ 의 왼쪽자식과 `del_min()`이 리턴하는 노드를 재 연결

# 최소값 삭제



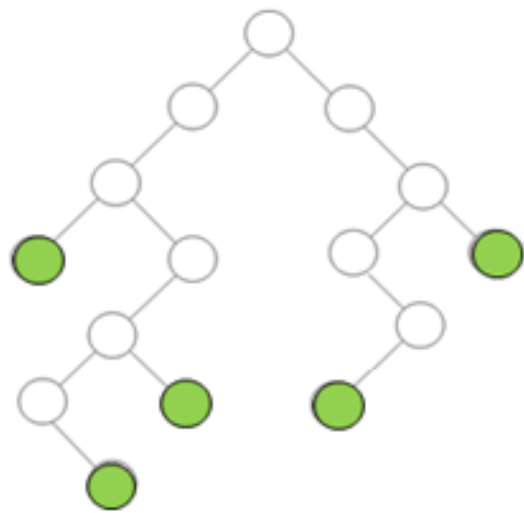
08 return n.right



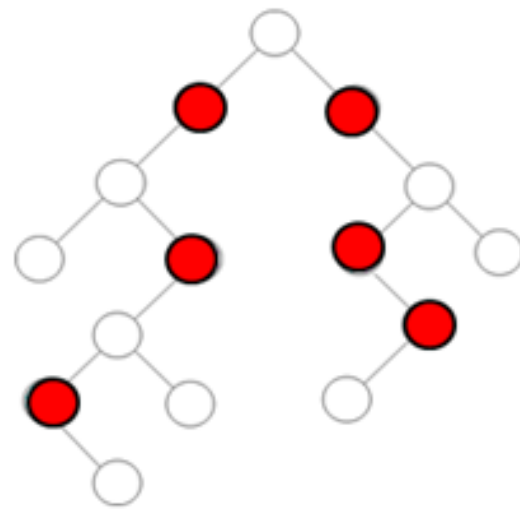


# 삭제연산: delete(key)

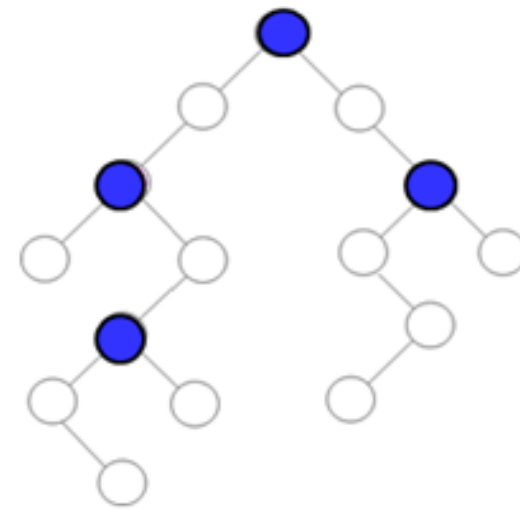
- 우선 삭제하고자 하는 노드를 찾은 후 이진탐색트리 조건을 만족하도록 삭제된 노드의 부모와 자식(들)을 연결해 주어야 함
- 삭제되는 노드가 자식이 없는 경우(case 0), 자식이 하나인 경우(case 1), 자식이 둘인 경우(case 2)로 나누어 delete 연산을 수행



case 0



case 1

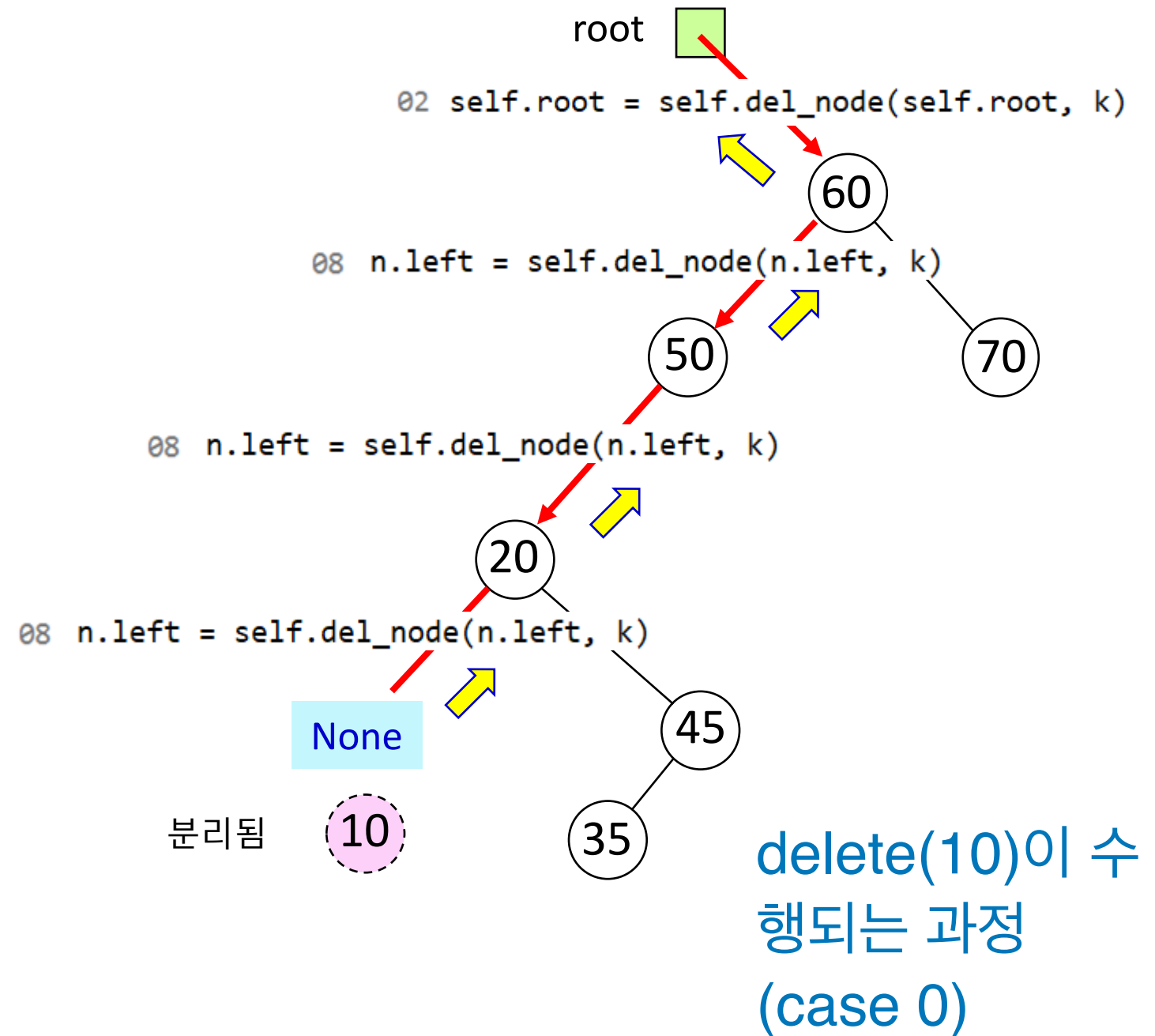
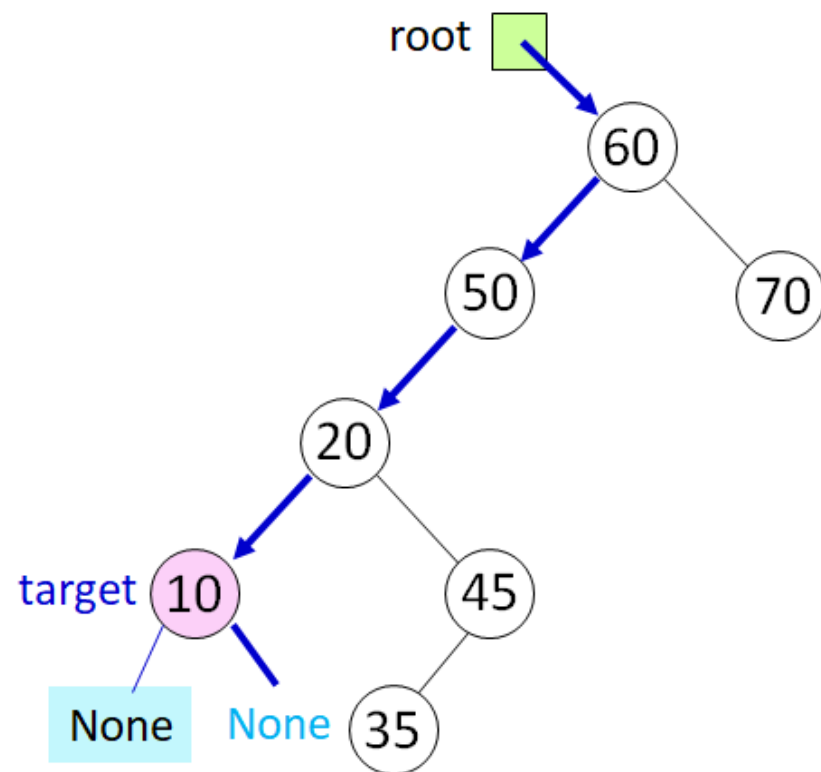


case 2

# 삭제연산: delete(key)

- **Case 0:** 삭제해야 할 노드  $n$ 의 부모가  $n$ 을 가리키던 레퍼런스를 None으로 만든다.
- **Case 1:**  $n$ 가 한쪽 자식인  $c$ 만 가지고 있다면,  $n$ 의 부모와  $n$ 의 자식  $c$ 를 직접 연결
- **Case 2:**  $n$ 의 부모는 하나인데  $n$ 의 자식이 둘이므로  $n$ 의 자리에 중위순회하면서  $n$ 을 방문하기 직전 노드(Inorder Predecessor, 중위 선행자) 또는 직후에 방문되는 노드(Inorder Successor, 중위 후속자)로 대체

# Case 0



# Case 0

delete(10)이 수행되는 과정 (case 0)

```
01 def delete(self, k): # 삭제 연산
02     self.root = self.del_node(self.root, k)
```

```
04 def del_node(self, n, k):
```

루트와 del\_node()가 리턴하는 노드를 재 연결

```
05     if n == None:
06         return None
```

```
07     if n.key > k:
08         n.left = self.del_node(n.left, k)
```

```
09     elif n.key < k:
10         n.right = self.del_node(n.right, k)
```

n의 왼쪽자식과 del\_node()가 리턴하는 노드를 재 연결

```
11     else:
12         if n.right == None:
13             return n.left
```

n의 오른쪽자식과 del\_node()가 리턴하는 노드를 재 연결

```
14         if n.left == None:
15             return n.right
```

```
16         target = n
```

target은 삭제될 노드

```
17         n = self.minimum(target.right)
```

target의 중위 후속자 찾아 n이 참조하게 함

```
18         n.right = self.del_min(target.right)
```

```
19         n.left = target.left
```

n의 오른쪽자식과 target의 오른쪽자식 연결

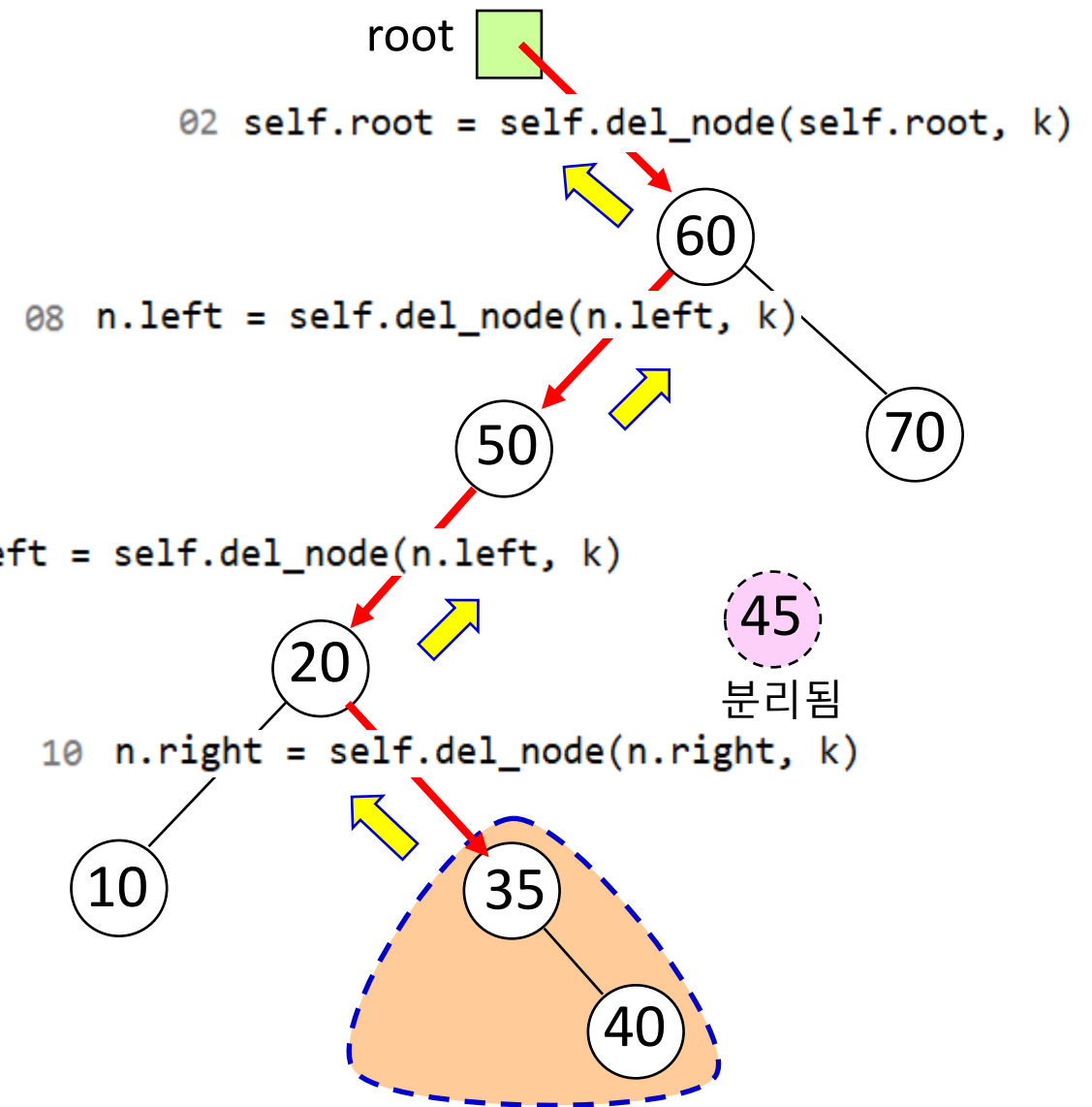
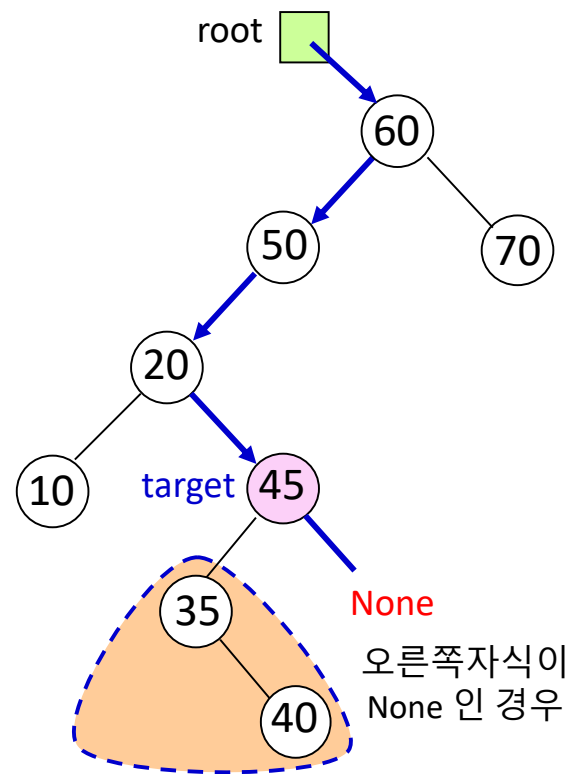
```
20     return n
```

n의 왼쪽자식과 target의 왼쪽자식 연결

target 오른쪽자식 트리중 가장 작은 키를 가진 n을 찾아서 target을 대체하게 함

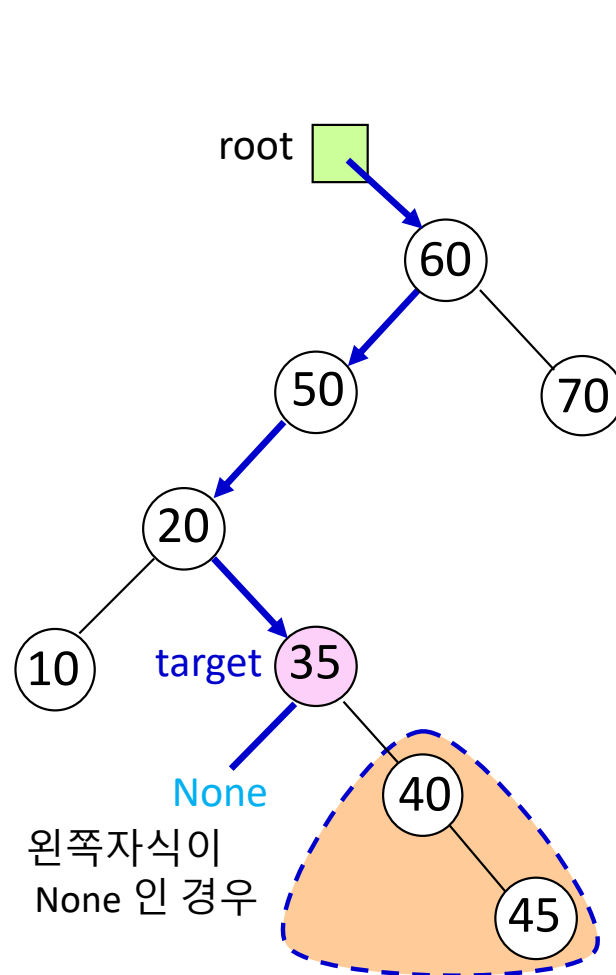
target 오른쪽자식 트리중 가장 작은 키를 가진 n을 찾아서 삭제 후, 삭제된 노드의 오른쪽 노드를 n의 오른쪽에 붙임

# Case 1

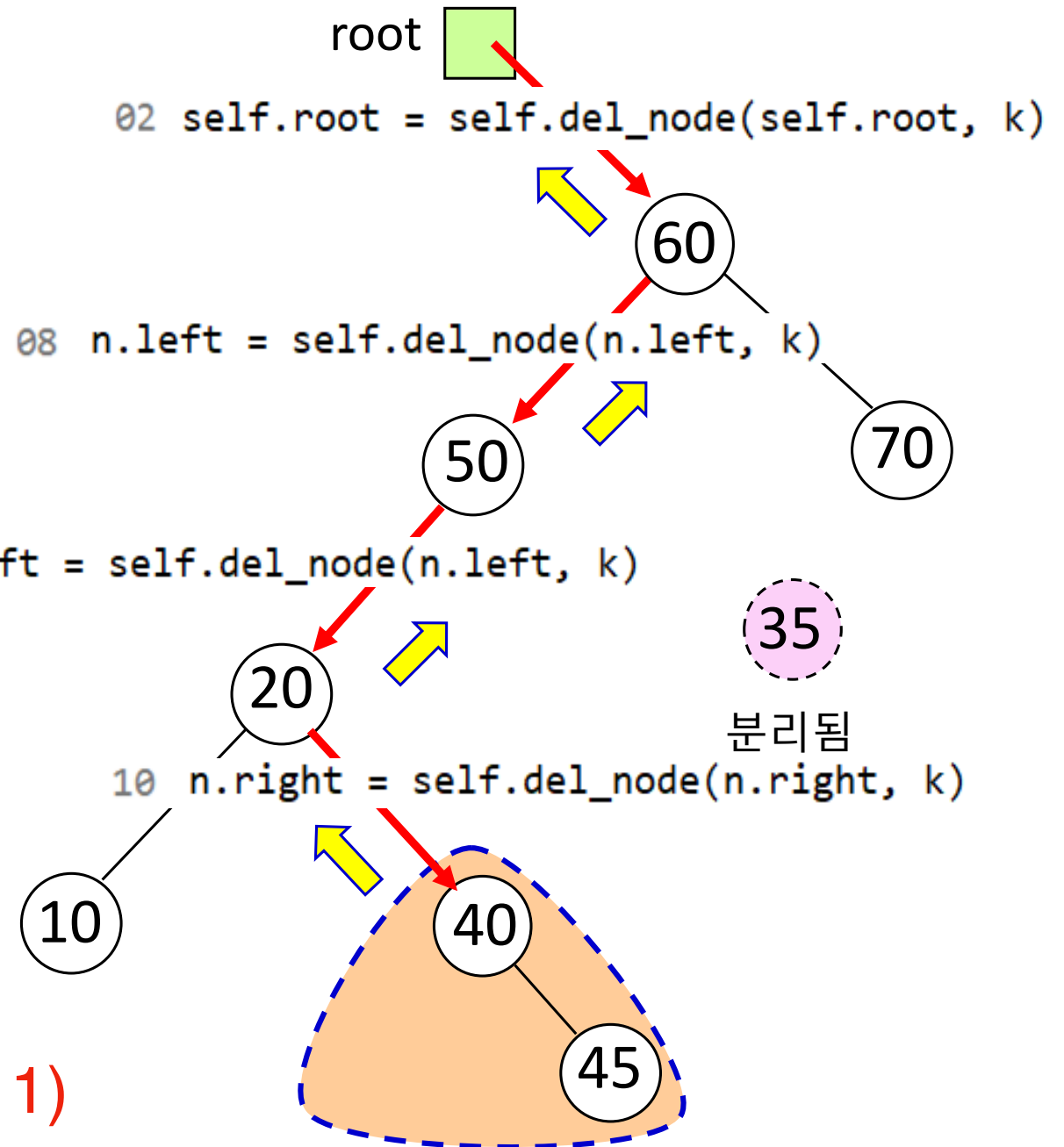


delete(45)가 수행되는 과정 (case 1)

# Case 1



delete(35)가 수행되는 과정 (case 1)



# Case 1

delete(10)이 수행되는 과정 (case 0)

```
01 def delete(self, k): # 삭제 연산
02     self.root = self.del_node(self.root, k)
```

```
04 def del_node(self, n, k):
```

루트와 del\_node()가 리턴하는 노드를 재 연결

```
05     if n == None:
06         return None
```

```
07     if n.key > k:
08         n.left = self.del_node(n.left, k)
```

```
09     elif n.key < k:
10         n.right = self.del_node(n.right, k)
```

n의 왼쪽자식과 del\_node()가 리턴하는 노드를 재 연결

```
11     else:
12         if n.right == None:
```

```
13             return n.left
14         if n.left == None:
```

```
15             return n.right
16         target = n
```

target은 삭제될 노드

```
17         n = self.minimum(target.right)
```

target의 중위 후속자 찾아 n이 참조하게 함

```
18         n.right = self.del_min(target.right)
```

```
19         n.left = target.left
```

n의 오른쪽자식과 target의 오른쪽자식 연결

```
20     return n
```

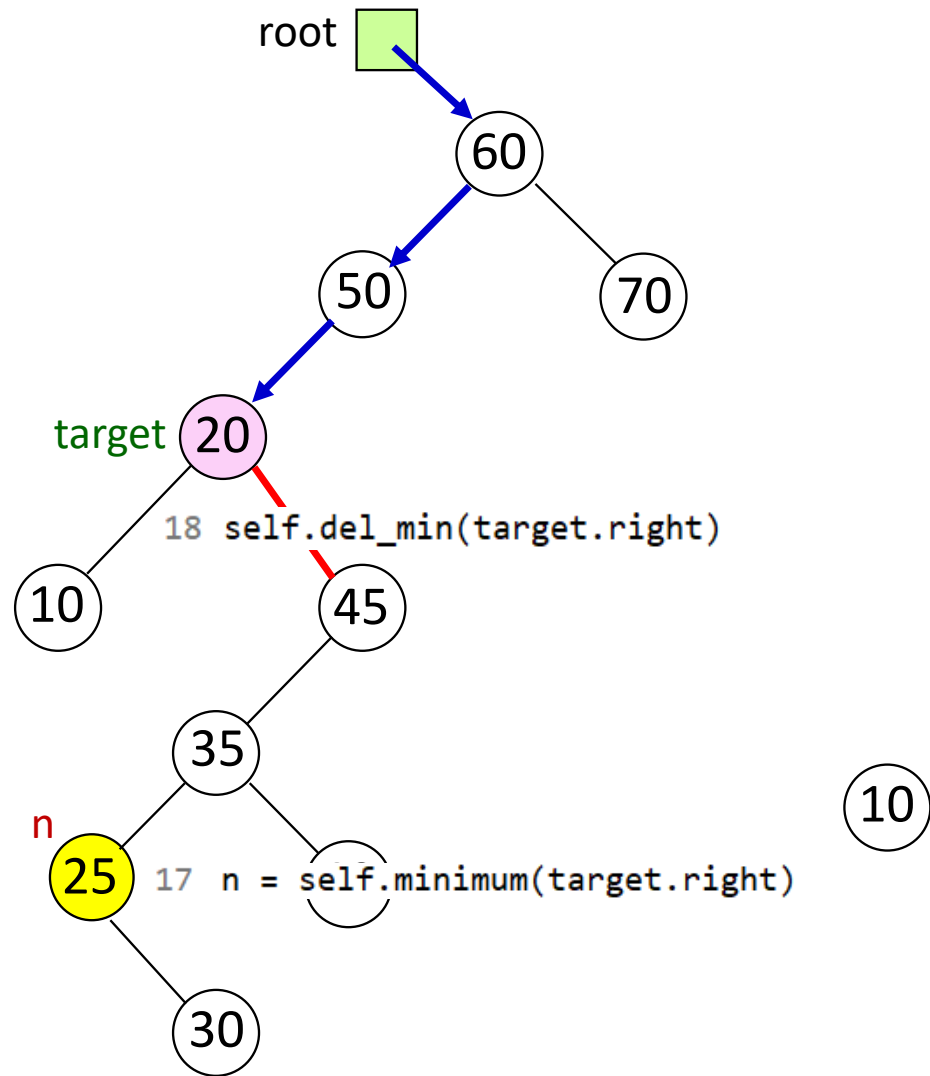
n의 왼쪽자식과 target의 왼쪽자식 연결

target 오른쪽자식 트리중 가장 작은 키를 가진 n을 찾아서 target을 대체하게 함

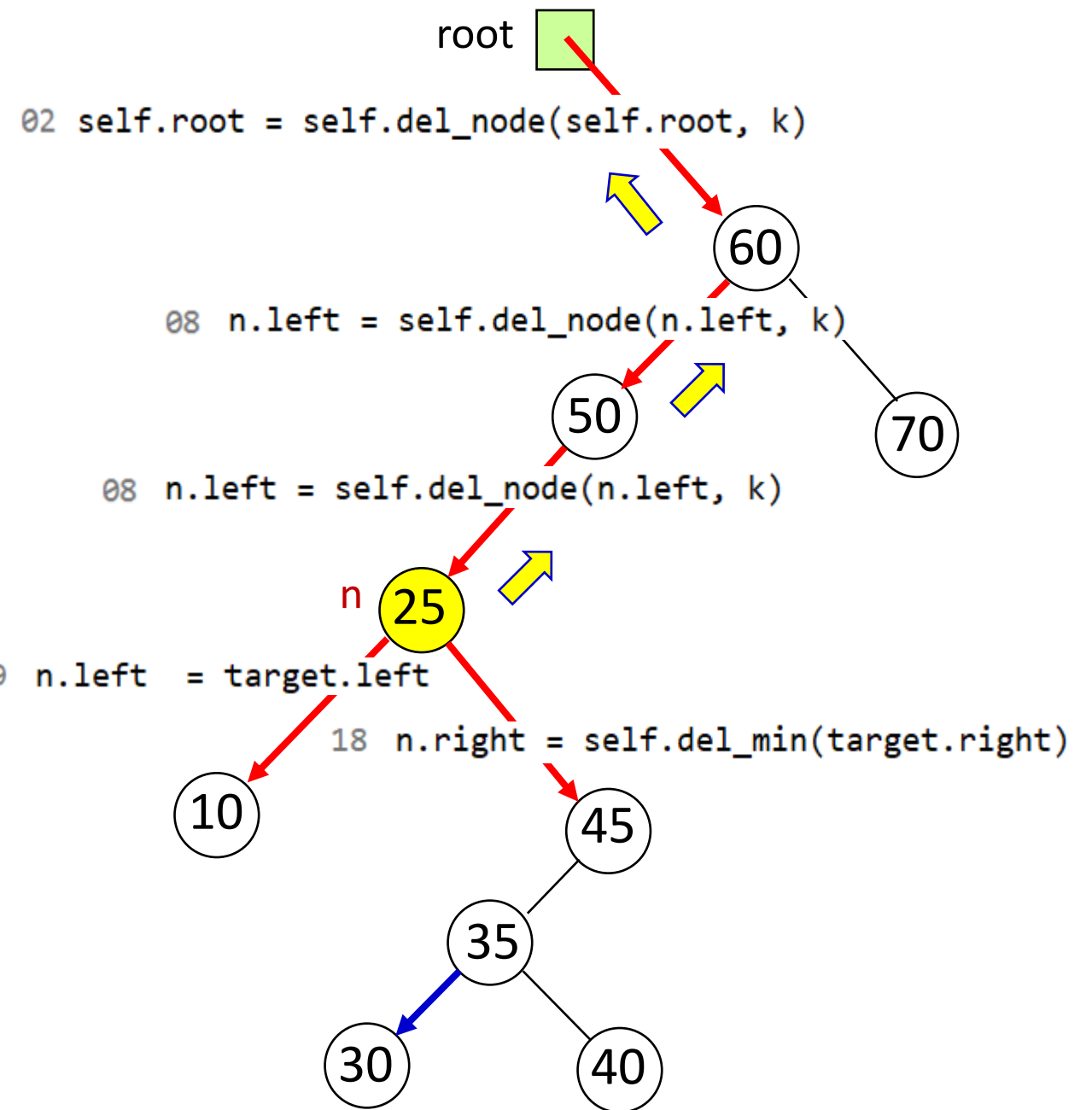
target 오른쪽자식 트리중 가장 작은 키를 가진 n을 찾아서 삭제 후, 삭제된 노드의 오른쪽 노드를 n의 오른쪽에 붙임

# Case 2

```
10 n.right = self.del_node(n.right, k)
```



delete(20)이 수행되는 과정





# Case 2

delete(10)이 수행되는 과정 (case 0)

```
01 def delete(self, k): # 삭제 연산
02     self.root = self.del_node(self.root, k)
```

```
04 def del_node(self, n, k):
```

루트와 del\_node()가 리턴하는 노드를 재 연결

```
05     if n == None:
06         return None
```

```
07     if n.key > k:
08         n.left = self.del_node(n.left, k)
```

```
09     elif n.key < k:
10         n.right = self.del_node(n.right, k)
```

n의 왼쪽자식과 del\_node()가 리턴하는 노드를 재 연결

```
11     else:
12         if n.right == None:
13             return n.left
```

n의 오른쪽자식과 del\_node()가 리턴하는 노드를 재 연결

```
14         if n.left == None:
15             return n.right
```

```
16         target = n
```

target은 삭제될 노드

```
17         n = self.minimum(target.right)
```

target의 중위 후속자 찾아 n이 참조하게 함

```
18         n.right = self.del_min(target.right)
```

```
19         n.left = target.left
```

n의 오른쪽자식과 target의 오른쪽자식 연결

```
20     return n
```

n의 왼쪽자식과 target의 왼쪽자식 연결

target 오른쪽자식 트리중 가장 작은 키를 가진 n을 찾아서 target을 대체하게 함

target 오른쪽자식 트리중 가장 작은 키를 가진 n을 찾아서 삭제 후, 삭제된 노드의 오른쪽 노드를 n의 오른쪽에 붙임

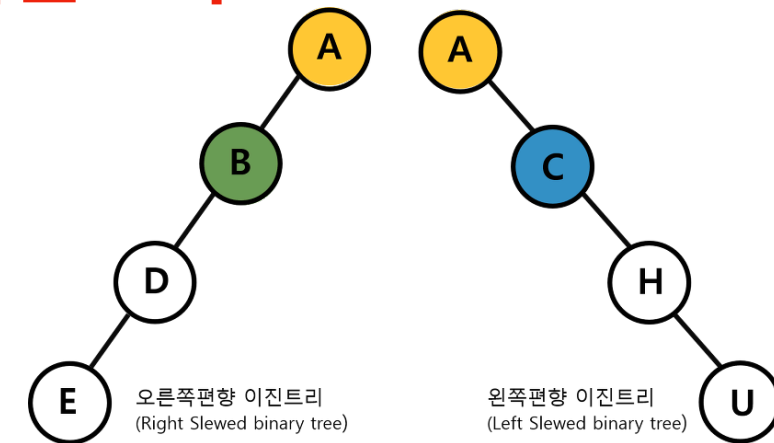
# 수행시간

- 이진탐색트리에서 탐색, 삽입, 삭제 연산은 공통적으로 **루트에서 탐색을 시작하여 최악의 경우에 이파리까지 내려가고**, 삽입과 삭제 연산은 다시 루트까지 거슬러 올라가야 함
- 트리를 한 층 내려갈 때는 재귀호출이 발생하고, 한 층을 올라갈 때는 재 연결이 수행되는데, 이들 각각은  $O(1)$  시간 소요
- **연산들의 수행시간은 각각 트리의 높이( $h$ )에 비례,  $O(h)$**

# 수행시간

- N개의 노드가 있는 이진탐색트리의 높이가 가장 낮은 경우는 완전 이진트리 형태일 때이고, **가장 높은 경우는 편향이진트리**
- 따라서 이진트리의 높이 h는 아래와 같다

$$\lceil \log(N+1) \rceil \approx \log N \leq h \leq N$$



- Empty 이진탐색트리에 랜덤하게 선택된 N개의 키를 삽입한다고 가정했을 때, 트리의 높이는 약  $1.39 \log N$

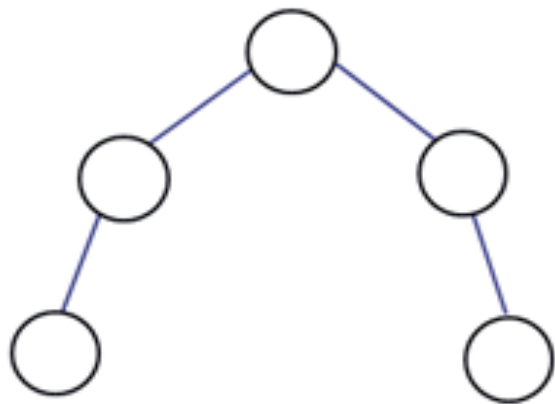
# AVL Tree

# AVL트리

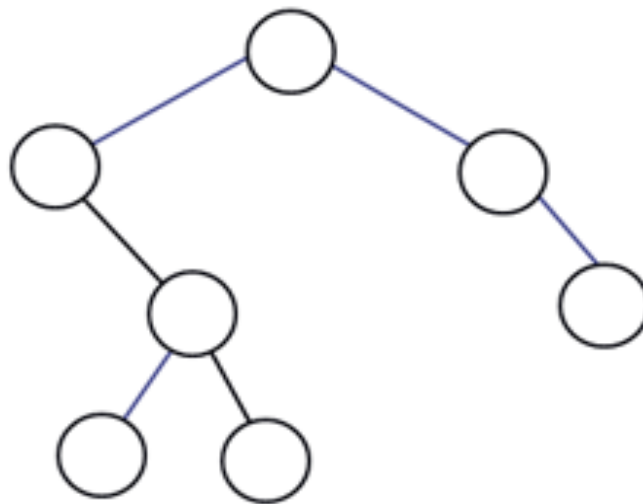
- AVL 트리는 트리가 한쪽으로 치우쳐 자라나는 현상을 방지하여 트리 높이의 균형(Balance)을 유지하는 이진탐색트리
- 균형(Balanced) 이진트리를 만들면 N개의 노드를 가진 트리의 높이가  $O(\log N)$ 이 되어 탐색, 삽입, 삭제 연산의 수행시간이  $O(\log N)$ 으로 보장
- **[핵심 아이디어] AVL트리는 삽입이나 삭제로 인해 균형이 깨지면 회전 연산을 통해 트리의 균형을 유지한다**

# AVL트리

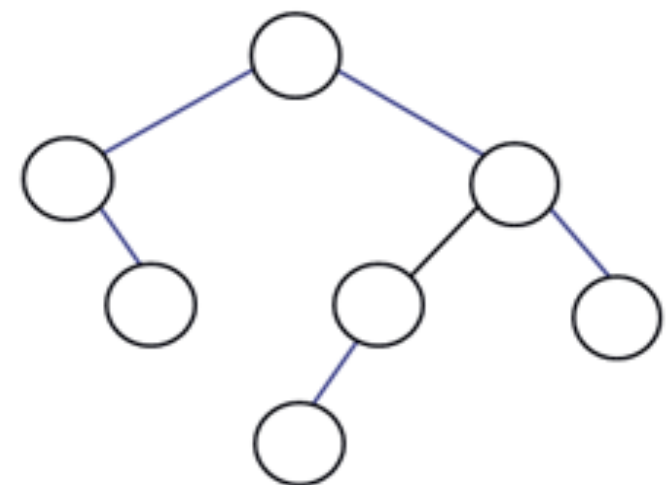
- AVL트리는 임의의 노드  $x$ 에 대해  $x$ 의 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 차이가 1을 넘지 않는 이진탐색트리이다.



(a)



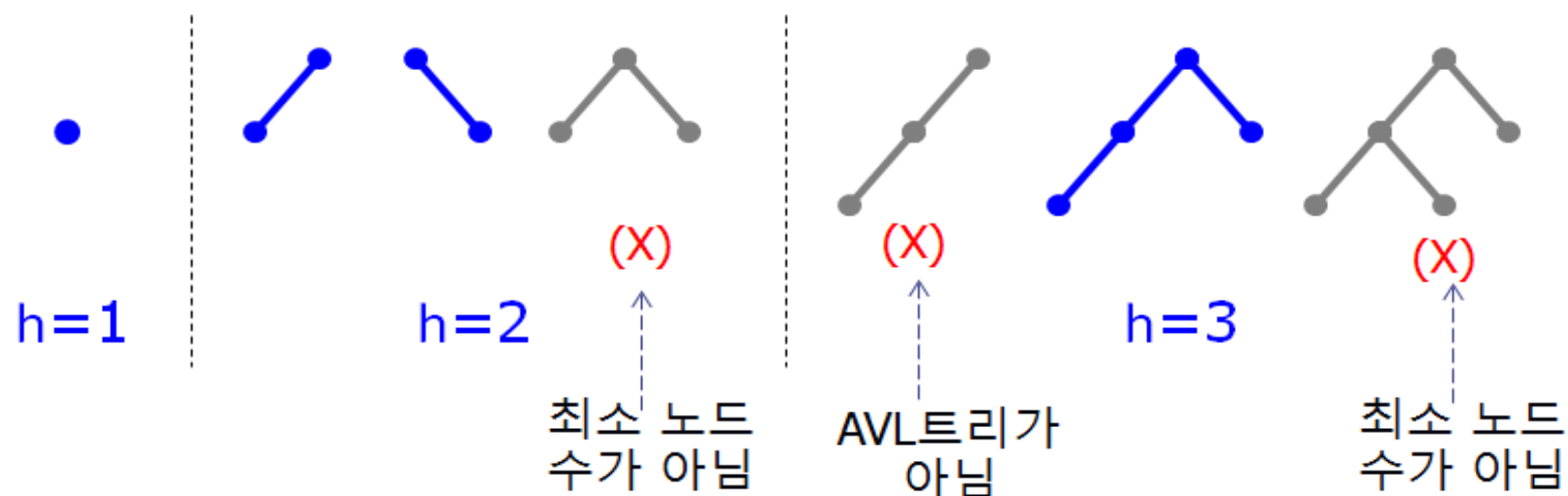
(b)



(c)

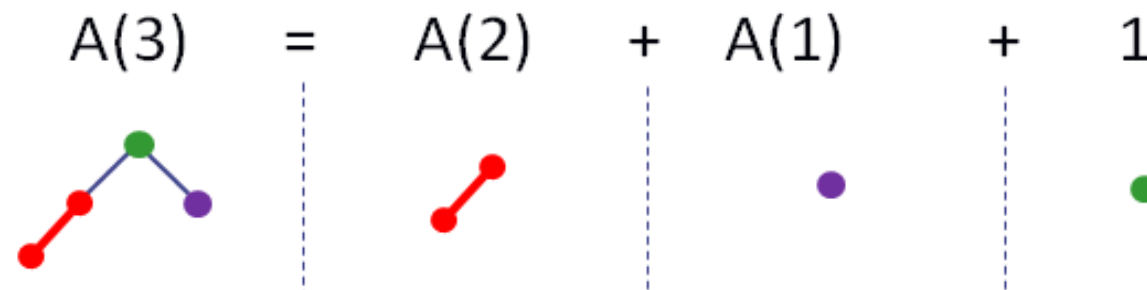
# AVL 트리

- [정리]  $N$ 개의 노드를 가진 AVL 트리의 높이는  $O(\log N)$ 이다.
- [증명]  $A(h)$  = 높이가  $h$ 인 AVL 트리를 구성하는 최소의 노드 수  
 $A(1) = 1, A(2) = 2, A(3) = 4$ 이다.



# AVL 트리

- $A(3)$ 을 재귀적으로 표현해보면



- $A(3)$ 이 위와 같이 구성되는 이유:
  - 높이가 3인 AVL 트리에는 루트와 루트의 왼쪽 서브트리와 오른쪽 서브트리가 존재해야 하고,
  - 각 서브트리 역시 최소 노드 수를 가진 AVL 트리여야 하므로
  - 또한 이 두 개의 서브트리의 높이 차이가 1일 때 전체 트리의 노드 수가 최소가 되기 때문



# AVL 트리

- 이를  $A(h)$ 에 대한 식으로 표현하면

$$A(h) = A(h-1) + A(h-2) + 1, \text{ 단, } A(0)=0, A(1)=1, A(2)=2$$

h	0	1	2	3	4	5	6	7	
A(h)	0	1	2	4	7	12	20	33	...
F(h)	0	1	1	2	3	5	8	13	...

- $A(h)$ 와 피보나치 수  $F(h)$ 와의 관계

$$A(h) = F(h+2) - 1$$

# AVL 트리

- 피보나치 수  $F(h) \approx \frac{\varphi^h}{\sqrt{5}}$  이므로,  $\varphi = (1 + \sqrt{5})/2$  이므로

$$A(h) \approx \frac{\varphi^{h+2}}{\sqrt{5} - 1}$$

- $A(h)$  = 높이가  $h$ 인 AVL트리에 있는 최소 노드 수이므로, 노드 수가  $N$ 인 임의의 AVL트리의 최대 높이를  $A(h) \leq N$ 의 관계에서 다음과 같이 계산할 수 있다.

$$A(h) \approx \phi^{h+2}/\sqrt{5} - 1 \leq N$$

$$\phi^{h+2} \leq \sqrt{5} (N + 1)$$

$$h \leq \log_{\phi}(\sqrt{5}(N+1)) - 2 \approx 1.44\log N = O(\log N). \square$$

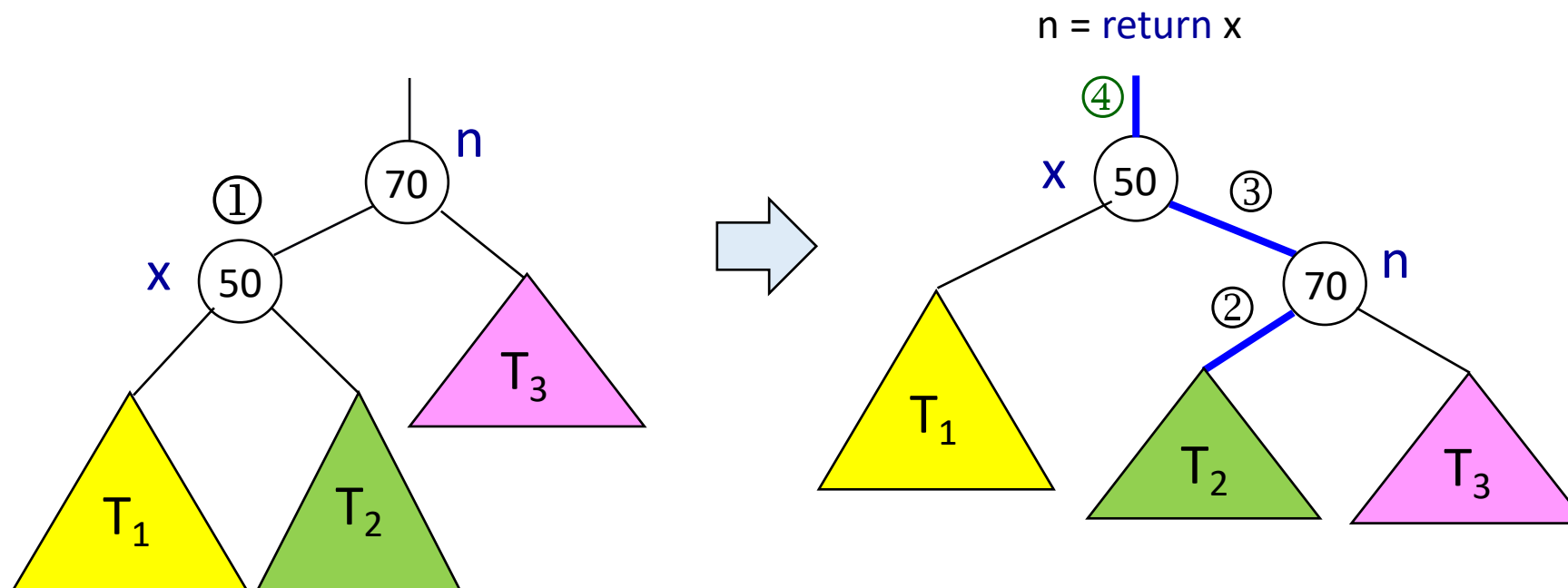
# AVL 트리의 회전 연산

- AVL 트리에서 삽입 또는 삭제 연산을 수행할 때 트리의 균형을 유지하기 위해 LL-회전, RR-회전, LR-회전, RL-회전 연산 사용
- 회전 연산은 2 개의 기본적인 연산으로 구현

# AVL 트리의 회전 연산:

## rotate\_right(n)

- rotate\_right(): 왼쪽 방향의 서브트리가 높아서 불균형이 발생할 때 서브트리를 오른쪽 방향으로 회전
- 노드 n의 왼쪽 자식 x를 노드 n의 자리로 옮기고, 노드 n을 노드 x의 오른쪽 자식으로 만들며, 이 과정에서 서브트리 T2가 노드 n의 왼쪽 서브트리로 이동



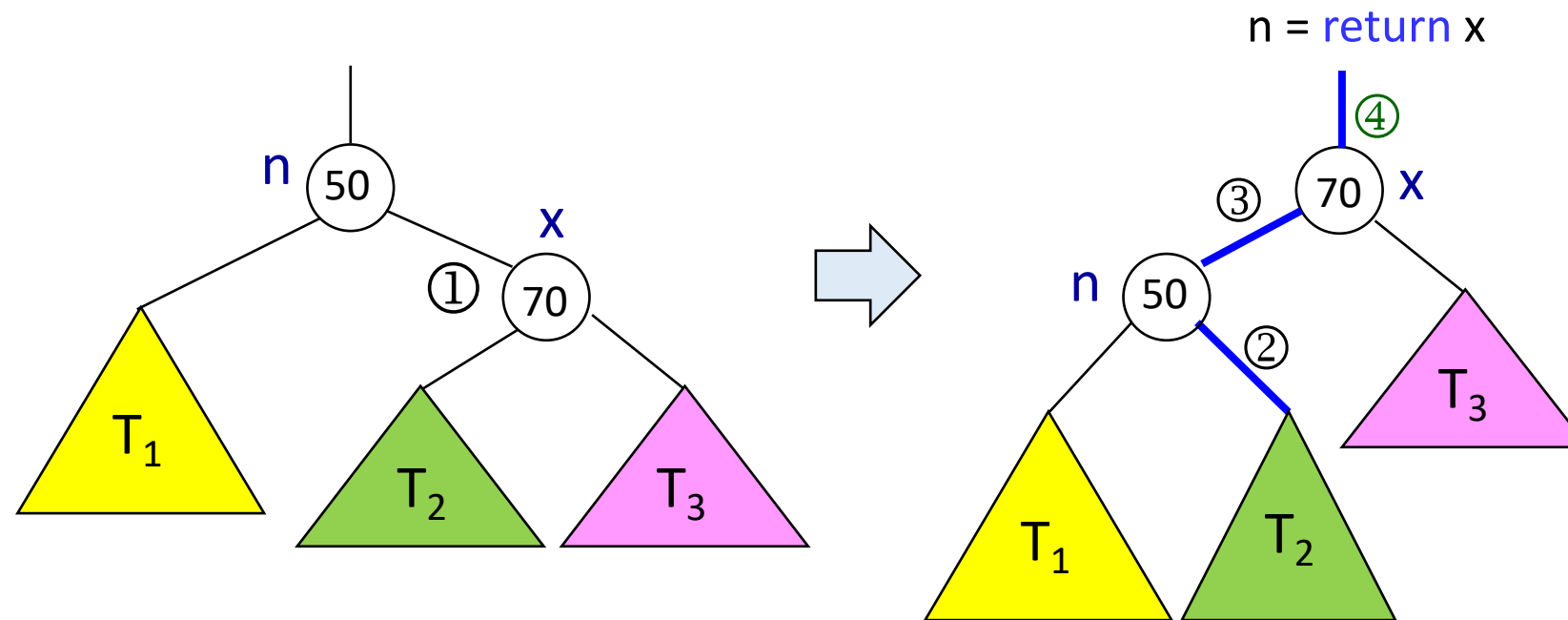
# AVL 트리의 회전 연산:

## rotate\_right(n)

- rotate\_right(): 왼쪽 방향의 서브트리가 높아서 불균형이 발생할 때 서브트리를 오른쪽 방향으로 회전
- 노드 n의 왼쪽 자식 x를 노드 n의 자리로 옮기고, 노드 n을 노드 x의 오른쪽 자식으로 만들며, 이 과정에서 서브트리 T2가 노드 n의 왼쪽 서브트리로 이동

```
01 def rotate_right(self, n): # 우로 회전
02     ① x = n.left
03     ② n.left = x.right
04     ③ x.right = n
05     n.height = max(self.height(n.left), self.height(n.right)) + 1
06     x.height = max(self.height(x.left), self.height(x.right)) + 1
07     ④ return x
```

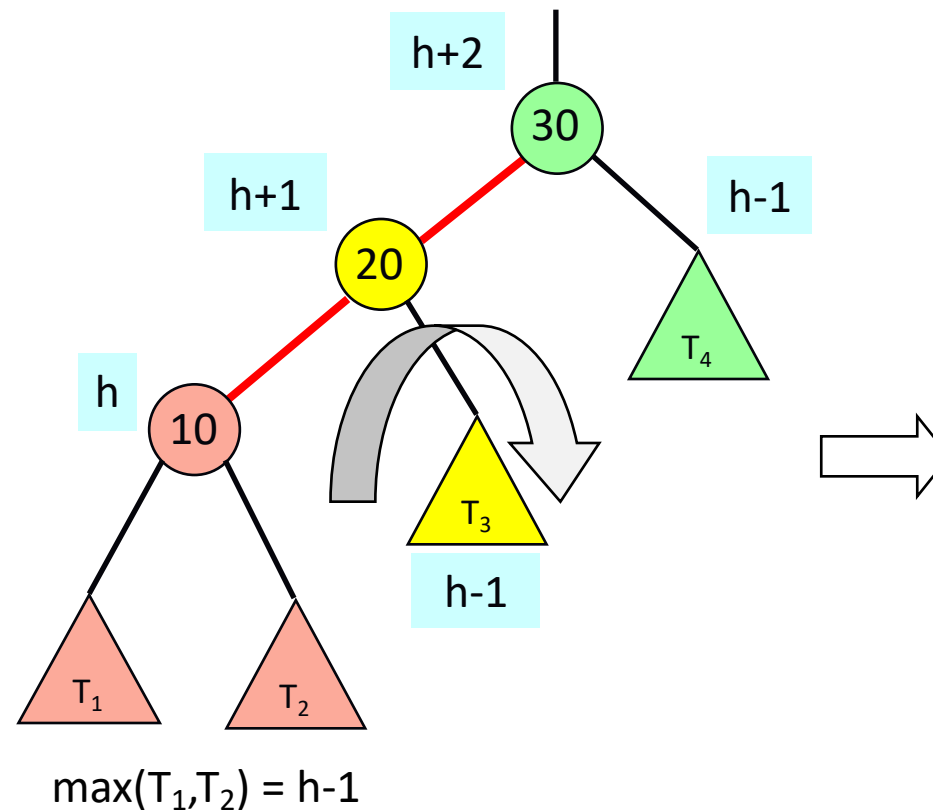
# AVL 트리의 회전 연산: rotate\_left(n)



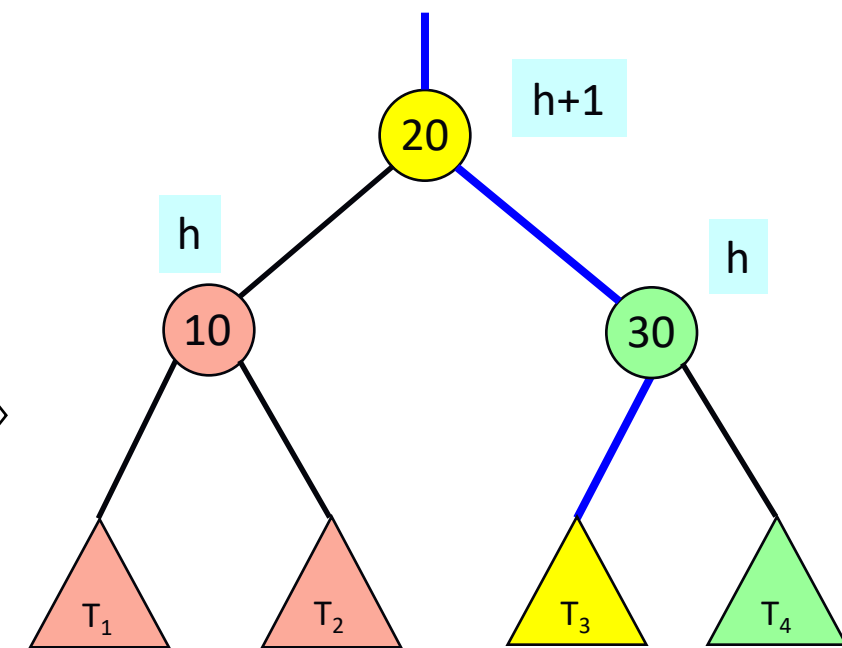
```
01 def rotate_left(self, n): # 좌로 회전
02     ① x = n.right
03     ② n.right = x.left
04     ③ x.left = n
05     n.height = max(self.height(n.left), self.height(n.right)) + 1
06     x.height = max(self.height(x.left), self.height(x.right)) + 1
07     ④ return x
```

# LL-회전

- (a) 노드 10의 왼쪽 서브트리(T1) 또는 오른쪽 서브트리(T2)에 새로운 노드 삽입
  - T1 또는 T2의 높이 =  $h-1$
  - 노드 30의 왼쪽과 오른쪽 서브트리의 높이 차이 = 2
  - 노드 30의 왼쪽(L) 서브트리의 왼쪽(L) 서브트리에 새로운 노드가 삽입되었기 때문
- (b)
  - 20이 30의 자리로 이동
  - 30을 20의 오른쪽 자식으로
  - T3은 30의 왼쪽 자식으로
  - T3에 있는 키들은 20과 30 사이 값을 가지므로 T3의 이동 전후 모두 이진탐색트리 조건이 만족



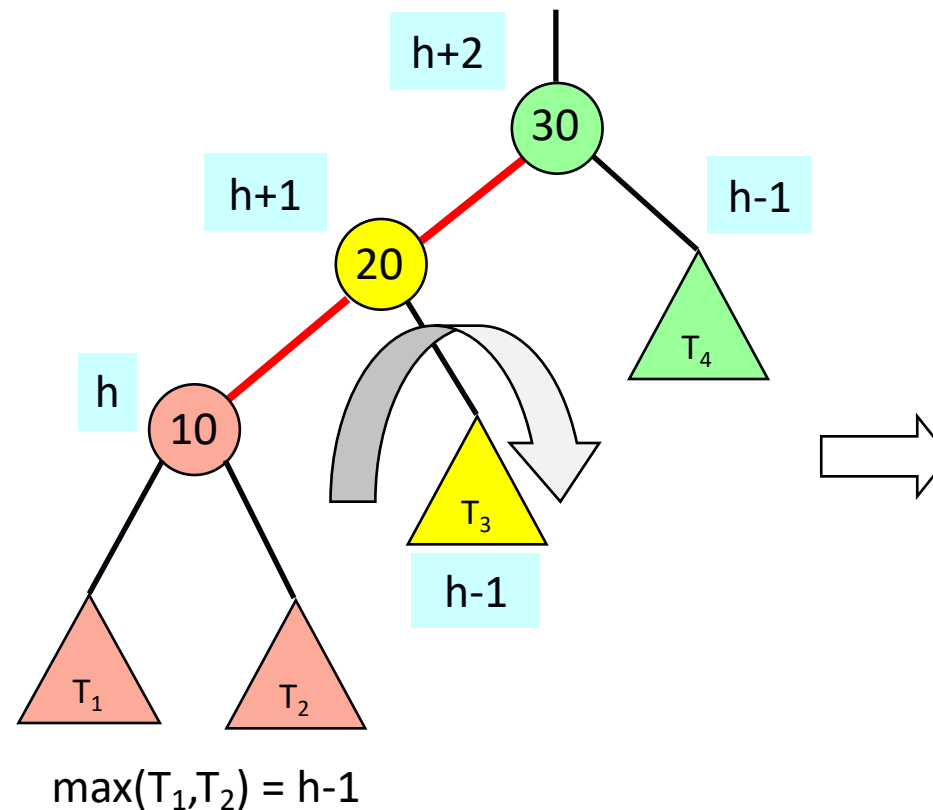
(a)  $T_1$  또는  $T_2$ 에 새 노드 삽입



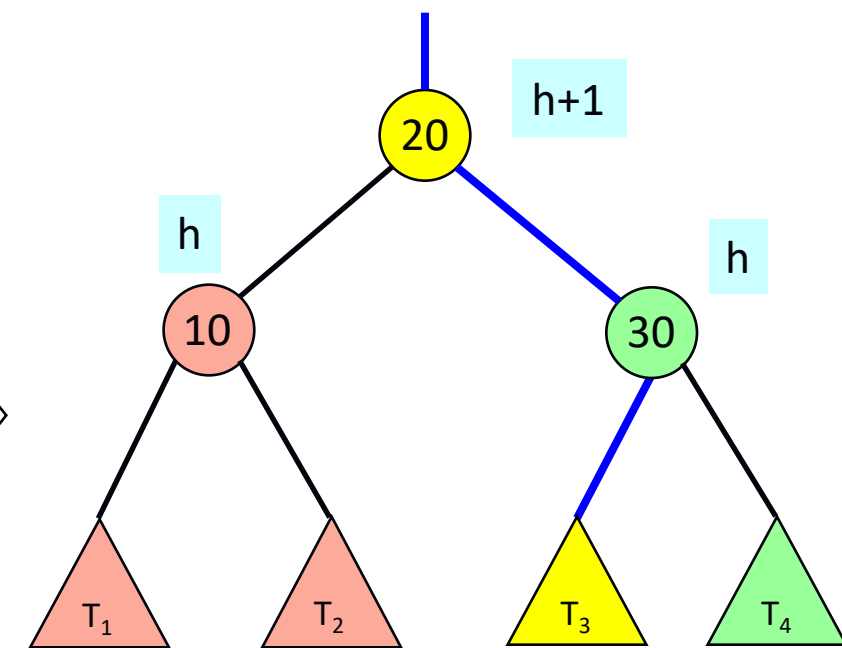
(b) LL-회전 후

# LL-회전

- (a) 노드 10의 왼쪽 서브트리( $T_1$ ) 또는 오른쪽 서브트리( $T_2$ )에 새로운 노드 삽입
  - $T_1$  또는  $T_2$ 의 높이 =  $h-1$
  - 노드 30의 왼쪽과 오른쪽 서브트리의 높이 차이 = 2
  - 노드 30의 왼쪽(L) 서브트리의 왼쪽(L) 서브트리에 새로운 노드가 삽입되었기 때문
- (b)
  - 20이 30의 자리로 이동
  - 30을 20의 오른쪽 자식으로
  - $T_3$ 은 30의 왼쪽 자식으로
  - $T_3$ 에 있는 키들은 20과 30 사이 값을 가지므로  $T_3$ 의 이동 전후 모두 이진탐색트리 조건이 만족

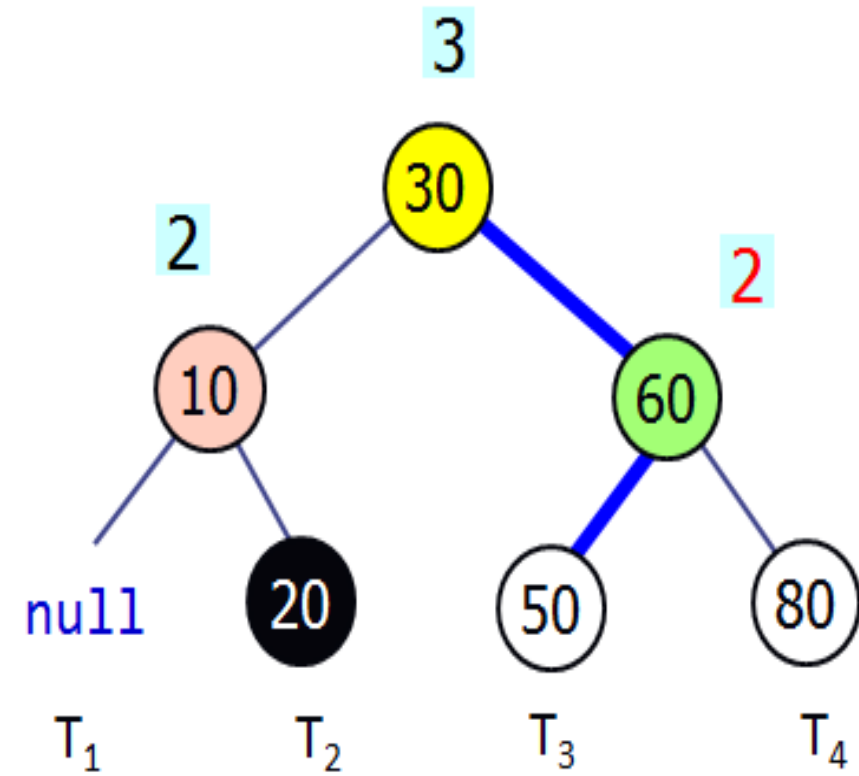
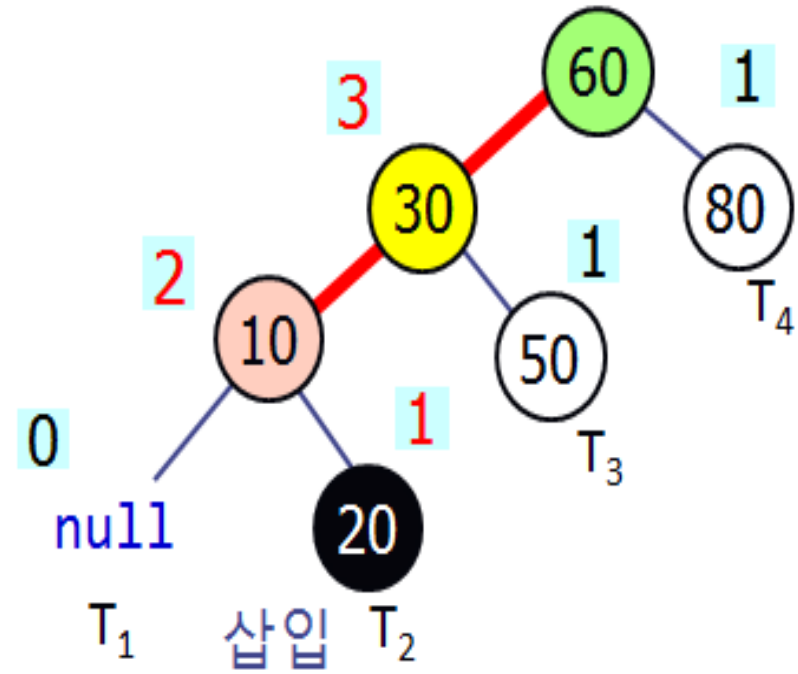


(a)  $T_1$  또는  $T_2$ 에 새 노드 삽입



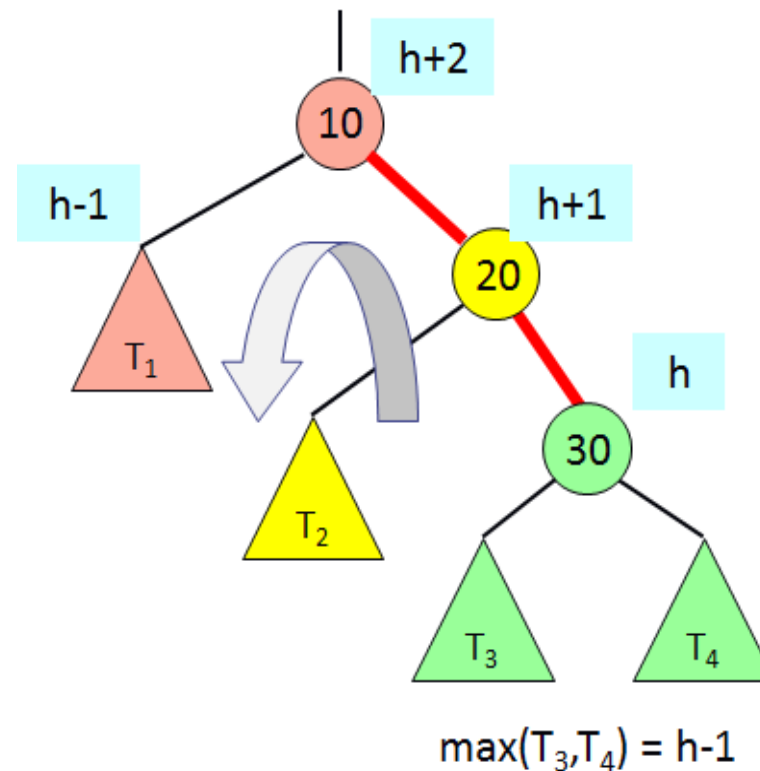


# LL-회전

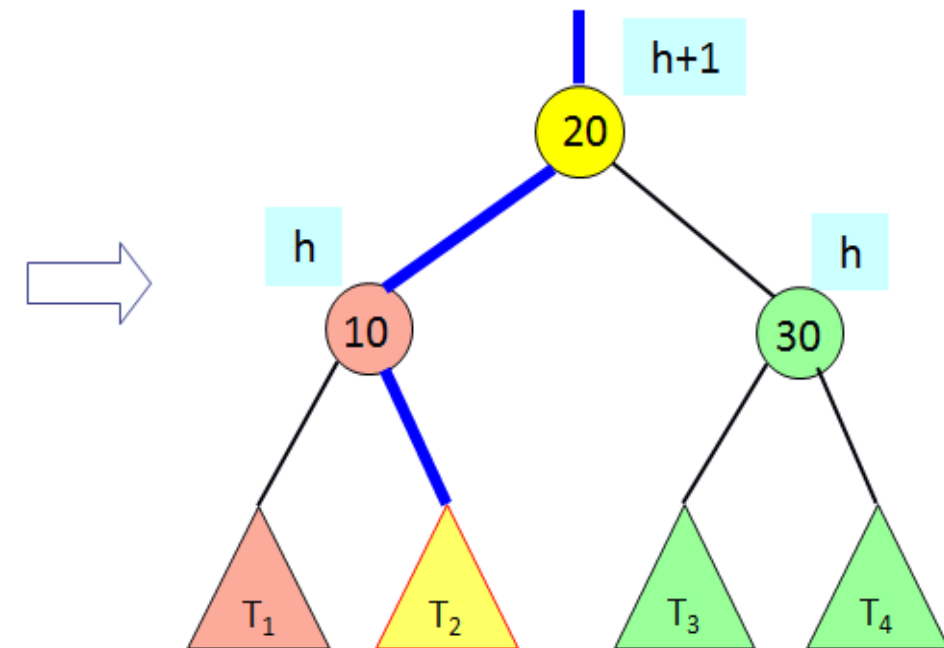


# RR-회전

- (a) 30의 왼쪽 서브트리(T3) 또는 오른쪽 서브트리(T4)에 새로운 노드 삽입
  - T3 또는 T4의 높이 =  $h-1$
  - 노드 10의 왼쪽과 오른쪽 서브트리의 높이 차이 = 2
  - 노드 10의 오른쪽(R) 서브트리의 오른쪽(R) 서브트리에 새로운 노드가 삽입되었기 때문
- (b)
  - 20이 10의 자리로 이동
  - 10을 20의 왼쪽 자식으로
  - T2는 10의 오른쪽 자식으로
  - T2에 있는 키들은 10과 20 사이 값일 것이므로 T2의 이동 전후 모두 이진탐색트리 조건이 만족
- RR-회전은 rotate\_left() 사용

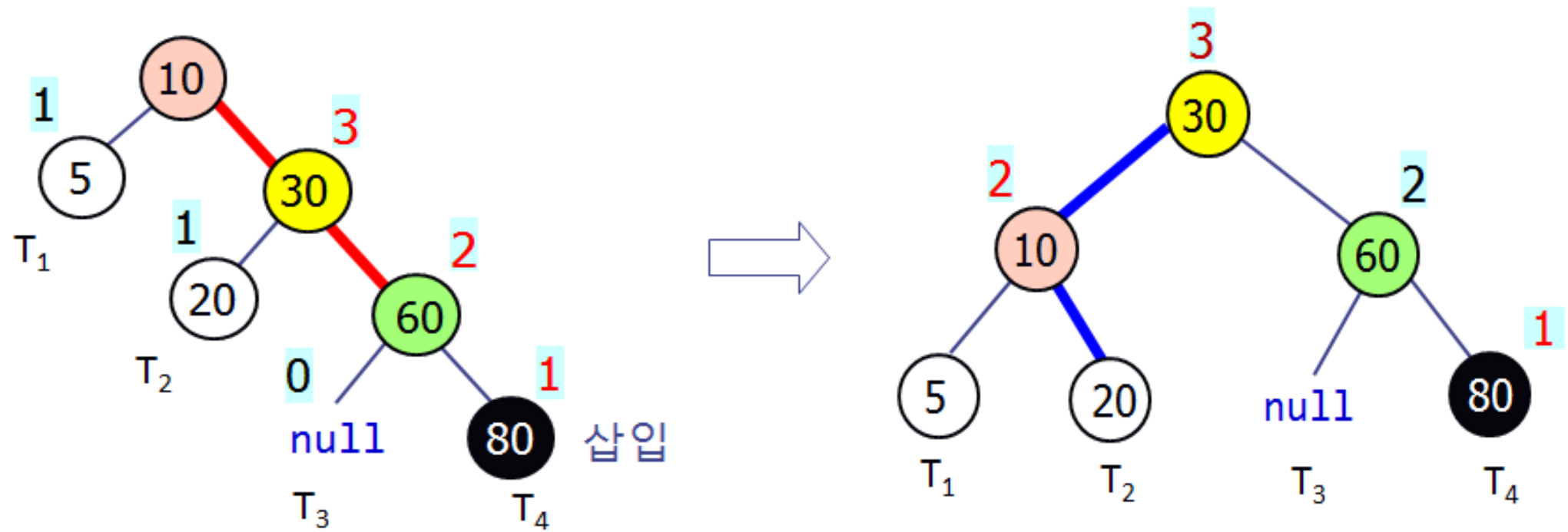


(a) T<sub>3</sub> 또는 T<sub>4</sub>에 새 노드 삽입



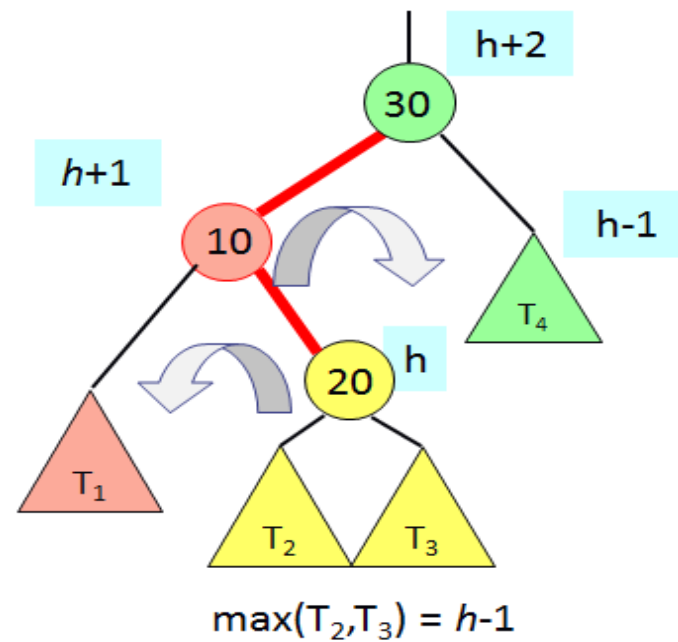
(b) RR-회전 후

# RR-회전

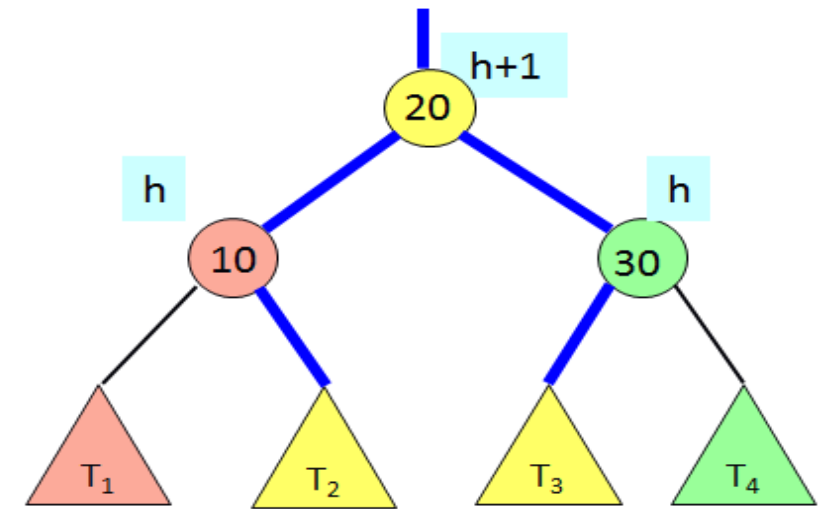


# LR-회전

- (a) 20의 왼쪽 서브트리(T2) 또는 오른쪽 서브트리(T3)에 새로운 노드가 삽입 되어 T2 또는 T3의 높이가  $h-1$ 이 됨에 따라 30의 왼쪽과 오른쪽 서브트리의 높이 차이가 2가 된 상태
- 30의 왼쪽(L) 서브트리의 오른쪽(R) 서브트리에서 새로운 노드가 삽입되었기 때문
- LR-회전은 `rotate-left(10)` 수행 후 `rotate_right(30)` 수행

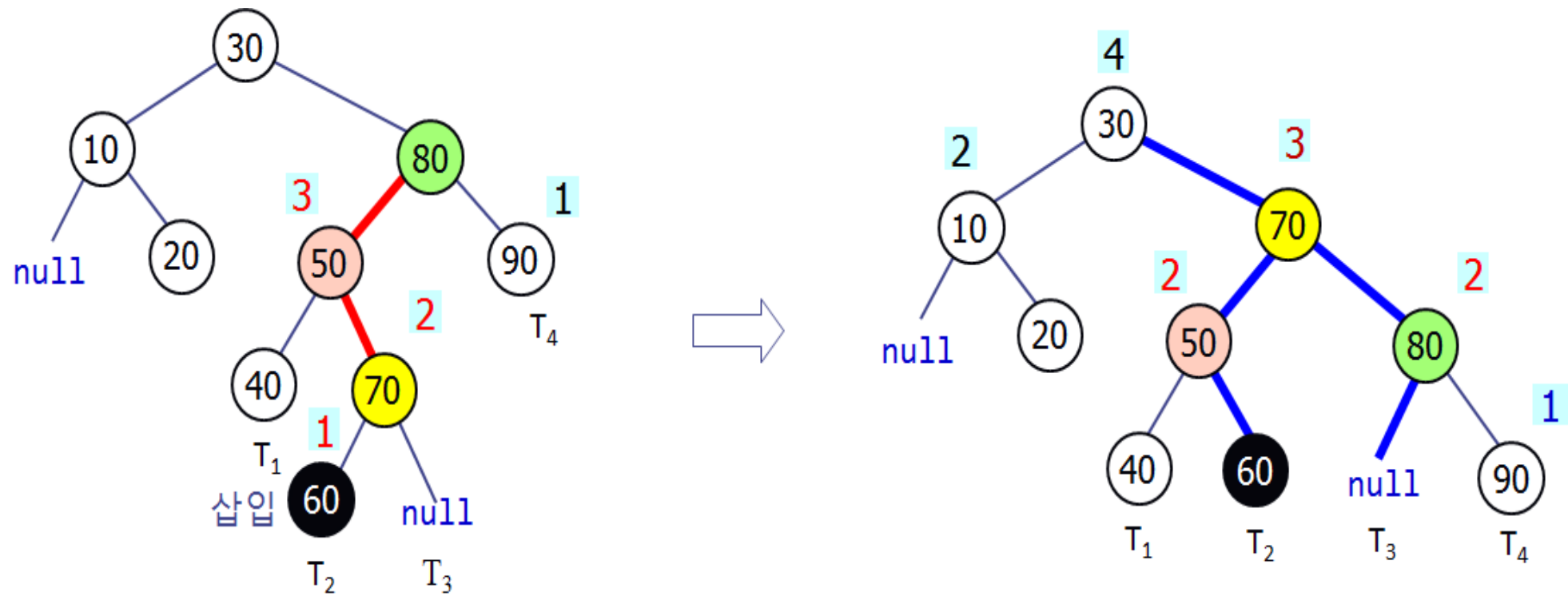


(a)  $T_2$  또는  $T_3$ 에 새 노드 삽입



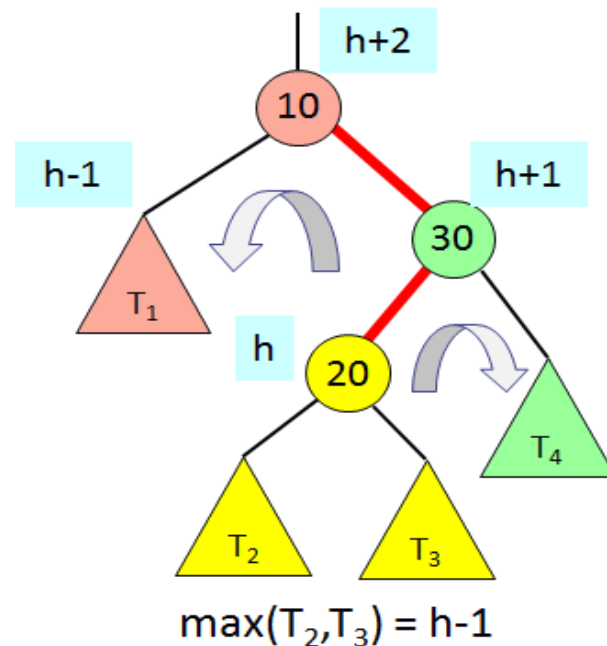
(b) LR-회전 후

# LR-회전

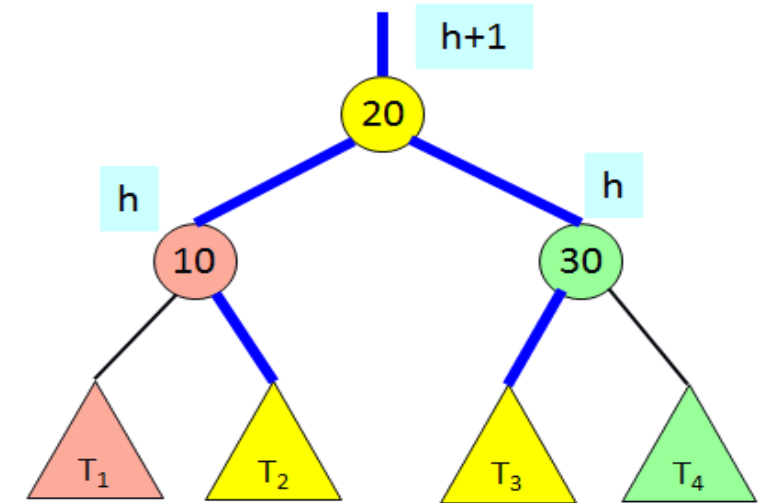


# RL-회전

- (a) 20의 왼쪽 서브트리 (T2) 또는 오른쪽 서브트리 (T3)에 새로운 노드가 삽입되어 T2 또는 T3의 높이가  $h-1$ 이 되고 10의 왼쪽과 오른쪽 서브트리의 높이 차이가 2가 된 상태
- 10의 오른쪽(R) 서브트리의 왼쪽(L) 서브트리에서 새로운 노드가 삽입되었기 때문
- RL-회전은 `rotate_right(30)` 수행한 후 `rotate_left(10)` 수행

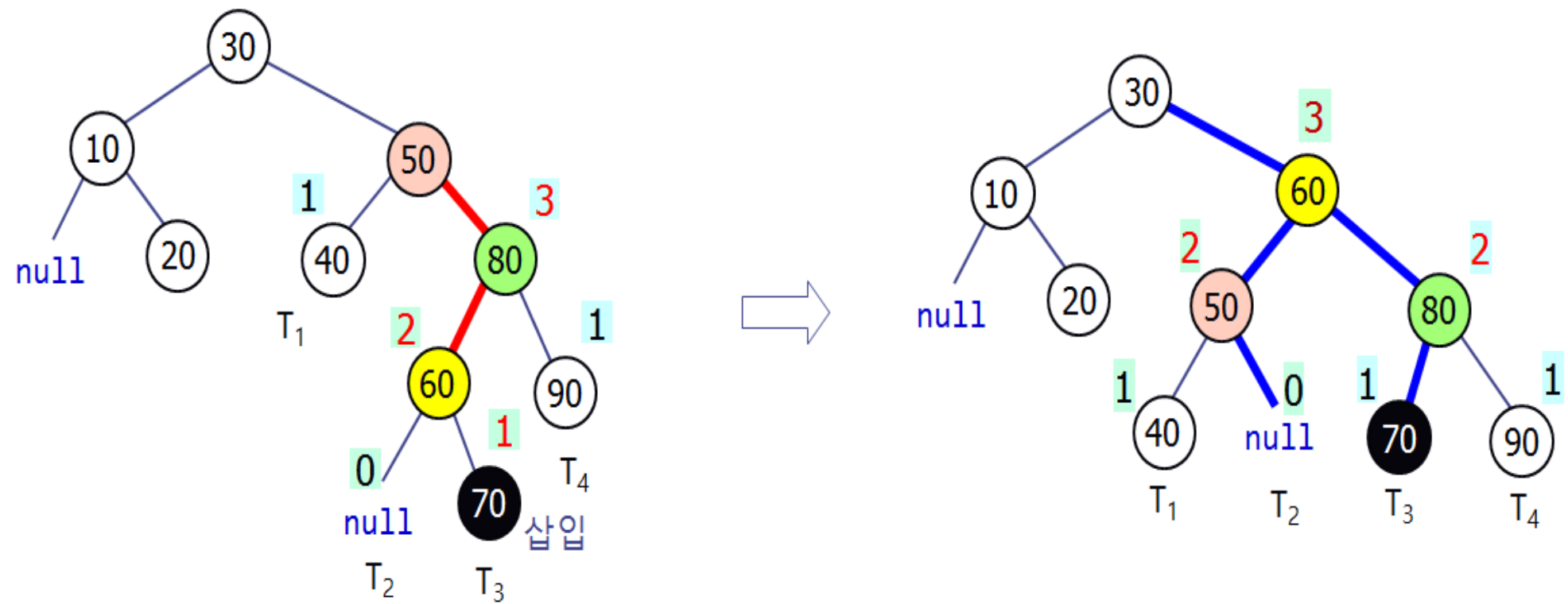


(a)  $T_2$  또는  $T_3$ 에 새 노드 삽입



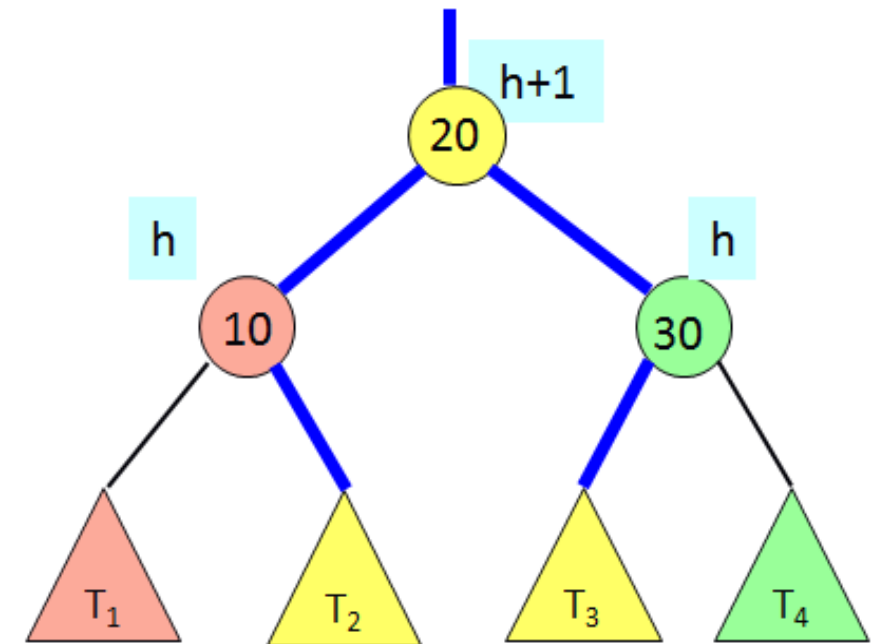
(b) RL-회전 후

# RL-회전



# 4가지 회전의 공통점

- 회전 후의 트리들이 모두 동일
  - 각 그림(a)의 트리에서 10, 20, 30이 어디에 위치하든지, 3개의 노드들 중에서 중간값을 가진 노드, 즉, 20이 위로 이동하면서 10 과 30이 각각 20의 좌우 자식이 되기 때문
- 각 회전 연산의 수행시간이  $O(1)$ 
  - 각 그림(b)에서 변경된 노드 레퍼런스 수가  $O(1)$  개이기 때문





# AVL 트리의 삽입연산

- AVL트리에서의 삽입은 두 단계로 수행
- [1 단계] 이진탐색트리의 삽입과 동일하게 새로운 노드 삽입
- [2 단계] 새로 삽입한 노드로부터 루트로 거슬러 올라가며 각 노드의 서브트리 높이 차이를 갱신
- 이 때 가장 먼저 불균형이 발생한 노드를 발견하면, 이 노드를 기준으로 새 노드가 어디에 삽입되었는지에 따라 적절한 회전연산을 수행

# AVL 트리의 삽입연산

```
01 class Node:
02     def __init__(self, key, value, height, left=None, right=None):
03         self.key = key
04         self.value = value
05         self.height = height
06         self.left = left
07         self.right = right
08
09 class AVL:
10     def __init__(self):
11         self.root = None
12
13     def height(self, n):
14         if n == None:
15             return 0
16         return n.height
17
18     def put(self, key, value): # 삽입 연산
19     def balance(self, n): # 불균형 처리
20     def bf(self, n): # bf 계산
21     def rotate_right(self, n): # 우로 회전
22     def rotate_left(self, n): # 좌로 회전
23     def delete(self, key): # 삭제 연산
24     def delete_min(self): # 최솟값 삭제
25     def min(self): # 최솟값 찾기
```

노드 생성자  
key, value, 노드의 높이,  
왼쪽, 오른쪽 자식노드 레퍼런스

트리 루트

노드 n의 높이 리턴

삭제 및 삭제 관련 연산

# AVL 트리의 삽입연산

bf(n): (노드 n의 왼쪽 서브트리 높이) - (오른쪽 서브트리 높이) 리턴

```
01 def bf(self, n): # bf 계산
02     return self.height(n._left) - self.height(n._right)
```

```
01 def balance(self, n): # 불균형 처리
02     if self.bf(n) > 1:
03         if self.bf(n._left) < 0:
04             n._left = self.rotate_left(n._left)
05             n = self.rotate_right(n)
06
07     elif self.bf(n) < -1:
08         if self.bf(n._right) > 0:
09             n._right = self.rotate_right(n._right)
10             n = self.rotate_left(n)
11     return n
```

노드 n에서 불균형 발생

노드 n의 왼쪽자식의 오른쪽 서브트리가 높은 경우

LR 회전

LL 회전

노드 n의 오른쪽자식의 왼쪽 서브트리가 높은 경우

RL 회전

RR 회전

```

01 def balance(self, n): # 불균형 처리
02     if self.bf(n) > 1:
03         if self.bf(n._left) < 0:
04             n._left = self.rotate_left(n._left)
05             n = self.rotate_right(n)
06
07     elif self.bf(n) < -1:
08         if self.bf(n._right) > 0:
09             n._right = self.rotate_right(n._right)
10             n = self.rotate_left(n)
11     return n

```

노드 n에서 불균형 발생

노드 n의 왼쪽자식의 오른쪽 서브트리가 높은 경우

LR 회전

LL 회전

노드 n의 오른쪽자식의 왼쪽 서브트리가 높은 경우

RL 회전

RR 회전

- `balance()`에서 line 02의 `bf(n) > 1`인 경우는 노드 n의 왼쪽 서브트리가 오른쪽 서브트리보다 높고, 그 차이가 1보다 큰 것으로 불균형 발생
- 이 때 `bf(n.left)`가 음수이면, n.left의 오른쪽 서브트리가 왼쪽 서브트리보다 높음
  - Line 04에서 `rotate_left(n.left)`를 수행하고 line 06에서 `rotate_right(n)`을 수행. 즉, LR-회전 수행
- `bf(n.left)`가 음수가 아니라면, line 06에서 LL-회전 만을 수행
- RR-회전과 RL-회전도 line 08~10에 따라 각각 수행되어 트리의 균형을 유지
- 참고로 현재 노드 n의 균형이 유지되어 있으면, 바로 line 11에서 노드 n의 레퍼런스를 리턴

# AVL 트리의 삽입연산

```
01 def put(self, key, value): # 삽입 연산
02     self.root = self.put_item(self.root, key, value)
03
04 def put_item(self, n, key, value):
05     if n == None:
06         return Node(key, value, 1)
07     if n.key > key:
08         n.left = self.put_item(n.left, key, value)
09     elif n.key < key:
10         n.right = self.put_item(n.right, key, value)
11     else:
12         n.value = value
13         return n
14     n.height = max(self.height(n.left), self.height(n.right)) + 1
15     return self.balance(n)
```

새 노드 생성,  
높이=1

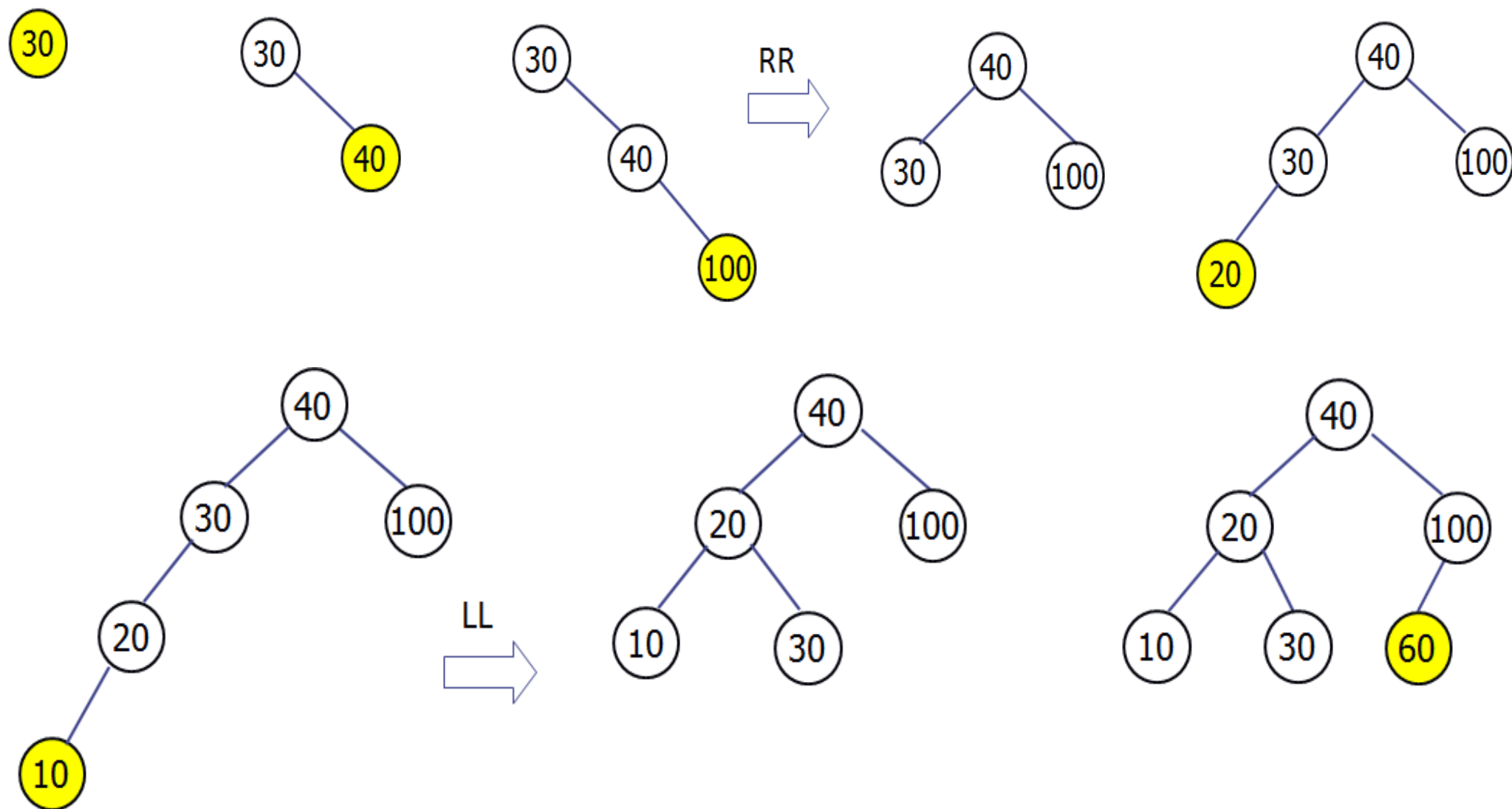
key가 이미 있으면  
value만 갱신

노드 n의 높이 갱신

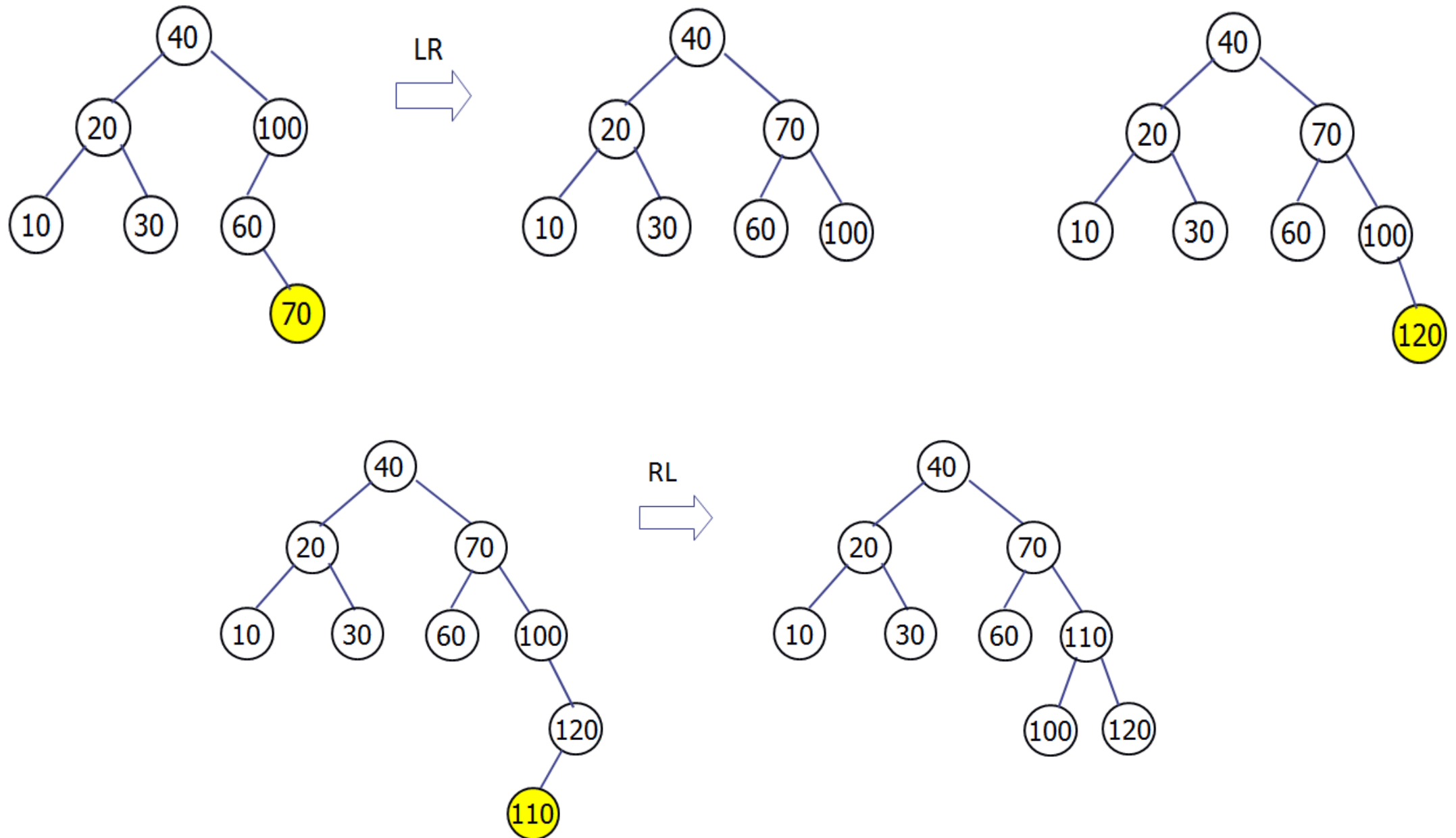
노드 n의 균형 유지

# AVL 트리의 삽입연산

- [예제] 30, 40, 100, 20, 10, 60, 70, 120, 110을 순차적으로 삽입



# AVL 트리의 삽입연산

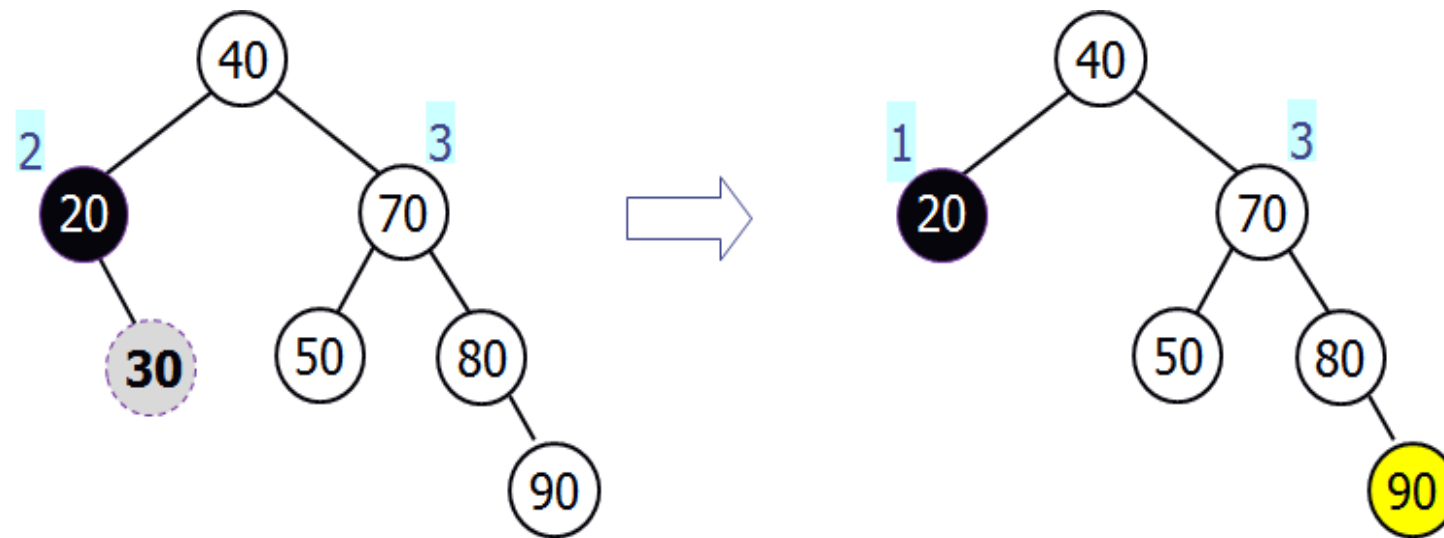


# AVL 트리의 삭제 연산

- AVL트리에서의 삭제는 두 단계로 진행
- [1단계] 이진탐색트리에서와 동일한 삭제 연산 수행
- [2단계] 삭제된 노드로부터 루트노드 방향으로 거슬러 올라가며 불균형이 발생한 경우 적절한 회전 연산 수행
- 회전 연산 수행 후에 부모에서 불균형이 발생할 수 있고, 이러한 일이 반복되어 루트에서 회전 연산을 수행해야 하는 경우도 발생



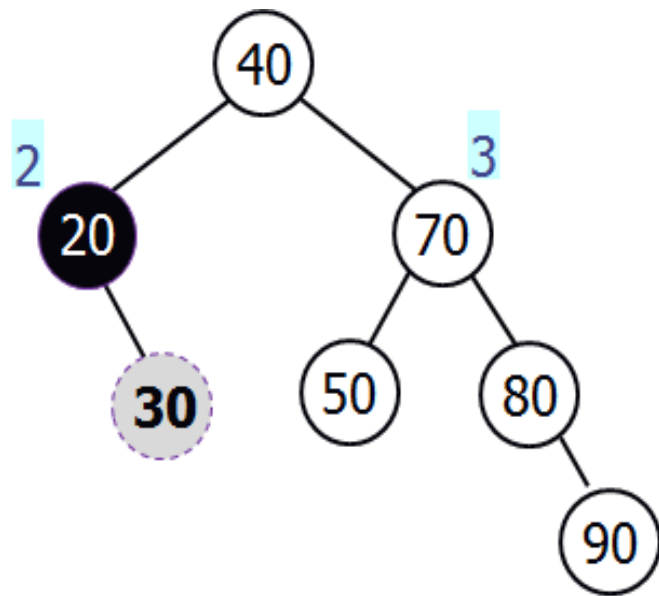
# AVL 트리의 삭제 연산



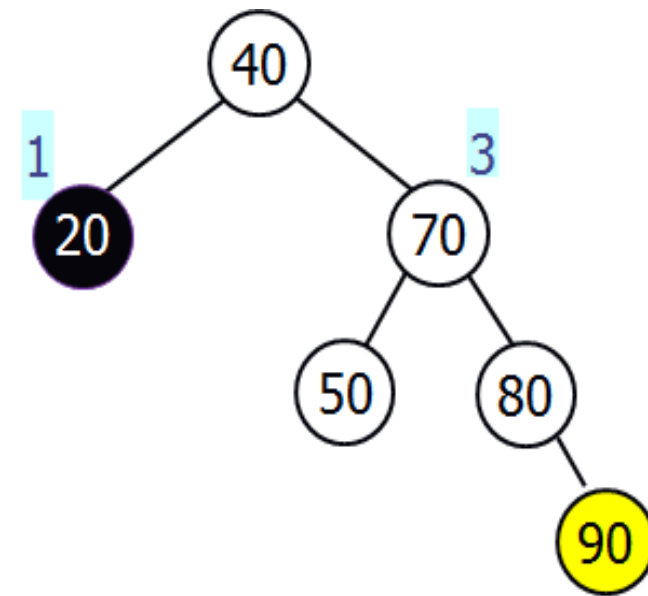
(a) 삭제 전

(b) 삭제 후 노드 40에서 불균형 발생

# AVL 트리의 삭제 연산



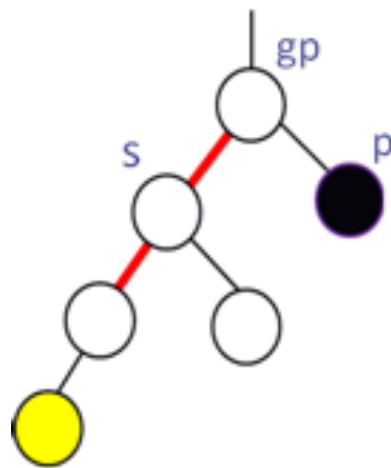
(a) 삭제 전



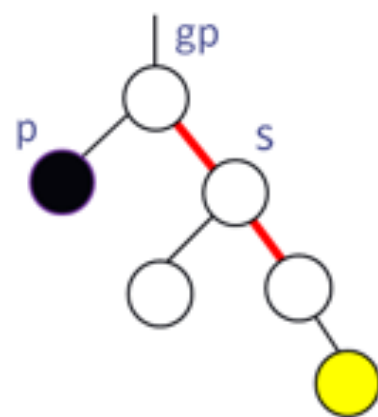
(b) 삭제 후 노드 40에서 불균형 발생

# AVL 트리의 삭제 연산

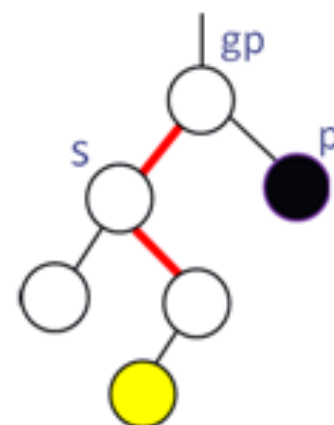
- [핵심 아이디어] 삭제 후 불균형이 발생하면 반대쪽에 삽입이 이루어져 불균형이 발생한 것으로 취급하자
- 삭제된 노드의 부모 =  $p$ ,  $p$ 의 부모 =  $gp$ ,  $p$ 의 형제 =  $s$
- $s$ 의 왼쪽과 오른쪽 서브트리 중에서 높은 서브트리에 마치 새 노드가 삽입된 것으로 간주



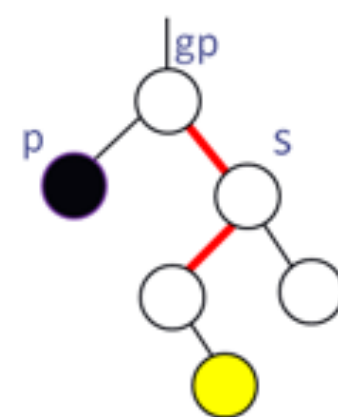
(a) LL-회전



(b) RR-회전



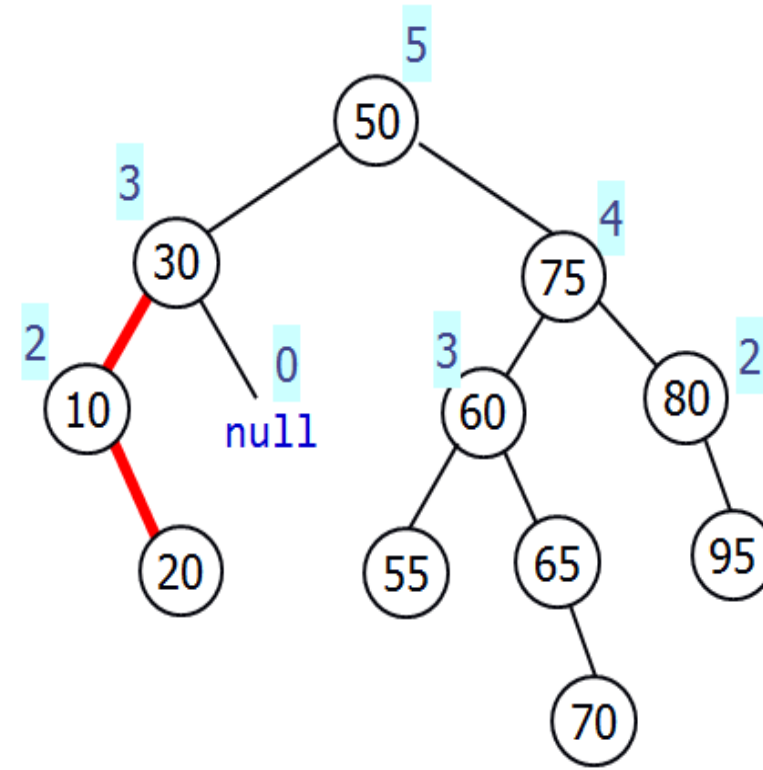
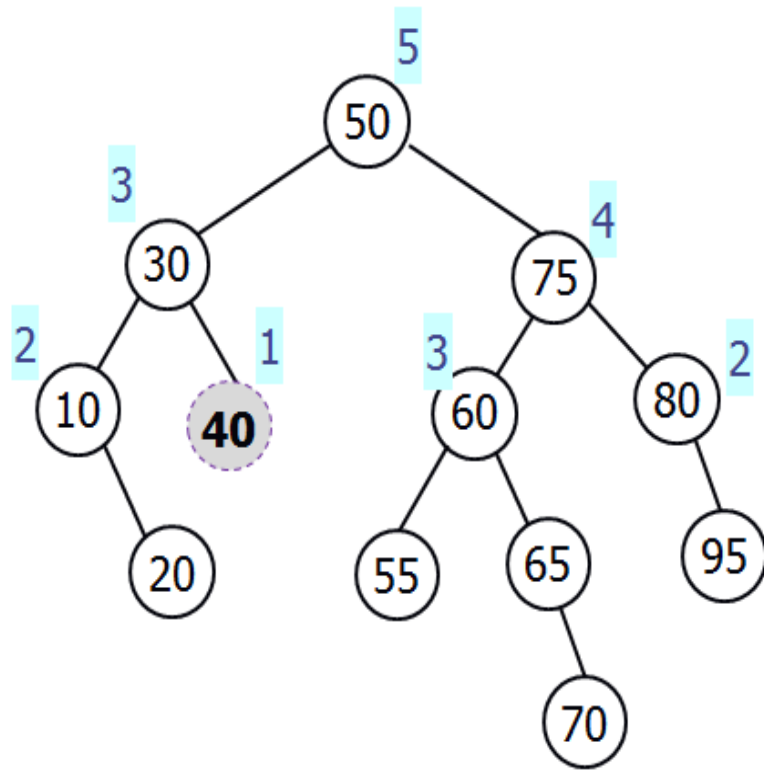
(c) LR-회전



(d) RL-회전

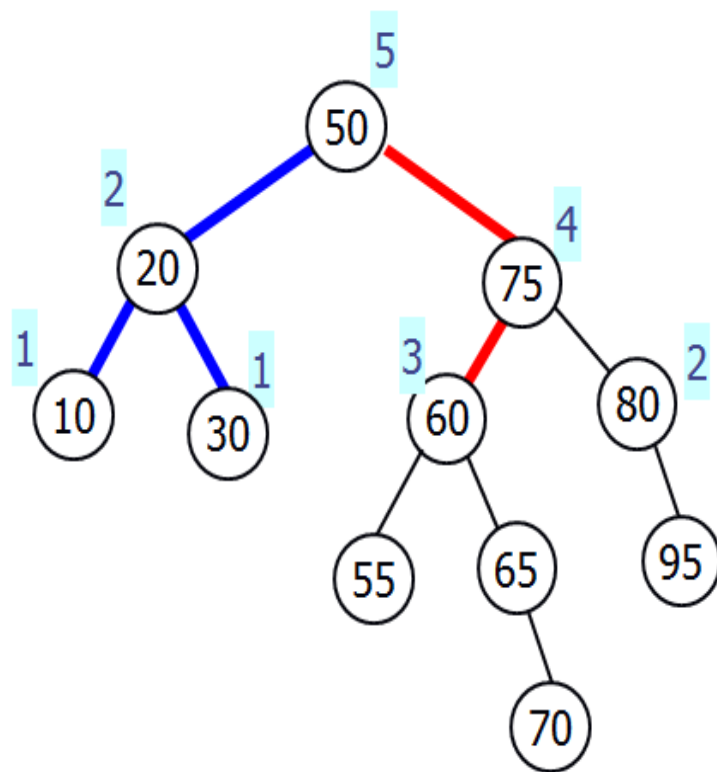
# AVL 트리의 삭제 연산

- 예제: 40을 삭제

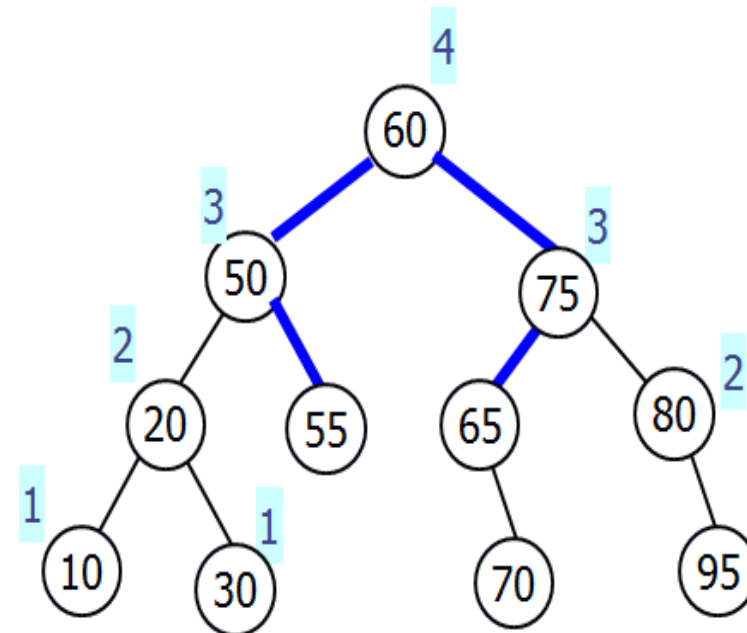


# AVL 트리의 삭제 연산

- 예제: 40을 삭제



LR-회전 후



RL-회전 후

# AVL Tree 테스트

```
01 from avl import AVL
02 if __name__ == '__main__':
03     t = AVL()
04     t.put(75, 'apple')
05     t.put(80, 'grape')
06     t.put(85, 'lime')
07     t.put(20, 'mango')
08     t.put(10, 'strawberry')
09     t.put(50, 'banana')
10     t.put(30, 'cherry')
11     t.put(40, 'watermelon')
12     t.put(70, 'melon')
13     t.put(90, 'plum')
14     print('전위순회:\t', end='')
15     t.preorder(t.root)
16     print('\n중위순회:\t', end='')
17     t.inorder(t.root)
18     print('\n75와 85 삭제 후:')
19     t.delete(75)
20     t.delete(85)
21     print('전위순회:\t', end='')
22     t.preorder(t.root)
23     print('\n중위순회:\t', end='')
24     t.inorder(t.root)
```

AVL 트리 객체 생성

10개의 항목 삽입

트리 순회 및 삭제 연산 수행

Console PyUnit

<terminated> main.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32]

전위순회: 75 40 20 10 30 50 70 85 80 90

중위순회: 10 20 30 40 50 70 75 80 85 90

75와 85 삭제 후:

전위순회: 40 20 10 30 80 50 70 90

중위순회: 10 20 30 40 50 70 80 90

# 수행시간

- AVL 트리에서의 탐색, 삽입, 삭제 연산은 공통적으로 루트부터 탐색을 시작하여 최악의 경우에 이파리까지 내려가고, 삽입이나 삭제 연산은 다시 루트까지 거슬러 올라가야
- 트리를 한 층 내려갈 때는 재귀호출하며, 한 층을 올라갈 때 불균형이 발생하면 적절한 회전 연산을 수행하는데, 이들 각각은  $O(1)$  시간 밖에 걸리지 않음
- 탐색, 삽입, 삭제 연산의 수행시간은 각각 AVL의 높이에 비례하므로 각 연산의 수행시간은  $O(\log N)$

# 수행시간

- 다양한 실험결과에 따르면, AVL 트리는 거의 정렬된 데이터를 삽입한 후에 랜덤 순서로 데이터를 탐색하는 경우 가장 좋은 성능을 보임
- 이진탐색트리는 랜덤 순서의 데이터를 삽입한 후에 랜덤 순서로 데이터를 탐색하는 경우 가장 좋은 성능을 보임<sup>a</sup>