

# 11주차 탐색트리

# 이진탐색

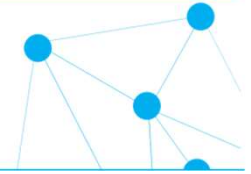


## 이진탐색(Binary Search)

정렬된 데이터의 중간에 위치한 항목을 기준으로 데이터를 두 부분으로 나누어 가며 특정 항목을 찾는 탐색 방법

```
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
```

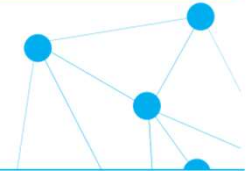
# 간단한 탐색 알고리즘



- 순차 탐색(sequential search)
  - 정렬되지 않은 배열을 처음부터 마지막까지 하나씩 검사
  - 가장 간단하고 직접적인 탐색 방법
  - 평균 비교 횟수:  $(n + 1)/2$ 번 비교 (최악의 경우:  $n$ 번)



# 순차 탐색 알고리즘

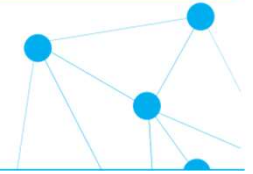


```
def sequential_search(A, key, low, high) :  
    for i in range(low, high+1) :  
        if A[i] == key :  
            return i  
    return None
```

# 순차탐색  
# i : low, low+1, ... high  
# 탐색 성공하면  
# 인덱스 반환  
# 탐색에 실패하면 None 반환

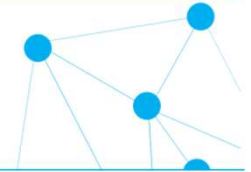
- 시간 복잡도:  $O(n)$

# 이진 탐색(binary search)

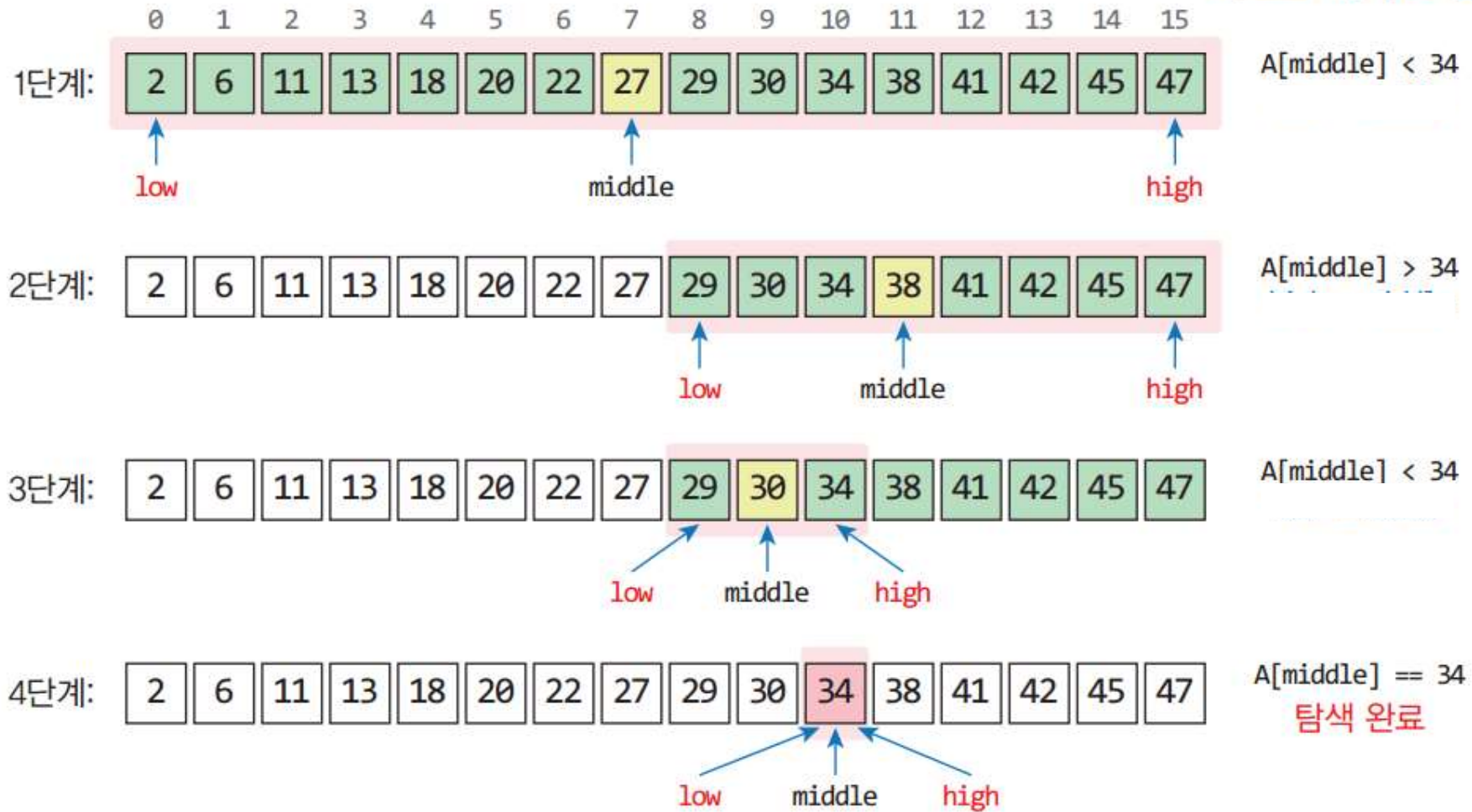


- 정렬된 배열의 탐색에 적합
  - 배열의 중앙에 있는 값을 조사하여 찾고자 하는 항목이 왼쪽 또는 오른쪽 부분 배열에 있는지를 알아내어 탐색의 범위를 반으로 줄여가며 탐색 진행
  - 예) 사전에서 단어 찾기
- (예) 10억 명중에서 특정한 이름 탐색
  - 이진탐색 : 단지 30번의 비교 필요
  - 순차 탐색 : 평균 5억 번의 비교 필요

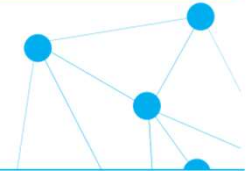
# 이진 탐색



리스트 A에서 34 탐색



# 이진 탐색 알고리즘

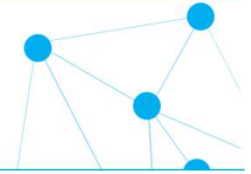


```
def binary_search(A, key, low, high) :  
    if (low <= high) :  
        middle = (low + high) // 2  
        if key == A[middle] :  
            return middle  
        elif (key < A[middle]) :  
            return binary_search(A, key, low, middle - 1)  
        else :  
            return binary_search(A, key, middle + 1, high)  
    return None
```

# 항목들이 남아 있으면(종료 조건)  
# 정수 나눗셈 //에 주의할 것.  
# 탐색 성공  
# 왼쪽 부분리스트 탐색  
# 오른쪽 부분리스트 탐색  
# 탐색 실패

- 시간 복잡도:  $O(\log n)$
- 반복으로 구현 가능

# 보간 탐색(interpolation search)



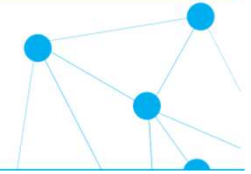
- 탐색키가 존재 할 위치를 예측하여 탐색
  - 예) 사전이나 전화번호부를 탐색할 때
    - 'ㅎ'으로 시작하는 단어는 사전의 뒷부분에서 찾을
    - 'ㄱ'으로 시작하는 단어는 앞부분에서 찾을
- 리스트를 불균등하게 분할하여 탐색
  - 탐색 값과 위치는 비례한다는 가정

$$\text{탐색위치} = \text{low} + (\text{high} - \text{low}) \cdot \frac{k - A[\text{low}]}{A[\text{high}] - A[\text{low}]}$$

```
middle = int(low + (high-low) * ((key - A[low]) / (A[high] - a[low])))
```



# 보간탐색



- 탐색 위치 계산 예

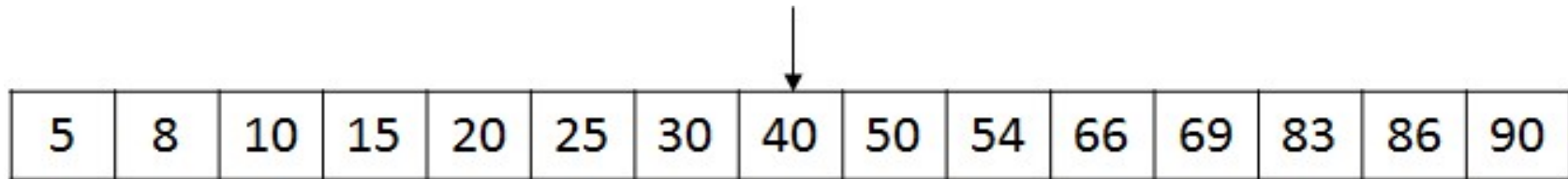
$$\begin{aligned}\text{탐색 위치} &= \frac{(k - \text{list}[\text{low}])}{\text{list}[\text{high}] - \text{list}[\text{low}]} * (\text{high} - \text{low}) + \text{low} \\ &= \frac{(55 - 3)}{(91 - 3)} * (9 - 0) + 0 \\ &= 5.31 \\ &\approx 5\end{aligned}$$

$$\text{탐색 위치} = (55 - 3) / (91 - 3) * (9 - 0) + 0 = 5.31 \div 5$$



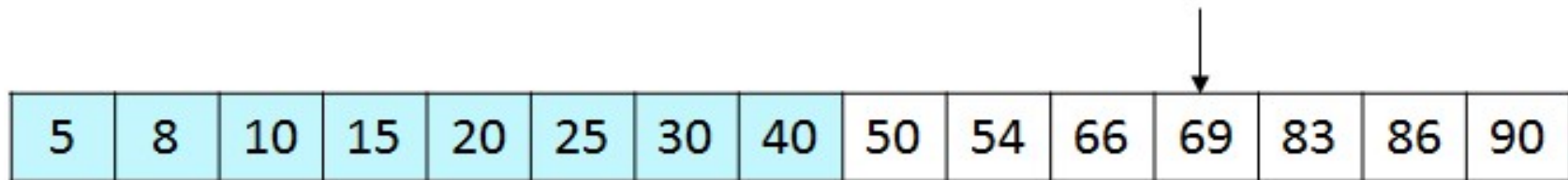
0	1	2	3	4	5	6	7	8	9
3	9	15	22	31	55	67	88	89	91

## 이진탐색으로 66을 찾는 과정



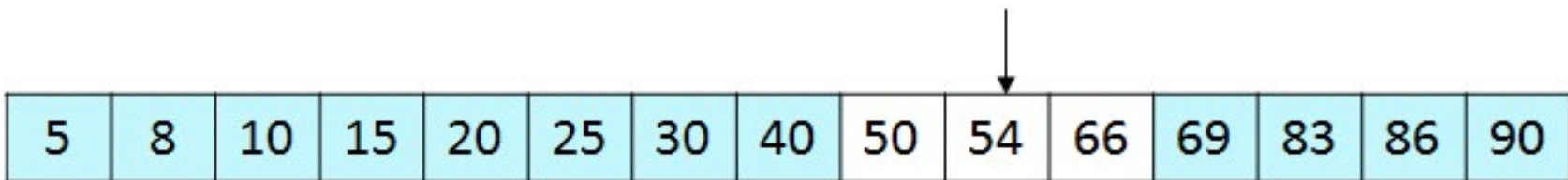
Initial array: [5, 8, 10, 15, 20, 25, 30, 40, 50, 54, 66, 69, 83, 86, 90]. The search range is the entire array. The middle element is 40.

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



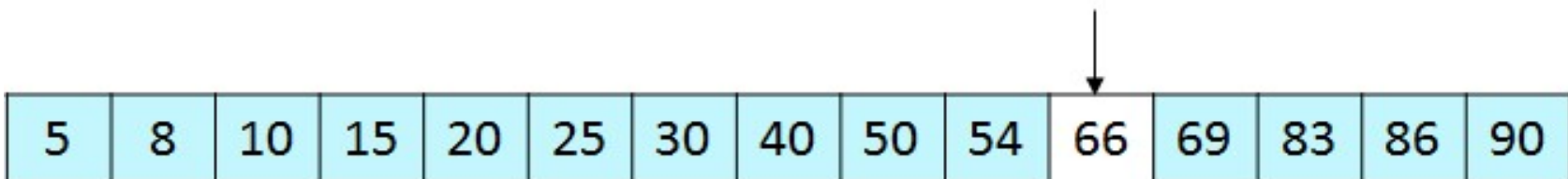
Search range: [5, 8, 10, 15, 20, 25, 30, 40]. The middle element is 30.

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



Search range: [40, 50, 54, 66, 69, 83, 86, 90]. The middle element is 69.

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



Search range: [40, 50, 54, 66]. The middle element is 54.

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

$$T(N) = T(N/2) + 1$$

$$T(1) = 1$$

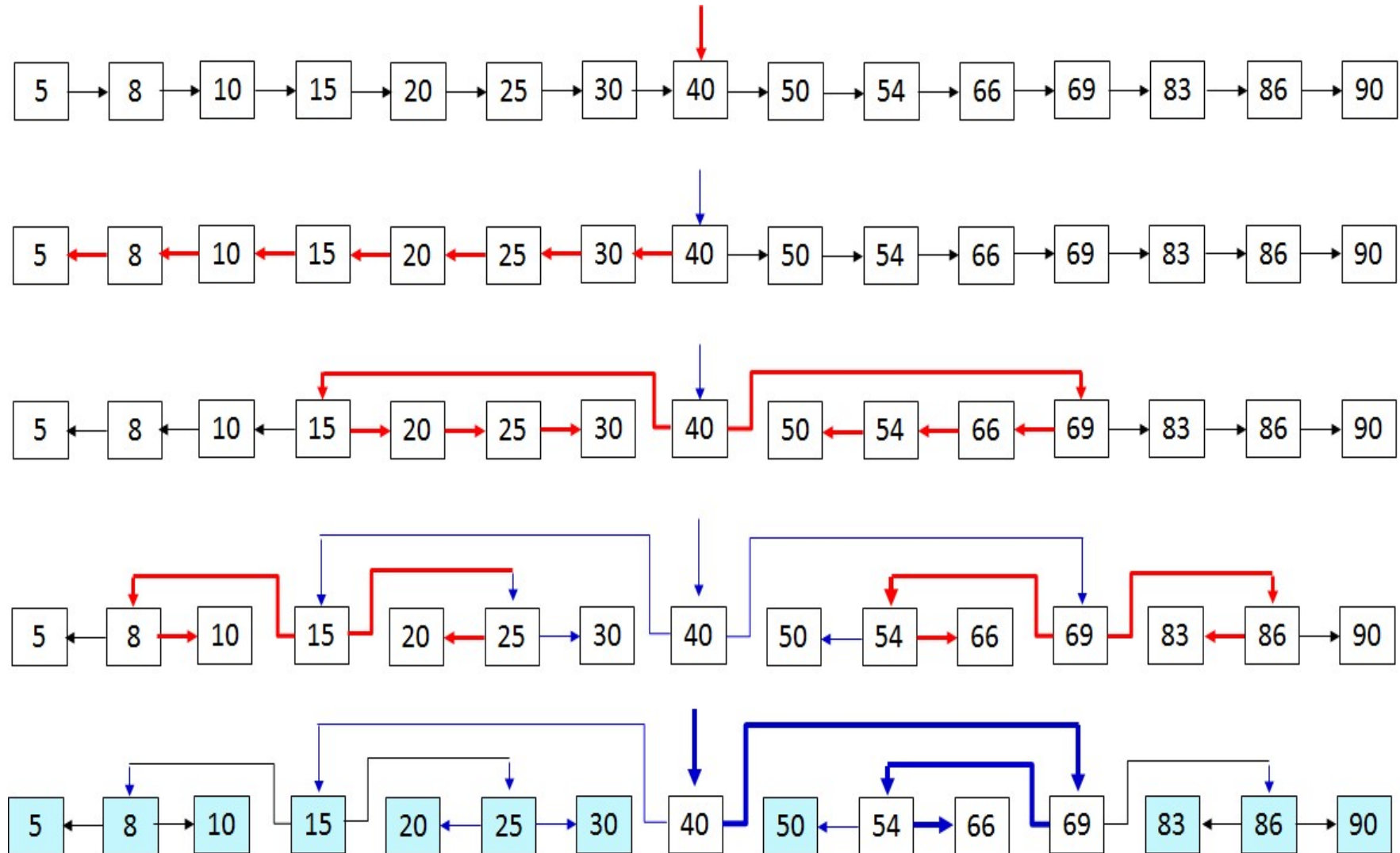
- $T(N) = T(N/2) + 1$ 
  - $= [T((N/2)/2) + 1] + 1 = T(N/2^2) + 2$
  - $= [T((N/2)/2^2) + 1] + 2 = T(N/2^3) + 3$
  - $= \dots = T(N/2^k) + k$
  - $= T(1) + k, \text{ if } N = 2^k, k = \log_2 N$
  - $= 1 + \log_2 N = O(\log N)$

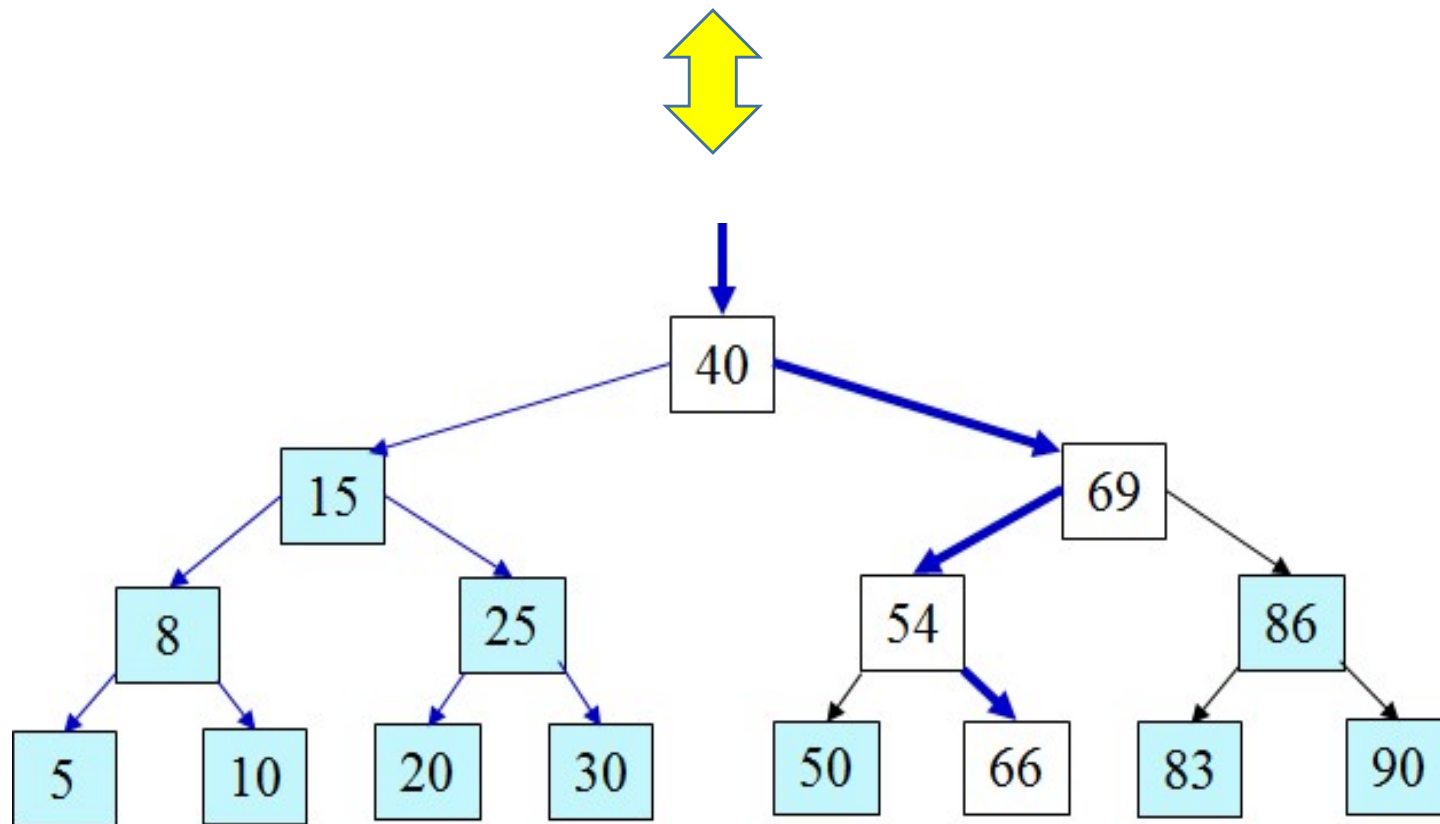
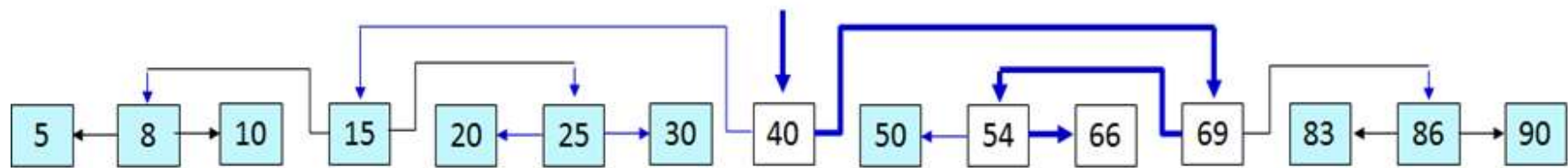
# 이진탐색트리

- 이진탐색트리(Binary Search Tree):

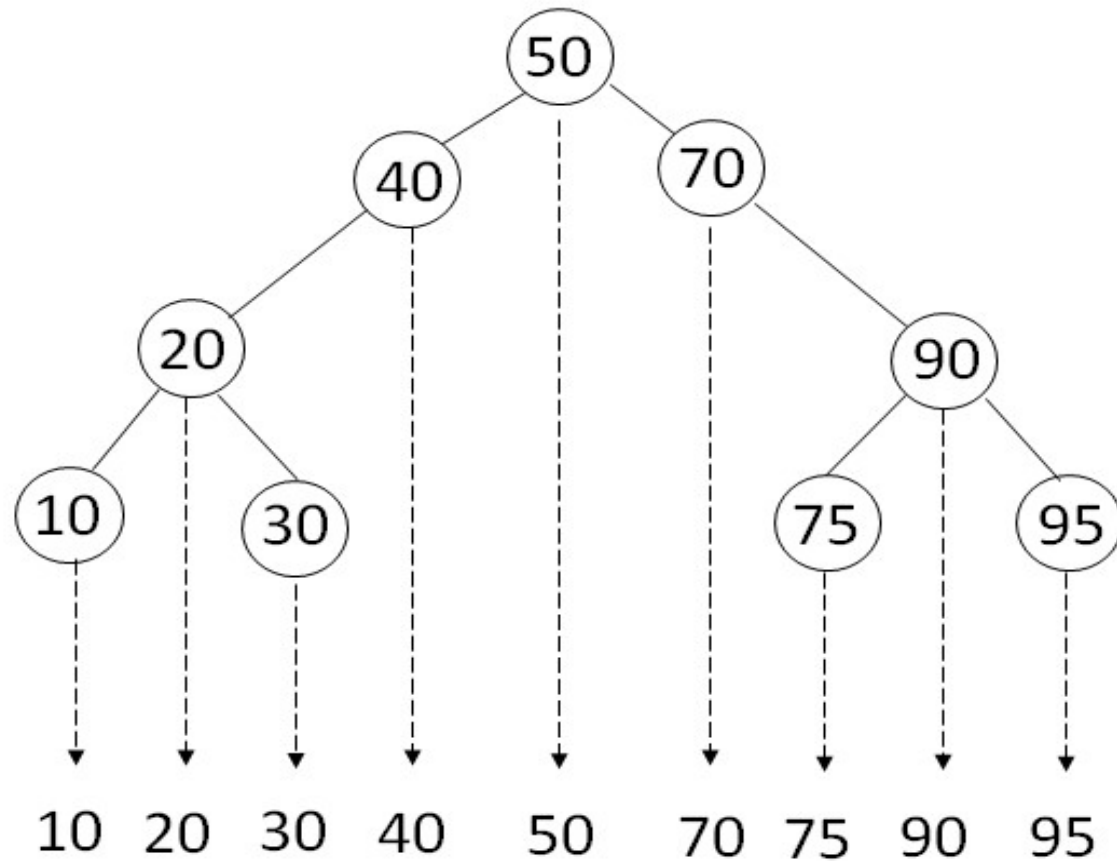
이진탐색(Binary Search)의 개념을 트리 형태의 구조에 접목한 자료구조

- 트리 형태의 자료구조에서 이진탐색을 수행하기 위해 1차원 리스트를 단순연결리스트로 만든 후, 점차 이진트리 형태로 변환하는 과정



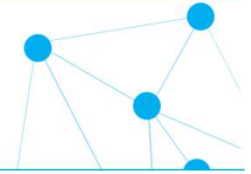


- 이진탐색트리의 특징 중의 하나는 트리를 중위순회(Inorder Traversal)하면 정렬되어 출력



정렬됨

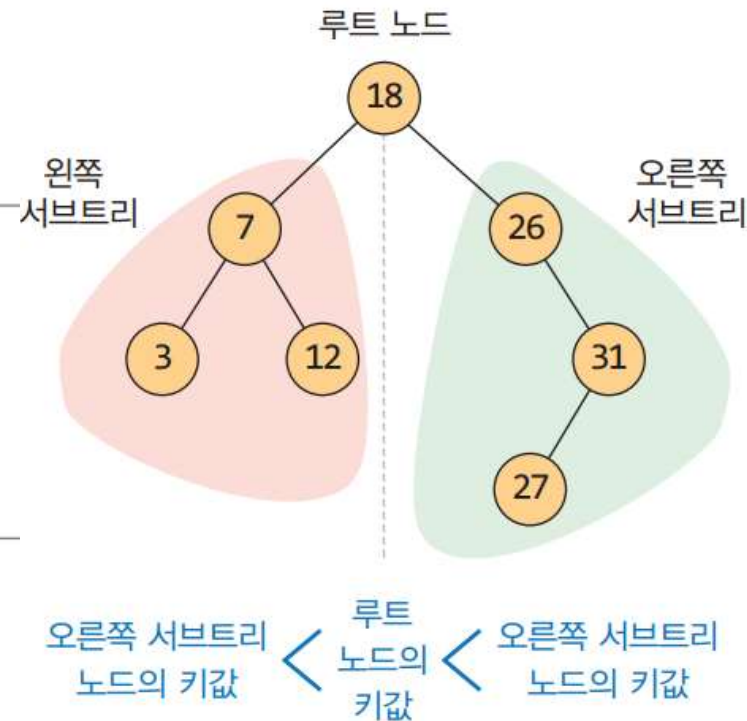
# 탐색트리



- 탐색을 위한 트리 기반의 자료구조이다.
- 이진탐색트리
  - 효율적인 탐색을 위한 이진트리 기반의 자료구조
  - 삽입, 삭제, 탐색:  $O(\log n)$

## 이진탐색트리

- 모든 노드는 유일한 키를 갖는다.
- 왼쪽 서브트리의 키들은 루트의 키보다 작다.
- 오른쪽 서브트리의 키들은 루트의 키보다 크다.
- 왼쪽과 오른쪽 서브트리도 이진탐색트리이다.



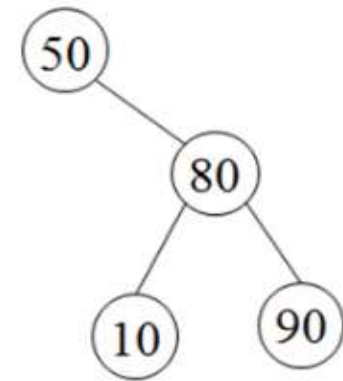
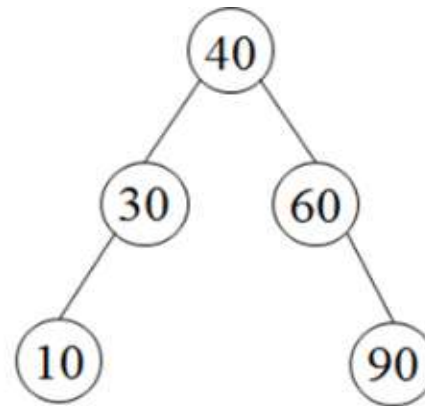
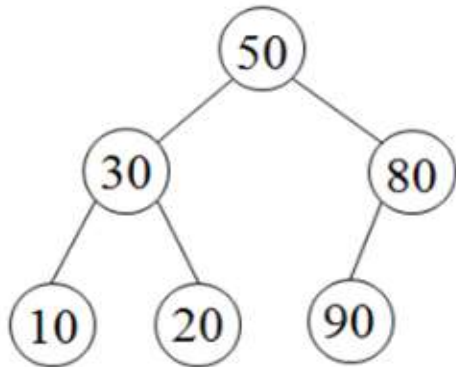




이진탐색트리는 이진트리로서 각 노드가 다음과 같은 조건을 만족한다.

- 각 노드  $n$ 의 키가  $n$ 의 왼쪽 서브트리에 있는 키들보다 (같거나) 크고,  $n$ 의 오른쪽 서브트리에 있는 키들보다 작다.

[이진탐색트리 조건]



어느 트리가 이진탐색트리인가?

## 이진탐색트리를 위한 BST 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

[프로그램 5-1] bst.py

# 탐색 연산

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

```
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
```

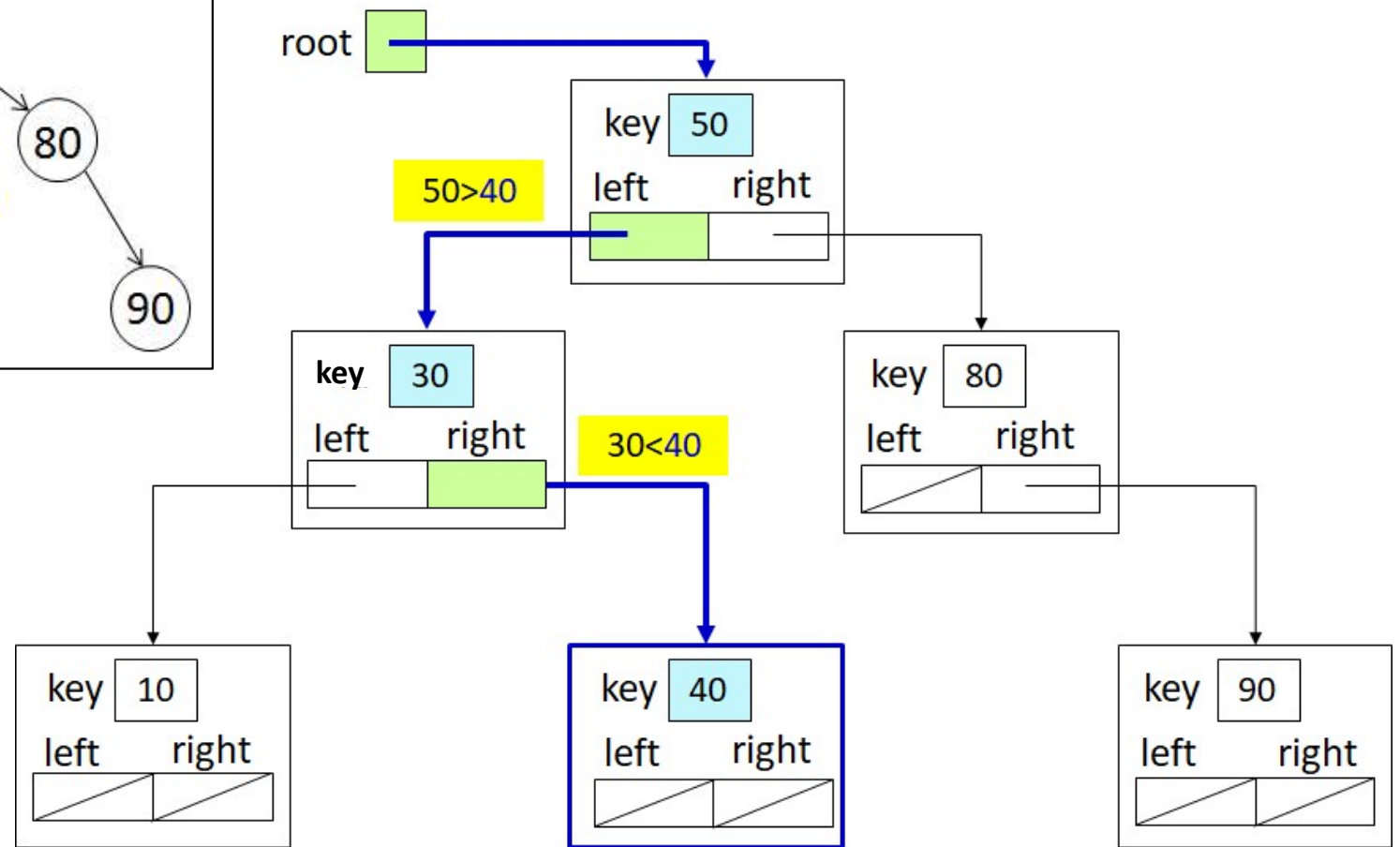
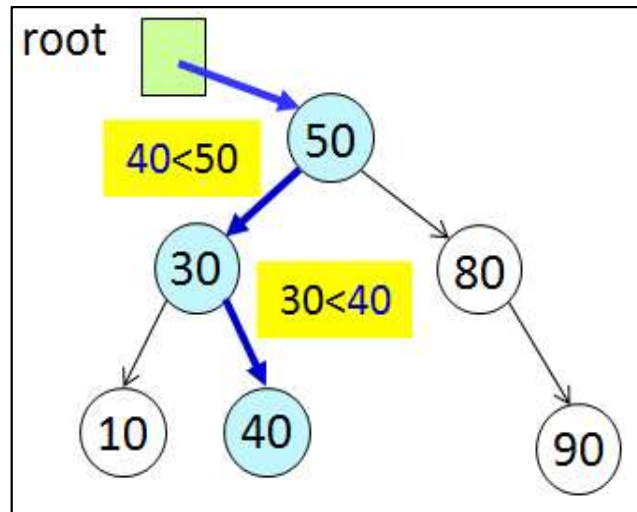
탐색 실패

k가 노드의 key보다 작으면  
왼쪽 서브트리 탐색

k가 노드의 key보다 크면  
오른쪽 서브트리 탐색

탐색 성공

## [예제] 40을 탐색하는 과정



# 삽입 연산

- 삽입은 탐색 연산과 거의 동일
- 탐색 중 None을 만나면 새 노드를 생성하여 부모노드와 연결
- 단, 이미 트리에 존재하는 키를 삽입한 경우, value만 갱신

```

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

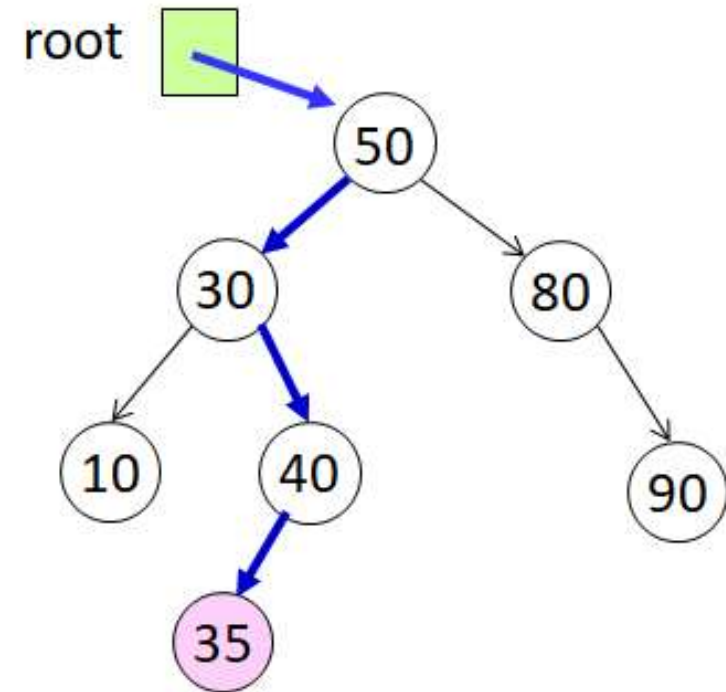
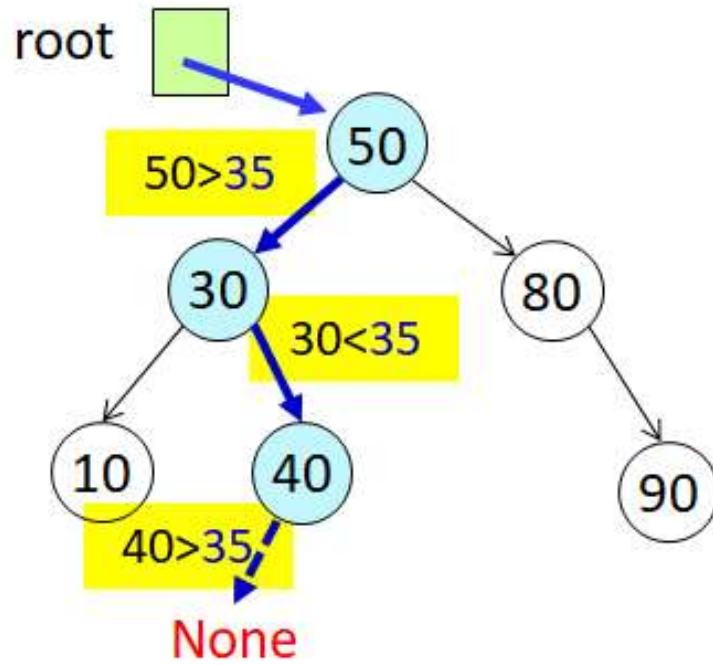
n의 왼쪽자식과 put\_item()이 리턴하는 노드를 재 연결

n의 오른쪽자식과 put\_item()이 리턴하는 노드를 재 연결

key가 이미 있으므로 value만 갱신

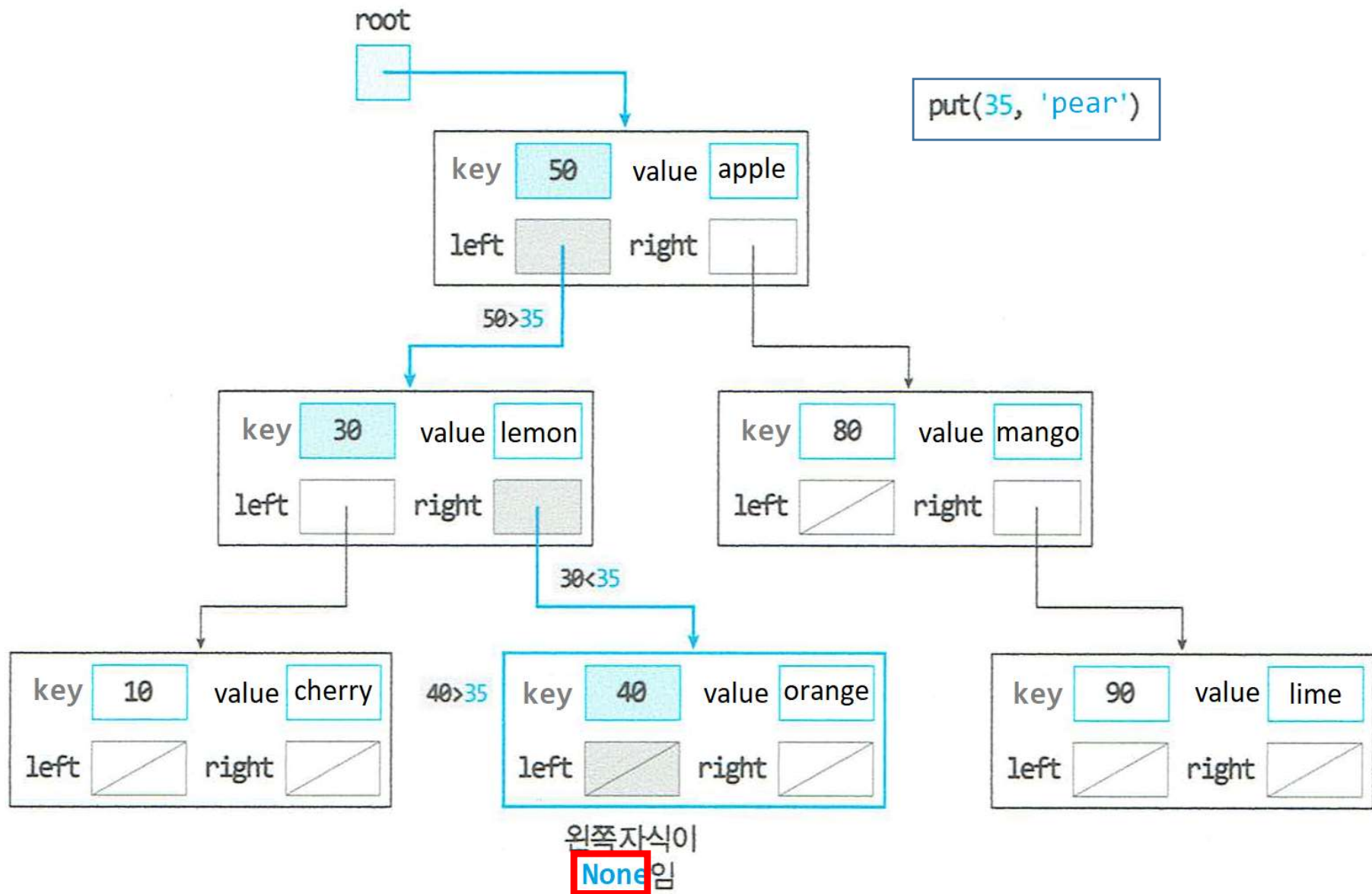
부모노드와 연결하기 위해 노드 n을 리턴

## [예제] 35를 삽입하는 과정

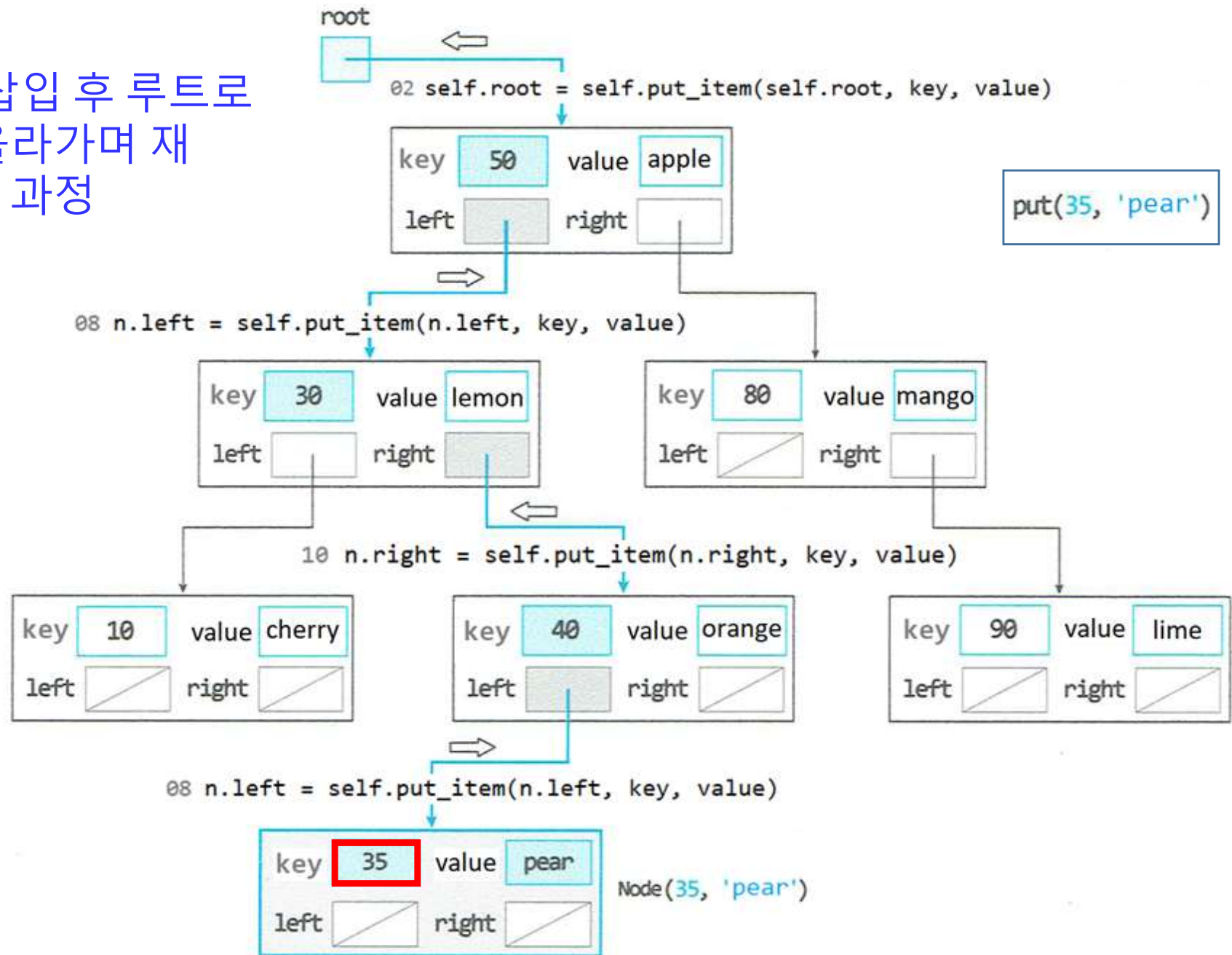




## <35를 삽입할 장소를 탐색하는 과정>



새 노드 삽입 후 루트로  
거슬러 올라가며 재  
연결하는 과정



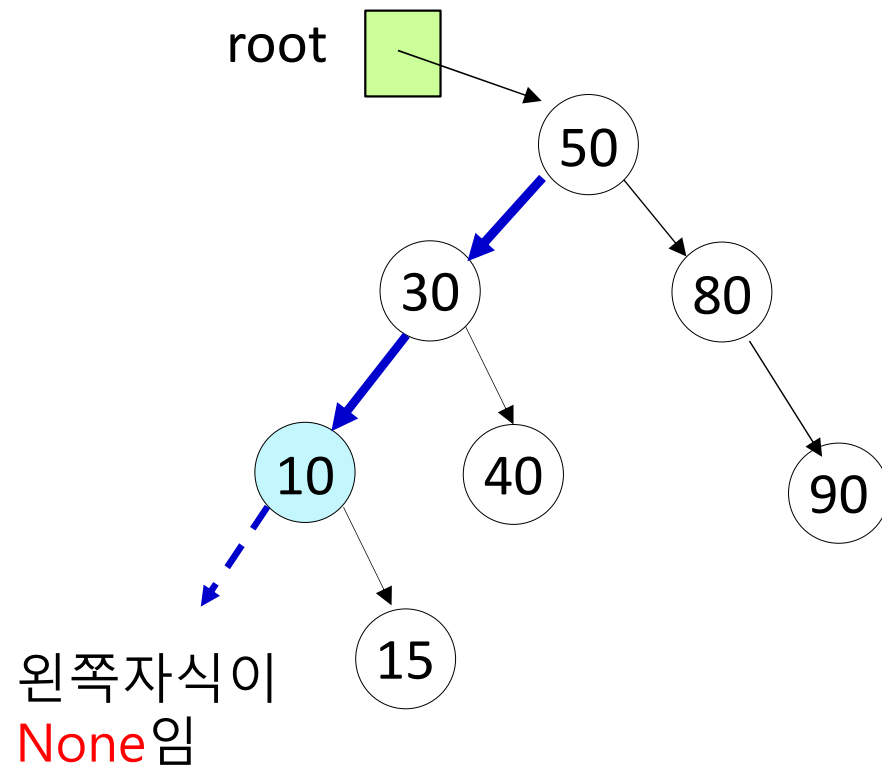
# 최솟값 찾기

- 최솟값은 루트노드로부터 왼쪽 자식을 따라 내려가며, None을 만났을 때 None의 부모가 가진 value
- minimum() 메소드는 delete()에서 사용

```
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



[그림 5-9] min()의 수행 과정

# 최솟값 삭제 연산

- 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 `None`이더라도 자식으로 연결
- `delete_min()`은 임의의 `value`를 가진 노드를 삭제하는 `delete()`에서 사용

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`

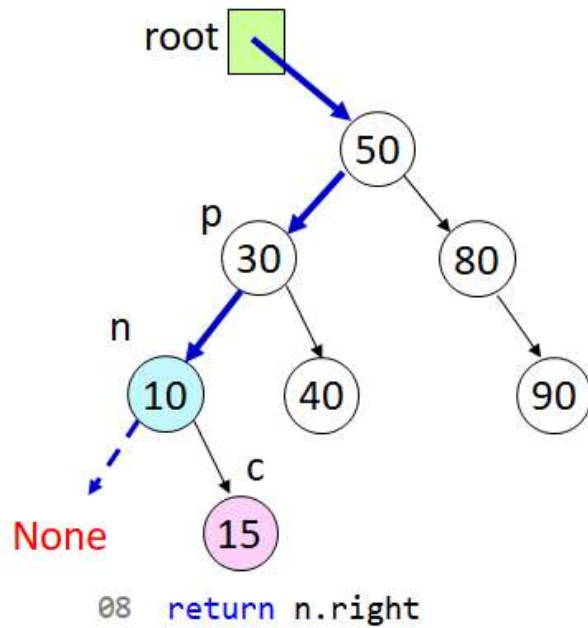
09 `n.left = self.del_min(n.left)`

10 `return n`

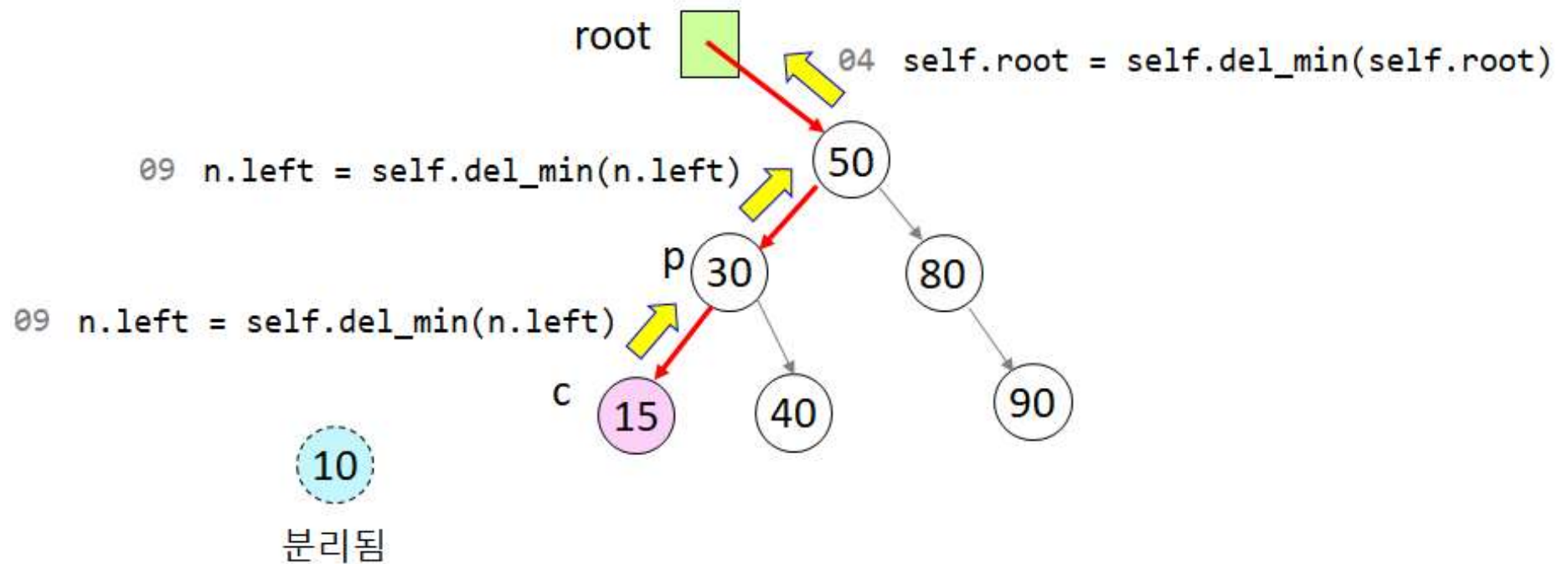
루트와 `del_min()`이 리턴  
하는 노드를 재 연결

최솟값 가진 노드의 오른쪽  
자식을 리턴

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

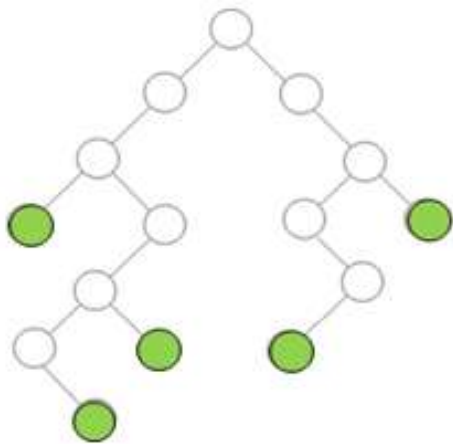


[그림 5-10] delete\_min()의 수행 과정

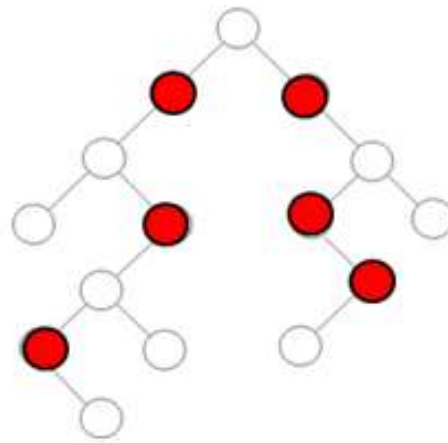


# 삭제 연산

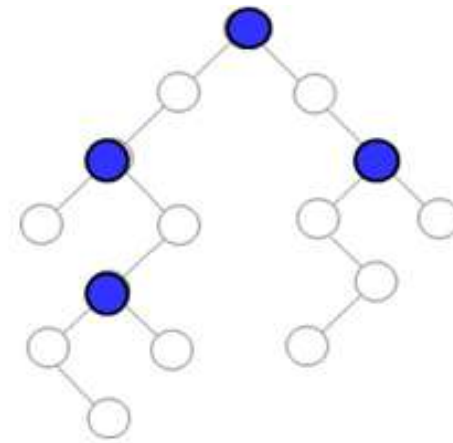
- 우선 삭제하고자 하는 노드를 찾은 후 이진탐색트리 조건을 만족하도록 삭제된 노드의 부모와 자식(들)을 연결해 주어야
- 삭제되는 노드가 자식이 없는 경우(case 0), 자식이 하나인 경우(case 1), 자식이 둘인 경우(case 2)로 나누어 delete 연산을 수행



case 0



case 1

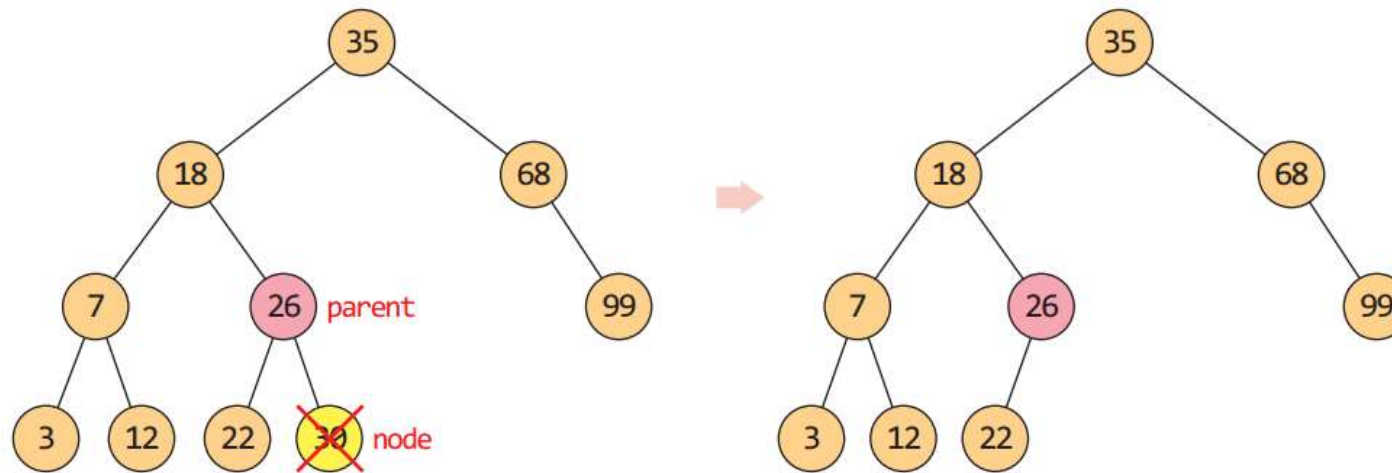
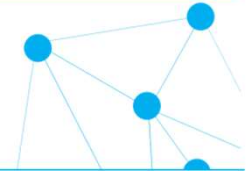


case 2

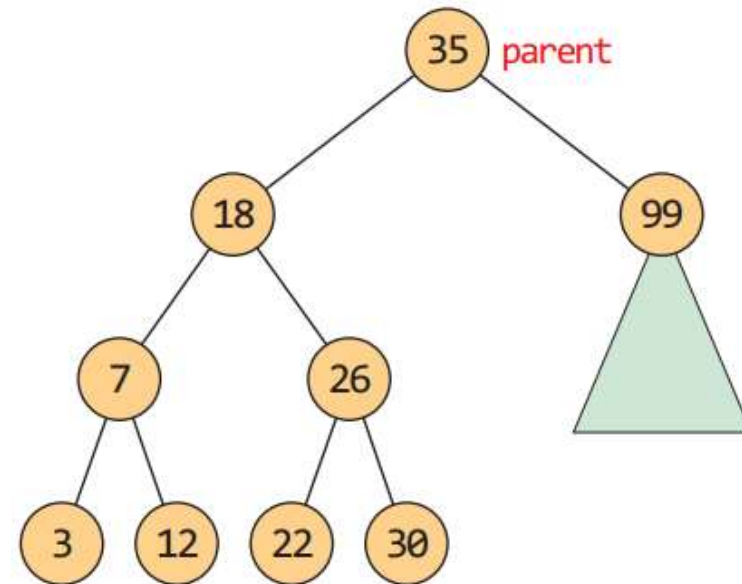
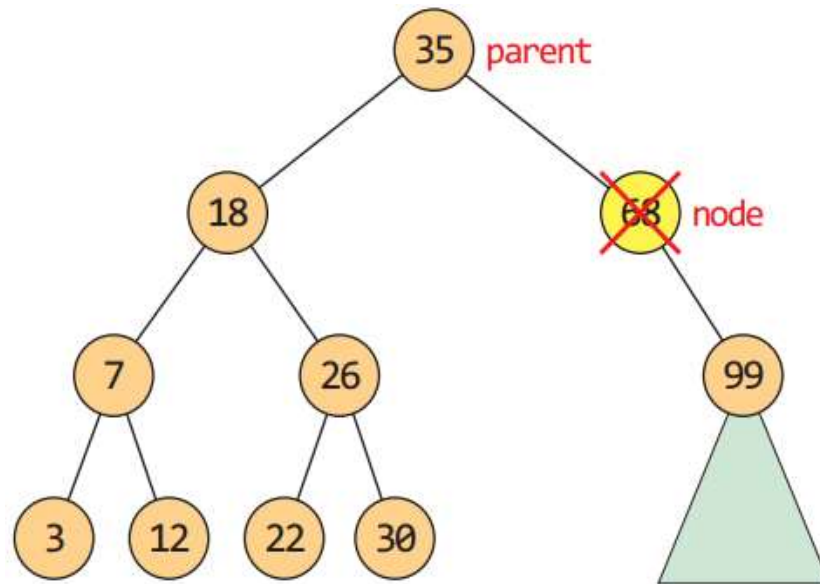
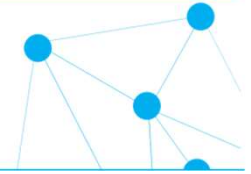


- **Case 0:** 삭제해야 할 노드  $n$ 의 부모가  $n$ 을 가리키던 레퍼런스를 None으로 만든다.
- **Case 1:**  $n$ 가 한쪽 자식인  $c$ 만 가지고 있다면,  $n$ 의 부모와  $n$ 의 자식  $c$ 를 직접 연결
- **Case 2:**  $n$ 의 부모는 하나인데  $n$ 의 자식이 둘이므로  $n$ 의 자리에 중위순회하면서  $n$ 을 방문하기 직전 노드(Inorder Predecessor, **중위 선행자**) 또는 직후에 방문되는 노드(Inorder Successor, **중위 후속자**)로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

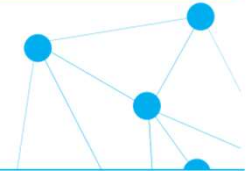
## Case 0: 단말 노드 삭제



# Case1: 자식이 하나인 노드의 삭제

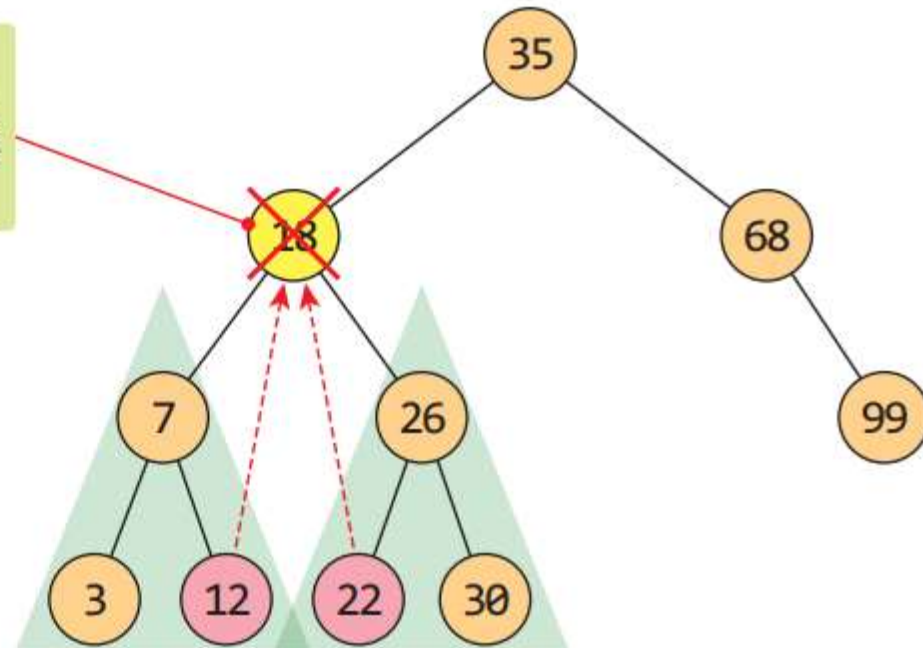


## Case 2: 두 개의 자식을 가진 노드 삭제

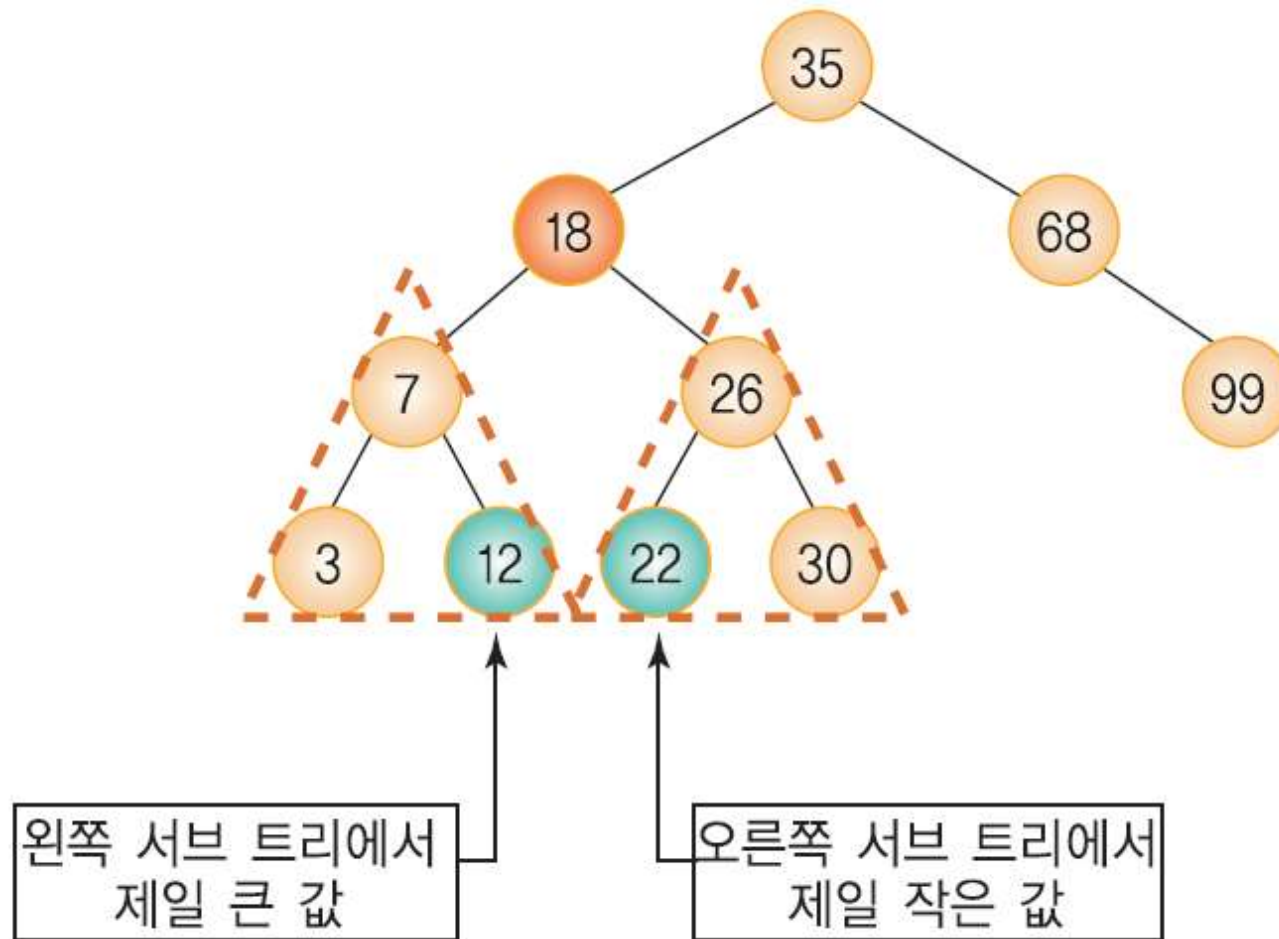


- 가장 비슷한 값을 가진 노드를 삭제 위치로 가져옴
- 후계 노드의 선택

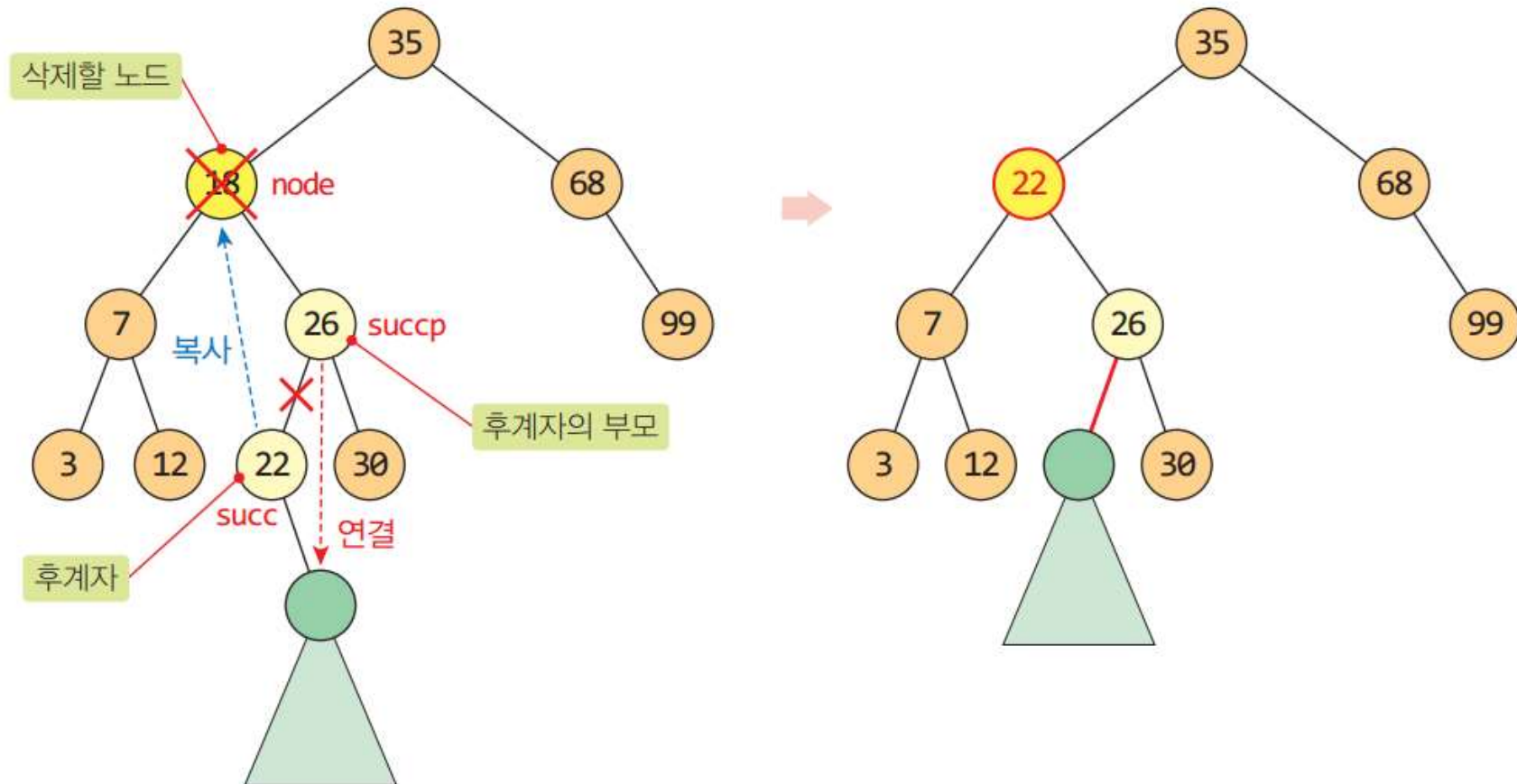
삭제할 위치에 왼쪽 서브트리의 가장 큰 노드나 오른쪽 서브트리의 가장 작은 노드가 들어가면 이진탐색트리의 조건을 계속 만족한다.



가장 비슷한 값은 어디에 있을까?



## [예제] 노드 18 삭제



```

01 def delete(self, k): # 삭제 연산
02     self.root = self.del_node(self.root, k)
03
04 def del_node(self, n, k):
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)
11     else:
12         if n.right == None:
13             return n.left
14         if n.left == None:
15             return n.right
16         target = n
17         n = self.minimum(target.right)
18         n.right = self.del_min(target.right)
19         n.left = target.left
20     return n

```

루트와 `del_node()`가 리턴하는 노드를 재 연결

`n`의 왼쪽자식과 `del_node()`가 리턴하는 노드를 재 연결

`n`의 오른쪽자식과 `del_node()`가 리턴하는 노드를 재 연결

`target`은 삭제될 노드

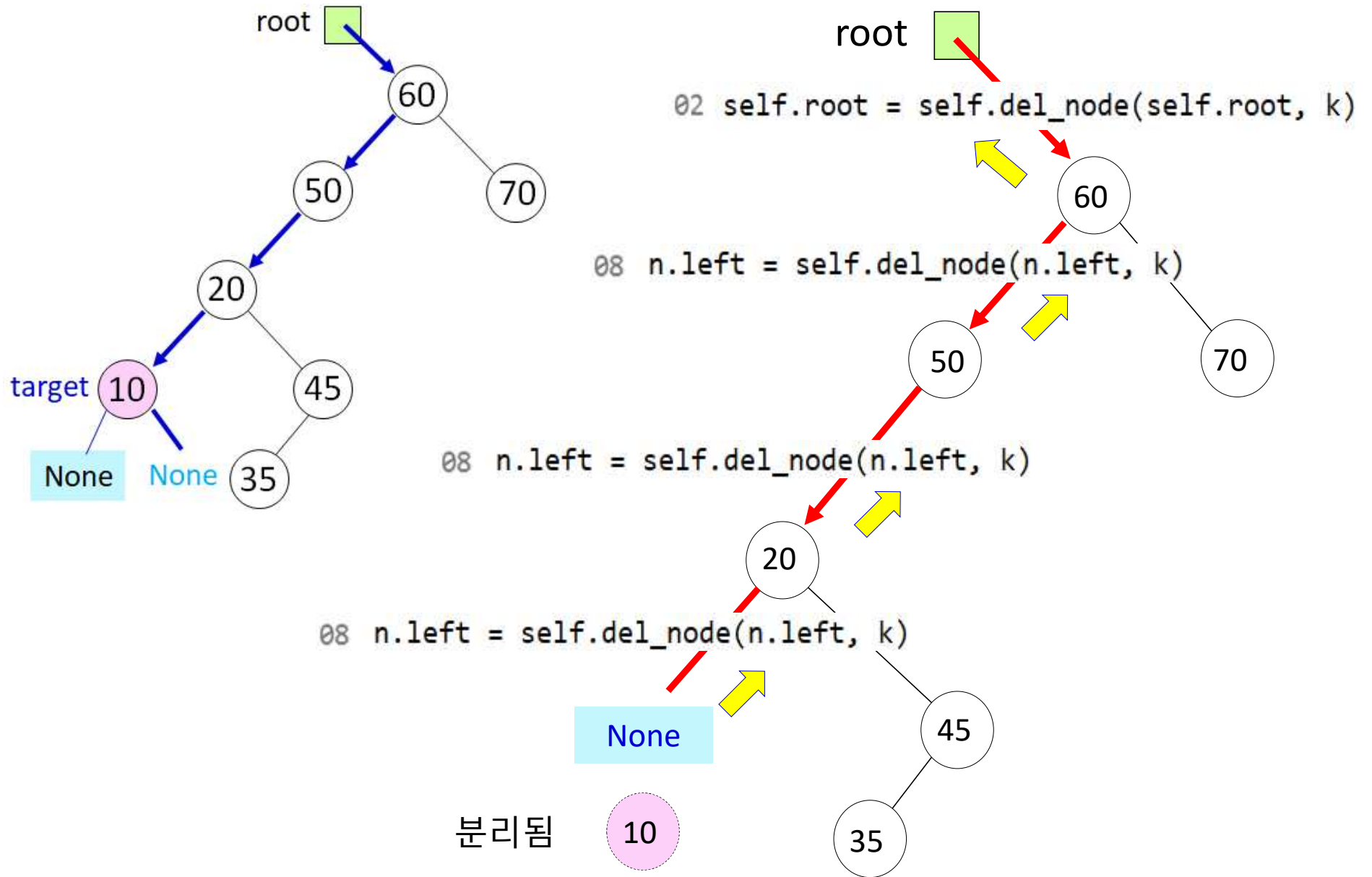
`target`의 중위 후속자 찾아 `n`이 참조하게 함

`n`의 오른쪽자식과 `target`의 오른쪽자식 연결

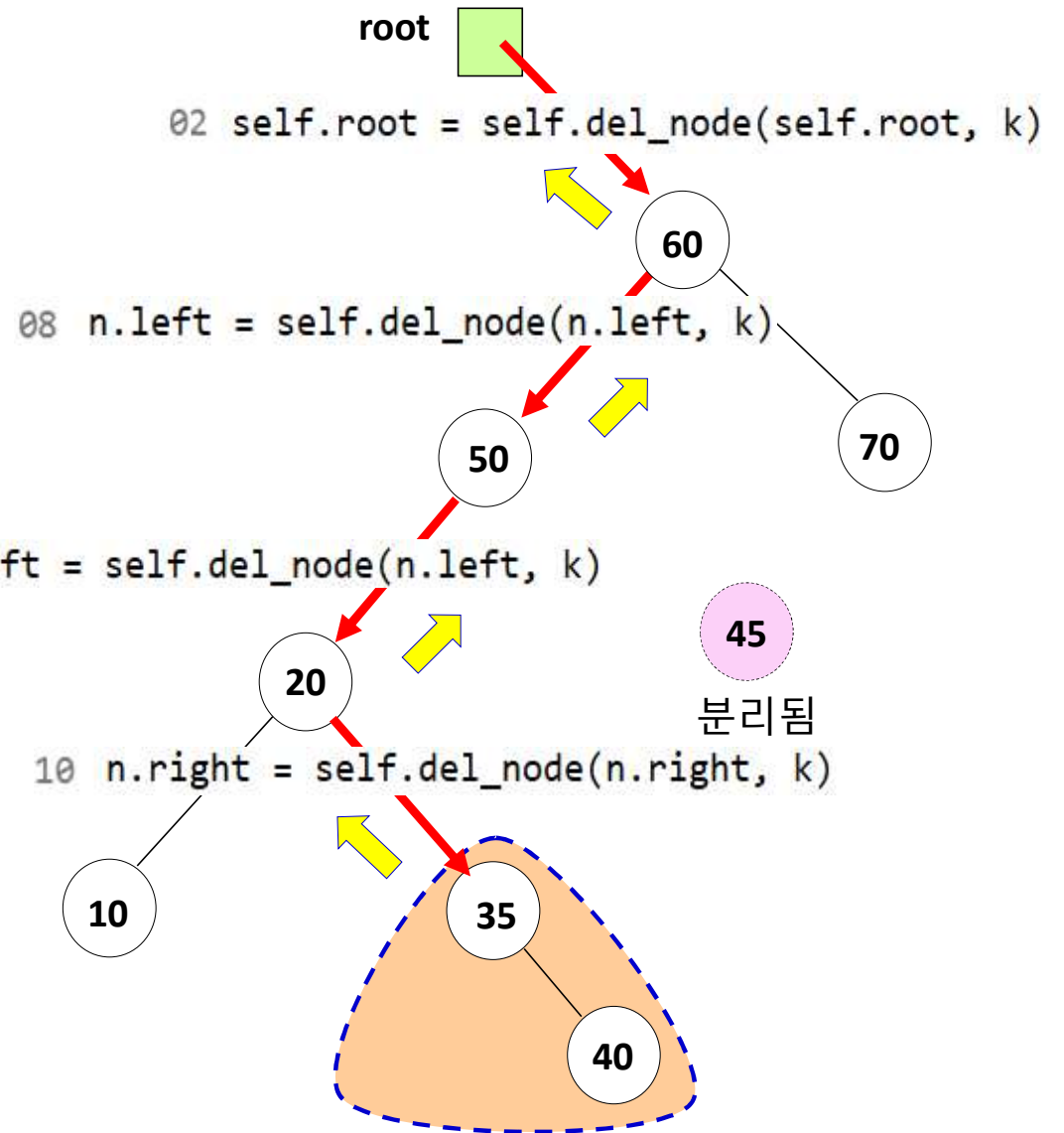
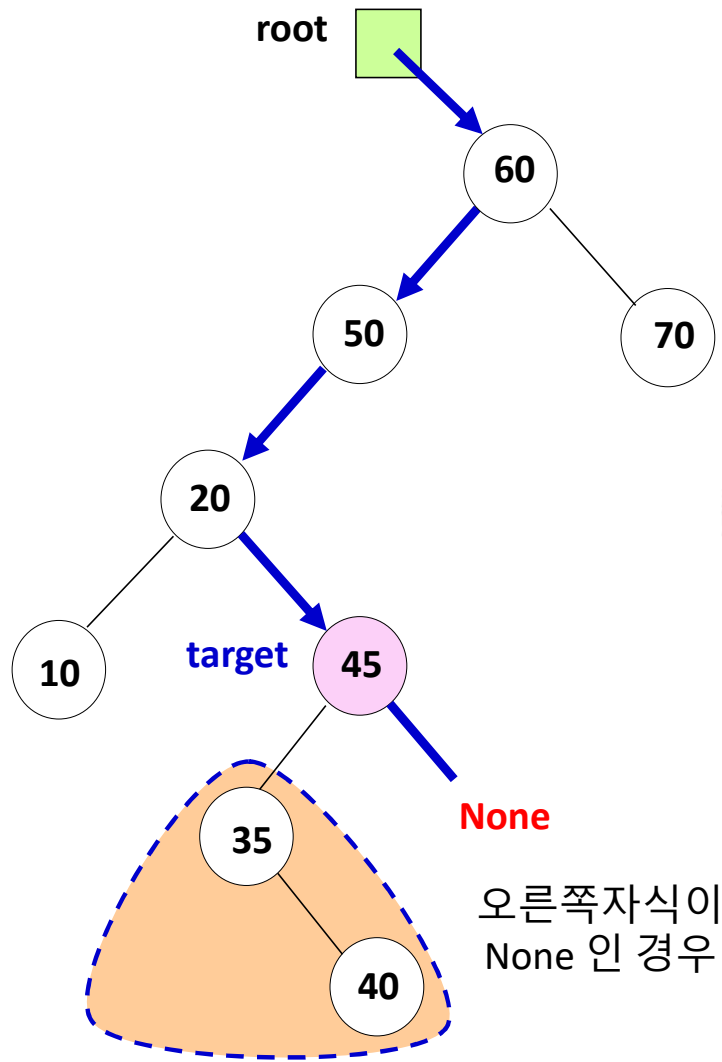
`n`의 왼쪽자식과 `target`의 왼쪽자식 연결



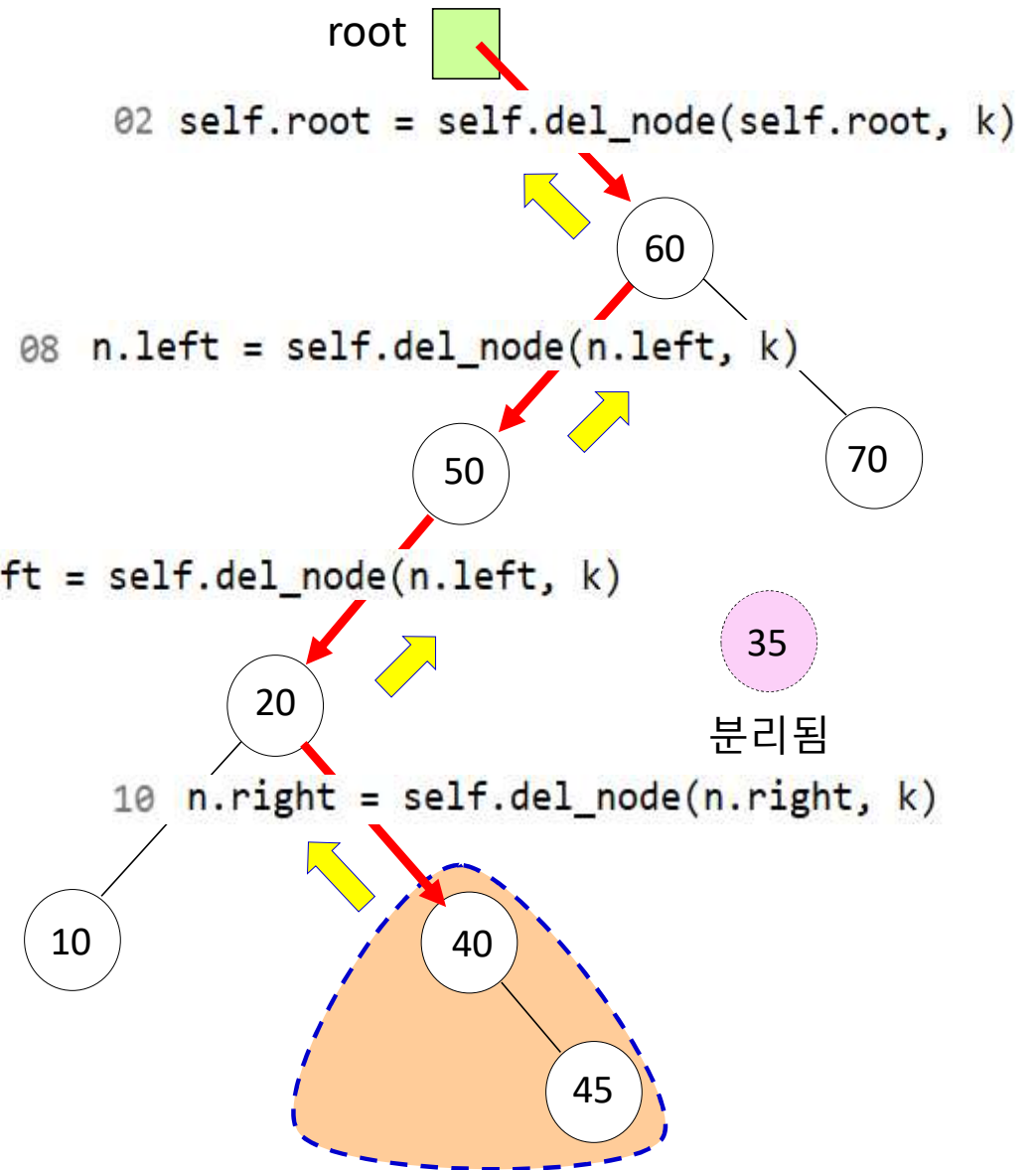
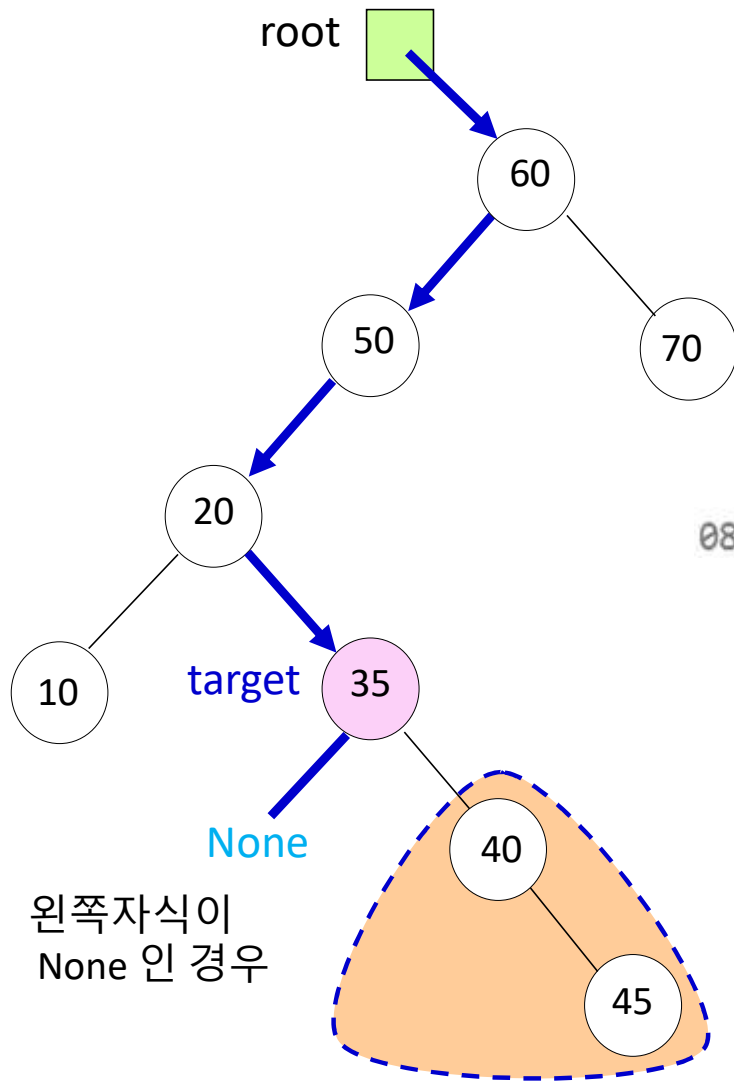
[그림 5-12] delete(10)이 수행되는 과정 (case 0)







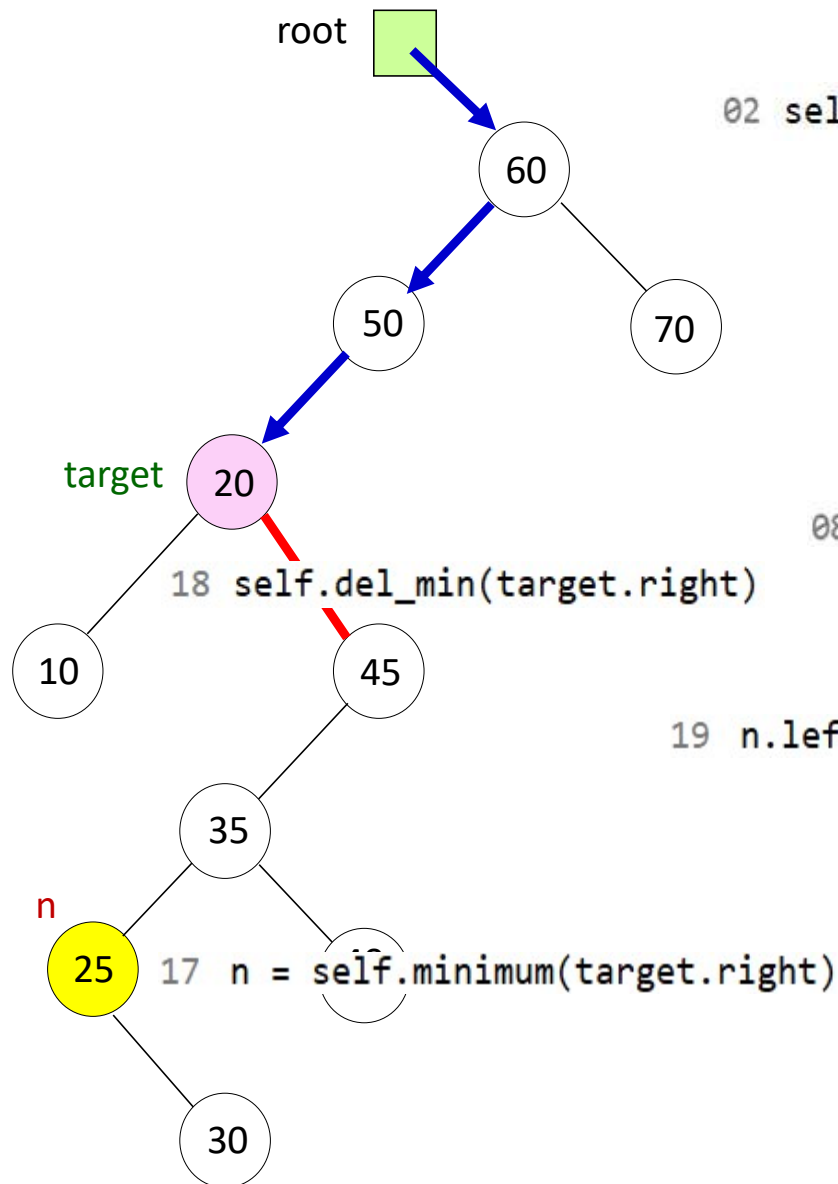
[그림 5-13] delete(45)가 수행되는 과정 (case 1)



delete(35)가 수행되는 과정 (case 1)

[그림 5-14] delete(20)이 수행되는 과정 (case 2)

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



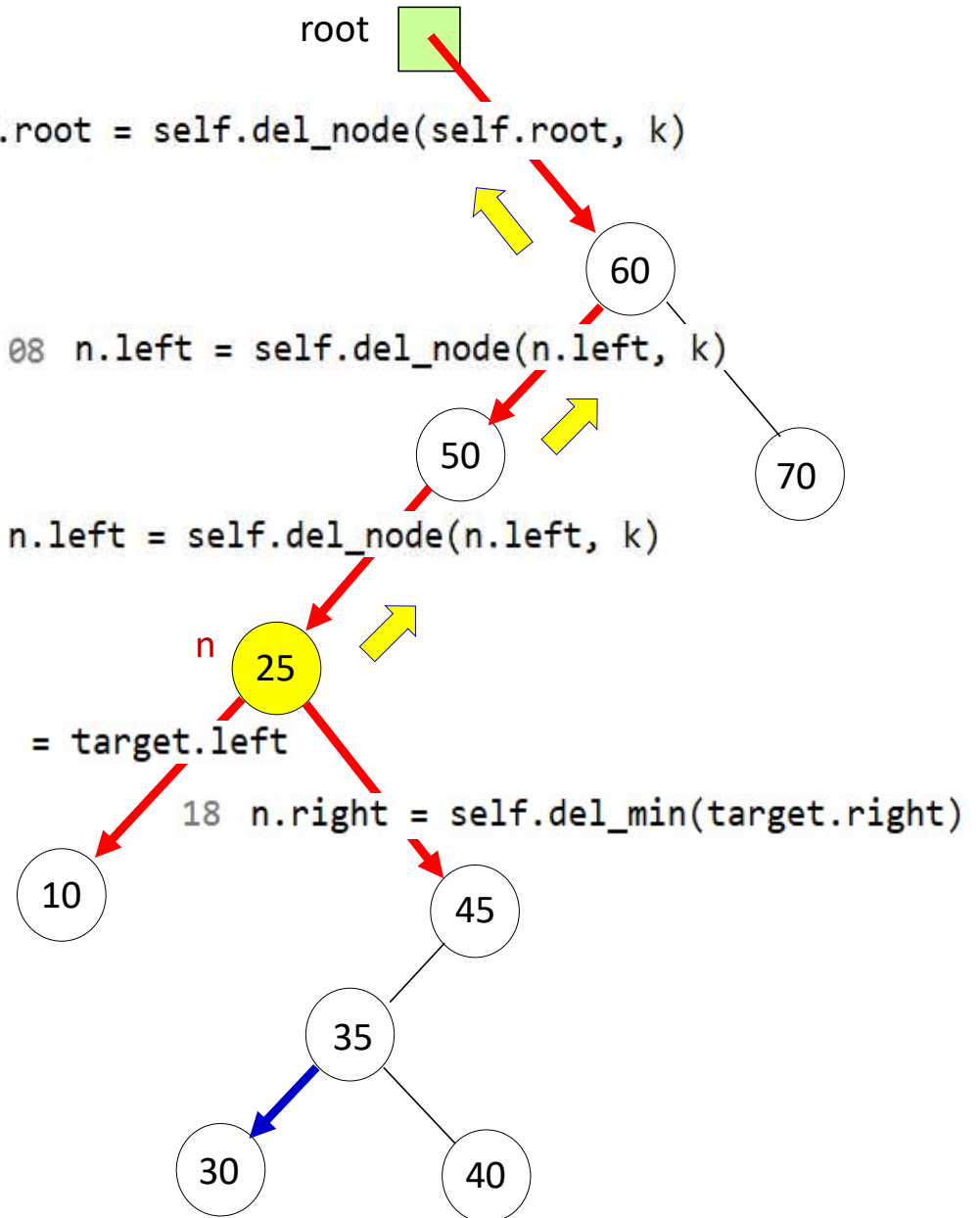
```
02 self.root = self.del_node(self.root, k)
```

```
08 n.left = self.del_node(n.left, k)
```

```
08 n.left = self.del_node(n.left, k)
```

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

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



## [프로그램 5-2] main.py

```

01 from bst import BST
02 if __name__ == '__main__':
03     t = BST()
04     t.put(500, 'apple')
05     t.put(600, 'banana')
06     t.put(200, 'melon')
07     t.put(100, 'orange')
08     t.put(400, 'lime')
09     t.put(250, 'kiwi')
10     t.put(150, 'grape')
11     t.put(800, 'peach')
12     t.put(700, 'cherry')
13     t.put(50, 'pear')
14     t.put(350, 'lemon')
15     t.put(10, 'plum')
16     print('전위순회:\t', end='')
17     t.preorder(t.root)
18     print('\n중위순회:\t', end='')
19     t.inorder(t.root)
20     print('\n250: ', t.get(250))
21     t.delete(200)
22     print('200 삭제 후:')
23     print('전위순회:\t', end='')
24     t.preorder(t.root)
25     print('\n중위순회:\t', end='')
26     t.inorder(t.root)

```

이진탐색트리  
객체 생성

1  
2  
개의  
항목  
삽입

트리  
순회  
및  
삭제  
연산  
수행

Console PyUnit

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

전위순회: 500 200 100 50 10 150 400 250 350 600 800 700

중위순회: 10 50 100 150 200 250 350 400 500 600 700 800

250: kiwi

200 삭제 후:

전위순회: 500 250 100 50 10 150 400 350 600 800 700

중위순회: 10 50 100 150 250 350 400 500 600 700 800

# 수행시간

- 이진 탐색트리에서 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,  $O(h)$

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

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

# AVL트리(균형이진탐색트리)

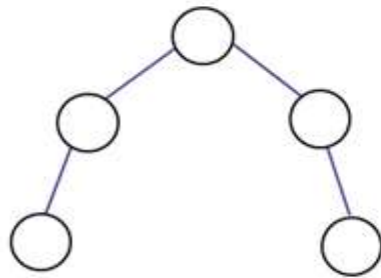
## [핵심 아이디어]

AVL트리는 삽입이나 삭제로 인해 균형이 깨지면 회전 연산을 통해 트리의 균형을 유지한다.

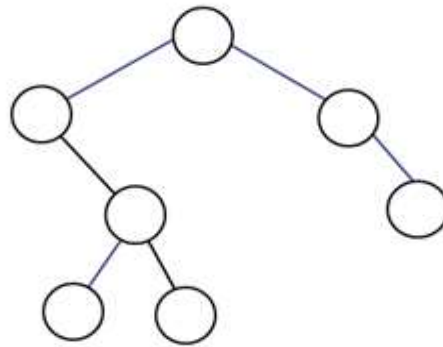
- AVL 트리는 트리가 한쪽으로 치우쳐 자라나는 현상을 방지하여 트리 높이의 균형 (Balance) 을 유지하는 이진탐색트리
- 균형(Balanced) 이진트리를 만들면 N개의 노드를 가진 트리의 높이가  $O(\log N)$ 이 되어 탐색, 삽입, 삭제 연산의 수행시간이  $O(\log N)$ 으로 보장



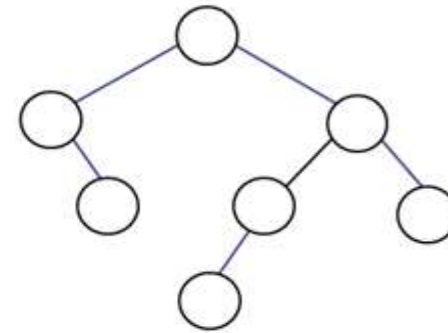
AVL트리는 임의의 노드  $x$ 에 대해  $x$ 의 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 차이가 1을 넘지 않는 이진탐색트리이다.



(a)



(b)



(c)

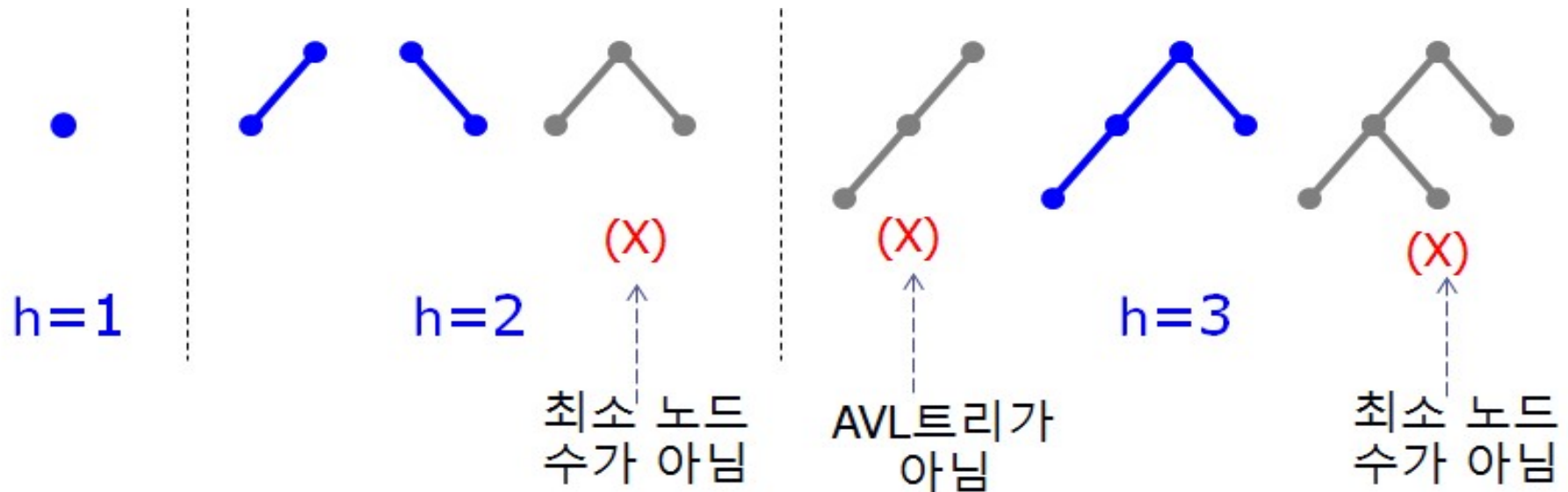
어느 트리가 AVL트리 형태를 갖추고 있나?



[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.



A(3)

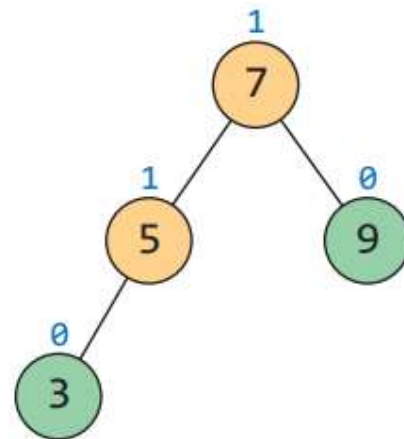


A(3)이 위와 같이 구성되는 이유:

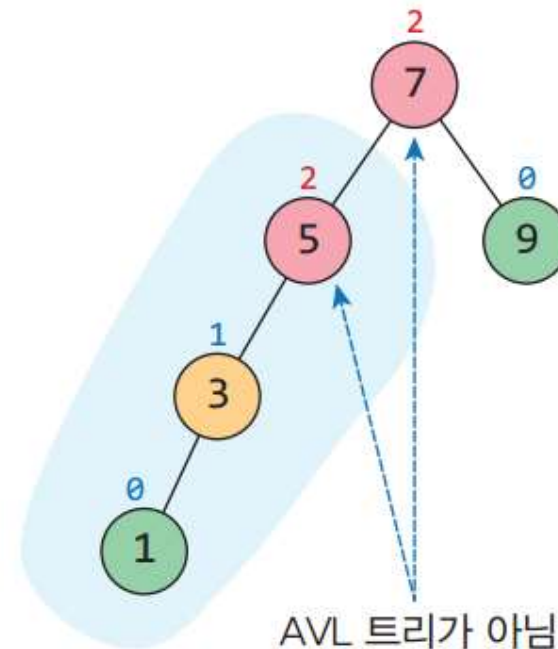
- 높이가 3인 AVL 트리에는 루트와 루트의 왼쪽 서브트리와 오른쪽 서브트리가 존재해야 하고,
- 각 서브트리 역시 최소 노드 수를 가진 AVL 트리여야 하므로
- 또한 이 두 개의 서브트리의 높이 차이가 1일 때 전체 트리의 노드 수가 최소가 되기 때문

- **균형 인수** : 왼쪽서브트리 높이 - 오른쪽서브트리 높이

- 균형 인수 +2, -2
- 균형 인수 +1, -1
- 균형 인수 0



AVL 트리



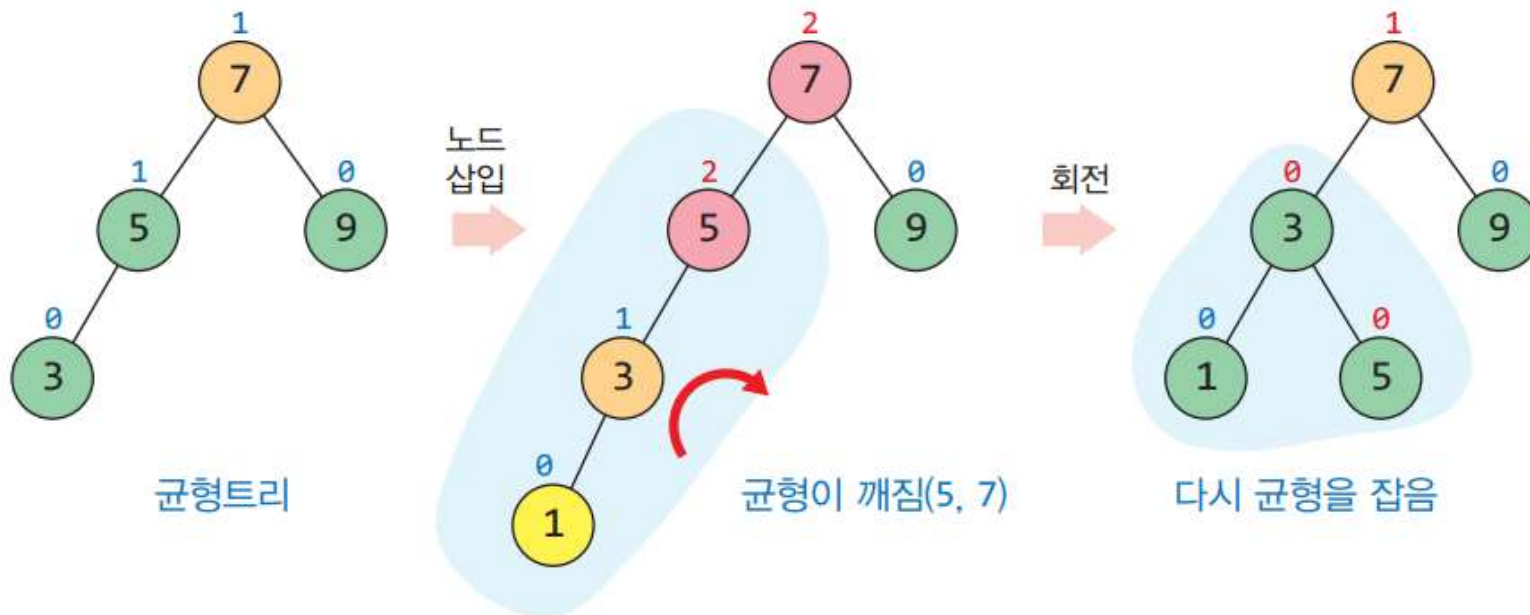
AVL 트리가 아님

# AVL 트리의 연산

- 탐색연산: 이진탐색트리와 동일
- 삽입과 삭제 시 균형 상태가 깨질 수 있음
- 삽입 연산
  - 삽입 위치에서 루트까지의 경로에 있는 조상 노드들의 균형 인수에 영향을 미침
  - 삽입 후에 불균형 상태로 변한 가장 가까운 조상 노드(균형 인수가  $\pm 2$ 가 된 가장 가까운 조상 노드)의 서브 트리들에 대하여 다시 재균형
  - 삽입 노드부터 균형 인수가  $\pm 2$ 가 된 가장 가까운 조상 노드까지 회전

# AVL 트리의 연산

- 노드 1을 트리에 추가 → 균형이 깨짐

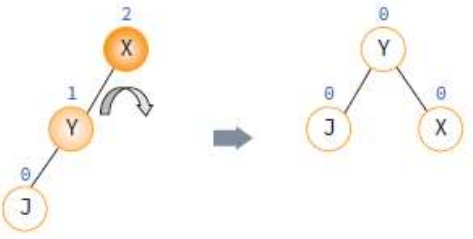
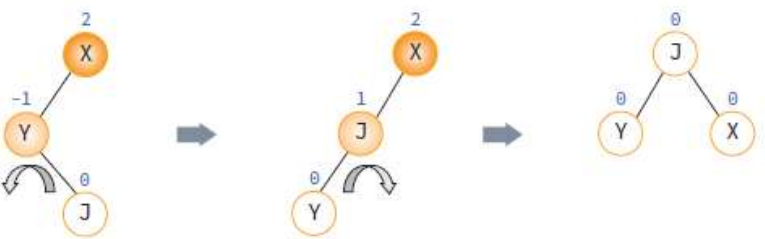
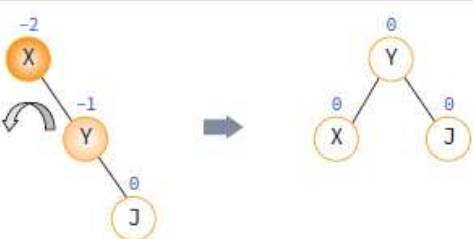
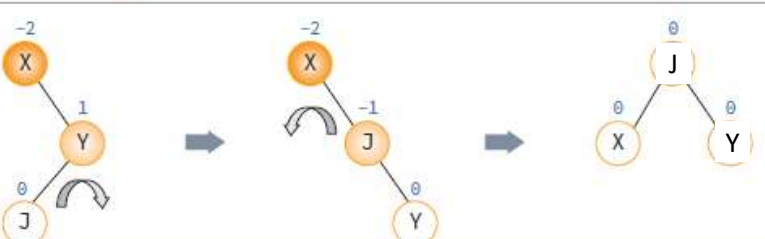


- 균형이 깨지는 4가지 경우
  - LL, LR, RL, RR 타입

# AVL 트리의 삽입연산

- AVL 트리의 균형이 깨지는 4가지 경우
  - 삽입된 노드 N으로부터 가장 가까우면서 균형 인수가  $\pm 2$ 가 된 조상 노드가 A라면
    - LL 타입: N이 A의 왼쪽서브트리의 왼쪽서브트리에 삽입
    - LR 타입: N이 A의 왼쪽서브트리의 오른쪽서브트리에 삽입
    - RR 타입: N이 A의 오른쪽서브트리의 오른쪽서브트리에 삽입
    - RL 타입: N이 A의 오른쪽서브트리의 왼쪽서브트리에 삽입
- 각 타입별 재균형 방법
  - LL 회전: A부터 N까지의 경로상 노드의 오른쪽 회전
  - LR 회전: A부터 N까지의 경로상 노드의 왼쪽-오른쪽 회전
  - RR 회전: A부터 N까지의 경로상 노드의 왼쪽 회전
  - RL 회전: A부터 N까지의 경로상 노드의 오른쪽-왼쪽 회전

# AVL 트리의 삽입연산

4가지의 경우	해결방법	설명
LL 타입		LL 회전: 오른쪽 회전
LR 타입		LR 회전: 왼쪽 회전 → 오른쪽 회전
RR 타입		RR 회전: 왼쪽 회전
RL 타입		RL 회전: 오른쪽 회전 → 왼쪽 회전

# AVL 트리의 회전 연산

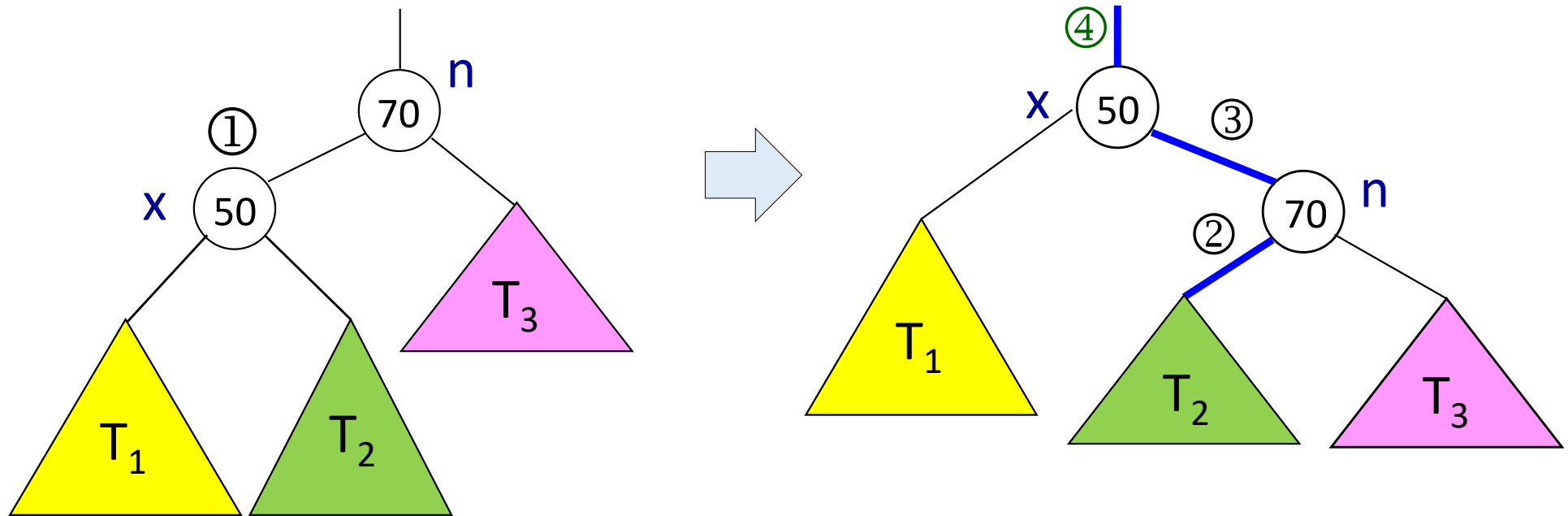
- AVL 트리에서 삽입 또는 삭제 연산을 수행할 때 트리의 균형을 유지하기 위해 LL-회전, RR-회전, LR-회전, RL-회전 연산 사용
- 회전 연산은 2 개의 기본적인 연산으로 구현
  - rotate\_right(), rotate\_left()



```

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

```

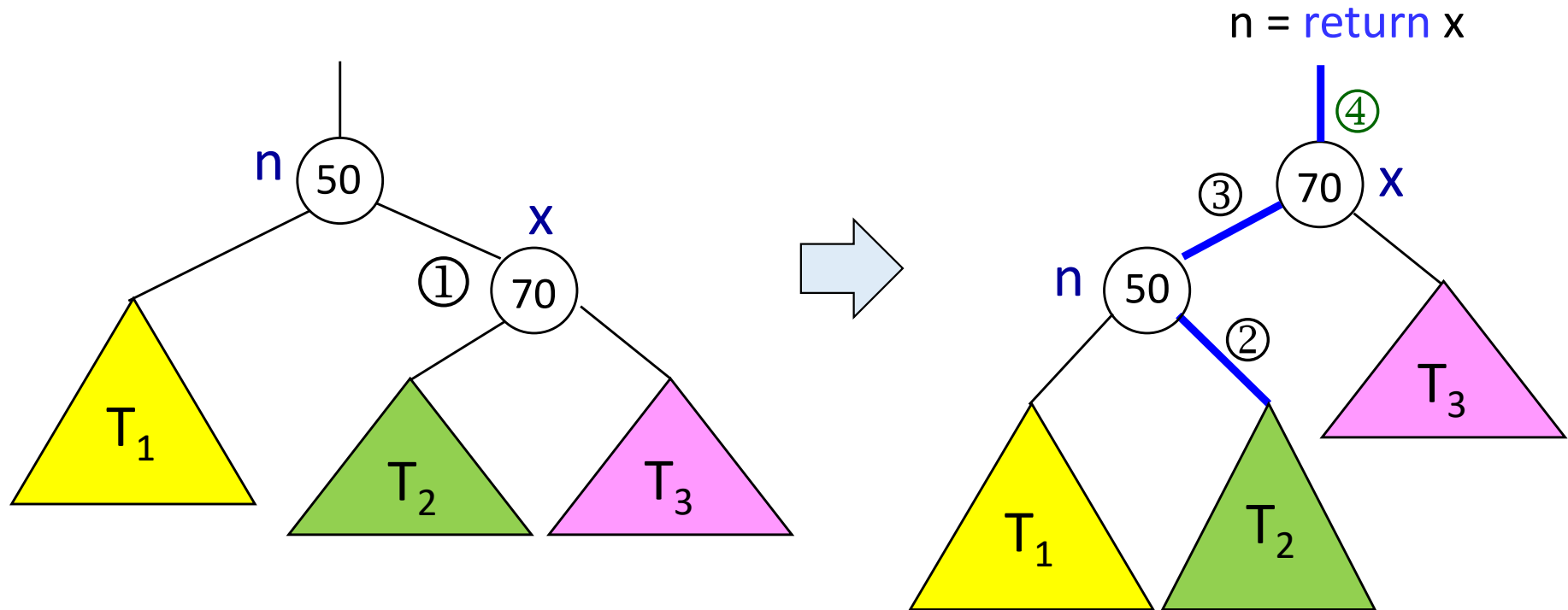


[그림 5-20] rotate\_right

```

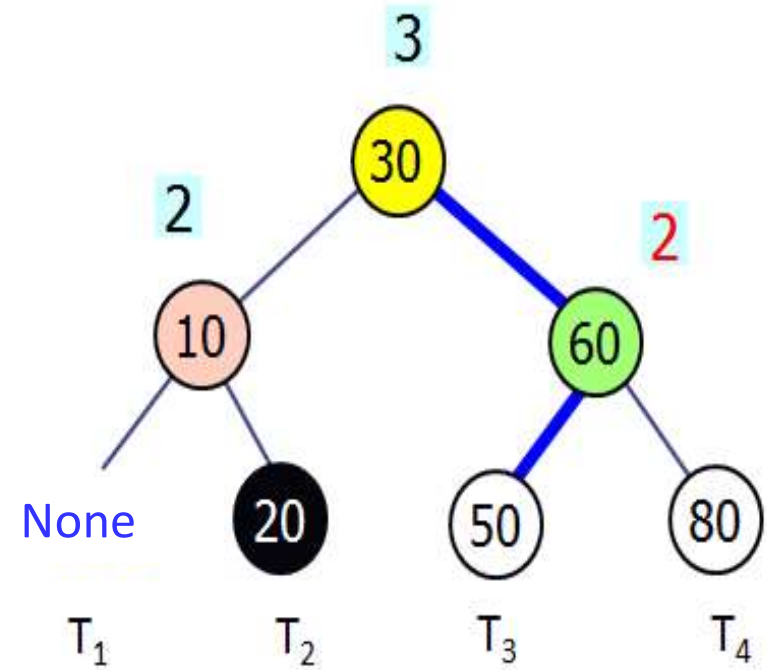
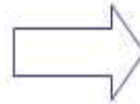
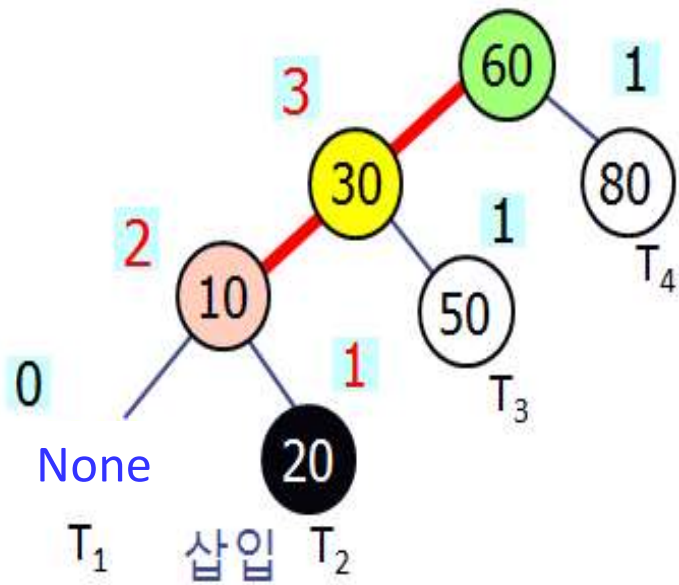
01 def rotate_left(self, n): # 좌로 회전
02     ① x = n.right
03     ② n.right = x.left
04     ③ x.left = n
05     n.height = max(self.height(n.left), self.height(n.right)) + 1
06     x.height = max(self.height(x.left), self.height(x.right)) + 1
07     ④ return x

```

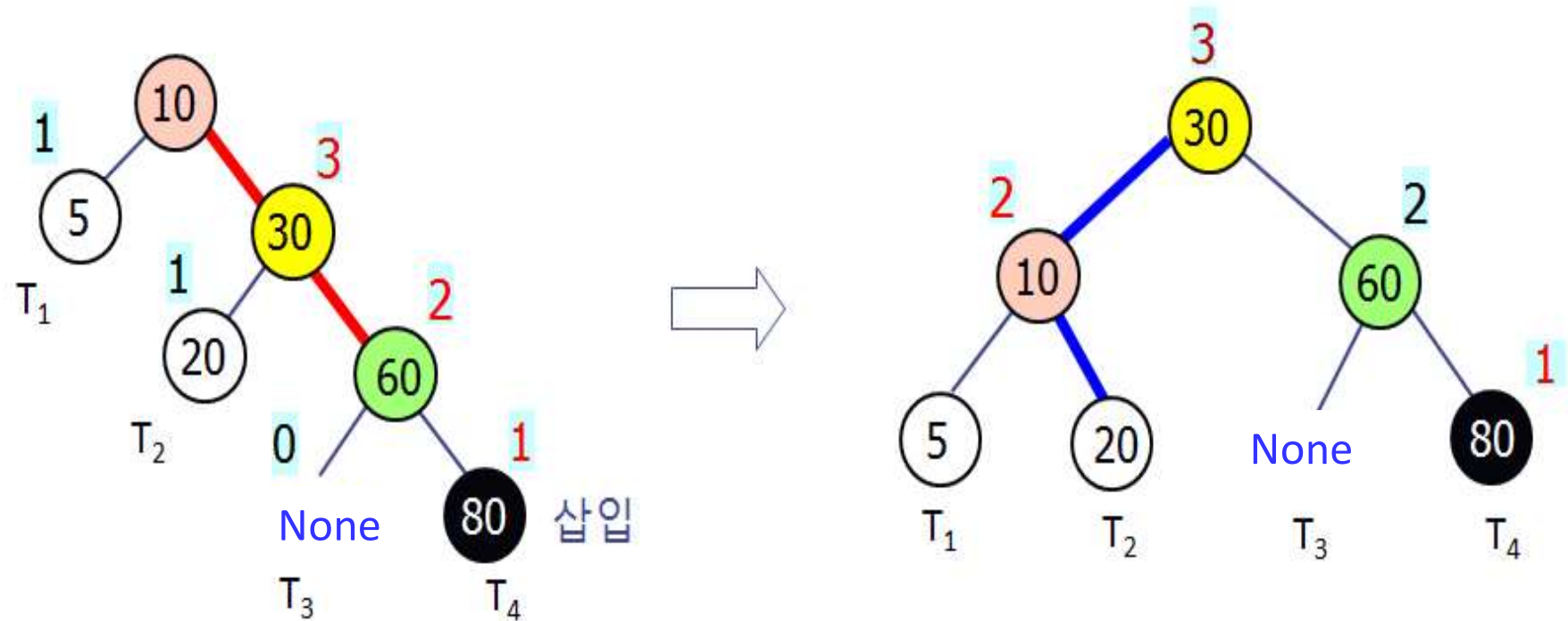


[그림 5-21] rotate\_left

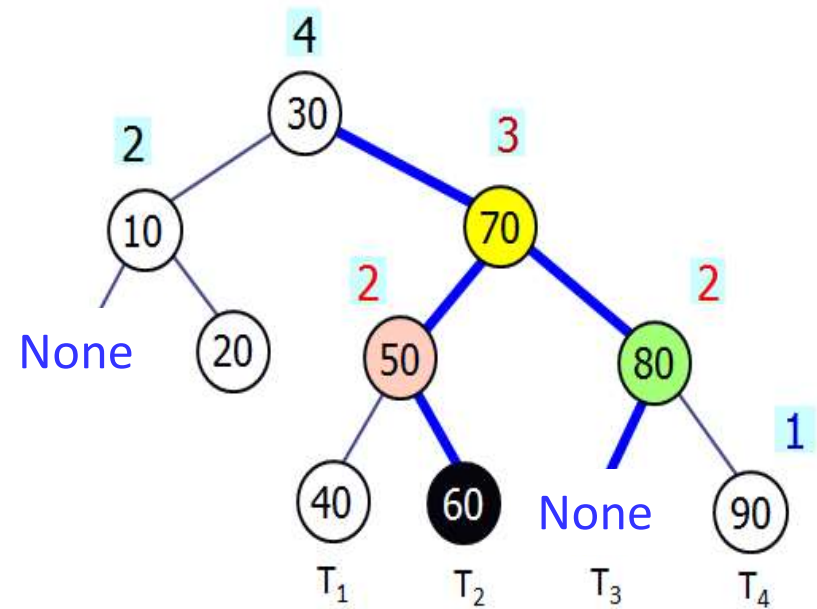
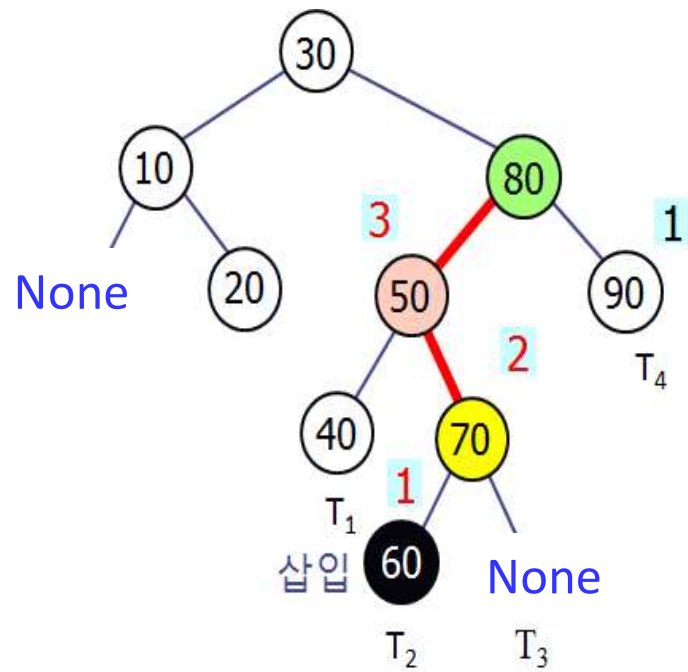
## [예제] LL-회전의 예



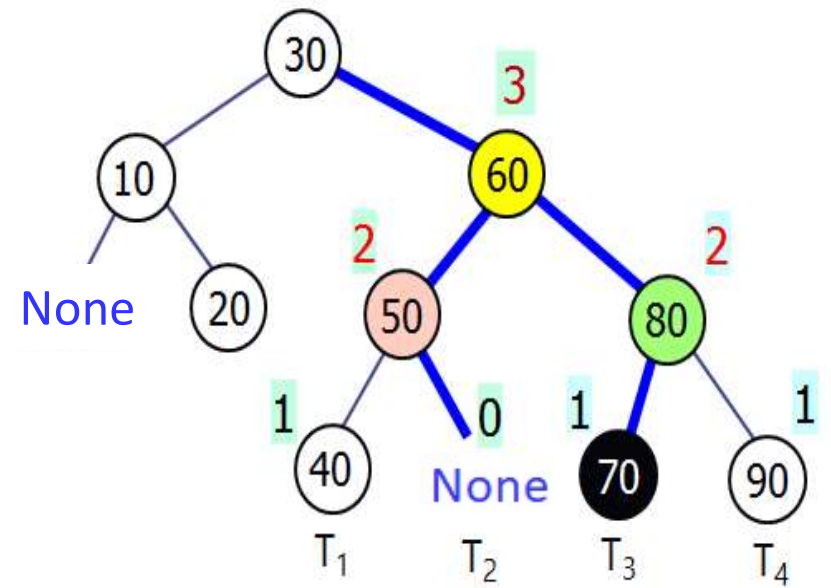
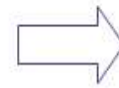
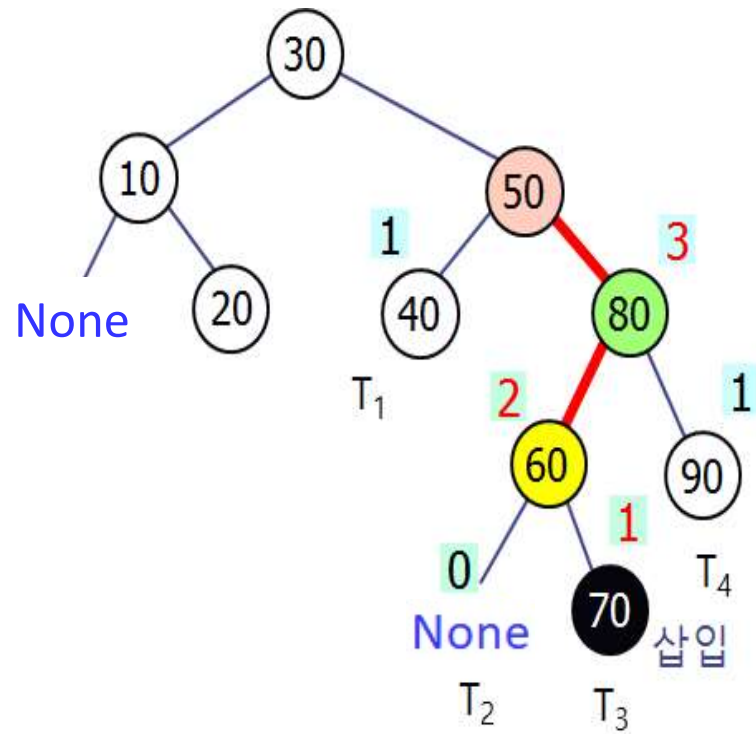
## [예제] RR-회전의 예



## [예제] LR-회전의 예



## [예제] RL-회전의 예





# AVL 트리를 위한 Node, AVL 클래스

```
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

```

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

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)

```



- `balance()`에서 line 02의 `bf(n) > 1`인 경우는 노드 `n`의 왼쪽 서브트리가 오른쪽 서브트리보다 높고, 그 차이가 1보다 큰 것으로 불균형 발생
- 이 때 `bf(n.left)`가 음수이면, `n.left`의 오른쪽 서브트리가 왼쪽 서브트리보다 높음
  - Line 04에서 `rotate_left(n.left)`를 수행하고 line 05에서 `rotate_right(n)`을 수행.  
즉, LR-회전 수행
- `bf(n.left)`가 음수가 아니라면, line 05에서 LL-회전 만을 수행
- RR-회전과 RL-회전도 line 08~10에 따라 각각 수행되어 트리의 균형을 유지
- 참고로 현재 노드 `n`의 균형이 유지되어 있으면, 바로 line 11에서 노드 `n`의 레퍼런스를 리턴

```
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)
```

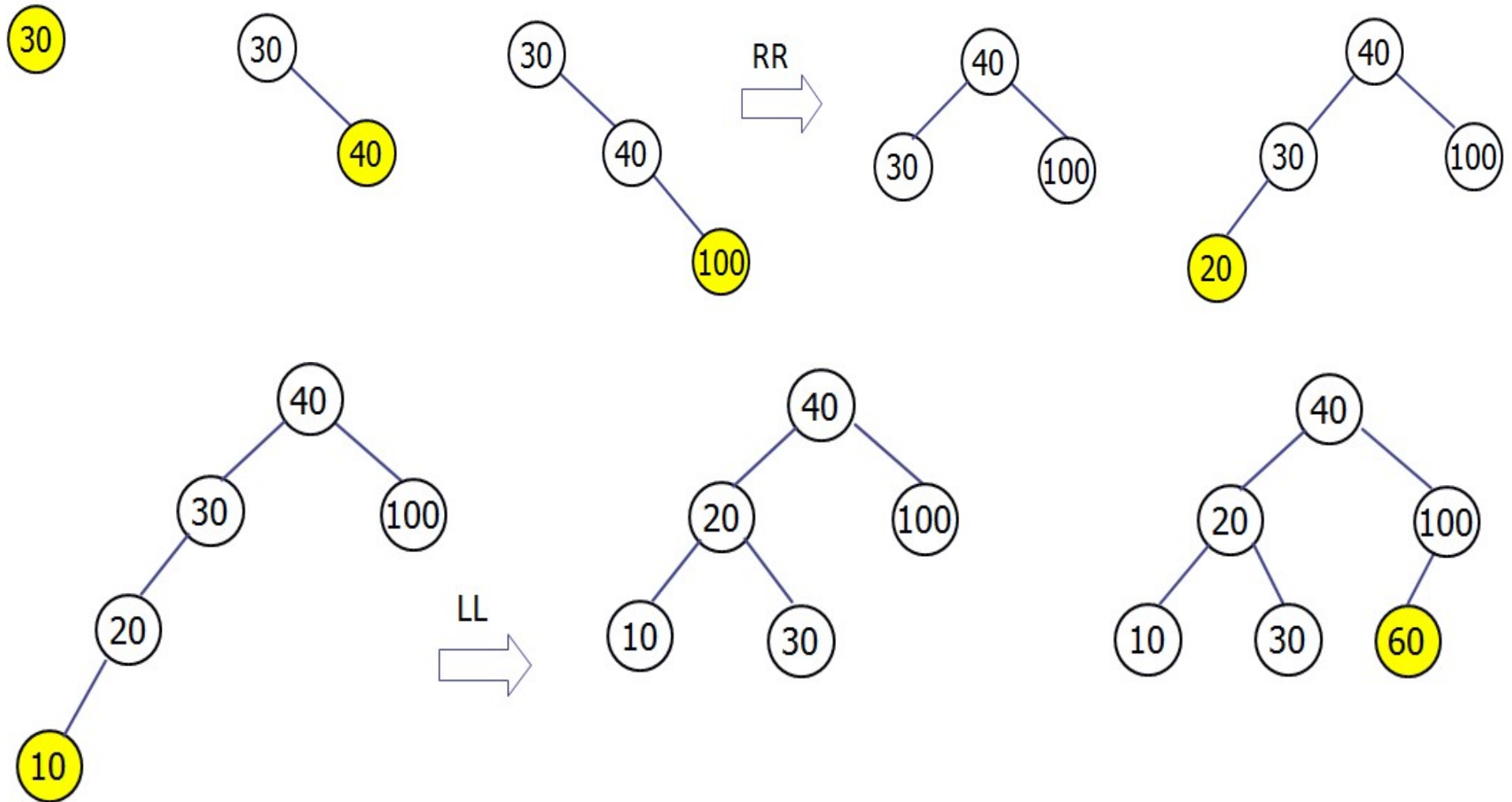
새 노드 생성,  
높이=1

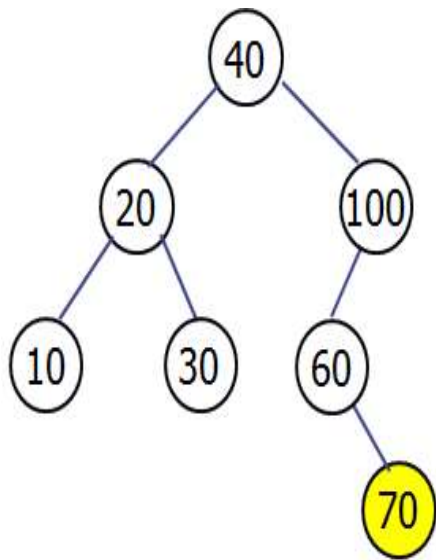
key가 이미 있으면  
value만 갱신

노드 n의 높이 갱신

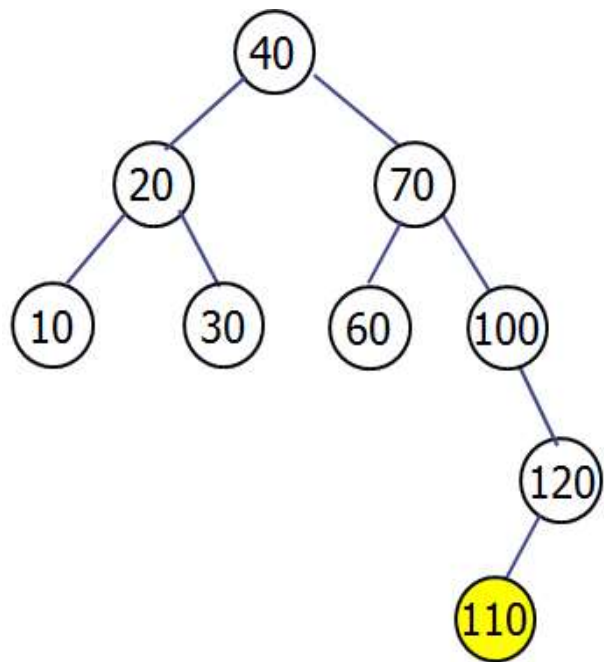
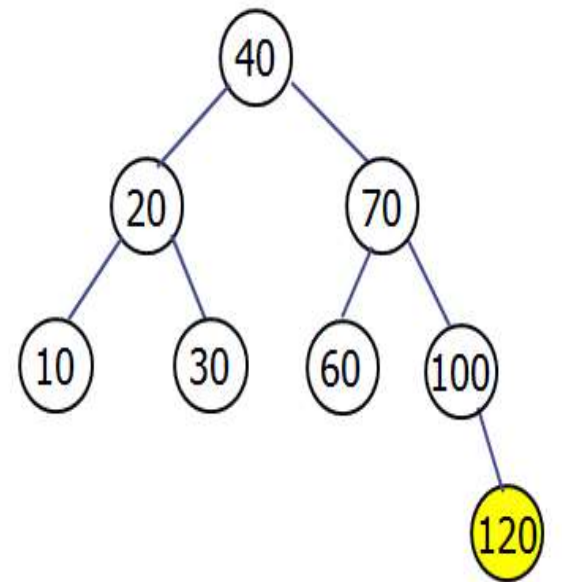
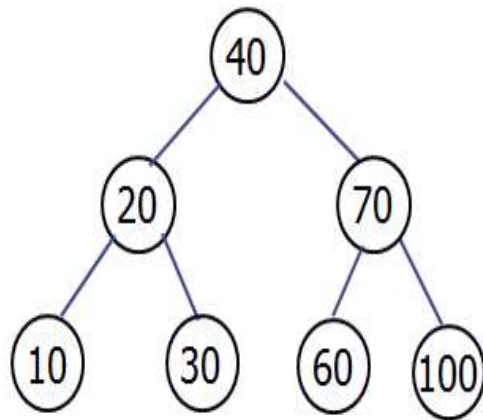
노드 n의 균형 유지

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

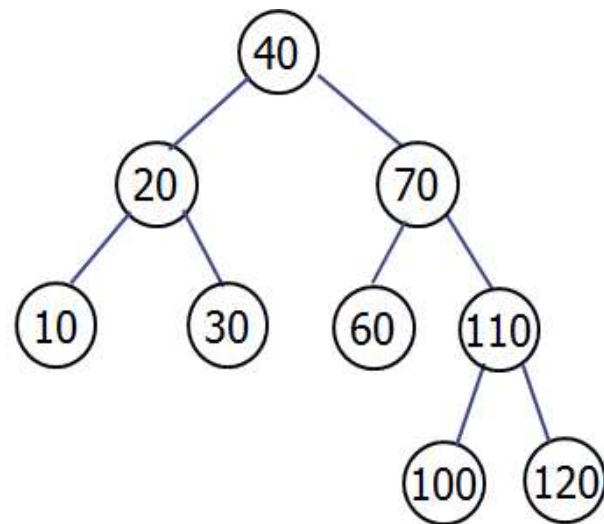




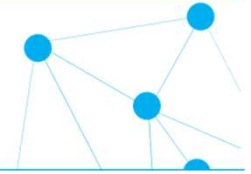
LR  
⇒



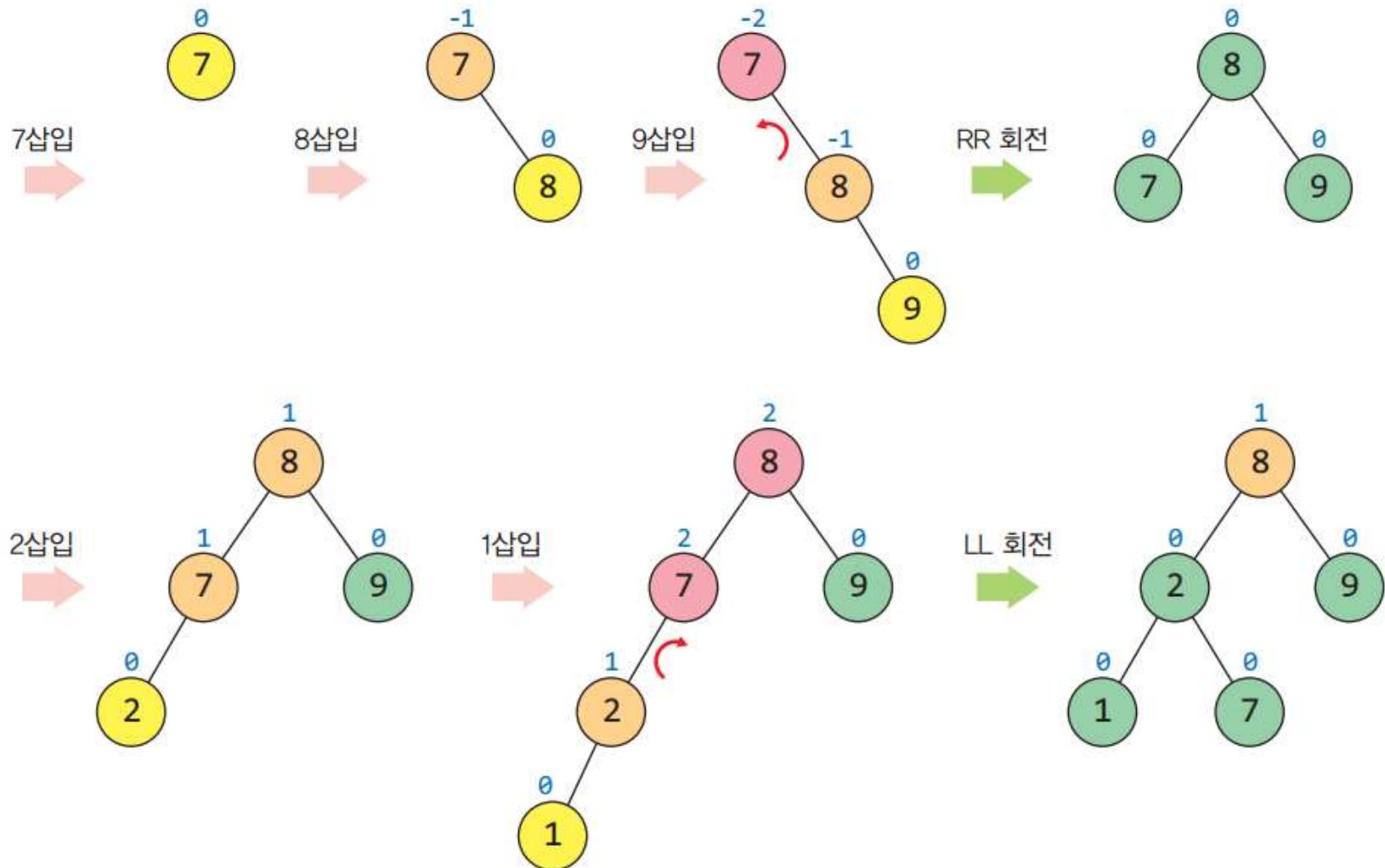
RL  
⇒



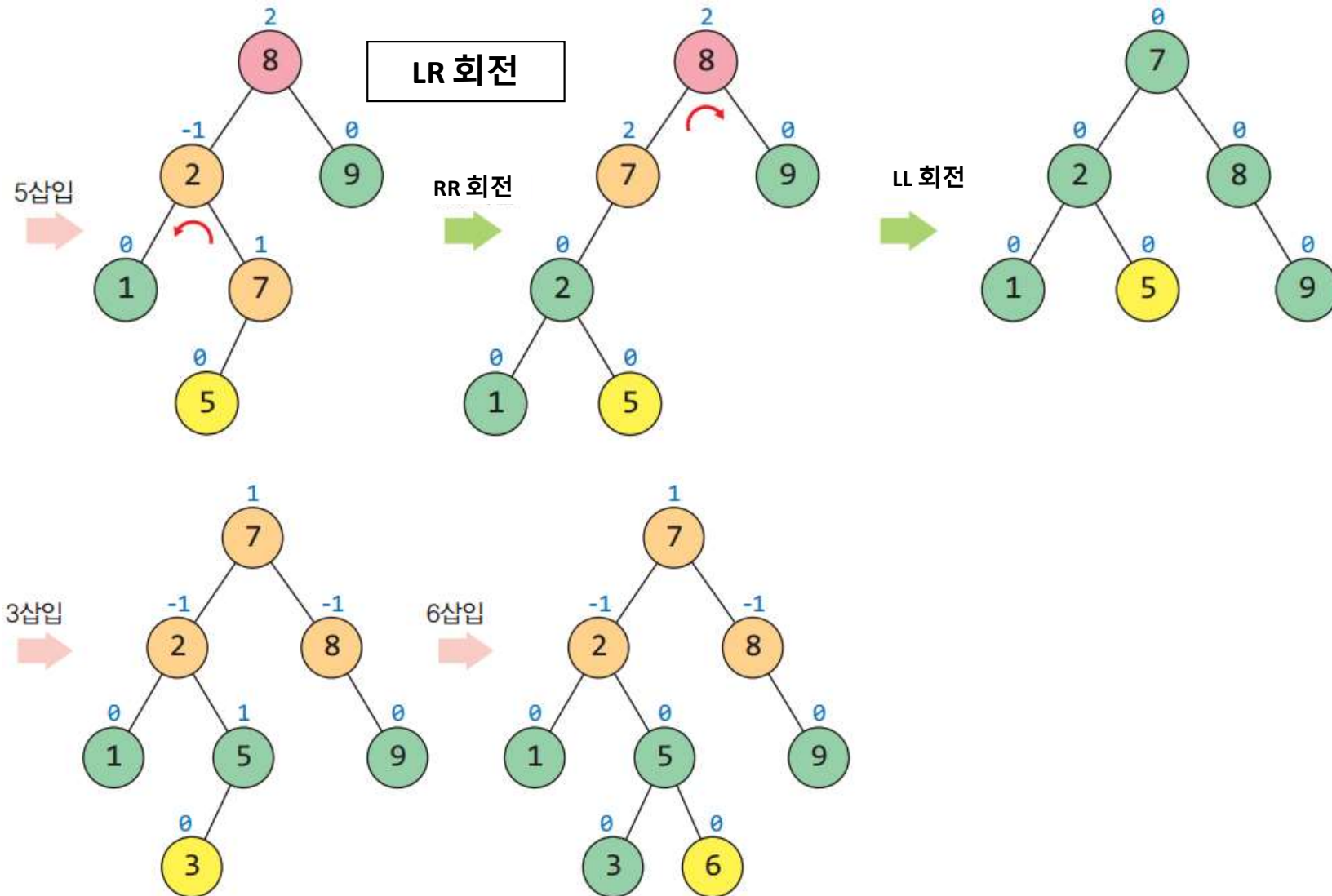
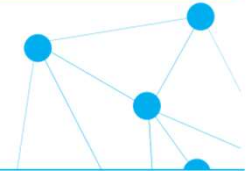
# AVL 트리 구축의 예



[7, 8, 9, 2, 1, 5, 3, 6, 4]

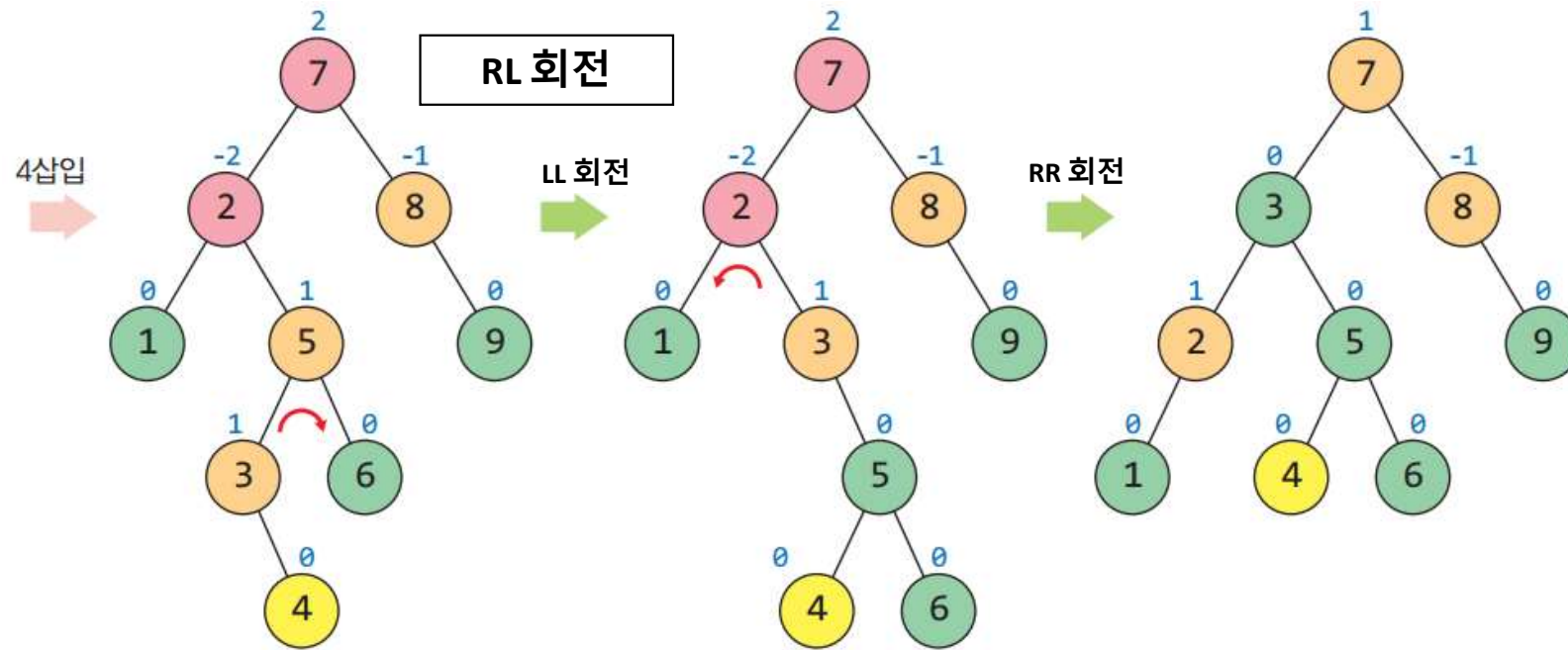
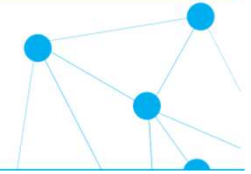


# AVL트리 구축의 예(계속)





# AVL트리 구축의 예(계속)

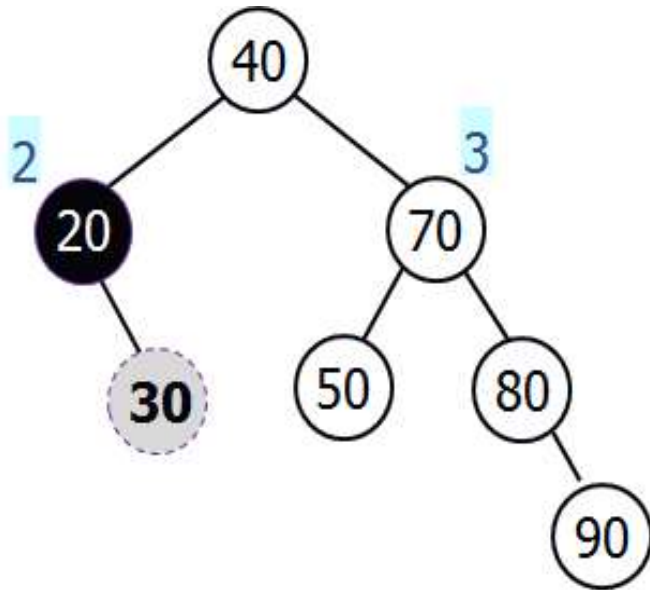


# 삭제 연산

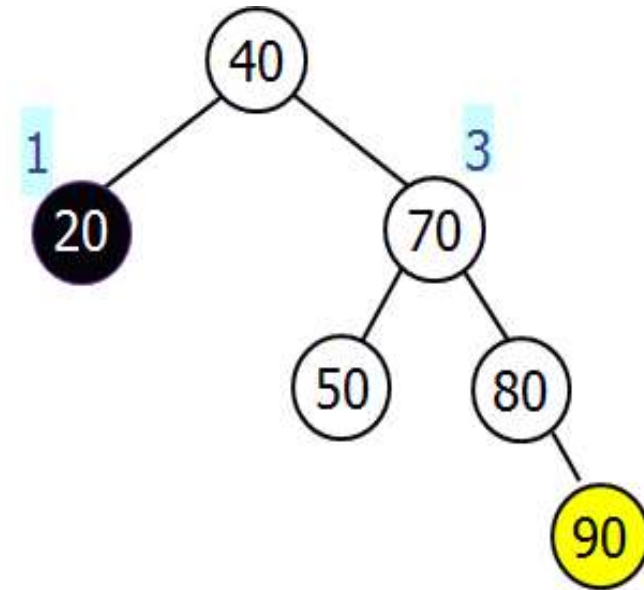
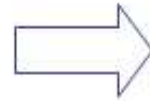
- AVL트리에서의 삭제는 두 단계로 진행
- [1단계] 이진탐색트리에서와 동일한 삭제 연산 수행
- [2단계] 삭제된 노드로부터 루트노드 방향으로 거슬러 올라가며 불균형이 발생한 경우 적절한 회전 연산 수행
  - 회전 연산 수행 후에 부모에서 불균형이 발생할 수 있고, 이러한 일이 반복되어 루트에서 회전 연산을 수행해야 하는 경우도 발생



## 30을 가진 노드의 삭제



(a) 삭제 전

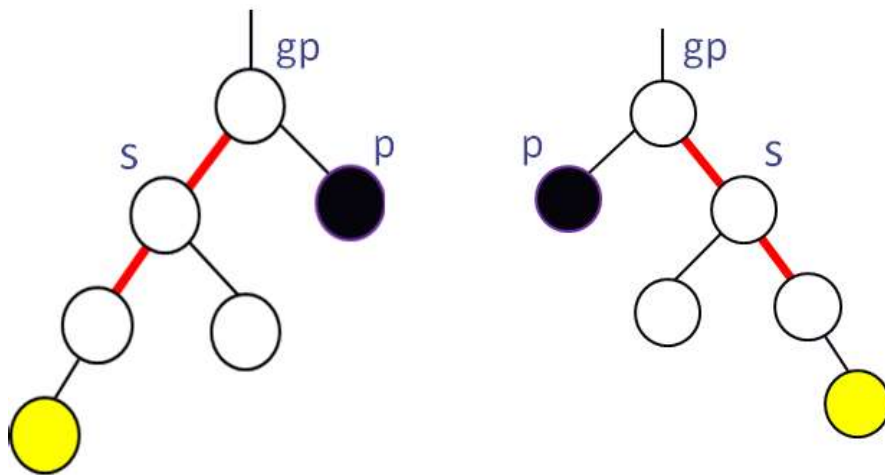


(b) 삭제 후 노드 40에서 불균형 발생

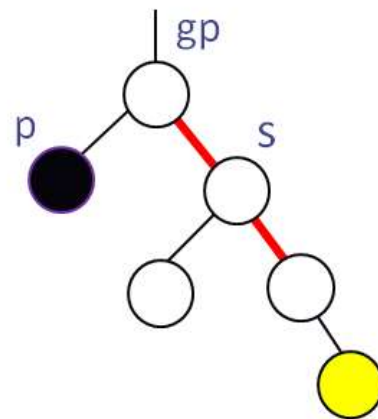
## [핵심 아이디어]

삭제 후 불균형이 발생하면 반대쪽에 삽입이 이루어져 불균형이 발생한 것으로 취급하자.

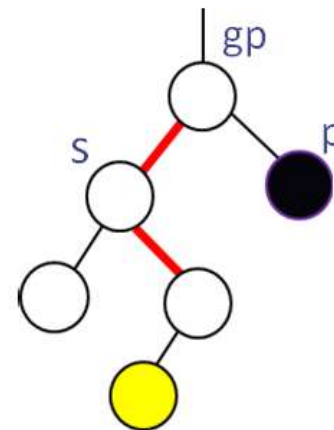
- 삭제된 노드의 부모 =  $p$ ,  $p$ 의 부모 =  $gp$ ,  $p$ 의 형제 =  $s$
- $s$ 의 왼쪽과 오른쪽 서브트리 중에서 높은 서브트리에 마치 새 노드가 삽입된 것으로 간주



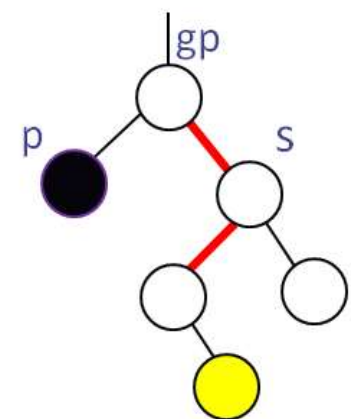
(a) LL-회전



(b) RR-회전

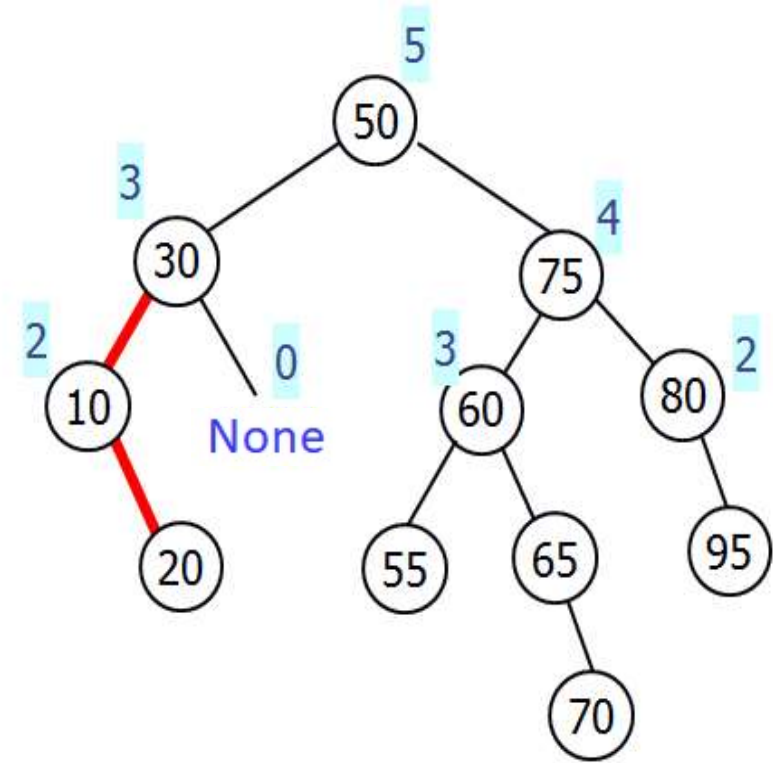
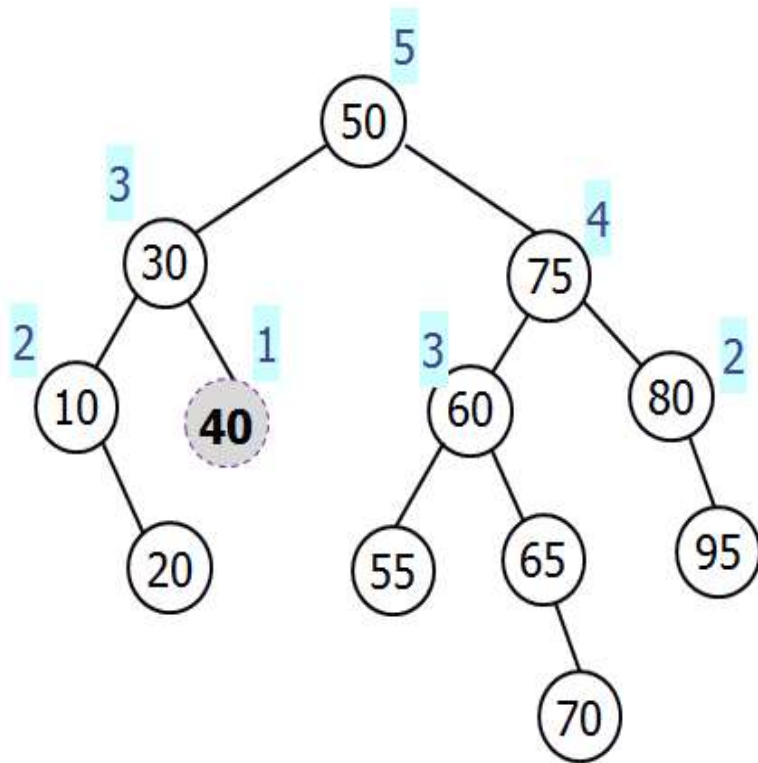


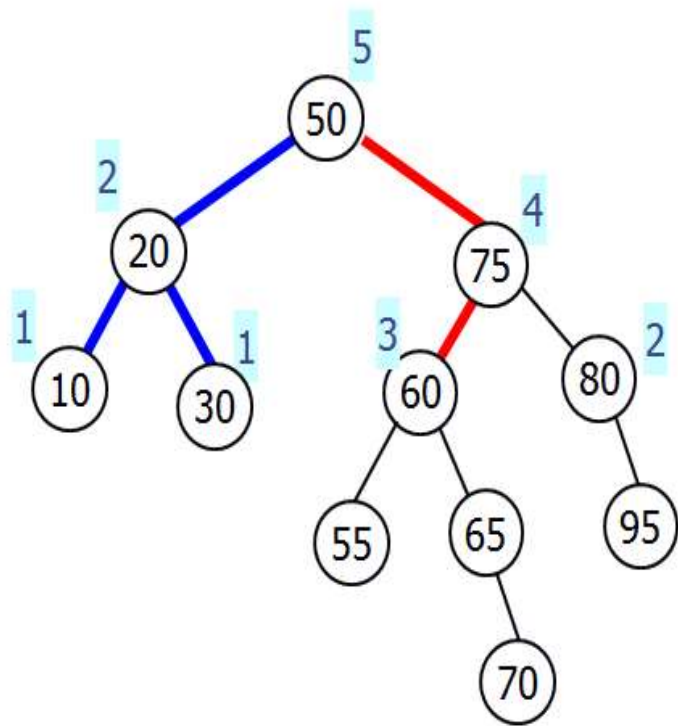
(c) LR-회전



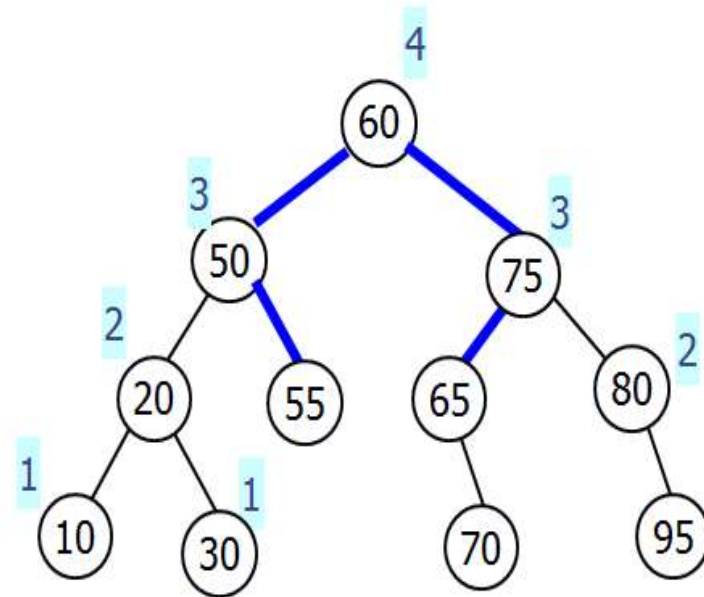
(d) RL-회전

## [예제] 40 삭제





LR-회전 후



RL-회전 후



# 수행시간

- AVL 트리에서의 탐색, 삽입, 삭제 연산은 공통적으로 루트부터 탐색을 시작하여 최악의 경우에 이파리까지 내려가고, 삽입이나 삭제 연산은 다시 루트까지 거슬러 올라가야 함.
- 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  $O(\log N)$

***FINISH***

