



Search Tree

Jin Hyun Kim
Fall, 2019

In this Class

- Binary Search Tree
- AVL Tree, 2-3 Tree, Red Black Tree
- B-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
```

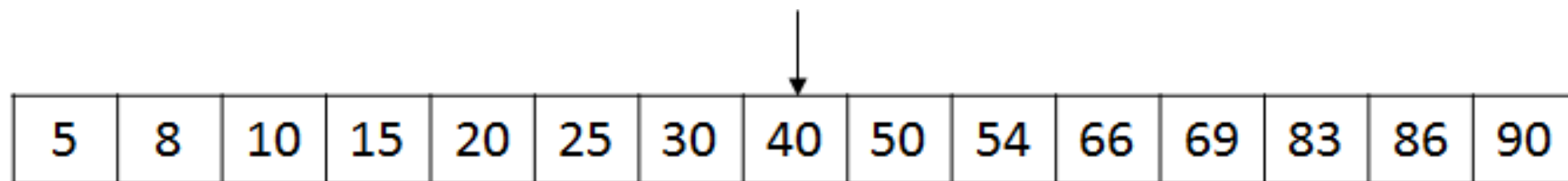
```
[3] if a[mid] == t: return mid # 탐색 성공
```

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

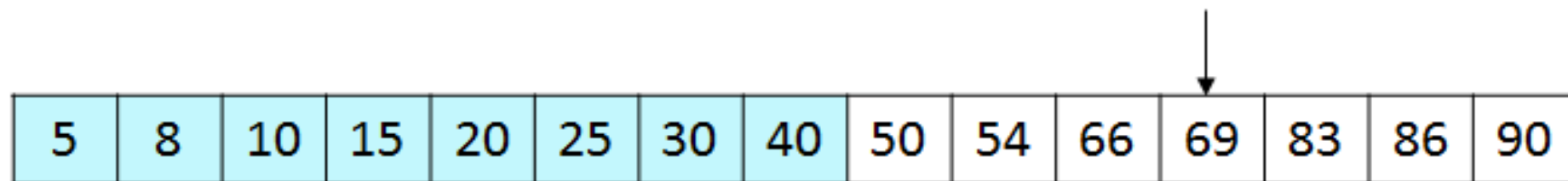
```
[5] else: binary_search(mid+1, right, t) # 뒷부분 탐색
```

이진탐색의 예

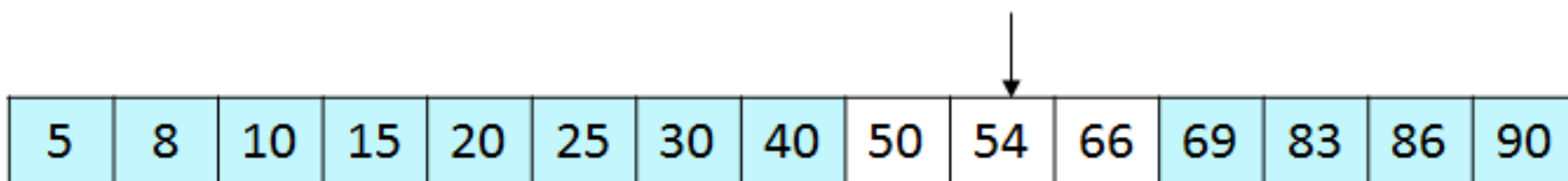
- 66을 찾아 가는 과정



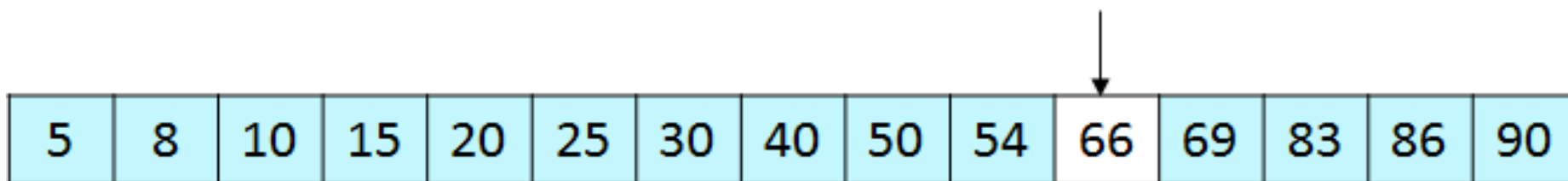
| | | | | | | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 5 | 8 | 10 | 15 | 20 | 25 | 30 | 40 | 50 | 54 | 66 | 69 | 83 | 86 | 90 |
|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|



| | | | | | | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 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

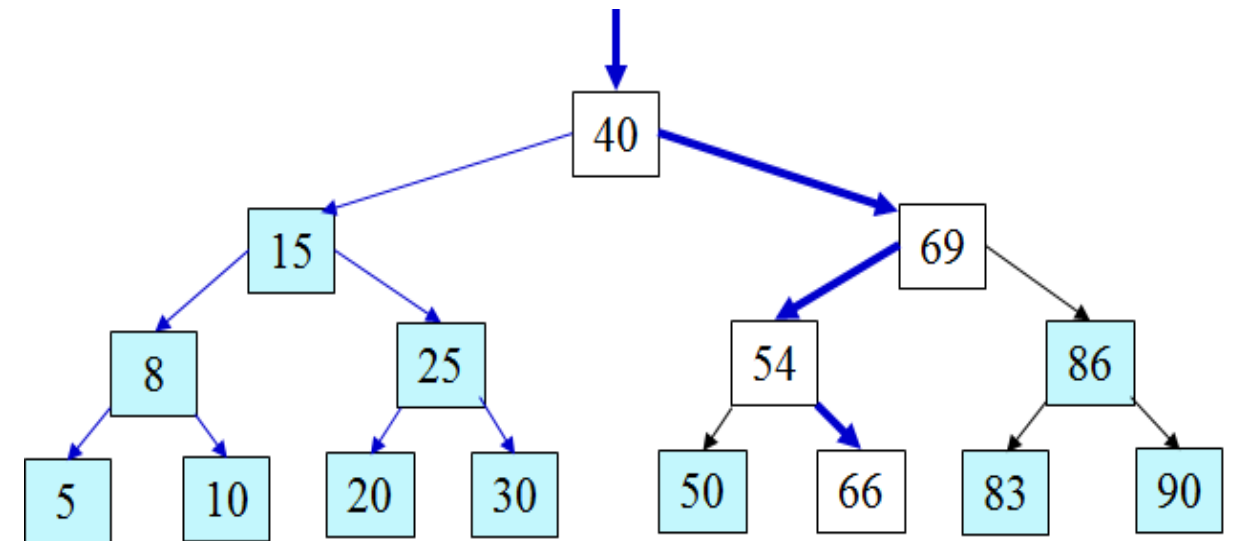
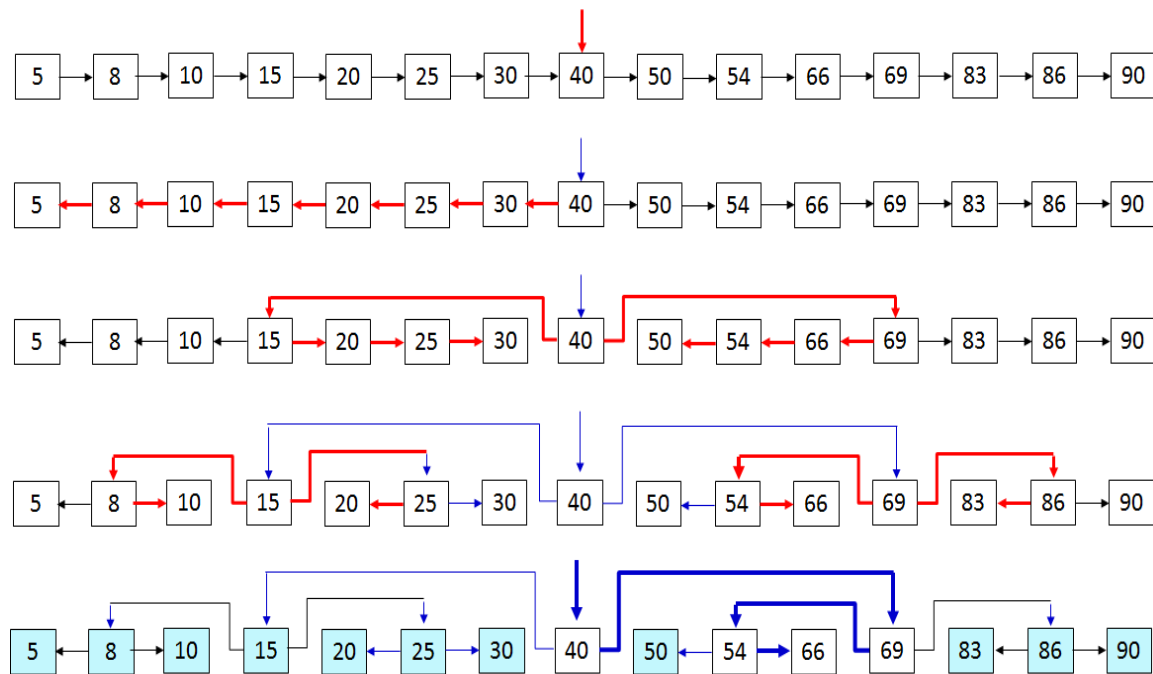
$$\begin{aligned} T(N) &= T(N/2) + 1 & T(N) &= T(N/2) + 1 \\ T(1) &= 1 & &= [T((N/2)/2) + 1] + 1 = T(N/2^2) + 2 \\ & & &= [T((N/2)/2^2) + 1] + 2 = T(N/2^3) + 3 \\ & & &= L = T(N/2^k) + k \\ & & &= T(1) + k, \text{ if } N = 2^k, k = \log_2 N \\ & & &= 1 + \log_2 N = O(\log N) \end{aligned}$$

이진탐색트리

Binary Search Tree

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

이진탐색과 이진트리



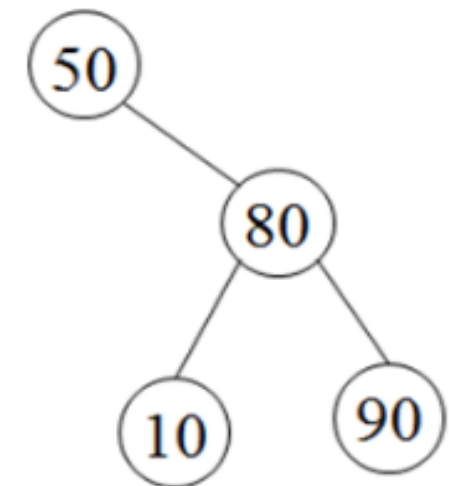
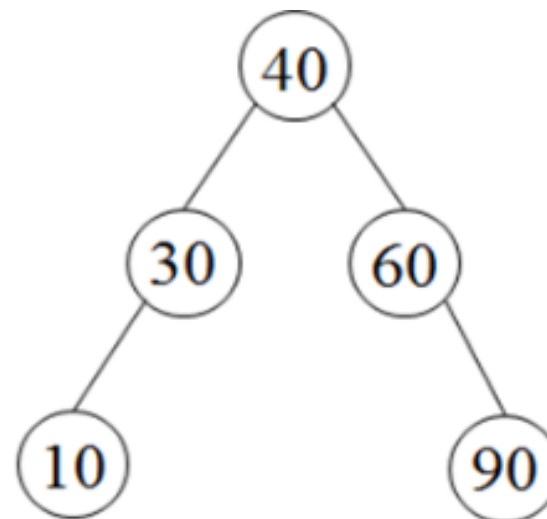
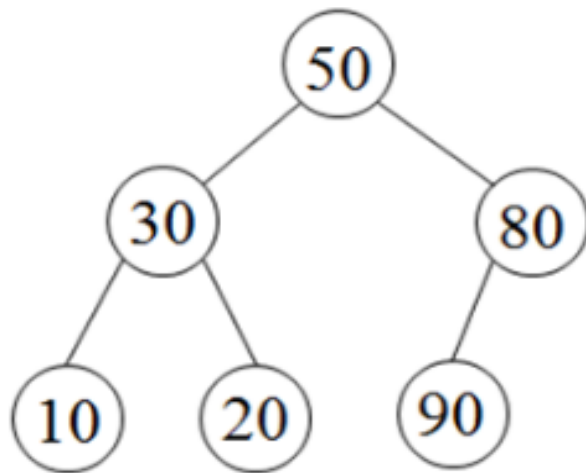
이진탐색트리

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

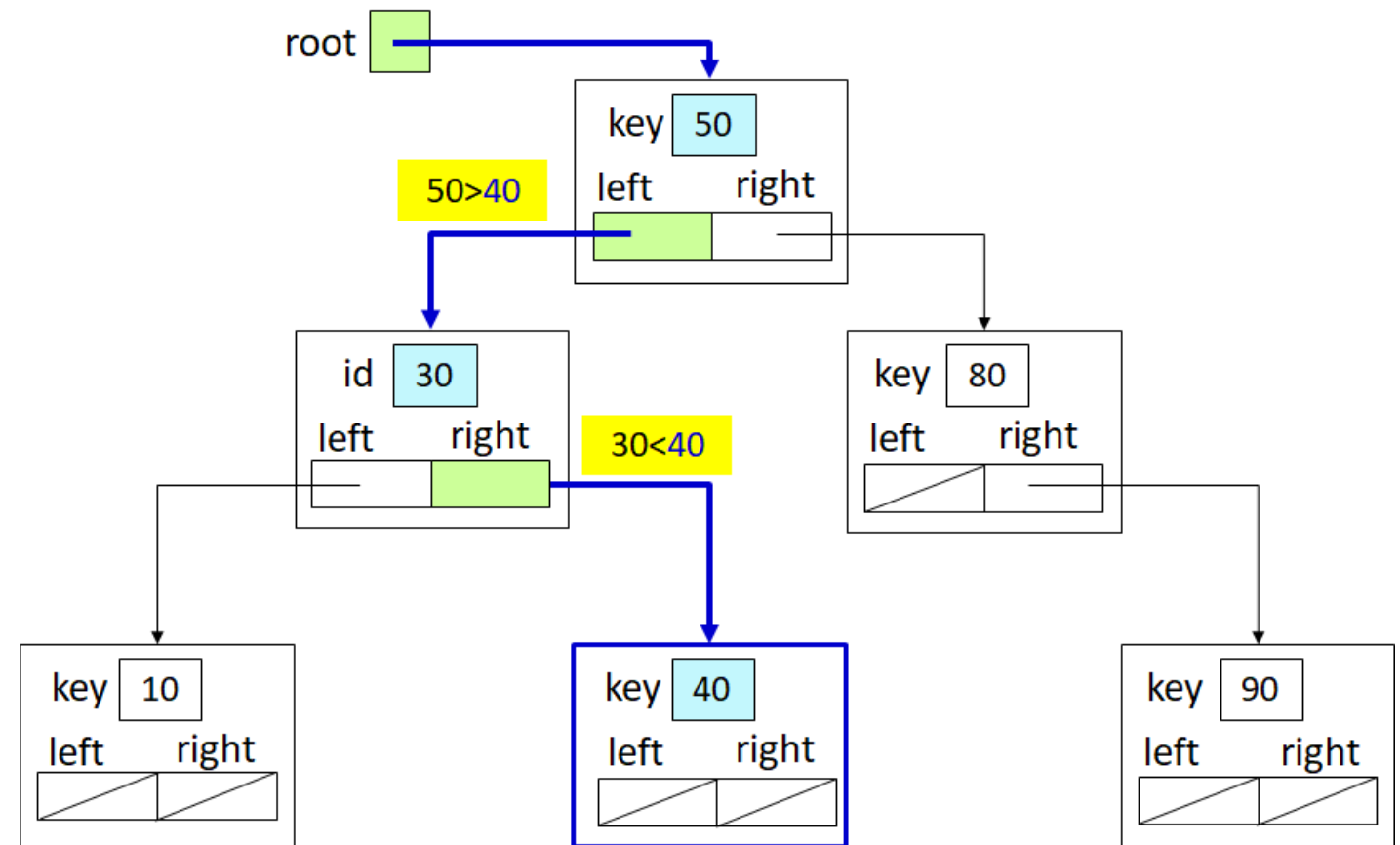
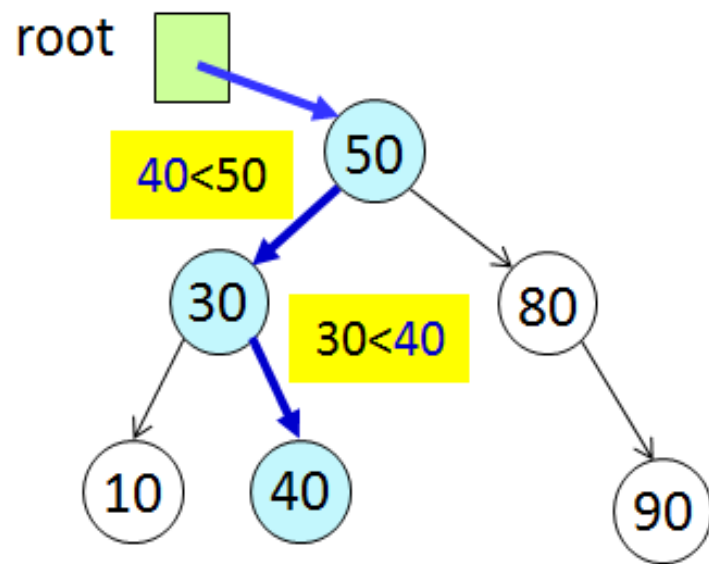
트리 루트

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

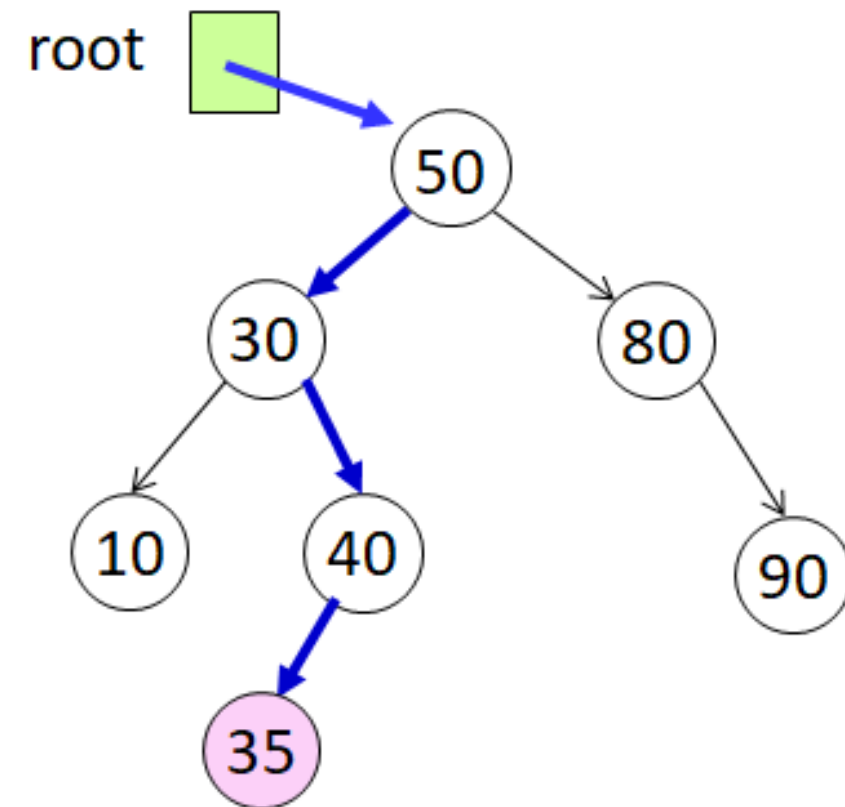
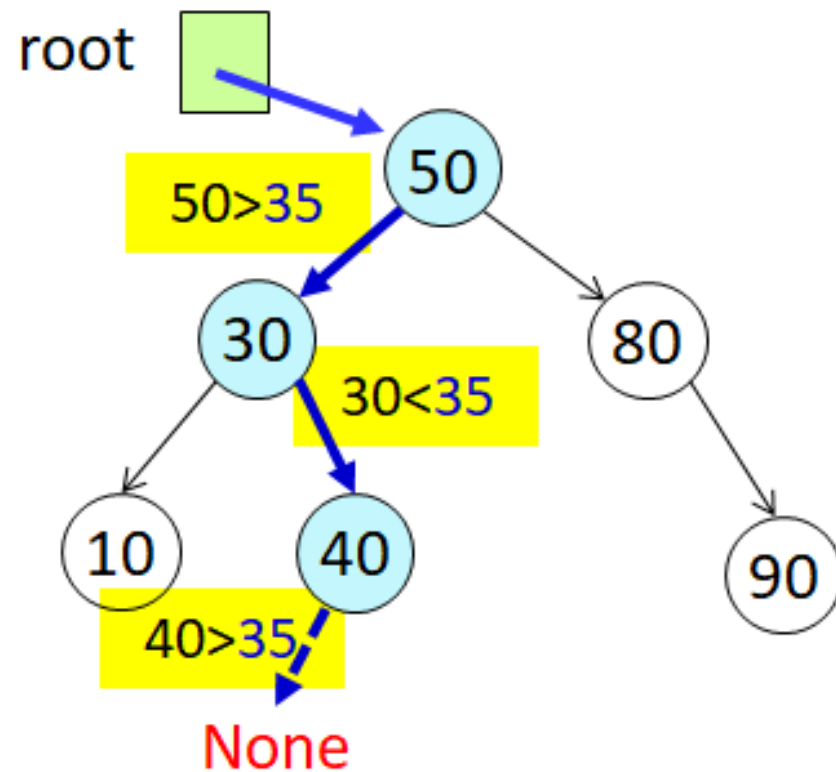
k가 노드의 key보다 크면
오른쪽 서브트리 탐색

탐색 성공

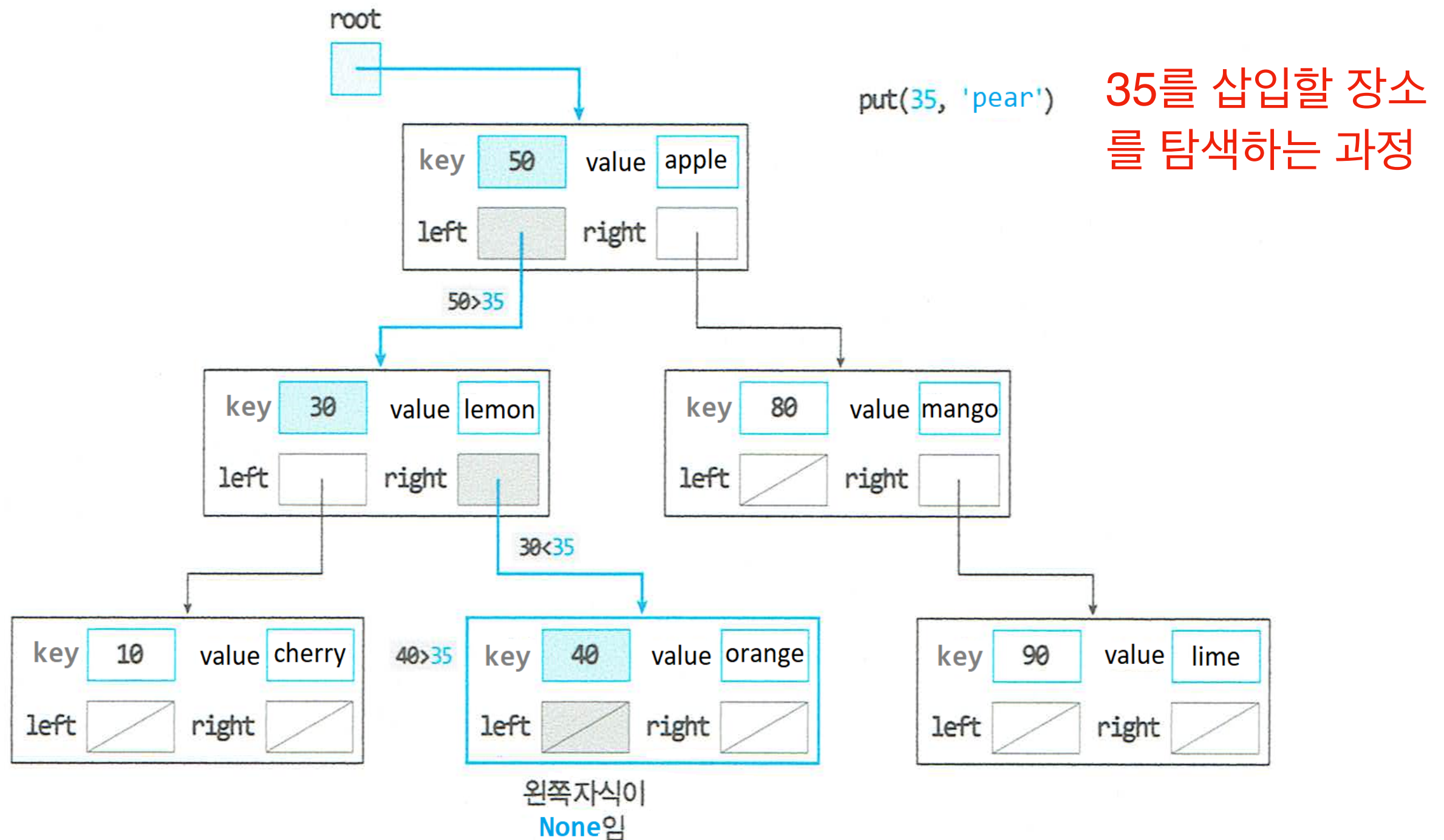
삽입연산: `put(key, value)`

- 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)

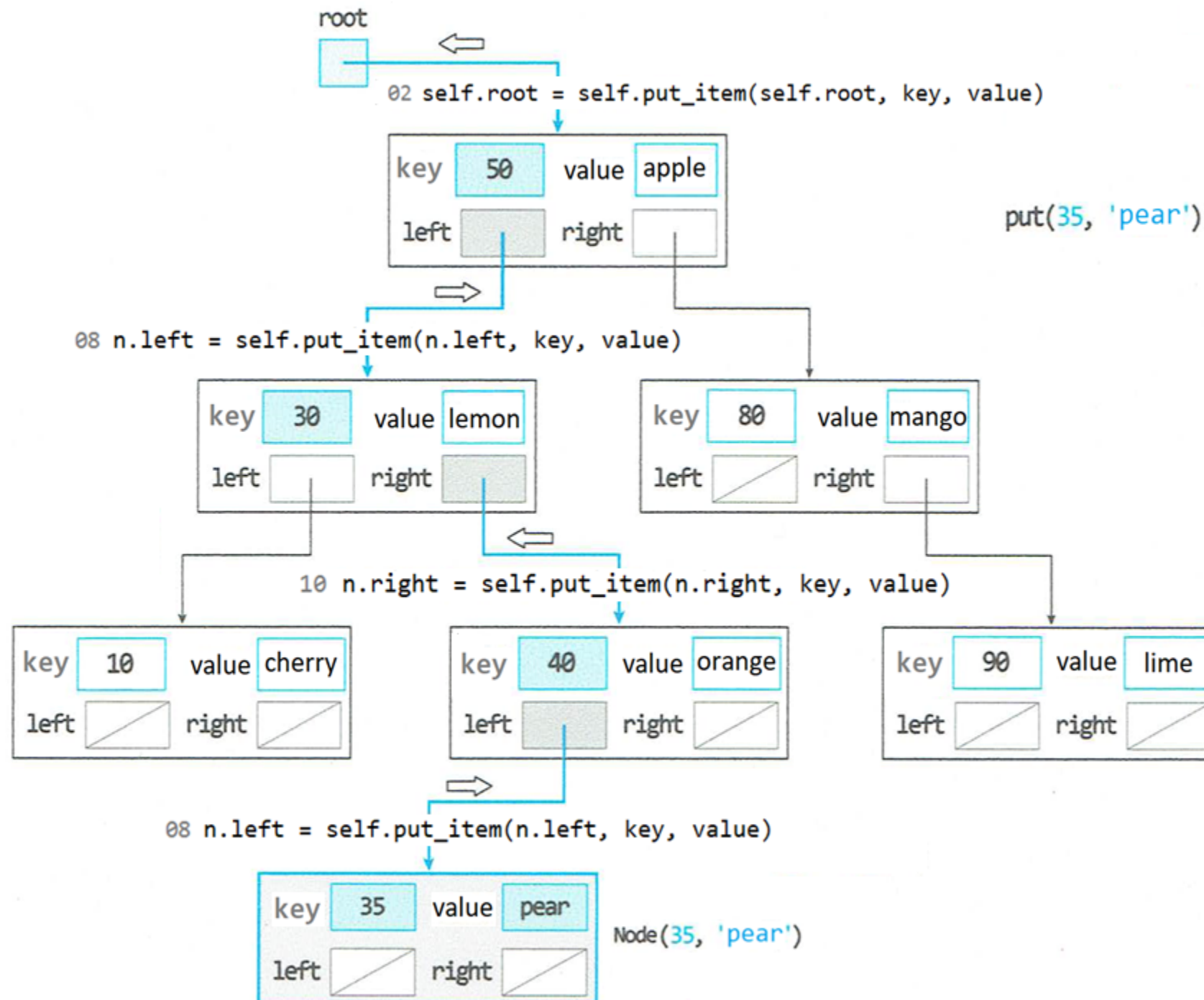


삽입연산: put(key, value)



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

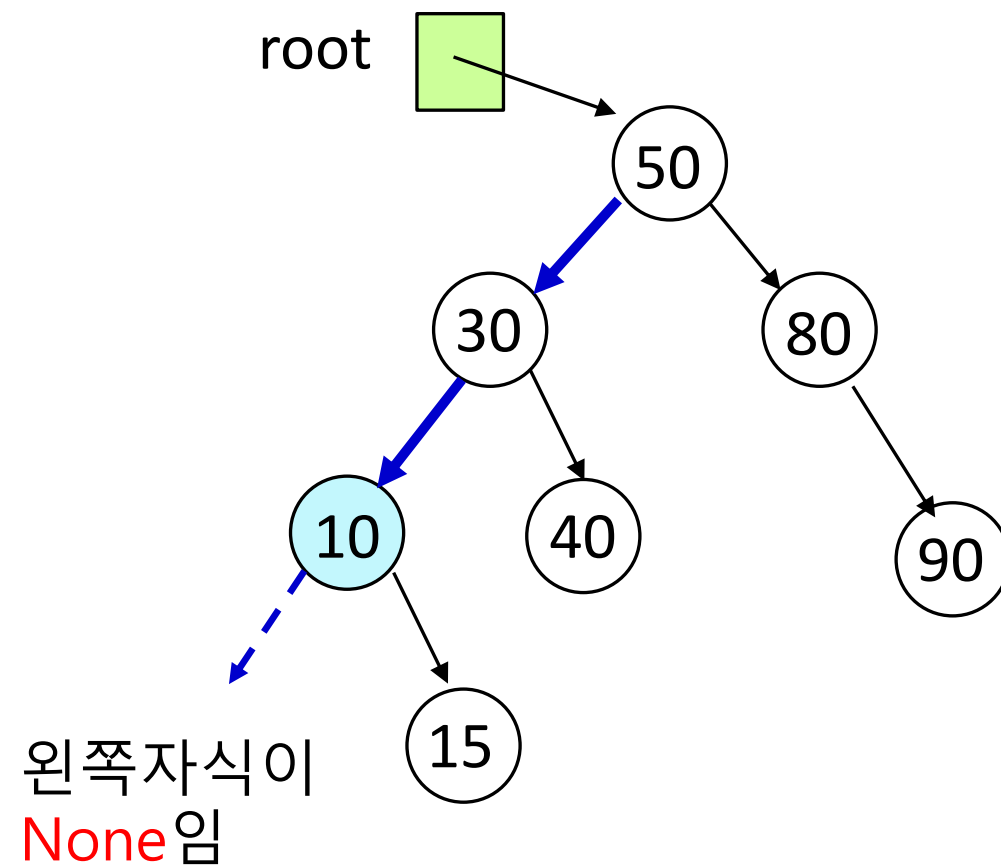
n의 오른쪽자식과 put_item()이 리턴하는 노드를 재 연결

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

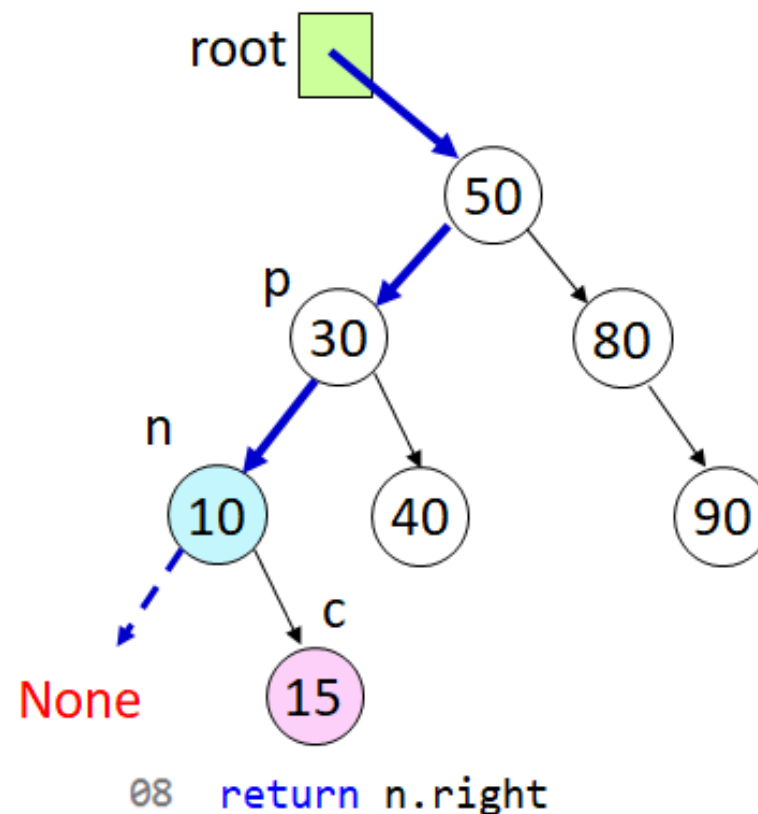
```
01 def min(self): # 최솟값 가진 노드 찾기
02     if self.root == None:
03         return None
04     return self.minimum(self.root)
05
06 def minimum(self, n):
07     if n.left == None:
08         return n
09     return self.minimum(n.left)
```

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 None 이더라도 자식으로 연결

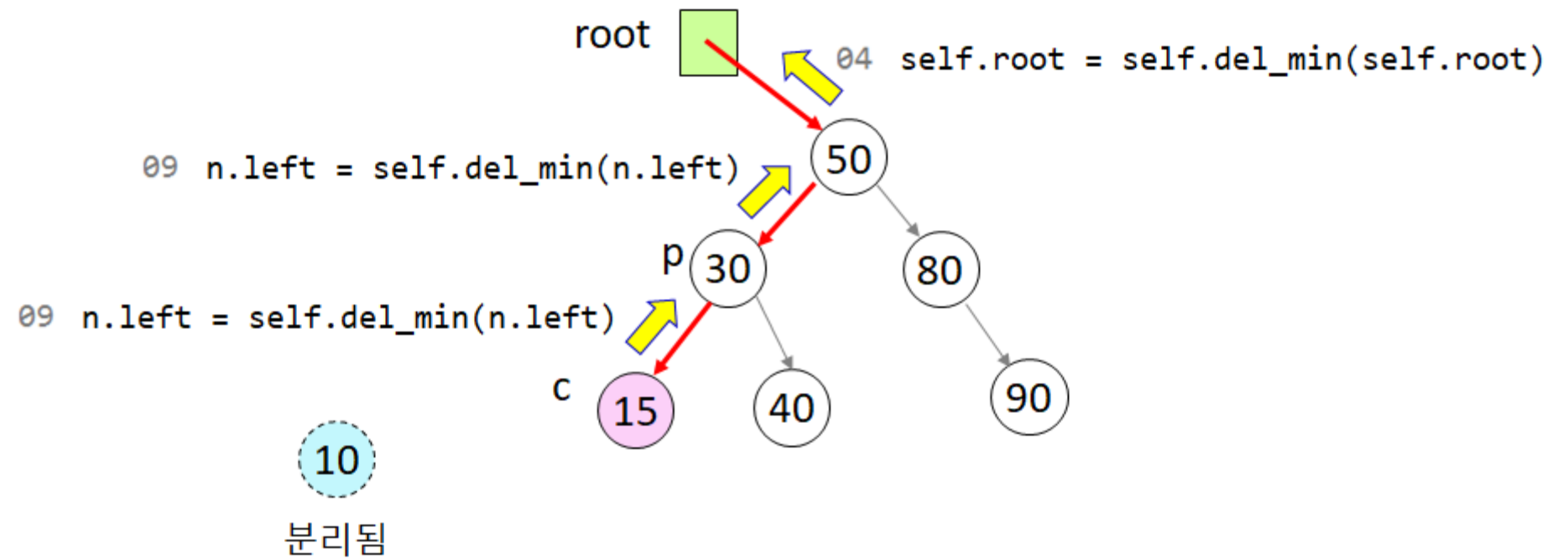
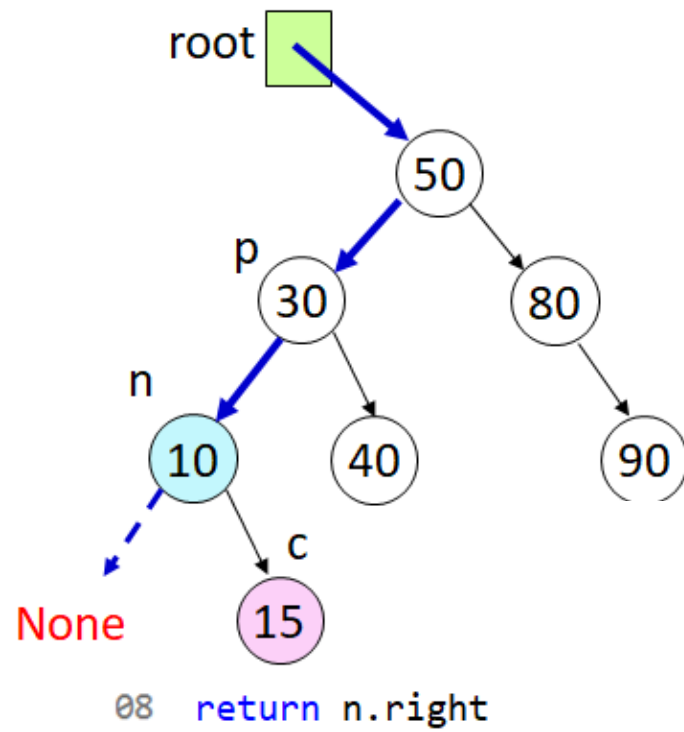
```
01 def delete_min(self): # 최솟값 삭제
02     if self.root == None:
03         print('트리가 비어 있음')
04     self.root = self.del_min(self.root)
05
06 def del_min(self, n):
07     if n.left == None:
08         return n.right
09     n.left = self.del_min(n.left)
10     return n
```

루트와 `del_min()`이 리턴하는 노드를 재 연결

최솟값 가진 노드의 오른쪽 자식을 리턴

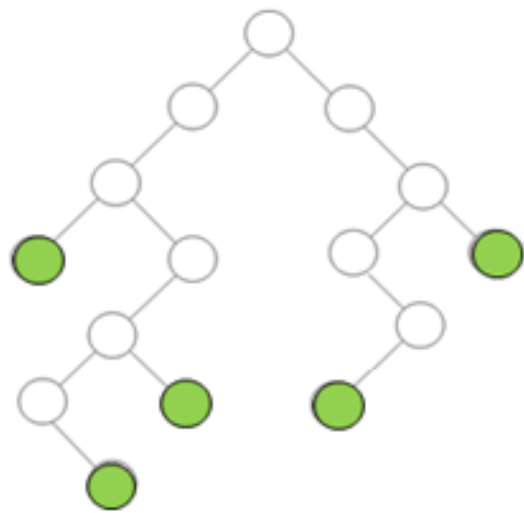
n 의 왼쪽자식과 `del_min()`이 리턴하는 노드를 재 연결

최소값 삭제

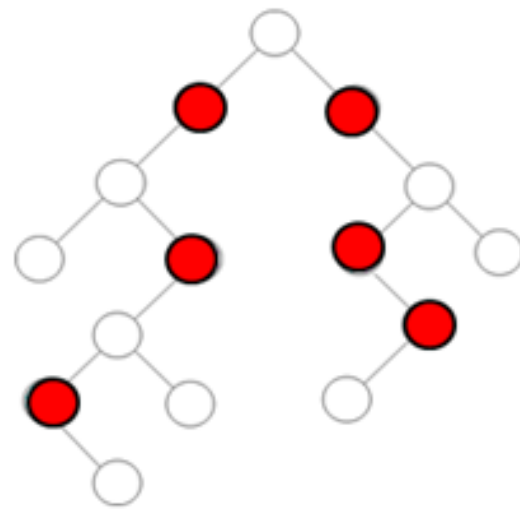


삭제연산: delete(key)

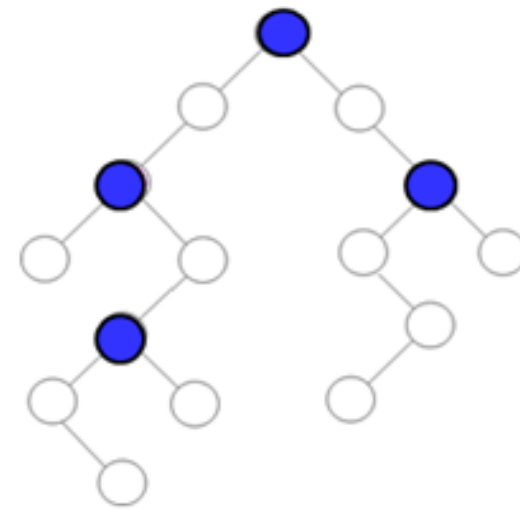
- 우선 삭제하고자 하는 노드를 찾은 후 이진탐색트리 조건을 만족하도록 삭제된 노드의 부모와 자식(들)을 연결해 주어야 함
- 삭제되는 노드가 자식이 없는 경우(case 0), 자식이 하나인 경우(case 1), 자식이 둘인 경우(case 2)로 나누어 delete 연산을 수행



case 0



case 1

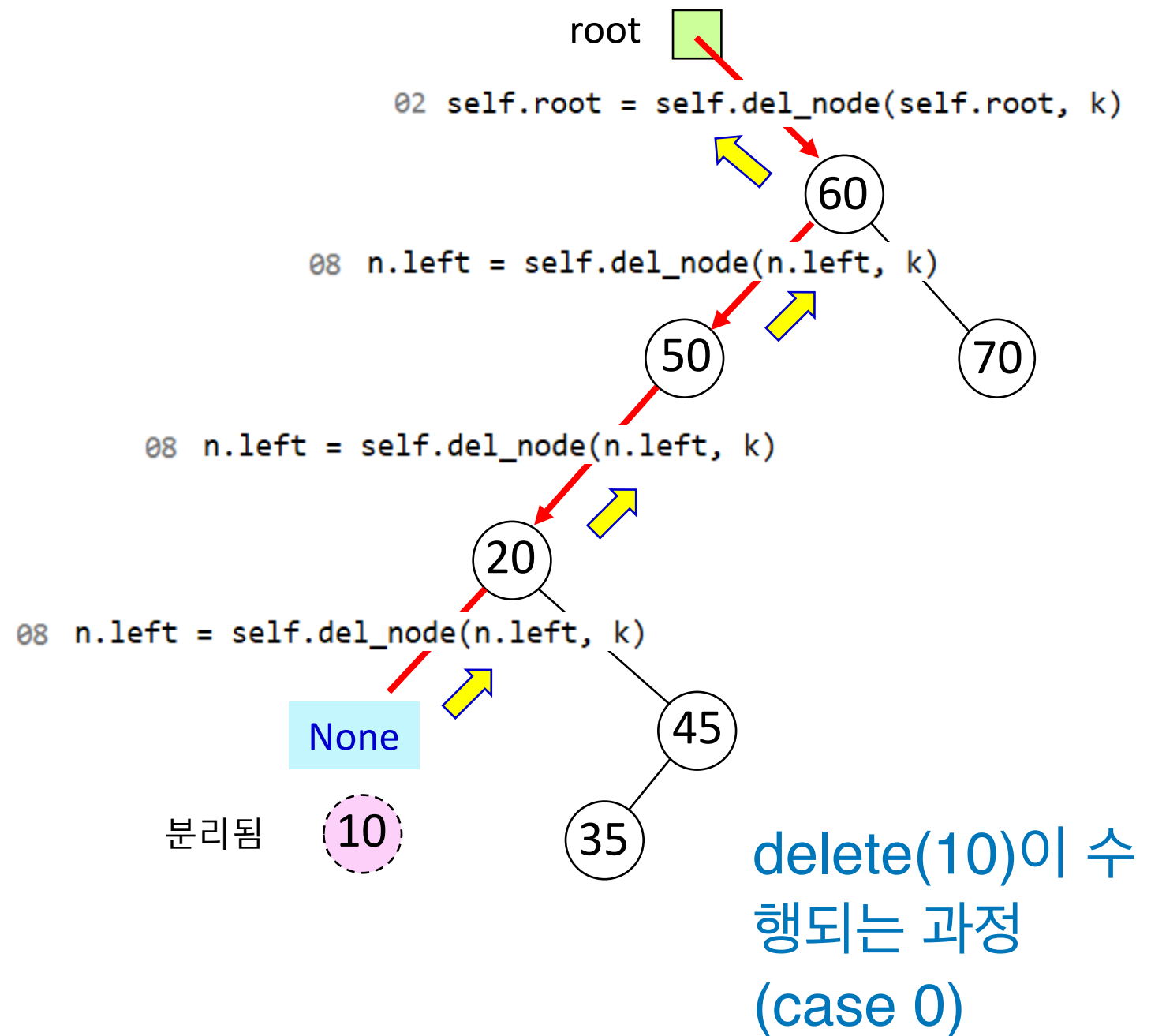
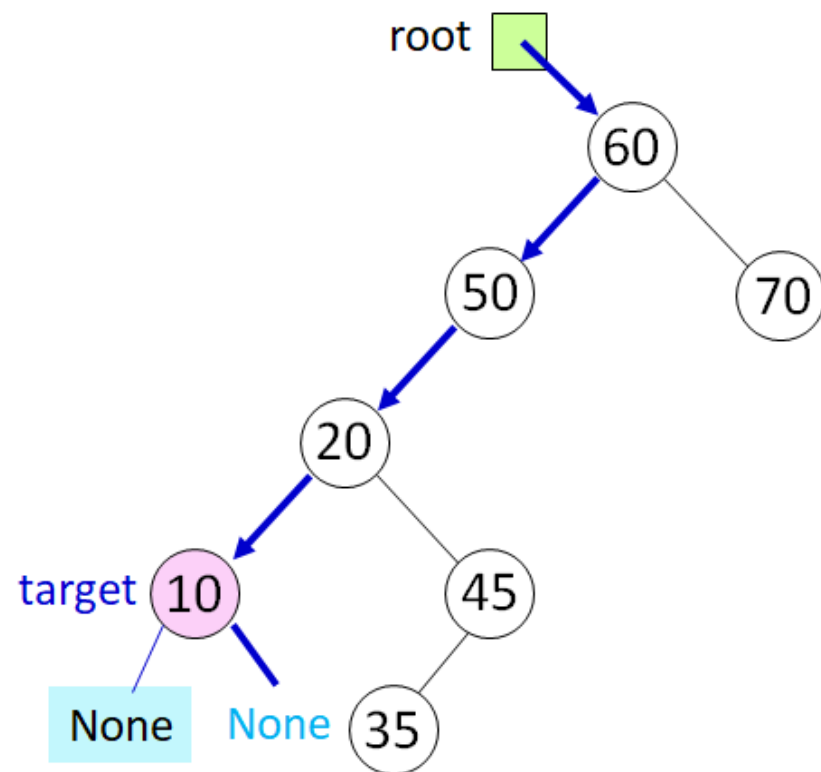


case 2

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)
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

n의 왼쪽자식과 del_node()가 리턴하는 노드를 재 연결

n의 오른쪽자식과 del_node()가 리턴하는 노드를 재 연결

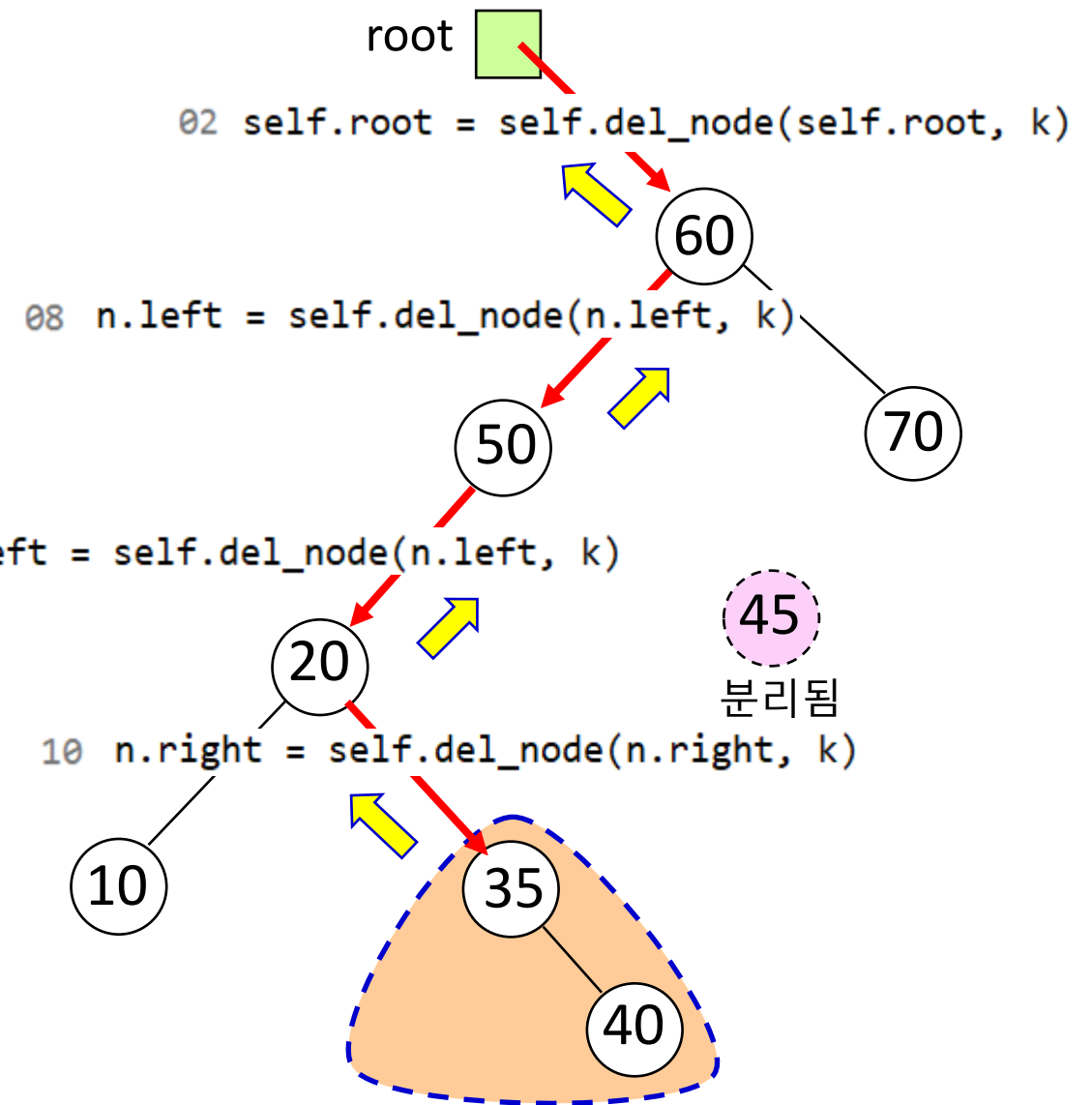
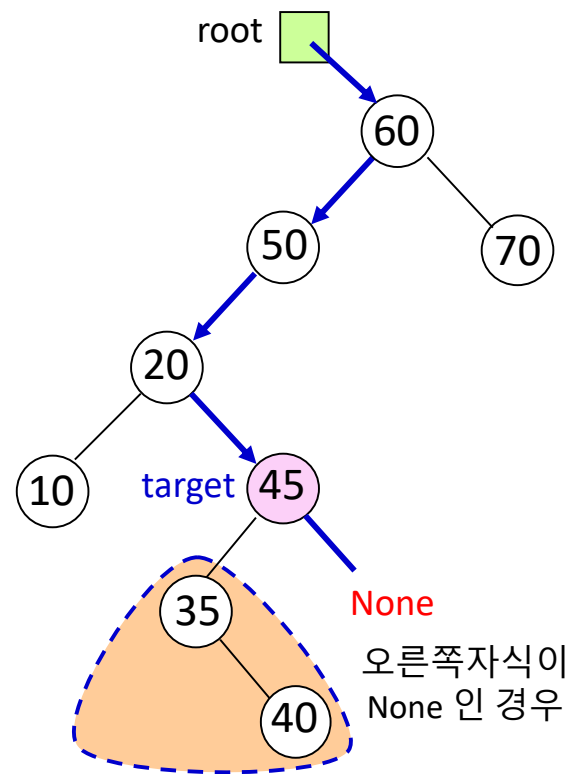
target은 삭제될 노드

target의 중위 후속자 찾아 n이 참조하게 함

n의 오른쪽자식과 target의 오른쪽자식 연결

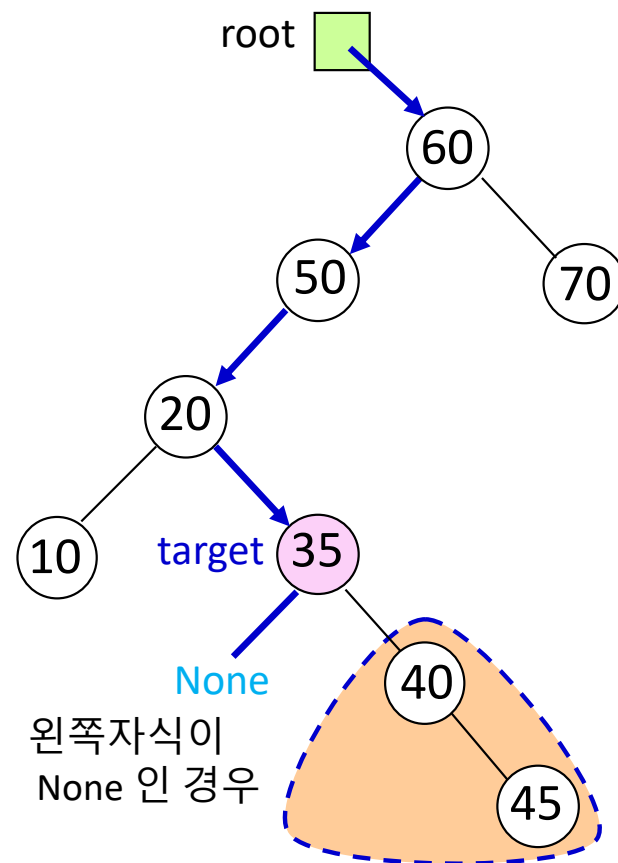
n의 왼쪽자식과 target의 왼쪽자식 연결

Case 1

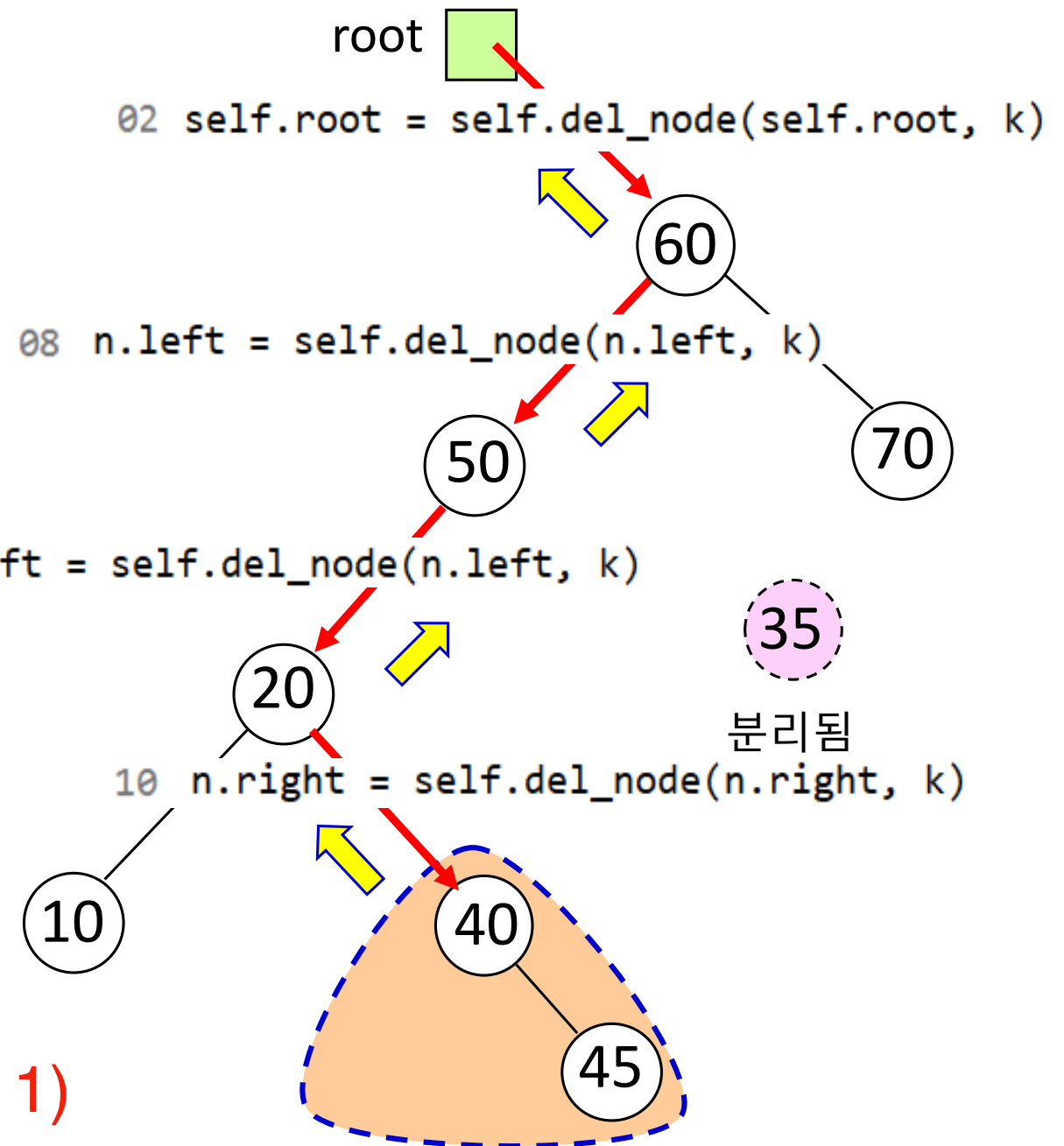


delete(45)가 수행되는 과정 (case 1)

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)
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

n의 왼쪽자식과 del_node()가 리턴하는 노드를 재 연결

n의 오른쪽자식과 del_node()가 리턴하는 노드를 재 연결

target은 삭제될 노드

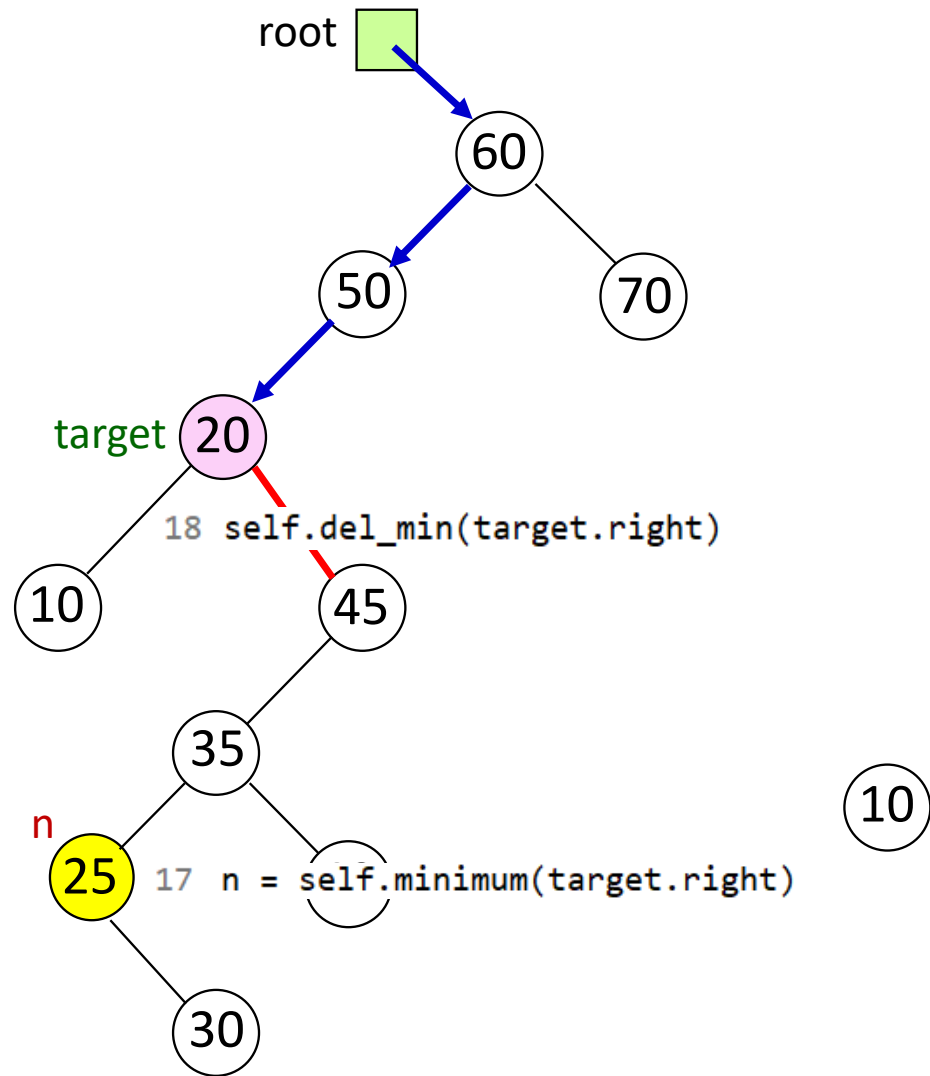
target의 중위 후속자 찾아 n이 참조하게 함

n의 오른쪽자식과 target의 오른쪽자식 연결

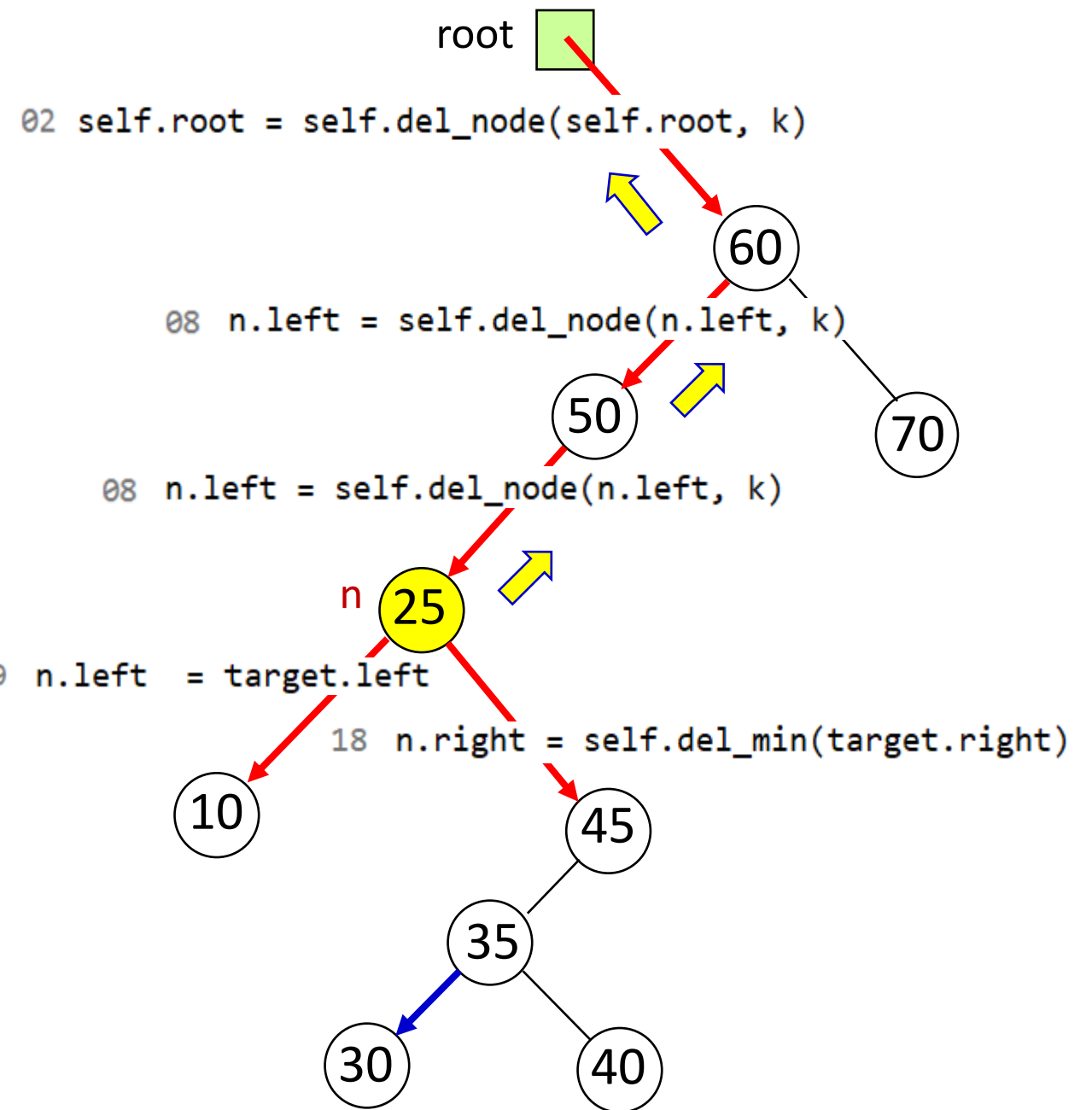
n의 왼쪽자식과 target의 왼쪽자식 연결

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)
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

n의 왼쪽자식과 del_node()가 리턴하는 노드를 재 연결

n의 오른쪽자식과 del_node()가 리턴하는 노드를 재 연결

target은 삭제될 노드

target의 중위 후속자 찾아 n이 참조하게 함

n의 오른쪽자식과 target의 오른쪽자식 연결

n의 왼쪽자식과 target의 왼쪽자식 연결