

제 5 장 탐색트리

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.
- 5장에서는 리스트 자료구조의 수행시간을 향상시키기 위한 트리 형태의 다양한 사전 자료구조들을 소개
 - 이진탐색트리, AVL트리, 2-3트리, 레드블랙트리, B-트리

5.1 이진탐색



이진탐색(Binary Search):

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)
```

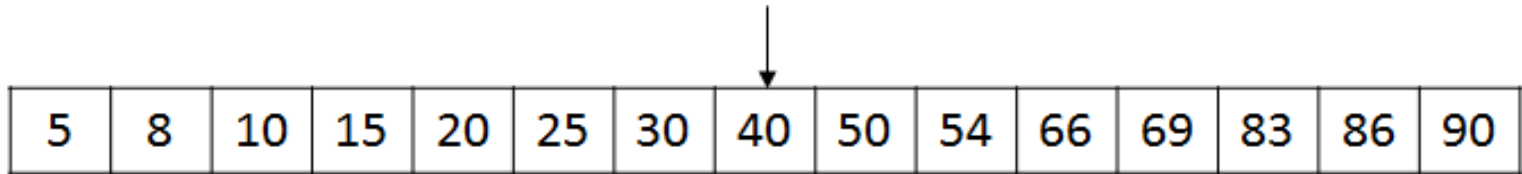
```
[2] mid = (left + right) // 2      # 중간 항목의 인덱스 계산
```

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

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

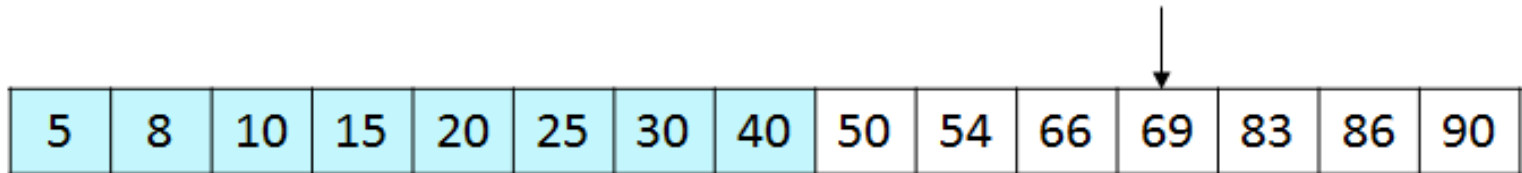
```
[5] else: binary_search(mid+1, right, t)       # 뒷부분 탐색
```

이진탐색으로 66을 찾는 과정



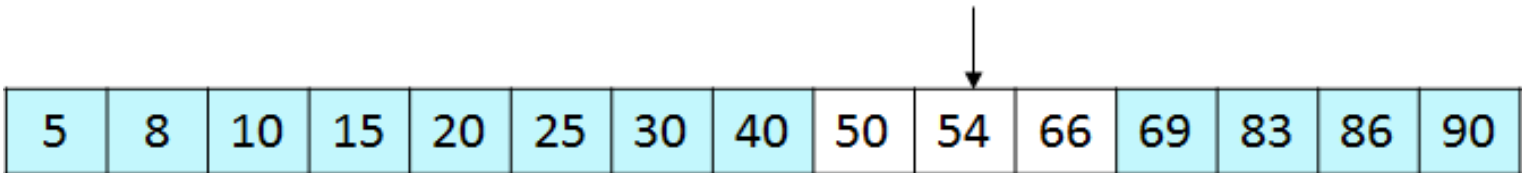
Initial array: 5, 8, 10, 15, 20, 25, 30, 40, 50, 54, 66, 69, 83, 86, 90. An arrow points to the element 40 at index 7.

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



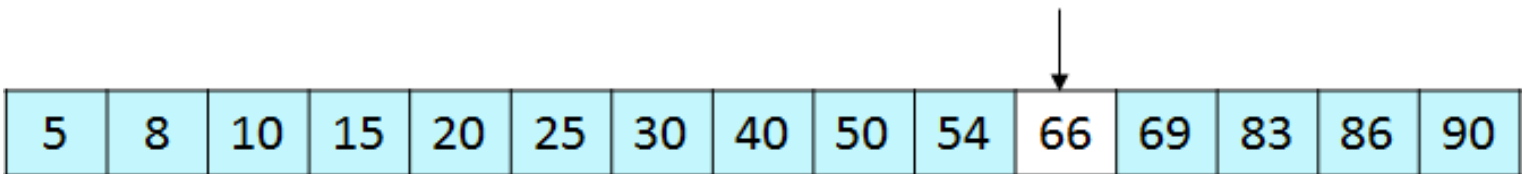
Array split: Elements less than 40 (5, 8, 10, 15, 20, 25, 30) are on the left, and elements greater than 40 (50, 54, 66, 69, 83, 86, 90) are on the right. An arrow points to the element 69 at index 11.

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



Array split: Elements less than 69 (5, 8, 10, 15, 20, 25, 30, 40, 50) are on the left, and elements greater than 69 (54, 66, 83, 86, 90) are on the right. An arrow points to the element 54 at index 9.

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



Array split: Elements less than 54 (5, 8, 10, 15, 20, 25, 30, 40, 50) are on the left, and elements greater than 54 (54, 66, 69, 83, 86, 90) are on the right. An arrow points to the element 66 at index 10.

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

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

$$T(1) = 1$$

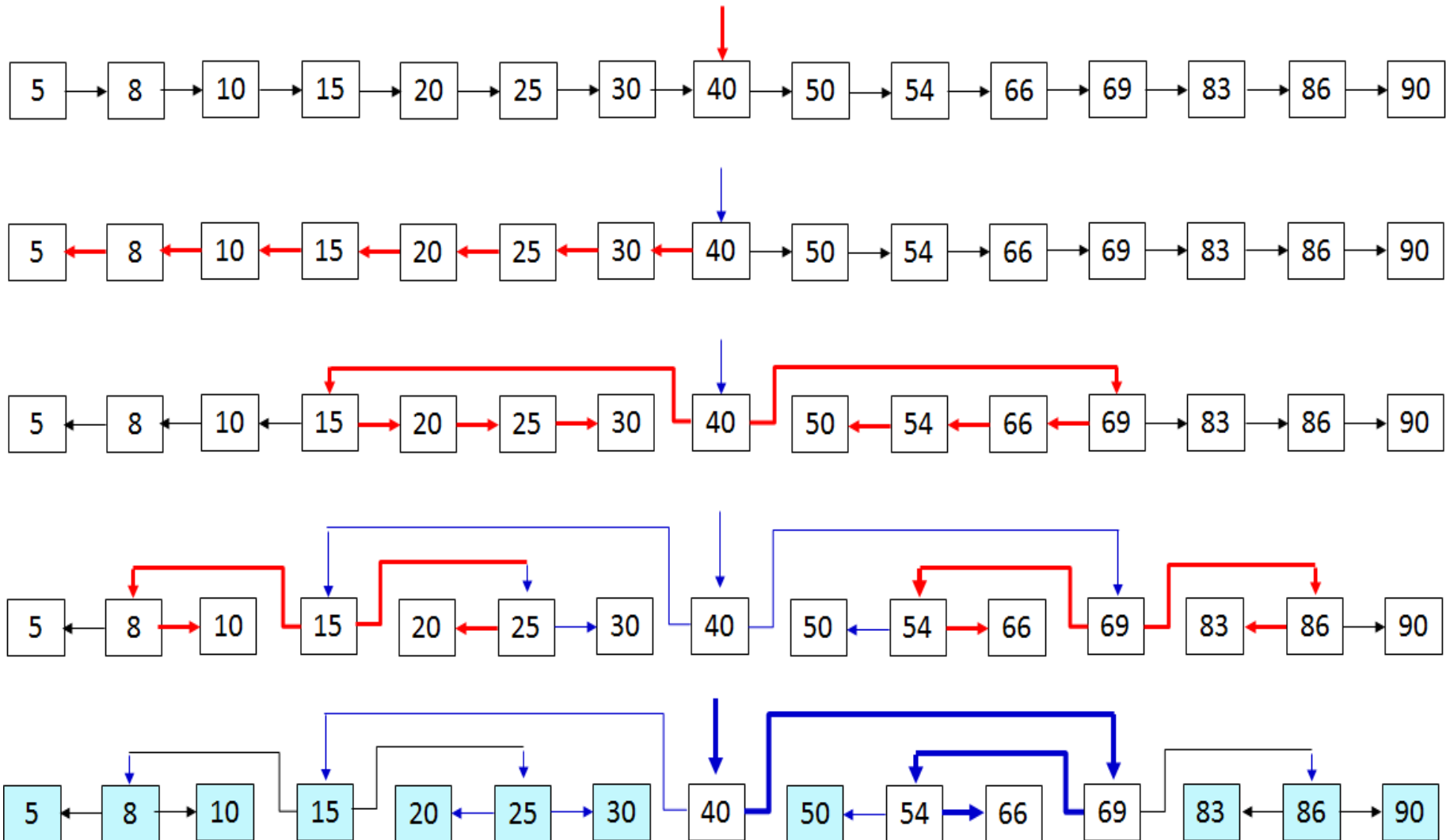
- $T(N) = T(N/2) + 1$
 $= [T((N/2)/2) + 1] + 1 = T(N/2^2) + 2$
 $= [T((N/2)/2^2) + 1] + 2 = T(N/2^3) + 3$
 $= \dots = T(N/2^k) + k$
 $= T(1) + k, \text{ if } N = 2^k, k = \log_2 N$
 $= 1 + \log_2 N = O(\log N)$

5.2 이진탐색트리

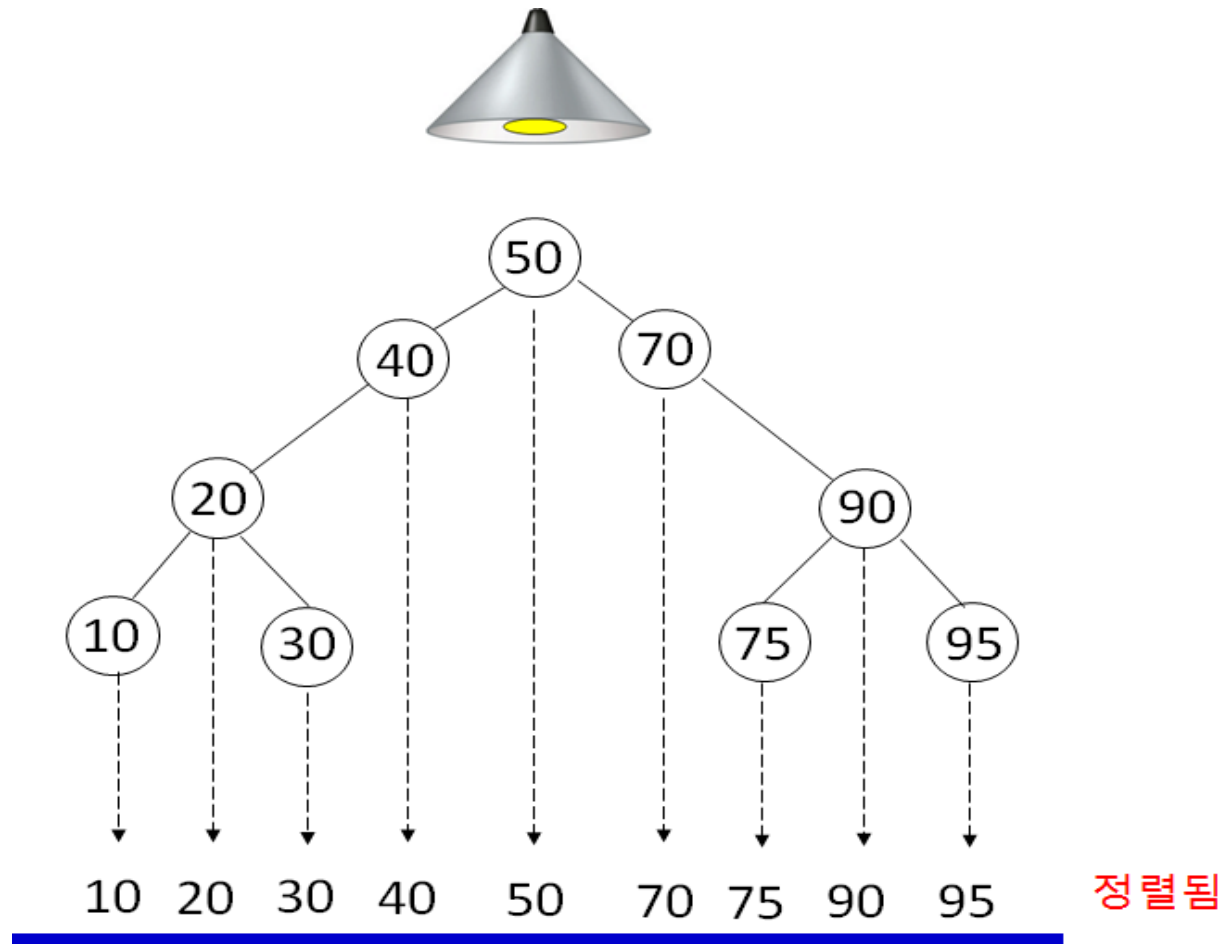
- 이진탐색트리(Binary Search Tree):

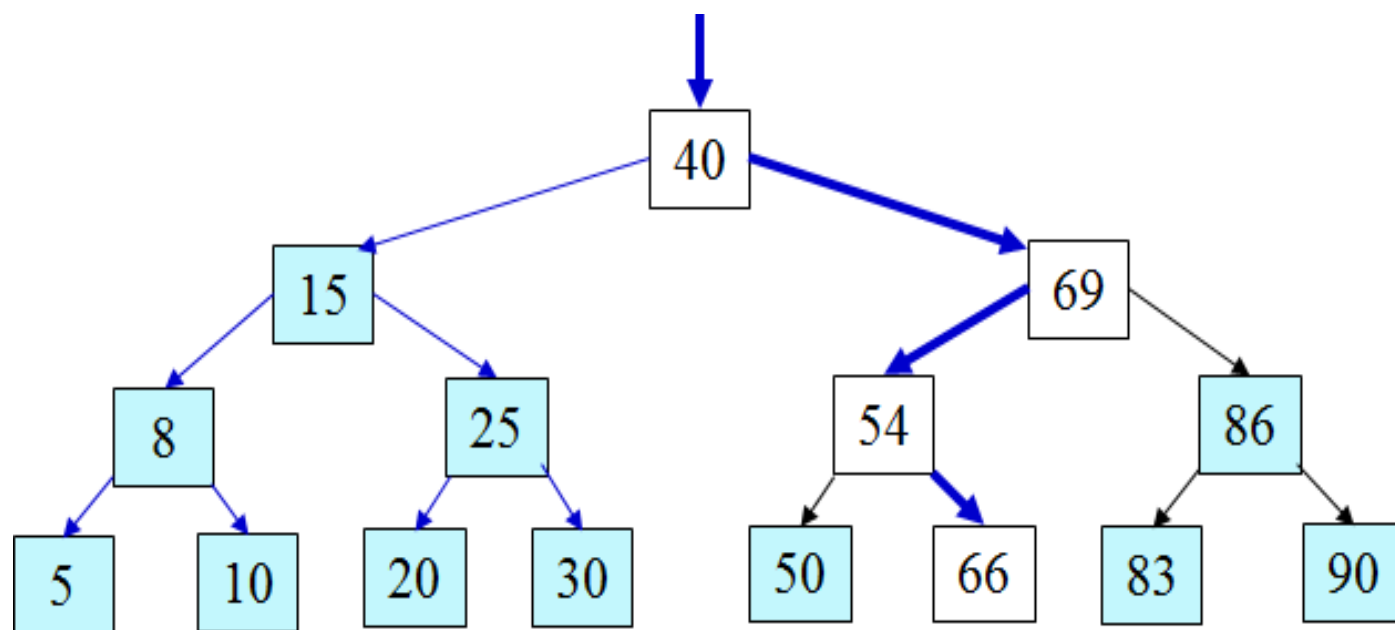
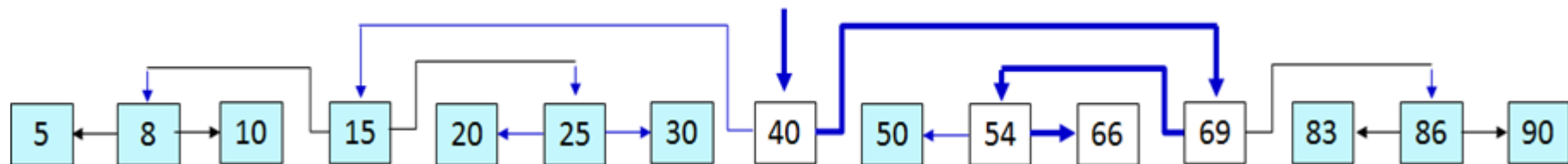
이진탐색(Binary Search)의 개념을 트리 형태의 구조에 접목한 자료구조

- 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

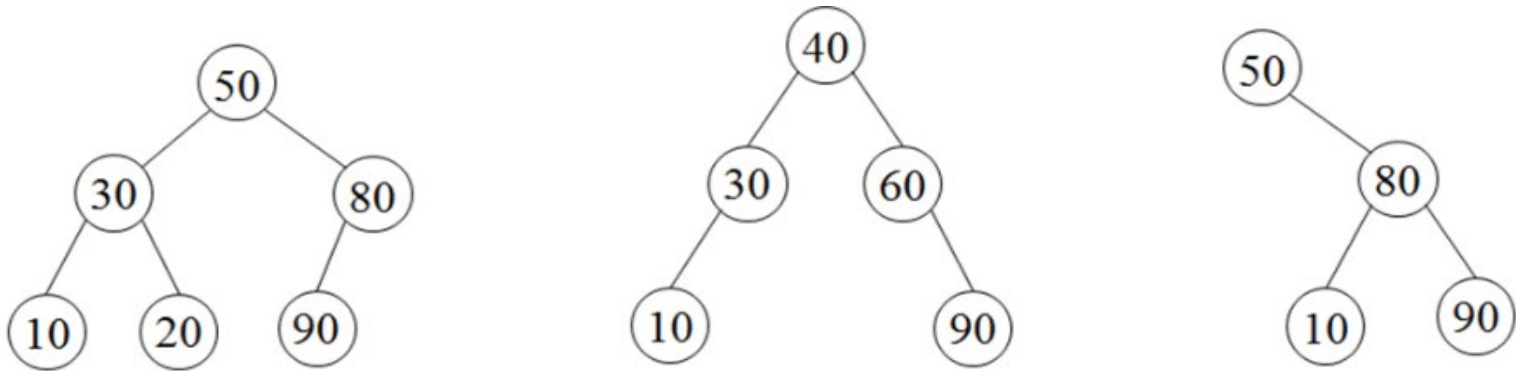






이진탐색트리는 이진트리로서 각 노드가 다음과 같은 조건을 만족한다.

- 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

5.2.1 탐색 연산

- 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

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

```
def get_item(self, n, k):
```

탐색 실패

```
    if n == None:
        return None
```

k가 노드의 key보다 작으면
왼쪽 서브트리 탐색

```
    if n.key > k:
        return self.get_item(n.left, k)
```

```
    elif n.key < k:
        return self.get_item(n.right, k)
```

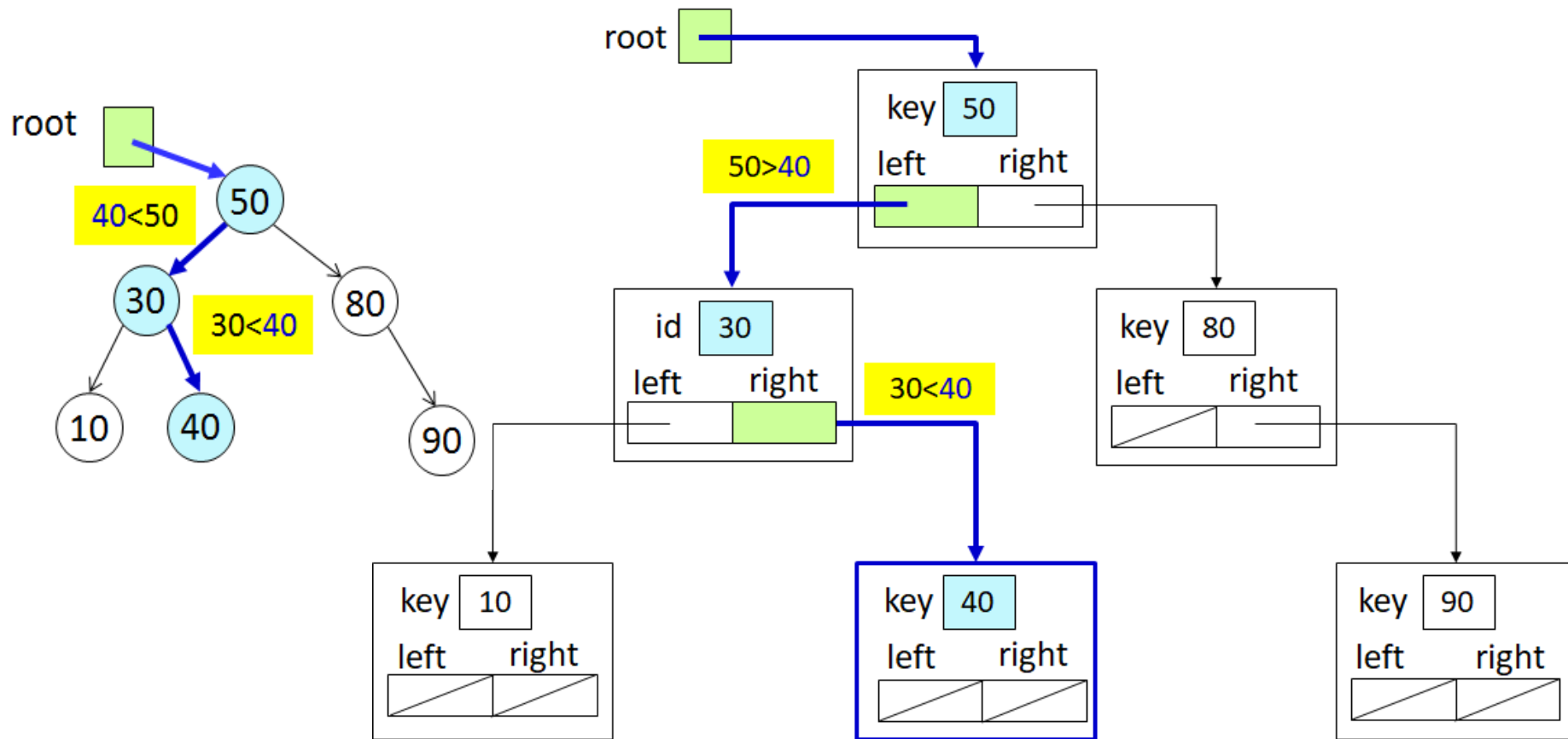
k가 노드의 key보다 크면
오른쪽 서브트리 탐색

```
    else:
```

```
        return n.value
```

탐색 성공

[예제] 40을 탐색하는 과정




5.1.2 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  루트와 put_item()이
리턴하는 노드를 재 연결

04 def put_item(self, n, key, value):

05 if n == None:

06 return Node(key, value)  새 노드 생성

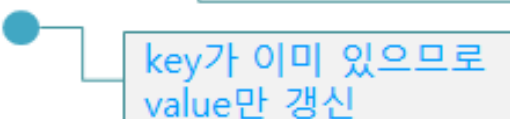
07 if n.key > key:  n의 왼쪽자식과 put_item()이
리턴하는 노드를 재 연결

08 n.left = self.put_item(n.left, key, value)

09 elif n.key < key:

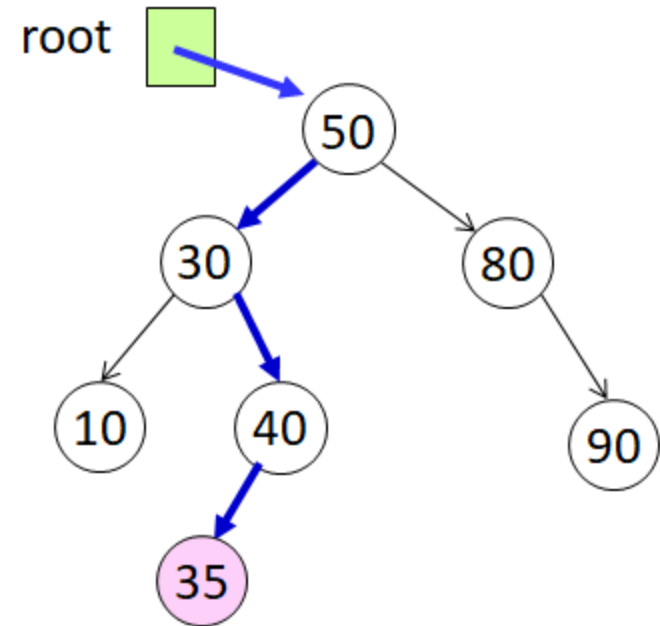
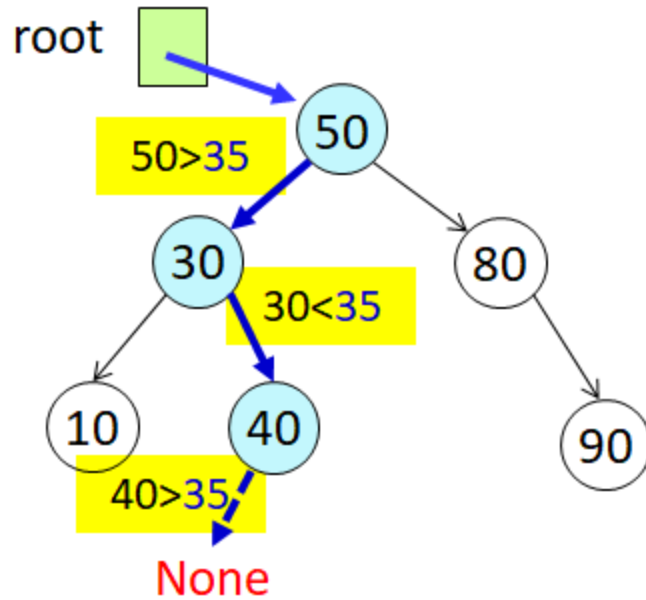
10 n.right = self.put_item(n.right, key, value)

11 else:  n의 오른쪽자식과 put_item()이
리턴하는 노드를 재 연결

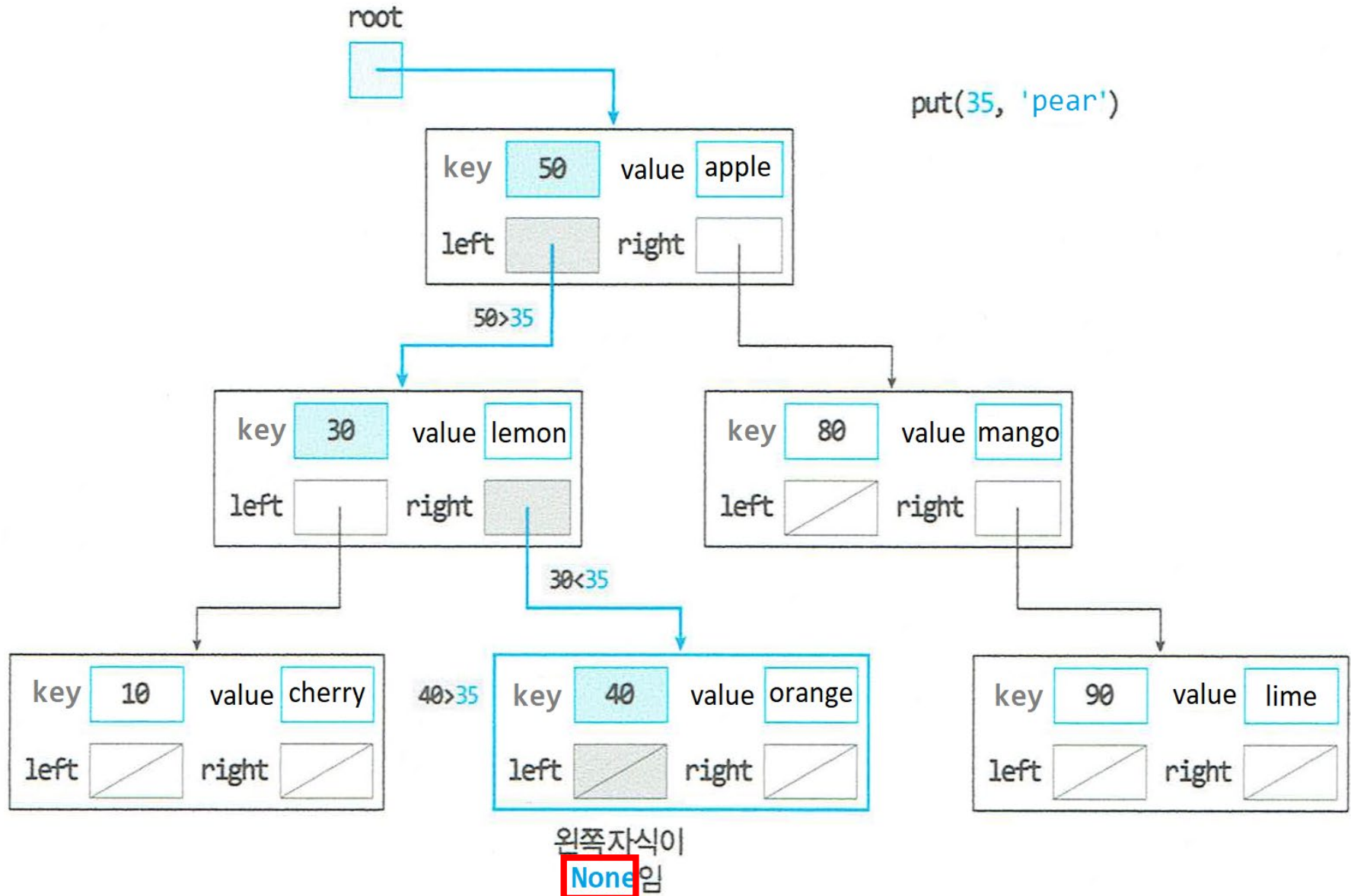
12 n.vlaue = value  key가 이미 있으므로
value만 갱신

13 return n  부모노드와 연결하기 위해
노드 n을 리턴

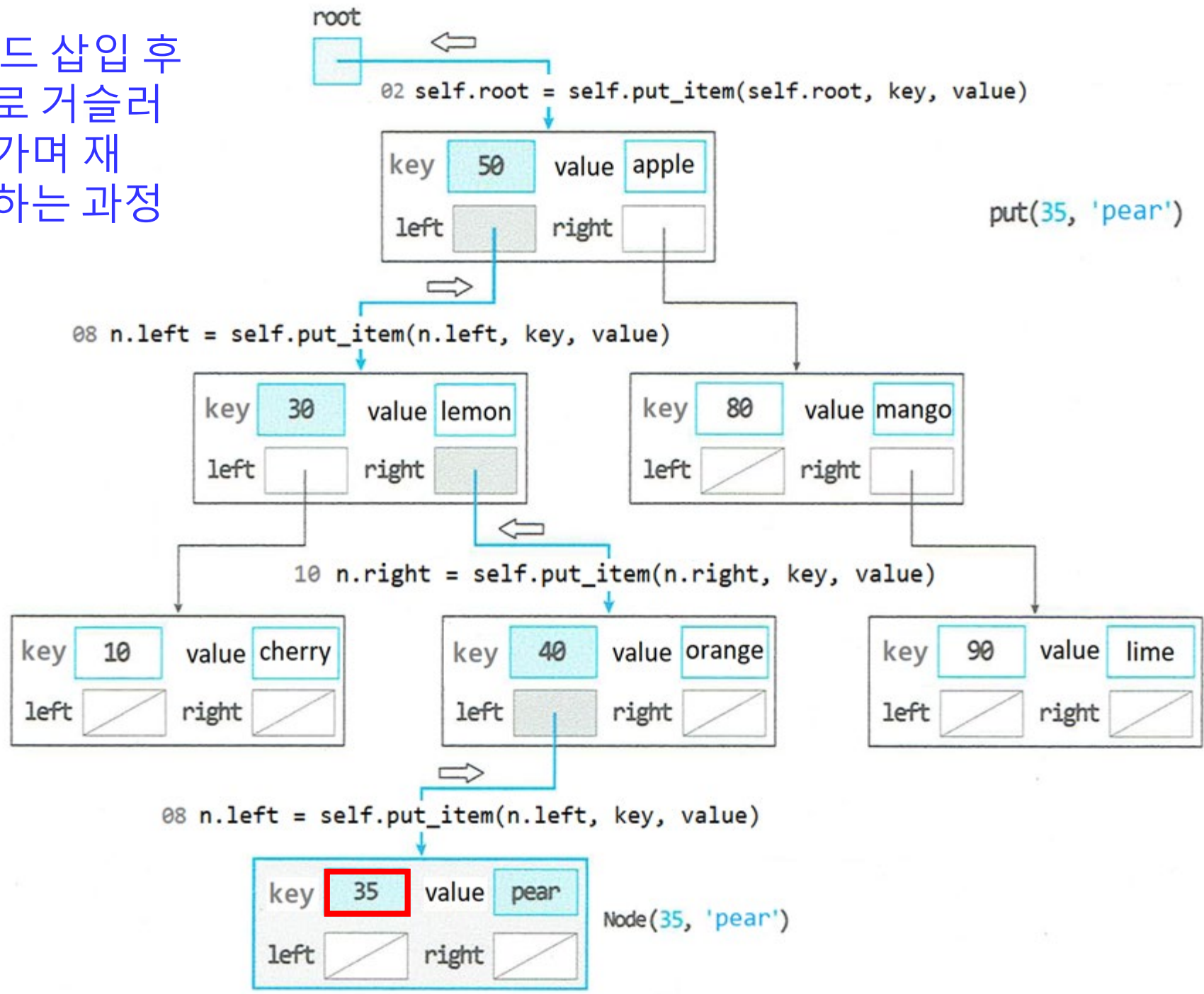
[예제] 35를 삽입하는 과정



35를 삽입할 장소를 탐색하는 과정



새 노드 삽입 후
루트로 거슬러
올라가며 재
연결하는 과정



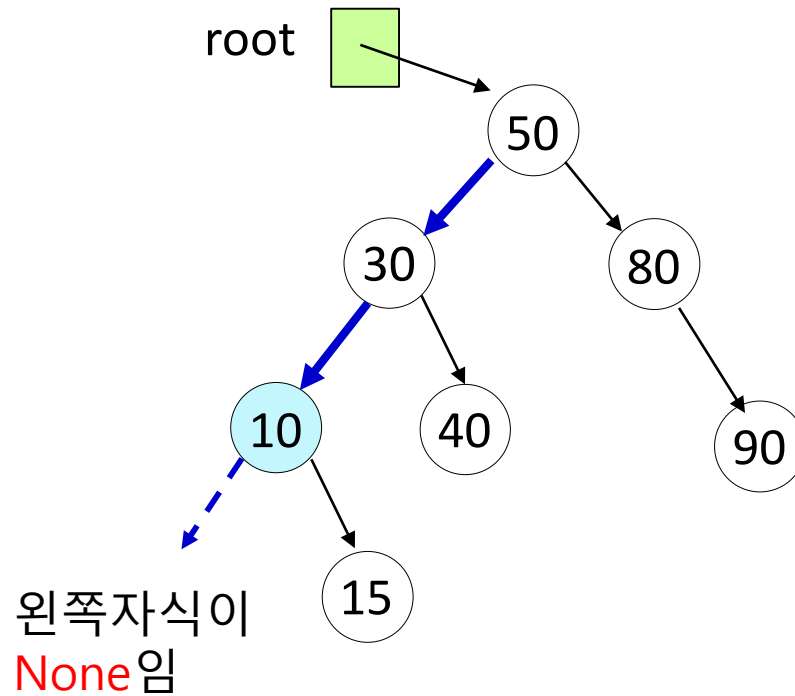
5.2.3 최솟값 찾기

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

5.2.4 최소값 삭제 연산

- 최소값을 가진 노드를 삭제하는 것은 최소값을 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
```

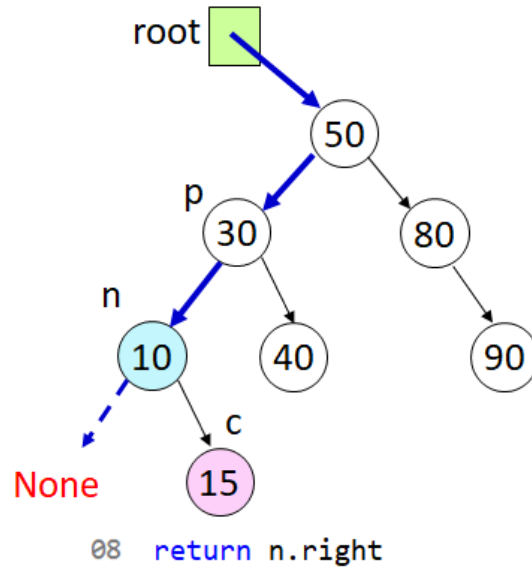
```
09     n.left = self.del_min(n.left)
```

```
10     return n
```

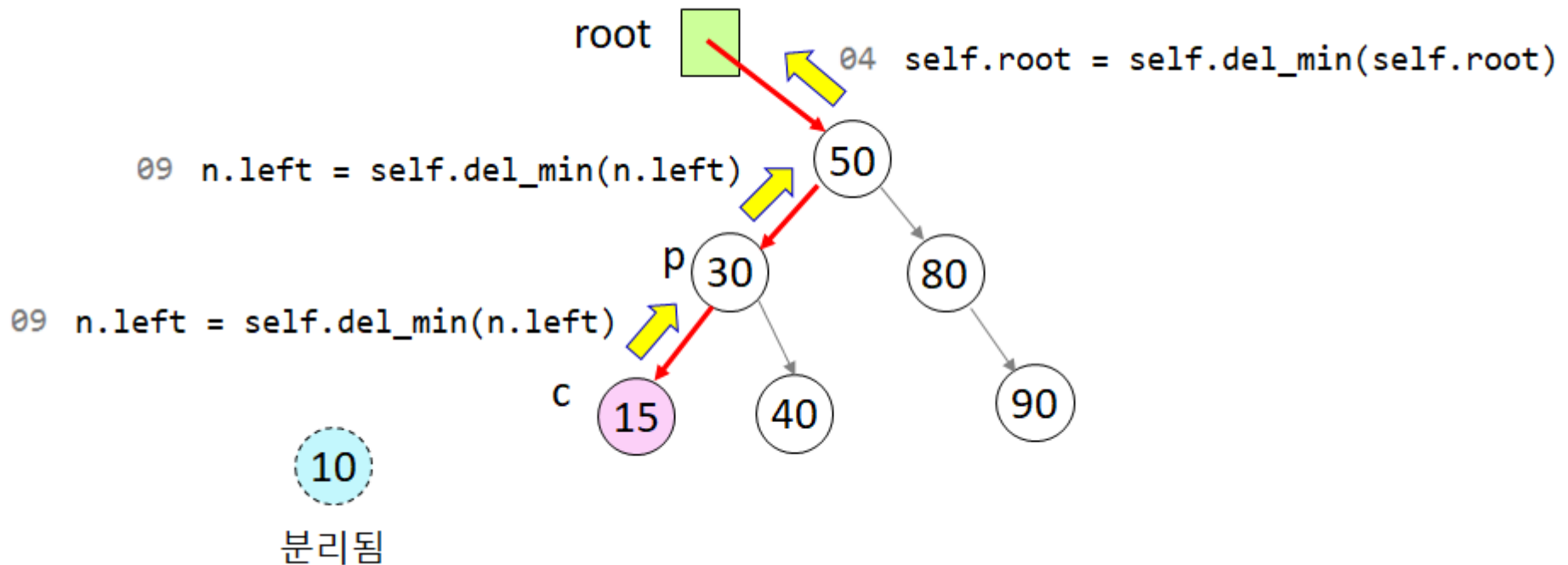
루트와 del_min()이 리턴
하는 노드를 재 연결

최솟값 가진 노드의 오른쪽
자식을 리턴

n의 왼쪽자식과 del_min()이
리턴하는 노드를 재 연결

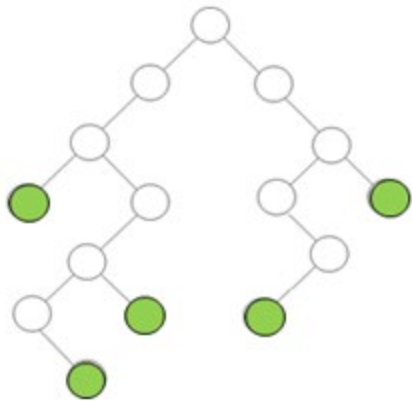


[그림 5-10] delete_min()의 수행 과정

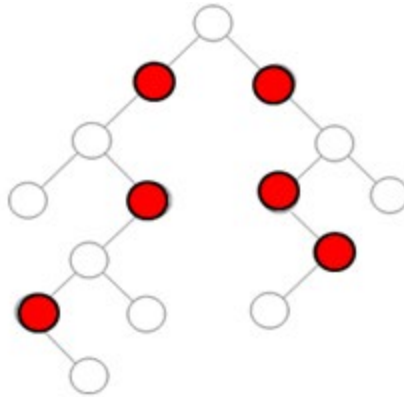


5.2.5 삭제 연산

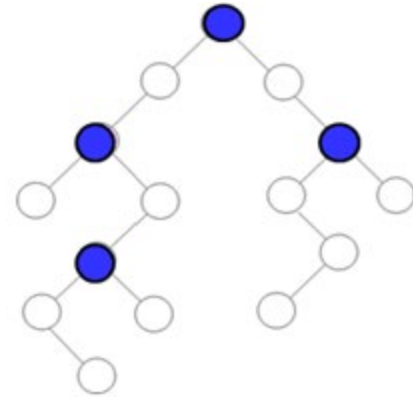
- 우선 삭제하고자 하는 노드를 찾은 후 이진탐색트리 조건을 만족하도록 삭제된 노드의 부모와 자식(들)을 연결해 주어야
- 삭제되는 노드가 자식이 없는 경우(case 0), 자식이 하나인 경우(case 1), 자식이 둘인 경우(case 2)로 나누어 delete 연산을 수행



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

```

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

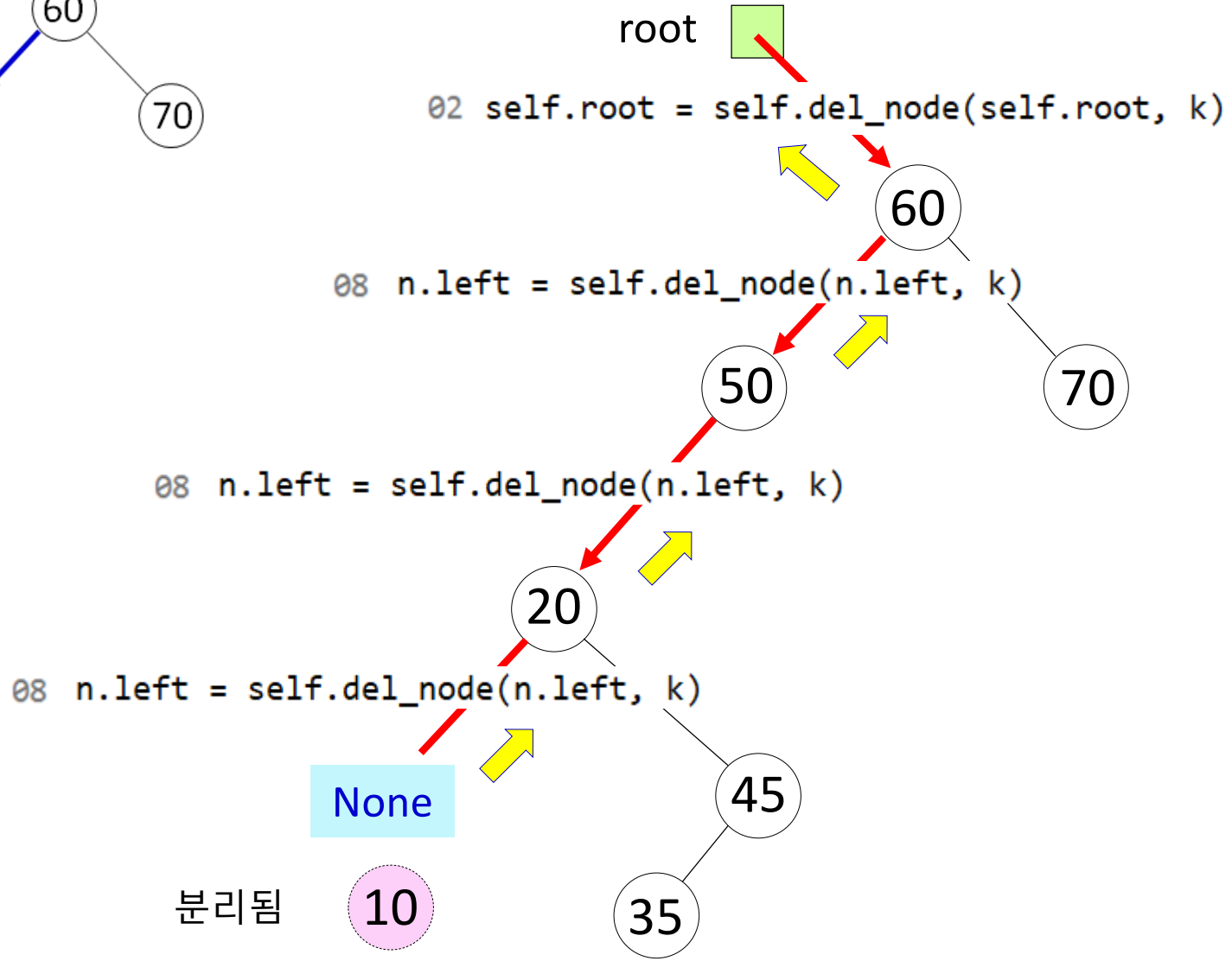
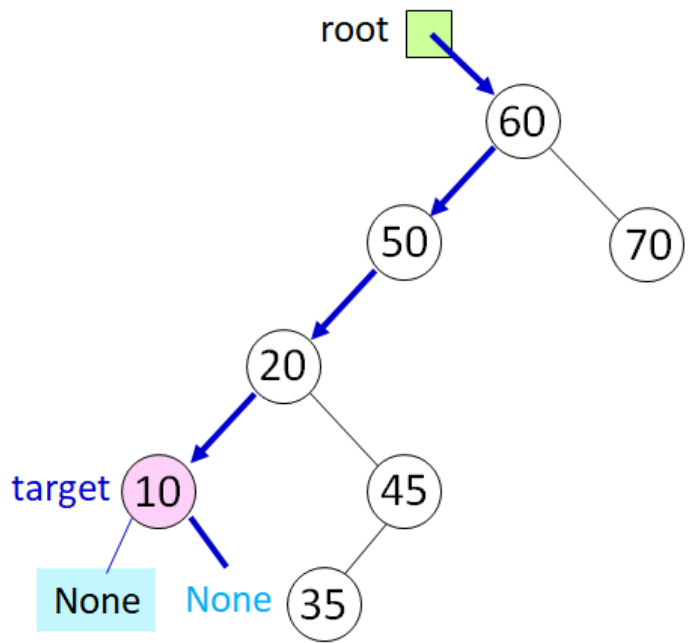
n의 오른쪽자식과 del_node()가 리턴하는 노드를 재 연결

target은 삭제될 노드

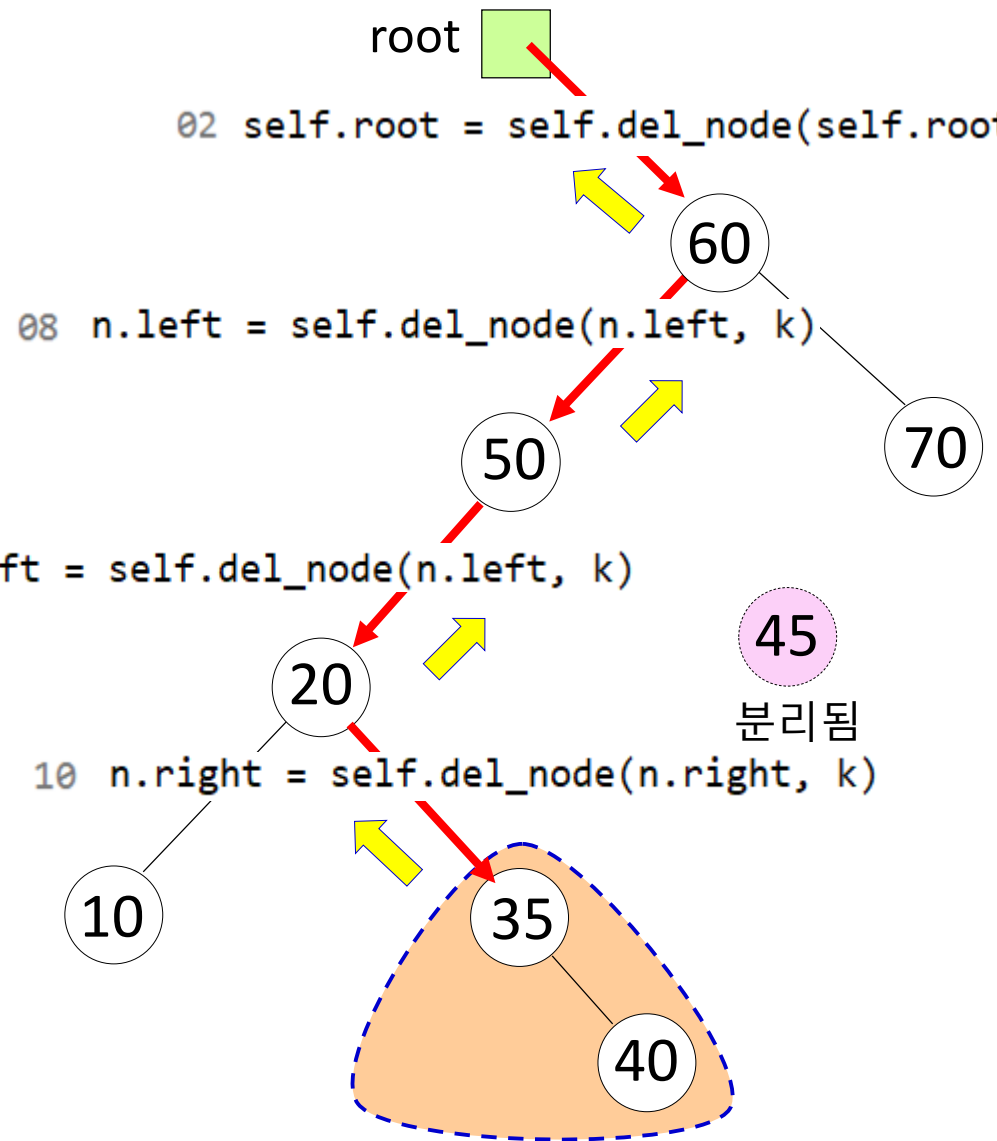
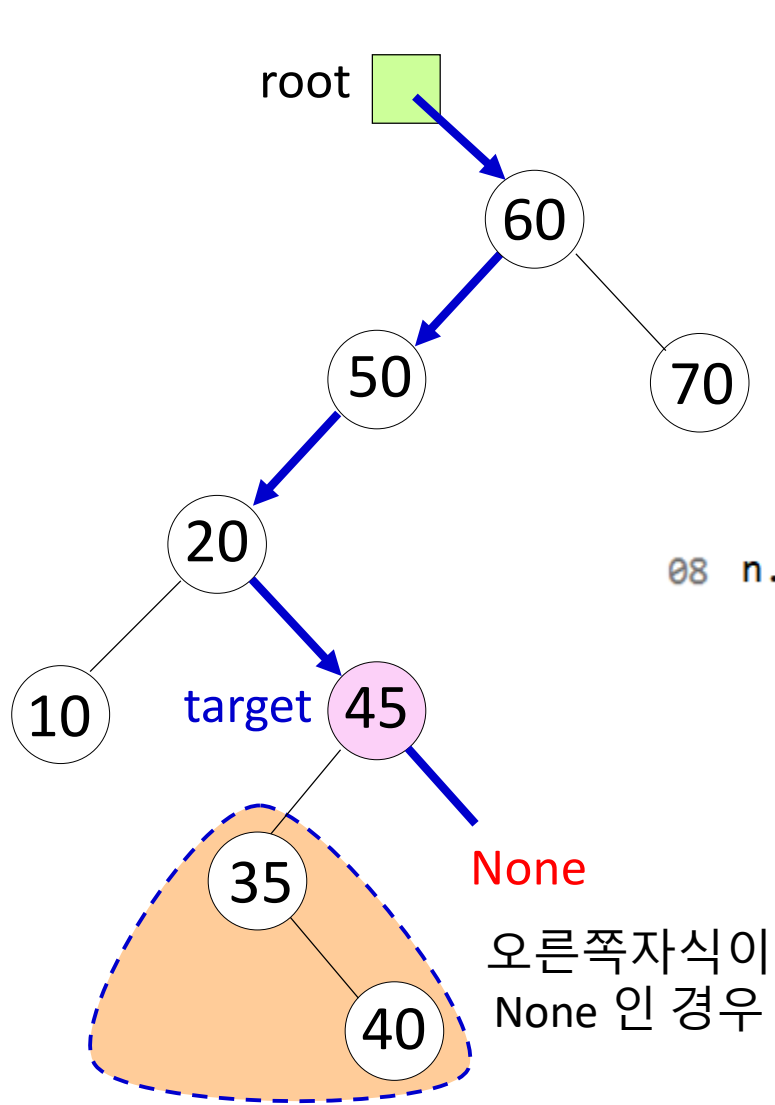
target의 중위 후속자 찾아 n이 참조하게 함

n의 오른쪽자식과 target의 오른쪽자식 연결

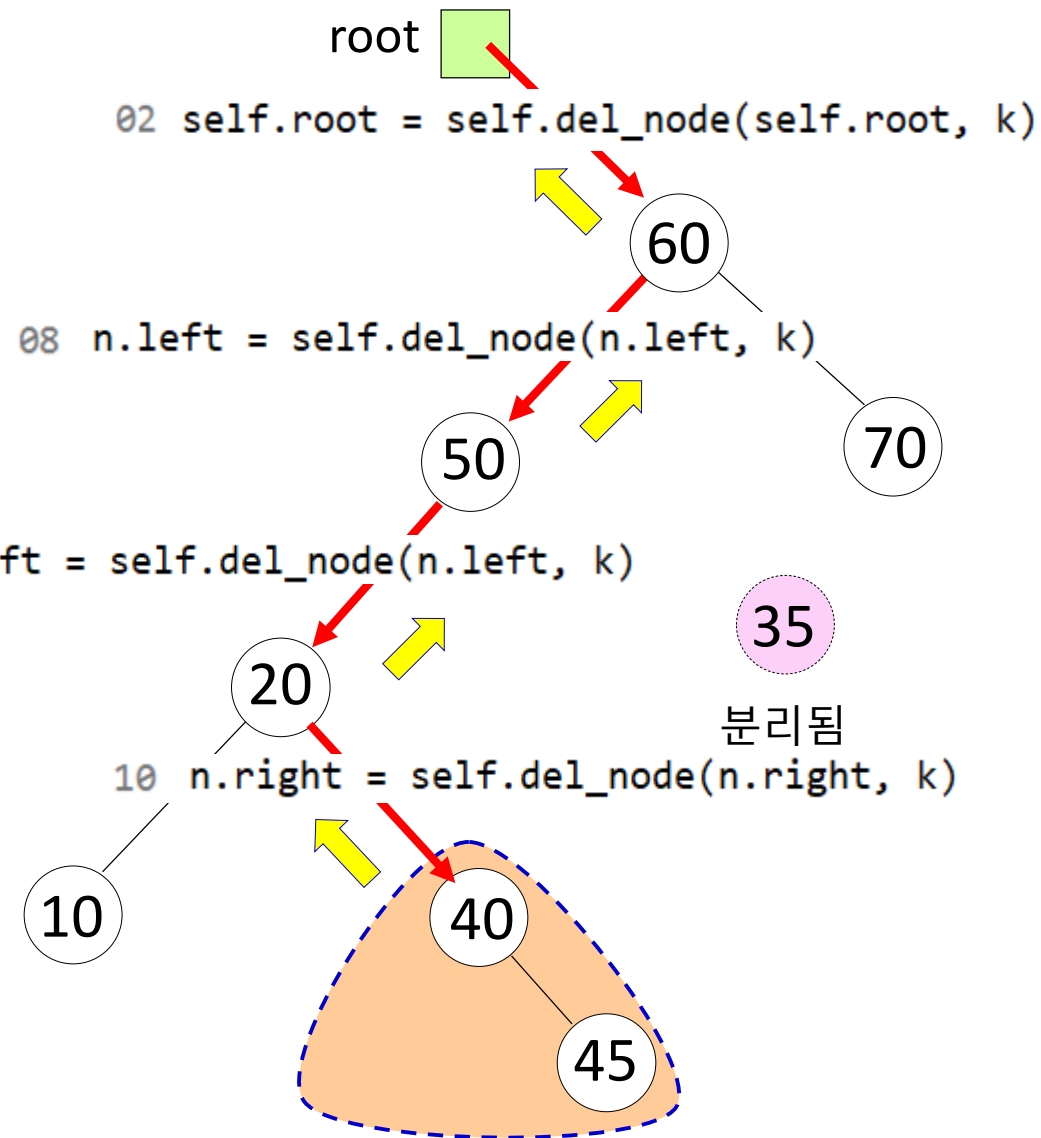
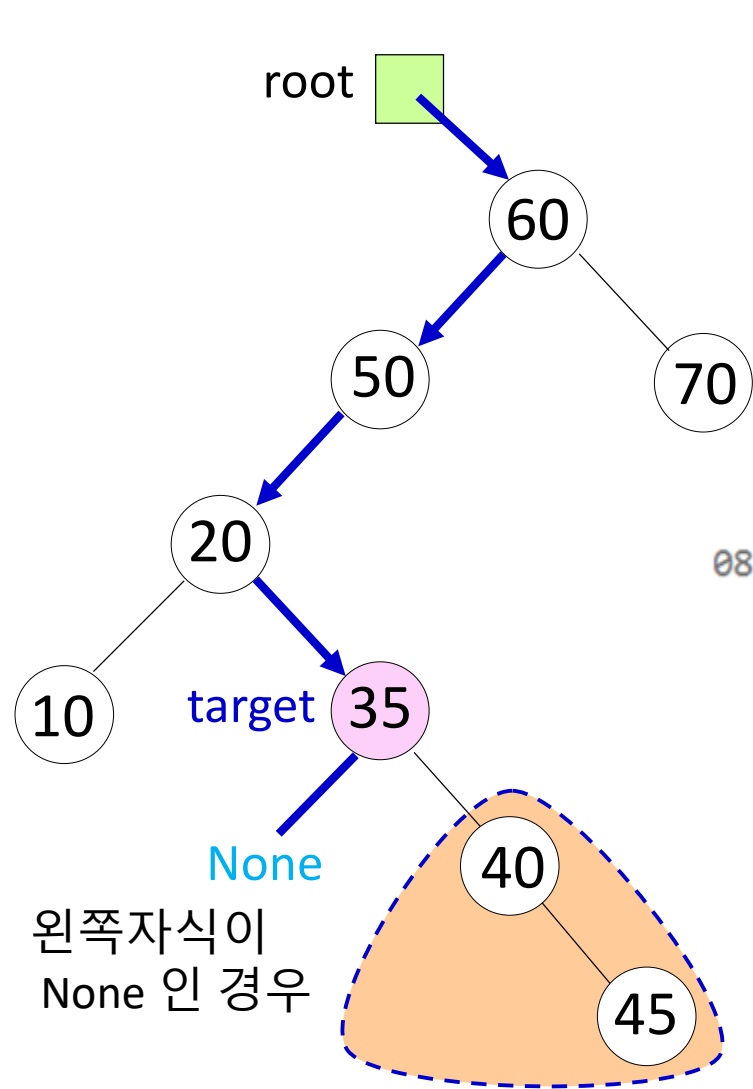
n의 왼쪽자식과 target의 왼쪽자식 연결



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

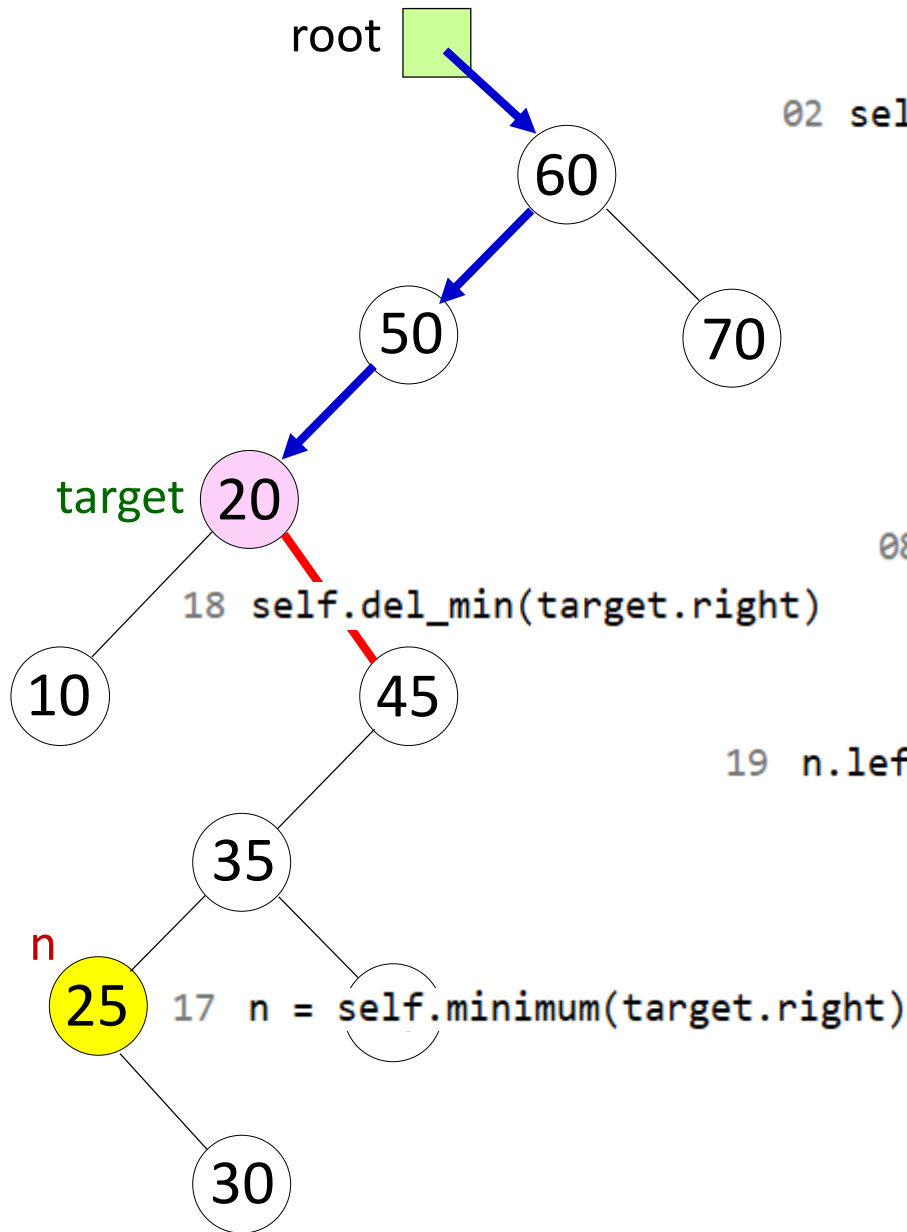


[그림 5-13] delete(45)가 수행되는 과정 (case 1)



delete(35)가 수행되는 과정 (case 1)

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



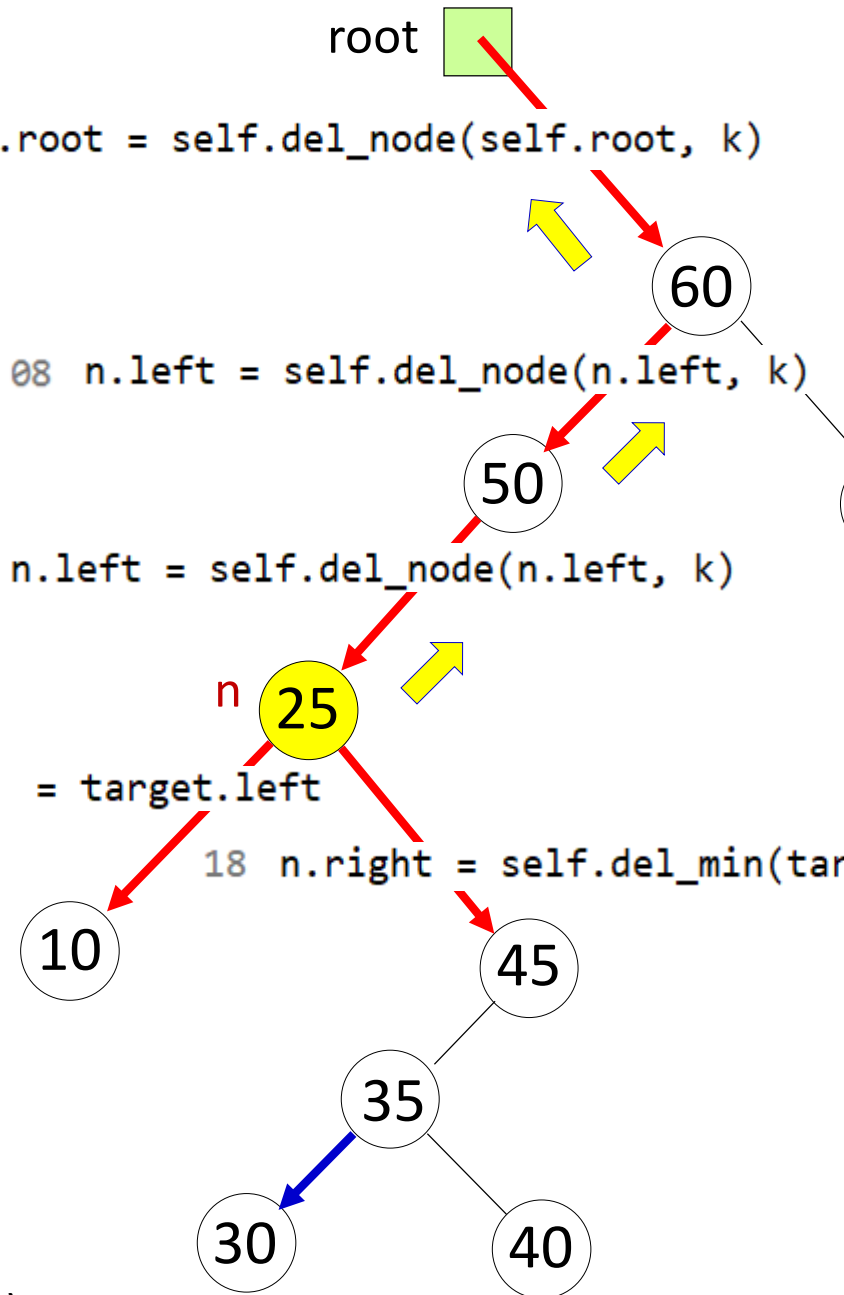
```
02 self.root = self.del_node(self.root, k)
```

```
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(tar
```



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

[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')

```

이진탐색트리
객체 생성

1
2
개의
노드
삽입

```

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)

```

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

- 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, $O(h)$

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

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

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

5.3 AVL트리

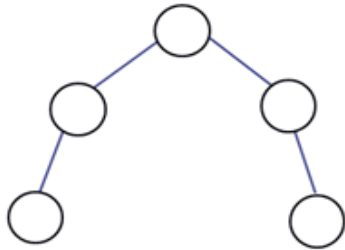
- AVL 트리는 트리가 한쪽으로 치우쳐 자라나는 현상을 방지하여 트리 높이의 균형(Balance)을 유지하는 이진탐색트리
- 균형(Balanced) 이진트리를 만들면 N 개의 노드를 가진 트리의 높이가 $O(\log N)$ 이 되어 탐색, 삽입, 삭제 연산의 수행시간이 $O(\log N)$ 으로 보장

[핵심 아이디어]

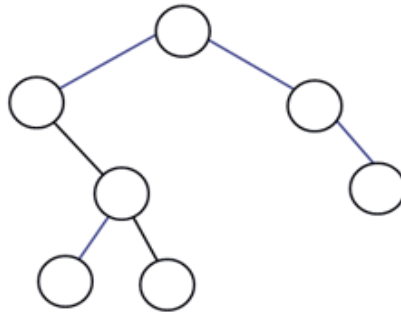
AVL트리는 삽입이나 삭제로 인해 균형이 깨지면 회전 연산을 통해 트리의 균형을 유지한다.



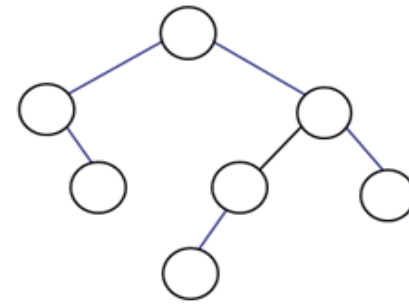
AVL트리는 임의의 노드 x 에 대해 x 의 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 차이가 1을 넘지 않는 이진탐색트리이다.



(a)



(b)



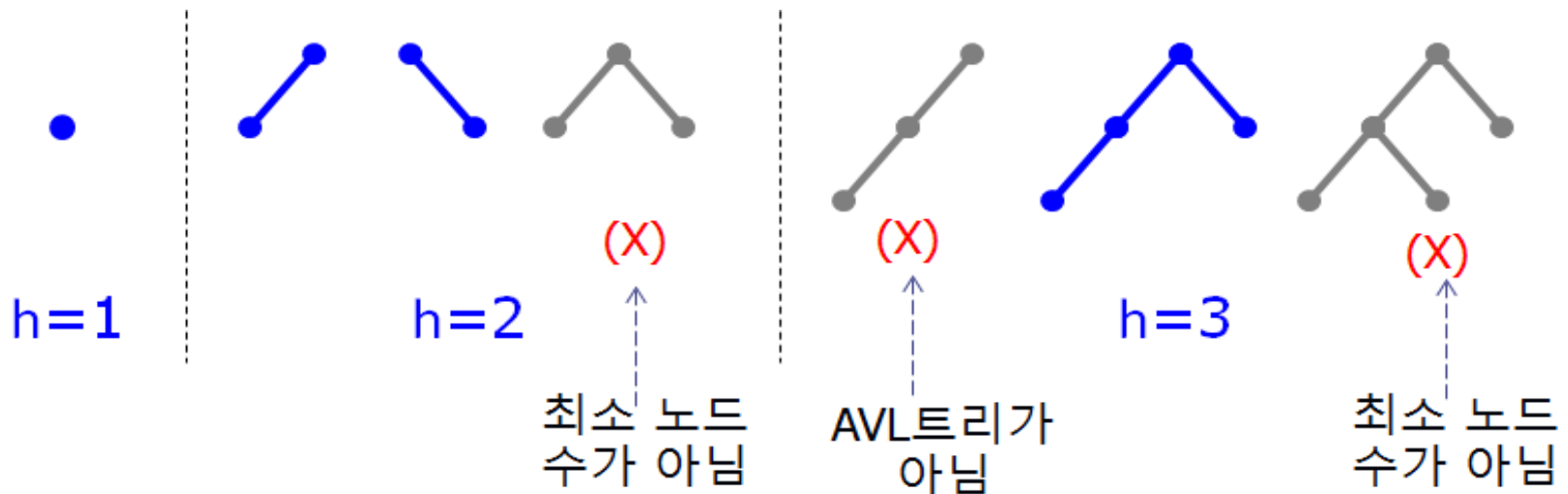
(c)

어느 트리가 AVL트리 형태를 갖추고 있나?

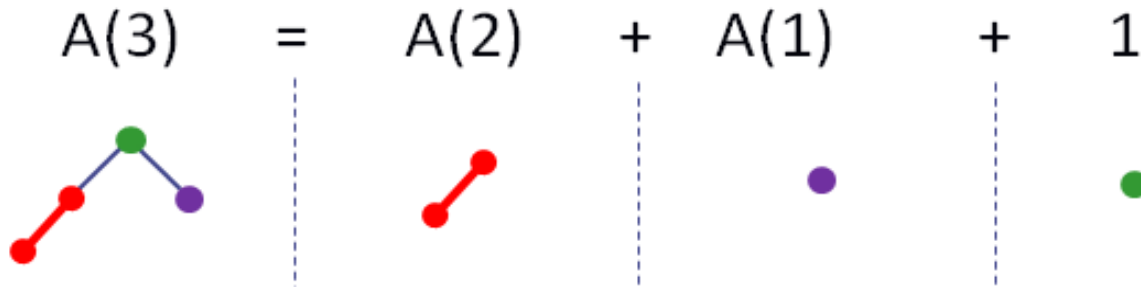
[정리] N 개의 노드를 가진 AVL 트리의 높이는 $O(\log N)$ 이다.

[증명] $A(h)$ = 높이가 h 인 AVL 트리를 구성하는 최소의 노드 수

$A(1) = 1, A(2) = 2, A(3) = 4$ 이다.



A(3)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

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

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

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

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

$A(h)$ = 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.

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

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

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

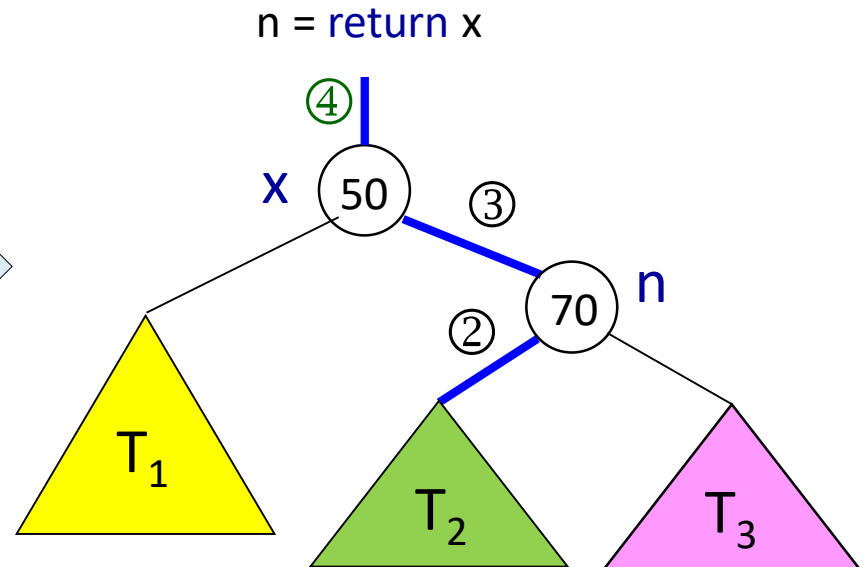
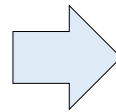
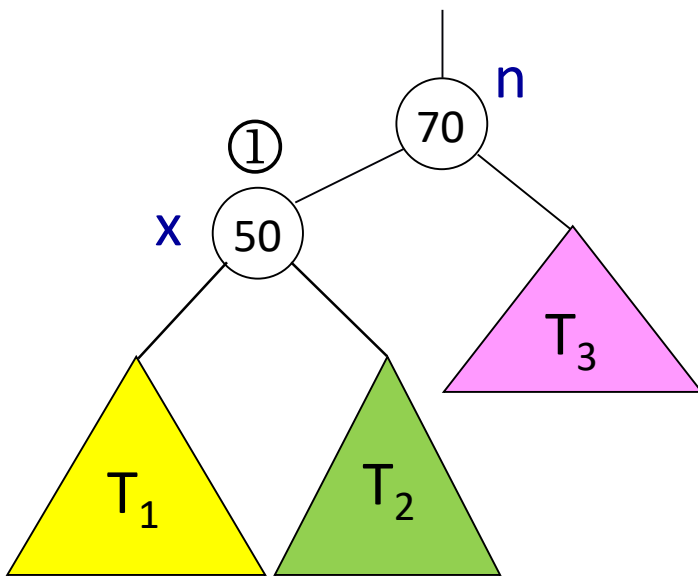
5.3.1 AVL 트리의 회전 연산

- 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()`: 왼쪽 방향의 서브트리가 높아서 불균형이 발생할 때 서브트리를 오른쪽 방향으로 회전
- 노드 n 의 왼쪽 자식 x 를 노드 n 의 자리로 옮기고, 노드 n 을 노드 x 의 오른쪽 자식으로 만들며, 이 과정에서 서브트리 τ_2 가 노드 n 의 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

```

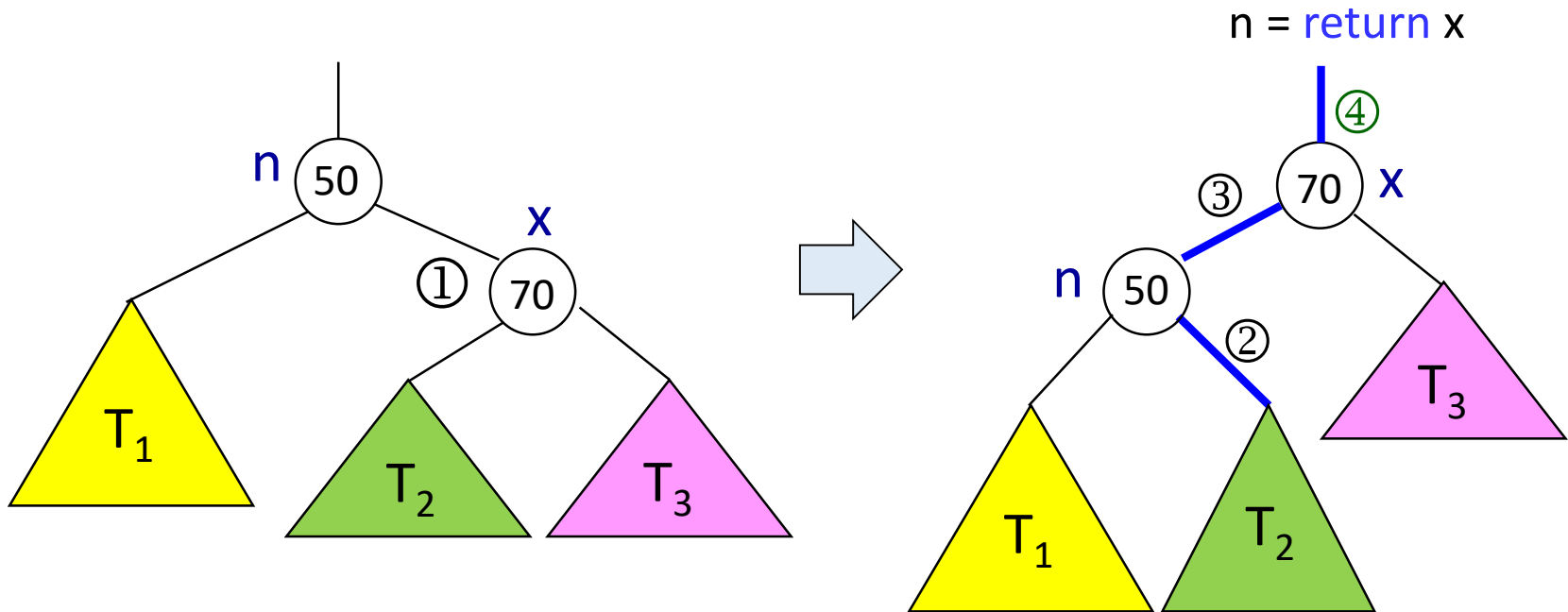


[그림 5-20] rotate_right

```

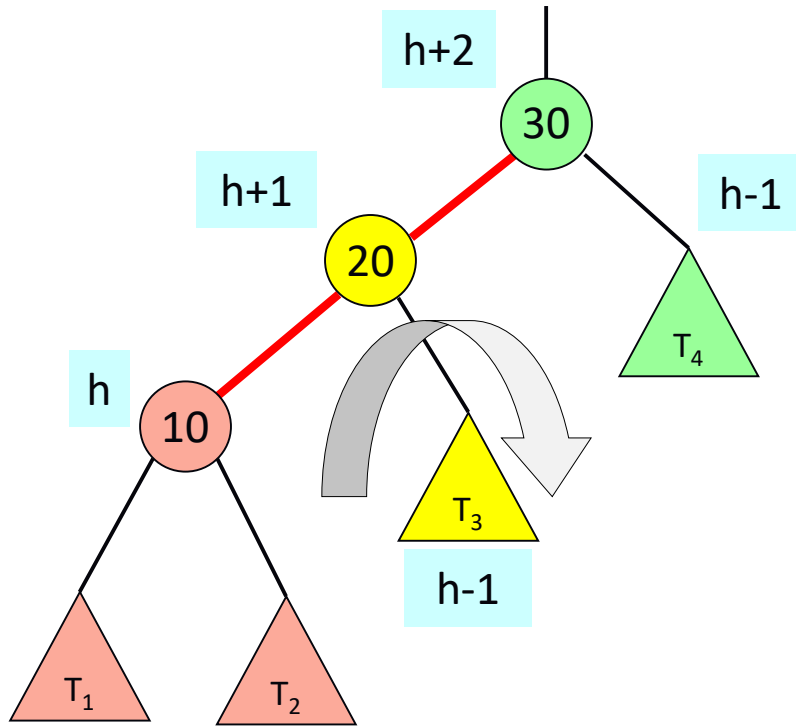
01 def rotate_left(self, n): # 좌로 회전
02     ① x = n.right
03     ② n.right = x.left
04     ③ x.left = n
05     n.height = max(self.height(n.left), self.height(n.right)) + 1
06     x.height = max(self.height(x.left), self.height(x.right)) + 1
07     ④ return x

```



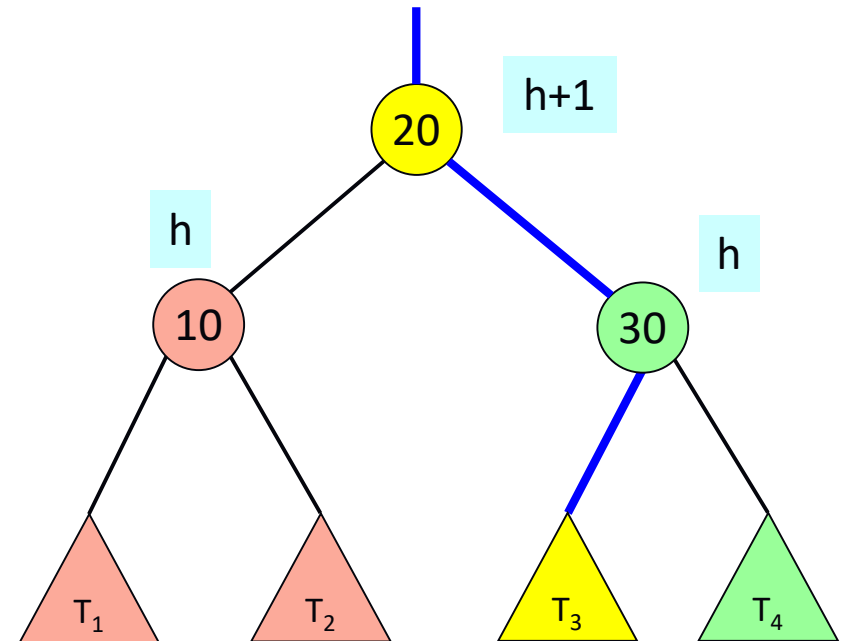
[그림 5-21] rotate_left

LL-회전



$$\max(T_1, T_2) = h-1$$

(a) T_1 또는 T_2 에 새 노드 삽입

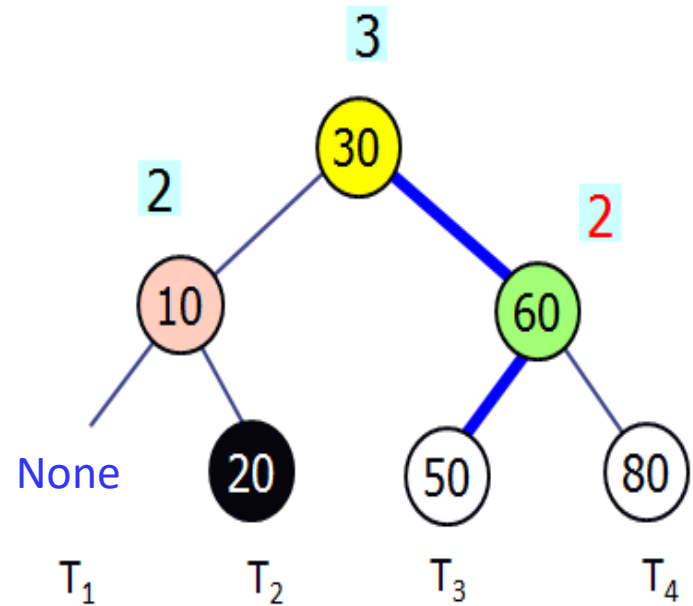
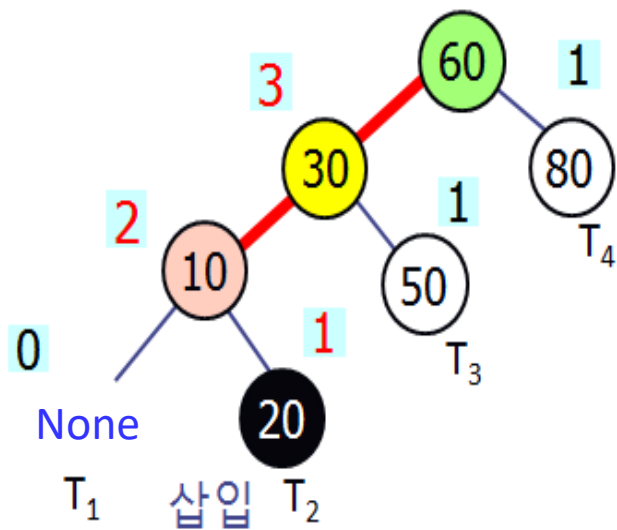


(b) LL-회전 후

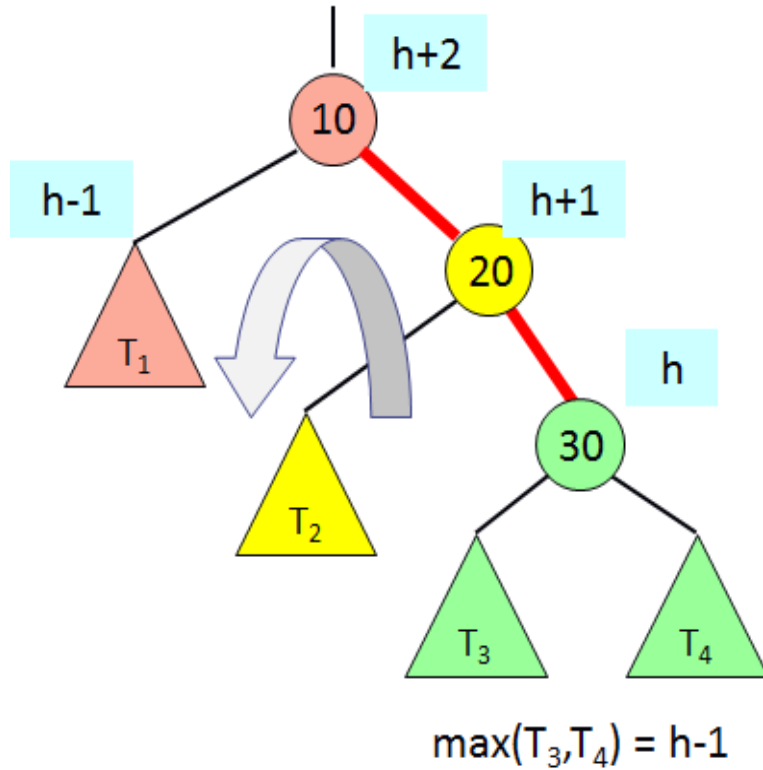
LL-회전

- (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) 서브트리**에 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
- LL-회전은 **rotate_right()**를 사용

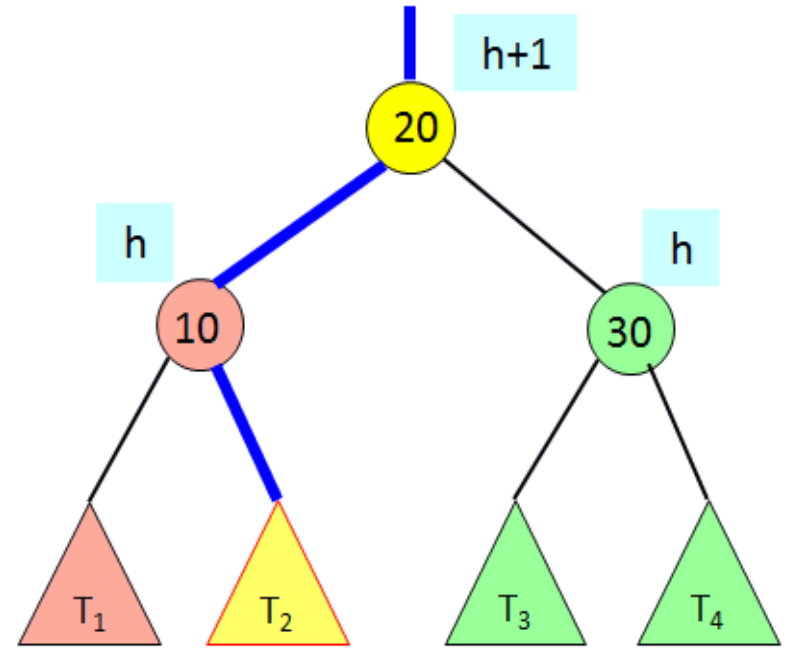
[예제] LL-회전의 예



RR-회전



(a) T_3 또는 T_4 에 새 노드 삽입

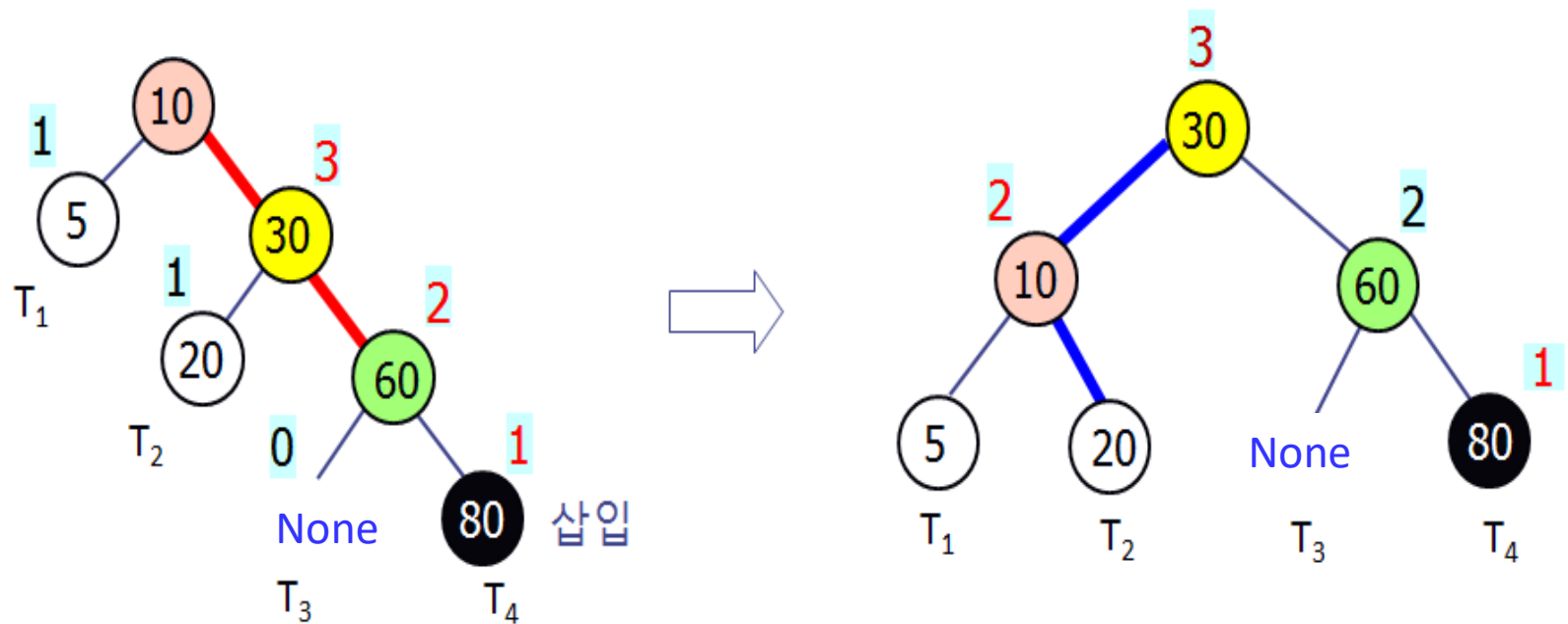


(b) RR-회전 후

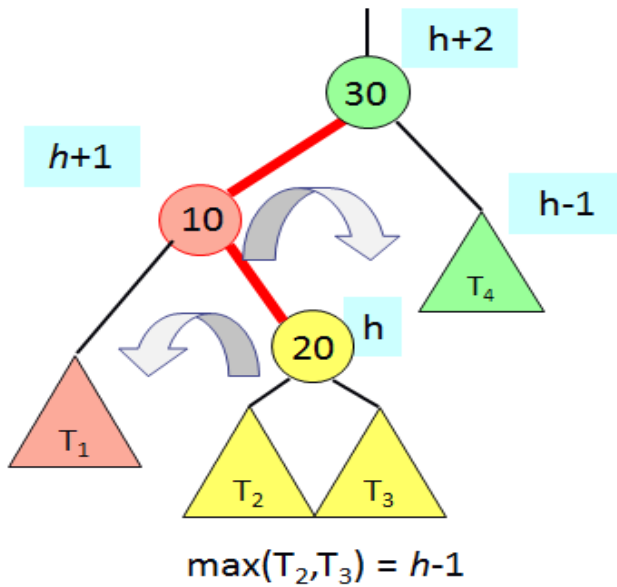
RR-회전

- (a) 30의 왼쪽 서브트리(T_3) 또는 오른쪽 서브트리(T_4)에 새로운 노드 삽입
 - T_3 또는 T_4 의 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
 - T_2 는 10의 오른쪽 자식으로
 - T_2 에 있는 키들은 10과 20 사이 값을 가지므로 T_2 의 이동 전후 모두 이진탐색트리 조건이 만족
- RR-회전은 `rotate_left()` 사용

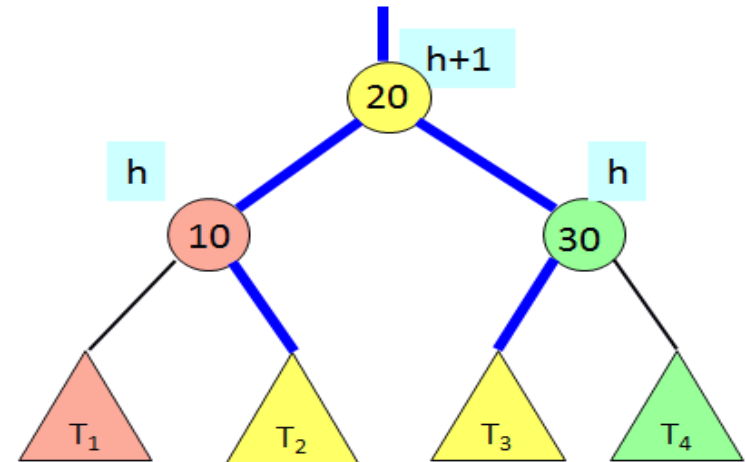
[예제] RR-회전의 예



LR-회전



(a) T_2 또는 T_3 에 새 노드 삽입



(b) LR-회전 후

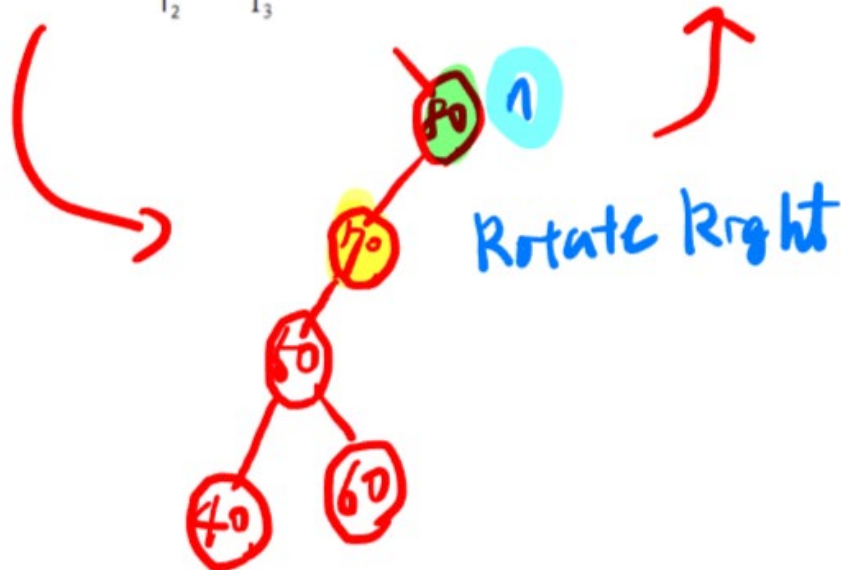
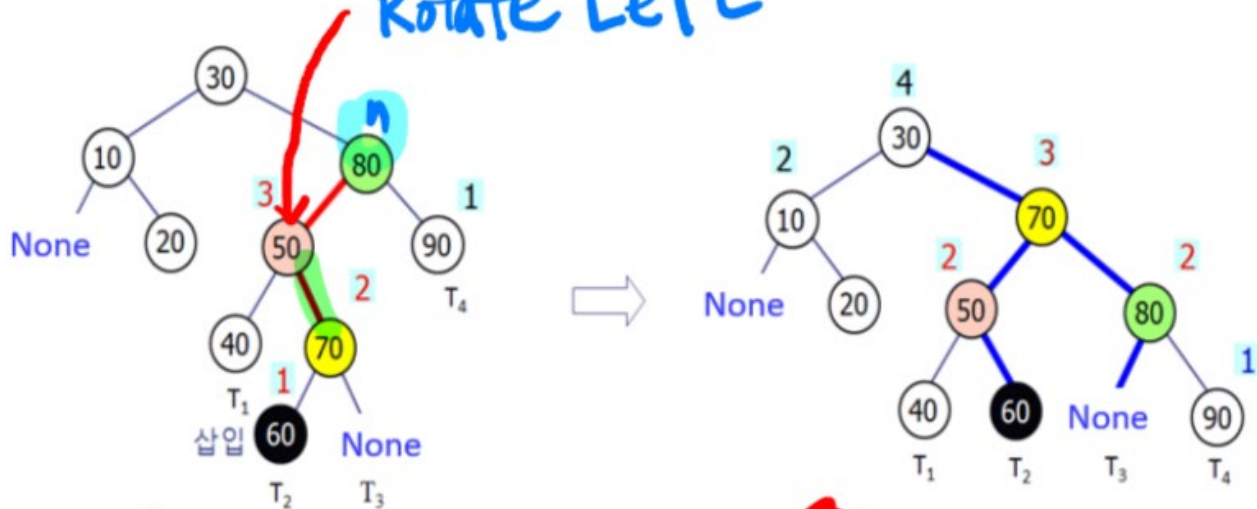
LR-회전

- (a) 20의 왼쪽 서브트리(T_2) 또는 오른쪽 서브트리(T_3)에 새로운 노드가 삽입 되어 T_2 또는 T_3 의 높이가 $h-1$ 이 됨에 따라 30의 왼쪽과 오른쪽 서브트리의 **높이 차이가 2**가 된 상태
- 30의 왼쪽(L) 서브트리의 오른쪽(R) 서브트리에서 새로운 노드가 삽입되었기 때문
- LR-회전은 rotate-left(10) 수행 후 rotate_right(30) 수행

[예제] LR-회전의 예

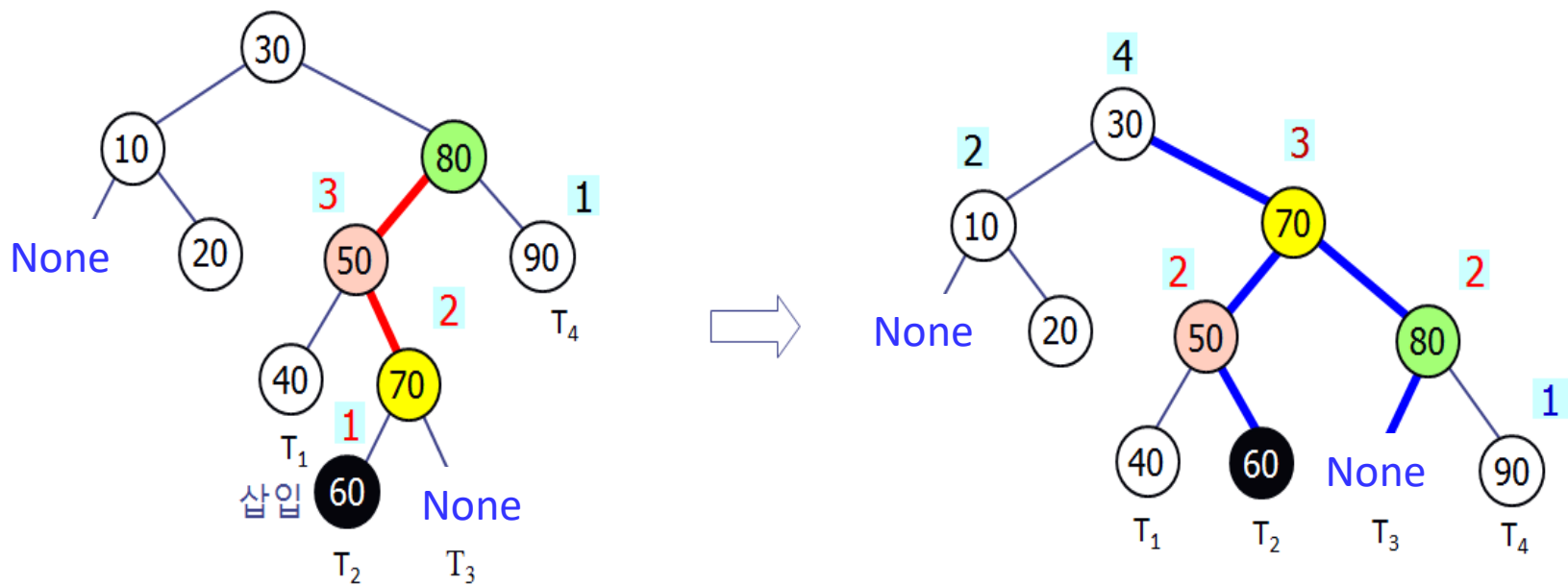
✓
n.left

Rotate Left

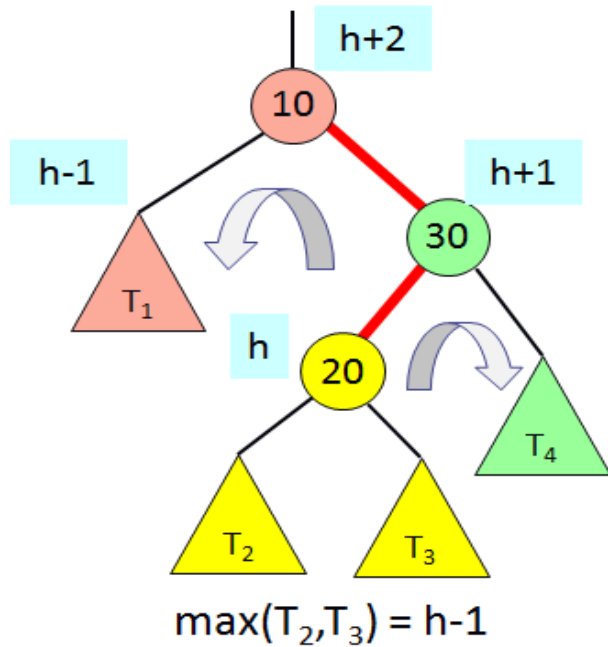


Rotate Right

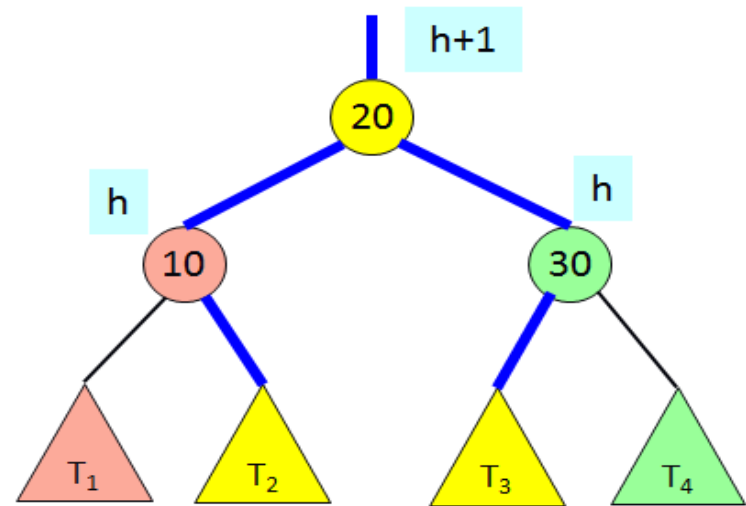
[예제] LR-회전의 예



RL-회전



(a) T_2 또는 T_3 에 새 노드 삽입

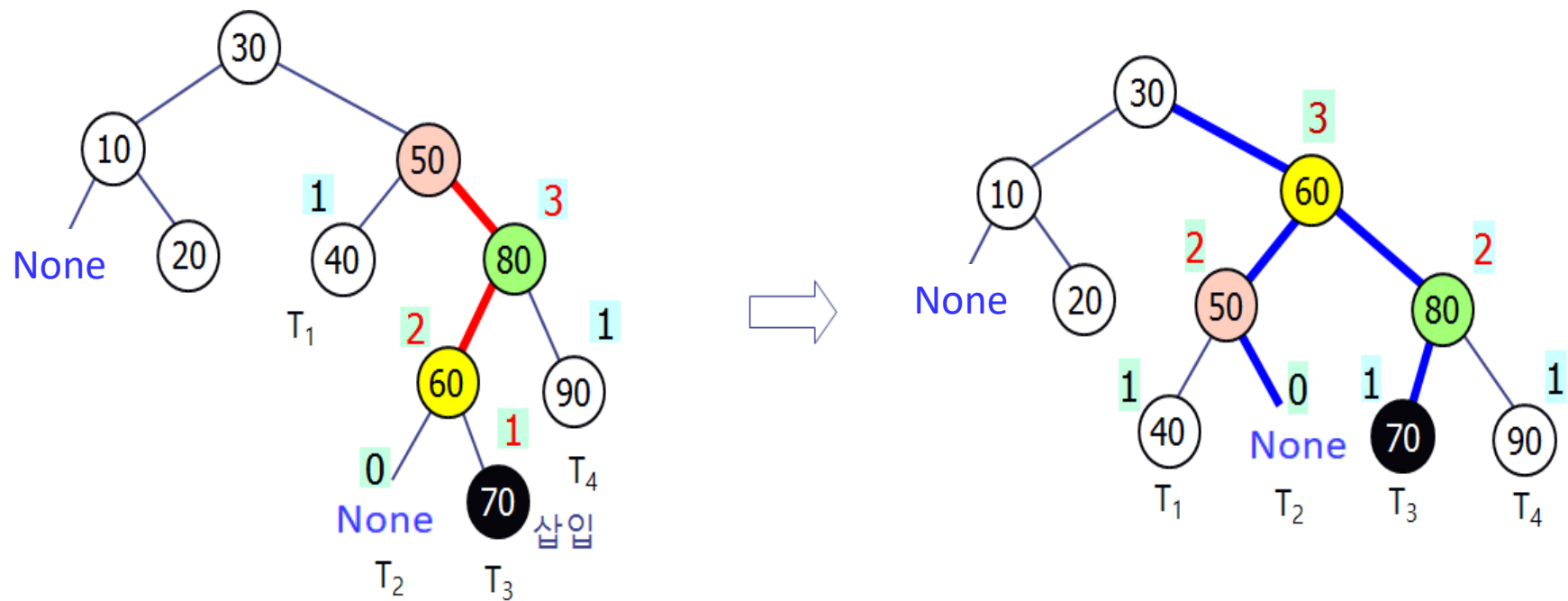


(b) RL-회전 후

RL-회전

- (a) 20의 왼쪽 서브트리(T_2) 또는 오른쪽 서브트리(T_3)에 새로운 노드가 삽입 되어 T_2 또는 T_3 의 높이가 $h-1$ 이 되고 10의 왼쪽과 오른쪽 서브트리의 **높이 차이가 2**가 된 상태
- 10의 오른쪽(R) 서브트리의 왼쪽(L) 서브트리에서 새로운 노드가 삽입되었기 때문
- RL-회전은 `rotate_right(30)` 수행한 후 `rotate_left(10)` 수행

[예제] RL-회전의 예



4종류의 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

5.3.2 삽입 연산

- 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 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)

```

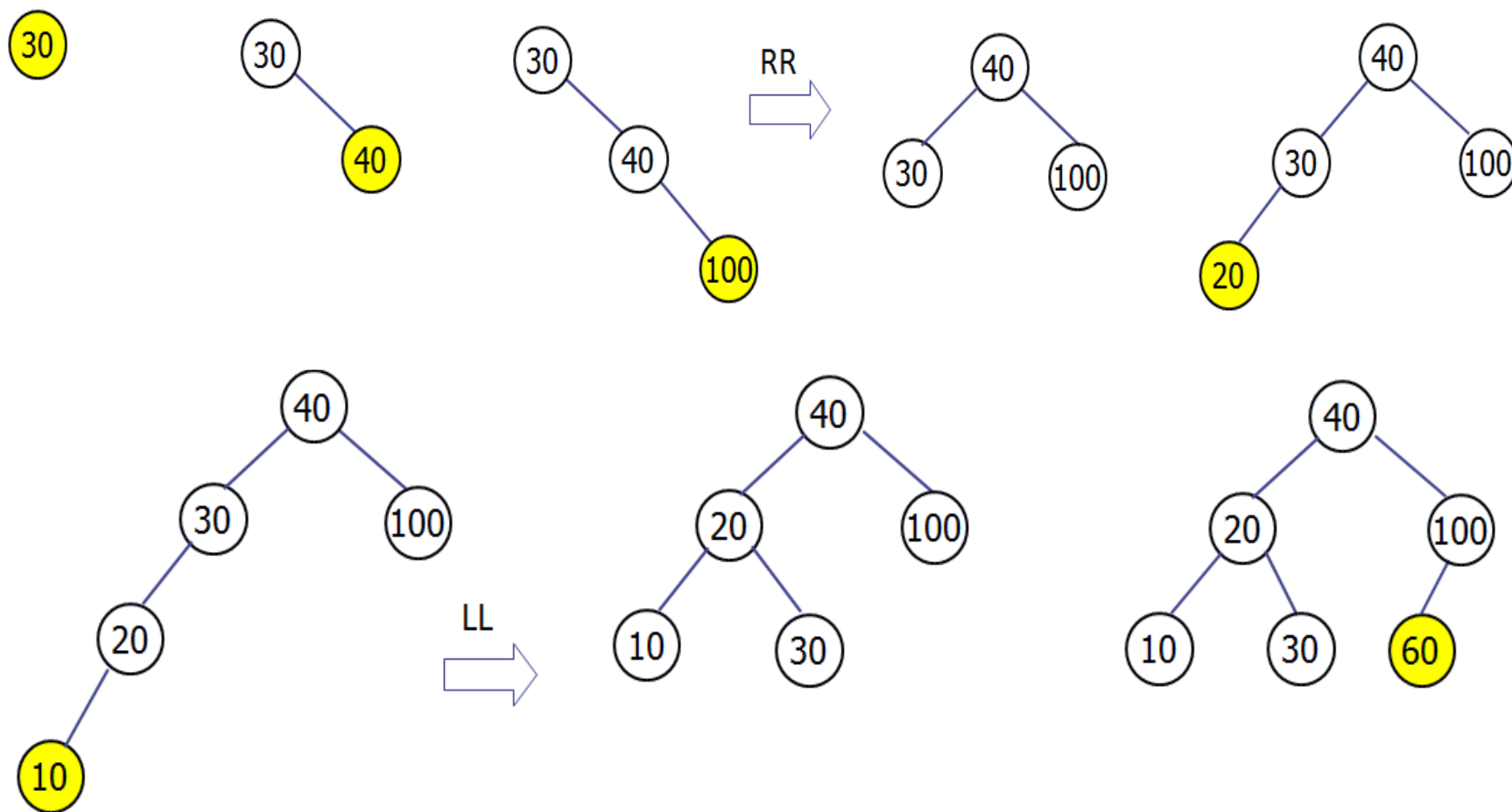
- `balance()`에서 line 02의 `bf(n) > 1`인 경우는 노드 `n`의 왼쪽 서브트리가 오른쪽 서브트리보다 높고, 그 차이가 1보다 큰 것으로 불균형 발생
- 이 때 `bf(n.left)`가 음수이면, `n.left`의 오른쪽 서브트리가 왼쪽 서브트리보다 높음
 - Line 04에서 `rotate_left(n.left)`를 수행하고 line 06에서 `rotate_right(n)`을 수행. 즉, LR-회전 수행
- `bf(n.left)`가 음수가 아니라면, line 06에서 LL-회전 만을 수행
- RR-회전과 RL-회전도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

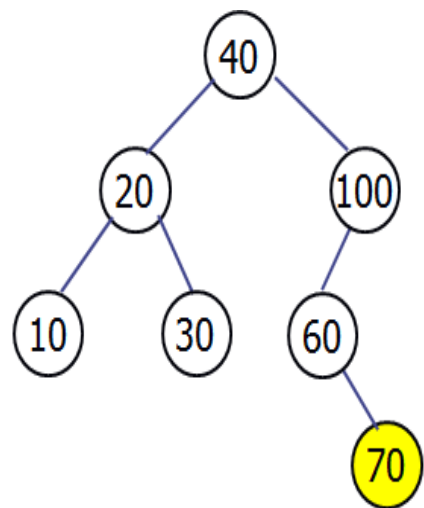
```
01 def put(self, key, value): # 삽입 연산
02     self.root = self.put_item(self.root, key, value)
03
04 def put_item(self, n, key, value):
05     if n == None:
06         return Node(key, value, 1)
07     if n.key > key:
08         n.left = self.put_item(n.left, key, value)
09     elif n.key < key:
10         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)
```

The diagram includes four callout boxes with blue lines pointing to specific lines of code:

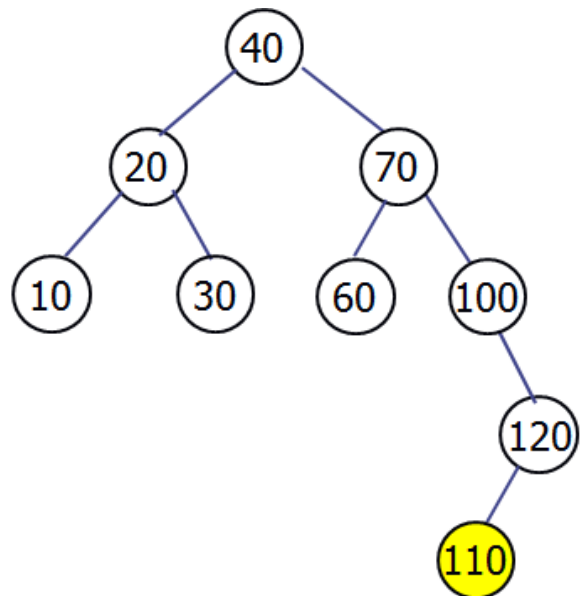
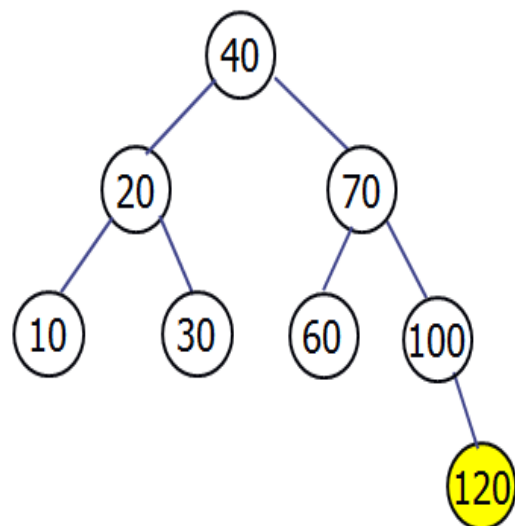
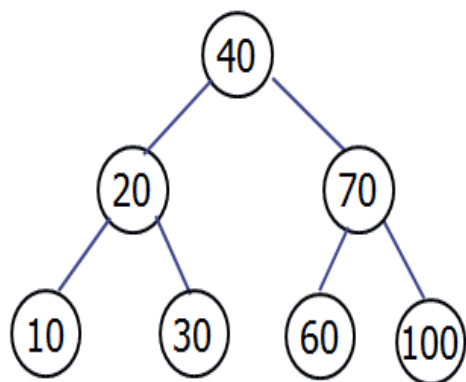
- A box at line 06 points to `return Node(key, value, 1)` with the text "새 노드 생성, 높이=1".
- A box at line 12 points to `n.value = value` with the text "key가 이미 있으면 value만 갱신".
- A box at line 14 points to `n.height = max(self.height(n.left), self.height(n.right)) + 1` with the text "노드 n의 높이 갱신".
- A box at line 15 points to `return self.balance(n)` with the text "노드 n의 균형 유지".

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

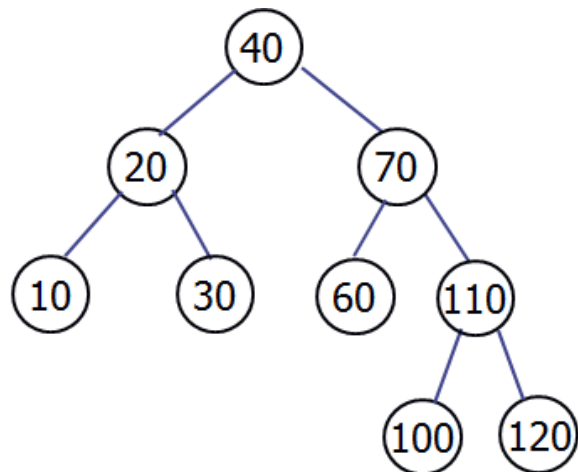




LR
⇒



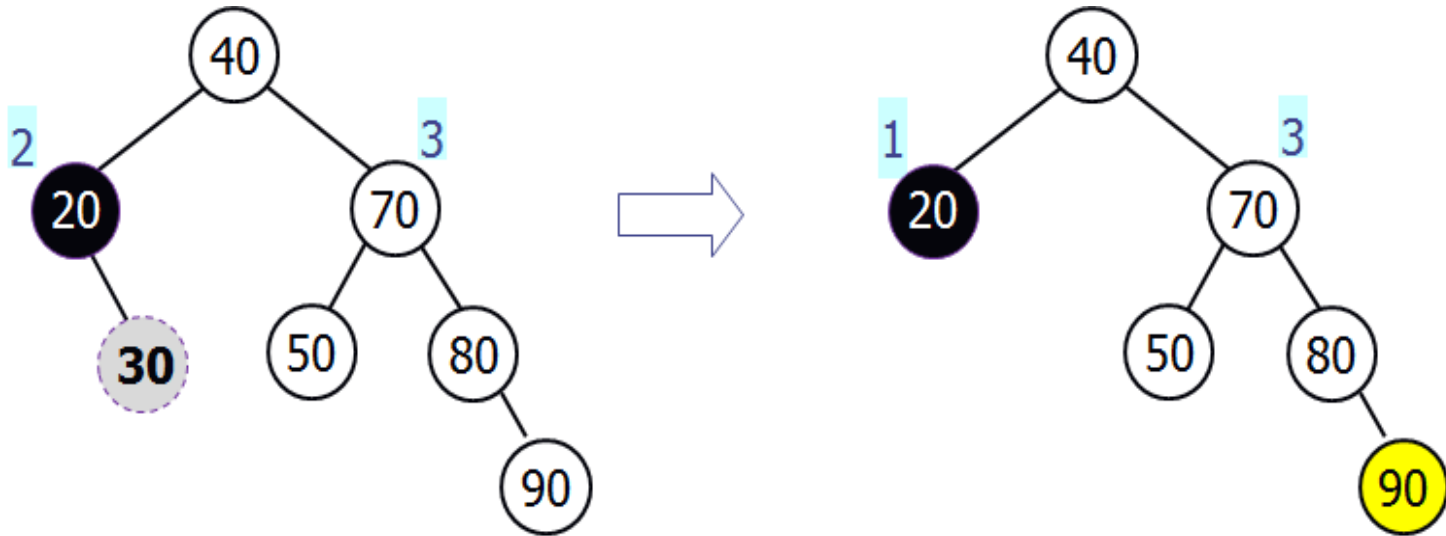
RL
⇒



5.3.3 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



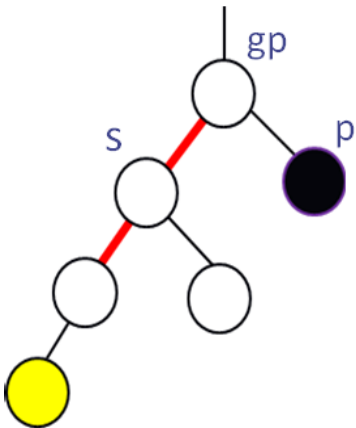
(a) 삭제 전

(b) 삭제 후 노드 40에서 불균형 발생

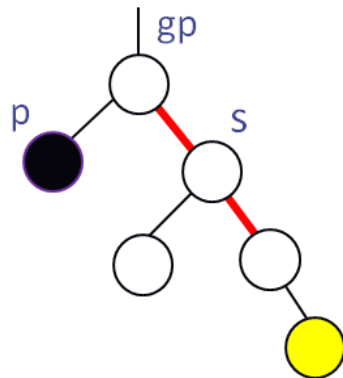
[핵심 아이디어]

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

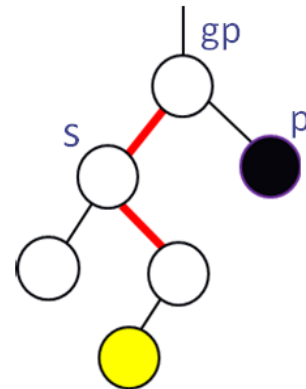
- 삭제된 노드의 부모 = p , p 의 부모 = gp , p 의 형제 = s
- s 의 왼쪽과 오른쪽 서브트리 중에서 높은 서브트리에 마치 새 노드가 삽입된 것으로 간주



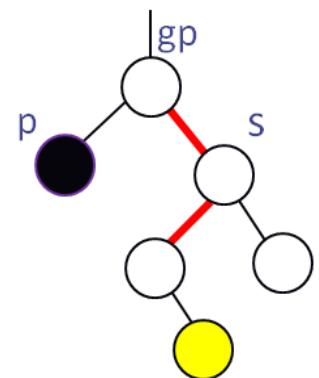
(a) LL-회전



(b) RR-회전

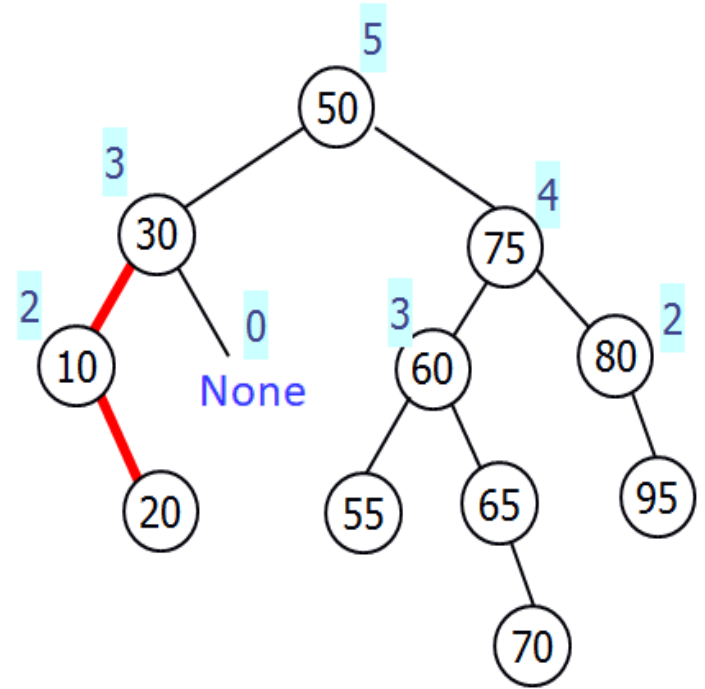
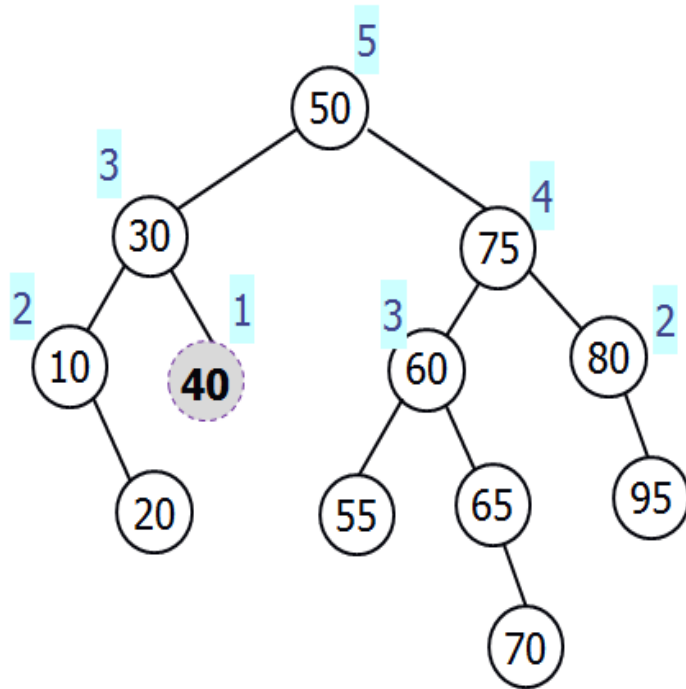


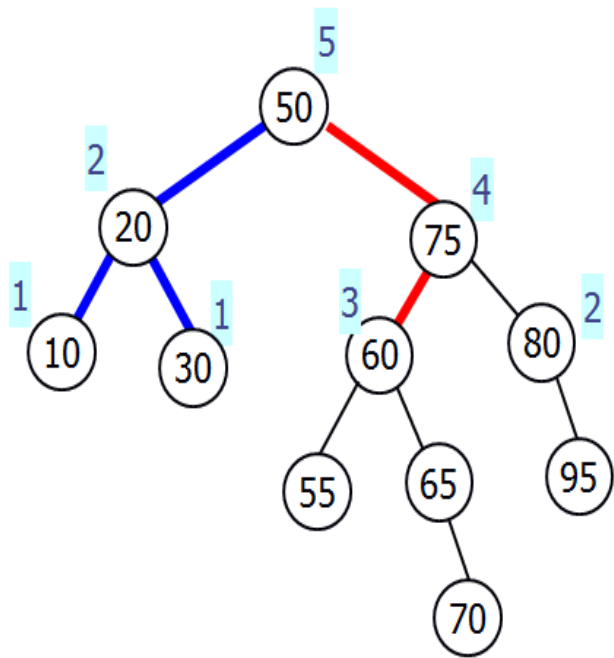
(c) LR-회전



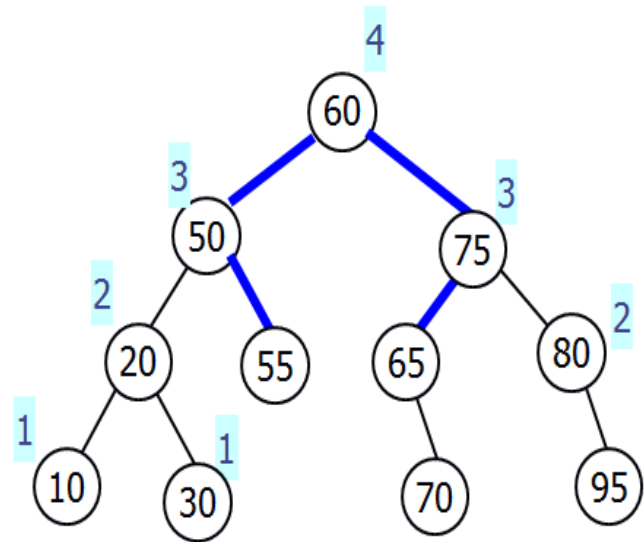
(d) RL-회전

[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
- 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 $O(\log N)$

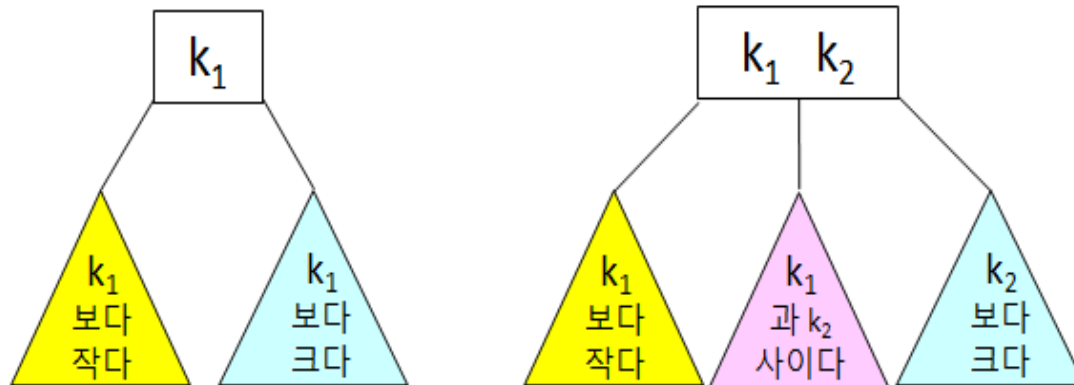
- 다양한 실험결과에 따르면, AVL 트리는 거의 정렬된 데이터를 삽입한 후에 랜덤 순서로 데이터를 탐색하는 경우 가장 좋은 성능을 보임
- 이진탐색트리는 랜덤 순서의 데이터를 삽입한 후에 랜덤 순서로 데이터를 탐색하는 경우 가장 좋은 성능을 보임

5.4 2-3 트리, 레드블랙트리

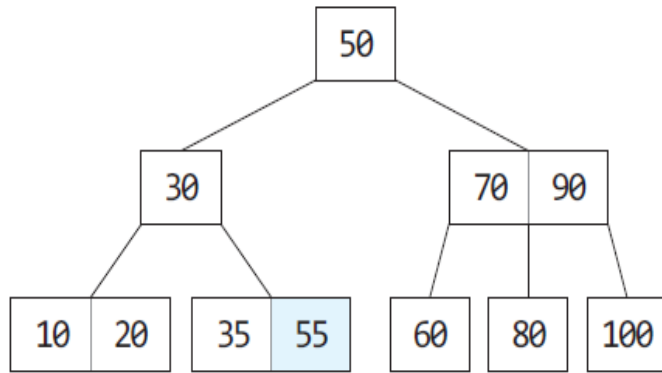
- 2-3 트리는 내부 노드의 차수가 2 또는 3인 균형 탐색트리
- 차수가 2인 노드 = 2-노드, 차수가 3인 노드 = 3-노드
- 2-노드: 한 개의 키를 가지며, 3-노드는 두 개의 키를 가짐
- 2-3 트리는 루트로부터 각 이파리까지 경로의 길이가 같고,
모든 이파리들이 동일한 층에 있는 완전한 균형트리
- 2-3 트리가 2-노드들만으로 구성되면 포화이진트리와 동일한 형태를 가짐

[핵심 아이디어] 2-3트리는 이파리노드들이 동일한 층에 있어야 하므로 트리가 위로 자라나거나 낮아진다.

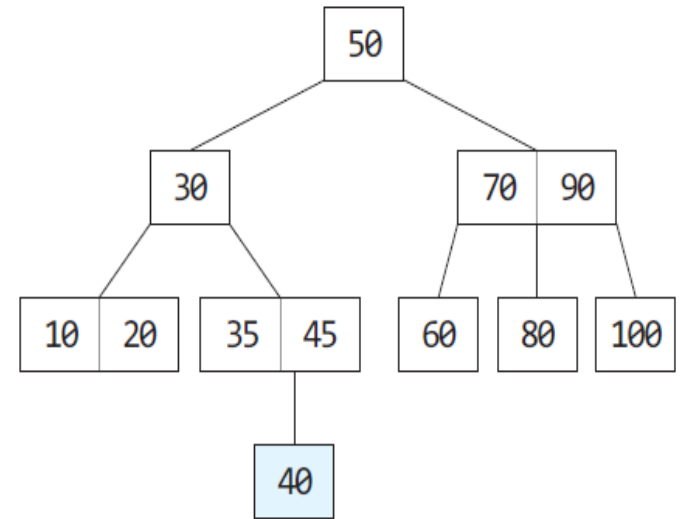
- 2-노드의 키가 k_1 이라면, 노드의 왼쪽 서브트리에는 k_1 보다 작은 키들이 있고, 오른쪽 서브트리에는 k_1 보다 큰 키들이 있다.
- k_1 과 k_2 를 가진 3-노드는 3개의 서브트리를 가지는데, 왼쪽 서브트리에는 k_1 보다 작은, 중간 서브트리에는 k_1 보다 크고 k_2 보다 작은, 오른쪽 서브트리에는 k_2 보다 큰 키들이 있다.



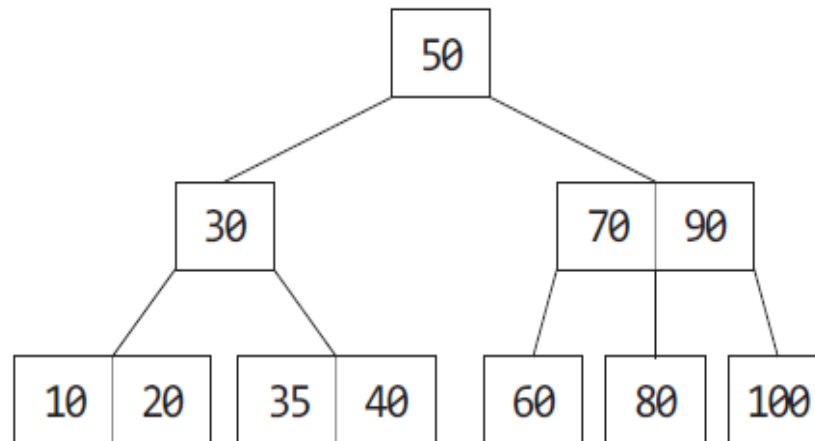
- 2-3 트리에서도 이진탐색트리에서의 중위순회와 유사한 방법으로 중위순회 수행
- 2-노드는 이진트리의 중위순회 방문과 동일
- k_1 과 k_2 를 가진 3-노드에서는 먼저 노드의 왼쪽 서브트리에 있는 모든 노드들을 방문한 후에 k_1 을 방문하고, 이후에 중간 서브트리에 있는 모든 노드들을 방문
- 다음으로 k_2 를 방문하고 마지막으로 오른쪽 서브트리에 있는 모든 노드들을 방문한다.
- 따라서 2-3트리에서 중위순회를 수행하면 키들이 정렬된 결과를 얻음



(a)



(b)



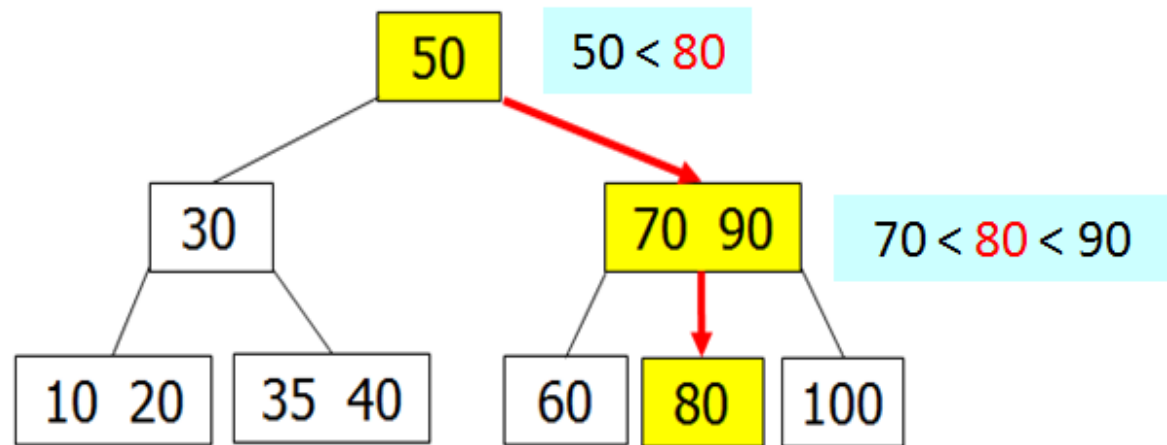
(c)

[그림 5-32] 어느 트리가 2-3 트리인가?

5.3.1 탐색 연산

- 루트에서 시작하여 방문한 노드의 키들과 탐색하고자 하는 키를 비교하며 다음 레벨의 노드를 탐색

[예제] 80 탐색



- 2-3 트리는 이진탐색트리에 비해 매우 우수한 성능을 보이나, 2-3 트리를 실제로 구현하기에 다소 복잡함
- 구현이 어려운 이유는 노드를 2 개의 타입으로 정의해야 하고, 분리 및 통합 연산에서의 다양한 경우를 고려해야 하기 때문
- 3-노드에서는 키를 2회 비교하는 것도 고려해야
- 2-3 트리는 좌편향(Left-Leaning) 레드블랙트리의 기본 형태를 제공

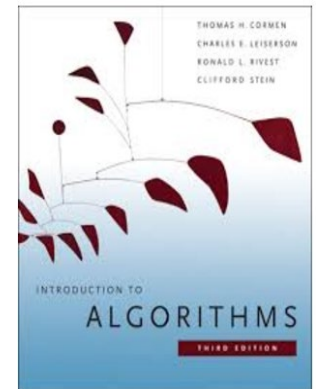
2-3-4 트리

- 2-3 트리를 확장한 2-3-4 트리는 노드가 자식을 4개까지 가질 수 있는 **완전균형트리**
- 2-3-4 트리의 장점: 2-3 트리보다 높이가 낮아 그 만큼 빠른 탐색, 삽입, 삭제 연산이 수행이 가능
- 2-3-4 트리에서는 삽입 연산을 루트부터 이파리로 내려가며 4-노드를 만날 때마다 **미리 분리 연산을 수행**할 수 있기 때문에 다시 이파리부터 위로 올라가며 분리 연산을 수행할 필요가 없고, 따라서 보다 효율적인 삽입 연산이 가능 (분리 연산은 B-트리에서 설명)

- 삭제 연산도 삽입 연산과 유사하게 루트로부터 이파리 방향으로 내려가며 2-노드를 만날 때마다 미리 통합 연산을 수행하므로 키를 삭제한 후 다시 루트 방향으로 올라가며 통합 연산을 수행할 필요 없음 (통합 연산은 B-트리에서 설명)
- 그러나 이러한 삽입과 삭제 연산도 이론적으로는 2-3 트리의 수행시간과 동일한 $O(\log_2 N)$

5.4.2 레드블랙트리

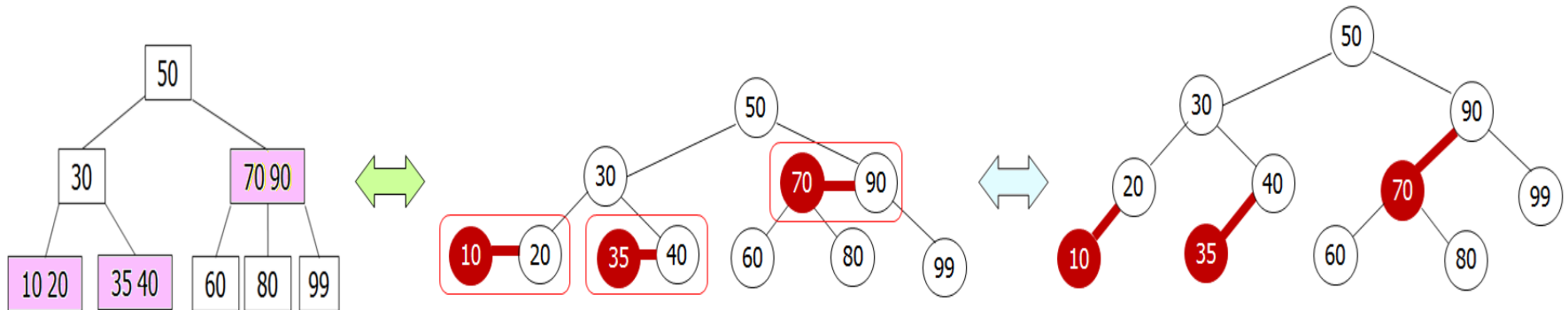
- 노드에 색을 부여하여 트리의 균형을 유지
- 탐색, 삽입, 삭제 연산의 수행시간이 각각 $O(\log N)$ 을 넘지 않는 매우 효율적인 자료구조
- 일반적인 레드블랙트리(Intro. to Algorithms, CLRS):
삽입이나 삭제를 수행할 때 트리의 균형을 유지하기 위해 상당히 많은 경우를 고려해야 한다는 단점이 있으며, 이에 따라 프로그램이 복잡하고, 그 길이도 증가함
- 좌편향 레드블랙(Left-Leaning Red-Black, LLRB)트리: 삽입이나 삭제 시 고려해야 하는 경우의 수가 매우 적어 프로그램의 길이도 일반 레드블랙트리 프로그램의 1/5정도에 불과



- LLRB 트리는 AVL 트리, 2-3 트리, 2-3-4 트리, 일반 레드블랙트리보다 매우 우수한 성능을 가짐
- Introduction to Algorithms (CLRS)에 소개된 레드블랙트리가 일반적으로 사용되며, 전문 프로그래머가 프로그램을 작성해도 적어도 400 line이 나 든다.

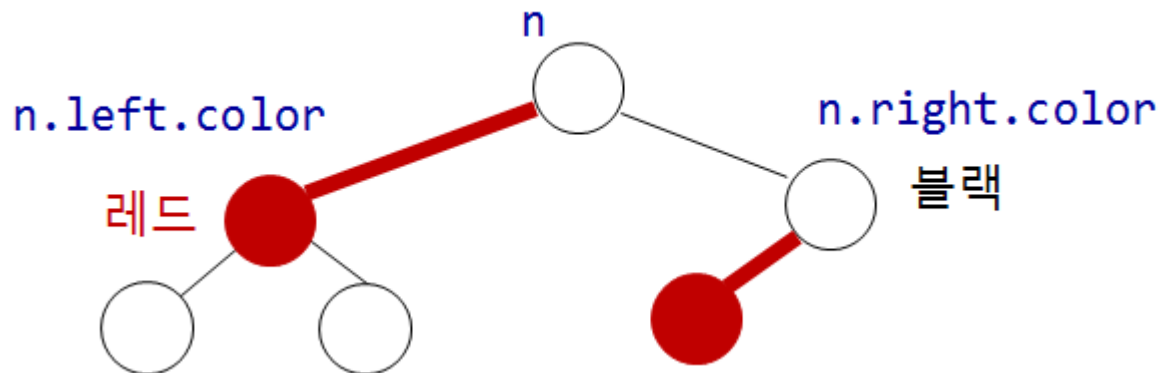
[핵심 아이디어]

LLRB 트리는 2-3 트리에서 3-노드의 두 개의 키를 두 노드에 분리 저장하고, 작은 키는 레드, 큰 키는 블랙으로 만든 형태와 같다.



LLRB 트리와 2-3 트리의 관계

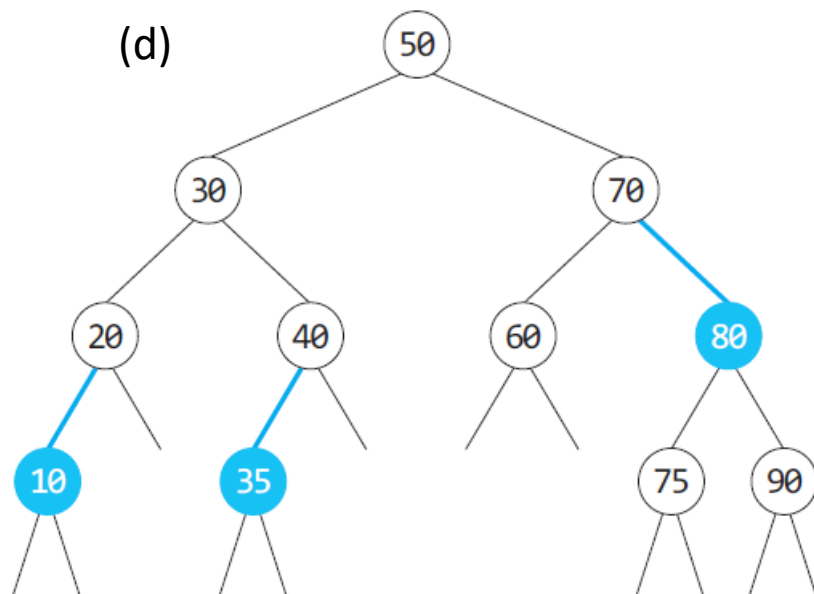
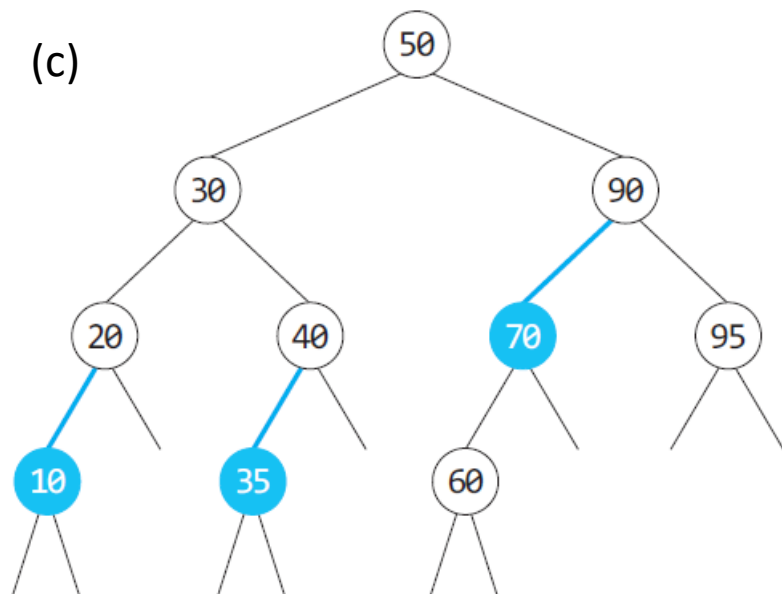
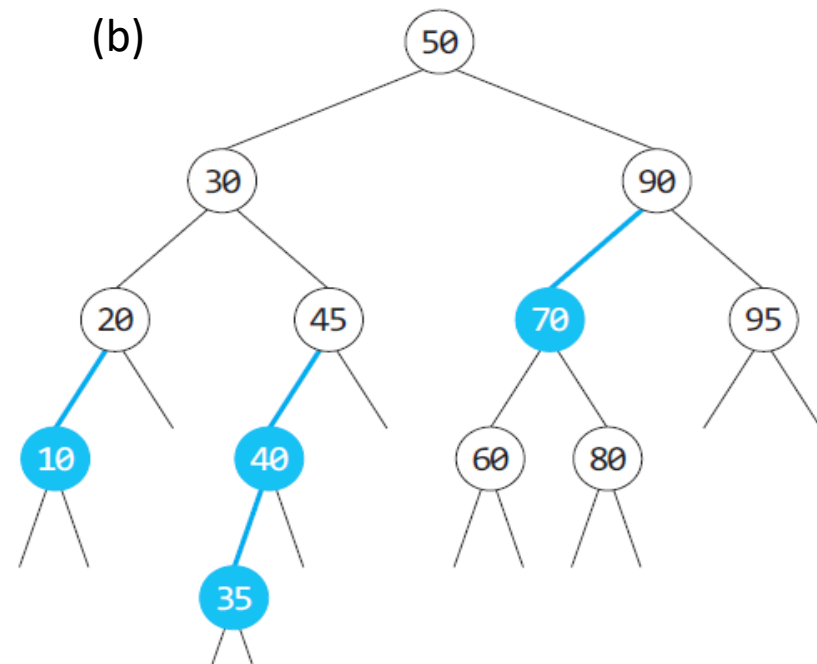
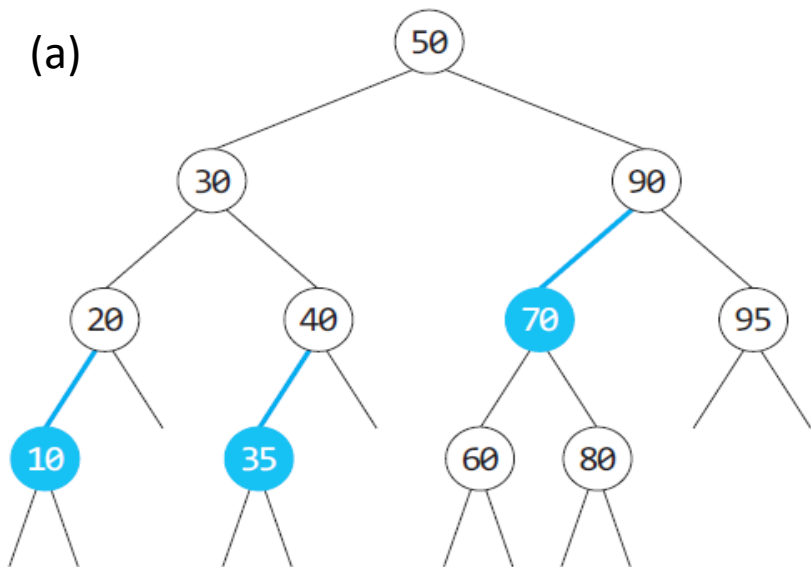
- LLRB트리는 개념적으로 2-3트리와 같기 때문에 2-3트리의 장점인 **완전균형트리의 형태를 내포**
- LLRB 트리의 노드는 블랙 또는 레드의 두 가지 색 정보를 가지며, 노드와 부모를 연결하는 link의 색은 노드의 색과 동일
- 따라서 LLRB 트리에서는 link의 색을 별도로 저장 안함
- 노드 n의 왼쪽 자식은 레드이고 그 연결 link도 레드이며, n의 오른쪽 자식은 블랙이고 그 연결 link도 블랙





LLRB트리 는 이진탐색트리로서 다음의 4가지 조건을 만족

- 루트와 None 은 블랙이다.
- 루트로부터 각 None까지 2개의 연속된 레드 link는 없다. (연속 레드 link 규칙)
- 루트로부터 각 None까지의 경로에 있는 블랙 link 수는 모두 같다. (동일 블랙 link 수 규칙)
- 레드 link는 왼쪽으로 기울어져 있다. (레드 link 좌편향 규칙)



[그림 5-35] 어떤 트리가 LLRB 트리인가?

Applications

- 레드블랙트리는 반드시 제한된 시간 내에 연산이 수행되어야 하는 경우에 매우 적합한 자료구조이다.
- 실제 응용사례로는 $\log N$ 시간보다 조금이라도 지체될 경우 매우 치명적인 상황을 야기할 수 있는
 - 항공 교통 관제(Air Traffic Control)
 - 핵발전소의 원자로(Nuclear Reactor) 제어
 - 심장박동 조정장치(Pacemakers) 등

- 레드블랙트리는 자바의 `java.util.TreeMap`과 `java.util.TreeSet`의 기본 자료구조로 사용되며,
- C++ 표준 라이브러리인 `map`, `multimap`, `set`, `multiset`에도 사용되고,
- 리눅스(Linux) 운영체제의 스케줄러에서도 레드블랙트리가 활용

5.5 B-트리

- 다수의 키를 가진 노드로 구성되어 **다방향 탐색(Multiway Search)**이 가능한 **균형 트리**
- 2-3 트리는 B-트리의 일종으로 노드에 키가 2 개까지 있을 수 있는 트리
- B-트리는 대용량의 데이터를 위해 고안되어 주로 데이터베이스에 사용

[핵심아이디어]

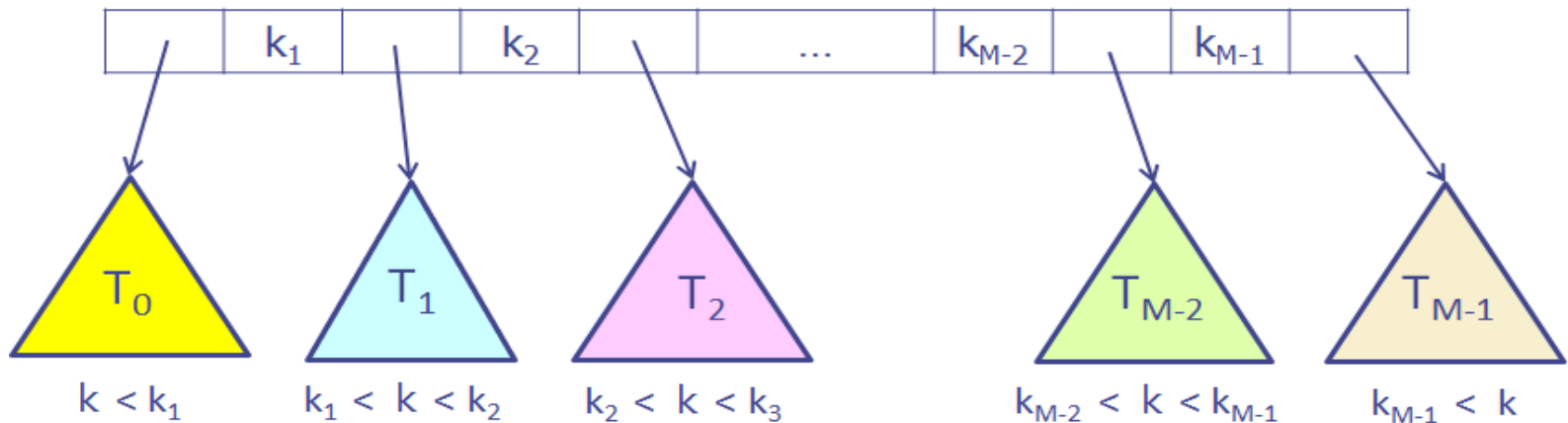
노드에 수백에서 수천 개의 키를 저장하여 트리의 높이를 낮추자.



차수가 M 인 B-트리

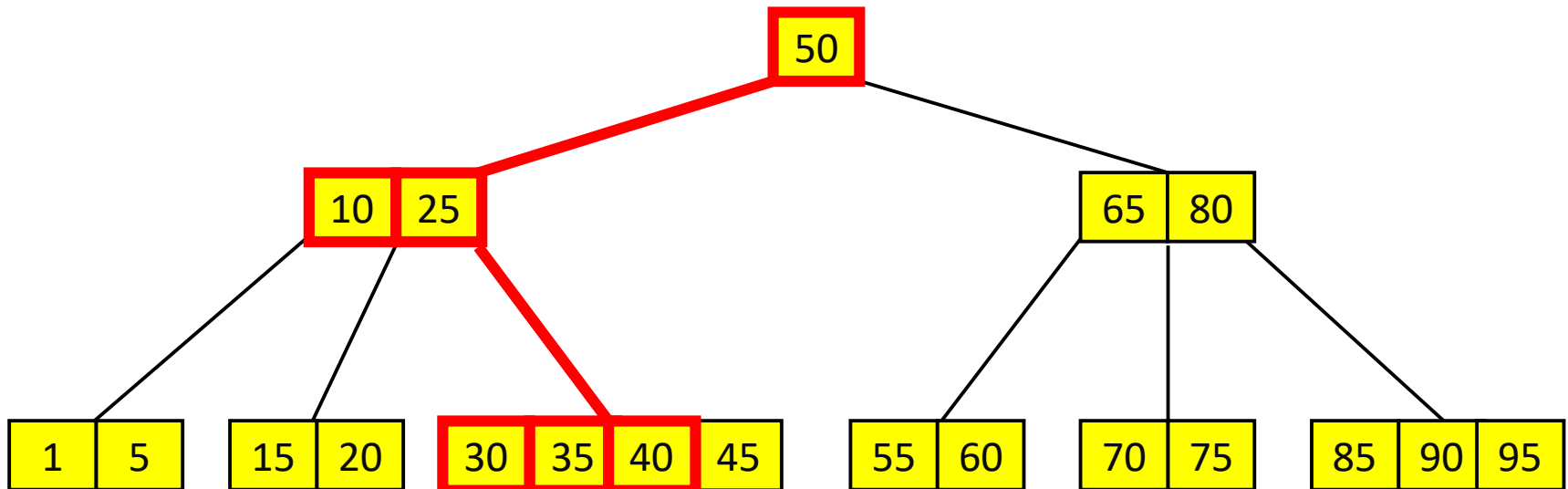
- 모든 이파리들은 동일한 깊이를 갖는다.
- 각 내부 노드의 자식 수는 $\lceil M/2 \rceil$ 이상 M 이하이다.
- 루트의 자식 수는 2 이상이다.

2-3 트리는 차수가 3인 B-트리이고, 2-3-4 트리는 차수가 4인 B-트리이다.



5.5.1 탐색 연산

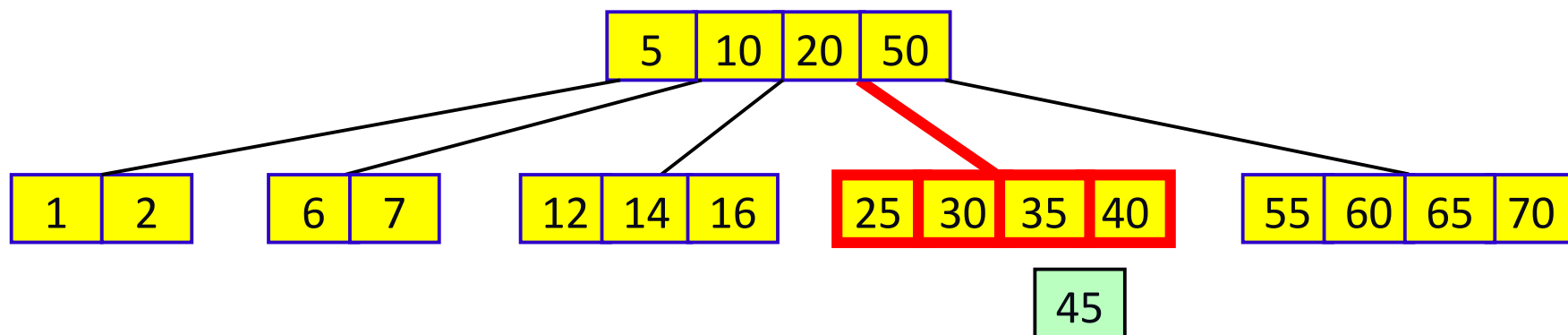
- B-트리에서의 탐색은 루트로부터 시작
- 방문한 각 노드에서는 탐색하고자 하는 키와 노드의 키들을 비교하여, 적절한 서브트리를 탐색
- 단, B-트리의 노드는 일반적으로 수백 개가 넘는 키를 가지므로 각 노드에서는 **이진탐색**을 수행



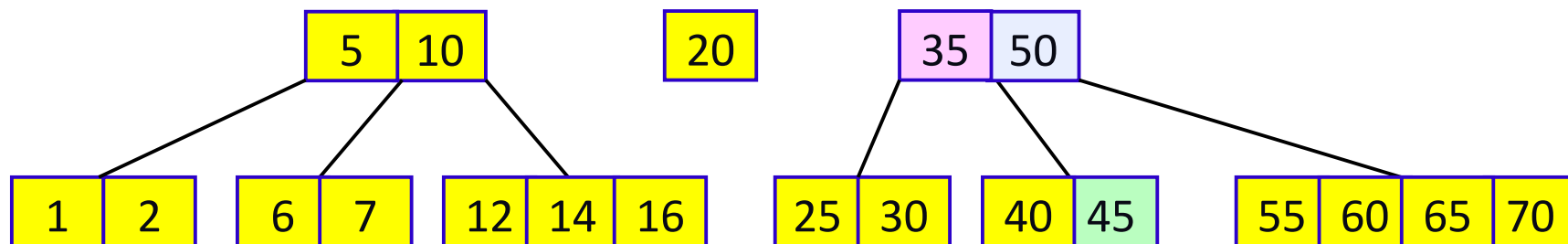
5.5.2 삽입 연산

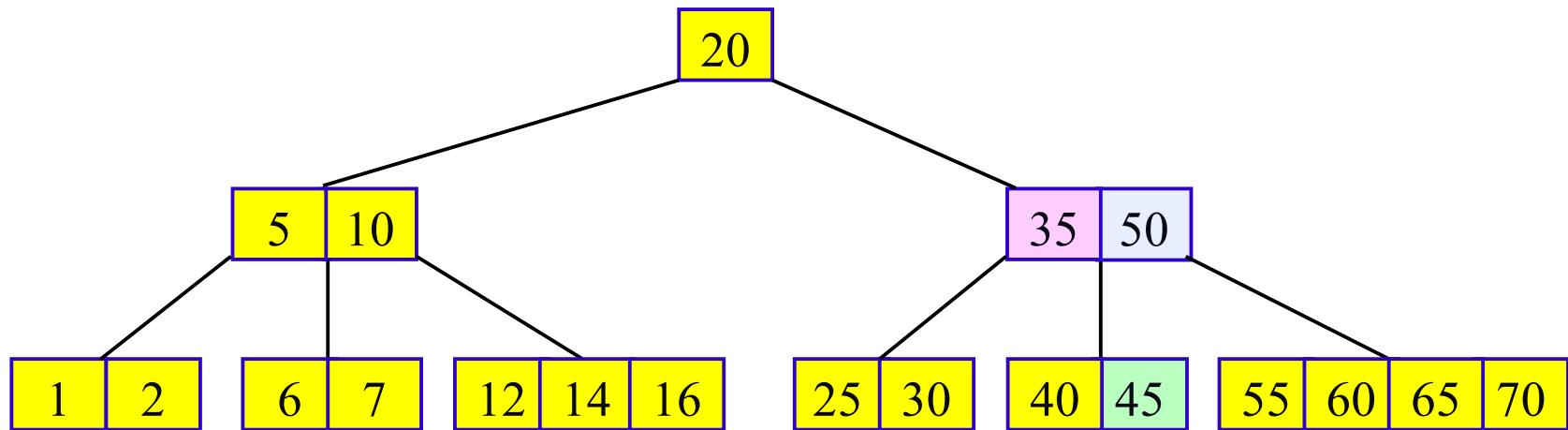
- B-트리에서의 삽입은 탐색과 동일한 과정을 거쳐 새로운 키가 저장되어야 할 이파리노드를 찾는다.
- 이파리에 새 키를 수용할 공간이 있다면, 노드의 키들이 정렬 상태를 유지하도록 새 키를 삽입
- 이파리가 이미 $M-1$ 개의 키를 가지고 있으면, 이 $M-1$ 개의 키들과 새로운 키 중에서 중간값이 되는 키(중간키)를 부모로 올려 보내고, 남은 $M-1$ 개의 키들을 $1/2$ 씩 나누어 각각 별도의 노드에 저장한다. [분리(Split) 연산]

삽입 45



중간키

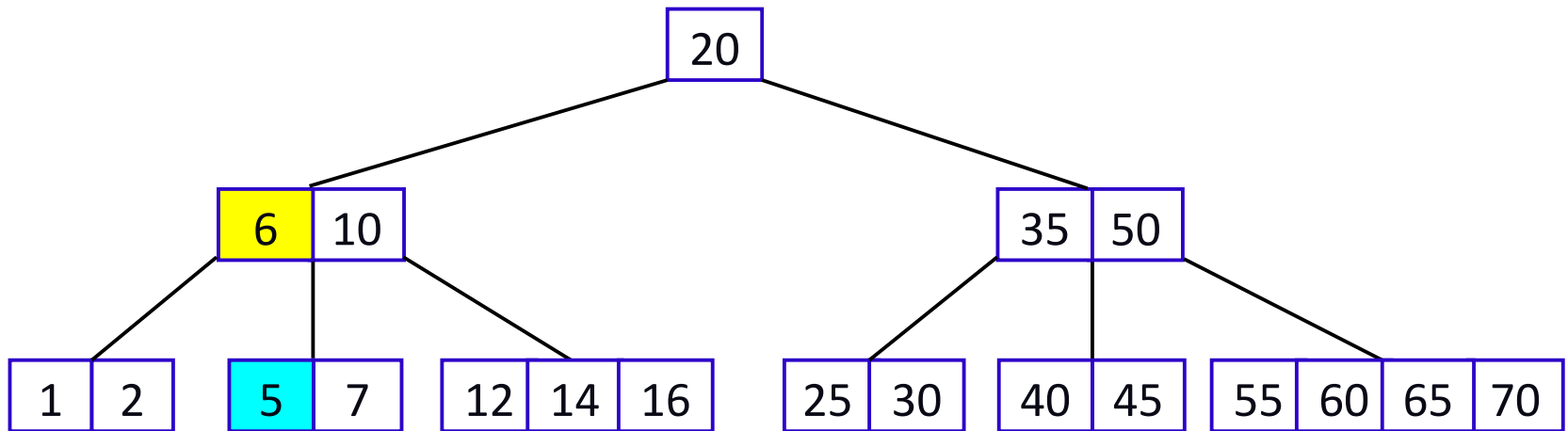
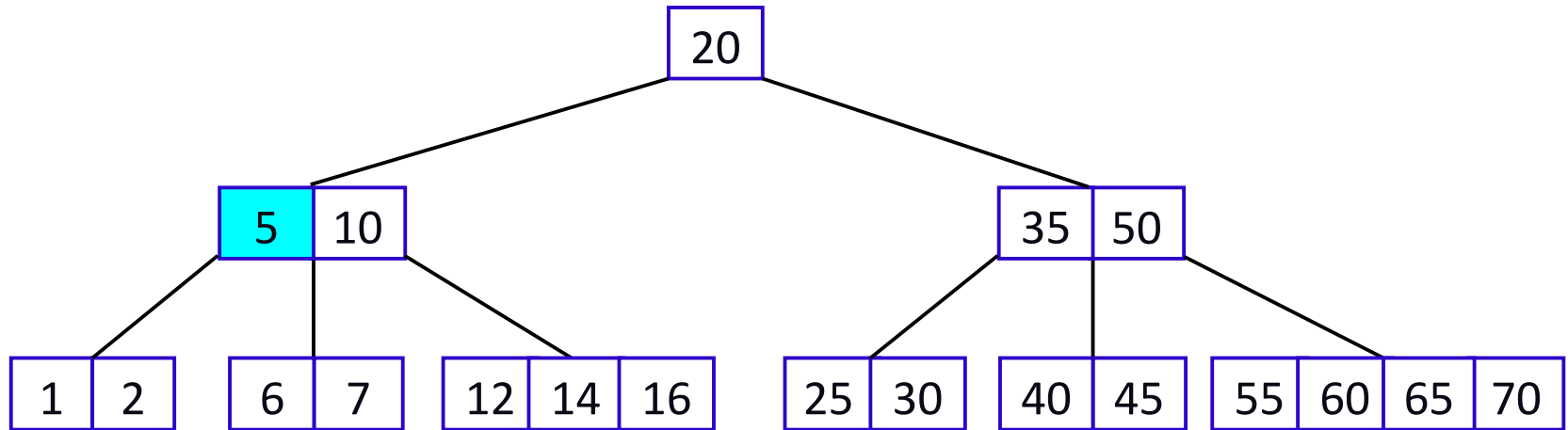


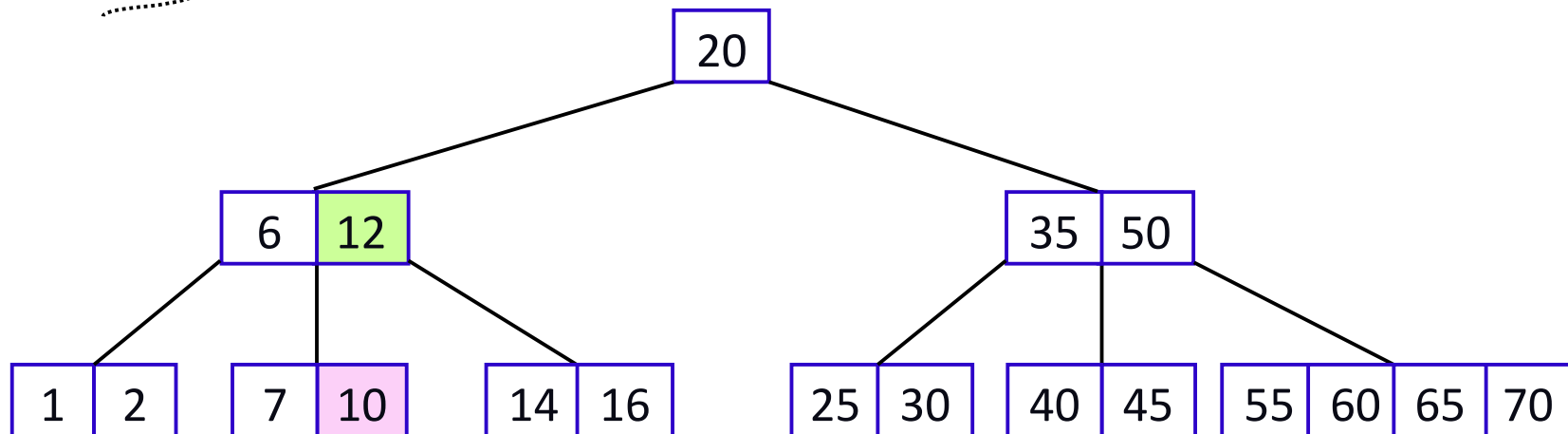
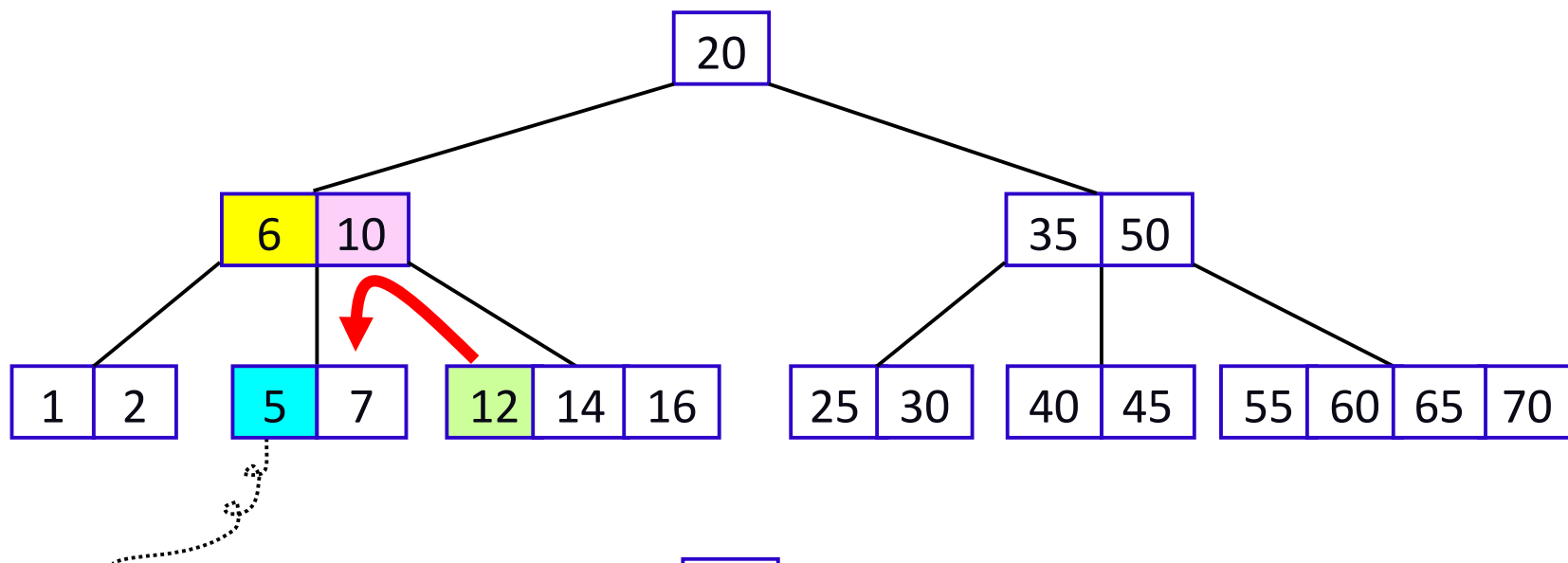


5.5.3 삭제 연산

- B-트리에서의 삭제는 항상 이파리에서 이루어진다.
- 만약 삭제할 키가 속한 노드가 이파리가 아니면, 이진탐색트리의 삭제와 유사하게 중위 선행자나 중위 후속자를 삭제할 키와 교환한 후에 이파리에서 삭제를 수행
- 삭제는 이동(Transfer) 연산과 통합(Fusion) 연산을 사용
- 이동 연산: 이파리노드에서 키가 삭제된 후에 키의 수가 $\lceil M/2 \rceil - 1$ 보다 작으면, 자식 수가 $\lceil M/2 \rceil$ 보다 작게 되어 B-트리 조건을 위반. 이 때 노드의 좌우의 형제들 중에서 도움을 줄 수 있는 노드로부터 1 개의 키를 부모를 통해 이동

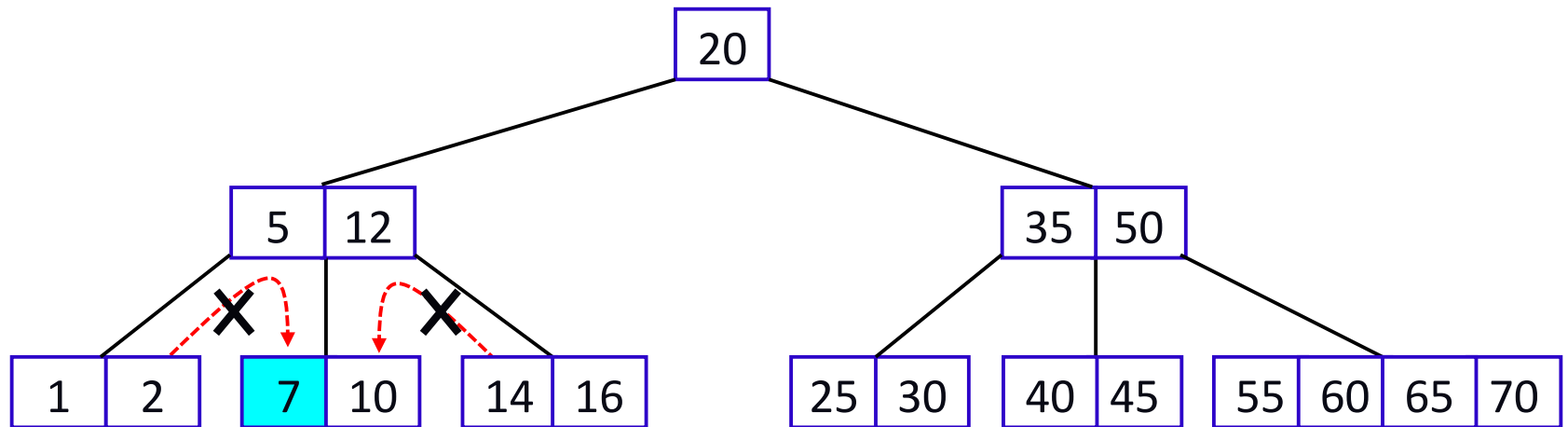
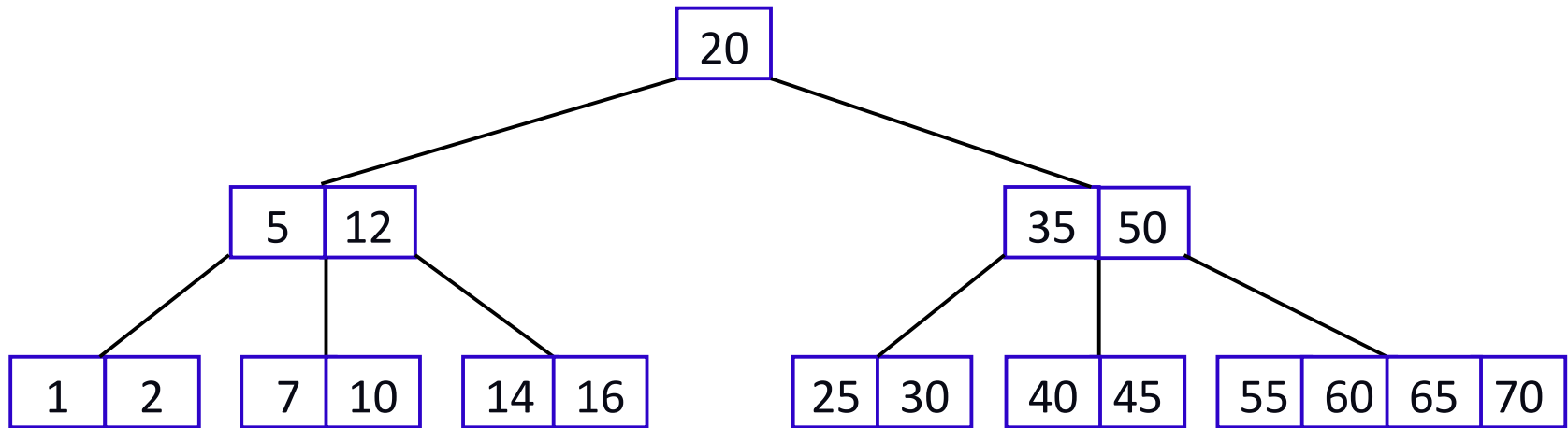
5를 삭제하는 과정



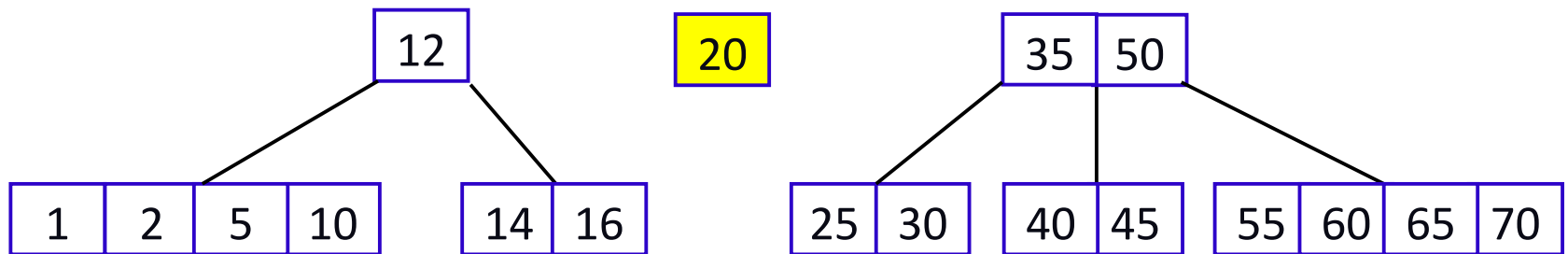
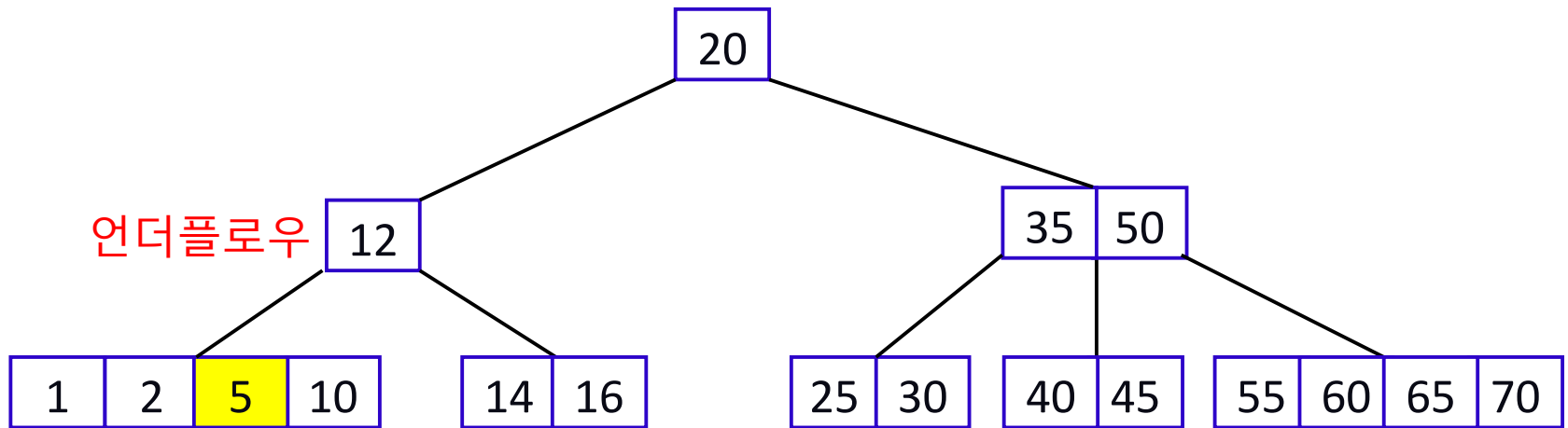


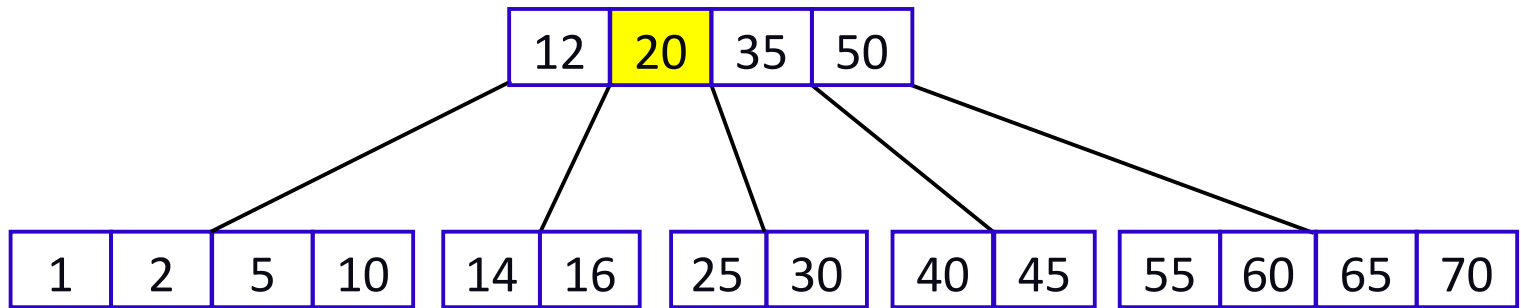
- **통합 연산:** 키가 삭제된 후 underflow가 발생한 노드 x 에 대해 이동 연산이 불가능한 경우, 노드 x 와 그의 형제를 1 개의 노드로 통합하고, 노드 x 와 그의 형제의 분기점 역할을 하던 부모의 키를 통합된 노드로 끌어내리는 연산

7을 삭제하는 과정



언더플로우

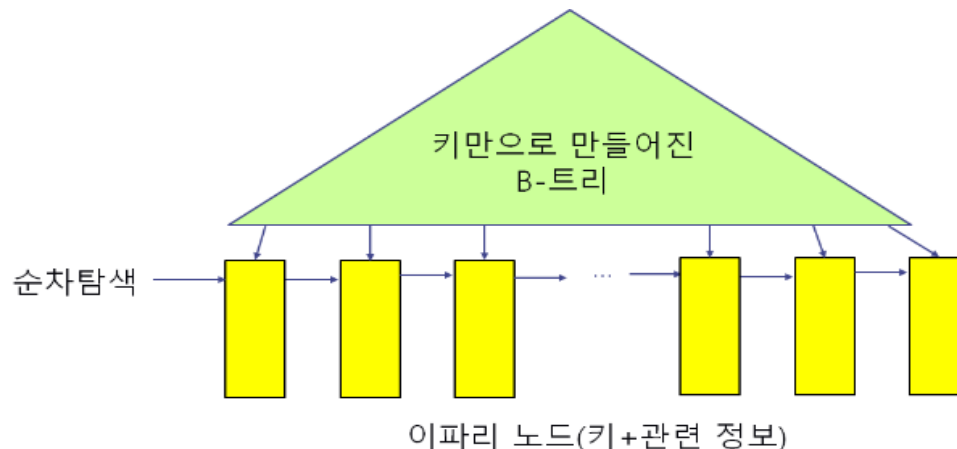




5.5.4 B-트리의 확장

- B*-트리¹는 B-트리로서 루트를 제외한 다른 노드의 자식 수가 $2/3M \sim M$ 이어야 한다.
 - 즉, 각 노드에 적어도 $2/3$ 이상이 키들로 채워져 있어야
- B-트리에 비해 B*-트리는 공간을 효율적으로 활용

- B⁺-트리는 실세계에서 가장 널리 활용되는 B-트리
- B⁺-트리는 **키들만으로 가지고 B-트리**를 구성, 이파리노드에 키와 관련(실제) 정보를 저장
- 키들로 구성된 B-트리는 탐색, 삽입, 삭제 연산을 위해 관련된 이파리노드를 빠르게 찾을 수 있도록 안내해주는 역할만을 수행
- B⁺-트리는 전체 레코드를 **순차적으로 접근**할 수 있도록 이파리들은 **연결리스트**로 구현



성능 분석

- B-트리에서 삽입이나 삭제 연산의 수행시간은 각각 B-트리의 높이에 비례. 차수가 M 이고 키의 개수가 N 인 B-트리의 최대 높이는 $O(\log_{M/2} N)$ 이다.
- B-트리는 키들의 비교 횟수보다 디스크와 메인 메모리 사이의 블록 이동(Transfer) 수를 최소화해야
- B-트리의 최고 성능을 위해선 1 개의 노드가 1 개의 디스크 페이지에 맞도록 차수 M 을 정함

- 실제로 B-트리들은 M의 크기를 수백에서 수천으로 사용
 - 예를 들어, $M = 200$ 이고 $N = 1$ 억이라면 B-트리의 연산은 4개의 디스크 블록만 메인 메모리로 읽어 들이면 처리 가능하다.
- 성능향상을 위해 루트는 메인 메모리에 상주시킨다.

Applications

- B-트리, B⁺-트리는 대용량의 데이터를 저장하고 유지하는 다양한 데이터베이스 시스템의 기본 자료구조로 활용
- Windows 운영체제의 파일 시스템인 HPFS(High Performance File System)
- 매킨토시 운영체제의 파일 시스템인 HFS(Hierarchical File System)과 HFS+
- 리눅스 운영체제의 파일 시스템인 ReiserFS, XFS, Ext3FS, JFS
- 상용 데이터베이스인 ORACLE, DB2, INGRES와 오픈소스 DBMS인 PostgreSQL에서 사용



요약

- 이진탐색트리는 이진탐색의 개념을 트리 형태의 구조에 접목시킨 자료구조
- 이진탐색트리의 각 노드 n 의 키가 n 의 왼쪽 서브트리의 키들보다 크고, n 의 오른쪽 서브트리의 키들보다 작다.
- 이진탐색트리 탐색, 삽입, 삭제 연산의 수행시간은 각각 **트리 높이에 비례**
- AVL트리는 임의의 노드 x 에 대해 노드 x 의 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 차이가 1을 넘지 않는 이진탐색트리
- AVL트리는 트리가 한쪽으로 치우쳐 자라나는 것을 LL, LR, RR, RL-회전 연산들을 통해 균형을 유지
- AVL트리의 탐색, 삽입, 삭제 연산의 수행시간은 각각 **$O(\log N)$**

- 2-3 트리는 내부 노드의 차수가 2~3인 완전 균형탐색트리
- 2-3 트리의 탐색, 삽입, 삭제 연산의 수행시간은 각각 트리의 높이에 비례하므로 $O(\log N)$
- 2-3-4 트리는 2-3 트리를 확장한 트리로서 4-노드까지 허용
- 2-3-4 트리에서는 루트로부터 이파리노드로 한 번만 내려가며 미리 분리 또는 통합 연산을 수행하는 효율적인 삽입 및 삭제가 가능
- 레드블랙트리는 노드의 색을 이용하여 트리의 균형을 유지하며, 탐색, 삽입, 삭제 연산의 수행시간이 각각 $O(\log N)$ 을 넘지 않는 매우 효율적인 자료구조

- N개의 노드를 가진 레드블랙트리의 높이 h 는 $2\log N$ 보다 크지 않다. 탐색, 삽입, 삭제의 수행시간은 $O(\log N)$
- B-트리는 다수의 키를 가진 노드로 구성되어 다방향 탐색이 가능한 완전 균형트리
- B*-트리는 B-트리로서 루트를 제외한 다른 노드의 자식 수가 $2/3M \sim M$ 이어야 한다. B*-트리는 노드의 공간을 B-트리보다 효율적으로 활용하는 자료구조
- B⁺-트리는 키들만을 가지고 B-트리를 만들고, 이파리노드에 키와 관련 정보를 저장
- B-트리는 몇 개의 디스크 페이지(블록)를 메인 메모리로 읽어 들이는지가 더 중요하므로 한 개의 노드가 한 개의 디스크 페이지에 맞도록 차수 M 을 정함