

관계 중심의 사고법

# 쉽게 배우는 알고리즘

## 6장. 검색트리

# 6장. 검색트리

나는 좀 더 응용력 있는 유형의 수학이라는 이유 때문에  
컴퓨터 과학을 하고 싶었다.

-로버트 타잔

# 학습목표

- 검색에서 레코드와 키의 역할을 구분한다.
- 이진 검색 트리에서 검색·삽입·삭제 작업의 원리를 이해한다.
- 이진 검색 트리의 균형이 작업의 효율성에 미치는 영향을 이해하고 ,
- 레드 블랙 트리의 삽입·삭제 작업의 원리를 이해한다.
- B-트리의 도입 동기를 이해하고 검색·삽입·삭제 작업의 원리를 이해한다.
- 검색 트리 관련 작업의 점근적 수행 시간을 이해한다.
- 일차원 검색의 기본 원리와 다차원 검색의 연관성을 이해한다.

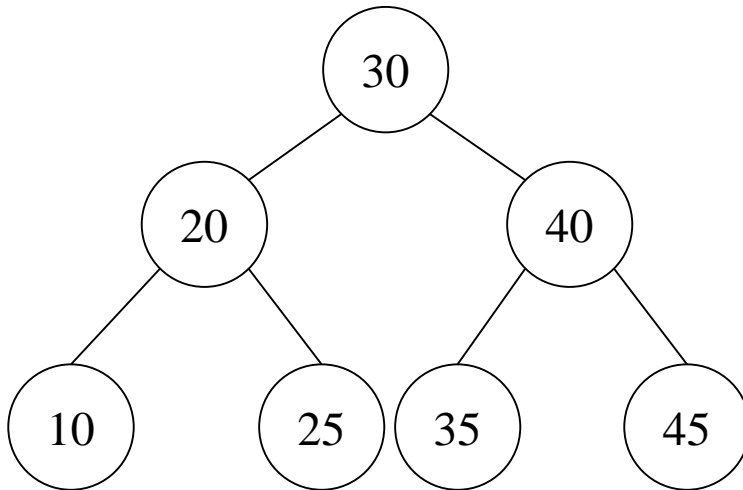
# 레코드, 키, 검색트리

- 레코드<sub>record</sub>
  - 개체에 대해 수집된 모든 정보를 포함하고 있는 저장 단위
  - e.g., 사람의 레코드
    - 주민번호, 이름, 집주소, 집 전화번호, 직장 전화번호, 휴대폰 번호, 최종 학력, 연소득, 가족 상황 등의 정보 포함
- 필드<sub>field</sub>
  - 레코드에서 각각의 정보를 나타내는 부분
  - e.g., 위 사람의 레코드에서 각각의 정보를 나타내는 부분
- 검색키<sub>search key</sub> 또는 키<sub>key</sub>
  - 다른 레코드와 중복되지 않도록 각 레코드를 대표할 수 있는 필드
  - 키는 하나의 필드로 이루어질 수도 있고, 두 개 이상의 필드로 이루어질 수도 있다
- 검색트리<sub>search tree</sub>
  - 각 노드가 규칙에 맞도록 하나씩의 키를 갖고 있다
  - 이를 통해 해당 레코드가 저장된 위치를 알 수 있다

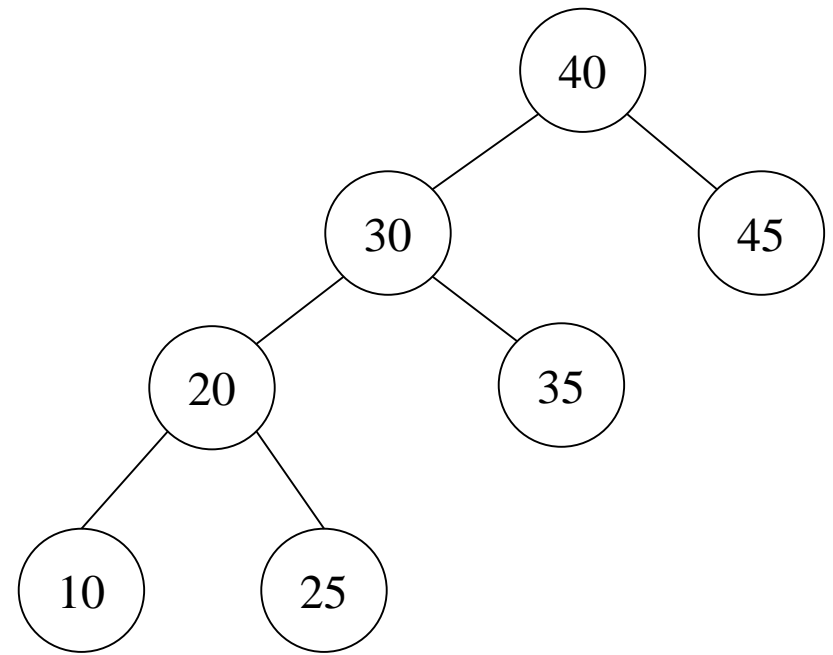
# 이진검색트리

- 이진검색트리의 각 노드는 키값을 하나씩 갖는다. 각 노드의 키값은 모두 달라야 한다.
- 최상위 레벨에 루트 노드가 있고, 각 노드는 최대 두 개의 자식을 갖는다.
- 임의의 노드의 키값은 자신의 왼쪽 자식 노드의 키값보다 크고, 오른쪽 자식의 키값보다 작다.

## 이진검색트리의 예



(a)



(b)

# 이진검색트리에서의 검색

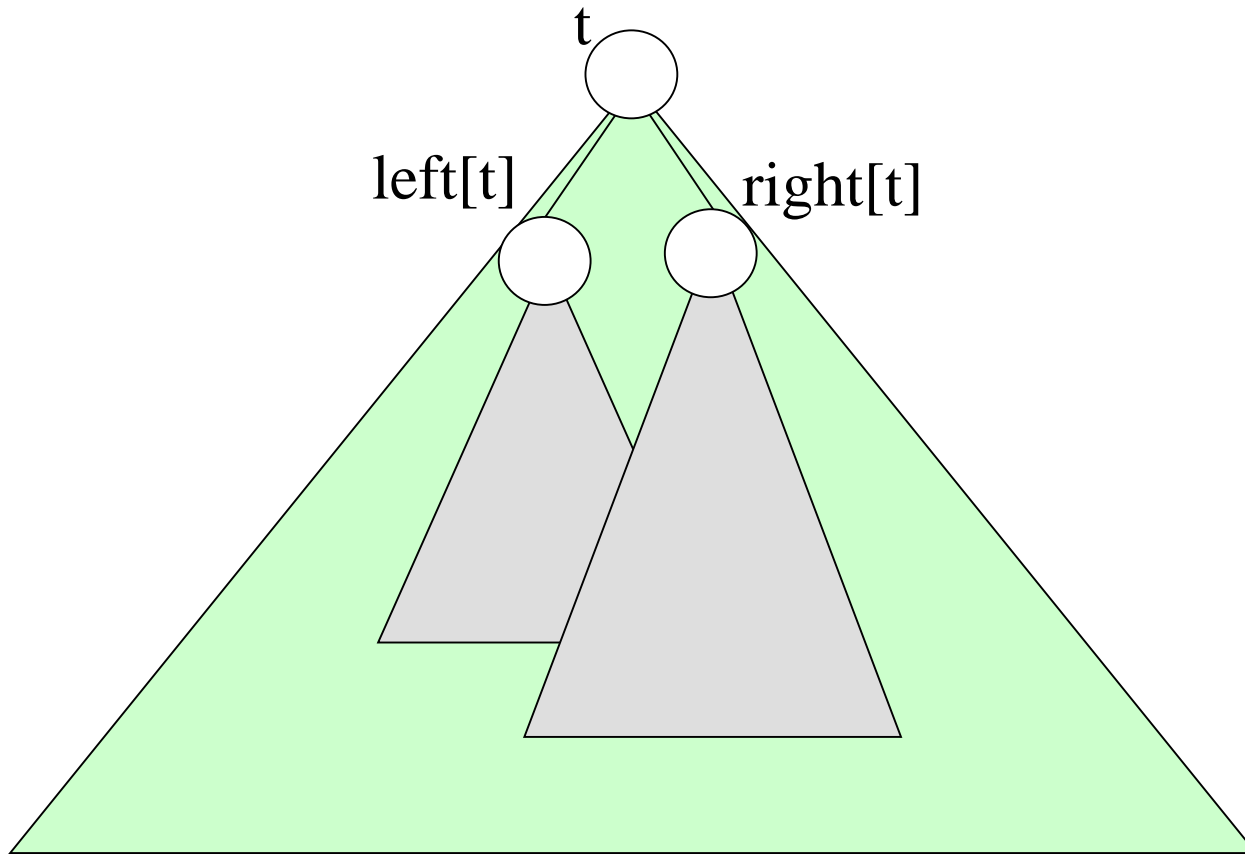
**treeSearch**( $t, x$ )

▷  $t$ : 트리의 루트 노드

▷  $x$ : 검색하고자 하는 키

```
{  
    if ( $t = \text{NIL}$  or  $\text{key}[t] = x$ ) then return  $t$ ;  
    if ( $x < \text{key}[t]$ )  
        then return treeSearch( $\text{left}[t], x$ );  
        else return treeSearch( $\text{right}[t], x$ );  
}
```

## 검색에서 재귀적 관점





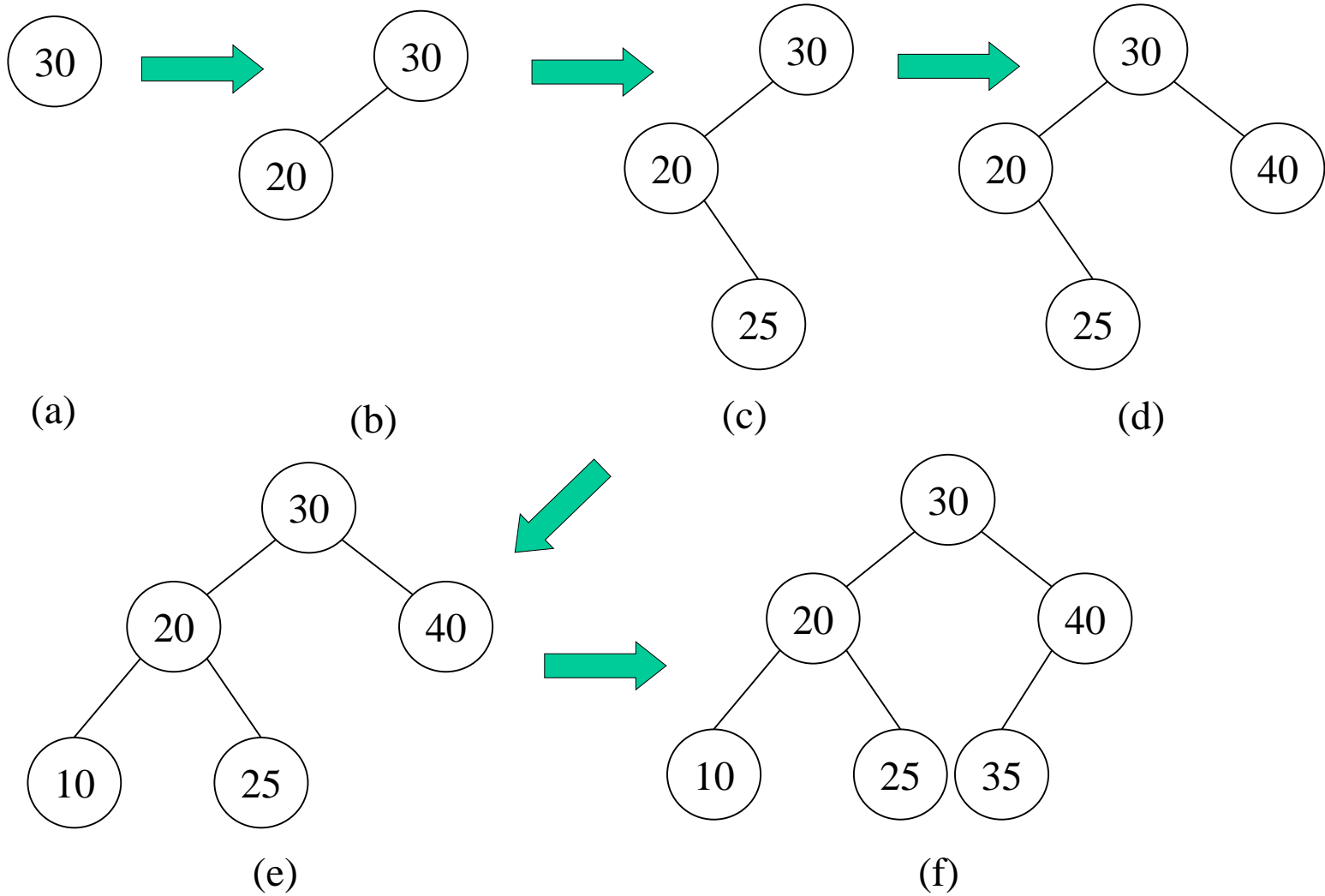
# 이진검색트리에서의 삽입

**treeInsert**( $t, x$ )

- ▷  $t$ : 트리의 루트 노드
- ▷  $x$ : 삽입하고자 하는 키
- ▷ 작업 완료 후 루트 노드의 포인터를 리턴한다

```
{  
    if ( $t = \text{NIL}$ ) then {  
         $\text{key}[r] \leftarrow x$ ;  $\text{left}[r] \leftarrow \text{NIL}$ ;  $\text{right}[r] \leftarrow \text{NIL}$ ;   ▷  $r$ : 새 노드  
        return  $r$ ;  
    }  
    if ( $x < \text{key}(t)$ )  
        then {  $\text{left}[t] \leftarrow \text{treeInsert}(\text{left}[t], x)$ ; return  $t$ ; }  
        else {  $\text{right}[t] \leftarrow \text{treeInsert}(\text{right}[t], x)$ ; return  $t$ ; }  
}
```

## 삽입의 예



# 이진검색트리에서의 삭제

$t$ : 트리의 루트 노드

$r$ : 삭제하고자 하는 노드

- 3가지 경우에 따라 다르게 처리한다
  - Case 1 :  $r$ 이 리프 노드인 경우
  - Case 2 :  $r$ 의 자식 노드가 하나인 경우
  - Case 3 :  $r$ 의 자식 노드가 두 개인 경우

# 이진검색트리에서의 삭제

Sketch-TreeDelete( $t, r$ )

▷  $t$ : 트리의 루트 노드

▷  $x$ : 삭제하고자 하는 키

{

**if** ( $r$ 이 리프 노드) **then**

▷ Case 1

        그냥  $r$ 을 버린다;

**else if** ( $r$ 의 자식이 하나만 있음) **then**

▷ Case 2

$r$ 의 부모가  $r$ 의 자식을 직접 가리키도록 한다;

**else**

▷ Case 3

$r$ 의 오른쪽 서브트리의 최소원소 노드  $s$ 를 삭제하고,  
         $s$ 를  $r$  자리에 놓는다;

}

# 이진검색트리에서의 삭제

$t$ : 트리의 루트 노드  
 $r$ : 삭제하고자 하는 노드  
 $p$ :  $r$ 의 부모 노드

```
treeDelete(t, r, p)
```

```
{
```

```
    if (r = t) then root ← deleteNode(t);
```

```
    else if (r = left[p])
```

```
        then left[p] ← deleteNode(r);
```

```
        else right[p] ← deleteNode(r);
```

```
}
```

```
deleteNode(r)
```

```
{
```

```
    if (left[r] = right[r] = NIL) then return NIL;
```

```
    else if (left[r] = NIL and right[r] ≠ NIL) then return right[r];
```

```
    else if (left[r] ≠ NIL and right[r] = NIL) then return left[r];
```

```
    else {
```

```
        s ← right[r];
```

```
        while (left[s] ≠ NIL)
```

```
            {parent ← s; s ← left[s];}
```

```
        key[r] ← key[s];
```

```
        if (s = right[r]) then right[r] ← right[s];
```

```
            else left[parent] ← right[s];
```

```
        return r;
```

```
    }
```

```
}
```

▷  $r$ 이 루트 노드인 경우

▷  $r$ 이 루트가 아닌 경우

▷  $r$ 이  $p$ 의 왼쪽 자식

▷  $r$ 이  $p$ 의 오른쪽 자식

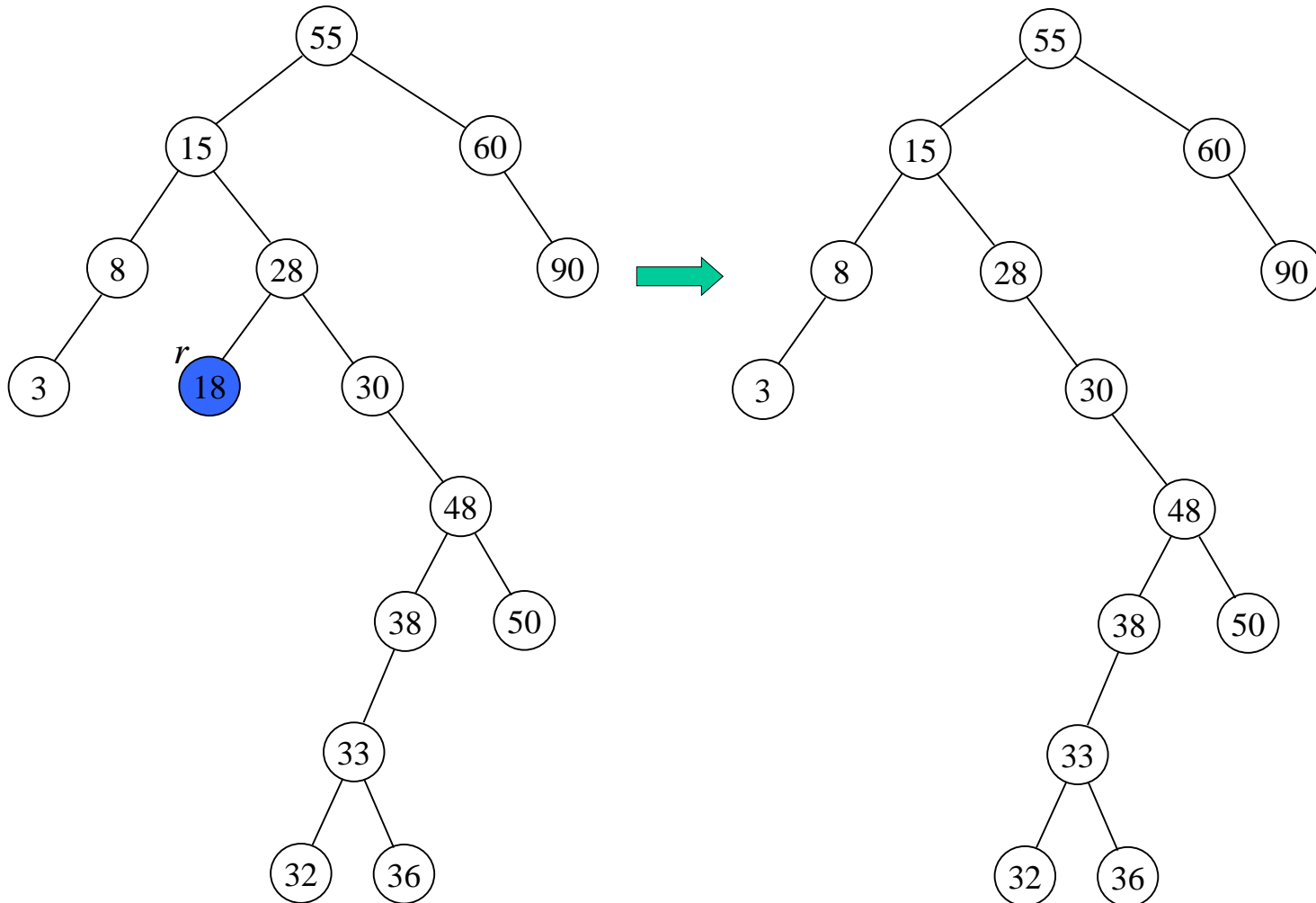
▷ Case 1

▷ Case 2-1

▷ Case 2-2

▷ Case 3

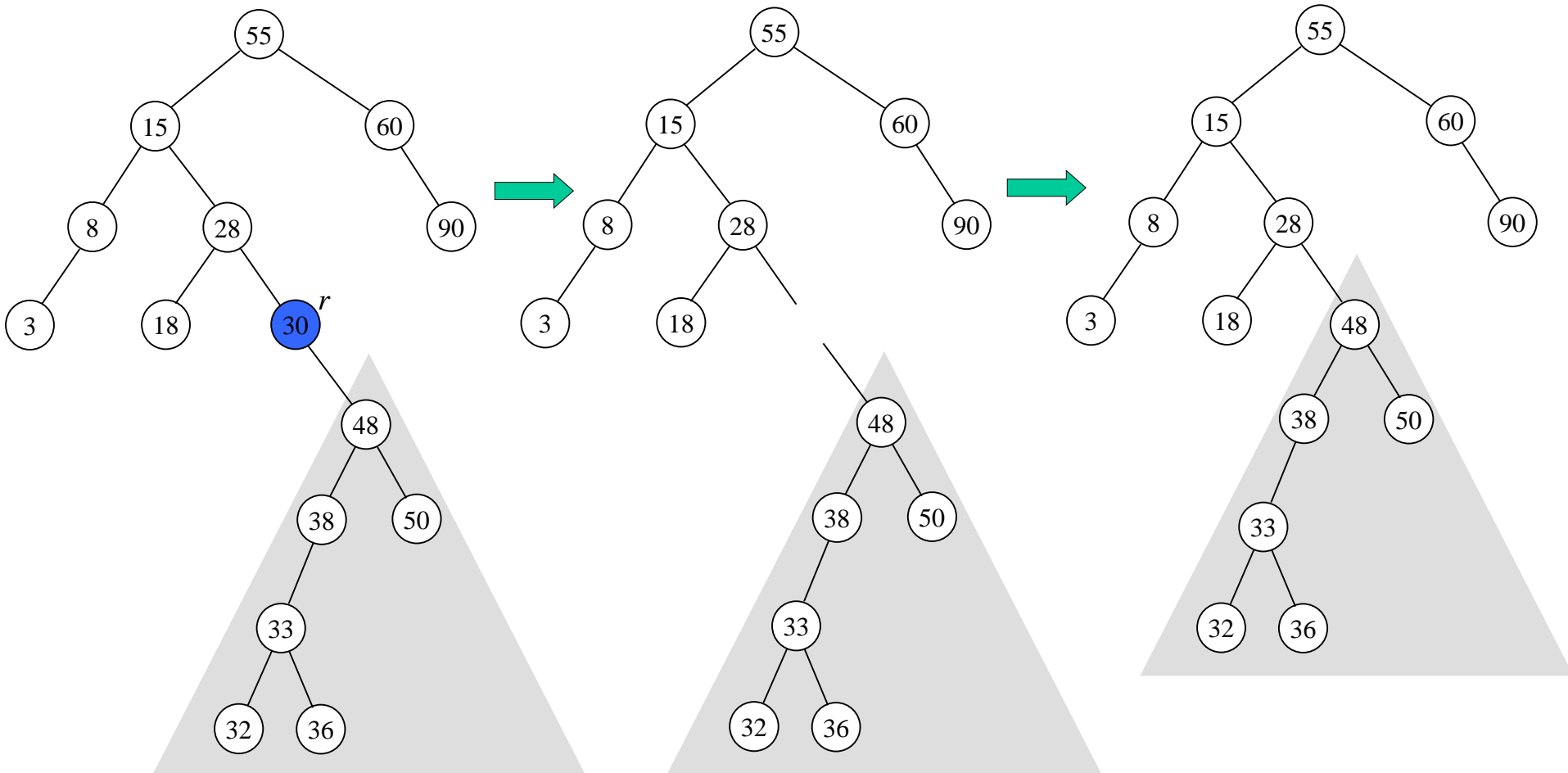
## 삭제의 예: Case 1



(a)  $r$ 의 자식이 없음

(b) 단순히  $r$ 을 제거한다

## 삭제의 예: Case 2

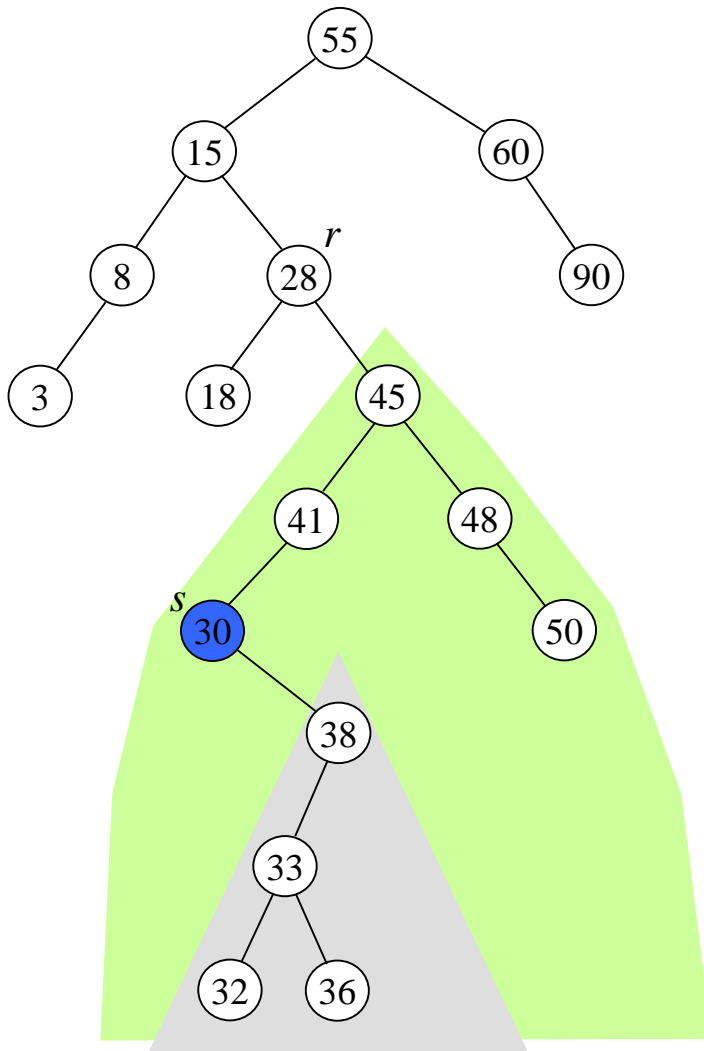


(a)  $r$ 의 자식이 하나뿐임

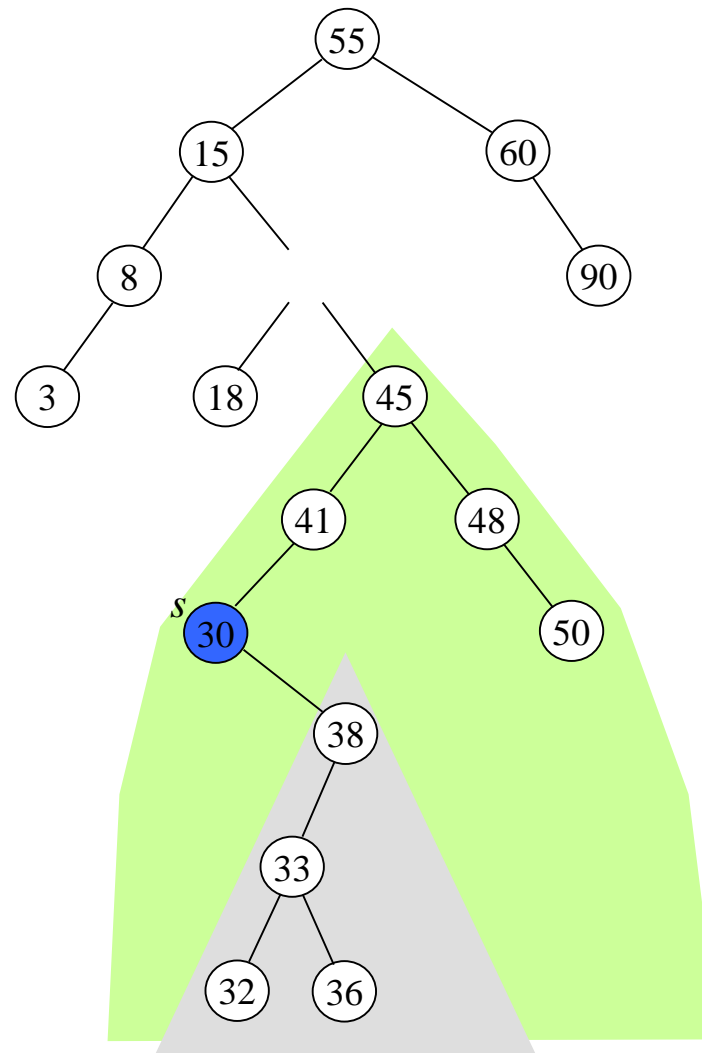
(b)  $r$ 을 제거

(c)  $r$  자리에  $r$ 의 자식을 놓는다

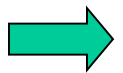
## 삭제의 예: Case 3



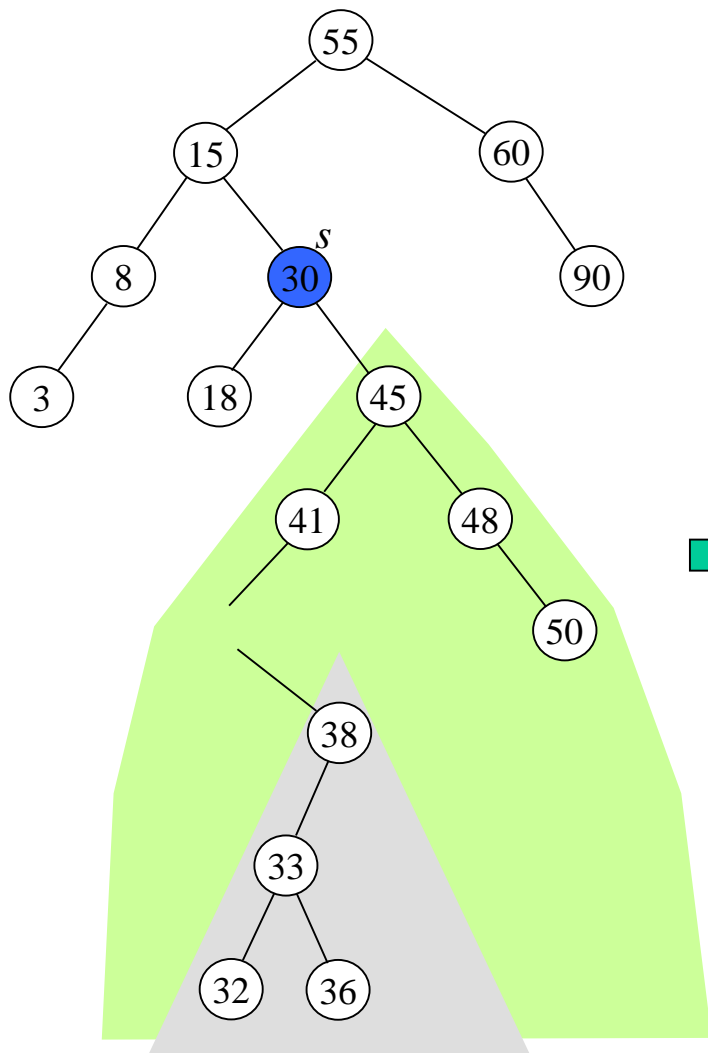
(a)  $r$ 의 직후원소  $s$ 를 찾는다



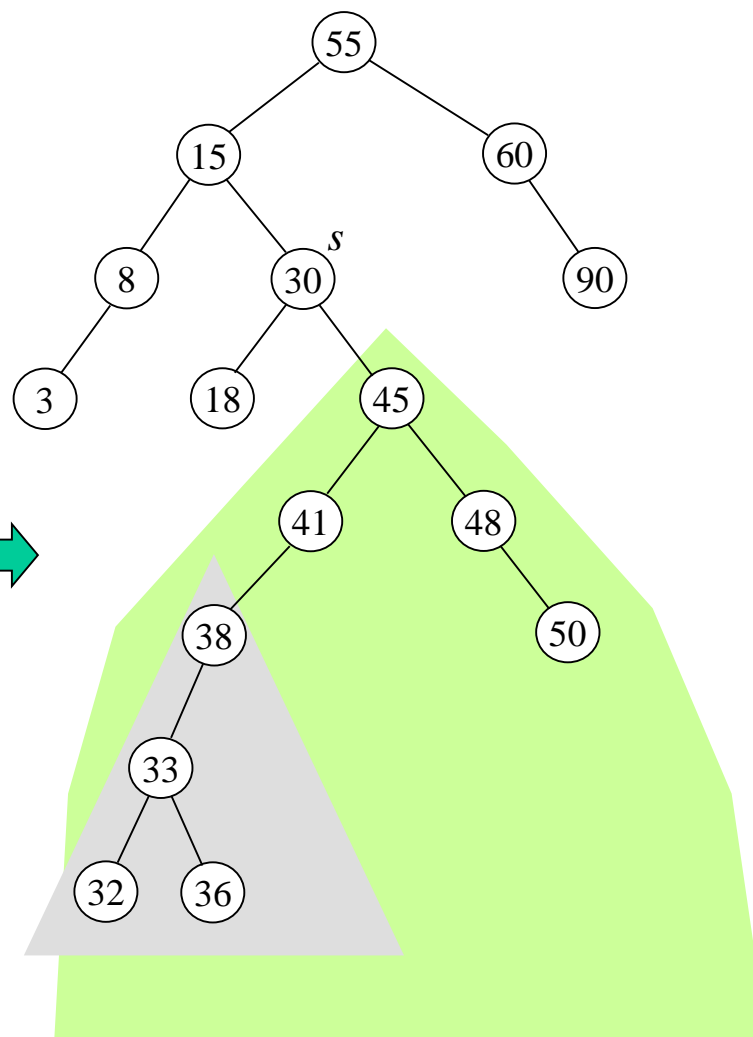
(b)  $r$ 을 없앤다







(c)  $s$ 를  $r$ 자리로 옮긴다

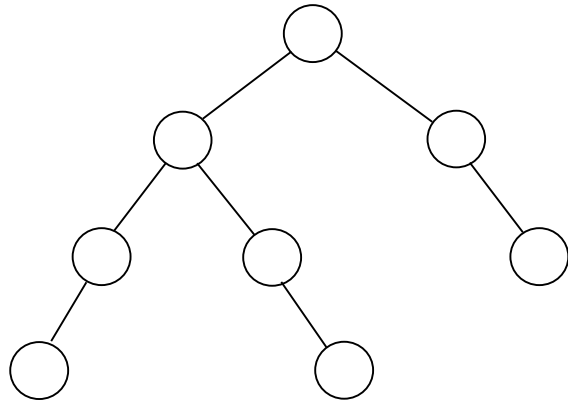


(d)  $s$ 가 있던 자리에  $s$ 의 자식을 놓는다

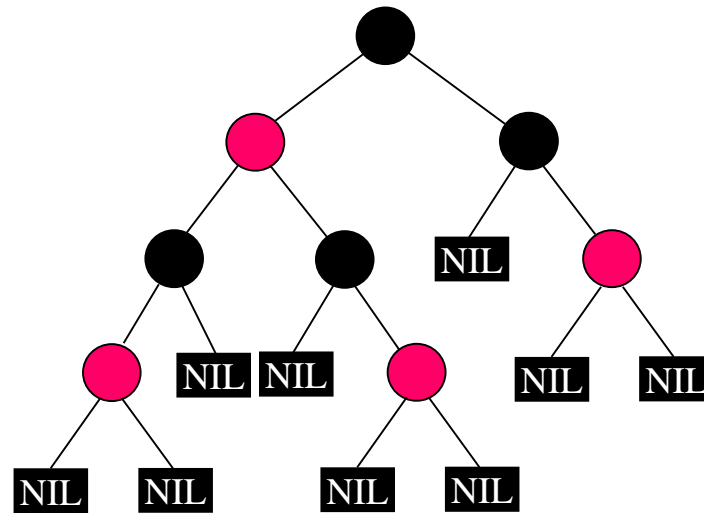
# 레드블랙트리

- 이진검색트리의 모든 노드에 블랙 또는 레드의 색을 칠하되 다음의 **레드 블랙특성**을 만족해야 한다
  - ① 루트는 블랙이다
  - ② 모든 리프는 블랙이다
  - ③ 노드가 레드이면 그 노드의 자식은 반드시 블랙이다
  - ④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다
- ✓ 여기서 리프 노드는 일반적인 의미의 리프 노드와 다르다.  
모든 NIL 포인터가 NIL이라는 리프 노드를 가리킨다고 가정한다.

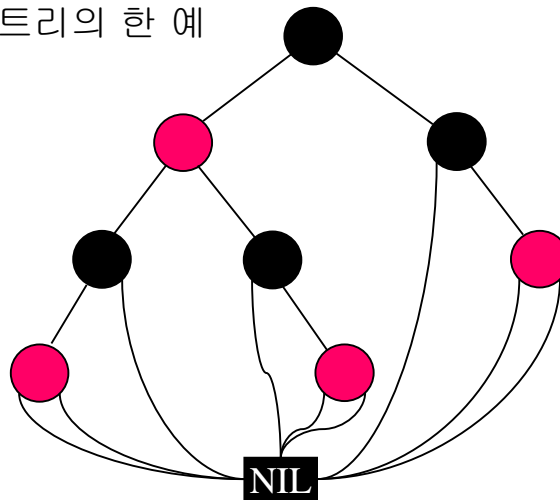
## 이진검색트리를 레드블랙트리로 만든 예



(a) 이진검색트리의 한 예



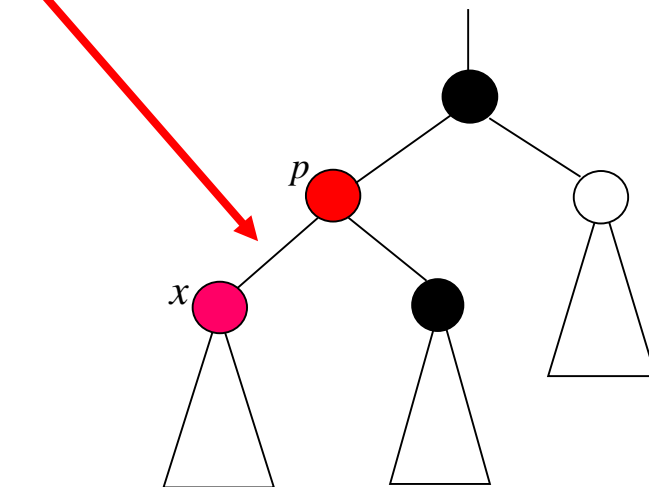
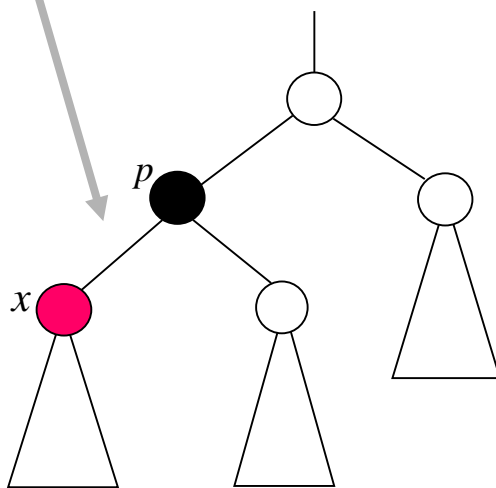
(b) (a)를 레드블랙트리로 만든 예



(c) 실제 구현시의 NIL 노드 처리 방법

# 레드블랙트리에서의 삽입

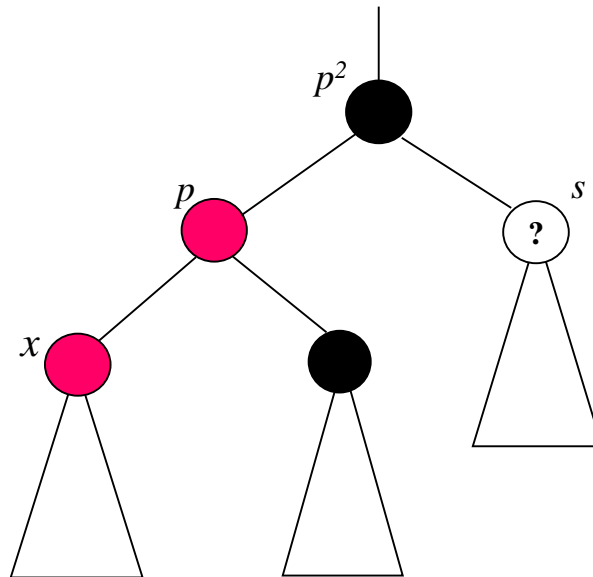
- 이진검색트리에서의 삽입과 같다. 다만 삽입 후 삽입된 노드를 레드로 칠한다. (이 노드를  $x$ 라 하자)
- 만일  $x$ 의 부모 노드  $p$ 의 색상이
  - 블랙이면 아무 문제 없다.
  - 레드이면 레드블랙특성 ③이 깨진다.



✓ 그러므로  $p$ 가 레드인 경우만 고려하면 된다

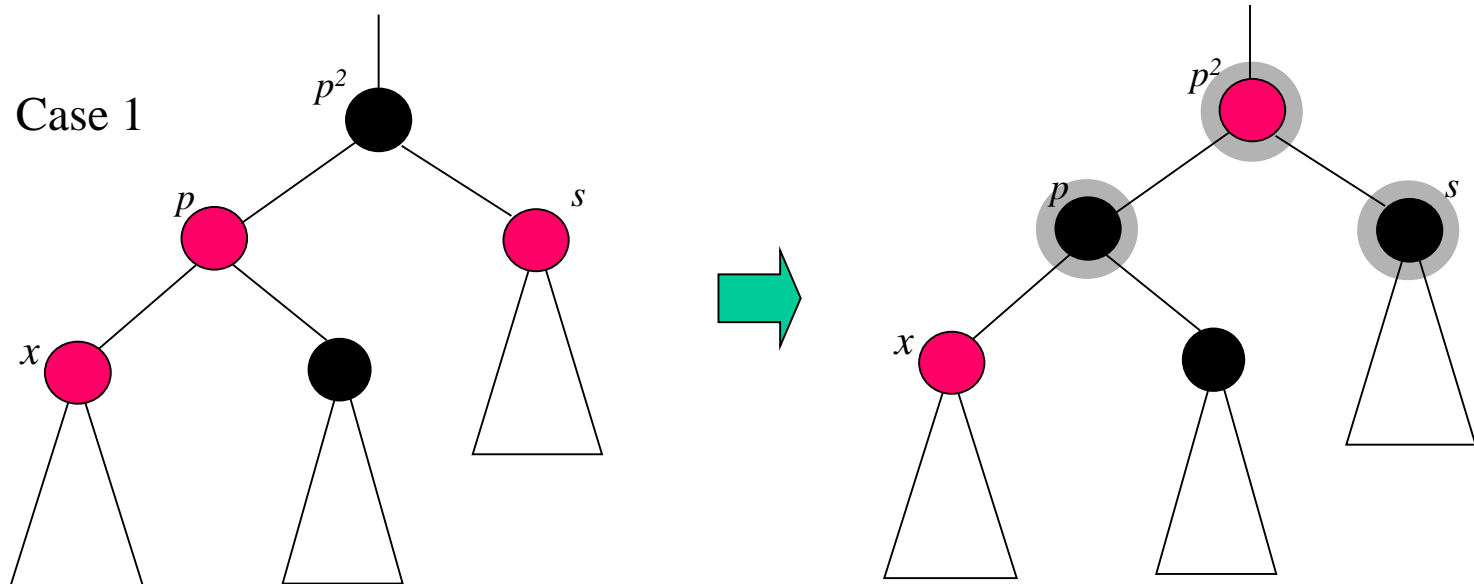
## 레드블랙트리에서의 삽입

- $p^2$ 와  $x$ 의 형제 노드는 반드시 블랙이다
- $s$ 의 색상에 따라 두 가지로 나눈다
  - Case 1:  $s$ 가 레드
  - Case 2:  $s$ 가 블랙



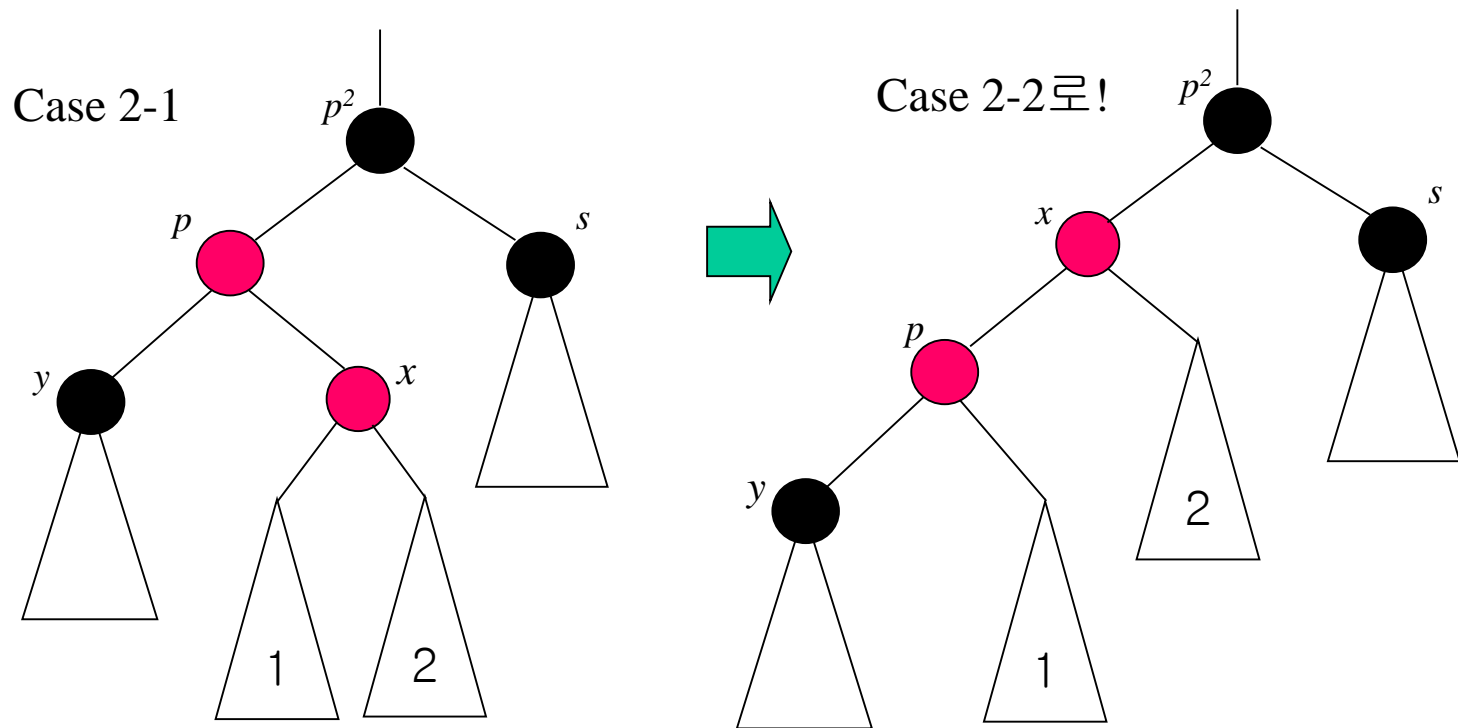
## Case 1: $s$ 가 레드

● : 색상이 바뀐 노드



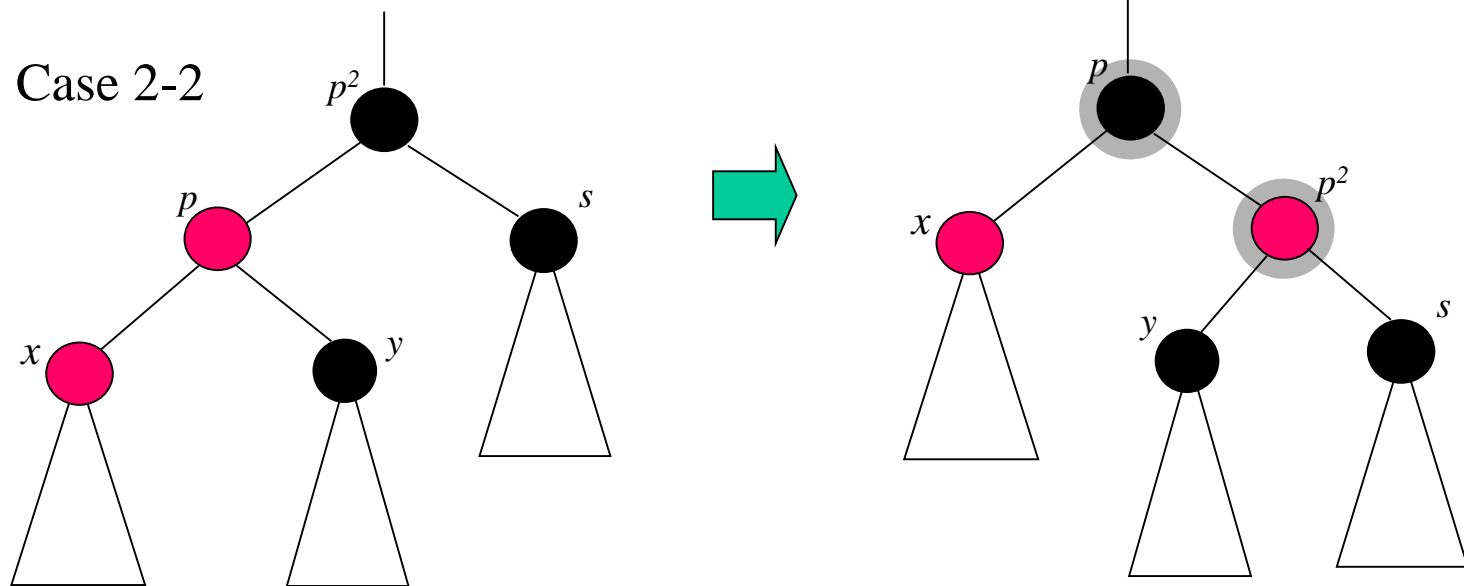
✓  $p^2$ 에서 방금과 같은 문제가 발생할 수 있다

Case 2-1:  $s$ 가 블랙이고,  $x$ 가  $p$ 의 오른쪽 자식



## Case 2-2: $s$ 가 블랙이고, $x$ 가 $p$ 의 왼쪽 자식

● : 색상이 바뀐 노드

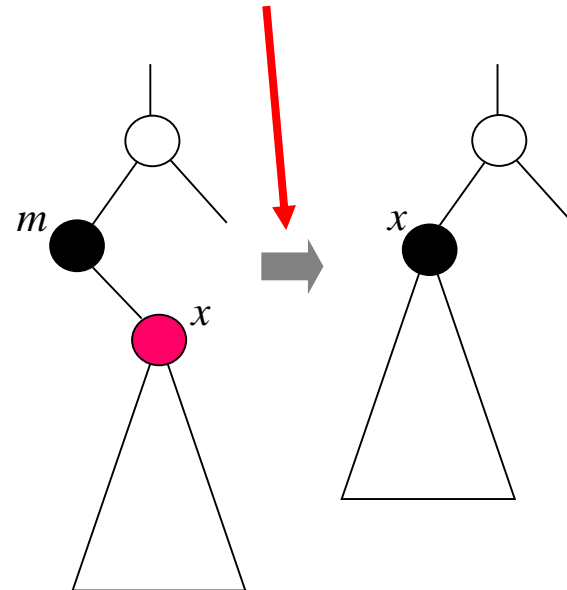
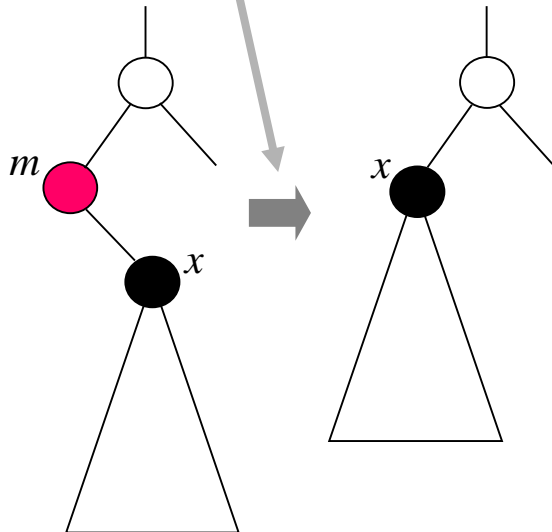


✓ 삽입 완료!

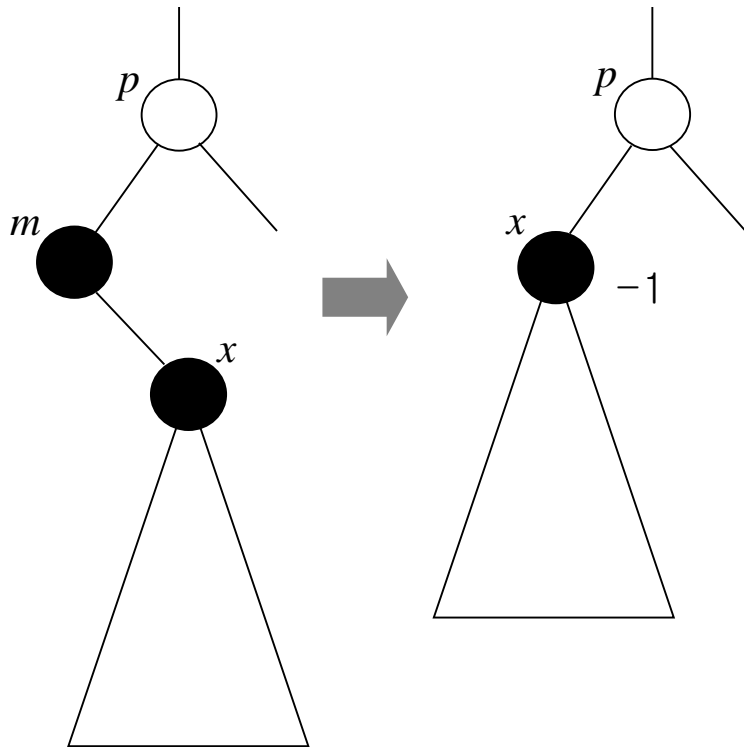


# 레드블랙트리에서의 삭제

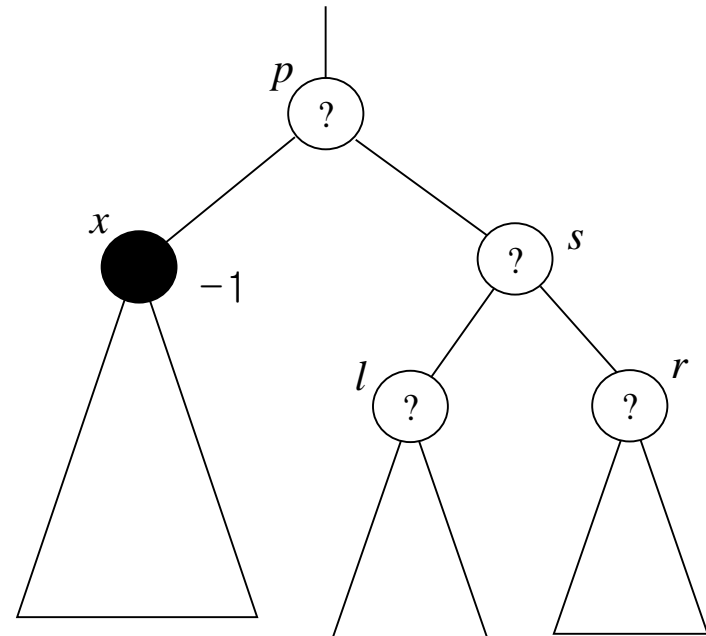
- 삭제 노드의 자식이 없거나 1개만을 가진 노드로 제한해도 된다
  - 이유는 텍스트의 p.146의 첫 문단 참조
  - 삭제 노드를  $m$ 이라 하자
- 삭제 노드가 레드이면 아무 문제 없다
- 삭제 노드가 블랙이라도 (유일한) 자식이 레드이면 문제 없다



✓  $x$  옆의  $-1$ 은 루트에서  $x$ 를 통해 리프에 이르는  
경로에서 블랙 노드의 수가 하나 모자람을 의미한다.

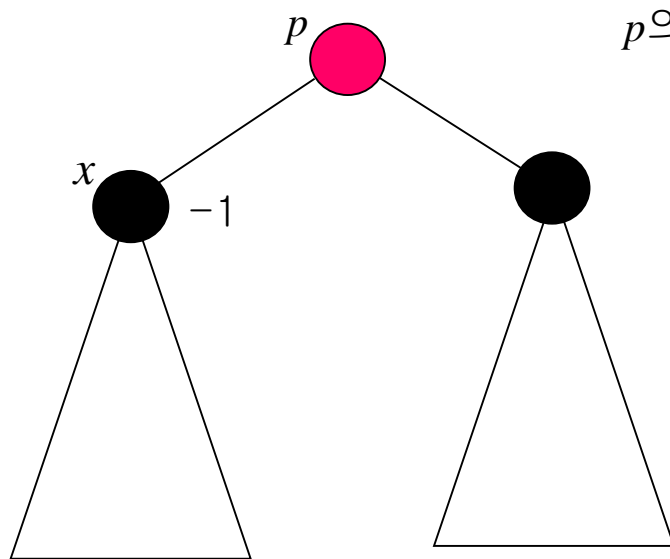


$m$  삭제 후 문제 발생  
(레드블랙특성 ④ 위반)



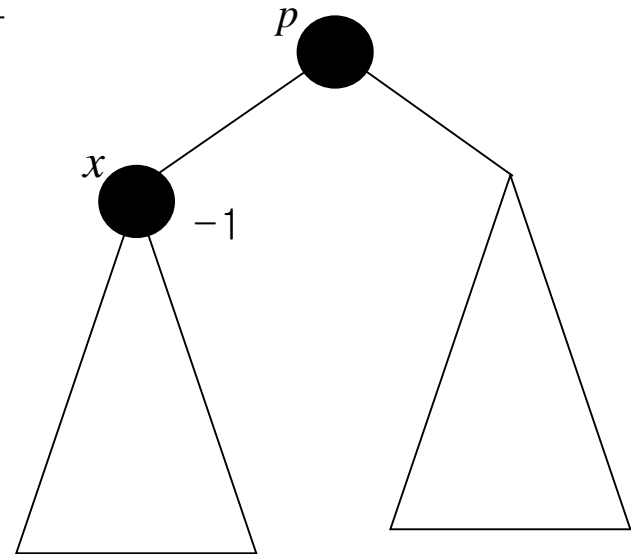
$x$ 의 주변 상황에 따라 처리 방법이 달라진다

## 경우의 수 나누기

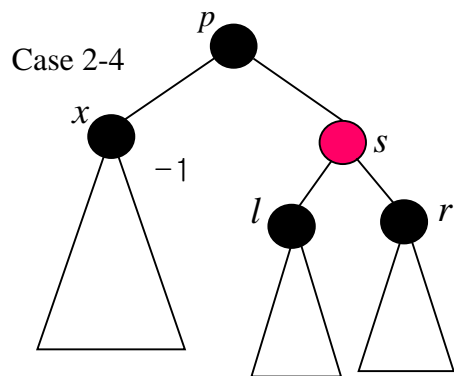
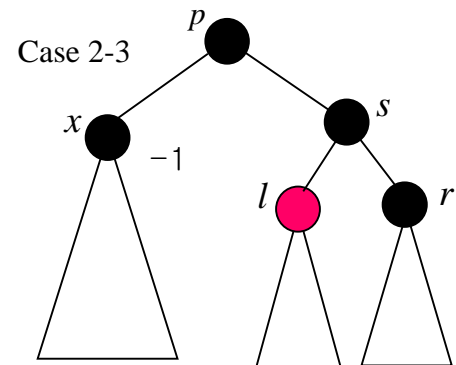
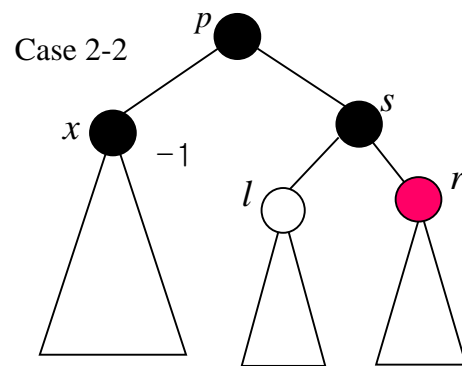
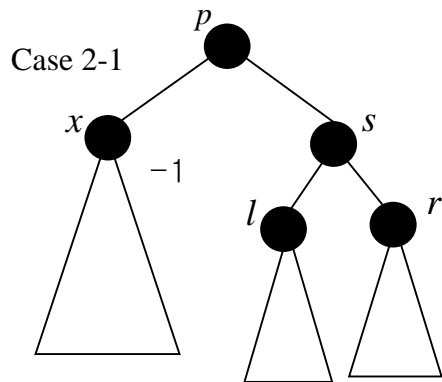
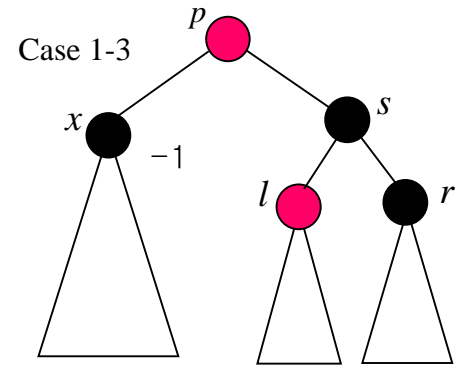
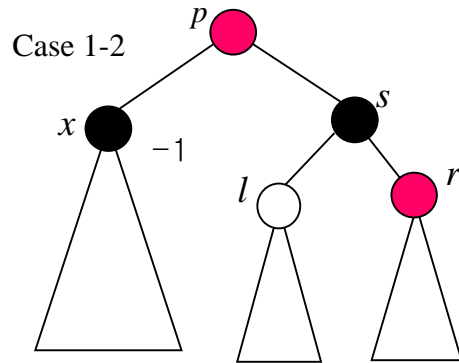
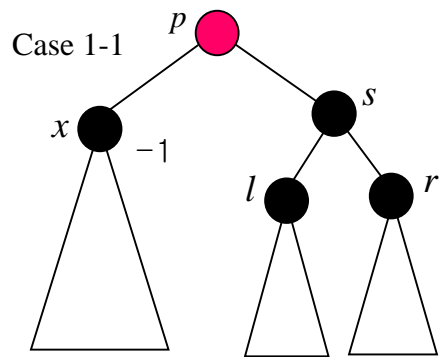


Case 1

$p$ 의 색상에 따라

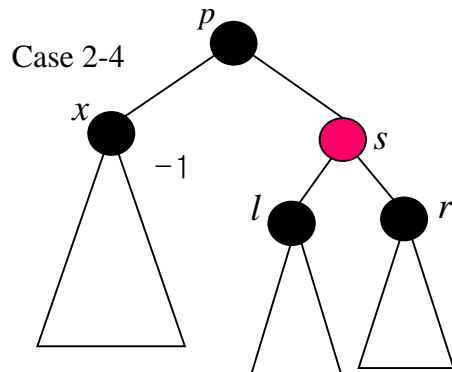
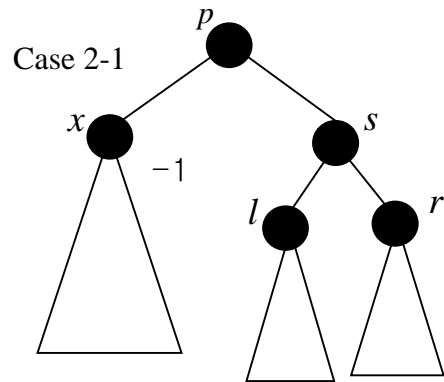
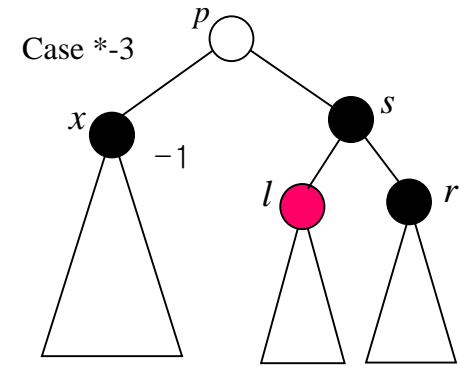
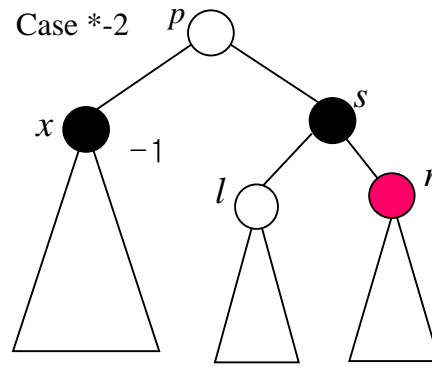
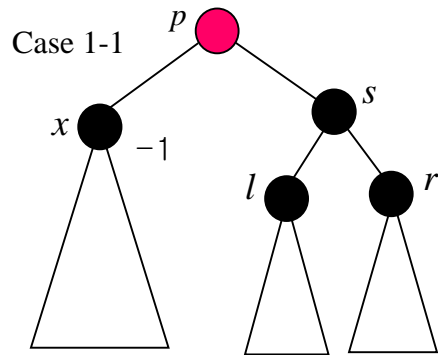


Case 2



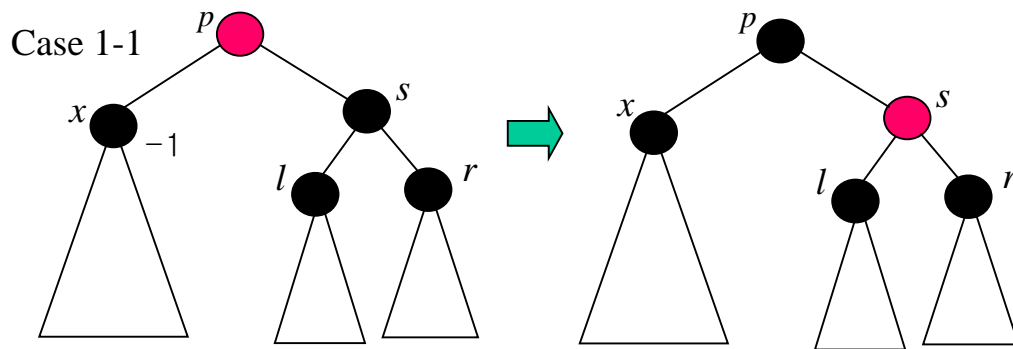
$s$ 의 색상에 따라

$p$ 의 색상에 따라

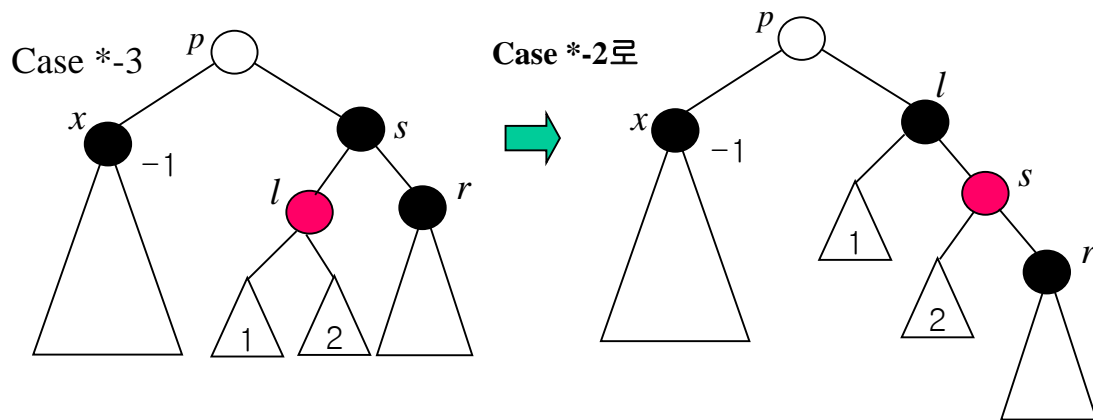
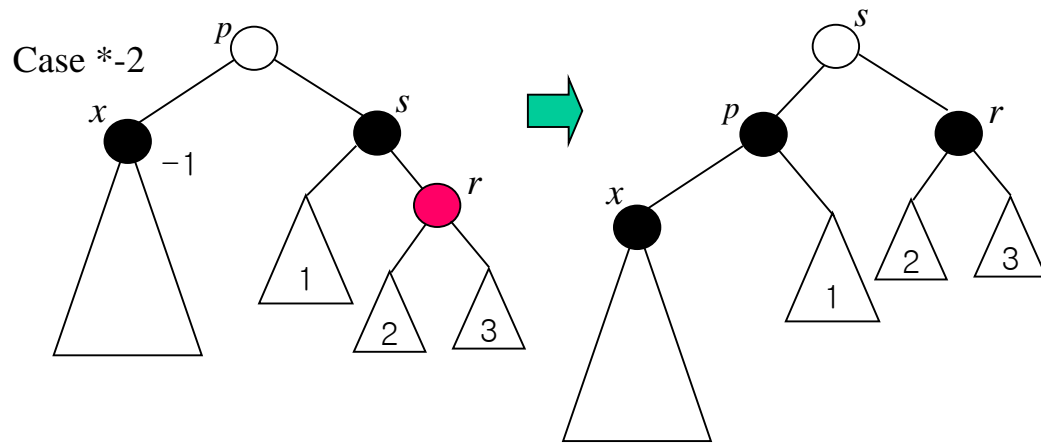


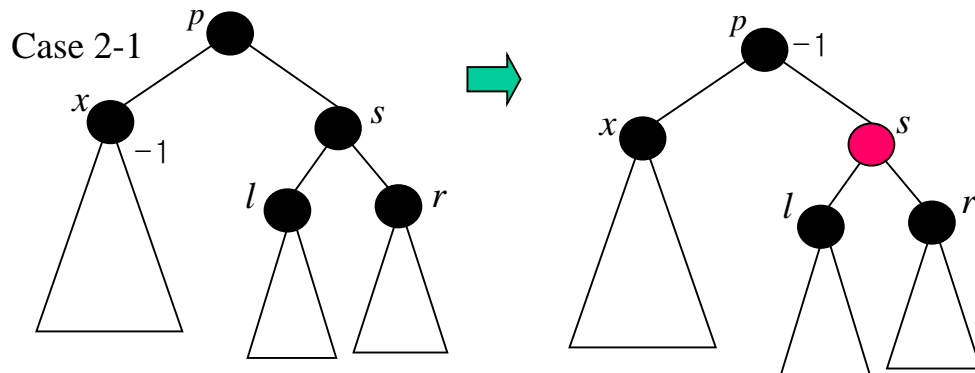
✓ 최종적으로 5가지 경우로 나뉜다

## 각 경우에 따른 처리

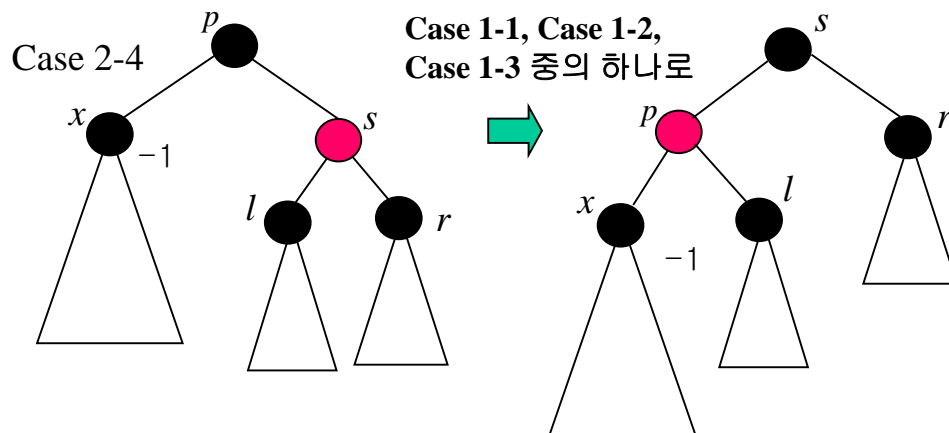


✓ 삭제 완료!





✓  $p$ 에서 방금과 같은 문제가 발생. 재귀적으로 처리.

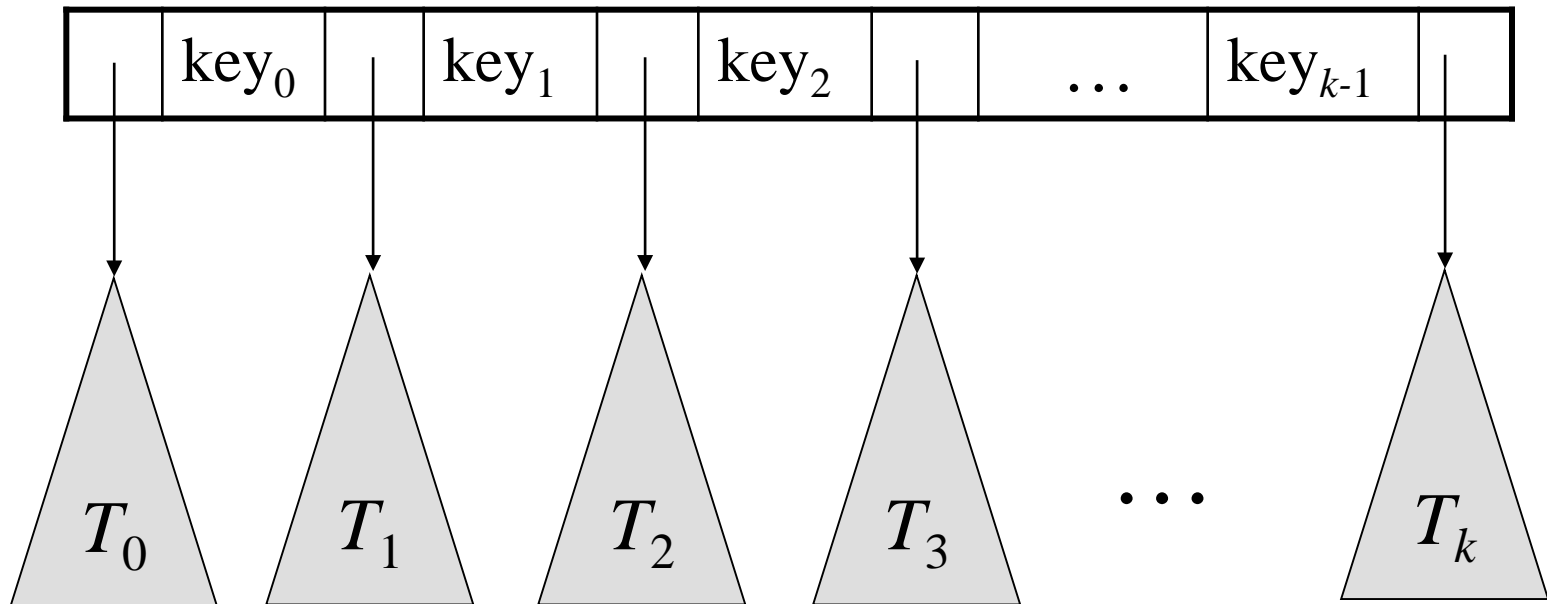




# B-트리

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

# 다진검색트리

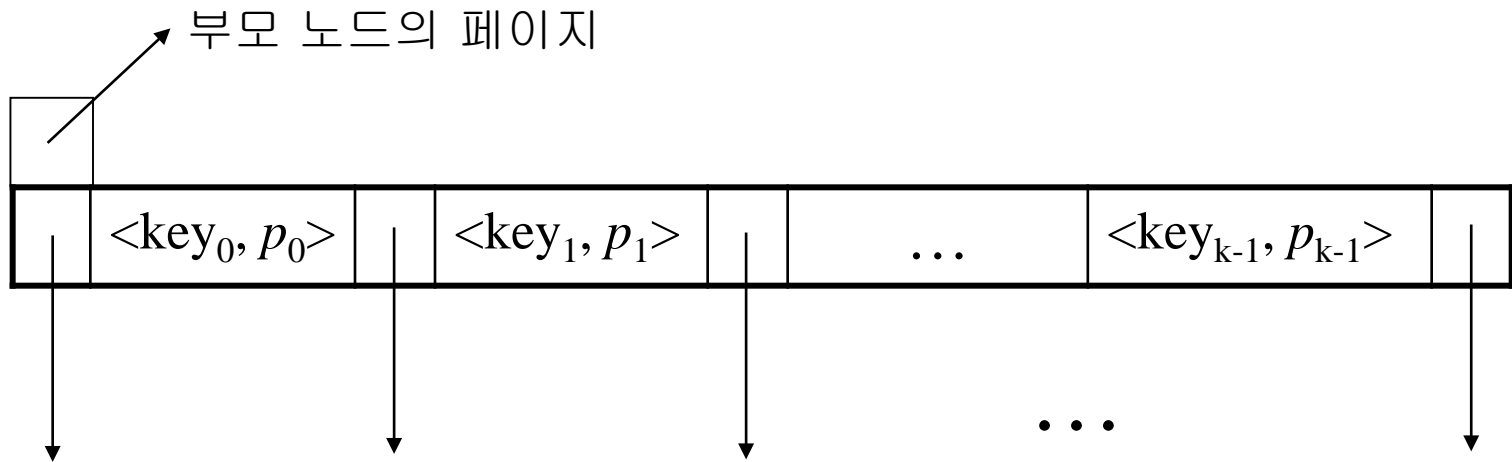


$$key_{i-1} < \triangle T_i < key_i$$

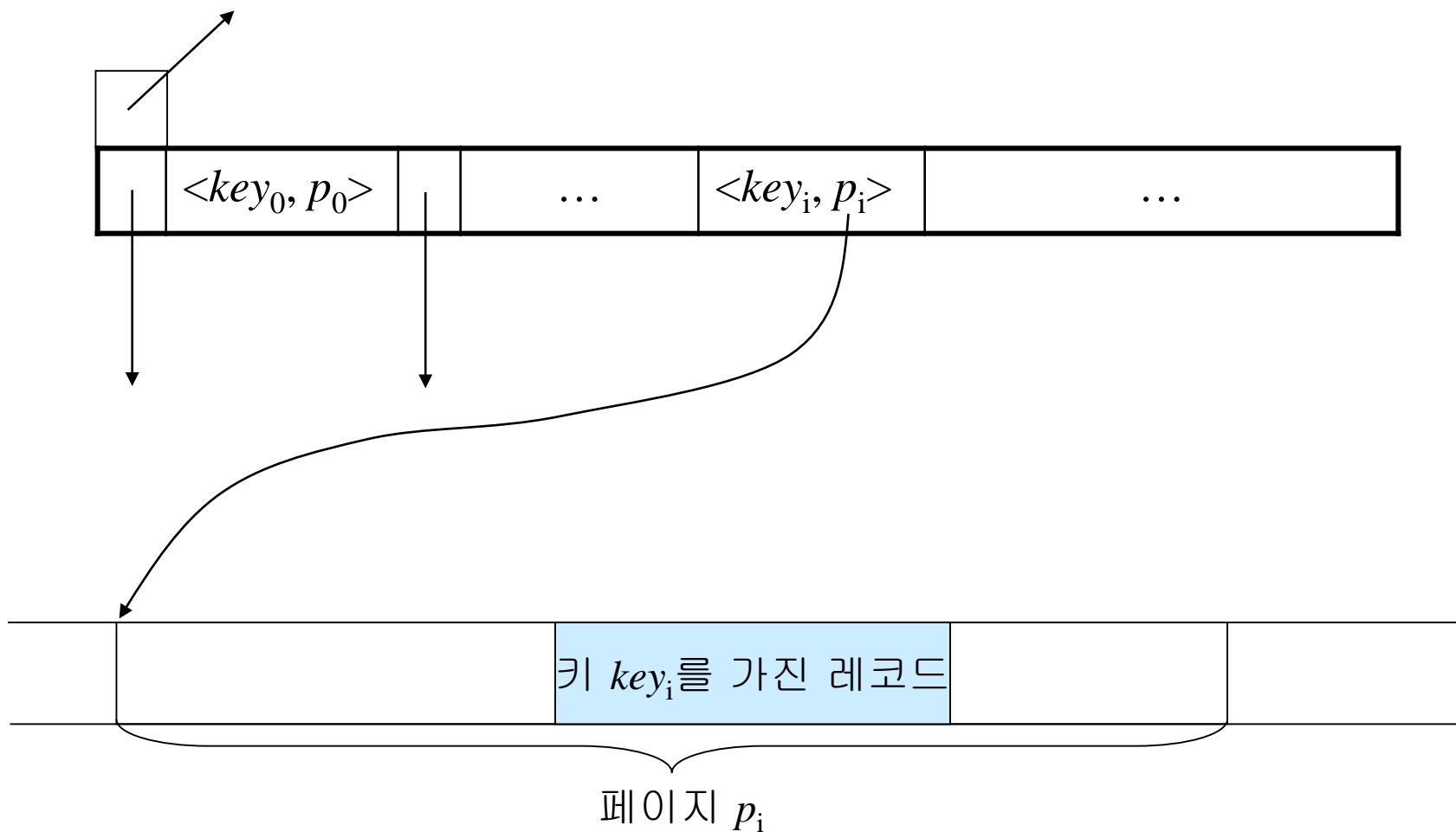
# B-트리

- B-트리는 균형잡힌 다진검색트리로 다음의 성질을 만족한다
  - 루트를 제외한 모든 노드는  $\lfloor k/2 \rfloor \sim k$  개의 키를 갖는다
  - 모든 리프 노드는 같은 깊이를 가진다

## B-트리의 노드 구조



## B-트리를 통해 레코드에 접근하는 과정



## B-트리에서의 삽입

```
BTreeInsert( $t, x$ )  
{
```

- ▷  $t$ : 트리의 루트 노드
- ▷  $x$ : 삽입하고자 하는 키

```
     $x$ 를 삽입할 리프 노드  $r$ 을 찾는다;  
     $x$ 를  $r$ 에 삽입한다;
```

```
    if ( $r$ 에 오버플로우 발생) then clearOverflow( $r$ );
```

```
}  
clearOverflow( $r$ )  
{
```

```
    if ( $r$ 의 형제 노드 중 여유가 있는 노드가 있음) then { $r$ 의 남은 키를 넘긴다};
```

```
    else {
```

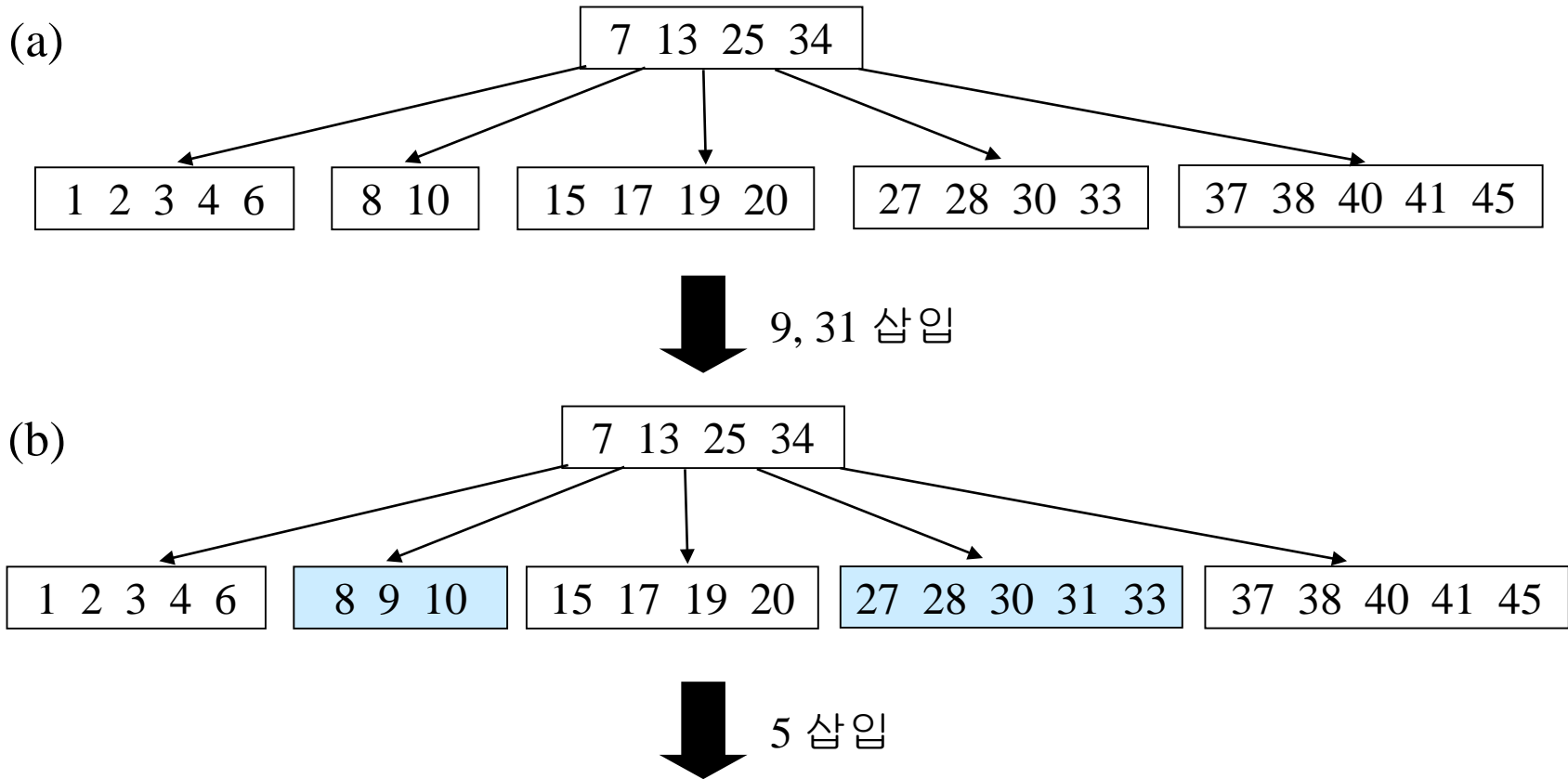
```
         $r$ 을 둘로 분할하고 가운데 키를 부모 노드로 넘긴다;
```

```
        if (부모 노드  $p$ 에 오버플로우 발생) then clearOverflow( $p$ );
```

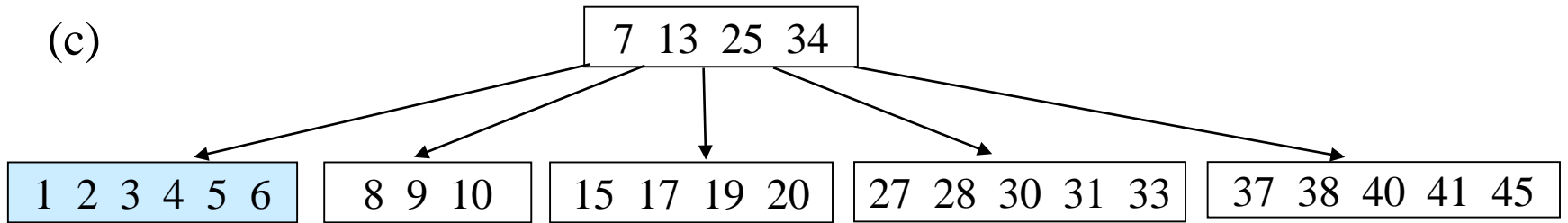
```
    }
```

```
}
```

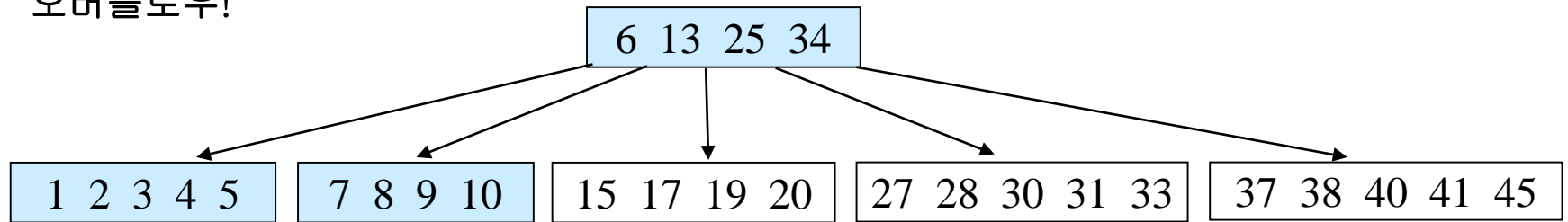
## B-트리에서 삽입의 예



(c)



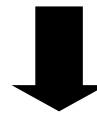
오버플로우!



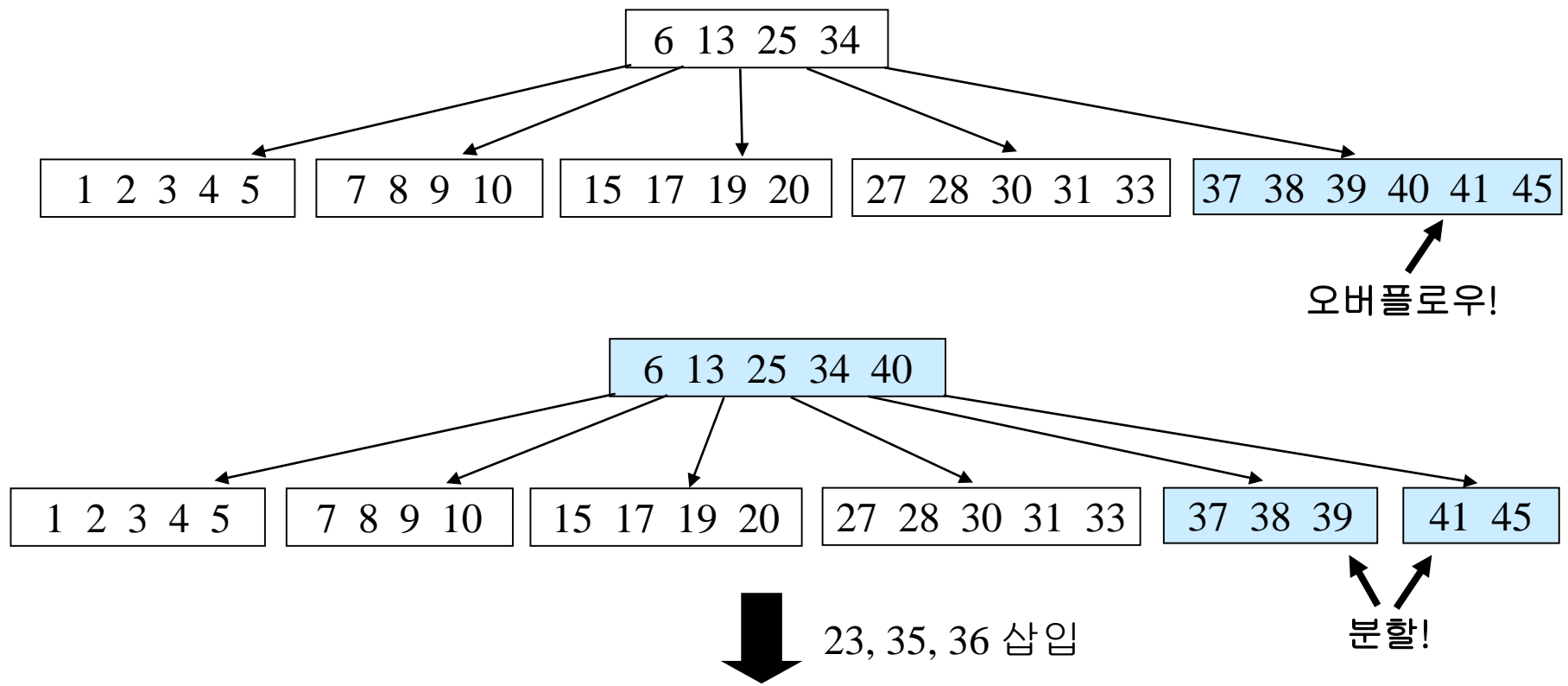
39 삽입



(d)



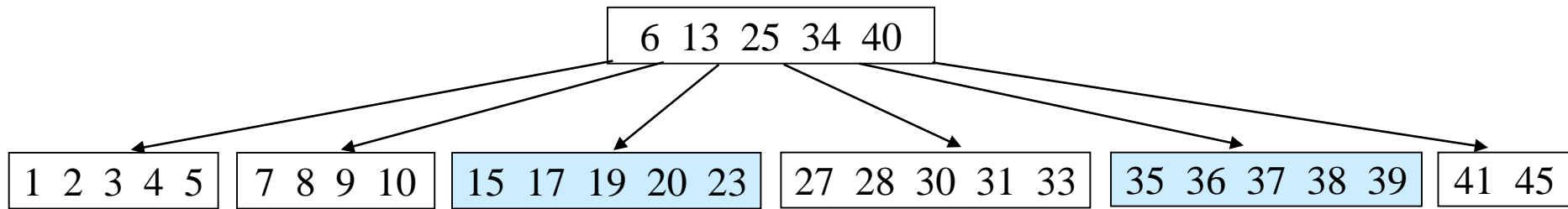
39 삽입



(e)



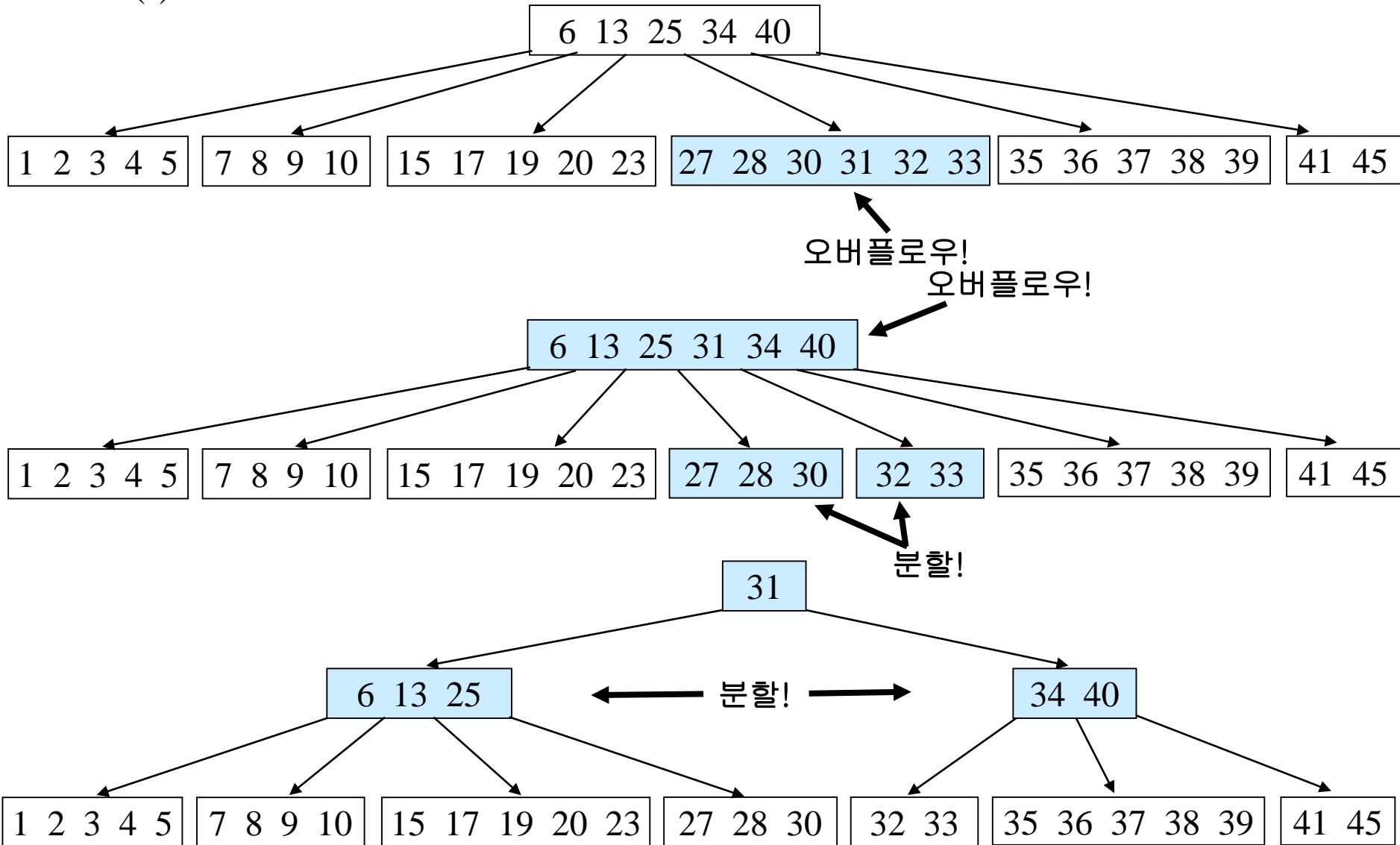
23, 35, 36 삽입



32 삽입

(f)

32 삽입



# B-트리에서의 삭제

BTreeDelete( $t, x, v$ )

{

**if** ( $v$ 가 리프 노드 아님) **then** {  
         $x$ 의 직후원소  $y$ 를 가진 리프 노드를 찾는다;  
         $x$ 와  $y$ 를 맞바꾼다;

    }

    리프 노드에서  $x$ 를 제거하고 이 리프 노드를  $r$ 이라 한다;

**if** ( $r$ 에서 언더플로우 발생) **then** clearUnderflow( $r$ );

}

clearUnderflow( $r$ )

{

**if** ( $r$ 의 형제 노드 중 키를 하나 내놓을 수 있는 여분을 가진 노드가 있음)

**then** {  $r$ 이 키를 넘겨받는다; }

**else** {

$r$ 의 형제 노드와  $r$ 을 합병한다;

**if** (부모 노드  $p$ 에 언더플로우 발생) **then** clearUnderflow( $p$ );

    }

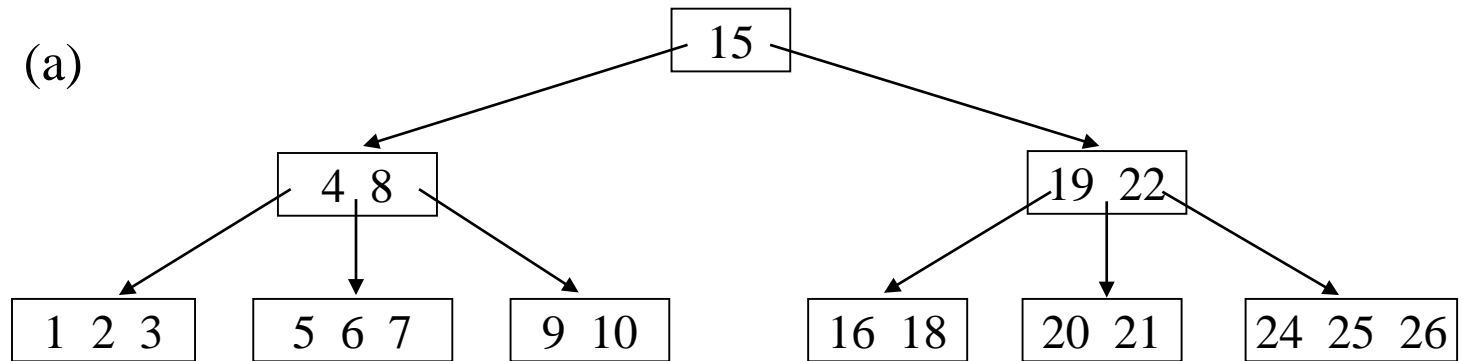
}

▷  $t$ : 트리의 루트 노드

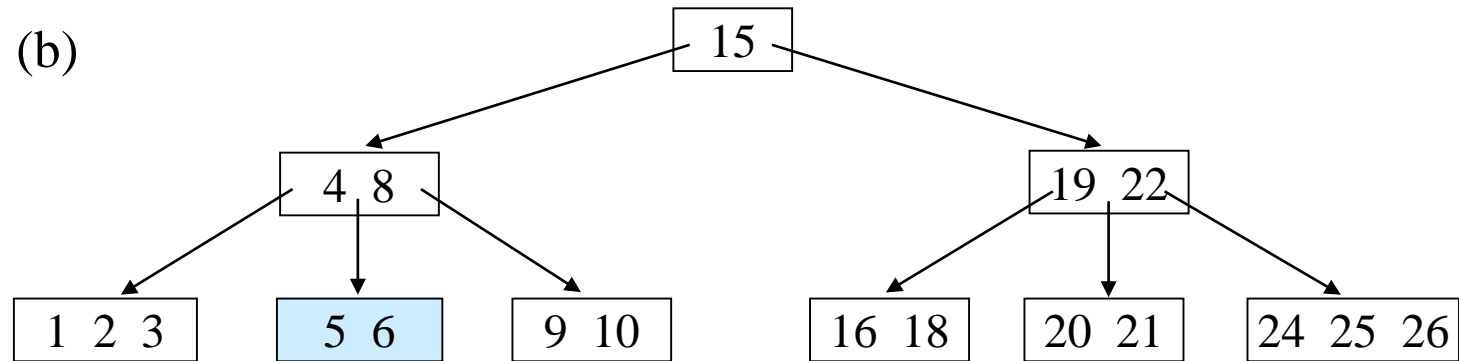
▷  $x$ : 삭제하고자 하는 키

▷  $v$ :  $x$ 를 갖고 있는 노드

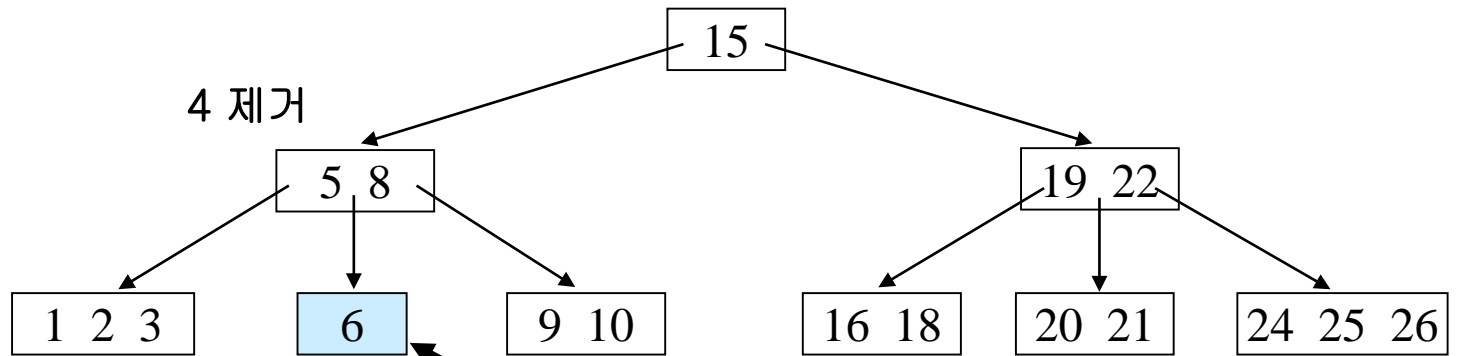
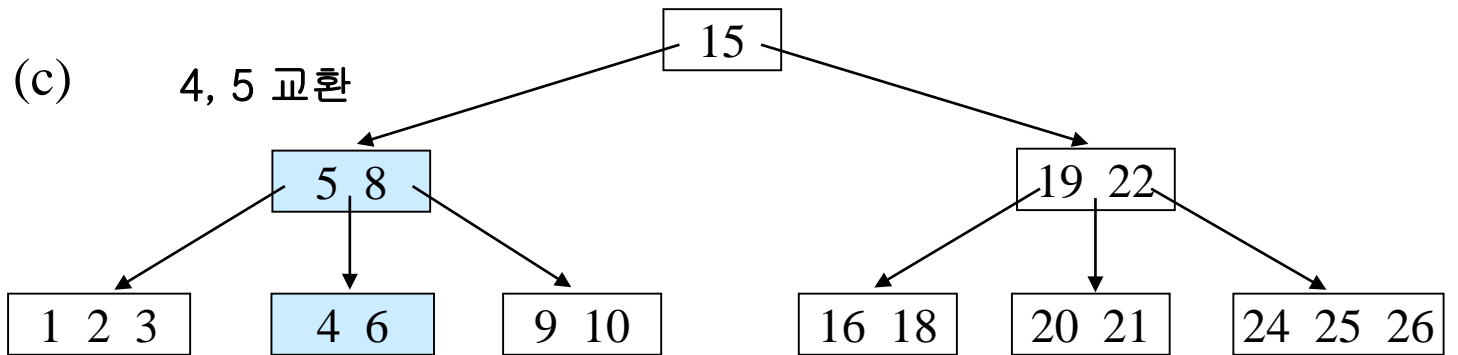
## B-트리에서 삭제의 예



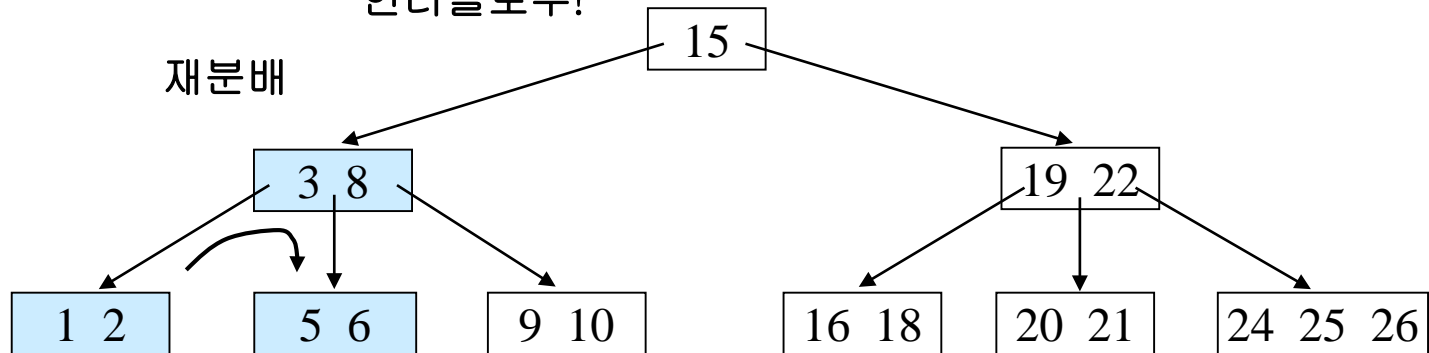
7 삭제



4 삭제

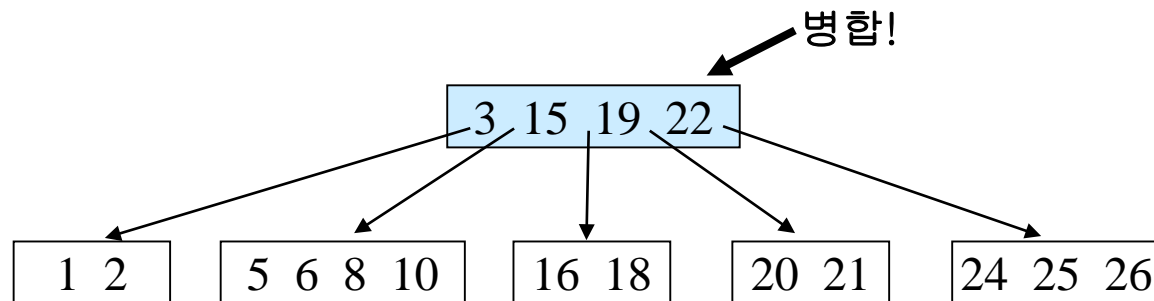
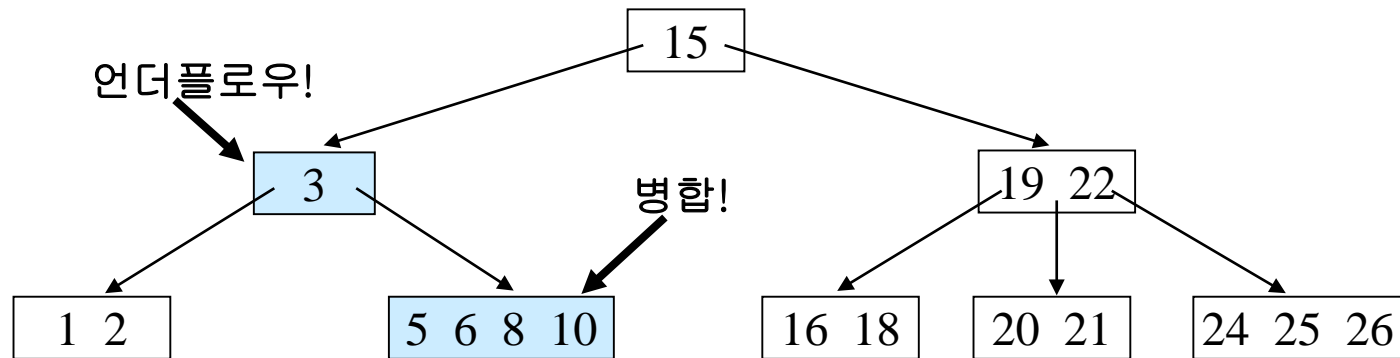
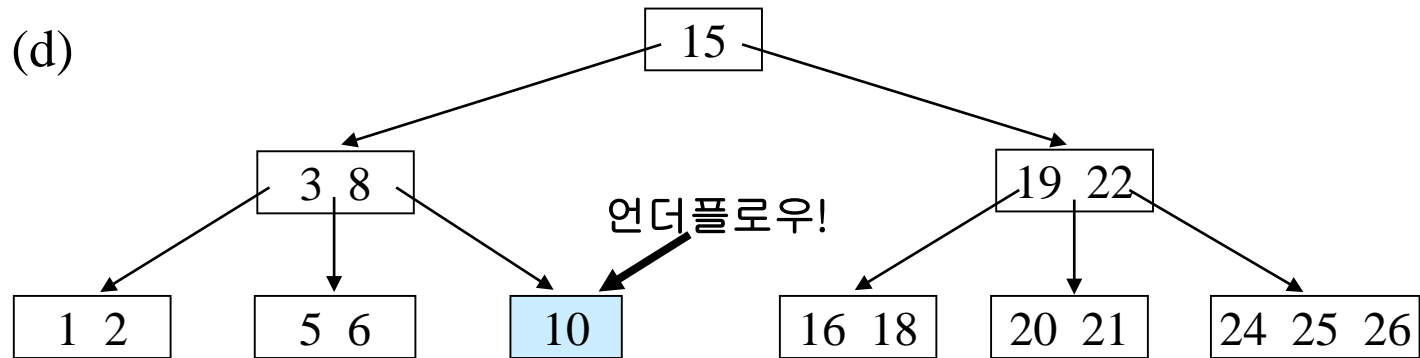


언더플로우!



9 삭제

(d)



# 다차원 검색

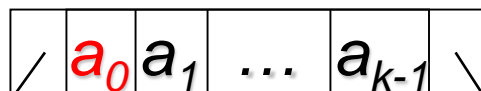
- 검색키가 두 개 이상의 필드로 이루어진 검색
- 3개의 다차원 검색트리 소개
  - KD-트리
  - KDB-트리
  - R-트리



# KD-Tree

- 각 레벨에서 필드를 번갈아가며 검색에 사용한다
  - 한 level에서는 하나의 필드만 사용한다
  - 총  $k$  개의 필드를 사용하는 검색이라면,  $k$  개의 level을 내려가면 검색에 사용하는 필드가 일치한다

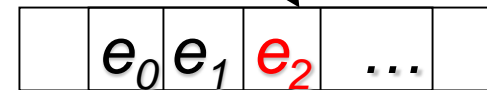
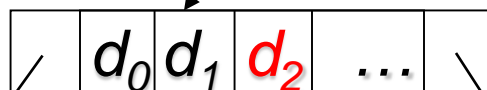
레벨 0



레벨 1



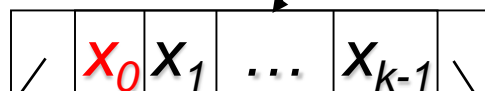
레벨 2

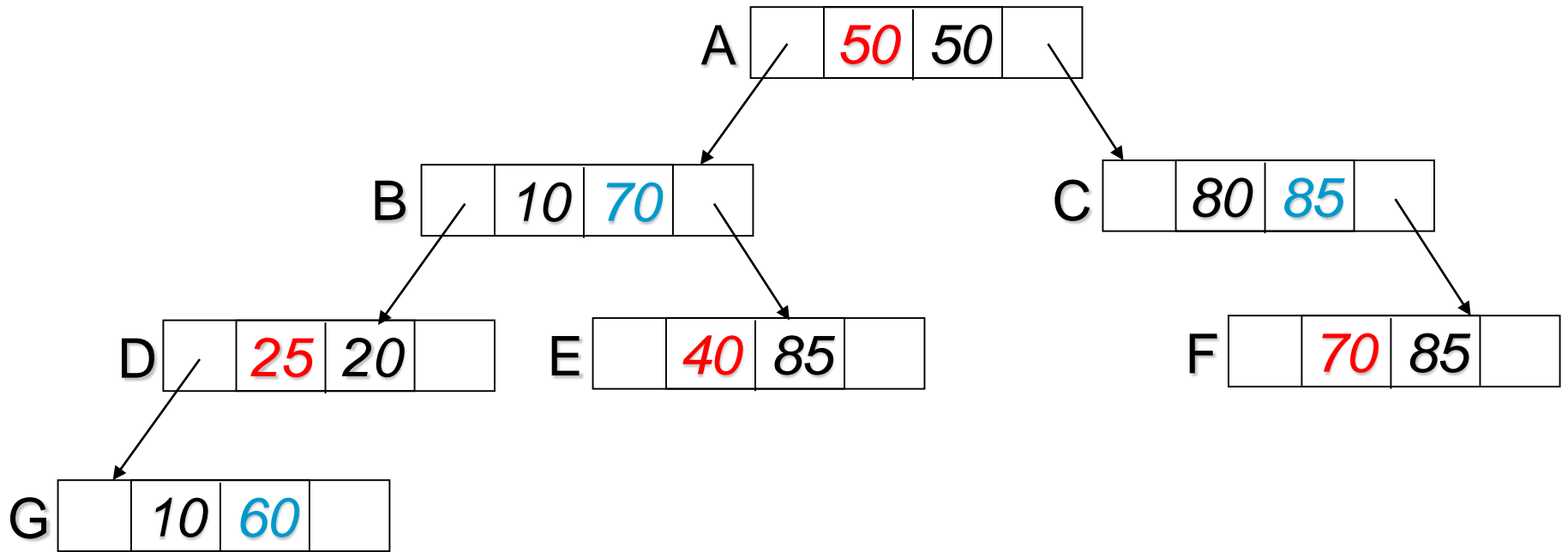


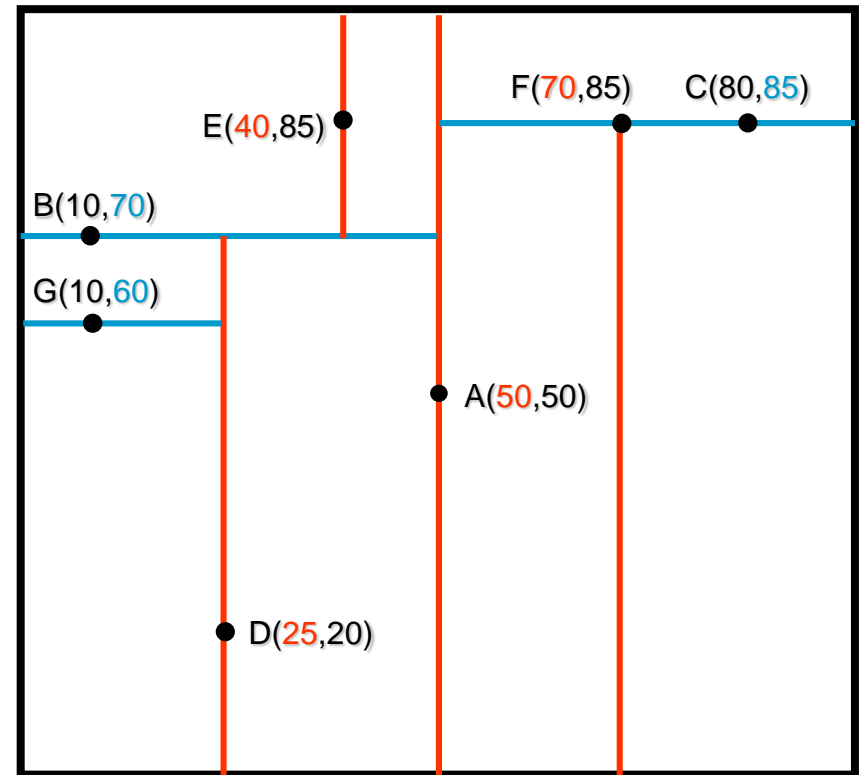
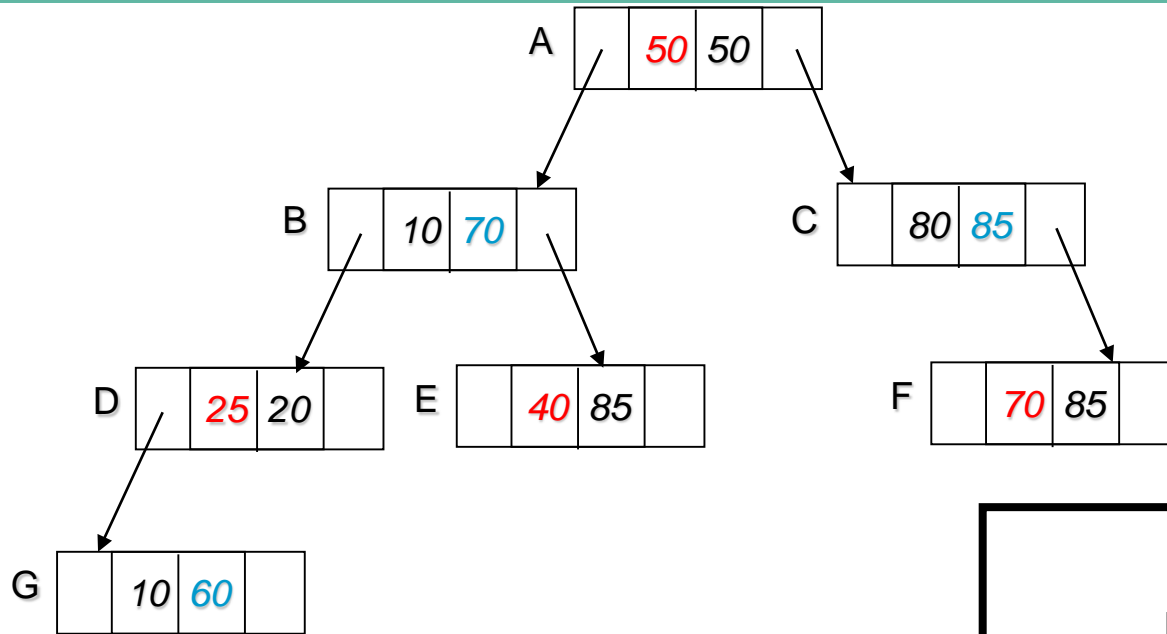
레벨 k-1



레벨 k

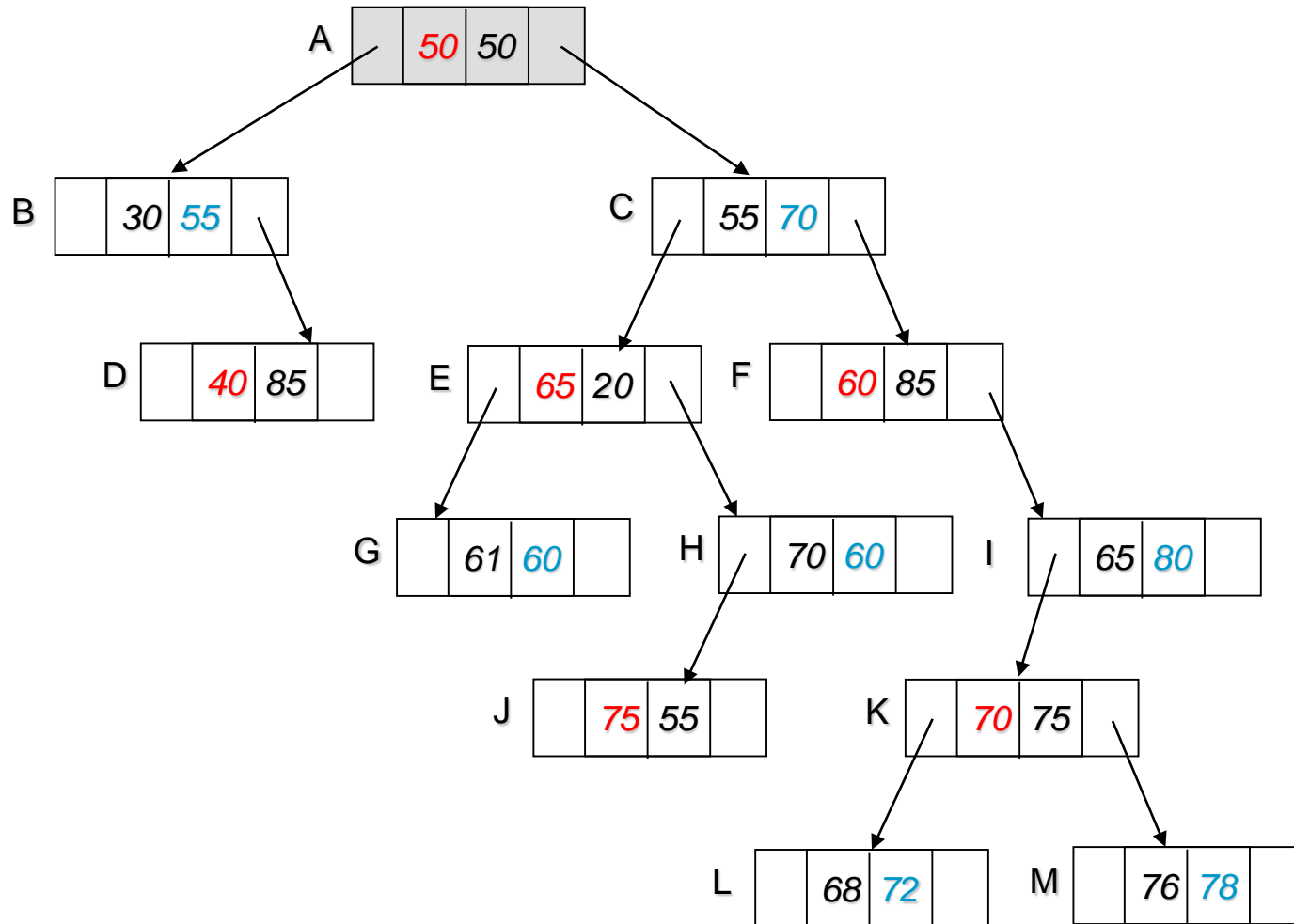




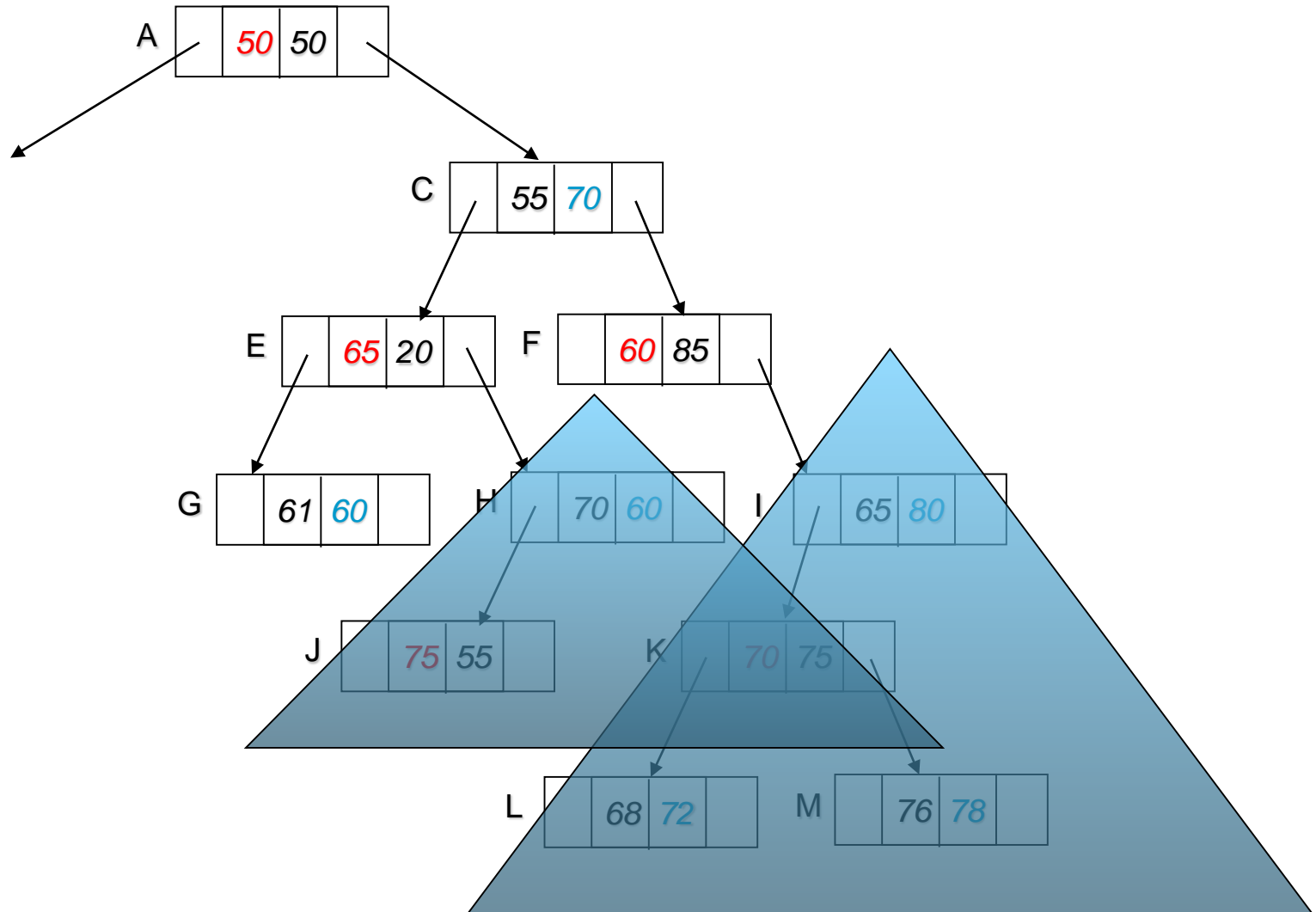


## 삭제의 예: 노드 A 삭제

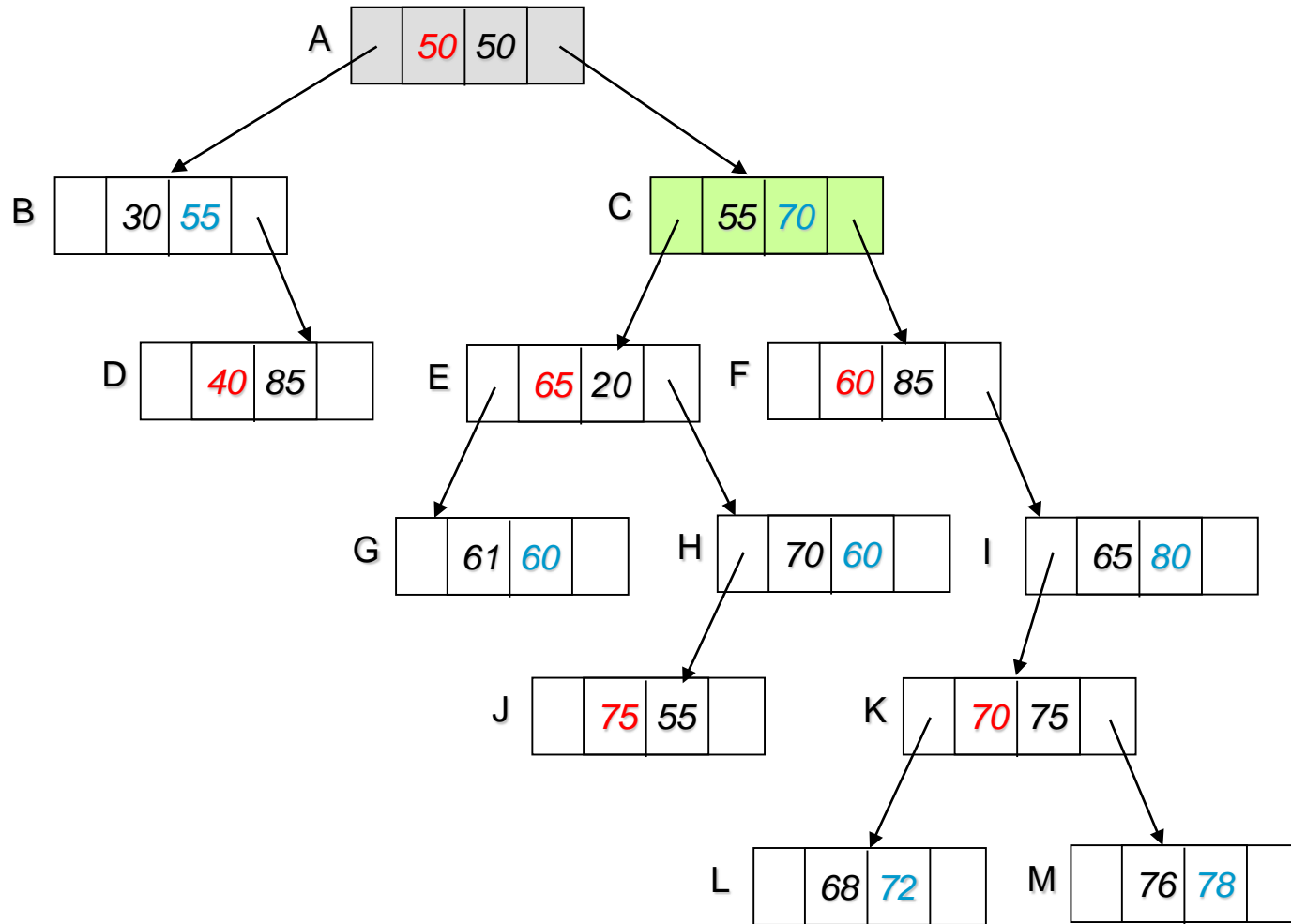
(a)



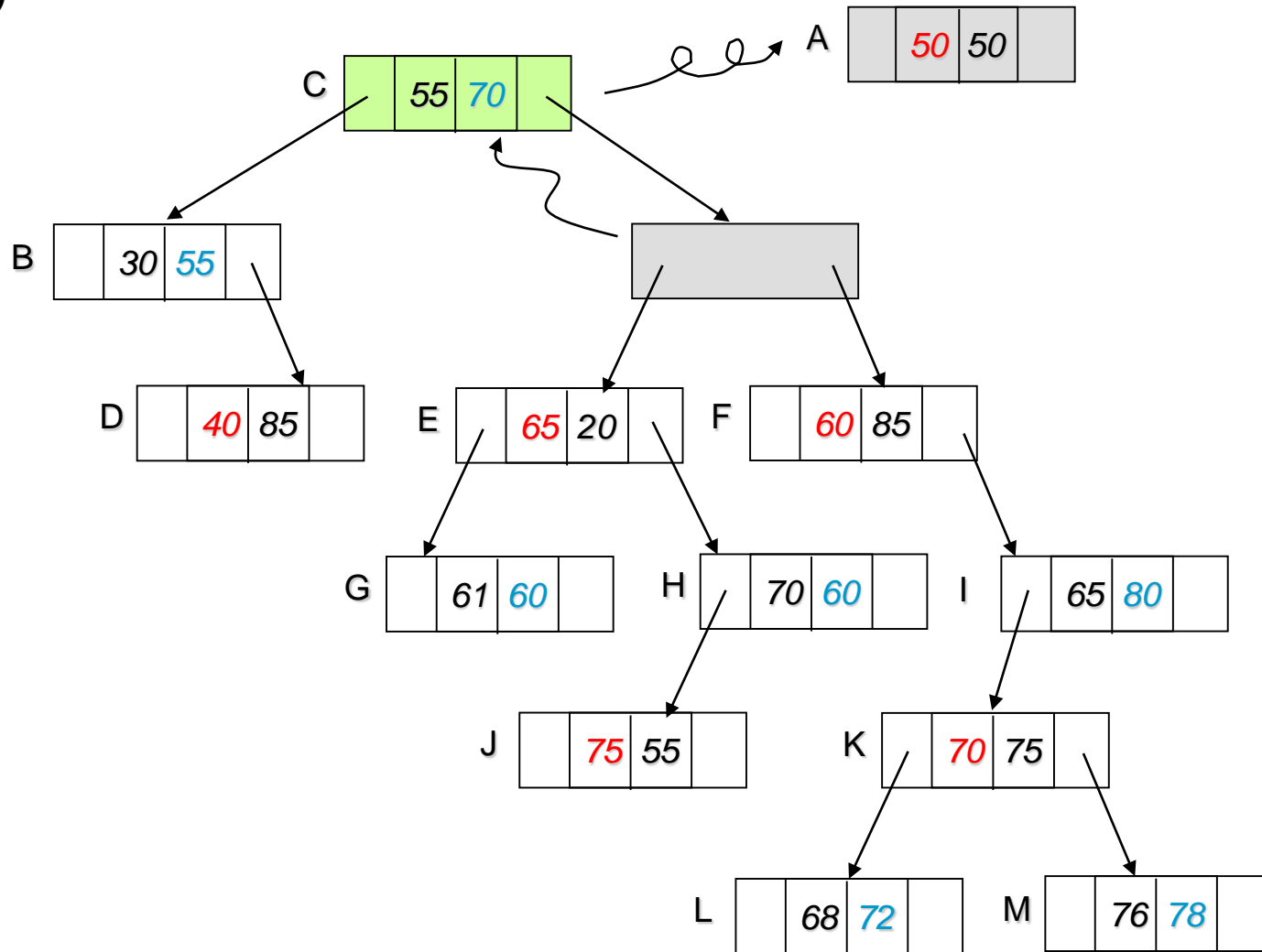
# 가장 작은 값 찾기에서 보지 않아도 되는 부분들



(b)

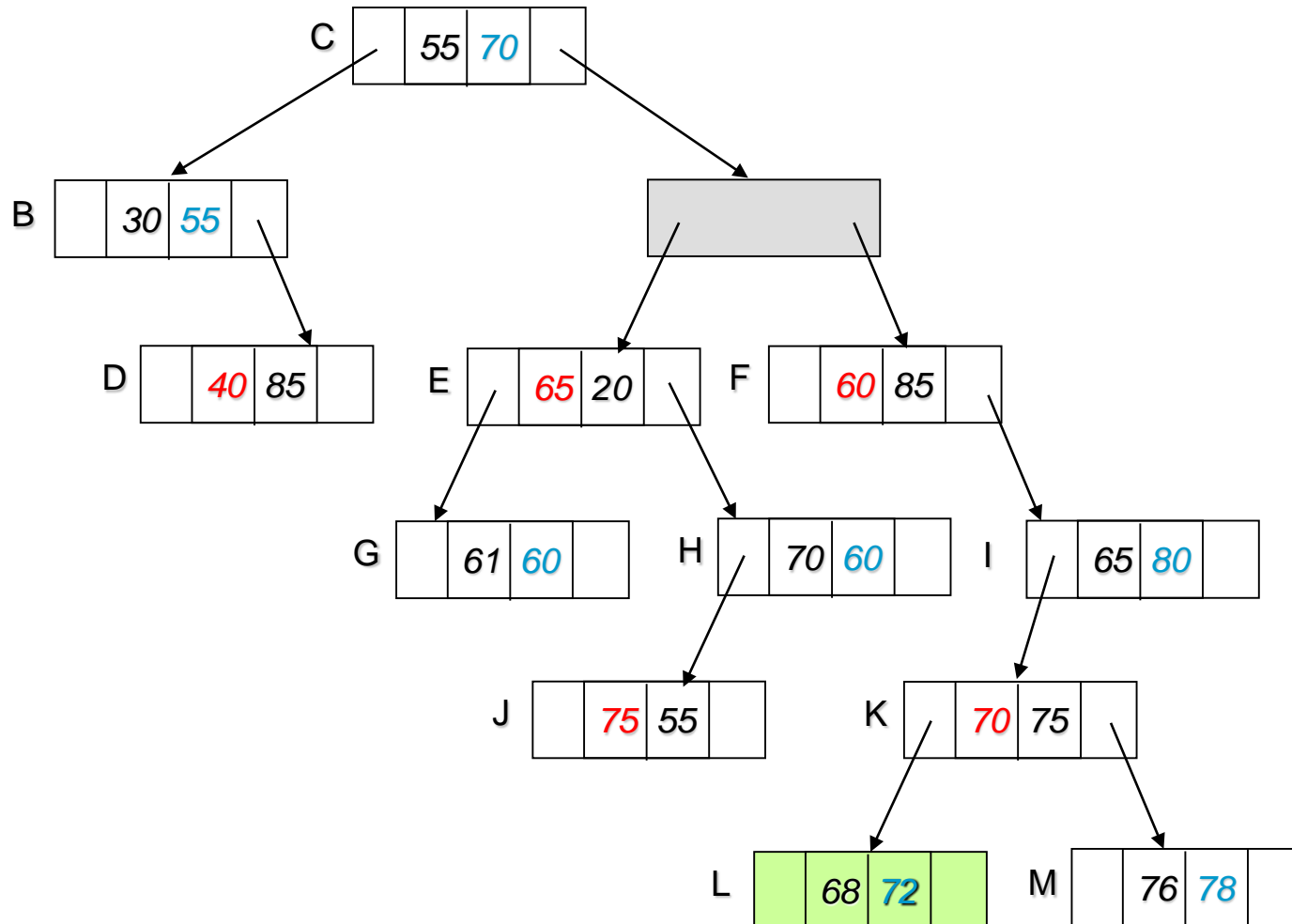


(c)

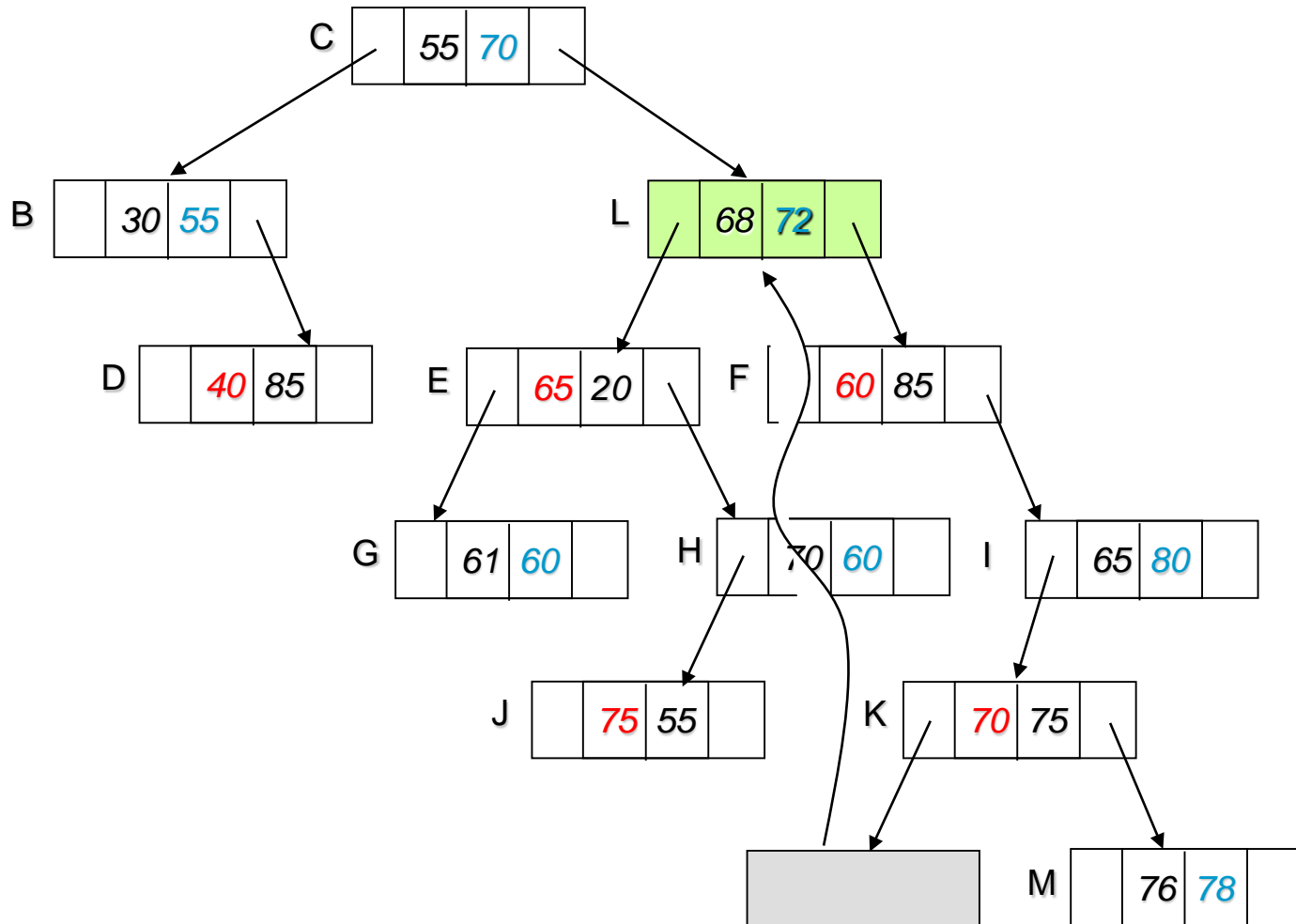




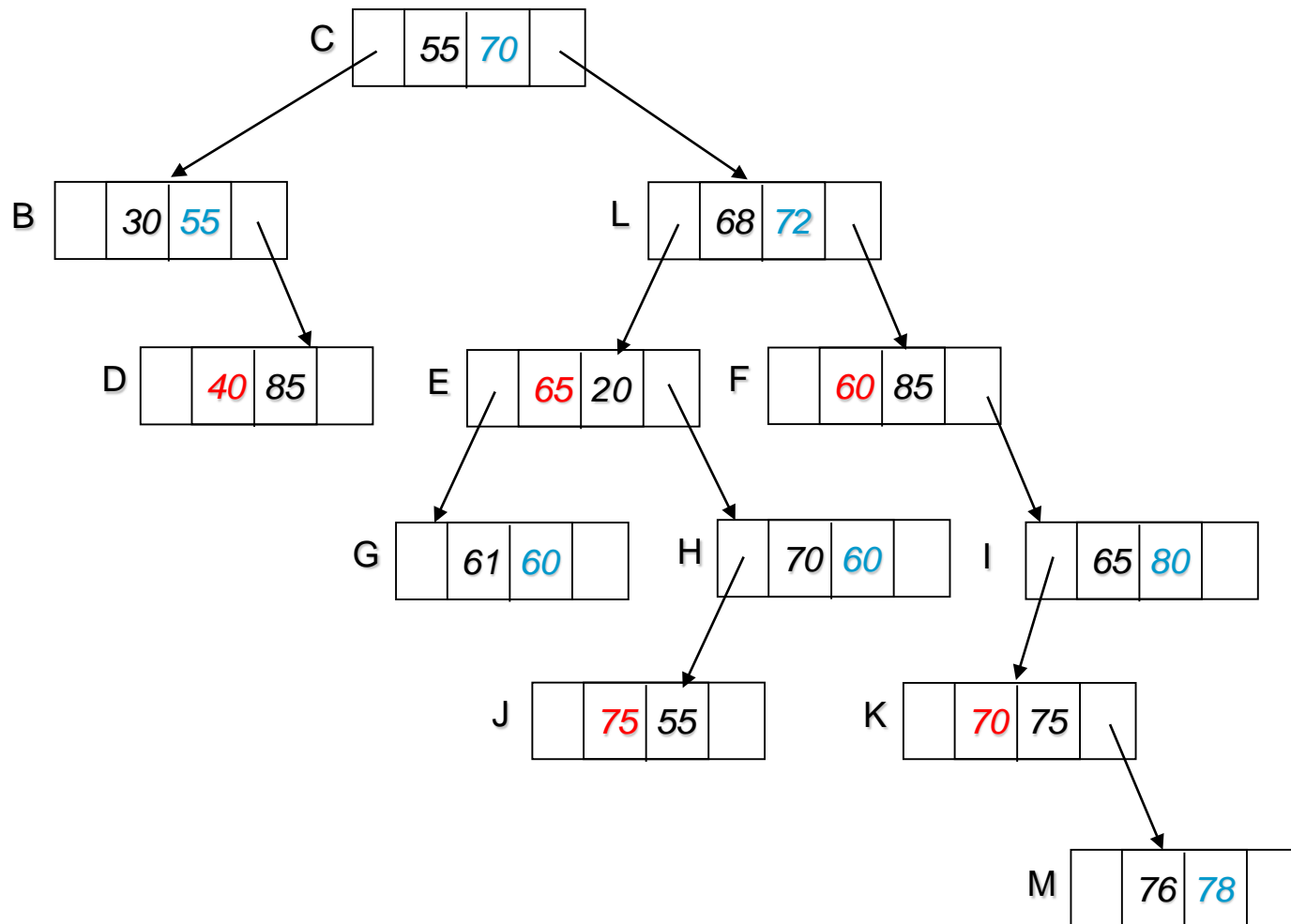
(d)



(e)



(f)

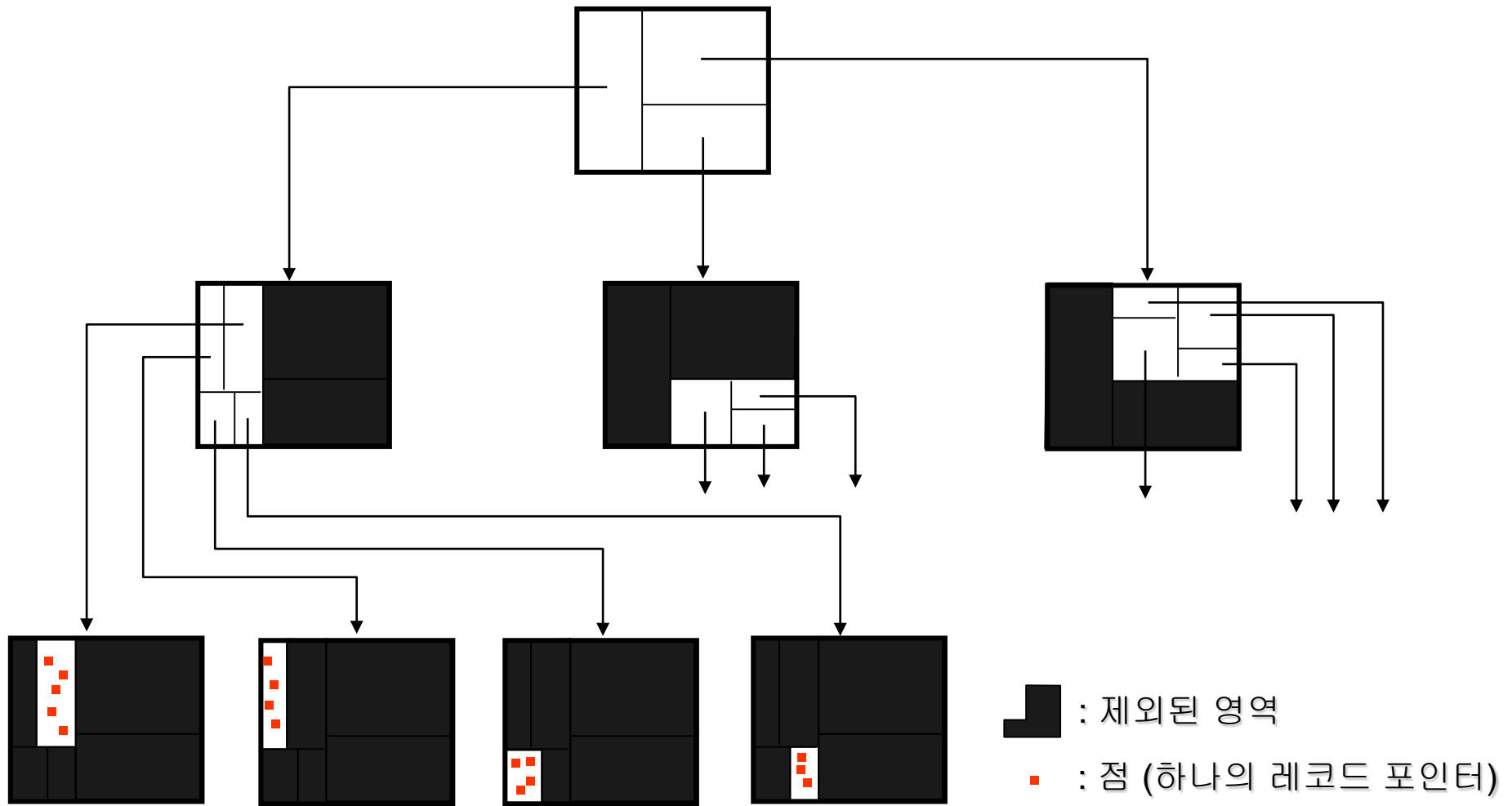


# KDB-Tree

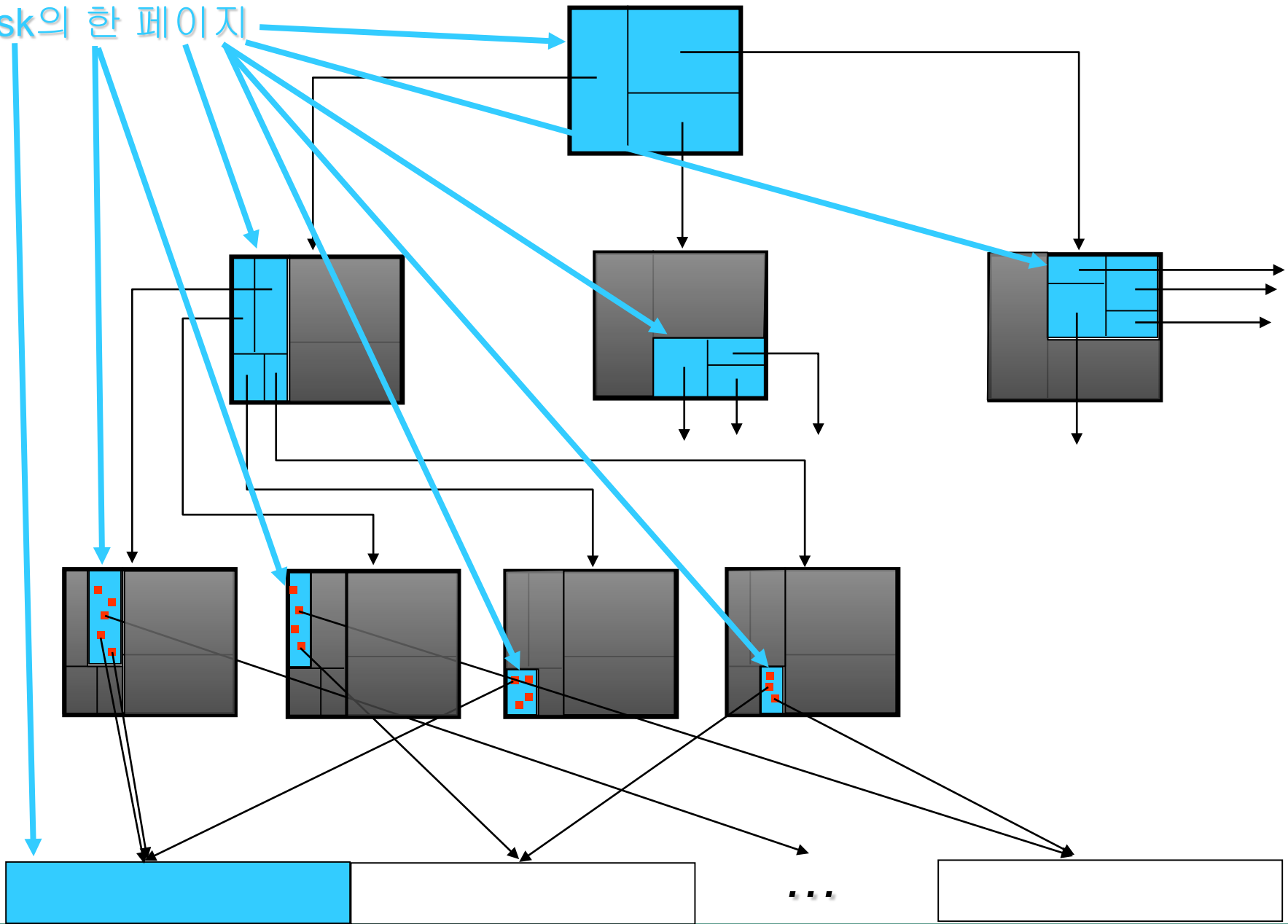
- KD-Tree와 B-Tree의 특성 결합
  - KD-Tree의 특성
    - 다차원 key
  - B-Tree의 특성
    - 디스크의 한 페이지가 한 노드와 일치
    - Balanced tree
- 각 레코드는  $k$ 차원 공간에서 하나의 점에 해당
  - 자신이 속한 공간을 담당하는 색인 node들을 따라감

# KDB-트리의 Node들

- Internal node 하나는  $k$ 차원 공간에서 한 영역을 담당한다
  - 루트 노드는  $k$  차원 공간 전체를 커버
  - 이하의 노드들은  $k$ 차원 공간의 부분 영역을 담당
  - 같은 level에 있는 모든 노드들은 서로 겹치는 영역이 없다
  - 같은 level에 있는 모든 노드들의 담당 영역을 합하면  $k$ 차원 공간 전체가 됨
- 리프 노드는 데이터 페이지 정보를 저장

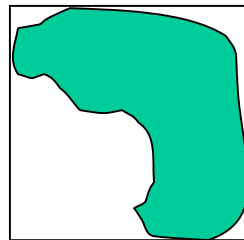
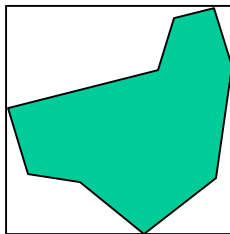


Disk의 한 페이지



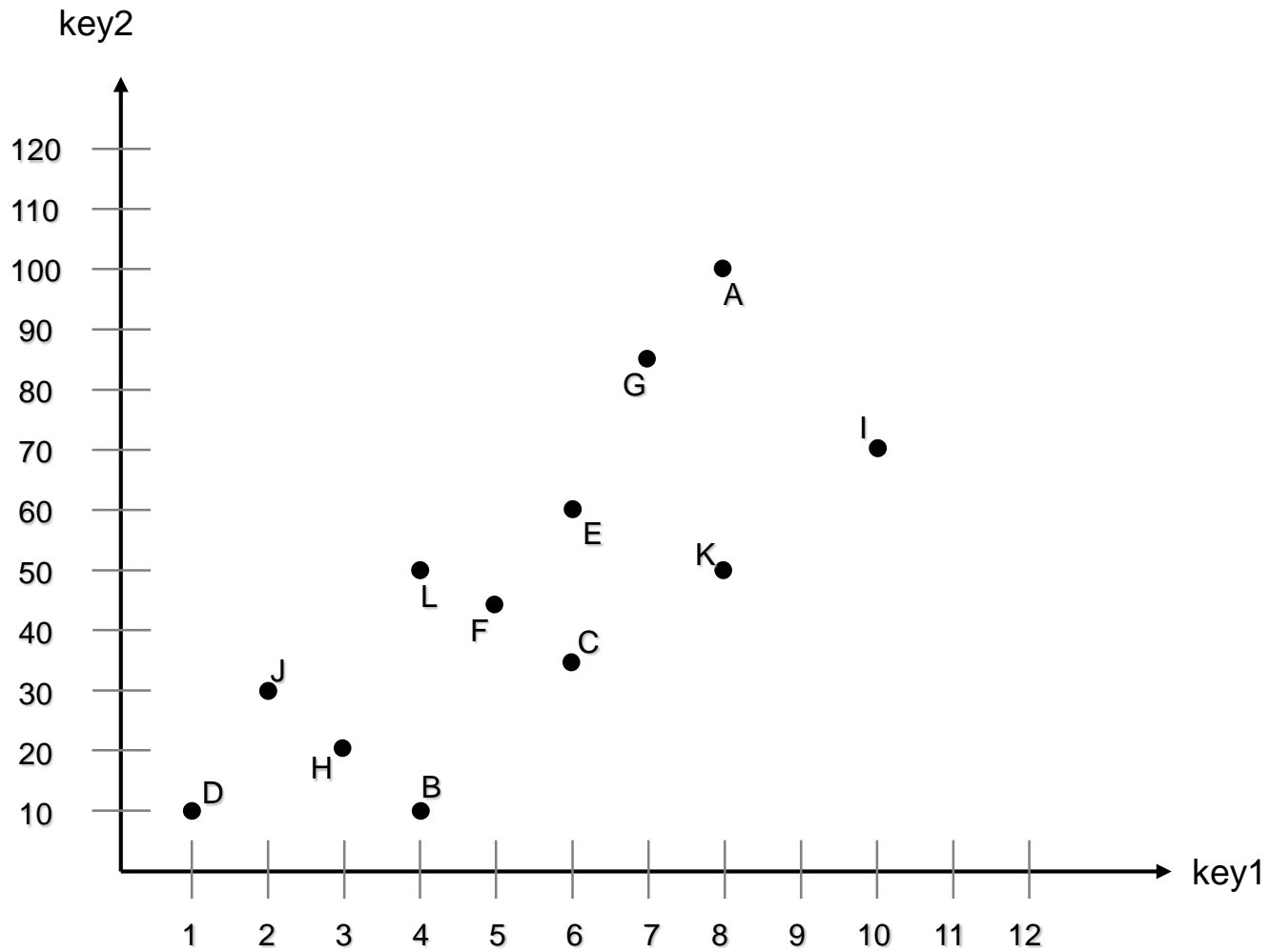
# R-Tree

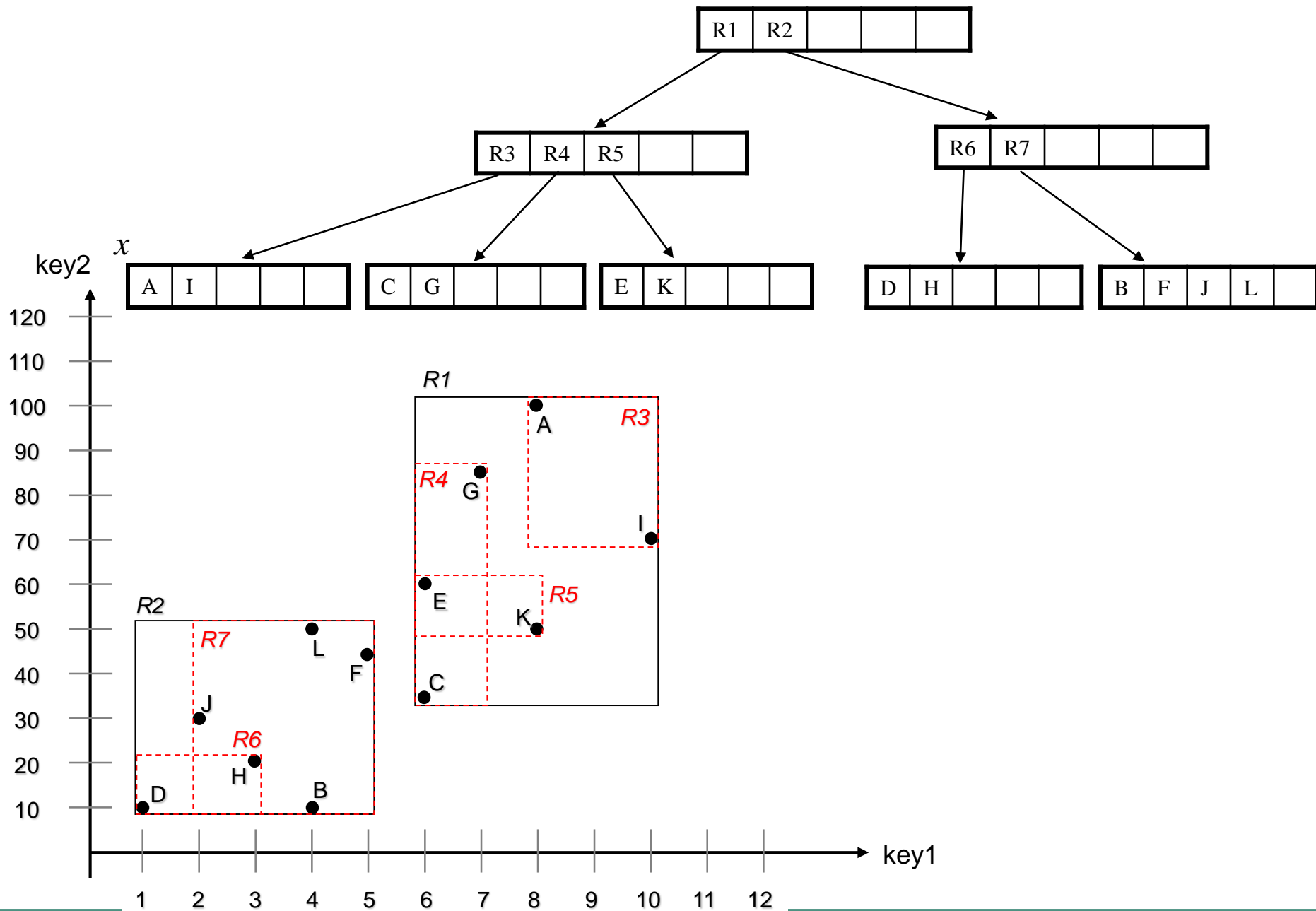
- B-트리의 다차원 확장
- 균형잡힌 검색트리
- 모든 레코드는 리프 노드에서만 가리킴
- 다차원 도형의 저장 가능
  - 점, 선, 면, 폐공간, 각종 도형
  - MBR(Minimum Bounding Rectangle)로 근사



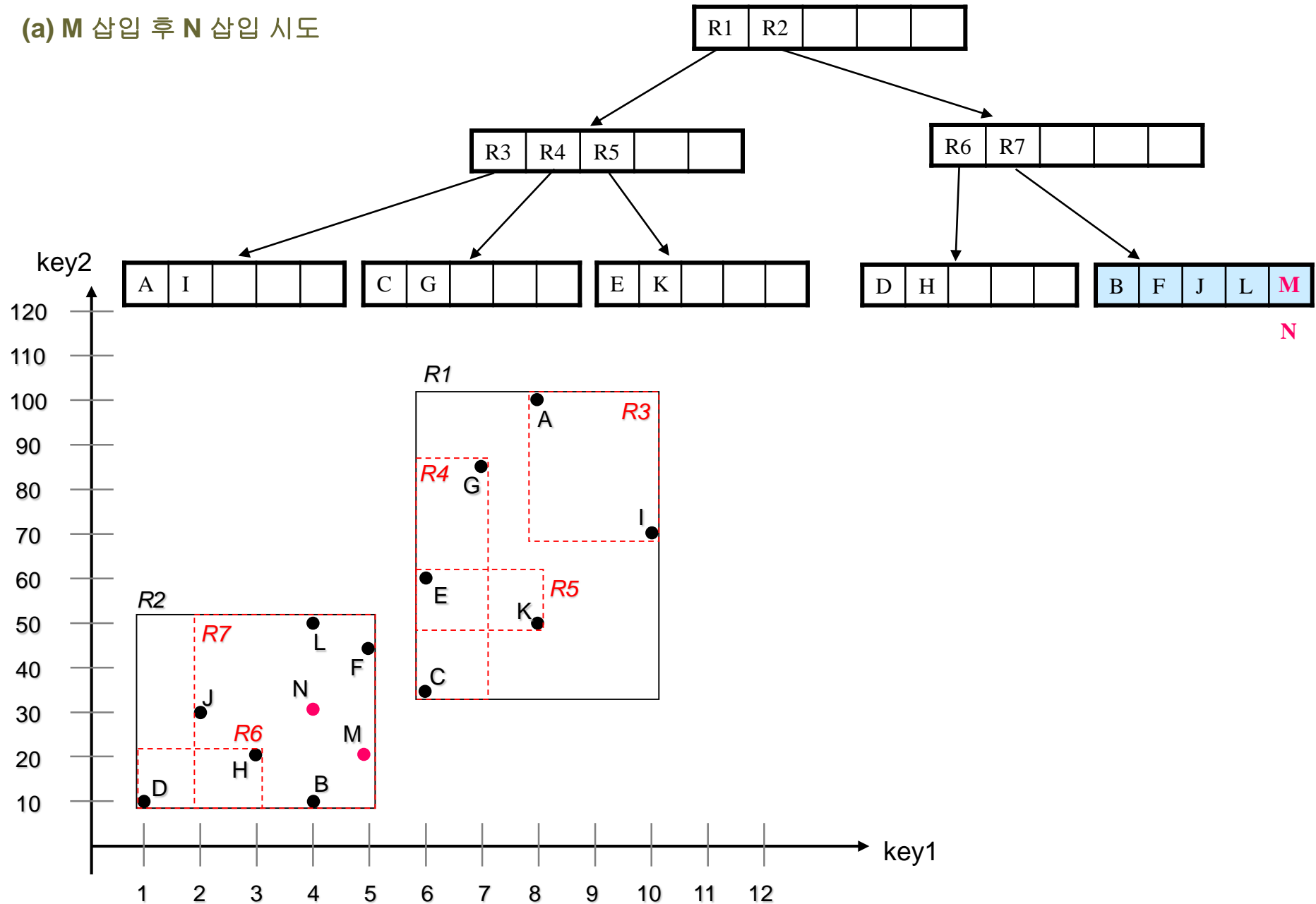


이름	Key1	Key2
A	8	100
B	4	10
C	6	35
D	1	10
E	6	40
F	5	45
G	7	85
H	3	20
I	10	70
J	2	30
K	8	50
L	4	50

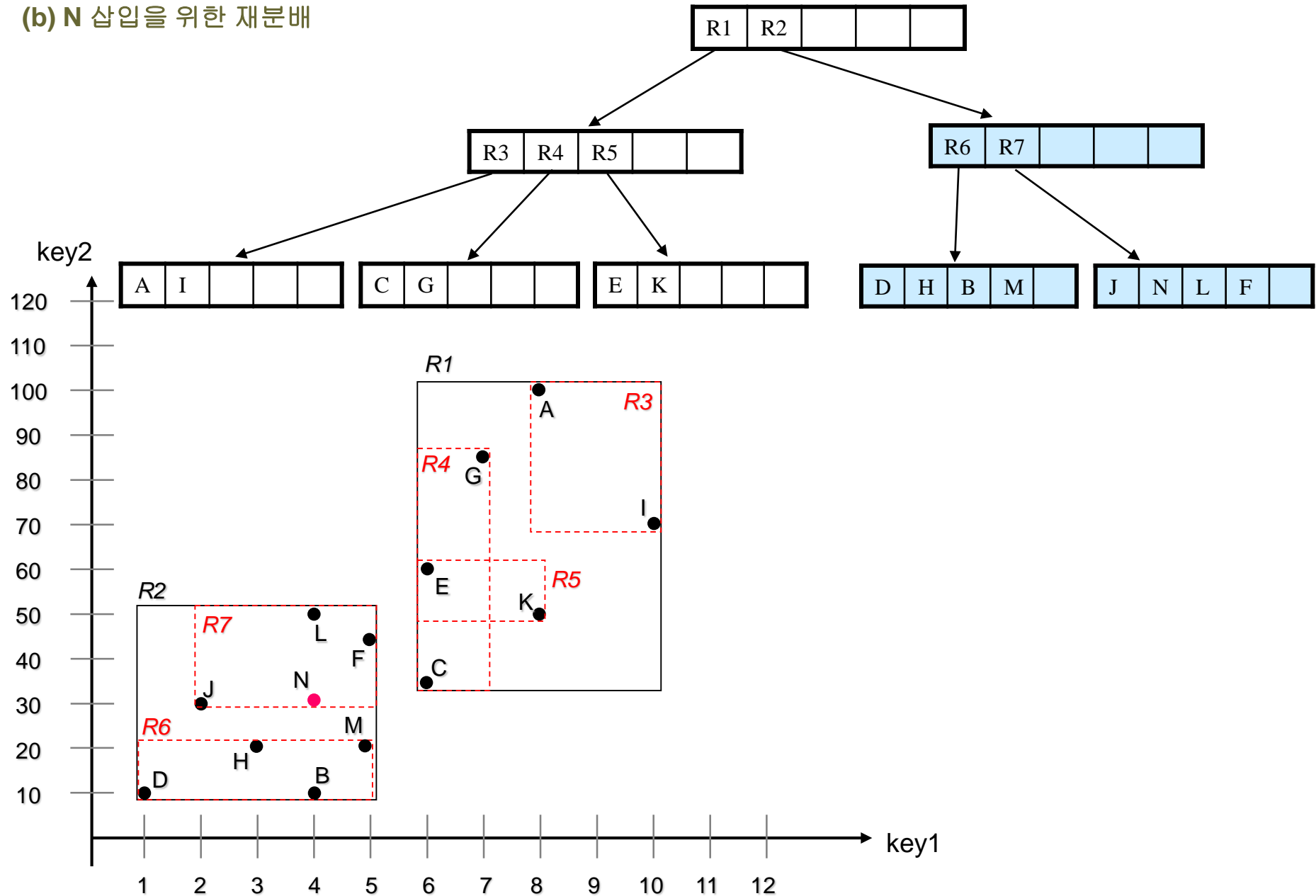




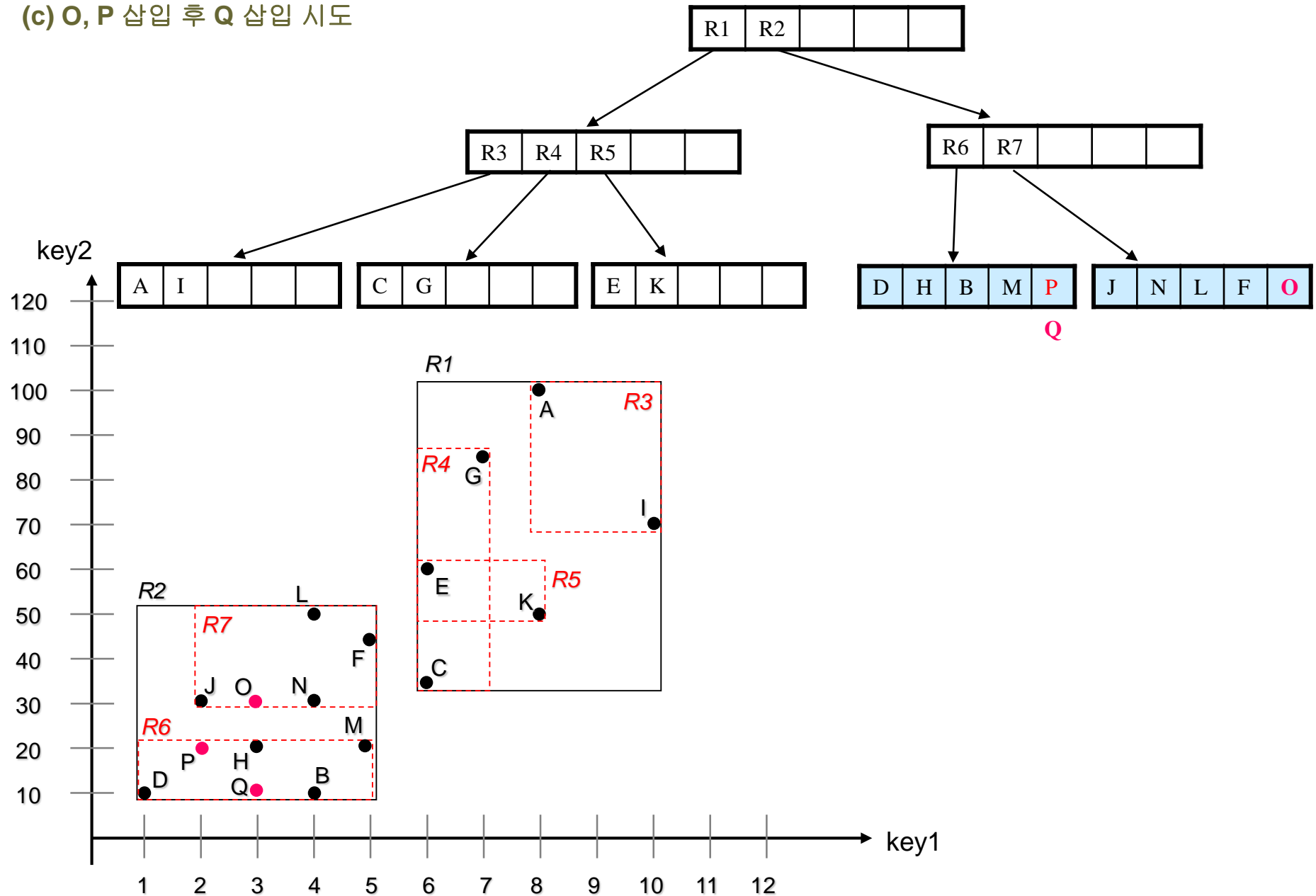
(a) M 삽입 후 N 삽입 시도



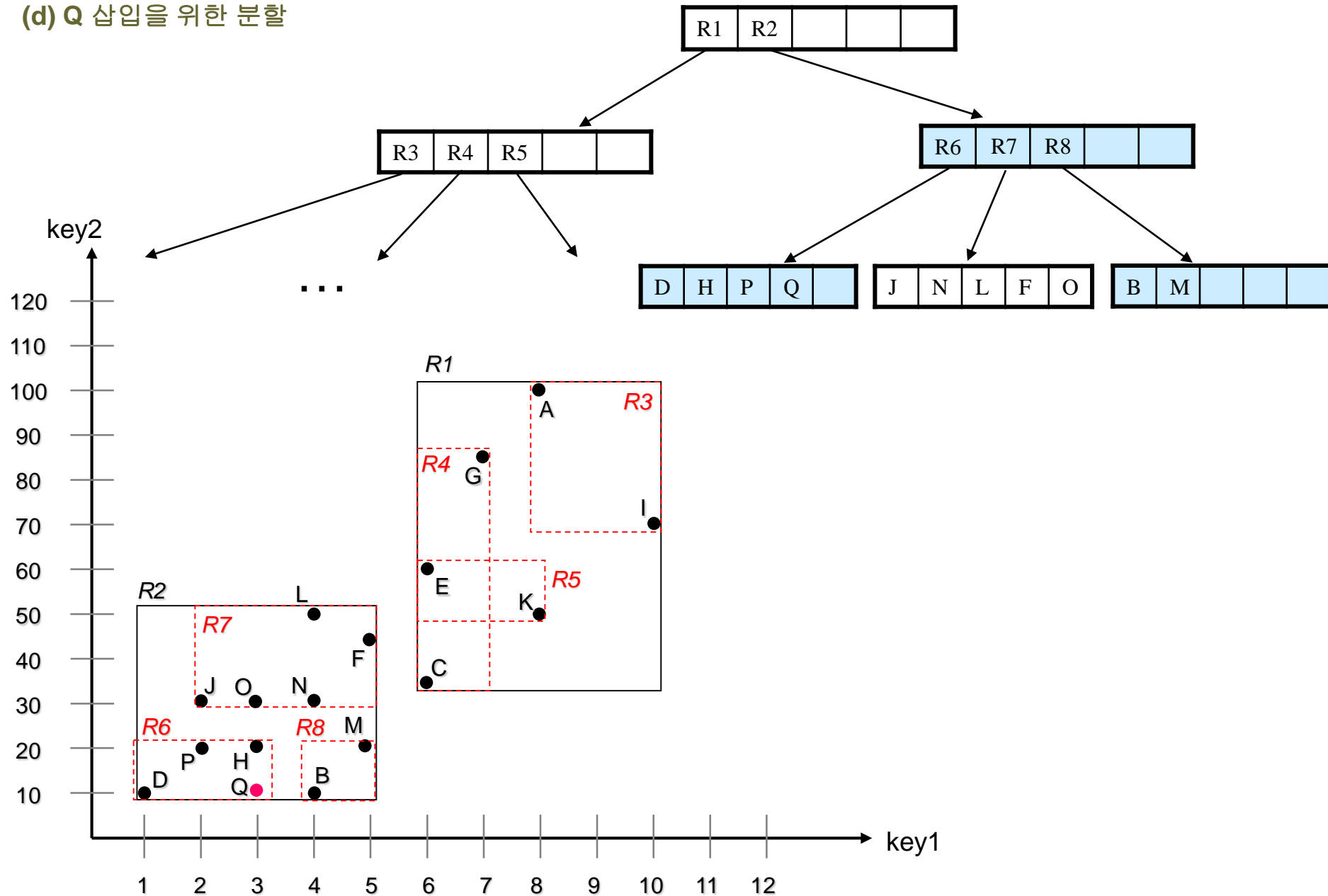
(b) N 삽입을 위한 재분배



(c) O, P 삽입 후 Q 삽입 시도



(d) Q 삽입을 위한 분할





**Thank you**

---