알고리즘

6장 검색 트리(search tree)

학습내용

1. 레코드, 키의 정의 및 검색 트리
2. 이진 검색 트리
3. 레드 블랙 트리
4. B-트리
5. 다차원 검색 트리

학습목표

- 검색에서 레코드와 키의 역할을 구분한다.
- 이진검색트리에서의 검색·삽입·삭제 작업의 원리를 이해한다.
- 이진검색트리의 균형이 작업의 효율성에 미치는 영향을 이해하고, 레드블랙트리의 삽입·삭제 작업의 원리를 이해한다.
- B-트리의 도입 동기를 이해하고 검색·삽입·삭제 작업의 원리를 이해한다.
- 검색트리 관련 작업의 점근적 수행시간을 이해한다.
- 일차원 검색의 기본 원리와 다차원 검색의 연관성을 이해한다.

학습내용

1. **레코드, 키의 정의 및 검색 트리**
2. 이진 검색 트리
3. 레드 블랙 트리
4. B-트리
5. 다차원 검색 트리

검색 트리

- 데이터의 저장과 검색은 자료구조와 알고리즘 분야에서 매우 중요한 주제
- 데이터의 저장/검색을 효율적으로 하기 위해서는 적절한 자료구조와 알고리즘을 사용하는 것이 중요하다.
  1. 데이터가 들어오는 순서대로 배열에 저장하는 방법:

     저장은 간단하지만 검색이 비효율적이다. 자료수가 n일 때
     - 새로운 자료 하나를 저장하는 시간은 $\Theta(1)$
     - 자료를 검색하는 시간은 평균 $\Theta(n)$
  2. 데이터를 트리 모양의 자료구조인 검색 트리에 저장하는 방법: 자료수가 n일 때
     - 새로운 자료 하나를 저장하는 시간은 평균 $\Theta(\log n)$
     - 자료를 검색하는 시간도 평균 $\Theta(\log n)$

레코드와 키

- 레코드(record)
  - 개체에 대해 수집된 모든 정보를 포함하고 있는 저장 단위
  - 예)사람 레코드: **주민등록번호**, 이름, 주소, 전화번호 등의 정보 포함
- 필드(field)
  - 레코드에서 각각의 정보를 나타내는 부분
  - 예) 주민등록번호 필드, 이름 필드, 주소 필드, 전화번호 필드
- 검색키(search key) 또는 키(key)
  - 다른 레코드와 중복되지 않도록 각 레코드를 대표하는 필드
  - 예) **주민등록번호** 필드
  - 키는 하나의 필드로 이루어질 수도 있고, 두 개 이상의 필드로 이루어질 수도 있다.
- 검색트리(search tree)
  - 트리 구조로서 각 노드가 규칙에 맞도록 하나씩의 키를 지니며,이를 통해 해당 레코드가 저장된 위치를 알 수 있다.

검색 트리의 분류

- 자식 노드 개수에 따라
  - 이진검색트리(binary search tree) : 자식 노드 최대 개수가 2
  - 다진검색트리(multi-way search tree) : 자식 노드 최대 개수가 3 이상

    ➔ k진검색트리(k-ary search tree) : 자식 노드 최대 개수가 k
- 저장 장소에 따라
  - 내부검색트리 : 검색트리가 메인 메모리에 존재
  - 외부검색트리 : 검색트리가 외부(주로 디스크)에 존재
- 검색키에 포함된 필드 수에 따라
  - 일차원 검색트리 : 필드가 하나

    예) 이진검색트리, B-트리, AVL-트리, 레드블랙트리, ...
  - 다차원 검색트리 : 필드가 두 개 이상

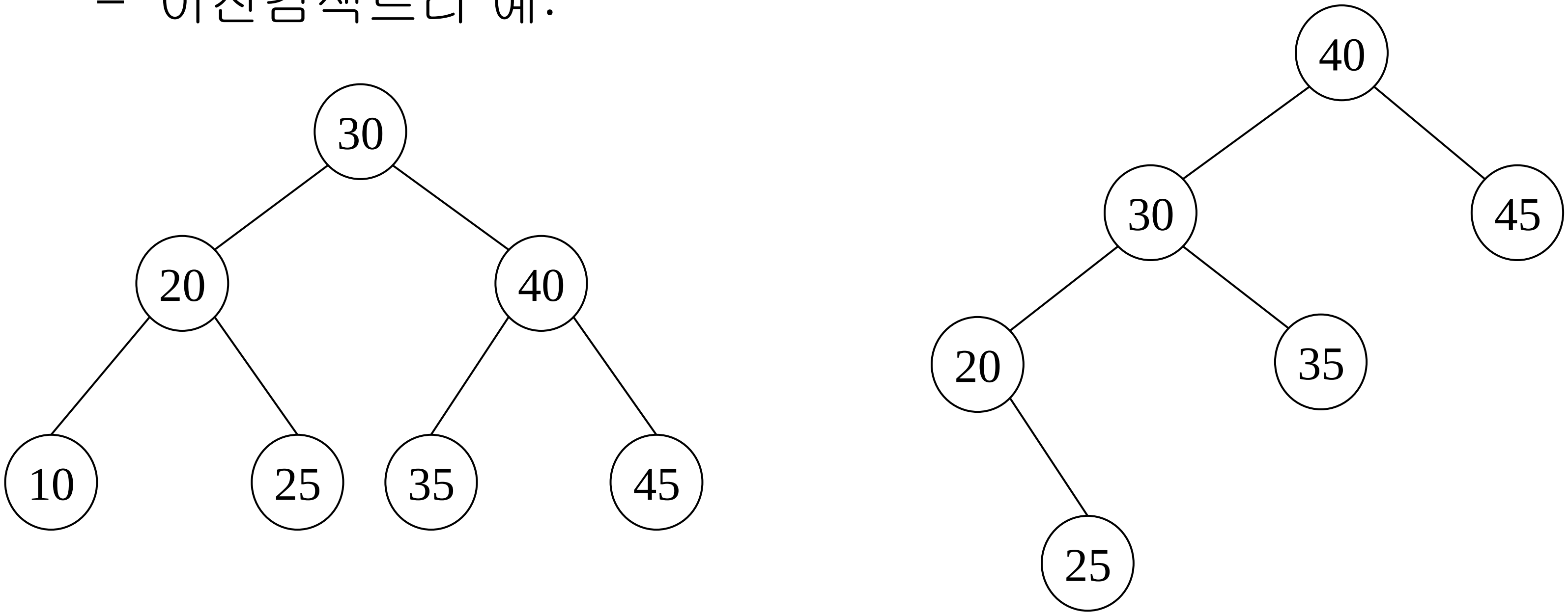
    예) KD-트리, KDB-트리, R-트리, ...

학습내용

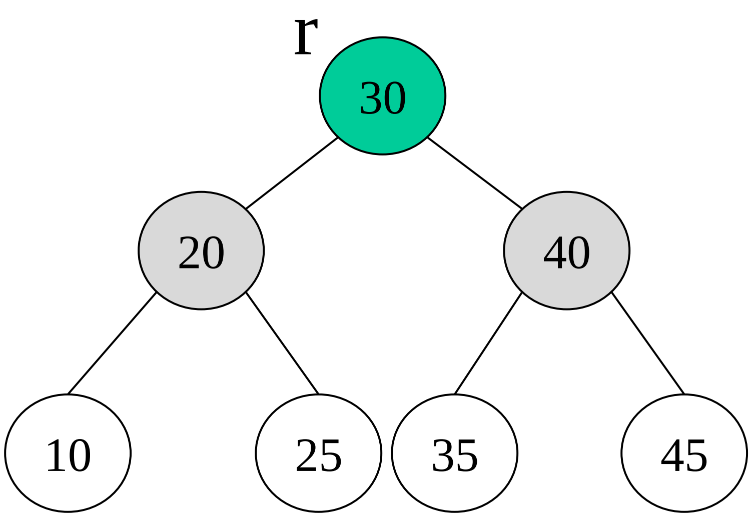
1. 레코드, 키의 정의 및 검색 트리
2. **이진 검색 트리**
3. 레드 블랙 트리
4. B-트리
5. 다차원 검색 트리

Binary Search Tree (BST)

- 이진검색트리(binary search tree)
  - 각 노드는 키 값을 하나씩 가지며, 키 값은 모두 다르다.
  - 최상위 레벨에 루트 노드(root node)가 있고, 각 노드는 최대 두 개의 자식을 갖는다.
  - 각 노드의 키값은 자신의 왼쪽 서브트리 모든 노드의 키 값보다 크고, 오른쪽 서브트리 모든 노드의 키 값보다 작다.
  - 이진검색트리 예:



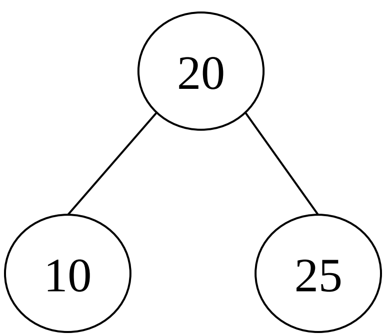
서브트리의 예



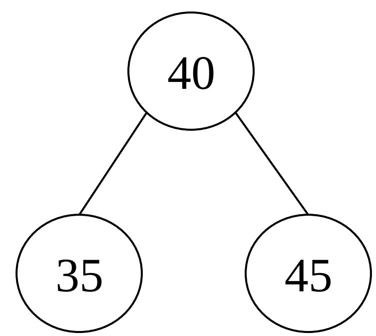
트리의 루트 노드는?

노드 r의 왼쪽 서브트리의 루트는?

노드 r의 오른쪽 서브트리의 루트는?



노드 r의 왼쪽 서브트리



노드 r의 오른쪽 서브트리

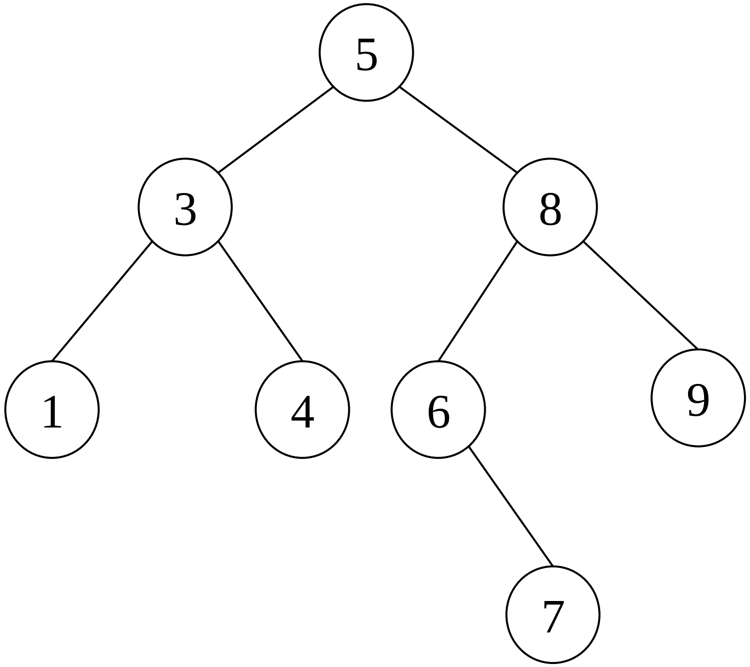
이진검색트리 연산

(1) 순회

(2) 검색

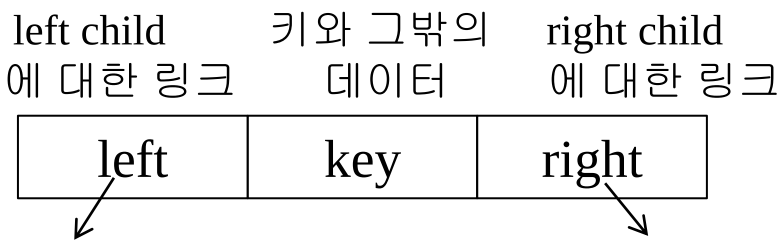
(3) 삽입

(4) 삭제

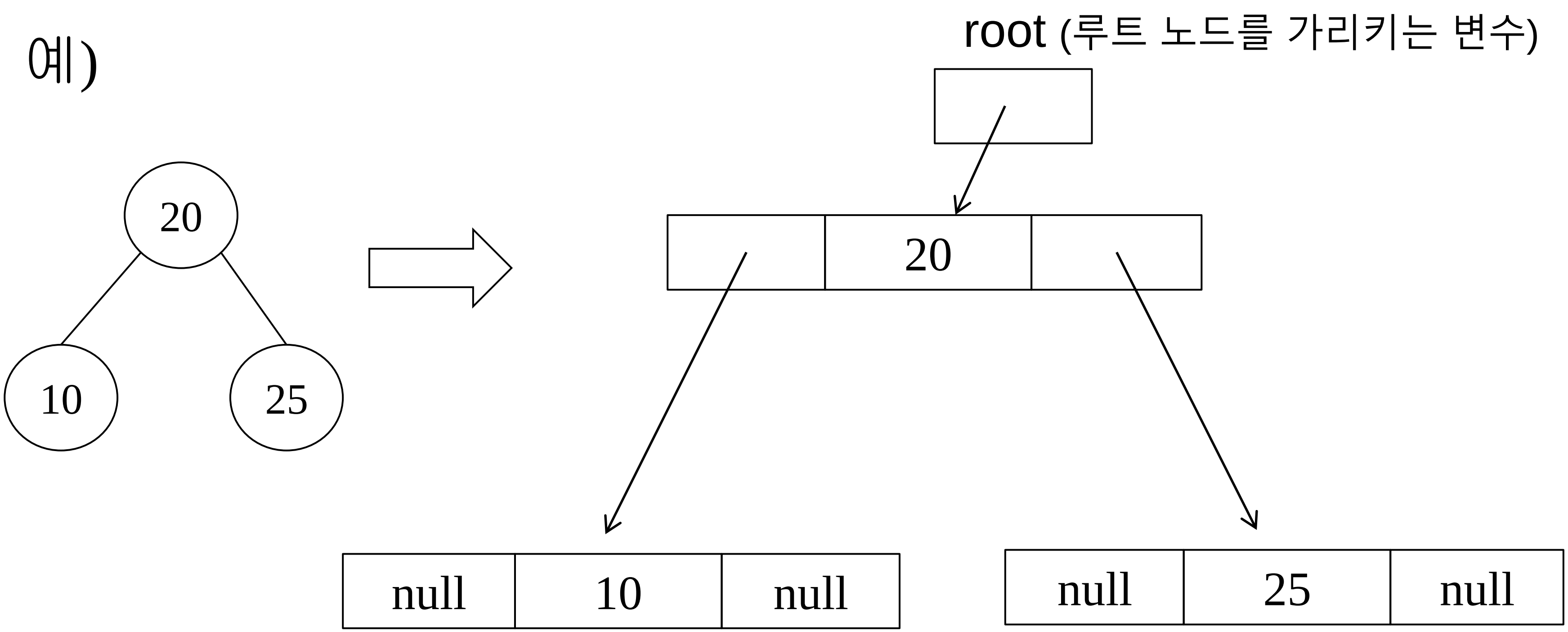


이진트리의 연결 자료구조 구현

노드 구조



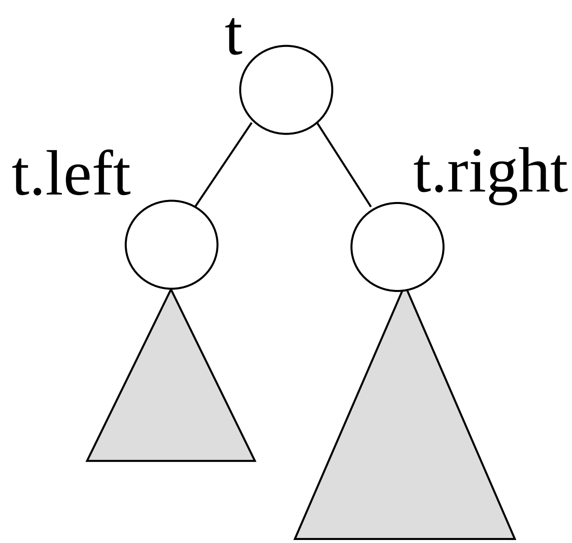
예)



(1) 이진트리 / 이진검색트리 순회

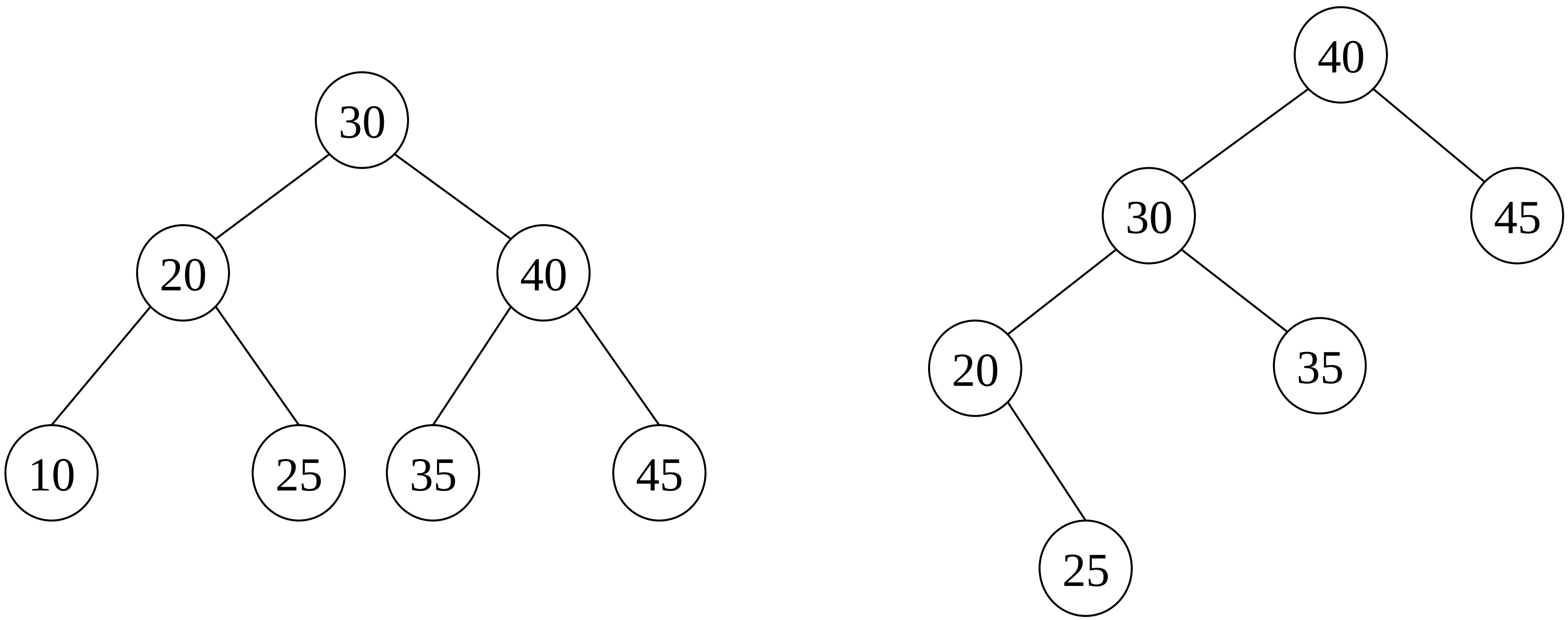
t: 트리의 루트 노드
트리의 노드들을 중순위(inorder)로 순회하여 key 값을 출력

```
treeInorderTraverse(t)
{
    if (t ≠ NIL) then
    {
        treeInorderTraverse(t.left);
        print t.key;
        treeInorderTraverse(t.right);
    }
}
```



BST 순회 예

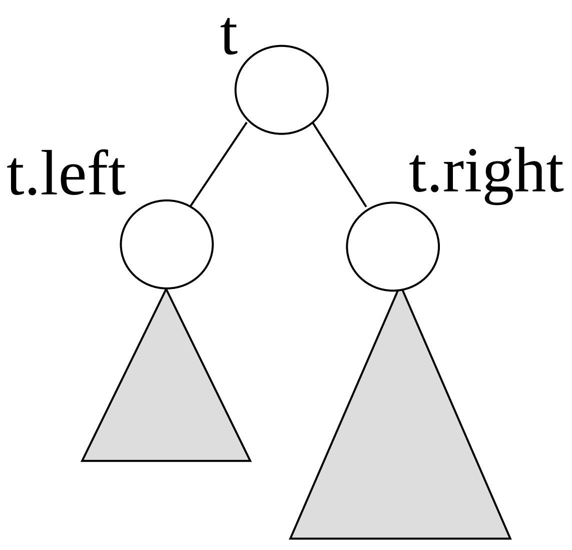
- 중순위 순회(inorder traversal)



(2) 이진검색트리 검색

t: 트리의 루트 노드
x: 검색하고자 하는 키
검색 성공시 해당 노드 리턴; 실패시 NIL(null 값) 리턴

```
treeSearch(t, x)
{
    if (t=NIL or t.key=x) then
            return t;
    if (x < t.key)   then
            return treeSearch(t.left, x);
    else
            return treeSearch(t.right, x);
}
```

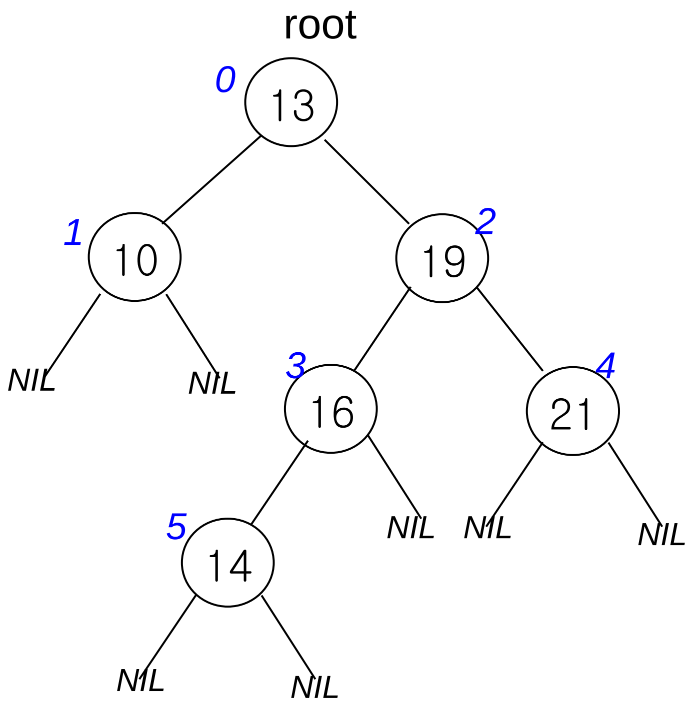


```
treeSearch(t, x)
{
    if (t=NIL or t.key=x) then
        return t;
    if (x < t.key)    then
        return treeSearch(t.left, x);
    else
        return treeSearch(t.right, x);
}
```

```
treeSearch(t, x)
{
    if (t=NIL or t.key=x) then
        return t;
    if (x < t.key)    then
        return treeSearch(t.left, x);
    else
        return treeSearch(t.right, x);
}
```

```
treeSearch(t, x)
{
    if (t=NIL or t.key=x) then
        return t;
    if (x < t.key)    then
        return treeSearch(t.left, x);
    else
        return treeSearch(t.right, x);
}
```

```
node = treeSearch(root, 16);
```

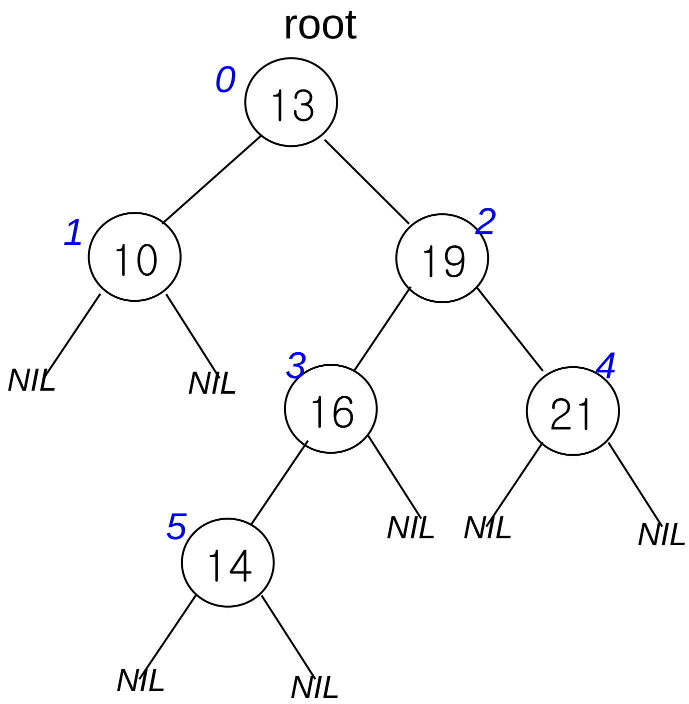


```
treeSearch(t, x)
{
    if (t=NIL or t.key=x) then
        return t;
    if (x < t.key)    then
        return treeSearch(t.left, x);
    else
        return treeSearch(t.right, x);
}
```

```
treeSearch(t, x)
{
    if (t=NIL or t.key=x) then
        return t;
    if (x < t.key)    then
        return treeSearch(t.left, x);
    else
        return treeSearch(t.right, x);
}
```

```
treeSearch(t, x)
{
    if (t=NIL or t.key=x) then
        return t;
    if (x < t.key)    then
        return treeSearch(t.left, x);
    else
        return treeSearch(t.right, x);
}
```

```
node = treeSearch(root, 7);
```



(3) 이진검색트리 삽입

t: 트리의 루트 노드
x: 삽입하고자 하는 키(트리에 존재하지 않는 키라고 가정)
삽입 후 루트 노드의 포인터를 리턴

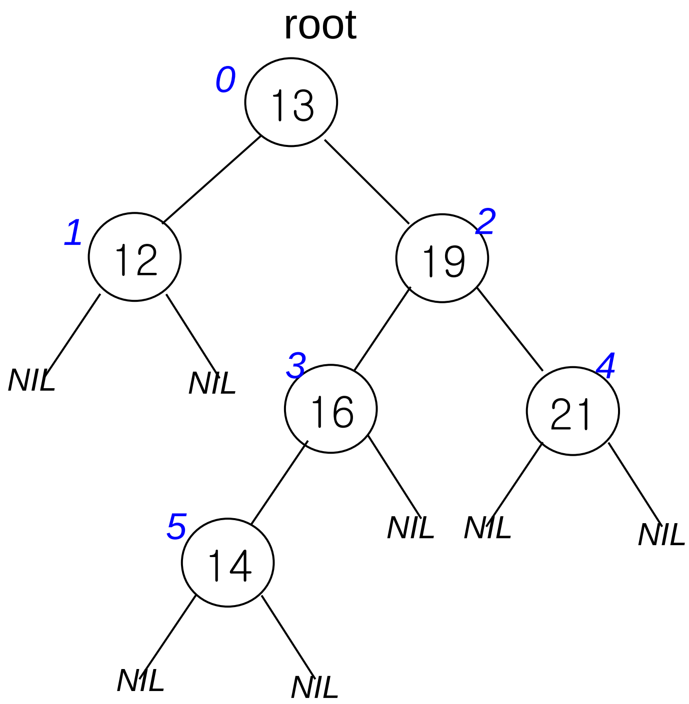
```
treeInsert(t, x)
{
    if (t = NIL) then {
        r.key ← x;          ▷ r : 새로 할당받은 노드
        return r;
    }
    if (x < t.key) then {
        t.left ← treeInsert(t.left, x);
        return t;
    }
    else {                  ▷ x > t.key
        t.right ← treeInsert(t.right, x);
        return t;
    }
}
```

```
treeInsert(t, x) {
    if (t = NIL) then {   ▷ r : 새로 할당받은 노드
        r.key ← x;   return r;
    }
    if (x < t.key) then {
        t.left ← treeInsert(t.left, x);   return t;
    }
    else {
        t.right ← treeInsert(t.right, x);   return t;
    }
}
```

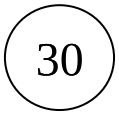
```
treeInsert(t, x) {
    if (t = NIL) then {   ▷ r : 새로 할당받은 노드
        r.key ← x;   return r;
    }
    if (x < t.key) then {
        t.left ← treeInsert(t.left, x);   return t;
    }
    else {
        t.right ← treeInsert(t.right, x);   return t;
    }
}
```

```
treeInsert(t, x) {
    if (t = NIL) then {   ▷ r : 새로 할당받은 노드
        r.key ← x;   return r;
    }
    if (x < t.key) then {
        t.left ← treeInsert(t.left, x);   return t;
    }
    else {
        t.right ← treeInsert(t.right, x);   return t;
    }
}
```

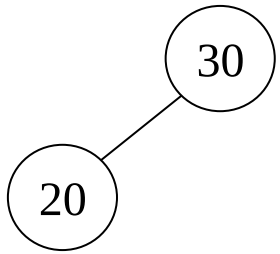
```
root = treeInsert(root, 11);
```



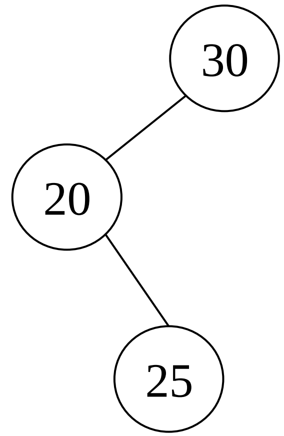
BST 삽입 예 30, 20, 25, 40, 10, 35



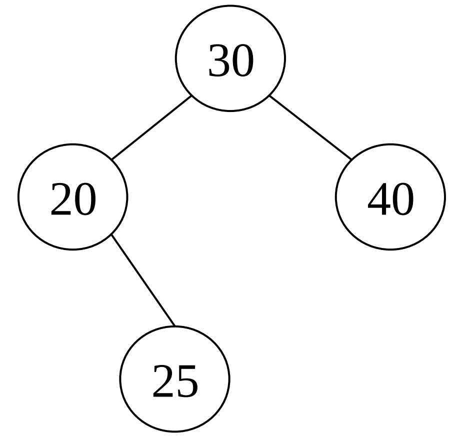
(a)



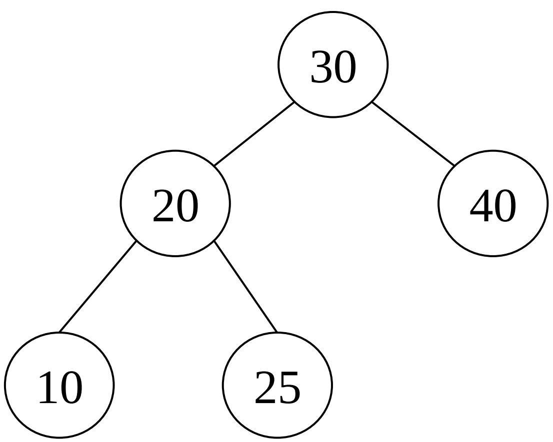
(b)



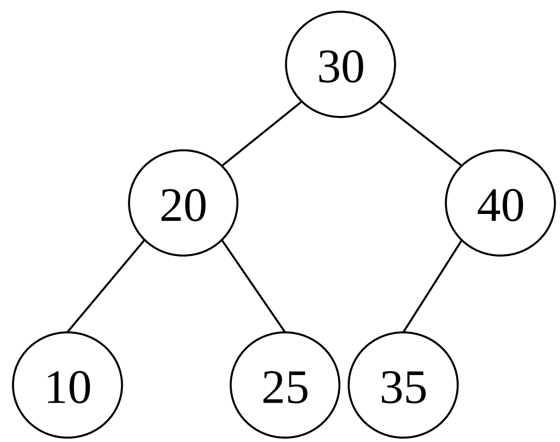
(c)



(d)



(e)



(f)

BST 삽입 예

10, 20, 25, 30, 35, 40

BST 삽입 예

40, 35, 30, 25, 20, 10

(4) 이진검색트리 삭제

t: 트리의 루트 노드
r: 삭제하고자 하는 노드

- 세가지 경우가 있으며, 각각 처리 방법이 다름
  - Case 1 : r이 리프 노드인 경우
  - Case 2 : r의 자식 노드가 하나인 경우
  - Case 3 : r의 자식 노드가 두 개인 경우

```
Sketch-TreeDelete(t, r)  // 삭제 알고리즘의 간단한 골격
{
    if (r이 리프 노드) then                          ▷ Case 1
        그냥 r을 버린다;
    else if (r의 자식이 하나만 있음) then            ▷ Case 2
        r의 부모가 r의 자식을 직접 가리키도록 한다;
    else                                             ▷ Case 3
        r의 오른쪽 서브트리의 최소원소 노드 s를 삭제하고,
        s를 r 자리에 놓는다;
}
```

BST 삭제 알고리즘

root: 트리의 루트 노드

r: 삭제하고자 하는 노드

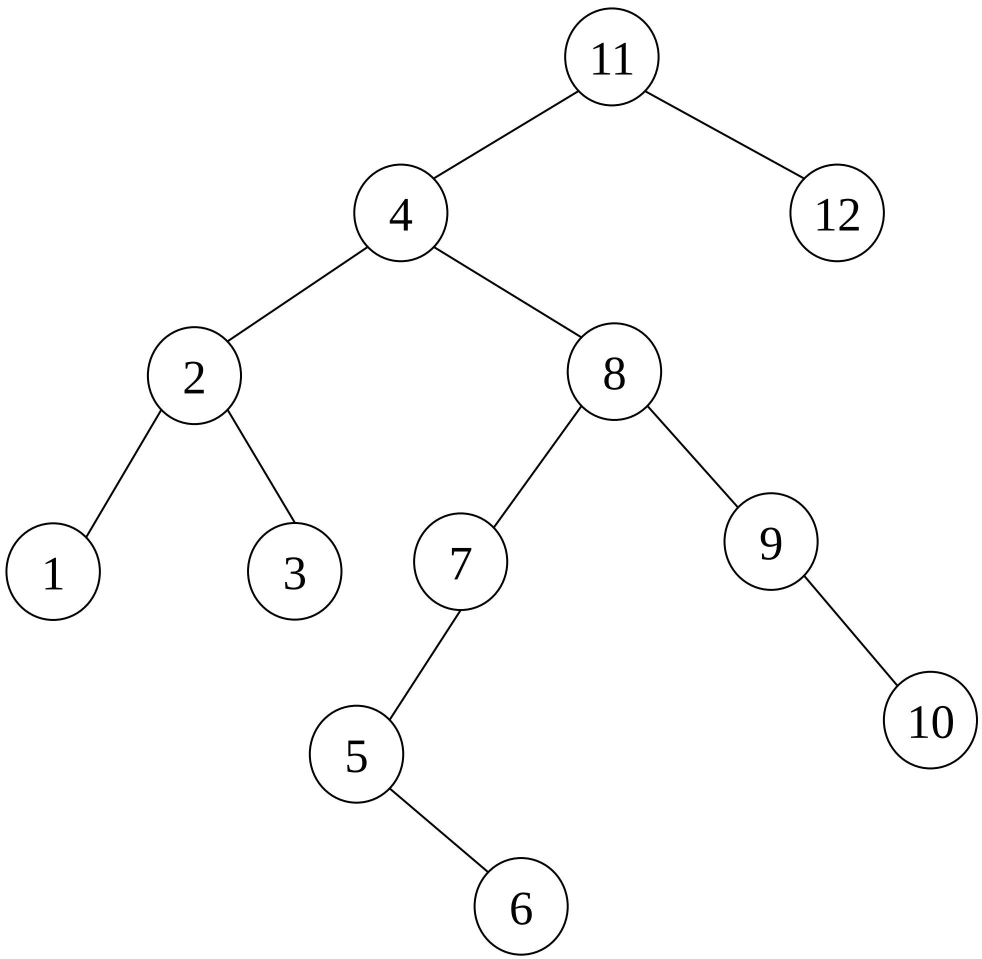
p: r의 부모 노드 (삭제하고자 하는 노드 r이 루트노드인 경우 제외)

```
treeDelete(r, p) ▷p의 자식 r을 삭제
{
    if (root = r) then                    ▷ r이 루트 노드
        root ← deleteNode(r);
    else if (r = p.left)  then            ▷ r이 p의 왼쪽 자식
        p.left ← deleteNode(r);
    else                                  ▷ r이 p의 오른쪽 자식
        p.right ← deleteNode(r);
}
deleteNode(r)   ▷ r을 삭제하고 r 대신 r의 부모에 연결할 노드를 리턴
{ …}
```

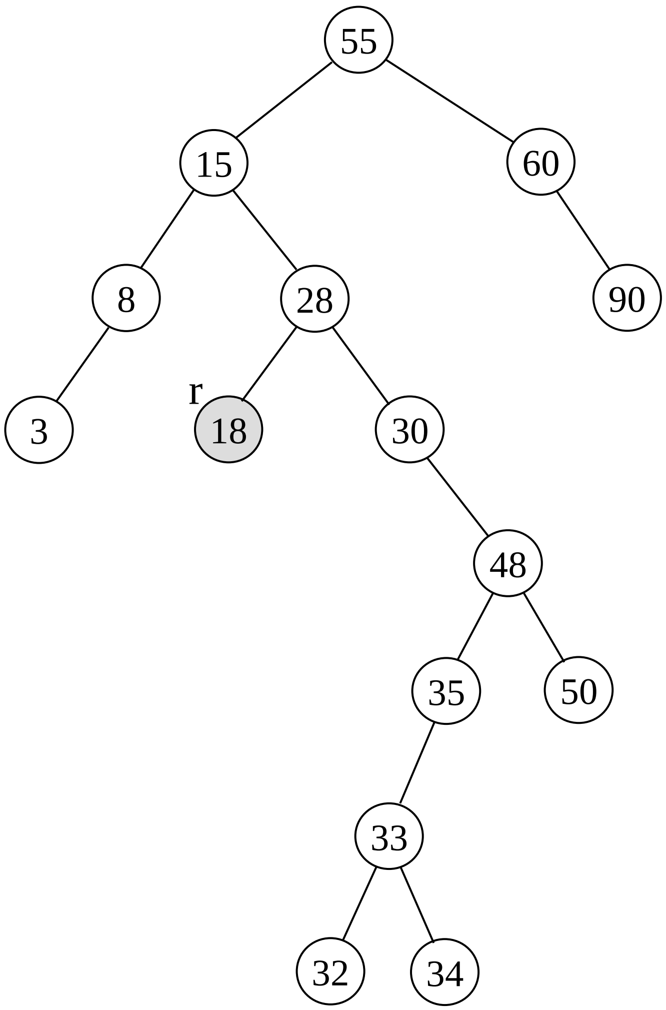
BST 삭제 알고리즘

```
deleteNode(r)   ▷ r을 삭제하고 r 대신 r의 부모에 연결할 노드를 리턴
{
    if (r.left = r.right = NIL) then return NIL;                        ▷ Case 1
    else if (r.left = NIL and r.right ≠ NIL) then return r.right;       ▷ Case 2-1
    else if (r.left ≠ NIL and r.right = NIL) then return r.left;        ▷ Case 2-2
    else {                                                              ▷ Case 3
        s ← r.right;
        while (s.left ≠ NIL) {    ▷ r의 right subtree 중에서 최소인 s 찾음
            parent ← s;
            s ← s.left;
        }
        ▷ s의 내용을 r에 복사한 후, s를 삭제
        r.key ← s.key;
        if (s = r.right) then r.right ← s.right;
        else parent.left ← s.right;
        return r;
    }
}
```

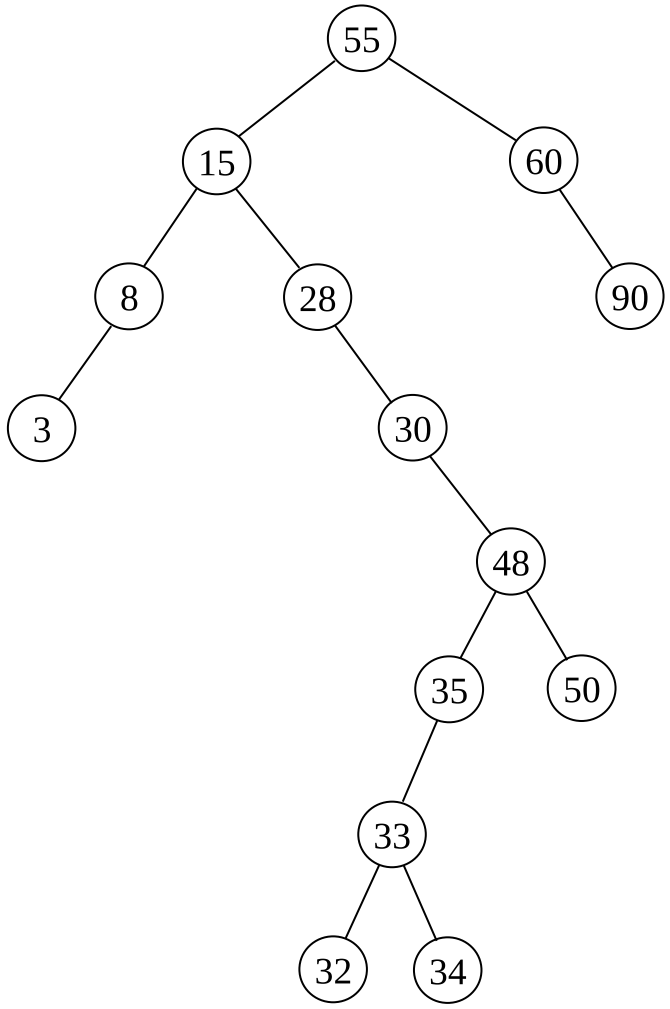
BST 삭제 알고리즘



BST 삭제 예 : Case 1

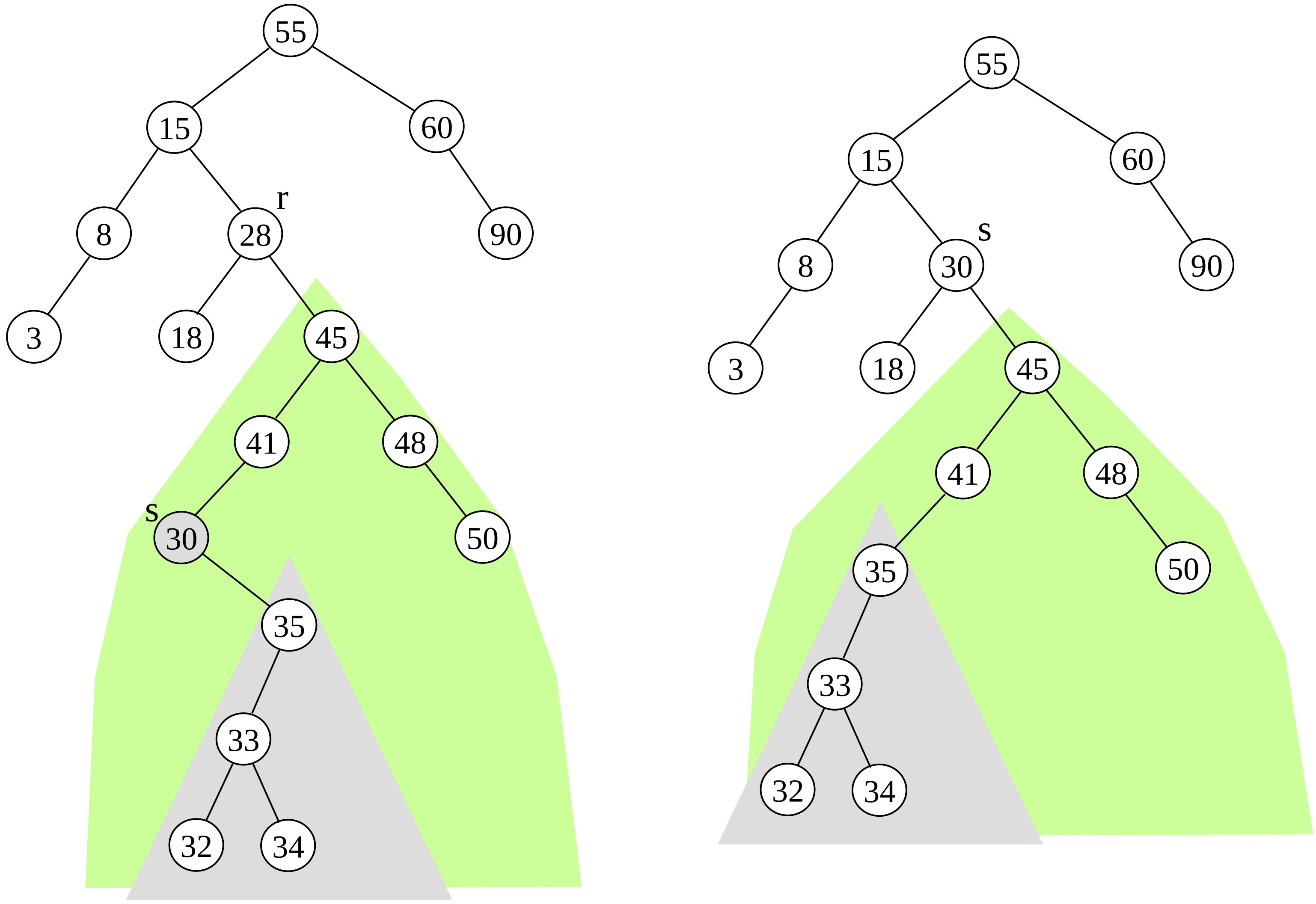


(a) r의 자식이 없음



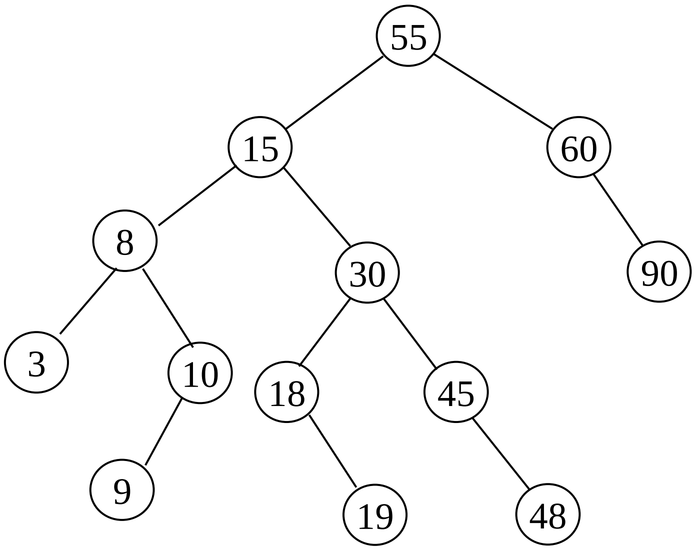
(b) 단순히 r을 제거한다

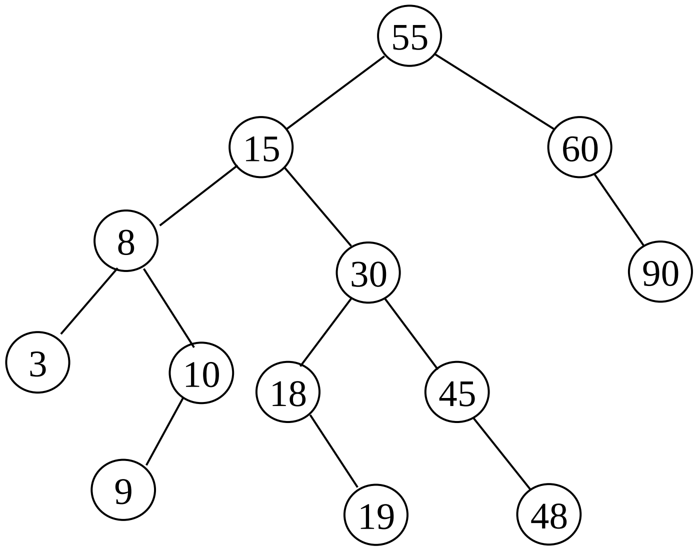
BST 삭제 예 : Case 3



r의 직후원소 s를 찾는다

BST 삭제 예 : Case 3



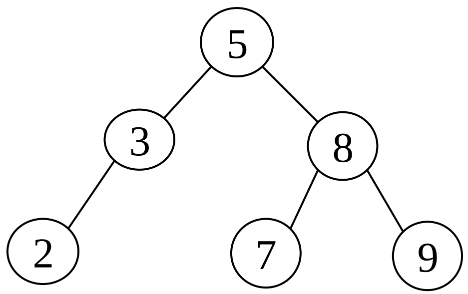


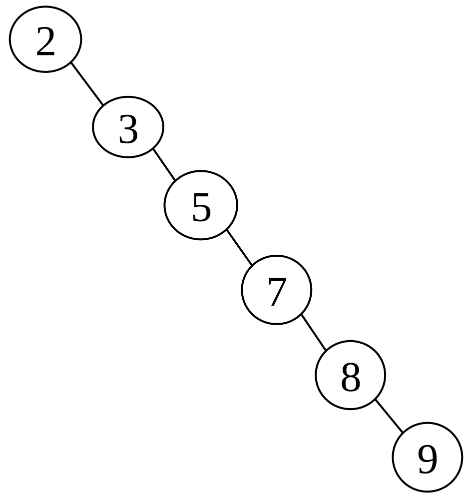
이진검색트리의 성능

- 높이가 h인 이진트리의 노드수 n은 $(h+1) \sim (2^{h+1}-1)$
  - 한 레벨에 최소한 한 개의 노드가 있어야 하므로 레벨 i의 노드수는 최소 1
    - 높이 h인 이진 트리의 최소 노드수 $n = h+1$
  - 자식노드가 최대 2개이므로 레벨 i의 노드수는 최대 2^i
    - 높이 h인 이진 트리의 최대 노드수 $n = 2^0+2^1+\ldots+2^h = 2^{h+1}-1$

이진검색트리의 성능

- 노드 수 n, 높이 h 인 이진검색트리에서 하나의 노드 검색/삽입/삭제 연산의 수행시간은 O(h)
  - 평균적으로는 O(log *n*)이지만
  - 최악의 경우는 O(*n*) – 균형이 많이 깨지는 경우. 예를 들어 오른쪽 아래 그림과 같은 우편향 트리





➔ **레드블랙트리, AVL 트리** 와 같은 균형 잡힌 트리(balanced tree)를 이용하면 최악의 경우 O(log *n*)의 성능을 얻을 수 있음

요약

- 검색트리는 데이터를 저장/검색/삭제하는 자료구조이다.
- 검색트리 중에서, 원소 수 n인 이진검색트리에서 원소 하나 저장/검색/삭제는
  - 평균 $\Theta(\log n)$의 시간이 걸린다.
  - 트리의 균형이 나쁘면 최악의 경우 $\Theta(n)$의 시간이 걸릴 수도 있다.
  - 최악의 경우에도 $\Theta(\log n)$ 시간이 보장되도록 하려면 균형 잡힌 검색트리를 이용해야 한다.

학습내용

1. 레코드, 키의 정의 및 검색 트리
2. 이진 검색 트리
3. **레드 블랙 트리**
4. B-트리
5. 다차원 검색 트리

Red-Black Tree (RB Tree)

- 레드 블랙 트리는 균형잡힌 이진 검색 트리
  - Binary search tree에 몇가지 조건을 추가하여 balanced tree가 되도록 변경시킨 것
  - 트리의 높이가 O(log *n*), 단 *n* 은 노드 수
  - 검색/삽입/삭제 연산의 수행시간은 O(log *n*)

- 참고: 다음을 구분해보세요.
  - 트리
  - 이진트리
  - 이진검색트리
  - 균형잡힌 이진검색트리

Red-Black Tree

- Red-Black tree는 binary search tree의 모든 노드에 red 또는 black의 색을 칠하되 다음과 같은 **레드 블랙 특성**을 만족해야 한다.

① 루트는 블랙이다.

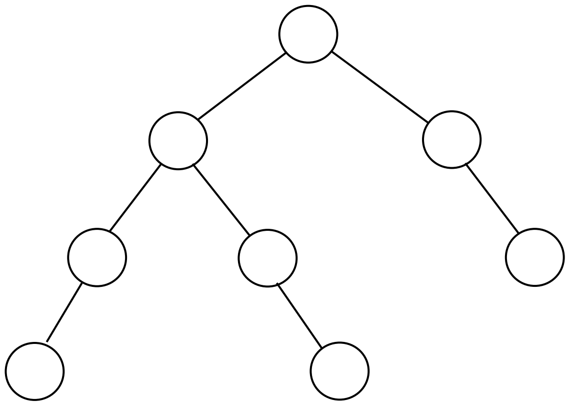
② 모든 리프(NIL)는 블랙이다.

③ 노드가 레드이면 그 노드의 자식은 반드시 블랙이다.

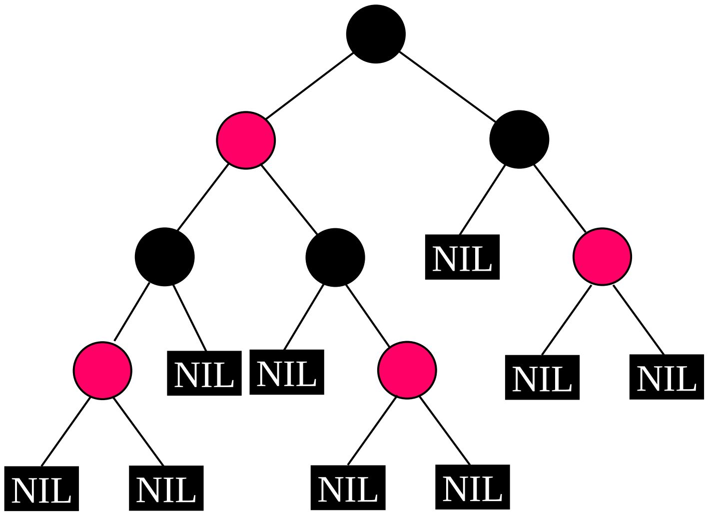
④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다.

✓ 여기서 리프 노드는 일반적인 의미의 리프 노드와 다르다.
NIL 포인터는 NIL 노드라는 블랙 리프 노드를 가리키는 것으로 본다.

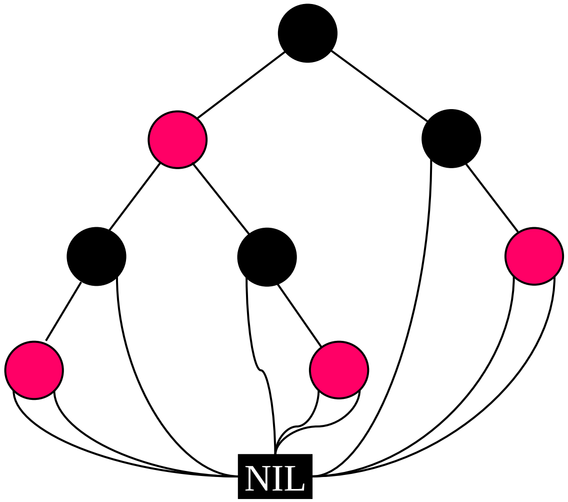
BST를 RB Tree로 만든 예



(a) BST의 예



(b) (a)를 RB Tree로 만든 예



(c) 실제 구현시 NIL 노드 처리 방법

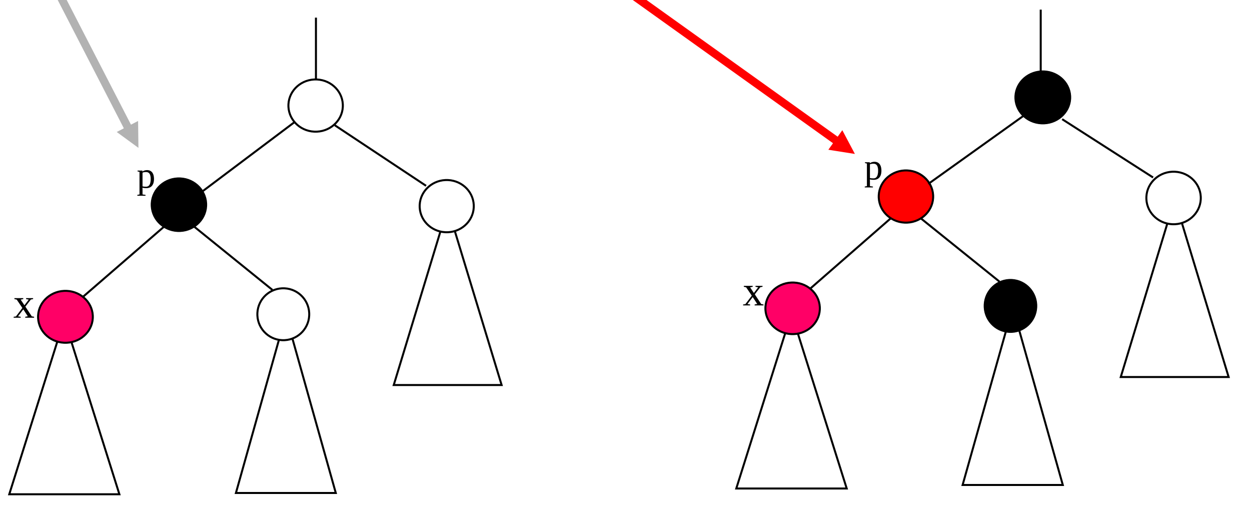
이진검색트리 회전

Red-Black Tree

- Red-Black tree 연산
  - 검색
    - 트리의 구조를 변경하지 않으므로 이진 검색 트리의 검색과 동일하다.
  - 삽입/삭제
    - 이진 검색 트리의 삽입/삭제와 기본적으로 동일한 방식으로 수행하되,
    - 삽입/삭제로 인해 레드 블랙 특성이
      - 깨지지 않으면 그대로 연산을 완료하고,
      - 깨지는 경우, 적절한 작업을 수행하여 레드 블랙 특성을 만족하도록 바로 잡는다.

RB Tree 삽입

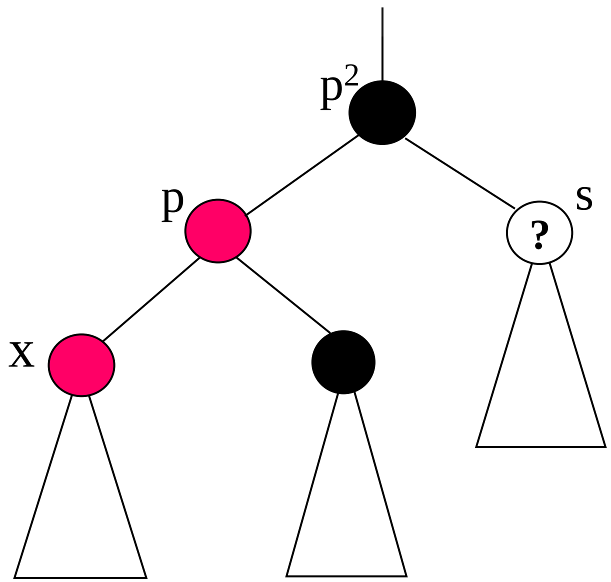
- 이진 검색 트리에서의 삽입 방법으로 노드를 삽입하되, 새로 삽입된 노드를 레드로 칠한다. (이 노드를 x라 하자)
  - 단, 비어있는 트리에 처음 삽입되는 노드는 루트 노드이므로 블랙으로 칠한다.
- 만일 x의 부모 노드 p의 색상이
  - 블랙이면 아무 문제 없다.
  - 레드이면 레드블랙특성 ③이 깨진다. --- (상황A)



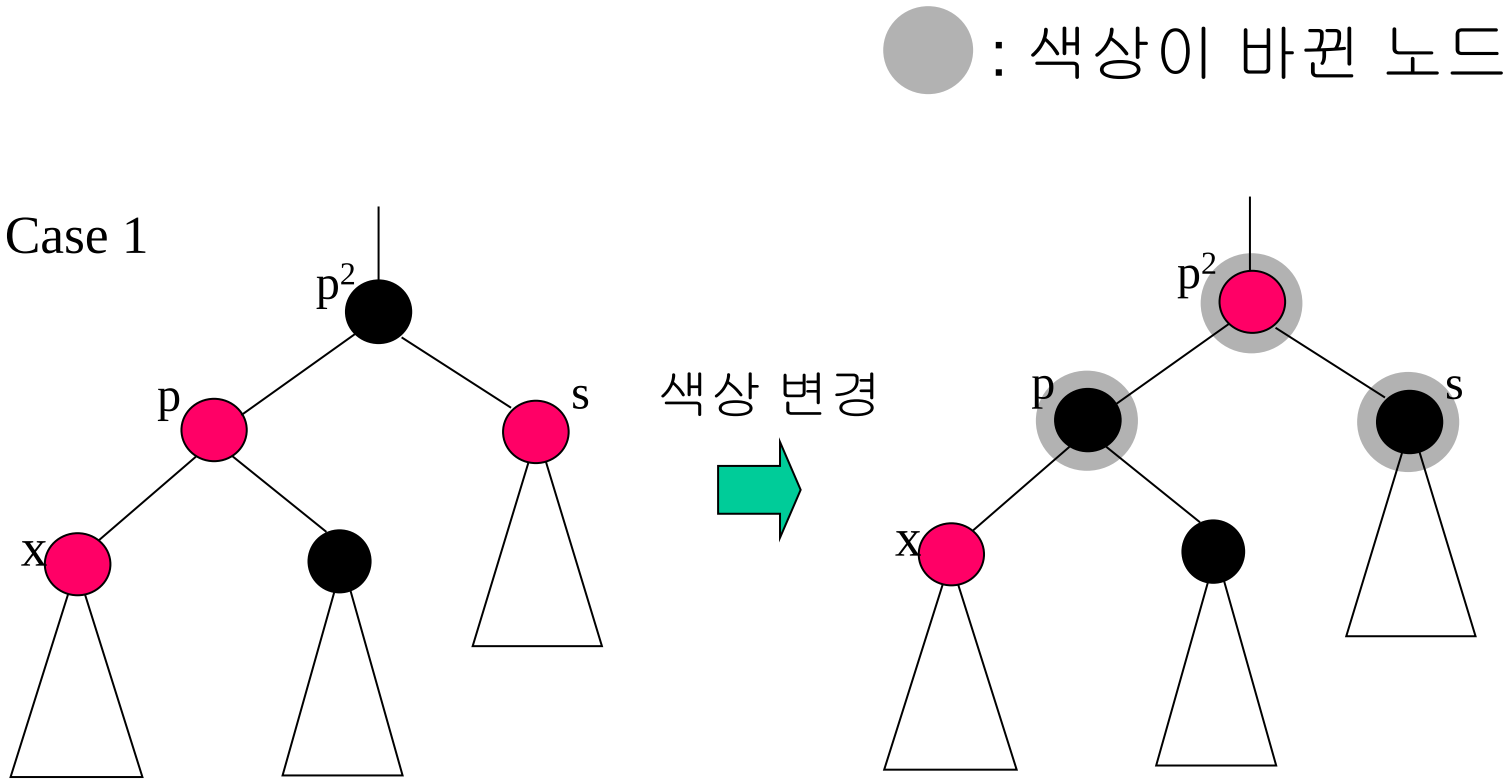
✓ 그러므로 p가 레드인 경우만 고려하면 된다.

주어진 조건: p is red

- p^2는 반드시 블랙이다.
- x의 형제 노드는 반드시 블랙이다.
- s의 색상에 따라 두 가지로 경우로 나눈다.
  - Case 1: s가 레드
  - Case 2: s가 블랙

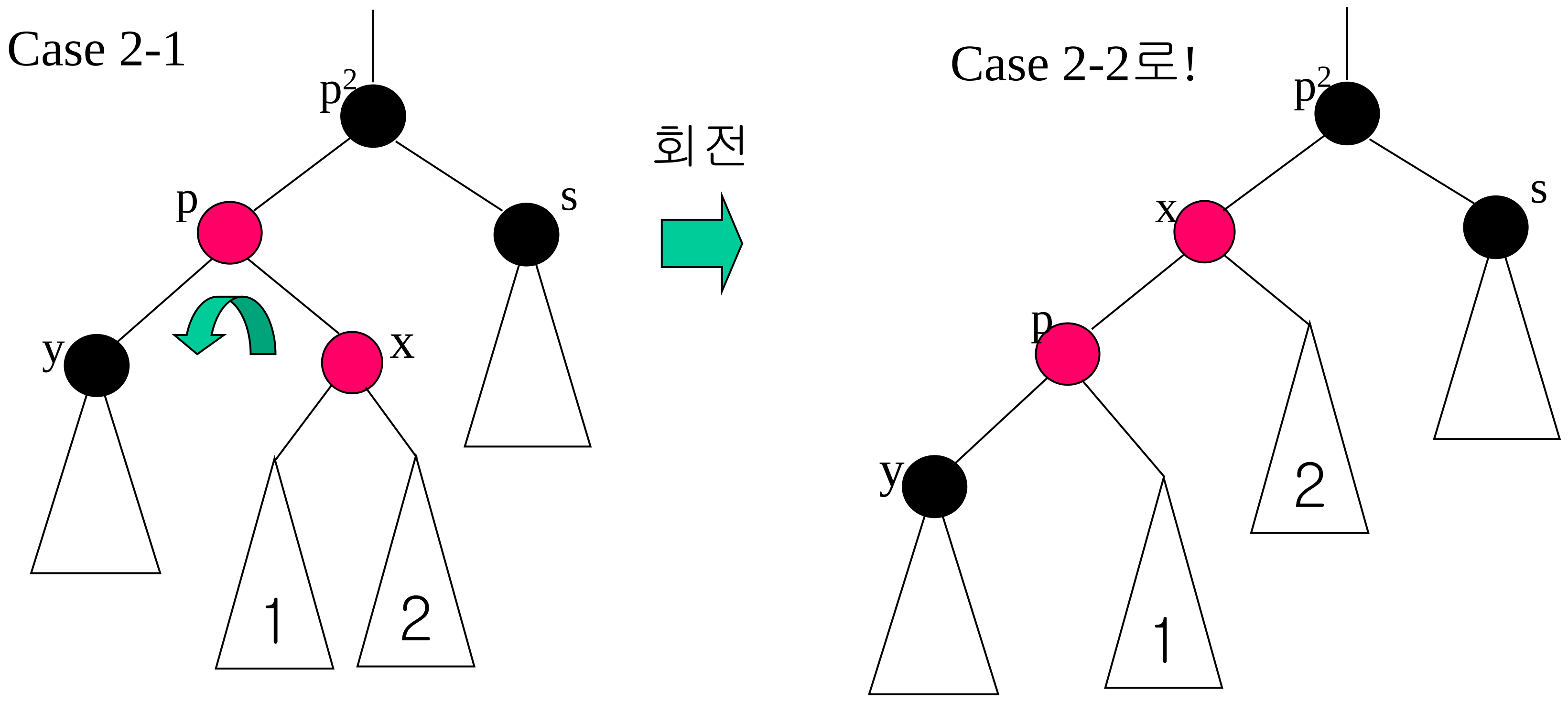


Case 1: s가 레드



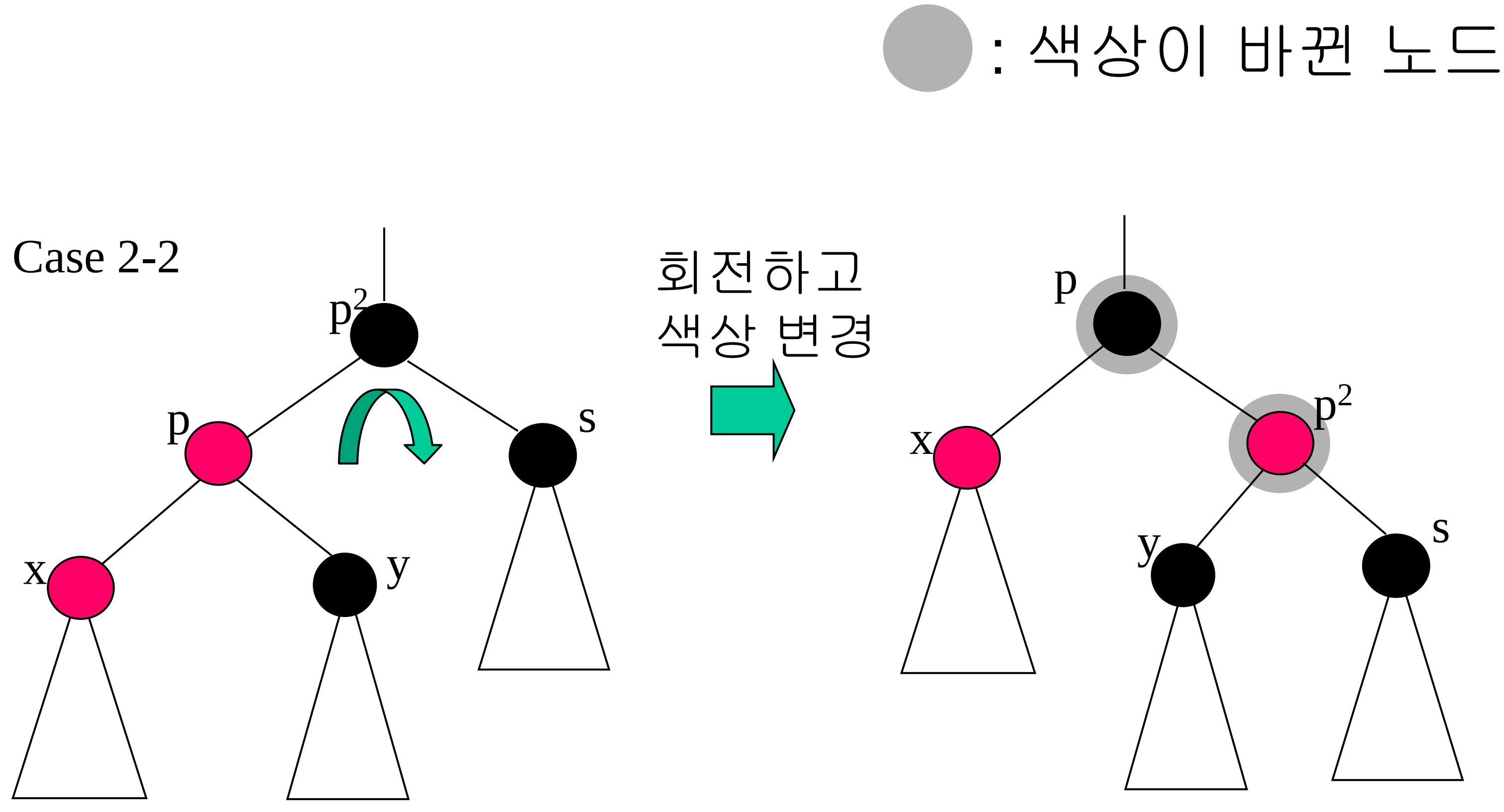
- ✓ p^2에서 조금 전(상황A)과 같은 문제가 발생할 수 있다: recursive problem!
  즉, p^2 의 부모가 블랙이면 삽입 연산 완료이지만,
  p^2 의 부모가 레드이면 p^2 를 x로 삼아 재귀적으로 문제를 해결해야 한다.

Case 2-1: s가 블랙이고, x가 p의 오른쪽 자식



✓ p가 p^2 의 오른쪽 자식인 경우는 위의 설명과 대칭적으로 처리하면 된다.

Case 2-2: s가 블랙이고, x가 p의 왼쪽 자식



✓삽입 완료!

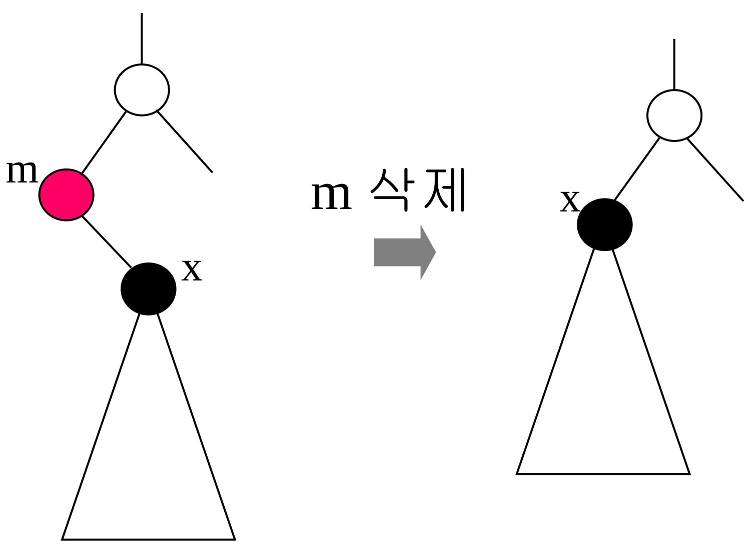
예) 다음과 같은 순서로 삽입되는 경우

1 2 3 4 10 8 9 7 5 6

RB Tree 삭제

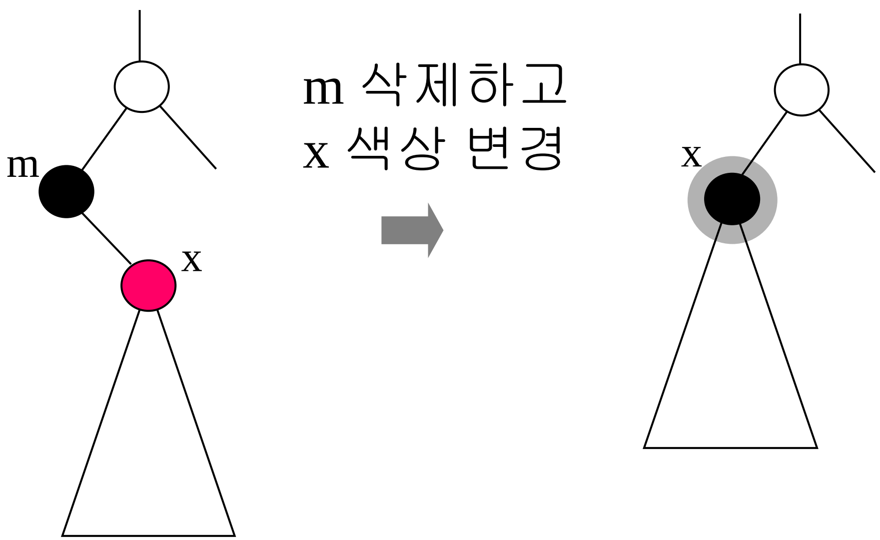
- 기본적으로 이진 검색 트리에서의 삭제 방법으로 노드를 삭제하되 필요한 경우 색상을 조정한다.
  - 삭제 노드의 자식이 없거나 1개만 가진 노드로 제한해도 됨 (삭제 노드를 m, 자식 노드를 x라 하자)
  - 삭제 노드에 대해 3가지로 나누어 처리한다. (1)~(3)

(1) 삭제 노드가 레드이면 간단히 삭제된다.



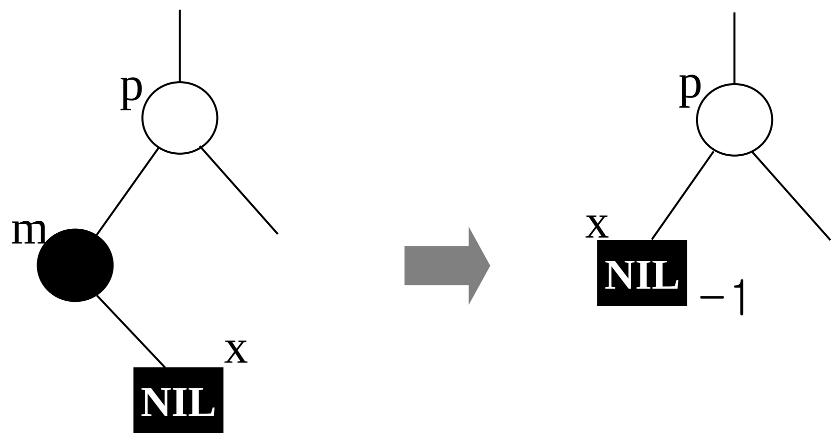
RB Tree 삭제

(2) 삭제 노드가 블랙이라도 (유일한) 자식이 레드이면 간단히 삭제된다.



RB Tree 삭제

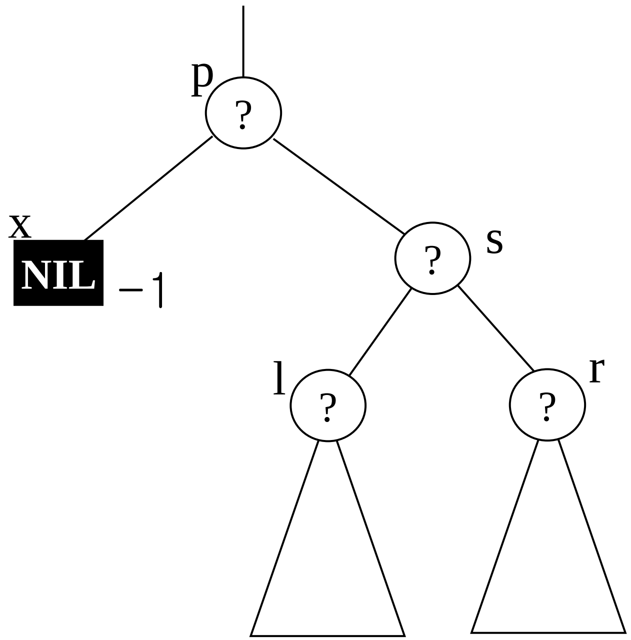
(3) 삭제 노드가 블랙이고 (유일한) 자식이 블랙인 경우는 간단히 삭제할 수 없다. (실제로 자식 노드가 모두 **NIL** 인 경우를 말한다.)



m을 제거한 후 문제 발생(레드블랙특성 ④ 위반)

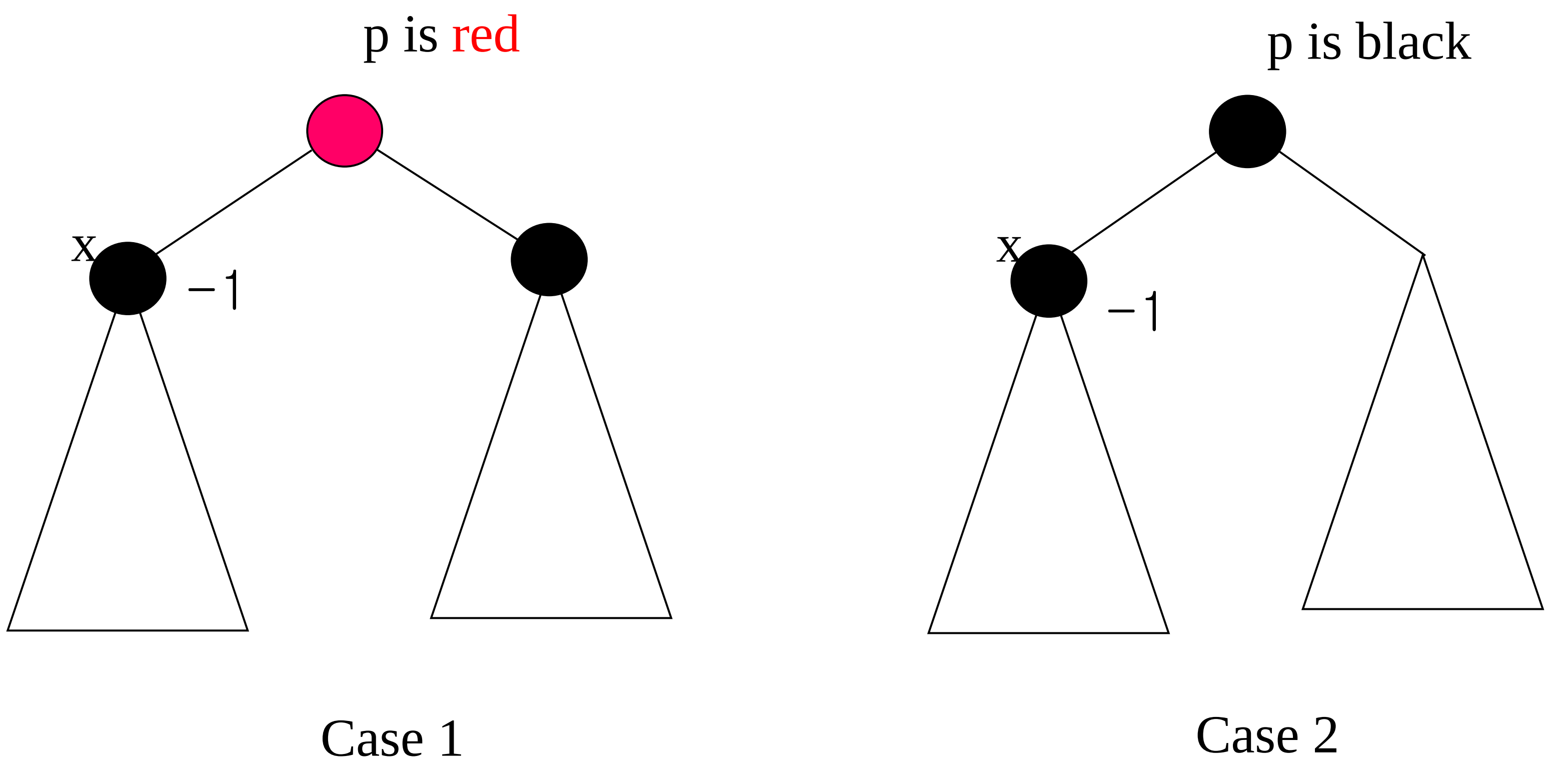
✓ x 노드 옆에 표시한 숫자 −1은 루트에서 x 를 통해 리프에 이르는 경로에서 블랙 노드의 수가 하나 모자람을 의미한다.

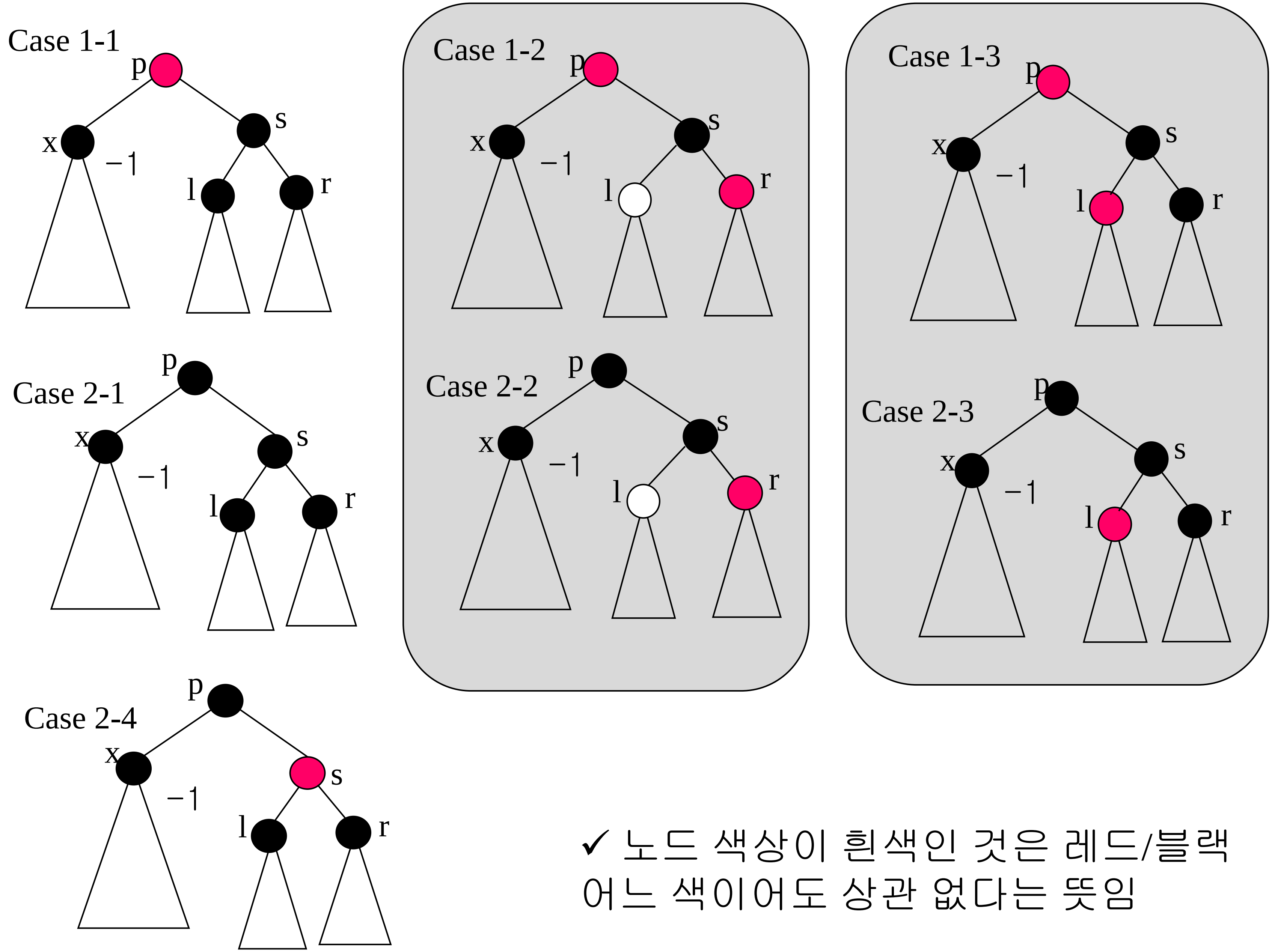
m을 제거한 후 x의 주변 상황에 따라 처리 방법이 달라짐



➔ 다음 세 슬라이드에서 Case 1-*, Case 2-* 로 나누어 설명한다.

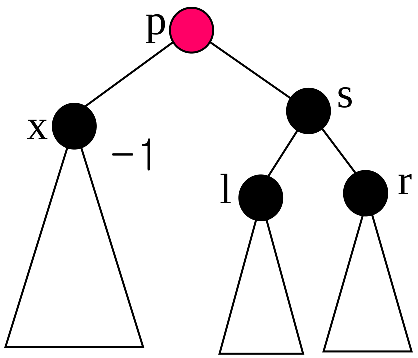
삭제 노드(m)가 블랙이고 (유일한) 자식(x)이 블랙인 경우
m을 삭제한 후 상황은 몇가지 경우로 나뉘는지 알아보자.



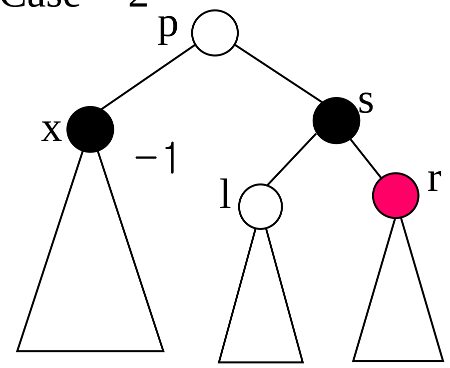


✓ 노드 색상이 흰색인 것은 레드/블랙 어느 색이어도 상관 없다는 뜻임

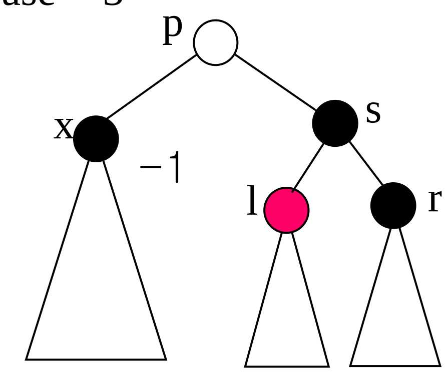
Case 1-1



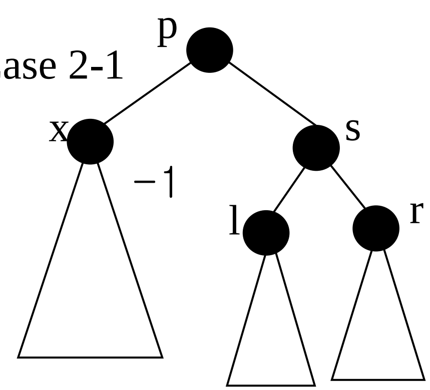
Case *-2



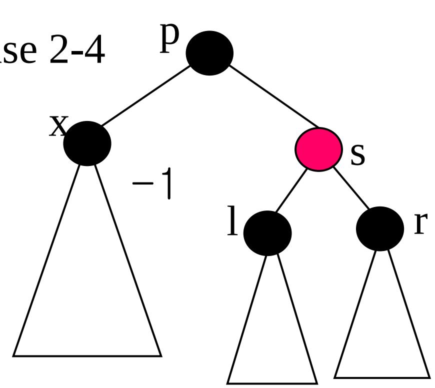
Case *-3



Case 2-1

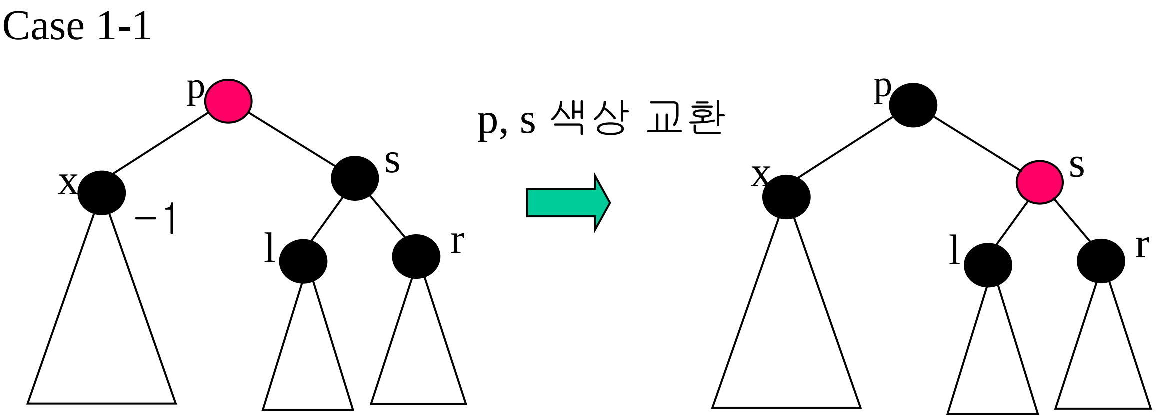


Case 2-4



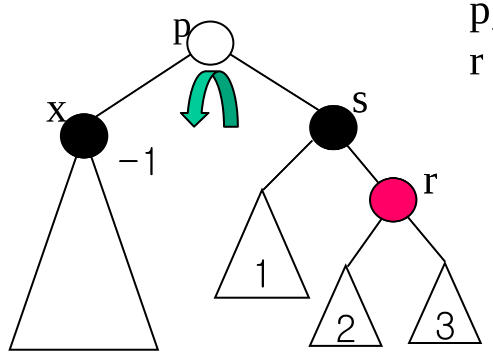
✓최종적으로 **5**가지 경우로 나뉜다.

각 경우에 따른 처리

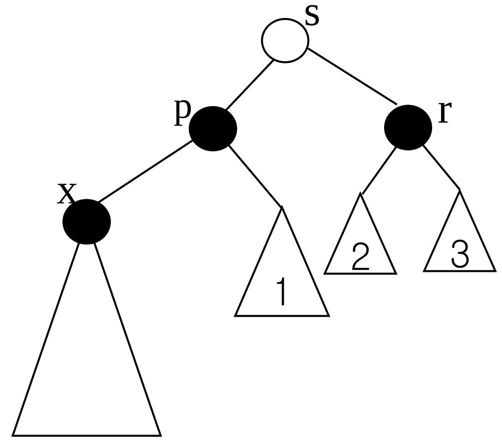


✓삭제 완료!

Case *-2

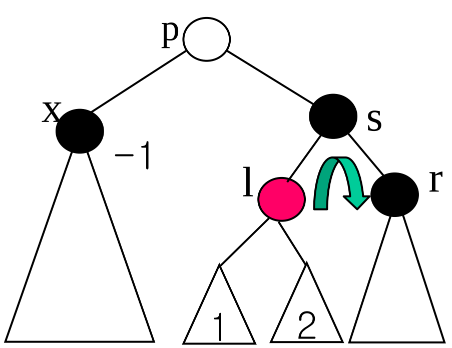


회전하고
p, s 색상 교환
r 색상 변경

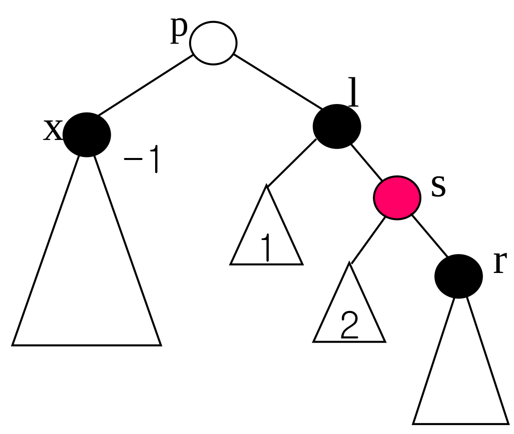


✓ 삭제 완료!

Case *-3

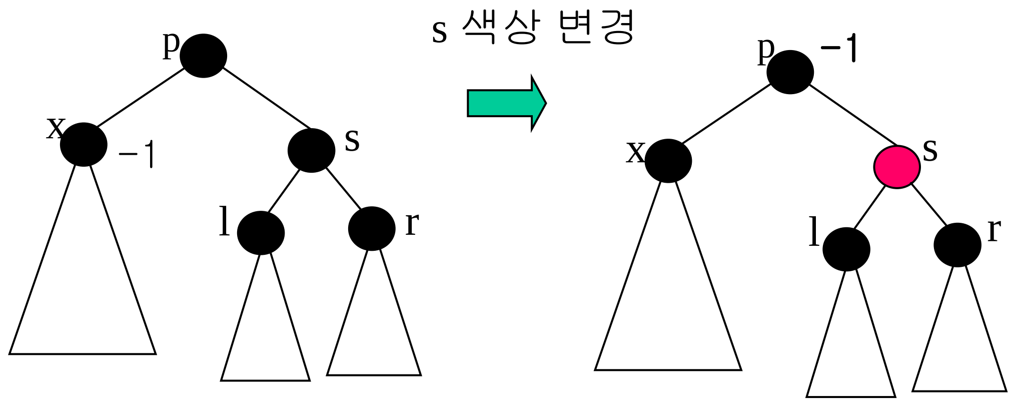


회전하고
l, s 색상 교환



Case *-2로

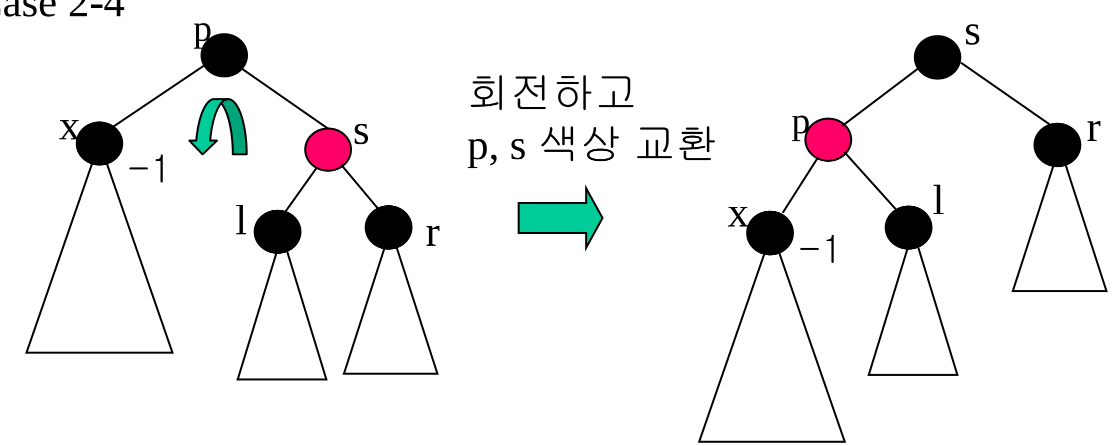
Case 2-1



✓ p가 -1이 되어 앞에서와 같은 문제가 발생 (recursive problem)

✓ p를 문제 발생 노드(x)로 하여 재귀적으로 다시 처리 시작

Case 2-4



Case 1-1,
Case 1-2,
Case 1-3 중 하나로

요약

- 이진 검색 트리(binary search tree)
  - 한 원소의 저장/검색/삭제에 평균 $\Theta(\log n)$, 최악 $\Theta(n)$ 시간이 든다.
- 균형 잡힌(balanced) 이진 검색 트리
  - 한 원소의 저장/검색/삭제에 최악의 경우에도 $\Theta(\log n)$ 시간이 보장된다.
  - 예) 레드-블랙 트리

학습내용

1. 레코드, 키의 정의 및 검색 트리
2. 이진 검색 트리
3. 레드 블랙 트리
4. **B-트리**
5. 다차원 검색 트리

B-Tree

- 검색트리가 방대하면 모두 메모리에 올려놓고 사용할 수 없으므로 디스크에 있는 상태로 작업해야 함
  - CPU 작업의 효율성보다 디스크 접근 횟수가 효율을 좌우
    - 디스크의 접근 단위는 블록(페이지)
    - 디스크에 한 번 접근하는 시간은 수십만개의 명령어의 처리 시간과 맞먹음
- 검색트리가 디스크에 저장되어 있다면(=외부 검색트리) 트리의 높이를 최소화하는 것이 유리
  - 검색트리의 분기수 늘리면(=다진 검색트리) 트리의 높이가 줄어듬

    예) 10억개 내외의 키

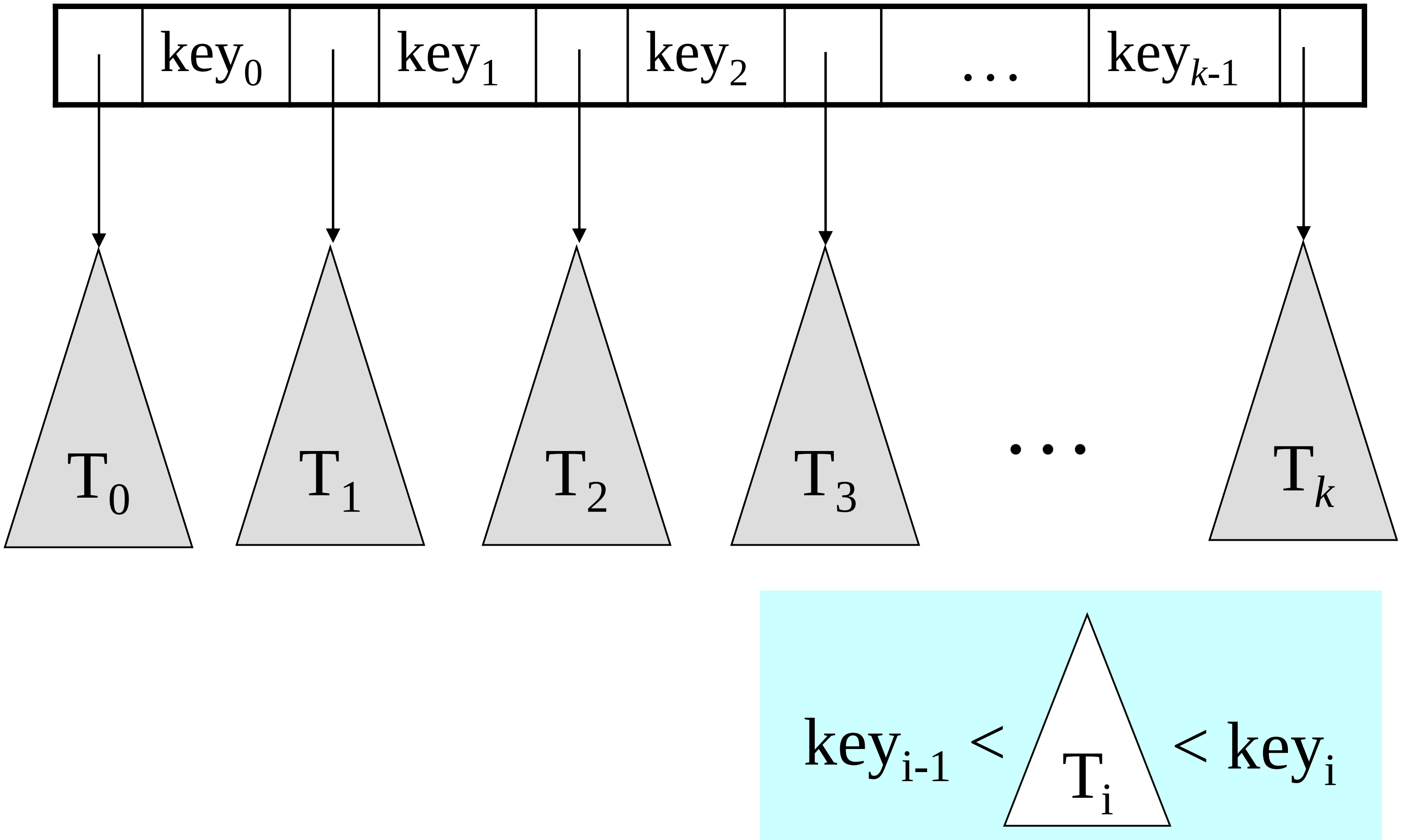
    이진검색트리 → 높이 30 정도

    256개의 분기를 지니는 트리 → 높이 5 정도
  - 분기 수는 블록의 크기를 고려하여 결정

B-트리

- B-트리는
  - 외부 검색트리(external search tree)
  - 다진 검색트리(multi-way search tree)
  - 트리의 균형을 유지하여 최악의 경우 디스크 접근 횟수를 줄임 (balanced tree)

다진 검색트리

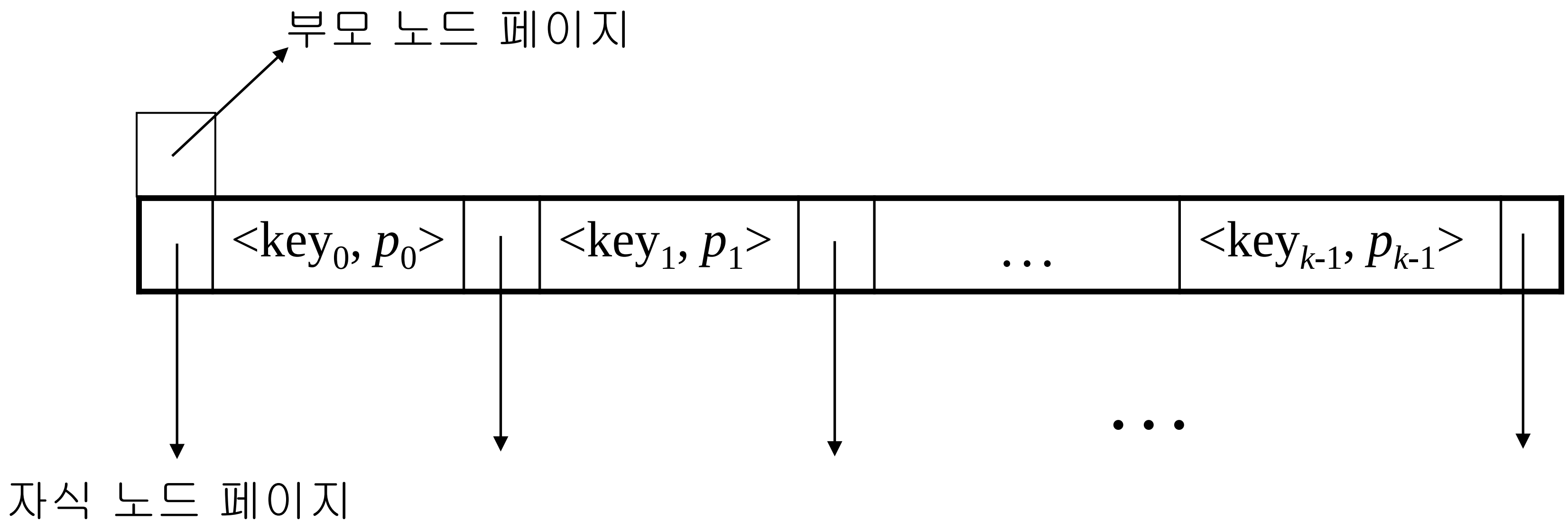


B-트리

- B-Tree는 균형잡힌 다진 검색트리로서 다음 성질을 만족
  - 루트를 제외한 모든 노드는 $\lfloor k/2 \rfloor \sim k$ 개의 키를 갖는다.
  - 모든 리프 노드는 레벨이 같다.
- 검색/삽입/삭제 연산의 수행시간은 $O(\log n)$
  - 균형잡힌 이진 검색 트리도 $O(\log n)$이지만,
  - B-트리의 경우, 이진 검색 트리에 비해 상수 인자가 상당히 작다.

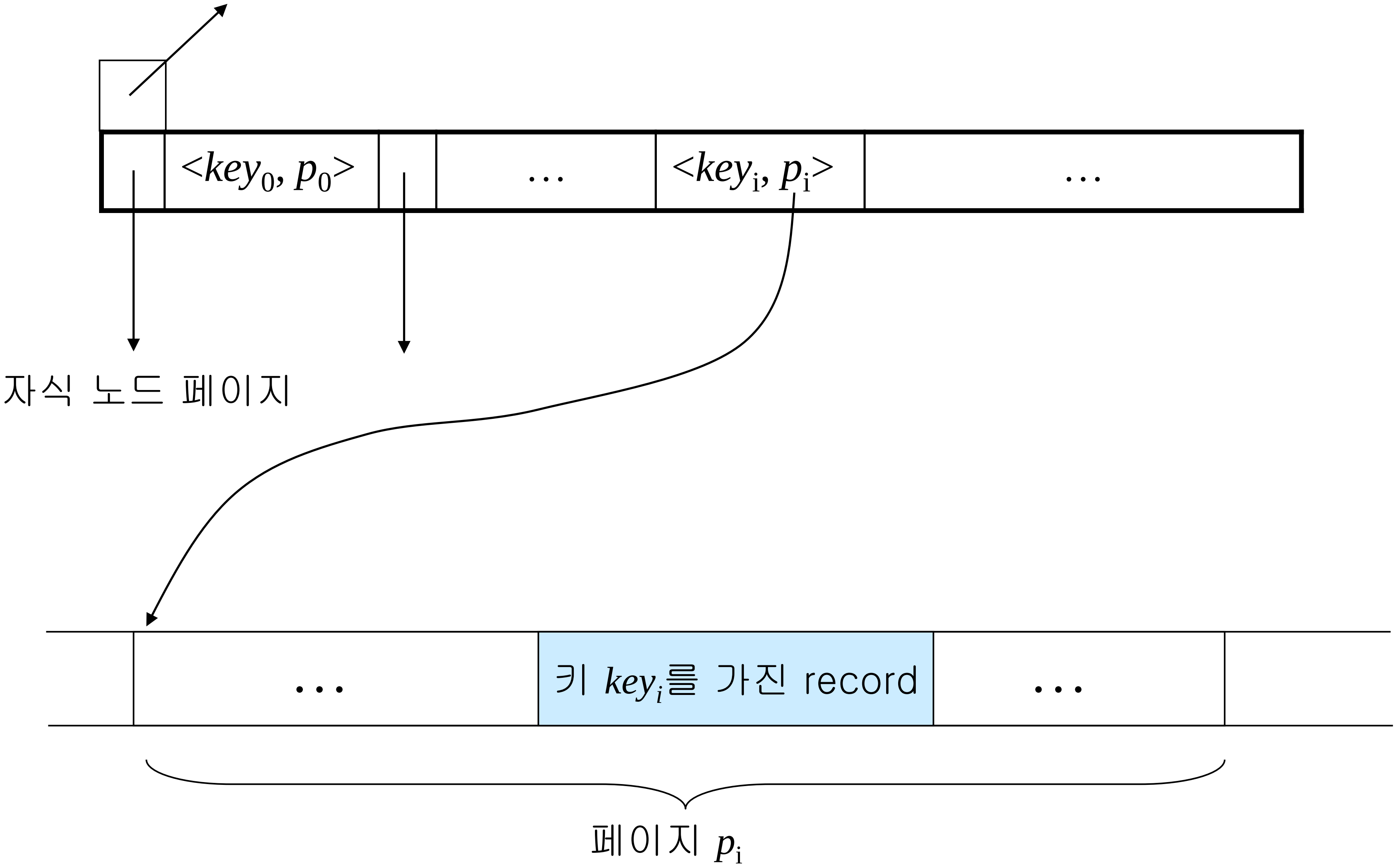
B-트리

- B-Tree의 노드 구조 – 하나의 페이지(블록)



- B-Tree에서 k의 값
  - 키와 분기 포인터 공간을 반영하여 디스크 한 블록이 수용할 수 있는 최대값을 잡음

B-트리에서 레코드에 접근하는 과정



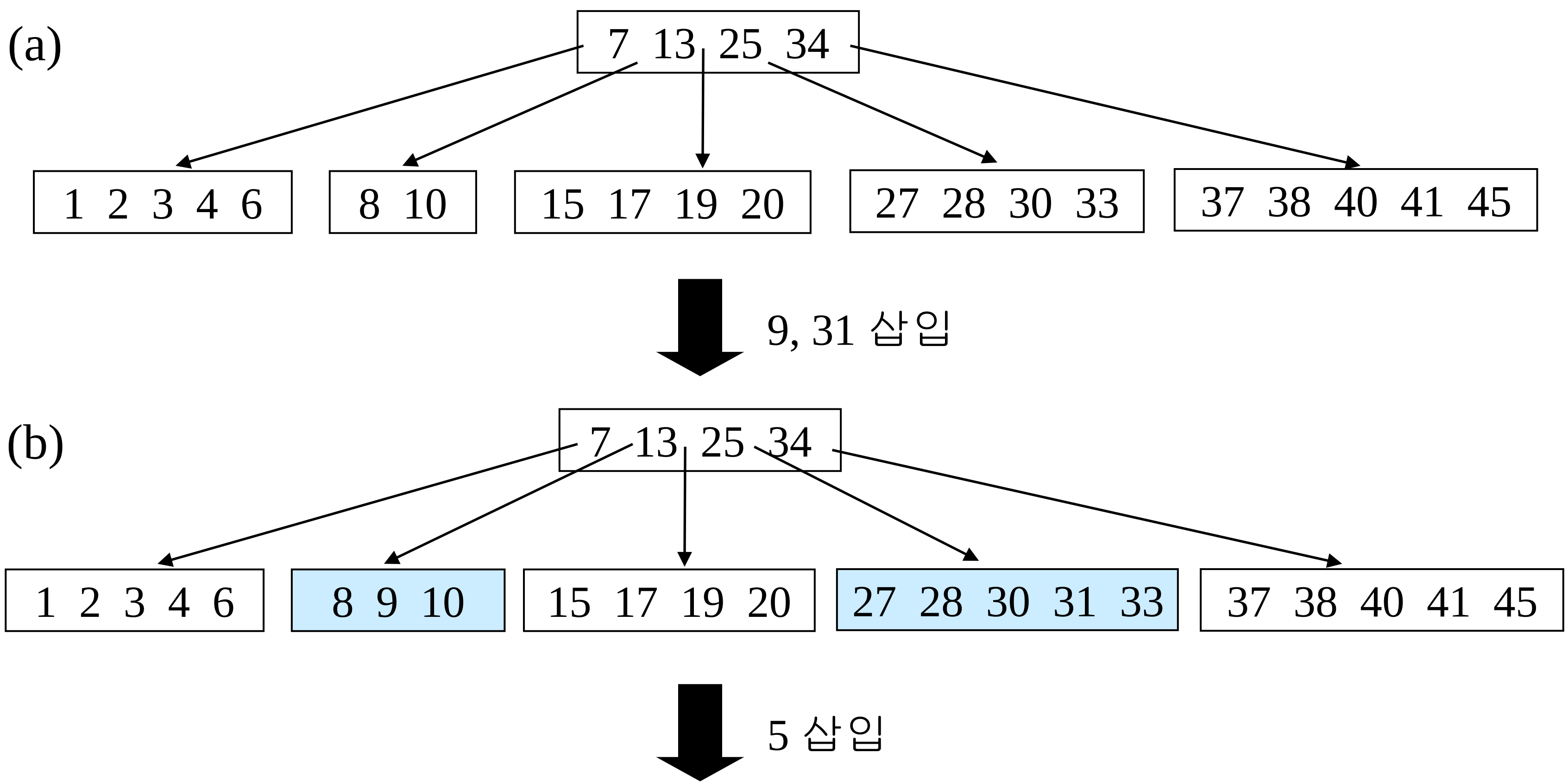
B-Tree 삽입 알고리즘

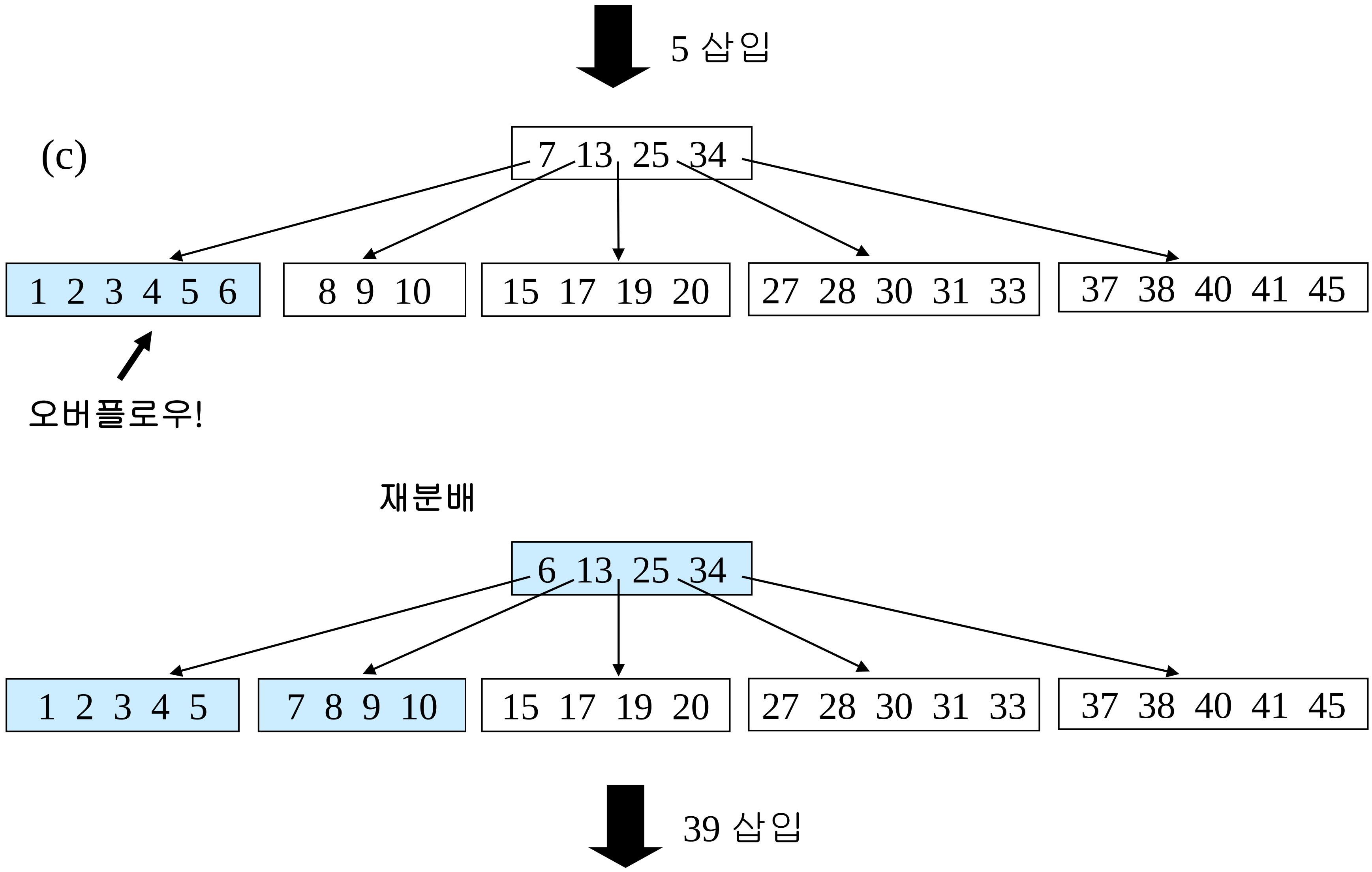
t : 트리의 루트 노드 x : 삽입하고자 하는 키

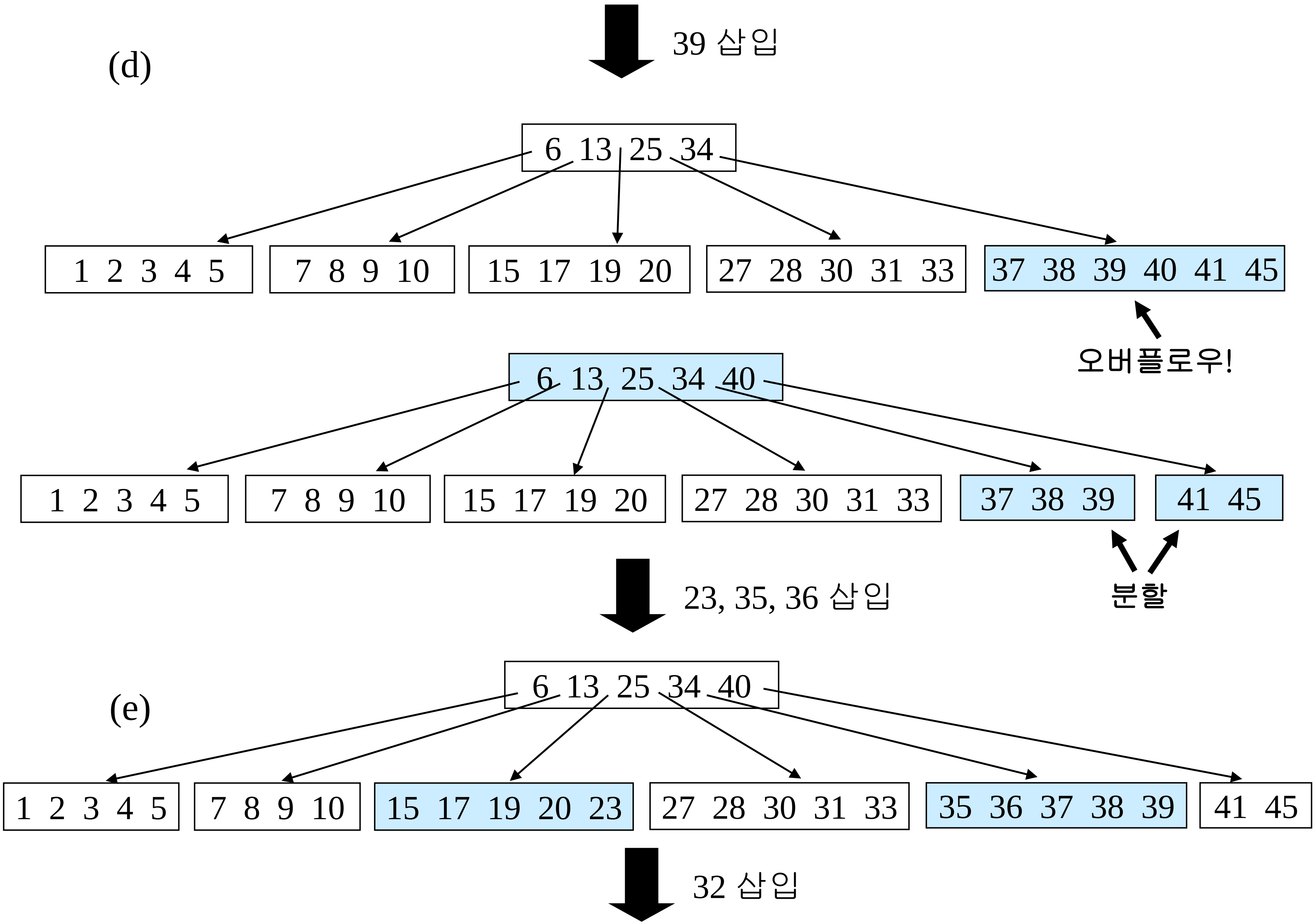
```
BTreeInsert(t, x)
{
        x를 삽입할 리프 노드 r을 찾음;
        x를 r에 삽입;

        if (r에 오버플로우 발생) then clearOverflow(r);
}

clearOverflow(r)
{
     if (r의 형제 노드 중 여유있는 노드가 있음) then
          r의 남는 키를 형제 노드에게 넘김;
     else {
          r을 둘로 분할하고 가운데 키를 부모 노드 p로 넘김;
          if (부모 노드 p에 오버플로우 발생) then clearOverflow(p);
     }
}
```

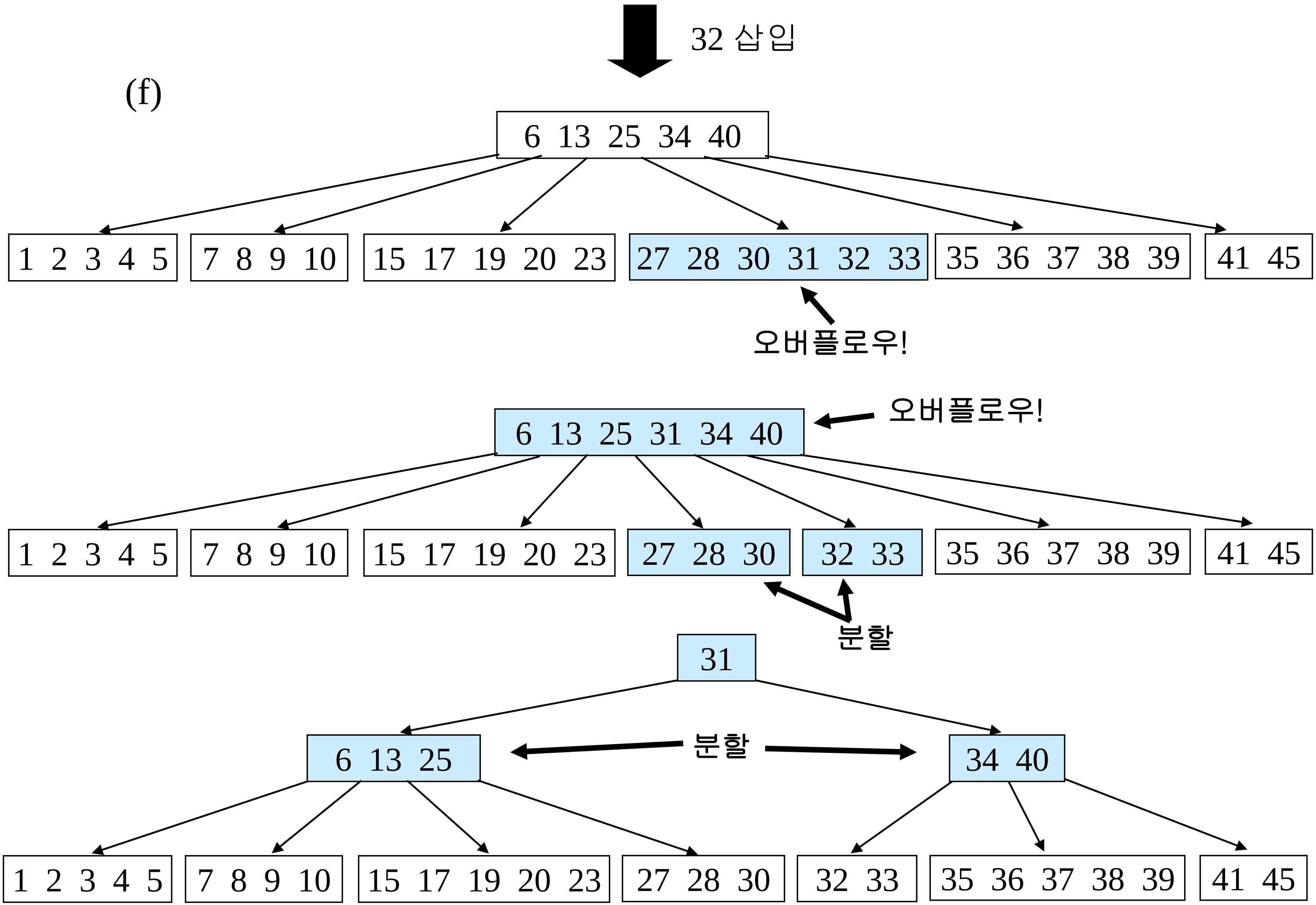
B-Tree 삽입 예

k = 5인 경우임 → 하나의 노드에 2~5개의 키가 저장됨




5 삽입
(c)
7 13 25 34
1 2 3 4 5 6
8 9 10
15 17 19 20
27 28 30 31 33
37 38 40 41 45
오버플로우!
재분배
6 13 25 34
1 2 3 4 5
7 8 9 10
15 17 19 20
27 28 30 31 33
37 38 40 41 45
39 삽입


(d)
39 삽입
6 13 25 34
1 2 3 4 5
7 8 9 10
15 17 19 20
27 28 30 31 33
37 38 39 40 41 45
오버플로우!
6 13 25 34 40
1 2 3 4 5
7 8 9 10
15 17 19 20
27 28 30 31 33
37 38 39
41 45
분할
23, 35, 36 삽입
(e)
6 13 25 34 40
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30 31 33
35 36 37 38 39
41 45
32 삽입

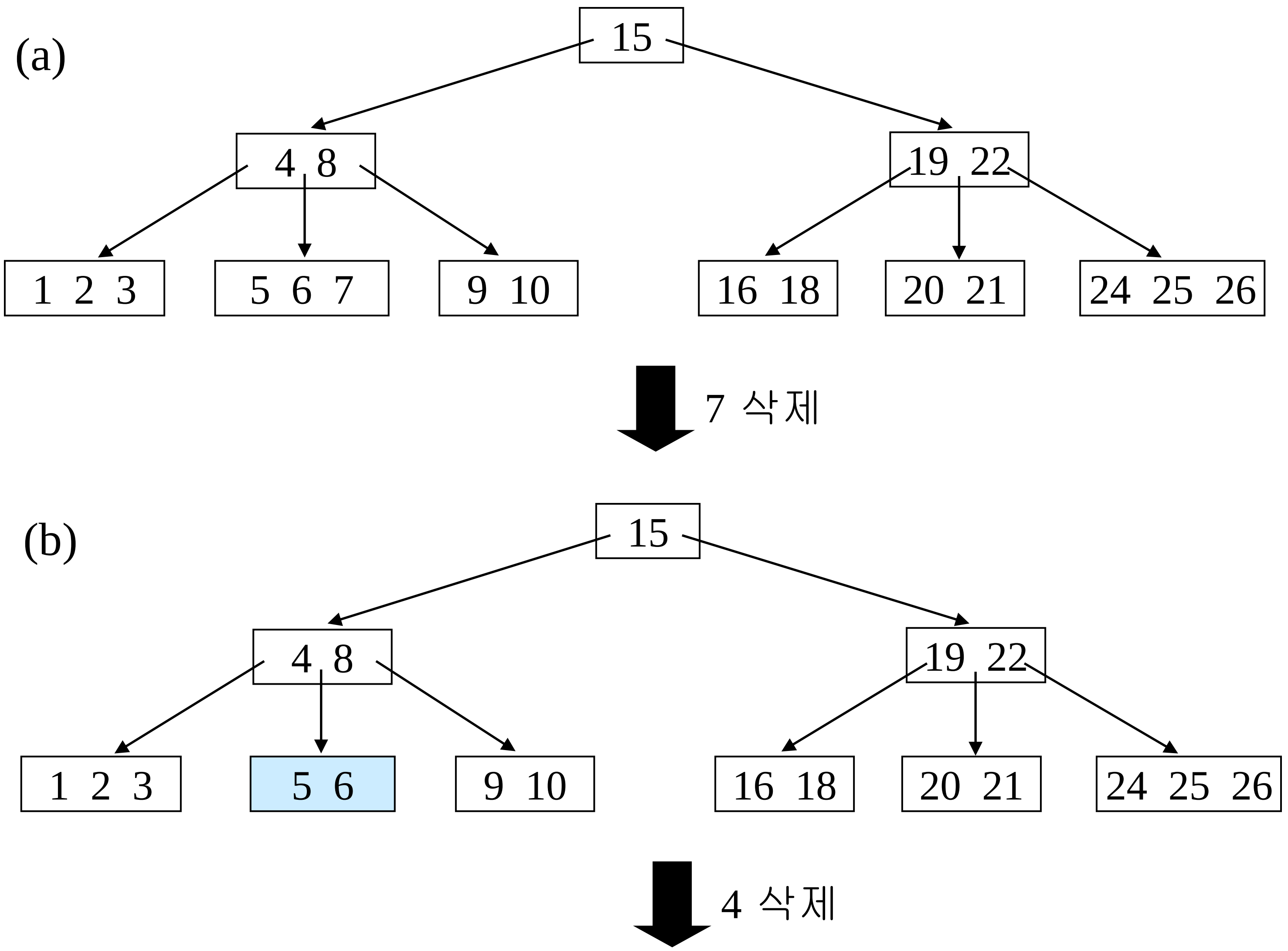

32 삽입
(f)
6 13 25 34 40
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30 31 32 33
35 36 37 38 39
41 45
오버플로우!
6 13 25 31 34 40
오버플로우!
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30
32 33
35 36 37 38 39
41 45
분할
31
6 13 25
분할
34 40
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30
32 33
35 36 37 38 39
41 45

B-Tree 삭제 알고리즘

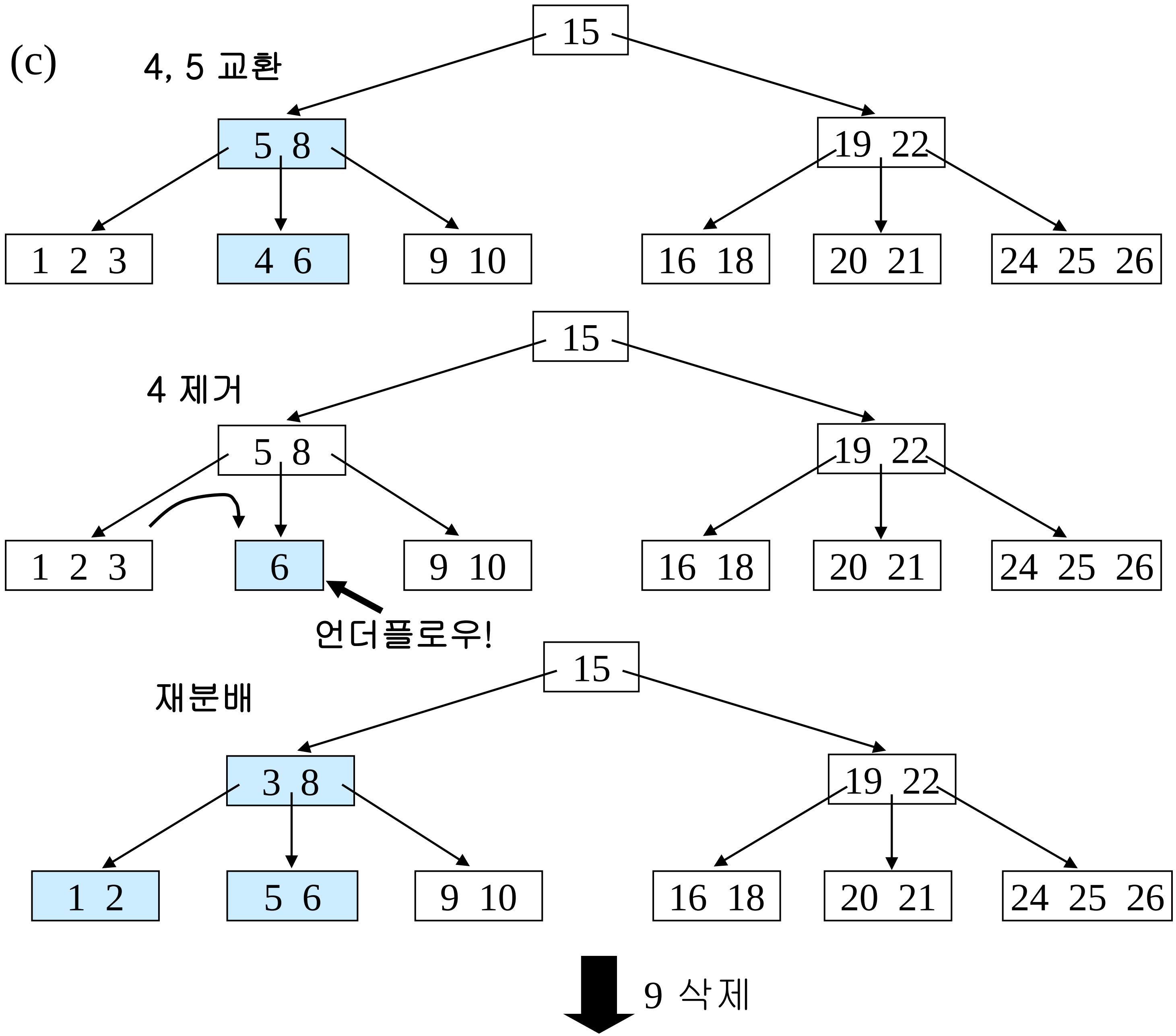
t : 트리의 루트 노드
x : 삭제하고자 하는 키
v : x를 갖고 있는 노드

```
BTreeDelete(t, x, v)
{
    if (v가 리프 노드 아님) then {
        x의 직후원소 y를 가진 리프 노드를 찾음;
        x와 y를 맞바꿈;
    }
    리프 노드에서 x를 제거하고 이 리프 노드를 r이라 함;
    if (r에서 언더플로우 발생) then clearUnderflow(r);
}

clearUnderflow(r)
{
    if (r의 형제 노드 중 키를 하나 내놓을 수 있는 여분을 가진 노드가 있음)
        then
            r이 형제 노드의 키를 넘겨받음;
        else {
            r의 형제 노드와 r을 병합하고 부모 노드에서 키를 하나 받음;
            if (부모 노드 p에 언더플로우 발생) then clearUnderflow(p);
        }
}
```

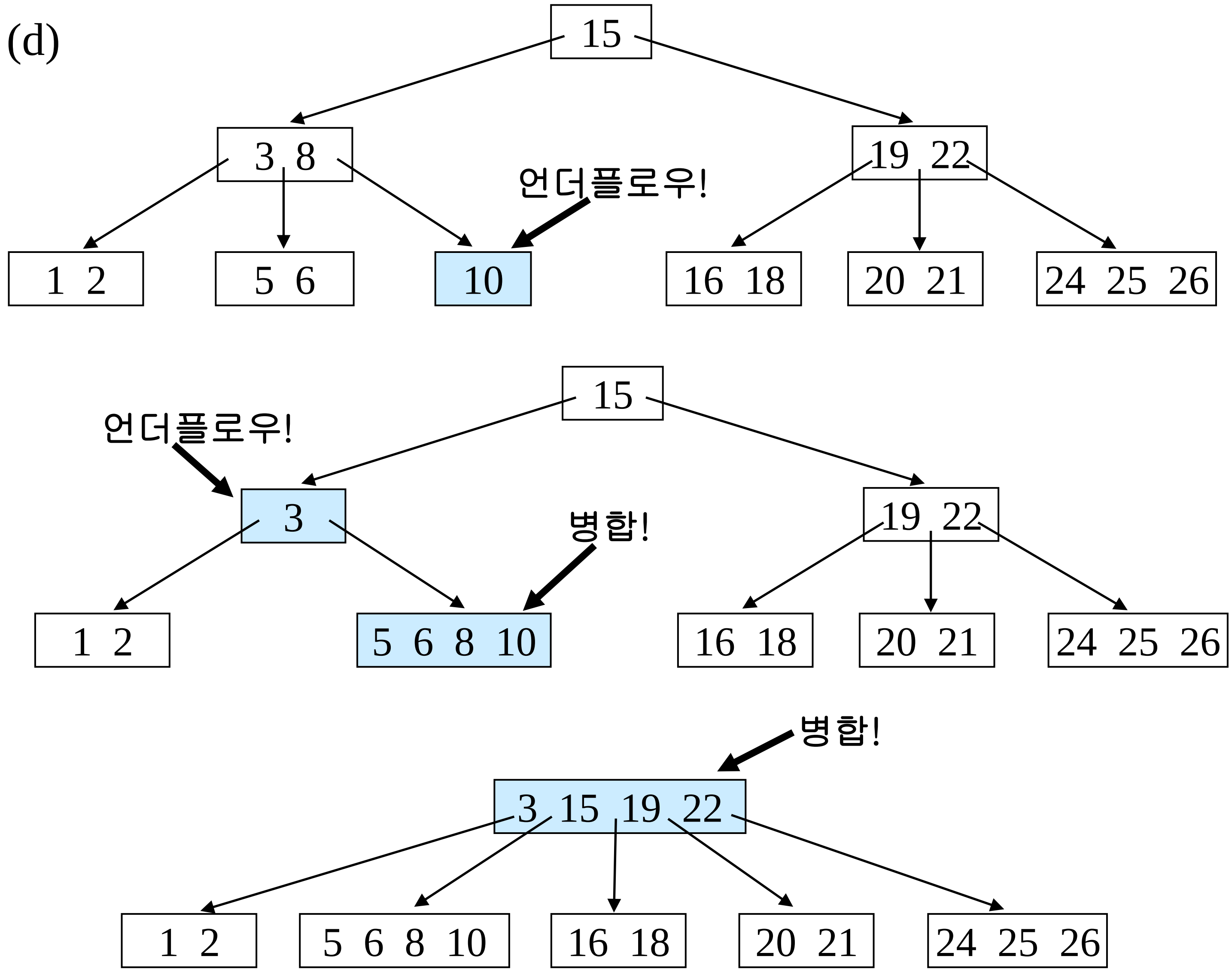
B-Tree 삭제 예



(a)
15
4 8
19 22
1 2 3
5 6 7
9 10
16 18
20 21
24 25 26
7 삭제
(b)
15
4 8
19 22
1 2 3
5 6
9 10
16 18
20 21
24 25 26
4 삭제


(c)
4, 5 교환
15
5 8
19 22
1 2 3
4 6
9 10
16 18
20 21
24 25 26
4 제거
15
5 8
19 22
1 2 3
6
9 10
16 18
20 21
24 25 26
언더플로우!
재분배
15
3 8
19 22
1 2
5 6
9 10
16 18
20 21
24 25 26
9 삭제

(d)


15
3 8
19 22
언더플로우!
1 2
5 6
10
16 18
20 21
24 25 26
15
언더플로우!
3
병합!
19 22
1 2
5 6 8 10
16 18
20 21
24 25 26
병합!
3 15 19 22
1 2
5 6 8 10
16 18
20 21
24 25 26

학습내용

1. 레코드, 키의 정의 및 검색 트리
2. 이진 검색 트리
3. 레드 블랙 트리
4. B-트리
5. **다차원 검색 트리**

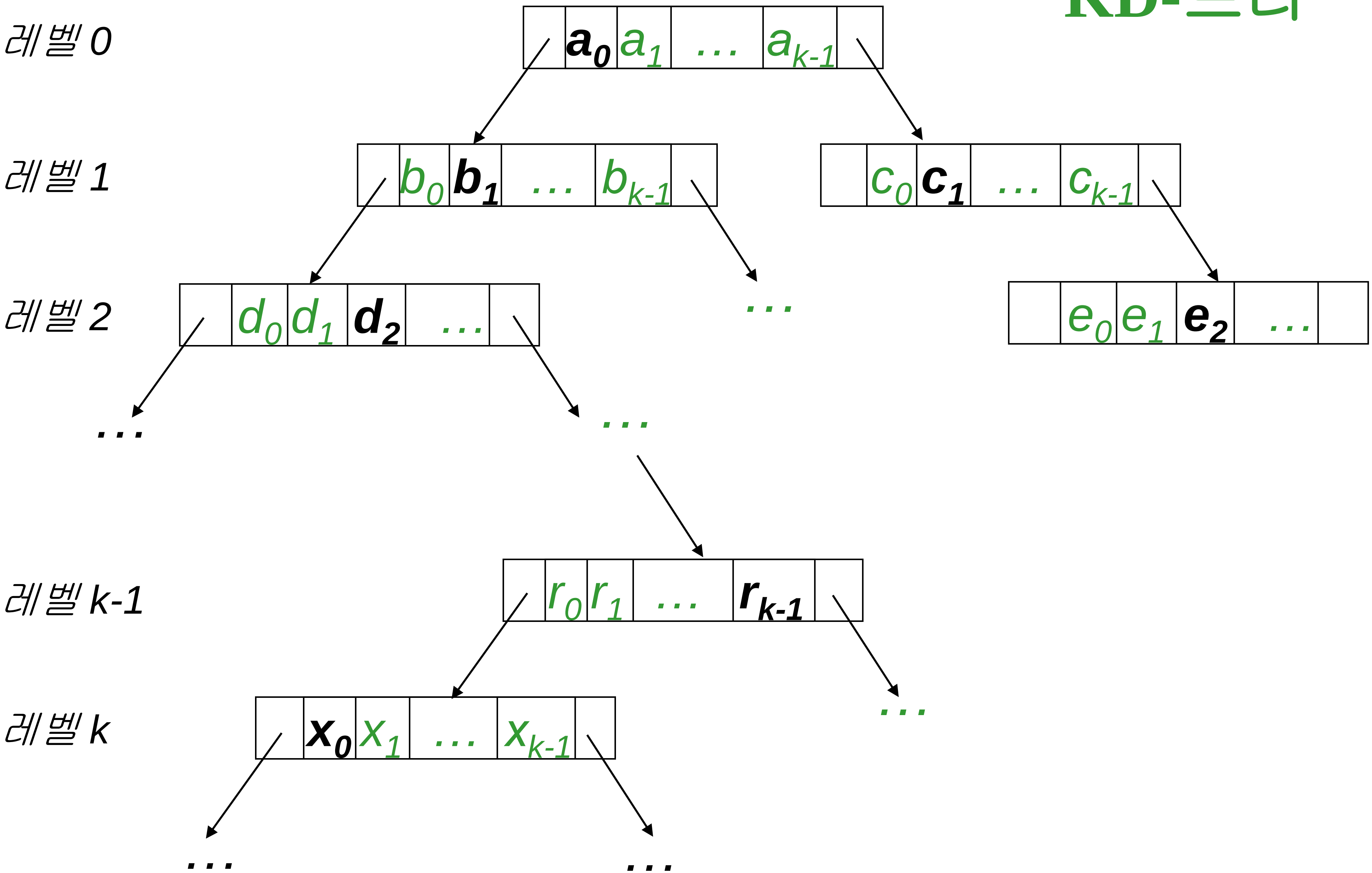
다차원 검색 트리

- 검색 키에 여러 필드를 사용하려면
  1. 여러 필드를 연결하여 하나의 필드처럼 다루는 방법
     - 각 필드가 갖는 의미를 활용할 수 없음
     - 예를 들어 키가 세개의 필드 (x, y, z) 이면 y의 값에 따라 작업하는 것이 불가능
  2. 다차원 검색 트리를 이용하는 방법
- 다차원 검색 트리
  - 검색키가 복수개의 필드로 이루어진 경우 복수개의 필드를 그대로 검색에 사용
  - KD-트리, KDB-트리, R-트리

KD-Tree

- 이진 검색 트리의 확장으로서, k개의 필드로 이루어진 검색키를 사용 ($k \geq 2$)
  - 검색 키 = (필드 0, 필드 1, … 필드 k−1)
  - 트리의 한 level에서는 하나의 필드만 사용
    - 레벨 i에서는 필드 i mod k 를 사용
    - 따라서 어떤 필드를 검색에 사용했으면, k개의 level을 내려간 다음 다시 그 필드를 검색에 사용

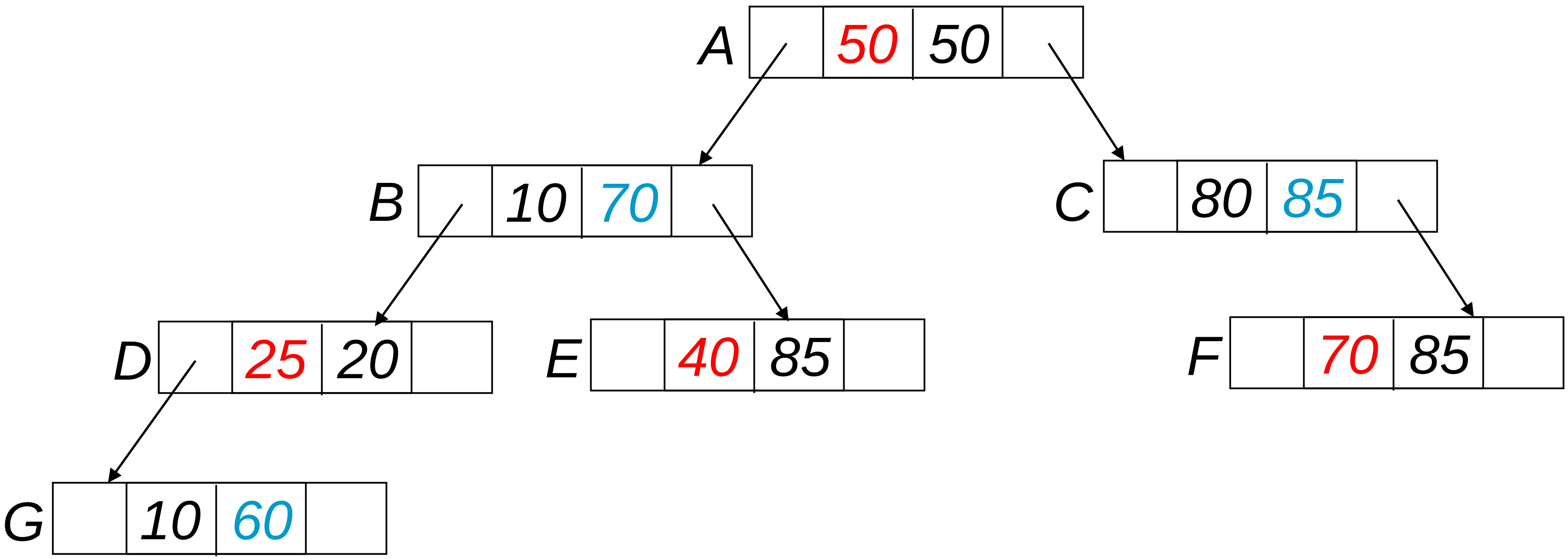
KD-트리

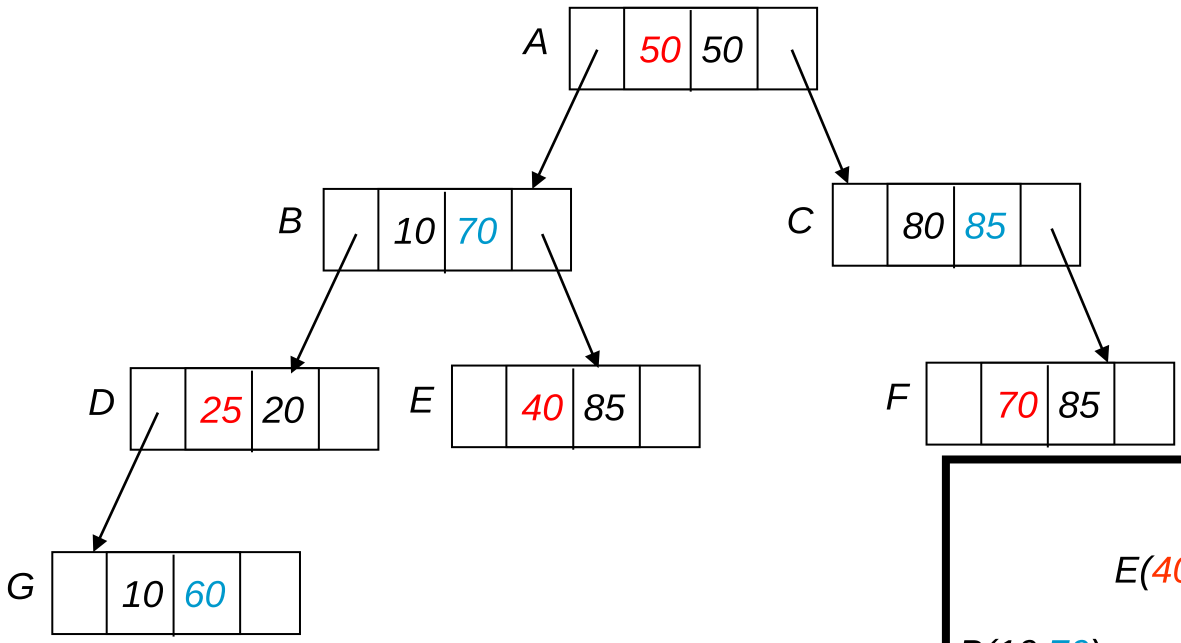


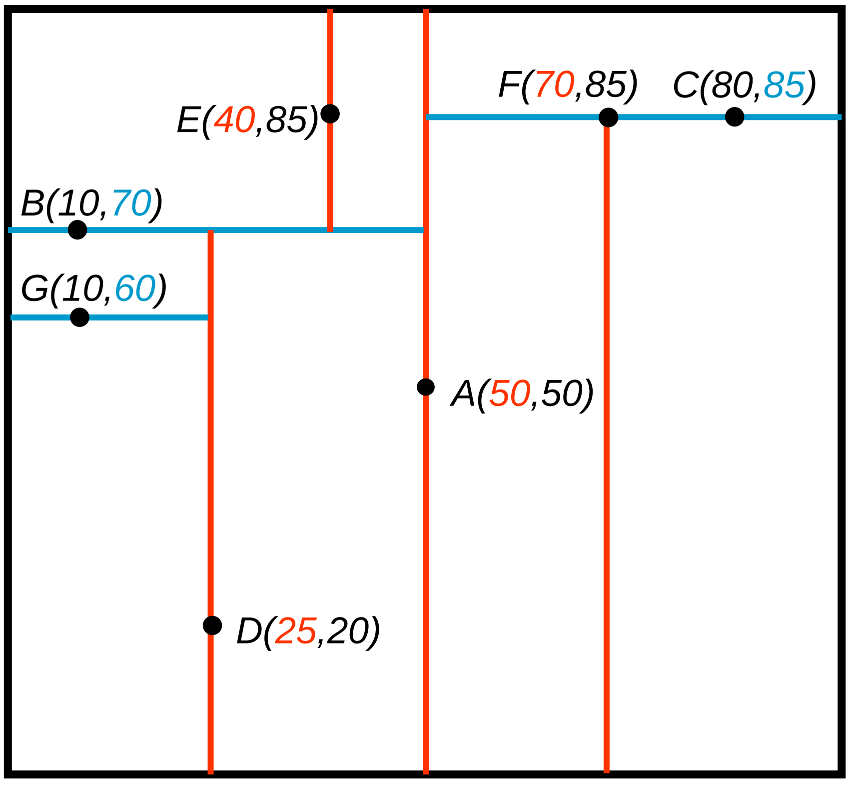
레벨 0
레벨 1
레벨 2
레벨 k-1
레벨 k
a_0 a_1 … a_{k-1}
b_0 b_1 … b_{k-1}
c_0 c_1 … c_{k-1}
d_0 d_1 d_2 …
e_0 e_1 e_2 …
r_0 r_1 … r_{k-1}
x_0 x_1 … x_{k-1}

이차원 KD-트리의 예

- ✓k = 2인 예
- ✓이 예에서는 필드값이 같으면 오른쪽으로 분기







✓ KD-트리의 각 노드는 다차원 공간의 한 점에 해당

✓ KD-트리에서 검색을 이용해 내려간다는 것은 다차원 공간에서 나누어진 결과에 따라 공간을 좁혀 나가는 것

요약

- B-트리는 검색 트리가 디스크에 저장되어 있는 경우에 유용한 검색 트리이다.
  - 블록 크기와 트리 노드 크기를 일치시킴으로써 가능한 최대의 분기를 추구한다.
  - 디스크 접근 횟수를 현저히 떨어뜨려 검색 트리의 저장/검색/삭제 효율을 높인다.
- 다차원 검색 트리는 키가 여러 개의 필드로 구성된 검색 트리이다.
  - 예) KD-트리