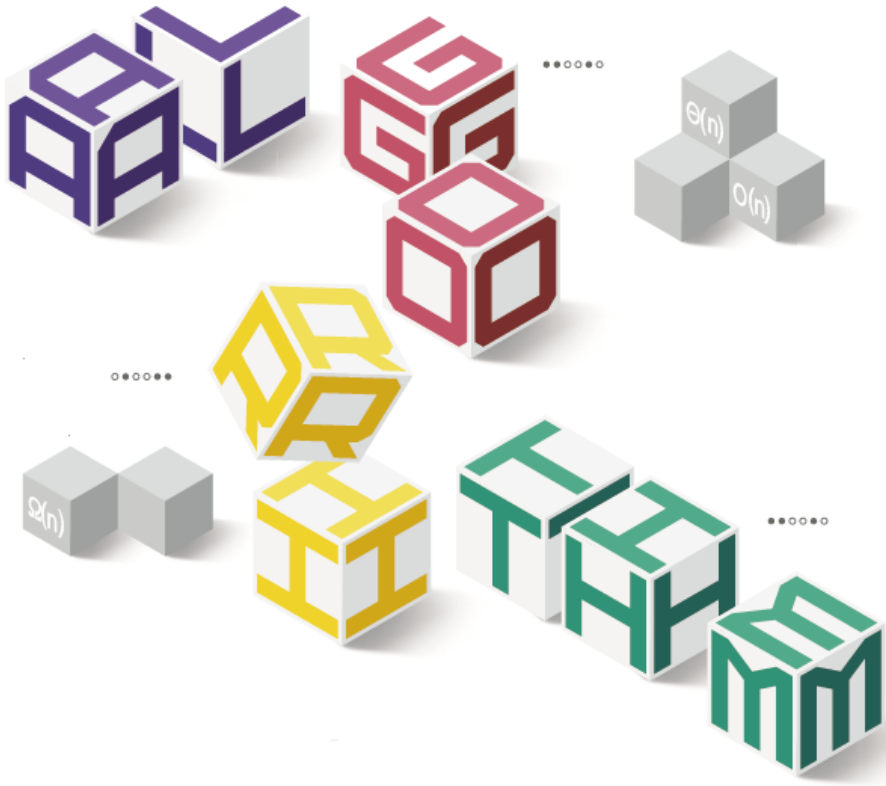


관계 중심의 사고법

쉽게 배우는 알고리즘

6장. 검색 트리

Search Trees

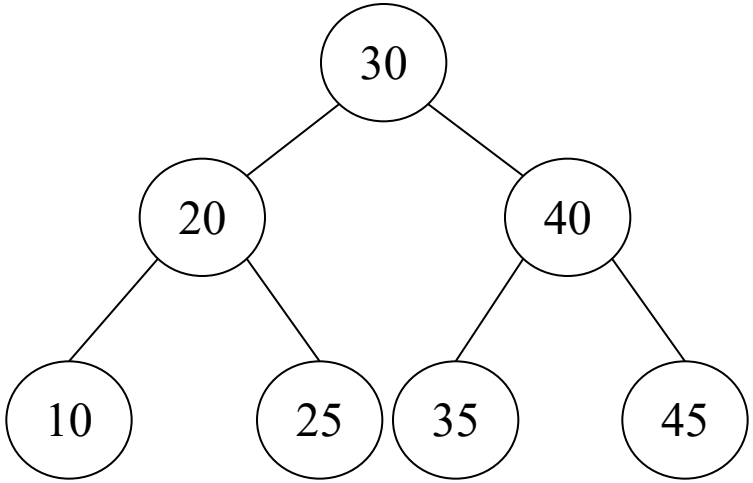
Record, Key, Search Tree

- Record
  - 개체에 대해 수집된 모든 정보를 포함하고 있는 저장 단위
  - e.g., 사람의 record
    - <주민번호, 이름, 집주소, 집 전화번호, 직장 전화번호, 휴대폰 번호, 최종 학력, 연소득, 가족 상황, ... > ← 이런 field들의 모음
- Field
  - record에서 각각의 정보를 나타내는 부분
  - e.g., 위 사람의 record에서 각각의 정보 주민번호, 이름, 집주소, …
- Search key or Key
  - 다른 record와 구별할 수 있도록 각 record를 대표할 수 있는 필드
  - Key는 하나의 field로 이루어질 수도 있고, 두 개 이상의 field로 이루어질 수도 있다
- Search Tree
  - 각 노드가 규칙에 맞도록 하나씩의 key를 갖고 있다
  - 이를 통해 해당 record가 저장된 위치를 알 수 있다

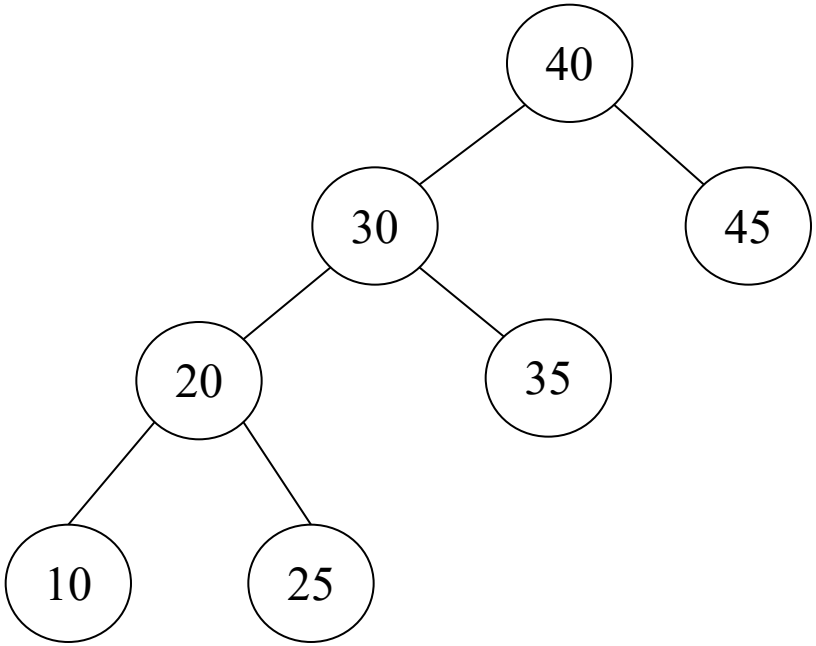
Binary Search Trees

- 각 노드는 하나씩의 key 값을 갖는다. 각 노드의 key 값은 다르다.
- 최상위 레벨에 루트 노드가 있고, 각 노드는 최대 두 개의 자식을 갖는다.
- 임의의 노드의 key값은 자신의 left subtree의 모든 노드 key 값보다 크고, right subtree의 모든 노드 key 값보다 작다.

Examples of Binary Search Trees

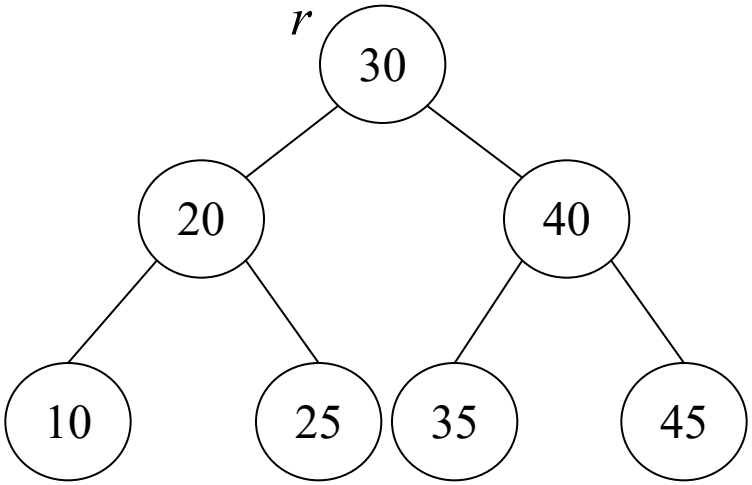


(a)

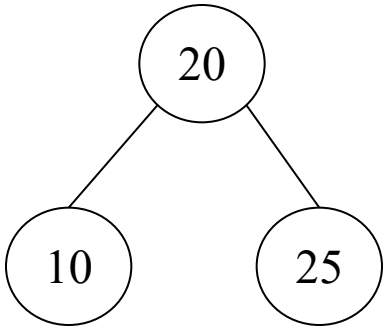


(b)

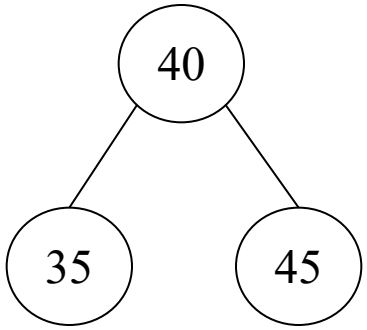
Examples of Subtrees



(a)



(b) 노드 r의 왼쪽 서브트리



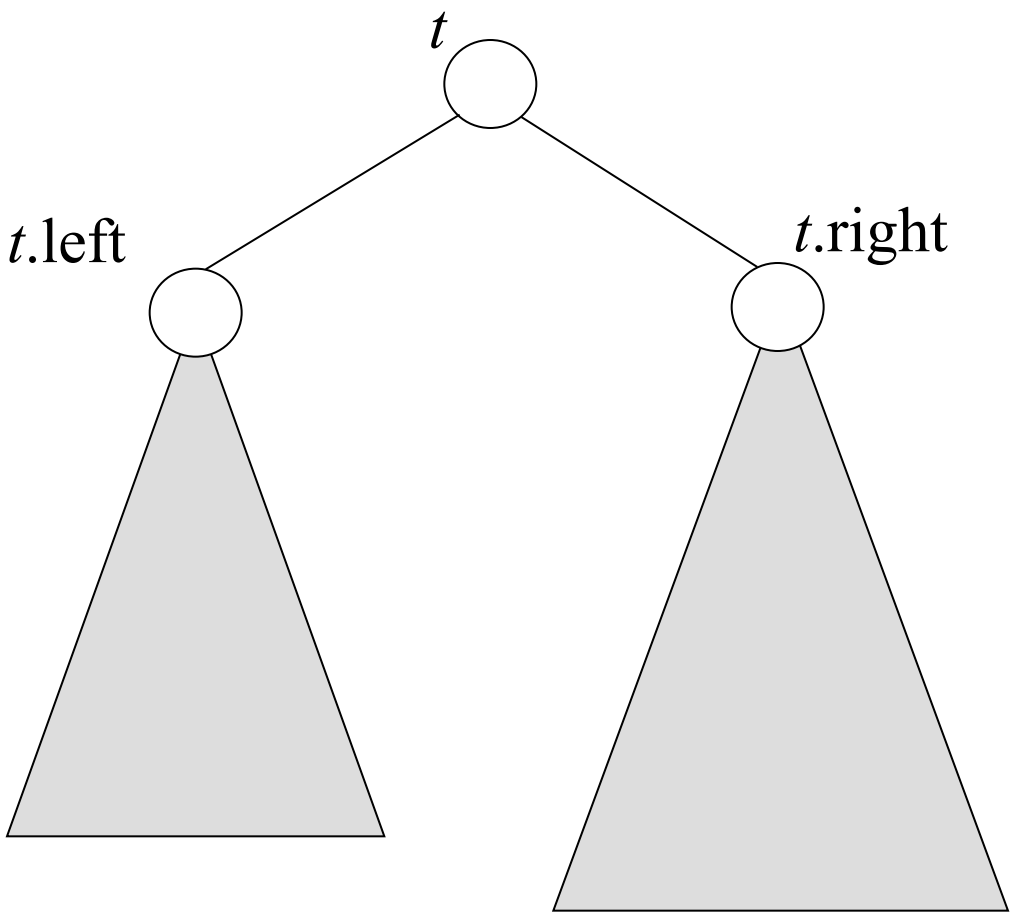
(c) 노드 r의 오른쪽 서브트리

Reminder: Search in Binary Search Trees

t: 트리의 루트 노드
x: 검색하고자 하는 키

```
TreeNode treeSearch(t, x)
▷ t : root node, x : key
{
        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);
}
```

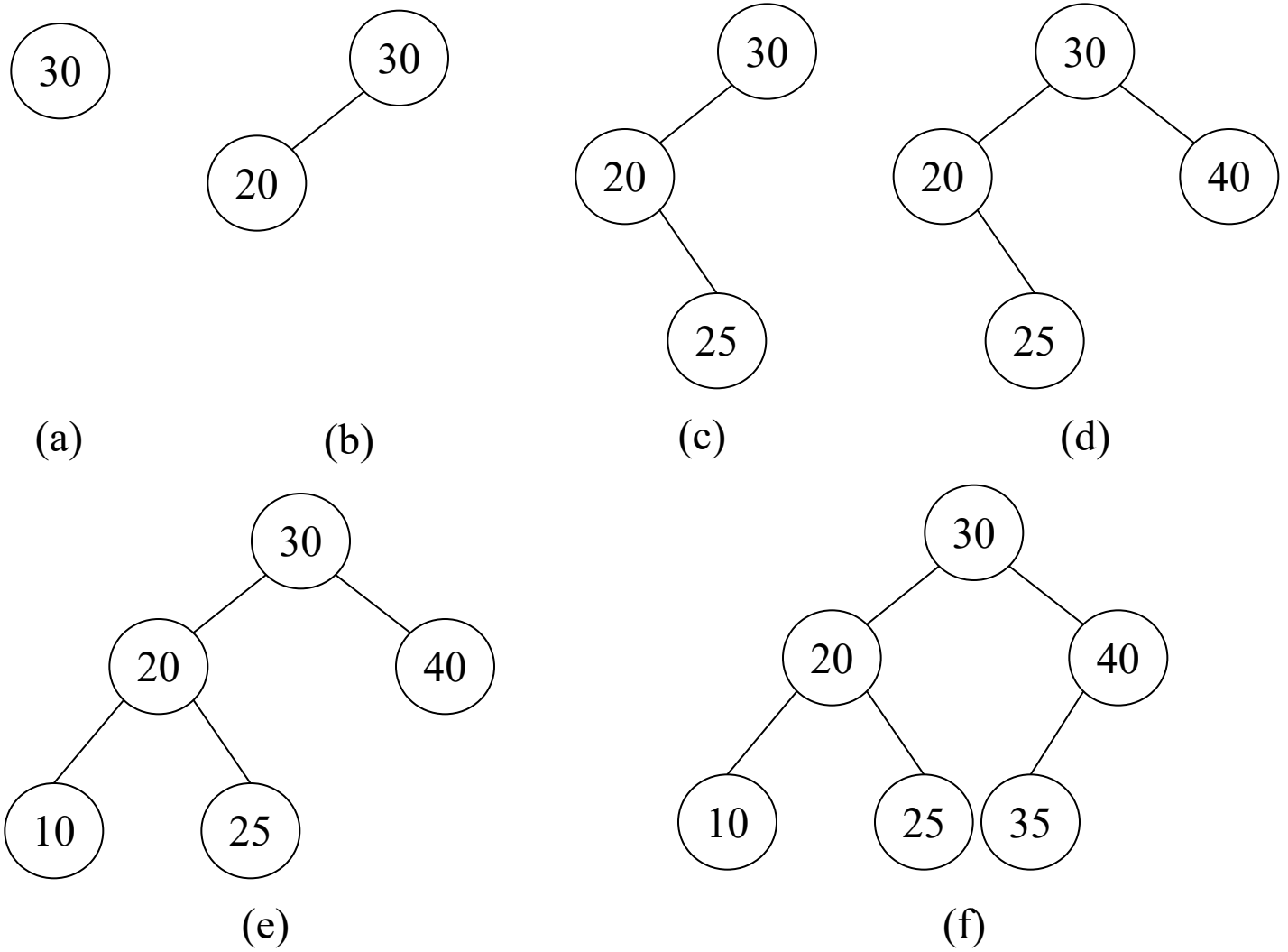
Recursive View in Search



Reminder: Insertion in Binary Search Trees

```
TreeNode treeInsert(t, x)
▷ t : root node, x : key to insert
{
    if (t=NIL) then {
        r.key ← x;                          ▷ r : 새 노드
        return r;
    }
    if (x < t.key)
        then {t.left ← treeInsert(t.left, x); return t;}
        else {t.right ← treeInsert(t.right, x); return t;}
}
```

Examples of Insertion



Reminder: Deletion in Binary Search Trees

t: 트리의 루트 노드
r: 삭제하고자 하는 노드

- 3가지 경우에 따라 다르게 처리한다
  - Case 1 : r이 lead node인 경우
  - Case 2 : r의 child가 1개인 경우
  - Case 3 : r의 child가 2개인 경우

Reminder: Deletion in Binary Search Trees

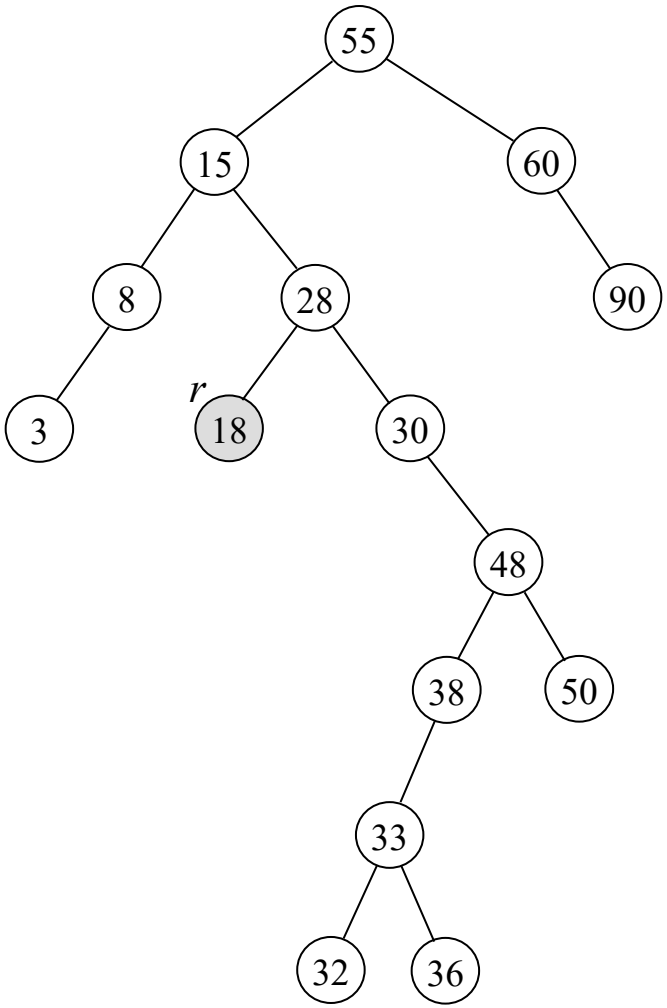
```
Sketch-TreeDelete(t, r)
▷ t : root node, r : node to delete
{
    if (r이 leaf node) then                                  ▷ Case 1
        그냥 r을 버린다;
    else if (r의 child가 하나만 있음) then                    ▷ Case 2
        r의 parent가 r의 (유일한) child를 직접 가리키도록 한다;
    else                                                     ▷ Case 3
        r의 right subtree의 minimum node s를 삭제하고,
        s의 내용을 r 자리로 복사한다;
}
```

Deletion in Binary Search Trees

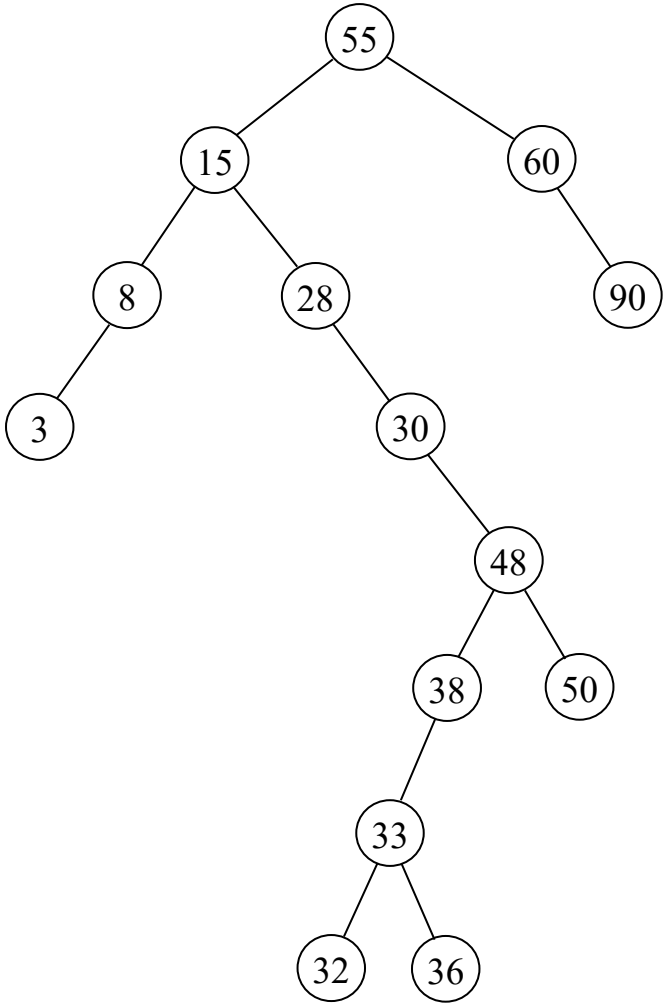
t: 트리의 루트 노드
r: 삭제하고자 하는 노드

```
treeDelete(t, r)
{
    if (r = t) then root ← deleteNode(t);          ▷ r이 루트 노드인 경우
    else if (r = r.parent.left)                    ▷ r이 루트가 아닌 경우
        then r.parent.left ← deleteNode(r);        ▷ r이 p의 왼쪽 자식
        else r.parent.right ← deleteNode(r);       ▷ r이 p의 오른쪽 자식
}
deleteNode(r)
{
    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)
                {parent ← s; s ← s.left;}
        r's data ← s's data;
        if (s = r.right) then r.right ← s.right;
                else parent.left ← s.right;
        return r;
    }
}
```

Example: Case 1

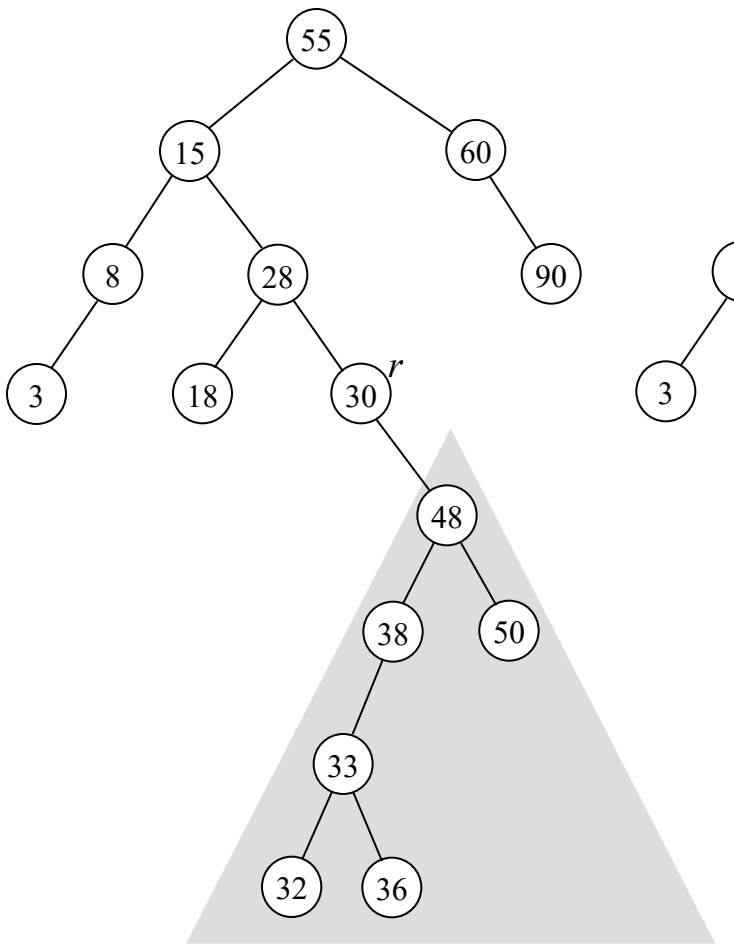


(a) r의 자식이 없음

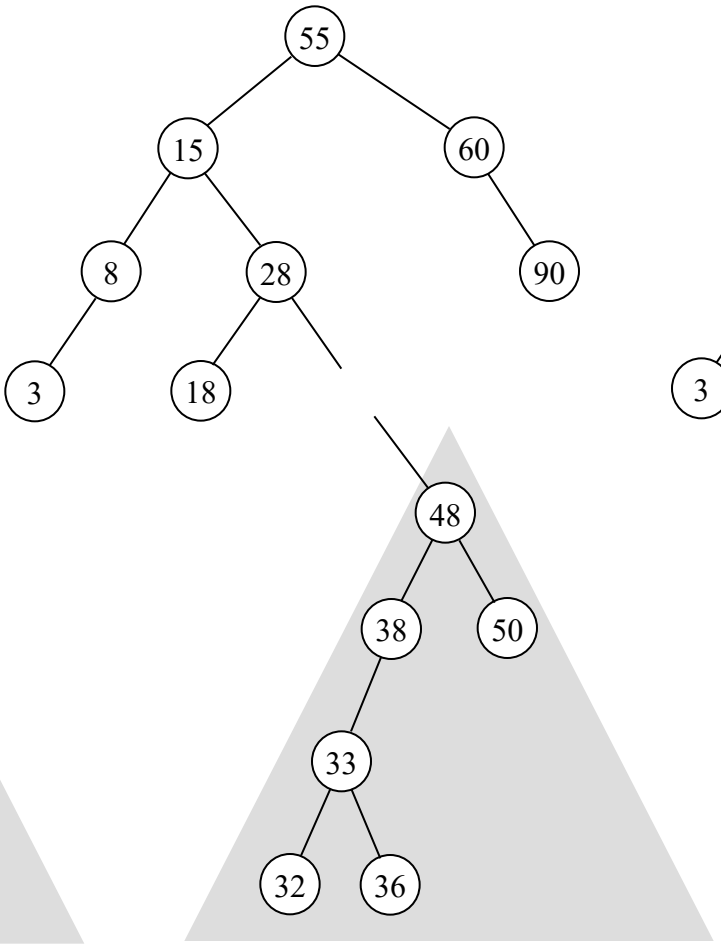


(b) 단순히 r을 제거한다

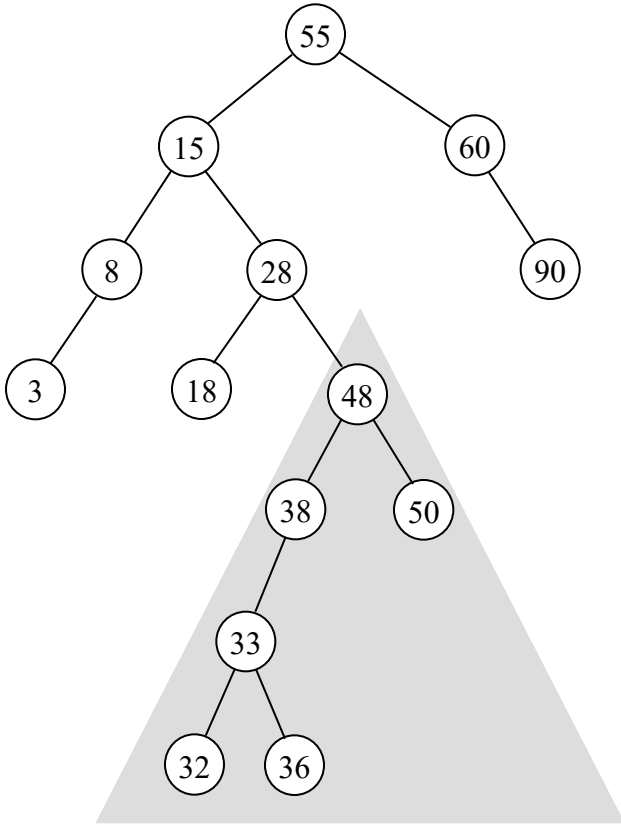
Example: Case 2



(a) r의 자식이 하나뿐임

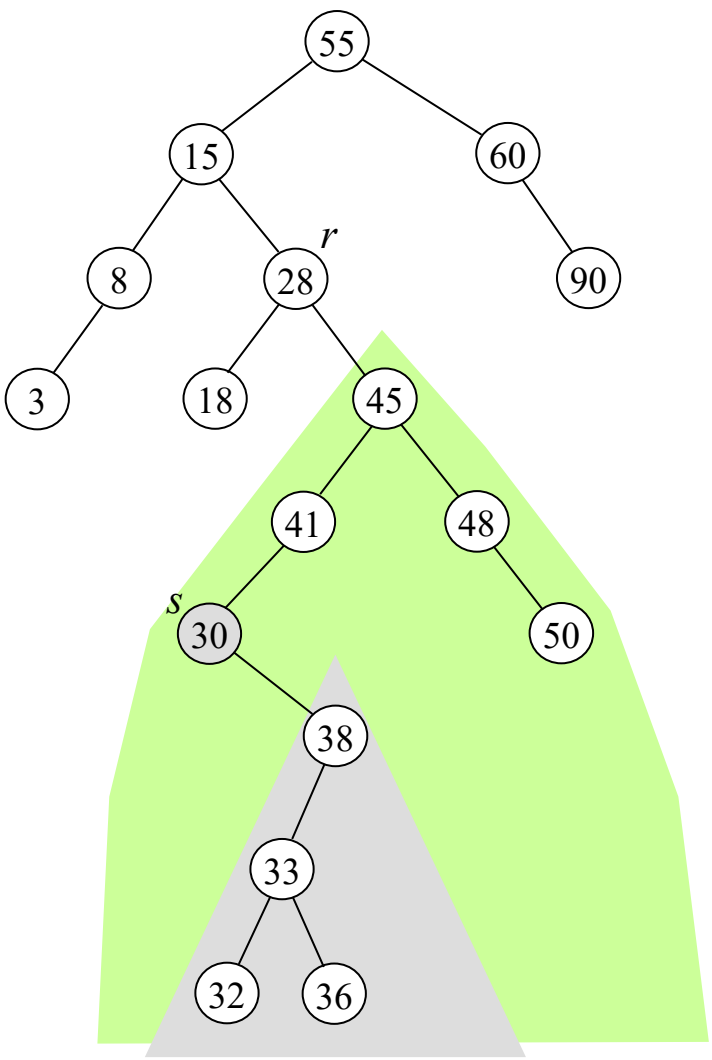


(b) r을 제거

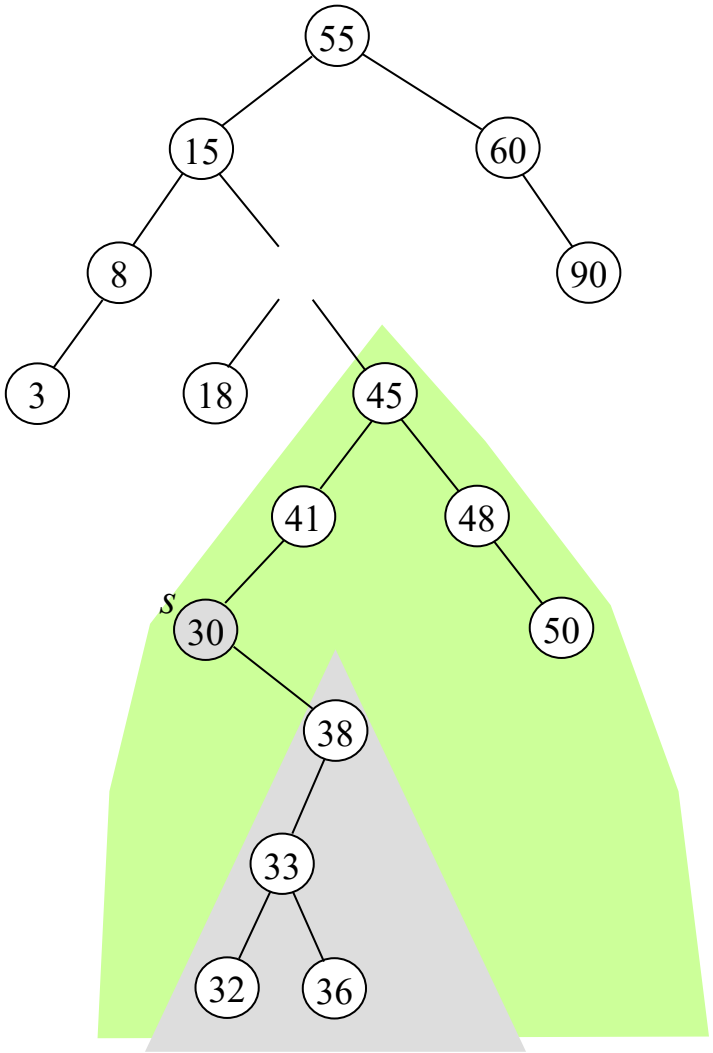


(c) r 자리에 r의 자식을 놓는다

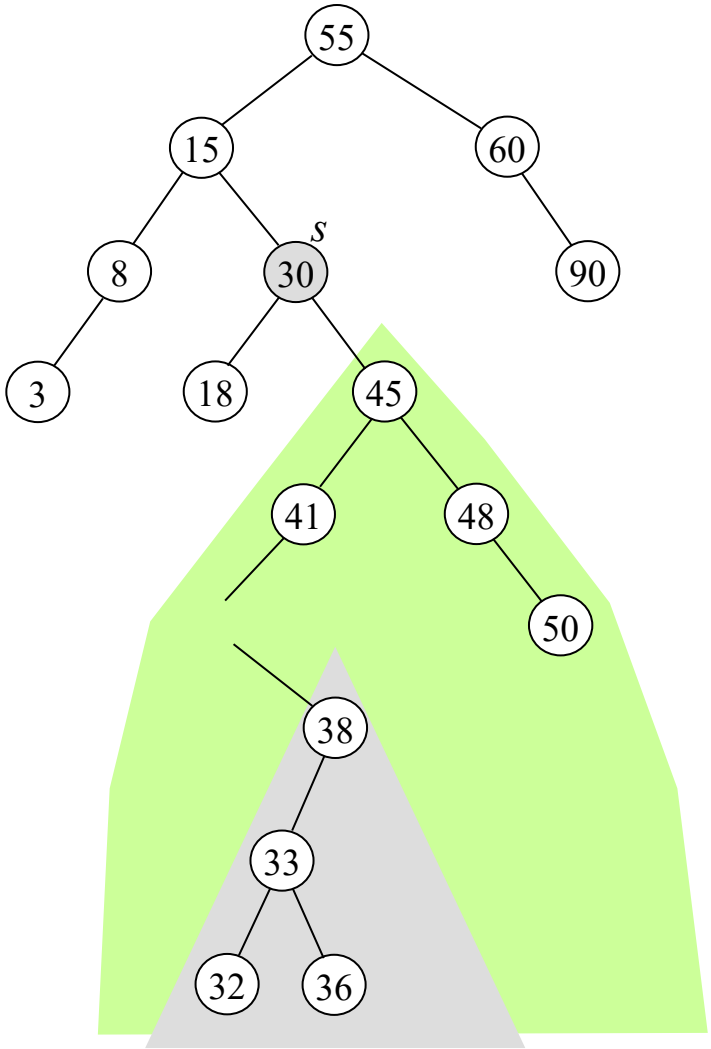
Example: Case 3



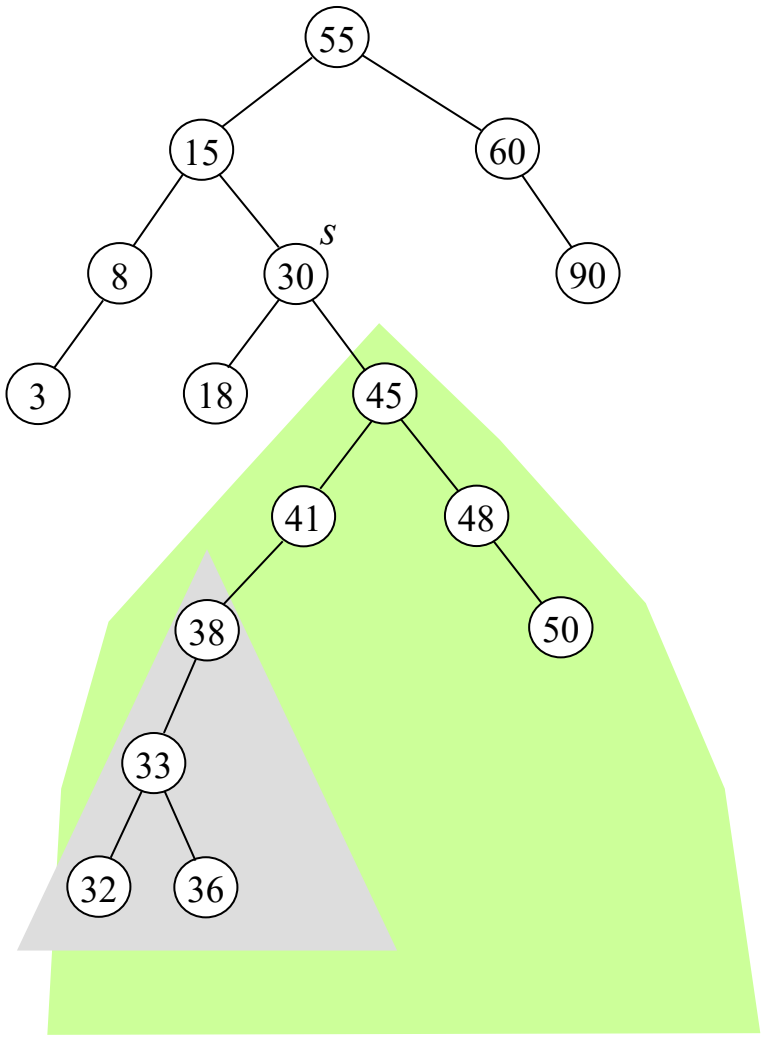
(a) r의 직후원소 s를 찾는다



(b) r을 없앤다



(c) s를 r자리로 옮긴다



(d) s가 있던 자리에 s의 자식을 놓는다

Theorem

- A sequence of n inserts into an empty binary search tree takes $O(n \log n)$ on average

  (Assume that every permutation of the input sequence is equally likely.)

<Proof>

뒷 페이지.

<Proof>

$D(n)$: the average IPL(Internal Path Length) of a binary tree with n nodes.

Clearly $D(0) = 0$, $D(1)=1$.

$$D(n) = \frac{1}{n}\sum_{k=1}^{n}[D(k-1) + (k-1) + D(n-k) + (n-k)] + 1$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} D(k) + n$$

Assume that $\exists c > 0$ s.t. $D(k) \leq ck\log k \; \forall k < n$.

Then, we verify that $D(n) \leq cn\log n$ (i.e., $D(n) = O(n\log n)$)

$$D(n) = \frac{2}{n}\sum_{k=0}^{n-1} D(k) + n$$

$$= \frac{2}{n}\sum_{k=2}^{n-1} D(k) + \Theta(n) \quad \leftarrow \text{D(0), D(1) absorbed}$$

$$\leq \frac{2}{n}\sum_{k=2}^{n-1} ck\log k + \Theta(n)$$

$$\leq \frac{2}{n}\int_{1}^{n} cx\log x\, dx + \Theta(n$$

$$= \frac{2c}{n}\left(\left[\frac{1}{2}x^2\log x\right]_1^n - \left[\frac{1}{4}x^2\right]_1^n\right) + \Theta(n)$$

$$= \frac{2c}{n}\left(\frac{1}{2}n^2 \log n - \frac{1}{4}n^2 + \frac{1}{4}\right) + \Theta(n)$$

$$= cn \log n - \frac{cn}{2} + \frac{c}{2n} + \Theta(n)$$

$$= cn \log n - \frac{cn}{2} + \Theta(n) \quad \leftarrow -\frac{cn}{2} \text{ absorbed}$$

$$\leq cn \log n$$

We can choose $c > 0$ s.t. $\frac{cn}{2}$ dominates $\Theta(n)$

$$\therefore D(n) = O(n \log n)$$

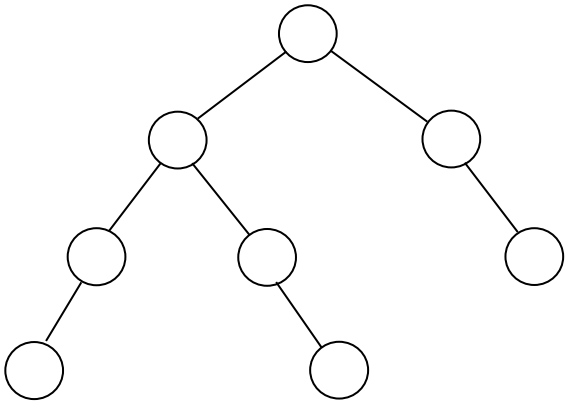
Red-Black Trees

- Every node in the search tree has a color: red or black.
- It has to satisfy the following properties (red-black properties):

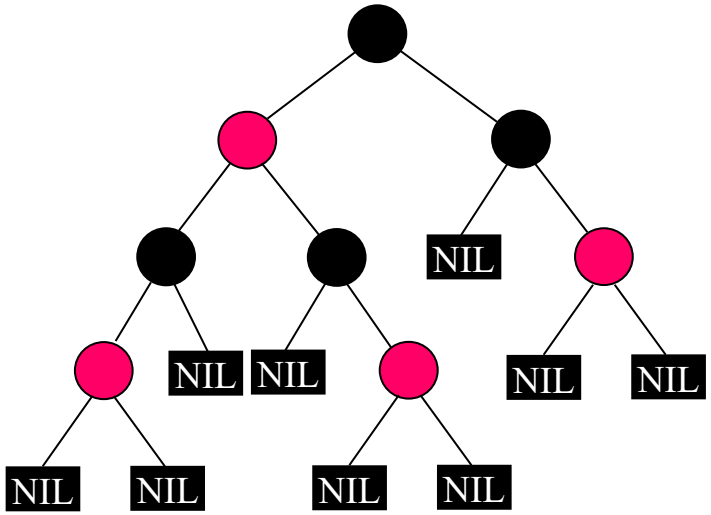
① Root is black
② Every leaf is black
③ If a node is red, its children should be black (no two consecutive reds)
④ In any path from the root to a leaf, the # of black nodes on the path is the same (이를 black height라 한다)

✓ 여기서 leaf 는 일반적인 의미의 leaf node와 다르다.
모든 NIL 포인터가 NIL이라는 leaf node를 가리킨다고 가정한다.

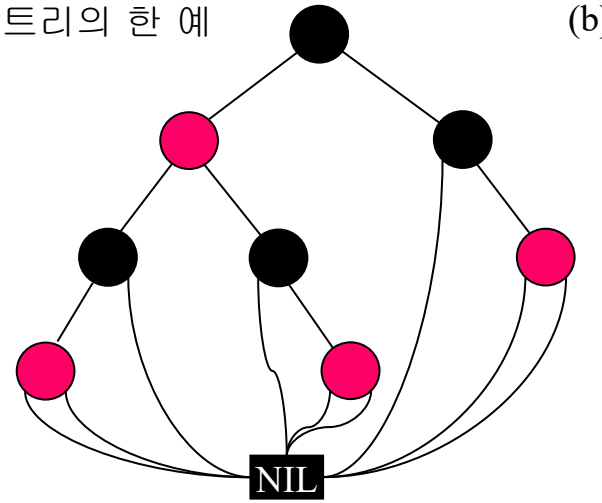
BST를 RB Tree로 만든 예



(a) 이진검색트리의 한 예



(b) (a)를 레드블랙트리로 만든 예



(c) 실제 구현시의 NIL 노드 처리 방법

Theorem

- In a red-black tree T of n nodes, the worst-case depth is $O(\log n)$

<Proof>

- For any node v in a red-black tree,
  the subtree rooted at v contains at least $2^{\mathrm{bh}(v)} - 1$ internal nodes. ---- ①
- By property 3, $h \leq 2\,\mathrm{bh}(T)$ // h : the height of the tree

$$\frac{h}{2} \leq \mathrm{bh}(T) \quad \text{---------- ②}$$

- From ① and ②, $n \leq 2^{\mathrm{bh}(T)} - 1$

$$\leq 2^{\frac{h}{2}} - 1$$

$$\Rightarrow\ h \leq 2\log(n+1)$$

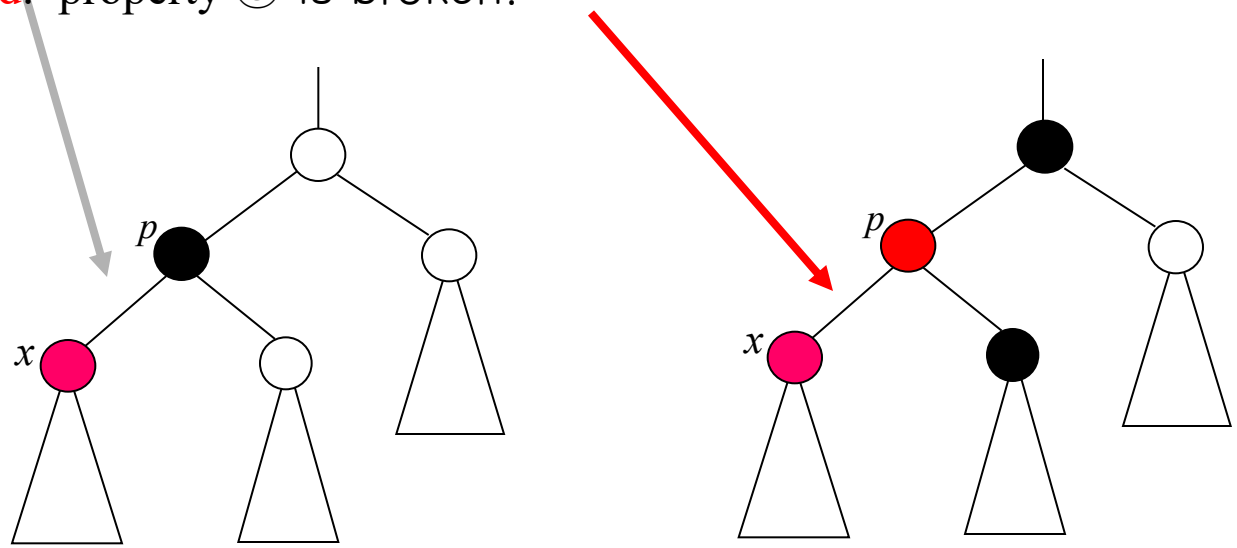
Optimal Binary Search Tree (Static)

Dynamic Programming에서 취급

Insertion in Red-Black Trees

항상 실패하는 검색 후에 매달린다.

- 이진검색트리에서의 삽입과 같다. 다만 삽입 후 삽입된 노드를 레드로 칠한다. (이 노드를 x라 하자)
- If x's parent p is
  - black: no problem!
  - red: property ③ is broken!

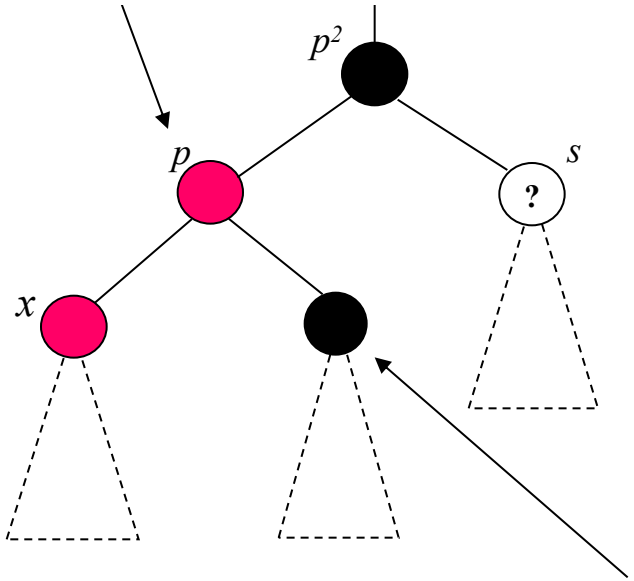


✓ 그러므로 p가 red인 경우만 고려하면 된다

Insertion in RB Trees

given condition: *p* is red

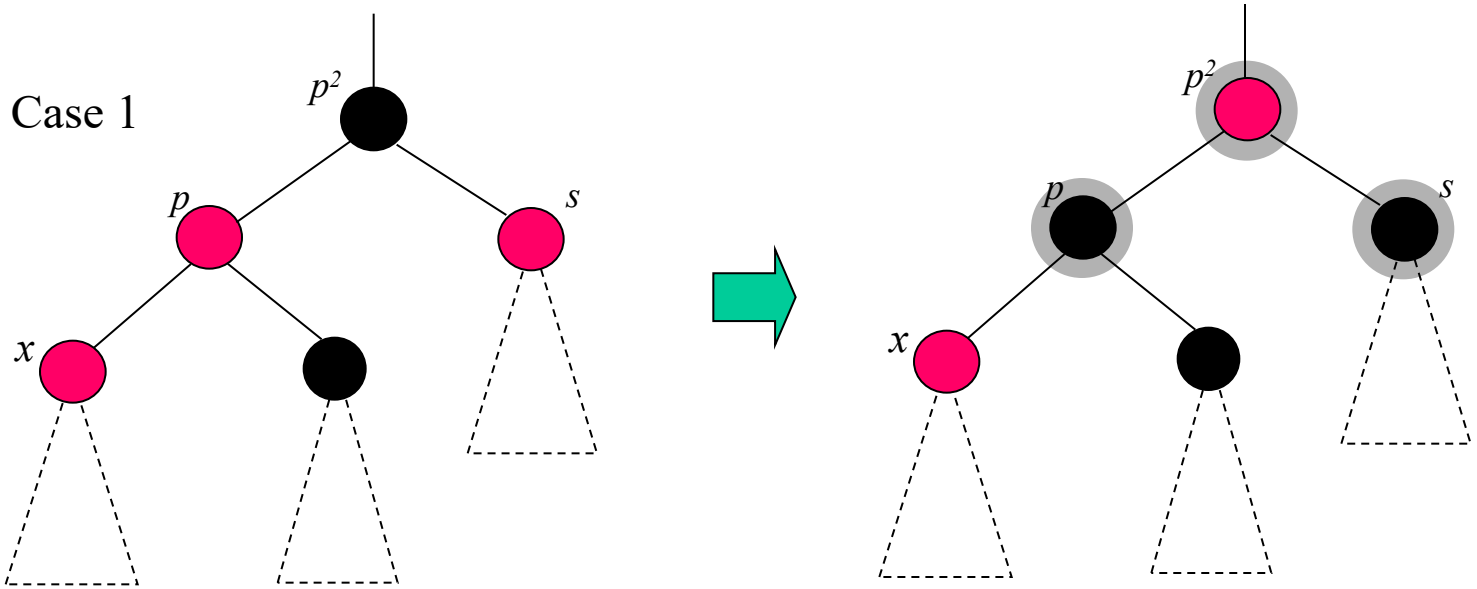
- *x*'s sibling(if any) and p^2 must be black by property ③
- Two cases by *s*'s color
  - Case 1: *s* is red
  - Case 2: *s* is black



질문: 삽입 직후에 이런 경우가 있을 수 있는가?

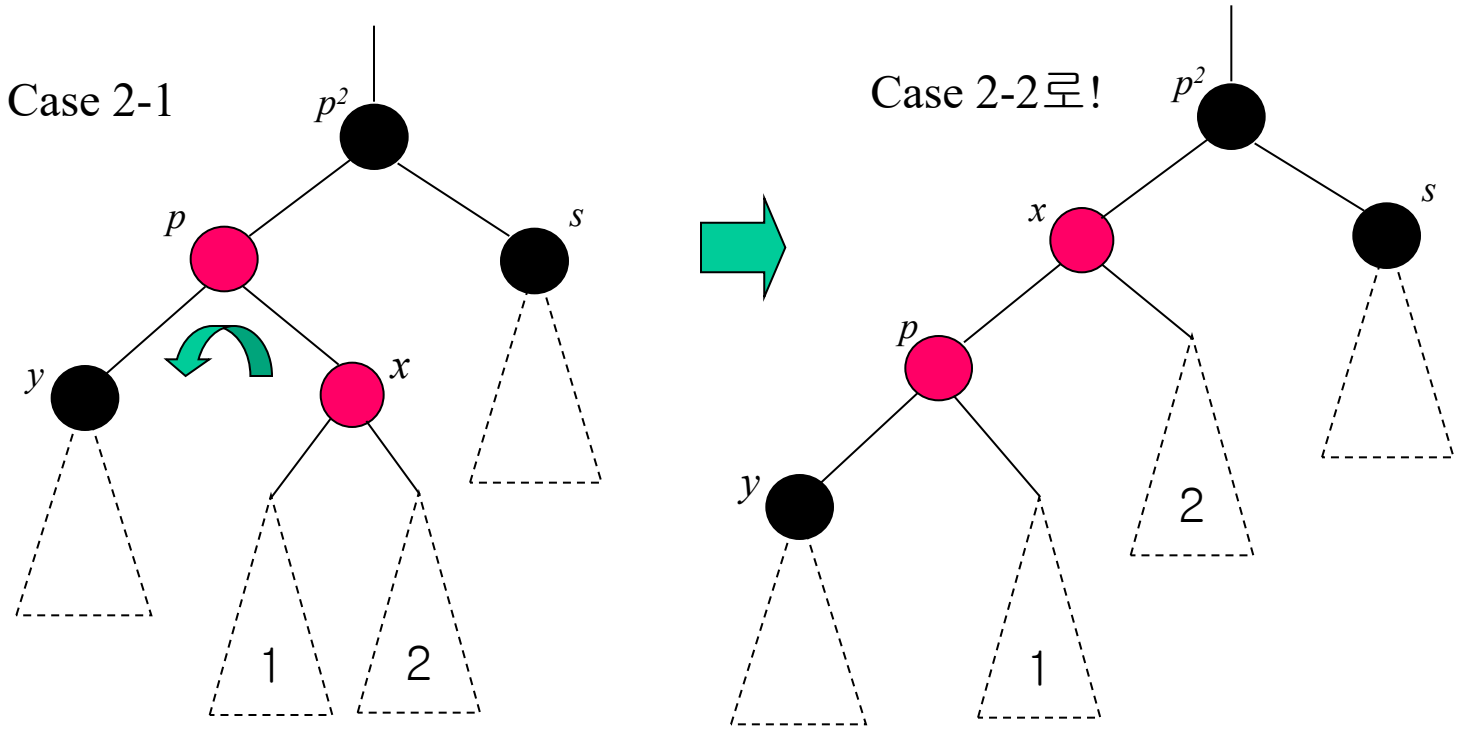
Case 1: s is red

: 색상이 바뀐 노드



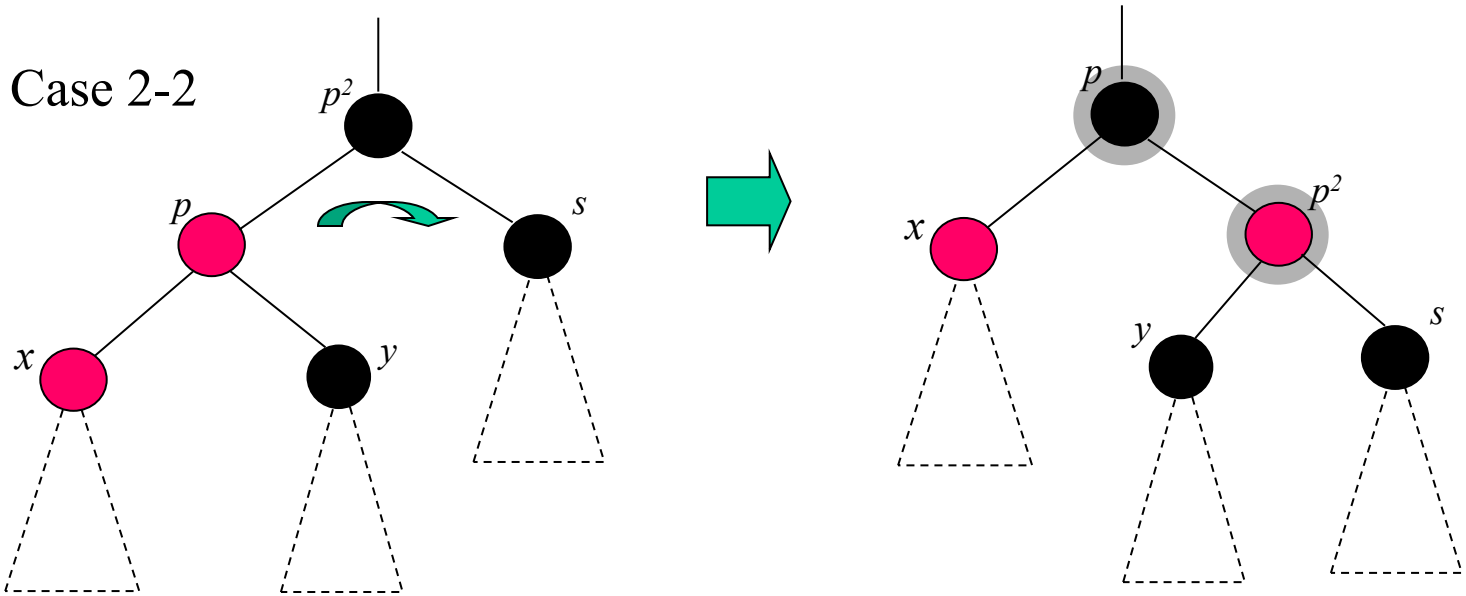
- ✓ p^3가 black이면 끝
- ✓ p^3가 red이면 p^2에서 방금과 같은 문제 발생: Recursive problem!

Case 2-1: *s* is black and *x* is *p*'s right child



Case 2-2: s is black and x is p's left child

: 색상이 바뀐 노드



✓ 삽입 완료!

수행 시간

1차 삽입: $\Theta(\log n)$

수선: $O(\log n)$

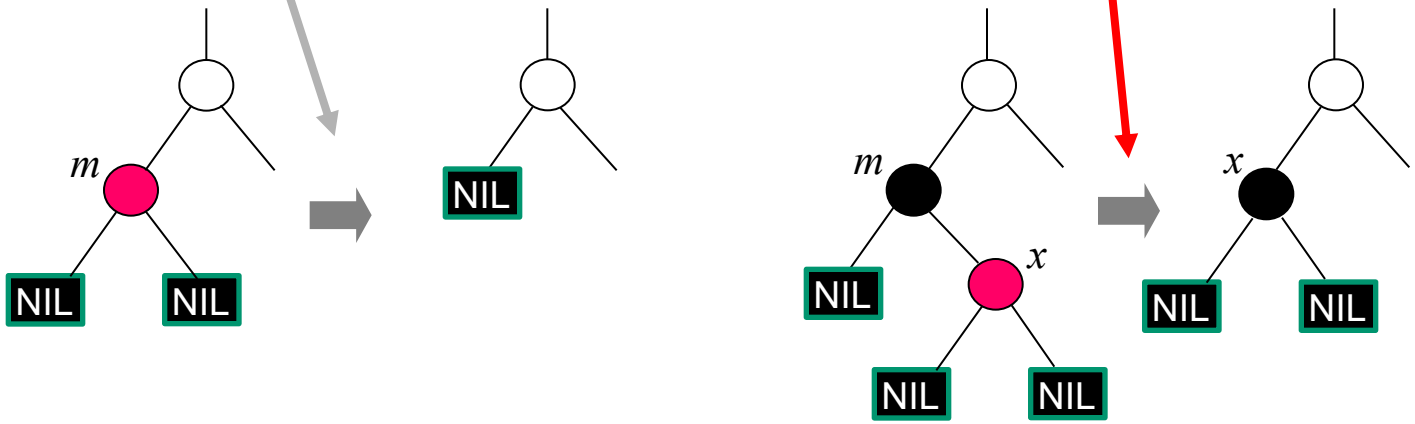
$\Theta(\log n)$ in total

Deletion in Red-Black Trees

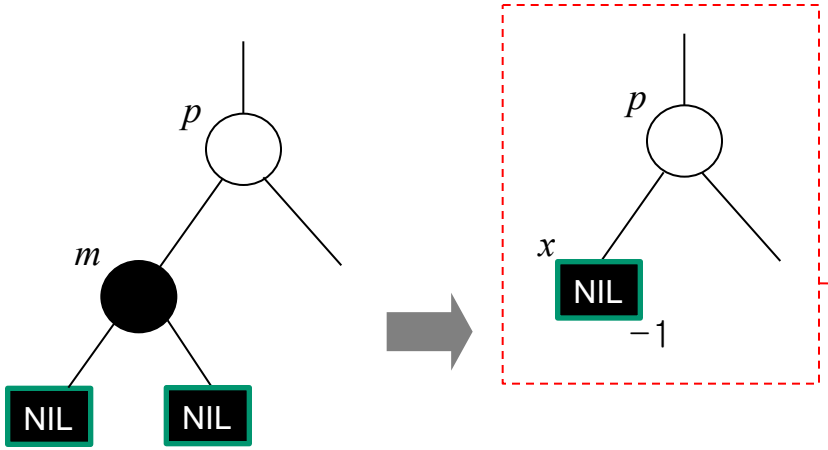
- We can restrict to the cases that the deleted node has

  no child or only one child

  - 이유: "쉽게 배우는 알고리즘" p.174의 첫 문단 참조
  - *m*: node to be deleted
- If *m* is red: no problem!
- Even when *m* is black, no problem if the (only) child is red!



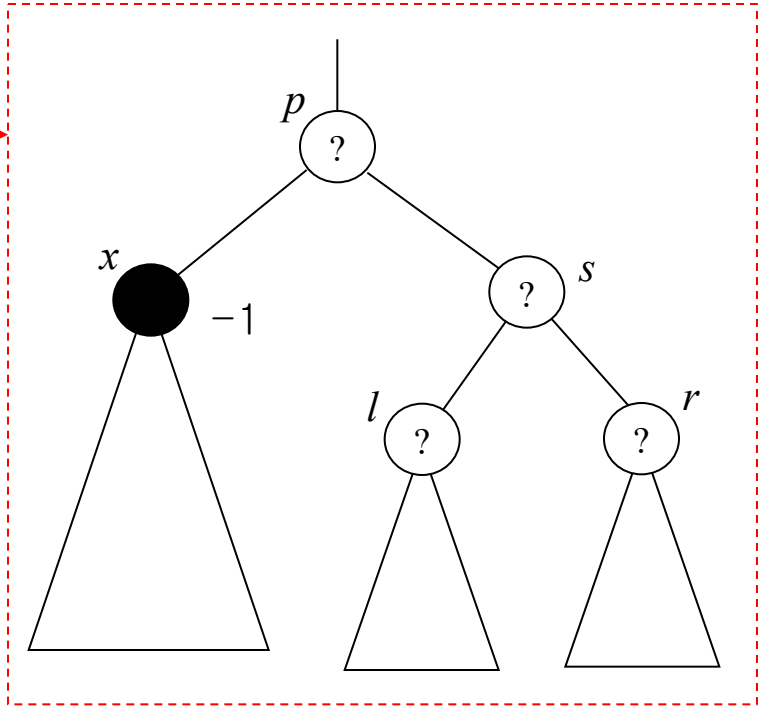
문제가 되는 케이스



m 삭제 후 문제 발생
(RB property ④ 위반)

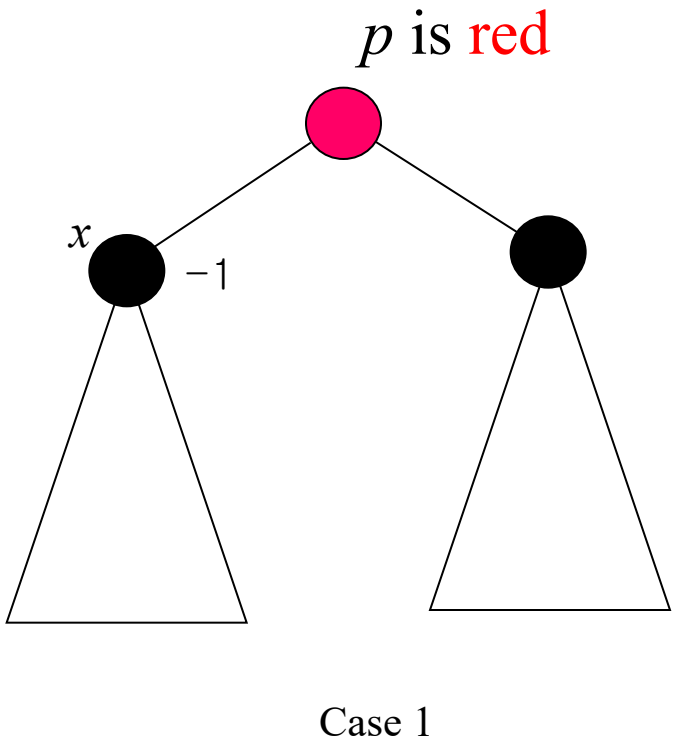
✓ *x* 옆의 −1은 root에서 *x* 를 통해 leaf에 이르는 경로에서 black node의 수가 하나 모자람을 의미한다.

✓ −1이 재귀적으로 올라가는 상황까지 반영한 그림. 왼쪽 그림은 이 그림의 한 예가 된다.

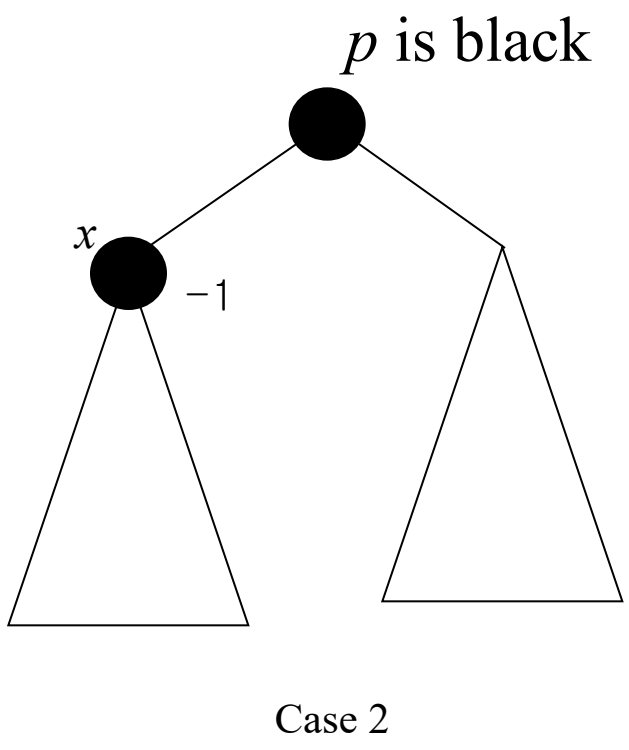


*x*의 주변 상황에 따라 처리 방법이 달라진다

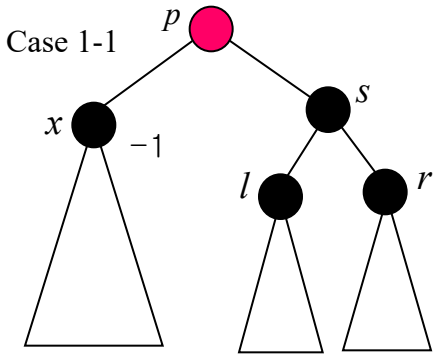
경우의 수 나누기

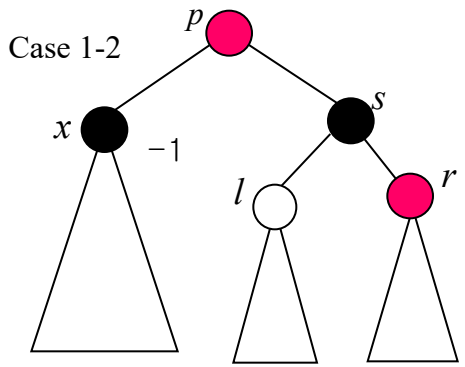


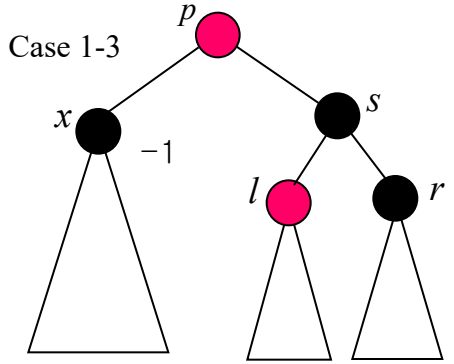
Case 1



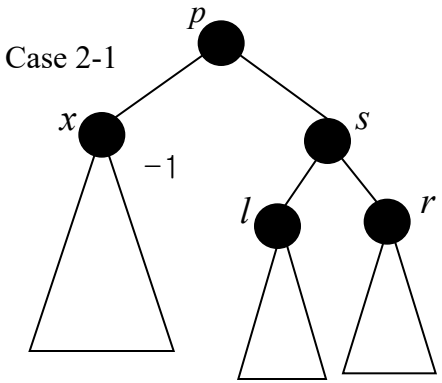
Case 2

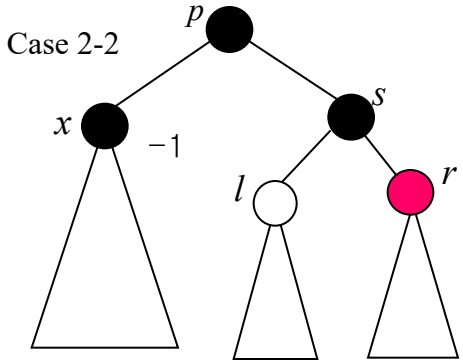


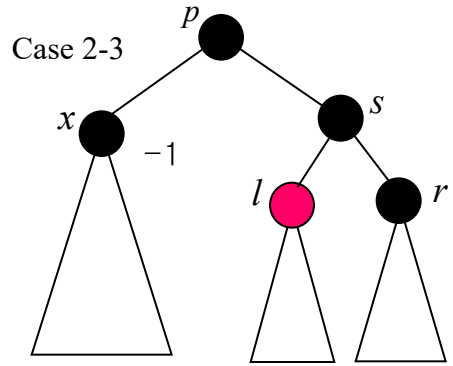


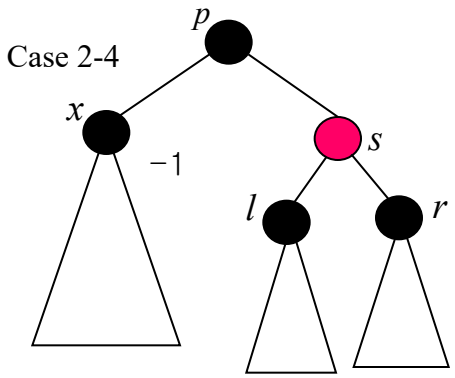


흰 색 -> don't care (둘다 해당)



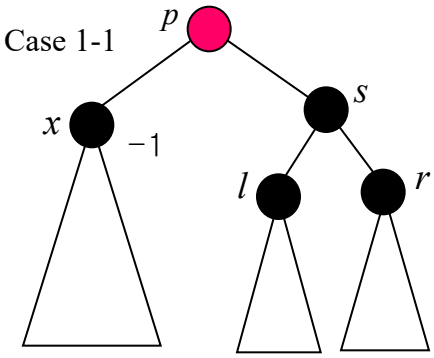


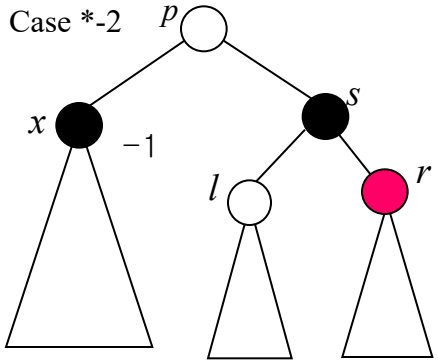


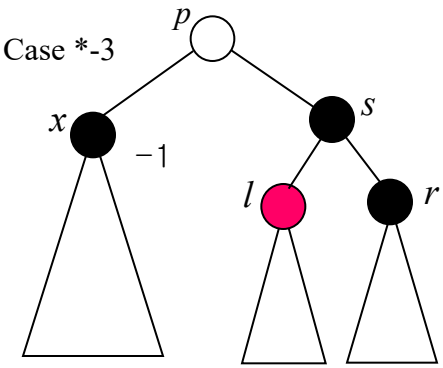


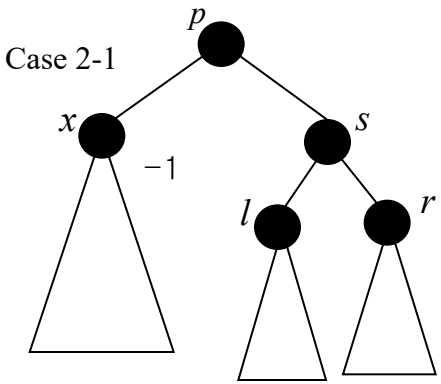
처리 방법이 동일하다

처리 방법이 동일하다

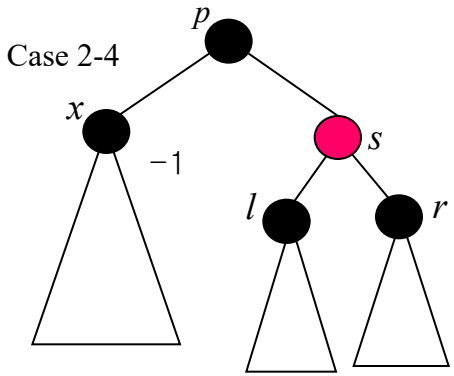




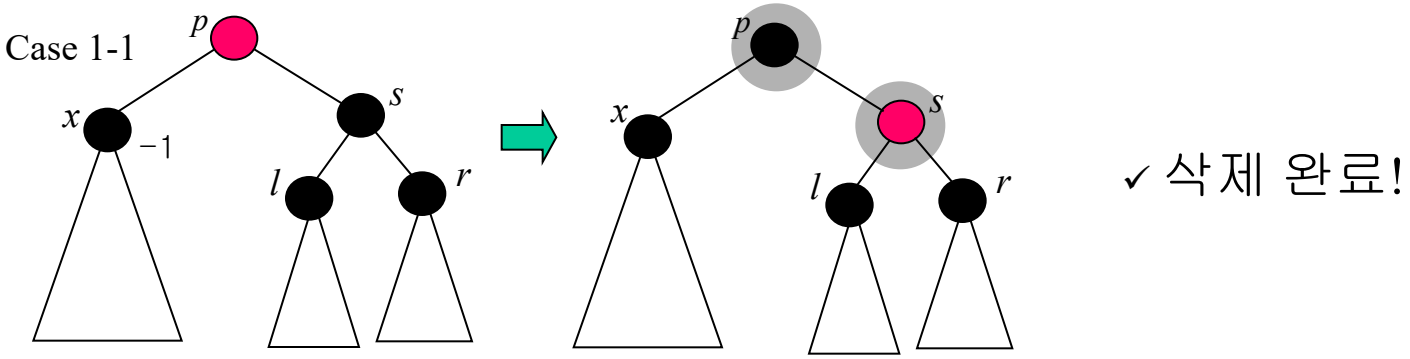




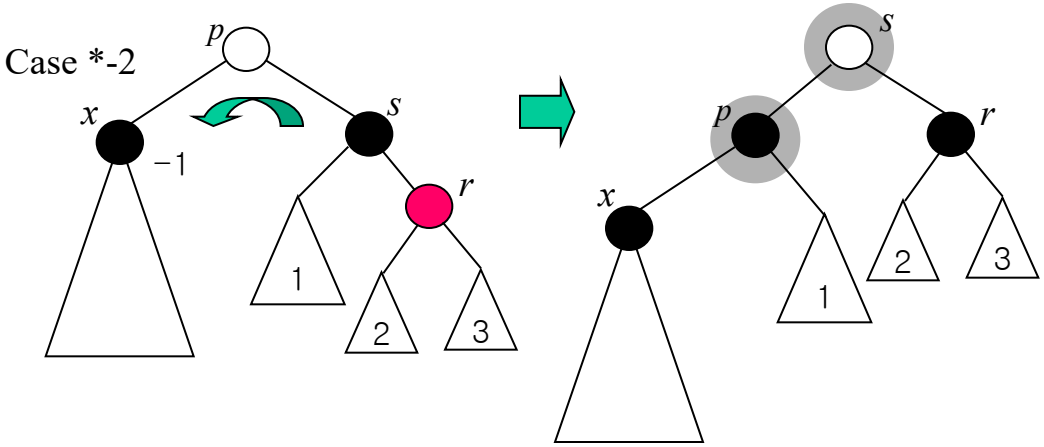
✓최종적으로 5가지 경우로 나뉜다



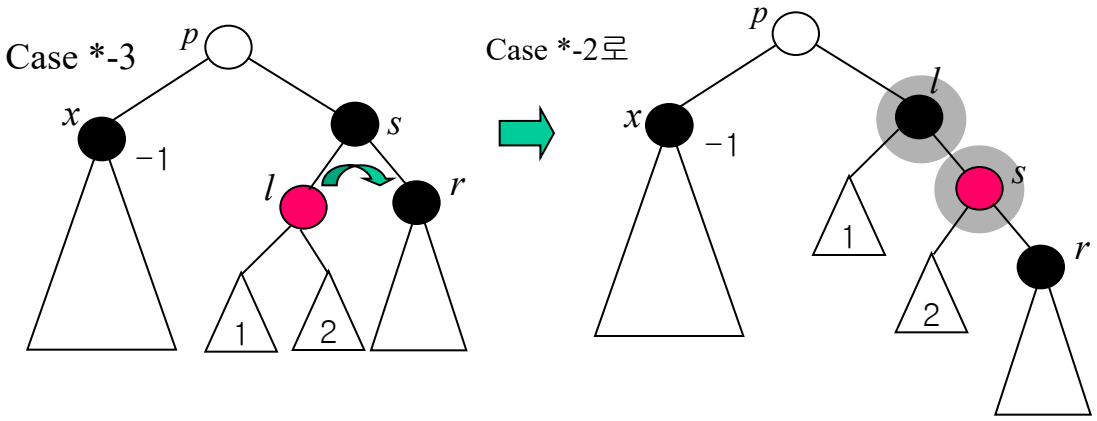
각 경우에 따른 처리

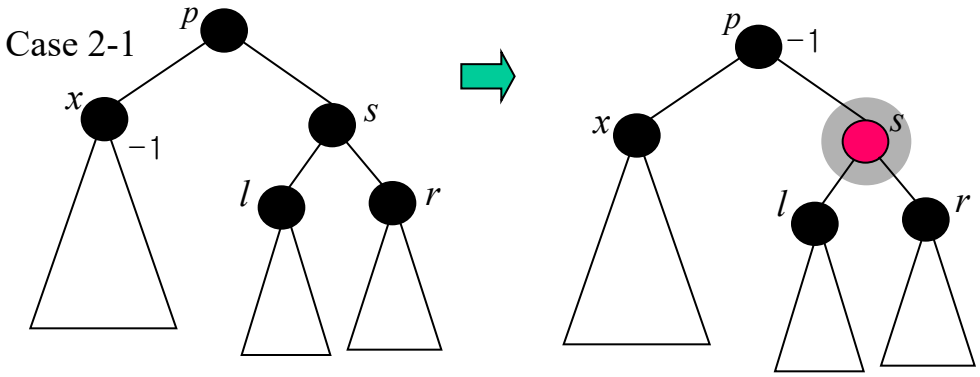


✓ 삭제 완료!

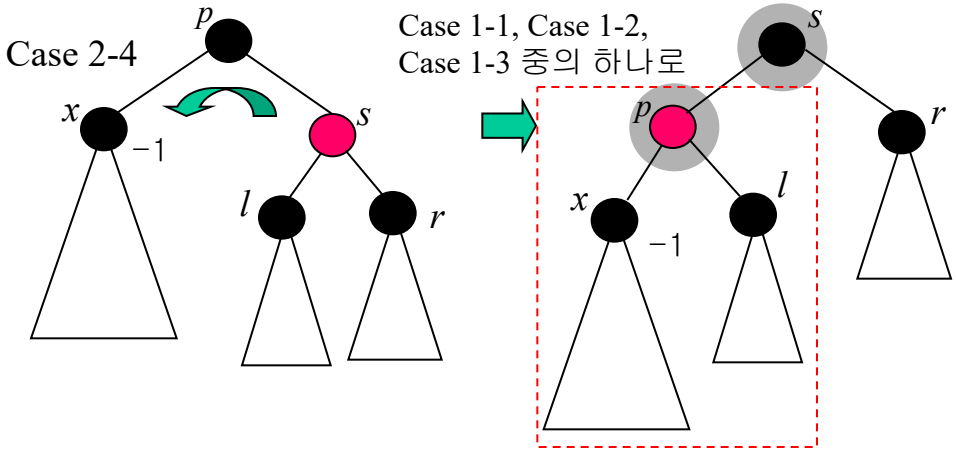


✓ 삭제 완료!





✓ p에서 방금과 같은 문제가 발생: recursive problem!



수행 시간

Case 2-1: $O(\log n)$

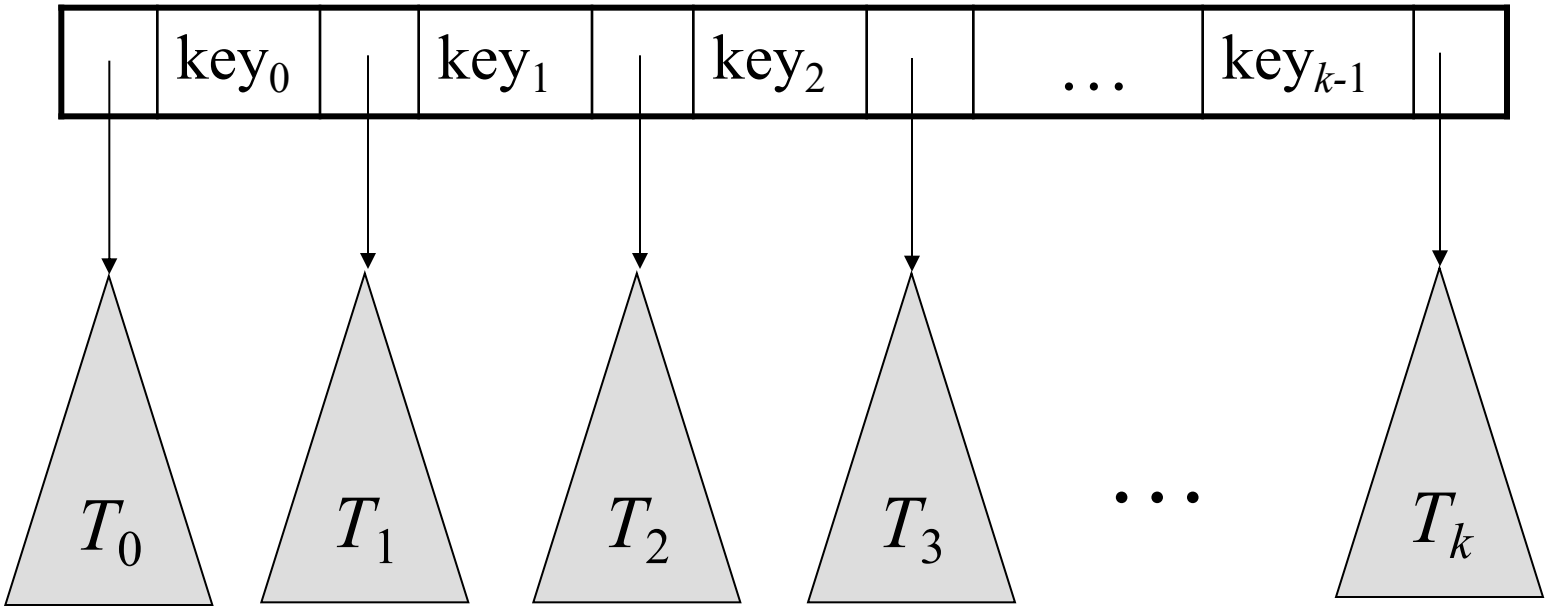
All other cases except Case 2-1: $\Theta(1)$

$\longrightarrow O(\log n)$

B-Trees

- Disk의 접근 단위는 block(page)
- Disk에 한 번 접근하는 시간은 수십만 명령어의 처리 시간과 맞먹는다
- Search tree가 disk에 저장되어 있다면 tree height를 최소화하는 것이 유리하다
- B-tree는 multi-way search tree가 balance를 유지하도록 하여 최악의 경우 disk access 횟수를 줄인 것이다

Multi-Way Search Trees

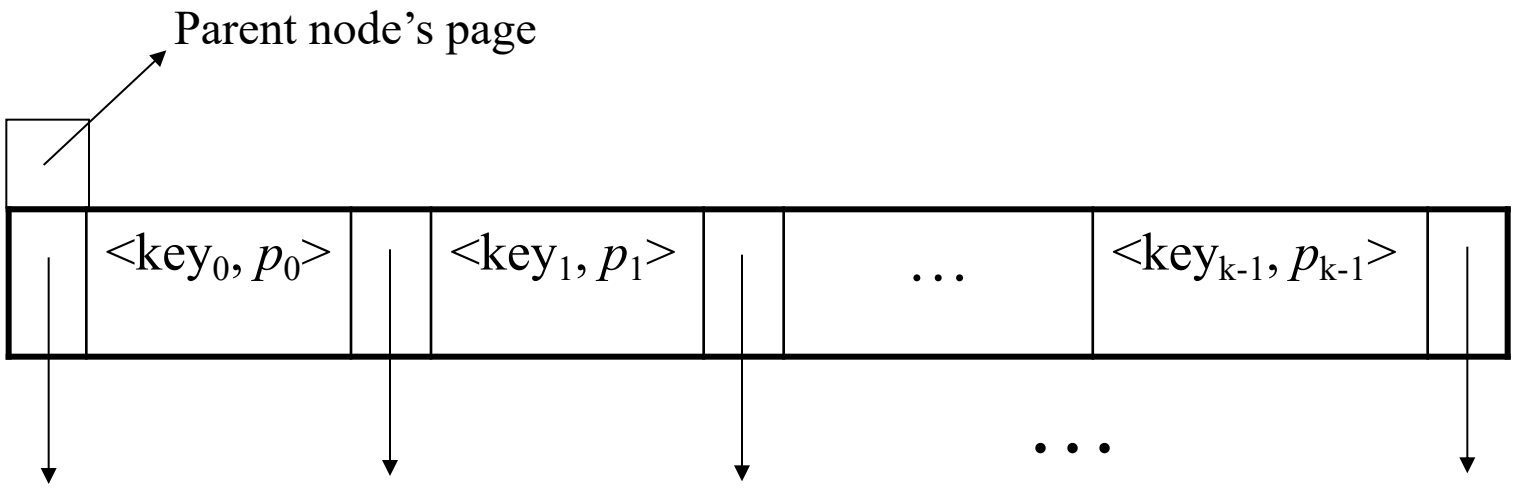


$$\text{key}_{i-1} < T_i < \text{key}_i$$

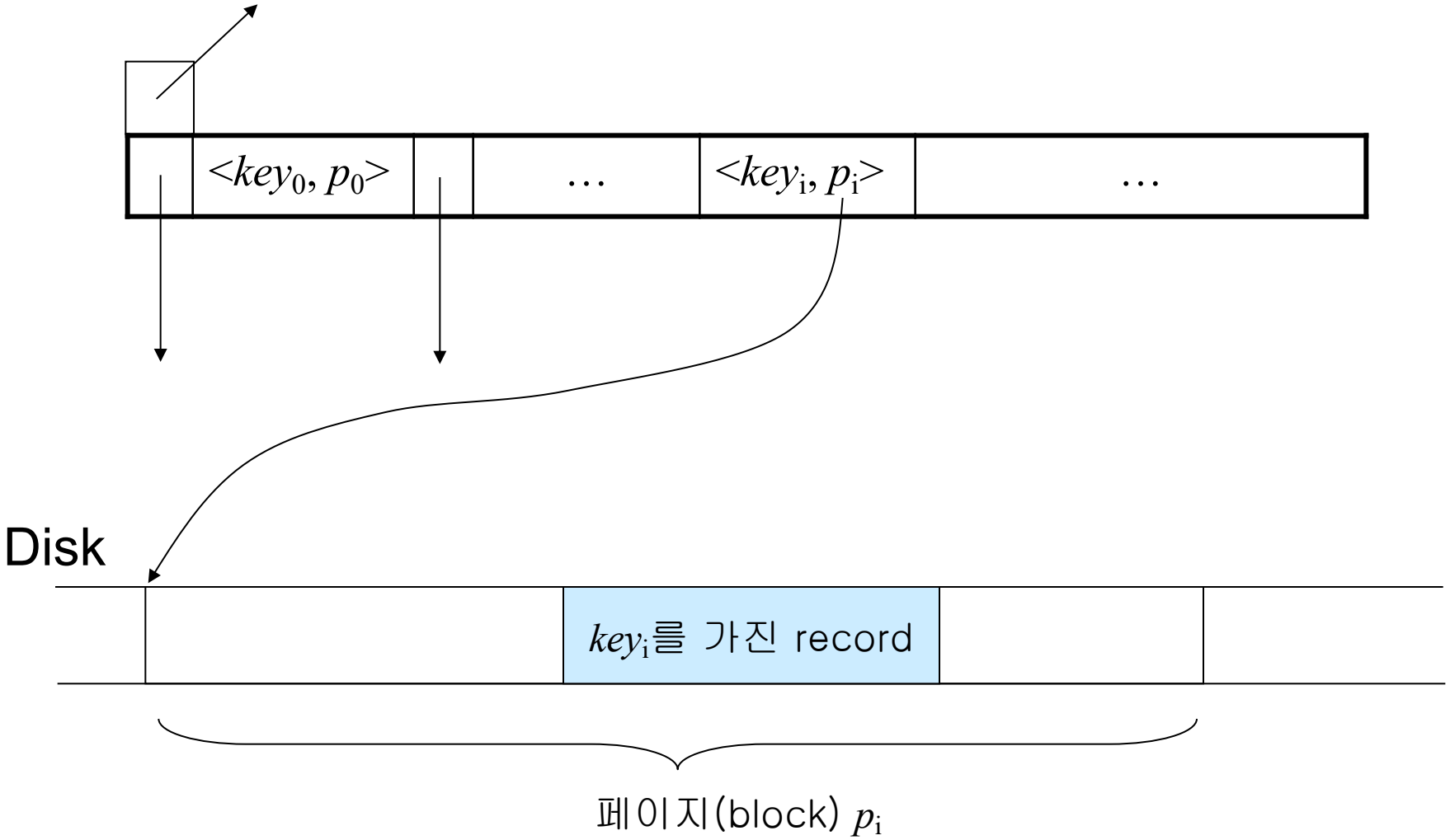
B-Trees

- Balanced multi-way search tree satisfying:
  - Every node except the root has $\lfloor k/2 \rfloor \sim k$ keys
  - Every leaf is located at the same depth

Node Structure of a B-Tree



B-Tree를 통해 record에 접근하는 과정



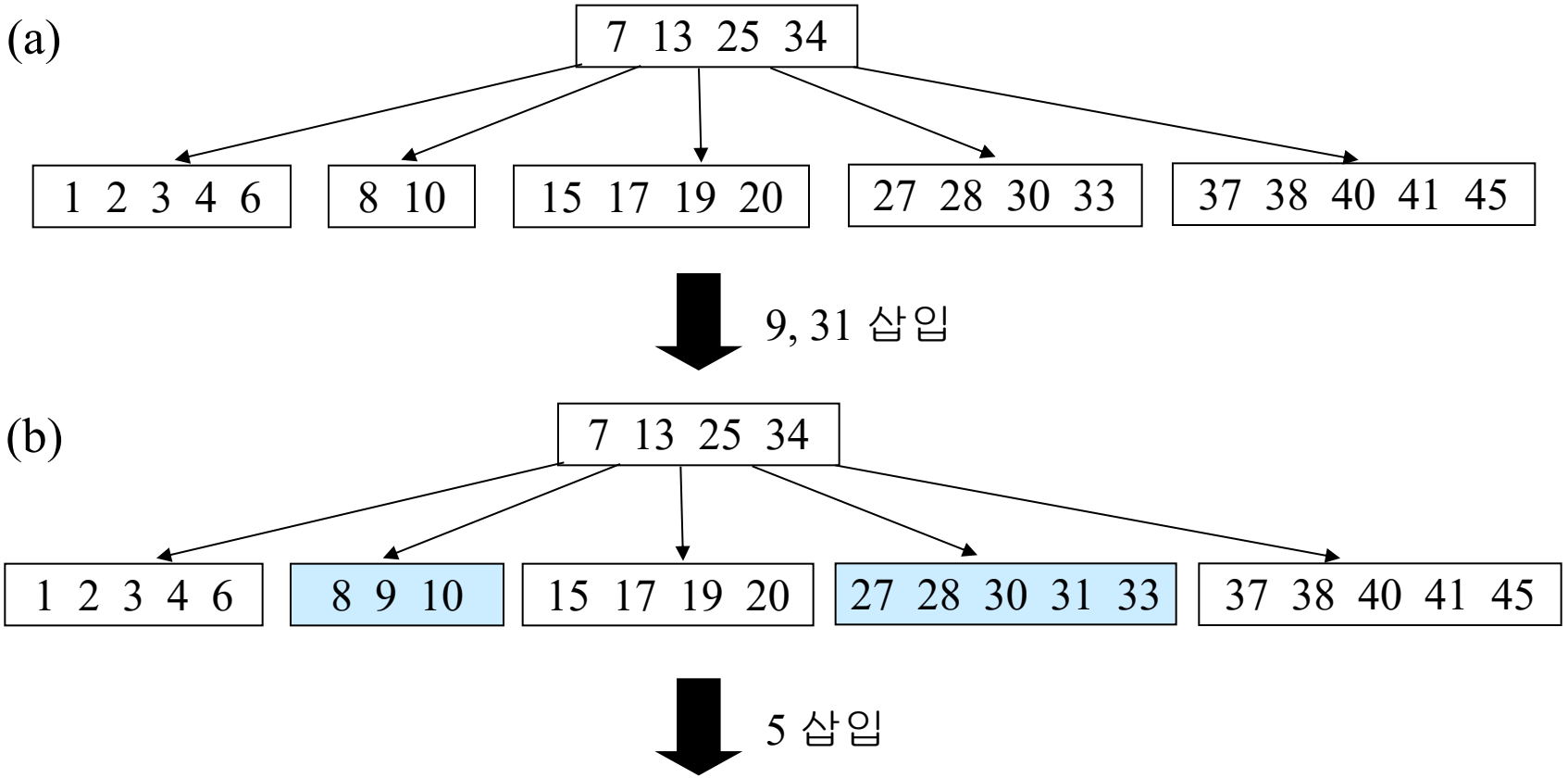
Insertion in B-Trees

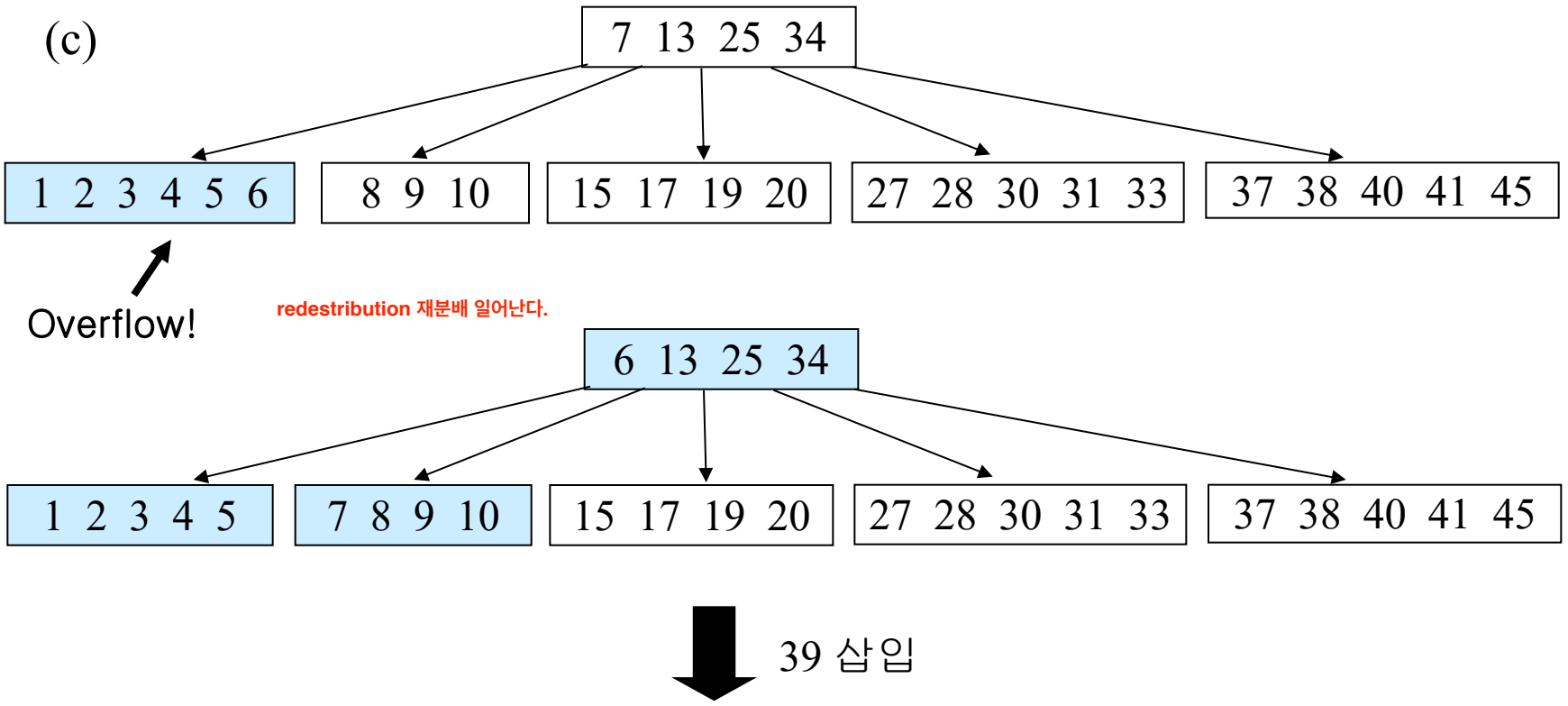
▷ t : 트리의 루트 노드
▷ x : 삽입하고자 하는 키

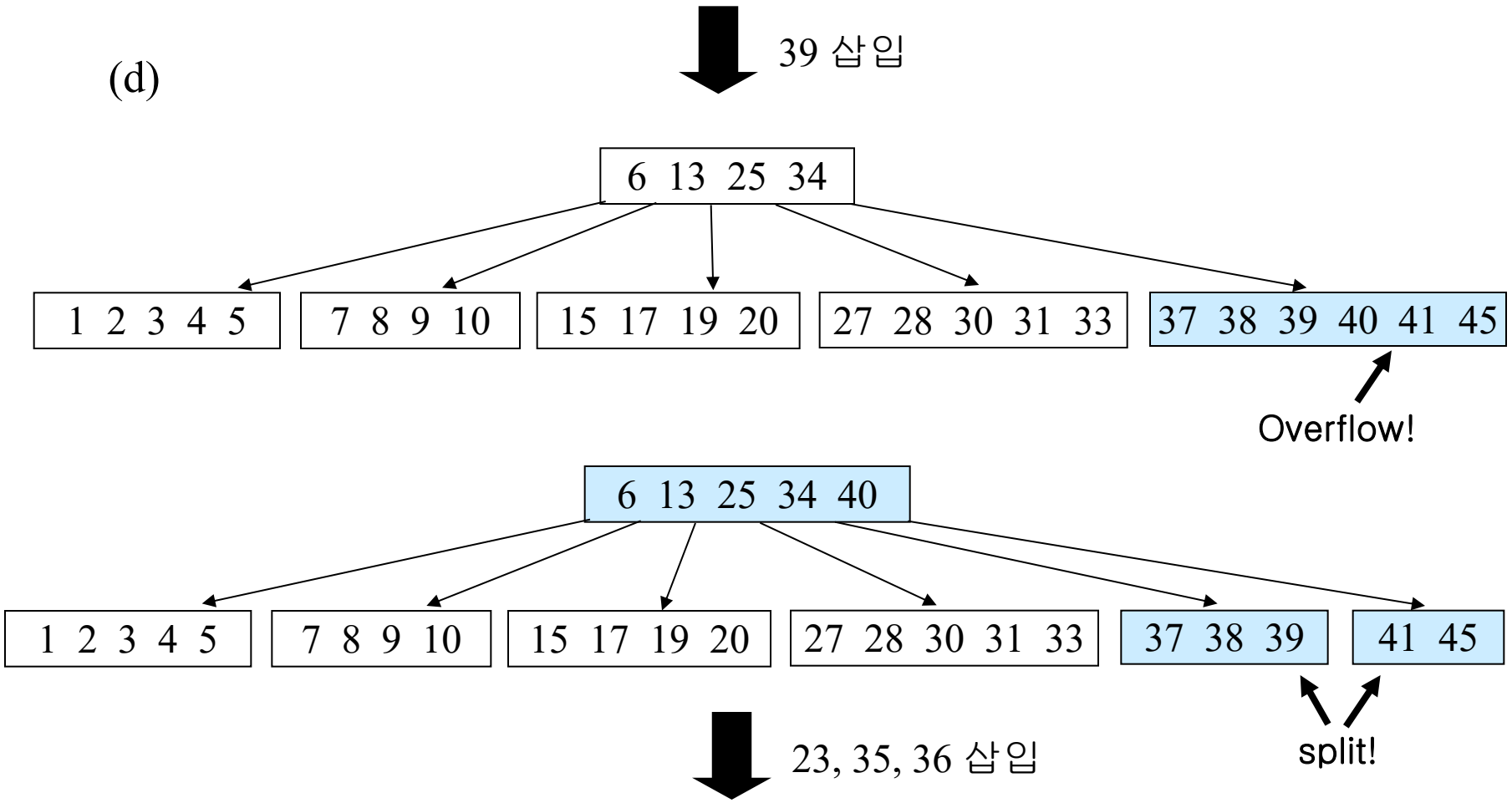
```
BTreeInsert(t, x)
{
        x를 삽입할 리프 노드 r을 찾는다;
        x를 r에 삽입한다;

        if (r에 오버플로우 발생) then clearOverflow(r);
}
clearOverflow(r)
{
    if (r의 형제 노드 중 여유가 있는 노드가 있음) then {r의 남는 키를 넘긴다};
    else {
            r을 둘로 분할하고 가운데 키를 부모 노드로 넘긴다;
            if (부모 노드 p에 오버플로우 발생) then clearOverflow(p);
    }
}
```

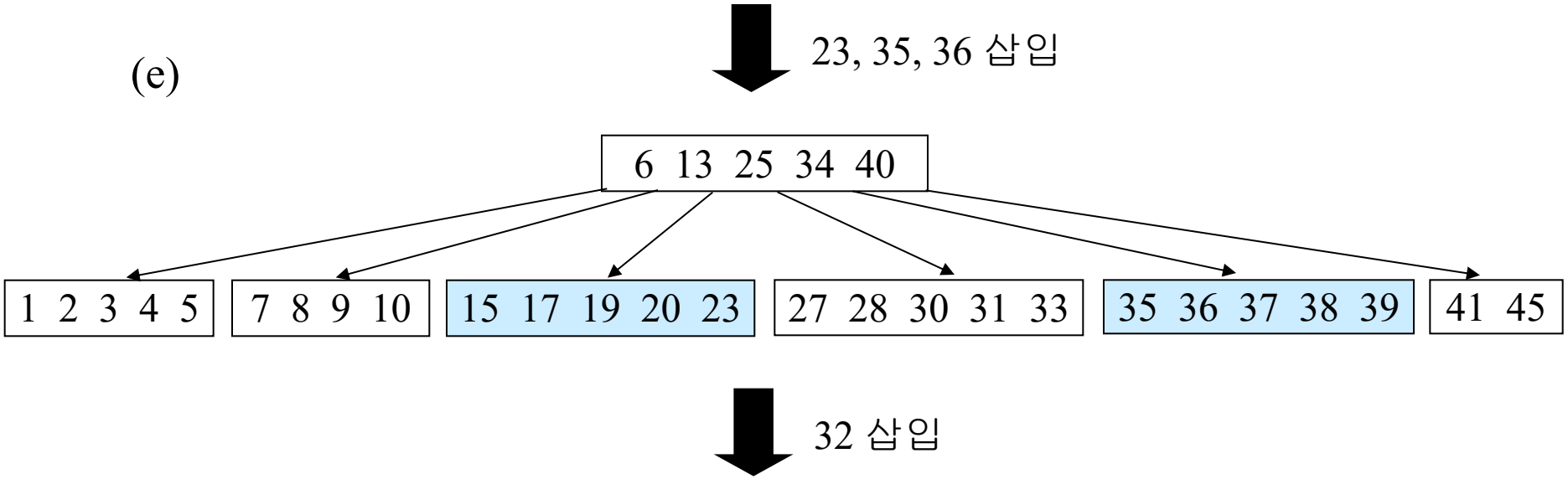
Insertion Example in B-Trees

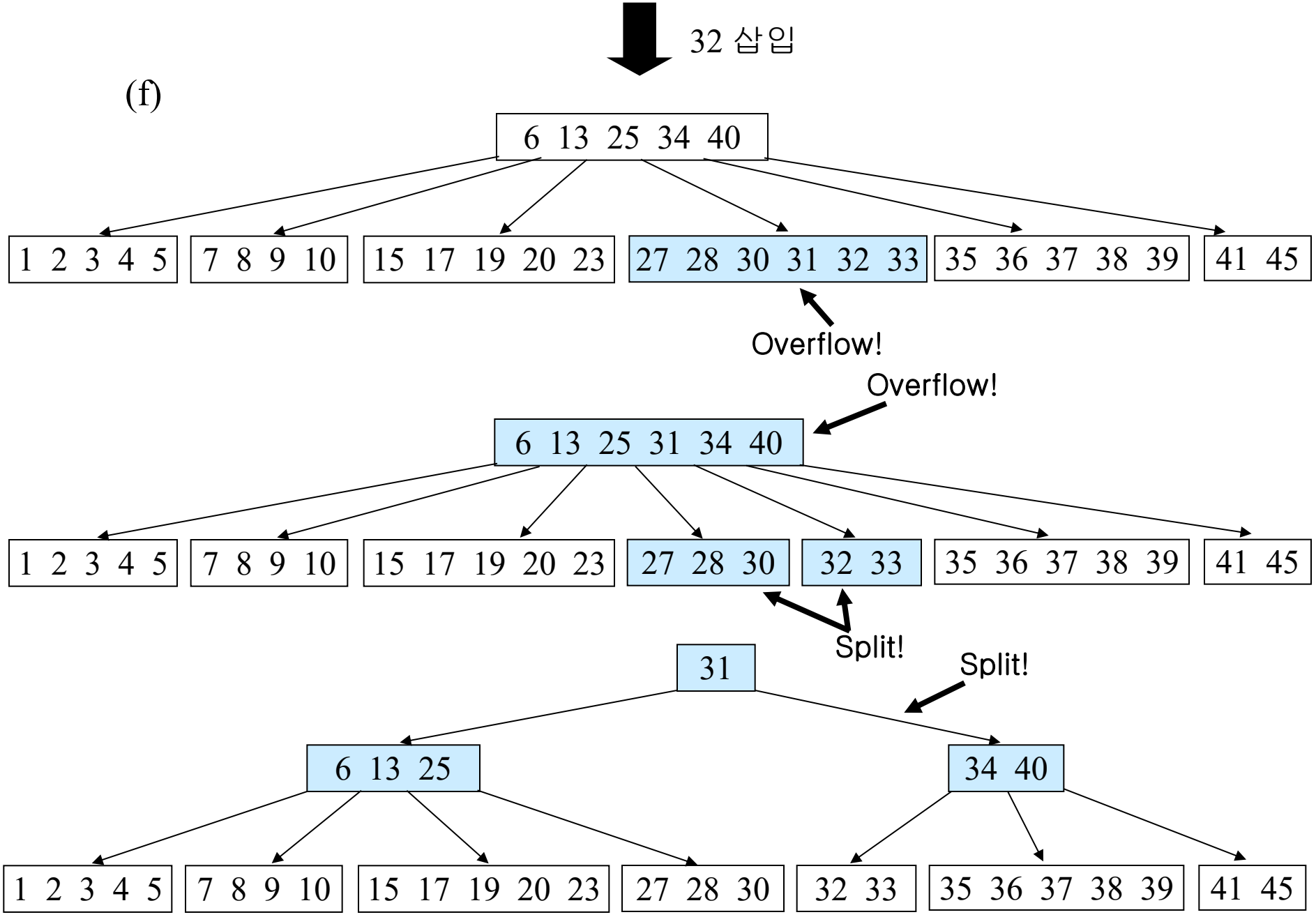



(c)
7 13 25 34
1 2 3 4 5 6
8 9 10
15 17 19 20
27 28 30 31 33
37 38 40 41 45
Overflow!
redestribution 재분배 일어난다.
6 13 25 34
1 2 3 4 5
7 8 9 10
15 17 19 20
27 28 30 31 33
37 38 40 41 45
39 삽입


(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
Overflow!
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
split!
23, 35, 36 삽입

(e)


23, 35, 36 삽입
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
Overflow!
Overflow!
6 13 25 31 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
Split!
Split!
31
6 13 25
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

수행 시간

1차 삽입: $\Theta(\log n)$

수선: $O(\log n)$

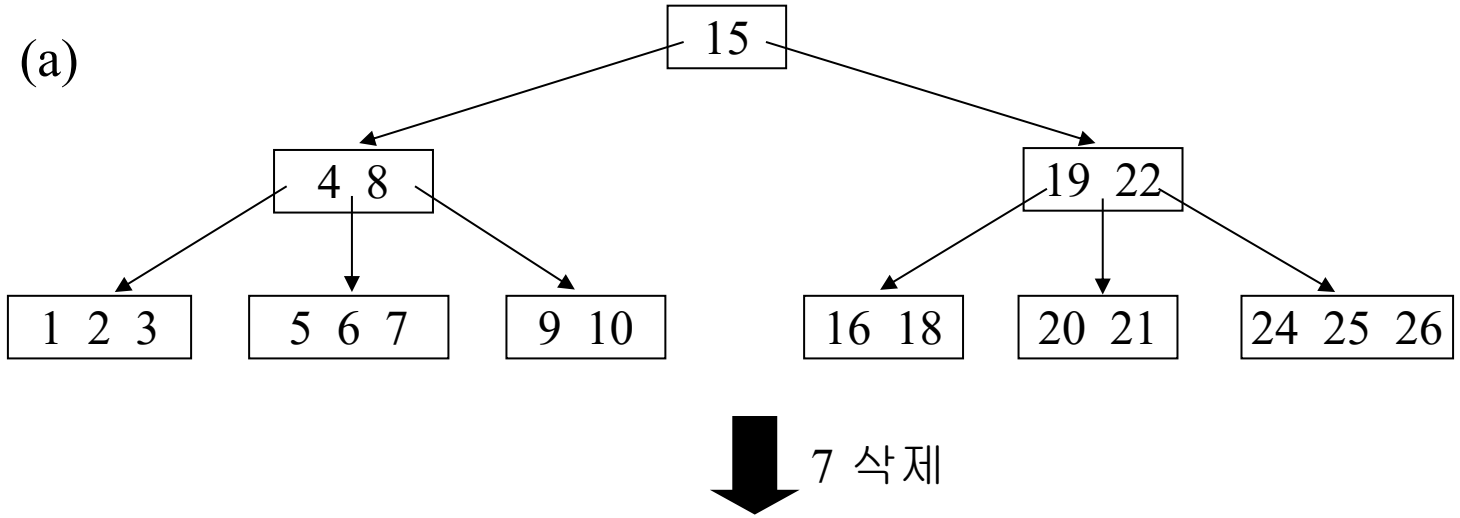
$\Theta(\log n)$ in total

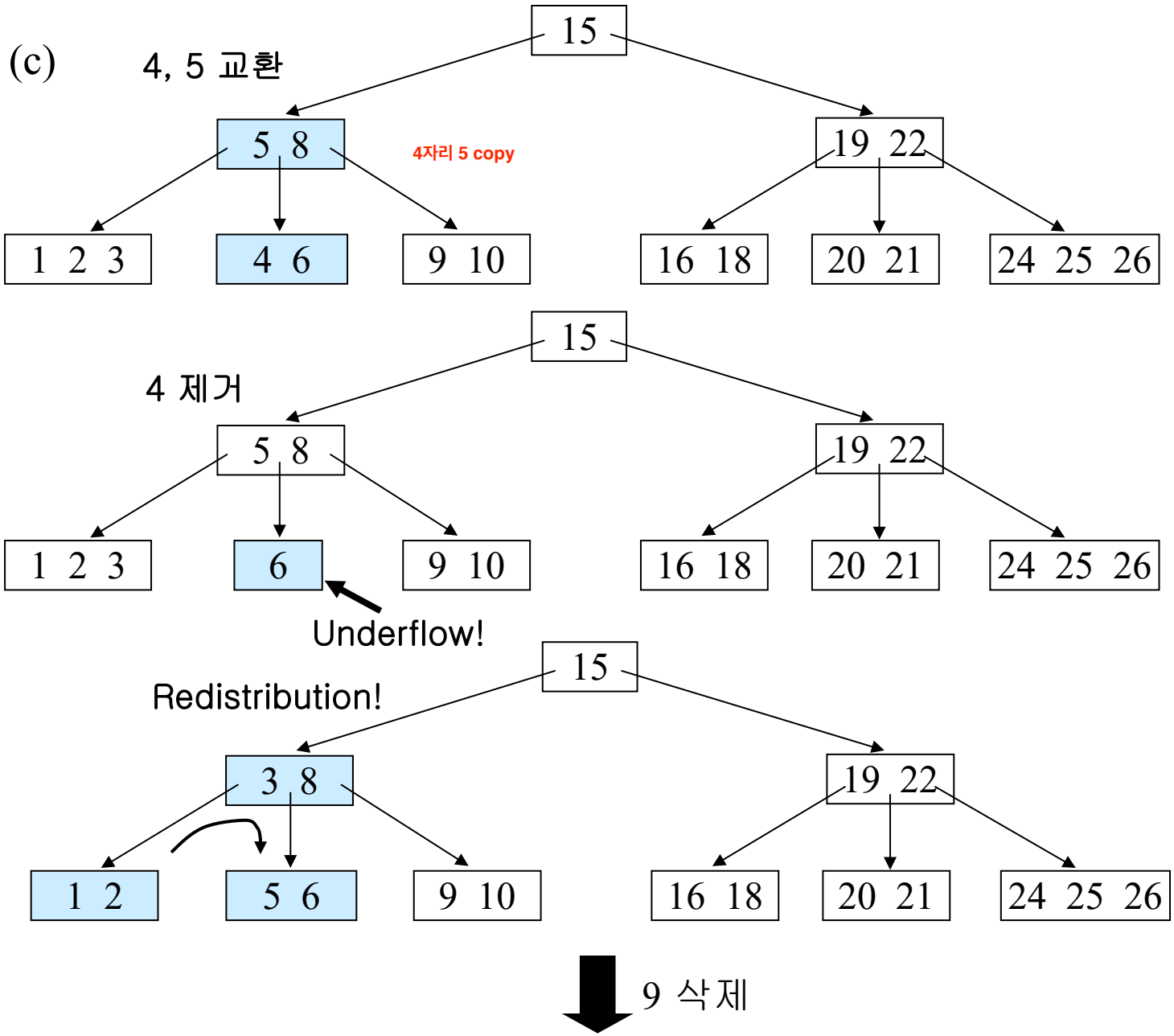
Deletion in B-Trees

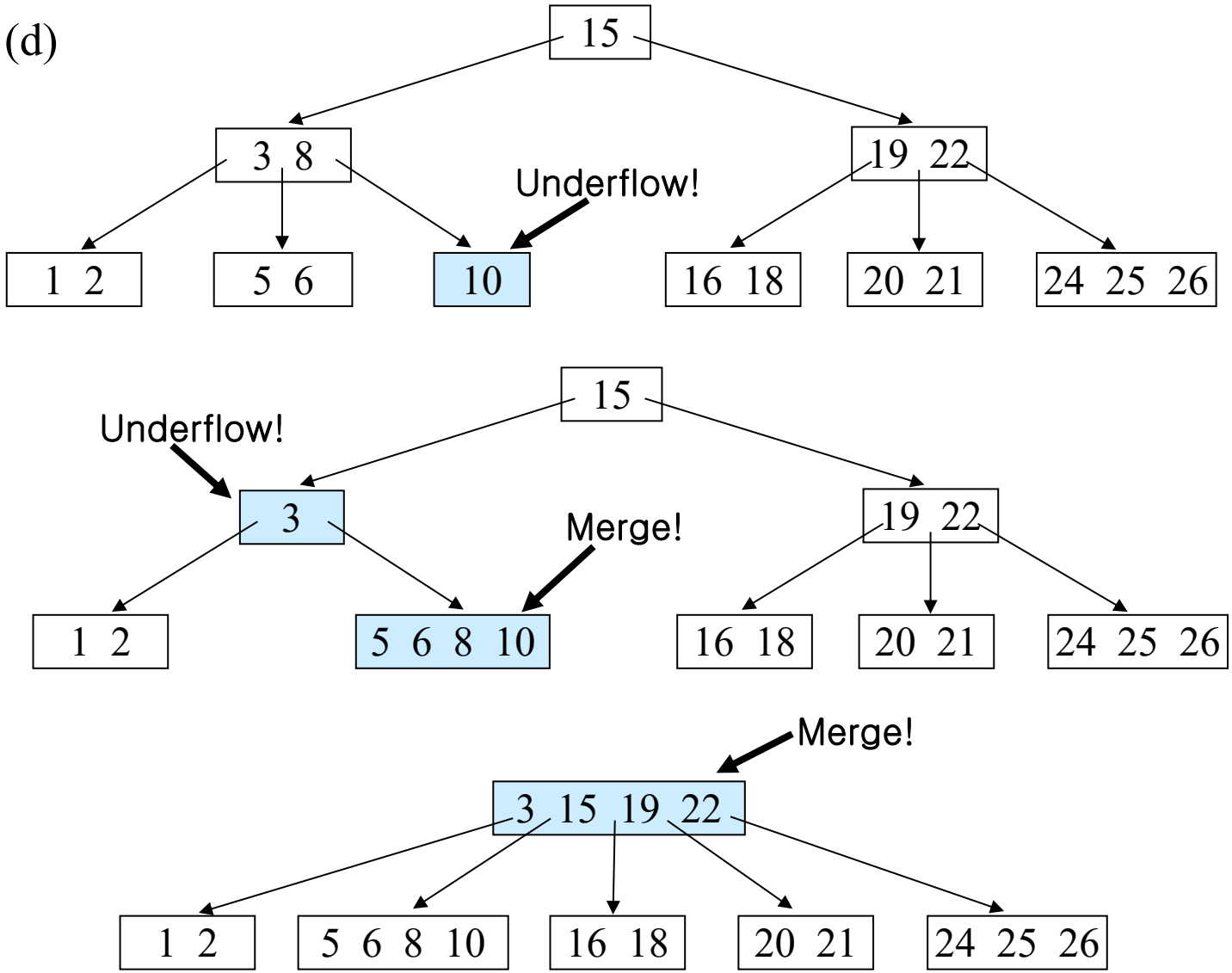
▷ t : 트리의 루트 노드
▷ x : 삭제하고자 하는 키
▷ v : x를 갖고 있는 노드

```
BTreeDelete(t, x, v)
{
    if (v가 리프 노드 아님) then {
        x의 직후원소 y를 가진 리프 노드를 찾는다;
        x와 y를 맞바꾼다;
    }
    리프 노드에서 x를 제거하고 이 리프 노드를 r이라 한다;
    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 키를 넘겨받는다;}
        else {
                r의 형제 노드와 r을 합병한다;
                if (부모 노드 p에 언더플로우 발생) then clearUnderflow(p);
        }
}
```

Deletion Example in B-Trees




(c)
4, 5 교환
15
5 8
4자리 5 copy
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
Underflow!
Redistribution!
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
Underflow!
1 2
5 6
10
16 18
20 21
24 25 26
15
Underflow!
3
Merge!
19 22
1 2
5 6 8 10
16 18
20 21
24 25 26
Merge!
3 15 19 22
1 2
5 6 8 10
16 18
20 21
24 25 26

수행 시간

삭제 원소가 leaf node에 있고,
수선 없이 해결될 때: $\Theta(1)$

직후 원소 찾기: $O(\log n)$

수선: $O(\log n)$

삭제 원소 검색까지 포함하면 $\Theta(\log n)$

$O(\log n)$ in total