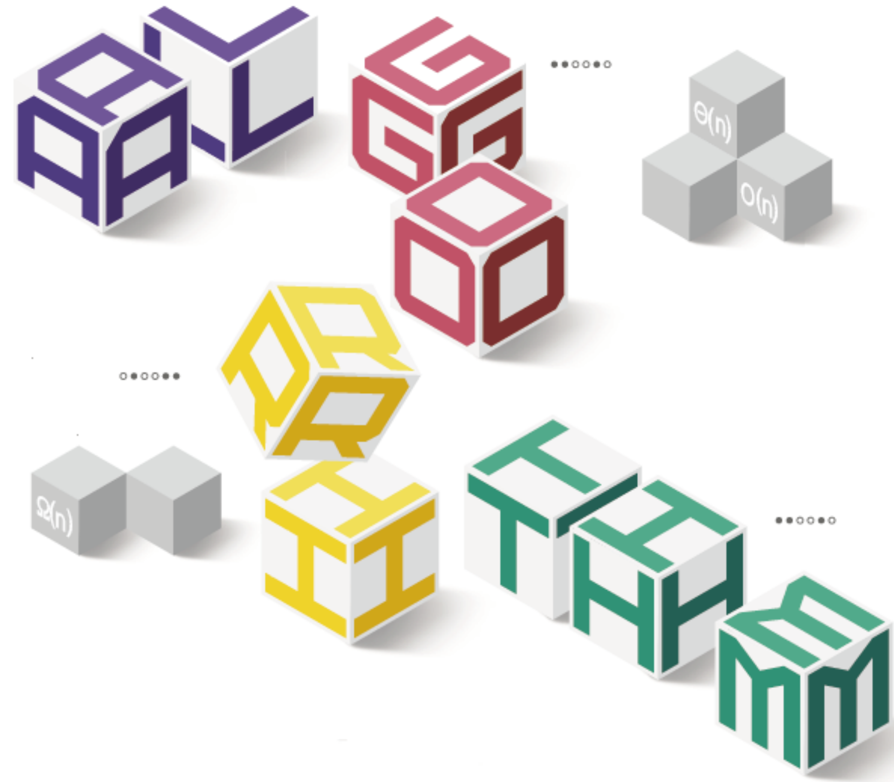


관계 중심의 사고법

쉽게 배우는 알고리즘

6장. 검색트리

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

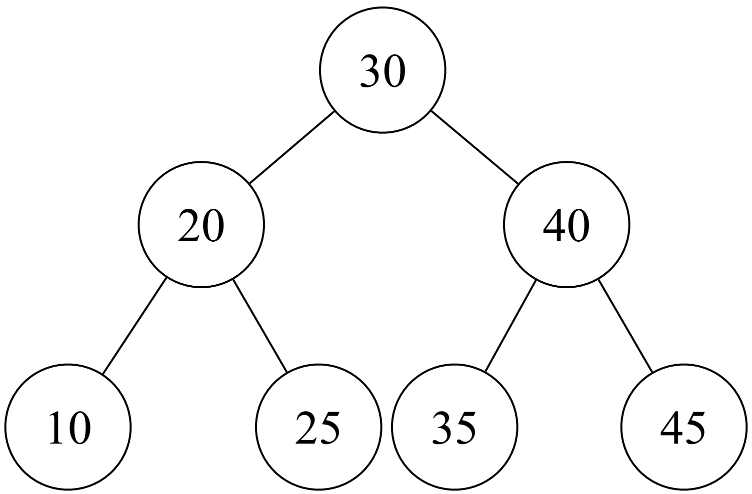
레코드, 키, 검색트리

- 레코드record
  - 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
- 필드field
  - 레코드에서 각각의 정보를 나타내는 부분
  - e.g., 위 사람의 레코드에서 각각의 정보를 나타내는 부분
- 검색키search key 또는 키key
  - 다른 레코드와 중복되지 않도록 각 레코드를 대표할 수 있는 필드
  - 키는 하나의 필드로 이루어질 수도 있고, 두 개 이상의 필드로 이루어질 수도 있다
- 검색트리search tree
  - 각 노드가 규칙에 맞도록 하나씩의 키를 갖고 있다
  - 이를 통해 해당 레코드가 저장된 위치를 알 수 있다

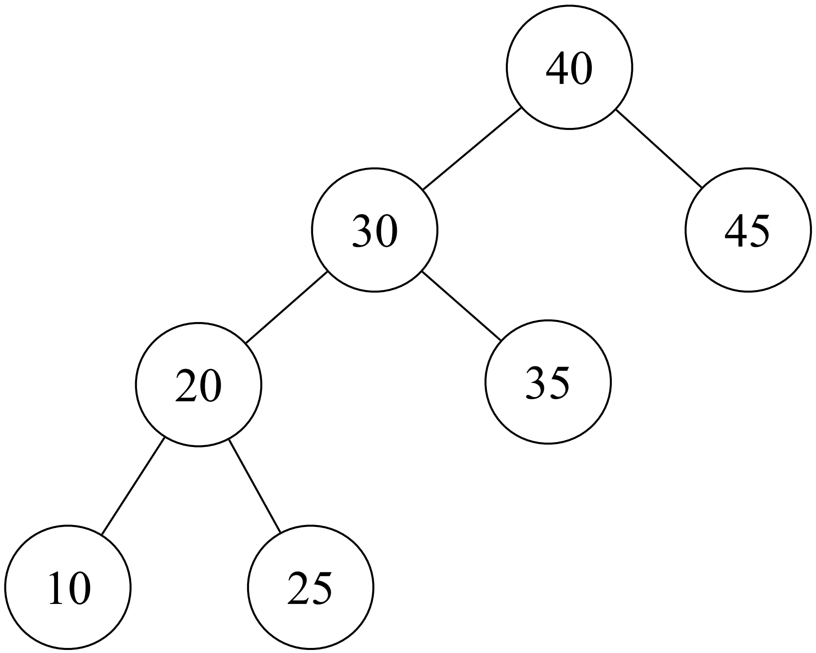
이진검색트리

- 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

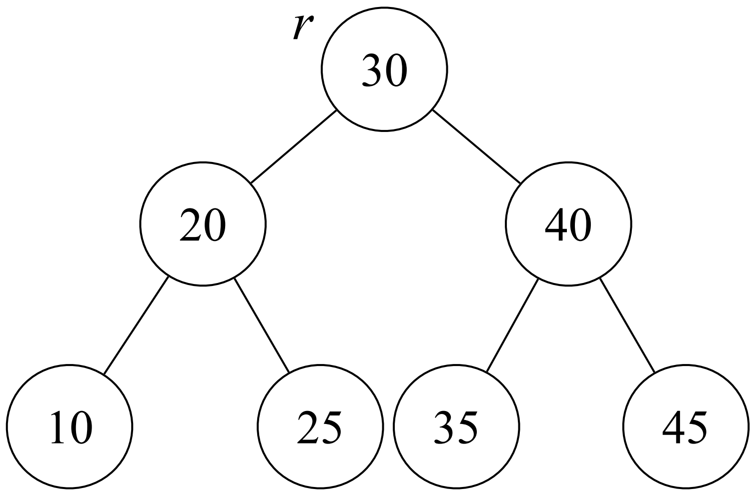


(a)

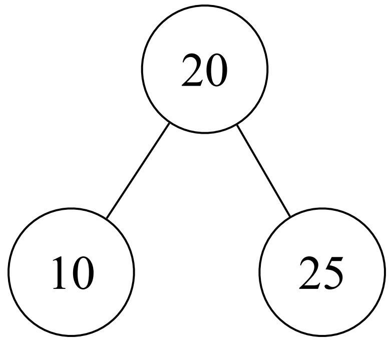


(b)

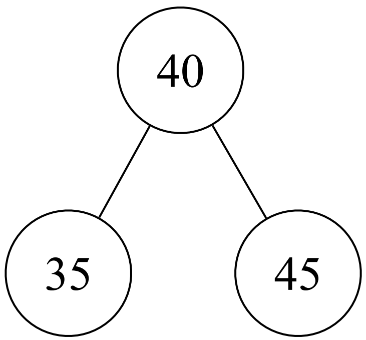
서브트리의 예



(a)



(b) 노드 r의 왼쪽 서브트리

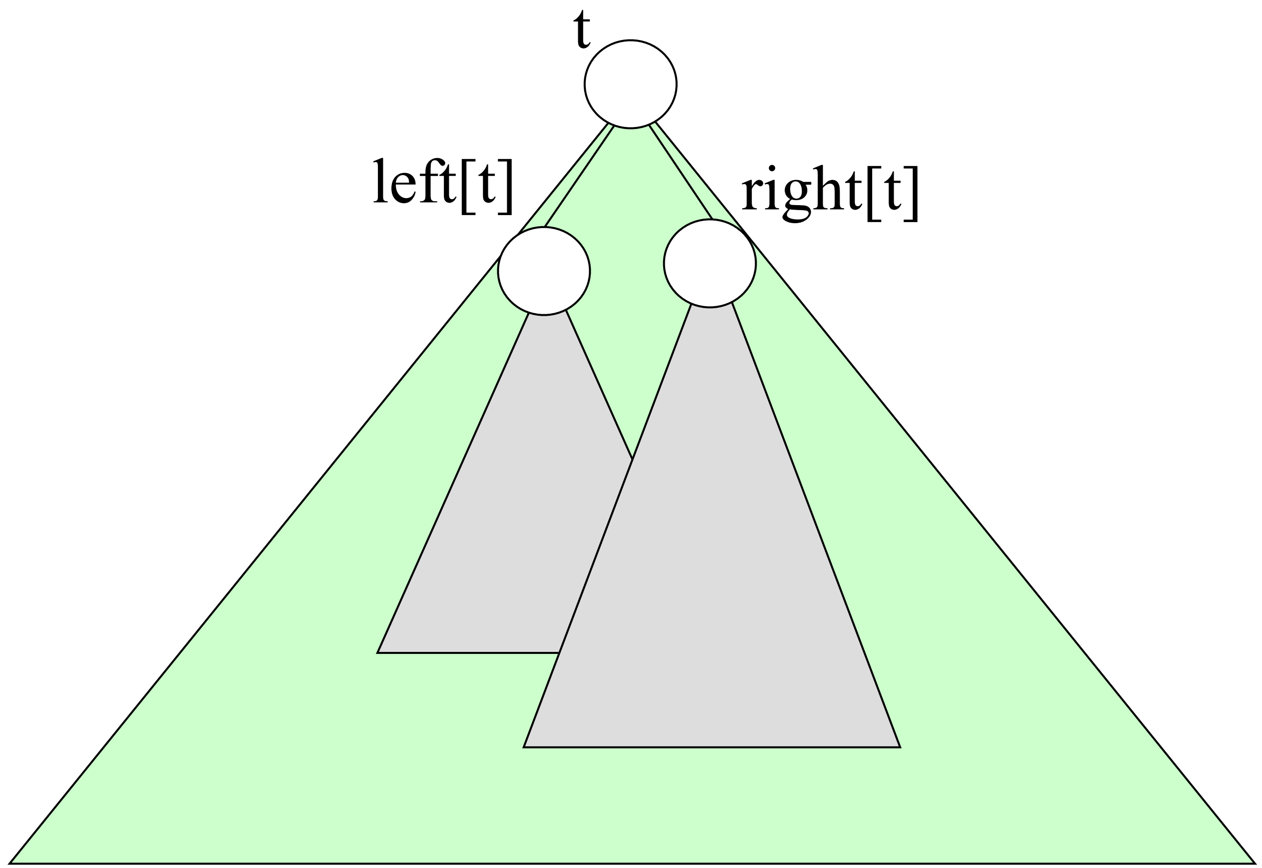


(c) 노드 r의 오른쪽 서브트리

이진검색트리에서의 검색

```
treeSearch(t, x)
▷ t: 트리의 루트 노드
▷ x: 검색하고자 하는 키
{
        if (t=NIL or key[t]=x) then return t;
        if (x < key[t])
                then return treeSearch(left[t], x);
                else return treeSearch(right[t], x);
}
```

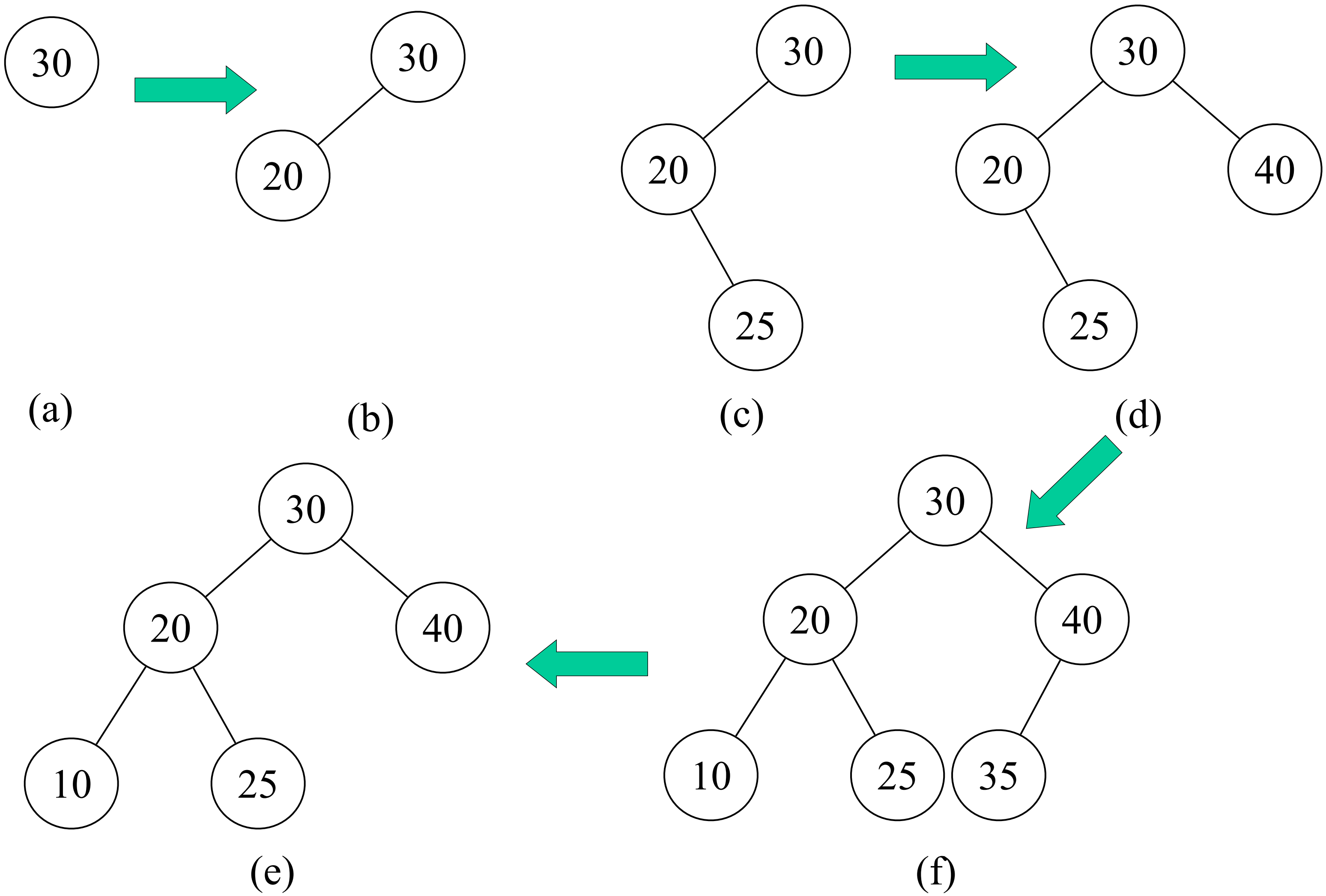
검색에서 재귀적 관점



이진검색트리에서의 삽입

```
treeInsert(t, x)
▷ t: 트리의 루트 노드
▷ x: 삽입하고자 하는 키
▷ 작업 완료 후 루트 노드의 포인터를 리턴한다
{
    if (t=NIL) then {
        key[r] ← x;  left[r] ← NIL;  right[r] ← NIL;    ▷ r : 새 노드
        return r ;
    }
    if (x < key(t))
        then {left[t] ← treeInsert(left[t], x); return t;}
        else {right[t] ← treeInsert(right[t], x); return t;}
}
```

삽입의 예



이진검색트리에서의 삭제

t: 트리의 루트 노드
r: 삭제하고자 하는 노드

- 3가지 경우에 따라 다르게 처리한다
  - Case 1 : r이 리프 노드인 경우
  - Case 2 : r의 자식 노드가 하나인 경우
  - Case 3 : r의 자식 노드가 두 개인 경우

이진검색트리에서의 삭제

```
Sketch-TreeDelete(t, r)
▷ t: 트리의 루트 노드
▷ x: 삭제하고자 하는 키
{
      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;
}
```

이진검색트리에서의 삭제

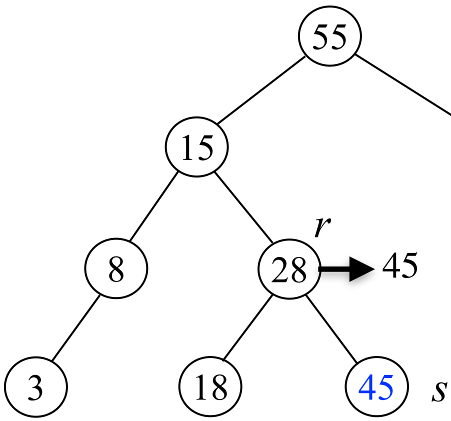
t: 트리의 루트 노드
r: 삭제하고자 하는 노드
p: *r*의 부모 노드

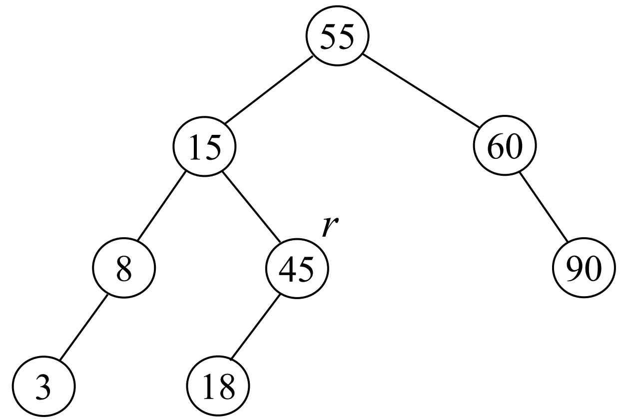
```
treeDelete(t, r, p)
{
    if (r = t) then root ← deleteNode(t);           ▷ r이 루트 노드인 경우
    else if (r = left[p])                           ▷ r이 루트가 아닌 경우
            then left[p] ← deleteNode(r);                ▷ r이 p의 왼쪽 자식
            else right[p] ← deleteNode(r);               ▷ r이 p의 오른쪽 자식
}
deleteNode(r)
{
    if (left[r] = right[r] = NIL) then return NIL;                        ▷ Case 1
    else if (left[r] = NIL and right[r] ≠ NIL) then return right[r];      ▷ Case 2-1
    else if (left[r] ≠ NIL and right[r] = NIL) then return left[r];       ▷ Case 2-2
    else {                                                                ▷ Case 3
            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;
    }
}
```

```c
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->data)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->data)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
```

```
    else
    {
        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->data = temp->data;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

이진검색트리에서의 삭제

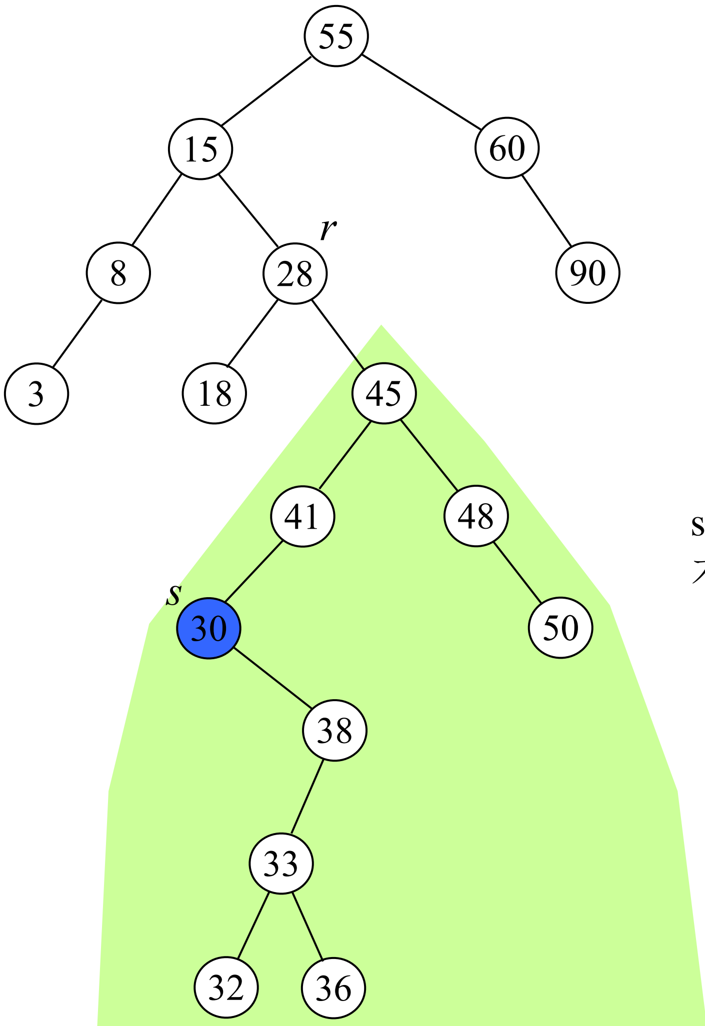




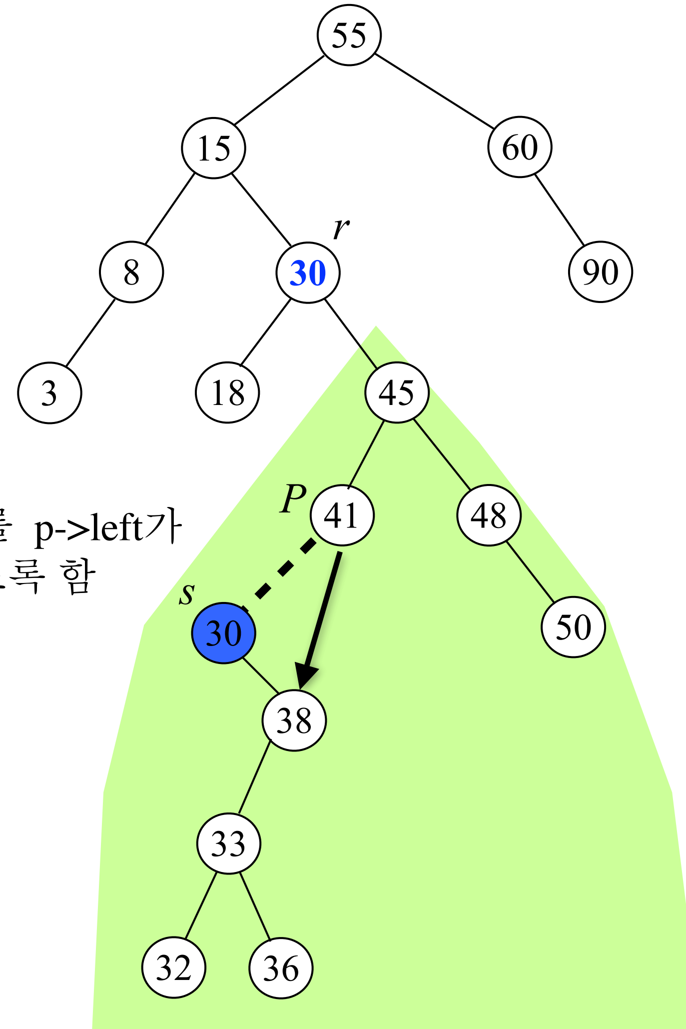
s->right (NIL)를 r->right 가 가리킴

if (s = right[r]) **then** **right[r] ← right[s]; // s**가 단말로드인 경우 **right[s]** 는 **NIL, right[r]**은 **NIL**
else left[parent] ← right[s];

이진검색트리에서의 삭제

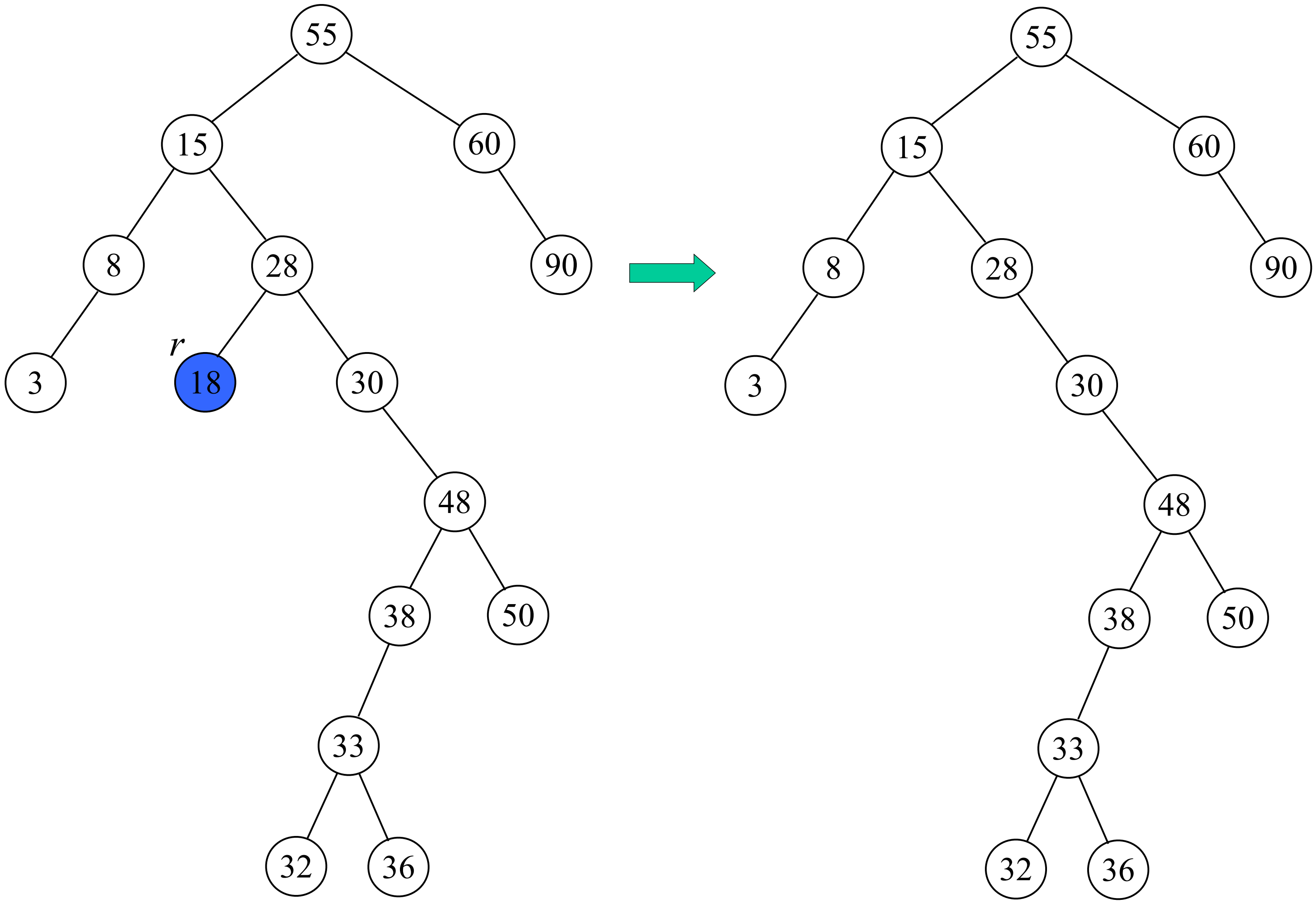


s->right를 p->left가
가리키도록 함



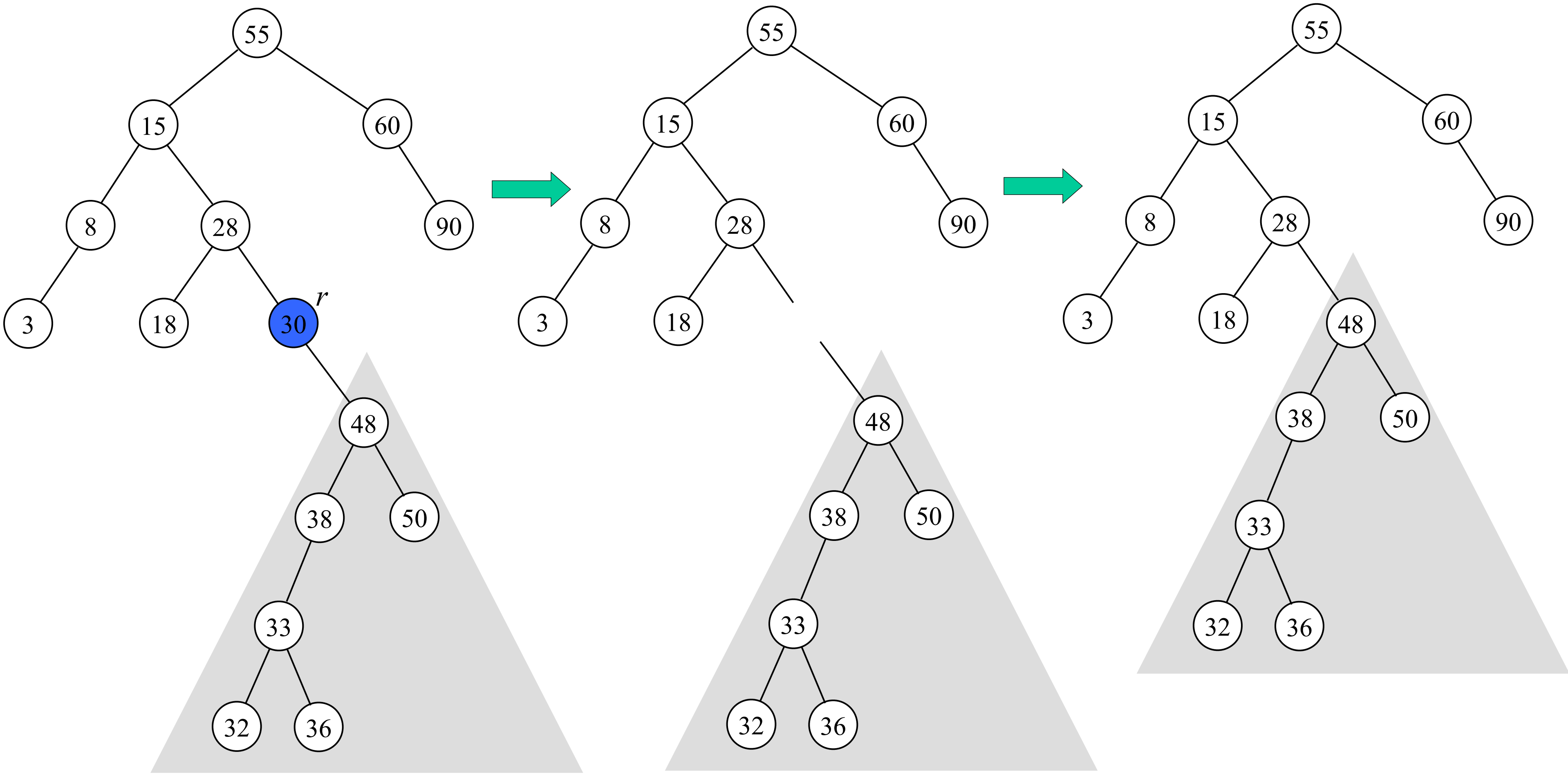
if (s = right[r]) **then** right[r] ← right[s]; // s가 단말로드인 경우
else left[parent] ← right[s]; // s가 자식 있는 경우

삭제의 예: Case 1



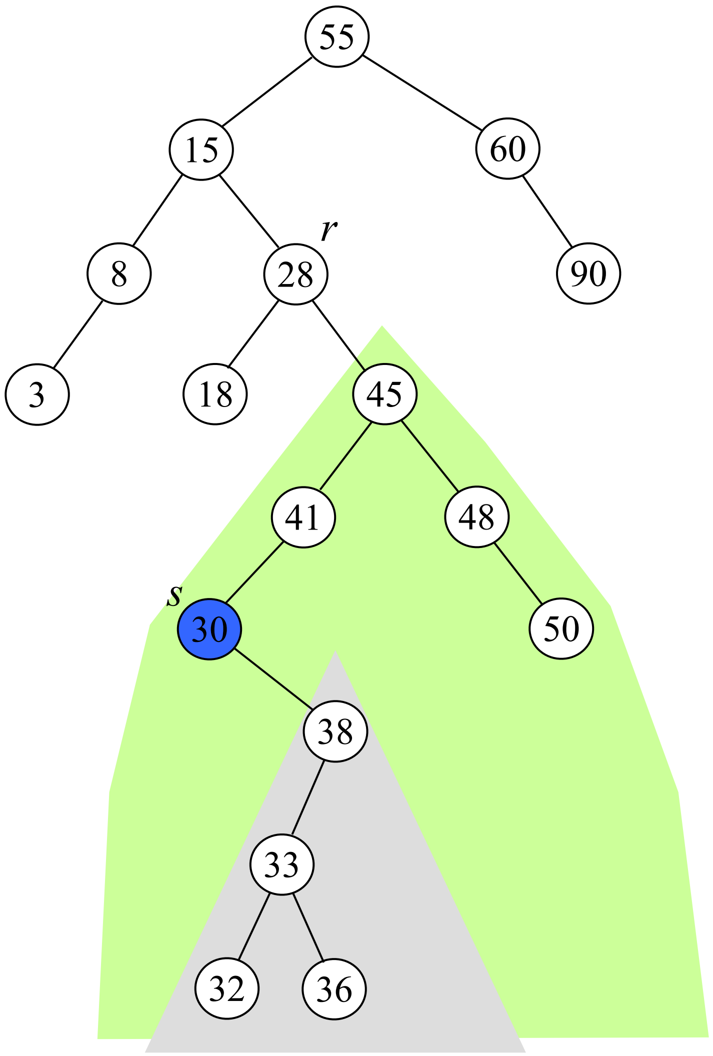
(a) *r*의 자식이 없음　　　　(b) 단순히 *r*을 제거한다

삭제의 예: Case 2

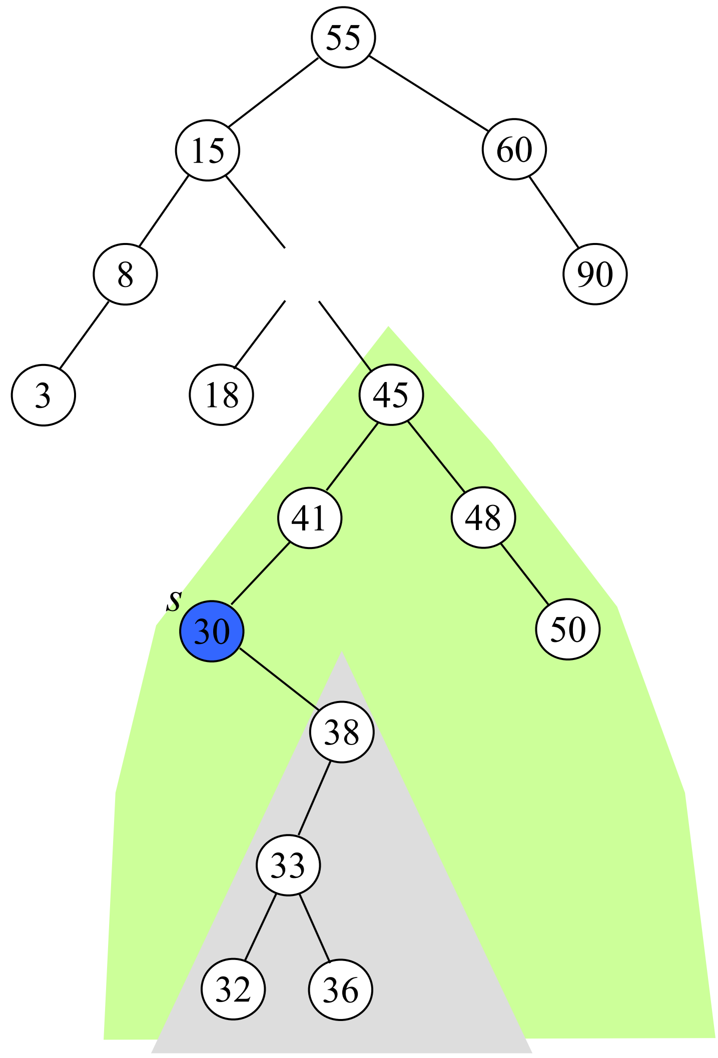


(a) *r*의 자식이 하나뿐임 (b) *r*을 제거 (c) *r* 자리에 *r*의 자식을 놓는다

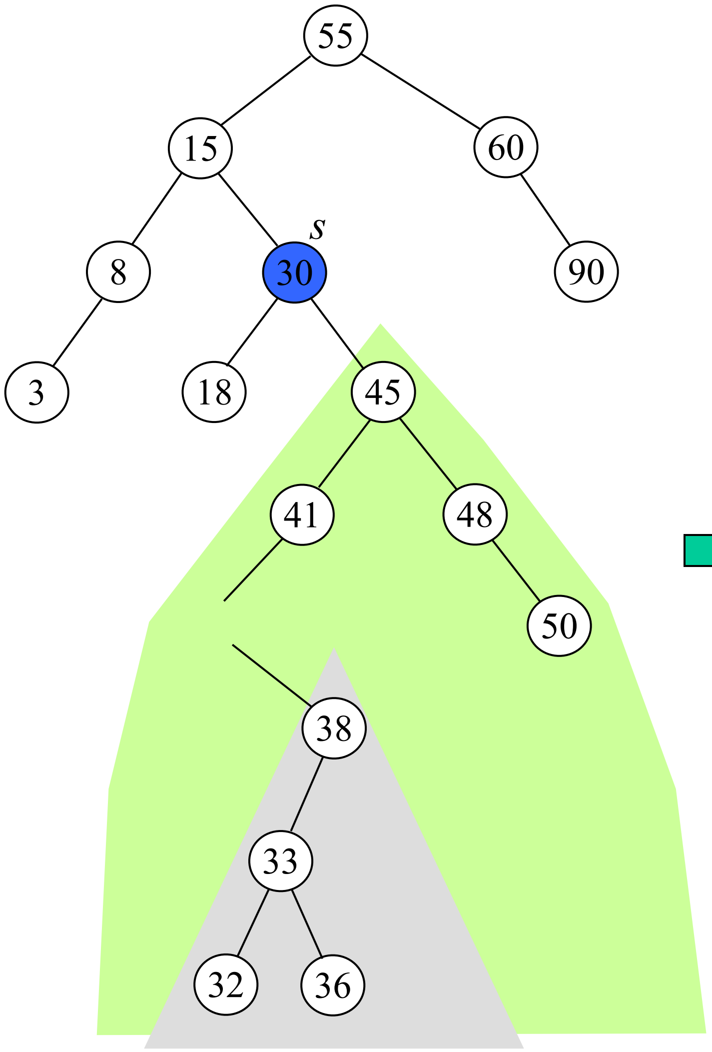
삭제의 예: Case 3



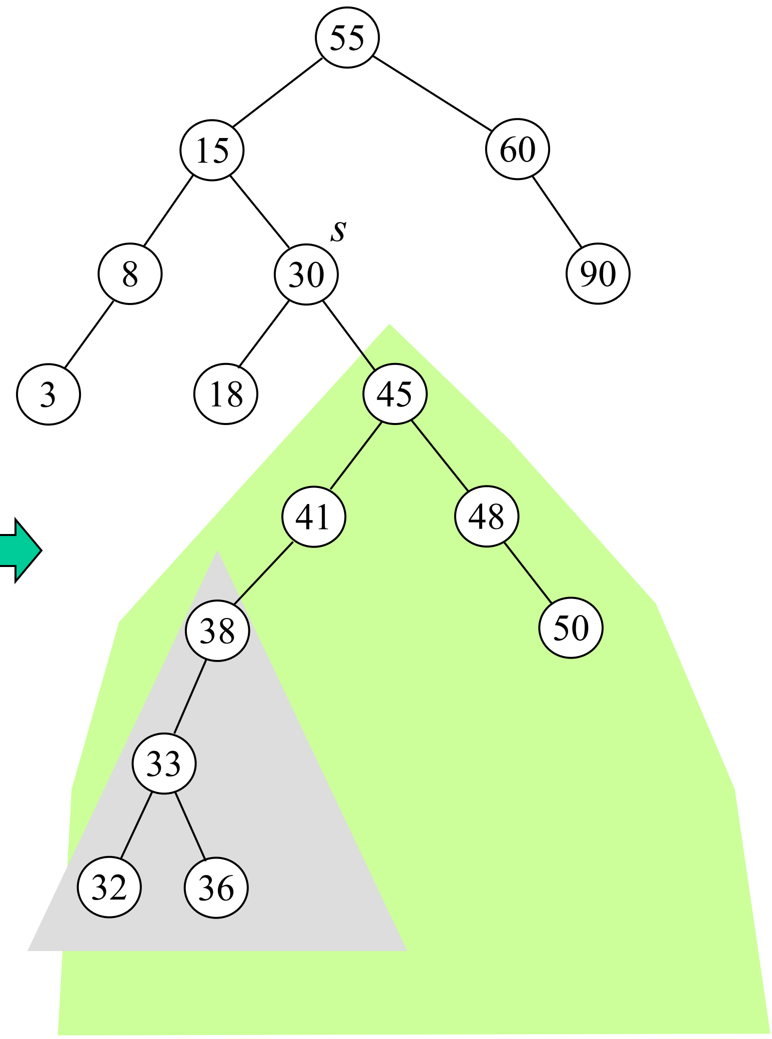
(a) *r*의 직후원소 *s*를 찾는다



(b) *r*을 없앤다



(c) s를 r자리로 옮긴다



(d) s가 있던 자리에 s의 자식을 놓는다

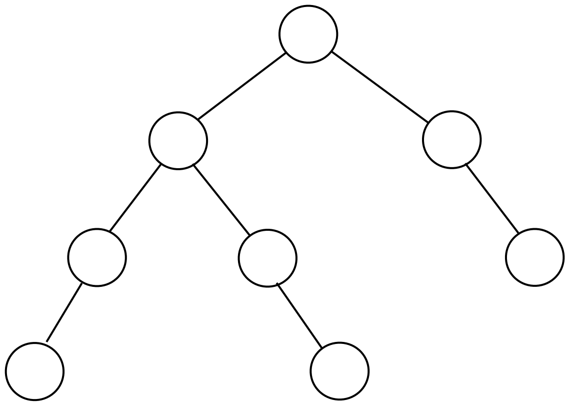
레드블랙트리

- 이진검색트리의 모든 노드에 블랙 또는 레드의 색을 칠하되 다음의 레드블랙 특성을 만족해야 한다

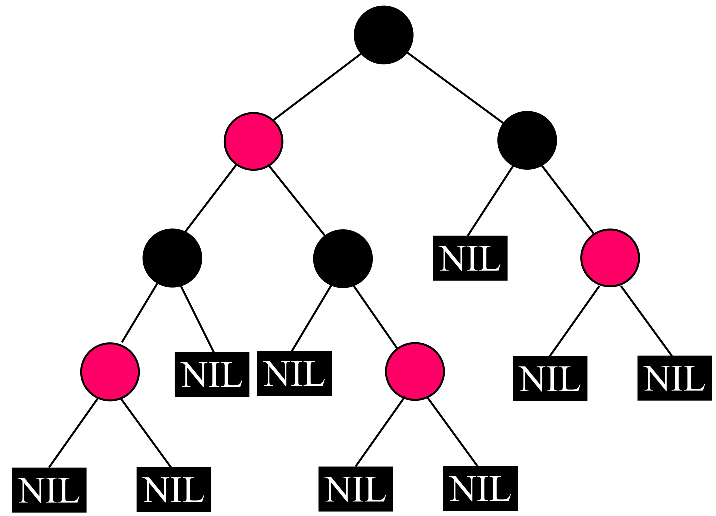
① 루트는 블랙이다
② 모든 리프는 블랙이다
③ 노드가 레드이면 그 노드의 자식은 반드시 블랙이다
④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다

✓ 여기서 리프 노드는 일반적인 의미의 리프 노드와 다르다.
모든 NIL 포인터가 NIL이라는 리프 노드를 가리킨다고 가정한다.

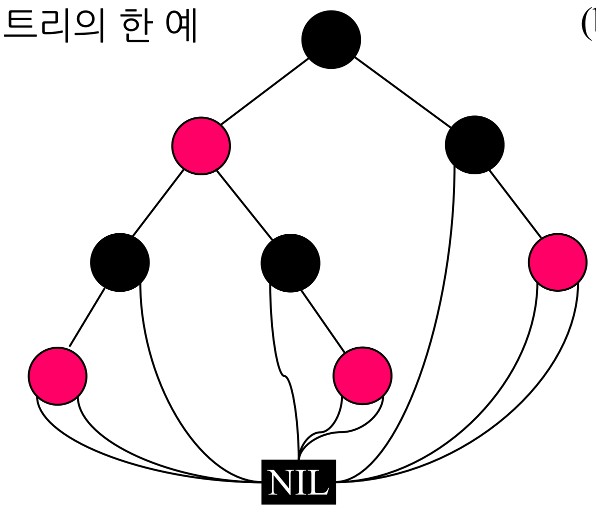
이진검색트리를 레드블랙트리로 만든 예



(a) 이진검색트리의 한 예



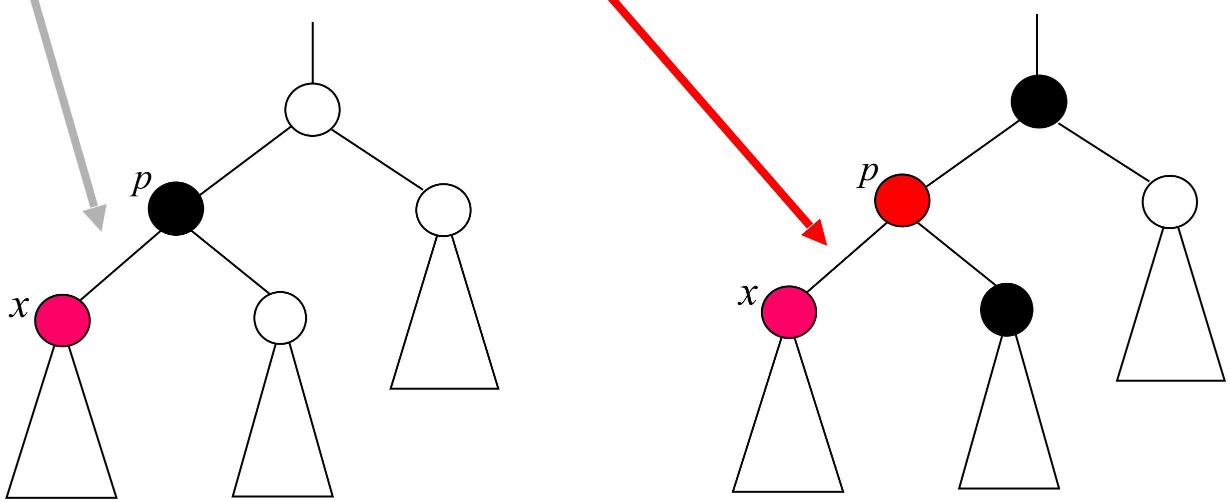
(b) (a)를 레드블랙트리로 만든 예



(c) 실제 구현시의 NIL 노드 처리 방법

레드블랙트리에서의 삽입

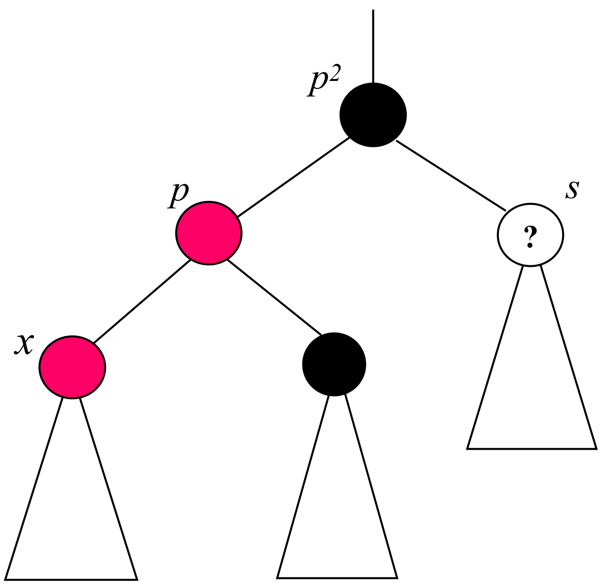
- 이진검색트리에서의 삽입과 같다. 다만 삽입 후 삽입된 노드를 레드로 칠한다. (이 노드를 x라 하자)
- 만일 x의 부모 노드 p의 색상이
  - 블랙이면 아무 문제 없다.
  - 레드이면 레드블랙특성 ③이 깨진다.



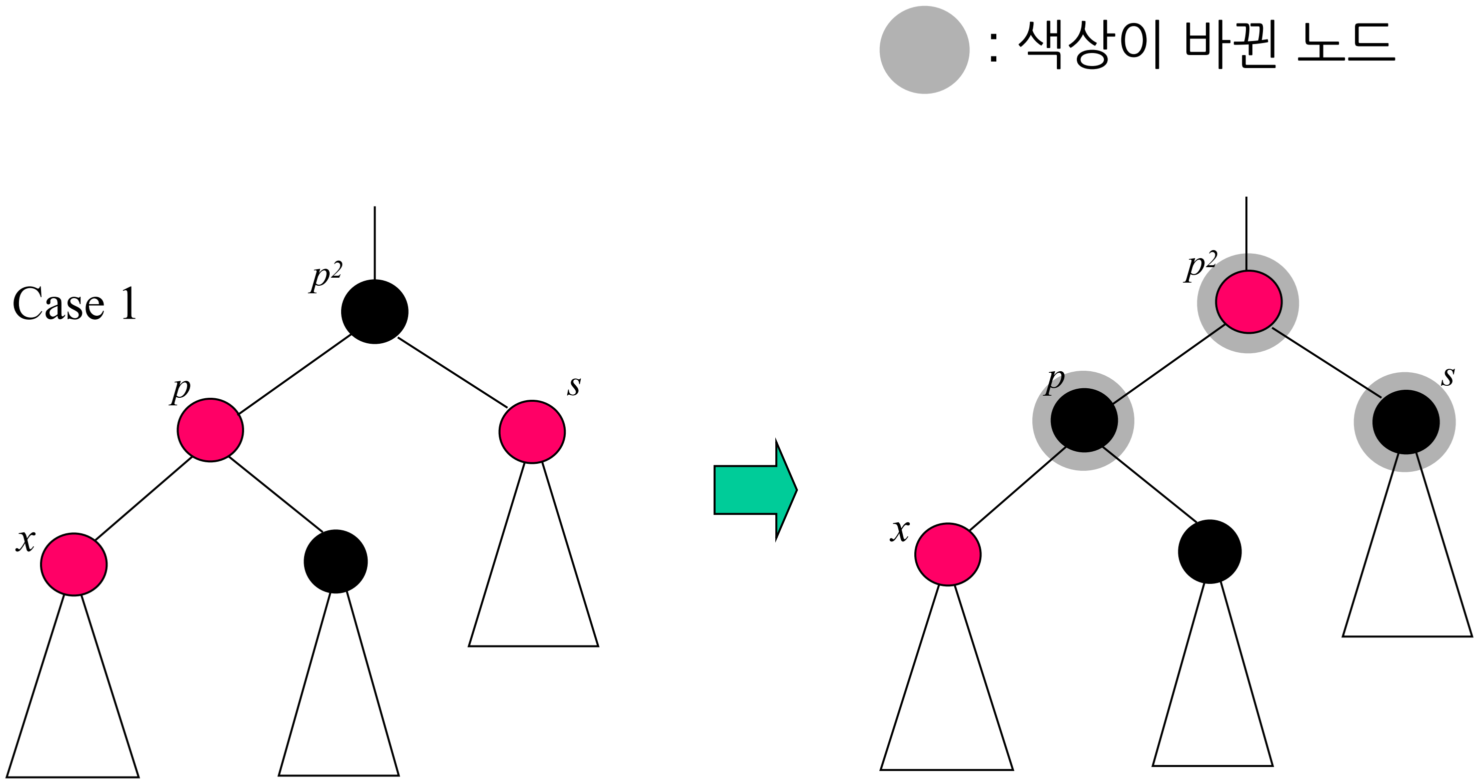
✓ 그러므로 p가 레드인 경우만 고려하면 된다

레드블랙트리에서의 삽입

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

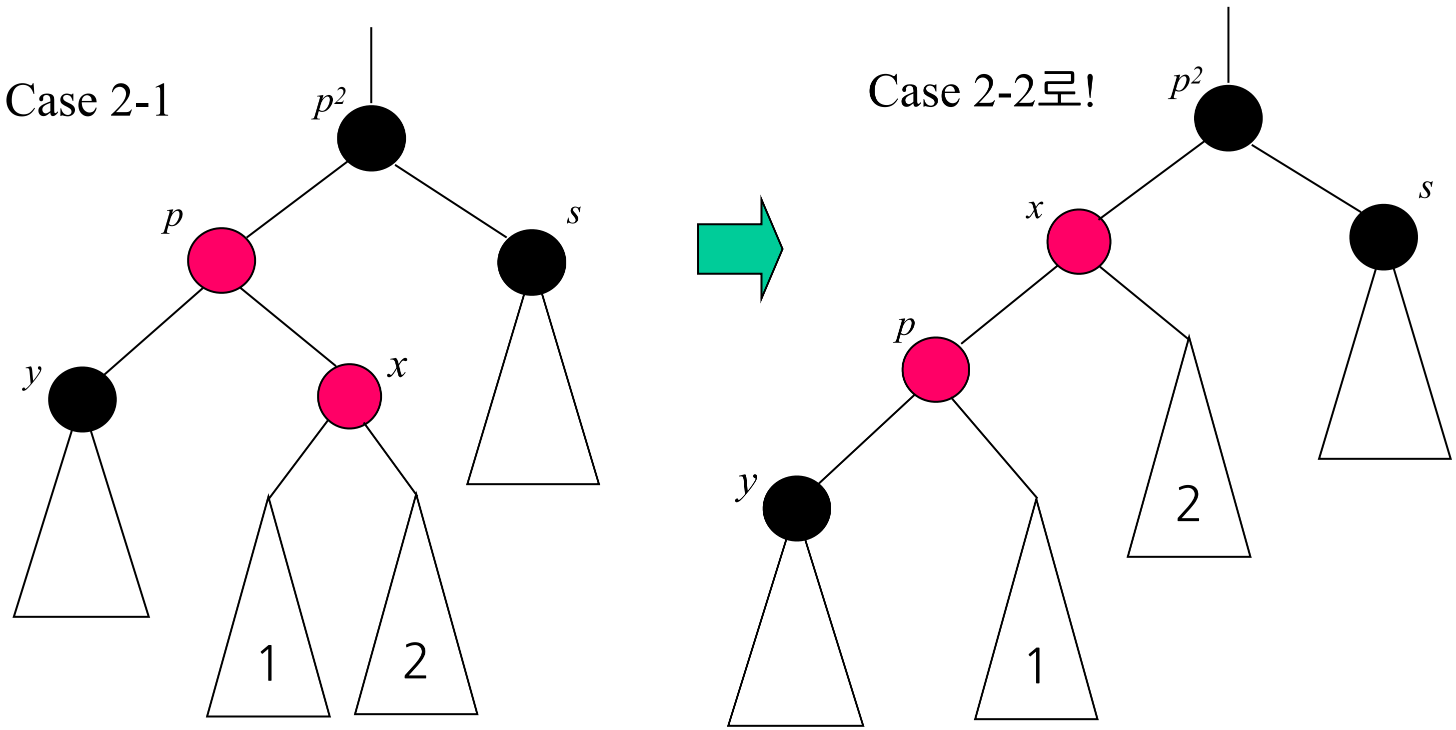


Case 1: s가 레드



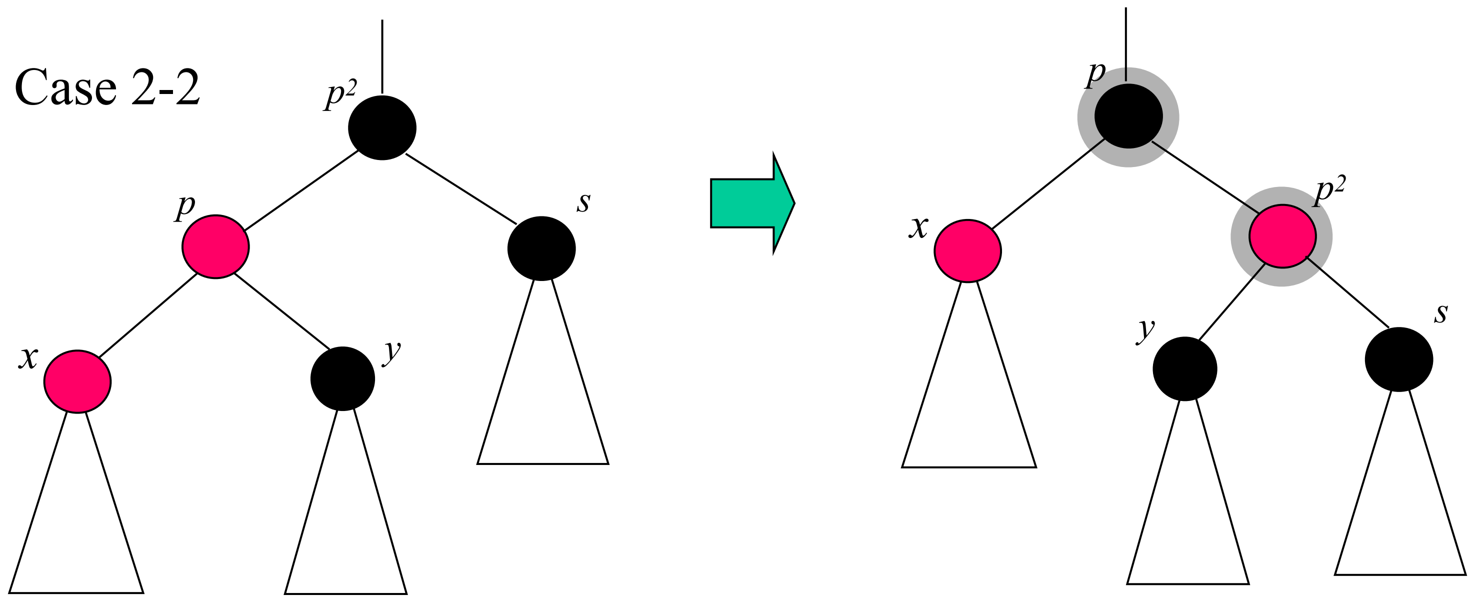
✓ p^2에서 방금과 같은 문제가 발생할 수 있다

Case 2-1: s가 블랙이고, x가 p의 오른쪽 자식



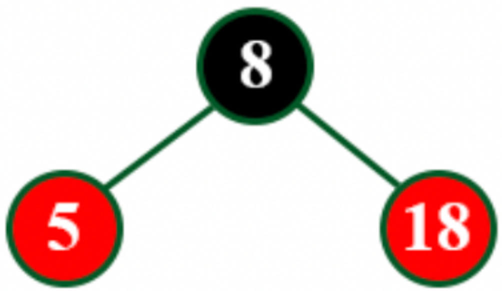
Case 2-2: s가 블랙이고, x가 p의 왼쪽 자식

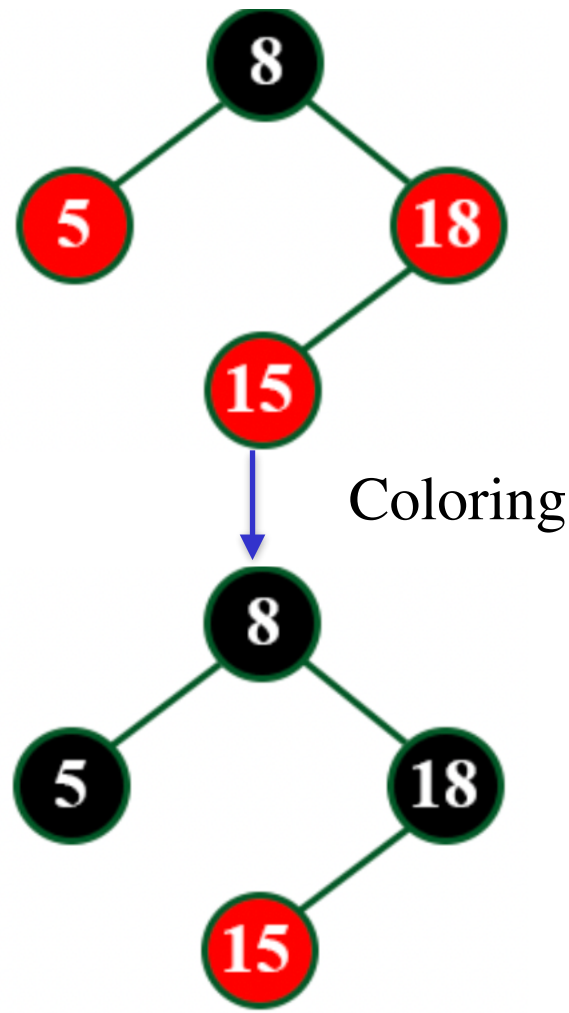
: 색상이 바뀐 노드



✓ 삽입 완료!

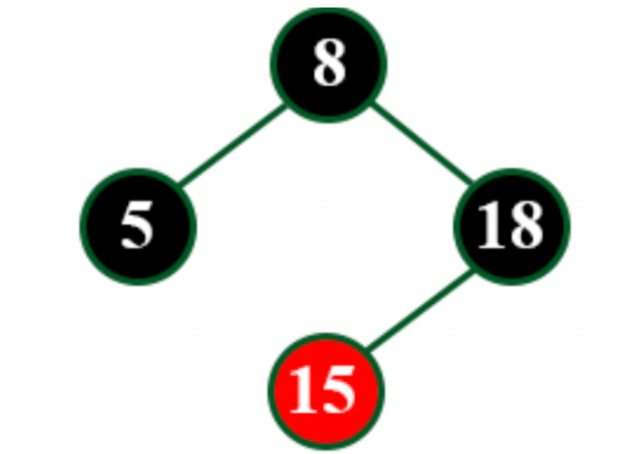
8, 18, 5, 15, 17, 25, 40, 80

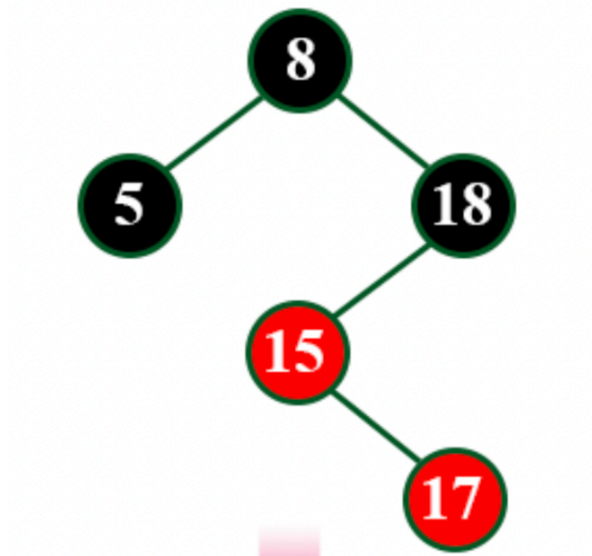




레드블랙트리에서의 삽입

8, 18, 5, 15, 17, 25, 40, 80

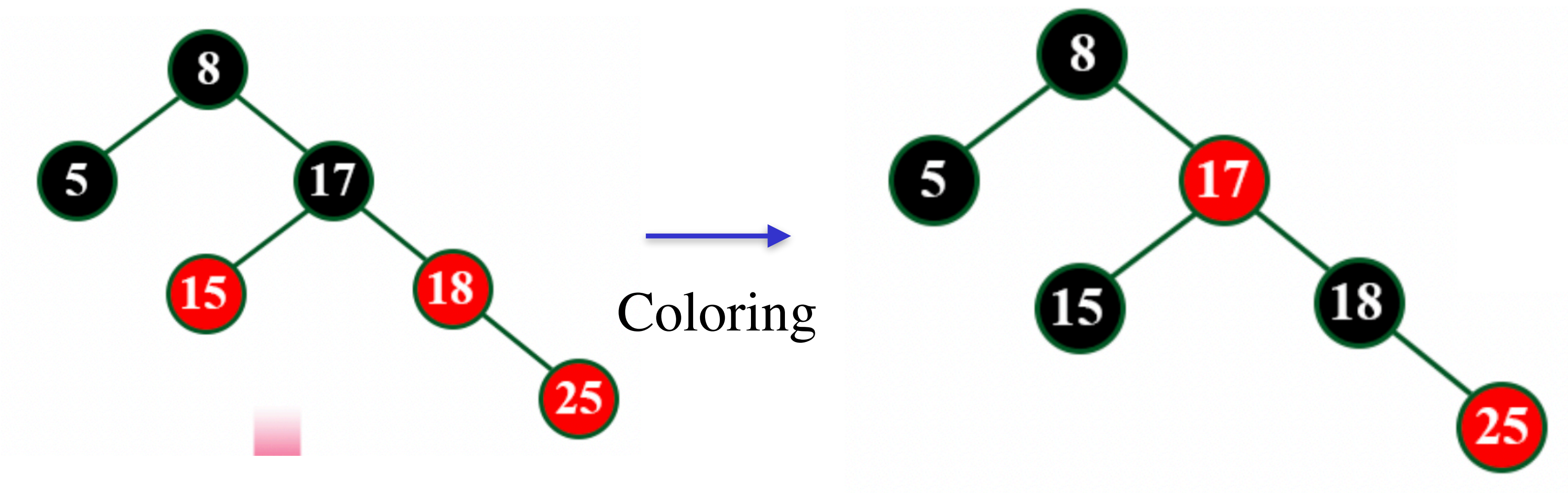




Left Rotating

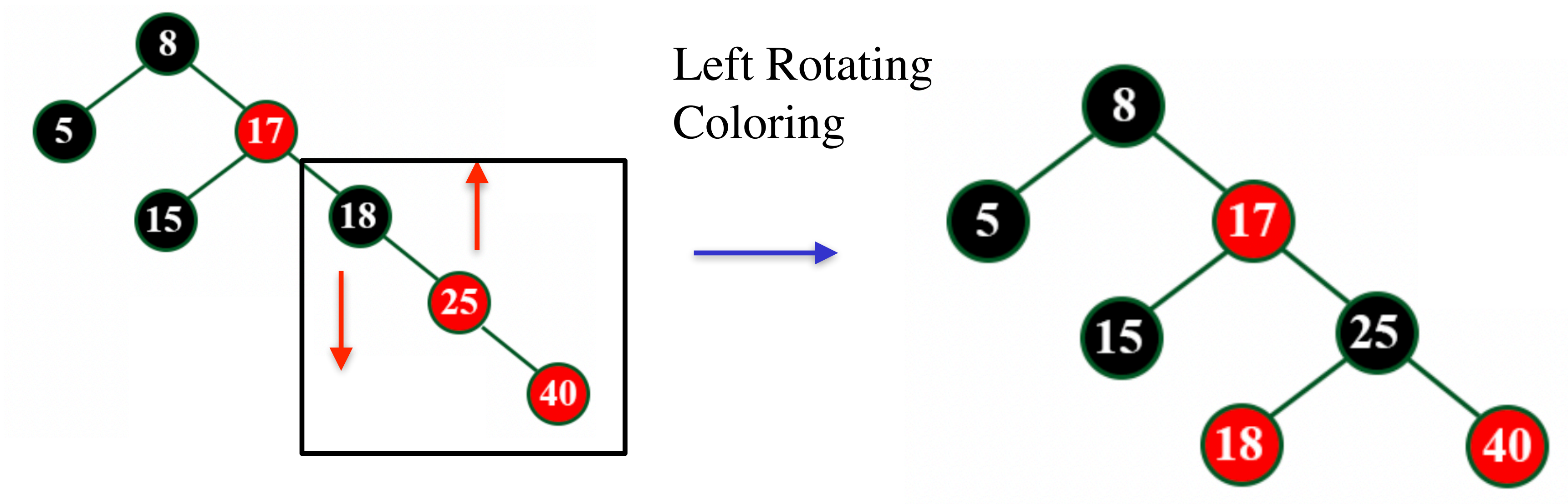
레드블랙트리에서의 삽입

8, 18, 5, 15, 17, 25, 40, 80

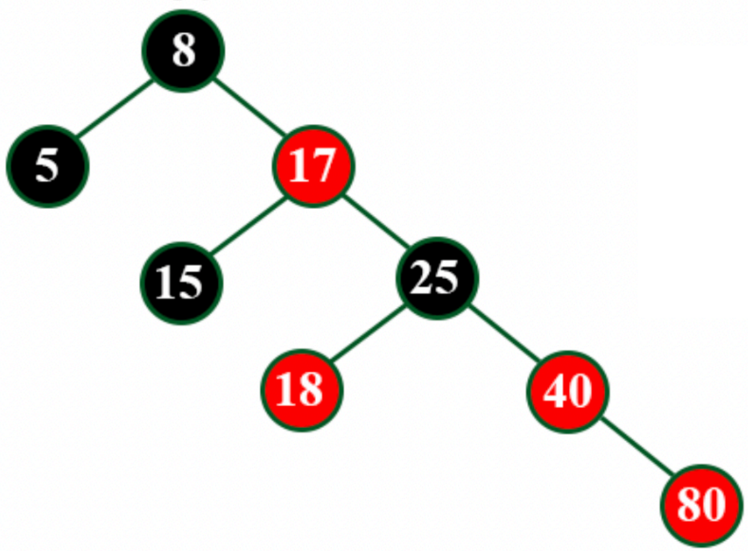


레드블랙트리에서의 삽입

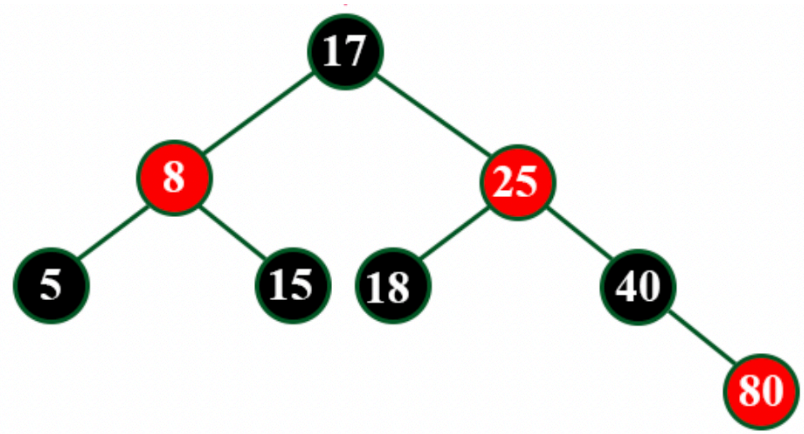
8, 18, 5, 15, 17, 25, 40, 80



8, 18, 5, 15, 17, 25, 40, 80



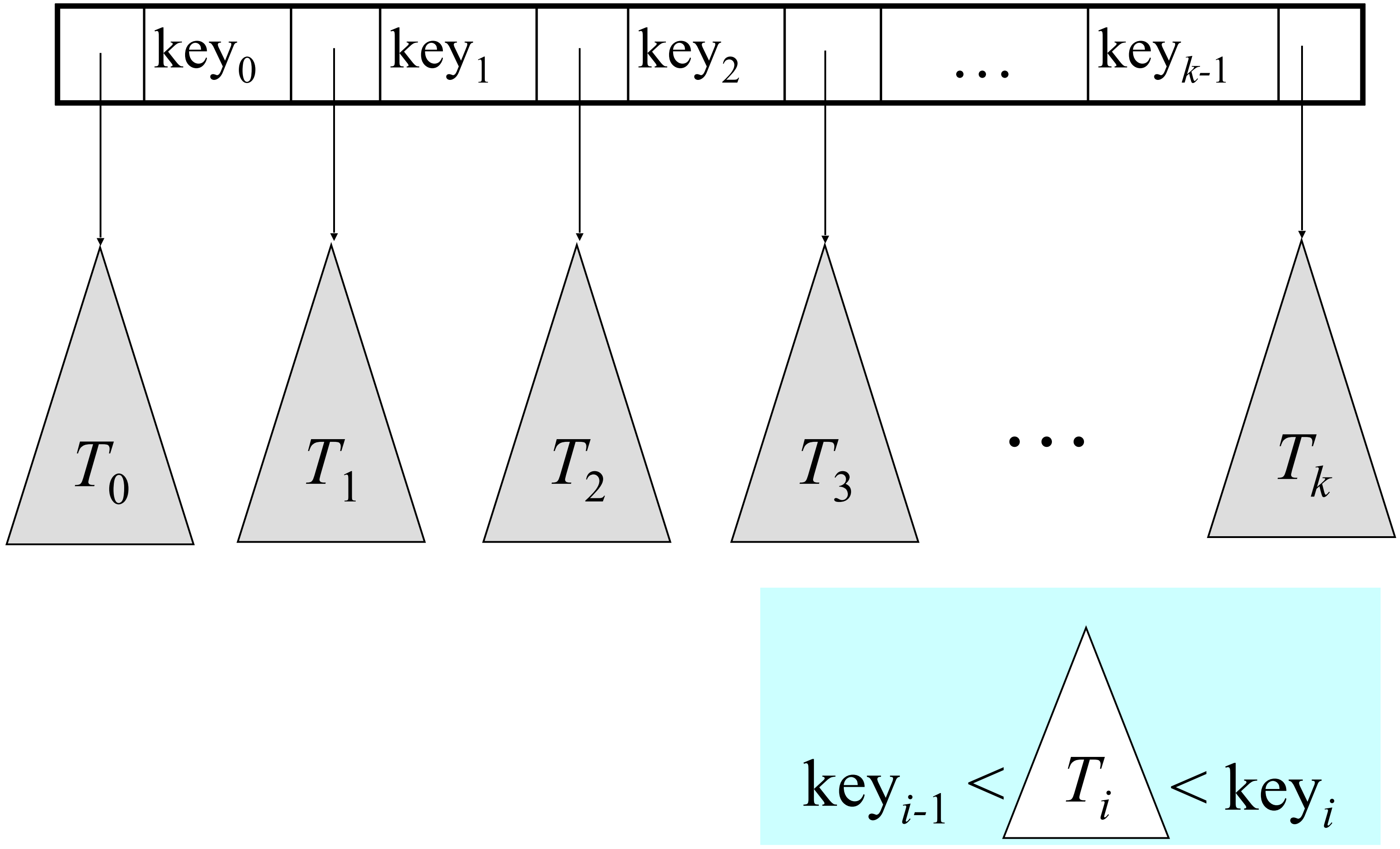
Rotating 과 Coloring ?



B-트리

- 디스크의 접근 단위는 블록(페이지)
  - 블록의 크기는 커지는 추세임
- 디스크에 한 번 접근하는 시간은 수십만 명령어의 처리 시간과 맞먹는다
- 검색트리가 디스크에 저장되어 있다면 트리의 높이를 최소화하는 것이 유리하다
  - 대용량 자료를 모두 메모리에 올려놓을 수는 없음
- B-트리는 다진검색트리가 균형을 유지하도록 하여 최악의 경우 디스크 접근 횟수를 줄인 것이다
  - 한번의 디스크 접근으로 읽어오는 자료의 수를 늘림
  - 분기수가 2를 넘도록 트리를 구성함 (k개의 키를 갖음, k+1의 자식을 가짐)

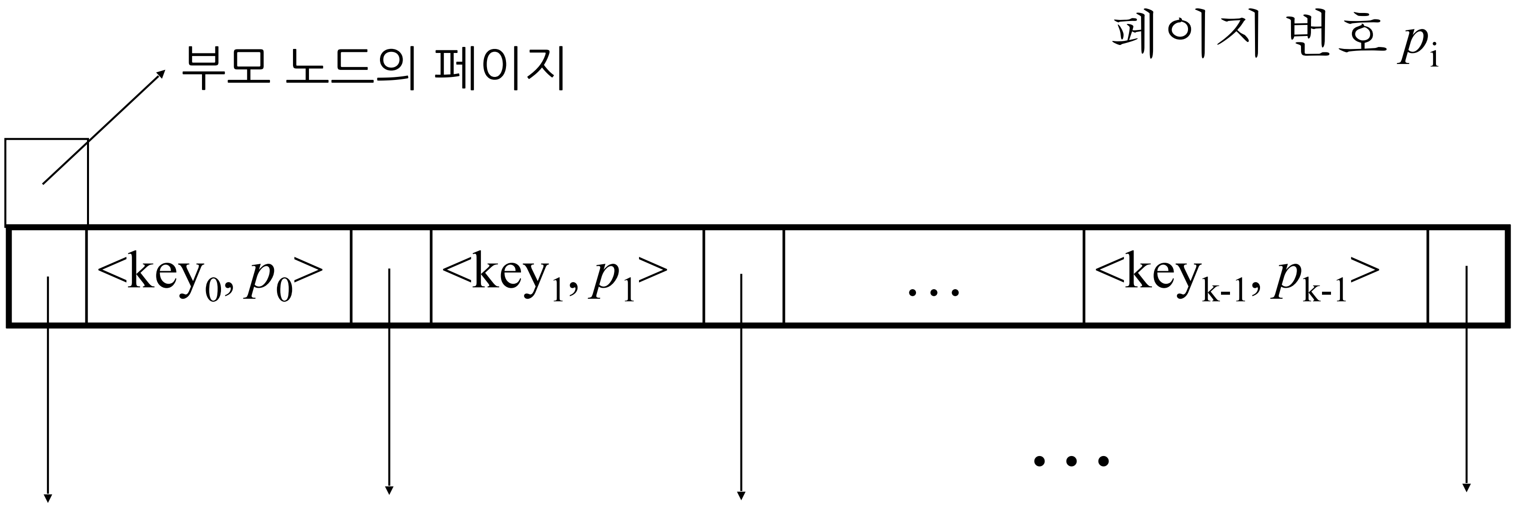
다진검색트리



B-트리

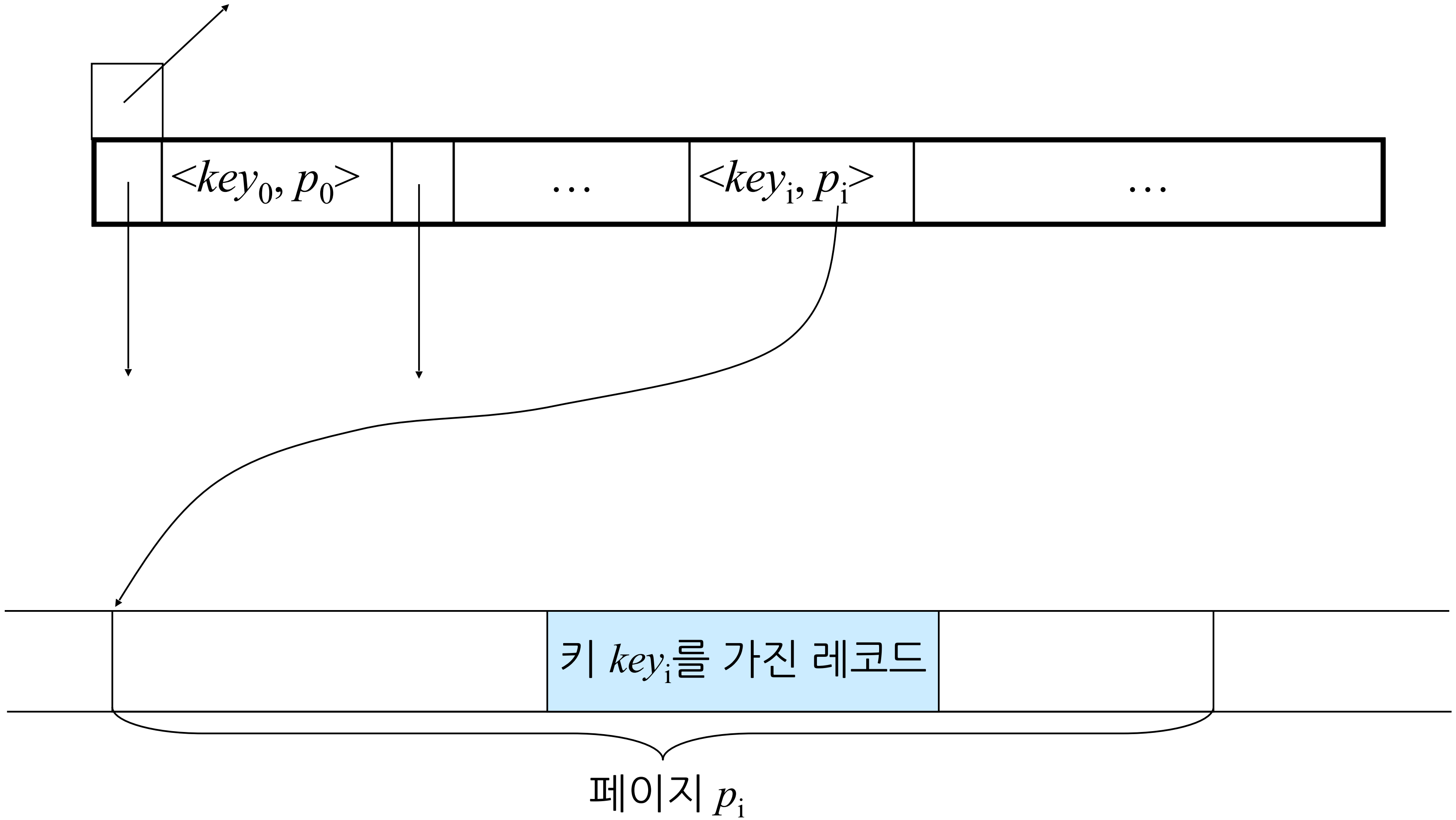
- B-트리는 균형잡힌 다진검색트리로 다음의 성질을 만족한다
  - 루트를 제외한 모든 노드는 $\lfloor k/2 \rfloor \sim k$ 개의 키를 갖는다
  - 모든 리프 노드는 같은 깊이를 가진다

B-트리의 노드 구조



페이지 크기가 8k(8192)이고 키가 16바이트, 페이지번호 4바이트

B-트리를 통해 레코드에 접근하는 과정

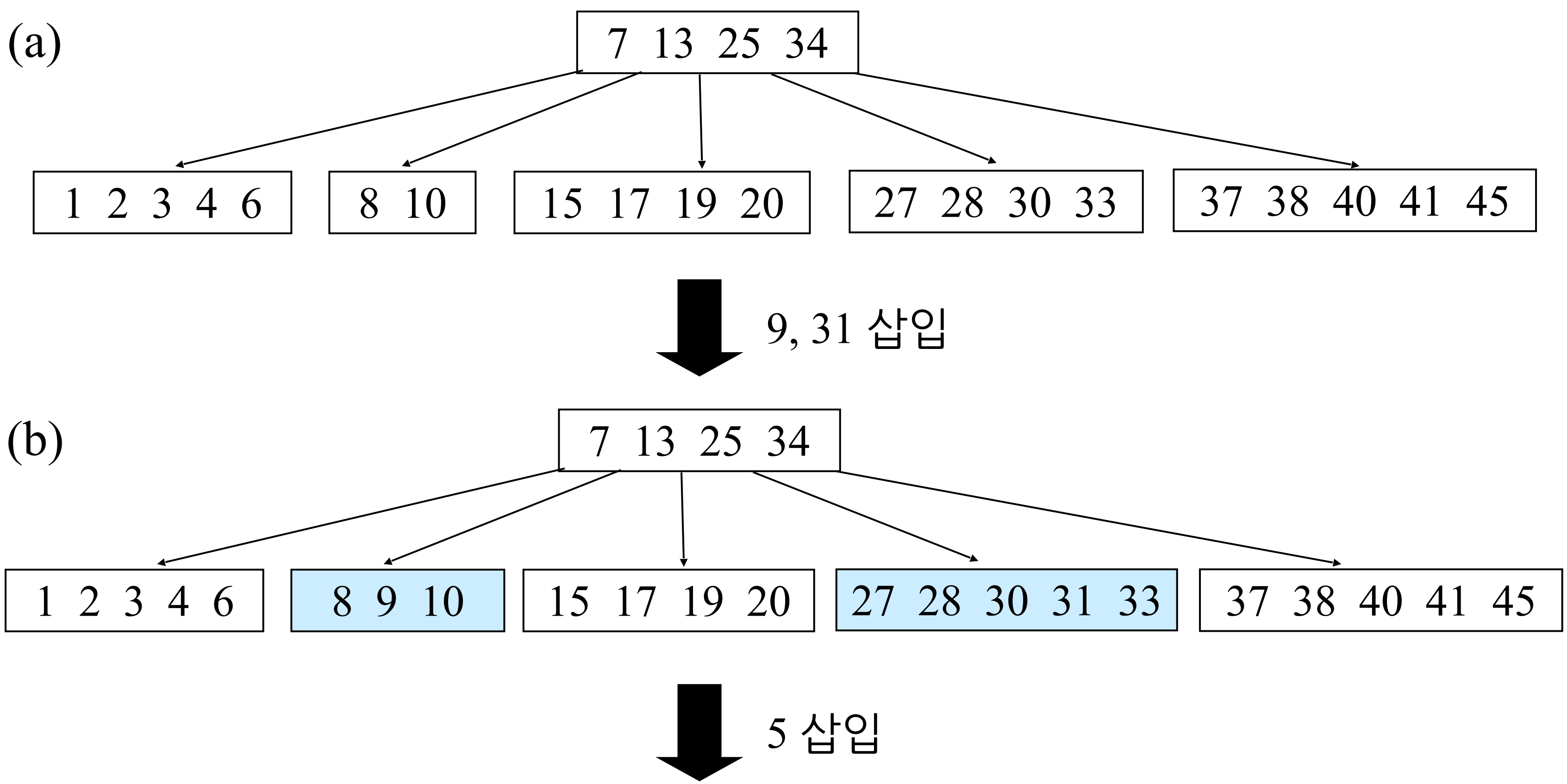


B-트리에서의 삽입

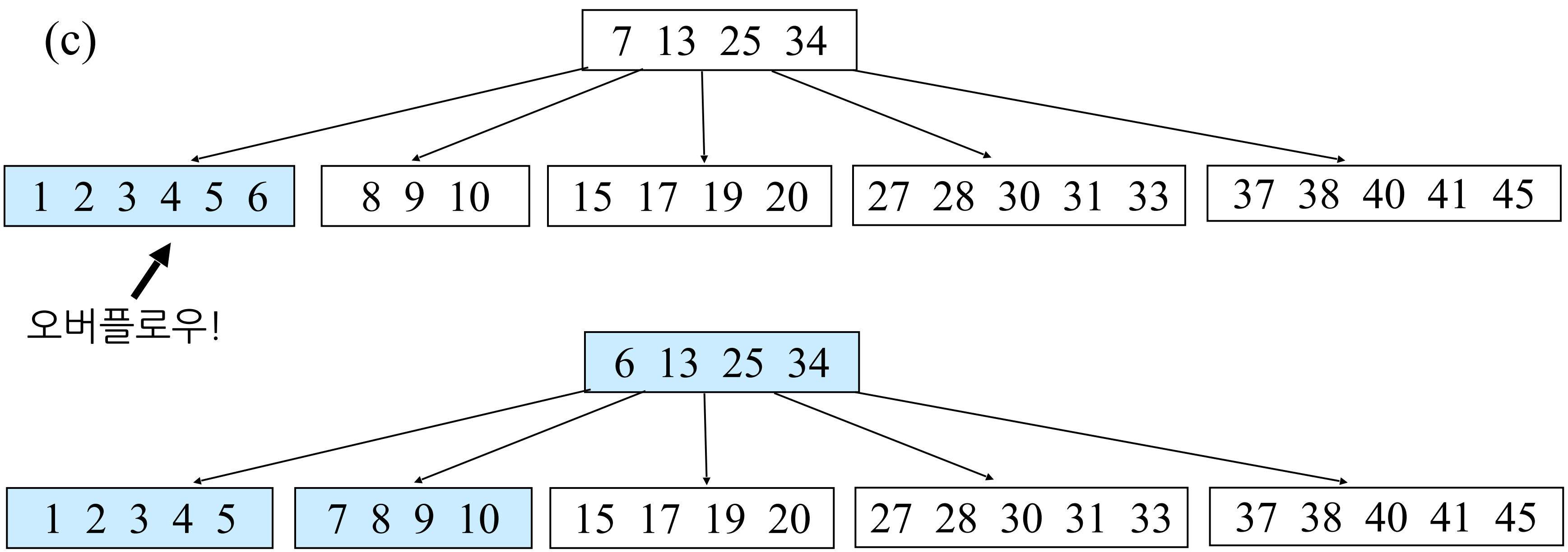
▷ t : 트리의 루트 노드
▷ x : 삽입하고자 하는 키

```
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);
    }
}
```

B-트리에서 삽입의 예 (최대 5개의 키)



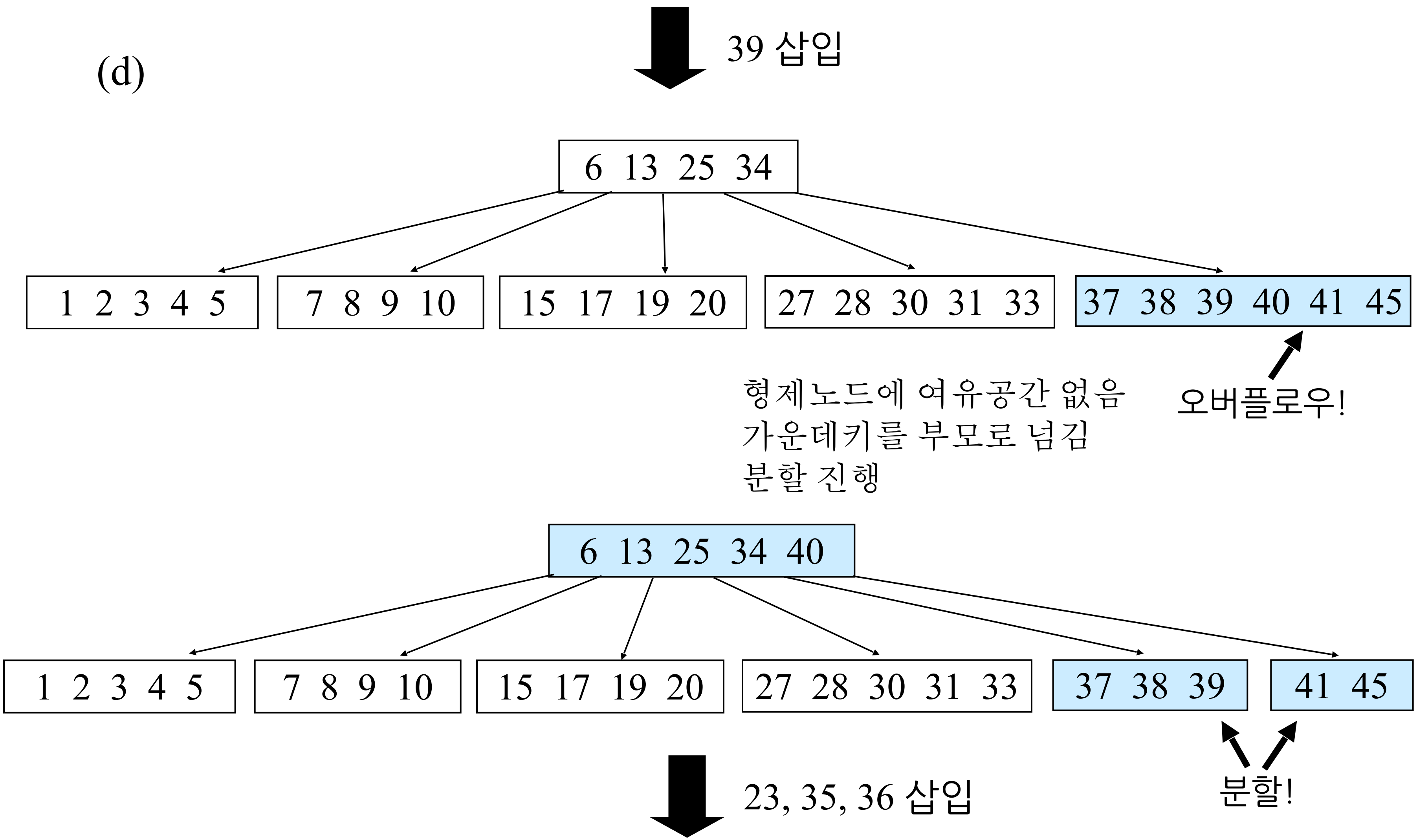
(c)



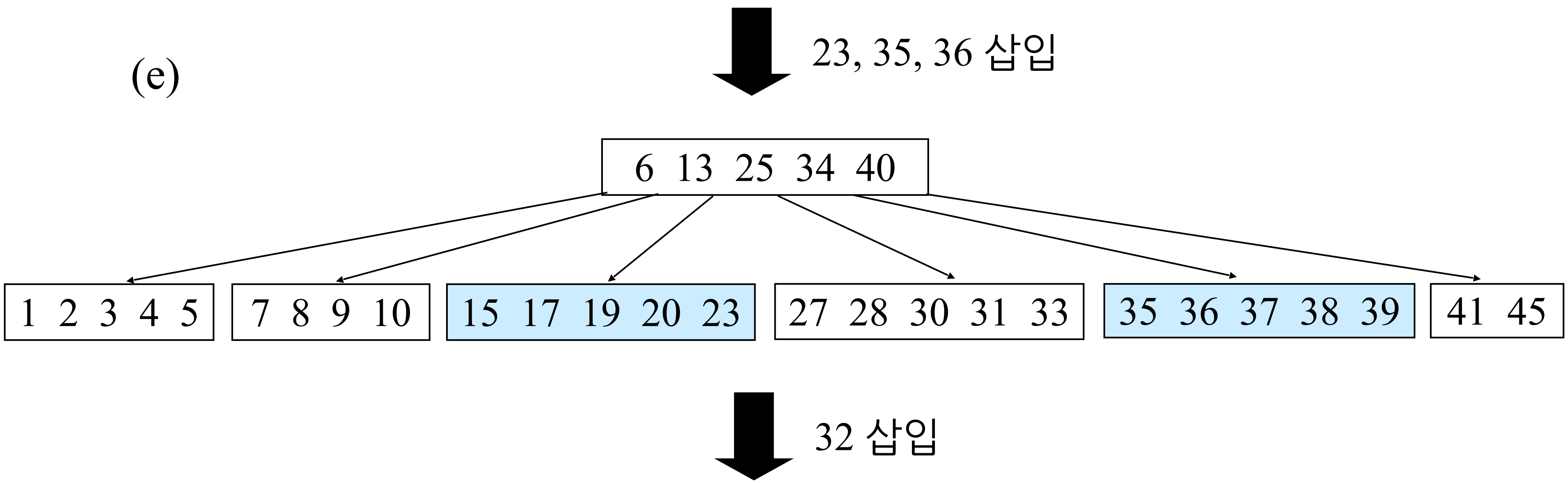
형제노드로 자료를 넘김
검색트리 성질이 유지되어야 함

39 삽입

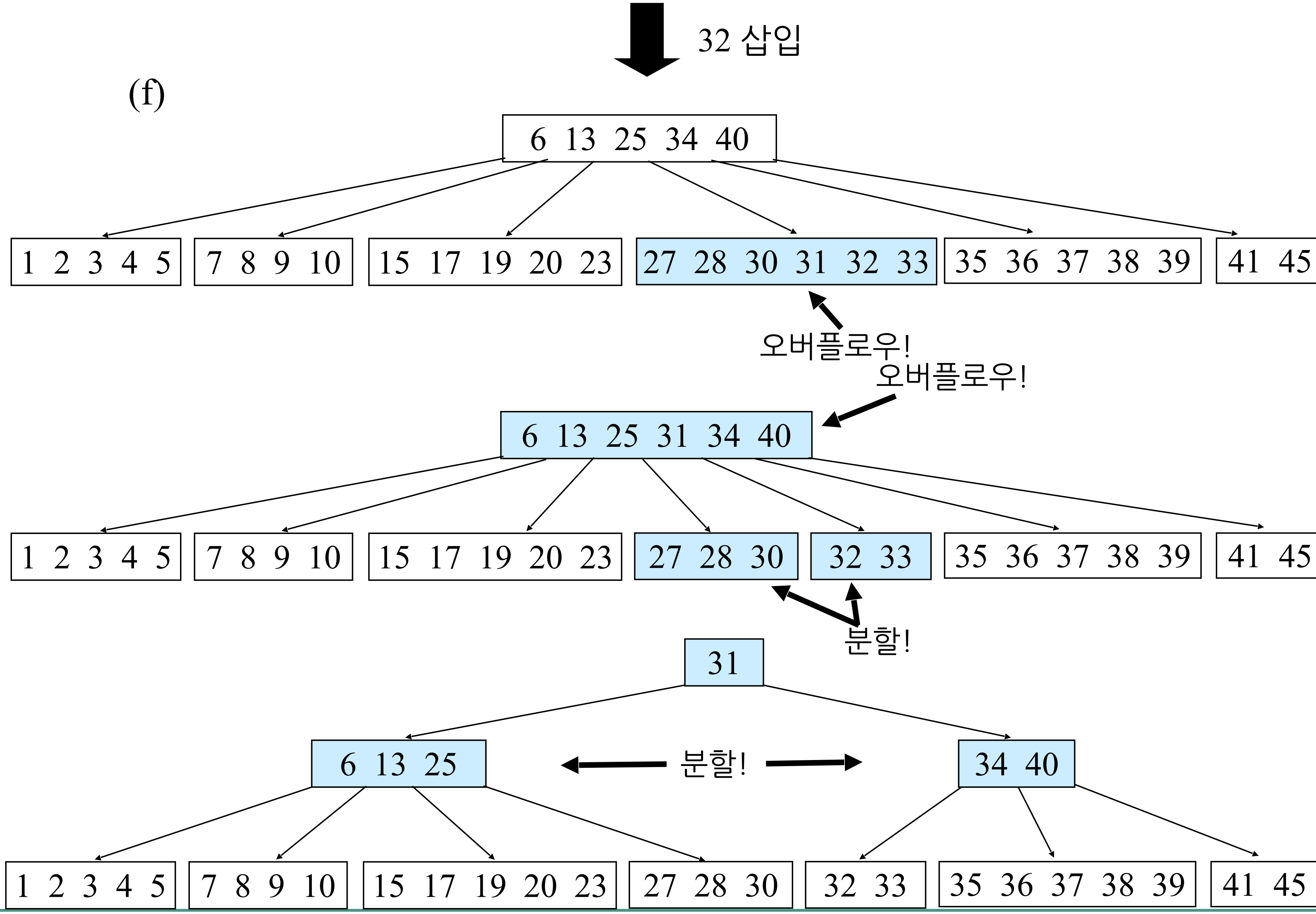
(d)


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
형제노드에 여유공간 없음
가운데키를 부모로 넘김
분할 진행
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
23, 35, 36 삽입
분할!

(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

(f)

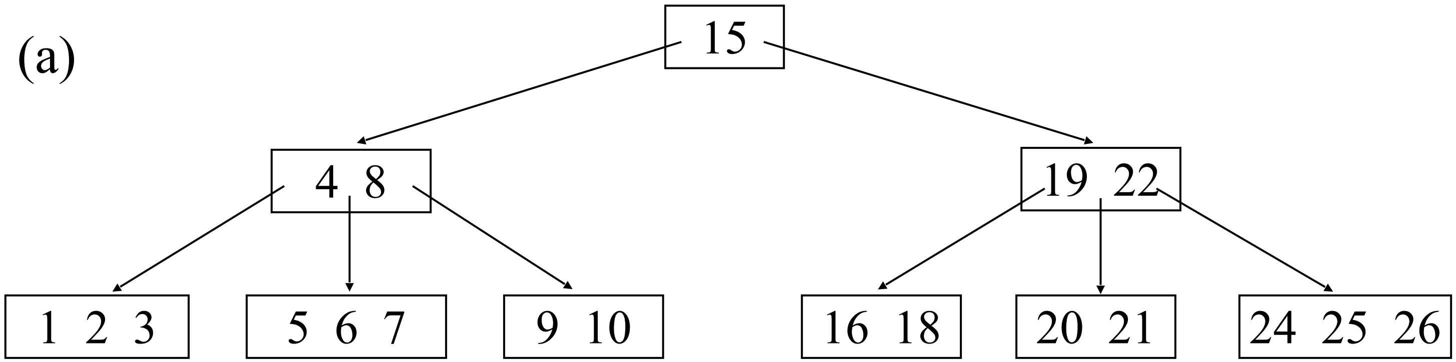

32 삽입
6 13 25 34 40
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30 31 32 33
35 36 37 38 39
41 45
오버플로우!
오버플로우!
6 13 25 31 34 40
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30
32 33
35 36 37 38 39
41 45
분할!
31
6 13 25
분할!
34 40
1 2 3 4 5
7 8 9 10
15 17 19 20 23
27 28 30
32 33
35 36 37 38 39
41 45

B-트리에서의 삭제

```
BTreeDelete(t, x, v)                              ▷ t : 트리의 루트 노드
{                                                 ▷ x : 삭제하고자 하는 키
    if (v가 리프 노드 아님) then {                 ▷ v : x를 갖고 있는 노드
        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);
        }
}
```

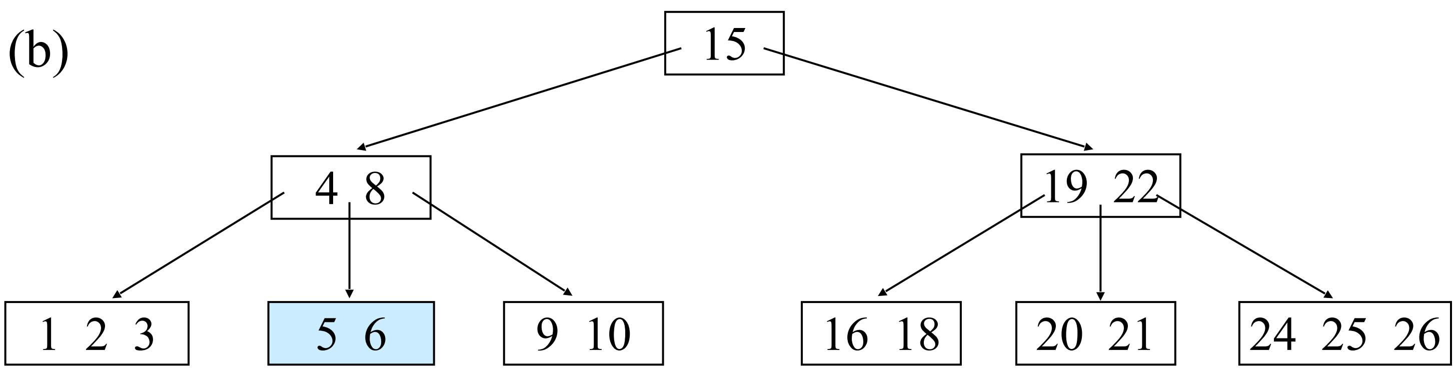
B-트리에서 삭제의 예

(a)



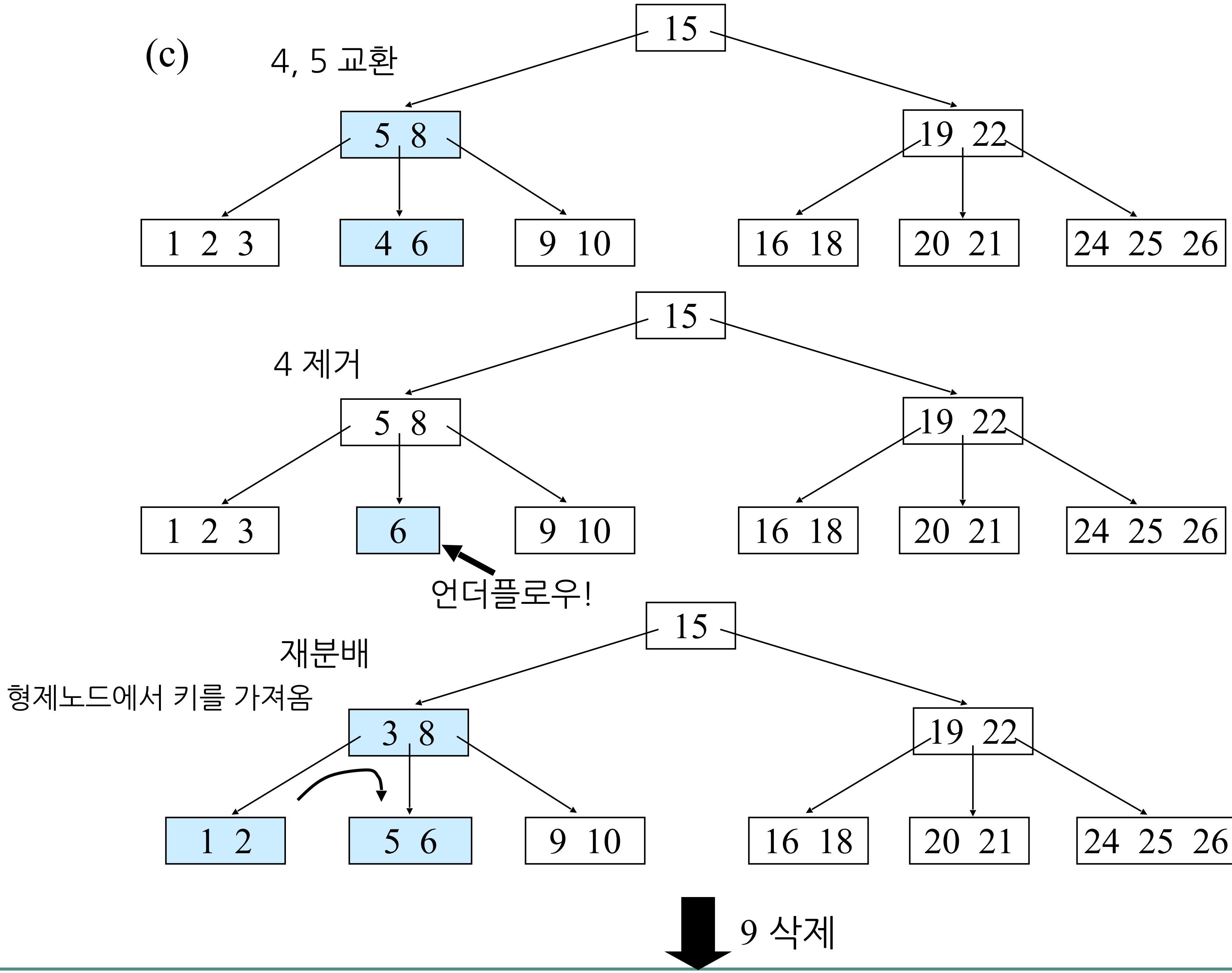
7 삭제

(b)

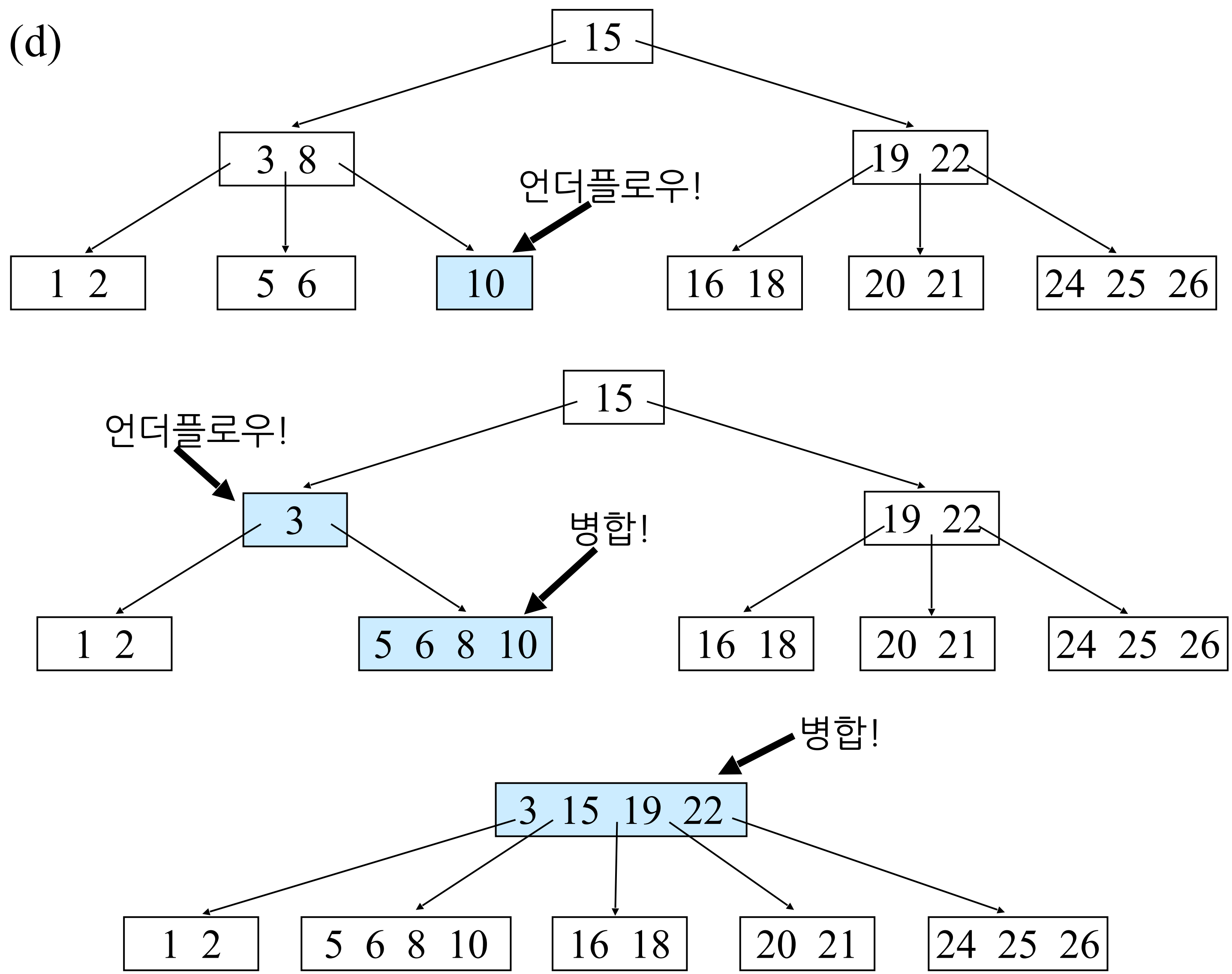


4의 다음 키값 5와 교환!

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
형제노드에서 키를 가져옴
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

Thank you