2021-2
알고리즘

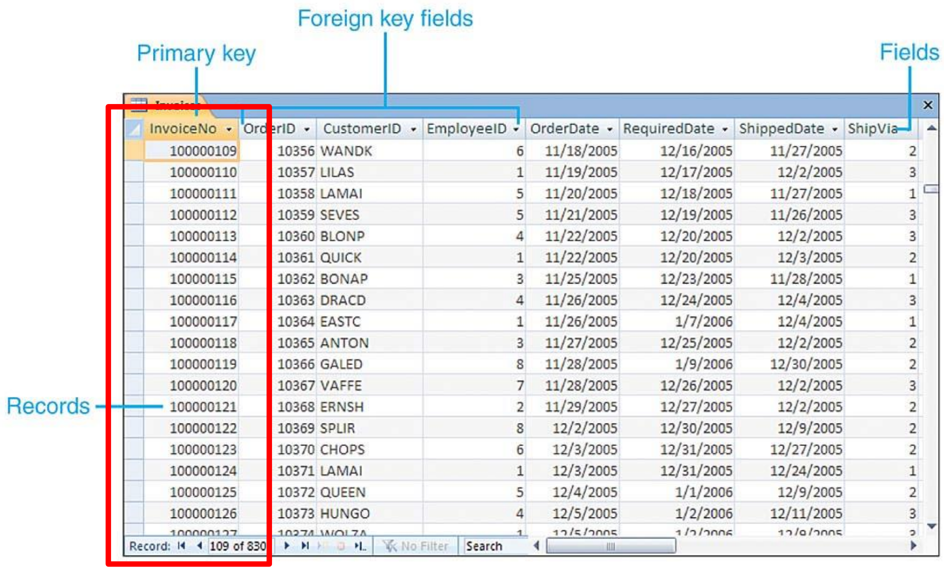
워밍업
이진검색트리
레드블랙트리
B-트리
다차원검색트리

한남대학교 컴퓨터공학과

워밍업

Review: 검색(Searching)

– 입력: primary key, 출력: a reocrd
– **Primary key를 저장, 검색하기 위한 자료 구조?**



InvoiceNo	OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia
100000109	10356	WANDK	6	11/18/2005	12/16/2005	11/27/2005	2
100000110	10357	LILAS	1	11/19/2005	12/17/2005	12/2/2005	3
100000111	10358	LAMAI	5	11/20/2005	12/18/2005	11/27/2005	1
100000112	10359	SEVES	5	11/21/2005	12/19/2005	11/26/2005	3
100000113	10360	BLONP	4	11/22/2005	12/20/2005	12/2/2005	3
100000114	10361	QUICK	1	11/22/2005	12/20/2005	12/3/2005	2
100000115	10362	BONAP	3	11/25/2005	12/23/2005	11/28/2005	1
100000116	10363	DRACD	4	11/26/2005	12/24/2005	12/4/2005	3
100000117	10364	EASTC	1	11/26/2005	1/7/2006	12/4/2005	1
100000118	10365	ANTON	3	11/27/2005	12/25/2005	12/2/2005	2
100000119	10366	GALED	8	11/28/2005	1/9/2006	12/30/2005	2
100000120	10367	VAFFE	7	11/28/2005	12/26/2005	12/2/2005	3
100000121	10368	ERNSH	2	11/29/2005	12/27/2005	12/2/2005	2
100000122	10369	SPLIR	8	12/2/2005	12/30/2005	12/9/2005	2
100000123	10370	CHOPS	6	12/3/2005	12/31/2005	12/27/2005	2
100000124	10371	LAMAI	1	12/3/2005	12/31/2005	12/24/2005	1
100000125	10372	QUEEN	5	12/4/2005	1/1/2006	12/9/2005	2
100000126	10373	HUNGO	4	12/5/2005	1/2/2006	12/11/2005	3

- **case 2: sorted array → binary search**
  – Search/Insert: avg. $O(\log n)$

0215	0320	0609	■ ■ ■	0714

bisect Module in Python

- **이진 탐색을 활용한 삽입 알고리즘**
  - 가정: 리스트가 정렬되어 있음
  - **bisect(a, x)** : x를 a에 삽입할 인덱스를 리턴
  - **insort(a, x)** : x를 a의 적절한 위치에 삽입

```python
def bisect_right(a, x, lo=0, hi=None):
    """Return the index where to insert item x in list a, assuming a

    if lo < 0:
        raise ValueError('lo must be non-negative')
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        # Use __lt__ to match the logic in list.sort() and in heapq
        if x < a[mid]: hi = mid
        else: lo = mid+1
    return lo
```

bisect Module in Python

```python
from bisect import bisect, insort

data = [215, 320, 609, 714]

index = bisect(data, 400);     print(index)
data.insert(index, 400);       print(data)

insort(data, 700);             print(data)
```

```
2
[215, 320, 400, 609, 714]
[215, 320, 400, 609, 700, 714]
```

연습문제

- SortedList 클래스를 작성해 보자.
  - 이 클래스는 리스트가 정렬된 상태를 유지하며,
  - 이진탐색을 활용해서 검색, 삽입, 삭제가 이루어진다.

이진검색트리

9~10장 구성

- Searching 검색 문제

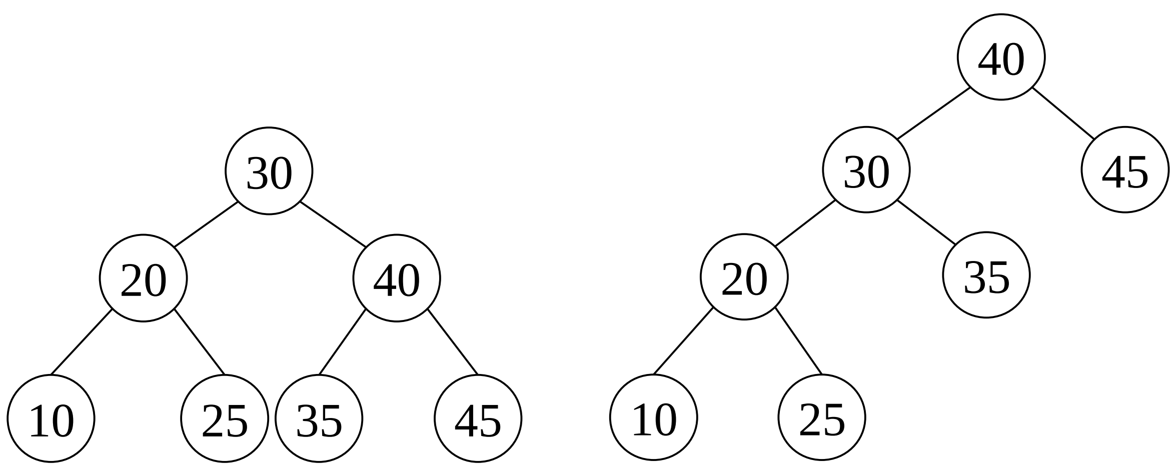
- Map
- 해시테이블 Hash Table
- 충돌 Collision
- 충돌 해결 Collision Resolution

- **Search Tree**
  - (Internal) Binary Search Tree
  - (Internal, Balanced) Red-Black Tree
  - (External) B-tree
  - (Multi-Dimentional) KD-Tree, etc.

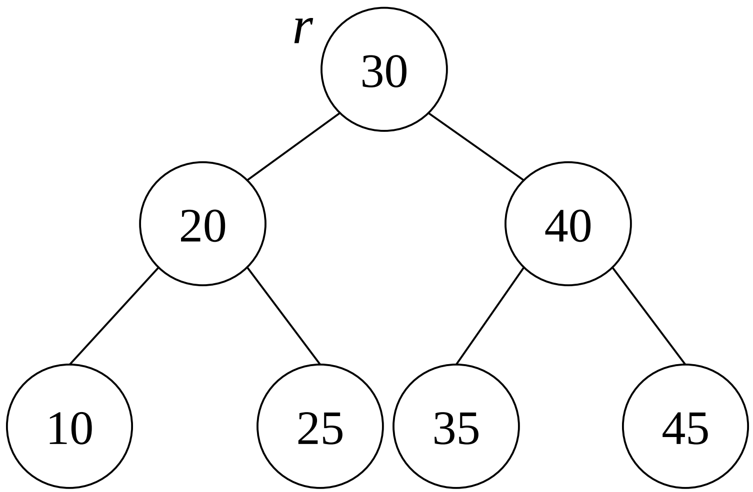
이진검색트리

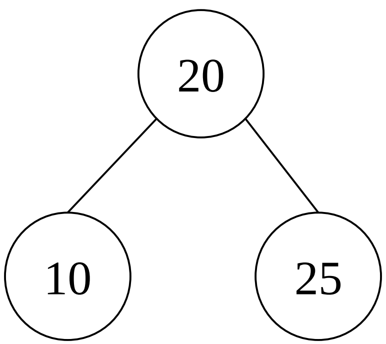
- **이진검색트리 (Binary Search Tree, BST)**
  - 각 노드는 고유한 키값을 하나씩 갖는다.
  - 최상위 레벨에 루트 노드가 있고, 각 노드는 **최대 두 개의 자식**을 갖는다.

- **임의의 노드의 키값은**
  – 자신의 왼쪽 자식 노드의 키값보다 크고, 오른쪽 자식의 키값보다 작다.
  – 같은 원소들이라도 BST의 모양은 다를 수 있다.

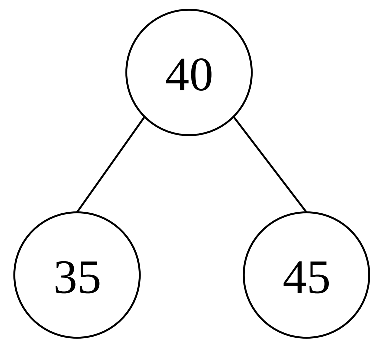


서브트리의 예



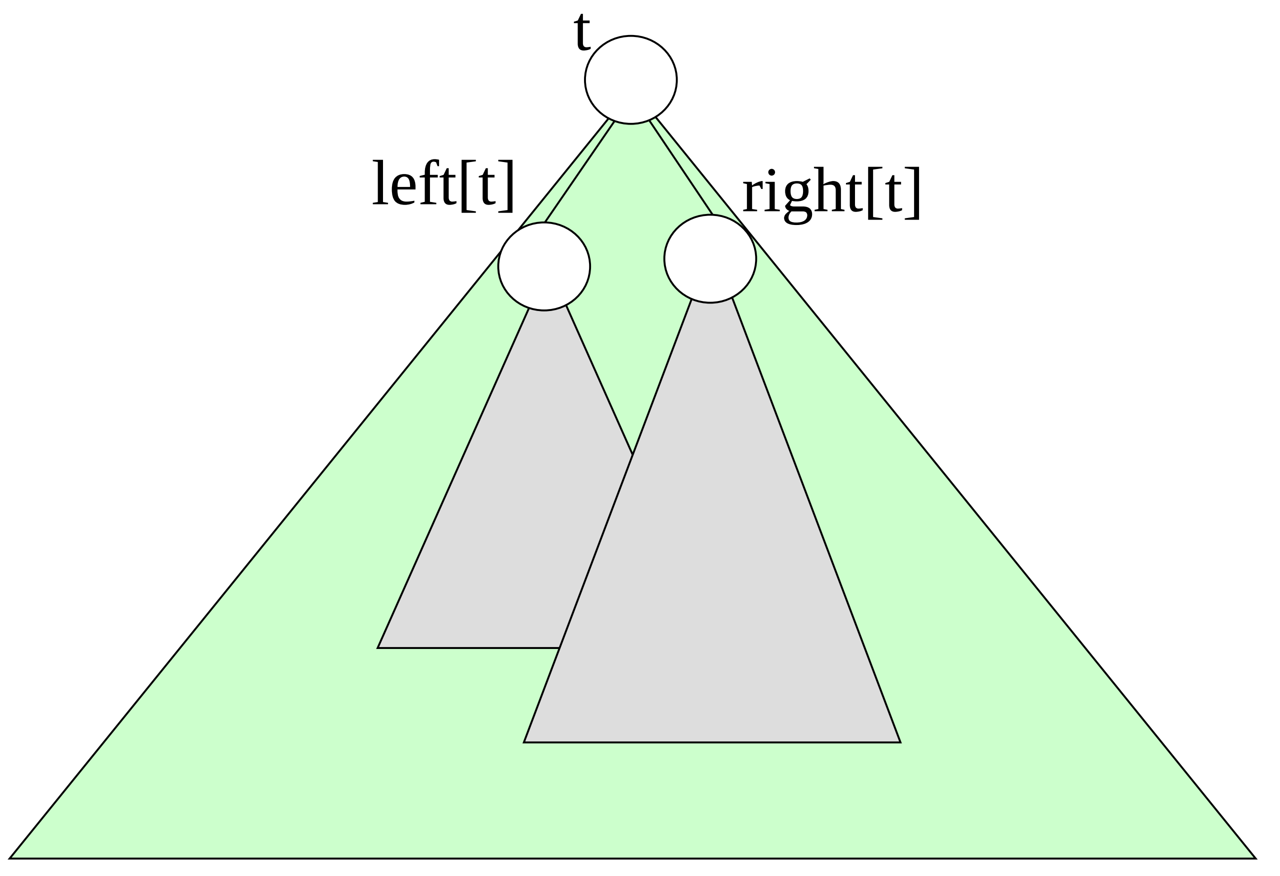


(b) 노드 r의 왼쪽 서브트리

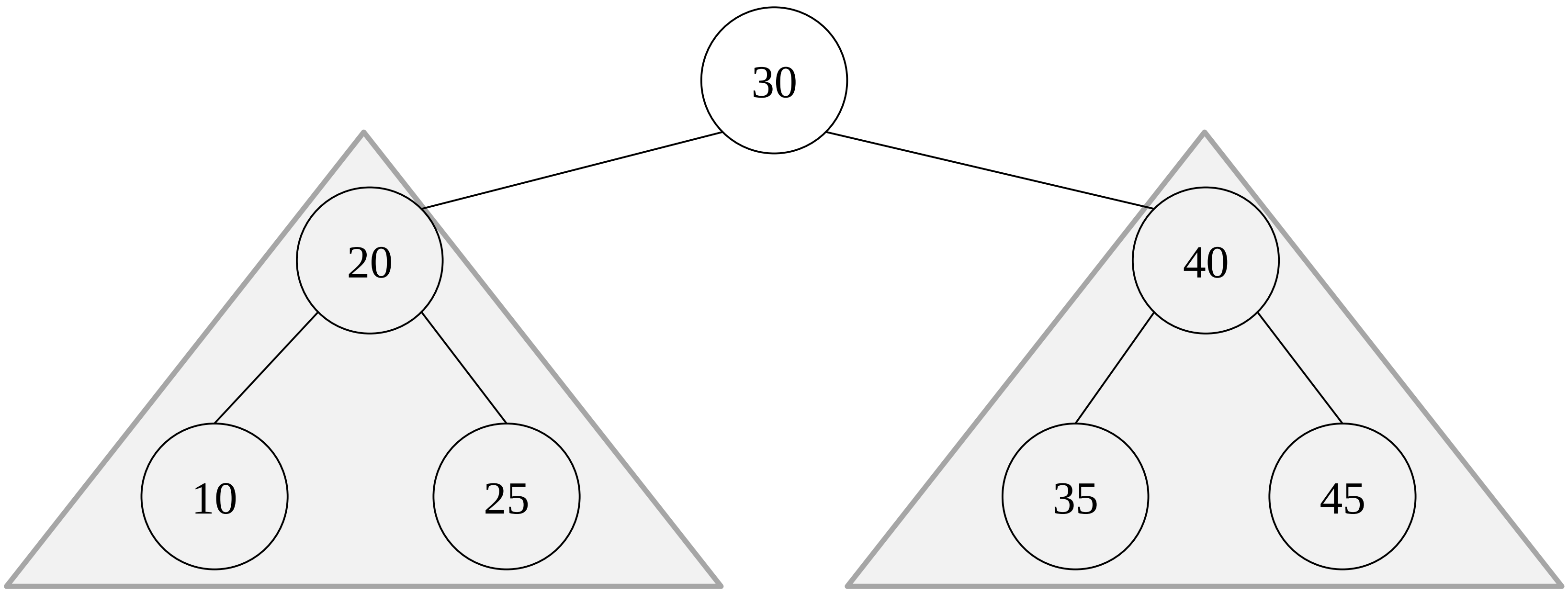


(c) 노드 r의 오른쪽 서브트리

재귀적 관점에서 바라본 이진검색트리

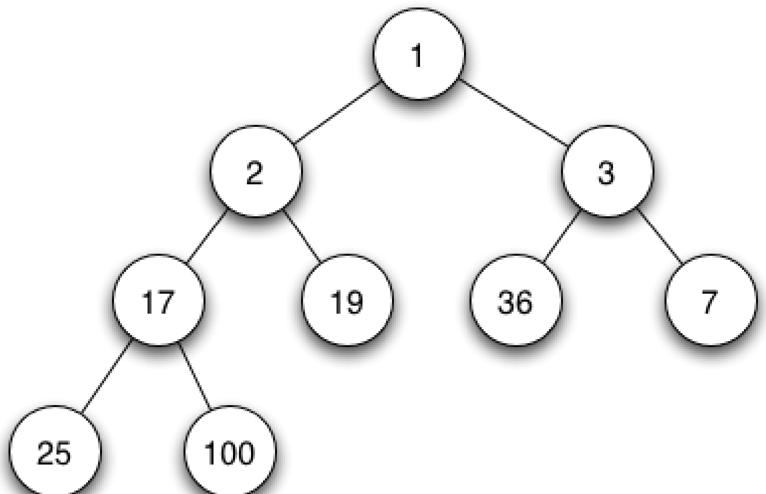


재귀적 관점에서 바라본 이진검색트리



이진검색트리 표현하기

- **1) 배열**
  - A[1..n]에서 **A[i]**의 왼쪽 자식은 **A[i*2]**, 오른쪽 자식은 **A[i*2+1]**이다.
  - BST는 heap과 달리 완전 이진 트리가 아니므로 빈 공간이 생길 수 있음



1	2	3	4	5	6	7	8	9
1	2	3	17	19	36	7	25	100

lv.0 lv.1 lv.2 lv.3

이진검색트리 표현하기

- **2) 구조체와 포인터**
  - 노드와 트리를 구분할 수도 있고,
  - 노드를 따로 구분하지 않고 tree와 subtree들로 다룰 수도 있음

```c
struct bst_t {
    int key;
    struct bst_t* left;
    struct bst_t* right;
};

typedef struct bst_t BST;
BST* root = NULL;
```

```c
BST* bst_alloc(int key) {
    BST* new_node =
        (BST*)malloc(sizeof(BST));
    new_node->key = key;
    return new_node;
}

int main()
{
    // 30을 루트 노드로 삽입
    root = bst_alloc(30);
    return 0;
}
```

이진검색트리 표현하기

- **3) 클래스**
  - 트리가 비어 있는 상태를 표현하려면 node와 tree를 구분하는 게 편함

```
class BST:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


root = None
```

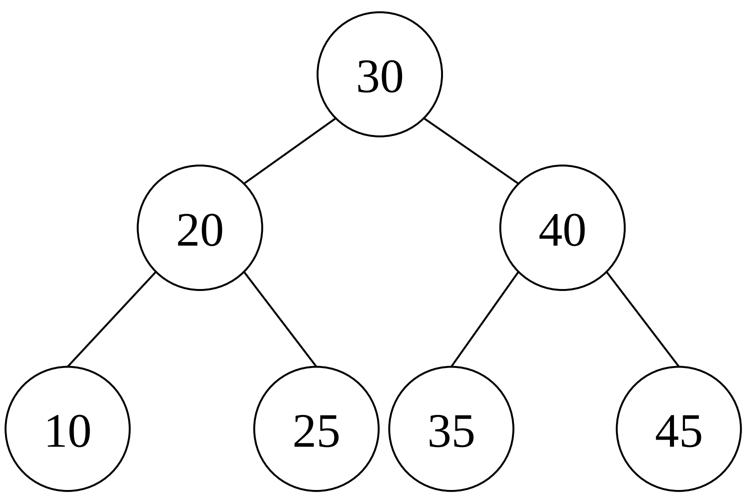
```
class BSTNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


class BST:
    def __init__(self):
        self.root = None


bst = BST()
```

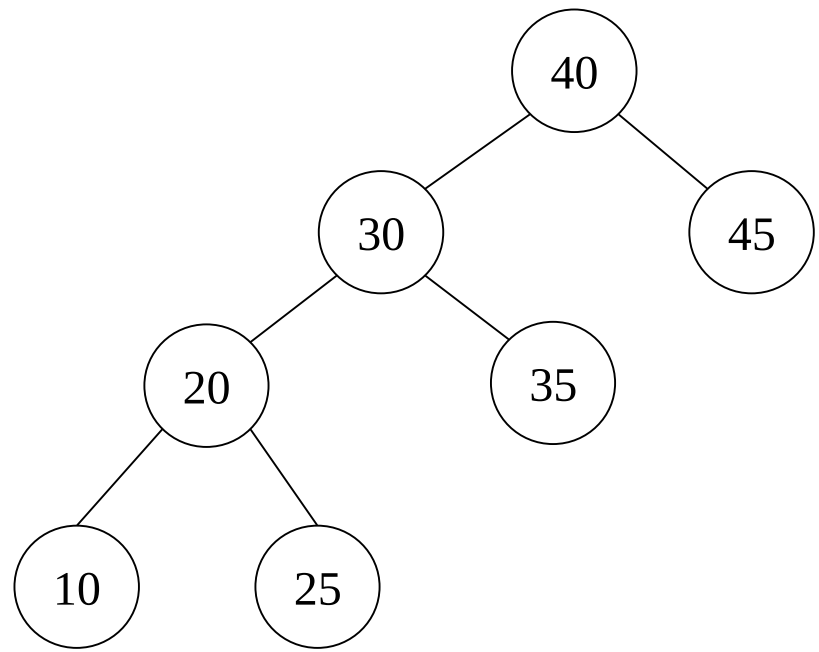
이진검색트리에서
검색, 삽입

이진검색트리에서 검색

- **아래 상태에서**
  - 35를 검색한 경우?
  - 44를 검색한 경우?

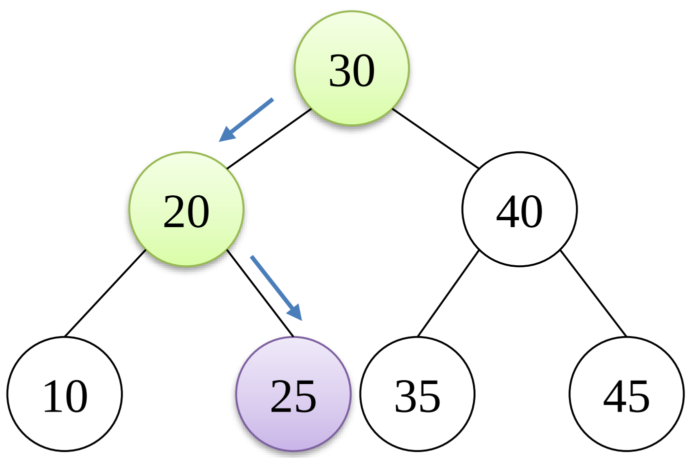
BST 특징:
왼쪽 자식은 부모보다 작고,
오른쪽 자식은 부모보다 크다.



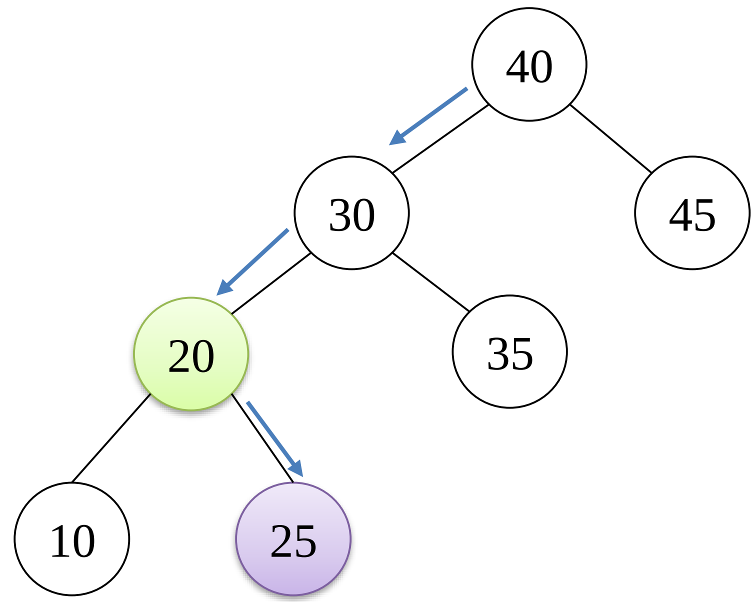


이진검색트리에서 검색

- **Search(25) – 성공한 검색**
  - 같은 원소들이라도 여러 가지 BST가 존재함



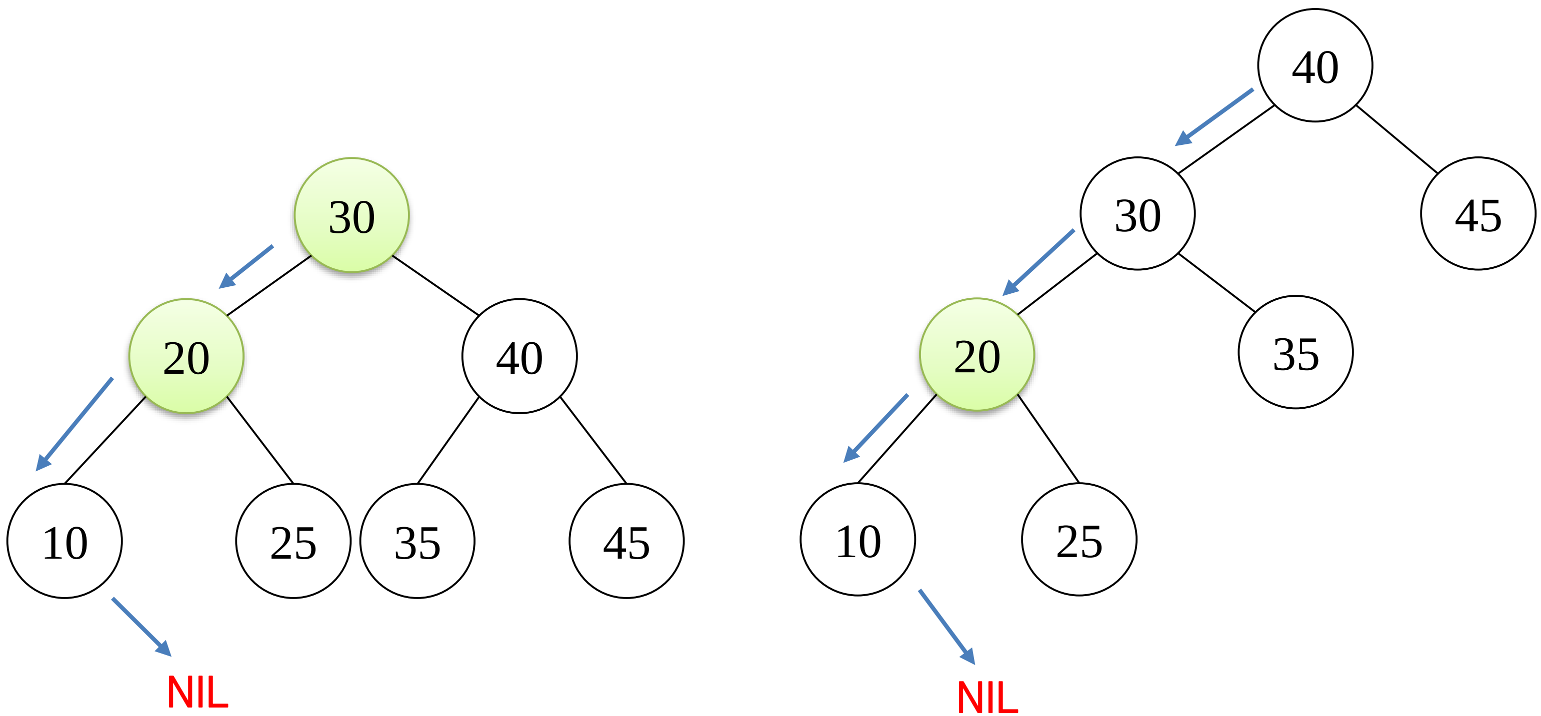
비교 횟수: 3



비교 횟수: 4

이진검색트리에서 검색

- Search(**15**) – 실패한 검색



이진검색트리에서 검색

```
bst_search(t, x) {
       트리 t가 비어 있으면 → 못 찾음; return False;

       트리 t에서 루트 노드의 키를 k라고 하면:
              k == x → 찾음; return t;
              x <  k → 왼쪽 자식 트리에서 찾는다.
              k <  x → 오른쪽 자식 트리에서 찾는다.
}
```

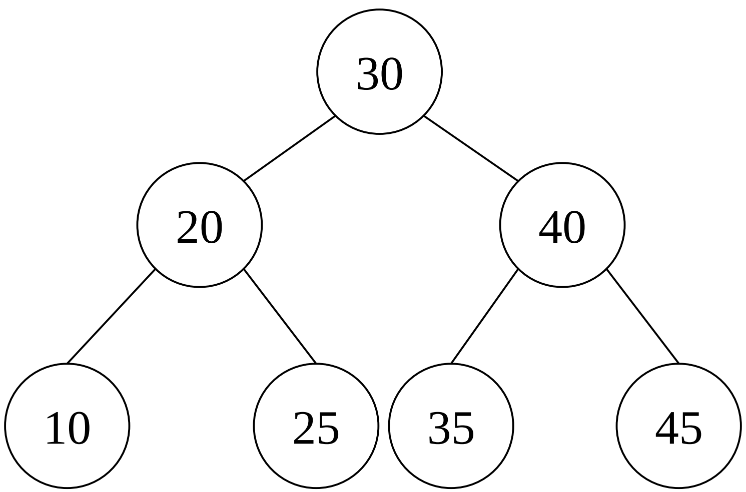
이진검색트리에서 검색

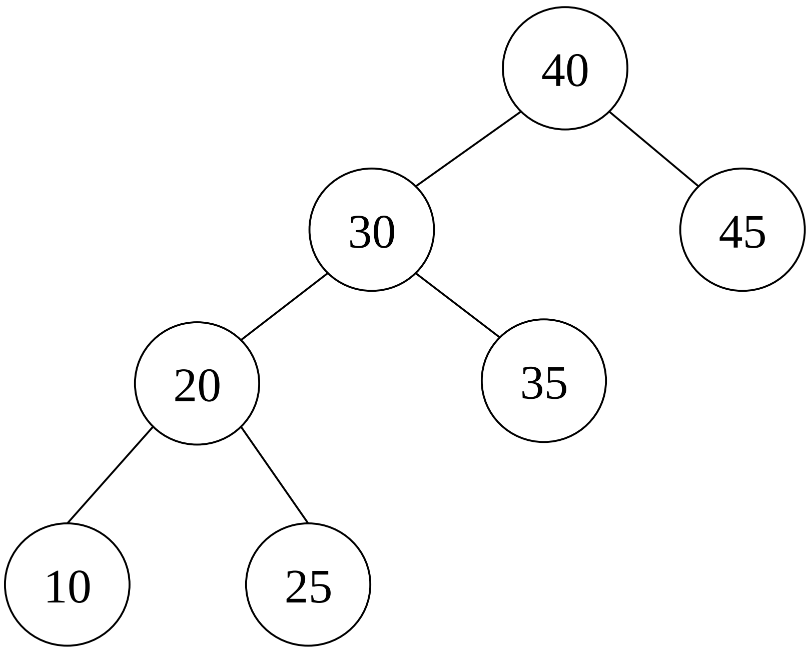
```
treeSearch(t, x)
▷ t: 트리의 루트 노드
▷ x: 검색하고자 하는 키
{
        if (t=NIL or key[t]=x) then return t;
        if (x < key[t])
                then return treeSearch(left[t], x);
                else return treeSearch(right[t], x);
}
```

이진검색트리에서 삽입

- **아래 상태에서**
  - 5를 삽입한 경우?
  - 29를 삽입한 경우?
  - 43을 삽입한 경우?

BST 특징:
왼쪽 자식은 부모보다 작고,
오른쪽 자식은 부모보다 크다.

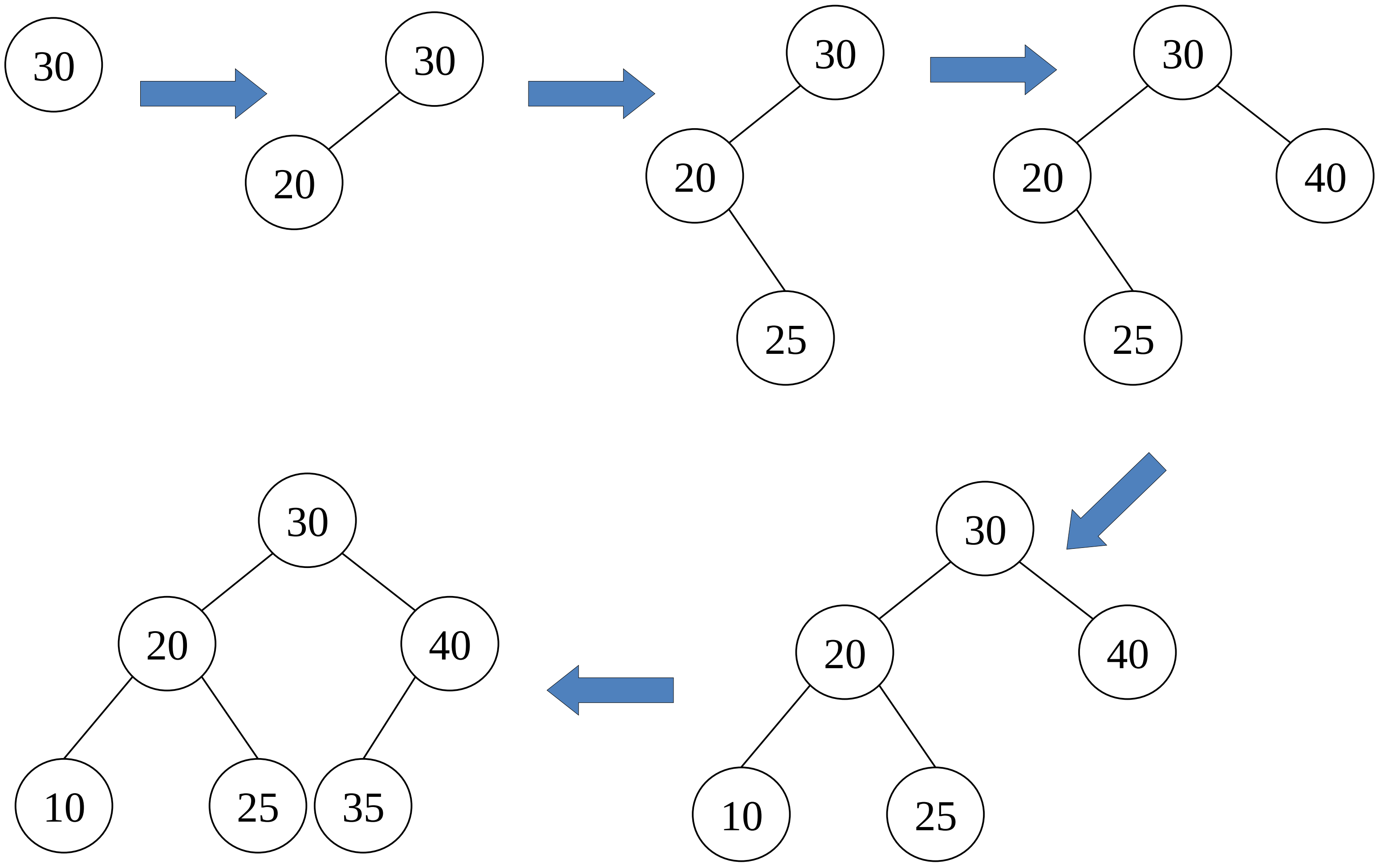




이진검색트리에서 삽입

```
# t: 현재 루트 노드, p: 부모 노드
bst_insert(t, x, p) {
       트리 t가 비어 있으면
              → 부모 p의 키값과 x를 비교해서
                 왼쪽이나 자식에 달아 준다.

       x <  t.key → 왼쪽 자식 트리에서 찾는다.
       t.key <  x → 오른쪽 자식 트리에서 찾는다.
}
```

이진검색트리에서 삽입



이진검색트리에서 삽입(ver.2)

```
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;}   ← 트리 전체를 리턴
        else {right[t] ← treeInsert(right[t], x); return t;}
}
```

연습문제

- **아래 순서대로 키를 삽입했을 때, 이진검색트리의 상태를 그려 보자.**
  - 1) 45, 40, 35, 30, 20, 10, 25
  - 2) 10, 20, 45, 40, 35, 30, 25
  - 3) 30, 20, 35, 10, 40, 25, 45

  - 이진검색트리의 평균적인 검색/삽입 시간은?

  - 이진검색트리의 검색/삽입에서 최악의 경우를 발생시키는 입력은?
  - 이 때 수행 시간은?

연습문제

- **bst_insert(), bst_search() 함수를 완성해 보자.**

```
class BST:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    def __repr__(self):
        return str(self.key)
```

```
def bst_insert(t, x):
    # 이미 존재하는 키는
    # 입력되지 않는다고 가정
    return t


def bst_search(t, x):
    return t
```

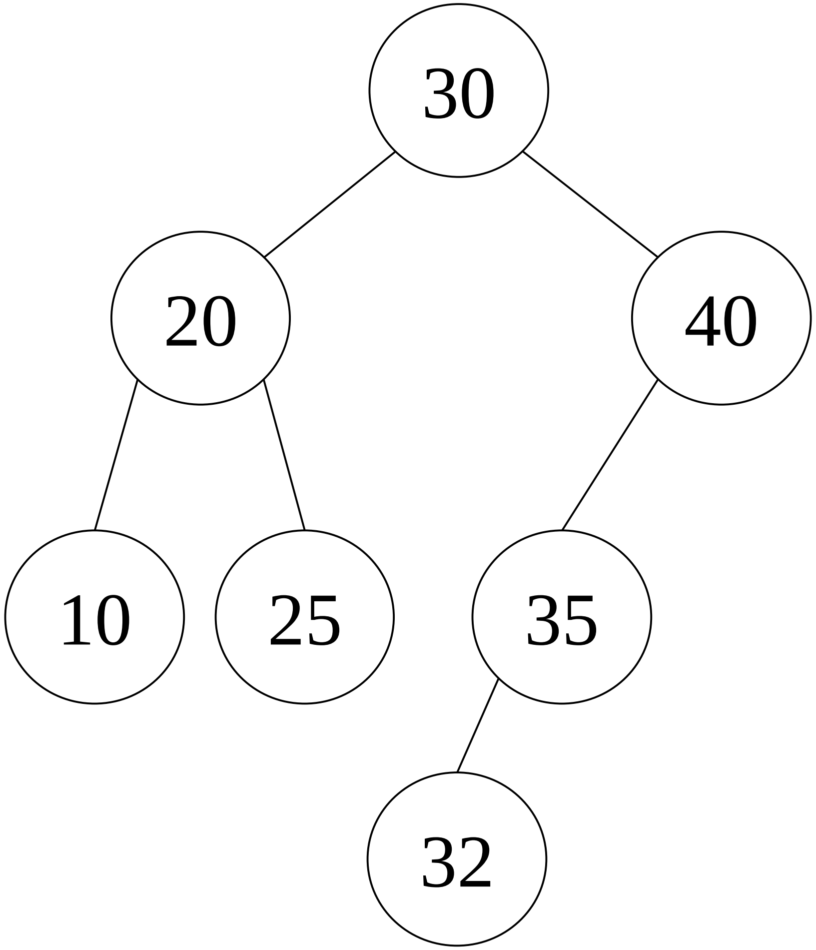
```
root = None
for x in [30, 20, 25, 40, 10, 35]:
    if not root: root = BST(x)
    else:        bst_insert(root, x)

print(bst_search(root, 25))
print(bst_search(root, 40))
```

이진검색트리에서 삭제

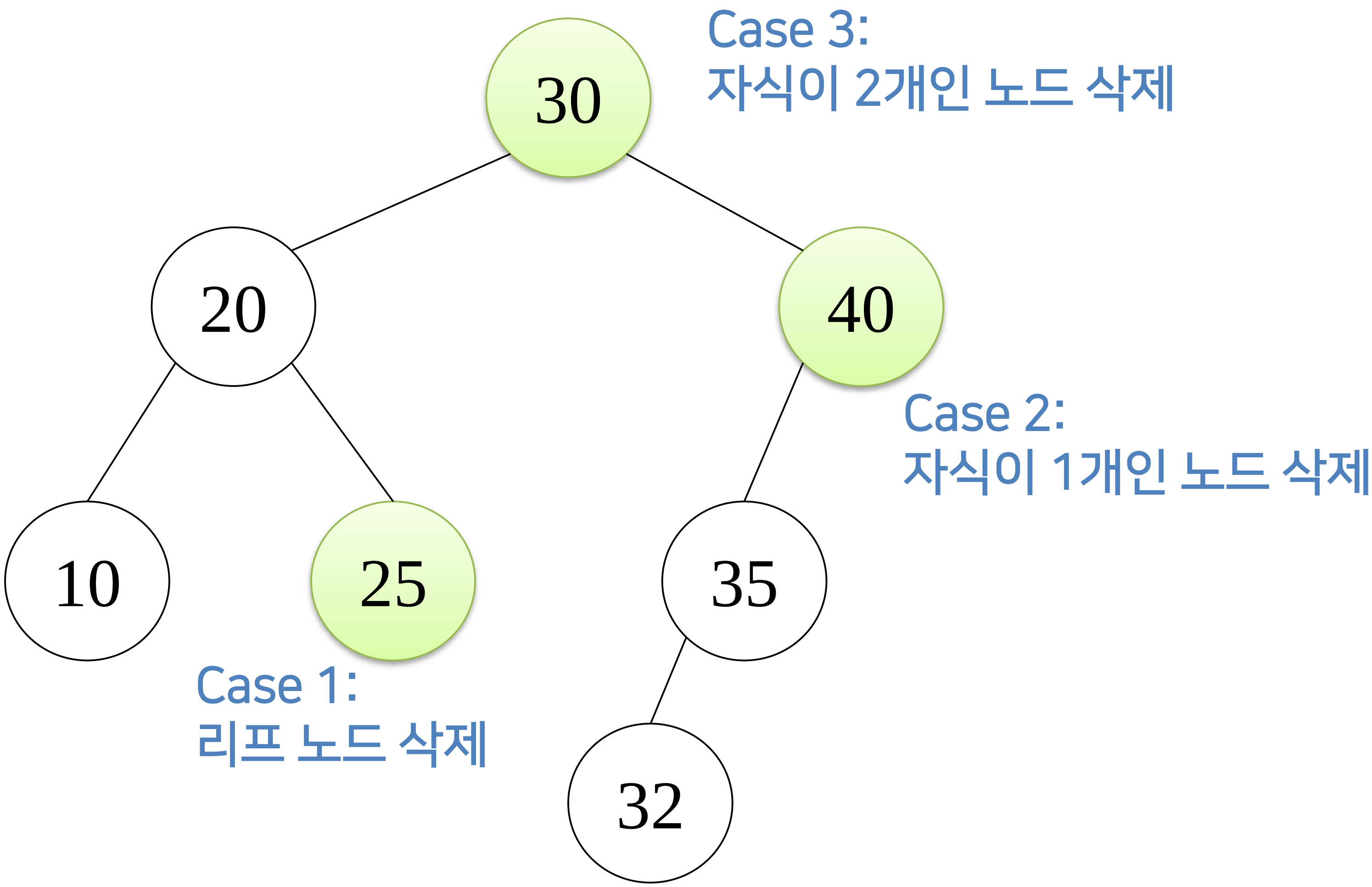
이진검색트리에서 삭제

- **아래 상태에서**
  - 25를 삭제한 경우?
  - 40를 삭제한 경우?
  - 30을 삭제한 경우?

BST 특징을 지켜줘야 함:
왼쪽 자식은 부모보다 작고,
오른쪽 자식은 부모보다 크다.



이진검색트리에서 삭제



30
20
40
10
25
35
32
Case 3:
자식이 2개인 노드 삭제
Case 2:
자식이 1개인 노드 삭제
Case 1:
리프 노드 삭제

이진검색트리에서 삭제

t: 트리의 루트 노드
r: 삭제하고자 하는 노드

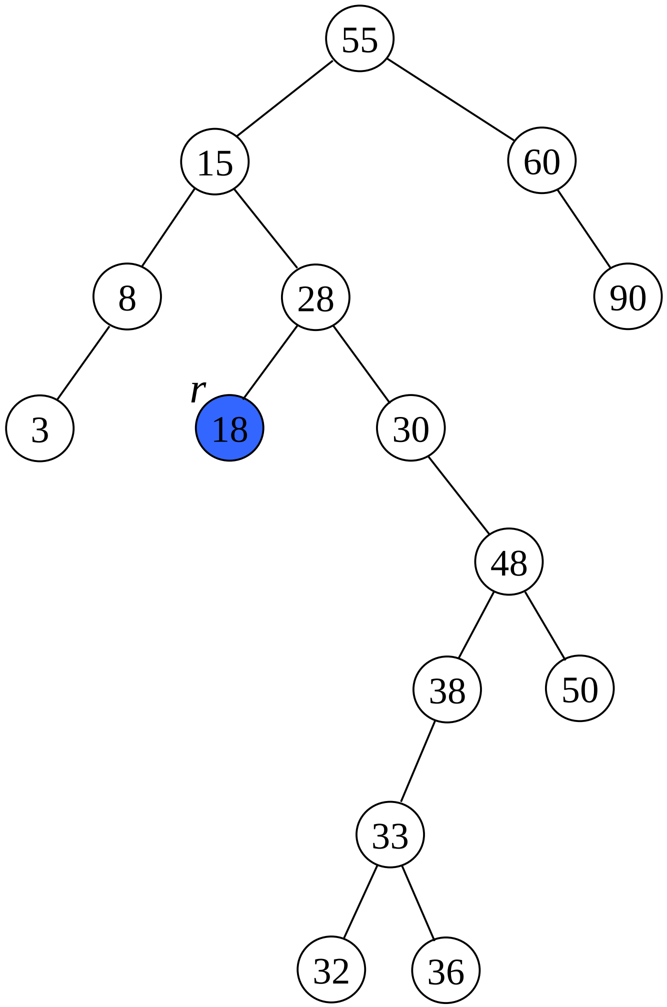
- **먼저 r의 위치를 찾는다.**

- **그 후 3가지 경우에 따라 다르게 처리한다.**
  - Case 1 : r이 리프leaf 노드인 경우
  - Case 2 : r의 자식 노드가 하나인 경우
  - Case 3 : r의 자식 노드가 두 개인 경우

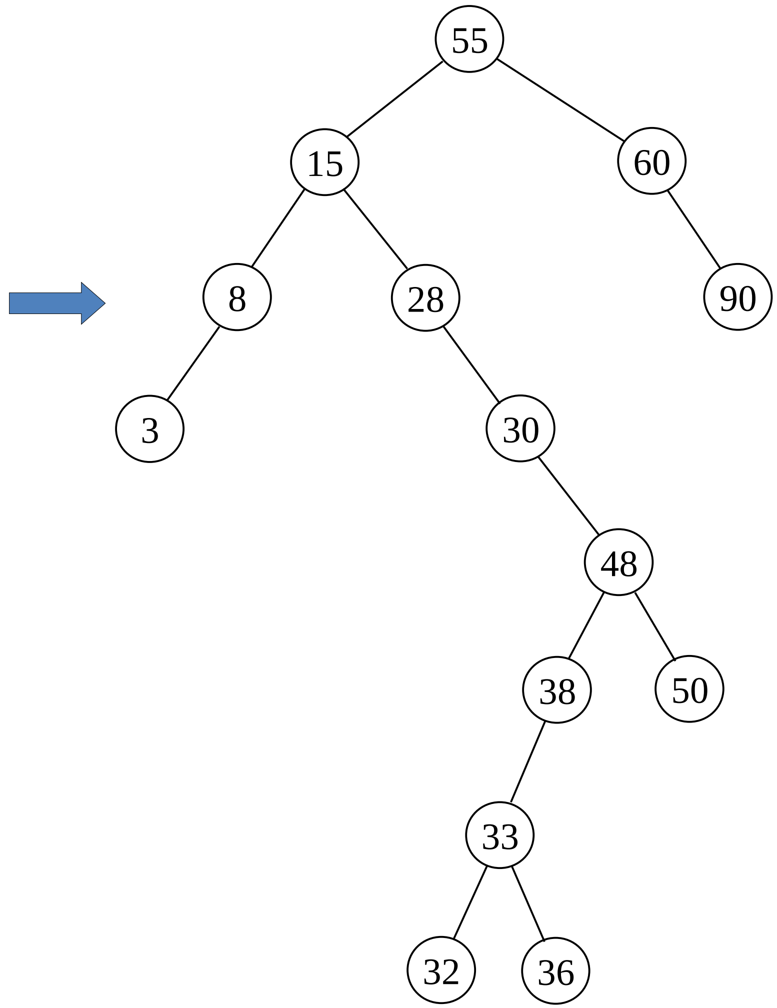
이진검색트리에서 삭제(Case 1)

```
Sketch-TreeDelete(t, r)
▷ t: 트리의 루트 노드
▷ r: 삭제하려는 노드
{
    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;
}
```

삭제의 예: Case 1



(a) *r*의 자식이 없음

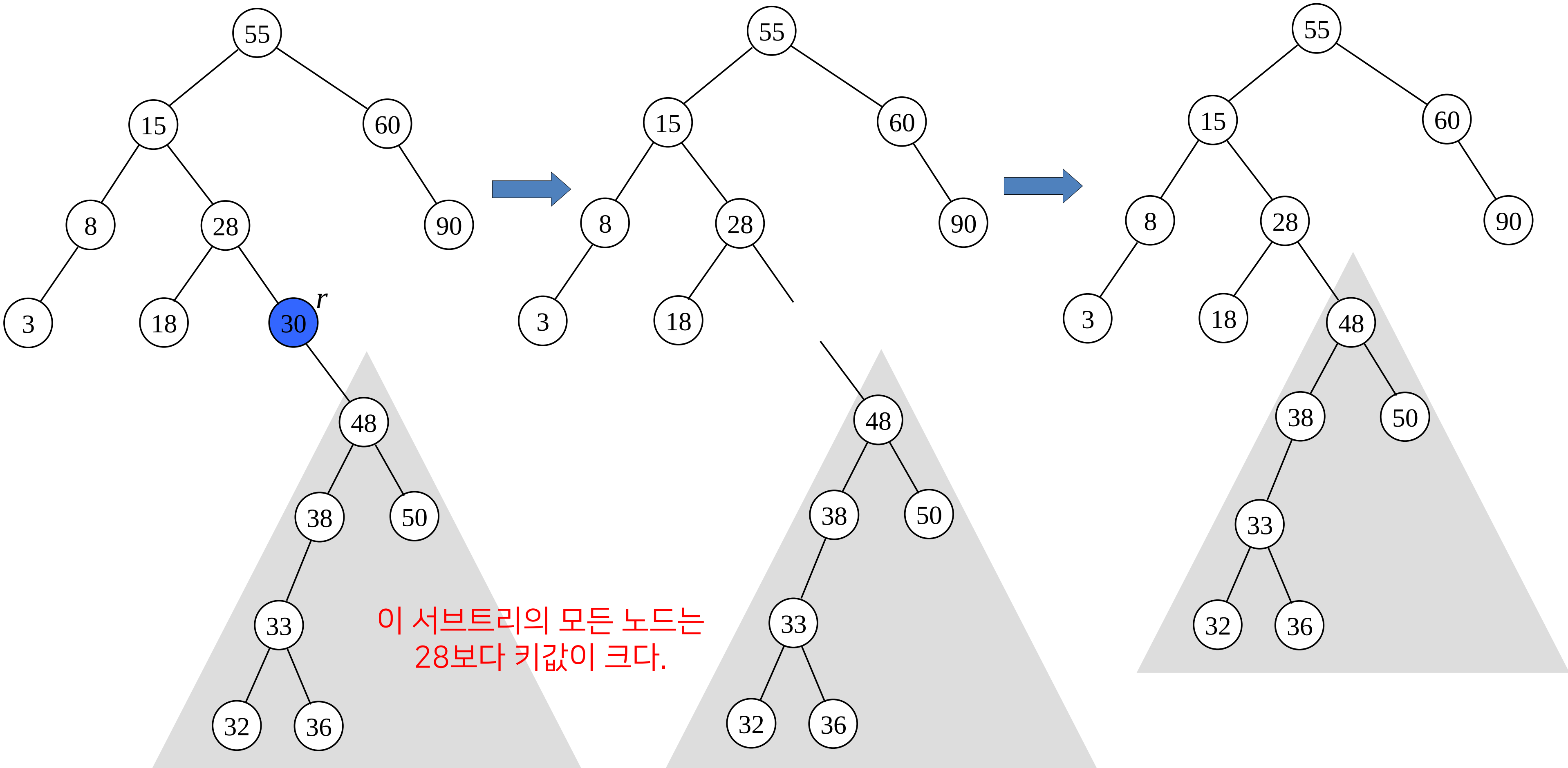


(b) 단순히 *r*을 제거한다

이진검색트리에서 삭제(Case 2)

```
Sketch-TreeDelete(t, r)
▷ t: 트리의 루트 노드
▷ r: 삭제하려는 노드
{
    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;
}
```

삭제의 예: Case 2



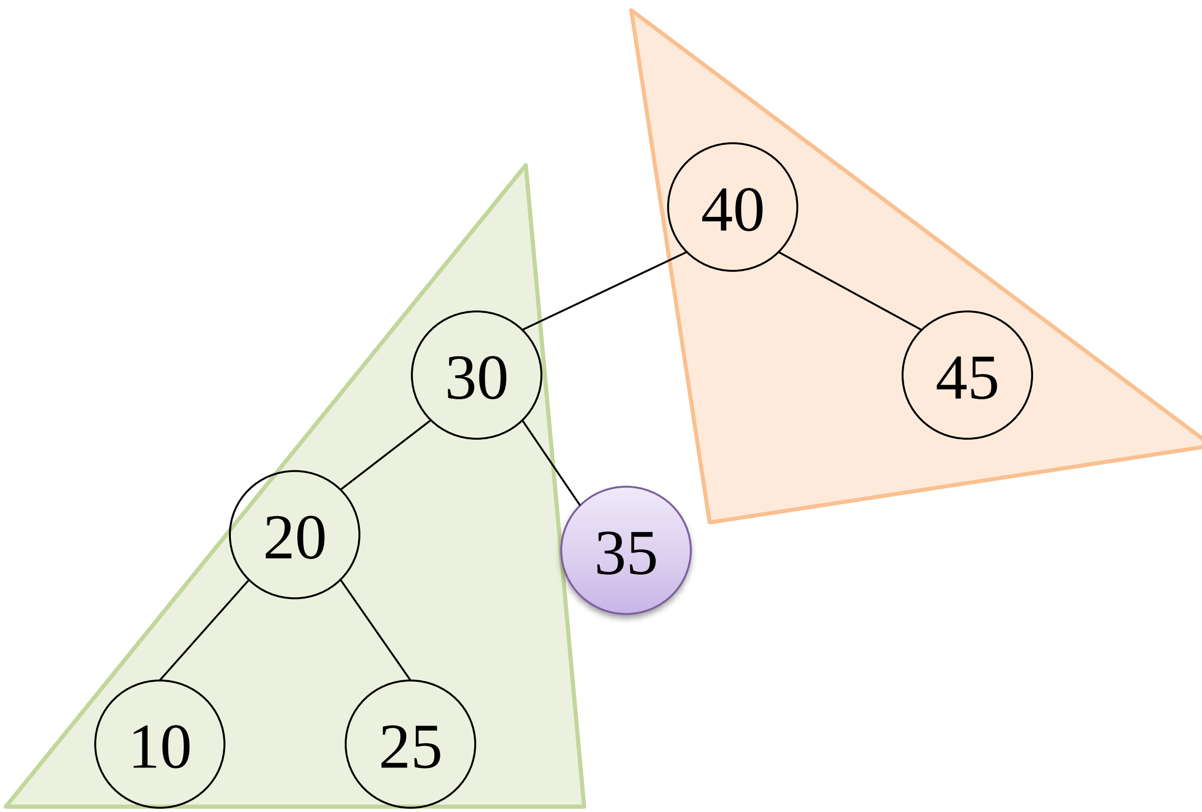
(a) *r*의 자식이 하나뿐임 (b) *r*을 제거 (c) *r* 자리에 *r*의 자식을 놓는다

이진검색트리에서 삭제(Case 3)

```
Sketch-TreeDelete(t, r)
▷ t: 트리의 루트 노드
▷ r: 삭제하려는 노드
{
    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;
}
```

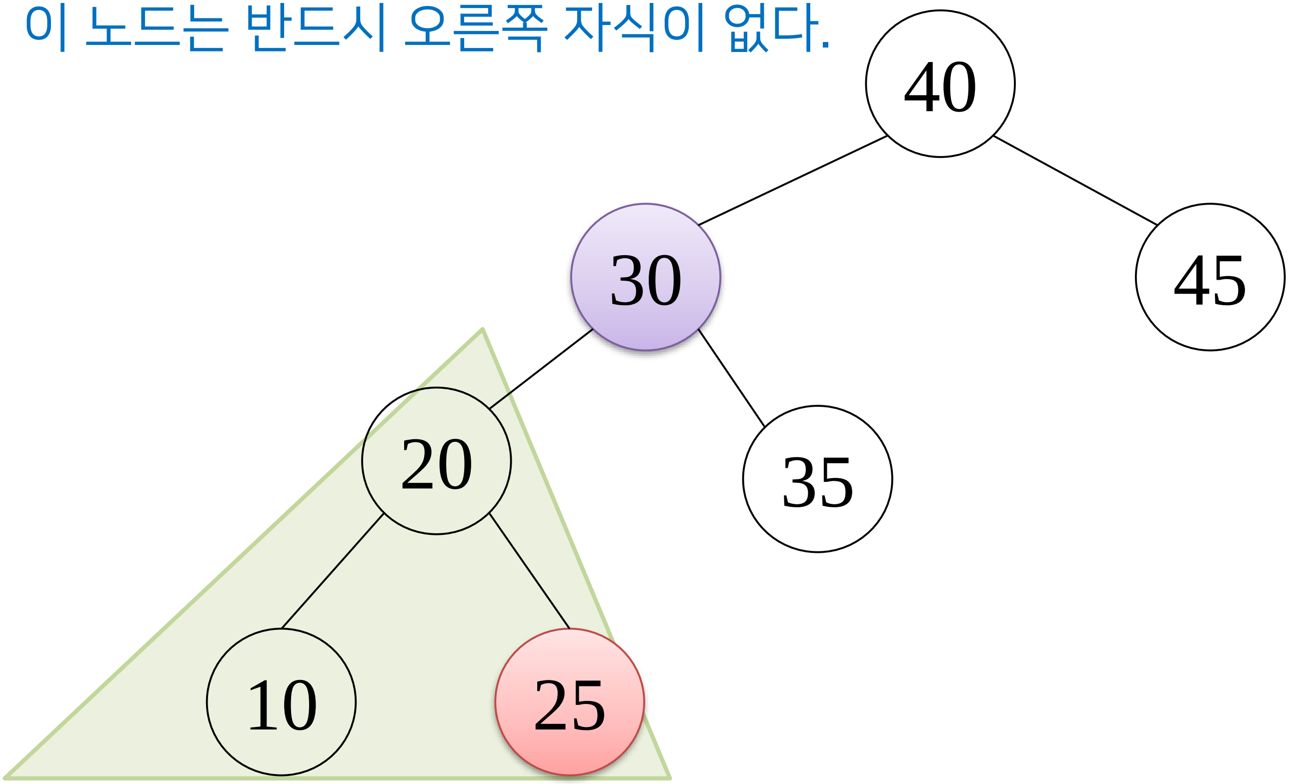
이진검색트리의 대소 관계

- **35를 기준으로**
  - 35보다 왼쪽에 있는 노드들은 35보다 작다.
  - 35보다 오른쪽에 있는 노드들은 35보다 크다.



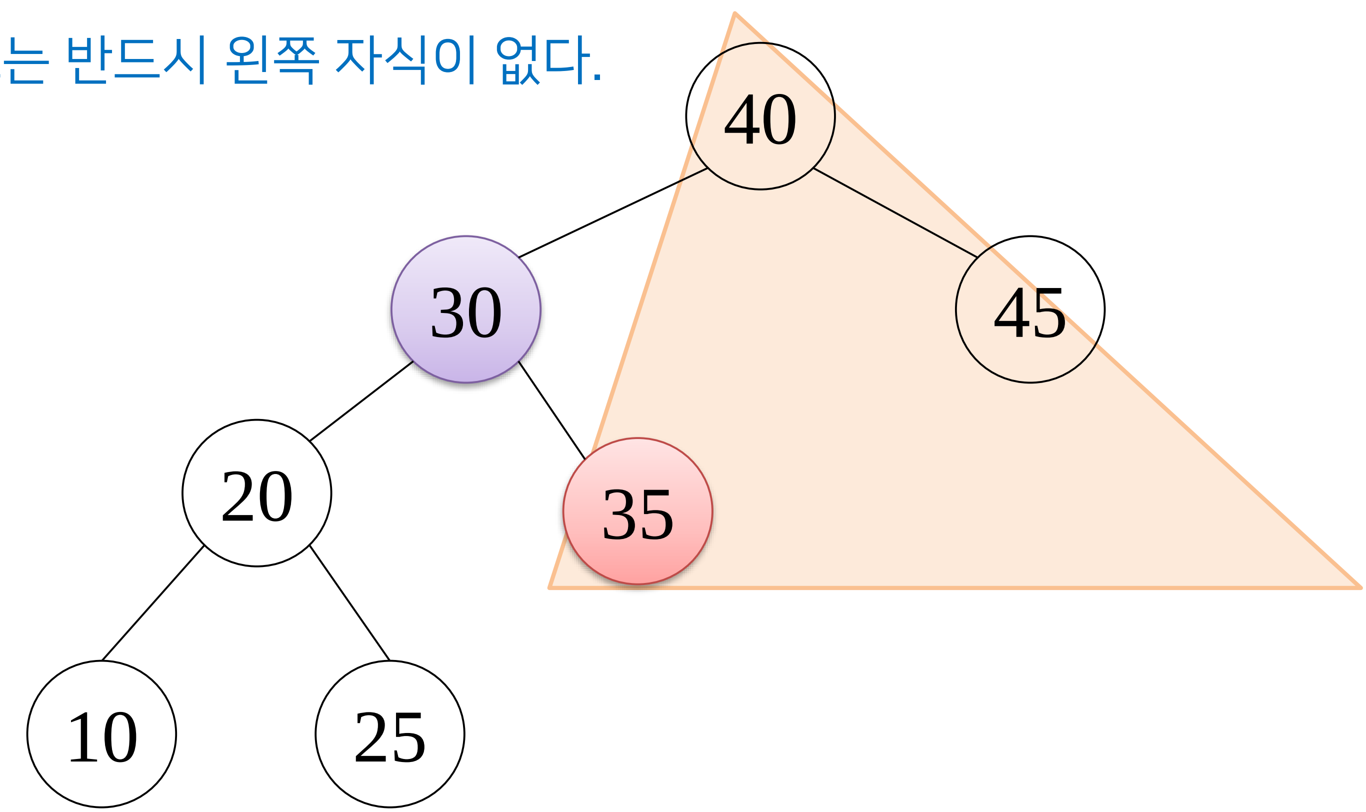
이진검색트리의 대소 관계

- **30을 기준으로**
  - 30보다 왼쪽에 있는 노드들은 30보다 작다.
  - 30의 왼쪽에 있는 노드 중 가장 오른쪽에 있는 노드는 <30보다 작은 노드> 중 최댓값
  - 이 노드는 반드시 오른쪽 자식이 없다.



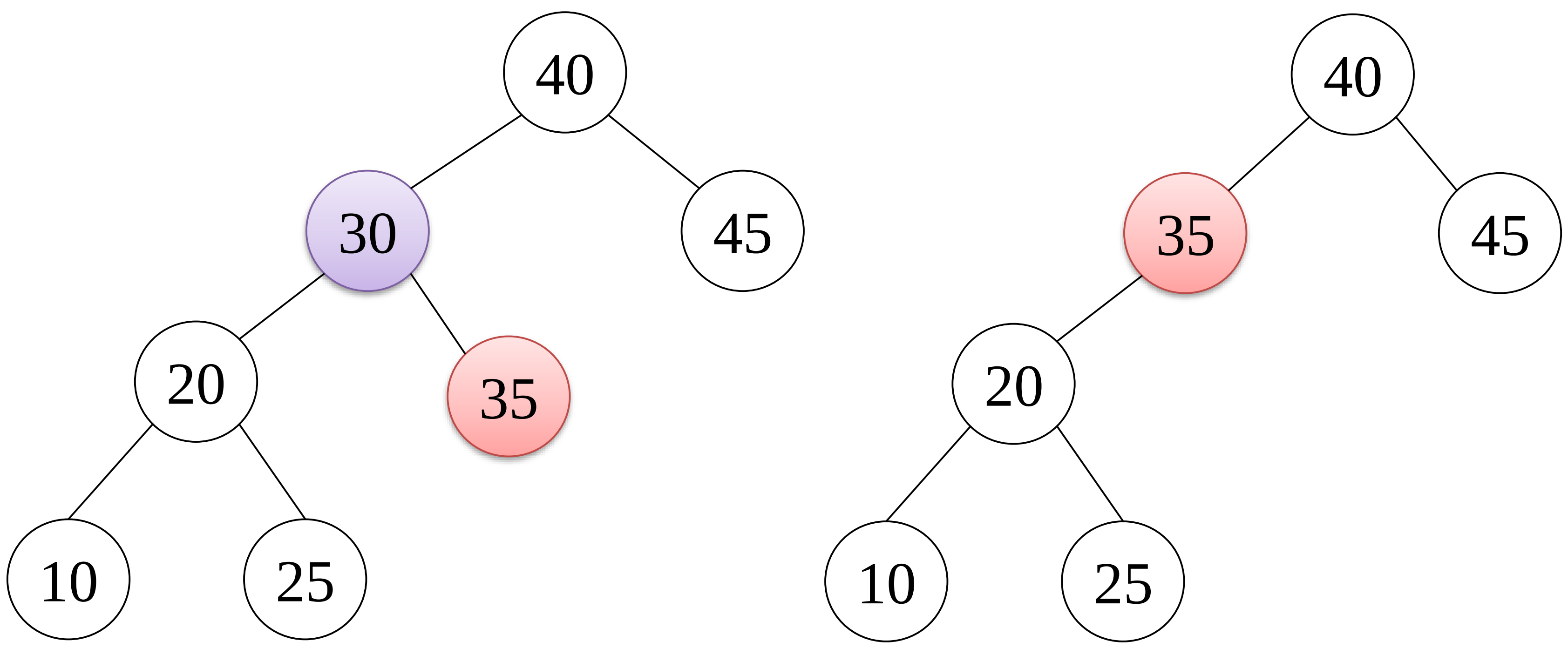
이진검색트리의 대소 관계

- **30을 기준으로**
  - 30보다 오른쪽에 있는 노드들은 30보다 크다.
  - 30의 오른쪽에 있는 노드 중 가장 왼쪽에 있는 노드는 <30보다 큰 노드> 중 최솟값
  - 이 노드는 반드시 왼쪽 자식이 없다.

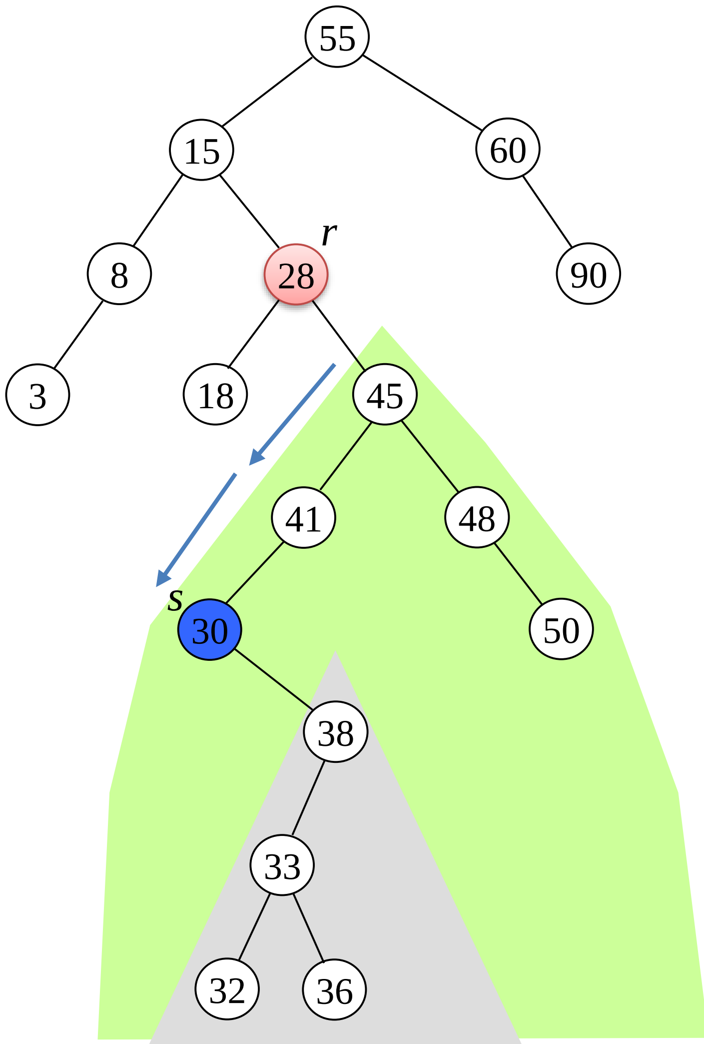


이진검색트리에서 삭제(Case 3)

- 30을 삭제하고
- 30과 가장 가까운 수(25, 35) 중 하나를 그 자리에 두면
- BST의 특성이 지켜진다.



삭제의 예: Case 3 (자식이 둘 다 있음)



(a) *r*의 직후원소 *s*를 찾는다

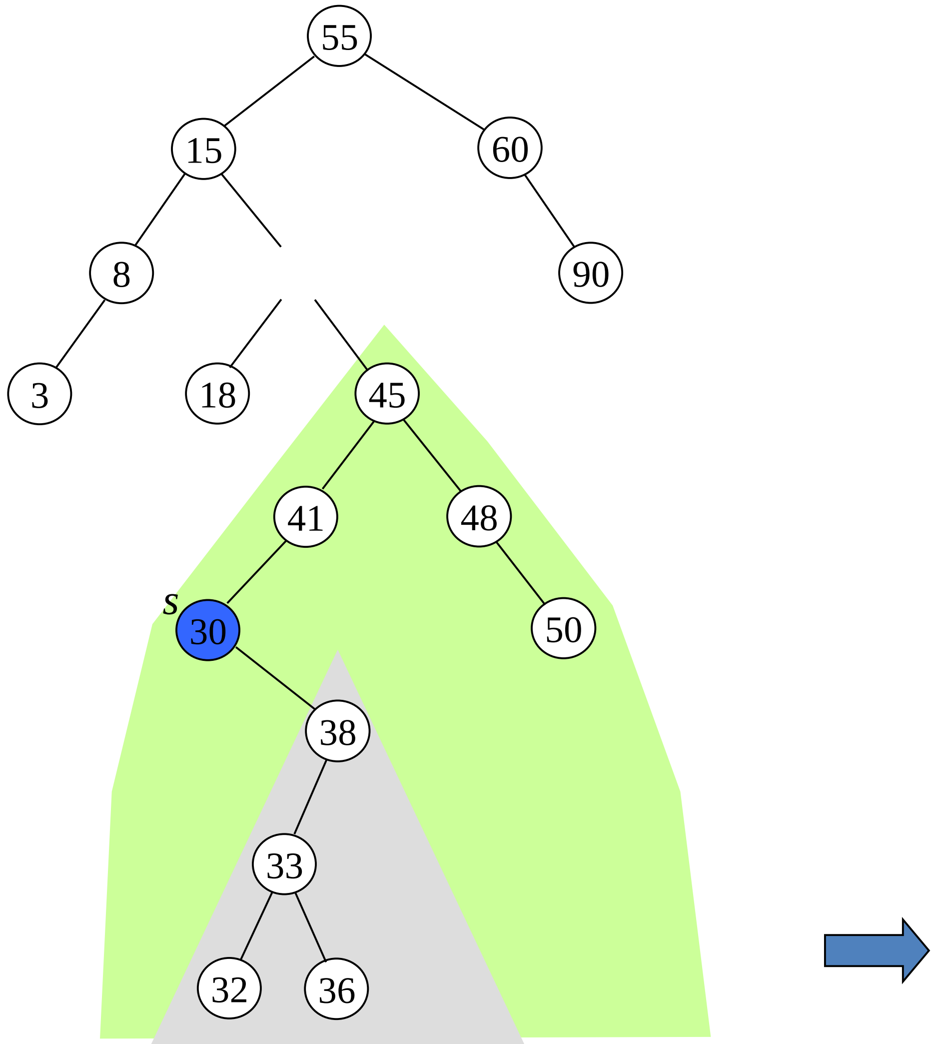
28을 삭제:

30은 <28보다 큰 모든 노드들> 중 최솟값

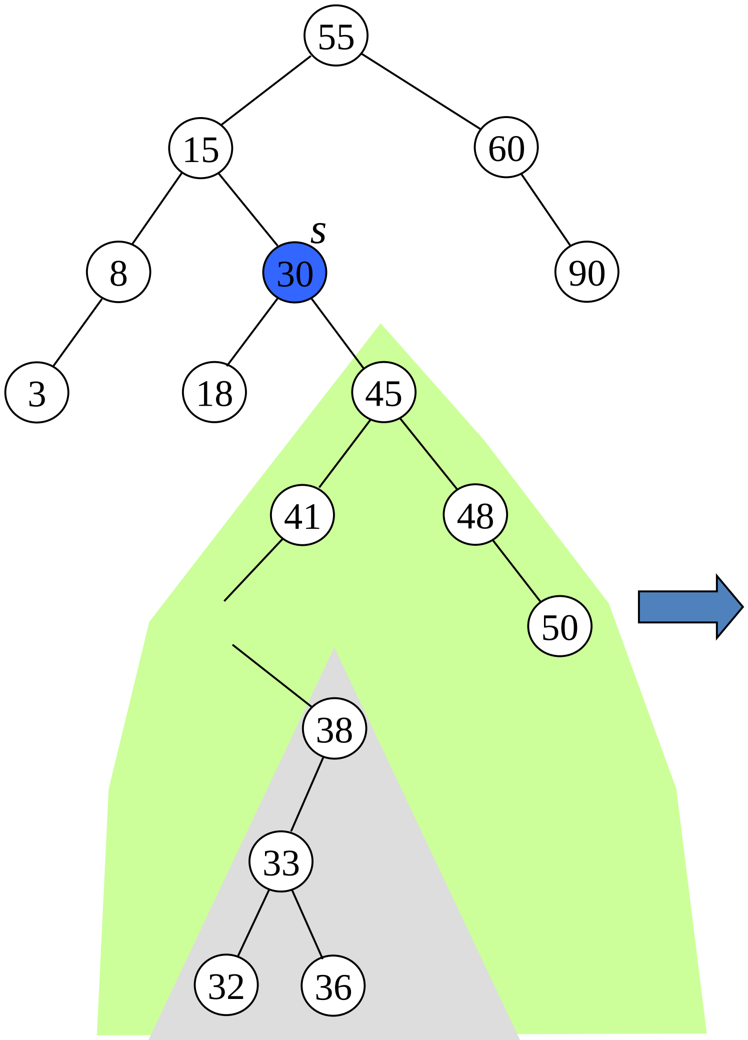
28의 오른쪽 서브트리에서
왼쪽으로 끝까지 가서 찾는다.

30은 반드시 왼쪽 자식이 없다.

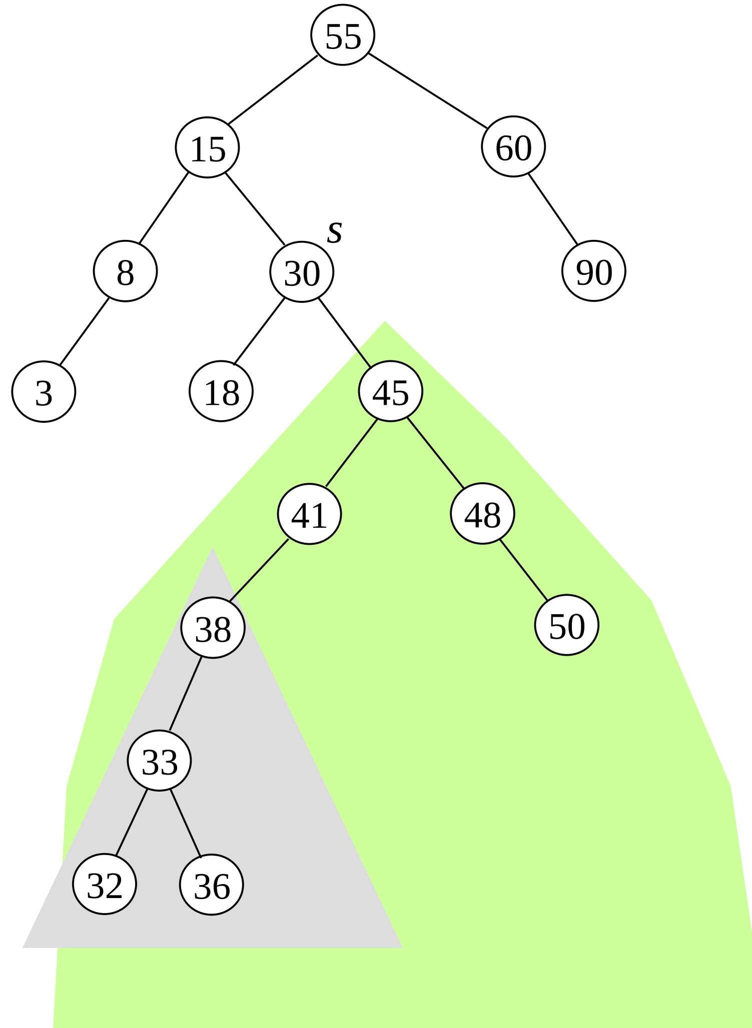
삭제의 예: Case 3



(b) *r*을 없앤다



(c) *s*를 *r*자리로 옮긴다



(d) *s*가 있던 자리에 *s*의 자식을 놓는다

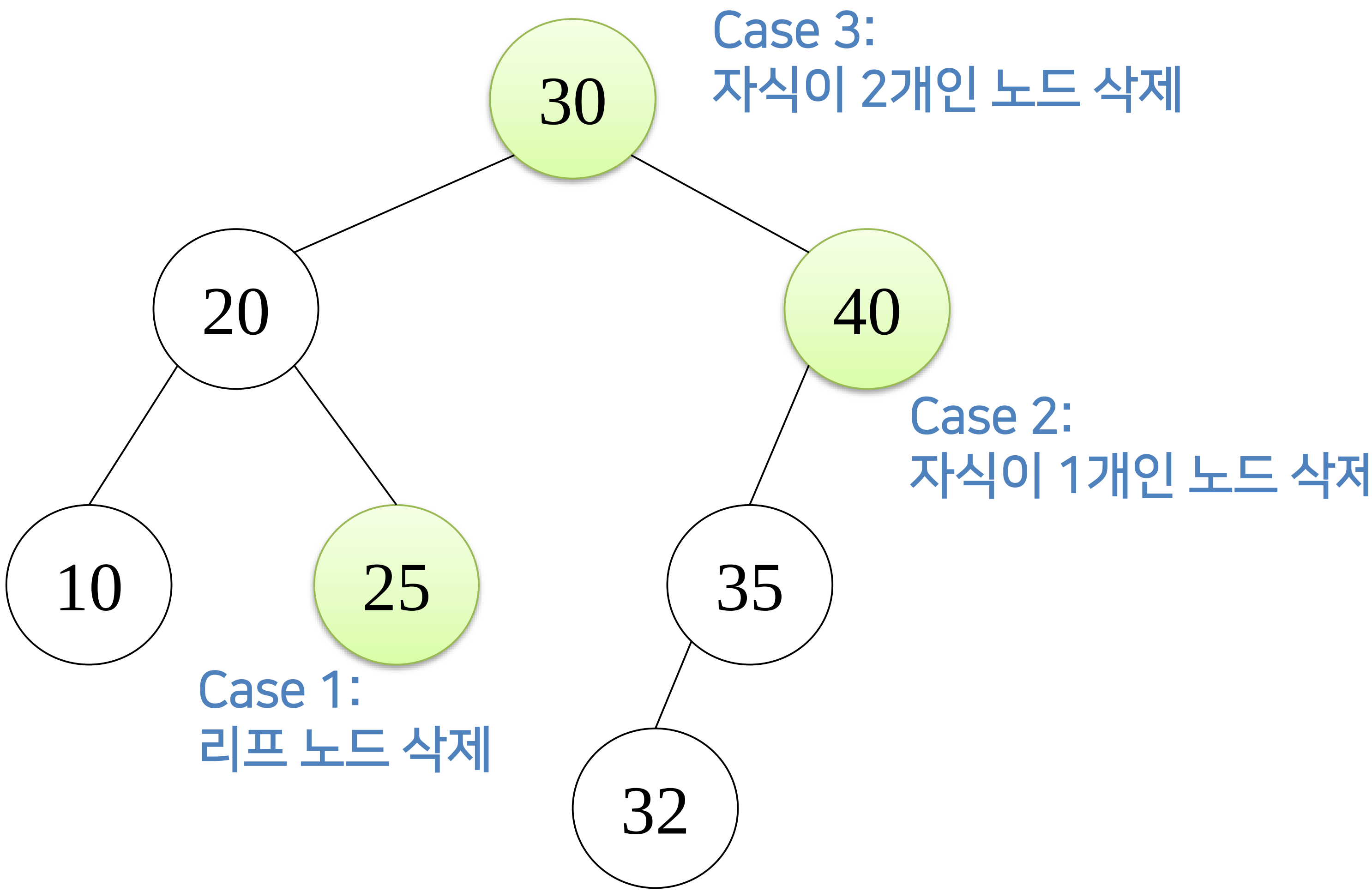
이진검색트리에서 삭제(Case 3)

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;
    }
}
```

연습문제

- 아래 BST에서 3가지 경우의 삭제 과정을 각각 그려 보자.

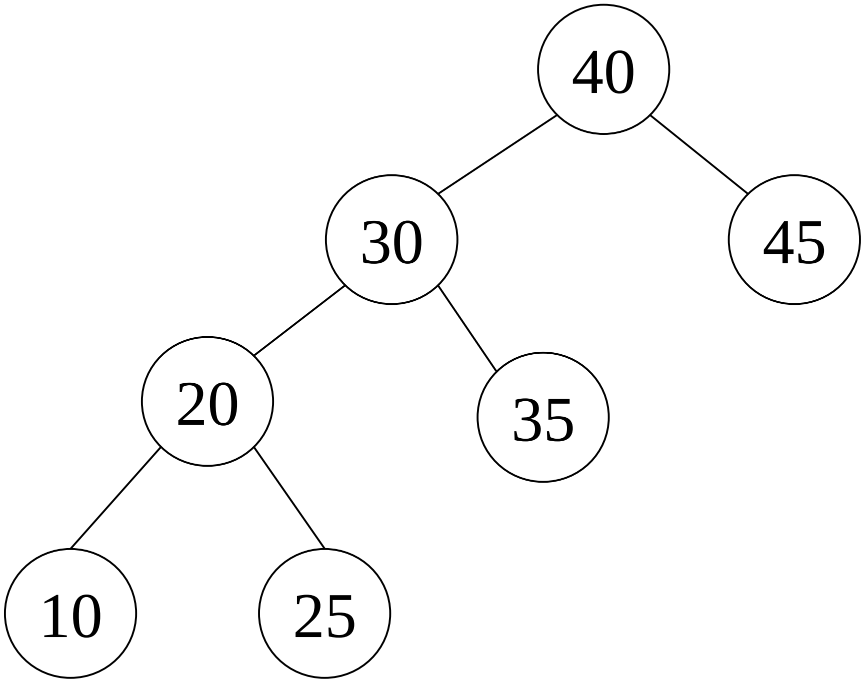


레드블랙트리

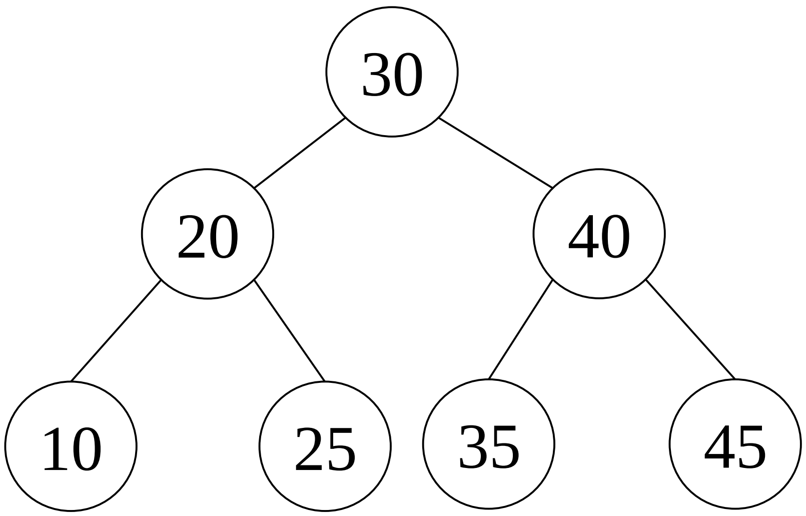
이진검색트리의 균형

- **불균형한 BST의 단점은?**

탐색 경로 길이(평균): 1.71

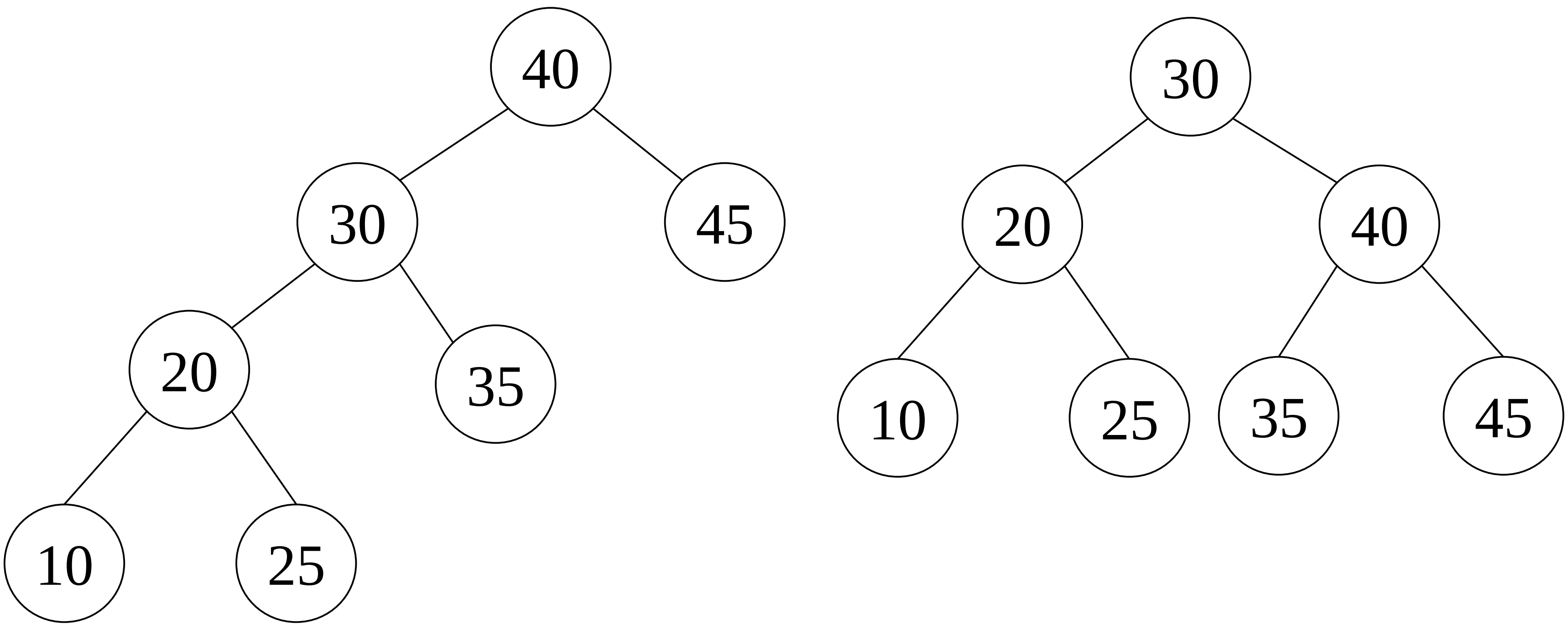


탐색 경로 길이(평균): 1.43



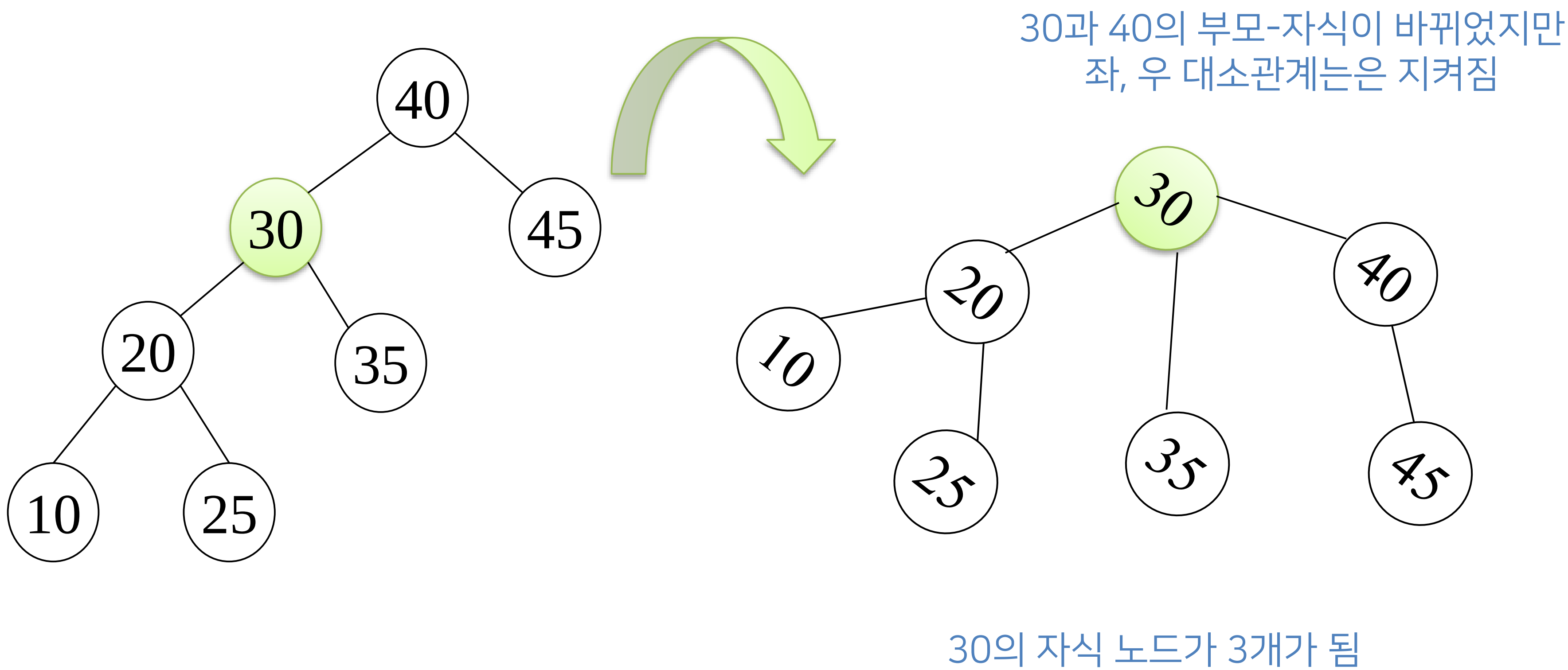
이진검색트리의 균형

- **BST의 균형을 잡는 방법은?**
- **입력이 들어오는 순서에 관여할 수 없다면?**

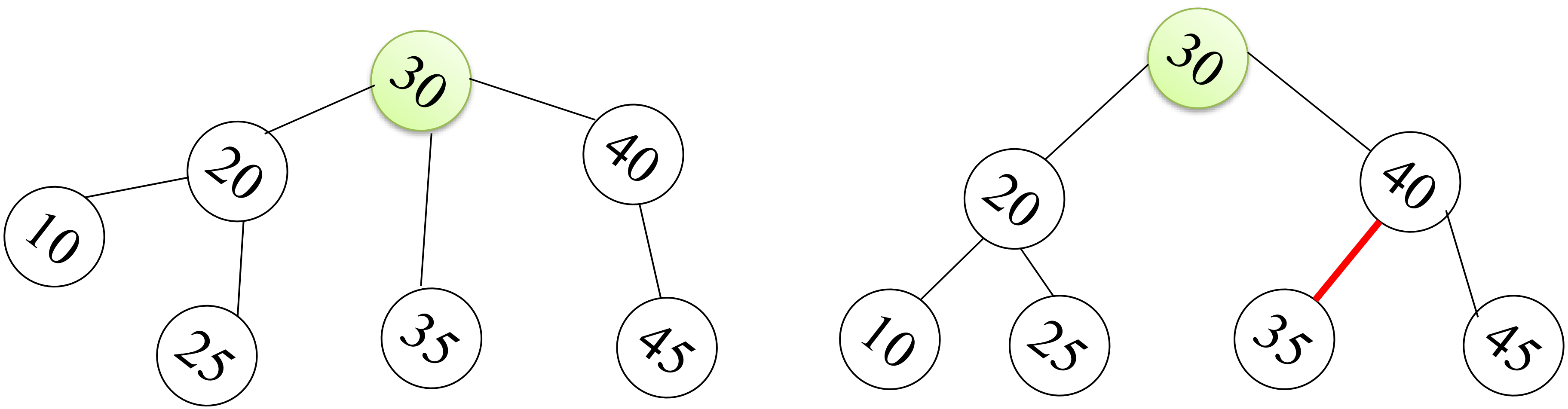


이진검색트리의 균형

- 30을 중심으로 회전시켜 보자.



이진검색트리의 균형



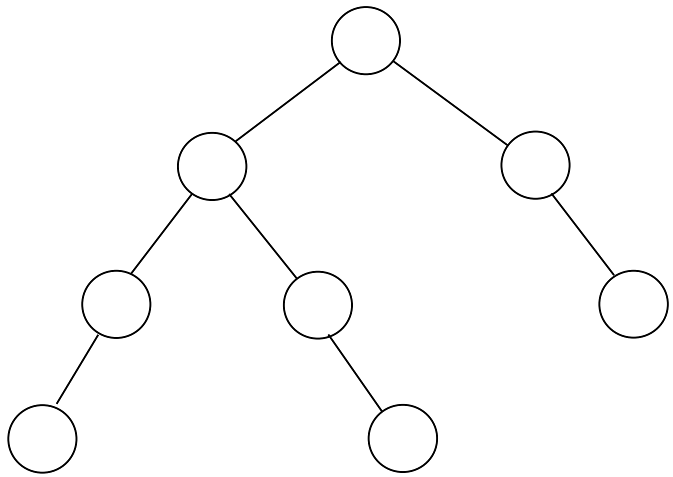
BST의 특징을 지키면서
트리의 최대 깊이가 1 감소!

레드블랙트리

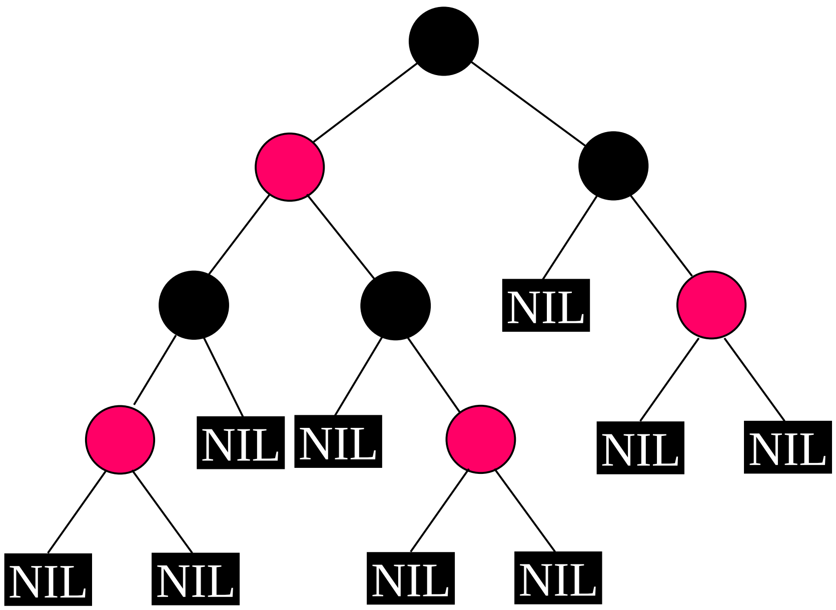
- **균형 잡힌 이진트리(Balanced Binary Tree)**
  - 리프 노드들의 깊이 차이가 최소가 되도록 유지

- **레드블랙트리(Red-Black Tree)**
  - 균형 잡힌 이진 탐색 트리
  - 모든 노드를 블랙 또는 레드로 각각 칠한다.
  - 단, 색을 칠할 때는 몇 가지 제약 조건을 만족시킴으로써 트리의 균형을 유지한다.

이진검색트리를 레드블랙트리로 만든 예

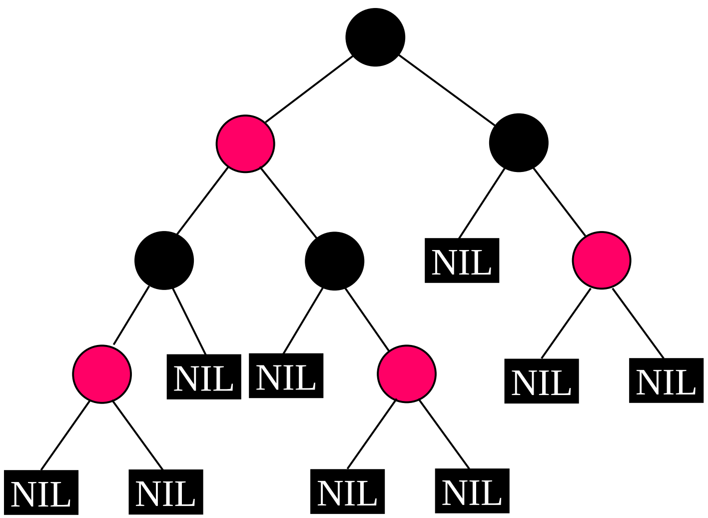


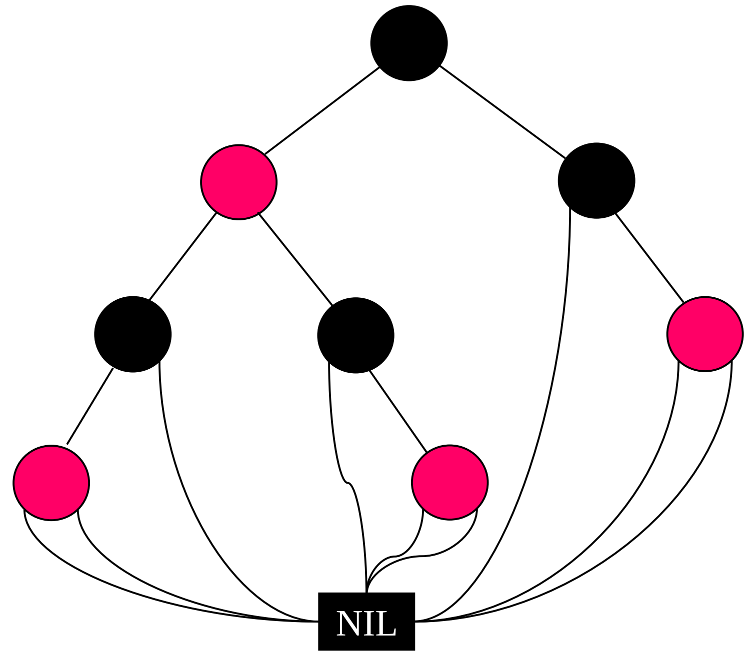
(a) 이진검색트리의 한 예



(b) (a)를 레드블랙트리로 만든 예

레드블랙트리





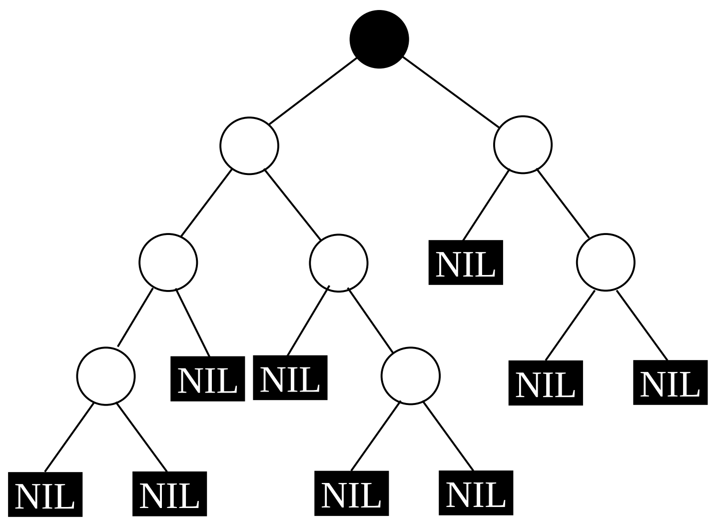
(c) 실제 구현시의 NIL 노드 처리 방법

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

레드블랙트리

- ## 레드블랙트리의 특징

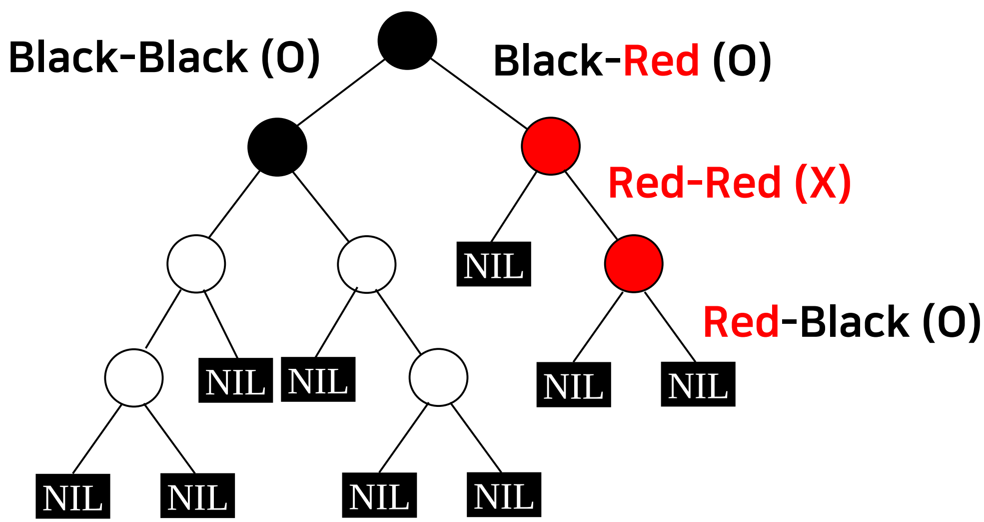
① 루트는 블랙
② 모든 리프는 블랙
③ 레드 노드의 자식은 반드시 블랙
④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다.



레드블랙트리

- **레드블랙트리의 특징**

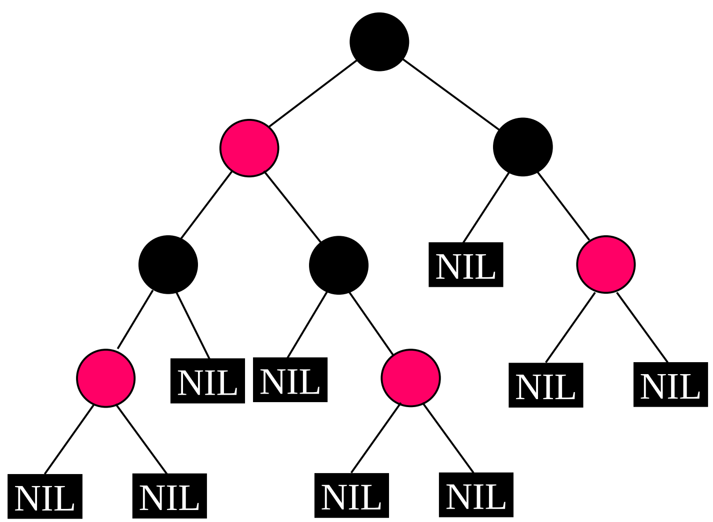
① 루트는 블랙
② 모든 리프는 블랙
③ 레드 노드의 자식은 반드시 블랙
④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다.



레드블랙트리

- **레드블랙트리의 특징**

① 루트는 블랙
② 모든 리프는 블랙
③ 레드 노드의 자식은 반드시 블랙
④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다.

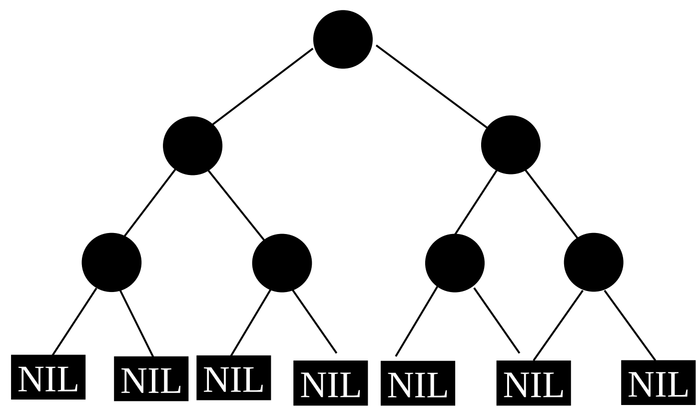


어떤 root-leaf 경로를 선택해도
블랙 노드는 3개

레드블랙트리

① 루트는 블랙
② 모든 리프는 블랙
③ 레드 노드의 자식은 반드시 블랙
④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다.

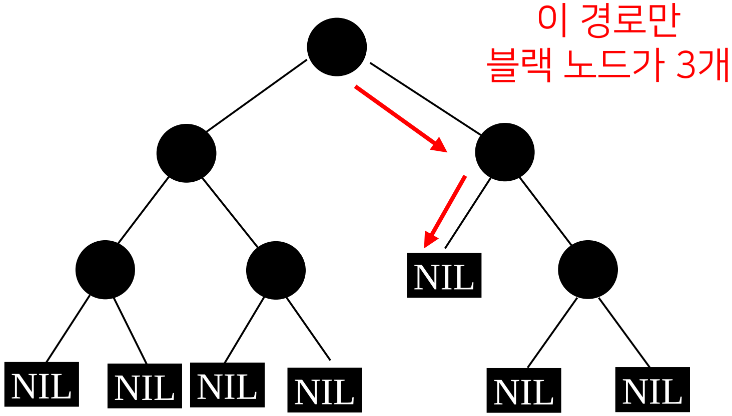
꽉 찬 이진 트리



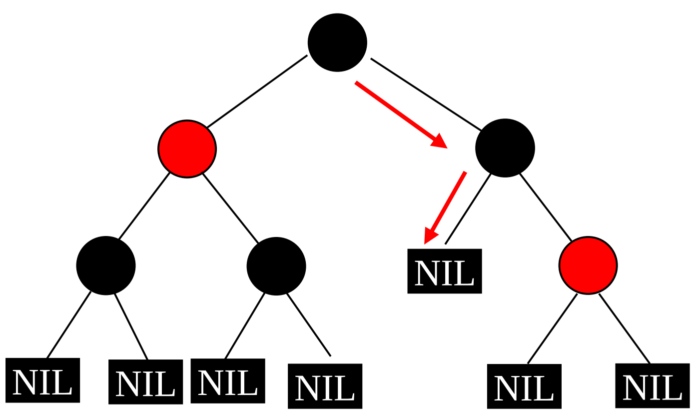
레드블랙트리

① 루트는 블랙
② 모든 리프는 블랙
③ 레드 노드의 자식은 반드시 블랙
④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다.

4번 규칙에 위배되는 경우



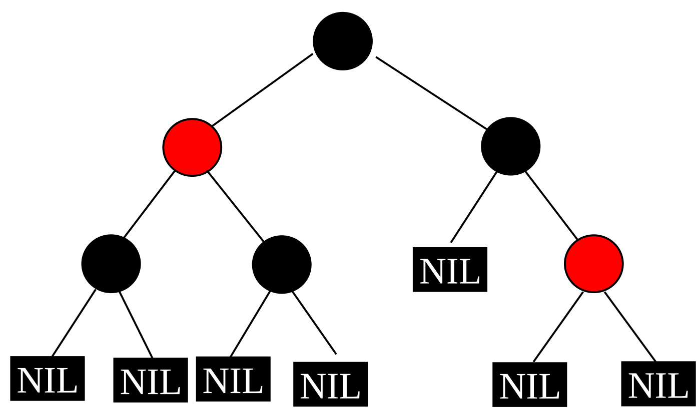
나머지 경로들의 블랙 노드 수를
1개씩 감소시킴



레드블랙트리

① 루트는 블랙
② 모든 리프는 블랙
③ 레드 노드의 자식은 반드시 블랙
④ 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다.

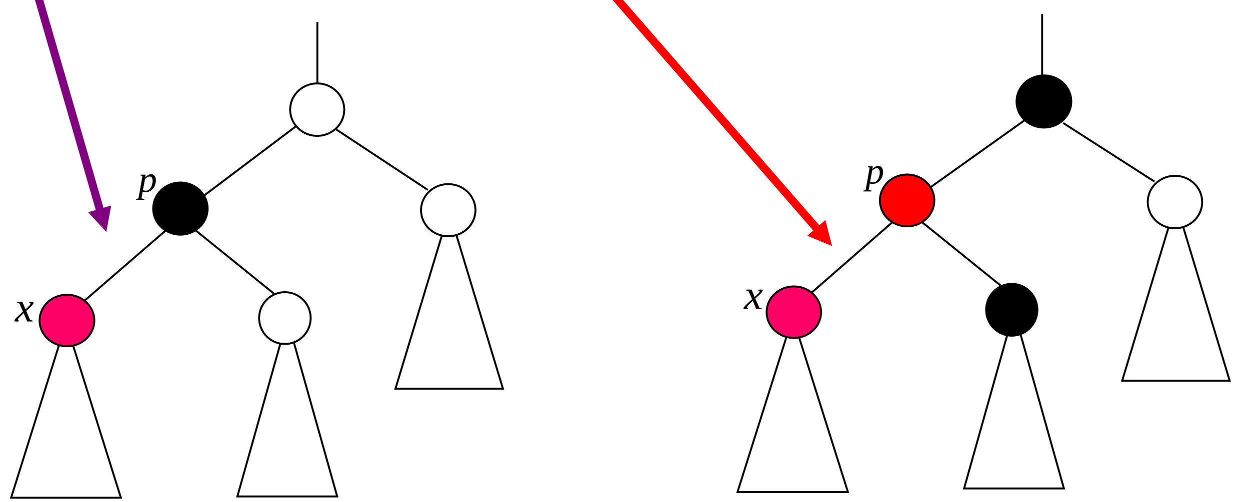
앞에서 보여준 것처럼, **트리를 적절히 회전시키면서**
3번과 4번 규칙이 지켜지도록 레드/블랙을 칠하면 트리의 균형을 유지할 수 있다.



레드블랙트리에서 삽입

레드블랙트리에서의 삽입

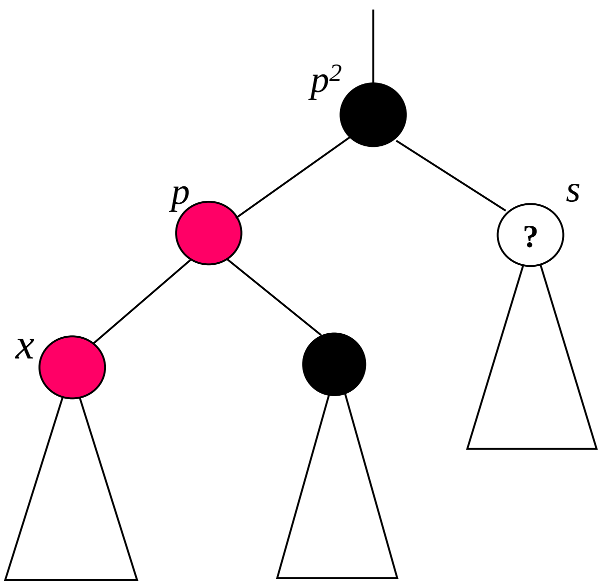
- 이진검색트리에서의 삽입과 같다. 다만 삽입 후 삽입된 노드를 레드로 칠한다. (이 노드를 x라 하자)
- 만일 x의 부모 노드 p의 색상이
  - 블랙이면 아무 문제 없다.
  - 레드이면 레드블랙특성 ③이 깨진다.



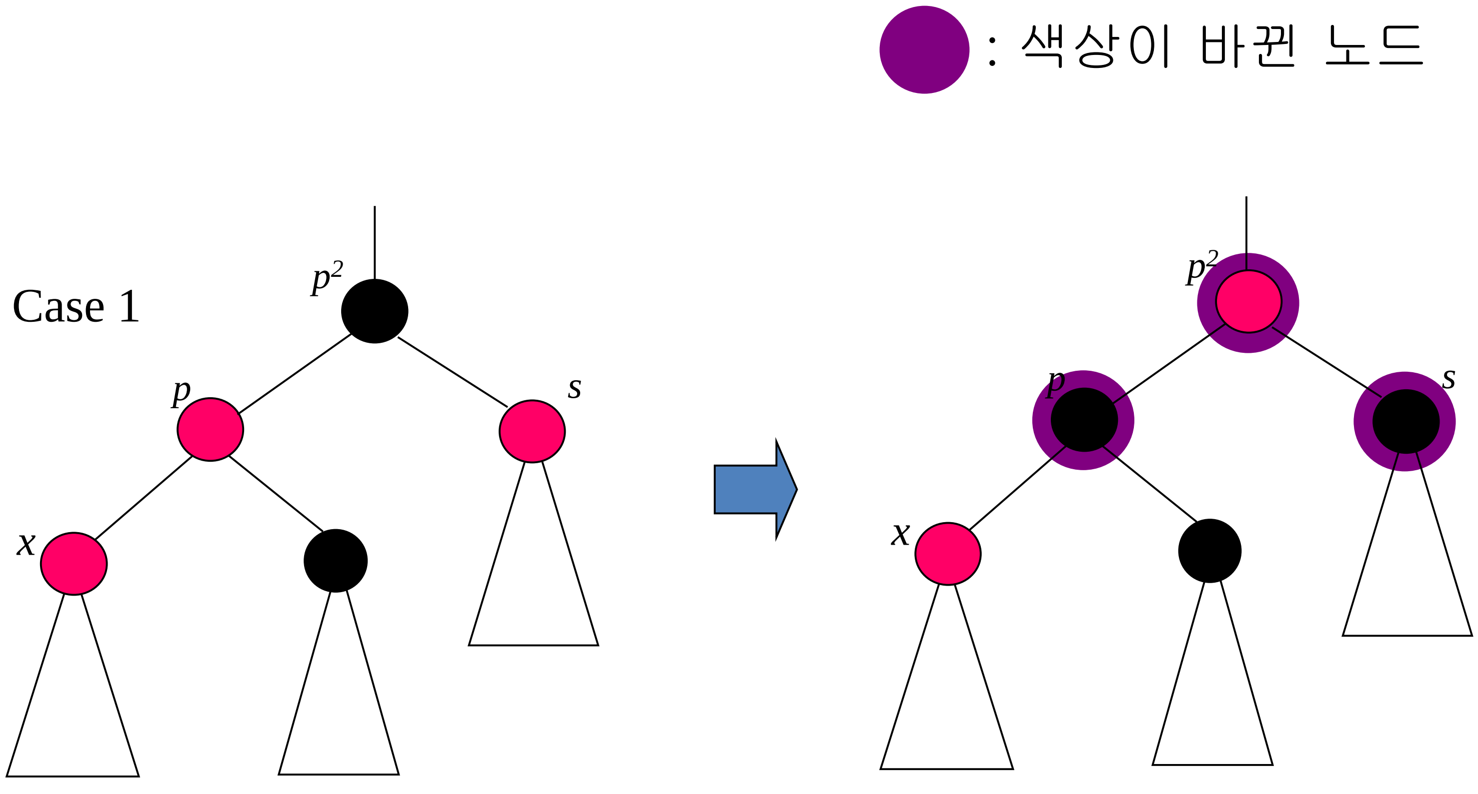
✓ 그러므로 p가 레드인 경우만 고려하면 된다

레드블랙트리에서의 삽입

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

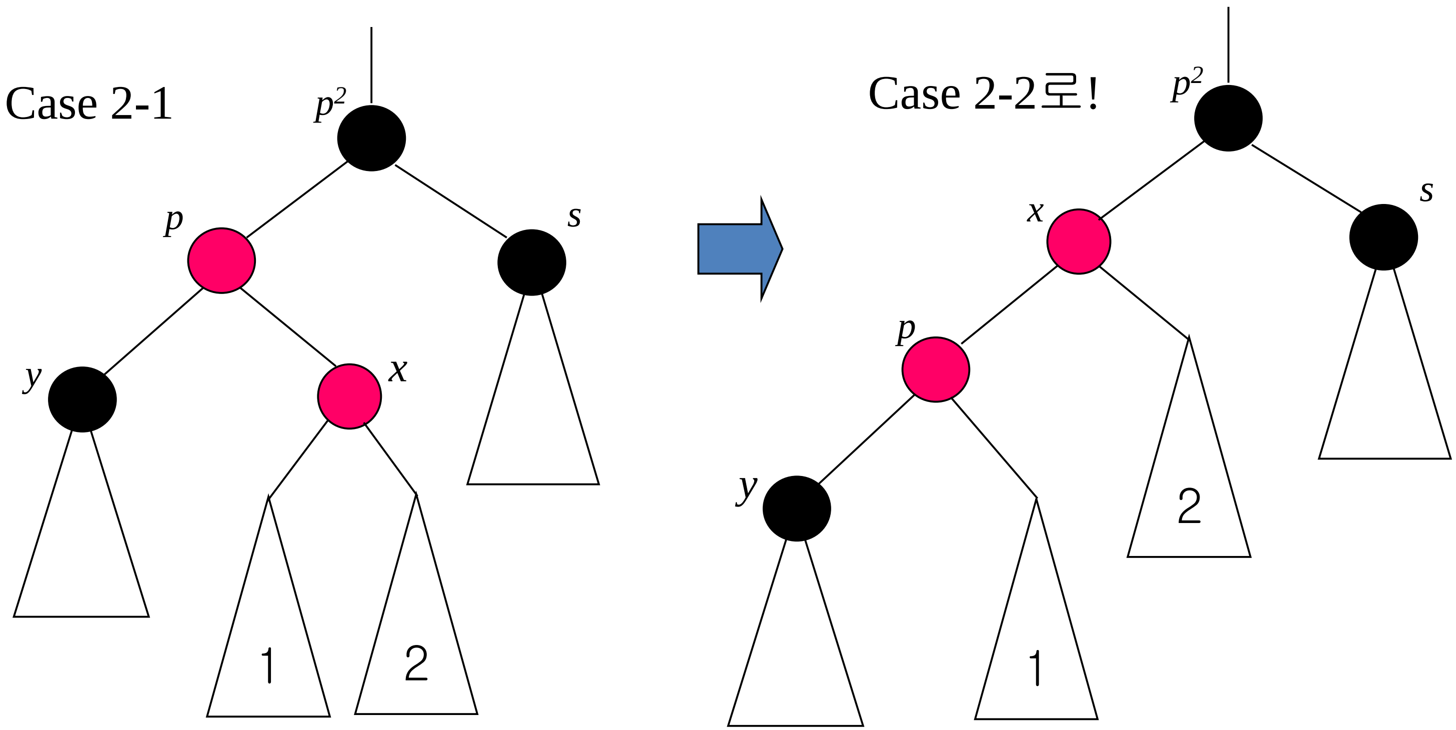


Case 1: *s*가 레드

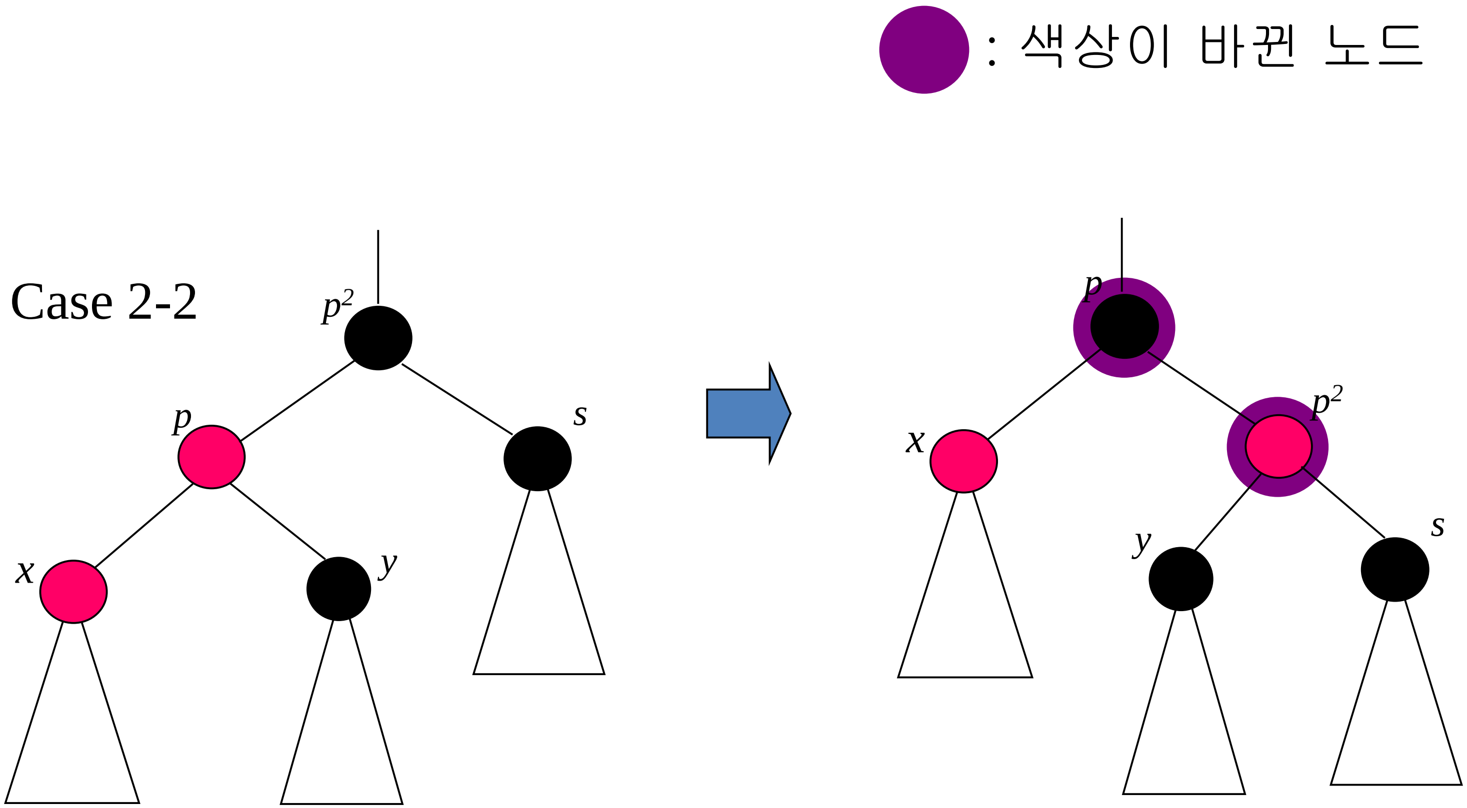


✓ p^2에서 방금과 같은 문제가 발생할 수 있다

Case 2-1: s가 블랙이고, x가 p의 오른쪽 자식



Case 2-2: s가 블랙이고, x가 p의 왼쪽 자식

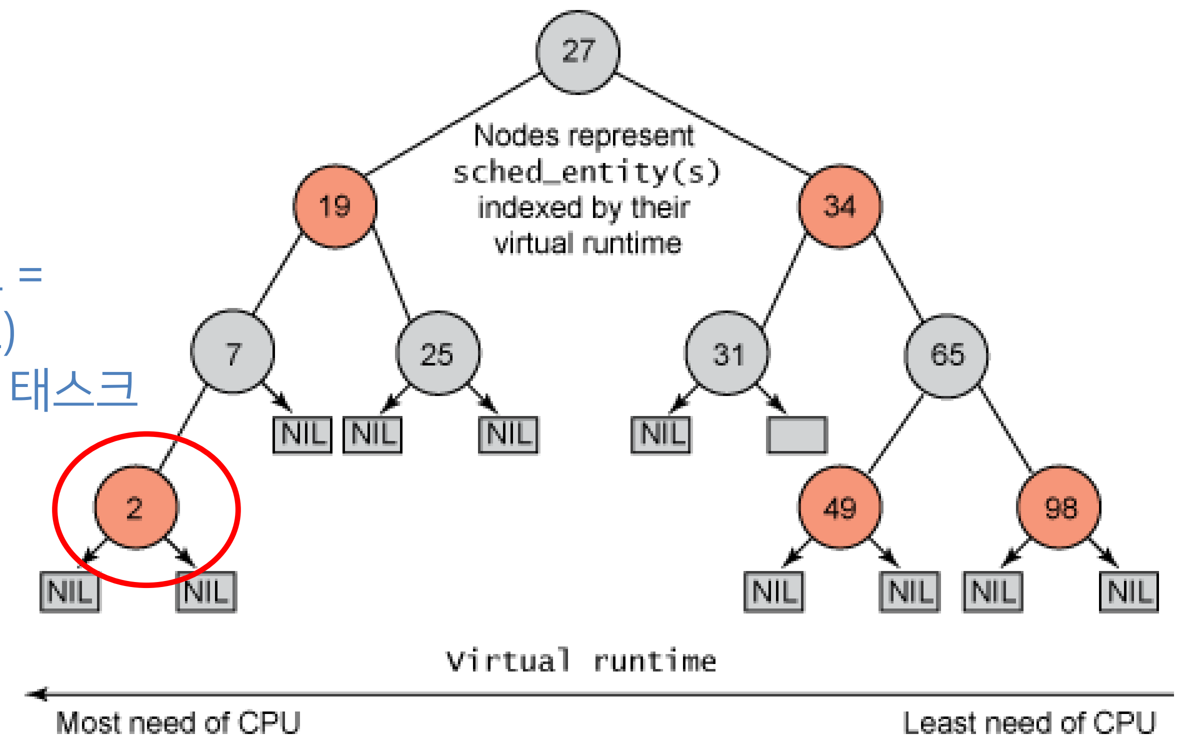


✓ 삽입 완료!

Linux CFS

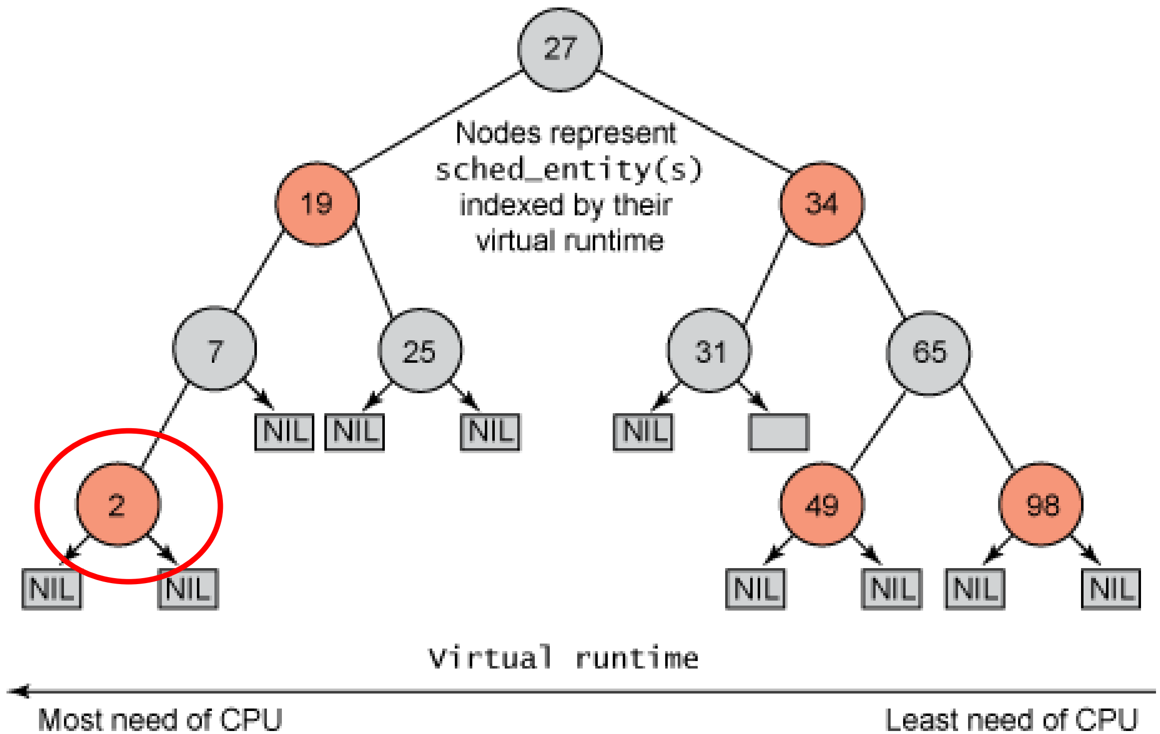
- **리눅스 CFS(Completely Fair Scheduler)**
  – RBTree가 사용된 대표적인 예
  – 각 태스크(프로세스)에 virtual runtime을 공정하게 분배하기 위해
  – 트리에서 가장 왼쪽 노드에 CPU를 할당한다(dispatch).

가장 왼쪽에 있는 노드 =
CPU를 (상대적으로)
가장 적게 할당받은 태스크



Linux CFS

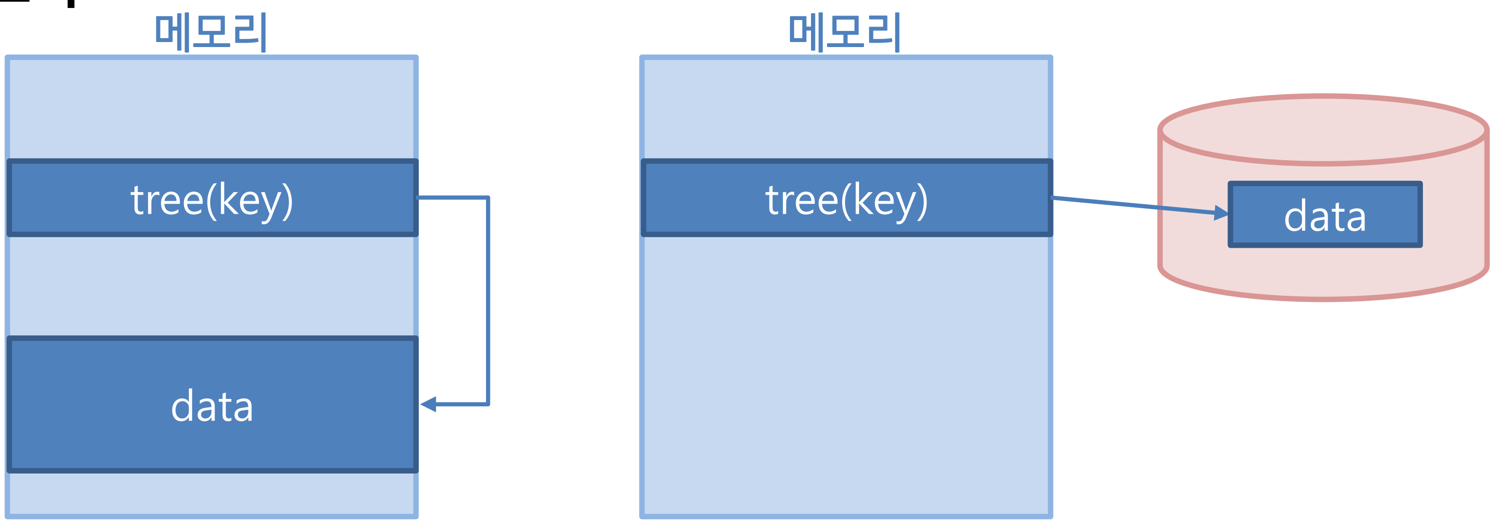
- **생각해볼 문제**
  - CFS에서 힙heap을 사용하지 않은 이유는 무엇인가?



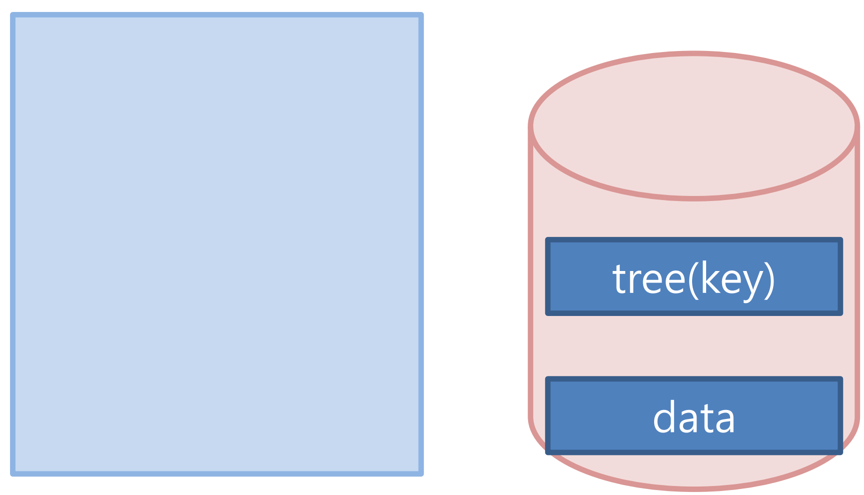
B-트리

내부 트리와 외부 트리

- **내부 트리**

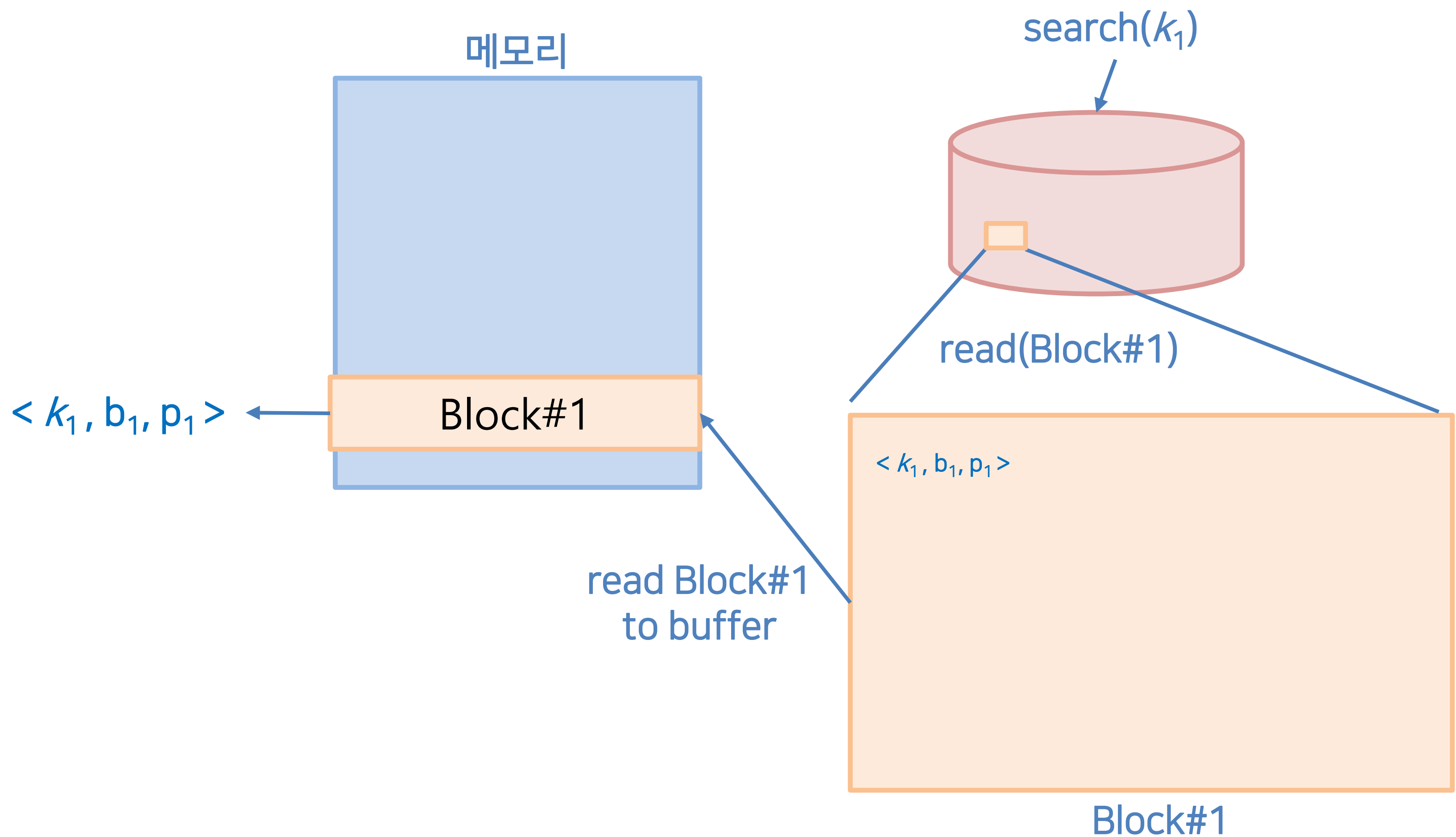


- **외부 트리**
  - 트리가 외부에 저장



외부 트리

- HDD, SSD 등은 일반적으로 블록(페이지) 단위 입출력



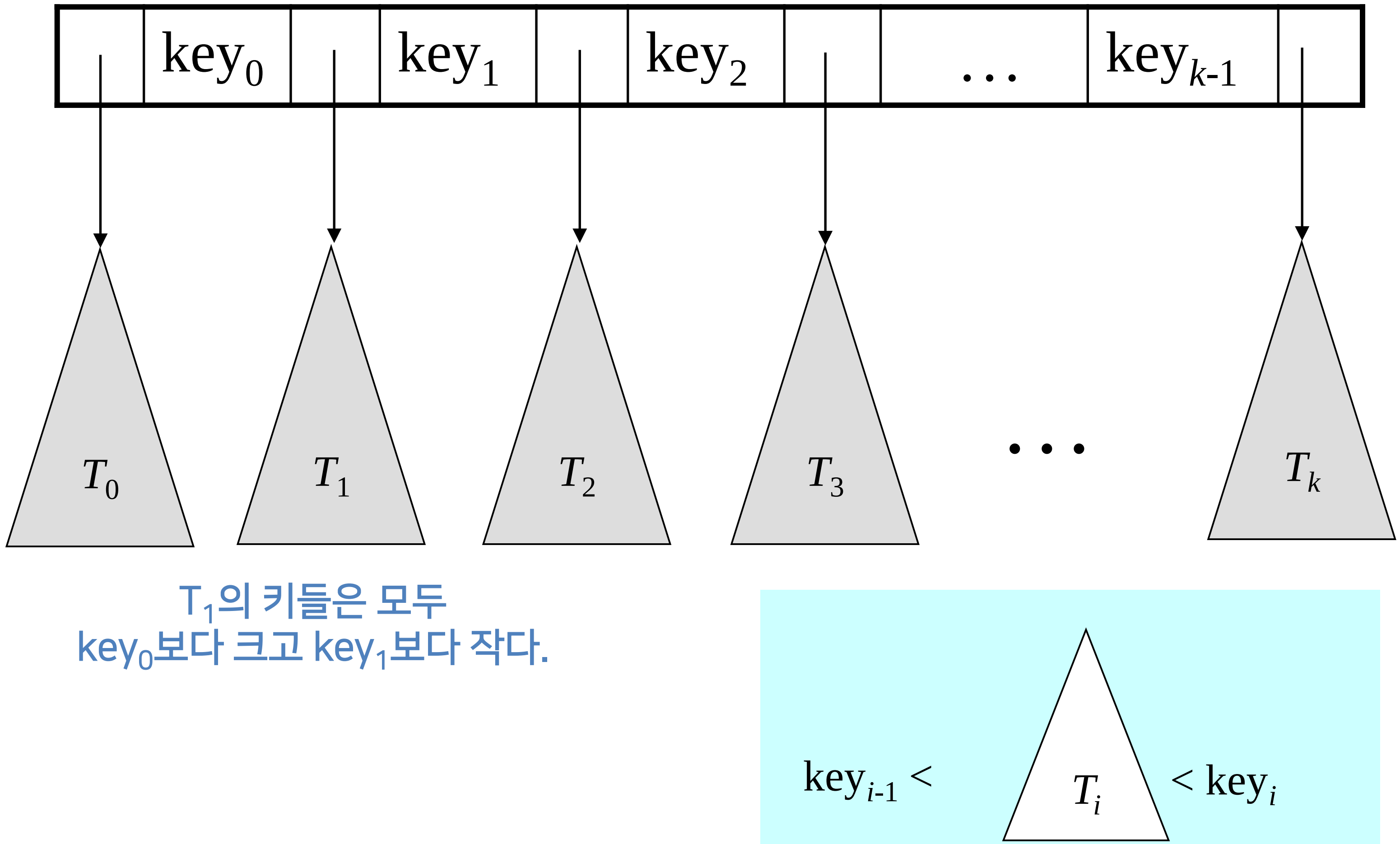
외부 트리

- 디스크의 접근 단위는 블록(페이지)
- 디스크 블록을 한 번 읽거나 쓰는 시간 >>>>>>> 명령어 처리 시간
- 검색트리가 디스크에 저장되어 있다면 **트리의 높이를 최소화**

- **B-트리**
  - 다진검색트리
  - Balanced
  - → 최악의 경우 디스크 접근 횟수를 줄인다.

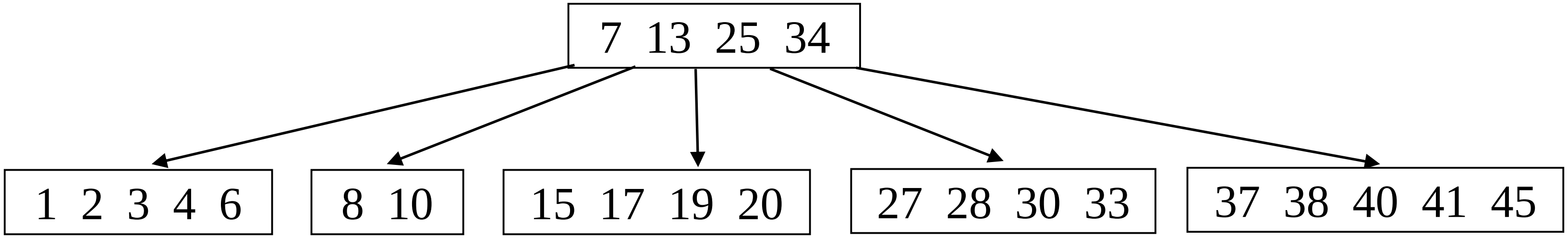
B-트리의 노드 구조

- **다진검색트리**



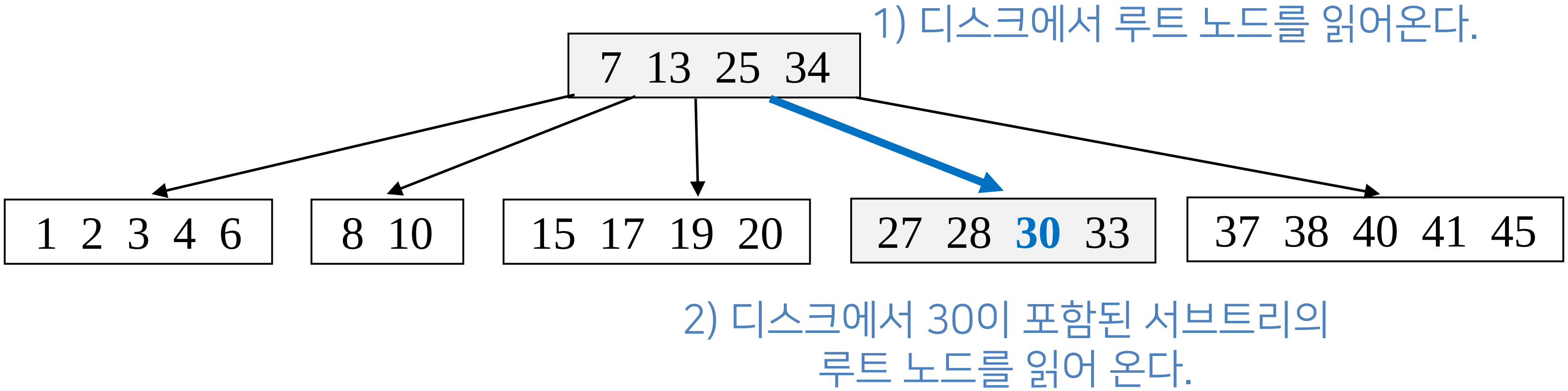
B-트리

- **B-트리는 다음의 성질을 만족한다:**
  - 루트를 제외한 모든 노드는 $\lfloor k/2 \rfloor \sim k$ 개의 키를 갖는다.
    - k는 디스크의 블록 크기에 따라 결정된다.
  - 모든 리프 노드는 같은 깊이를 가진다.



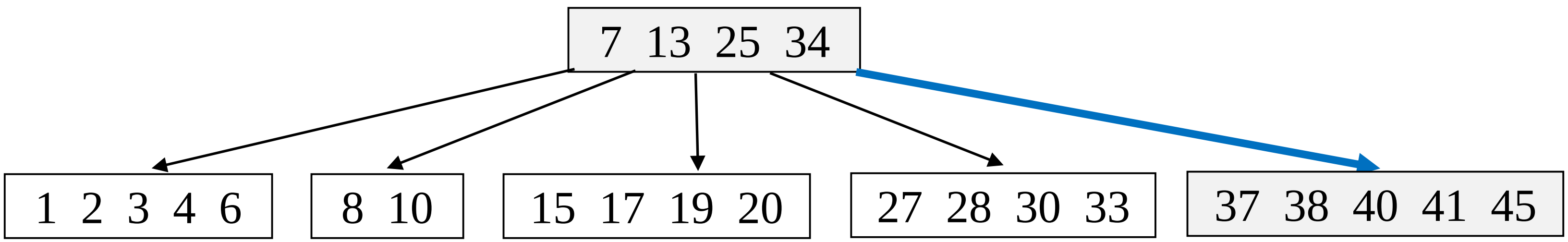
B-트리

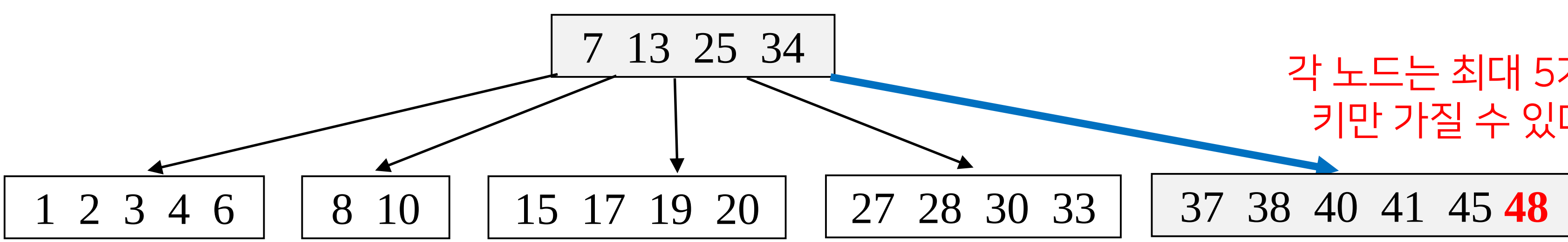
- **30을 검색하는 과정**
  - 트리가 깊어질수록 디스크 접근 횟수가 늘어남



B-트리

- **48을 삽입하면? (k=5)**
  - 디스크 접근 횟수를 줄이려면 **트리 깊이를 최소로 만든다.**
  - 어떻게?

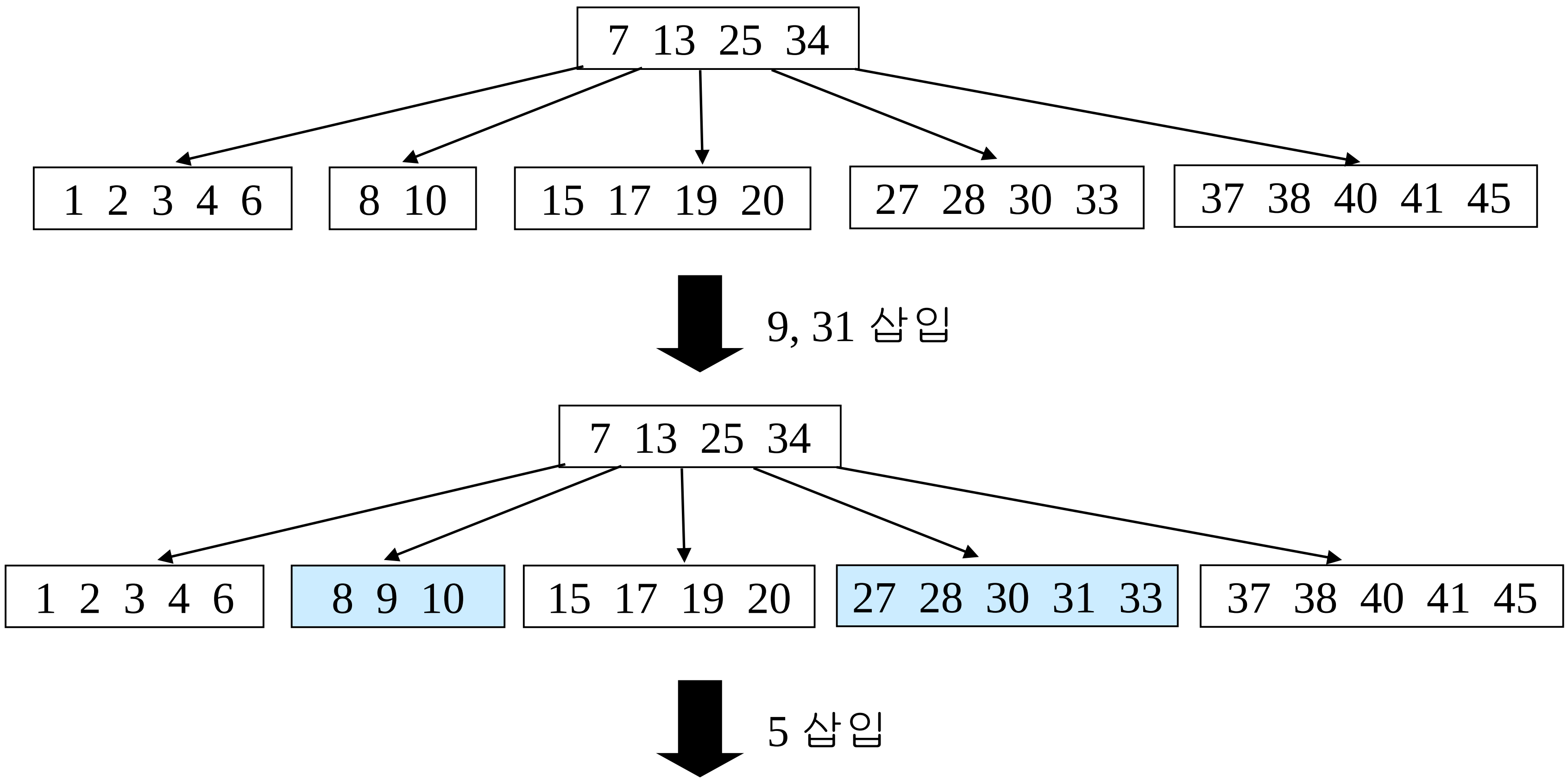




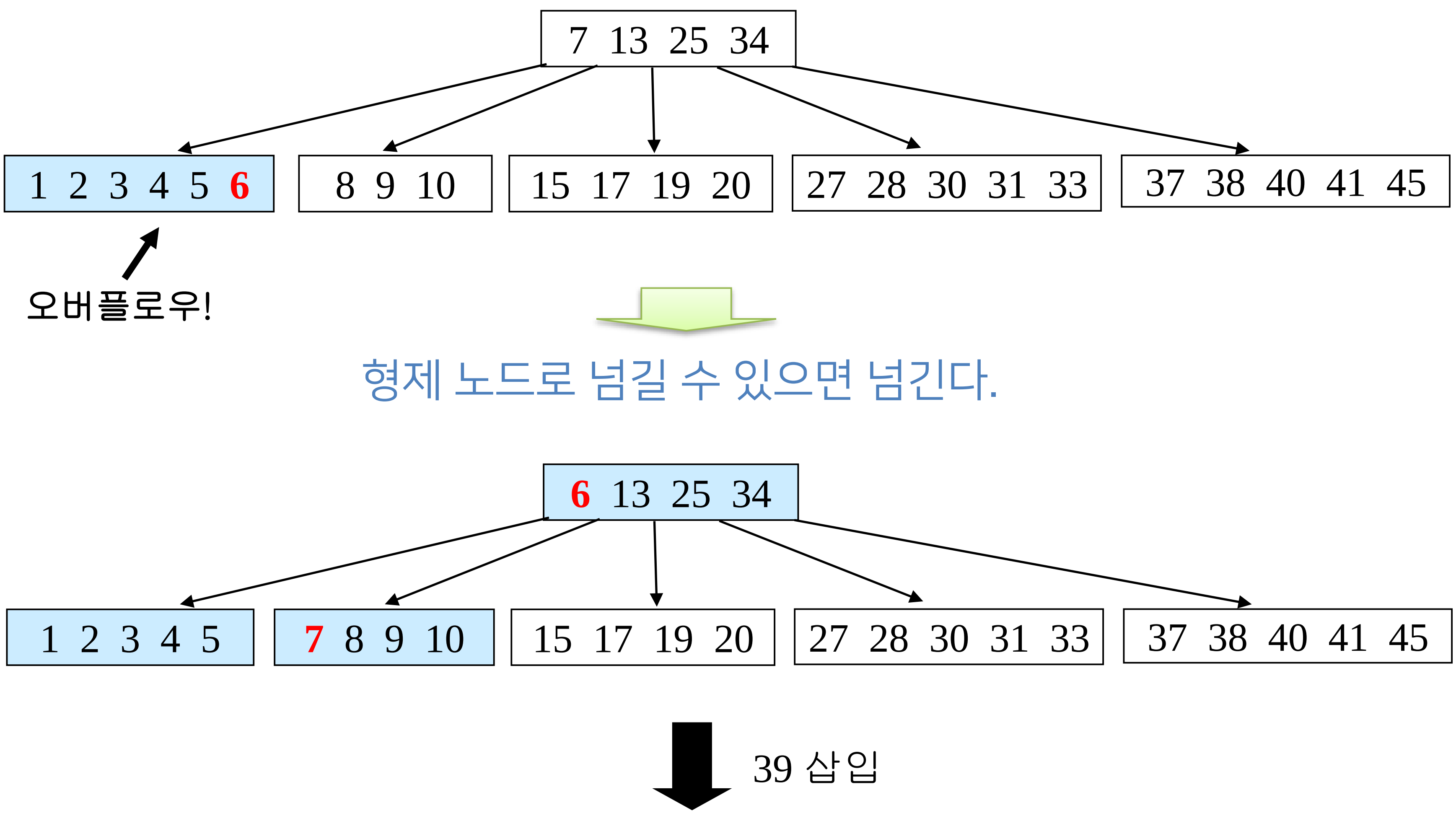
각 노드는 최대 5개의
키만 가질 수 있다!

B-트리에서 삽입

B-트리에서 삽입

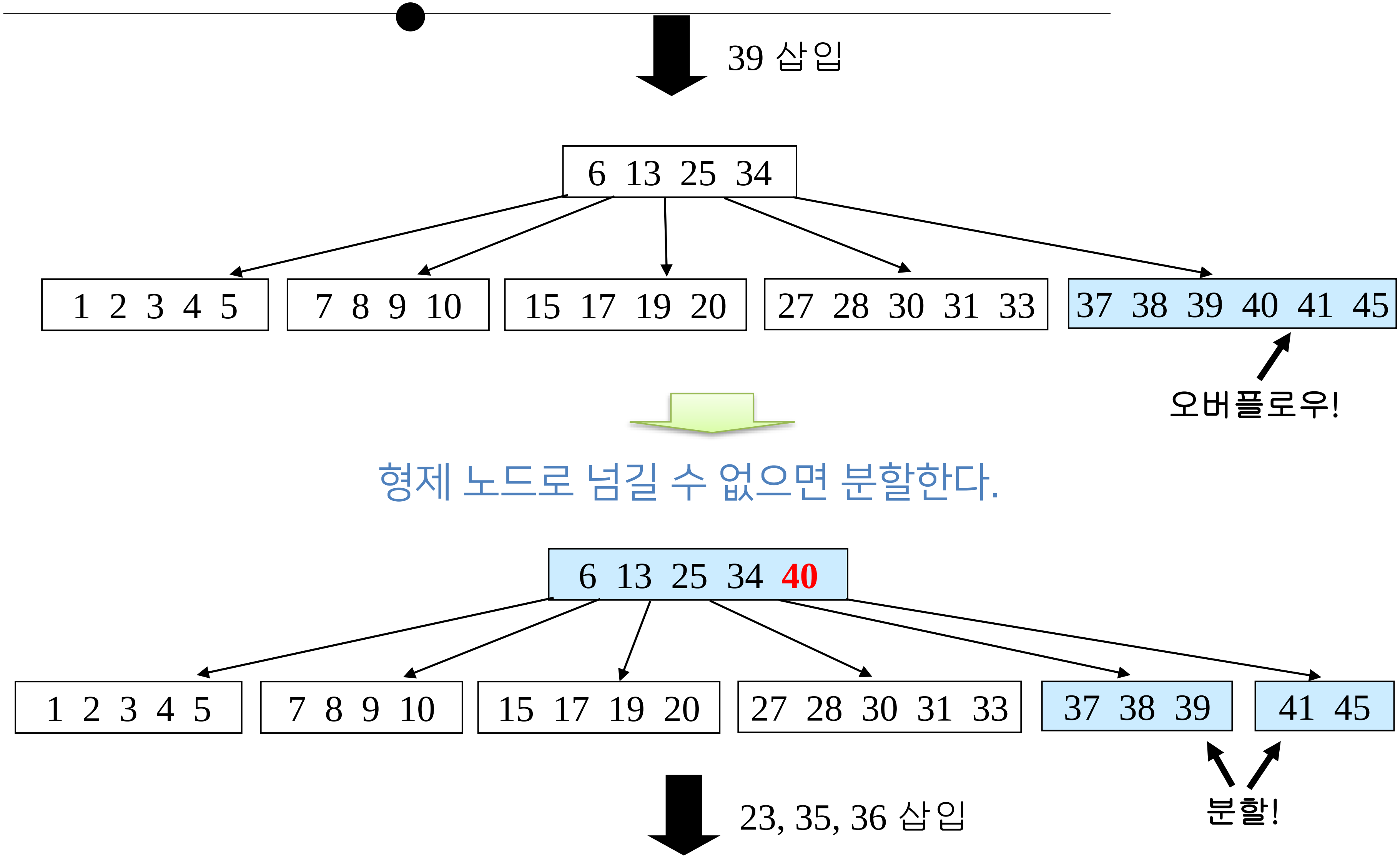

7 13 25 34
1 2 3 4 6
8 10
15 17 19 20
27 28 30 33
37 38 40 41 45
9, 31 삽입
7 13 25 34
1 2 3 4 6
8 9 10
15 17 19 20
27 28 30 31 33
37 38 40 41 45
5 삽입

B-트리에서 삽입



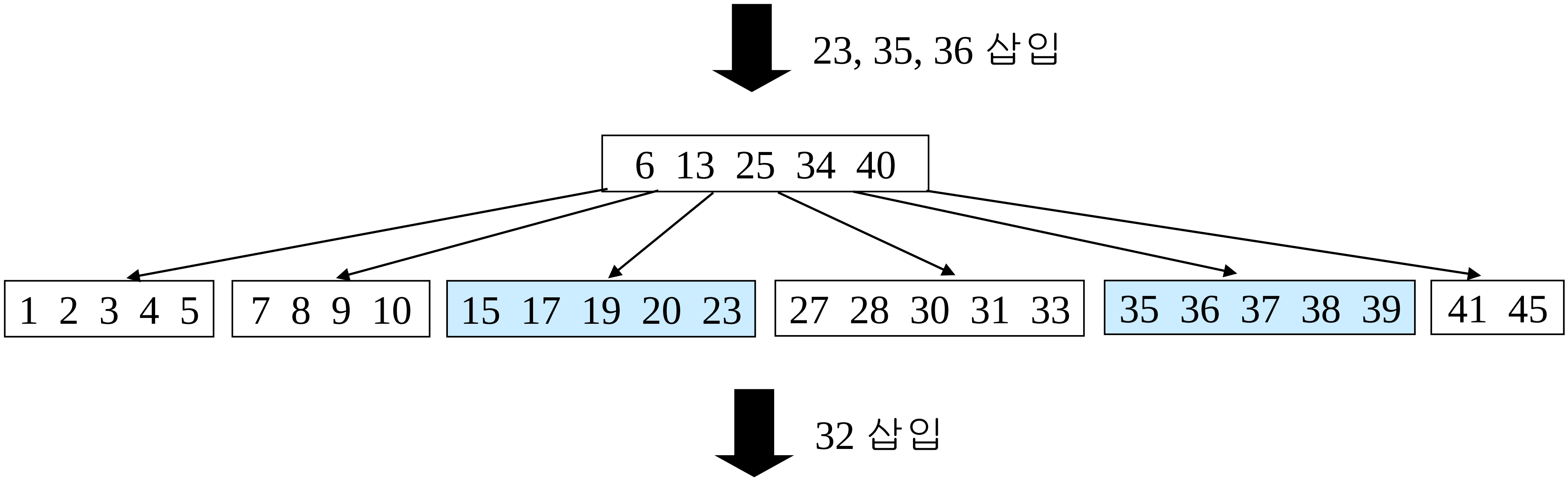
7 13 25 34
1 2 3 4 5 6
8 9 10
15 17 19 20
27 28 30 31 33
37 38 40 41 45
오버플로우!
형제 노드로 넘길 수 있으면 넘긴다.
6 13 25 34
1 2 3 4 5
7 8 9 10
15 17 19 20
27 28 30 31 33
37 38 40 41 45
39 삽입

B-트리에서 삽입

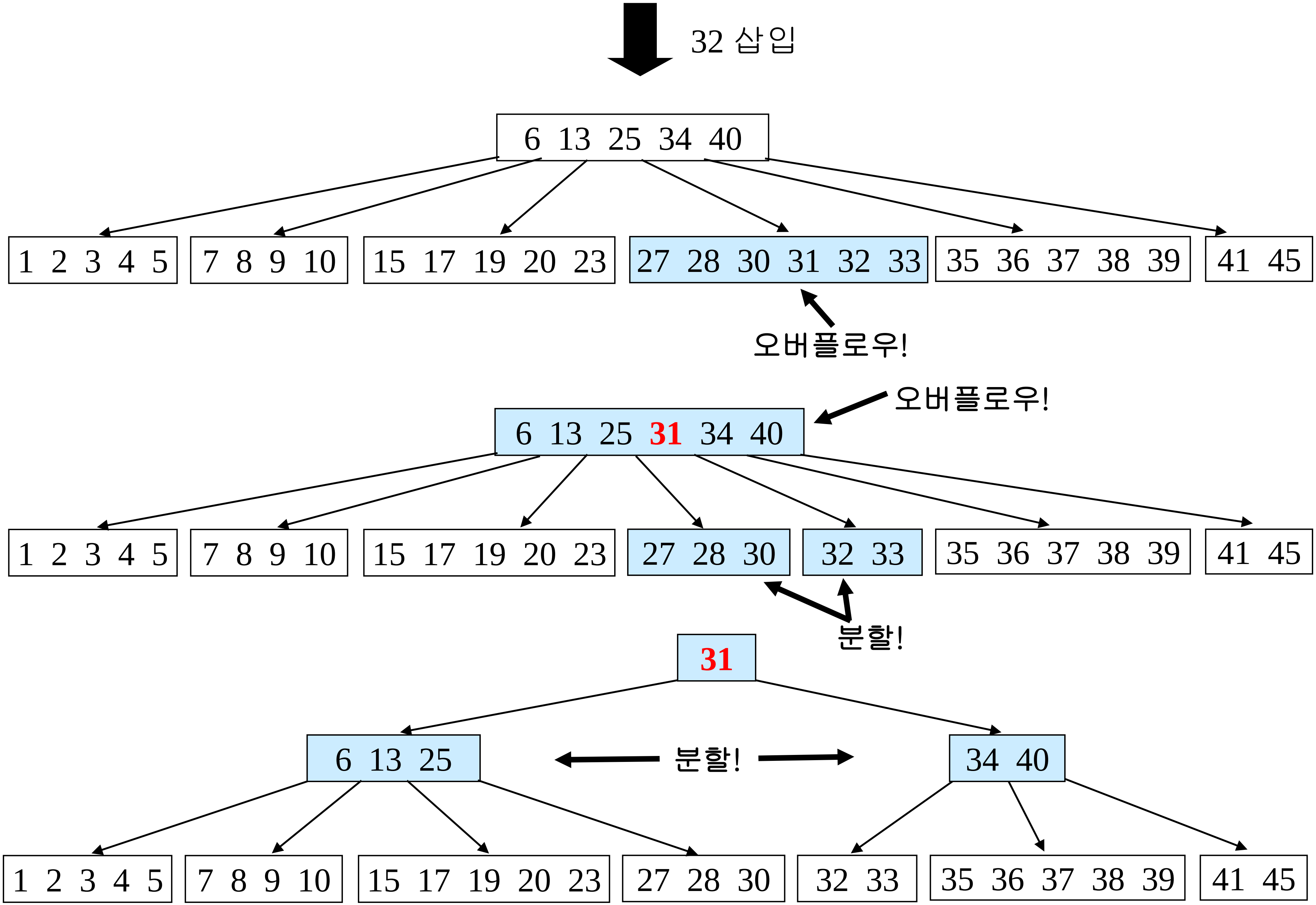


39 삽입
6 13 25 34
1 2 3 4 5
7 8 9 10
15 17 19 20
27 28 30 31 33
37 38 39 40 41 45
오버플로우!
형제 노드로 넘길 수 없으면 분할한다.
6 13 25 34 40
1 2 3 4 5
7 8 9 10
15 17 19 20
27 28 30 31 33
37 38 39
41 45
분할!
23, 35, 36 삽입

B-트리에서 삽입



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
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-트리에서 삽입

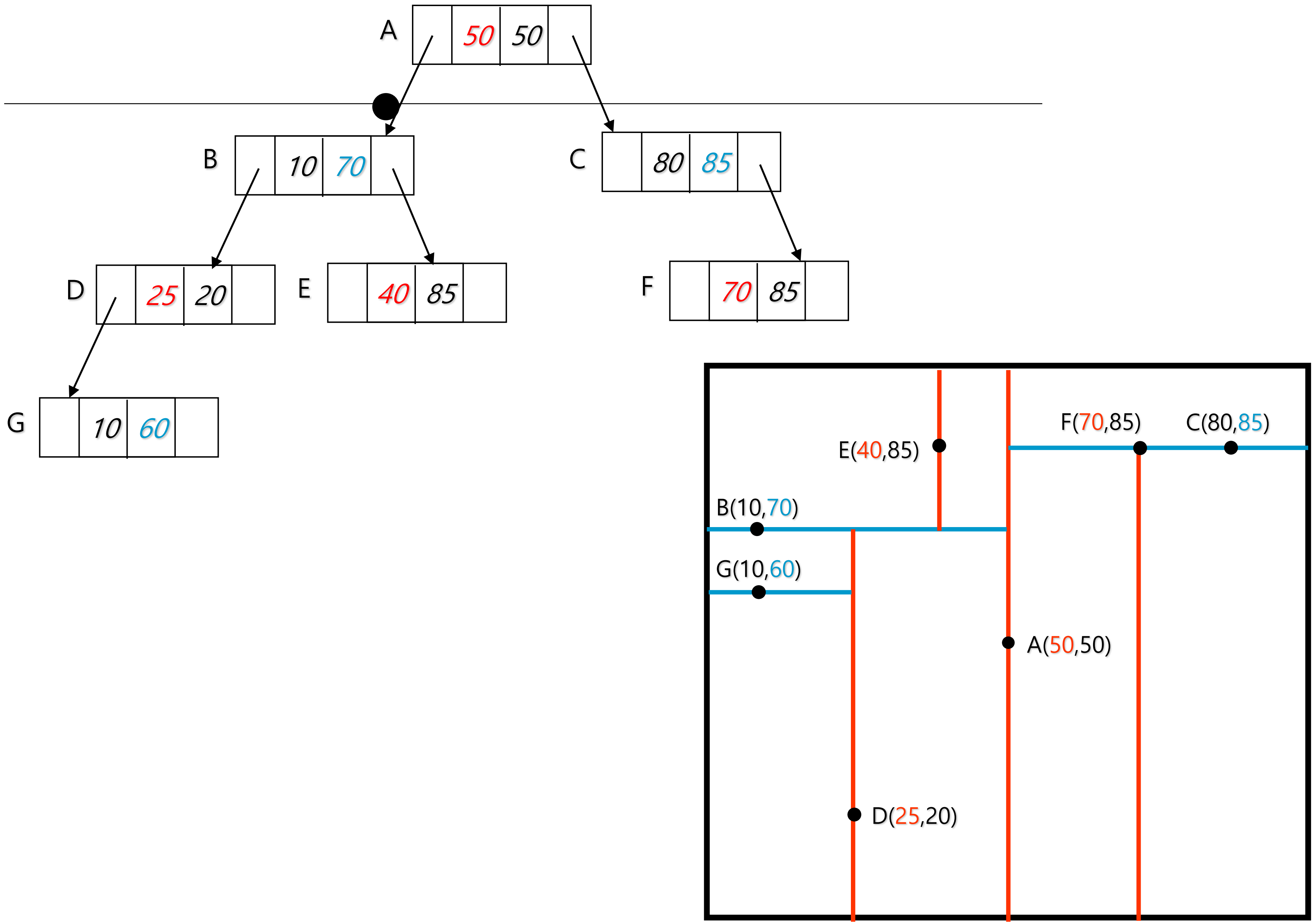
▷ *t* : 트리의 루트 노드
▷ *x* : 삽입하고자 하는 키

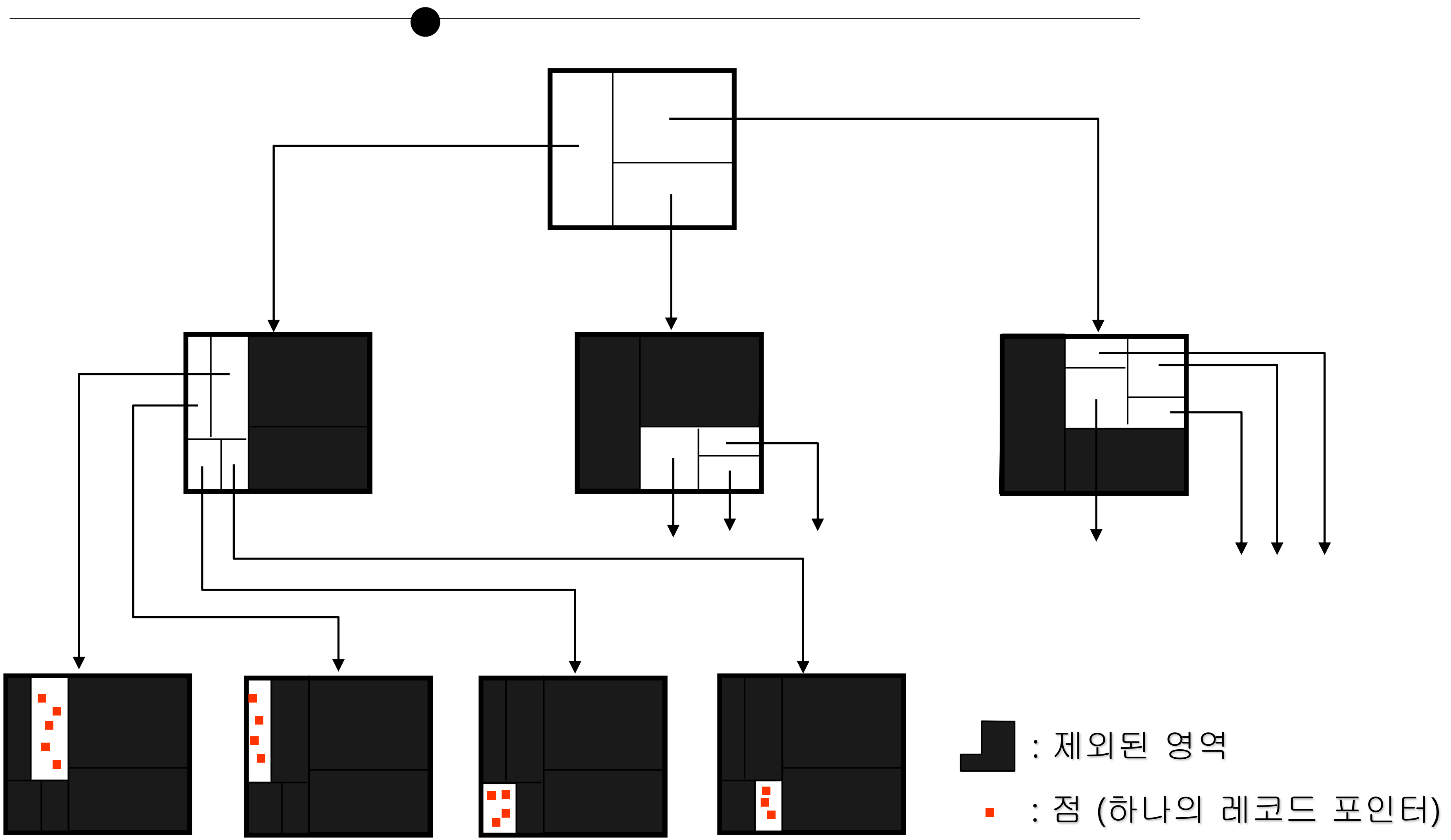
```
BTreeInsert(t, x)
{
        x를 삽입할 리프 노드 r을 찾는다;
        x를 r에 삽입한다;
        if (r에 오버플로우 발생) then clearOverflow(r);
}
clearOverflow(r)
{
    if (r의 형제 노드 중 여유가 있는 노드가 있음) then {r의 남는 키를 넘긴다};
    else {
            r을 둘로 분할하고 가운데 키를 부모 노드로 넘긴다;
            if (부모 노드 p에 오버플로우 발생) then clearOverflow(p);
    }
}
```

다차원 검색 트리

다차원 검색 트리

- **검색키가 두 개 이상의 필드로 이루어진 검색 트리**
  - KD-트리
  - KDB-트리
  - R-트리
  - 그리드 파일


A
50
50
B
10
70
C
80
85
D
25
20
E
40
85
F
70
85
G
10
60
E(40,85)
F(70,85)
C(80,85)
B(10,70)
G(10,60)
A(50,50)
D(25,20)


: 제외된 영역
: 점 (하나의 레코드 포인터)