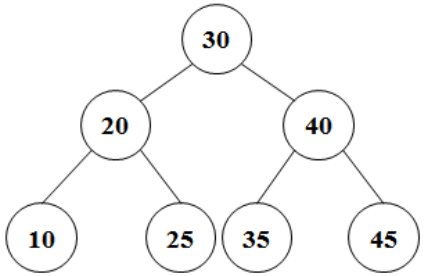
1 이진 탐색 트리

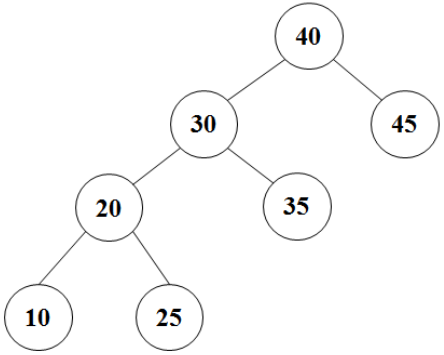
1 이진 탐색 트리

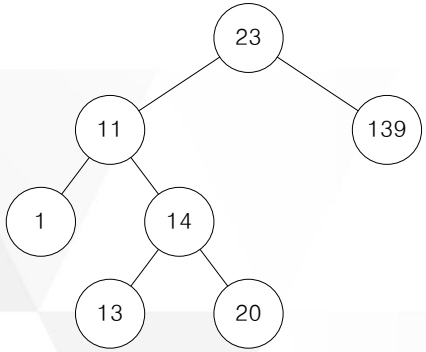
- 탐색 트리의 기본
- 데이터의 삽입, 삭제, 탐색 등이 자주 발생하는 경우에 효율적인 구조
- 이진 트리이면서 같은 값을 갖는 노드가 없어야 함
- 최상위 레벨에 루트 노드가 있고 각 노드는 최대 두 개의 자식을 가짐
- 왼쪽 서브 트리에 있는 모든 데이터는 현재 노드의 값보다 작고, 오른쪽 서브 트리에 있는 모든 노드의 데이터는 현재 노드의 값보다 큼

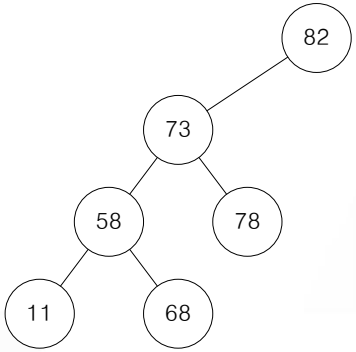
1 이진 탐색 트리

2 이진 탐색 트리의 예



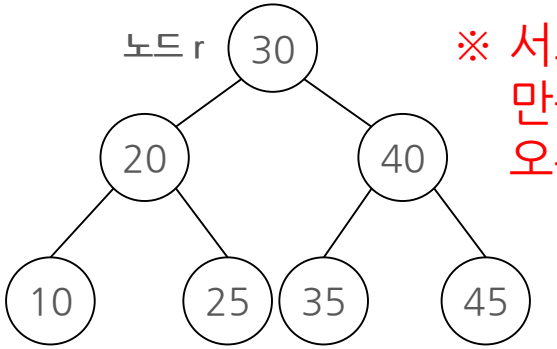




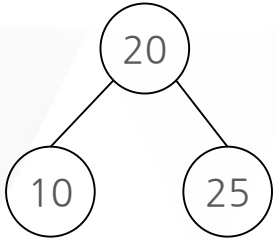


1 이진 탐색 트리

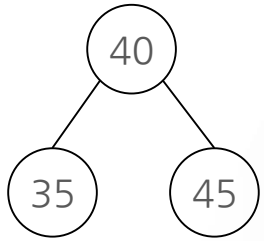
3 서브 트리의 예



(a)



(b) 노드 r의 왼쪽 서브트리



(c) 노드 r의 오른쪽 서브트리

※ 서브 트리도 역시 이진 탐색 트리의 성질을 만족해야 하며 루트의 왼쪽 자식은 루트보다 작고 오른쪽 자식은 루트보다 큼

4 이진 탐색 트리에서의 탐색

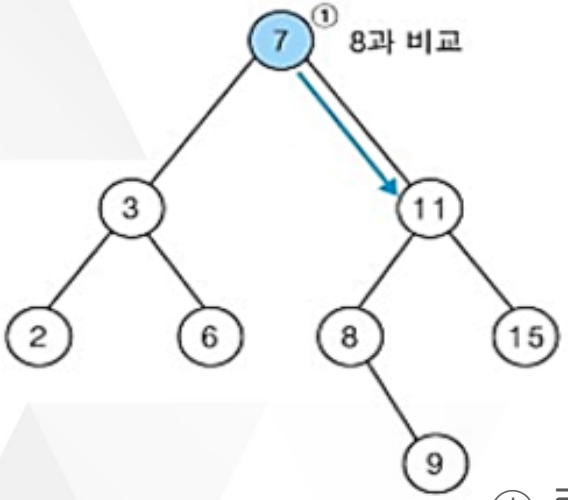
- 데이터 탐색은 루트에서부터 시작됨
- 루트 노드의 데이터와 찾으려는 데이터를 비교하여 루트 노드와 찾으려는 데이터가 같으면 탐색은 성공적으로 종료함

  그렇지 않고 루트 노드의 데이터가 찾으려는 데이터보다 작으면 루트 노드의 오른쪽 서브 트리를 탐색해가고, 루트 로드의 데이터가 찾으려는 데이터보다 크면 루트 노드의 왼쪽 서브 트리를 탐색해 감

4 이진 탐색 트리에서의 탐색

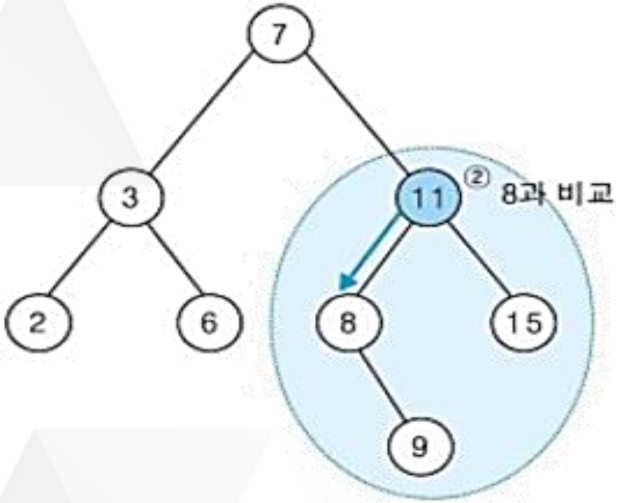
```
treeSearch(t, x)
▷ t: 트리의 루트 노드
▷ x: 검색하고자 하는 키
{
        if (t=NIL or key[t]=x) then return t;
        if (x < key[t])
                then return treeSearch(left[t], x);
                else return treeSearch(right[t], x);
}
```

5 이진 탐색 트리에서의 탐색 예



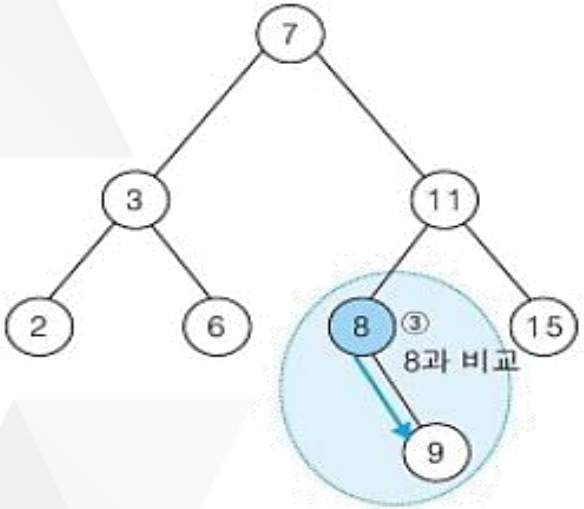
① 루트 노드의 데이터 7이
찾으려는 데이터 8보다 작으므로
오른쪽 서브 트리를 탐색함

5 이진 탐색 트리에서의 탐색 예



② 오른쪽 서브 트리의
루트 노드 데이터 11이
찾으려는 데이터 8보다 크므로
왼쪽 서브 트리를 탐색함

5 이진 탐색 트리에서의 탐색 예



③ 왼쪽 서브 트리의
루트 노드가 8이므로
원하는 노드의 탐색이
성공하여 탐색 완료

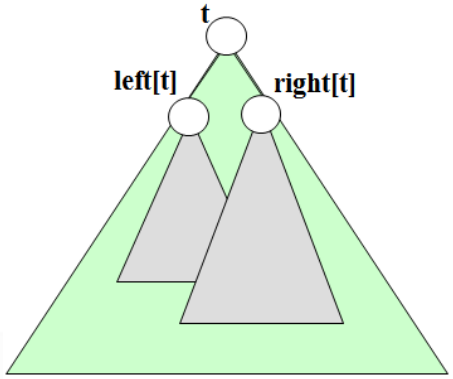
※ 만약 단말 노드에 이를 때까지 같은 데이터를
찾지 못하면 탐색에 실패하는 것임

1 이진 탐색 트리

6 이진 트리 탐색에서 재귀적 관점

- 이진 트리 탐색에서 루트 노드 t에서
  왼쪽 자식 Left[t]로 분기하는 것은 Left[t]가 새로운
  루트가 되었을 뿐 앞의 과정과 똑같은 작업임
- 즉, 루트 노드와 찾으려는 데이터의 대소 비교를 하고
  나면 자신과 성격은 똑같으면서 더 작은 문제를 만남

↳ **재귀적임**



2 이진 탐색 트리에서 삽입

2 이진 탐색 트리에서 삽입

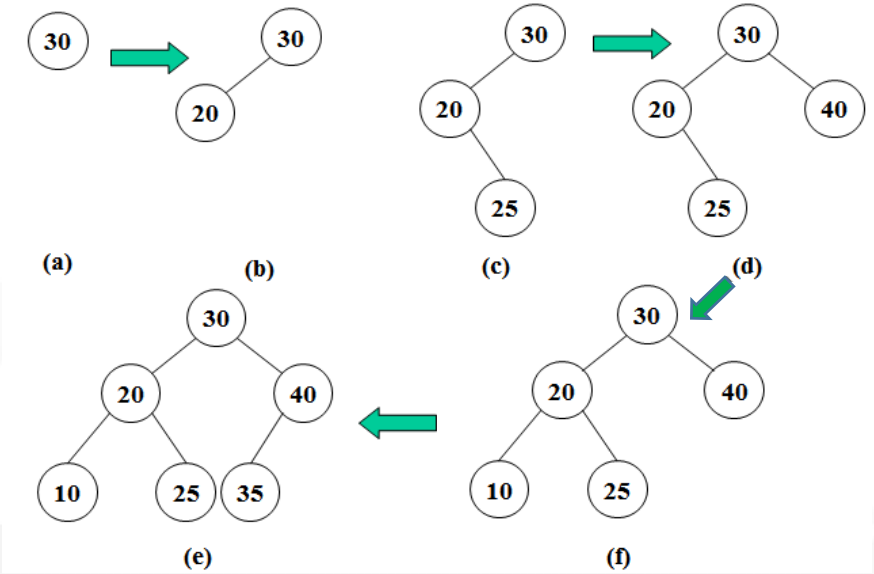
1 데이터 삽입

- 이진 탐색 트리에서의 삽입은 탐색 동작을 통해 이루어짐
- 탐색에 성공하면 삽입은 실패함(이는 삽입하려는 데이터가 이미 존재한다는 의미인데 이진 탐색 트리는 같은 데이터를 갖는 노드가 없어야 함)
- 탐색에 실패하면 삽입을 할 수 있으며 탐색이 종료된 지점의 데이터를 값으로 하는 노드가 삽입됨
- 삽입할 위치는 루트 노드에서부터 시작되며 삽입할 노드의 데이터가 비교하는 노드의 데이터보다 작으면 왼쪽 서브 트리로 진행하고 크면 오른쪽 서브 트리로 진행함

2 이진 탐색 트리에서 삽입

1 데이터 삽입

- 예) 데이터가 30, 20, 25, 40, 10, 35의 순서로 원소가 삽입되는 경우 이진 탐색 트리가 만들어지는 과정



2 이진 탐색 트리에서 삽입

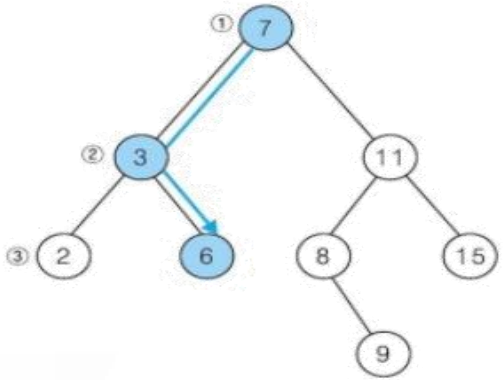
1 데이터 삽입

```
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;}
}
```

2 이진 탐색 트리에서 삽입

2 데이터 삽입의 예

◆ 예) 다음의 트리에서 데이터가 5인 노드를 삽입하는 과정



풀이)

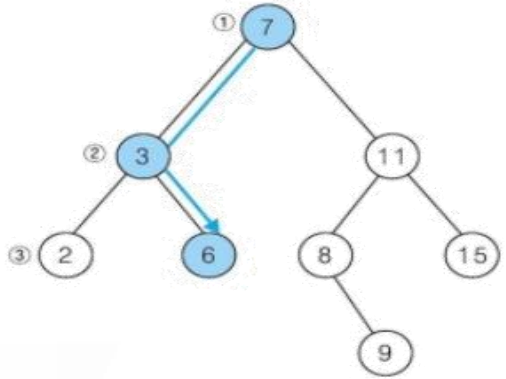
① 삽입하려는 데이터 5가 루트 노드의
데이터 7보다 작으므로 왼쪽 서브 트리로 진행

② 왼쪽 서브 트리의 루트 노드인 데이터 3보다
삽입할 데이터 5가 크므로 오른쪽 서브 트리로 진행

2 이진 탐색 트리에서 삽입

2 데이터 삽입의 예

◆ 예) 다음의 트리에서 데이터가 5인 노드를 삽입하는 과정



풀이)

③ 오른쪽 서브 트리의 루트 노드는 6이지만 단말 노드이므로 삽입할 위치를 찾는 동작은 더 이상 진행하지 않음(만약 같은 데이터를 가진 노드를 발견하면 삽입 실패로 종료됨)

2 이진 탐색 트리에서 삽입

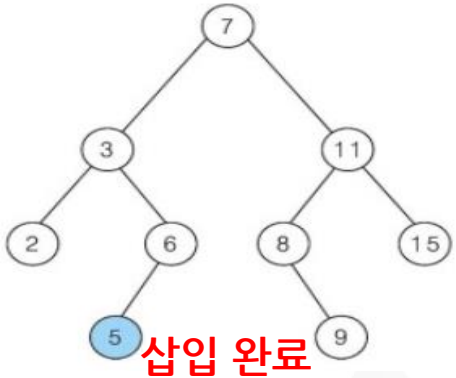
2 데이터 삽입의 예

◆ 예) 다음의 트리에서 데이터가 5인 노드를 삽입하는 과정

풀이)

④ 단말 노드의 데이터 6보다 삽입할 데이터 5가 작으므로 단말 노드의 왼쪽 자식 노드로 삽입 완료

※ 만약 삽입할 노드의 데이터가 크면 오른쪽 자식 노드로 삽입



2 이진 탐색 트리에서 삽입

3 이진 탐색 트리의 특징

- 이진 탐색 트리는 좌우의 균형이 잘 잡힌 탐색 트리인데 균형이 잘 맞으면 탐색의 효율이 높음
- 탐색 트리에서 효율은 트리의 깊이와 밀접한 관계가 있음

↳ 트리의 모양이 극단적으로 왼쪽이나 오른쪽으로 치우쳐 있는 경우 탐색의 효율이 나쁨(사향 이진 트리)

2 이진 탐색 트리에서 삽입

3 이진 탐색 트리의 특징

- 사향 이진 트리(Skewed Binary Tree)

  예) 데이터가 10, 20, 25, 30, 40, 45 의 순서로 원소가 삽입될 경우에 만들어지는 트리의 모양은 극단적으로 오른쪽으로 치우쳐 있음(오른쪽 사향 이진 트리)

  → 이 트리에서 45를 검색할 경우 10, 20, 25, 30, 40, 45 의 순서로 트리의 모든 원소와 비교해야 함

  → 40을 검색할 경우에는 10, 20, 25, 30, 40 의 순서로 다섯번의 비교가 필요함(검색 효율이 나쁨)

3 이진 탐색 트리에서 삭제

3 이진 탐색 트리에서 삭제

1 데이터 삭제

- 이진 탐색 트리에서 노드를 삭제하는 동작은 삭제할 노드의 위치에 따라 다음과 같이 세 가지로 구분됨
- 세 가지 경우에 따라 다르게 처리함

3 이진 탐색 트리에서 삭제

1 데이터 삭제

세 가지 경우

- t : 트리의 루트 노드
  r : 삭제하고자 하는 노드

  ① Case 1 : r이 리프 노드인 경우
  ② Case 2 : r의 자식 노드가 하나인 경우
  ③ Case 3 : r의 자식 노드가 두개인 경우

3 이진 탐색 트리에서 삭제

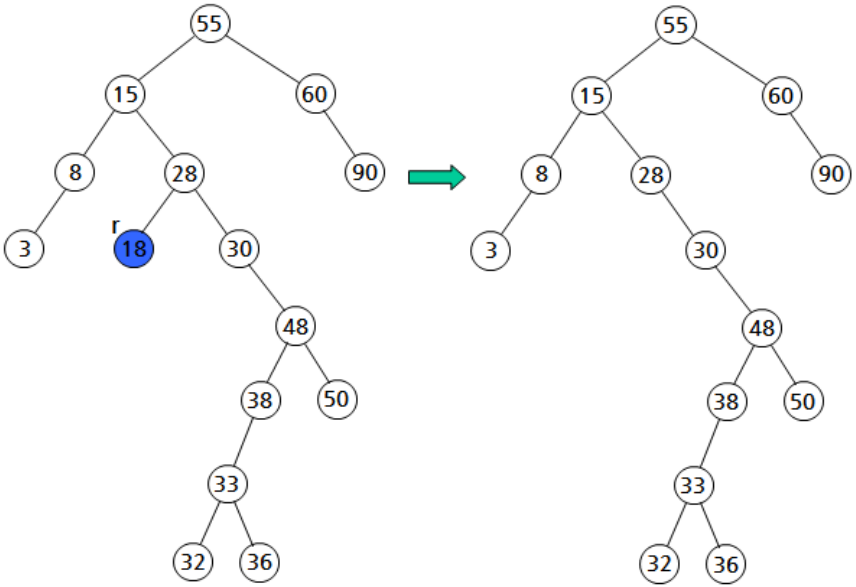
1 데이터 삭제

```
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;
}
```

3 이진 탐색 트리에서 삭제

2 데이터 삭제의 예

- Case 1 : r이 리프 노드인 경우

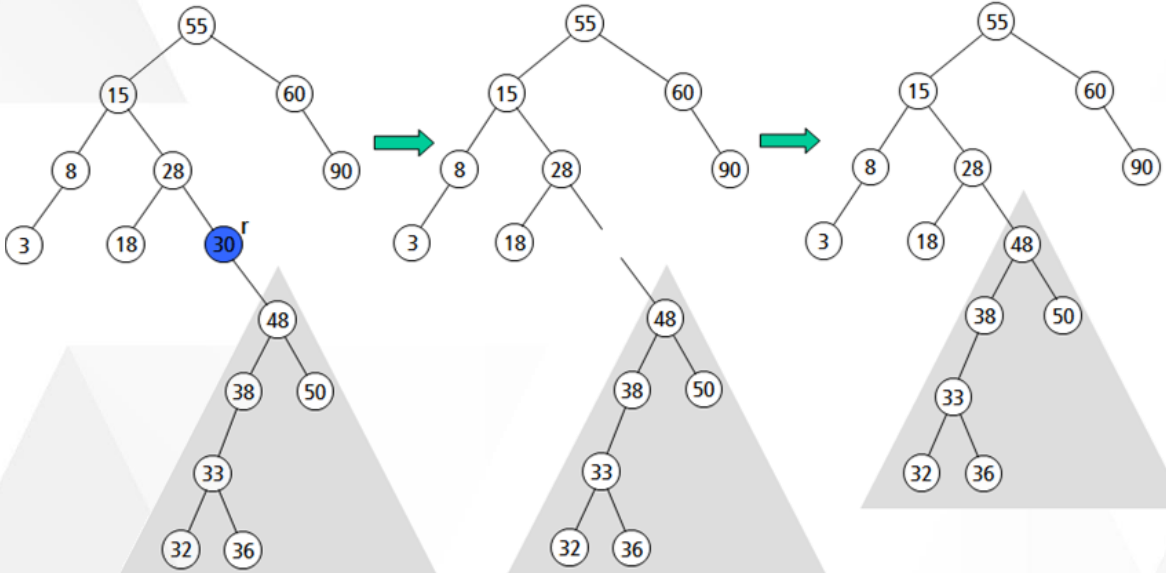


(a) r의 자식이 없음 (b) 단순히 r을 제거한다

3 이진 탐색 트리에서 삭제

2 데이터 삭제의 예

- Case 2 : r의 자식 노드가 하나인 경우



(a) r의 자식이 하나뿐임 (b) r을 제거 (c) r자리에 r의 자식을 놓는다

3 이진 탐색 트리에서 삭제

2 데이터 삭제의 예

- Case 3 : r의 자식 노드가 두개인 경우
  - 자식이 두개인 경우가 조금 복잡함
  - r의 부모가 r을 가리키던 포인터는 하나인데 두 자식 중 하나만 이 포인터를 사용하고 나머지 하나는 연결을 끊을 수도 없음 (r의 주변 구조는 그대로 유지해야 함)

3 이진 탐색 트리에서 삭제

2 데이터 삭제의 예

- Case 3 : r의 자식 노드가 두개인 경우

〈방법〉

① 우선 r 자리에 옮겨놓아도 이진 탐색 트리의 성질을 전혀 깨지 않는 원소를 찾음

3 이진 탐색 트리에서 삭제

2 데이터 삭제의 예

◆ Case 3 : r의 자식 노드가 두개인 경우

〈방법〉

② 그런 원소는 왼쪽 서브 트리의 원소들보다 크고
오른쪽 서브 트리의 원소들보다 작아야 함

↳ 트리 전체에는 딱 2개가 있음

- r의 왼쪽 서브 트리에서 가장 큰 원소
(크기 순으로 r의 직전 원소)
- r의 오른쪽 서브 트리에서 가장 작은 원소
(크기 순으로 r의 직후 원소)

3 이진 탐색 트리에서 삭제

2 데이터 삭제의 예

- Case 3 : r의 자식 노드가 두개인 경우

〈방법〉

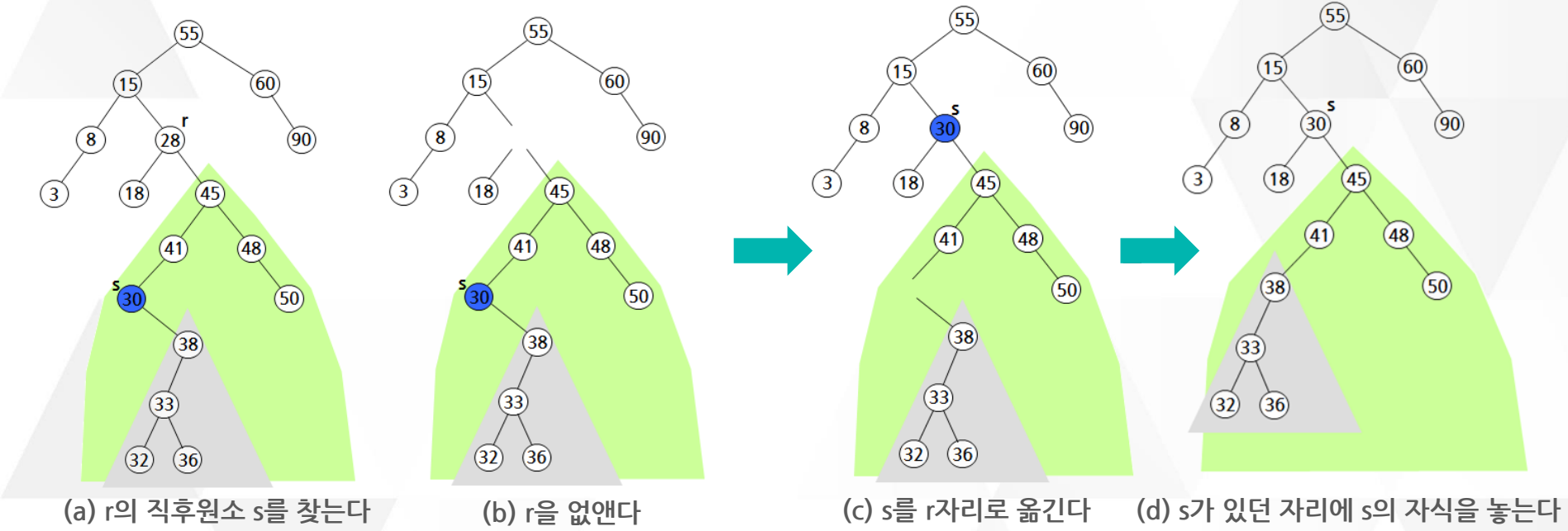
③ 둘 중 하나를 택해 키를 r 자리로 옮김
(여기서는 직후 원소를 택함)

④ 그런 다음 직후 원소가 들어 있던 노드를 삭제함

3 이진 탐색 트리에서 삭제

2 데이터 삭제의 예

◆ Case 3 : r의 자식 노드가 두개인 경우



(a) r의 직후원소 s를 찾는다 (b) r을 없앤다 (c) s를 r자리로 옮긴다 (d) s가 있던 자리에 s의 자식을 놓는다

3 이진 탐색 트리에서 삭제

2 데이터 삭제의 예

삭제 작업의 수행시간

- Case1과 Case2는 상수 시간이 듦
- Case3는 노드 r의 직후 원소를 찾는데 최악의 경우 트리의 높이에 비례하는 시간이 듦

↳ 삭제 작업을 위한 최악의 시간은 트리의 높이에 따라 O(log n)과 O(n) 사이에 결정됨

3 이진 탐색 트리에서 삭제

2 데이터 삭제의 예

삭제 작업의 수행시간

- 이진 탐색 트리에서의 탐색, 삽입, 삭제 연산의 시간 복잡도는 트리의 높이를 h라고 했을 때 O(h)가 됨
- 따라서 n개의 노드를 가지는 이진 탐색 트리의 경우, 균형 잡힌 이진 트리의 높이는 log n 이므로 이진 탐색 트리 연산의 평균적인 경우의 시간 복잡도는 O(log n)임