선수 지식 - 자료구조
이진 탐색 트리

이진 탐색 트리 | 다양한 알고리즘의 기본이 되는 자료구조 이해하기

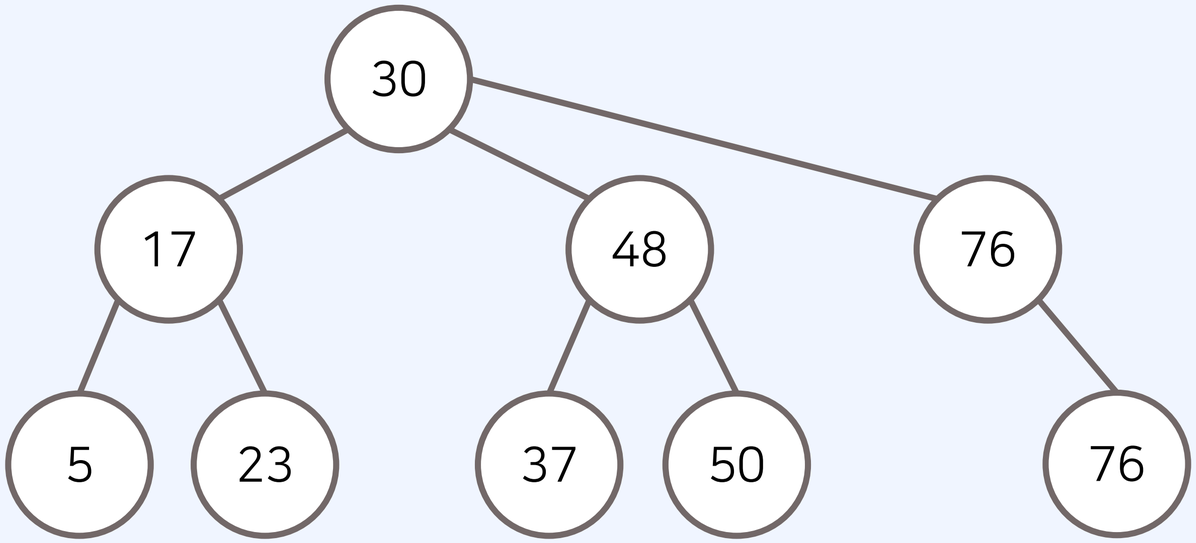
강사 나동빈

선수 지식 – 자료구조

이진 탐색 트리

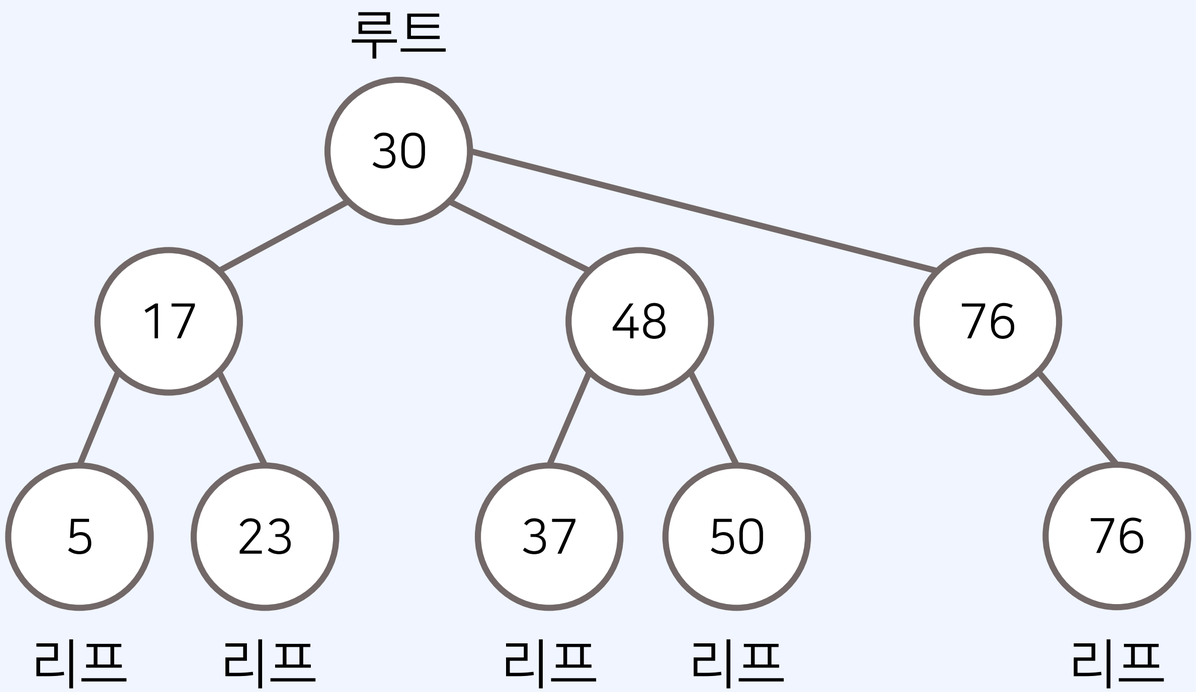
트리(Tree)

- 트리는 가계도와 같이 계층적인 구조를 표현할 때 사용할 수 있는 자료구조다.
- 나무(tree)의 형태를 뒤집은 것과 같이 생겼다.



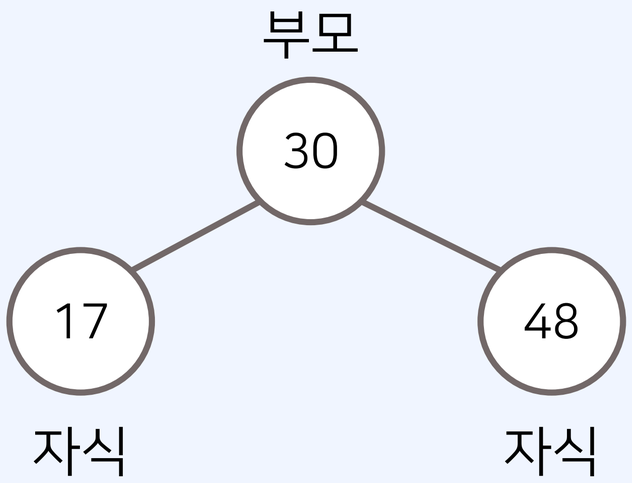
트리 용어 정리

- 루트 노드(root node): 부모가 없는 최상위 노드
- 단말 노드(leaf node): 자식이 없는 노드



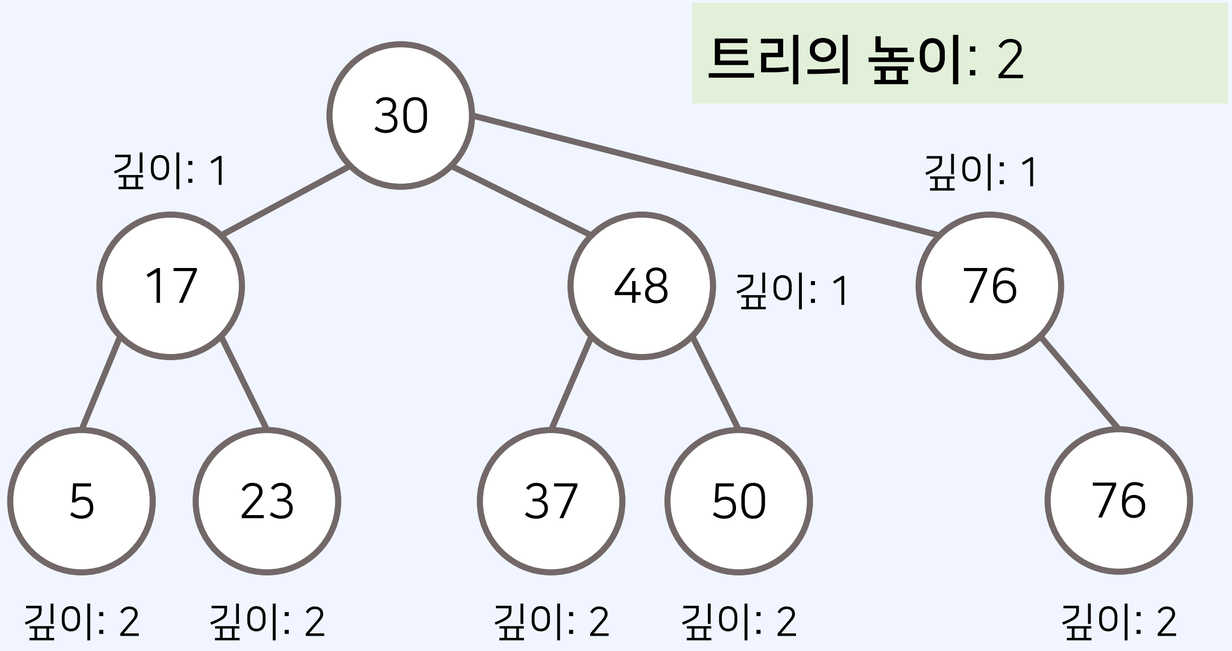
트리 용어 정리

- **트리(tree)**에서는 부모와 자식 관계가 성립한다.
- **형제 관계**: 17을 값으로 가지는 노드와 48을 가지는 노드 사이의 관계



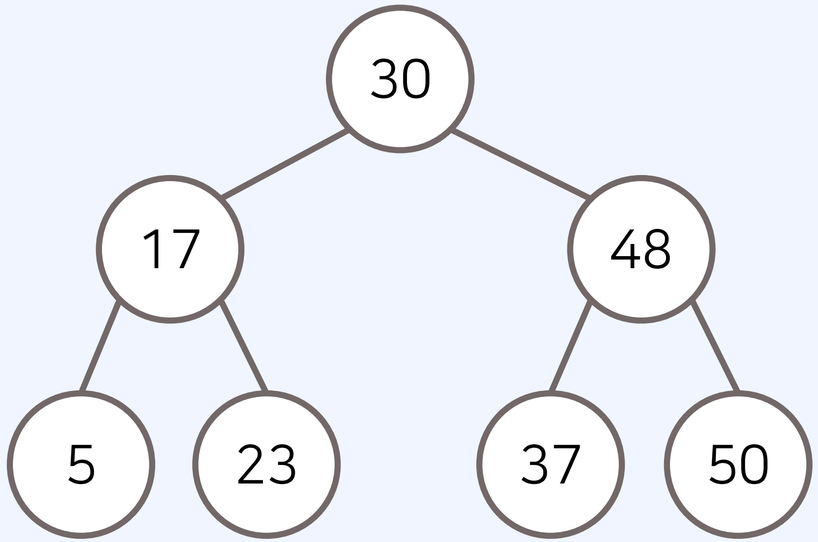
트리 용어 정리

- **깊이(depth)**: 루트 노드에서의 길이(length)
- 이때, 길이란 출발 노드에서 목적지 노드까지 거쳐야 하는 간선의 수를 의미한다.
- 트리의 높이(height)은 루트 노드에서 가장 깊은 노드까지의 길이를 의미한다.



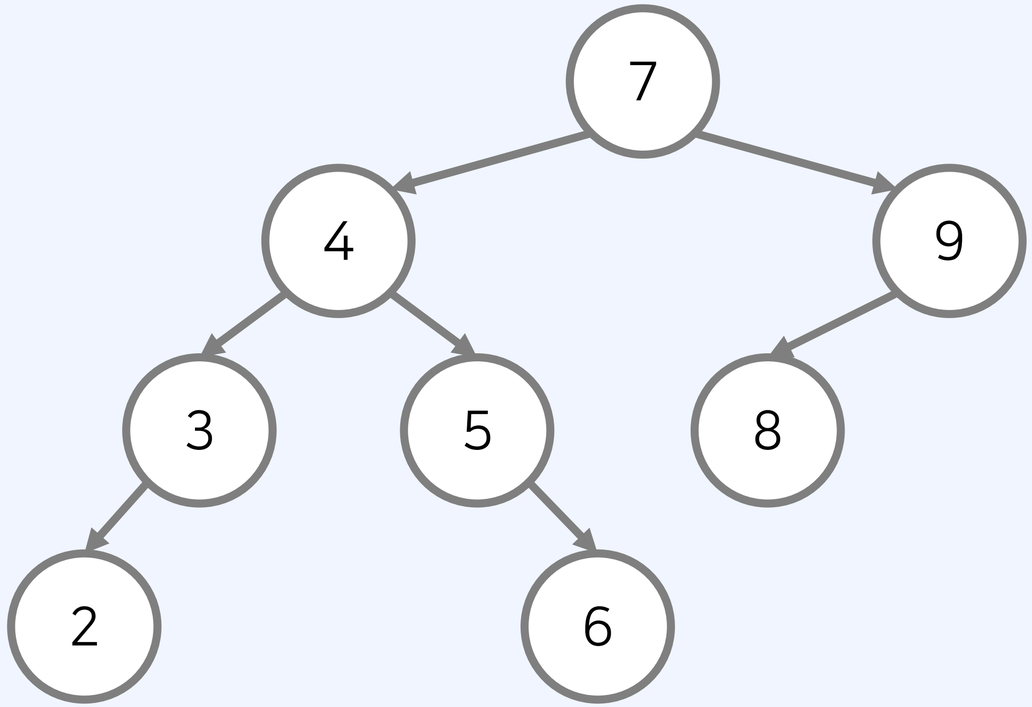
이진 트리(Binary Tree)

- 이진 트리는 최대 2개의 자식을 가질 수 있는 트리를 말한다.



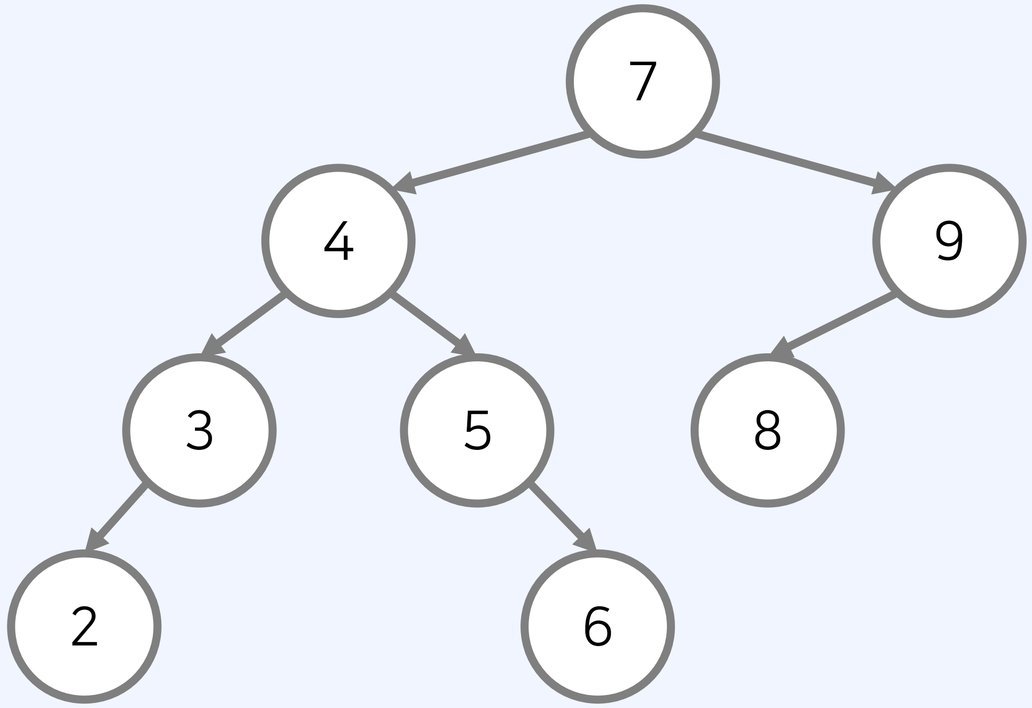
이진 탐색 트리(Binary Search Tree)

- **다수의 데이터**를 관리(조회, 저장, 삭제)하기 위한 가장 기본적인 자료구조 중 하나다.



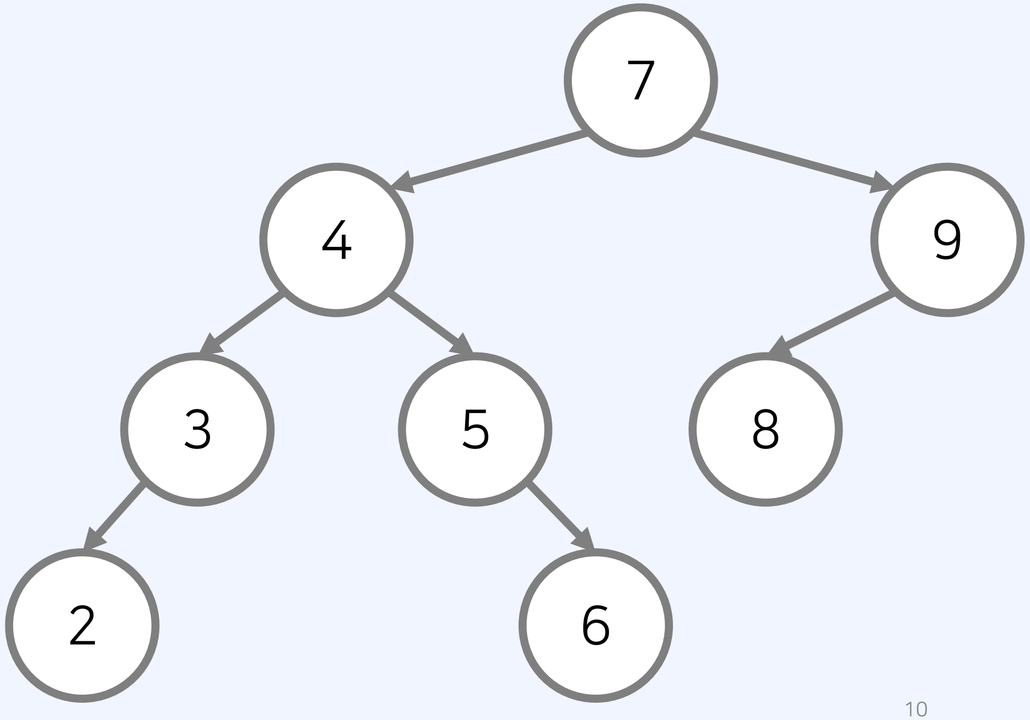
이진 탐색 트리(Binary Search Tree)

- 이진 탐색 트리의 성질: **왼쪽 자식 노드 < 부모 노드 < 오른쪽 자식 노드**



이진 탐색 트리(Binary Search Tree)의 성질

- 특정한 노드의 키(key) 값보다 그 **왼쪽 자식 노드**의 키(key) 값이 더 작다.
- 특정한 노드의 키(key) 값보다 그 **오른쪽 자식 노드**의 키(key) 값이 더 크다.
- 특정한 노드의 왼쪽 서브 트리, 오른쪽 서브 트리 모두 이진 탐색 트리다.



이진 탐색 트리 – 삽입 연산

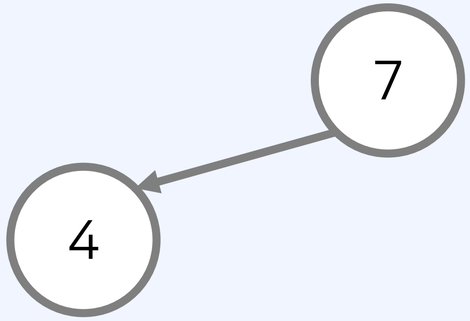
- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삽입할 위치를 찾는다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 삽입
- **삽입할 노드 목록 예시**: [7, 4, 5, 9, 6, 3, 2, 8]

이진 탐색 트리 – 삽입 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삽입할 위치를 찾는다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 삽입
- **삽입할 노드 목록 예시**: [7, 4, 5, 9, 6, 3, 2, 8]

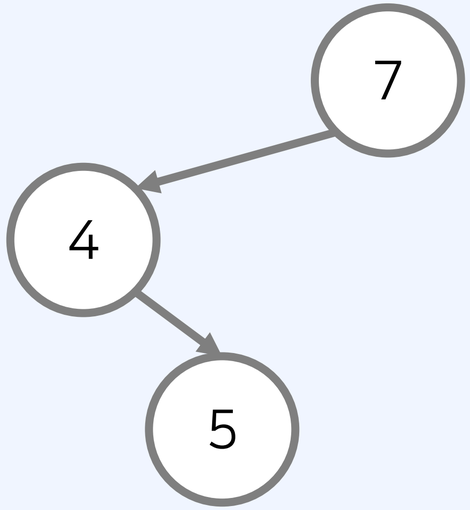
이진 탐색 트리 – 삽입 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삽입할 위치를 찾는다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 삽입
- **삽입할 노드 목록 예시**: [7, 4, 5, 9, 6, 3, 2, 8]



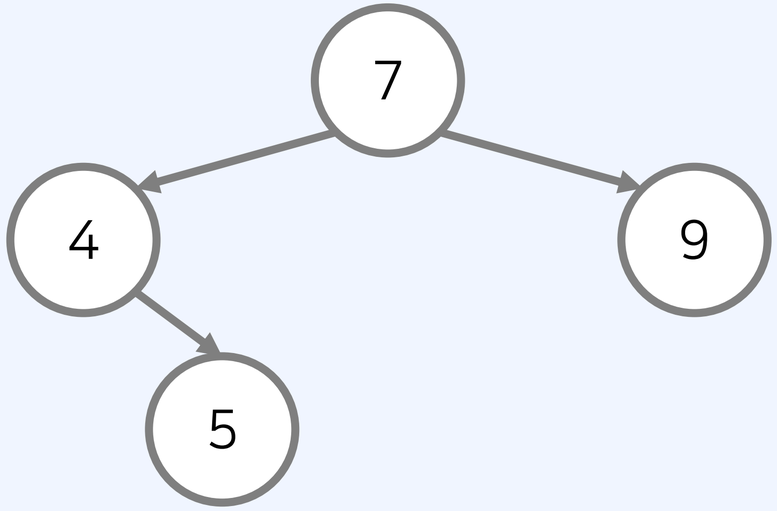
이진 탐색 트리 – 삽입 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삽입할 위치를 찾는다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 삽입
- **삽입할 노드 목록 예시**: [7, 4, 5, 9, 6, 3, 2, 8]



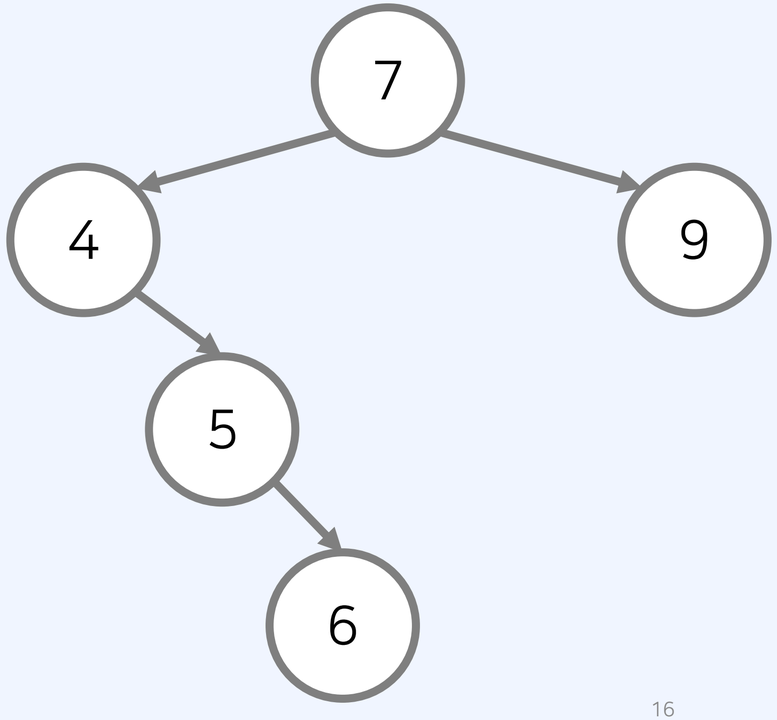
이진 탐색 트리 – 삽입 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삽입할 위치를 찾는다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 삽입
- **삽입할 노드 목록 예시**: [7, 4, 5, 9, 6, 3, 2, 8]



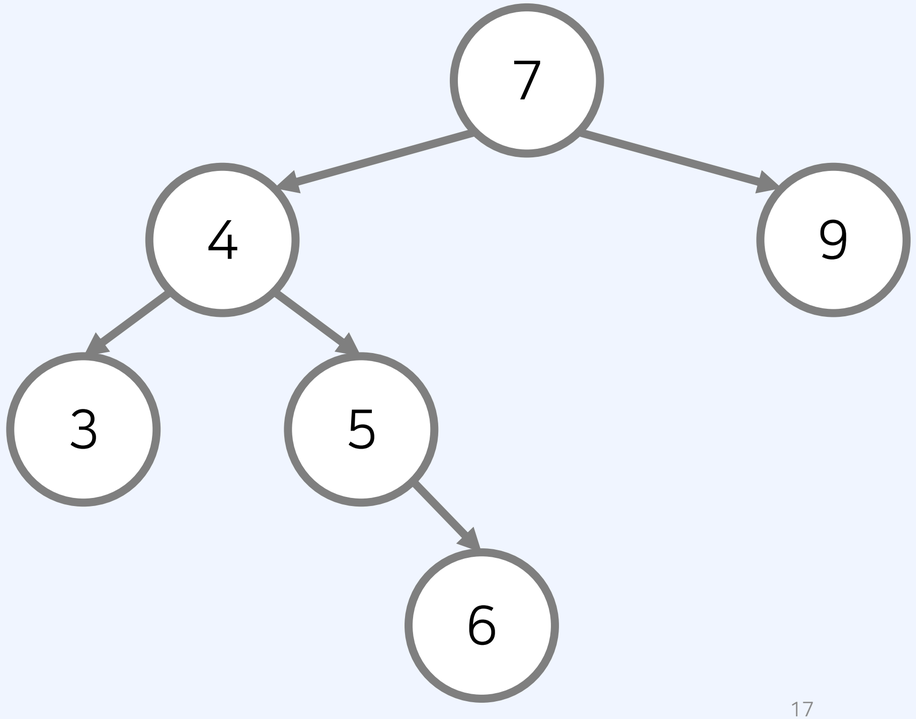
이진 탐색 트리 – 삽입 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삽입할 위치를 찾는다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 삽입
- **삽입할 노드 목록 예시**: [7, 4, 5, 9, 6, 3, 2, 8]



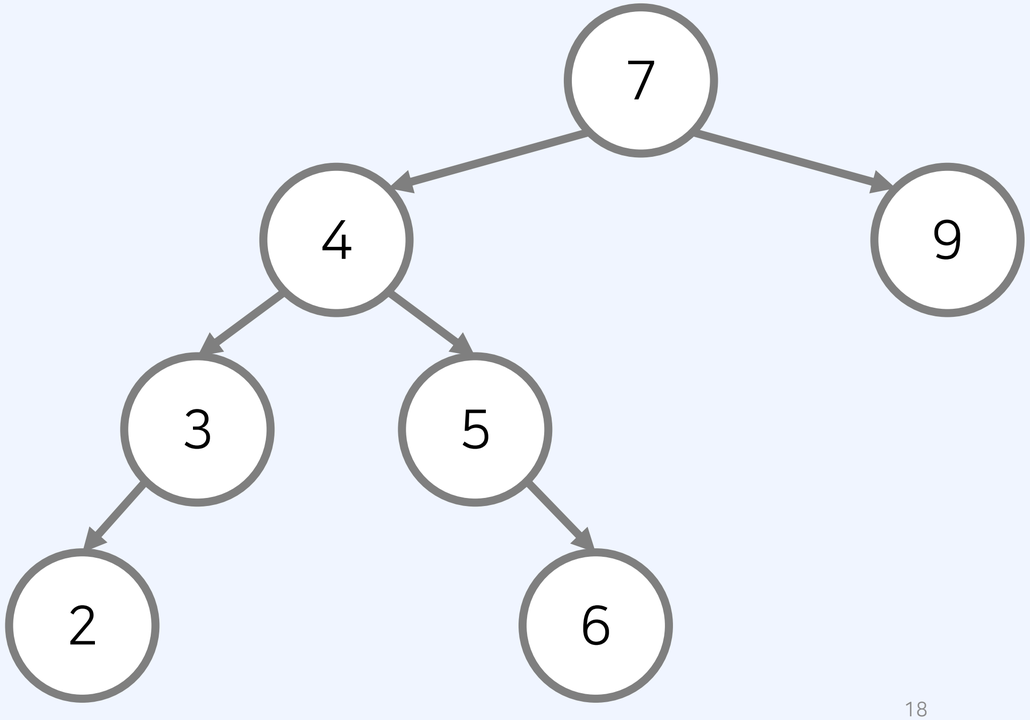
이진 탐색 트리 – 삽입 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삽입할 위치를 찾는다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 삽입
- **삽입할 노드 목록 예시**: [7, 4, 5, 9, 6, 3, 2, 8]



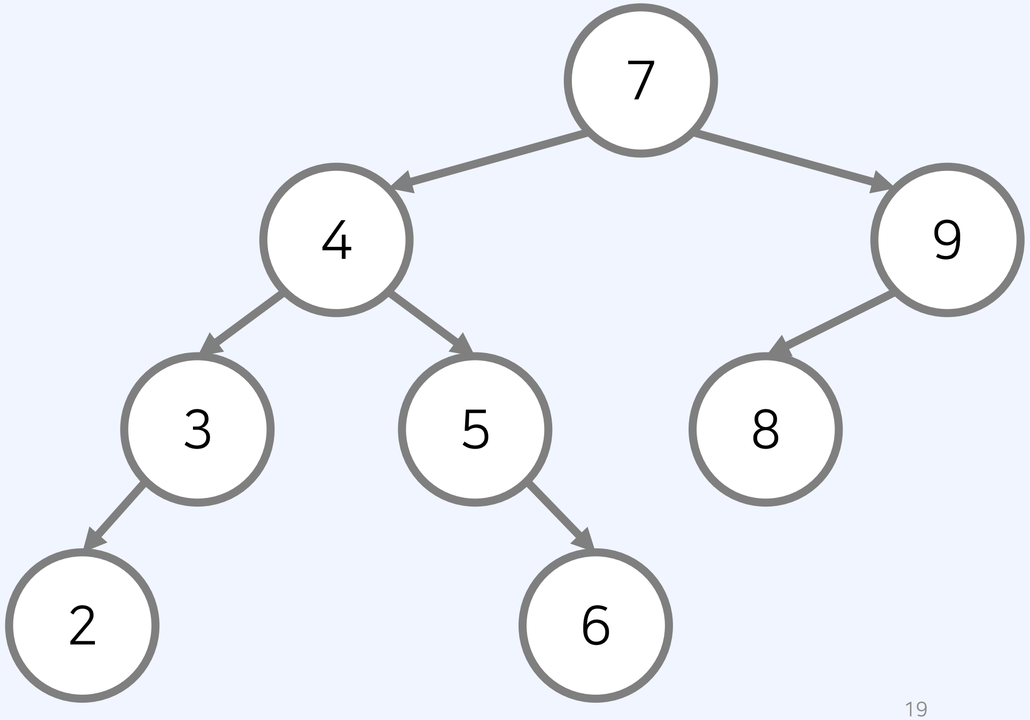
이진 탐색 트리 – 삽입 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삽입할 위치를 찾는다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 삽입
- **삽입할 노드 목록 예시**: [7, 4, 5, 9, 6, 3, 2, 8]



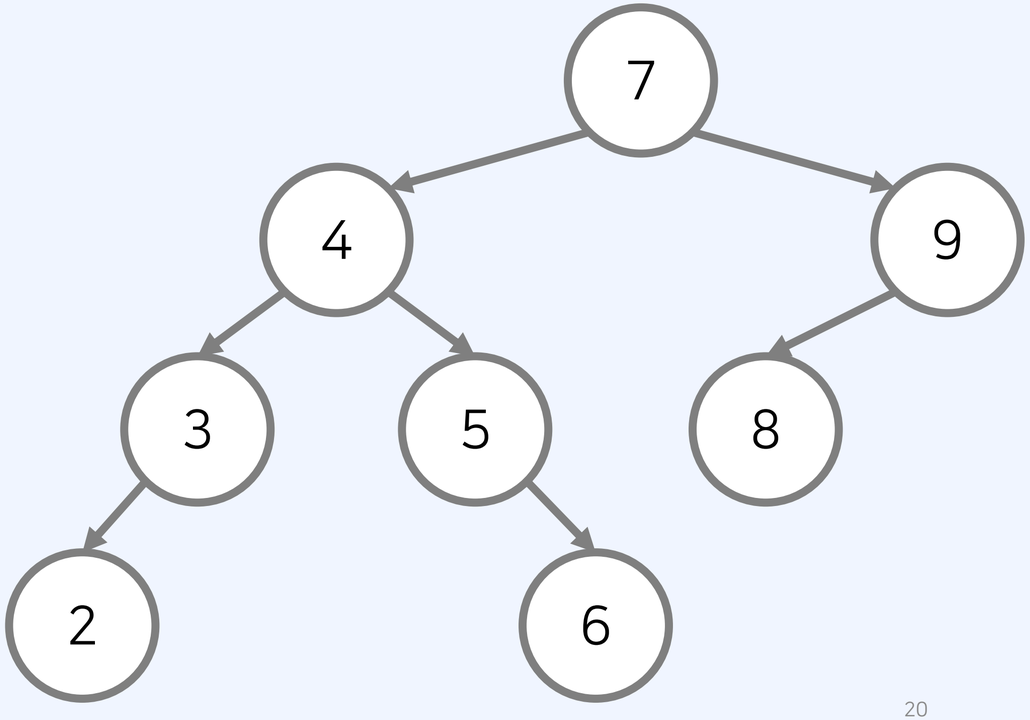
이진 탐색 트리 – 삽입 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삽입할 위치를 찾는다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 삽입
- **삽입할 노드 목록 예시**: [7, 4, 5, 9, 6, 3, 2, 8]



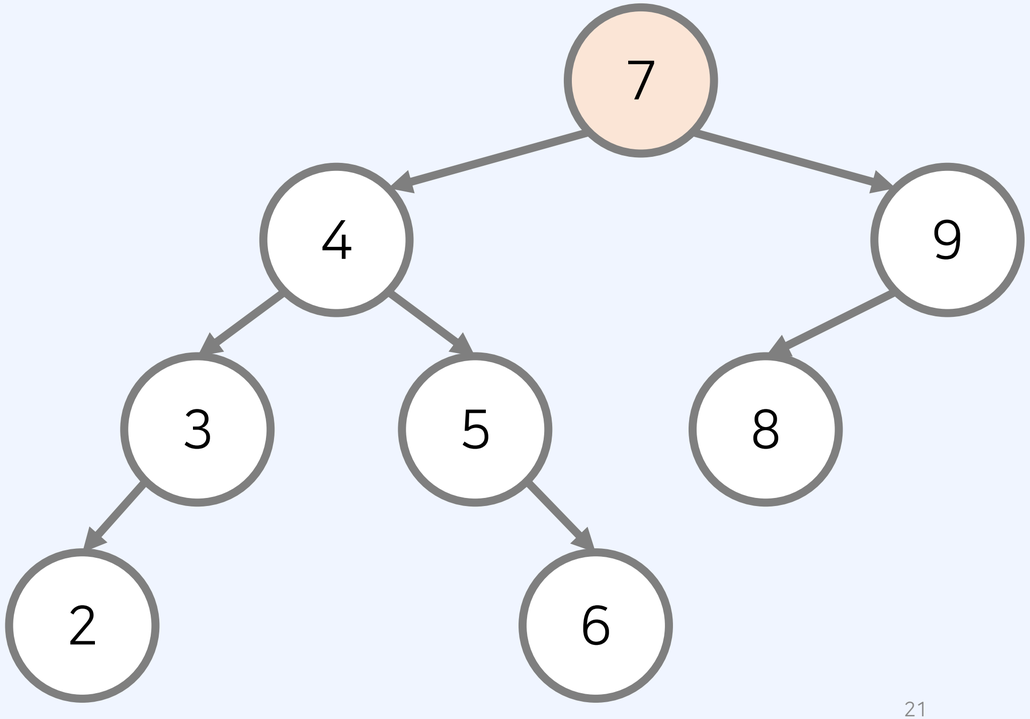
이진 탐색 트리 – 조회 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 찾고자 하는 원소를 조회한다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 조회
- **조회할 노드 목록 예시**: 5번 노드



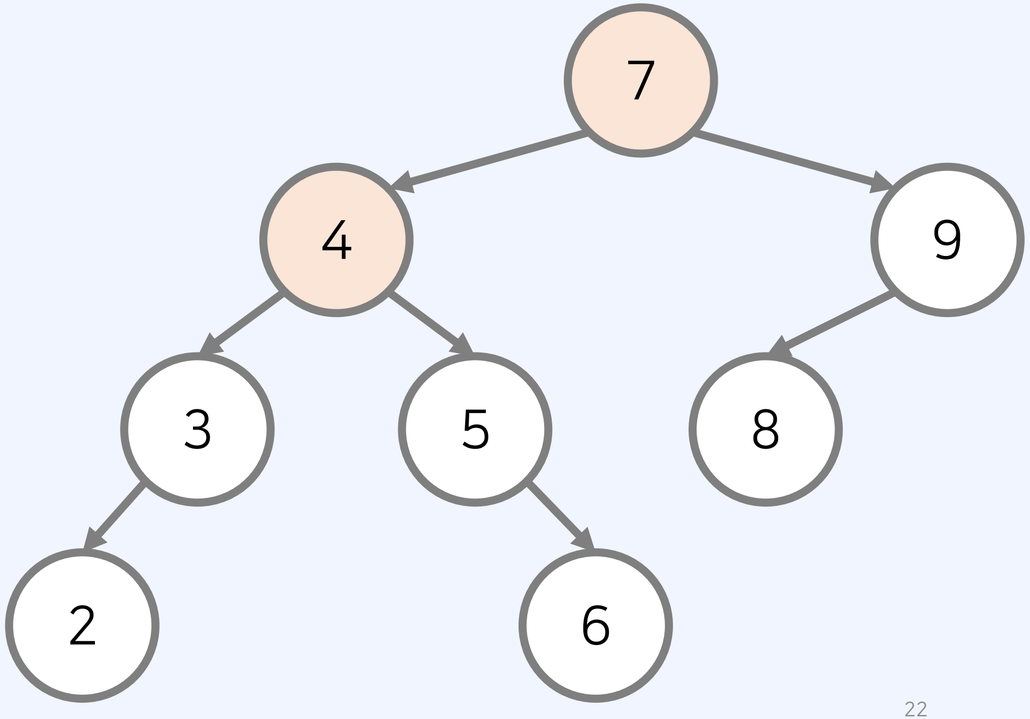
이진 탐색 트리 – 조회 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 찾고자 하는 원소를 조회한다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 조회
- **조회할 노드 목록 예시**: 5번 노드



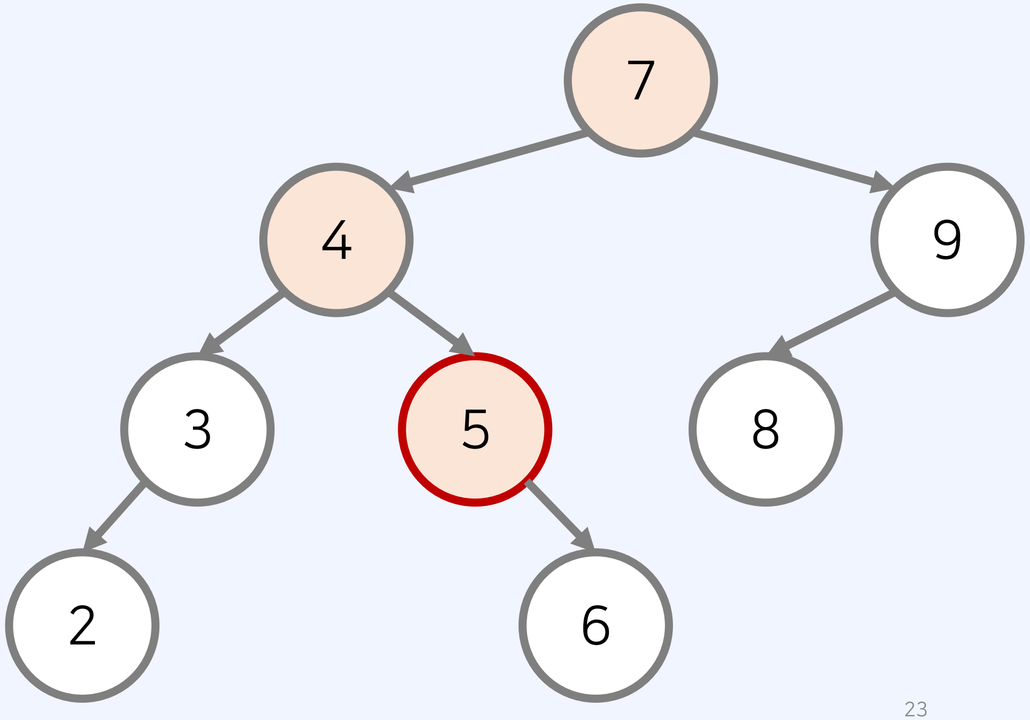
이진 탐색 트리 – 조회 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 찾고자 하는 원소를 조회한다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 조회
- **조회할 노드 목록 예시**: 5번 노드



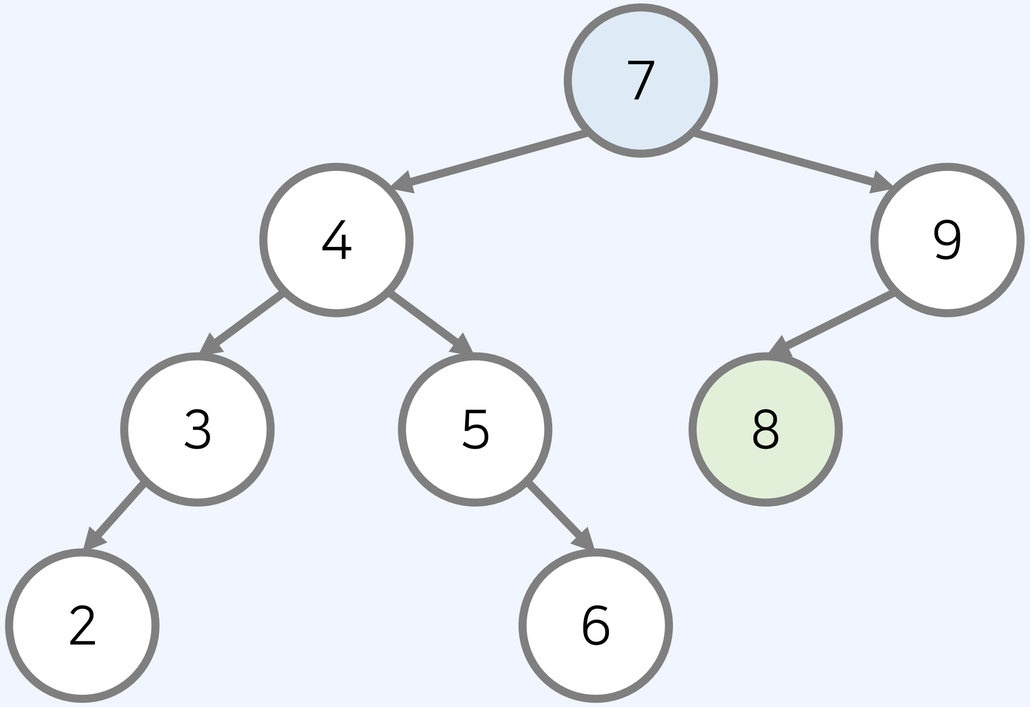
이진 탐색 트리 – 조회 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 찾고자 하는 원소를 조회한다.
- ① 삽입할 노드의 키(key)가 작으면 왼쪽으로, ② 삽입할 노드의 키(key)가 크면 오른쪽으로 조회
- **조회할 노드 목록 예시**: 5번 노드



이진 탐색 트리 – 삭제 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삭제할 원소에 접근한다.
- **삭제할 노드 목록 예시**: 7번 노드



Case #1 왼쪽 자식이 없는 경우 → 오른쪽 자식으로 대체

Case #2 오른쪽 자식이 없는 경우 → 왼쪽 자식으로 대체

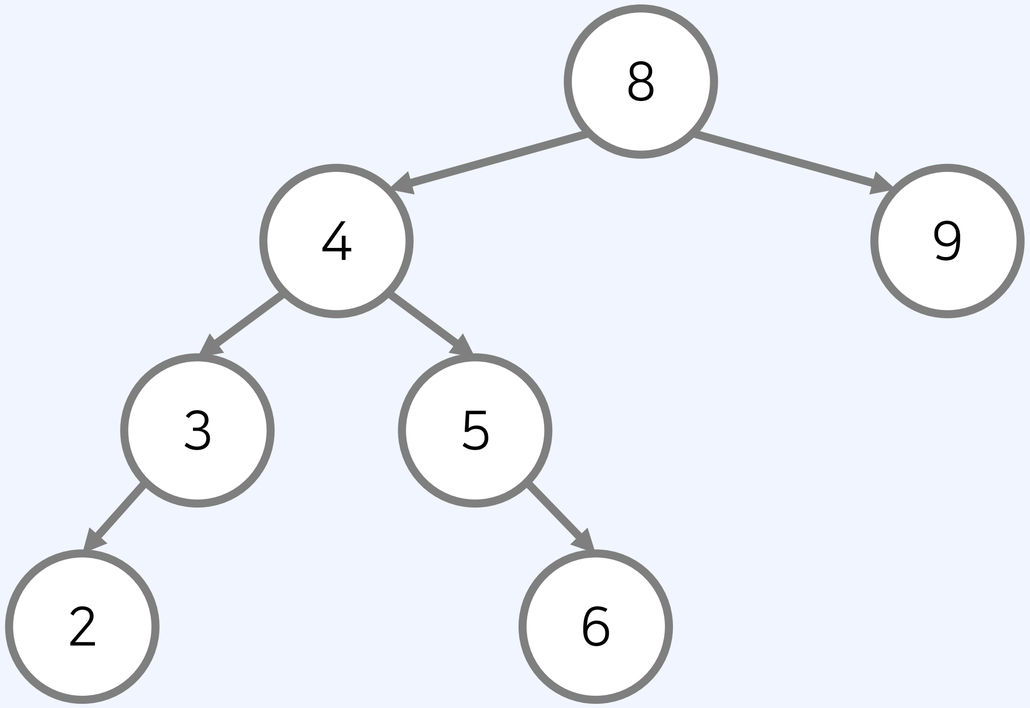
Case #3 왼쪽, 오른쪽이 모두 있는 경우 → 오른쪽 서브 트리에서 가장 작은 노드로 대체

: 삭제할 노드

: 대체할 노드

이진 탐색 트리 – 삭제 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삭제할 원소에 접근한다.
- **삭제할 노드 목록 예시**: 7번 노드



Case #1 왼쪽 자식이 없는 경우 → 오른쪽 자식으로 대체

Case #2 오른쪽 자식이 없는 경우 → 왼쪽 자식으로 대체

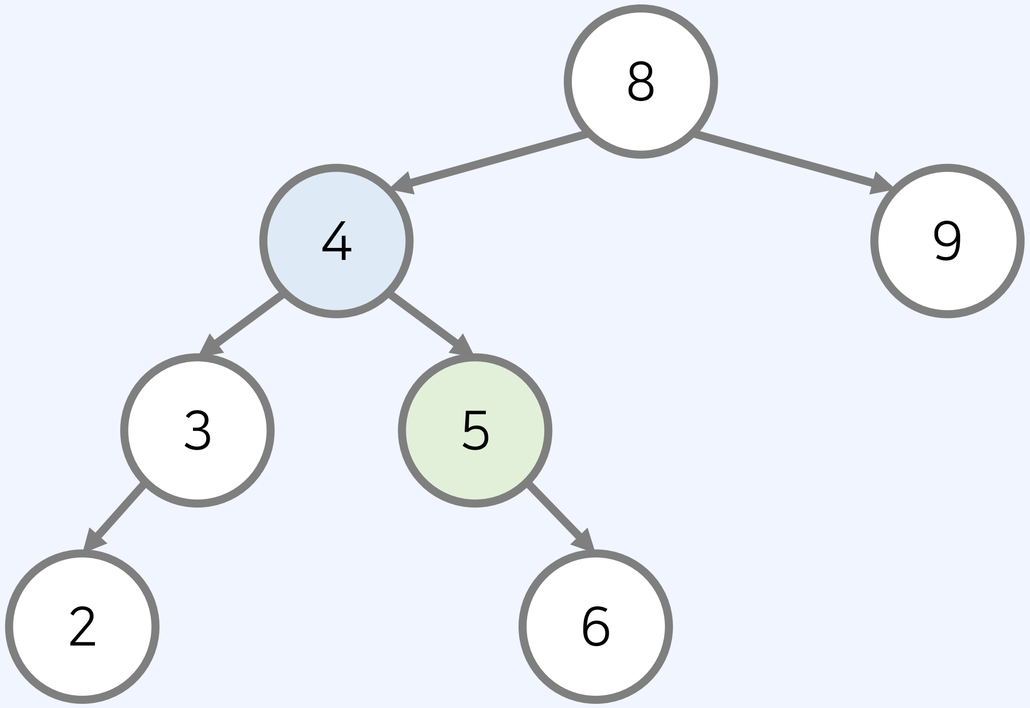
Case #3 왼쪽, 오른쪽이 모두 있는 경우 → 오른쪽 서브 트리에서 가장 작은 노드로 대체

: 삭제할 노드

: 대체할 노드

이진 탐색 트리 – 삭제 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삭제할 원소에 접근한다.
- **삭제할 노드 목록 예시**: 4번 노드



Case #1 왼쪽 자식이 없는 경우 → 오른쪽 자식으로 대체

Case #2 오른쪽 자식이 없는 경우 → 왼쪽 자식으로 대체

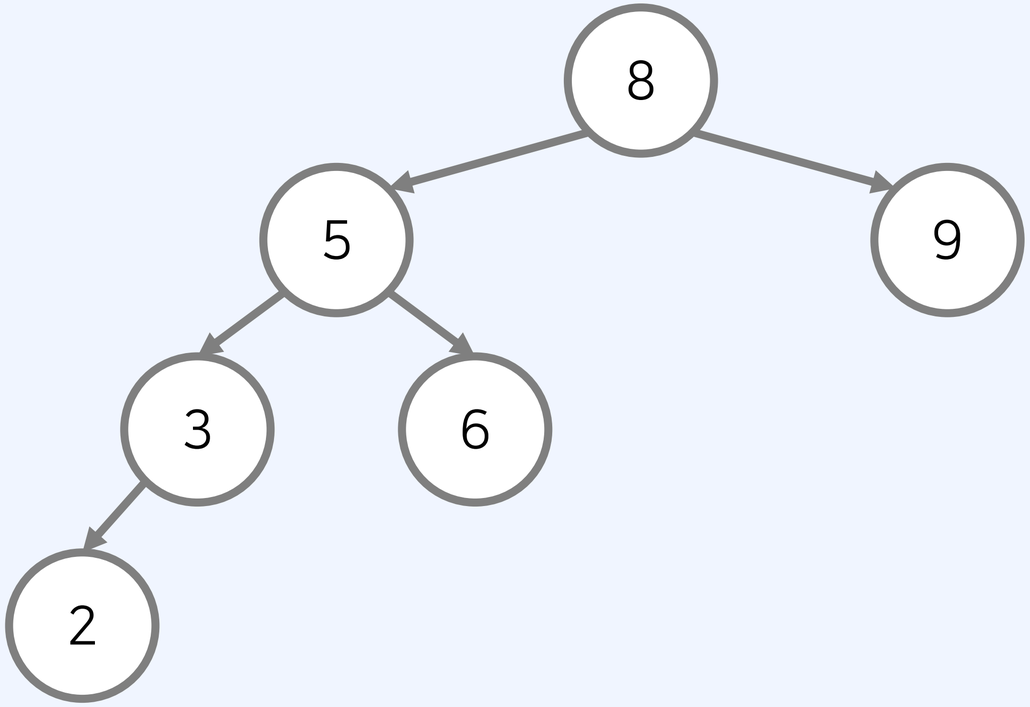
Case #3 왼쪽, 오른쪽이 모두 있는 경우 → 오른쪽 서브 트리에서 가장 작은 노드로 대체

: 삭제할 노드

: 대체할 노드

이진 탐색 트리 – 삭제 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삭제할 원소에 접근한다.
- **삭제할 노드 목록 예시**: 4번 노드



Case #1 왼쪽 자식이 없는 경우 → 오른쪽 자식으로 대체

Case #2 오른쪽 자식이 없는 경우 → 왼쪽 자식으로 대체

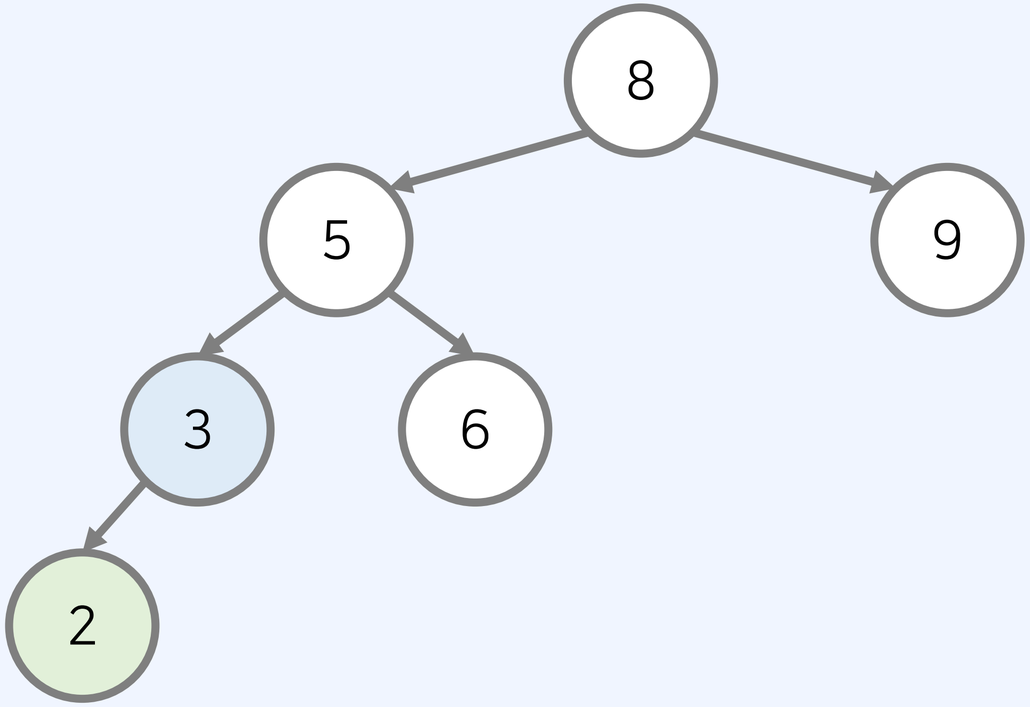
Case #3 왼쪽, 오른쪽이 모두 있는 경우 → 오른쪽 서브 트리에서 가장 작은 노드로 대체

: 삭제할 노드

: 대체할 노드

이진 탐색 트리 – 삭제 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삭제할 원소에 접근한다.
- **삭제할 노드 목록 예시**: 3번 노드



Case #1 왼쪽 자식이 없는 경우 → 오른쪽 자식으로 대체

Case #2 오른쪽 자식이 없는 경우 → 왼쪽 자식으로 대체

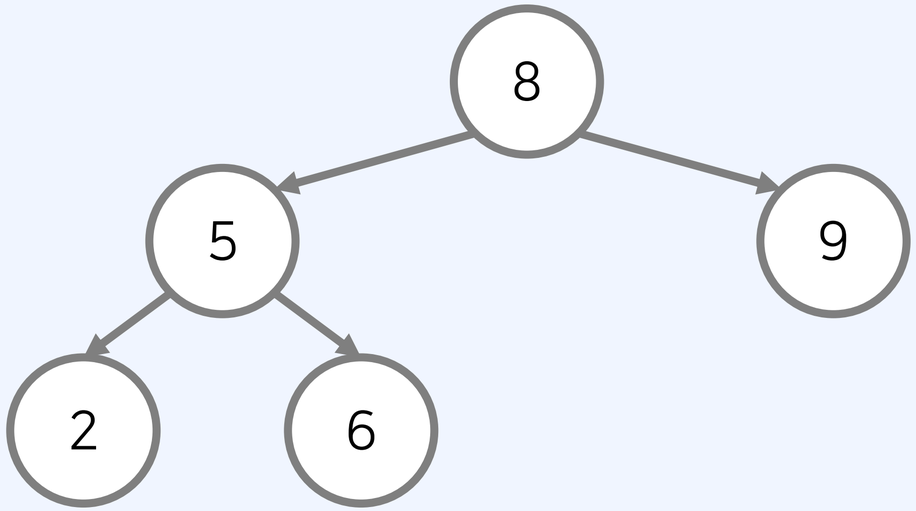
Case #3 왼쪽, 오른쪽이 모두 있는 경우 → 오른쪽 서브 트리에서 가장 작은 노드로 대체

: 삭제할 노드

: 대체할 노드

이진 탐색 트리 – 삭제 연산

- 루트 노드에서 출발하여 아래쪽으로 내려오면서, 삭제할 원소에 접근한다.
- **삭제할 노드 목록 예시**: 3번 노드



Case #1 왼쪽 자식이 없는 경우 → 오른쪽 자식으로 대체

Case #2 오른쪽 자식이 없는 경우 → 왼쪽 자식으로 대체

Case #3 왼쪽, 오른쪽이 모두 있는 경우 → 오른쪽 서브 트리에서 가장 작은 노드로 대체

: 삭제할 노드

: 대체할 노드

트리의 순회

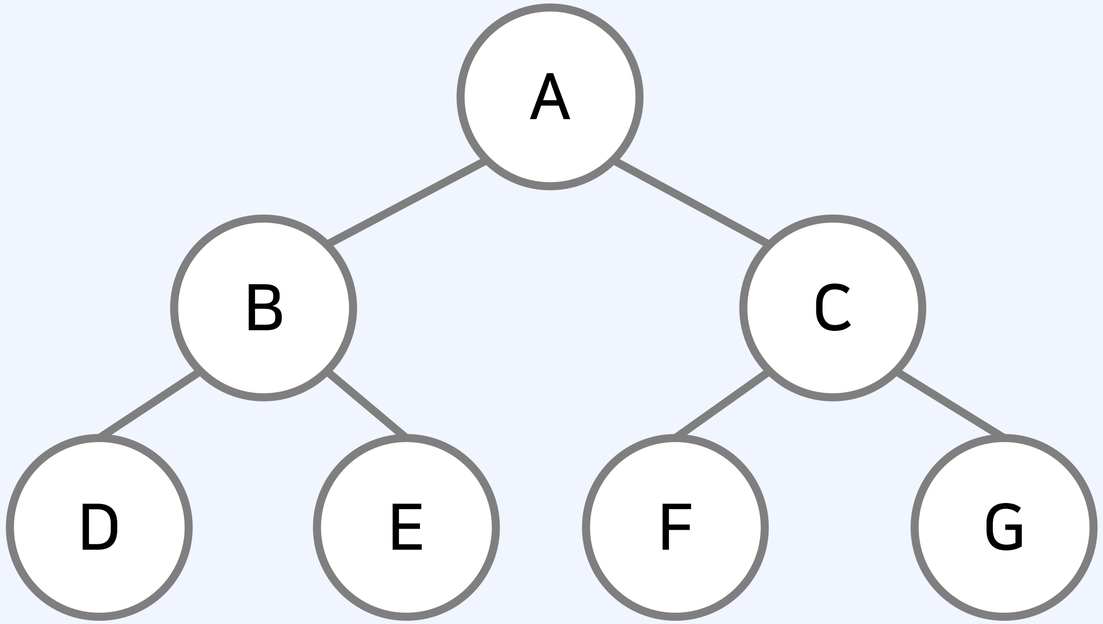
- 트리에 포함되어 있는 정보를 모두 출력하고자 할 때, 어떤 방식을 사용할 수 있을까?
- 바로 순회(traversal)를 사용할 수 있다.
- 트리의 모든 노드를 특정한 순서(조건)에 따라서 방문하는 방법을 순회(traversal)라고 한다.

① **전위 순회(pre-order traverse)**: 루트 방문 → 왼쪽 자식 방문 → 오른쪽 자식 방문

② **중위 순회(in-order traverse)**: 왼쪽 자식 방문 → 루트 방문 → 오른쪽 자식 방문

③ **후위 순회(post-order traverse)**: 왼쪽 자식 방문 → 오른쪽 자식 방문 → 루트 방문

트리의 순회 한 눈에 보기



- 전위 순회(pre-order traverse): A → B → D → E → C → F → G
- 중위 순회(in-order traverse): D → B → E → A → F → C → G
- 후위 순회(post-order traverse): D → E → B → F → G → C → A

트리의 순회 – 전위 순회(Preorder Traversal)

- **방문 방법**: 현재 노드 → 왼쪽 자식 노드 → 오른쪽 자식 노드

```
def _preorder(self, node):
    if node:
        print(node.key, end=' ')
        self._preorder(node.left)
        self._preorder(node.right)
```

트리의 순회 – 중위 순회(Inorder Traversal)

- **방문 방법**: 왼쪽 자식 노드 → 현재 노드 → 오른쪽 자식 노드

```
def _inorder(self, node):
    if node:
        self._inorder(node.left)
        print(node.key, end=' ')
        self._inorder(node.right)
```

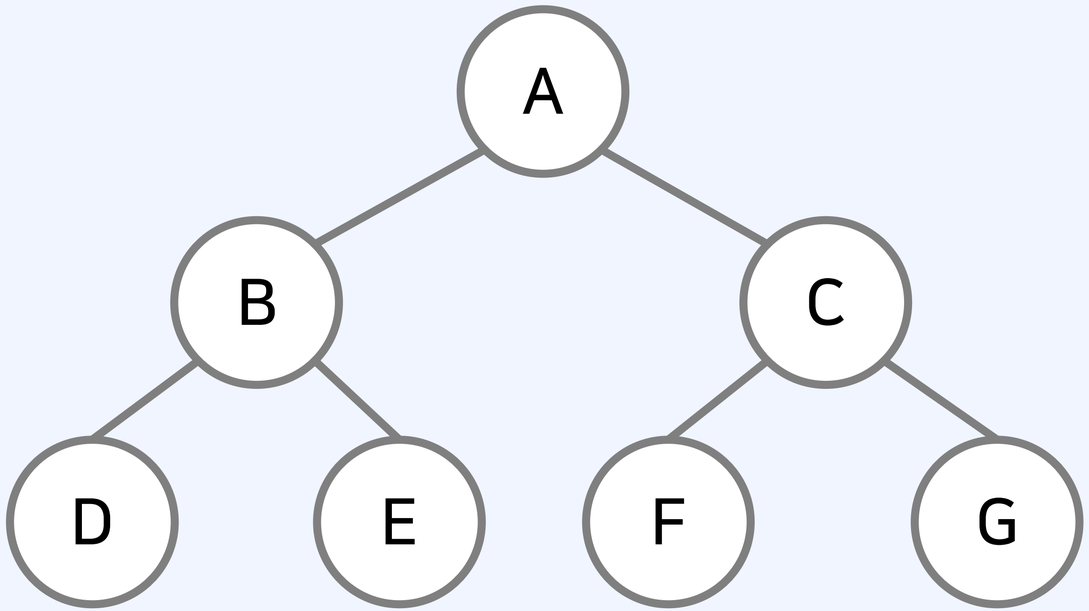
트리의 순회 – 후위 순회(Postorder Traversal)

- **방문 방법**: 왼쪽 자식 노드 → 오른쪽 자식 노드 → 현재 노드

```
def _postorder(self, node):
    if node:
        self._postorder(node.left)
        self._postorder(node.right)
        print(node.key, end=' ')
```

트리의 순회 – 레벨 순회(Level Order Traversal)

- 낮은 레벨(루트)부터 높은 레벨까지 순차적으로 방문한다.
- 단순히 루트 노드에서부터 **너비 우선 탐색(BST)**를 진행하면 된다.



- **레벨 순회 순회(level-order traverse)**: A → B → C → D → E → F → G

이진 탐색 트리의 구현

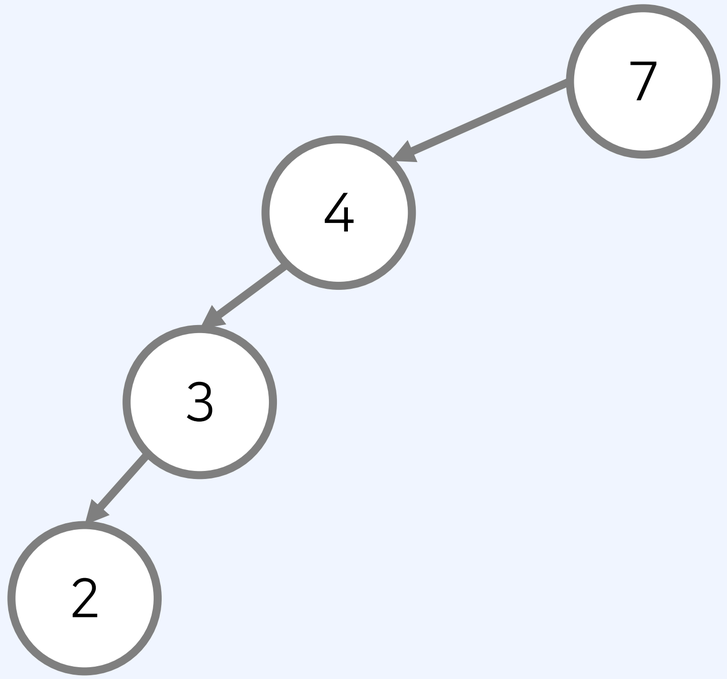
- 다른 메서드에서 사용되는 메서드는 이름 앞에 **언더바(_) 기호**를 붙인다.

```
def search(self, node, key):
    return self._search(self.root, key)

def _search(self, node, key):
    if node is None or node.key == key:
        return node

    # 현재 노드의 key보다 작은 경우
    if node.key > key:
        return self._search(node.left, key)
    # 현재 노드의 key보다 큰 경우
    elif node.key < key:
        return self._search(node.right, key)
```

편향 이진 트리(Skewed Binary Tree)

- 편향 이진 트리는 다음의 두 가지 속성을 가진다.

1. 같은 높이의 이진 트리 중 최소 개수의 노드 개수를 가진다.
2. 왼쪽 혹은 오른쪽으로 한 방향에 대한 서브 트리를 가진다.



이진 탐색 트리의 시간 복잡도

- 노드의 개수가 N개일 때, 시간 복잡도는 다음과 같다.
- 트리의 높이(height)을 H라고 할 때, 엄밀한 시간 복잡도는 $O(H)$다.
- 이상적인 경우 $H = \log_2 N$로 볼 수 있다.
- 하지만 최악의 경우(편향된 경우) $H = N$로 볼 수 있다.

※ 최악의 경우 시간 복잡도 ※

	조회	삽입	삭제	수정
균형 잡힌 이진 탐색 트리	$O(logN)$	$O(logN)$	$O(logN)$	$O(logN)$
편향 이진 탐색 트리	$O(N)$	$O(N)$	$O(N)$	$O(N)$

[참고] 균형 잡힌 트리: AVL 트리

- 이진 탐색 트리는 편향 트리가 될 수 있으므로, 최악의 경우 $O(N)$을 요구한다.
- 반면에 **AVL 트리**는 균형이 갖춰진 이진 트리다.
- 간단한 구현 과정으로 완전 이진 트리에 가까운 형태를 유지하도록 한다.