25th

Topological sorting

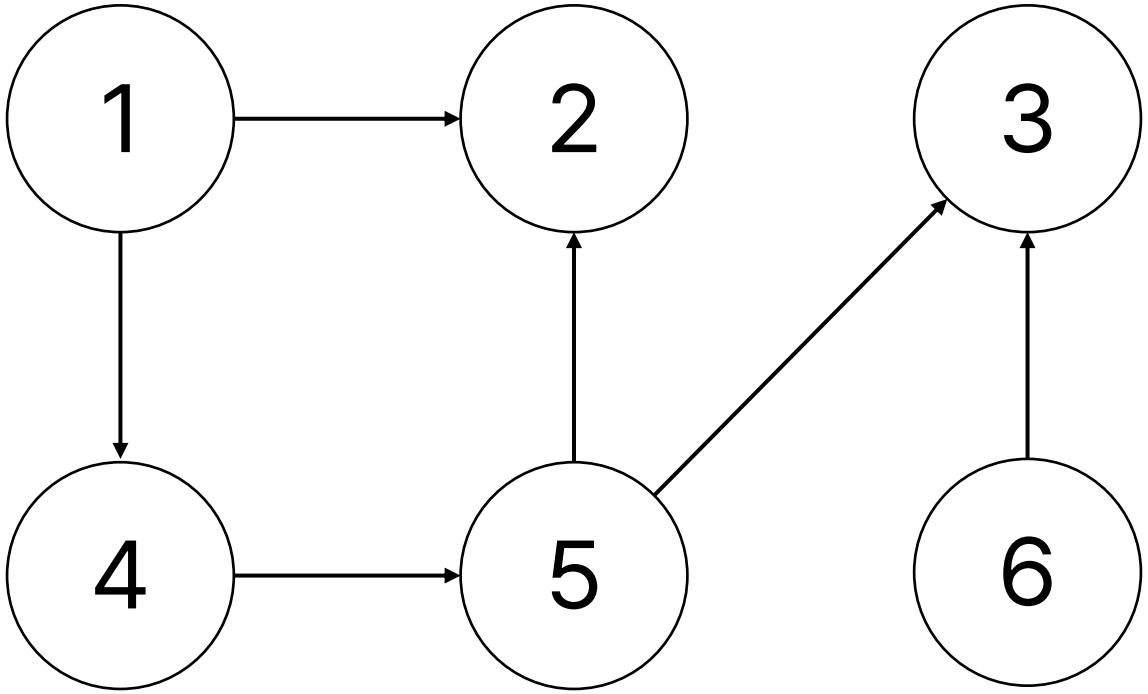
Topological sorting

- 위상 정렬
- 그래프를 간선의 방향에 맞추어 노드를 정렬하는 것
- DAG에서만 가능하다

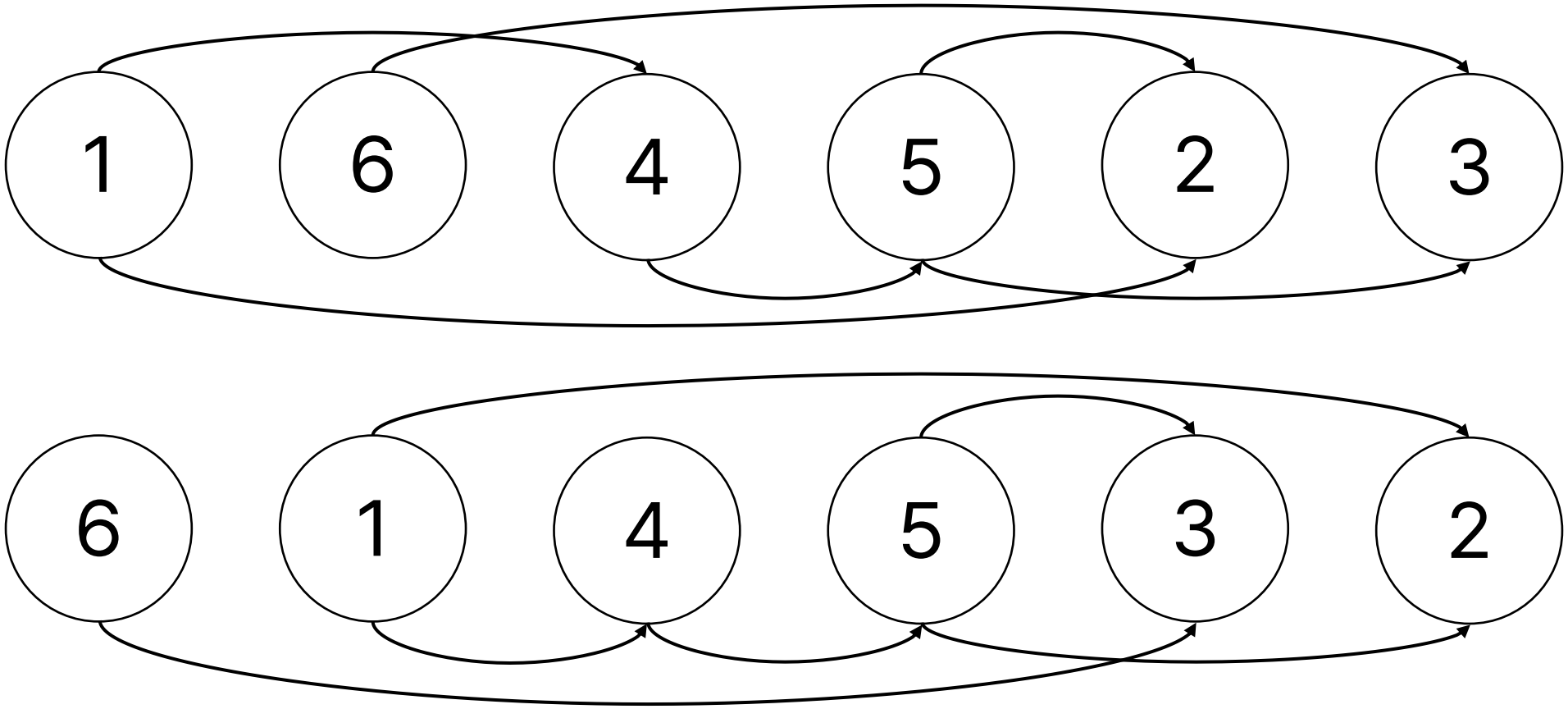
Topological sorting

- 그래프에서 간선이 의미하는 것은 순서를 지칭한다
- A->B간선이 의미하는 것은 A 이후에 B가 나와야 한다는 것을 의미한다
- 이러한 간선들의 방향성을 모두 유지하면서 노드들을 정렬하는 것을 위상 정렬이라고 한다

Topological sorting



Topological sorting



Topological sorting

- 선수 과목
- 대학교에서는 과목을 수강하기 위해서는 특정 과목을 들어야 하는 경우가 있다
- ex) 운영체제를 듣기 위해서는 시스템 프로그래밍을 먼저 들어야한다
- 이러한 경우, 시스템 프로그래밍에서 운영체제로 가는 간선이 존재한다
- 해당 간선의 방향을 유지하기 위해 시스템 프로그래밍이 운영체제보다 먼저 나와야한다

Topological sorting

- 스타크래프트 건물
- 요구 조건이 없는 건물은 바로 건설할 수 있다
- 요구 조건이 있는 건물은 특정 건물을 먼저 건설하고 그 다음에 건설해야 한다

- 이러한 조건에서 건물의 순서를 올바르게 정렬하는 것이 위상 정렬이다

Topological sorting

https://liquipedia.net/starcraft2/Zerg_Tech_Tree

Topological sorting

- 스포닝 풀 다음에 하이브를 건설하는 것은 위상 정렬된 건설 순서
- 반대로 스파이어를 건설하고 스포닝 풀을 건설하는 것은 잘못된 위상 정렬

- 즉, 간선의 방향성을 유지해야 한다

Topological sorting

- 이 내용들을 그래프 관점에서 살펴보자
- 위상 정렬을 하는 것은 정점 번호를 하나로 나열하는 것으로 생각할 수 있다
- 모든 간선의 시작점이 끝점보다 먼저 나오게 순회하면 위상 정렬이 된 것이다

Directed Acyclic Graph

- 유향 비순환 그래프 줄여서 DAG라고 부른다
- 간선의 방향성이 없는 Undirected Graph인 경우, 간선은 양방향으로 존재한다 생각한다
- 간선이 양방향으로 존재하는 경우, 정방향 간선과 역방향 간선이 공존하는 것을 의미하므로 어떠한 경우에도 간선의 방향을 맞출 수 없다

Directed Acyclic Graph

- Cycle이 존재하는 경우도 마찬가지이다
- Cycle에 포함된 두 노드 A, B 사이에는 A->B 경로와 B->A 경로가 동시에 존재한다
- A, B순서로 놓아도 B->A 경로가 존재하며 B, A 순서로 놓아도 A->B 경로가 존재한다.
- 따라서 사이클이 존재하는 그래프에서 위상정렬은 불가능하다
- 따라서 위상 정렬은 DAG에서만 가능하다

Topological sorting

- DAG라는 가정하에 위상 정렬을 해보자
- 순회를 이용해 위상 정렬을 할 수 있다
- BFS를 이용한 방법과 DFS를 이용한 방법이 있다

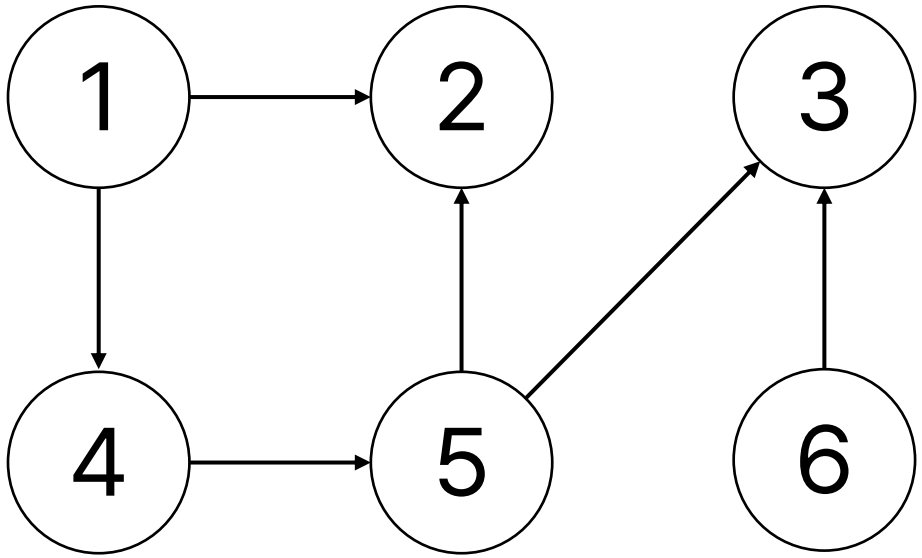
Topological sorting BFS

- BFS의 큐에는 탐색이 가능한 노드들을 넣는 것을 활용
- 우선 시작할 때는 In Degree가 0인 노드들만 올 수 있다
- In Degree가 0인 노드들을 큐에 넣고 탐색한다
- 탐색한 노드에서 시작하는 간선들이 존재한다면 도착 노드의 In Degree를 감소시킨다
- In Degree가 0이 되었다는 것은 선행되어야 할 모든 노드를 탐색했다는 뜻이다
- In Degree가 0이 되면 큐에 넣는다

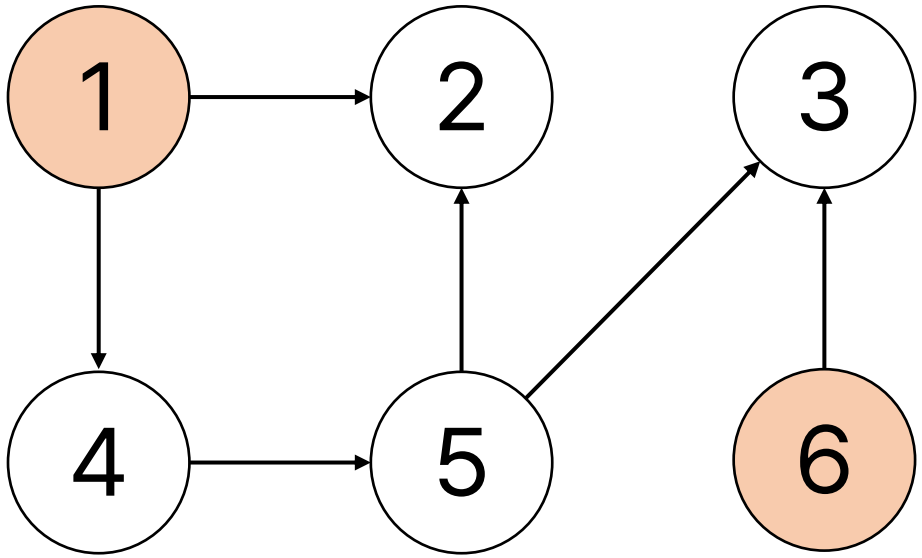
Topological sorting BFS

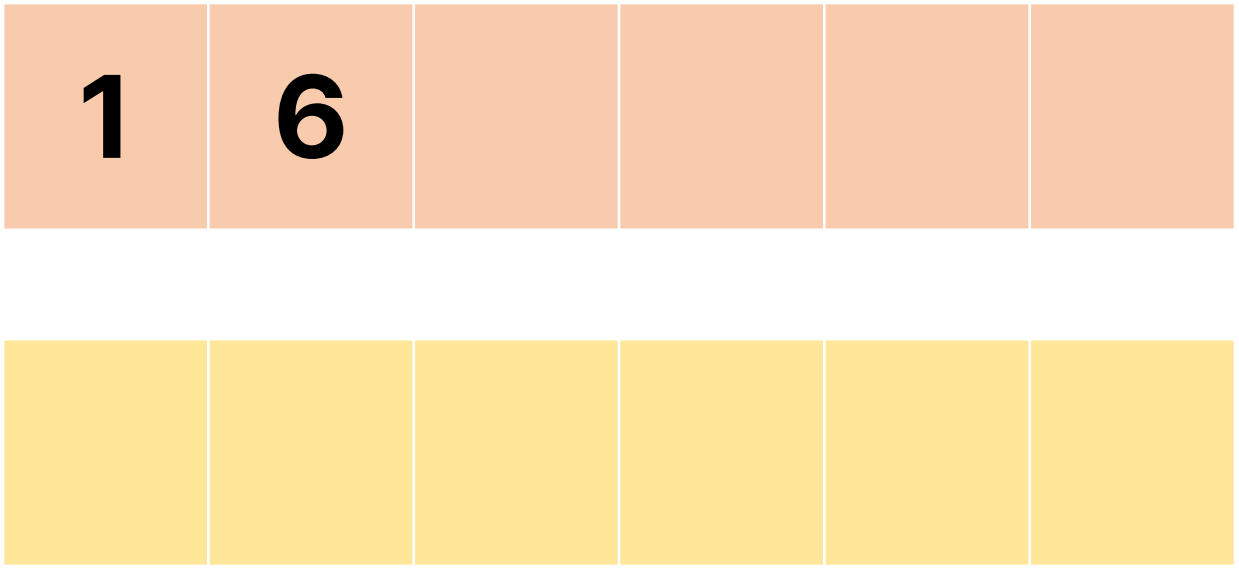
- 만일 사이클이 존재한다면 사이클에 포함된 모든 노드들은 입력 차수가 1이상이다
- 해당 노드들은 어떠한 경우에도 큐에 들어갈 수 없으므로 탐색되지 않는다
- 앞선 BFS를 마치고 나서 탐색되지 않은 노드들이 존재한다면 사이클이 존재하는 그래프, 위상 정렬이 불가능한 그래프이다

Topological sorting BFS

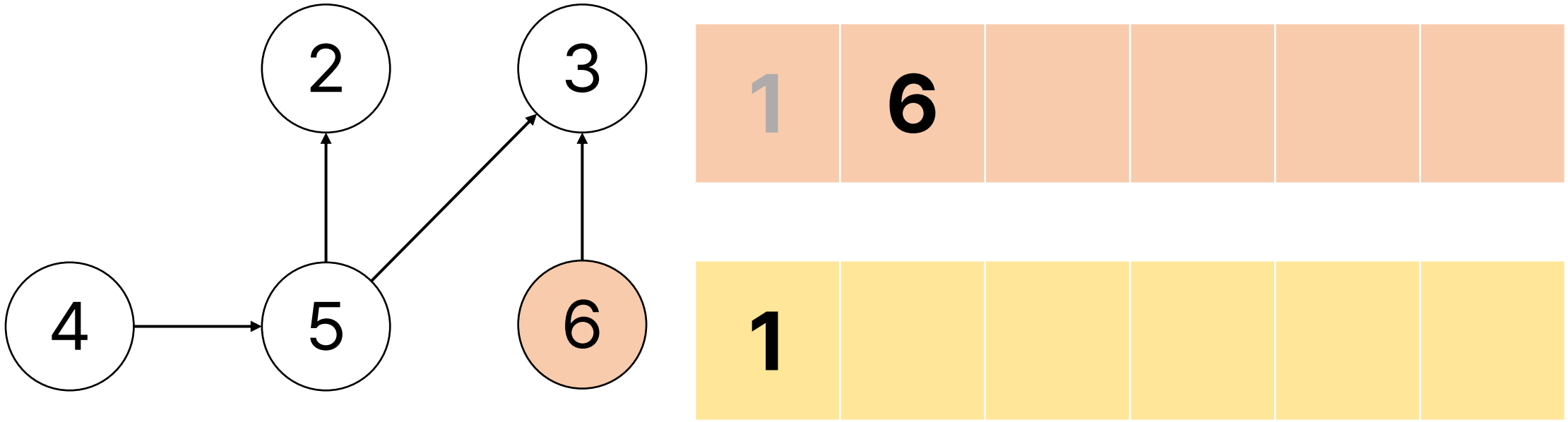


Topological sorting BFS

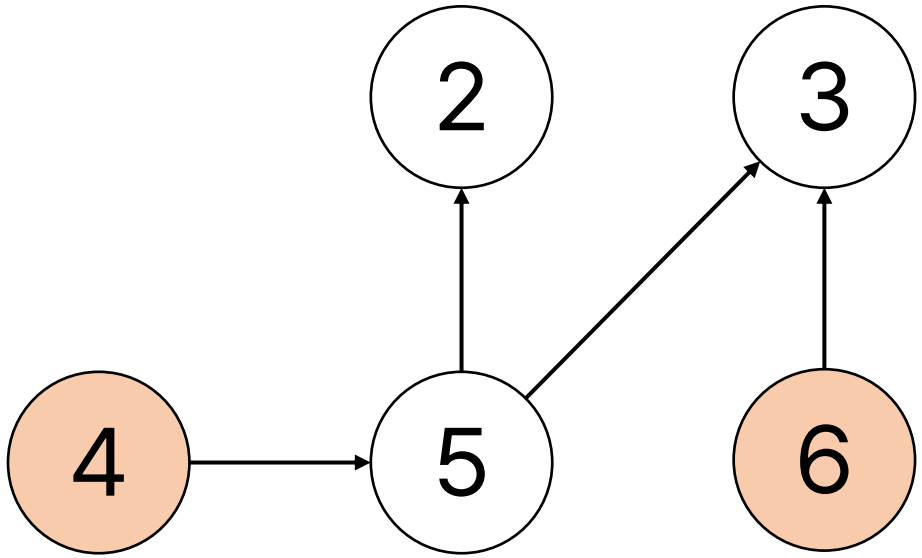




Topological sorting BFS



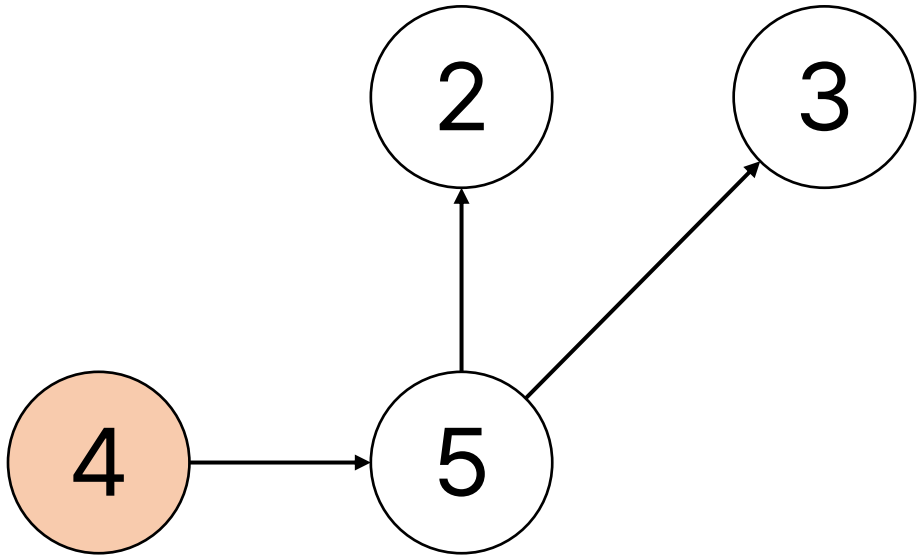
Topological sorting BFS



1	6	4			

1					

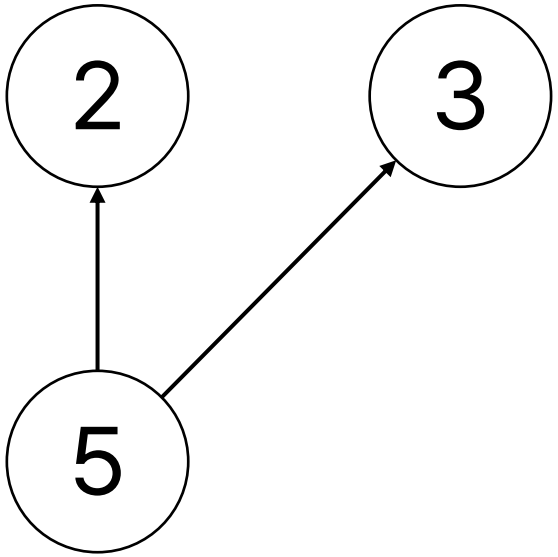
Topological sorting BFS

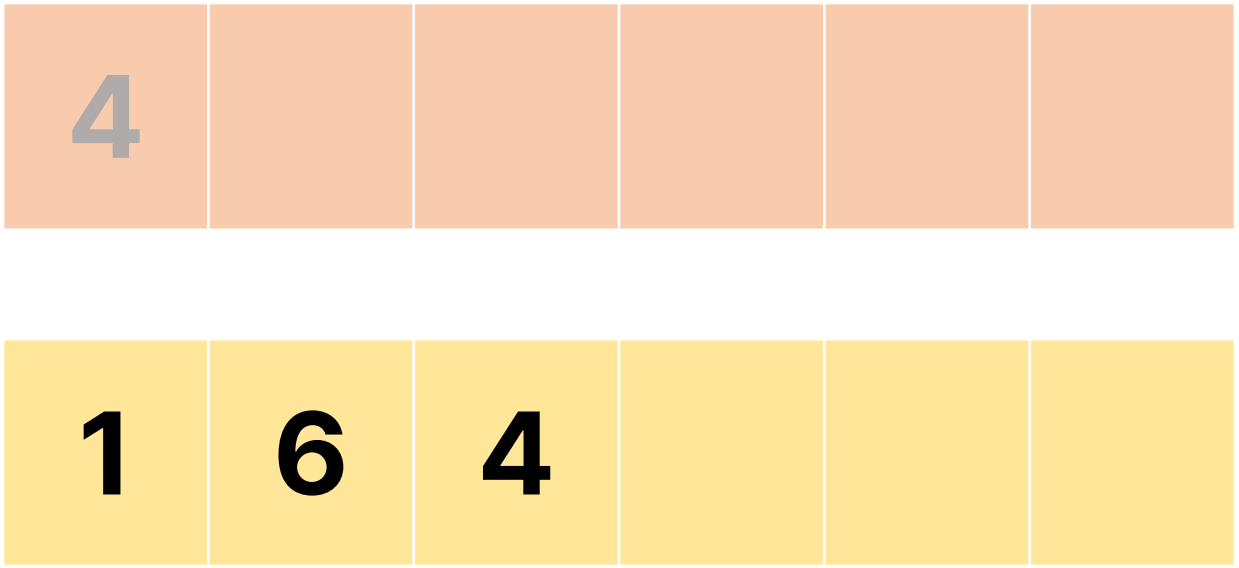


6	4				

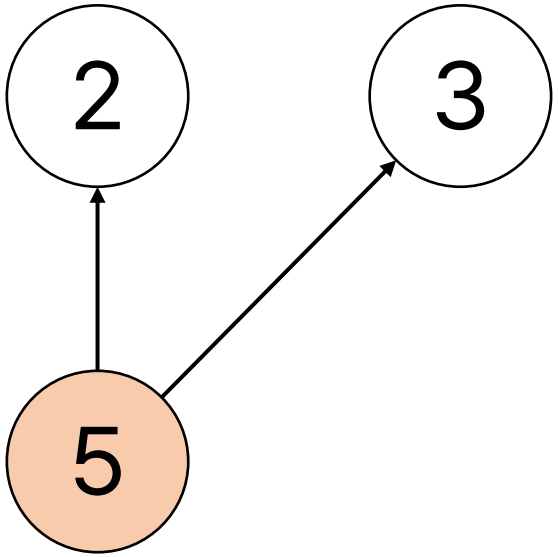
1	6				

Topological sorting BFS





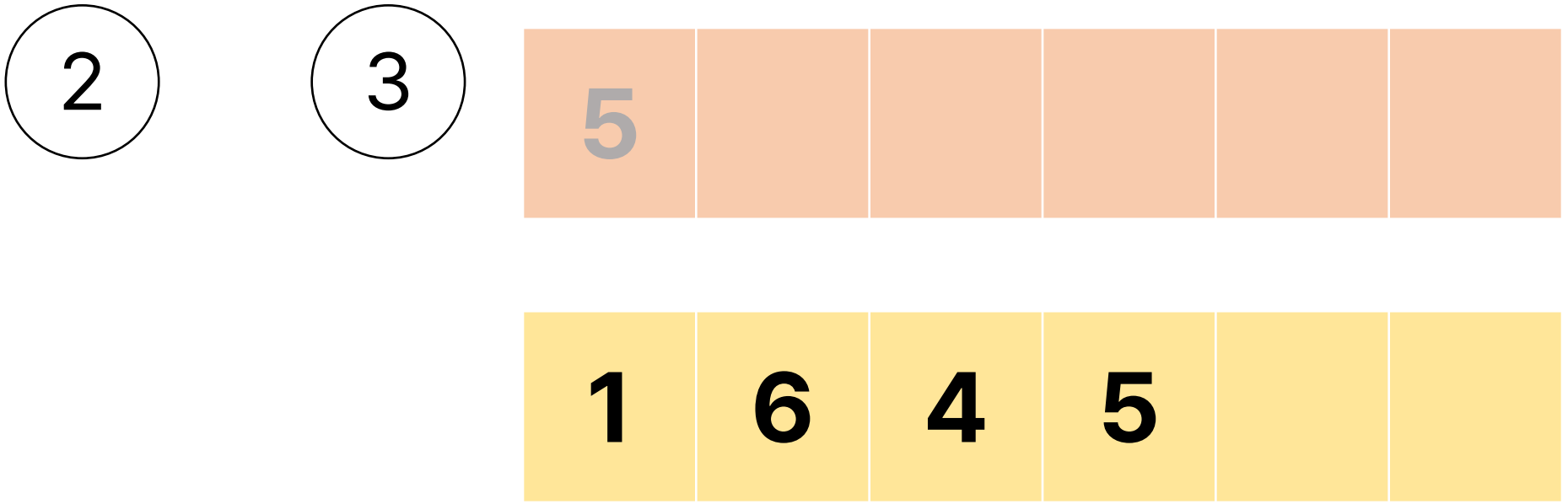
Topological sorting BFS



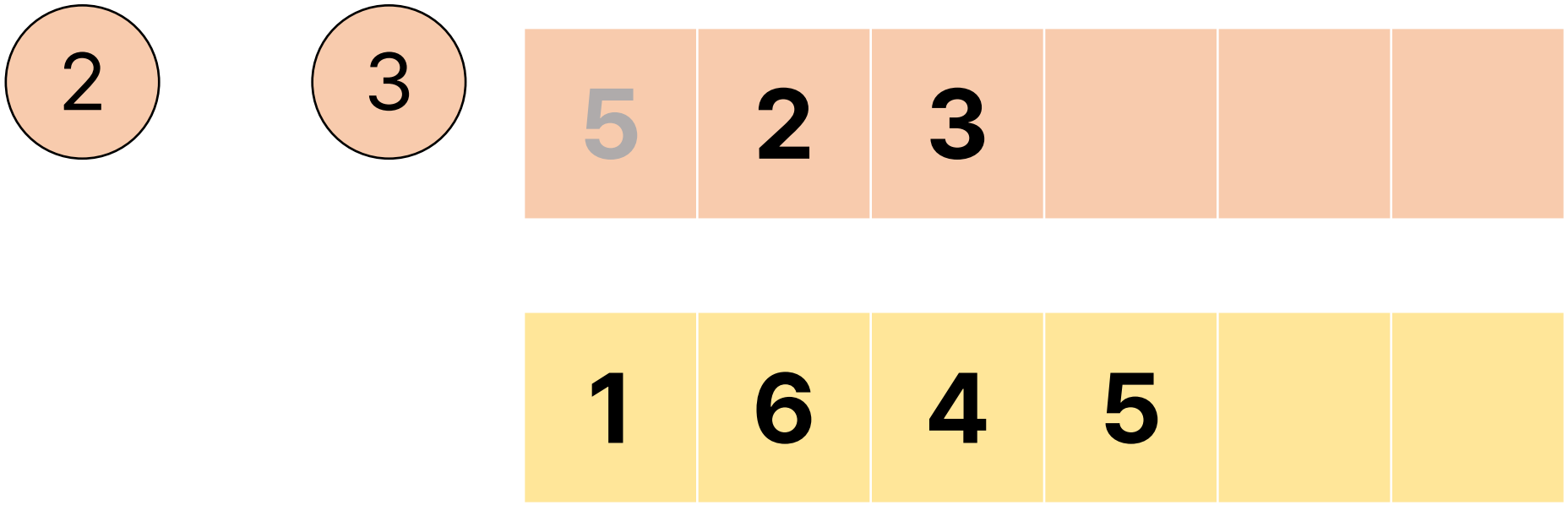
4	5				

1	6	4			

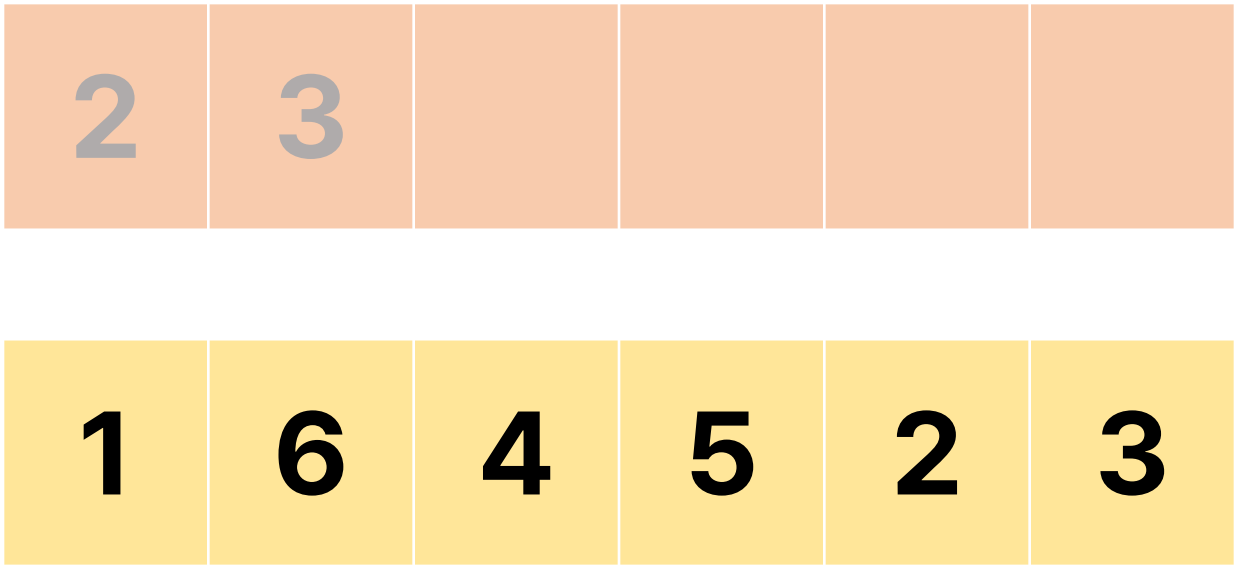
Topological sorting BFS



Topological sorting BFS



Topological sorting BFS



Topological sorting BFS

```
vector<int> edges[MAX_NODE];
int in_degree[MAX_NODE];
queue<int> bfs_queue;

int s, e;
for (int i = 0; i < MAX_EDGE; i++) {
    cin >> s >> e;
    edges[s].push_back(e);
    in_degree[e]++;
}
```

Topological sorting BFS

```
for (int i = 0; i < MAX_NODE; i++) {
    if (in_degree[i] != 0)
        continue;
    bfs_queue.push(i);
}

while (!bfs_queue.empty()) {
    int cur = bfs_queue.front();
    bfs_queue.pop();

    for (auto dst : edges[cur]) {
        in_degree[dst]--;
        if (in_degree[dst] == 0)
           bfs_queue.push(dst);
    }
}
```

Topological sorting DFS

- 아무 노드에서나 DFS 시작하고 DFS가 끝나는 순서(노드에서 나가는 순서)대로 기록한다
- 기록된 순서(DFS가 끝나는 순서)의 역순이 위상 정렬된 결과이다

- 단, DAG가 보장되어야 한다

Topological sorting DFS

- 역순이 위상 정렬이므로 마지막에 넣은 데이터가 위상 정렬에서 제일 첫번째 노드이다
- LIFO이므로 저장에 스택을 사용한다

Topological sorting DFS

```
vector<int> edges[MAX_NODE];
bool visited[MAX_NODE];
stack<int> st;

int s, e;
for (int i = 0; i < MAX_EDGE; i++) {
    cin >> s >> e;
    edges[s].push_back(e);
}
```

Topological sorting DFS

```
for (int i = 0; i < MAX_NODE; i++) {
    if (visited[i])
        continue;
    dfs(i);
}

void dfs(int cur) {
    visited[cur] = true;
    for (auto next : edges[cur]) {
        if (visited[next])
            continue;
        dfs(next);
    }
    st.push(cur);
}
```

Trees

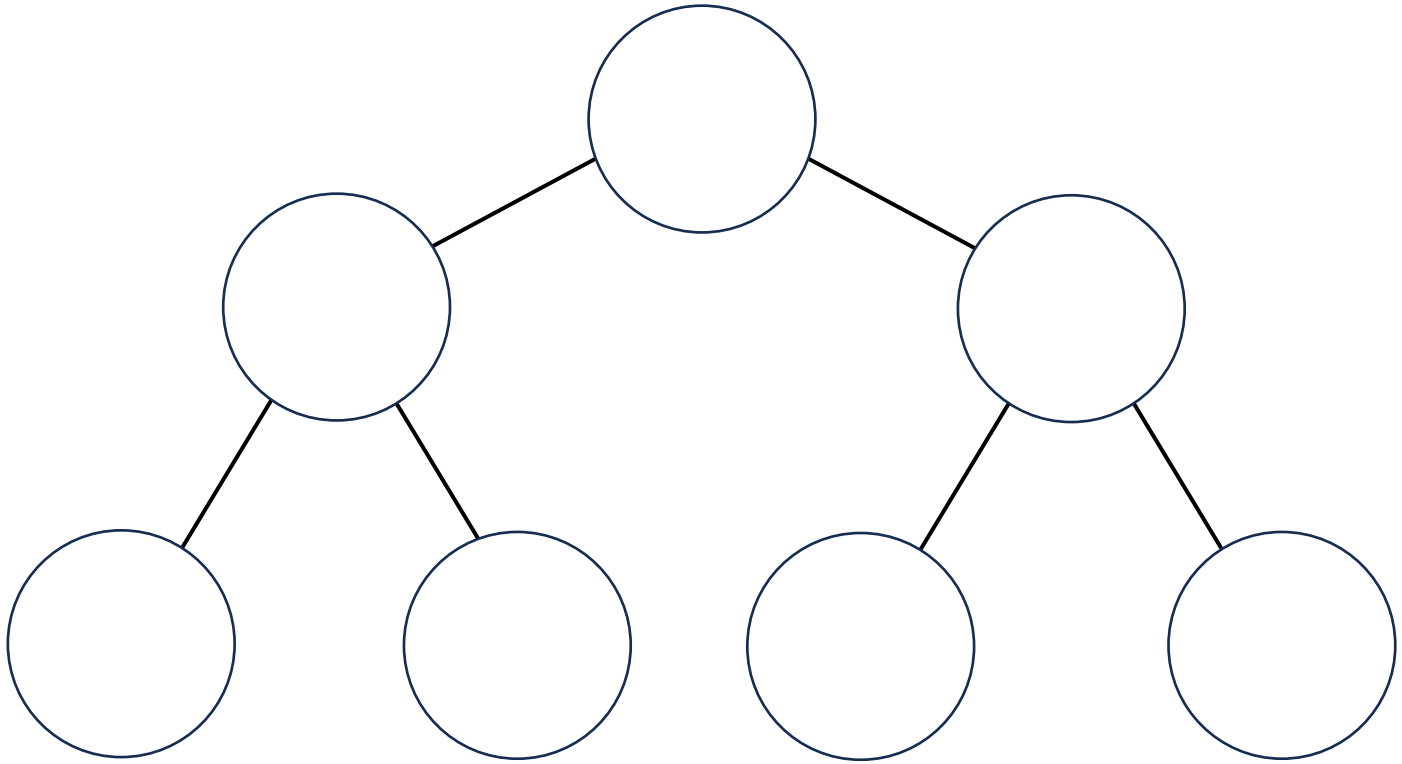
Trees

- 그래프의 한 종류이다
- 사이클이 존재하지 않고 하나의 연결 요소로만 이루어진 그래프이다
- 여러 개의 연결 요소인 경우, 트리가 여러 개인 경우는 숲(Forest)이라고 한다

Trees

- N개의 노드가 존재하는 경우 N-1개의 간선이 존재한다
- 루트 노드가 존재하며, 계층적 구조이다
- 부모 자식관계가 정해져 있다
- 각 노드가 가지고 있는 자식의 개수가 최대 K개 일 때, K진 트리라고 한다

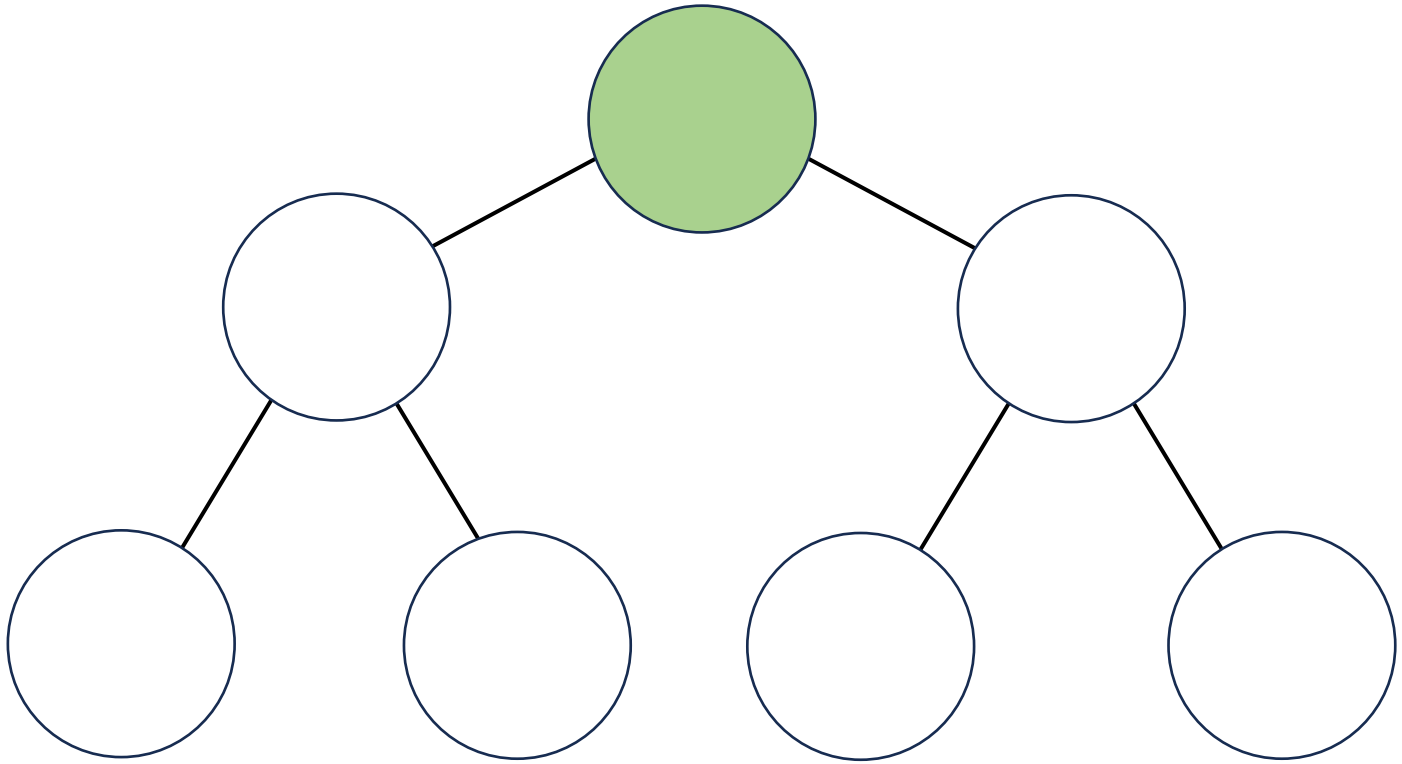
Trees



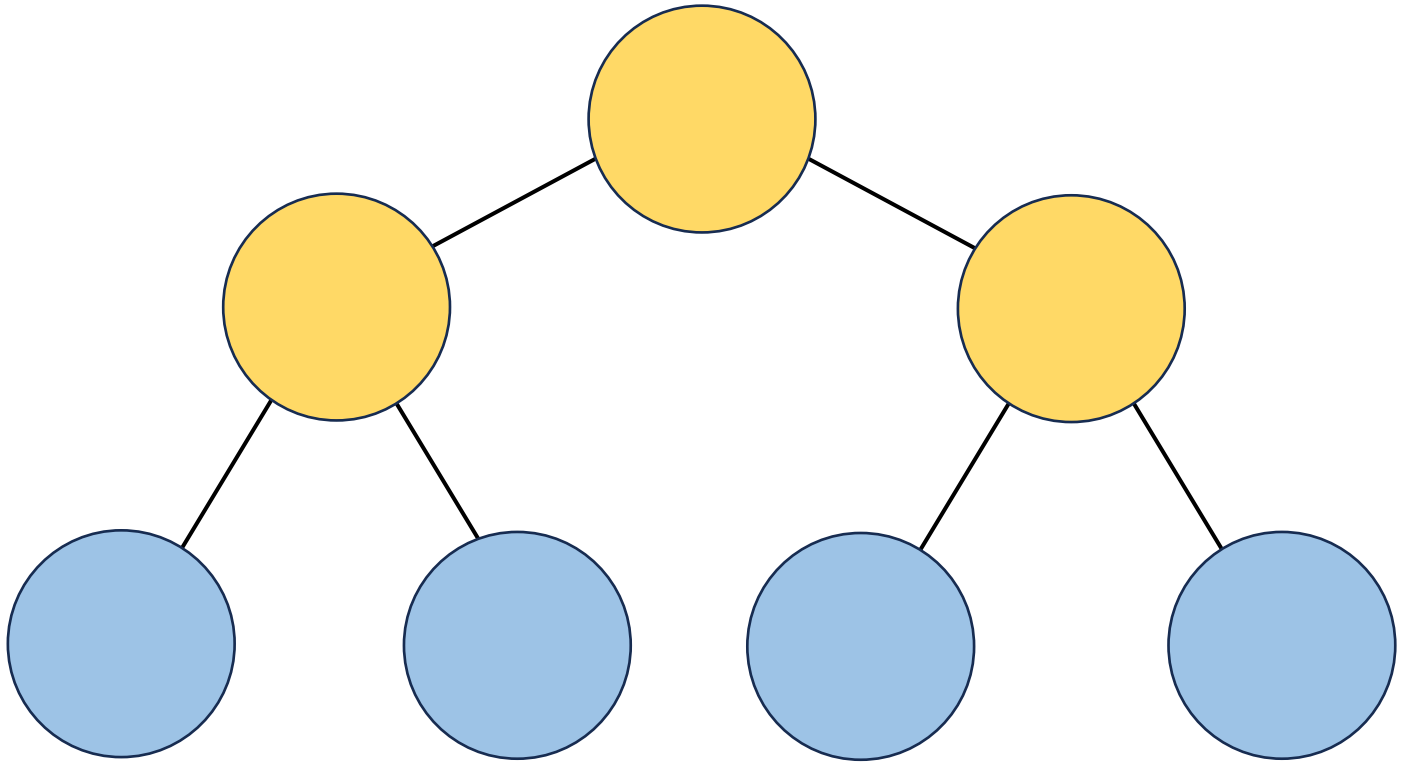
Trees

- Internal node: 자식 노드가 하나 이상 존재하는 노드
- External node, Leaf node: 자식 노드가 존재하지 않는 노드
- Sibling, 형제 노드: 같은 부모 노드를 가지고 있는 노드
- Ancestor, 조상: 부모관계로 있는 노드들, 부모 노드, 부모의 부모, 조부모의 부모, ...
- Descendant, 후손: 조상의 역, 내 하위에 있는 노드들
- Subtree: 트리 내에서 특정 노드를 루트 노드로 하는 트리

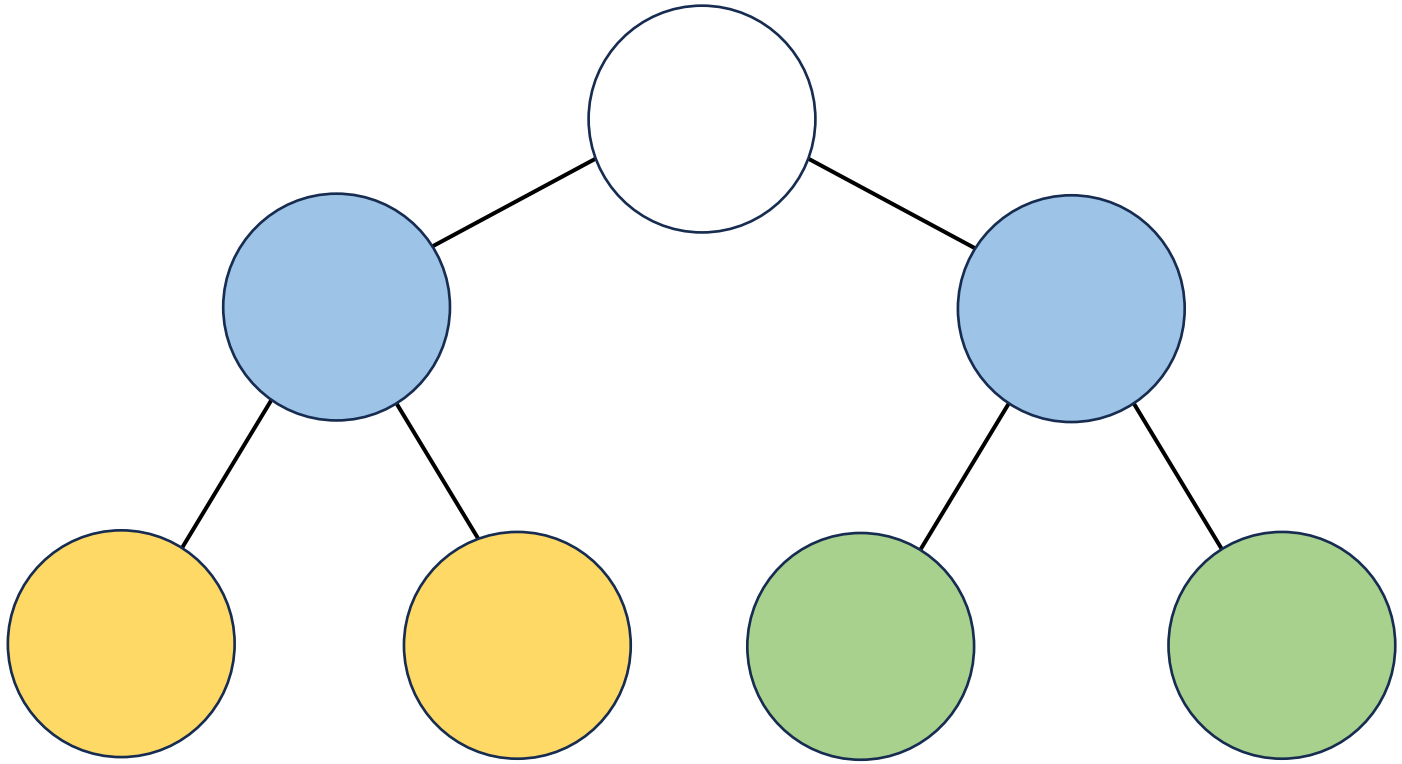
Root node



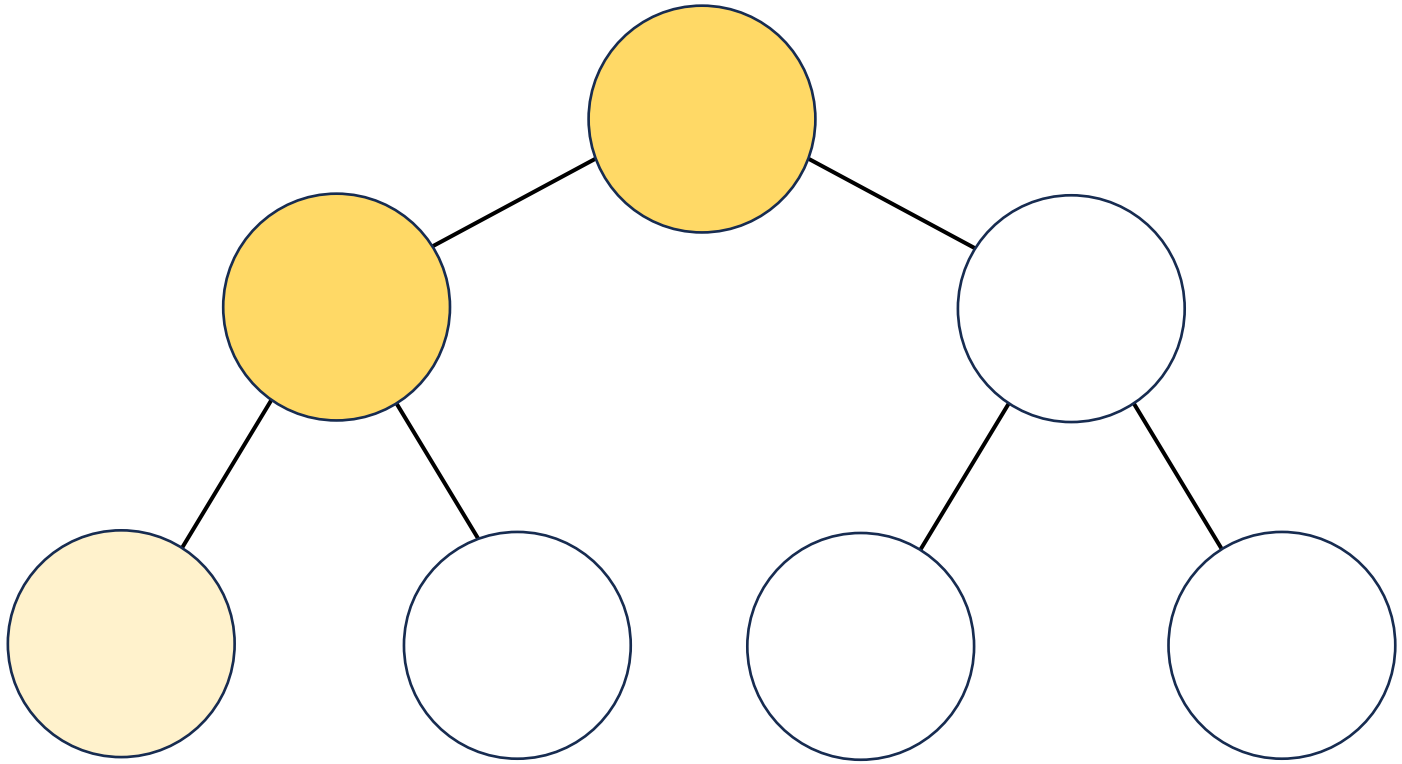
Internal/External node



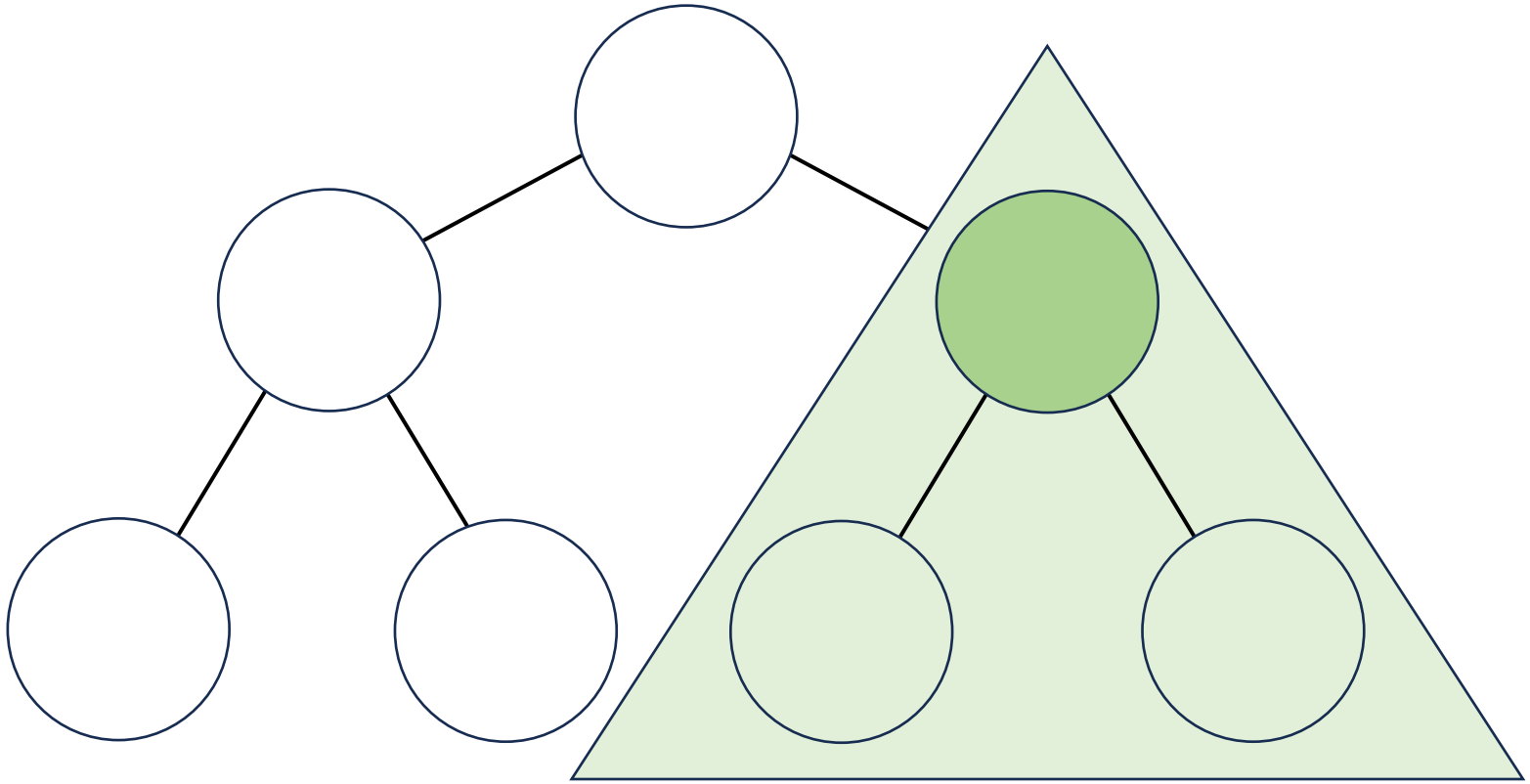
Sibling



Ancestor/Descendant



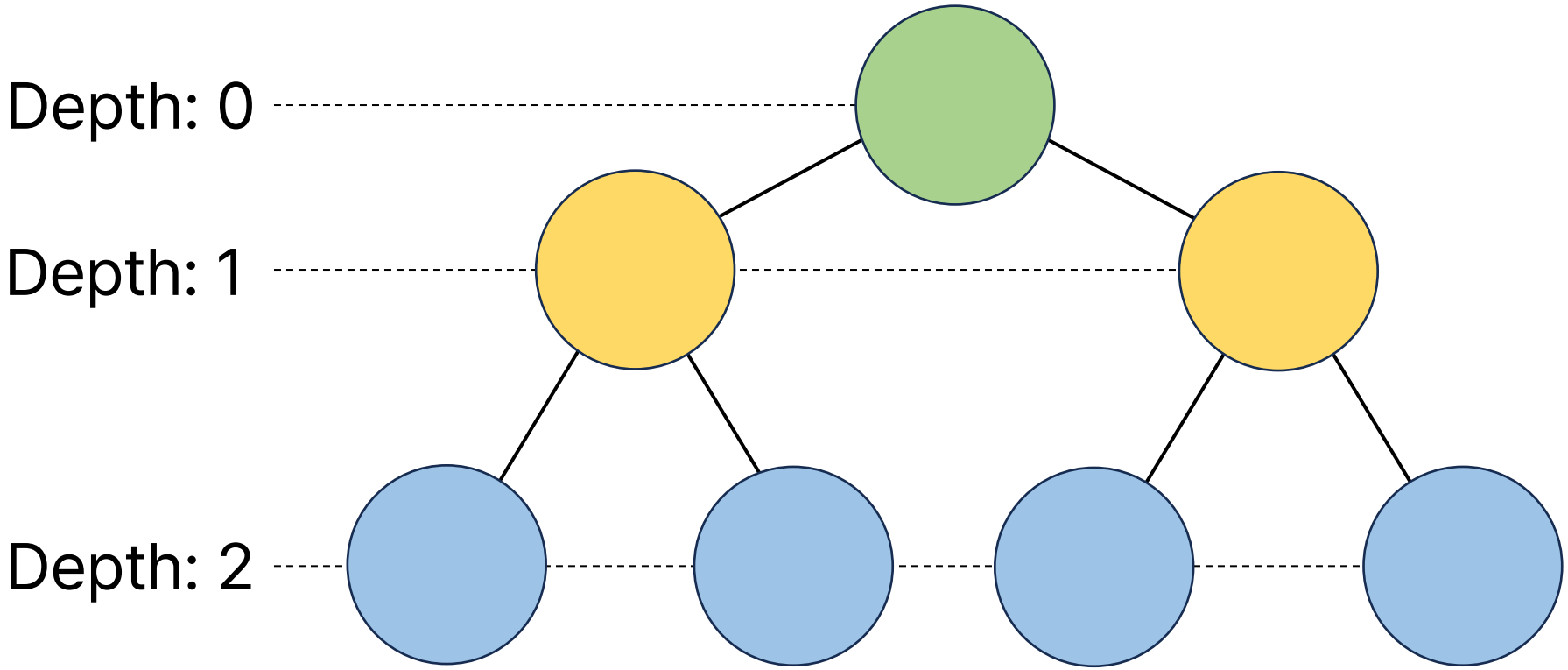
Subtree



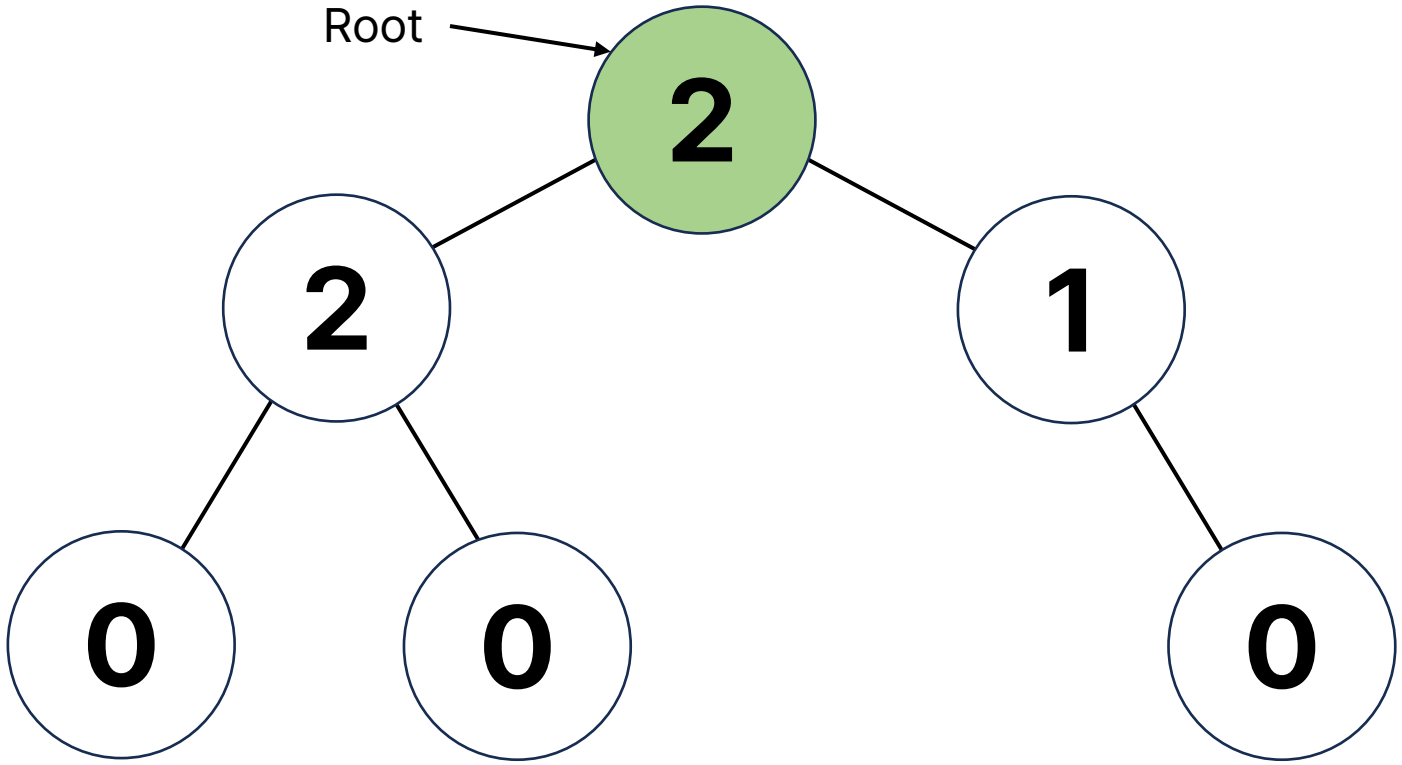
Trees

- Depth, 깊이: 조상의 수, 루트 노드는 0이고 자식 노드로 내려갈 때 마다 1씩 증가한다
- Level: 같은 깊이를 가지는 노드들의 집합
- Height, 높이: 깊이 중 최대값
- Degree, 차수: 노드의 차수, 트리의 차수로 나뉜다
- Degree of a node, 노드의 차수: 자식 노드의 개수
- Degree of a tree, 트리의 차수: 노드의 차수 중 최대값

Depth/Level



Degree



Trees

- 트리의 각 노드는 본인의 데이터와 자식 노드들을 가지고 있다

```
struct Node {
    int data;
    vector<Node *> child;

    Node() : data(0), child(vector<Node *>()) {}
    Node(int d) : data(d), child(vector<Node *>()) {}
};
```

Trees

- 부모와 자식의 관계가 정확하게 주어지는 경우, 부모 노드에 자식 노드를 추가한다

```
Node nodes[MAX_NODE];
int p, c;
for (int i = 0; i < MAX_NODE - 1; i++) {
    cin >> p >> c;
    nodes[p].child.push_back(&nodes[c]);
}
```

Trees

- 부모와 자식의 관계가 주어지지 않는 경우, 루트 노드에서 순회를 시작해 부모와 자식 관계를 찾아야한다
- 트리의 모든 노드는 부모로 가는 간선 하나를 가지고 있고, 그 외의 간선은 모두 자식 노드로 향하는 간선이다
- DFS나 BFS를 수행할 때 항상 부모 노드부터 방문 후 자식 노드를 방문한다
- 노드를 방문 했을 때, 연결되어 있는 노드들 중 방문한 노드는 나의 부모, 그 외에는 모두 자식 노드이다

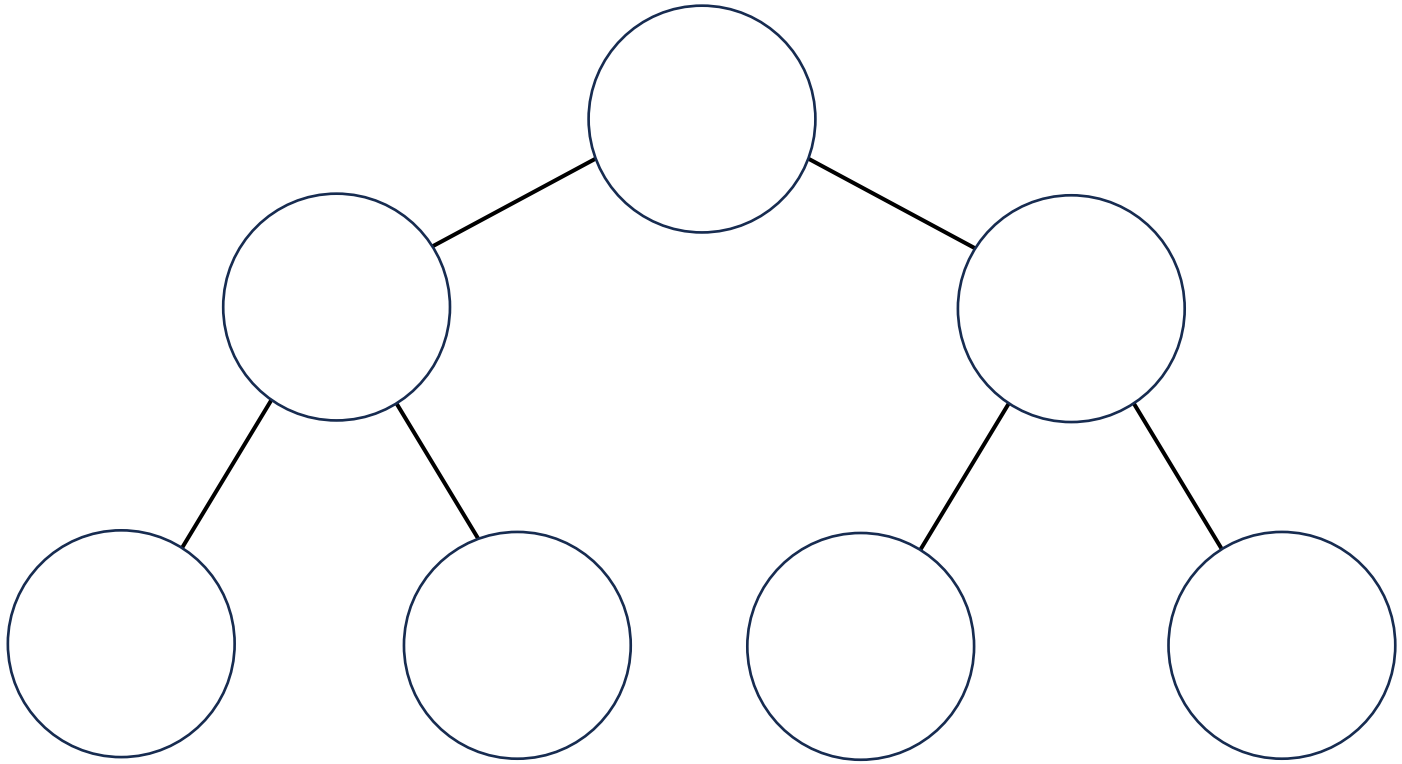
Trees

```
Node nodes[MAX_NODE];
vector<int> edges[MAX_NODE];
bool visited[MAX_NODE];

void dfs(int cur) {
    visited[cur] = true;
    for (auto next : edges[cur]) {
        if (visited[cur])
            continue;
        nodes[cur].child.push_back(&nodes[next]);
        dfs(next);
    }
}
```

Binary Trees

- 최대 2개의 자식 노드를 가진 트리를 이진 트리라고 한다
- 2개의 자식 노드를 각각 왼쪽 자식 노드 오른쪽 자식 노드라고 한다
- 리프 노드를 제외하고 모든 노드가 2개의 자식 노드를 가지는 트리를 완전 이진 트리라고 한다
- 완전 이진 트리는 높이가 h일 때, 2^h - 1개의 노드를 가진다

Binary Trees



Binary Trees

- 완전 이진 트리라면 루트부터 각 층의 노드 개수가 1, 2, 4, ... 이다
- 한 층 내려갈 때 마다 2배가 되므로 2^0, 2^1, 2^2, ...이다
- 따라서 높이가 h 일 때 $2^{h-1} + ... + 2^1 + 2^0$이므로 전체 노드 수는 2^h - 1개 이다

Traversal

- 자식을 순회하는 것과 본인을 순회하는 것의 순서에 따라 3가지로 나뉜다
- Pre-order: 자식보다 본인을 먼저 탐색하는 것, 본인 -> 왼쪽 자식 -> 오른쪽 자식 순서로 탐색한다
- In-order: 본인을 자식 탐색 중간에 하는 것, 왼쪽 자식 -> 본인 -> 오른쪽 자식 순서로 탐색한다
- Post-order: 본인보다 자식을 먼저 탐색하는 것, 왼쪽 자식 -> 오른쪽 자식 -> 본인 순서로 탐색한다

Traversal

```
struct Node {
    int data;
    Node *l, *r;
};
```

Pre-order

```
void pre_order(Node *x) {
    // do something
    x->data;

    if (x->l)
        pre_order(x->l);

    if (x->r)
        pre_order(x->r);
}
```

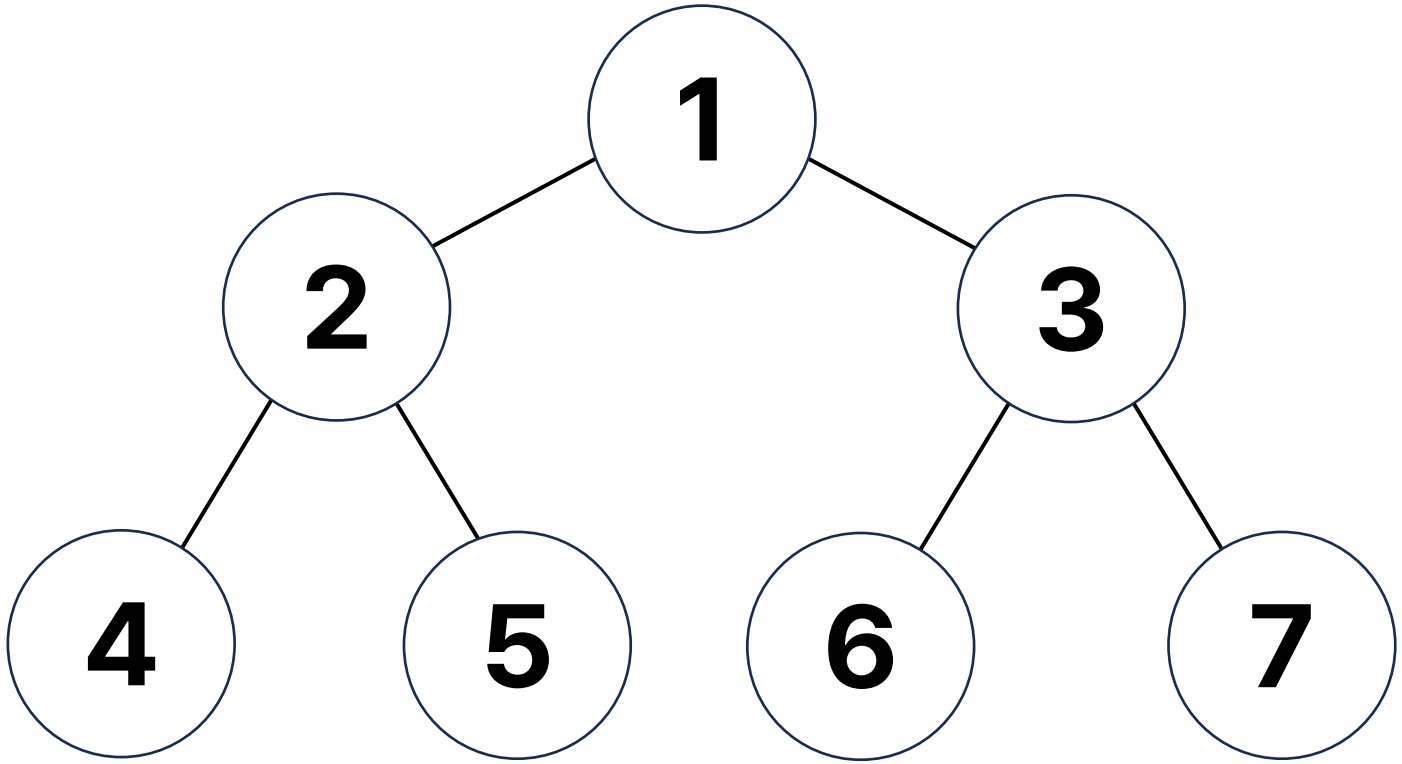
In-order

```
void in_order(Node *x) {
    if (x->l)
        in_order(x->l);

    // do something
    x->data;

    if (x->r)
        in_order(x->r);
}
```

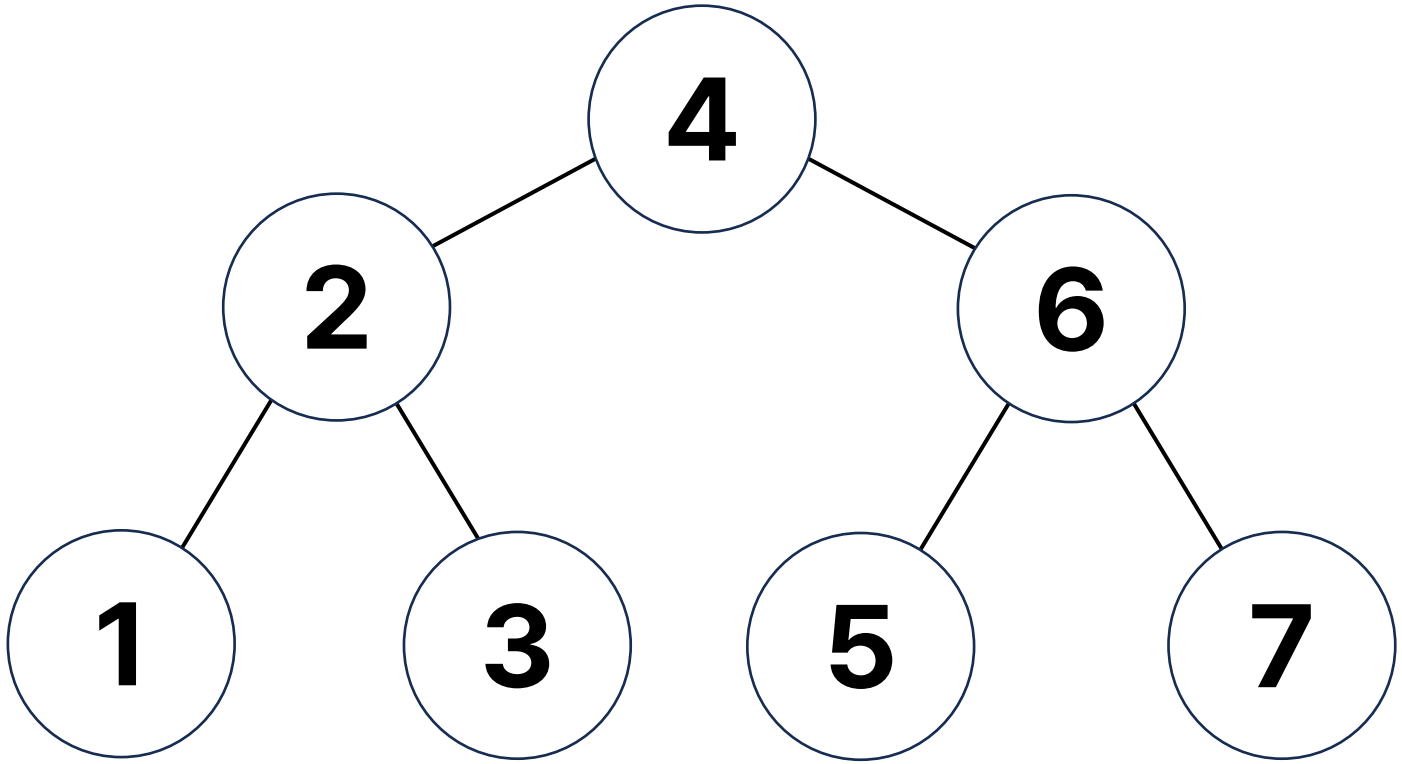
Post-order

```
void post_order(Node *x) {
    if (x->l)
        post_order(x->l);

    if (x->r)
        post_order(x->r);

    // do something
    x->data;
}
```

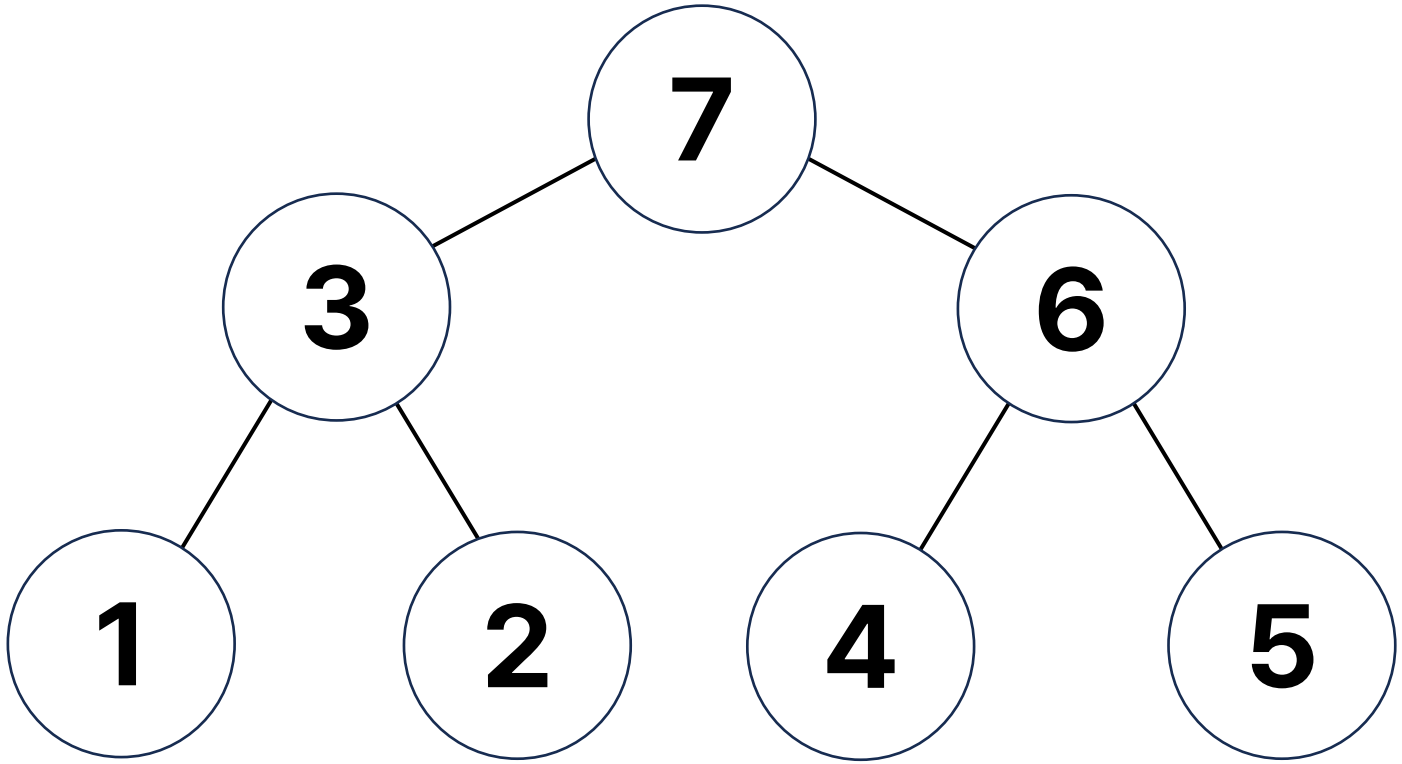
Pre-order



In-order



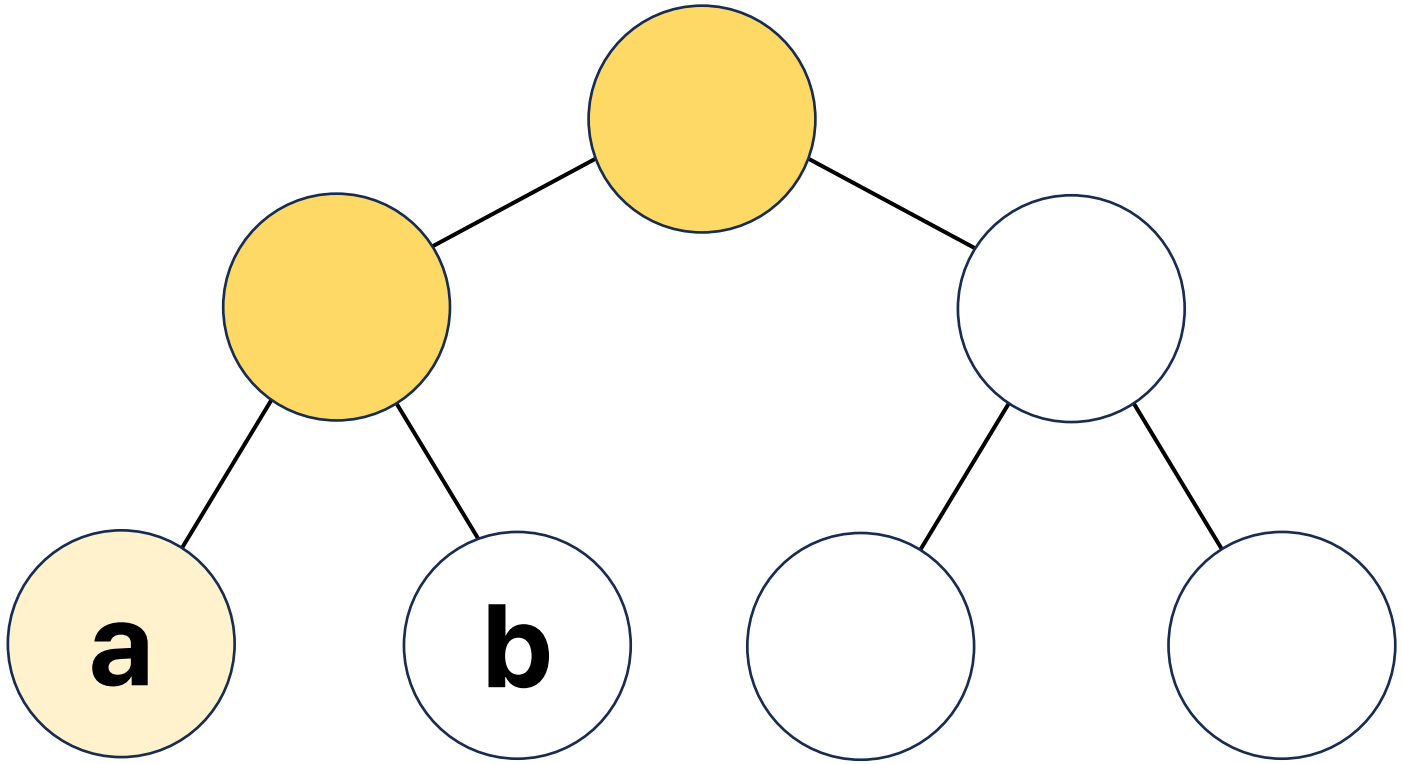
Post-order



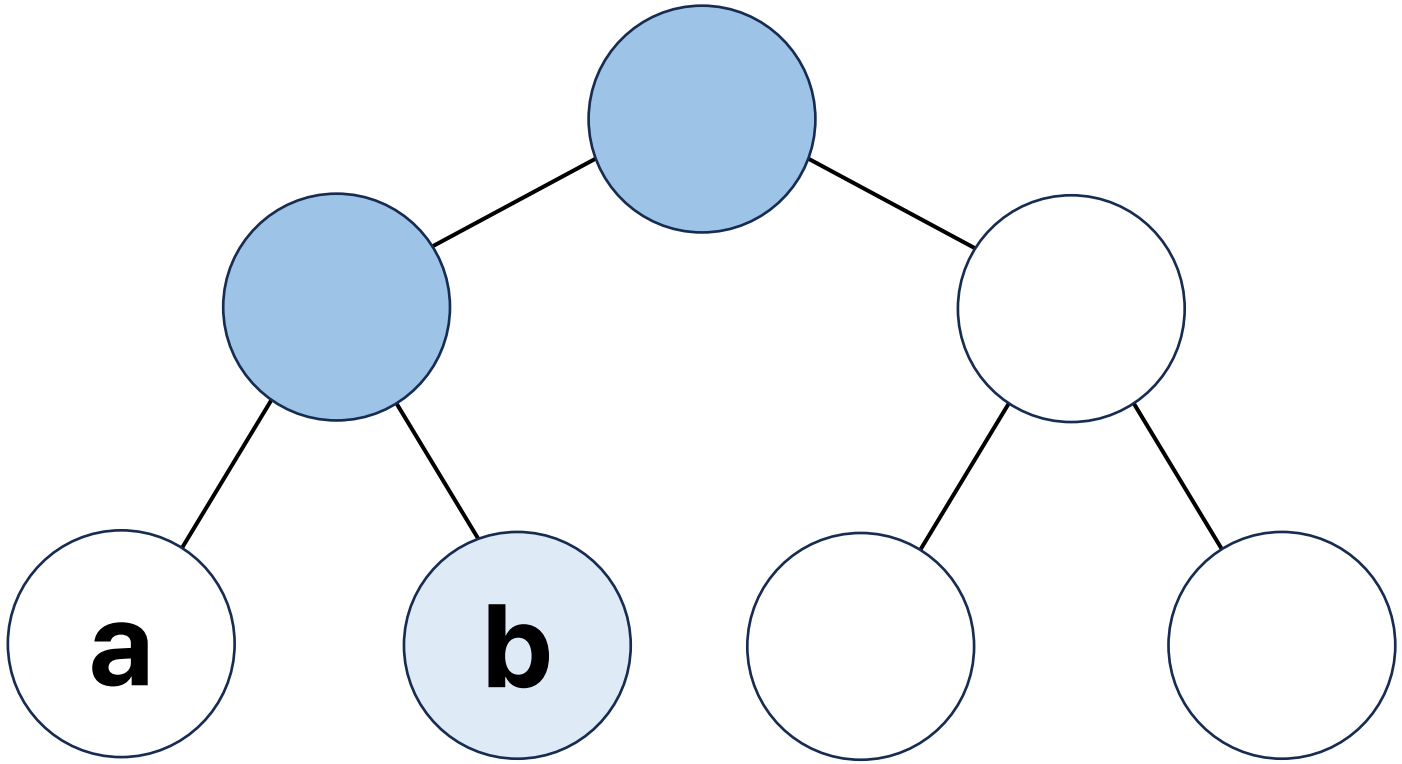
Lowest Common Ancestor

- 두 노드의 공통 조상들 중 두 노드에 가장 가까운 노드를 최소 공통 조상이라고 한다
- 브루트포스로 탐색한다면 모든 노드를 탐색해야 하므로 O(n)의 시간 복잡도
- 개선된 알고리즘을 사용하면 O(log n)에 탐색할 수 있다

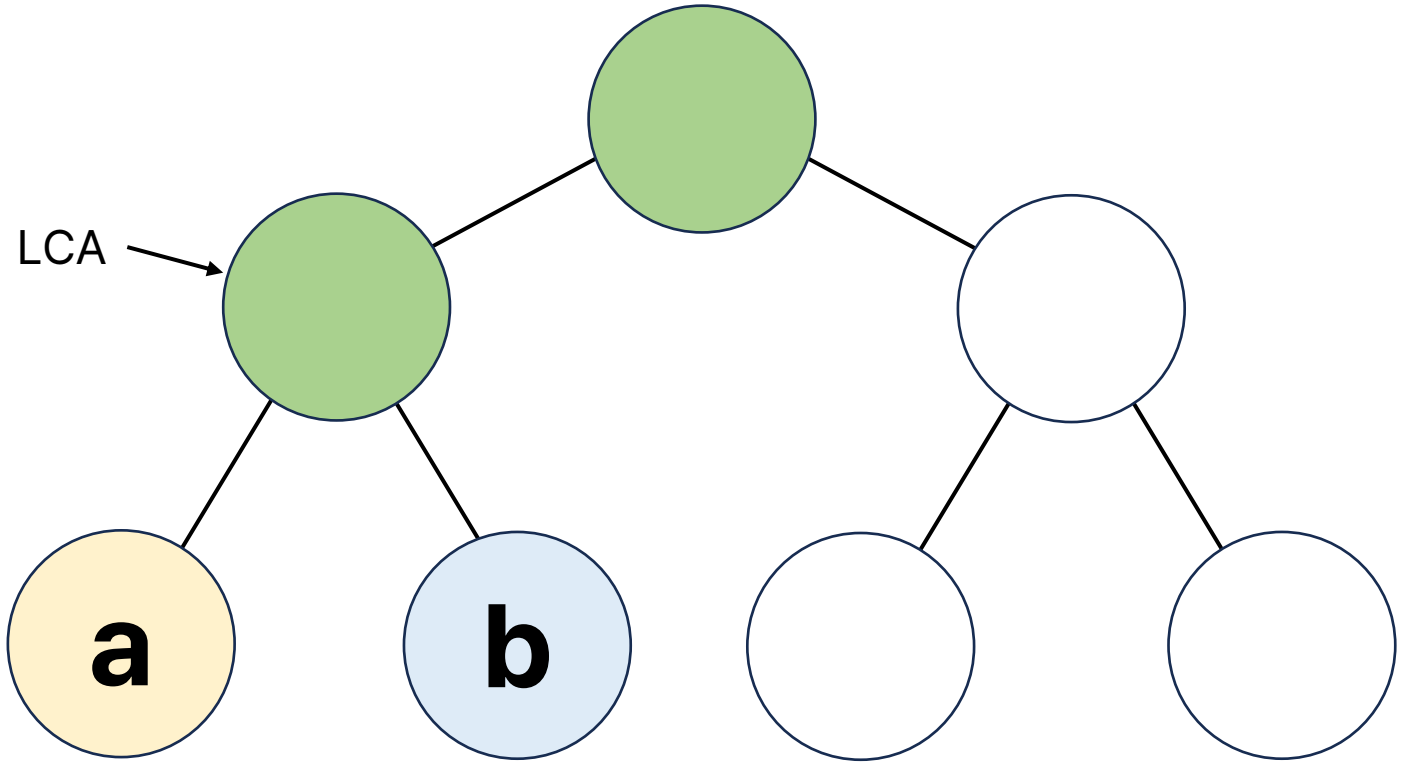
Lowest Common Ancestor



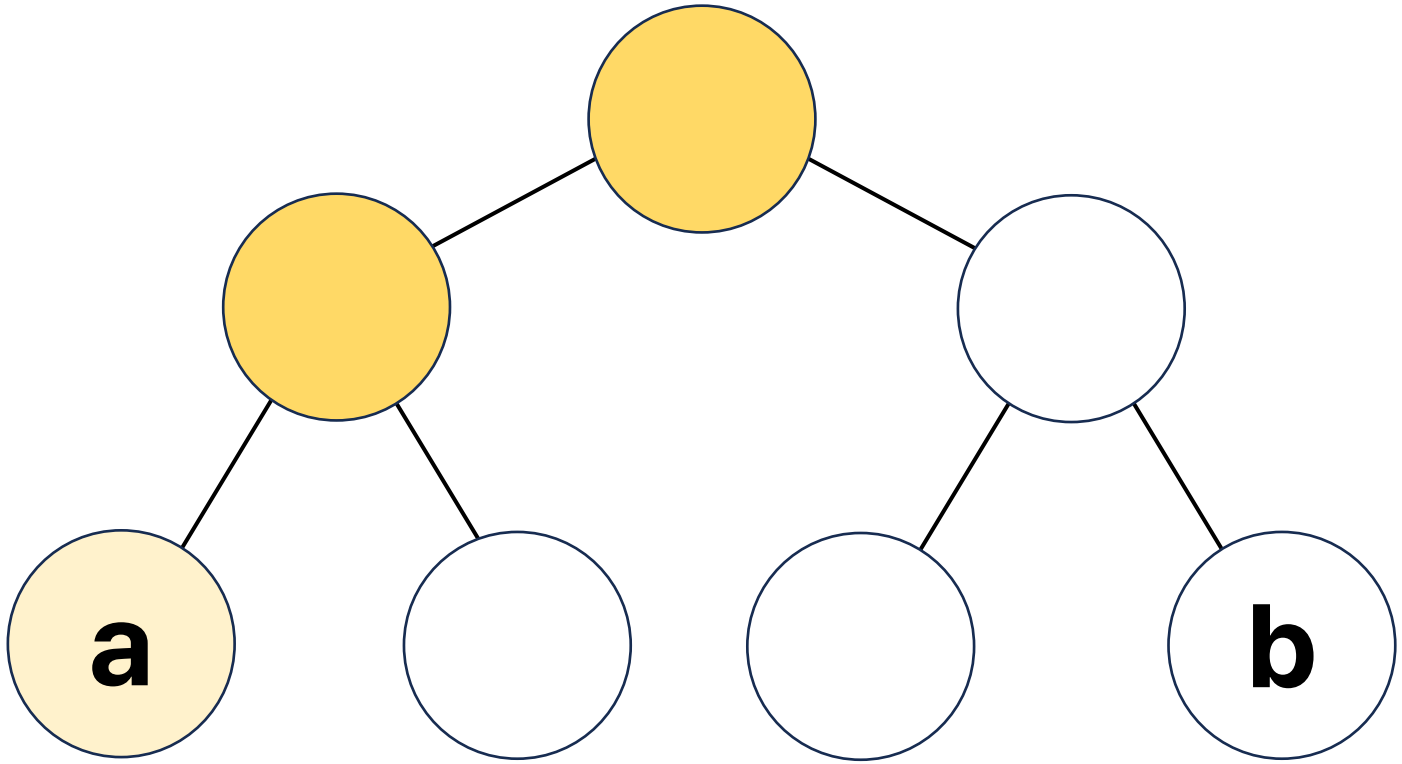
Lowest Common Ancestor



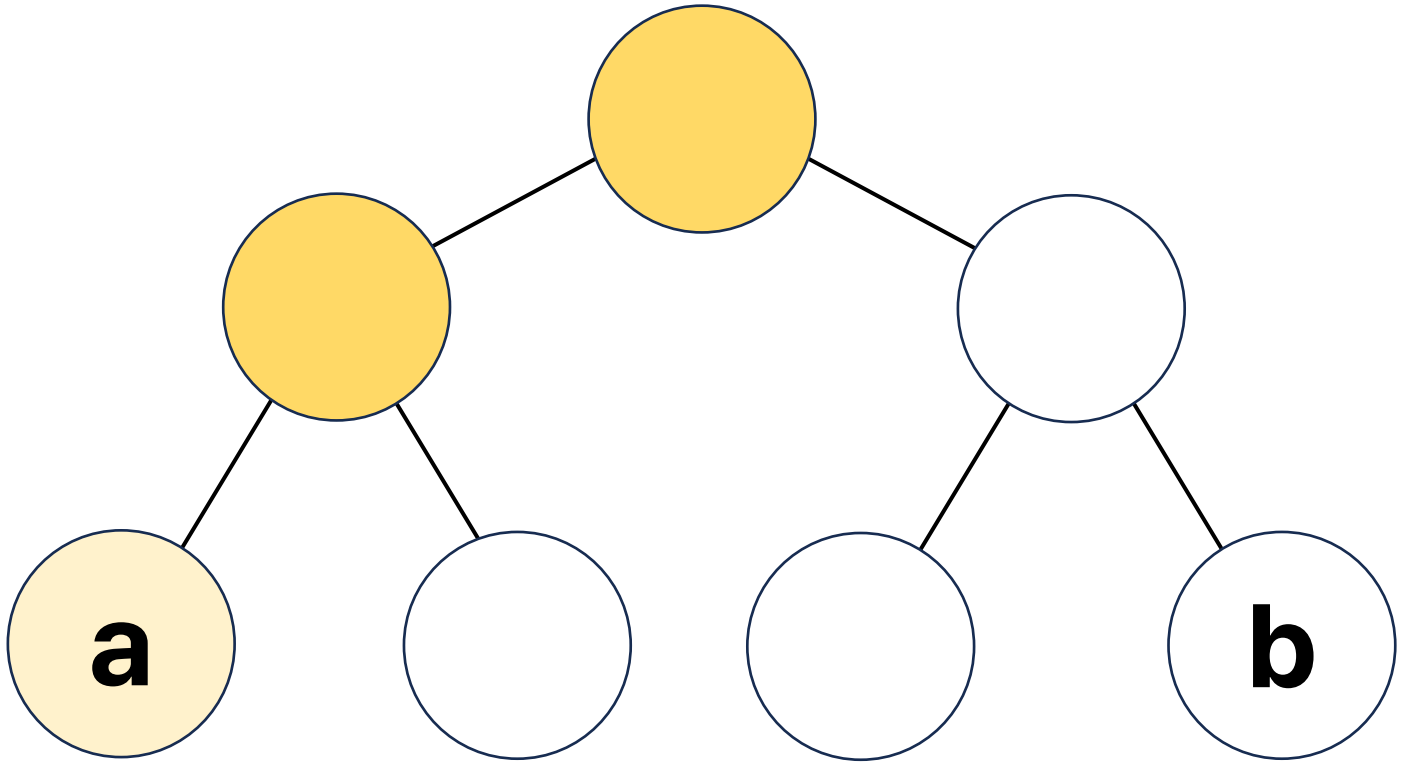
Lowest Common Ancestor



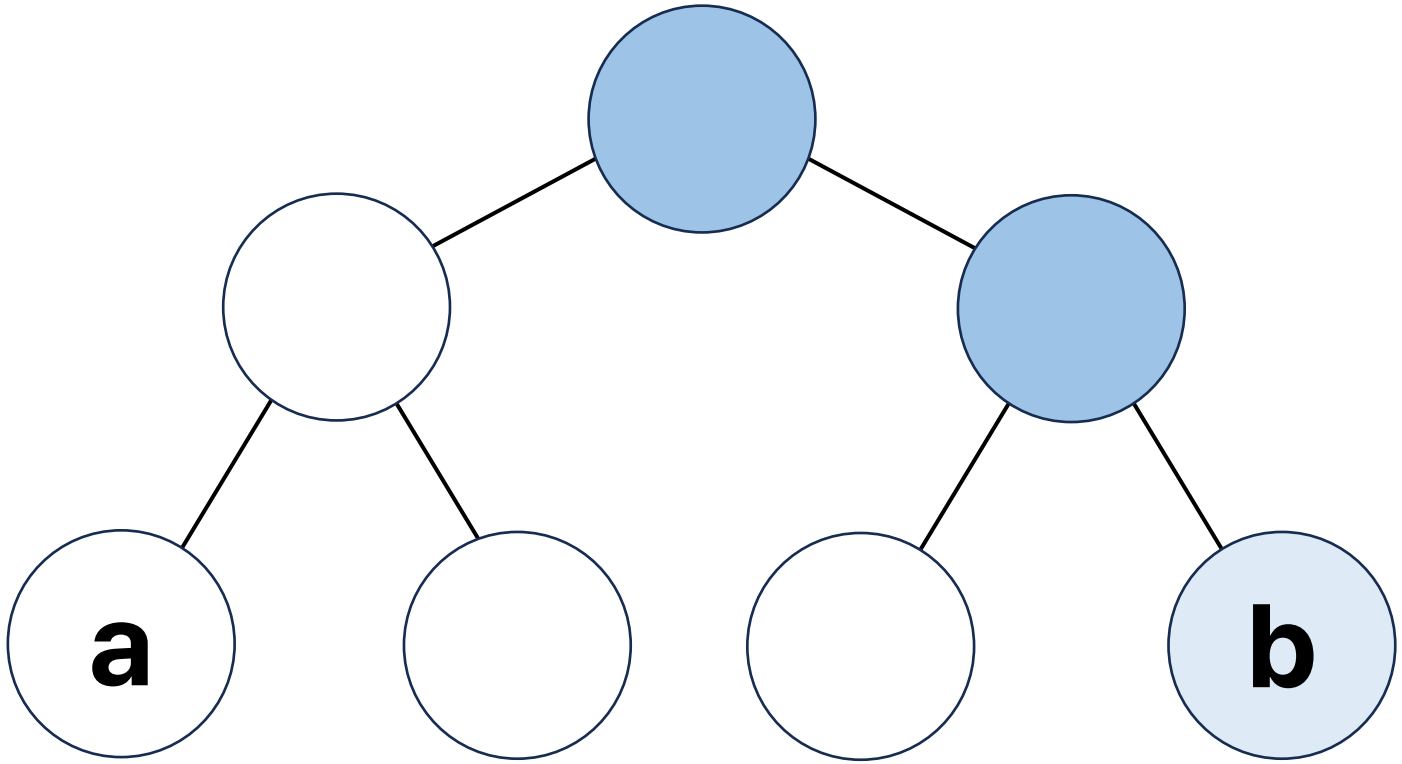
Lowest Common Ancestor



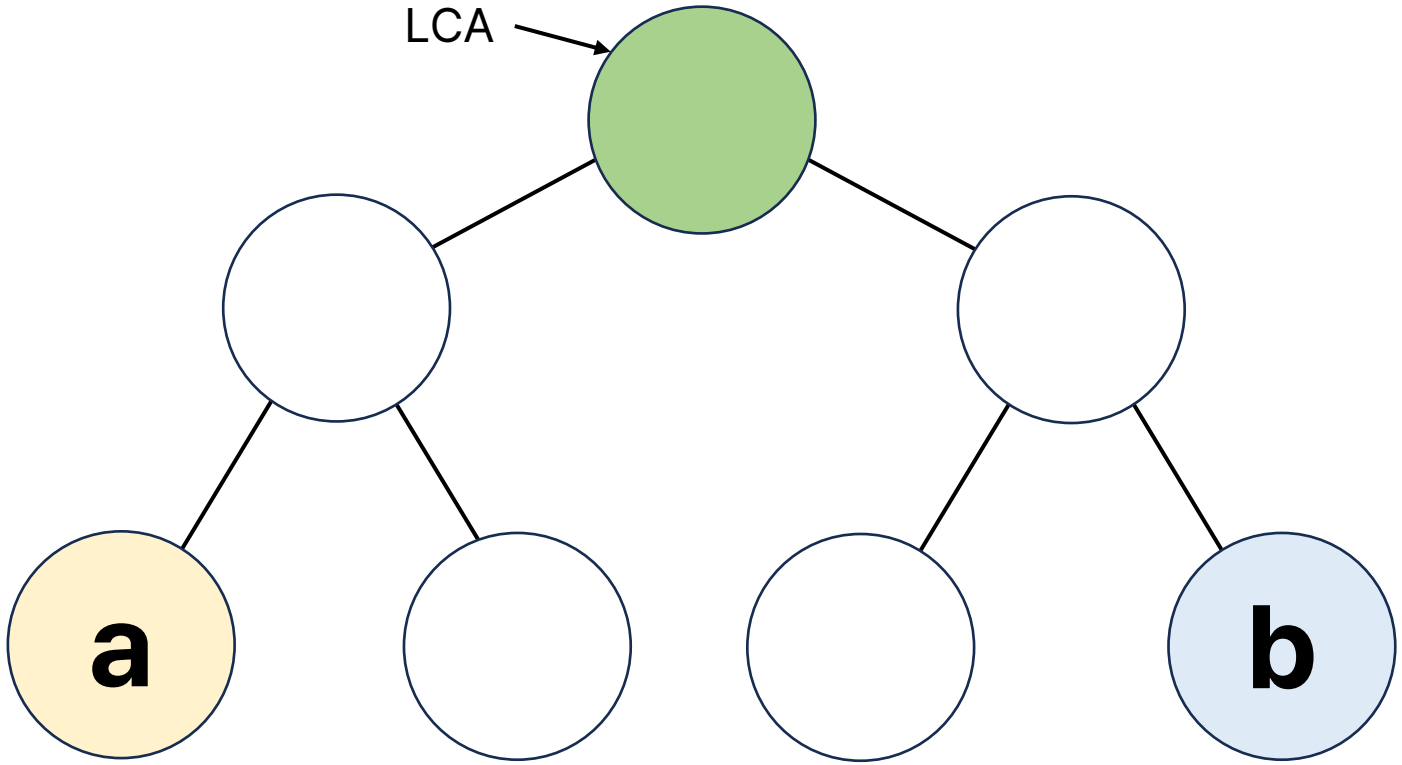
Lowest Common Ancestor



Lowest Common Ancestor



Lowest Common Ancestor



문제

- 트리 순회 BOJ 1991
- 트리의 부모 찾기 BOJ 11725
- 완전 이진 트리 BOJ 9934
- 거리가 k이하인 트리 노드에서 사과 수확하기 BOJ 25516
- 부동산 다툼 BOJ 20364