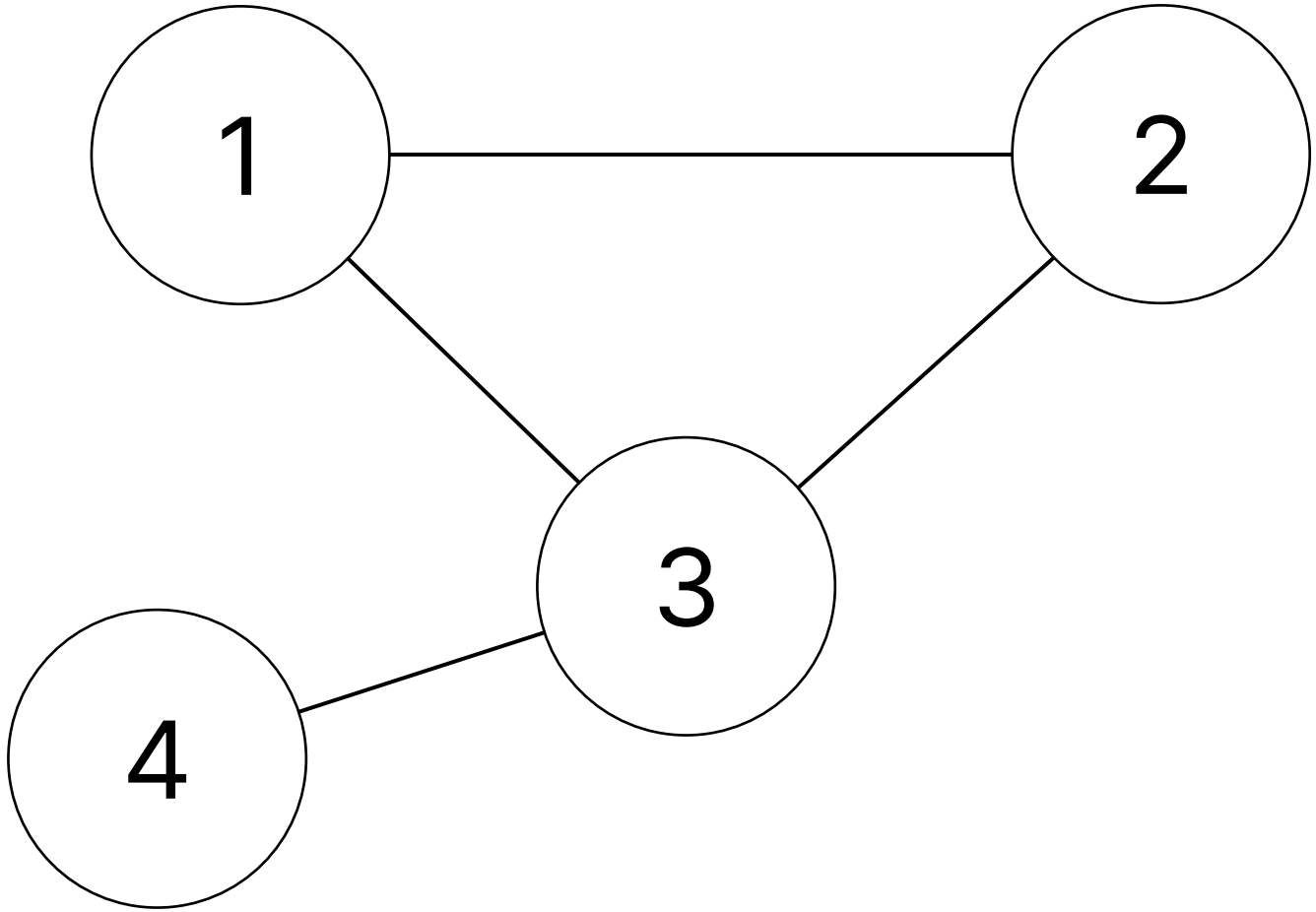
Graph

Graph

- 정점(Node, Vertex)과 간선(Edge)으로 이루어진 자료구조
- 지금까지는 단순히 변수 또는 구조체의 값만 사용했다면 그래프는 변수와 변수의 관계를 표현한 자료구조
- 네트워크 모델이라고 한다
- ex) 지하철 노선도의 표현, SNS 팔로우, 친구 관계

Graph



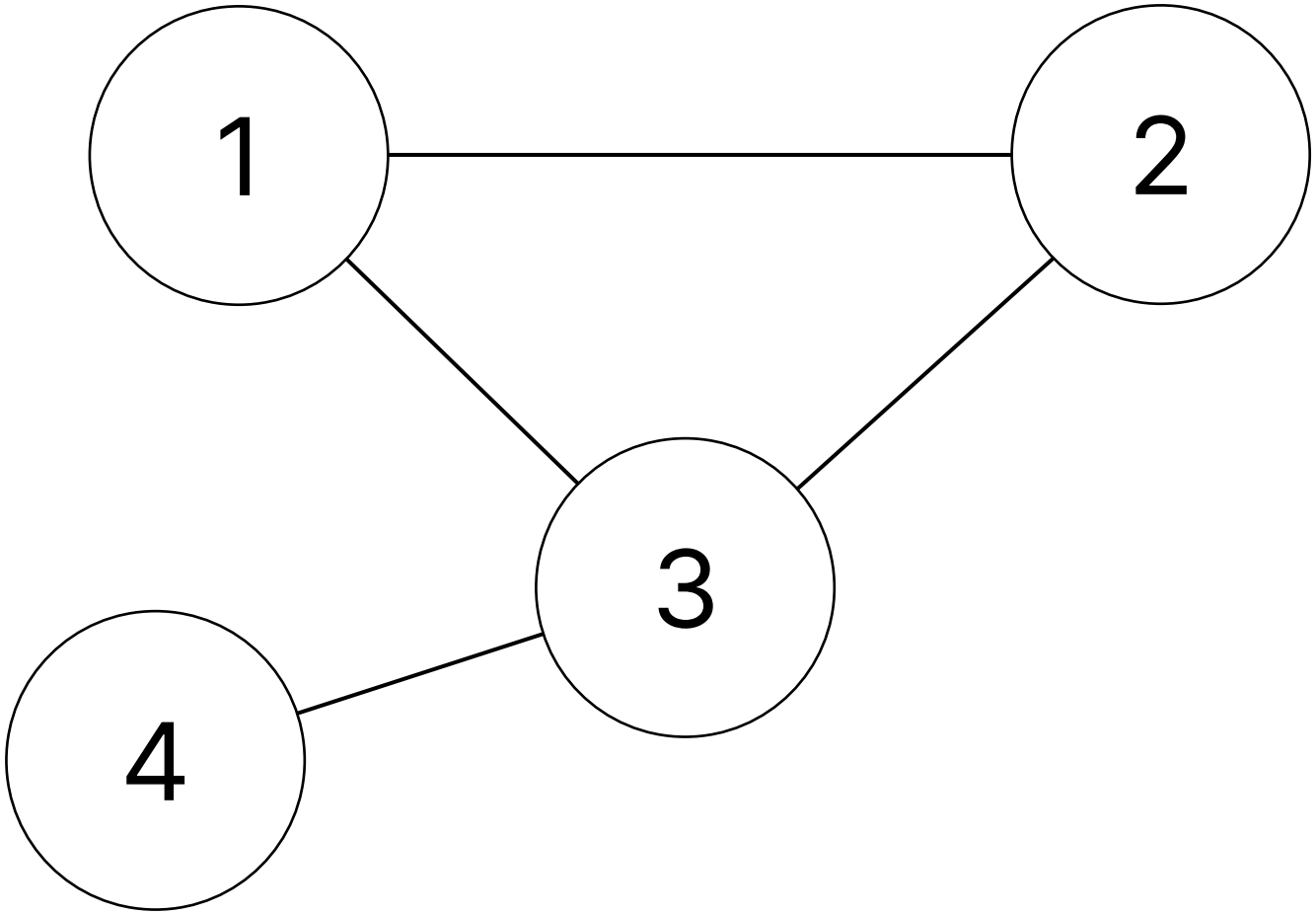
Directed, Undirected Graph

- 간선의 방향성의 유무
- 간선의 방향성이 존재

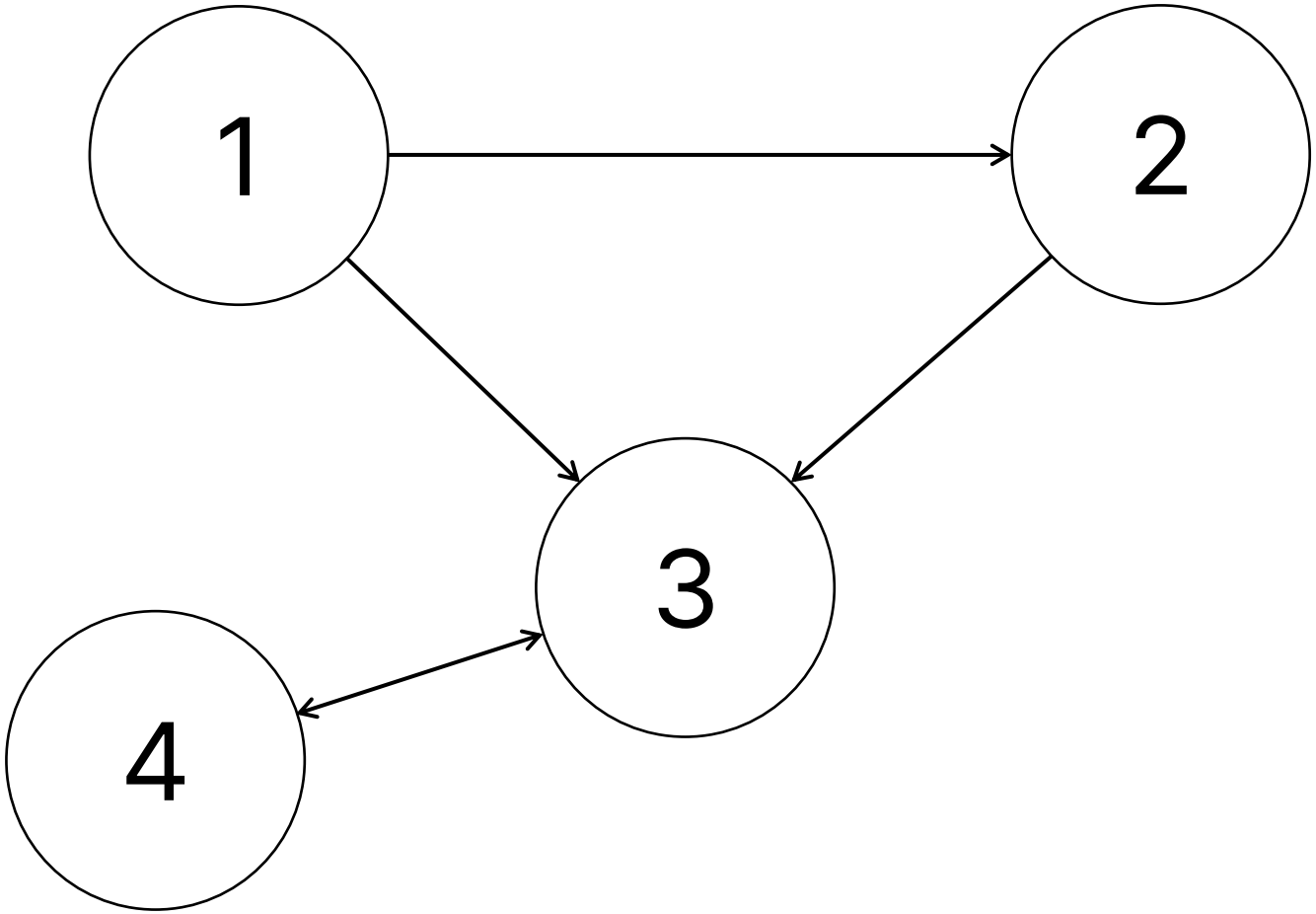
  : Directed Graph, 유향 그래프, 방향 그래프
- 간선의 방향성이 존재하지 않다, 간선이 양방향

  : Undirected Graph, 무향 그래프, 양방향 그래프

Undirected Graph

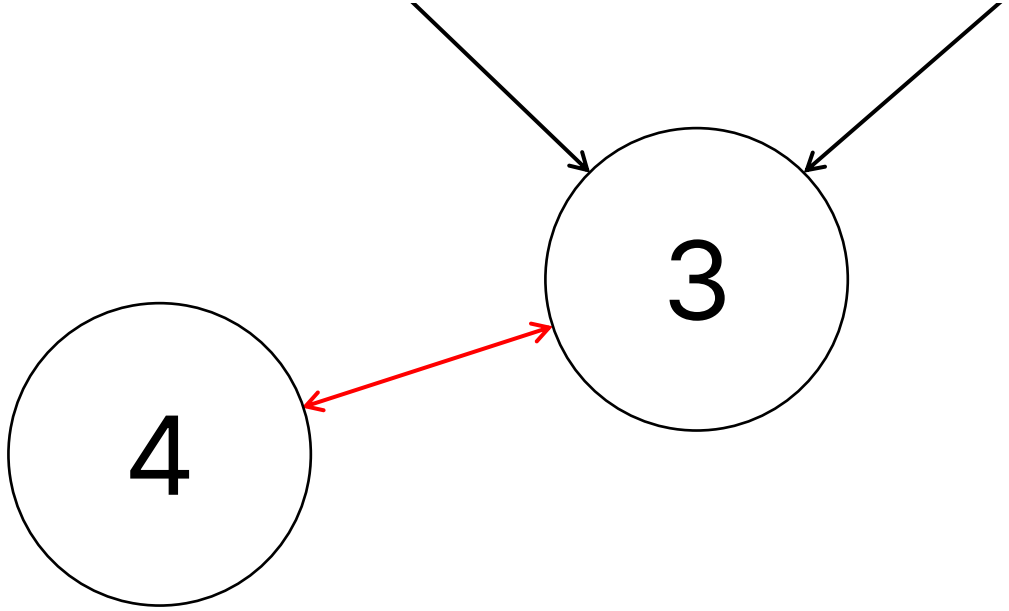


Directed Graph



Directed Graph

- 방향 그래프에서 양방향인 것은 간선이 2개 존재하는 것이다
- 3->4 간선과 4->3 간선이 같이 존재하는 것



Directed, Undirected Graph

- 차도는 일반적으로 양방향이므로 Undirected라고 생각할 수 있다
- 차도를 한 방향씩 살펴본다면 일방통행의 경우 간선이 1개, 양방향의 경우 간선이 2개 존재하는 Directed라고 생각할 수 있다
- 인도는 도로에 사람들의 방향을 지정하지 않았으므로 Undirected이다
- 물이 흐르는 파이프도 양방향으로 흐를 수 있으므로 Undirected이다

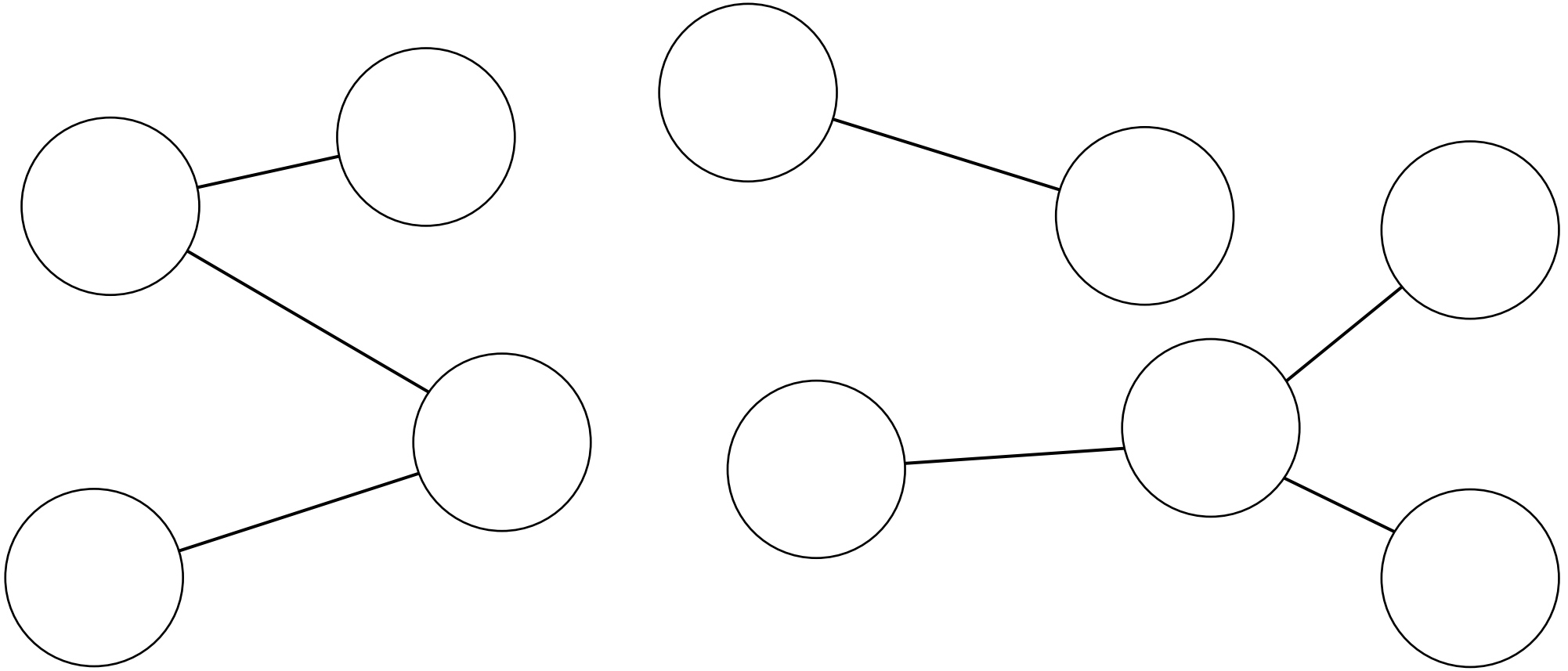
Directed, Undirected Graph

- Instagram(SNS)에서 팔로우를 하는 것을 생각해보자
- 사람은 노드, 팔로우를 간선으로 생각할 수 있다
- 처음 상태는 간선이 없는 노드 하나이다
- A가 B를 팔로우를 하는 경우 A에서 B로 향하는 간선을 추가한 것이라고 생각할 수 있다
- 맞팔은 간선이 2개 존재하는 형태라 생각할 수 있다

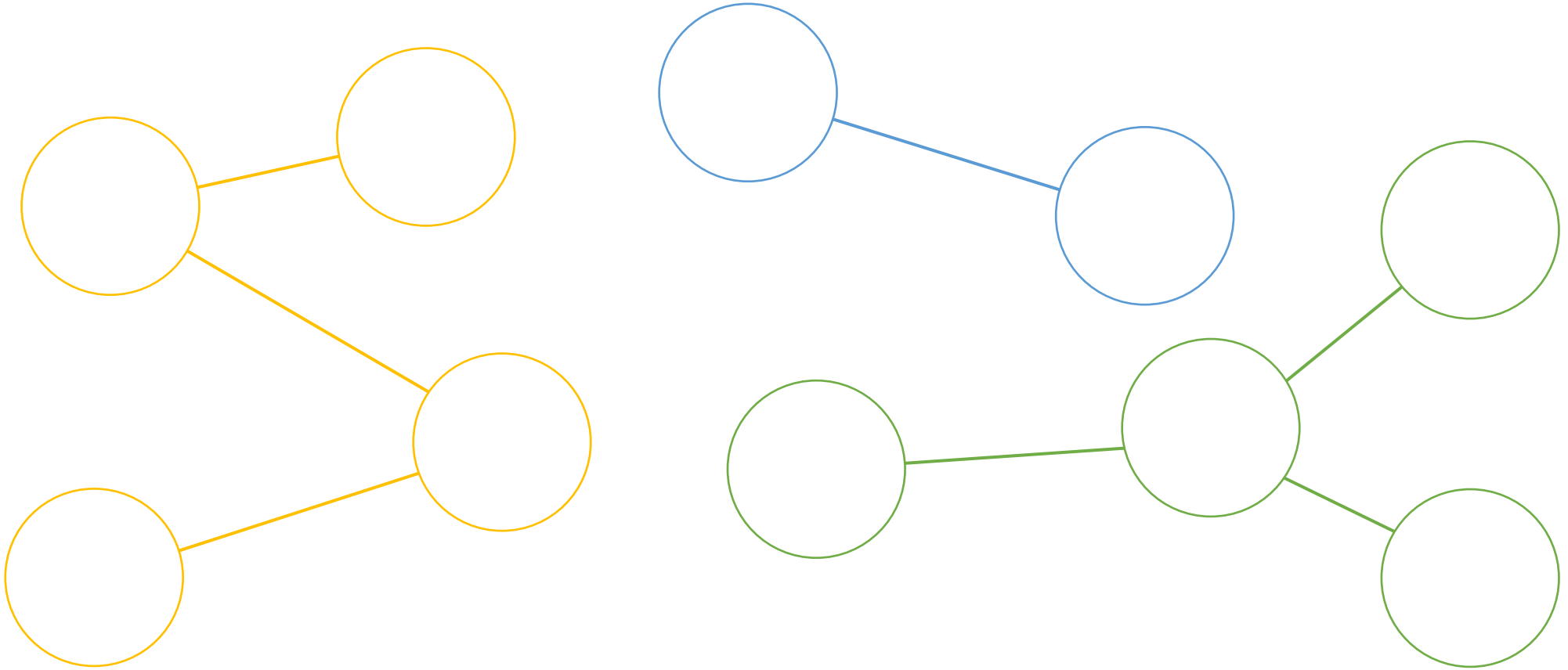
Connected Component

- 간선들로 연결된 노드들의 집합을 연결 요소라고 부른다
- 연결 요소가 몇 개인가? -> 몇 개의 묶음으로 나눌 수 있는가

Connected Component

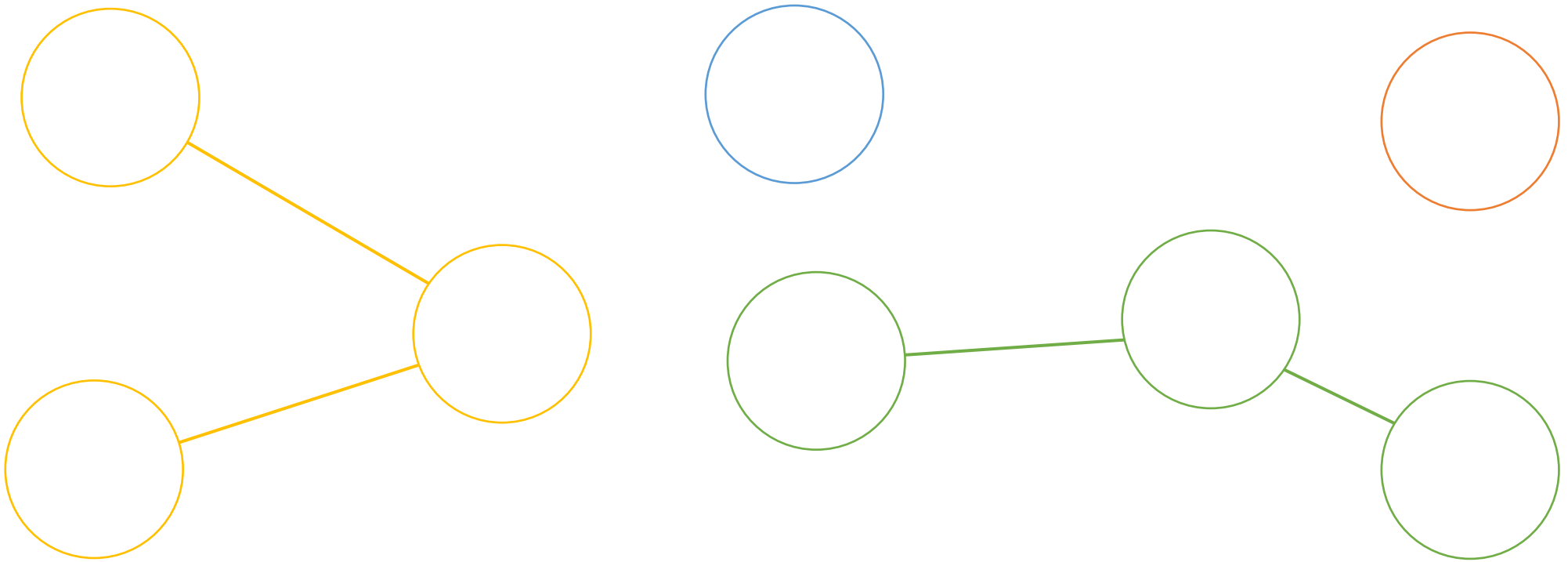


Connected Component



Connected Component

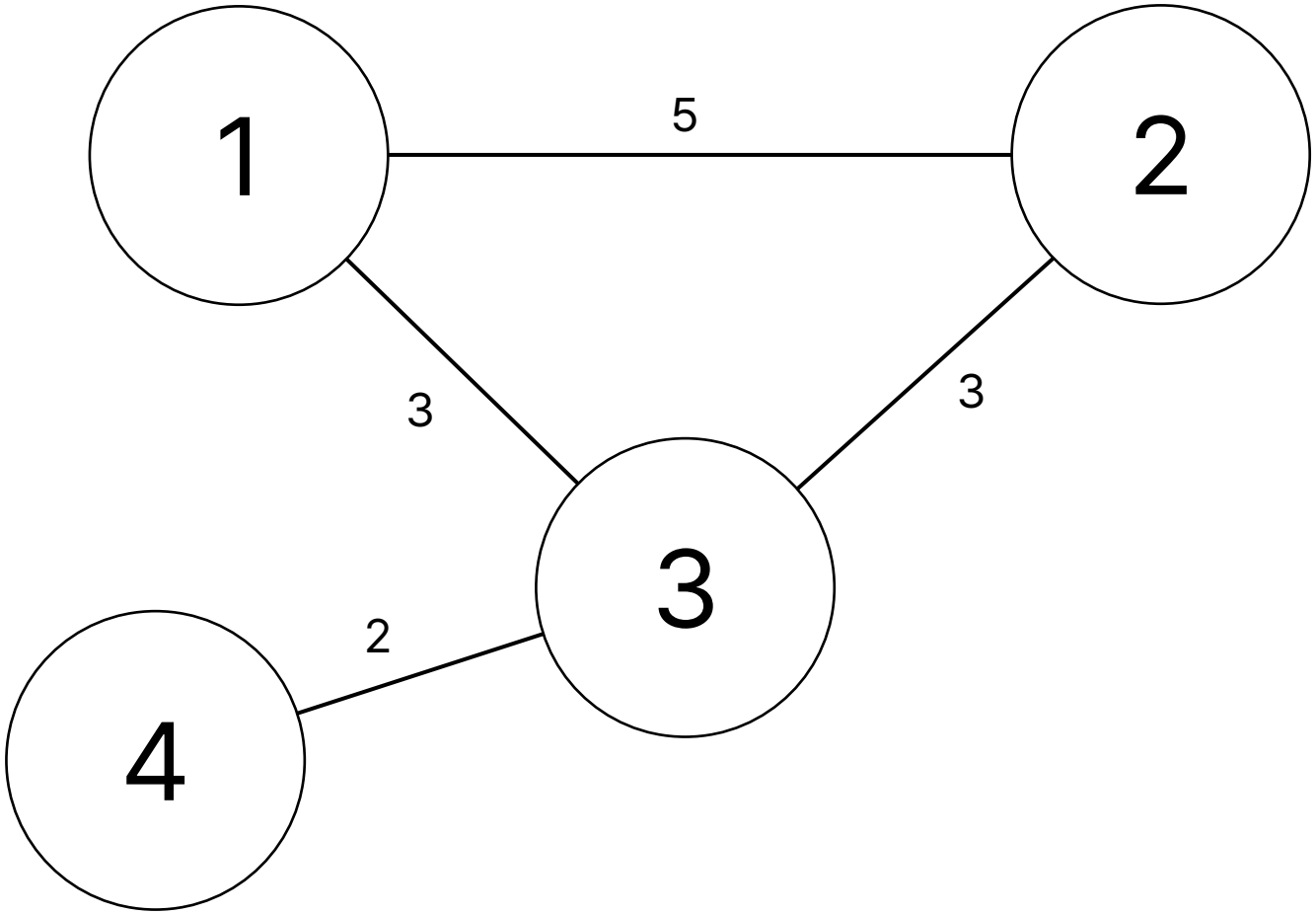
- 노드에 간선이 존재하지 않는 경우, 노드 하나가 하나의 연결 요소다



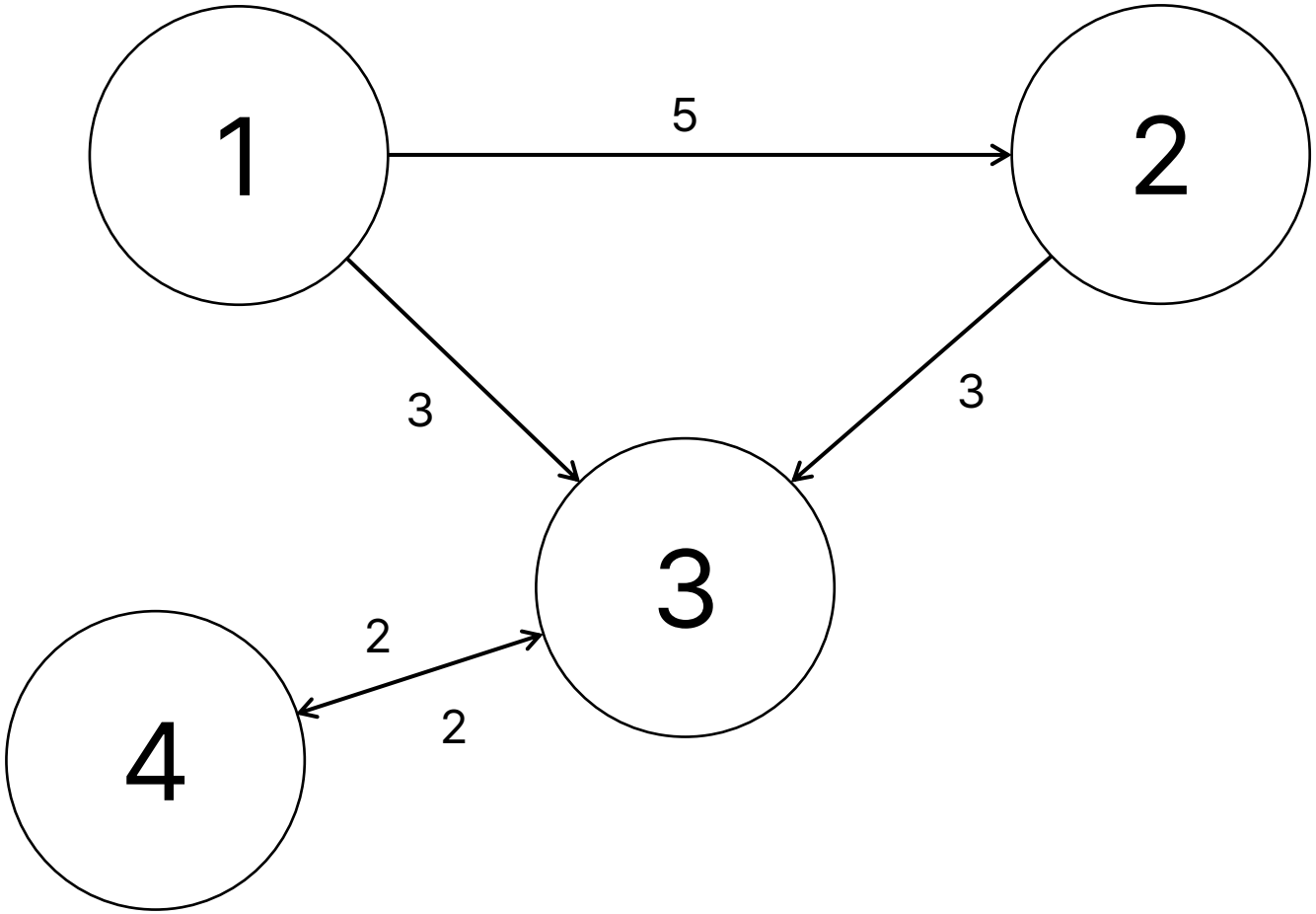
Weight

- 각 간선에는 가중치가 존재할 수 있다
- 간선의 가중치가 존재하지 않는 것은 모든 간선의 가중치가 동일하다고 생각할 수 있다
- ex) 도로를 간선으로 생각하면 단순히 지도를 표시한 것은 가중치가 없는 그래프, 거리, 소요 시간, 통행료 등을 추가한다면 이들을 가중치로 표현할 수 있다

Weight



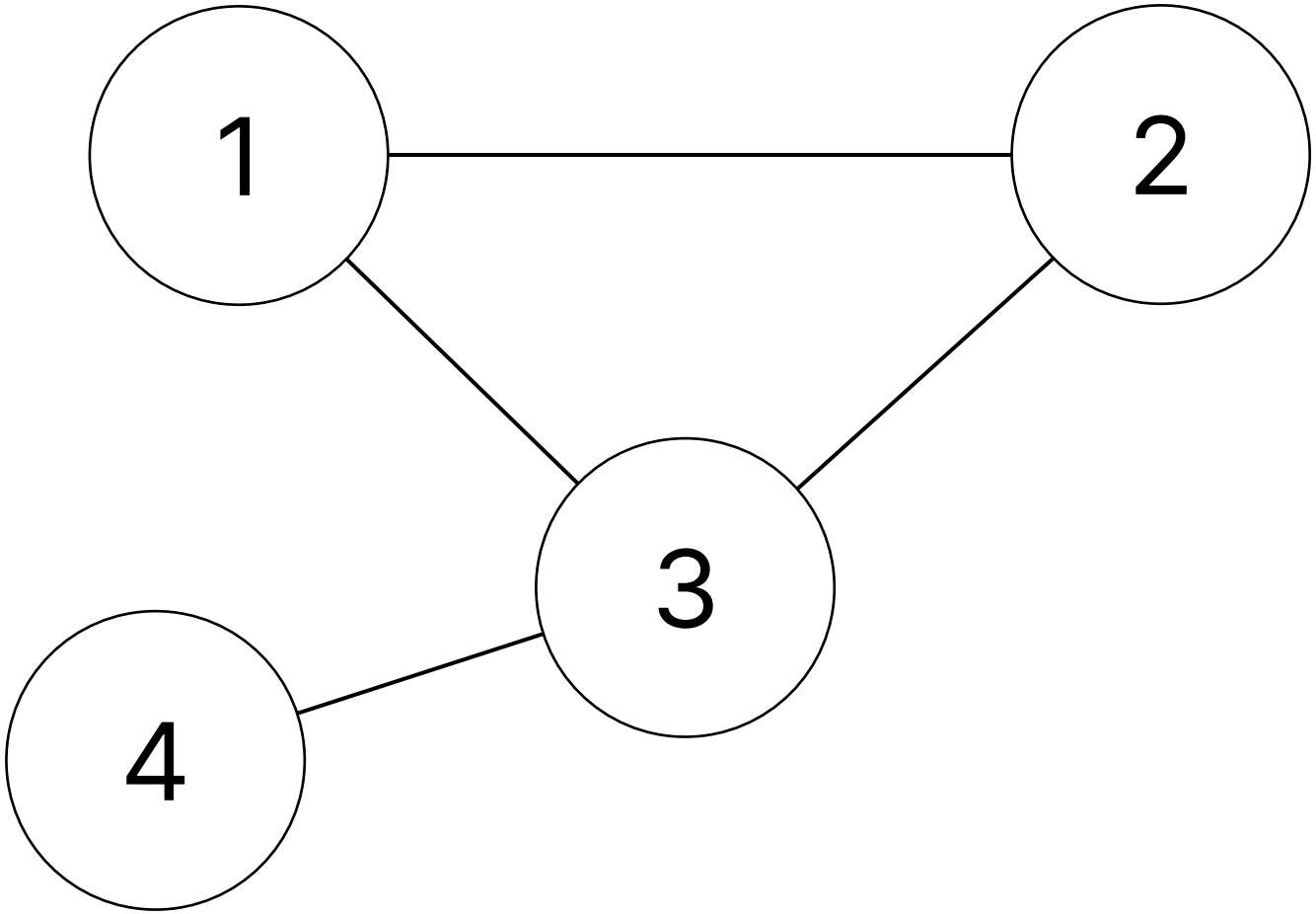
Weight



Degree

- 차수는 정점에 연결되어 있는 간선의 수를 의미한다
- 양방향 그래프에서는 간선의 방향성이 없기 때문에 노드와 연결된 간선의 수가 곧 차수이다
- 방향 그래프에서는 진입 차수(In degree)와 진출 차수(Out degree)로 나뉜다
- 진입 차수: 간선의 도착점이 해당 노드인 간선들의 수
- 진출 차수: 간선의 출발점이 해당 노드인 간선들의 수

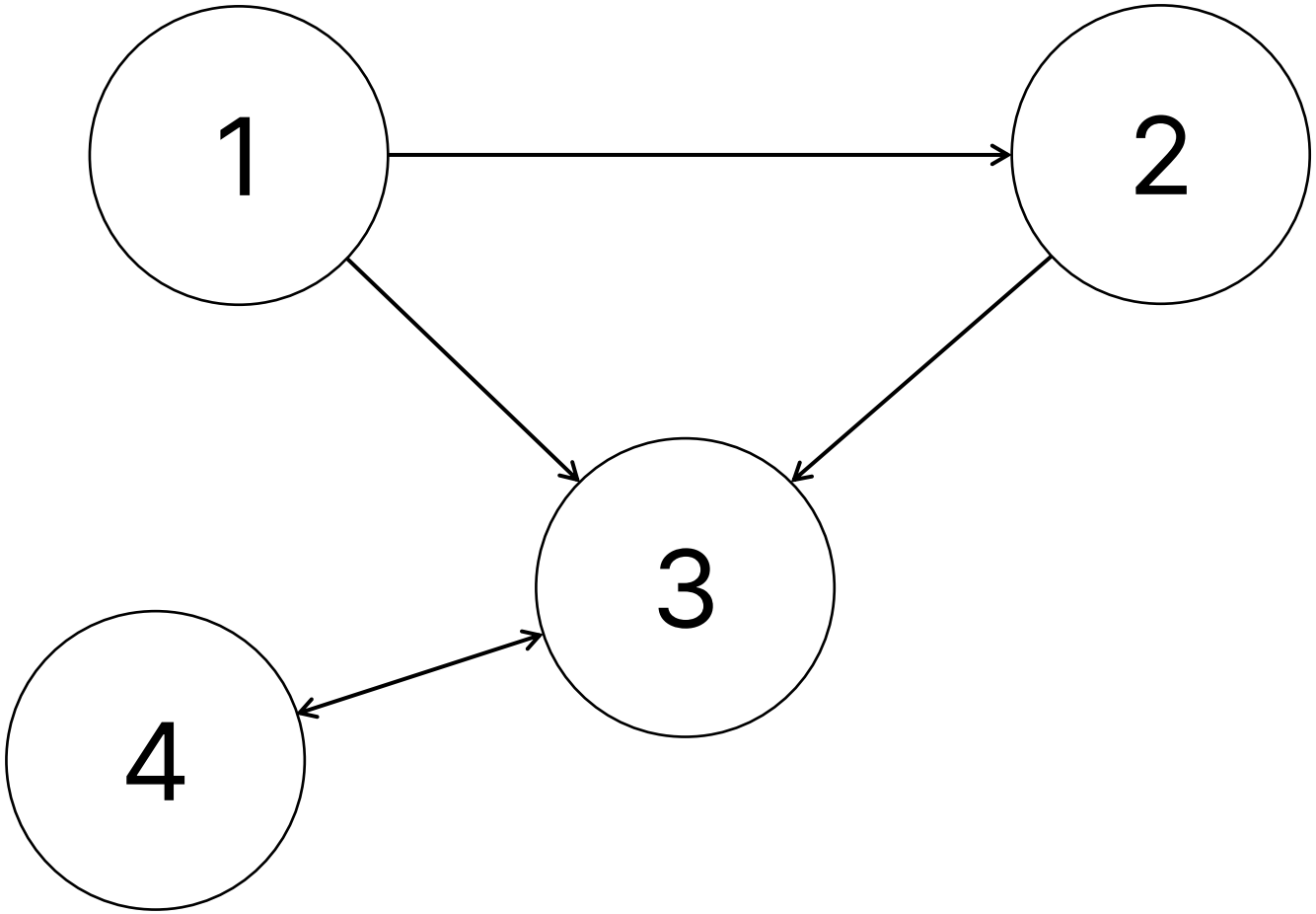
Degree



Degree

- 1번 노드와 연결된 간선은 2개, 1번 노드의 Degree는 2
- 2번 노드와 연결된 간선은 2개, 2번 노드의 Degree는 2
- 3번 노드와 연결된 간선은 3개, 3번 노드의 Degree는 3
- 4번 노드와 연결된 간선은 1개, 4번 노드의 Degree는 1

Degree



Degree

- 1번 노드로 들어오는 간선은 0개, 1번 노드의 In Degree는 0
- 1번 노드에서 나가는 간선은 2개, 1번 노드의 Out Degree는 2
- 2번노드 In Degree: 1, Out Degree: 1
- 3번노드 In Degree: 3, Out Degree: 1
- 4번노드 In Degree: 1, Out Degree: 1

그래프의 종류

- 숲(Forest): 사이클이 없는 무향 그래프
- 트리(Tree): 사이클이 없고 연결 요소가 하나인 그래프
- 완전 그래프(Complete Graph): 모든 정점이 서로 인접하게 연결된 그래프
- 이분 그래프(Bipartite Graph): 인접한 노드를 서로 다른 집합으로 분리해 모든 노드들을 두 가지 집합으로 분할할 수 있는 그래프
- Directed Acyclic Graph, DAG: 사이클이 없는 방향 그래프

그래프의 표현 방법

- 해당 노드와 인접, 직접적으로 연결된 노드들로 그래프를 표현

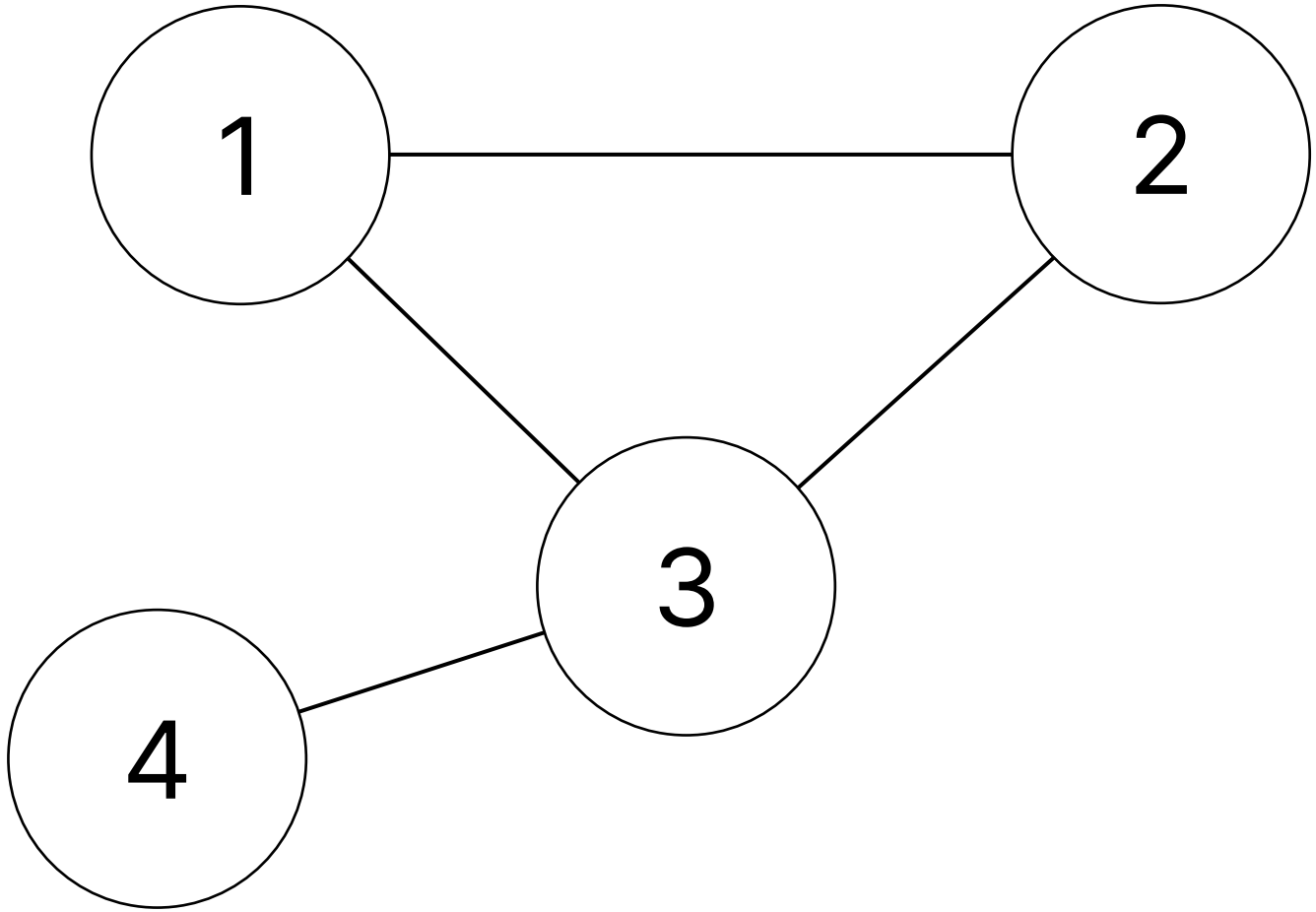
- 인접 리스트(Adjacency List)
- 인접 행렬(Adjacency Matrix)

+ 간선 리스트

Adjacency List

- 각 노드마다 리스트가 존재
- 각 리스트는 해당 노드와 인접한 노드들을 저장한다
- 시작점과 끝점이 중복 되는 간선을 저장할 수 있다

Adjacency List



Adjacency List

- 1번 노드와 인접한 노드: 2, 3
- 2번 노드와 인접한 노드: 1, 3
- 3번 노드와 인접한 노드: 1, 2, 4
- 4번 노드와 인접한 노드: 3

Adjacency List

1	1번 노드와 인접한 노드들
2	2번 노드와 인접한 노드들
3	3번 노드와 인접한 노드들
4	4번 노드와 인접한 노드들

Adjacency List

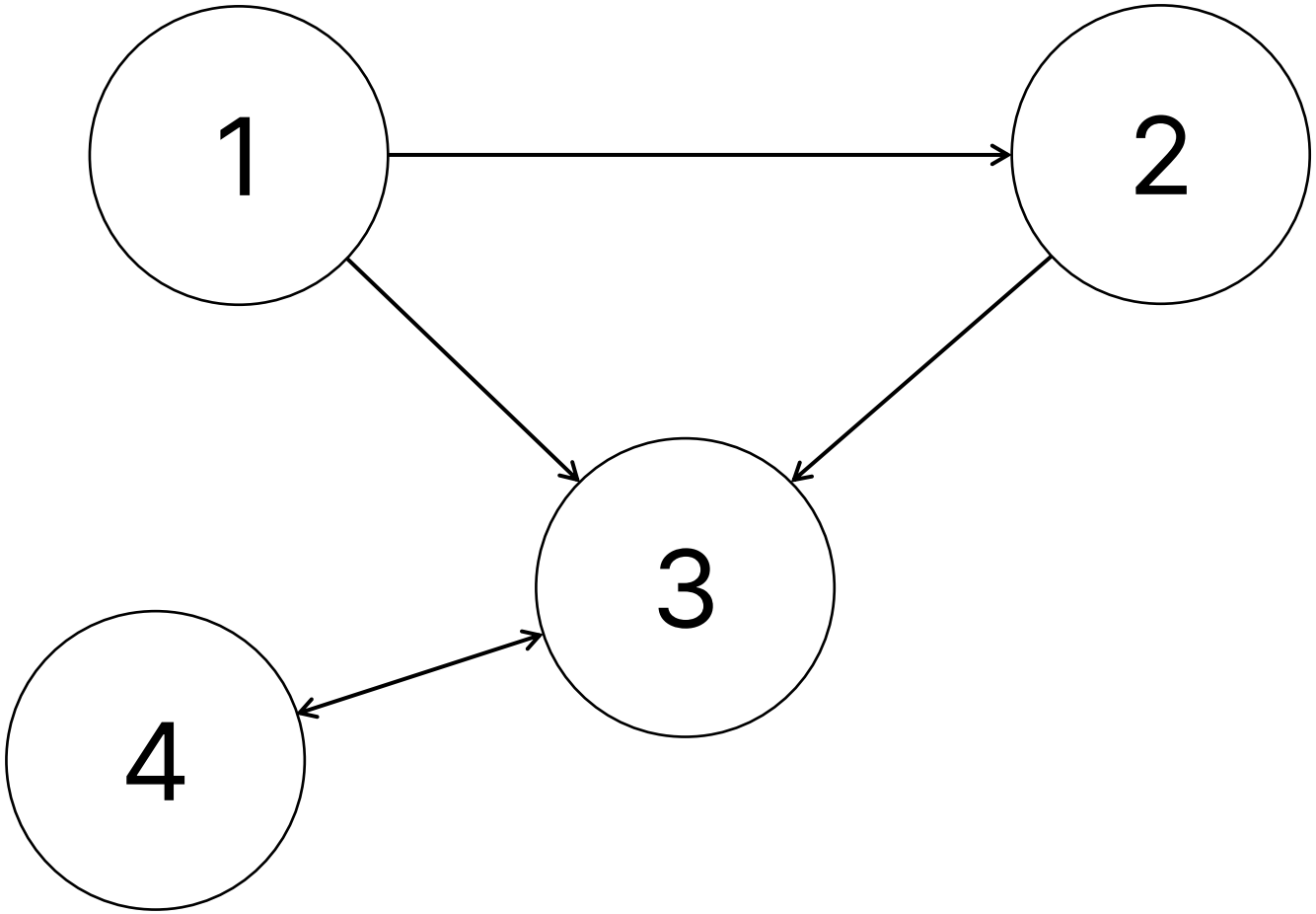
1	2	3	
2	1	3	
3	1	2	4
4	3		

Adjacency List

```cpp
vector<int> edges[MAX_NODE];
int s, e;

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

Adjacency List



Adjacency List

- 간선만을 저장하기 때문에 방향성이 존재하는 경우 도착점들을 저장한다
- 1번 노드와 인접한 노드: 2, 3
- 2번 노드와 인접한 노드: ~~1~~, 3
- 3번 노드와 인접한 노드: ~~1~~, ~~2~~, 4
- 4번 노드와 인접한 노드: 3

Adjacency List

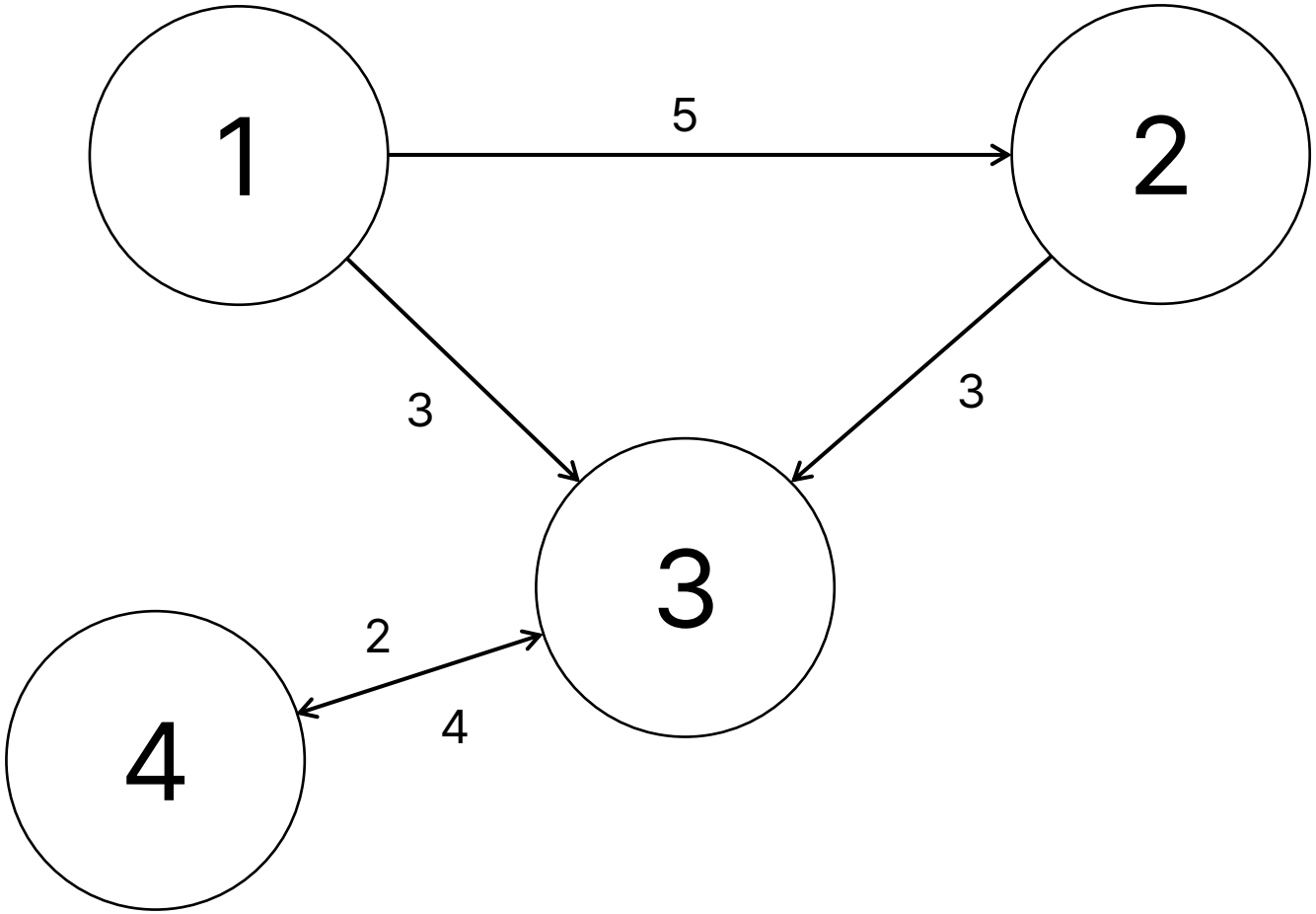
1	2	3
2	3	
3	4	
4	3	

Adjacency List

```cpp
vector<int> edges[MAX_NODE];
int s, e;

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

Adjacency List



Adjacency List

- 가중치가 존재하는 경우 가중치를 같이 저장할 수 있다
- pair<int, int>등을 사용해 {도착점, 가중치} 쌍으로 저장한다
- 구조체를 이용해 도착지와 가중치를 포함하도록 저장한다

Adjacency List

1	{2, 5}	{3, 3}
2	{3, 3}	
3	{4, 2}	
4	{3, 4}	

Adjacency List

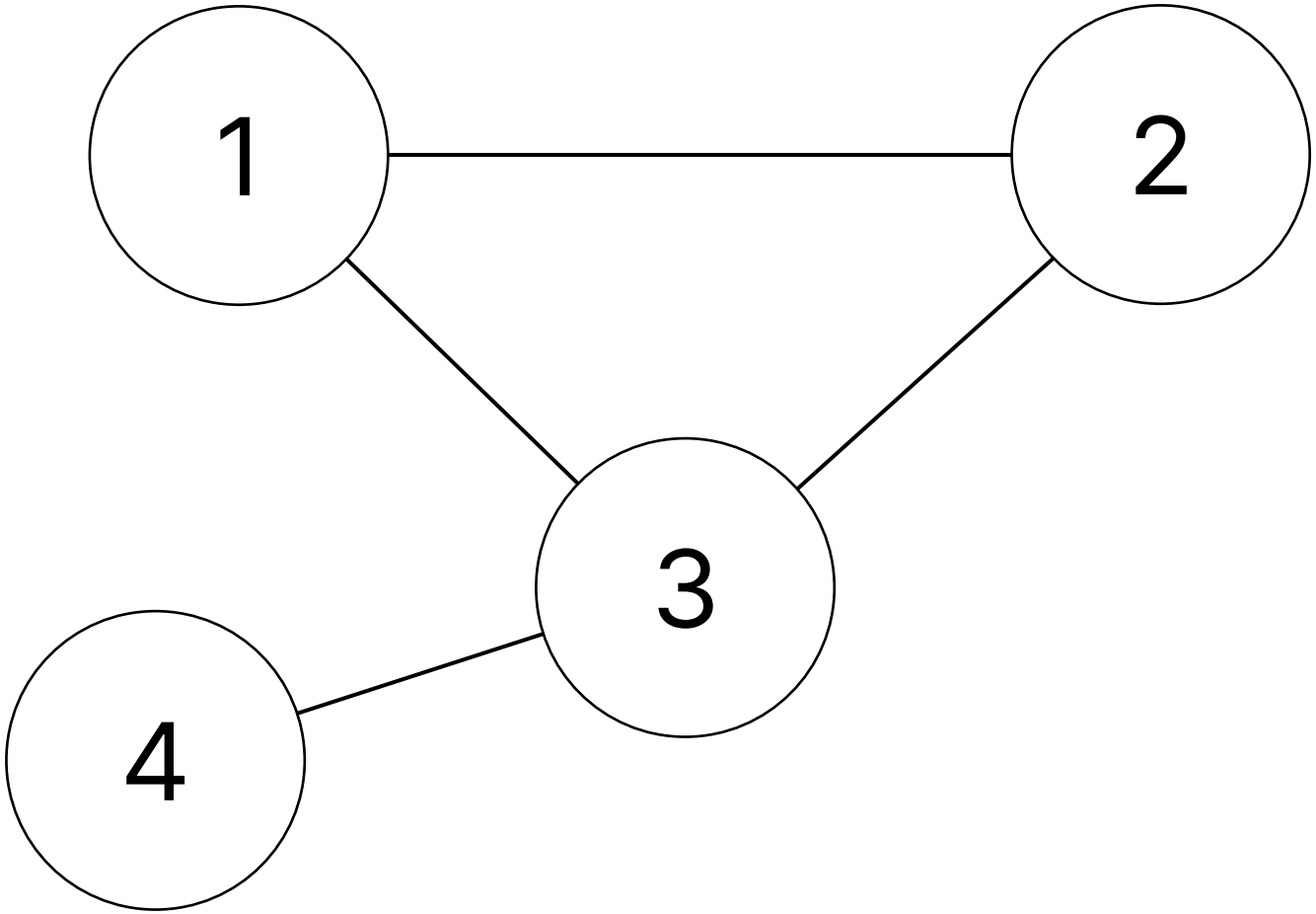
```cpp
vector<pair<int, int>> edges[MAX_NODE];
int s, e, w;

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

Adjacency Matrix

- 행렬로 노드의 연결 유무를 저장한다
- arr[i][j]는 i번 노드에서 출발해 j번 노드로 가는 간선의 유무 또는 가중치를 저장한다
- 양방향 그래프는 대칭을 보임

Adjacency Matrix



Adjacency Matrix

- 1번 노드와 2번 노드를 연결하는 노드를 살펴보자
- 1->2번 간선도 존재하며 2->1번으로 가는 간선도 존재한다는 뜻이다
- 즉, 양방향 그래프에서는 [i][j]와 [j][i]가 동일하다

Adjacency Matrix

	1	2	3	4
1	X	O	O	X
2	O	X	O	X
3	O	O	X	O
4	X	X	O	X

Adjacency Matrix

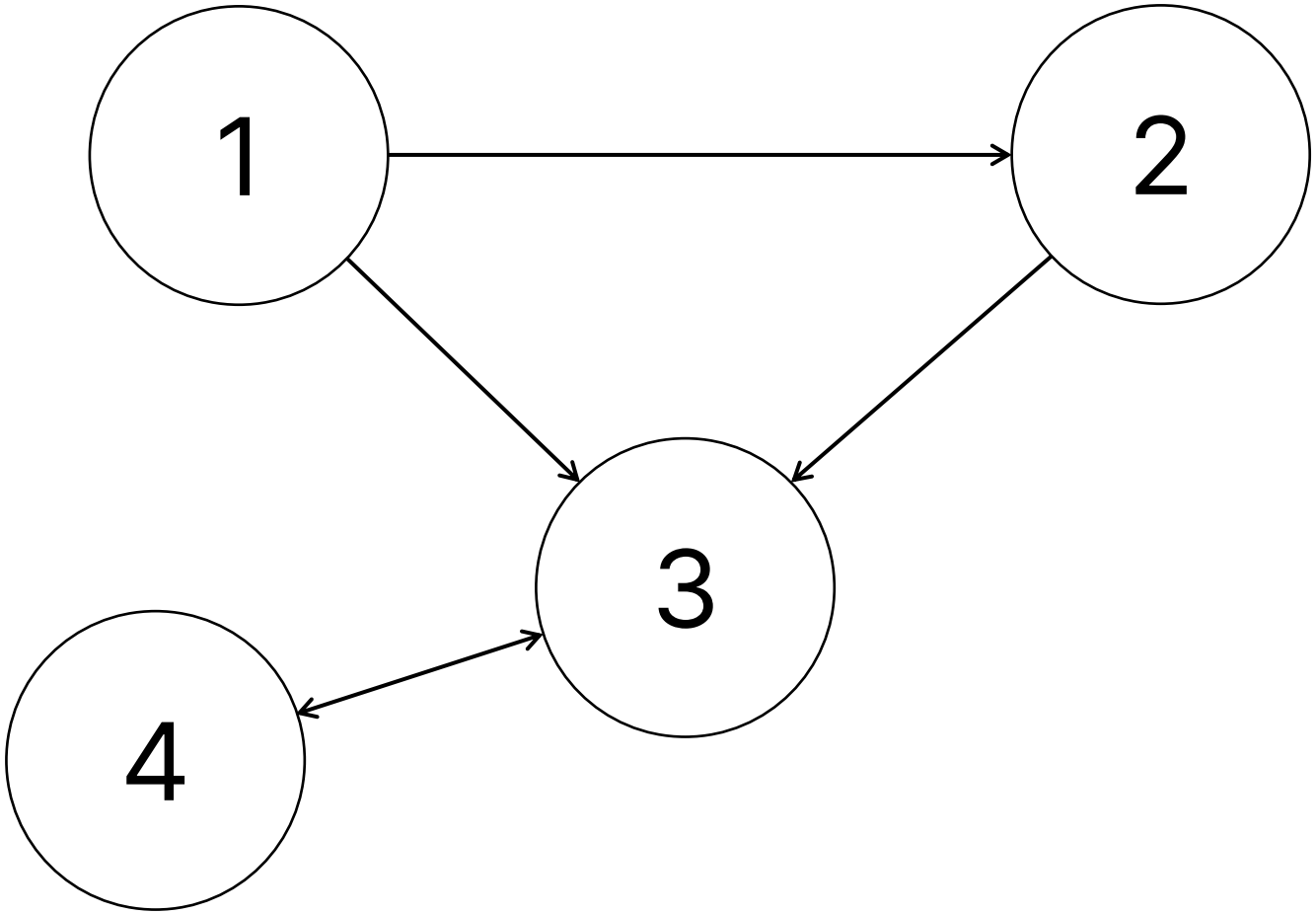
	1	2	3	4
1	F	T	T	F
2	T	F	T	F
3	T	T	F	T
4	F	F	T	F

Adjacency Matrix

```
bool edges[MAX_NODE][MAX_NODE];
int s, e;

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

Adjacency Matrix



Adjacency Matrix

	1	2	3	4
1	X	O	O	X
2	X	X	O	X
3	X	X	X	O
4	X	X	O	X

Adjacency Matrix

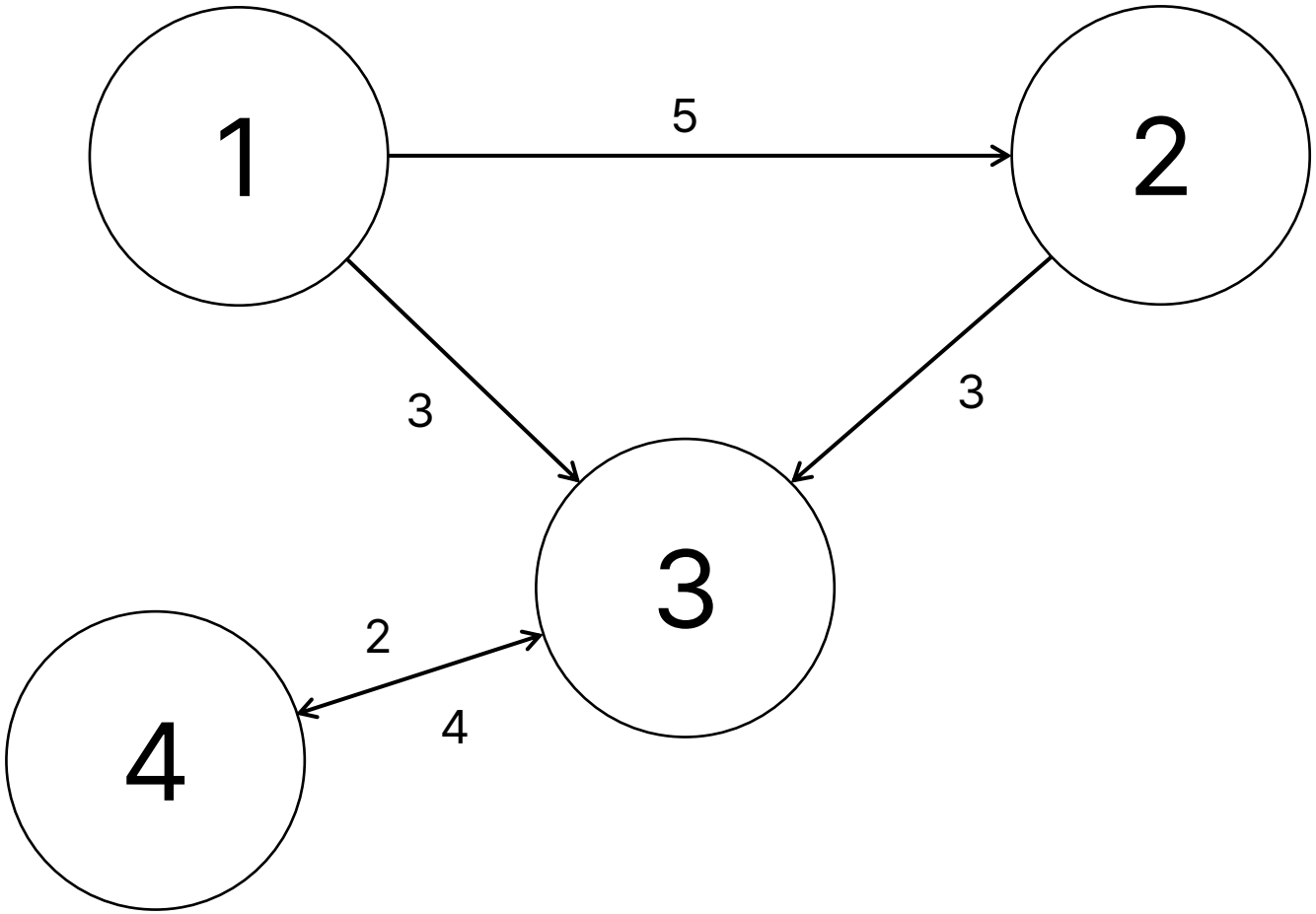
	1	2	3	4
1	F	T	T	F
2	F	F	T	F
3	F	F	F	T
4	F	F	T	F

Adjacency Matrix

```
bool edges[MAX_NODE][MAX_NODE];
int s, e;

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

Adjacency Matrix



Adjacency Matrix

	1	2	3	4
1	0	5	3	0
2	0	0	3	0
3	0	0	0	2
4	0	0	4	0

Adjacency Matrix

```
int edges[MAX_NODE][MAX_NODE];
int s, e, w;

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

Traversal

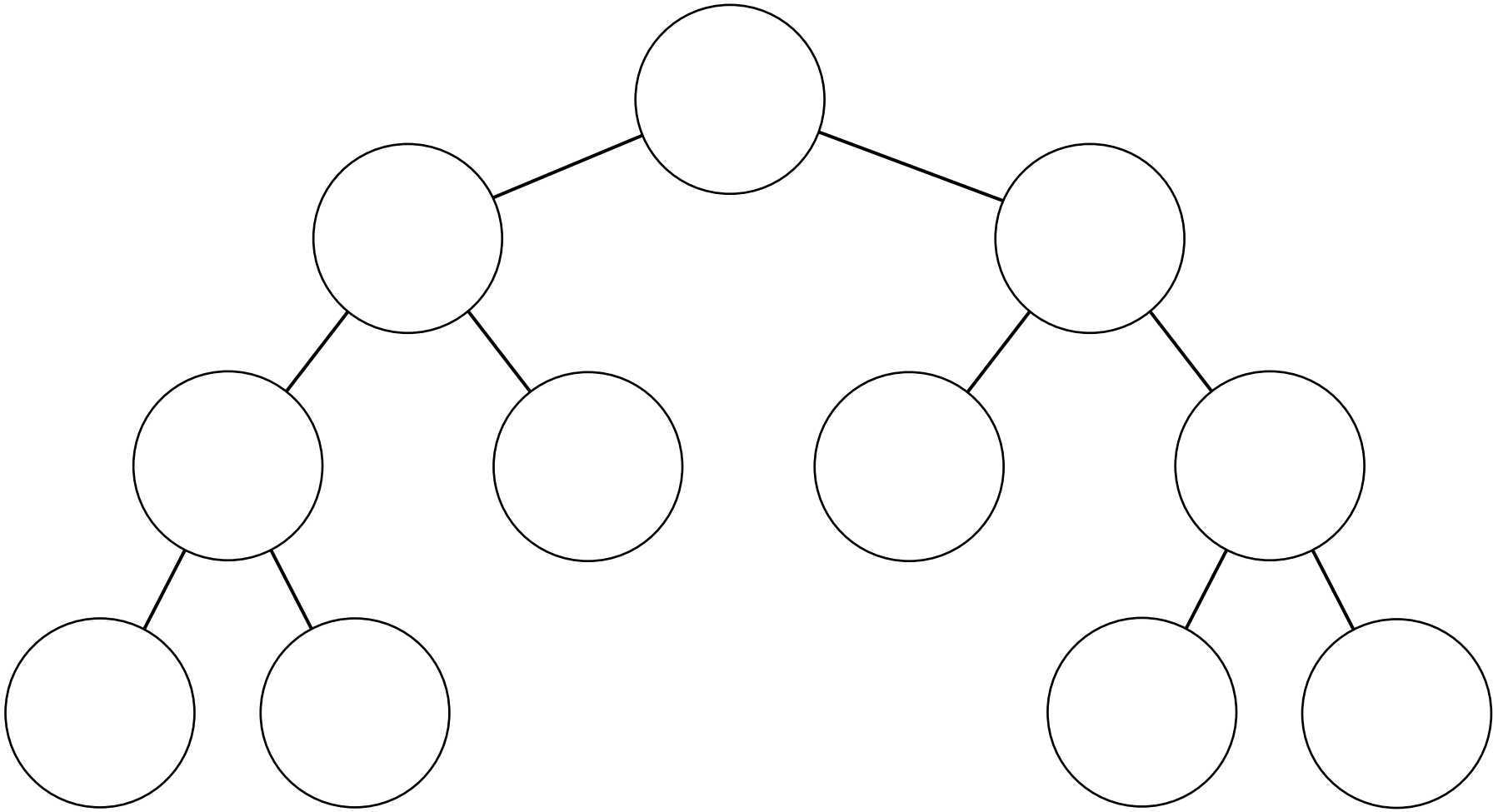
- 연결된 모든 정점을 탐색하는 것
- 연결된 노드들 중 어떤 것을 먼저 방문할지에 따라 방법이 나뉜다

- 깊이 우선 탐색, Depth First Search, DFS
- 너비 우선 탐색, Breath First Search , BFS

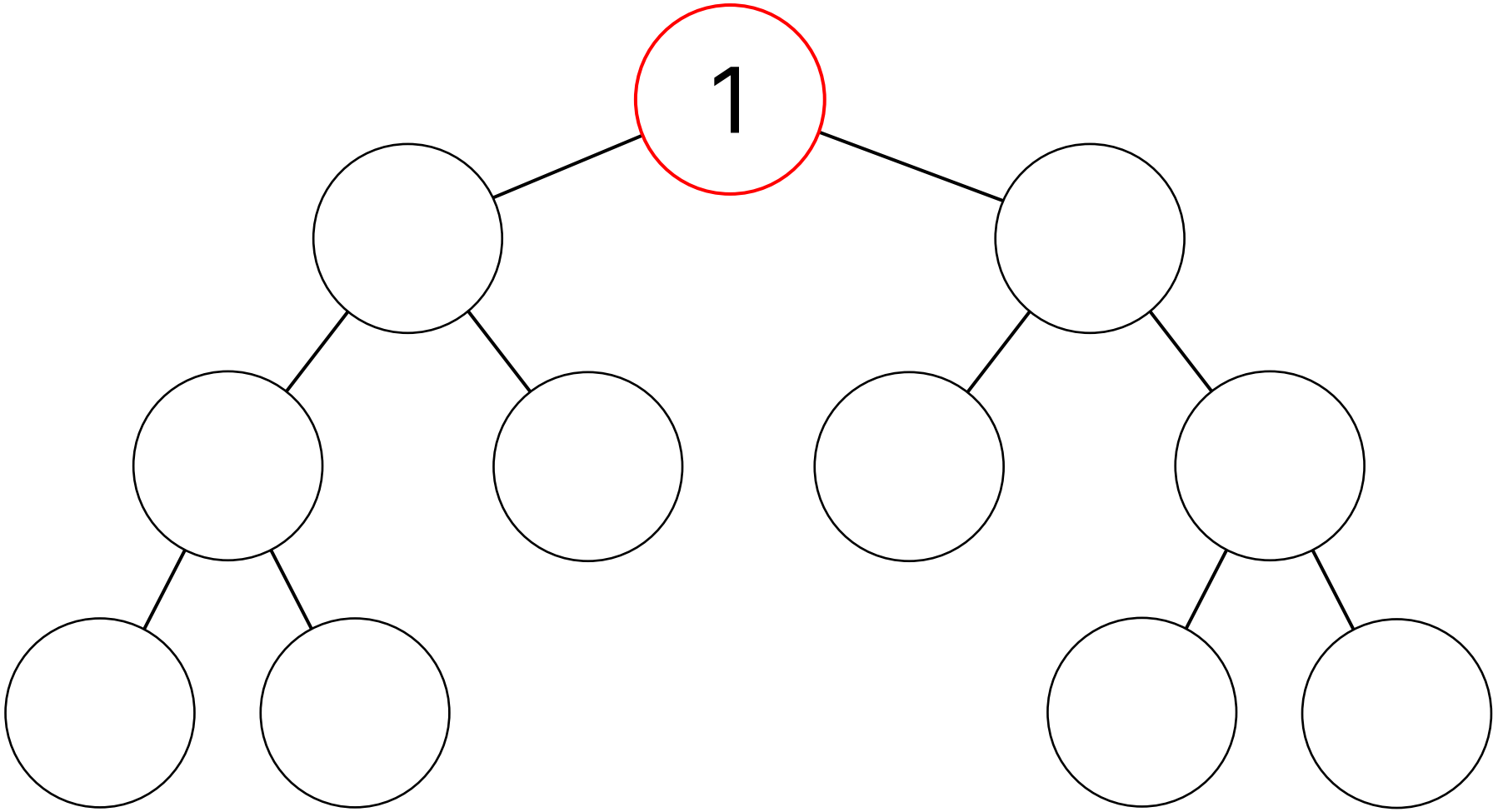
DFS

- 깊이 우선 탐색
- 현재 노드와 연결된 노드 중 이동할 수 있는 노드(방문하지 않은 노드)가 존재한다면 해당 노드로 이동한다
- 더 이상 이동할 수 있는 노드가 없다면 이전 노드로 돌아간다
- 이 두 과정을 반복한다

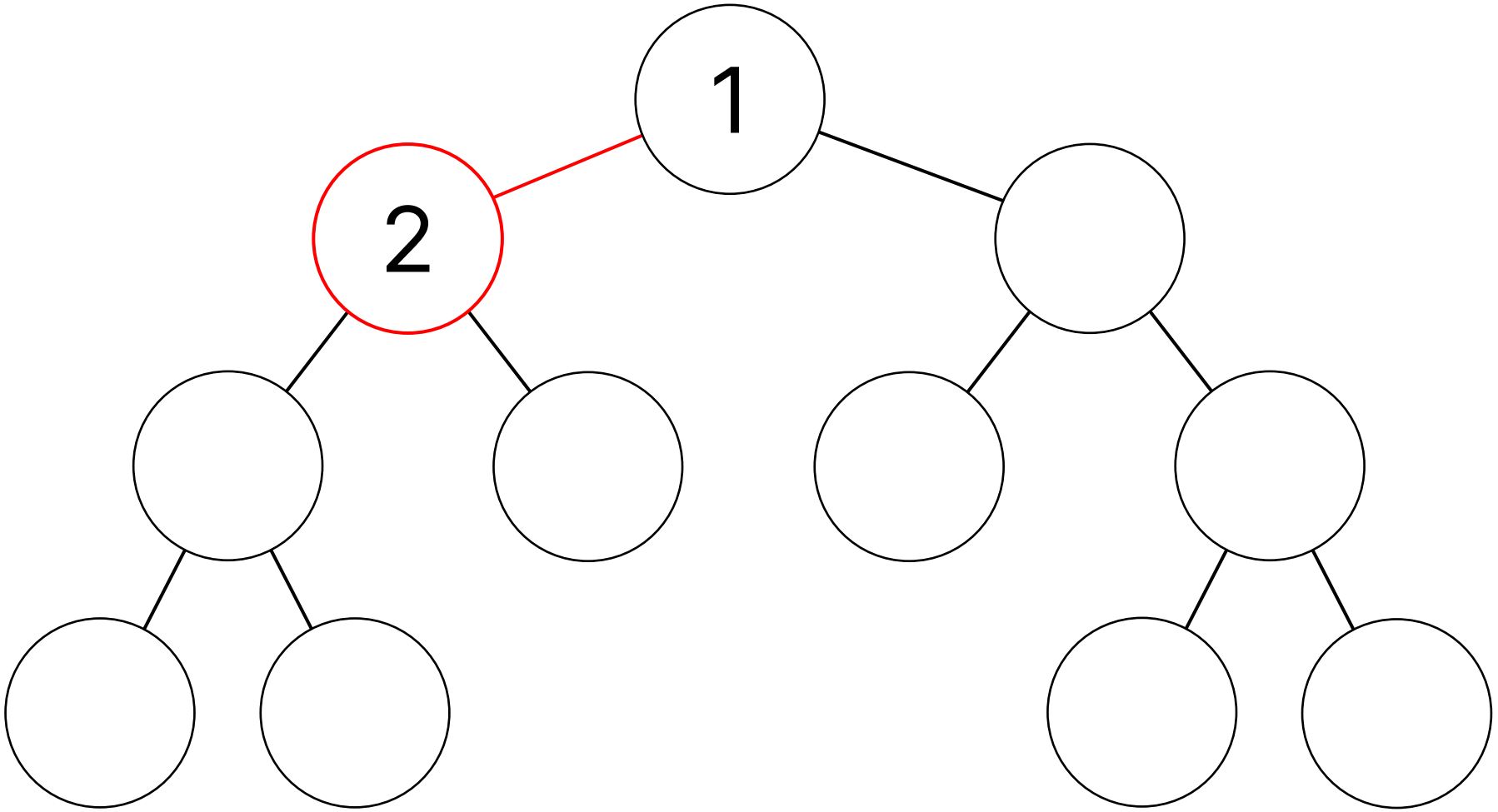
DFS



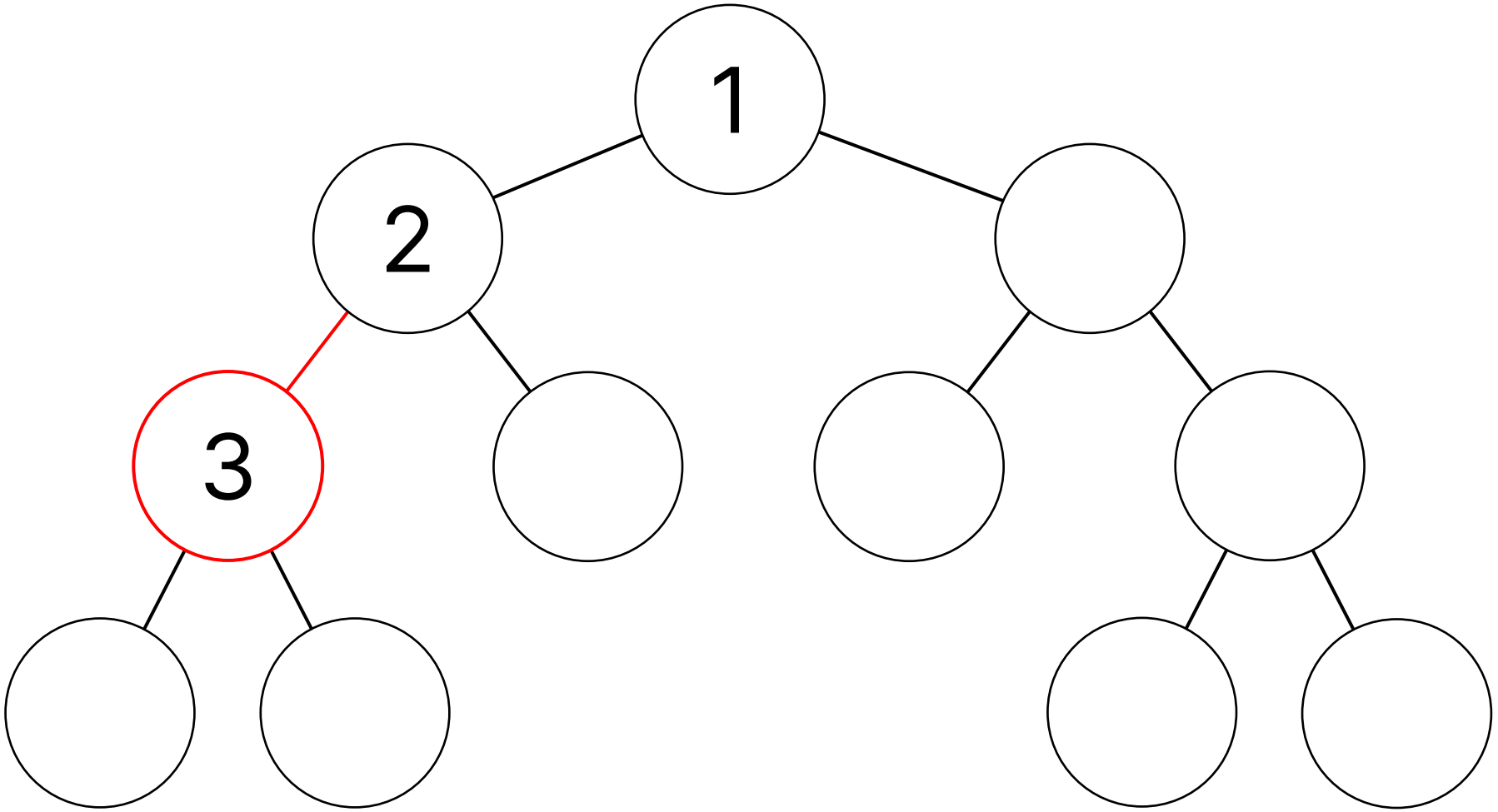
DFS



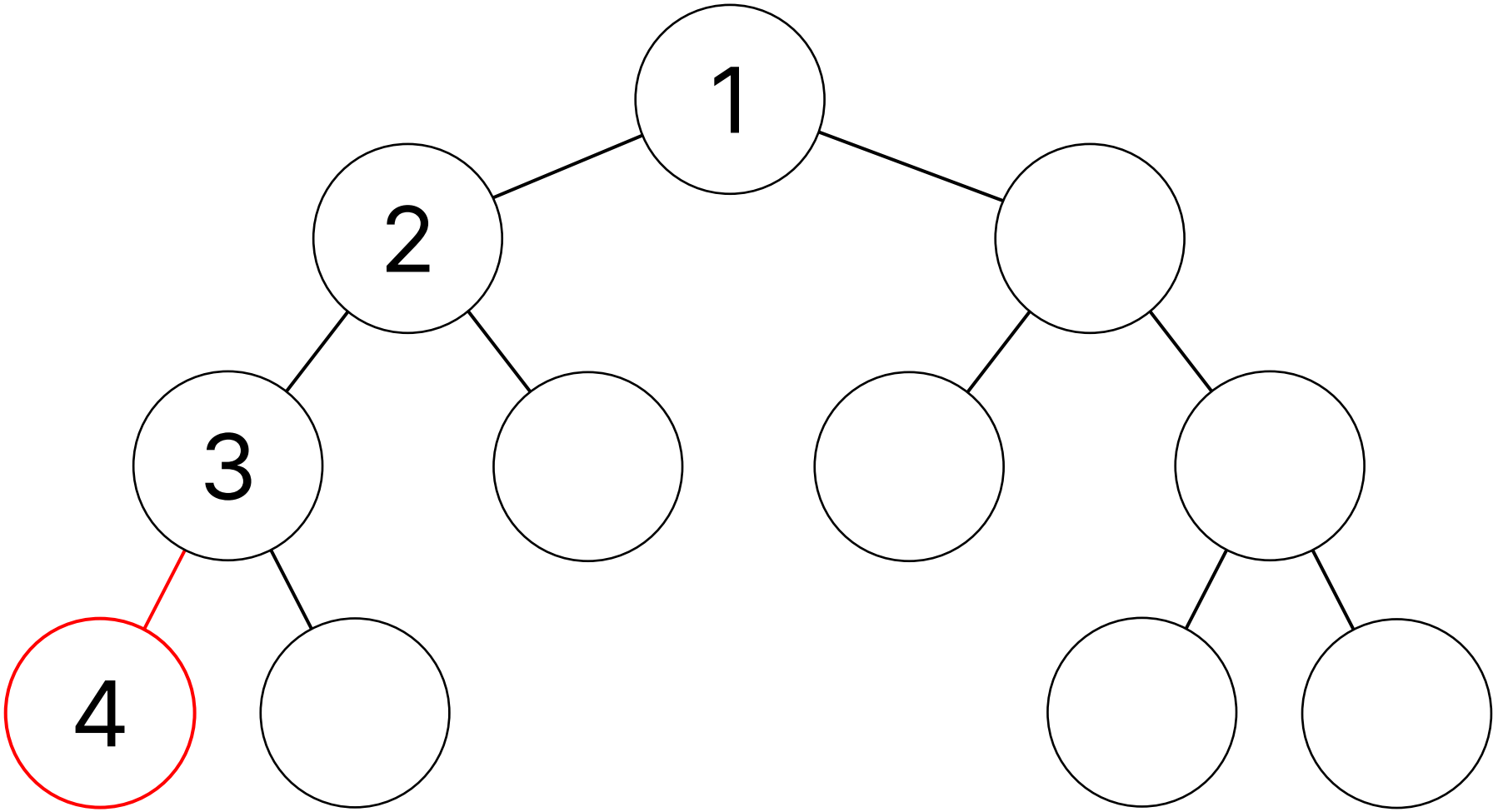
DFS



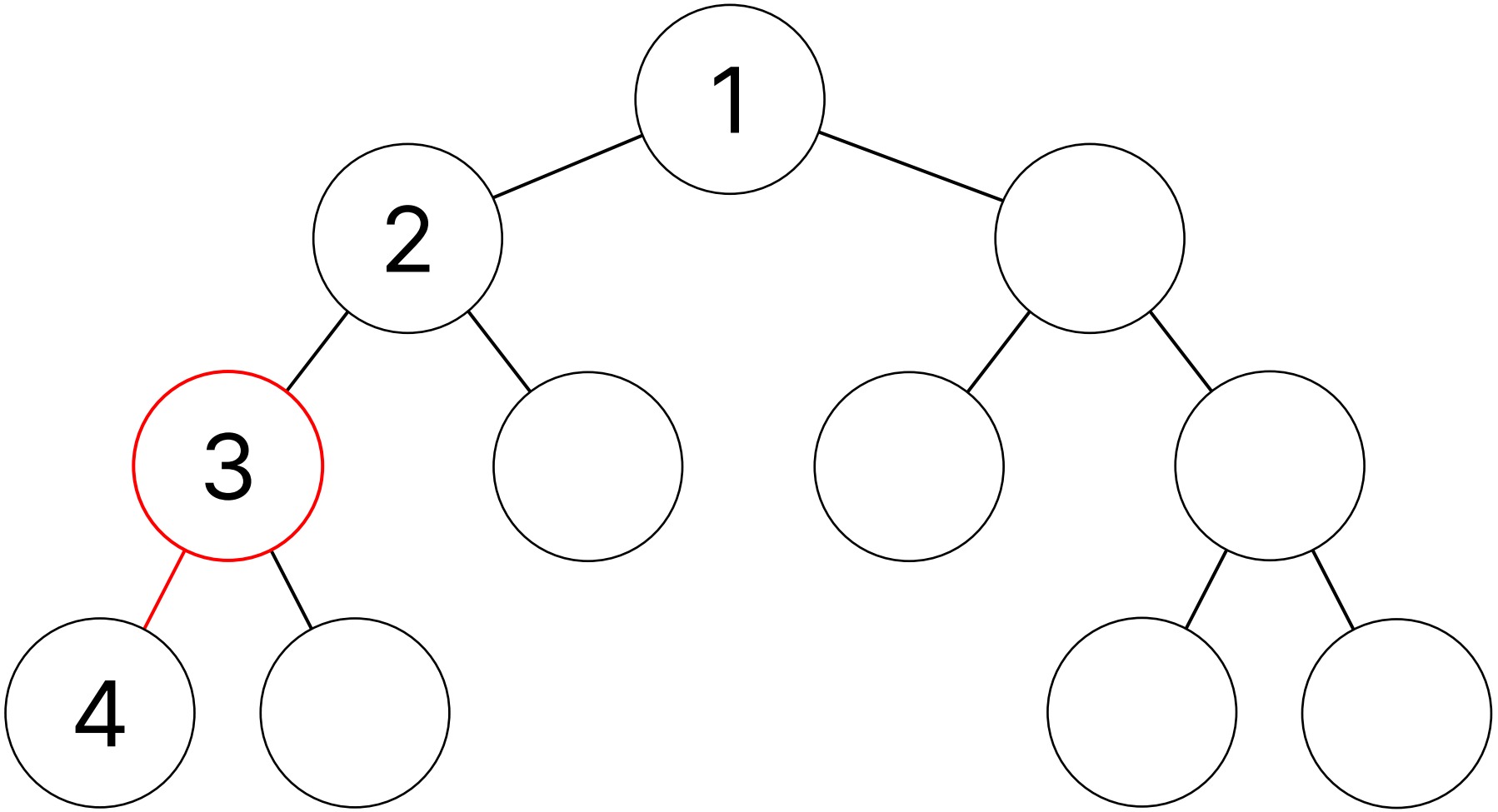
DFS



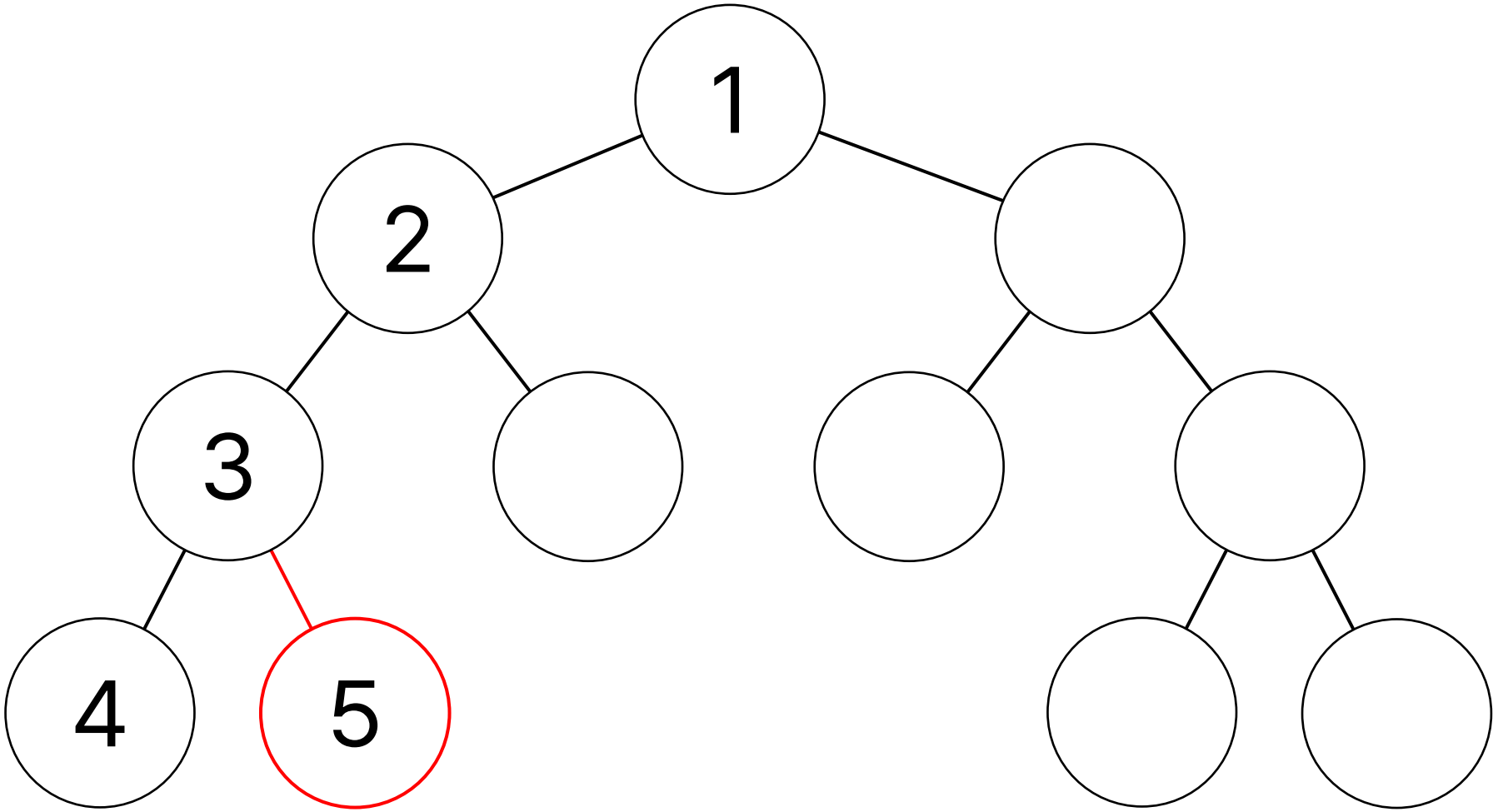
DFS



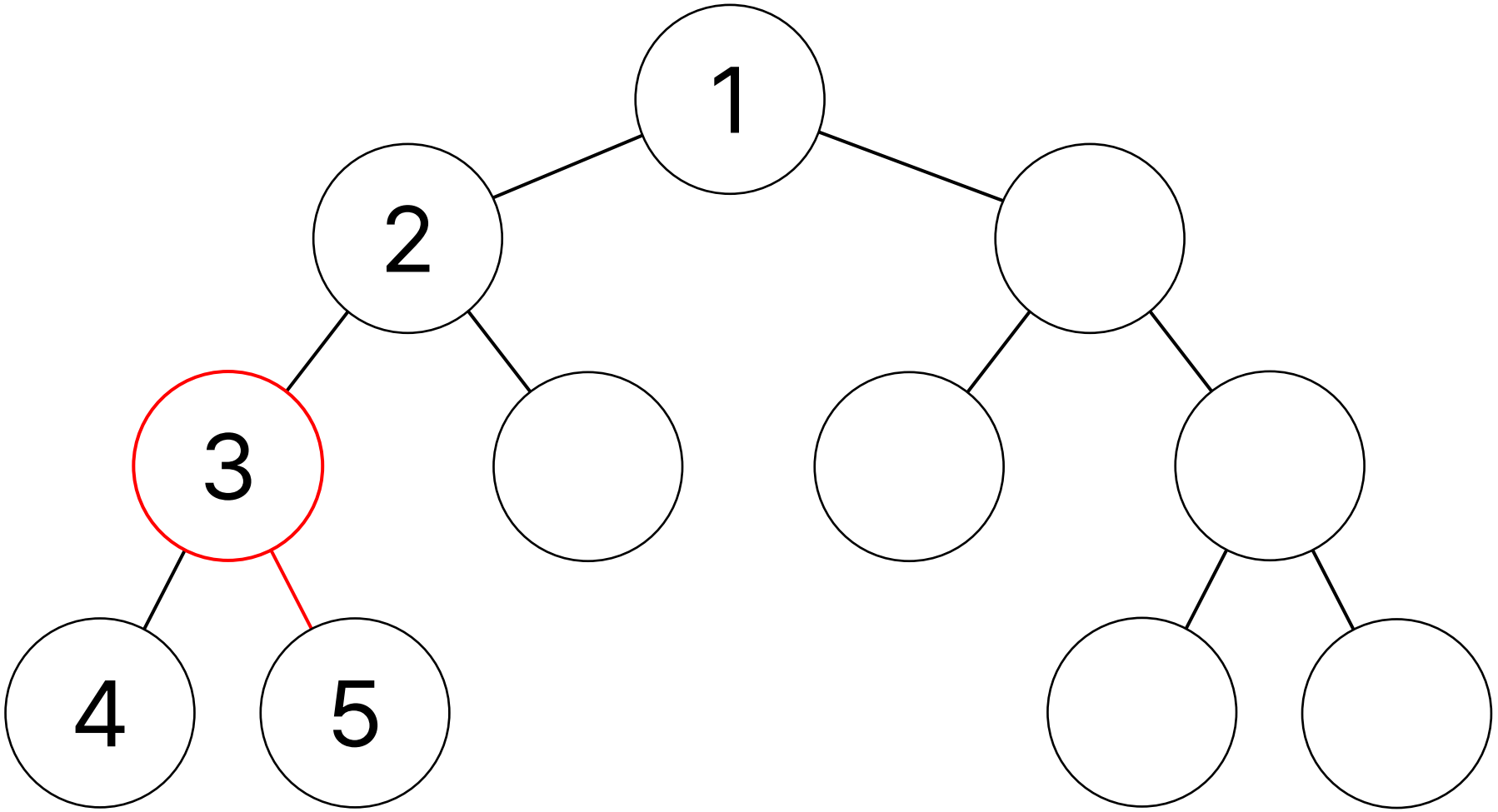
DFS



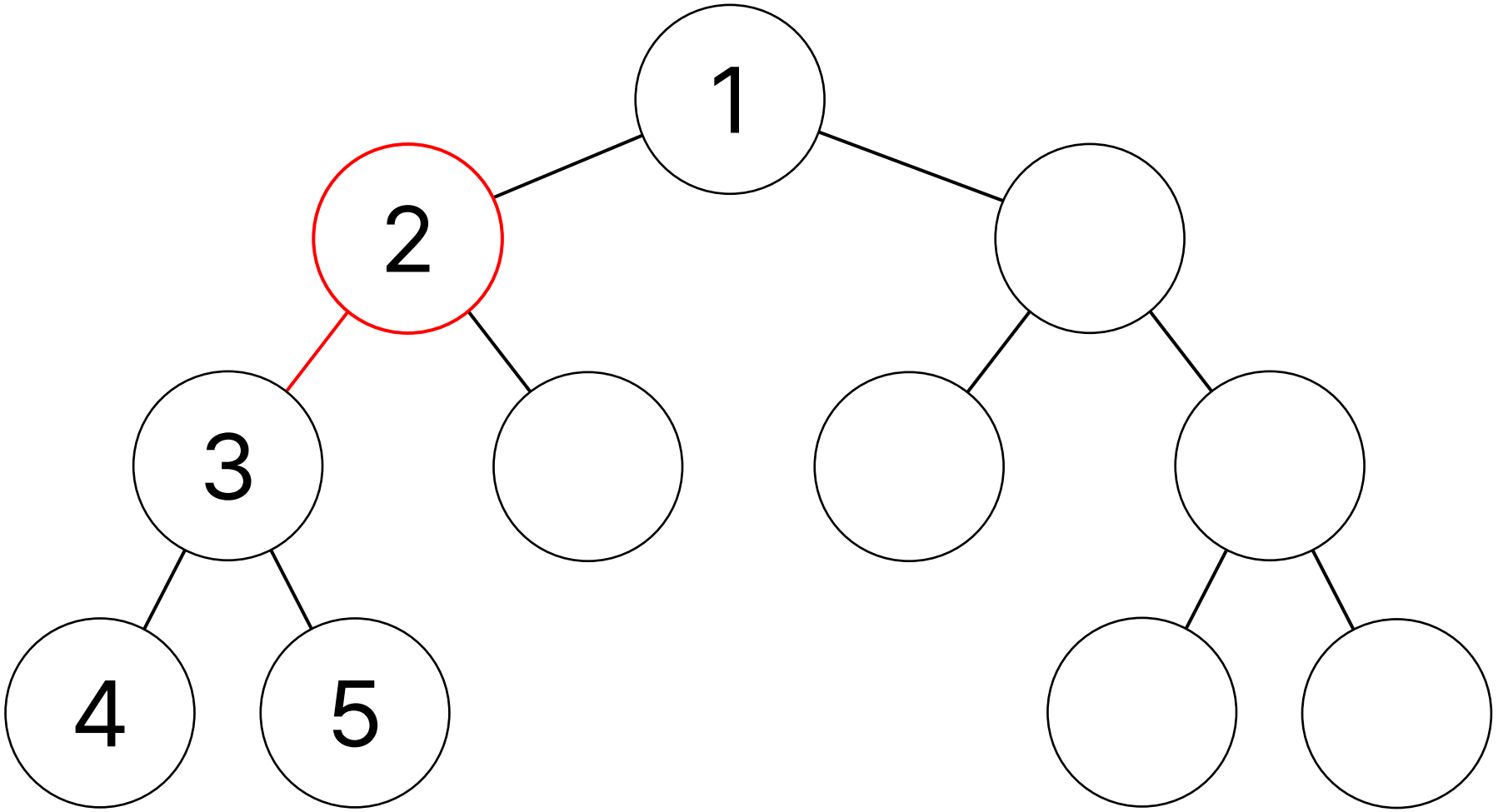
DFS



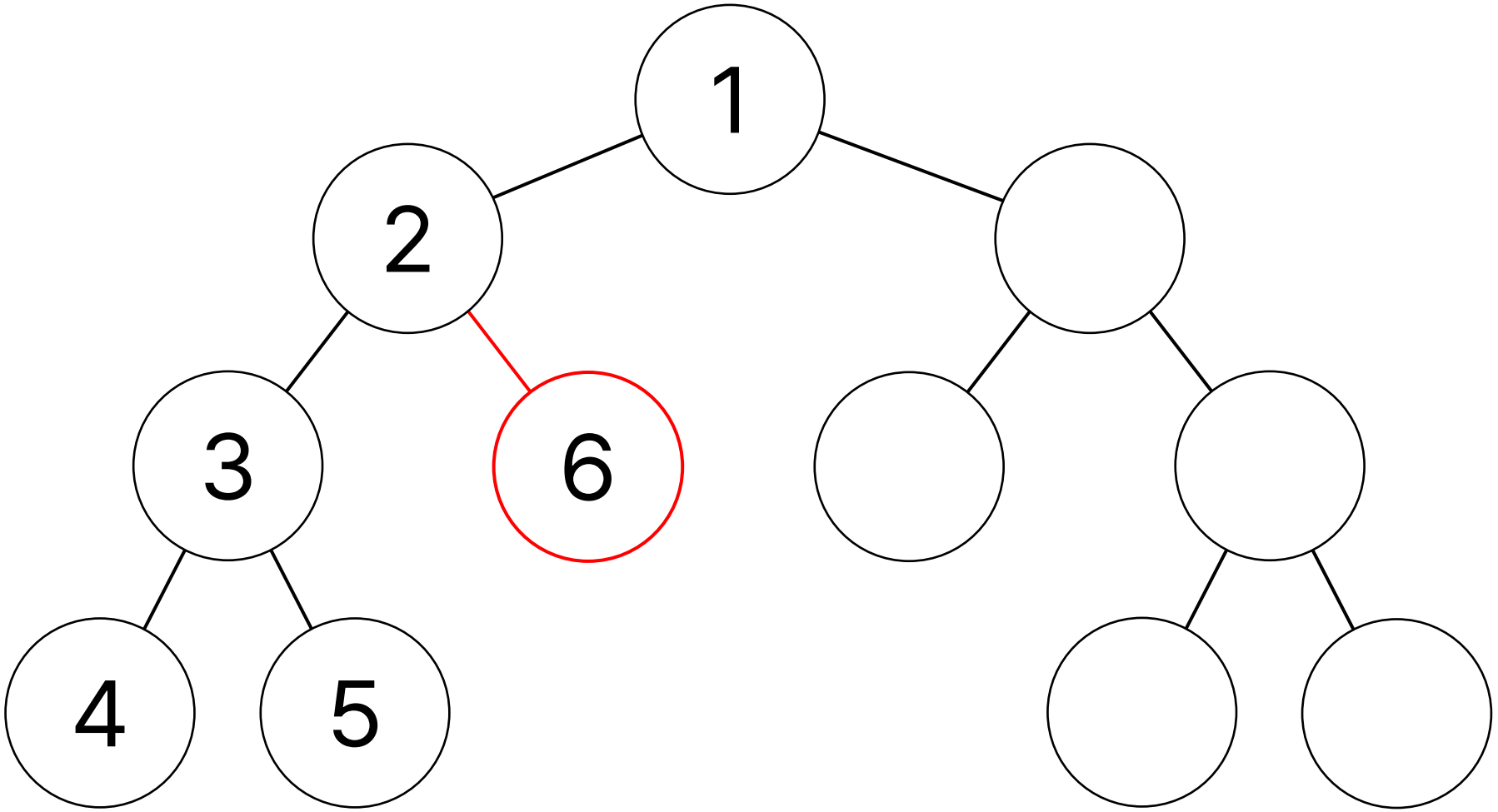
DFS



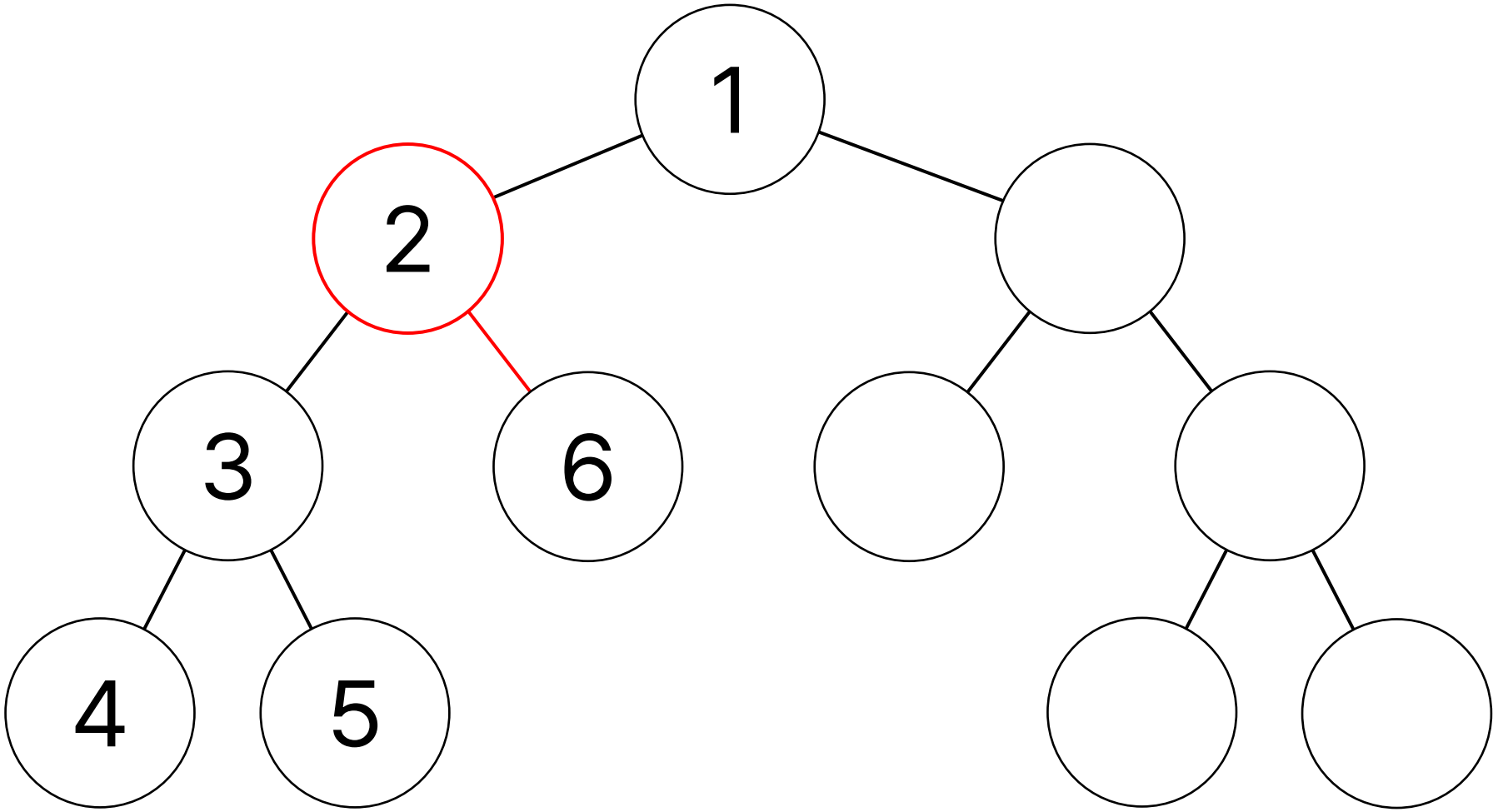
DFS



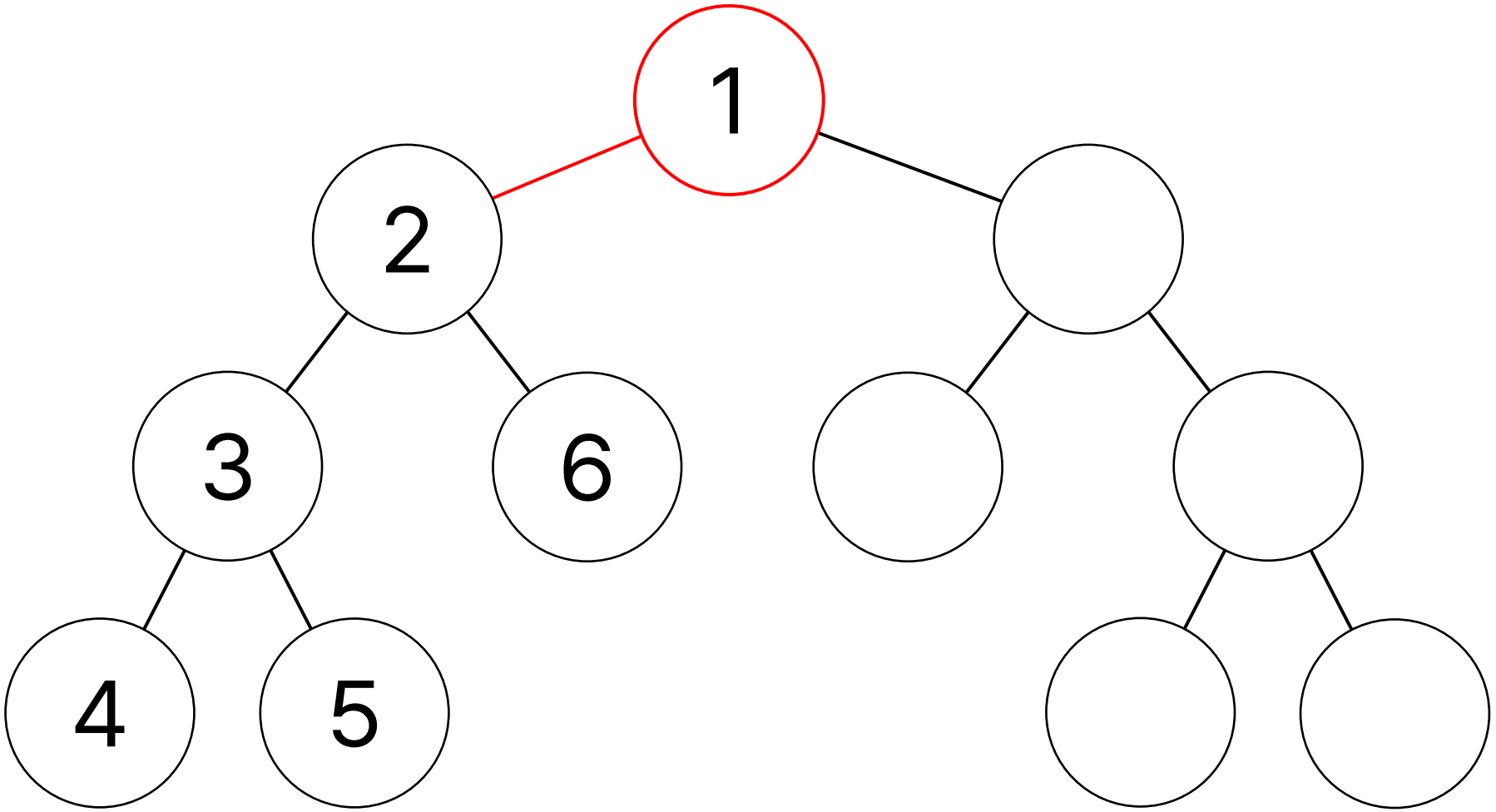
DFS



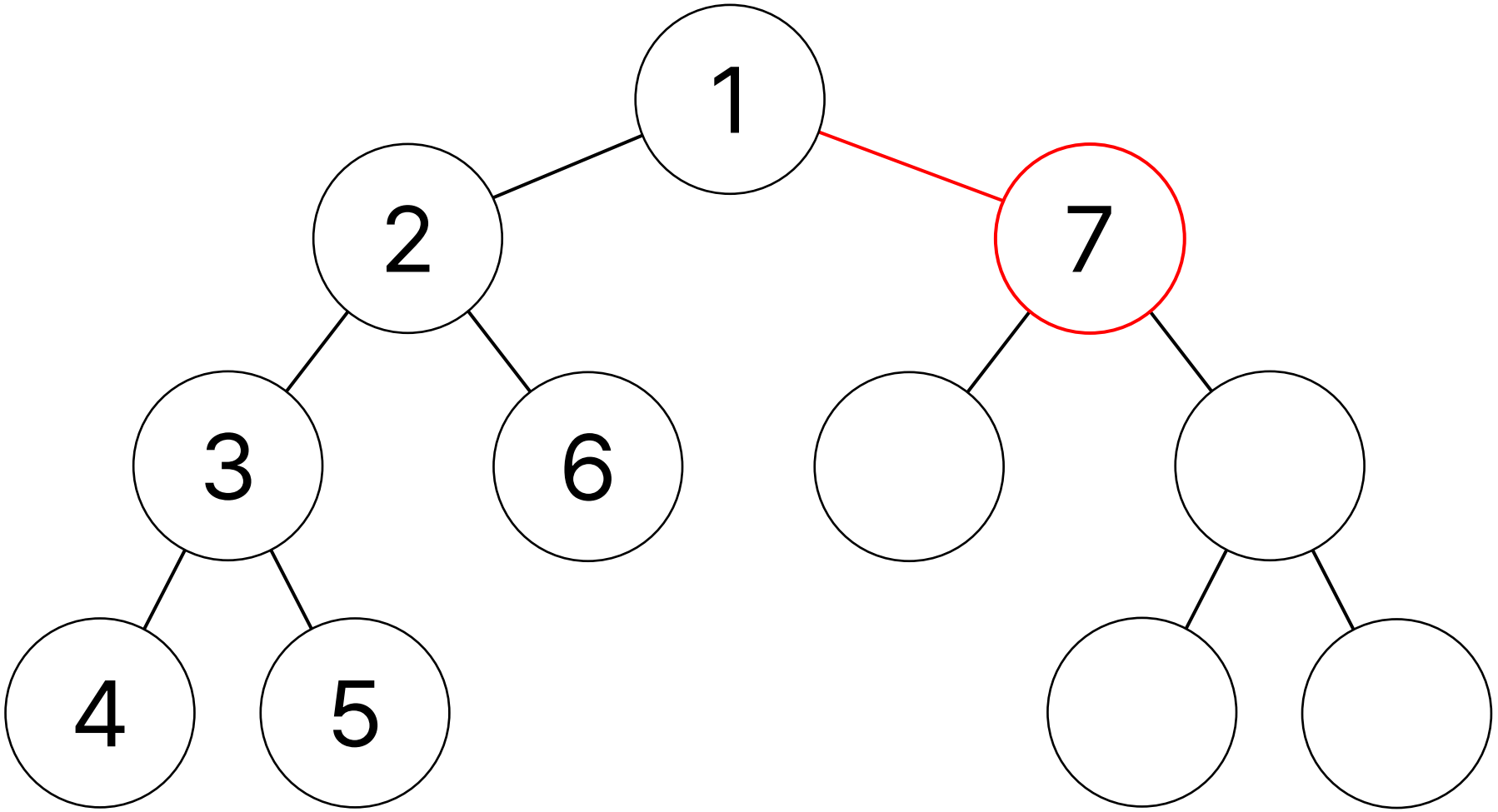
DFS



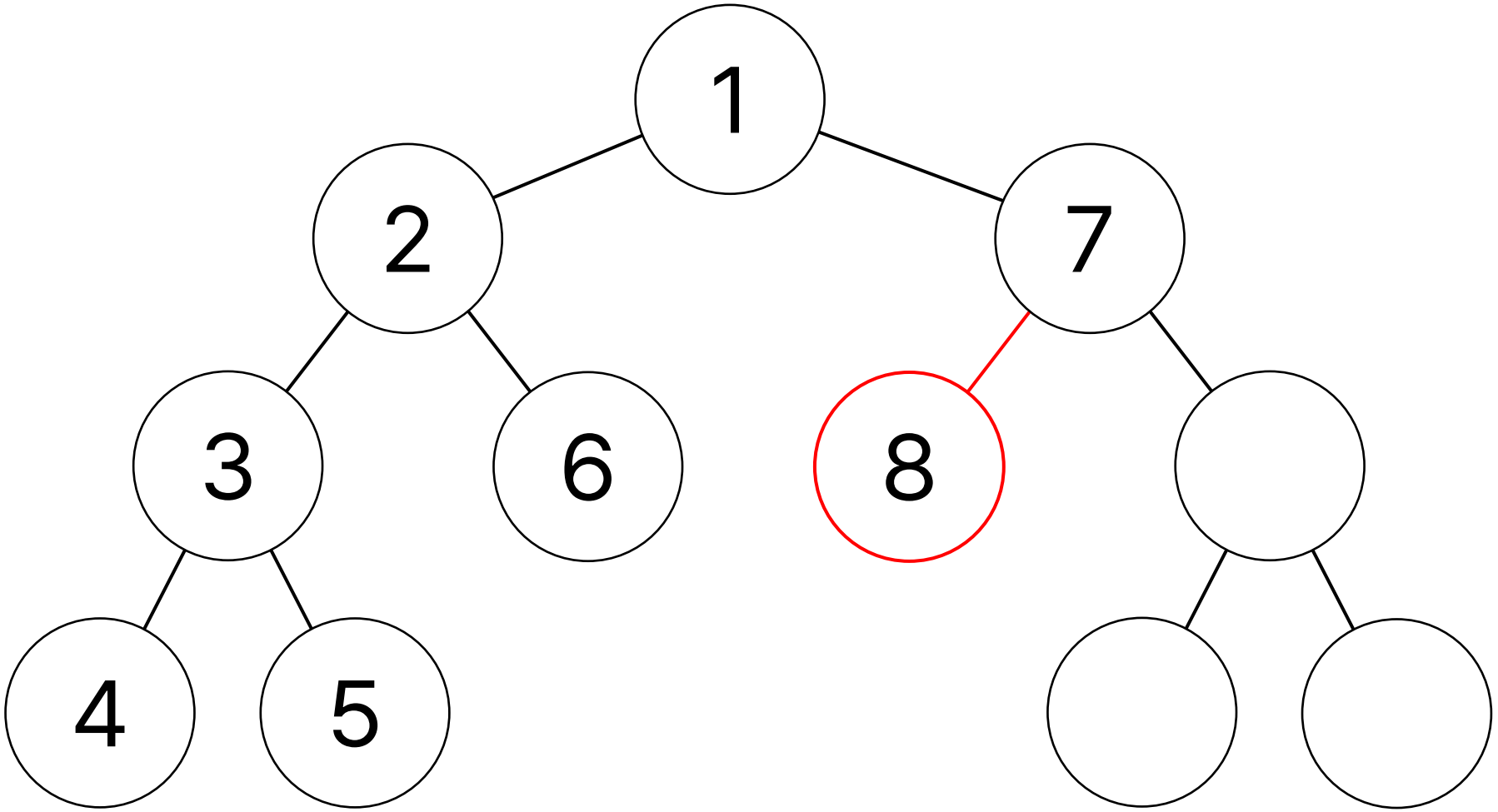
DFS



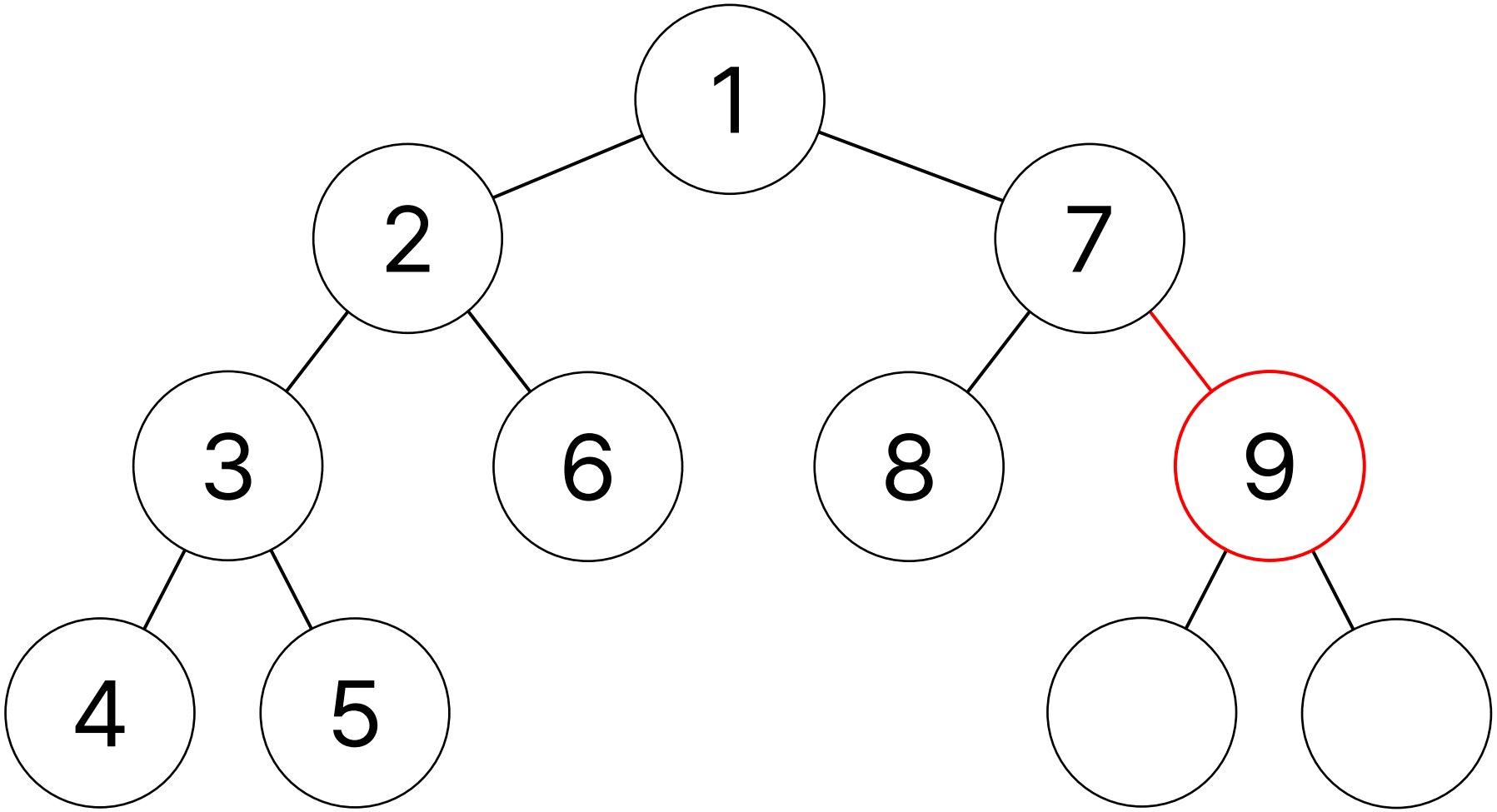
DFS



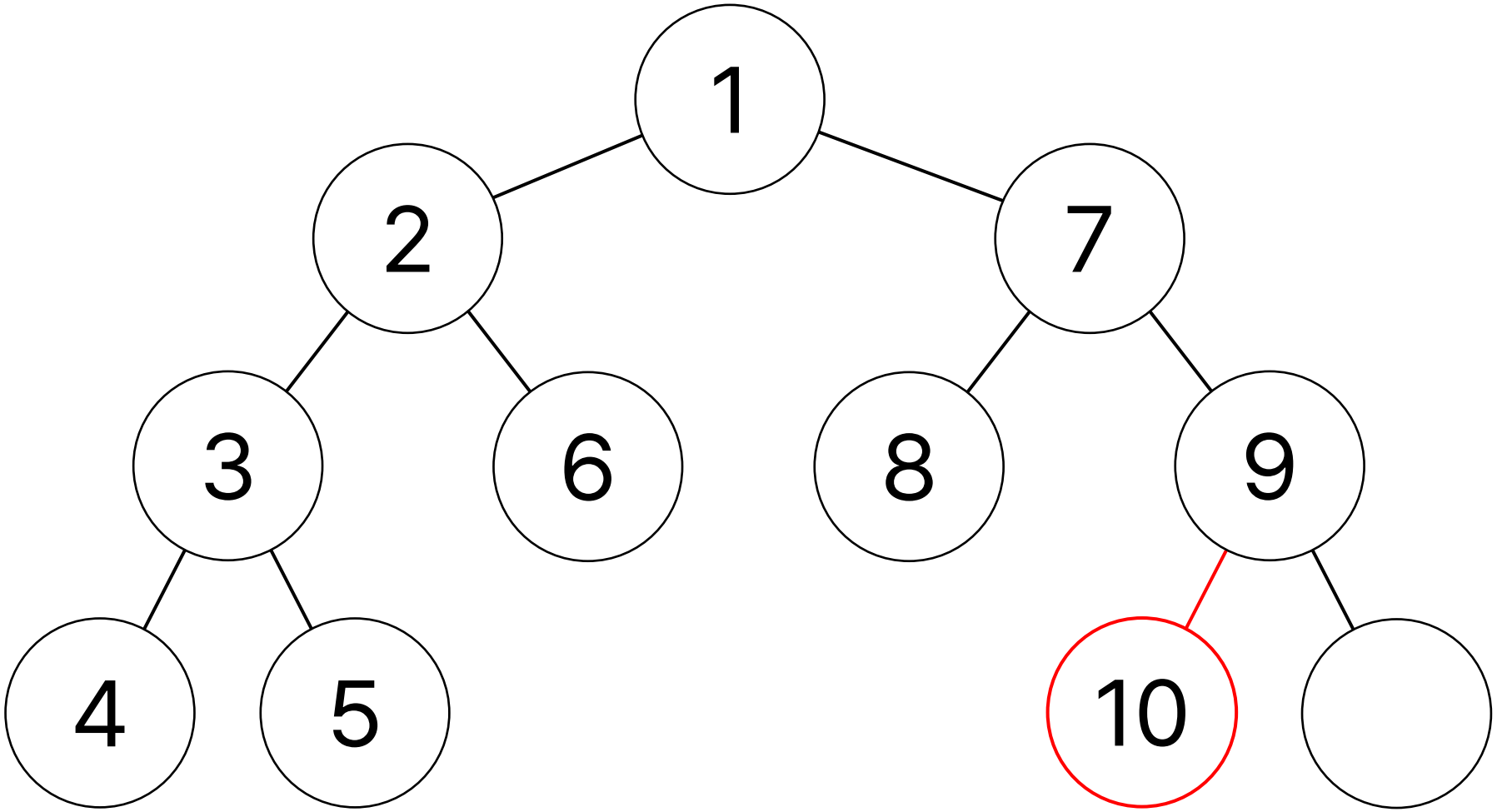
DFS



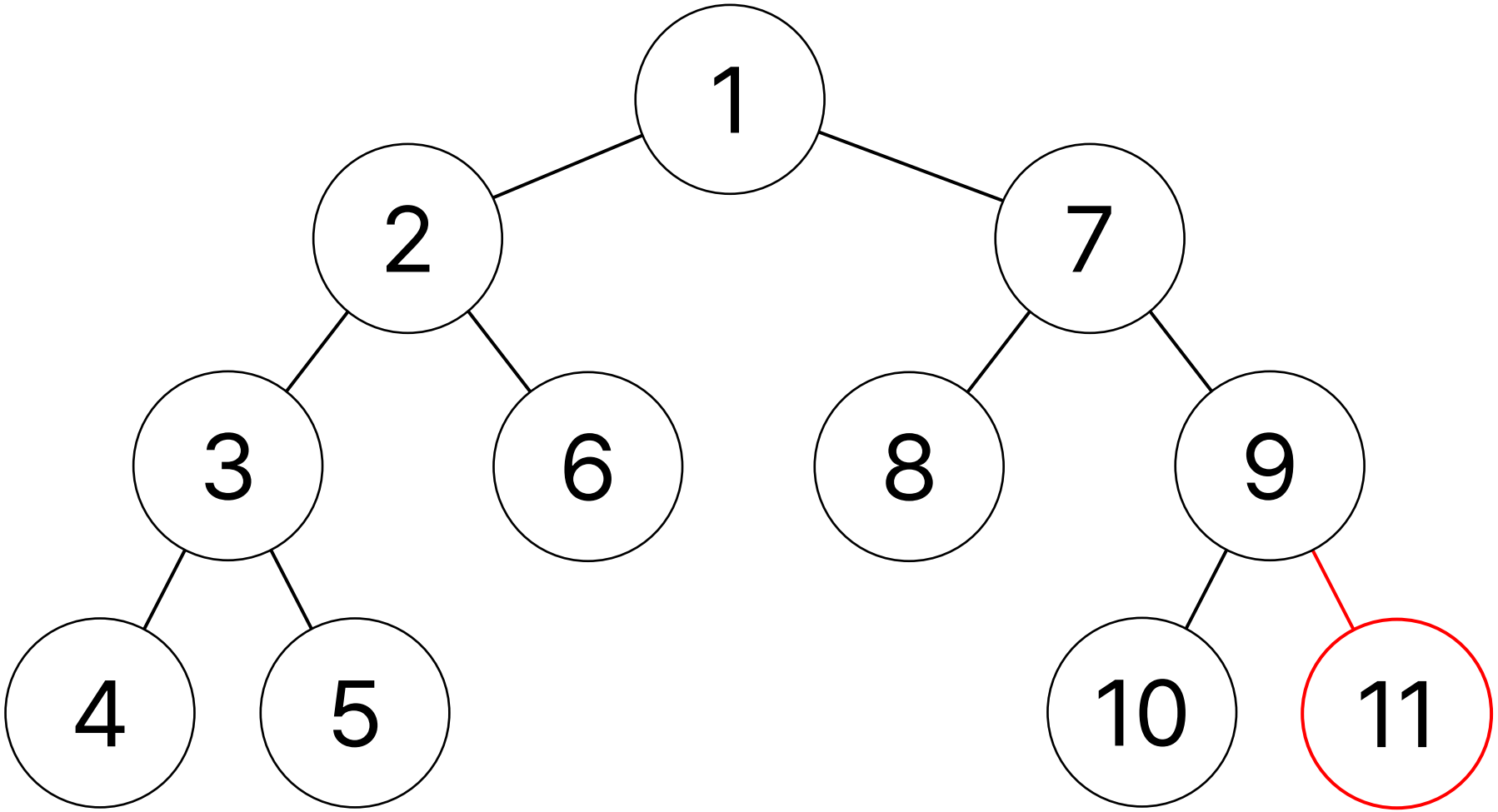
DFS



DFS



DFS



DFS

- 인접 리스트를 이용한다
- 노드를 방문했는지 기억할 배열이 필요하다

  bool 또는 int 배열을 사용한다
- 반복문으로 구현이 가능하지만 재귀적으로 수행되므로 함수로 구현이 가능하다
- 함수로 구현하는 것이 간단해 주로 함수로 구현한다

DFS

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

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

DFS

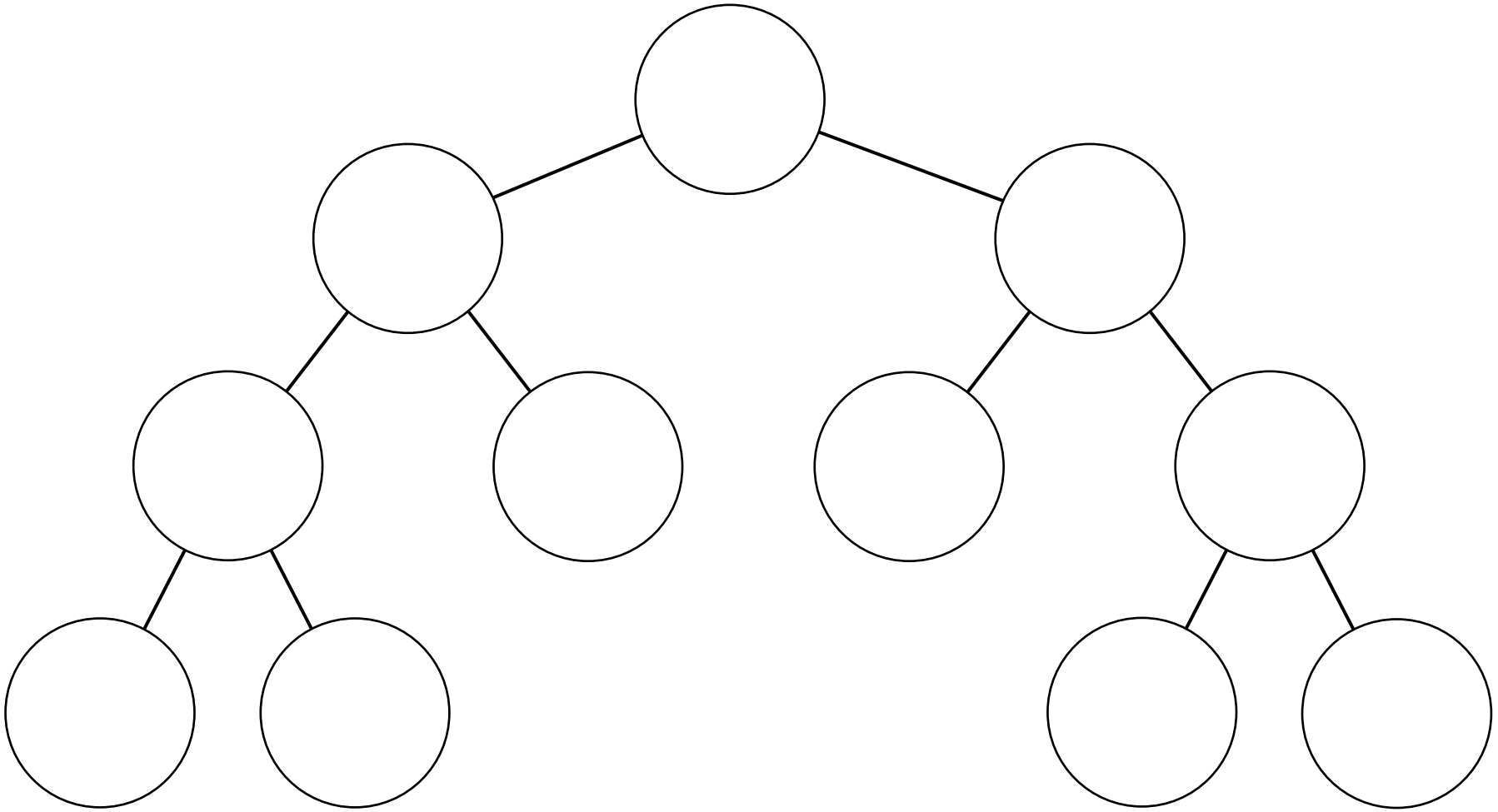
```cpp
vector<int> edges[MAX_NODE];
bool visited[MAX_NODE];

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

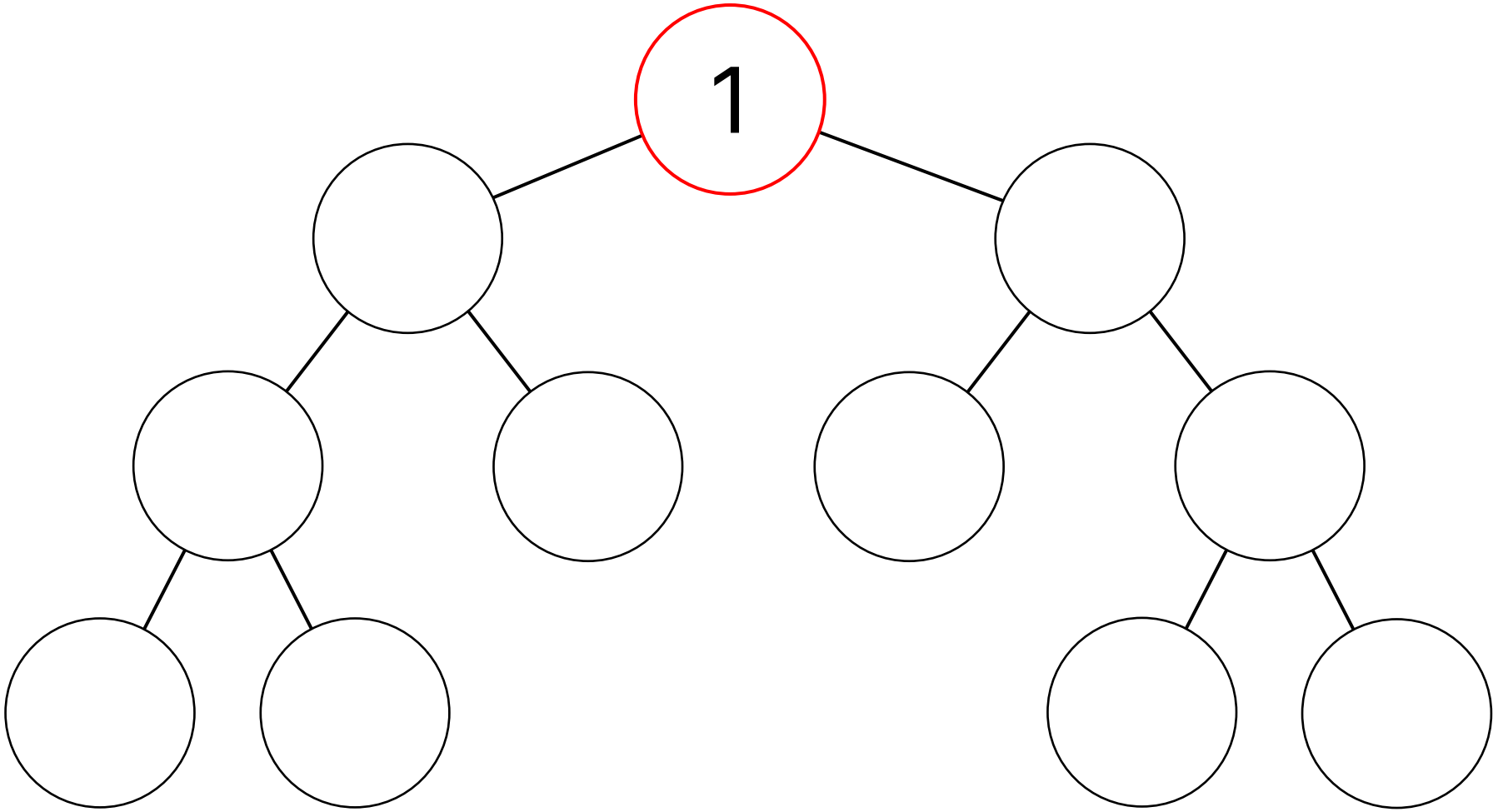
BFS

- 너비 우선 탐색
- 시작 노드로부터 가까운 노드들부터 방문하는 방식
- 가까운 노드부터 탐색하기 때문에 최단거리를 탐색할 때 사용한다

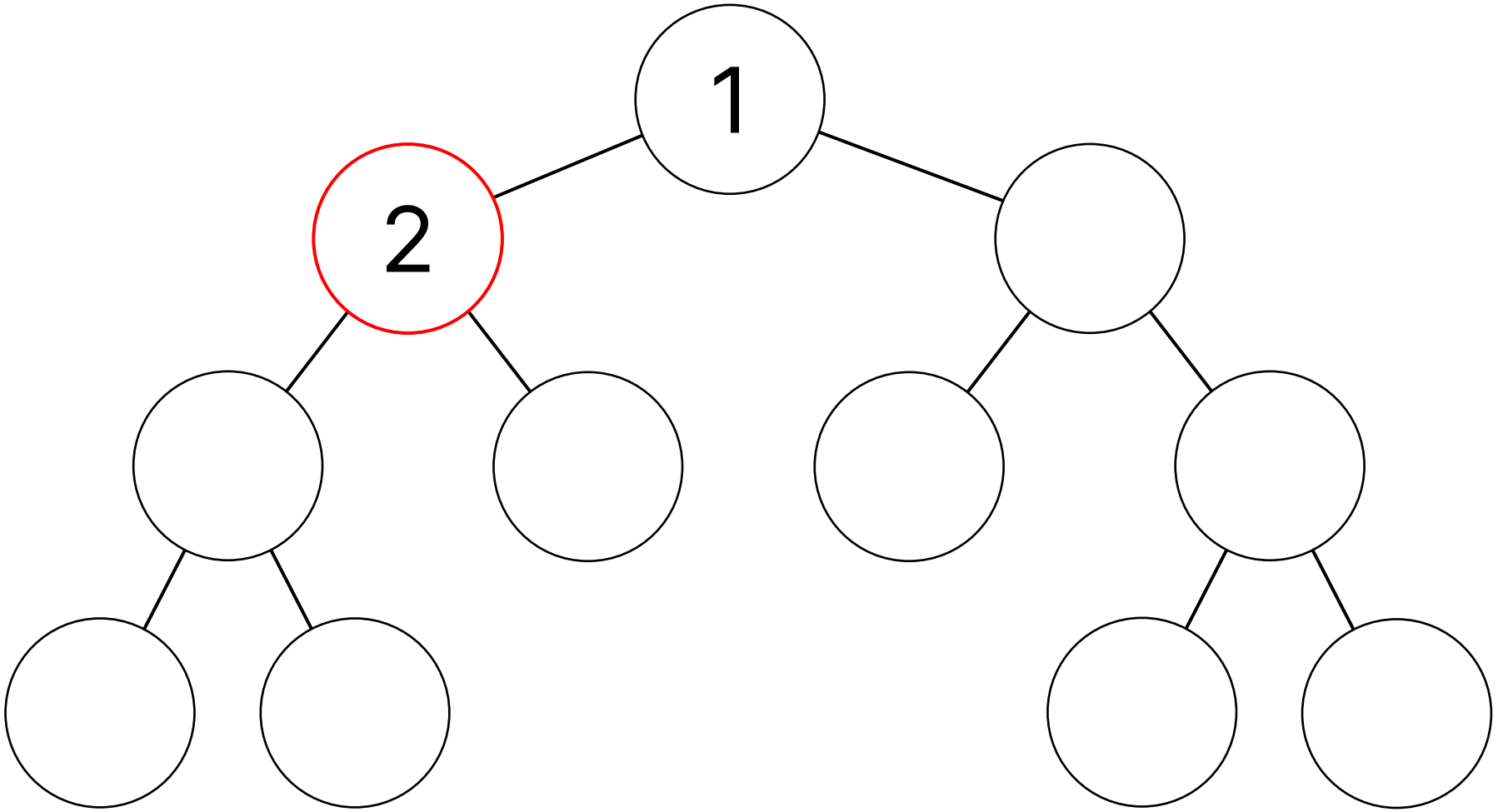
BFS



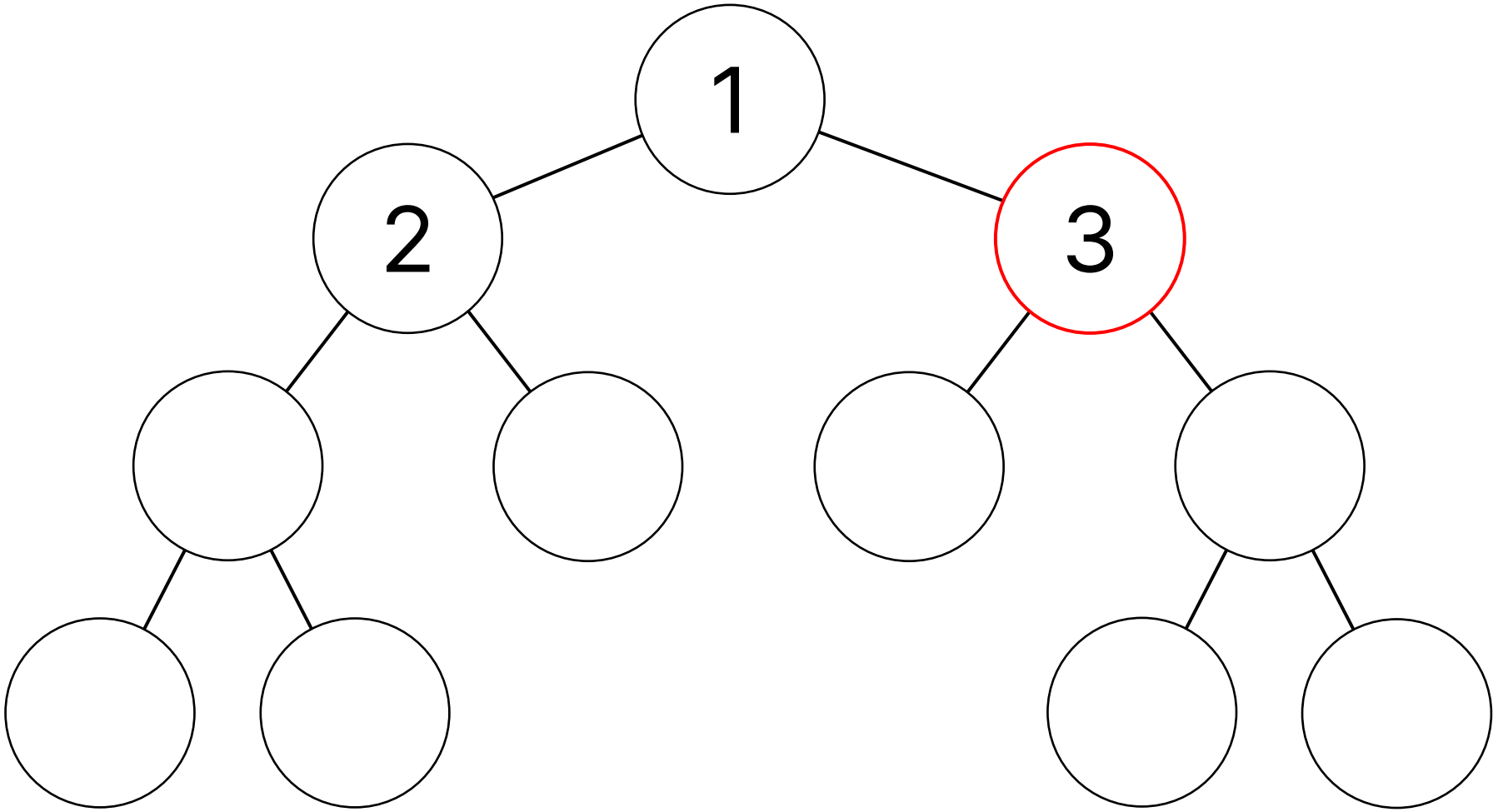
BFS



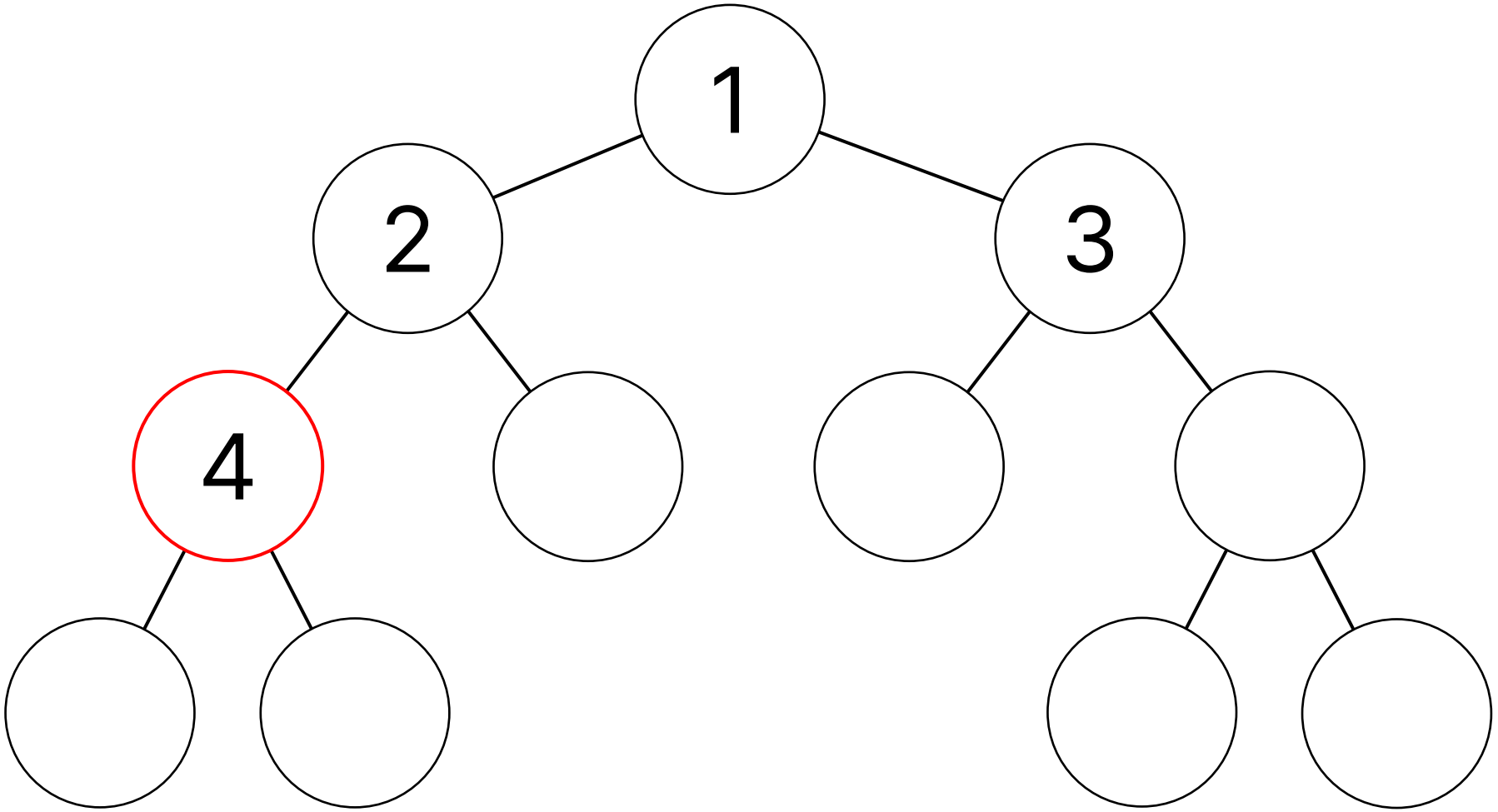
BFS



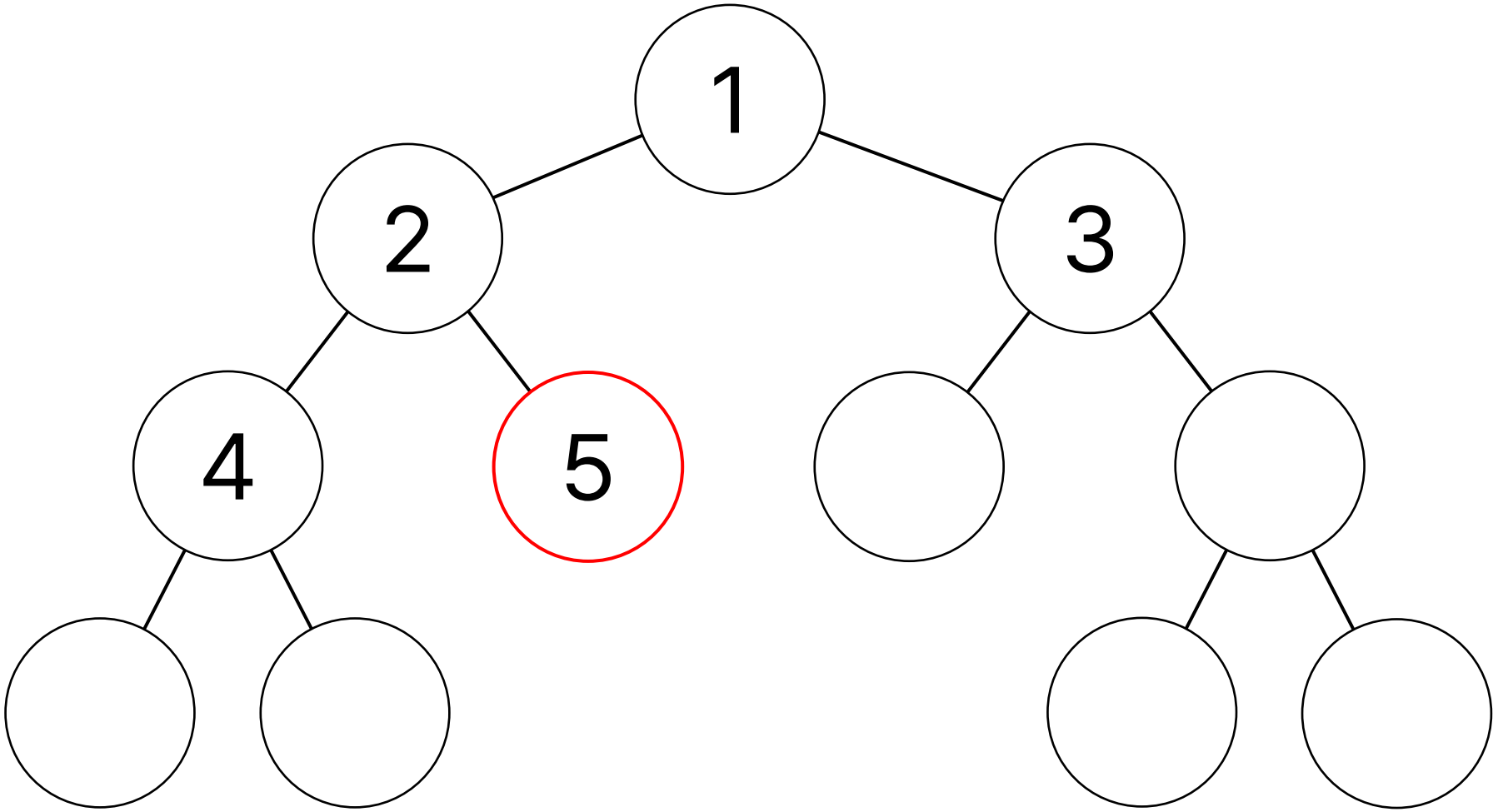
BFS



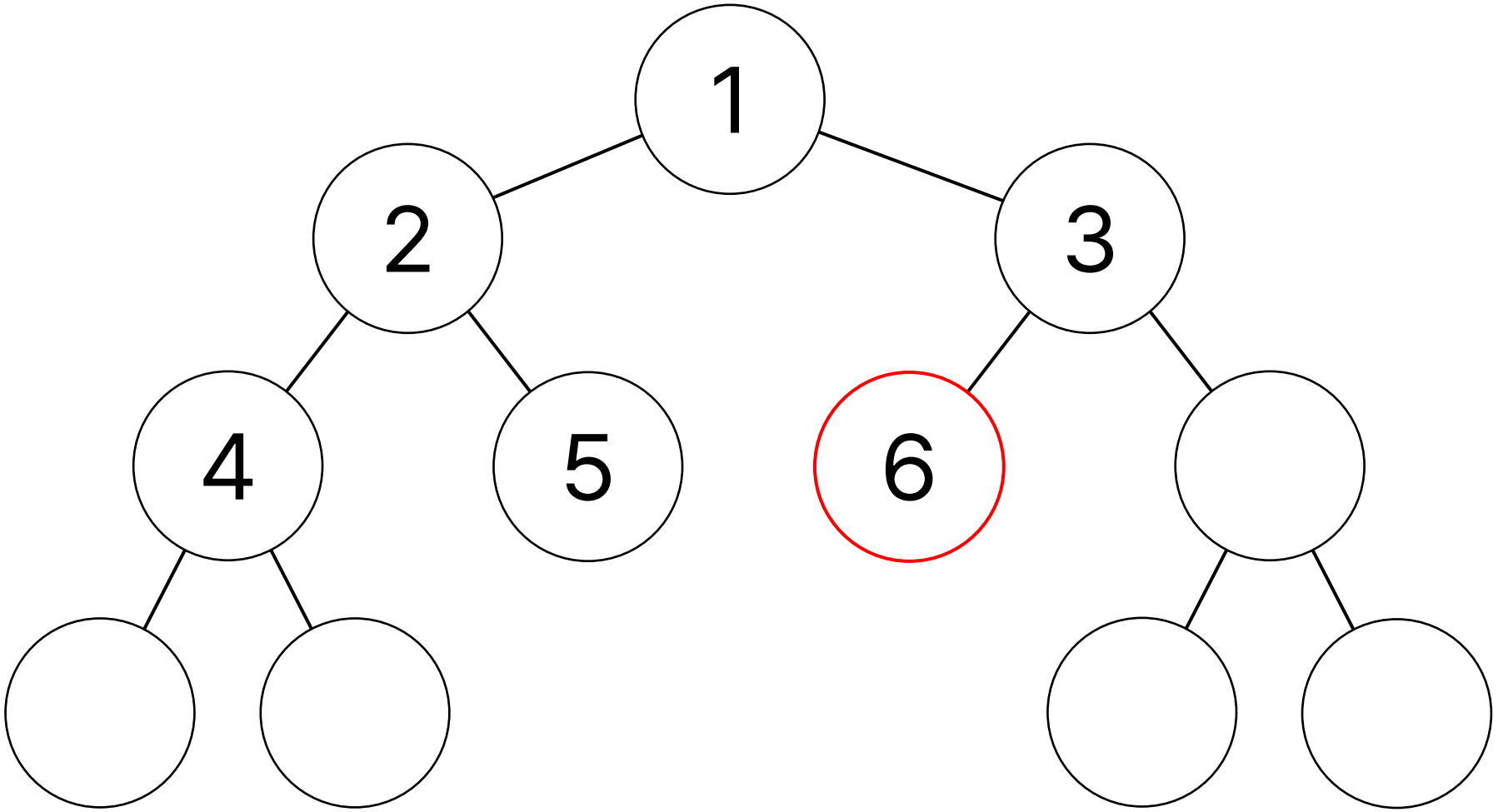
BFS



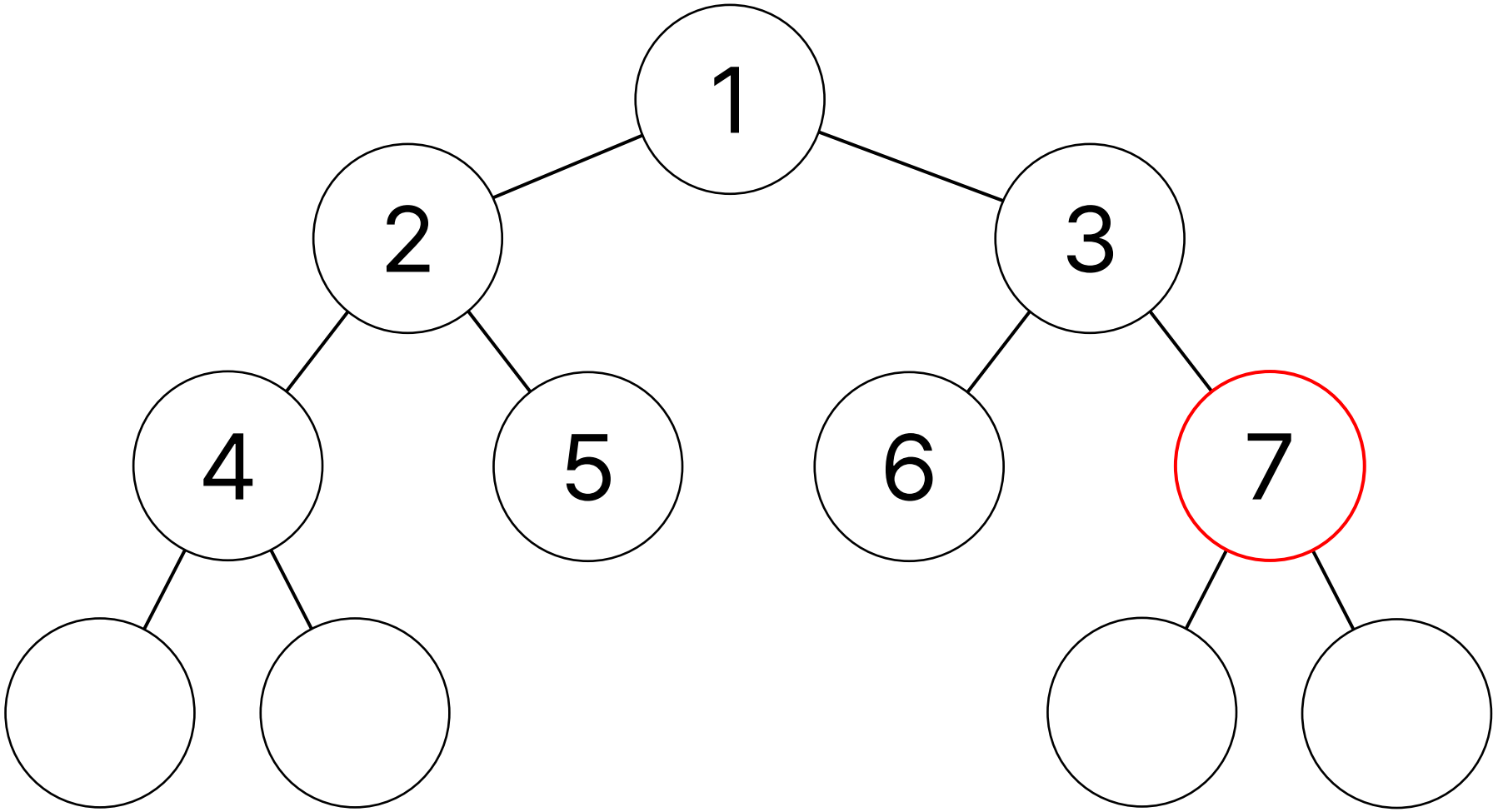
BFS



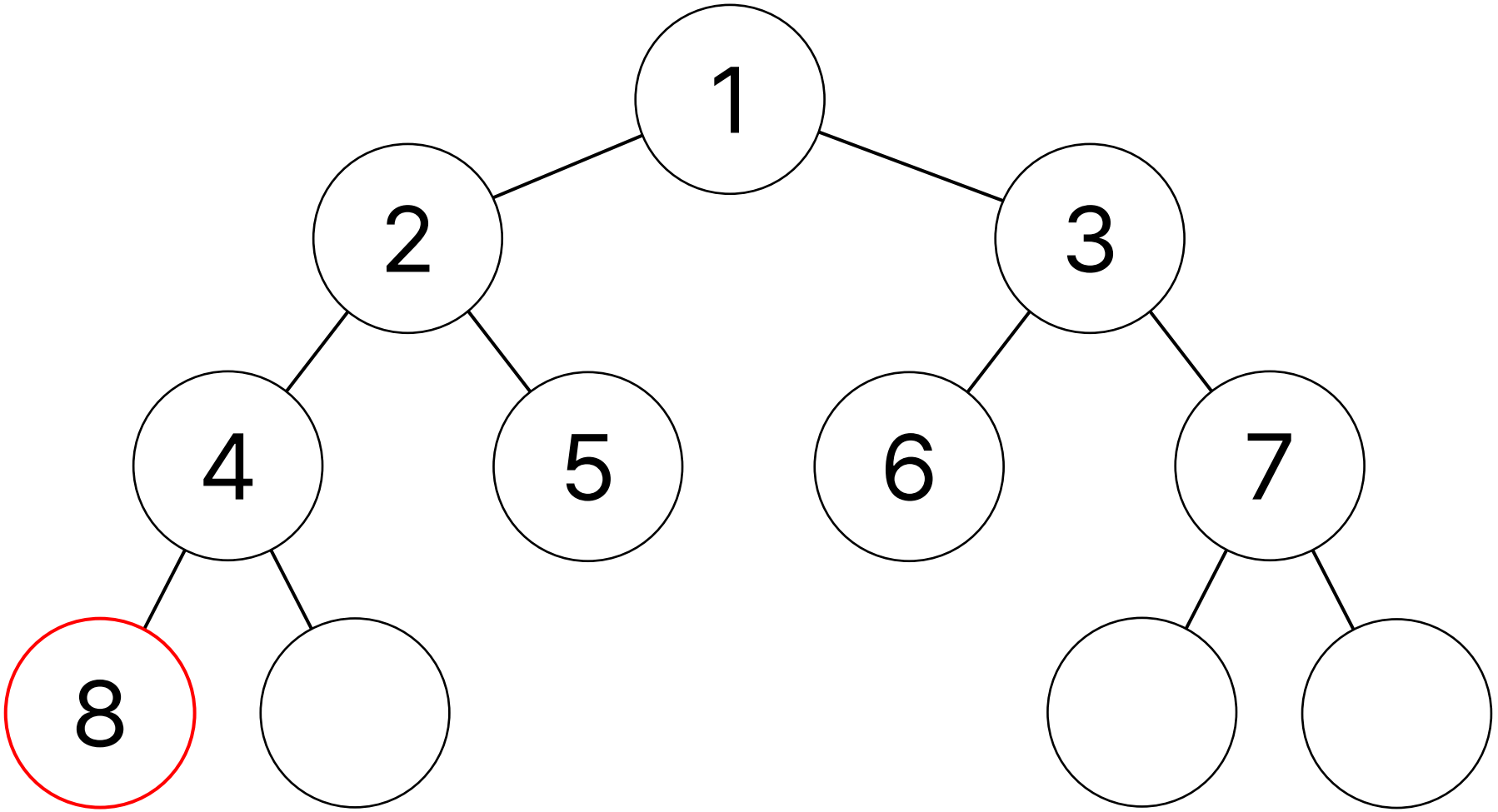
BFS



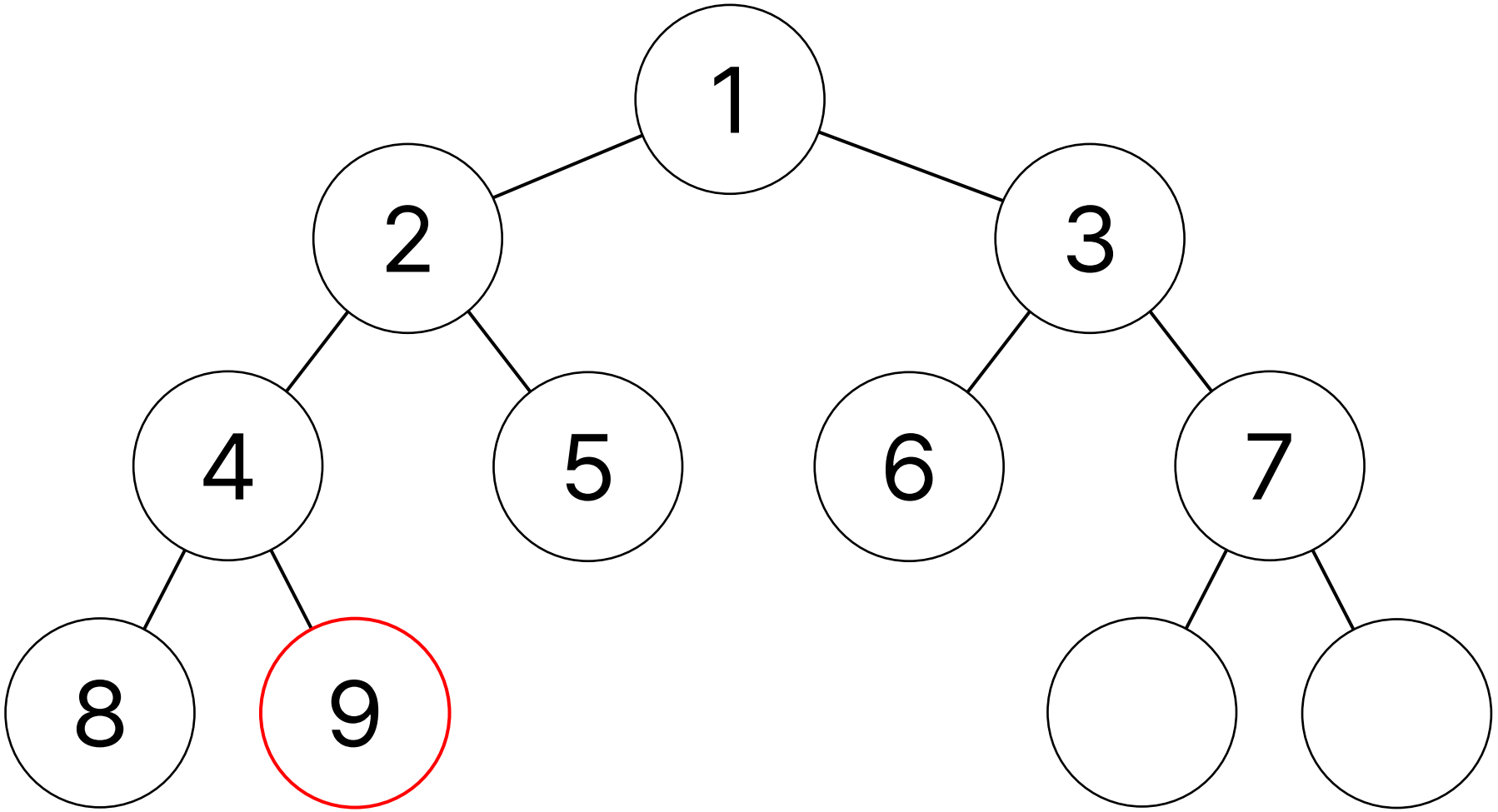
BFS



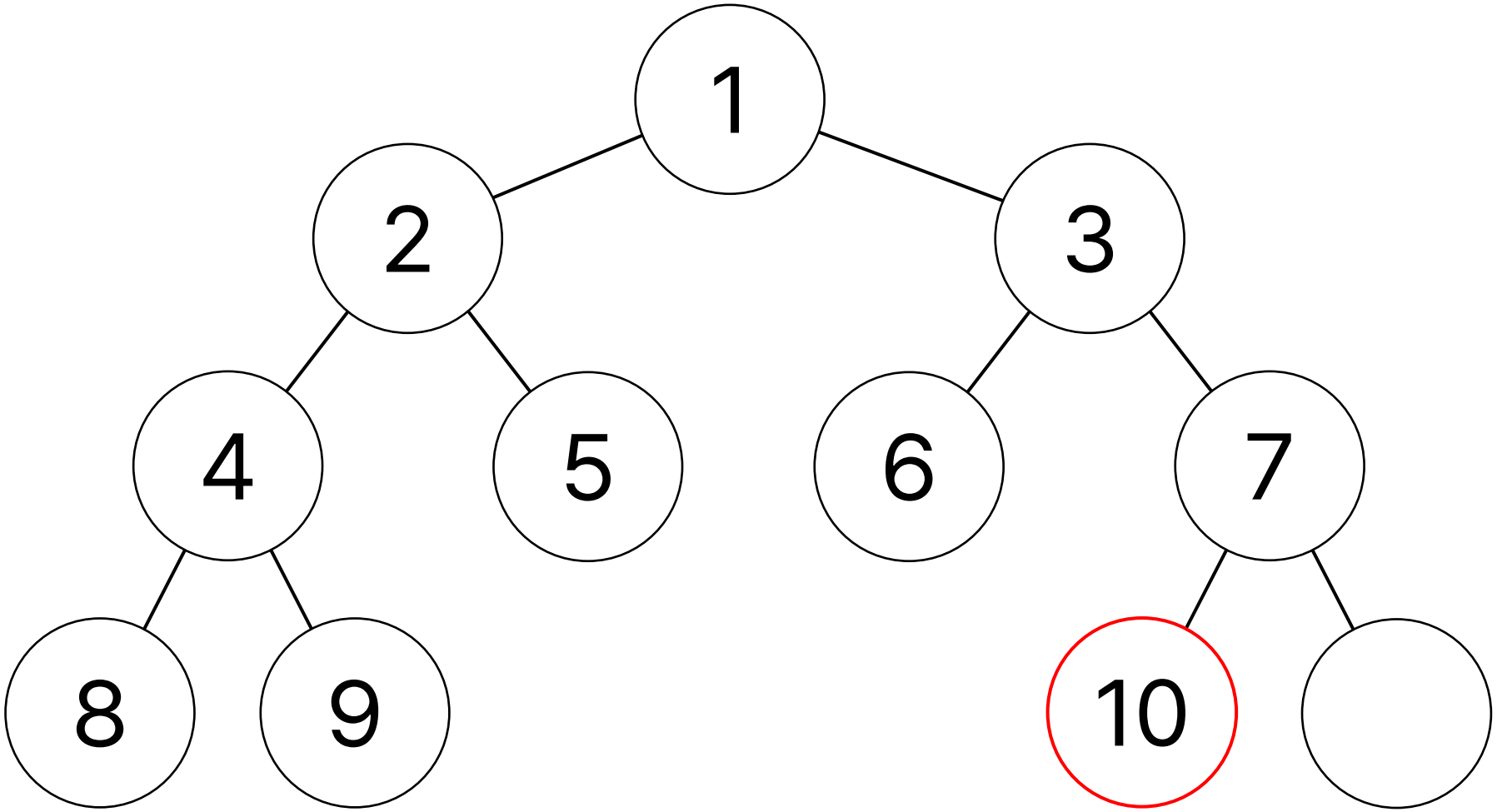
BFS



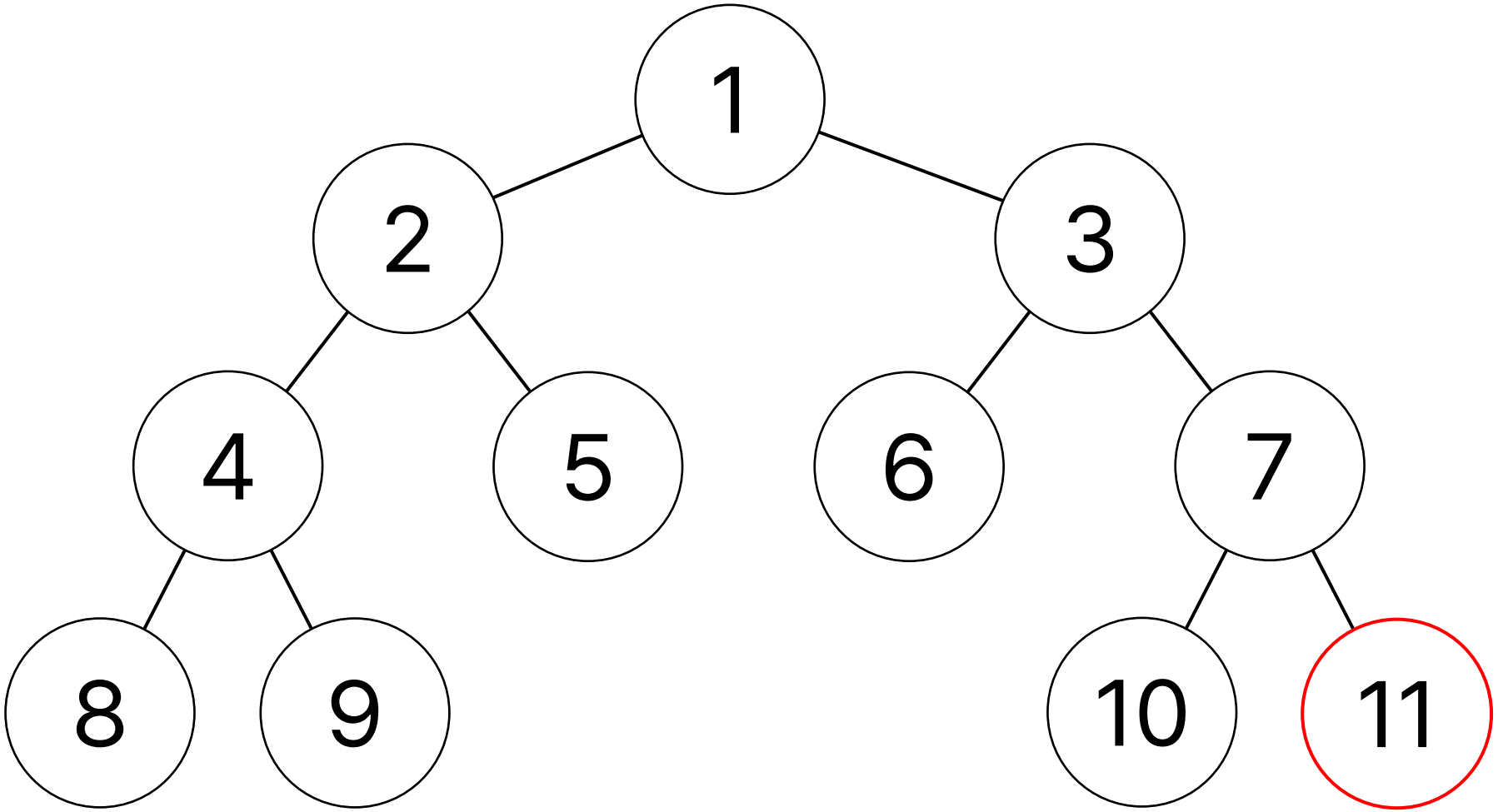
BFS



BFS



BFS



BFS

- 현재 노드와 이어져 있는 노드로 바로 이동할 수 없다
- 따라서 Queue를 이용해 다음에 탐색할 노드를 대기열로 관리한다
- 앞선 예시에서 1번 노드와 제일 가까운 노드는 2와 3이 있다

  2와 연결된 노드로 바로 이동하는 경우, 더 가까운 3번 노드를 건너뛰고 다음 노드를 탐색하기 때문에 순서가 잘못된다
- 따라서 큐를 이용해 탐색한다

BFS

- 마찬가지로 방문했는지 확인할 배열이 필요하다
- 대기열을 사용해야하므로 큐가 필요하다

BFS

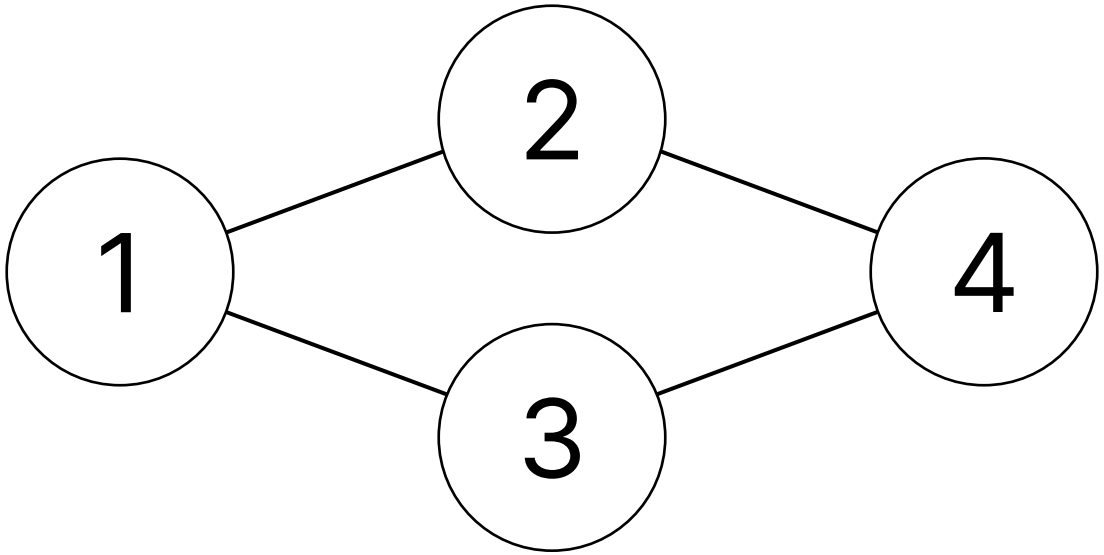
```cpp
vector<int> edges[MAX_NODE];
queue<int> bfs_queue;
bool visited[MAX_NODE];

void bfs(int start) {
    bfs_queue.push(start);
    visited[start] = true;

    while (!bfs_queue.empty()) {
        int cur = bfs_queue.front();
        bfs_queue.pop();
        for (auto next : edges[cur]) {
            if (!visited[next]) {
                bfs_queue.push(next);
                visited[next] = true;
            }
        }
    }
}
```

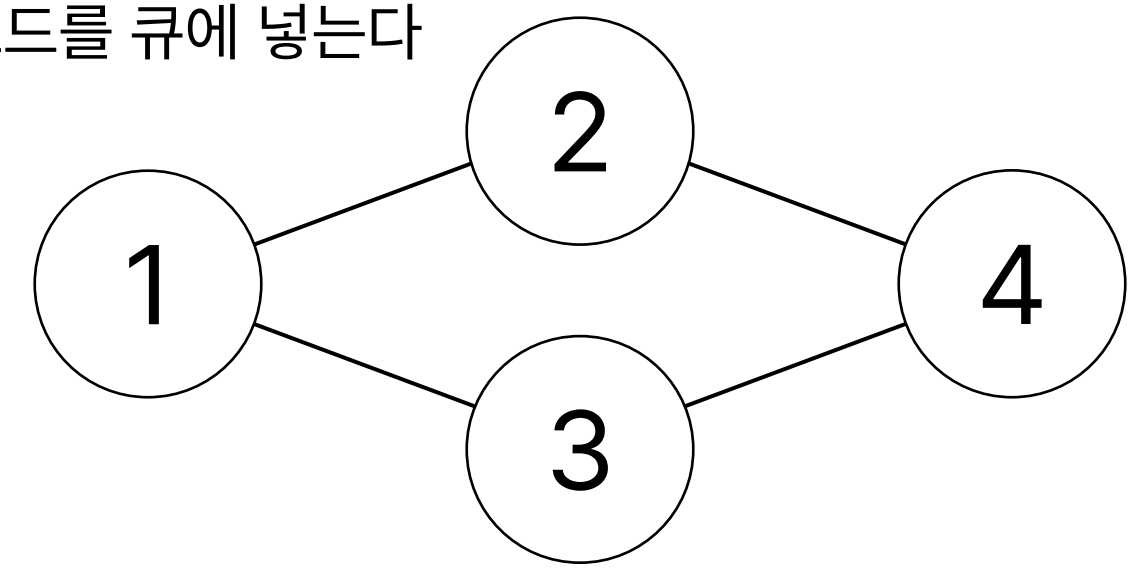
BFS

- DFS와 다르게 BFS에서는 큐에 넣을 때 방문 체크를 해야한다
- 다음 그래프를 살펴보자



BFS

- 1번 노드는 2번과 3번 노드를 큐에 넣는다
- 2번 노드는 4번 노드를 큐에 넣는다
- 3번 노드도 4번 노드를 큐에 넣는다



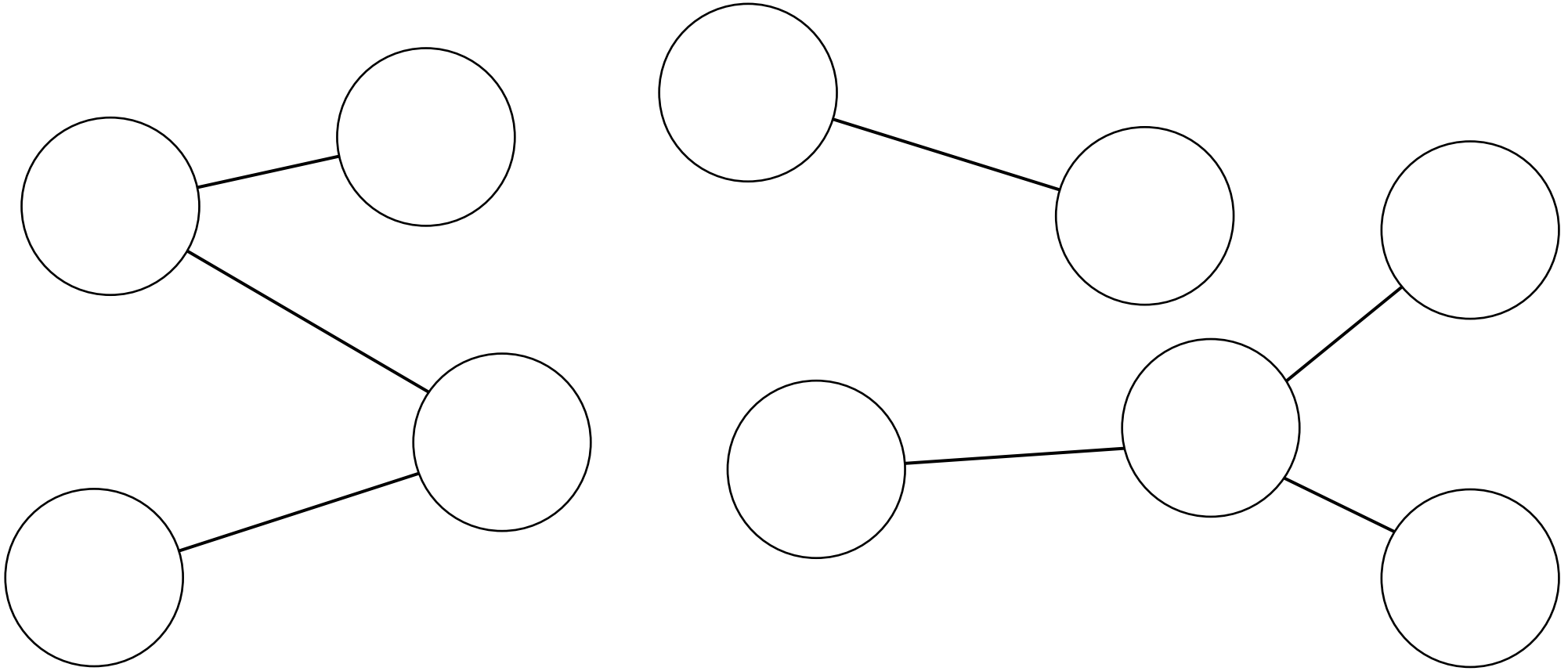
BFS

- 큐에 들어간 노드들은 탐색이 확정이 되었다
- 탐색하기전 해당 노드를 다시 큐에 넣는다면 중복된 노드가 큐에 들어간다
- 하나의 노드를 여러번 탐색하게 된다

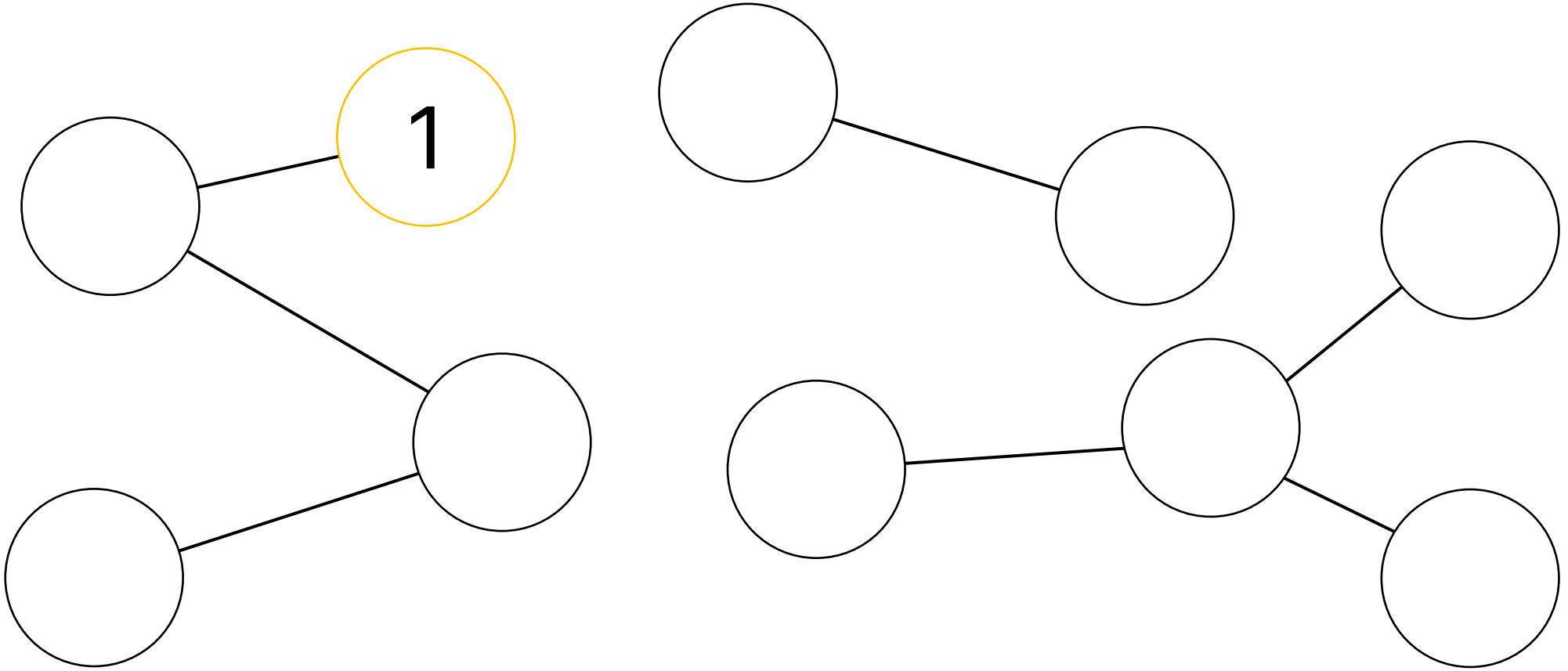
Traversal

- 연결된 모든 정점을 탐색하는 것
- 즉, 한 번의 순회로 하나의 연결 요소를 모두 탐색한다
- 전체 그래프에서 연결 요소의 개수를 세는 방법은 순회를 몇 번 하는 가로 셀 수 있다

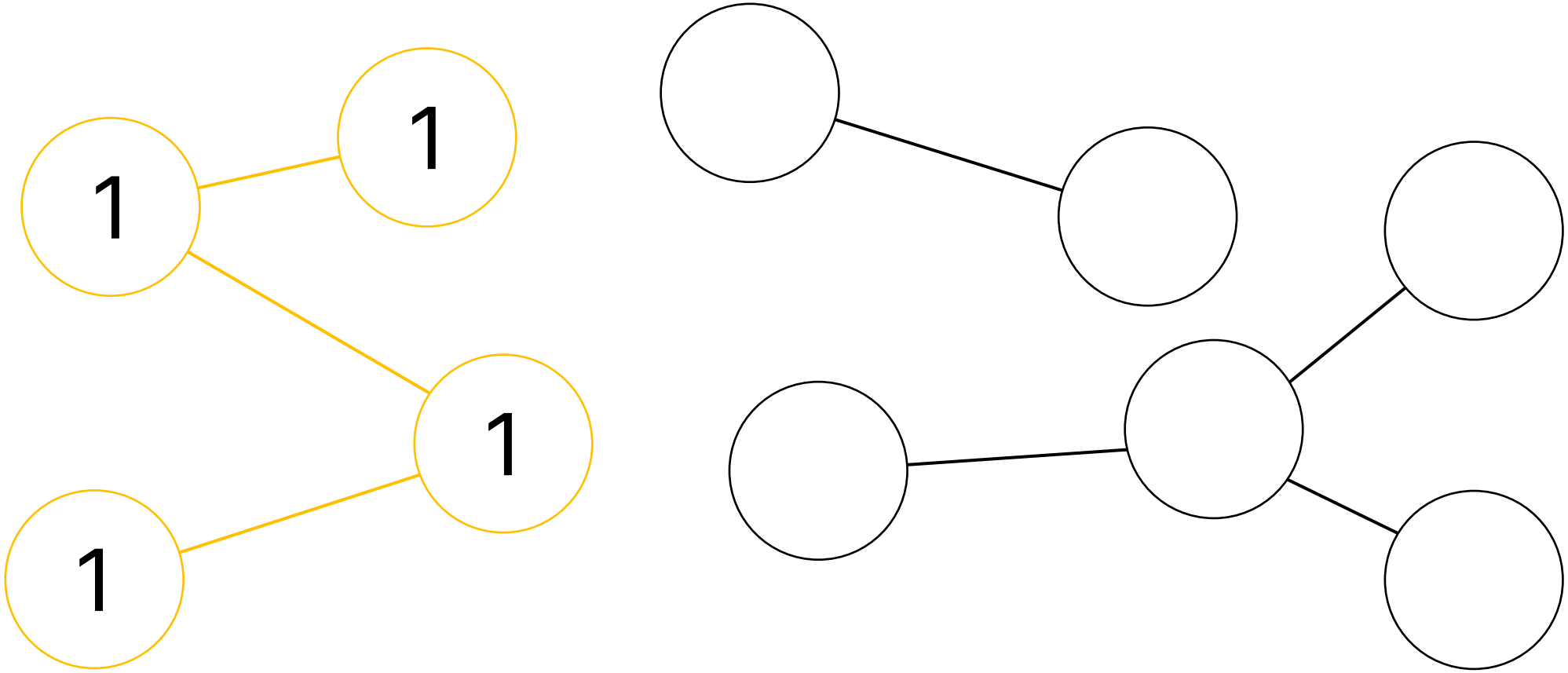
Connected Component



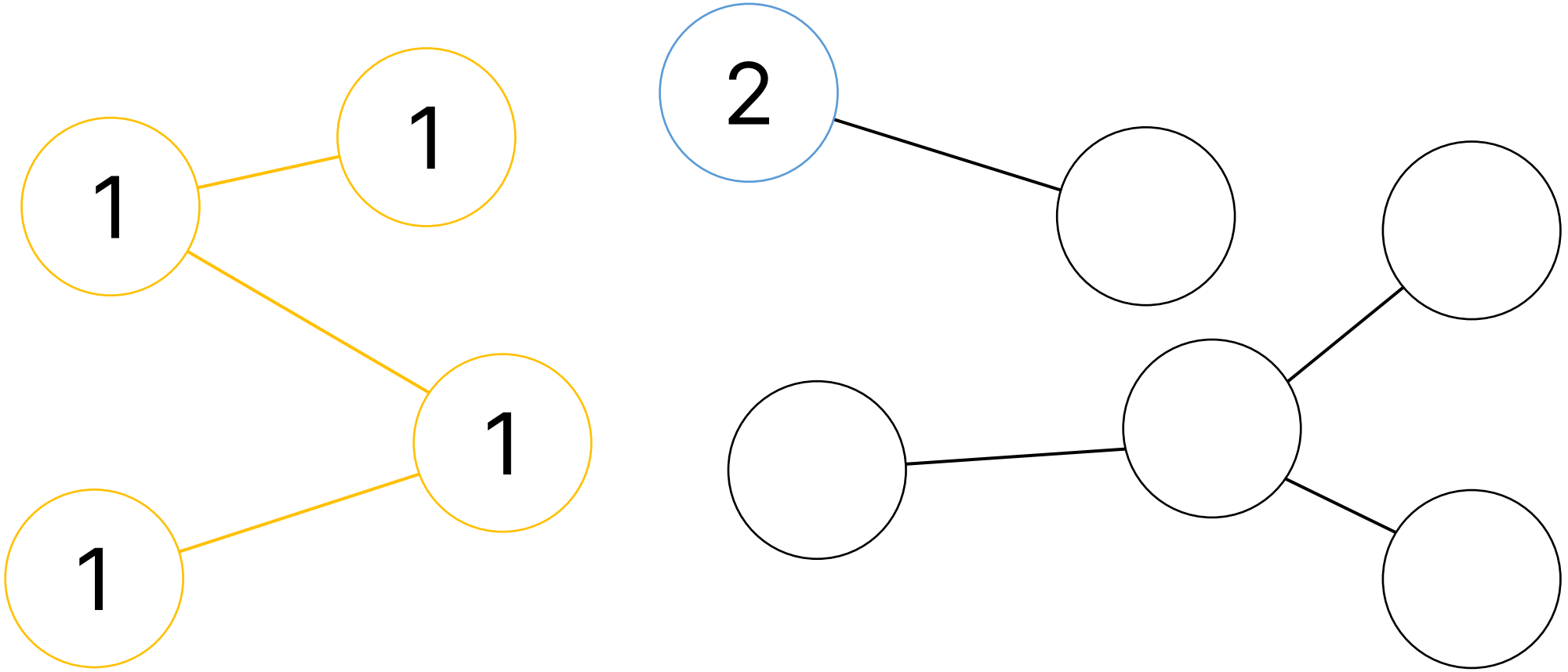
Connected Component



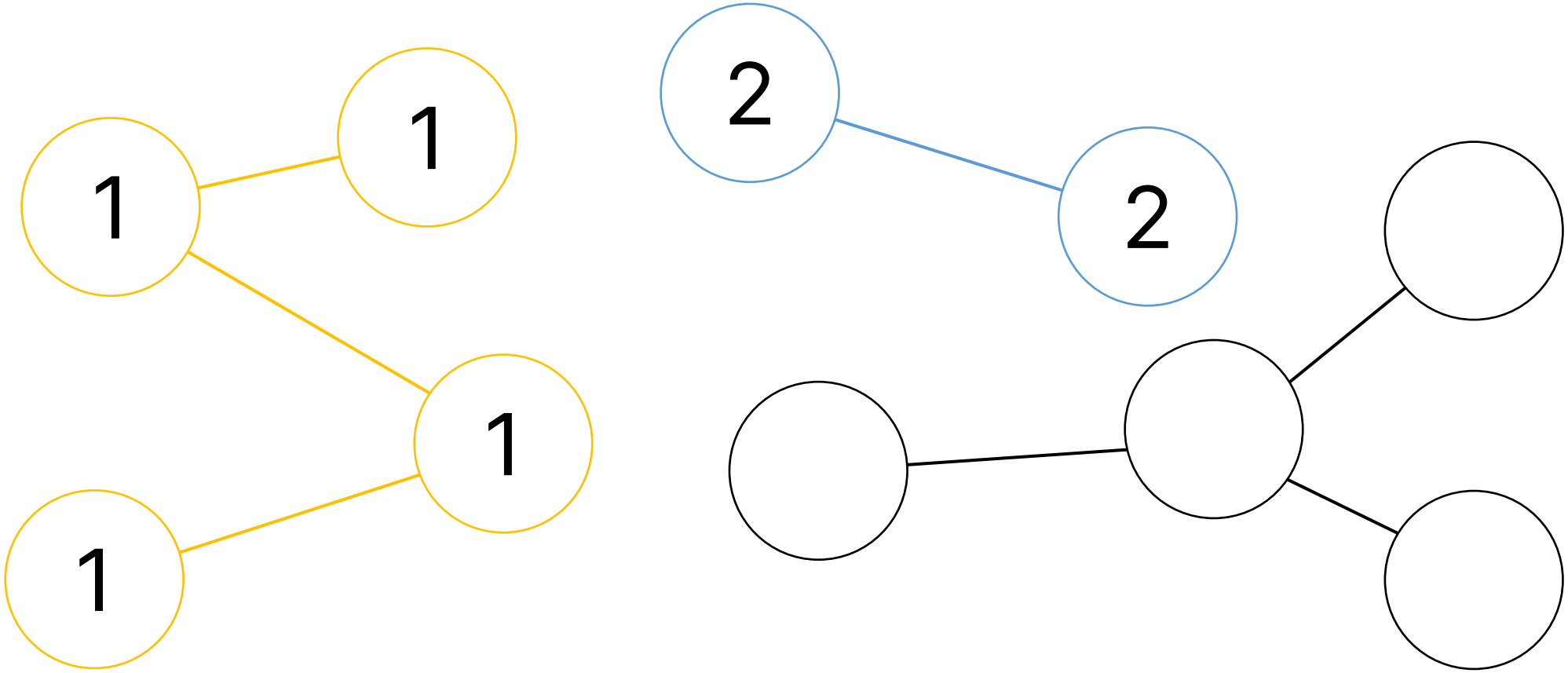
Connected Component



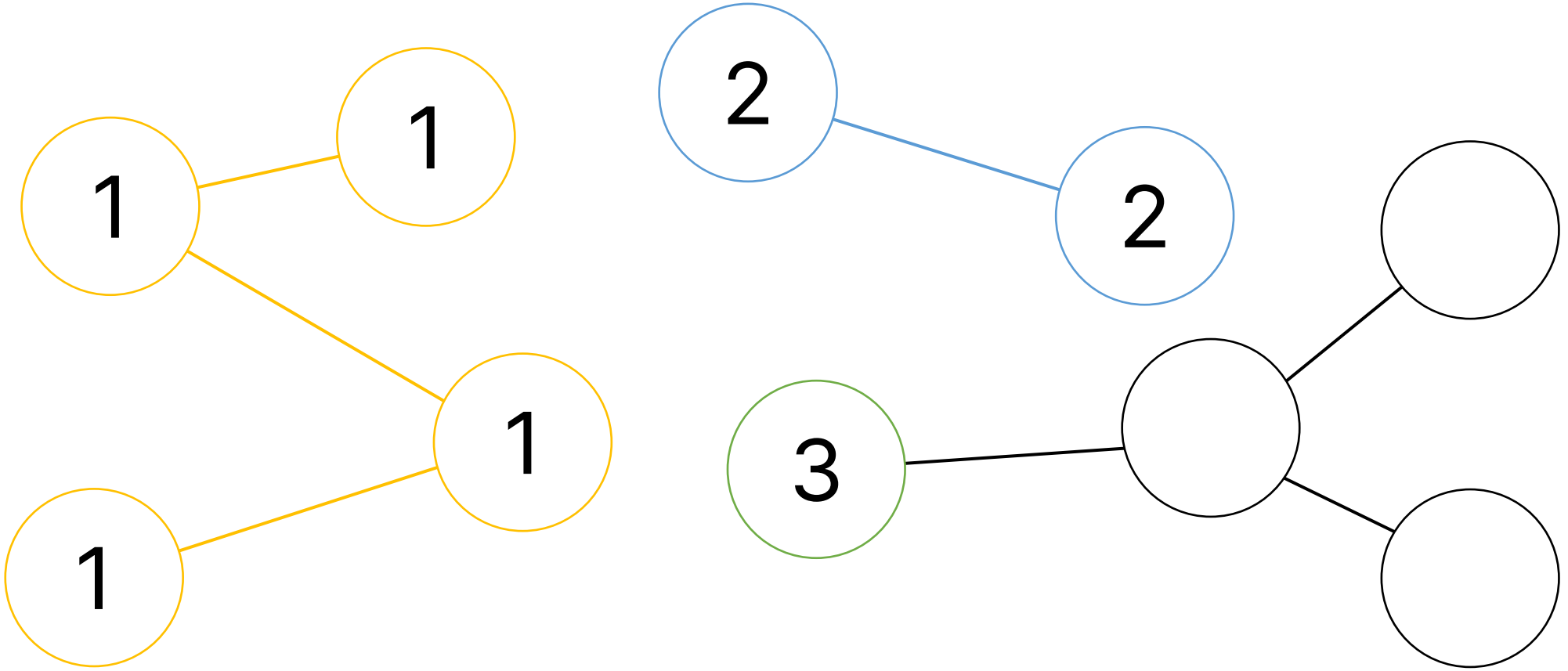
Connected Component



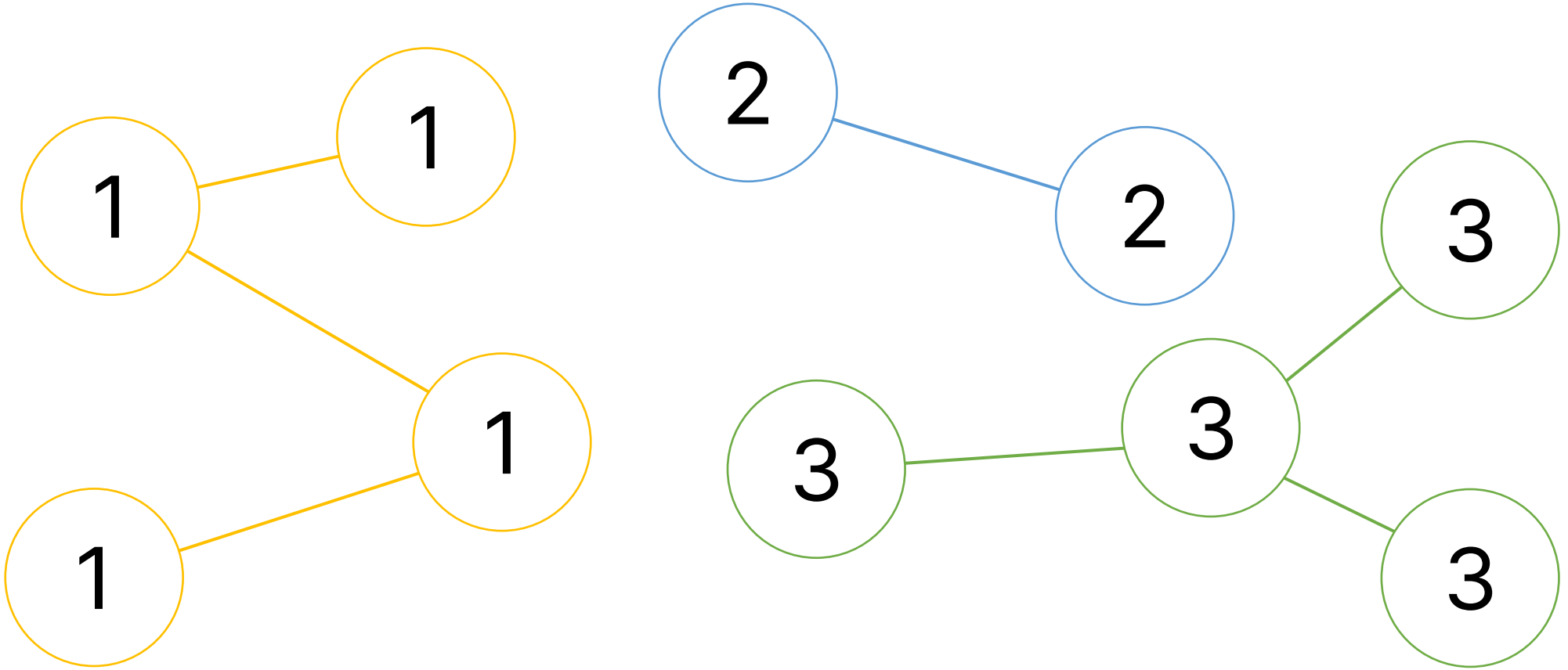
Connected Component



Connected Component



Connected Component



Connected Component

- 순회를 총 3번 실행해 모든 노드를 방문했으므로 연결 요소는 3개이다

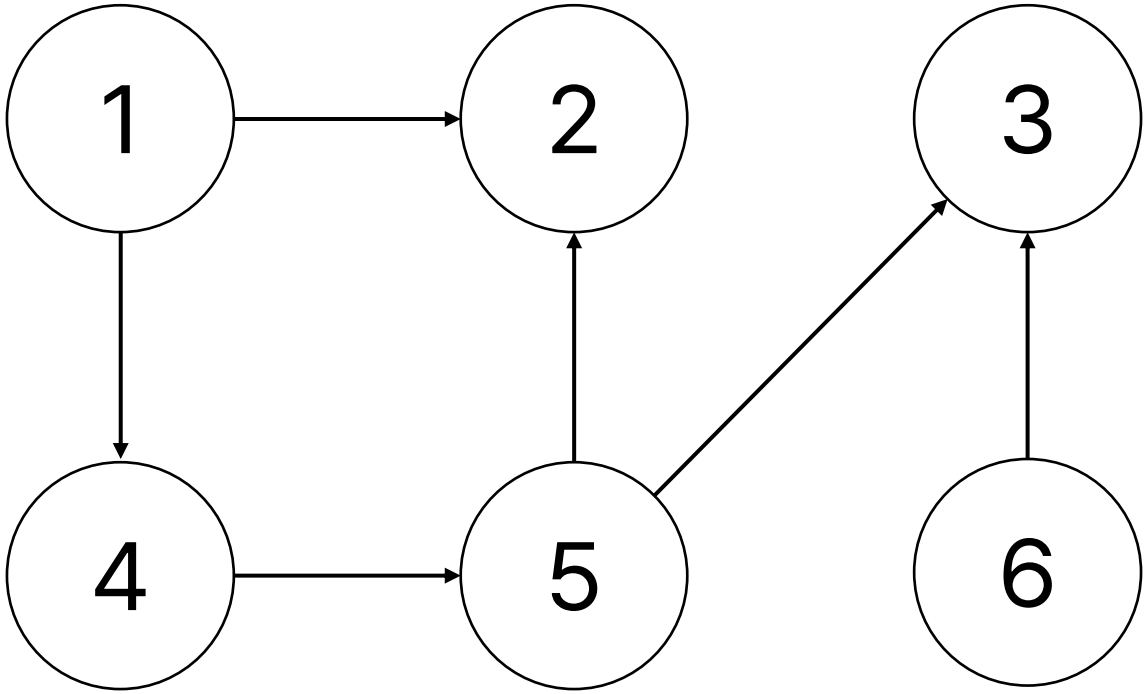
Topological sorting

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

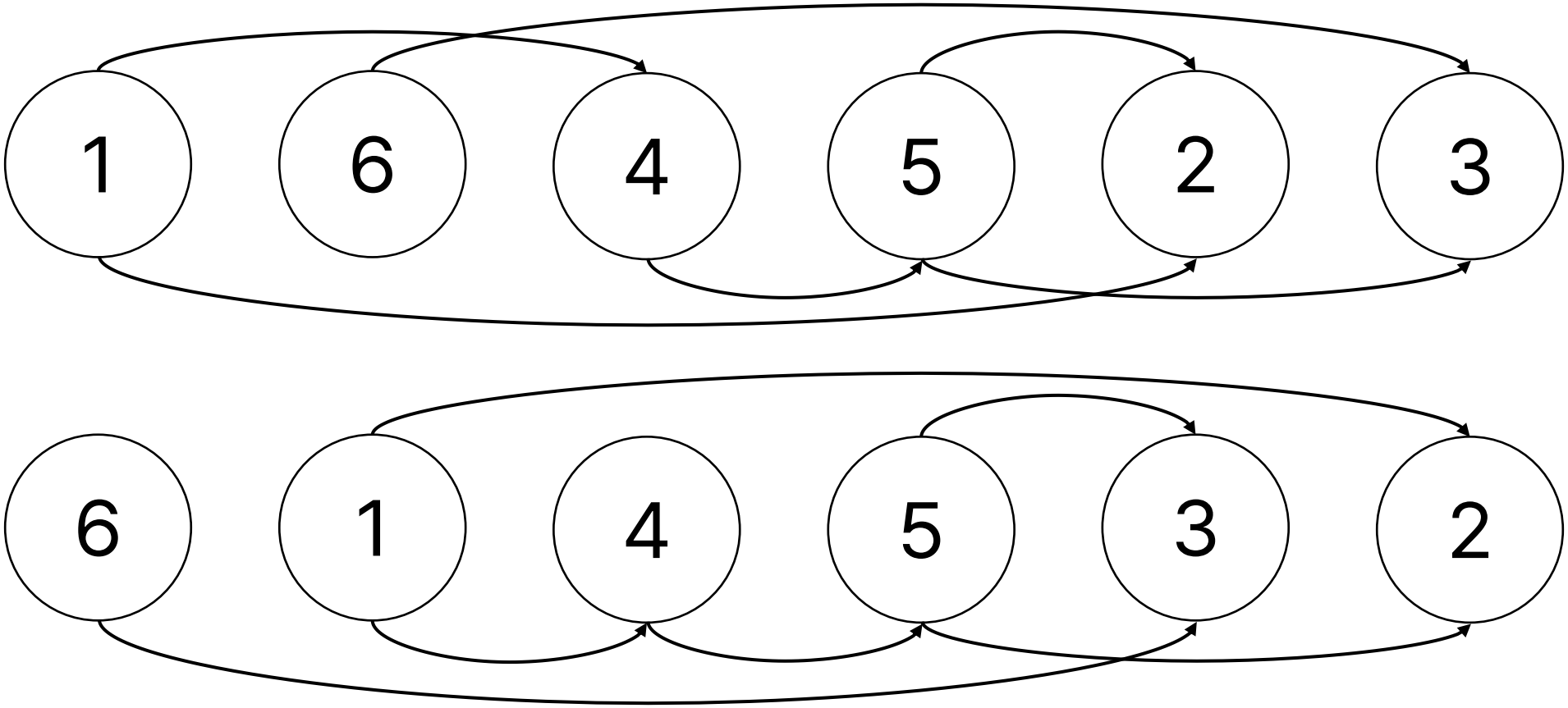
Topological sorting

- 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

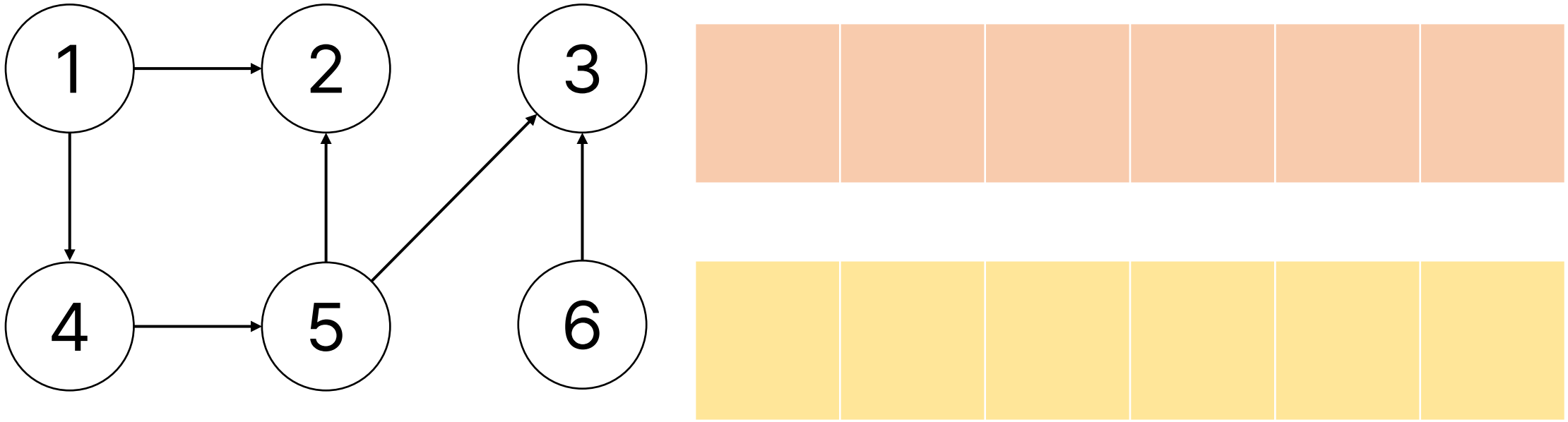
Topological sorting BFS

- 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

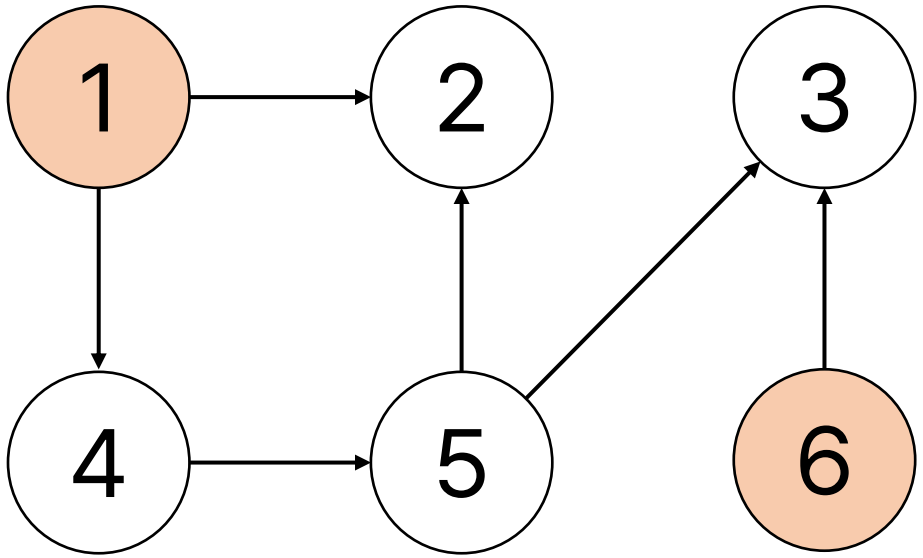
Topological sorting BFS

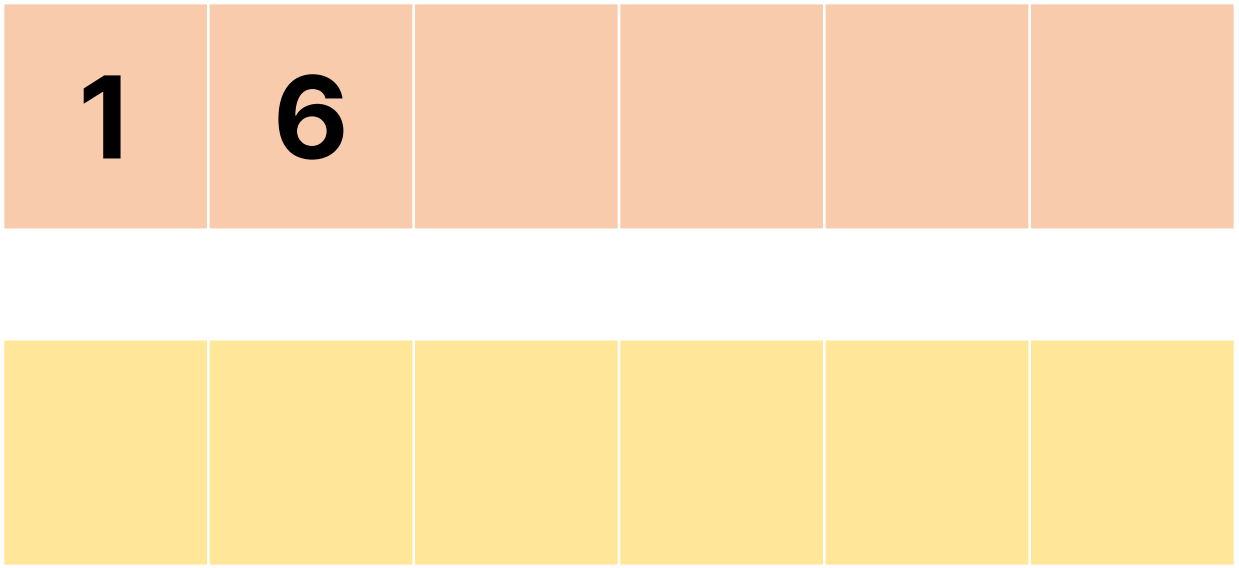
- 만일 사이클이 존재한다면 사이클에 포함된 모든 노드들은 입력 차수가 1이상이다
- 해당 노드들은 어떠한 경우에도 큐에 들어갈 수 없으므로 탐색되지 않는다
- 앞선 BFS를 마치고 나서 탐색되지 않은 노드들이 존재한다면 사이클이 존재하는 그래프, 위상 정렬이 불가능한 그래프이다

Topological sorting BFS

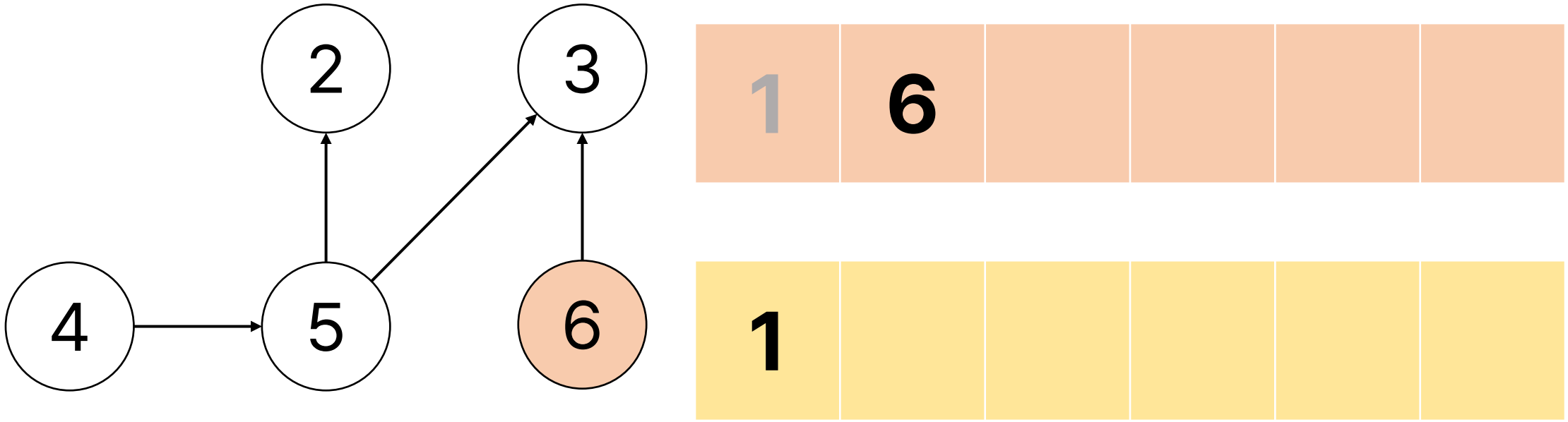


Topological sorting BFS

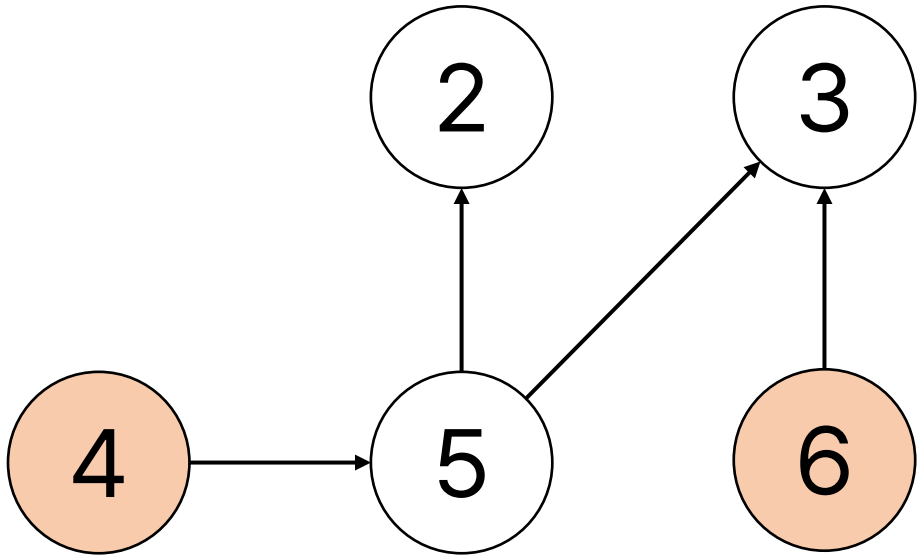




Topological sorting BFS



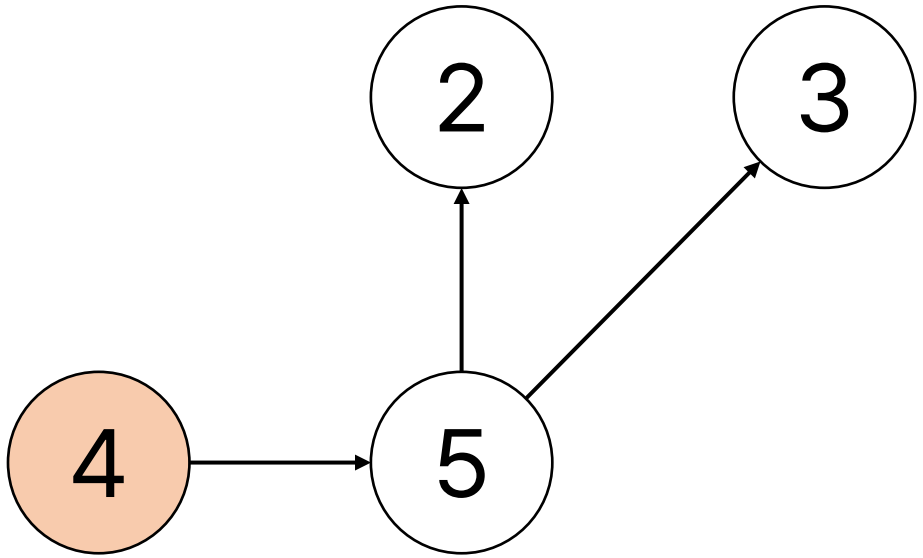
Topological sorting BFS

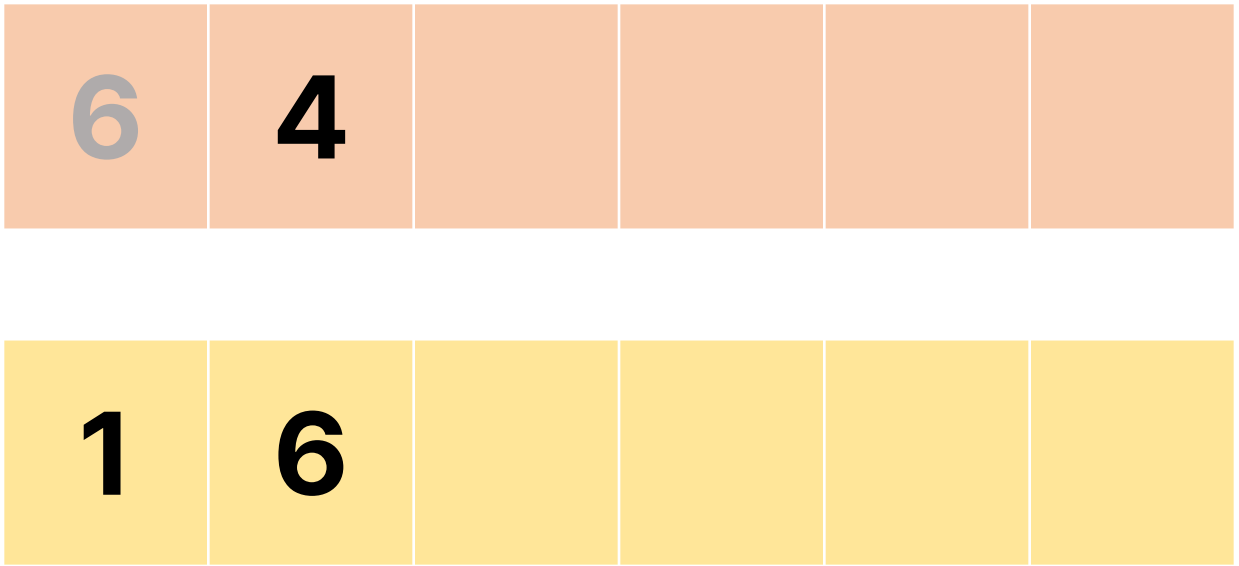


1	6	4			

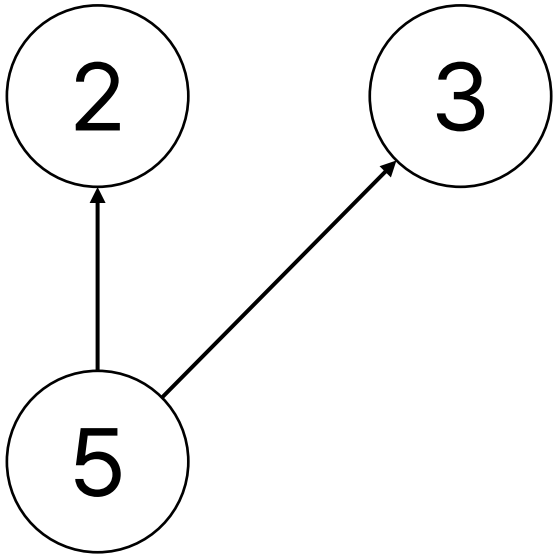
1					

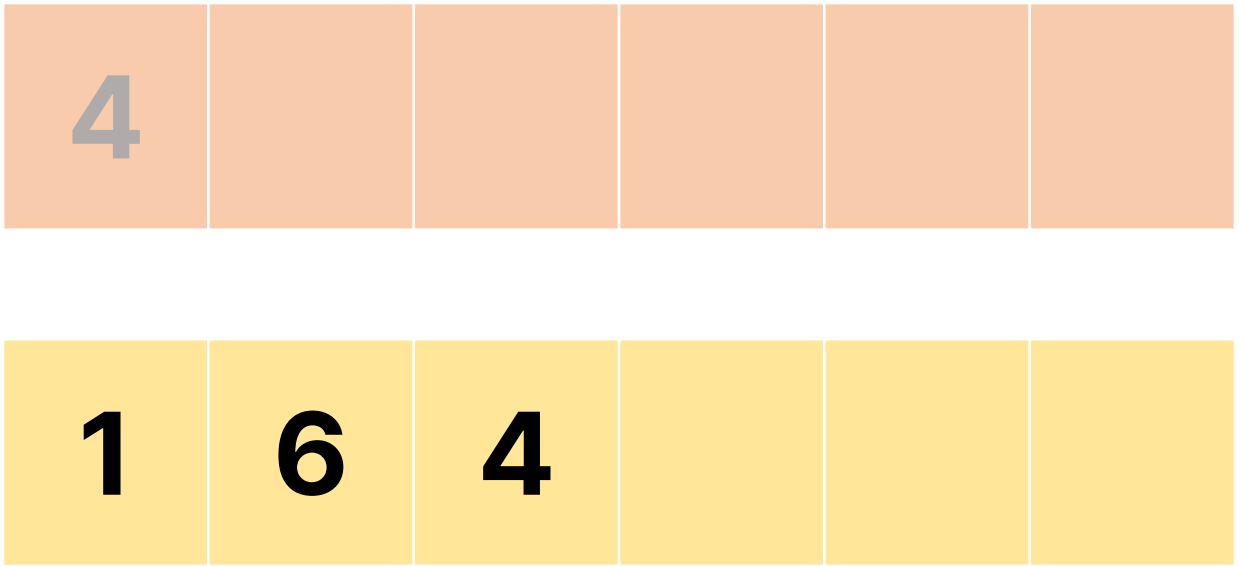
Topological sorting BFS



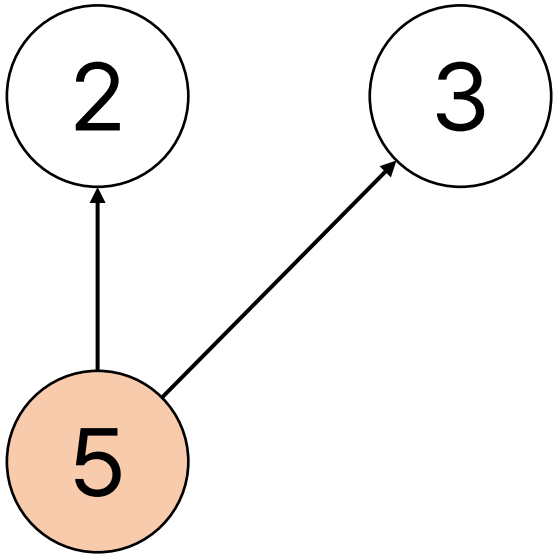


Topological sorting BFS





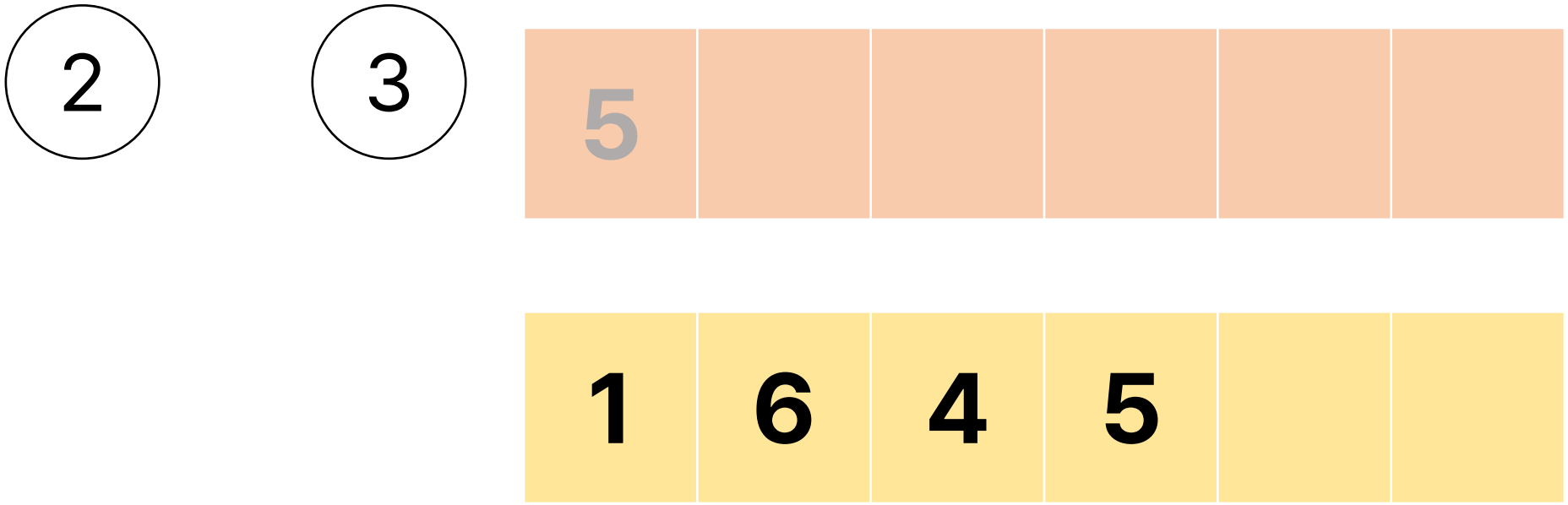
Topological sorting BFS



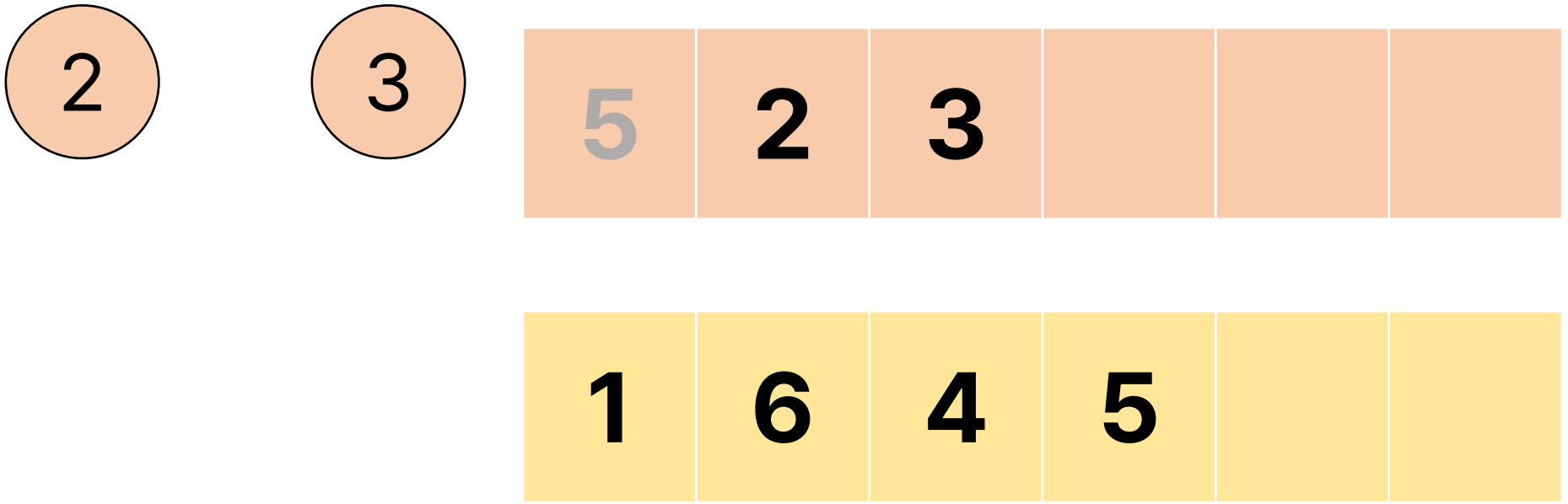
4	5				

1	6	4			

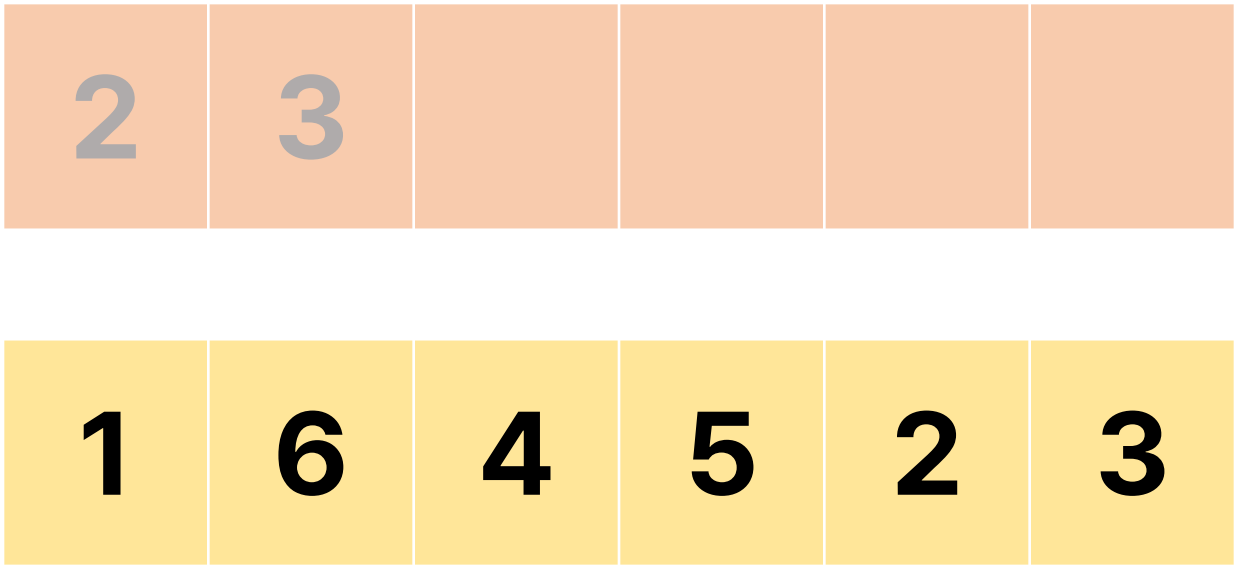
Topological sorting BFS



Topological sorting BFS



Topological sorting BFS



Topological sorting BFS

```cpp
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

```cpp
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);
}
```

연습 문제

4963 : 섬의 개수

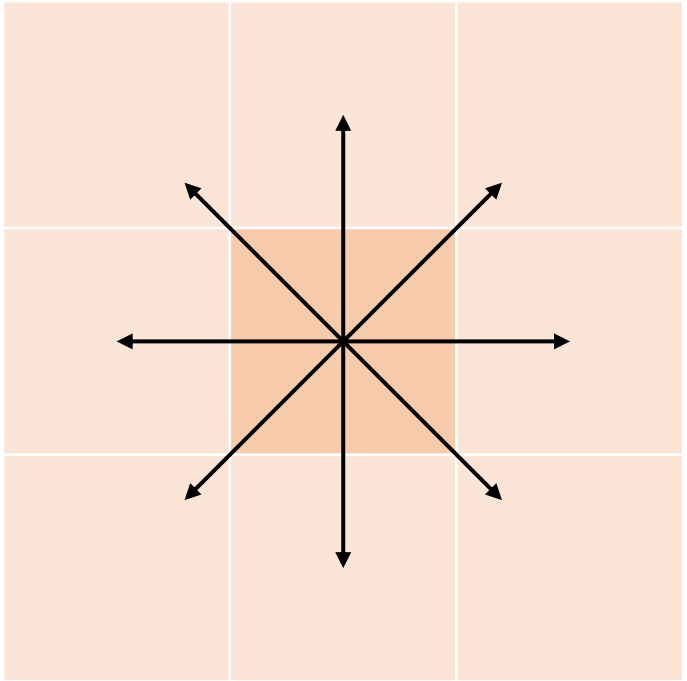
섬의 개수 BOJ 4963

- 하나의 섬은 모두 연결되어 있다
- 하나의 연결요소는 하나의 섬이다
- 연결요소의 개수가 곧 섬의 개수다

- 여러 개의 테스트 케이스로 이루어져 있으므로 초기화에 유의하자

섬의 개수 BOJ 4963

- 가로, 세로, 대각선 방향으로 이어져 있으므로 간선이 존재한다고 생각할 수 있다



섬의 개수 BOJ 4963

```
void dfs(int x, int y) {
    visited[x][y] = true;
    if (x + 1 < h and arr[x + 1][y] and !visited[x + 1][y])
        dfs(x + 1, y);
    if (y + 1 < w and arr[x][y + 1] and !visited[x][y + 1])
        dfs(x, y + 1);
    if (x - 1 >= 0 and arr[x - 1][y] and !visited[x - 1][y])
        dfs(x - 1, y);
    if (y - 1 >= 0 and arr[x][y - 1] and !visited[x][y - 1])
        dfs(x, y - 1);
    if (x + 1 < h and y + 1 < w and arr[x + 1][y + 1] and !visited[x + 1][y + 1])
        dfs(x + 1, y + 1);
    if (x + 1 < h and y - 1 >= 0 and arr[x + 1][y - 1] and !visited[x + 1][y - 1])
        dfs(x + 1, y - 1);
    if (x - 1 >= 0 and y + 1 < w and arr[x - 1][y + 1] and !visited[x - 1][y + 1])
        dfs(x - 1, y + 1);
    if (x - 1 >= 0 and y - 1 >= 0 and arr[x - 1][y - 1] and !visited[x - 1][y - 1])
        dfs(x - 1, y - 1);
}
```

섬의 개수 BOJ 4963

- 굉장히 복잡해진 코드가 나온다
- 모든 노드가 이동하는 방향은 동일하다
- 좌표 평면에서는 상하좌우 또는 상하좌우대각선으로 이동하는 경우가 많다
- 이를 이용해 dx, dy배열로 반복문으로 순회를 할 수 있다

섬의 개수 BOJ 4963

```
bool valid(int x, int y) {
    if (x < 0 or h <= x)
        return false;
    if (y < 0 or w <= y)
        return false;
    return arr[x][y] and !visited[x][y];
}
```

섬의 개수 BOJ 4963

```
int dx[] = {1, -1, 0, 0, 1, 1, -1, -1};
int dy[] = {0, 0, 1, -1, 1, -1, 1, -1};

void dfs(int x, int y) {
    visited[x][y] = true;
    for (int i = 0; i < 8; i++) {
        if (valid(x + dx[i], y + dy[i]))
            dfs(x + dx[i], y + dy[i]);
    }
}
```

Reference

- https://github.com/justiceHui/SSU-SCCC-Study
- https://github.com/justiceHui/Sunrin-SHARC