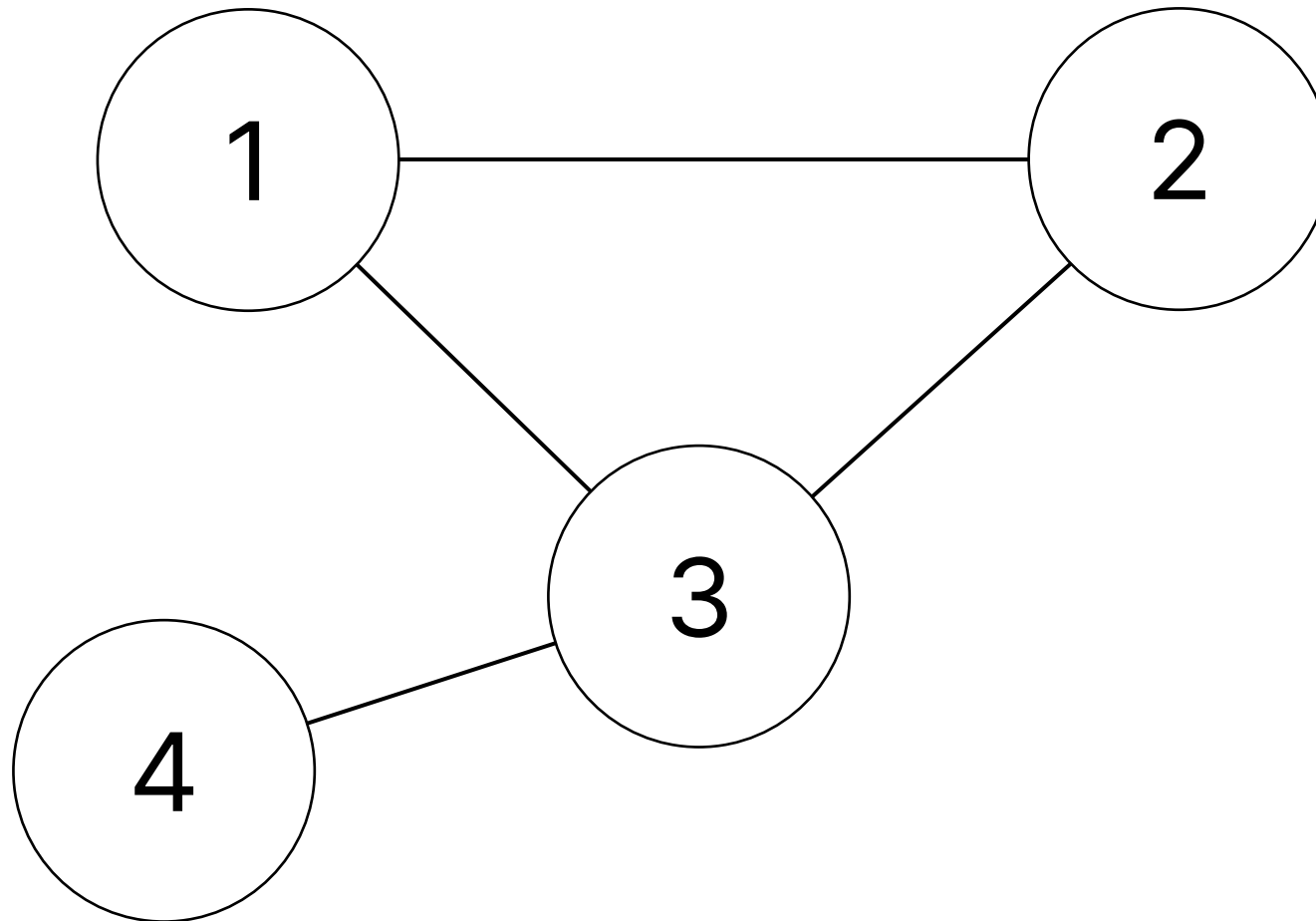


Graph

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

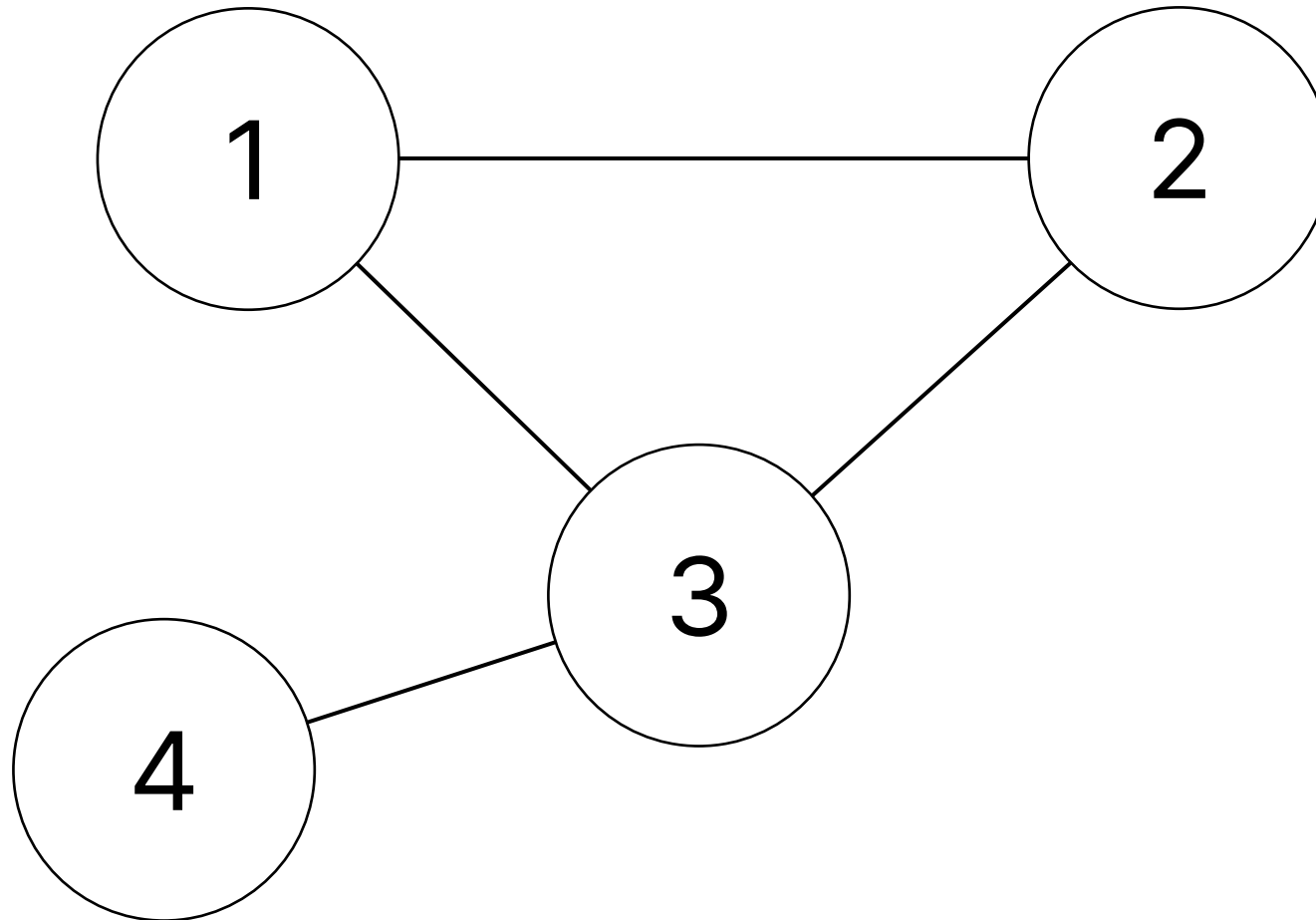
Graph



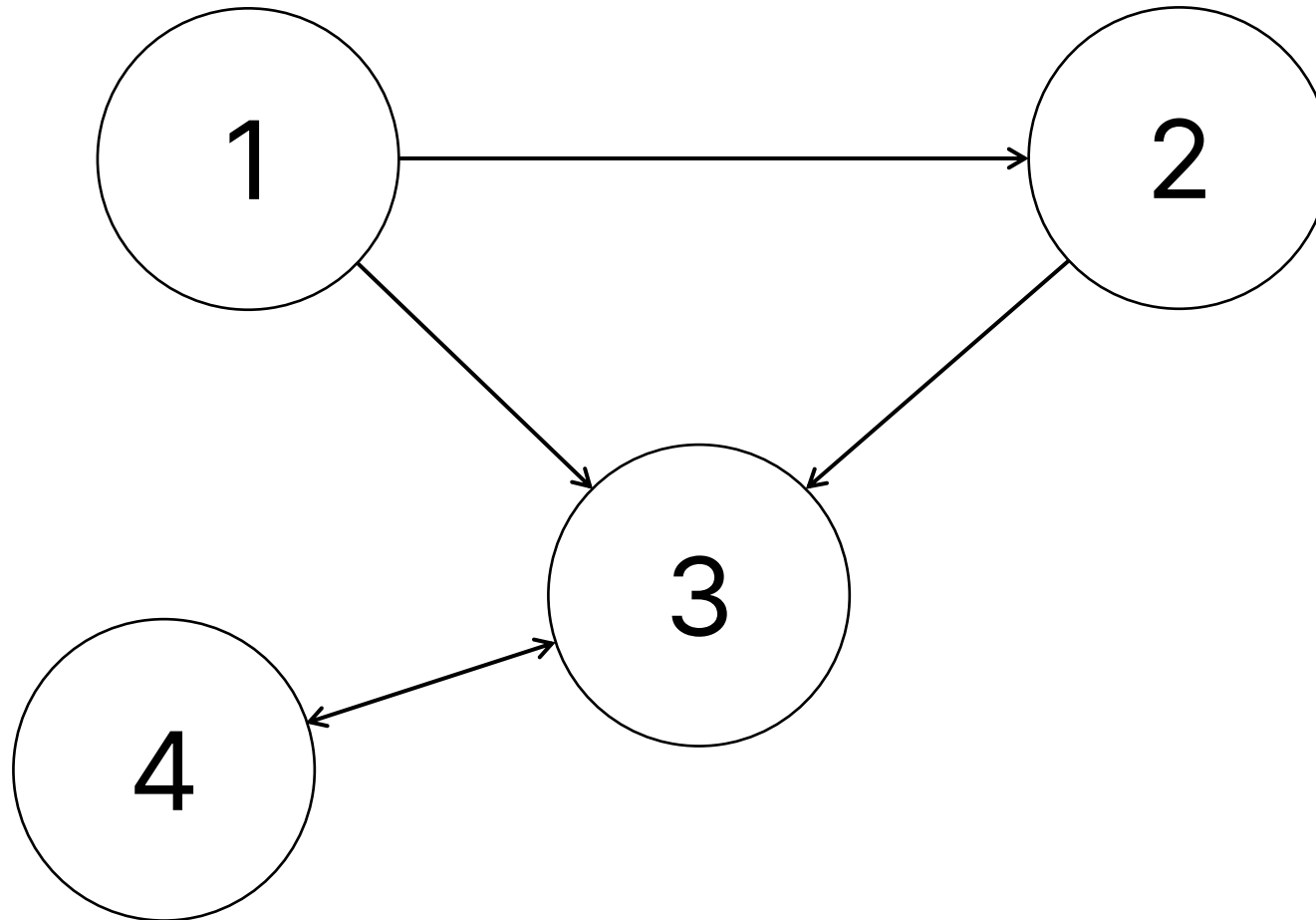
Directed, Undirected Graph

- 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

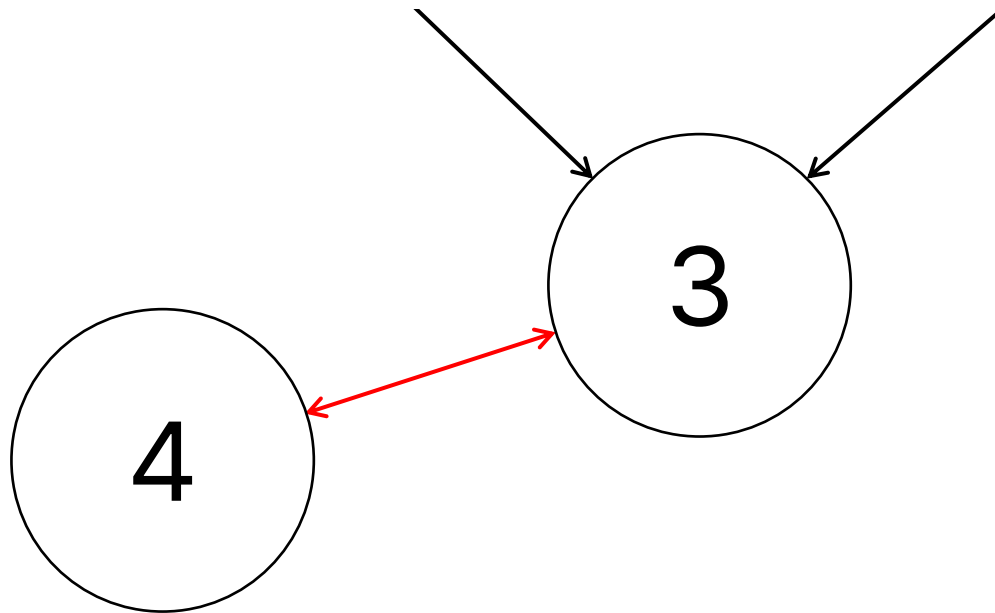


Directed Graph



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

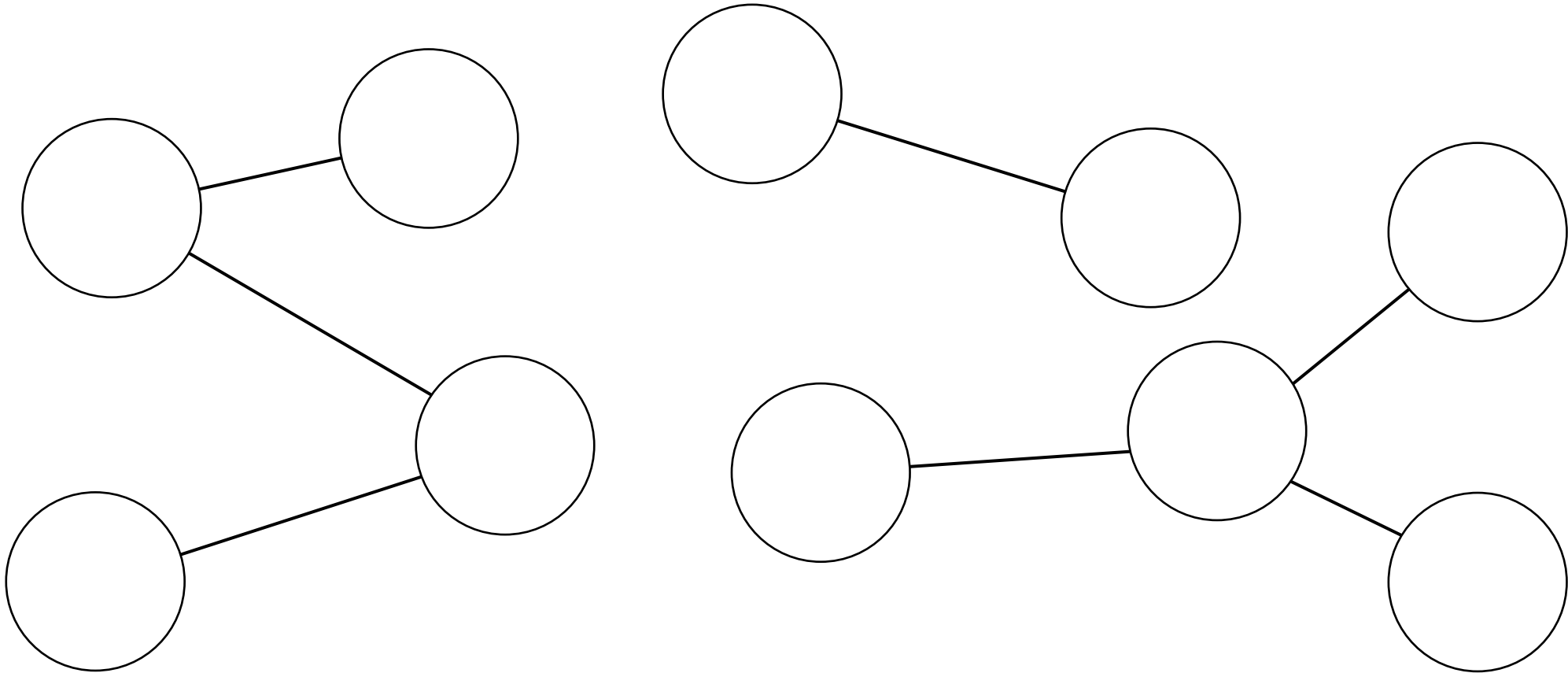
Directed, Undirected Graph

- 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

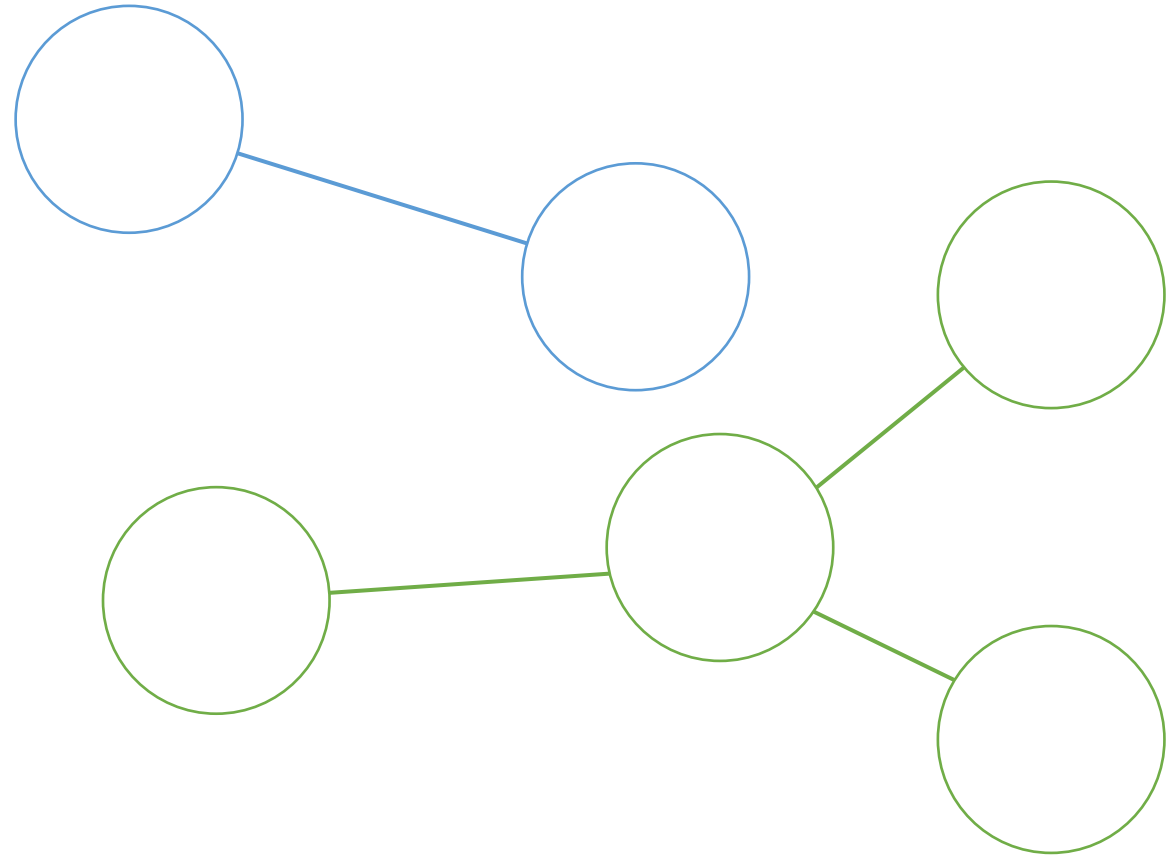
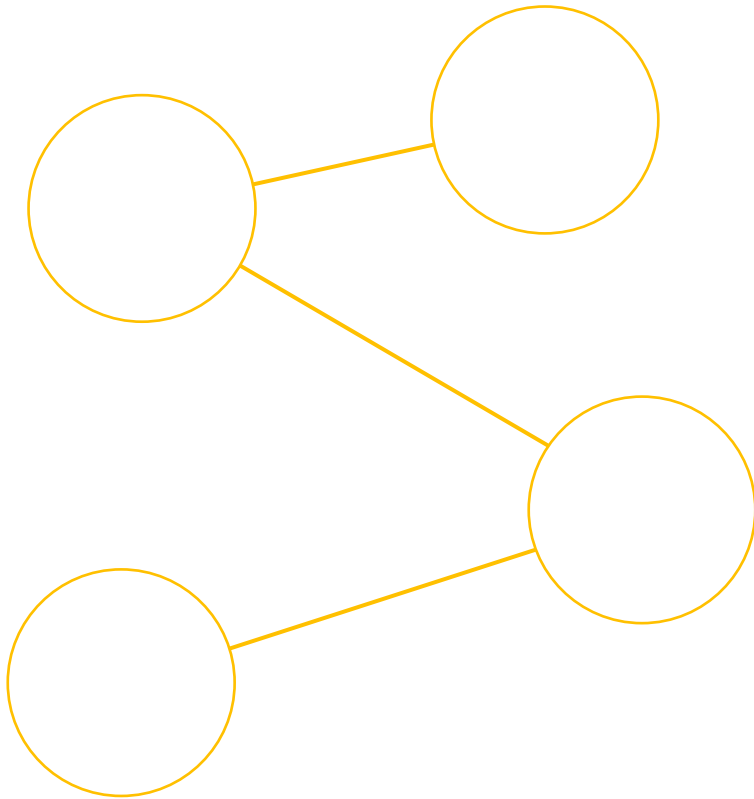
Connected Component

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

Connected Component

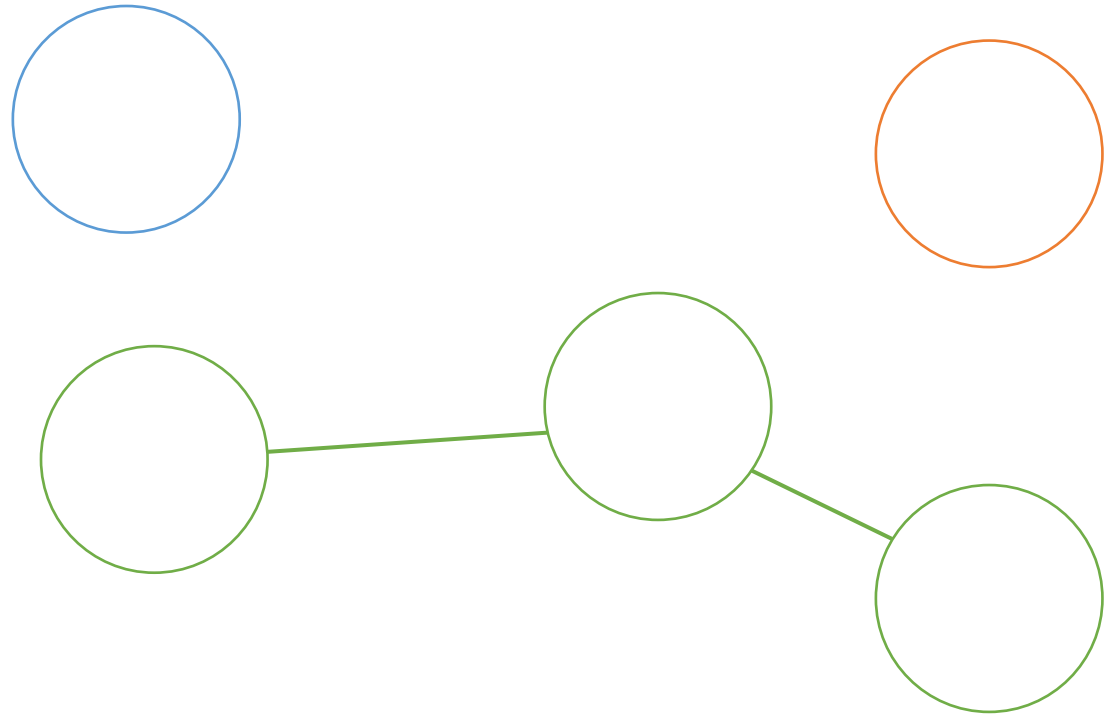
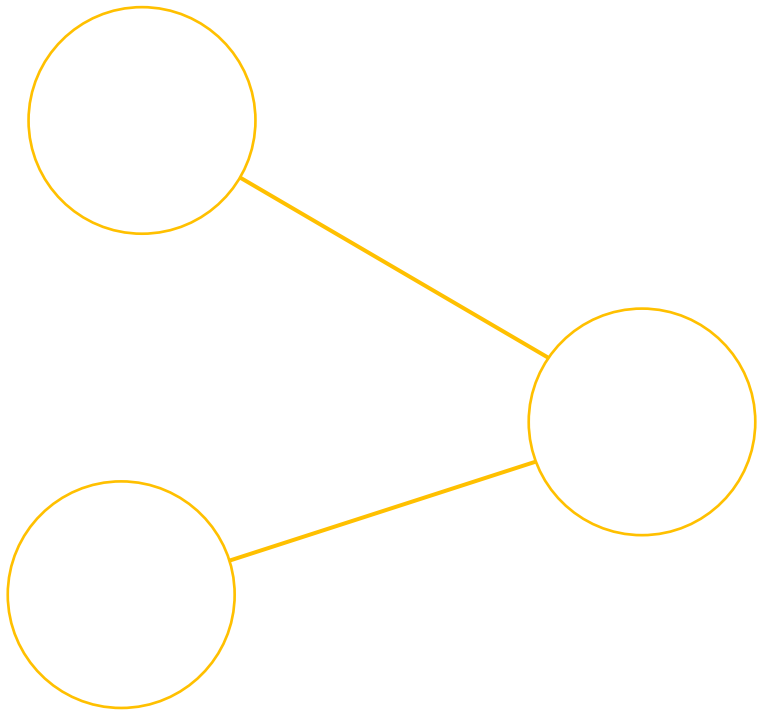


Connected Component



Connected Component

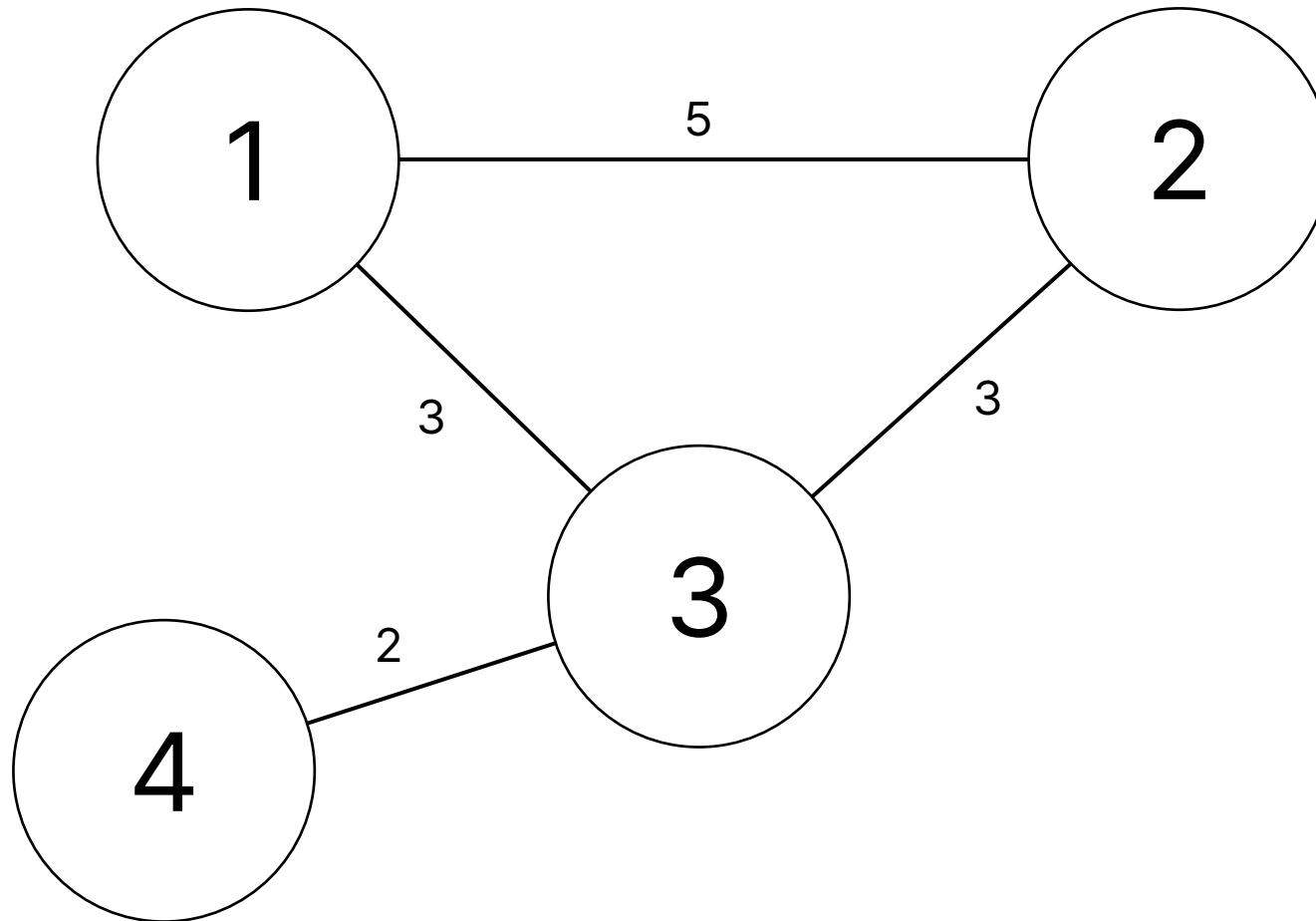
- 노드에 간선이 존재하지 않는 경우, 노드 하나가 하나의 연결 요소다



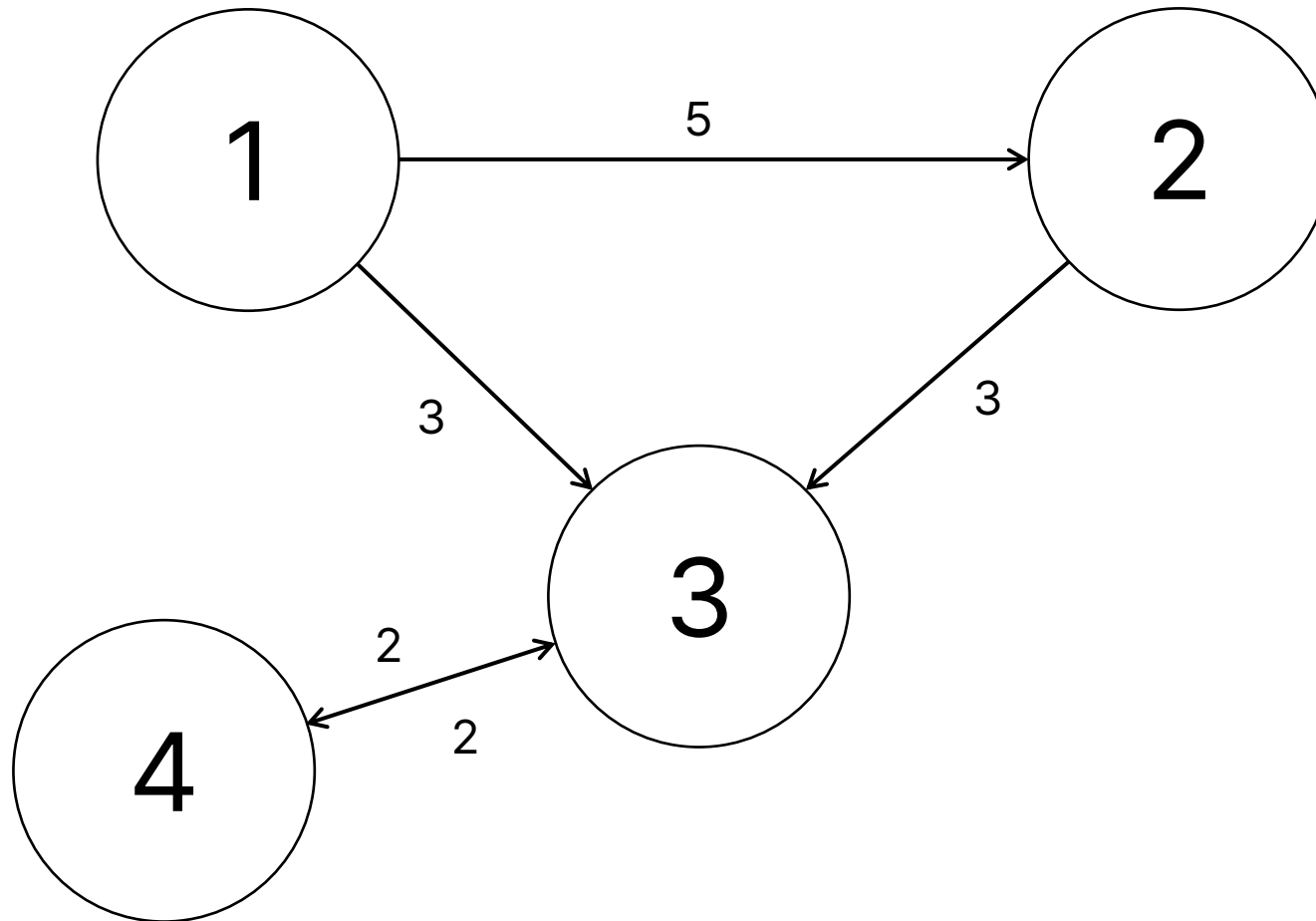
Weight

- 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



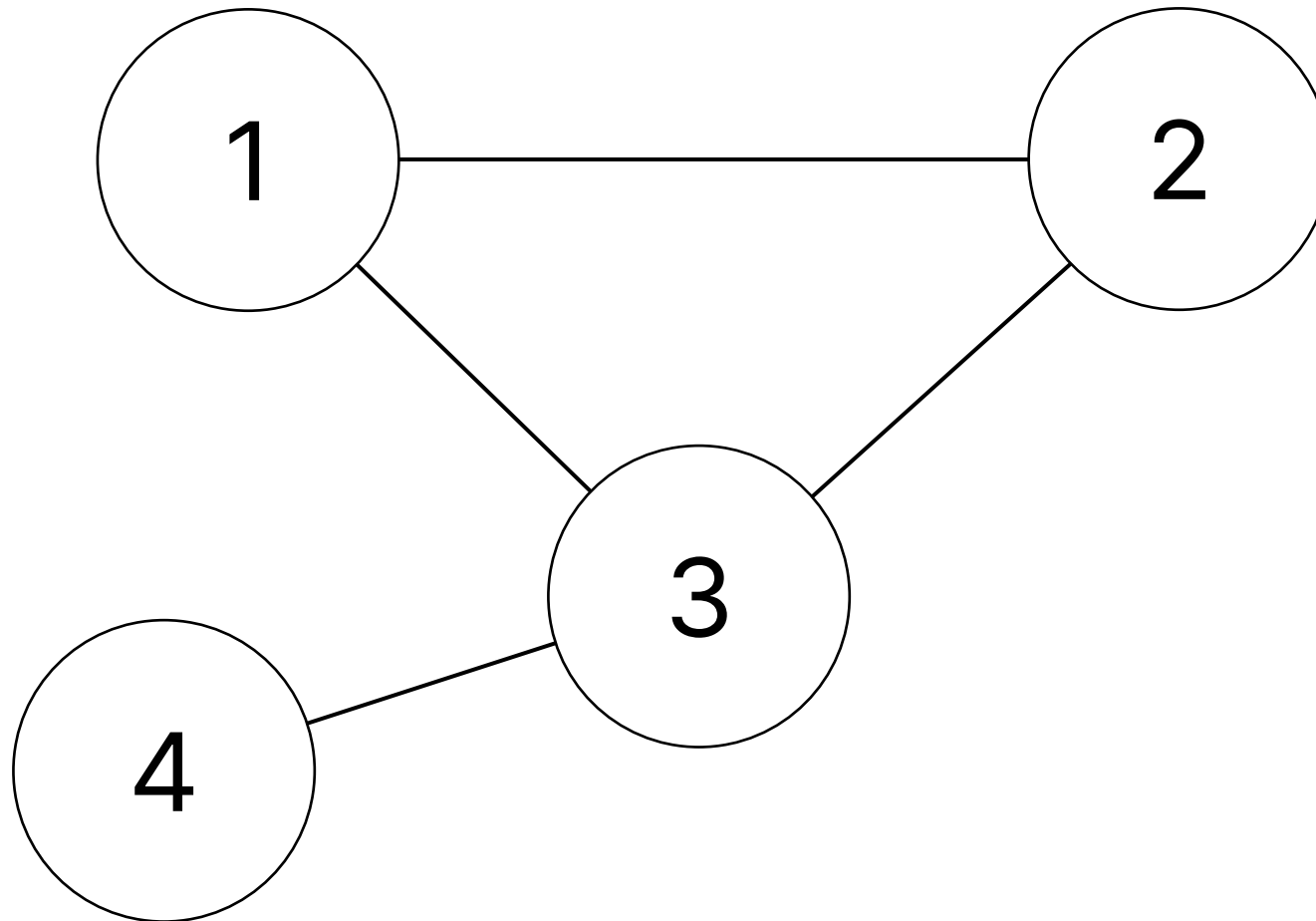
Weight



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

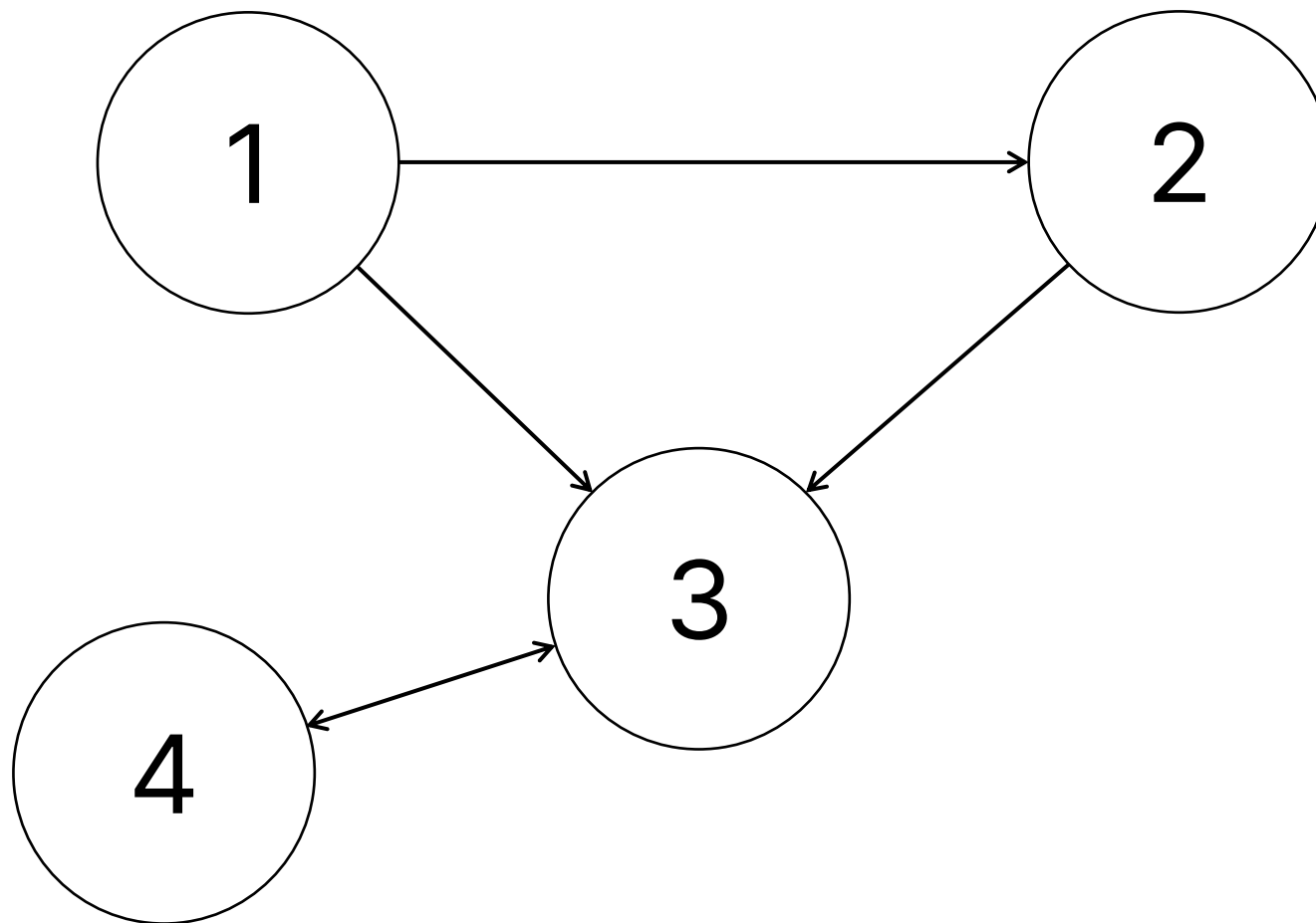
Degree



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

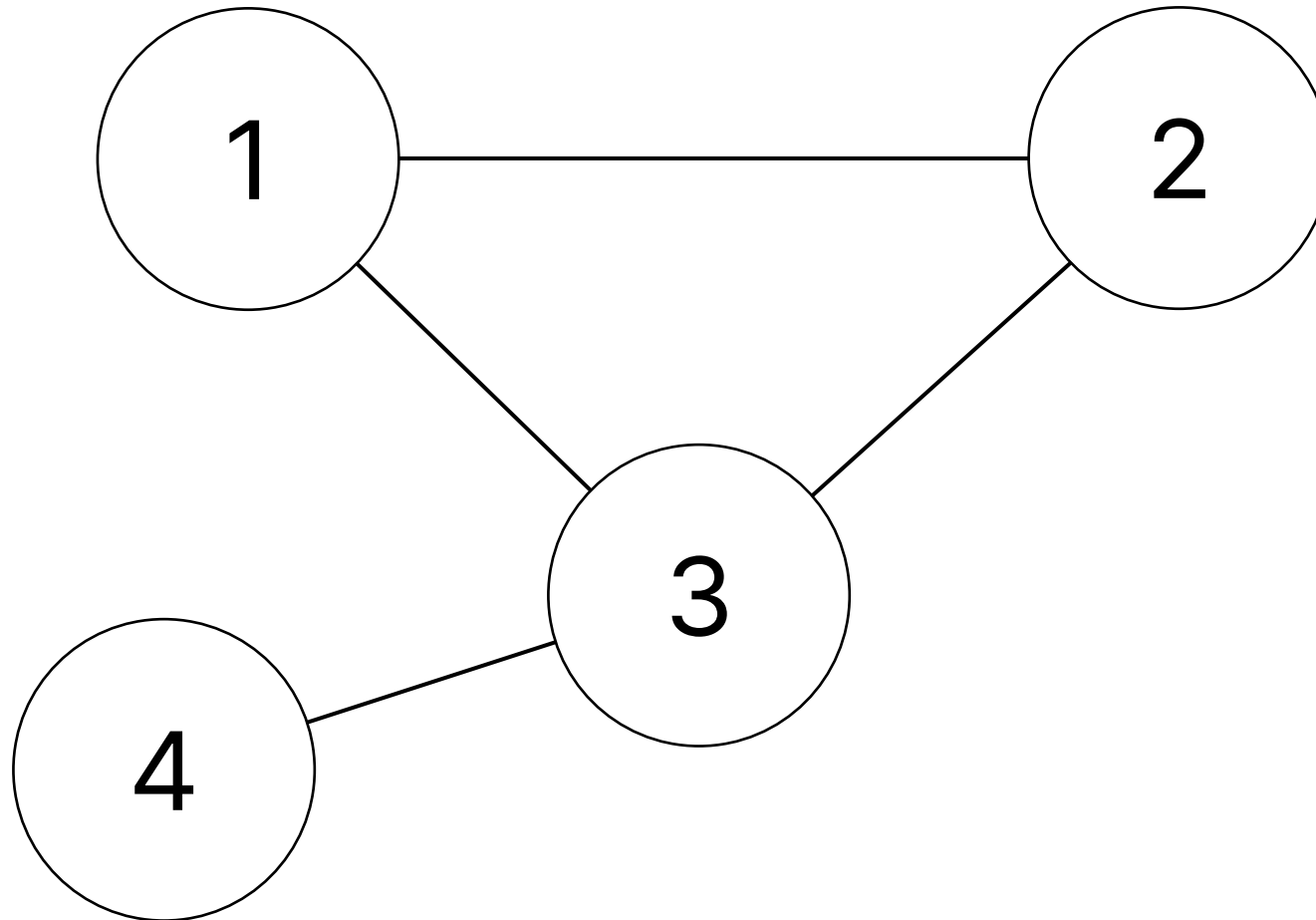
그래프의 표현 방법

- 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

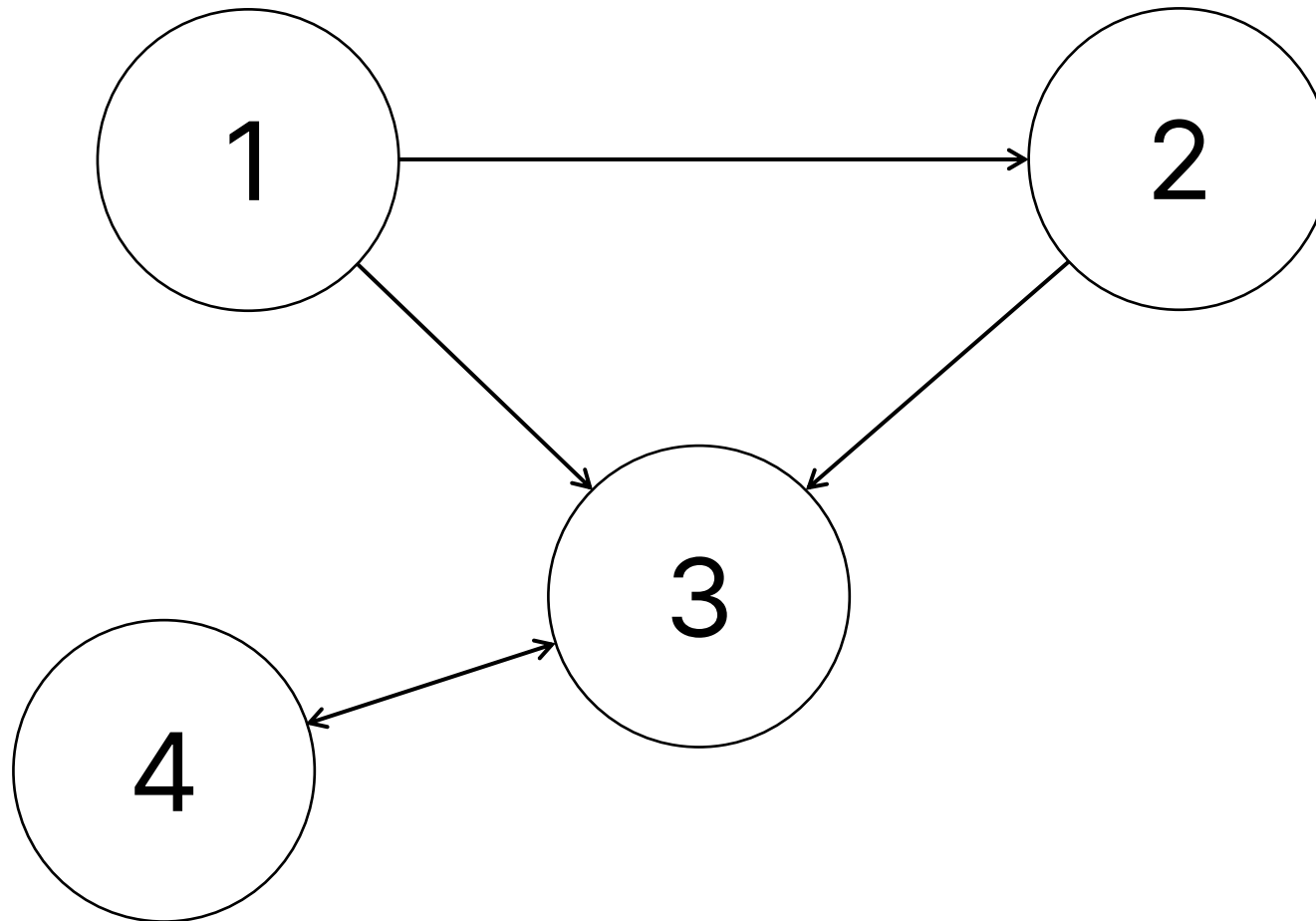
1	2	3	
2	1	3	
3	1	2	4
4	3		

Adjacency List



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

Adjacency List

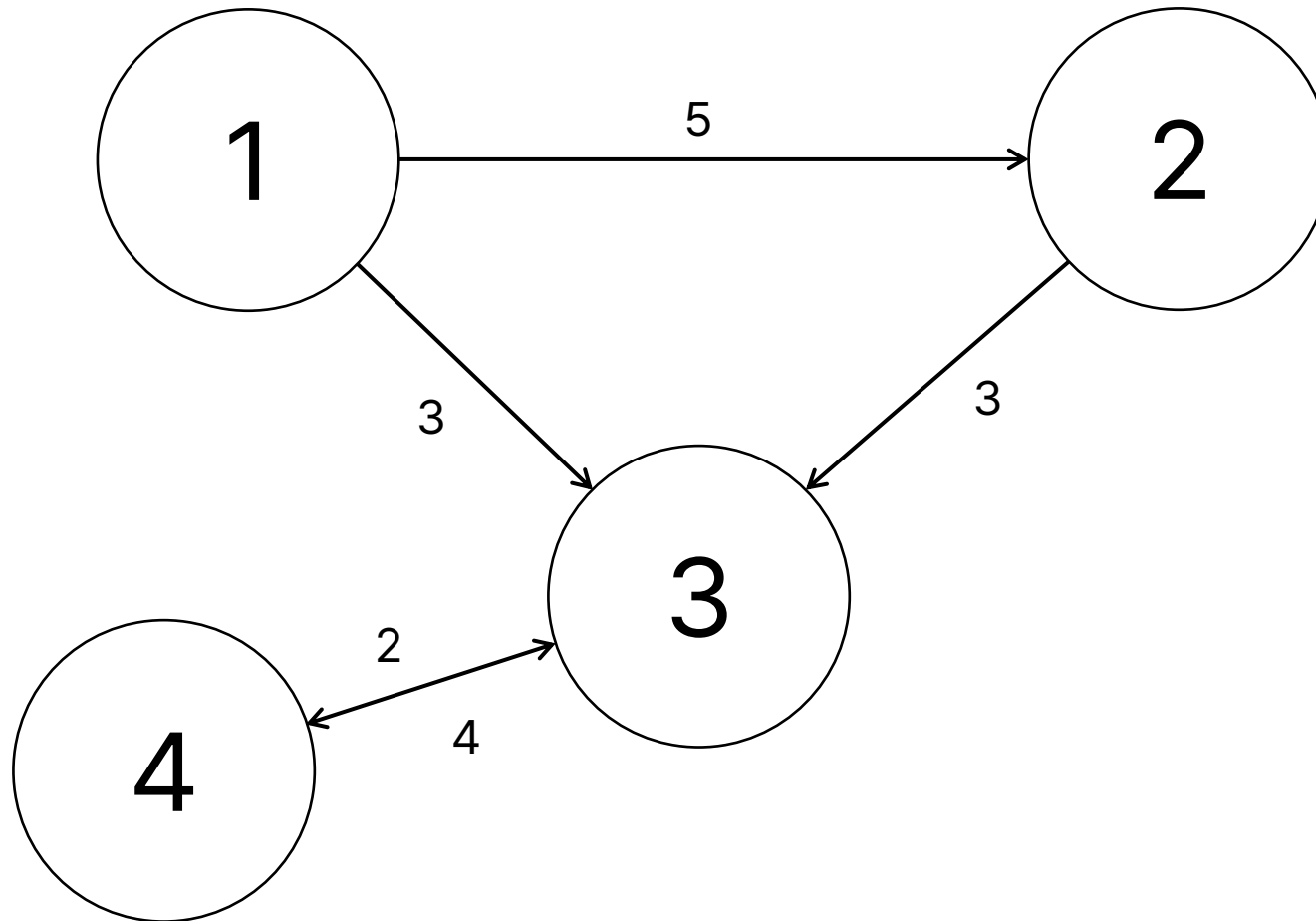
1	2	3
2	3	
3	4	
4	3	

Adjacency List



```
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

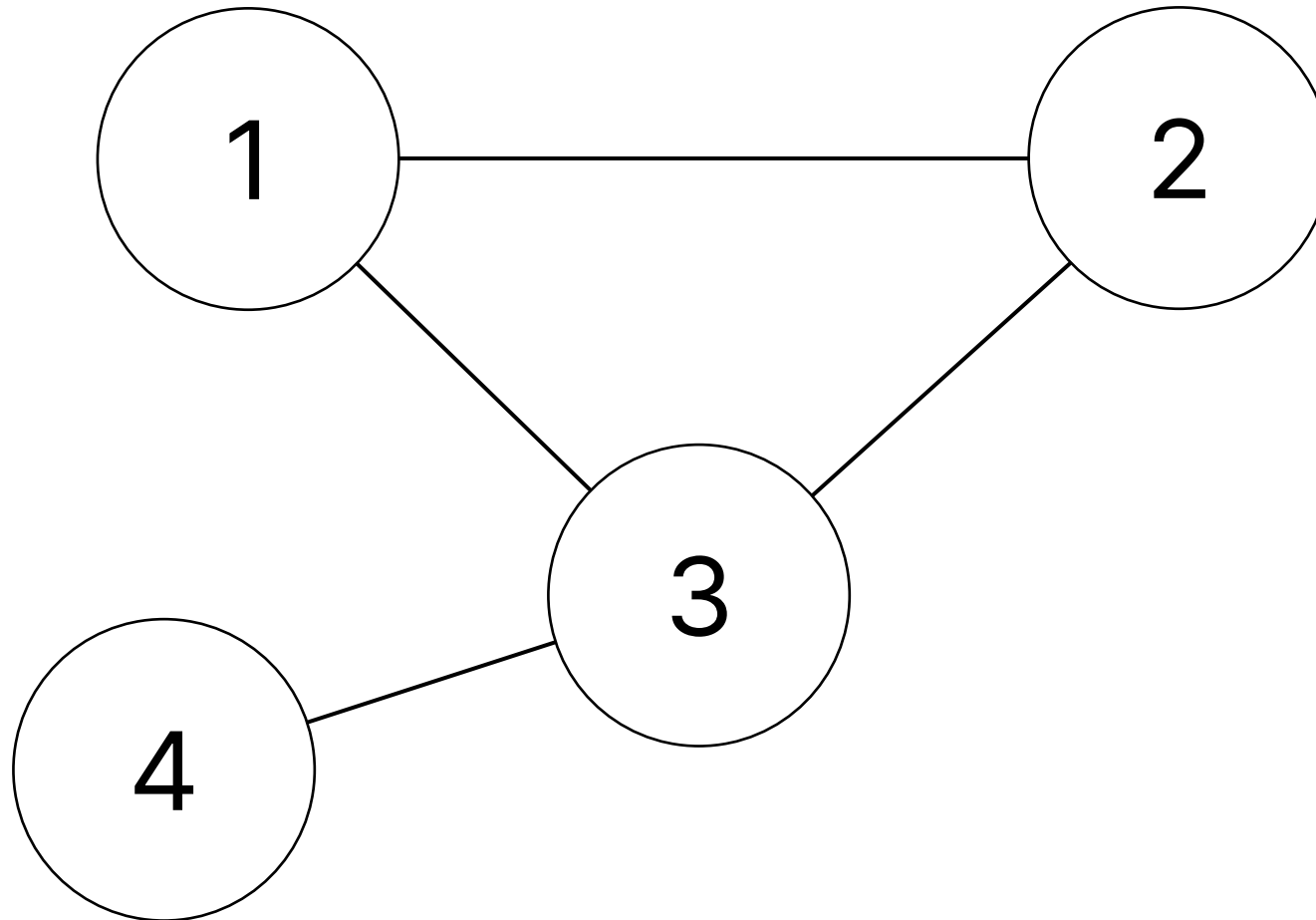


```
vector<pair<int, int>> edges[MAX_NODE];  
int s, e, w;  
  
for(int i = 0; i < MAX_EDGE; i++) {  
    cin >> s >> e >> w;  
    edges[s].push_back({e, w});  
}
```

Adjacency Matrix

- 행렬로 노드의 연결 유무를 저장한다
- $arr[i][j]$ 는 i 번 노드에서 출발해 j 번 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

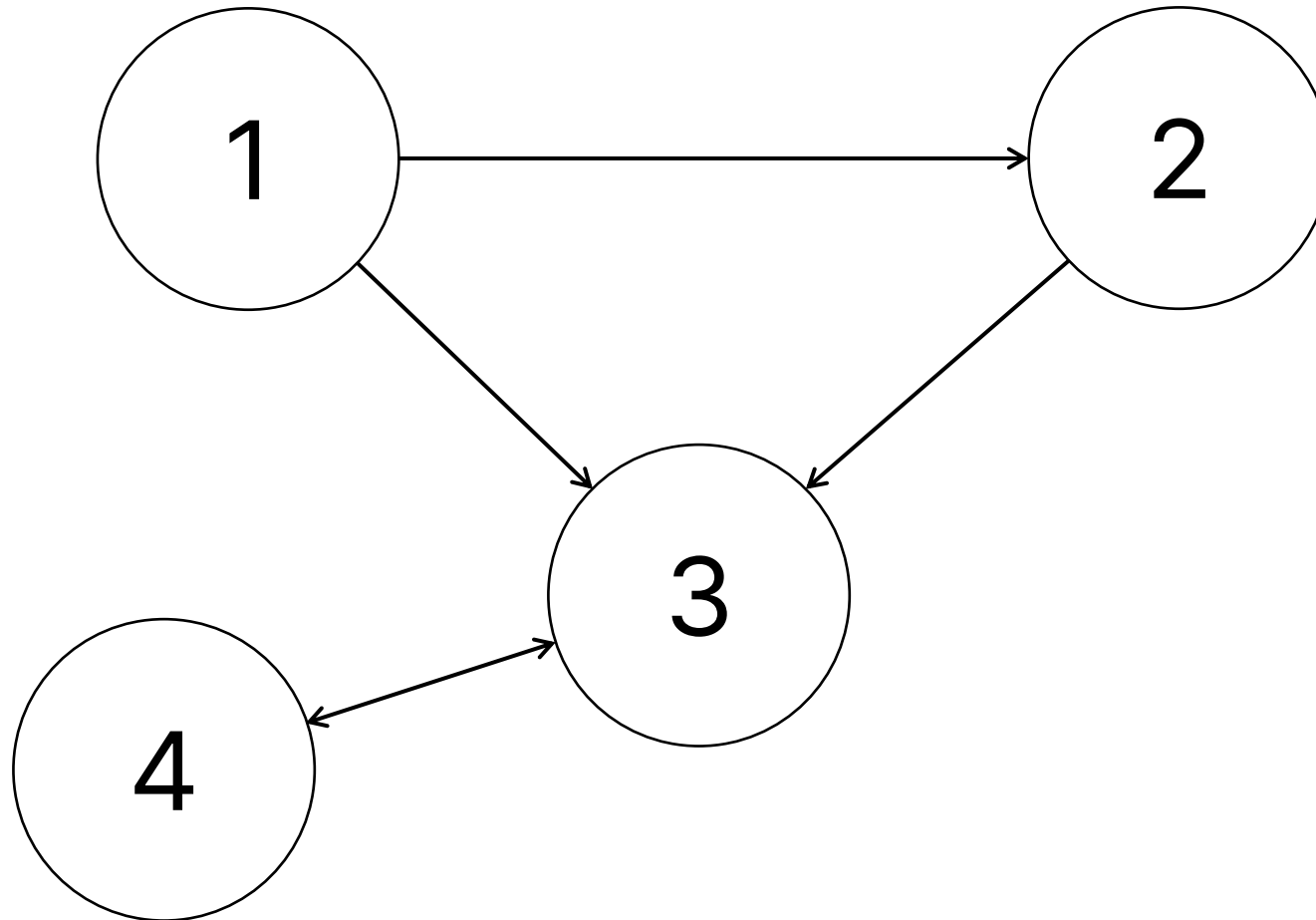
	1	2	3	4
1	F	T	T	F
2	T	F	T	F
3	T	T	F	T
4	F	F	T	F

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

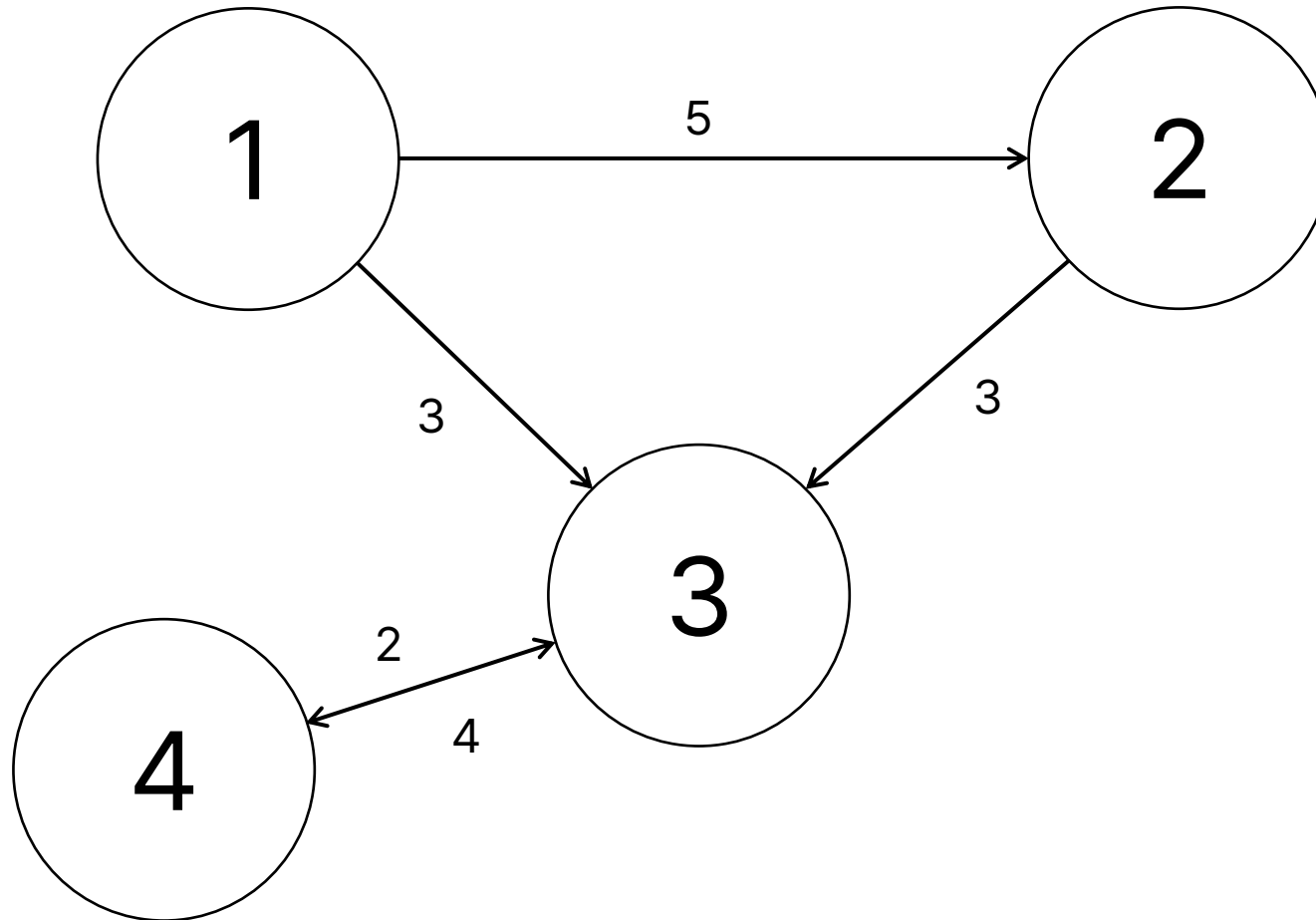
	1	2	3	4
1	F	T	T	F
2	F	F	T	F
3	F	F	F	T
4	F	F	T	F

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;
}
```

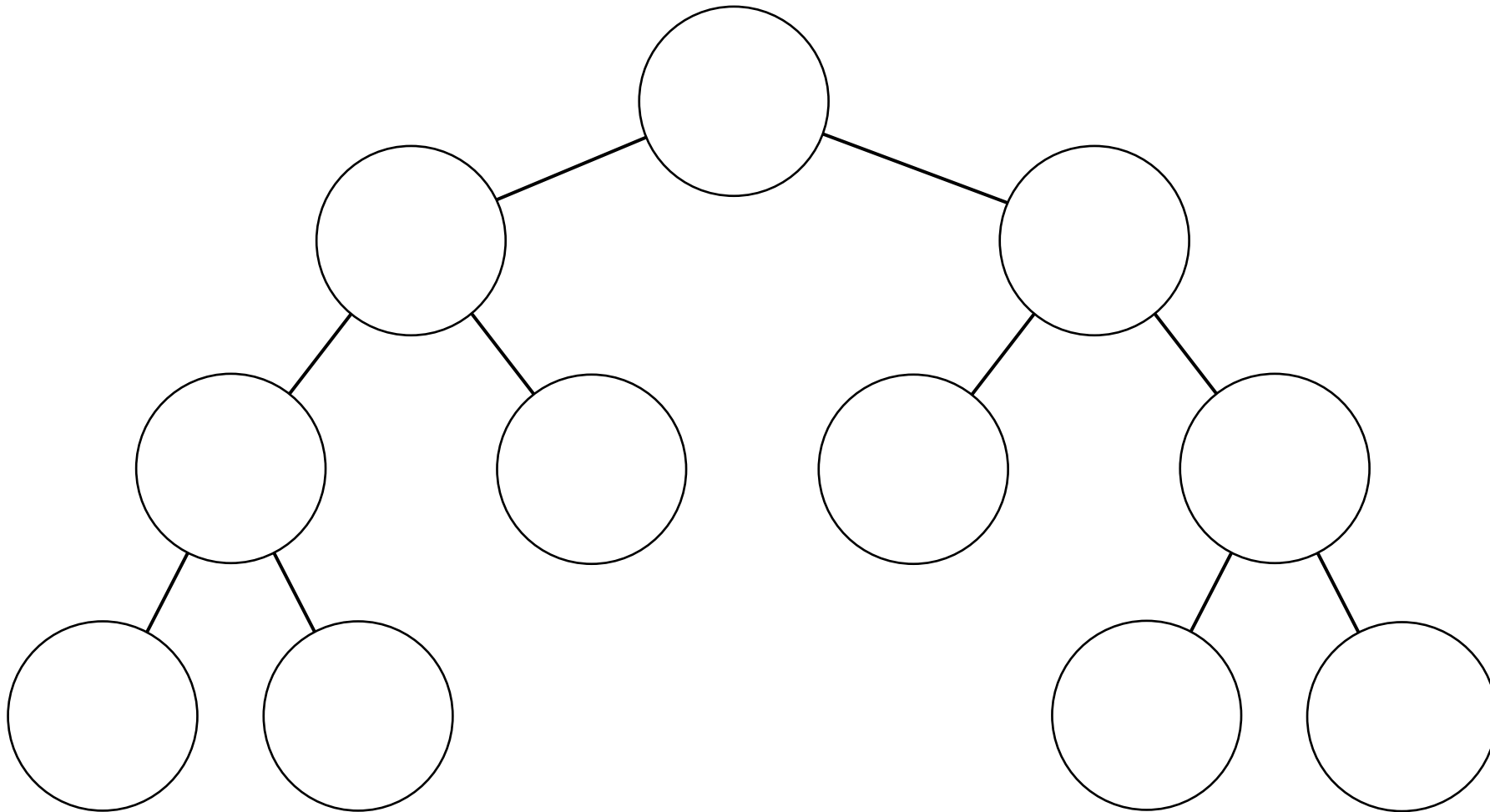
Traversal

- 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

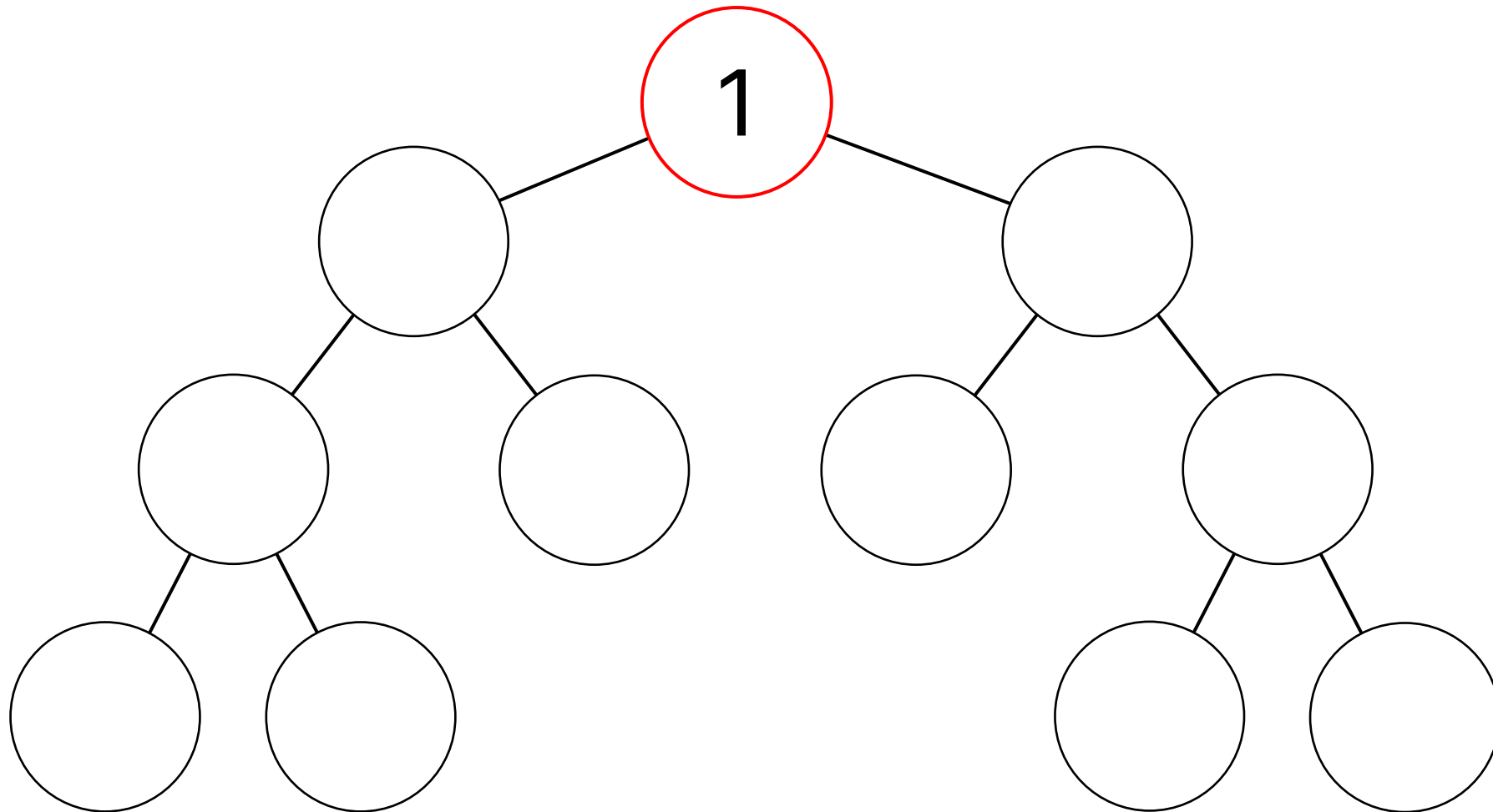
DFS

- 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

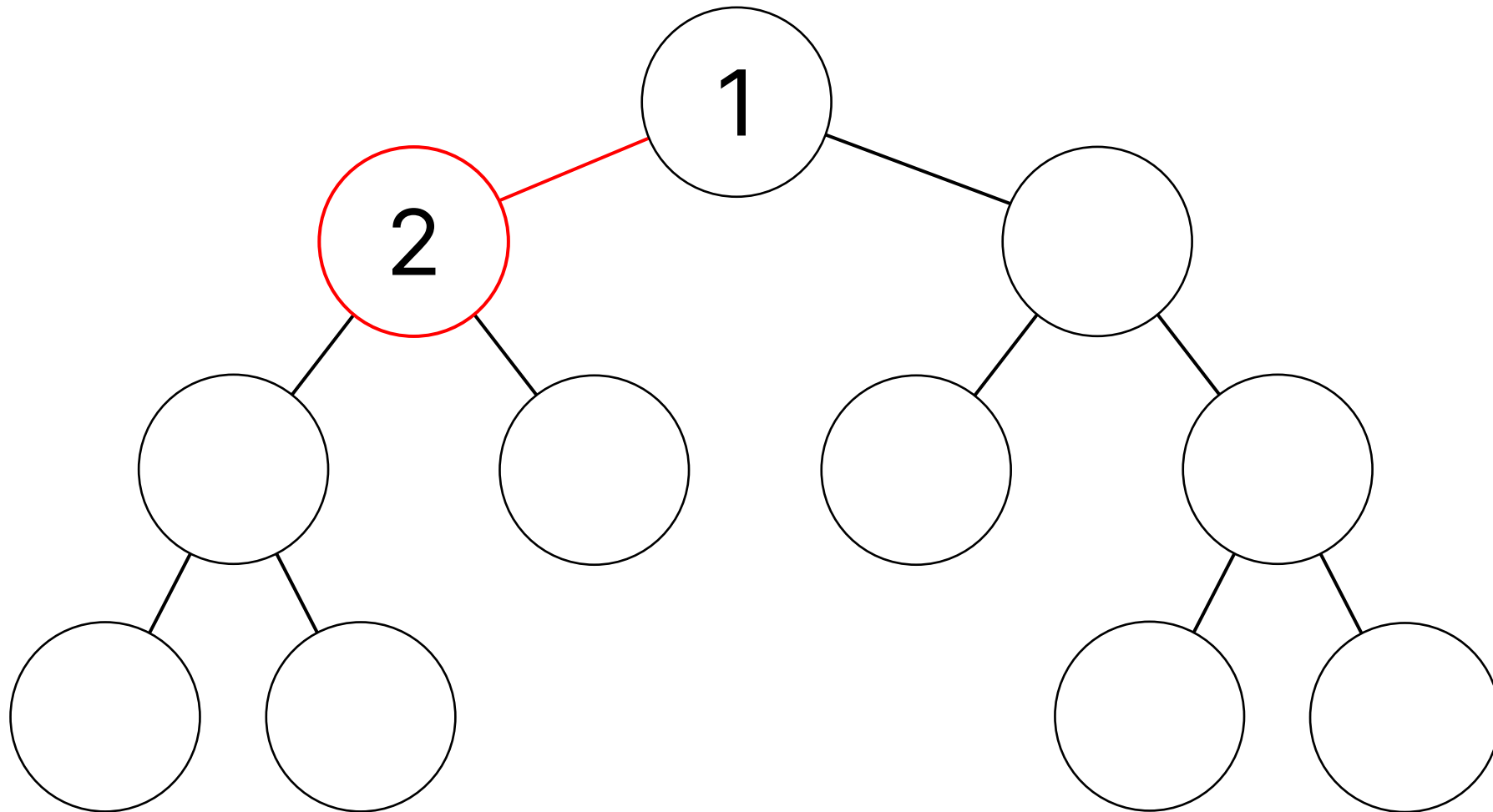
DFS



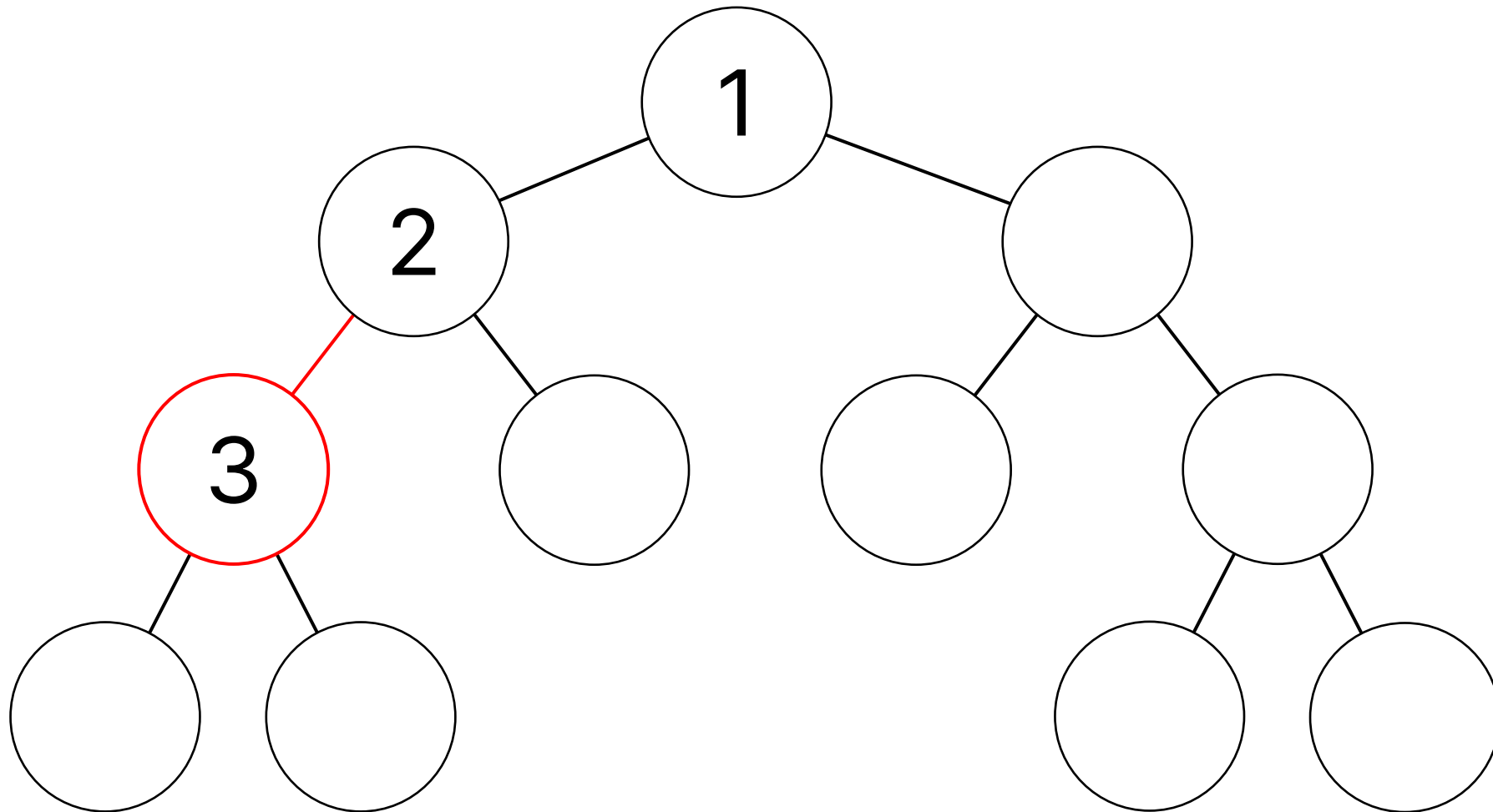
DFS



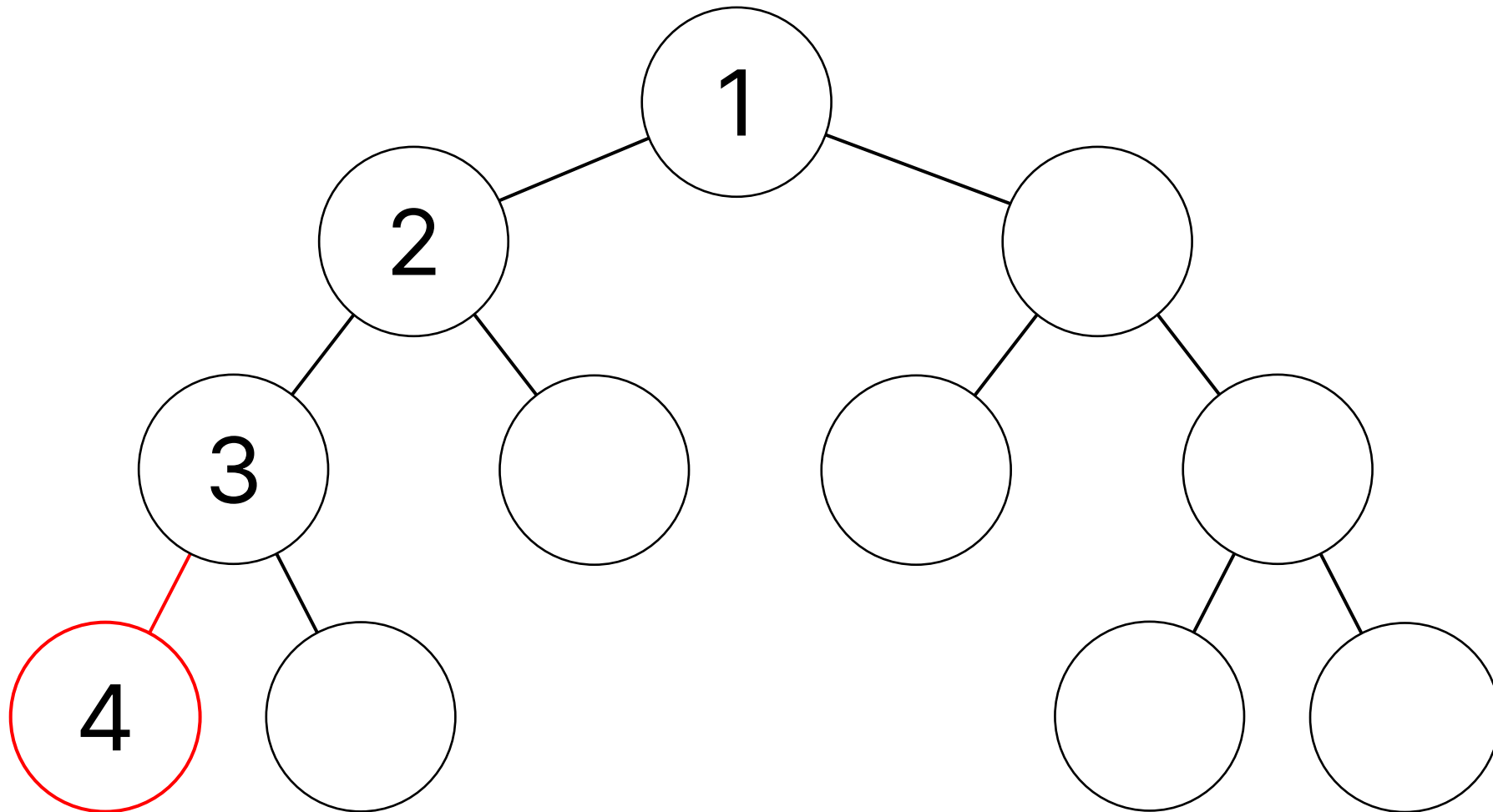
DFS



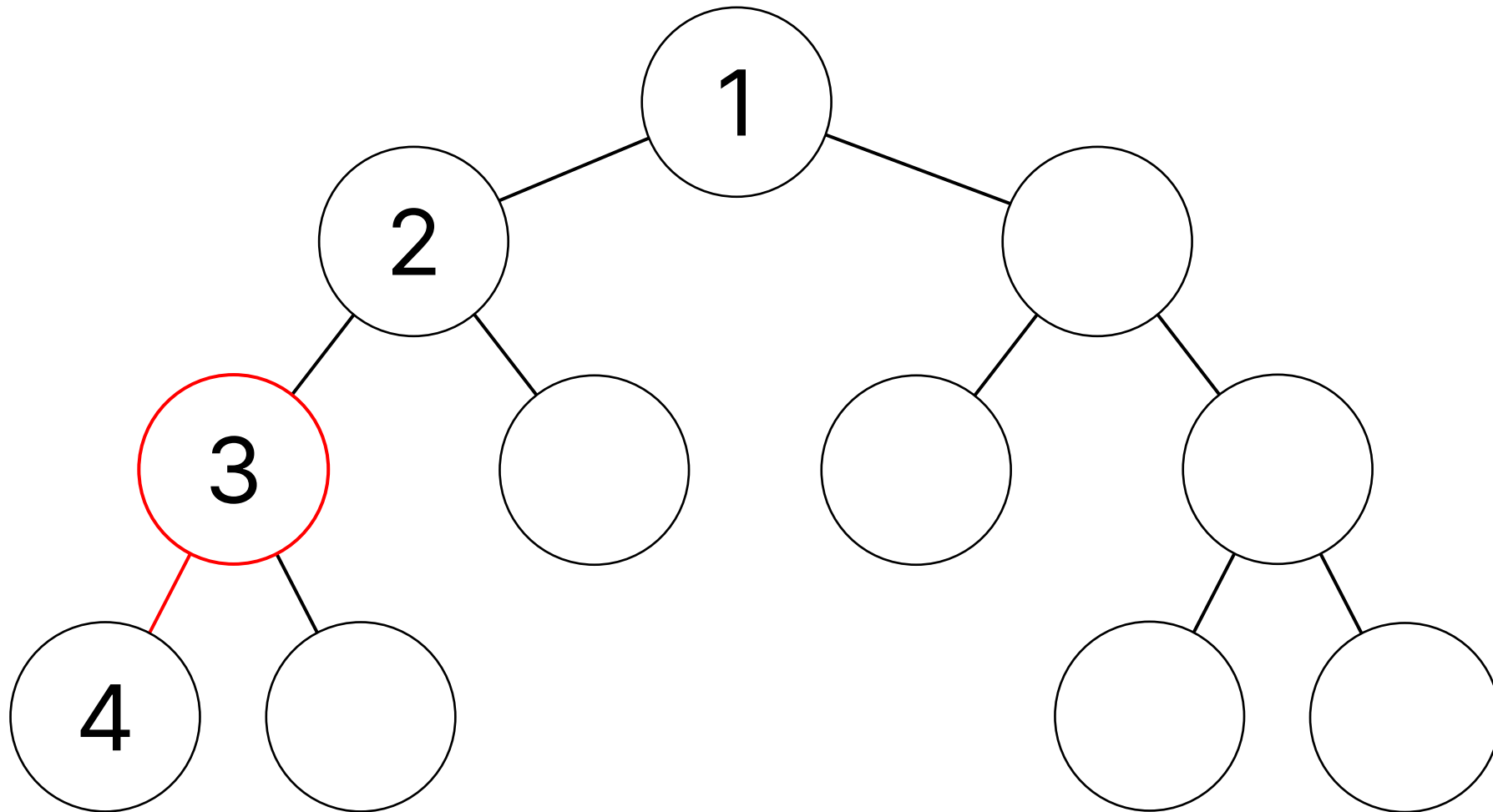
DFS



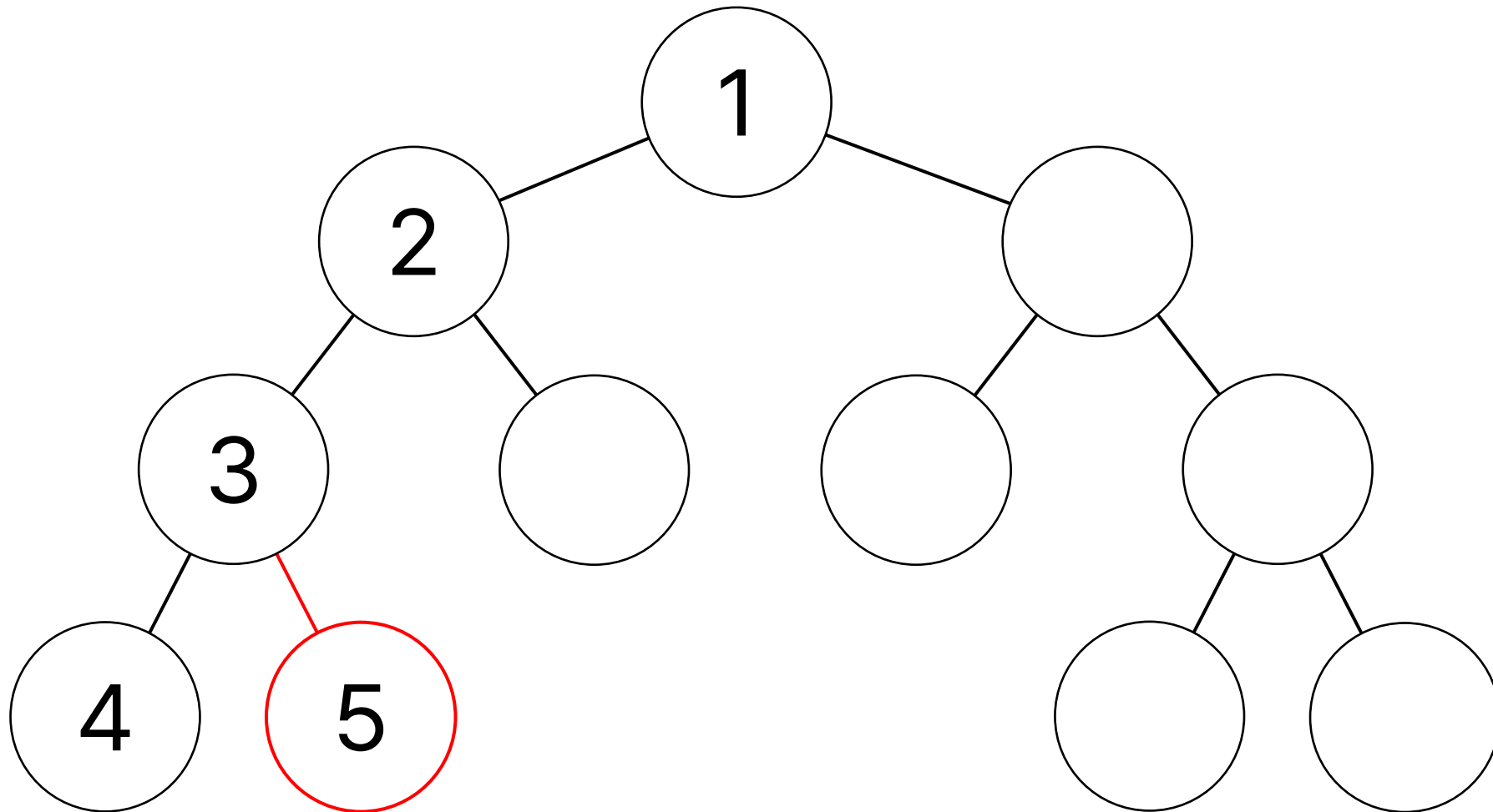
DFS



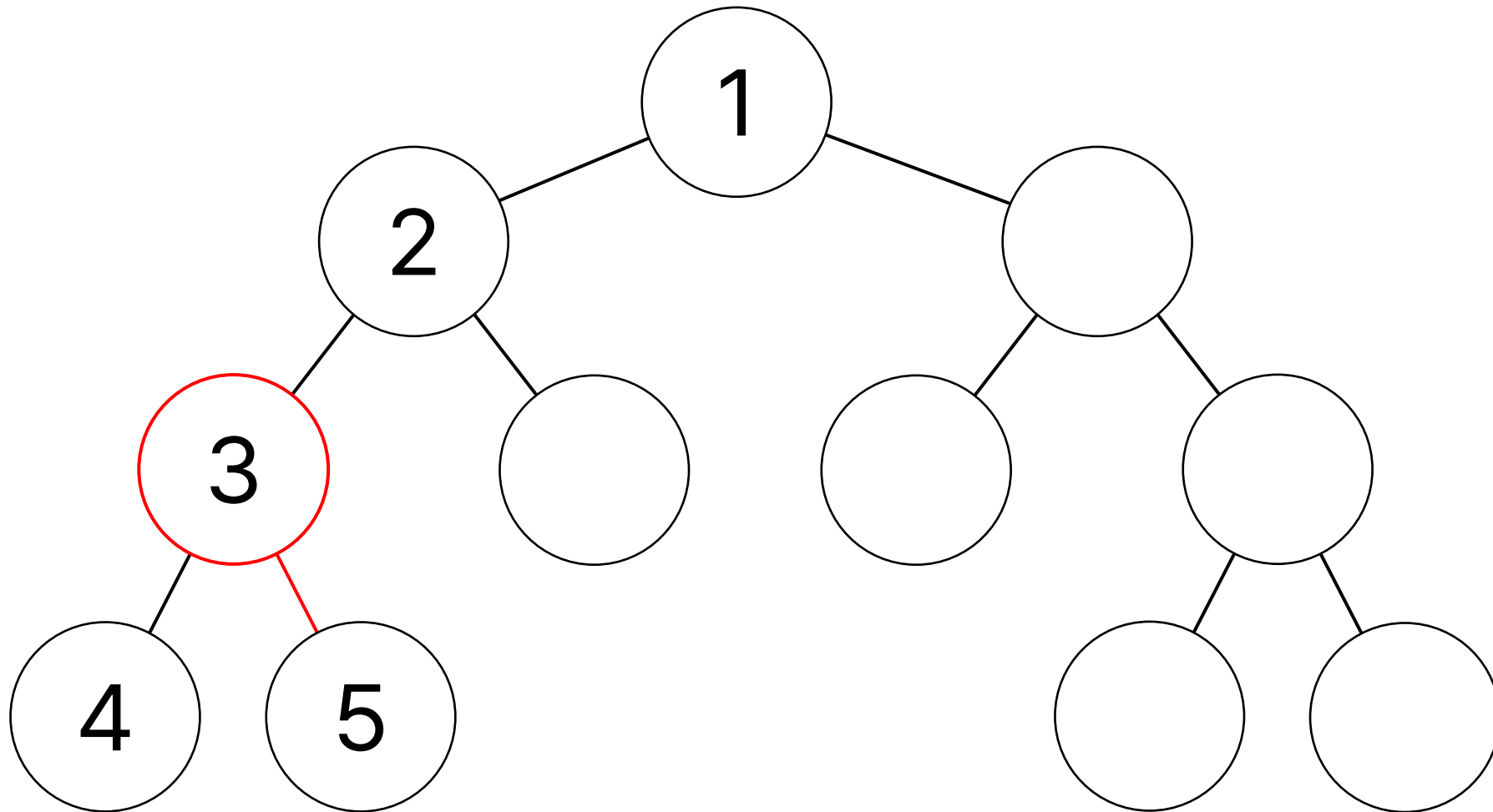
DFS



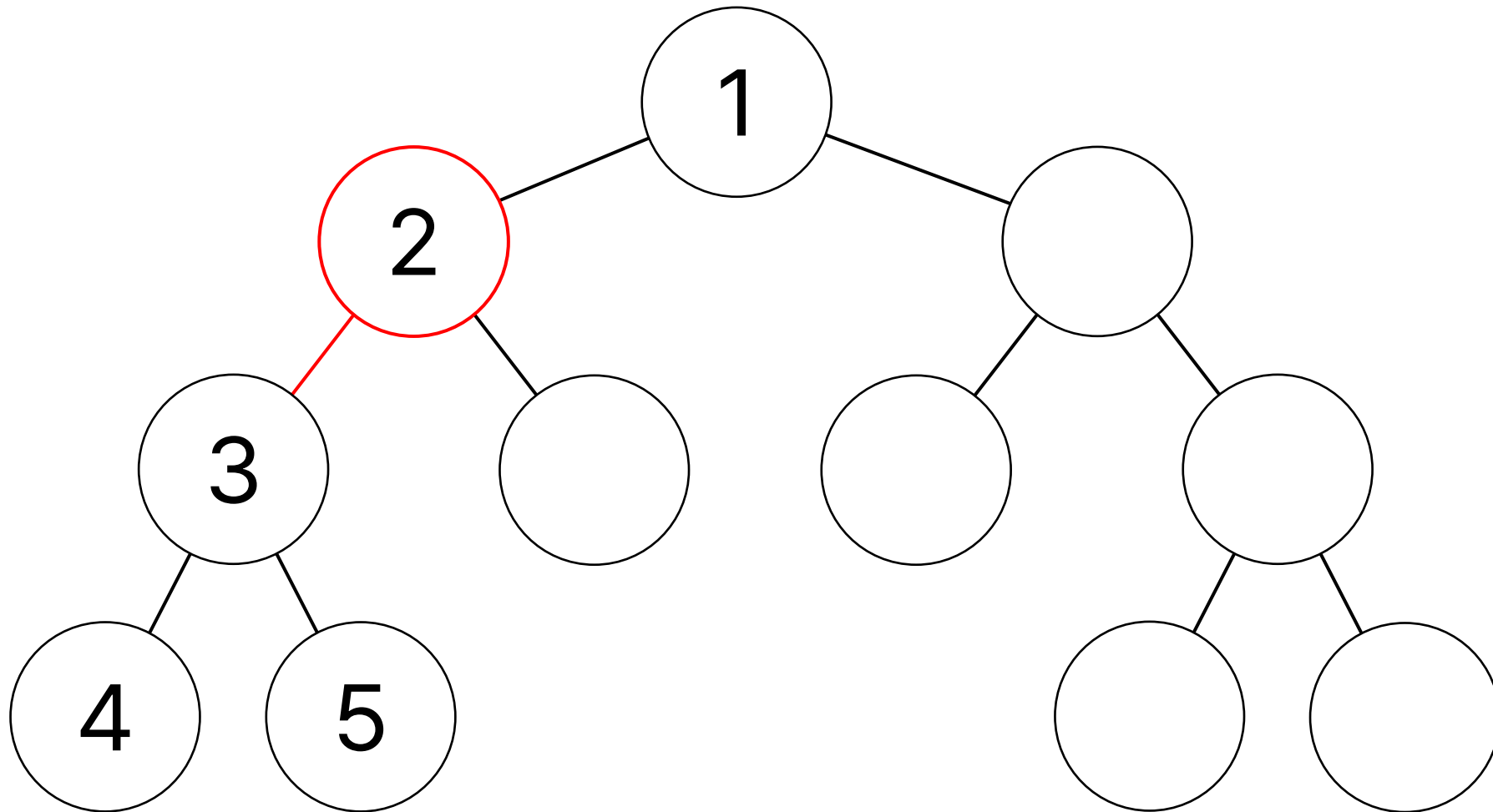
DFS



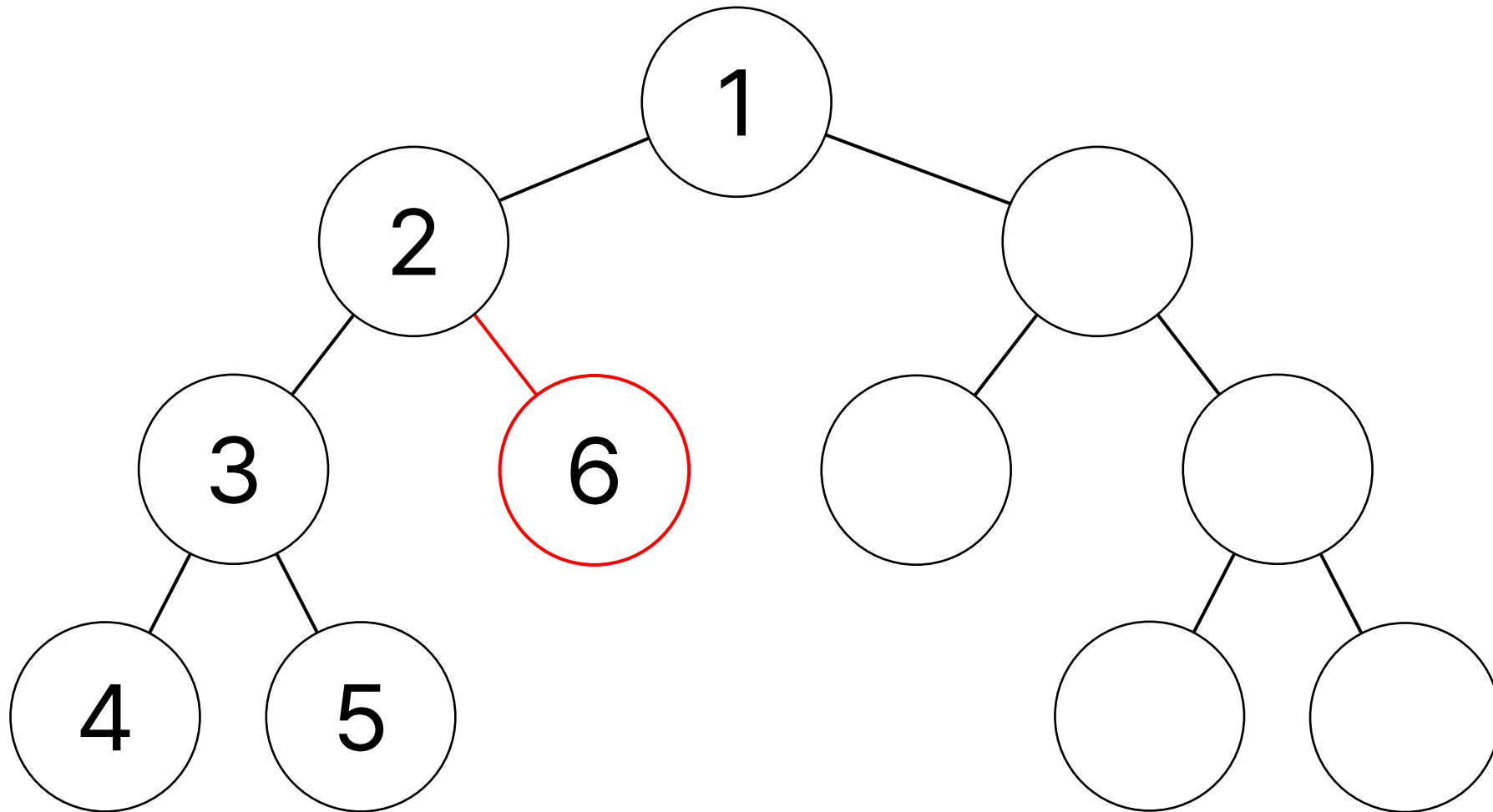
DFS



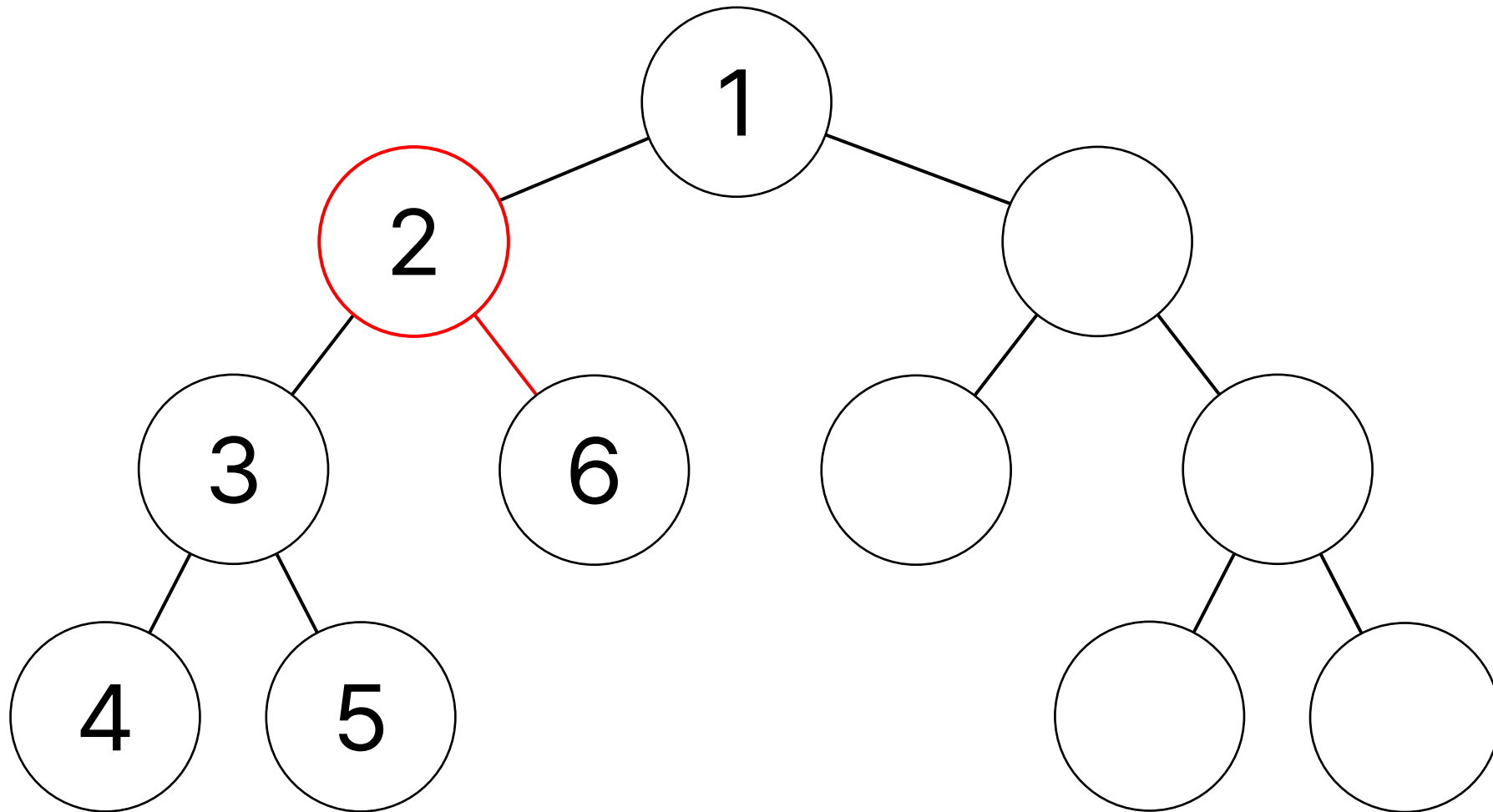
DFS



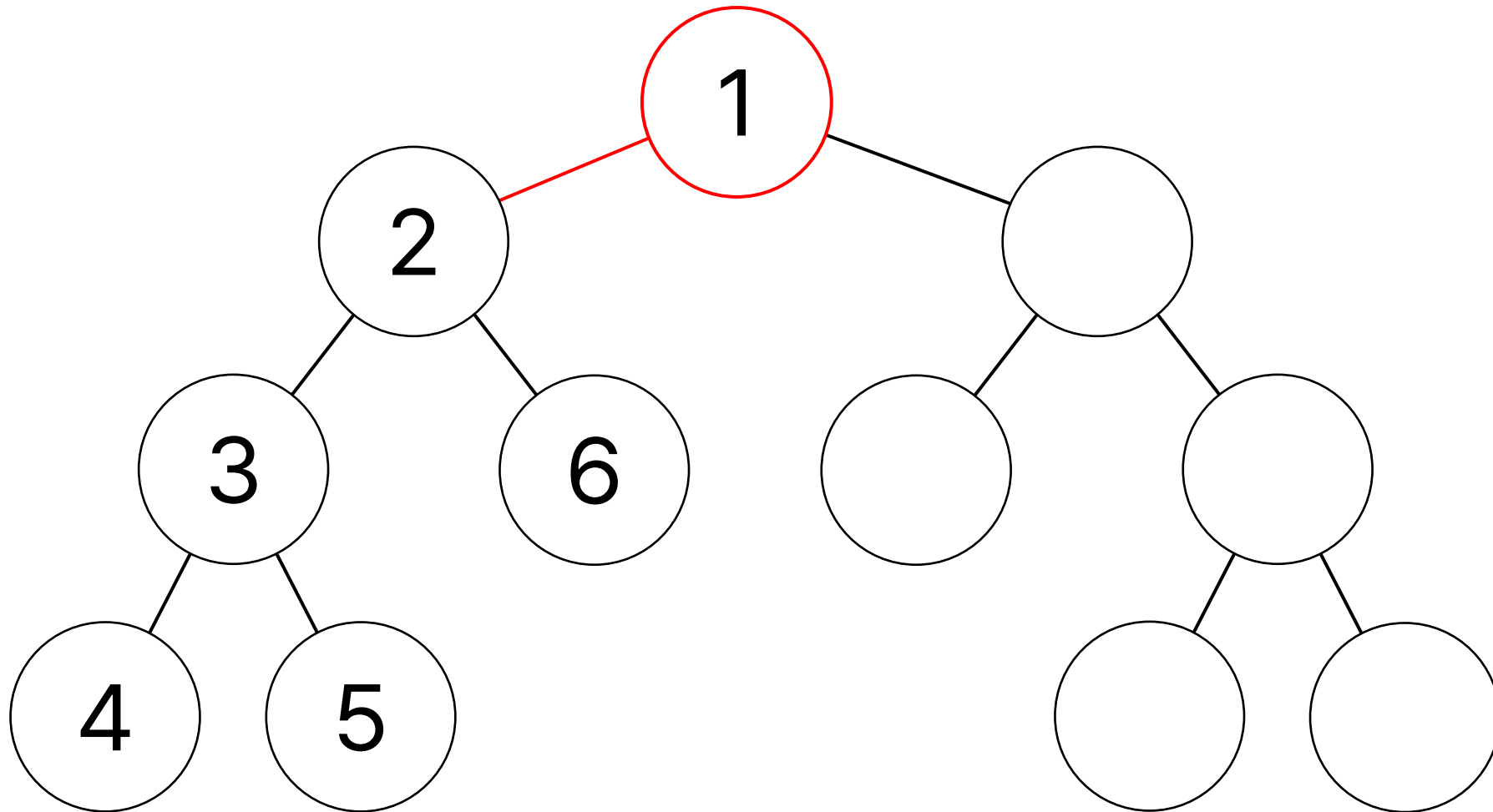
DFS



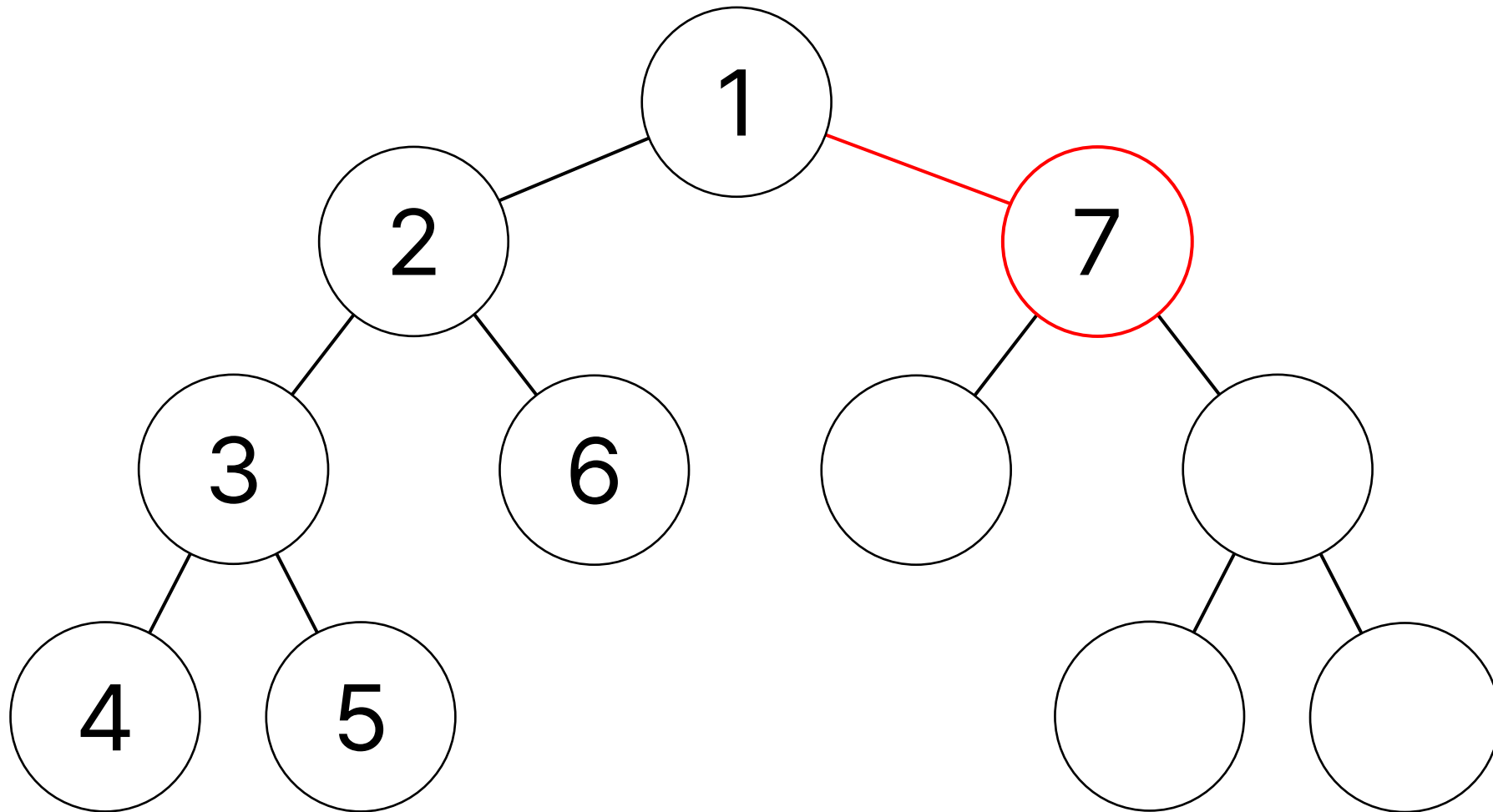
DFS



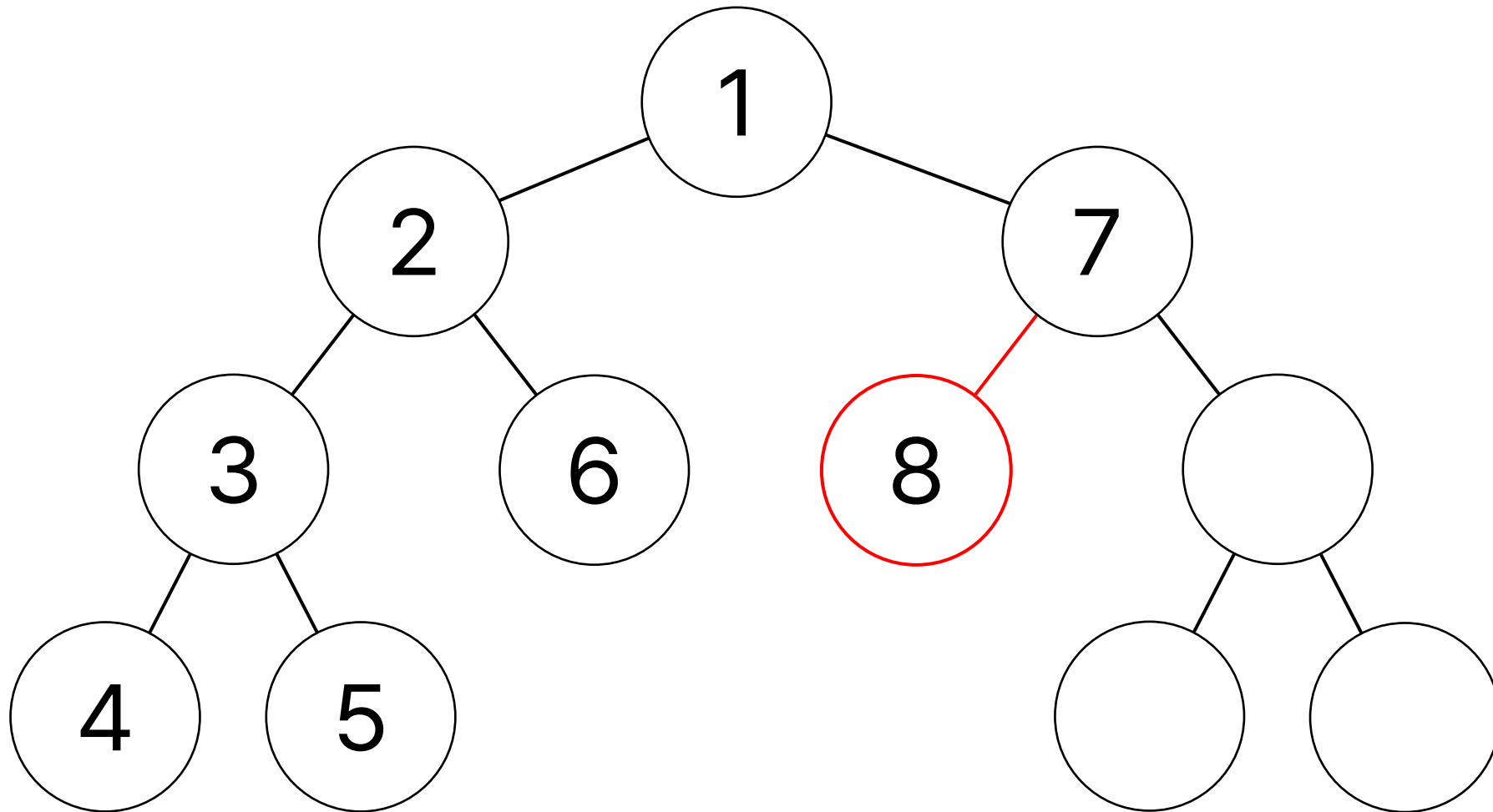
DFS



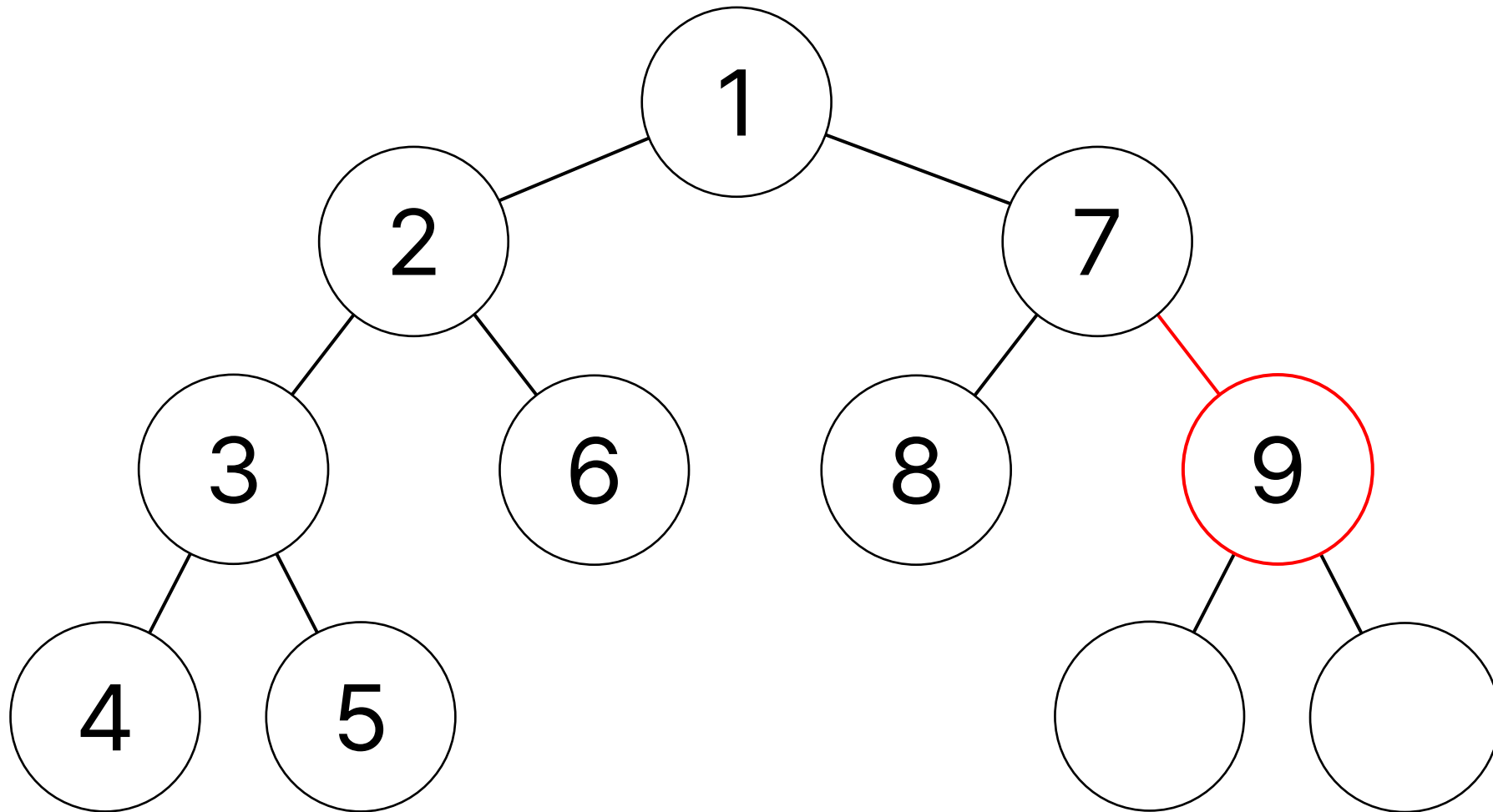
DFS



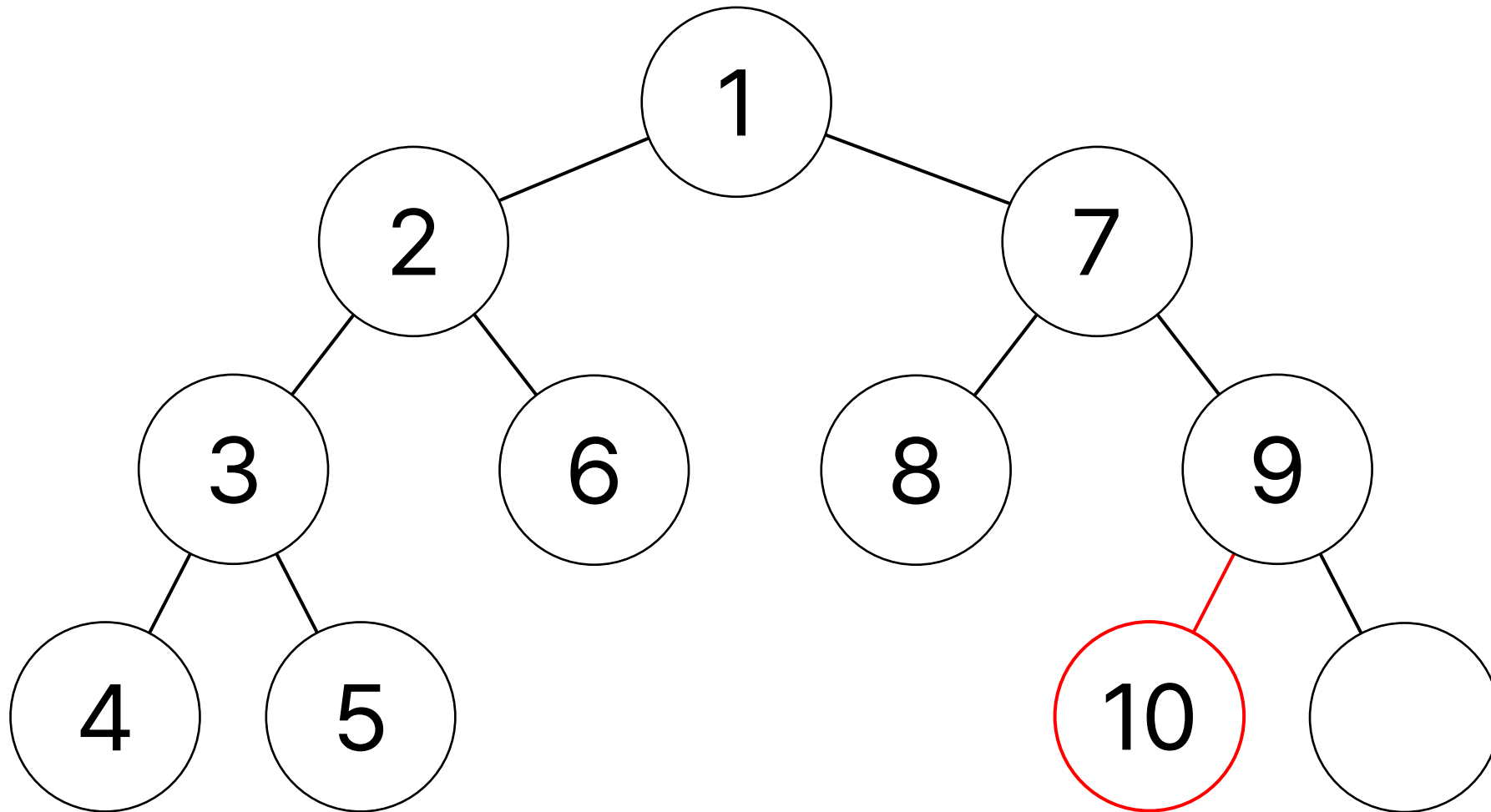
DFS



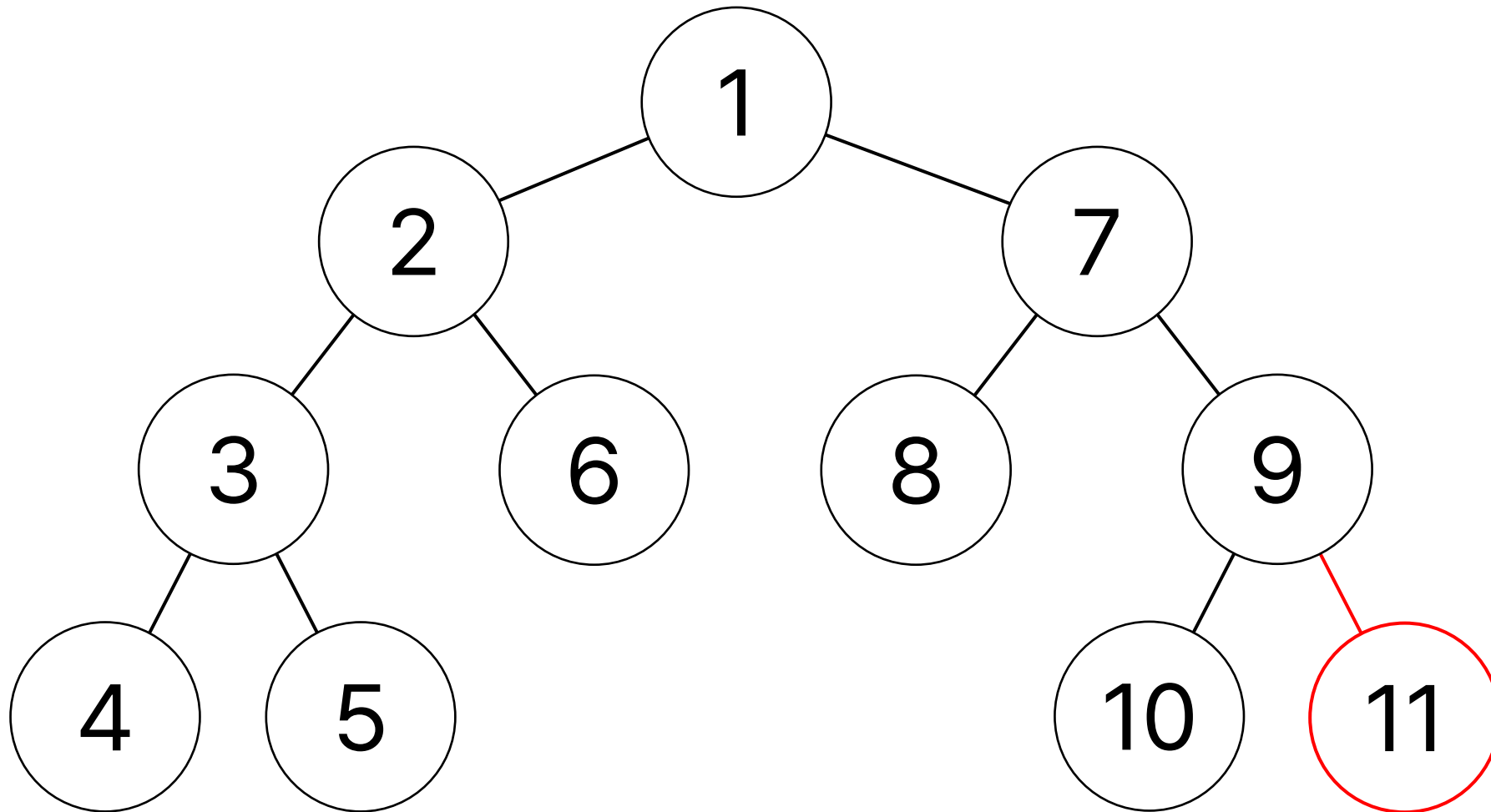
DFS



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



```
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

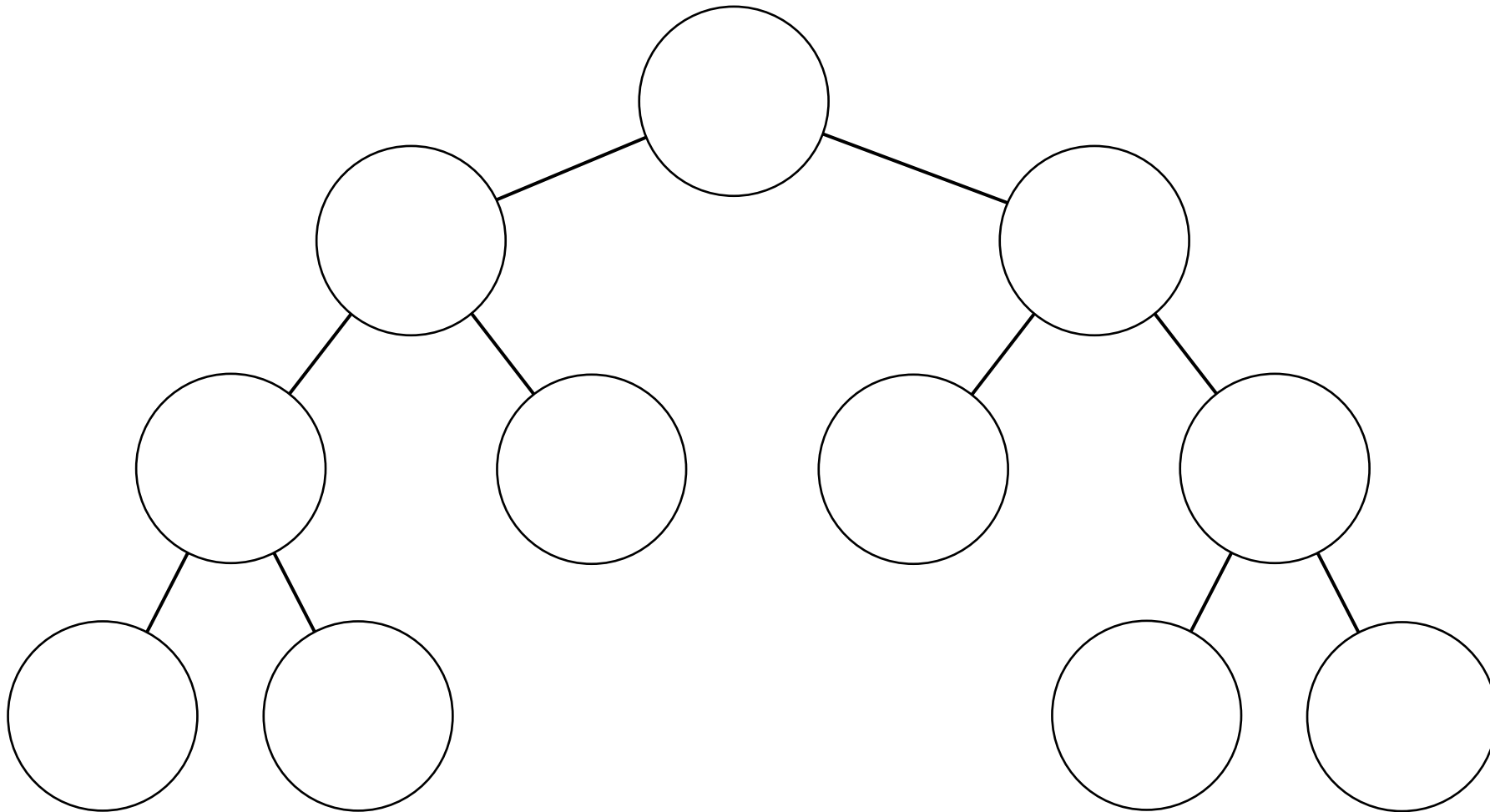


```
vector<int> edges[MAX_NODE];  
bool visited[MAX_NODE];  
  
void dfs(int cur) {  
    if (visited[cur])  
        return;  
    visited[cur] = true;  
    for (auto next : edges[cur])  
        dfs(next);  
}
```

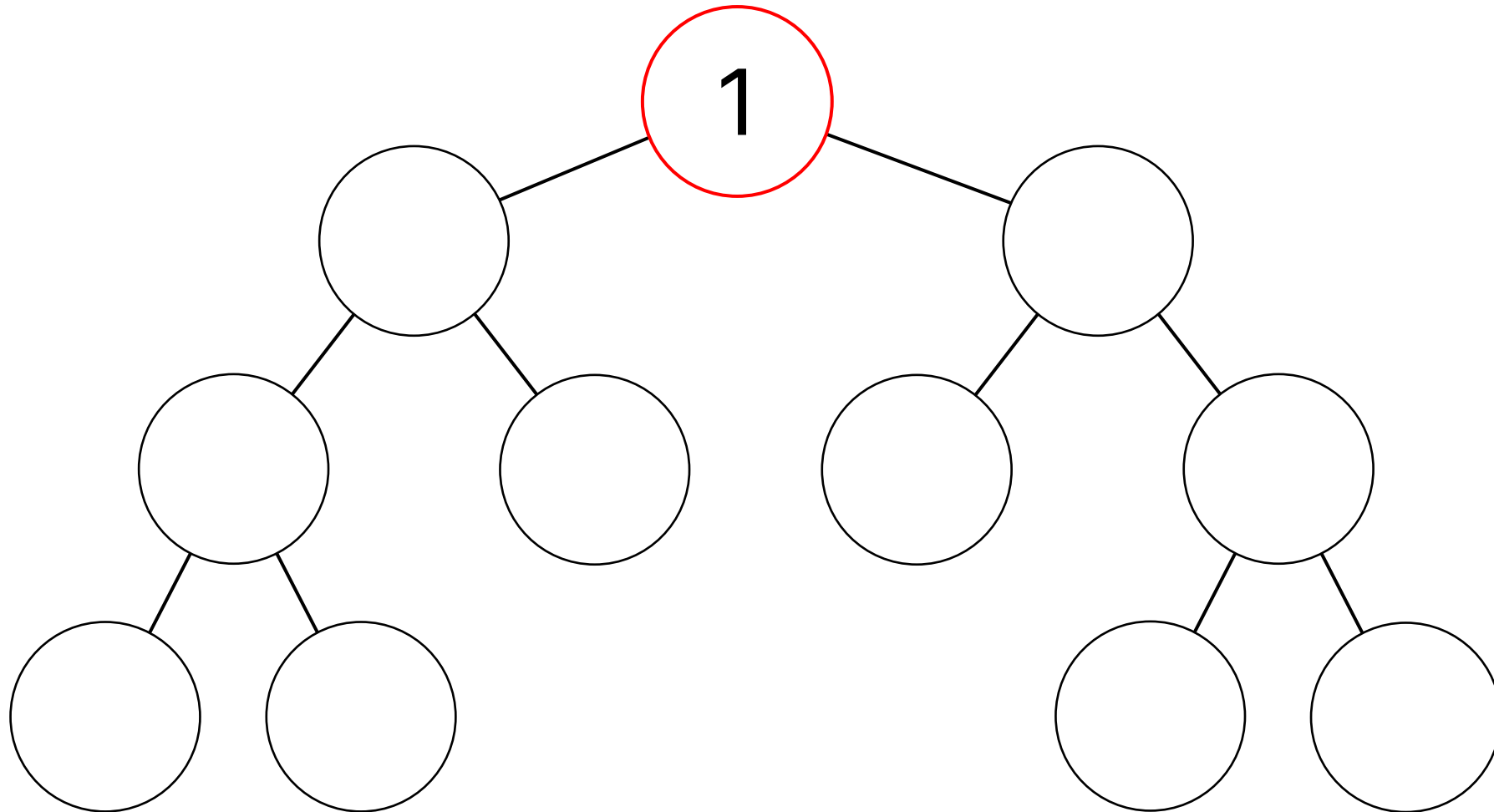

BFS

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

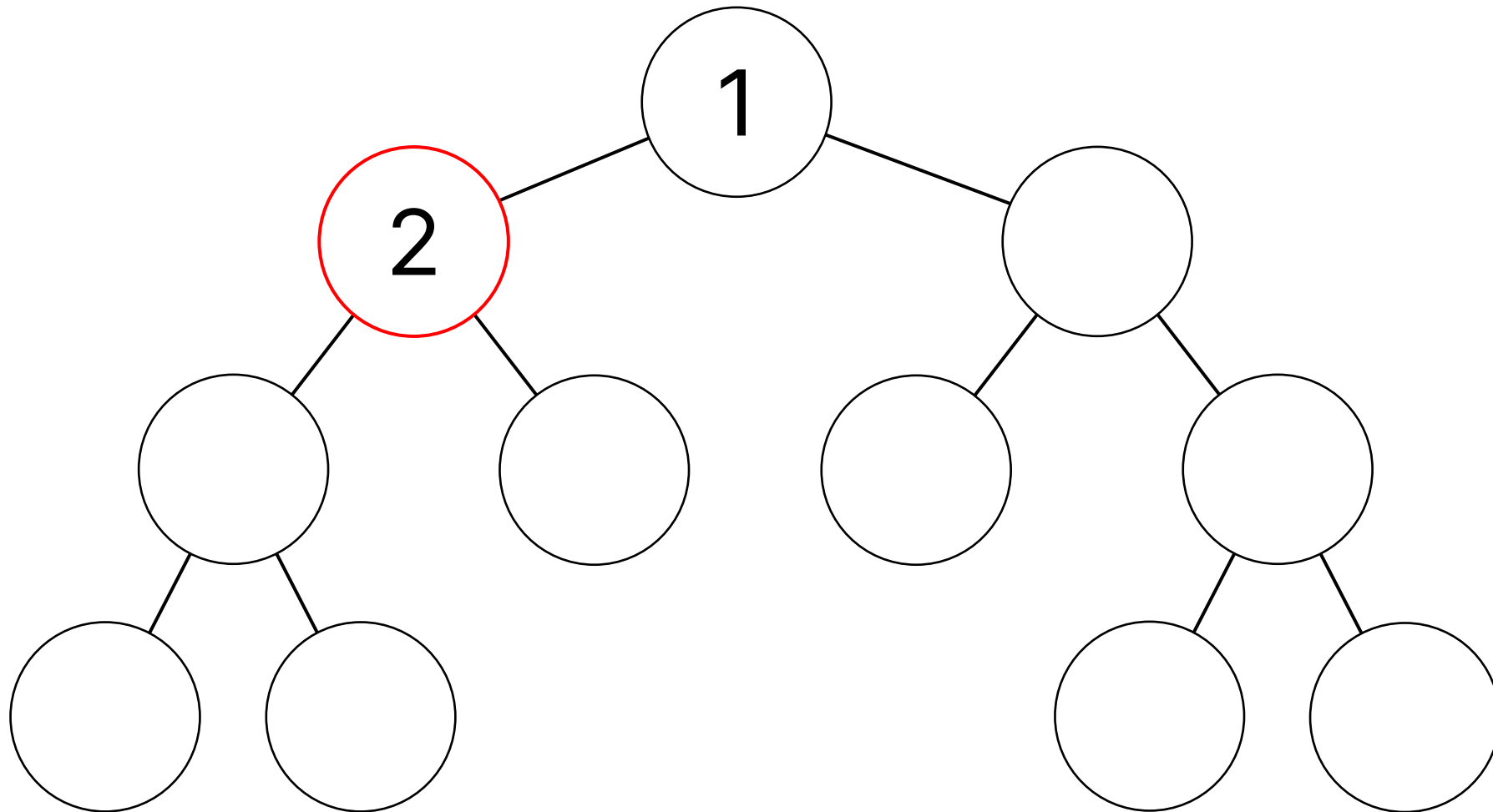
BFS



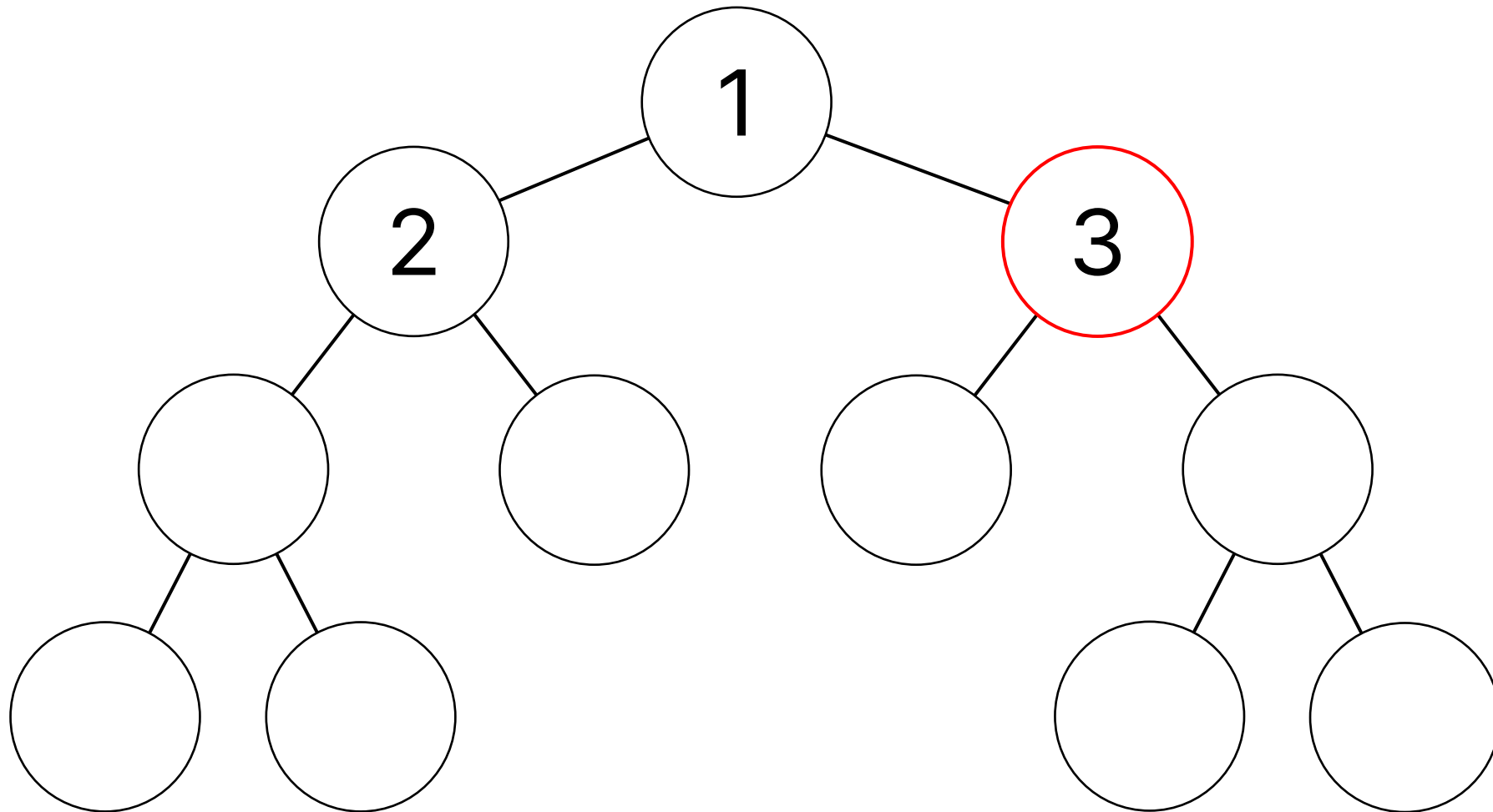
BFS



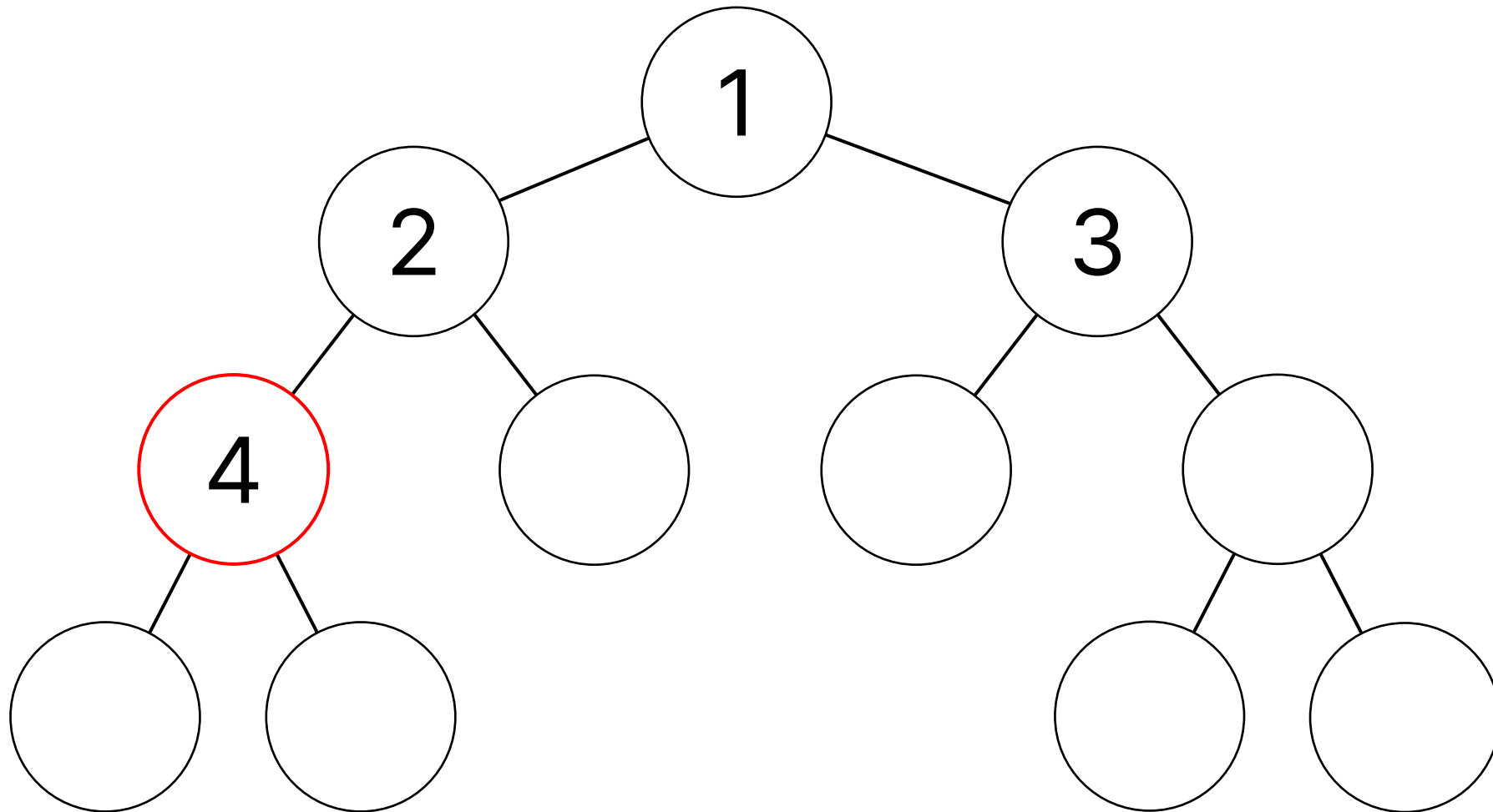
BFS



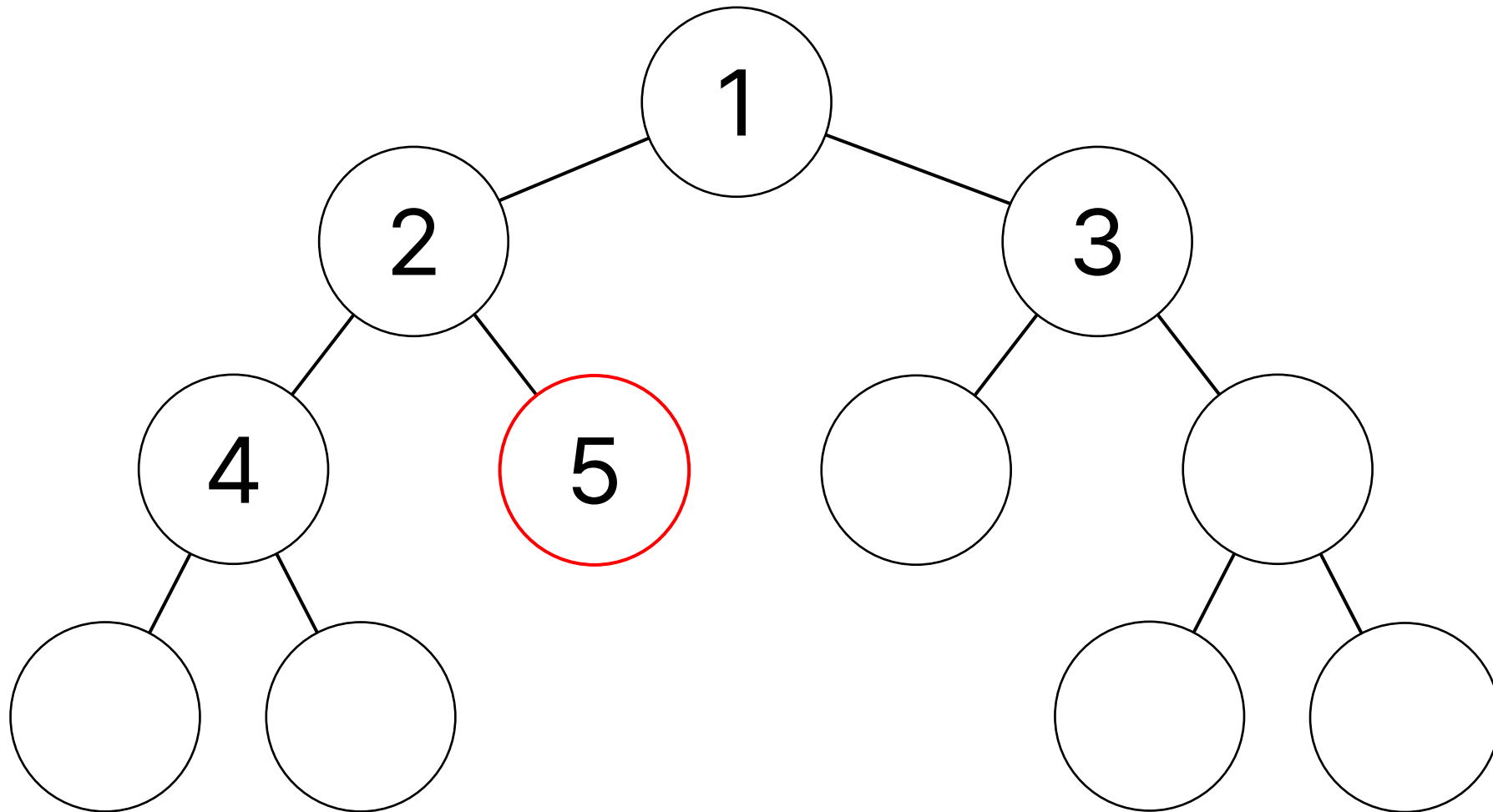
BFS



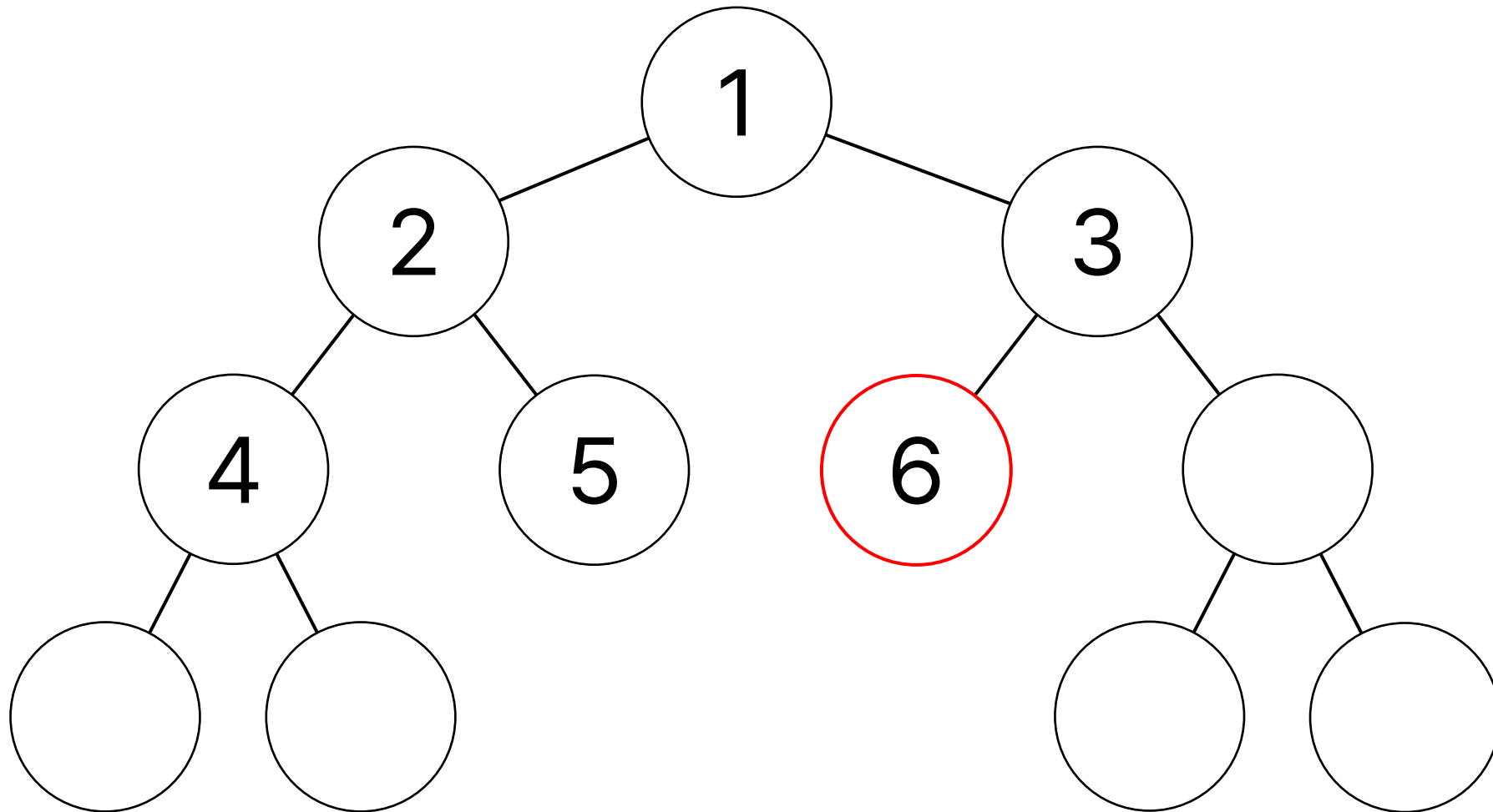
BFS



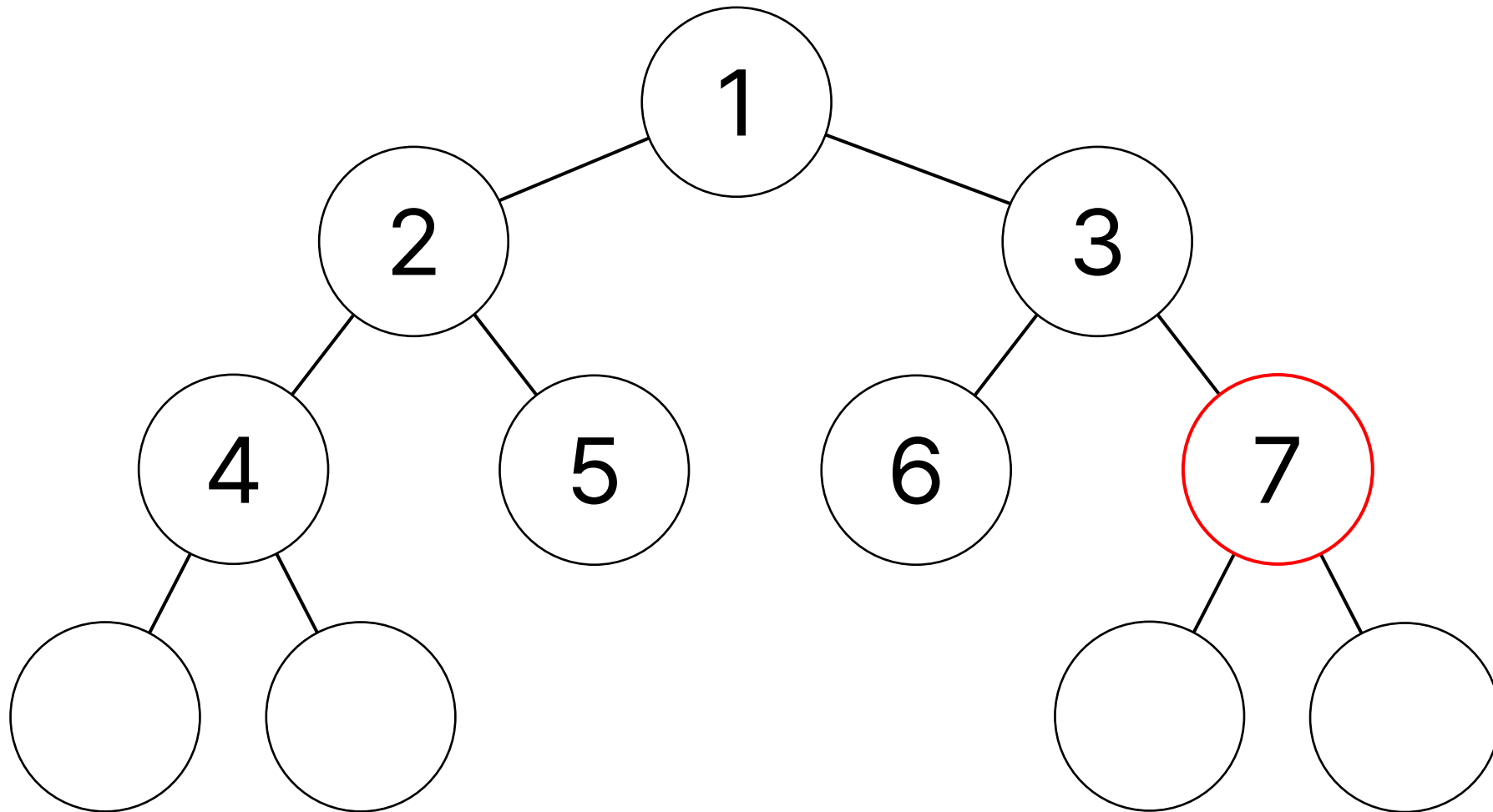
BFS



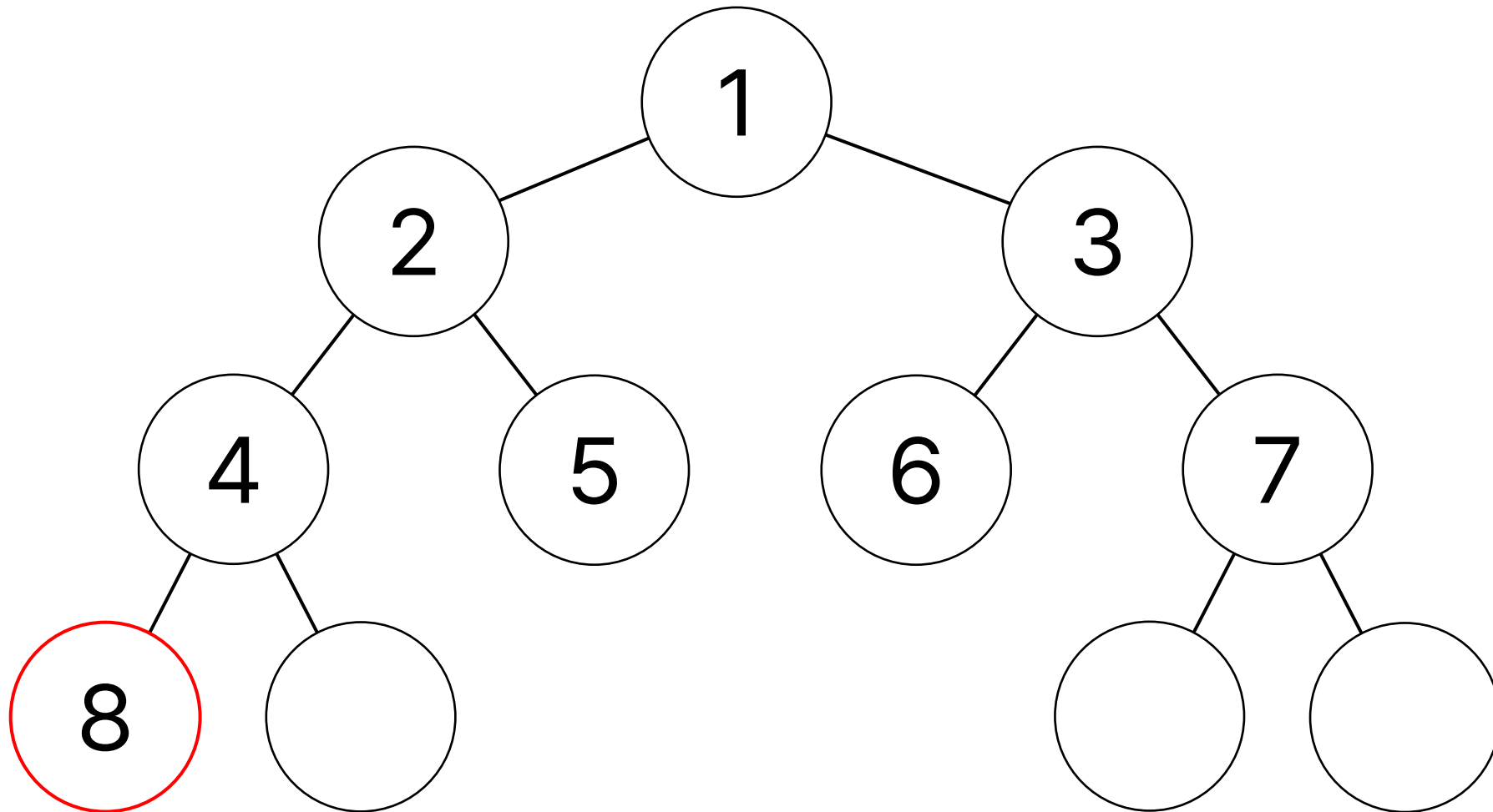
BFS



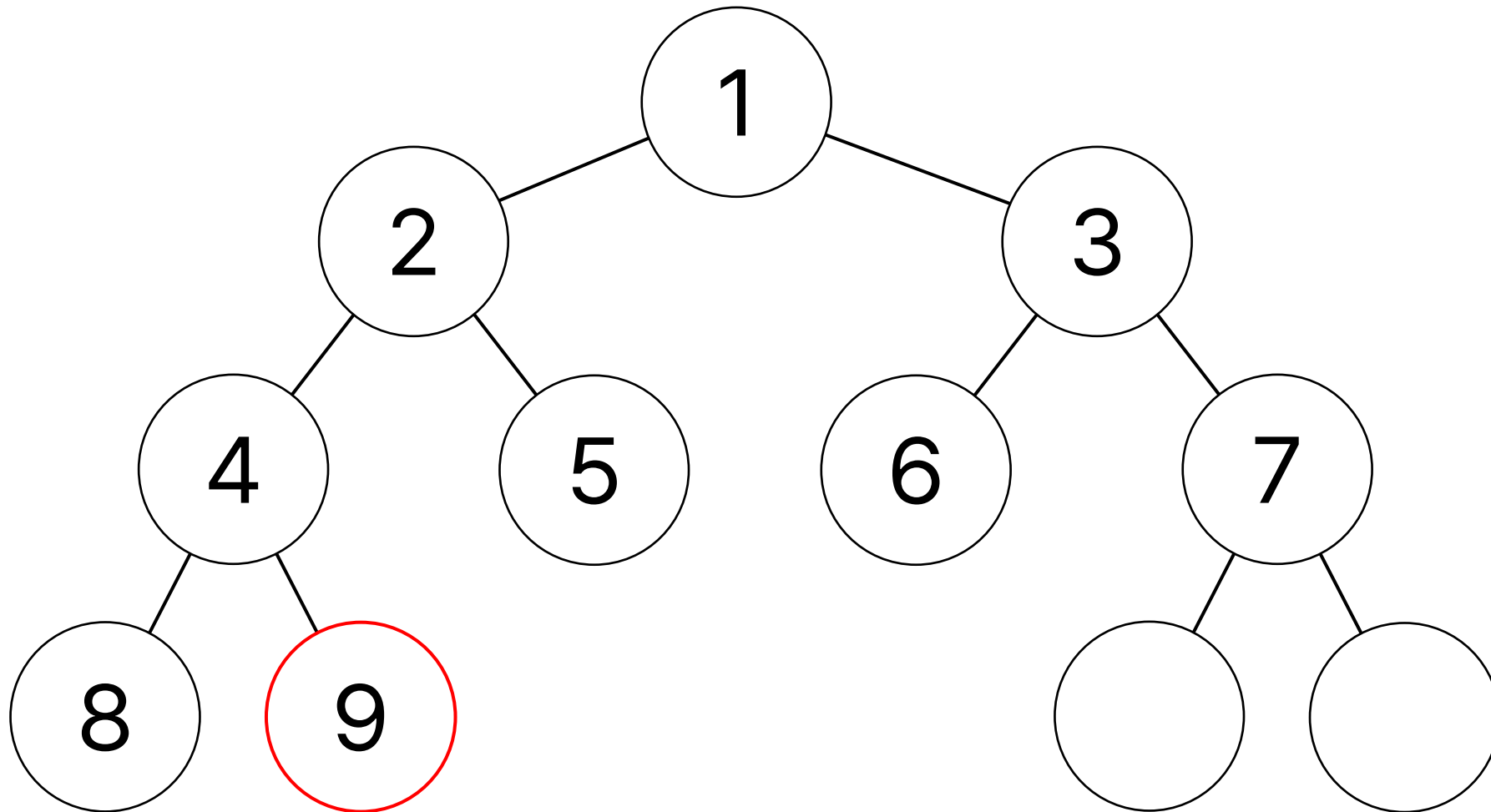
BFS



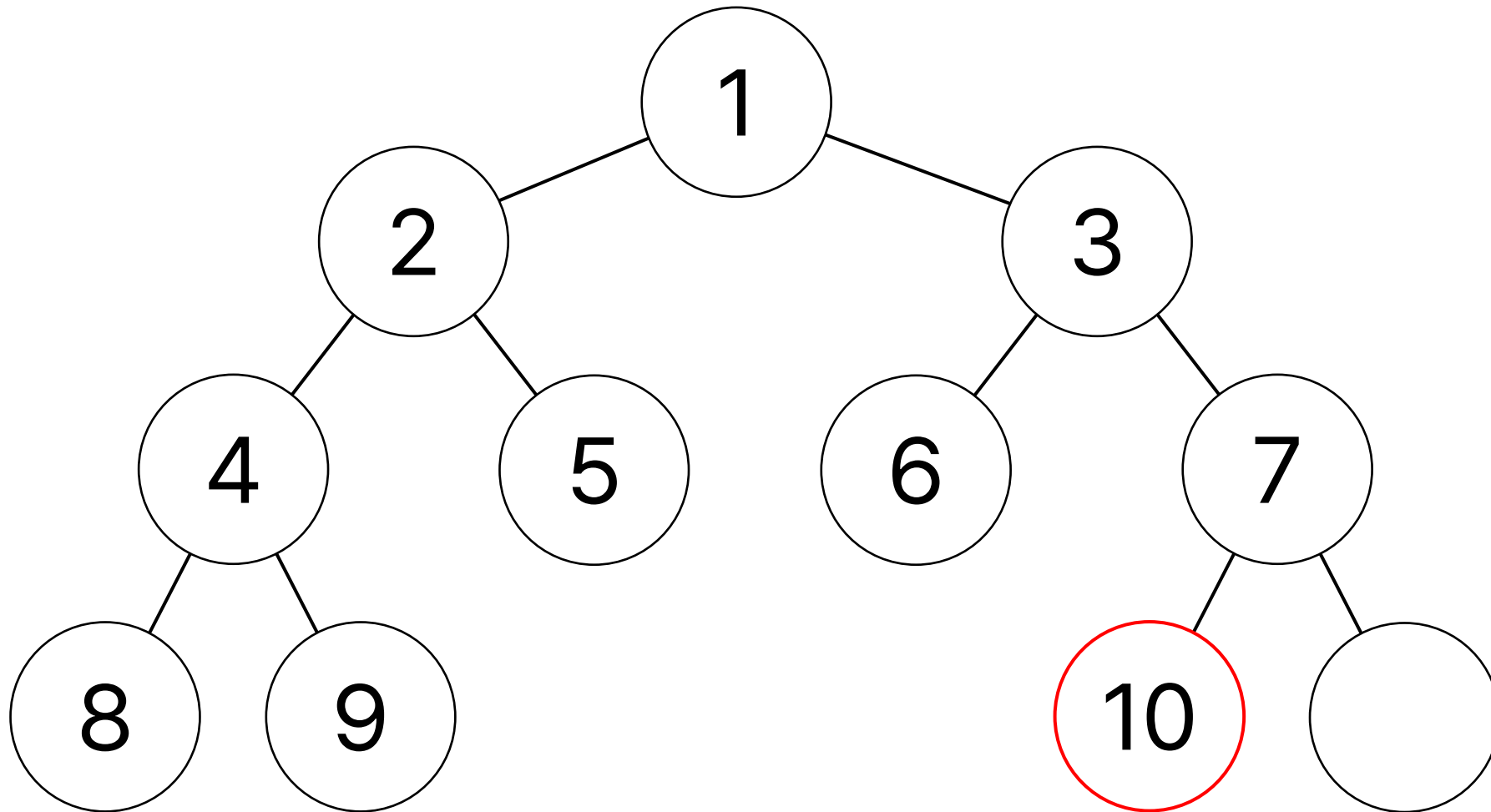
BFS



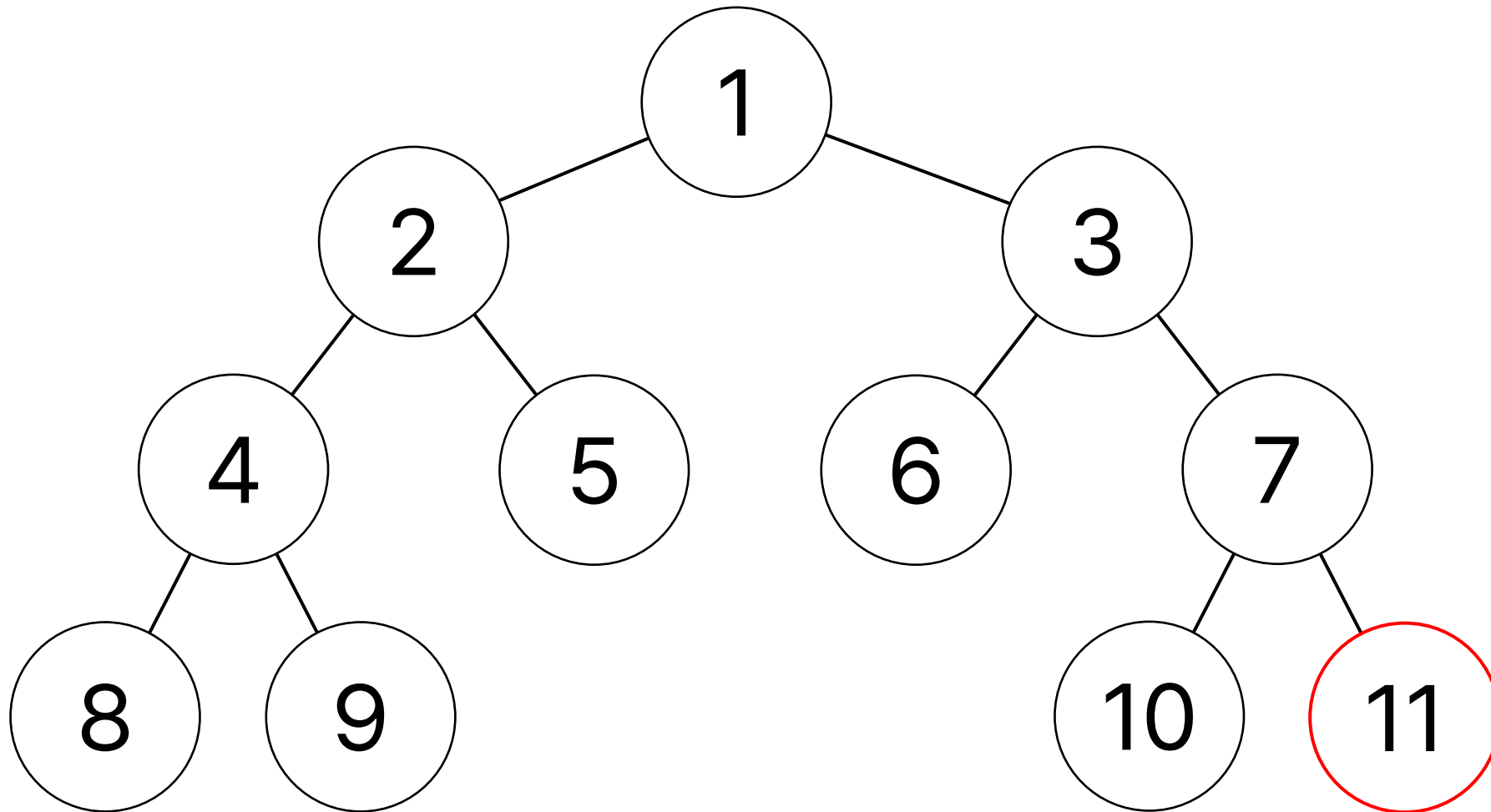
BFS



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



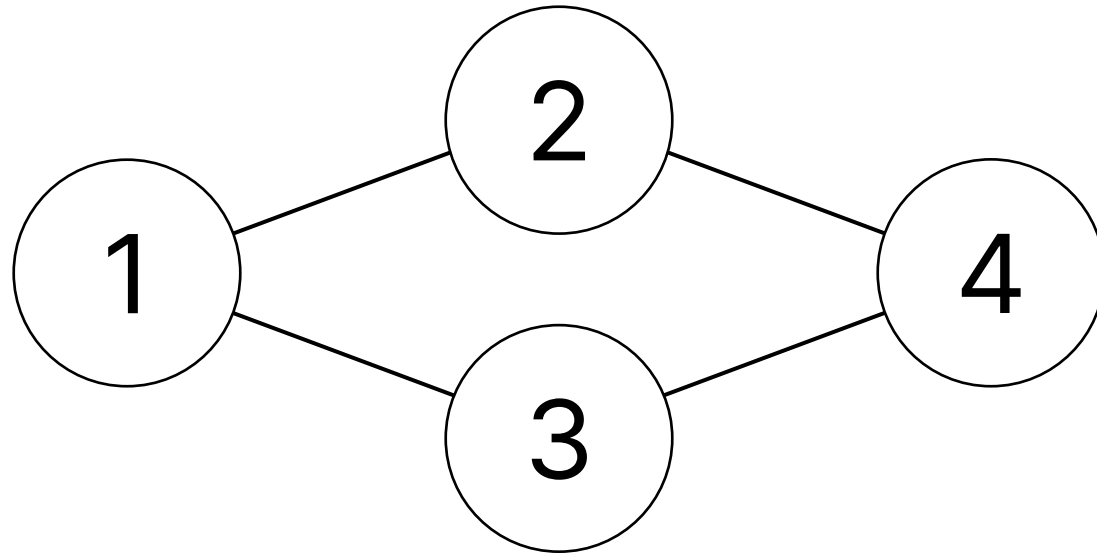
```
vector<int> edges[MAX_NODE];
queue<int> bfs_queue;
bool visited[MAX_NODE];

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;
            }
        }
    }
}
```

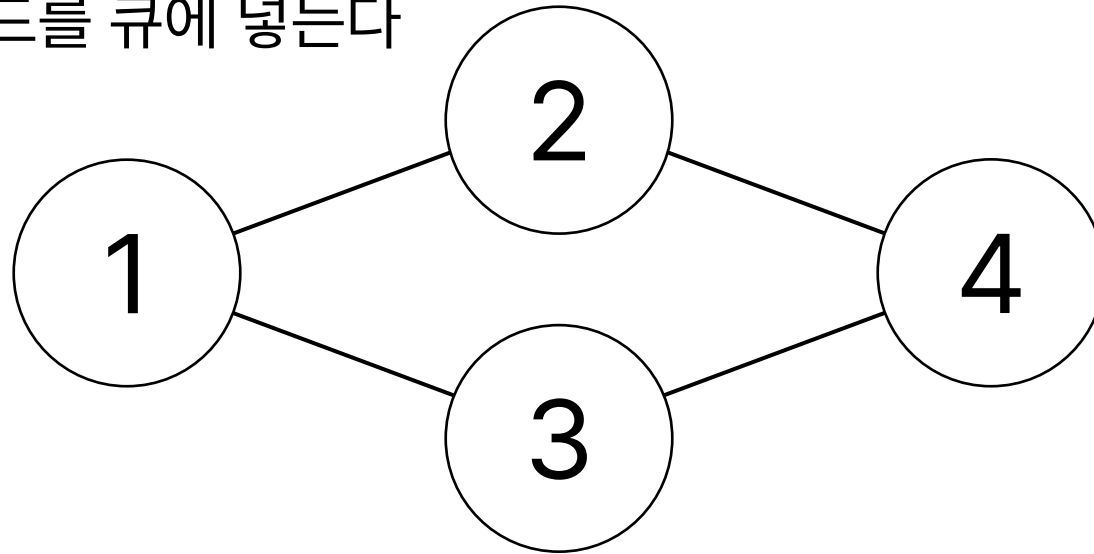

BFS

- 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



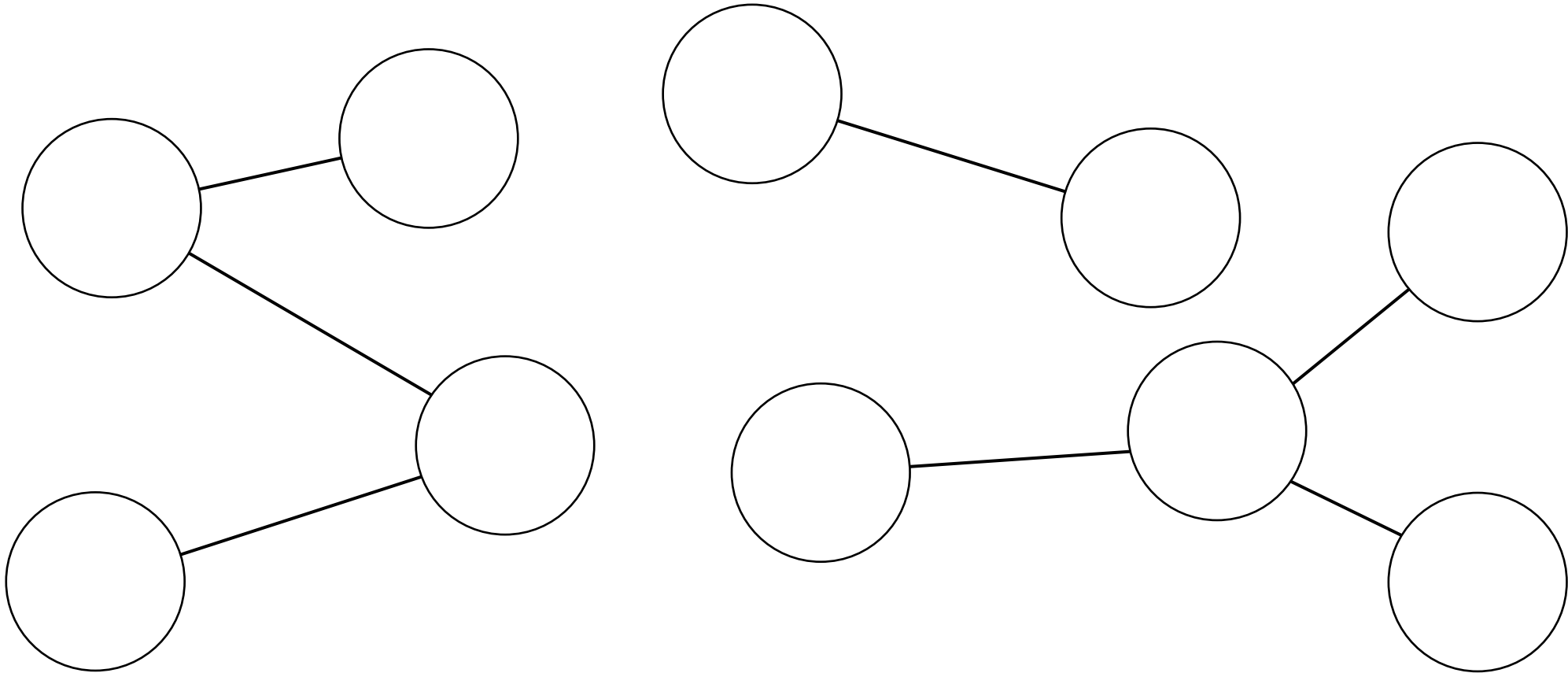
BFS

- 큐에 들어간 노드들은 탐색이 확정되었다
 - 탐색하기전 해당 노드를 다시 큐에 넣는다면 중복된 노드가 큐에 들어간다
 - 하나의 노드를 여러번 탐색하게 된다
-
- BFS를 사용하는 문제에서 메모리 초과가 발생하는 경우 제일 먼저 확인
->큐에 넣는 순간에 방문 여부를 체크 했는가?

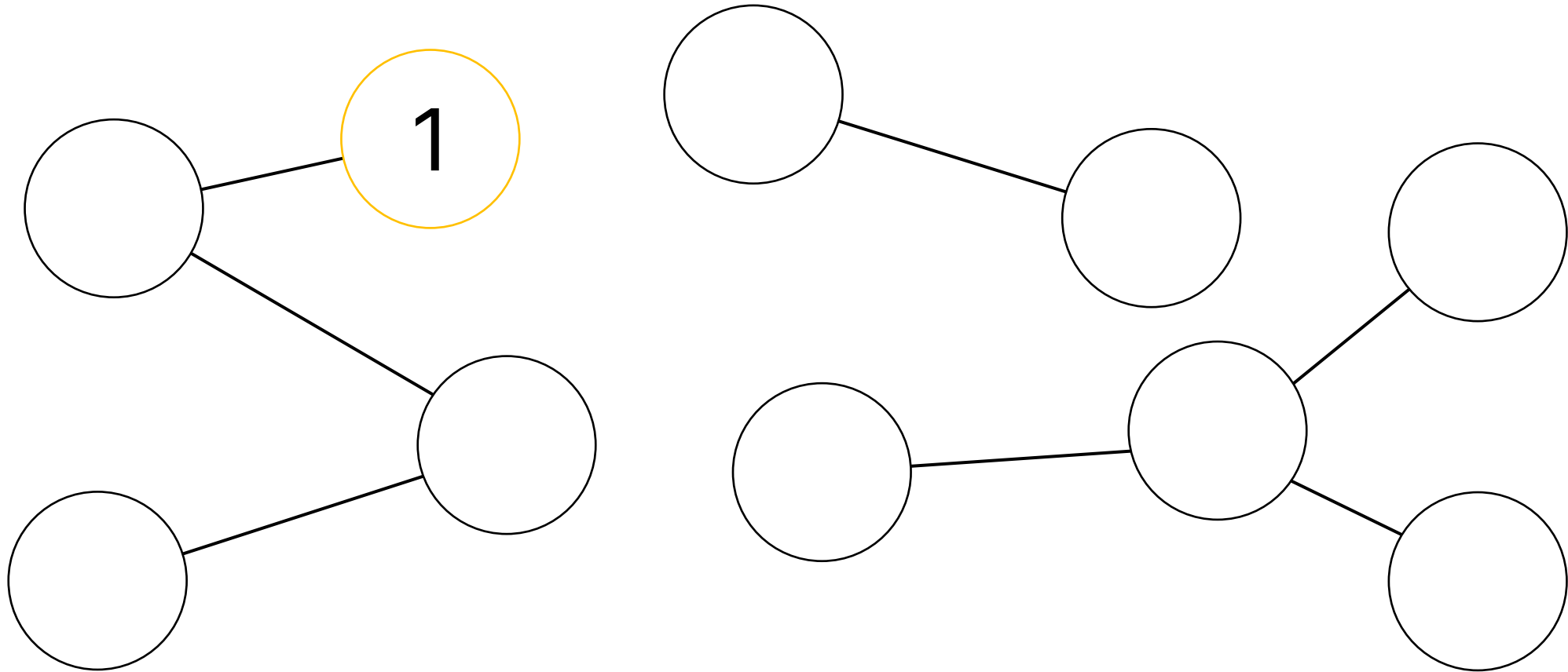
Traversal

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

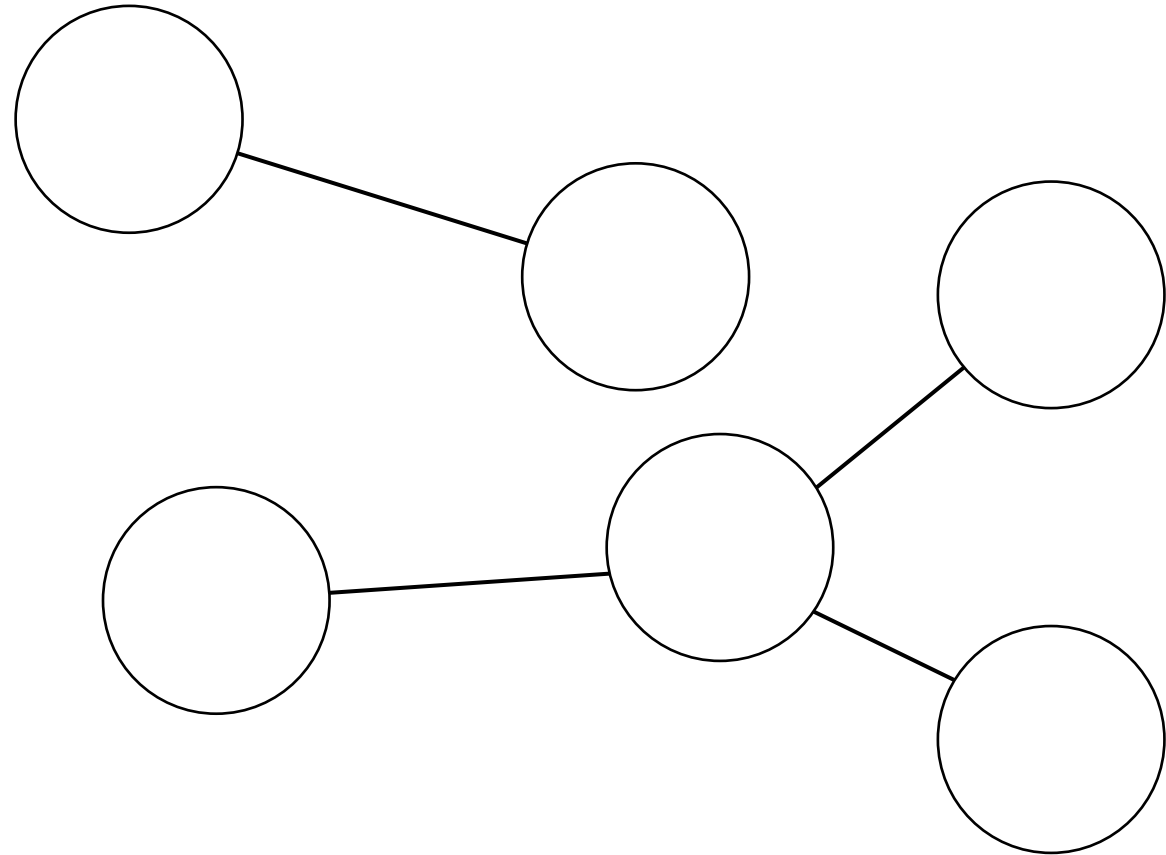
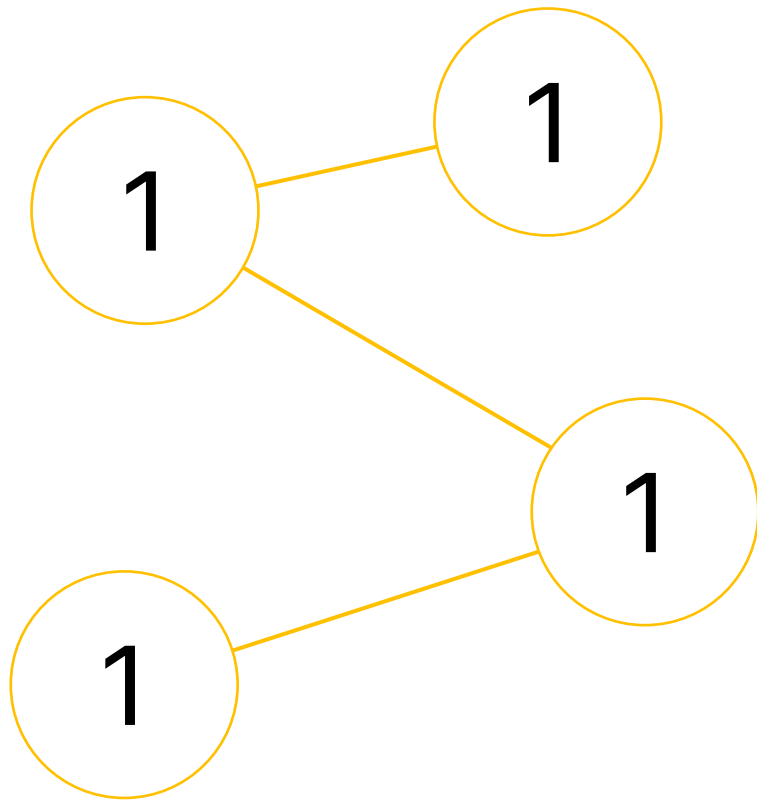
Connected Component



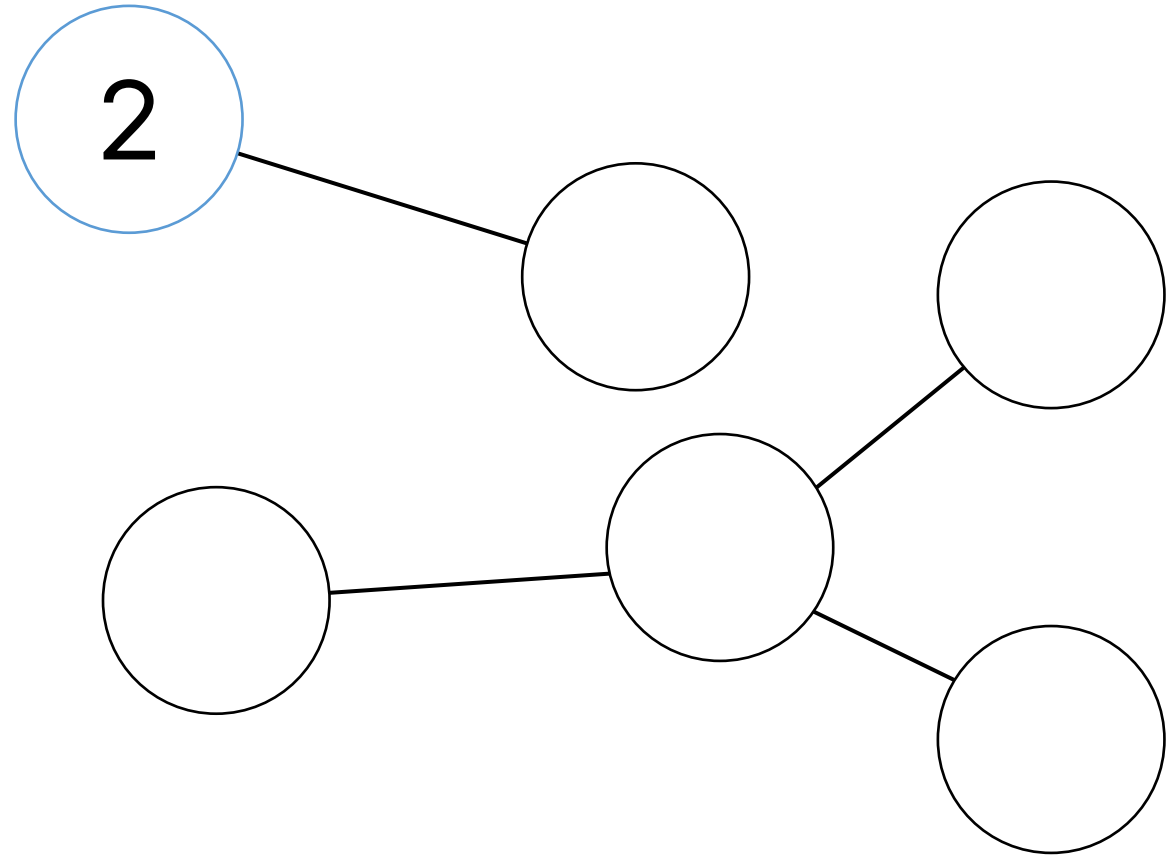
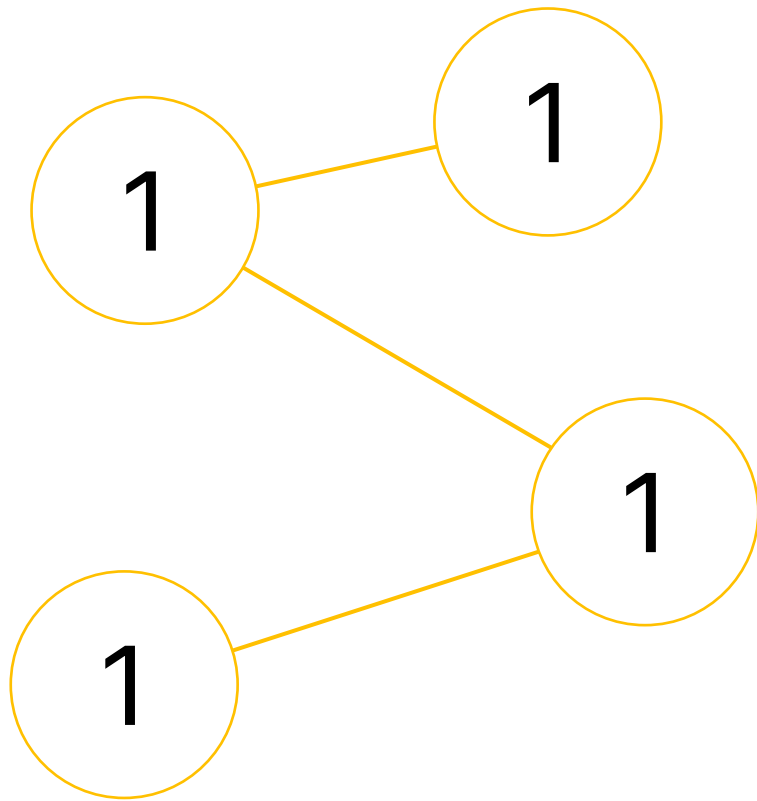
Connected Component



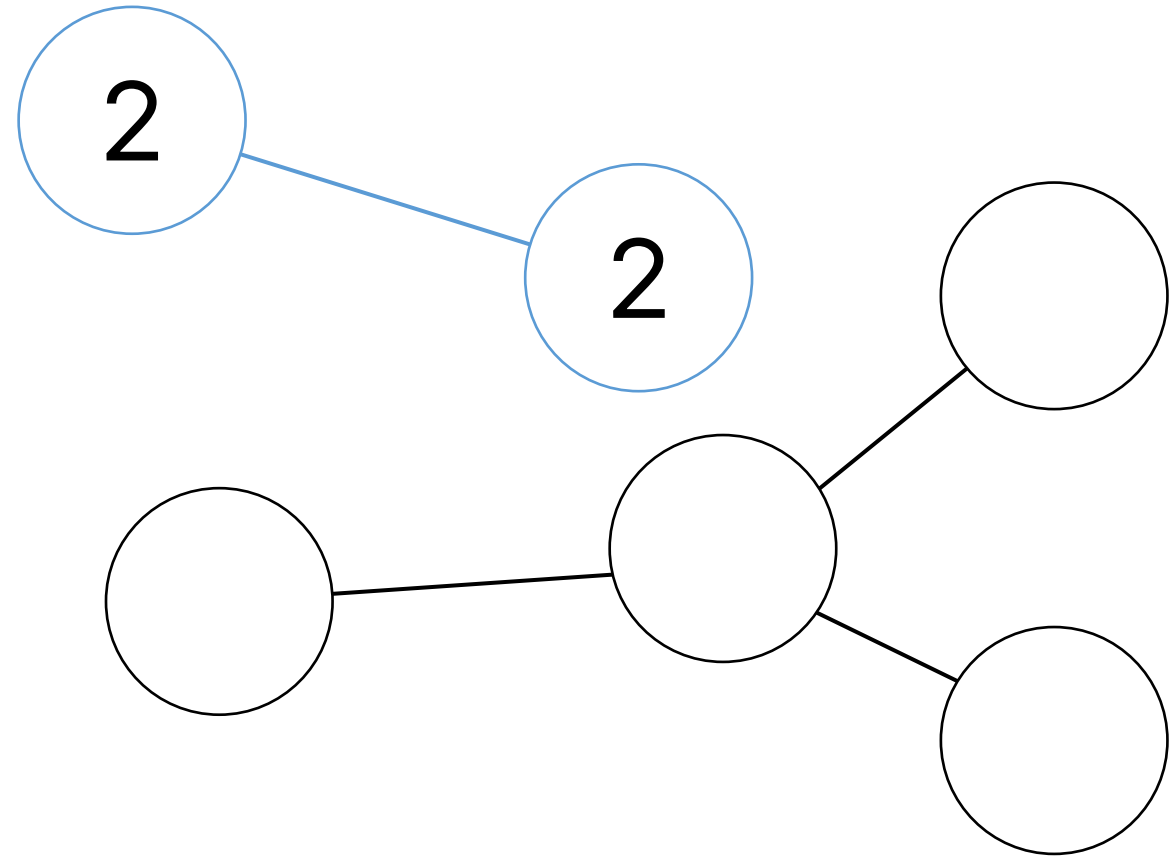
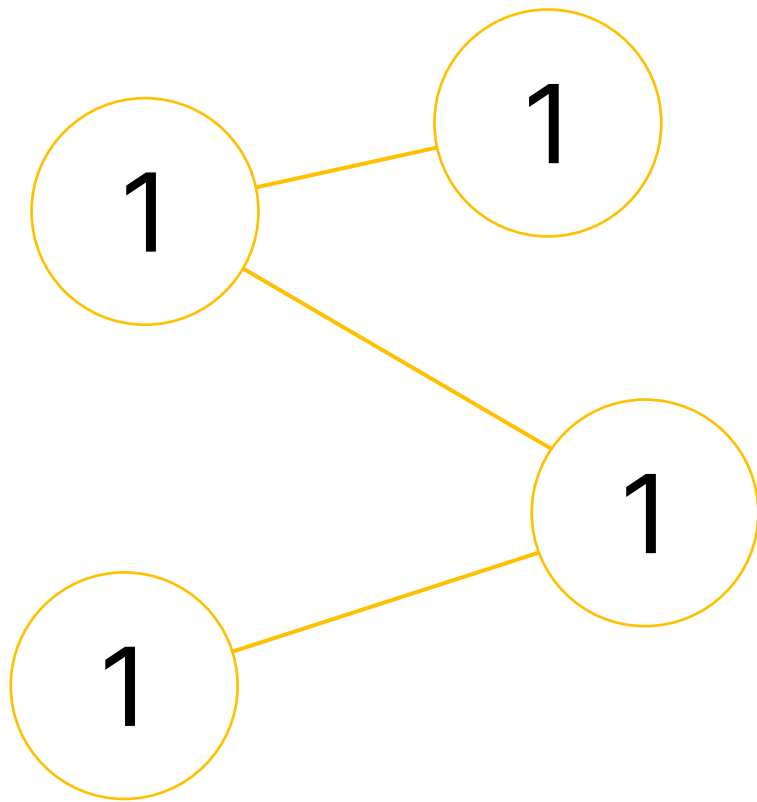
Connected Component



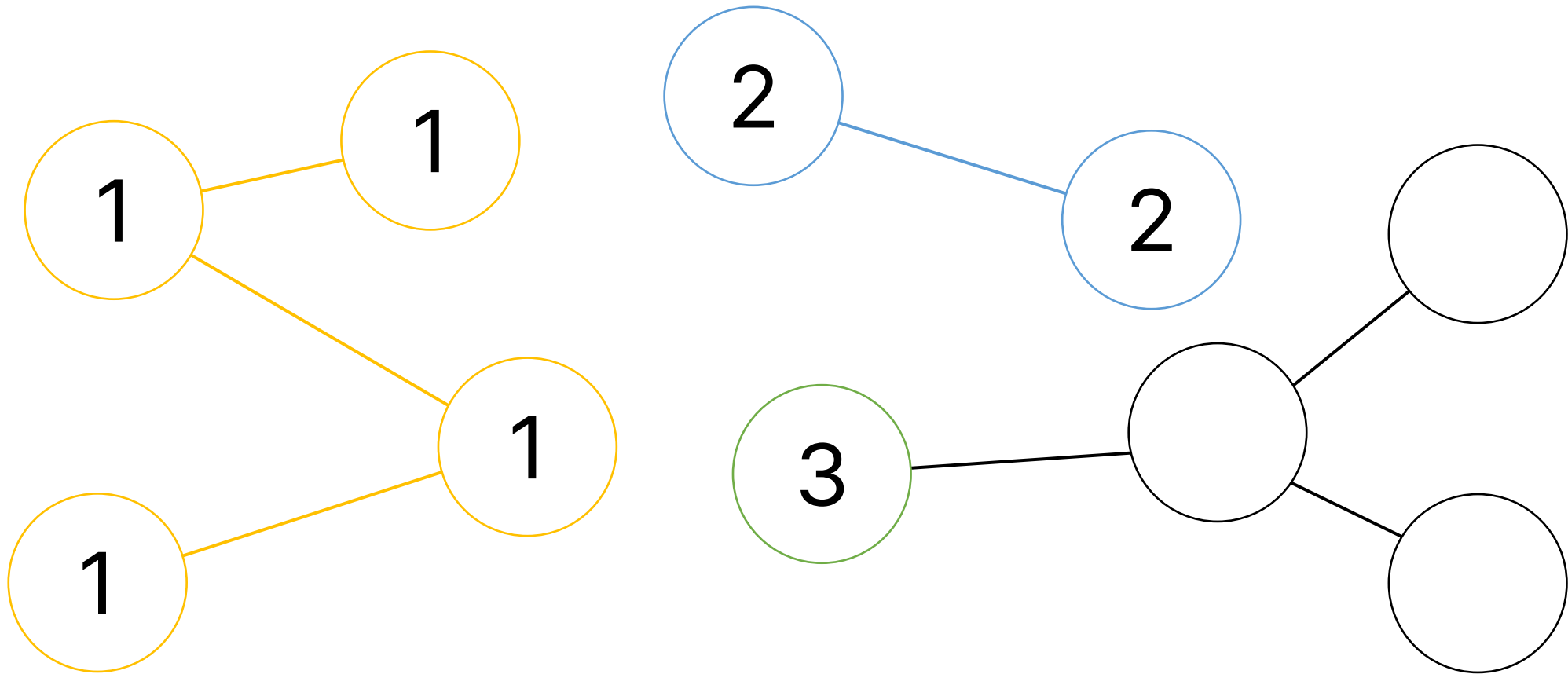
Connected Component



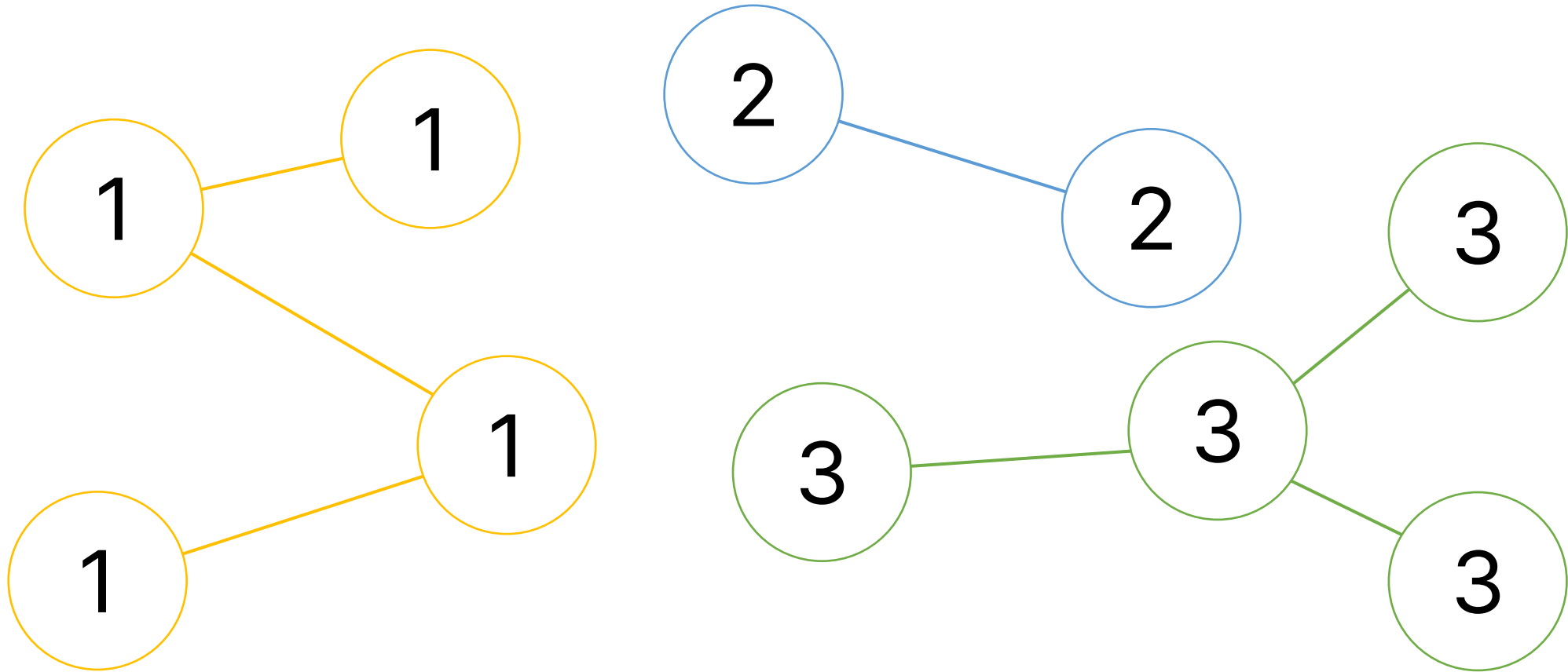
Connected Component



Connected Component



Connected Component



Connected Component

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

연습 문제

4963

: 섬의 개수

7562

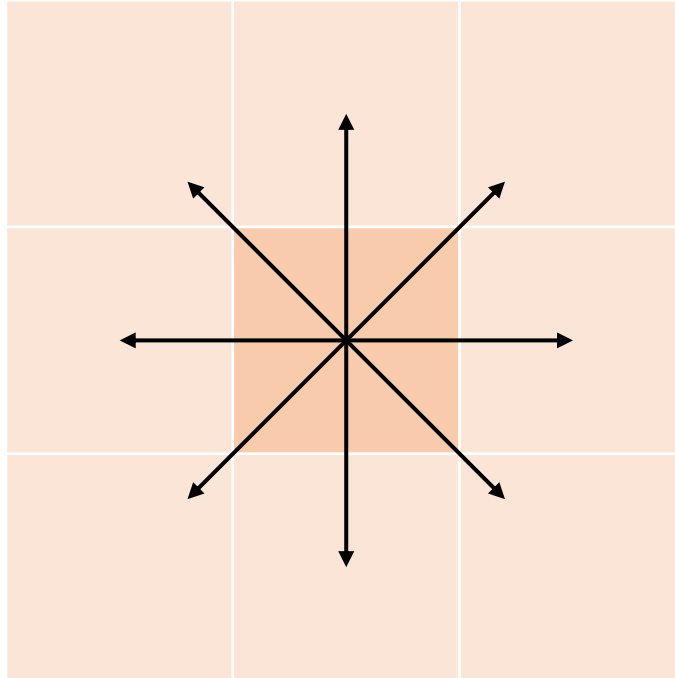
: 나이트의 이동

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

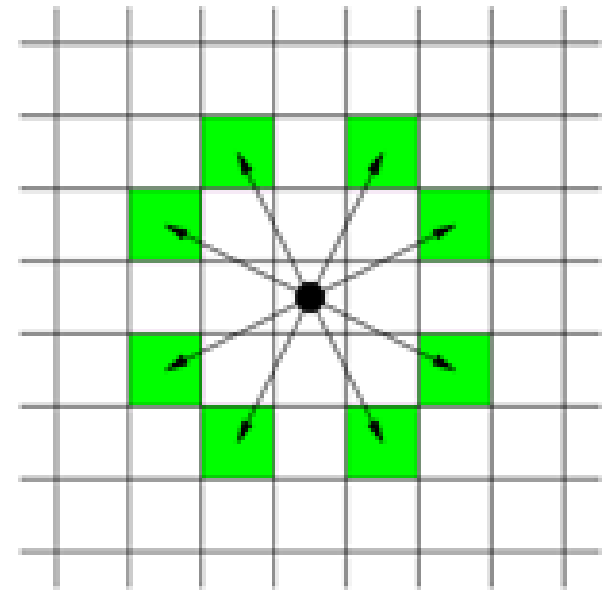


```
int dx[] = {1, -1, 0, 0, 1, 1, -1, -1};
int dy[] = {0, 0, 1, -1, 1, -1, 1, -1};

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]);
    }
}
```

나이트의 이동 BOJ 7562

- 체스판의 크기와 나이트의 현재 좌표, 나이트가 이동하려는 좌표가 주어진다.
- 각 테스트 케이스마다 나이트가 몇 칸만에 이동할 수 있는지 출력하라

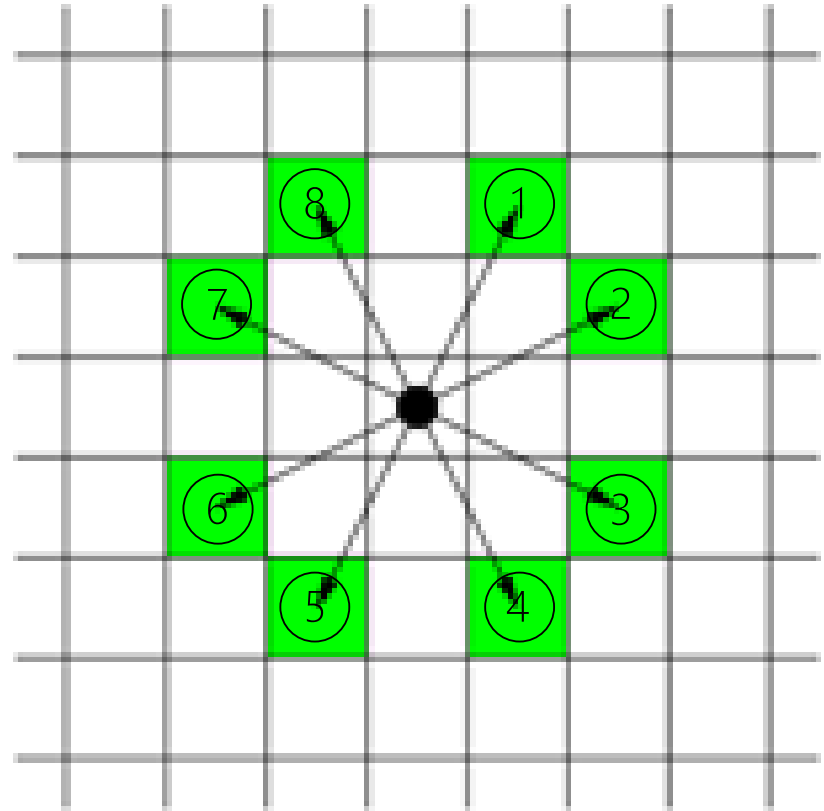


나이트의 이동 BOJ 7562

- 최단거리를 찾는 문제이므로 BFS를 이용해서 답을 알아낼 수 있다
- 큐에 다음 노드를 추가하기 위해서 이동하려는 좌표를 알아내고 이동하려는 좌표가 유효한 좌표인지 확인해야 한다
- 나이트가 이동할 수 있는 좌표는 총 8가지이므로 단순히 8개의 if문을 사용할 수 있지만 dx, dy배열과 반복문을 사용해 간단하게 나타낼 수 있다

나이트의 이동 BOJ 7562

- 아래(x)랑 오른쪽(y)으로 가는 것을 좌표가 증가한다고 하자
- 1번의 이동점은 $[x - 2][y + 1]$ 로 나타낼 수 있다
- $[x + dx][y + dy]$ 형태를 사용해 dx, dy 배열을 나이트의 이동에 맞춰 만들고 섬의 개수 문제와 동일하게 풀이하면 된다



나이트의 이동 BOJ 7562

- 다음과 같이 나타낼 수 있다

	1	2	3	4	5	6	7	8
dx	-2	-1	1	2	2	1	-1	-2
dy	1	2	2	1	-1	-2	-2	-1

Reference

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