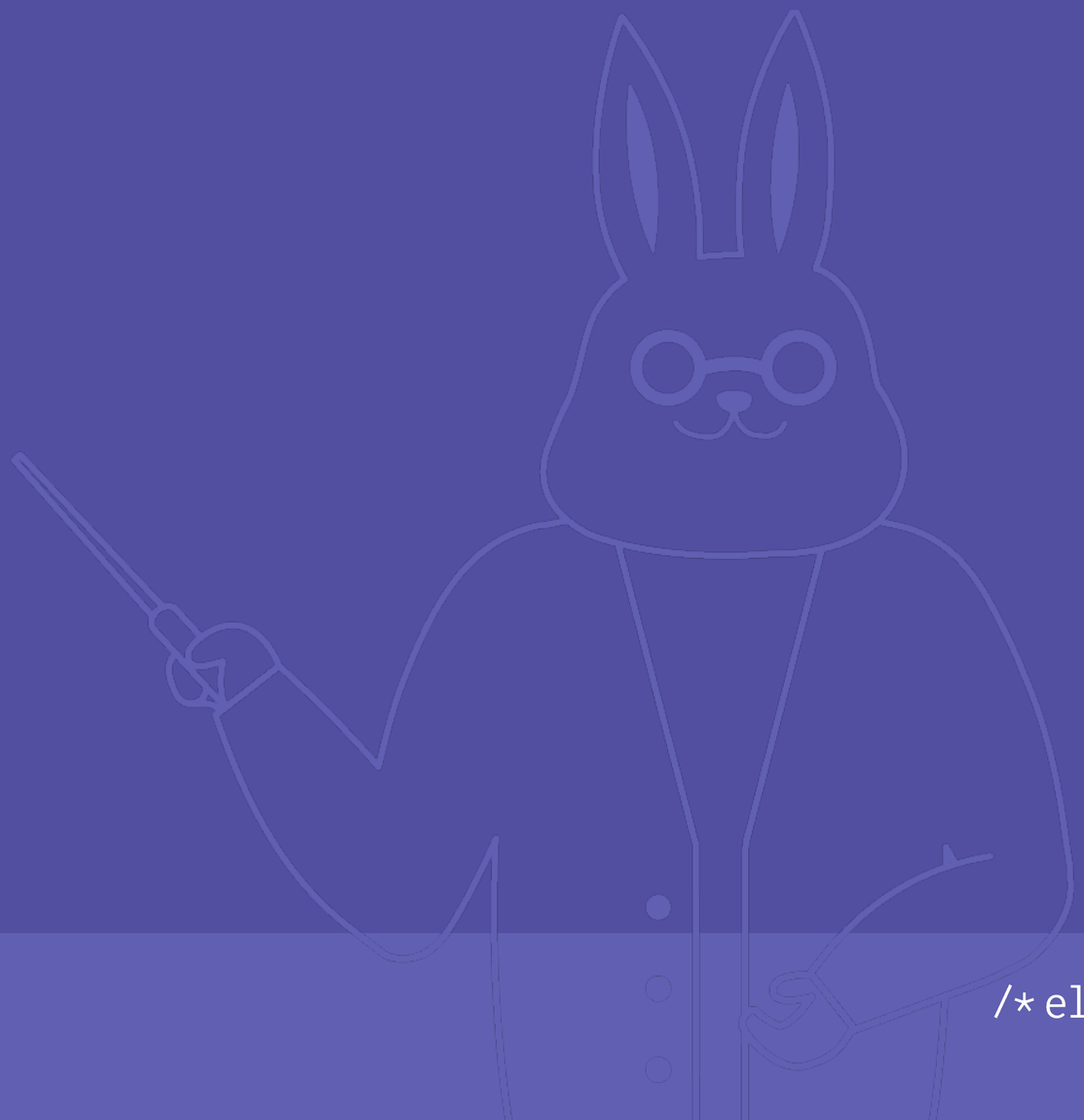
자료구조

3장 트리

김경민 선생님



Contents

01. 비선형구조와 트리

02. 트리의 표현 방법

03. 트리 순회하기

04. 트리의 활용

05. 트리 실습

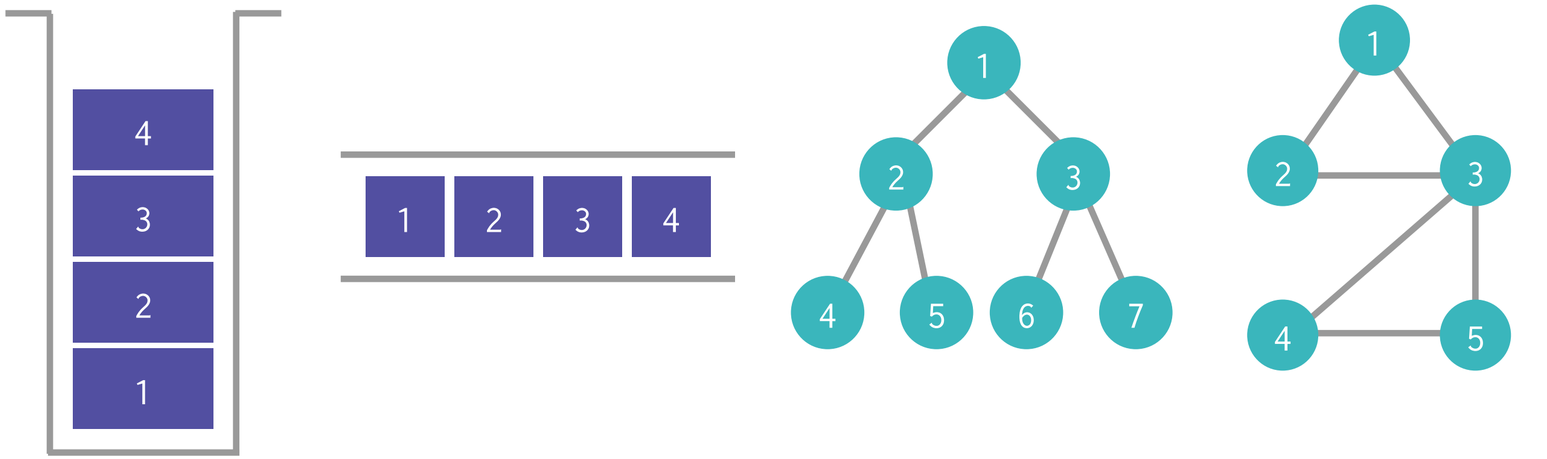
01

비선형구조와 트리

대표적인 자료구조의 예시

선형 구조

비선형 구조



스택 (Stack)　　큐 (Queue)　　트리 (Tree)　　그래프 (Graph)

대표적인 자료구조의 예시

선형 구조 : 자료가 **순서**를 가지고 **연속**되어 있음

비선형 구조 : 선형 구조에 **해당하지 않는** 자료구조

대표적인 자료구조의 예시

비선형 구조의 대표적 예시는 **트리**와 **그래프**이다.

그래프

트리는 **그래프의 특수한 형태** 중 하나이다.

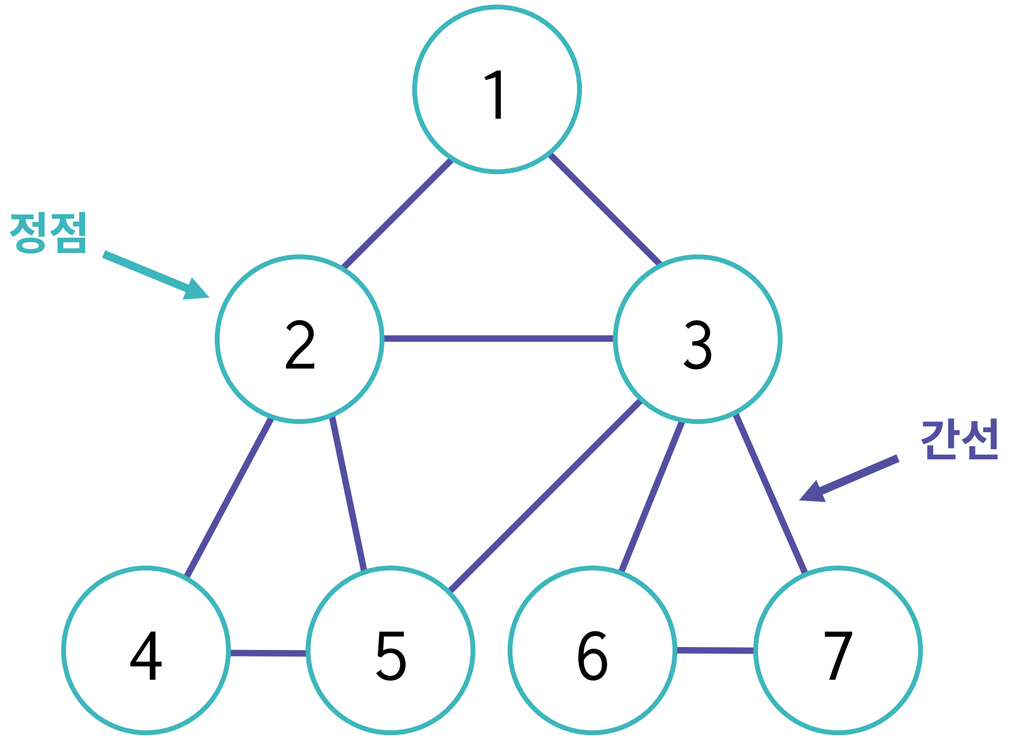
트리를 이해하기 위해 우선 그래프의 정의부터 알아보자.

그래프

정점(vertex)과 **간선(edge)**으로 이루어져 있는 자료구조

정점(vertex): 자료, 상태 등 뭔가를 담고 있음
간선(edge): 정점 간의 관계를 나타냄

(정점은 '노드'라고도 표현한다.)

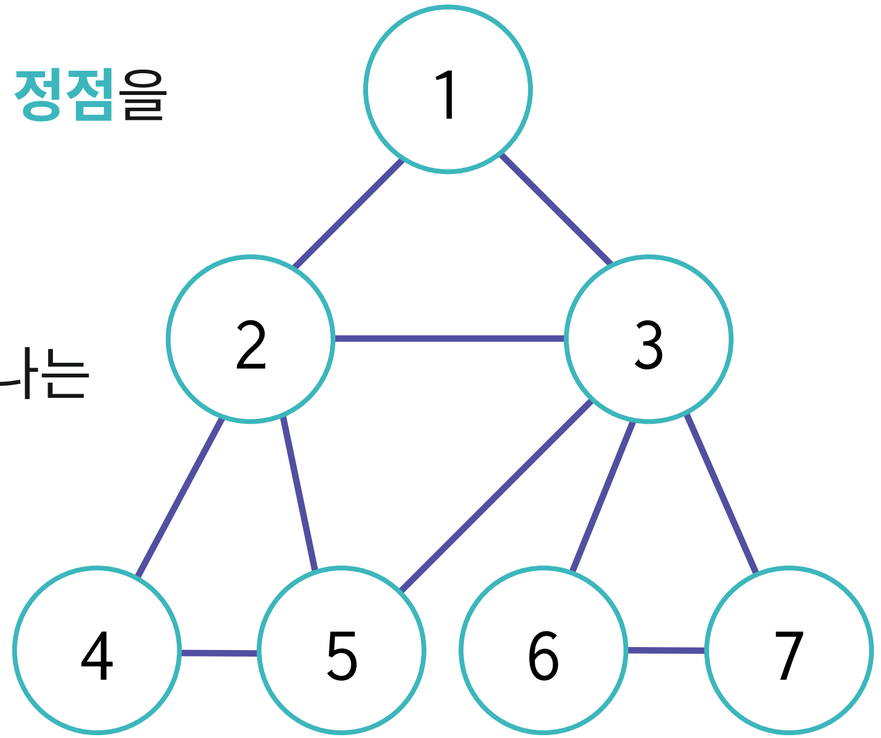


그래프

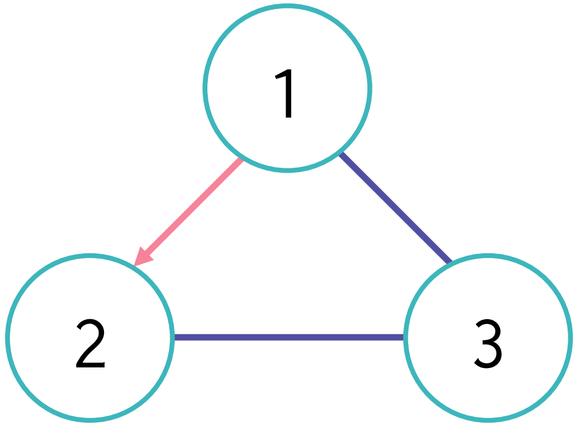
어떤 **정점**에서 **간선**을 통해 다른 정점으로 이동할 수 있다.

어떤 정점에서 **다른 정점으로 이동하기 위해 거치는 모든 정점**을 **경로**라고 한다.

예를 들어 정점 4에서 정점 6으로 이동하는 경로들 중 하나는 **4 → 5 → 3 → 6**이 될 수 있다.



그래프



그래프의 간선은 **방향**이 있을 수도, 없을 수도 있다.

위와 같은 그래프에서 **정점 1에서 2로** 이동할 수는 있지만, 2에서 1로 이동할 수는 없다.

방향이 있는 간선을 포함한 그래프를 **유향 그래프**라고 한다.

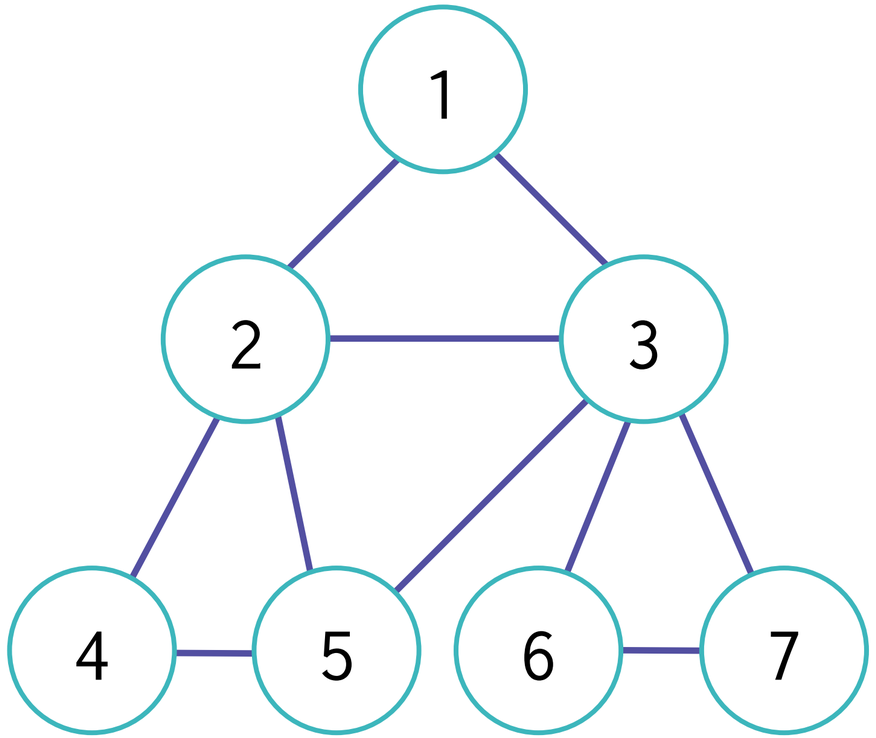
그래프

어떤 정점에서 출발하여 자기 자신으로 돌아오는 경로가 있을 수 있다.

이와 같이 **처음 시작한 정점**으로 다시 돌아오는 경로를

'사이클' 이라고 한다.

예를 들어 **3 → 6 → 7 → 3** 은 사이클이다.

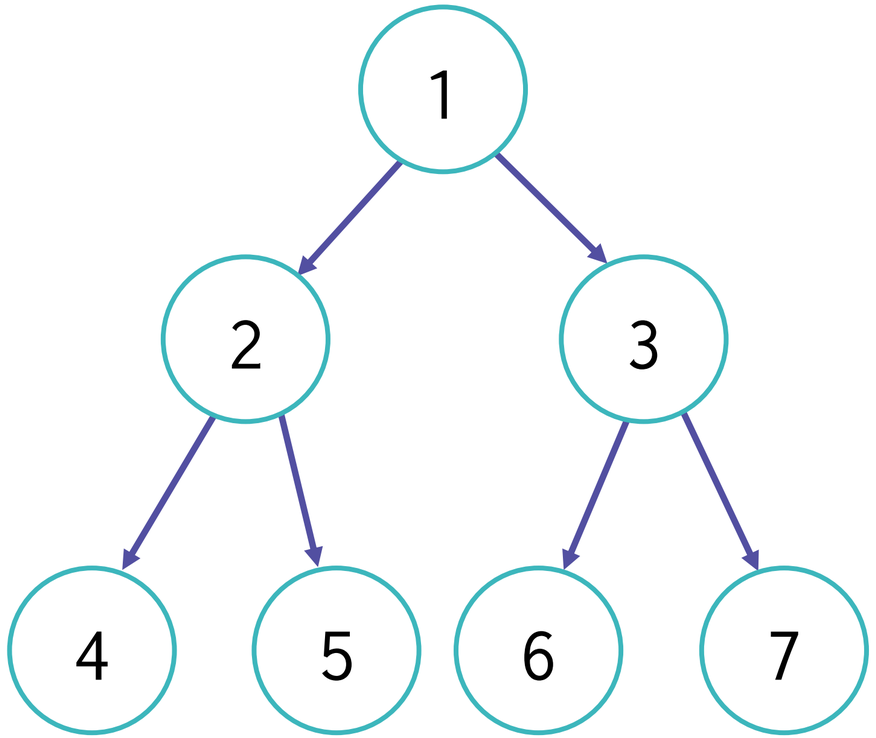


트리

그래프 중 아래의 **특별한 성질**을 갖는 그래프를 트리라고 한다.

트리의 여러 성질은 다음과 같다.

- 트리의 간선들은 모두 **방향성**을 갖는다.
- 어떤 정점을 가리키는 정점의 개수는 **최대 1개**이다.
- 어떤 **정점**에서 다른 **정점**으로 이동할 수 있는 **경로는 1개**다.
- 트리는 **사이클**을 갖지 않는다.



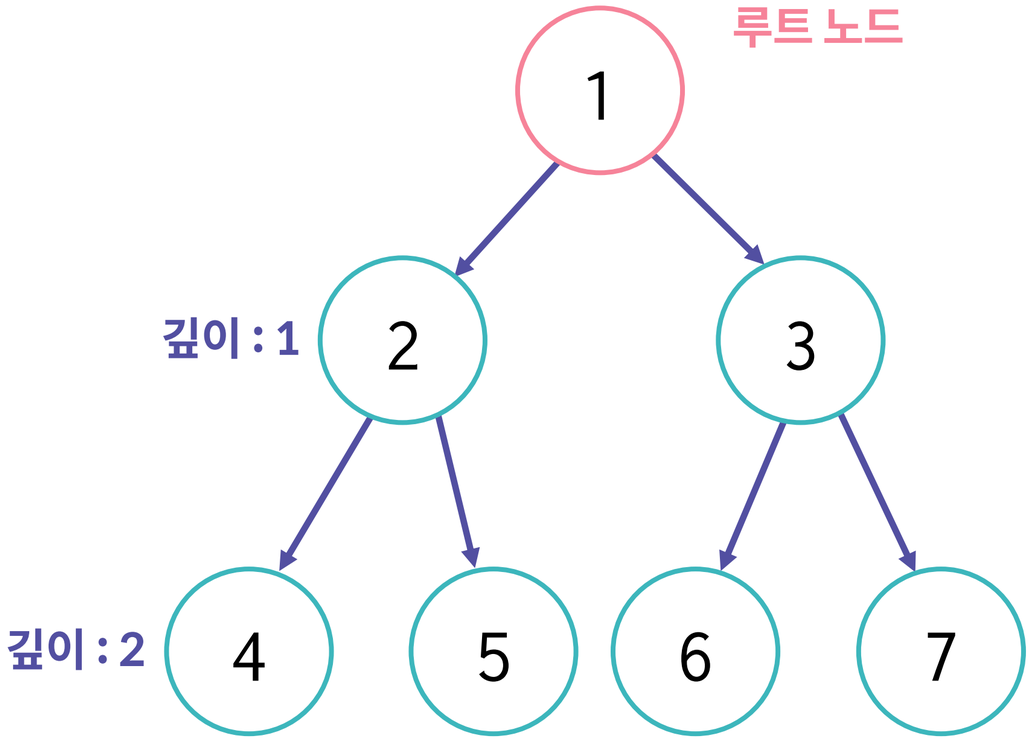
트리

트리에서 다른 어떠한 정점도 **가리키지 않는 정점**을

루트 노드(Root Node) 라고 한다.

정점 1이 **루트 노드**에 해당한다.

또, 루트 노드로부터의 거리를 **깊이**라고 한다.



트리

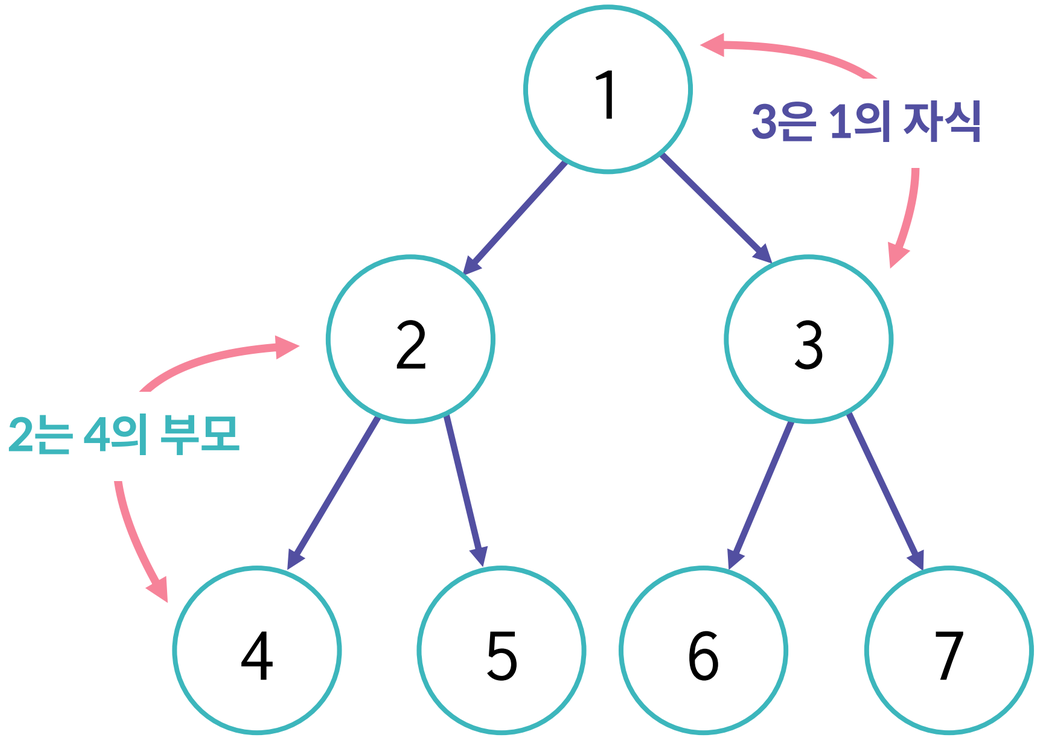
임의의 정점 A가 **다른 정점** B을 가리킬 때

A를 B의 **부모 노드(Parent Node)**라고 하고,

B를 A의 **자식 노드(Child Node)**라고 한다.

예를 들어 **정점 2**는 **정점 5**의 **부모 노드**이고,

정점 3은 **정점 1**의 **자식 노드**이다.

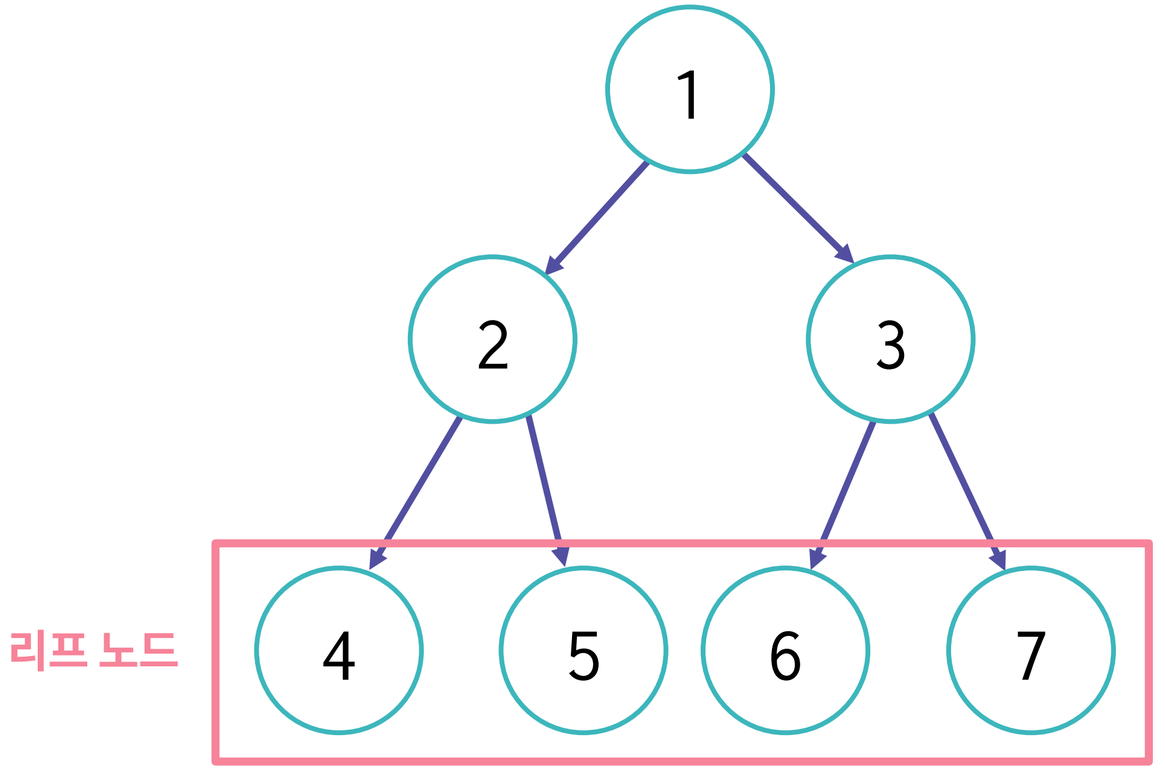


트리

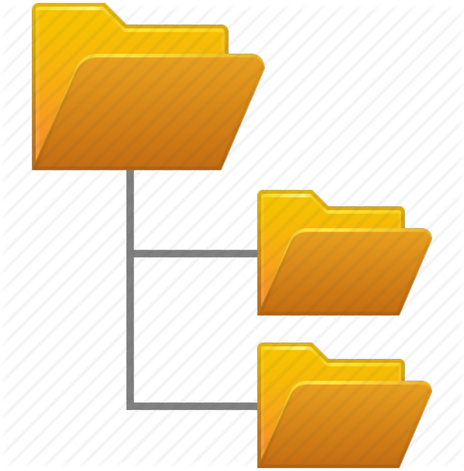
가리키는 정점이 없는 정점들을

리프 노드(Leaf Node)라고 한다.

정점 4, 5, 6, 7이 **리프 노드**에 해당한다.



트리

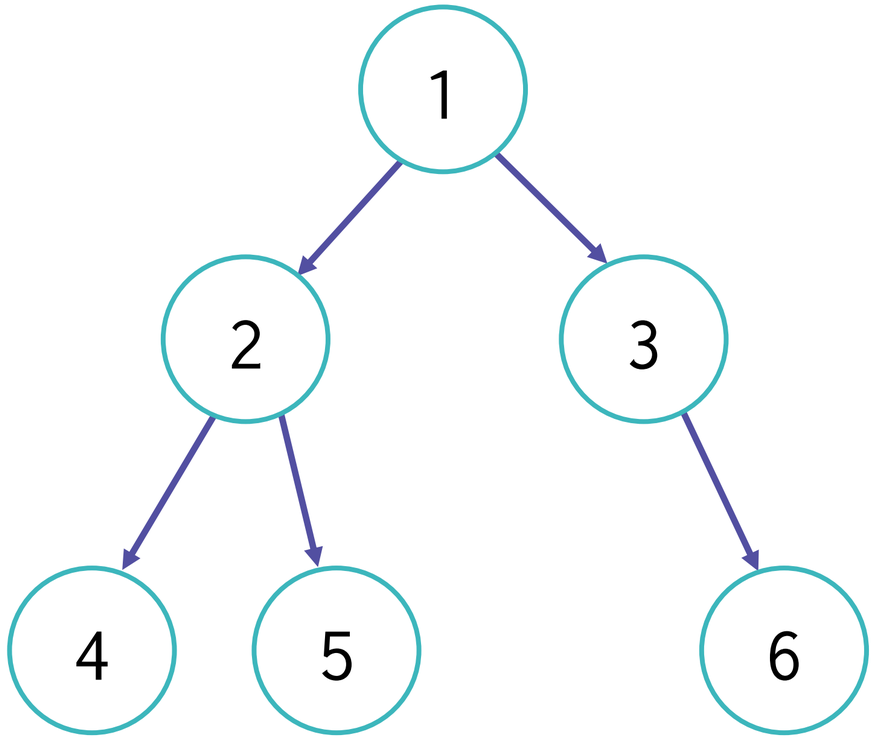


트리는 **계층적**인 구조로 되어 있는 자료구조이다.
운영체제에서 **파일**을 **분류**하기 위해 사용하는 **디렉터리(폴더)**는
트리 구조의 대표적인 예시이다.

이진 트리

각 정점들이 자식 노드를 **최대 2개까지**만 갖는 트리를 **이진 트리**라고 한다.

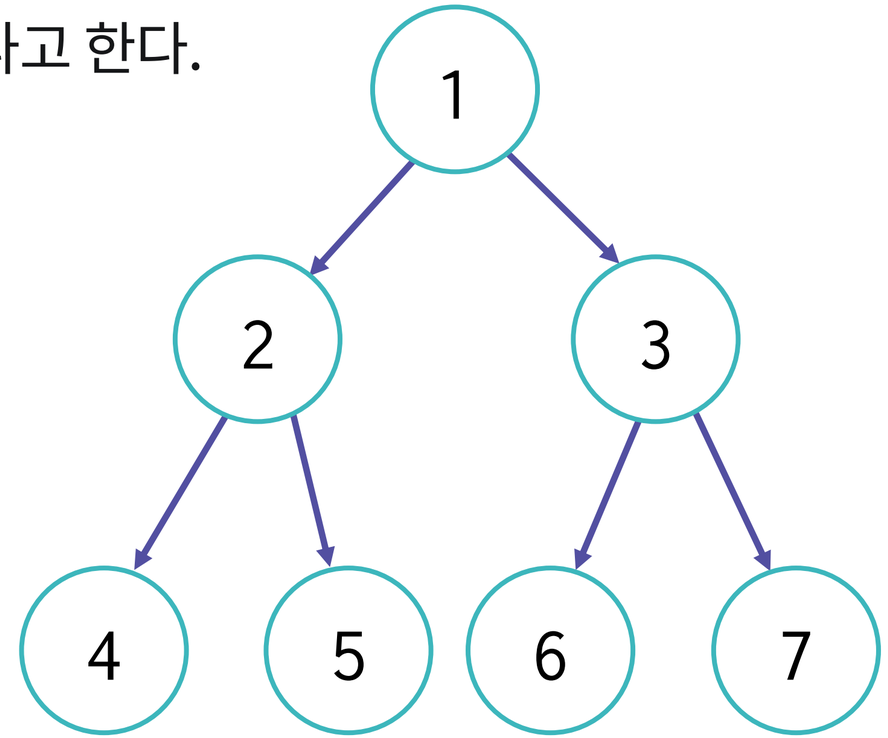
이진 탐색 트리 등 유용하게 활용되는 트리 중에는 대부분 **이진 트리**를 응용한 것이 많다.



포화 이진 트리

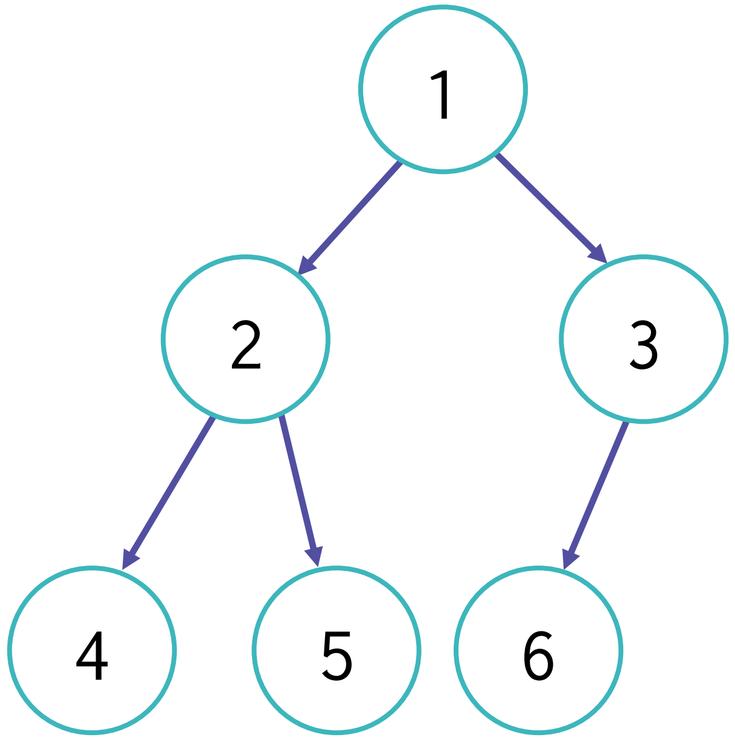
리프 노드를 제외한 모든 정점이 **항상** 자식을 **2개**씩 갖고 있으면서
모든 리프 노드의 **깊이**가 동일한 트리를 **포화 이진 트리**라고 한다.

포화 이진 트리의 높이를 h라고 할 때,
정점의 개수는 항상 $2^h - 1$개이다.



완전 이진 트리

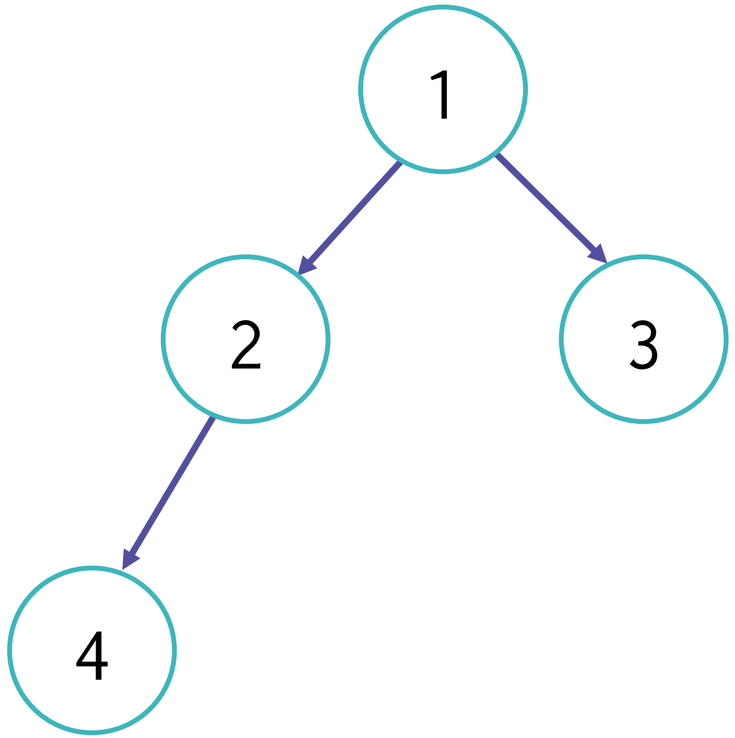
마지막 깊이를 제외하고 모든 정점이 **완전히** 채워져 있으며,
마지막 깊이의 정점들은 가능한 한 **왼쪽**에 있는 트리를
완전 이진 트리라고 한다.



완전 이진 트리

또는 **포화 이진 트리**에서 마지막 깊이의 정점이
오른쪽에서부터 **일부 제거**된 트리라고 볼 수 있다.

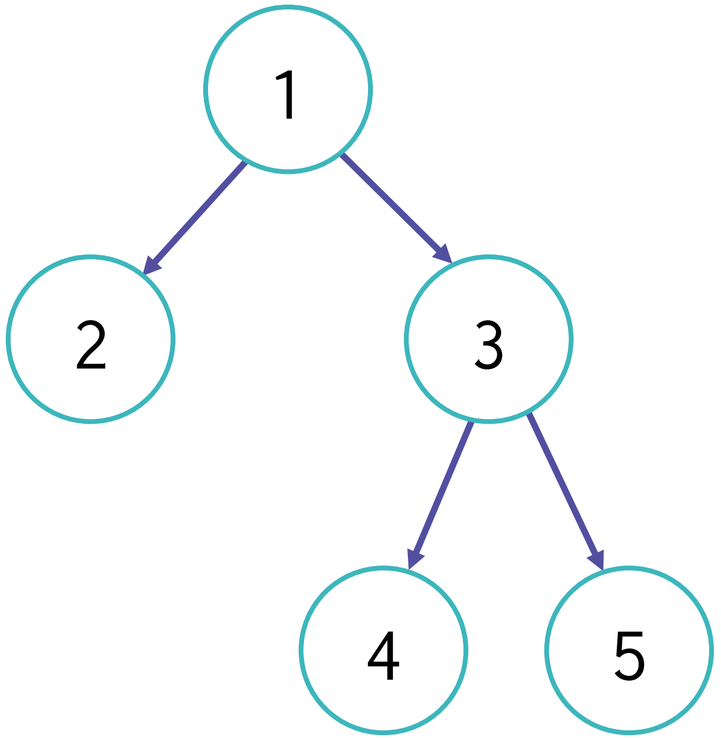
완전 이진 트리의 높이가 h일 때
정점의 개수는 2^{h-1}개 이상 $2^h - 1$개 이하이다.



정 이진 트리

정 이진 트리는 **리프 노드**를 제외한 모든 노드들이
두 개의 자식 노드를 갖고 있는 이진 트리이다.

즉, 모든 정점은 **0개 또는 2개**의 자식 노드를 갖는다.



02

트리의 표현 방법

트리의 표현 방법

이진 트리를 표현하는 방법을 알아보자.

트리의 표현 방법

Example

```
class TreeNode :
    def __init__(self) :
        self.left = None
        self.right = None
```

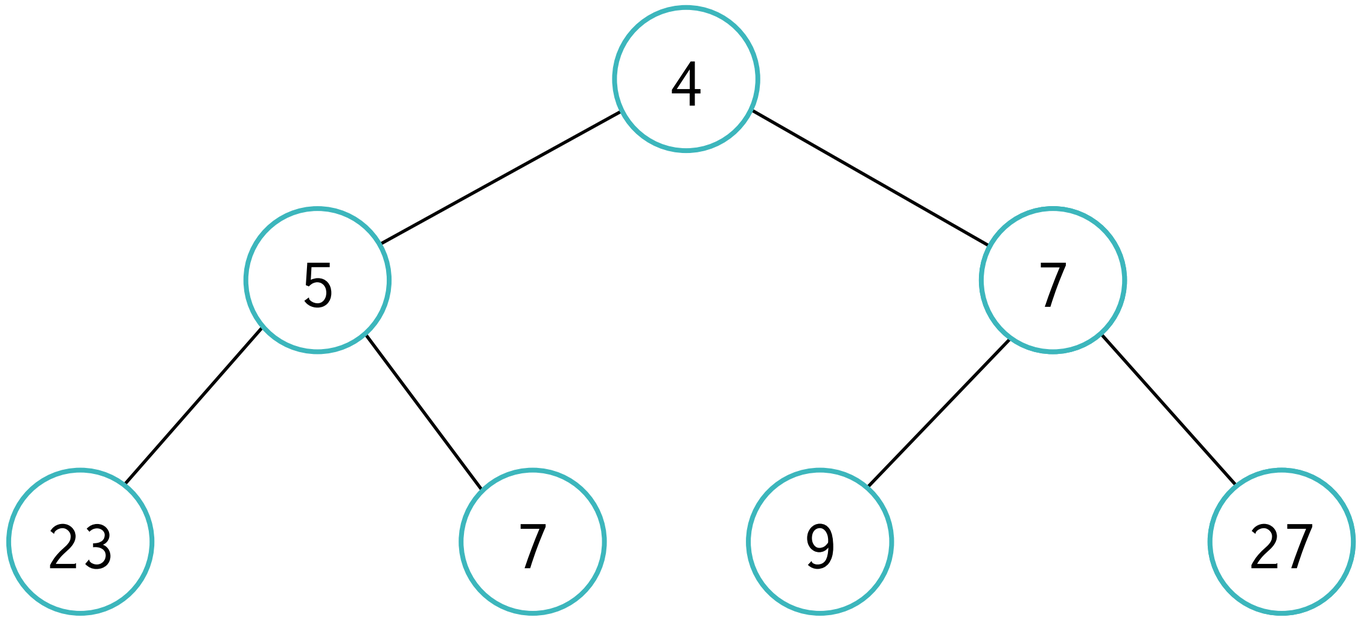
이진 트리의 각 노드는 **왼쪽** 또는 **오른쪽** 자식을 갖고 있으므로
위와 같은 클래스로 표현할 수 있다.

트리의 표현 방법

완전 이진 트리의 경우, **배열**을 이용하여 간단하게 구현이 가능하다.

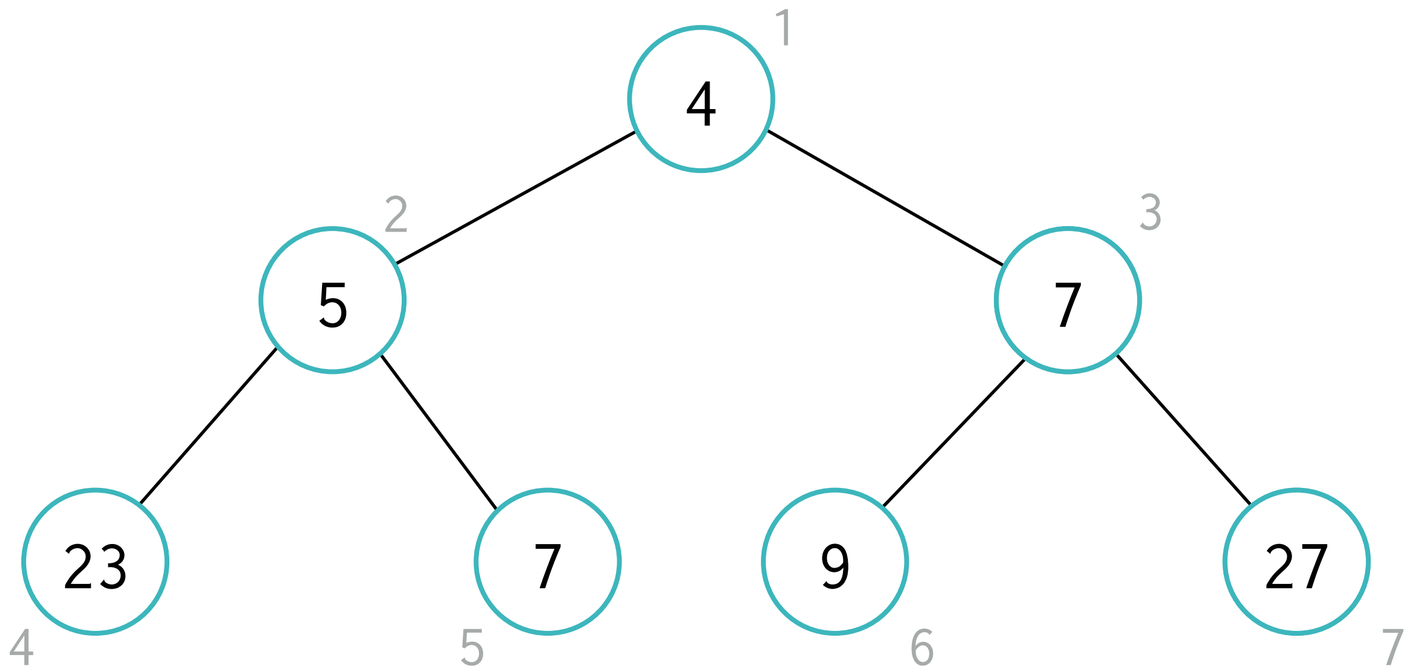
트리의 표현 방법

아래와 같은 완전 이진 트리가 존재할 때 각 정점에 순서대로 **번호**를 붙일 수 있다.



트리의 표현 방법

어떤 정점의 번호가 n이면
왼쪽 자식은 $2n$, 오른쪽 자식은 $2n + 1$이다.



트리의 표현 방법

따라서 **배열**로 **완전 이진 트리**를 표현할 수 있게 된다.

0	1	2	3	4	5	6	7
X	4	5	7	23	7	9	27

트리의 표현 방법

또, 트리는 **그래프의 일종**이므로 그래프를 표현할 때 사용하는
인접 행렬, 인접 리스트, 간선 리스트를 사용할 수도 있다.

(본 강의에서는 다루지 않습니다.)

03

트리 순회하기

트리 순회하기

트리 순회란

트리의 **모든 노드**를 방문하는 **순서**이다.

트리 순회하기

트리에 들어있는 **자료에 접근**하기 위해 순회를 해야 한다.

트리 순회하기

배열, 연결 리스트 등 **선형 구조**는 각 자료가 **순서**를 가지지만

비선형 구조에 해당하는 **트리**는 정해진 순서가 존재하지 않는다.

트리 순회하기

트리의 모든 노드를 방문하는 순서는 크게 두 가지 종류가 있다.

DFS(깊이 우선 탐색)과 **BFS(너비 우선 탐색)**이다.

(그래프의 순회 방법과 동일하다.)

트리 순회하기 - 깊이 우선 탐색(DFS)

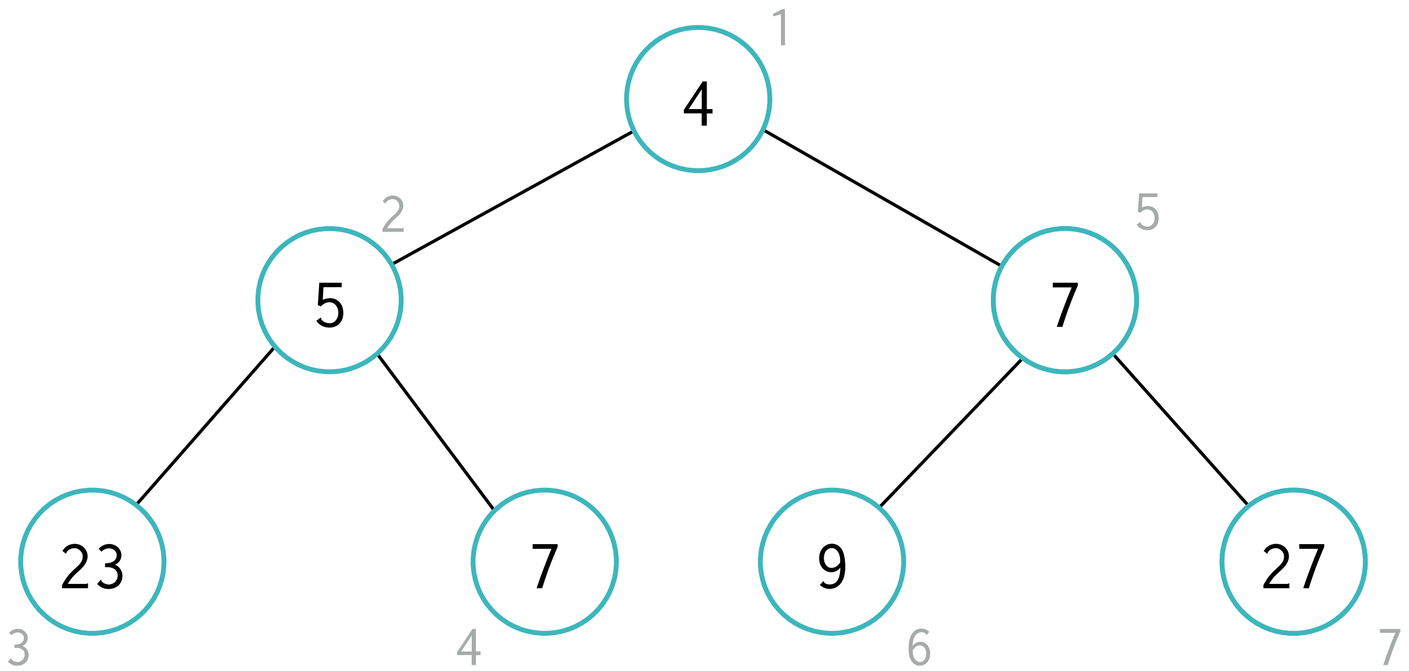
트리의 DFS에는 세 가지 종류가 있다.

1. 전위 순회
2. 중위 순회
3. 후위 순회

03 트리 순회하기

트리 순회하기 - 전위 순회

전위 순회의 경우 정점을 방문하는 순서는 다음과 같다.

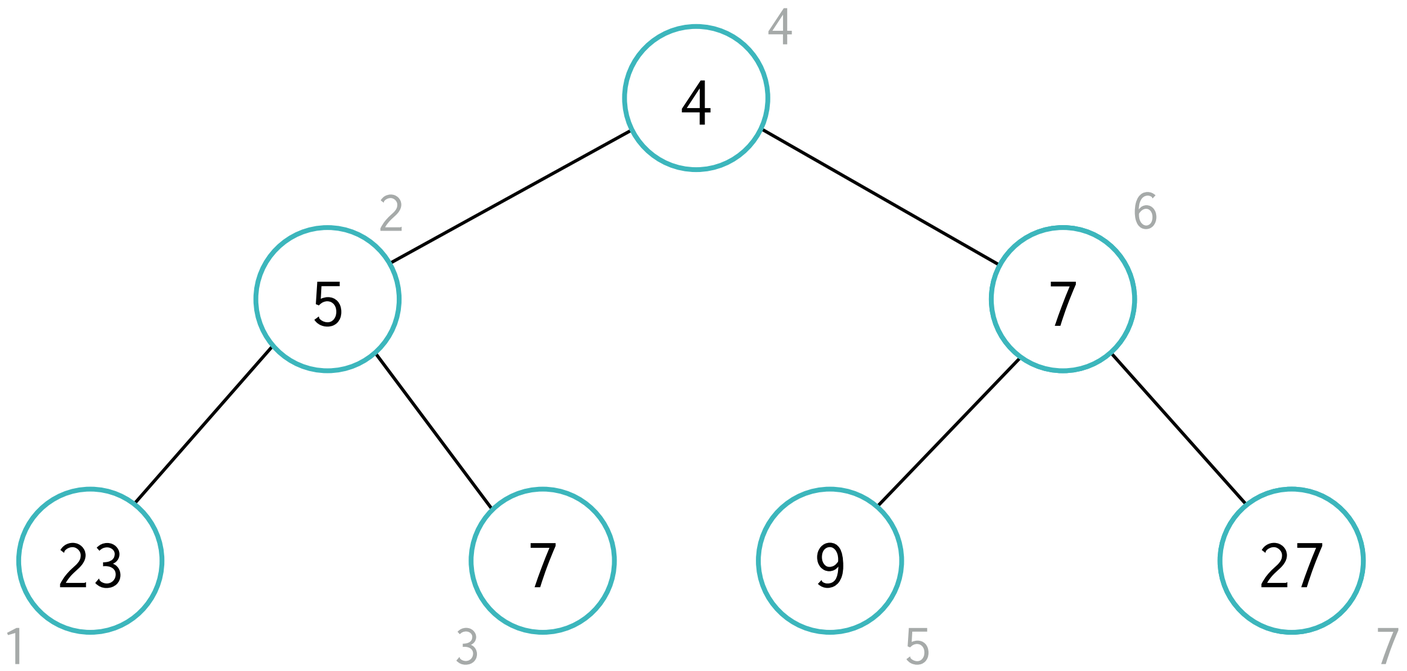


/* elice */

03 트리 순회하기

트리 순회하기 - 중위 순회

중위 순회의 경우 정점을 방문하는 순서는 다음과 같다.

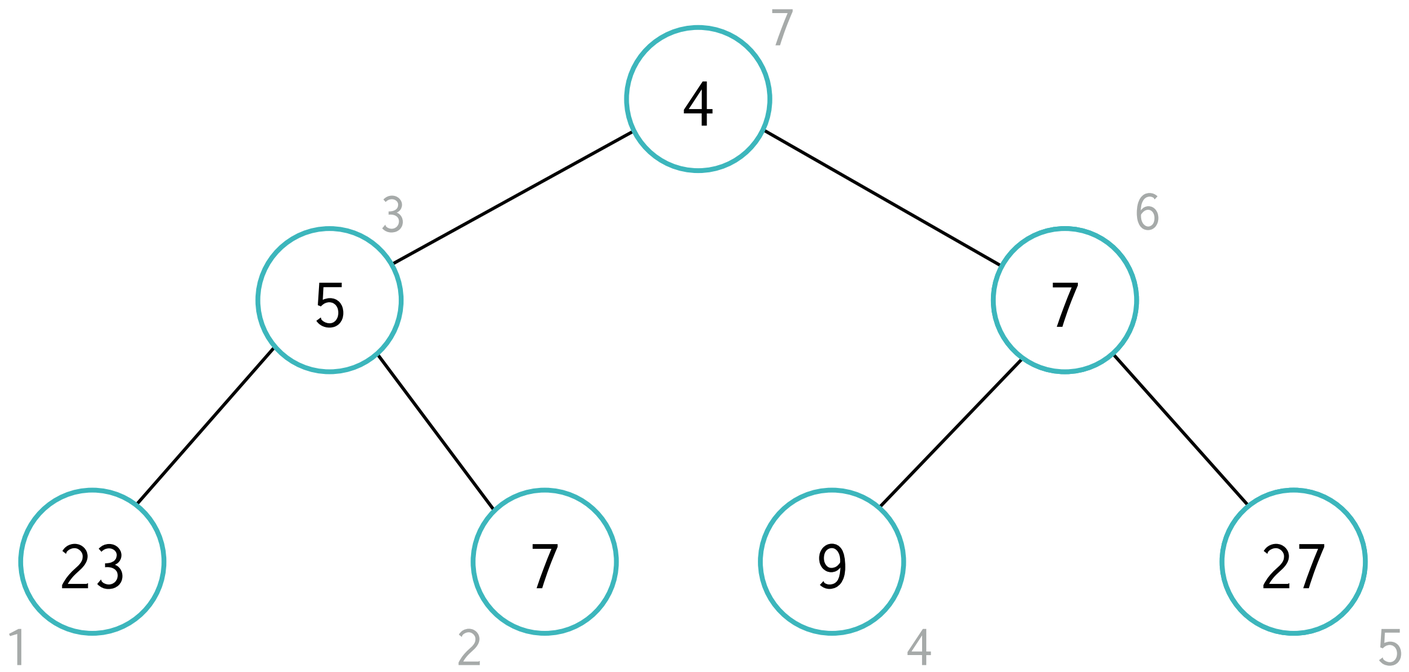


/* elice */

03 트리 순회하기

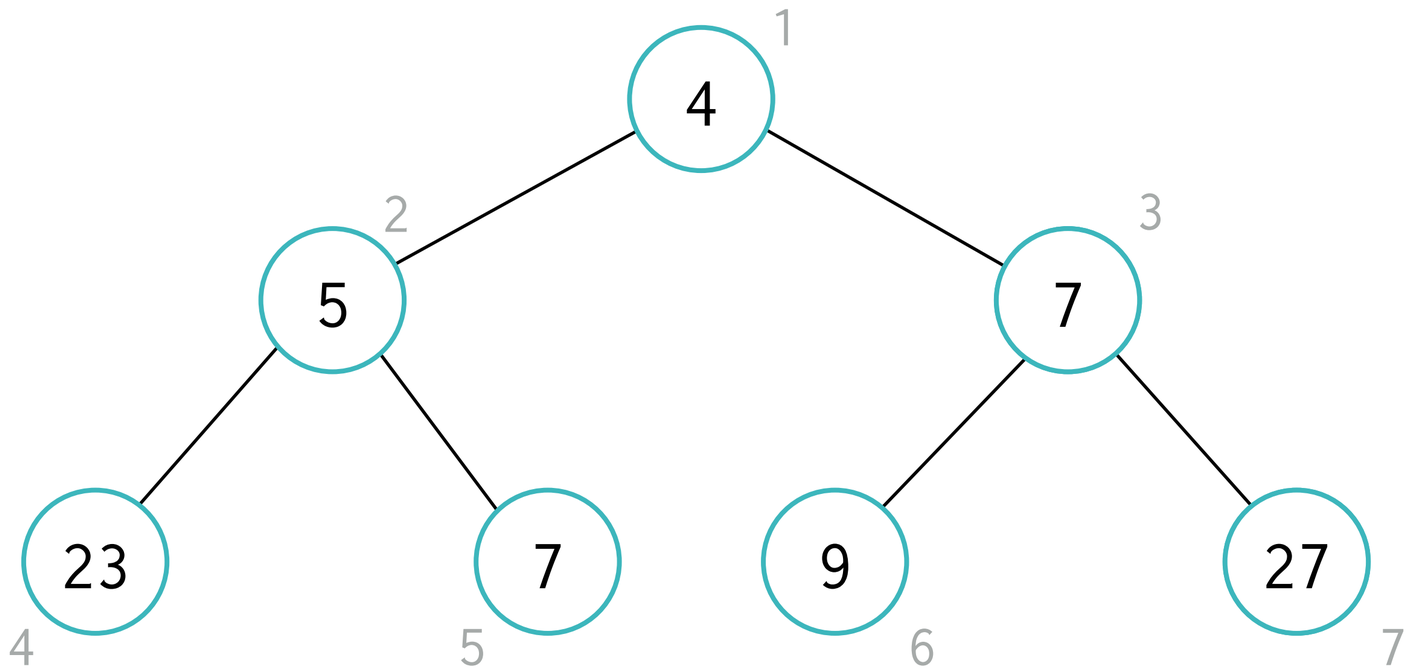
트리 순회하기 - 후위 순회

후위 순회의 경우 정점을 방문하는 순서는 다음과 같다.



트리 순회하기 - 너비 우선 탐색(BFS)

너비 우선 탐색의 경우 정점을 방문하는 순서는 다음과 같다.



트리 순회하기 - DFS

DFS는 **재귀 호출**을 사용하는 알고리즘으로,

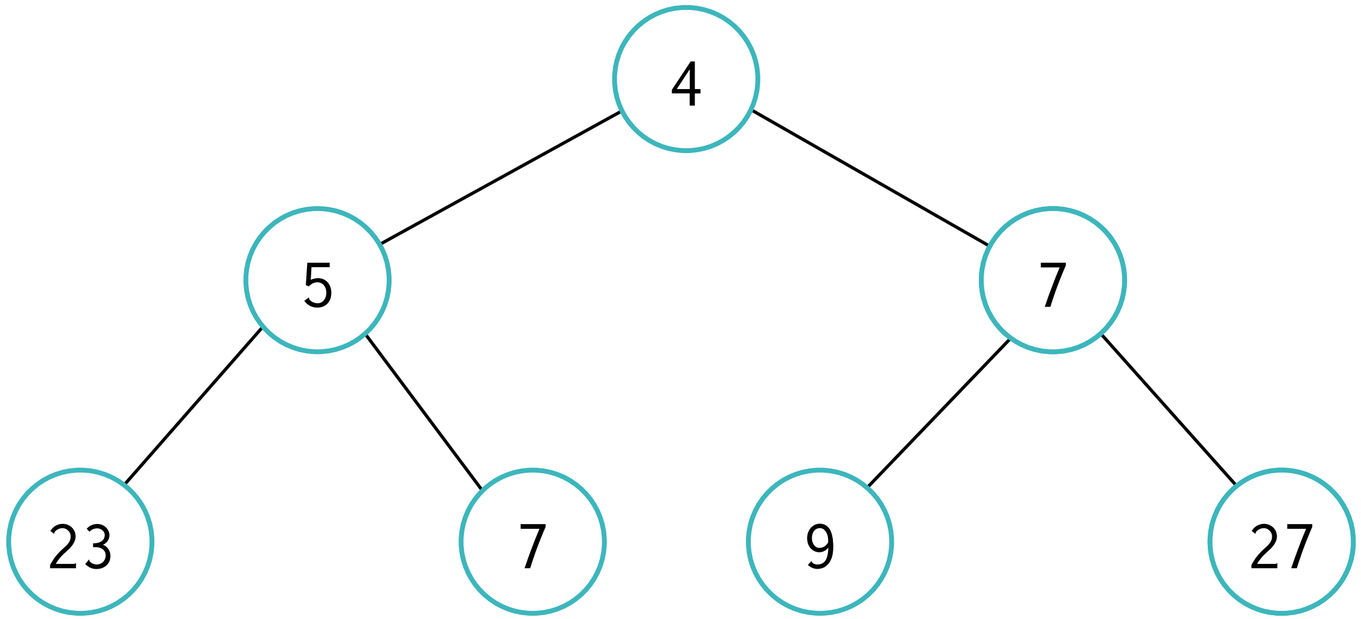
DFS를 이해하기 위해서는 트리의 **재귀적 특성**을 알아야 한다.

재귀 호출은 **스택**의 특성을 이용하므로 **스택**을 이용한다고 볼 수 있다.

(우아한 방법은 아니지만, DFS를 구현할 때 재귀 호출을 사용하지 않고 스택으로 구현할 수도 있다.)

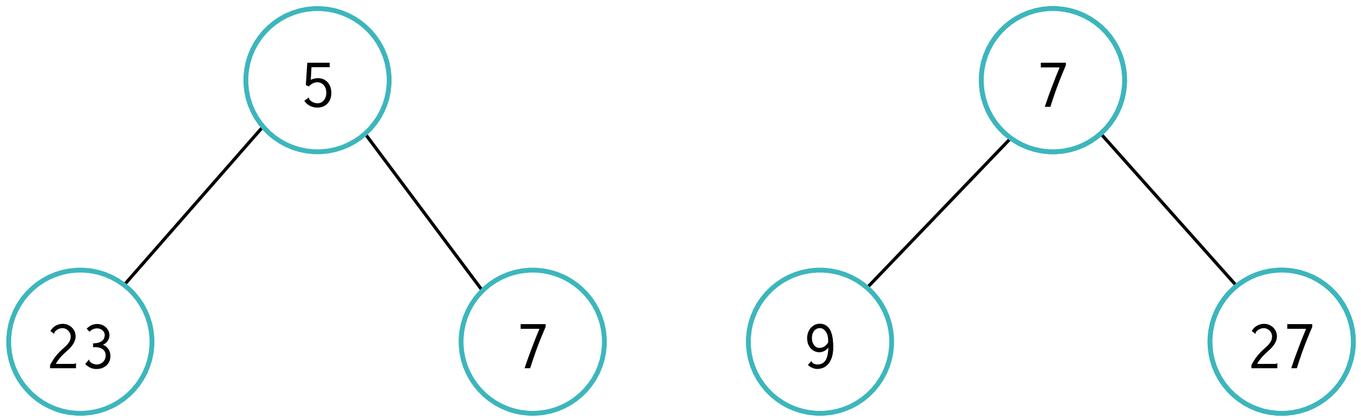
트리 순회하기 - DFS

아래 그림은 정점이 **7개**인 트리이다.



트리 순회하기 - DFS

이 트리에서 루트 노드를 제외하면 두 개의 **작은 트리**가 만들어진다.
이와 같이 기존 트리에서 하위의 트리 구조를 **서브 트리**라고 한다.



트리 순회하기 - DFS

DFS의 세 가지 방법은 **정점을 언제 방문하는지**에 따라 순서가 달라지며
재귀 호출을 이용한다는 기본적인 개념 자체는 동일하다.

전위 순회 : **루트 방문** → **왼쪽 서브 트리 순회** → **오른쪽 서브 트리 순회**

중위 순회 : **왼쪽 서브 트리 순회** → **루트 방문** → **오른쪽 서브 트리 순회**

후위 순회 : **왼쪽 서브 트리 순회** → **오른쪽 서브 트리 순회** → **루트 방문**

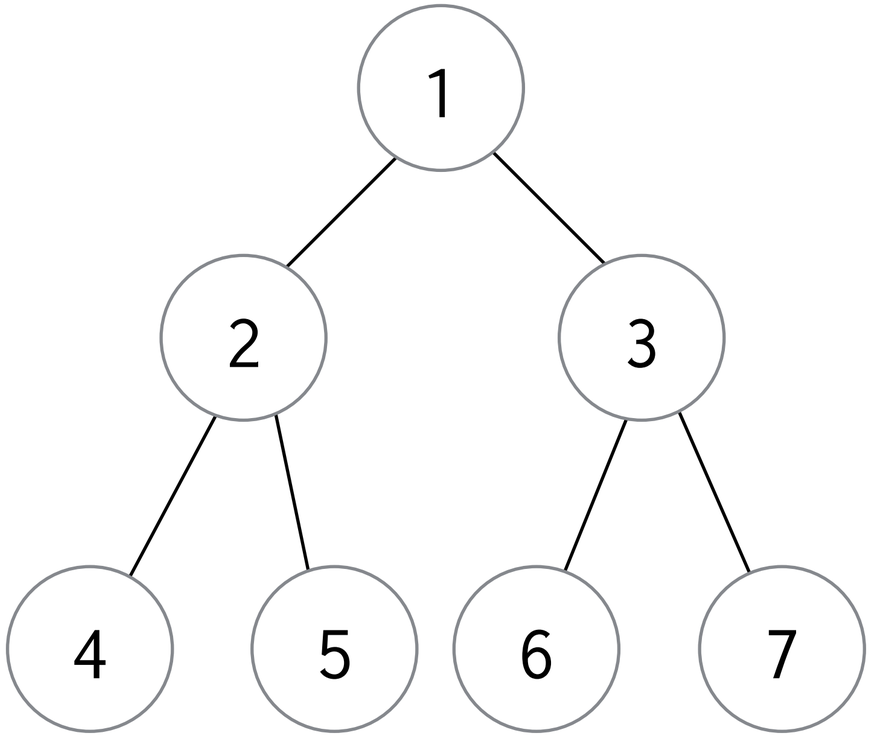
트리 순회하기 - DFS

전체 트리를 **순회**하기 위해
서브 트리를 **순회**한다.

순회를 위해 순회한다 → **재귀 호출**

트리 순회하기 - 전위 순회

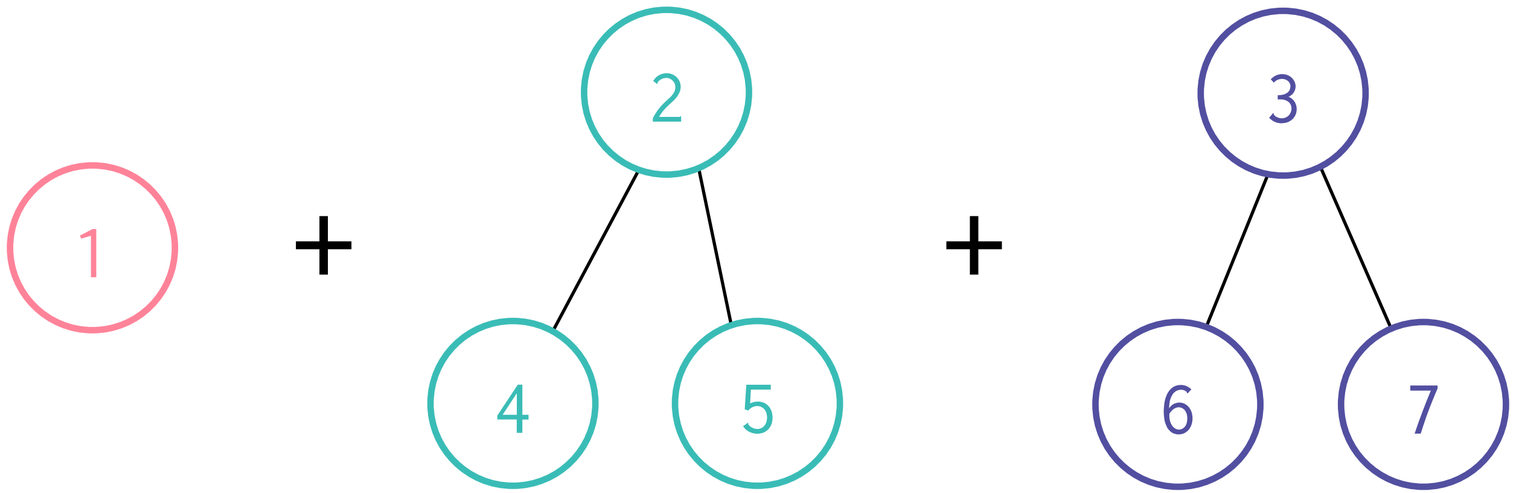
이 트리를 **전위 순회**한 결과는?



03 트리 순회하기

트리 순회하기 - 전위 순회

루트노드 왼쪽 전위 순회 오른쪽 전위 순회

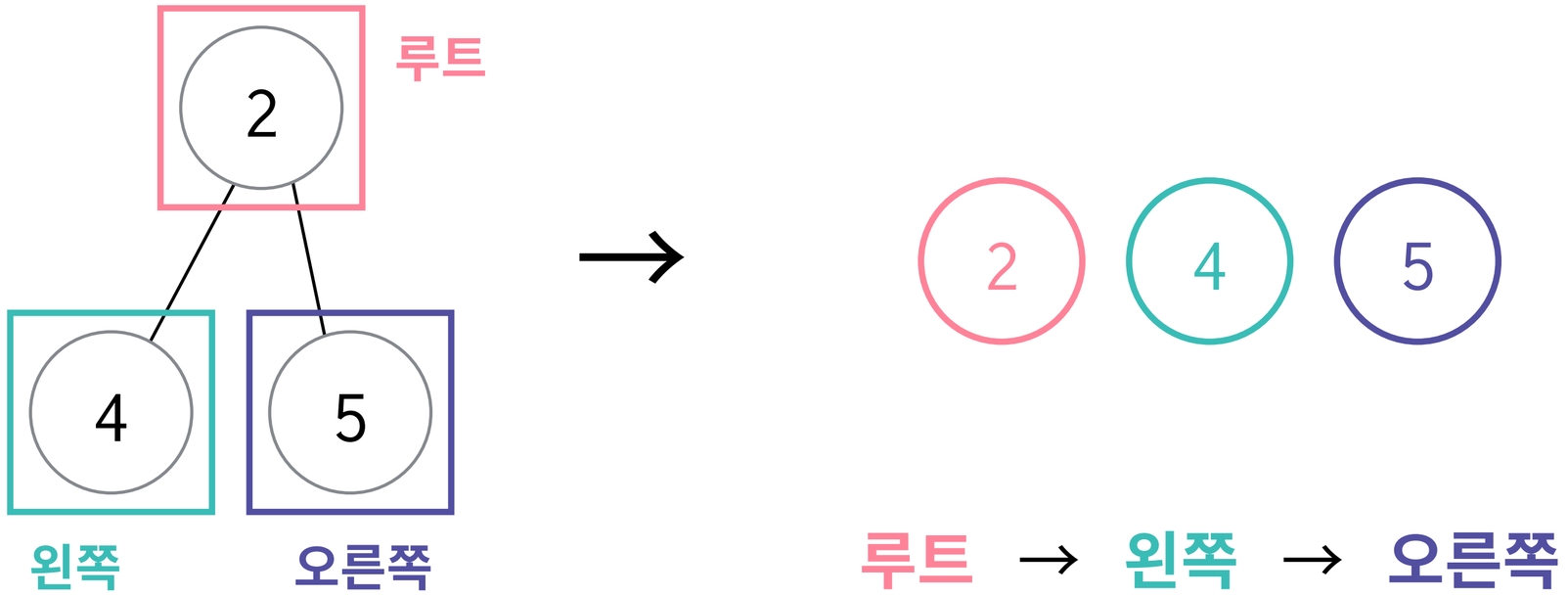


03 트리 순회하기

트리 순회하기 - 전위 순회

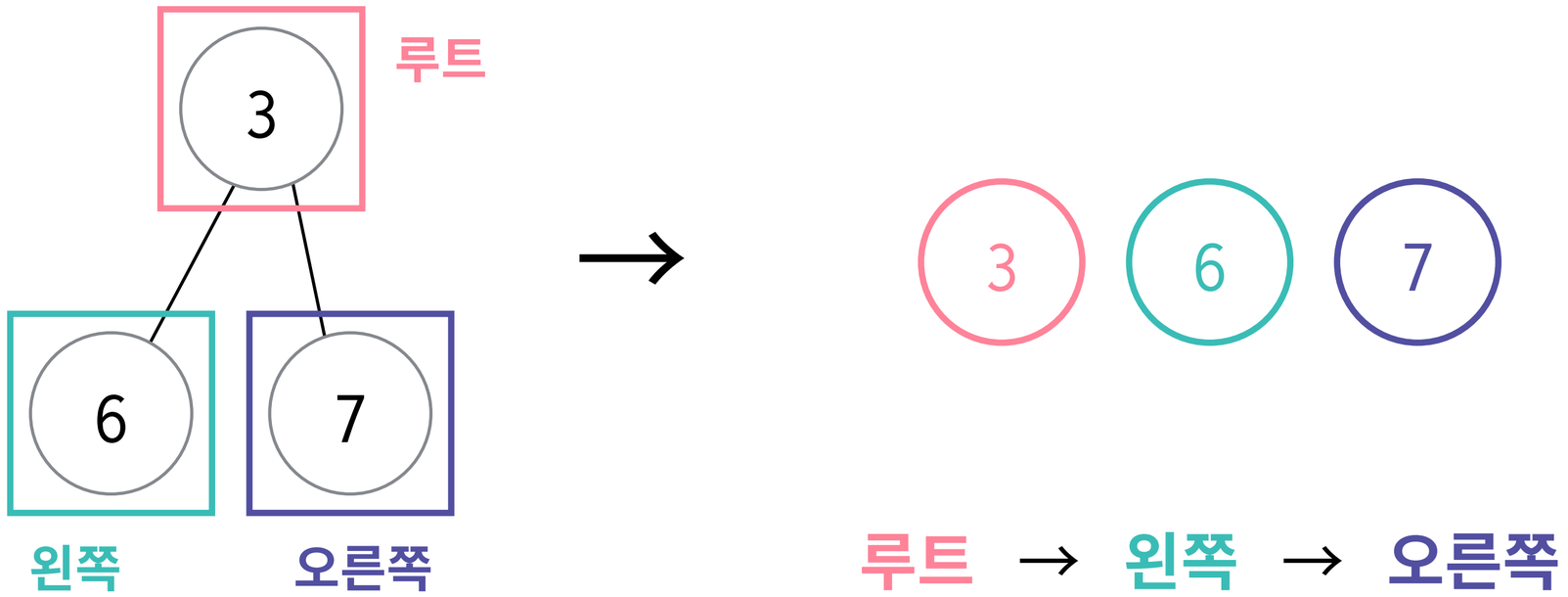
트리 순회하기 - 전위 순회

서브 트리의 전위 순회



트리 순회하기 - 전위 순회

서브 트리의 전위 순회

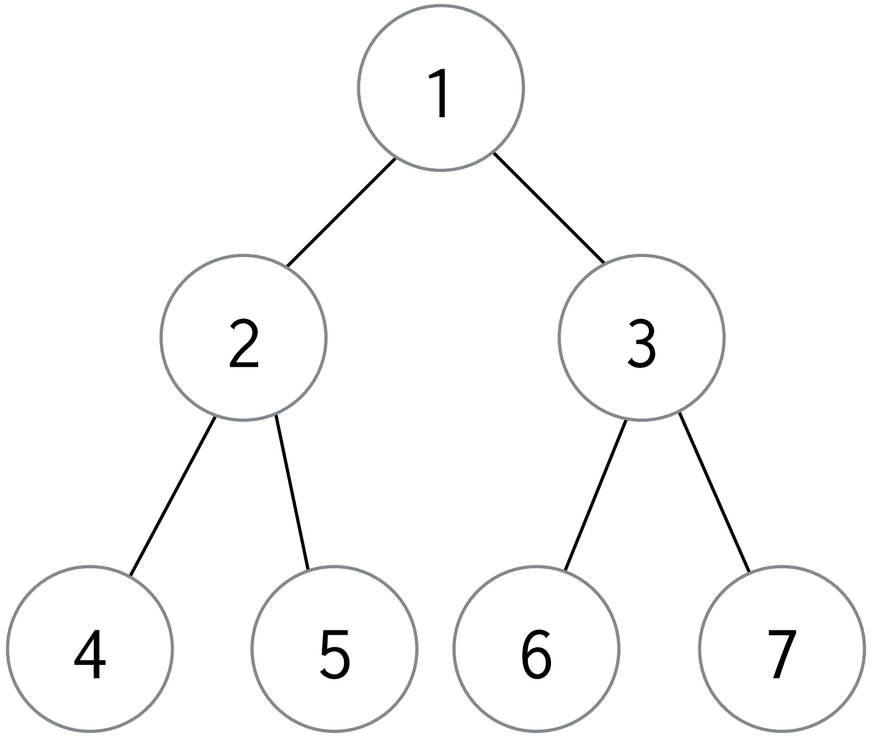


트리 순회하기 - 전위 순회

전위 순회의 최종 결과는 아래와 같다.

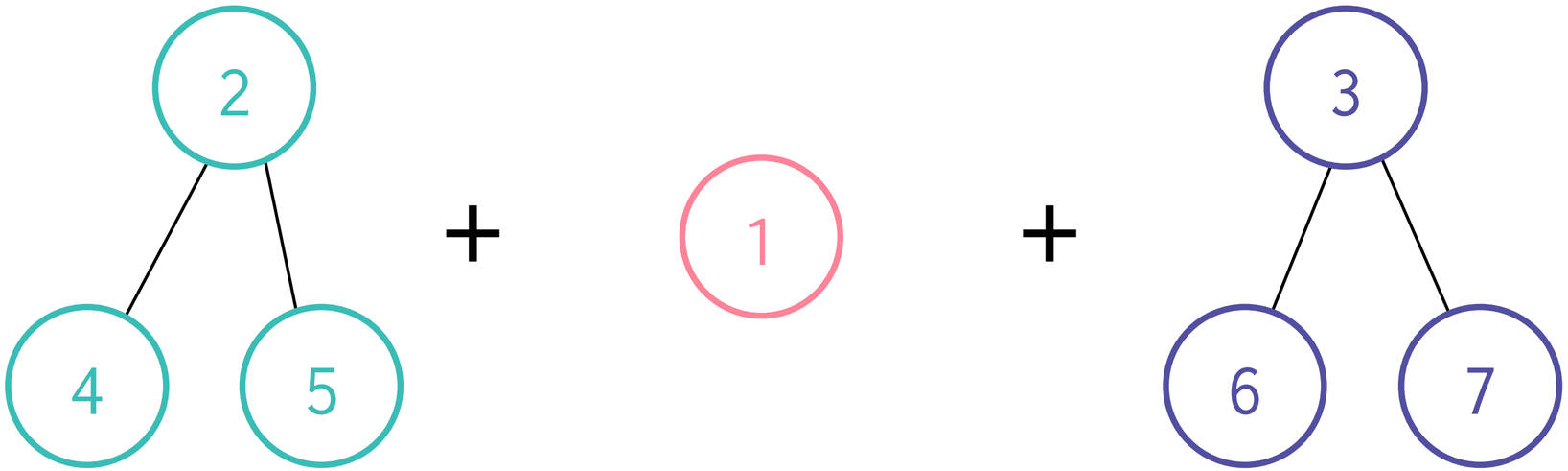
트리 순회하기 - 중위 순회

이 트리를 **중위 순회**한 결과는?



트리 순회하기 - 중위 순회

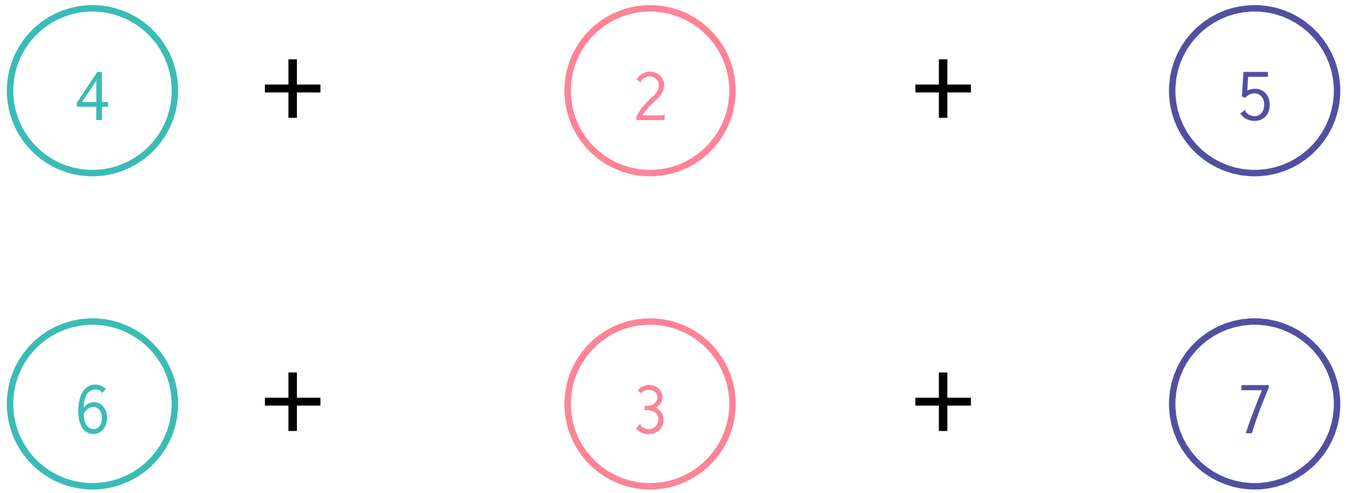
왼쪽 중위 순회 + 루트노드 + 오른쪽 중위 순회



03 트리 순회하기

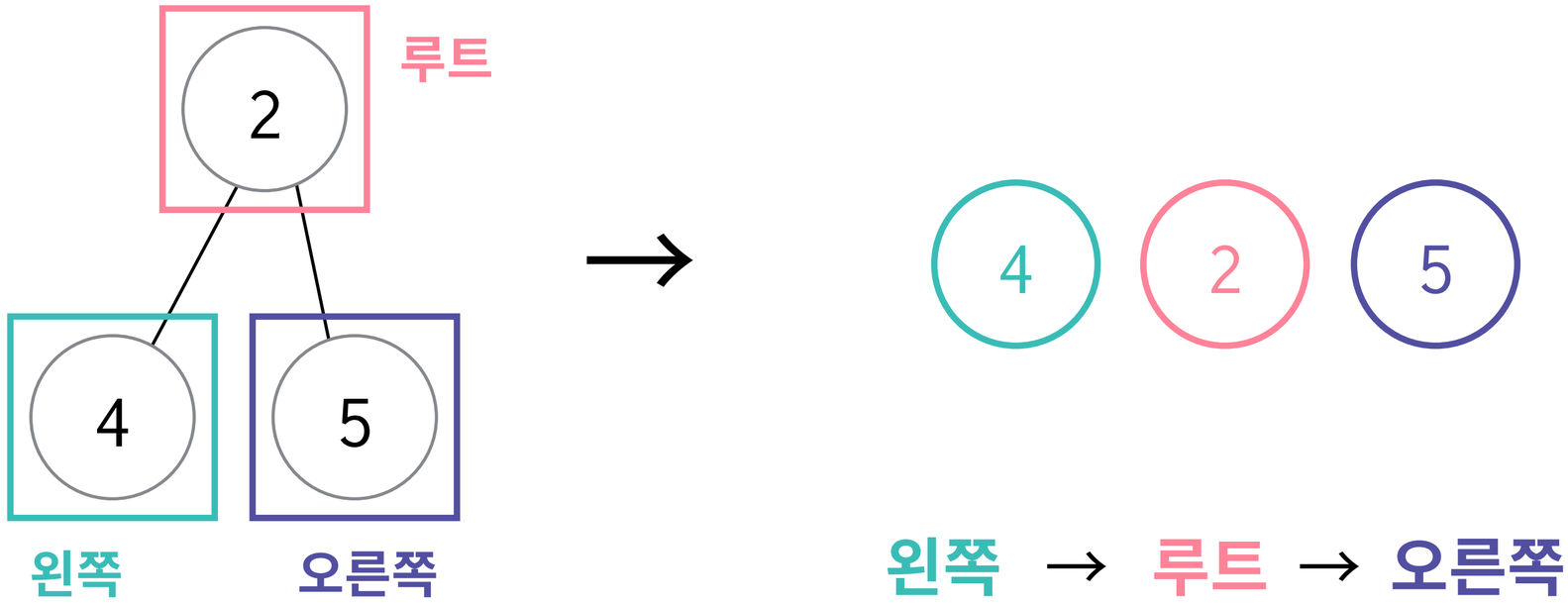
트리 순회하기 - 전위 순회

왼쪽 중위 순회 루트노드 오른쪽 중위 순회



트리 순회하기 - 중위 순회

서브 트리의 중위 순회

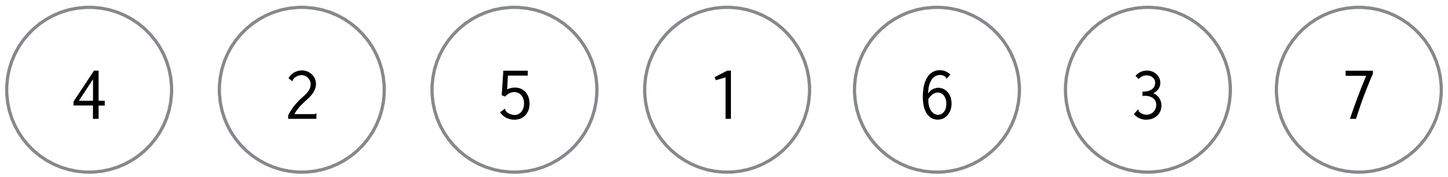


트리 순회하기 - 중위 순회

서브 트리의 중위 순회

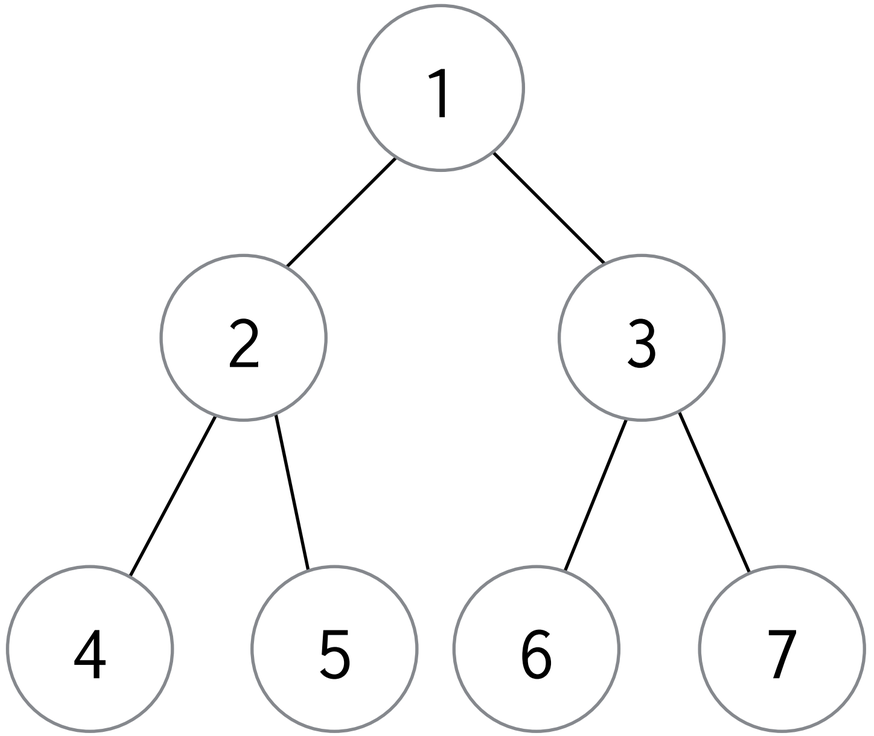
트리 순회하기 - 중위 순회

중위 순회의 최종 결과는 아래와 같다.



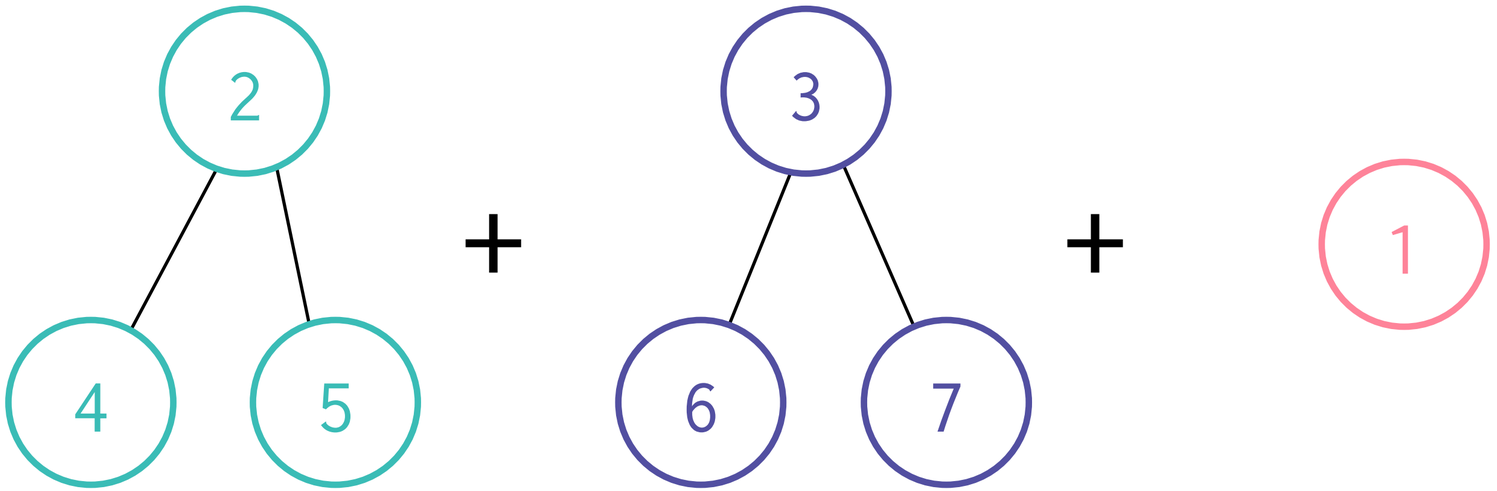
트리 순회하기 - 후위 순회

이 트리를 **후위 순회**한 결과는?



트리 순회하기 - 후위 순회

왼쪽 후위 순회 + 오른쪽 후위 순회 + 루트노드



03 트리 순회하기

트리 순회하기 - 후위 순회

왼쪽 후위 순회 오른쪽 후위 순회 루트노드

트리 순회하기 - 후위 순회

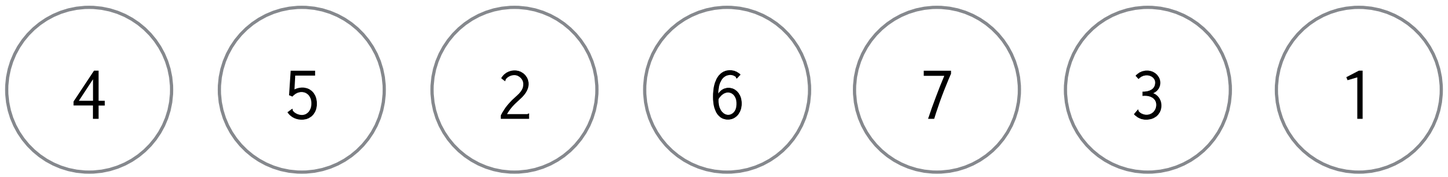
서브 트리의 후위 순회

트리 순회하기 - 후위 순회

서브 트리의 후위 순회

트리 순회하기 - 후위 순회

후위 순회의 최종 결과는 아래와 같다.



트리 순회하기 - BFS

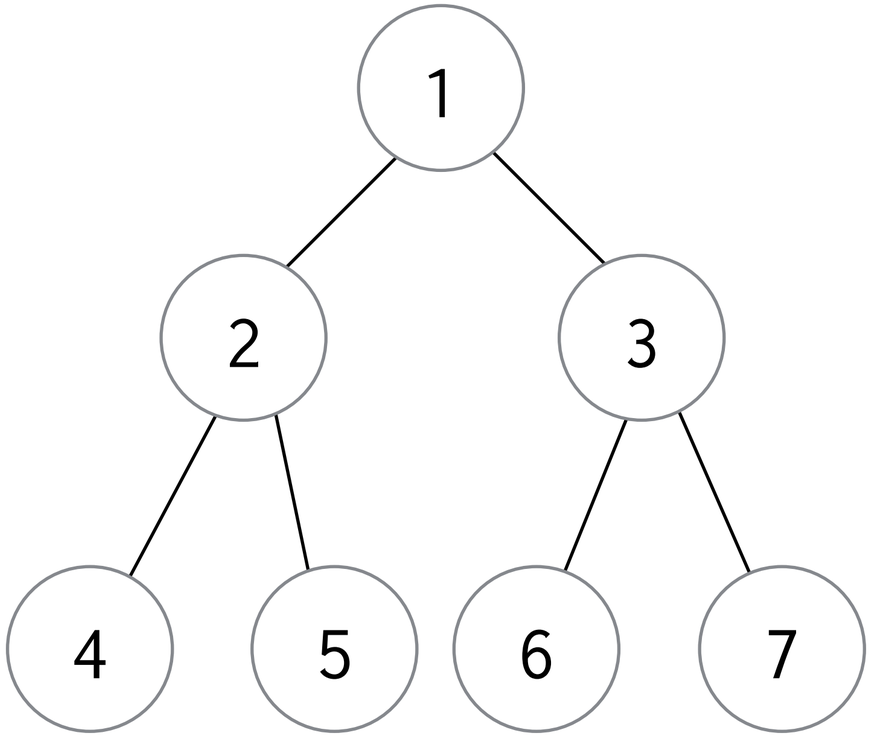
트리(그래프)의 **BFS**는 **큐** 자료구조를 이용하여 구현한다.

현재 정점과 이웃한 정점일수록 먼저 방문해야 하므로

FIFO 자료구조인 **큐**를 이용해야 한다.

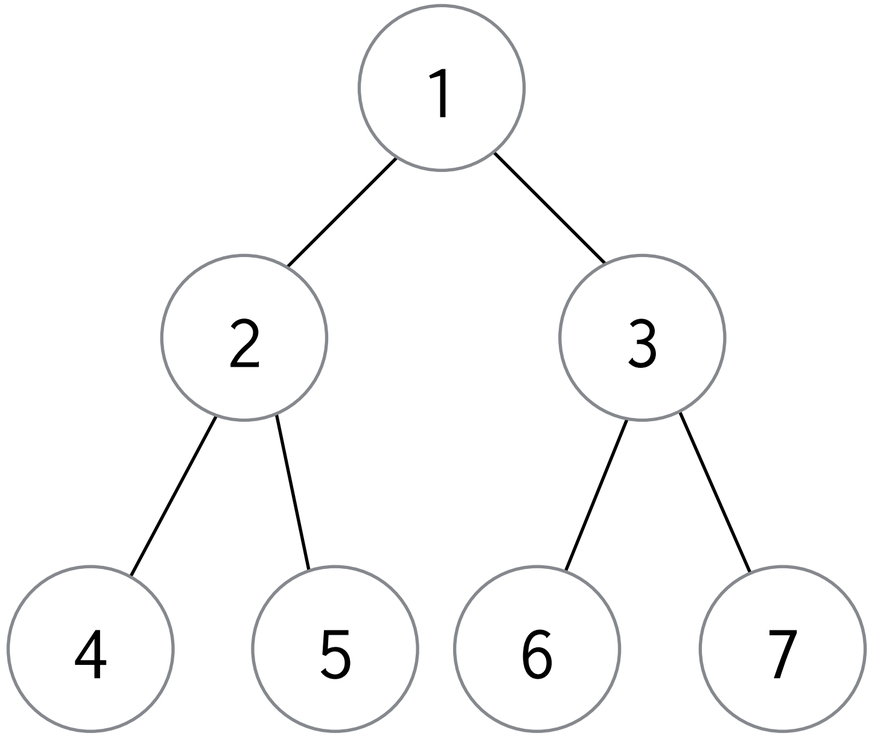
트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?



트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

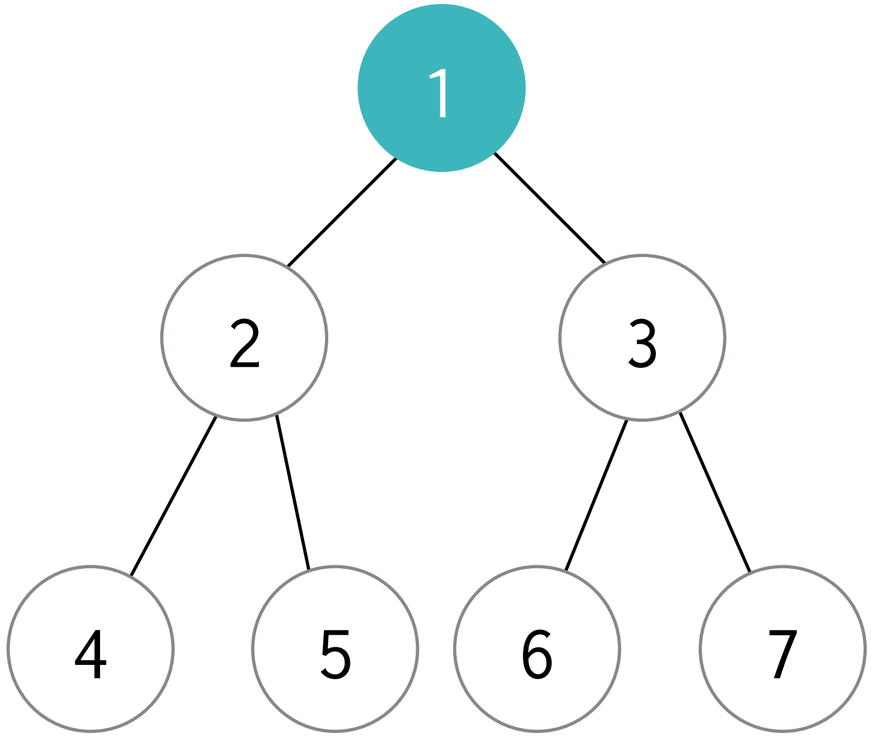


queue = [1]

방문한 노드 :

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

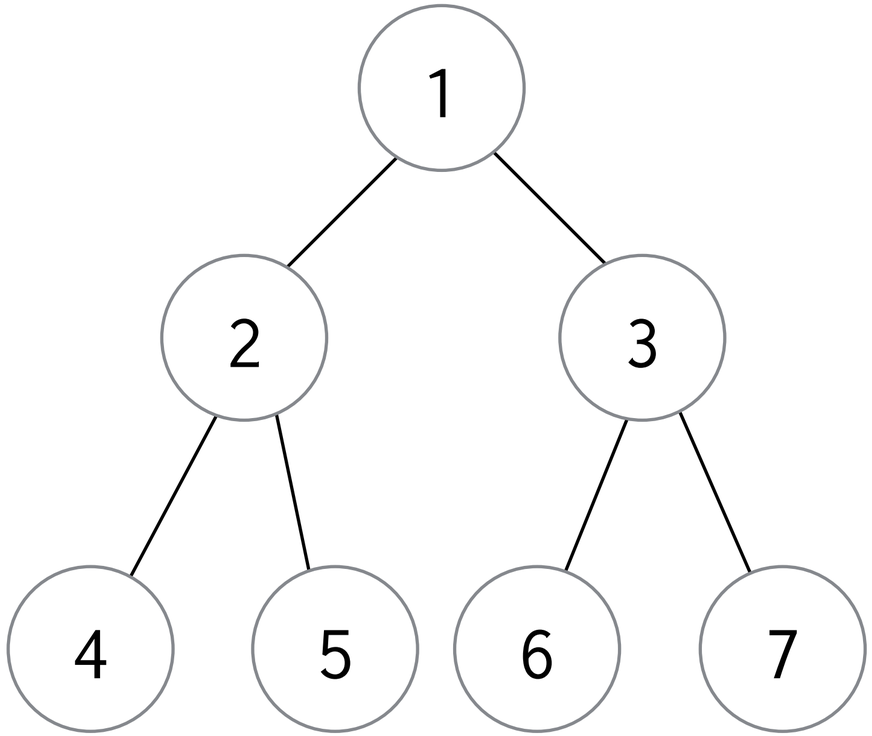


queue = [1]

방문한 노드 : 1

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

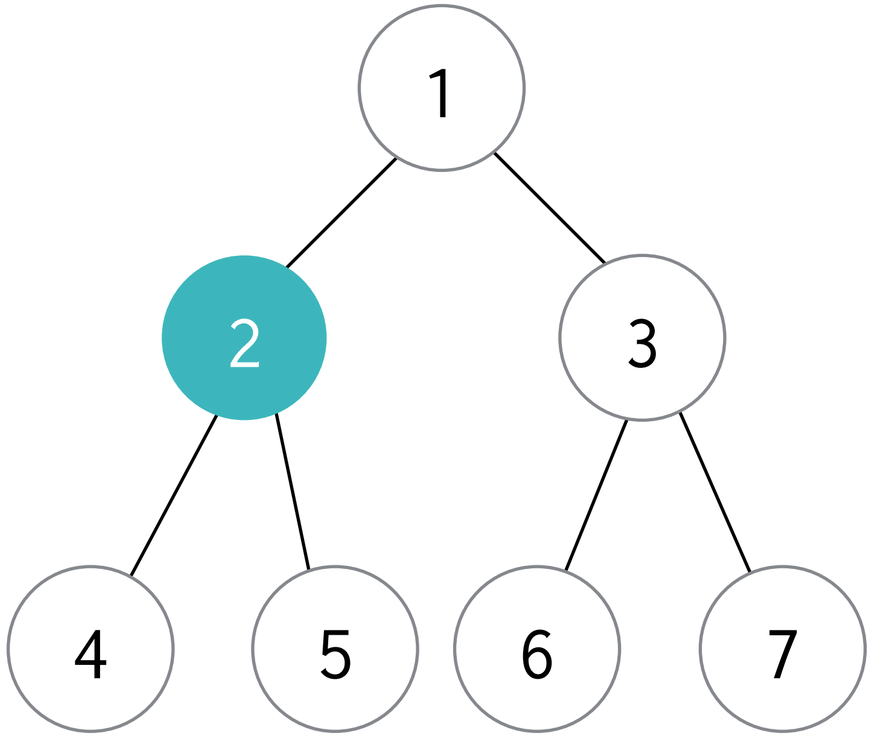


queue = [2, 3]

방문한 노드 : 1

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

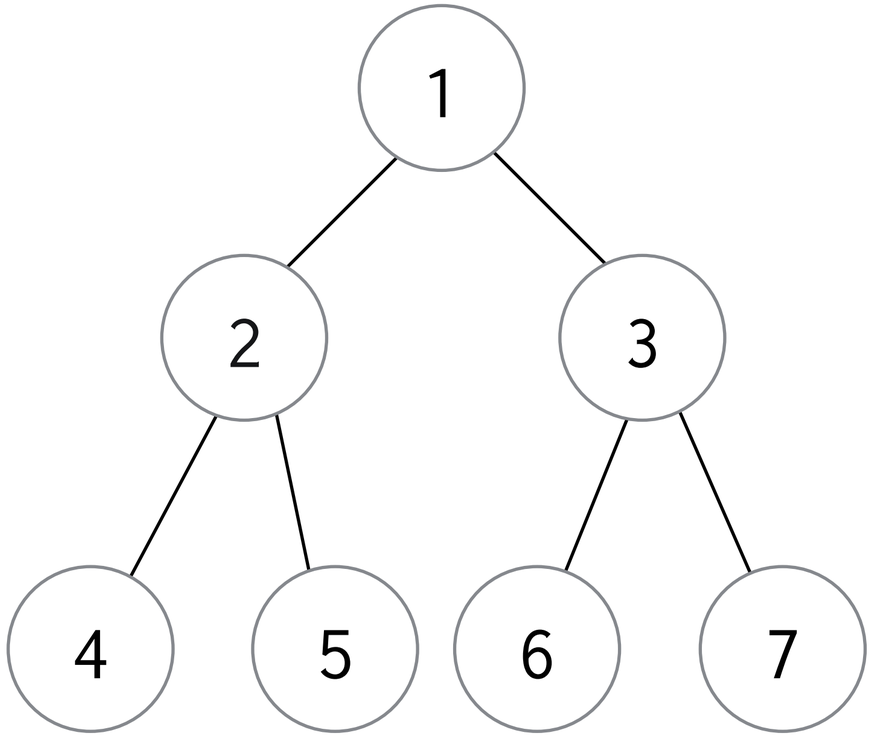


queue = [2, 3]

방문한 노드 : 1, 2

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

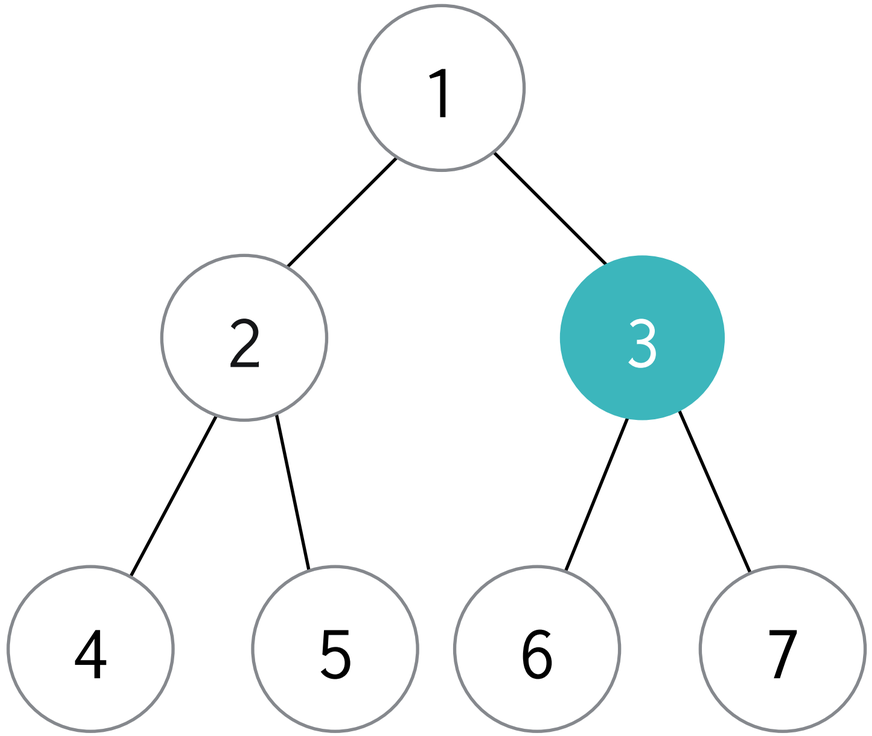


queue = [3, 4, 5]

방문한 노드 : 1, 2

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

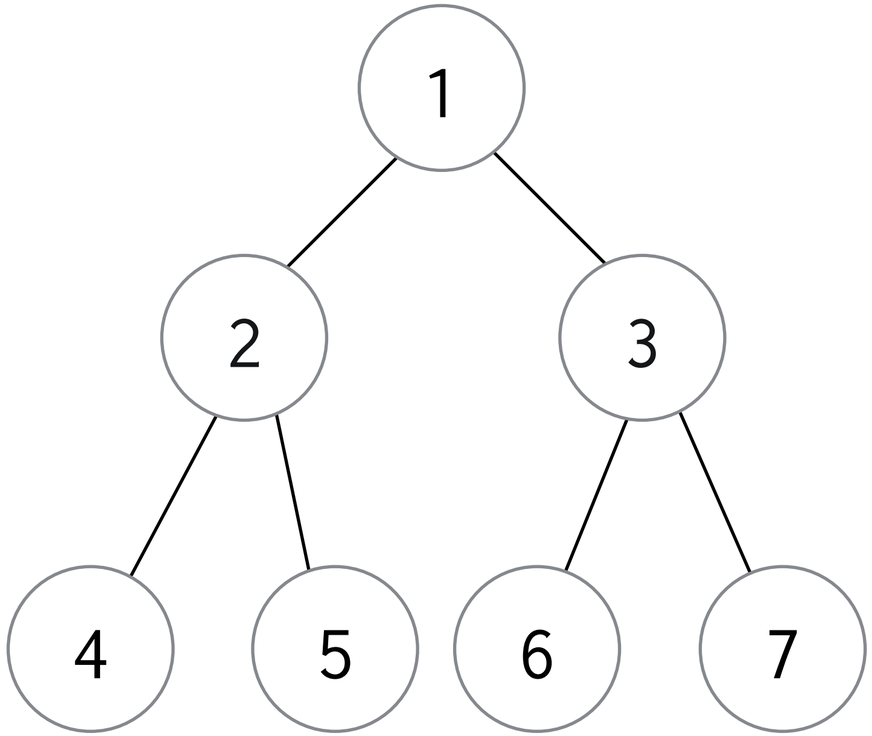


queue = [3, 4, 5]

방문한 노드 : 1, 2, 3

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

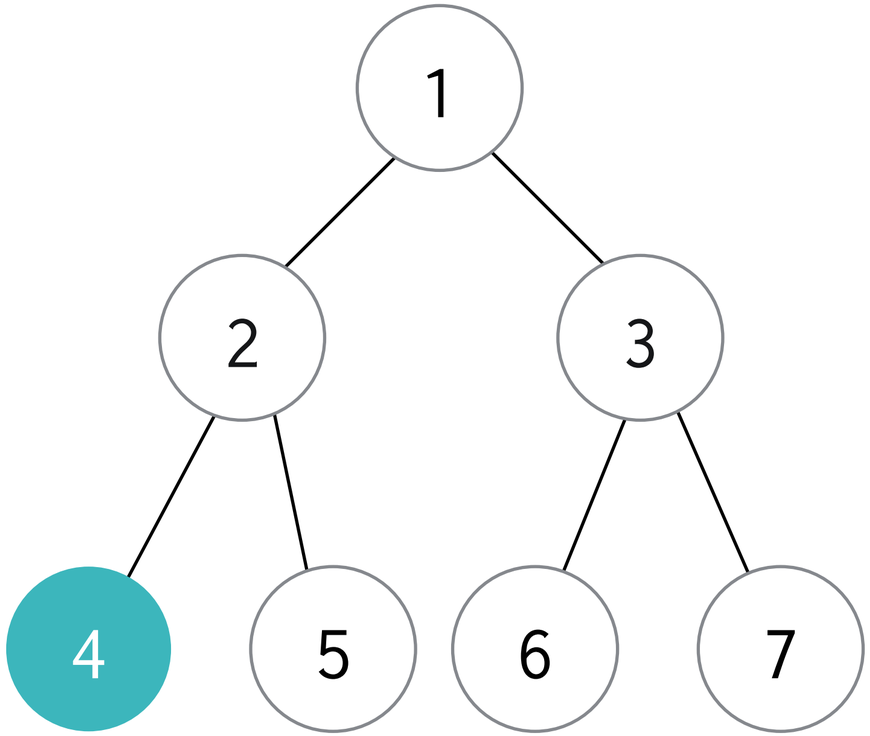


queue = [4, 5, 6, 7]

방문한 노드 : 1, 2, 3

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

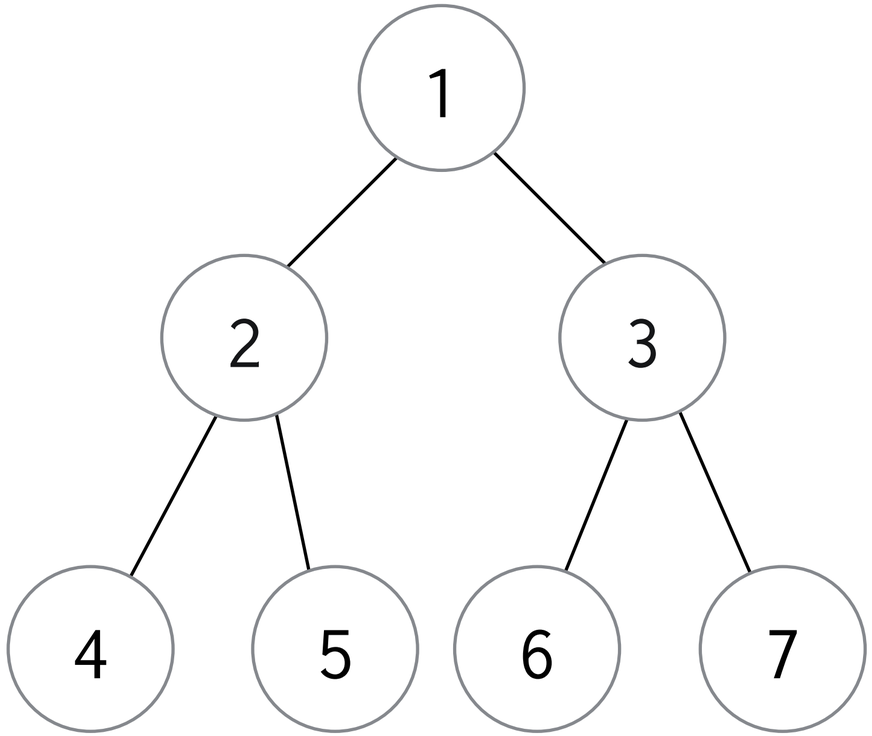


queue = [4, 5, 6, 7]

방문한 노드 : 1, 2, 3, 4

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

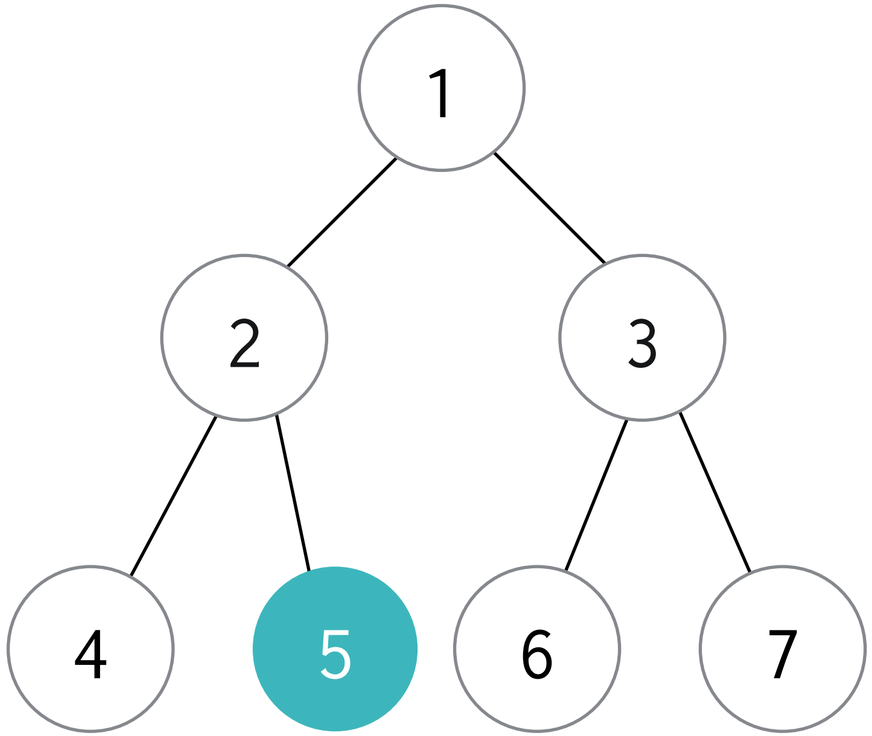


queue = [5, 6, 7]

방문한 노드 : 1, 2, 3, 4

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

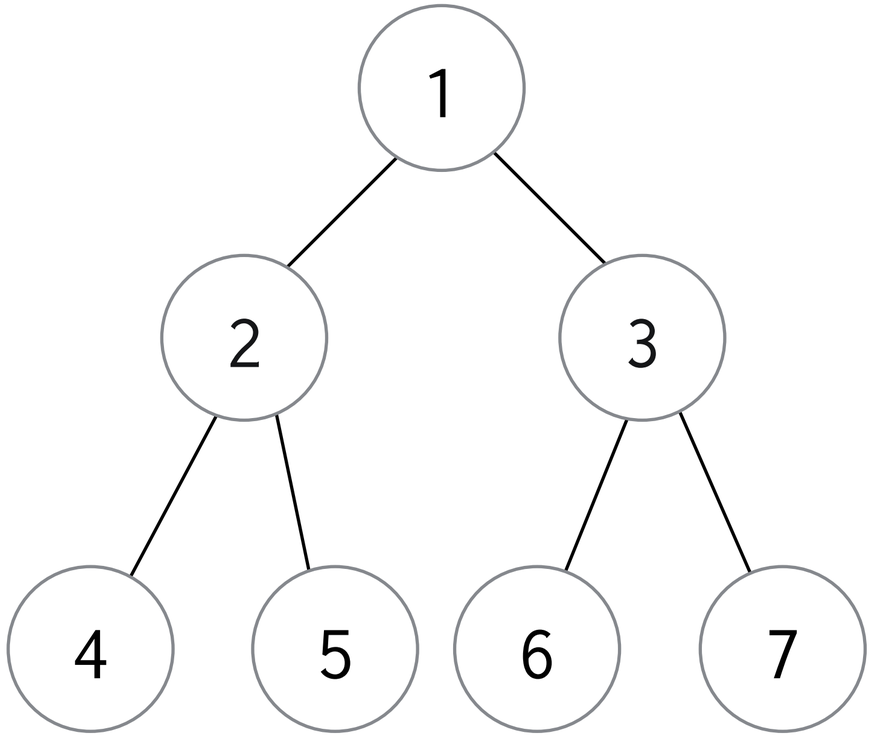


queue = [5, 6, 7]

방문한 노드 : 1, 2, 3, 4, 5

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

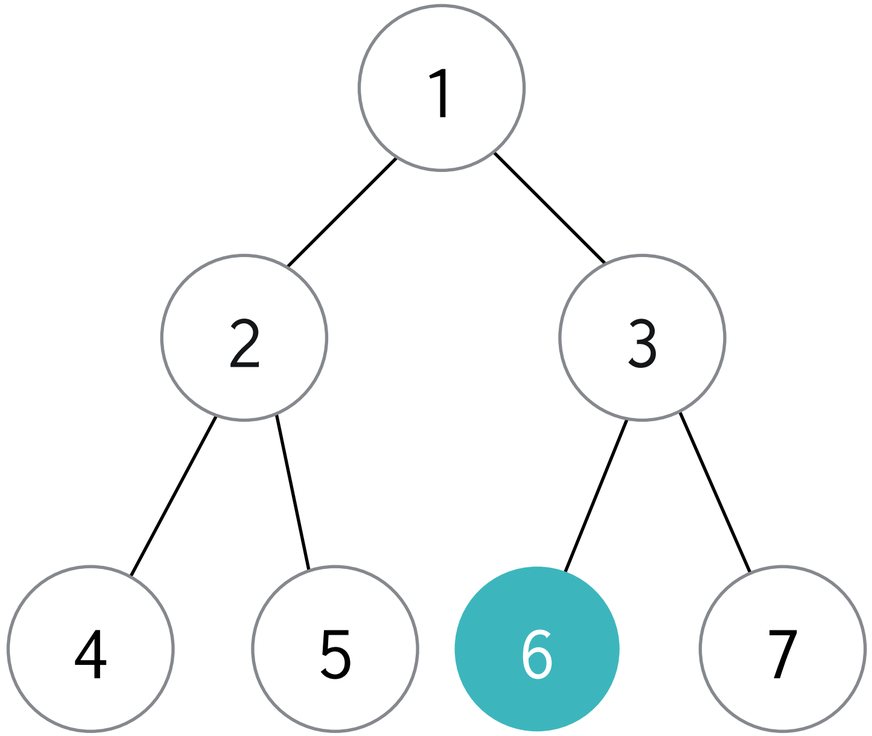


queue = [6, 7]

방문한 노드 : 1, 2, 3, 4, 5

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

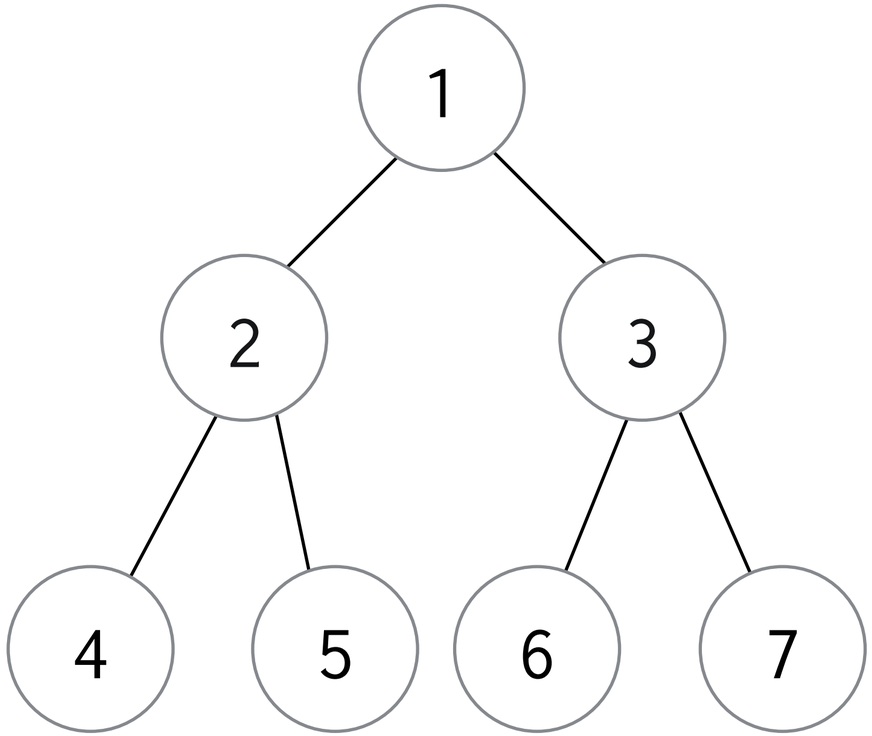


queue = [6, 7]

방문한 노드 : 1, 2, 3, 4, 5, 6

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

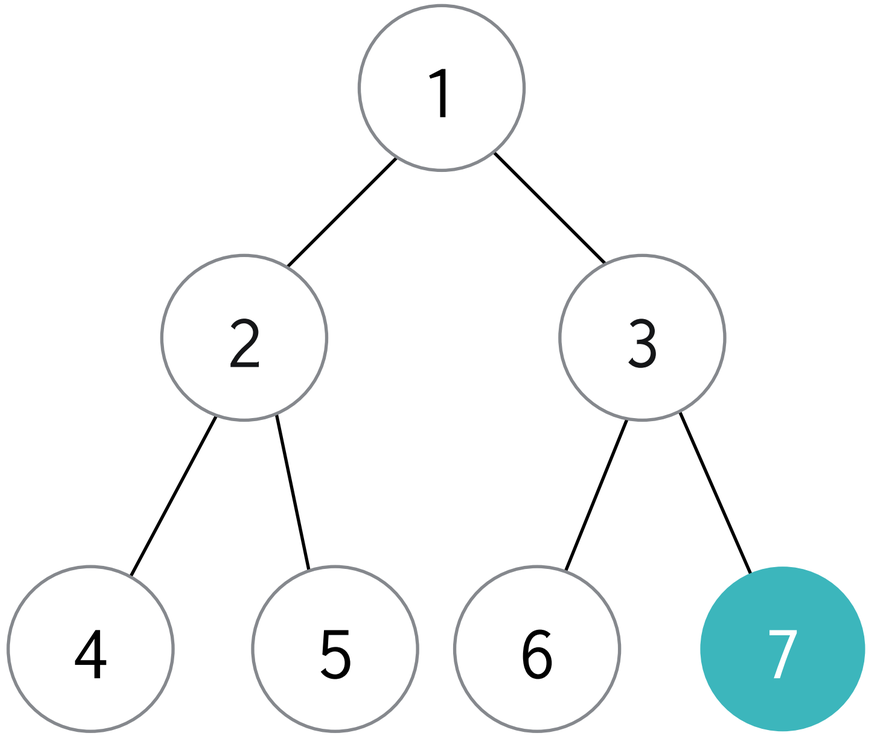


queue = [7]

방문한 노드 : 1, 2, 3, 4, 5, 6

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?

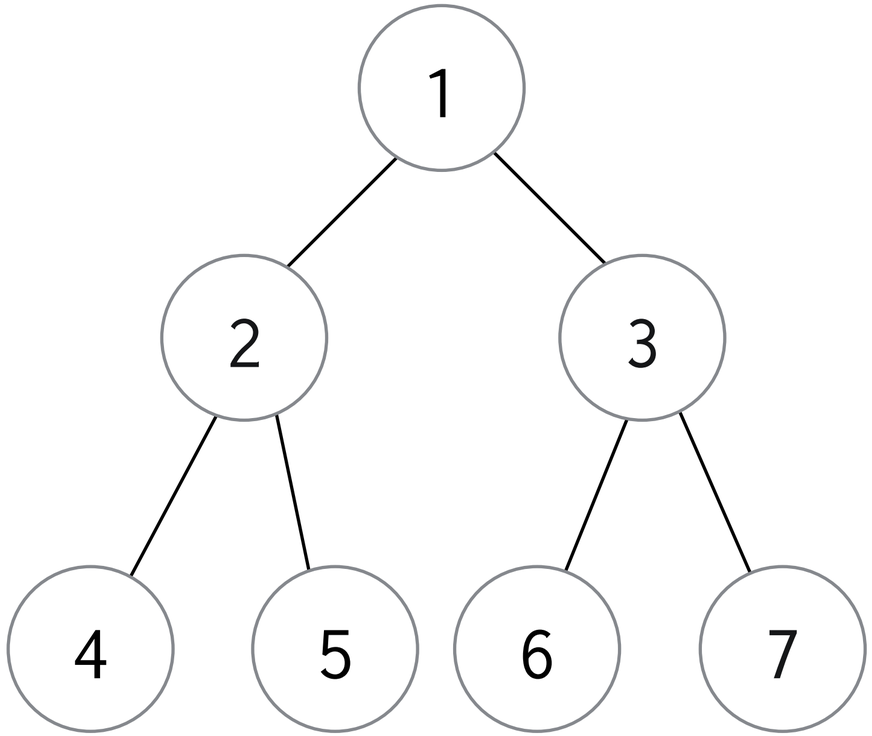


queue = [7]

방문한 노드 : 1, 2, 3, 4, 5, 6, 7

트리 순회하기 - BFS

이 트리를 **너비 우선 탐색**한 결과는?



queue = []

방문한 노드 : 1, 2, 3, 4, 5, 6, 7

03 트리 순회하기

[실습1] 이진 트리 순회 (1)

이진 트리가 주어졌을 때 **전위, 중위, 후위 순회한 결과**를 출력해야 한다.
정점의 개수와, 각 정점에 대한 정보가 주어진다.

입력 예시

```
5
1 2 3
2 4 5
3 -1 -1
4 -1 -1
5 -1 -1
```

출력 예시

```
1 2 4 5 3
4 2 5 1 3
4 5 2 3 1
```

/* elice */

03 트리 순회하기

[실습2] 이진 트리 순회 (2)

이진 트리가 주어졌을 때
너비 우선 탐색을 수행한 결과를 출력해야 한다.

입력 예시

```
5
1 2 3
2 4 5
3 -1 -1
4 -1 -1
5 -1 -1
```

출력 예시

```
1 2 3 4 5
```

/* elice */

03 트리 순회하기

[실습3] 이진 트리 만들기

주어진 입력값을 저장하는 **이진 트리**를 구현해보자.
올바르게 구현한 경우 정상적으로 순회 결과가 출력된다.

입력 예시

```
5
1 2 3
2 4 5
3 -1 -1
4 -1 -1
5 -1 -1
```

출력 예시

```
1 2 4 5 3
4 2 5 1 3
4 5 2 3 1
```

/* elice */

04

트리의 활용

이진 탐색 트리

컴퓨터에서 트리를 **활용하는 예시**는

대표적으로 **이진 탐색 트리**가 있다.

정렬된 상태를 유지하는 배열의 시간 복잡도

임의의 자료들을 담고 있는 자료구조 A가 있다고 가정할 때,

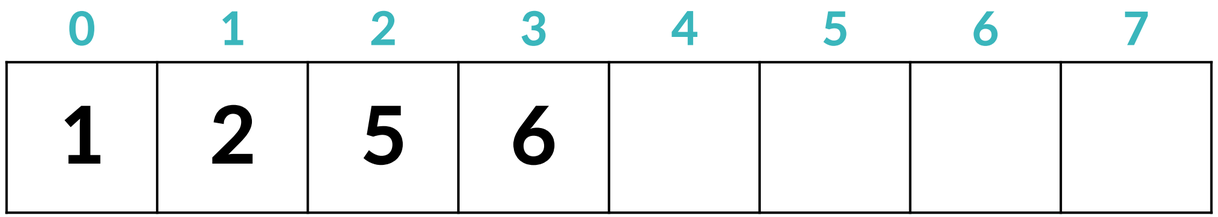
A는 **항상 정렬된 상태**를 유지해야 한다고 생각해보자.

만약 A가 **배열**이라면 자료의 추가, 삭제, 탐색은 다음과 같다.

1장에서 배운 **배열**의 연산을 다시 복습해보자.

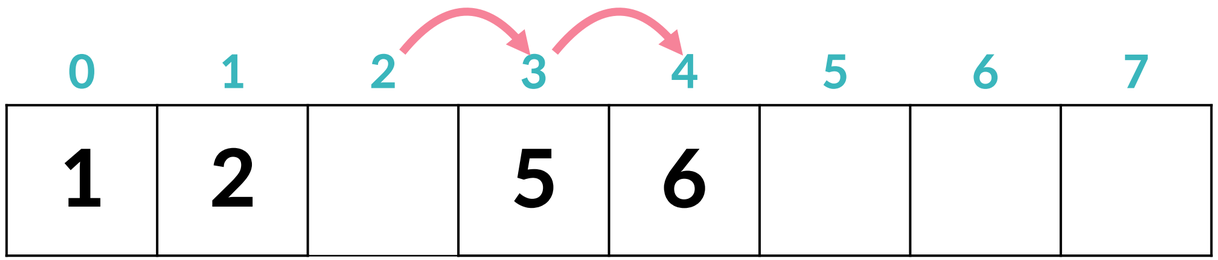
정렬된 상태를 유지하는 배열의 시간 복잡도

4를 새로 추가한다.



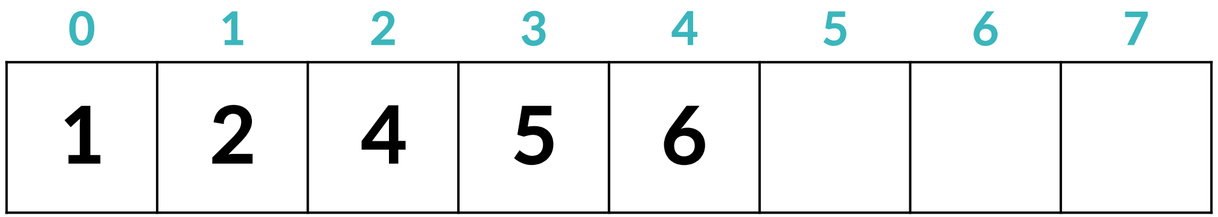
정렬된 상태를 유지하는 배열의 시간 복잡도

정렬을 유지해야 하기 때문에
4가 들어갈 공간을 마련해주어야 한다.



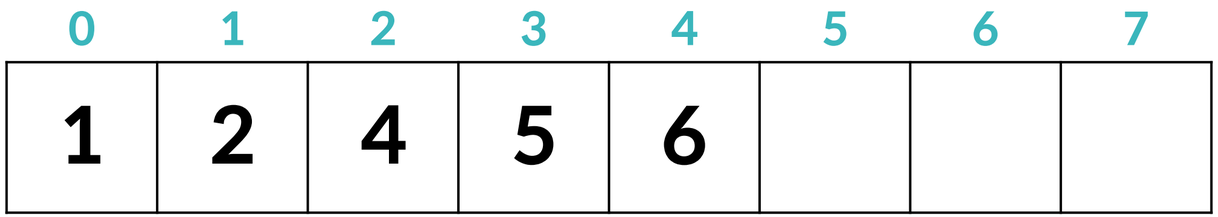
정렬된 상태를 유지하는 배열의 시간 복잡도

정렬이 유지되도록 새로운 자료를 추가한다.



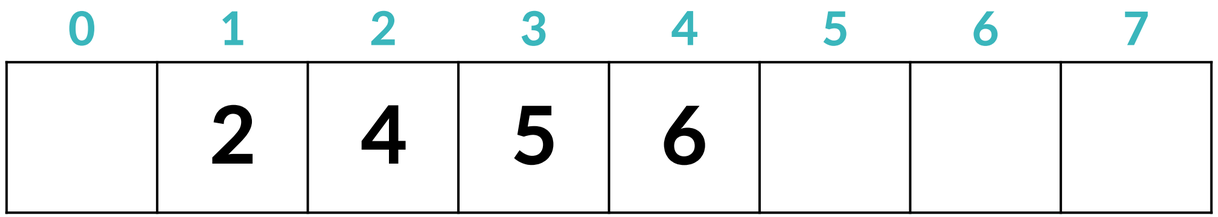
정렬된 상태를 유지하는 배열의 시간 복잡도

이번에는 1을 삭제하는 경우이다.



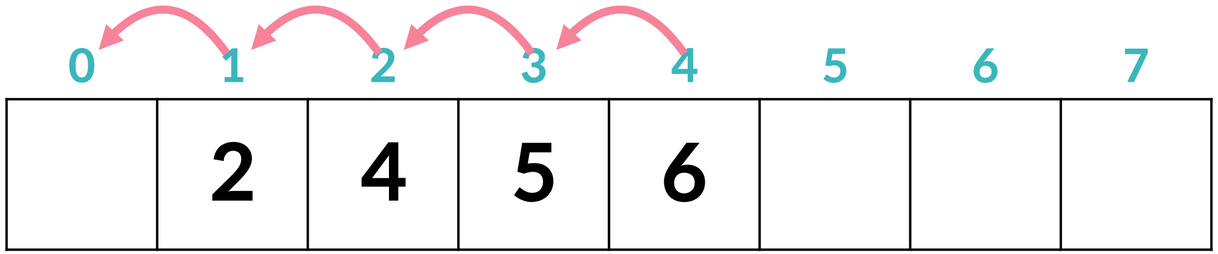
정렬된 상태를 유지하는 배열의 시간 복잡도

이번에는 1을 삭제하는 경우이다.



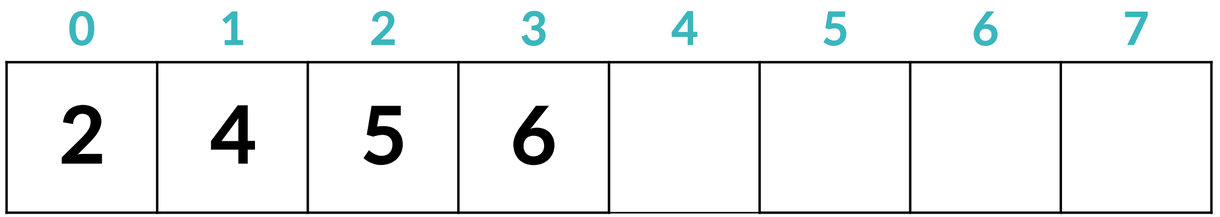
정렬된 상태를 유지하는 배열의 시간 복잡도

배열의 비어있는 인덱스를 채워야 하므로,
1이 **삭제되고 생긴 빈자리**를 채우기 위해 당겨와야 한다.



정렬된 상태를 유지하는 배열의 시간 복잡도

정렬된 상태를 유지하는 배열의 삭제는
일반 배열의 삭제와 동일하다.



정렬된 상태를 유지하는 배열의 시간 복잡도

정렬된 상태를 유지하는 배열의 **추가와 삭제** 연산은

1장에서 배운 배열의 특성대로 $O(n)$의 시간 복잡도를 가진다.

정렬된 상태를 유지하는 배열의 시간 복잡도

정렬된 자료구조에서 사용할 수 있는 탐색 알고리즘인

이진 탐색을 이용하면 정렬된 배열 내에서의 **자료 탐색**을

$O(logn)$만에 수행할 수 있다.

정렬된 상태를 유지하는 배열의 시간 복잡도

정렬된 배열의 **중간값**과 **찾는 값**의 크기를 비교하고,

중간값보다 **작은** 경우에는 중간값을 기준으로 **좌측**,

중간값보다 **큰** 경우에는 중간값을 기준으로 **우측**을 대상으로 다시 탐색한다.

정렬된 상태를 유지하는 배열의 시간 복잡도

1부터 n까지의 자연수 중 상대방이 생각하고 있는 숫자를 맞히는
'업 다운 게임'에서 **최적의 전략**으로써 사용할 수 있다.

예를 들어 이진 탐색을 사용하여 1부터 100까지 자연수 중
상대방이 생각하고 있는 숫자를 맞히기까지 걸리는 횟수는
$O(log100) = 6.64\ldots$ 이므로 **최악의 경우** 7회이다.

정렬된 상태를 유지하는 배열의 시간 복잡도

따라서 정렬된 배열의 삽입, 삭제, 탐색의 시간 복잡도는 다음과 같다.

삽입	삭제	탐색
$O(n)$	$O(n)$	$O(logn)$

이진 탐색 트리

정렬된 상태를 유지해야 하는 자료구조 A가 **트리**로 구현되어 있다면

더 **효율적**으로 연산이 가능하다.

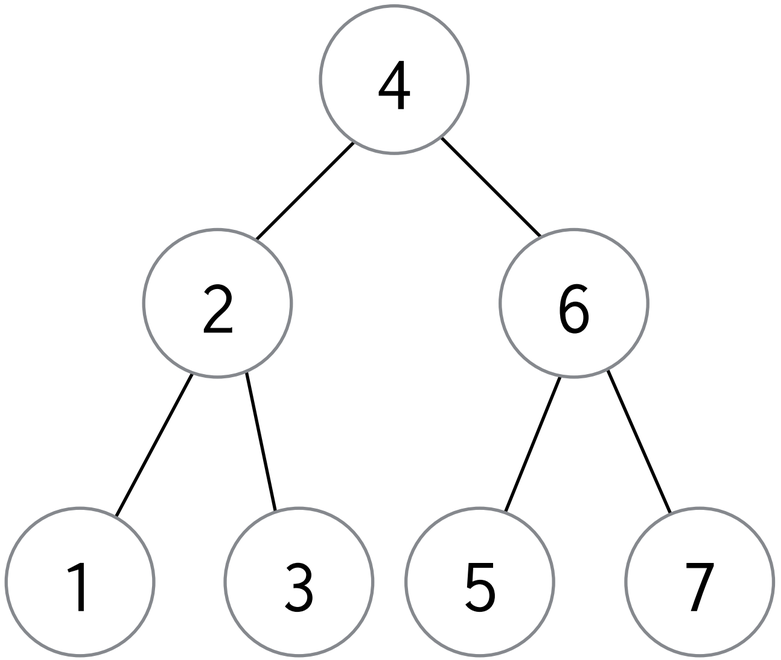
이진 탐색 트리

이진 탐색 트리는 항상 정렬된 상태를 유지하는 자료구조이며

어떤 정점의 **왼쪽** 서브 트리는 그 정점보다 **같거나 작은 정점**들로만,

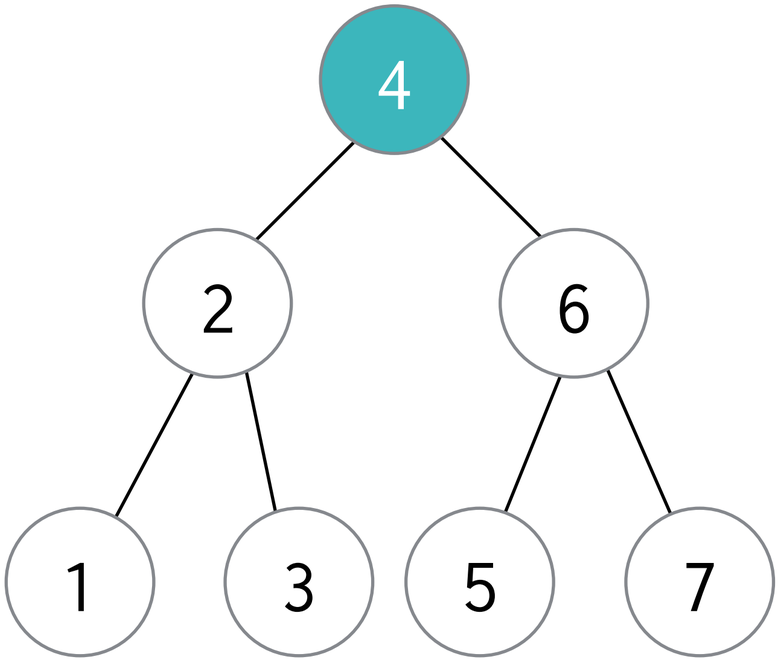
오른쪽 서브 트리는 그 정점의 값보다 **큰 정점**들로만 이루어져 있다.

이진 탐색 트리



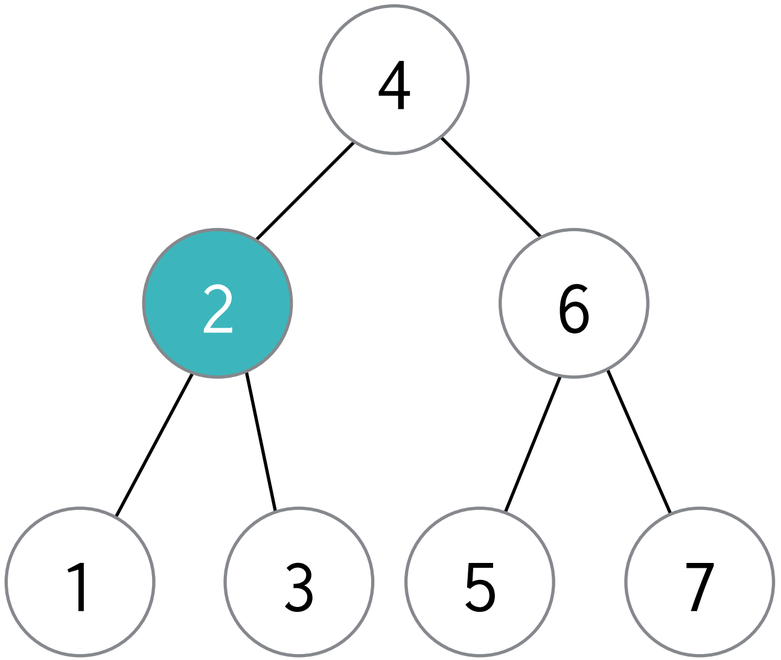
0 추가

이진 탐색 트리



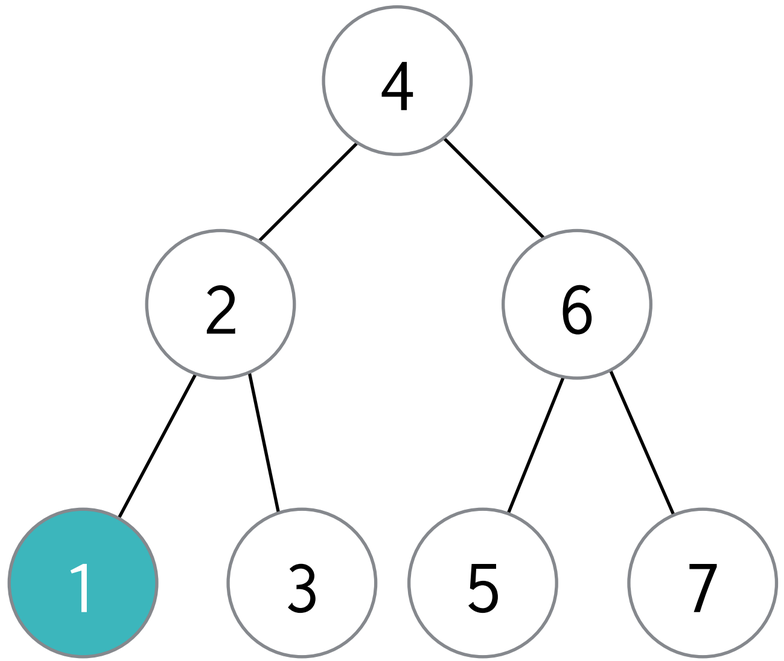
0 추가

이진 탐색 트리



0 추가

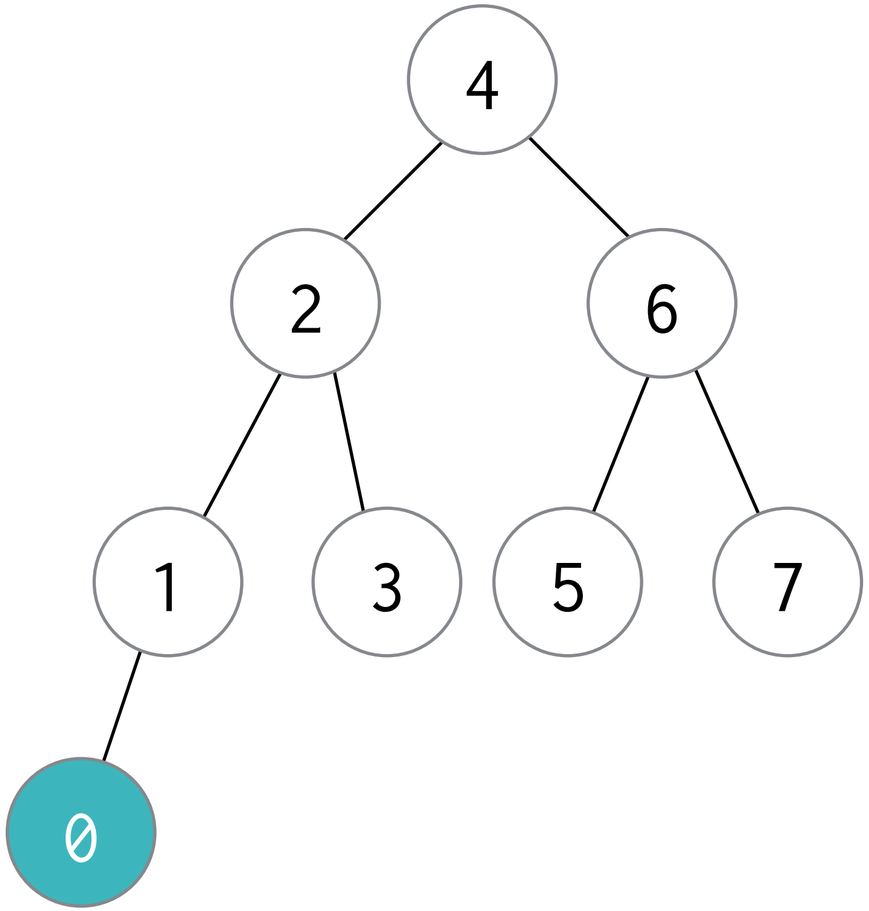
이진 탐색 트리



0 추가

04 트리의 활용

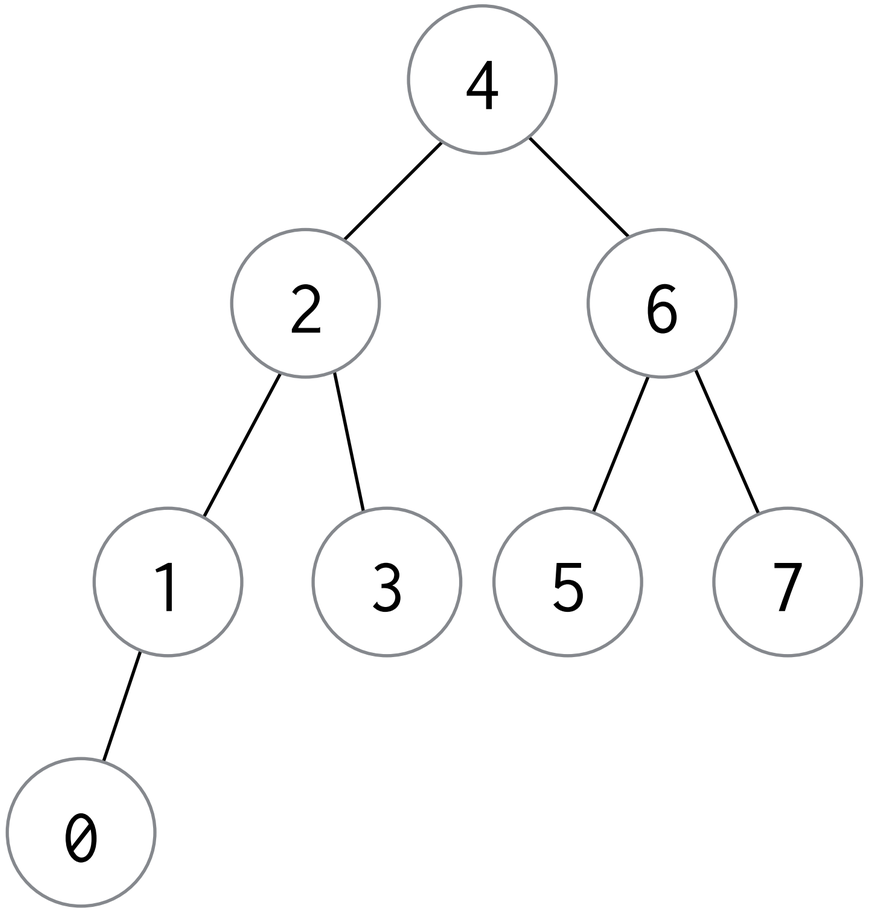
이진 탐색 트리



0은 1보다 작으므로
1의 왼쪽 자식으로 들어간다.

04 트리의 활용

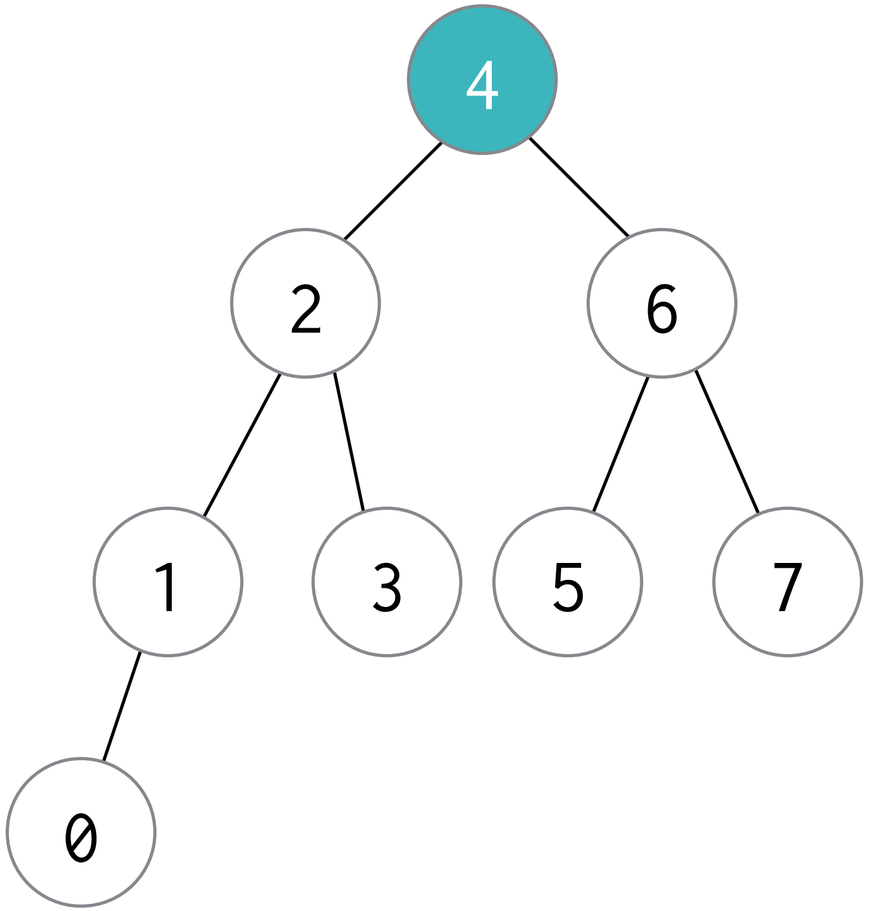
이진 탐색 트리



5 삭제

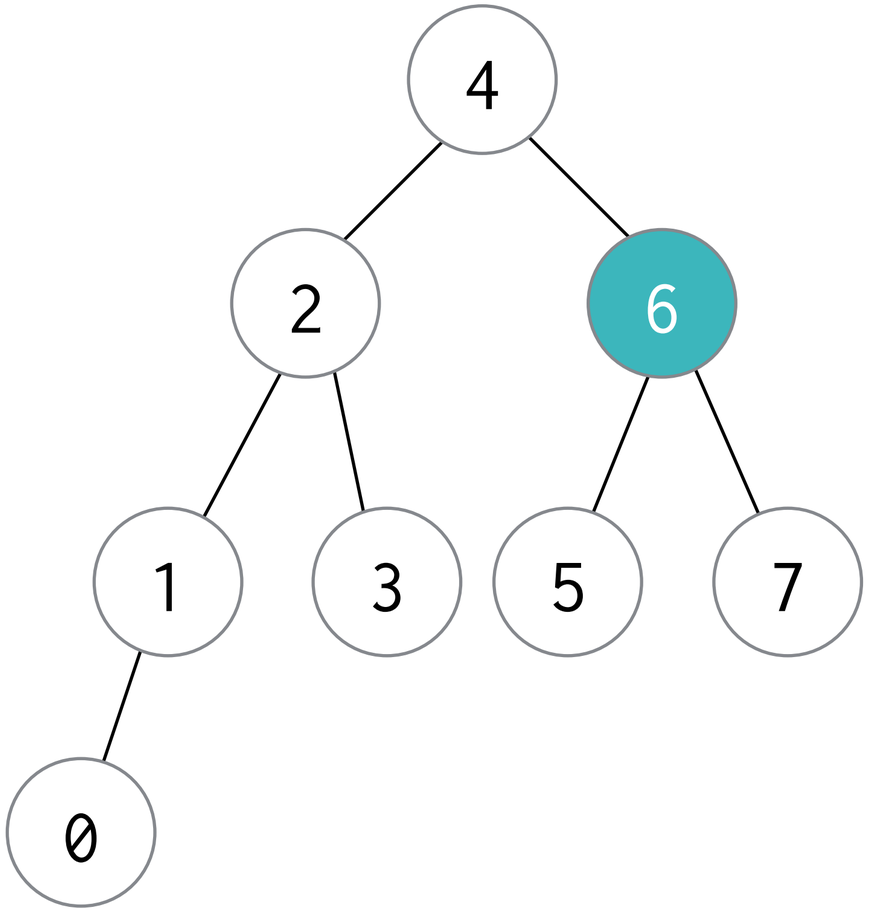
04 트리의 활용

이진 탐색 트리



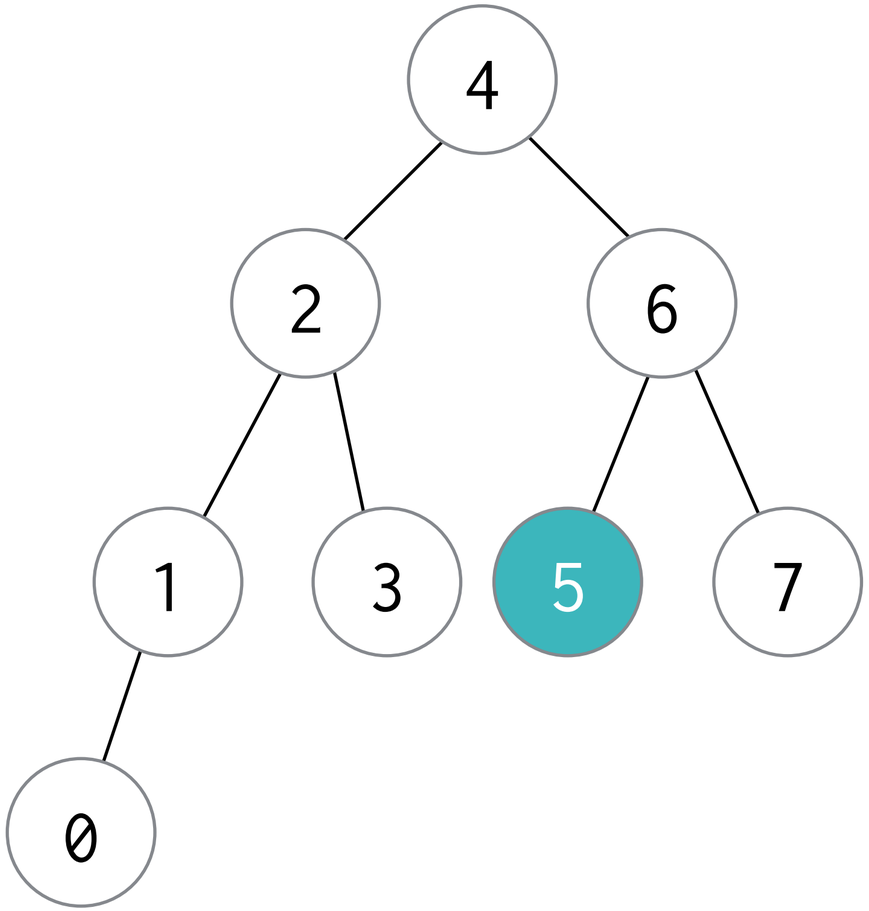
5 삭제

이진 탐색 트리



5 삭제

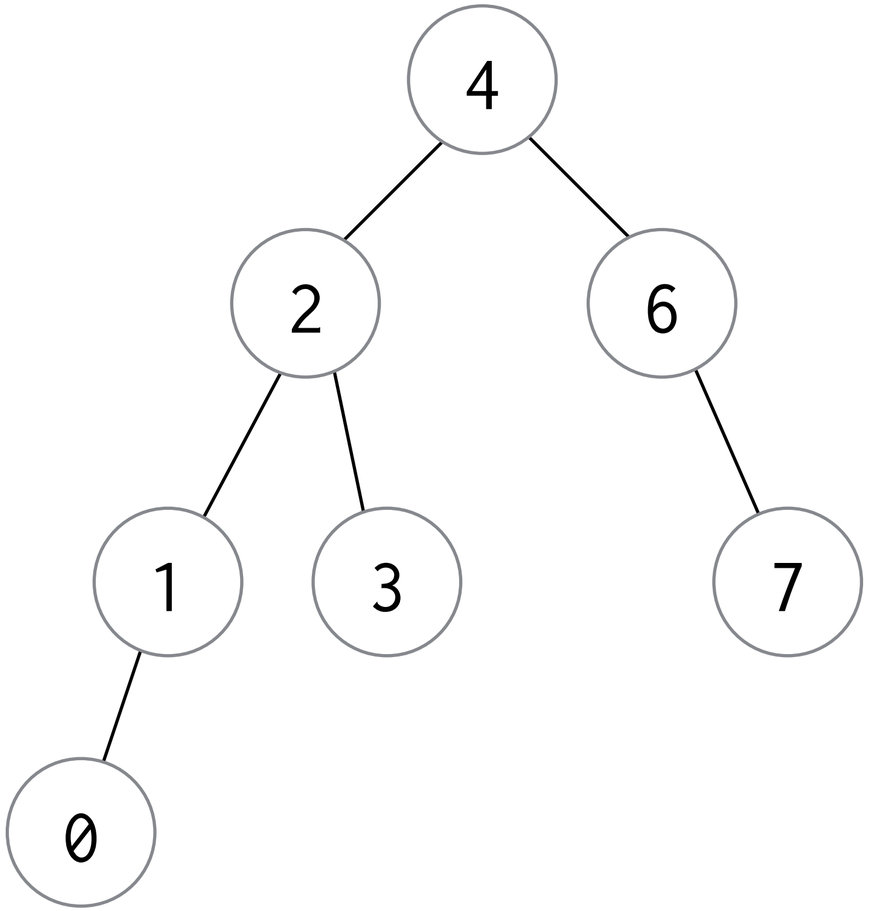
이진 탐색 트리



5 삭제

04 트리의 활용

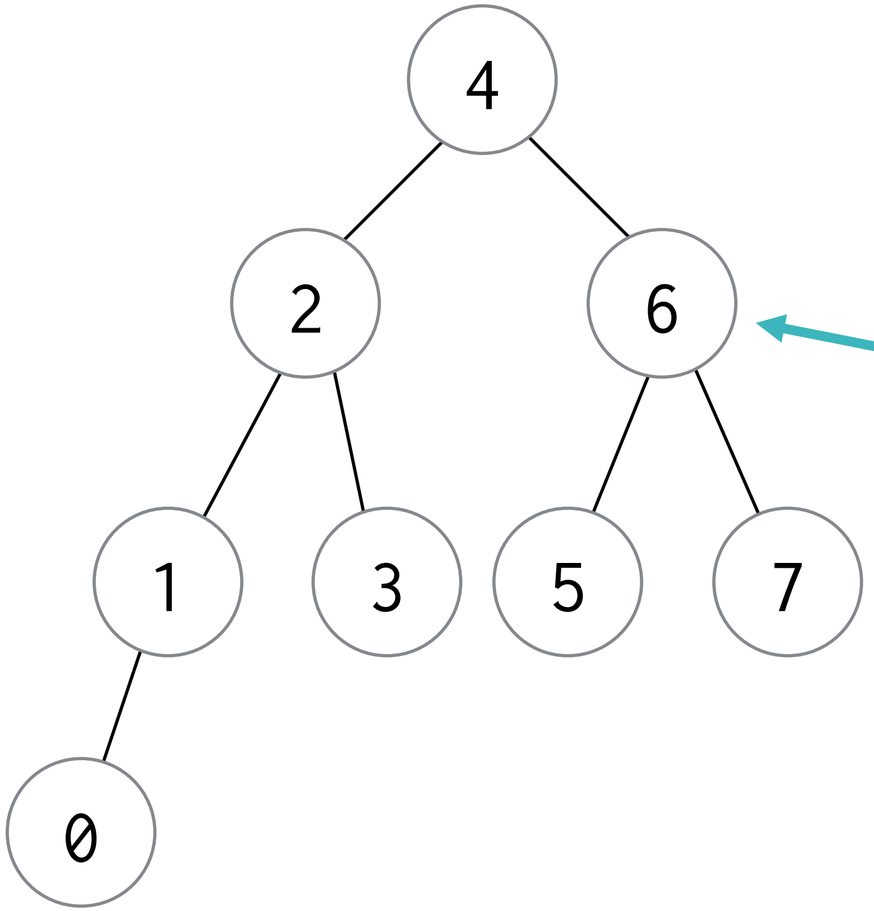
이진 탐색 트리



5 삭제

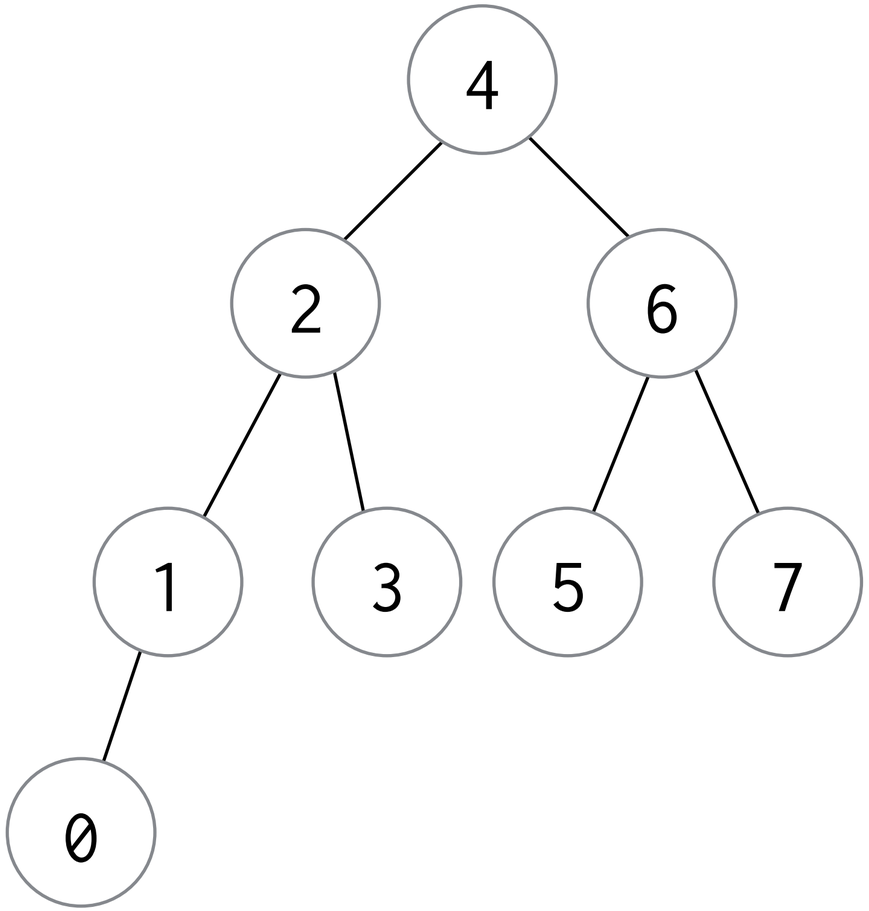
/* elice */

이진 탐색 트리



만약 6과 같이
중간에 있는 정점을
삭제해야 하는 경우에는?

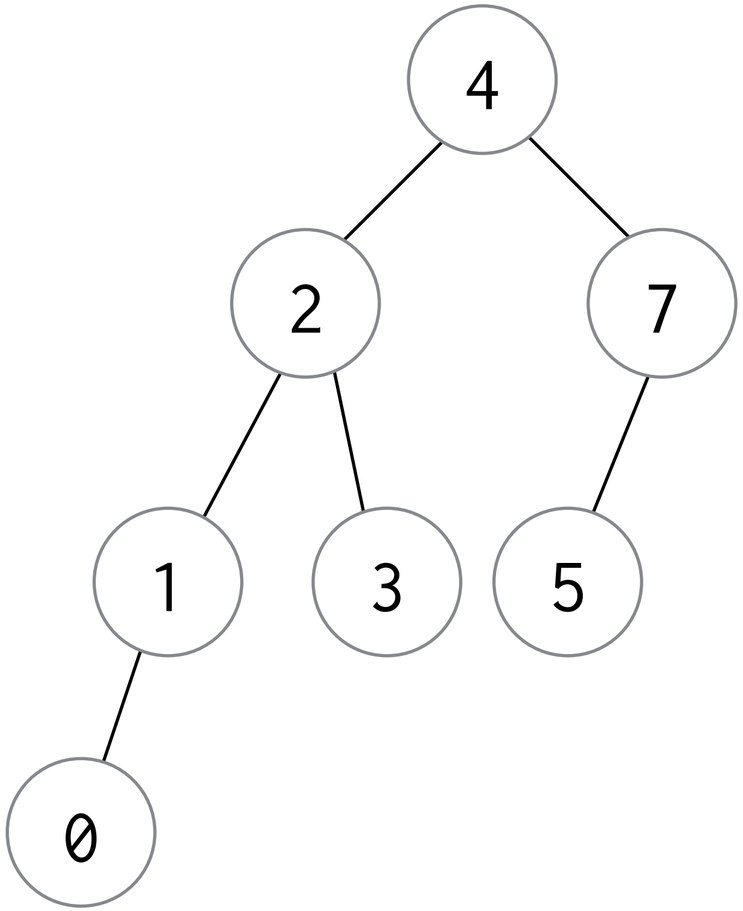
이진 탐색 트리



6은 루트 노드를 기준으로
오른쪽 서브 트리에 존재한다.

6의 오른쪽 서브 트리 중
가장 왼쪽에 있는 정점인 7이
6의 위치를 대신한다.

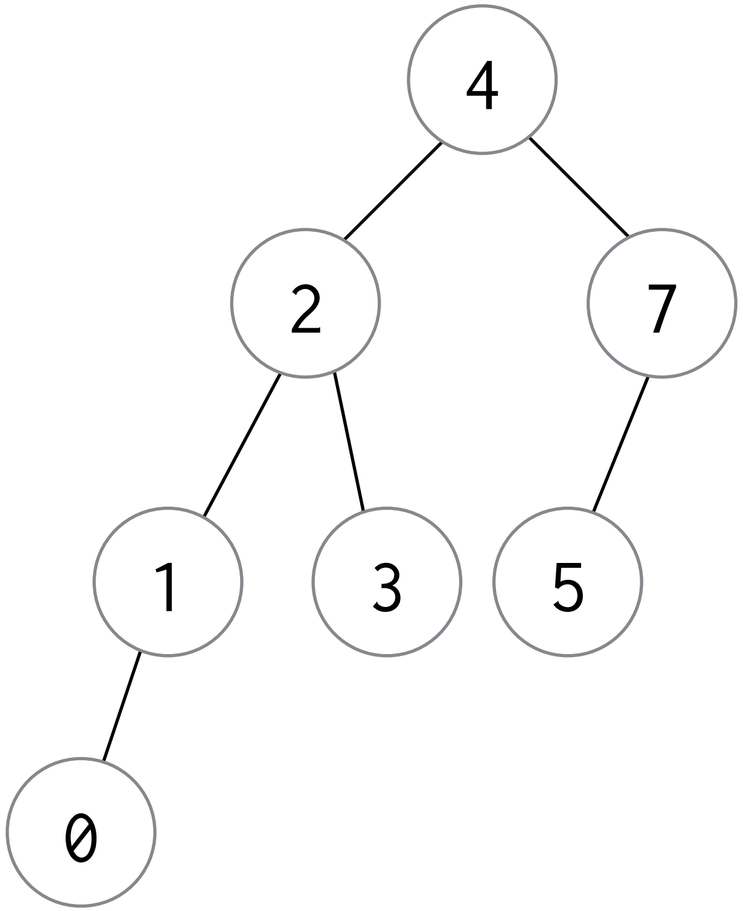
이진 탐색 트리



6이 삭제된 모습이다.

04 트리의 활용

이진 탐색 트리



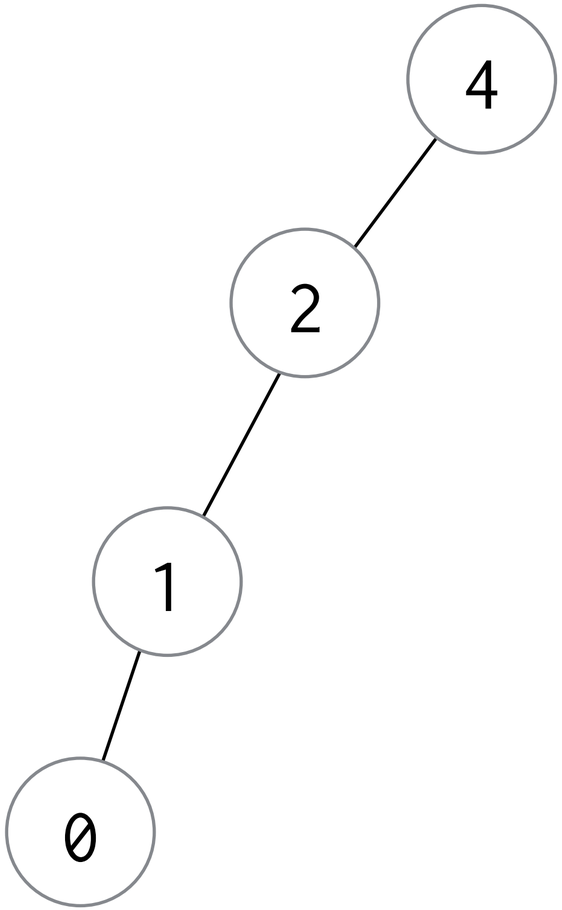
이진 탐색 트리에서
각 요소를 오름차순으로 탐색하기 위해서
중위 순회를 이용할 수 있다.

이진 탐색 트리

이진 탐색 트리의 삽입, 삭제 시간복잡도는
트리의 높이에 비례한다.

삽입	삭제	탐색
$O(logn)$	$O(logn)$	$O(logn)$

이진 탐색 트리의 문제점



한쪽으로 **편향**된
이진 탐색 트리도 존재할 수 있다.

이를 **편향 이진 트리**라고 부른다.

이 경우 이진 탐색 트리의 장점을
살리지 못한다는 문제점이 존재한다.

이진 탐색 트리의 문제점

실제로 많이 활용되는 이진 탐색 트리는

자가 균형 이진 탐색 트리라고도 불리는 '레드블랙 트리'를 사용한다.

(레드블랙 트리에 대한 자세한 내용은 학습 범위를 벗어나므로 다루지 않습니다.)

05

트리 실습

[실습4] 트리의 높이

주어진 트리의 **높이**를 구하는 코드를 작성해야 한다.

입력 예시

```
5
1 2 3
2 4 5
3 -1 -1
4 -1 -1
5 -1 -1
```

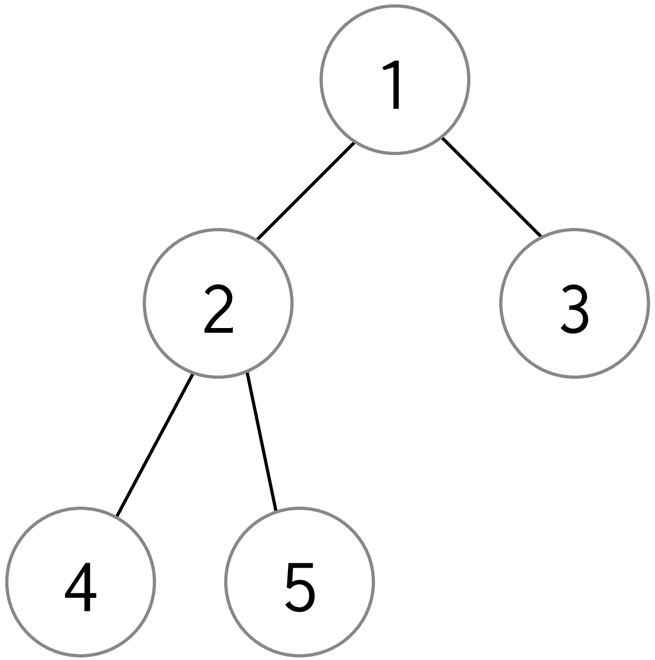
출력 예시

```
3
```

/* elice */

[실습4] 트리의 높이

예시로 주어진 트리는 아래의 모습을 하고 있다.

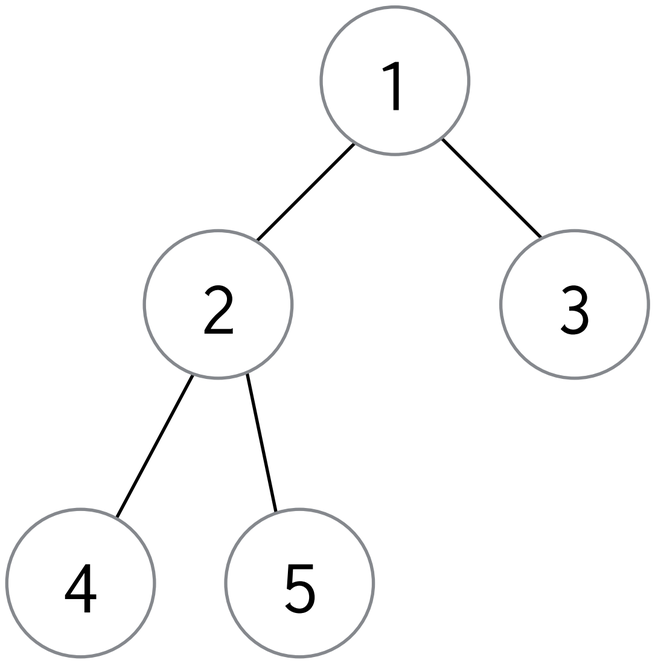


[실습4] 트리의 높이

트리를 DFS로 순회하다 보면 언젠가 **리프 노드**에 도달하게 되는데,

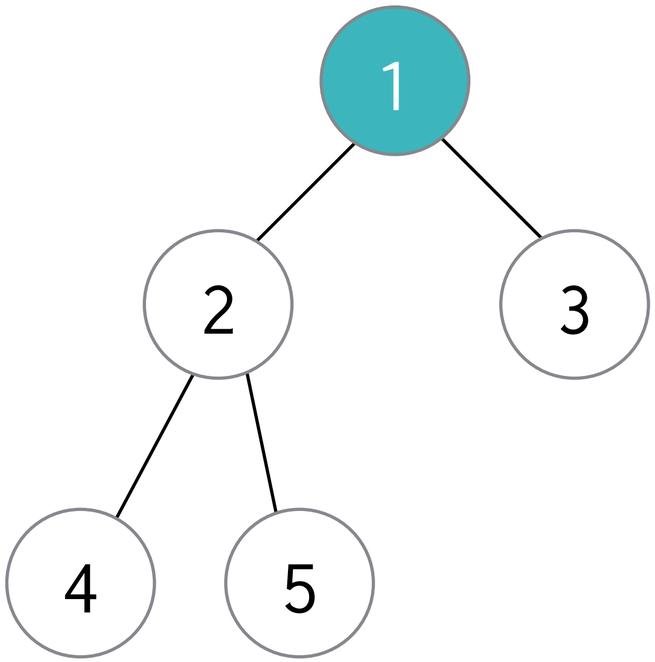
이때 각 노드가 루트 노드로부터 **얼마나 떨어져 있는지** 계산할 수 있다.

[실습4] 트리의 높이

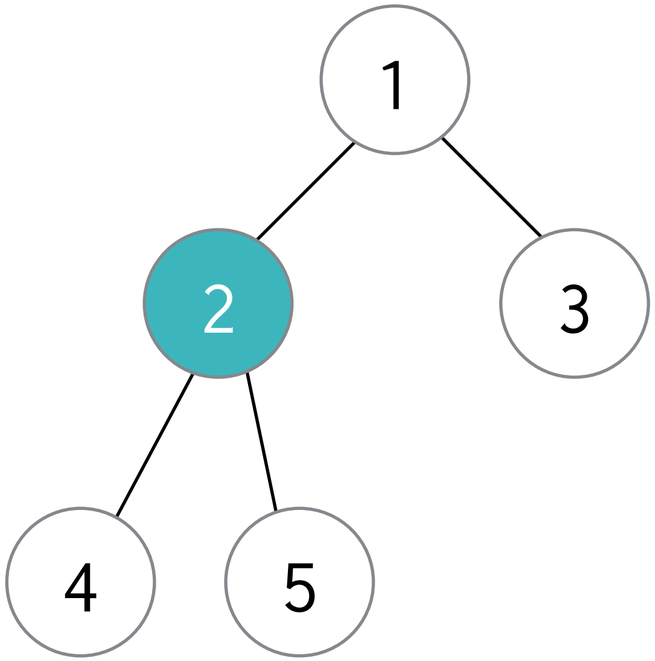


05 트리 실습

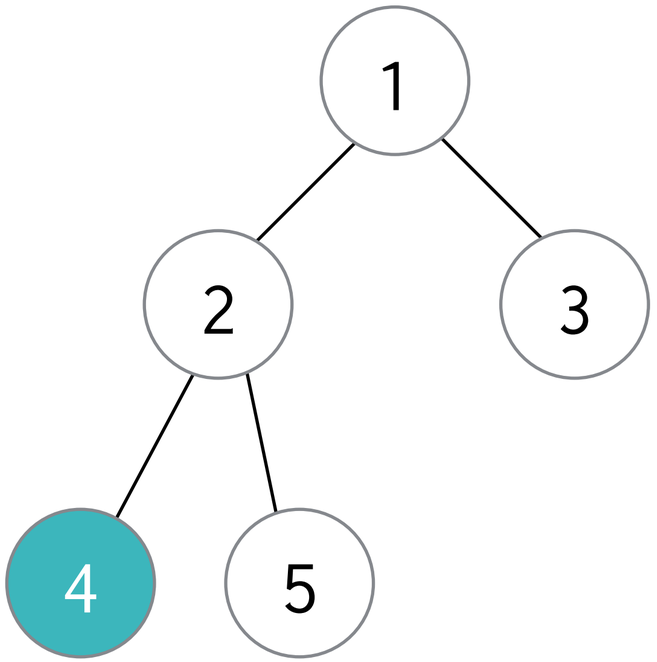
[실습4] 트리의 높이



[실습4] 트리의 높이

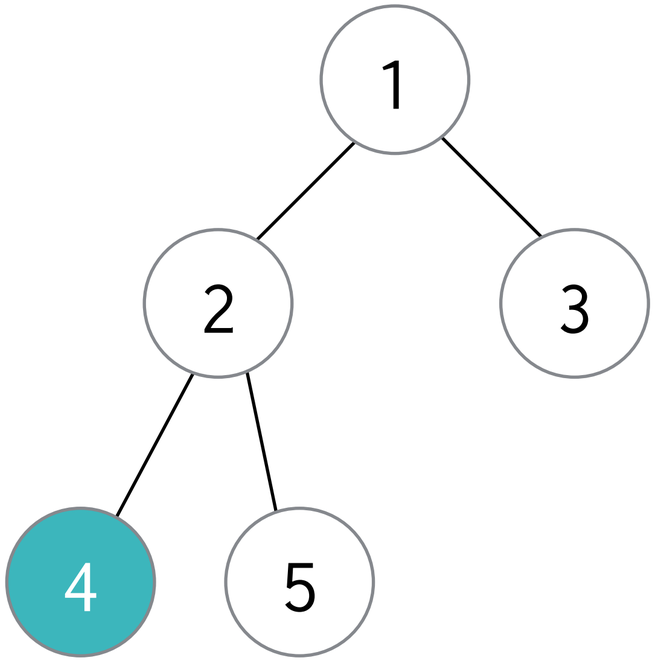


[실습4] 트리의 높이



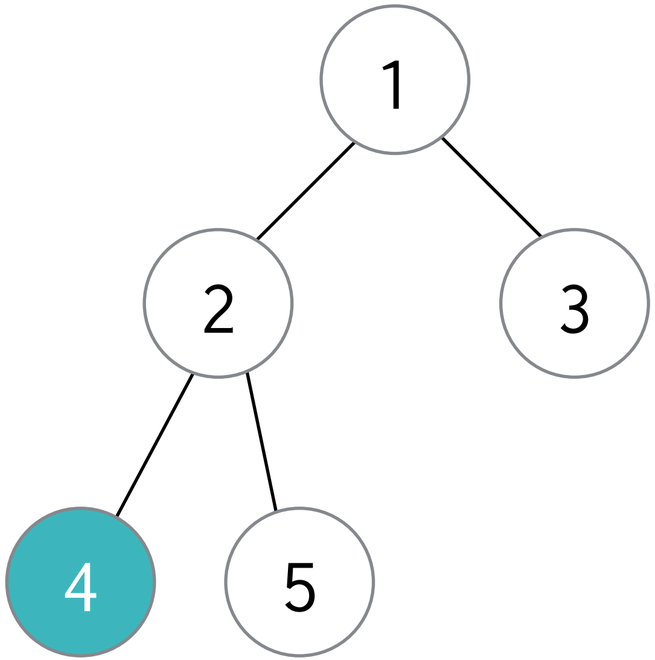
[실습4] 트리의 높이

4는 리프 노드이므로 **정점 4**의 깊이는 **2**가 된다.



[실습4] 트리의 높이

이와 같이 모든 **리프 노드**에 대해 깊이를 구하고,
가장 큰 값에 1을 더해 출력해주면 된다.



[실습5] 트리의 너비

주어진 트리의 **너비**가 가장 큰 레벨과 그 레벨의 너비를 계산해야 한다.
레벨이란, **깊이가 같은 노드들**의 집합을 의미하며 **루트 노드부터 1로 시작**한다.

입력 예시

```
5
1 2 3
2 4 5
3 -1 -1
4 -1 -1
5 -1 -1
```

출력 예시

```
2 4
```

/* elice */

[실습5] 트리의 너비

트리의 각 정점을 격자로 정리하여 풀이 과정을 이해해보자.

같은 레벨의 노드는 **같은 행**에 위치해야 하고,
한 열에는 **하나의 정점**만 위치해야 한다.

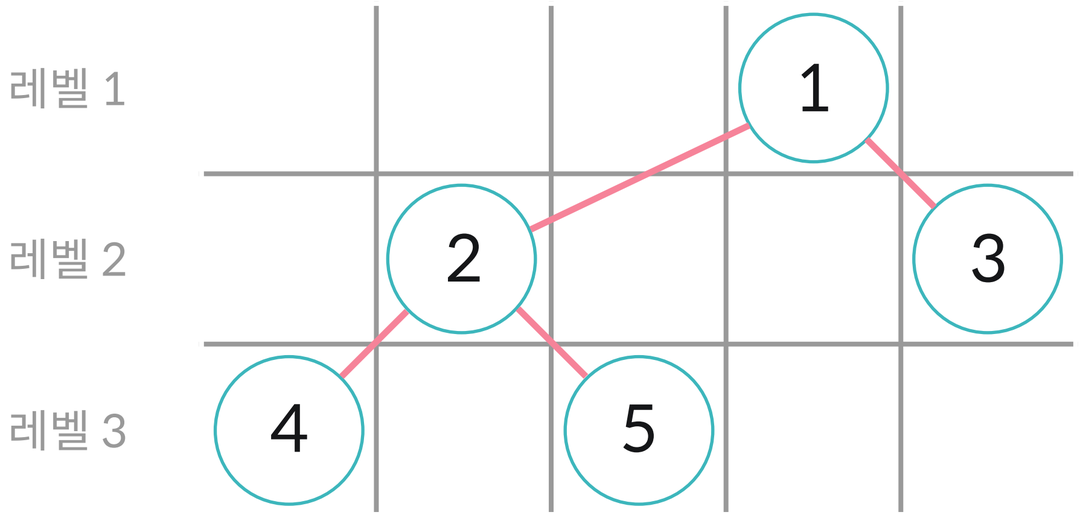
[실습5] 트리의 너비

또, 한 정점의 **왼쪽** 서브 트리의 정점들은 모두
그 정점보다 **왼쪽의 열**에 위치해야 하며

오른쪽 서브 트리의 정점들은 모두
그 정점보다 **오른쪽의 열**에 위치해야 한다.

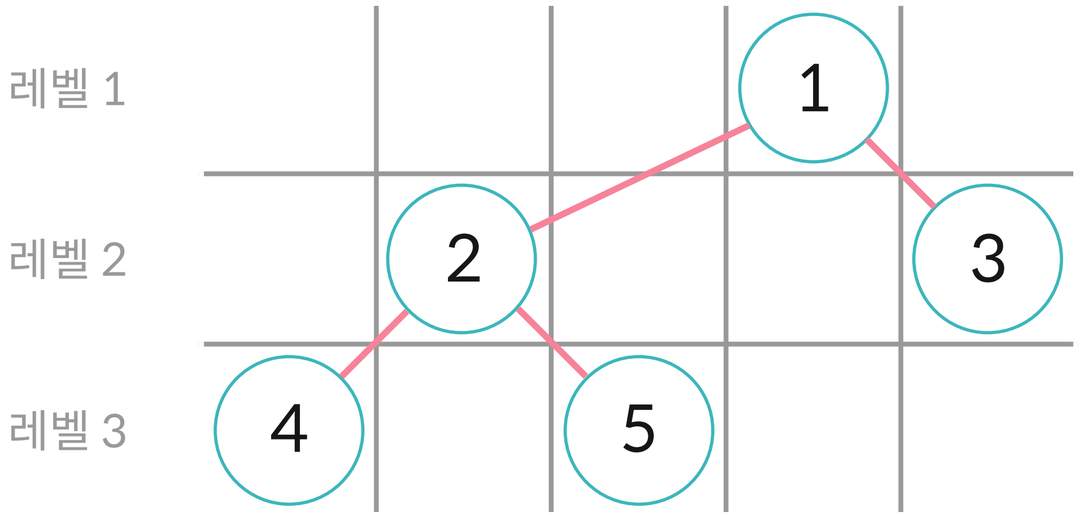
[실습5] 트리의 너비

이러한 규칙으로 트리를 격자에 넣어보면 다음과 같은 모양으로 정리된다.



[실습5] 트리의 너비

이때 가장 너비가 긴 레벨은 2이고, 그 너비는 4이다.



[실습5] 트리의 너비

정점의 **행**은 각 정점의 **깊이**를 구하면서 구할 수 있다.

그런데 **열**은 어떻게 계산해야 할까?

[실습5] 트리의 너비

어떤 정점 A의 **왼쪽 서브 트리의 정점들**의 열이 모두 확정되었다면

비로소 A의 열도 확정지을 수 있다.

그리고서 **오른쪽 서브 트리**의 정점들의 위치를 계산하면 된다.

[실습5] 트리의 너비

왼쪽 서브 트리를 **먼저 확정** 짓는다 = **왼쪽** 서브 트리를 먼저 **방문**한다

즉 **중위 순회**를 이용하여 트리의 너비를 구할 수 있다.