알고리즘 멘토링

- Heap & Dijkstra –

- 주민찬 -

리스트의 최대 최소

- List에서 max value를 찾는 방법은?
- A = [1,5,4,7,3,2]

- List에서 min value를 찾는 방법은?
- A = [1,5,4,7,3,2]

리스트의 최대 최소

- List에서 min value를 차례대로 찾는 방법은?
- A = [1,5,4,7,3,2]

- List에서 max value를 차례대로 찾는 방법은?
- A = [1,5,4,7,3,2]

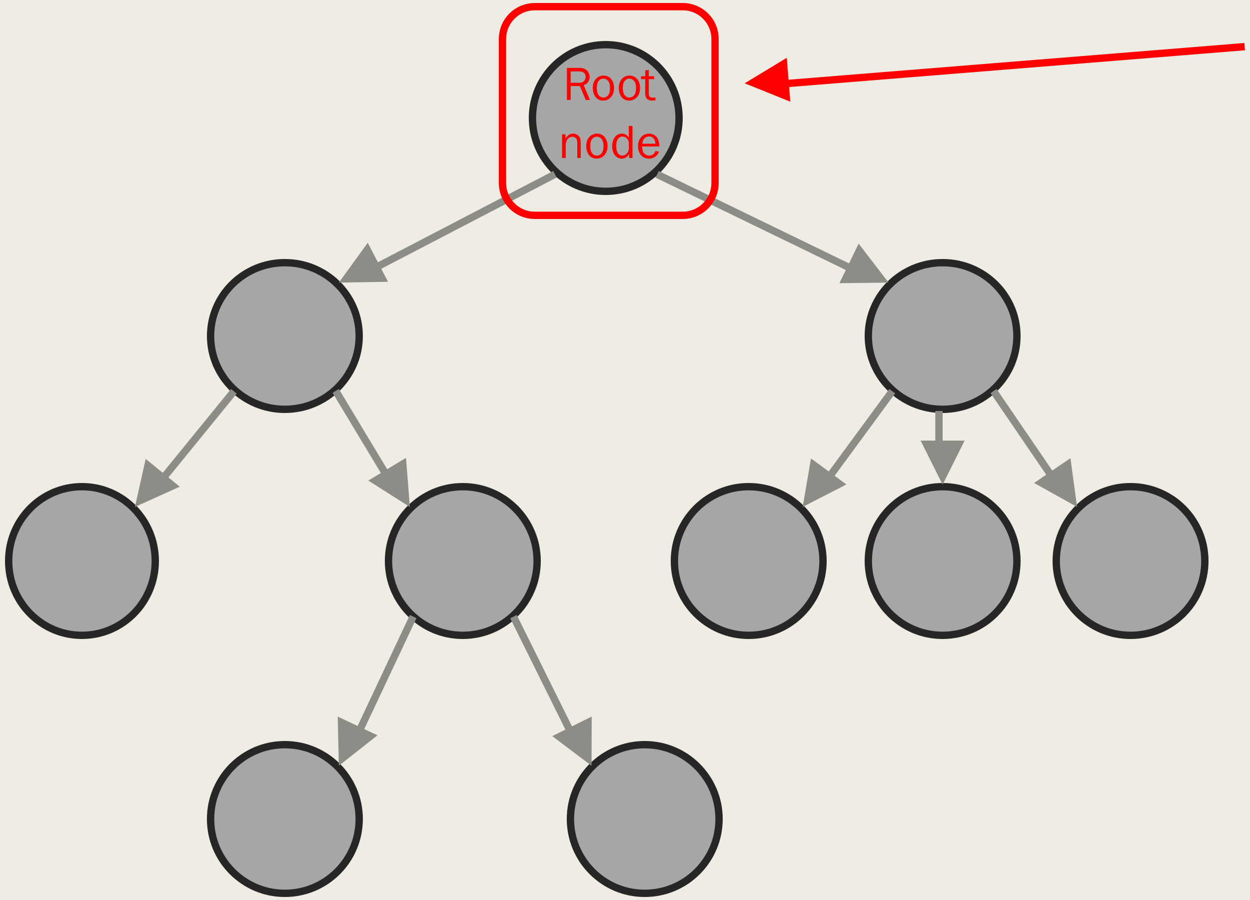
Heap

- 리스트에 값을 추가하고 최대값 구하고 이 과정을 반복

- 아이디어: 리스트에 값을 추가할 때 자동으로 최대값에 따라 정렬할 수는 없을까?

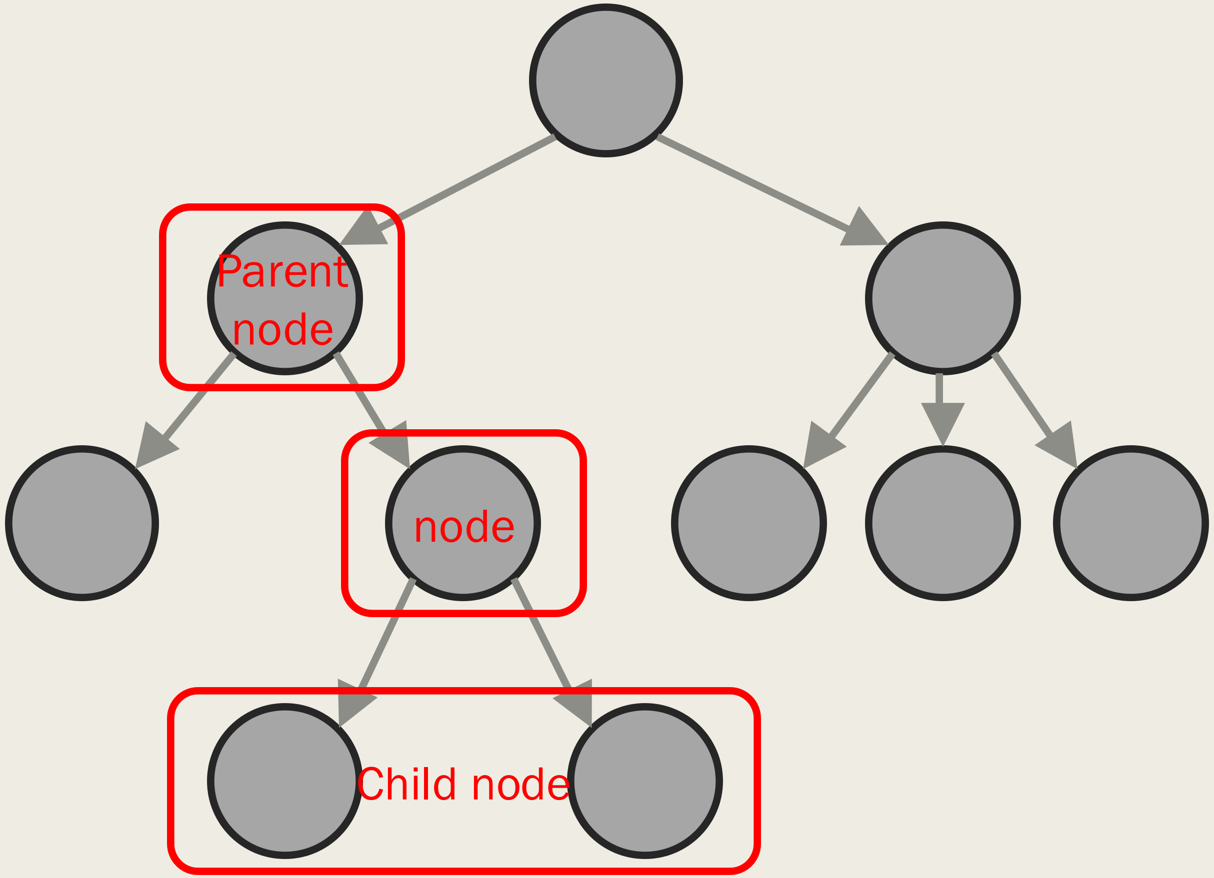
- Heap
  - *update :* $O(logn)$
  - *Find max :* $O(logn)$

Tree



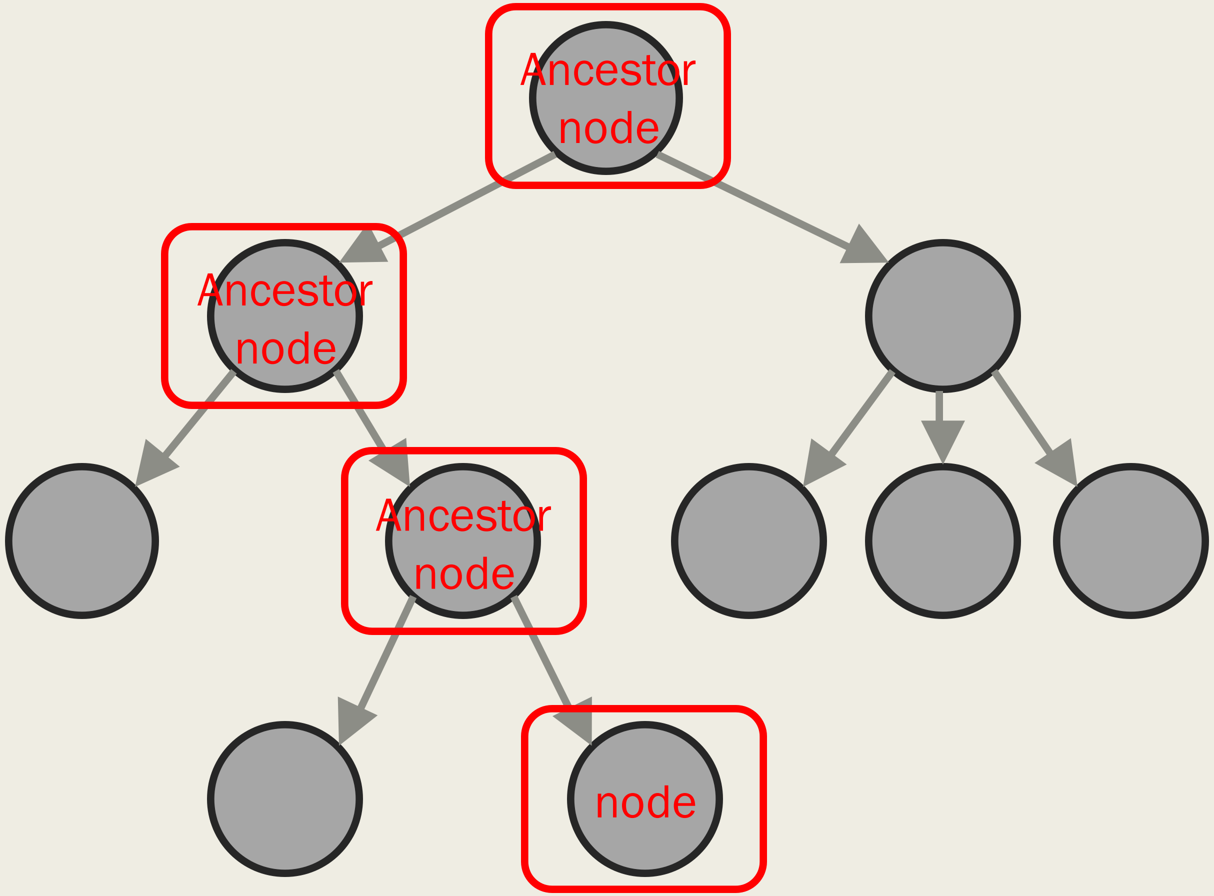
1. Root node
2. Parent node
3. Child node
4. Ancestor node
5. Sibling node
6. Leaf node
7. Height
8. Depth (level)
9. Breadth
10. Degree

Tree



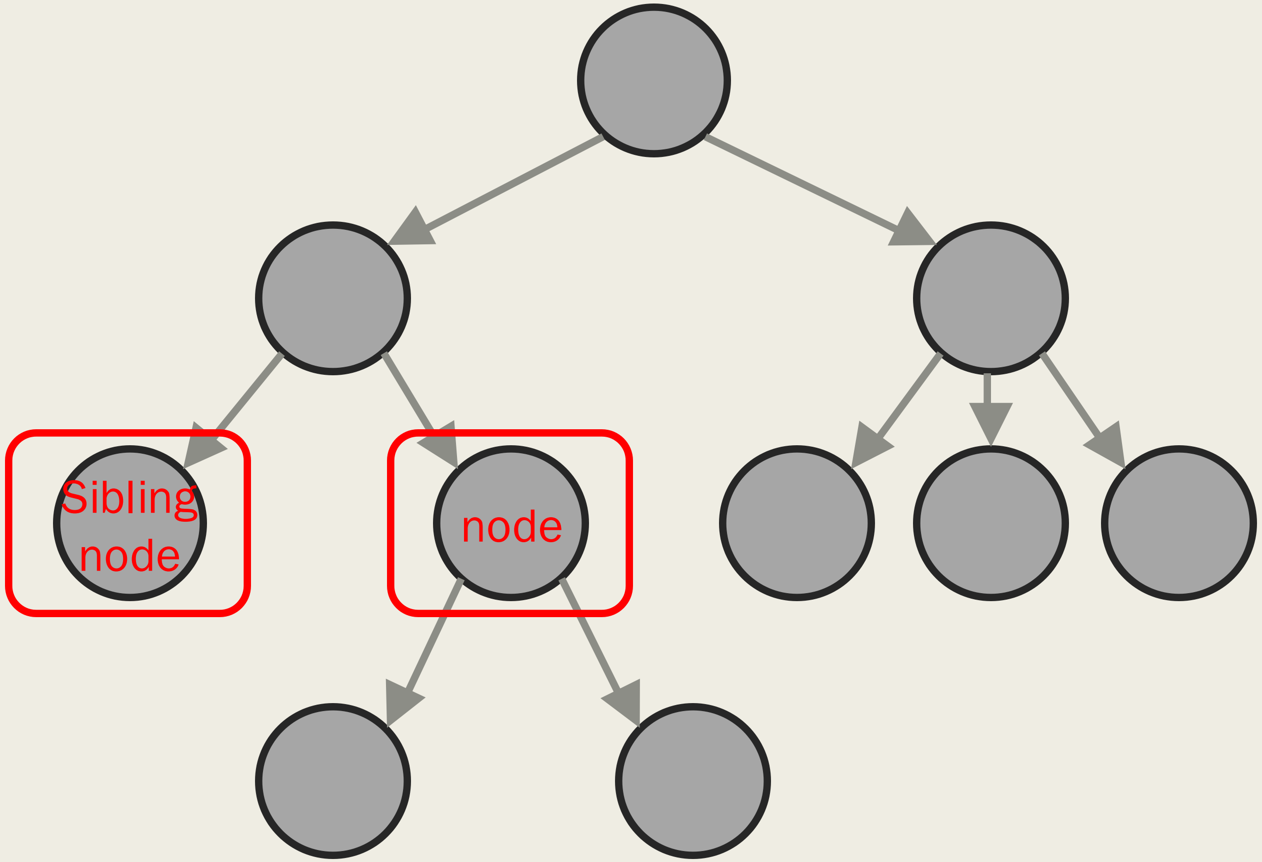
1. Root node
2. Parent node
3. Child node
4. Ancestor node
5. Sibling node
6. Leaf node
7. Height
8. Depth (level)
9. Breadth
10. Degree

Tree



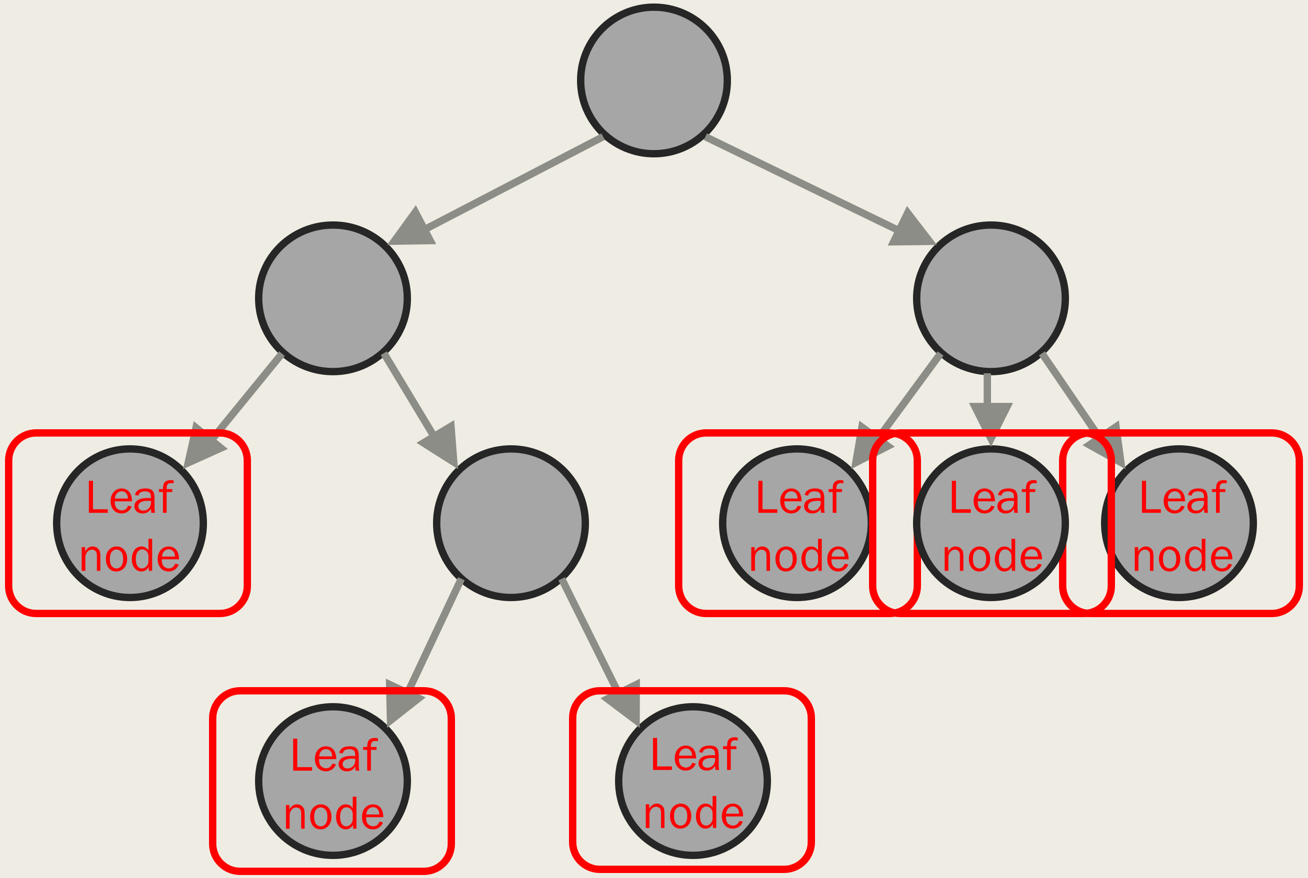
1. Root node
2. Parent node
3. Child node
4. Ancestor node
5. Sibling node
6. Leaf node
7. Height
8. Depth (level)
9. Breadth
10. Degree

Tree



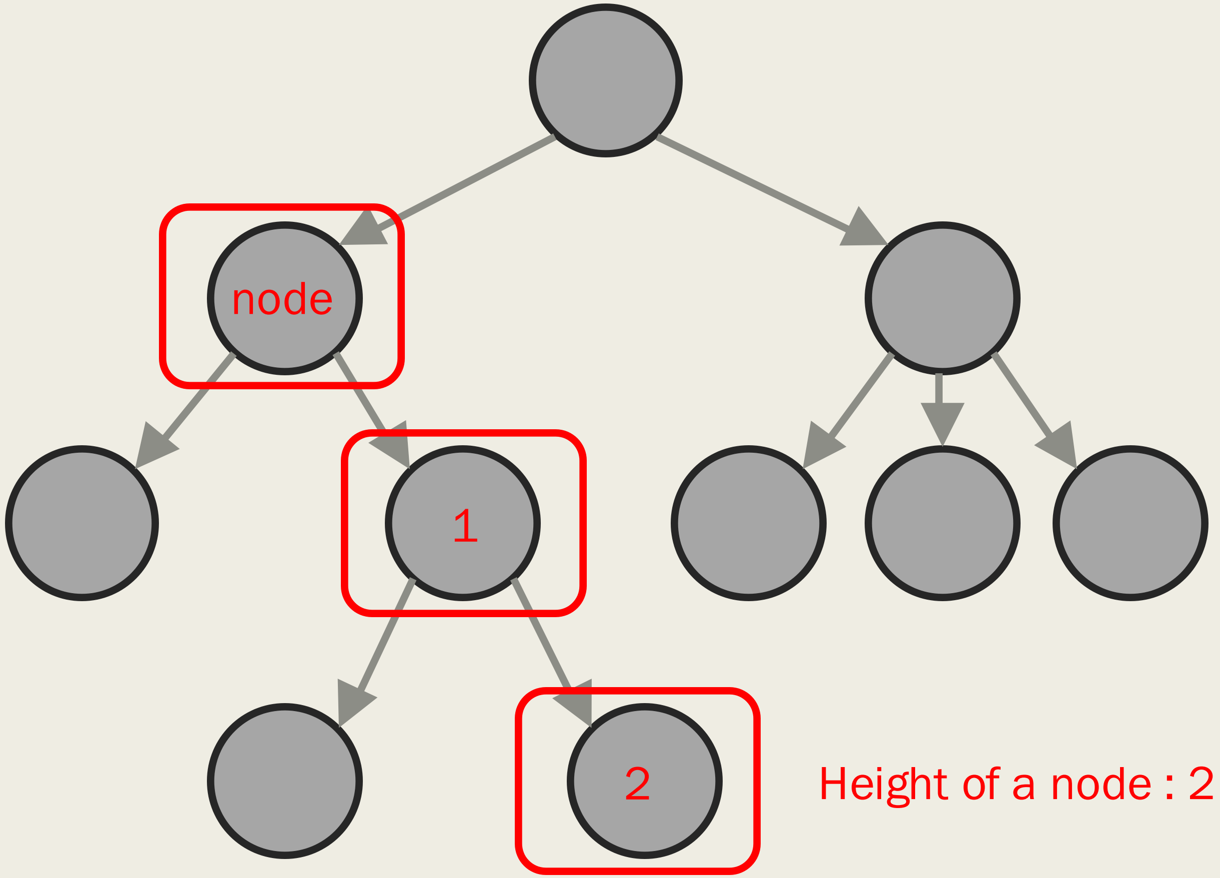
1. Root node
2. Parent node
3. Child node
4. Ancestor node
5. Sibling node
6. Leaf node
7. Height
8. Depth (level)
9. Breadth
10. Degree

Tree



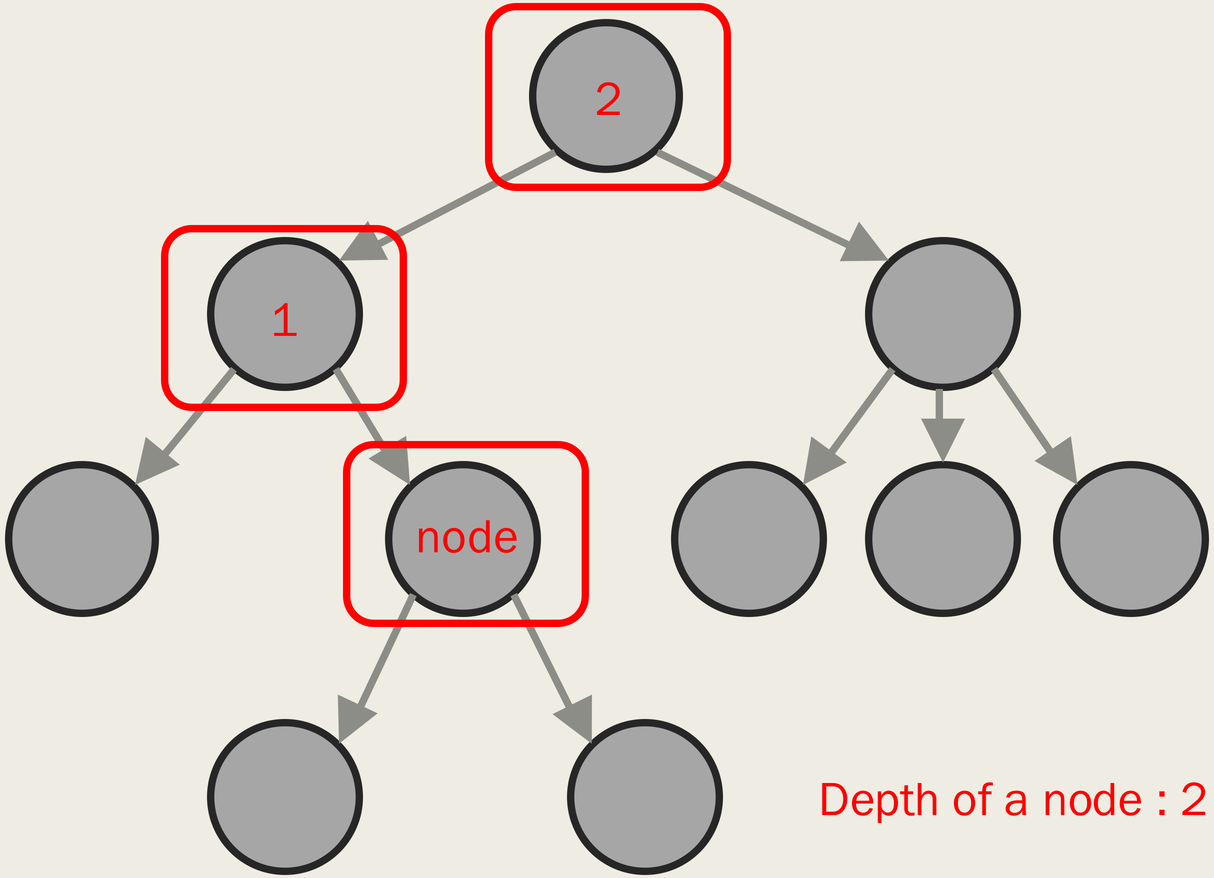
1. Root node
2. Parent node
3. Child node
4. Ancestor node
5. Sibling node
6. Leaf node
7. Height
8. Depth (level)
9. Breadth
10. Degree

Tree



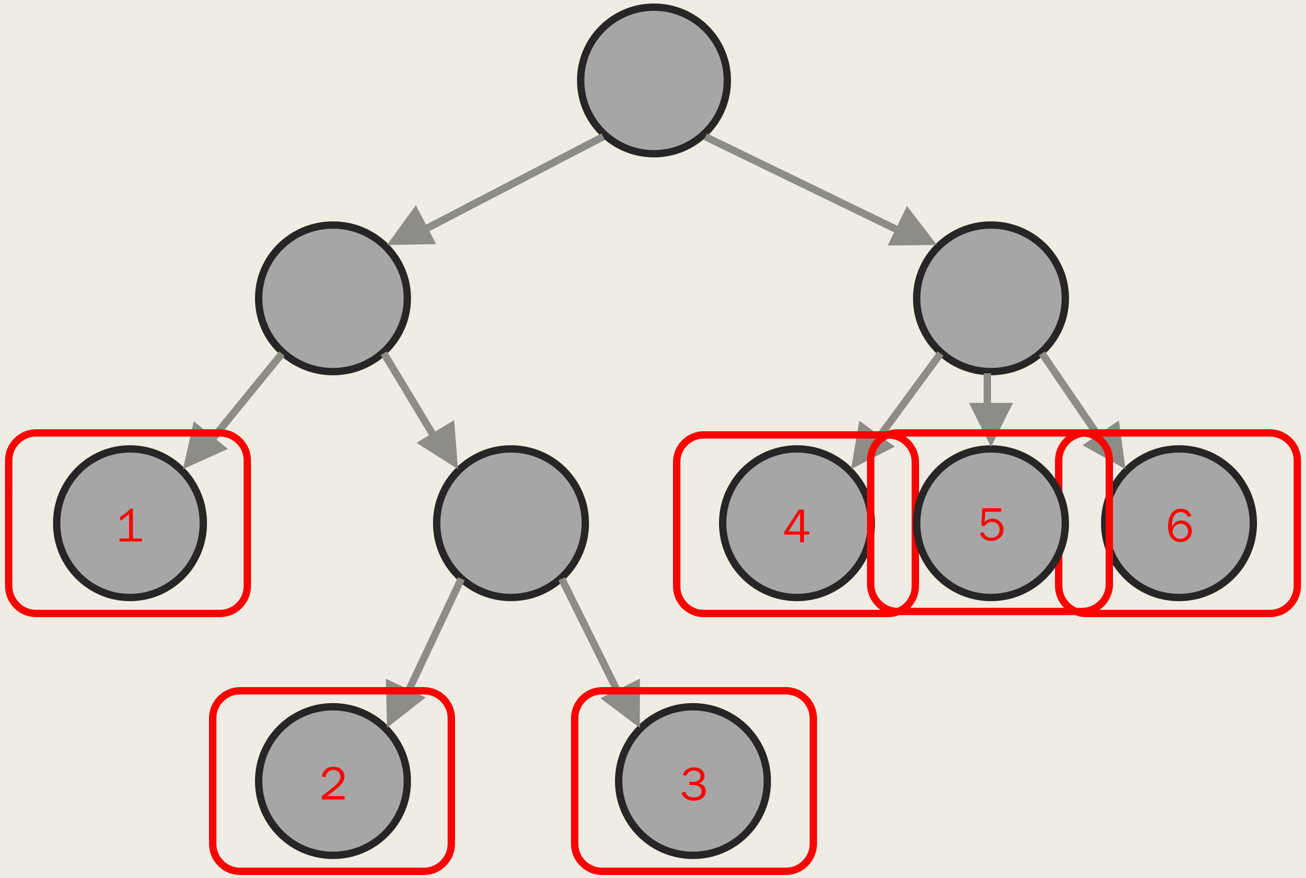
1. Root node
2. Parent node
3. Child node
4. Ancestor node
5. Sibling node
6. Leaf node
7. Height
8. Depth (level)
9. Breadth
10. Degree

Tree



1. Root node
2. Parent node
3. Child node
4. Ancestor node
5. Sibling node
6. Leaf node
7. Height
8. Depth (level)
9. Breadth
10. Degree

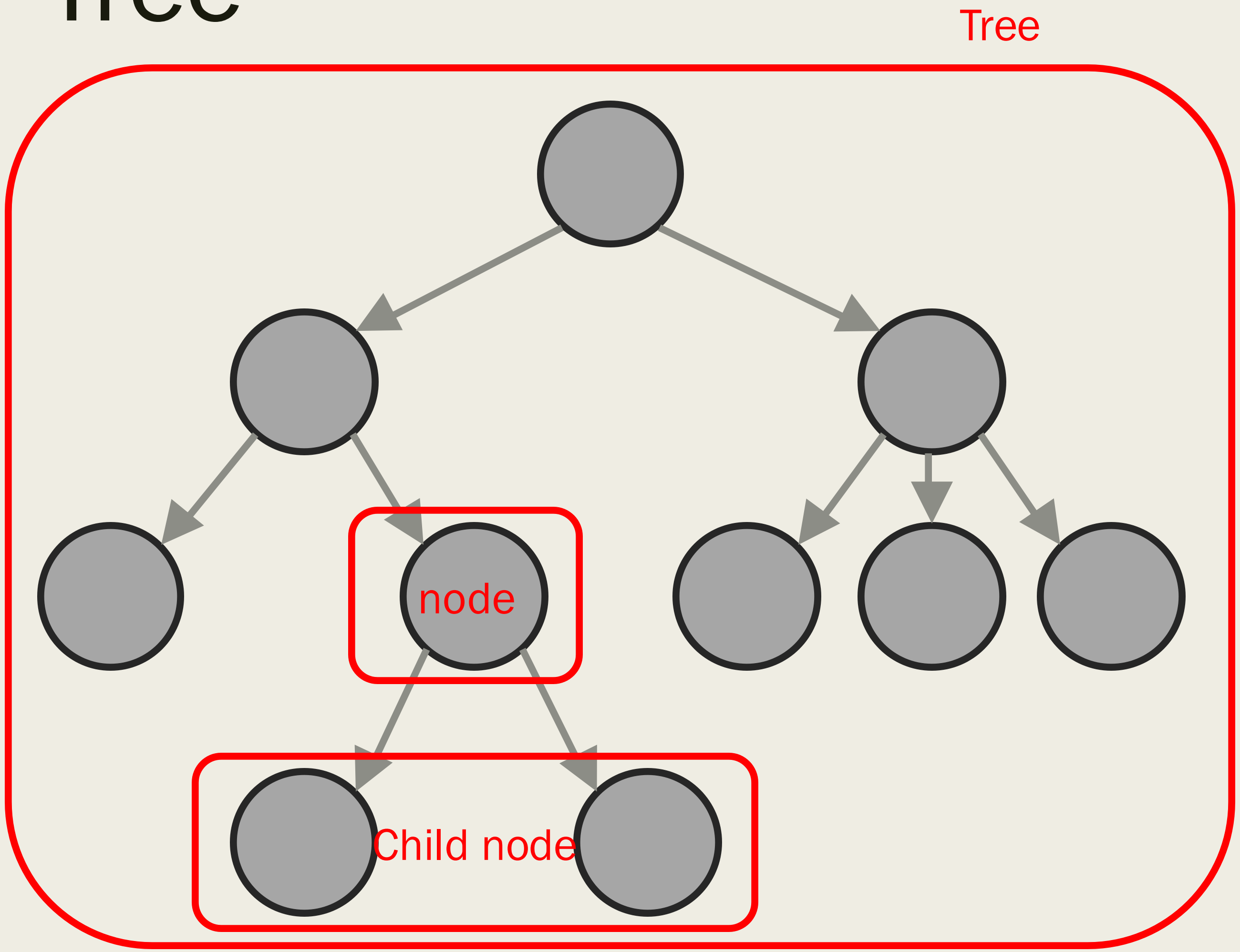
Tree



Breadth(the number of leaves) : 6

1. Root node
2. Parent node
3. Child node
4. Ancestor node
5. Sibling node
6. Leaf node
7. Height
8. Depth (level)
9. Breadth
10. Degree

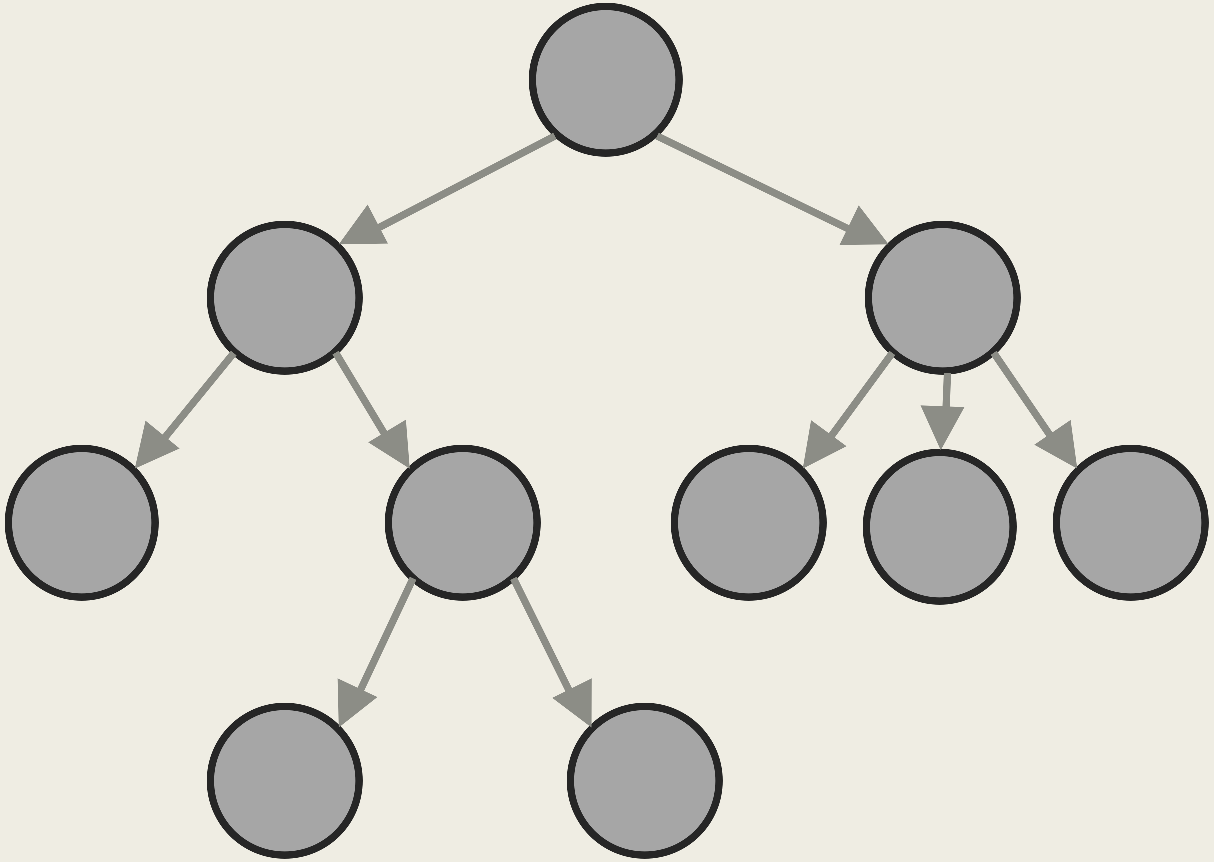
Tree



Degree of a node(the number of children) : 2
Degree of tree(maximum degree of a node in tree) : 3

1. Root node
2. Parent node
3. Child node
4. Ancestor node
5. Sibling node
6. Leaf node
7. Height
8. Depth (level)
9. Breadth
10. Degree

Binary Tree



Binary Tree : degree of tree = 2

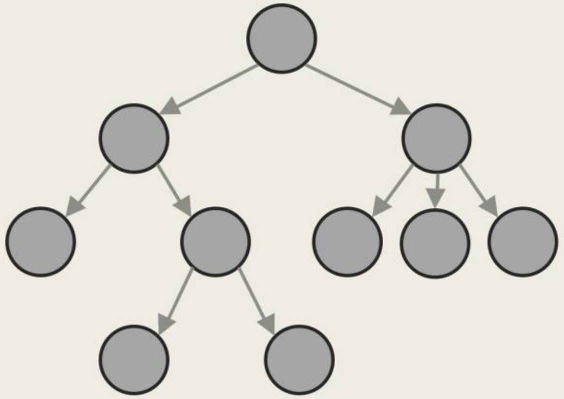
Complete Binary Tree

Complete Binary Tree 조건

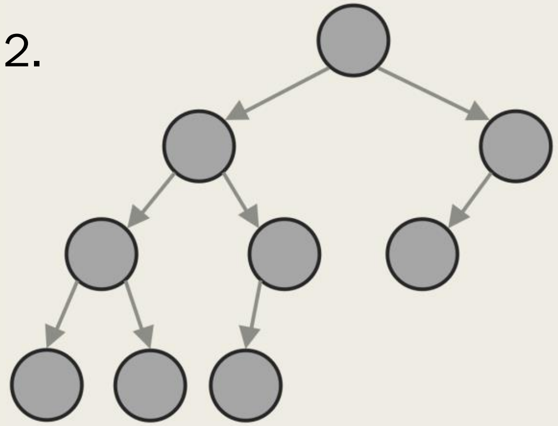
1. Binary Tree여야 한다.
2. 마지막 level을 제외하고는 다 채워져 있어야 한다.
3. 마지막 level은 다 채워져 있을 필요는 없지만 왼쪽에서 채워져 있어야 한다.

Complete Binary Tree

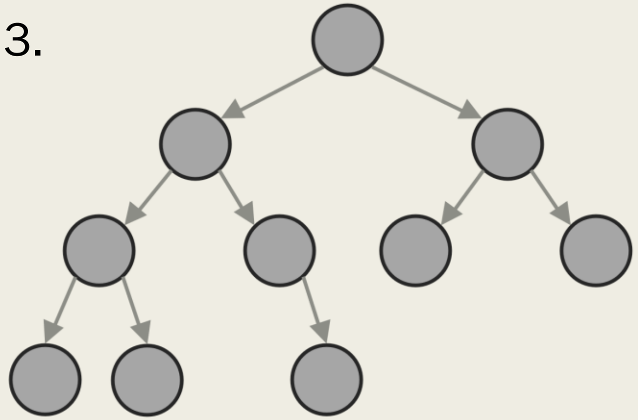
1.



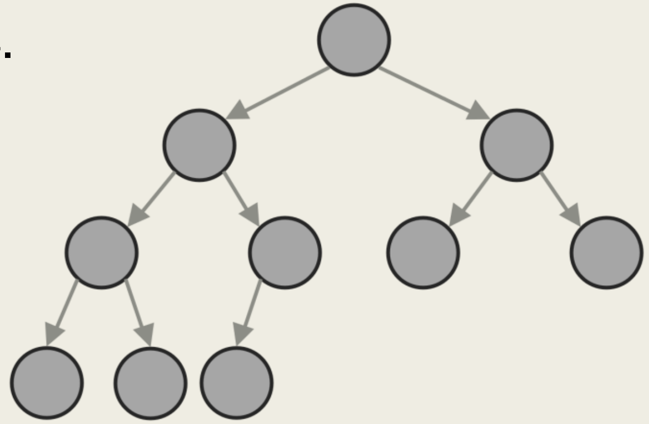
2.



3.



4.



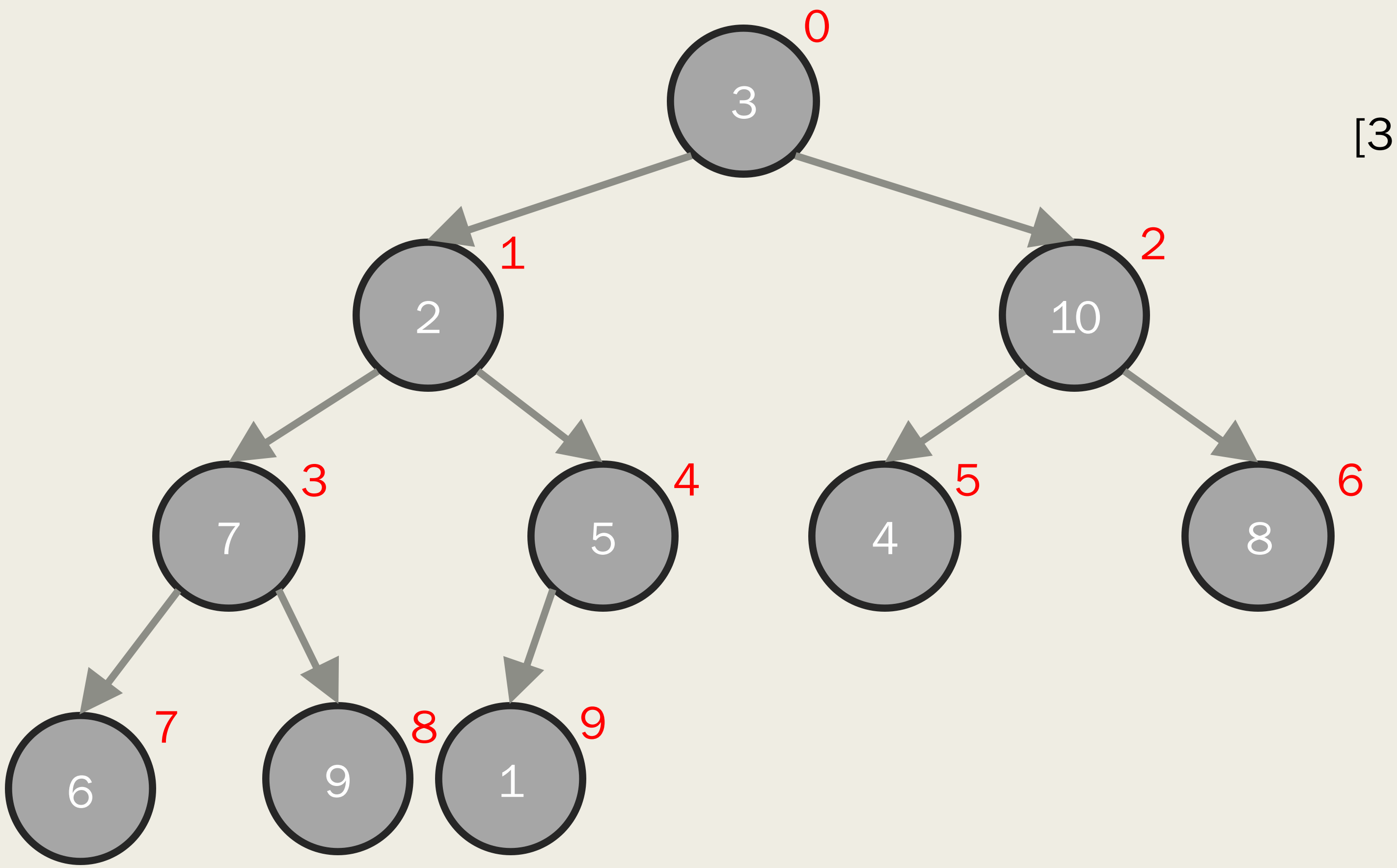
Complete Binary Tree

Complete Binary Tree 장점

1. List로 표현이 가능
2. 내 자식 노드를 바로 탐색 가능

Complete Binary Tree

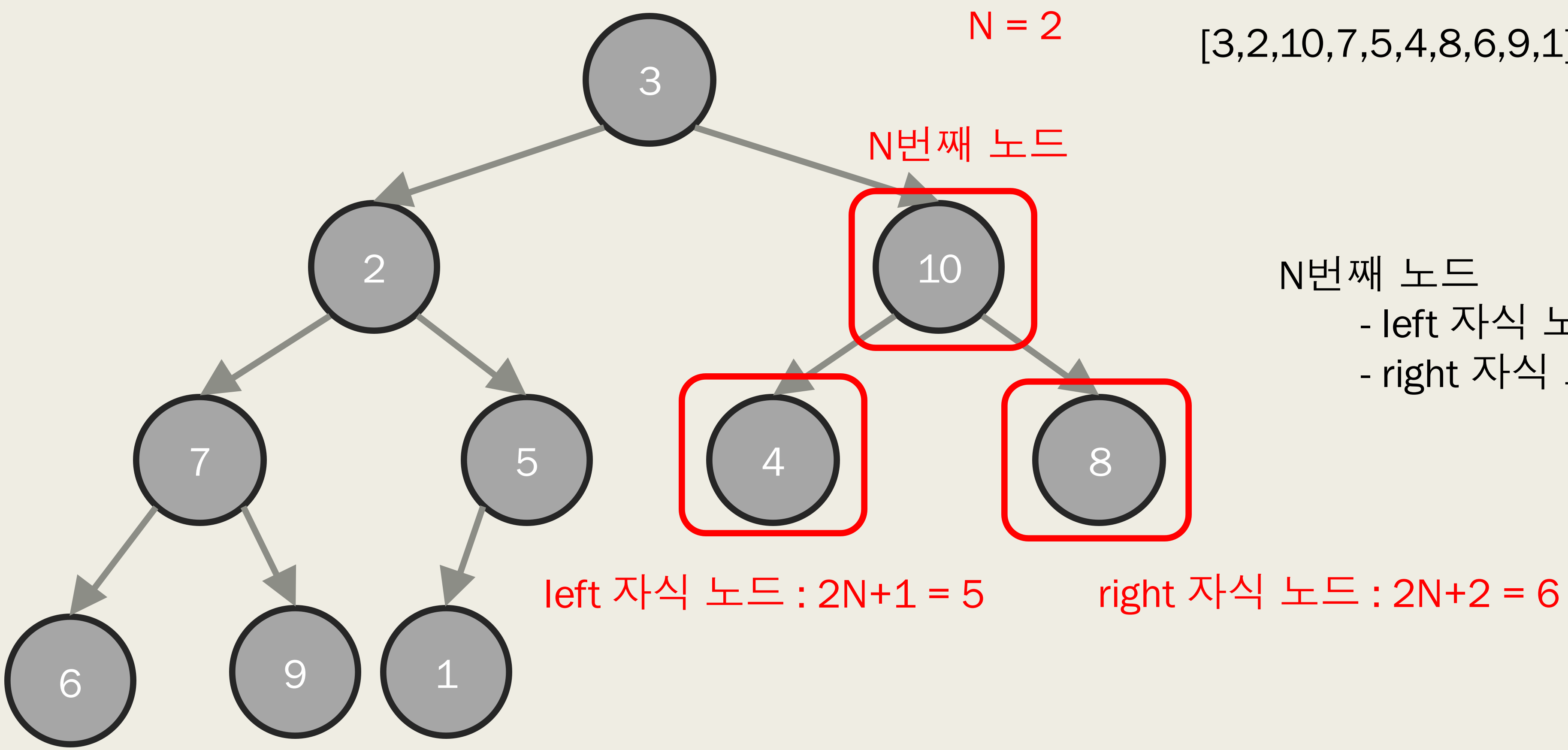
1. List로 표현이 가능



[3,2,10,7,5,4,8,6,9,1]

Complete Binary Tree

2. 내 자식 노드를 바로 탐색 가능

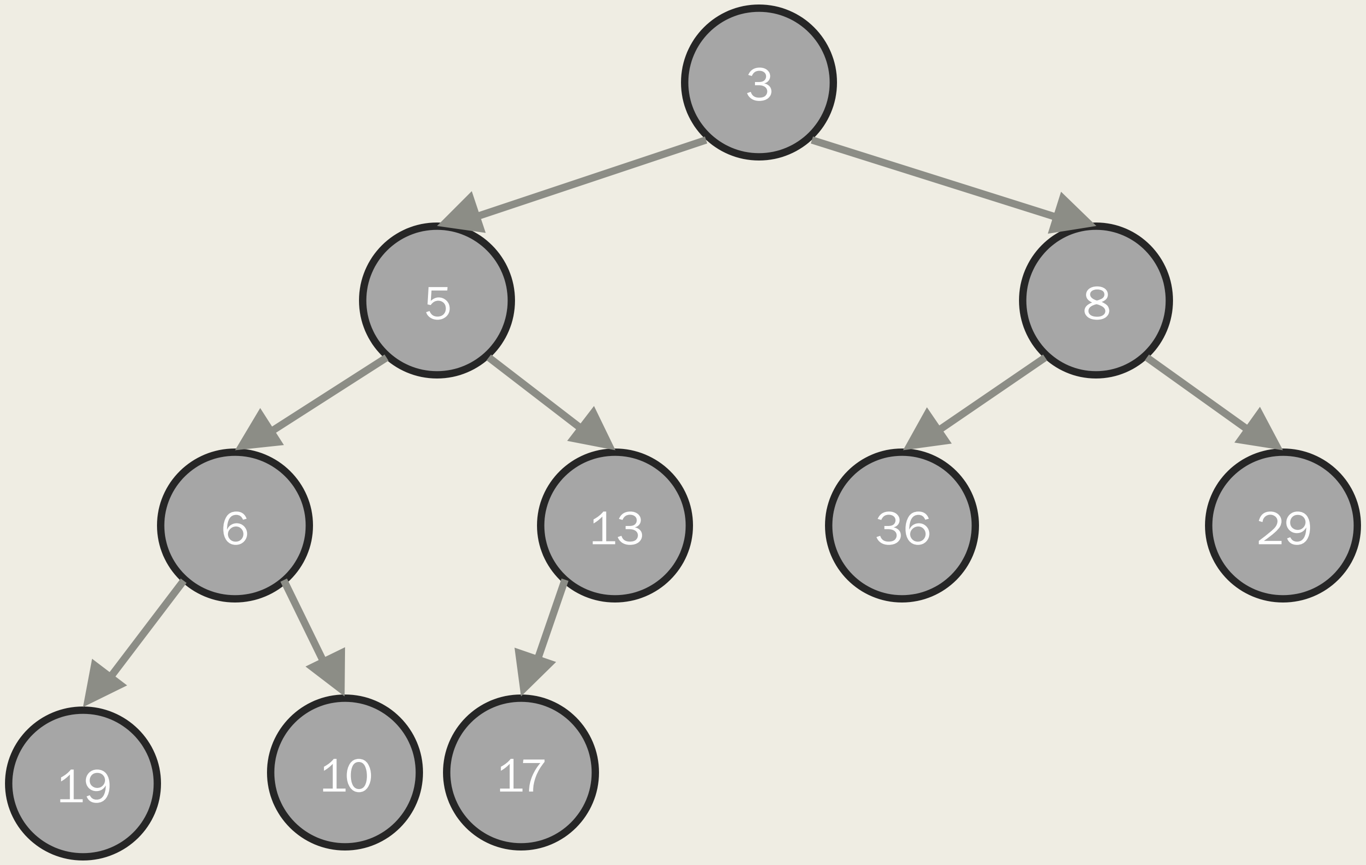


[3,2,10,7,5,4,8,6,9,1]

N번째 노드

- left 자식 노드 : 2N+1
- right 자식 노드 : 2N+2

Heap

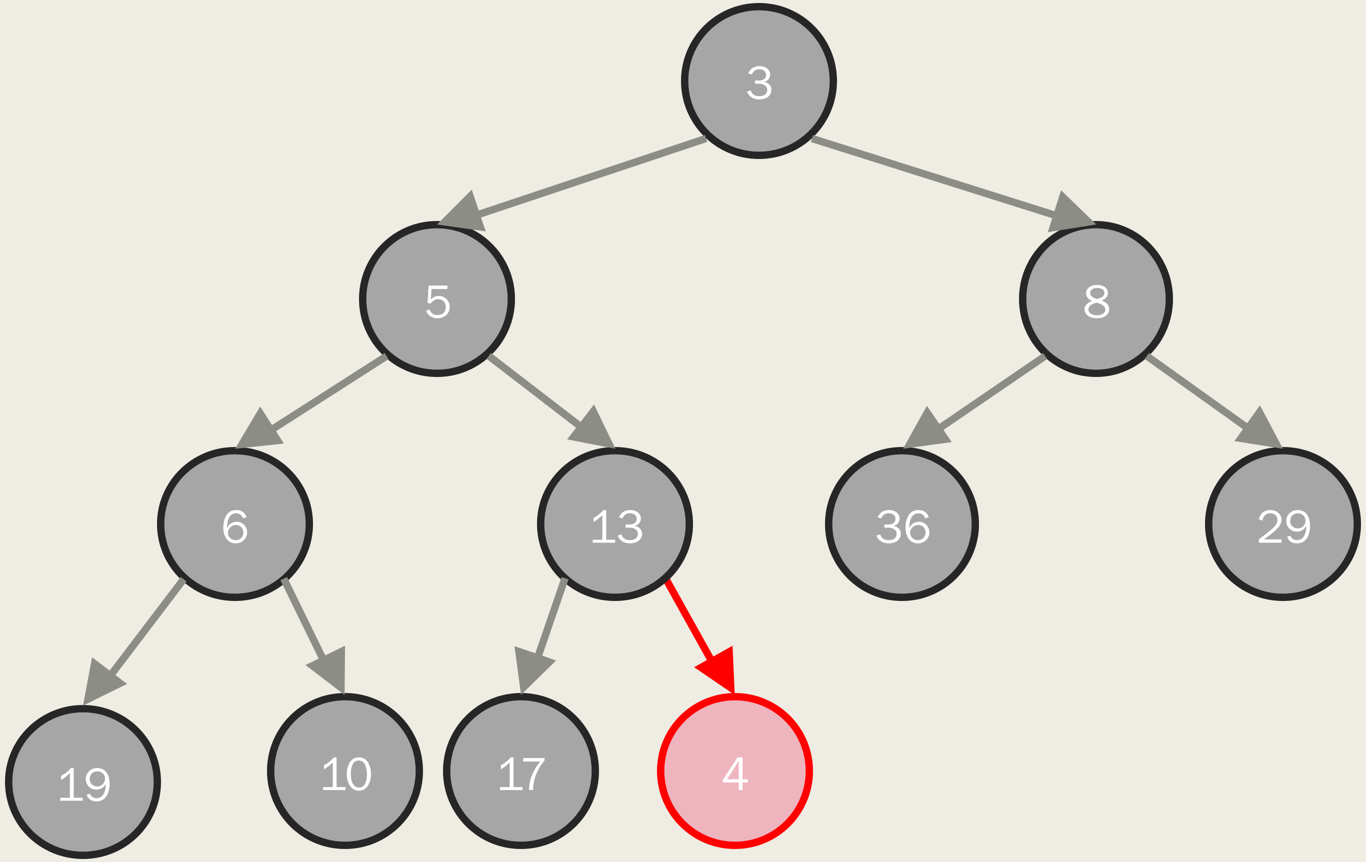


[3, 5, 8, 6, 13, 36, 29, 19, 10, 17]

Heap 특징

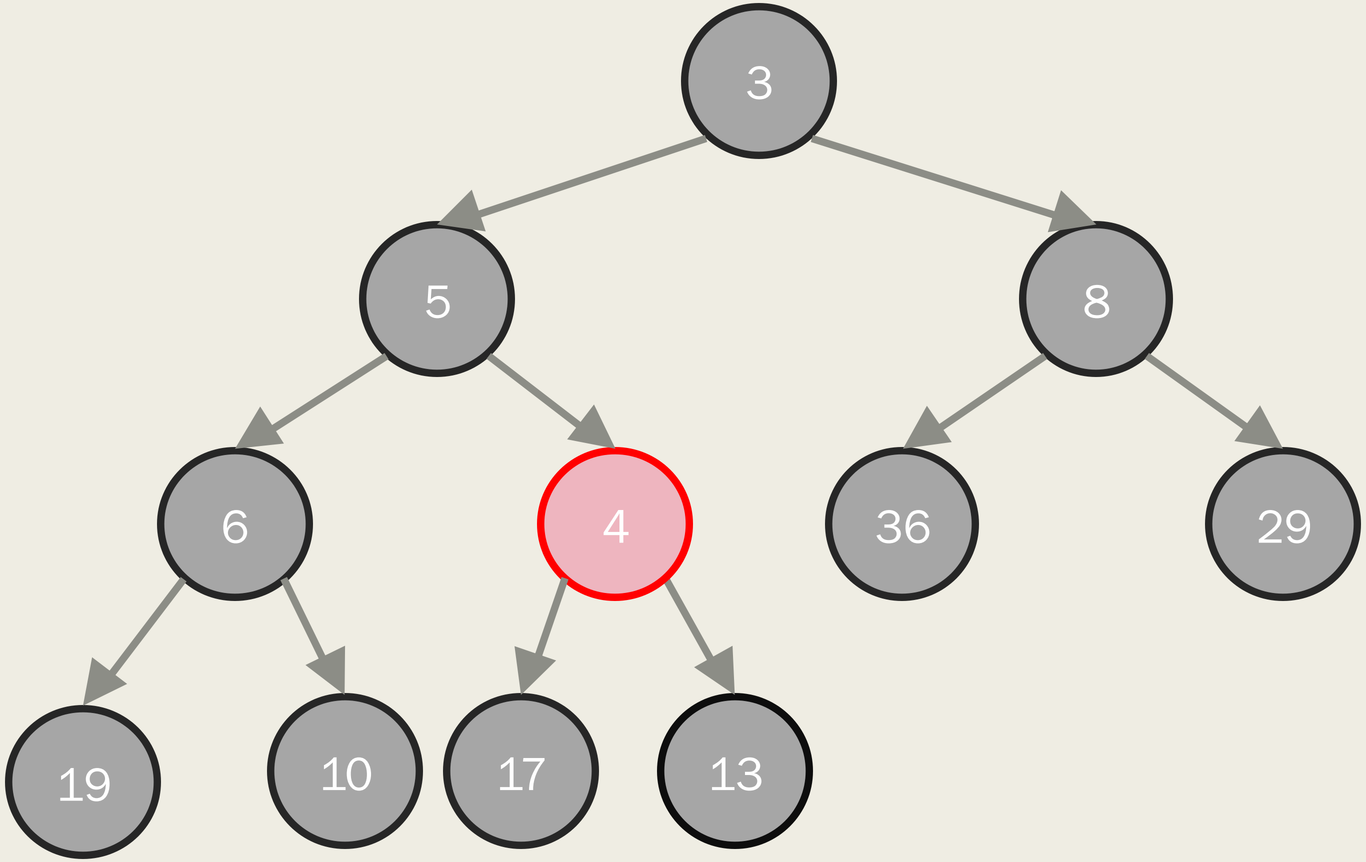
1. 항상 0번째 index에 최소값 저장
2. 그 노드의 자식들은 항상 노드의 값보다 큰 값을 가진다.
3. 삭제, 삽입 $O(logn)$에 처리

Heap - 삽입



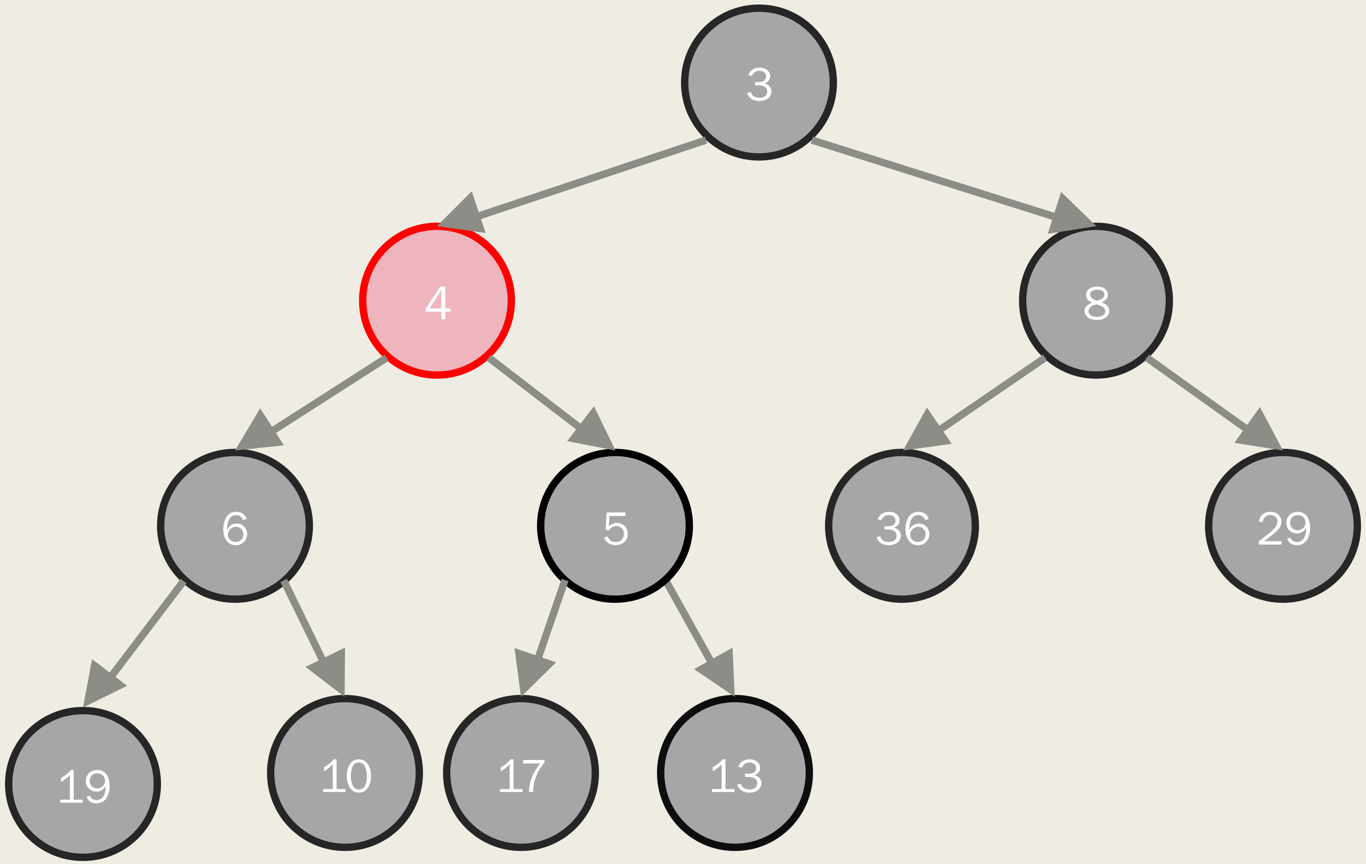
- 삽입할 값을 맨 뒤에 삽입
- 그 다음 그의 부모와 값 비교

Heap - 삽입



- 삽입할 값을 맨 뒤에 삽입
- 그 다음 그의 부모와 값 비교
- 부모보다 작을 경우 교체

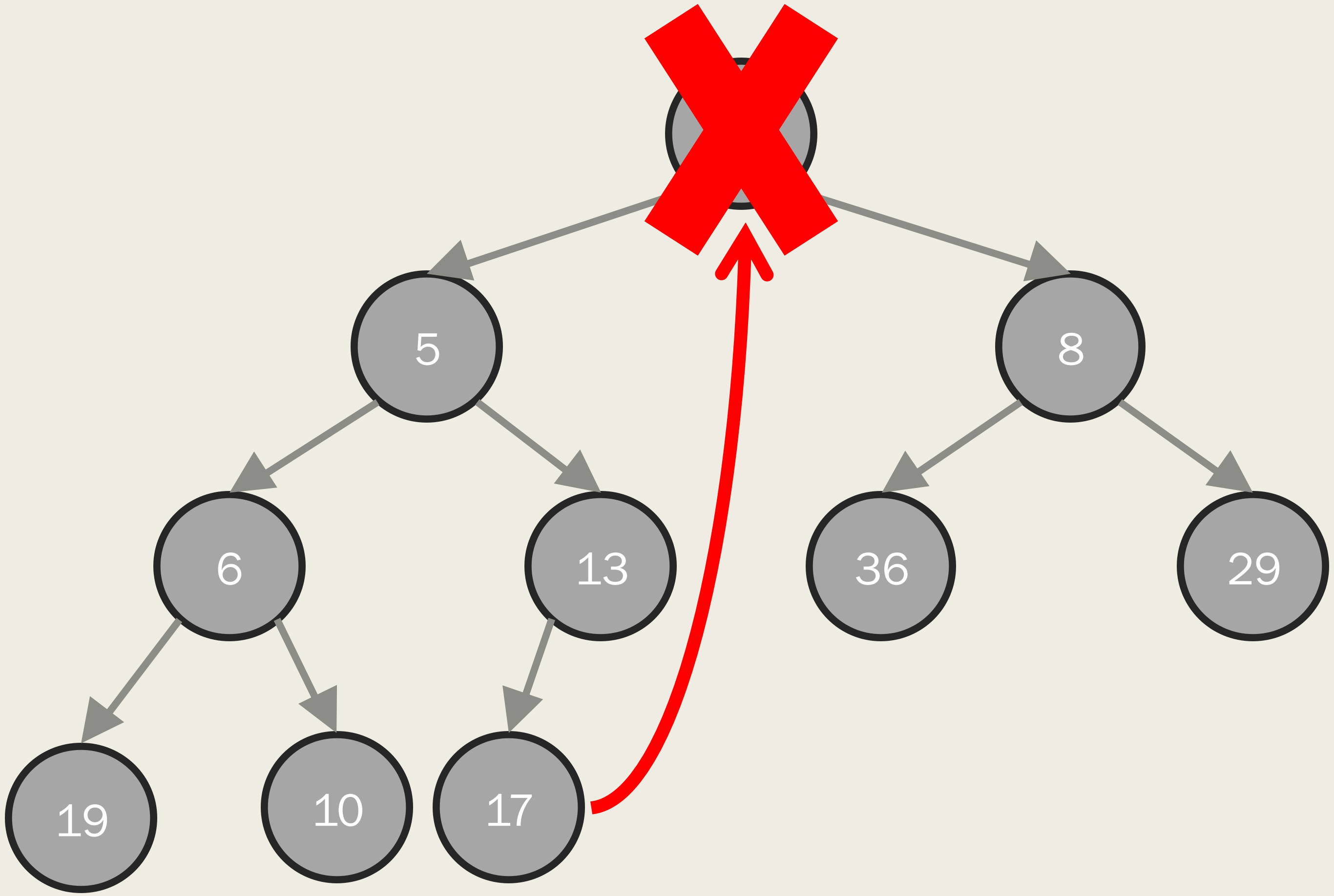
Heap - 삽입



- 삽입할 값을 맨 뒤에 삽입
- 그 다음 그의 부모와 값 비교
- 부모보다 작을 경우 교체
- 부모보다 클 경우 정지

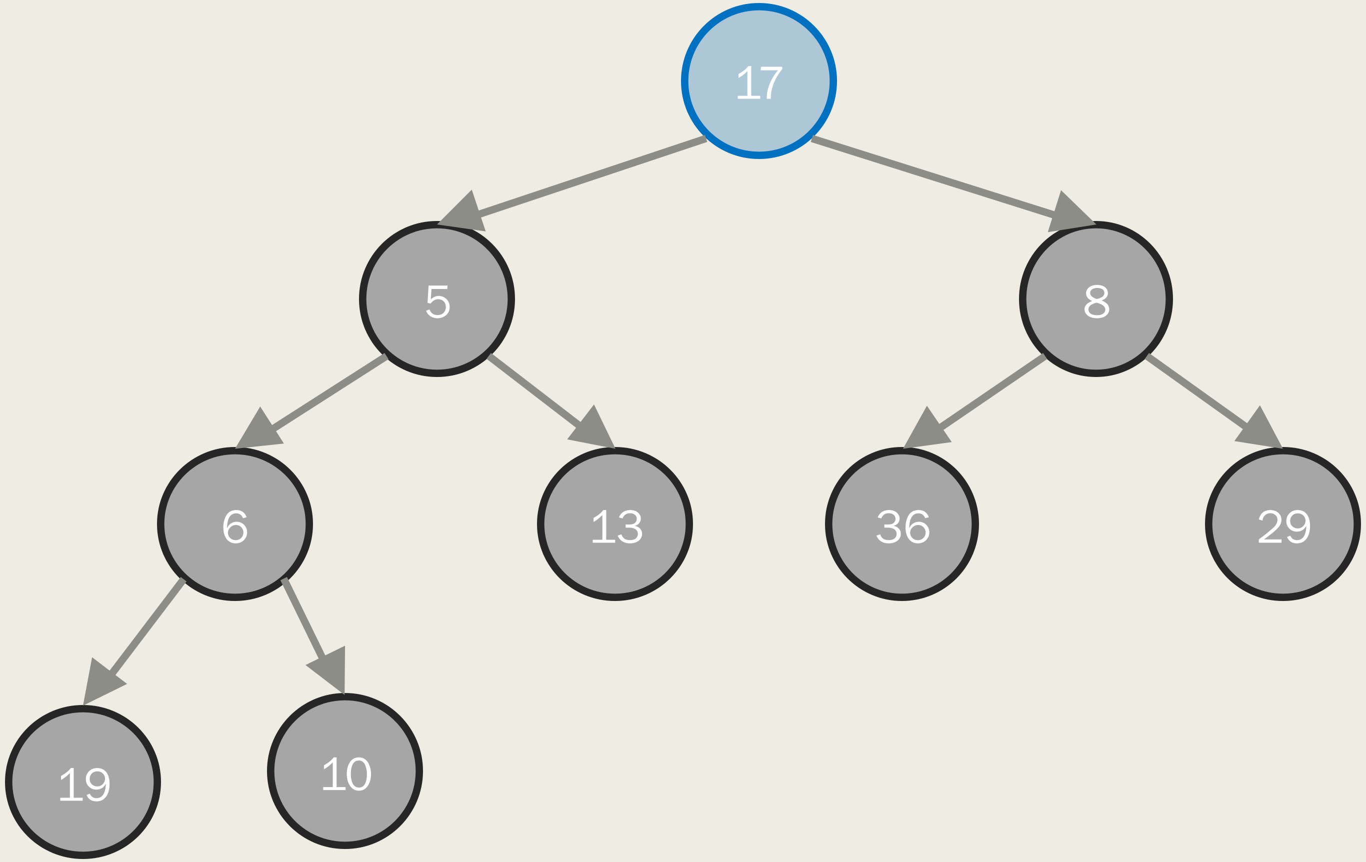
- [3,4,8,6,5,36,29,19,10,17,13]

Heap - 삭제



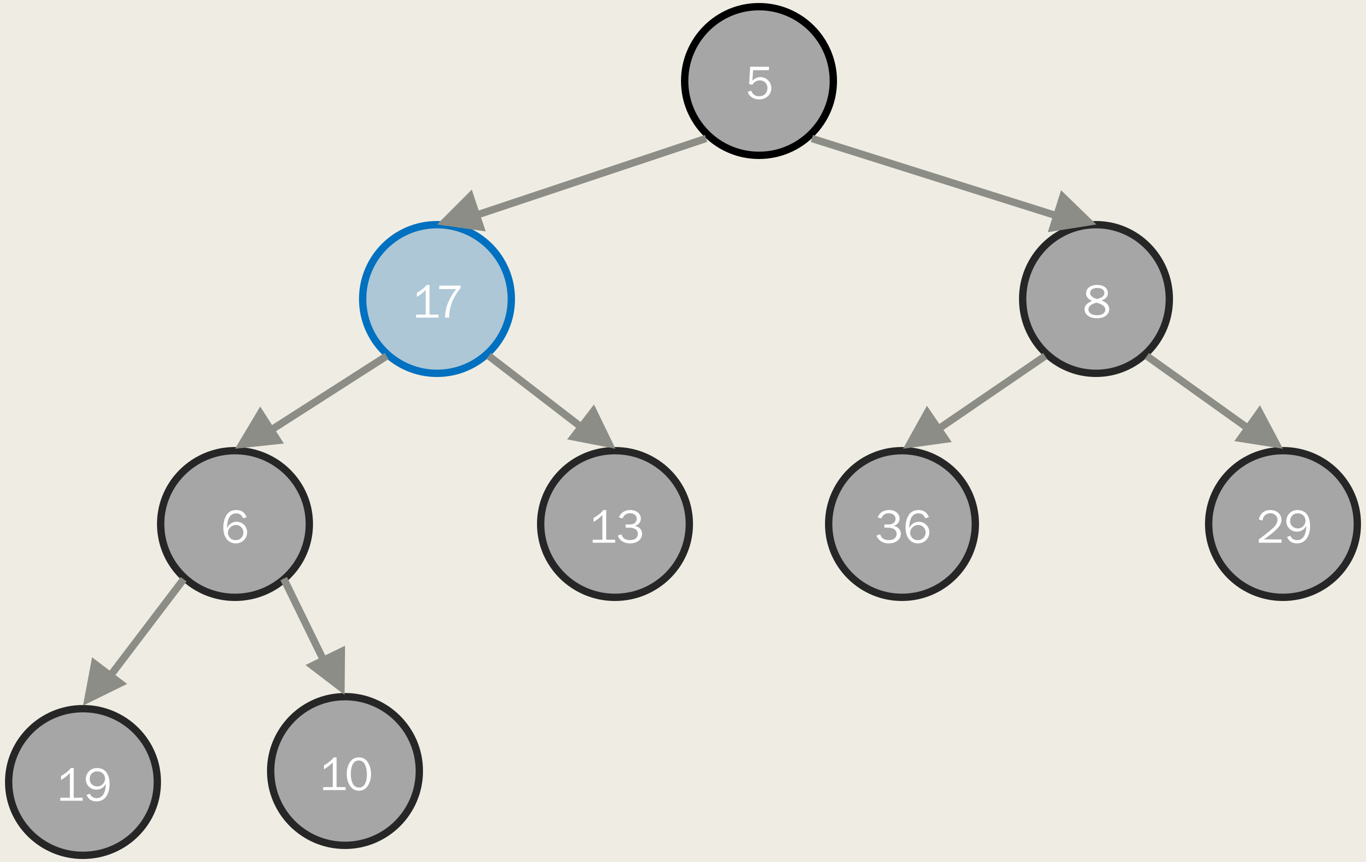
- 삭제할 값을 삭제 후 그 자리에 맨 뒤에 값을 넣어 줌

Heap - 삭제



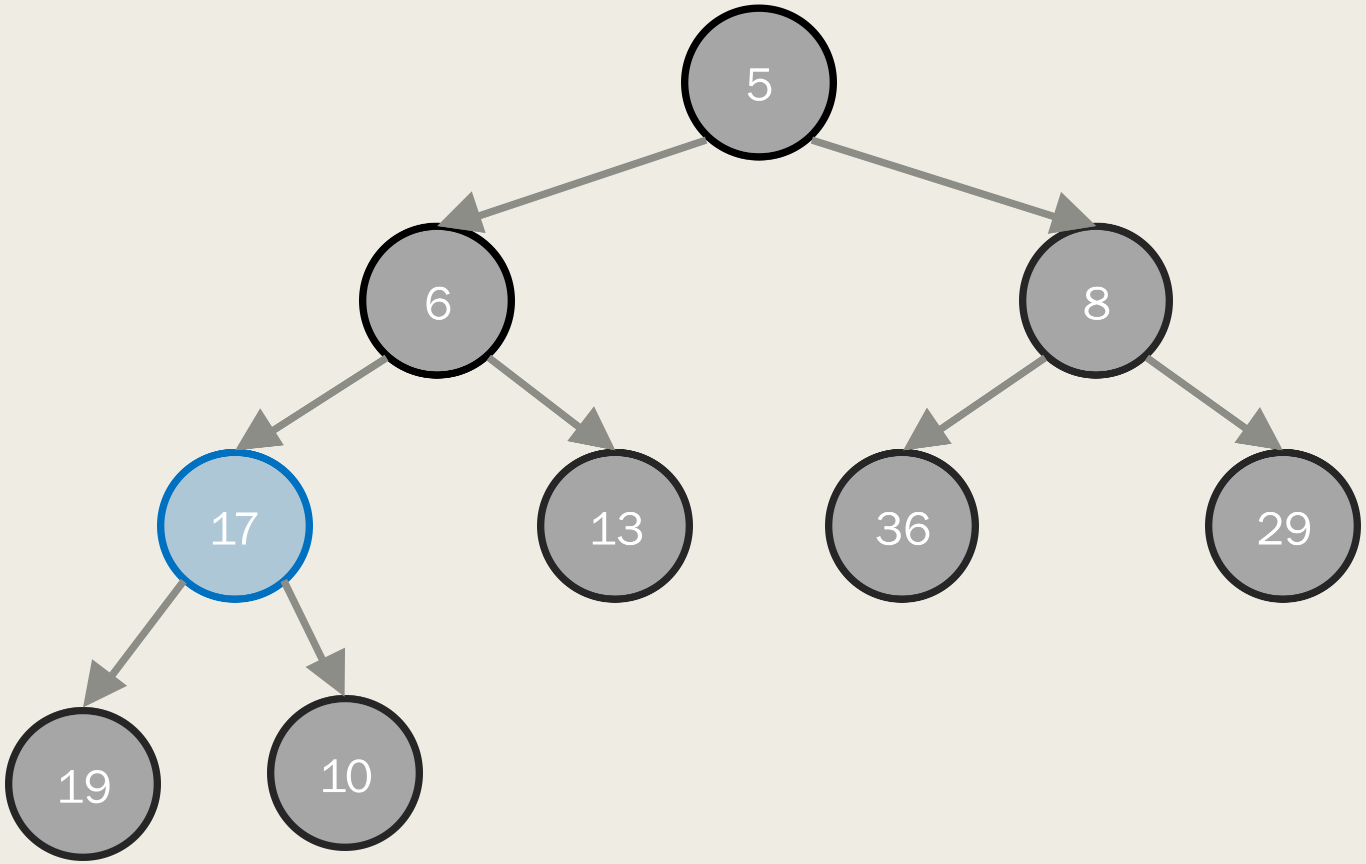
- 삭제할 값을 삭제 후 그 자리에 맨 뒤에 값을 넣어 줌
- 그 다음 그의 왼쪽 자식과 값 비교

Heap - 삭제



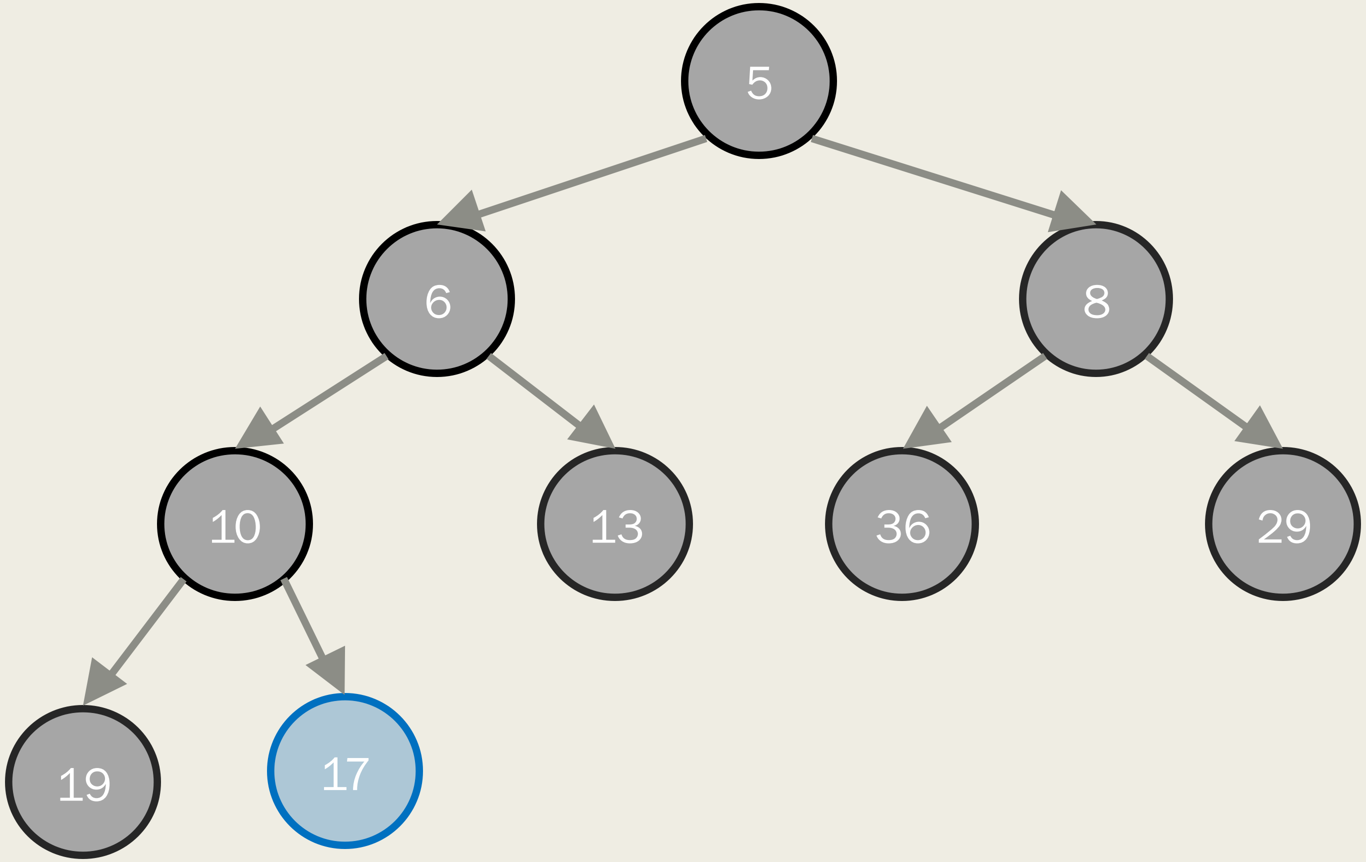
- 삭제할 값을 삭제 후 그 자리에 맨 뒤에 값을 넣어 줌
- 그 다음 그의 왼쪽 자식과 값 비교
- 왼쪽 자식보다 클 경우 교체

Heap - 삭제



- 삭제할 값을 삭제 후 그 자리에 맨 뒤에 값을 넣어 줌
- 그 다음 그의 왼쪽 자식과 값 비교
- 왼쪽 자식보다 클 경우 교체
- 왼쪽 자식보다 작을 경우 그의 오른쪽 자식과 값 비교

Heap - 삭제



- 삭제할 값을 삭제 후 그 자리에 맨 뒤에 값을 넣어 줌
- 그 다음 그의 왼쪽 자식과 값 비교
- 왼쪽 자식보다 클 경우 교체
- 왼쪽 자식보다 작을 경우 그의 오른쪽 자식과 값 비교
- 오른쪽 자식보다 클 경우 교체
- 오른쪽 자식보다 작을 경우 정지

- [5,6,8,10,13,36,29,19,17]

Heap

- 직접 구현할 필요는 없다.
- 이 역시 파이썬은 강력한 라이브러리가 있기에 import만 해주면 된다.
- import heapq
- heapq.heapify(A)   # A : List
- heapq.heappop(A)   # A : List
- heapq.heappush(A,item)   # A : List, item : 넣을 값

Heap

```python
import heapq

print("---------------------------------")
print("Heap - 삽입 (4)")

A = [3, 5, 8, 6, 13, 36, 29, 19, 10, 17]
print(A)

heapq.heappush(A,4)
print(A)

print("---------------------------------")
print("Heap - 삭제")

A = [3, 5, 8, 6, 13, 36, 29, 19, 10, 17]
print(A)

print(heapq.heappop(A))
print(A)
```

```
---------------------------------
Heap - 삽입 (4)
[3, 5, 8, 6, 13, 36, 29, 19, 10, 17]
[3, 4, 8, 6, 5, 36, 29, 19, 10, 17, 13]
---------------------------------
Heap - 삭제
[3, 5, 8, 6, 13, 36, 29, 19, 10, 17]
3
[5, 6, 8, 10, 13, 36, 29, 19, 17]
```

Min Heap

```python
print("--------------------------------")
print("Min Heap")

A = [5,1,4,2,3]
heapq.heapify(A)
print(A)

print(heapq.heappop(A))
print(A)

heapq.heappush(A,-5)
print(heapq.heappop(A))
print(A)
```

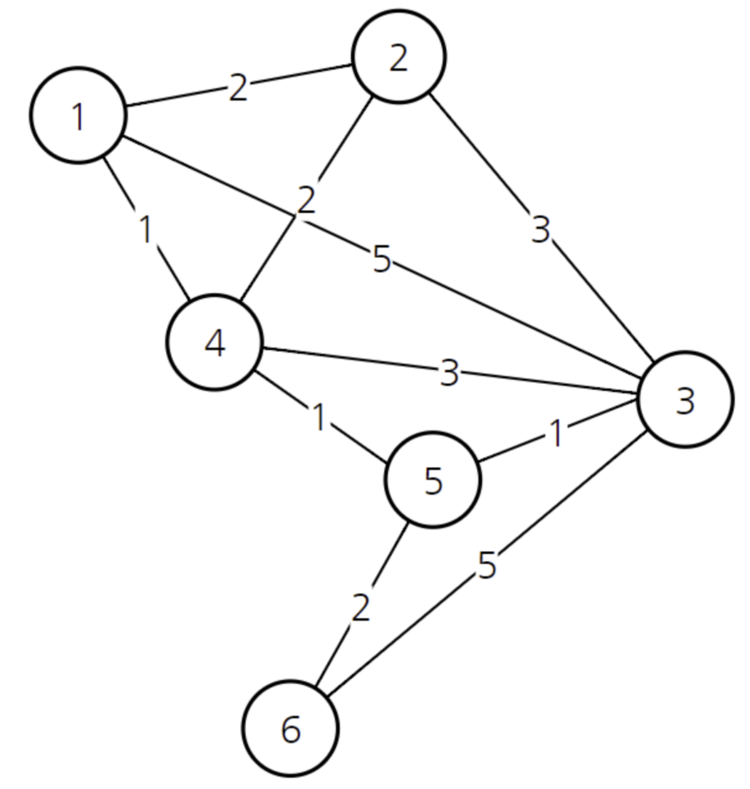
```
--------------------------------
Min Heap
[1, 2, 4, 5, 3]
1
[2, 3, 4, 5]
-5
[2, 3, 4, 5]
```

Max Heap

```python
print("---------------------------------")
print("Max Heap")

A = [5,1,4,2,3]
for i in range(len(A)):
    A[i] *= -1
heapq.heapify(A)
print(A)

print(-heapq.heappop(A))
print(A)

heapq.heappush(A,-100)
print(-heapq.heappop(A))
print(A)
```

```
-----------------------------------
Max Heap
[-5, -3, -4, -2, -1]
5
[-4, -3, -1, -2]
100
[-4, -3, -1, -2]
```

Weighted Graph



- 모든 간선에 가중치가 있는 그래프를 weighted graph라고 한다.
- 이 Graph 또한 dictionary를 이용하여 만들 수 있다.
- {1:[(2,2),(5,3),(1,4)]}

{1 : [(2,2),(5,3),(1,4)], 2 : [(2,1),(3,3),(2,4)], 3 : [(5,1),(3,2),(3,4),(1,5),(5,6)],
4 : [(1,1),(2,2),(3,3),(1,5)], 5 : [(1,3),(1,4),(2,6)], 6 : [(5,3),(2,5)]}

Weighted Graph

- 백준에서 입력은 보통 node 수(N)와 edge 수(M)를 공백으로 구분하여 먼저 입력
- M개의 줄에 걸쳐 u, v, w를 공백으로 구분하여 입력해준다.
- 이때 node u와 node v가 가중치 w인 edge로 이어져 있다는 뜻

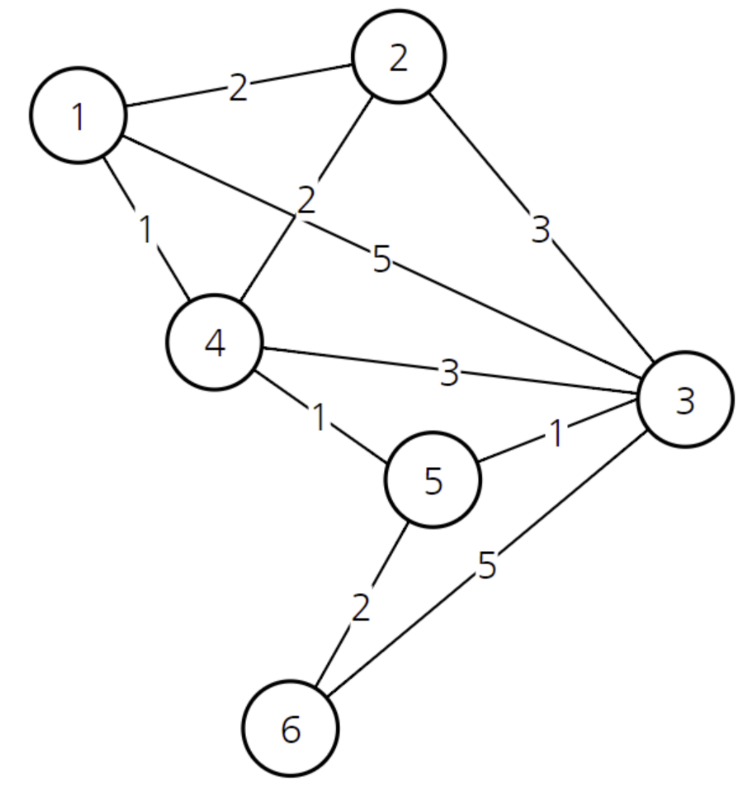
- Ex)

```
6 10
1 2 2
1 3 5
1 4 1
2 3 3
2 4 2
3 4 3
3 5 1
3 6 5
4 5 1
5 6 2
```

```python
N, M = map(int,input().split())
graph = {i:[] for i in range(1,N+1)}

for _ in range(M):
    u, v, w = map(int,input().split())
    graph[u].append((w,v))
    graph[v].append((w,u))

print(graph)
```

```
{1: [(2, 2), (5, 3), (1, 4)], 2: [(2, 1), (3, 3), (2, 4)], 3: [(5, 1), (3, 2), (3, 4), (1, 5), (5, 6)],
 4: [(1, 1), (2, 2), (3, 3), (1, 5)], 5: [(1, 3), (1, 4), (2, 6)], 6: [(5, 3), (2, 5)]}
```

Dijkstra



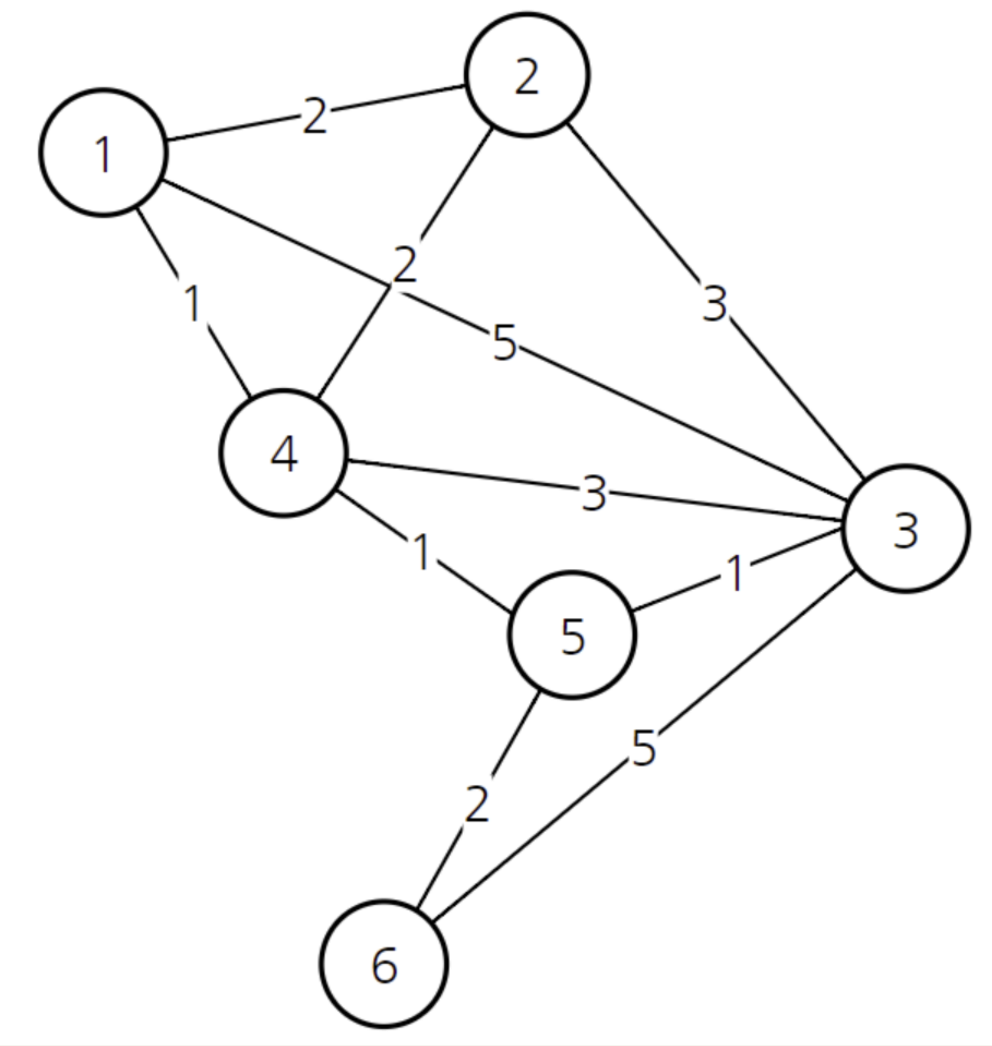
1에서 3으로 가는 최단 경로는?

Dijkstra 알고리즘은 Weighted Graph에서
i지점으로부터 j지점으로 가는 최단거리를
구하는 알고리즘이다.

구하는 방식은 여러가지
다만 Heap을 이용하는 것이 일반적

구동방식
1. 출발 노드를 설정합니다.
2. 출발 노드를 기준으로 각 노드의 최소 비용을 저장합니다.
3. 방문하지 않은 노드 중에서 가장 비용이 적은 노드를 선택합니다.
4. 해당 노드를 거쳐서 특정한 노드로 가는 경우를 고려하여 최소 비용을 갱신합니다.
5. 위 과정에서 3번~4번을 반복

Dijkstra



시간 복잡도: $O((V+E)logV)$

비슷한 알고리즘
Bellman-Ford, Floyd-Warshall

실습!!