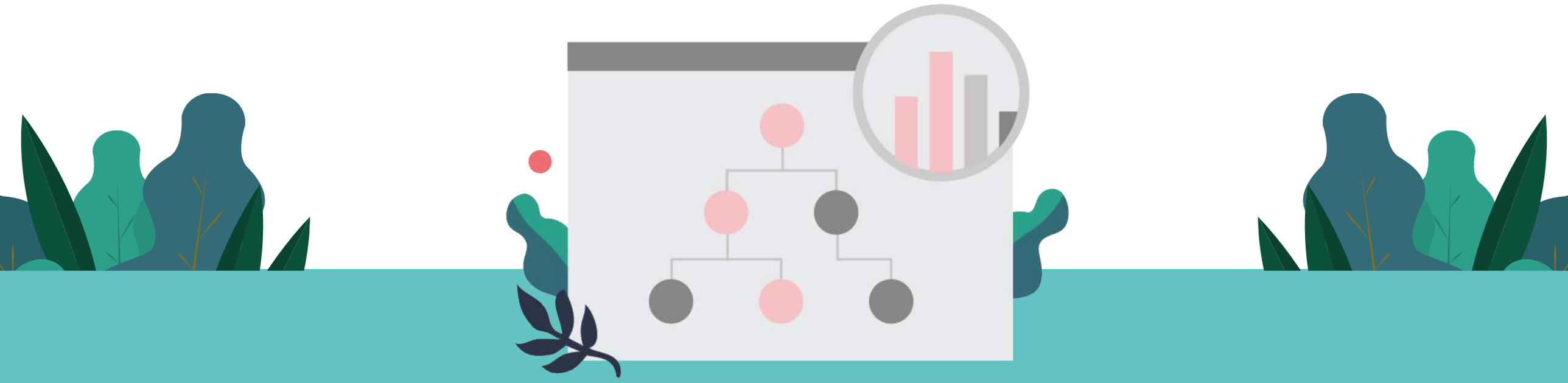
/* 데이터 구조 및 알고리즘 */

신현규 강사, 화/목 20:00

힙



/* elice */

주차별 커리큘럼

1주차	과정 소개, 배열, 연결리스트, 클래스
2주차	스택, 큐, 해싱
3주차	시간복잡도
4주차	트리, 트리순회, 재귀호출
5주차	**힙**
6주차	그래프 소개, DFS
7주차	그래프 심화, BFS
8주차	강의 요약, 알고리즘 과정 소개

샴푸 통 디자인

+ 디자인이 간단

- 쓰기가 다소 불편

+ 쓰기 좀 더 편함

- 하지만 여전히 불편함

+ 쓰기 간편함

- 많은 양을 얻기 힘듦

+ 우주에서 쓸수있음

- 중력이 있으면 굳이..

그래서 스택은 언제 쓰나요 ?

스택은 자료구조

무슨 자료를 저장하는가 ?

상태 (Status)

실생활의 예 : 자료구조 관점

1. 미역
2. 국간장
3. 후추
4. 고기
5. 냄비

1. 열쇠 찾기

1. 문 열기
2. 서랍 열기
3. 카드 꺼내기
4. 집 나오기

실생활의 예 : 자료구조 관점

1. 미역
2. 국간장
3. 후추
4. 고기
5. 냄비

1. 열쇠 찾기

1. 문 열기
2. 서랍 열기
3. 카드 꺼내기
4. 집 나오기

미역국 끓이기

실생활의 예 : 자료구조 관점

1. ~~미역~~
2. ~~국간장~~
3. ~~후추~~
4. 고기
5. 냄비

1. 열쇠 찾기

1. 문 열기
2. 서랍 열기
3. 카드 꺼내기
4. 집 나오기

마트. (4)

미역국 끓이기

실생활의 예 : 자료구조 관점

1. ~~미역~~
2. ~~국간장~~
3. ~~후추~~
4. 고기
5. 냄비

1. 열쇠 찾기

1. ~~문 열기~~
2. 서랍 열기
3. 카드 꺼내기
4. 집 나오기

집. (2)

마트. (4)

미역국 끓이기

실생활의 예 : 자료구조 관점

1. ~~미역~~
2. ~~국간장~~
3. ~~후추~~
4. 고기
5. 냄비

1. ~~열쇠 찾기~~

1. ~~문 열기~~
2. 서랍 열기
3. 카드 꺼내기
4. 집 나오기

집. (2)

마트. (4)

미역국 끓이기

실생활의 예 : 자료구조 관점

1. ~~미역~~
2. ~~국간장~~
3. ~~후추~~
4. 고기
5. 냄비

1. ~~열쇠 찾기~~

1. ~~문 열기~~
2. ~~셔랍 열기~~
3. ~~카드 꺼내기~~
4. ~~집 나오기~~

마트. (4)

미역국 끓이기

실생활의 예 : 자료구조 관점

1. ~~미역~~
2. ~~국간장~~
3. ~~후추~~
4. ~~고기~~
5. ~~냄비~~

1. ~~열쇠 찾기~~

1. ~~문 열기~~
2. ~~셔랍 열기~~
3. ~~카드 꺼내기~~
4. ~~집 나오기~~

미역국 끓이기

실생활의 예 : 자료구조 관점

1. ~~미역~~
2. ~~국간장~~
3. ~~후추~~
4. ~~고기~~
5. ~~냄비~~

1. ~~열쇠 찾기~~

1. ~~문 열기~~
2. ~~셔랍 열기~~
3. ~~카드 꺼내기~~
4. ~~집 나오기~~

그래서 스택은 언제 쓰나요 ?

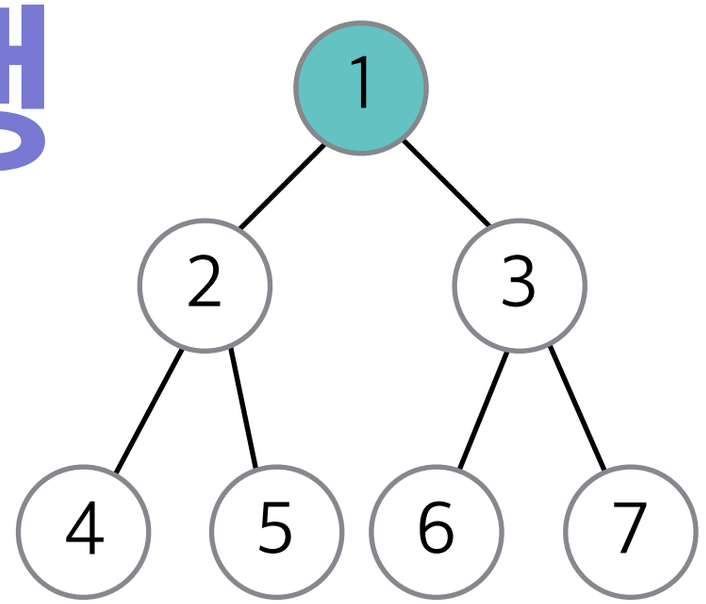
"상태"의 의존상태가 생길 때

A라는 일을 마치기 위해서 B라는 일을 먼저 끝내야 할 때

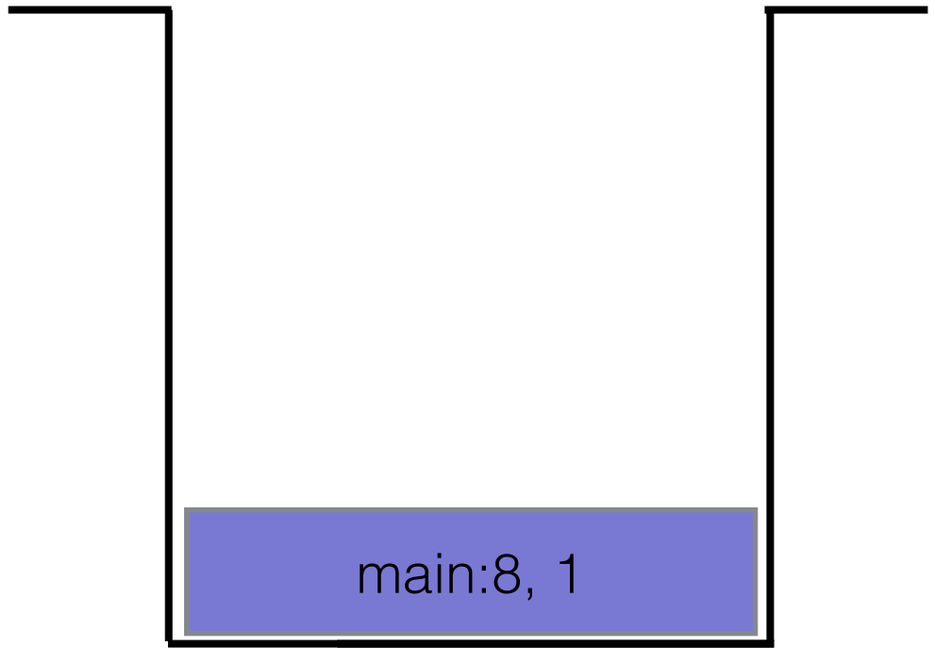
재귀호출

Computational Thinking

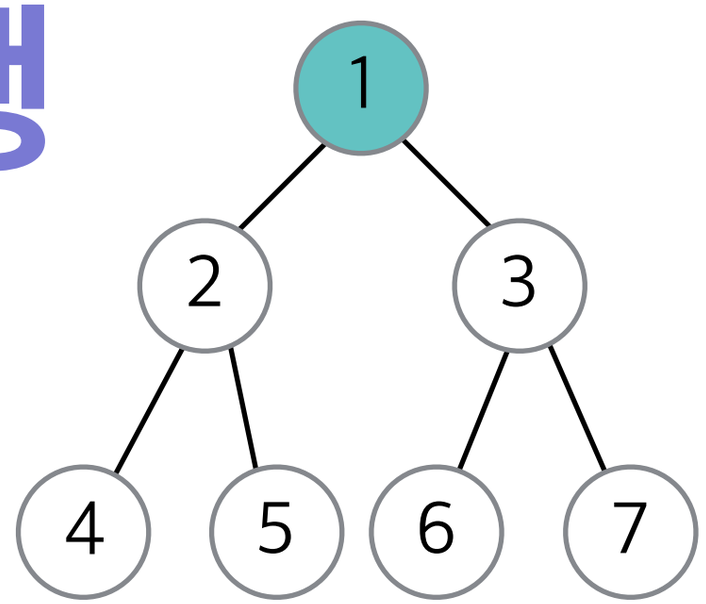
재귀함수의 실행



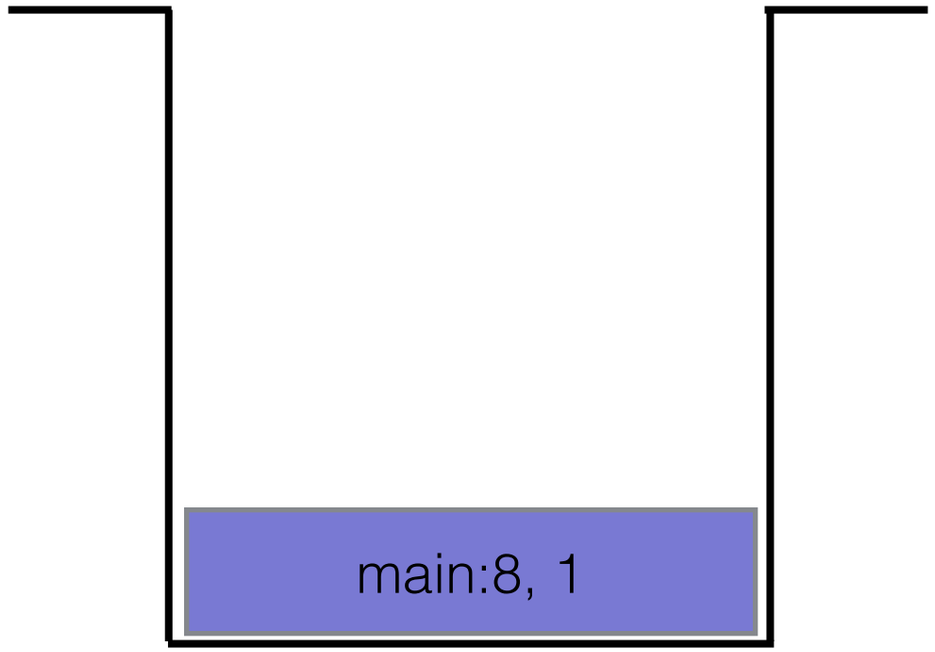

```
1  def postorder(tree) :
2      result = []

3      if tree == None :
4          return []

5      result = postorder(tree.left)
6      result = result + postorder(tree.right)
7      result.append(tree.index)

8      return result
```



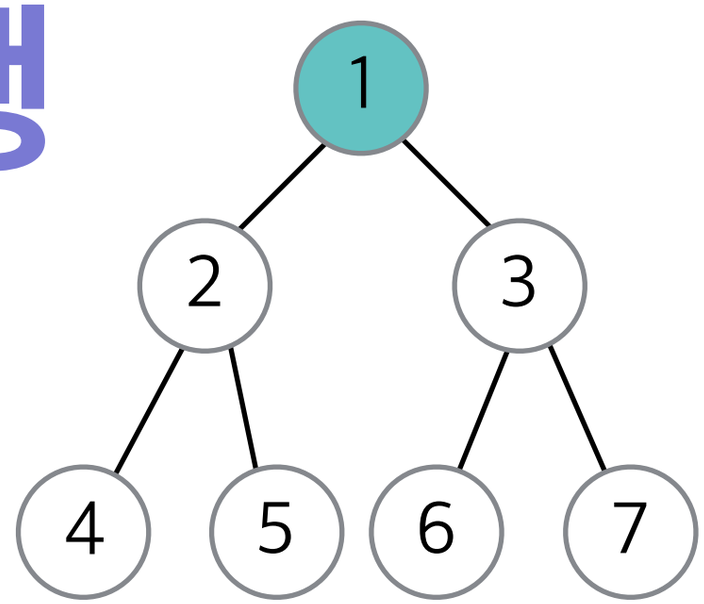
재귀함수의 실행




```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

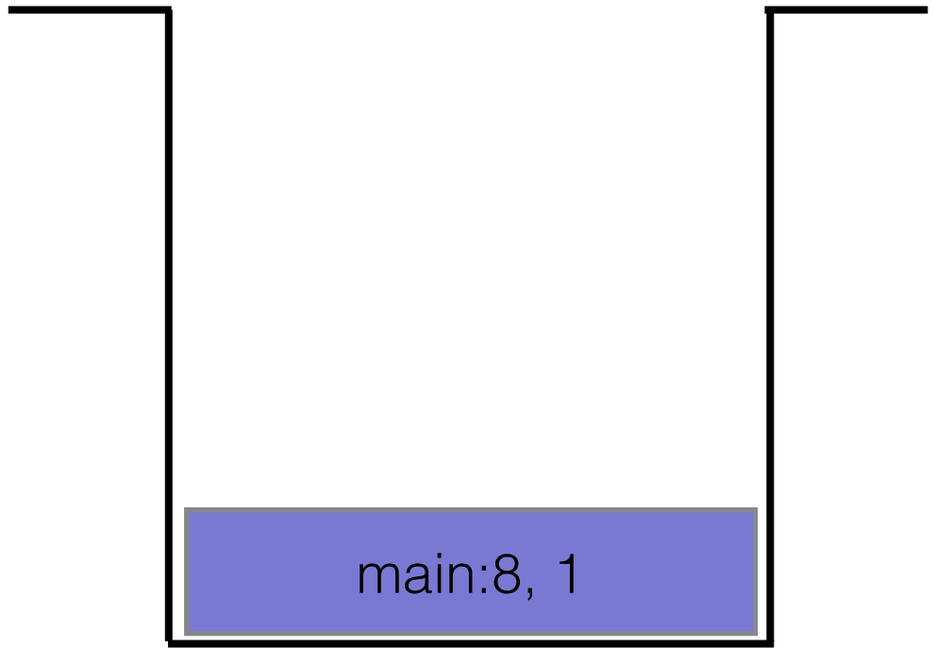


재귀함수의 실행

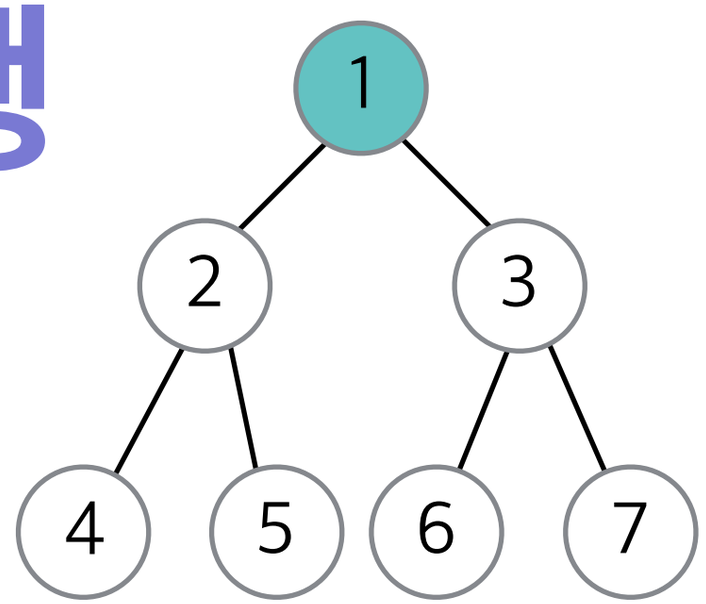



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[]

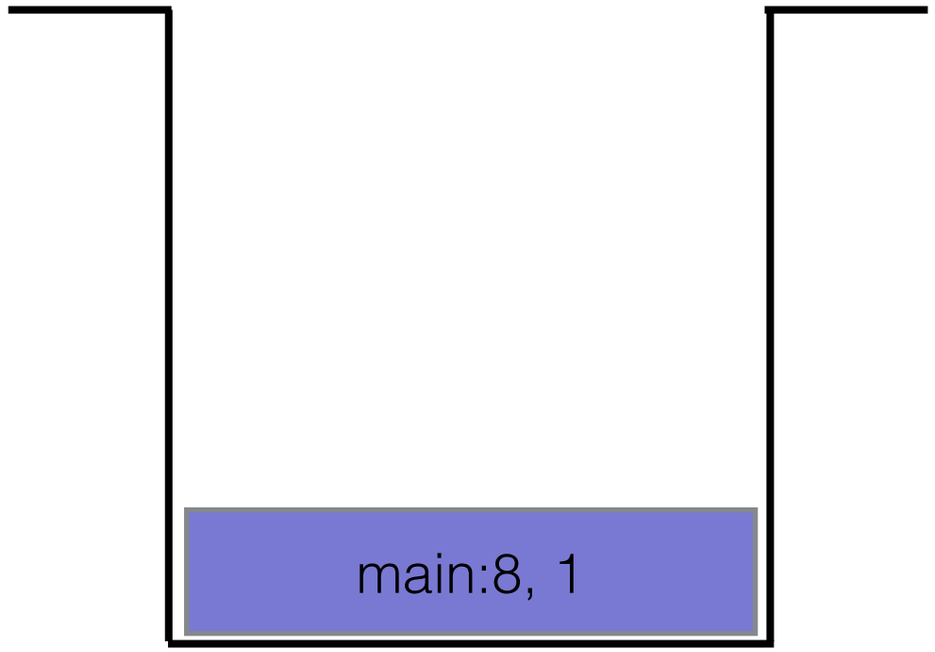


재귀함수의 실행

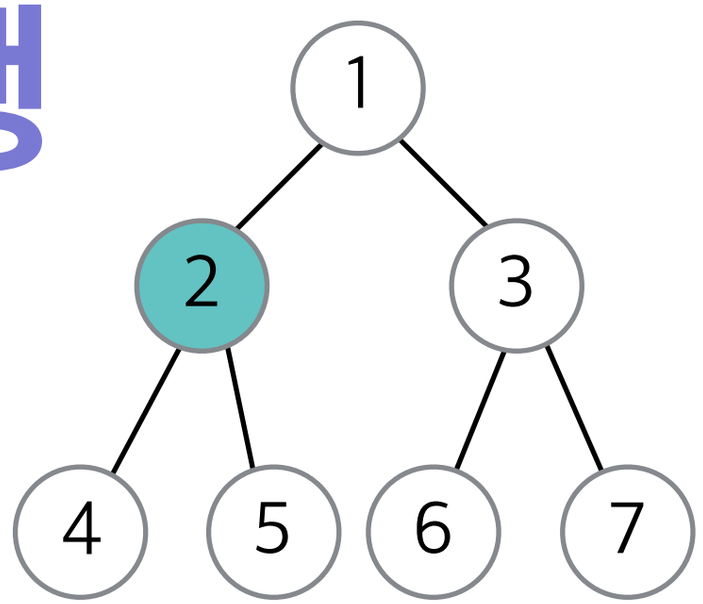



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

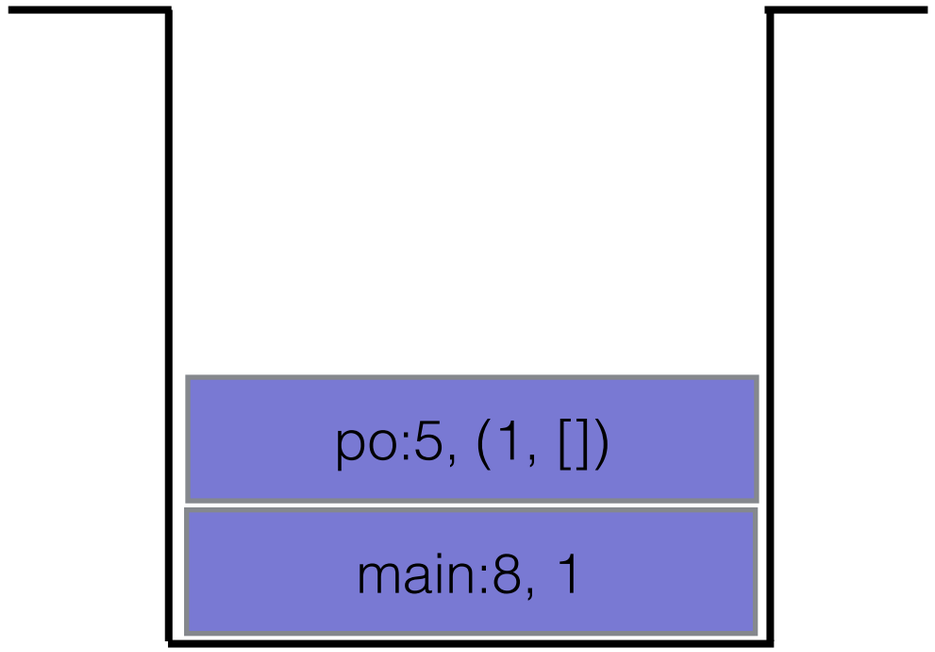
[]



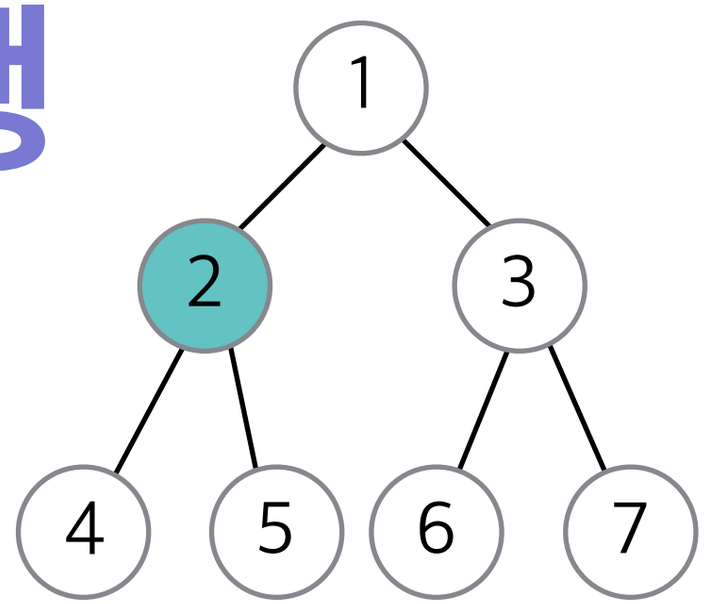
재귀함수의 실행



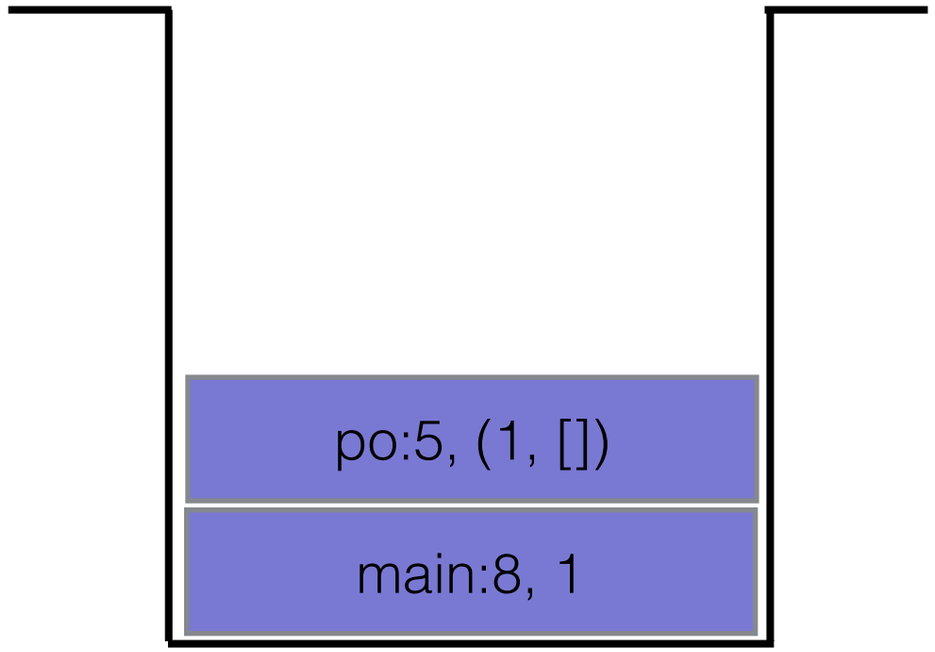

```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```



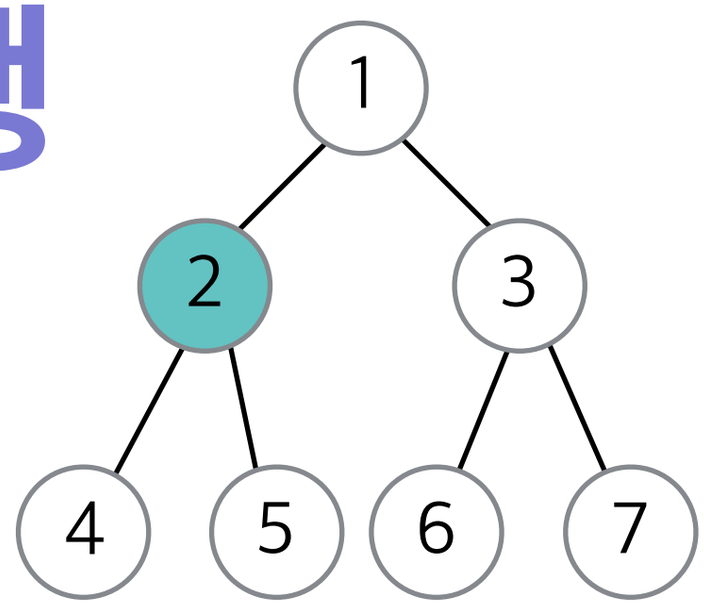
재귀함수의 실행




```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

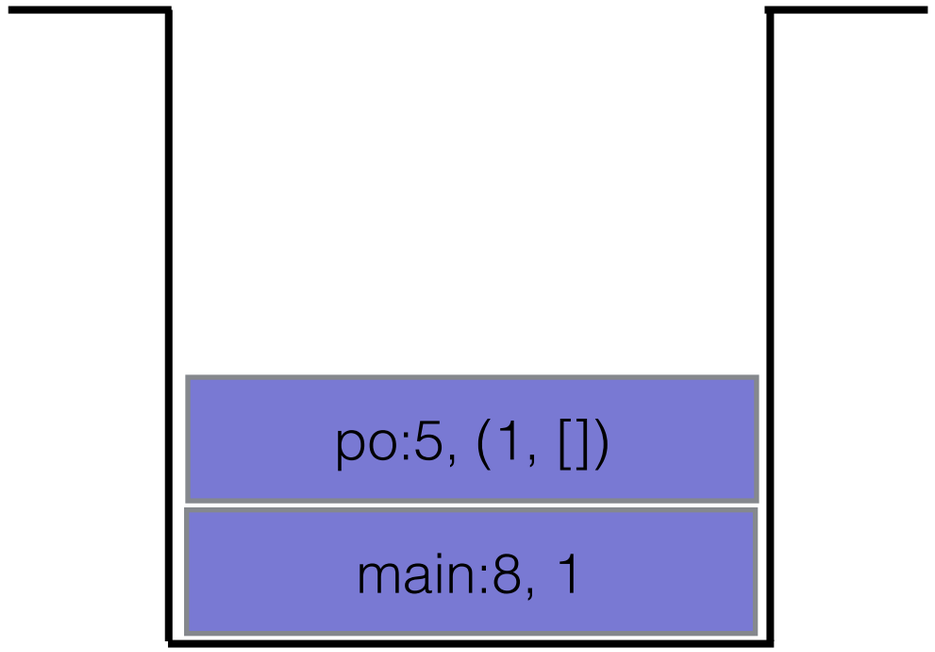


재귀함수의 실행

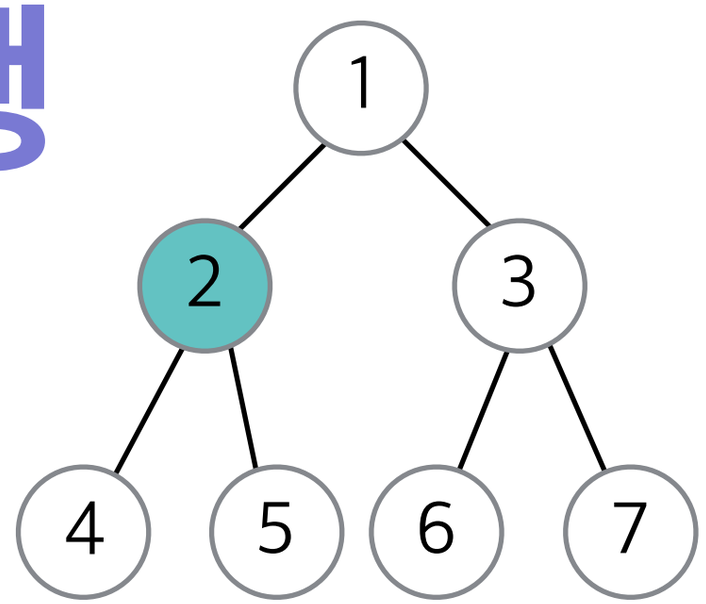



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[]

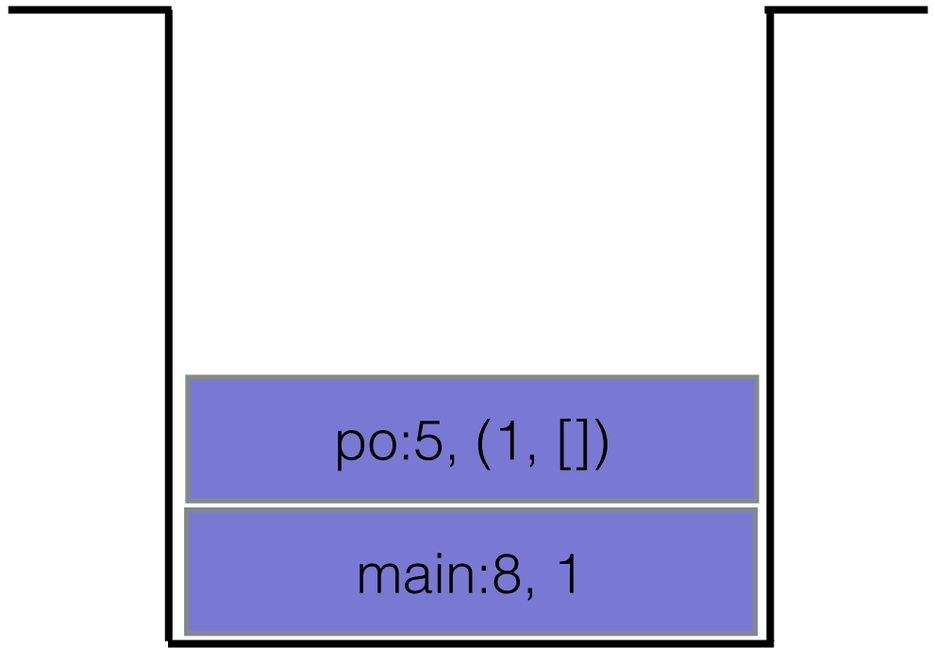


재귀함수의 실행

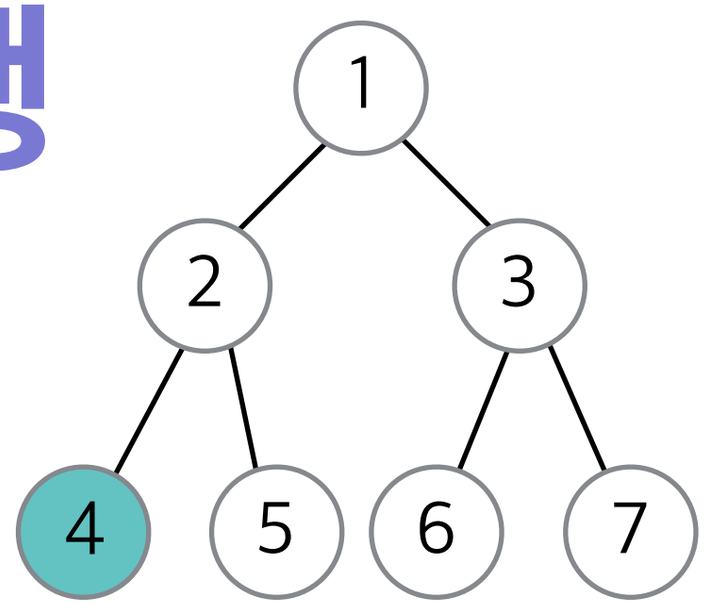



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

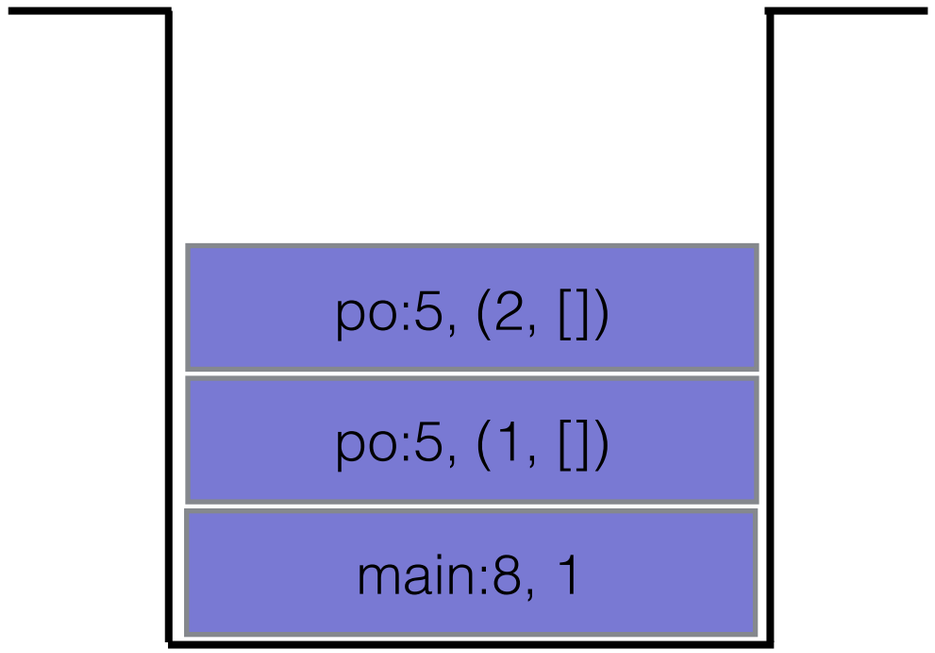
[]



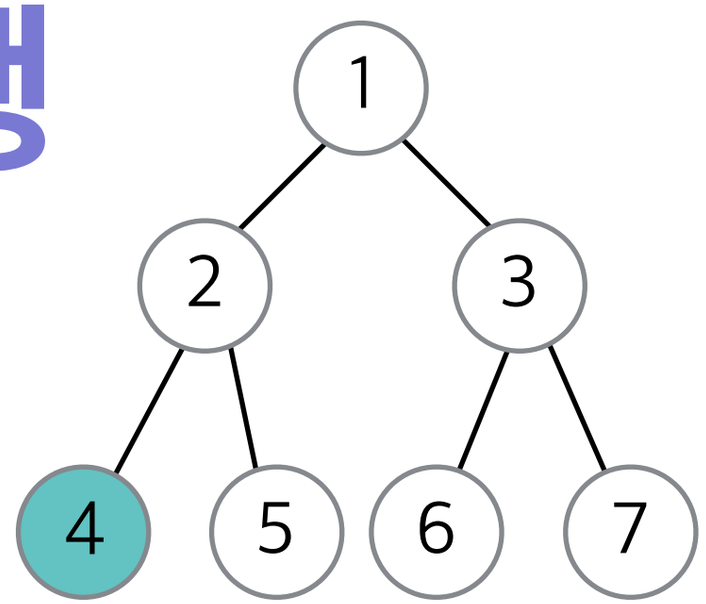
재귀함수의 실행



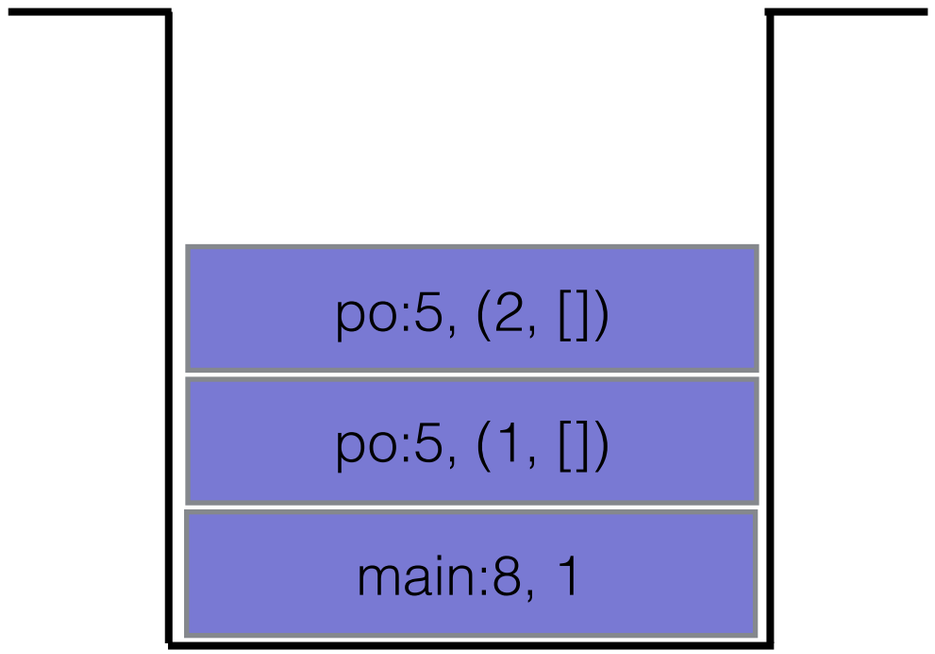

```
1  def postorder(tree) :
2      result = []

3      if tree == None :
4          return []

5      result = postorder(tree.left)
6      result = result + postorder(tree.right)
7      result.append(tree.index)

8      return result
```



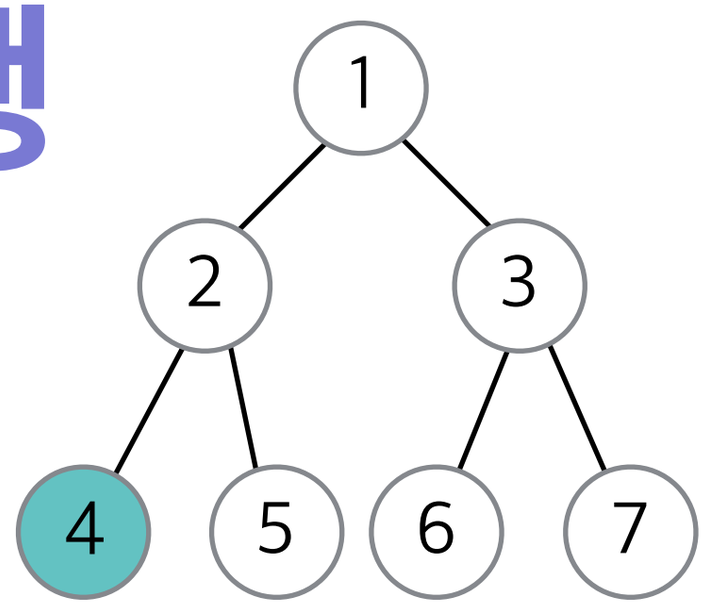
재귀함수의 실행




```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

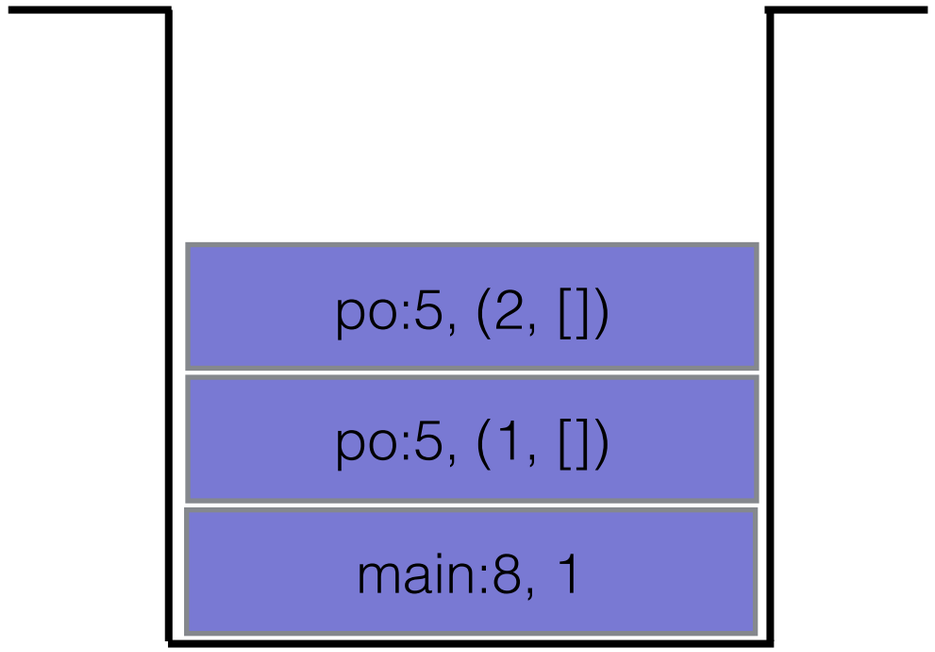


재귀함수의 실행

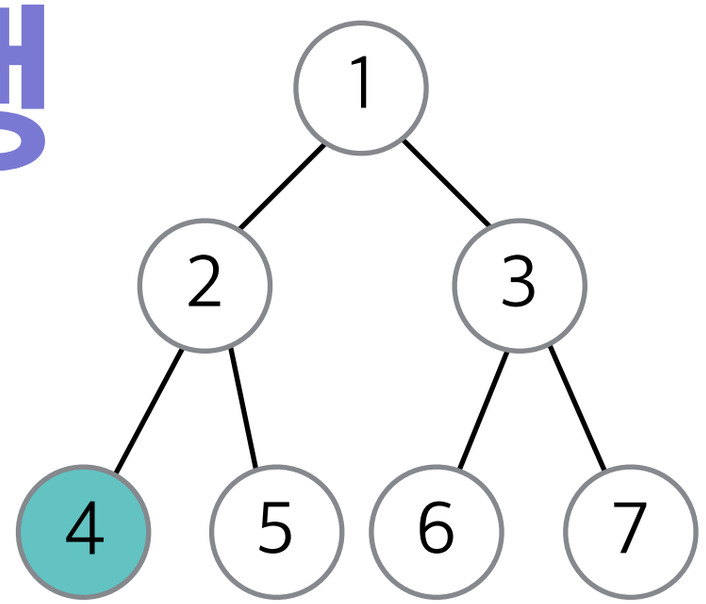



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[]

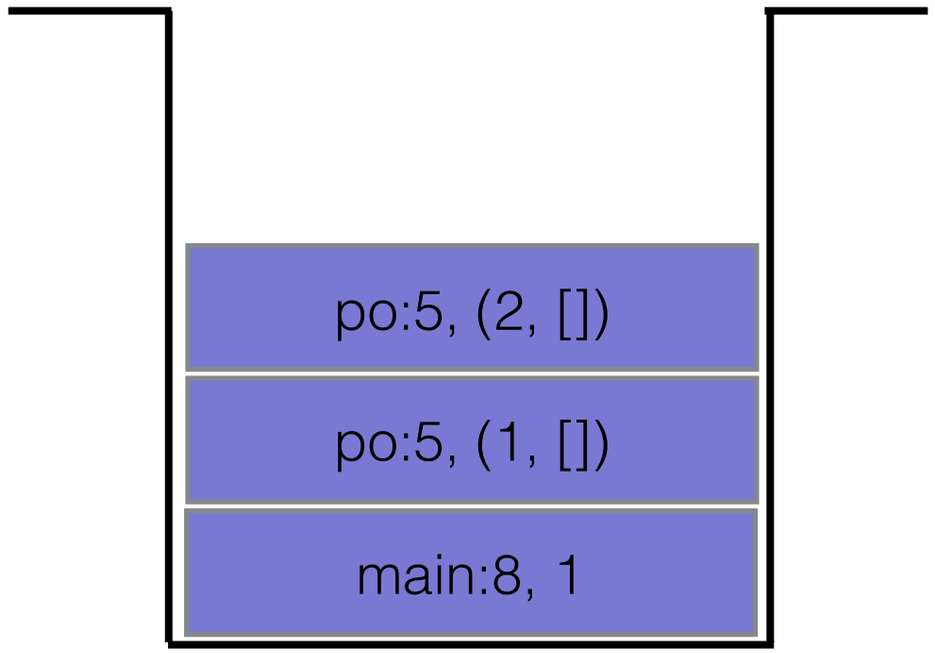


재귀함수의 실행

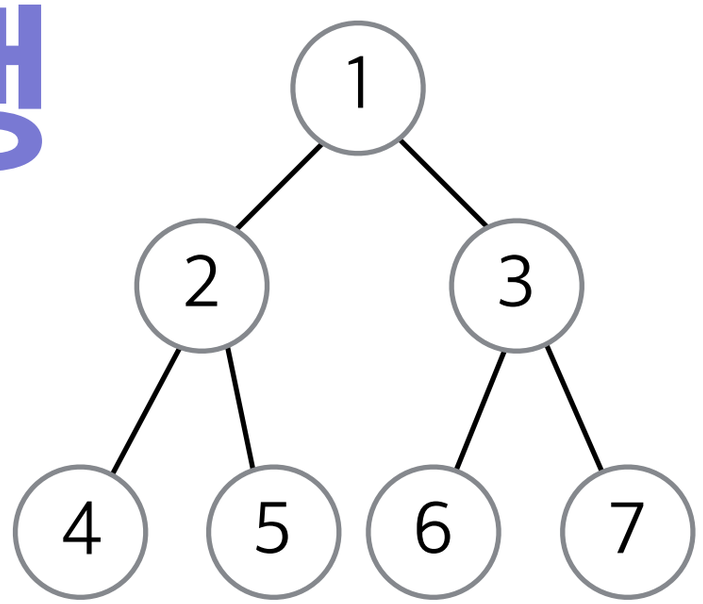



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

→5      result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

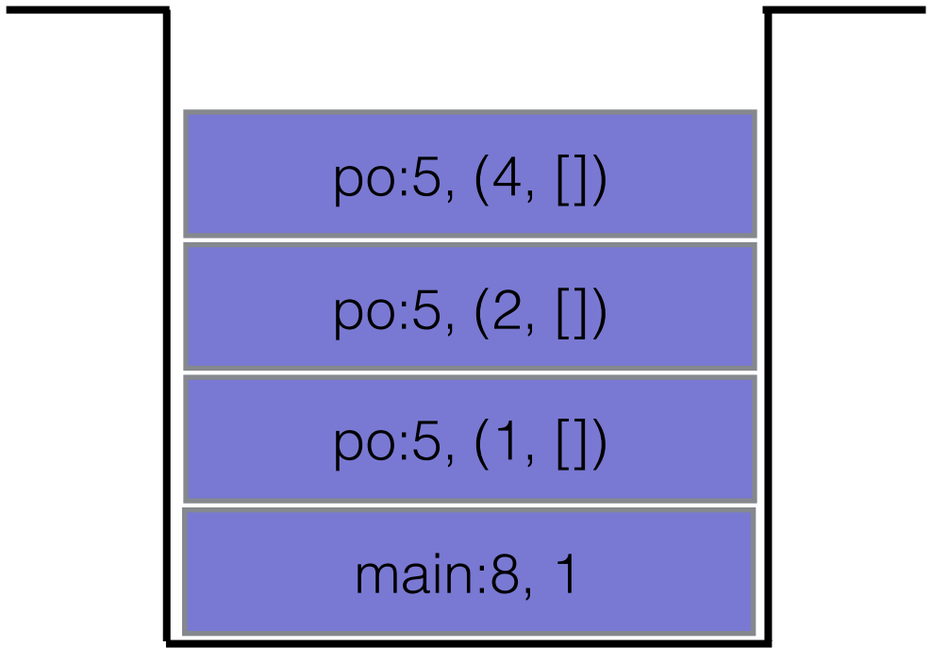
[]



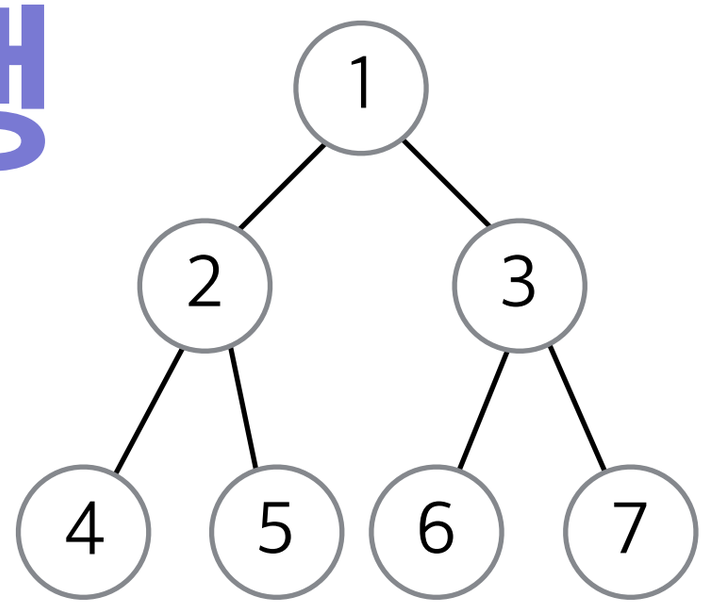
재귀함수의 실행



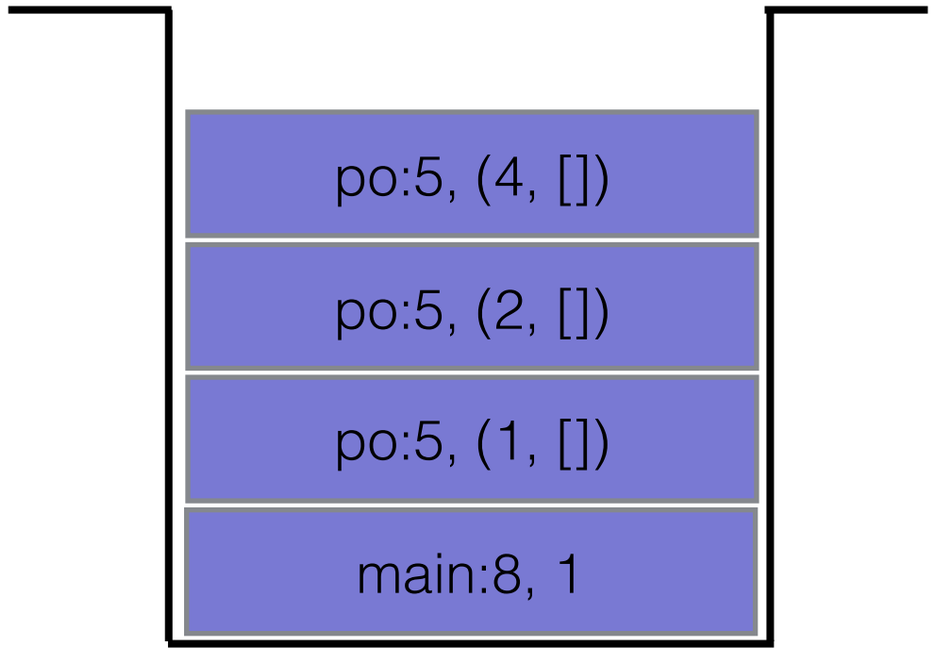

```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```



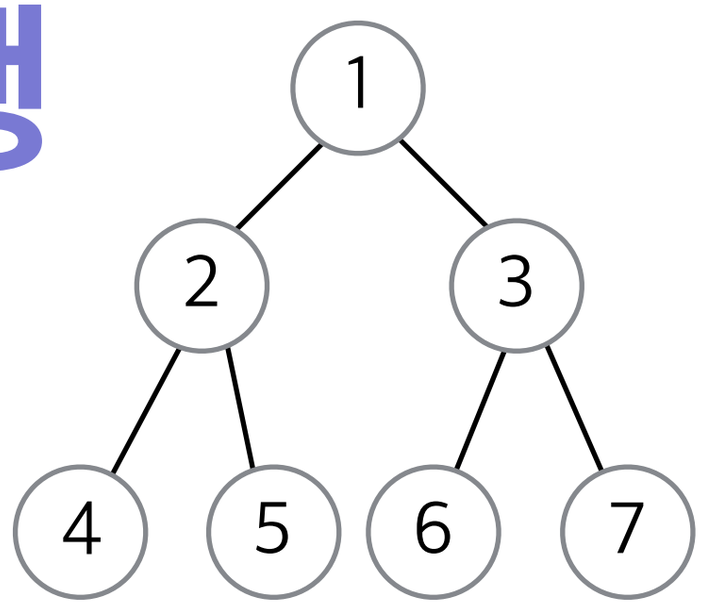
재귀함수의 실행




```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

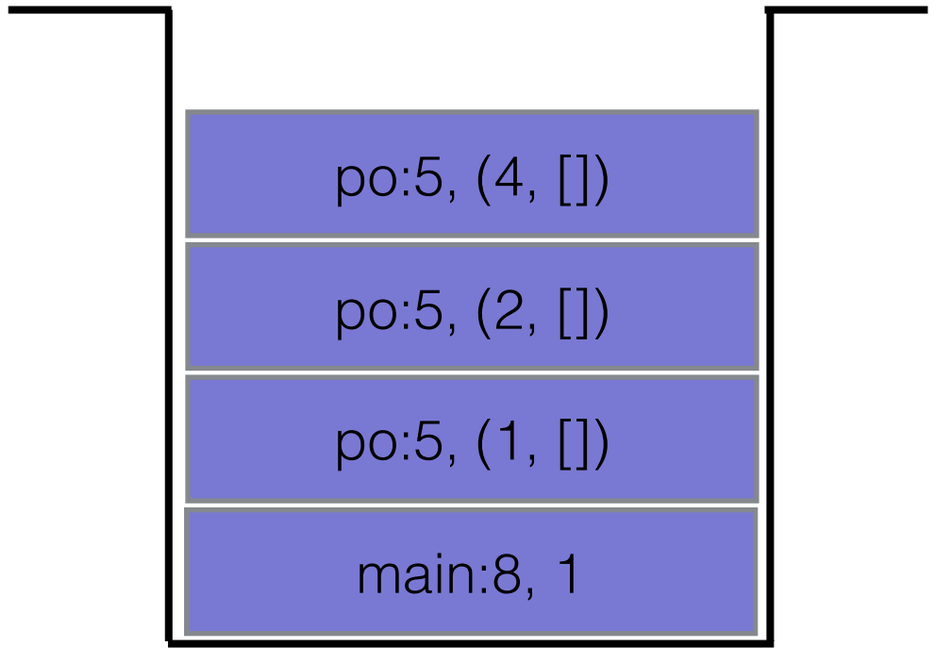


재귀함수의 실행

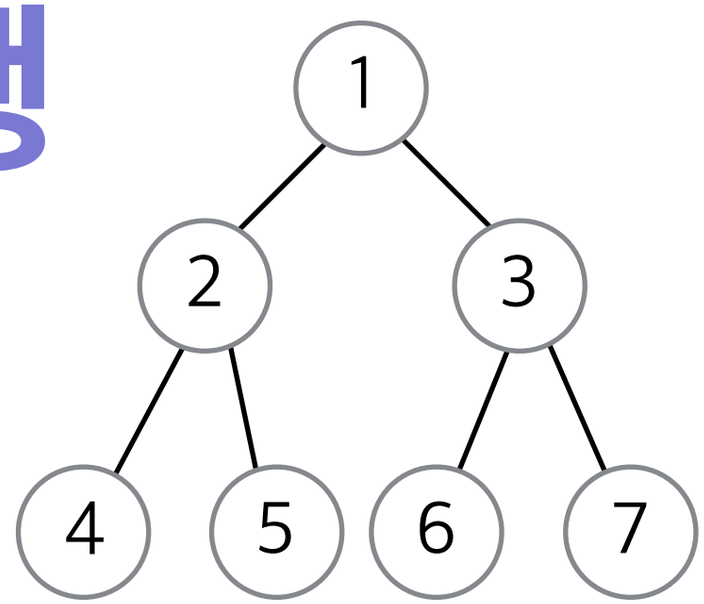



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

[]

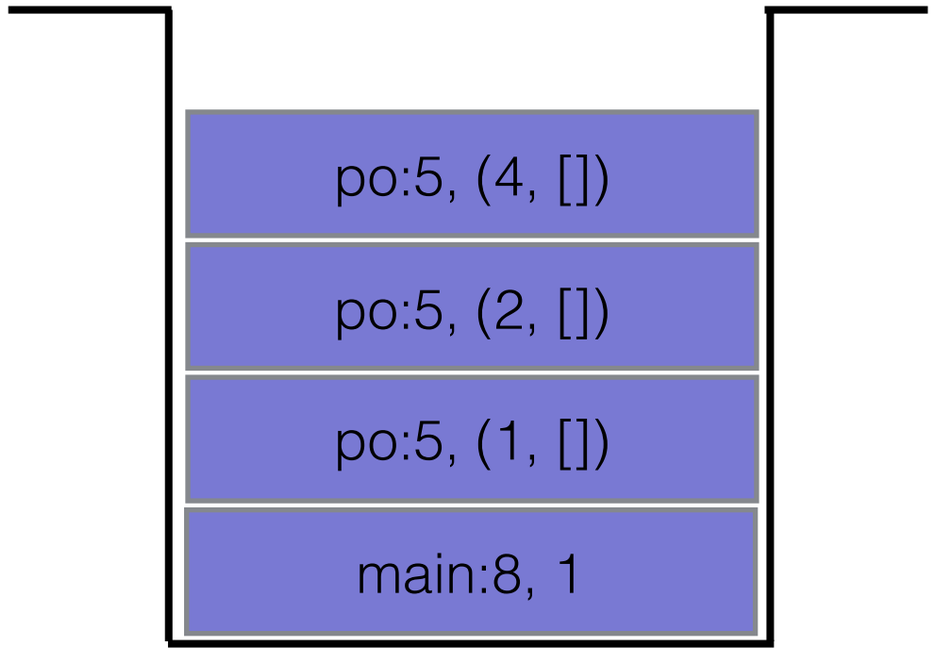


재귀함수의 실행



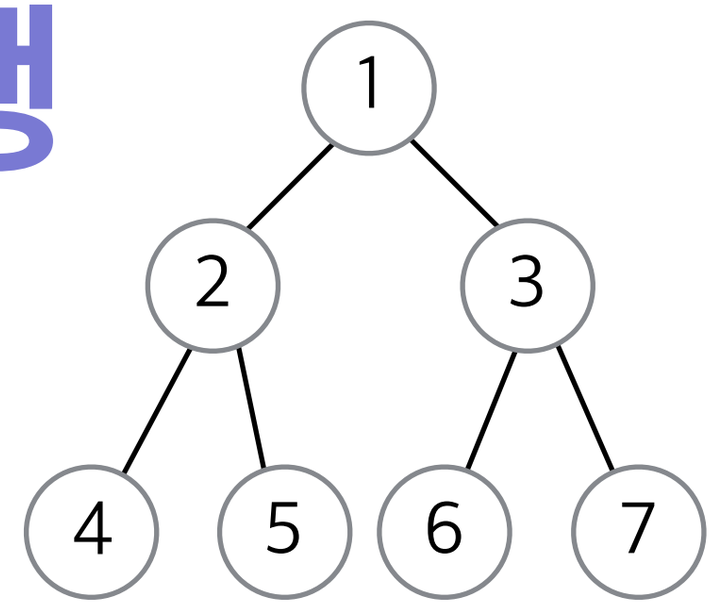

```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

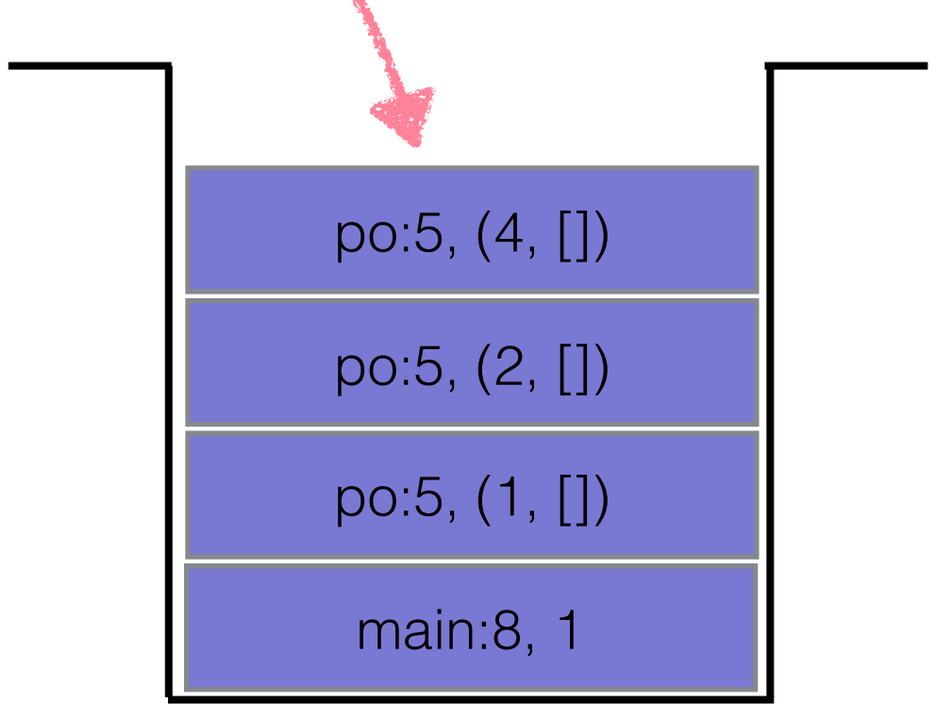
[]



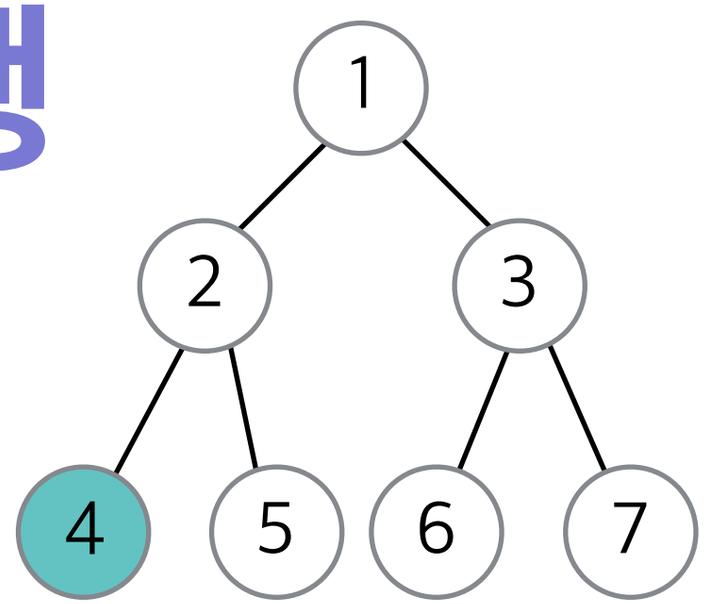
재귀함수의 실행

```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```



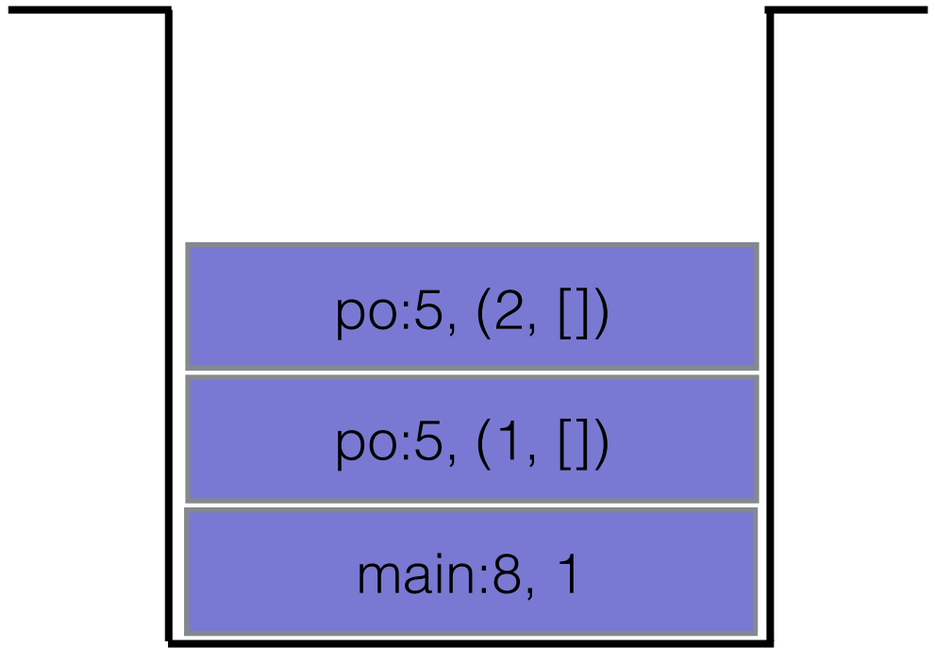


재귀함수의 실행

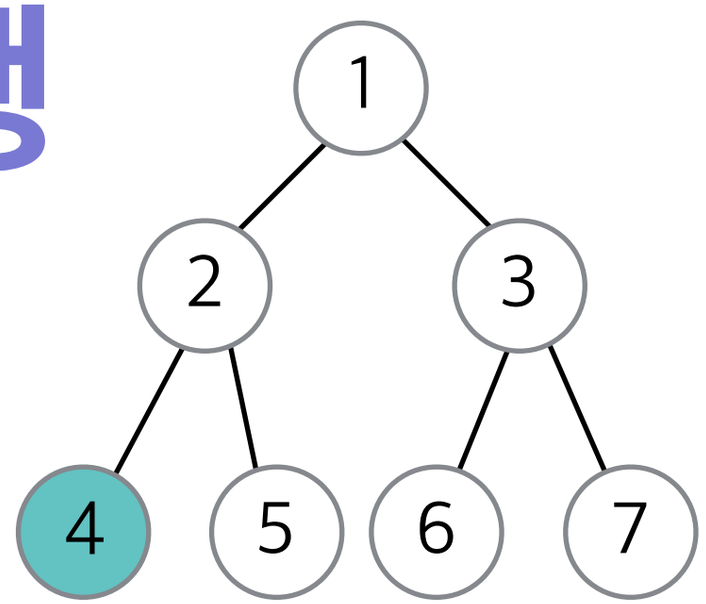



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[]

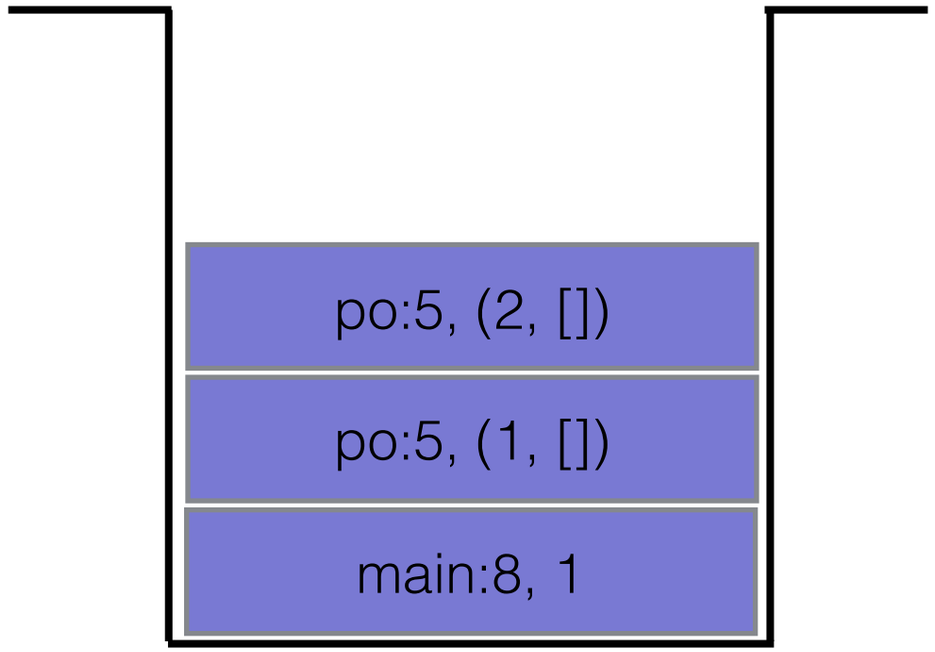


재귀함수의 실행

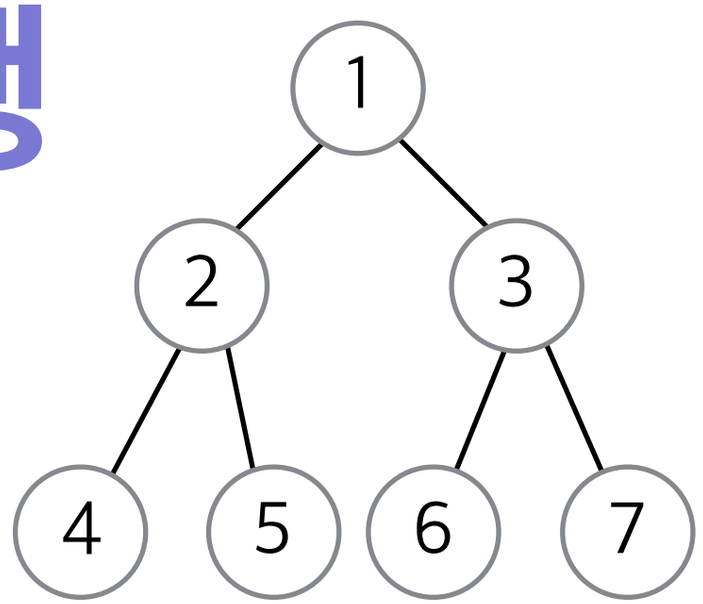



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

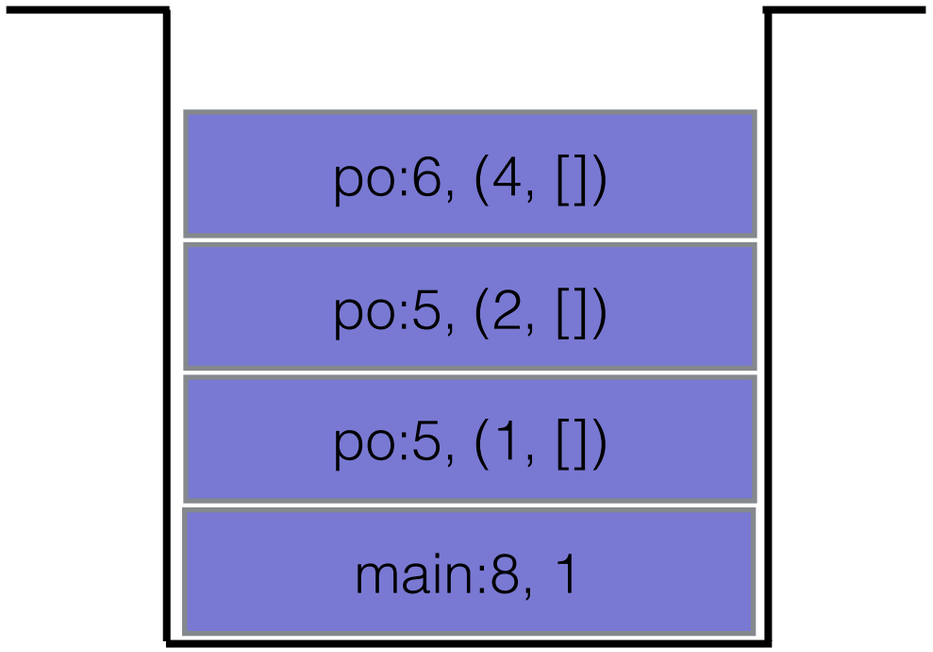
[]



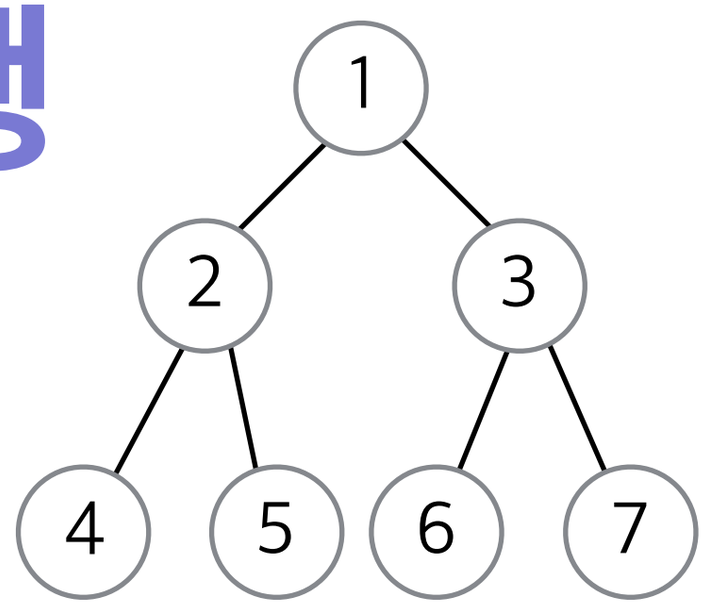
재귀함수의 실행



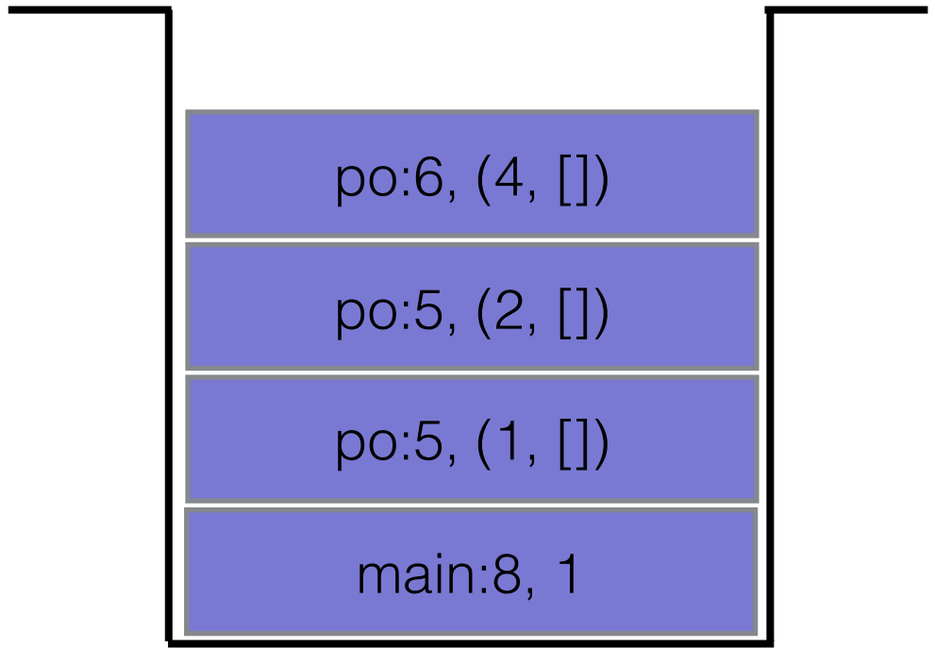

```
1  def postorder(tree) :
2      result = []

3      if tree == None :
4          return []

5      result = postorder(tree.left)
6      result = result + postorder(tree.right)
7      result.append(tree.index)

8      return result
```



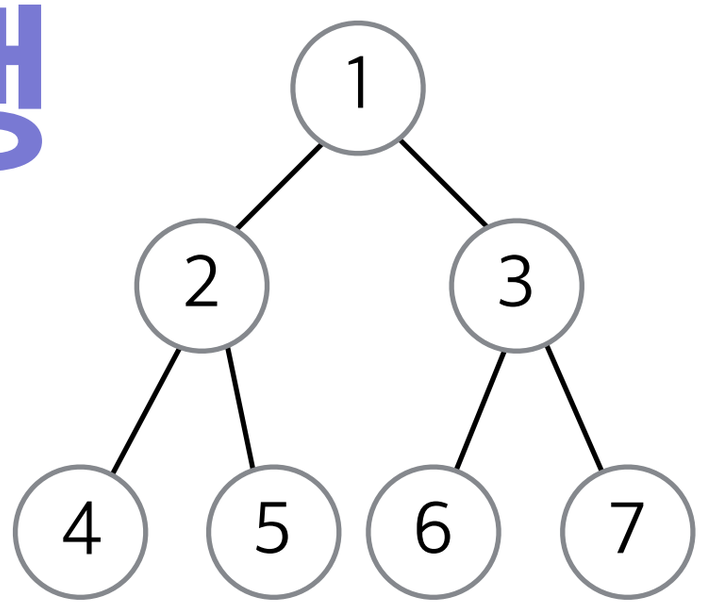
재귀함수의 실행




```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

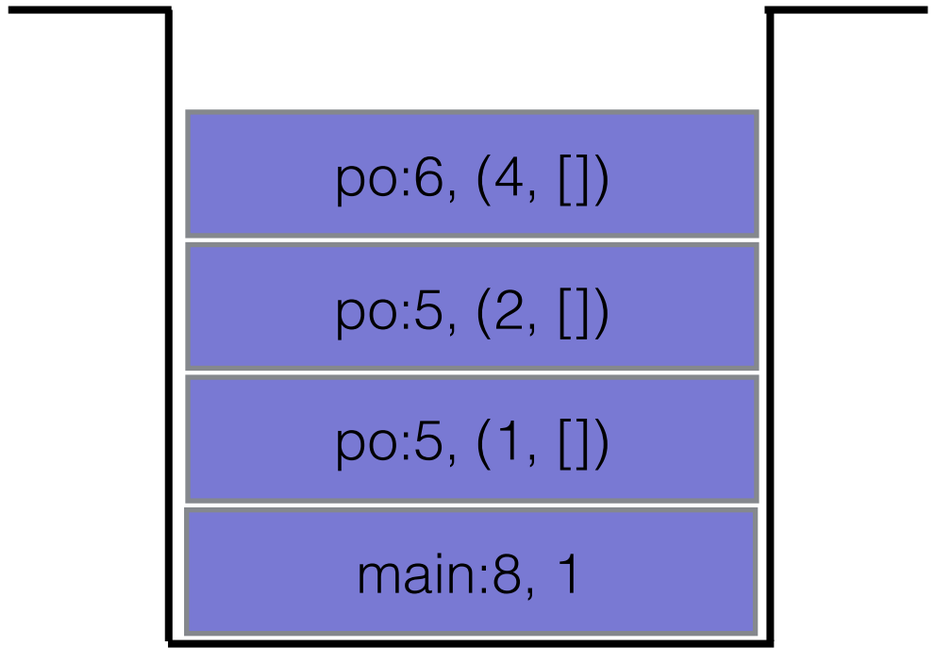


재귀함수의 실행

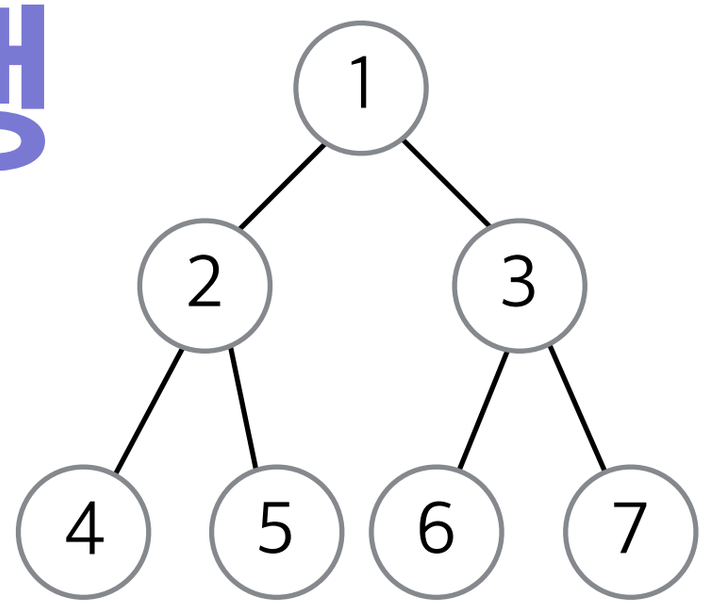



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

[]

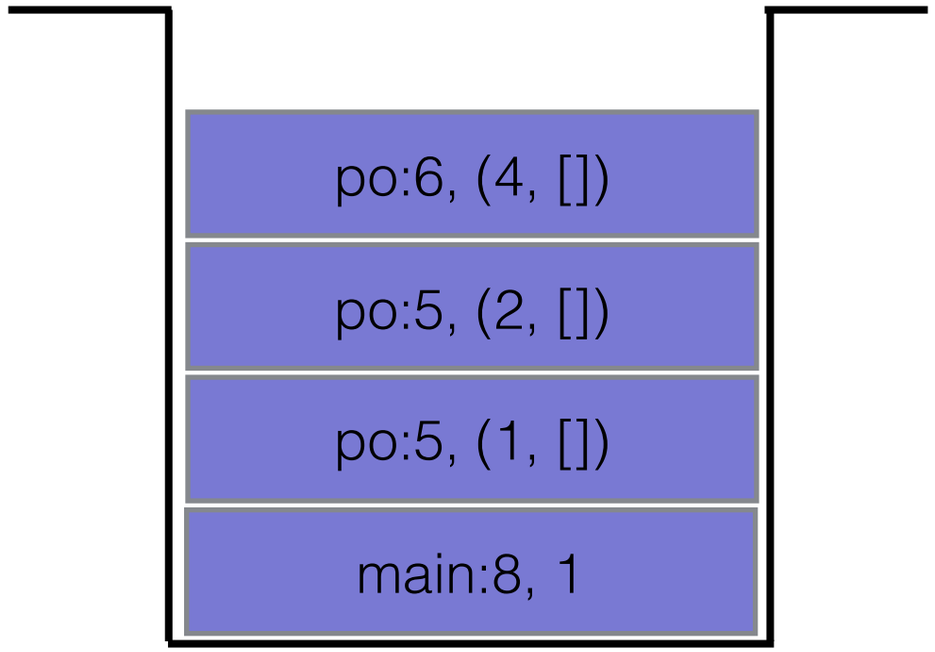


재귀함수의 실행



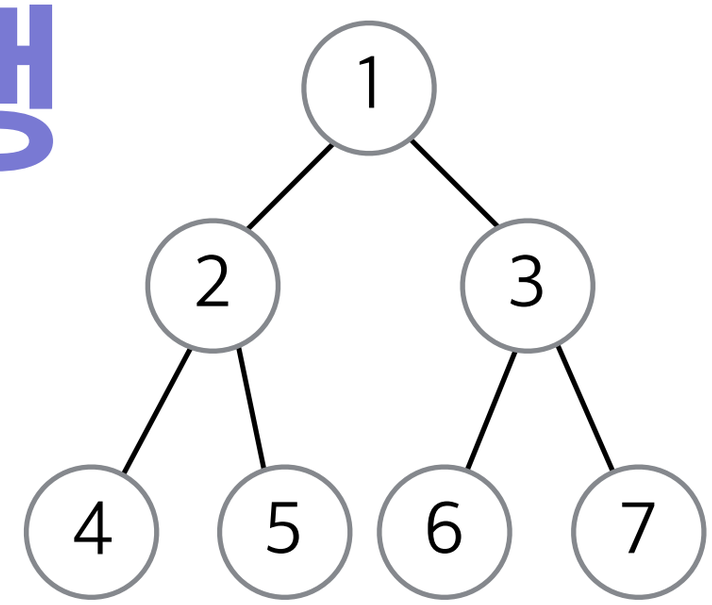

```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

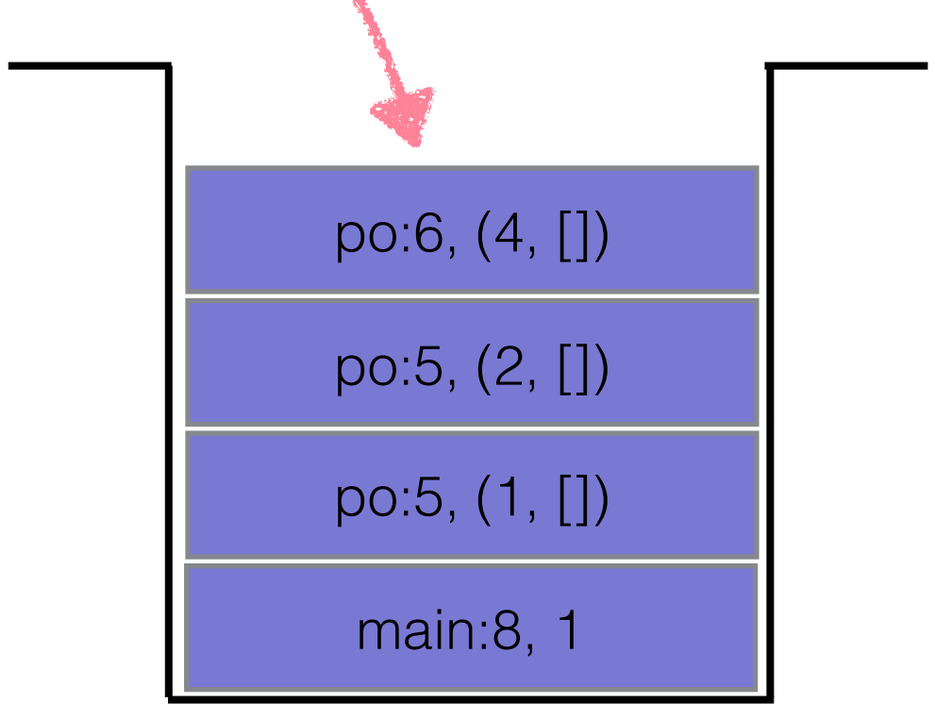
[]



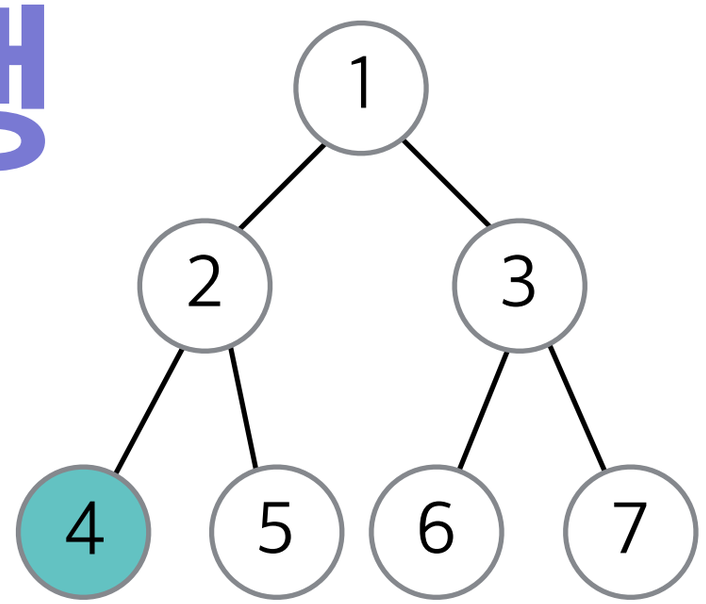
재귀함수의 실행

```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```



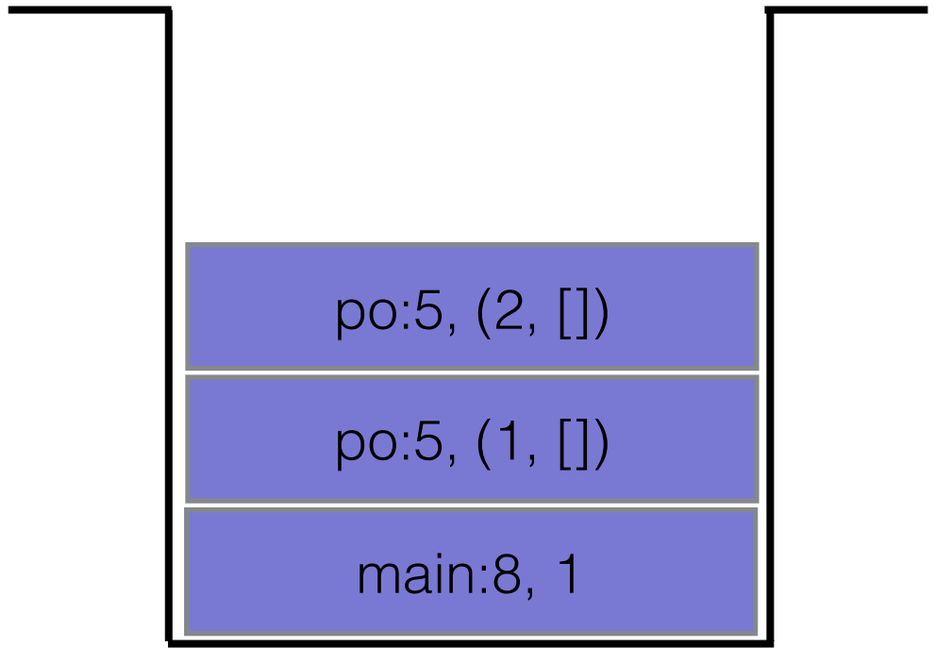


재귀함수의 실행

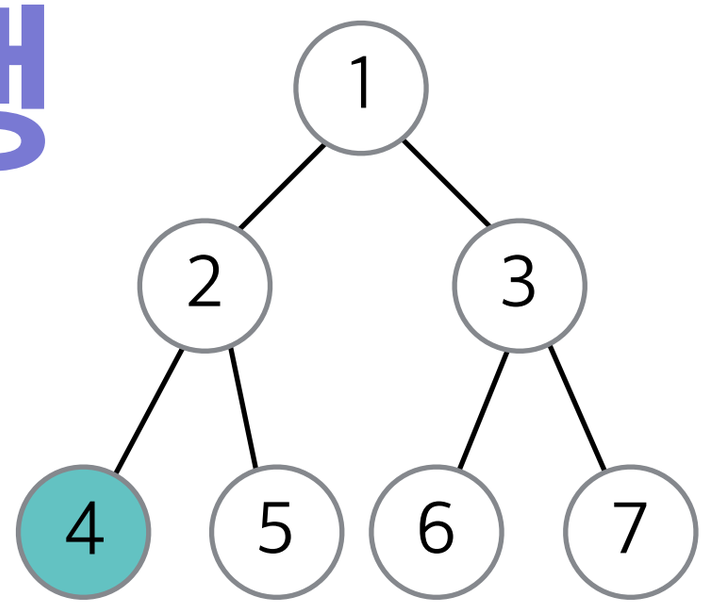



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[]

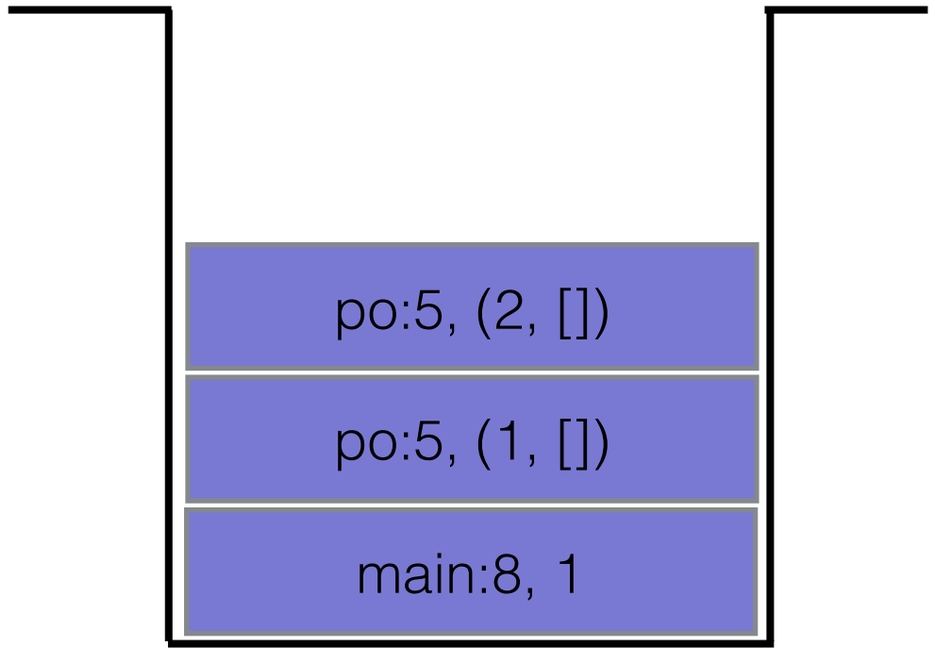


재귀함수의 실행

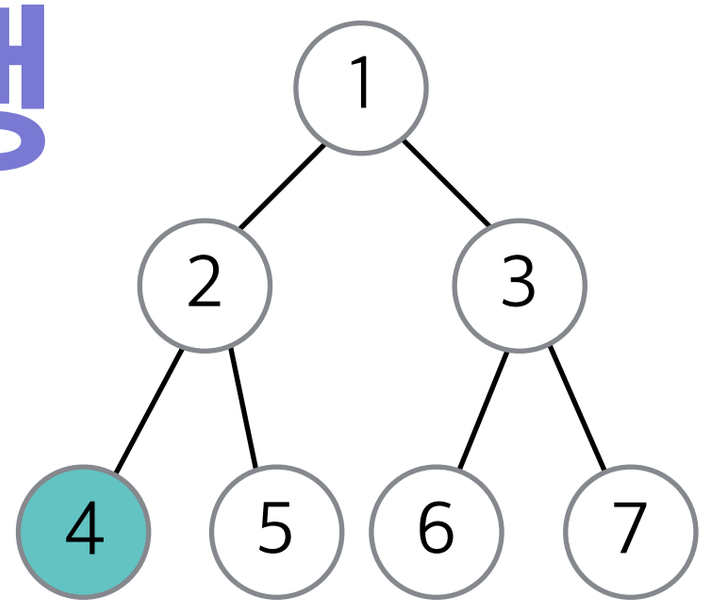



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[]

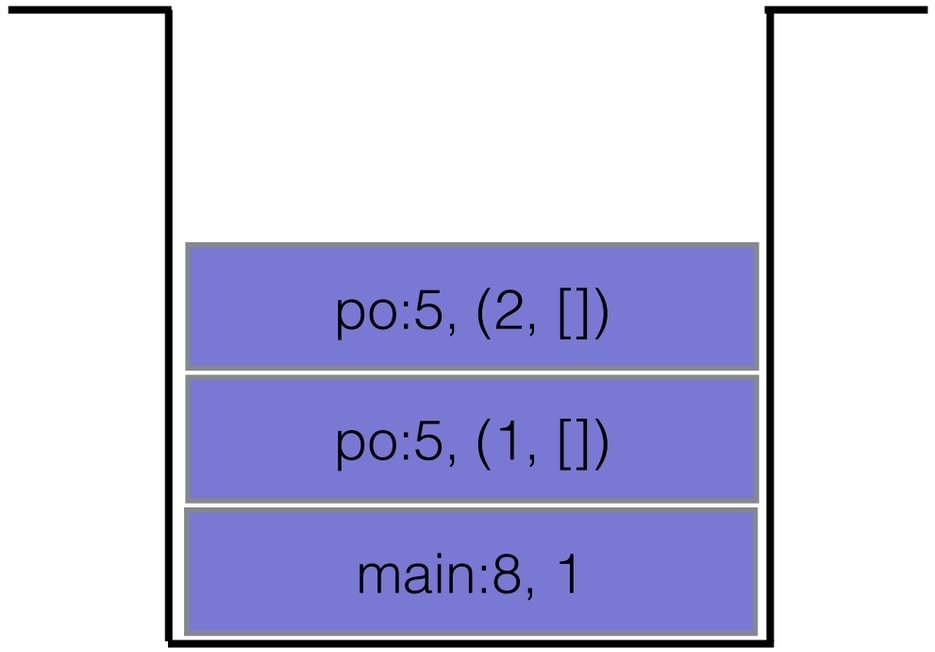


재귀함수의 실행

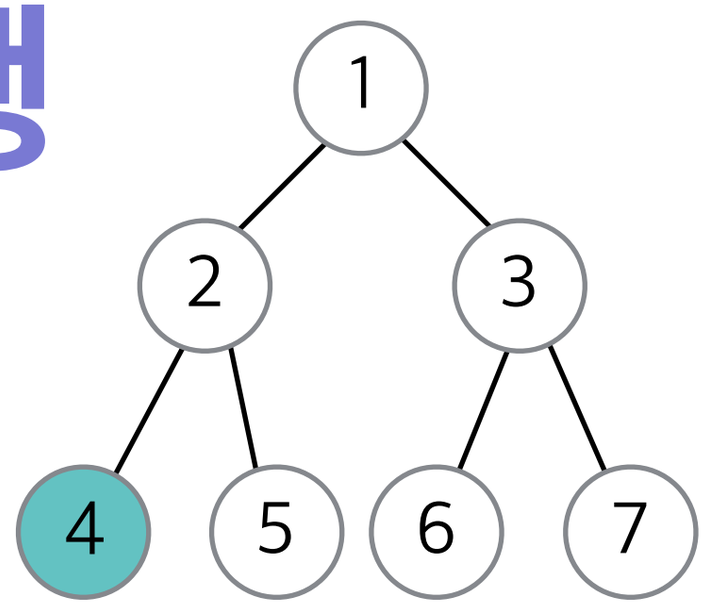



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
→ 8     return result
```

[4]

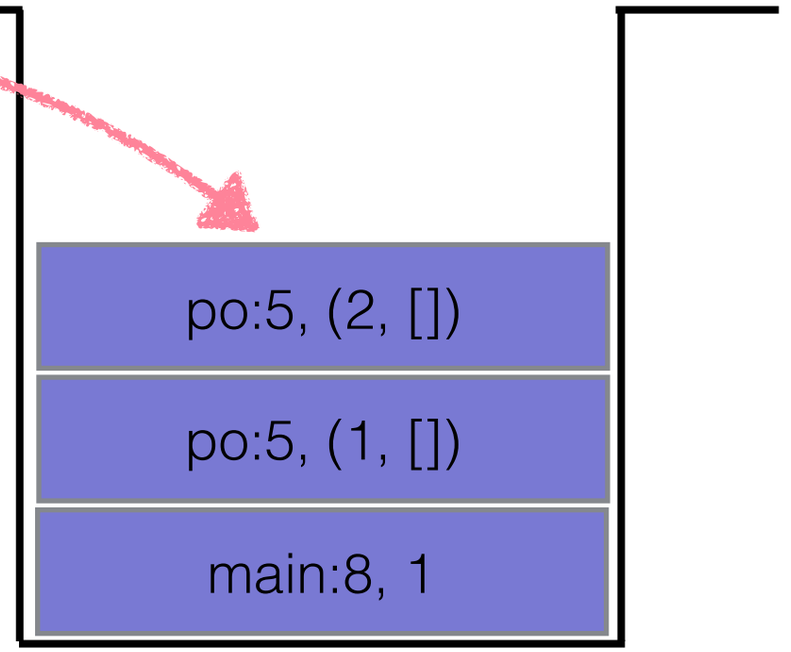


재귀함수의 실행

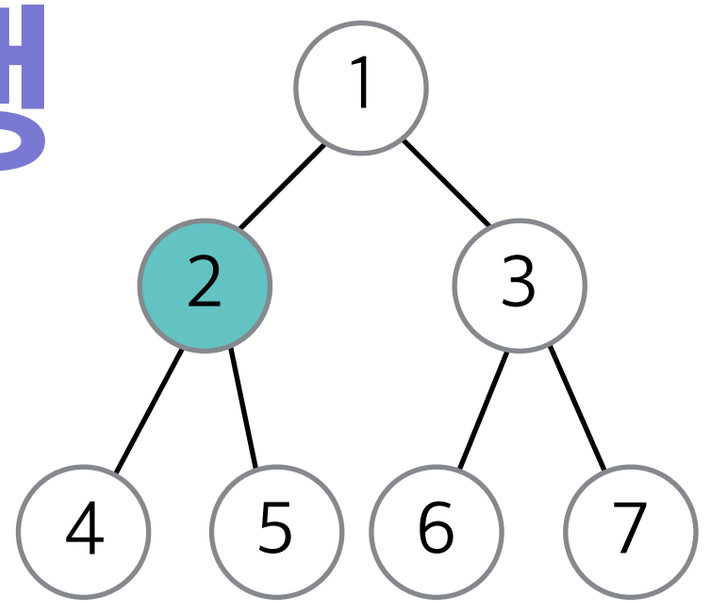



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

→ 8     return result
```

[4]

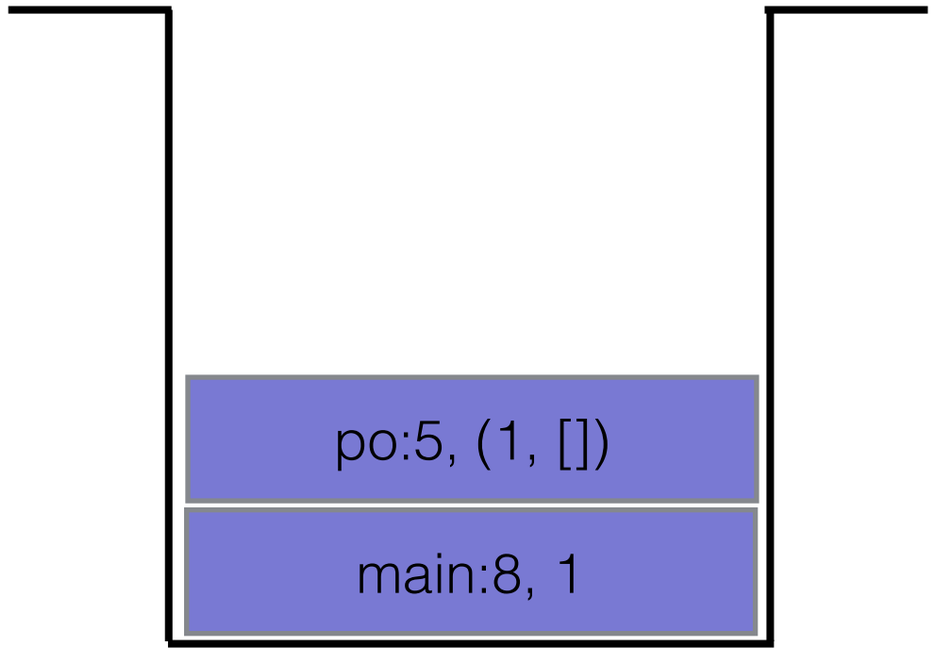


재귀함수의 실행

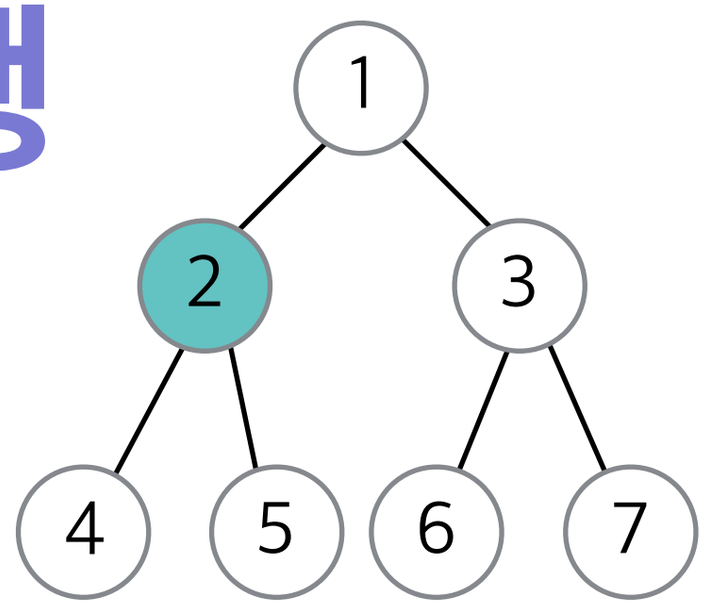



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

[]

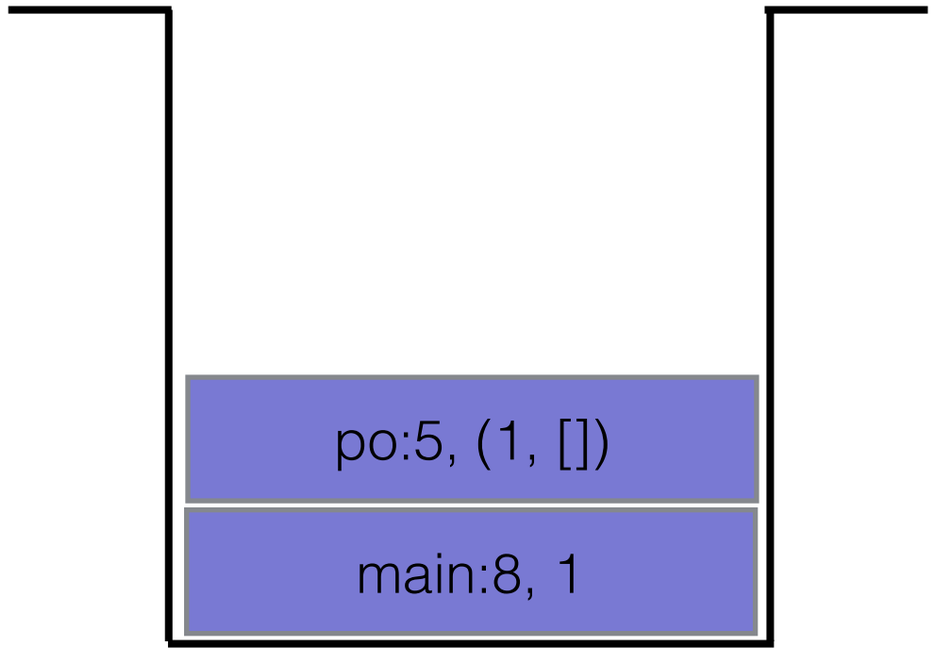


재귀함수의 실행

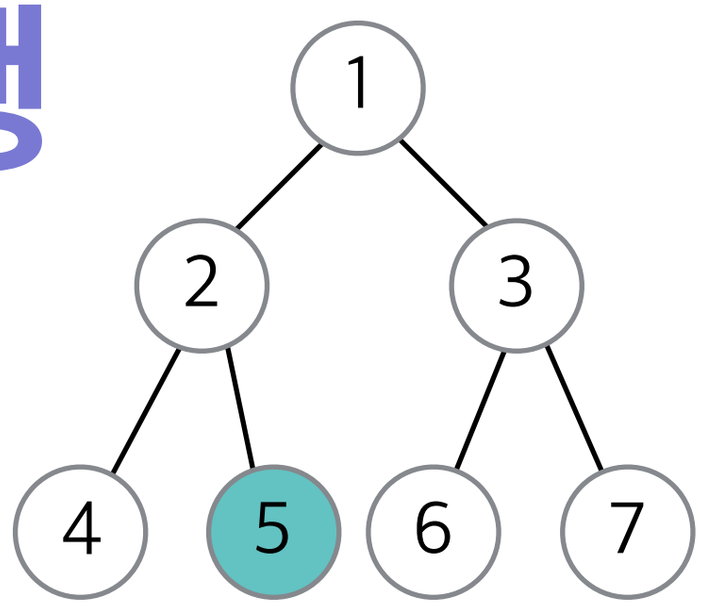



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

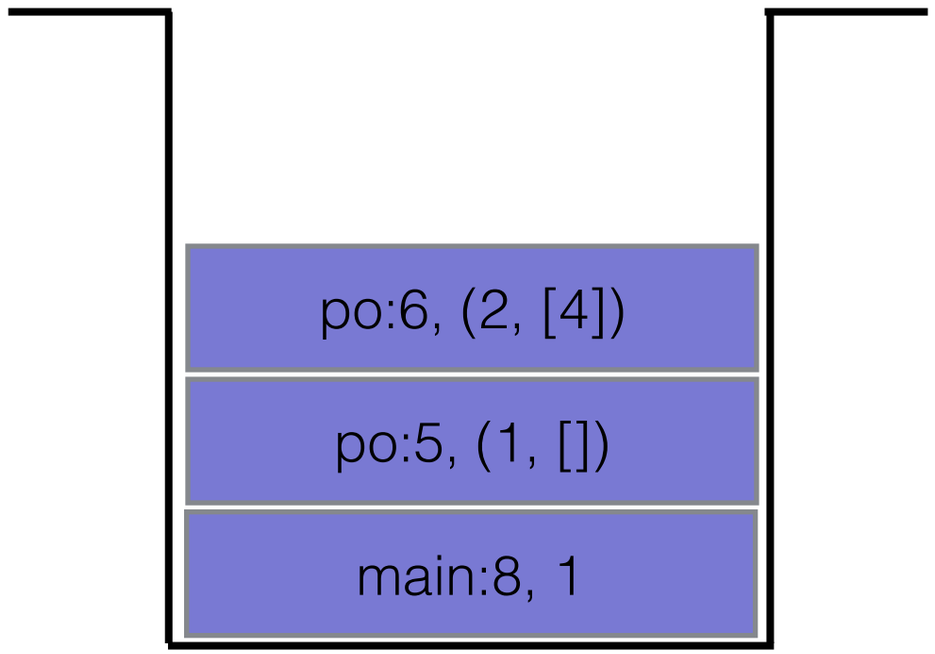
[4]



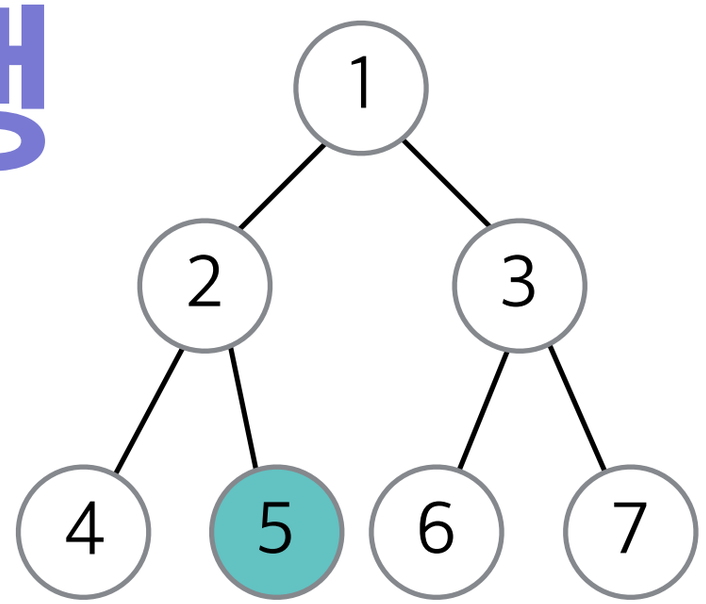
재귀함수의 실행



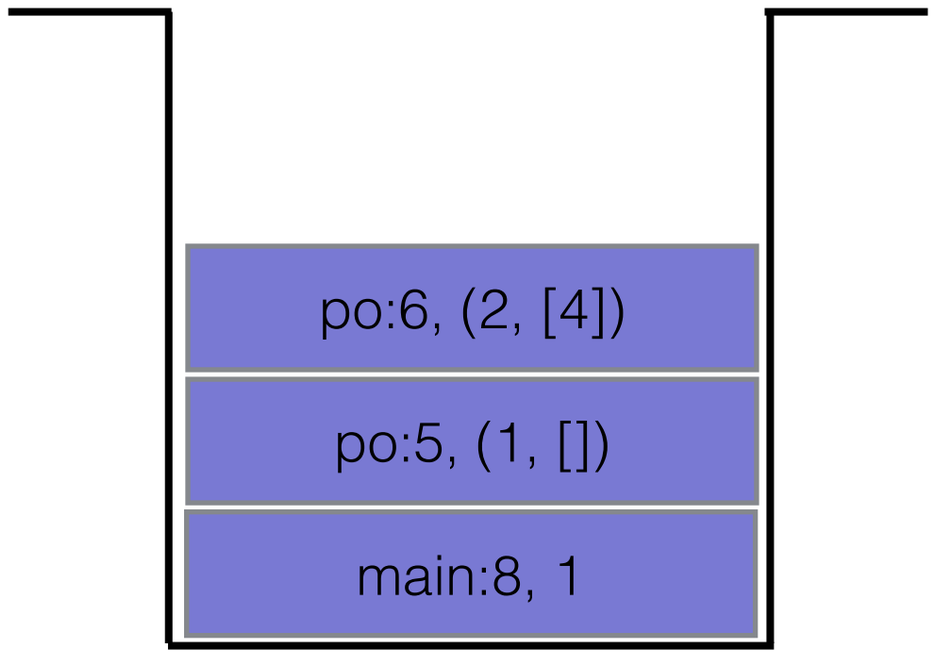

```
1  def postorder(tree) :
2      result = []

3      if tree == None :
4          return []

5      result = postorder(tree.left)
6      result = result + postorder(tree.right)
7      result.append(tree.index)

8      return result
```



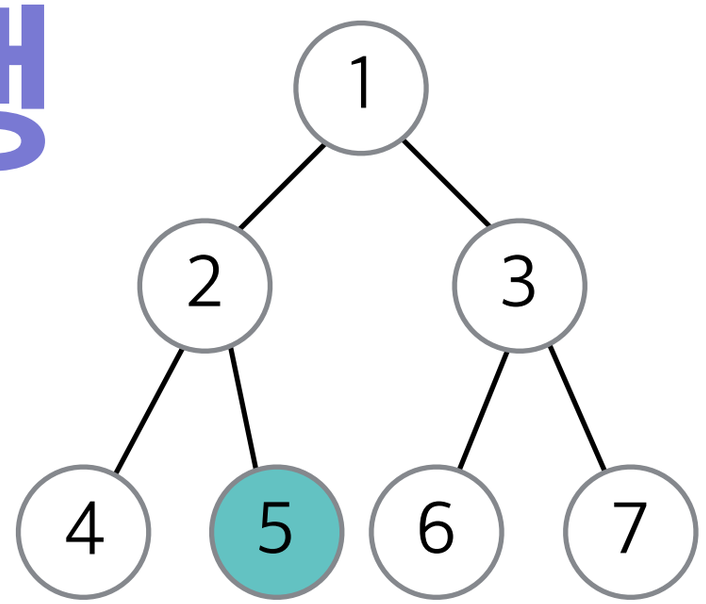
재귀함수의 실행




```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

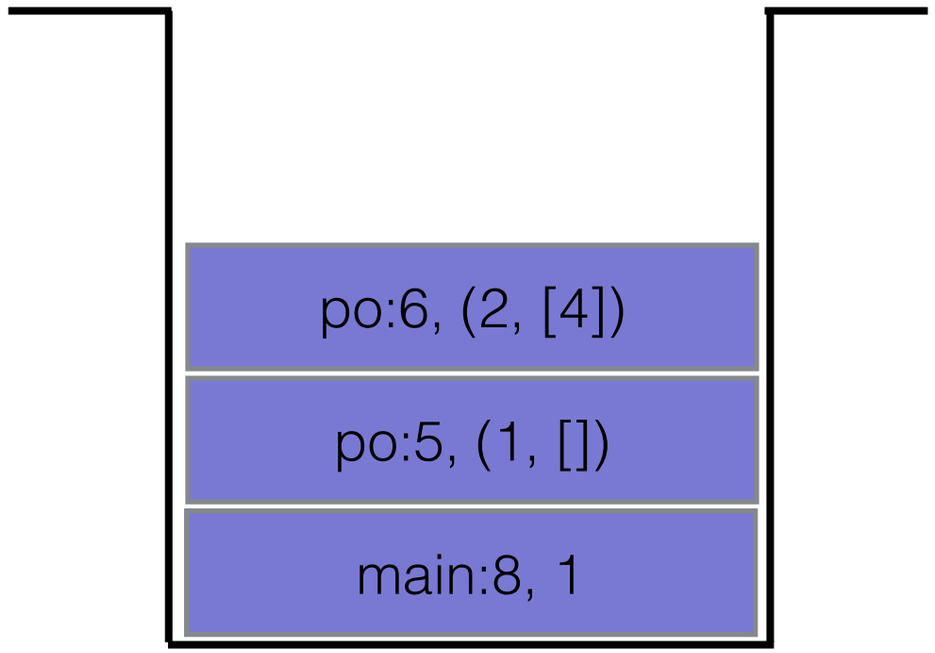


재귀함수의 실행

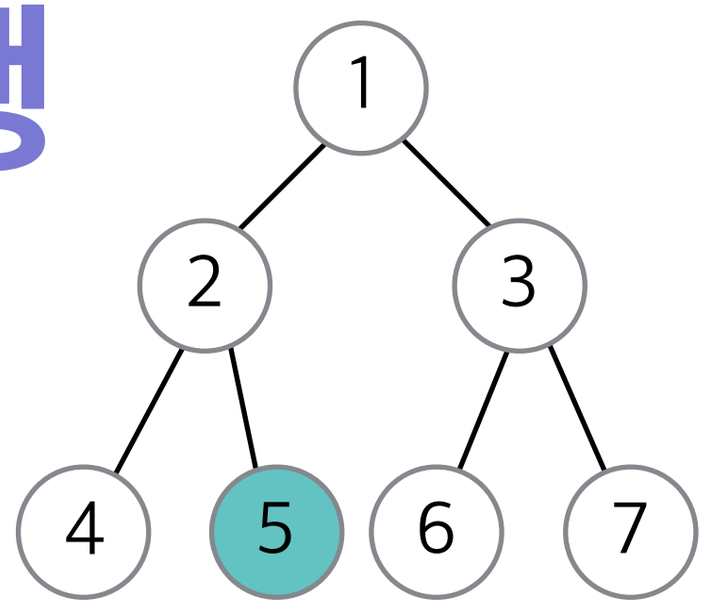



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[]

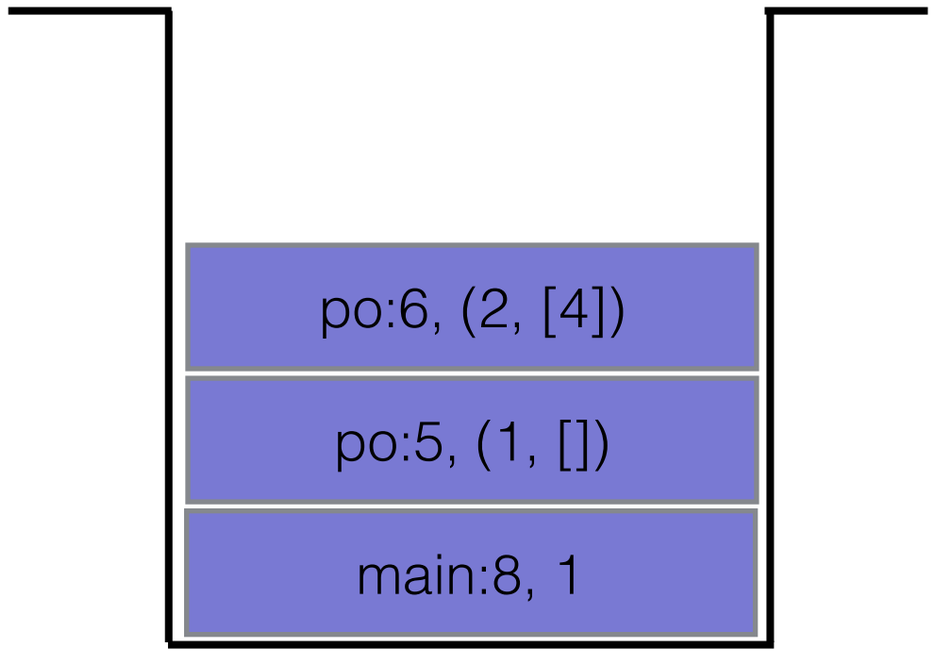


재귀함수의 실행

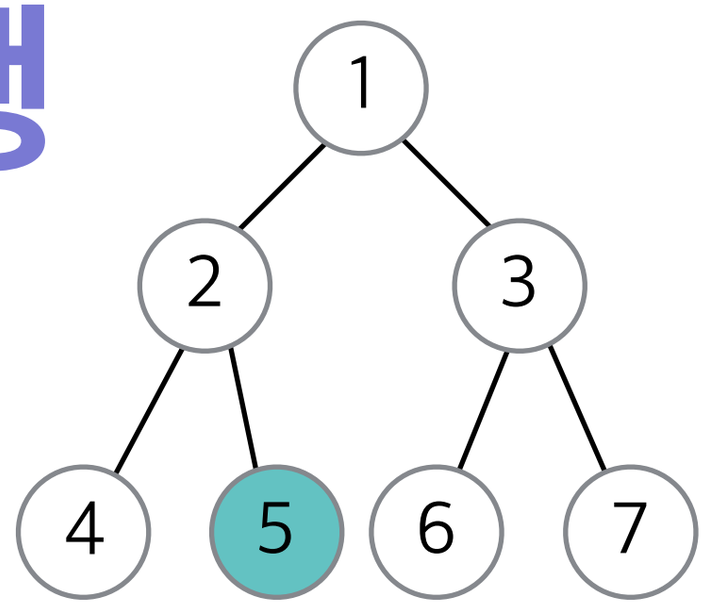



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

8       return result
```

[]

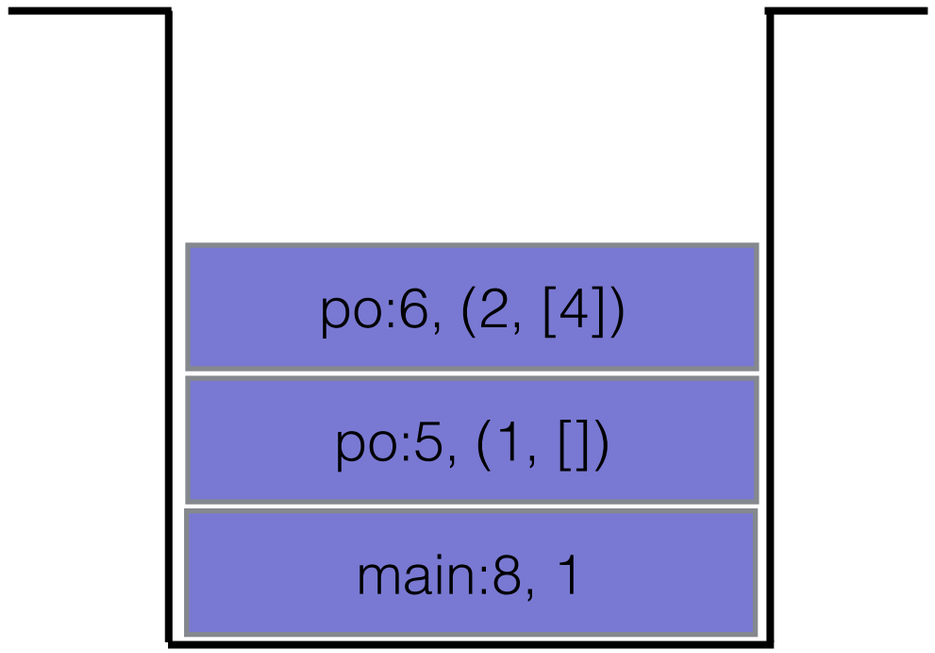


재귀함수의 실행

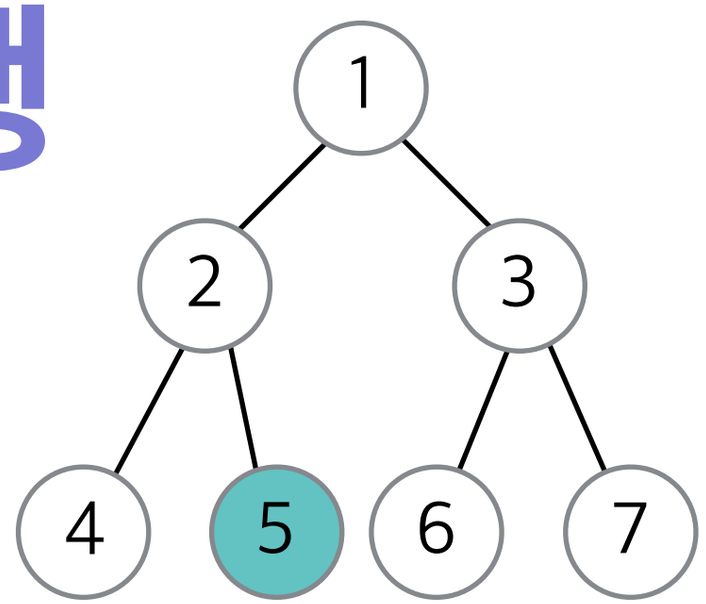



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[]

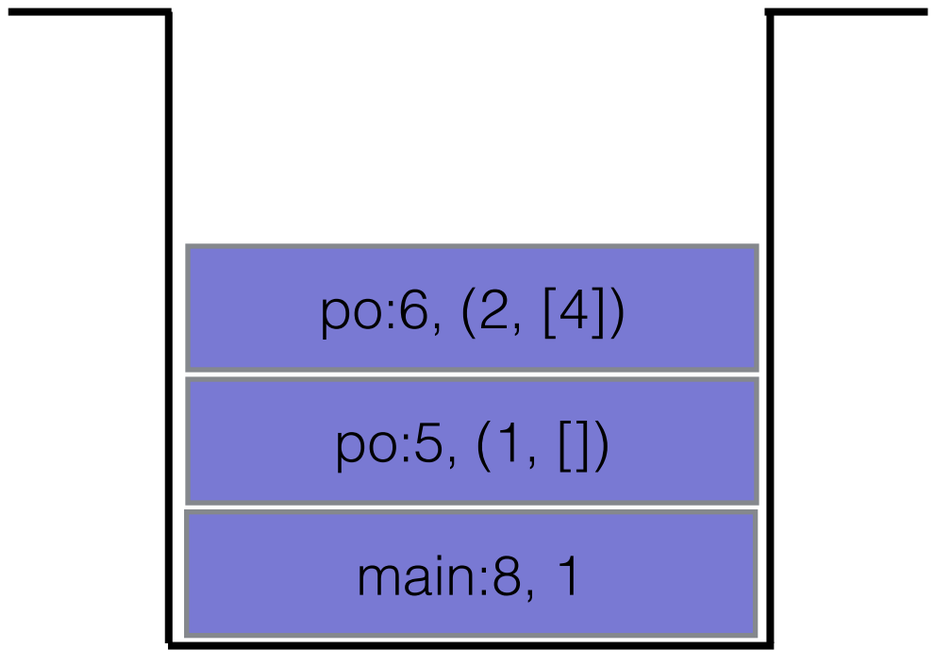


재귀함수의 실행

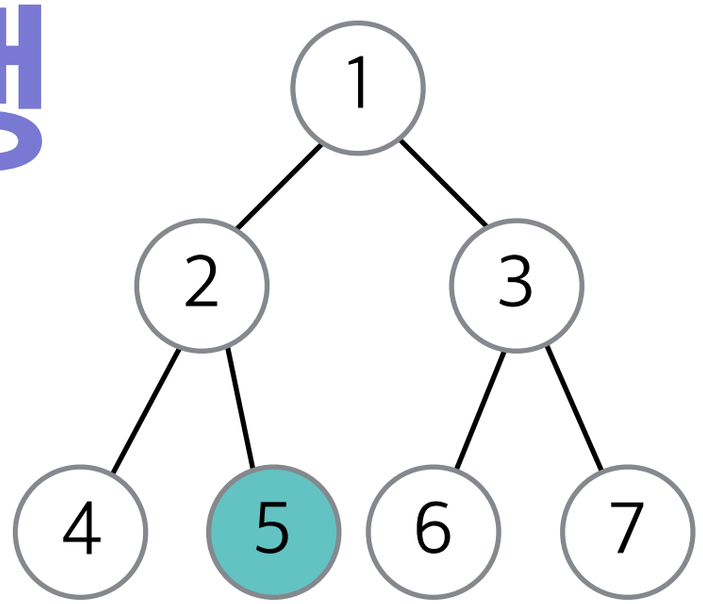



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
→7      result.append(tree.index)
8       return result
```

[]

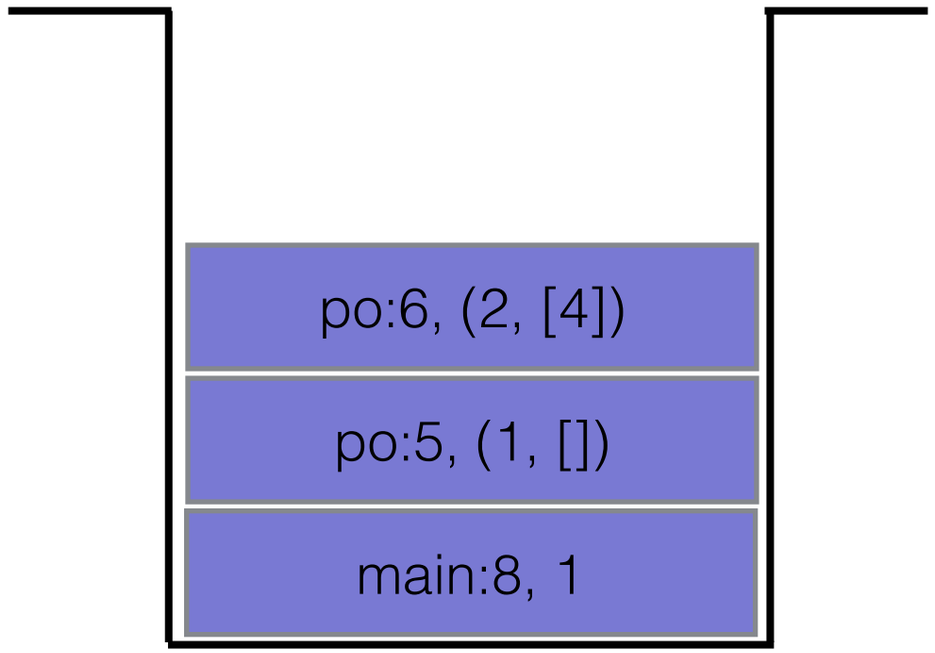


재귀함수의 실행

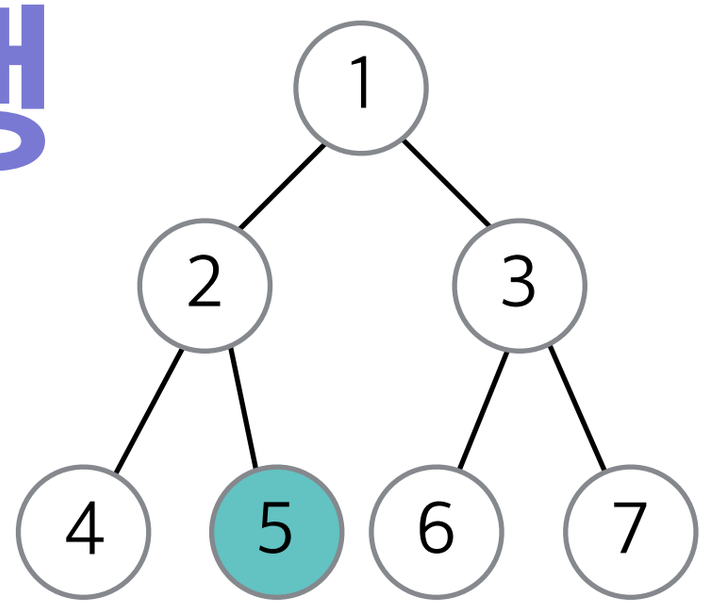



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

→ 8     return result
```

[5]

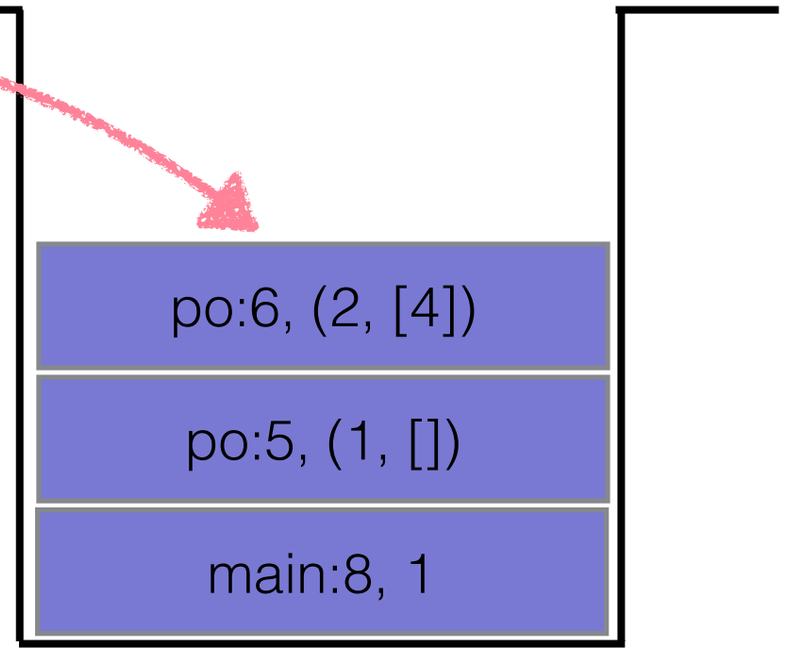


재귀함수의 실행

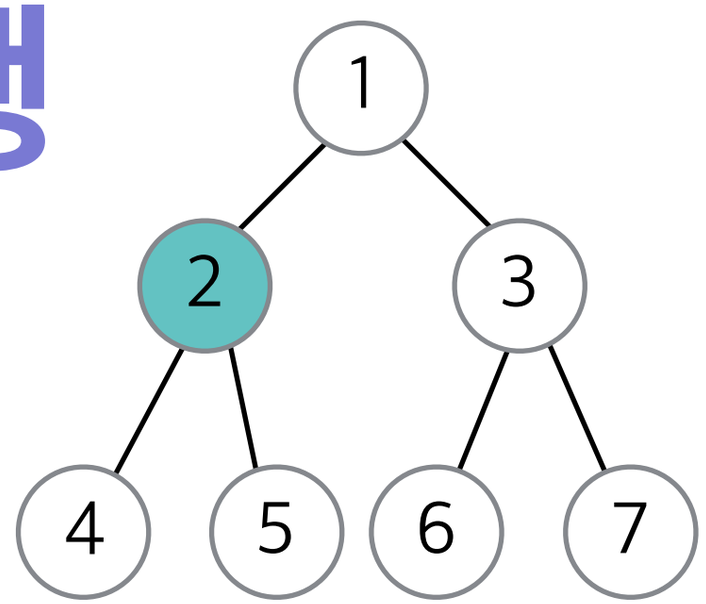



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

→ 8     return result
```

[5]

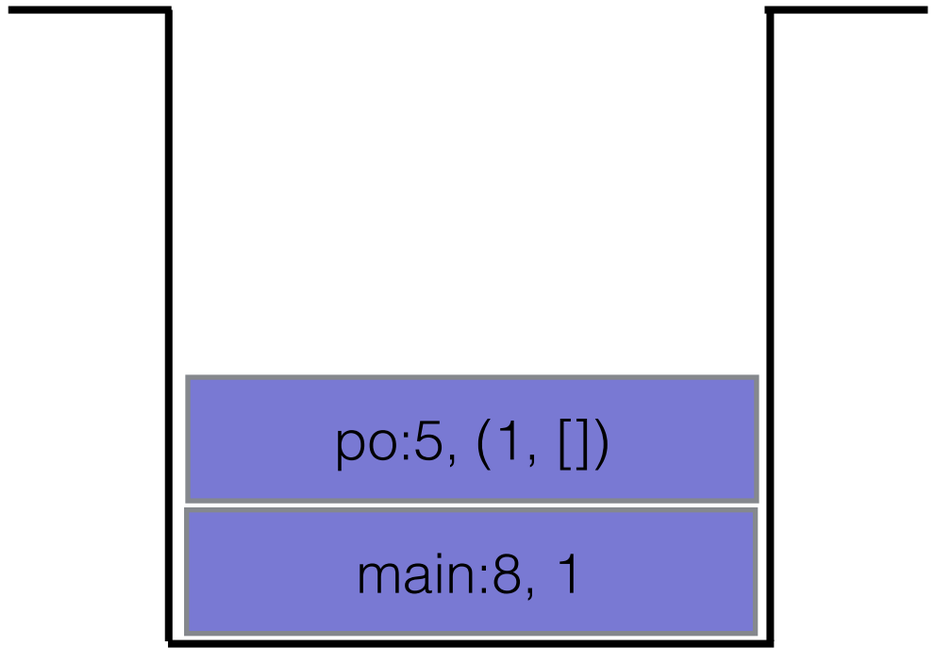


재귀함수의 실행

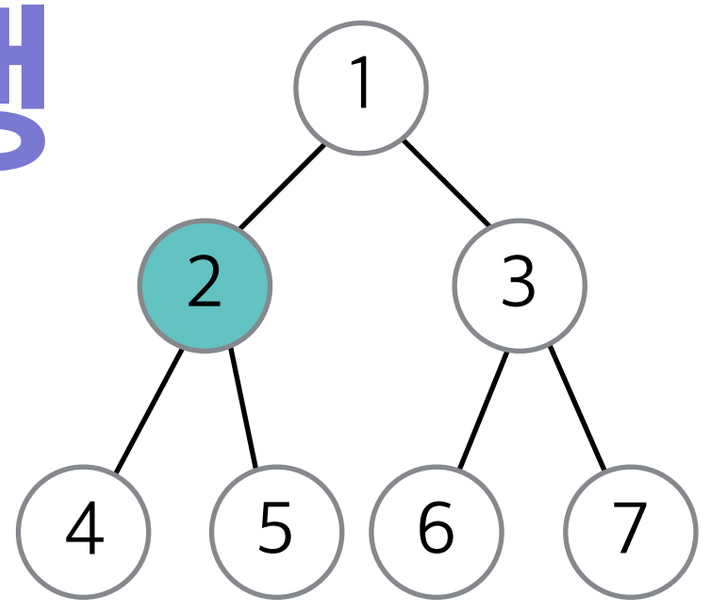



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[4]

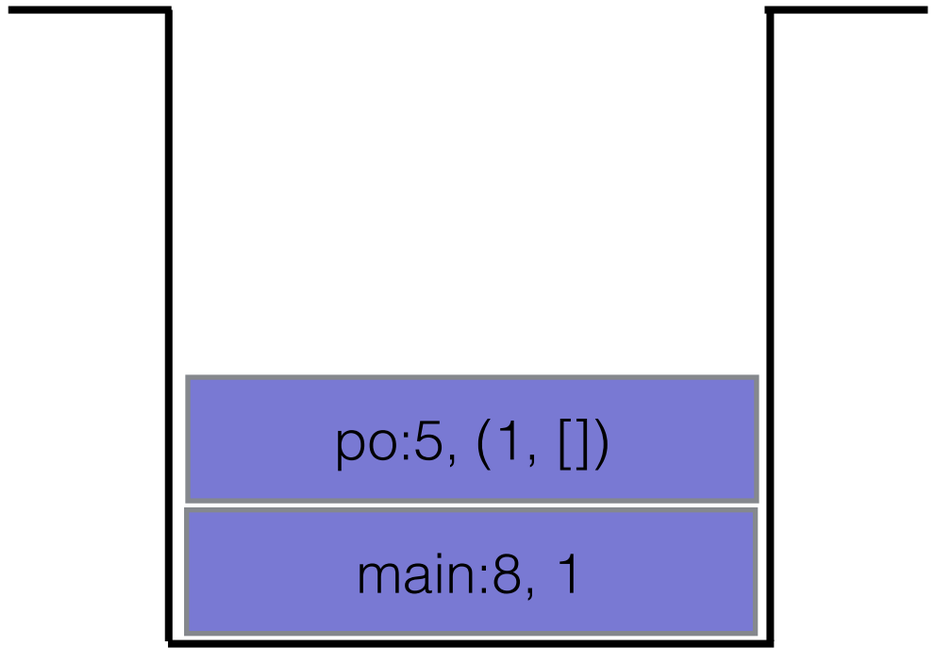


재귀함수의 실행

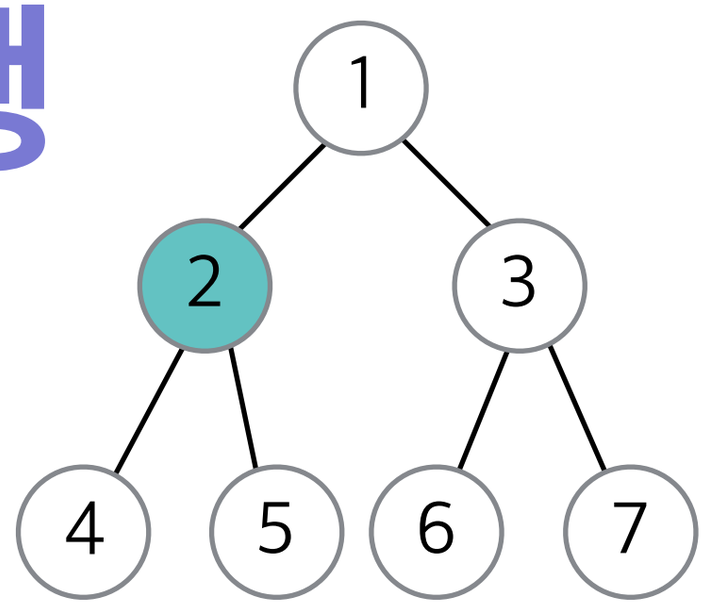



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[4, 5]

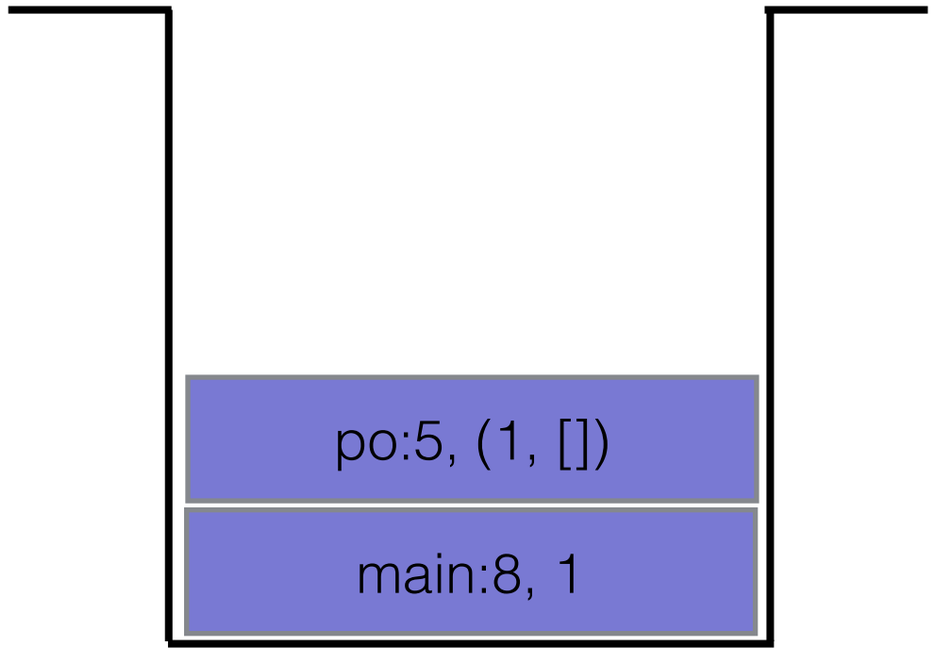


재귀함수의 실행

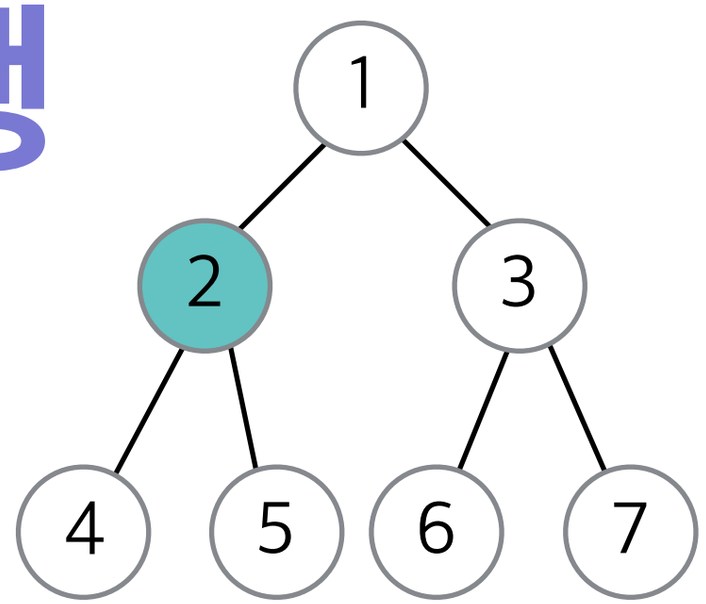



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
→ 8     return result
```

[4, 5, 2]

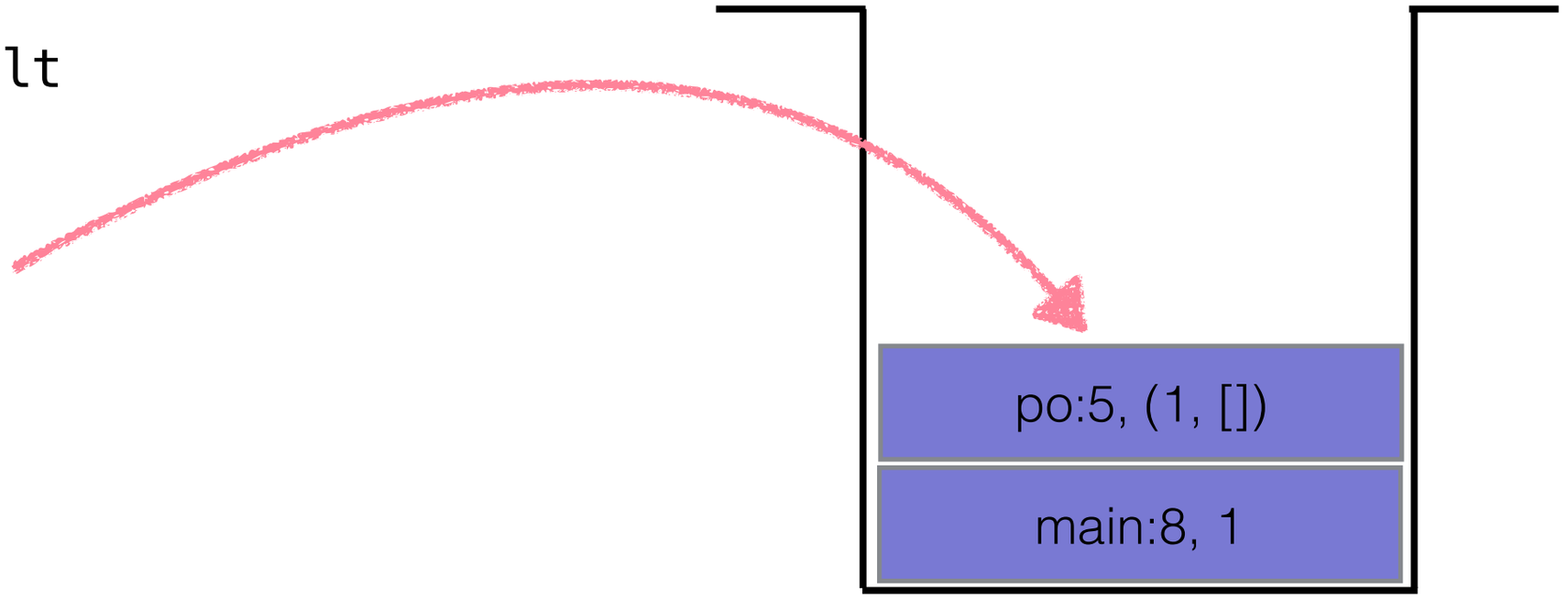


재귀함수의 실행

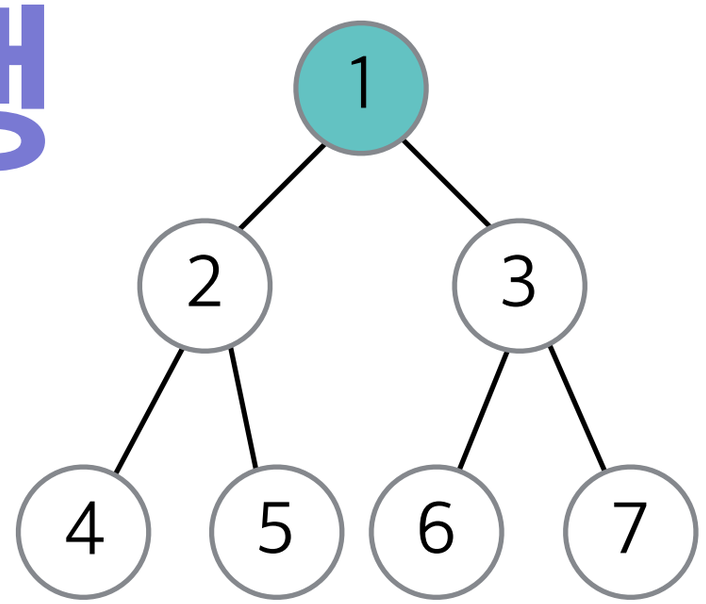



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

→ 8     return result
```

[4, 5, 2]

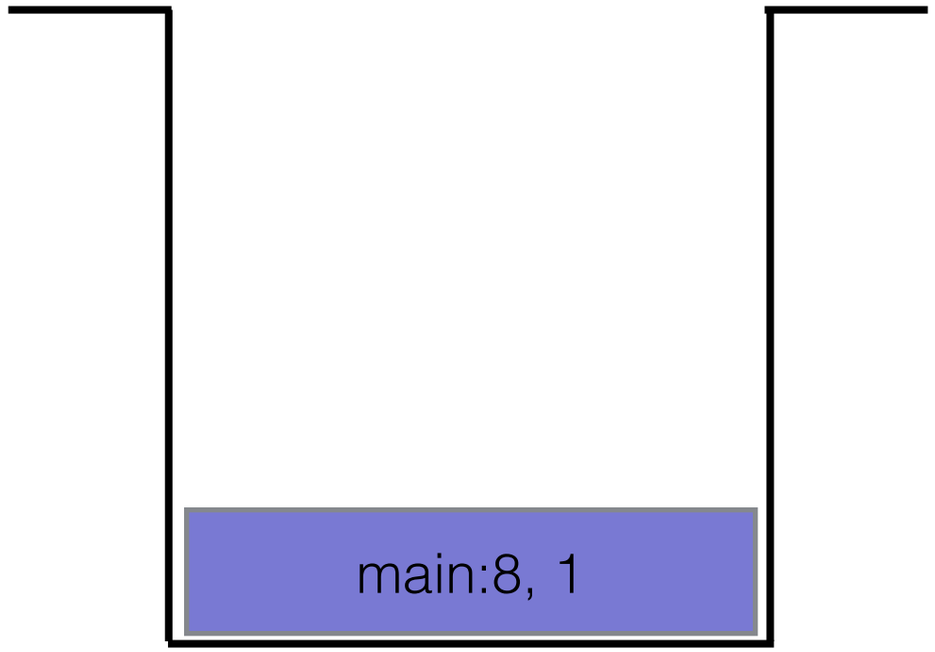


재귀함수의 실행

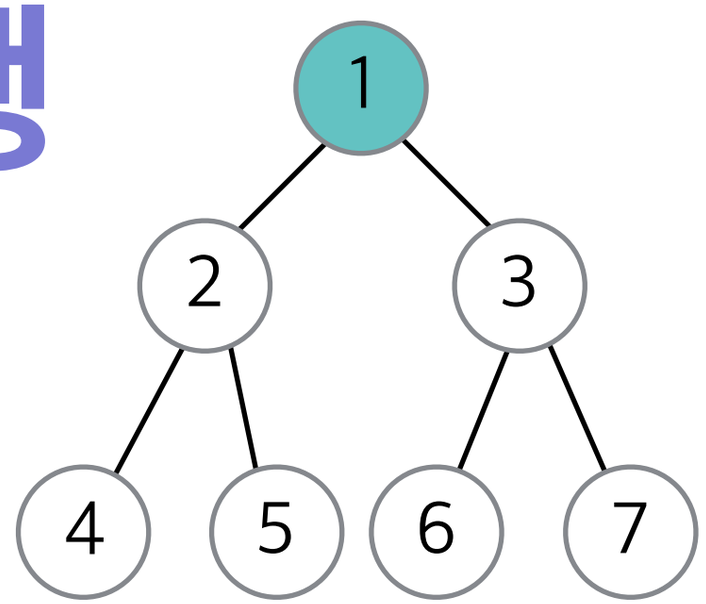



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[]

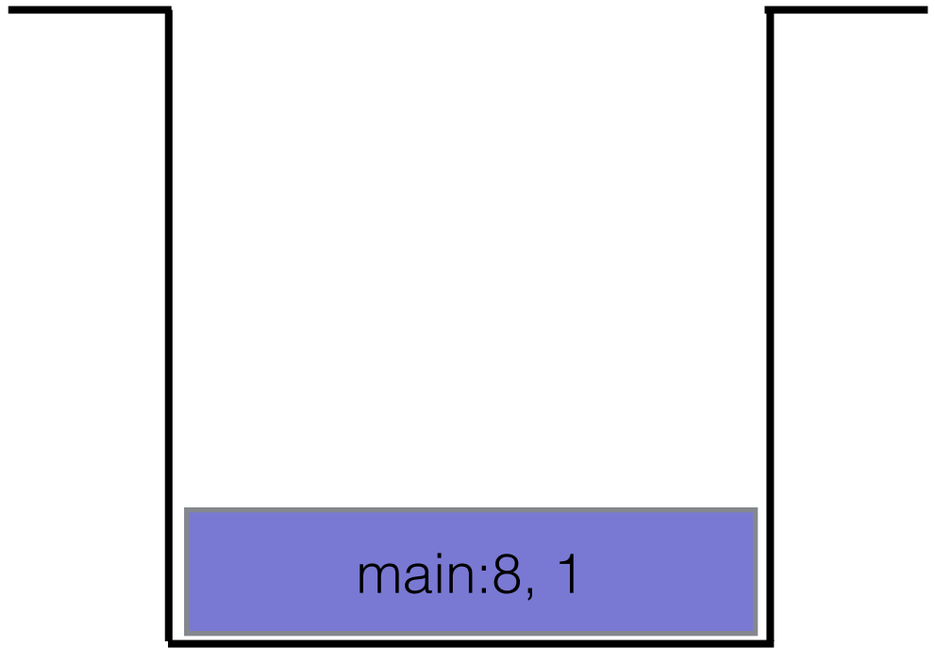


재귀함수의 실행

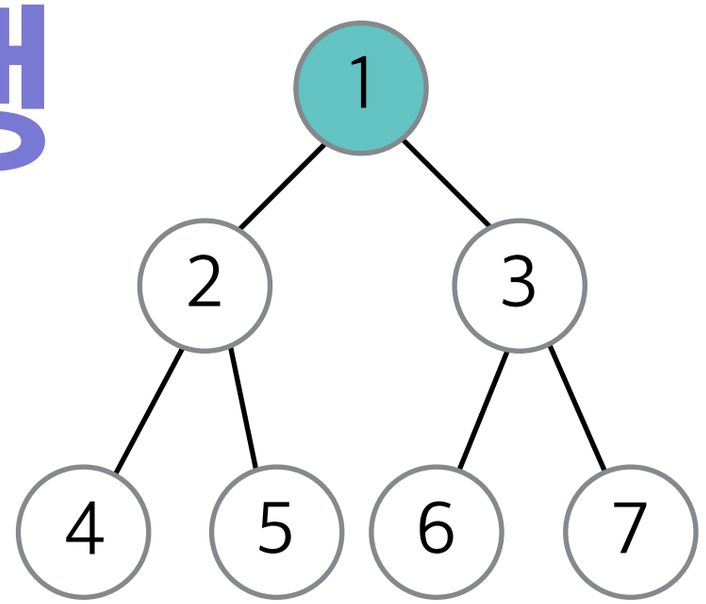



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)
8       return result
```

[4, 5, 2]

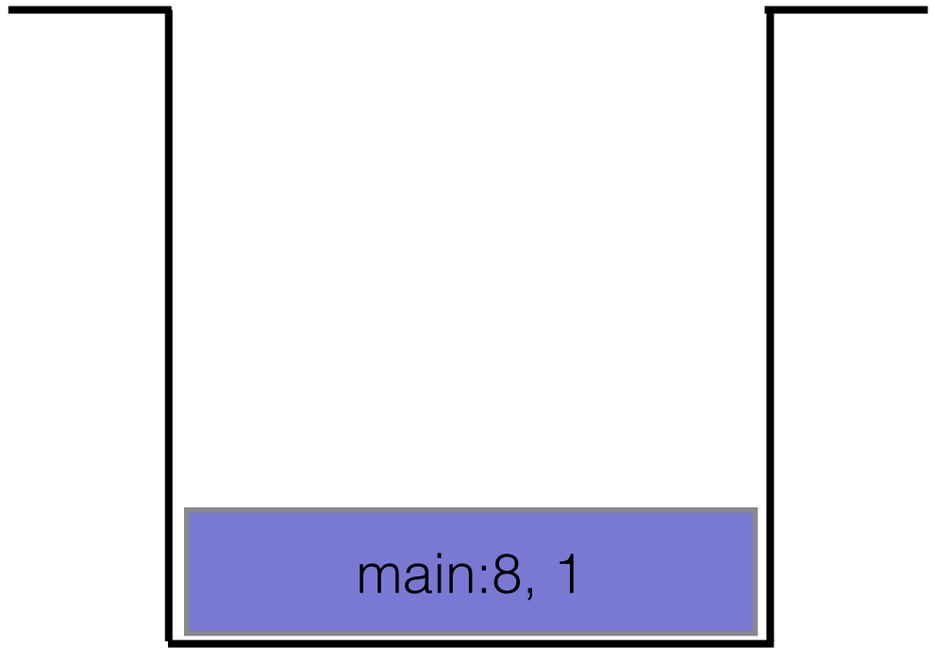


재귀함수의 실행

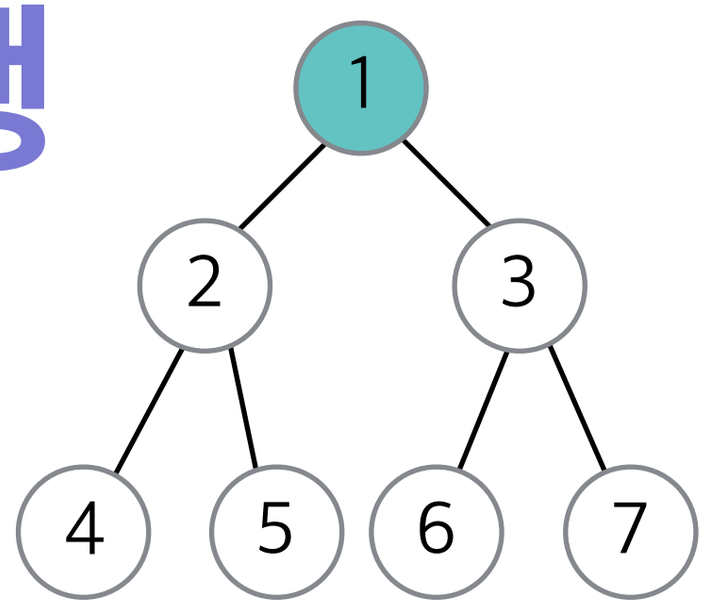



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
→ 7     result.append(tree.index)
8       return result
```

[4, 5, 2, 6, 7, 3]

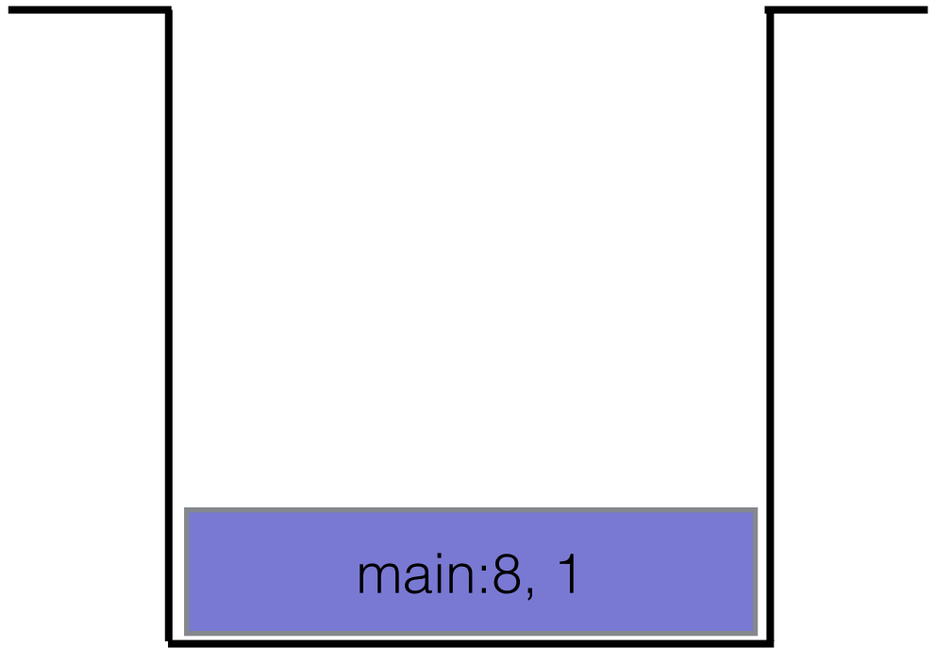


재귀함수의 실행

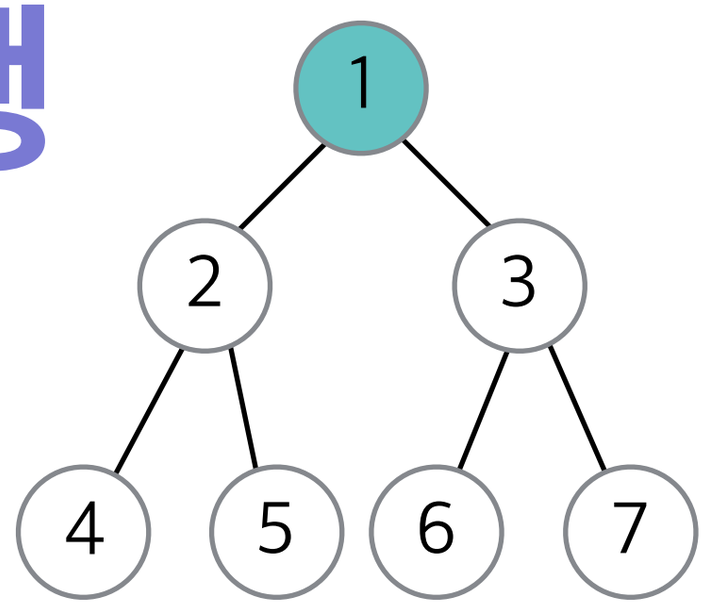



```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

→ 8     return result
```

[4, 5, 2, 6, 7, 3, 1]

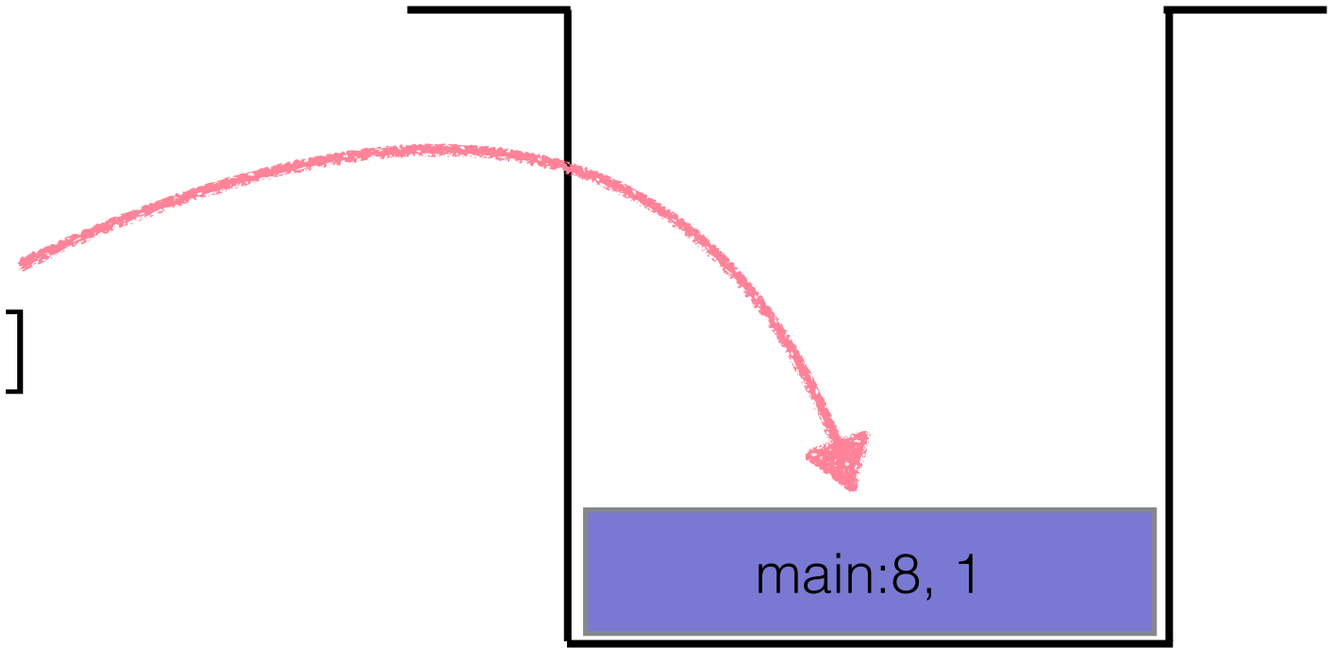


재귀함수의 실행




```
1   def postorder(tree) :
2       result = []

3       if tree == None :
4           return []

5       result = postorder(tree.left)
6       result = result + postorder(tree.right)
7       result.append(tree.index)

→ 8     return result
```

[4, 5, 2, 6, 7, 3, 1]



재귀함수의 올바른 디자인 및 해석

재귀함수를 디자인하기 위해서는 다음 세 가지 단계를 명심하자

1. 함수의 정의를 명확히 한다.

2. 기저 조건(Base condition)에서 함수가 제대로 동작하게 작성한다.

3. 그 후, 함수가 (작은 input에 대하여) 제대로 동작한다고 가정하고 함수를 완성한다.

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \ldots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

getPower(m, n) = (getPower(m, n//2))^2 n이 짝수

getPower(m, n) = m x getPower(m, n-1) n이 홀수

[예제 3] 거듭제곱 구하기

$$m^n = m \times m \times \ldots \times m$$

getPower(m, n) : m^n 을 반환하는 함수

```
def getPower(m, n) :
    if n == 0 :
        return 1
    elif n % 2 == 0 :
        temp = getPower(m, n//2)
        return temp * temp
    else :
        return getPower(m, n-1) * m
```

[예제 3] 거듭제곱 구하기

$m^n = m \times m \times \ldots \times m$

getPower(m, n) : m^n 을 반환하는 함수

```
def getPower(m, n) :
    if n == 0 :
        return 1
    elif n % 2 == 0 :
        temp = getPower(m, n//2)
        return temp * temp
    else :
        return getPower(m, n-1) * m
```

O(log n)

요약

의미단위로 작성된 코드가 좋은 코드이다

→ 코드를 이해한다 = 각 함수가 무슨 일을 하는지 설명할 수 있다

트리는 코드가 실행되고 있는 상태를 나타내는 자료구조이다

→ 물론, 코드를 의미 단위로 나타냈을 때 파악이 가능한 사실이다

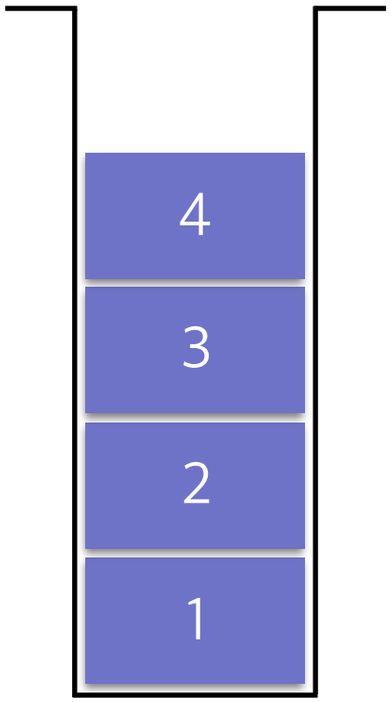
코드를 하나하나 따라가는 것은 컴퓨터가 해야 할 일이다

→ 우리는 앞으로 코드가 하는 일, 더 나아가 코드의 의미에 집중한다

컴퓨터를 이용한 문제 해결 과정

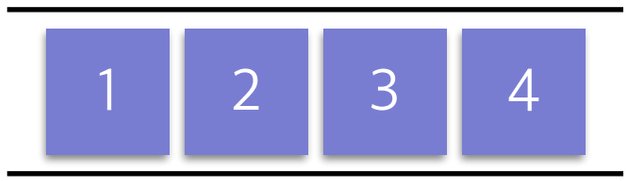
1. 문제를 정확히 이해한다
2. 문제를 해결하는 알고리즘을 개발한다
3. 알고리즘이 **문제를 해결한다는 것을 증명한다**
4. 알고리즘이 **제한시간 내에 동작한다는 것을 보인다**
5. 알고리즘을 코드로 작성한다
6. 제출 후 만점을 받고 매우 기뻐한다

대표적인 자료구조



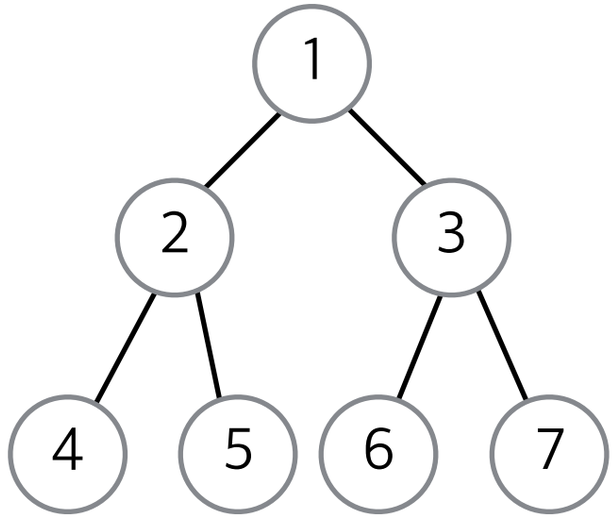
스택 (Stack)

Last In First Out

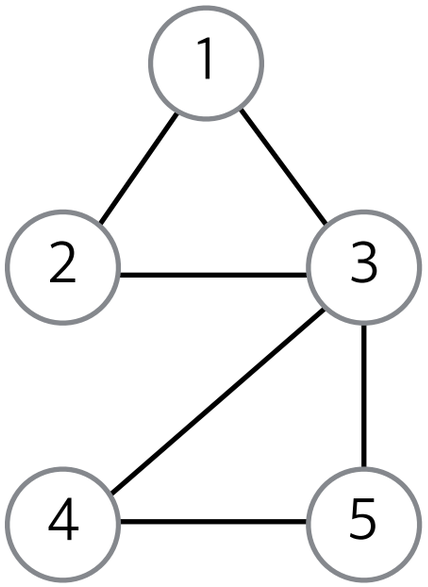


큐 (Queue)

First In First Out



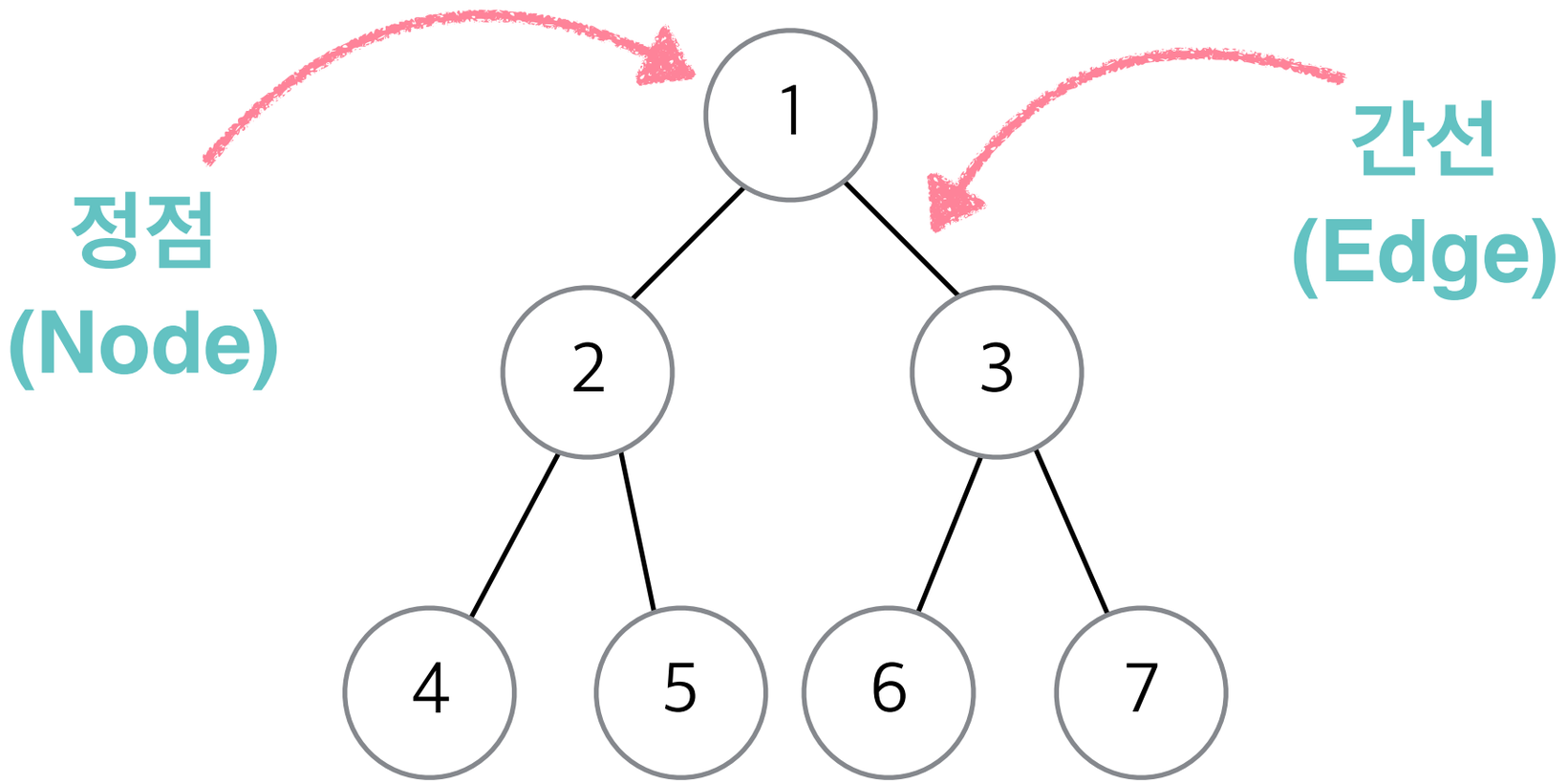
트리 (Tree)



그래프 (Graph)

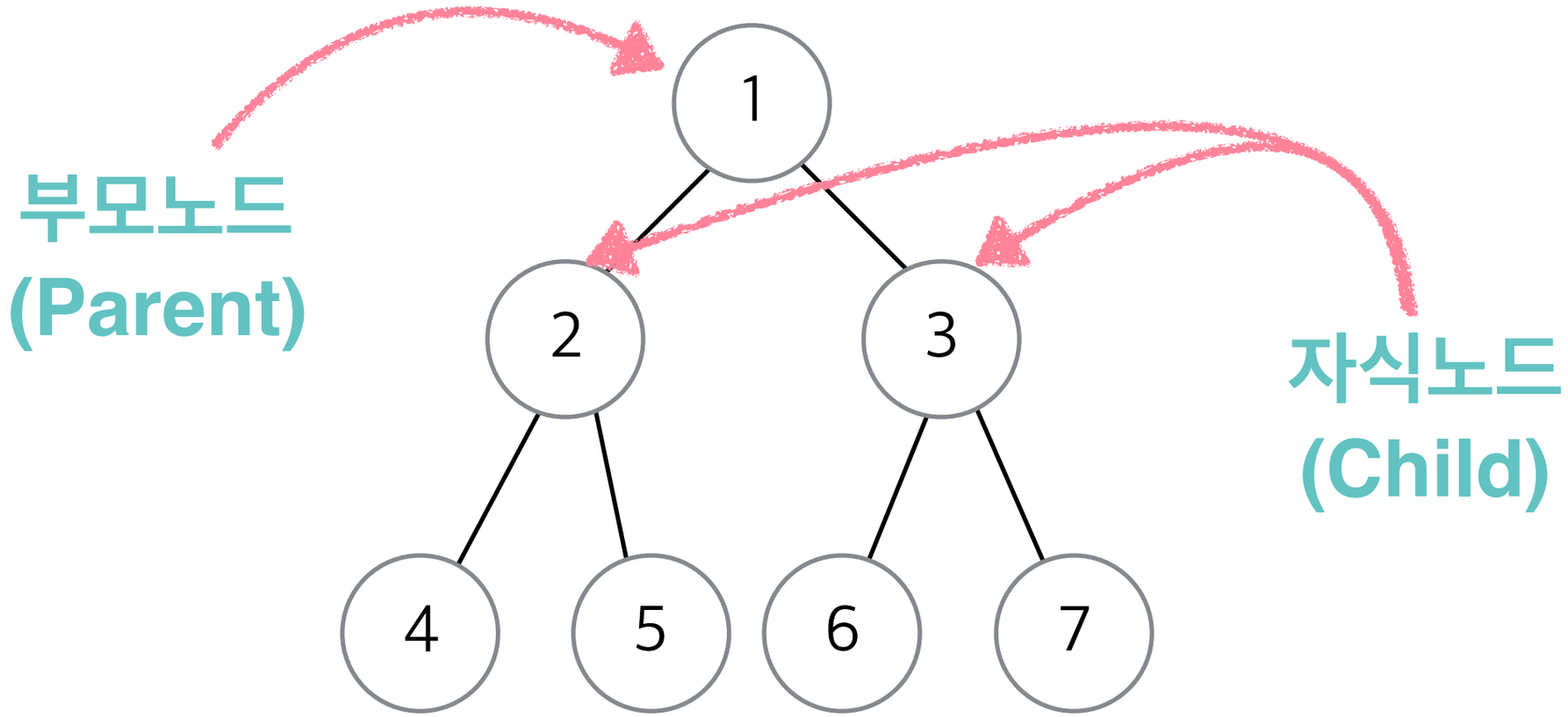
트리

아래와 같이 생긴 자료구조



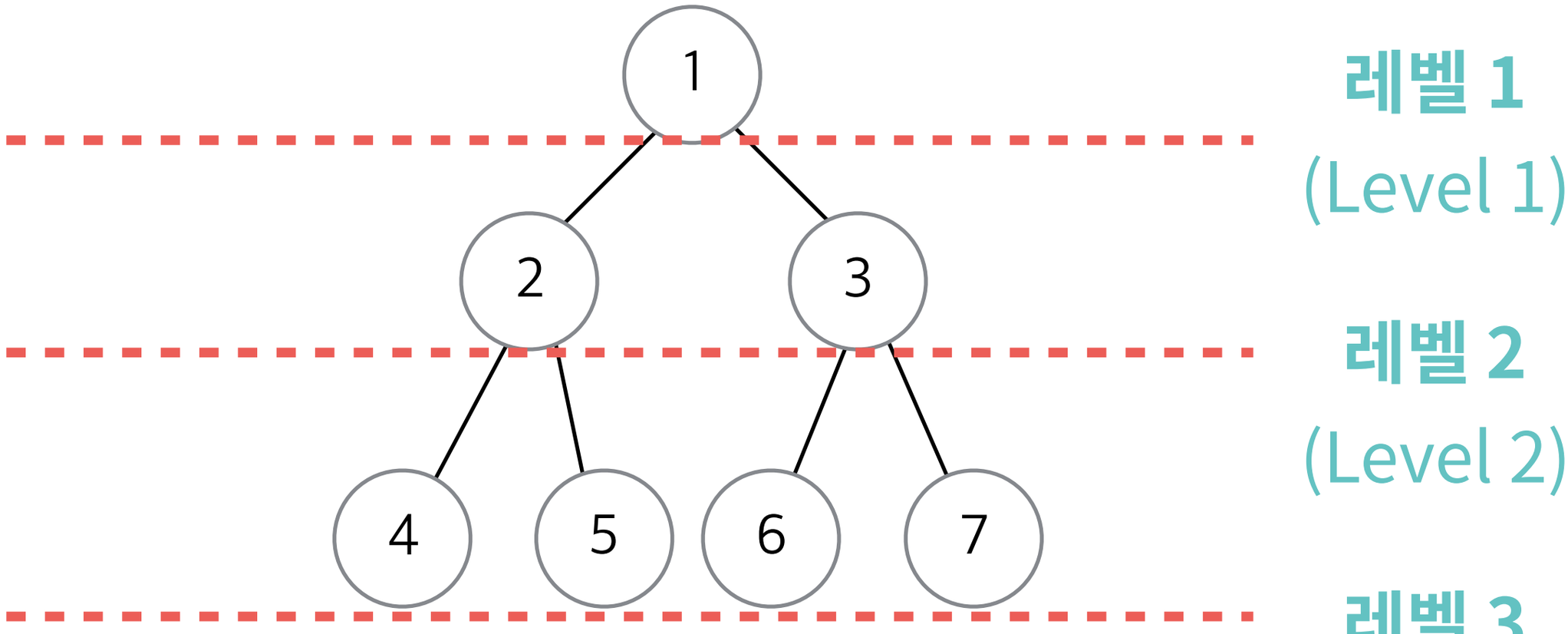
트리

아래와 같이 생긴 자료구조



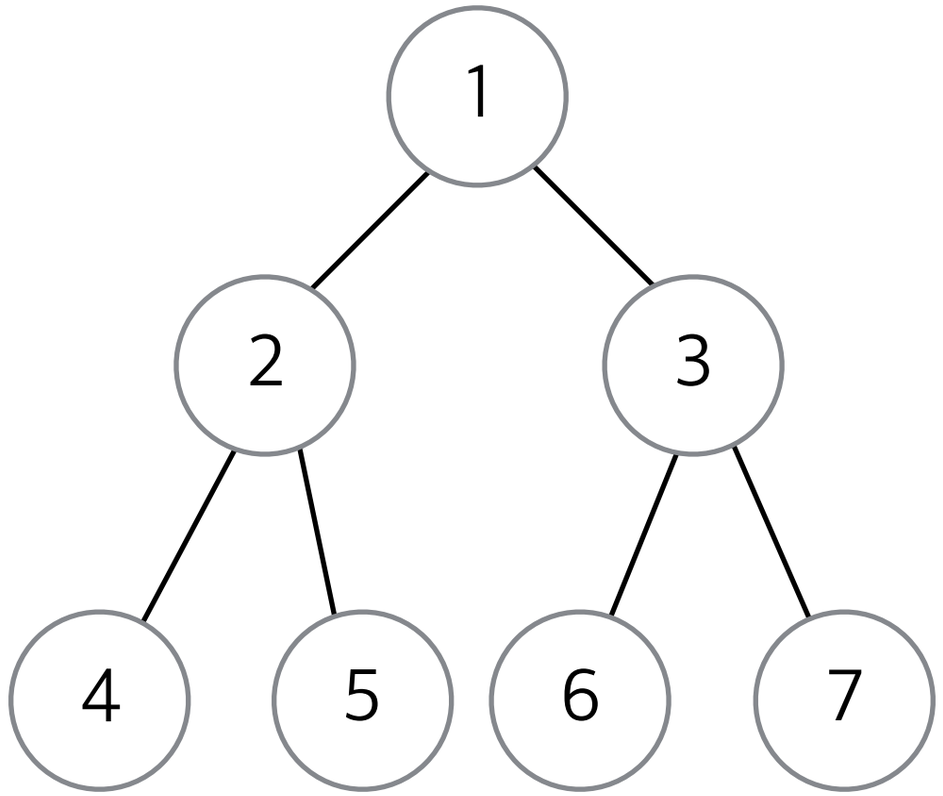
트리

아래와 같이 생긴 자료구조



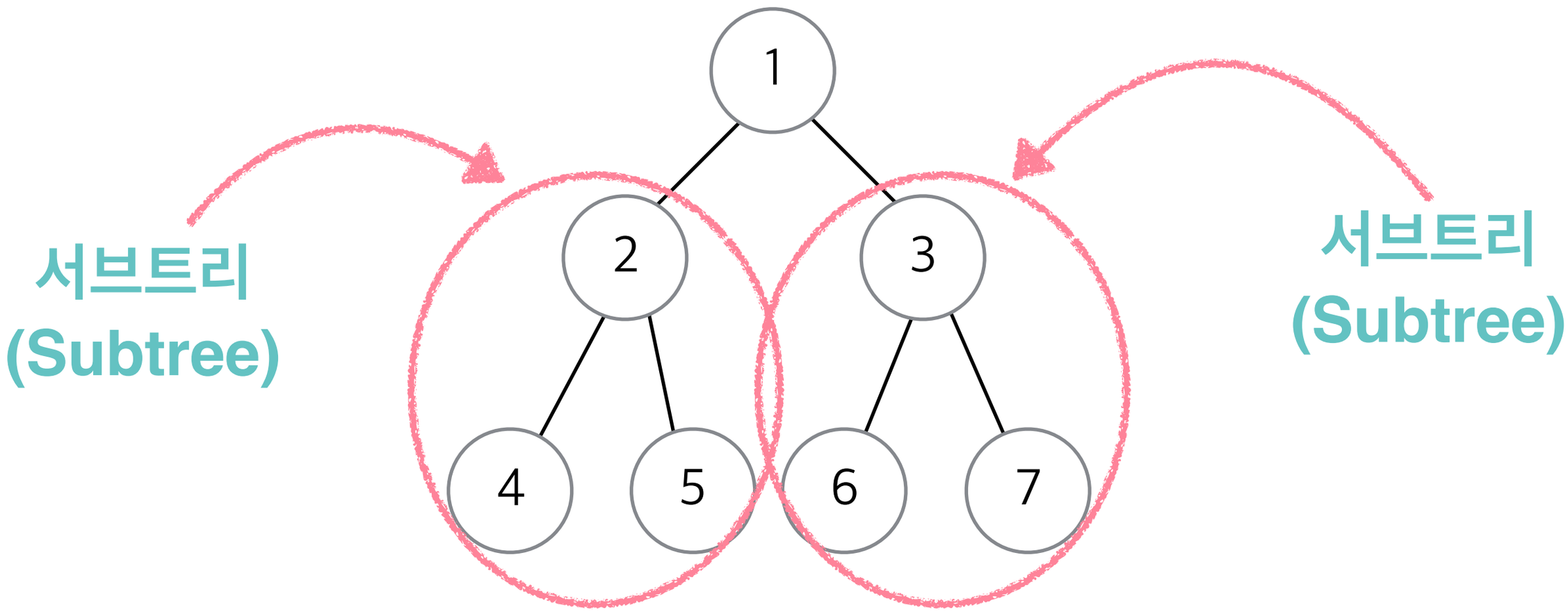
트리의 재귀적 성질

트리는 그 안에 또 트리가 존재한다



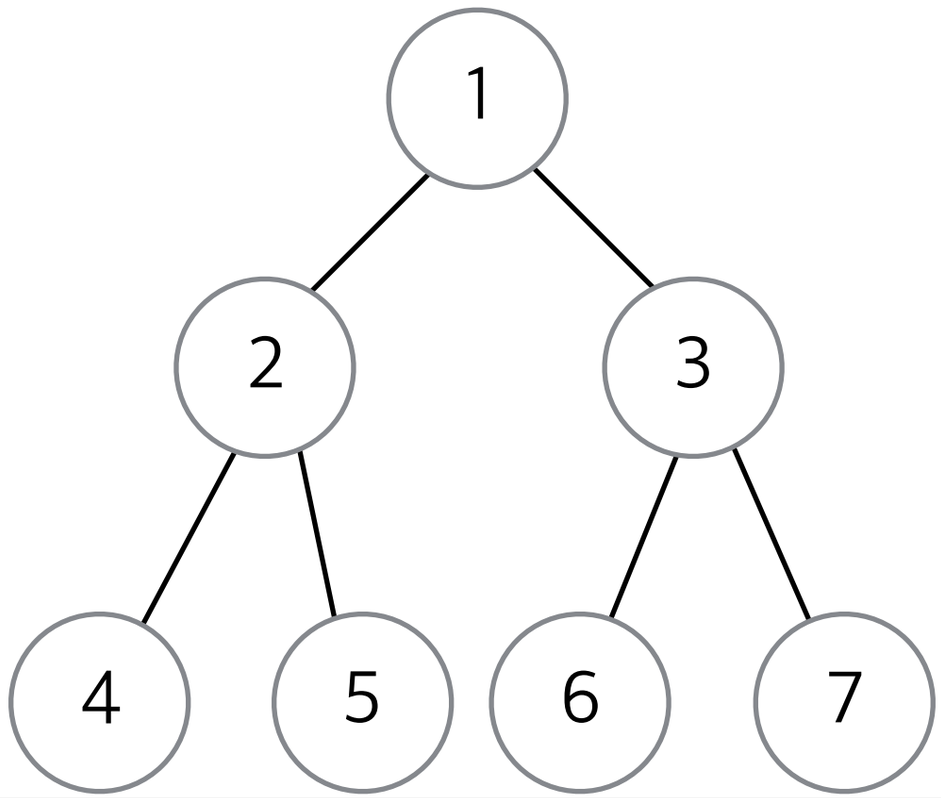
트리의 재귀적 성질

트리는 그 안에 또 트리가 존재한다



트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함



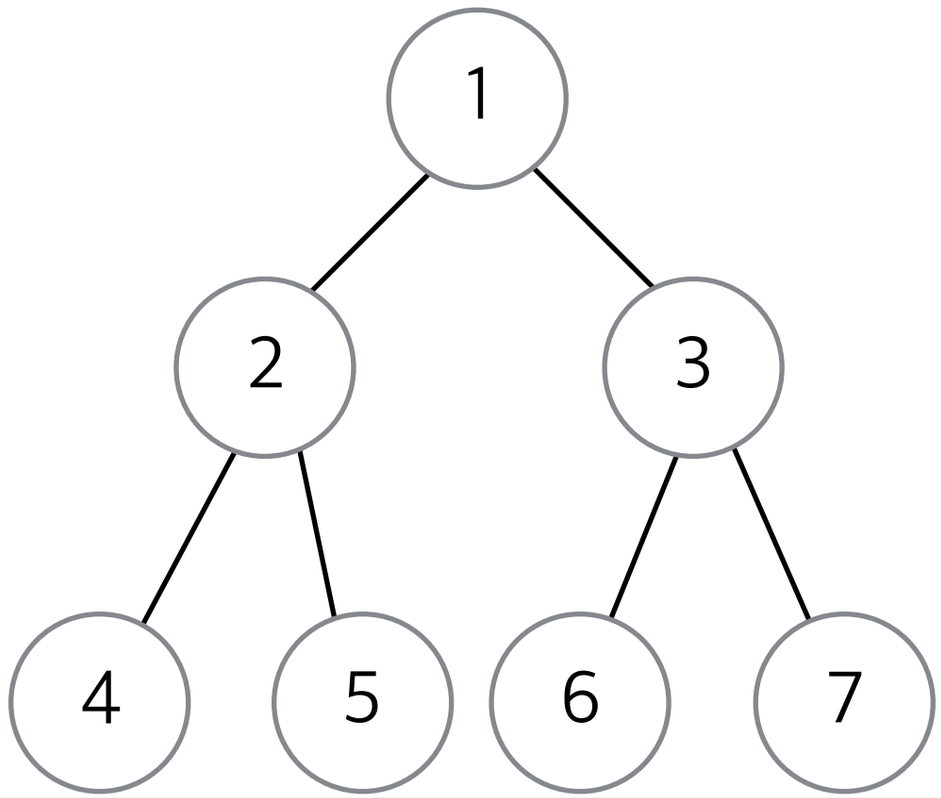
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



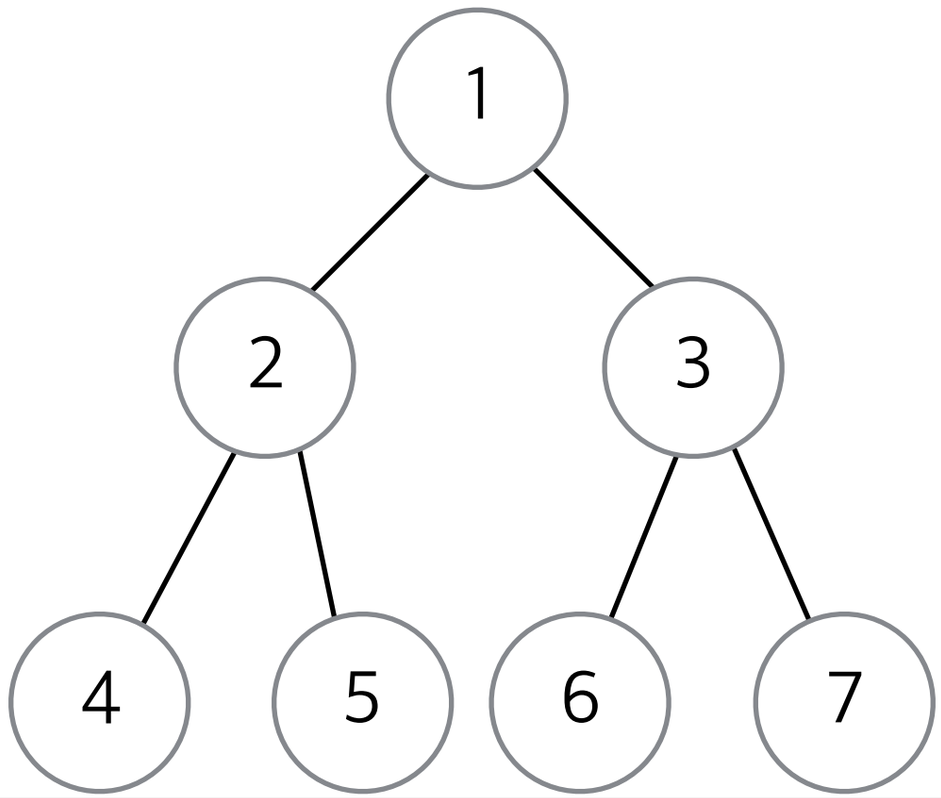
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



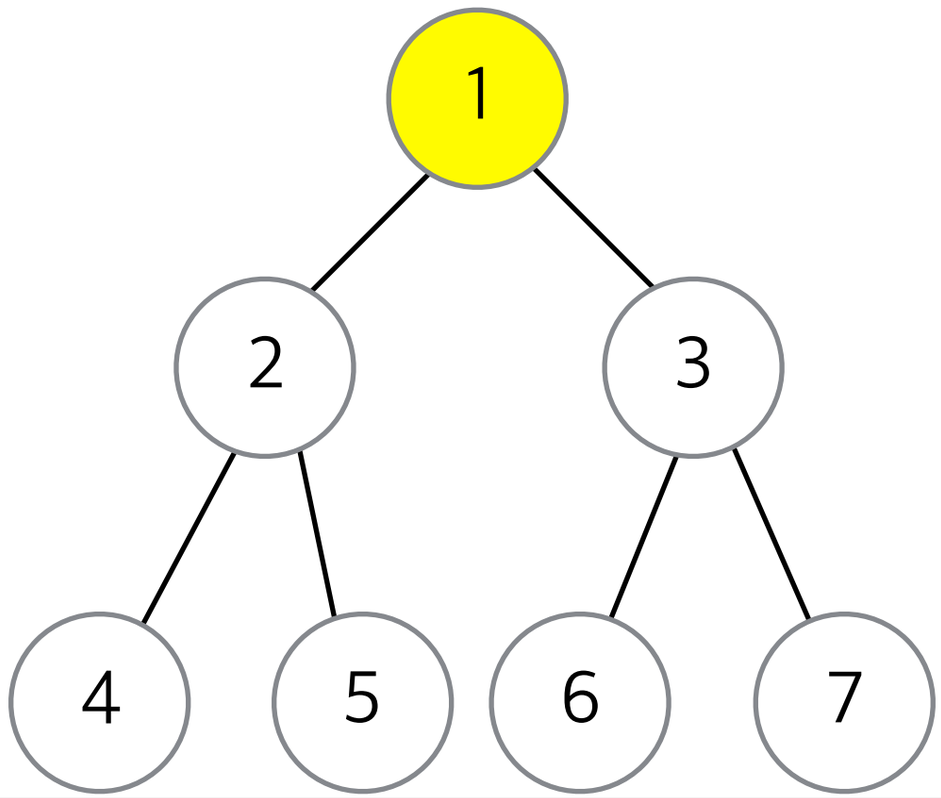
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



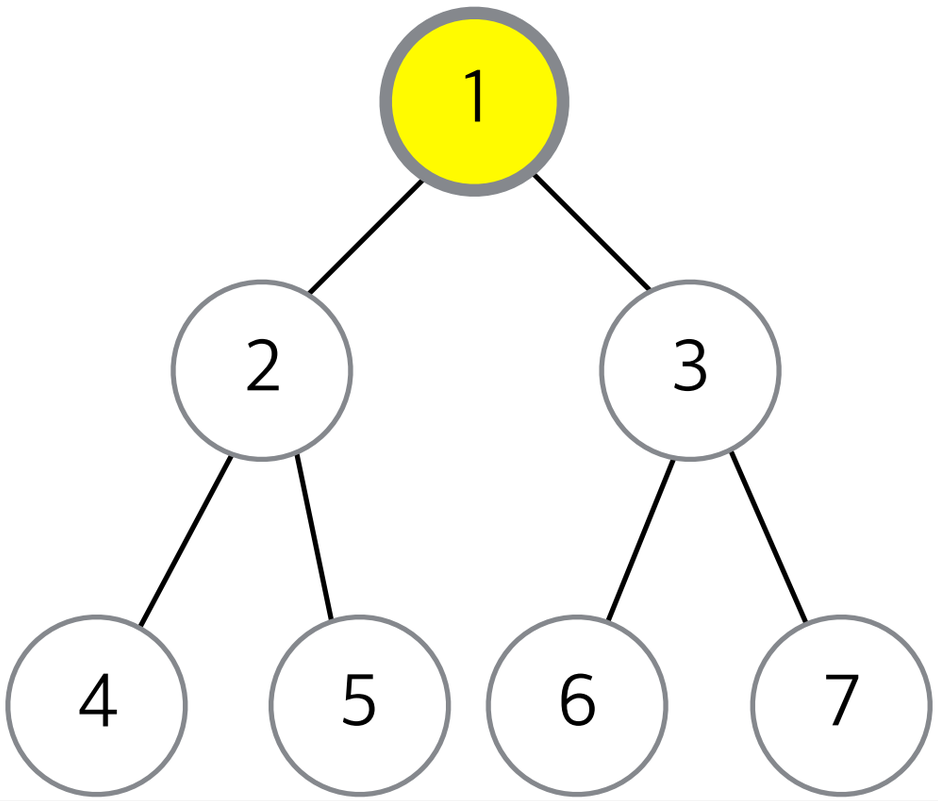
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



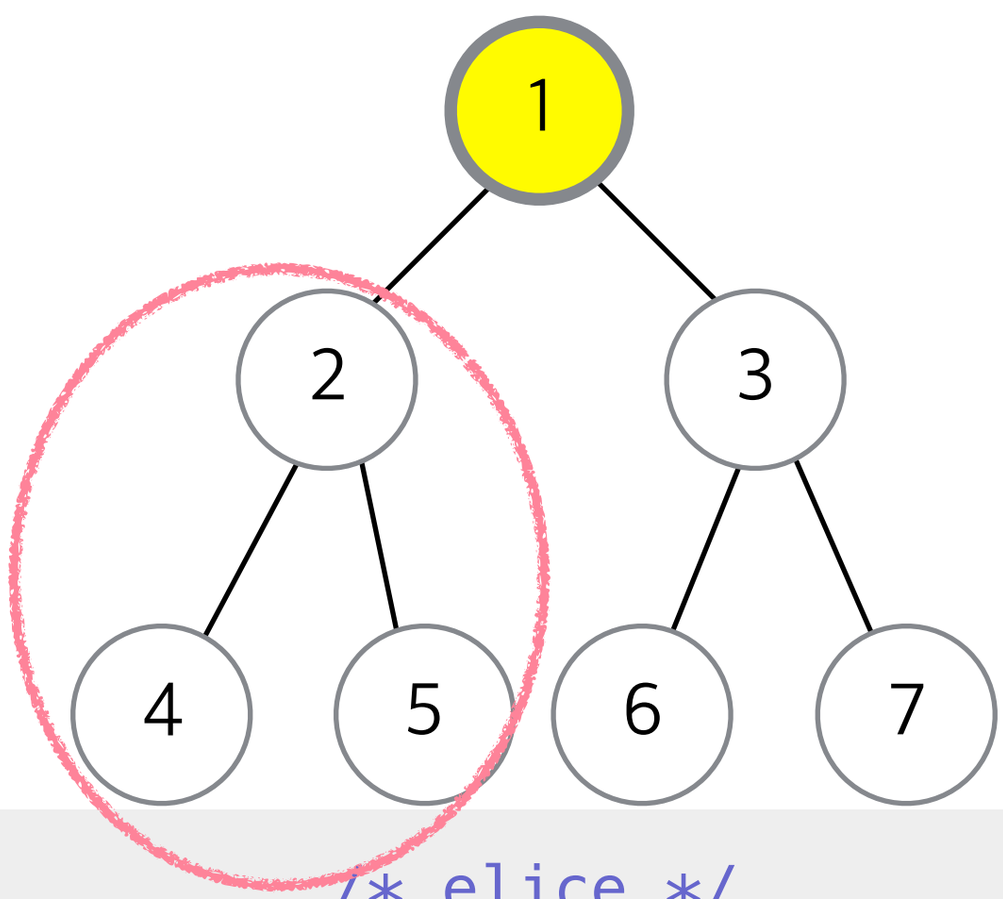
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



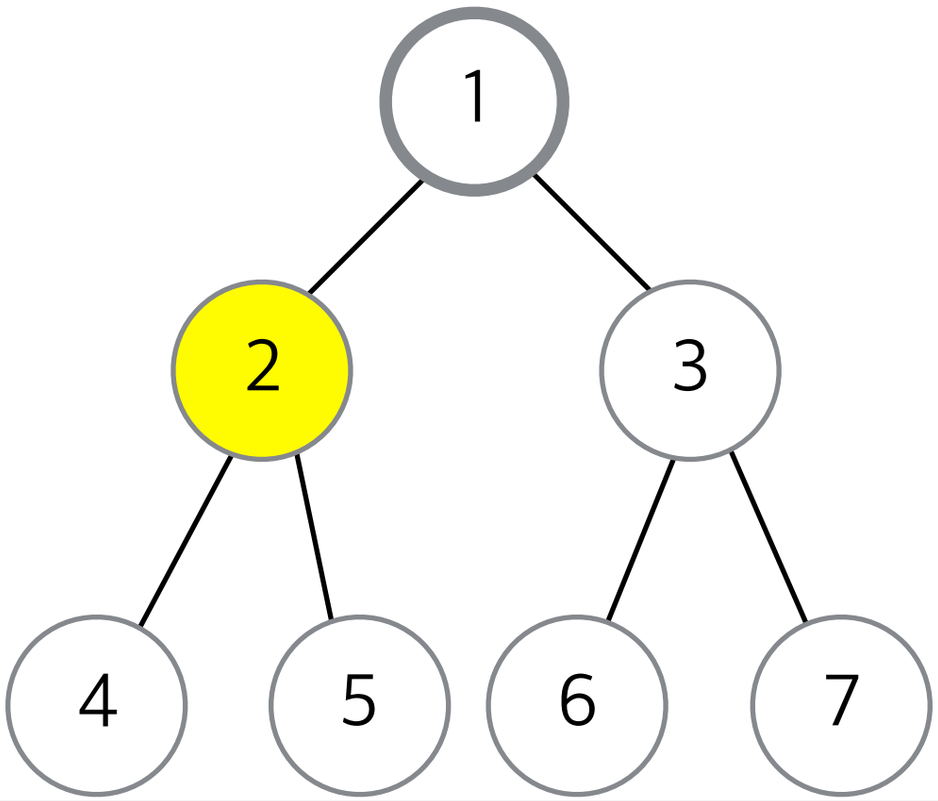
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



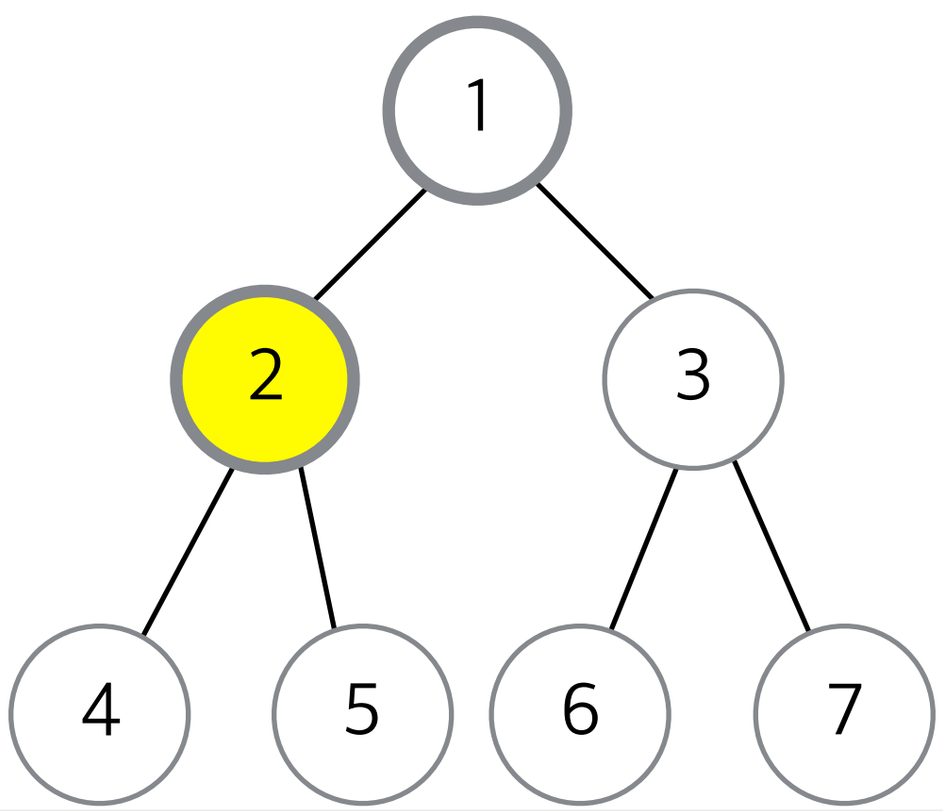
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



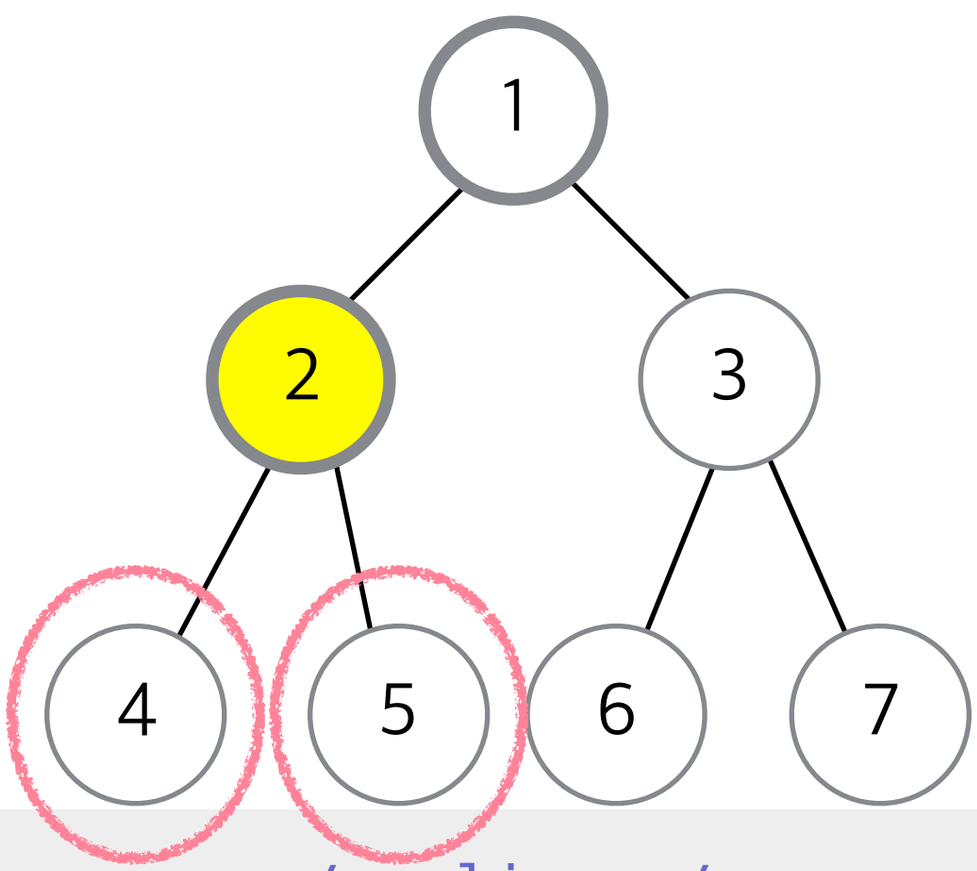
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



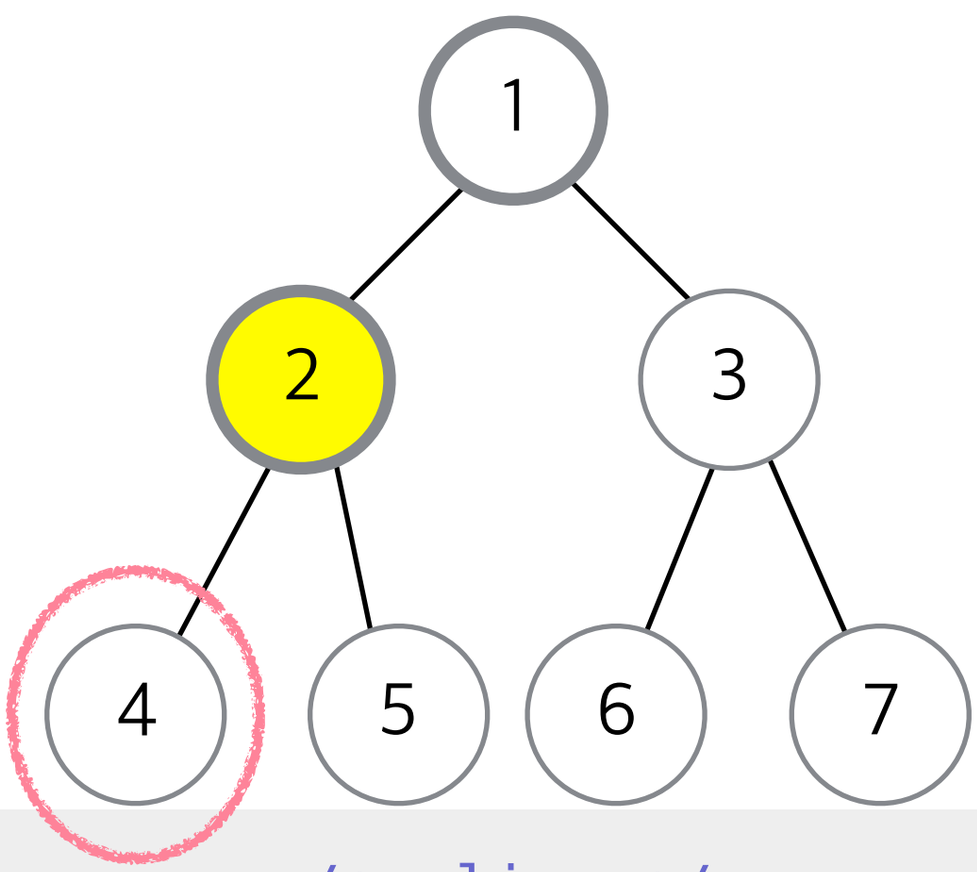
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



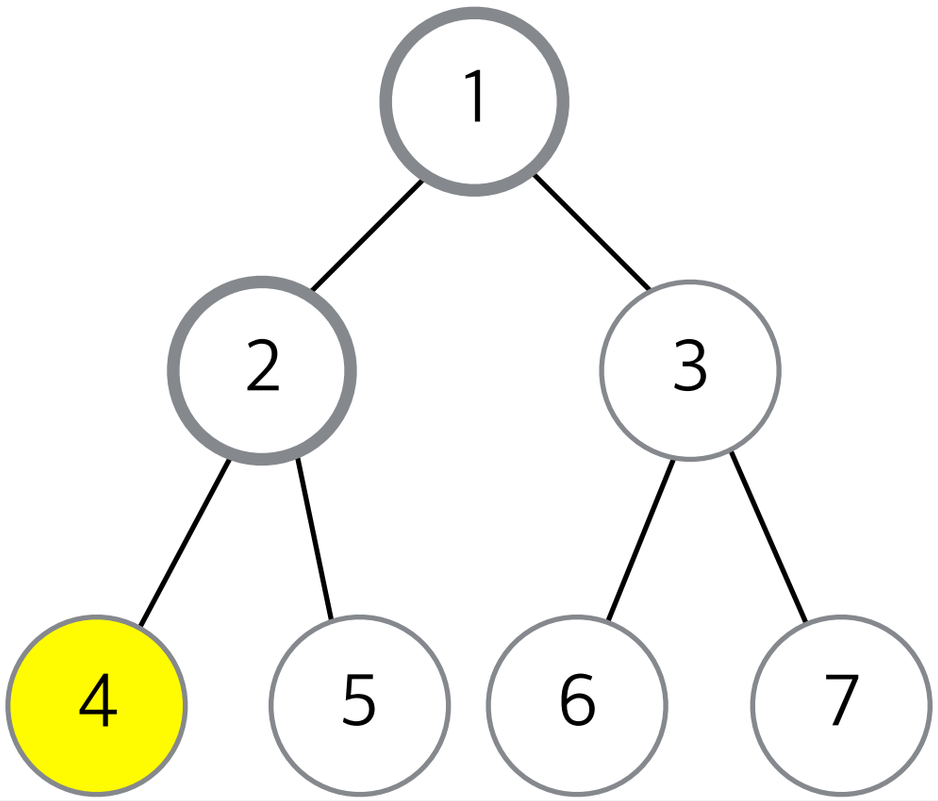
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



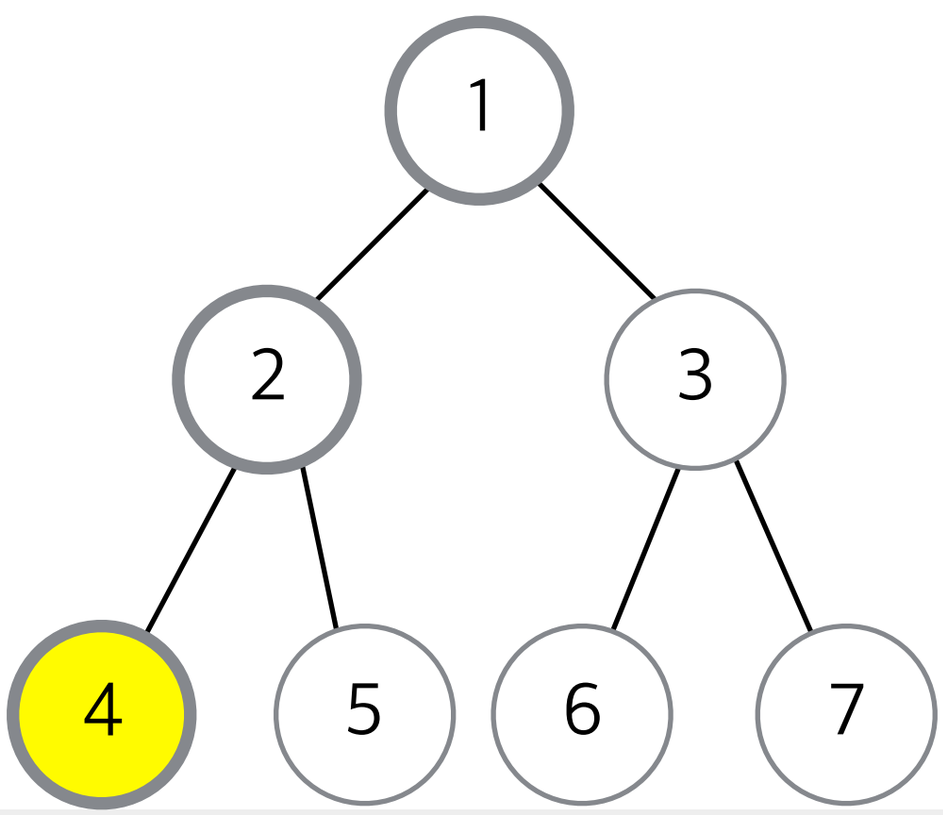
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



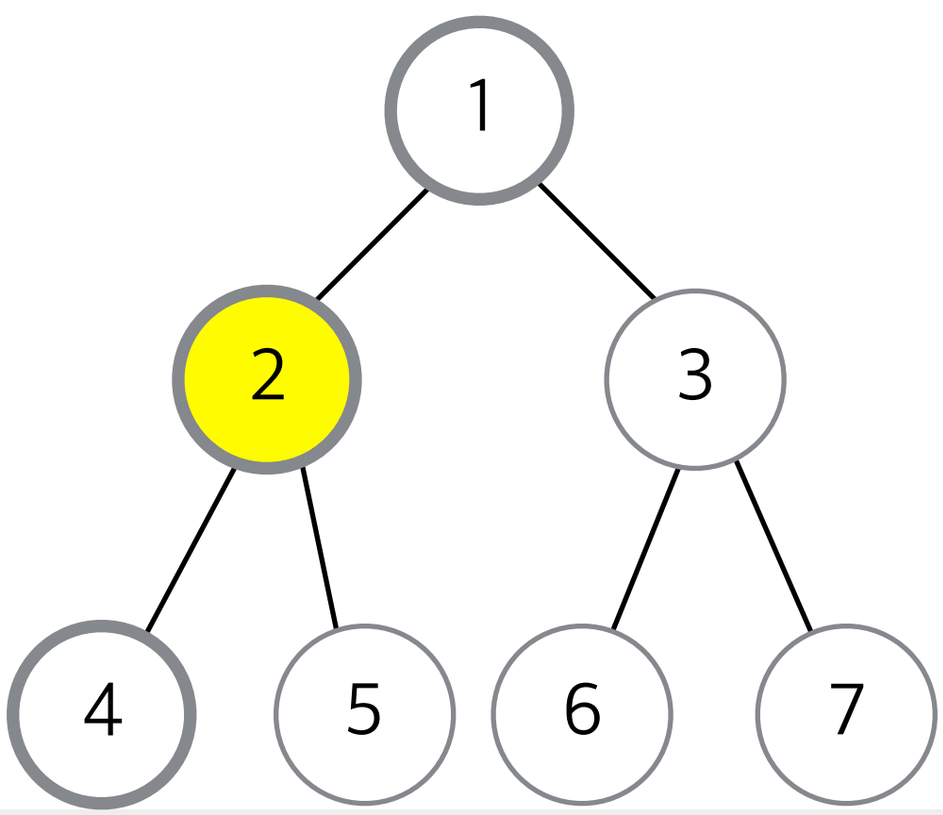
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



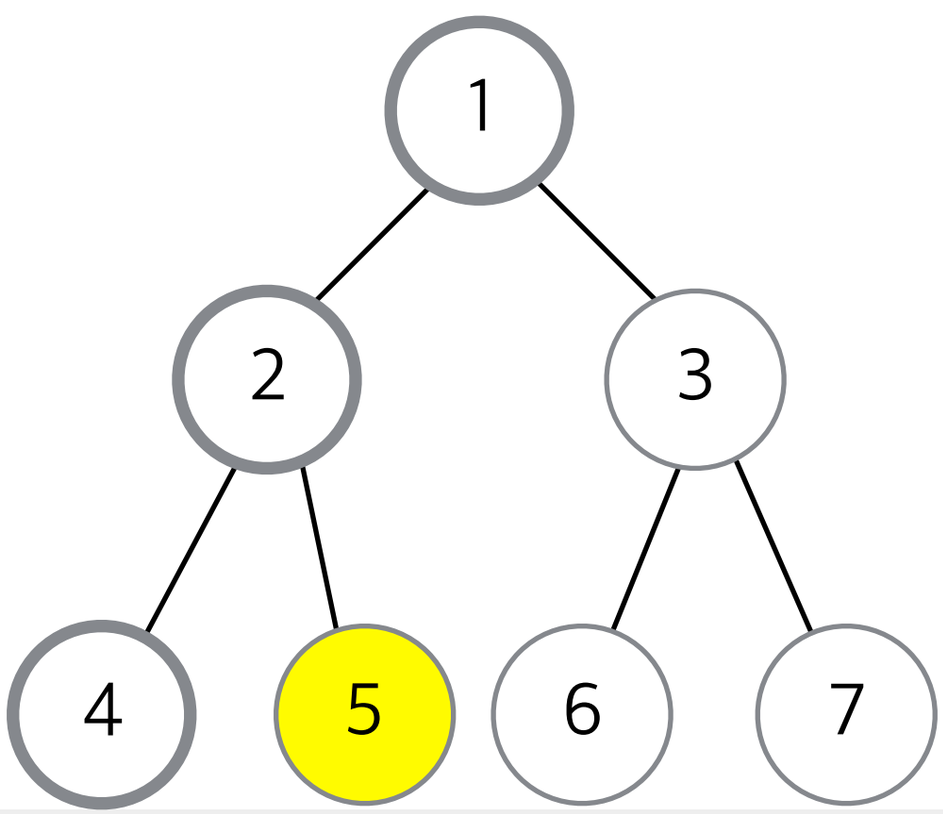
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



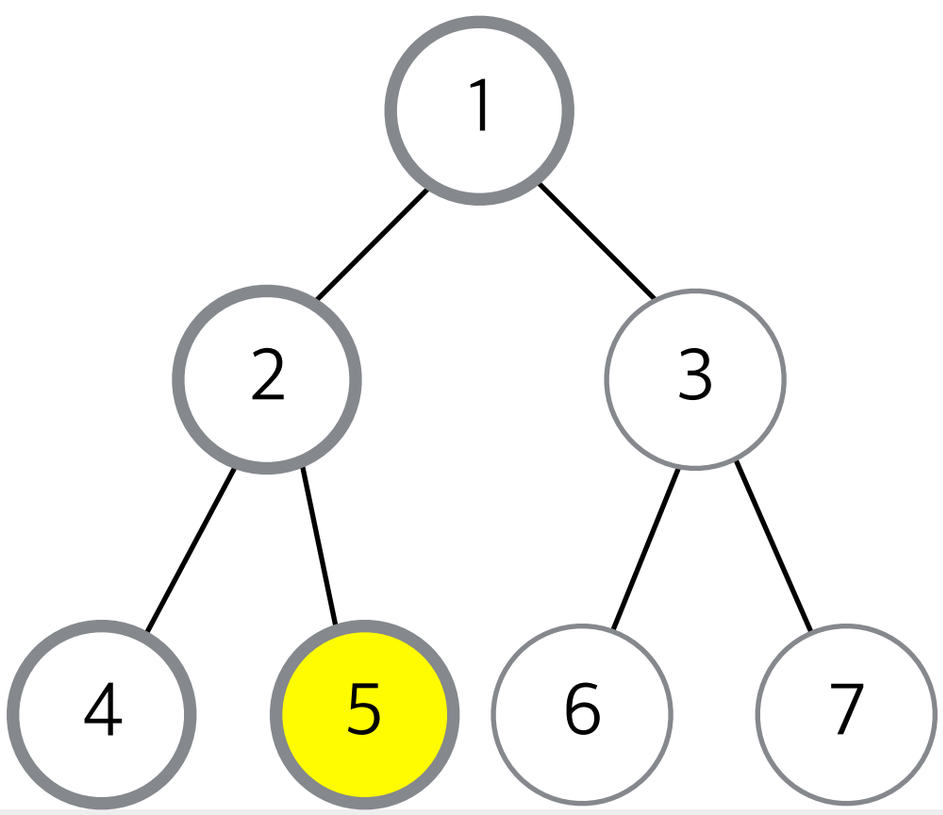
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



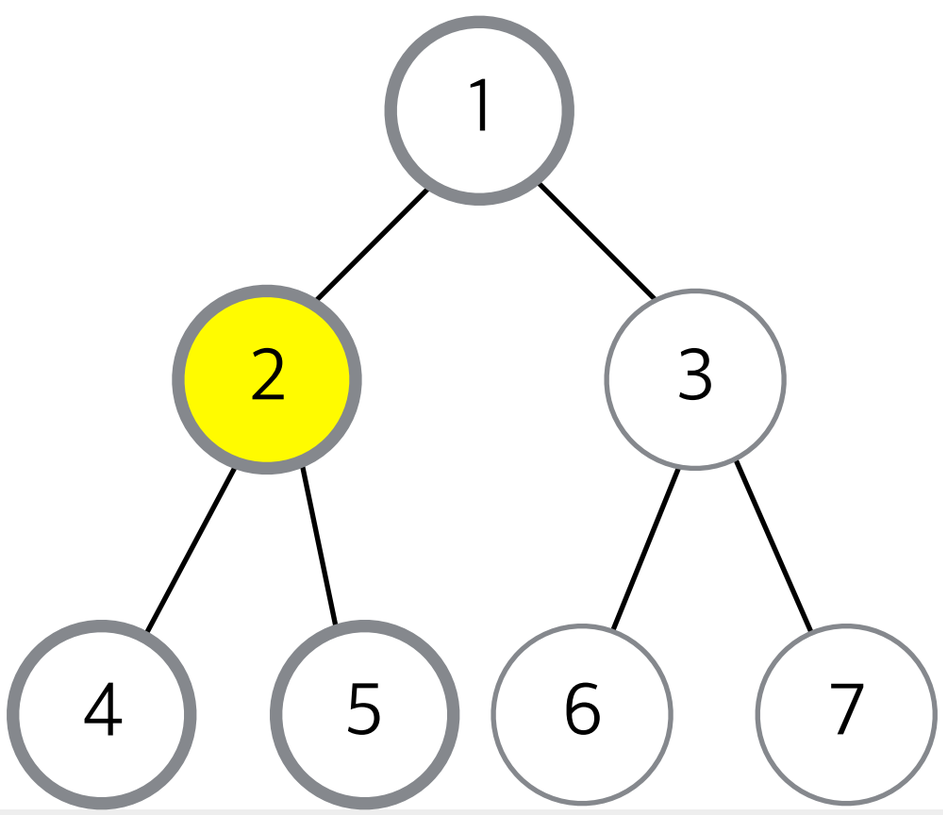
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



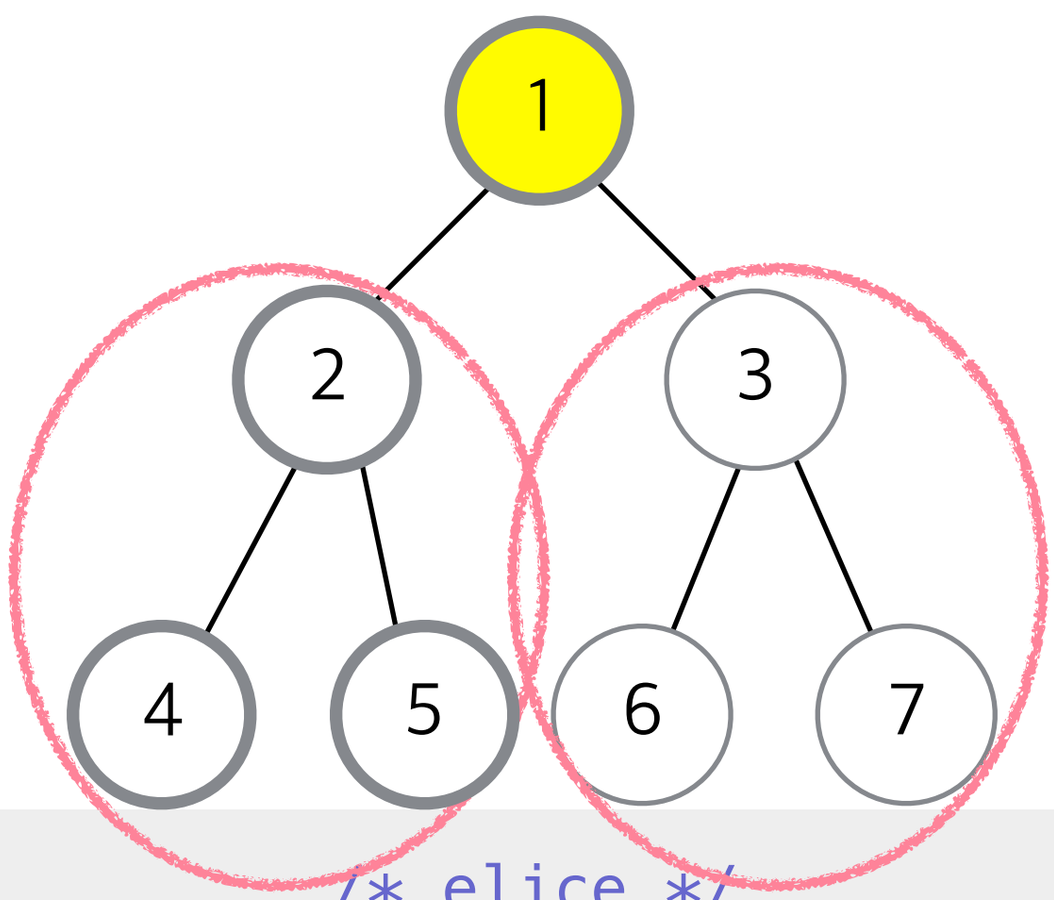
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



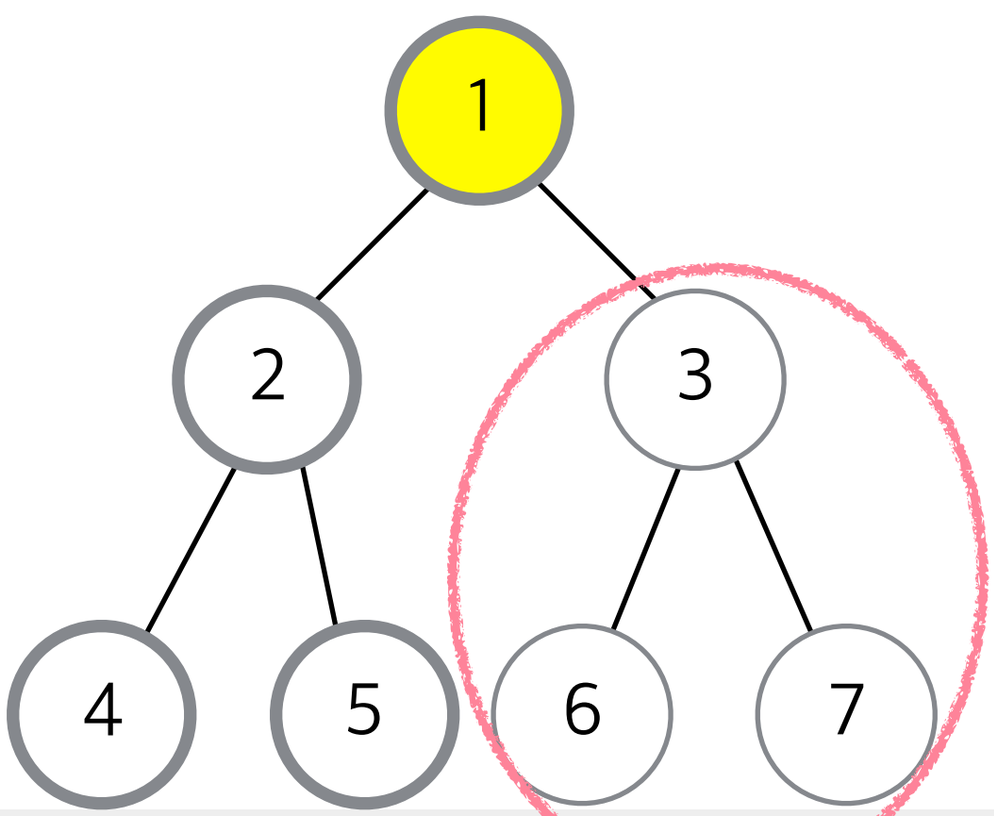
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



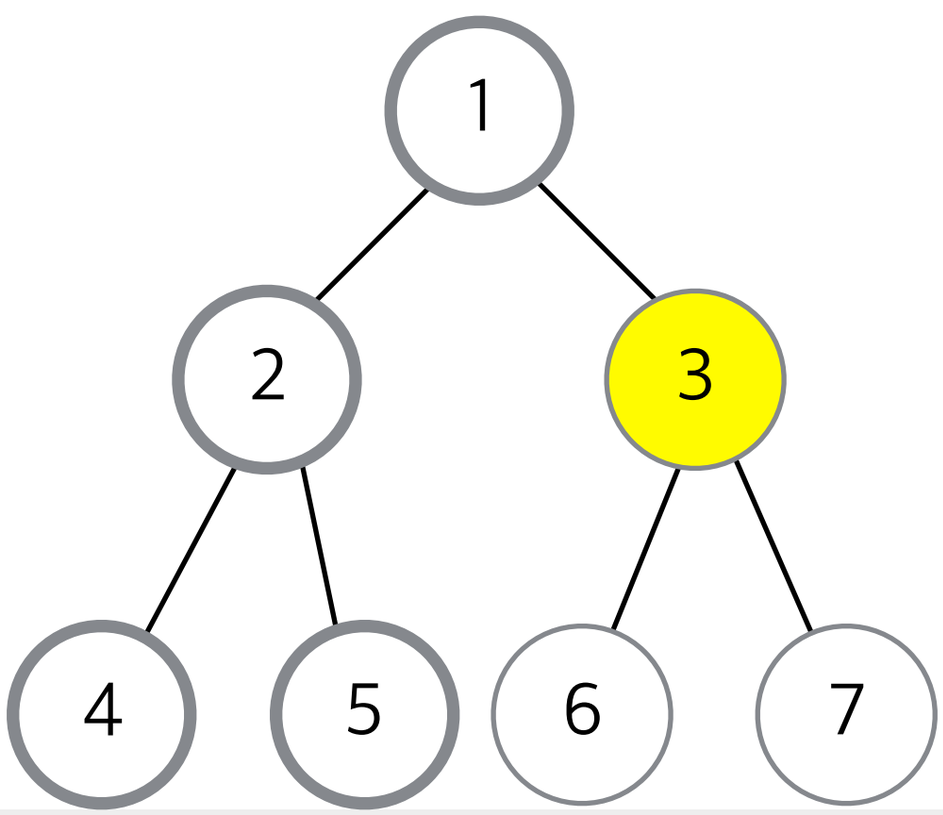
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



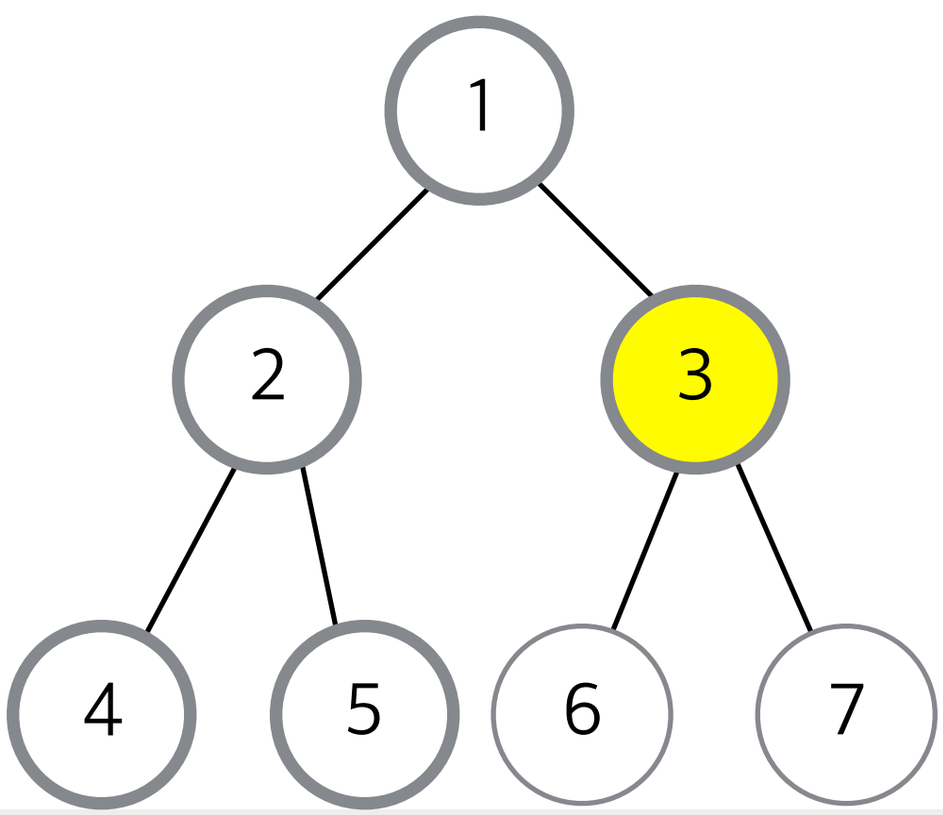
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



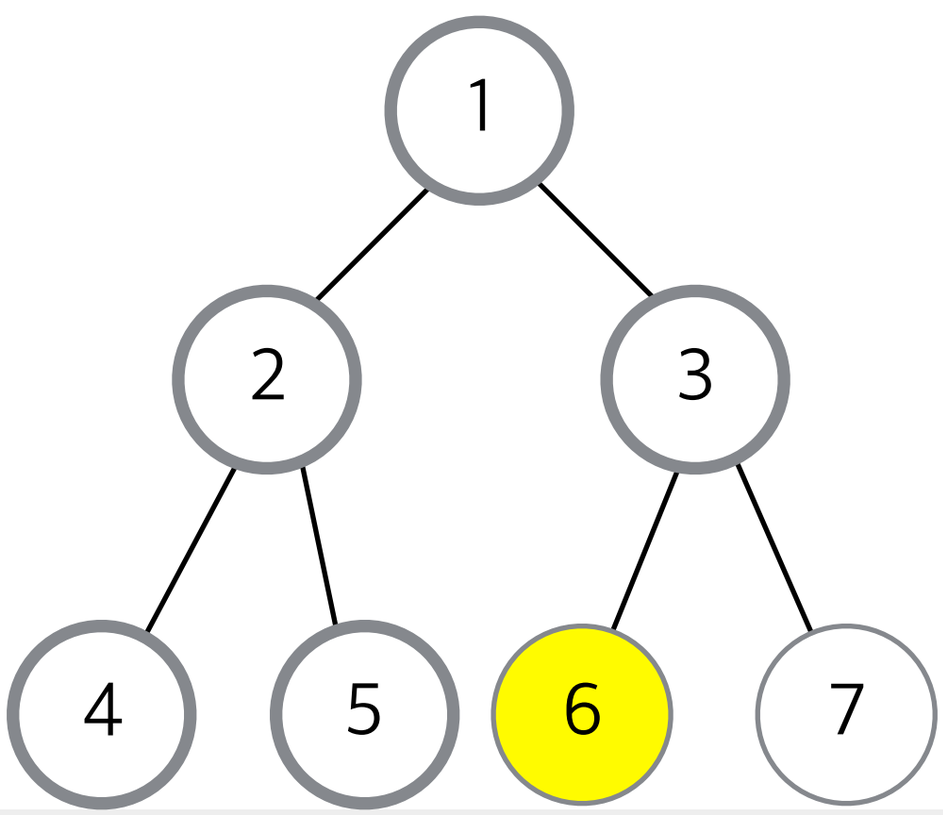
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



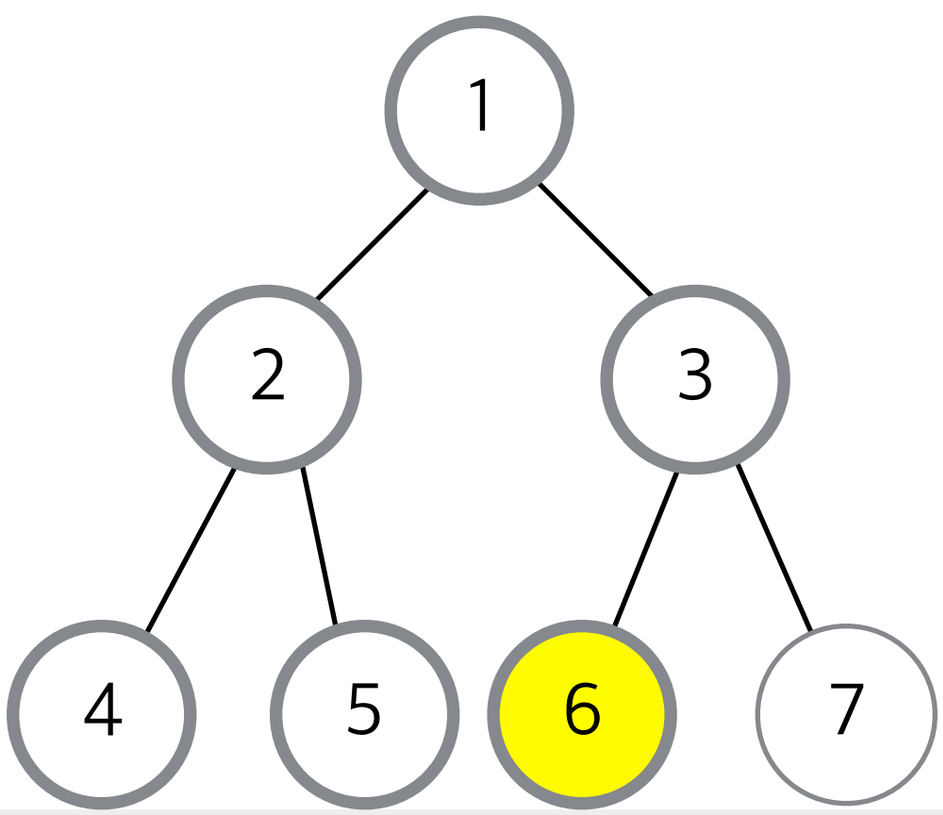
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



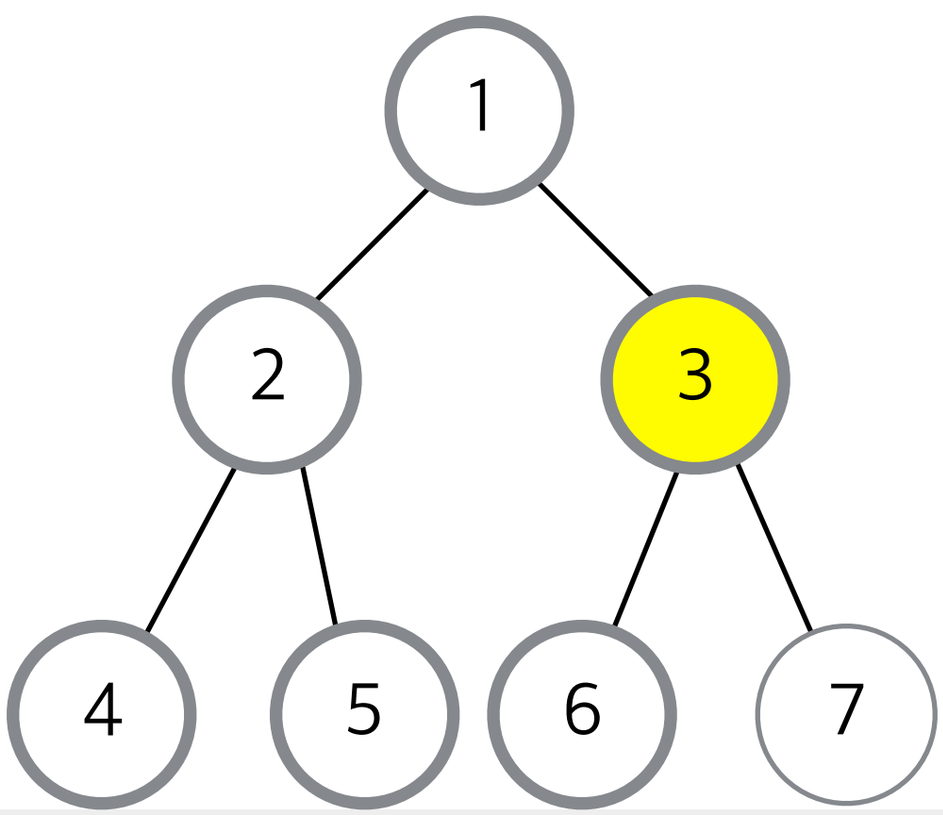
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



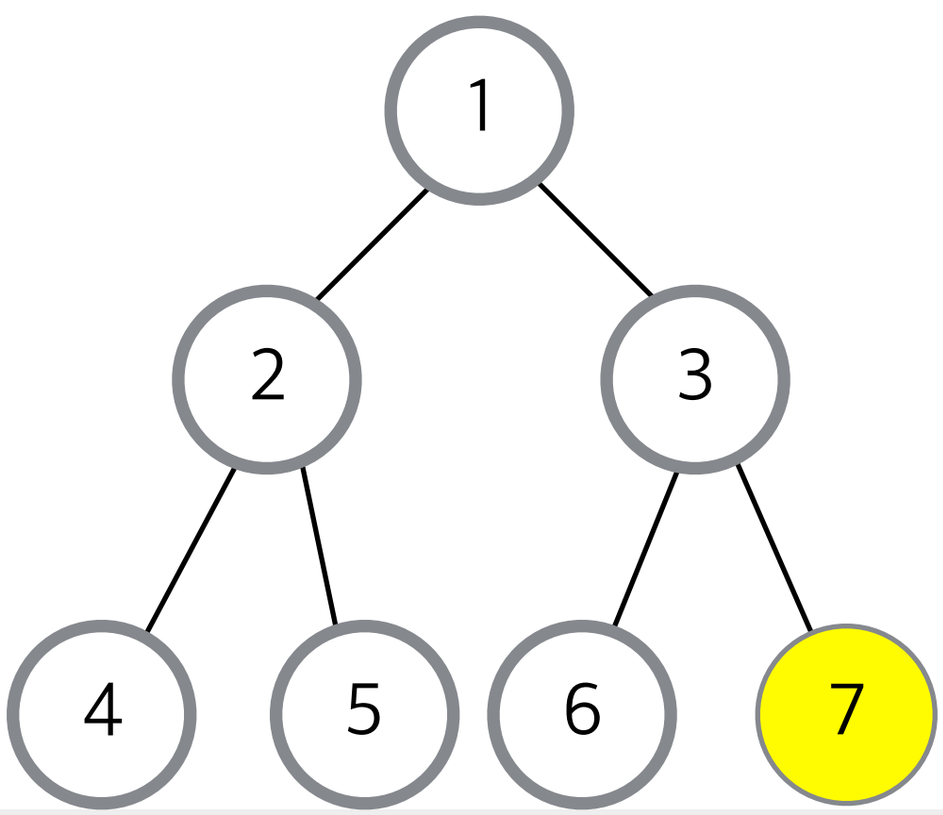
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



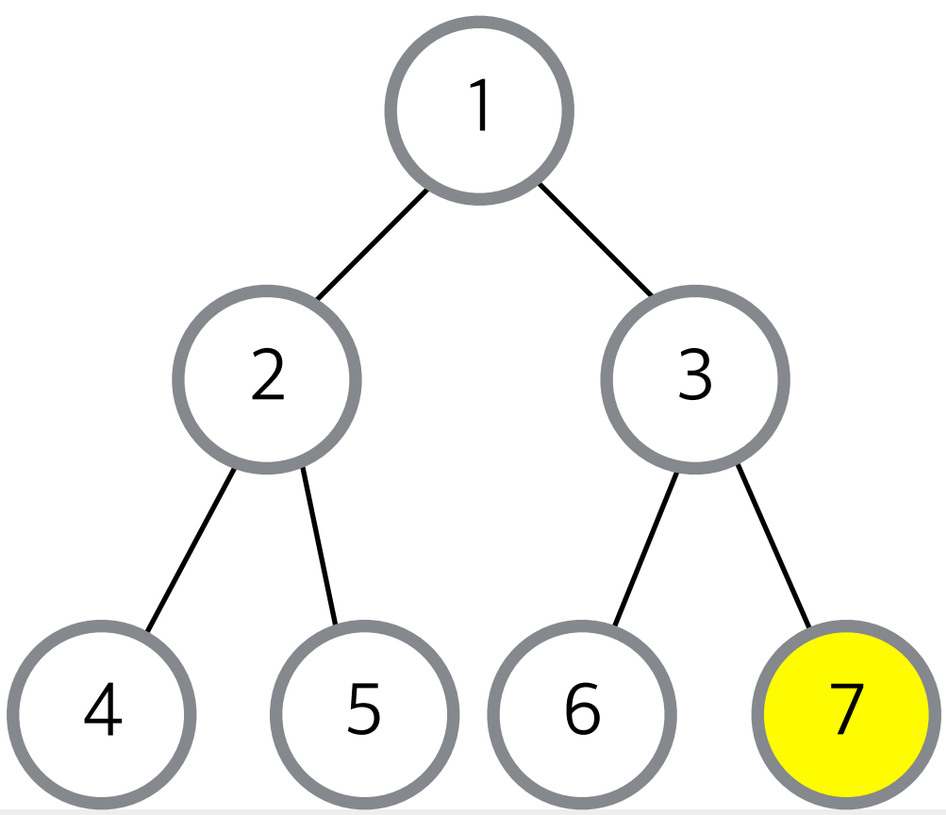
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



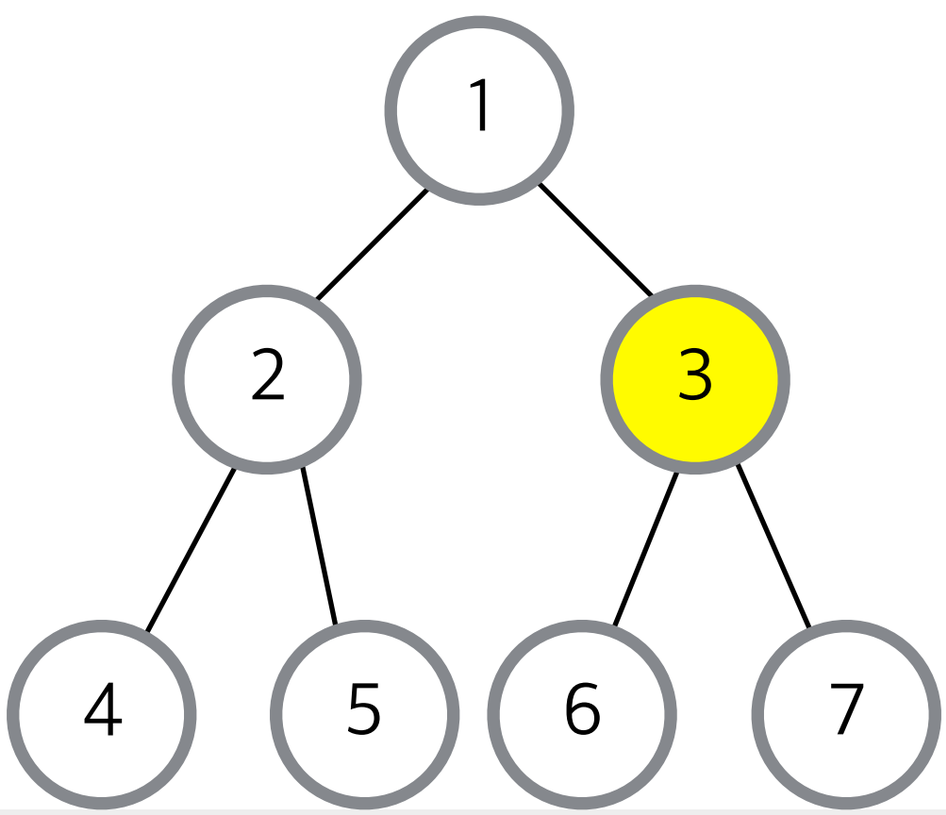
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



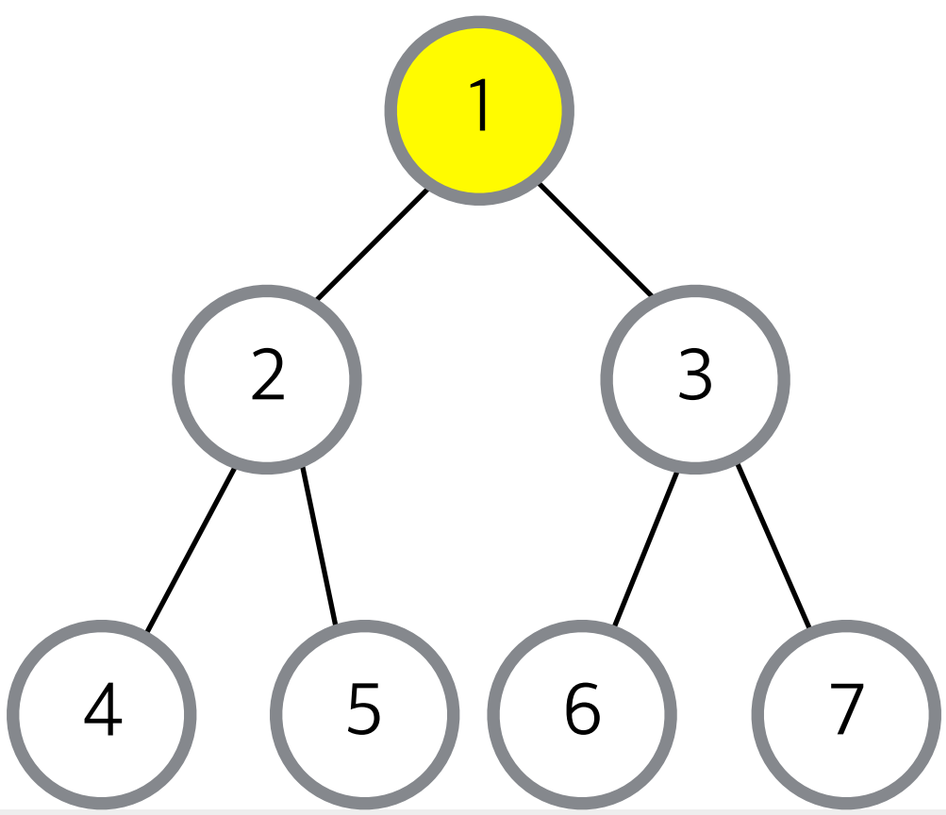
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



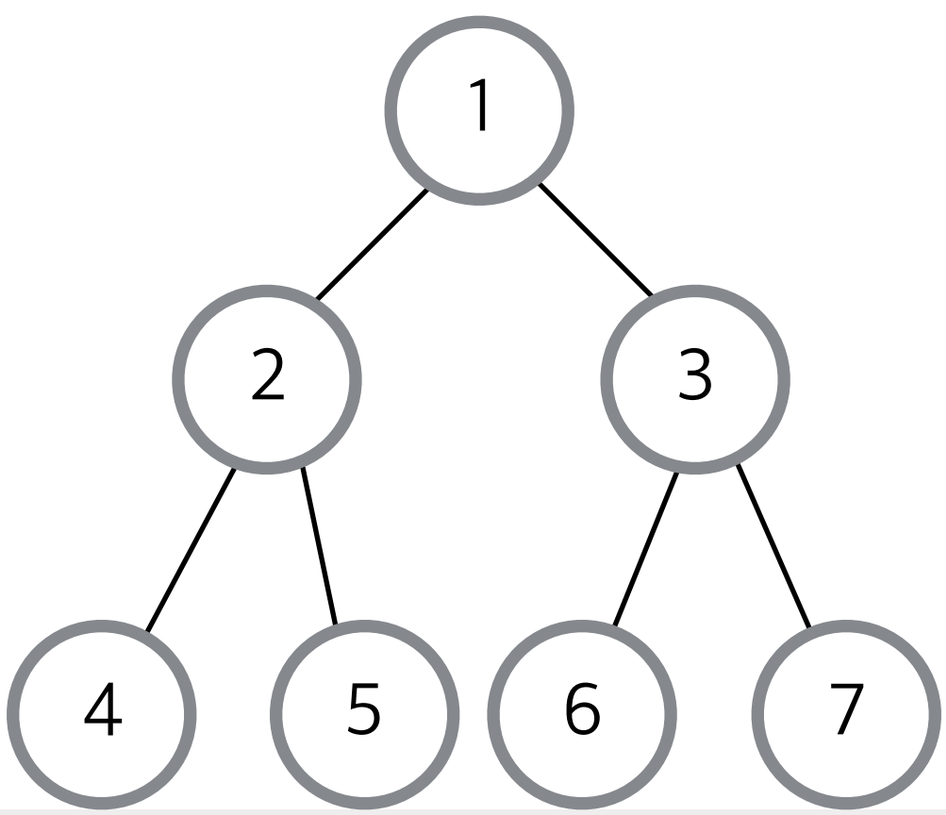
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R

중위순회 : L - Root - R

후위순회 : L - R - Root



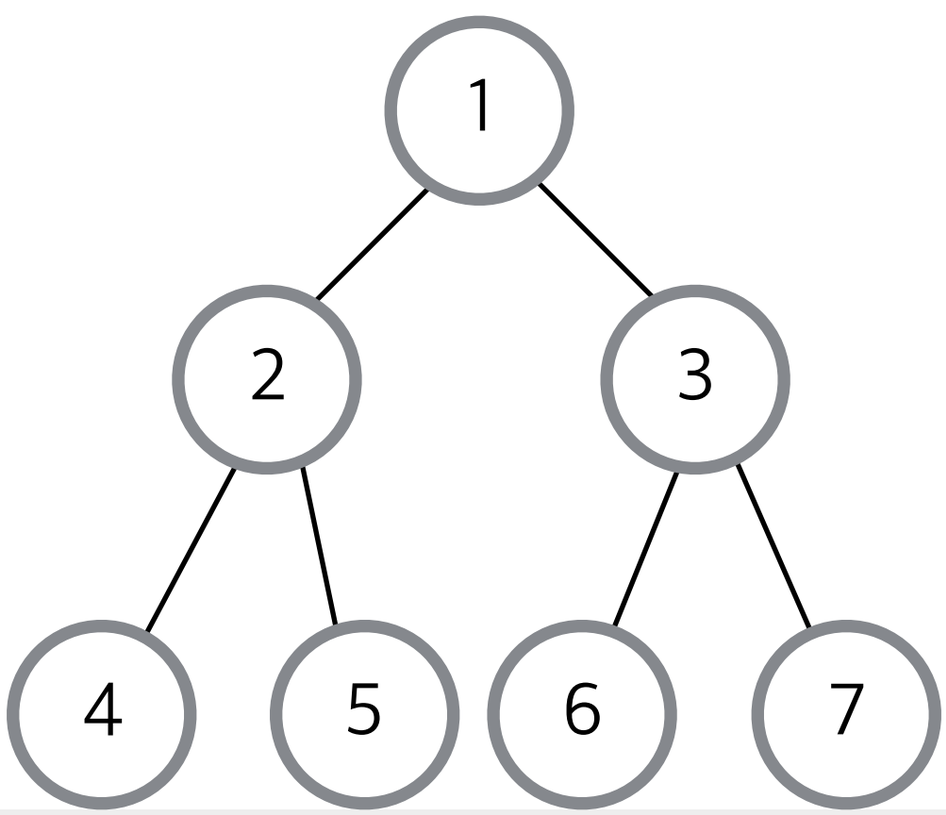
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R 1 2 4 5 3 6 7

중위순회 : L - Root - R

후위순회 : L - R - Root



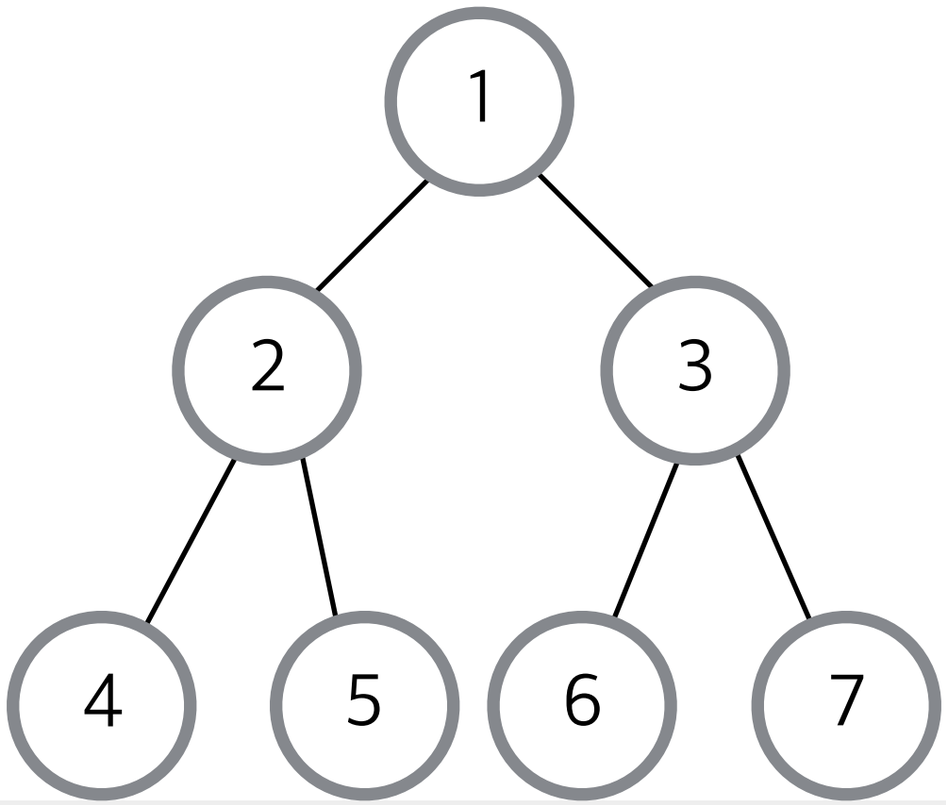
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R 1 2 4 5 3 6 7

중위순회 : L - Root - R

후위순회 : L - R - Root



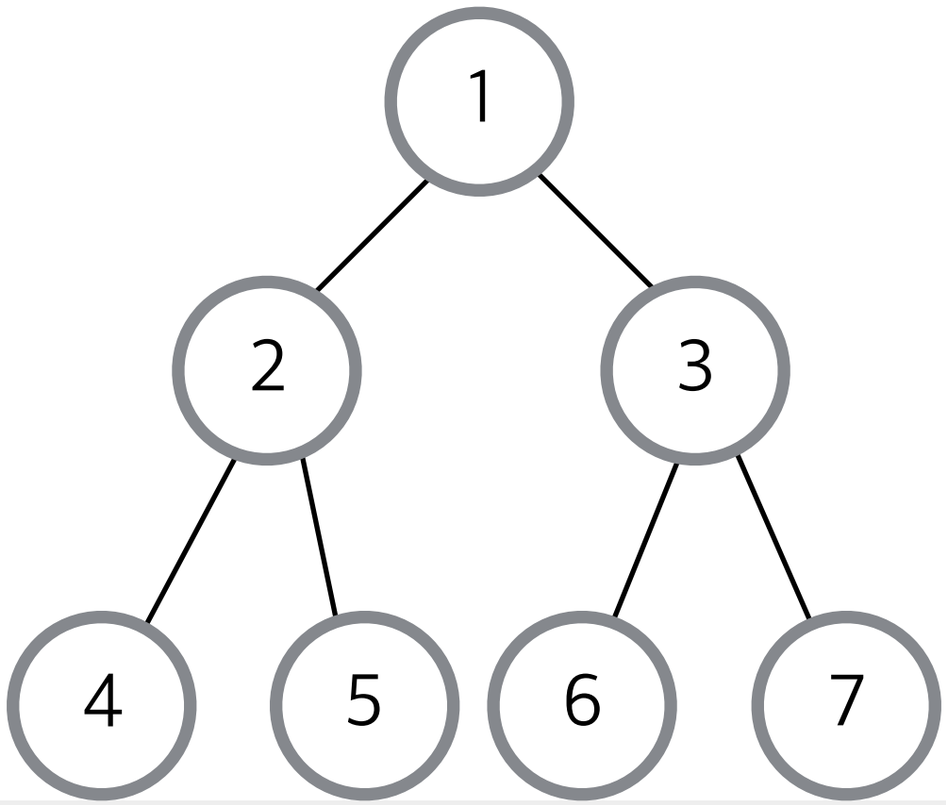
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R　　1 2 4 5 3 6 7

중위순회 : L - Root - R　　**4 2 5 1 6 3 7**

후위순회 : L - R - Root



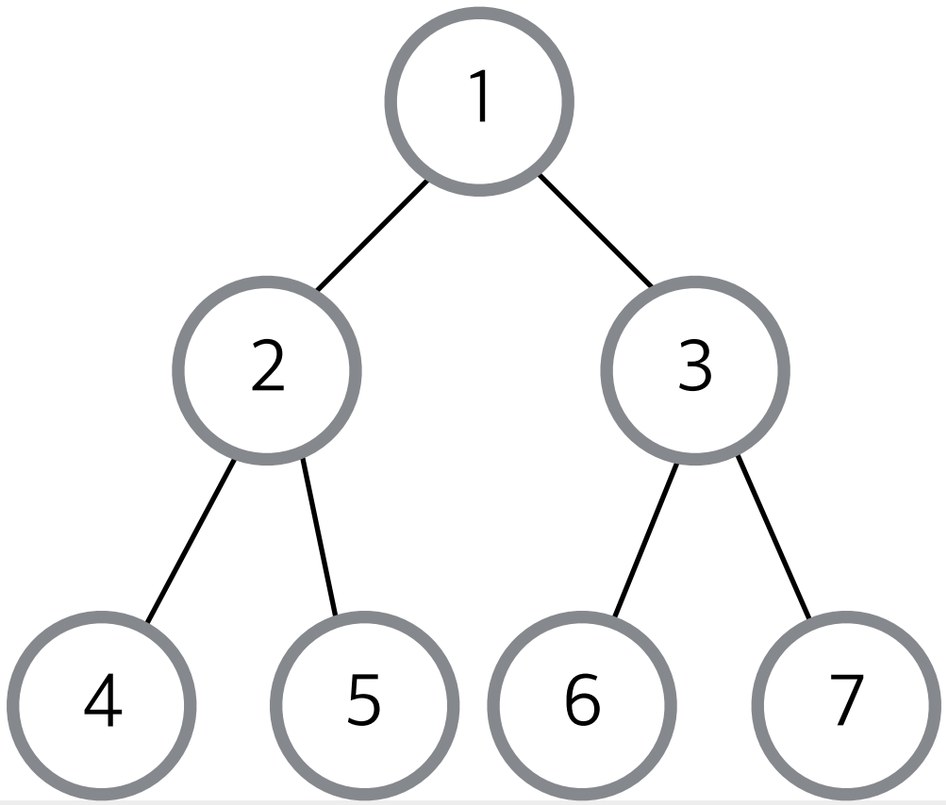
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R　　1 2 4 5 3 6 7

중위순회 : L - Root - R　　4 2 5 1 6 3 7

후위순회 : L - R - Root



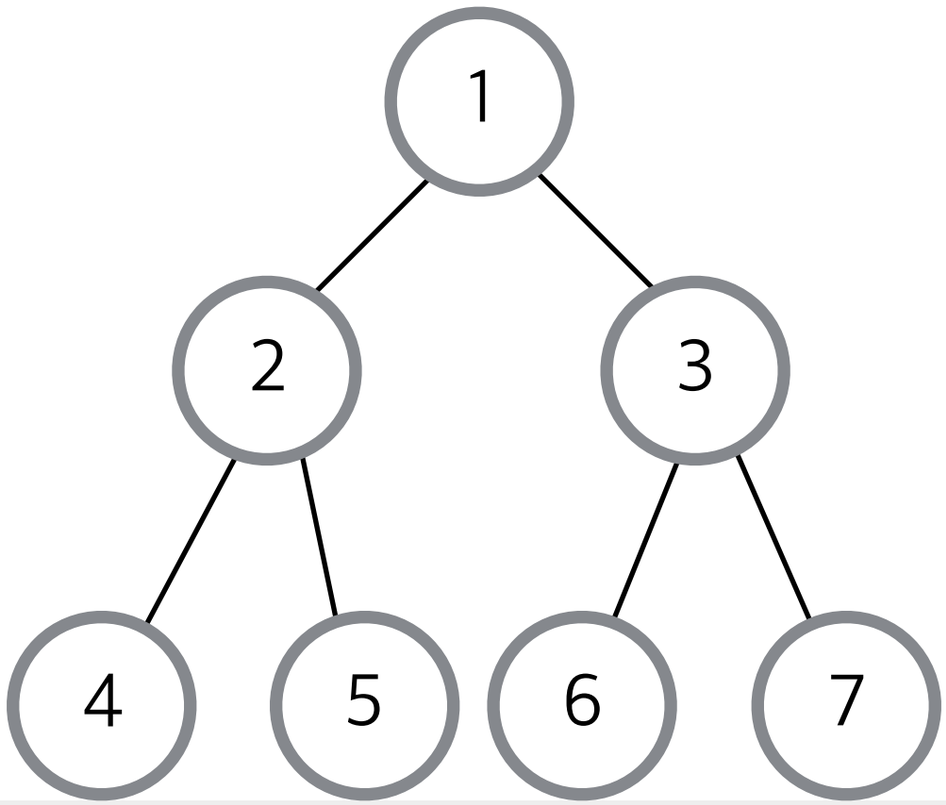
트리 순회

트리 내에 어떠한 자료가 담겨있는지를 알기 위함

전위순회 : Root - L - R　　1 2 4 5 3 6 7

중위순회 : L - Root - R　　4 2 5 1 6 3 7

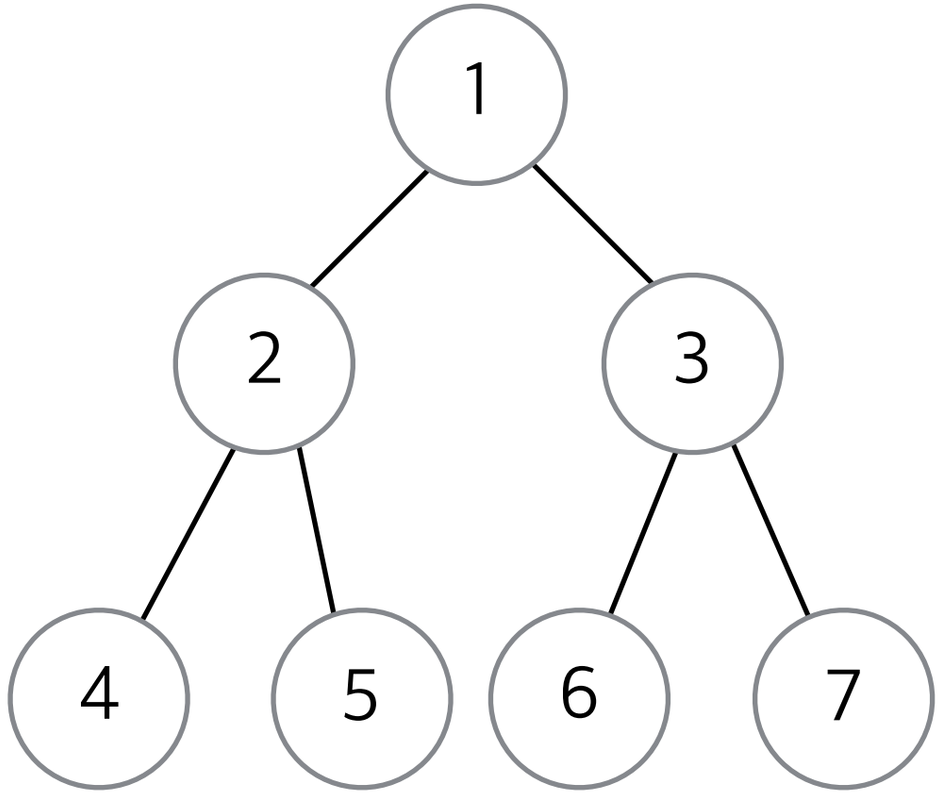
후위순회 : L - R - Root　　**4 5 2 6 7 3 1**



도대체 왜 하필 트리여야만 하는가 ?

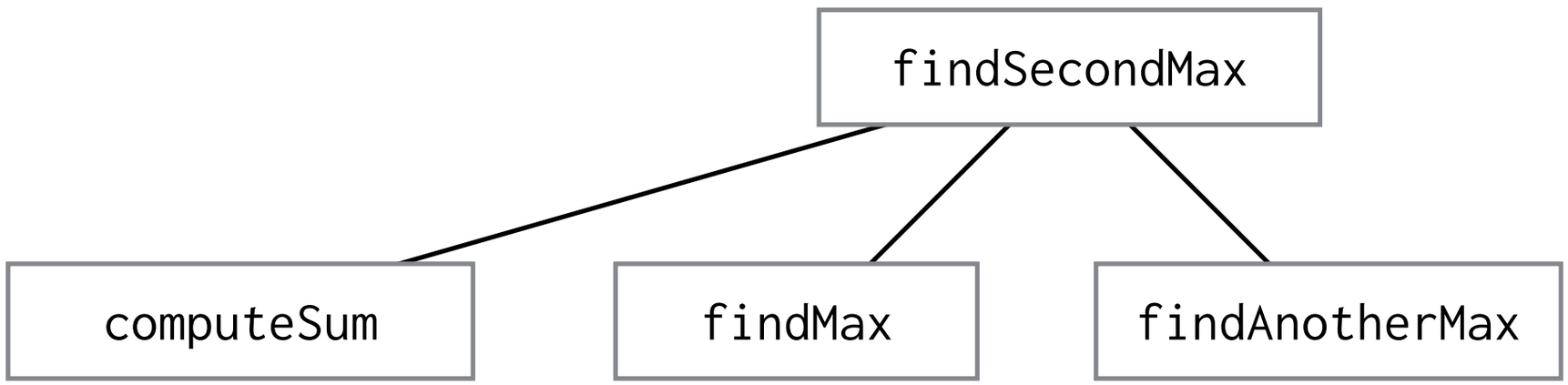
1) 정점에 무슨 자료를 담는가 ? 코드가 실행되는 상태

2) 간선은 어떤 의미인가 ? 코드 A가 코드 B를 부른다



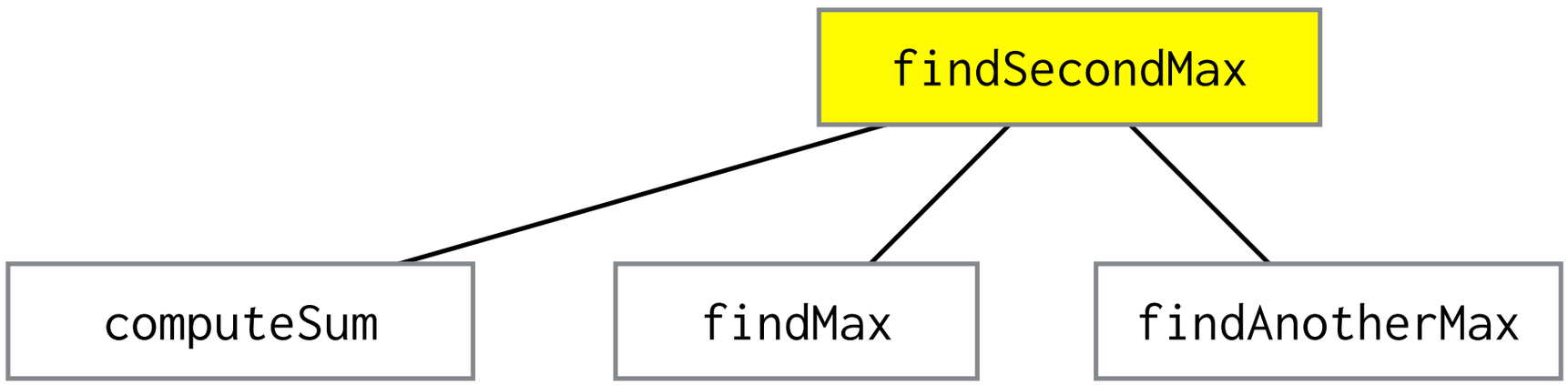
도대체 왜 하필 트리여야만 하는가?

```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```



도대체 왜 하필 트리여야만 하는가?

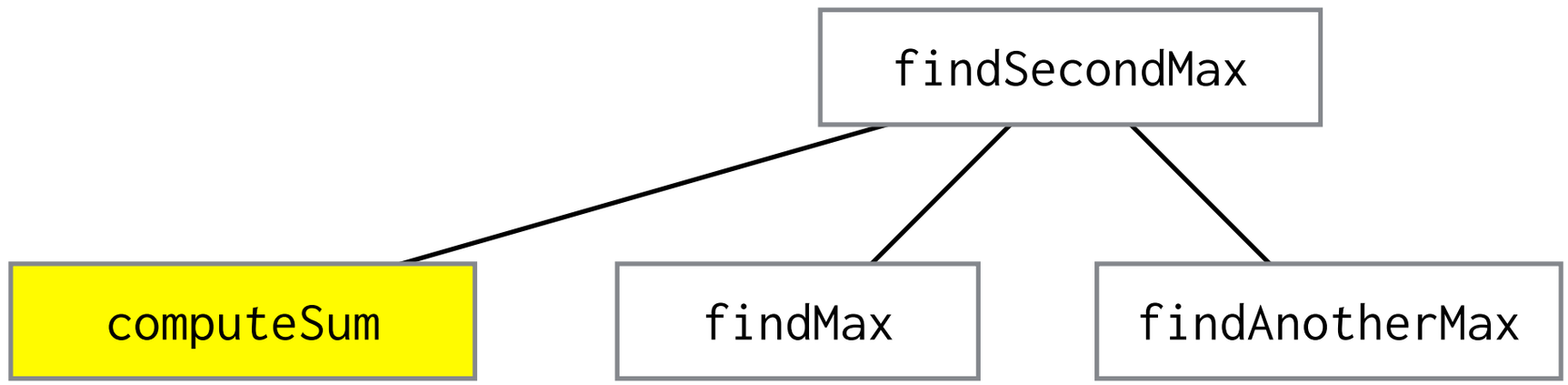
```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```



도대체 왜 하필 트리여야만 하는가?

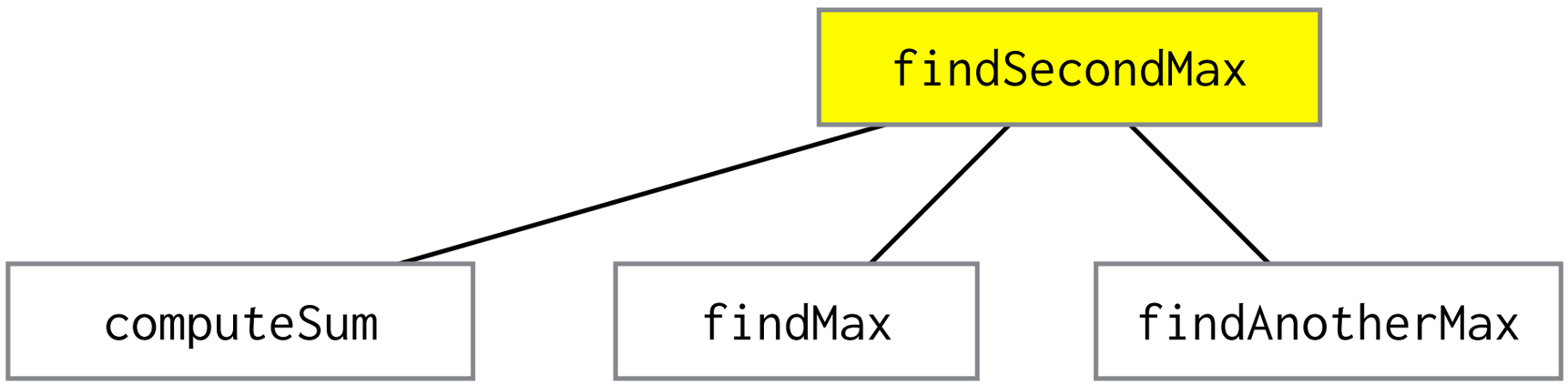
```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```

```
def computeSum(myMatrix) :
    return sum(myMatrix)
```



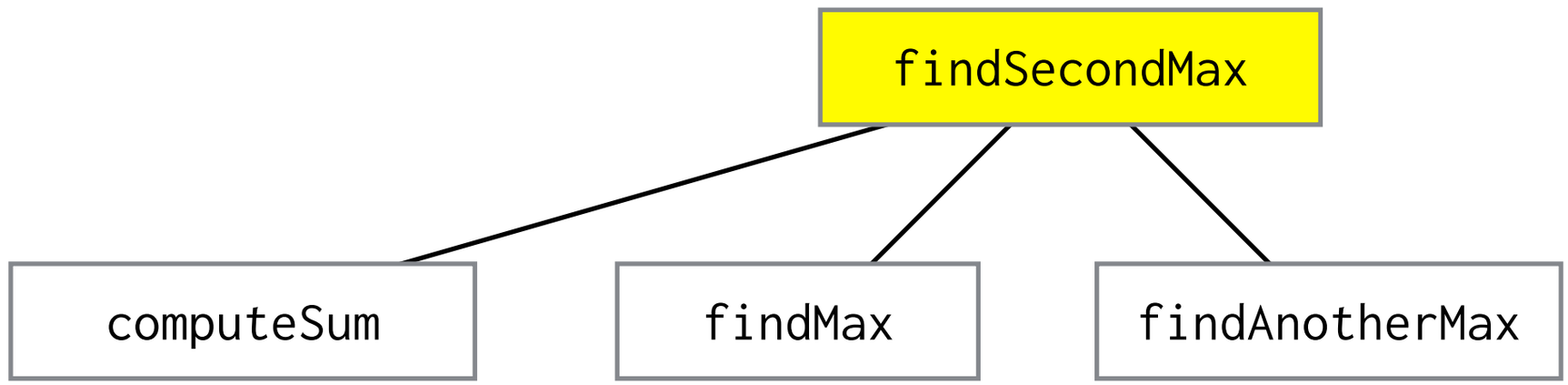
도대체 왜 하필 트리여야만 하는가?

```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```



도대체 왜 하필 트리여야만 하는가?

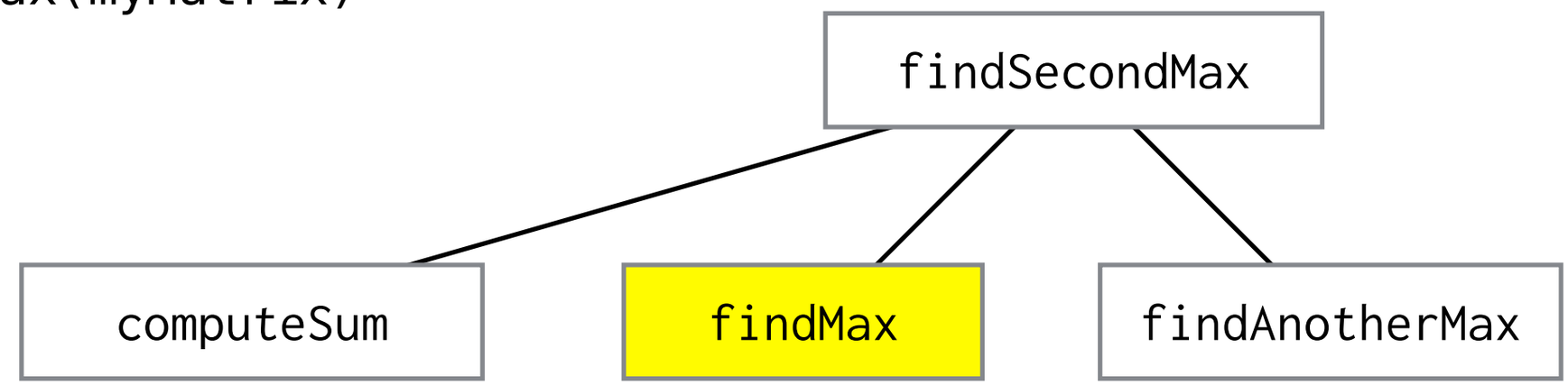
```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```



도대체 왜 하필 트리여야만 하는가?

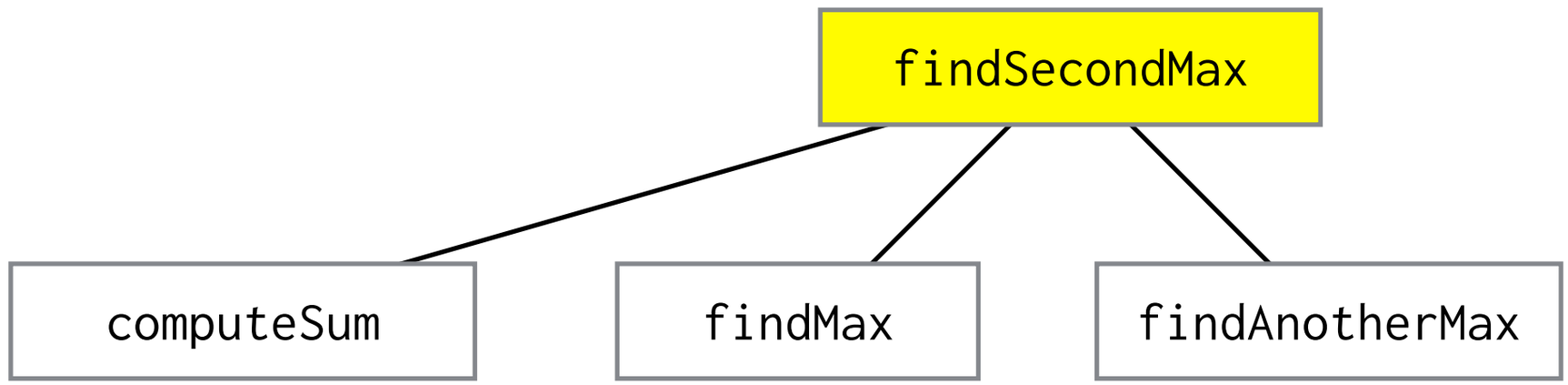
```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```

```
def findMax(myMatrix) :
    return max(myMatrix)
```



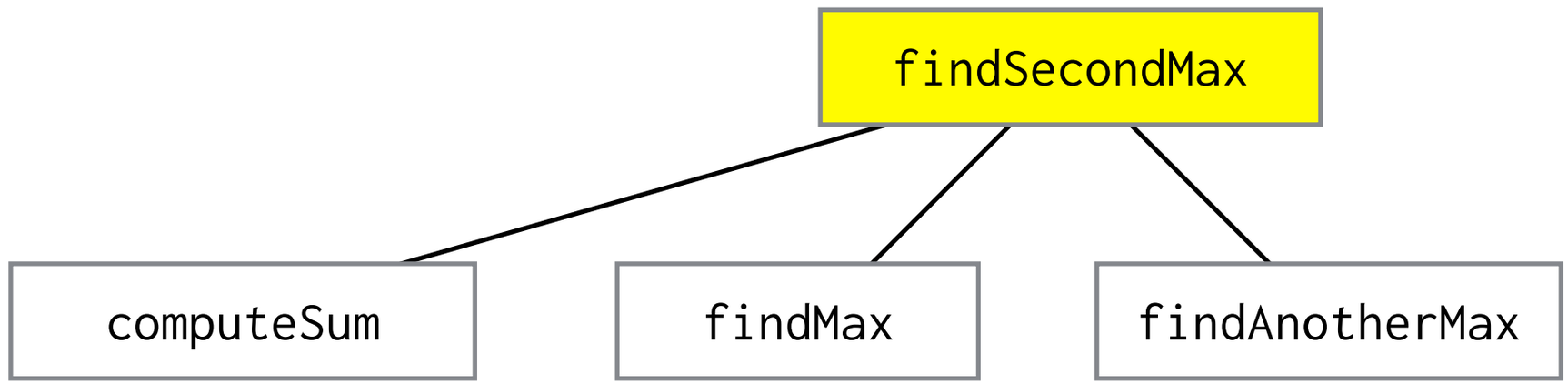
도대체 왜 하필 트리여야만 하는가?

```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```



도대체 왜 하필 트리여야만 하는가 ?

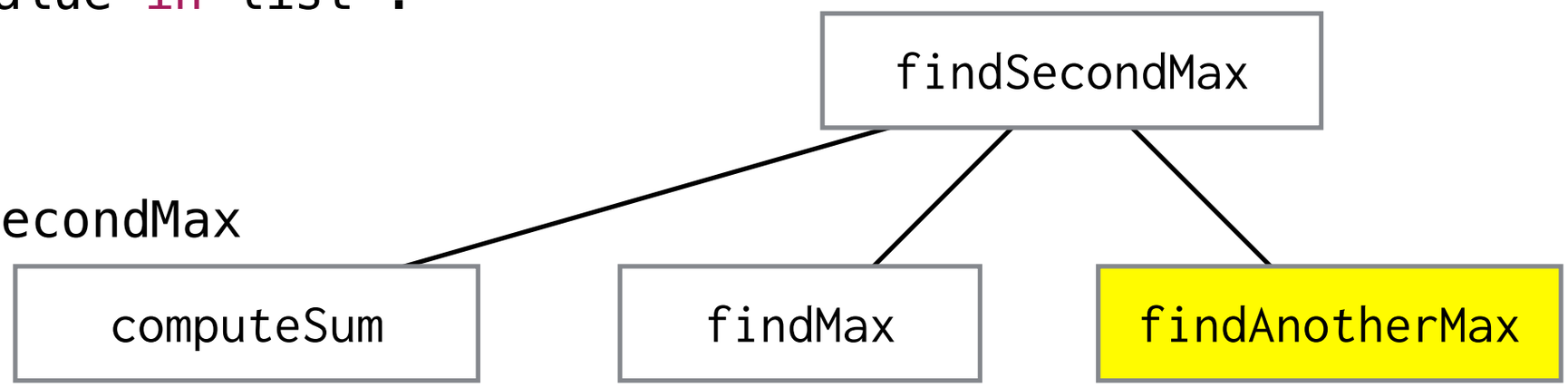
```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```



도대체 왜 하필 트리여야만 하는가?

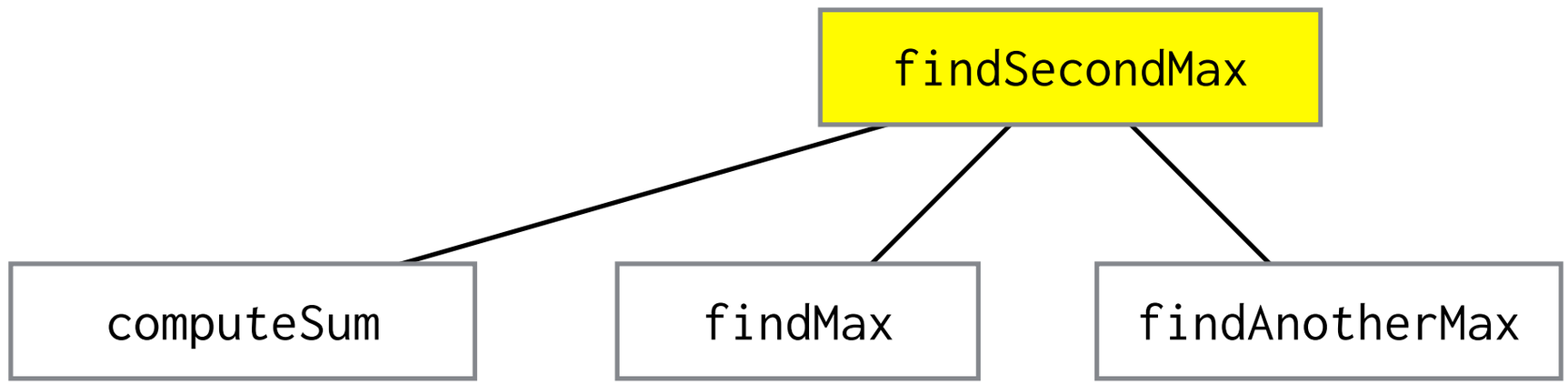
```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```

```
def findAnotherMax (myMatrix, value) :
   secondMax = -1

   for list in myMatrix :
      for value in list :

   ...

   return secondMax
```



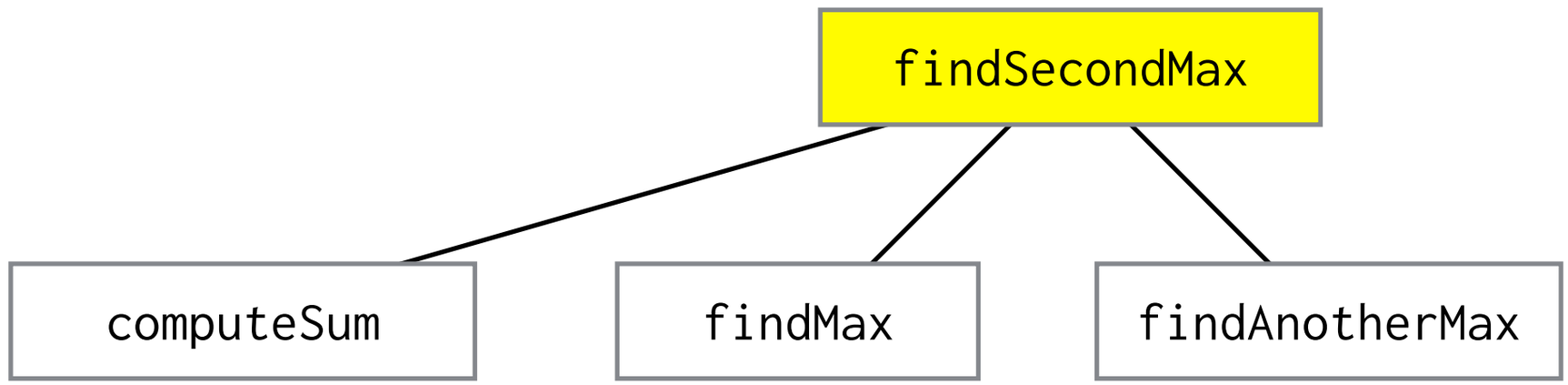
도대체 왜 하필 트리여야만 하는가?

```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```



도대체 왜 하필 트리여야만 하는가?

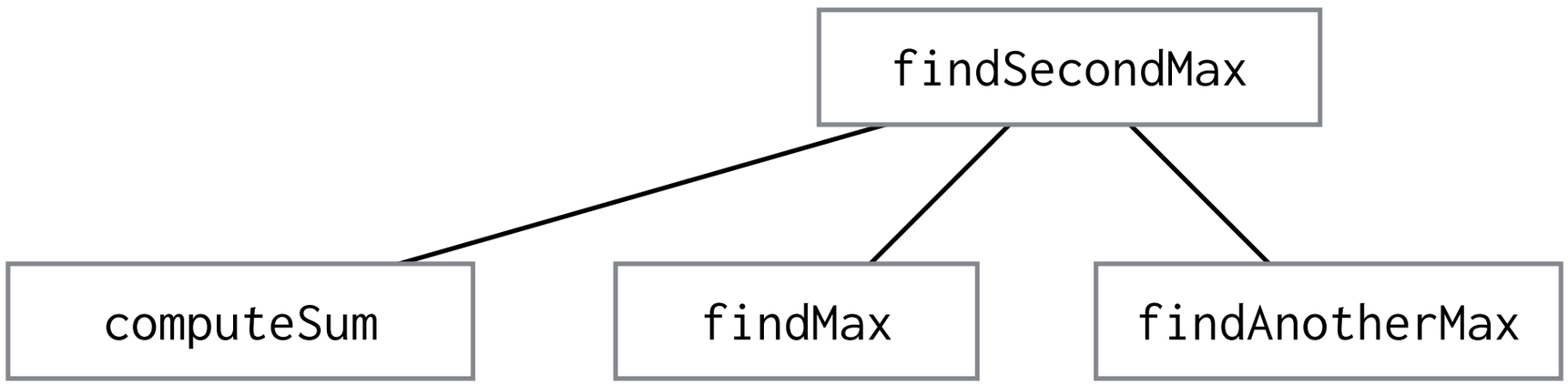
```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```



도대체 왜 하필 트리여야만 하는가?

```
def findSecondMax(myMatrix) :
    sum = computeSum(myMatrix)

    maxValue = findMax(myMatrix)
    secondMaxValue = findAnotherMax(myMatrix, maxValue)

    return (sum, maxValue, secondMaxValue)
```

코드 실행 = 트리를 후위순회



요약

의미단위로 작성된 코드가 좋은 코드이다

→ 코드를 이해한다 = 각 함수가 무슨 일을 하는지 설명할 수 있다

트리는 코드가 실행되고 있는 상태를 나타내는 자료구조이다

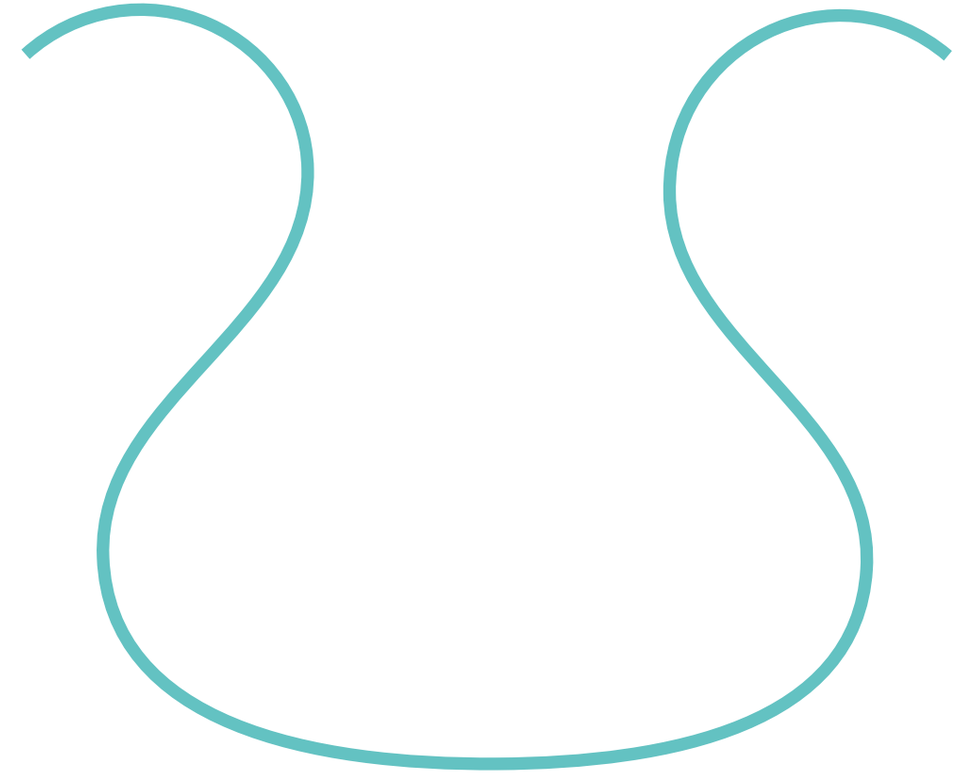
→ 물론, 코드를 의미 단위로 나타냈을 때 파악이 가능한 사실이다

코드를 하나하나 따라가는 것은 컴퓨터가 해야 할 일이다

→ 우리는 앞으로 코드가 하는 일, 더 나아가 코드의 의미에 집중한다

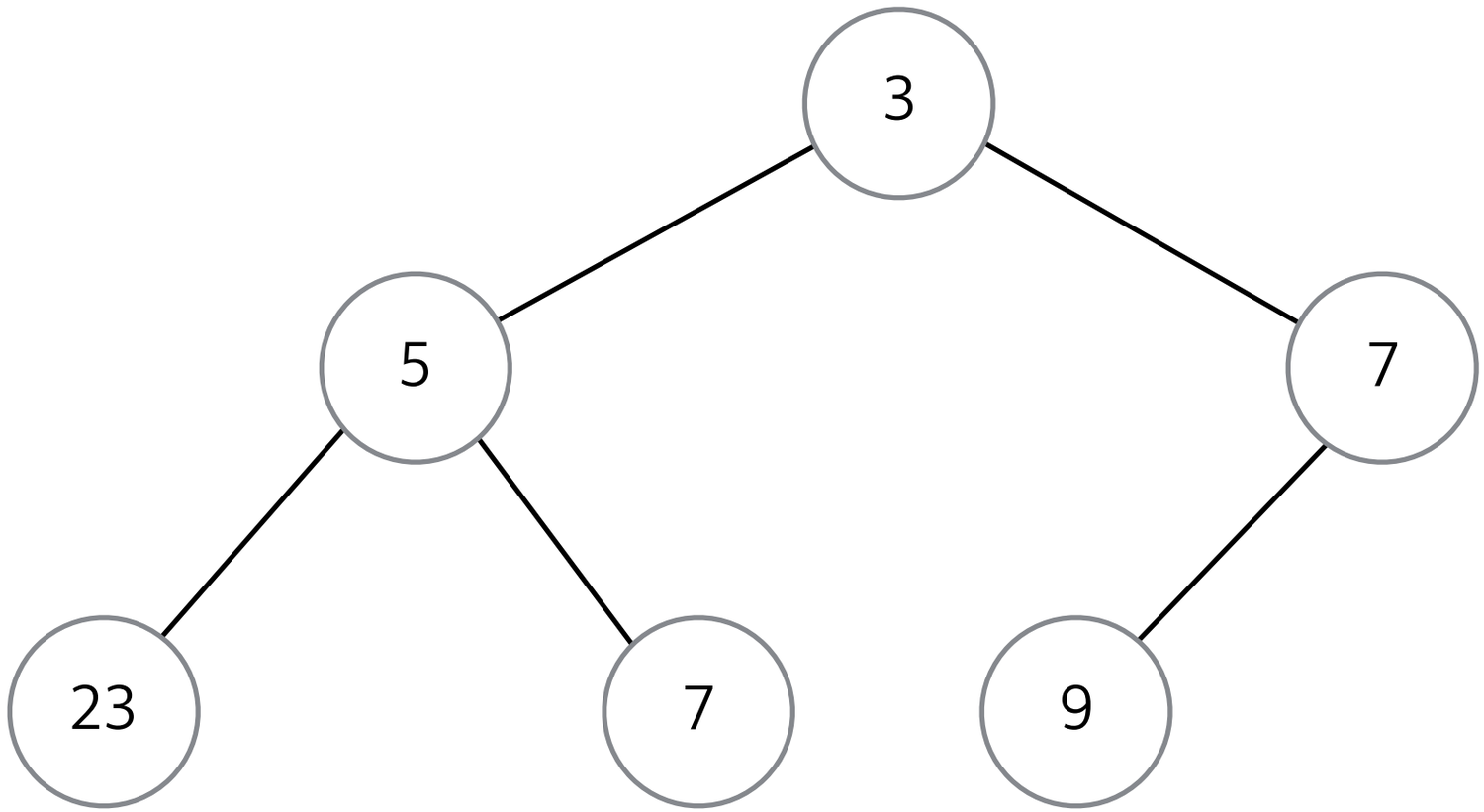
우선순위 큐

원소를 제거할 시, 가장 우선순위가 높은 원소를 제거



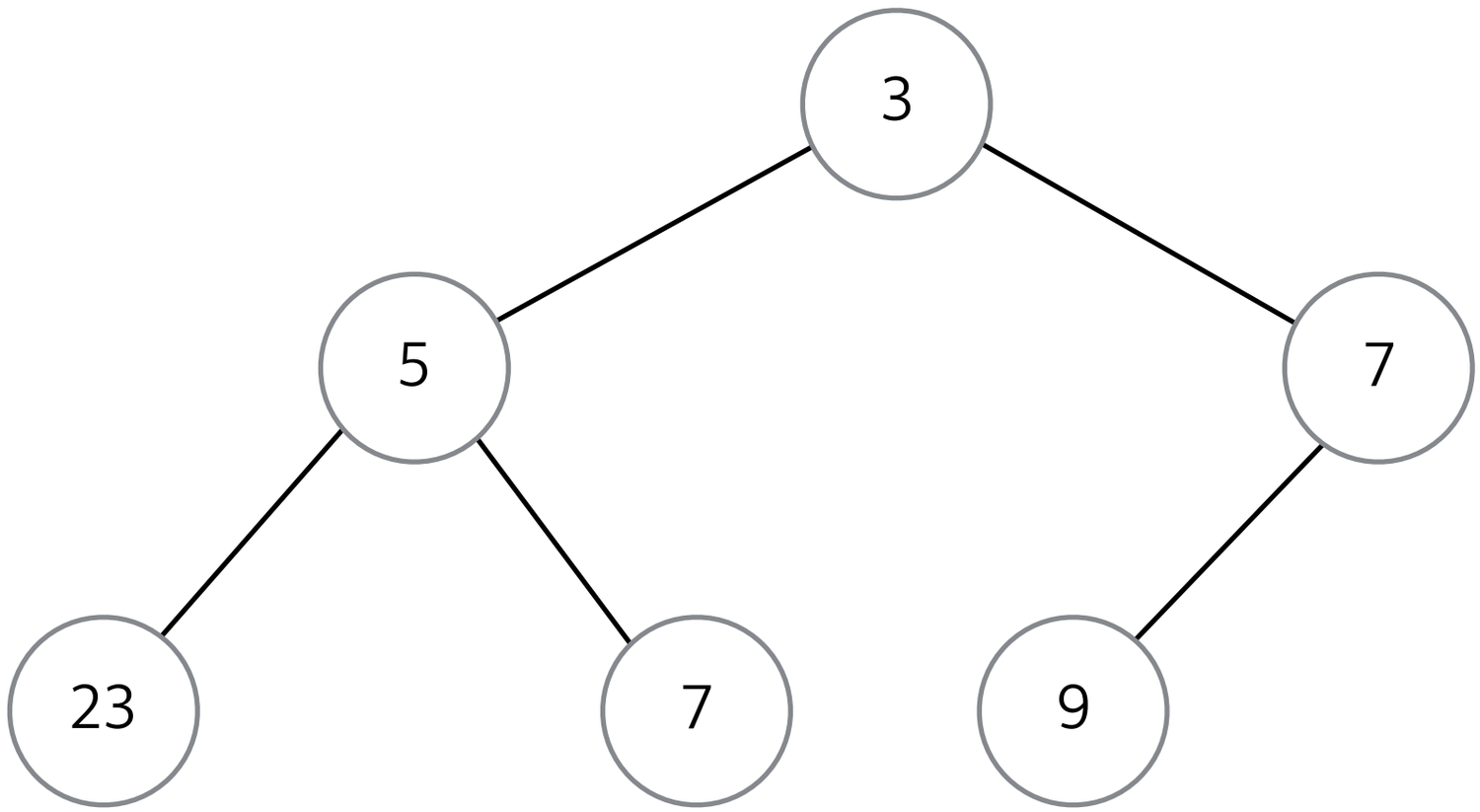
힙

부모의 값이 항상 자식보다 작은 완전 이진 트리



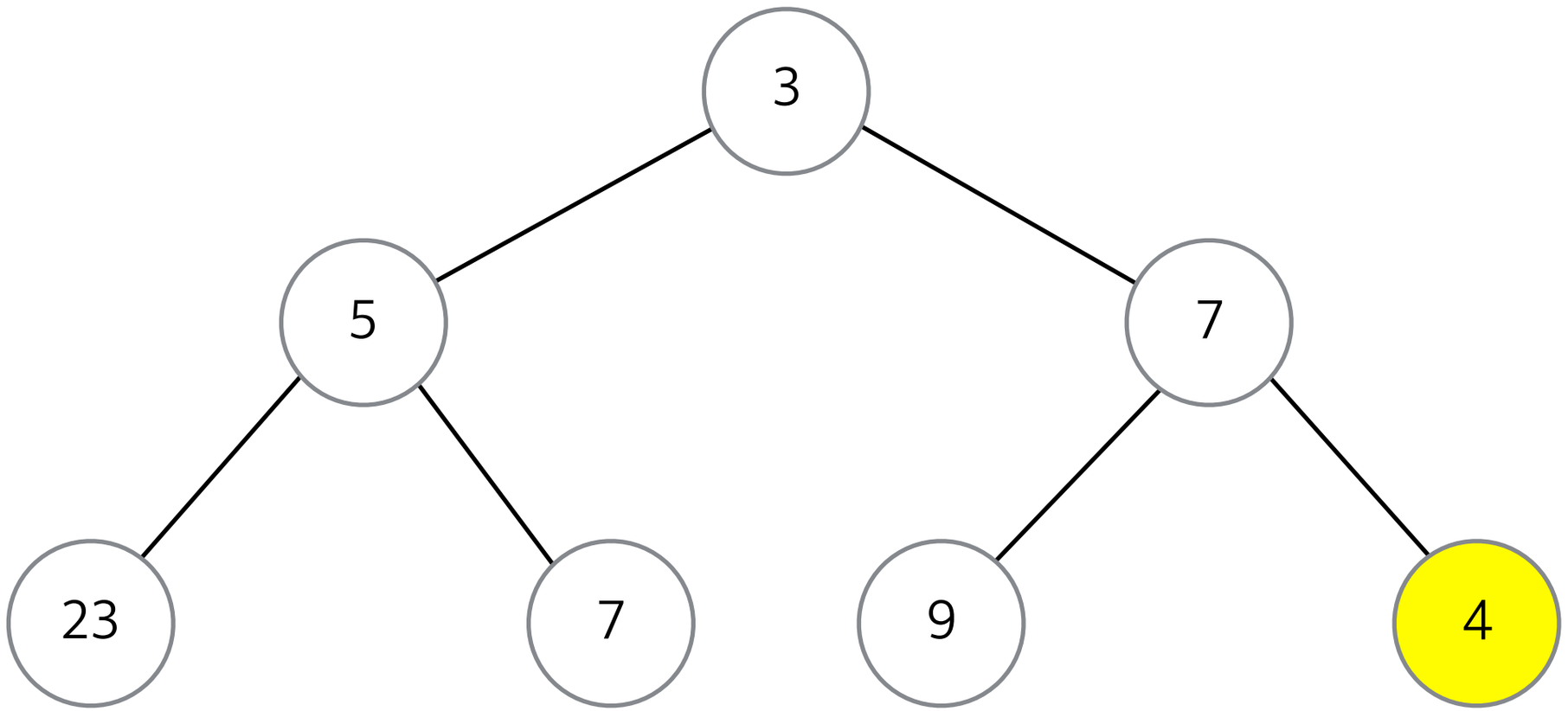
힙 : 값 삽입

heap.insert(4)



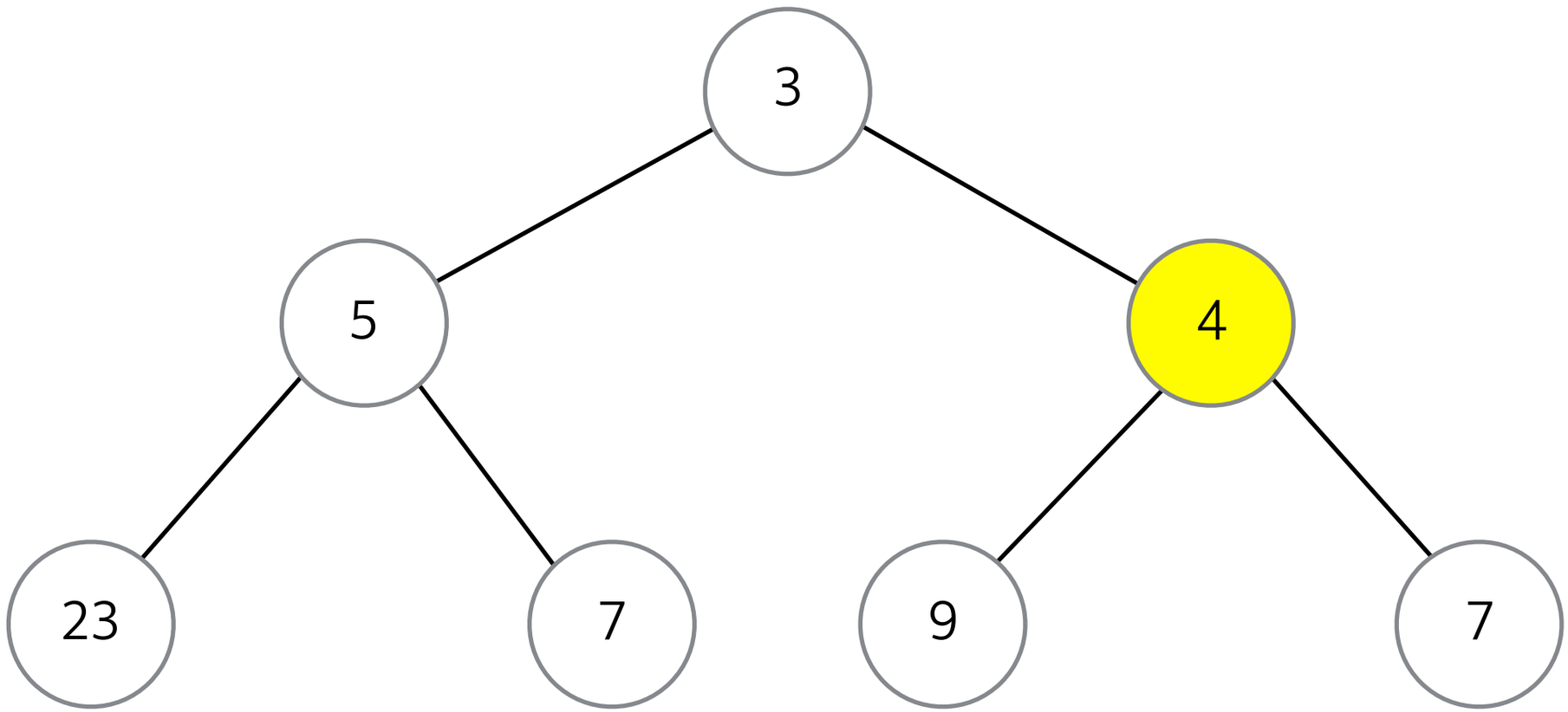
힙 : 값 삽입

heap.insert(4)



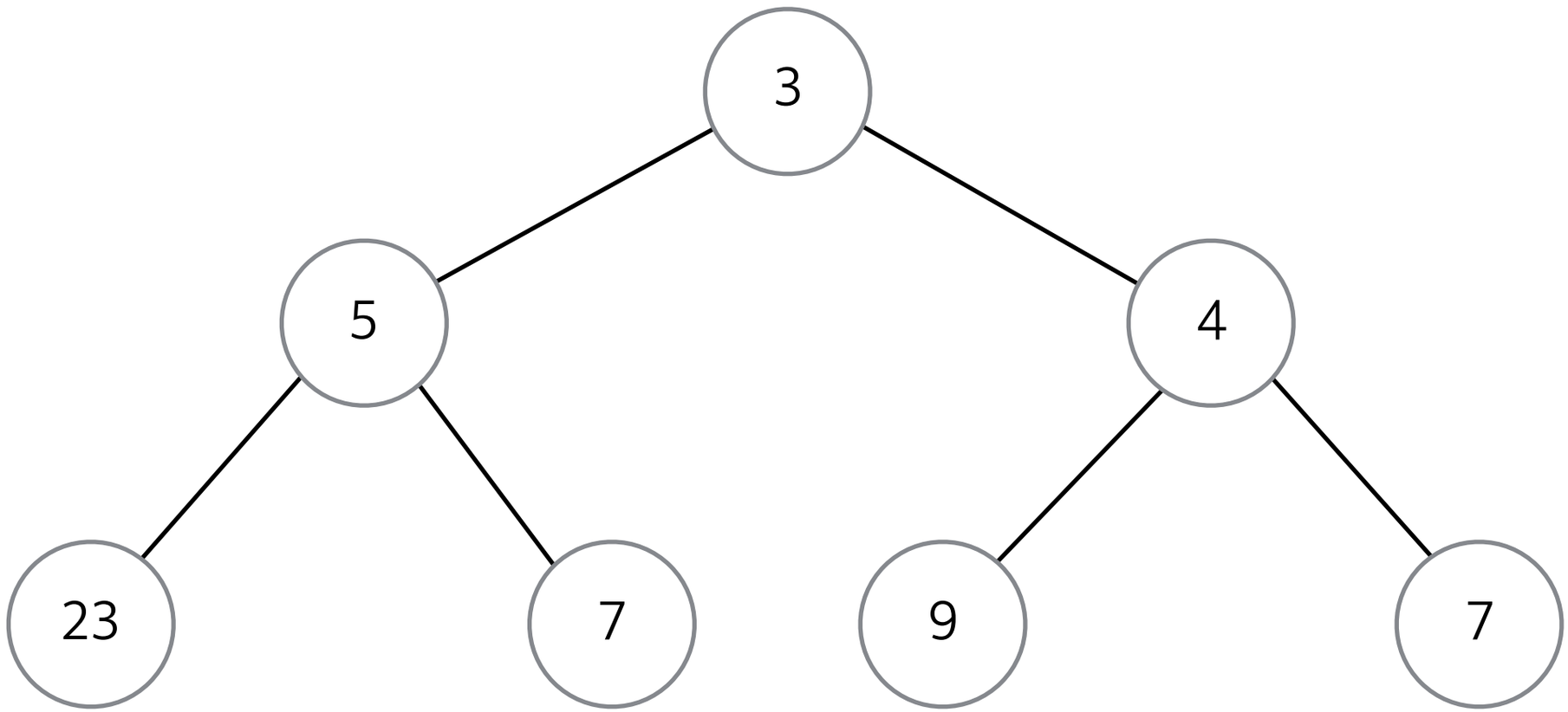
힙 : 값 삽입

heap.insert(4)



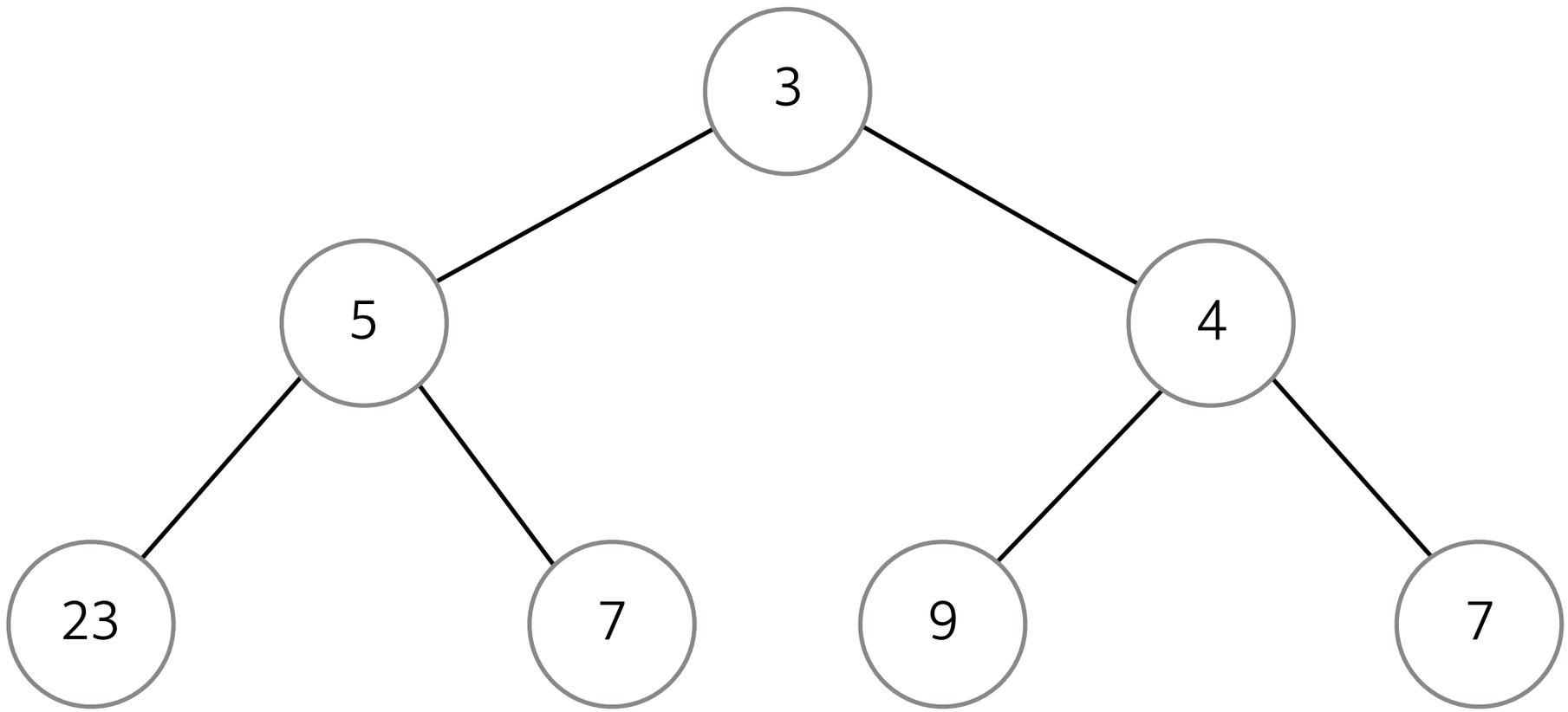
힙 : 값 삽입

heap.insert(4)



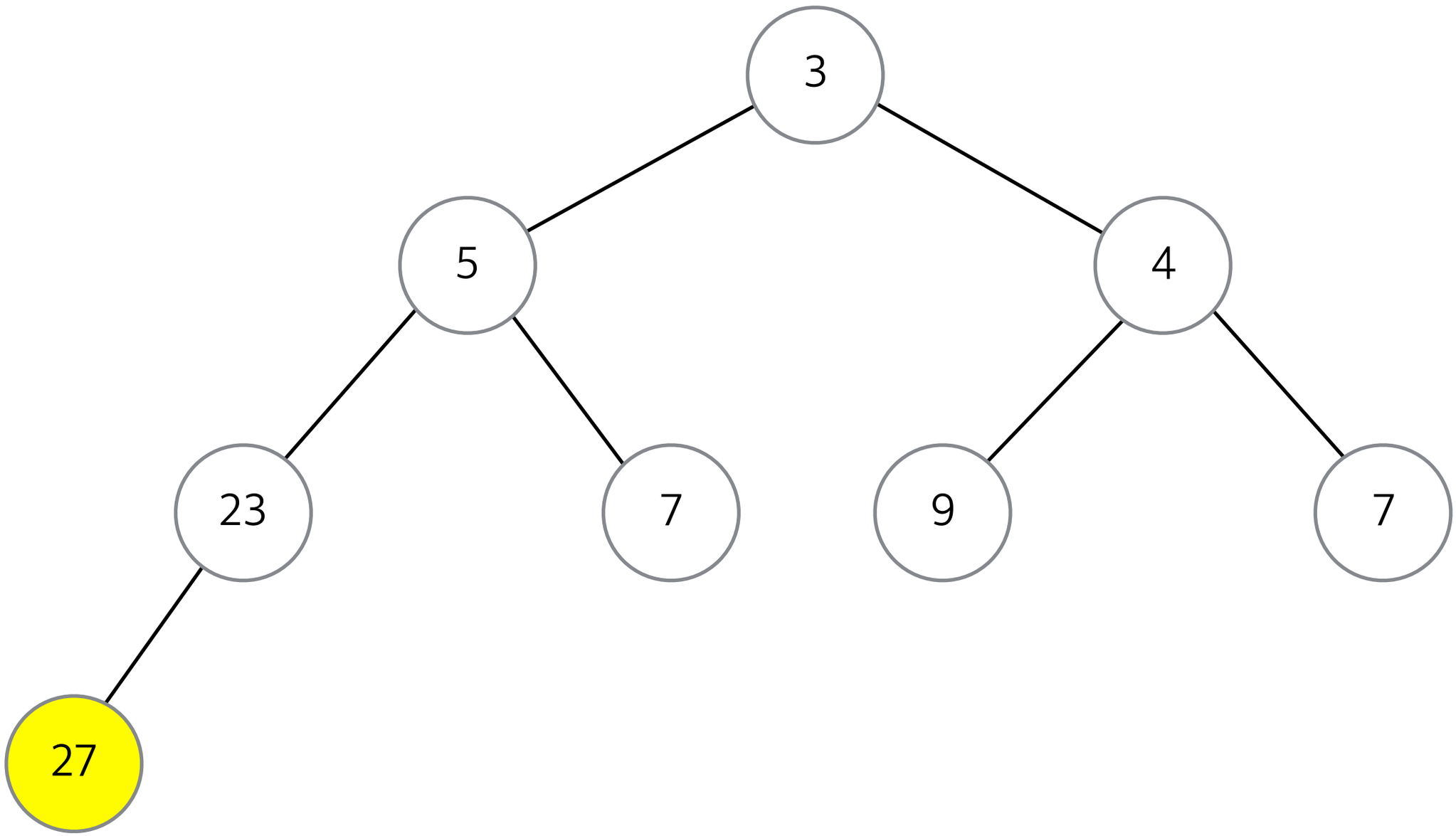
힙 : 값 삽입

heap.insert(27)



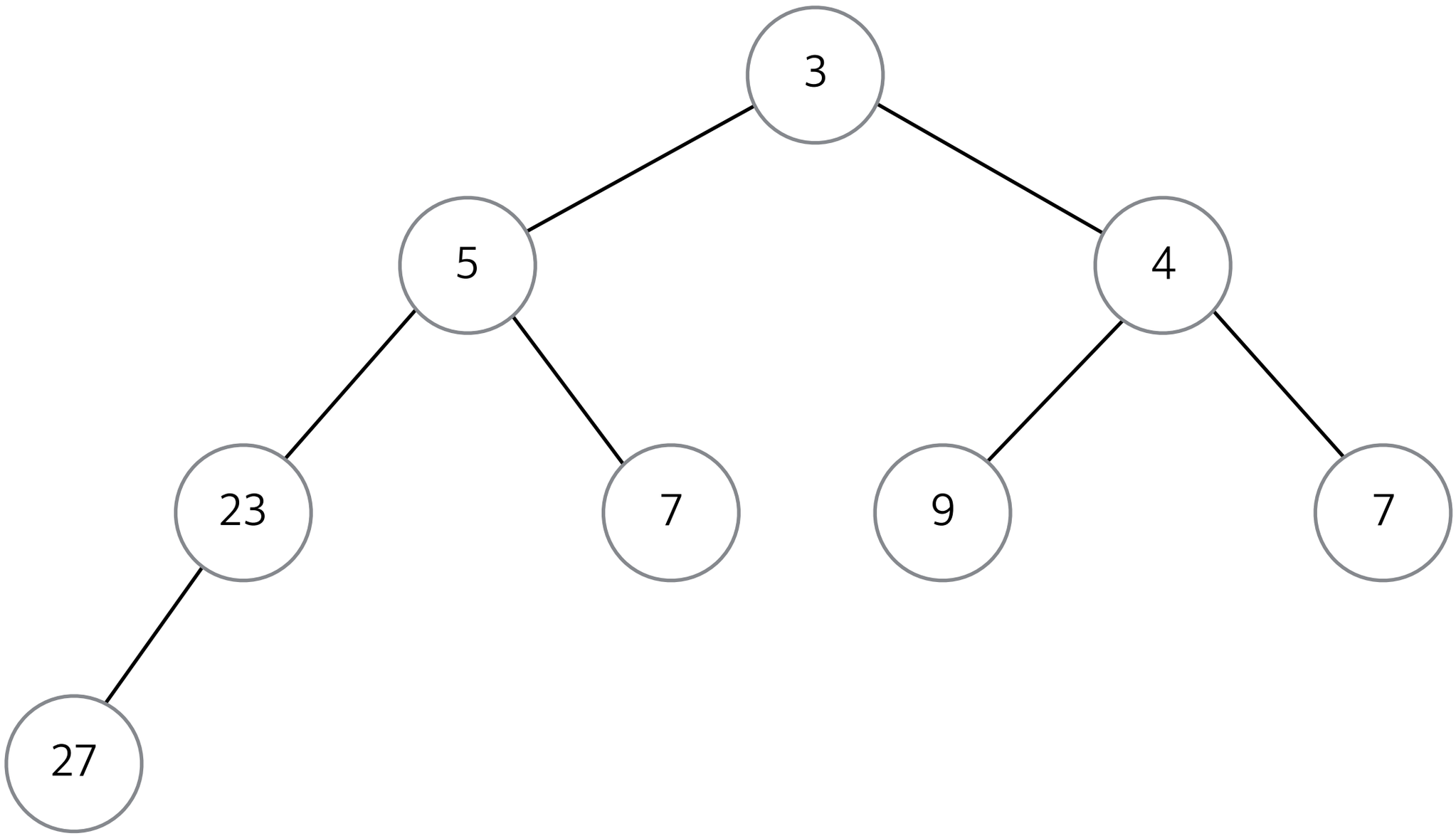
힙 : 값 삽입

heap.insert(27)



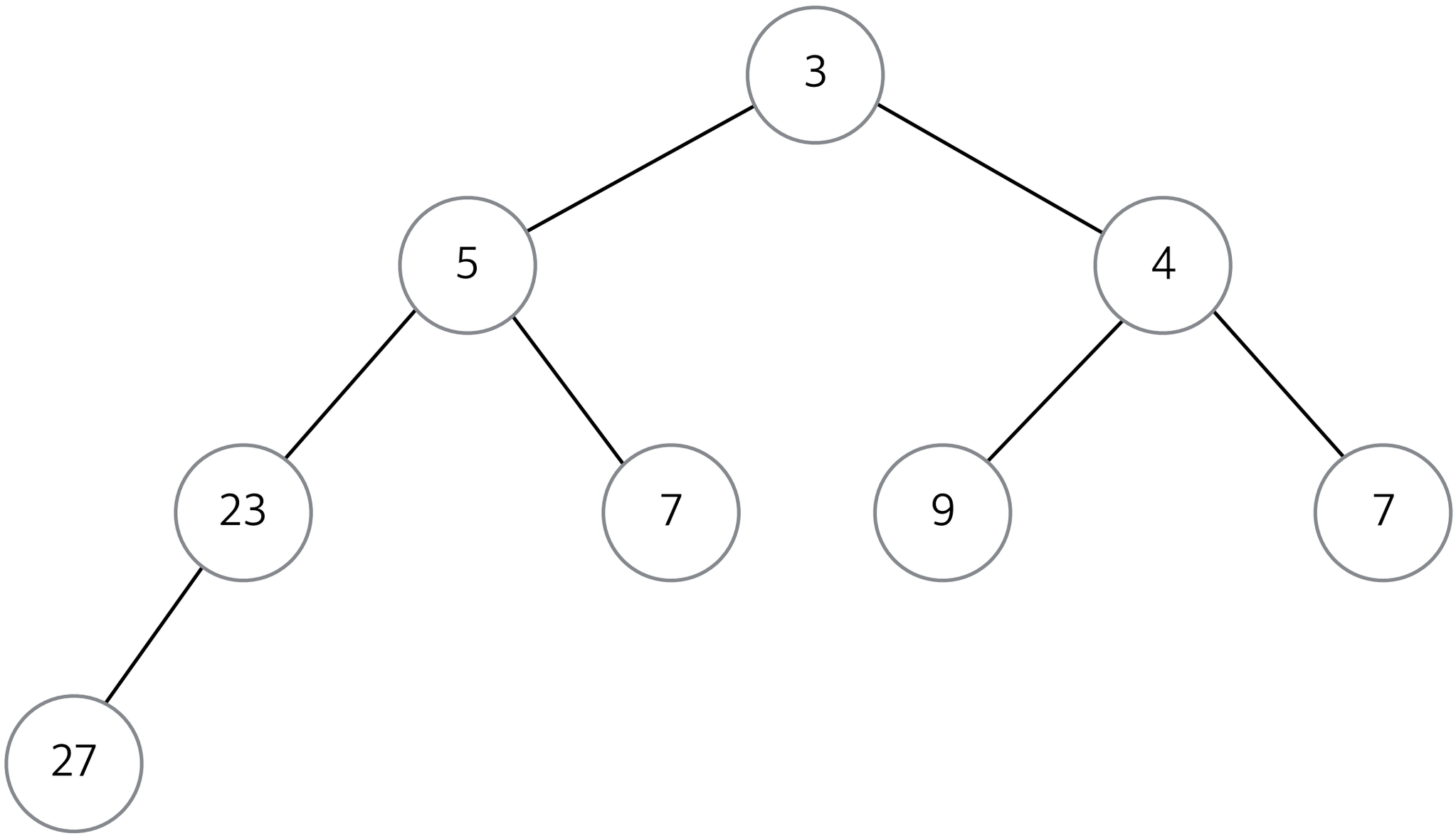
힙 : 값 삽입

heap.insert(27)



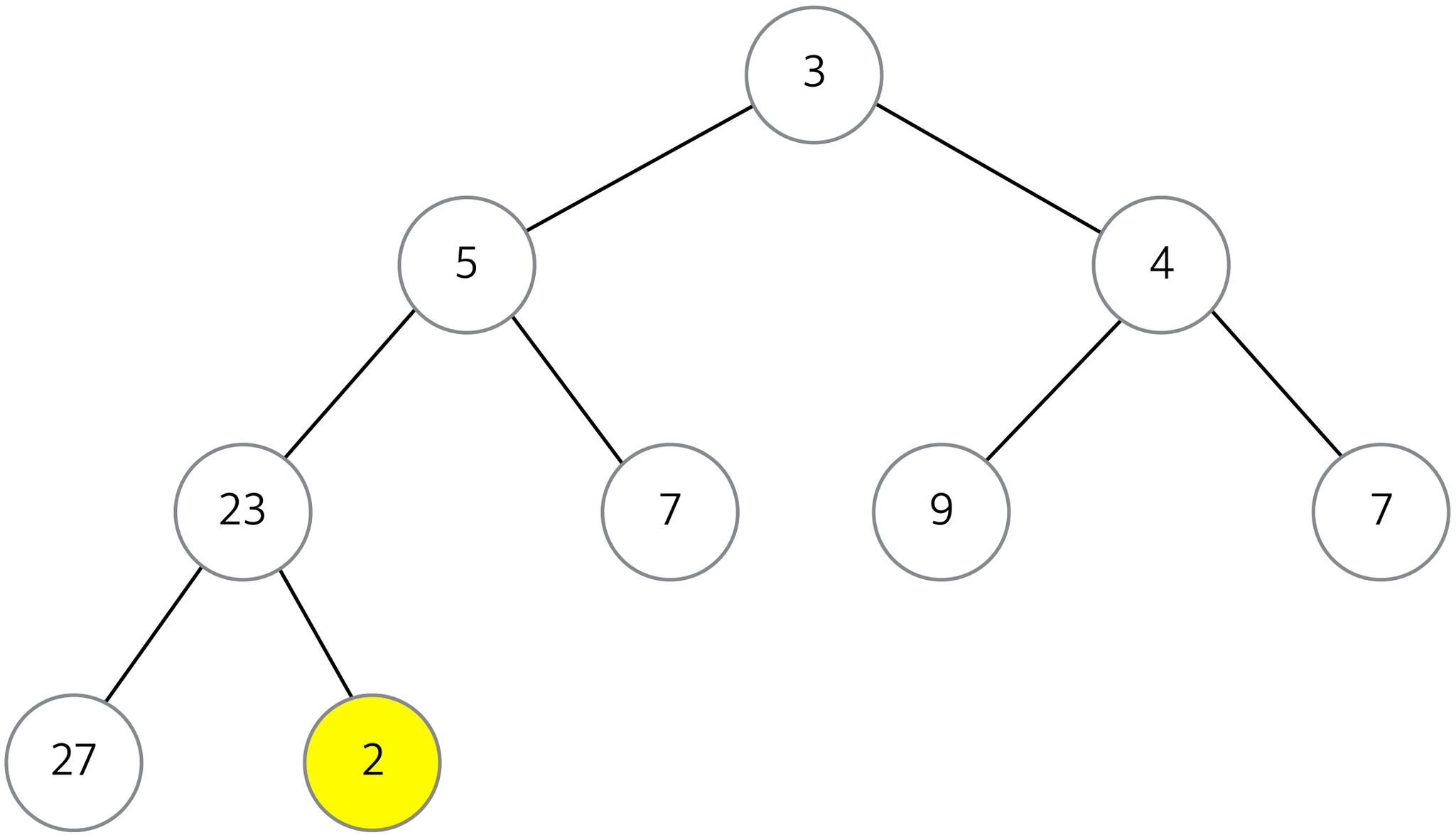
힙 : 값 삽입

heap.insert(2)



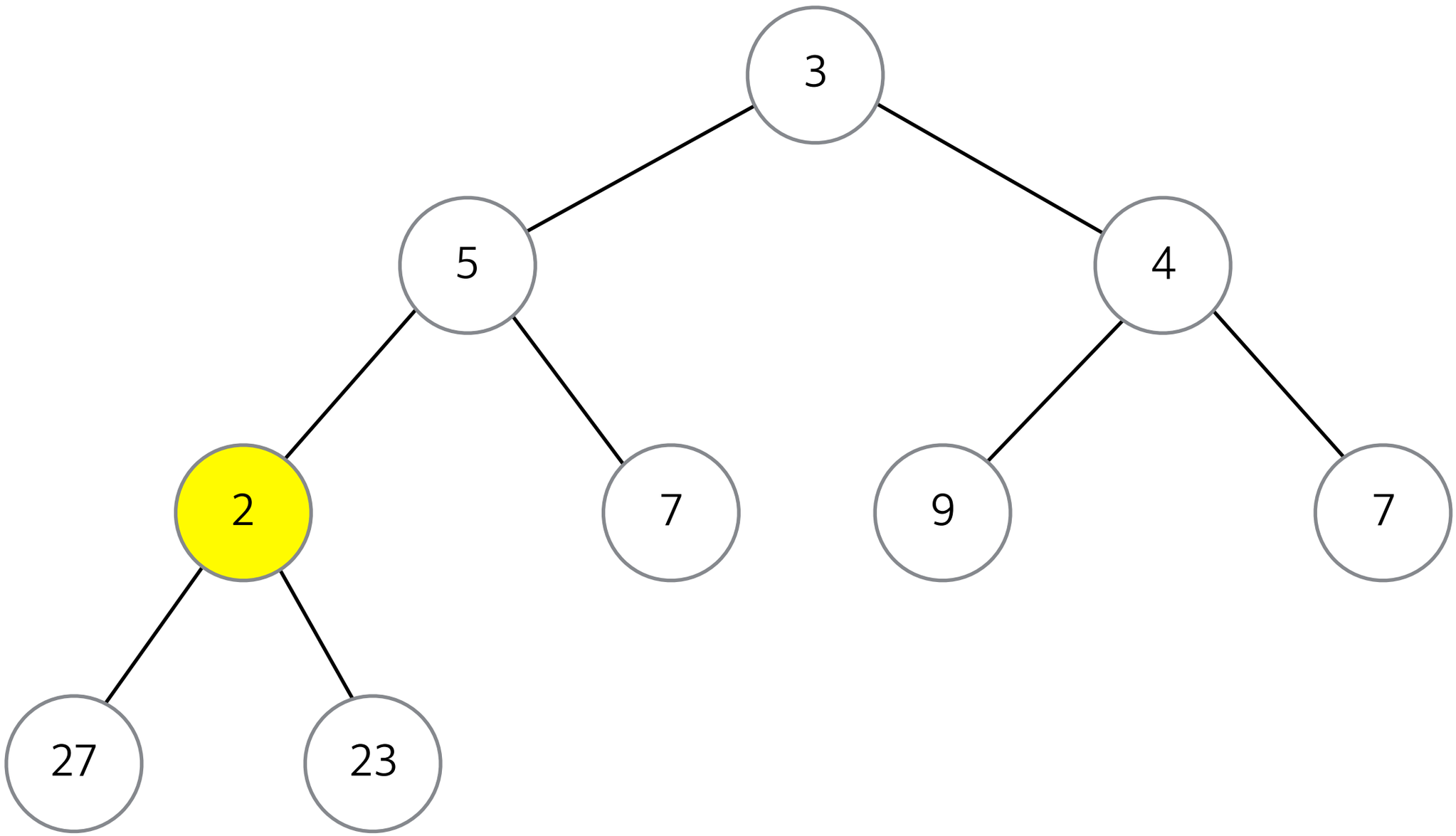
힙 : 값 삽입

heap.insert(2)



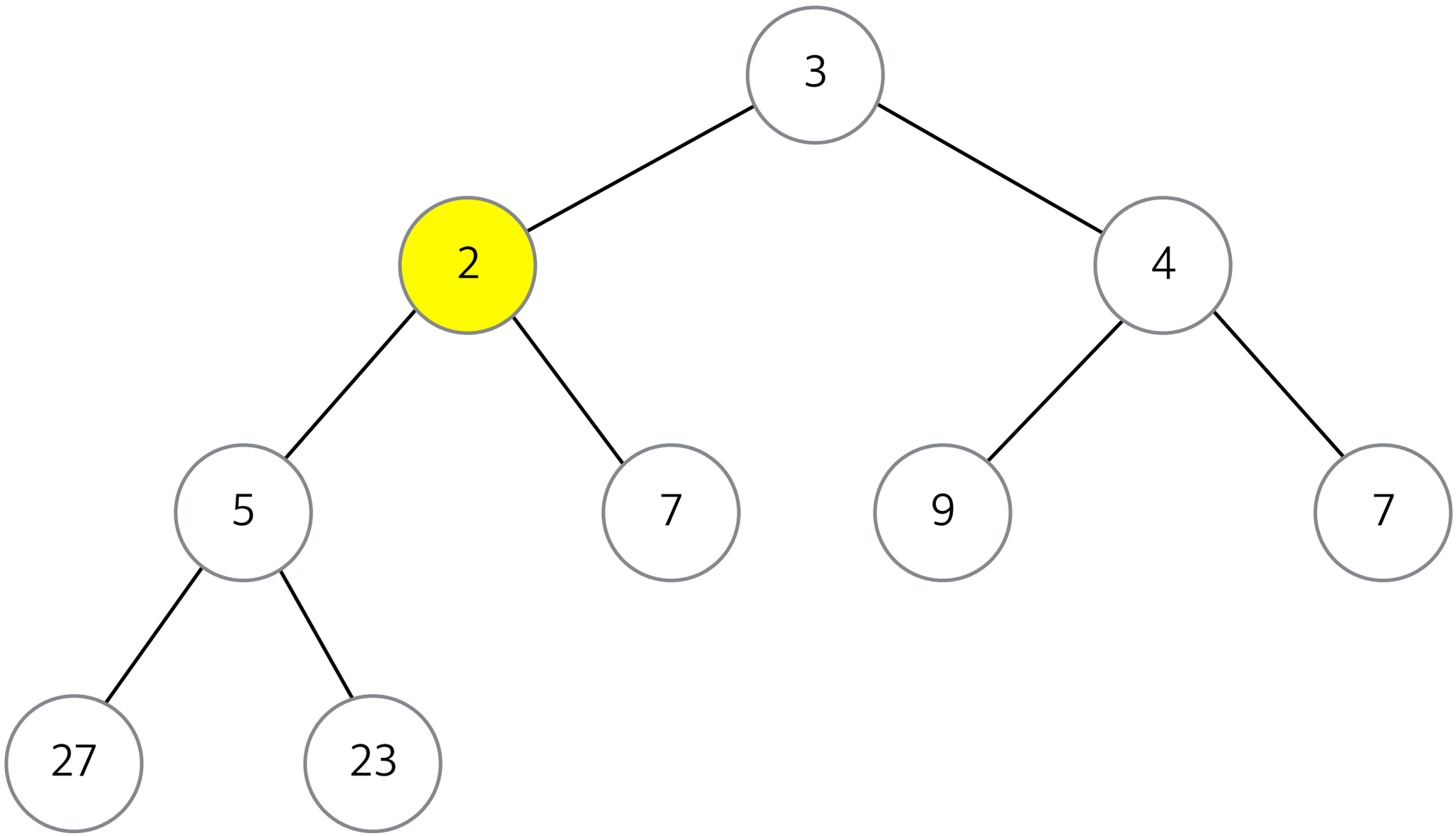
힙 : 값 삽입

heap.insert(2)



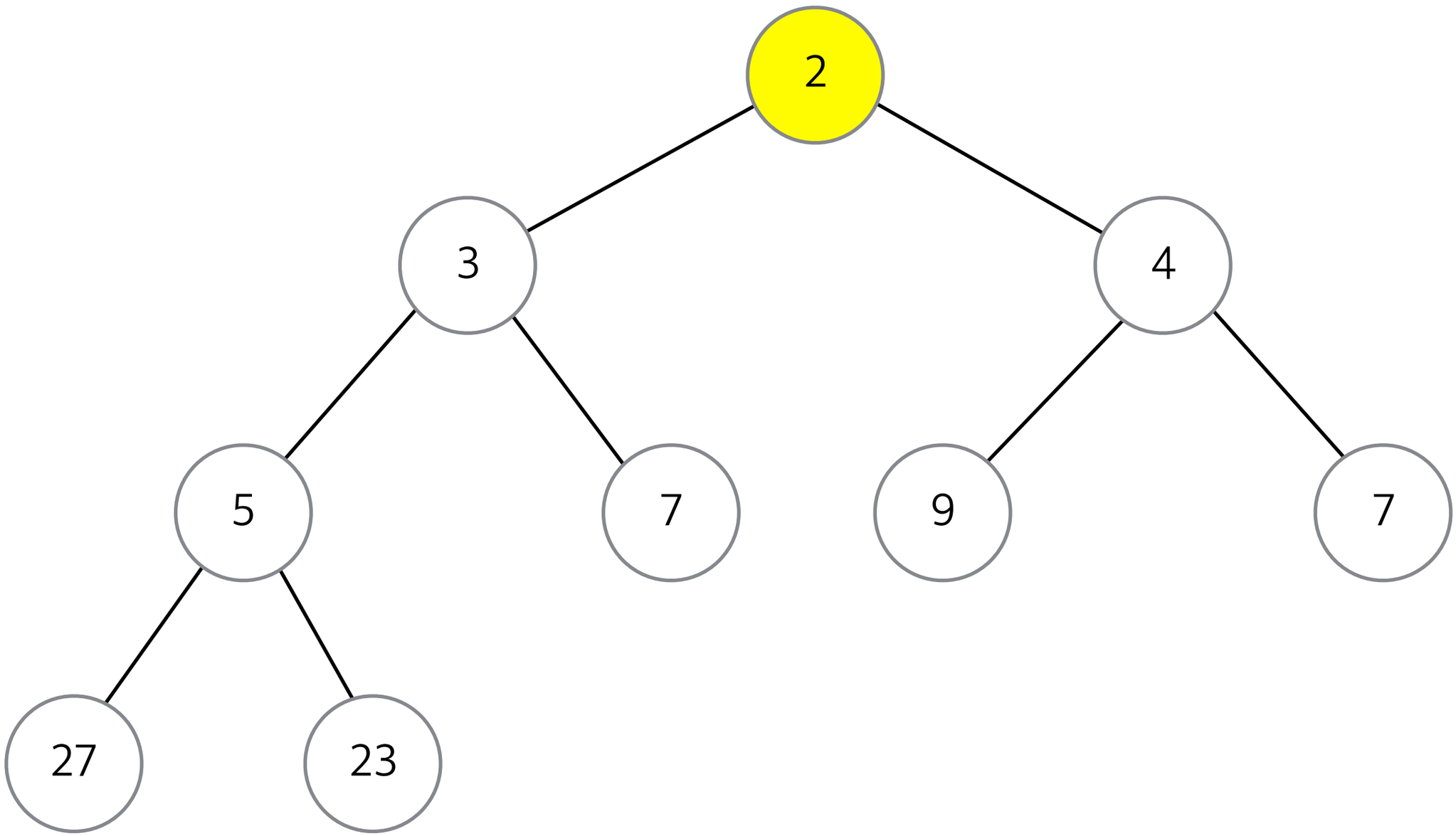
힙 : 값 삽입

heap.insert(2)



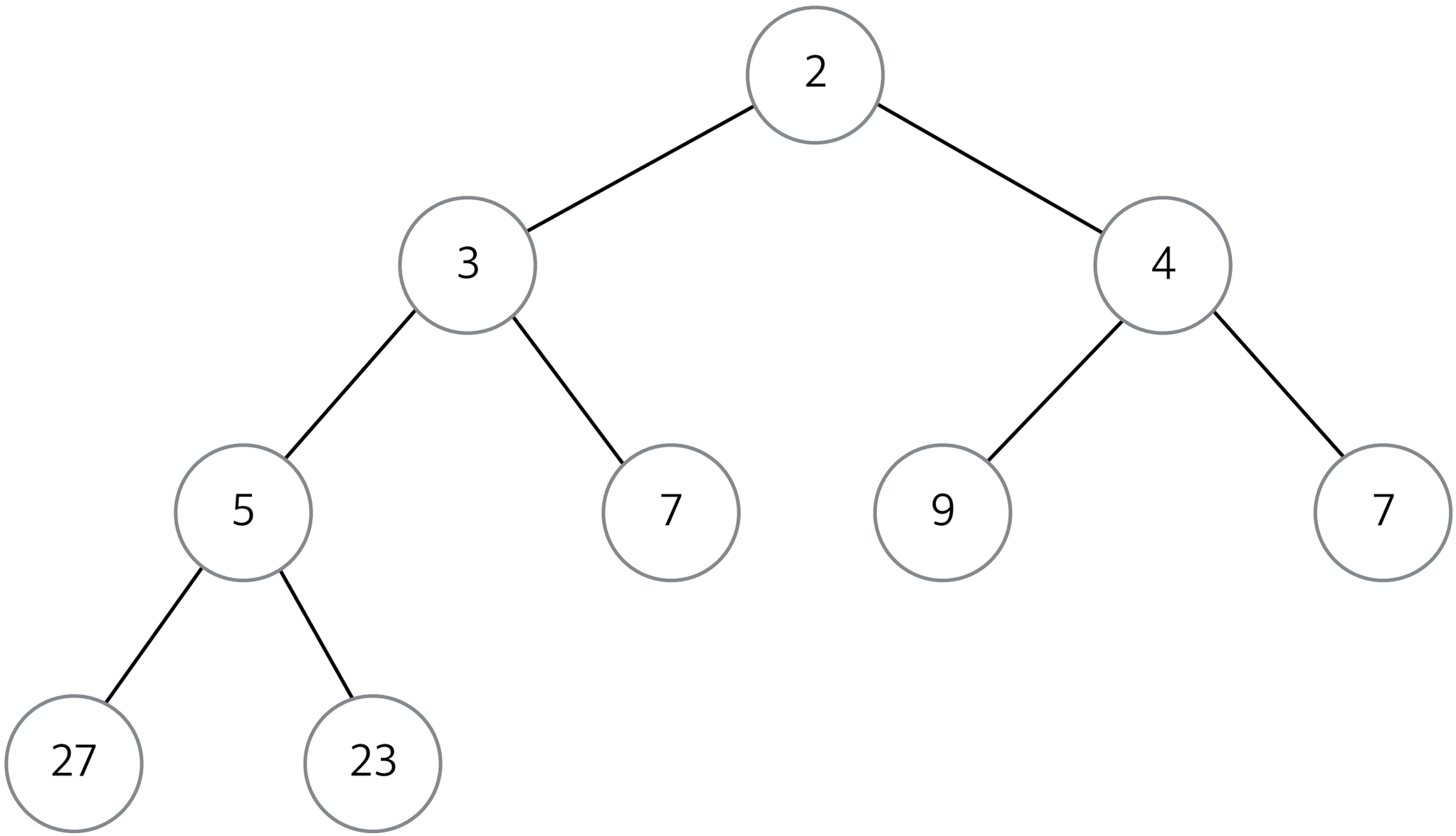
힙 : 값 삽입

heap.insert(2)

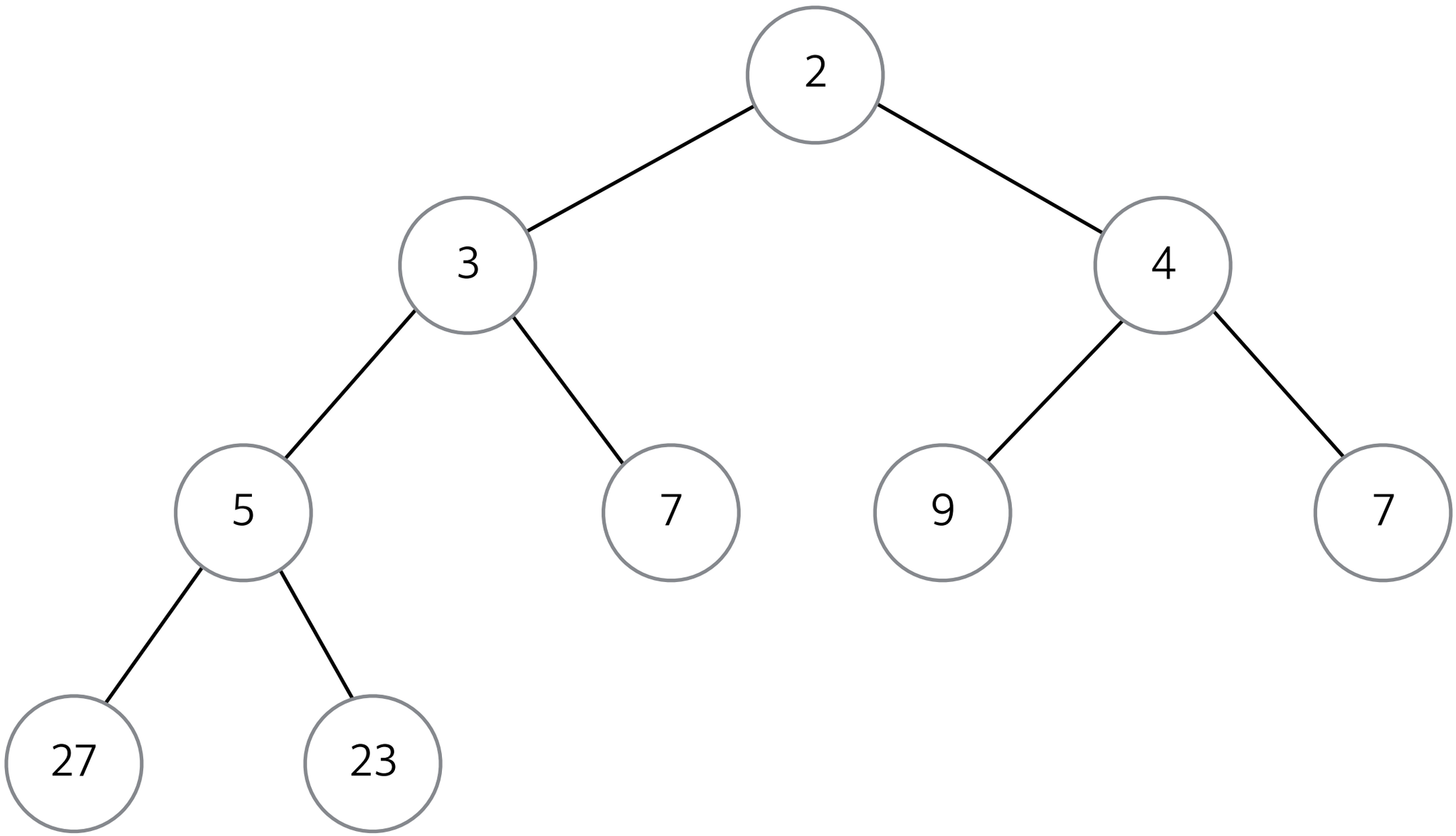


힙 : 값 삽입

heap.insert(2)

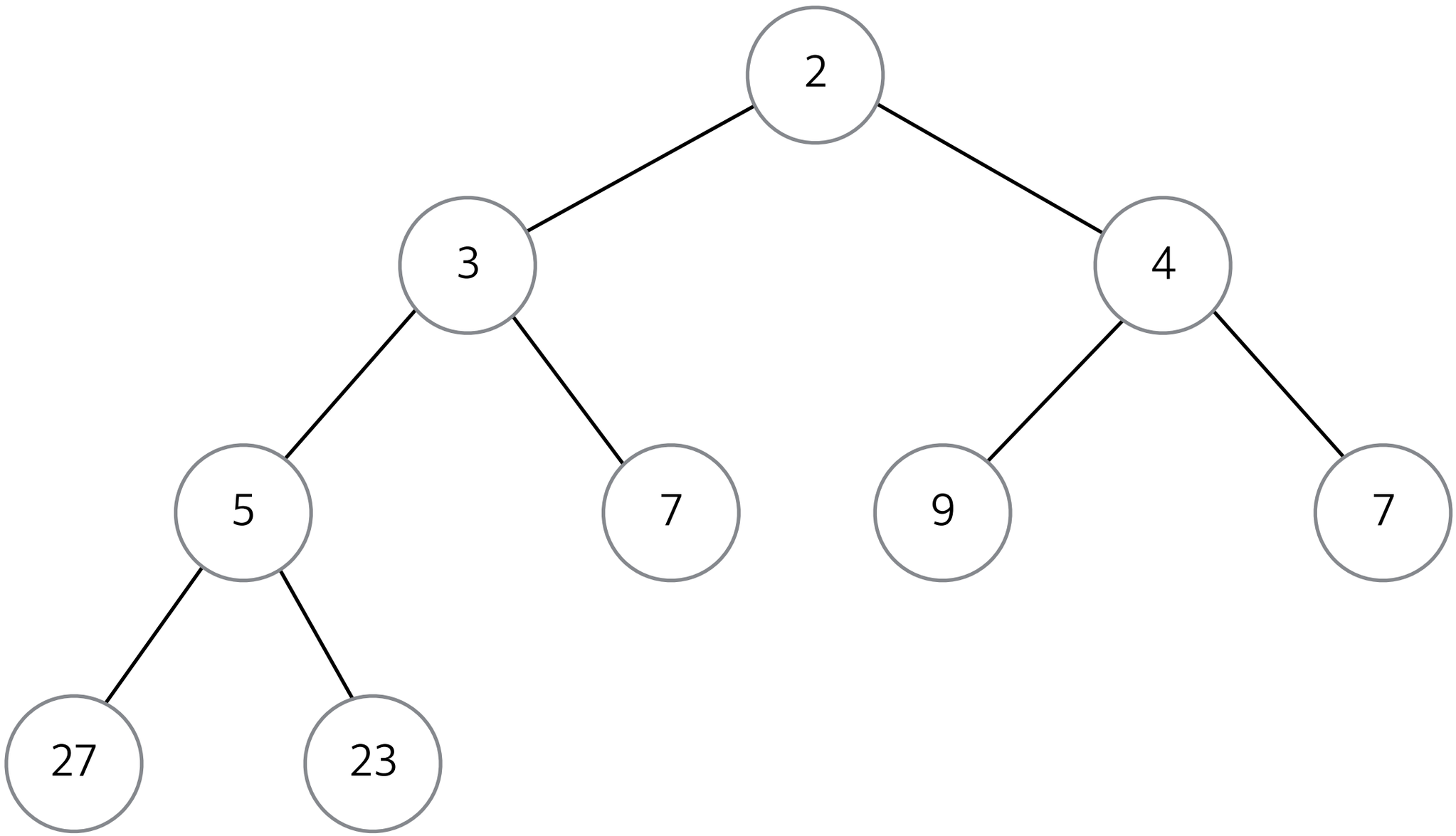


힙 : 값 삽입의 시간복잡도



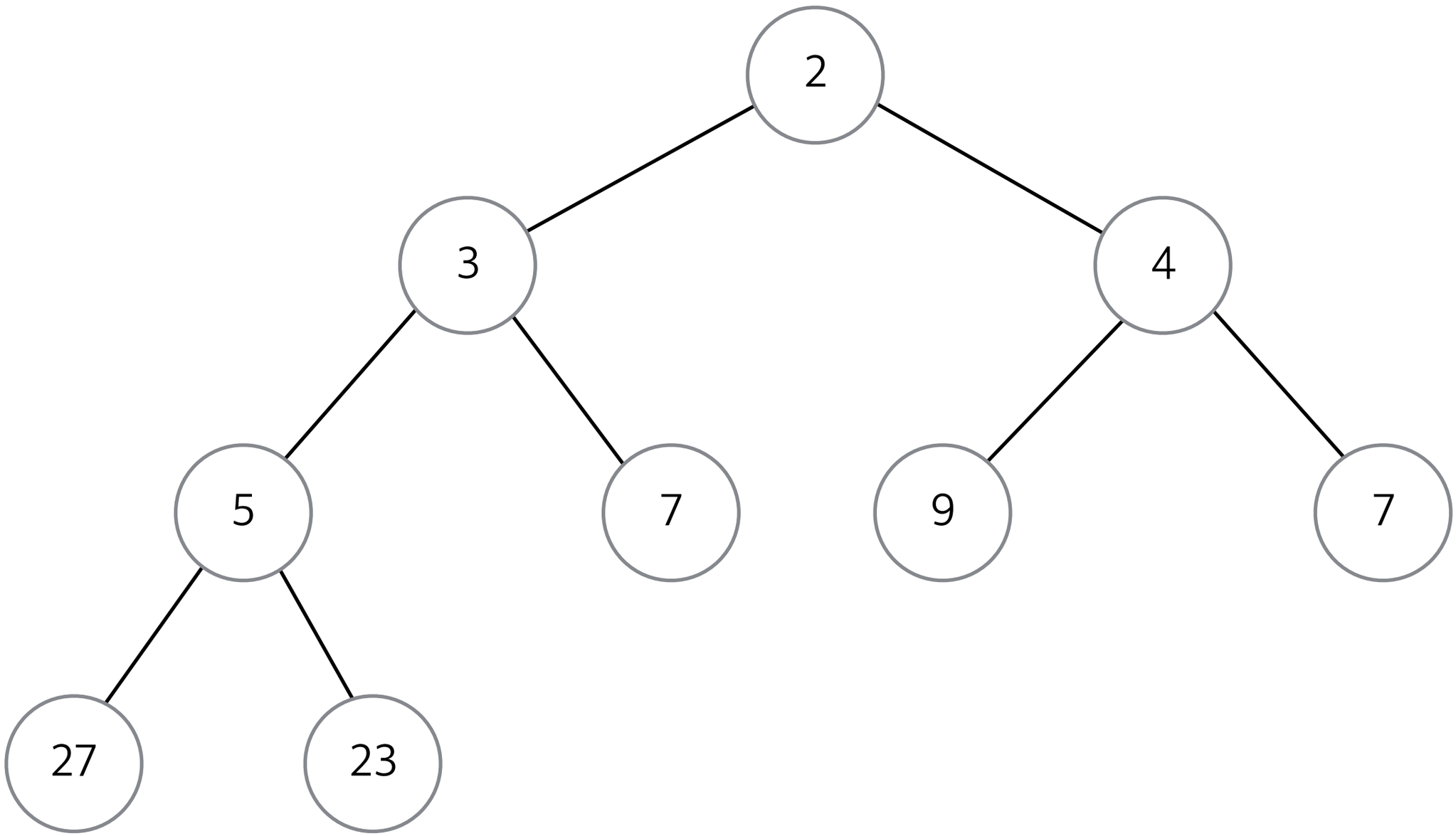
힙 : 값 삽입의 시간복잡도

O(log n)



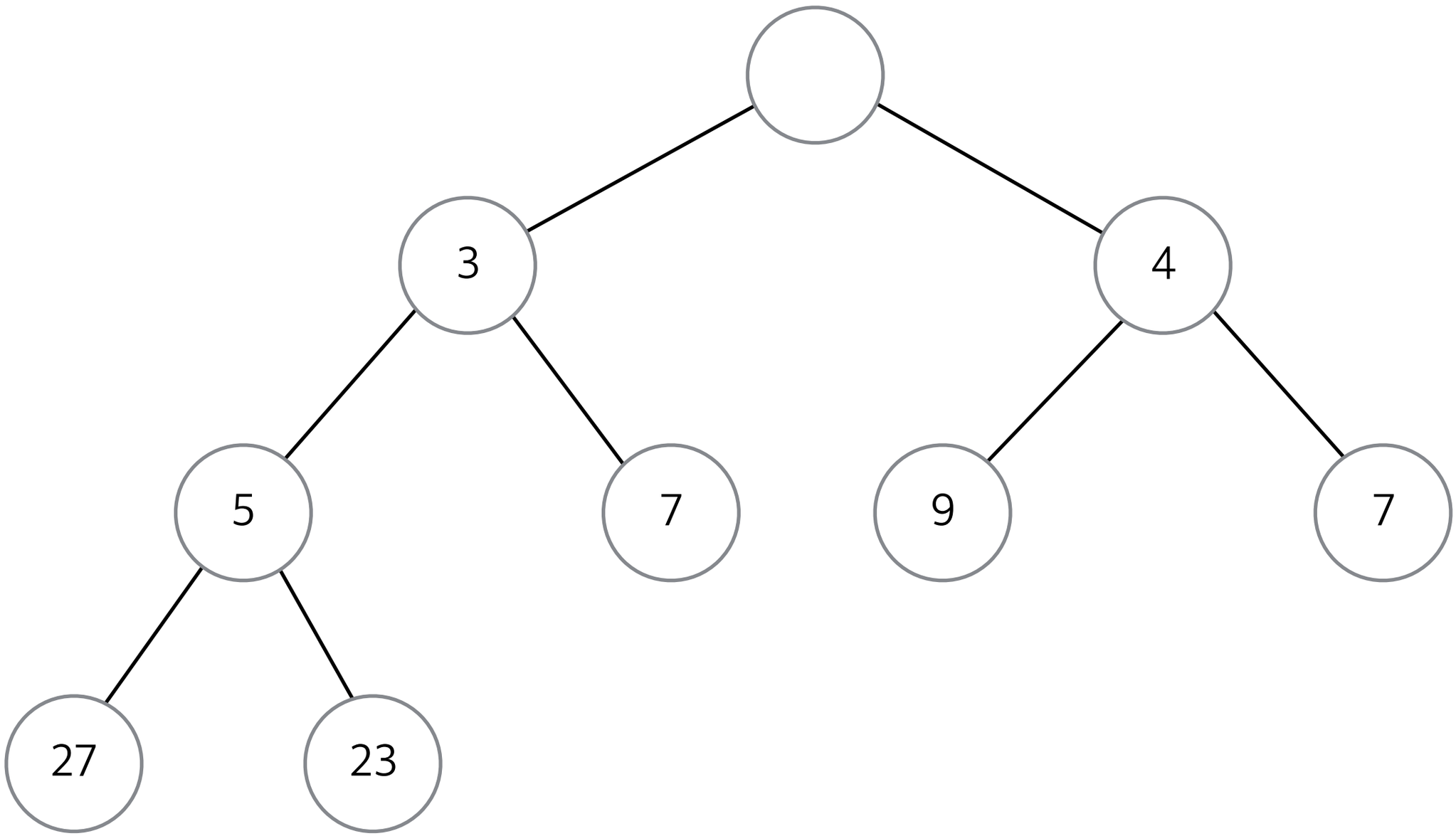
힙 : 값 삭제

heap.pop()



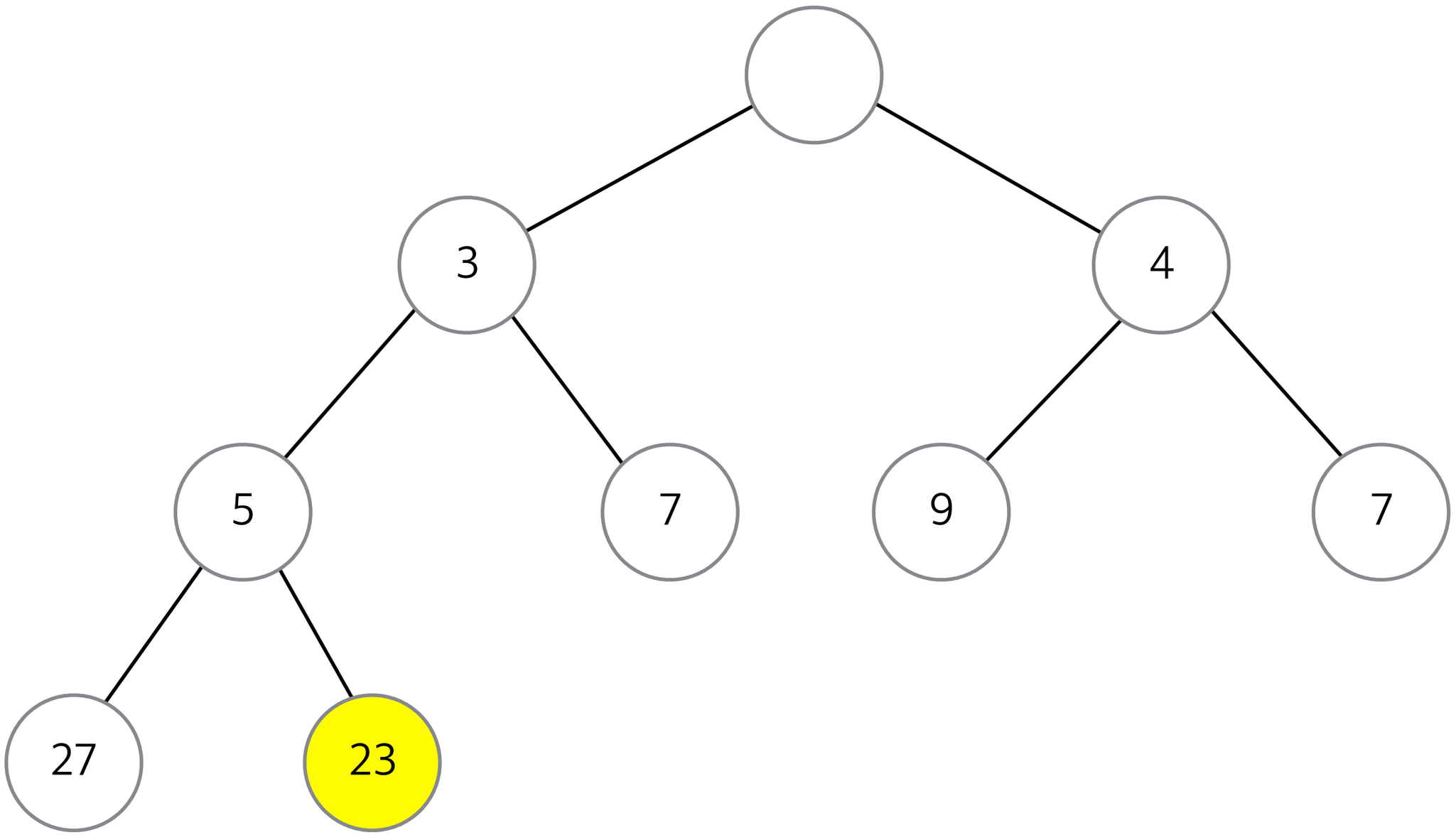
힙 : 값 삭제

heap.pop()



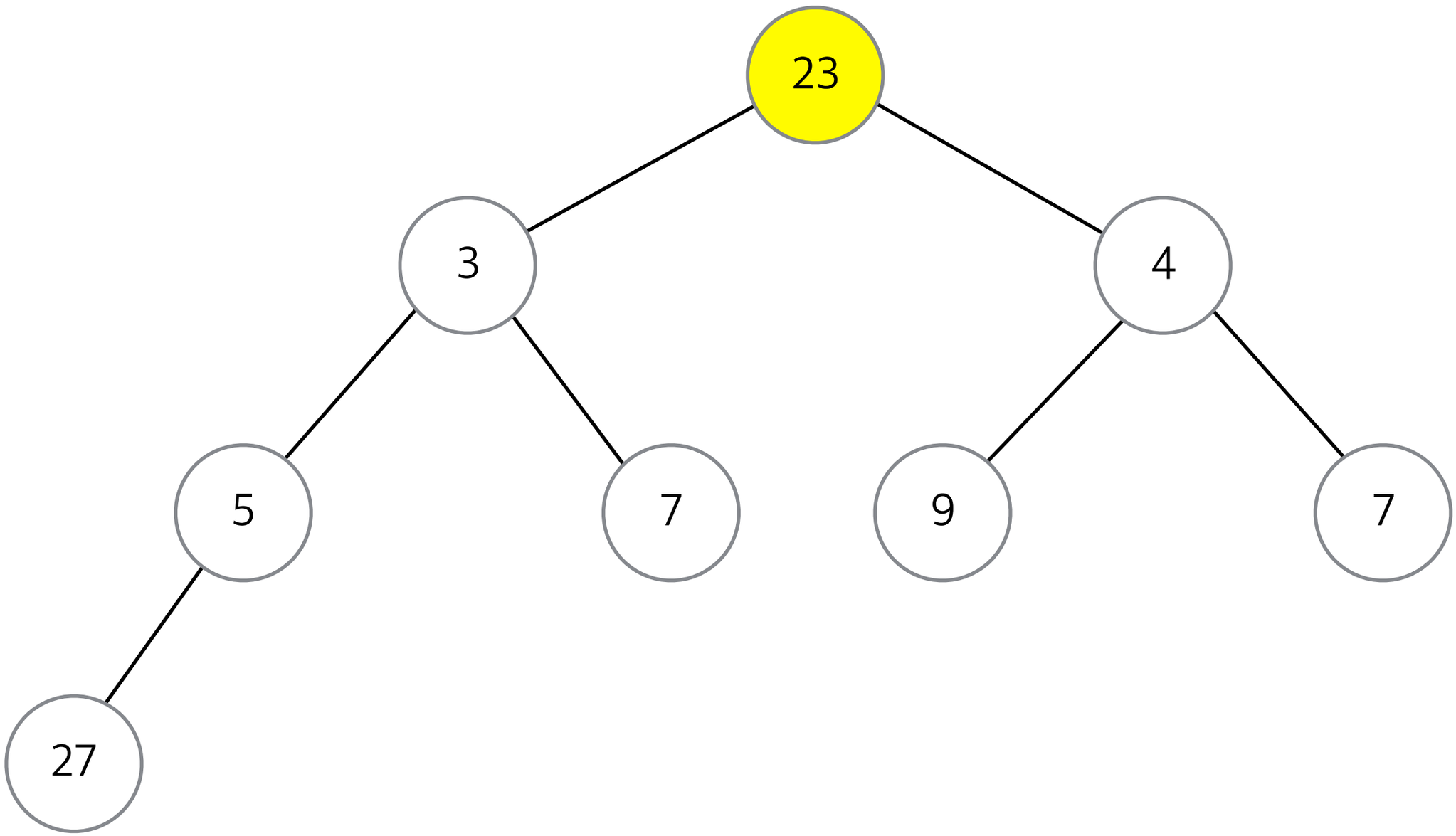
힙 : 값 삭제

heap.pop()



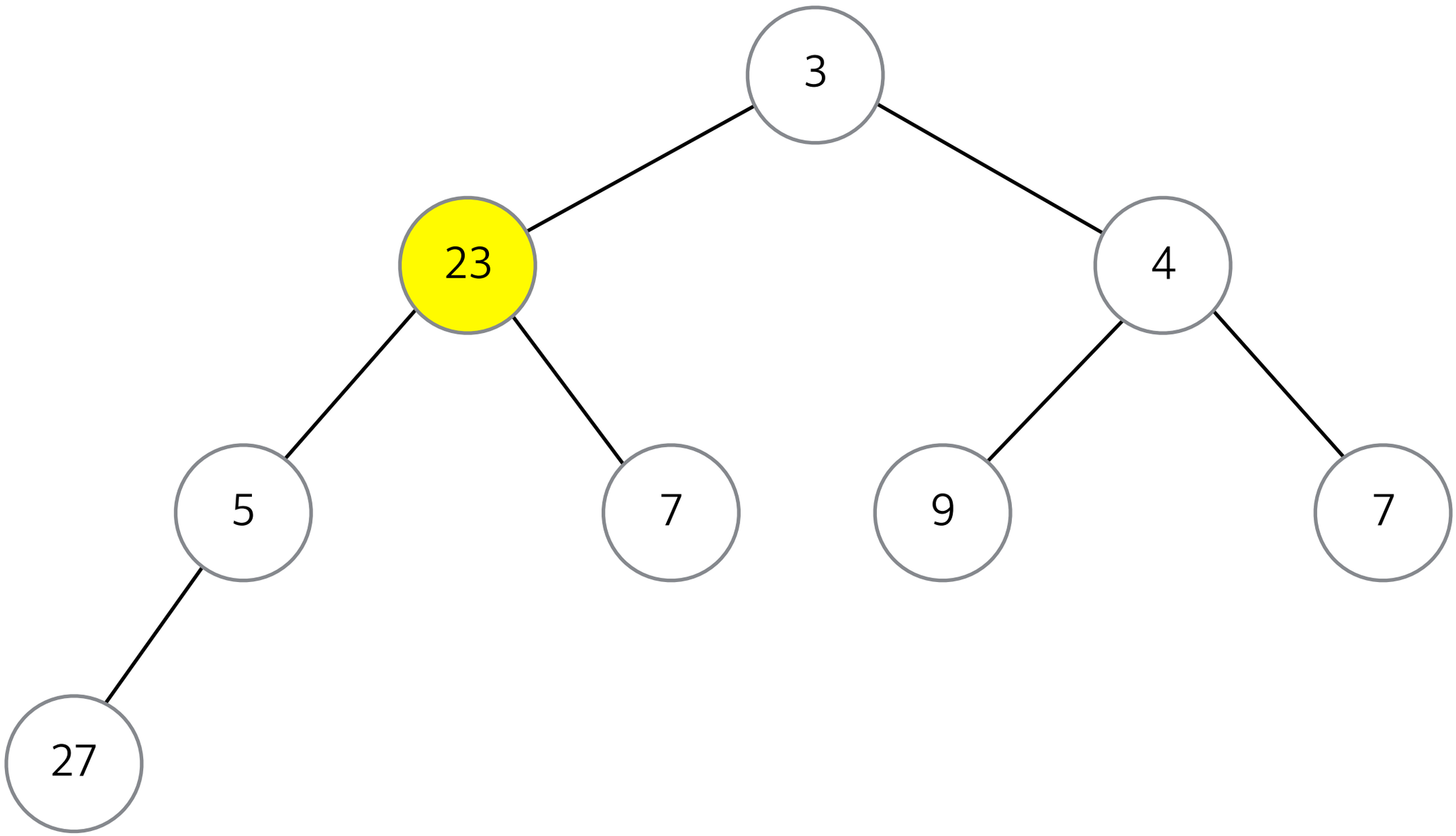
힙 : 값 삭제

heap.pop()



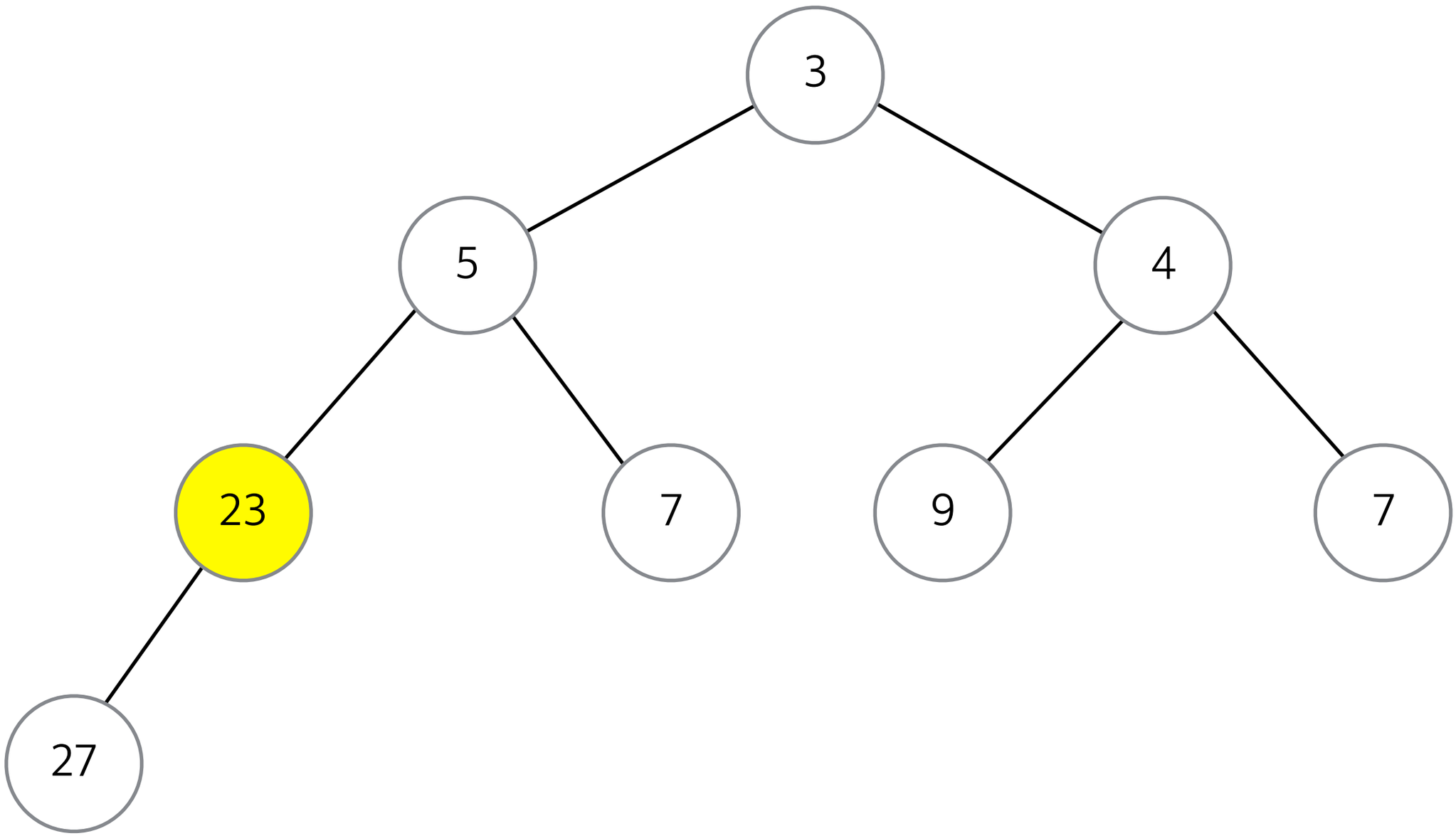
힙 : 값 삭제

heap.pop()



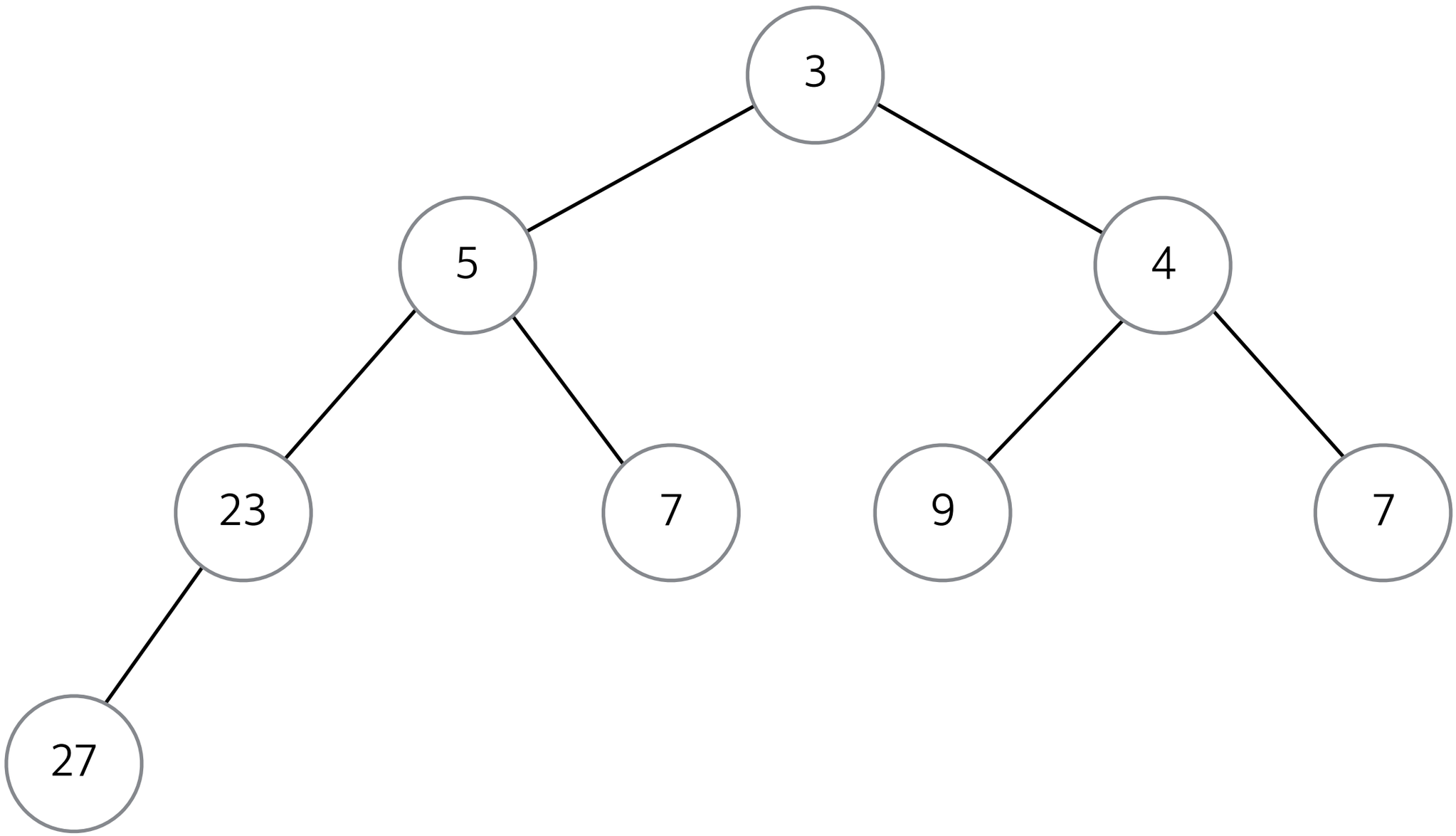
힙 : 값 삭제

heap.pop()

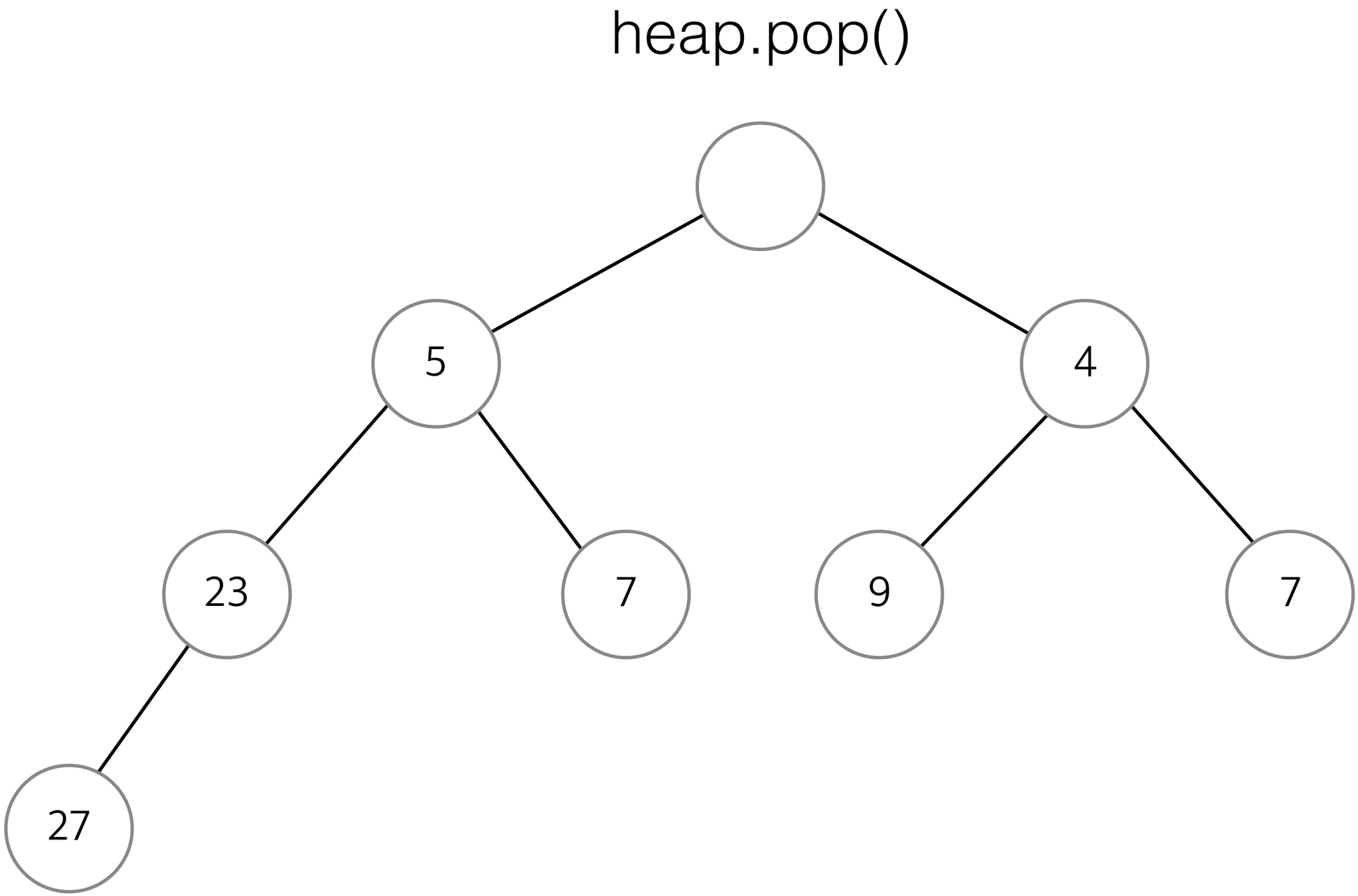


힙 : 값 삭제

heap.pop()

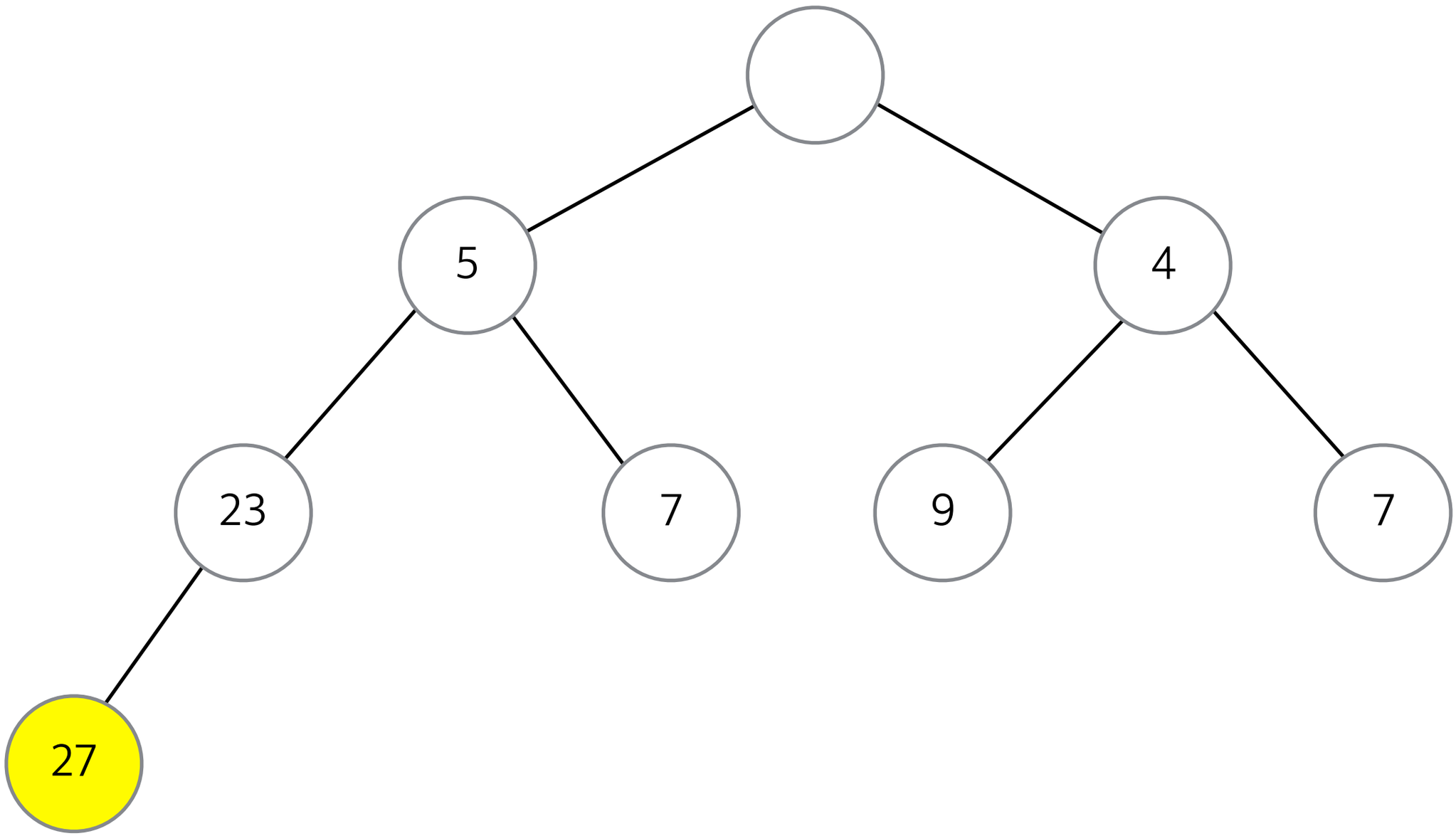


힙 : 값 삭제



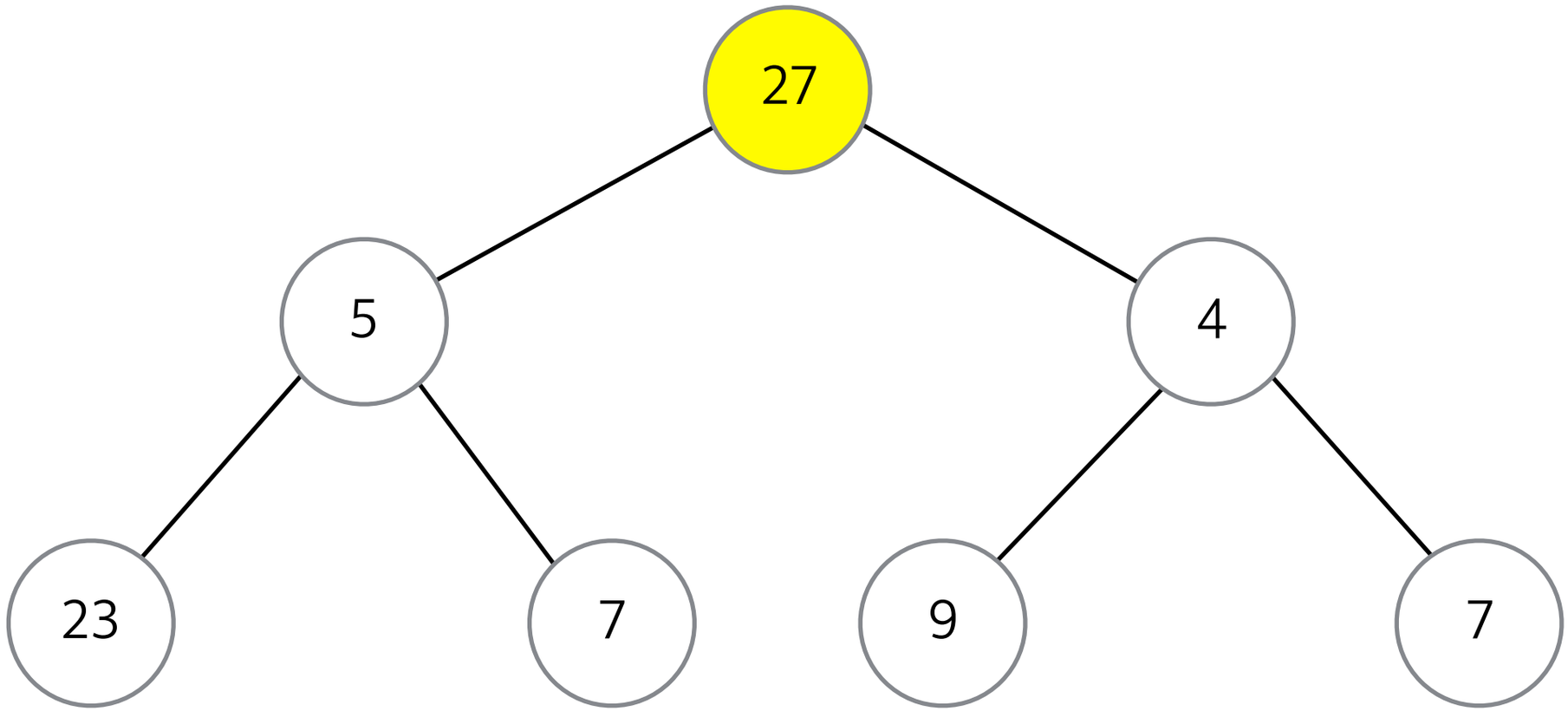
힙 : 값 삭제

heap.pop()



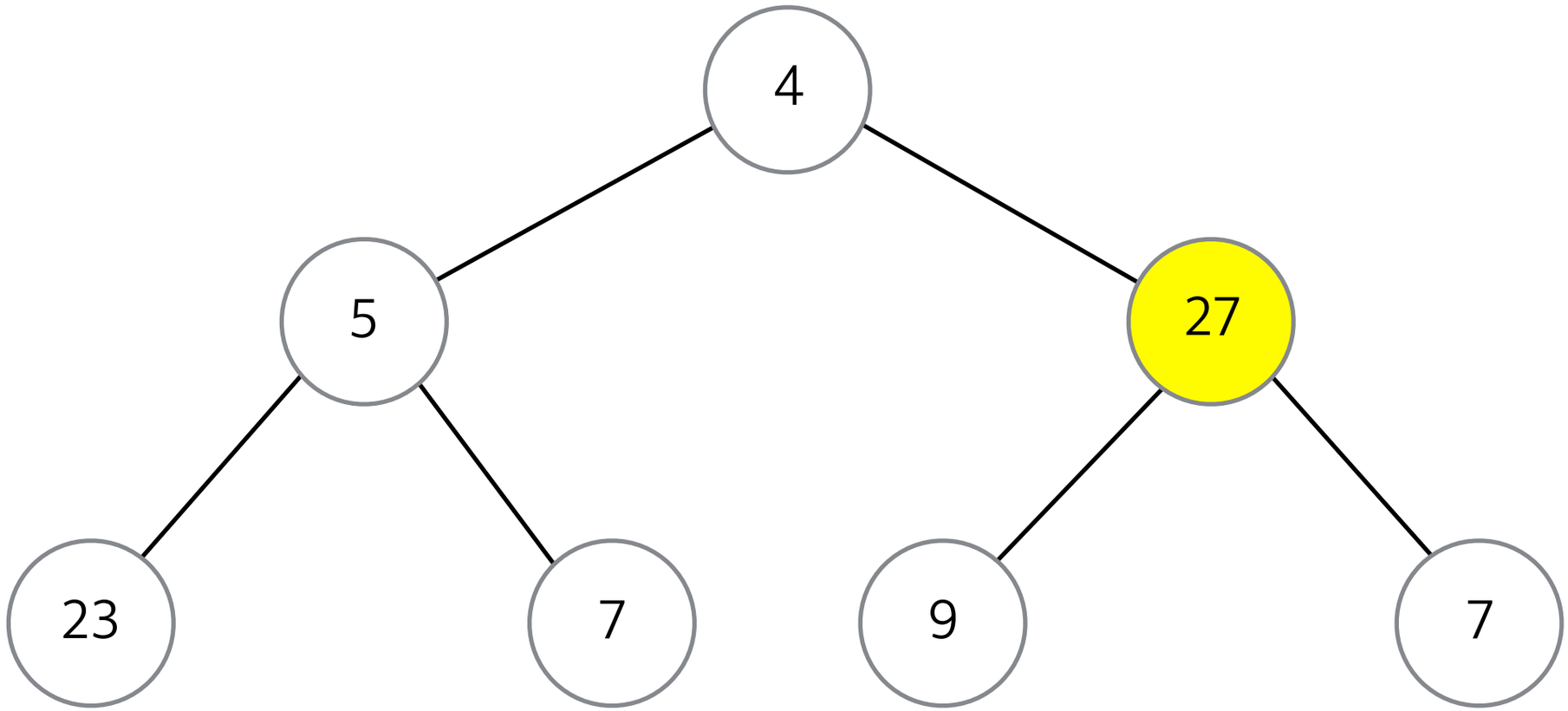
힙 : 값 삭제

heap.pop()



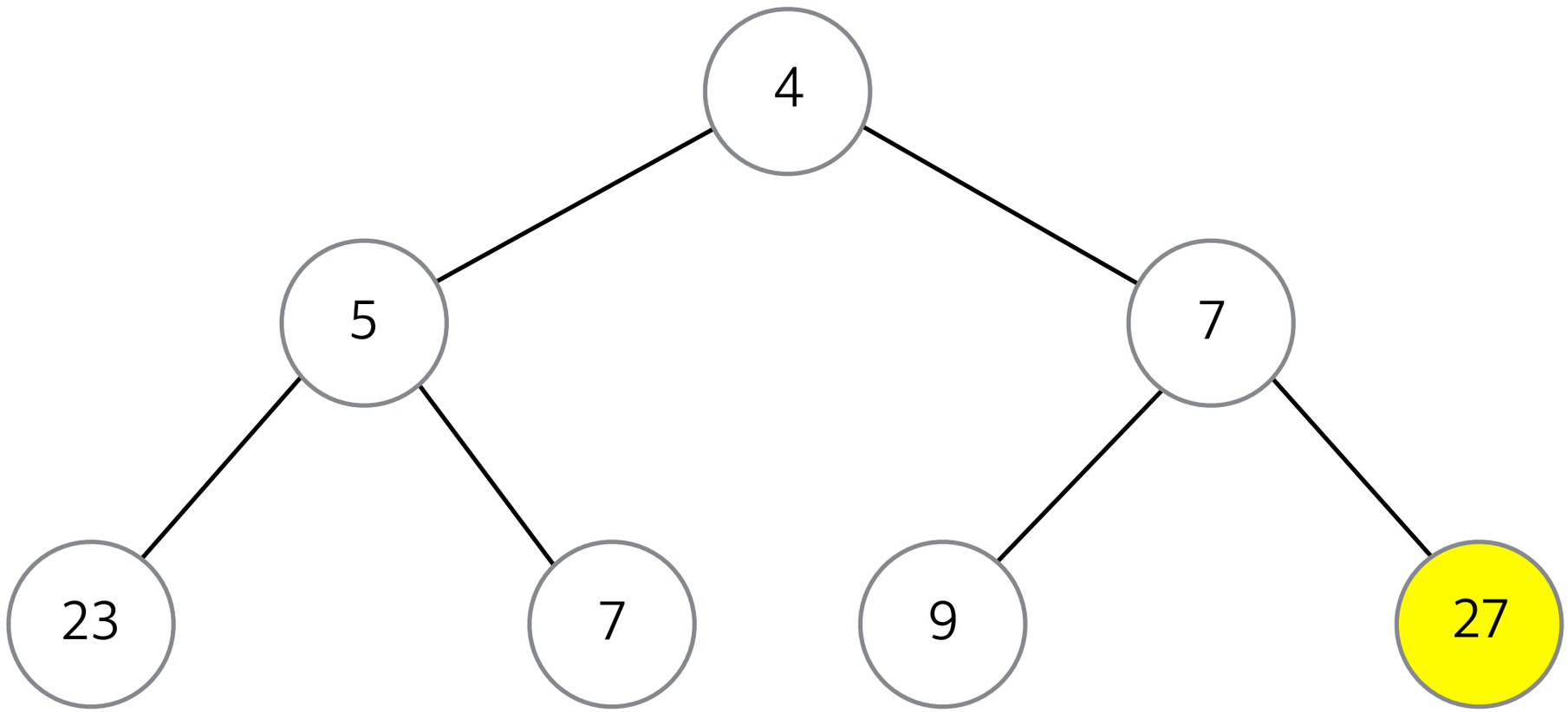
힙 : 값 삭제

heap.pop()



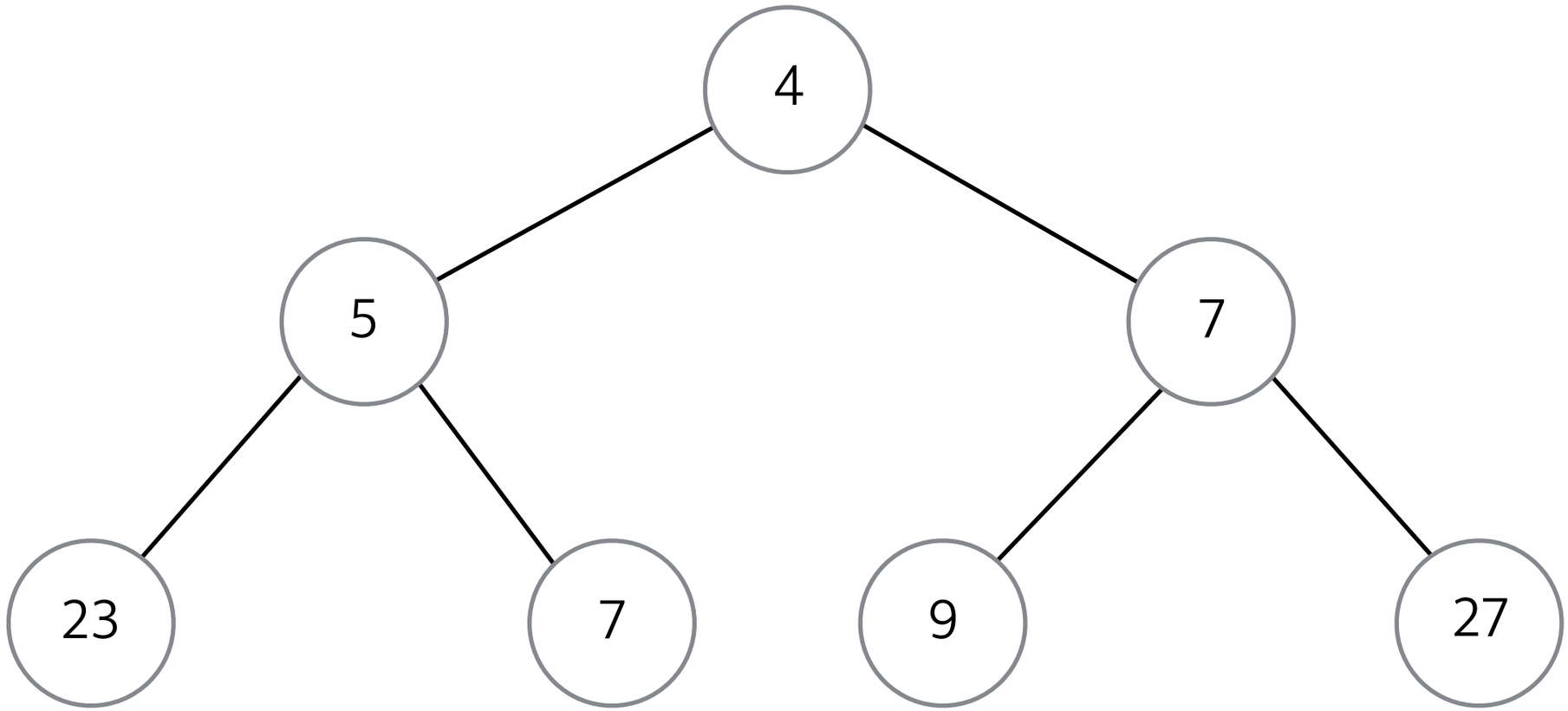
힙 : 값 삭제

heap.pop()

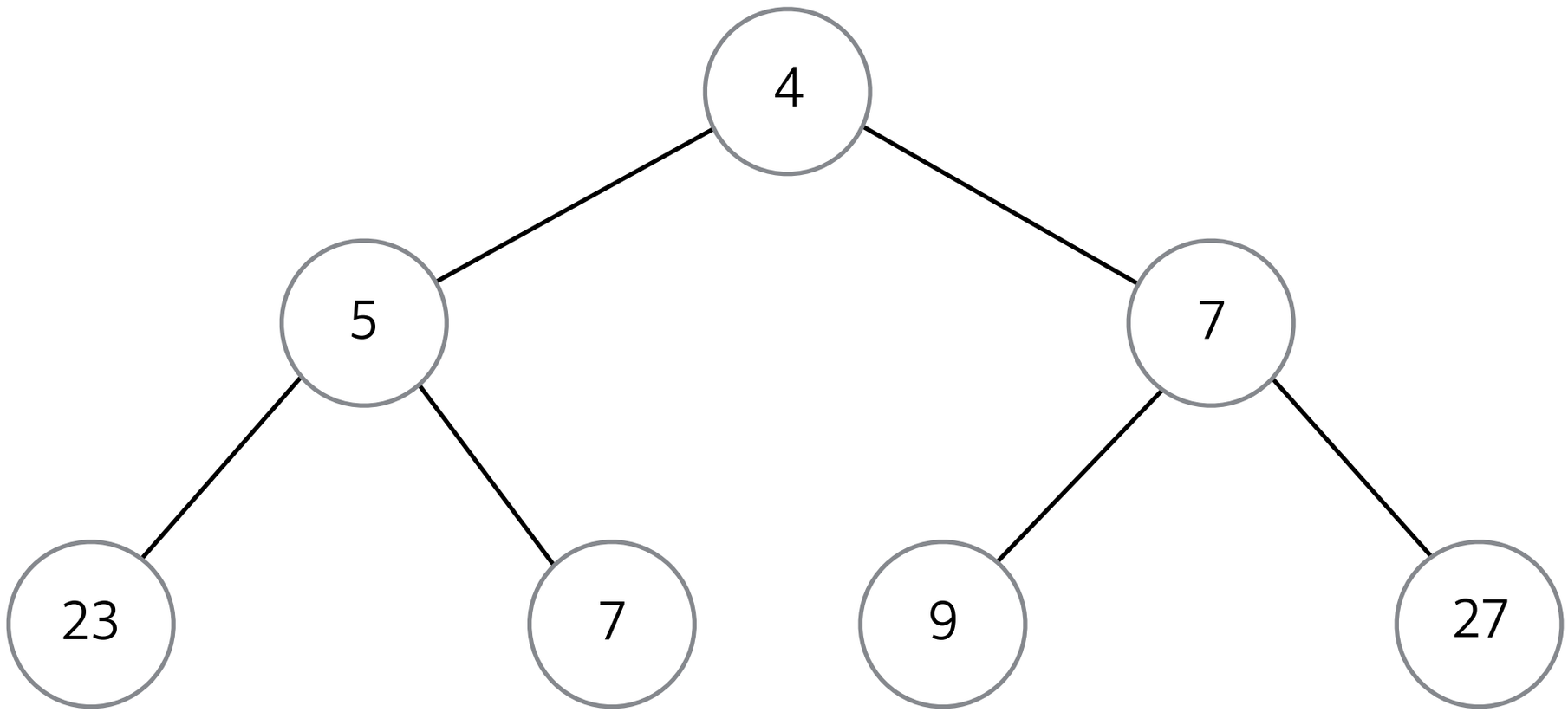


힙 : 값 삭제

heap.pop()

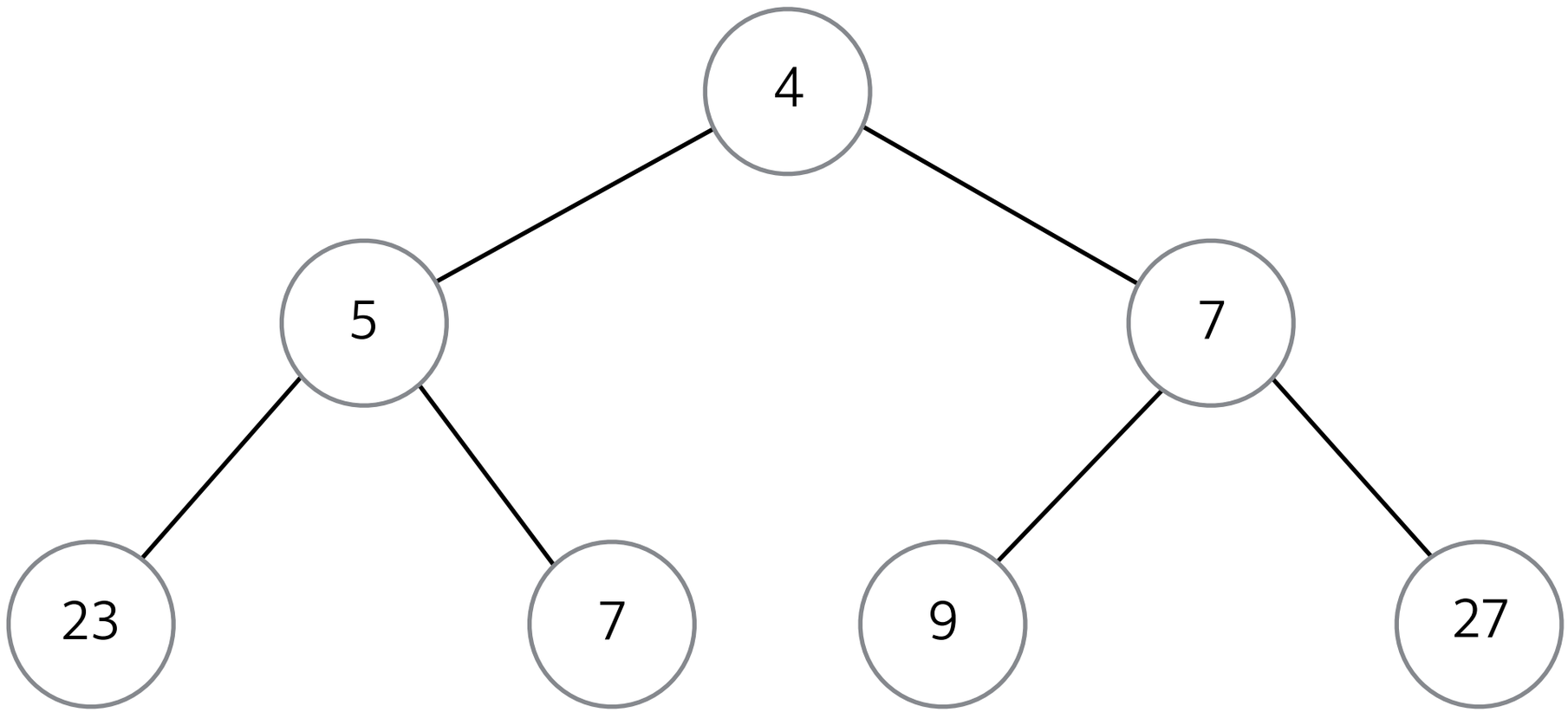


힙 : 값 삭제의 시간복잡도



힙 : 값 삭제의 시간복잡도

O(log n)



우선순위 큐의 구현 요약

	리스트	힙
값의 삽입	O(1)	O(log n)
값의 삭제	O(n)	O(log n)

주차별 커리큘럼

1주차	과정 소개, 배열, 연결리스트, 클래스
2주차	스택, 큐, 해싱
3주차	시간복잡도
4주차	트리, 트리순회, 재귀호출
5주차	**힙**
6주차	그래프 소개, DFS
7주차	그래프 심화, BFS
8주차	강의 요약, 알고리즘 과정 소개

[예제 1] 숫자 합병

K개의 정렬된 배열을 모두 합쳐 하나의 정렬된 배열을 출력

(단, 1 < k, n < 100,000)

입력의 예

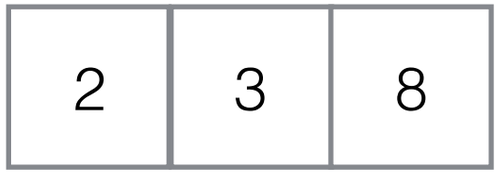
```
3
1 2 4 6
2 3 8
2 3 5 8 10
```

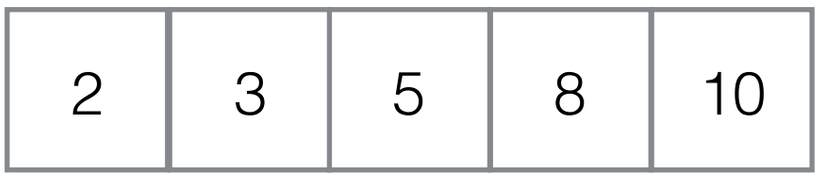
출력의 예

```
1 2 2 2 3 3 4 5 6 8 8 10
```

[예제 1] 숫자 합병

모두 합쳐서 다시 정렬





[예제 1] 숫자 합병

모두 합쳐서 다시 정렬

1	2	4	6	2	3	8	2	3	5	8	10

[예제 1] 숫자 합병

모두 합쳐서 다시 정렬

1	2	2	2	3	3	4	5	6	8	8	10

[예제 1] 숫자 합병

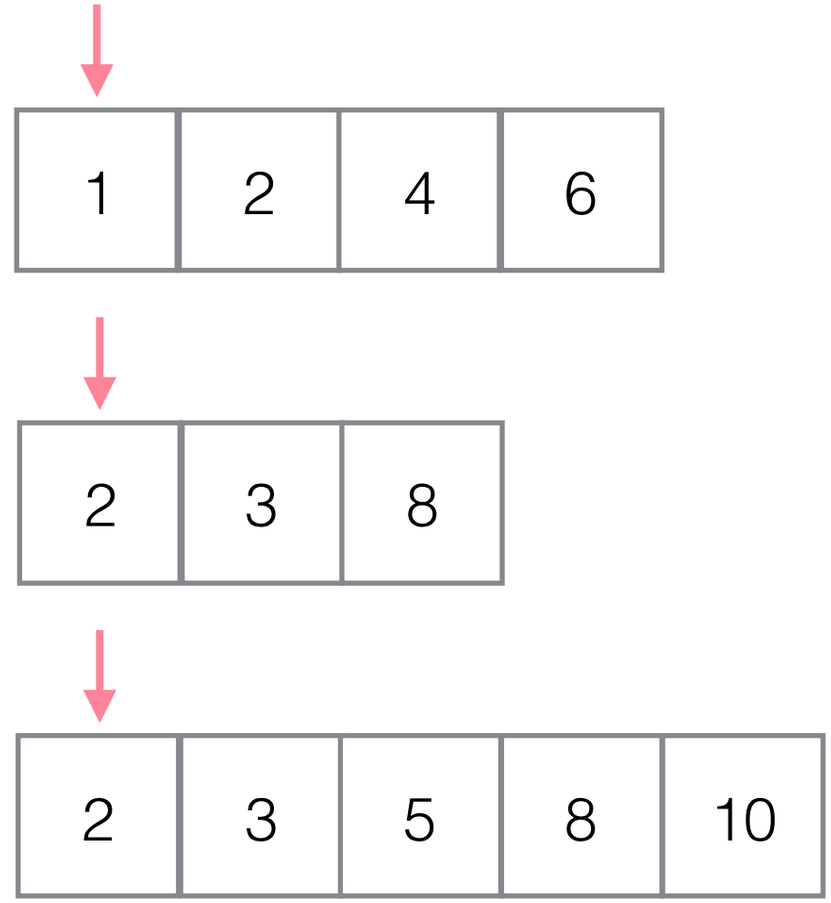
모두 합쳐서 다시 정렬

1	2	2	2	3	3	4	5	6	8	8	10

O(n log n)

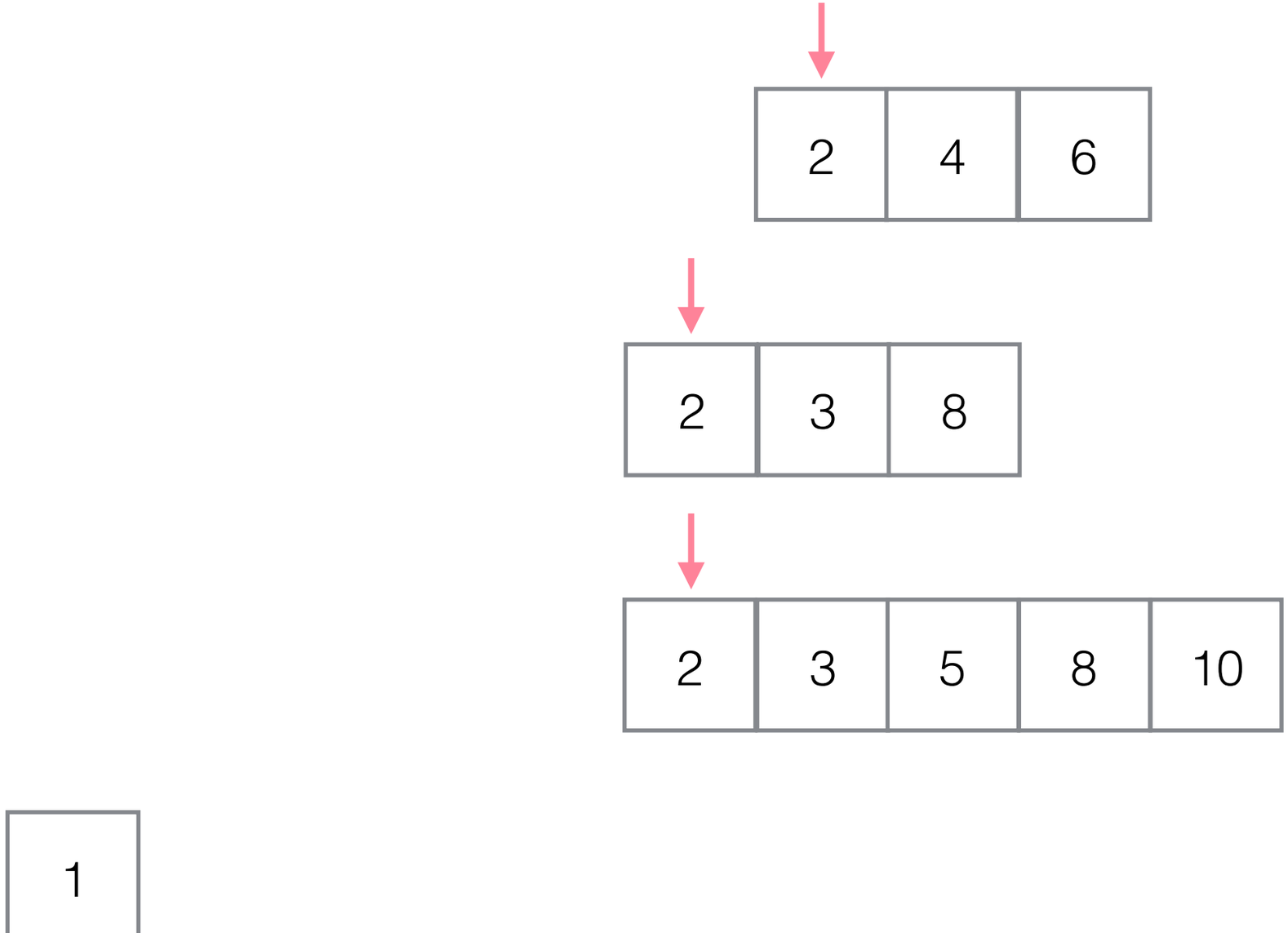
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



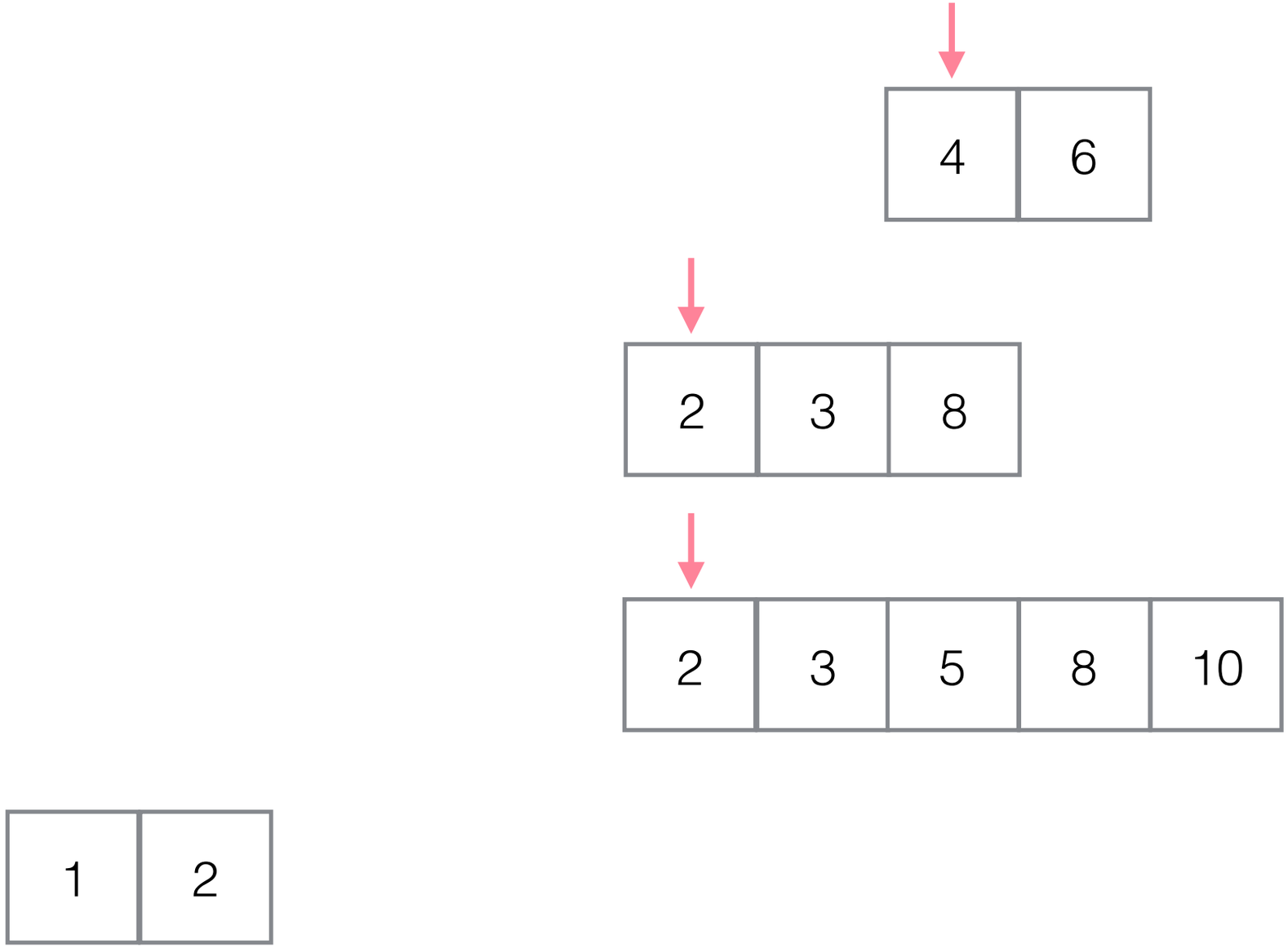
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



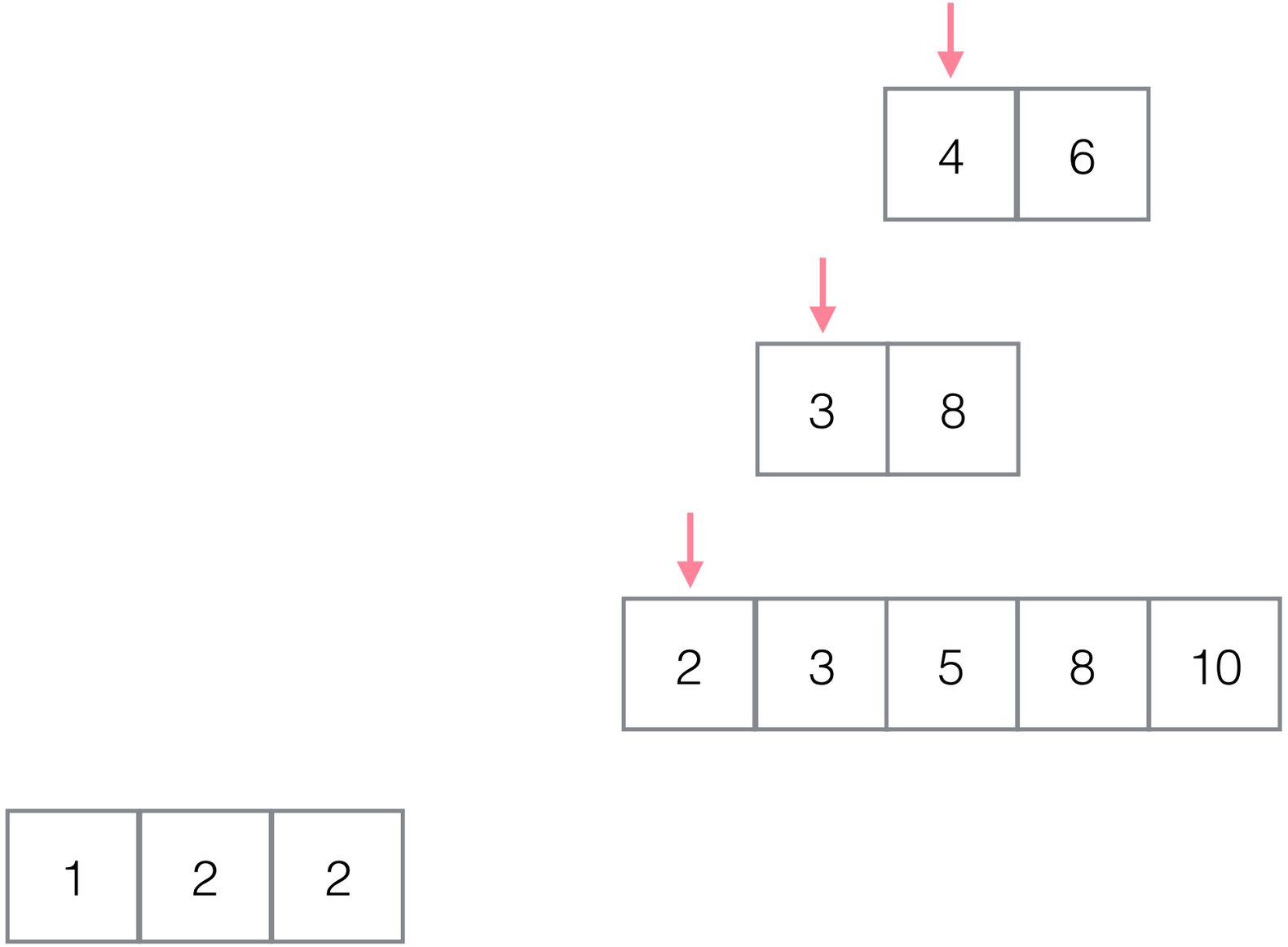
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



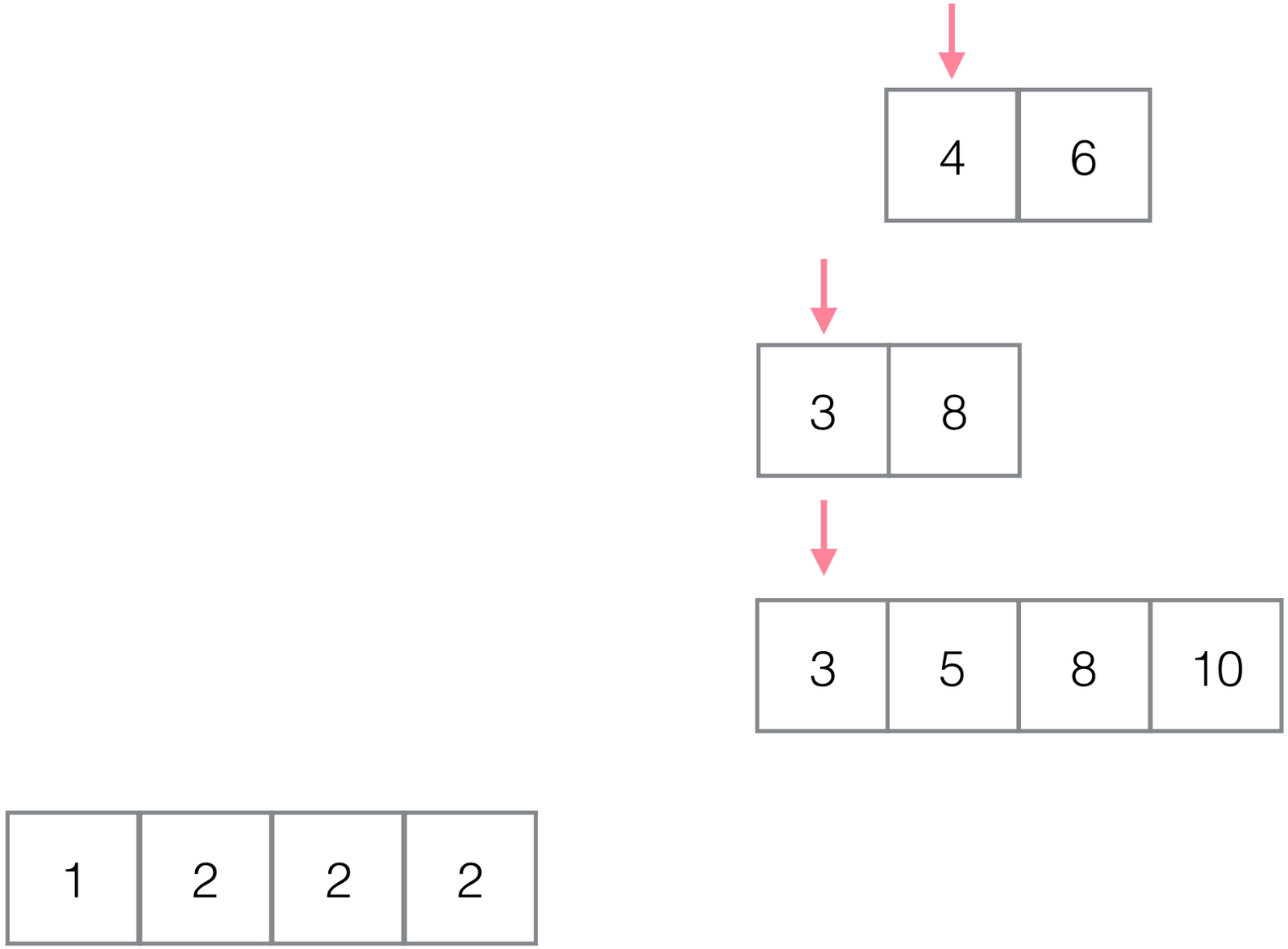
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



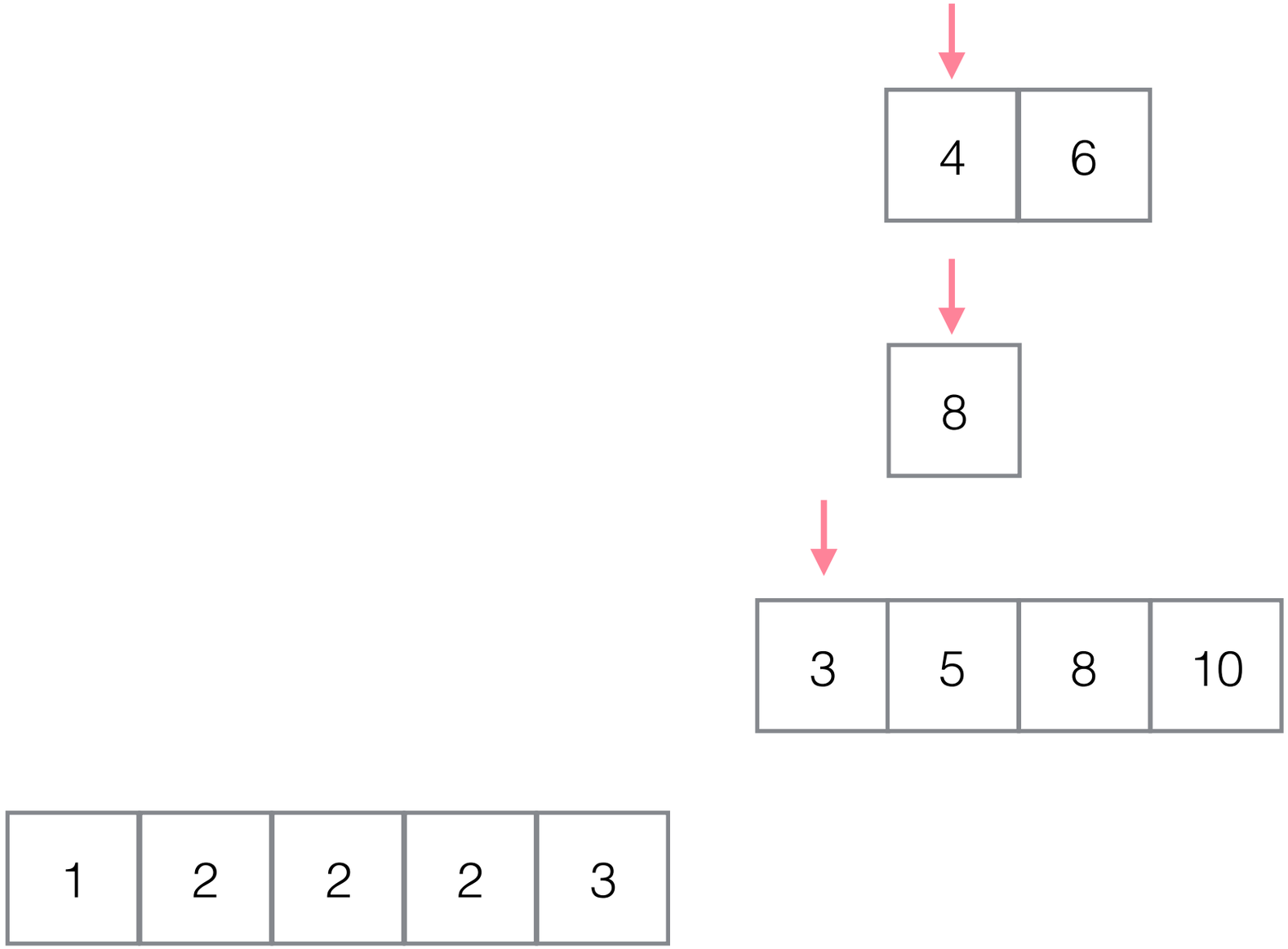
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



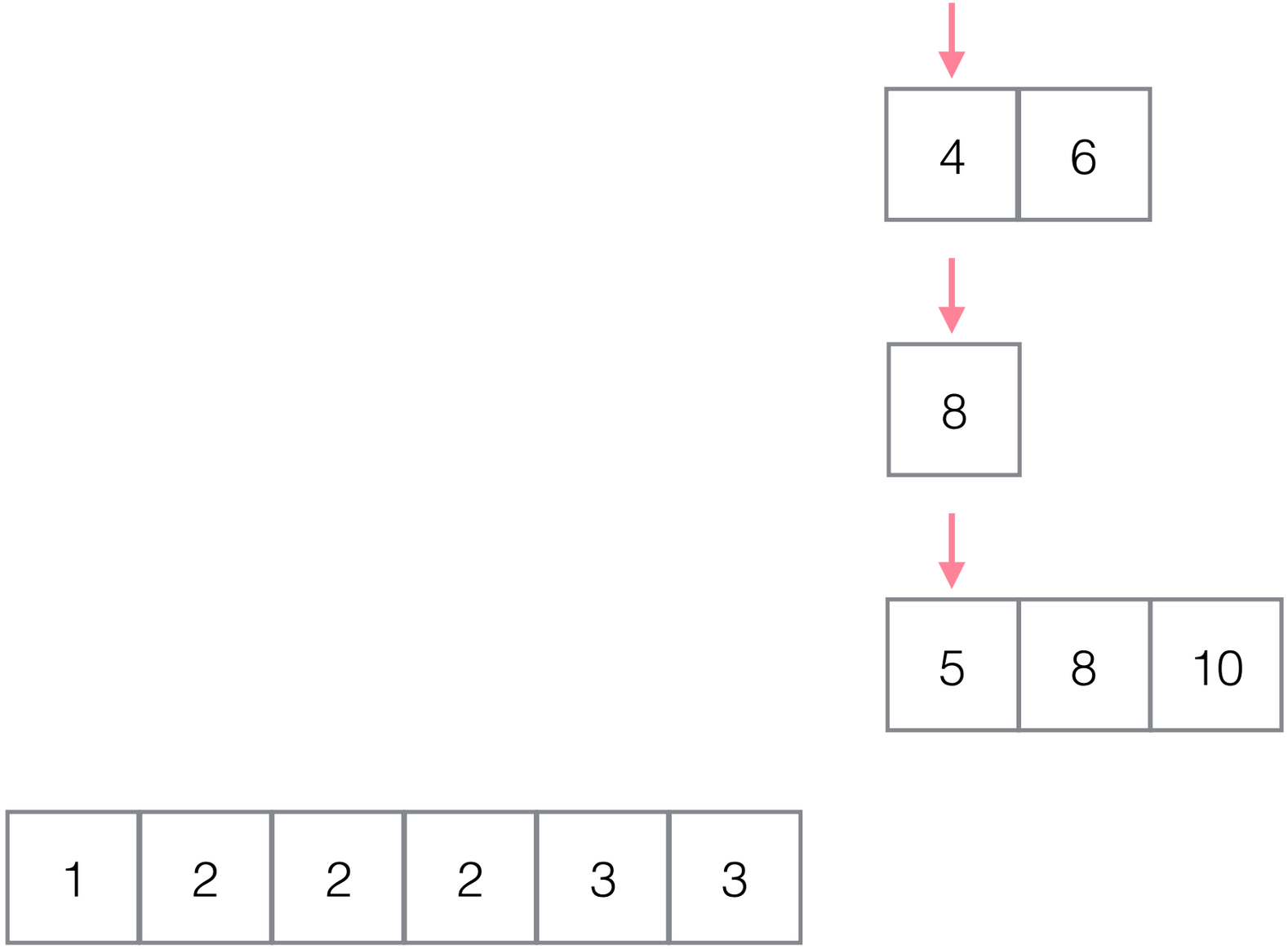
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



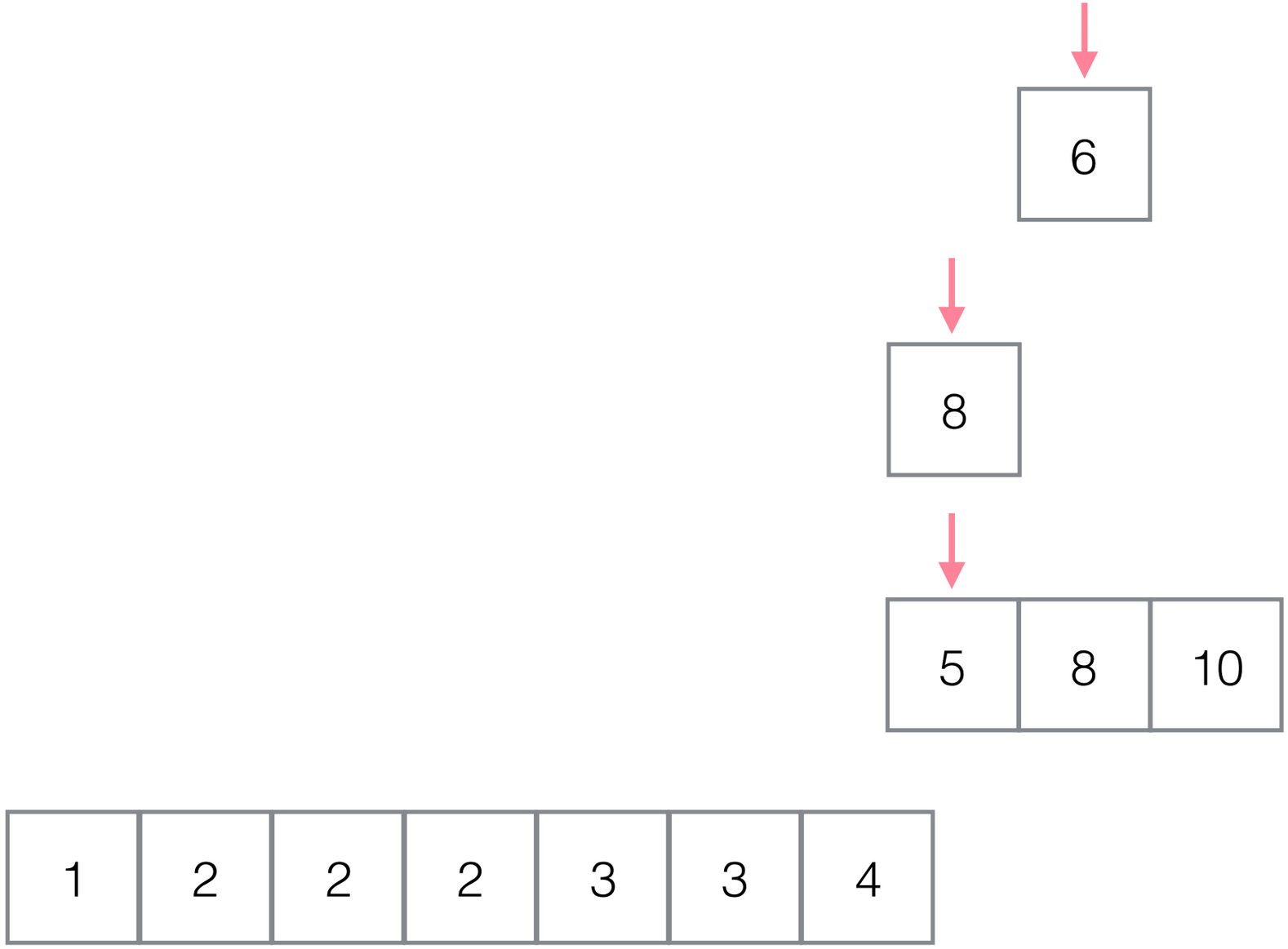
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



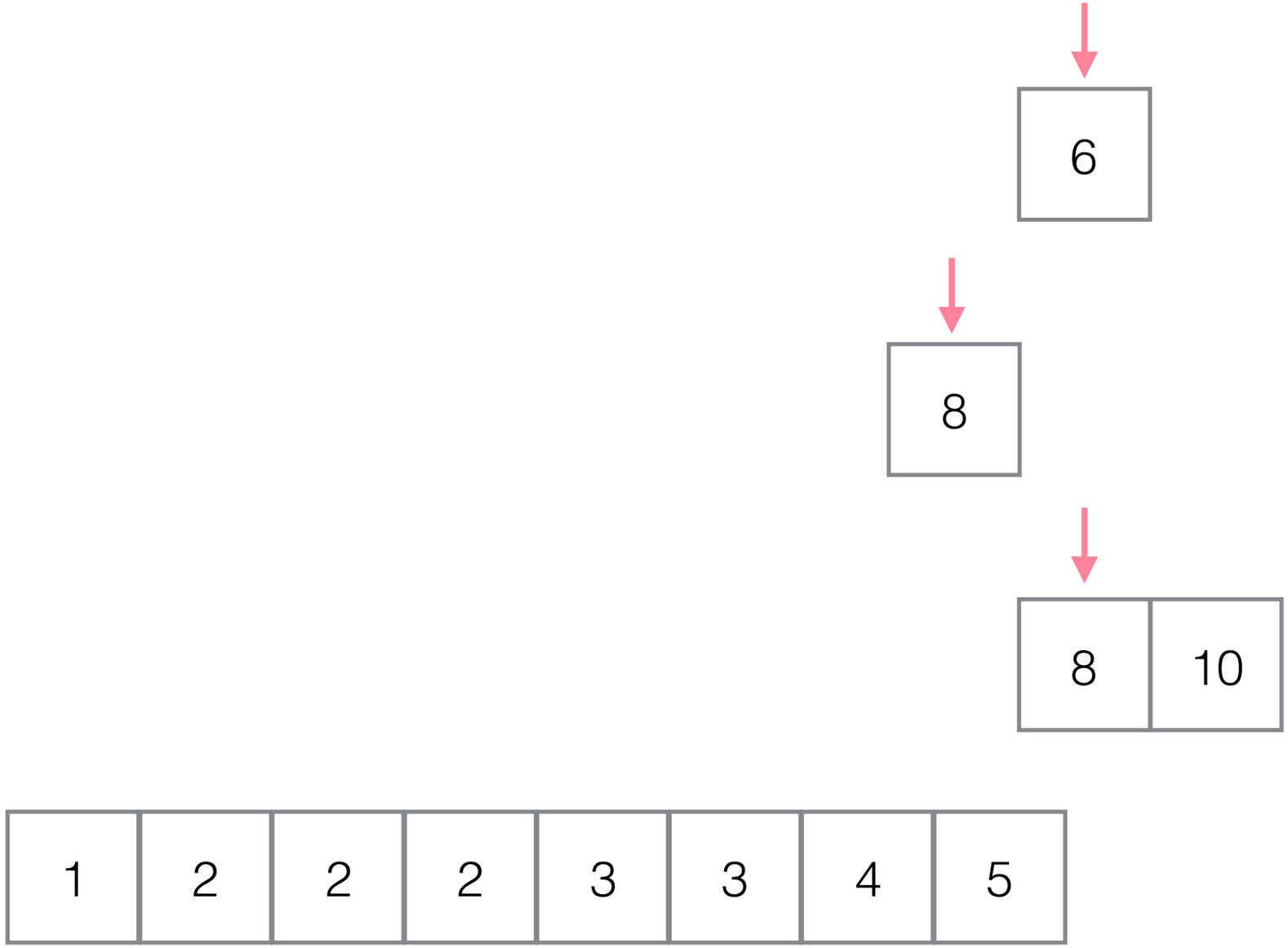
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



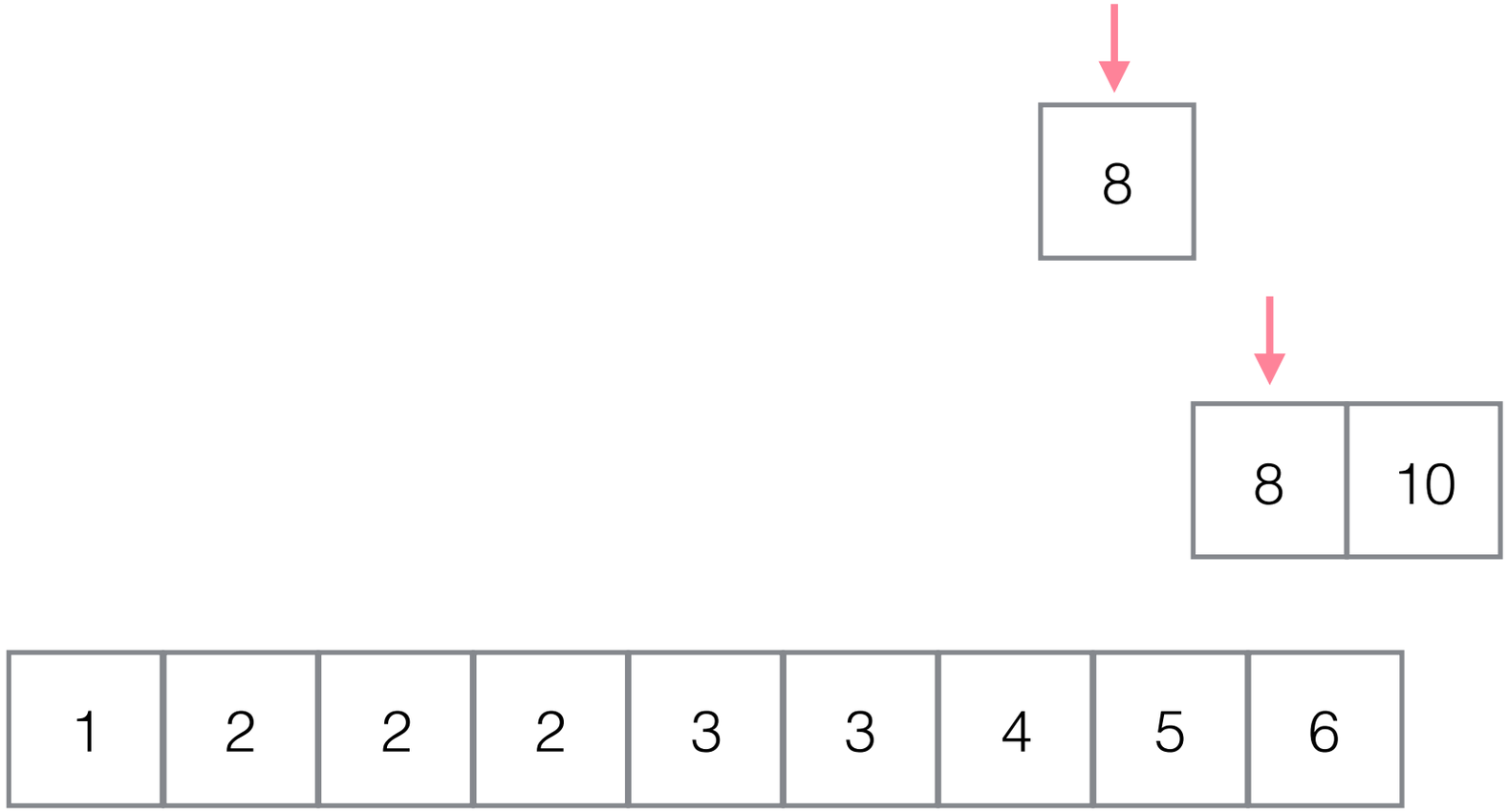
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



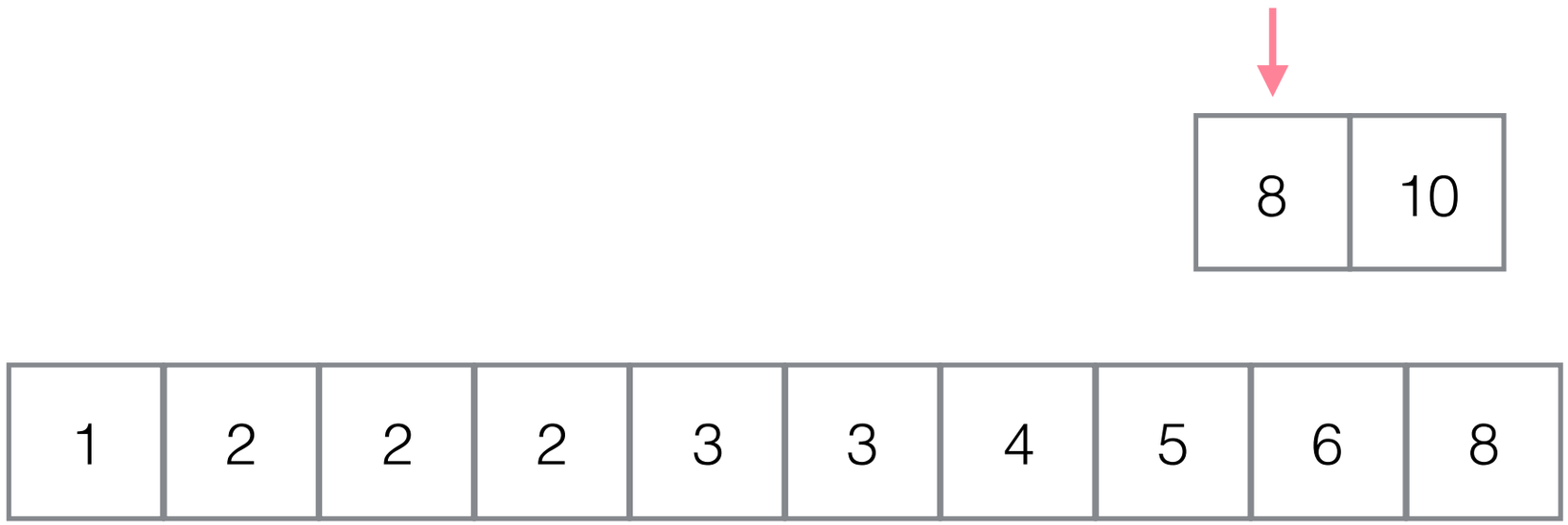
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



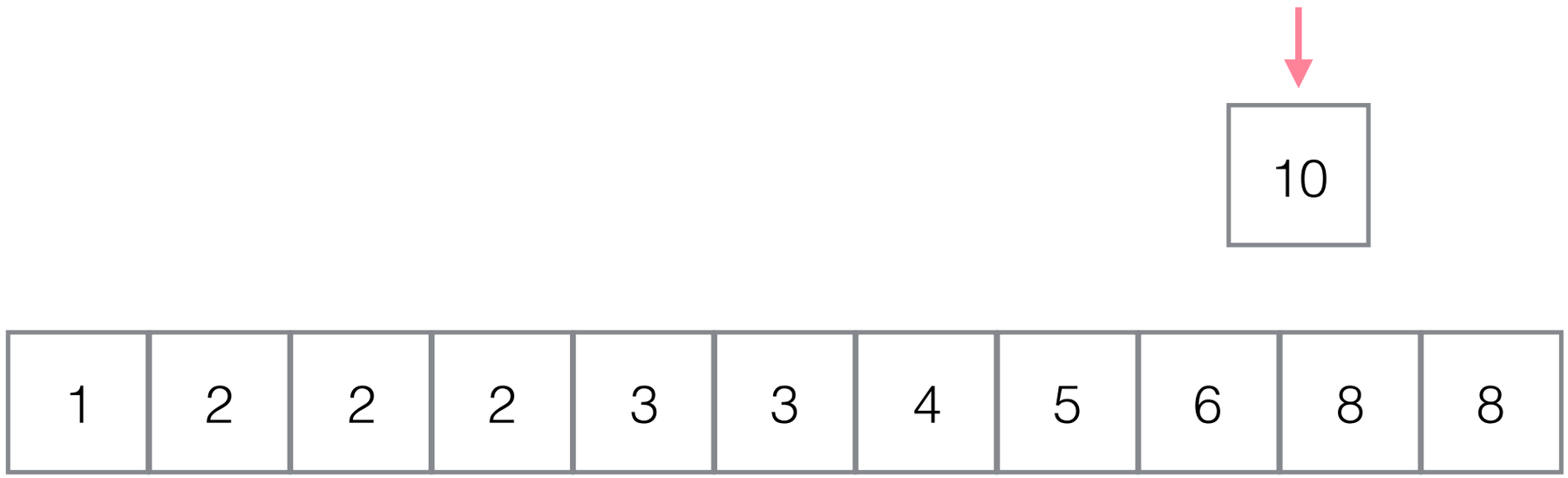
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



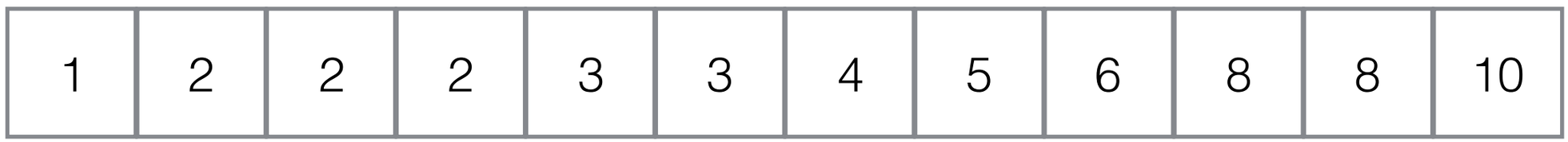
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교



[예제 1] 숫자 합병

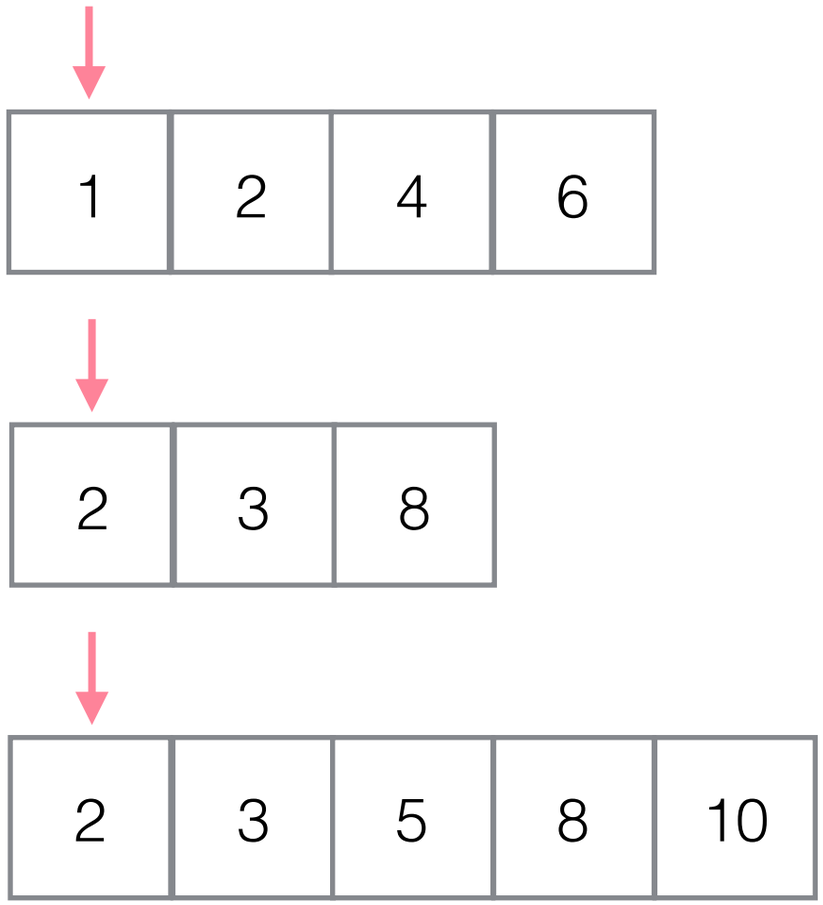
K개의 큐를 만들어, 가장 앞의 값을 비교



[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

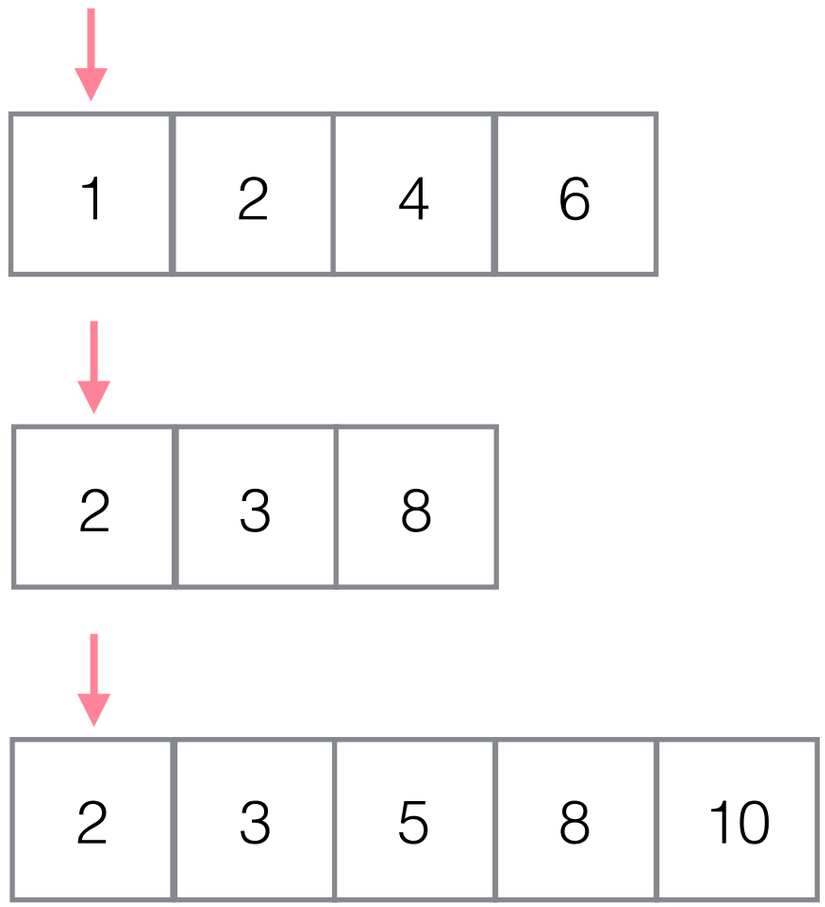


[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 찾자



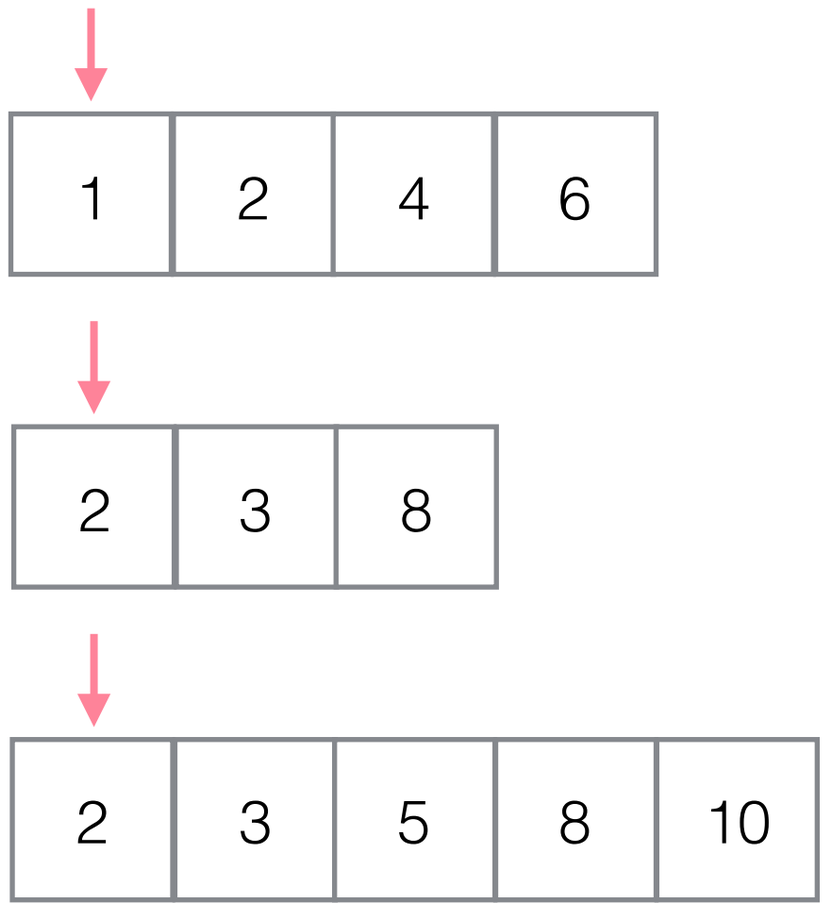
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 찾자

O(kn)

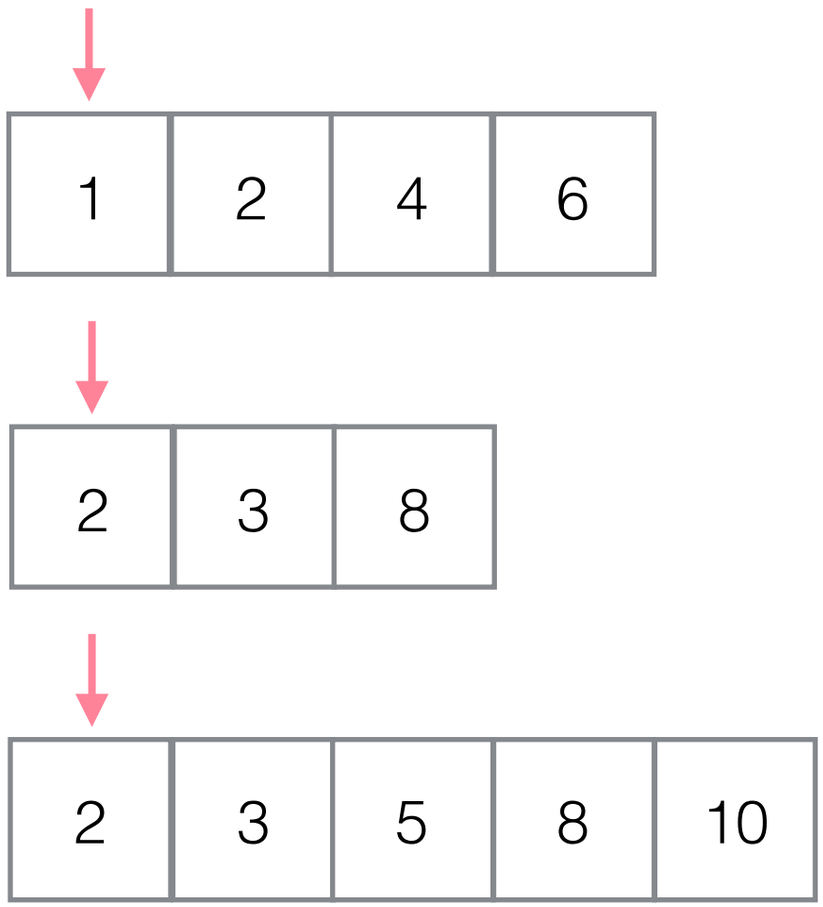


[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 **잘** 찾자

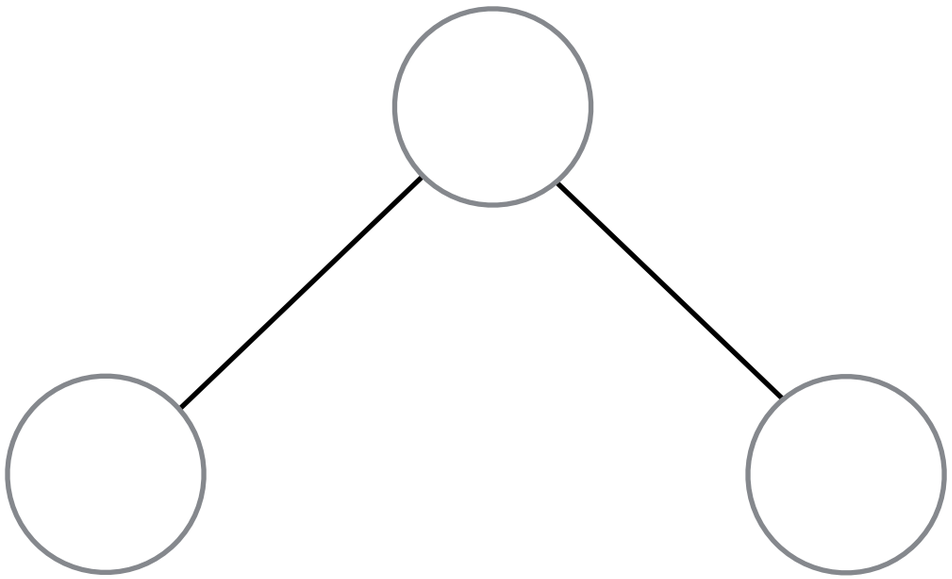


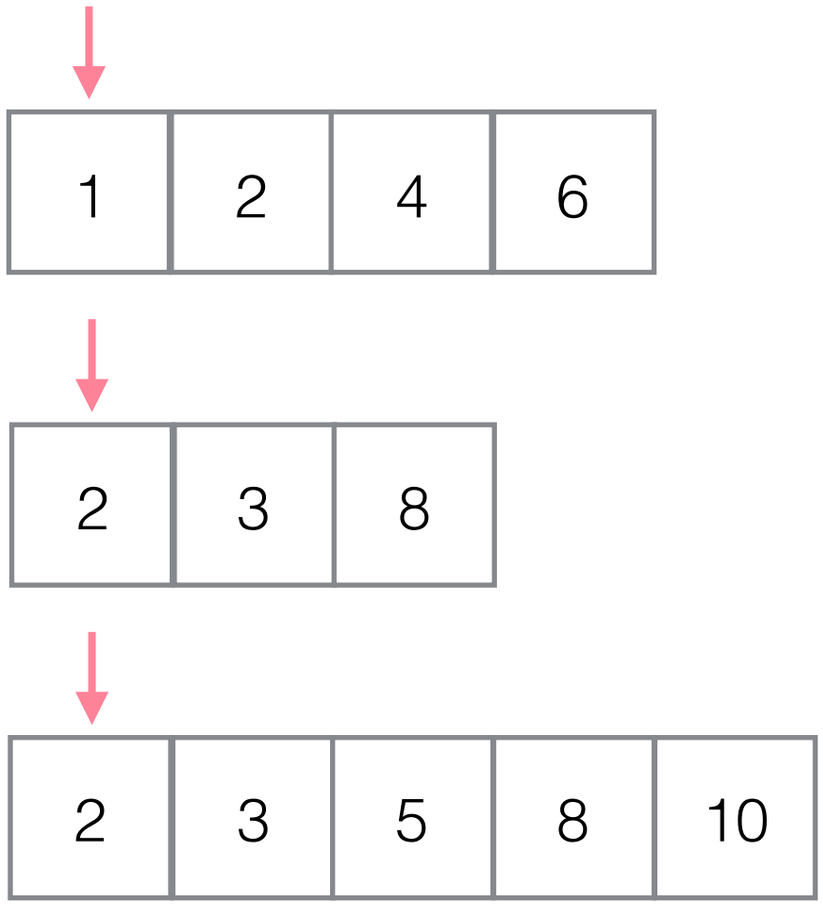
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 **잘** 찾자



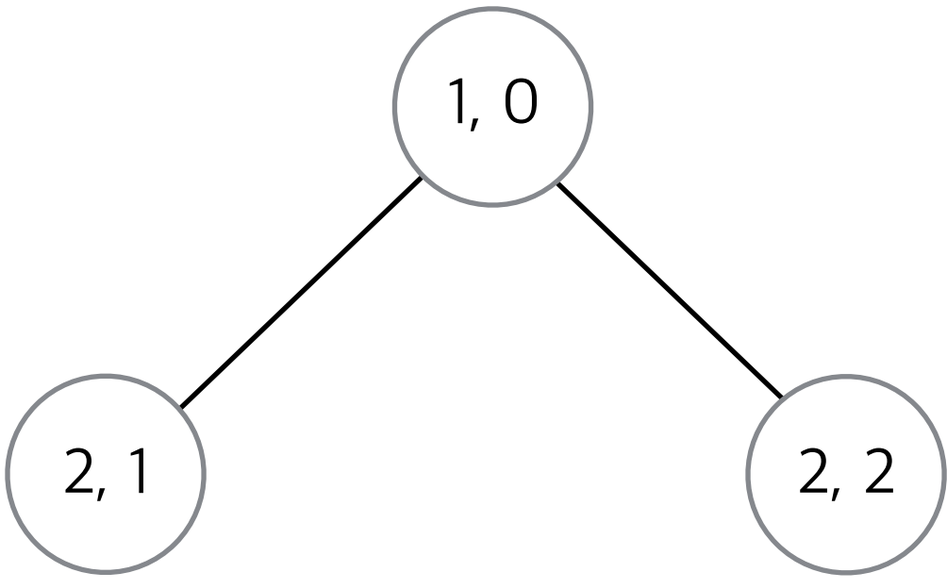


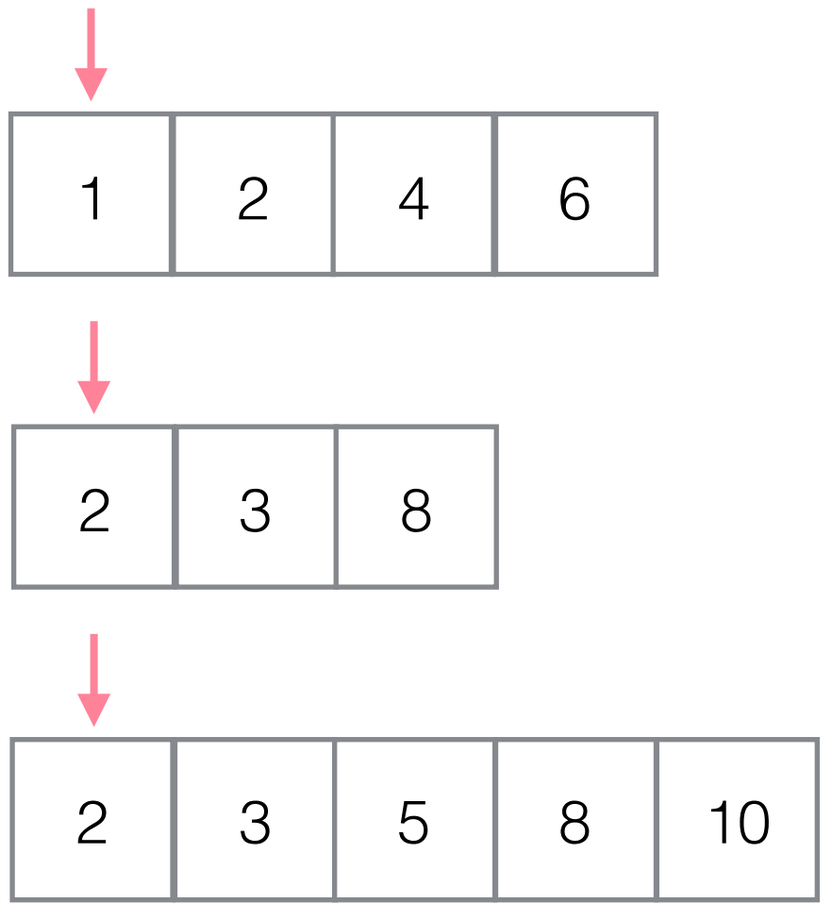
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 **잘** 찾자

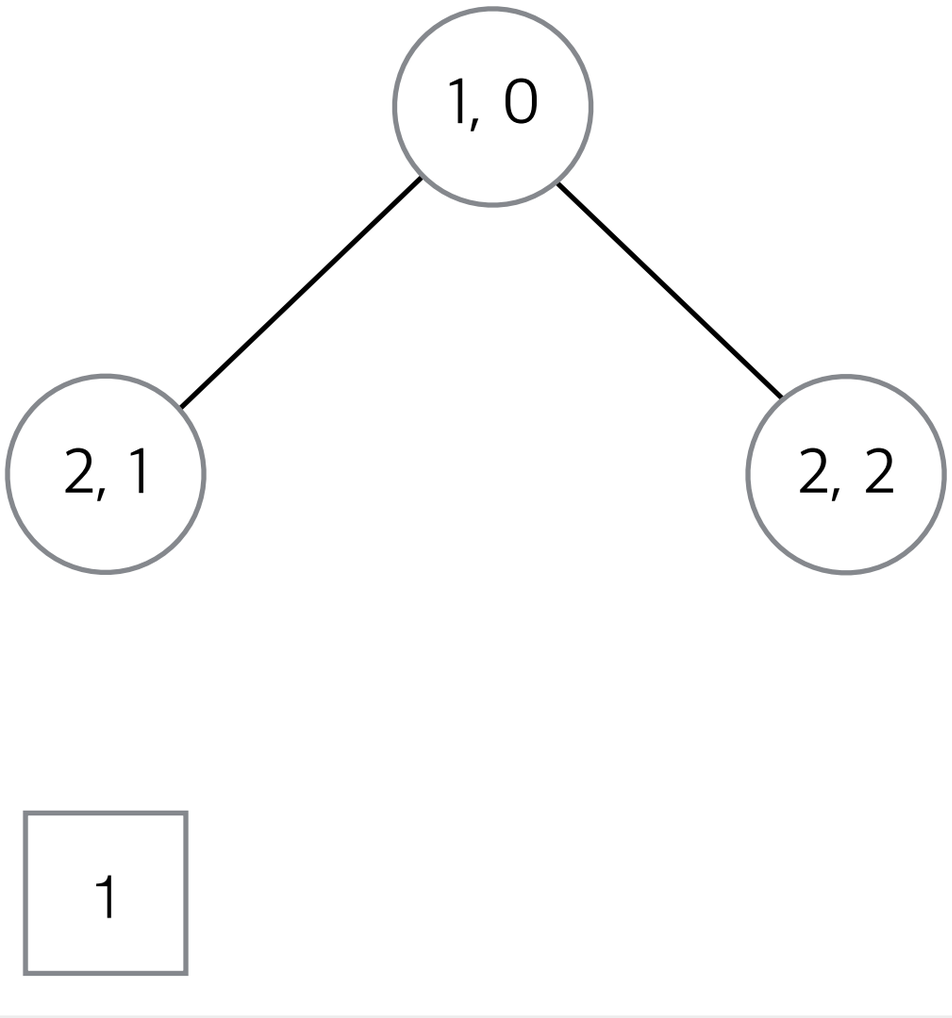




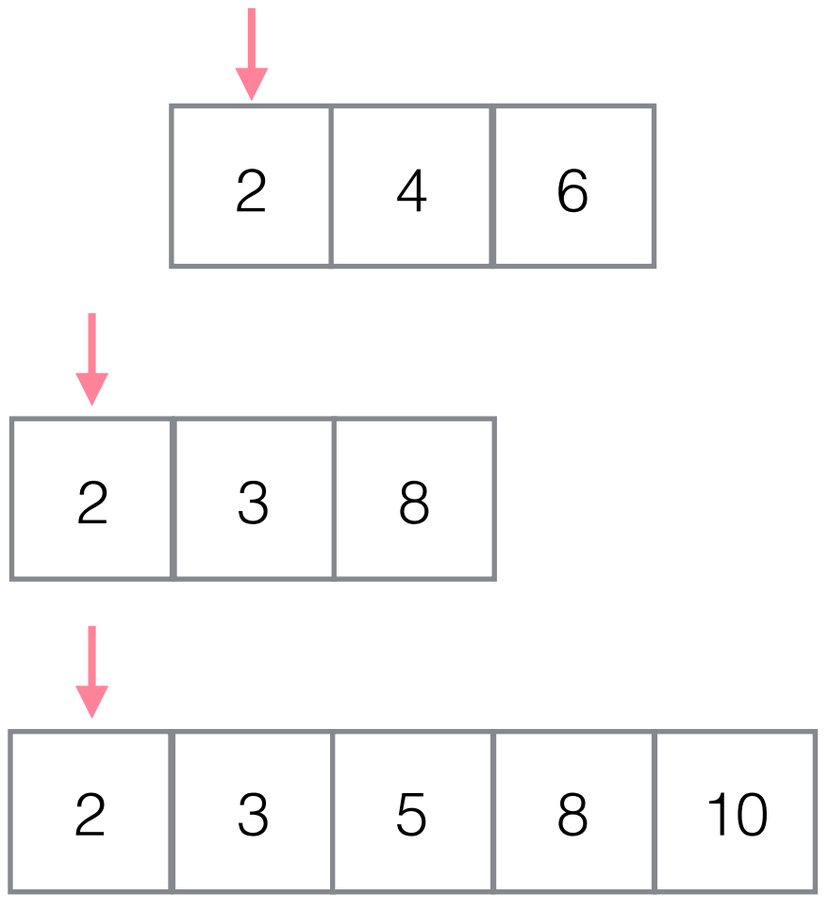
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?



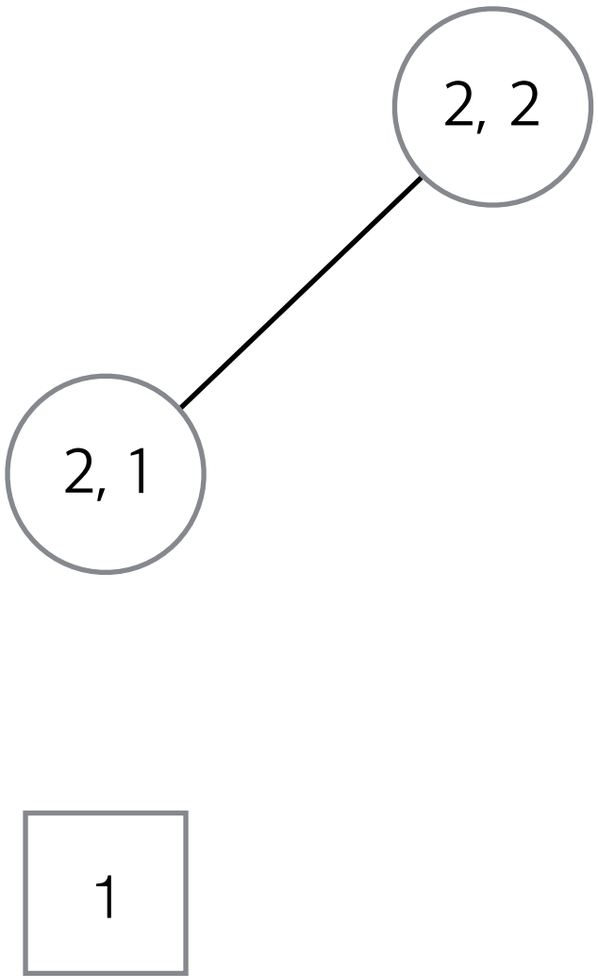
최솟값을 **잘** 찾자



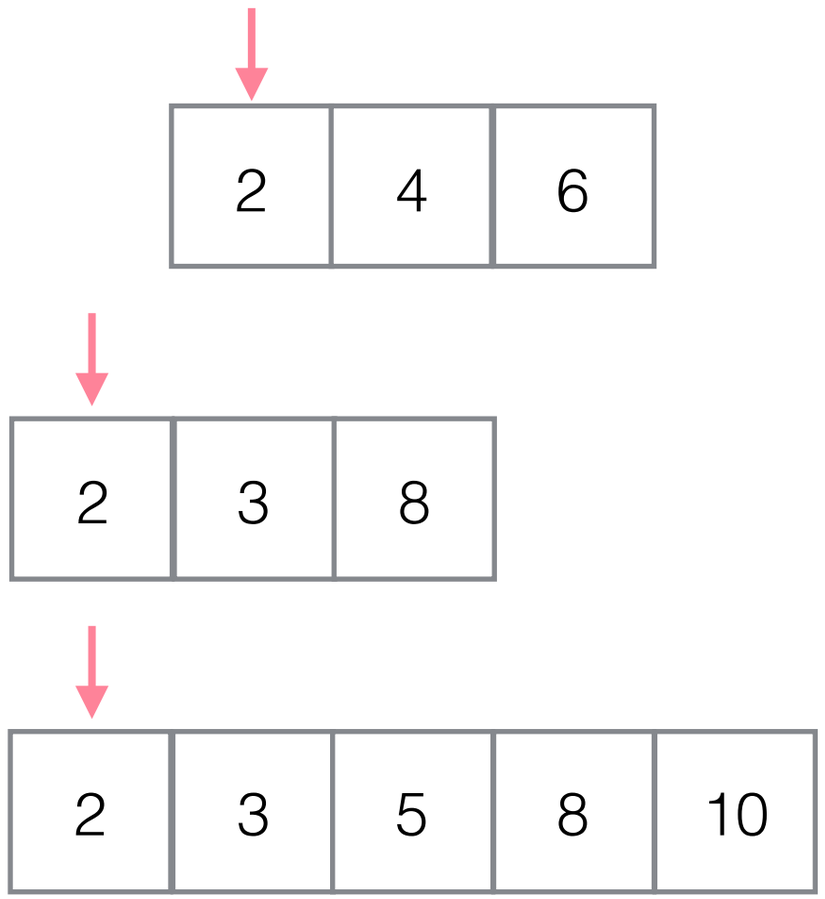
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?



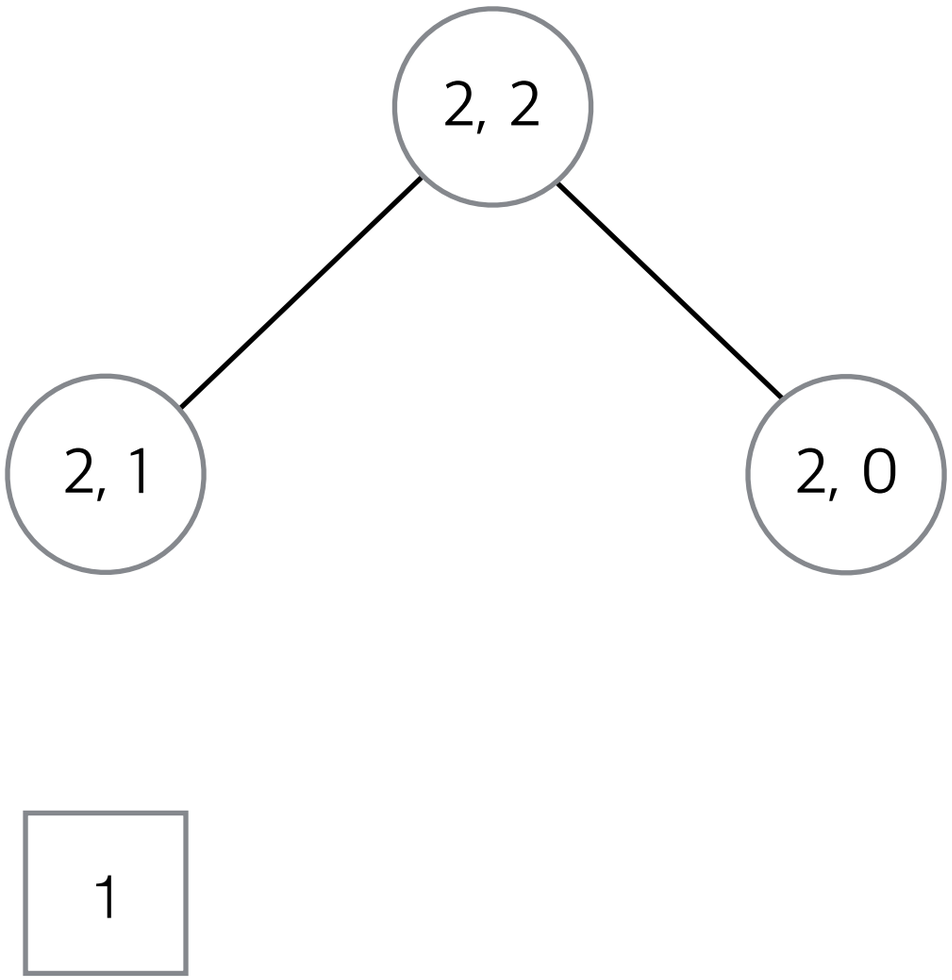
최솟값을 **잘** 찾자



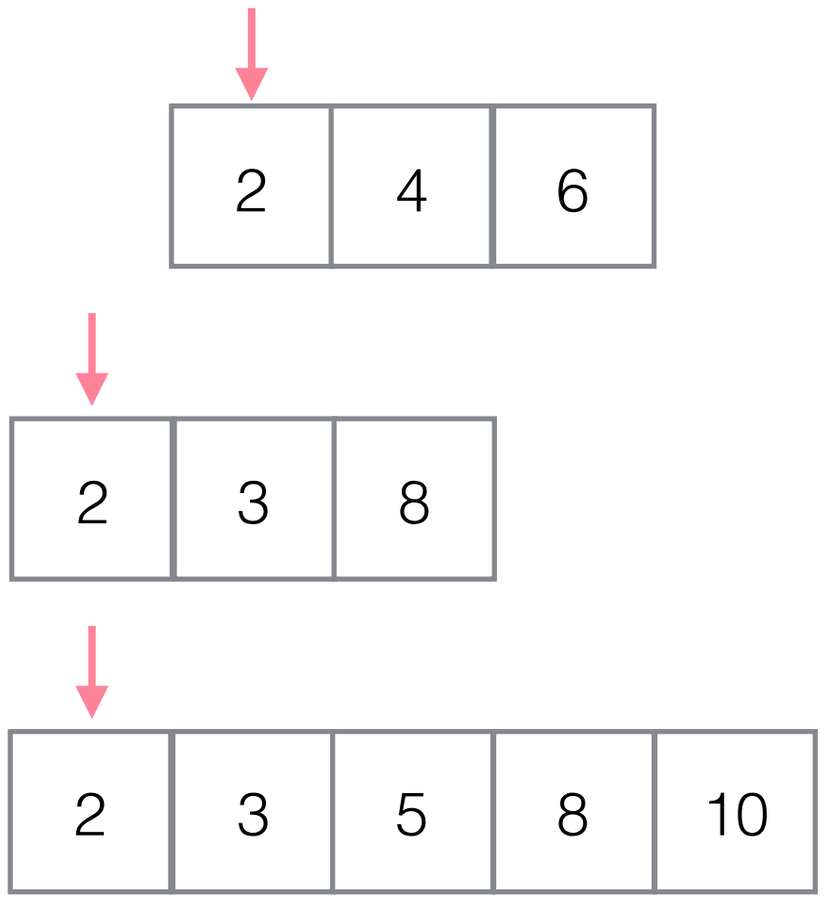
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?



최솟값을 **잘** 찾자

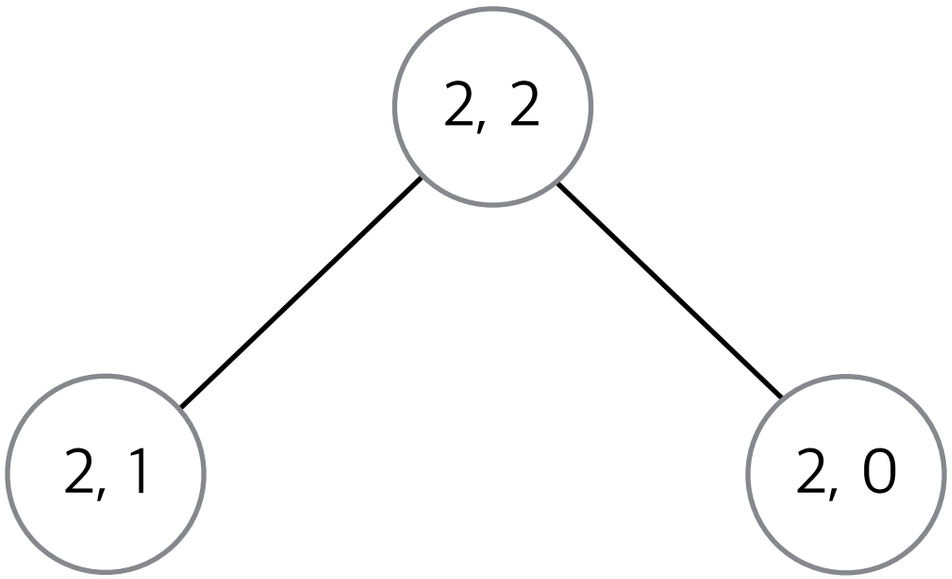


[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 **잘** 찾자

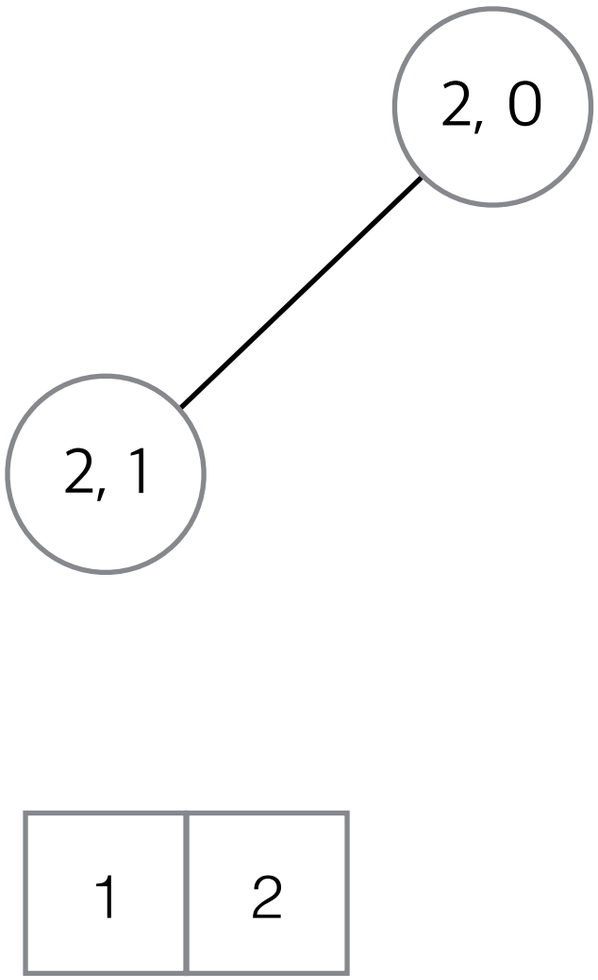


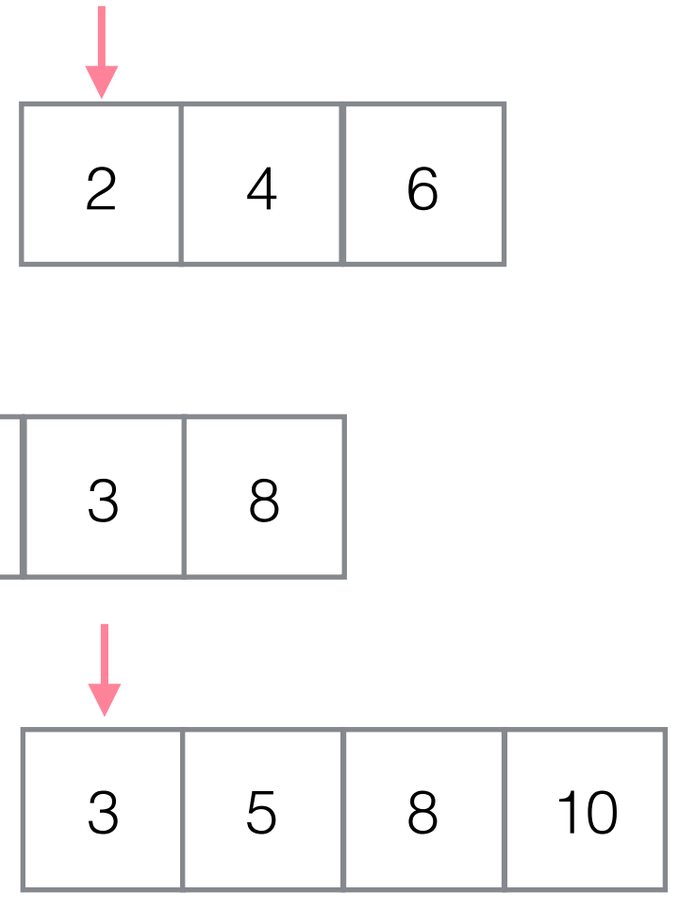
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 **잘** 찾자



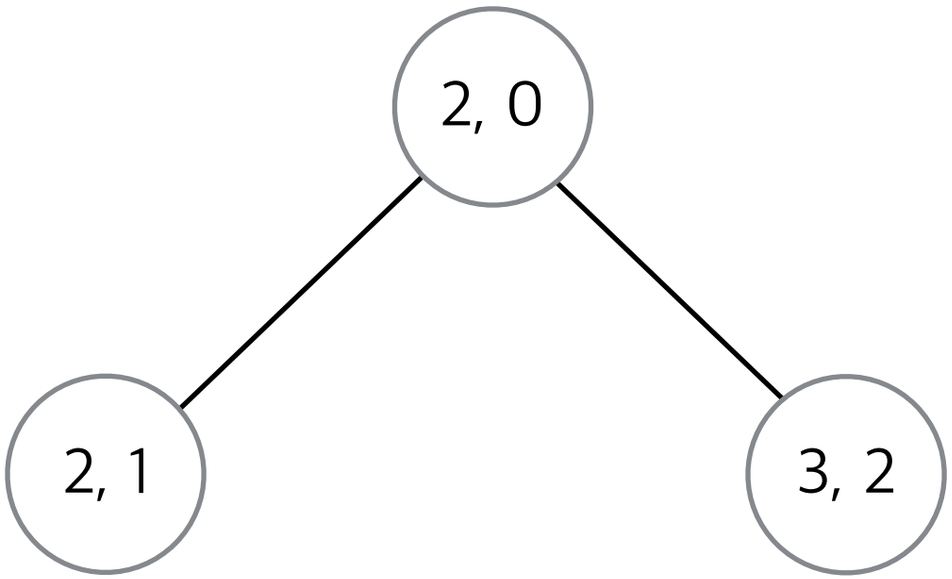


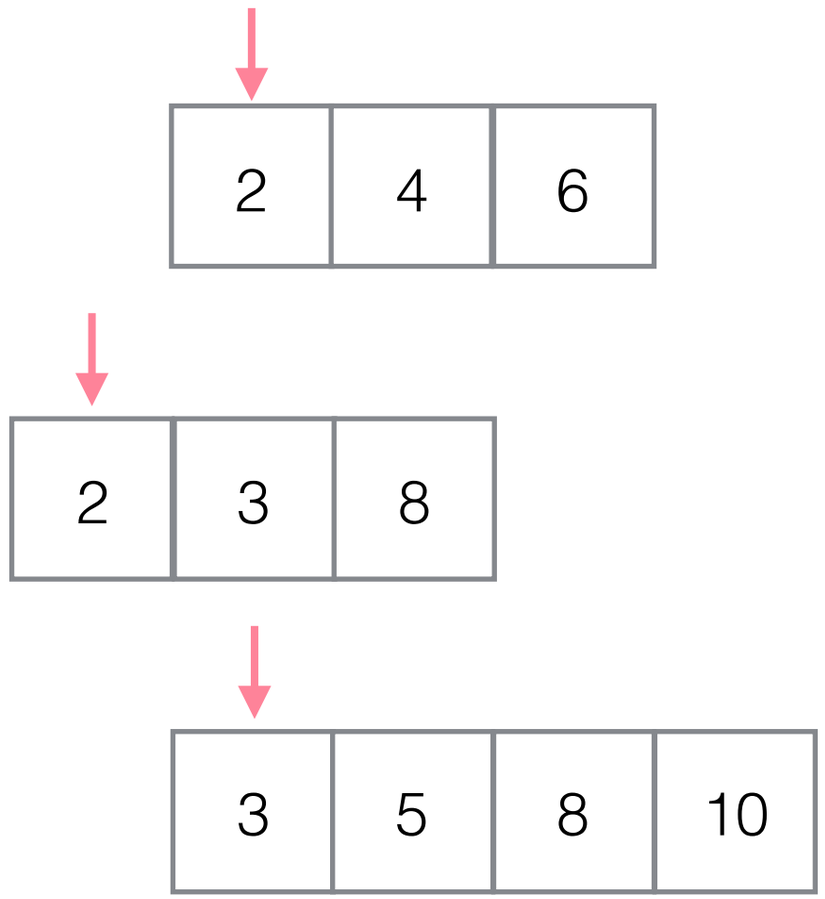
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 **잘** 찾자



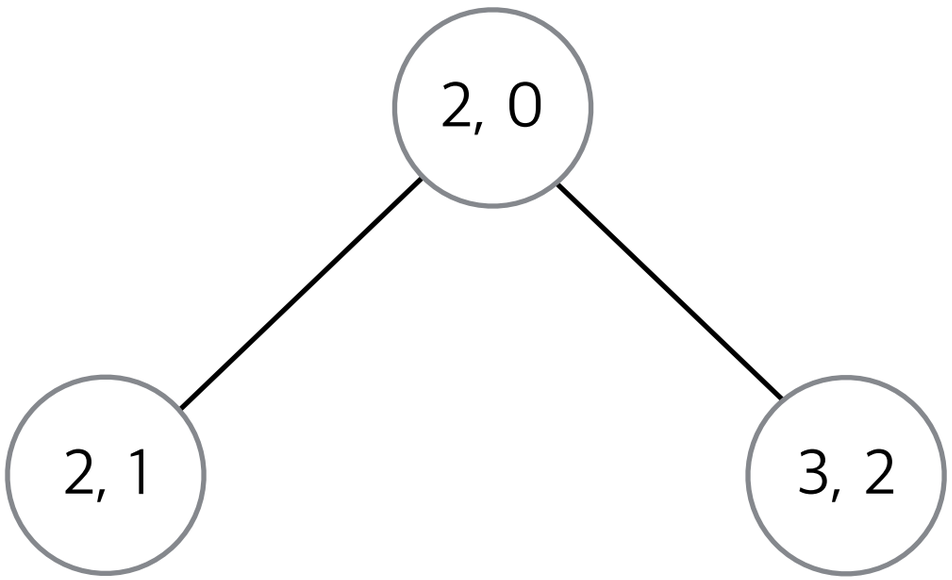


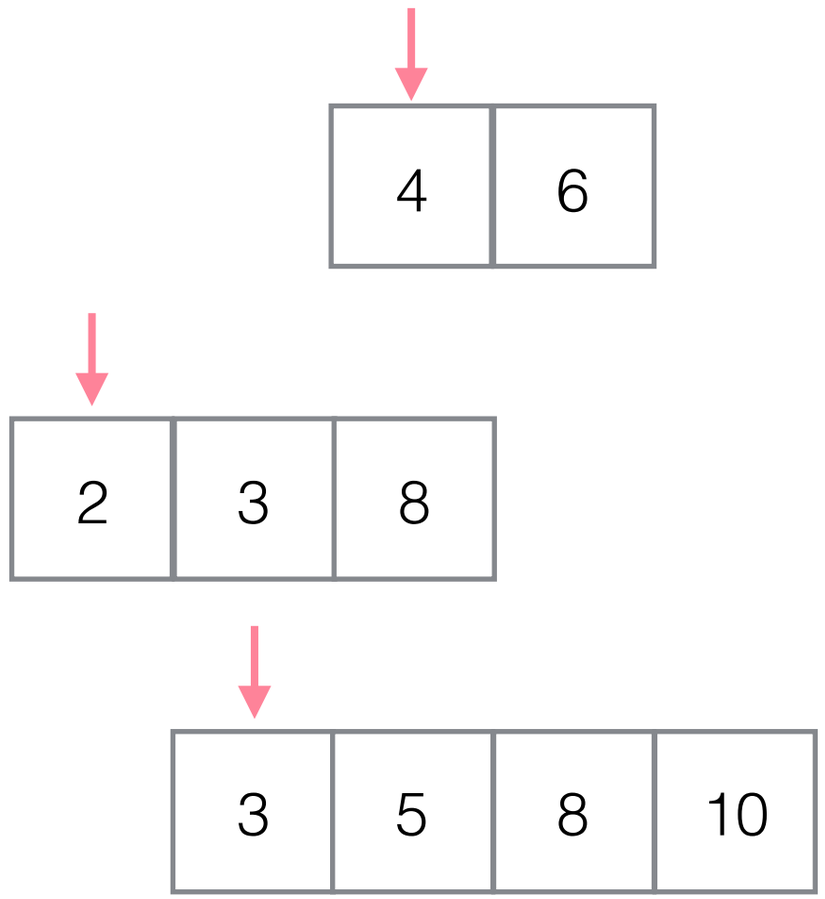
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 **잘** 찾자



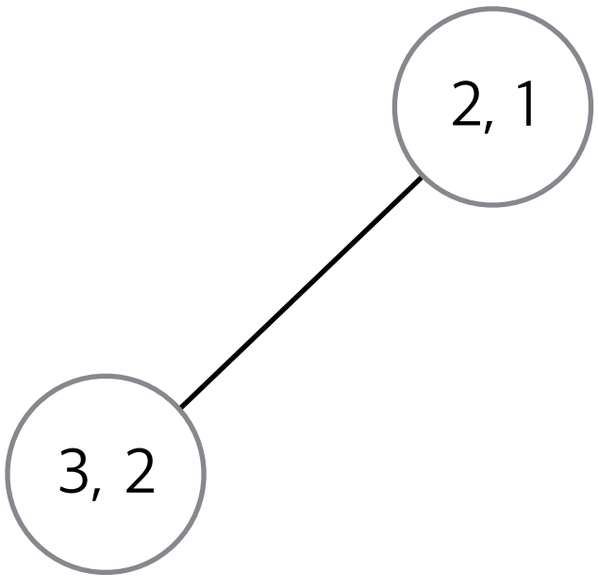


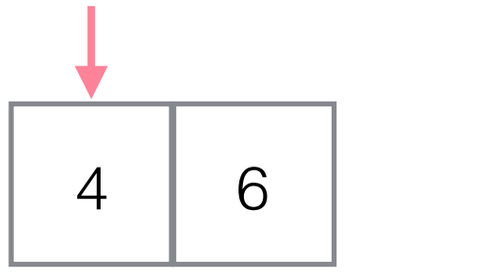
[예제 1] 숫자 합병

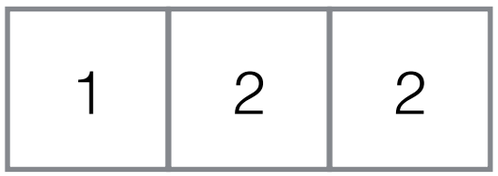
K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 **잘** 찾자





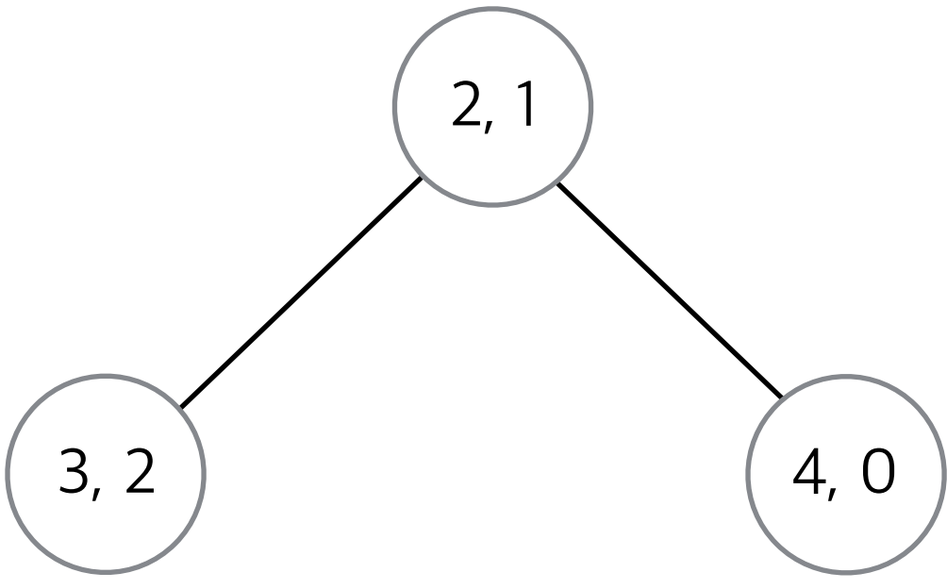


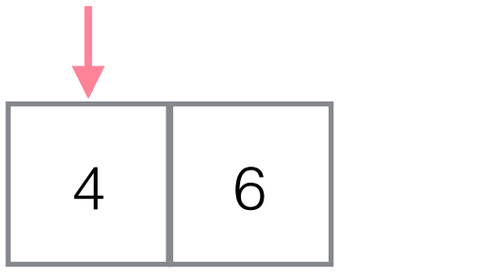
[예제 1] 숫자 합병

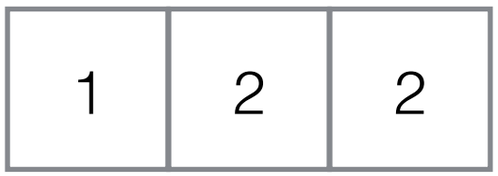
K개의 큐를 만들어, 가장 앞의 값을 비교

어떻게?

최솟값을 **잘** 찾자





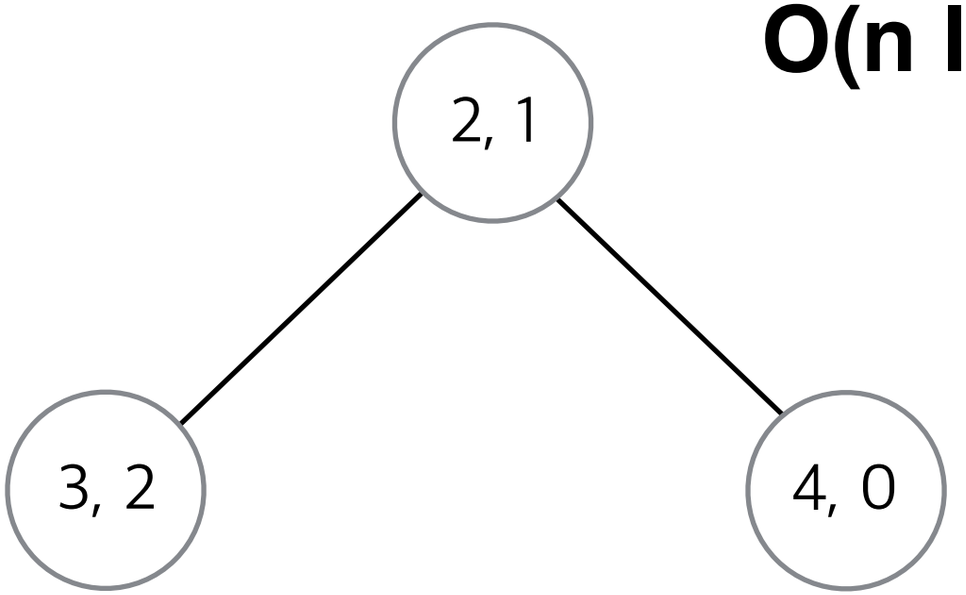


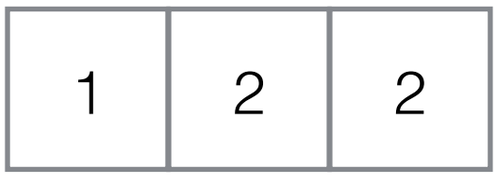
[예제 1] 숫자 합병

K개의 큐를 만들어, 가장 앞의 값을 비교

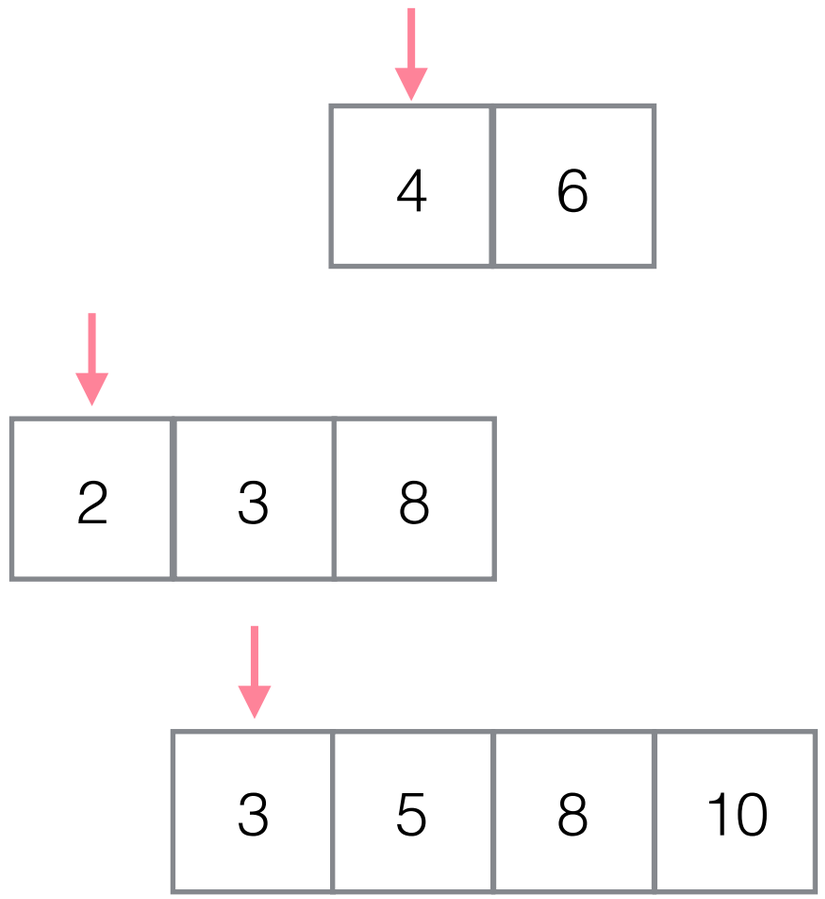
어떻게?

O(n log k)

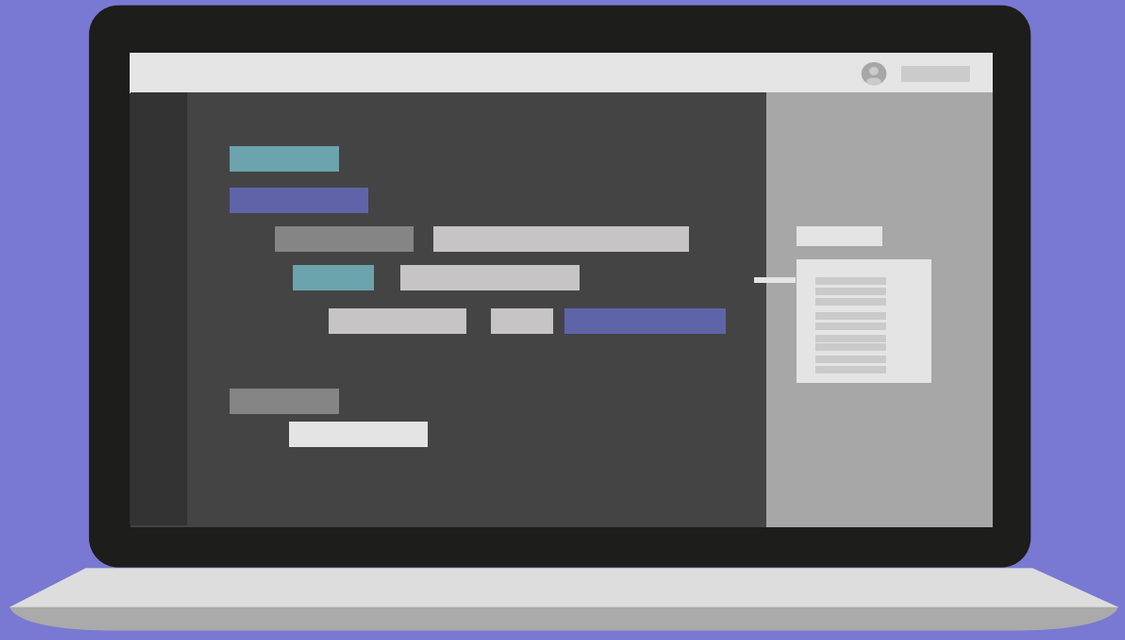




최솟값을 **잘** 찾자



[예제 1] 숫자 합병



이론과 현실의 차이

이론적으로 빠른 것과 현실적으로 빠른 것은 다르다

컴퓨터 내부는 매우 복잡하게 움직임

알고리즘도 좋지만, 컴퓨터 자체의 이해도 필요

시스템 프로그래밍, 컴퓨터 구조

요약

트리는 직접 구현할 일이 거의 없다

그래프의 구현을 배우면 트리 구현을 자연스럽게 습득

목적이 확실한 자료구조가 더 익히기 쉽다

힙은 스택/큐/트리처럼 심오한 의미가 있진 않다

트리 문제라도 트리를 구현할 필요가 없을 수 있다

굳이 트리를 저장하지 않아도 되는 경우가 있다

감사합니다!

신현규

E-mail : hyungyu.sh@kaist.ac.kr

Kakao : yougatup

/* elice */

문의 및 연락처

academy.elice.io

contact@elice.io

facebook.com/elice.io

blog.naver.com/elicer