

# 7장. 우선순위 큐: 힙



## ■ 주요 내용

- 01 힙이란
- 02 힙 작업 알고리즘과 구현
- 03 힙 수행 시간

## ■ 학습목표

- 우선순위 큐의 의미를 이해한다.
- 우선순위 큐의 추상적 구조를 이해한다.
- 우선순위 큐의 대표 케이스로 힙의 객체 구조를 이해한다.
- 힙 작업 알고리즘들의 원리를 이해한다.
- 힙의 파이썬 구현을 연습한다.

# 01 힙이란

# 정적Static vs. 동적Dynamic 데이터 집합

## ■ 정적Static 데이터 집합

- 한번 구축되고 나면 변하지 않음

## ■ 동적Dynamic 데이터 집합

- 데이터가 계속 변함

- Dictionary (Table)

- 삽입, 삭제, 검색을 지원하는 동적 데이터 집합을 지칭

- 배열, 리스트, 검색 트리, 해시 테이블, ...  
(inefficient)

- 우선순위 큐Priority queue

- 삽입, 최우선 원소 삭제, 최우선 원소 검색을 지원하는 동적 데이터 집합

- 배열, 리스트, 검색 트리, 힙, ...  
(inefficient)

# 비교

---

## ■ 삭제

- Table은 삭제할 원소 제공
- 우선순위 큐는 삭제할 원소 불필요 (자동 결정)
- 우선순위가 가장 높은 원소만 삭제 가능

## ■ 삽입

- Table과 우선순위 큐 둘 다 삽입할 원소 제공함

## ■ 원소 값 중복

- Table은 불허
- 우선순위 큐는 허용

# ADT 우선순위 큐 Priority Queue

최우선 순위는 최대 원소 또는 최소 원소 중 한 쪽  
둘은 대칭적  
여기서는 최대 원소를 최우선 원소로 가정

원소를 삽입한다

최대 원소를 알려주면서 삭제한다

최대 원소를 알려준다

우선순위 큐가 비어 있는지 확인한다

우선순위 큐를 깨끗이 비운다

그림 8-1 ADT 우선순위 큐

주목할 사실:

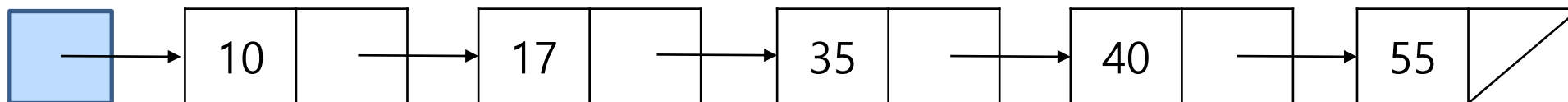
우선순위 큐에서 유일하게 접근 가능한 원소는 최대 원소뿐

# 비효율적인 우선순위 큐

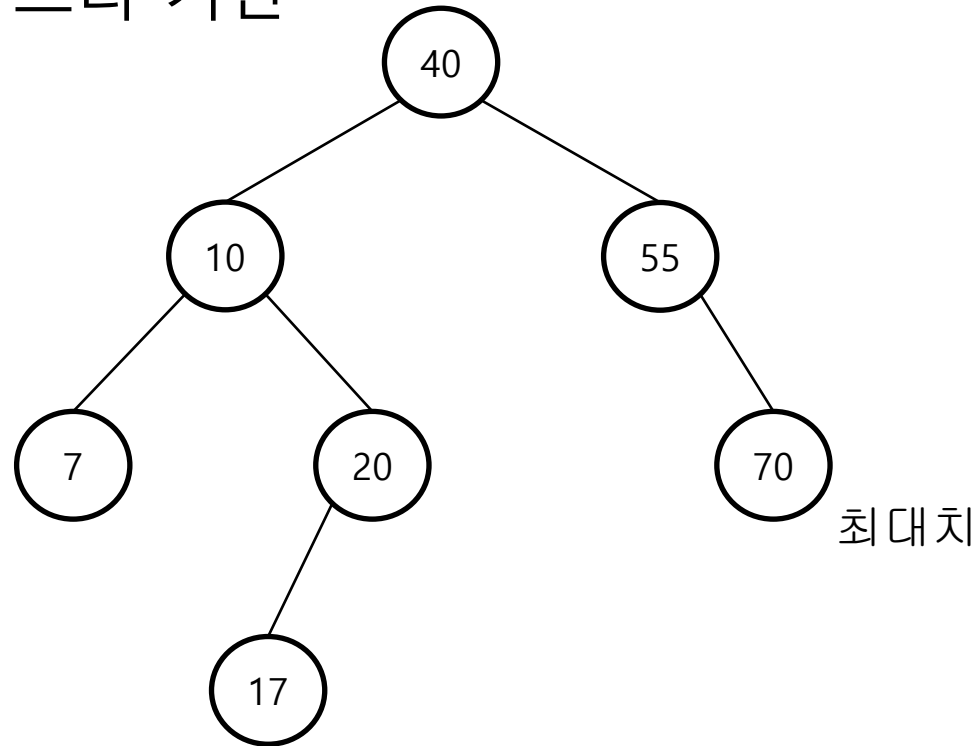
배열 리스트

10	35	40	17	95	50	48	33	9			
----	----	----	----	----	----	----	----	---	--	--	--

연결 리스트



## 이진 검색 트리 기반

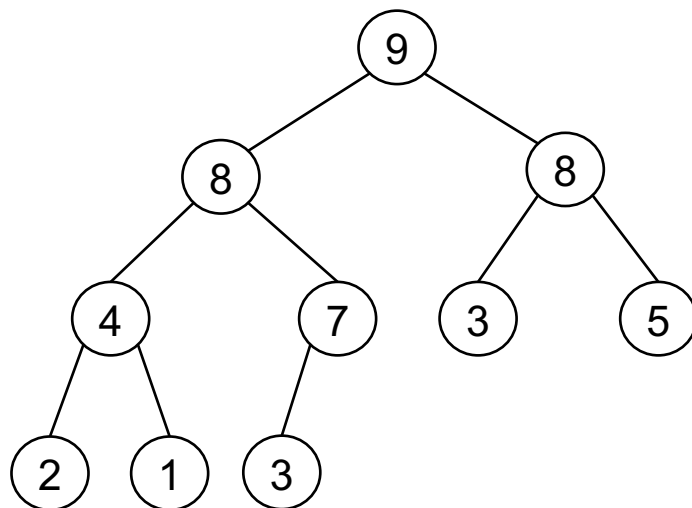


이진 검색 트리(10장)도 우선순위 큐로 부적당  
동일한 key가 2개 이상일 때 별도로 처리를 해주어야 한다  
이게 아니라도 우선순위 큐 용도로는 너무 과하다



# 힉 Heap: 대표적인 우선순위 큐

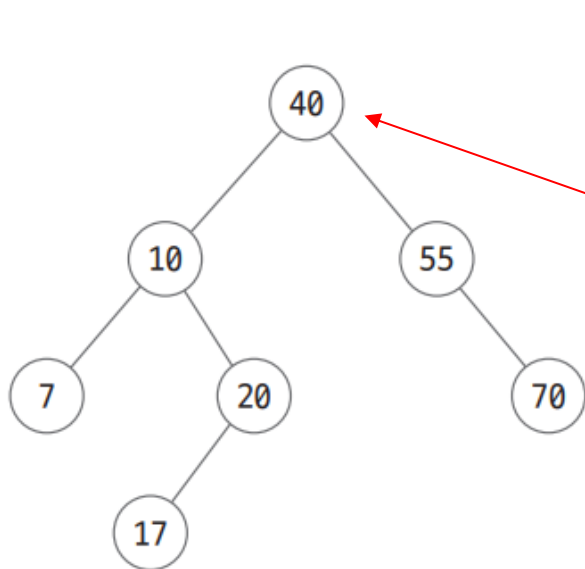
- 우선순위를 관리하기 위한 특수한 성질을 가진 자료구조



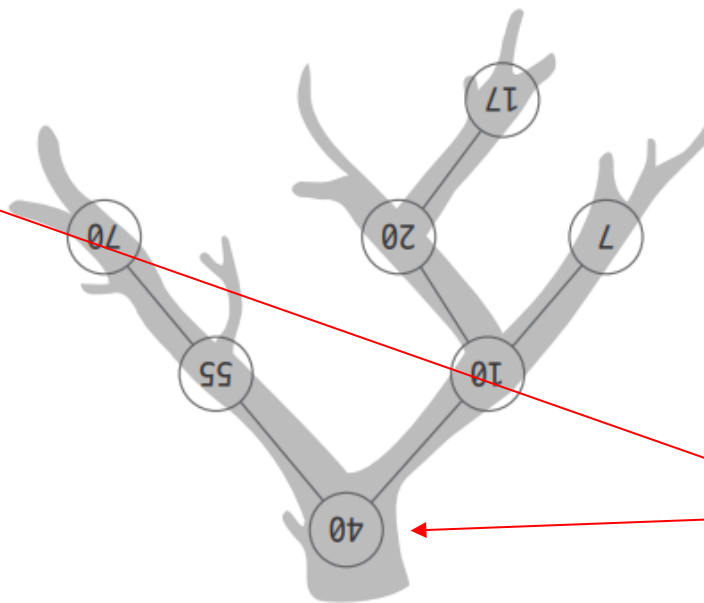
# 힉 Heap을 정의하기 전에..

## 이진 트리 Binary Tree

노드 중 하나가 루트  
각 노드는 최대 2개까지 자식 Child을 가질 수 있다



(a) 이진 트리의 예

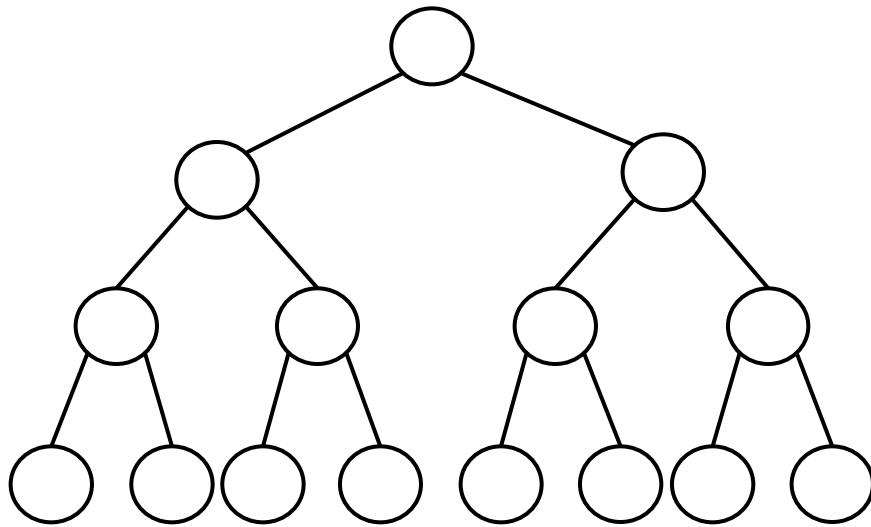


(b) (a)를 뒤집은 형태

뿌리 Root라 부른다

## 포화 이진 트리 Full Binary Tree

루트로부터 시작해서 모든 노드가 정확히 두 개씩의 자식 노드를 가지도록 꽉 채워진 트리



노드 수가  $2^k - 1$ 일 때만 가능  
0, 1, 3, 7, 15, 31, ...

그림 8-3 (a) 포화 이진 트리의 예

## 완전 이진 트리 Complete Binary Tree

루트로부터 시작해서 가능한 지점까지  
모든 노드가 정확히 두 개씩의 자식 노드를 가진다

노드의 수가 맞지 않아 **full binary tree**를 만들 수 없으면 맨 마지막 레벨은 왼쪽부터 채워나간다

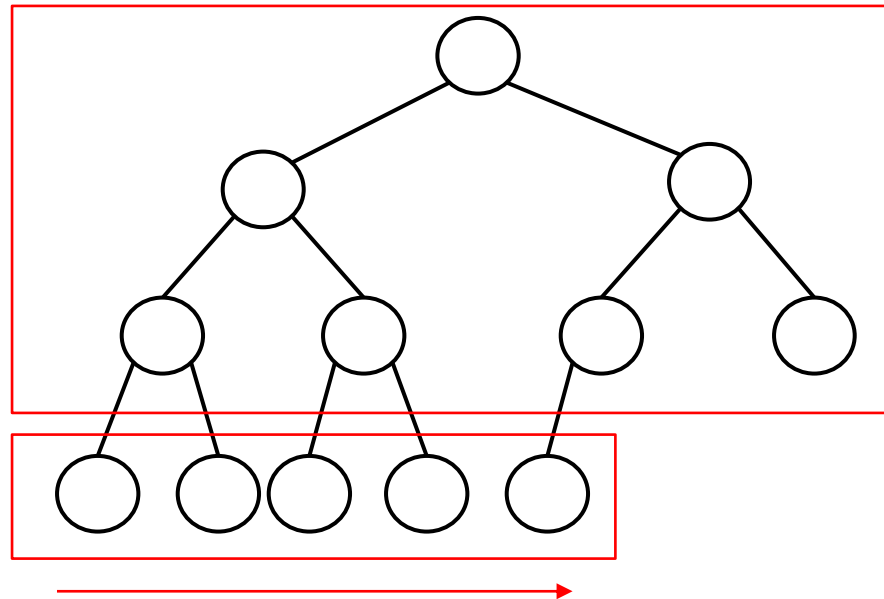


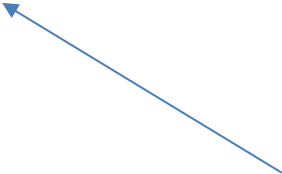
그림 8-3 (b) 완전 이진 트리의 예

# 힙: 대표적인 우선순위 큐

## ■ 힙은 다음 두 조건을 만족해야 한다

1. 완전 이진 트리
2. 힙 특성<sup>Heap Property</sup>: 모든 노드는 값을 갖고, 자식 노드(들) 값보다 크거나 같다

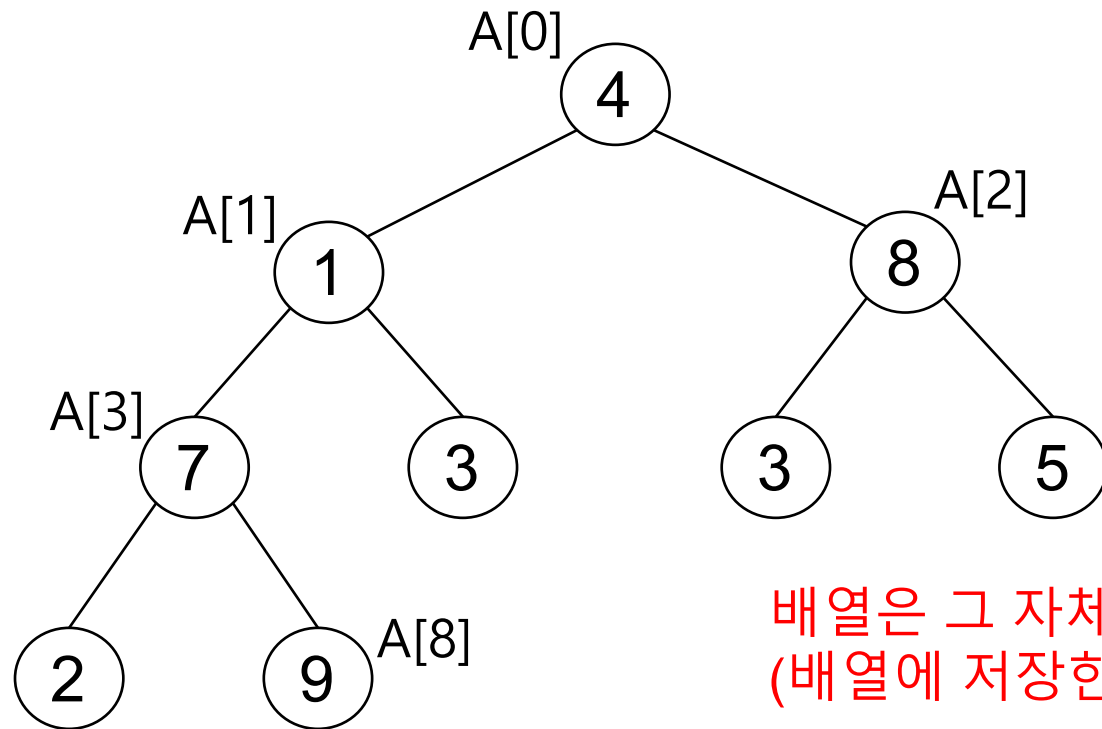
결과적으로, 루트 노드가 제일 큰 원소를 갖게 됨



## ■ 최대 힙<sup>Maxheap</sup> : 최소 힙<sup>Minheap</sup>

- 루트가 최대값:최소값을 가짐
- 둘은 대칭적. 하나만 배우면 다른 것은 쉽다.
- 여기서는 최대힙으로

# 힙은 배열과 안성맞춤



A[ ]	4	1	8	7	3	3	5	2	9
------	---	---	---	---	---	---	---	---	---

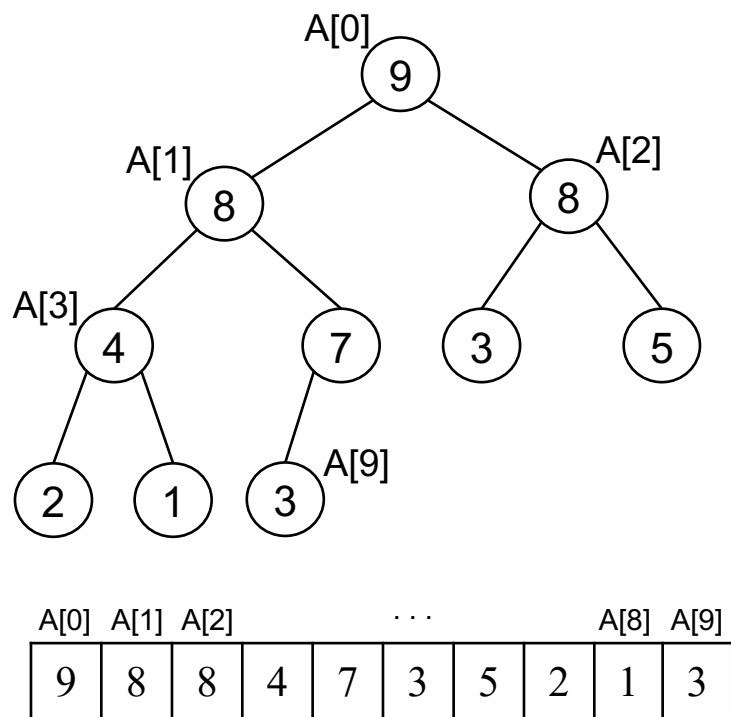
노드  $i$ 의 자식 노드:  $2i+1, 2i+2$

노드  $i$ 의 부모 노드:  $\lfloor (i-1)/2 \rfloor$

배열은 그 자체로 완전 이진 트리로 볼 수 있다  
(배열에 저장한다는 사실로 완전 이진 트리 조건(조건 1)은 자동 만족)

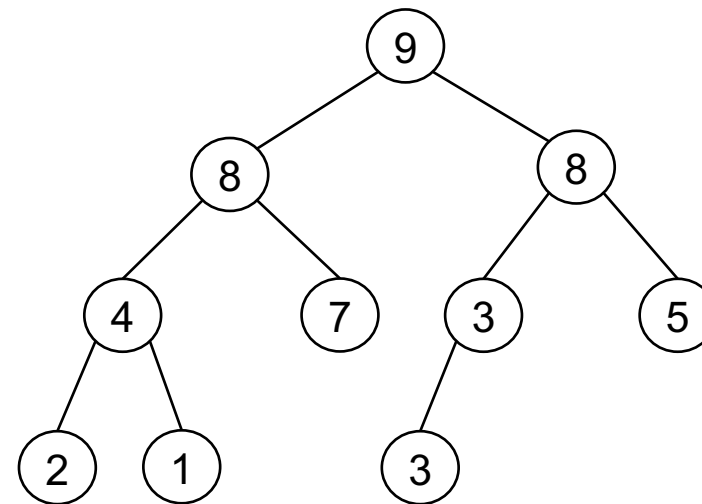
파이썬 내장 리스트도 배열로 간주할 수 있다.  
단, 배열은 현재 원소가 총 몇 개인지 사용자가 관리해야 하지만  
파이썬 내장 리스트는 파이썬에서 자동 관리해준다.

# 힙의 예



10개의 원소로 구성된 힙과  
대응되는 배열

그림 8-4 10개의 원소로 구성된 힙과 대응 리스트



힙특성은 만족하지만  
완전 이진 트리를 만족하지 못하는 예

그림 8-5 힙 조건 1을 만족하지 않아 힙이 아닌 예

# 힙 객체 구조

필드:

\_\_A[ ]

◀ heap 원소들이 저장되는 리스트

작업:

insert(x)

◀ heap에 원소 x를 삽입한다

deleteMax()

◀ heap의 최대 원소를 알려주면서 삭제한다

max()

◀ heap의 최대 원소를 알려준다

buildHeap()

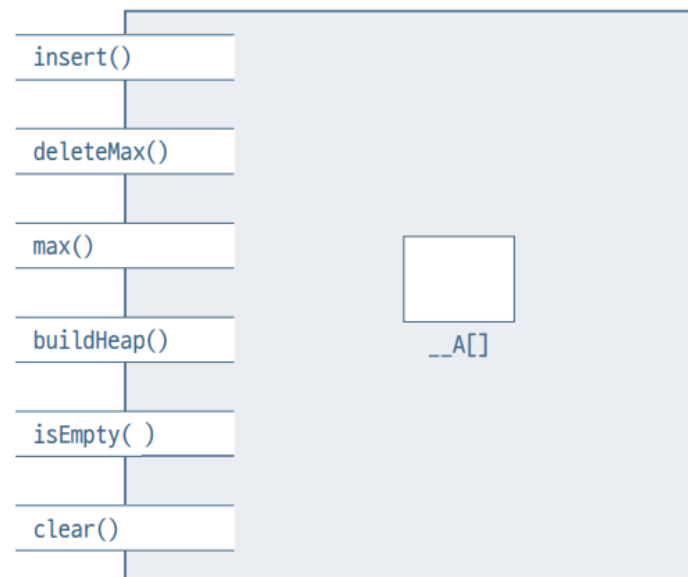
◀ 배열 A[ ]를 heap으로 만든다

isEmpty()

◀ heap이 빈 heap인지 알려준다

clear()

◀ heap을 깨끗이 청소한다

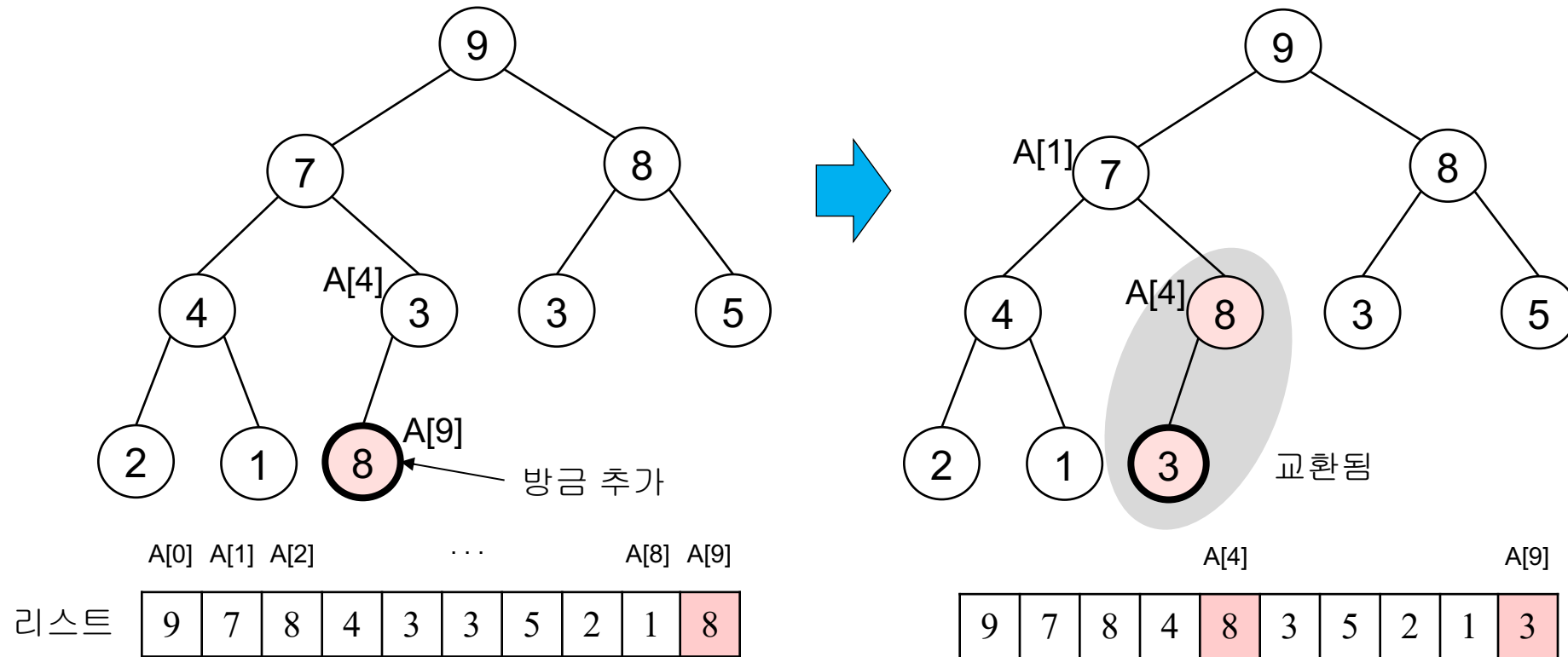


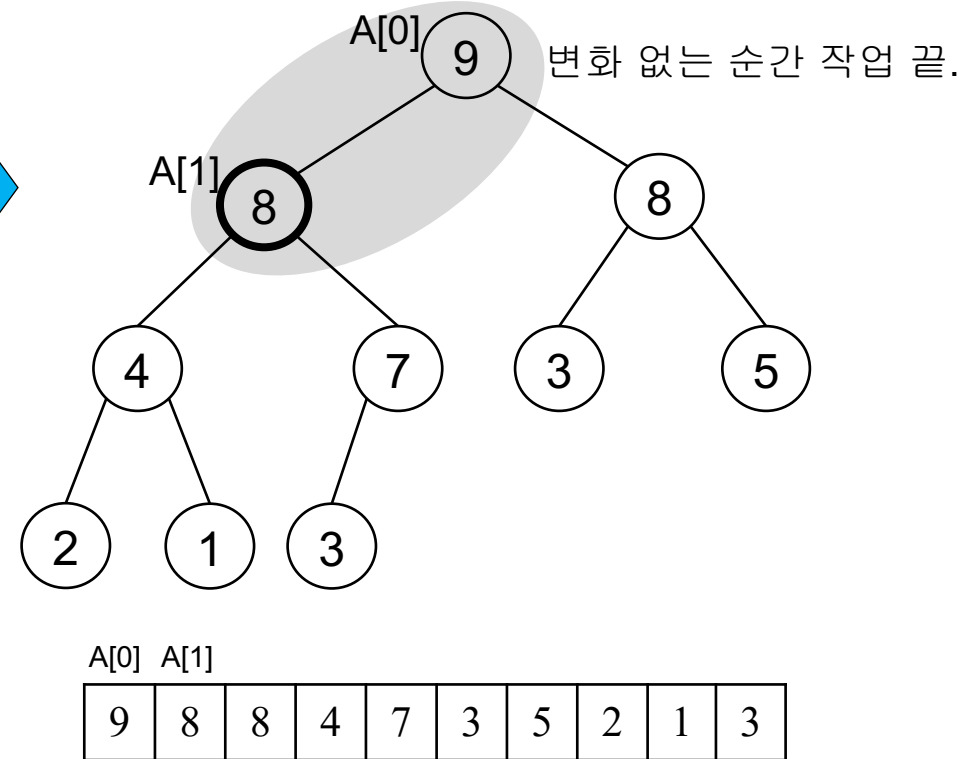
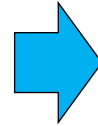
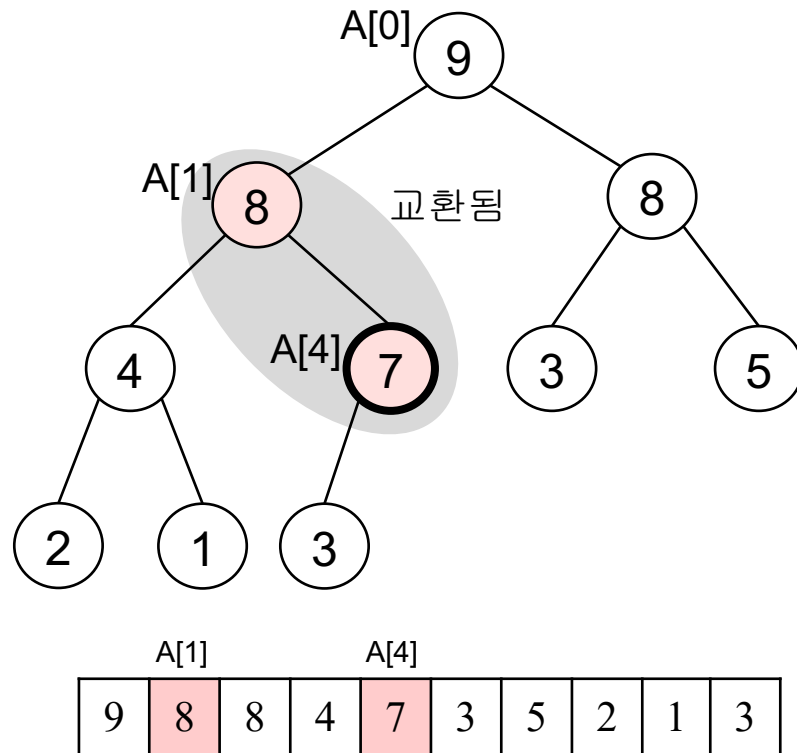


## 02 힙 작업 알고리즘과 구현

# 삽입 Insertion

## 삽입의 예(그림 8-8 주어진 힙에 원소 8을 삽입하는 과정)





아래에서 시작하여  
조정하면서 위로 올라가는 작업을  
**PercolateUp**이라 한다

스며오르기

## Insert():

1. 삽입 원소를 리스트의 맨 끝에 추가
2. 힙특성을 만족하도록 **스며오르기**

# 알고리즘 insert()

## 알고리즘 8-1 힙에 원소 삽입하기

◀ 힙  $A[0 \dots n-1]$ 에 원소를 삽입한다(추가한다)

insert(x):                      ◀ x : 삽입할 원소

$i \leftarrow n$

$A[i] \leftarrow x$

$\text{parent} \leftarrow \lfloor (i-1)/2 \rfloor$

    while ( $i > 0$  and  $A[i] > A[\text{parent}]$ )

$A[i] \leftrightarrow A[\text{parent}]$                       ◀ 맞바꾸기

$i \leftarrow \text{parent}$

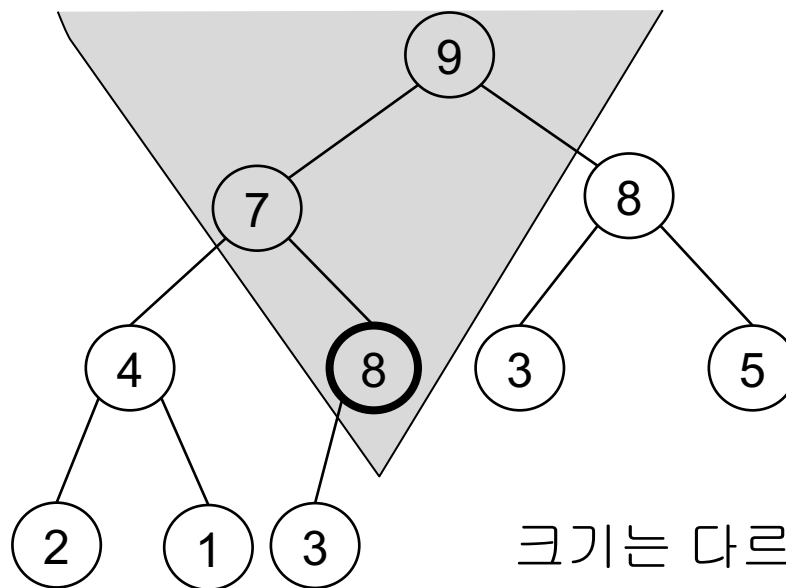
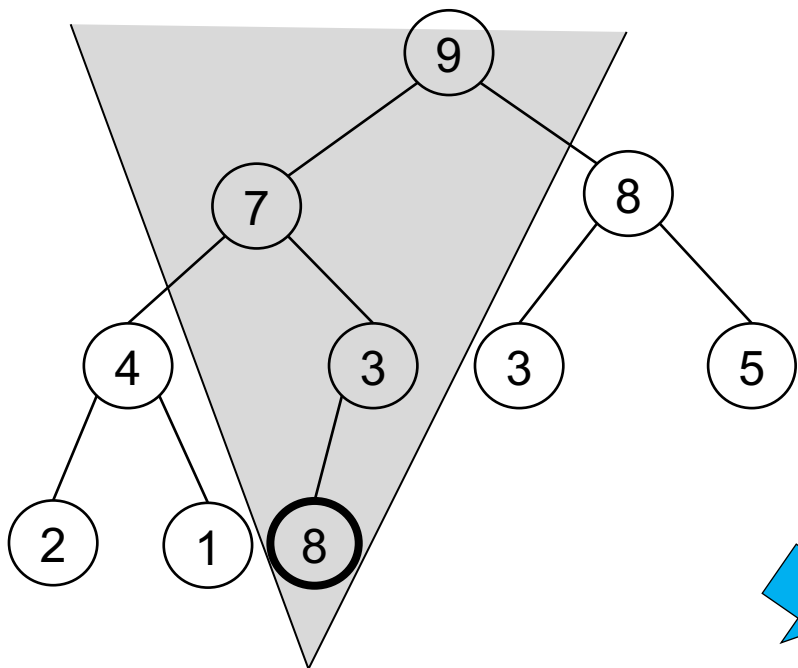
$\text{parent} \leftarrow (i-1)/2$

$n++$                       ◀ 힙 크기 1 증가

파이썬 내장 리스트는  $n$ 을 자동 관리해주므로 구현 시에 명시적으로  $n$ 을 증가시킬 필요 없다

len(리스트\_이름) 함수 기본 제

# percolateUp()의 재귀적 관점



크기는 다르지만 똑같은  
percolateUp 문제를 만났다

# insert()의 재귀 알고리즘 버전

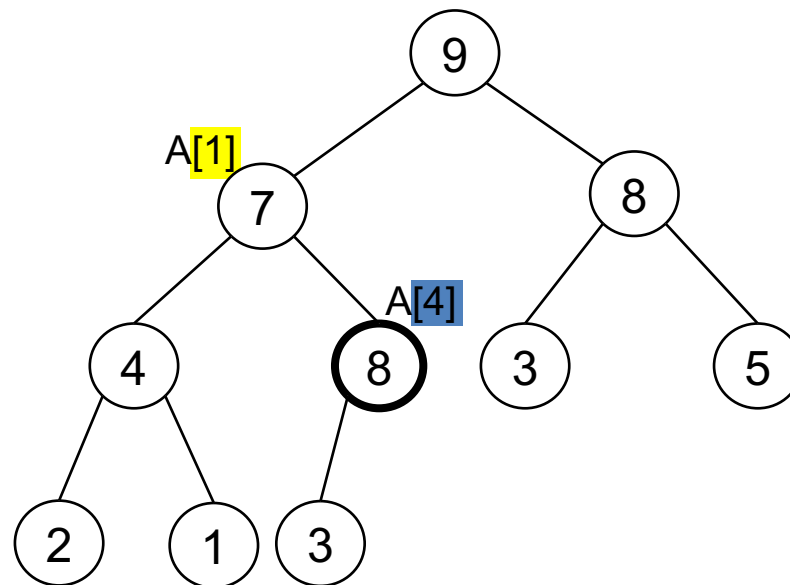
Animation

알고리즘 8-2 힙에 원소 삽입하기 (재귀 알고리즘 버전)

- ◀  $A[i]$ 에서 시작해서  $A[0...i]$ 가 힙성질을 만족하도록 수선한다
- ◀  $A[0...i-1]$ 은 힙성질을 만족하고 있음

```
percolateUp(A[], i):  
    parent ←  $(i-1)/2$   
    if ( $i > 0$  &&  $A[i] > A[\text{parent}]$ )  
         $A[i] \leftrightarrow A[\text{parent}]$   
        percolateUp(A, parent)
```

```
insert(A[], x):  
     $A[n] \leftarrow x$   
    percolateUp(A, n)  
     $n++$       ◀ 힙 크기 1 증가
```



한 번의 `percolateUp` 이 전부:  $O(\log n)$

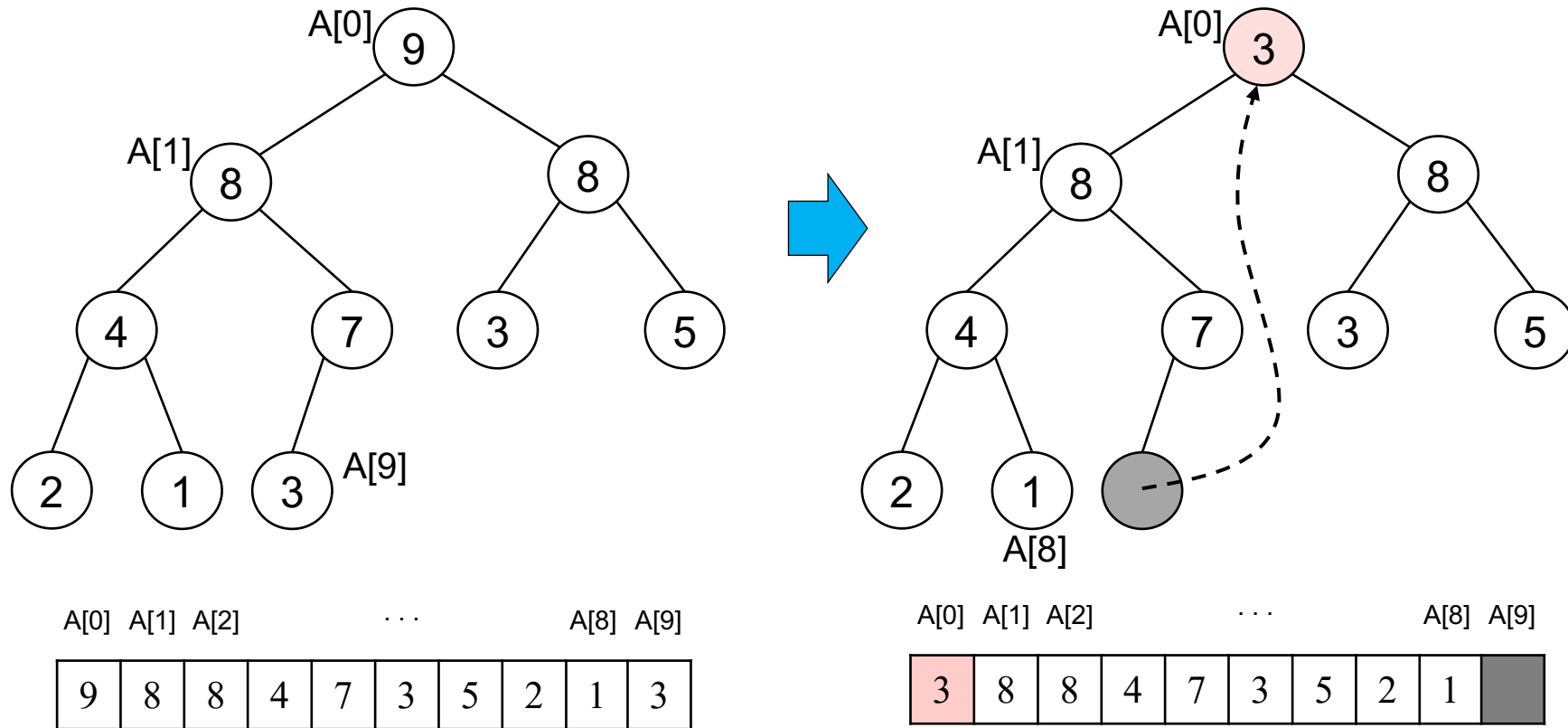
최악의 경우:  $\Theta(\log n)$

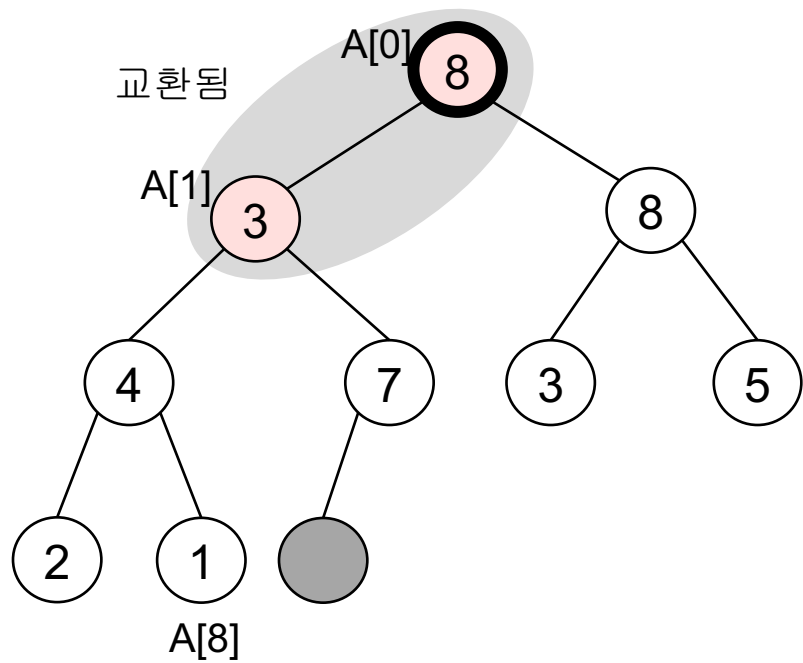
최선의 경우:  $\Theta(1)$



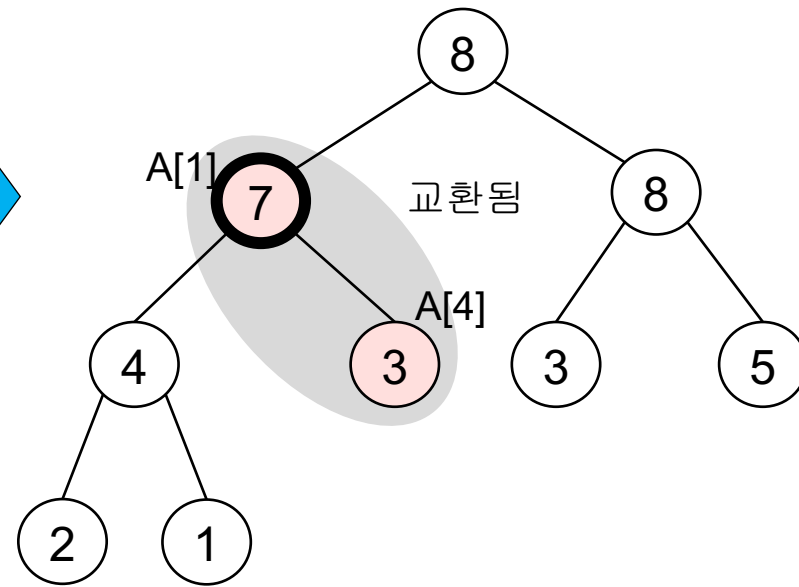
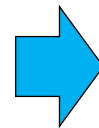
# 삭제 Deletion

삭제의 예(그림 8-9 힙  $A[0..9]$ 에서 원소를 삭제하는 과정)

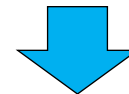


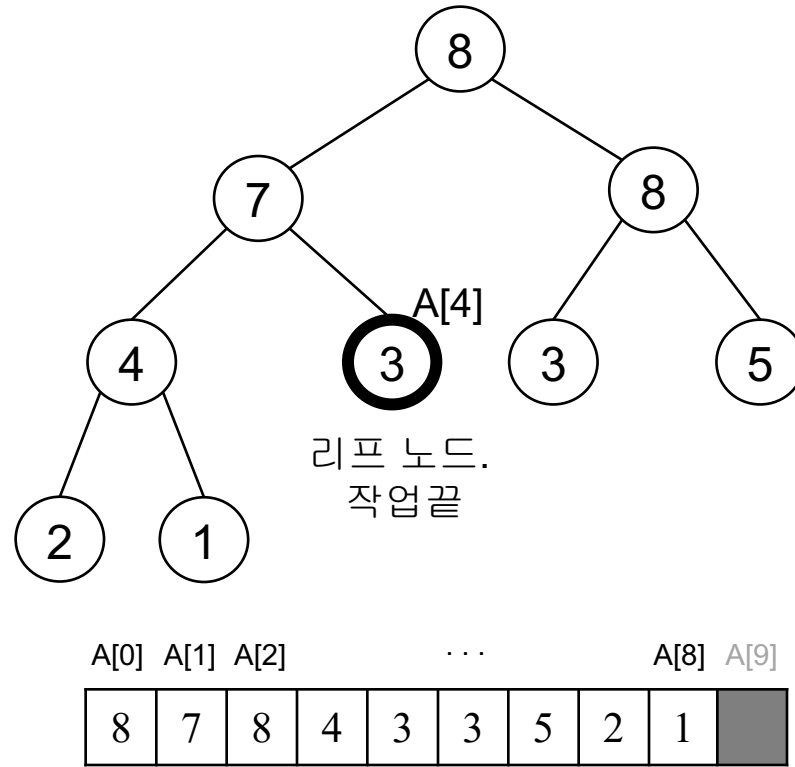


A[0]	A[1]				...			A[8]	A[9]
8	3	8	4	7	3	5	2	1	



	A[1]			A[4]				A[8]	A[9]
8	7	8	4	3	3	5	2	1	



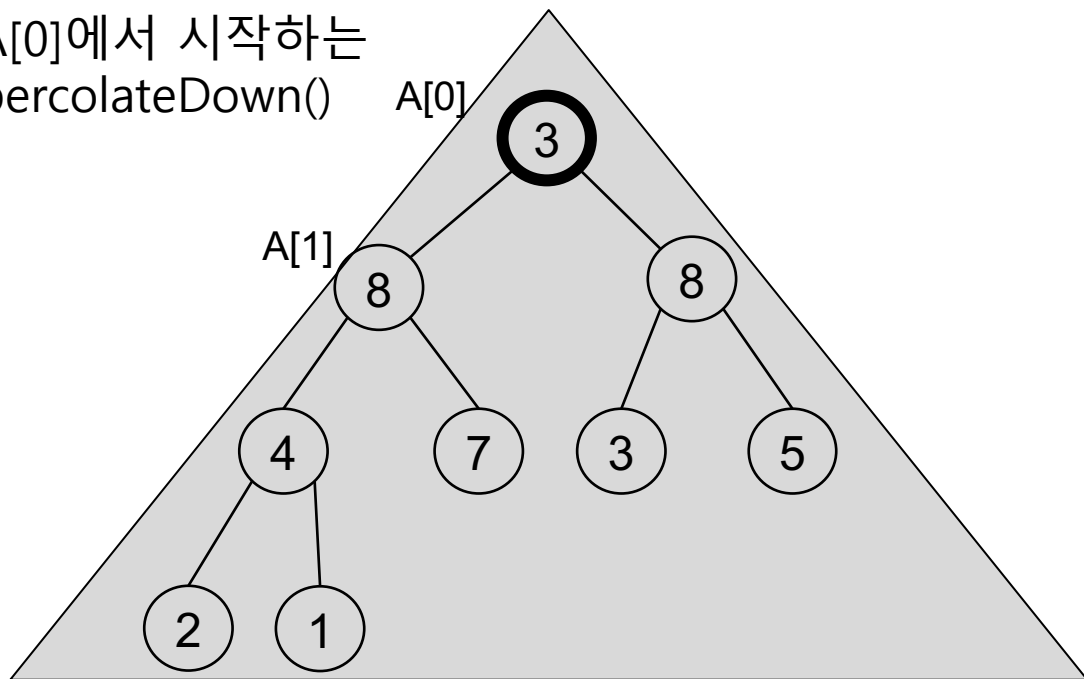


루트에서 시작하여  
조정하면서 아래로 내려가는 작업을  
**PercolateDown**이라 한다

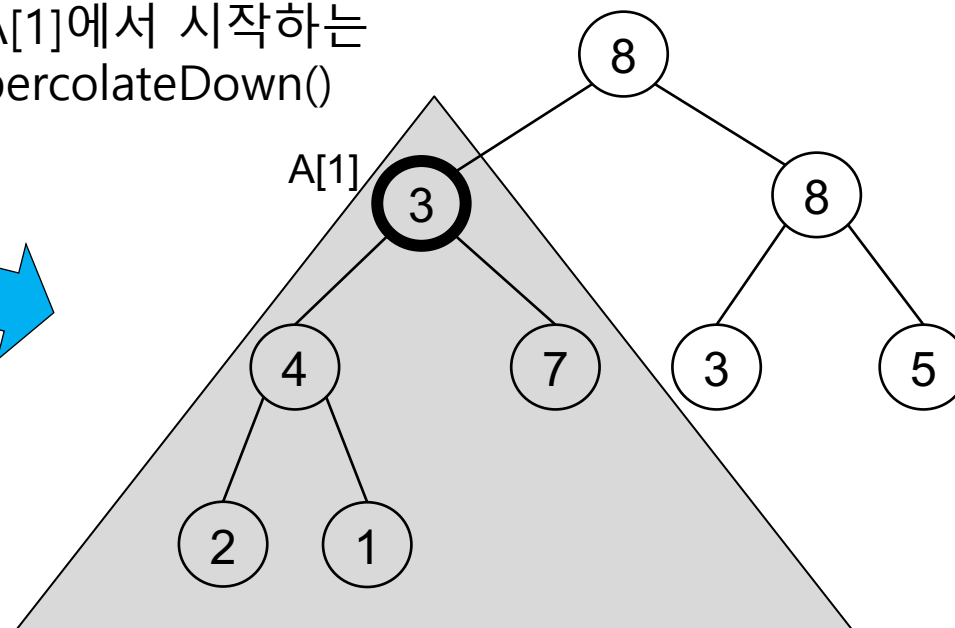
스며내리기

# percolateDown()의 재귀적 관점

A[0]에서 시작하는  
percolateDown()



A[1]에서 시작하는  
percolateDown()



크기는 다르지만 똑같은  
percolateDown 문제를 만났다



## deleteMax():

1. 루트 원소를 리턴
2. 맨 끝 원소를 루트로 이동
3. 힙특성을 만족하도록 **스며내리기**

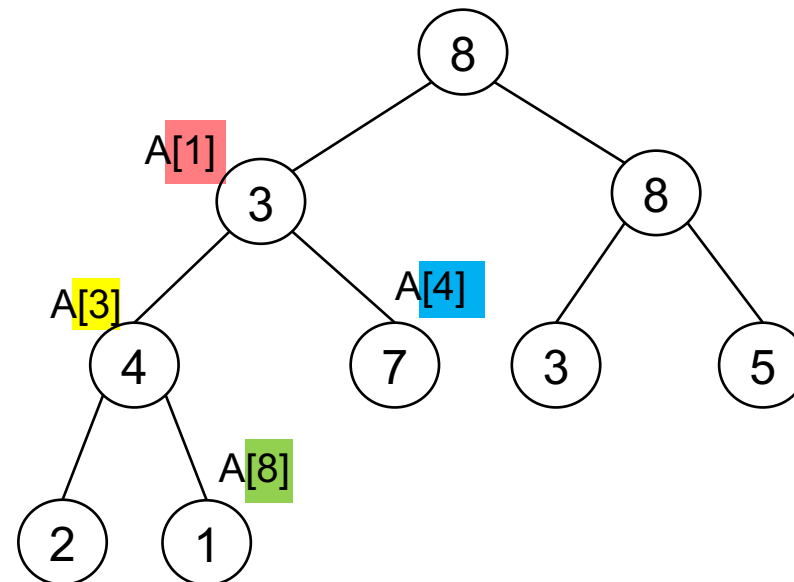
# 알고리즘 percolateDown()과 deleteMax()

Animation

```
percolateDown(A[], k):      ◀ A[k...n-1]을 수선
  child ← 2k+1      ◀ left child
  right ← 2k+2      ◀ right child
  if (child ≤ n-1)
    if (right ≤ n-1 && A[child] < A[right])
      child ← right
    ◀ 이 시점에서 child는 A[2k+1]와 A[2k+2] 중 큰 원소의 인덱스
    if (A[k] < A[child])
      A[k] ↔ A[child] ◀ 맞바꾸기
      percolateDown(A, child)
```

```
deleteMax(A[]):
  max ← A[0]
  A[0] ← A[n-1]
  n--
  percolateDown(A, 0)
  return max
```

알고리즘 8-3 힙에서 원소 삭제하기



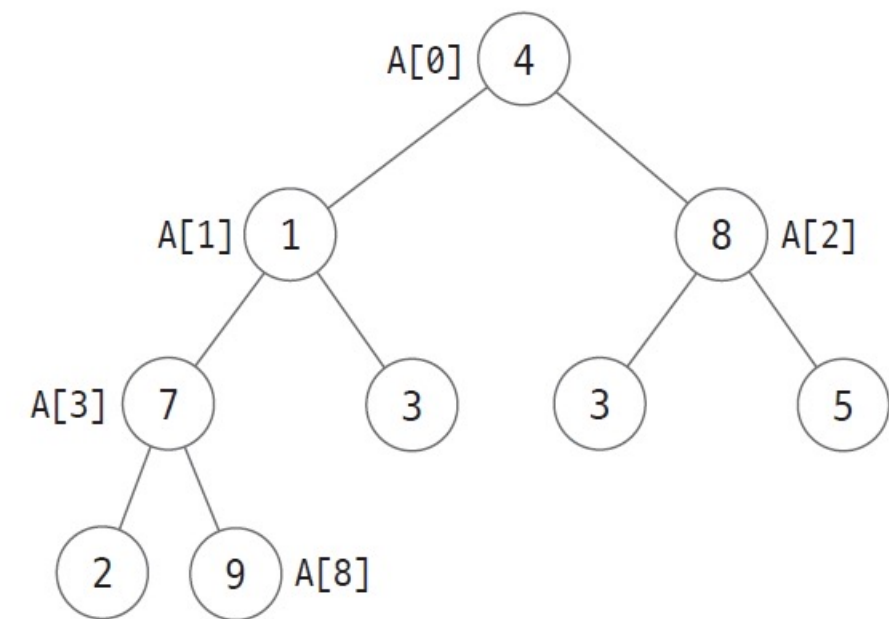
한 번의 percolateDown이 전부:  $O(\log n)$

최악의 경우:  $\Theta(\log n)$

최선의 경우:  $\Theta(1)$

# 힙 만들기

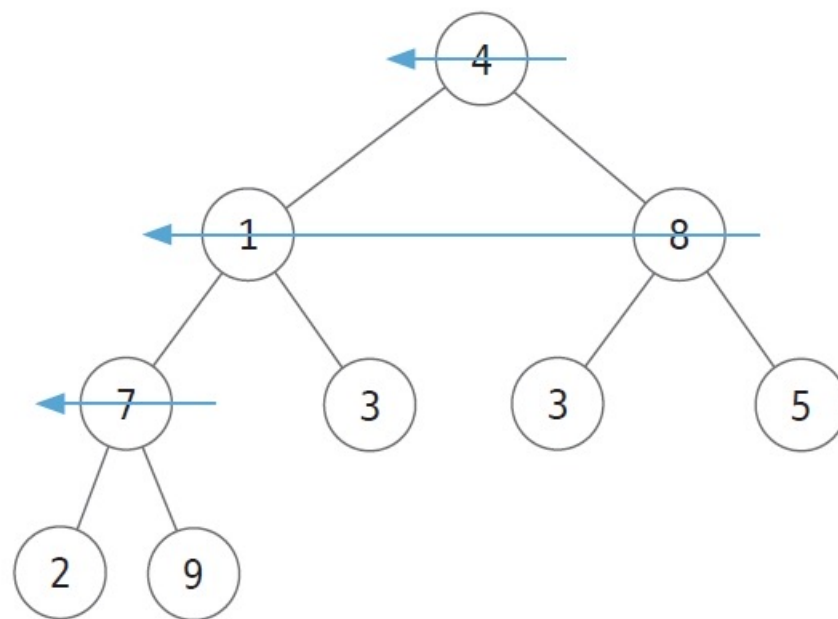
그림 8-11 [그림 8-10]의 완전 이진 트리를 힙으로 수선하는 과정



A[ ]

4	1	8	7	3	3	5	2	9
---	---	---	---	---	---	---	---	---

(a) 9개의 원소로 구성된 리스트와 대응되는 완전 이진 트리



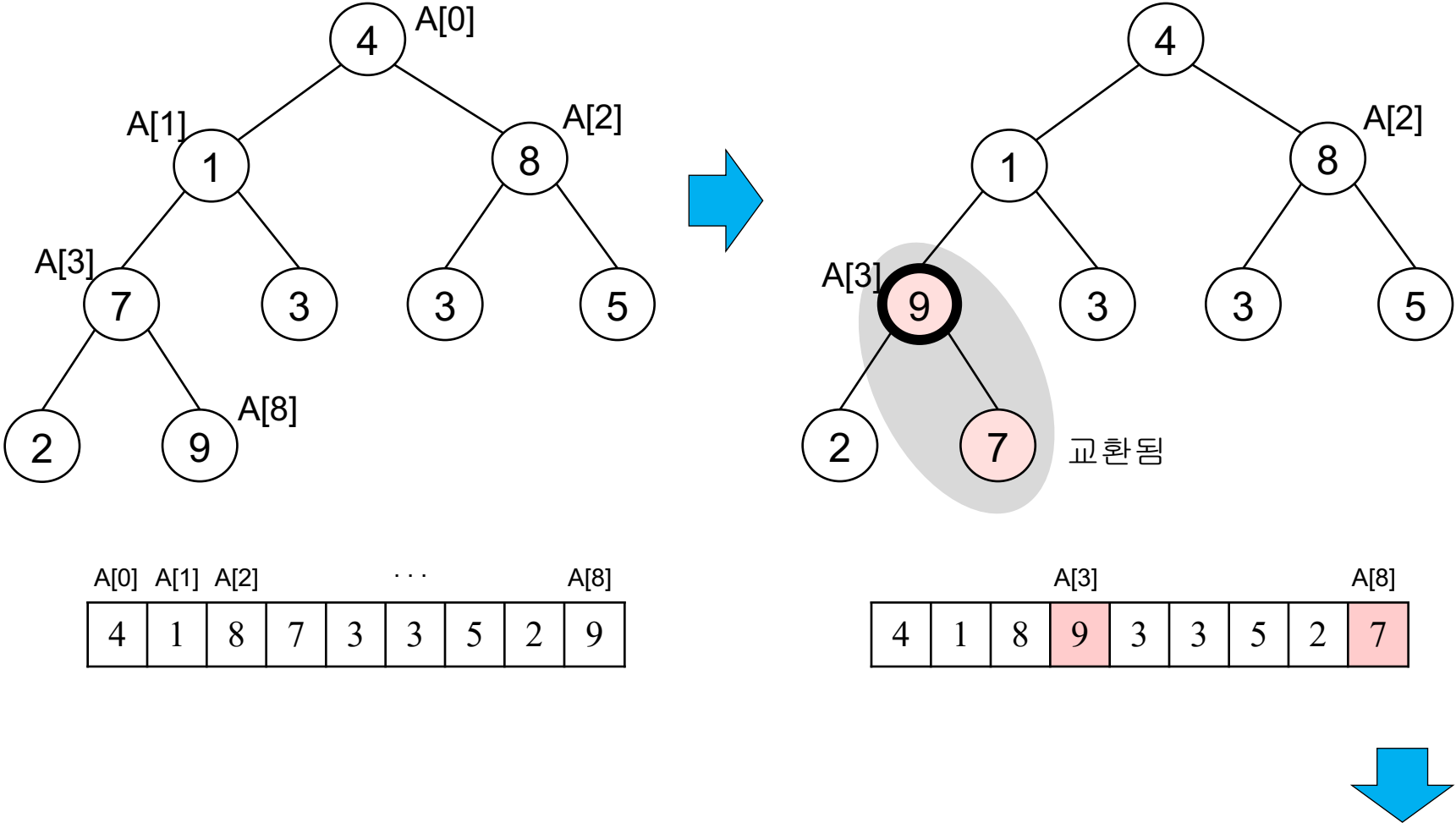
(b) 수선 과정에서 기준이 되는 노드의 순서

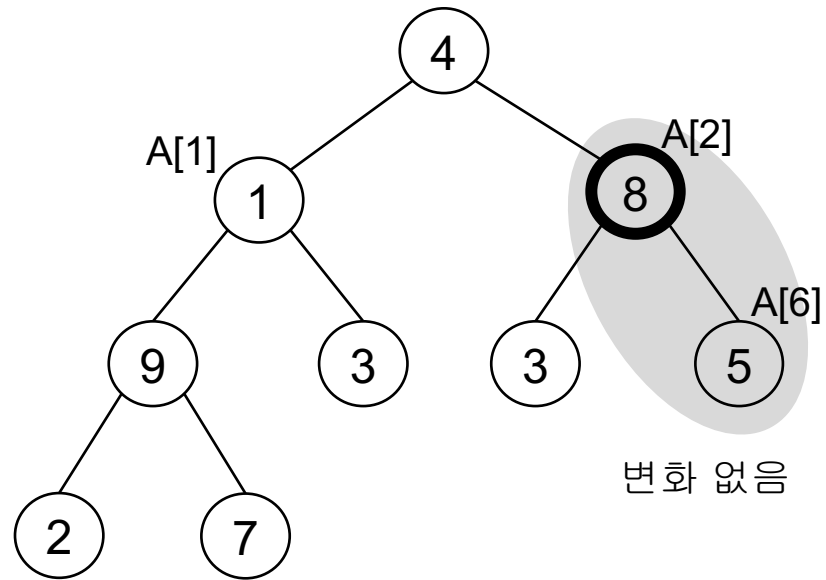
그림 8-10 9개의 원소로 구성된 리스트와 완전 이진 트리, 수선 순서



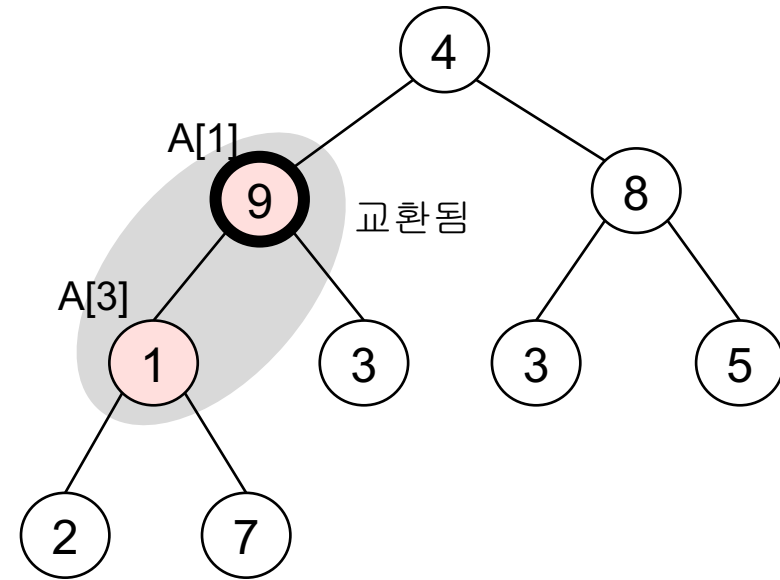
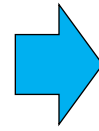
# 힙 만들기

○ : 기준 노드



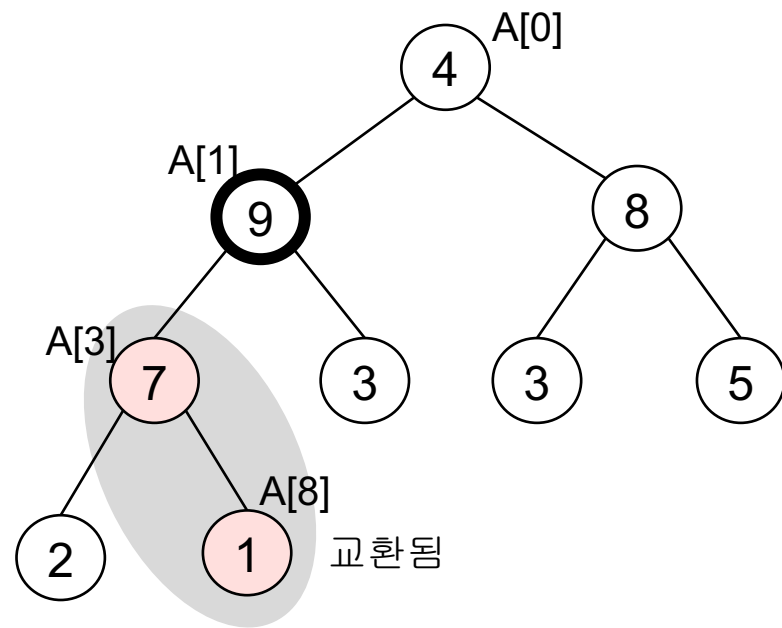


A[2]					A[6]			
4	1	8	9	3	3	5	2	7

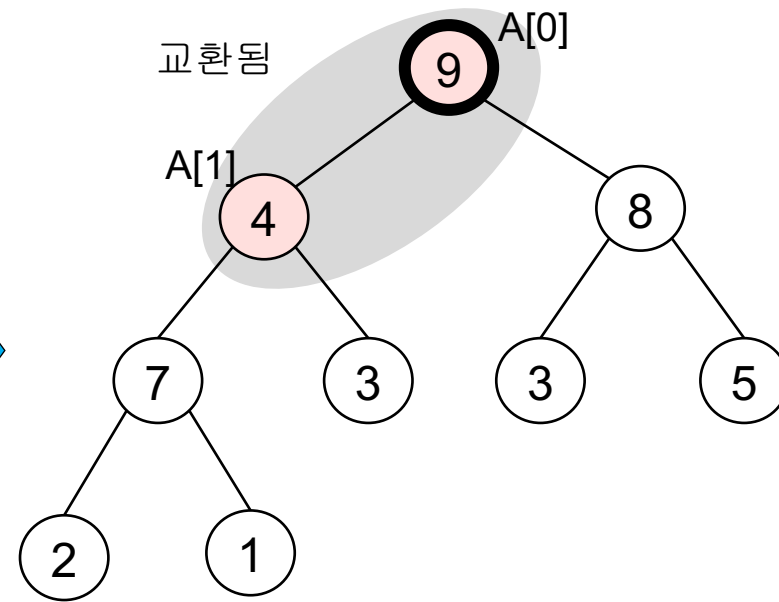


A[1]		A[3]						
4	9	8	1	3	3	5	2	7



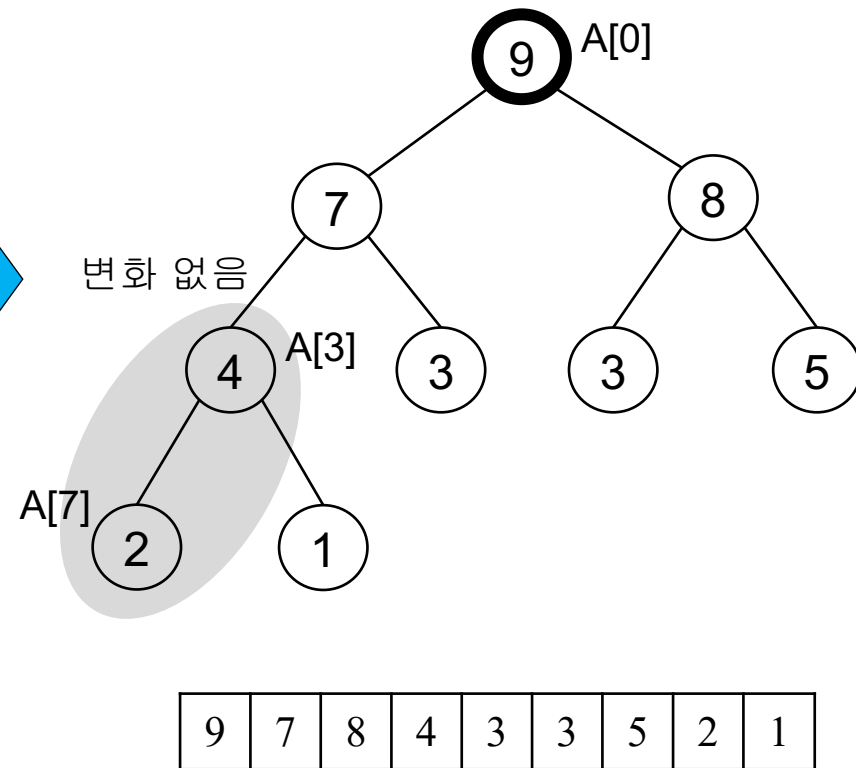
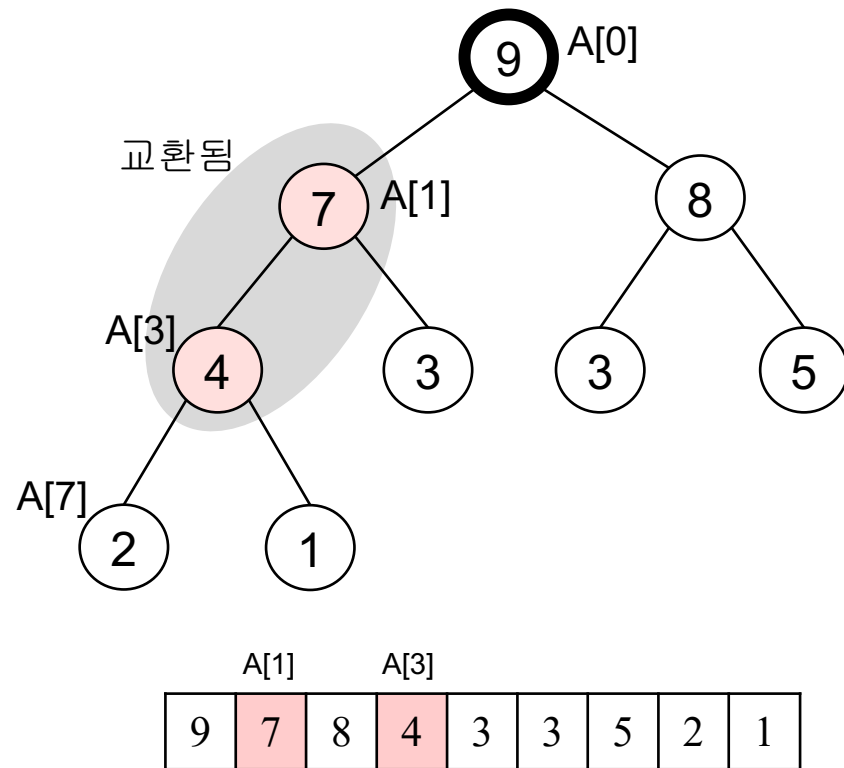


			A[3]					A[8]
4	9	8	7	3	3	5	2	1

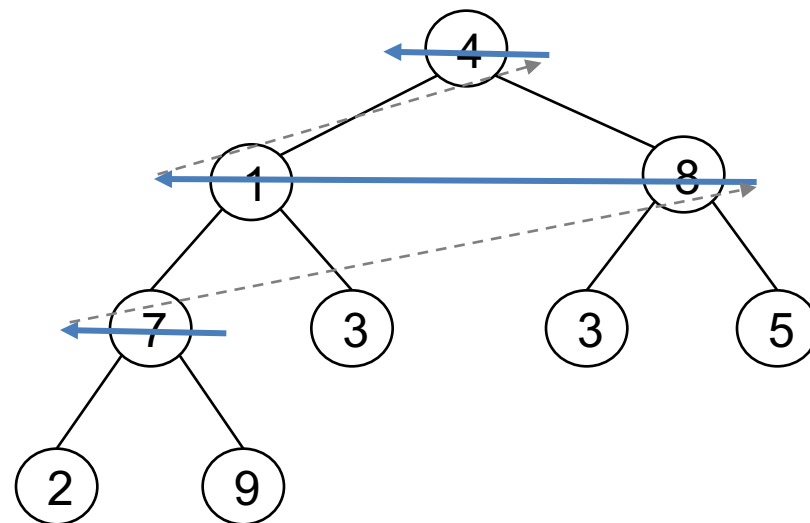
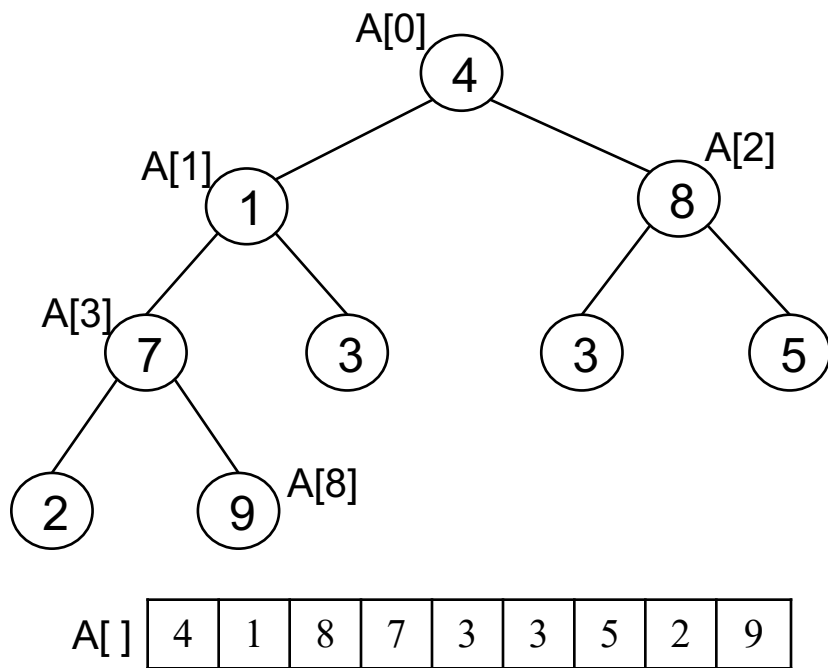


A[0]	A[1]							
9	4	8	7	3	3	5	2	1





# percolateDown 순서



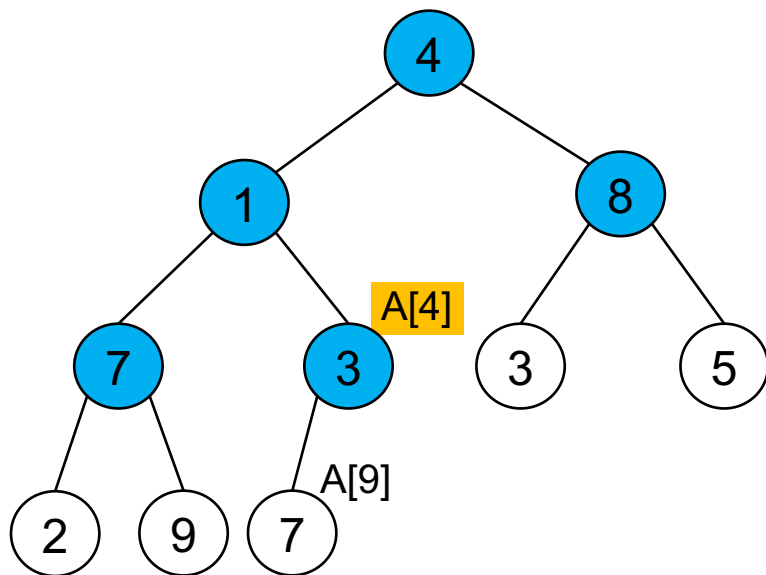
percolateDown 순서

# 알고리즘 buildHeap()

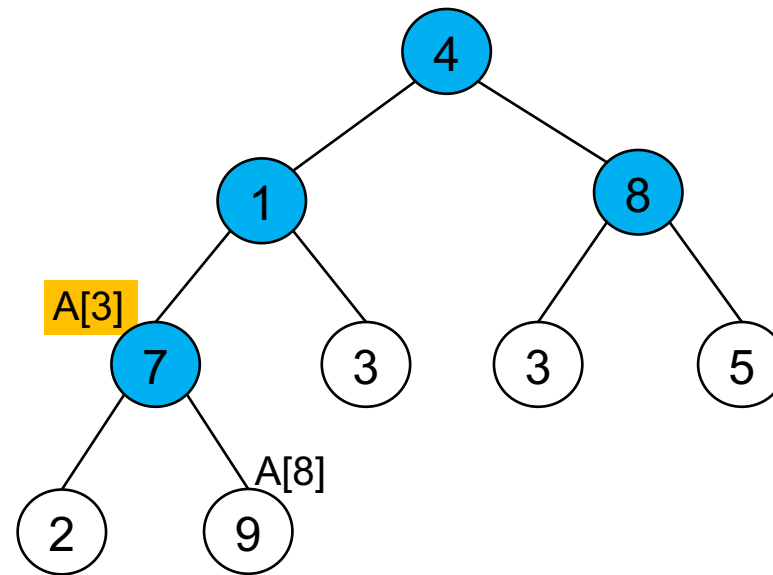
Animation

코드 8-4 힙 만들기

```
def buildHeap(self):  
    for i in range( (len(self.__A) - 2) // 2, -1, -1): # 역순으로 A[0]까지  
        self.percolateDown(i)
```



예 1



예 2

percolateDown들의 시간을 모두 합친 것:  $\theta(n)$

percolateDown은 총  $\left\lfloor \frac{n}{2} \right\rfloor$ 번

- 이 중 반은 1 레벨에 걸침
- 이 중 1/4은 2 레벨에 걸침
- 이 중 1/8은 3 레벨에 걸침

...

- 이 중 1개는  $\lfloor \log_2 n \rfloor$  레벨에 걸침

이들을 가중합 하면  $\theta(n)$ 이 된다

# 기타 작업

코드 8-5 힙의 최댓값 구하기

```
def max(self):  
    return self.__A[0]
```

코드 8-6 힙이 비었는지 확인하기

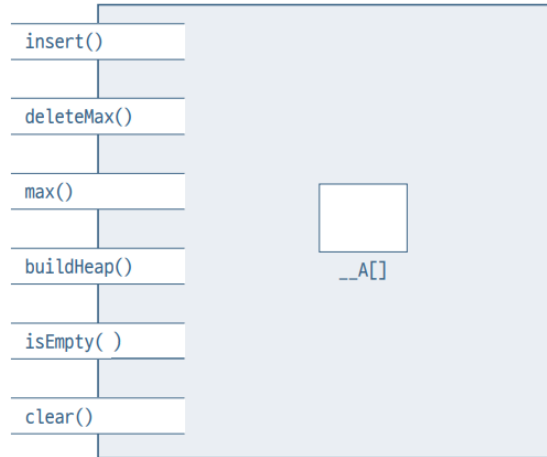
```
def isEmpty(self):  
    return len(self.__A) == 0
```

코드 8-7 힙 비우기

```
clear():  
    self.__A = []
```



# 파이썬 구현



힙 객체 구조

```
class Heap:
    def __init__(self, *args):
        if len(args) != 0:
            self.__A = args[0] # 파라미터로 온 리스트
        else:
            self.__A = []

    def insert(self, x):
        self.__A.append(x)
        self.__percolateUp(len(self.__A)-1)

    def __percolateUp(self, i:int):
        parent = (i - 1) // 2
        if i > 0 and self.__A[i] > self.__A[parent]:
            self.__A[i], self.__A[parent] = self.__A[parent], self.__A[i]
            self.__percolateUp(parent)

    def deleteMax(self):
        # heap is in self.__A[0...len(self.__A)-1]
        if (not self.isEmpty()):
            max = self.__A[0]
            self.__A[0] = self.__A.pop() # pop(): 리스트의 끝원소 삭제 후 리턴
            self.__percolateDown(0)
            return max
        else:
            return None
```

```
def __percolateDown(self, i:int):
    # Percolate down w/ self.__A[i] as the root
    child = 2 * i + 1 # left child
    right = 2 * i + 2 # right child
    if (child <= len(self.__A)-1):
        if (right <= len(self.__A)-1 and self.__A[child] < self.__A[right]):
            child = right # index of larger child
        if self.__A[i] < self.__A[child]:
            self.__A[i], self.__A[child] = self.__A[child], self.__A[i]

def max(self):
    return self.__A[0]

def buildHeap(self):
    for i in range((len(self.__A) - 2) // 2, -1, -1):
        self.__percolateDown(i)

def isEmpty(self) -> bool:
    return len(self.__A) == 0

def clear(self):
    self.__A = []

def size(self) -> int:
    return len(self.__A)
```