우선순위 큐: 힙

1. ADT 우선순위 큐
2. 힙의 정의
3. 힙의 작업들
4. Java 구현

1. ADT 우선순위 큐

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 Priority Queue

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

ADT PriorityQueue
원소 x를 삽입한다
최대 원소를 알려주면서 삭제한다
최대 원소를 알려준다

주목할 사실:

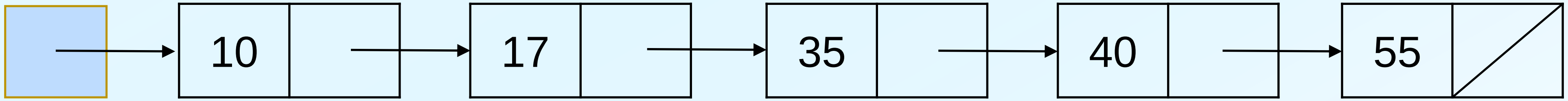
우선순위 큐에서 유일하게 접근 가능한 원소는 최대 원소뿐

비효율적인 우선순위 큐

배열 리스트

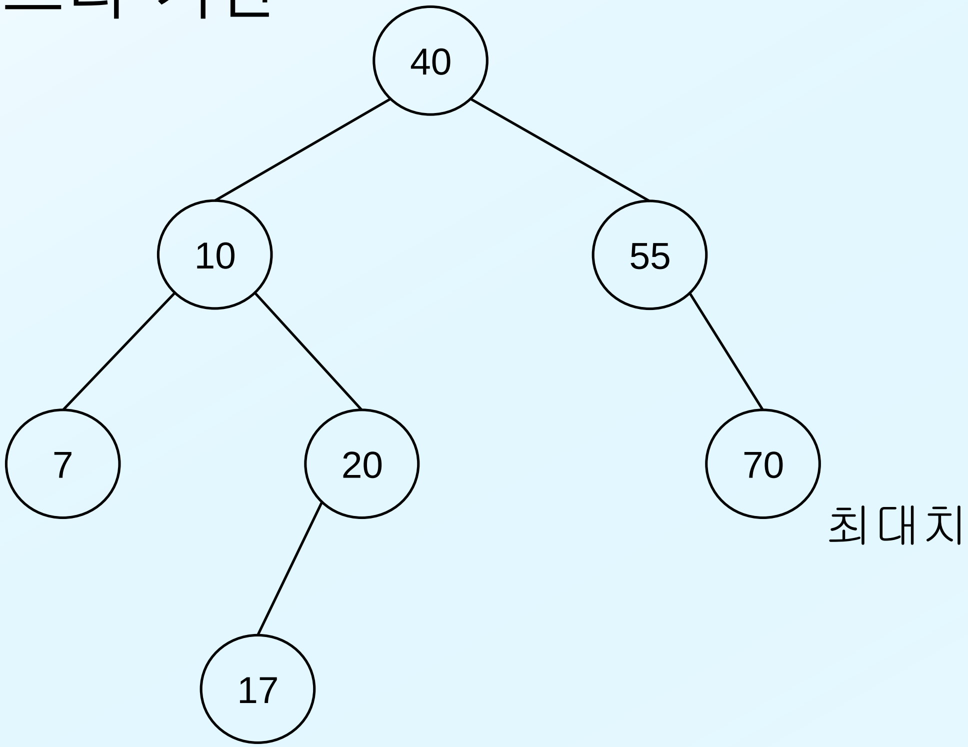
item[]	10	35	40	17	95	50	48	33	9			

연결 리스트



이진 검색 트리(10장)도 우선순위 큐로 부적당

이진 검색 트리 기반



동일한 key가 2개 이상일 때 별도로 처리를 해주어야 한다
이게 아니라도 우선순위 큐 용도로는 너무 과하다

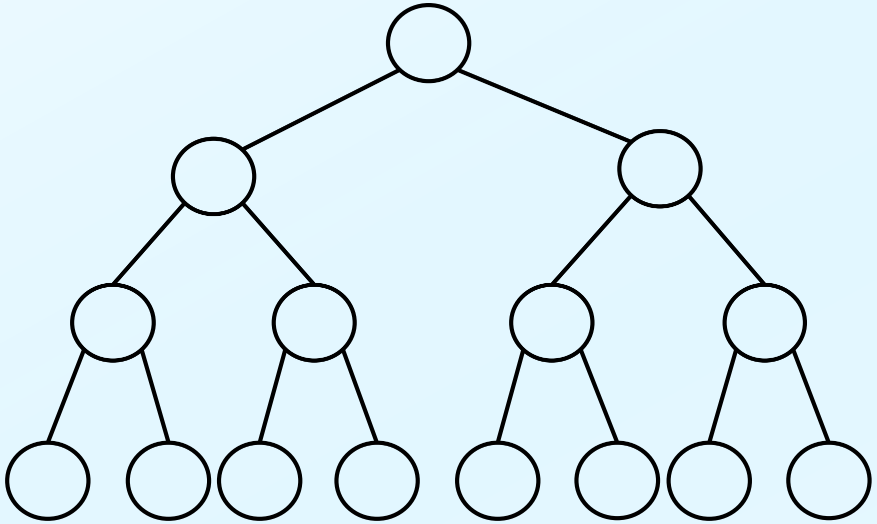
2. Heap의 정의

힙Heap을 정의하기 전에..

포화 이진 트리Full Binary Tree

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

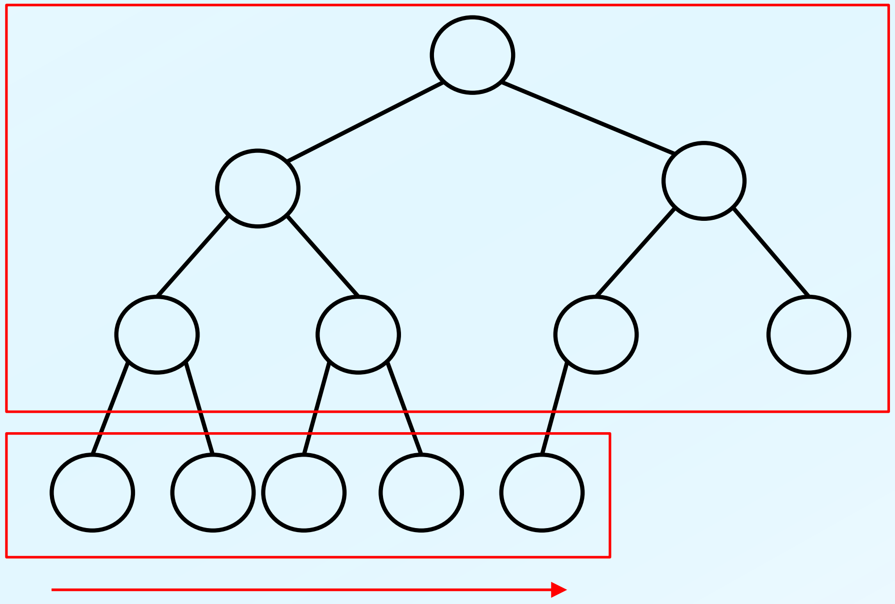
노드 수가 2^k-1일 때만 가능



완전 이진 트리 Complete Binary Tree

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

노드의 수가 맞지 않아 full binary tree를 만들 수 없으면
맨 마지막 레벨은 왼쪽부터 채워나간다



힙: 대표적인 우선순위 큐

힙은 다음을 두가지 조건을 만족해야 한다

힙특성Heap property이라 한다

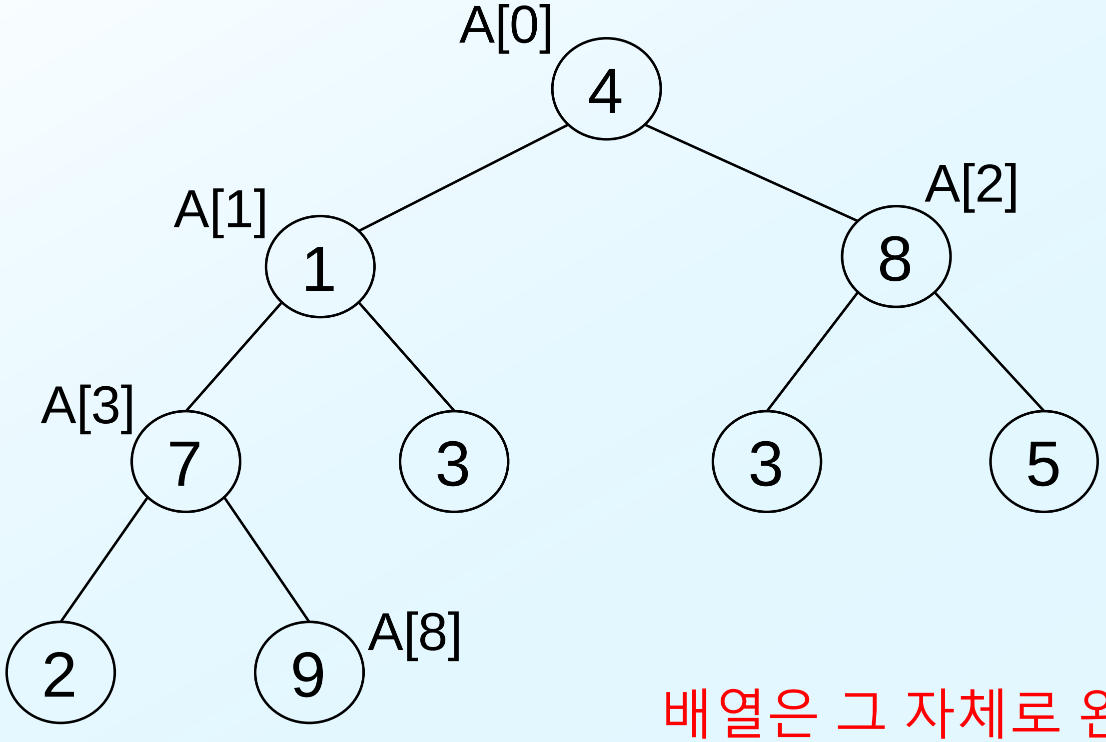
1. 완전 이진 트리
2. 각 노드의 원소 값이 자신의 자식 노드 원소보다 크거나 같다

결과적으로, 루트 노드가 제일 큰 원소를 갖게 됨

최대힙Maxheap : 최소힙Minheap

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

힙은 배열과 안성맞춤

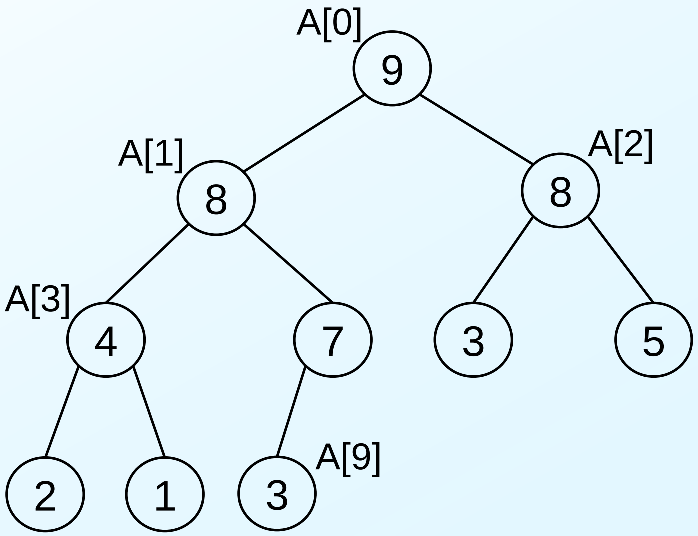


A[ ]									
	4	1	8	7	3	3	5	2	9

노드 i의 자식 노드: $2i+1$, $2i+2$
노드 i의 부모 노드: $\lfloor (i-1)/2 \rfloor$

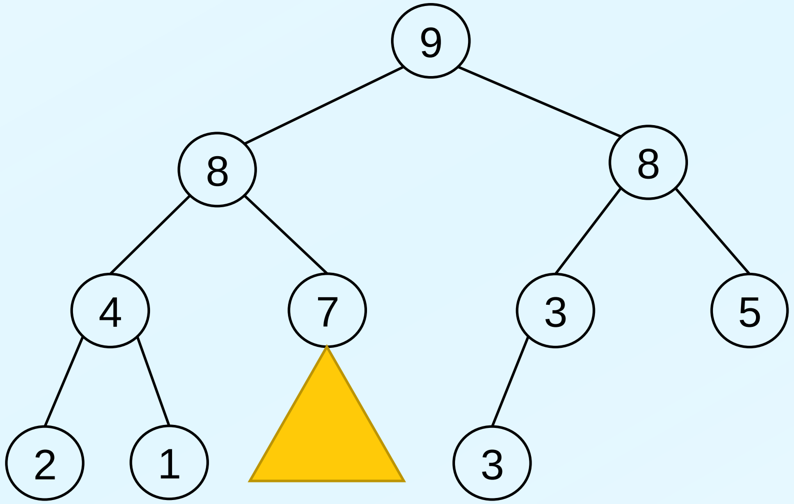
배열은 그 자체로 완전 이진 트리로 볼 수 있다
(배열에 저장한다는 사실로 완전 이진 트리 조건은 자동 만족)

힙의 예



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

10개의 원소로 구성된 힙과
대응되는 배열



힙특성은 만족하지만
완전 이진 트리를 만족하지 못하는 예

3. Heap의 작업들

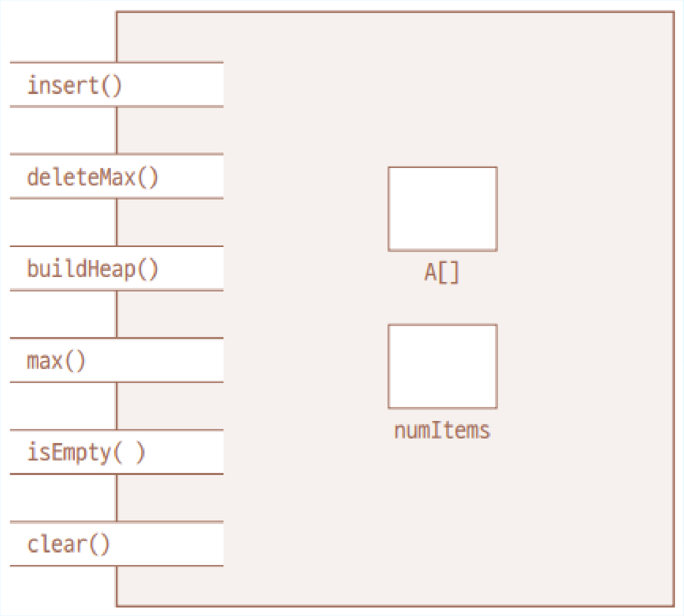
힙 객체 구조

필드:

A[ ] ◀ heap 원소들이 저장되는 배열
numItems ◀ heap에 있는 원소의 총 수

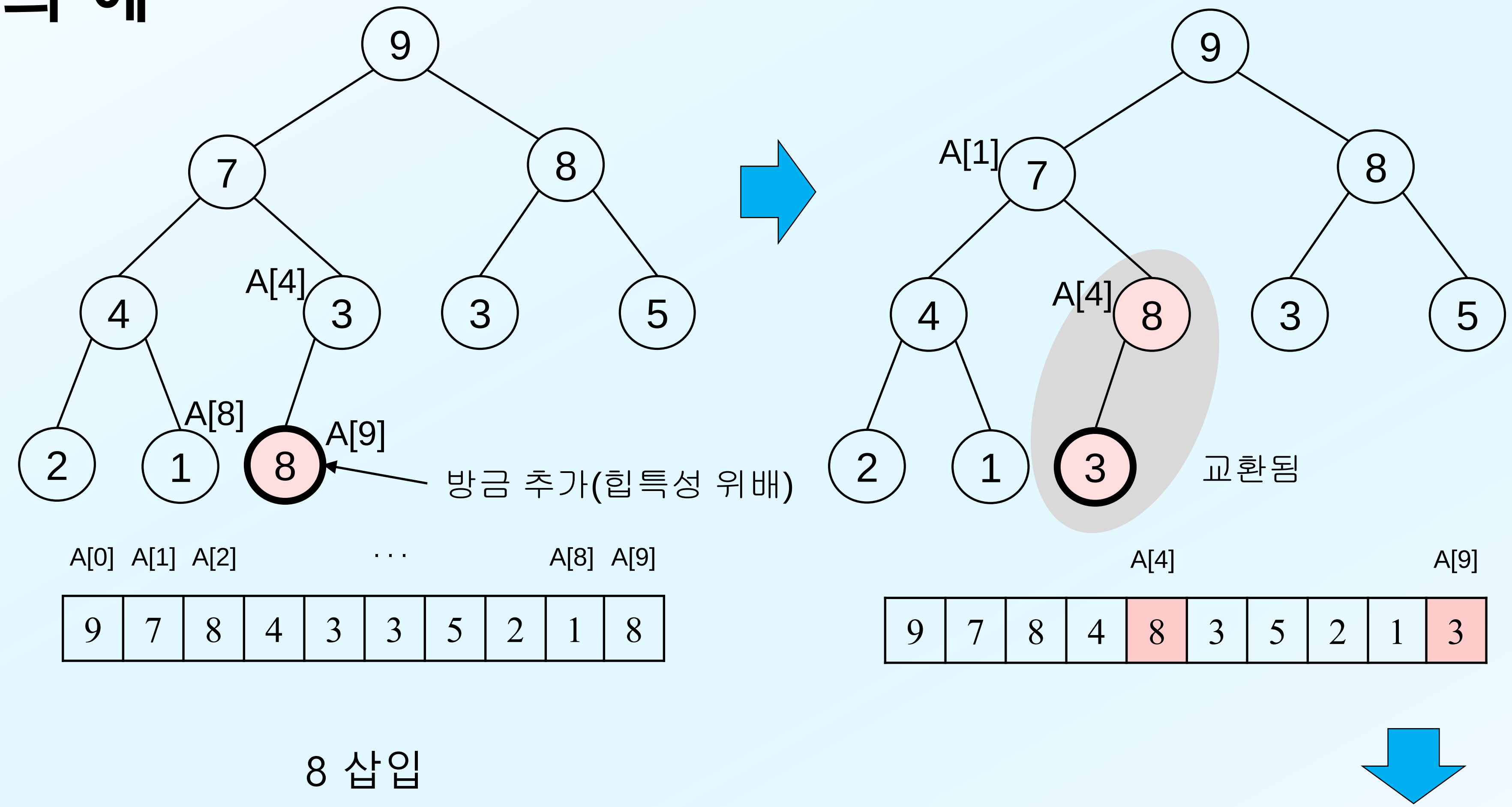
작업:

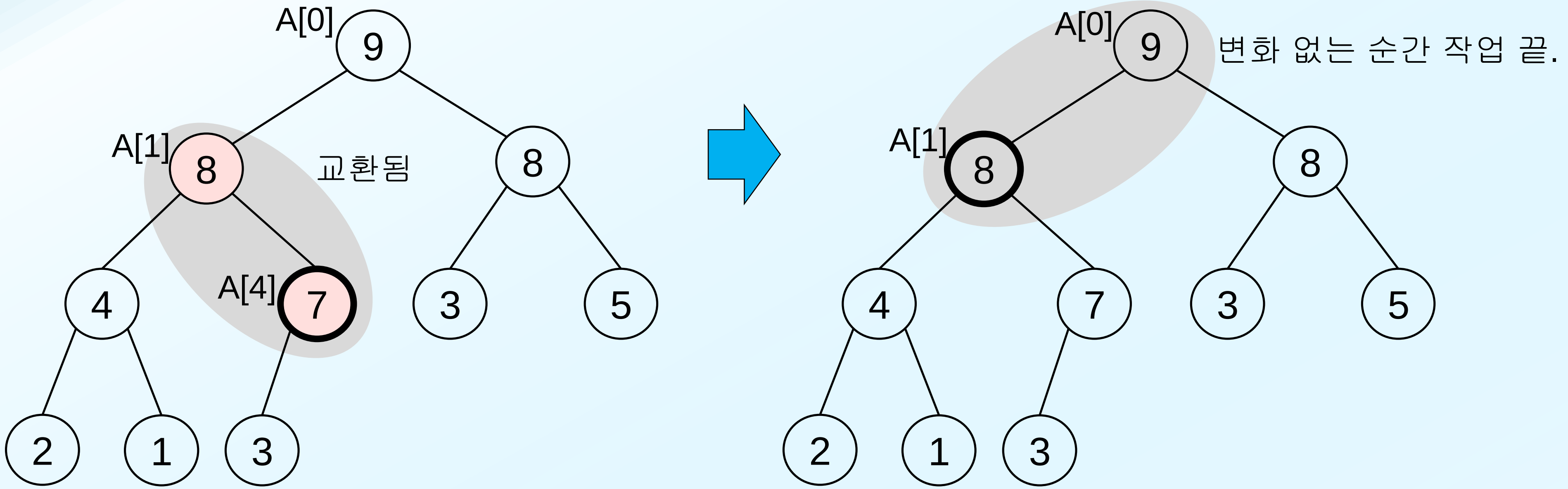
insert(x) ◀ heap에 원소 x를 삽입한다
deleteMax() ◀ heap의 최대 원소를 알려주면서 삭제한다
buildHeap() ◀ 배열 A[ ]를 heap으로 만든다
max() ◀ heap의 최대 원소를 알려준다
isEmpty() ◀ heap이 빈 heap인지 알려준다
clear() ◀ heap을 깨끗이 청소한다



삽입 Insertion

삽입의 예





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

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

아래에서 시작하여
조정하면서 위로 올라가는 작업을
PercolateUp이라 한다

스며오르기

Insert():

1. 삽입 원소를 맨 끝에 추가
2. 힙특성을 만족하도록 percolateUp

알고리즘 insert()

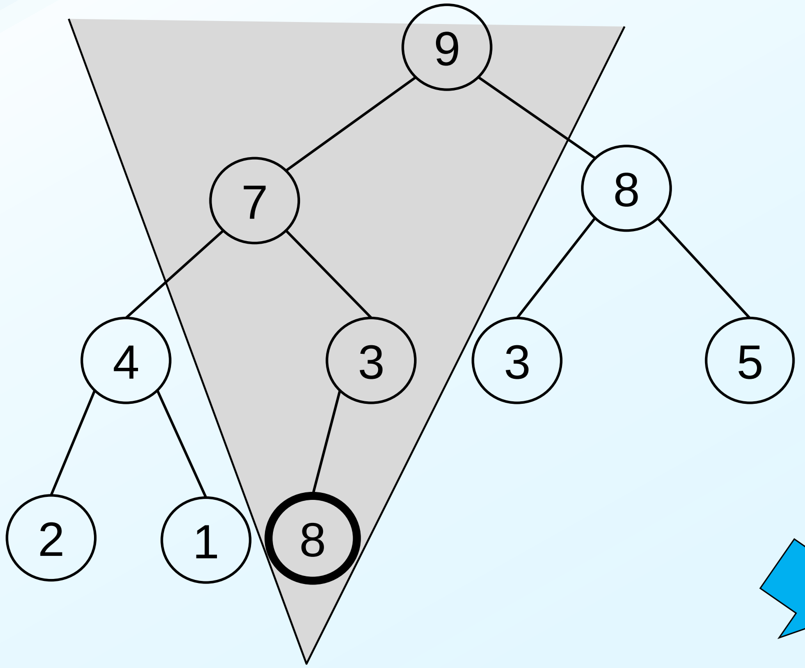
```
◀ 힙 A[0... n−1]에 원소를 삽입한다(추가한다)
insert(A[], x):      ◀ x : 삽입할 원소
        i ← n
        A[i] ← x
        parent ← (i−1)/2
        while ( i > 0 && A[i] > A[parent] )
                A[i] ↔ A[parent]   ◀ 맞바꾸기
                i ← parent
                parent ← (i−1)/2
        n++   ◀ 힙의 크기 1 증가
```

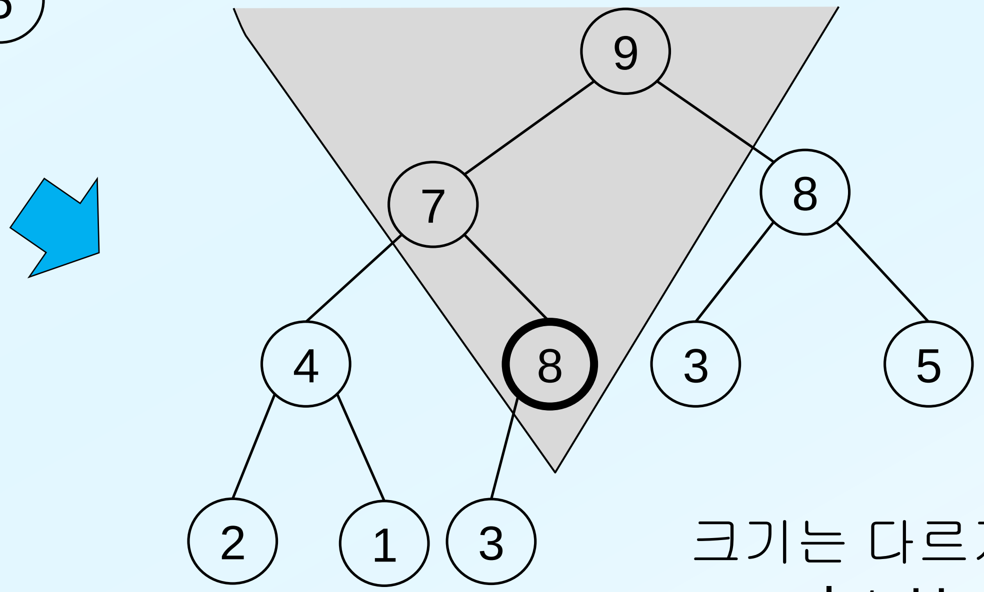
*n*은 앞의 numItems에 해당

지나가는 길에..

위 코드에서 while문의 “i<0” 조건을 빼면?

percolateUp()의 재귀적 관점



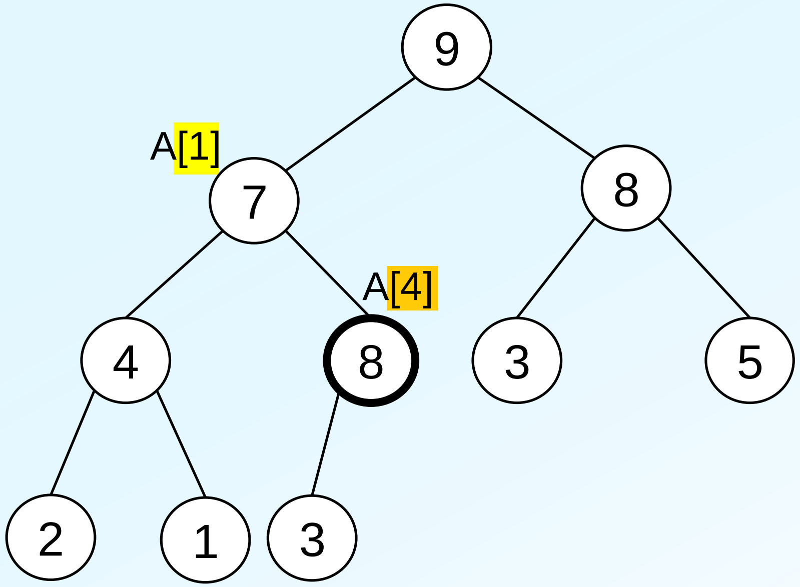


크기는 다르지만 똑같은
percolateUp 문제를 만났다

insert()의 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[parent]) ◀ "i > 0"는 없어도 됨
                A[i] ↔ A[parent]
                percolateUp(A, parent)
```

```
insert(A[], x):
        A[n] ← x
        percolateUp(A, n)
        n++    ◀ 힙의 크기가 1 증가
```



삽입 작업의 수행 시간

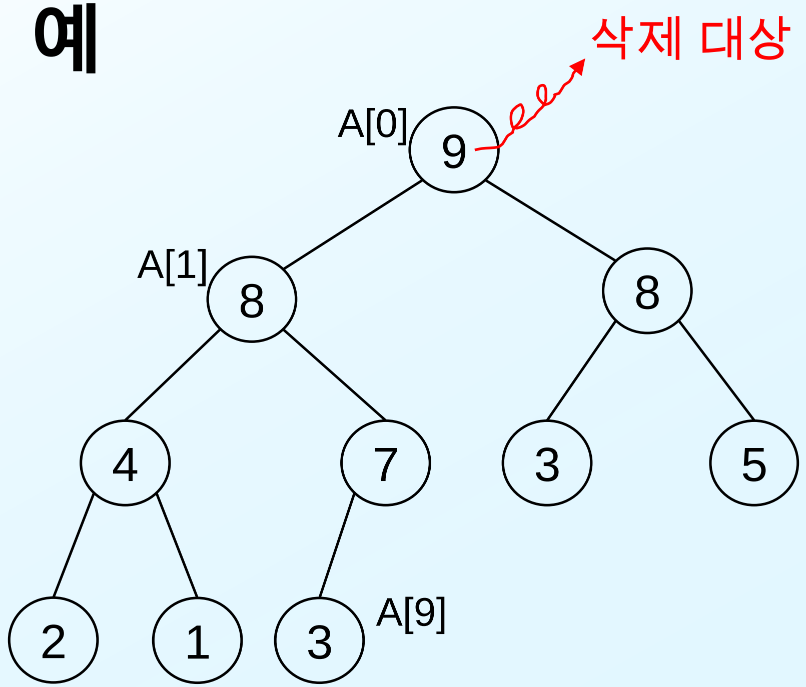
한 번의 percolateUp 이 전부: $O(\log n)$

최악의 경우: $\Theta(\log n)$

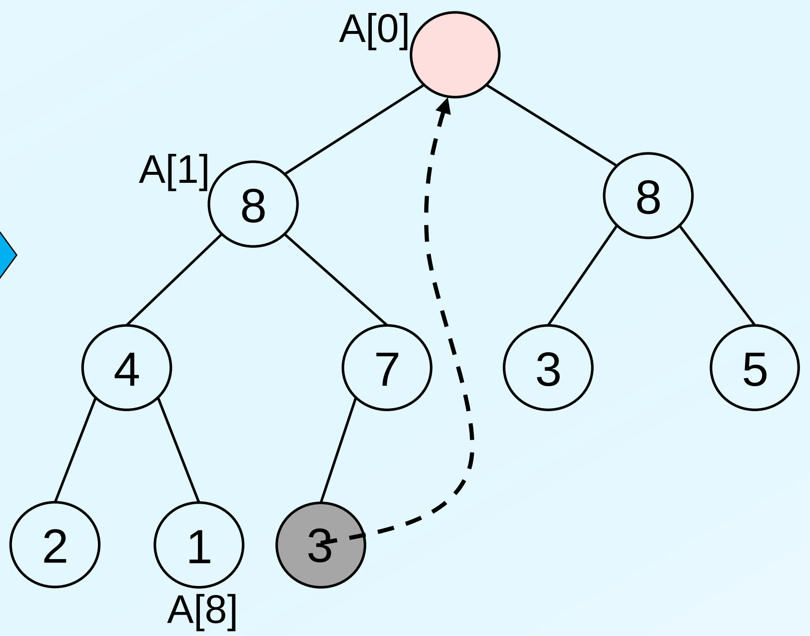
최선의 경우: $\Theta(1)$

삭제 Deletion

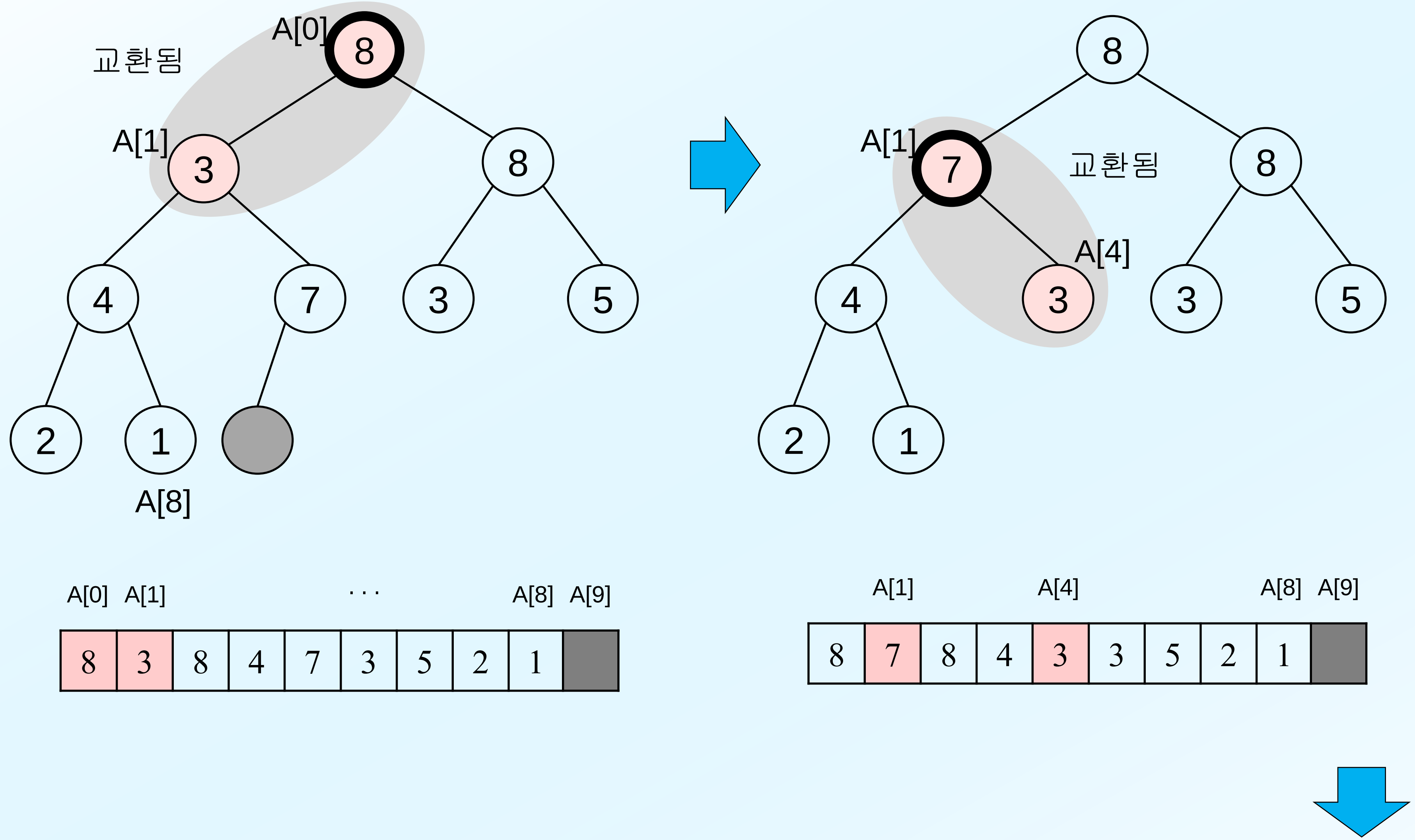
삭제의 예

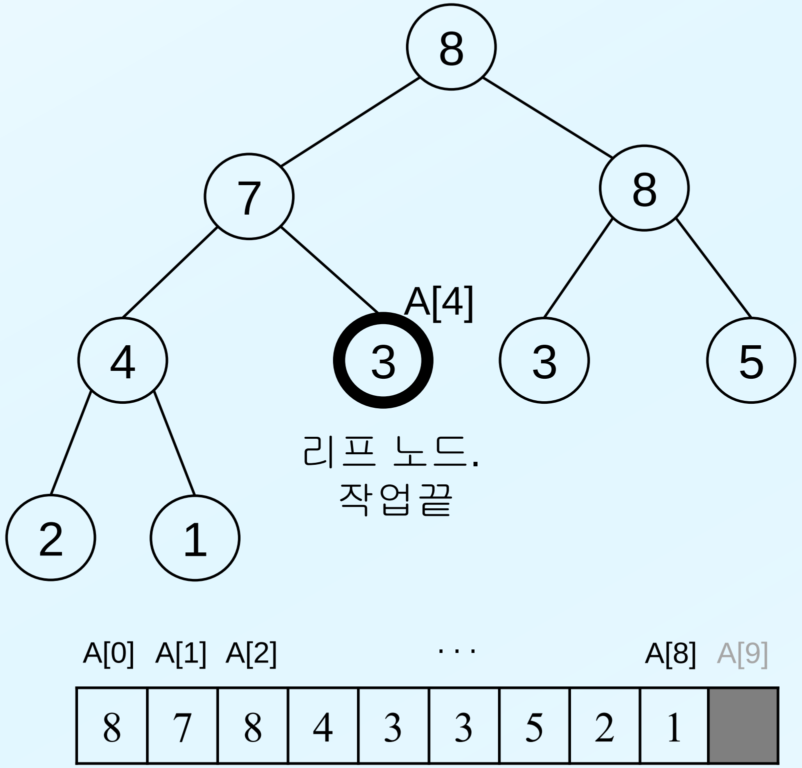


A[0]	A[1]	A[2]			· · ·			A[8]	A[9]
9	8	8	4	7	3	5	2	1	3



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


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

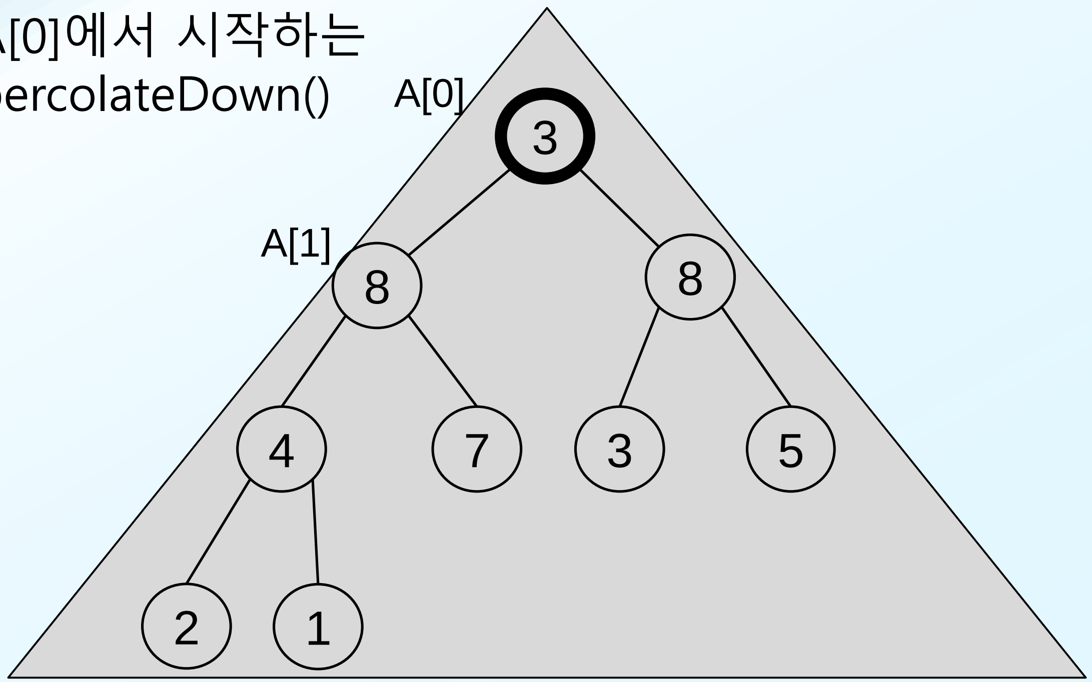


루트에서 시작하여
조정하면서 아래로 내려가는 작업을
PercolateDown이라 한다

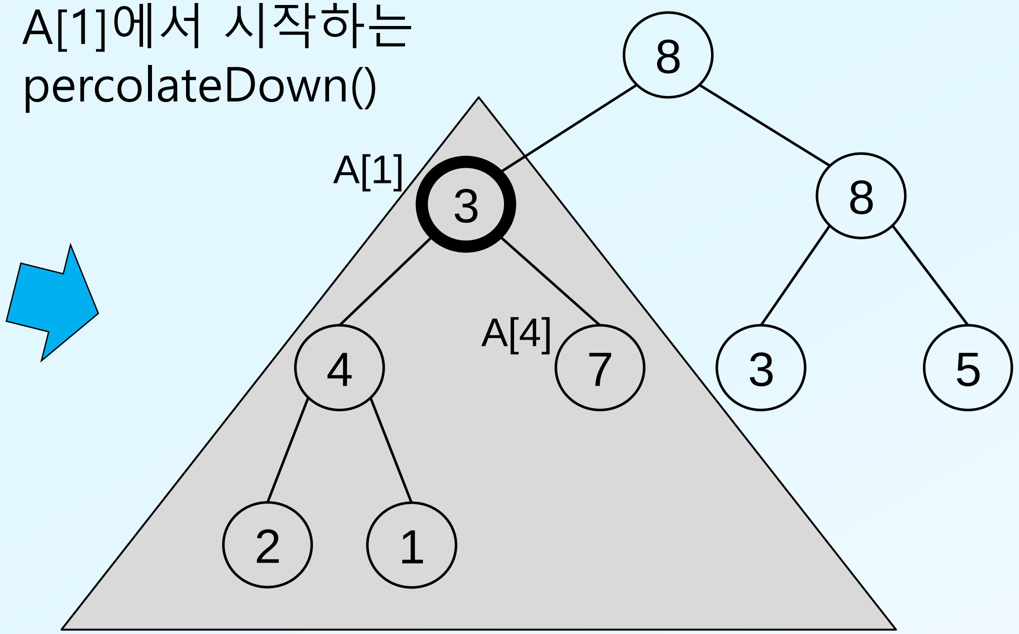
스며내리기

percolateDown()의 재귀적 관점

A[0]에서 시작하는
percolateDown()



A[1]에서 시작하는
percolateDown()



크기는 다르지만 똑같은
percolateDown 문제를 만났다

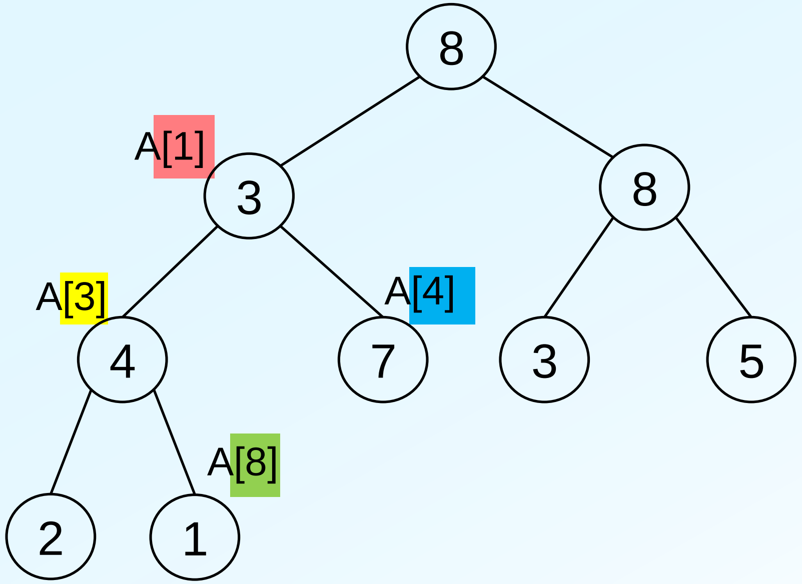
deleteMax():

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

알고리즘 percolateDown()과 deleteMax()

```
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
```



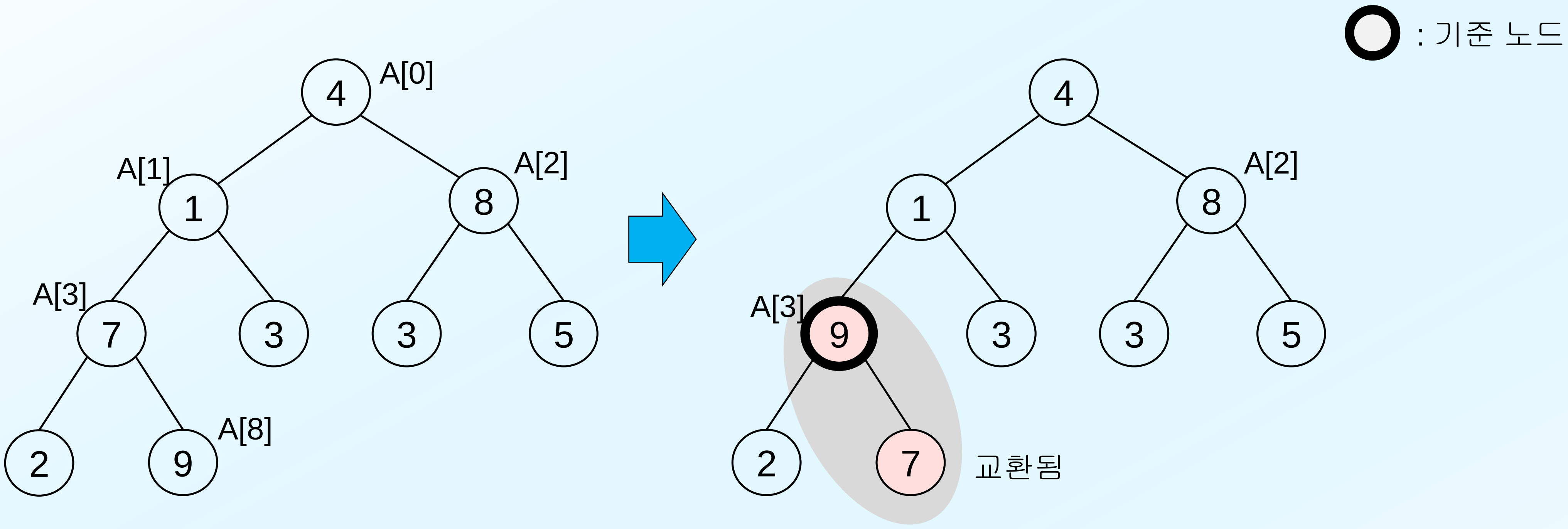
삭제 작업의 수행 시간

한 번의 percolateDown이 전부: $O(\log n)$

최악의 경우: $\Theta(\log n)$

최선의 경우: $\Theta(1)$

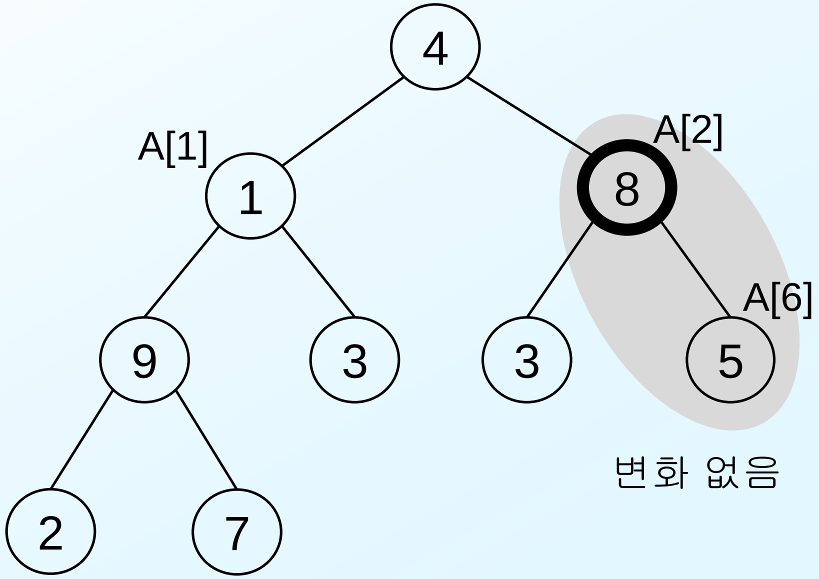
힙 만들기



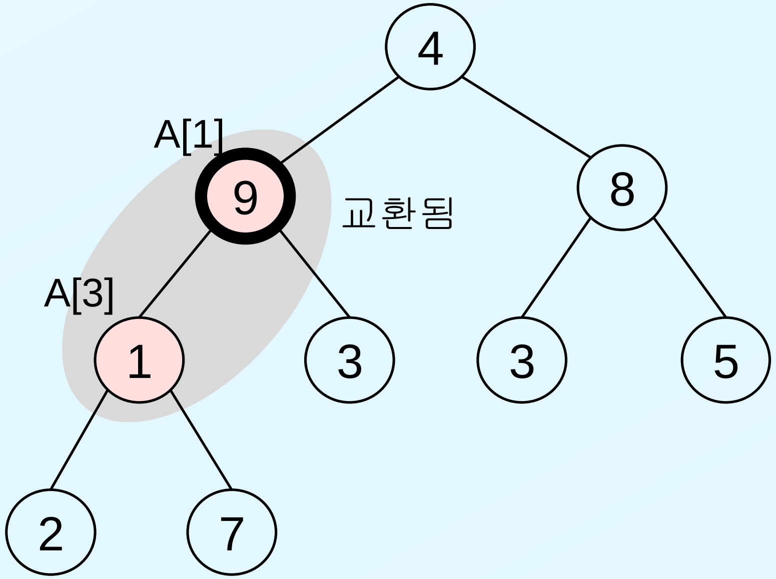
A[0]	A[1]	A[2]	· · ·					A[8]
4	1	8	7	3	3	5	2	9

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

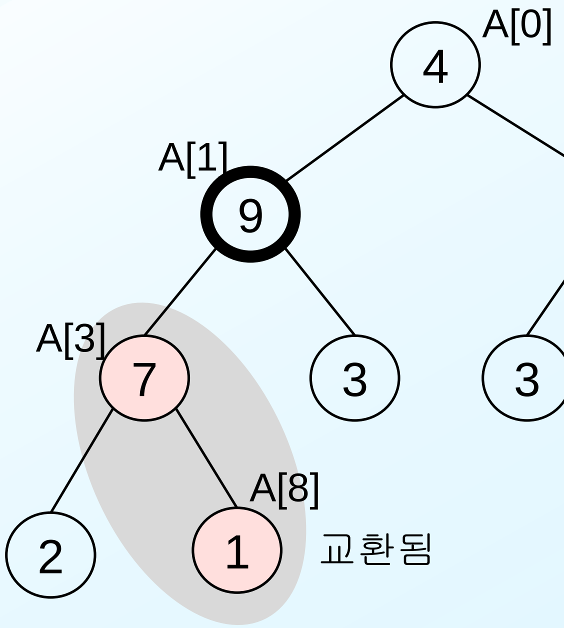
아무렇게나 저장된 배열의 예



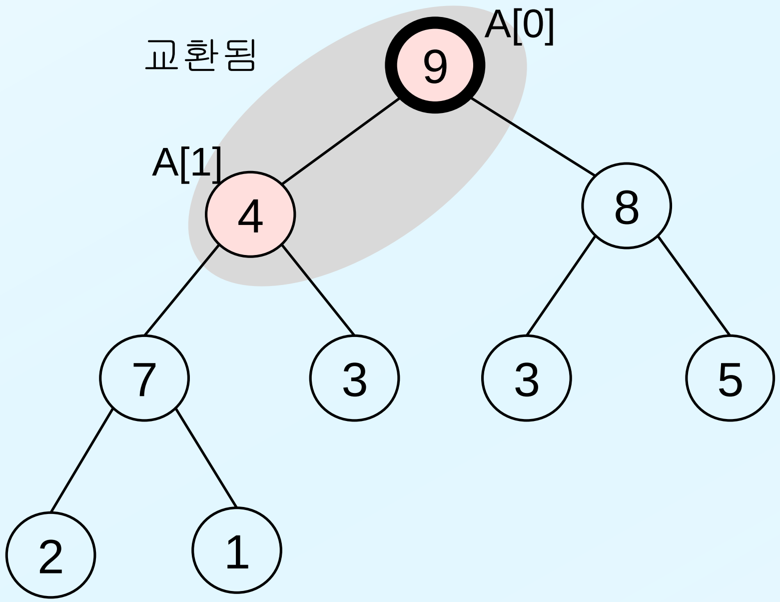
		A[2]				A[6]		
4	1	8	9	3	3	5	2	7



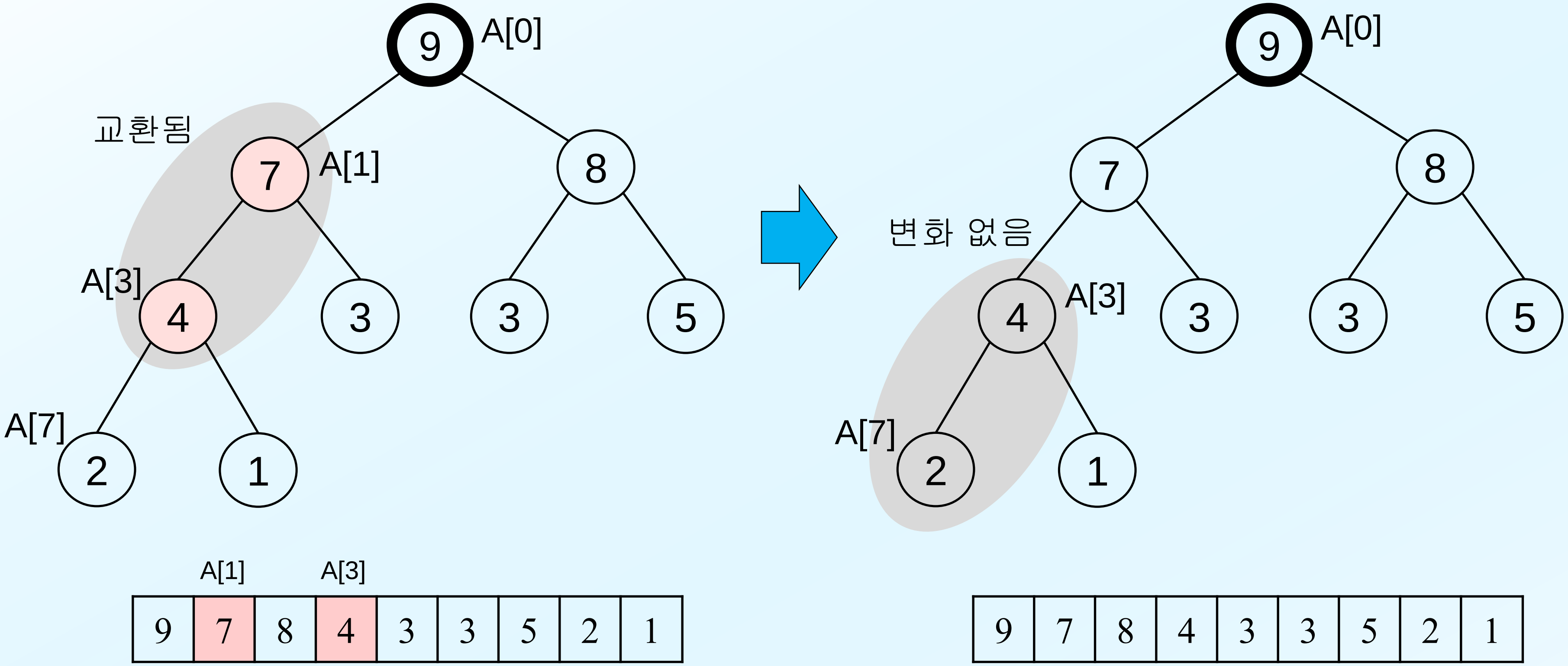
	A[1]		A[3]					
4	9	8	1	3	3	5	2	7



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

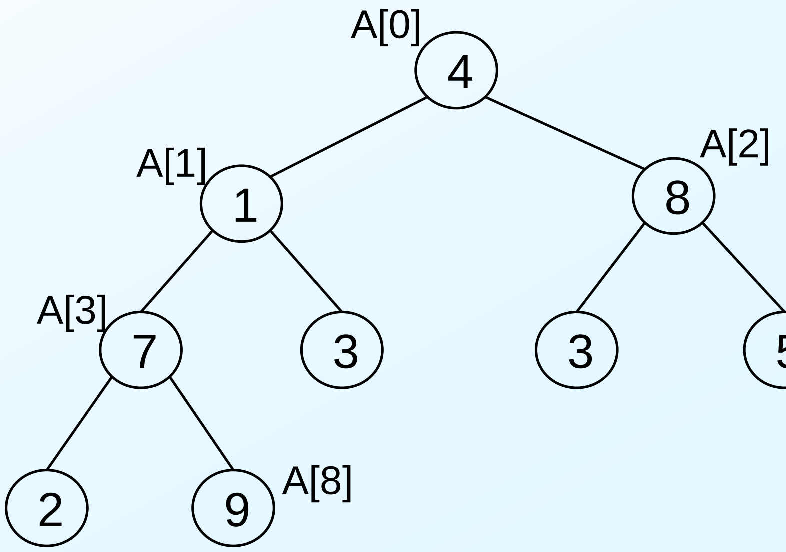


A[0]	A[1]							
9	4	8	7	3	3	5	2	1

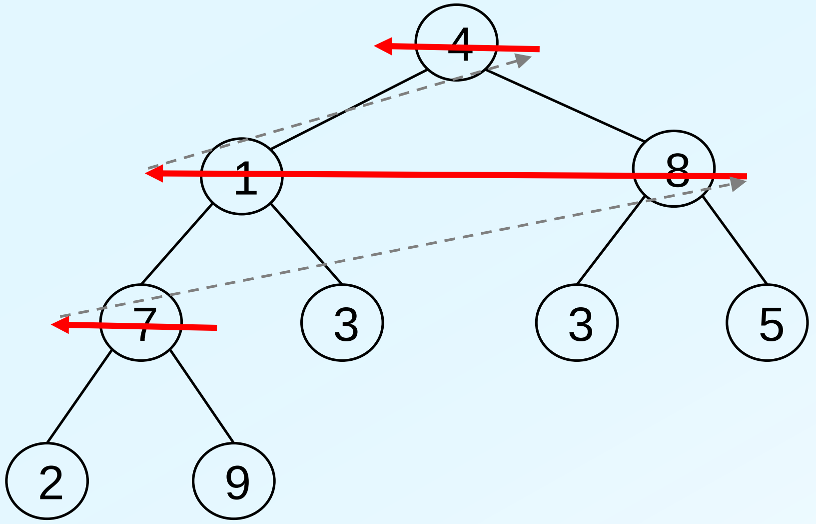


A[0]
교환됨
A[1]
A[3]
A[7]
A[1] A[3]
9 7 8 4 3 3 5 2 1
A[0]
변화 없음
A[3]
A[7]
9 7 8 4 3 3 5 2 1

buildHeap에서 percolateDown 순서



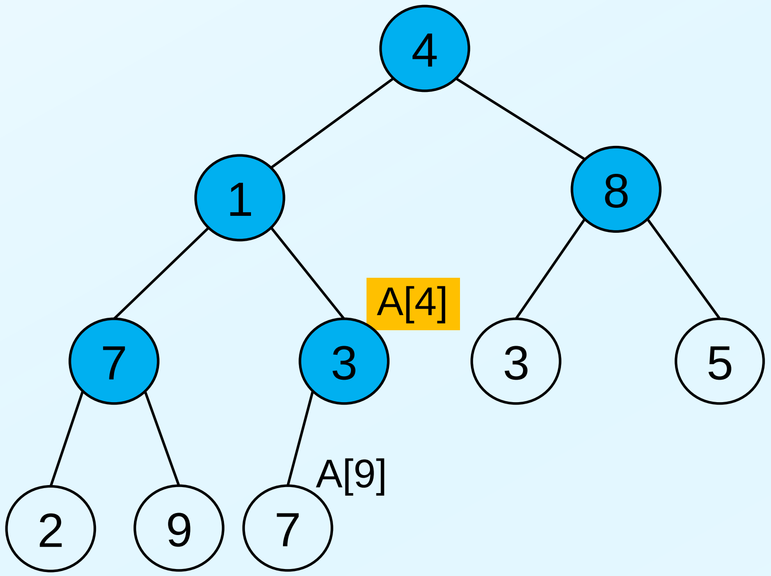
A[ ]	4	1	8	7	3	3	5	2	9



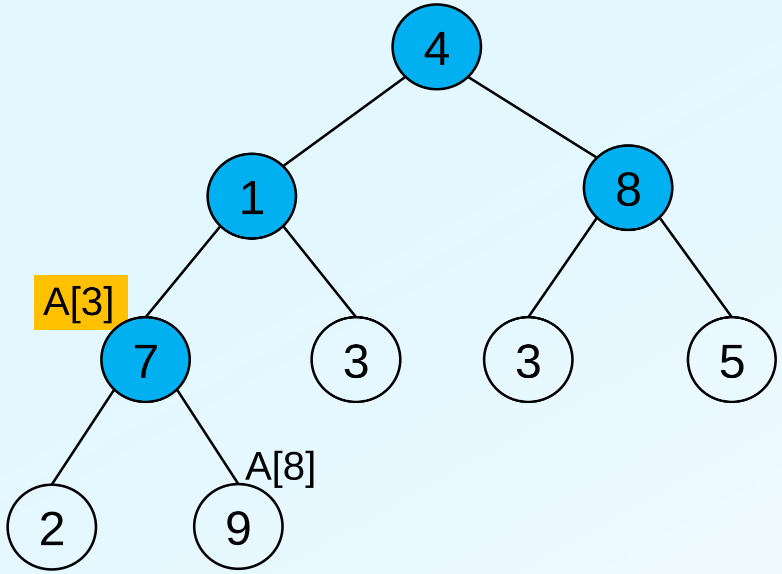
PercolateDown 순서 (기준 원소의 순서)

알고리즘 buildHeap()

◀ A[0... n−1]을 힙특성을 만족하도록 만든다
buildHeap(A[]):
 for i ← (n-2)/2 **downto** 0
  percolateDown(A, i)



예 1



예 2

buildHeap()의 수행 시간

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

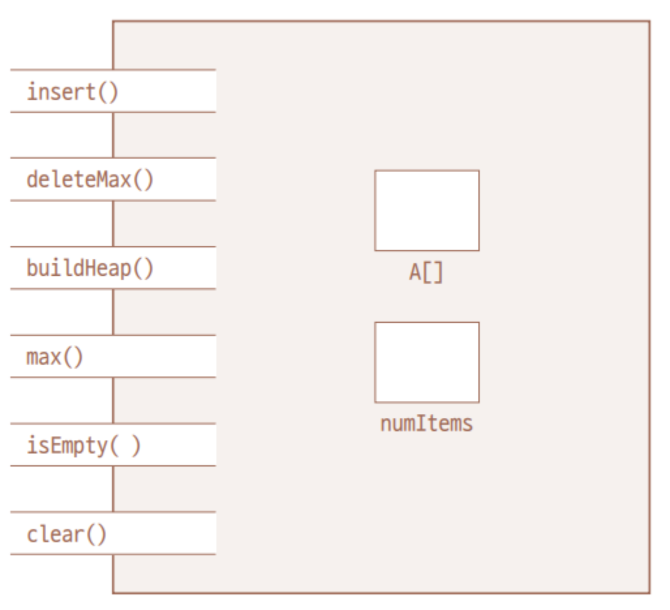
```
max():
        return A[0]
```

```
isEmpty():
    if (numItems = 0)
        return true
    else
        return false
```

```
clear():
        numItems ← 0
```

4. Java 구현

객체 구조




```
public interface PQInterface<E> {
        public void insert(E newItem) throws Exception;
        public E deleteMax() throws Exception;
        public E max() throws Exception;
        public void buildHeap();
        public boolean isEmpty();
        public void clear();
}
```

```
public class Heap<E extends Comparable> implements PQInterface<E>{
        private E[] A;
        private int numItems;
        public Heap(int n) {
                A = (E[]) new Comparable[n];
                numItems = 0;
        }
        public Heap(E[] B, int numElements) { // 바깥에서 만들어진 배열을 받는다
                A = B; // 배열 레퍼런스 복사
                numItems = numElements;
        }
        public void insert(E newItem) throws PQException { ❶
        // 힙 A[0...numItems-1]에 원소 newItem을 삽입한다(추가한다)
                if (numItems < A.length) {
                        A[numItems] = newItem;
                        percolateUp(numItems);
                        numItems++;
                } else throw new PQException("Overflow in insert()"); ❷
        }
        public E deleteMax() throws PQException {
        // 힙 A[0...numItems-1]에서 최댓값을 삭제하면서 리턴한다
                if (!isEmpty()) {
                        E max = A[0];
                        A[0] = A[numItems-1];
                        numItems--;
                        percolateDown(0);
                        return max;
                } else throw new PQException("HeapErr: DeleteMax()-Underflow");
        }
…
```

1/3 of class Heap

```
public E max() throws PQException {
        if (!isEmpty()) {
                return A[0];
        } else throw new PQException("HeapErr: Max()-Empty!");
}
public void buildHeap() {
        if (numItems >= 2)
                for (int i = (numItems-2)/2; i >= 0; i--)
                        percolateDown(i);
}
public boolean isEmpty() { // 힙이 비어있는지 알려준다
        return numItems == 0;
}
public void clear() {
        A = (E[]) new Comparable[A.length];
        numItems = 0;
}
private void percolateUp(int i) {
// A[i]에서 시작해서 힙성질을 만족하도록 수선한다
// A[0...i-1]은 힙성질을 만족하고 있음
        parent = (i-1)/2;
        // if (parent >= 0 && A[i].compareTo(A[parent]) > 0) {
        if (A[i].compareTo(A[parent]) > 0) { // 위와 같은 뜻
                E tmp = A[i];
                A[i] = A[parent];
                A[parent] = tmp;
                percolateUp(parent);
        }
}
…
```

```java
    private void percolateDown(int i) {
    // A[i]를 루트로 스며내리기
    // A[n]: last item, boundary
        int child = 2*i + 1;            // left child
        int rightChild = 2*i + 2;       // right child
        if (child <= numItems-1) {
            if (rightChild <= numItems-1
                && A[child].compareTo(A[rightChild]) < 0)
                child = rightChild;  // index of larger child
            if (A[i].compareTo(A[child]) < 0) {
            .   E tmp = A[i];
                A[i] = A[child];
                A[child] = tmp;
                percolateDown(child);
            }
        }
    }
} // End Heap<>
```


- 배열 크기를 미리 주지 않고 ArrayList<>를 키워가면서 구현할 수도 있다.
- But, 좀 더 복잡.

앞의 코드는 Exception을 포함하고 있다

이 클래스의 객체를 생성해서 사용할 때는

```
Heap h = new Heap<Integer>(5);
Try { ❸
        h.insert(1);
        h.insert(10);
        h.insert(40);
        h.insert(30);
        h.insert(5);
        h.delete();
        h.insert(50);
        h.insert(20); // 에러내기 for try-catch 테스트
        // 앞에서 HeapException 객체가 날아와서
        // 여기서부터는 수행하지 못하고 catch로 넘어감
        h.insert(100);
        h.insert(25);
} catch (PQException ex ❹) {
        System.out.println("HeapException: " + ex.getMessage());
        /* 추가로 필요한 처리가 있으면 여기서 */
}
```

```
public class PQException extends Exception {
        public PQException(String msg) {
                super(msg);
        }
}
```

수행 결과

```
HeapException: Overflow in insert()
```

앞의 PQException 대신
위 에러에 대해서 자바에서도 발생시키는 IndexOutOfBoundsException이나
이들의 상위 클래스인 Exception을 사용해도 된다.
이들은 이미 자바가 만들어놓은 클래스이므로
굳이 PQException 클래스를 따로 만들지 않아도 된다.

```
public class Heap<E extends Comparable> implements PQInterface<E>{
        …
        public void insert(E newItem) throws Exception { ❶
                if (numItems < A.length) {
                        …
                } else throw new Exception("Overflow in insert()"); ❷
        }
        …
```

```
Heap h = new Heap<Integer>(5);
Try { ❸
        …
} catch (Exception ex ❹) {
        System.out.println("HeapException: " + ex.getMessage());
        /* 추가로 필요한 처리가 있으면 여기서 */
}
```

잠재적 문제: 프로그램의 다른 곳에서도 에러가 발생했을 때 자바가 Exception을 던질 수 있음. 이 때는 사용자의 **catch**가 우선권을 가지므로 마치 Heap.***()에서 발생한 에러처럼 처리하게 된다. 즉, Heap.insert()-❷와 같은 곳에서 발생하지 않은 에러가 ❹에서 처리되는 문제가 발생. (<쉽게 배우는 자료구조, p.94>)

❶, ❷의 PQException을 던지는 부분은 그대로 두고
❹에서 Exception으로 해놓아도 ❷에서 던진 에러 객체를 받아낸다.
Exception이 모든 에러 객체의 상위 클래스이기 때문.

```
public class Heap<E extends Comparable> implements PQInterface<E>{
        …
        public void insert(E newItem) throws PQException { ❶
                if (numItems < A.length) {
                        …
                } else throw new PQException(“Overflow in insert()”); ❷
        }
        …
```

```
Heap h = new Heap<Integer>(5);
Try { ❸
        …
} catch (Exception ex ❹) { // 이렇게 해도 ❷에서 던진 객체를 받아낸다. But …
        System.out.println(“HeapException: ” + ex.getMessage());
        /* 추가로 필요한 처리가 있으면 여기서 */
}
```

똑같은 문제: 프로그램의 다른 곳에서도 에러가 발생했을 때 사용자의 **catch**가 우선권을 가지므로 마치 Heap.***()에서 발생한 에러처럼 처리하게 된다. 즉, Heap.insert()-❷와 같은 곳에서 발생하지 않은 에러가 ❹에서 처리되는 문제가 발생.