

2021-2 알고리즘

3. 정렬 알고리즘 III



한남대학교 컴퓨터공학과


5장 구성



- 퀵 정렬(2)
- 힙 정렬(1), (2)
- 기수 정렬, 계수 정렬

퀵 정렬(2)

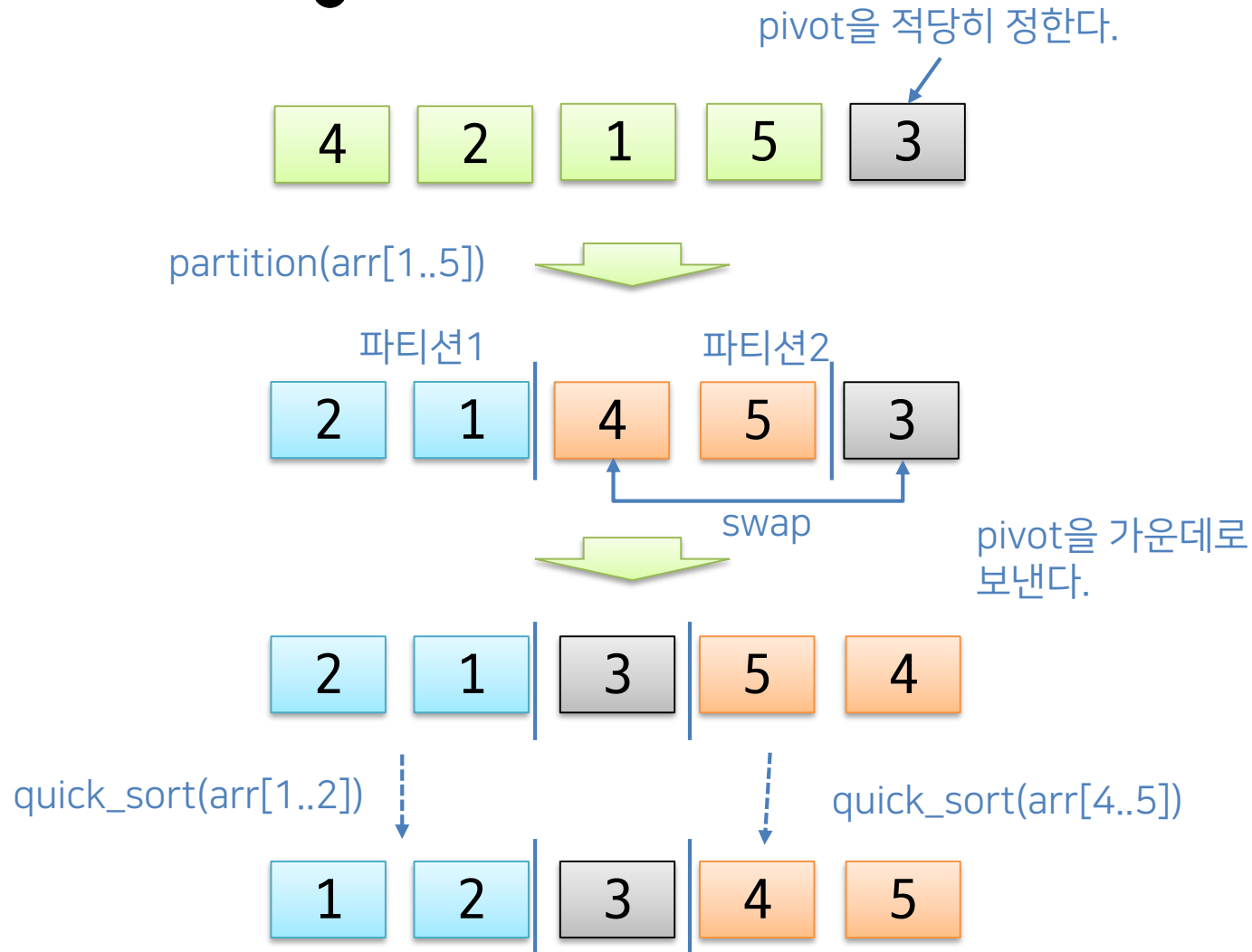


• 퀵 정렬(Quick Sort)



- In-place 퀵 정렬에서 분할하기(partition)
 - 실제로 두 개의 배열을 사용하는 것이 아니라
 - 하나의 배열을 논리적으로 두 개의 파티션으로 분할한다.
 - 따라서 병합 작업이 따로 필요 없다.
- 
- 

퀵 정렬(Quick Sort)



퀵 정렬(Quick Sort)

quickSort(A[], p , r) ▷ A[$p \dots r$]을 정렬한다

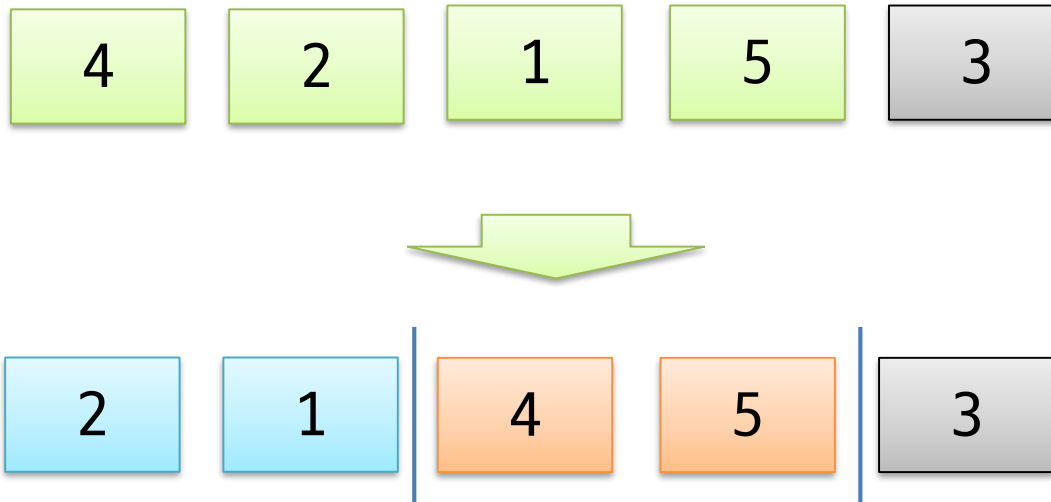
```
{  
    if ( $p < r$ ) then {  
         $q = \text{partition}(A, p, r)$ ; ▷ 분할  
        quickSort(A,  $p$ ,  $q-1$ ); ▷ 왼쪽 부분 배열 정렬  
        quickSort(A,  $q+1$ ,  $r$ ); ▷ 오른쪽 부분 배열 정렬  
    }  
}
```

partition(A[], p , r)

```
{  
    배열 A[ $p \dots r$ ]의 원소들을 A[ $r$ ]을 기준으로 양쪽으로 재배치하고  
    A[ $r$ ]이 자리한 위치를 리턴한다;  
}
```

퀵 정렬(Quick Sort)

- <Sub-Problem: in-place partitioning>



퀵 정렬(Quick Sort)

- <Sub-Solution>

- 1) 우선 pivot은 제외한다.



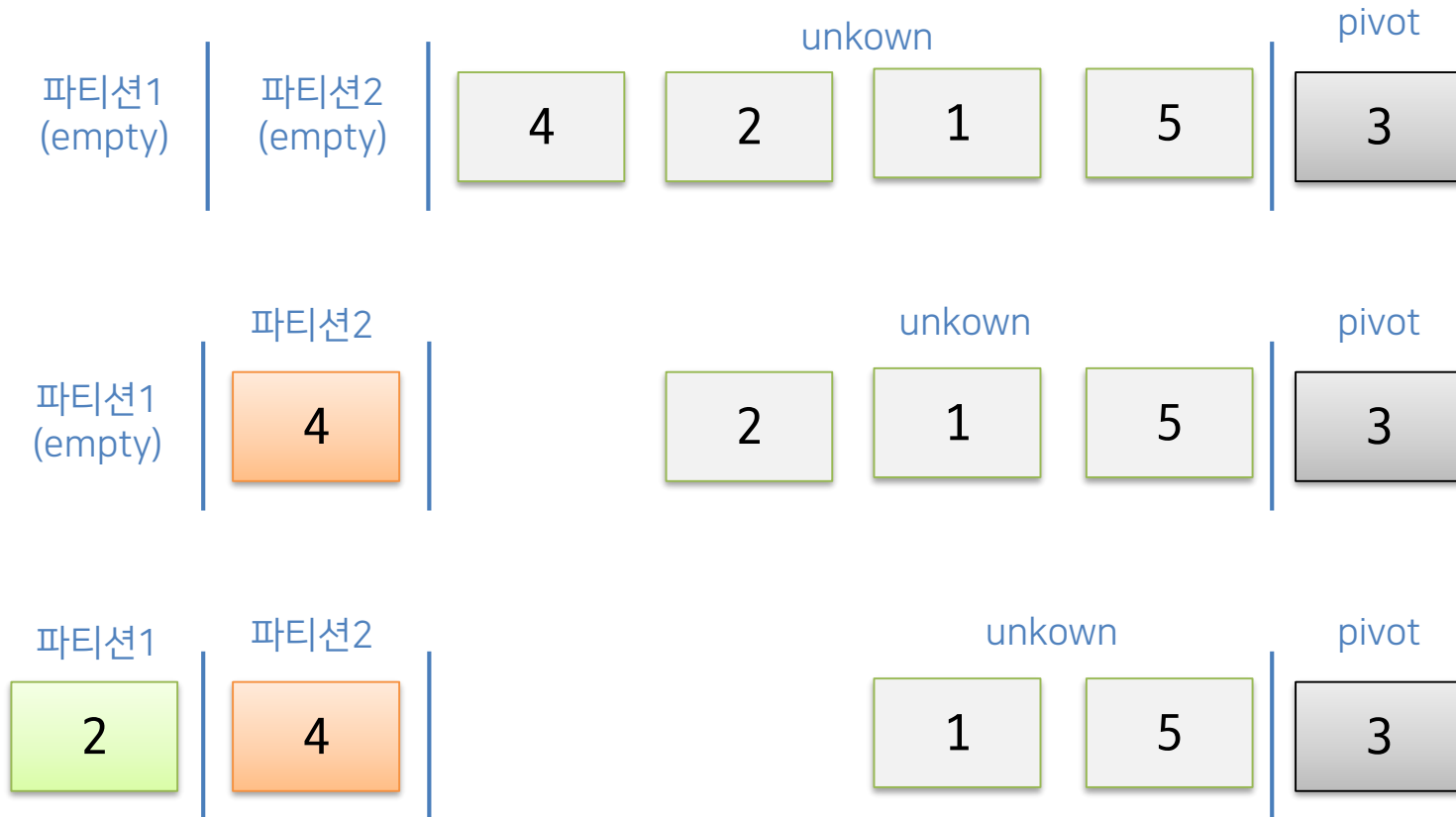
- 2) 남은 원소들을 논리적으로 세 부분으로 나눈다.
 - 파티션1: pivot보다 작은 원소들
 - 파티션2: pivot보다 큰 원소들
 - unknown: 소속이 결정되지 않은 원소들



퀵 정렬(Quick Sort)

- <Sub-Solution>

- 3) unknown에서 가장 앞의 원소를 pivot과 비교해서
- pivot보다 작으면 파티션1, 아니면 파티션2에 넣는다.



퀵 정렬(Quick Sort)

- <Sub-Solution>

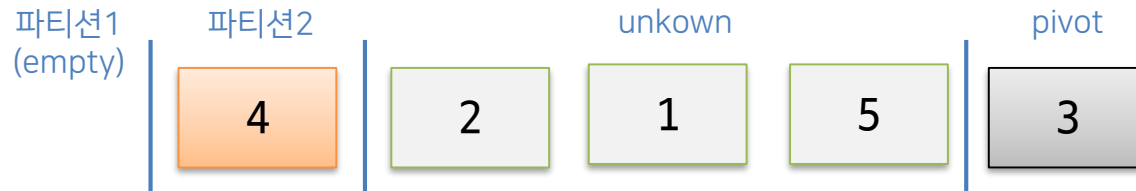
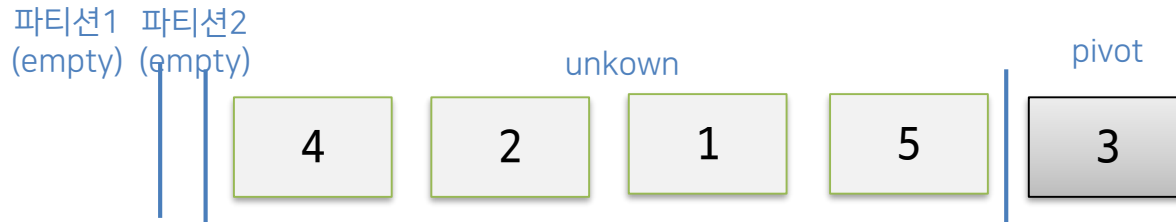
- 논리적으로는 각 원소의 소속을 옮기고 있지만,
- 실제로 구현할 때는 파티션의 구분선을 이동시킨다.



퀵 정렬(Quick Sort)

- <Sub-Solution>

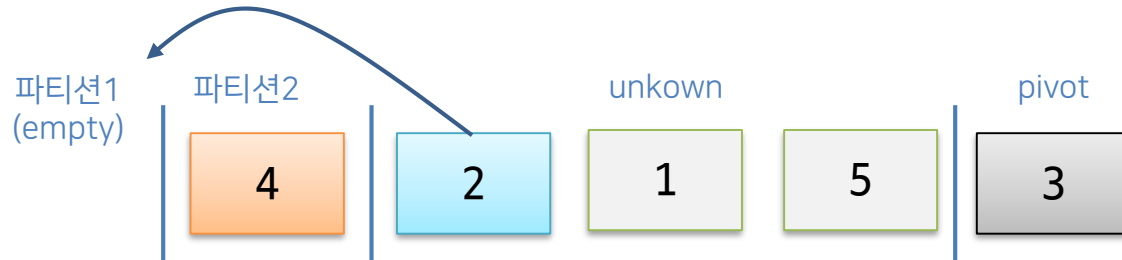
- 원소가 어느 파티션에 속할지만 정해지면
- 그 안에서의 위치는 관계 없다는 점을 이용한다.
- unknown의 첫 번째 원소가 pivot보다 큰 경우 → 파티션2
 - 두 번째 구분선을 한 칸 옮김(파티션2 크기 1 증가)



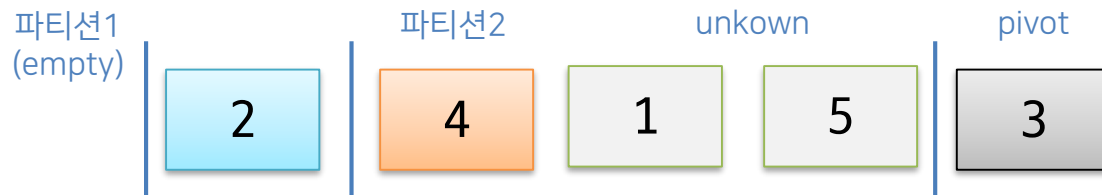
퀵 정렬(Quick Sort)

- <Sub-Solution>

- unknown의 첫 번째 원소가 pivot보다 작은 경우 → 파티션1



- 1) 파티션2의 첫 번째 원소와 unknown의 첫 번째 원소를 교환

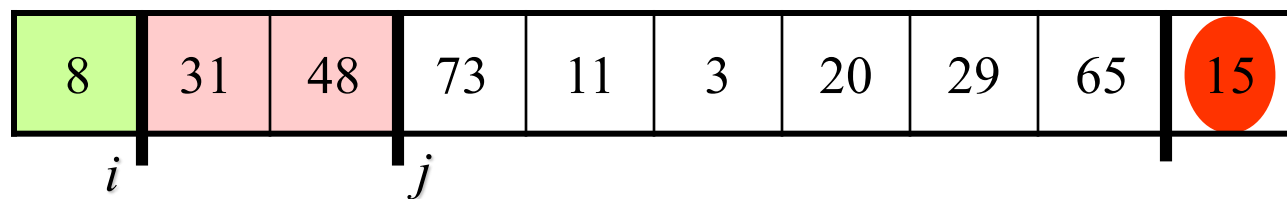
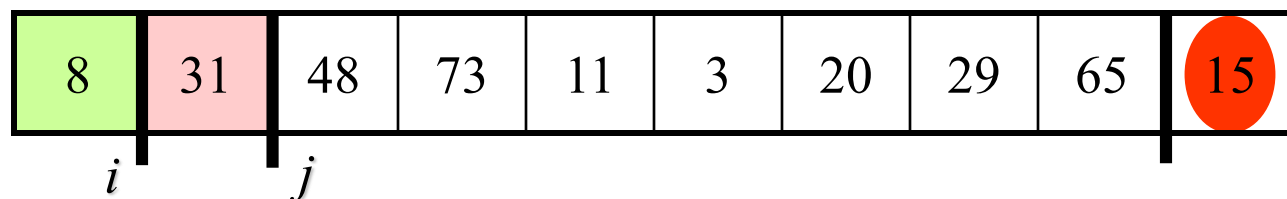
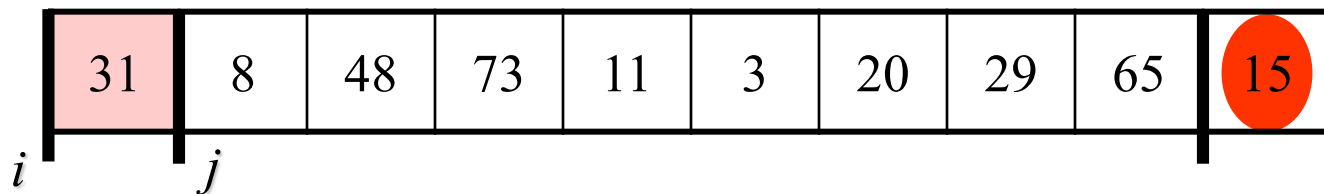
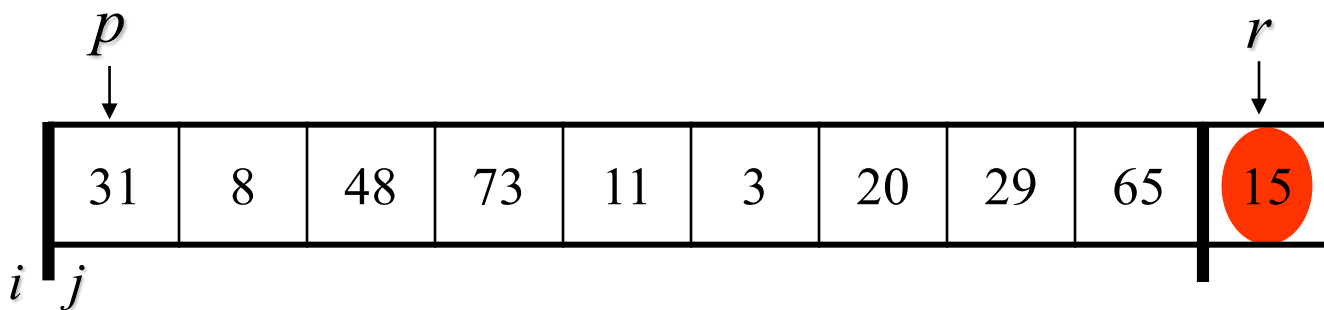


- 2) 두 구분선을 모두 한 칸 씩 옮김(파티션1 크기 1 증가, 파티션2는 shift)

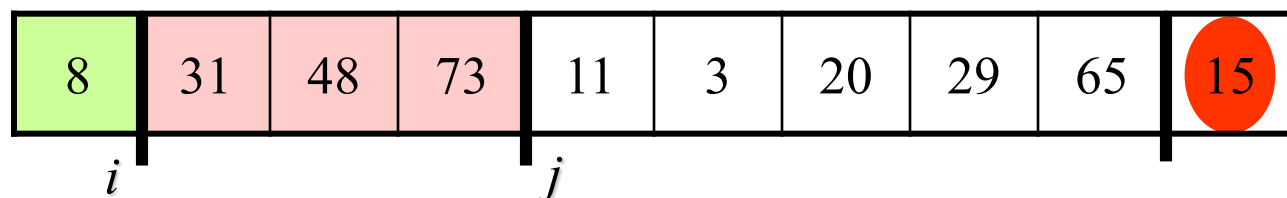


분할

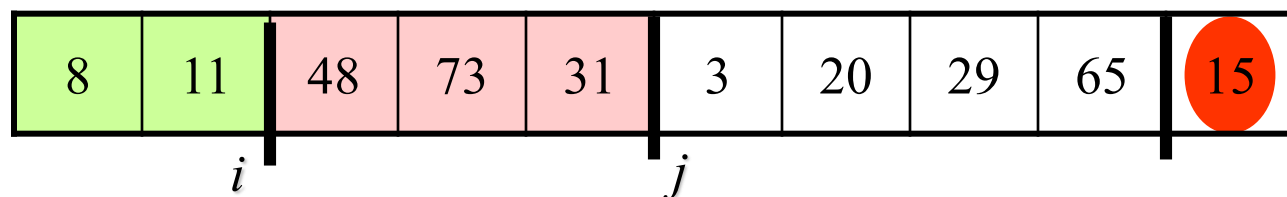
(partition)



— (a)

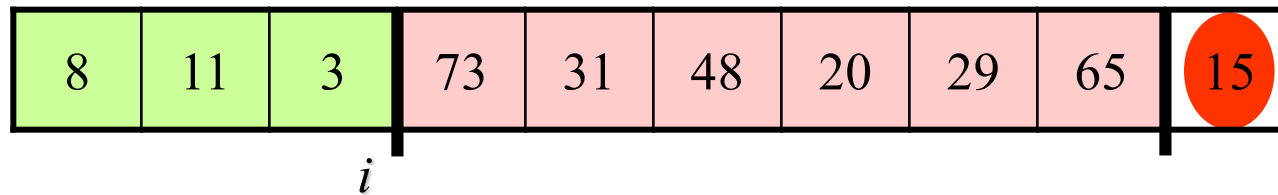
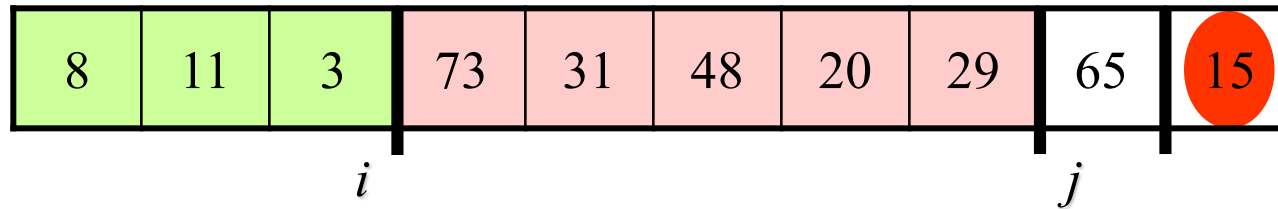
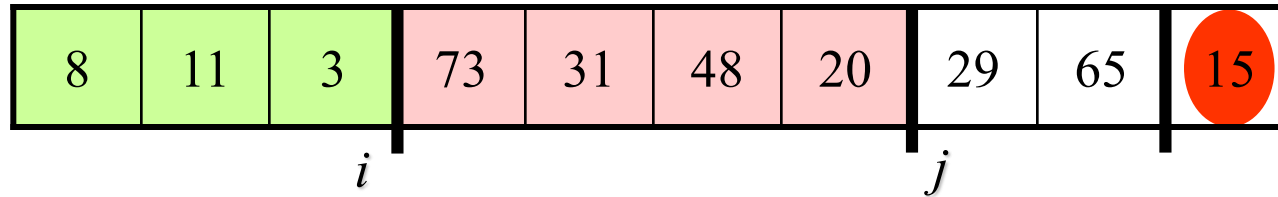
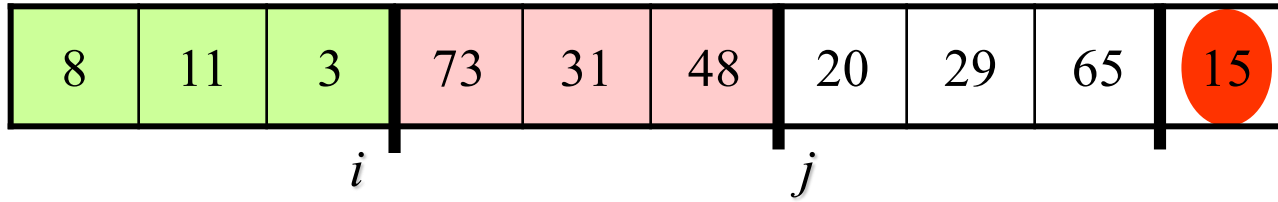


— (b)

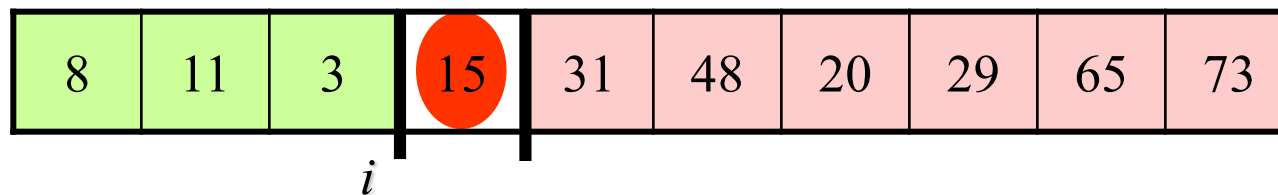


— (c)

분할 (partition)



— (d)



— (e)

연습문제

- 아래 입력에 대한 퀵 정렬의 분할 과정(in-place)을 직접 시뮬레이션해 보자.



퀵 정렬(Quick Sort)

- In-place partitioning(1 .. n)

```
partition(arr[1..n]) {  
    // i: 파티션2의 첫 번째 인덱스  
    // j: unknown의 첫 번째 인덱스  
    pivot <- arr[n]  
    i <- 0  
    j <- 1  
    while j <= n-1 {  
        // arr[j]를 파티션1 또는 파티션2로 이동시킴  
        if (arr[j] > pivot) {  
            j++;  
        } else {  
            swap arr[i], arr[j]  
            i++; j++;  
        }  
    }  
    // pivot을 두 파티션 사이로 옮겨주고 위치를 리턴한다.  
    swap arr[i], arr[n]  
    return i  
}
```


퀵 정렬(Quick Sort)

- In-place partitioning(1 .. n)
 - j는 어떤 경우에도 1 증가하므로 for문을 사용한다.

```
partition(arr[1..n]) {  
    // i: 파티션2의 첫 번째 인덱스  
    // j: unknown의 첫 번째 인덱스  
    pivot <- arr[n]  
    i <- 0  
    for j from 1 to n-1 {  
        if (arr[j] < pivot) {  
            swap arr[i], arr[j]  
            i++;  
        }  
    }  
    swap arr[i], arr[n]  
    return i  
}
```

퀵 정렬(Quick Sort)

- In-place partitioning(**p** .. **r**)

```
partition(arr[p..r]) {  
    i <- 0  
    for j from p to r-1 {  
        if (arr[j] < arr[r]) {  
            swap arr[i], arr[j]  
            i++;  
        }  
    }  
    swap arr[i], arr[r]  
    return i  
}
```

힙 정렬(1)

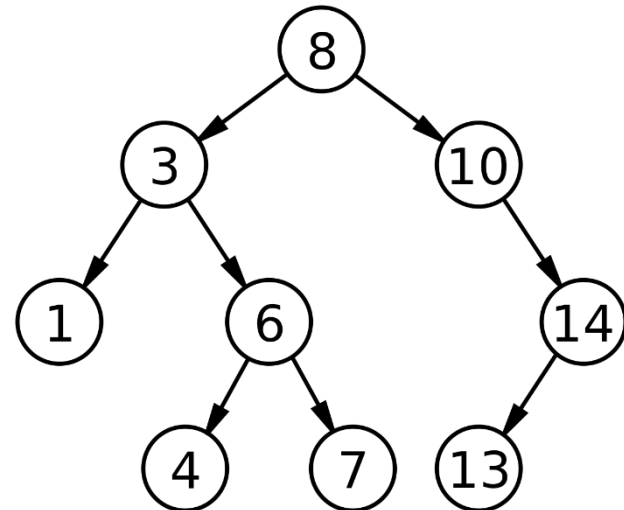
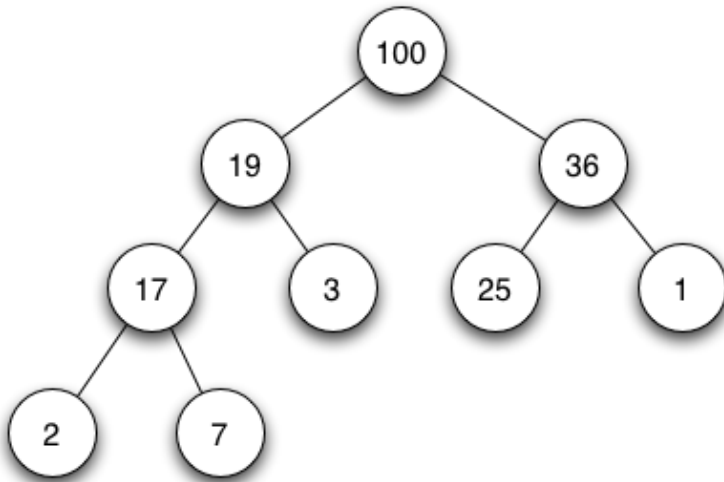


정렬 알고리즘(Sorting Algorithms)

- 기본 정렬 알고리즘: $O(n^2)$
 - 버블 정렬(Bubble Sort)
 - 선택 정렬(Selection Sort)
 - 삽입 정렬(Insertion Sort)
- 고급 정렬 알고리즘: $O(n \log n)$
 - 병합 정렬(Merge Sort)
 - 퀵 정렬(Quick Sort)
 - **힙 정렬(Heap Sort)**
- 기타:
 - 셸 정렬(Shell Sort)
 - 기수 정렬(Radix Sort)
 - 계수 정렬(Counting Sort)
 - 등

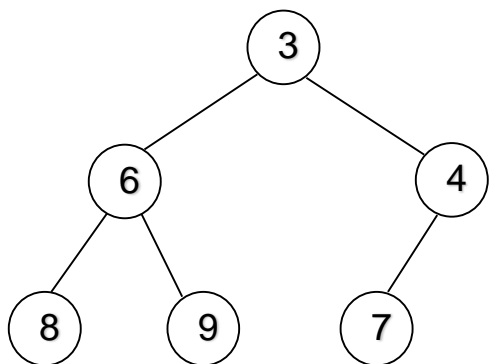
힉(heap)

- 이진 트리(binary tree)에서
 - leaf node: 자식이 없는 노드
 - inner node: leaf가 아닌 노드
- 완전 이진 트리(Complete binary tree)
 - 모든 inner node는 2개의 자식을 갖는다.
 - 가장 깊은 레벨의 노드들은 왼쪽부터 채워진다.

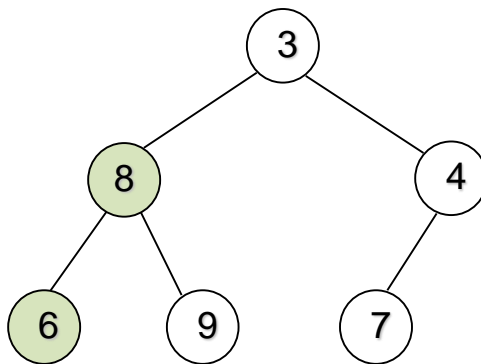


힉(heap)

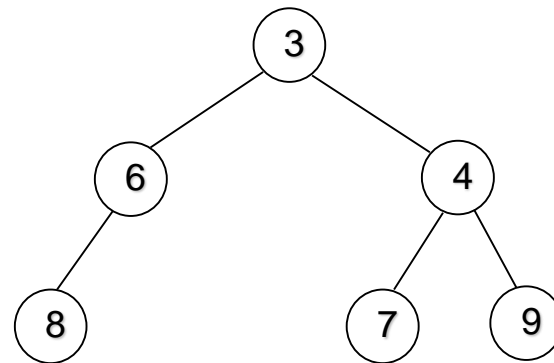
- 힉(heap)의 정의
 - 아래 성질을 만족하는 완전 이진 트리
 - 부모 노드의 값은 자식 노드의 값보다 작거나 같다.



힉



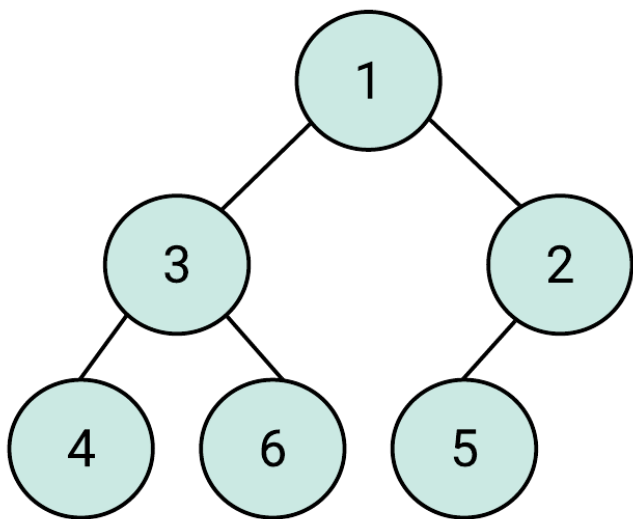
힉 아님



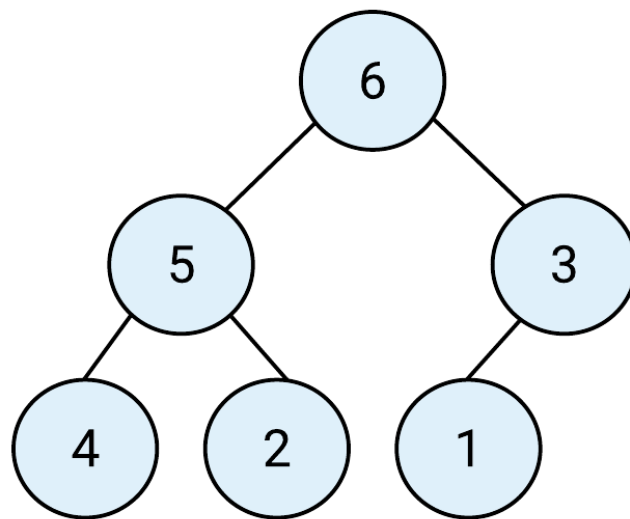
힉 아님

힉(heap)

- Min Heap: 부모 노드의 값은 자식 노드의 값보다 작거나 같다.
- Max heap: 부모 노드의 값은 자식 노드의 값보다 크거나 같다.



Min heap

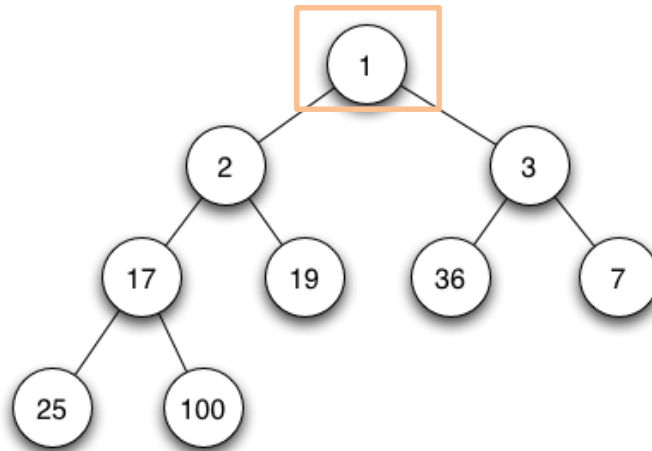


Max Heap

힉(heap)

- **힉heap의 성질**

- 항상 루트 노드는 최솟값이 된다.
- 즉, 항상 최솟값을 상수 시간 $O(1)$ 에 찾을 수 있다.



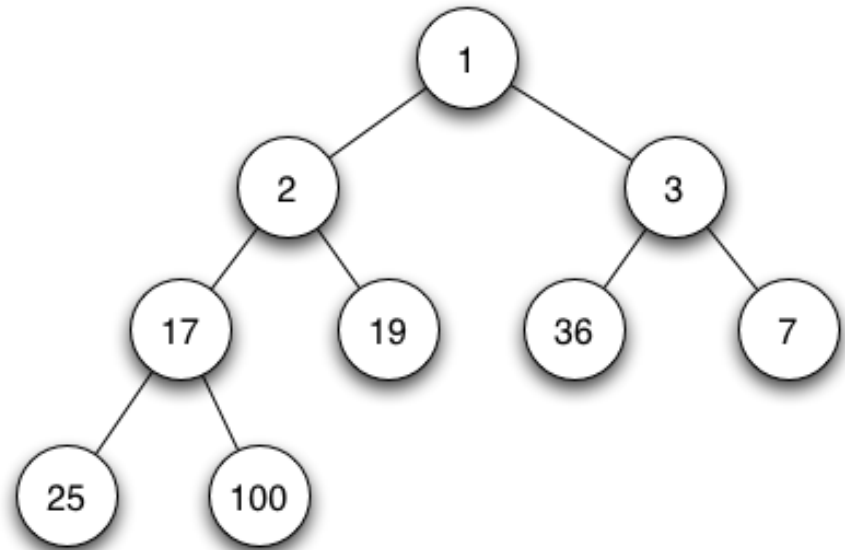
- **힉 정렬(Heap Sort)**

- 주어진 배열을 힉으로 만든 다음,
- 차례대로 하나씩 힉에서 제거함으로써 정렬한다.

힙(heap)

- 힙heap의 표현: 포인터/참조

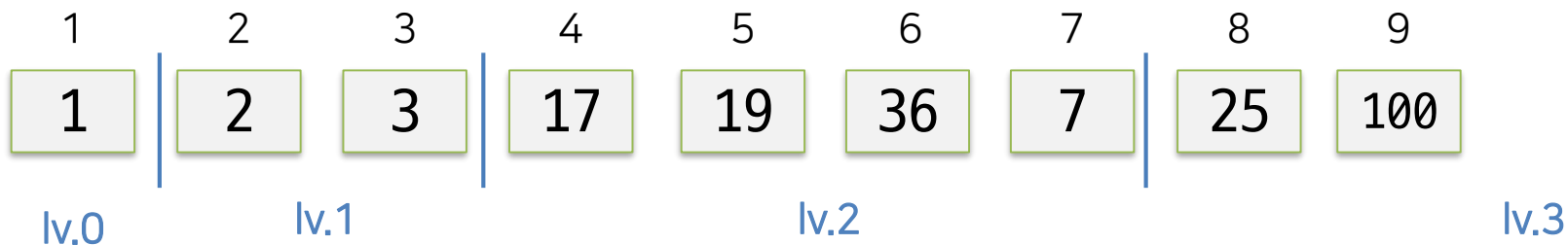
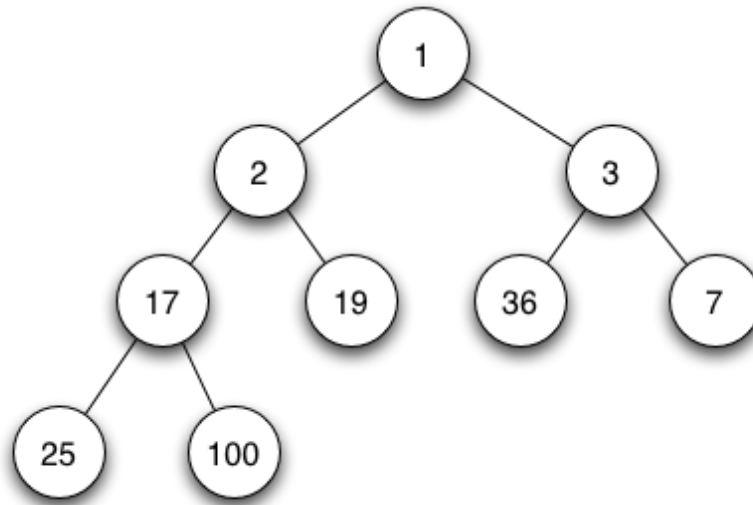
```
struct heap {  
    int key;  
    struct heap* left;  
    struct heap* right;  
}  
  
struct heap root;
```



힙(heap)

- 힙heap의 표현: 배열

- $A[1..n]$ 에서 $A[i]$ 의 왼쪽 자식은 $A[i*2]$, 오른쪽 자식은 $A[i*2+1]$ 이다.



힙 정렬(Heap Sort)

- 힙 정렬(Heap Sort)
 - 주어진 배열을 힙으로 만든 다음,
 - 차례대로 하나씩 힙에서 제거함으로써 정렬한다.

```
heapSort(A[1..n])
```

```
{
```

```
    A[]를 힙으로 만든다;
```

```
    힙에 원소가 없을 때까지 반복 {
```

```
        루트 노드를 제거;
```

```
        heapify(A[]); ▷ 힙 성질을 만족하도록 수선한다.
```

```
    }
```

```
}
```

힙 정렬(Heap Sort)

- Python에서는
 - `heapq` 모듈을 사용하면 쉽게 구현할 수 있다.
 - `heapq`: 힙의 배열 표현을 구현한 모듈
- `heapq` 사용 예)

```
from heapq import *  
  
data = [65, 100, 83, 77, 23, 11, 59, 96]  
heapify(data)  
print(data)
```

```
[11, 23, 59, 77, 100, 83, 65, 96]
```

힙 정렬(Heap Sort)

- Heap Sort (Python)

```
from heapq import *

def heapsort(heap):
    heapify(heap)  # data[]를 힙으로 만든다.
    while heap:
        min_val = heappop(heap)  # 최솟값을 뽑아내고 수선
        print(min_val, end=' ')

data = [65, 100, 83, 77, 23, 11, 59, 96]
heapsort(data)
```

11 23 59 65 77 83 96 100

힙 정렬(Heap Sort)

- 힙 정렬의 수행 시간

```
from heapq import *  
  
def heapsort(heap):  
    heapify(heap)    data[]를 힙으로 만든다:  $O(n\log n)$   
    while heap:  
        min_val = heappop(heap)    최솟값을 뽑아내고 수선:  $O(1) + O(\log n)$   
        print(min_val, end=' ')  
  
data = [65, 100, 83, 77, 23, 11, 59, 96]  
heapsort(data)
```

$$T(n) = n\log n + n * (1 + \log n) = O(n\log n)$$

힙 정렬(2)



힙 정렬(Heap Sort)

heapSort(A[], n)

▷ $A[1 \dots n]$ 을 정렬한다.

{

$A[]$ 를 힙(배열)으로 만든다;

힙에 원소가 없을 때까지 반복 {

루트 노드 $A[1]$ 을 제거;

▷ 마지막 원소와 교환한 후, 힙크기--;

heapify();

▷ 마지막 원소가 루트로 올라왔으므로

▷ 힙 성질을 만족하도록 수선한다.

}

}

힙 정렬(Heap Sort)

heapSort(A[], n)

▷ $A[1 \dots n]$ 을 정렬한다

{

 buildHeap(A, n); ▷ 힙 만들기

for $i \leftarrow n$ **downto** 2 {

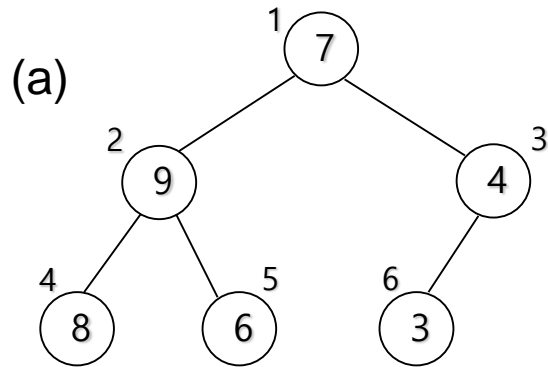
$A[1] \leftrightarrow A[i]$; ▷ 원소 교환

 heapify(A, 1, $i-1$);

 }

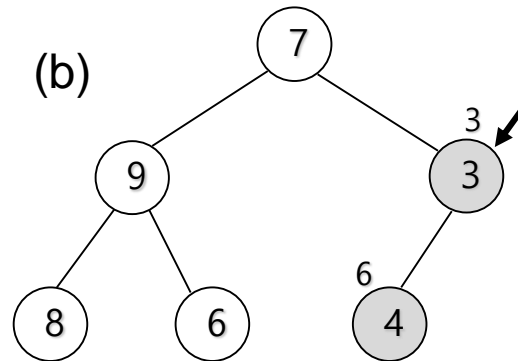
}

✓ 최악의 경우에도 $O(n \log n)$ 시간 소요!

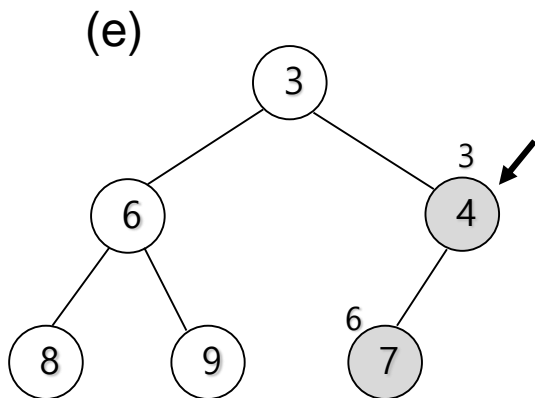
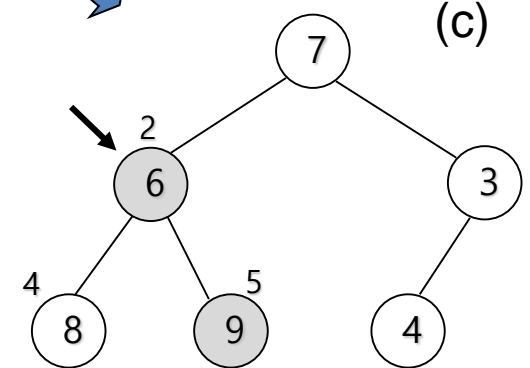


A

1	2	3	4	5	6
7	9	4	8	6	3

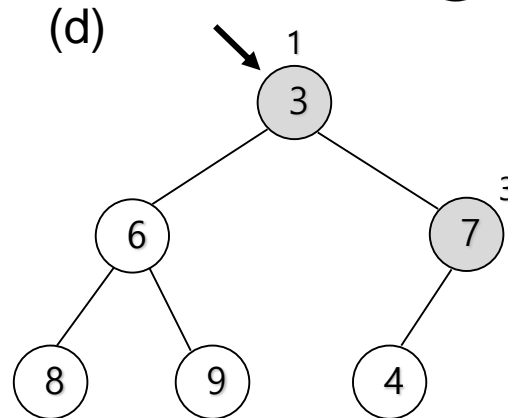


buildHeap()

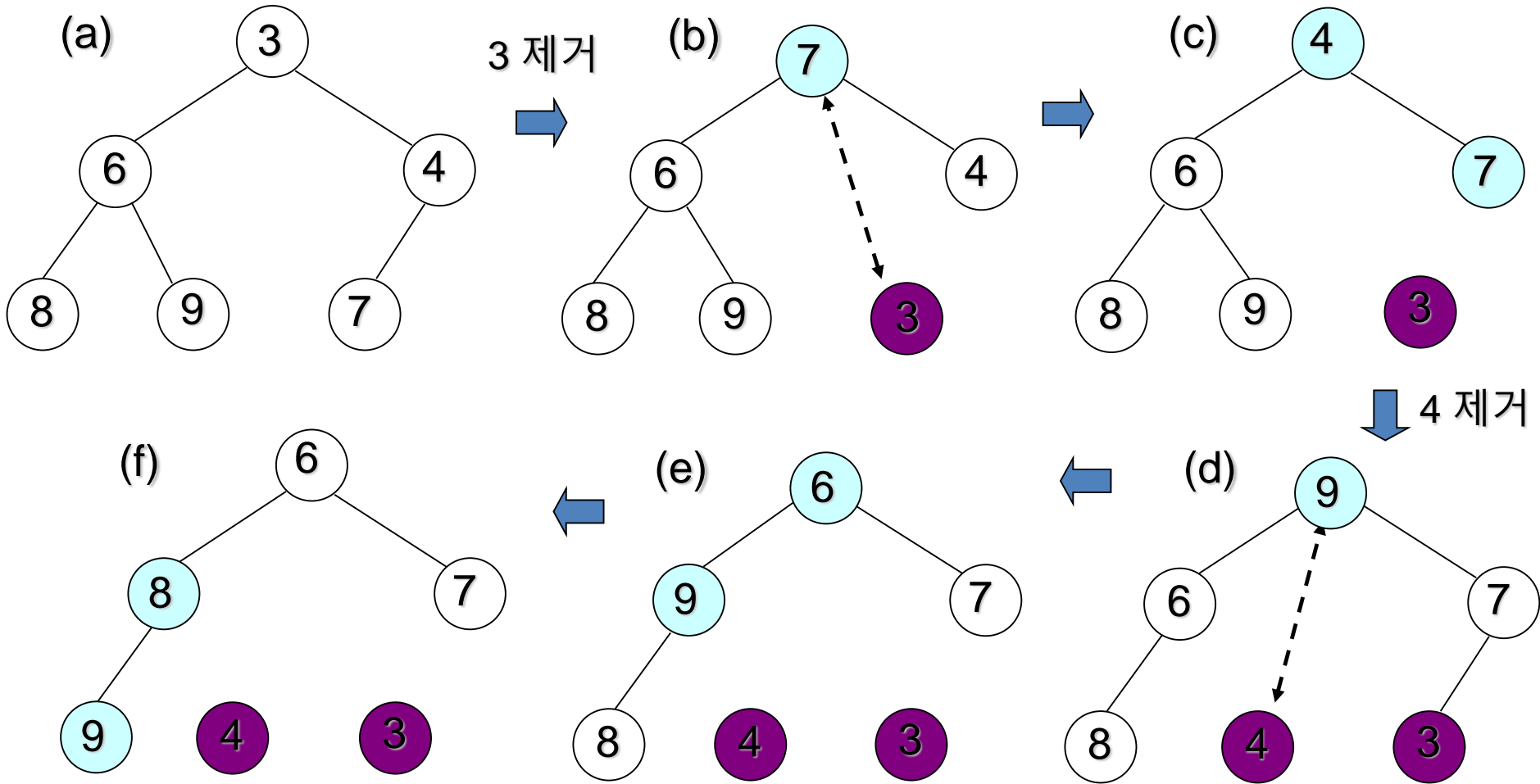


A

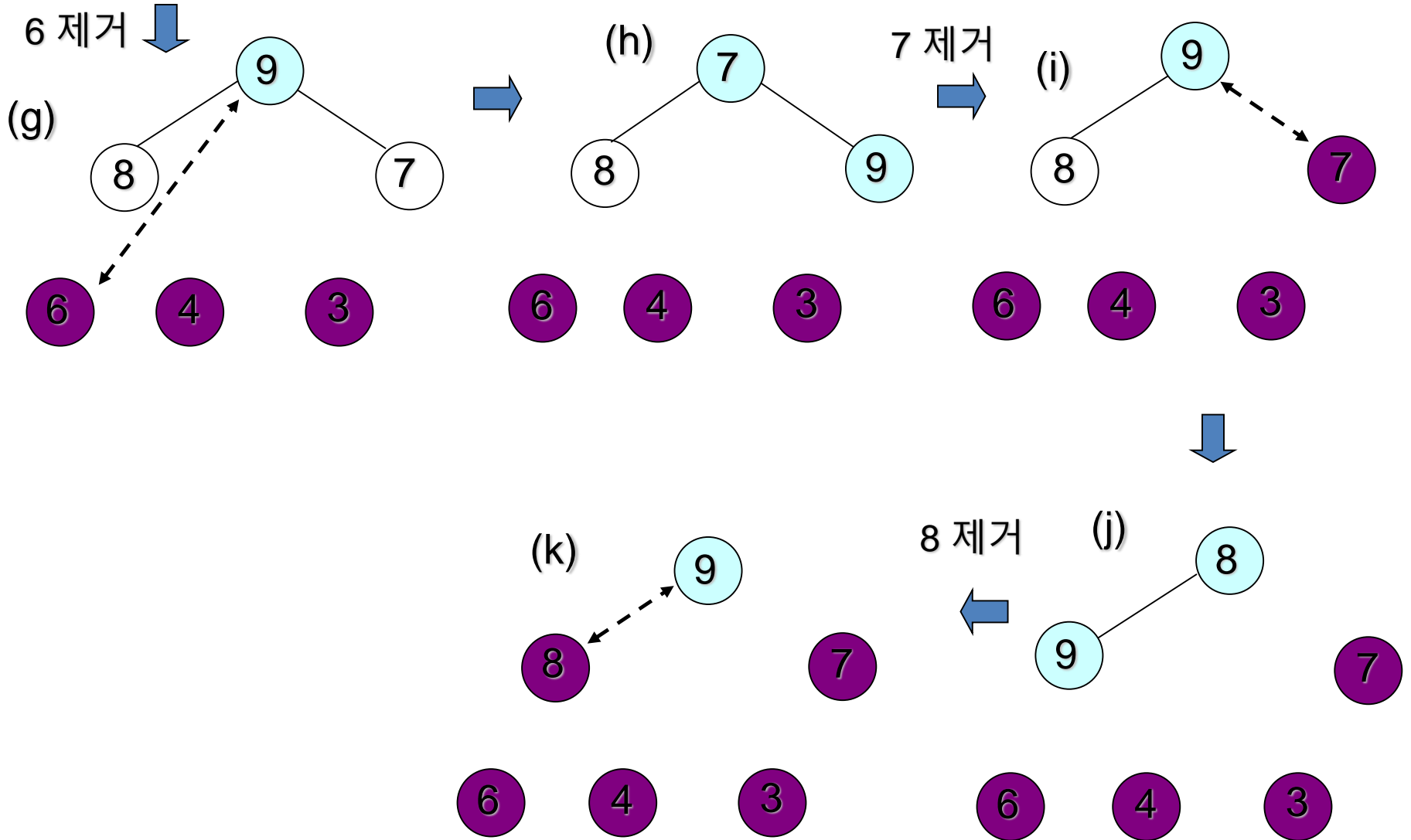
1	2	3	4	5	6
3	6	4	8	9	7



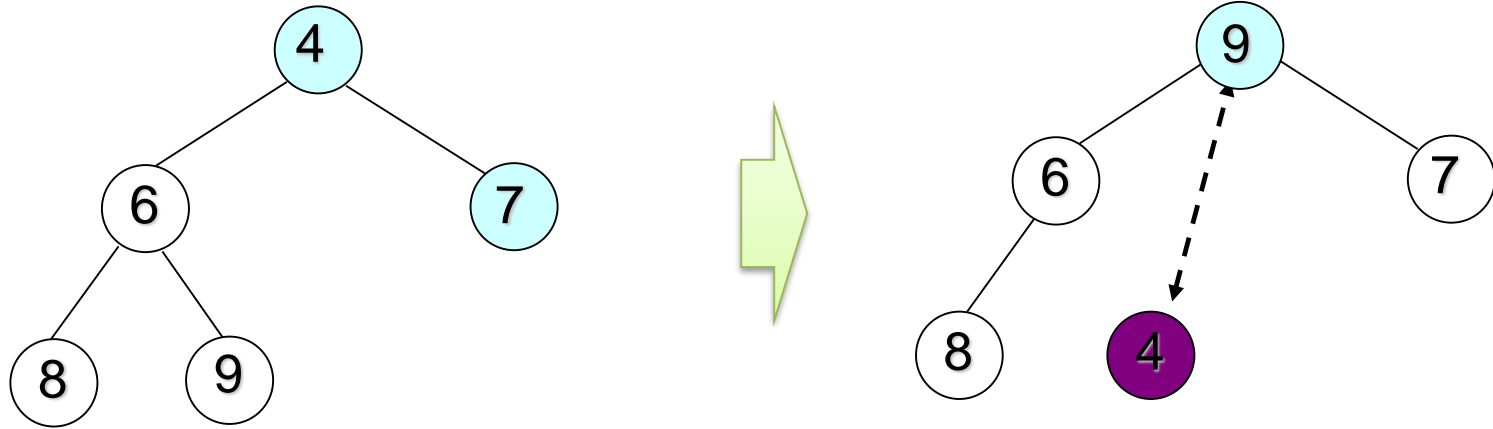
정렬



정렬



heapify()



루트 노드 4와 마지막 노드 9를 교환하고 4를 제거

→ 9와 6, 9와 7은 힙 성질을 만족하지 않는다.

→ **부모와 자식 노드를 교환**해서 힙 성질을 만족시키고, 이를 재귀적으로 반복

heapify()

heapify(A[], i, last_idx)

▷ A[i..last_idx]가 힙 성질을 만족하도록 수선한다.

{

왼쪽 자식이 더 작으면

$A[i] \leftrightarrow A[i*2];$

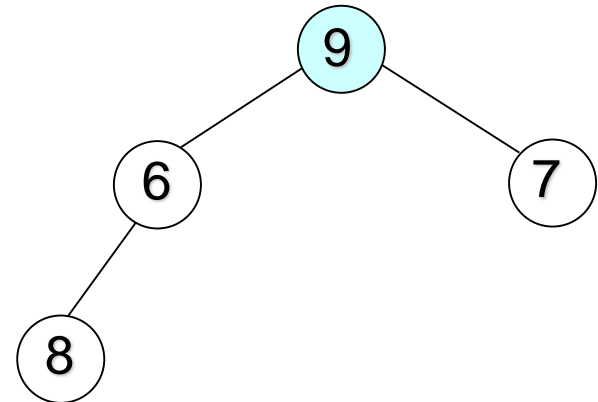
heapify(A, i*2, last_idx);

오른쪽 자식이 더 작으면

$A[i] \leftrightarrow A[i*2+1];$

heapify(A, i*2+1, last_idx);

}



기수 정렬, 계수 정렬



$\Theta(n)$ 정렬

- 두 원소를 비교하는 것을 기본 연산으로 하는 정렬의 하한선은 $\Omega(n \log n)$ 이다
- 그러나 원소들이 특수한 성질을 만족하면 $\Theta(n)$ 정렬도 가능하다
 - 계수정렬 Counting Sort
 - 원소들의 크기가 모두 $-O(n) \sim O(n)$ 범위에 있을 때
 - 기수정렬 Radix Sort
 - 원소들이 모두 k 이하의 자릿수를 가졌을 때 (k : 상수)

기수정렬 Radix Sort

radixSort($A[]$, n , k)

- ▷ 원소들이 각각 최대 k 자리수인 $A[1 \dots n]$ 을 정렬한다
- ▷ 가장 낮은 자리수를 1번째 자리수라 한다

{

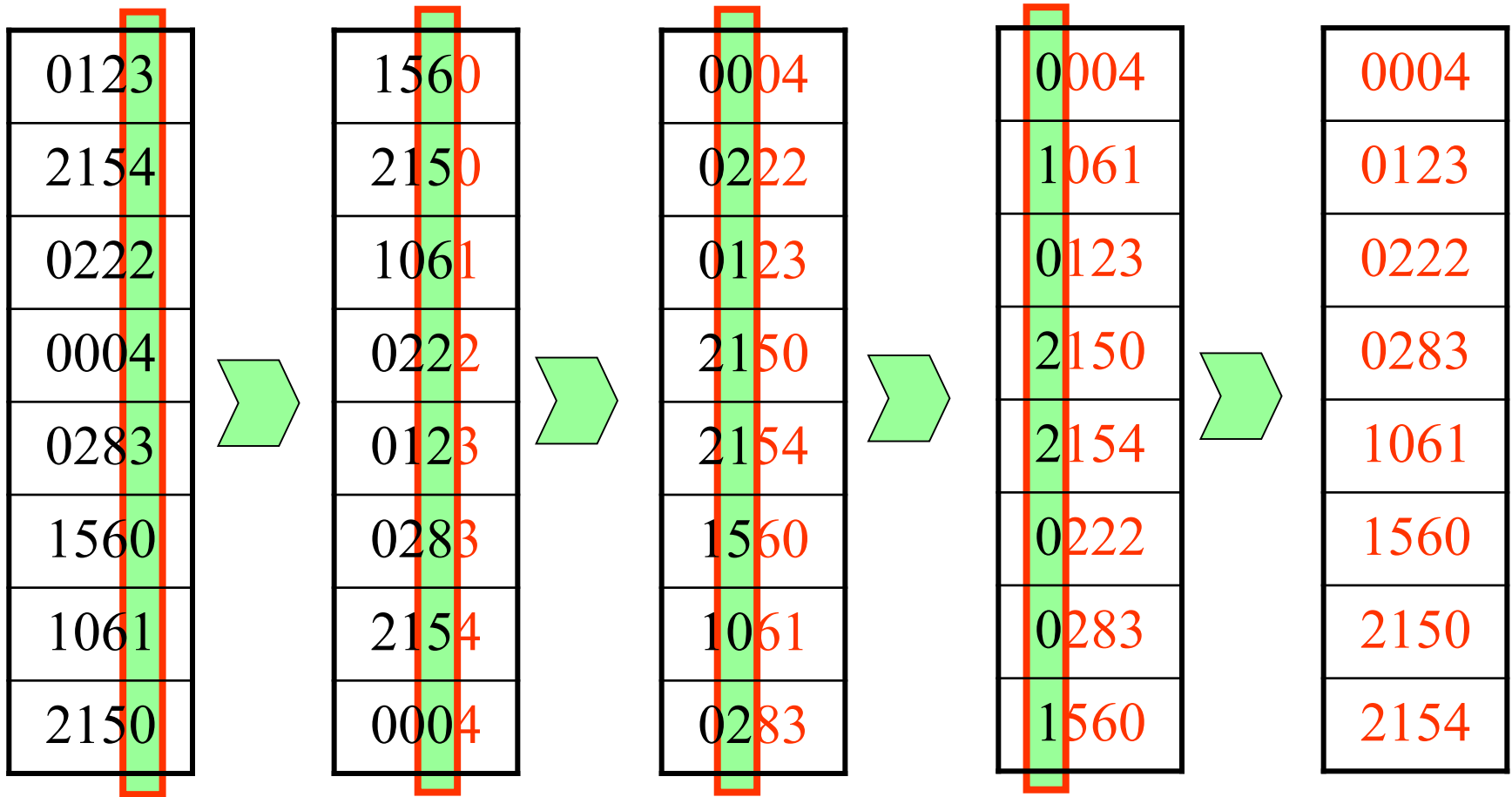
for $i \leftarrow 1$ to k

i 번째 자리수에 대해 $A[1 \dots n]$ 을 안정을 유지하면서 정렬한다;

}

✓ **안정성 정렬** Stable sort

- 같은 값을 가진 원소들은 정렬 후에도 원래의 순서가 유지되는 성질을 가진 정렬을 일컫는다.



✓ Running time: $\Theta(n)$ ← d : a constant

계수정렬 Counting Sort

countingSort(A, B, n)

▷ $A[1 \dots n]$: 입력 배열

▷ $B[1 \dots n]$: 배열 A를 정렬한 결과

{

for $i = 1$ **to** k

$C[i] \leftarrow 0$;

for $j = 1$ **to** n

$C[A[j]]++$;

▷ 이 시점에서의 $C[i]$: 값이 i 인 원소의 총 수

for $i = 1$ **to** k

$C[i] \leftarrow C[i] + C[i-1]$;

▷ 이 시점에서의 $C[i]$: i 보다 작거나 같은 원소의 총 개수

for $j \leftarrow n$ **downto** 1 {

$B[C[A[j]]] \leftarrow A[j]$;

$C[A[j]]--$;

 }

}

효율성 비교

	Worst Case	Average Case
Selection Sort	n^2	n^2
Bubble Sort	n^2	n^2
Insertion Sort	n^2	n^2
Mergesort	$n \log n$	$n \log n$
Quicksort	n^2	$n \log n$
Counting Sort	n	n
Radix Sort	n	n
Heapsort	$n \log n$	$n \log n$