



제3장

정렬

- ⦿ **Bubble sort**
  - ⦿ **Insertion sort**
  - ⦿ **Selection sort**
  - ⦿ **Quicksort**
  - ⦿ **Merge sort**
  - ⦿ **Heap sort**
  - ⦿ **Radix sort**
- 
- The diagram illustrates the classification of sorting algorithms. It features a vertical list of seven algorithms on the left, each preceded by a circular icon. To the right of the list, two light blue curly braces group the algorithms into three categories: 'simple, slow' (containing Bubble sort, Insertion sort, and Selection sort), 'fast' (containing Quicksort, Merge sort, and Heap sort), and 'O(N)' (containing Radix sort).
- ⦿ **Bubble sort**
  - ⦿ **Insertion sort**
  - ⦿ **Selection sort**
  - ⦿ **Quicksort**
  - ⦿ **Merge sort**
  - ⦿ **Heap sort**
  - ⦿ **Radix sort**
- simple, slow
- fast
- $O(N)$

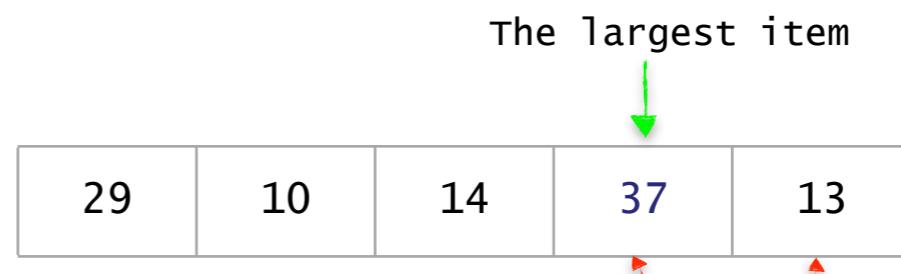
# 기본적인 정렬 알고리즘

## 각 루프마다

- 최대 원소를 찾는다
  - 최대 원소와 맨 오른쪽 원소를 교환한다
  - 맨 오른쪽 원소를 제외한다
- 하나의 원소만 남을 때까지 위의 루프를 반복

# Selection Sort

initial array



after 1<sup>st</sup> swap



after 2<sup>nd</sup> swap



after 3<sup>rd</sup> swap



after 4<sup>th</sup> swap

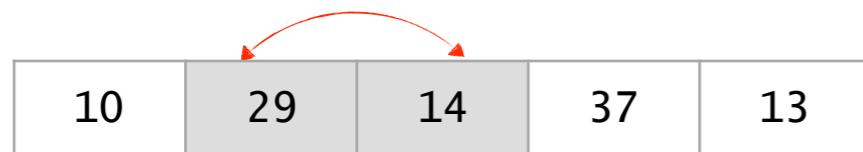


```
selectionSort(A[], n)    ▷ 배열 A[1...n]을 정렬한다.  
{  
    for last ← n downto 2 { -----①  
        A[1...last] 중 가장 큰 수 A[k]를 찾는다; -----②  
        A[k] ↔ A[last]; ▷ A[k]와 A[last]의 값을 교환 -----③  
    }  
}
```

## 실행시간:

- ①의 for 루프는  $n-1$ 번 반복
  - ②에서 가장 큰 수를 찾기 위한 비교횟수:  $n-1, n-2, \dots, 2, 1$
  - ③의 교환은 상수시간 작업
- 시간복잡도  $T(n)=(n-1)+(n-2)+\dots+2+1 = O(n^2)$

## Bubble Sort



pass 1



pass 2

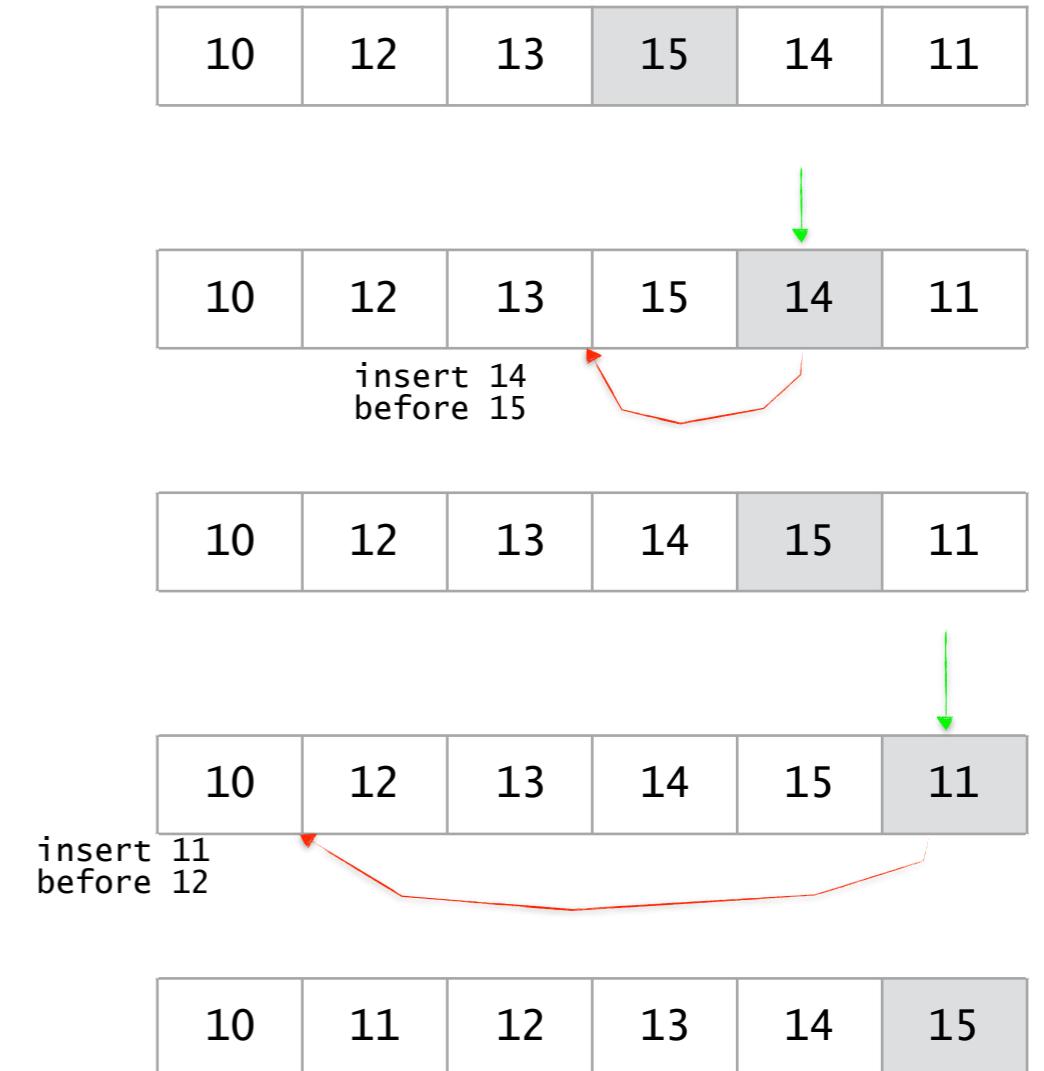
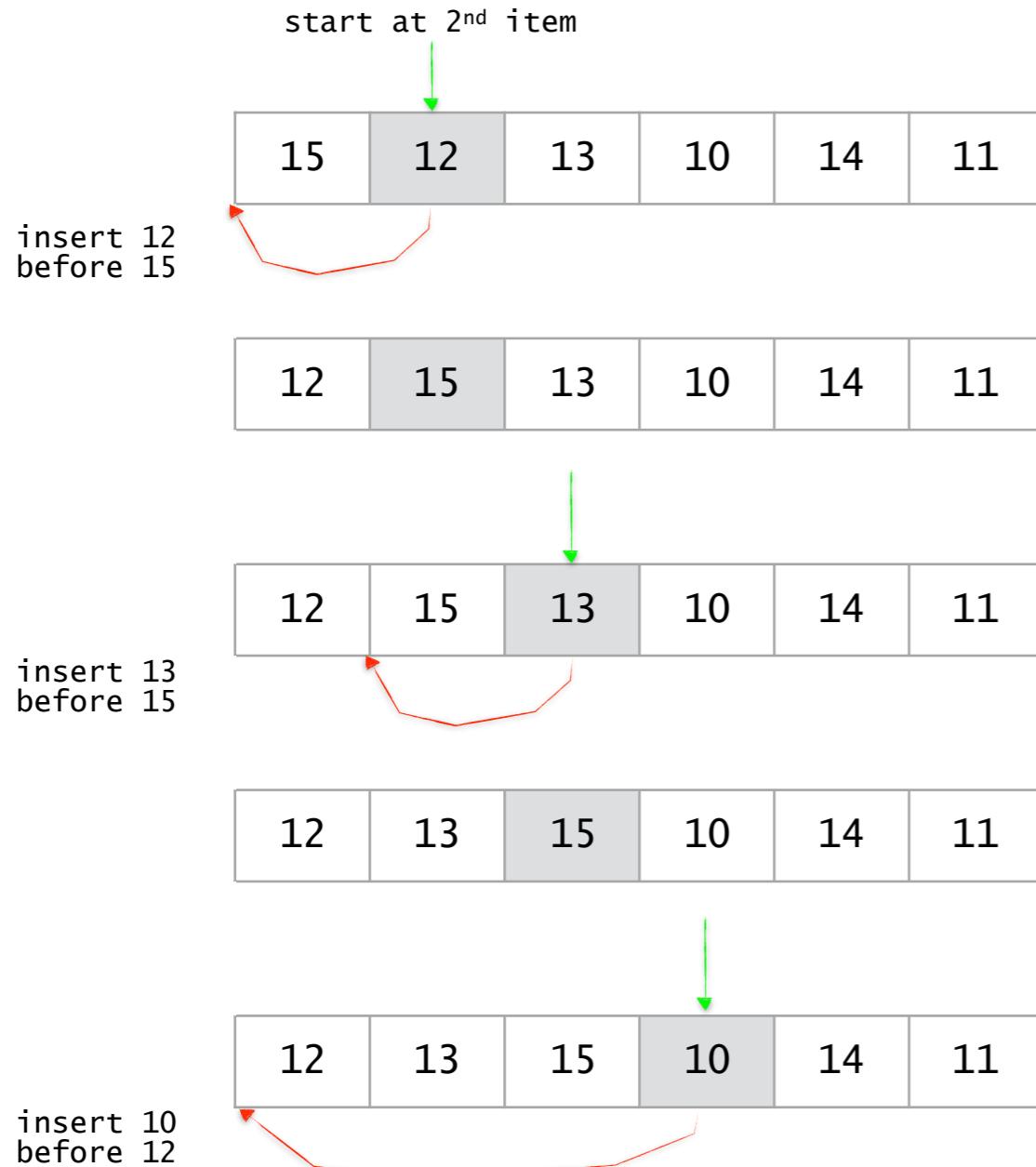
실행시간:  $(n-1) + (n-2) + \dots + 2 + 1 = O(n^2)$

```
bubbleSort(A[], n)    ▷ 배열 A[1...n]을 정렬한다.  
{  
    for last ← n downto 2 { -----①  
        for i ← 1 to last-1 -----②  
            if (A[i]>A[i+1]) then A[i] ↔ A[i+1]; ▷ 교환 -----③  
    }  
}
```

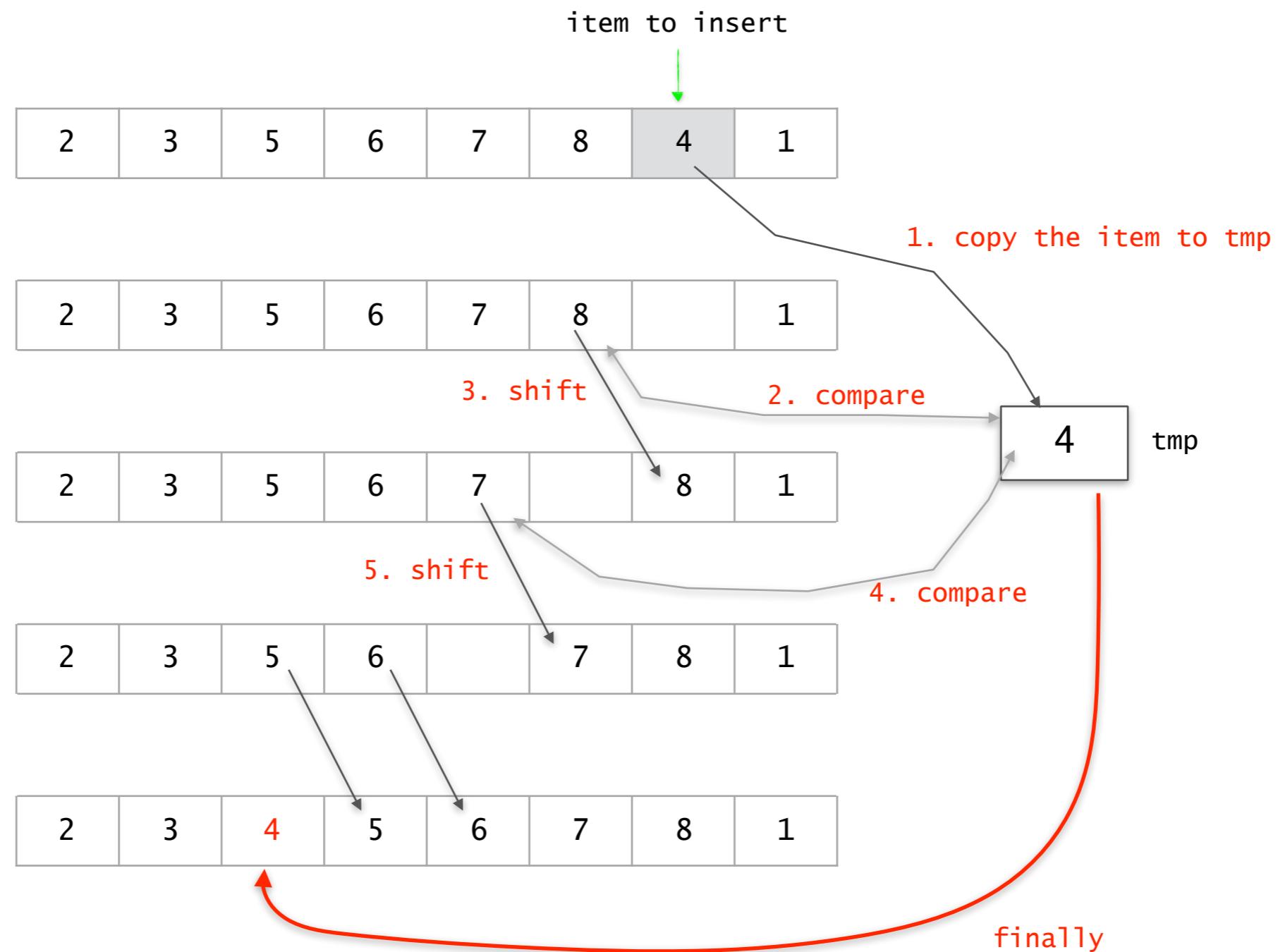
## ④ 수행시간:

- ①의 for 루프는  $n-1$ 번 반복
  - ②의 for 루프는 각각  $n-1, n-2, \dots, 2, 1$ 번 반복
  - ③의 교환은 상수시간 작업
- ④  $T(n) = (n-1) + (n-2) + \dots + 2 + 1 = O(n^2)$

# 삽입정렬 (Insertion Sort)



# Insertion



insertionSort(A[], n)    ▷ 배열 A[1...n]을 정렬한다.

{

    for i ← 2 to n { -----①

        A[1…i]의 적당한 자리에 A[i]를 삽입한다. -----③

}

## ▣ 수행시간:

①의 for 루프는  $n-1$ 번 반복

②의 삽입은 최악의 경우  $i-1$ 번 비교

▣ 최악의 경우:  $T(n)=(n-1)+(n-2)+\dots+2+1 = O(n^2)$

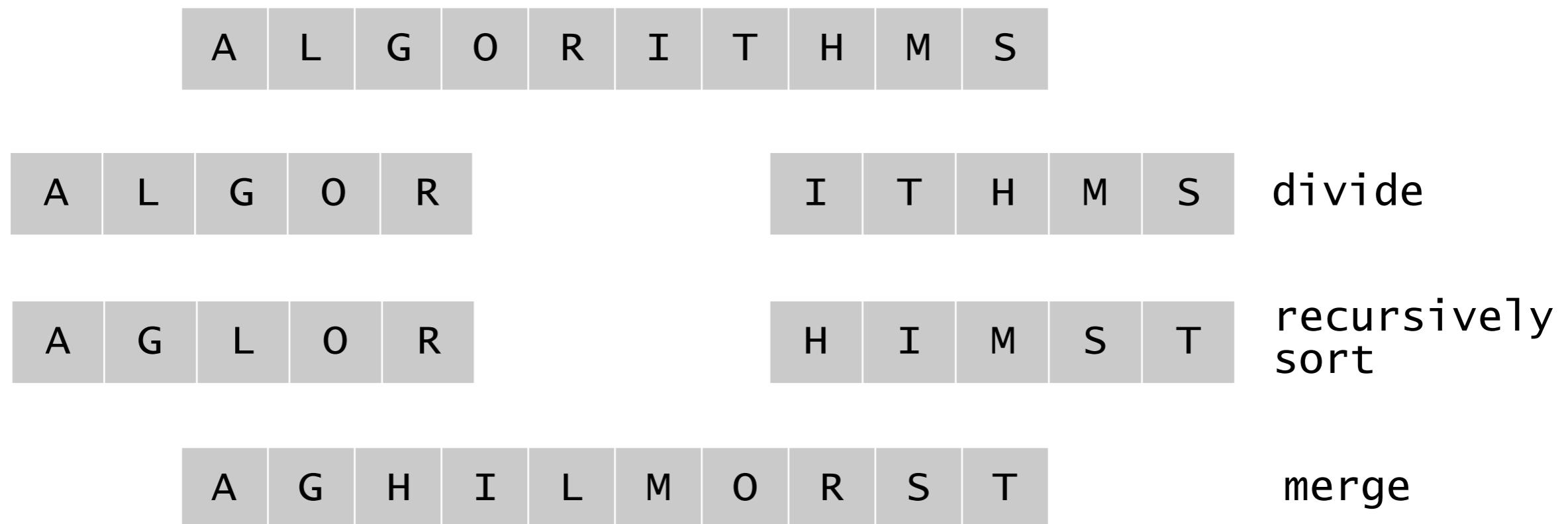
**분할정복법**

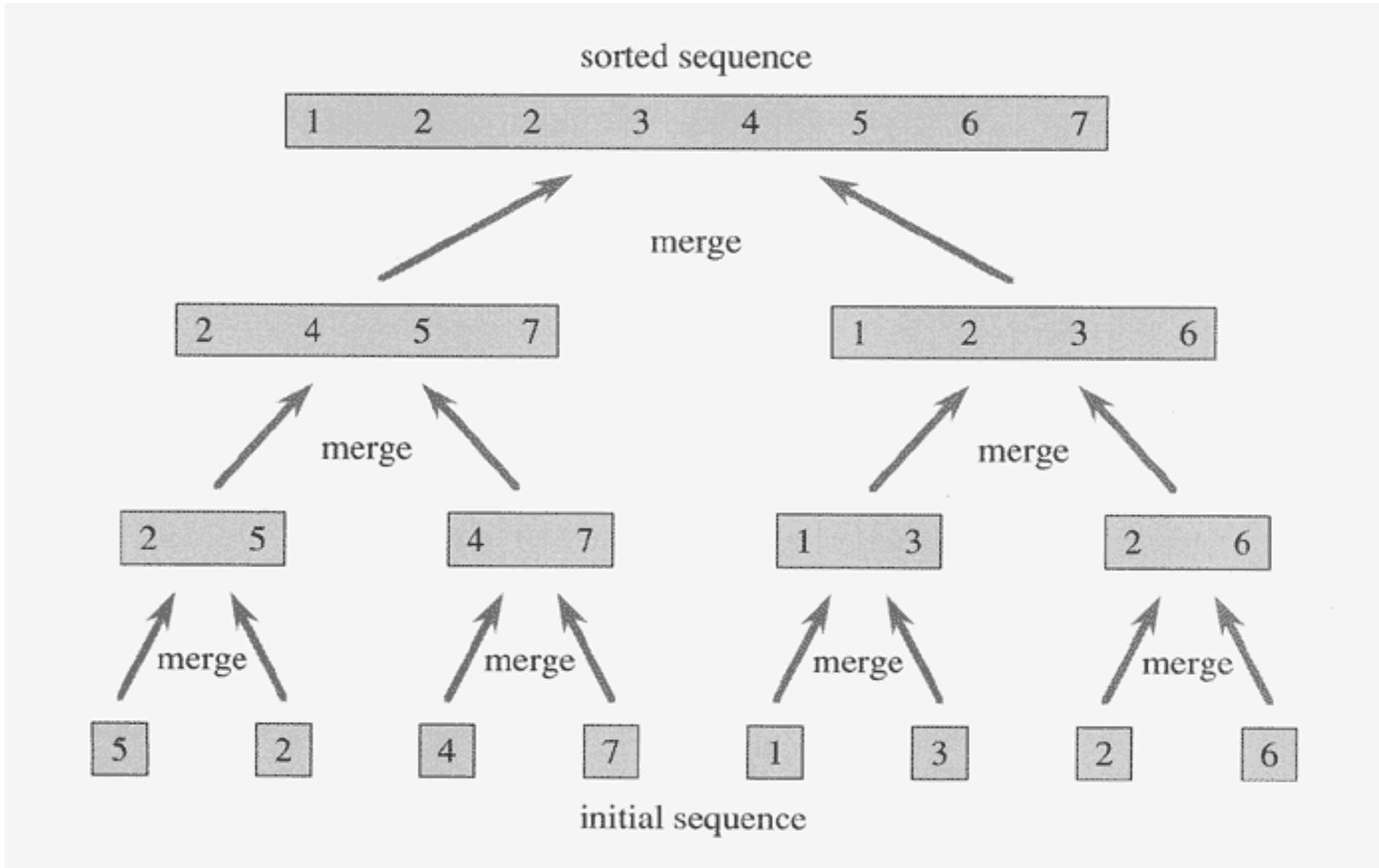
**Divide-And-Conquer**

- ➊ 분할: 해결하고자 하는 문제를 작은 크기의 동일한 문제들로 분할
- ➋ 정복: 각각의 작은 문제를 순환적으로 해결
- ➌ 합병: 작은 문제의 해를 합하여(merge) 원래 문제에 대한 해를 구함

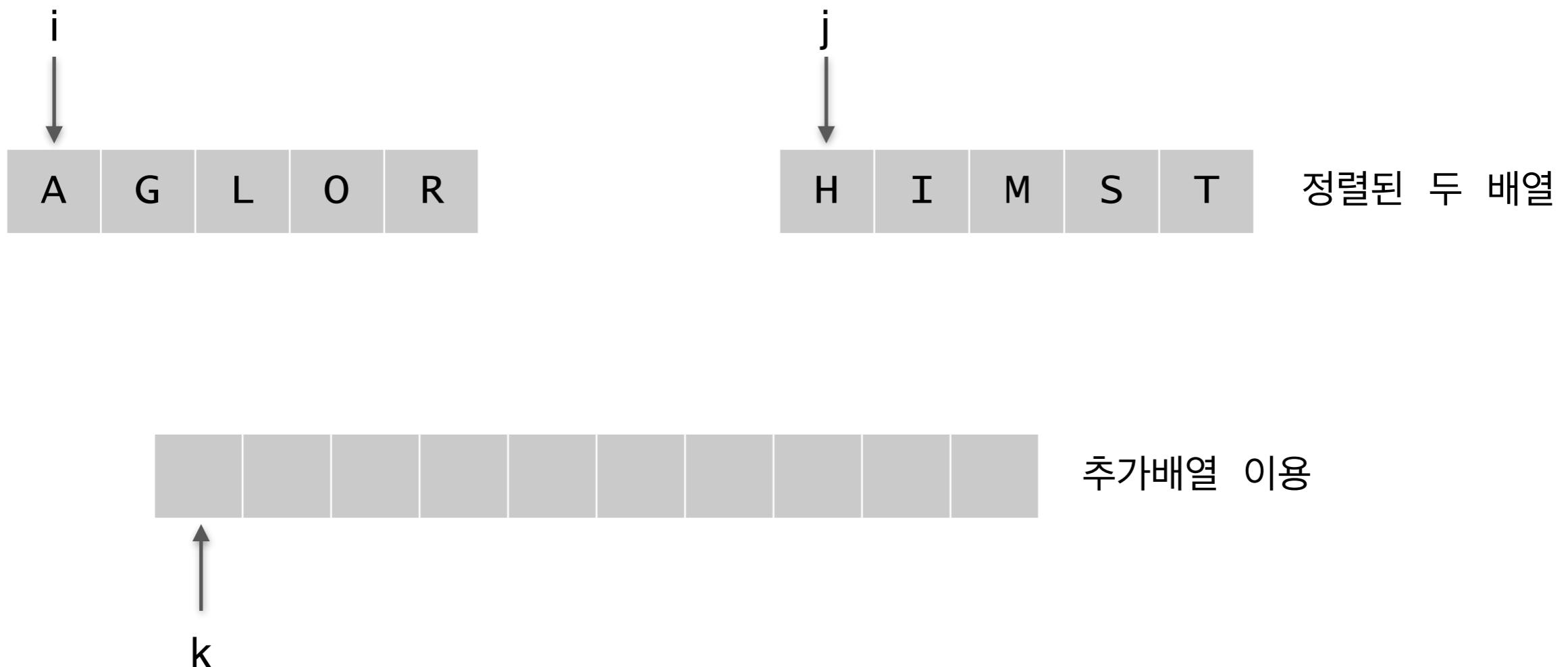
## 합병정렬(merge sort)

- 데이터가 저장된 배열을 절반으로 나눔
- 각각을 순환적으로 정렬
- 정렬된 두 개의 배열을 합쳐 전체를 정렬

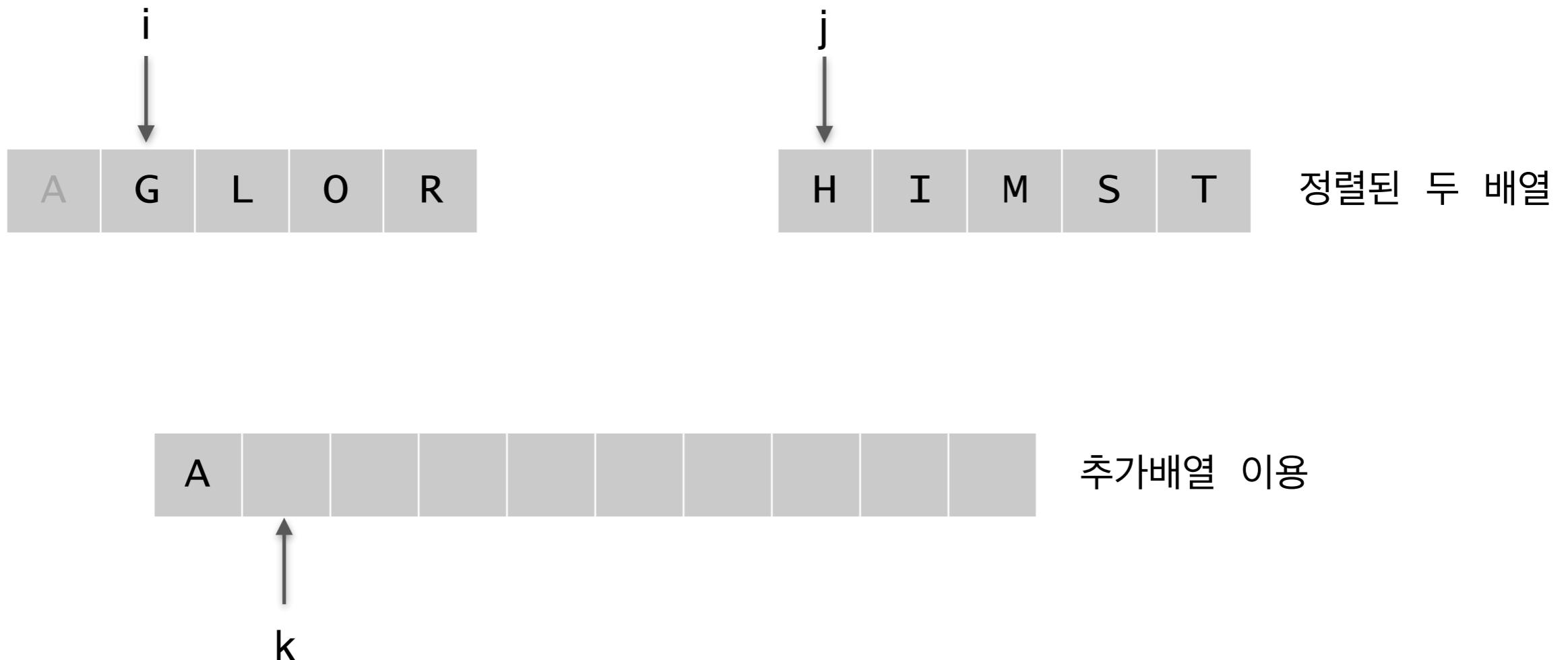




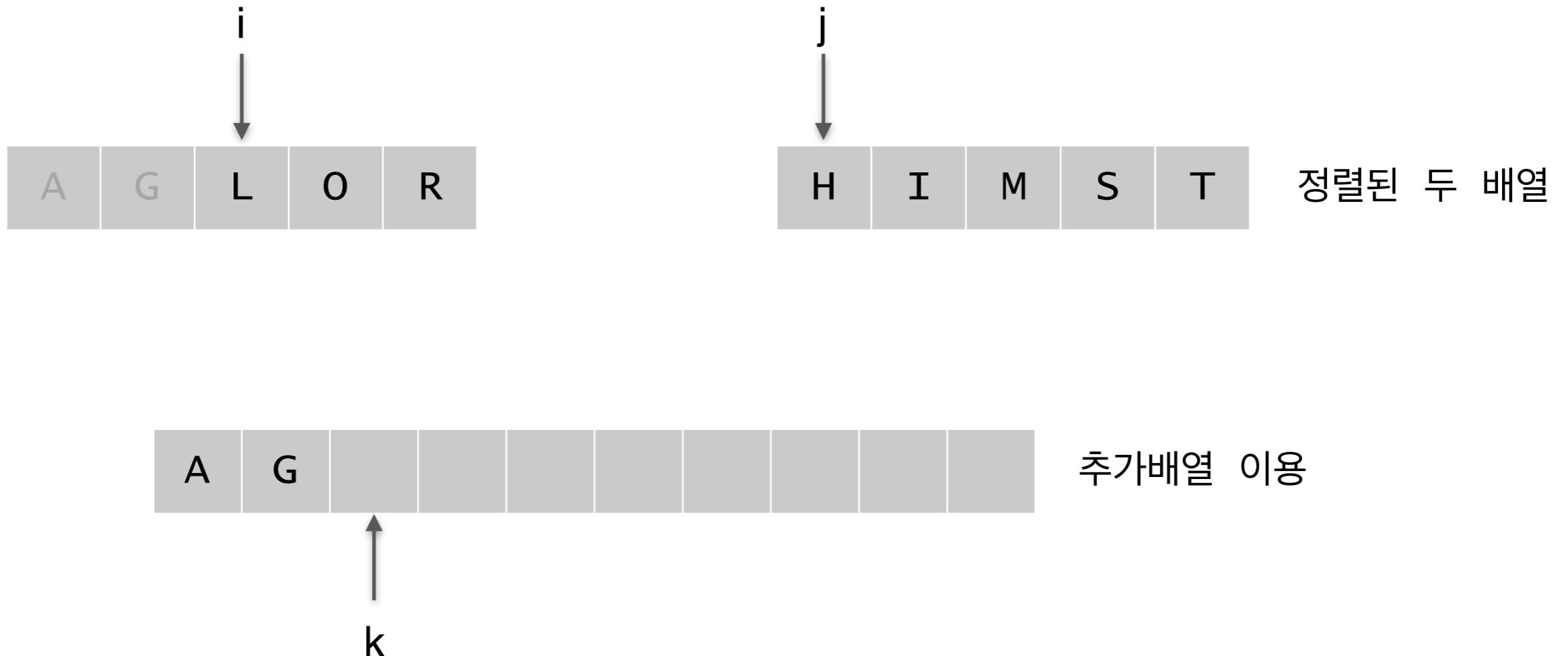
## 합병



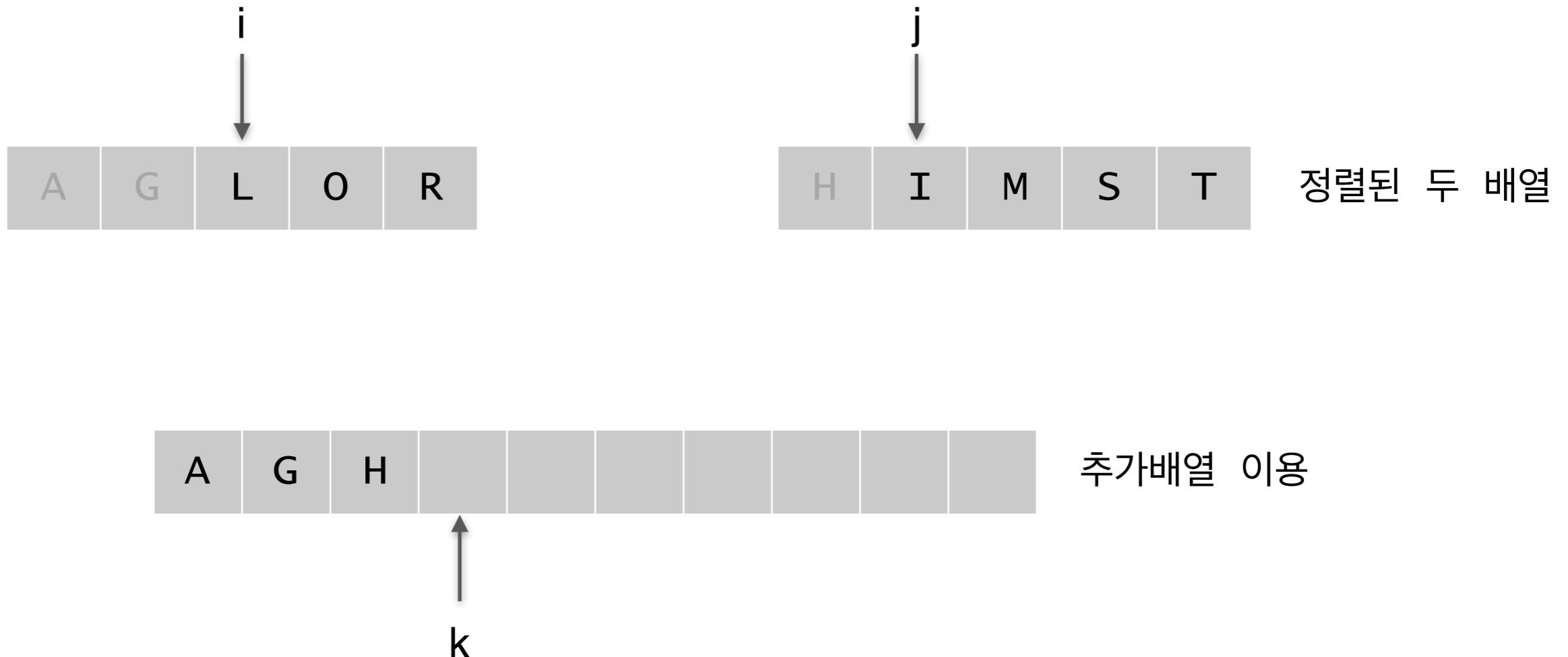
# 합병



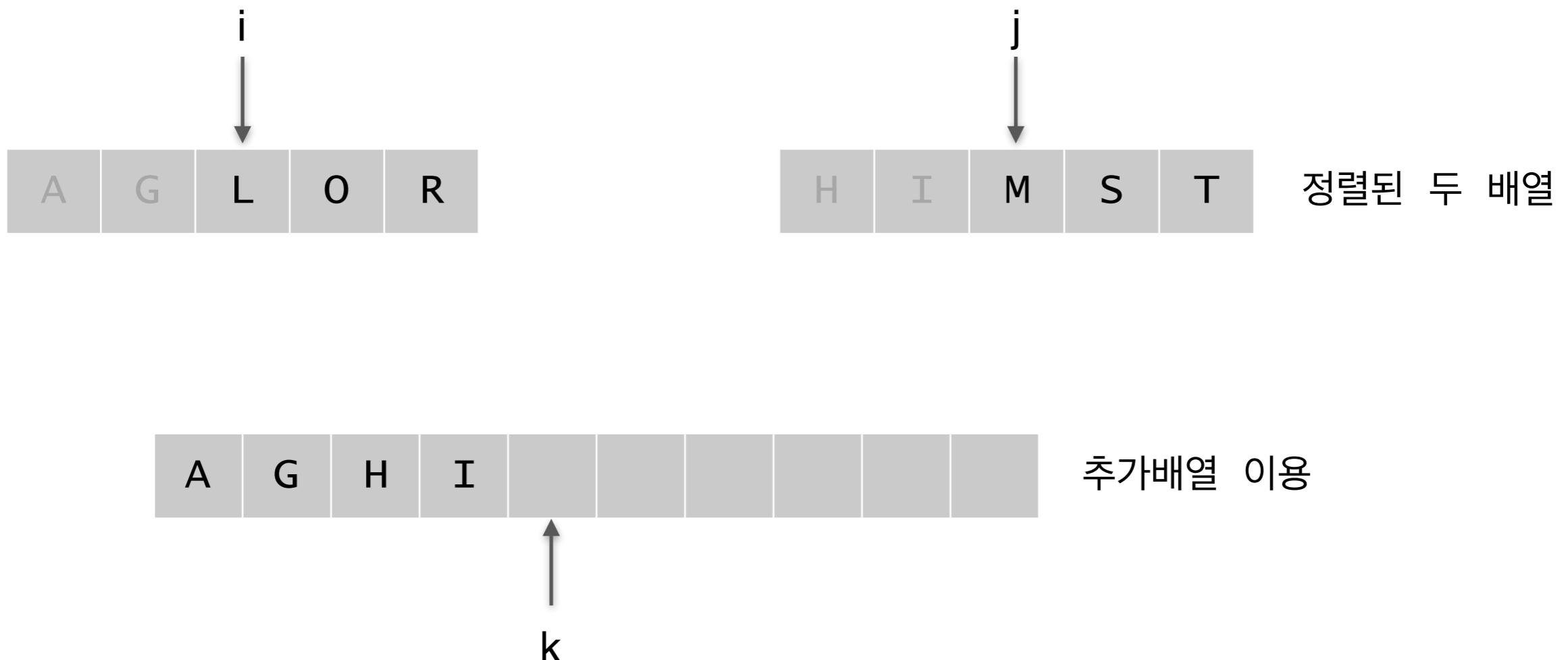
# 합병



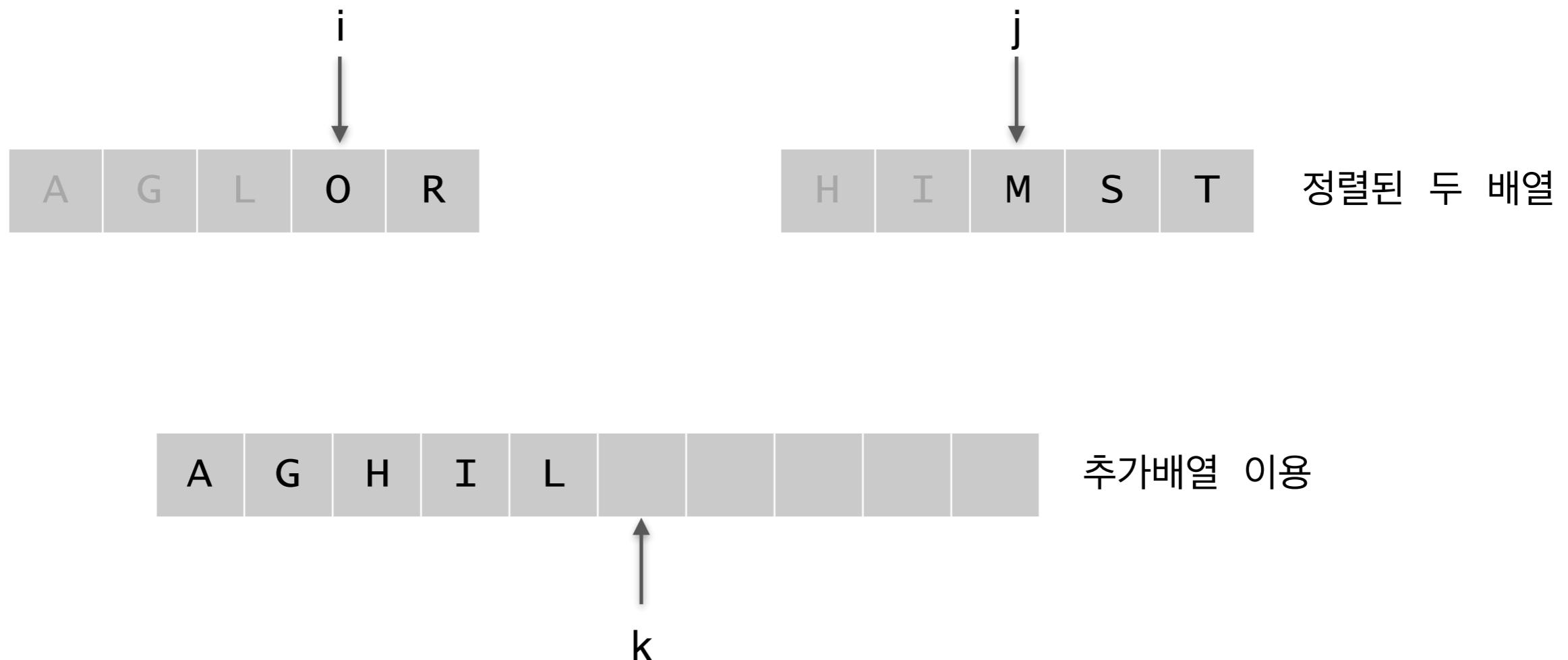
# 합병



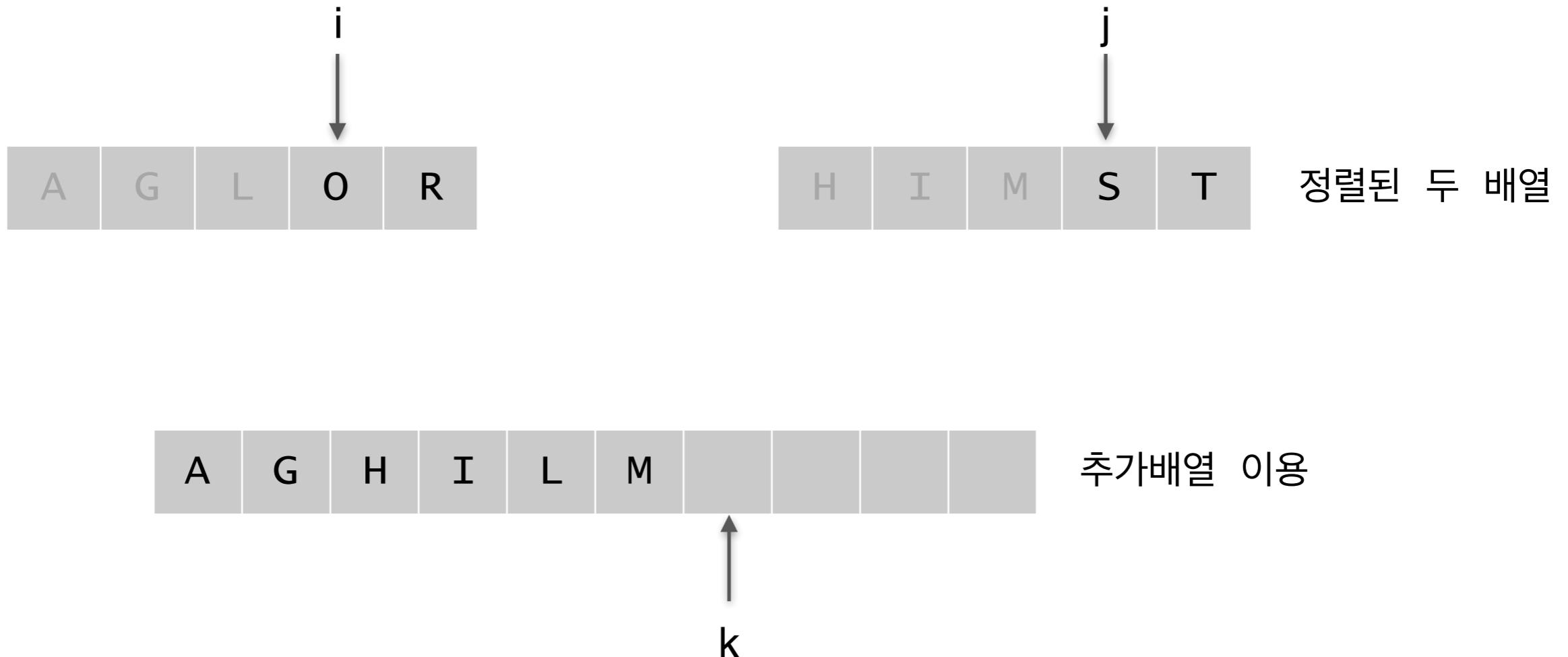
## 합병



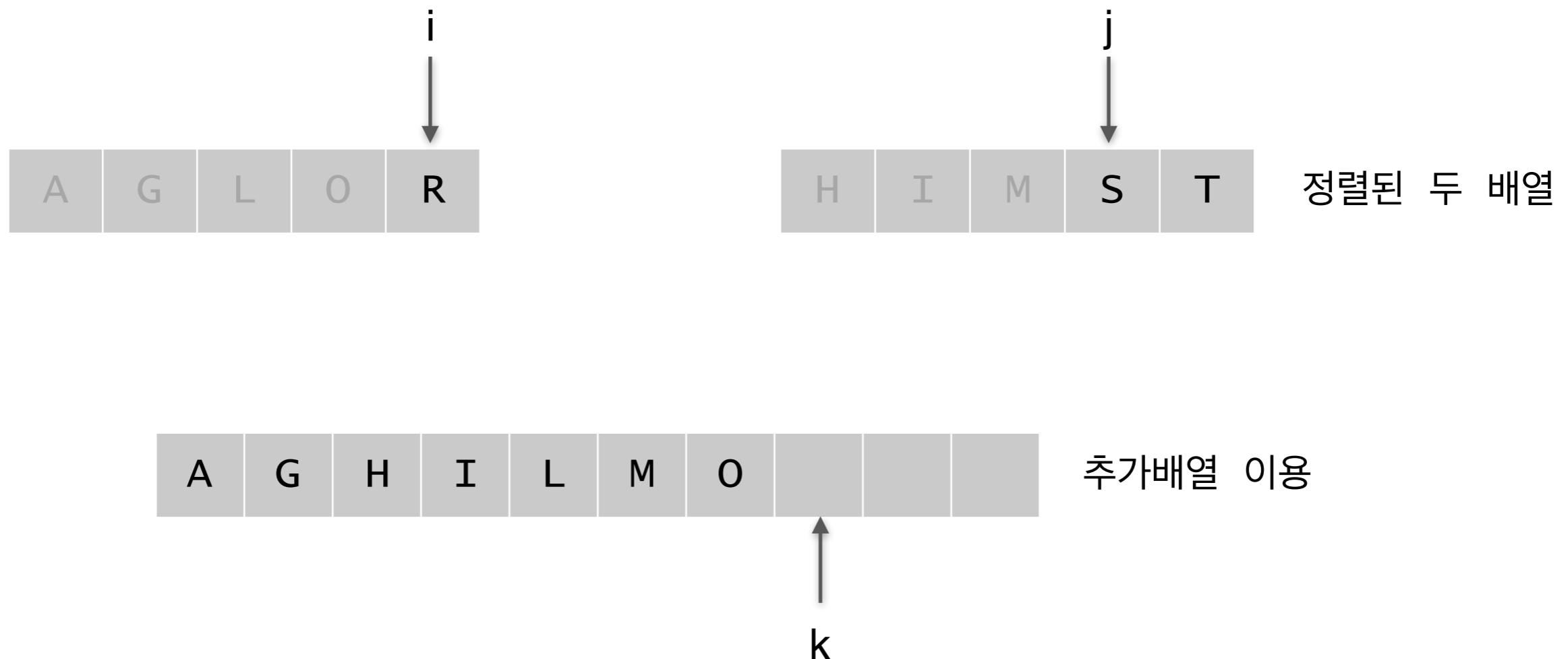
# 합병



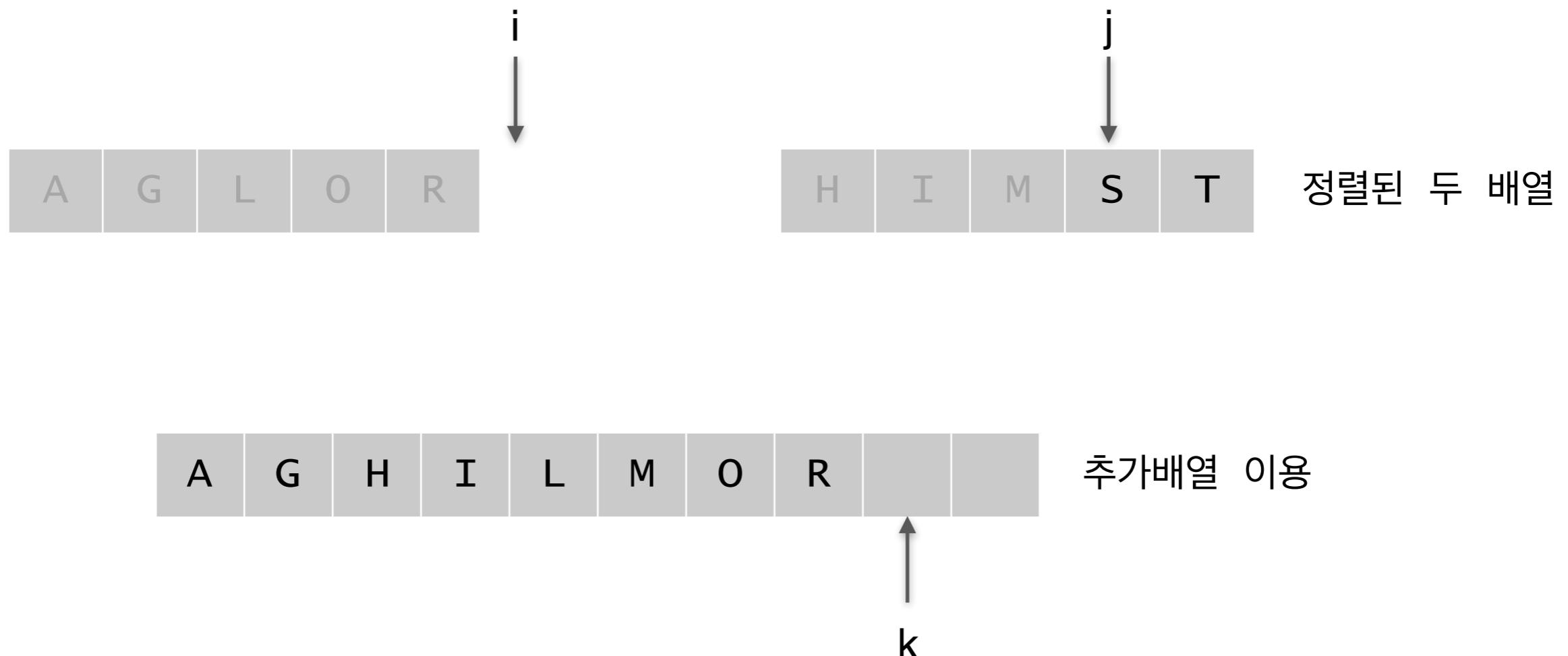
# 합병



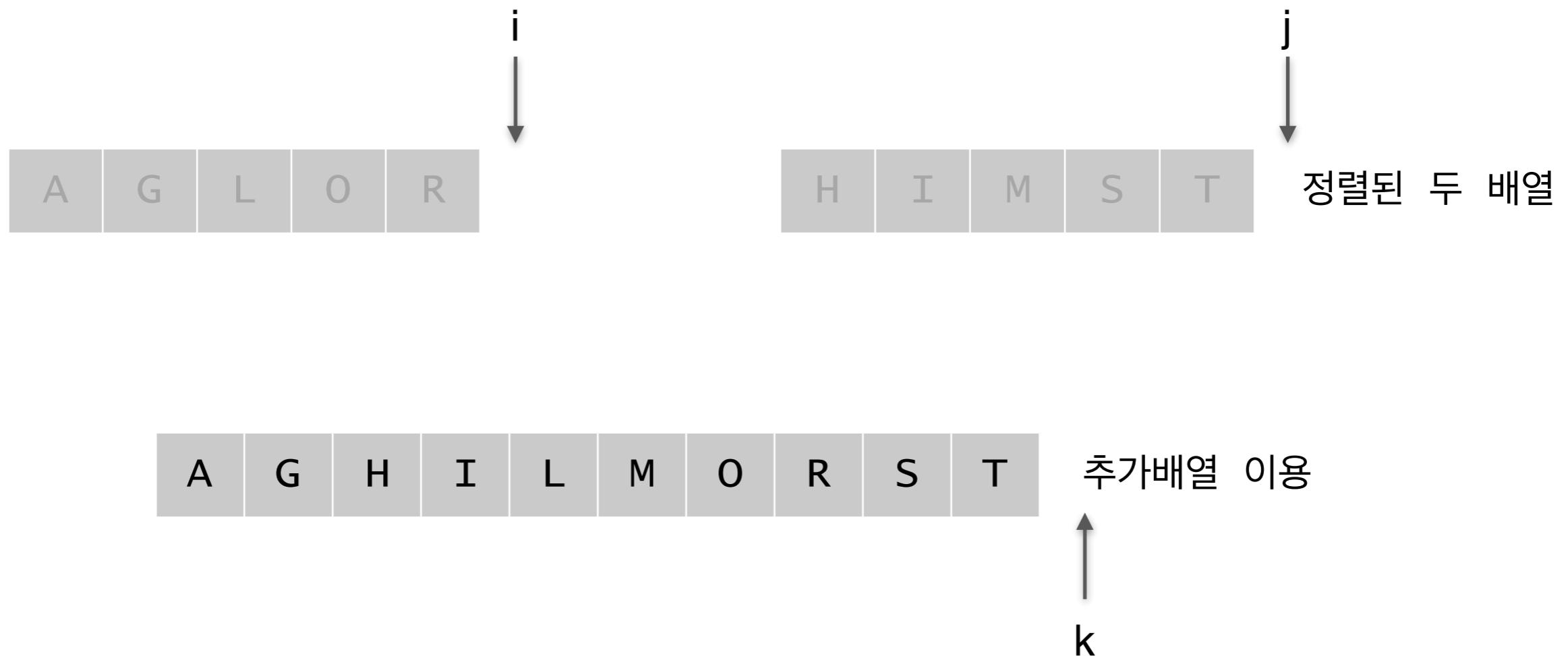
## 합병



# 합병



## 합병

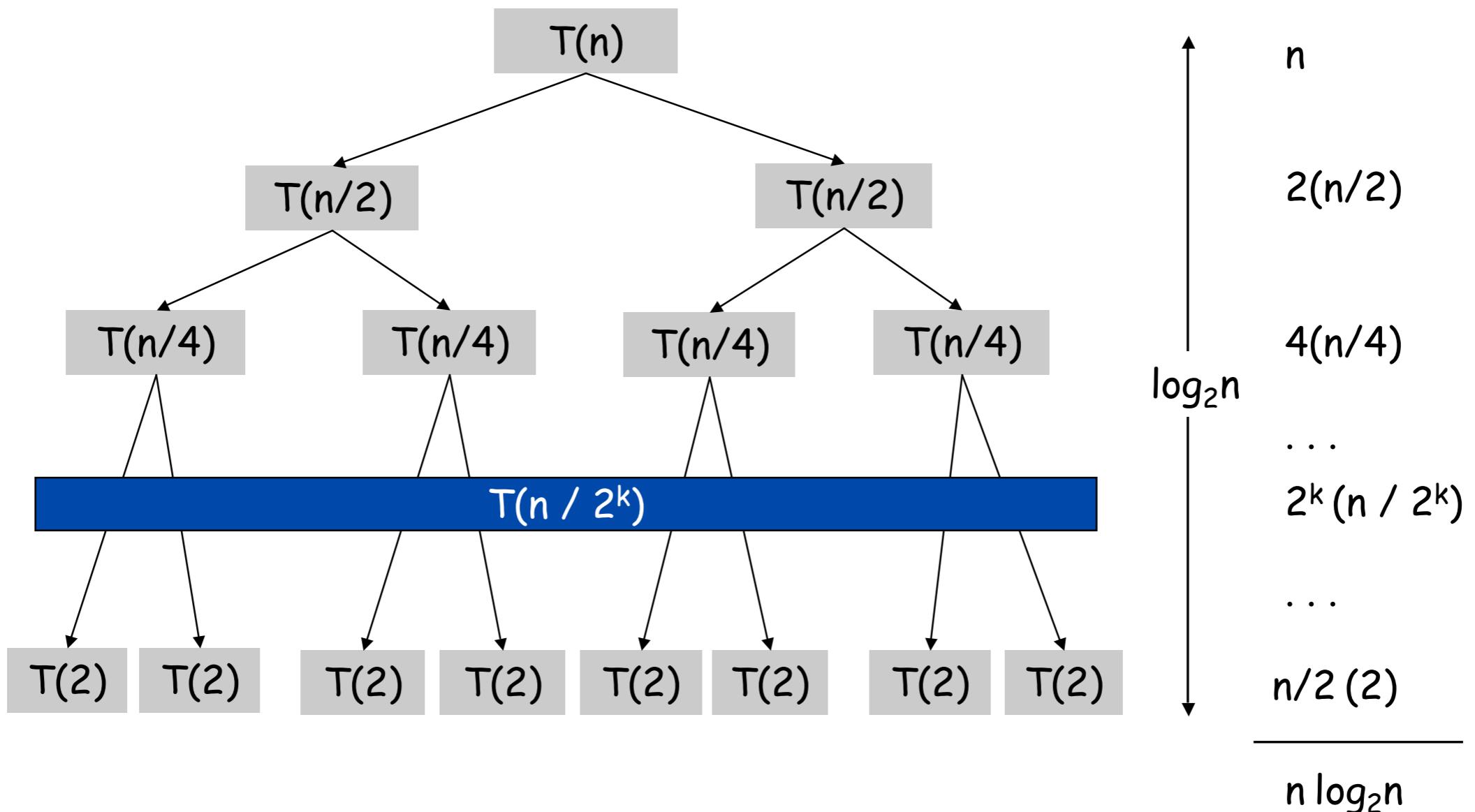


## Mergesort

```
mergeSort(A[], p, r)    ▷ A[p...r]을 정렬한다
{
    if (p < r) then {
        q ← (p+q)/2;      -----① ▷ p, q의 중간 지점 계산
        mergeSort(A, p, q); -----② ▷ 전반부 정렬
        mergeSort(A, q+1, r); -----③ ▷ 후반부 정렬
        merge(A, p, q, r); -----④ ▷ 합병
    }
}
```

```
merge(A[], p, q, r)
{
    정렬되어 있는 두 배열 A[p...q]와 A[q+1...r]을 합하여
    정렬된 하나의 배열 A[p...r]을 만든다.
}
```

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{otherwise.} \end{cases}$$



## ➊ 분할정복법

- ➌ 분할: 배열을 다음과 같은 조건이 만족되도록 두 부분으로 나눈다.

$$\begin{matrix} \text{elements} \\ \text{in lower parts} \end{matrix} \leq \begin{matrix} \text{elements} \\ \text{in upper parts} \end{matrix}$$

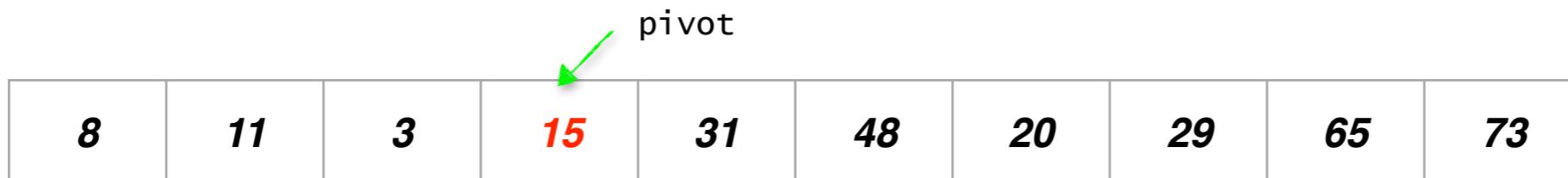
- ➍ 정복: 각 부분을 순환적으로 정렬한다.
- ➎ 합병: nothing to do

# Quicksort

- 정렬할 배열이 주어짐. 마지막 수를 **기준(pivot)**으로 삼는다.



- 기준보다 작은 수는 기준의 왼쪽에 나머지는 기준의 오른쪽에 오도록 재배치(분할)한다.



- 기준의 왼쪽과 오른쪽을 각각 **순환적으로** 정렬한다 (정렬 완료)

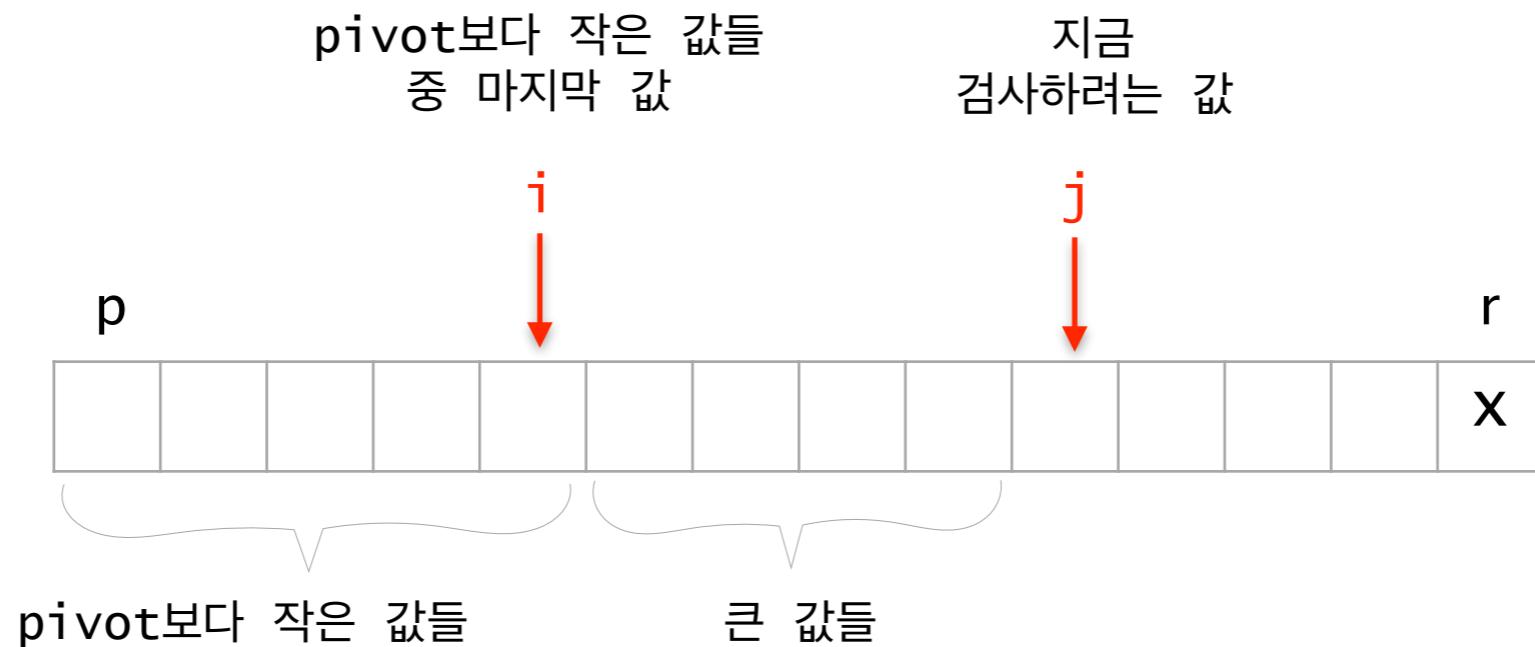


# Quicksort

```
quickSort(A[], p, r)           ▷ A[p...r]을 정렬한다
{
    if (p<r) then {
        q = partition(A, p, r); ▷ 분할
        quickSort(A, p, q-1);   ▷ 왼쪽 부분배열 정렬
        quickSort(A, q+1, r);   ▷ 오른쪽 부분배열 정렬
    }
}
```

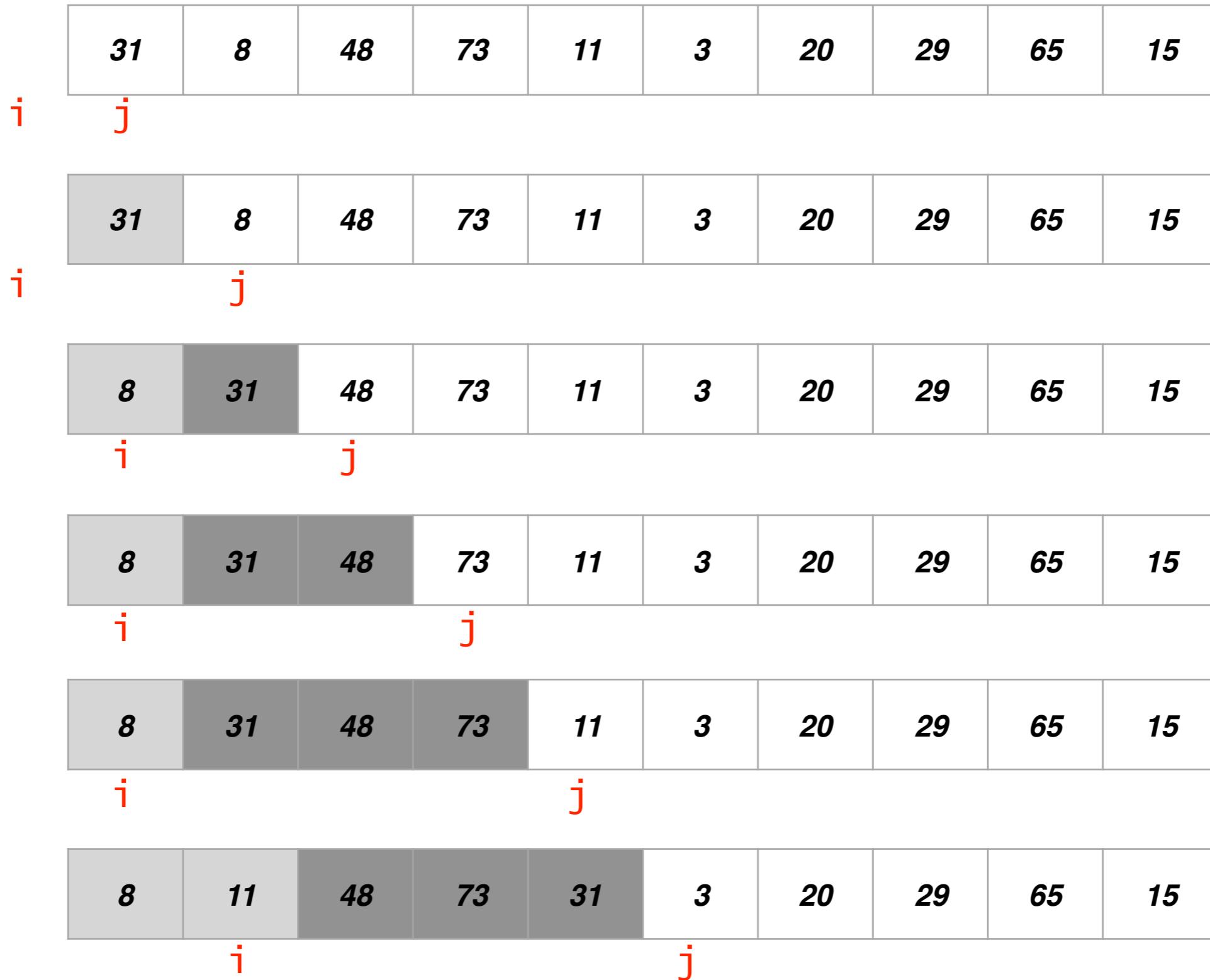
```
partition(A[], p, r)
{
    배열 A[p...r]의 원소들을 A[r]을 기준으로 양쪽으로 재배치하고
    A[r]이 자리한 위치를 return 한다;
}
```

# Partition

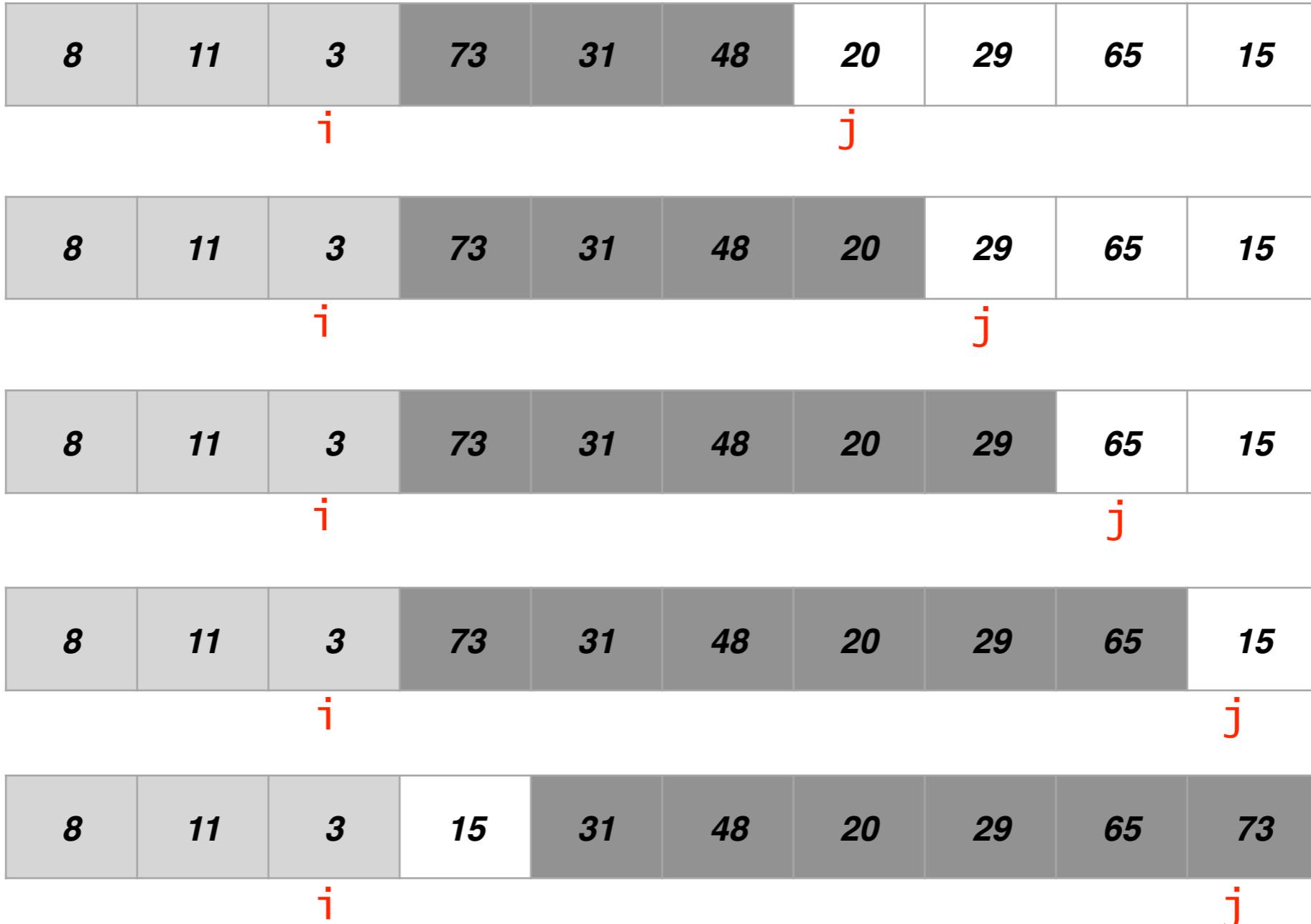


```
if  $A[j] \geq x$ 
     $j \leftarrow j + 1;$ 
else
     $i \leftarrow i + 1;$ 
    exchange  $A[i]$  and  $A[j]$ ;
     $j \leftarrow j + 1;$ 
```

## Partition의 예



## Partition의 예



## Partition

```
Partition(A, p, r)
```

```
{
```

```
    x←A[r];
```

```
    i←p-1;
```

```
    for j←p to r-1
```

```
        if A[j] ≤ x then
```

```
            i←i+1;
```

```
            exchange A[i] and A[j];
```

```
exchange A[i+1] and A[r];
```

```
return i+1;
```

```
}
```



- 항상 한 쪽은 0개, 다른 쪽은 n-1개로 분할되는 경우

$$\begin{aligned} T(n) &= T(0) + T(n - 1) + \Theta(n) \\ &= T(n - 1) + \Theta(n) \\ &= T(n - 2) + T(n - 1) + \Theta(n - 1) + \Theta(n) \\ &\quad \dots \\ &= \Theta(1) + \Theta(2) + \dots + \Theta(n - 1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

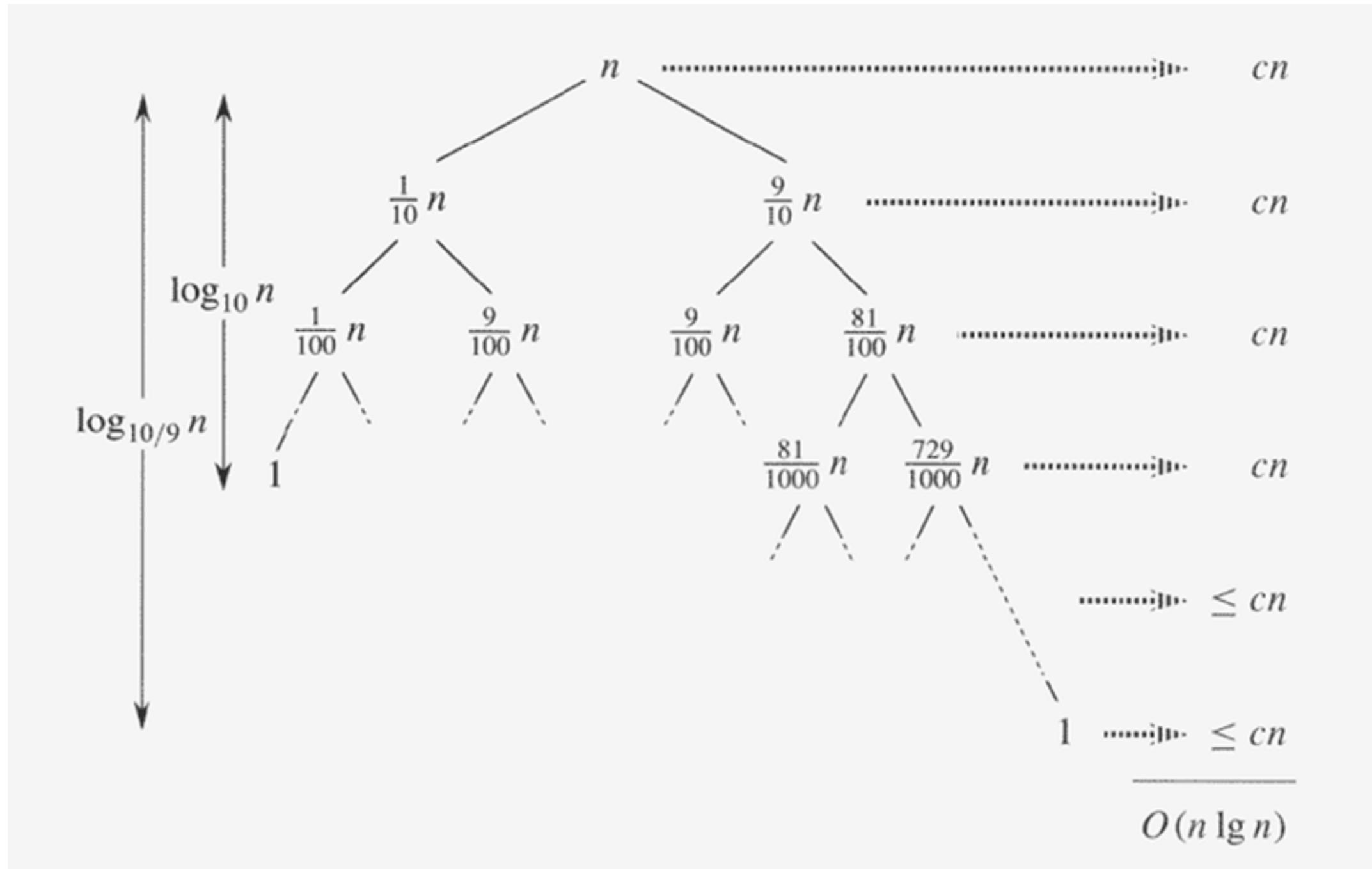
- 이미 정렬된 입력 데이터 (마지막 원소를 피봇으로 선택하는 경우)

## ◎ 항상 절반으로 분할되는 경우

$$\begin{aligned}T(n) &= 2T(n/2) + \Theta(n) \\&= \Theta(n \log n)\end{aligned}$$

## Balanced Partition

항상 한쪽이 적어도  $1/9$  이상이 되도록 분할된다면?



- “평균” 혹은 “기대값”이란?

$p(I)$ :  $I$ 가 입력으로 들어올 확률  
 $T(I)$ :  $I$ 를 정렬하는데 걸리는 시간

$$A(n) = \sum_{\forall \text{ input instance } I \text{ of size } n} p(I)T(I)$$

- 그러나,  $p(I)$ 를 모름
- $p(I)$ 에 관한 적절한 가정을 한 후 분석
- 예: 모든 입력 인스턴스가 동일한 확률을 가진다면

$$p(I) = \frac{1}{n!}$$

## 평균시간복잡도

rank of pivot	probability	result of partition	running time
1	1/n	0:n-1	A(0)+A(n-1)
2	1/n	1:n-2	A(1)+A(n-2)
3	1/n	2:n-3	A(2)+A(n-3)
...		...	
n-1	1/n	n-2:1	A(n-2)+A(1)
n	1/n	n-1:0	A(n-1)+A(0)

$$A(n) = \frac{n}{2} \sum_{i=0}^{n-1} A(i) + \Theta(n) = \Theta(n \log_2 n)$$

## 첫번째 값이나 마지막 값을 피봇으로 선택

- 이미 정렬된 데이터 혹은 거꾸로 정렬된 데이터가 최악의 경우
- 현실의 데이터는 랜덤하지 않으므로 (거꾸로) 정렬된 데이터가 입력으로 들어올 가능성은 매우 높음
- 따라서 좋은 방법이라고 할 수 없음

## “Median of Three”

- 첫번째 값과 마지막 값, 그리고 가운데 값 중에서 중간값(median)을 피봇으로 선택
- 최악의 경우 시간복잡도가 달라지지는 않음

## Randomized Quicksort

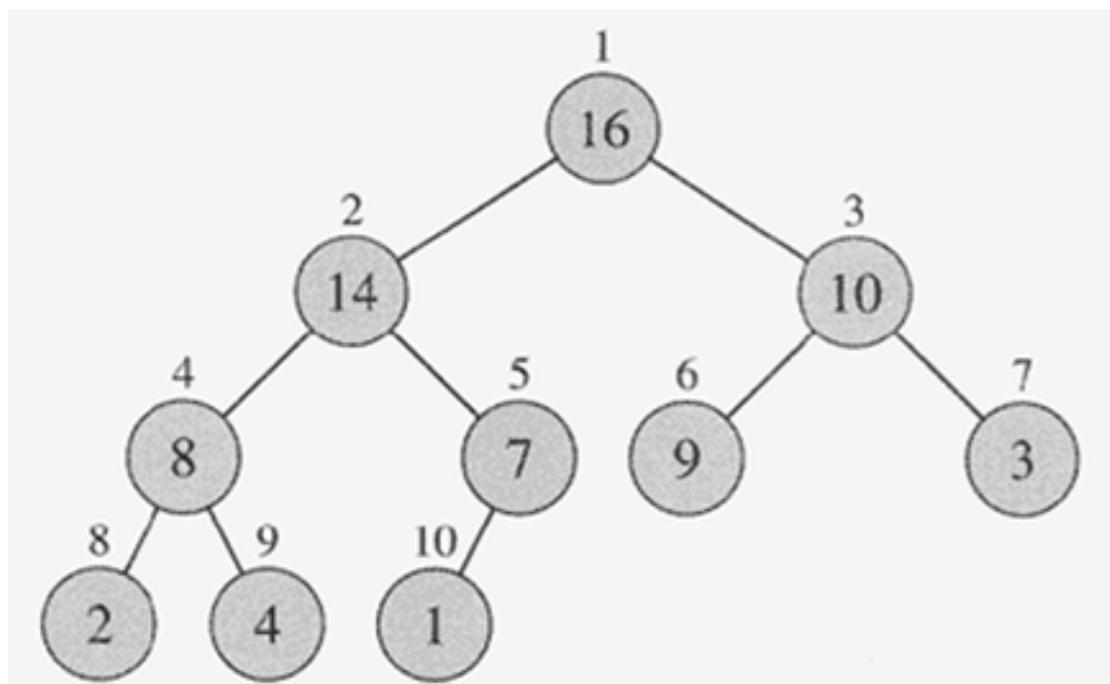
- 피봇을 랜덤하게 선택
- no worst case instance, but worst case execution
- 평균 시간복잡도  $O(N \log N)$

# Heap과 Heapsort

- ☞ 최악의 경우 시간복잡도  $O(n \log_2 n)$
- ☞ Sorts in place - 추가 배열 불필요
- ☞ 이진 힙(binary heap) 자료구조를 사용

## Heap은

- complete binary tree이면서
- heap property를 만족해야



max heap property:

부모는 자식보다  
크거나 같다

or

min heap property:

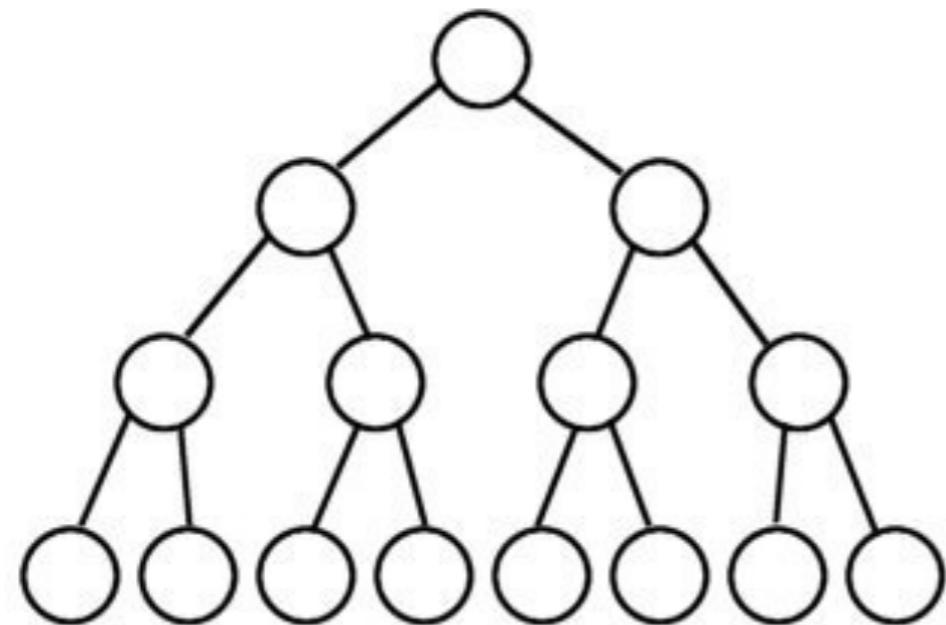
부모는 자식보다  
작거나 같다

Height =  $\Theta(\log_2 n)$

이 수업에서는 max heap을 다룸

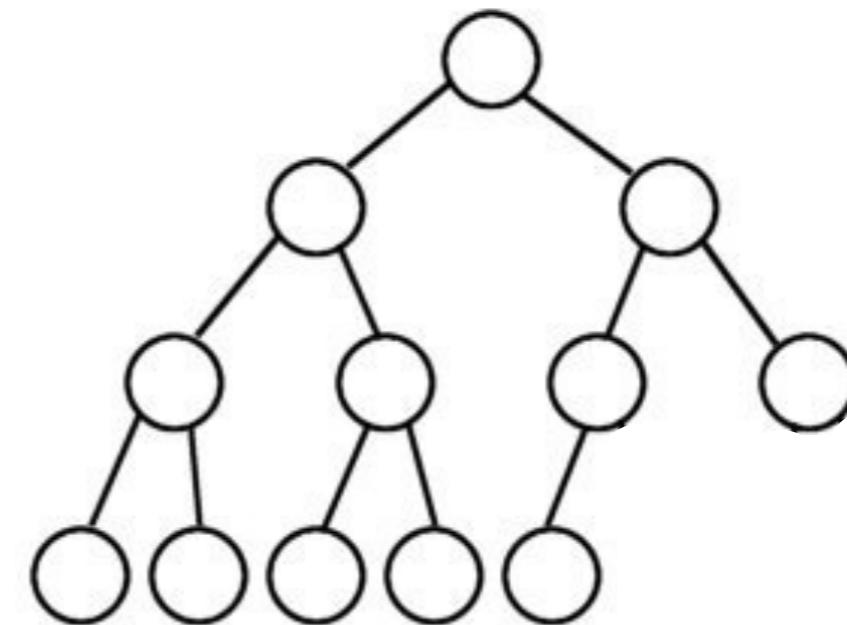
## Full v.s. Complete Binary Trees

full binary tree



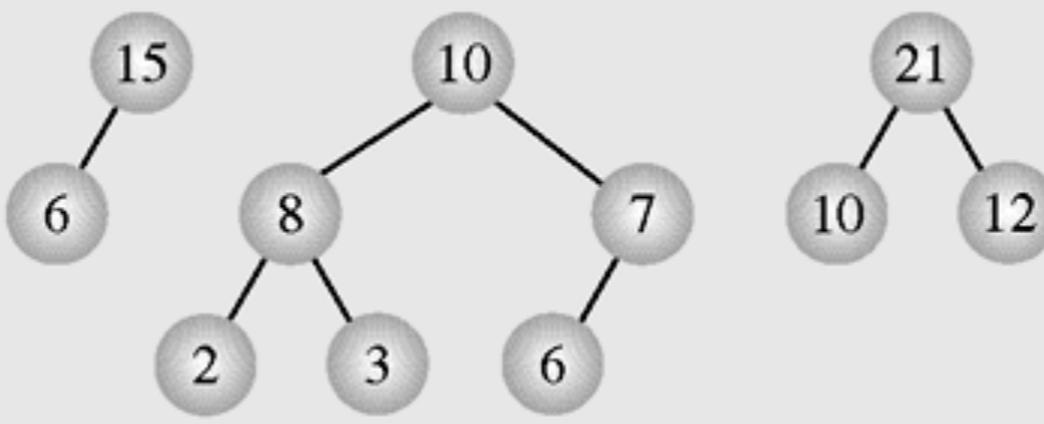
모든 레벨에 노드들이 꽉 차있는 형태

complete binary tree

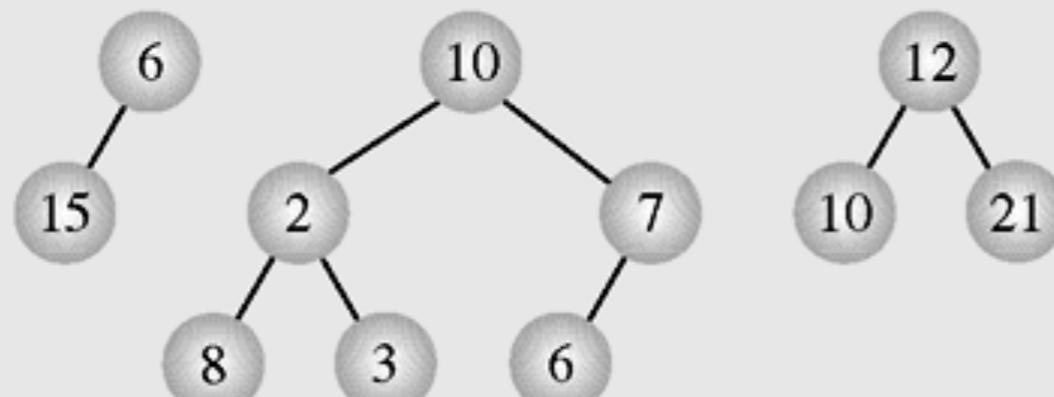


마지막 레벨을 제외하면 완전히  
꽉 차있고, 마지막 레벨에는  
가장 오른쪽 부터 연속된 몇 개의 노드가  
비어있을 수 있음

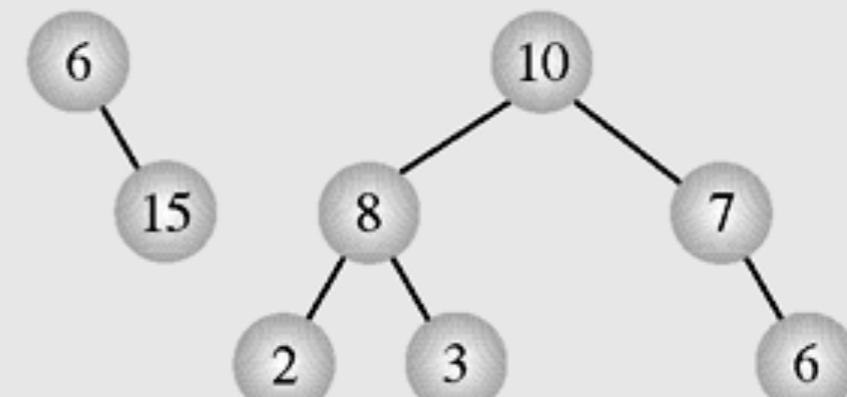
# Heap



(a)



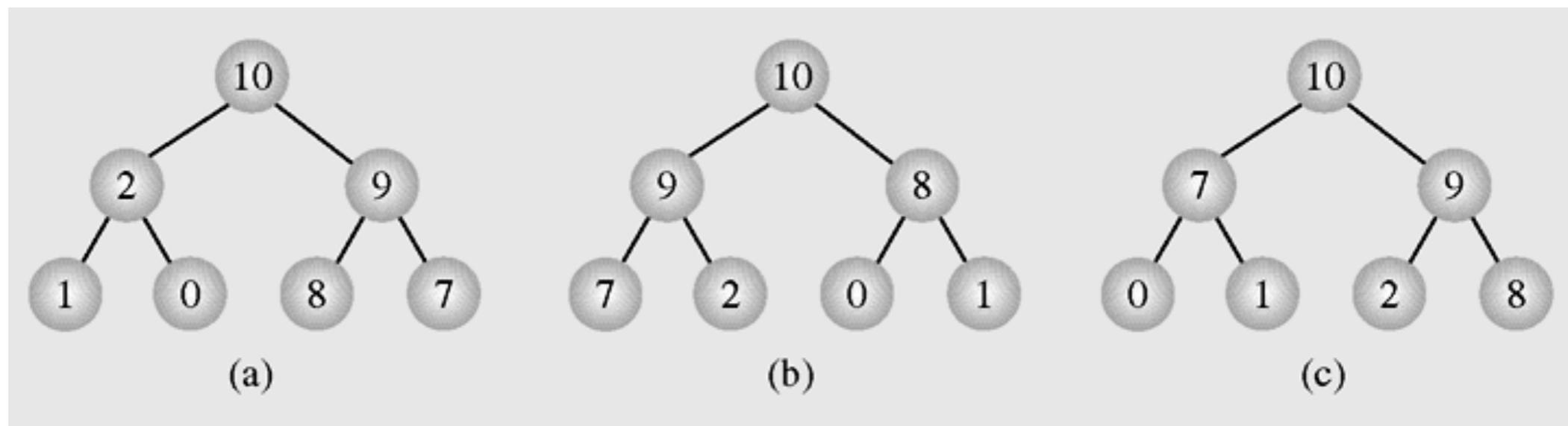
(b)



(c)

(a) heaps, (b,c) nonheaps

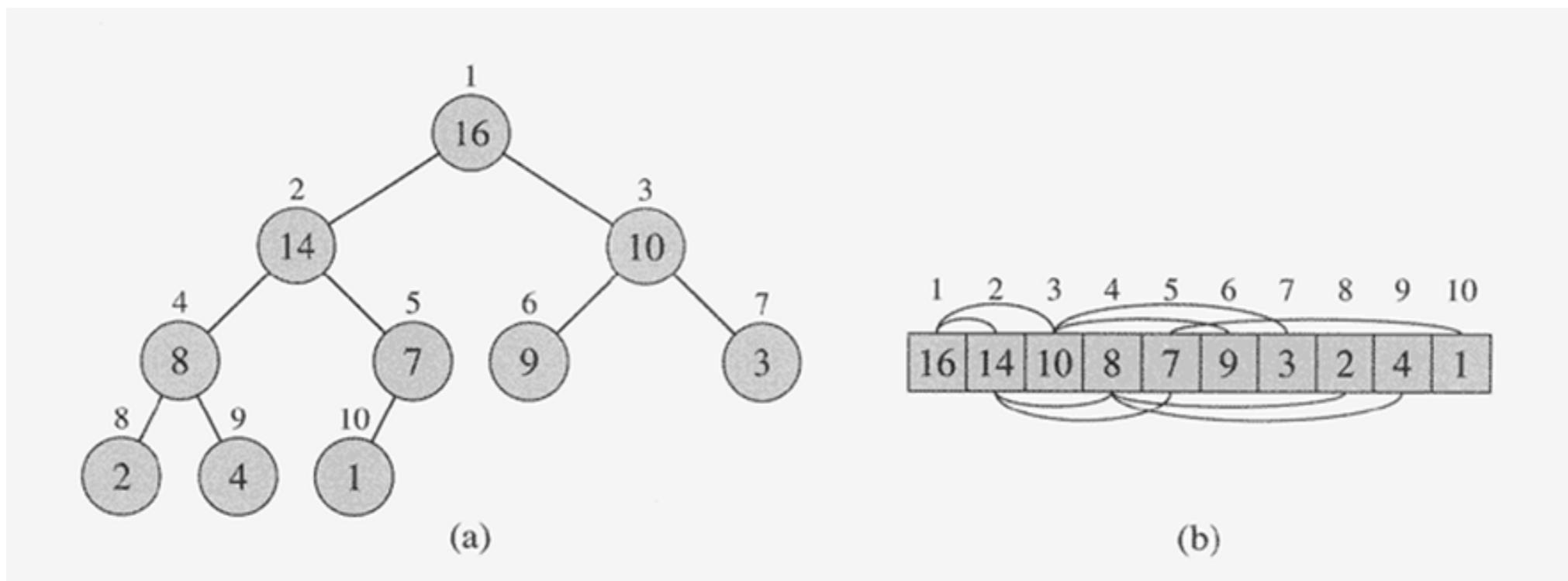
# Heaps



동일한 데이터를 가진 서로 다른 힙  
즉 힙은 유일하지 않음

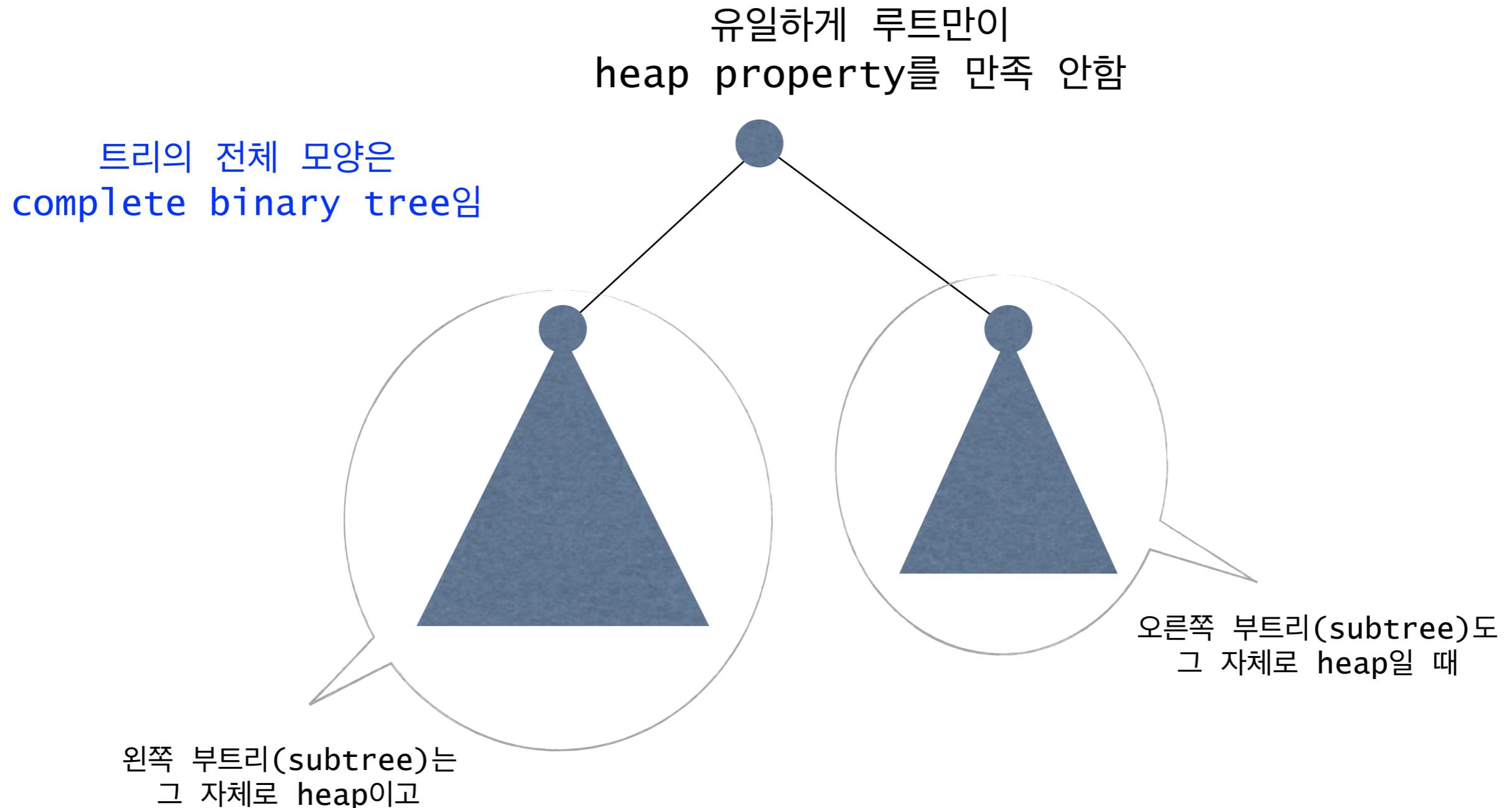
- 힙은 일차원 배열로 표현가능:  $A[1..n]$

- 루트 노드  $A[1]$ .
- $A[i]$ 의 부모 =  $A[i/2]$
- $A[i]$ 의 왼쪽 자식 =  $A[2i]$
- $A[i]$ 의 오른쪽 자식 =  $A[2i+1]$

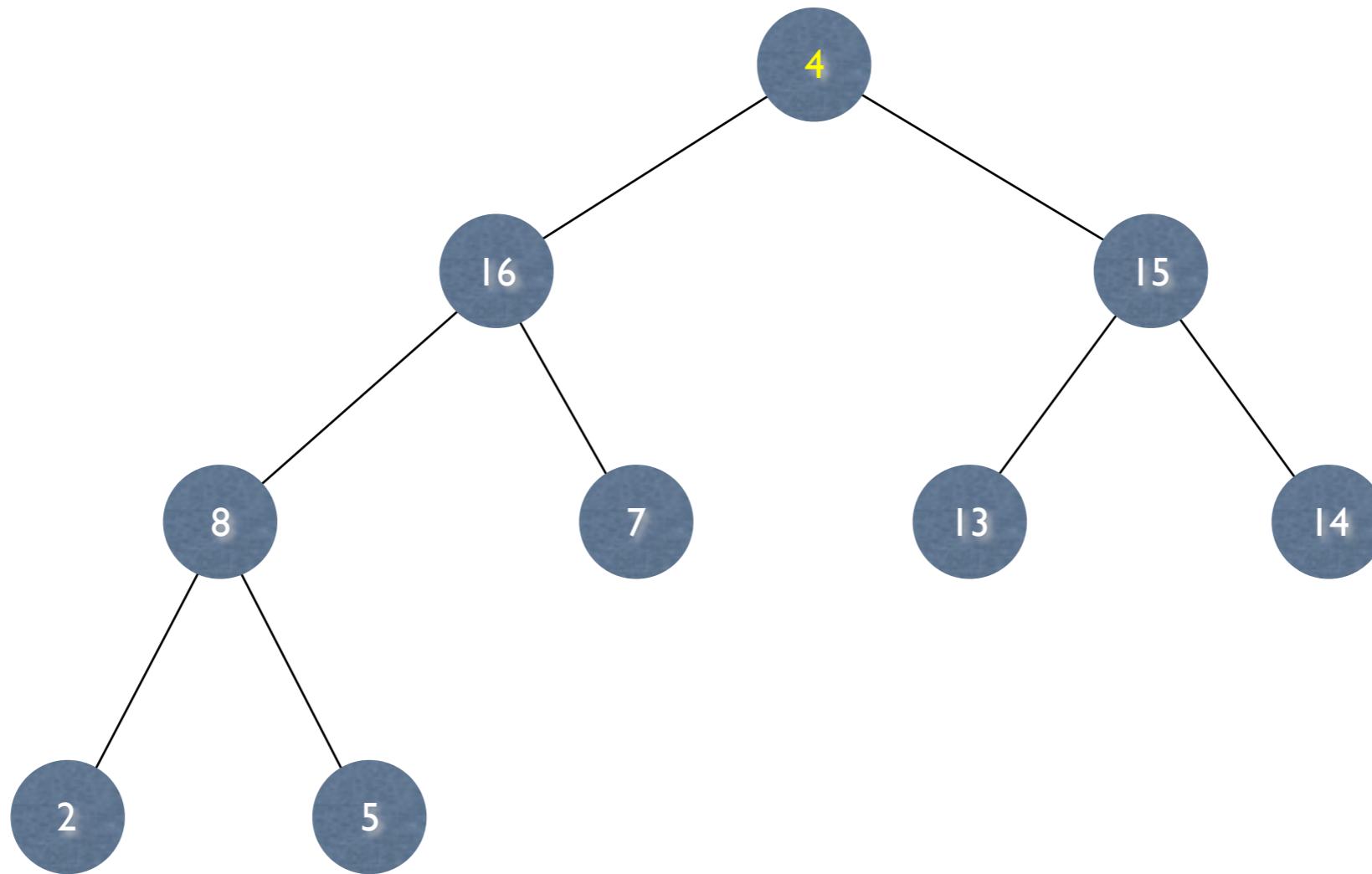


## 기본 연산: MAX-HEAPIFY

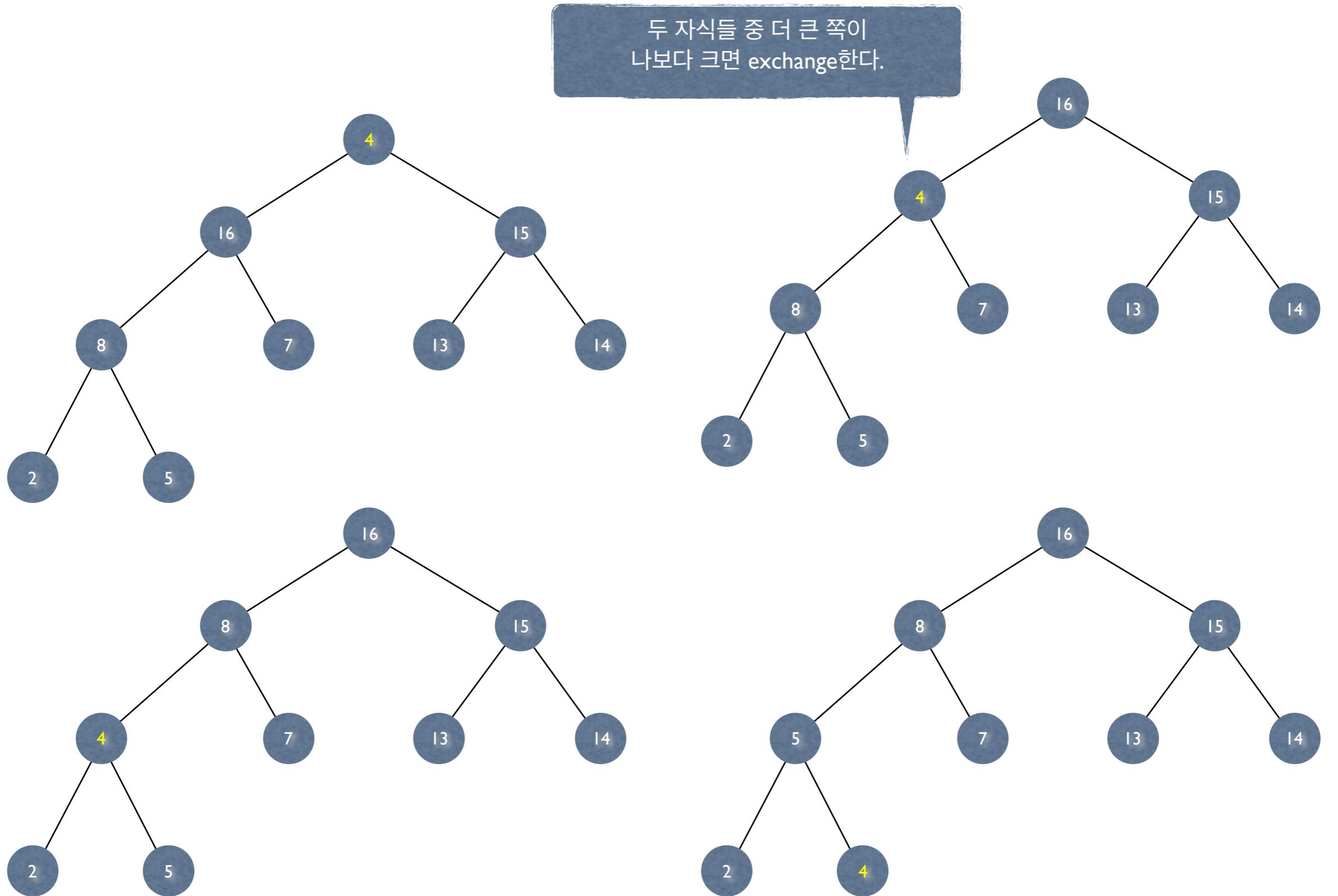
- 전체를 힙으로 만들어라!



## MAX-HEAPIFY



## MAX-HEAPIFY



## MAX-HEAPIFY: recursive version

노드  $i$ 를 루트로하는 서브트리를 heapify한다.

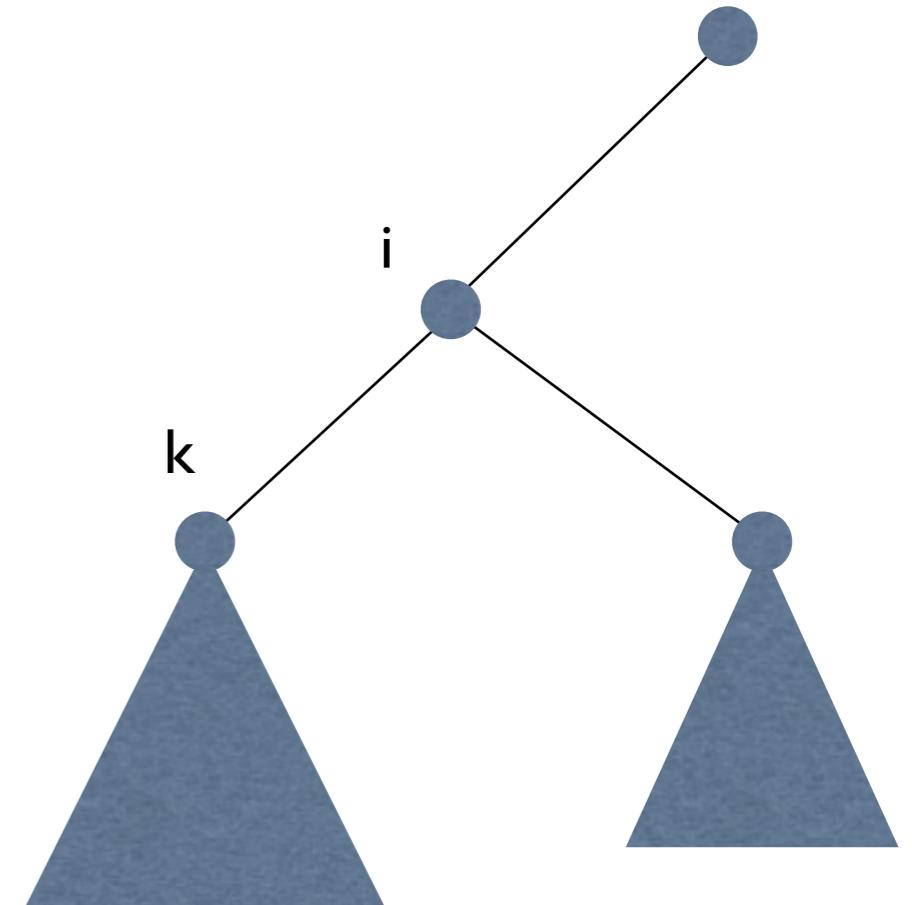
MAX-HEAPIFY( $A, i$ )

{

```
if there is no child of  $A[i]$ 
    return;
 $k \leftarrow$  index of the biggest child of  $i$ ;
if  $A[i] \geq A[k]$ 
    return;
exchange  $A[i]$  and  $A[k]$ ;
MAX-HEAPIFY( $A, k$ );
```

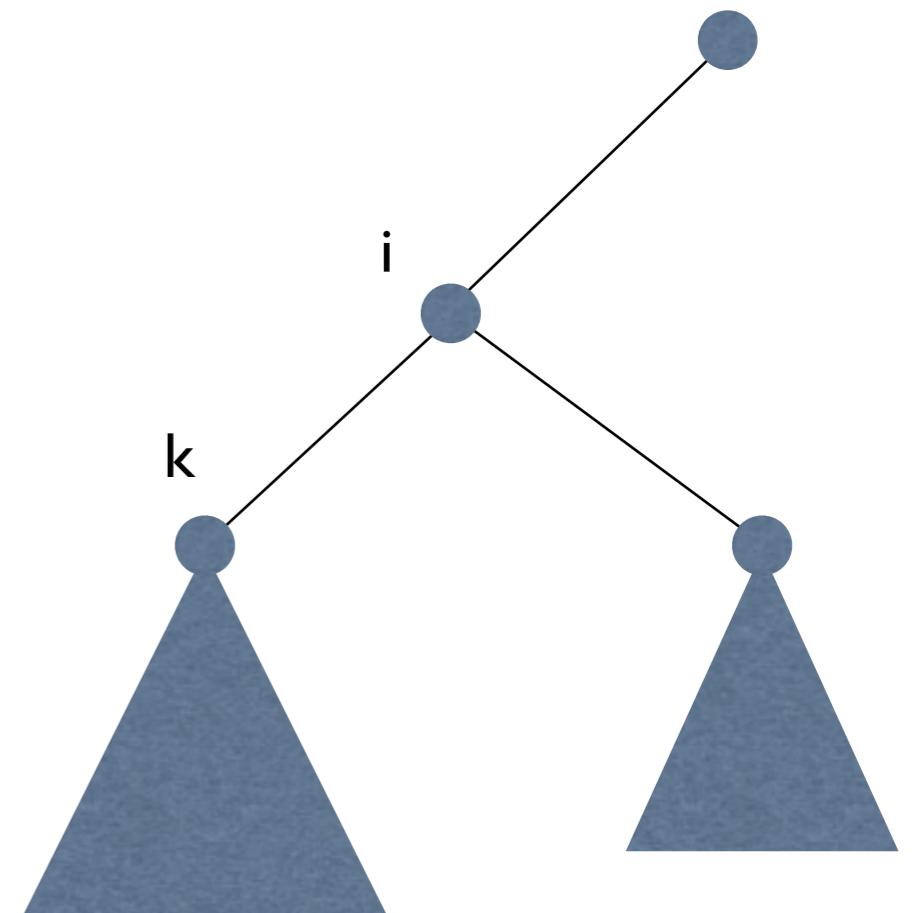
}

root 노드에 대한 heapify는  
MAX-HEAPIFY(1)을 호출하면 됨

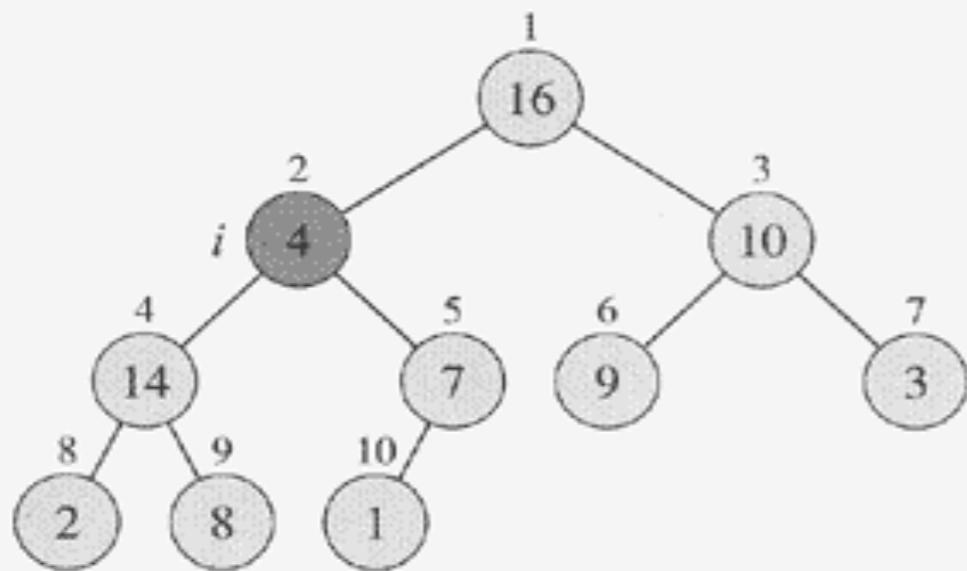


## MAX-HEAPIFY: iterative version

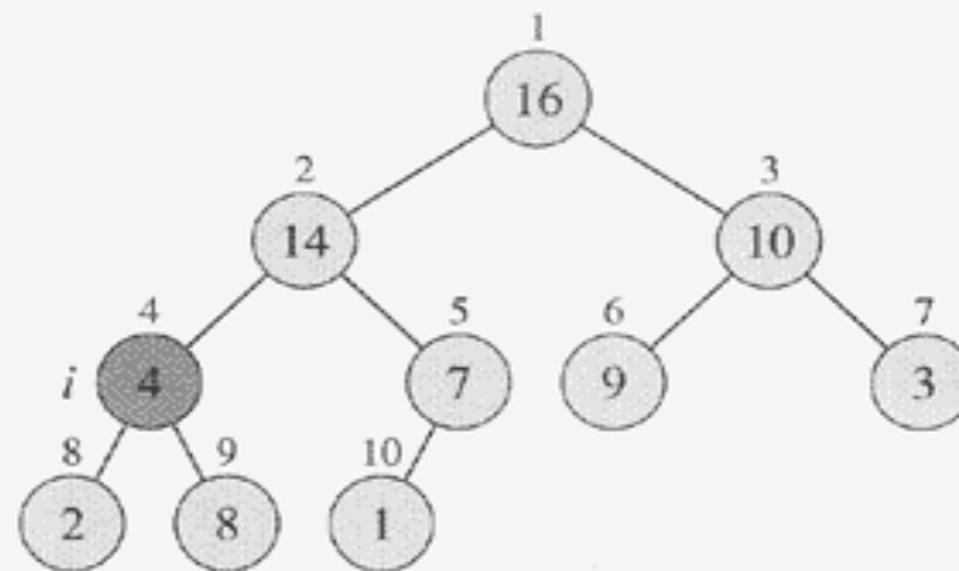
```
MAX-HEAPIFY(A, i)
{
    while A[i] has a child do
        k ← index of the biggest child of i;
        if A[i] ≥ A[k]
            return;
        exchange A[i] and A[k];
        i = k;
    end.
}
```



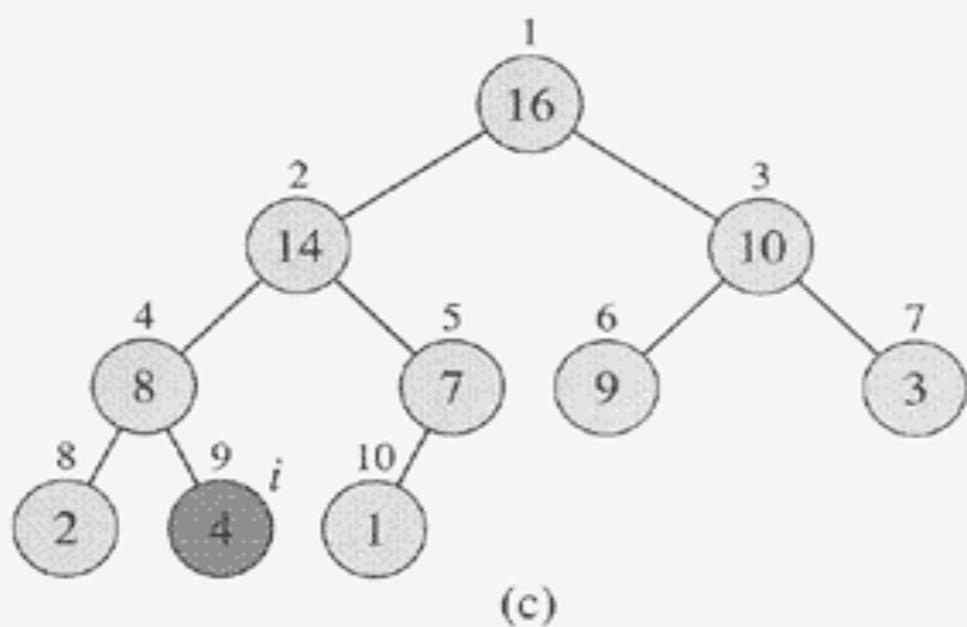
## MAX-HEAPIFY(A,2)



(a)



(b)



(c)

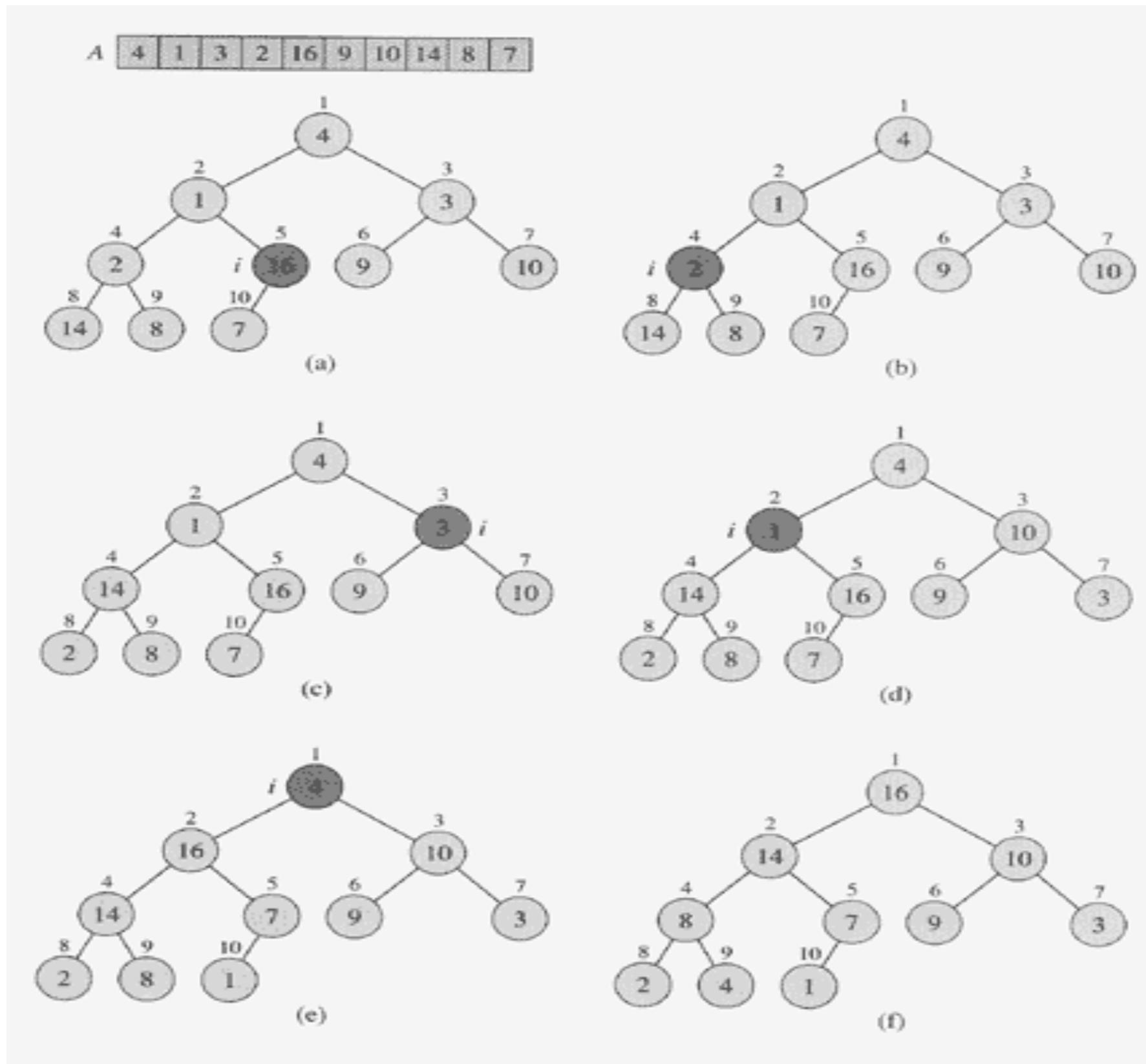
## 정렬할 배열을 힙으로 만들기

BUILD-MAX-HEAP( $A$ )

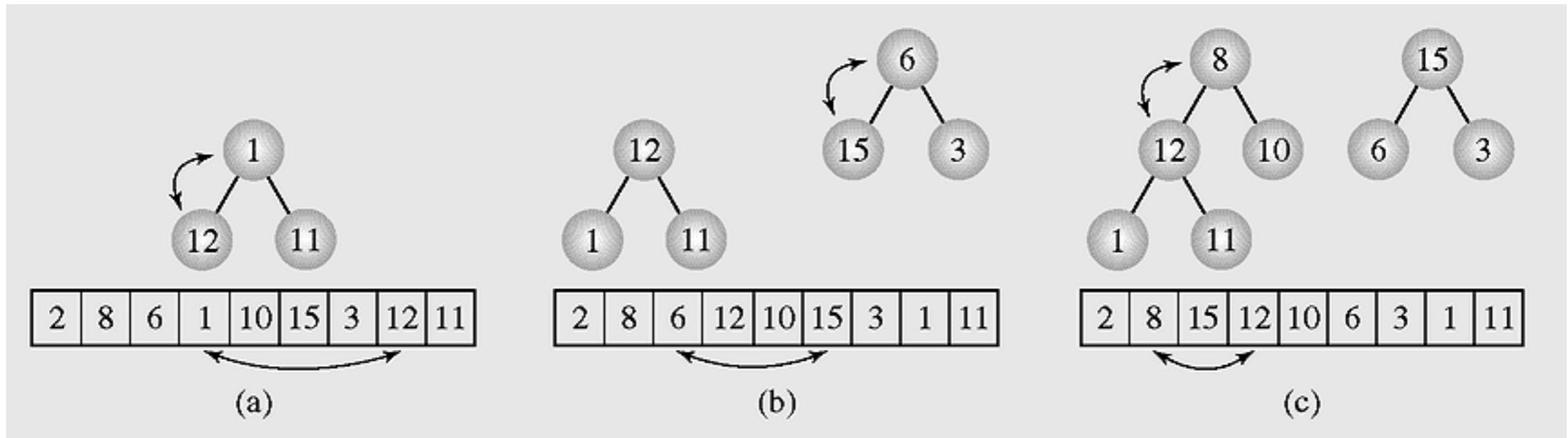
```
1   $heap\text{-}size}[A] \leftarrow length[A]$ 
2  for  $i \leftarrow \lfloor length[A]/2 \rfloor$  downto 1
3    do MAX-HEAPIFY( $A, i$ )
```

시간복잡도:  $O(n)$

# 정렬할 배열을 힙으로 만들기

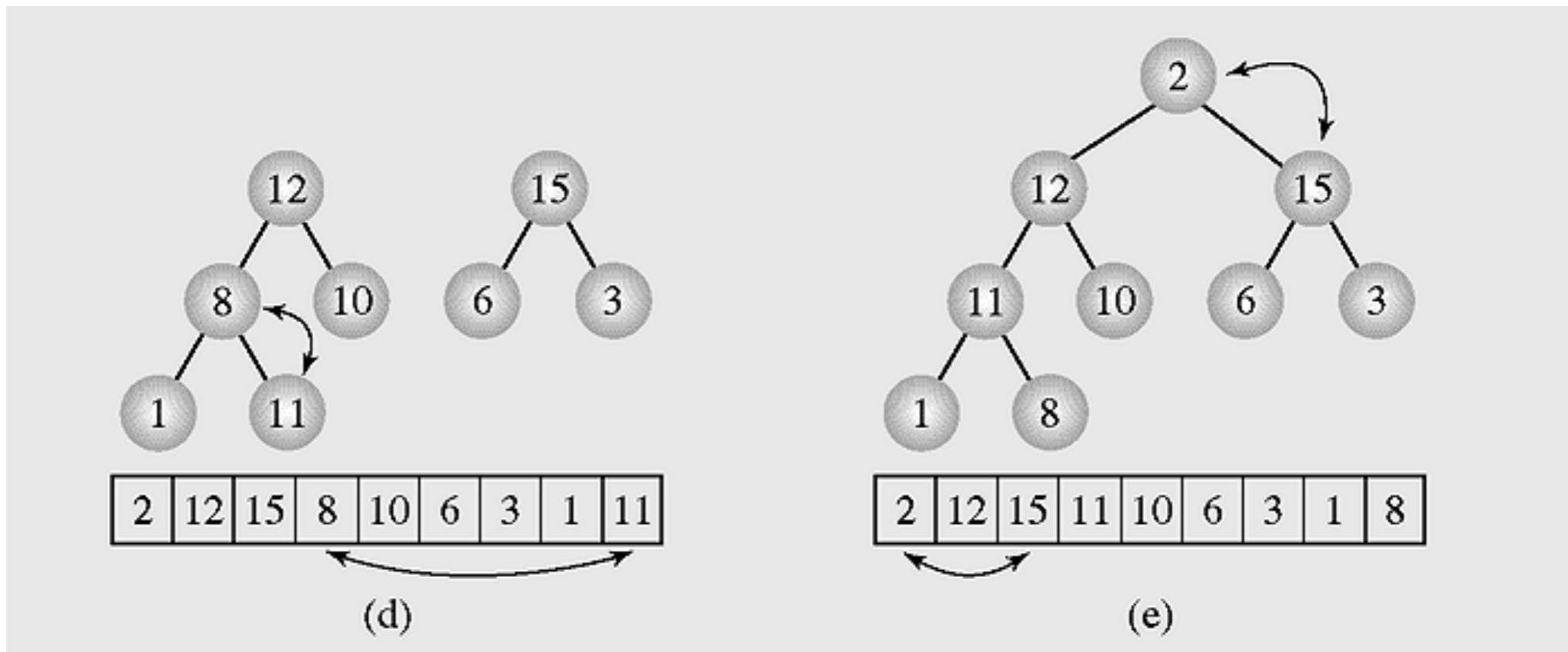


## 정렬할 배열을 힙으로 만들기



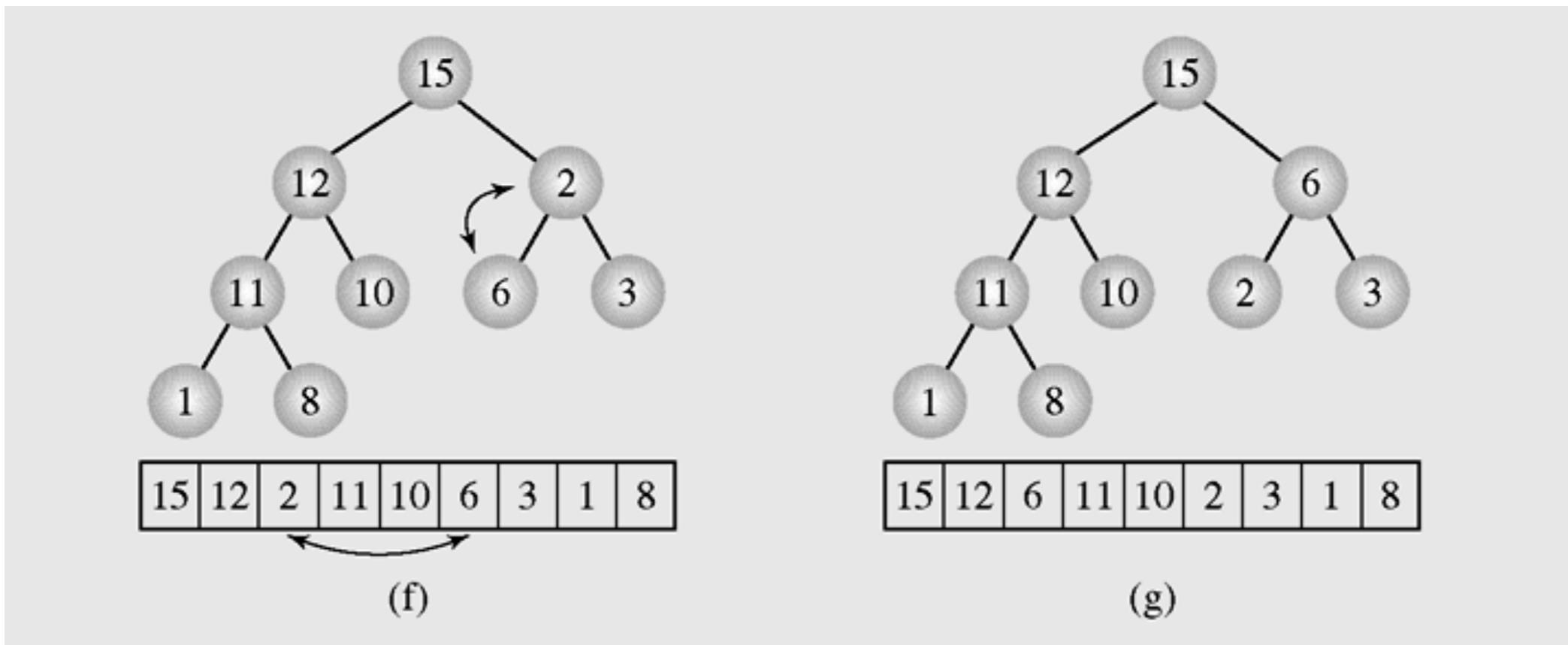
배열 [2, 8, 6, 1, 10, 15, 3, 12, 11] 을 힙으로 만드는 과정

## 정렬할 배열을 힙으로 만들기



배열 [2, 8, 6, 1, 10, 15, 3, 12, 11] 을 힙으로 만드는 과정

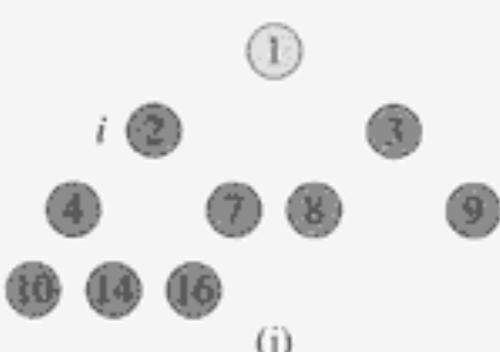
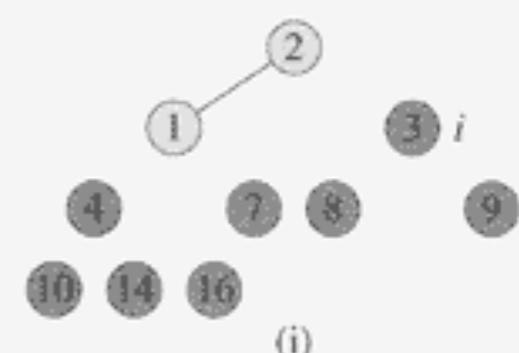
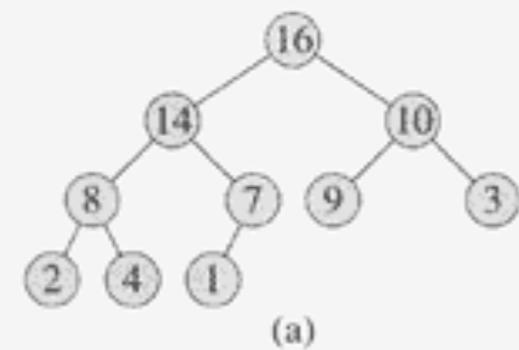
## 정렬할 배열을 힙으로 만들기



배열 [2, 8, 6, 1, 10, 15, 3, 12, 11] 을 힙으로 만드는 과정

- ⦿ 주어진 데이터로 힙을 만든다.
- ⦿ 힙에서 최대값(루트)을 가장 마지막 값과 바꾼다.
- ⦿ 힙의 크기가 1 줄어든 것으로 간주한다. 즉, 가장 마지막 값은 힙의 일부가 아닌 것으로 간주한다.
- ⦿ 루트노드에 대해서 HEAPIFY(1)한다.
- ⦿ 2~4번을 반복한다.

# Heapsort



<i>A</i>	1	2	3	4	7	8	9	10	14	16
----------	---	---	---	---	---	---	---	----	----	----

(k)

## Heapsort와 시간복잡도

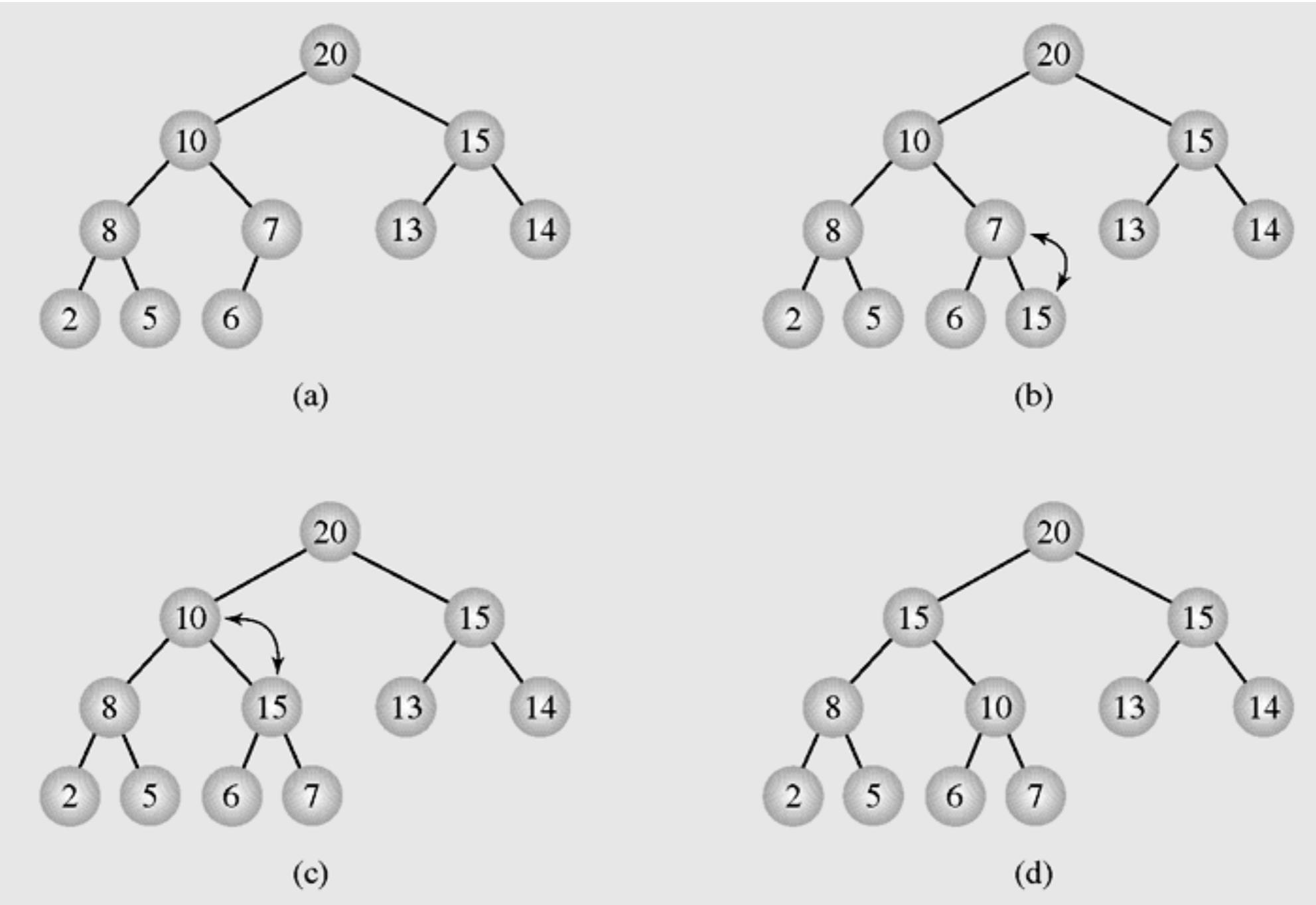
HEAPSORT(A)

1. BUILD-MAX-HEAP(A) :  $O(n)$
2. for  $i \leftarrow \text{heap\_size}$  downto 2 do :  $n-1$  times
3. exchange  $A[1] \leftrightarrow A[i]$  :  $O(1)$
4.  $\text{heap\_size} \leftarrow \text{heap\_size} - 1$  :  $O(1)$
5. MAX-HEAPFIY(A, 1) :  $O(\log_2 n)$

Total time:  $O(n \log_2 n)$

- ☞ 최대 우선순위 큐 (maximum priority queue)는 다음의 두 가지 연산을 지원하는 자료구조
  - ☞ INSERT( $x$ ): 새로운 원소  $x$ 를 삽입
  - ☞ EXTRACT\_MAX(): 최대값을 삭제하고 반환
- ☞ 최소 우선순위 큐 (minimum priority queue)는 EXTRACT-MAX 대신 EXTRACT-MIN을 지원하는 자료구조
- ☞ MAX HEAP을 이용하여 최대 우선순위 큐를 구현

# INSERT

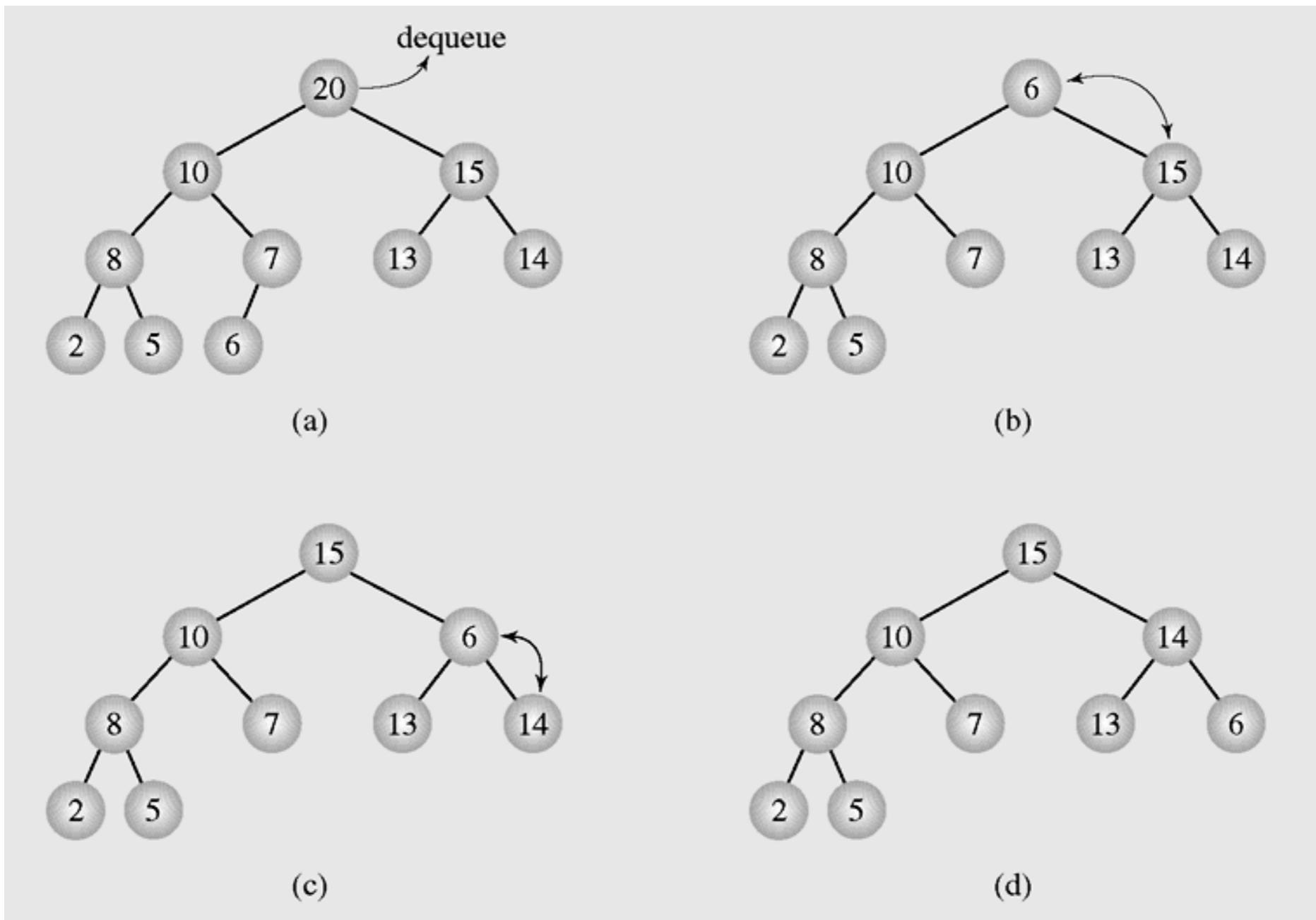


## INSERT

```
MAX-HEAP-INSERT(A, key)    {
    heap_size = heap_size+1;
    A[heap_size] = key;
    i = heap_size;
    while ( i>1 and A[PARENT(i)] < A[i] )    {
        exchange A[i] and A[PARENT(i)];
        i = PARENT(i);
    }
}
```

시간복잡도  $O(\log_2 n)$

## EXTRACT\_MAX()



## EXTRACT\_MAX()

HEAP-EXTRACT-MAX( $A$ )

- 1   **if**  $heap\text{-size}[A] < 1$
- 2       **then error** “heap underflow”
- 3     $max \leftarrow A[1]$
- 4     $A[1] \leftarrow A[heap\text{-size}[A]]$
- 5     $heap\text{-size}[A] \leftarrow heap\text{-size}[A] - 1$
- 6    MAX-HEAPIFY( $A, 1$ )
- 7   **return**  $max$

시간복잡도  $O(\log_2 n)$

# Comparison Sort

## Comparison sort

- 데이터들간의 **상대적 크기관계**만을 이용해서 정렬하는 알고리즘
- 따라서 데이터들간의 크기 관계가 정의되어 있으면 어떤 데이터에든 적용가능 (문자열, 알파벳, 사용자 정의 객체 등)
- 버블소트, 삽입정렬, 합병정렬, 퀵소트, 힙정렬 등

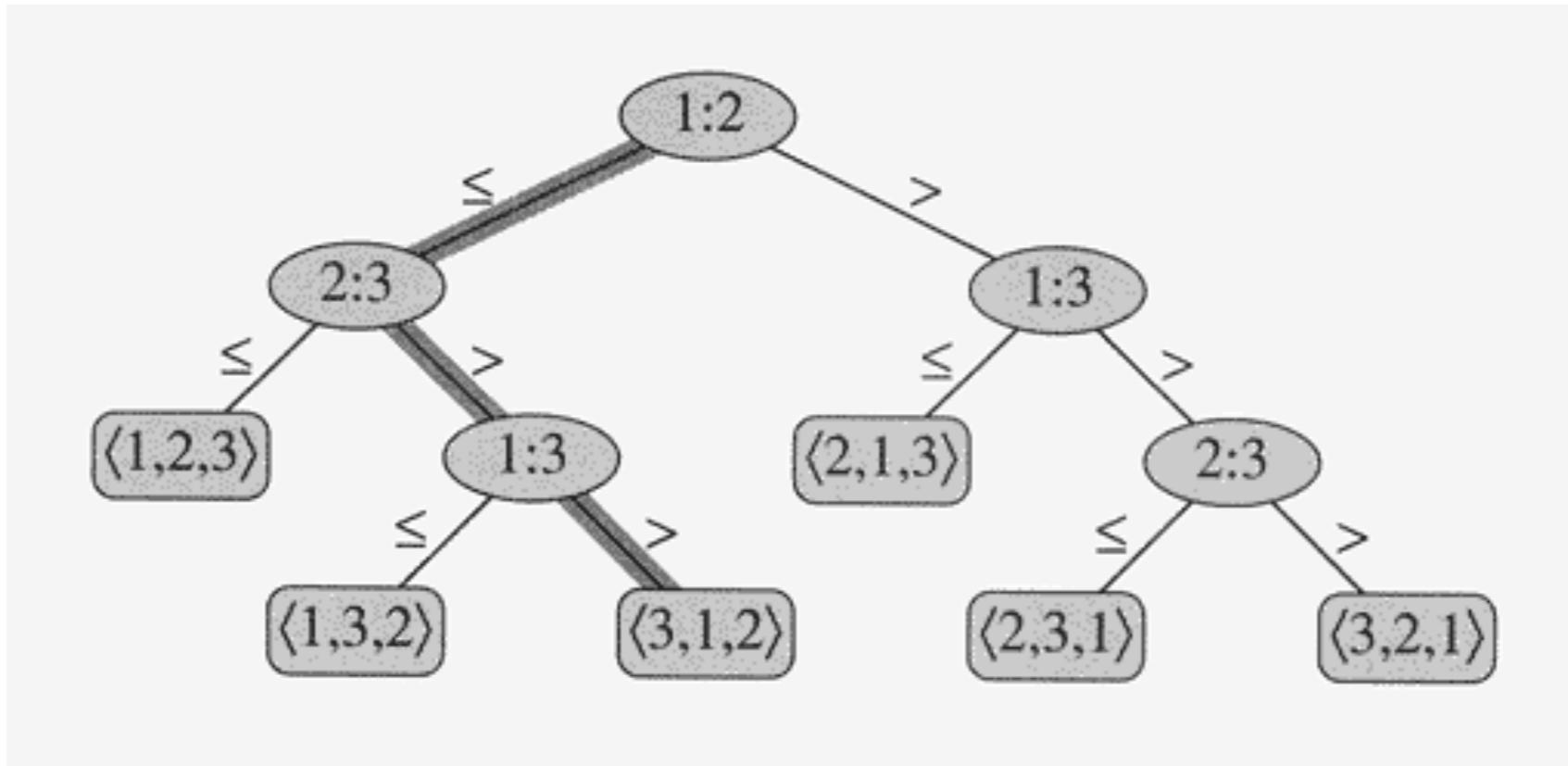
## Non-comparison sort

- 정렬할 데이터에 대한 **사전지식**을 이용 - 적용에 제한
- Bucket sort
- Radix sort

## ④ 하한 (Lower bound)

- 입력된 데이터를 한번씩 다 보기 위해서 최소  $\Theta(n)$ 의 시간복잡도 필요
- 합병정렬과 힙정렬 알고리즘들의 시간복잡도는  $\Theta(n \log_2 n)$
- 어떤 comparison sort 알고리즘도  $\Theta(n \log_2 n)$ 보다 나을 수 없다.

- Abstraction of any comparison sort.



3개의 값을 정렬하는 삽입정렬알고리즘에  
대한 decision tree

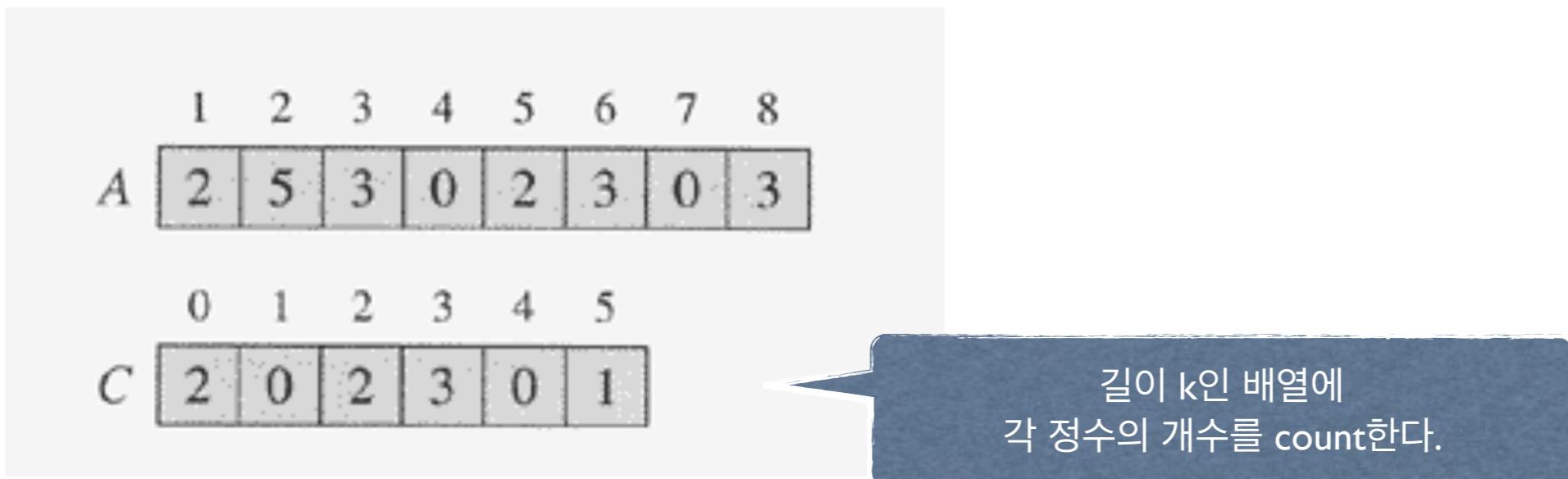
- ⦿ Leaf노드의 개수는  $n!$ 개. 왜냐하면 모든 순열(permuation)에 해당하므로
- ⦿ 최악의 경우 시간복잡도는 트리의 높이
- ⦿ 트리의 높이는
  - ⦿  $\text{height} \geq \log_2 n! = \Theta(n \log_2 n)$

# 선형시간 정렬 알고리즘

## ☞ Counting Sort

- ☞  $n$ 개의 정수를 정렬하라. 단 모든 정수는 0에서  $k$ 사이의 정수이다.
- ☞ 예:  $n$ 명의 학생들의 시험점수를 정렬하라. 단 모든 점수는 100이하의 양의 정수이다.

## ④ k=5인 경우의 예



0 두개, 2 두개...



0	0	2	2	3	3	3	5
---	---	---	---	---	---	---	---

## Counting Sort

```
int A[n];
int C[k] = {0,};
for ( int i=1; i<=n; i++ )
    C[A[i]]++;
for ( int s=1, i=0; i<=k; i++ )    {
    for ( int j=0; j<C[i]; j++ )    {
        A[s++] = j;
    }
}
```

Is it okay?

No. 대부분의 경우 정렬할 key값  
들은 레코드의 일부분이기 때문



## Counting Sort

COUNTING-SORT( $A, B, k$ )

```
1  for  $i \leftarrow 0$  to  $k$ 
2      do  $C[i] \leftarrow 0$ 
3  for  $j \leftarrow 1$  to  $\text{length}[A]$ 
4      do  $C[A[j]] \leftarrow C[A[j]] + 1$ 
5   $\triangleright C[i]$  now contains the number of elements equal to  $i$ .
6  for  $i \leftarrow 1$  to  $k$ 
7      do  $C[i] \leftarrow C[i] + C[i - 1]$ 
8   $\triangleright C[i]$  now contains the number of elements less than or equal to  $i$ .
9  for  $j \leftarrow \text{length}[A]$  downto 1
10     do  $B[C[A[j]]] \leftarrow A[j]$ 
11      $C[A[j]] \leftarrow C[A[j]] - 1$ 
```

- ⦿  $\Theta(n+k)$ , 또는  $\Theta(n)$  if  $k=O(n)$ .
- ⦿  $k$ 가 클 경우 비실용적
- ⦿ **Stable 정렬 알고리즘**
  - ⦿ “입력에 동일한 값이 있을때 입력에 먼저 나오는 값이 출력에서도 먼저 나온다”
  - ⦿ Counting정렬은 stable하다.

- n개의 d자리 정수들
- 가장 낮은 자리수부터 정렬

329	720	720	329
457	355	329	355
657	436	436	436
839	.....	457	.....
436	657	355	657
720	329	457	720
355	839	657	839

```
RADIX-SORT( $A, d$ )
```

```
1   for  $i \leftarrow 1$  to  $d$ 
2       do use a stable sort to sort array  $A$  on digit  $i$ 
```

시간 복잡도  $O(d(n+k))$

비교

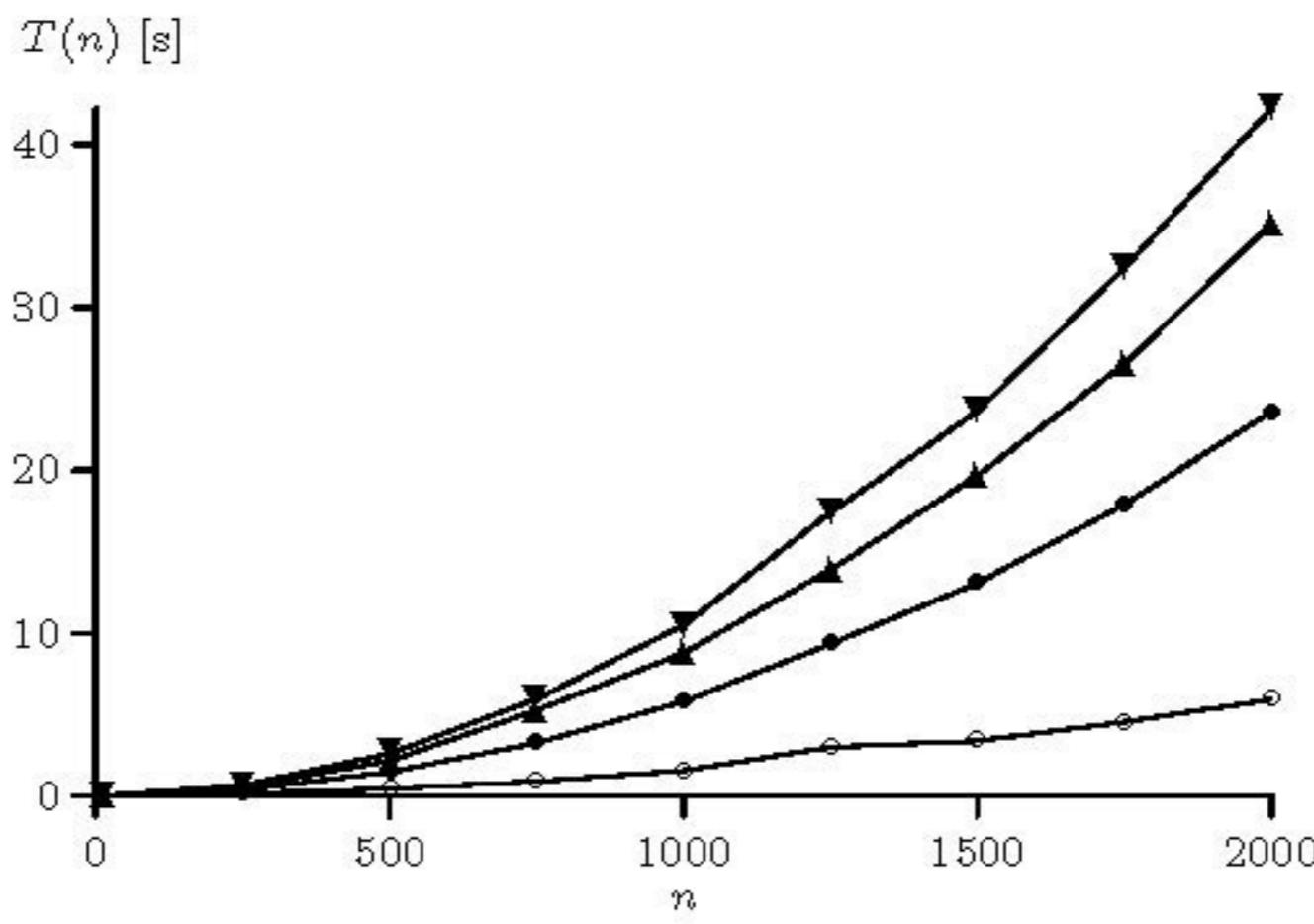
# 정렬 알고리즘들

Sort Algorithm	$T(n)$
Bubble sort	$\Theta(n^2)$
Insertion sort	$\Theta(n^2)$
Selection sort	$\Theta(n^2)$
Quick sort	worst $\Theta(n^2)$ and average $\Theta(n \log_2 n)$
Merge sort	$\Theta(n \log_2 n)$
Heap sort	$\Theta(n \log_2 n)$
Counting sort	$\Theta(n+k)$
Radix sort	$\Theta(d(n+k))$

---

Figure 15.14: Actual running times of the  $\mathcal{O}(n^2)$  sorts.

---



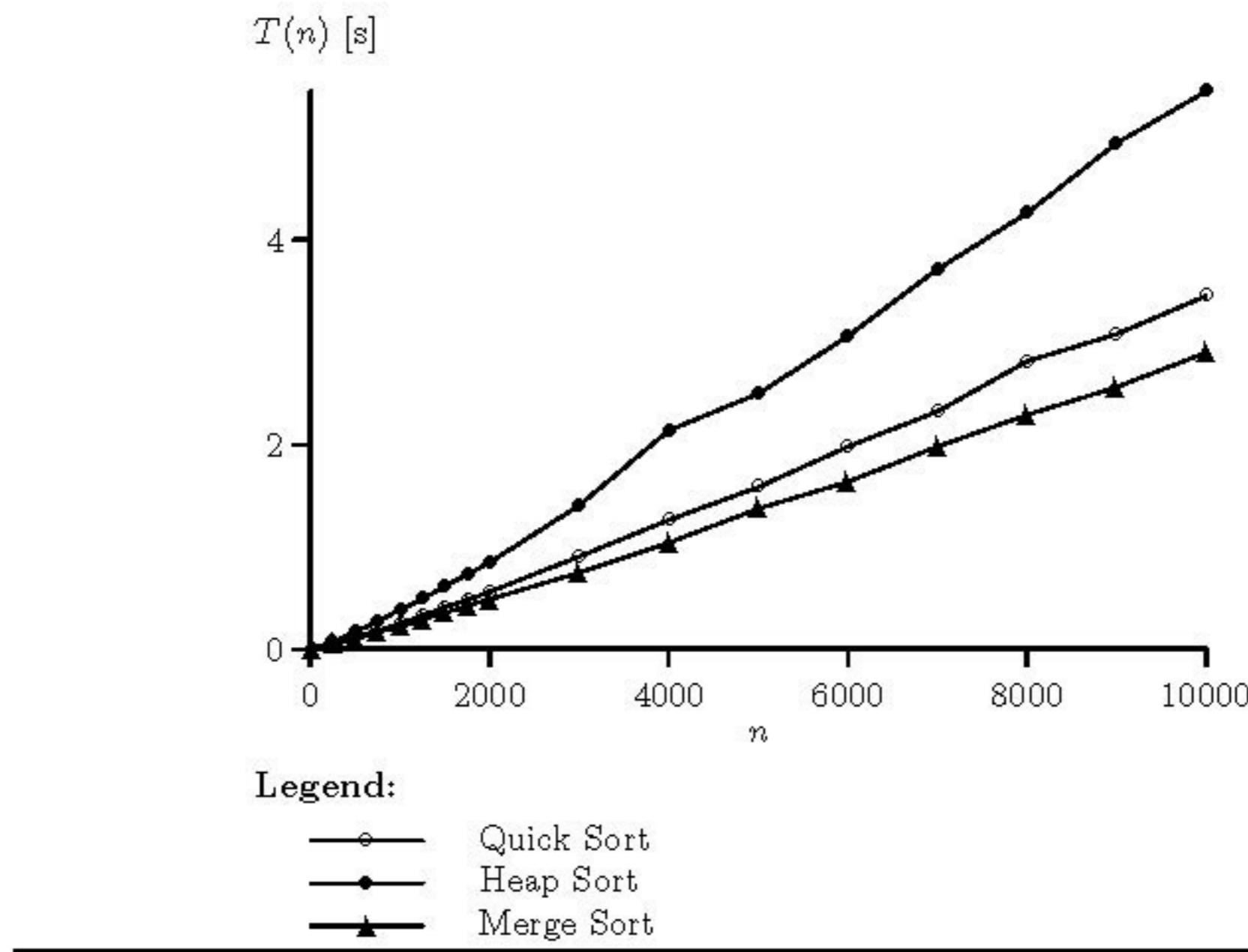
Legend:

- Binary Insertion Sort
- Straight Insertion Sort
- ▲— Straight Selection Sort
- ▼— Bubble Sort

---

Figure 15.15: Actual running times of the  $O(n \log n)$  sorts.

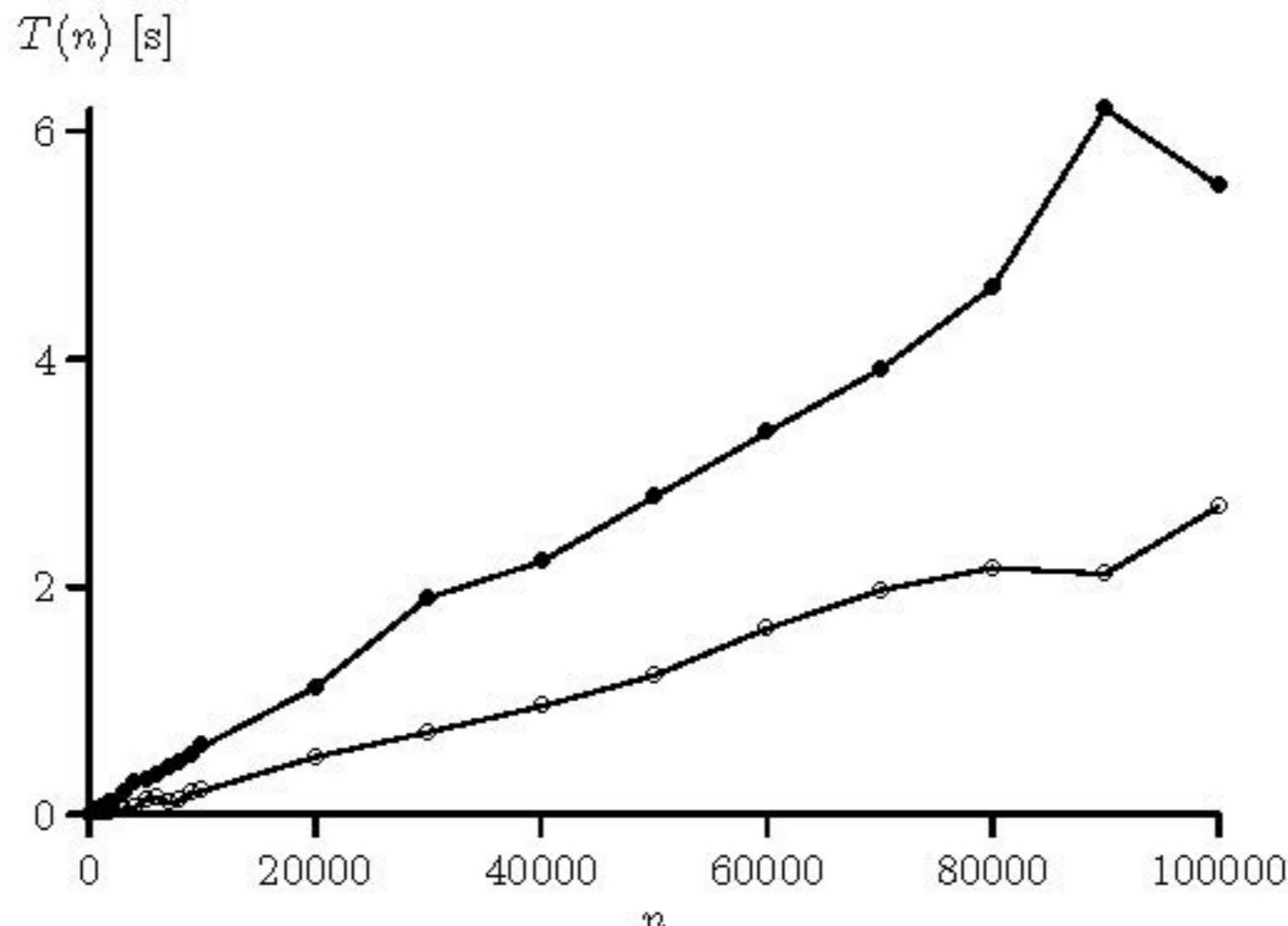
---



---

Figure 15.16: Actual running times of the  $\mathcal{O}(n)$  sorts.

---



Legend:

- Bucket Sort ( $m = 1024$ )
  - Radix Sort ( $R = 256, p = 4$ )
-

# Sorting in Java

- Arrays 클래스가 primitive 타입 데이터를 위한 정렬 메서드를 제공

```
int [] data = new int [capacity];  
  
// data[0]에서 data[capacity-1]까지 데이터가 꽉 차있는 경우  
Arrays.sort(data);  
  
// data[0]에서 data[size-1]까지 size개의 데이터만 있는 경우  
Arrays.sort(data, 0, size);
```

- int 이외의 다른 primitive 타입 데이터(double, char 등)에 대해서도 제공

## 객체의 정렬: 문자열

```
String [] fruits = new String [] { "Pineapple", "Apple",
                                  "Orange", "Banana"};
Arrays.sort(fruits);
for(String name: fruits)
    System.out.println(name);
```

Primitive 타입 데이터와 마찬가지로  
Arrays.sort 메서드로 정렬된다.

## ArrayList 정렬: 문자열

```
List<String> fruits = new ArrayList<String>();  
fits.add("Pineapple");  
fits.add("Apple");  
fits.add("Orange");  
fits.add("Banana");  
  
Collections.sort(fruits);  
  
for(String name: fruits)  
    System.out.println(name);
```

Collections.sort 메서드로 정렬된다.

## 객체의 정렬: 사용자 정의 객체

```
public class Fruit {  
    public String name;  
    public int quantity;  
    public Fruit(String name, int quantity) {  
        this.name = name;  
        this.quantity = quantity;  
    }  
}
```

```
// somewhere in your program  
Fruit [] fruits = new Fruit[4];  
fruits[0] = new Fruit("Pineapple",70);  
fruits[1] = new Fruit("Apple",100);  
fruits[2] = new Fruit("Orange",80);  
fruits[3] = new Fruit("Banana",90);
```

```
Arrays.sort(fruits);
```

어떻게 될까?

## 객체의 정렬: Comparable Interface

```
public class Fruit implements Comparable<Fruit> {  
    public String name;  
    public int quantity;  
    public Fruit(String name, int quantity) {  
        this.name = name;  
        this.quantity = quantity;  
    }  
  
    public int compareTo(Fruit other) {  
        return name.compareTo(other.name);  
    }  
}  
  
// somewhere in your program  
Fruit [] fruits = new Fruit[4];  
fruits[0] = new Fruit("Pineapple",70);  
fruits[1] = new Fruit("Apple",100);  
fruits[2] = new Fruit("Orange",80);  
fruits[3] = new Fruit("Banana",90);  
  
Arrays.sort(fruits);
```

이름 순으로 정렬된다.

## 만약 재고수량으로 정렬하고 싶다면?

```
public class Fruit implements Comparable<Fruit> {  
    public String name;  
    public int quantity;  
    public Fruit(String name, int quantity) {  
        this.name = name;  
        this.quantity = quantity;  
    }  
    public int compareTo(Fruit other) {  
        return quantity - other.quantity;  
    }  
}  
  
// somewhere in your program  
Fruit [] fruits = new Fruit[4];  
fruits[0] = new Fruit("Pineapple",70);  
fruits[1] = new Fruit("Apple",100);  
fruits[2] = new Fruit("Orange",80);  
fruits[3] = new Fruit("Banana",90);  
Arrays.sort(fruits);
```

재고수량 순으로 정렬된다.

## 두 가지 기준을 동시에 지원하려면?

- ❶ 하나의 객체 타입에 대해서 2가지 이상의 기준으로 정렬을 지원하려면
- ❷ Comparator를 사용

## Comparator

Comparator 클래스를 extends하며 compare 메서드를 overriding하는 새로운 이름 없는 클래스를 정의한 후 그 클래스의 객체를 하나 생성한다.

```
Comparator<Fruit> nameComparator = new Comparator<Fruit>() {  
    public int compare(Fruit fruit1, Fruit fruit2) {  
        return fruit1.name.compareTo(fruit2.name);  
    }  
};
```

```
Comparator<Fruit> quantComparator = new Comparator<Fruit>() {  
    public int compare(Fruit fruit1, Fruit fruit2) {  
        return fruit1.quantity - fruit2.quantity;  
    }  
};
```

```
Arrays.sort(fruits, nameComparator);  
// or  
Arrays.sort(fruits, quantComparator);
```

Comparator 객체들을 어디에 둘 것인가?

# Comparator

```
public class Fruit {  
    public String name;  
    public int quantity;  
    public Fruit(String name, int quantity) {  
        this.name = name;  
        this.quantity = quantity;  
    }  
  
    public static Comparator<Fruit> nameComparator = new Comparator<Fruit>() {  
        public int compare(Fruit fruit1, Fruit fruit2) {  
            return fruit1.name.compareTo(fruit2.name);  
        }  
    };  
  
    public static Comparator<Fruit> quantComparator = new Comparator<Fruit>(){  
        public int compare(Fruit fruit1, Fruit fruit2) {  
            return fruit1.quantity - fruit2.quantity;  
        }  
    };  
}
```

데이터 객체의 static member로 둔다. 정렬할 때는  
Arrays.sort(fruits, Fruit.nameComparator);

# Sorting in C

# 함수 포인터

- 함수 역시 메모리의 일정영역에 저장되므로 모든 함수는 주소를 가진다.
- C언어에서는 데이터에 대한 포인터와 마찬가지로 함수에 대한 포인터를 정의하고 사용할 수 있다.
- 함수 포인터를 매개변수로 사용하는 경우는 매우 흔하다.
  - 이벤트 핸들러 혹은 call-back 함수
  - Generic programming

## 함수 포인터를 매개변수로 사용하는 예

- 함수 `integrate`는 매개변수로 주어지는 임의의 함수 `f`를 지정된 구간에 대해 적분하는 함수이다.

```
double integrate(double (*f)(double), double a, double b);
```

- 여기서 `*f`를 둘러싼 괄호는 `f`가 함수 포인터임을 표시
- 다른 방법:

```
double integrate(double f(double), double a, double b);
```

## 함수 포인터를 매개변수로 사용하는 예

- 함수 `integrate`를 이용하여 `sin`함수를 0에서  $\pi/2$  까지 적분:

```
result = integrate(sin, 0.0, PI/2);
```

- 위의 예에서 `sin`처럼 함수의 이름에 괄호가 따라오지 않을 경우 컴파일러는 함수 포인터를 생성한다.
- `integrate`함수 내에서는 함수 포인터 `f`를 이용하여 `f`가 가리키는 함수를 다음과 같이 호출한다.

```
y = (*f)(x);
```

- `(*f)(x)` 대신 그냥 `f(x)`로 사용 가능

# qsort 함수

- ☞ qsort는 <stdlib.h>에 정의된 빠른 정렬 알고리즘
- ☞ 어떤 타입의 배열이든 정렬할 수 있는 generic-정렬 알고리즘
- ☞ qsort를 호출하기 위해서는 배열에 저장된 임의의 두 원소의 크기관계를 어떻게 결정하는지 알려줘야 한다.
- ☞ 데이터의 크기관계를 비교하는 함수의 포인터를 qsort에게 매개변수로 넘겨줌으로써 가능하다.
- ☞ 크기 비교 함수
  - ☞ 배열 상의 임의의 두 원소의 주소 p와 q가 주어졌을 때 다음과 같이 반환한다.
    - ☞ Negative      if \*p is “less than” \*q
    - ☞ Zero            if \*p is “equal to” \*q
    - ☞ Positive        if \*p is “greater than” \*q

# qsort 함수

## ☞ qsort의 프로토타입:

```
void qsort(void *base, size_t nmemb, size_t size,  
          int (*compar)(const void *, const void *) );
```

비교 함수가 두 개의 매개변수를 받고 정수값을  
반환하는 함수라는 것을 표시한다.

- ☞ base : 배열에서 정렬할 첫번째 원소의 주소
- ☞ nmemb : 정렬할 원소의 개수
- ☞ size : 정렬할 각 원소의 크기 (바이트 수)
- ☞ compar : 비교 함수의 포인터

## qsort 함수

```
typedef struct fruit {
    char *name;
    int quantity;
} Fruit;

Fruit fruits[MAX]; // 배열 fruits[0]…fruits[n-1]에 데이터가 저장되어 있다고 가정
int n;

int compare_fruits_by_name(const void *p, const void *q) {
    return strcmp(((Fruit *)p)->name, ((Fruit *)q)->name);
}

int compare_fruits_by_qty(const void *p, const void *q) {
    return ((Fruit *)p)->quantity - ((Fruit *)q)->quantity;
}

int main()
{
    ...
    qsort(fruits, n, sizeof(Fruit), compare_fruits_by_name);
    ...
    qsort(fruits, n, sizeof(Fruit), compare_fruits_by_qty);
    ...
}
```