04장. 정렬

Youn-Hee Han
LINK@KOREATECH
http://link.koreatech.ac.kr

은유, 그것은 정신적 상호 연관성의
피륙을 짜는 방법이다.
은유는 살아있다는 것의 바탕이다.

-그레고리 베이트슨(언어학자, 기호학자)

<u>은유(metaphor)</u>
직유보다 한 단계 발전된 비유법으로
사물의 본뜻을 숨기고 주로 보조 관념들 만을 간단하게 제시하는 것
은근한 비유로 직유보다 더 인상적인 표현을 할 수 있는 것이 은유법이지만
은유를 남용하면 문맥이 어지럽고 문장의 뜻이 모호해질 수 있으므로 주의해야 한다.

'그대의 눈은 샛별같이 밝다 ' → 직유
'그대의 눈은 샛별이다' → 은유

학습 목표

- 기본 정렬 알고리즘을 이해한다.

- 정렬을 귀납적 관점에서 볼 수 있도록 한다.
  - 보편성에서 구체성을 유도하는 추론

- 2~3장에서 배운 기법을 사용해 각 정렬의 수행 시간을 분석할 수 있도록 한다.

- 비교 정렬의 한계를 이해하고, 선형 시간 정렬이 가능한 조건과 선형 시간 정렬 알고리즘을 이해한다.
  - 비교 정렬: 대부분 $O(n^2)$과 $O(nlogn)$ 사이
  - Input이 특수한 성질을 만족하는 경우: $O(n)$

01. 기본적인 정렬 알고리즘

기본적인 정렬 알고리즘

◈ 평균적으로 $\Theta(n^2)$의 시간이 소요되는 정렬 알고리즘들

- 선택정렬 (Selection Sorting)
- 버블정렬 (Bubble Sorting)
- 삽입정렬 (Insertion Sorting)

선택 정렬

◈ 선택 정렬 의사 코드

알고리즘 4-1	선택 정렬

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

- 각 루프마다
  - 최대 원소를 찾는다
  - 최대 원소와 (아직 정렬이 되지 않은) 맨 오른쪽 원소를 교환한다
  - 맨 오른쪽 원소를 (정렬이 된 것으로) 제외한다
- 하나의 원소만 남을 때까지 위의 루프를 반복

선택 정렬

◈ 선택 정렬 수행 예

알고리즘 4-1 | 선택 정렬

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

정렬할 배열이 주어진다

8	31	48	73	3	65	20	29	11	15

가장 큰 수를 찾는다(73)

8	31	48	73	3	65	20	29	11	15

73을 맨 오른쪽 수(15)와 자리 바꾼다

8	31	48	15	3	65	20	29	11	73

❶의 첫 번째 루프

맨 오른쪽 수를 제외한 나머지에서 가장 큰 수를 찾는다(65)

8	31	48	15	3	65	20	29	11	73

65를 맨 오른쪽 수(11)와 자리 바꾼다

8	31	48	15	3	11	20	29	65	73

❶의 두 번째 루프

맨 오른쪽 두 수를 제외한 나머지에서 가장 큰 수를 찾는다(48)

8	31	48	15	3	11	20	29	65	73

⋮

앞의 작업을 반복하면서 계속 제외해나간다

⋮

8을 맨 오른쪽 수(3)와 자리 바꾼다

8	3	11	15	20	29	31	48	65	73

정렬이 완료된 최종 배열

3	8	11	15	20	29	31	48	65	73

선택 정렬

◈ 선택 정렬의 점근적 복잡도 분석

알고리즘 4-1	선택 정렬

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

- **수행 시간 분석:**
  - **①의 for 루프는 n-1번 반복**
  - **②에서 가장 큰 수를 찾기 위한 비교횟수:** $n-1, n-2, \cdots, 2, 1$
  - **③의 교환은 상수 시간 작업**
- $T(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \boldsymbol{\Theta(n^2)}$

Worst case

Every case

버블 정렬

◈ 버블 정렬 의사 코드

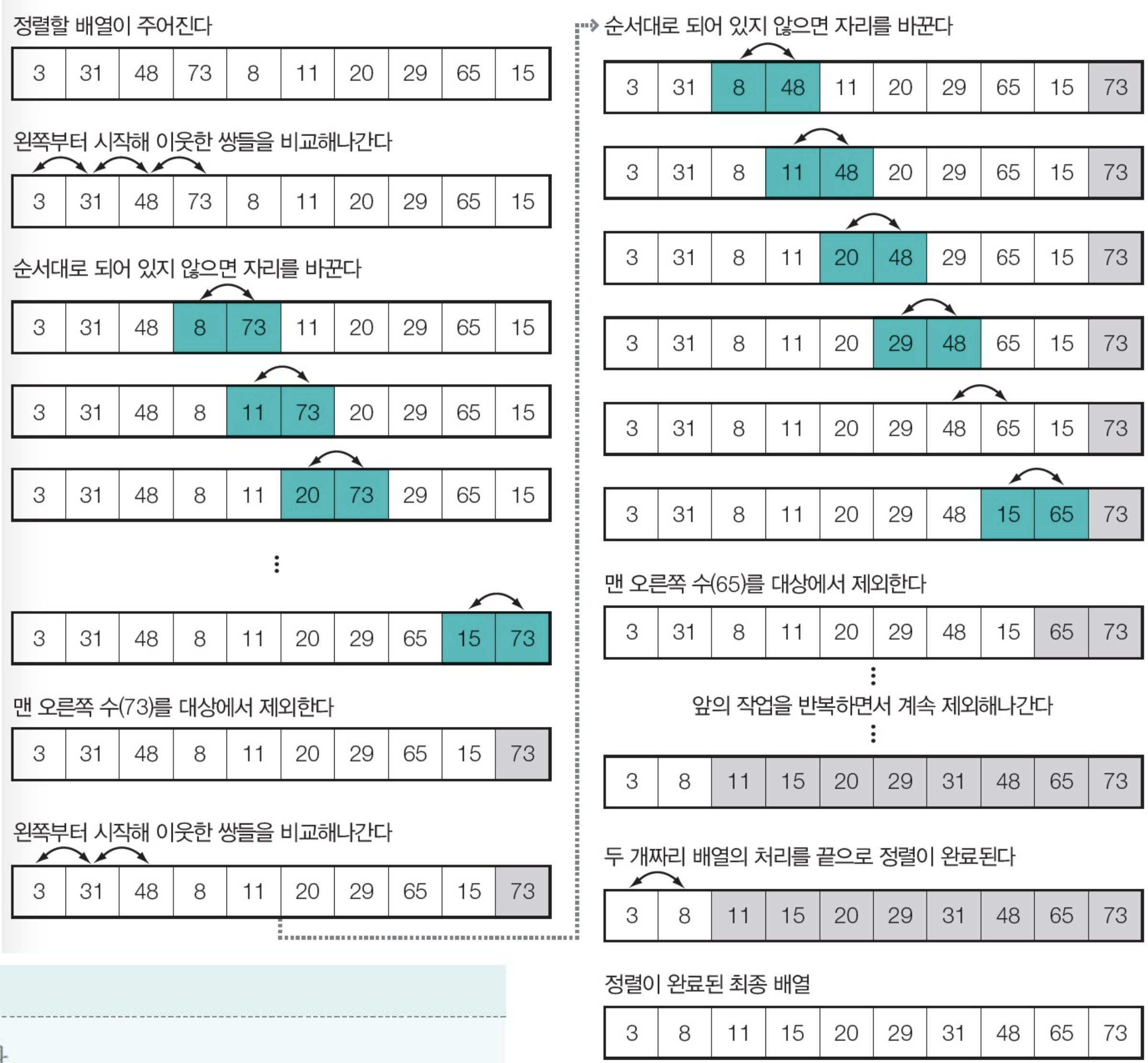
알고리즘 4-2 버블 정렬

```
bubbleSort(A[], n)        ▷ A[1 … n]을 정렬한다.
{
  ❶ for last ← n downto 2
      ❷ for i ← 1 to last−1
          ❸ if (A[i] > A[i+1]) then A[i] ↔ A[i+1];      ▷ 원소 교환
}
```

- 바깥쪽 루프마다 제일 큰 원소를 맨 오른쪽으로 보내고 정렬할 배열의 크기를 하나씩 줄인다.
- 안쪽 루프마다 정렬할 배열 내에서 왼쪽부터 이웃한 수를 비교하면서 순서가 제대로 되어 있지 않으면 바꾸어 나간다.

버블 정렬

◈ 버블 정렬 수행 예



알고리즘 4-2 | **버블 정렬**

```
bubbleSort(A[], n)      ▷ A[1 ··· n]을 정렬한다.
{
  ❶ for last ← n downto 2
      ❷ for i ← 1 to last−1
          ❸ if (A[i] > A[i+1]) then A[i] ↔ A[i+1];      ▷ 원소 교환
}
```

버블 정렬

◈ 버블 정렬의 점근적 복잡도 분석

알고리즘 4-2	버블 정렬

```
bubbleSort(A[], n)      ▷ A[1 … n]을 정렬한다.
{
  ❶ for last ← n downto 2
      ❷ for i ← 1 to last−1
          ❸ if (A[i] > A[i+1]) then A[i] ↔ A[i+1];      ▷ 원소 교환
}
```

– **수행 시간 분석:**
  - **①의 for 루프는 $n-1$번 반복**
  - **②의 for 루프내 비교는 $n-1, n-2, \cdots, 2, 1$번 반복**
  - **③의 원소 교환은 상수 시간 작업**

– $T(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \boldsymbol{\Theta(n^2)}$

Worst case

Every case

버블 정렬

◈ 변형된 버블 정렬과 점근적 복잡도 분석

```
bubbleSort(A[], n)    ▷ A[1 … n]을 정렬한다
{
        for last ← n downto 2 {
                sorted = TRUE;
                for i ← 1 to last-1
                        if (A[i] > A[i+1]) then {
                                A[i] ↔ A[i+1];
                                sorted = FALSE;
                        }
                if (sorted == TRUE) then return
        }
}
```

이미 정렬이 되어 있는 배열인 경우 첫번째 바깥 루프 수행에서 안쪽 루프 수행을 통한 n-1번 비교 이후 곧바로 종료

- Worst Case
  - $T(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \boldsymbol{\Theta(n^2)}$
- Best Case: 이미 정렬이 되어 있는 배열인 경우
  - $T(n) = n - 1 = \boldsymbol{\Theta(n)}$

삽입 정렬

◈ 삽입 정렬 의사 코드

알고리즘 4-3 | **삽입 정렬**

```
insertionSort(A[], n)      ▷ A[1 … n]을 정렬한다.
{
  ❶ for i ← 2 to n
      ❷ A[1 … i]의 적합한 자리에 A[i]를 삽입한다;
}
```

- 바깥쪽 루프마다 이미 정렬된 배열의 크기를 하나씩 늘려 나간다.
- 안쪽 루프마다 정렬할 배열 내에서 i번째 원소의 위치를 찾아 삽입한다.

삽입 정렬

◈ 삽입 정렬 수행 예

알고리즘 4-3 | **삽입 정렬**

insertionSort(A[], n) ▷ $A[1 \cdots n]$을 정렬한다.
{
  ❶ for $i \leftarrow 2$ to n
    ❷ $A[1 \cdots i]$의 적합한 자리에 $A[i]$를 삽입한다;
}

삽입 정렬

◈ 삽입 정렬 수행 예

알고리즘 4-3	삽입 정렬

insertionSort(A[], n) ▷ $A[1 \cdots n]$을 정렬한다.
{
  ❶ for $i \leftarrow 2$ to n
    ❷ $A[1 \cdots i]$의 적합한 자리에 $A[i]$를 삽입한다;
}

```python
arr = [20, 10, 30, 50, 40]

for i in range(1, len(arr)):
    key = arr[i]

    j = i - 1
    while (j >= 0 and arr[j] > key):
        arr[j + 1] = arr[j]
        j = j - 1

    arr[j + 1] = key
    print(arr)
```

삽입 정렬

◈ 삽입 정렬 점근적 복잡도 분석

알고리즘 4-3	삽입 정렬

```
insertionSort(A[], n)        ▷ A[1 … n]을 정렬한다.
{
  ❶ for i ← 2 to n
      ❷ A[1 … i]의 적합한 자리에 A[i]를 삽입한다;
}
```

- 수행 시간:
  - ①의 for 루프는 $n - 1$번 반복
  - ②의 삽입은 최악의 경우 $i - 1$회 비교
- Worst Case
  - $T(n) = 1 + 2 + \cdots + (n-2) + (n-1) = \boldsymbol{\Theta(n^2)}$
- Best Case: 이미 정렬이 되어 있는 배열인 경우
  - $T(n) = \underbrace{1 + 1 + \cdots + 1 + 1}_{n-1} = \boldsymbol{\Theta(n)}$

선택 정렬 vs. 버블 정렬 vs. 삽입 정렬

◈ 선택 정렬 vs. 버블 정렬 vs. 삽입 정렬

- **선택 정렬**과 **버블 정렬**
  - 아직 정렬되지 않은 배열의 크기를 하나씩 줄임
  - Every Case: $T(n) = \boldsymbol{\Theta(n^2)}$
- **변형된 버블 정렬**
  - Worst Case: $T(n) = \boldsymbol{\Theta(n^2)}$
  - Best Case: $T(n) = \boldsymbol{\Theta(n)}$
- **삽입 정렬**
  - 이미 정렬된 배열의 크기를 하나씩 늘림
  - Worst Case: $T(n) = \boldsymbol{\Theta(n^2)}$
  - Best Case: $T(n) = \boldsymbol{\Theta(n)}$
- [참고] https://evan-moon.github.io/2018/10/13/sort-algorithm/

02. 고급 정렬 알고리즘

병합 정렬 Merge Sort

◈ 병합 정렬 의사 코드

알고리즘 4-4 | **병합 정렬**

```
mergeSort(A[], p, r)        ▷ A[p … r]을 정렬한다.
{
    if (p < r) then {
      ❶ q ← ⌊(p+r)/2⌋;           ▷ p, r의 중간 지점 계산
      ❷ mergeSort(A, p, q);      ▷ 전반부 정렬
      ❸ mergeSort(A, q+1, r);    ▷ 후반부 정렬
      ❹ merge(A, p, q, r);       ▷ 병합
    }
}

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

- **전략**
  - [분할] 배열을 반으로 나누어서 2 개의 부분배열로 분할한다. 각 부분 배열의 크기가 1이 될 때까지 계속하여 분할한다.
  - [정복(병합)] 인접한 부분 배열 2개(정렬된 부분 배열)를 병합하여 정렬된 1개의 배열을 얻어낸다.

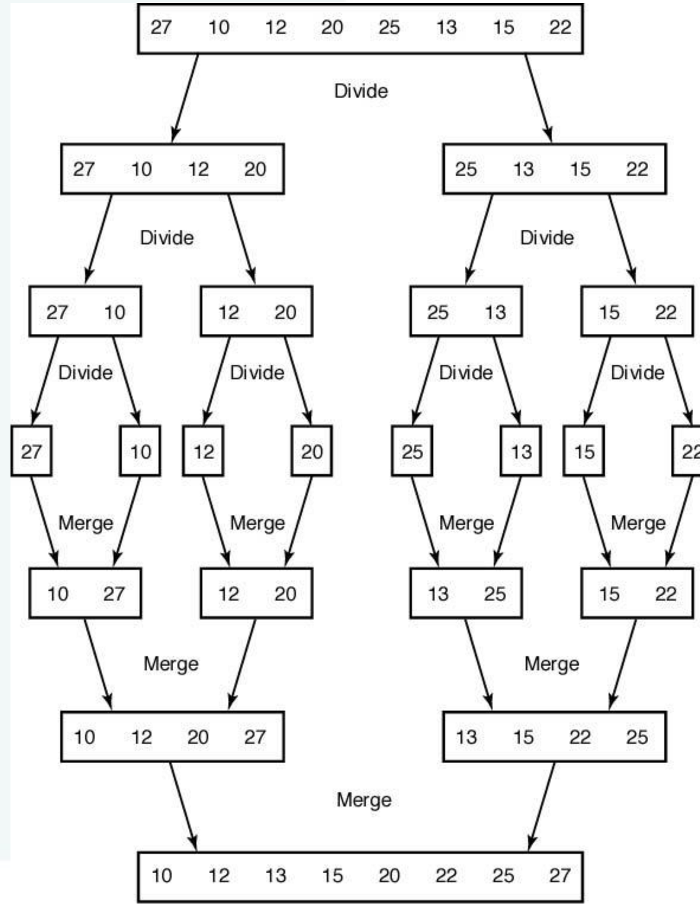
병합 정렬

◈ 병합 정렬 의사 코드

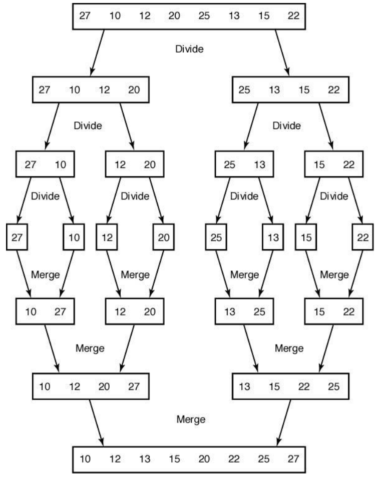
알고리즘 4-4 | **병합 정렬**

```
mergeSort(A[], p, r)        ▷ A[p … r]을 정렬한다.
{
    if (p < r) then {
        ❶ q ← ⌊(p+r)/2⌋;            ▷ p, r의 중간 지점 계산
        ❷ mergeSort(A, p, q);        ▷ 전반부 정렬
        ❸ mergeSort(A, q+1, r);      ▷ 후반부 정렬
        ❹ merge(A, p, q, r);         ▷ 병합
    }
}

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

그림 4-4 병합 정렬의 작동 과정을 보여주는 예



병합 정렬



◈ 병합 정렬 호출 예

mergeSort(A, 1, 8)

mergeSort(A, 1, 4) mergeSort(A, 5, 8)

mergeSort(A, 1, 2) mergeSort(A, 3, 4) mergeSort(A, 5, 6) mergeSort(A, 7, 8)

mergeSort(A, 1, 1) mergeSort(A, 3, 3) mergeSort(A, 5, 5) mergeSort(A, 7, 7)

mergeSort(A, 2, 2) mergeSort(A, 4, 4) mergeSort(A, 6, 6) mergeSort(A, 8, 8)

merge(A, 1, 1, 2) merge(A, 3, 3, 4) merge(A, 5, 5, 6) merge(A, 7, 7, 8)

merge(A, 1, 2, 4) merge(A, 5, 6, 8)

merge(A, 1, 4, 8)

병합 정렬

◈ 병합 정렬 의사 코드

```
void merge(A[], p, q, r)
▷ A[p…q]와 A[q+1…r]을 병합후
  A[p…r]을 정렬된 상태로 구성
▷ A[p…q]와 A[q+1…r]은 이미
  정렬되어 있음
{
   i←p; j←q+1; t←1;
   while (i≤q and j≤r) {
      if (A[i]≤A[j])
      then tmp[t++] ← A[i++];
      else tmp[t++] ← A[j++];
   }
   while (i≤q)
      tmp[t++] ← A[i++];
   while (j≤r)
      tmp[t++] ← A[j++];
   i←p;t←1;
   while (i≤r)
      A[i++] ← tmp[t++];
}
```

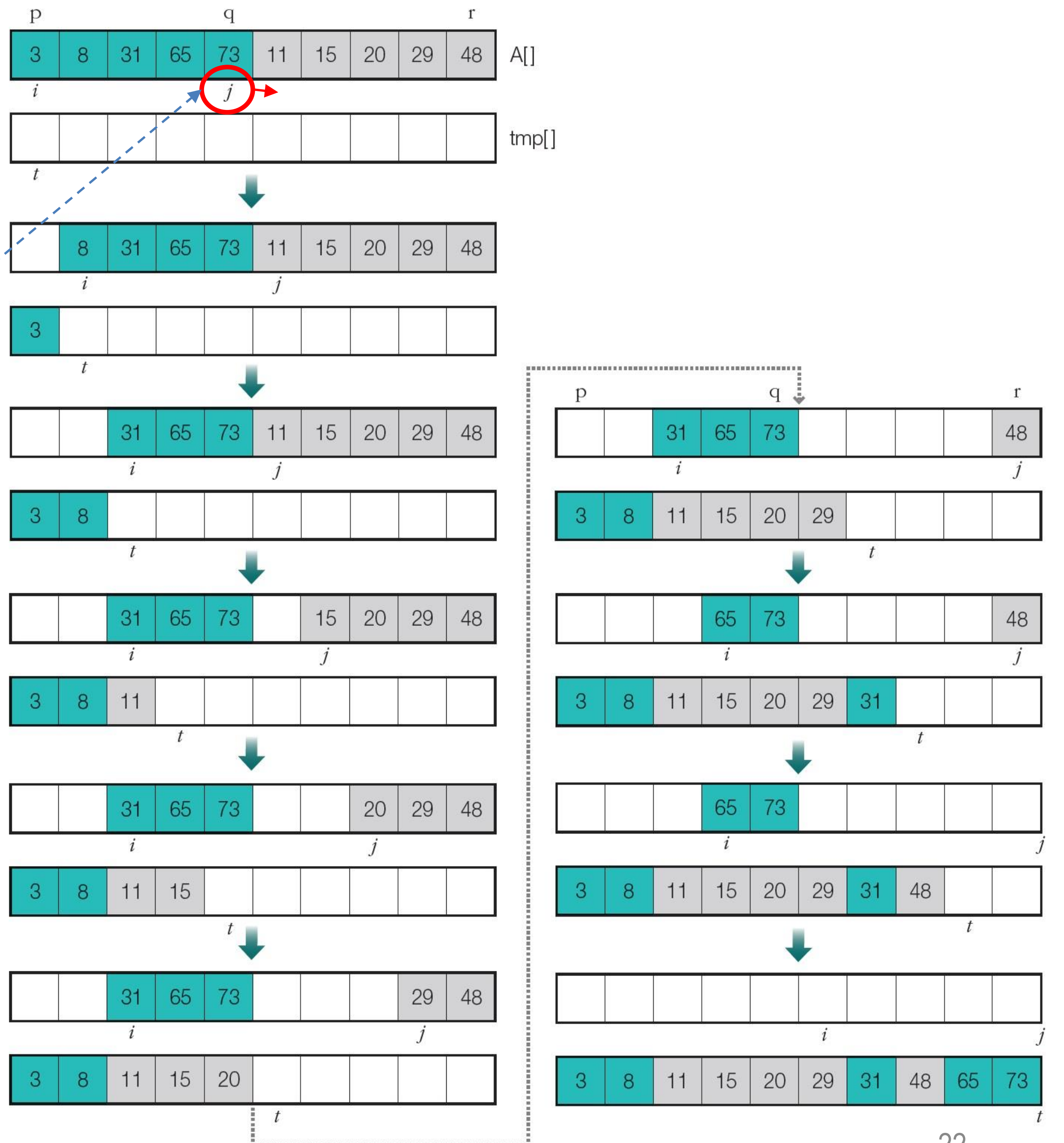


그림 4-5 병합 과정을 보여주는 예

병합 정렬

◈ 병합 정렬 복잡도 분석 (Case I)

– 1) merge() 함수 복잡도
  - 단위연산:
    - A[i]≤A[j]
  - 최악의 경우 ① 루프를 통한 단위연산 수행 횟수

$$(q-p+1)+(r-q-1+1)=r-p+1$$

- i == q and j == r 인 상태에서 i나 j 중 하나가 q 또는 r을 벗어나서 해당 루프가 종료

  - 따라서, 최악의 경우 복잡도

$$\widehat{\boldsymbol{T}}(\boldsymbol{p},\boldsymbol{q},\boldsymbol{r})=\boldsymbol{r}-\boldsymbol{p}+\mathbf{1}$$

```
void merge(A[], p, q, r)
{
      i←p; j←q+1; t←1;
 ① while (i≤q and j≤r) {
        if (A[i]≤A[j])
            then tmp[t++] ← A[i++];
            else tmp[t++] ← A[j++];
    }
 ② while (i≤q)
        tmp[t++] ← A[i++];
 ③ while (j≤r)
        tmp[t++] ← A[j++];
      i←p;t←1;
 ④ while (i≤r)
        A[i++] ← tmp[t++];
}
```

병합 정렬

◈ 병합 정렬 복잡도 분석 (Case I)

– 2) mergeSort() 함수 복잡도

- 단위연산: mergeSort() 함수에 존재하는 merge() 함수 내부에서의 단위 연산 (`A[i]≤A[j]`)

- 최초 입력 배열의 개수 n에 대해 n=2^{k}라고 가정해도 일반성을 잃지 않는다. 이 때,
  - ➢ $r - p + 1 = n$
  - ➢ $q - p + 1 = \frac{n}{2}$
  - ➢ $r - q = \frac{n}{2}$

```
mergeSort(A[], p, r)        ▷ A[p … r]을 정렬한다.
{
    if (p < r) then {
        ❶ q ← ⌊(p+r)/2⌋;             ▷ p, r의 중간 지점 계산
        ❷ mergeSort(A, p, q);        ▷ 전반부 정렬
        ❸ mergeSort(A, q+1, r);      ▷ 후반부 정렬
        ❹ merge(A, p, q, r);         ▷ 병합
    }
}
```

- **최악의 경우** 복잡도

$$T(n) = T(q-p+1) + T(r-q) + \hat{T}(p,q,r)$$
$$= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + r - p + 1 = 2T\left(\frac{n}{2}\right) + n$$

병합 정렬

◈ 병합 정렬 복잡도 분석 (Case I)

– 3) 마스터 정리 이용

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{for } a \geq 1 \text{ and } b > 1}$$

- $a = 2$, $b = 2$, $f(n) = n$

- [마스터 정리의 근사 버전]에 의해
  - ➢ $h(n) = n^{\log_b a} = n^{\log_2 2} = n$
  - ➢ $\frac{f(n)}{h(n)} = \Theta(1)$이므로, $T(n) = \Theta(h(n)logn) = \Theta(nlogn)$이다.

따라서, 이런 경우 최악의 경우를 고려하여 최종적인 병합 정렬의 점근적 복잡도 분석 결과는 $O(nlogn)$ 라고 정하는 것이 적합

병합 정렬

◈ 병합 정렬 복잡도 분석 (Case II)

– 1) merge() 함수 복잡도

- 단위연산:
  - ➢ tmp[t++] ← A[...]
    또는 A[i++] ←tmp[t++]
- 모든 경우 ①, ②, ③ 루프를 통한 단위연산 수행 횟수

$$r - p + 1$$

- 모든 경우 ④ 루프를 통한 단위연산 수행 횟수

$$r - p + 1$$

- 따라서, 모든 경우 복잡도

$$\hat{\boldsymbol{T}}(\boldsymbol{p},\boldsymbol{q},\boldsymbol{r}) = \boldsymbol{r} - \boldsymbol{p} + \mathbf{1}$$

```
void merge(A[], p, q, r)
{
      i←p; j←q+1; t←1;
 ① while (i≤q and j≤r) {
        if (A[i]≤A[j])
           then tmp[t++] ← A[i++];
           else tmp[t++] ← A[j++];
     }
 ② while (i≤q)
        tmp[t++] ← A[i++];
 ③ while (j≤r)
        tmp[t++] ← A[j++];
     i←p;t←1;
 ④ while (i≤r)
        A[i++] ← tmp[t++];
}
```

병합 정렬

◈ 병합 정렬 복잡도 분석 (Case II)

– 2) mergeSort() 함수 복잡도

- 단위연산: mergeSort() 함수에 존재하는 merge() 함수 내부에서의 단위 연산 (`tmp[t++] ← A[…]` 또는 `A[i++] ←tmp[t++]`)

```
mergeSort(A[], p, r)        ▷ A[p … r]을 정렬한다.
{
    if (p < r) then {
        ❶ q ← ⌊(p+r)/2⌋;           ▷ p, r의 중간 지점 계산
        ❷ mergeSort(A, p, q);       ▷ 전반부 정렬
        ❸ mergeSort(A, q+1, r);     ▷ 후반부 정렬
        ❹ merge(A, p, q, r);        ▷ 병합
    }
}
```

- 최초 입력 배열의 개수 n에 대해 $n=2^k$라고 가정해도 일반성을 잃지 않는다. 이 때,
  - ➢ $r - p + 1 = n$
  - ➢ $q - p + 1 = \frac{n}{2}$
  - ➢ $r - q = \frac{n}{2}$

- 모든 경우 복잡도

$$T(n) = T(q-p+1) + T(r-q) + \hat{T}(p,q,r)$$
$$= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + r - p + 1 = 2T\left(\frac{n}{2}\right) + n$$

병합 정렬

◈ 병합 정렬 복잡도 분석 (Case II)

– 3) 마스터 정리 이용

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{for } a \geq 1 \text{ and } b > 1$$

- $a = 2$, $b = 2$, $f(n) = n$

- [마스터 정리의 근사 버전]에 의해
  - $h(n) = n^{\log_b a} = n^{\log_2 2} = n$
  - $\frac{f(n)}{h(n)} = \Theta(1)$이므로, $T(n) = \Theta(h(n)logn) = \Theta(nlogn)$이다.

따라서, 이런 경우 모든 경우를 고려하여 최종적인 병합 정렬의 점근적 복잡도 분석 결과는 모든 경우 $\Theta(nlogn)$ 라고 정하는 것이 적합

퀵 정렬 Quick Sort

◈ 퀵 정렬 (Quick Sort)

- 1962년에 영국의 호아(C.A.R. Hoare)의 의해서 고안
- 퀵 정렬이란 이름이 오해의 여지가 있음
  - 사실 가장 빠른 정렬 알고리즘이라고 할 수는 없다.
- 주어진 배열을 두 개로 분할하고, 각각을 정렬한다.
  - 합병정렬과 동일?
- 합병정렬과 다른점
  - 합병정렬은 아무생각없이 두 부분으로 나누는 반면에, 퀵정렬은 분할할 때부터 <u>**기준 원소(pivot)**</u> 중심으로, 이보다 작은 것은 왼편, 큰 것은 오른편에 위치시킨다.
    - ➢ 즉, 정렬(정복과정)이 분할 과정에서 함께 행해진다.
  - 각 부분 정렬이 끝난 후, 합병정렬은 "합병"이라는 후처리 작업이 필요하나, 빠른정렬은 필요로 하지 않는다.

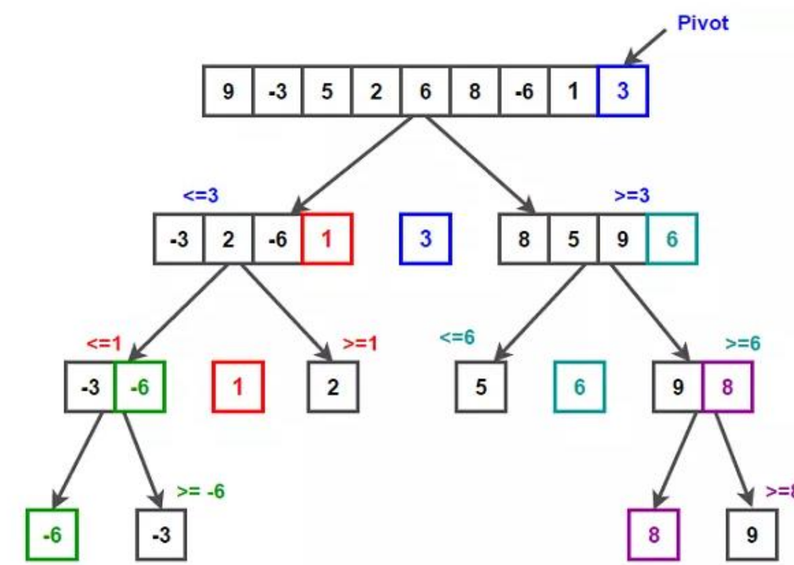
퀵 정렬

◈ 퀵 정렬 의사 코드

알고리즘 4-5 퀵 정렬

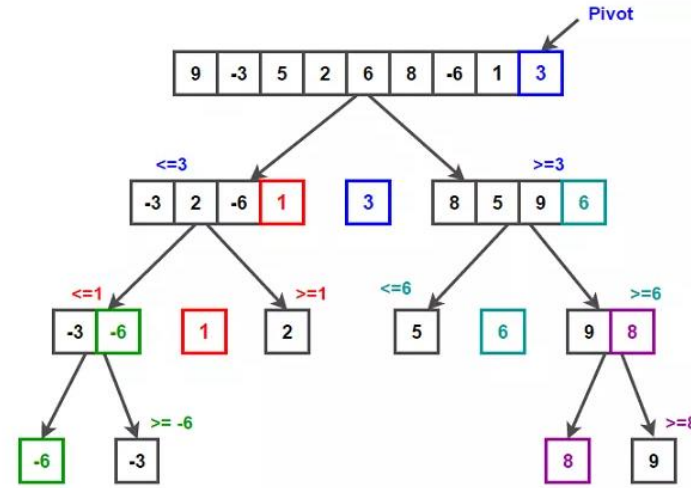
```
quickSort(A[], p, r)          ▷ A[p … r]을 정렬한다.
{
   ❶ if (p < r) then {
        ❷ q ← partition(A, p, r);      ▷ 분할
        ❸ quickSort(A, p, q−1);        ▷ 왼쪽 부분 배열 정렬
        ❹ quickSort(A, q+1, r);        ▷ 오른쪽 부분 배열 정렬
     }
}

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

그림 4-6 퀵 정렬의 작동 과정을 보여주는 예

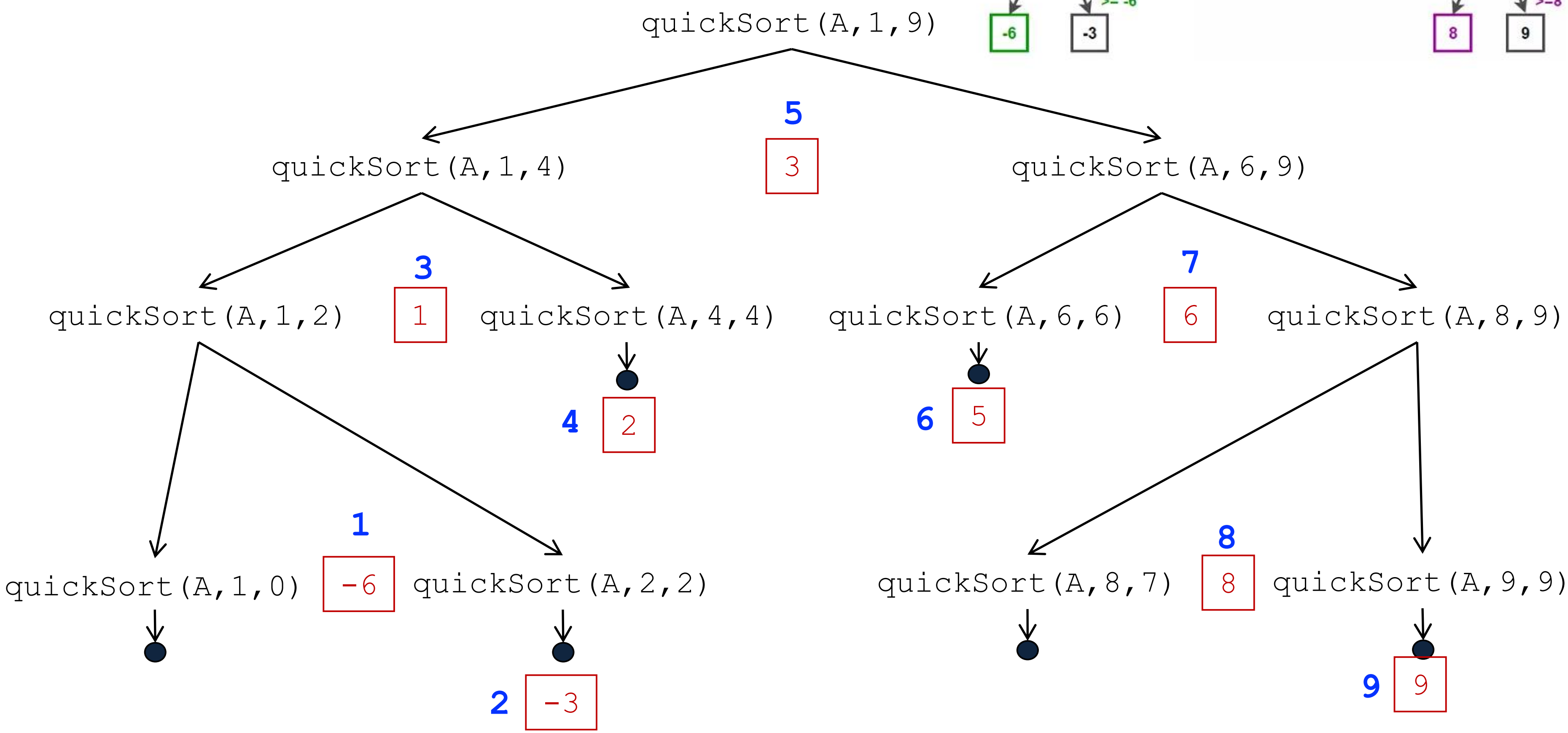


퀵 정렬

◈ 퀵 정렬 호출 예



Pivot
9 -3 5 2 6 8 -6 1 3
<=3
-3 2 -6 1
3
>=3
8 5 9 6
<=1
-3 -6
1
>=1
2
<=6
5
6
>=6
9 8
-6
>= -6
-3
8
>=8
9



quickSort(A,1,9)
5
3
quickSort(A,1,4)
quickSort(A,6,9)
3
1
quickSort(A,1,2)
quickSort(A,4,4)
4
2
7
6
quickSort(A,6,6)
quickSort(A,8,9)
6
5
1
-6
quickSort(A,1,0)
quickSort(A,2,2)
2
-3
8
8
quickSort(A,8,7)
quickSort(A,9,9)
9
9

퀵 정렬

◈ 퀵 정렬 의사 코드

알고리즘 4-6 분할

```
partition(A[], p, r)
{
    x ← A[r];                  ▷ 기준원소
    i ← p-1;                   ▷ i는 1구역의 끝지점
    for j ← p to r-1           ▷ j는 3구역의 시작 지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
                               ▷ 의미는 i값 증가 후 A[i] ↔ A[j] 교환
    A[i+1] ↔ A[r];             ▷ 기준원소와 2구역 첫 원소 교환
    return i+1;
}
```

1구역(□) : 15보다 작거나 같은 원소들
2구역(■) : 15보다 큰 원소들
3구역(□) : 아직 정해지지 않은 원소들
4구역(●) : 15 자신

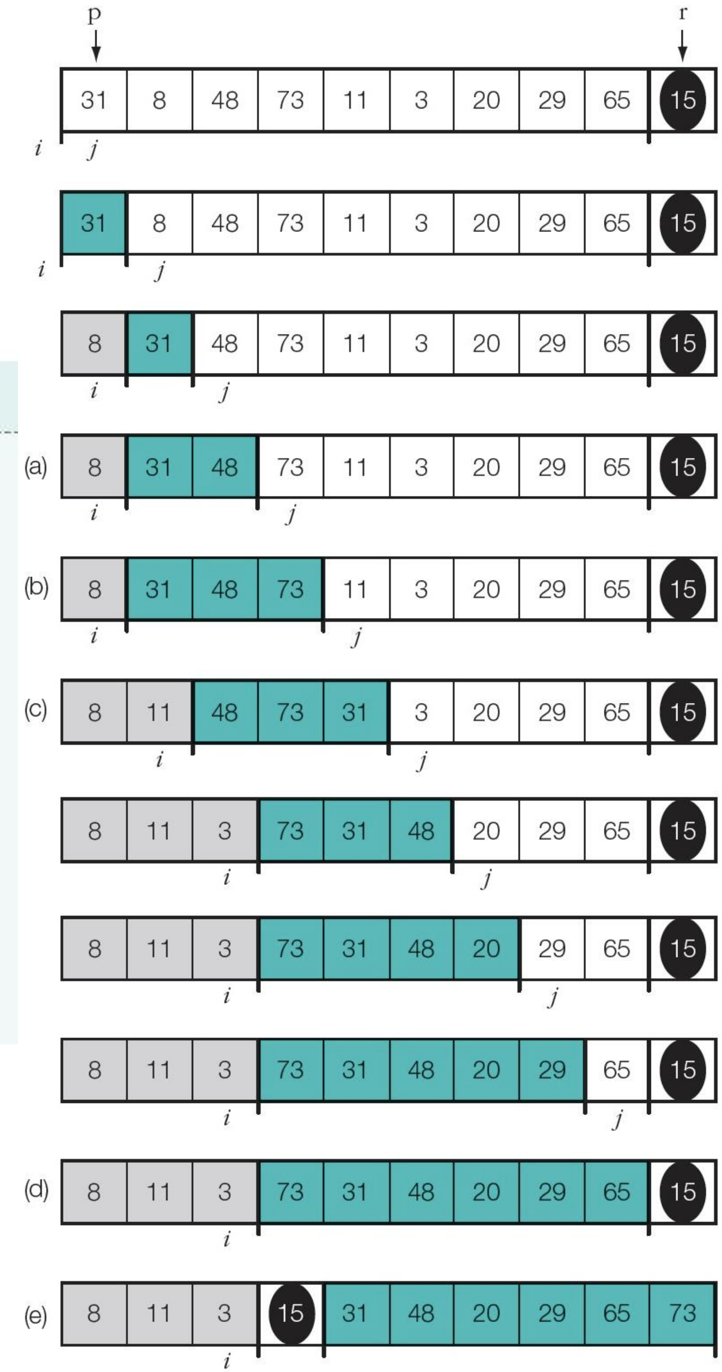


그림 4-7 분할의 작동 과정을 보여주는 예

퀵 정렬

◈ 퀵 정렬 복잡도 분석

– 1) partition() 함수 복잡도

- 단위연산: `A[j]≤x` 비교연산
- 모든 경우 j-루프의 반복횟수와 단위 연산 수행 횟수는 동일

$$p \le j \le r - 1$$

- 따라서, 모든 경우 복잡도

$$\hat{T}(p,r) = r - 1 - p + 1 = r - p$$

```
partition (A[], p, r)
{
    x ← A[r];                ▷ 기준원소
    i ← p−1;                 ▷ i는 1구역의 끝지점
    for j ← p to r−1         ▷ j는 3구역의 시작 지
        if (A[j] ≤ x) then A[++i] ↔ A[j];
                             ▷ 의미는 i값 증가 후
    A[i+1] ↔ A[r];           ▷ 기준원소와 2구역
    return i+1;
}
```

$r - p + 1 = n$ 라고 가정할 때
독립적인 partition() 함수의 점근적 복잡도는 $\Theta(n)$

퀵 정렬

```
quickSort(A[], p, r)        ▷ A[p … r]을 정렬한다.
{
  ❶ if (p < r) then {
      ❷ q ← partition(A, p, r);     ▷ 분할
      ❸ quickSort(A, p, q−1);       ▷ 왼쪽 부분 배열 정렬
      ❹ quickSort(A, q+1, r);       ▷ 오른쪽 부분 배열 정렬
    }
}
```

◈ 퀵 정렬 복잡도 분석

- 2) quickSort(A,) 함수 복잡도
  - 단위연산: quickSort(A,) 함수에 존재하는 partition() 함수 내부에서의 `A[j]≤x` 비교연산
  - 최초 입력 배열의 개수 n에 대해 $n=2^k$라고 가정해도 일반성을 잃지 않는다. 이 때,
    - $r - p + 1 = n$
    - $q - 1 - p + 1 = \frac{n}{2}$
    - $r - q - 1 + 1 = \frac{n}{2}$
  - 이상적인 경우 (분할이 항상 반반씩 균등하게 된다고 가정하면) 병합 정렬과 동일 상황

$$\left.\begin{aligned} T(n) &= T(q-1-p+1) + T(r-q-1+1) + \hat{T}(p,q,r) \\ &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + r - p + 1 = 2T\left(\frac{n}{2}\right) + n \end{aligned}\right\} \quad T(n) = \Theta(nlogn)$$

퀵 정렬

```
quickSort(A[], p, r)        ▷ A[p … r]을 정렬한다.
{
  ❶ if (p < r) then {
       ❷ q ← partition(A, p, r);     ▷ 분할
       ❸ quickSort(A, p, q-1);       ▷ 왼쪽 부분 배열 정렬
       ❹ quickSort(A, q+1, r);       ▷ 오른쪽 부분 배열 정렬
     }
}
```

◈ 퀵 정렬 복잡도 분석

– 2) quickSort(A,) 함수 복잡도

- **최악의 경우**: 분할이후 한쪽에는 원소가 하나도 없고, 다른 한쪽에 모든 원소가 몰리는 경우

$$T(n) = T(n-1) + n - 1$$

$$\left.\begin{array}{l} T(n) - T(n-1) = n - 1 \\ T(n-1) - T(n-2) = n - 2 \\ T(n-2) - T(n-3) = n - 3 \\ \qquad\qquad\cdots \\ T(3) - T(2) = 2 \\ T(2) - T(1) = 1 \\ \hline T(n) - T(1) = 1 + 2 + \cdots + (n-2) + (n-1) \\ \Rightarrow \\ T(n) = n(n-1)/2 \quad (\because T(1) = 0) \end{array}\right\} \; T(n) = \Theta(n^2)$$

퀵 정렬

```
quickSort(A[], p, r)        ▷ A[p … r]을 정렬한다.
{
  ❶ if (p < r) then {
        ❷ q ← partition(A, p, r);     ▷ 분할
        ❸ quickSort(A, p, q−1);       ▷ 왼쪽 부분 배열 정렬
        ❹ quickSort(A, q+1, r);       ▷ 오른쪽 부분 배열 정렬
     }
}
```

◈ 퀵 정렬 복잡도 분석

– 2) quickSort(A,) 함수 복잡도

- **평균의 경우**:
  - ➢ 기준 원소가 첫 번째로 크면 두 분할의 크기 → $0\!: n-1$
  - ➢ 기준 원소가 두 번째로 크면 두 분할의 크기 → $1\!: n-2$
  - ➢ 기준 원소가 세 번째로 크면 두 분할의 크기 → $2\!: n-3$
  - ➢ …
  - ➢ 기준 원소가 가장 작으면(n 번째로 크면) 두 분할의 크기 → $n-1\!: 0$

- 즉, 기준 원소가 i 번째로 클 때 두 분할의 크기를 고려한 복잡도 식

$$T(n) = T(i-1) + T(n-i) + n - 1$$

- 모든 가능한 경우에 대한 평균 식

$$\boxed{T(n) = \frac{1}{n}\sum_{i=1}^{n} T(i-1) + T(n-i) + n - 1} \quad \Big\} \quad T(n) = \Theta(nlogn)$$

수학적 증명생략

퀵 정렬

◈ 퀵 정렬 올바름 증명

– [정리 4-1, 페이지 109]

[알고리즘 4-5]의 퀵 정렬은 제대로 정렬을 한다.

- [증명 생략]
  - 고등학교때 배운 수학적 귀납법 사용

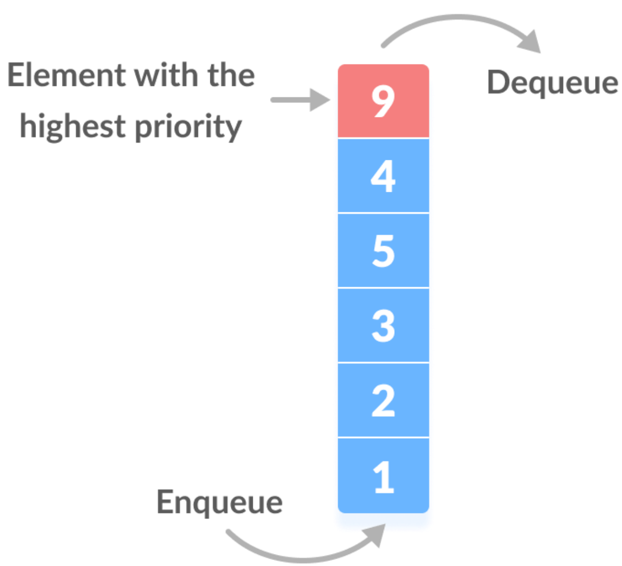
힙 정렬 Heap Sort

◈ What is Priority Queue (우선순위 큐)?

- 응급실에서 환자 치료의 예
- 큐: 먼저 온 사람을 먼저 치료
- 스택: 나중에 온 사람을 먼저 치료
- 우선순위 큐 (Priority Queue): 우선순위가 높은 사람 (예: 위급한 사람)을 먼저 치료

◈ Priority Queue (우선순위 큐)

- 우선 순위 큐는
  각 노드에 "우선순위 값" 필드 필요



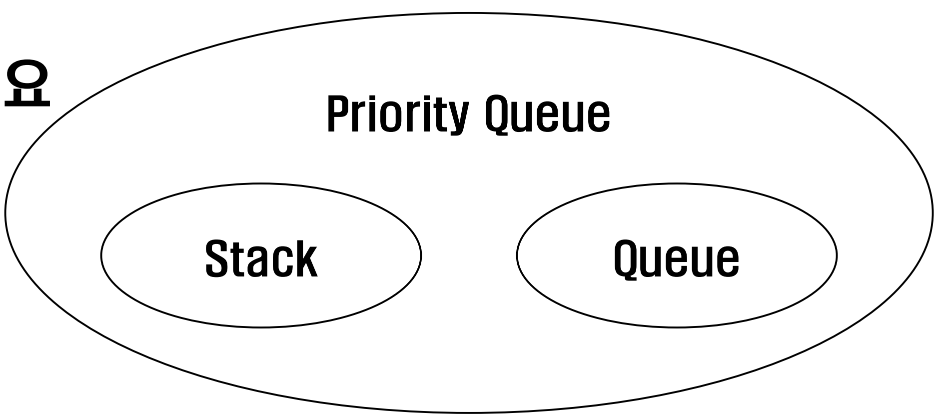
힙 정렬 Heap Sort

◈ Priority Queue vs. Stack/Queue

- 스택과 큐
  - 스택과 큐는 원소가 추가되는 시간에 따라 자료구조를 조직화
- 우선순위 큐
  - 시간을 포함한 여러 가지 가치를 우선순위로 가지는 자료구조

◈ 우선순위 큐는 스택이나 큐 보다 일반적인 구조

- 큐나 스택은 우선순위 큐의 특수한 형태로서 시간에 그 우선순위를 부여한 것임
- 따라서 우선순위 필드가 불필요



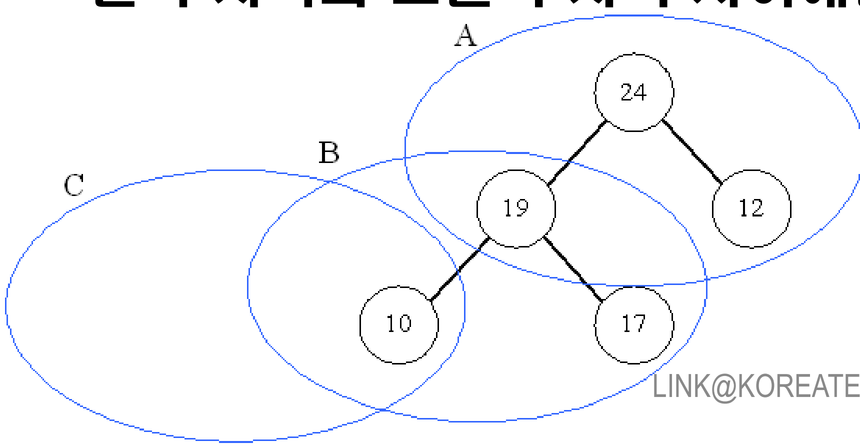
힙 정렬 Heap Sort

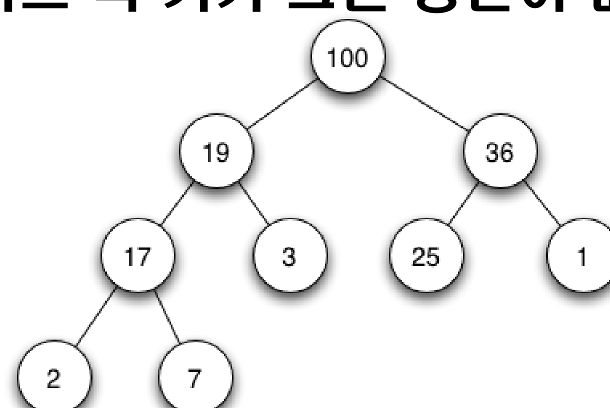
◈ Heap

- a binary tree structure with the following properties
  - The tree is complete or nearly complete
- 부모의 키가 왼쪽 자식 및 오른쪽 자식의 키보다 크거나 같다.
  - 키: Priority
- The subtrees are in turn heaps
- 지원되는 연산: Insertion, Deletion, (optionally)Peek
- Heap is <u>an excellent structure to implement priority queue</u>

◈ 주의할 특징

- 왼쪽 자식과 오른쪽 자식 사이에는 어느 쪽 키가 크던 상관이 없다.



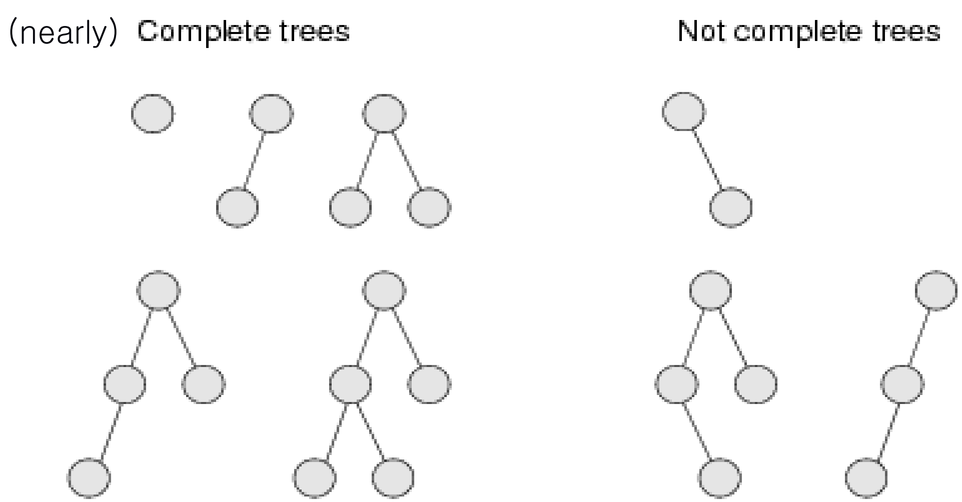


힙 정렬 Heap Sort

◈ Heap의 속성

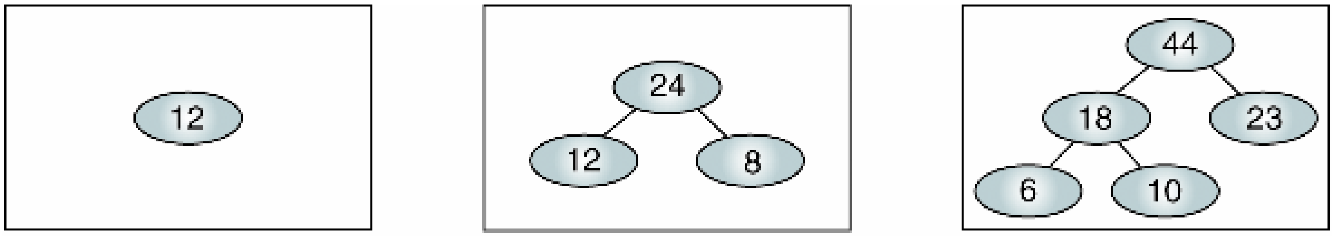
A heap is a special kind of tree. It has two properties:

[Completeness] – The tree is (nearly) complete, which means that nodes are added from top to bottom, left to right, without leaving any spaces.
[Heapness] – The item in the tree with the highest priority is at the top of the tree, and the same is true for every subtree.



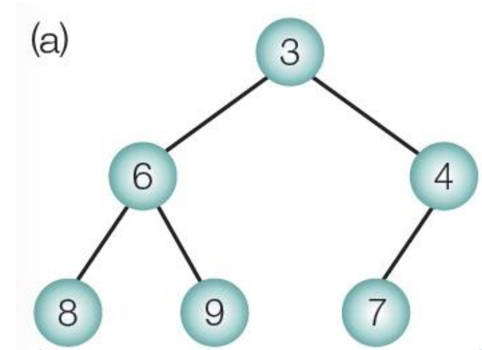
힙 정렬 Heap Sort

◈ Examples of Heaps



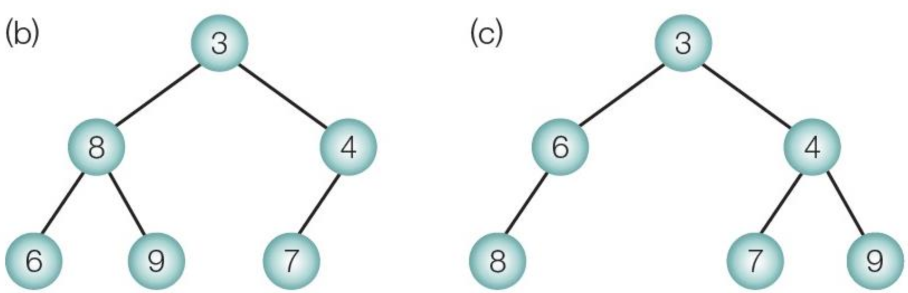
(a) Root-only heap (b) Two-level heap (c) Three-level heap

max heap

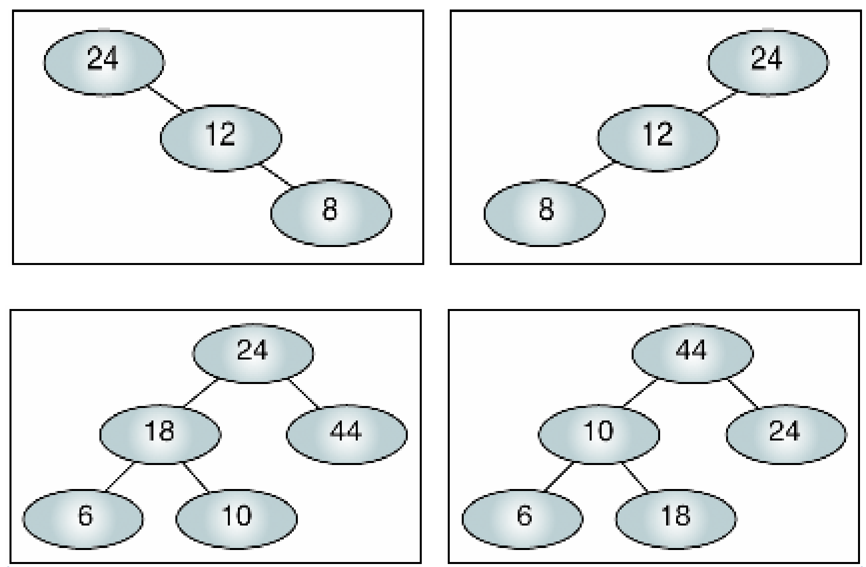


min heap

◈ Invalid heaps (Why?)



min heap

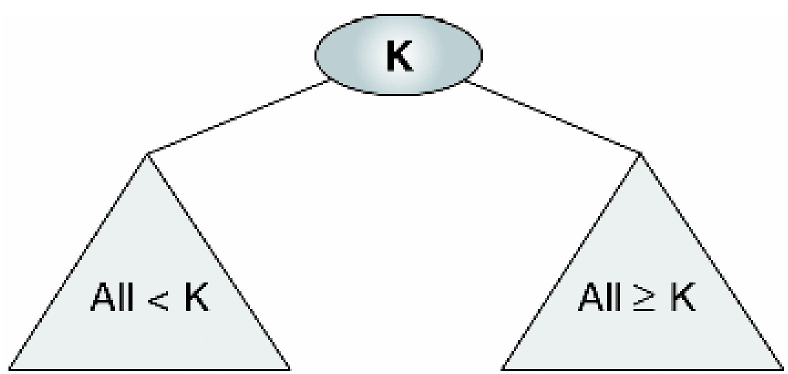


max heap

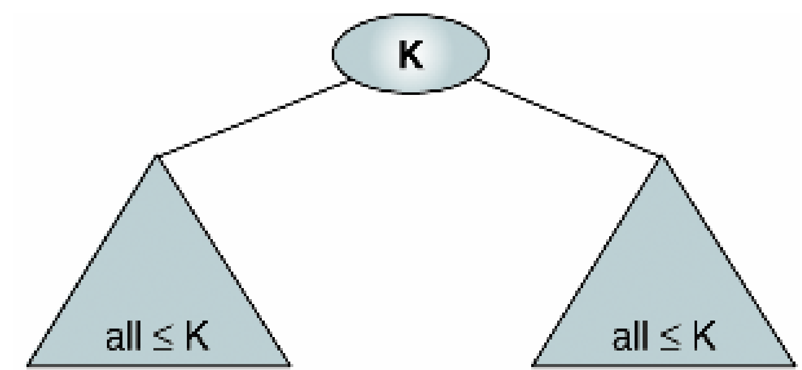
힙 정렬 Heap Sort

◈ 정렬 관점에서…

- BST (Binary Search Tree)는 약한 의미로 정렬
  - 왼쪽 자식 키보다 오른쪽 자식 키가 크다
- Heap도 약한 의미로 정렬
  - 부모의 키가 자식의 키보다 우선 순위가 높다
  - 왼쪽 자식 키와 오른쪽 자식 키는 무관하다



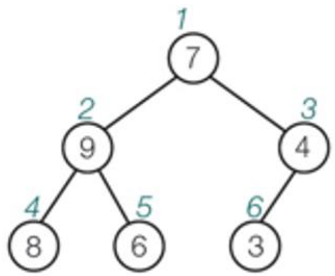
BST (Binary Search Tree)



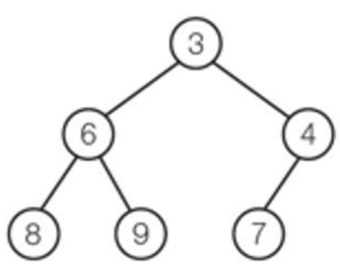
(max) Heap

힙 구성

◈ Heap 만들기 의사 코드



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



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

알고리즘 4-7 | 힙 만들기

Min Heap 기준

```
buildHeap(A[], n)        ▷ A[1 … n]을 힙으로 만든다.
{
  ❶ for i ← ⌊n/2⌋ downto 1
        heapify(A, i, n);
}

heapify(A[], k, n)        ▷ A[k]를 루트로 하는 트리를 힙성질을 만족하도록 수선한다.
▷ A[k]의 두 자식을 루트로 하는 서브 트리는 힙성질을 만족하고 있다.
▷ n은 최대 인덱스(전체 배열의 크기)
{
    left ← 2k; right ← 2k+1;
    ▷ 작은 자식 고르기. smaller : A[2k]와 A[2k+1] 중에 작은 원소
    if (right ≤ n) then {                    ▷ k가 두 자식을 가지는 경우
        if (A[left] < A[right]) then smaller ← left;
                                 else smaller ← right;
    }
    else if (left ≤ n) then smaller ← left;  ▷ k의 왼쪽 자식만 있는 경우
    else return;                             ▷ A[k]가 리프 노드임. 끝남.
    ▷ 재귀적 조정
    if (A[smaller] < A[k]) then {
        A[k] ↔ A[smaller];
        heapify(A, smaller, n);
    }
}
```

더 큰 값을 자식 노드로 내려보낸 이후 해당 자식 노드를 루트로 취급하여 다시 heapify 재귀 호출

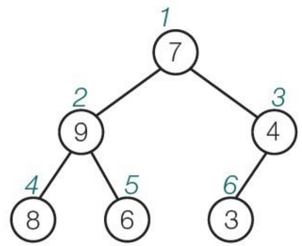
힙 구성

◈ Heap 만들기 수행 예

알고리즘 4-7 | 힙 만들기

```
buildHeap(A[], n)        ▷ A[1 … n]을 힙으로 만든다.
{
  ❶ for i ← ⌊n/2⌋ downto 1
        heapify(A, i, n);
}

heapify(A[], k, n)       ▷ A[k]를 루트로 하는 트리를 힙성질을 만족하도록 수선한다.
▷ A[k]의 두 자식을 루트로 하는 서브 트리는 힙성질을 만족하고 있다.
▷ n은 최대 인덱스(전체 배열의 크기)
{
    left ← 2k; right ← 2k+1;
    ▷ 작은 자식 고르기. smaller : A[2k]와 A[2k+1] 중에 작은 원소
    if (right ≤ n) then {                      ▷ k가 두 자식을 가지는 경우
        if (A[left] < A[right]) then smaller ← left;
                                else smaller ← right;
    }
    else if (left ≤ n) then smaller ← left;    ▷ k의 왼쪽 자식만 있는 경우
    else return;                               ▷ A[k]가 리프 노드임. 끝남.
    ▷ 재귀적 조정
    if (A[smaller] < A[k]) then {
        A[k] ↔ A[smaller];
        heapify(A, smaller, n);
    }
}
```

(a)



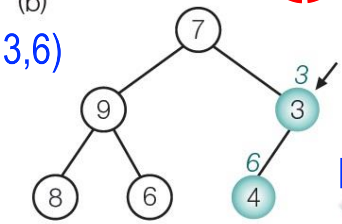
	1	2	3	4	5	6
A	7	9	4	8	6	3

heapify(A,3,6)

left←6, right ←7
smaller ← 6, A[3]⟷A[6]
heapify(A,6,6): 호출 후 끝

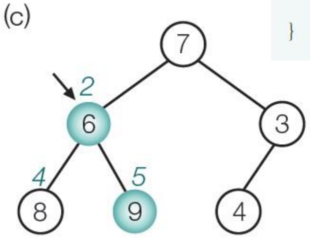
	1	2	3	4	5	6
A	7	9	3	8	6	4

(b)



heapify(A,2,6)

(c)



left←4, right ←5, smaller ← 5, A[2]⟷A[5]
heapify(A,5,6): 호출 후 끝

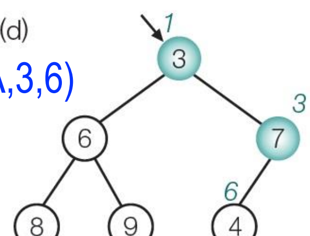
	1	2	3	4	5	6
A	7	6	3	8	9	4

heapify(A,1,6)

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

left←2, right ←3, smaller ← 3, A[1]⟷A[3]
heapify(A,3,6): 호출 후 내부 수행!!!

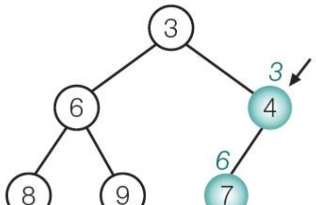
(d)



heapify(A,3,6)

left←6, right ←7, smaller ← 6, A[3]⟷A[6]
heapify(A,6,6): 호출 후 끝

(e)



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

그림 4-10 임의의 배열을 힙으로 수선하는 예

힙 구성

◈ Heap 구성의 점근적 복잡도 분석

알고리즘 4-7 | 힙 만들기

```
buildHeap(A[], n)        ▷ A[1 ··· n]을 힙으로 만든다.
{
  ❶ for i ← ⌊n/2⌋ downto 1
        heapify(A, i, n);
}

heapify(A[], k, n)       ▷ A[k]를 루트로 하는 트리를 힙성질을 만족하도록 수선한다.
▷ A[k]의 두 자식을 루트로 하는 서브 트리는 힙성질을 만족하고 있다.
▷ n은 최대 인덱스(전체 배열의 크기)
{
    left ← 2k; right ← 2k+1;
    ▷ 작은 자식 고르기. smaller : A[2k]와 A[2k+1] 중에 작은 원소
    if (right ≤ n) then {                   ▷ k가 두 자식을 가지는 경우
        if (A[left] < A[right]) then smaller ← left;
                                 else smaller ← right;
    }
    else if (left ≤ n) then smaller ← left;  ▷ k의 왼쪽 자식만 있는 경우
    else return;                             ▷ A[k]가 리프 노드임. 끝남.
    ▷ 재귀적 조정
    if (A[smaller] < A[k]) then {
        A[k] ↔ A[smaller];
        heapify(A, smaller, n);
    }
}
```

buildHeap()에서 heapify()를 호출하는 횟수는 $\left\lfloor \frac{n}{2} \right\rfloor$

길이가 n인 트리의 높이는 아무리 길어도 $\log_2 n$이므로 heapify의 점근적 복잡도는 $O(logn)$

buildHeap()의 점근적 복잡도: $\boldsymbol{O(nlogn)}$

또는

$\Theta(n)$

증명 생략
[P.114 note 참조]

힙 정렬

◈ Heap 정렬 의사 코드

알고리즘 4-8 힙 정렬

```
heapSort(A, n)        ▷ A[1 ··· n]을 정렬한다.
{
    buildHeap(A, n);
  ❶ for i ← n downto 2 {
        A[1] ↔ A[i];        ▷ 원소 교환
        heapify(A, 1, i−1);
    }
}
```

	1	2	3	4	5	6	
(a)	3	6	4	8	9	7	
(b)	7	6	4	8	9	3	원소 교환
(c)	4	6	7	8	9	3	heapify
(d)	9	6	7	8	4	3	원소 교환
(e)	6	9	7	8	4	3	heapify
(f)	6	8	7	9	4	3	heapify
(g)	9	8	7	6	4	3	원소 교환
(h)	7	8	9	6	4	3	heapify
(i)	9	8	7	6	4	3	원소 교환
(j)	8	9	7	6	4	3	heapify
(k)	9	8	7	6	4	3	원소 교환

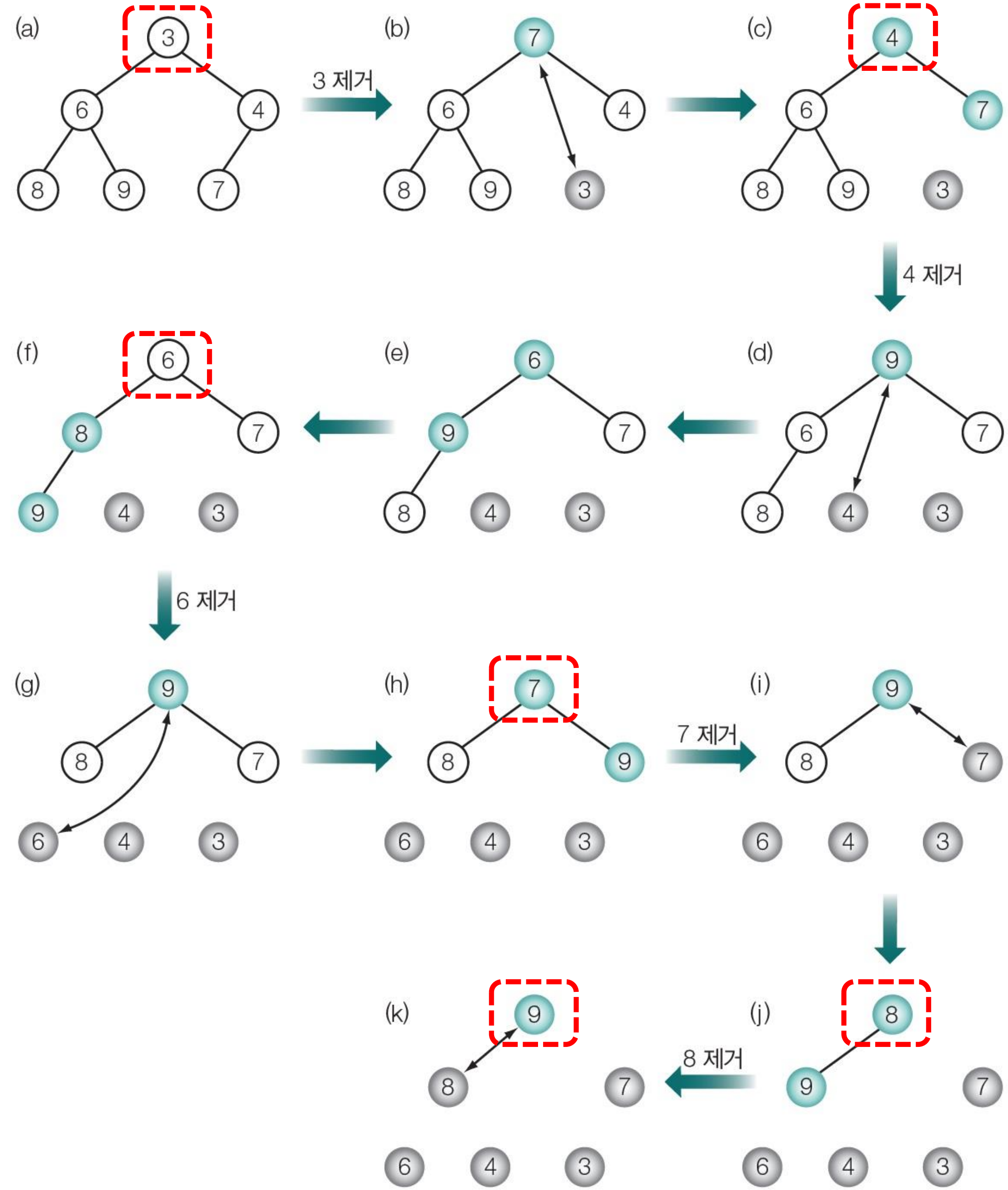


그림 4-11 힙 정렬의 작동 과정을 보여주는 예

힙 정렬

◈ Heap 정렬의 점근적 복잡도 분석

알고리즘 4-8 힙 정렬

```
heapSort(A, n)        ▷ A[1 ⋯ n]을 정렬한다.
{
    buildHeap(A, n);
  ❶ for i ← n downto 2 {
        A[1] ↔ A[i];       ▷ 원소 교환
        heapify(A, 1, i-1);
    }
}
```

buildHeap: $O(nlogn)$ (or $\Theta(n)$)

heapify: $O(logn)$

for loop: $O(nlogn)$

heapSort: $O(nlogn)$

03. 비교 정렬 시간의 하한

비교 정렬 시간의 하한

◈ 비교 정렬

- 정렬의 대상이 되는 각 원소들의 값을 비교하는 것이 핵심이 되는 정렬 알고리즘을 통칭

◈ 비교 정렬 시간의 하한

- 최악의 경우: 적어도 $\Theta(nlogn)$ → $\boldsymbol{\Omega(nlogn)}$
- 증명 생략

각각의 (비교) 정렬 복잡도 분석

◈ 정렬 복잡도 분석

		최선	평균	최악
기본	선택	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
	버블I	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
	버블II	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
	삽입	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
고급	병합	$\Theta(nlogn)$	$\Theta(nlogn)$	$\Theta(nlogn)$
	퀵	$\Theta(nlogn)$	$\Theta(nlogn)$	$\Theta(n^2)$
	힙	$\Theta(nlogn)$	$\Theta(nlogn)$	$\Theta(nlogn)$

04. 특수 정렬 알고리즘

특수 정렬 알고리즘

◈ 더 효율적인 정렬 알고리즘?

– "최악의 경우 정렬 시간이 $\Theta(nlogn)$보다 더 빠를 수 없을까?"

  - 만약 원소들이 특수한 성질을 만족할 때 해당 원소의 자릿수 등을 고려하며 정렬할 수 있음

◈ 특수 정렬 알고리즘 종류

– 기수 정렬 (Radix Sort)
– 계수 정렬 (Counting Sort)
– 자세한 설명은 생략

Questions & Answers