



9 정렬

IT CookBook, C로 배우는 쉬운 자료구조(개정 3판)

❖ 학습목표

- 자료의 정렬 방법을 알아본다.
- 정렬 방법에 대한 알고리즘을 이해한다.
- 정렬 알고리즘의 효율을 알아본다.

❖ 내용

- 정렬의 이해 | 선택 정렬
- 버블 정렬 | 퀵 정렬
- 삽입 정렬 | 셸 정렬
- 병합 정렬 | 기수 정렬
- 힙 정렬 | 트리 정렬

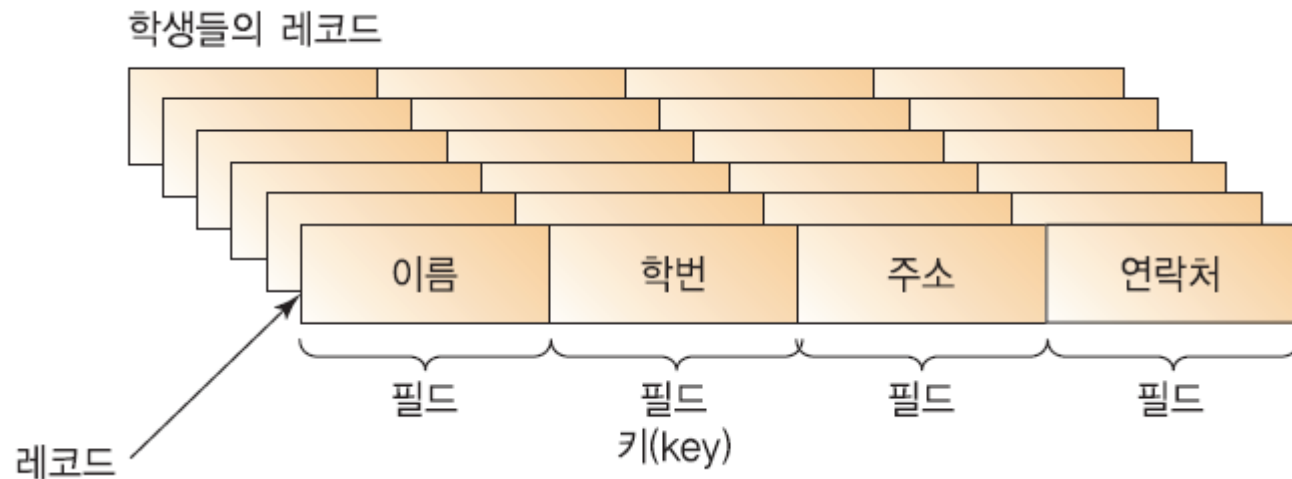


정렬이란?

- ❖ 정렬은 물건을 크기 순으로 오름차순이나 내림차순으로 나열하는 것
- ❖ 정렬은 컴퓨터공학을 포함한 모든 과학기술 분야에서 가장 기본적이고 중요한 알고리즘 중 하나
- ❖ 정렬은 자료 탐색에 있어서 필수적
(예) 만약 영어사전에서 단어들이 알파벳 순으로 정렬되어 있지 않다면?



- ❖ 일반적으로 정렬시켜야 될 대상은 레코드(record)
- ❖ 레코드는 필드(field)라는 보다 작은 단위로 구성
- ❖ 키필드로 레코드와 레코드를 구별한다.



- ❖ 모든 경우에 최적인 정렬 알고리즘은 없음
- ❖ 각 응용 분야에 적합한 정렬 방법 사용해야 함
 - 레코드 수의 많고 적음
 - 레코드 크기의 크고 작음
 - Key의 특성(문자, 정수, 실수 등)
 - 메모리 내부/외부 정렬
- ❖ 정렬 알고리즘의 평가 기준
 - 비교 횟수의 많고 적음
 - 이동 횟수의 많고 적음



❖ 단순하지만 비효율적인 방법

- 선택정렬, 삽입정렬, 버블정렬 등

❖ 복잡하지만 효율적인 방법

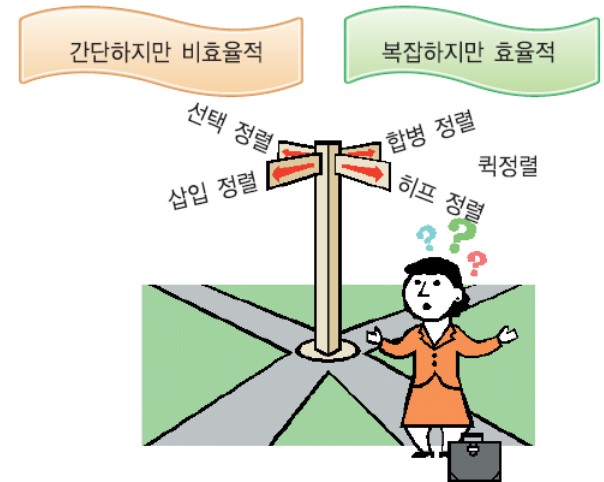
- 퀵정렬, 힙정렬, 합병정렬, 기수정렬 등

❖ 내부 정렬(internal sorting)

- 모든 데이터가 주기억장치에 저장되어진 상태에서 정렬

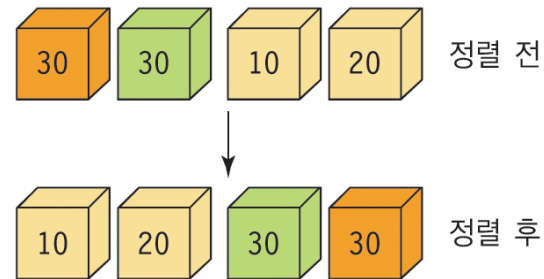
❖ 외부 정렬(external sorting)

- 외부기억장치에 대부분의 데이터가 있고 일부만 주기억장치에 저장된 상태에서 정렬



❖ 정렬 알고리즘의 안정성(stability)

- 동일한 키 값을 갖는 레코드들의 상대적인 위치가 정렬 후에도 바뀌지 않음
- 안정하지 않은 정렬의 예 ==>



2. 선택 정렬

❖ 선택 정렬(selection sort)

- 전체 원소들 중에서 기준 위치에 맞는 원소를 선택하여 자리를 교환하는 방식으로 정렬
- 수행 방법
 - 전체 원소 중에서 가장 작은 원소를 찾아 첫 번째 원소와 자리를 교환
 - 두 번째로 작은 원소를 찾아 선택하여 두 번째 원소와 자리를 교환
 - 세 번째로 작은 원소를 찾아 선택하여 세 번째 원소와 자리를 교환
 - 이 과정을 반복하면서 정렬을 완성



2. 선택 정렬

- 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료를 선택 정렬 방법으로 정렬하는 과정
 - ① 1단계 : 첫째 자리를 기준 위치로 정하고, 전체 중 가장 작은 원소 2를 선택한 후 기준 위치에 있는 원소 69와 자리를 교환



2. 선택 정렬

- ② 2단계 : 둘째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소인 8을 선택한 후 기준 위치에 있는 원소 10과 자리를 교환



2. 선택 정렬

- ③ 3단계 : 셋째 자리를 기준 위치로 정하고, 나머지 원소 중 가장 작은 원소인 10을 선택한 후 기준 위치에 있는 원소 30과 자리를 교환



2. 선택 정렬

- ④ 4단계 : 넷째 자리를 기준 위치로 정하고, 나머지 원소 중 가장 작은 원소인 16을 선택한 후 기준 위치에 있는 원소 69와 자리를 교환



2. 선택 정렬

- ⑤ 5단계 : 다섯째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 22를 선택한 후 기준위치에 있는 원소 69와 자리를 교환



2. 선택 정렬

- ⑥ 6단계 : 여섯째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 30을 선택한 후 기준위치에 있는 원소 30과 자리를 교환(제자리)

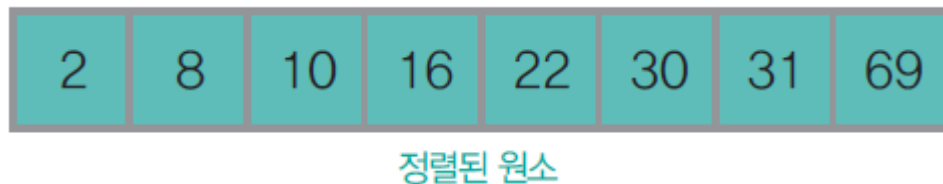


2. 선택 정렬

- ⑦ 7단계 : 일곱째 자리를 기준 위치로 정하고, 나머지 원소 중에서 가장 작은 원소 31을 선택한 후 기준 위치에 있는 원소 69와 자리를 교환(제자리)



- ⑧ 마지막에 남은 원소는 전체 원소 중에서 가장 큰 원소로, 마지막 자리에 남아 이미 정렬된 상태가 되므로 실행을 종료



2. 선택 정렬

❖ 선택 정렬 알고리즘

- 크기가 n 인 배열 $a[]$ 의 원소를 선택 정렬하는 알고리즘

알고리즘 9-1 선택 정렬

```
selectionSort(a[], n)
  for (i ← 1; i < n; i ← i + 1) do {
    a[i], ..., a[n - 1] 중에서 가장 작은 원소 a[k]를 선택해 a[i]와 교환한다.
  }
end selectionSort()
```



2. 선택 정렬

- 메모리 사용공간
 - n개의 원소에 대하여 n개의 메모리 사용
- 비교횟수
 - 1단계 : 첫 번째 원소를 기준으로 n개의 원소 비교
 - 2단계 : 두 번째 원소를 기준으로 마지막 원소까지 n-1개의 원소 비교
 - 3단계 : 세 번째 원소를 기준으로 마지막 원소까지 n-2개의 원소 비교
 - i 단계 : i 번째 원소를 기준으로 n-i개의 원소 비교

$$\text{전체 비교횟수} : (n-1) + (n-2) + \dots + 2 + 1 = \sum_{i=1}^{n-1} n - i = \frac{n(n-1)}{2}$$

- 어떤 경우에서나 비교횟수가 같으므로 시간 복잡도는 $O(n^2)$ 이 됨



2. 선택 정렬

- 선택 정렬하기 프로그램 : [교재 500p](#)
- 실행 결과

```
CA: 명령 프롬프트

정렬할 원소 : 69 10 30 2 16 8 31 22

<<<<<<<<< 선택 정렬 수행 >>>>>>>>>

1단계 : 2  10  30  69  16   8  31  22
2단계 : 2   8  30  69  16  10  31  22
3단계 : 2   8  10  69  16  30  31  22
4단계 : 2   8  10  69  69  30  31  22
5단계 : 2   8  10  16  22  30  31  69
6단계 : 2   8  10  16  22  30  31  69
7단계 : 2   8  10  16  22  30  31  69
```



❖ 비교 횟수

- $(n - 1) + (n - 2) + \cdots + 1 = n(n - 1)/2 = O(n^2)$

❖ 이동 횟수

- $3(n - 1)$

❖ 전체 시간적 복잡도: $O(n^2)$

❖ 안정성을 만족하지 않음

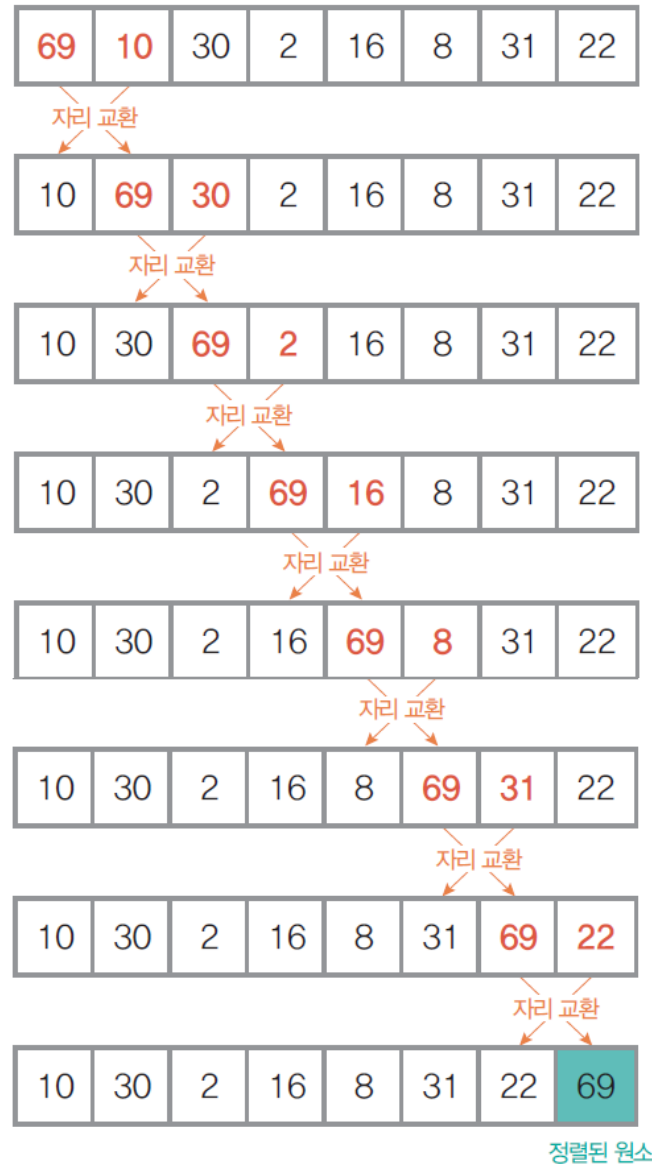


❖ 버블 정렬 bubble sort의 이해

- 인접한 두 개의 원소를 비교하여 자리를 교환하는 방식
 - 첫 번째 원소부터 마지막 원소까지 반복하여 한 단계가 끝나면 가장 큰 원소가 마지막 자리로 정렬
 - 첫 번째 원소부터 인접한 원소끼리 계속 자리를 교환하면서 맨 마지막 자리로 이동하는 모습이 물 속에서 물 위로 올라오는 물방울 모양과 같다고 하여 버블(bubble) 정렬이라 함.
- 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 버블 정렬 방법으로 정렬하는 과정
 - ① 1단계 : 인접한 두 원소를 비교하여 자리를 교환하는 작업을 첫 번째 원소부터 마지막 원소까지 차례로 반복하여 가장 큰 원소 69를 마지막 자리로 정렬.



3. 버블 정렬



정렬된 원소



3. 버블 정렬

② 2단계 : 정렬하지 않은 원소에 대해 두 번째 버블 정렬을 수행하면, 나머지 원소 중에서 가장 큰 원소 31이 끝에서 둘째 자리로 정렬

10	30	2	16	8	31	22	69
----	----	---	----	---	----	----	----

10	30	2	16	8	31	22	69
----	----	---	----	---	----	----	----

자리 교환

10	2	30	16	8	31	22	69
----	---	----	----	---	----	----	----

자리 교환

10	2	16	30	8	31	22	69
----	---	----	----	---	----	----	----

자리 교환

10	2	16	8	30	31	22	69
----	---	----	---	----	----	----	----

10	2	16	8	30	31	22	69
----	---	----	---	----	----	----	----

자리 교환

10	2	16	8	30	22	31	69
----	---	----	---	----	----	----	----

정렬된 원소

3. 버블 정렬

- ③ 3단계 : 세 번째 버블 정렬을 수행하면, 나머지 원소 중에서 가장 큰 원소 30이 끝에서 셋째 자리로 정렬

10	2	16	8	30	22	31	69
----	---	----	---	----	----	----	----

자리 교환

2	10	16	8	30	22	31	69
---	----	----	---	----	----	----	----

2	10	16	8	30	22	31	69
---	----	----	---	----	----	----	----

자리 교환

2	10	8	16	30	22	31	69
---	----	---	----	----	----	----	----

2	10	8	16	30	22	31	69
---	----	---	----	----	----	----	----

자리 교환

2	10	8	16	22	30	31	69
---	----	---	----	----	----	----	----

정렬된 원소

3. 버블 정렬

- ④ 4단계 : 네 번째 버블 정렬을 정렬하지 않은 원소에 대해 수행하면, 나머지 원소 중에서 가장 큰 원소 22가 끝에서 넷째 자리로 정렬

2	10	8	16	22	30	31	69
---	----	---	----	----	----	----	----

2	10	8	16	22	30	31	69
---	----	---	----	----	----	----	----

자리 교환

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

정렬된 원소



3. 버블 정렬

- ⑤ 5단계 : 다섯 번째 버블 정렬을 수행하면, 나머지 원소 중에서 가장 큰 원소 16이 끝에서 다섯째 자리로 정렬

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

정렬된 원소



3. 버블 정렬

- ⑥ 6단계 : 여섯 번째 버블 정렬을 수행하면, 나머지 원소 중에서 가장 큰 원소 10이 끝에서 여섯째 자리로 정렬

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

정렬된 원소



3. 버블 정렬

⑦ 7단계 : 나머지 원소 중에서 가장 큰 원소 8이 끝에서 일곱째 자리로 정렬

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

정렬된 원소

⑧ 마지막에 남은 첫째 원소는 전체 원소 중에서 가장 작은 원소로, 가장 앞자리에 남아 이미 정렬된 상태가 되므로 실행을 종료

2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----



2	8	10	16	22	30	31	69
---	---	----	----	----	----	----	----

정렬된 원소



3. 버블 정렬

❖ 버블 정렬 알고리즘

- 크기가 n 인 배열 $a[]$ 의 원소를 버블 정렬하는 알고리즘

알고리즘 9-2 버블 정렬

```
bubbleSort(a[], n)
  for (i ← n - 1; i ≥ 0; i ← i - 1) do {
    for (j ← 0; j < i; j ← j + 1) do {
      if (a[j] > a[j + 1]) then {
        temp ← a[j];
        a[j] ← a[j + 1];
        a[j + 1] ← temp;
      }
    }
  }
end bubbleSort()
```



3. 버블 정렬

- 메모리 사용공간
 - n개의 원소에 대하여 n개의 메모리 사용
- 연산 시간
 - 최선의 경우 : 자료가 이미 정렬되어있는 경우
 - 비교횟수 : i번째 원소를 (n-i)번 비교하므로, $\frac{n(n-1)}{2}$ 번
 - 자리교환횟수 : 자리교환이 발생하지 않음
 - 최악의 경우 : 자료가 역순으로 정렬되어있는 경우
 - 비교횟수 : i번째 원소를 (n-i)번 비교하므로, $\frac{n(n-1)}{2}$ 번
 - 자리교환횟수 : i번째 원소를 (n-i)번 교환하므로, $\frac{n(n-1)}{2}$ 번
 - 최선의 경우와 최악의 경우에 대한 평균 시간 복잡도를 빅-오 Big Oh 표기법으로 나타내면 $O(n^2)$ 이 됨



3. 버블 정렬



- 전체 비교 횟수: $1+2+3+\dots+(n-1) = \frac{n(n-1)}{2}$
- 전체 자리 교환 횟수: 0
- 시간 복잡도: $O(n^2)$

(a) 순차 정렬된 경우: 최선의 경우

- 전체 비교 횟수: $1+2+3+\dots+(n-1) = \frac{n(n-1)}{2}$
- 전체 자리 교환 횟수: $1+2+3+\dots+(n-1) = \frac{n(n-1)}{2}$
- 시간 복잡도: $O(n^2)$

(b) 역순 정렬된 경우: 최악의 경우

그림 9-2 순차 정렬된 경우와 역순 정렬된 경우의 버블 정렬 비교



2. 버블 정렬

- 버블 정렬하기 프로그램 : [교재 507p](#)
- 실행 결과

```
영랑 프로그래밍
정렬할 원소 : 69 10 30 2 16 8 31 22
<<<<<<<<< 버블 정렬 수행 >>>>>>>>>>>>
1단계>>
  10 69 30 2 16 8 31 22
  10 30 69 2 16 8 31 22
  10 30 2 69 16 8 31 22
  10 30 2 16 69 8 31 22
  10 30 2 16 8 69 31 22
  10 30 2 16 8 31 69 22
  10 30 2 16 8 31 22 69
2단계>>
  10 30 2 16 8 31 22 69
  10 2 30 16 8 31 22 69
  10 2 16 30 8 31 22 69
  10 2 16 8 30 31 22 69
  10 2 16 8 30 31 22 69
  10 2 16 8 30 22 31 69
3단계>>
  2 10 16 8 30 22 31 69
  2 10 16 8 30 22 31 69
  2 10 8 16 30 22 31 69
  2 10 8 16 30 22 31 69
  2 10 8 16 22 30 31 69
```

```
4단계>>
  2 10 8 16 22 30 31 69
  2 8 10 16 22 30 31 69
  2 8 10 16 22 30 31 69
  2 8 10 16 22 30 31 69
5단계>>
  2 8 10 16 22 30 31 69
  2 8 10 16 22 30 31 69
  2 8 10 16 22 30 31 69
6단계>>
  2 8 10 16 22 30 31 69
  2 8 10 16 22 30 31 69
7단계>>
  2 8 10 16 22 30 31 69
```

❖ 퀵 정렬(quick sort)의 이해

- 정렬할 전체 원소에 대해서 정렬을 수행하지 않고, 기준 값을 중심으로 왼쪽 부분 집합과 오른쪽 부분 집합으로 분할하여 정렬하는 방법
 - 왼쪽 부분 집합에는 기준 값보다 작은 원소들을 이동시키고, 오른쪽 부분 집합에는 기준 값보다 큰 원소들을 이동시킴
 - 기준 값 : 피벗(pivot)
 - 일반적으로 전체 원소 중에서 가운데에 위치한 원소를 선택
- 퀵 정렬은 다음의 두 가지 기본 작업을 반복 수행하여 완성
 - 분할(divide)
 - 정렬할 자료들을 기준값을 중심으로 두 개로 나눠 부분집합을 만듦
 - 정복(conquer)
 - 부분집합 안에서 기준값의 정렬 위치를 정함



■ 퀵 정렬 동작 규칙

- ① 왼쪽 끝에서 오른쪽으로 움직이면서 크기를 비교하여 피벗보다 크거나 같은 원소를 찾아 L로 표시한다. 단, L은 R과 만나면 더 이상 오른쪽으로 이동하지 못하고 멈춘다.
- ② 오른쪽 끝에서 왼쪽으로 움직이면서 피벗보다 작은 원소를 찾아 R로 표시한다. 단, R은 L과 만나면 더 이상 왼쪽으로 이동하지 못하고 멈춘다.
- ③-a ①과 ②에서 찾은 L 원소와 R 원소가 있는 경우, 서로 교환하고 L과 R의 현재 위치에서 ①과 ② 작업을 다시 수행한다.
- ③-b ① ~ ②를 수행하면서 L과 R이 같은 원소에서 만나 멈춘 경우, 피벗과 R의 원소를 서로 교환한다. 교환된 자리를 피벗 위치로 확정하고 현재 단계의 퀵 정렬을 끝낸다.
- ④ 피벗의 확정된 위치를 기준으로 만들어진 새로운 왼쪽 부분집합과 오른쪽 부분집합에 대해서 ① ~ ③의 퀵 정렬을 순환적으로 반복 수행하는데, 모든 부분집합의 크기가 1 이하가 되면 전체 퀵 정렬을 종료한다.

그림 9-3 퀵 정렬을 위한 동작 규칙



4. 퀵 정렬

- 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 퀵 정렬 방법으로 정렬하는 과정 (가운데 원소를 피봇으로 정함)

(1) 1단계 : ① 원소 2를 피봇으로 선택하고 퀵 정렬을 시작 ② L은 정렬 범위의 왼쪽 끝에서 오른쪽으로 움직이면서 피봇보다 크거나 같은 원소를 찾고, R은 정렬 범위의 오른쪽 끝에서 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾음. L은 원소 69를 찾았지만, R은 피봇보다 작은 원소를 찾지 못한 상태로 원소 69에서 L과 만남. L과 R이 만나 더 이상 진행할 수 없는 상태가 되었으므로, ③ 원소 69를 피봇과 자리를 교환하고 피봇 원소 2의 위치를 확정



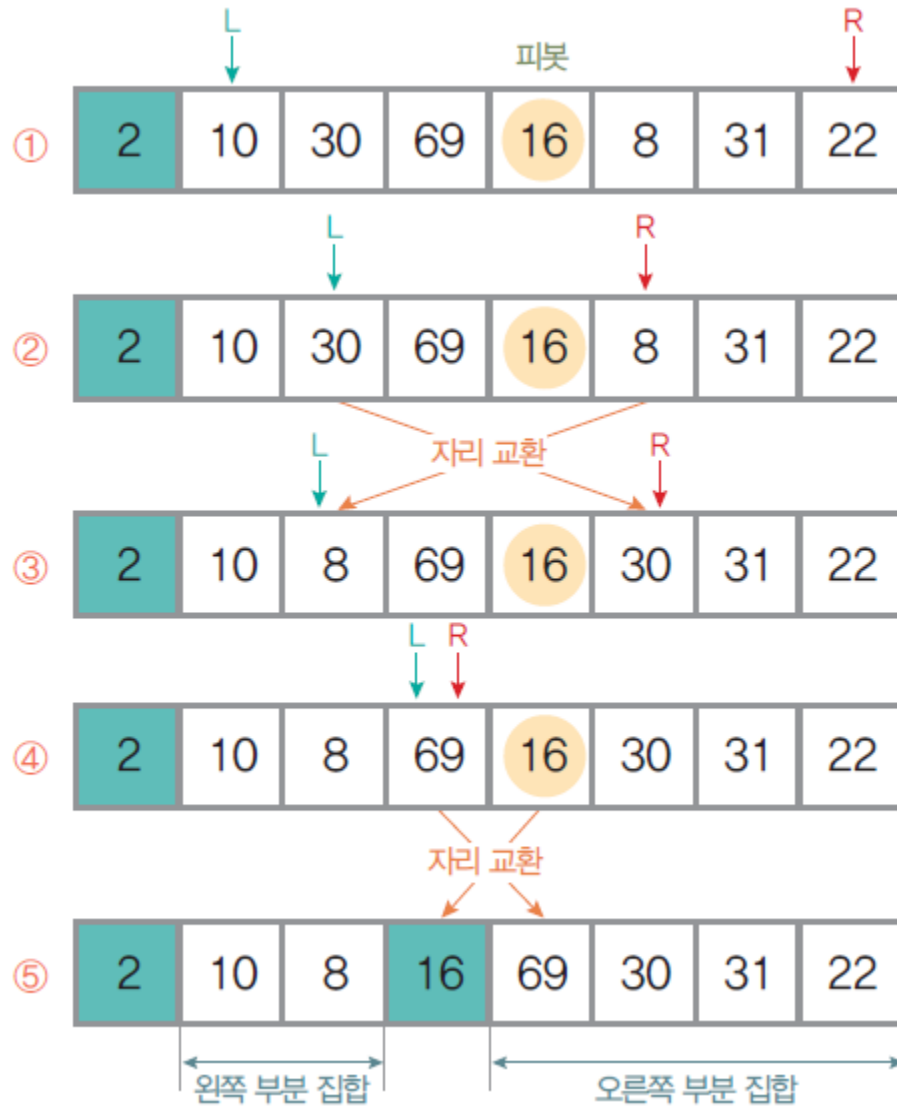
4. 퀵 정렬

(2) 2단계 : 위치가 확정된 피벗 2의 왼쪽 부분집합은 공집합이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분집합에 대해서 퀵 정렬을 수행.

- ① 오른쪽 부분집합의 원소가 일곱 개이므로 가운데 있는 원소 16을 피벗으로 선택하고 퀵 정렬을 시작.
- ② L은 오른쪽으로 움직이면서 피벗보다 크거나 같은 원소인 30을 찾고, R은 왼쪽으로 움직이면서 피벗보다 작은 원소인 8을 찾음.
- ③ L이 찾은 30과 R이 찾은 8을 서로 자리를 교환한 후 현재 위치에서 L은 다시 오른쪽으로 움직이면서 피벗보다 크거나 같은 원소 69를 찾고 R은 피벗보다 작은 원소를 찾음.
- ④ R이 원소 69에서 L과 만나 더 이상 진행할 수 없는 상태가 되었으므로,
- ⑤ 원소 69를 피벗과 교환하고 피벗 원소 16의 위치를 확정



4. 퀵 정렬



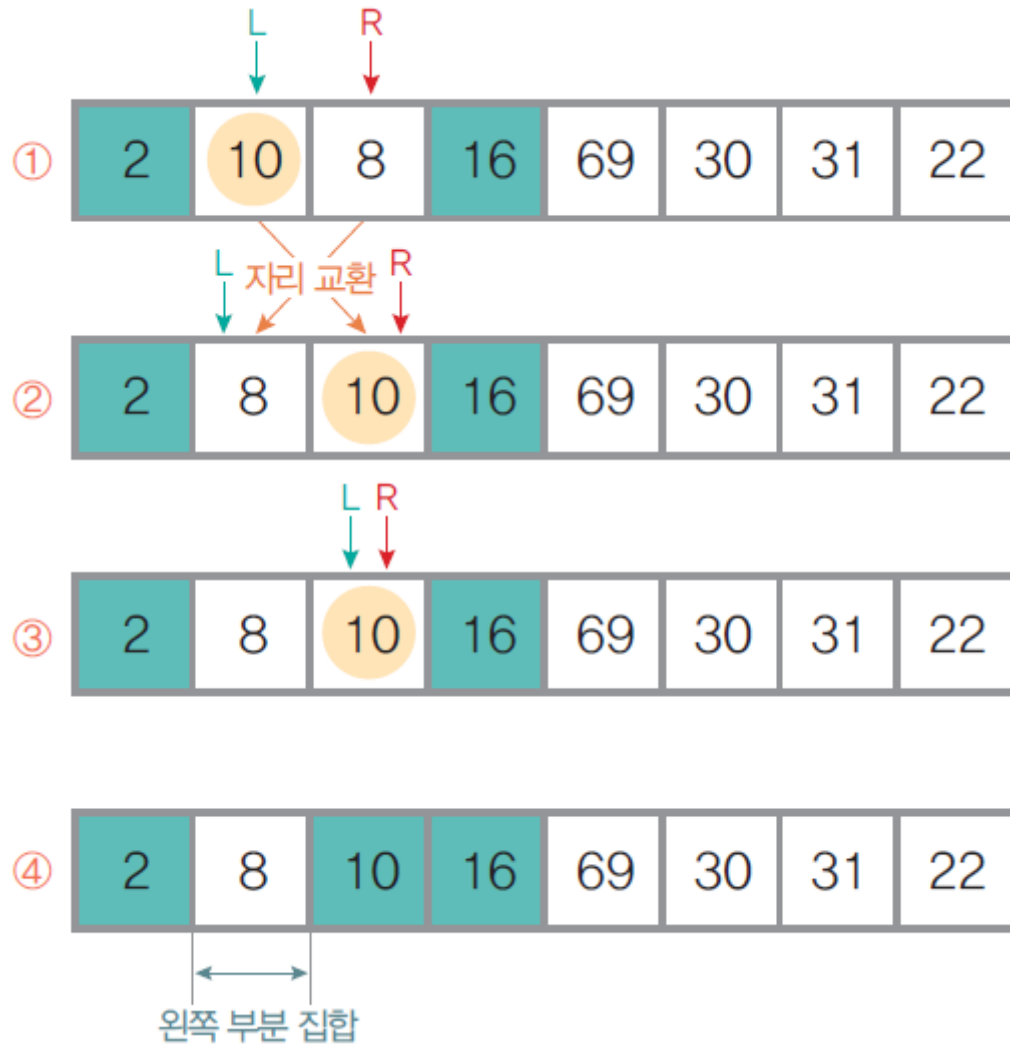
4. 퀵 정렬

(3) 3단계 : 위치가 확정된 피벗 16을 기준으로 새로 생긴 왼쪽 부분집합에서 퀵 정렬을 수행

- ① 원소 10을 피벗으로 선택하고 L은 오른쪽으로 움직이면서 피벗보다 크거나 같은 원소를, R은 왼쪽으로 움직이면서 피벗보다 작은 원소를 찾음
- ② L이 찾은 10과 R이 찾은 8을 서로 교환
- ③ 현재 위치에서 L은 다시 오른쪽으로 움직이다가 원소 10에서 R과 만나서 멈추게 됨. 더 이상 진행할 수 없는 상태가 되었으므로,
- ④ R의 원소와 피벗을 교환하고 피벗 원소 10의 위치를 확정한다(이 경우에는 R과 피벗 위치가 같았으므로, 자리를 교환하기 전과 후의 상태가 같음)



4. 퀵 정렬



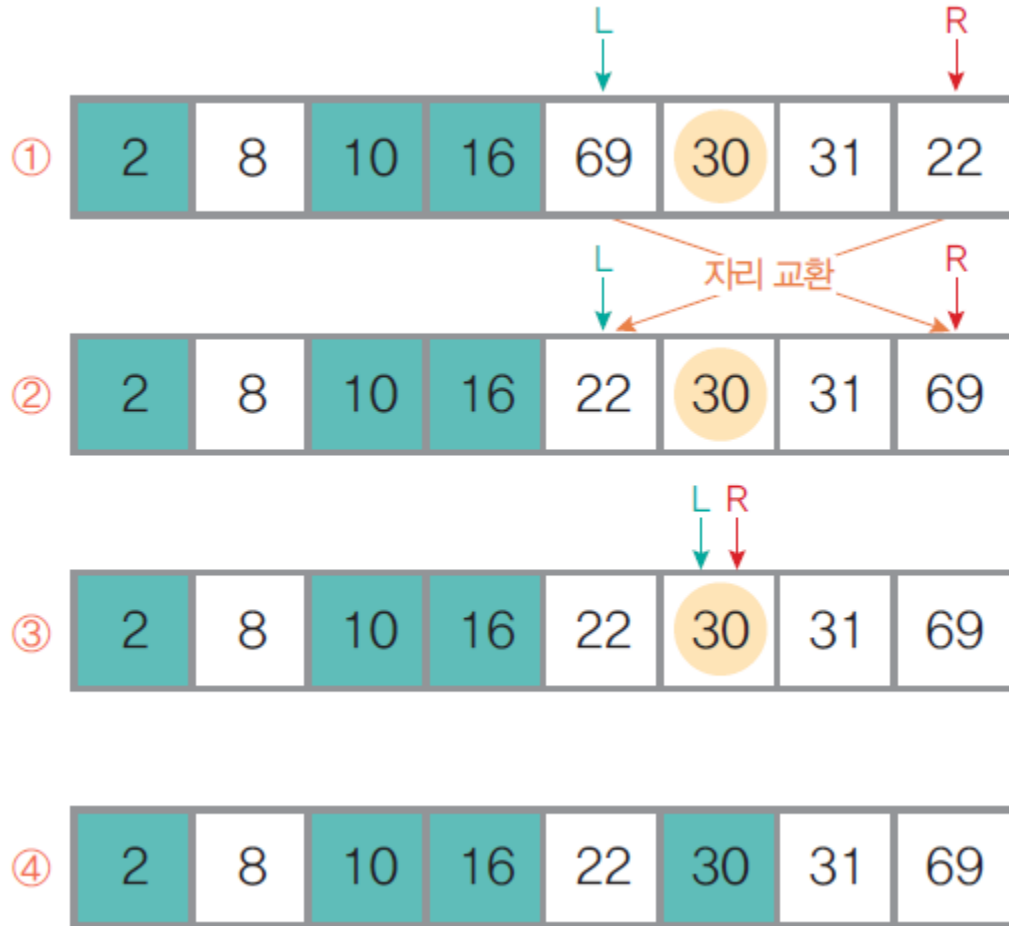
4. 퀵 정렬

(4) 4단계 : 피봇 10의 왼쪽 부분집합은 원소가 한 개이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분집합은 공집합이므로 역시 퀵 정렬을 수행하지 않음
이제 [2]에서 피봇이었던 원소 16의 오른쪽 부분집합에 대해서 퀵 정렬을 수행

- ① 오른쪽 부분집합의 원소가 네 개이므로 가운데 있는 원소 30을 피봇으로 선택. L은 오른쪽으로 이동하면서 피봇보다 크거나 같은 원소를 찾고, R은 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾음
- ② L이 찾은 69와 R이 찾은 22를 서로 교환. 현재 위치에서 다시 L은 피봇보다 크거나 같은 원소를 찾아 오른쪽으로 움직이고 R은 피봇보다 작은 원소를 찾아 왼쪽으로 움직이다가
- ③ 원소 30에서 L과 R이 만나 멈춤. 더 이상 진행할 수 없으므로
- ④ R의 원소와 피봇을 교환하고 피봇 원소30의 위치가 확정
(이 경우에 R과 피봇 위치가 같았으므로, 자리를 교환하기 전과 후의 상태가 같음).

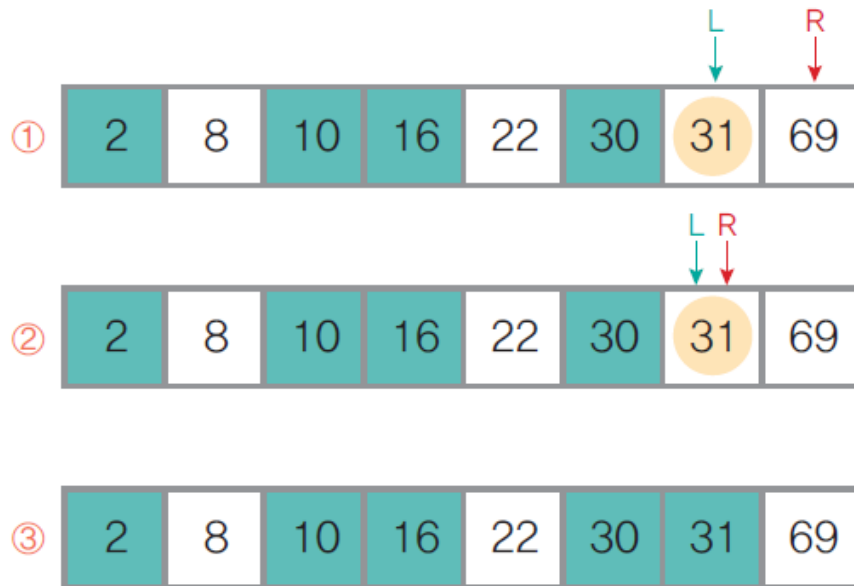


4. 퀵 정렬



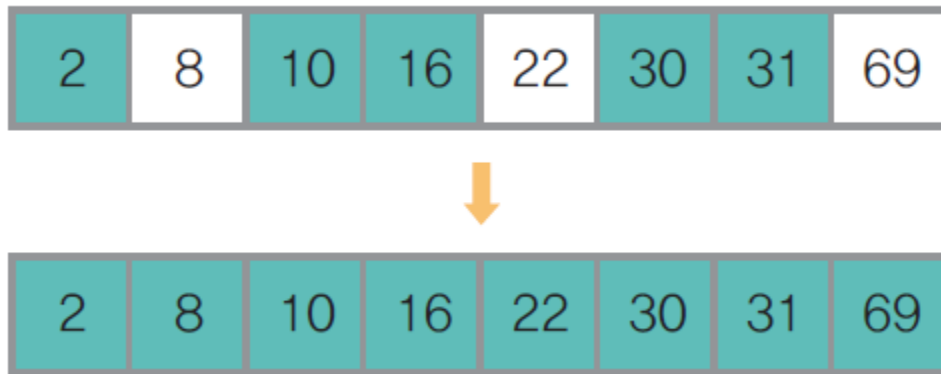
4. 퀵 정렬

- (5) 5단계 : 피벗 30의 왼쪽 부분집합의 원소가 한 개이므로 퀵 정렬을 수행하지 않고, 오른쪽 부분집합에 대해서 퀵 정렬을 수행
- ① 이때 피벗은 31을 선택하고 L은 오른쪽으로 움직이면서 피벗보다 크거나 같은 원소를 찾고 R은 왼쪽으로 움직이면서 피벗보다 작은 원소를 찾음
 - ② L과 R이 원소 31에서 만나 더 이상 진행하지 못하는 상태가 되어
 - ③ 원소 31을 피벗과 교환하여 위치를 확정(이 경우에는 R과 피벗 위치가 같았으므로, 자리를 교환하기 전과 후의 상태가 같음).



4. 퀵 정렬

- (6) 6단계 : 피봇 31의 오른쪽 부분집합의 원소가 한 개이므로 퀵 정렬을 수행하지 않음. 모든 부분 집합의 크기가 1 이하이므로 전체 퀵 정렬을 종료



4. 퀵 정렬

❖ 퀵 정렬 알고리즘

- 크기가 n 인 배열 $a[]$ 의 원소를 퀵 정렬하는 알고리즘

알고리즘 9-3 퀵 정렬

```
quickSort(a[], begin, end)
  if (m < n) then {
    p ← partition(a, begin, end);
    quickSort(a[], begin, p-1);
    quickSort(a[], p+1, end);
  }
end quickSort()
```



4. 퀵 정렬

- 퀵 정렬 알고리즘은 배열 $a[]$ 를 두 개로 분할하는 `partition()` 연산이 필요

알고리즘 9-4 파티션 분할

```
partiton(a[], begin, end)
    pivot ← (begin + end) / 2;
    L ← begin;
    R ← end;
    while (L < R) do {
        while (a[L] < a[pivot] and L < R) do L++;
        while (a[R] ≥ a[pivot] and L < R) do R--;
        if (L < R) then { // L의 원소와 R의 원소 교환
            temp ← a[L];
            a[L] ← a[R];
            a[R] ← temp;
        }
    }
    temp ← a[pivot]; // R의 원소와 피벗 원소 교환
    a[pivot] ← a[R];
    a[R] ← temp;
    return R; // 교환된 R의 자리를 분할 기준 자리로 반환
end partiton()
```

4. 퀵 정렬

- 메모리 사용공간
 - n 개의 원소에 대하여 n 개의 메모리 사용
- 연산 시간
 - 최선의 경우
 - 피봇에 의해서 원소들이 왼쪽 부분 집합과 오른쪽 부분 집합으로 정확히 $n/2$ 개씩 2등분이 되는 경우가 반복되어 수행 단계 수가 최소가 되는 경우
 - 최악의 경우
 - 피봇에 의해 원소들을 분할하였을 때 한개와 $n-1$ 개로 한쪽으로 치우쳐 분할되는 경우가 반복되어 수행 단계 수가 최대가 되는 경우
 - 평균 시간 복잡도 : $O(n \log_2 n)$
 - 같은 시간 복잡도를 가지는 다른 정렬 방법에 비해서 자리 교환 횟수를 줄임으로써 더 빨리 실행되어 실행 시간 성능이 좋은 정렬 방법임.



4. 퀵 정렬

- 퀵 정렬하기 프로그램 1 : [교재 514p](#)
- 실행 결과

```
명령 프롬프트

[ 초기 상태 ]
69 10 30 2 16 8 31 22

[ 1단계 : pivot=2 ]
2 10 30 69 16 8 31 22

[ 2단계 : pivot=16 ]
2 10 8 16 69 30 31 22

[ 3단계 : pivot=10 ]
2 8 10 16 69 30 31 22

[ 4단계 : pivot=30 ]
2 8 10 16 22 30 31 69

[ 5단계 : pivot=31 ]
2 8 10 16 22 30 31 69
```



4. 퀵 정렬

- 퀵 정렬하기 프로그램 2 : [교재 514p](#)
 - 피봇 위치를 첫째 원소로 정하여 퀵 정렬
- 실행 결과

```
명령 프롬프트
[ 초기 상태 ]
69 10 30 2 16 8 31 22

[ 1단계 : pivot=69 ]
22 10 30 2 16 8 31 69

[ 2단계 : pivot=22 ]
8 10 16 2 22 30 31 69

[ 3단계 : pivot=8 ]
2 8 16 10 22 30 31 69

[ 4단계 : pivot=16 ]
2 8 10 16 22 30 31 69

[ 5단계 : pivot=30 ]
2 8 10 16 22 30 31 69
```



❖ 삽입 정렬 insert sort의 이해

- 정렬되어있는 부분집합에 정렬할 새로운 원소의 위치를 찾아 삽입하는 방법
- 정렬할 자료를 두 개의 부분집합 $S^{\text{Sorted Subset}}$ 와 $U^{\text{Unsorted Subset}}$ 로 가정
 - 부분집합 S : 정렬된 앞부분의 원소들
 - 부분집합 U : 아직 정렬되지 않은 나머지 원소들
 - 정렬되지 않은 부분집합 U 의 원소를 하나씩 꺼내서 이미 정렬되어있는 부분집합 S 의 마지막 원소부터 비교하면서 위치를 찾아 삽입
 - 삽입 정렬을 반복하면서 부분집합 S 의 원소는 하나씩 늘리고 부분집합 U 의 원소는 하나씩 감소하게 함. 부분집합 U 가 공집합이 되면 삽입 정렬이 완성

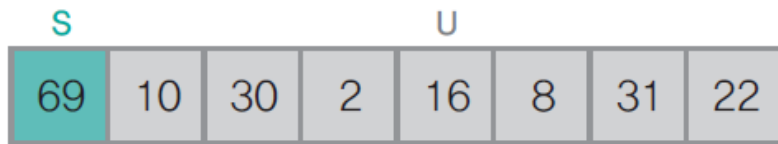


5. 삽입 정렬

❖ 삽입 정렬 수행 과정

- 정렬하지 않은 {69, 10, 30, 2, 16, 8, 31, 22} 자료를 삽입 정렬 방법으로 정렬하는 과정

0. 초기 상태 : 첫째 원소는 정렬되어 있는 부분집합 S로, 나머지 원소는 정렬되지 않은 원소들의 부분집합 U로 가정



$S = \{69\}$

$U = \{10, 30, 2, 16, 8, 31, 22\}$

- ① 1단계 : U의 첫째 원소 10을 S의 마지막 원소 69와 비교하면 $10 < 69$ 이므로 원소 10은 원소 69의 앞자리가 됨. 더 이상 비교할 S의 원소가 없으므로 찾은 위치에 원소 10을 삽입



삽입



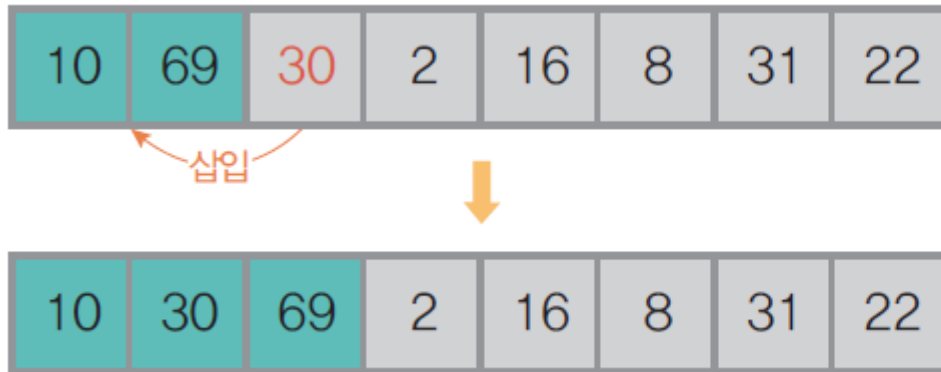
$S = \{10, 69\}$

$U = \{30, 2, 16, 8, 31, 22\}$



5. 삽입 정렬

- ② 2단계 : 현재 U의 첫째 원소 30을 S의 마지막 원소 69와 비교하면 $30 < 69$ 이므로 원소 69의 앞자리 원소 10과 비교. $30 > 10$ 이므로 원소 10과 69 사이에 삽입



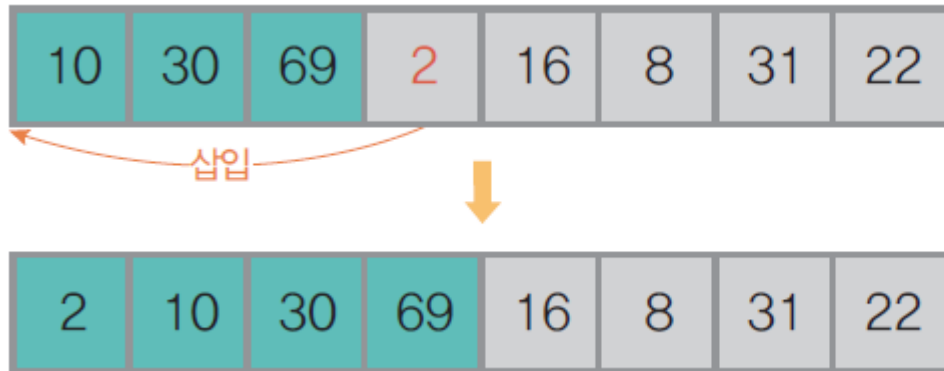
$S = \{10, 30, 69\}$

$U = \{2, 16, 8, 31, 22\}$



5. 삽입 정렬

- ③ 3단계 : 현재 U의 첫째 원소 2를 S의 마지막 원소 69와 비교하면 $2 < 69$ 이므로 원소 69의 앞자리 원소 30과 비교. $2 < 30$ 이므로 다시 원소 30의 앞자리 원소 10과 비교. $2 < 10$ 이고 더 이상 비교할 S의 원소가 없으므로 원소 10 앞에 삽입



$S = \{2, 10, 30, 69\}$
 $U = \{16, 8, 31, 22\}$



5. 삽입 정렬

- ④ 4단계 : 현재 U의 첫째 원소 16을 S의 마지막 원소 69와 비교하면 $16 < 69$ 이므로 그 앞자리 원소 30과 비교. $16 < 30$ 이므로 다시 그 앞자리 원소 10과 비교. $16 > 10$ 이므로 원소 10과 30 사이에 삽입

2	10	30	69	16	8	31	22
---	----	----	----	----	---	----	----

삽입

2	10	16	30	69	8	31	22
---	----	----	----	----	---	----	----

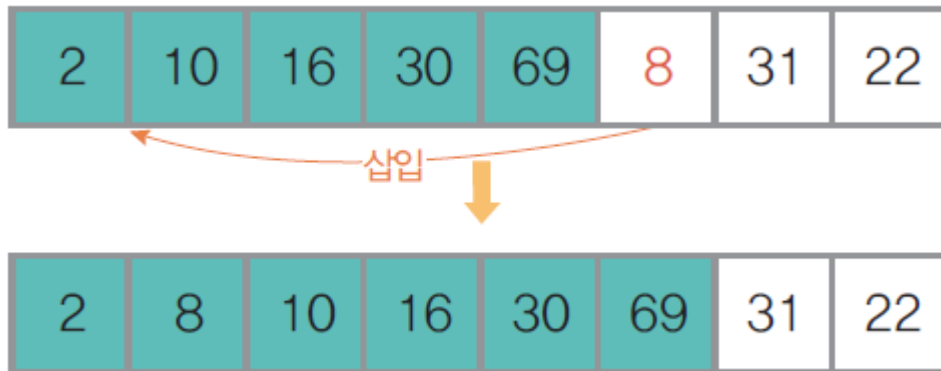
$S = \{2, 10, 16, 30, 69\}$

$U = \{8, 31, 22\}$



5. 삽입 정렬

- ⑤ 5단계 : 현재 U의 첫 번째 원소 8을 S의 마지막 원소 69와 비교하면 $8 < 69$ 이므로 그 앞자리 원소 30과 비교. $8 < 30$ 이므로 그 앞자리 원소 16과 비교. $8 < 16$ 이므로 그 앞자리 원소 10과 비교. $8 < 10$ 이므로 다시 그 앞자리 원소 2와 비교. $8 > 2$ 이므로 원소 2와 10 사이에 삽입

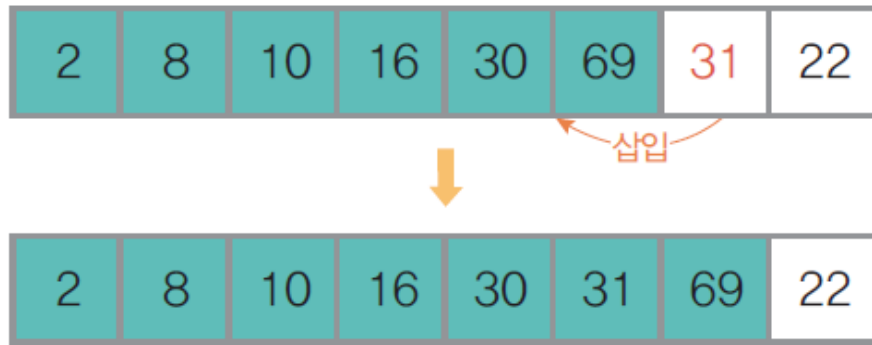


$S = \{2, 8, 10, 16, 30, 69\}$
 $U = \{31, 22\}$



5. 삽입 정렬

- ⑥ 6단계 : 현재 U의 첫째 원소 31을 S의 마지막 원소 69와 비교하면 $31 < 69$ 이므로 그 앞자리 원소 30과 비교. $31 > 30$ 이므로 원소 30과 69 사이에 삽입

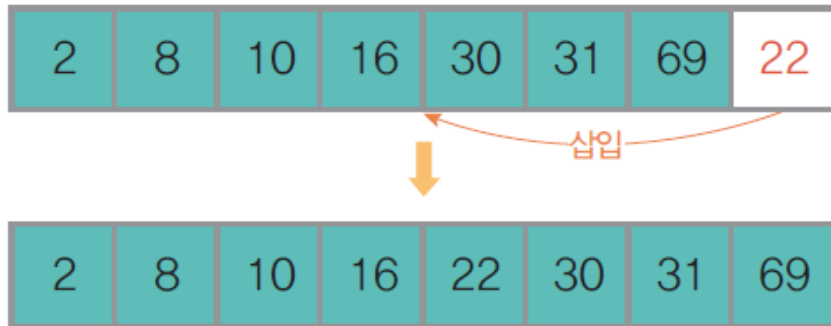


$S = \{2, 8, 10, 16, 30, 31, 69\}$
 $U = \{22\}$



5. 삽입 정렬

- ⑦ 7단계 : 현재 U의 첫째 원소 22를 S의 마지막 원소 69와 비교하면 $22 < 69$ 이므로 그 앞자리 원소 31과 비교. $22 < 31$ 이므로 그 앞자리 원소 30과 비교. $22 < 30$ 이므로 다시 그 앞자리 원소 16과 비교. $22 > 16$ 이므로 원소 16과 30 사이에 삽입. U가 공집합이 되었으므로 실행을 종료



$S = \{2, 8, 10, 16, 22, 30, 31, 69\}$
 $U = \{ \}$



5. 삽입 정렬

- 크기가 n 인 배열 $a[]$ 의 원소를 삽입 정렬하는 알고리즘

알고리즘 9-5 삽입 정렬

```
insertionSort(a[], n)
  for (i ← 1; i < n; i ← i + 1) do {
    temp ← a[i];
    j ← i;
    if (a[j - 1] > temp) then move ← true; // 삽입 자리를 만들기 위해 이동할 원소 표시
    else move ← false;
    while (move and j > 0) do {           // 삽입 자리를 만들기 위해
      a[j] ← a[j - 1];                   // 삽입 자리 이후의 원소를 뒤로 이동
      j ← j - 1;
    }
    a[j] ← temp;                          // 삽입 자리에 원소 저장
  }
end insertionSort()
```

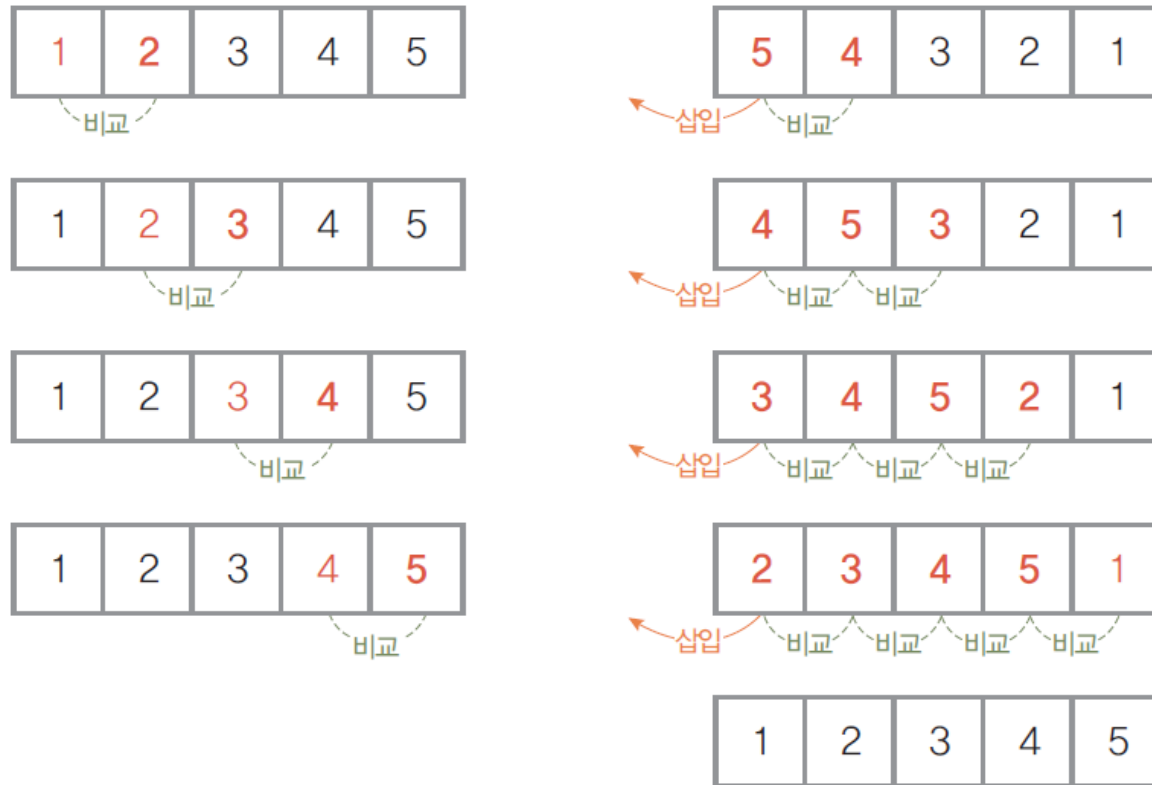


5. 삽입 정렬

- 메모리 사용공간
 - n 개의 원소에 대하여 n 개의 메모리 사용
- 연산 시간
 - 최선의 경우 : 원소들이 이미 정렬되어있어서 비교횟수가 최소인 경우
 - 이미 정렬되어있는 경우에는 바로 앞자리 원소와 한번만 비교
 - 전체 비교횟수 = $n-1$
 - 시간 복잡도 : $O(n)$
 - 최악의 경우 : 모든 원소가 역순으로 되어있어서 비교횟수가 최대인 경우
 - 전체 비교횟수 = $1+2+3+\dots+(n-1) = n(n-1)/2$
 - 시간 복잡도 : $O(n^2)$
 - 삽입 정렬의 평균 비교횟수 = $n(n-1)/4$
 - 평균 시간 복잡도 : $O(n^2)$



5. 삽입 정렬



- 전체 비교 횟수 : $n-1$
 - 전체 자리 이동 횟수 : 0
- 시간 복잡도 : $O(n)$

(a) 순차 정렬된 경우

- 전체 비교 횟수 : $1+2+3+\dots+(n-1) = \sum_{i=1}^{n-1} i$
 - 전체 자리 이동 횟수 : $1+2+3+\dots+(n-1) = \sum_{i=1}^{n-1} i$
 - 실행 연산 횟수 : $\sum_{i=1}^{n-1} (\text{비교 횟수} + \text{자리 이동 횟수}) = \sum_{i=1}^{n-1} (i + i)$
- 시간 복잡도 : $O(n^2)$

(b) 역순 정렬된 경우

그림 9-4 순차 정렬된 경우와 역순 정렬된 경우의 삽입 정렬 비교

5. 삽입 정렬

- 삽입 정렬하기 프로그램 : [교재 522p](#)
- 실행 결과

```
명령 프롬프트

정렬할 원소 : 69 10 30 2 16 8 31 22

<<<<<<<<< 삽입 정렬 수행 >>>>>>>>>

1단계 : 10 69 30 2 16 8 31 22
2단계 : 10 30 69 2 16 8 31 22
3단계 : 2 10 30 69 16 8 31 22
4단계 : 2 10 16 30 69 8 31 22
5단계 : 2 8 10 16 30 69 31 22
6단계 : 2 8 10 16 30 31 69 22
7단계 : 2 8 10 16 22 30 31 69
```



❖ 셀 정렬 shell sort의 이해

- 일정한 간격(interval)으로 떨어져있는 자료들끼리 부분집합을 구성하고 각 부분집합에 있는 원소들에 대해서 삽입 정렬을 수행하는 작업을 반복하면서 전체 원소들을 정렬하는 방법
 - 전체 원소에 대해서 삽입 정렬을 수행하는 것보다 부분집합으로 나누어 정렬하게 되면 비교연산과 교환연산 감소
- 셀 정렬의 부분집합
 - 부분집합의 기준이 되는 간격을 매개변수 h 에 저장
 - 한 단계가 수행될 때마다 h 의 값을 감소시키고 셀 정렬을 순환 호출
 - h 가 1이 될 때까지 반복
- 셀 정렬의 성능은 매개변수 h 의 값에 따라 달라짐
 - 정렬할 자료의 특성에 따라 매개변수 생성 함수를 사용
 - 일반적으로 사용하는 h 의 값은 원소 개수의 $1/2$ 을 사용하고 한 단계 수행될 때마다 h 의 값을 반으로 감소시키면서 반복 수행

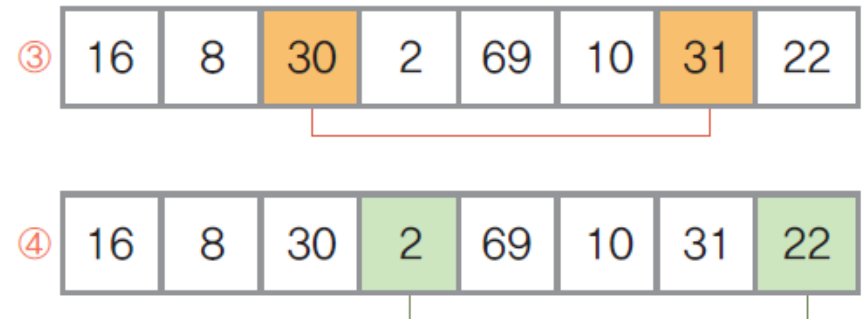
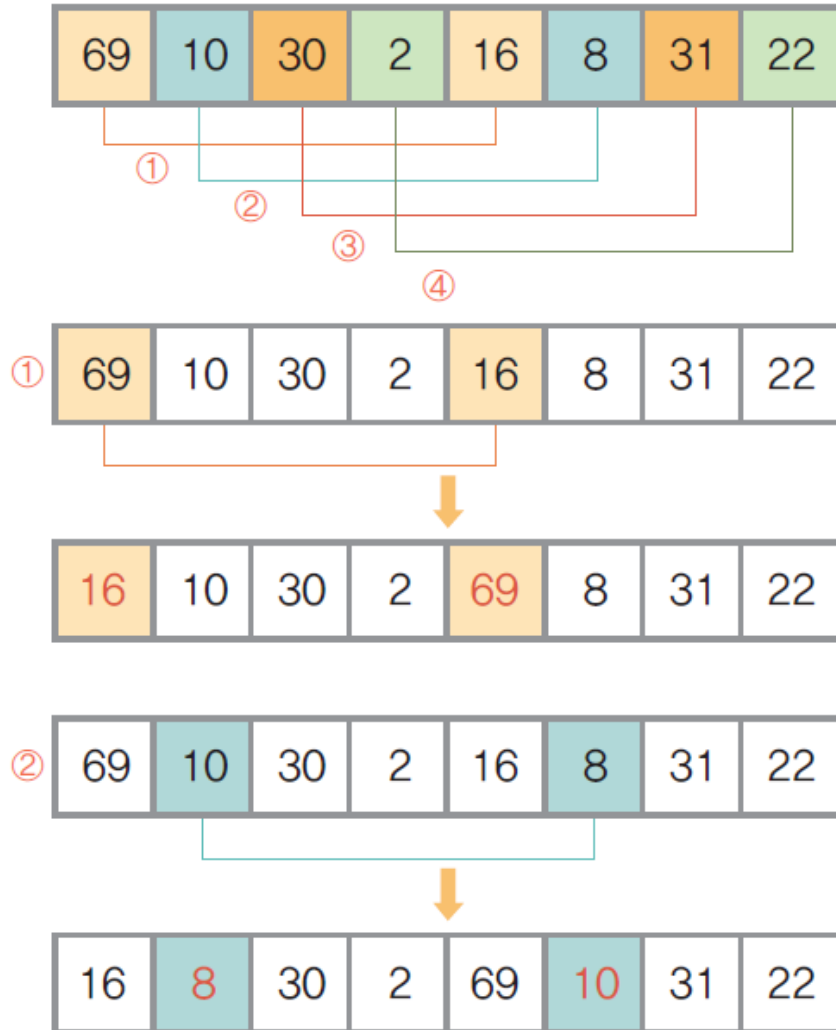


6. 셀 정렬

- 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 셀 정렬 방법으로 정렬하는 과정
 - (1) 원소 개수가 여덟 개이므로 매개변수 h 는 4에서 시작한다. $h=4$ 이므로 간격이 4만큼 떨어져 있는 원소들을 같은 부분집합으로 만들면 네 개의 부분집합이 만들어짐(같은 부분집합은 동일한 색으로 표시)
 - ① 첫 번째 부분집합 {69, 16}에 대해서 삽입 정렬을 수행하여 정렬
 - ② 두 번째 부분집합 {10, 8}에 대해서 삽입 정렬을 수행
 - ③ 세 번째 부분집합 {30, 31}에 대해서 삽입 정렬을 수행. $30 < 31$ 이므로 자리 이동은 이루어지지 않음.
 - ④ 네 번째 부분집합 {2, 22}에 대해서 삽입 정렬을 수행. $2 < 22$ 이므로 자리 이동은 이루어지지 않음

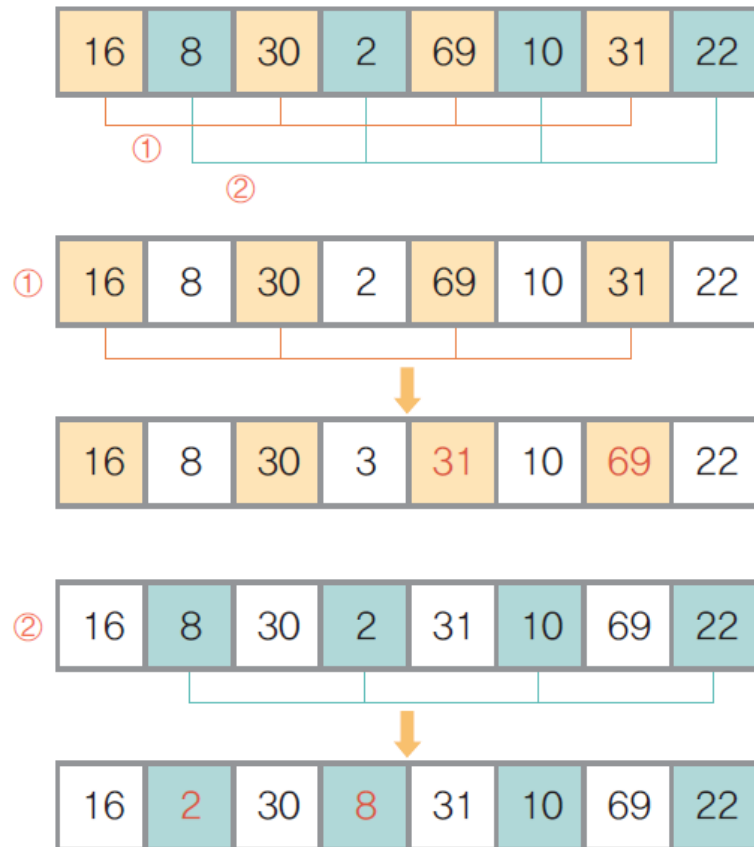


6. 셀 정렬



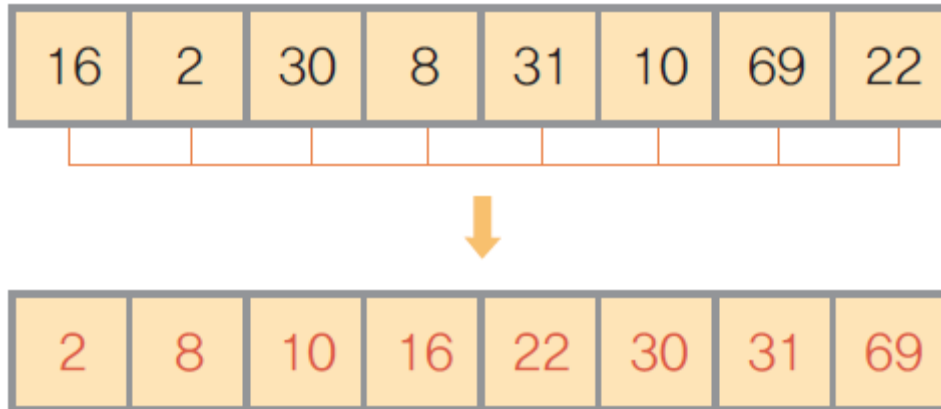
6. 셀 정렬

- (2) 이제 h 를 2로 변경하고 다시 셀 정렬을 시작. $h=2$ 이므로 간격이 2만큼 떨어진 원소들을 같은 부분집합으로 만들면 두 개의 부분집합이 만들짐
- ① 첫 번째 부분집합 {16, 30, 69, 31}에 대해 삽입 정렬을 수행하여 정렬
 - ② 두 번째 부분집합 {8, 2, 10, 22}에 대해 삽입 정렬을 수행하여 정렬



6. 셀 정렬

- (3) 이제 h 를 1로 변경하고 다시 셀 정렬을 시작. $h=1$ 이므로 간격이 1만큼 떨어져 있는 원소들을 같은 부분집합으로 만들면 한 개의 부분집합이 만들어짐. 즉 전체 원소에 대해서 삽입 정렬을 수행



❖ 셀 정렬 알고리즘

알고리즘 9-6 셀 정렬

```
shellSort(a[], n)
  interval ← n;
  while (interval ≥ 1) do {
    interval ← interval / 2;
    for (i ← 0; i < interval; i ← i + 1) do {
      intervalSort(a[], i, n, interval);
    }
  }
end shellSort()
```

알고리즘 9-7 셀 부분집합에서의 삽입 정렬

```
intervalSort(a[], begin, end, interval)
  for (i ← begin+interval; i ≤ end; i ← i+interval) do {
    item ← a[i];
    for (j ← i-interval; j ≥ begin and item < a[j]; j ← j-interval) do
      a[j+interval] ← a[j];
    a[j+interval] ← item;
  }
end intervalSort()
```



6. 셀 정렬

- 메모리 사용공간
 - n 개의 원소에 대하여 n 개의 메모리와 매개변수 h 에 대한 저장공간 사용
- 연산 시간
 - 비교횟수
 - 처음 원소의 상태에 상관없이 매개변수 h 에 의해 결정
 - 일반적인 시간 복잡도 : $O(n^{1.25})$
 - 셀 정렬은 삽입 정렬의 시간 복잡도 $O(n^2)$ 보다 개선된 정렬 방법



6. 셀 정렬

- 셀 정렬하기 프로그램 : [교재 525p](#)
- 실행 결과

```
명령 프롬프트
정렬할 원소 : 69 10 30 2 16 8 31 22

<<<<<<<<< 셀 정렬 수행 >>>>>>>>>

interval=4 >> 16 8 30 2 69 10 31 22

interval=2 >> 16 2 30 8 31 10 69 22

interval=1 >> 2 8 10 16 22 30 31 69
```



❖ 병합 정렬(merge sort)의 이해

- 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 방법
- 부분집합으로 분할(divide)하고, 각 부분집합에 대해서 정렬 작업을 완성(conquer)한 후에 정렬된 부분집합들을 다시 결합(combine)하는 분할 정복(divide and conquer) 기법 사용
- 병합 정렬 방법의 종류
 - 2-way 병합 : 2개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법
 - n-way 병합 : n개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법



7. 병합 정렬

■ 2-way 병합 정렬 과정

- 분할^{Divide} : 자료들을 두 개의 부분집합으로 분할한다.
- 정복^{Conquer} : 부분집합에 있는 원소를 정렬한다.
- 결합^{Combine} : 정렬된 부분집합들을 하나의 집합으로 정렬하여 결합한다.

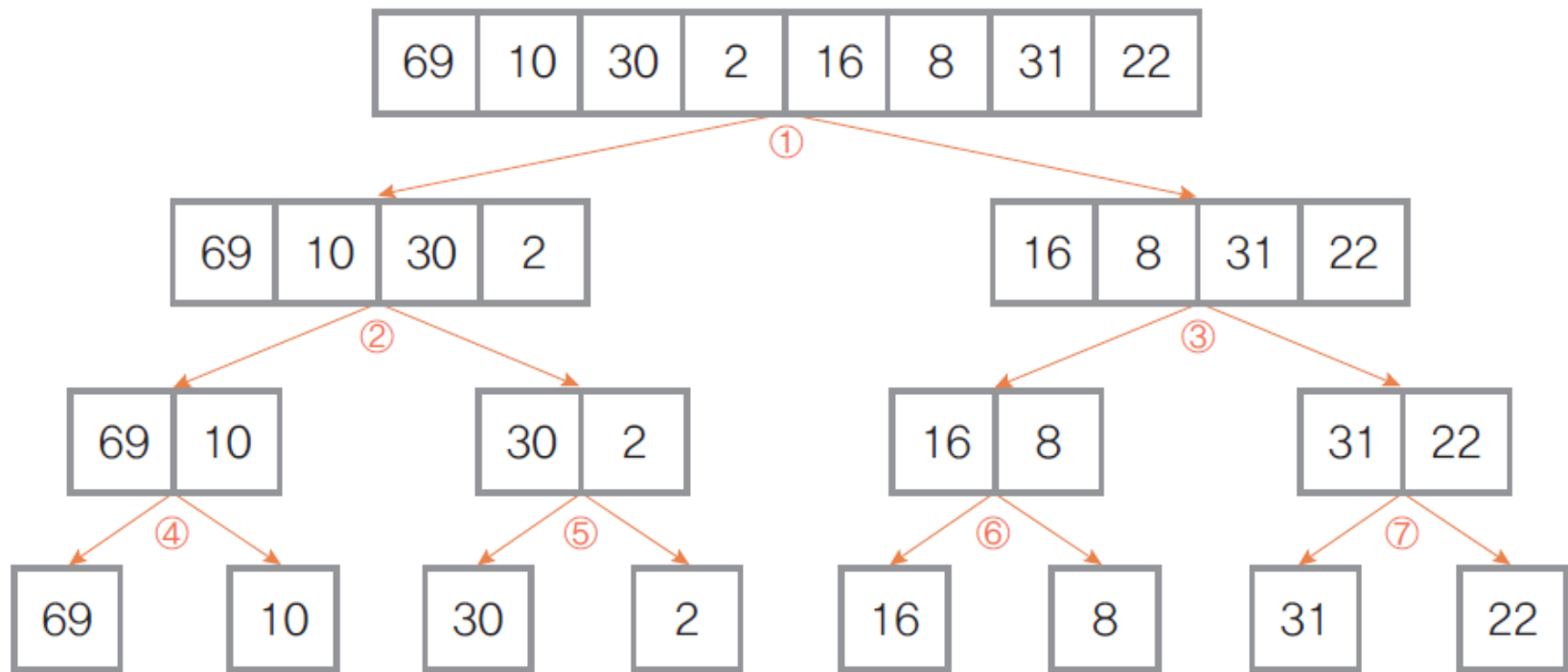
그림 9-5 2-way 병합 정렬 과정



7. 병합 정렬

- 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 병합 정렬 방법으로 정렬하는 과정

① 분할 단계 : 정렬할 전체 자료의 집합에 대해서 최소 원소의 부분집합이 될 때까지 분할 작업을 반복하여 한 개의 원소를 가진 부분집합 8개 만들

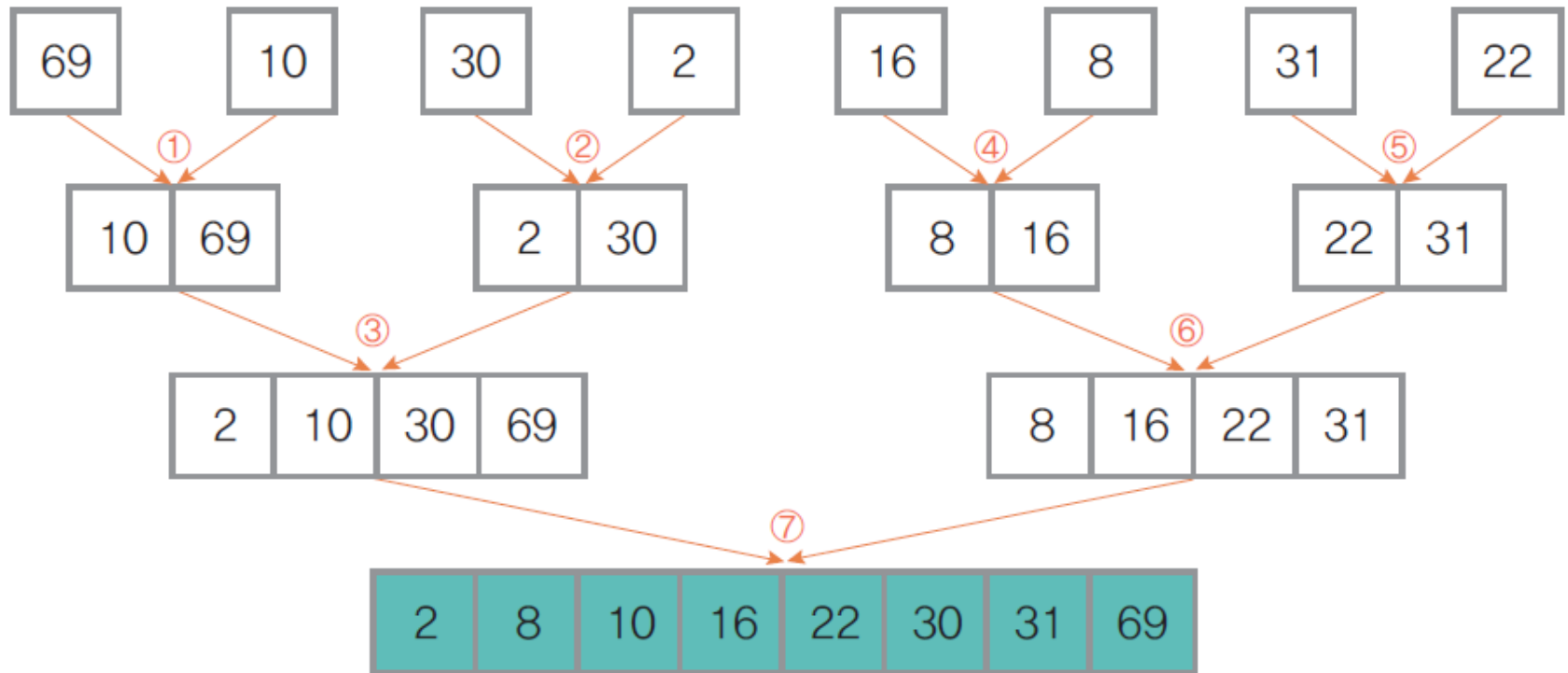


부분집합 여덟 개



7. 병합 정렬

- ② 정복과 결합 단계 : 부분집합 두 개를 정렬하여 하나로 결합. 전체 원소가
집합 하나로 묶일 때까지 반복



7. 병합 정렬

❖ 병합 정렬 알고리즘

알고리즘 9-8 병합 정렬

```
mergeSort(a[], m, n)
  if (a[m:n]의 원소 수 > 1) then {
    전체 집합을 부분집합 두 개로 분할;
    mergeSort(a[], m, middle);           // 왼쪽 부분집합을 다시 두 개로 분할
    mergeSort(a[], middle + 1, n);       // 오른쪽 부분집합을 다시 두 개로 분할
    merge(a[m:middle], a[middle + 1:n]); // 부분집합 두 개를 하나로 병합
  }
end mergeSort()
```



7. 병합 정렬

알고리즘 9-9 부분집합의 병합

```
merge(a[m:middle], a[middle + 1:n])
  i ← m;           // 첫 번째 부분집합의 첫째 원소 설정
  j ← middle + 1; // 두 번째 부분집합의 첫째 원소 설정
  k ← m;           // 병합 집합의 첫째 원소 설정

  while (i <= middle and j <= n) do {
    if (a[i] <= a[j]) then {
      sorted[k] ← a[i];
      i ← i + 1;
    }
    else {
      sorted[k] ← a[j];
      j ← j + 1;
    }
    k ← k + 1;
  }
  if (i > middle) then 두 번째 부분집합에 남아 있는 원소를 병합 집합 sorted에 복사;
  else 첫 번째 부분집합에 남아 있는 원소를 병합 집합 sorted에 복사;
end merge()
```



7. 병합 정렬

- 메모리 사용공간
 - 각 단계에서 새로 병합하여 만든 부분집합을 저장할 공간이 추가로 필요
 - 원소 n 개에 대해서 $(2 \times n)$ 개의 메모리 공간 사용
- 연산 시간
 - 분할 단계 : n 개의 원소를 두개로 분할하기 위해서 $\log_2 n$ 번의 단계 수행
 - 병합 단계 : 부분집합의 원소를 비교하면서 병합하는 단계에서 최대 n 번의 비교연산 수행
 - 전체 병합 정렬의 시간 복잡도 : $O(n \log_2 n)$



7. 병합 정렬

- 병합 정렬하기 프로그램 : [교재 530p](#)
- 실행 결과

```
C:\ 명령 프롬프트
정렬할 원소 >> 69 10 30 2 16 8 31 22

<<<<<<<<< 병합 정렬 수행 >>>>>>>>>>>>

병합 정렬 >> 10 69 30 2 16 8 31 22
병합 정렬 >> 10 69 2 30 16 8 31 22
병합 정렬 >> 2 10 30 69 16 8 31 22
병합 정렬 >> 2 10 30 69 8 16 31 22
병합 정렬 >> 2 10 30 69 8 16 22 31
병합 정렬 >> 2 10 30 69 8 16 22 31
병합 정렬 >> 2 8 10 16 22 30 31 69
```

❖ 기수 정렬(radix sort)의 이해

- 원소의 키값을 나타내는 기수를 이용한 정렬 방법
 - 정렬할 원소의 키 값에 해당하는 버킷(bucket)에 원소를 분배하였다가 버킷의 순서대로 원소를 꺼내는 방법을 반복하면서 정렬
 - 원소의 키를 표현하는 기수만큼의 버킷 사용
 - 예) 10진수로 표현된 키 값을 가진 원소들을 정렬할 때에는 0부터 9까지 10개의 버킷 사용
 - 키 값의 자리수 만큼 기수 정렬을 반복
 - 키 값의 일의 자리에 대해서 기수 정렬을 수행하고,
 - 다음 단계에서는 키 값의 십의 자리에 대해서,
 - 그리고 그 다음 단계에서는 백의 자리에 대해서 기수 정렬 수행
 - 한 단계가 끝날 때마다 버킷에 분배된 원소들을 버킷의 순서대로 꺼내서 다음 단계의 기수 정렬을 수행해야 하므로 큐를 사용하여 버킷을 만들

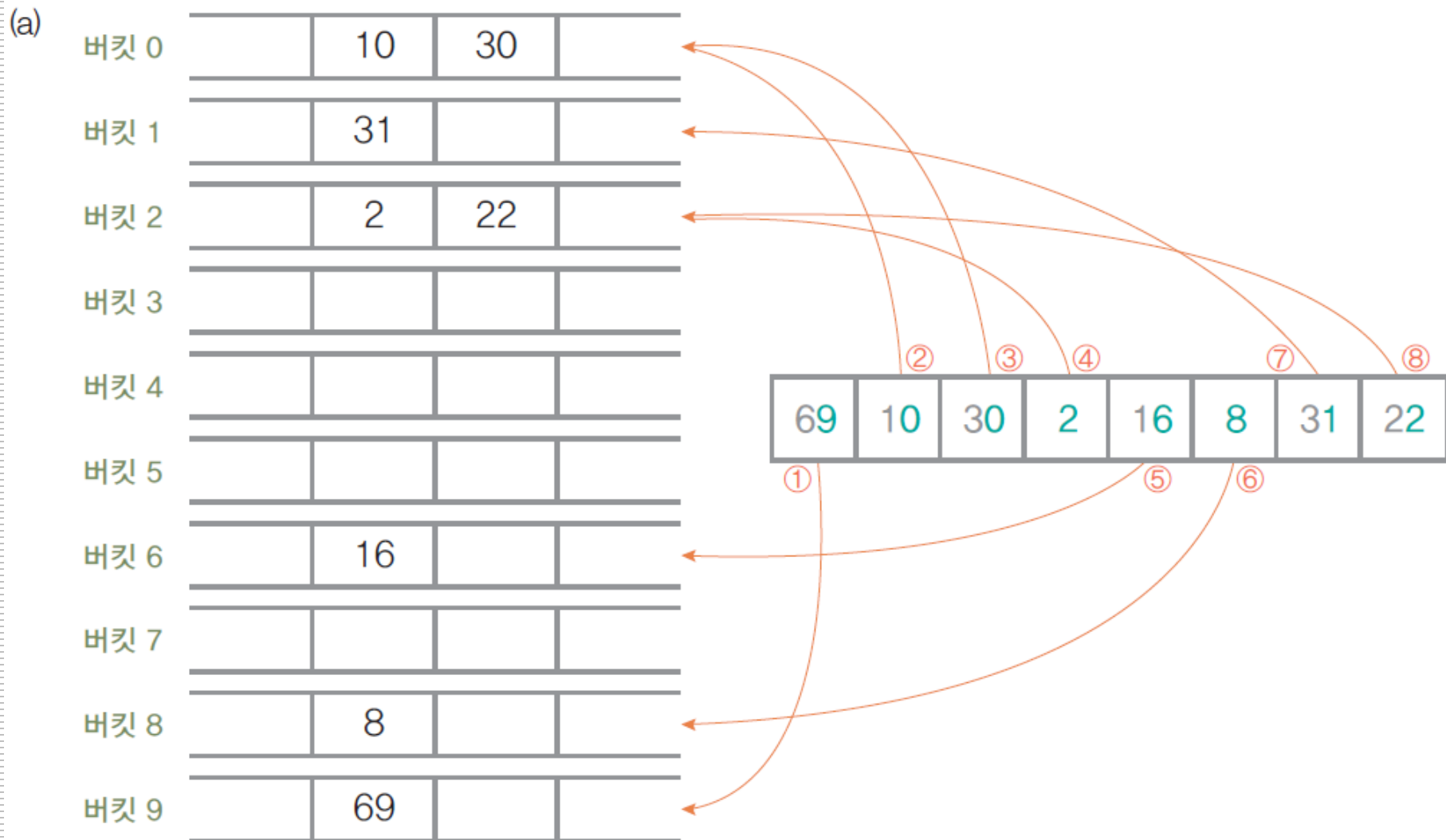


8. 기수 정렬

- {69, 10, 30, 2, 16, 8, 31, 22}의 자료를 기수 정렬 방법으로 정렬
 - 키값이 10진수이므로 0부터 9까지 열 개의 버킷을 사용하고, 키값의 최대 자릿수가 두 자리이므로 기수 정렬을 두 번 반복 수행
- ① 키값의 1의 자리에 대해서 기수 정렬을 수행
 - (a) 정렬할 원소의 키값의 1의 자리에 맞춰 버킷에 분배
 - (b) 버킷에 분배되어 있는 원소들을 버킷 0부터 버킷 9까지 순서대로 꺼내고, 꺼낸 순서대로 저장



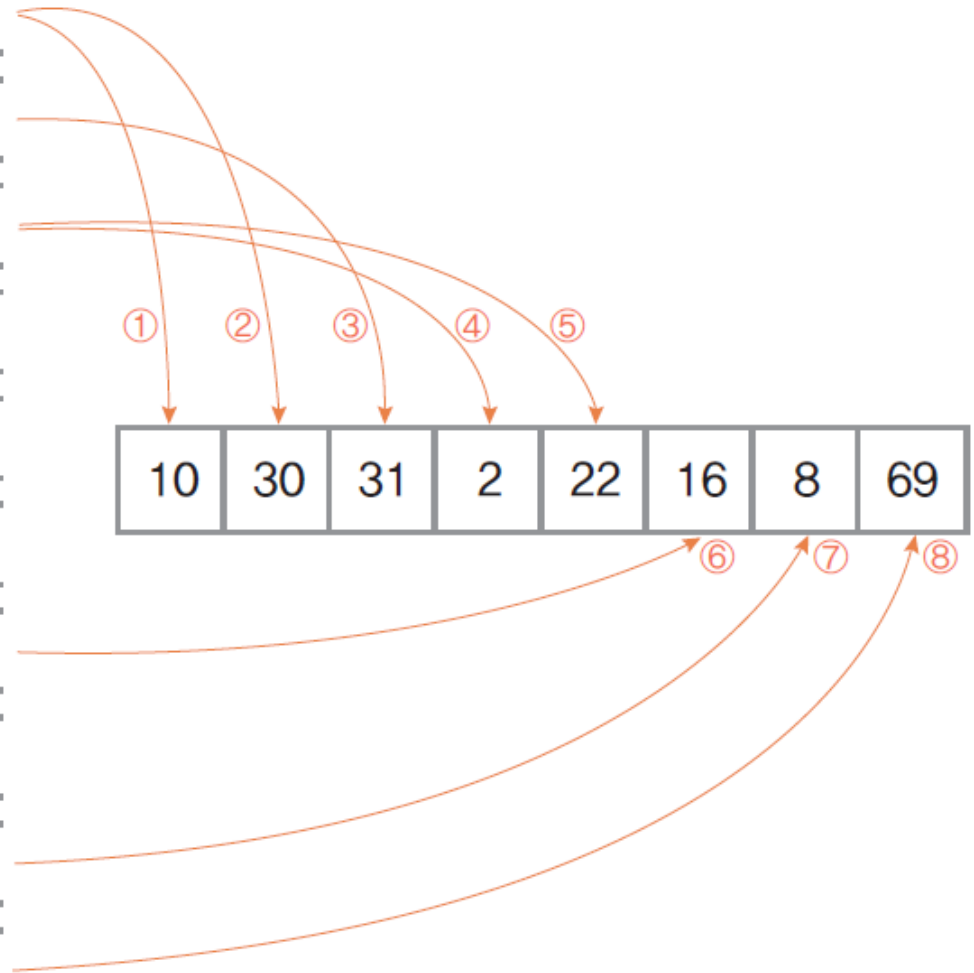
8. 기수 정렬



8. 기수 정렬

(b)

버킷 0	10	30	
버킷 1	31		
버킷 2	2	22	
버킷 3			
버킷 4			
버킷 5			
버킷 6	16		
버킷 7			
버킷 8	8		
버킷 9	69		

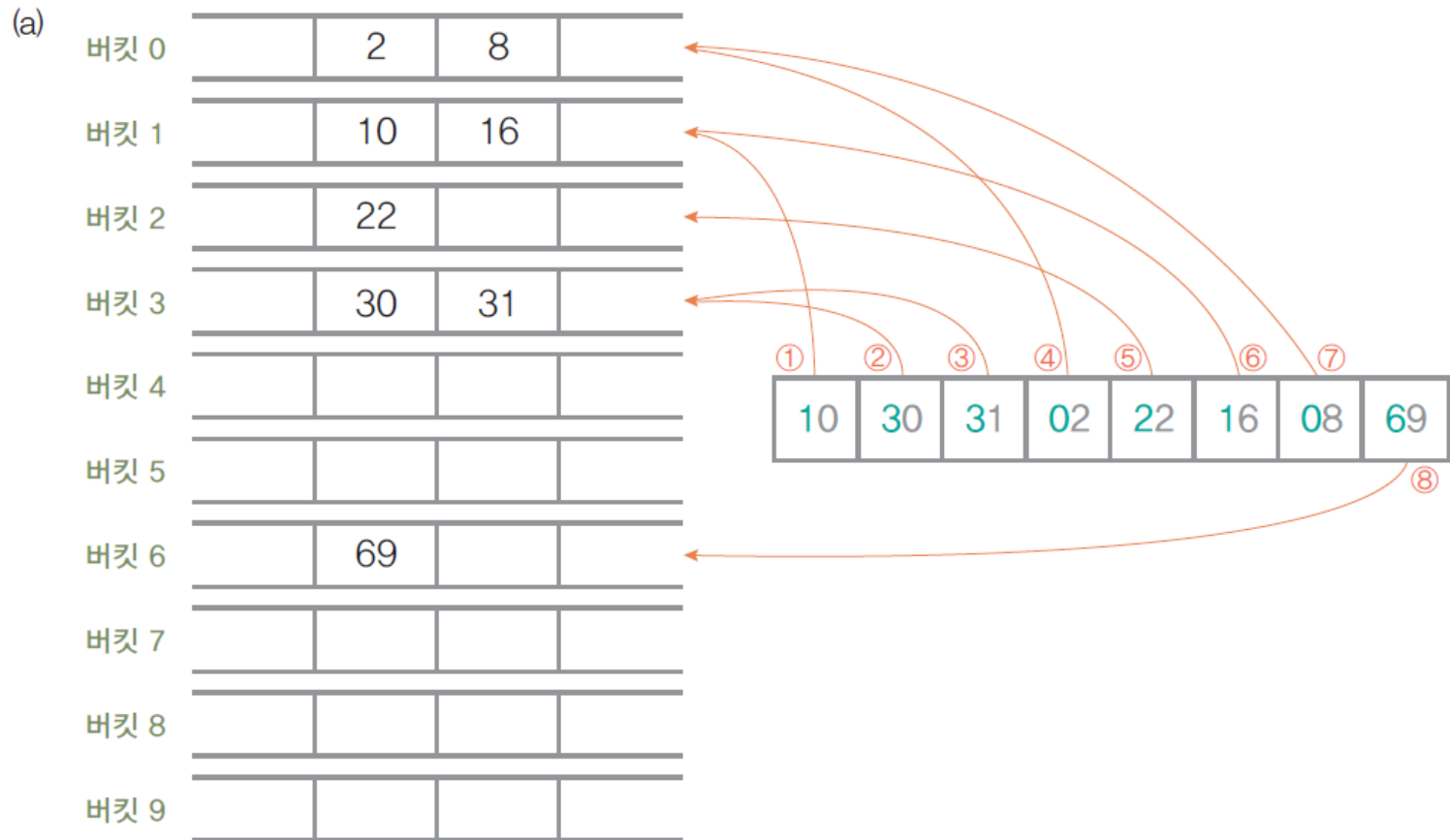


8. 기수 정렬

② 키값의 10의 자리에 대해서 기수 정렬을 수행.

(a) 정렬할 원소의 10의 자리에 대해서 버킷에 분배.

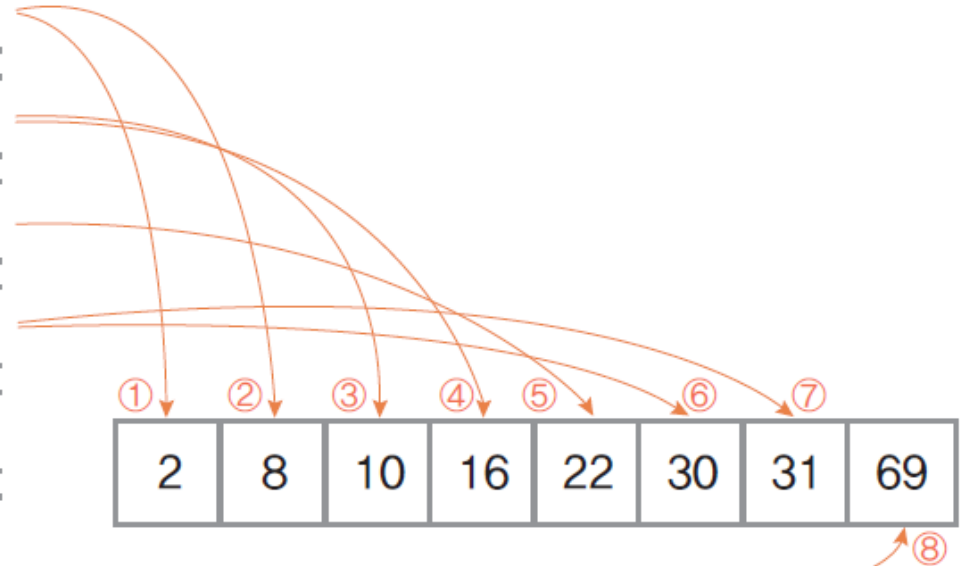
(b) 버킷에 분배되어 있는 원소들을 버킷 0부터 버킷 9까지 순서대로 꺼내고, 꺼낸 순서대로 저장하면 전체 원소에 대한 정렬이 완성



8. 기수 정렬

(b)

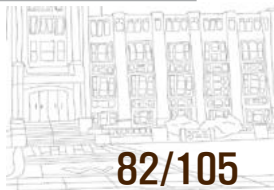
버킷 0		2	8	
버킷 1		10	16	
버킷 2		22		
버킷 3		30	31	
버킷 4				
버킷 5				
버킷 6		69		
버킷 7				
버킷 8				
버킷 9				



❖ 기수 정렬 알고리즘

알고리즘 9-10 기수 정렬

```
radixSort(a[], n)
  for (k ← 1; k ≤ n; k ← k + 1) do {
    for (i ← 0; i < n; i ← i + 1) do {
      enqueue(Q[k], a[i]);      // k번째 자릿수 값에 따라서 해당 버킷 큐에 저장
    }
    p ← - 1;
    for (i ← 0; i ≤ 9; i ≤ i + 1) do {
      while (Q[i] ≠ NULL) do {
        p ← p + 1;
        a[p] ← dequeue(Q[i]);  // 버킷 큐 순서대로 원소를 꺼내어 저장
      }
    }
  }
end radixSort()
```

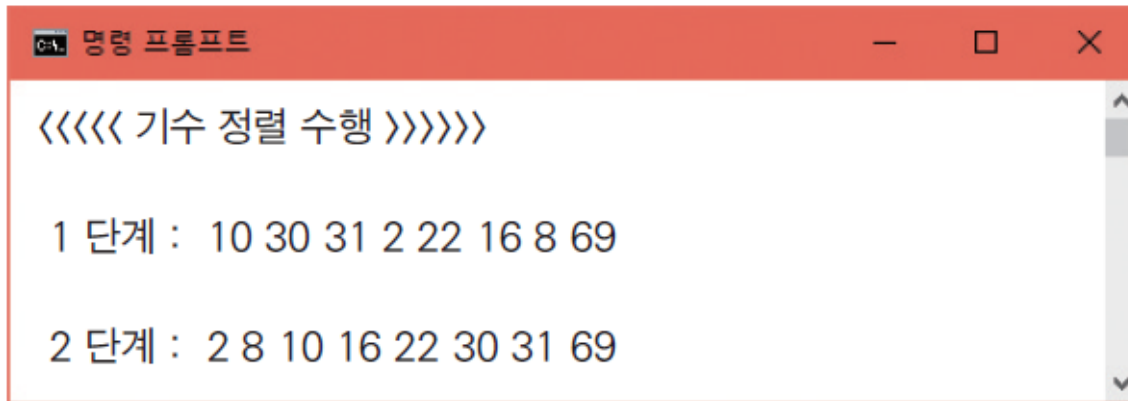


- 메모리 사용공간
 - 원소 n 개에 대해서 n 개의 메모리 공간 사용
 - 기수 r 에 따라 버킷 공간이 추가로 필요
- 연산 시간
 - 연산 시간은 정렬할 원소의 수 n 과 키 값의 자릿수 d 와 버킷의 수를 결정하는 기수 r 에 따라서 달라진다.
 - 정렬할 원소 n 개를 r 개의 버킷에 분배하는 작업 : $(n+r)$
 - 이 작업을 자릿수 d 만큼 반복
 - 수행할 전체 작업 : $d(n+r)$
 - 시간 복잡도 : $O(d(n+r))$



8. 기수 정렬

- 기수 정렬하기 프로그램 : [교재 535p](#)
- 실행 결과



```
C:\> 명령 프롬프트

<<<<< 기수 정렬 수행 >>>>>

1 단계 : 10 30 31 2 22 16 8 69

2 단계 : 2 8 10 16 22 30 31 69
```



❖ 힙 정렬 heap sort의 이해

- 7장의 힙 자료구조를 이용한 정렬 방법
- 힙에서는 항상 가장 큰 원소가 루트 노드가 되고 삭제 연산을 수행하면 항상 루트 노드의 원소를 삭제하여 반환
 - 최대 힙에 대해서 원소의 개수만큼 삭제 연산을 수행하여 내림차순으로 정렬 수행
 - 최소 힙에 대해서 원소의 개수만큼 삭제 연산을 수행하여 오름차순으로 정렬 수행



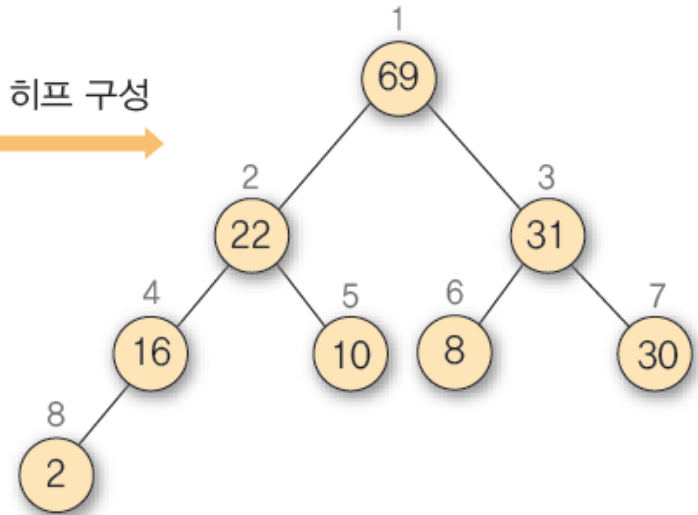
9. 힙프 정렬

- 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 힙프 정렬 방법으로 정렬하는 과정

(1) 초기 상태 : 정렬할 원소에 대해 7장에서 설명한 삽입 연산을 이용해 최대 힙프를 구성

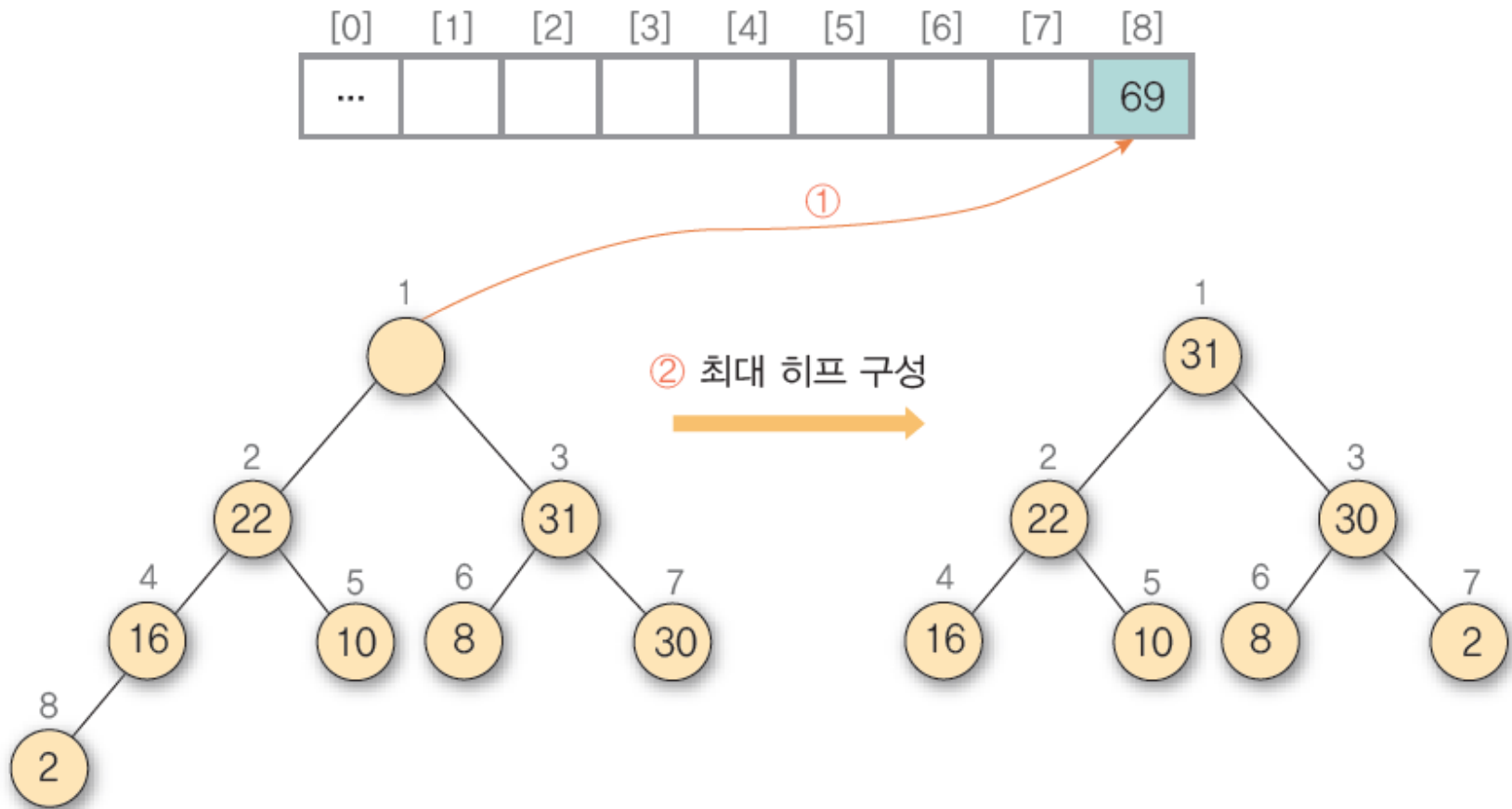
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
...	69	10	30	2	16	8	31	22

최대 힙프 구성



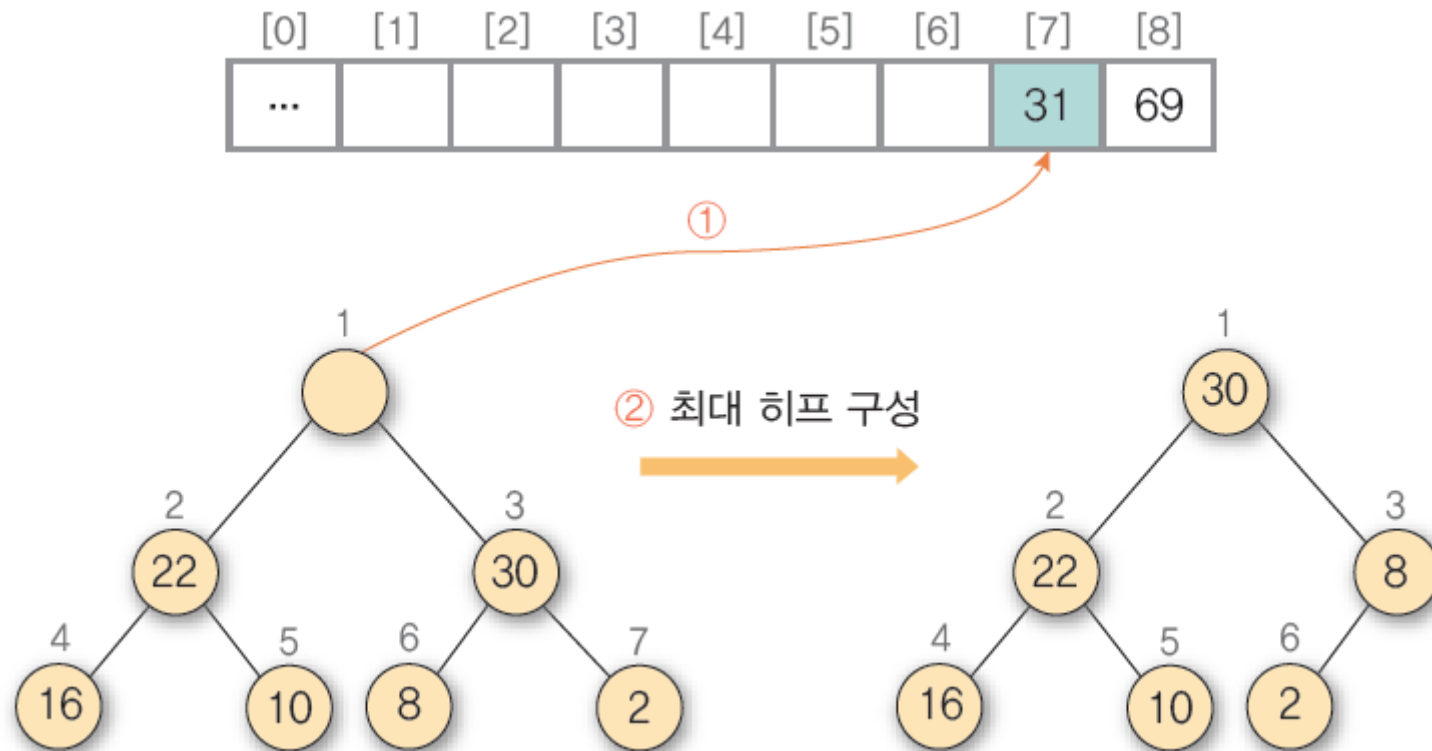
9. 힙프 정렬

- (2) ① 힙프에 삭제 연산을 수행하여 루트 노드의 원소 69를 구한 후 배열의 마지막 자리에 저장 ② 나머지 힙프를 최대 힙프로 재구성. 원소를 삭제하고 힙프를 재구성하는 작업은 7장의 [알고리즘7-12]에 따라 수행



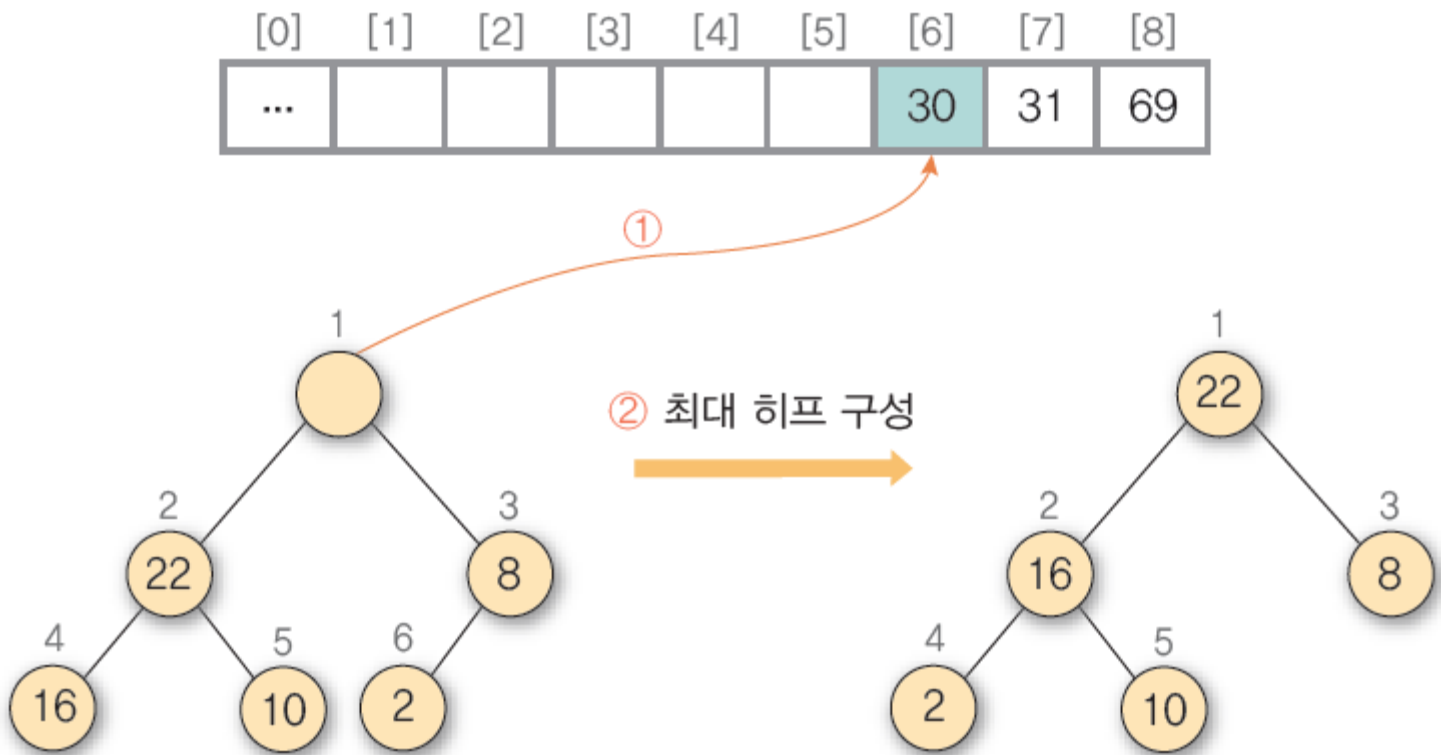
9. 힙프 정렬

- (3) ① 힙프에 삭제 연산을 수행하여 루트 노드의 원소 31을 구한 후 배열의 비어 있는 마지막 자리에 저장 ② 나머지 힙프를 최대 힙프로 재구성



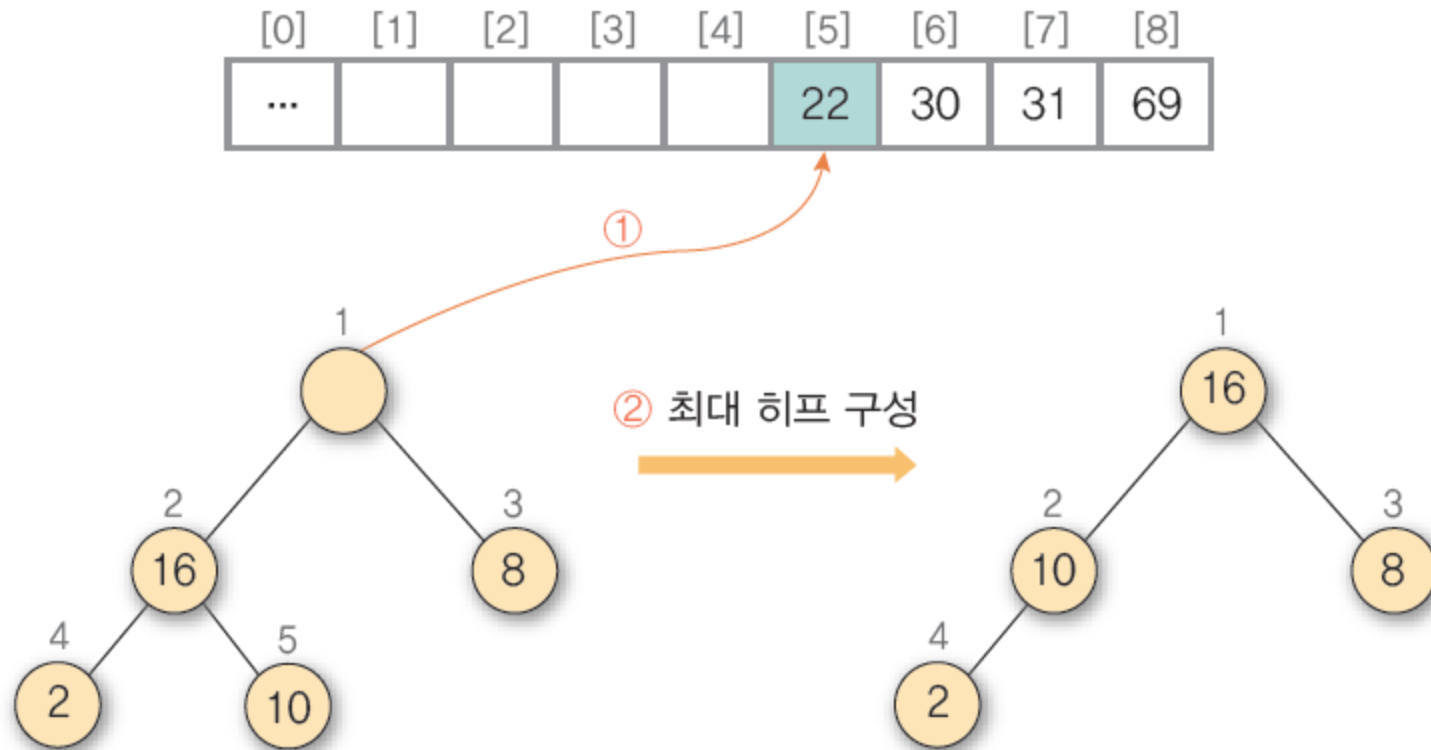
9. 힙프 정렬

- (4) ① 힙프에 삭제 연산을 수행하여 루트 노드의 원소 30을 구한 후 배열의
비어 있는 마지막 자리에 저장 ② 나머지 힙프를 최대 힙프로 재구성



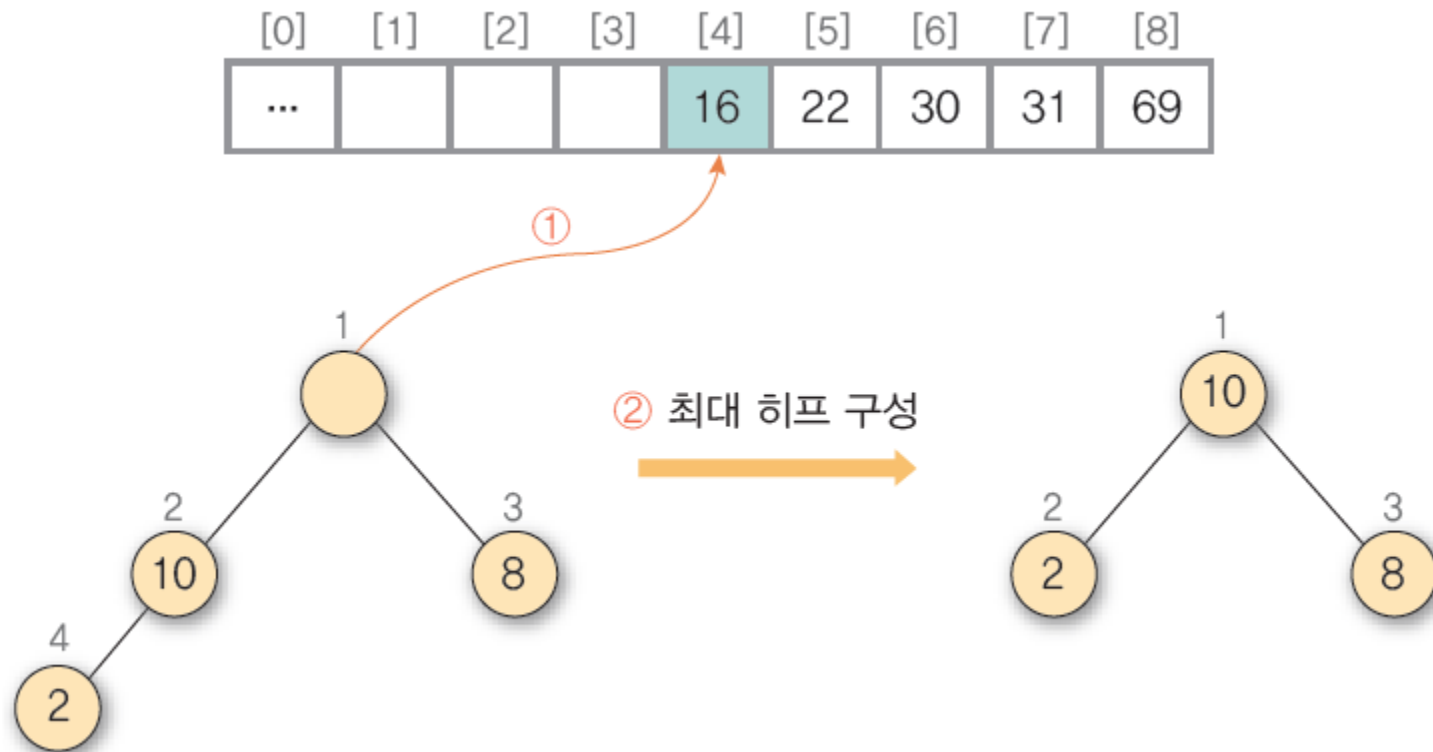
9. 힙 정렬

- (5) ① 힙에 삭제 연산을 수행하여 루트 노드의 원소 22를 구한 후 배열의
비어 있는 마지막 자리에 저장 ② 나머지 힙을 최대 힙으로 재구성



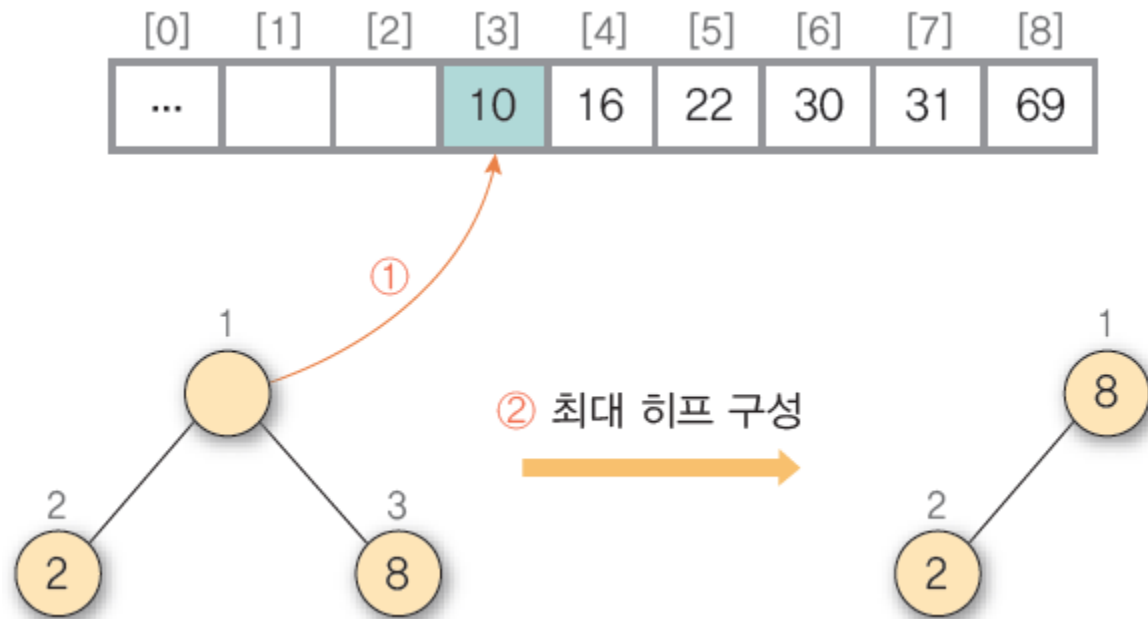
9. 힙 정렬

- (6) ① 힙에 삭제 연산을 수행하여 루트 노드의 원소 16을 구한 후 배열의
비어 있는 마지막 자리에 저장 ② 나머지 힙을 최대 힙으로 재구성



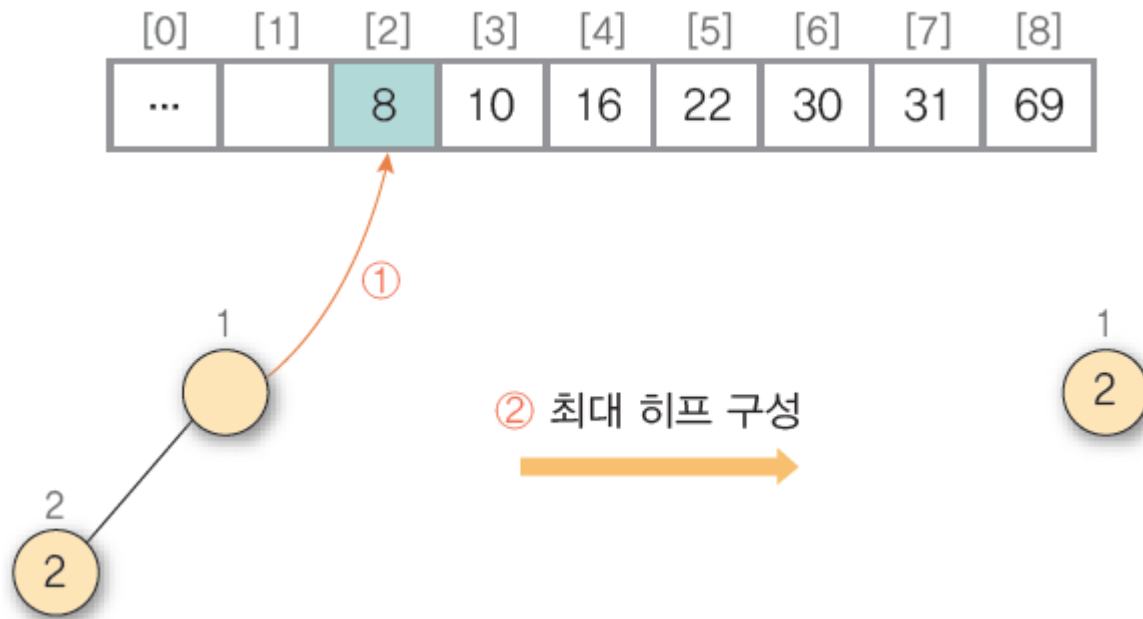
9. 힙 정렬

- (7) ① 힙에 삭제 연산을 수행하여 루트 노드의 원소 10을 구한 후 배열의
비어 있는 마지막 자리에 저장 ② 나머지 힙을 최대 힙으로 재구성



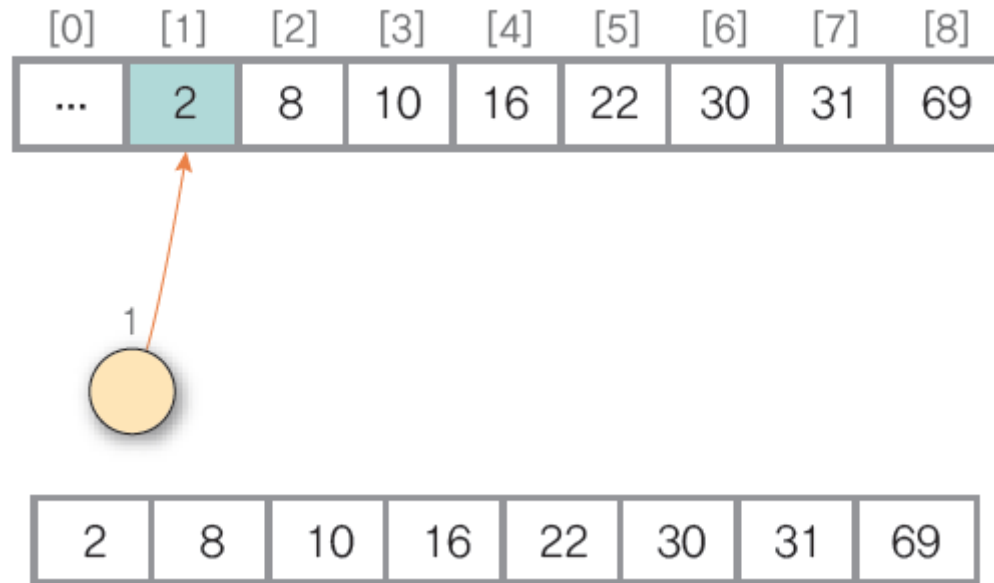
9. 힙프 정렬

- (8) ① 힙프에 삭제 연산을 수행하여 루트 노드의 원소 8을 구한 후 배열의
비어 있는 마지막 자리에 저장 ② 나머지 힙프를 최대 힙프로 재구성



9. 힙프 정렬

- (9) 힙프에 삭제 연산을 수행하여 루트 노드의 원소 2를 구한 후 배열의 비어 있는 마지막 자리에 저장. 나머지 힙프를 최대 힙프로 재구성해야 하는데 공백 힙프가 되었으므로 힙프 정렬을 종료



❖ 힙 정렬 알고리즘

알고리즘 9-11 힙 정렬

```
heapSort(a[])
  n ← a.length - 1;
  for (i ← n / 2; i ≥ 1; i ← i - 1) do // 배열 a[]를 힙으로 변환
    makeHeap(a, i, n);
  for (i ← n - 1; i ≥ 1; i ← i - 1) do {
    temp ← a[1]; // 힙의 루트 노드 원소를
    a[1] ← a[i + 1]; // 배열의 비어 있는
    a[i + 1] ← temp; // 마지막 자리에 저장
    makeHeap(a, 1, i);
  }
end heapSort()
```



9. 힙 정렬

- 배열을 힙 구조로 재구성하는 makeHeap() 연산에 대한 알고리즘

알고리즘 9-12 힙 재구성

```
makeHeap(a[], h, m)
  for (j ← 2 * h; j ≤ m; j ← 2 * j) do {
    if (j < m) then
      if (a[j] < a[j + 1]) then j ← j + 1;
    if (a[h] ≥ a[j]) then exit;
    else a[j / 2] ← a[j];
  }
  a[j / 2] ← a[h];
end makeHeap()
```



9. 힙 정렬

- 메모리 사용공간
 - 원소 n 개에 대해서 n 개의 메모리 공간 사용
 - 크기 n 의 힙 저장 공간
- 연산 시간
 - 힙 재구성 연산 시간
 - n 개의 노드에 대해서 완전 이진 트리는 $\log_2(n+1)$ 의 레벨을 가지므로 완전 이진 트리를 힙으로 구성하는 평균시간은 $O(\log_2 n)$
 - n 개의 노드에 대해서 n 번의 힙 재구성 작업 수행
 - 평균 시간 복잡도 : $O(n \log_2 n)$



❖ 트리 정렬^{tree sort}의 이해

- 7장의 이진 탐색 트리를 이용하여 정렬하는 방법
- 트리 정렬 수행 방법
 - 정렬할 원소들을 이진 탐색 트리로 구성
 - 이진 탐색 트리를 중위 우선 순회 함
 - 중위 순회 경로가 오름차순 정렬이 됨



10. 트리 정렬

- 정렬되지 않은 {69, 10, 30, 2, 16, 8, 31, 22}의 자료들을 트리 정렬 방법으로 정렬하는 과정

- 정렬할 원소 여덟 개를 차례대로 삽입하여 이진 탐색 트리를 구성
- 이진 탐색 트리를 중위 순회 방법으로 순회하면서 원소를 저장



❖ 트리 정렬 알고리즘

- 이진 탐색 트리에서의 삽입 연산 `insert()`와 중위 순회 연산 `inorder()`는 7장 알고리즘을 응용하여 사용함

알고리즘 9-13 트리 정렬

```
treeSort(a[], n)
  for (i ← 0; i < n; i ← i + 1) do
    insert(BST, a[i]);      // 이진 탐색 트리의 삽입 연산
    inorder(BST);           // 중위 순회 연산
end treeSort()
```



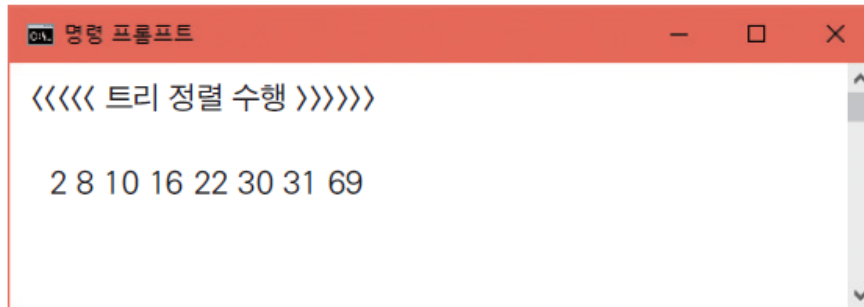
10. 트리 정렬

- 메모리 사용공간
 - 원소 n 개에 대해서 n 개의 메모리 공간 사용
 - 크기 n 의 이진 탐색 트리 저장 공간
- 연산 시간
 - 노드 한 개에 대한 이진 탐색 트리 구성 시간 : $O(\log_2 n)$
 - n 개의 노드에 대한 시간 복잡도 : $O(n \log_2 n)$



10. 트리 정렬

- 트리 정렬하기 프로그램 : [교재 544p](#)
- 실행 결과



```
명령 프롬프트
<<<< 트리 정렬 수행 >>>>

2 8 10 16 22 30 31 69
```



정렬 알고리즘의 비교

알고리즘	최선	평균	최악
삽입 정렬	$O(n)$	$O(n^2)$	$O(n^2)$
선택 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$
버블 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$
셸 정렬	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$
퀵 정렬	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
힙 정렬	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
합병 정렬	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
기수 정렬	$O(dn)$	$O(dn)$	$O(dn)$



정렬 알고리즘의 실험 예 (정수 60,000개)

알고리즘	실행 시간(단위:sec)
삽입 정렬	7.438
선택 정렬	10.842
버블 정렬	22.894
셸 정렬	0.056
히프 정렬	0.034
합병 정렬	0.026
퀵 정렬	0.014



Thank You

