ALGORITHM
알고리즘

Lecture 05

정렬 (3)

컴퓨터과학과 | 이관용 교수

학습목차

1 | 힙 정렬

2 | 계수 정렬

3 | 기수 정렬

4 | 버킷 정렬

01. 힙 정렬

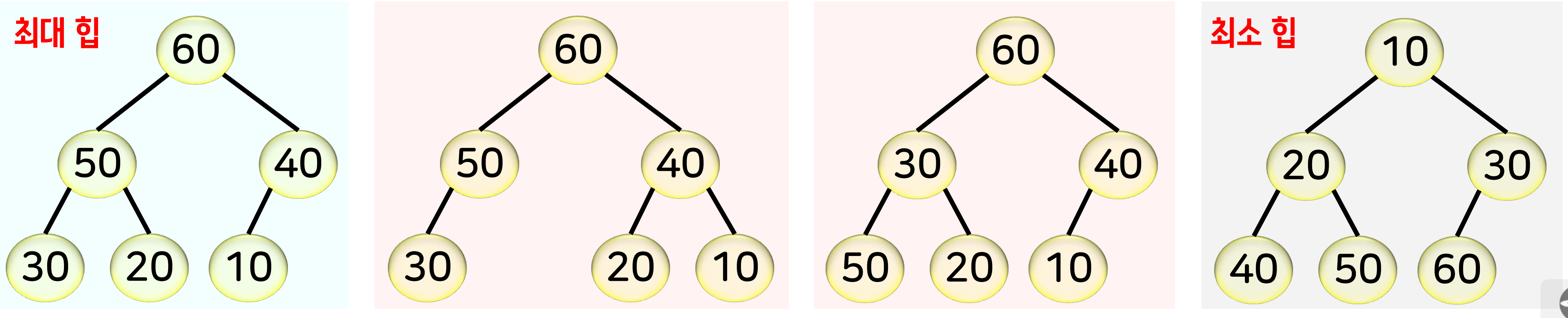
힙 자료구조

힙 정렬 → '힙' 자료구조의 장점을 활용한 정렬

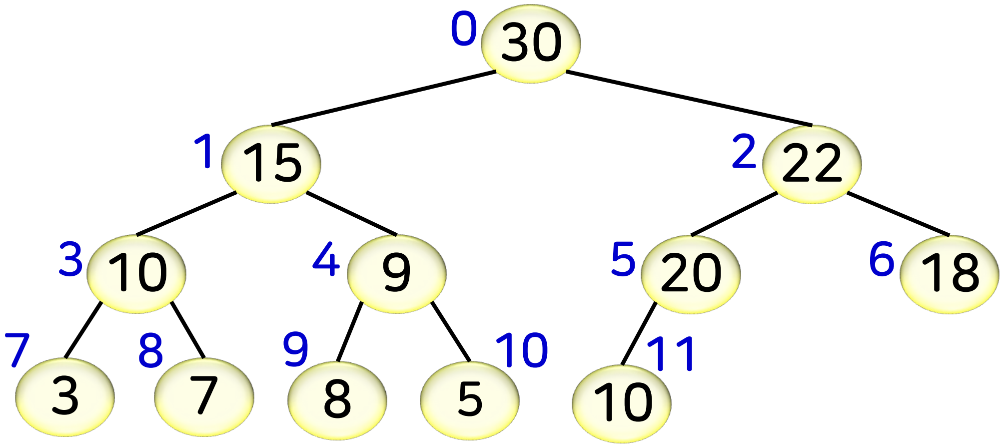
- 임의의 값 삽입과 최댓값 삭제가 쉬움

(최대) 힙 heap vs 최소 힙

- 완전 이진 트리 complete binary tree
- 각 노드의 값은 자신의 자식 노드의 값보다 크거나 같아야 함



힙의 구현



일차원 배열

0	1	2	3	4	5	6	7	8	9	10	11
30	15	22	10	9	20	18	3	7	8	5	10

i

$2i+1$

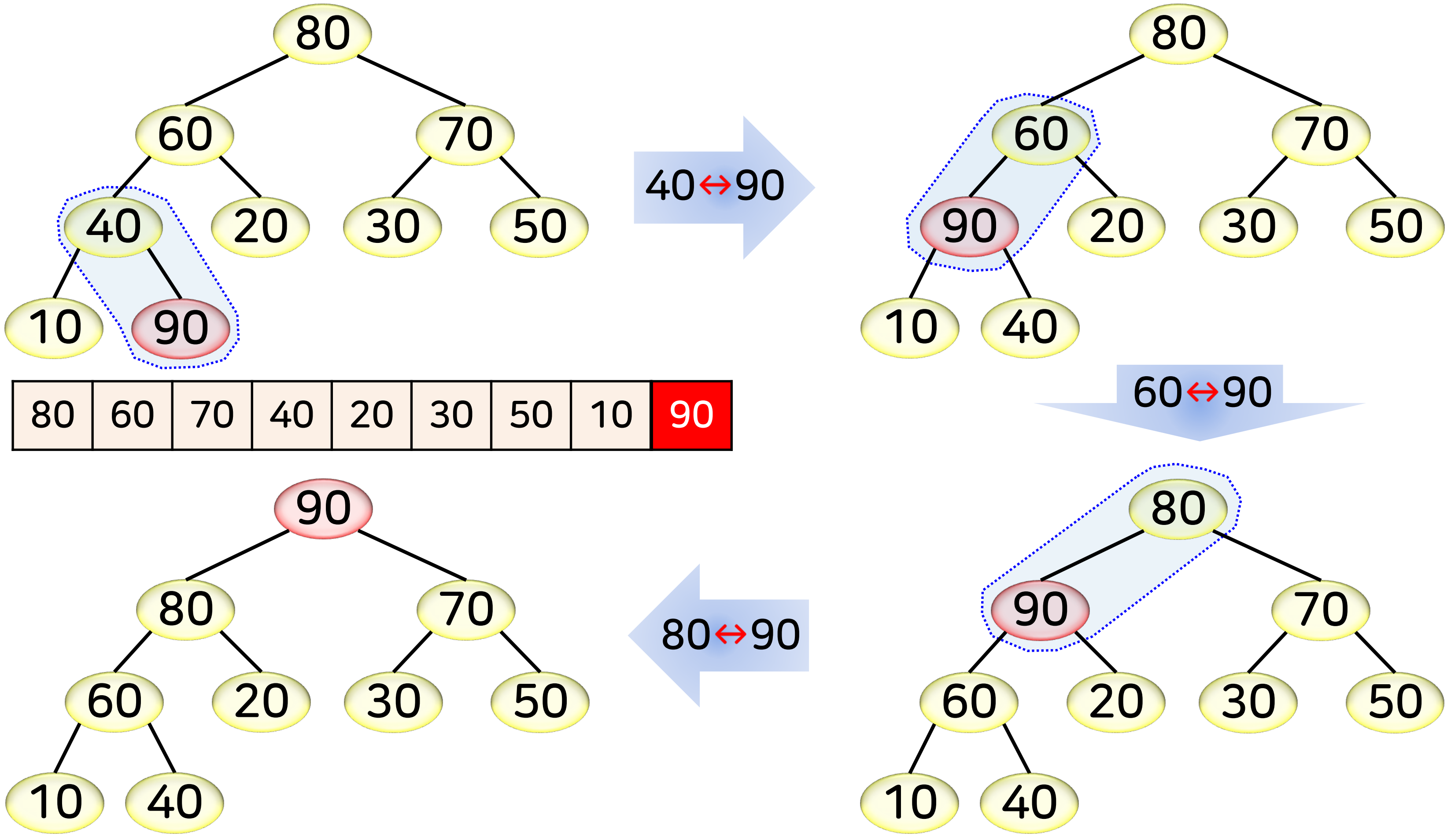
$2i+2$

j

$\left\lceil \frac{j}{2} \right\rceil - 1$ 또는 $\left\lfloor \frac{j-1}{2} \right\rfloor$

힙의 장점

임의의 값의 삽입

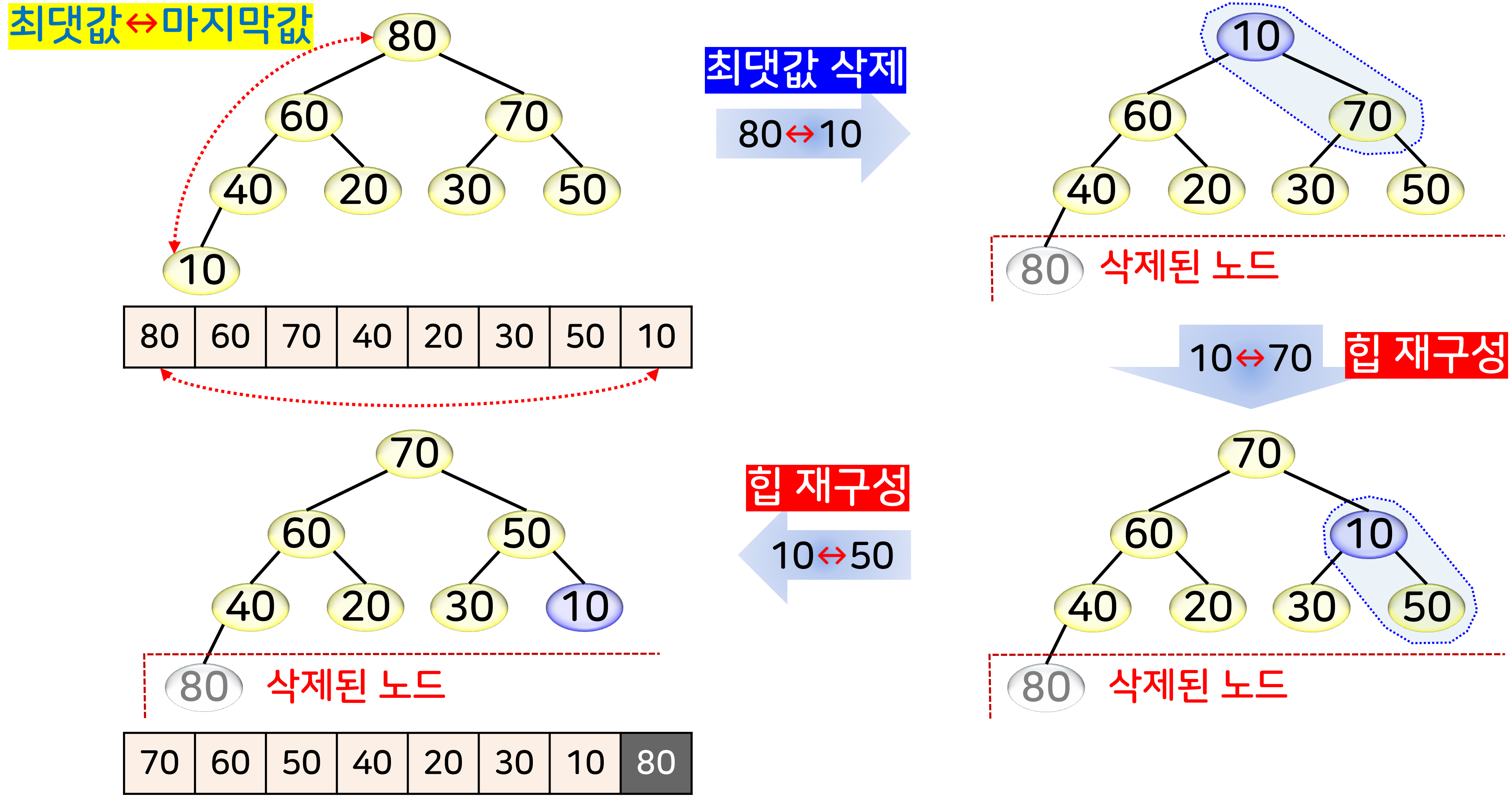


80	60	70	40	20	30	50	10	90

90	80	70	60	20	30	50	10	40

힙의 장점

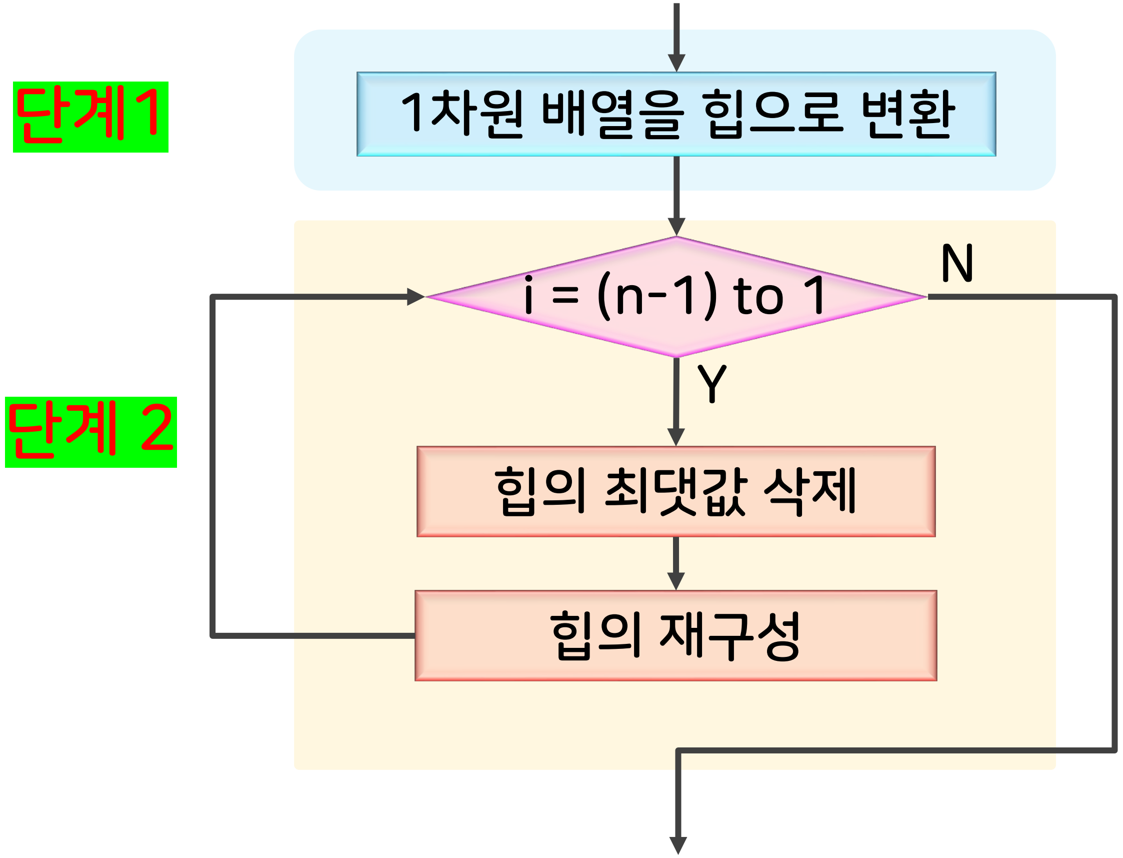
최댓값 삭제


최댓값↔마지막값
80 60 70 40 20 30 50 10
최댓값 삭제
80↔10
삭제된 노드
10↔70
힙 재구성
힙 재구성
10↔50
삭제된 노드
삭제된 노드
70 60 50 40 20 30 10 80

힙 정렬

힙 구조의 장점을 활용한 정렬 방식

힙 정렬의 처리 과정

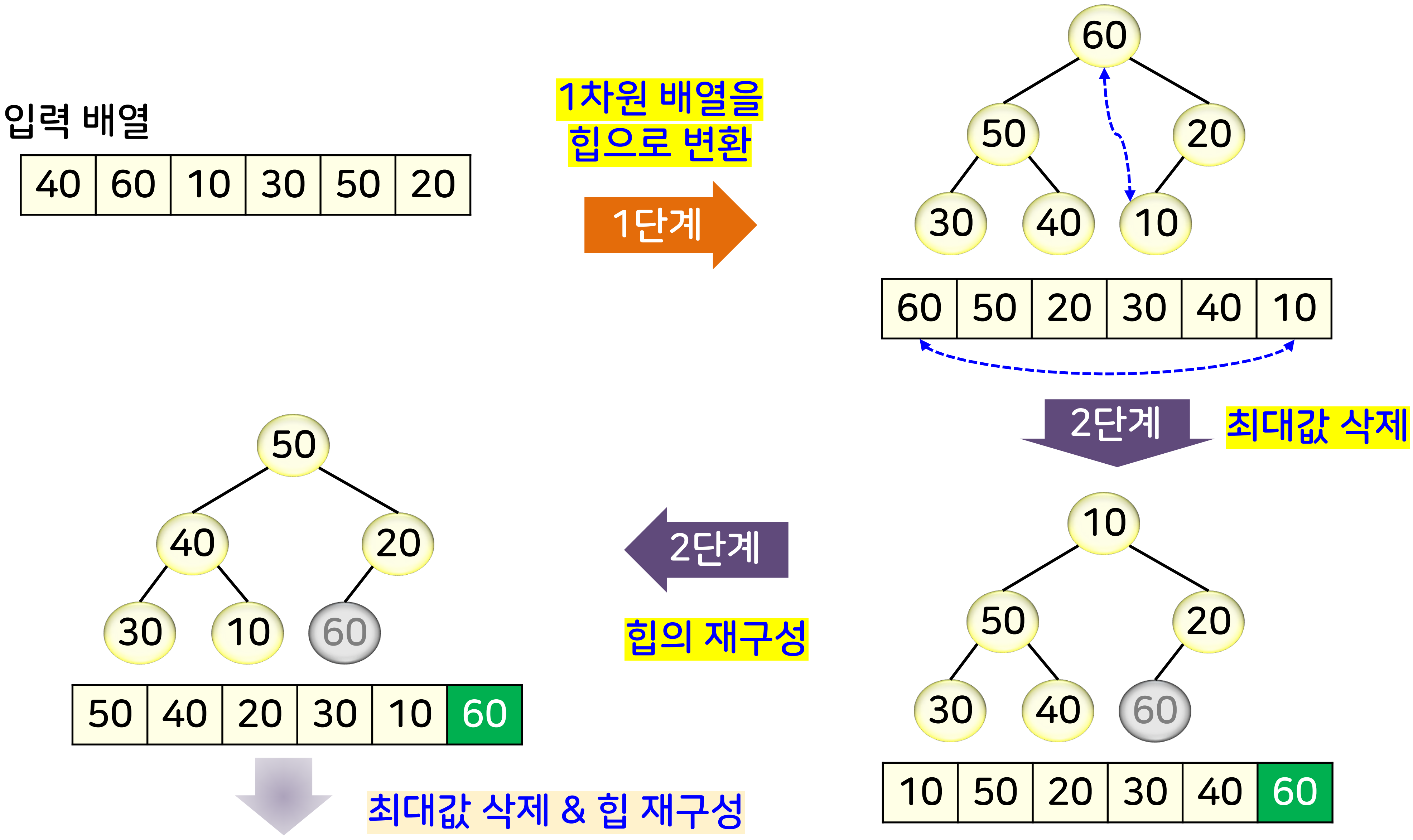


힙 정렬 알고리즘

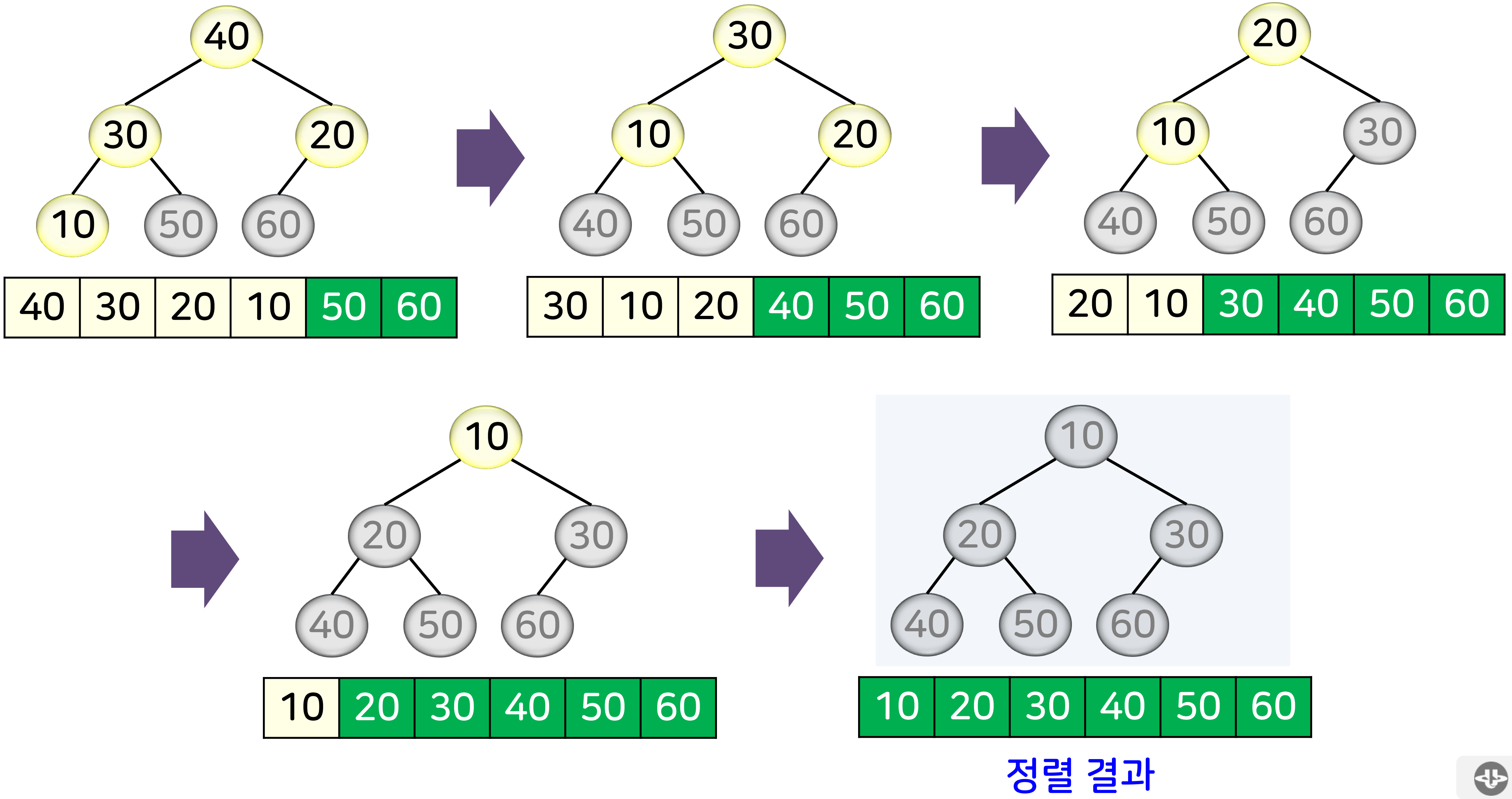

```
HeapSort (A[ ], n)
{
  // --- 단계1 --(새 노드의 삽입)-------
  for (i=0; i<n; i++) {
    par = ⌈ i/2 ⌉-1;
    while ( par>=0 && A[par]<A[i] ) {
      A[par]과 A[i]의 교환;
      i = par;
      par = ⌊ (i-1)/2 ⌋;
    }
  }
  // ------------- 단계2 ----------------
  return (A);
}
```

```
for (i=n-1; i>0; i-- ) {   // ----- 단계2 -----
  최댓값 A[0]와 마지막노드 A[i]의 교환;
  cur = 0;  lch = 1;  rch = 2;
  do {       // ------- 힙의 재구성 ---------
    if ( rch < i && A[lch] < A[rch] ) lch = rch;
    if ( A[lch] > A[cur] ) {
      A[cur]과 A[lch]의 교환;
      cur = lch;
      lch = cur*2 + 1;
      rch = cur*2 + 2;
    }
    else lch = i;
  } while ( lch < i )
}
```

힙 정렬의 처리 과정


입력 배열
40 60 10 30 50 20
1차원 배열을
힙으로 변환
1단계
60 50 20 30 40 10
60 50 20 30 40 10
2단계
최대값 삭제
10 50 20 30 40 60
10 50 20 30 40 60
2단계
힙의 재구성
50 40 20 30 10 60
50 40 20 30 10 60
최대값 삭제 & 힙 재구성

힙 정렬의 처리 과정


40 30 20 10 50 60
30 10 20 40 50 60
20 10 30 40 50 60
10 20 30 40 50 60
10 20 30 40 50 60
정렬 결과

초기 힙 구축

1차원 입력 배열을 힙으로 변환하는 것

두 가지 접근 방법

- **방법 1**
  - ✓ 주어진 입력 배열의 각 원소를 힙에 삽입하는 과정을 반복
- **방법 2**
  - ✓ 주어진 입력 배열을 우선 완전 이진 트리로 만든 후,
    각 노드에 대해 아래에서 위로 그리고 오른쪽에서 왼쪽으로 진행하면서
    해당 노드의 아랫부분이 힙의 조건을 만족할 수 있도록
    트리를 따라 내려가면서 자신의 자손 노드들과의 위치 교환을 계속해 나가는 방법

초기 힙 구축_방법 1

60	20	70	10	80	30	50	40

입력 배열

60

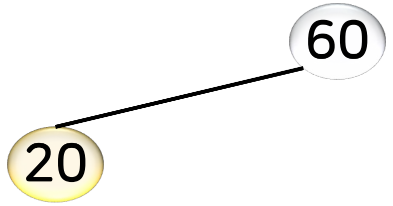
0
60

결과 배열

초기 힙 구축_방법 1

60	20	70	10	80	30	50	40

입력 배열



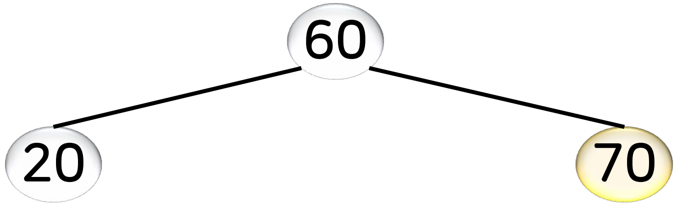
0	1
60	20

결과 배열

초기 힙 구축_방법 1

60	20	70	10	80	30	50	40

입력 배열



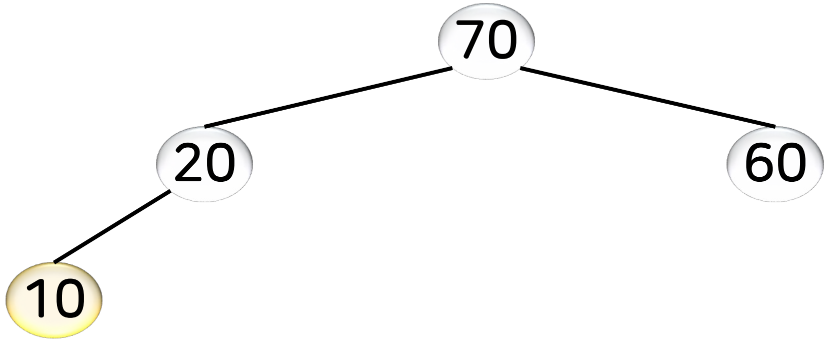
0	1	2
60	20	70

결과 배열

초기 힙 구축_방법 1

60	20	70	10	80	30	50	40

입력 배열



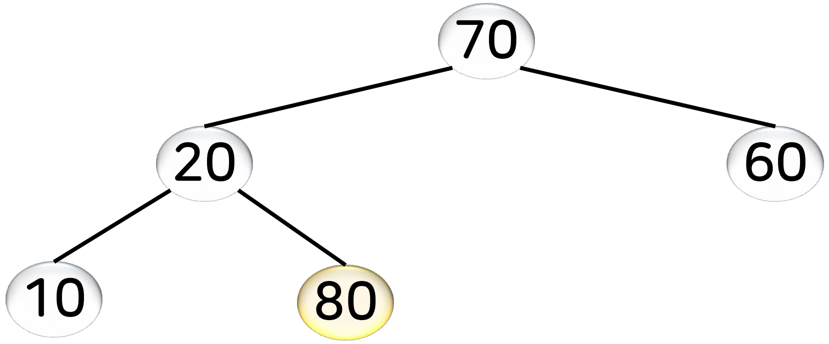
0	1	2	3
70	20	60	10

결과 배열

초기 힙 구축_방법 1

60	20	70	10	80	30	50	40

입력 배열



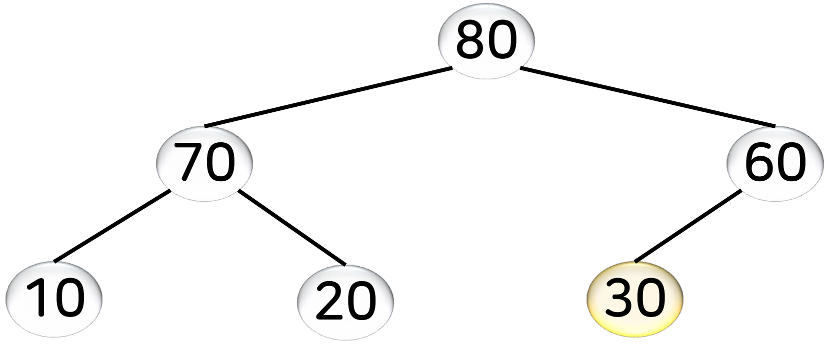
0	1	2	3	4
70	20	60	10	80

결과 배열

초기 힙 구축_방법 1

60	20	70	10	80	30	50	40

입력 배열



0	1	2	3	4	5
80	70	60	10	20	30

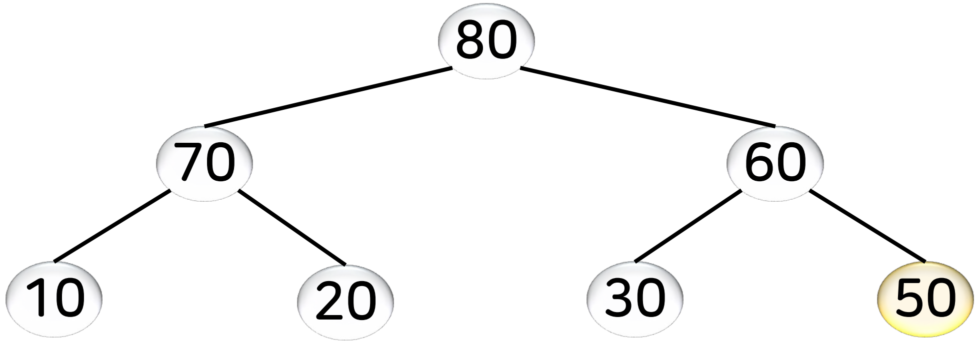
결과 배열

초 방법 1

01

60	20	70	10	80	30	50	40

입력 배열



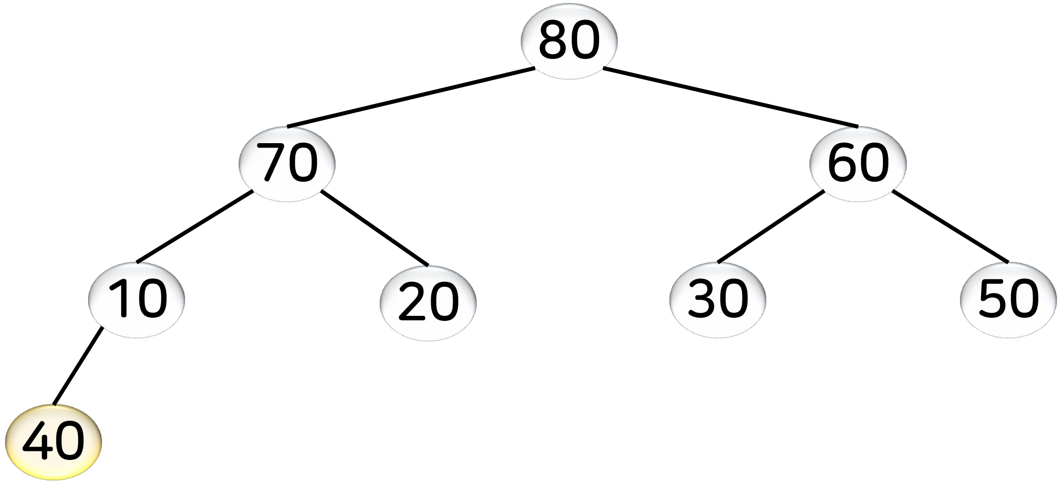
0	1	2	3	4	5	6
80	70	60	10	20	30	50

결과 배열

초기 힙 구축_방법 1

60	20	70	10	80	30	50	40

입력 배열



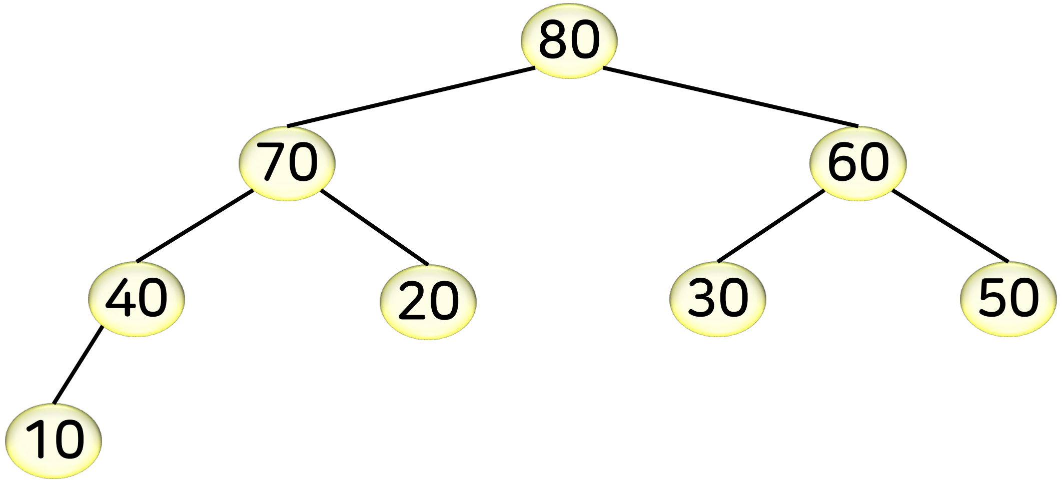
0	1	2	3	4	5	6	7
80	70	60	10	20	30	50	40

결과 배열

초기 힙 구축_방법 1

60	20	70	10	80	30	50	40

입력 배열

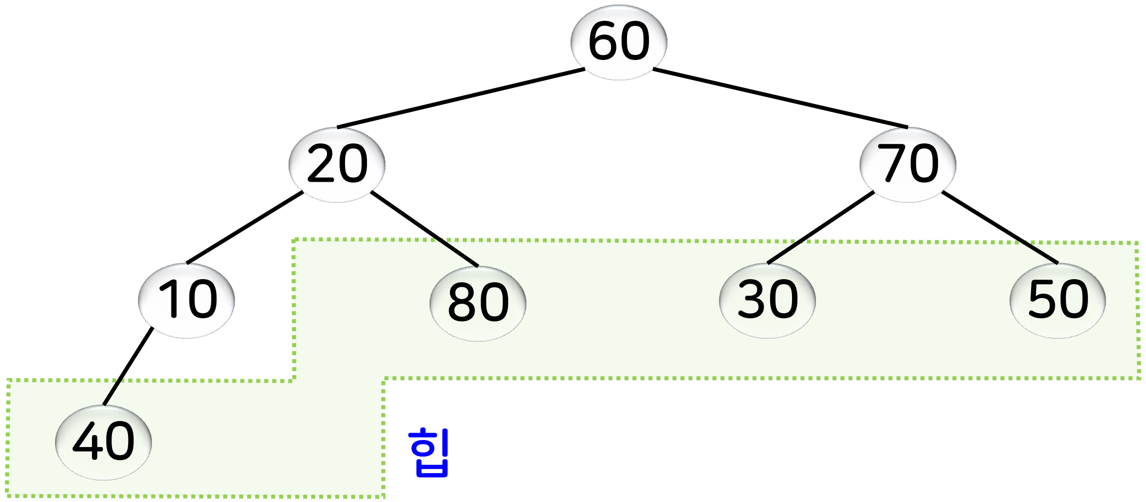


0	1	2	3	4	5	6	7
80	70	60	40	20	30	50	10

힙으로 변환된 최종 상태

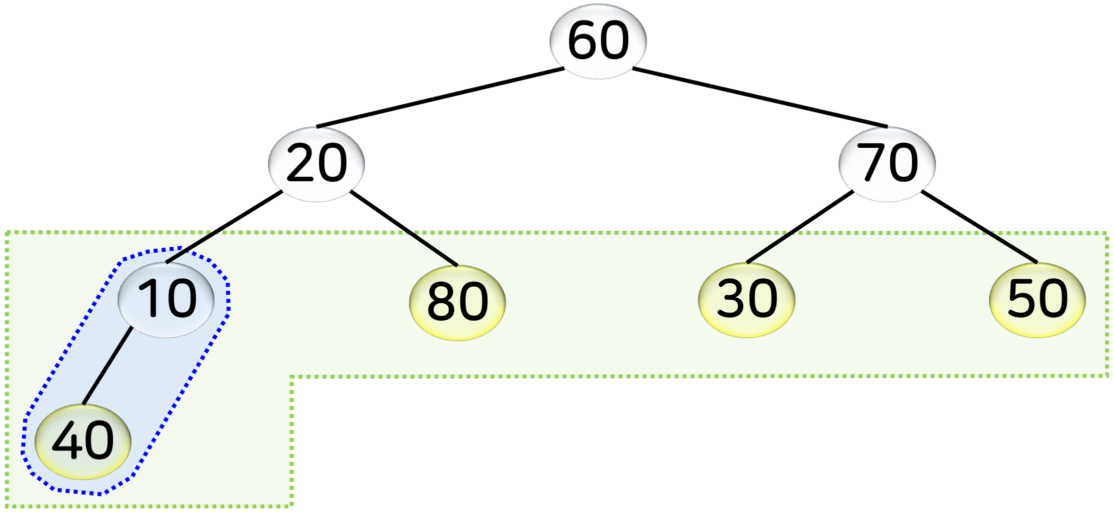
초기 힙 구축_방법 2

0	1	2	3	4	5	6	7
60	20	70	10	80	30	50	40



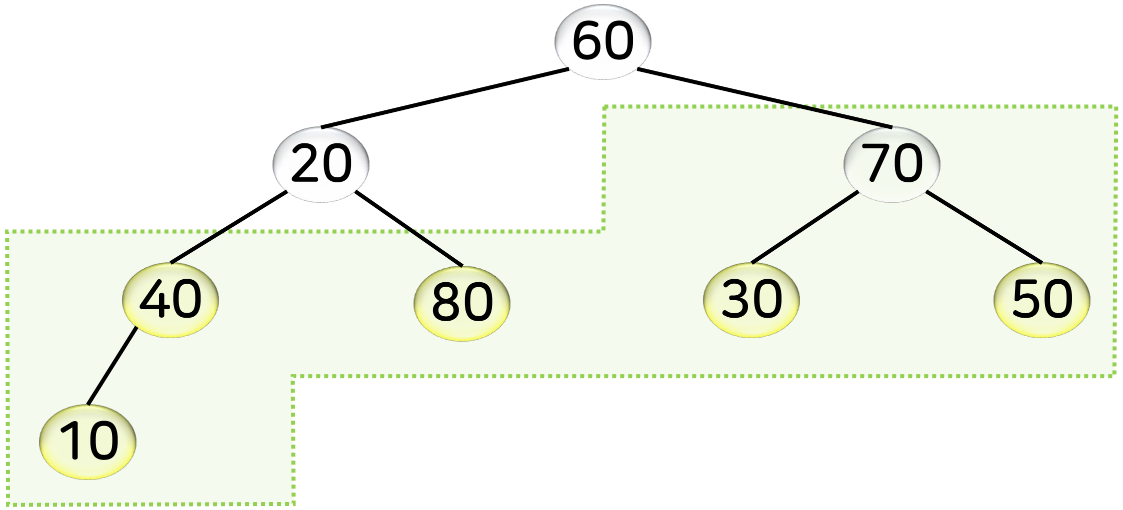
초기 힙 구축_방법 2

0	1	2	3	4	5	6	7
60	20	70	10	80	30	50	40



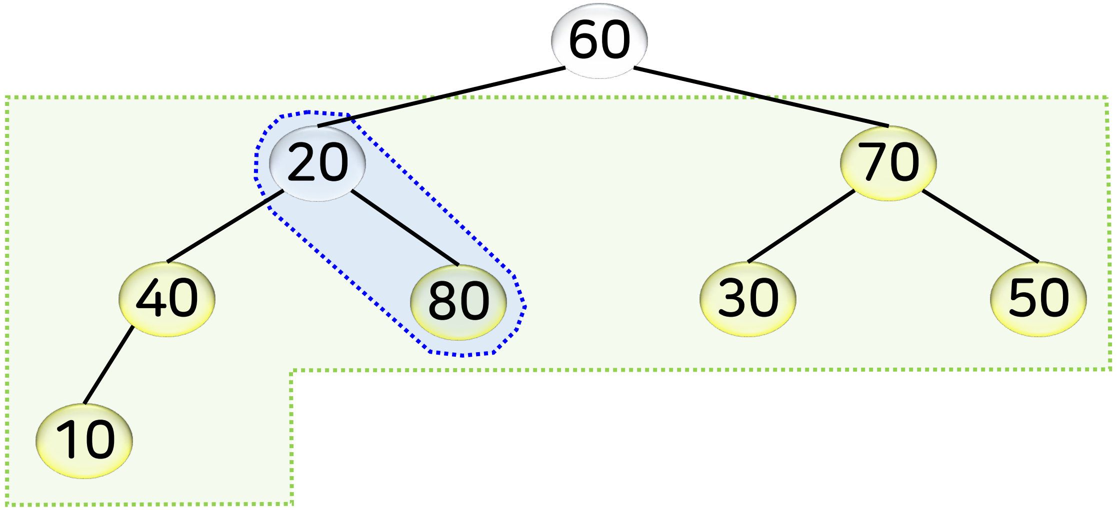
초기 힙 구축_방법 2

0	1	2	3	4	5	6	7
60	20	70	40	80	30	50	10



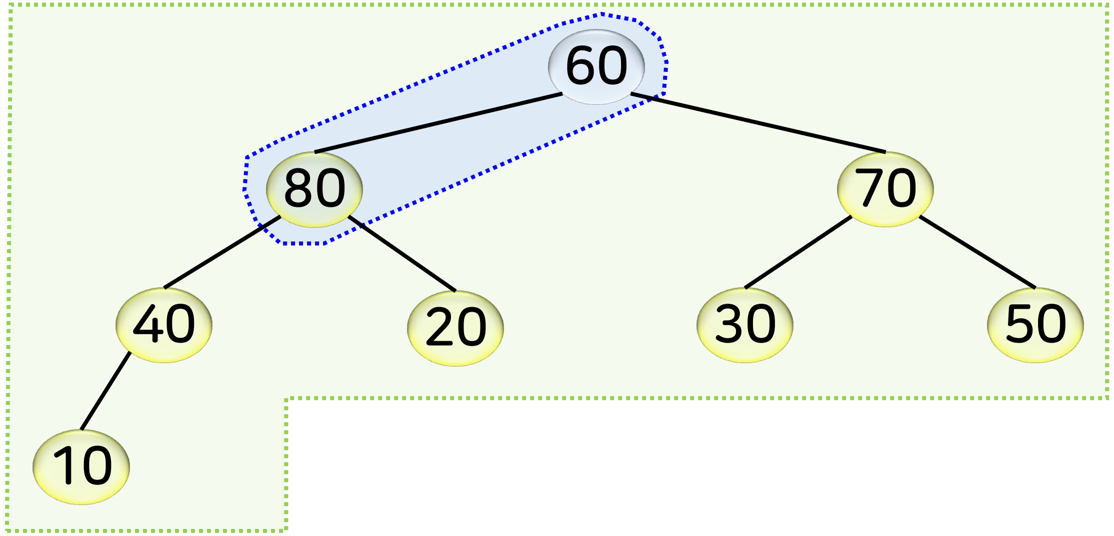
초기 힙 구축_방법 2

0	1	2	3	4	5	6	7
60	20	70	40	80	30	50	10



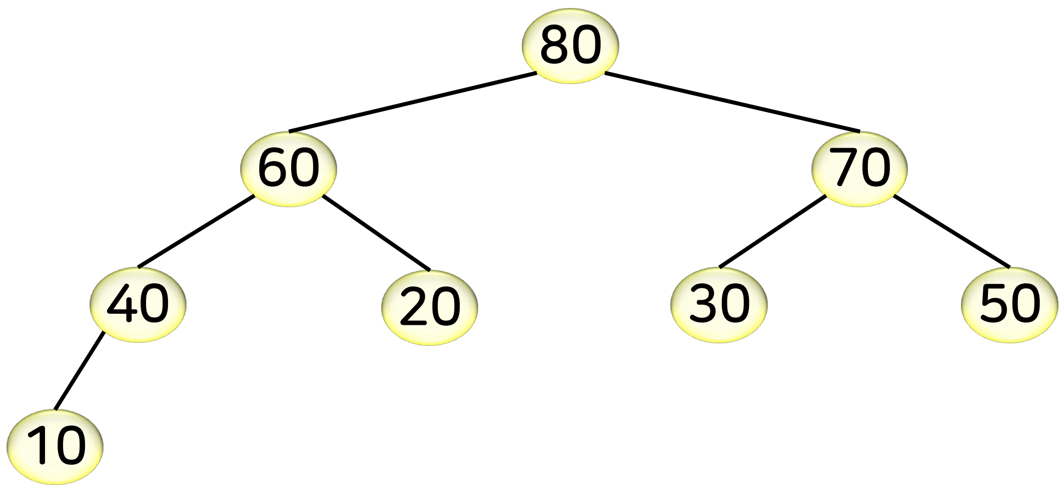
초기 힙 구축_방법 2

0	1	2	3	4	5	6	7
60	80	70	40	20	30	50	10



초기 힙 구축_방법 2

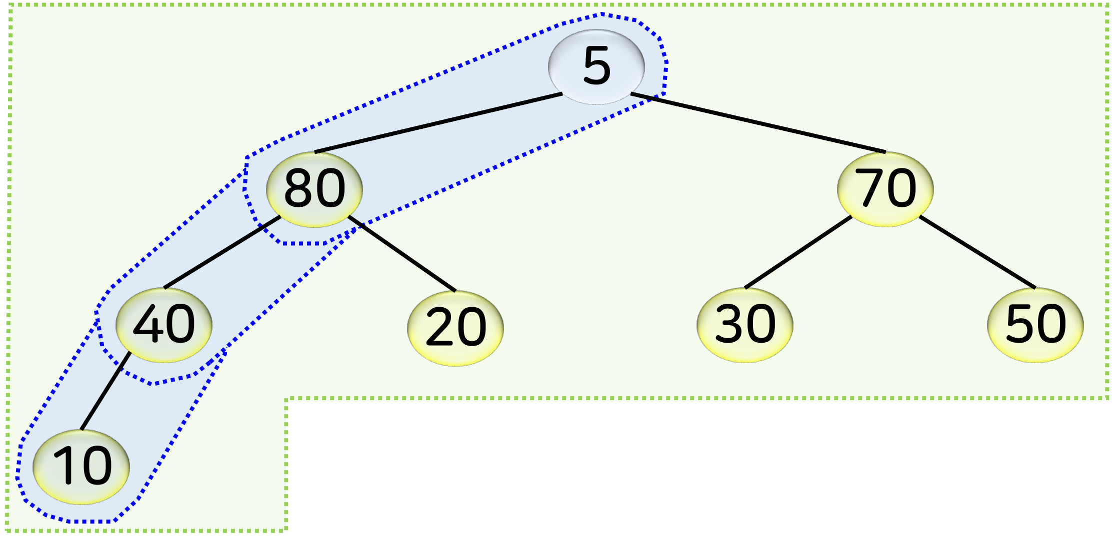
0	1	2	3	4	5	6	7
80	60	70	40	20	30	50	10



힙으로 변환된 최종 상태

초기 힙 구축_방법 2_추가 설명

0	1	2	3	4	5	6	7
5	80	70	40	20	30	50	10

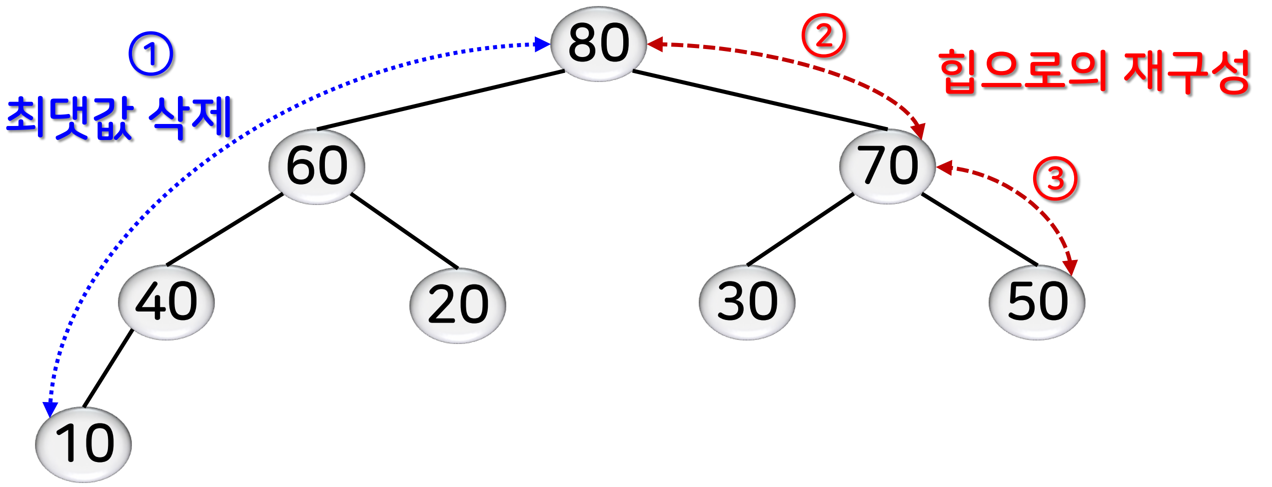


→ 비교/위치 교환은 힙의 조건이 만족될 때까지 트리를 따라 내려가면서 계속 진행

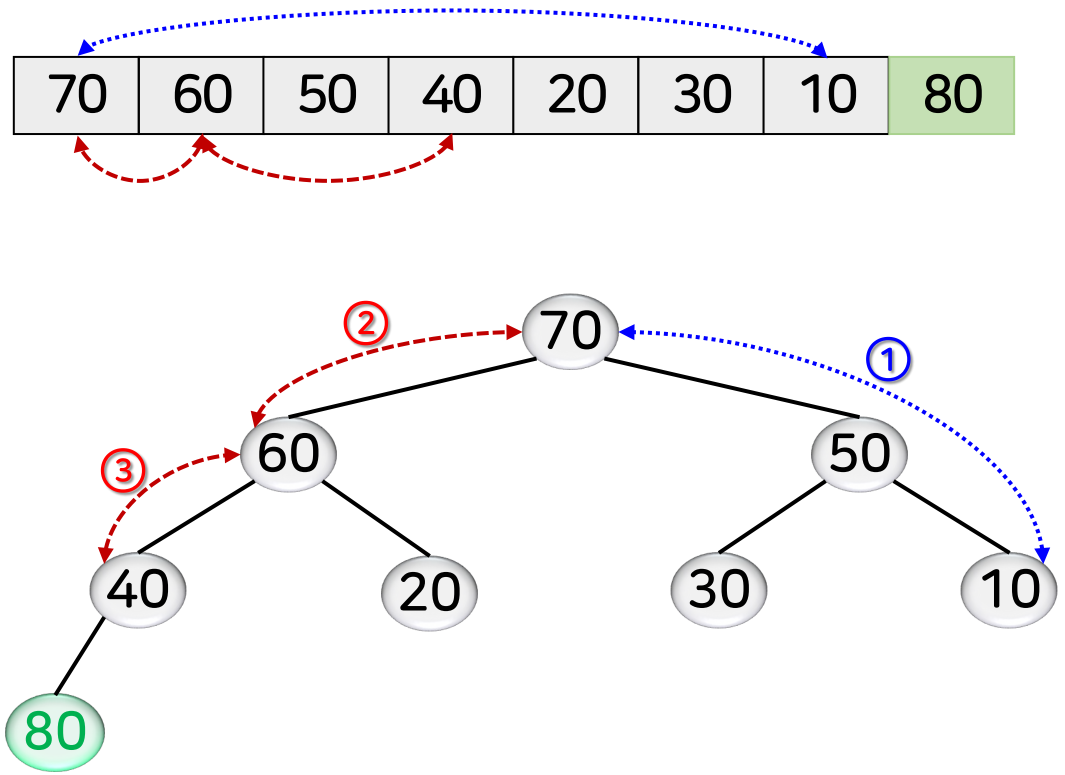
힙 정렬_두 번째 단계

A[ ]={ 60, 20, 70, 10, 80, 30, 50, 40 }

초기 힙	80	60	70	40	20	30	50	10

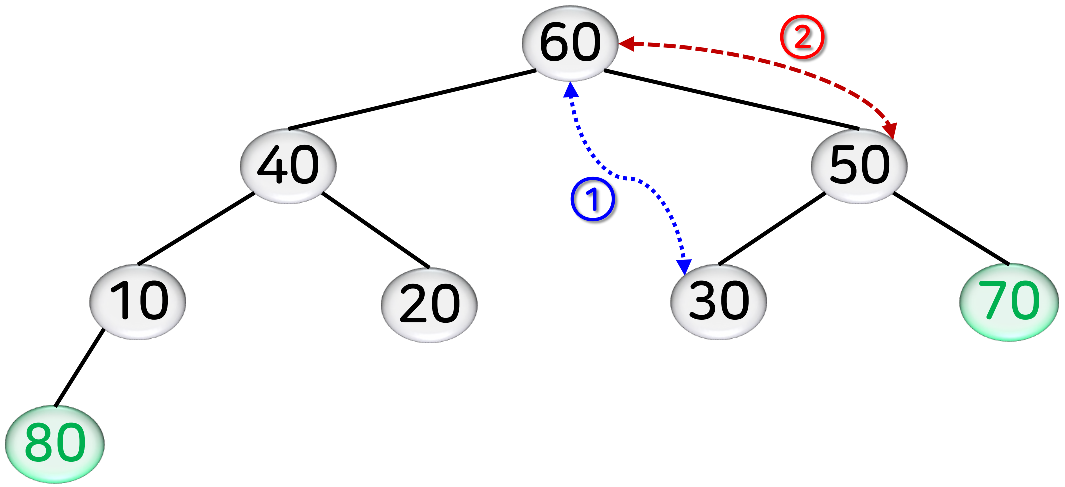


힙 정렬_두 번째 단계



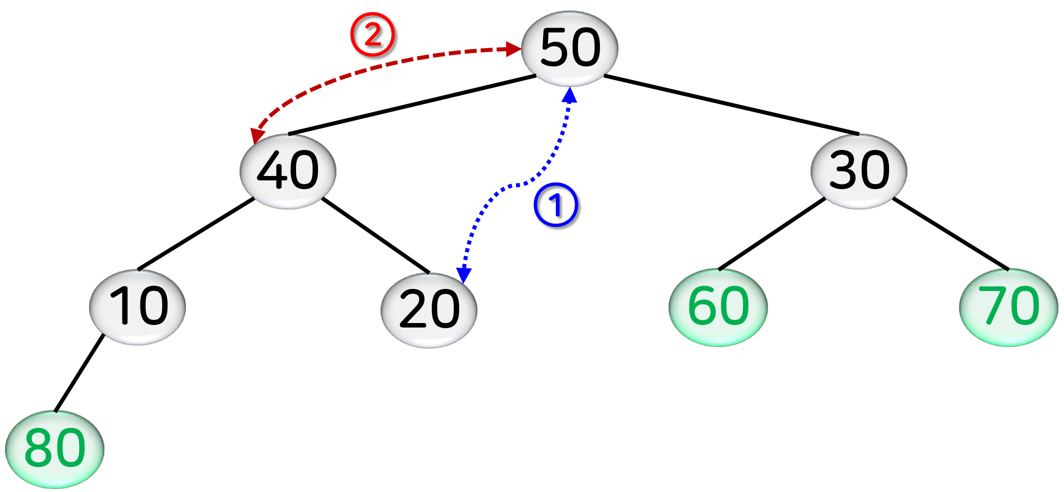
힙 정렬_두 번째 단계

60	40	50	10	20	30	70	80

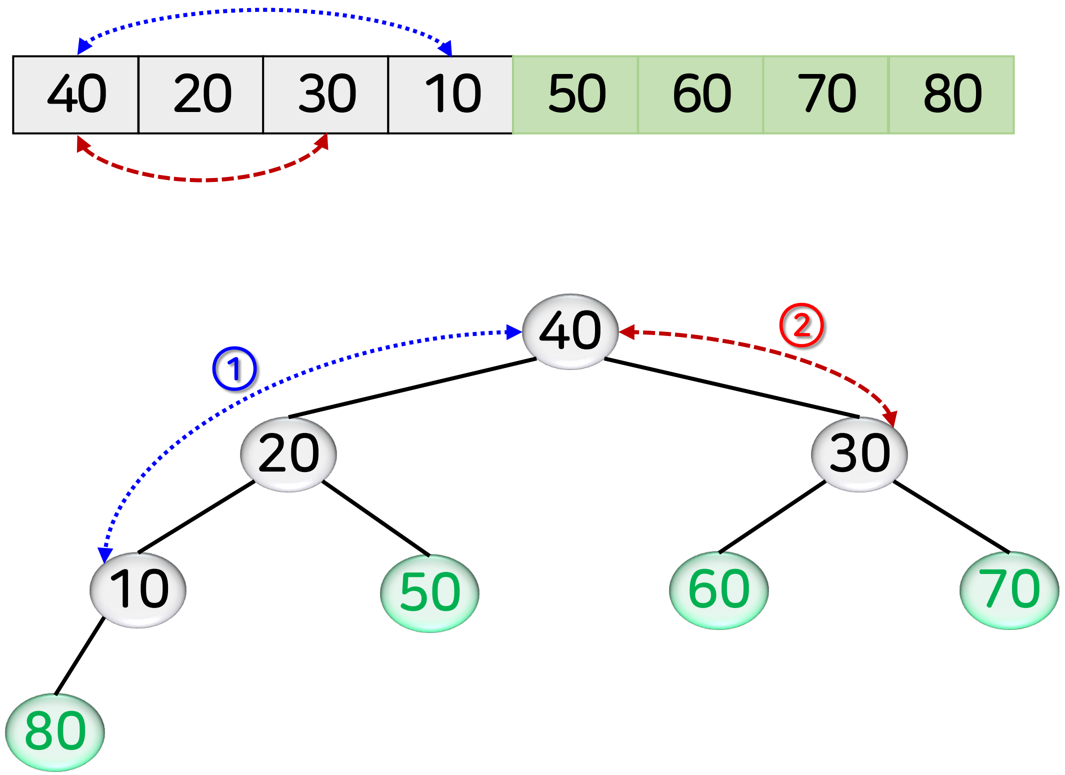


힙 정렬_두 번째 단계

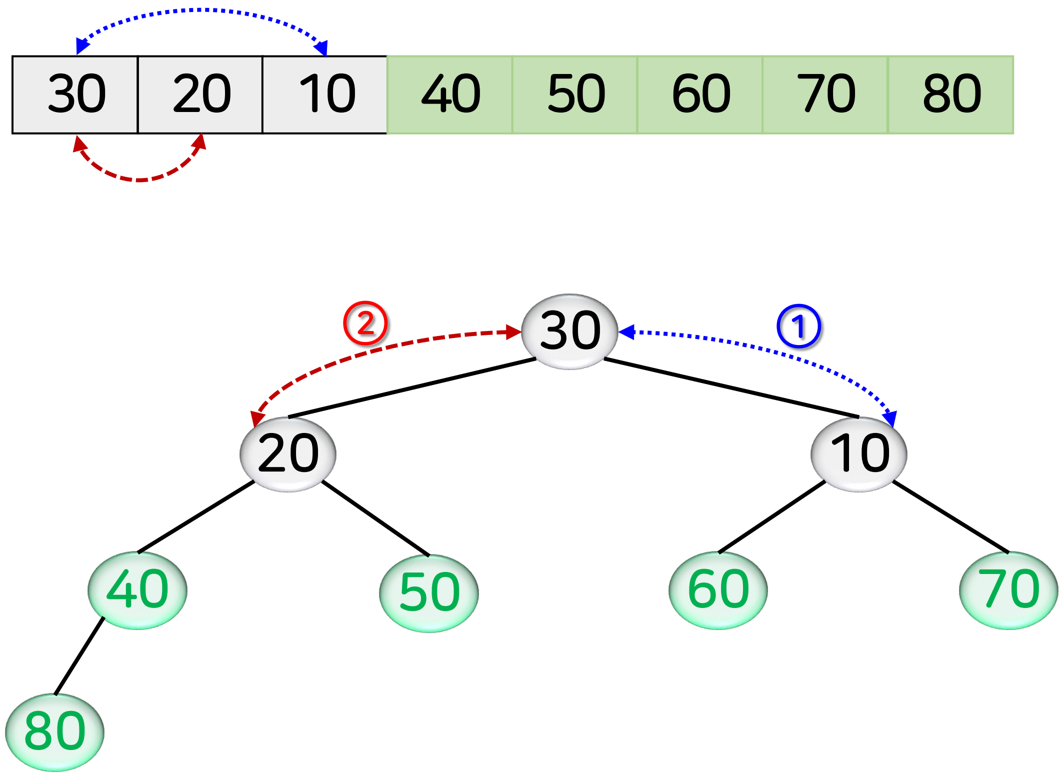
50	40	30	10	20	60	70	80



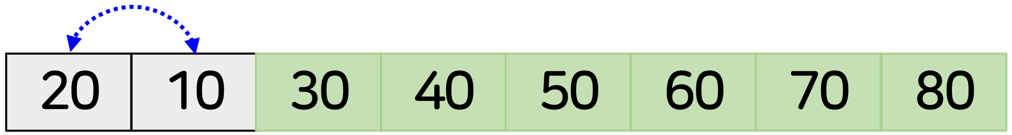
힙 정렬_두 번째 단계

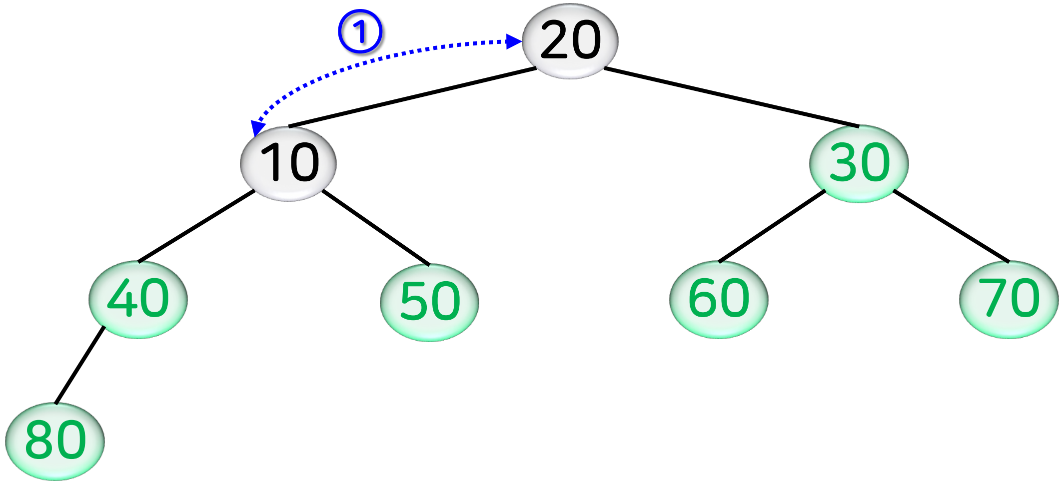


힙 정렬_두 번째 단계



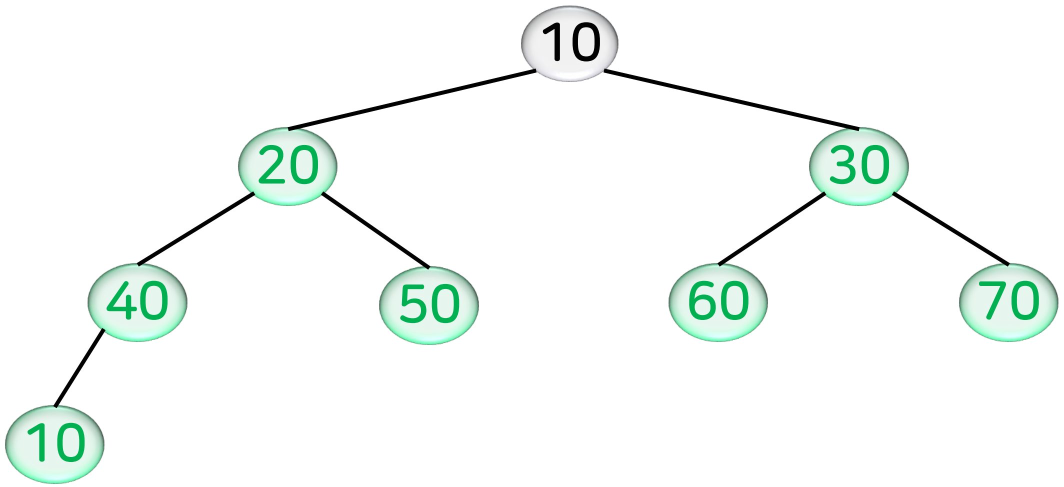
힙 정렬_두 번째 단계





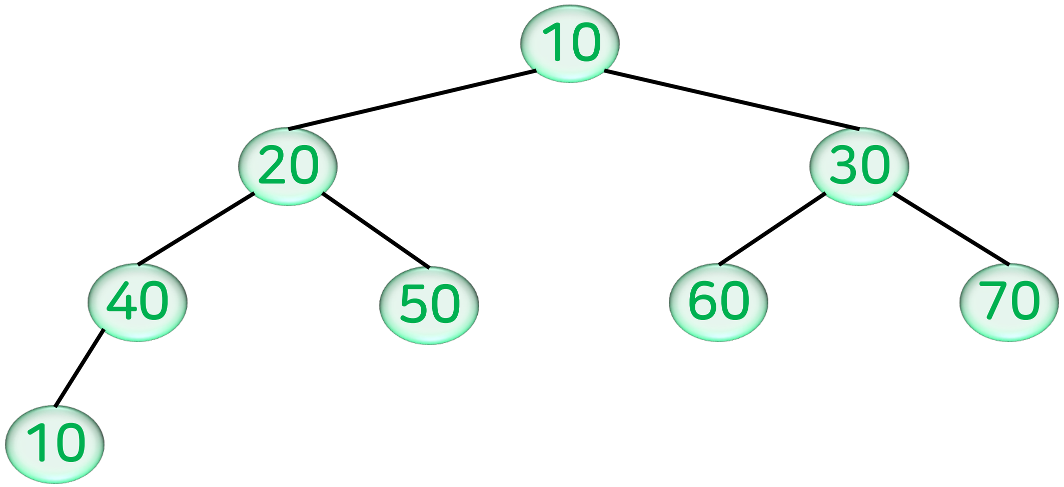
힙 정렬_두 번째 단계

10	20	30	40	50	60	70	80



힙 정렬_두 번째 단계

10	20	30	40	50	60	70	80



정렬 결과

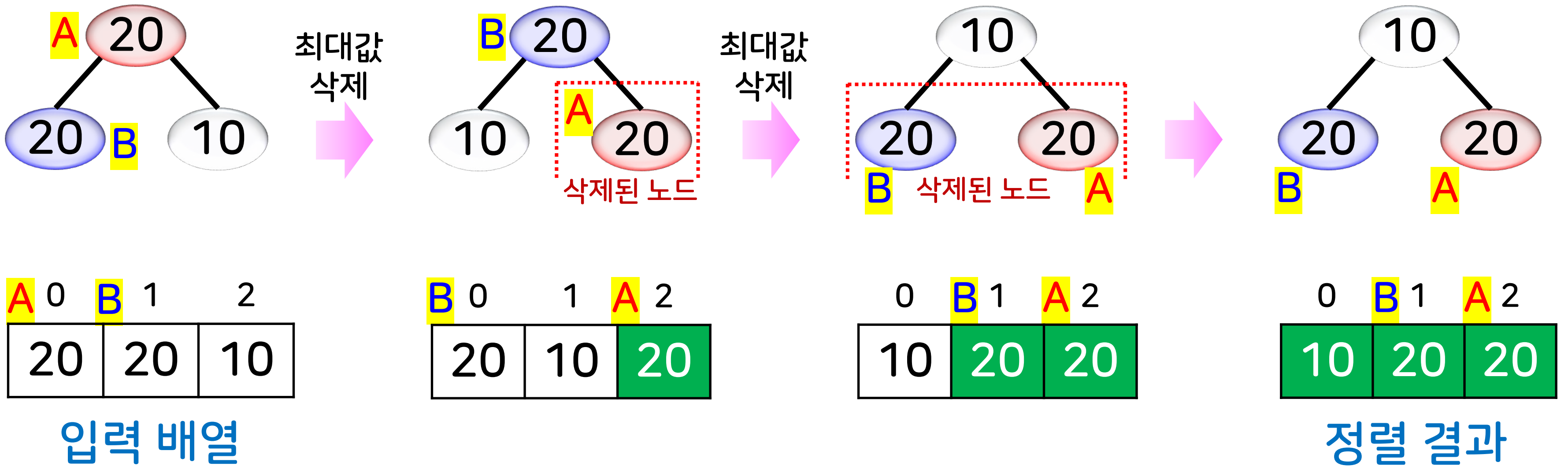
성능과 특징

최선, 최악, 평균 수행시간 → O(nlogn)

- 초기 힙 생성, 최댓값 삭제 및 힙 재구성
  - ✓ 바깥 루프 → 입력 크기 n에 비례
  - ✓ 안쪽 루프 → 완전 이진 트리의 높이 logn에 비례

성능과 특징

안정적이지 않은 정렬 알고리즘



제자리 정렬 알고리즘

02. 계수 정렬

정렬 알고리즘

비교 기반의 정렬 알고리즘

- 기본 성능 $O(n^2)$ → 선택 정렬, 버블 정렬, 삽입 정렬, 셸 정렬
- 향상된 평균 성능 O(nlogn) → 퀵 정렬, 합병 정렬, 힙 정렬
- 비교 기반 정렬 알고리즘 성능의 하한 → O(nlogn)

  → 아무리 빨라도 O(nlogn) 보다 효율적인 알고리즘은 구할 수 없음

이미 얻어진 데이터 분포 정보를 활용하는 정렬 알고리즘

- 계수 정렬, 기수 정렬, 버킷 정렬

  → 선형 시간 O(n)이 가능

계수 정렬

주어진 데이터 중에서 자신보다 작거나 같은 값을 갖는 데이터의 개수를 계산하여 정렬할 위치를 찾아 정렬하는 방식

- 입력값이 어떤 작은 정수 범위 내에 있다는 것을 알고 있는 경우에 적용 가능
- k보다 작거나 같은 값을 갖는 데이터의 개수
  → 정렬 순서상 k의 마지막 위치
- 자신보다 작거나 같은 값을 갖는 데이터의 개수의 효율적인 계산 방법
  ✓ 입력값의 범위 a~b에 해당하는 크기의 배열 COUNT[a..b]를 할당하고, 주어진 값들을 한 번 쭉 훑으면 각 입력값의 출현횟수의 누적값 계산이 가능

계수 정렬 알고리즘


```
CountingSort (A[1..n], n)
{
  MIN = MAX = A[1];
  for (i=2; i<=n; i++) {                       // 입력값의 범위 MIN~MAX 계산
    if (A[i] < MIN) MIN = A[i];
    if (A[i] > MAX) MAX = A[i];
  }
  for (j=MIN; j <= MAX; j++)  COUNT[j] = 0;
  for (i=1; i <= n; i++)  COUNT[ A[i] ]++;     // 각 입력값의 출현횟수 계산
  for (j=MIN+1; j <= MAX; j++)                 // 각 입력값의 출현횟수의 누적값 계산
    COUNT[j] = COUNT[j] + COUNT[j-1];
  for (i=n; i > 0; i--) {                      // 입력 배열 A[] → 정렬 배열 B[]
    B[ COUNT[ A[i] ] ] = A[i];
    COUNT[ A[i] ]--;
  }
  return (B);
}
```

계수 정렬의 예

입력값의 범위: 4~9

A[ ]	7	5	9	8	4	5	7	5

	4	5	6	7	8	9
COUNT[ ]	0	0	0	0	0	0

각 입력값의 출현 횟수 계산: COUNT[A[i]]++

7	5	9	8	4	5	7	5

4	5	6	**7**	8	9
0	0	0	1	0	0

7	**5**	9	8	4	5	7	5

4	**5**	6	7	8	9
0	1	0	1	0	0

7	5	**9**	8	4	5	7	5

4	5	6	7	8	**9**
0	1	0	1	0	1

7	5	9	**8**	4	5	7	5

4	5	6	7	**8**	9
0	1	0	1	1	1

7	5	9	8	**4**	5	7	5

4	5	6	7	8	9
1	1	0	1	1	1

계수 정렬의 예

7	5	9	8	4	5	7	5

4	5	6	7	8	9
1	2	0	1	1	1

7	5	9	8	4	5	7	5

4	5	6	7	8	9
1	2	0	2	1	1

7	5	9	8	4	5	7	5

4	5	6	7	8	9
1	3	0	2	1	1

각 입력값의 출현 횟수의 **누적값** 계산: COUNT[i] = COUNT[i] + COUNT[i-1]

4	5	6	7	8	9
1	3	0	2	1	1
1	4	4	6	7	8

예상되는 정렬 결과 B[ ]

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

계수 정렬의 예

입력값의 정렬: B[COUNT[A[i]]] = A[i], COUNT[A[i]]--

A[ ]

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

COUNT[ ]

4	5	6	7	8	9
1	4	4	6	7	8

B[COUNT [A[i]]] = A[i]

B[COUNT [A[8]]] = A[8]

B[COUNT [5]] = 5

B[4] = 5

COUNT [A[i]] --

COUNT [A[8]] --

COUNT [5] --

B[ ]

1	2	3	4	5	6	7	8

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

계수 정렬의 예

입력값의 정렬: B[COUNT[A[i]]] = A[i], COUNT[A[i]]--

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

B[COUNT [A[i]]] = A[i]

B[COUNT [A[7]]] = A[7]

B[COUNT [7]] = 7

B[6] = 7

	1	2	3	4	5	6	7	8
B[ ]				5				

COUNT[ ]

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

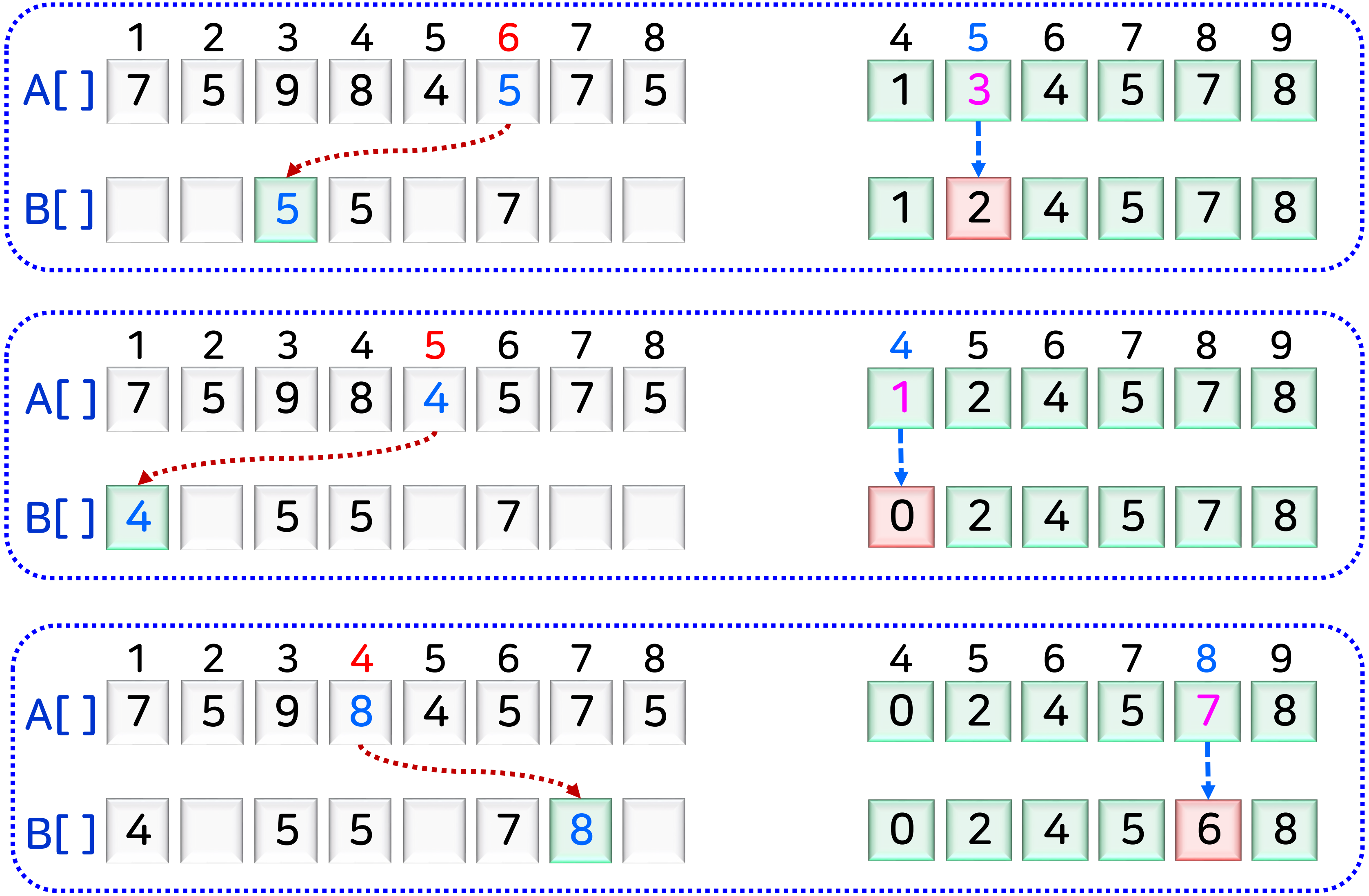
COUNT [A[i]] --

COUNT [A[7]] --

COUNT [7] --

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

계수 정렬의 예


1 2 3 4 5 6 7 8
A[ ] 7 5 9 8 4 5 7 5
B[ ] 5 5 7
4 5 6 7 8 9
1 3 4 5 7 8
1 2 4 5 7 8
1 2 3 4 5 6 7 8
A[ ] 7 5 9 8 4 5 7 5
B[ ] 4 5 5 7
4 5 6 7 8 9
1 2 4 5 7 8
0 2 4 5 7 8
1 2 3 4 5 6 7 8
A[ ] 7 5 9 8 4 5 7 5
B[ ] 4 5 5 7 8
4 5 6 7 8 9
0 2 4 5 7 8
0 2 4 5 6 8

계수 정렬의 예

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

4	5	6	7	8	9
0	2	4	5	6	8
0	2	4	5	6	7

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

4	5	6	7	8	9
0	2	4	5	6	7
0	1	4	5	6	7

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

정렬 결과

4	5	6	7	8	9
0	1	4	5	6	7
0	1	4	4	6	7

성능과 특징


```
CountingSort (A[1..n], n)
{
  MIN = MAX = A[1];
  for (i=2; i<=n; i++) {
    if (A[i] < MIN) MIN = A[i];
    if (A[i] > MAX) MAX = A[i];
  }
  for (j=MIN; j <= MAX; j++)  COUNT[j] = 0;
  for (i=1; i <= n; i++)  COUNT[ A[i] ]++;
  for (j=MIN+1; j <= MAX; j++)
    COUNT[j] = COUNT[j] + COUNT[j-1];
  for (i=n; i > 0; i--) {
    B[ COUNT[ A[i] ] ] = A[i];
    COUNT[ A[i] ]--;
  }
  return (B);
}
```

O(n)

O(k)

k = MAX-MIN+1

O(n+k) → k = O(n) → O(n)

성능과 특징

입력값의 범위가 데이터의 개수보다 작거나 비례할 때 유용

- 입력값의 범위를 k라고 할 때 O(n+k) 시간

  → k=O(n)이 되어야 선형 시간 O(n)에 동작

안정적인 정렬 알고리즘

- 입력 배열 A[ ]의 오른쪽의 것부터 뽑아서 결과 배열 B[ ]의 오른쪽에서부터 저장

제자리 정렬 알고리즘이 아님

- 입력 배열 A[1..n] + (COUNT[a..b], B[1..n])

보편적이지 못한 정렬 알고리즘

- 입력값의 범위를 미리 알아야 함 + 추가적인 배열이 필요

03. 기수 정렬

기수 정렬

입력값을 자릿수별로 구분해서 부분적으로 비교하여 정렬하는 방식

- 주어진 데이터의 값을 자릿수별로 나누고, 각 자릿수에 대해 계수 정렬과 같은 안정적인 정렬 알고리즘을 적용하여 정렬
  - ✓ LSD Least Significant Digit 기수 정렬 → 낮은 자리부터 높은 자리로 진행, "Right-to-Left"
  - ✓ MSD Most Significant Digit 기수 정렬 → 높은 자리부터 낮은 자리로 진행, "Left-to-Right"

기수 정렬 알고리즘


```
RadixSort (A[ ], n)
{
  for (i=1; i<=d; i++) {   // d 자릿수, LSD 기수 정렬
    각 데이터의  i자리의 값에 대해서 안정적인 정렬 알고리즘 적용;
  }
  return (A);
}
```

기수 정렬의 예_LSD 기수 정렬

입력 288 265 482 095 361 068 185 100

일

0: 100
1: 361
2: 482
3:
4:
5: 265 095 185
6:
7:
8: 288 068
9:

십

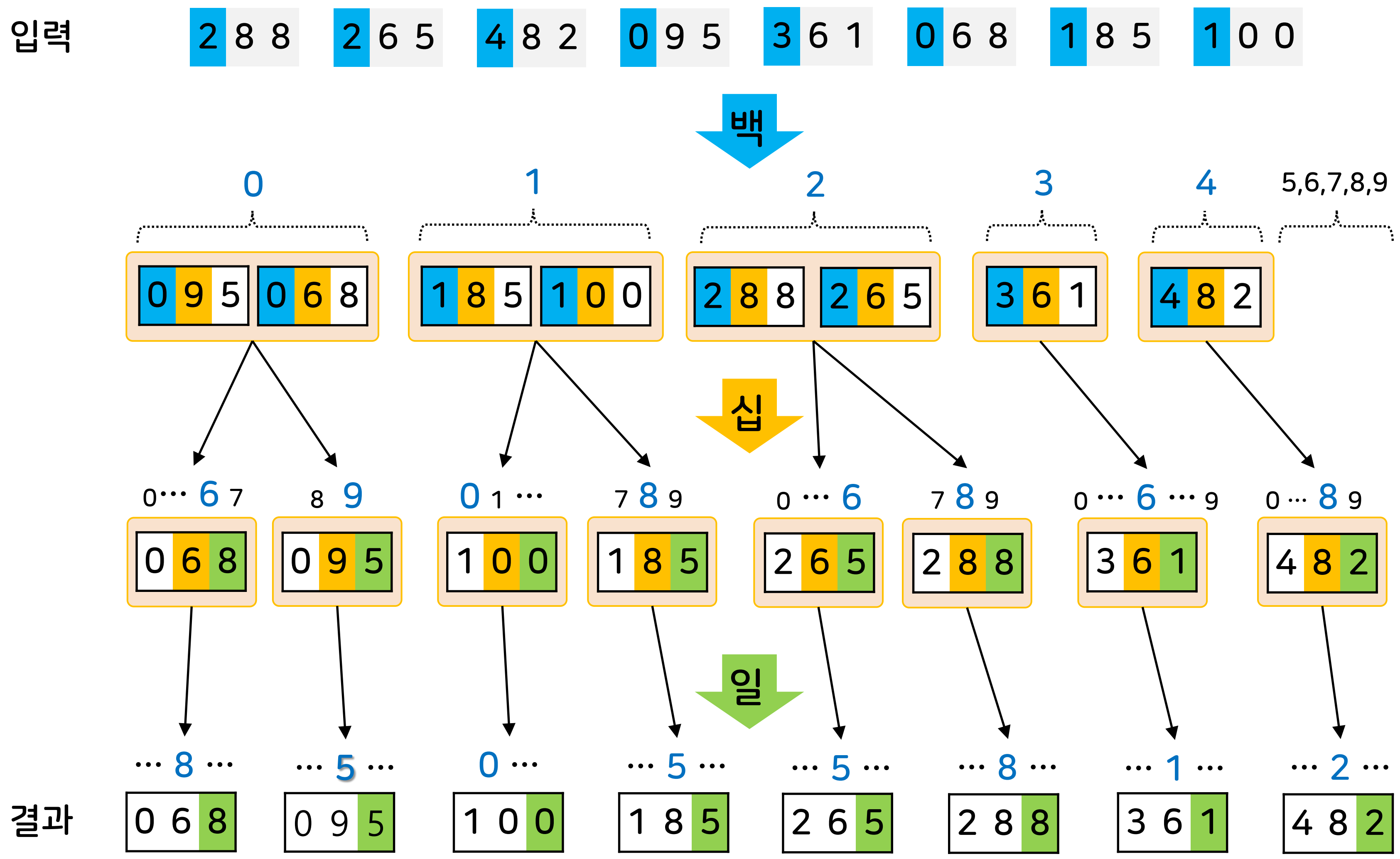
0: 100
1:
2:
3:
4:
5:
6: 361 265 068
7:
8: 482 185 288
9: 095

백

0: 068 095
1: 100 185
2: 265 288
3: 361
4: 482
5:
6:
7:
8:
9:

결과 068 095 100 185 265 288 361 482

기수 정렬의 예_MSD 기수 정렬



성능과 특징


```
RadixSort (A[ ], n)
{
  for (i=1; i<=d; i++) {
    각 데이터의  i자리의 값에 대해서 안정적인 정렬 알고리즘 적용;
  }
  return (A);
}
```

계수 정렬 사용 → O(n)

O(dn)

자릿수 d를 상수로 취급

O(n)

성능과 특징

입력 데이터의 자릿수가 상수일 때 유용

- d 자릿수 n개의 숫자들에 대해 계수 정렬을 적용하면 O(dn)

  → 여기서 d를 입력 크기 n가 무관한 상수로 간주하면 O(n)

안정적인 정렬 알고리즘

- 각 자릿수별로 안정적인 정렬 알고리즘을 적용하므로 기수 정렬도 안정적

제자리 정렬 알고리즘이 아님

- 계수 정렬 적용 → 전체 데이터 개수와 진법 크기만큼의 추가 공간이 필요

04. 버킷 정렬

버킷 정렬

① 주어진 데이터들의 값의 범위를 균등하게 나누어 여러 개의 버킷을 만든 뒤,

② 각 데이터를 해당하는 버킷에 넣고,

③ 각 버킷을 삽입 정렬과 같은 안정적인 정렬을 수행한 후,

④ 버킷 순서대로 각 데이터를 나열하는 정렬 방식

- 입력값의 범위 내에서 값이 확률적으로 균등하게 분포될 때 선형 시간에 동작

버킷 정렬 알고리즘

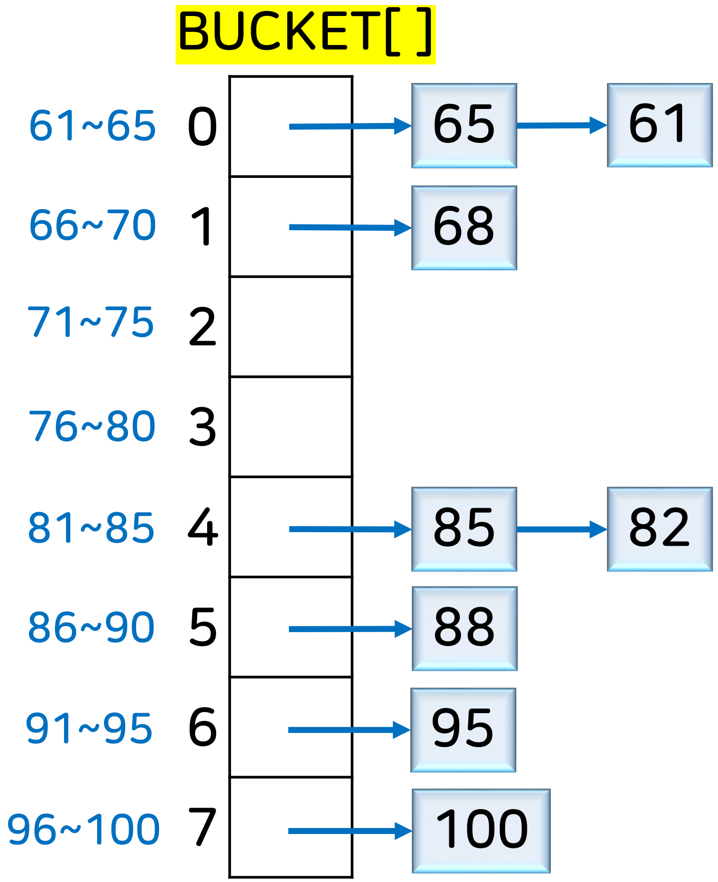

```
BucketSort (A[ ], n)
{
  MIN = MAX = A[0];
  for (i=1; i < n; i++) {              // 입력값의 범위 MIN~MAX 계산
     if (A[i] < MIN) MIN = A[i];
     if (A[i] > MAX) MAX = A[i];
  }
  INTERVAL = ⌊(MAX – MIN + 1) / n⌋;  // 버킷 구간의 크기 계산
  for (i=0; i < n; i++)                // 각 데이터를  해당 버킷에 넣기
     A[i]를 BUCKET[⌊(A[i] - MIN)/ INTERVAL ⌋]에 삽입;
  for (i=0; i < n; i++)                // 버킷별로 정렬
     삽입 정렬에 의해 BUCKET[i]를 정렬;
  BUCKET[0], BUCKET[1], …의 순서대로 데이터를 배열 B[ ]에 삽입;
  return (B);
}
```

버킷 정렬의 예

A[ ] = { 85 | 68 | 65 | 100 | 88 | 61 | 82 | 95 }

입력값의 범위: 61 ~ 100

구간의 크기 5 (= ⌊(MAX – MIN + 1) / n⌋ = ⌊(100-61+1) / 8⌋)



① 여러 개의 버킷 만들기

② 각 데이터를 해당 버킷에 넣기

③ 각 버킷을 삽입 정렬하기

④ 버킷 순서대로 각 데이터 나열하기

정렬 결과

B[ ] = { }

성능과 특징


```
BucketSort (A[ ], n)
{
  MIN = MAX = A[0];
  for (i=1; i < n; i++) {
    if (A[i] < MIN) MIN = A[i];
    if (A[i] > MAX) MAX = A[i];
  }
  INTERVAL = ⌊(MAX − MIN + 1) / n⌋;
  for (i=0; i < n; i++)
    A[i]를 BUCKET[⌊(A[i] - MIN)/ INTERVAL⌋]에 삽입;
  for (i=0; i < n; i++)
    삽입 정렬에 의해 BUCKET[i]를 정렬;
  BUCKET[0], BUCKET[1], …의 순서대로 데이터를 배열 B[ ]에 삽입;
  return (B);
}
```

O(n)

O(n)

O(n)

O(n)

→ O(n)

성능과 특징

입력 데이터의 값이 확률적으로 균등하게 분포할 때 유용

- 버킷별 정렬 → 데이터들이 각 버킷에 균등하게 들어갈 때 효율적인 정렬이 가능

버킷의 개수가 입력 데이터의 개수에 비례해야 유용

- 버킷의 개수를 $\lfloor n/k \rfloor$개로 정하면 각 버킷에는 상수(k) 개의 데이터가 존재
  → 각 버킷을 상수 시간에 정렬 가능 → 선형 시간의 동작이 가능

안정적인 정렬 알고리즘

- 데이터를 버킷에 넣을 때 그리고 각 버킷의 정렬 과정에서 상대적인 순서를 유지

제자리 정렬 알고리즘이 아님

- 추가적인 저장 공간(BUCKET[ ]과 크기 n의 배열 B[ ])이 필요

정리하기

1. 힙 정렬

- 힙 구조의 장점(임의의 값 삽입과 최댓값 삭제가 쉬움)을 활용한 정렬 방식
- 두 단계의 처리 과정, 두 가지의 초기 힙 구축 방법, O(nlogn), 불안정적, 제자리

2. 계수 정렬

- 데이터 분포 기반 → 입력값의 범위가 데이터의 개수보다 작거나 비례하는 경우 → O(n)
- 안정적, 제자리 정렬이 아님

3. 기수 정렬

- 데이터 분포 기반 → 입력 데이터의 자릿수가 상수인 경우 → O(n)
- 안정적, 제자리 정렬이 아님

4. 버킷 정렬

- 데이터 분포 기반 → 입력 데이터의 값이 확률적으로 균등하게 분포하는 경우 → O(n)
- 안정적, 제자리 정렬이 아님

ALGORITHM
알고리즘

다음시간에는

Lecture 06

탐색 (1)

컴퓨터과학과 | 이관용 교수