정렬 알고리즘_2

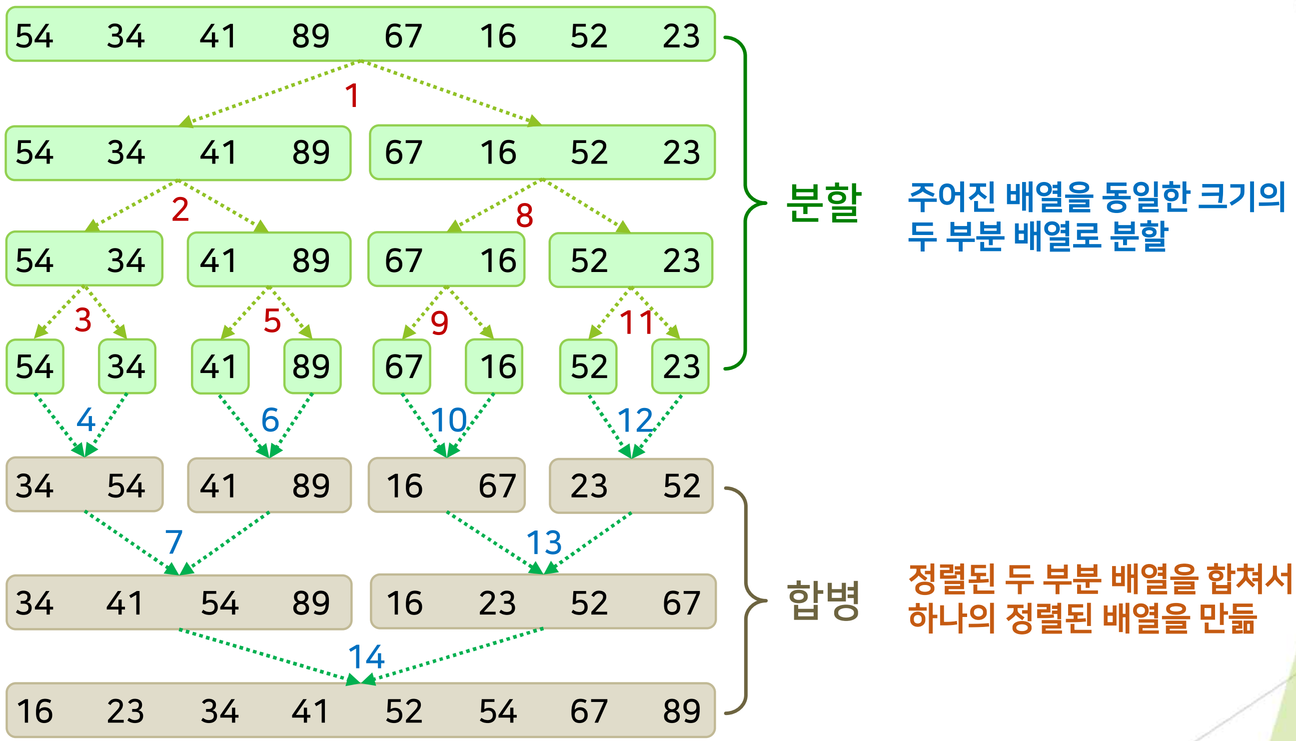
2020년도 2학기 최 희석

목차

- 합병 정렬과 퀵 정렬
- 힙 정렬
- 비교 기반 정렬의 하한
- 계수 정렬
- 기수 정렬

합병 정렬과 퀵 정렬

합병 정렬

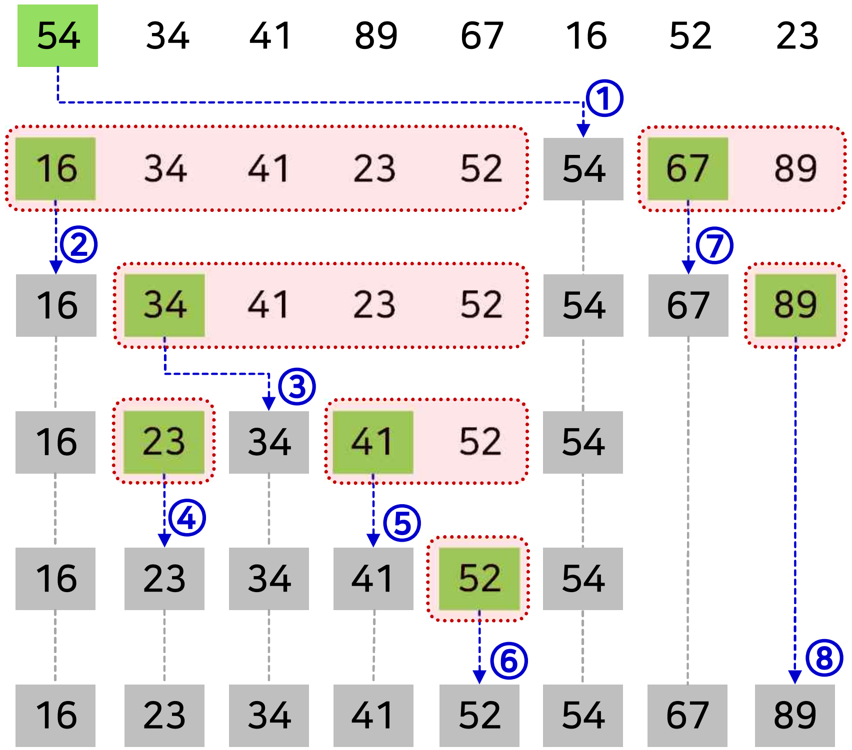

54 34 41 89 67 16 52 23
1
54 34 41 89
67 16 52 23
2
8
54 34
41 89
67 16
52 23
3
5
9
11
54
34
41
89
67
16
52
23
4
6
10
12
34 54
41 89
16 67
23 52
7
13
34 41 54 89
16 23 52 67
14
16 23 34 41 52 54 67 89
분할
주어진 배열을 동일한 크기의
두 부분 배열로 분할
합병
정렬된 두 부분 배열을 합쳐서
하나의 정렬된 배열을 만듦

합병 정렬의 성능과 특징

- 최선, 최악, 평균 수행 시간이 모두 **O(nlogn)**
- 안정적인 정렬 알고리즘
- 제자리 정렬 알고리즘이 아님

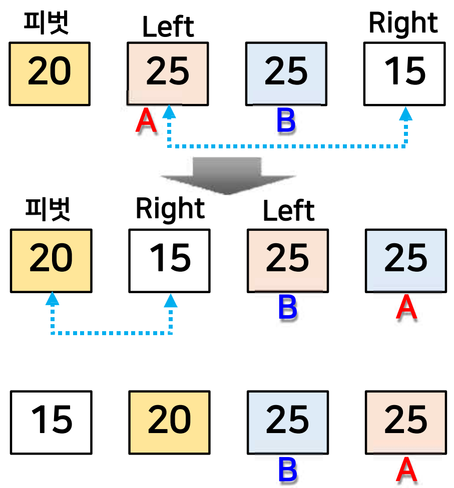
```
MergeSort (A[ ], n)
{
   if (n > 1) {
      Mid = ⌊n/2⌋;
      B[0..Mid-1] = MergeSort(A[0..Mid-1], Mid);
      C[Mid..n-1] = MergeSort(A[Mid..n-1], n-Mid);
      A[0..n-1] = Merge(B[0..Mid-1], C[0..n-Mid-1], Mid, n-Mid);
   }
   return A;
}
```

퀵 정렬


54 34 41 89 67 16 52 23
①
16 34 41 23 52 54 67 89
② ⑦
16 34 41 23 52 54 67 89
③
16 23 34 41 52 54
④ ⑤
16 23 34 41 52 54
⑥ ⑧
16 23 34 41 52 54 67 89

퀵 정렬의 성능과 특징

- 최악 수행 시간 O(n2), 최선/평균 수행 시간 O(nlogn)
  - 피벗 선택의 임의성만 보장되면 평균적인 성능을 보일 가능성이 매우 높음
- 안정적이지 않은 정렬 알고리즘



- 제자리 정렬 알고리즘

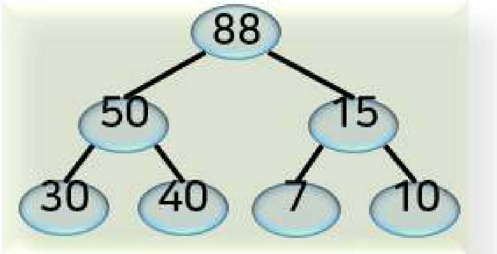
힙 정렬

힙 정렬의 개념과 원리

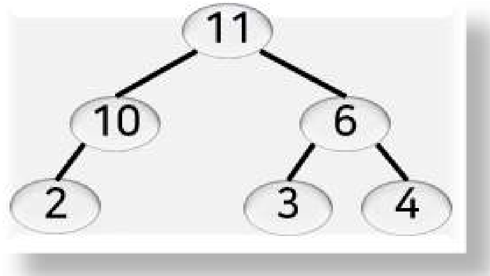
- 힙(heap) 자료구조의 장점을 활용한 정렬 방식
  - **임의의 값 삽입**과 **최댓값** 삭제가 용이

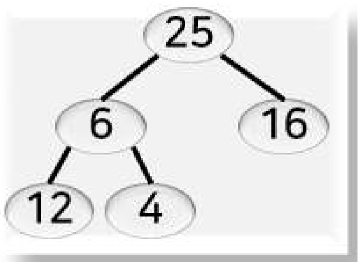
- **(최대)** 힙 heap?
  - 완전 이진 트리
    - 리프 노드 제외, 모든 레벨의 노드가 모두 채워진 형태
    - 마지막 레벨의 노드들은 가능한 한 왼쪽부터 채워지는 구조
  - 각 노드의 값은 자신의 자식 노드의 값보다 **크거나** 같다

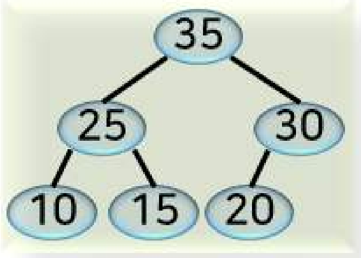
힙 정렬의 개념과 원리



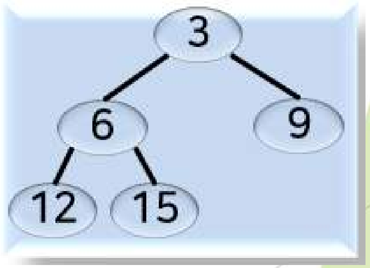
최대 Heap





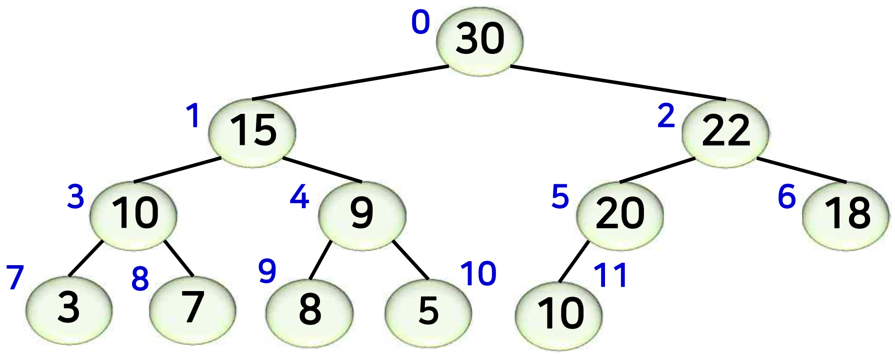


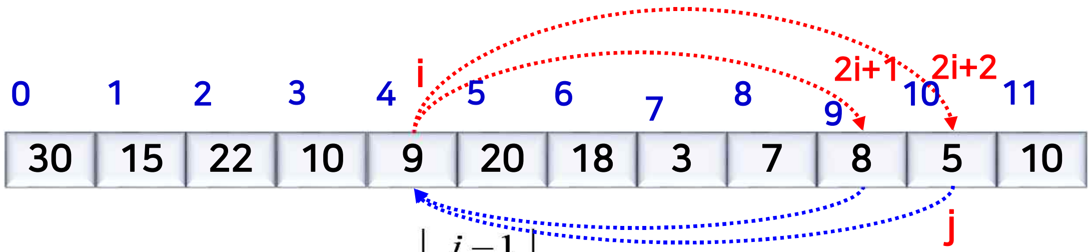
최대 Heap



최소 Heap

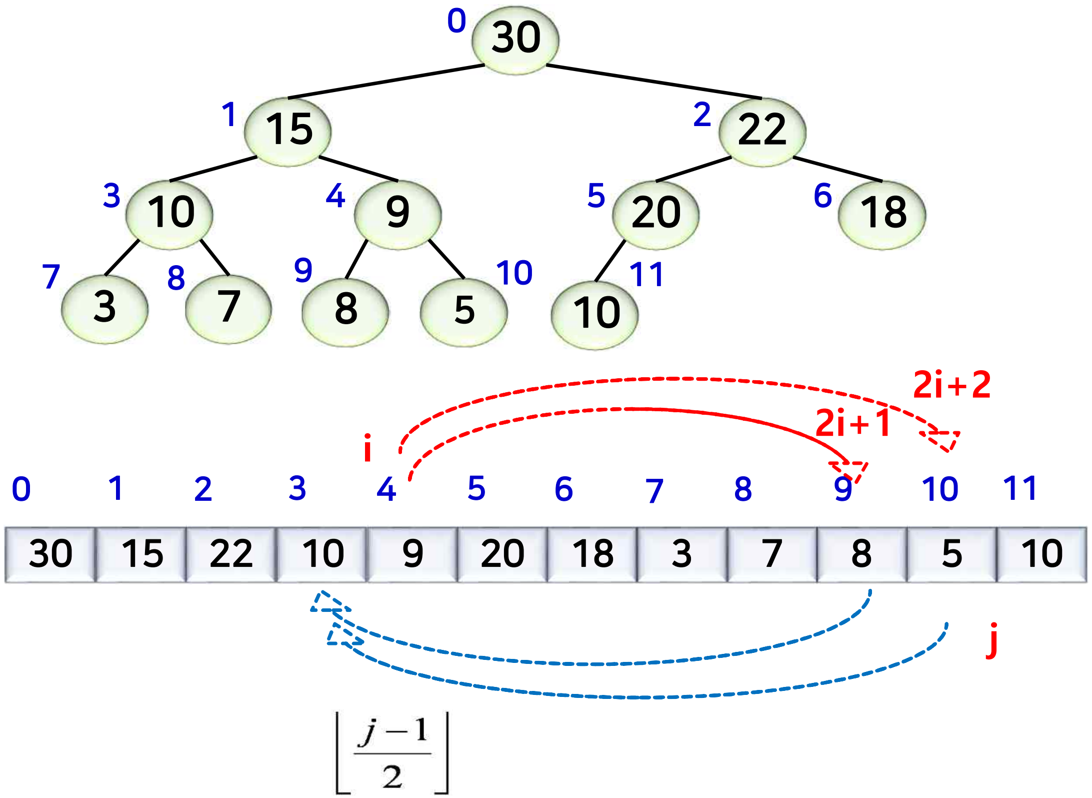
힙 정렬





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

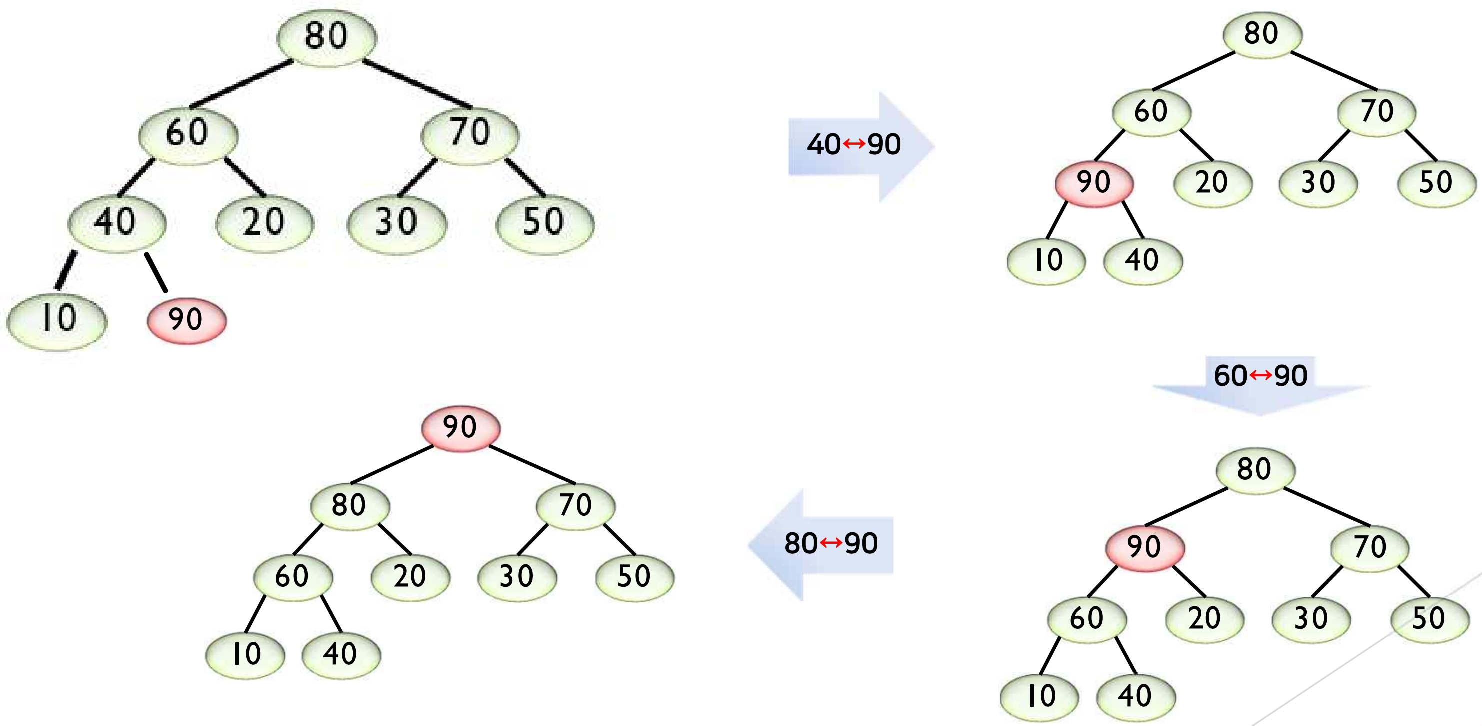
힙의 구현



0
30
1
15
2
22
3
10
4
9
5
20
6
18
7
3
8
7
9
8
10
5
11
10
2i+2
2i+1
i
0 1 2 3 4 5 6 7 8 9 10 11
30 15 22 10 9 20 18 3 7 8 5 10
j
$\left\lfloor \frac{j-1}{2} \right\rfloor$

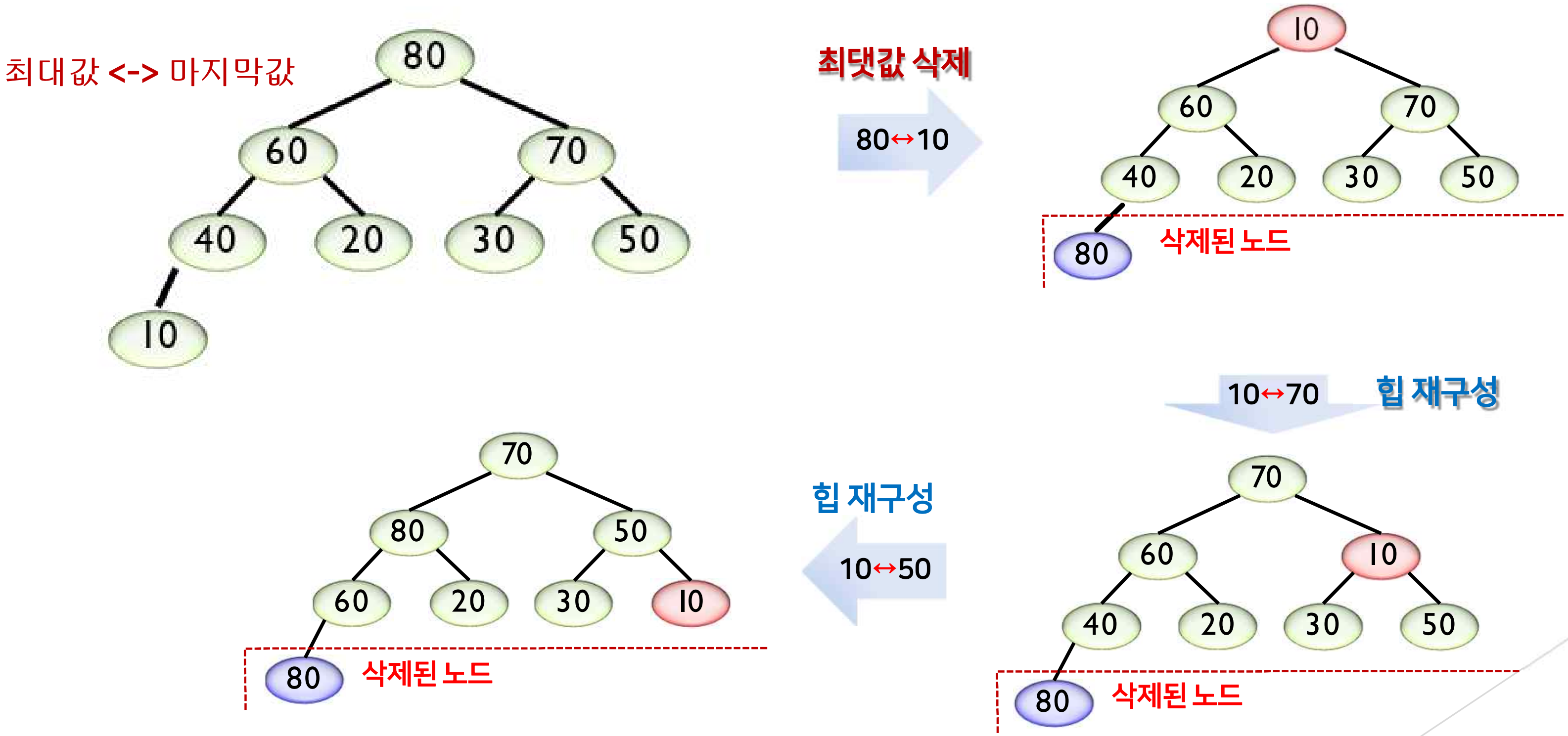
힙의 장점

- 임의의 값 삽입 과정 -> 예) 90넣기

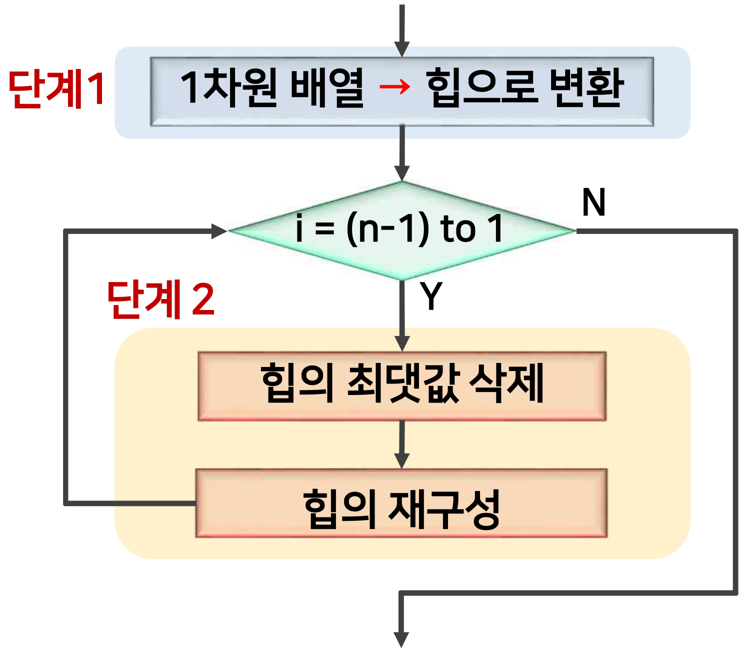


힙의 장점

- 최대값 삭제 과정



힙 정렬의 처리 과정



힙 정렬의 알고리즘

```
HeapSort (A[ ], n)
{
  // ------------- 단계1 ----------------
  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/2⌉-1;
    }
  }
  // ------------- 단계2 ----------------
  return A;
}
```

```
for (i=n-1; i>0; i-- ) {
  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 )
}
```

초기 힙 구축

- 1차원 입력 배열 -> 힙
- 두 가지 접근 방법
- 방법1

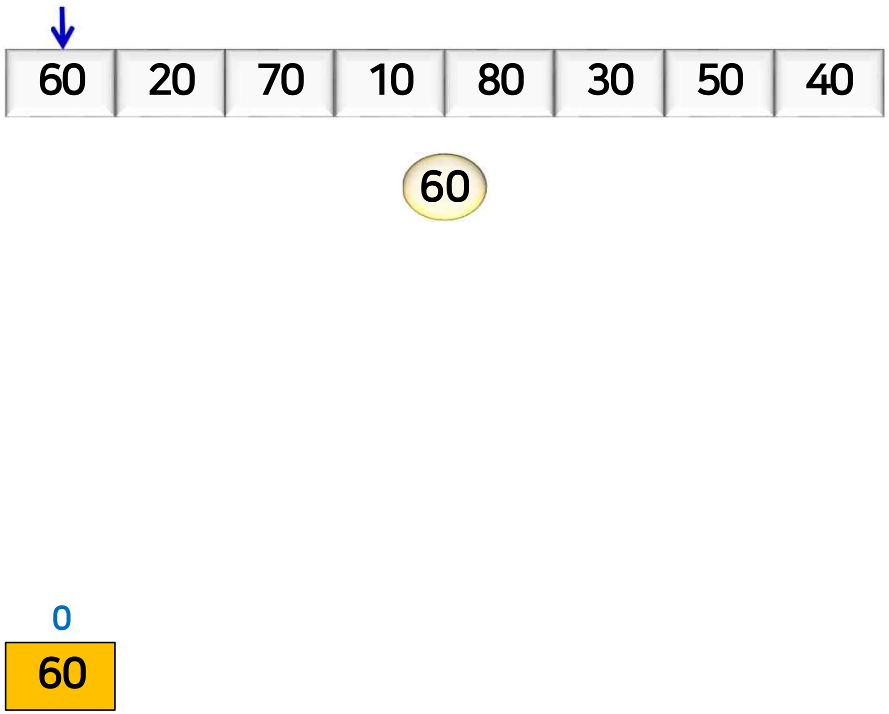
  → 주어진 입력 배열의 각 원소에 대한 힙 에서의 **삽입** 과정을 반복
- 방법2

  → 주어진 입력 배열을 우선 완전 이진 트리로 만든 다음에 **아래에서 위로** 그리 고 **오른쪽에서 왼쪽**으로 진행하면서 각 노드에 대해서 힙의 조건을 만족할 수 있도록 조정

초기 힙 구축_방법1

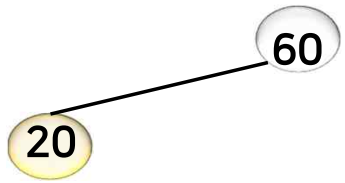
60	20	70	10	80	30	50	40

초기 힙 구축_방법1



초기 힙 구축_방법1

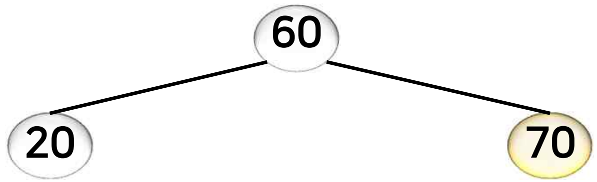
60	20	70	10	80	30	50	40



0	1
60	20

초기 힙 구축_방법1

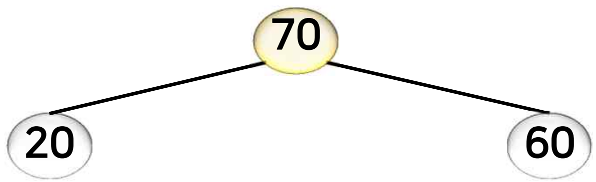
60	20	70	10	80	30	50	40



0	1	2
60	20	70

초기 힙 구축_방법1

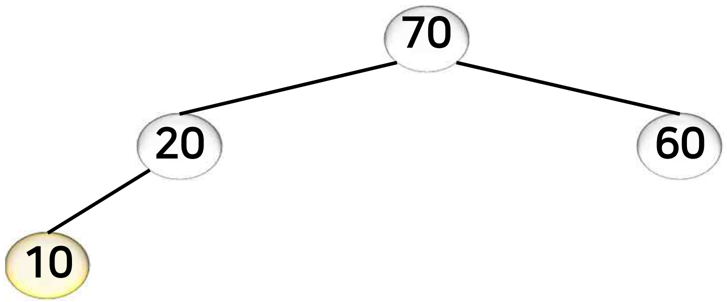
60	20	70	10	80	30	50	40



0	1	2
70	20	60

초기 힙 구축_방법1

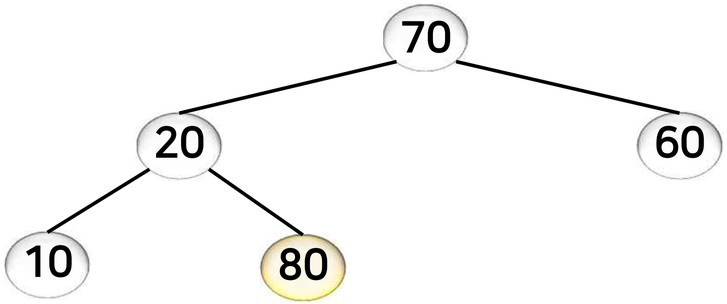
60	20	70	10	80	30	50	40



0	1	2	3
70	20	60	10

초기 힙 구축_방법1

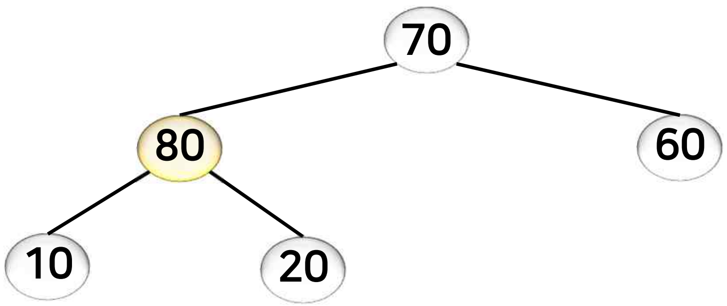
60	20	70	10	80	30	50	40



0	1	2	3	4
70	20	60	10	80

초기 힙 구축_방법1

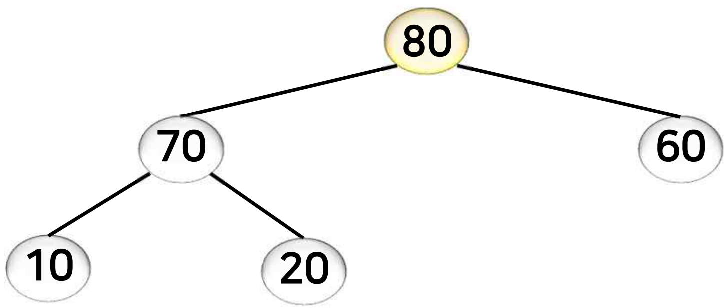
60	20	70	10	80	30	50	40



0	1	2	3	4
70	80	60	10	20

초기 힙 구축_방법1

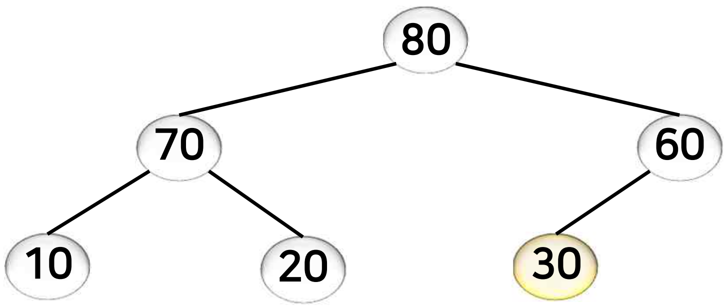
60	20	70	10	80	30	50	40



0	1	2	3	4
80	70	60	10	20

초기 힙 구축_방법1

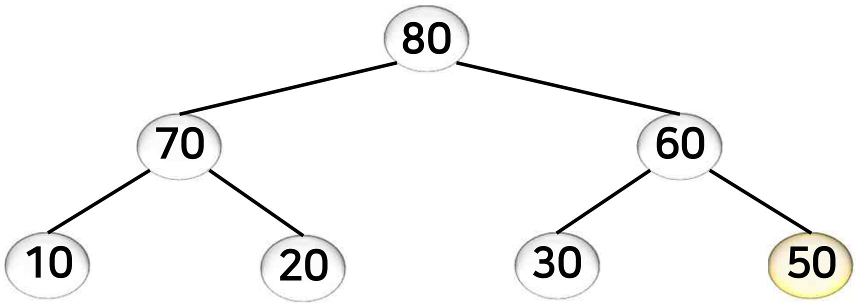
60	20	70	10	80	30	50	40



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

초기 힙 구축_방법1

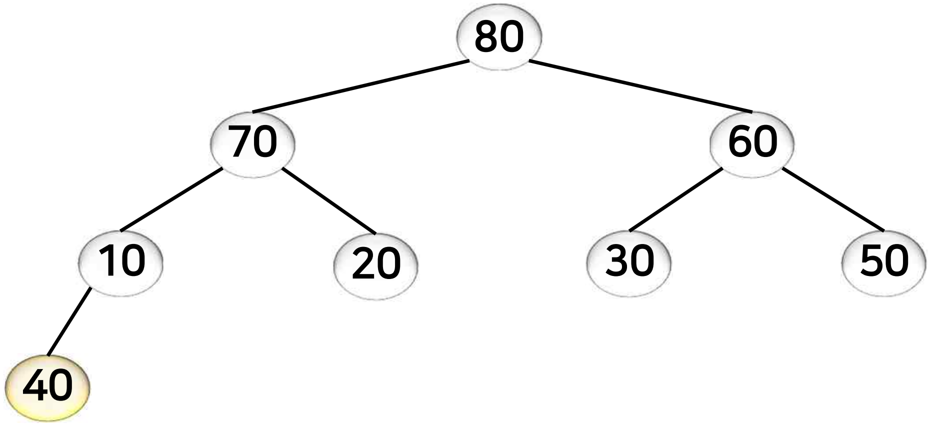
60	20	70	10	80	30	50	40



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

초기 힙 구축_방법1

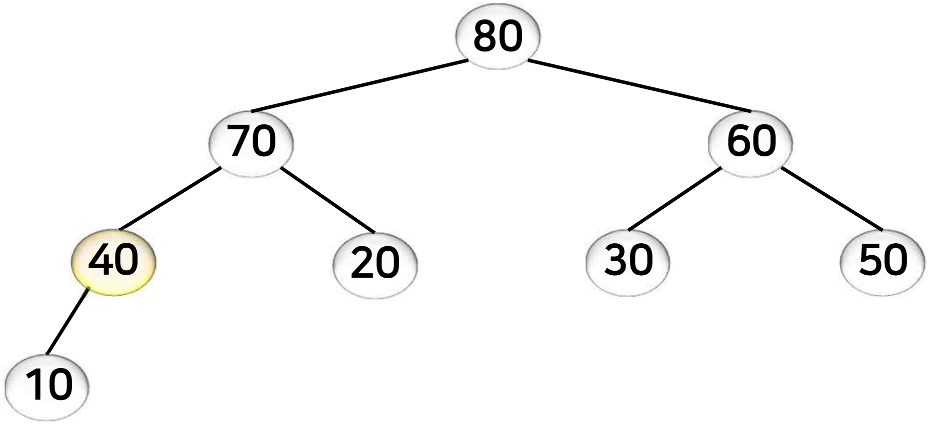
60	20	70	10	80	30	50	40



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

초기 힙 구축_방법1

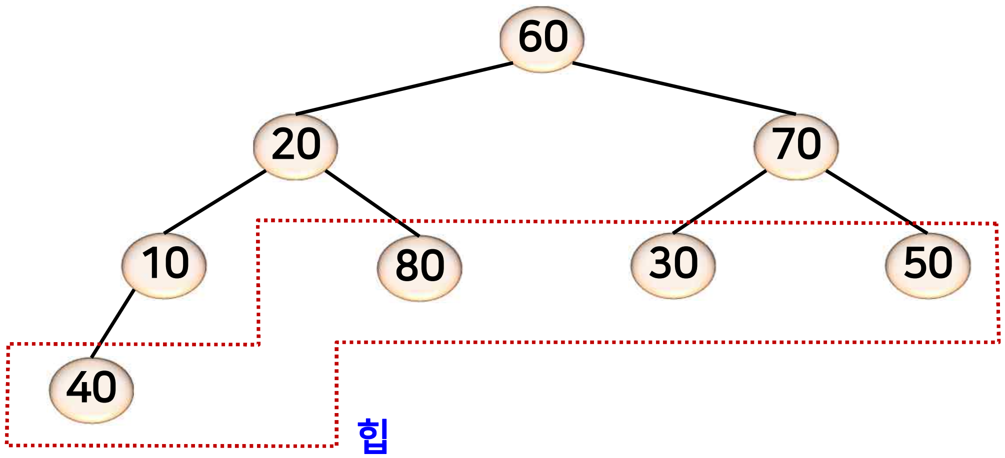
60	20	70	10	80	30	50	40



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

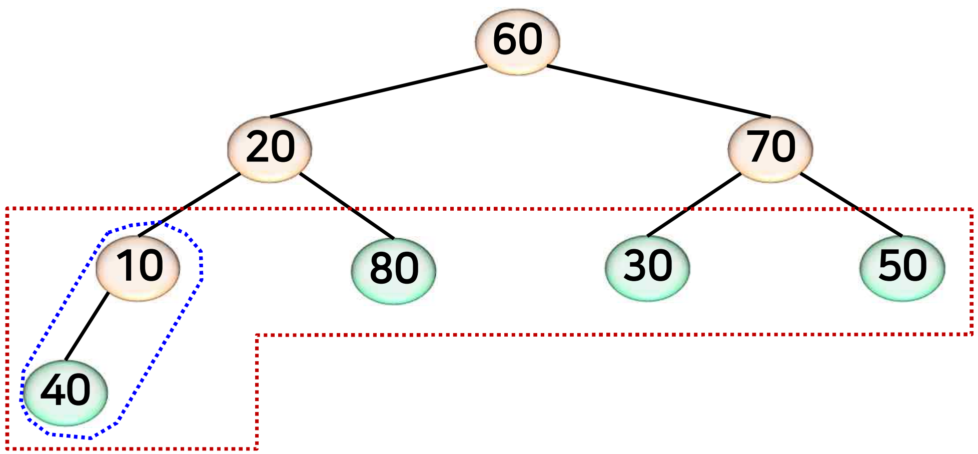
초기 힙 구축_방법2

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



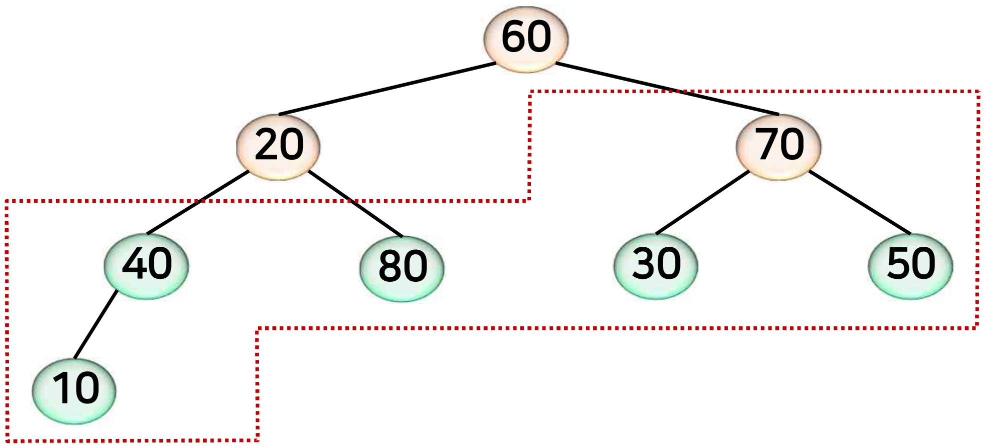
초기 힙 구축_방법2

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



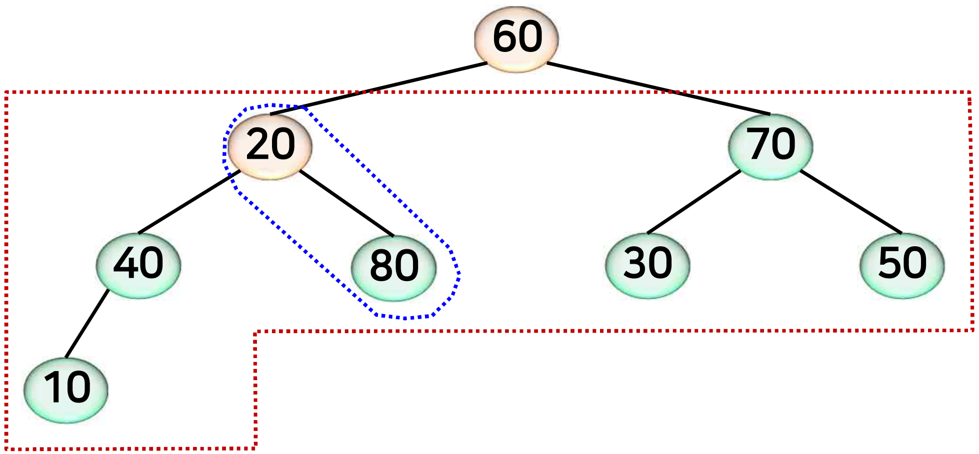
초기 힙 구축_방법2

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



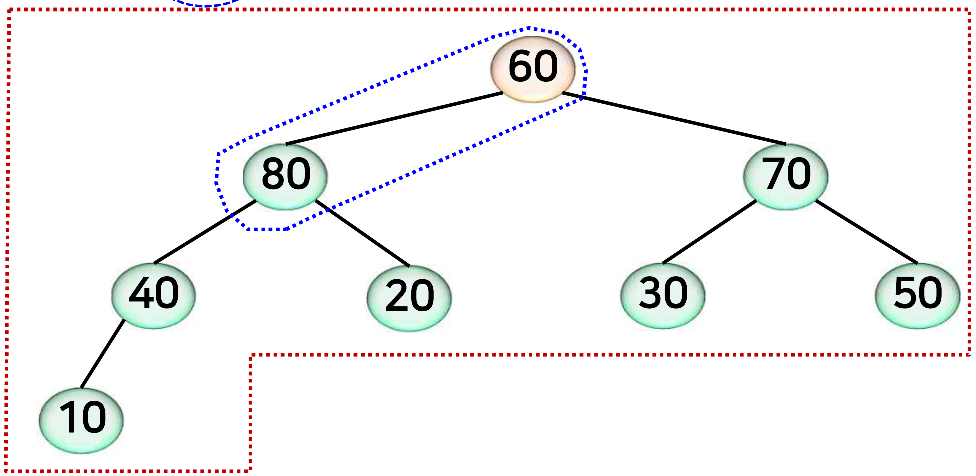
초기 힙 구축_방법2

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



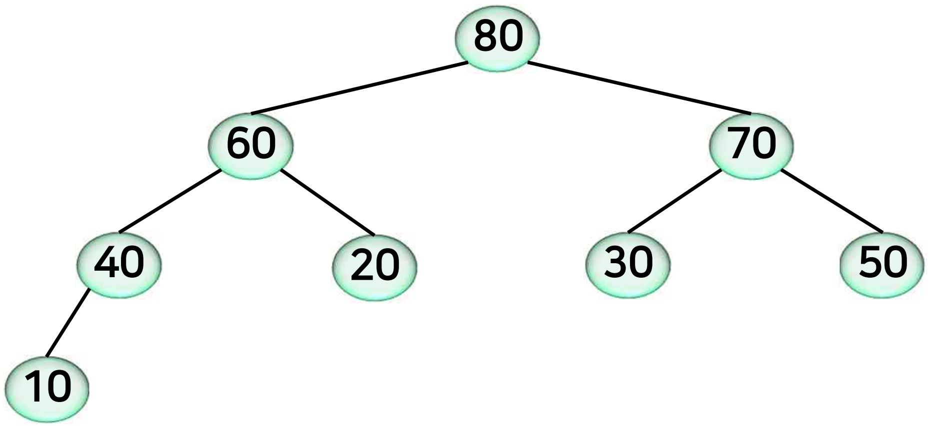
초기 힙 구축_방법2

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



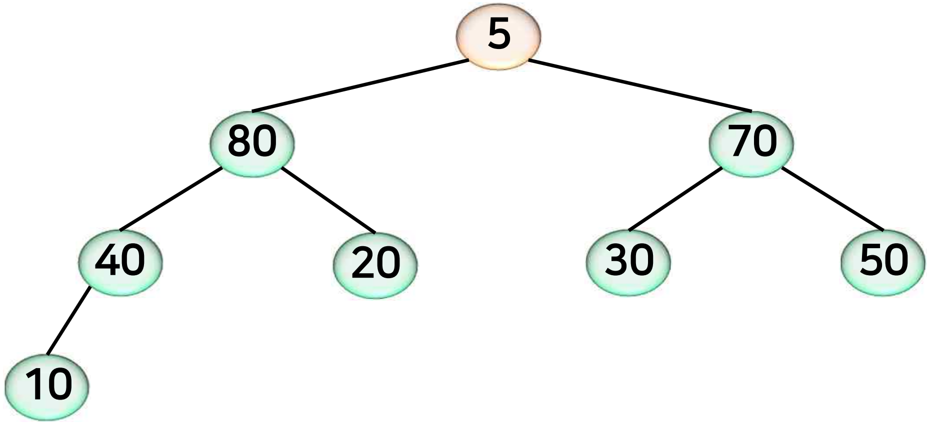
초기 힙 구축_방법2

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



초기 힙 구축_방법2_보충설명

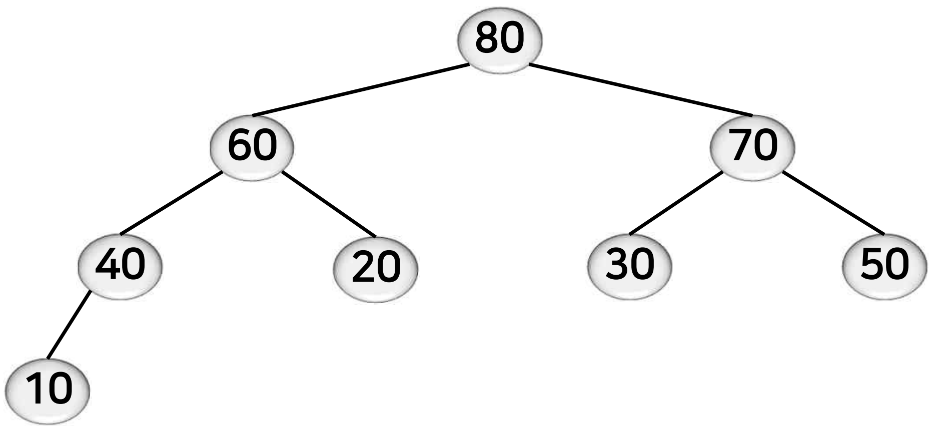
5	80	70	40	20	30	50	10



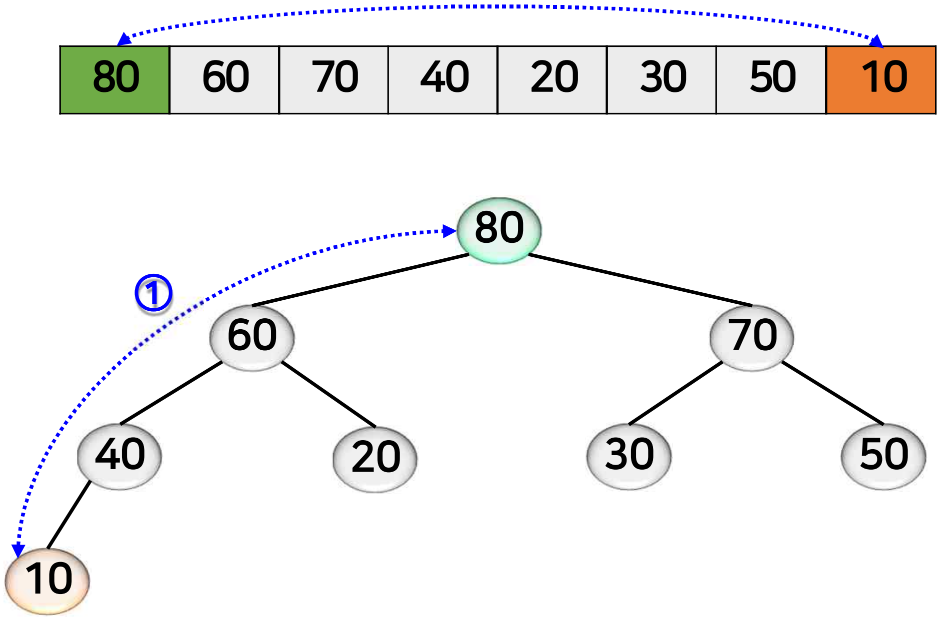
80	40	70	10	20	30	50	5

힙 정렬의 두 번째 단계

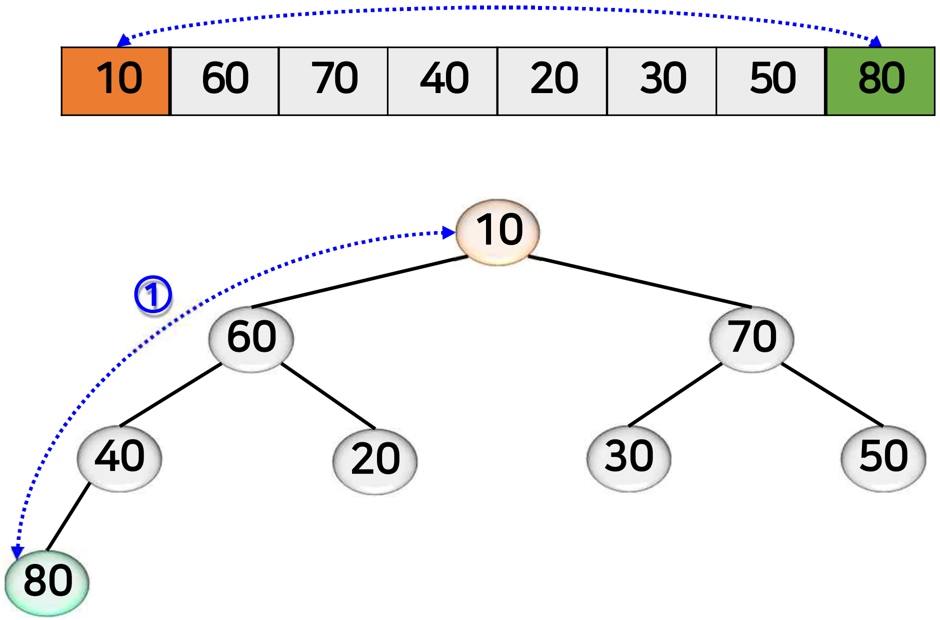
80	60	70	40	20	30	50	10



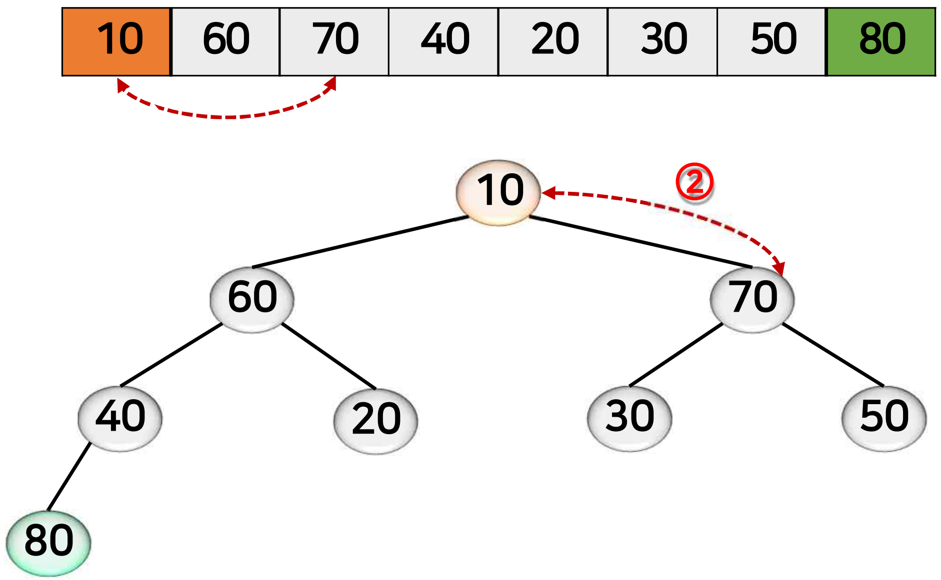
힙 정렬의 두 번째 단계



힙 정렬의 두 번째 단계

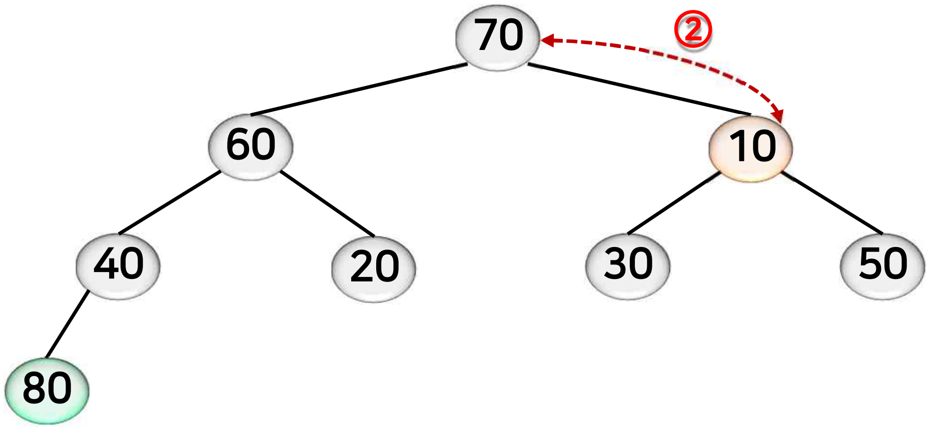


힙 정렬의 두 번째 단계

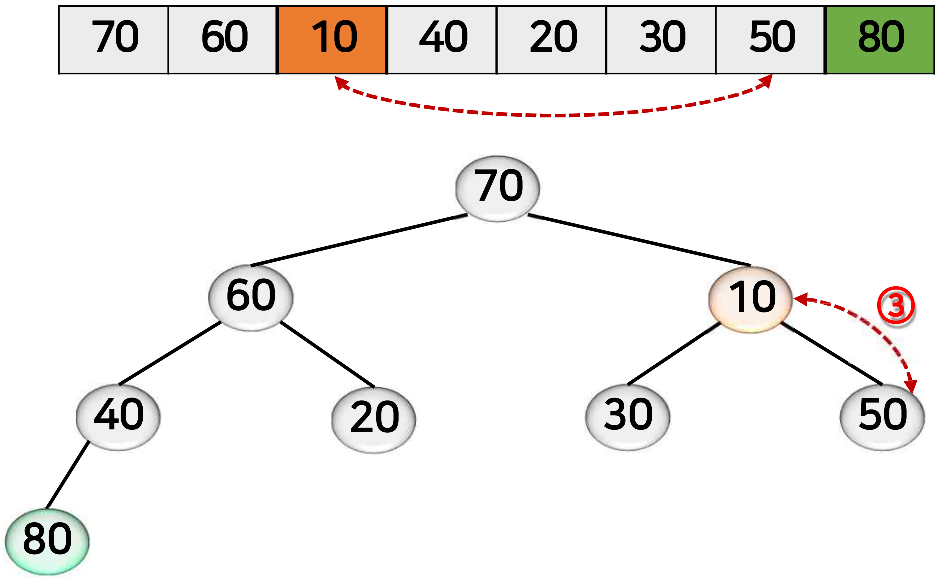


힙 정렬의 두 번째 단계

70	60	10	40	20	30	50	80

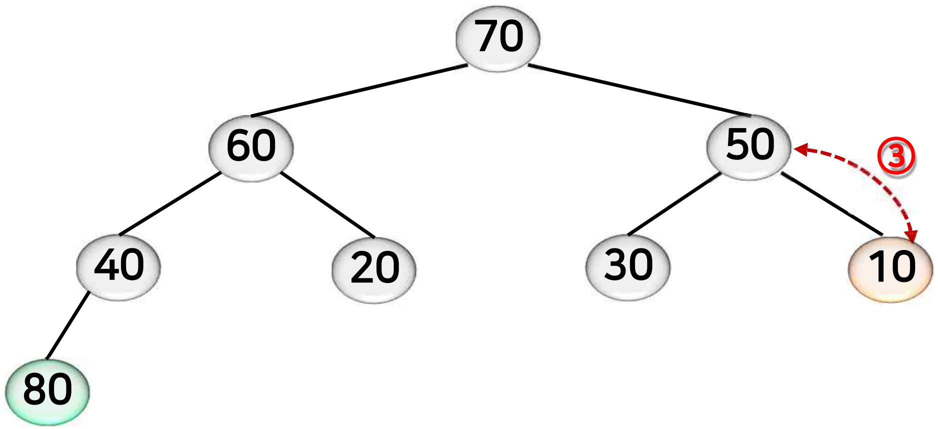


힙 정렬의 두 번째 단계

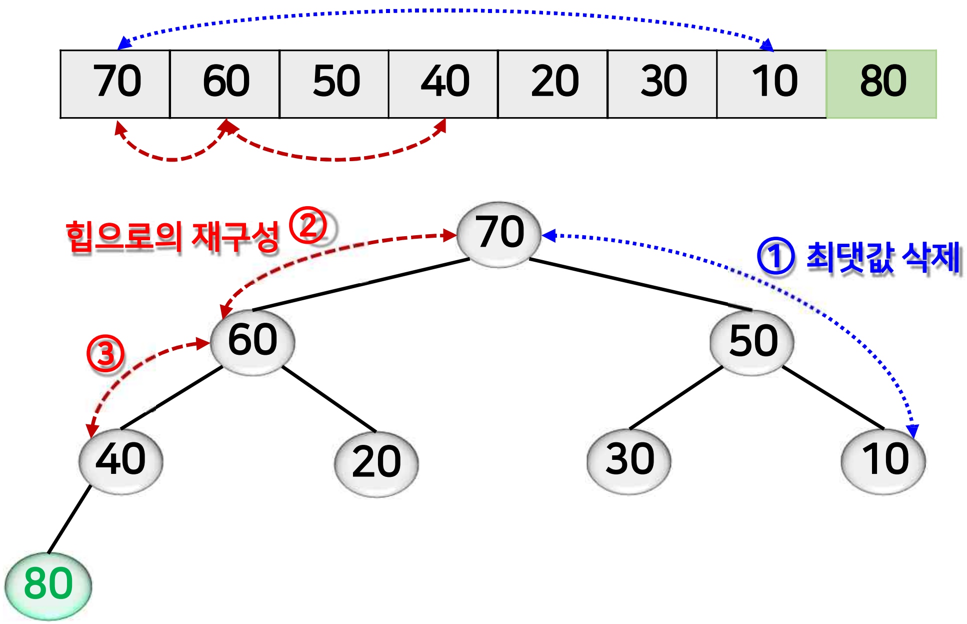


힙 정렬의 두 번째 단계

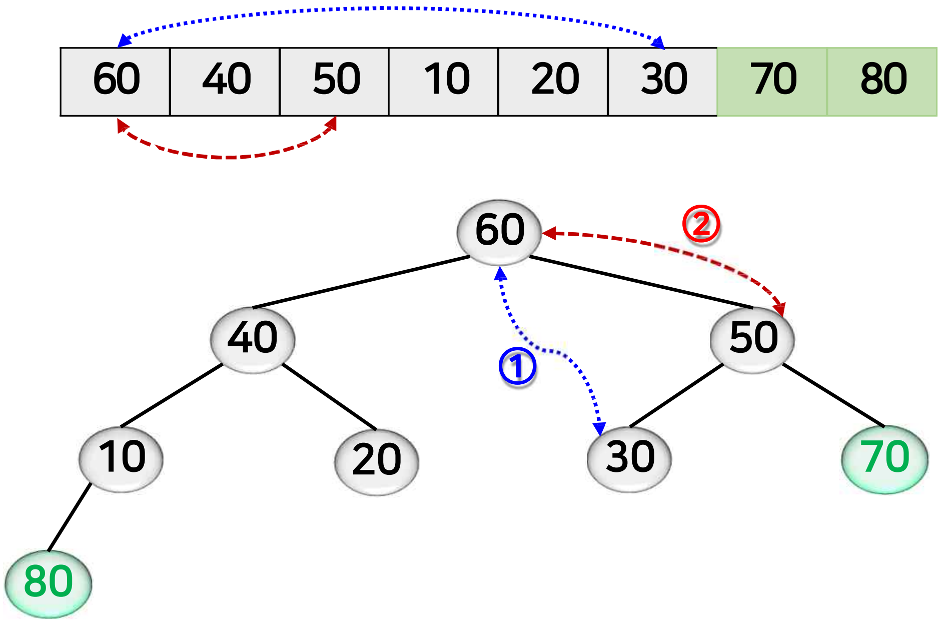
70	60	50	40	20	30	10	80



힙 정렬의 두 번째 단계

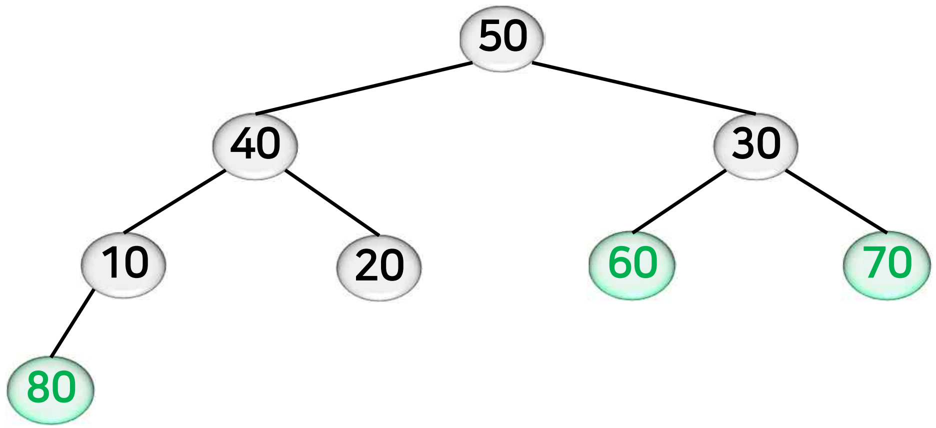


힙 정렬의 두 번째 단계

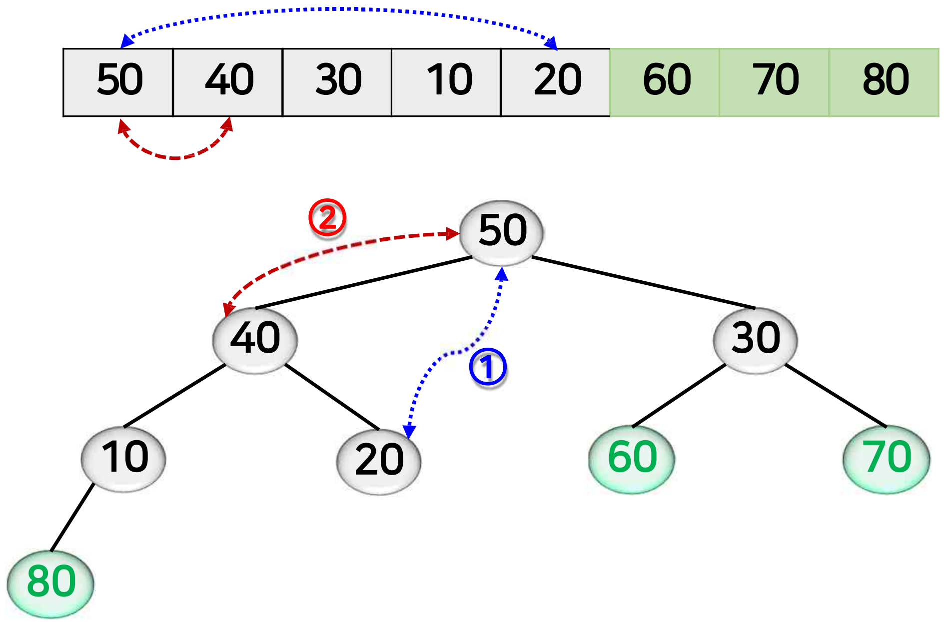


힙 정렬의 두 번째 단계

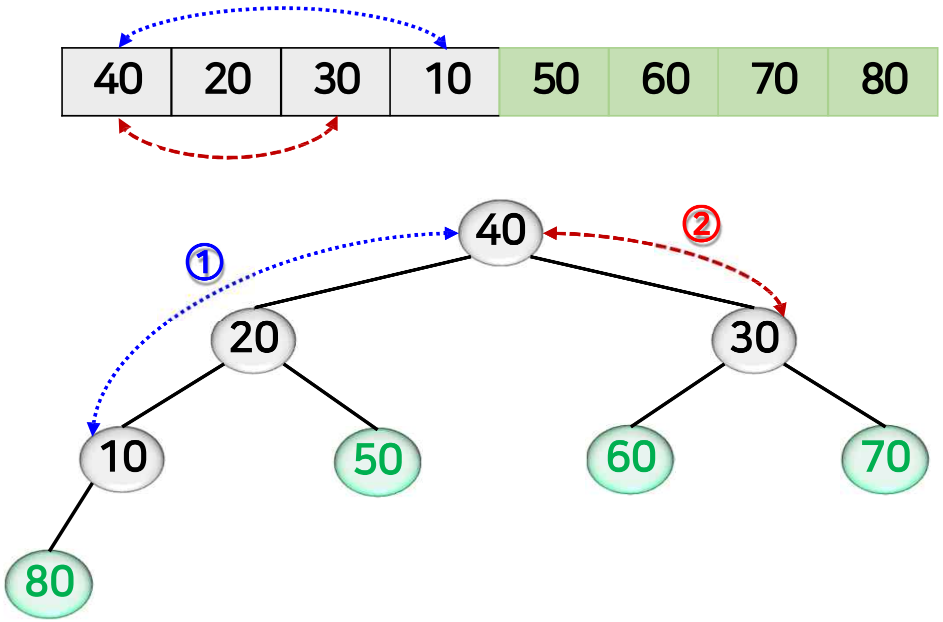
50	40	30	10	20	60	70	80



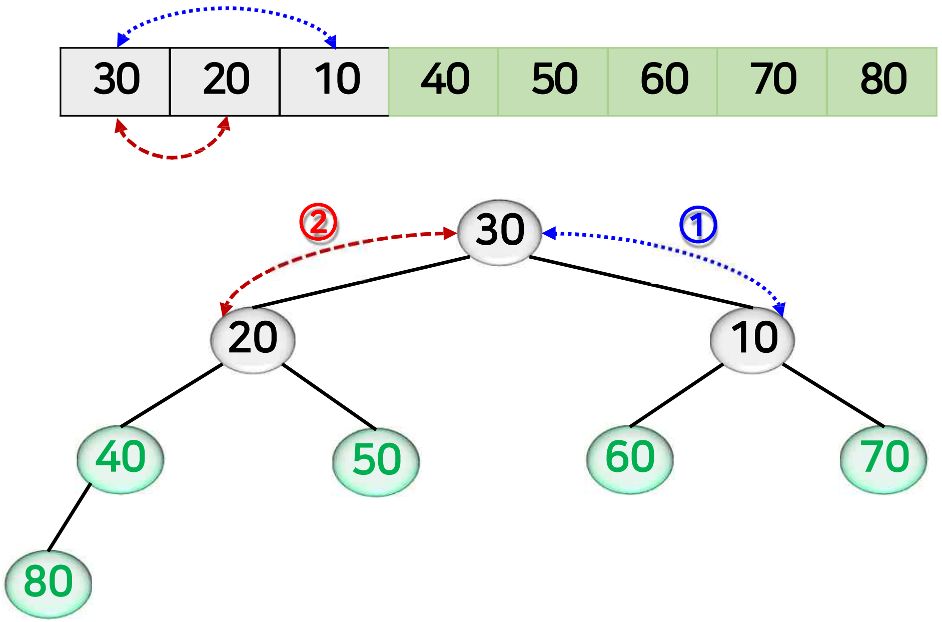
힙 정렬의 두 번째 단계



힙 정렬의 두 번째 단계

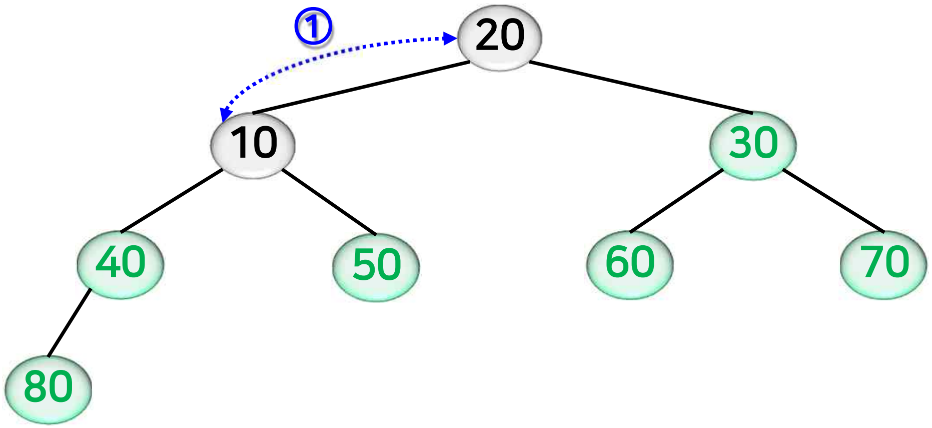


힙 정렬의 두 번째 단계



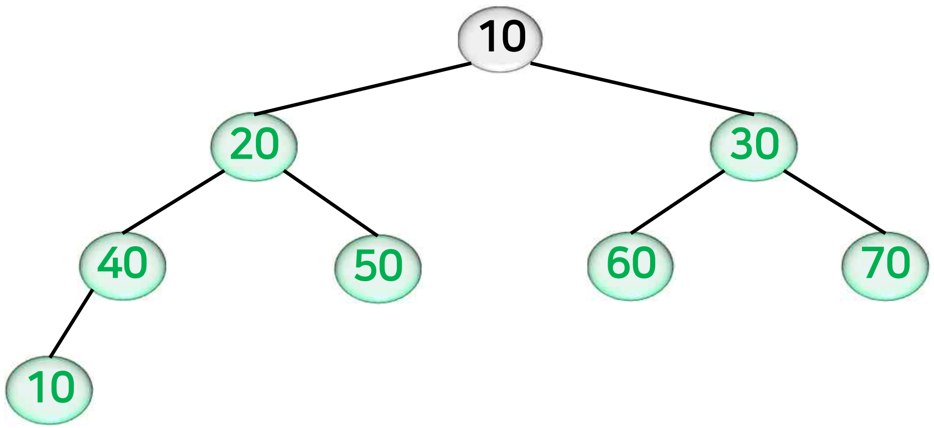
힙 정렬의 두 번째 단계

20	10	30	40	50	60	70	80



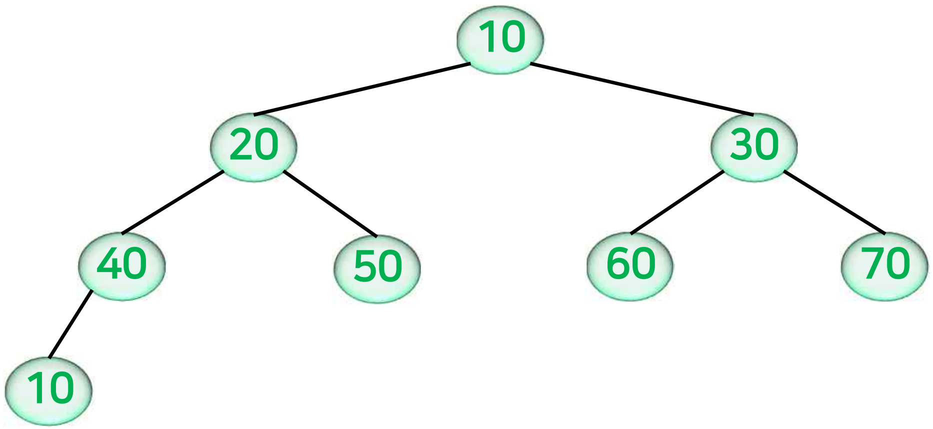
힙 정렬의 두 번째 단계

10	20	30	40	50	60	70	80



힙 정렬의 두 번째 단계

10	20	30	40	50	60	70	80

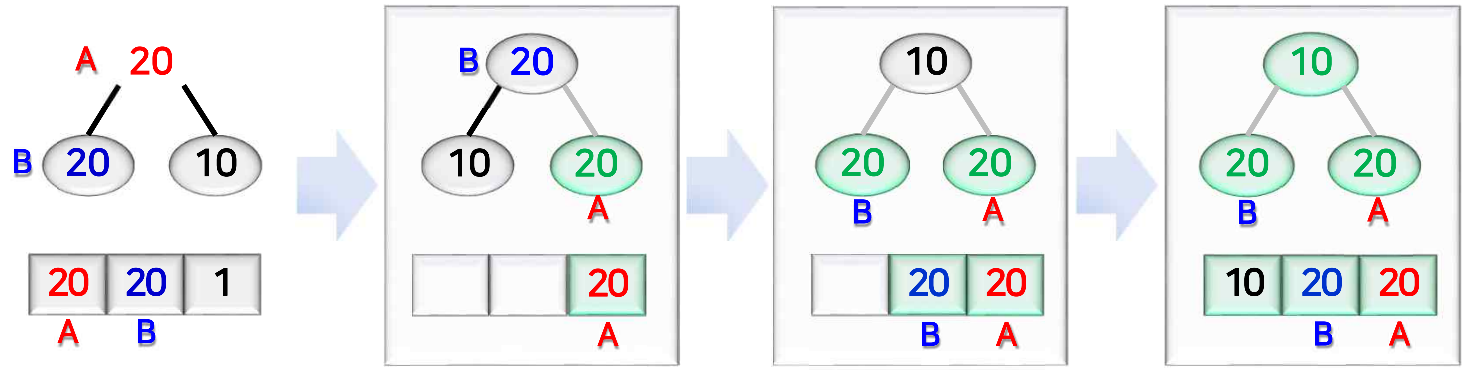


힙 정렬의 성능 분석

- 최선, 최악, 평균의 경우 모두 O(nlogn)
- 초기 힙 생성, 최댓값 삭제 및 힙 재구성
  - 바깥 루프 → 입력 크기 n에 비례
  - 안쪽 루프 → 완전 이진 트리의 높이 logn에 비례

힙 정렬의 특징

- 안정적이지 않은 정렬 알고리즘



- 제자리 정렬 알고리즘

비교 기반 정렬의 하한

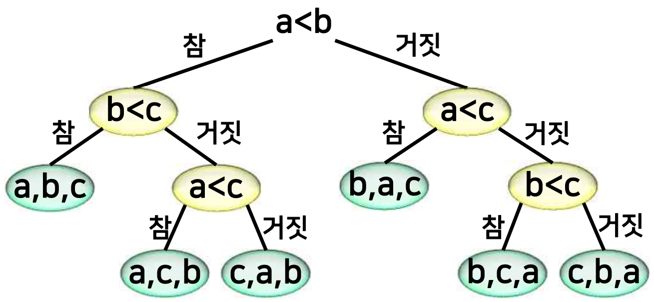
비교 기반 정렬 알고리즘

- 키값과 키값을 직접적으로 비교해서 크고 작음에 따라 순서를 정하는 방식
  - 기본 성능 $O(n^2)$인 알고리즘 → 버블 정렬, 선택 정렬, 삽입 정렬, 셸 정렬
  - 향상된 성능 O(nlogn)인 알고리즘 → 합병 정렬, 퀵 정렬, 힙 정렬

- 비교 기반으로서 O(nlogn)보다 더 효율적인 성능의 정렬 알고리즘을 개발할 수 있는가?
  - 비교 기반 정렬 알고리즘의 **하한**이 O(nlogn)인가?
    - 하한 lower bound → 최소한으로 필요한 성능, 가장 좋은 성능

비교 기반 정렬 알고리즘의 특징

- n=3인 세 개의 다른 숫자 a, b, c를 정렬하는 **결정 트리**



n개의 서로 다른 값을 정렬

→ 정확히 n!개의 리프 노드를 갖는 결정 트리

n개의 서로 다른 값을 정렬하는 데 필요한 키의 비교 횟수

n!개의 리프 노드를 가진 결정 트리의 높이

m개의 리프 노드를 가진 이진 트리의 높이 $h \geq \lceil \log m \rceil$

n개의 서로 다른 값을 정렬하는 비교 기반 알고리즘의 최소의 비교 횟수 $\lceil \log(n!) \rceil$

$\log(n!) \geq n\log n - 1.45n$

비교 기반 정렬 알고리즘의 하한 O(nlogn)

계수 정렬

계수 정렬

- 비교 기반이 아닌 데이터의 분포를 이용한 정렬
  - 계수 counting 정렬, 기수 radix 정렬 → 선형 시간 O(n)

- 주어진 원소 중에서 자신보다 작거나 같은 값을 갖는 원소의 개수를 계산하여counting 정렬할 위치를 찾아 정렬하는 방식
  - 입력값이 어떤 작은 정수 범위 내에 있다는 것을 알고 있는 경우에만 적용
  - 입력값의 범위 a~b에 해당하는 크기의 배열 COUNT [a .. b]를 할당하고, 각 입력값을 한 번 훑으면서 각 입력값의 출현 횟수의 누적값을 계산하여 이용

계수 정렬의 알고리즘

```
CountingSort (A[ ], 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;
}
```

입력: A[1..n] : 입력 배열
n : 입력 크기
출력: B[1..n] : 정렬된 배열

계수 정렬의 적용 예

입력값의 범위: 4~9

A							
7	5	9	8	4	5	7	5

COUNT

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

각 값의 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 적용 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

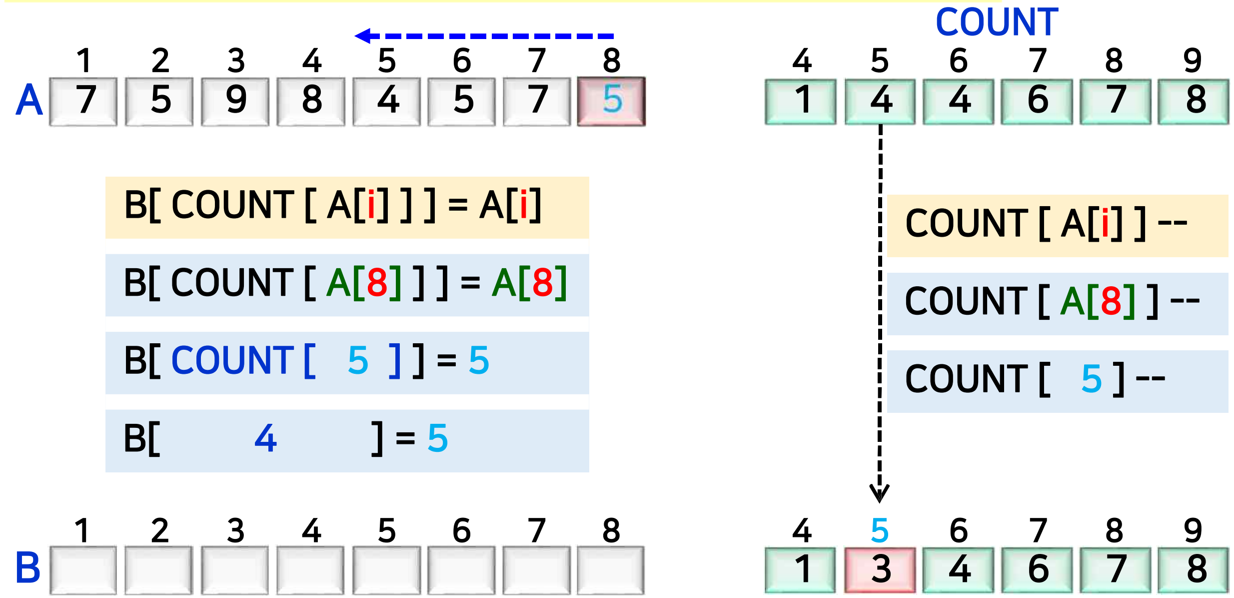
입력값의 출현 횟수의 누적값 계산: COUNT[i] = COUNT[i] + COUNT[i-1]

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

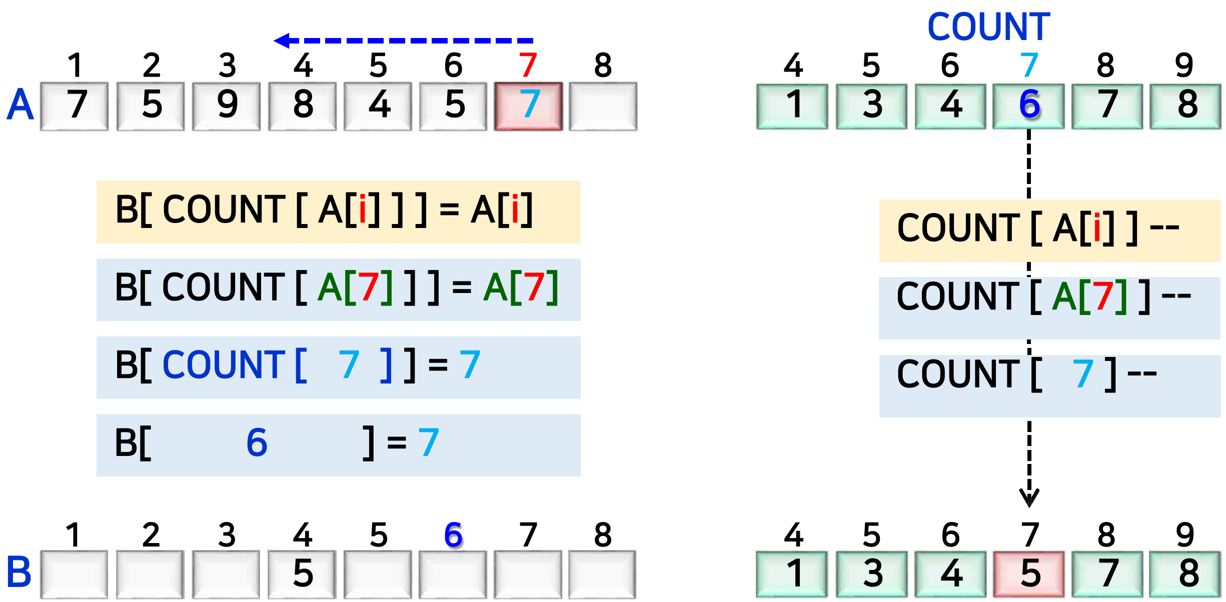
1	4	4	6	7	8

계수 정렬의 적용 예

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



계수 정렬의 적용 예



COUNT
1 2 3 4 5 6 7 8
A 7 5 9 8 4 5 7
4 5 6 7 8 9
1 3 4 6 7 8
B[COUNT [A[i]]] = A[i]
B[COUNT [A[7]]] = A[7]
B[COUNT [7]] = 7
B[6] = 7
COUNT [A[i]] --
COUNT [A[7]] --
COUNT [7] --
1 2 3 4 5 6 7 8
B 5
4 5 6 7 8 9
1 3 4 5 7 8

계수 정렬의 적용 예

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

계수 정렬의 적용 예


1 2 3 4 5 6 7 8
A 7 5 9
B 4 5 5 7 8
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
B 4 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
정렬 결과
B 4 5 5 5 7 8 9
4 5 6 7 8 9
0 1 4 5 6 7
0 1 4 4 6 7

계수 정렬의 성능분석

```
CountingSort (A[ ], 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)

k = MAX-MIN+1

O(k)

O(n+k) → 입력값의 범위 k가 입력 크기 n에 비례하면 → O(n)

계수 정렬

- 입력값의 범위가 입력 원소의 개수보다 작거나 비례할 때 유용
  - 입력값의 범위를 k라고 할 때 O(n+k) 시간
    → 보통 k=O(n)일 때 사용하므로 결국 선형 시간 O(n)을 가짐

- 안정적인 정렬 알고리즘

- 제자리 정렬 알고리즘이 아님
- 보편적이지 못한 방법
  - 입력값의 범위를 알아야 함
  - 추가적인 배열이 필요 → COUNT[], B[]

기수 정렬

기수 정렬의 개념과 원리

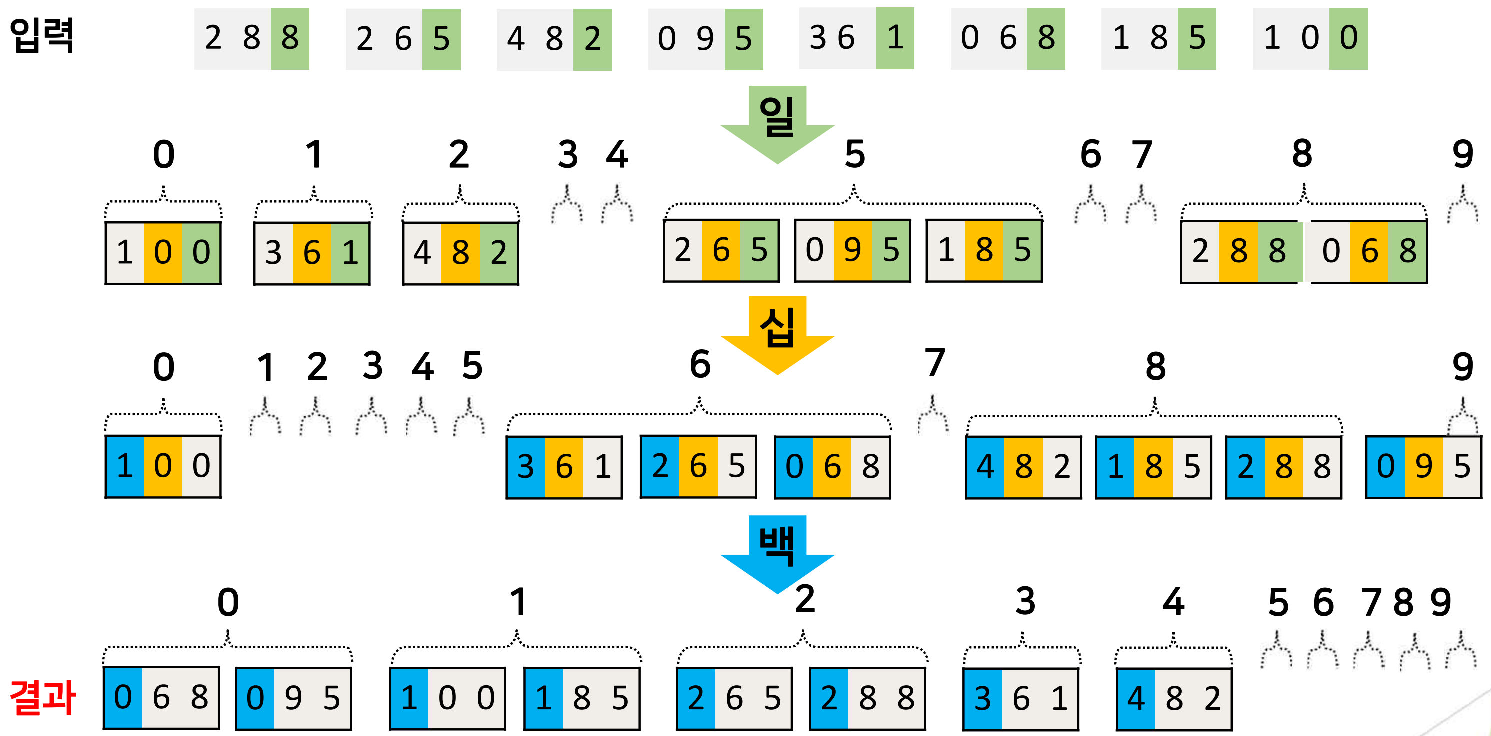
- 입력값을 자릿수별로 부분적으로 비교하는 정렬 방식
  - 주어진 원소의 키값을 자릿수별로 나누고, 각 자릿수별로 계수 정렬과 같은 **안정적인 정렬 알고리즘**을 반복적으로 적용하여 정렬

  - **LSD Least Significant Digit 기수 정렬** → 낮은 자리부터 높은 자리로 진행
  - MSD Most Significant Digit 기수 정렬 → 높은 자리부터 낮은 자리로 진행

기수 정렬의 알고리즘

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

LSD 기수 정렬의 적용 예



기수 정렬의 성능 분석

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

계수 정렬을 사용하면 O(n)

O(dn)

d가 입력 크기 n보다 매우 작으면

O(n)

기수 정렬의 특징

- 입력 원소의 값의 자릿수가 상수일 때 유용
  - d 자릿수 n개의 숫자들에 대해 계수 정렬을 적용하면 **O(dn)**

    → 여기서 d를 상수로 간주하면 **O(n)**
- 안정적 정렬 알고리즘
- 제자리 정렬 알고리즘이 아님
  - 계수 정렬 → 전체 데이터 개수와 진법 크기만큼의 추가적인 공간이 필요

정렬 알고리즘의 비교

방식	알고리즘	실행시간	안정적	제자리
비교 기반	버블	$O(n^2)$	○	○
	선택	$O(n^2)$	×	○
	삽입	$O(n^2)$	○	○
	셸	$O(n^2)$	×	○
	합병	$O(n\log n)$	○	×
	퀵	$O(n^2)$, $O(n\log n)$	×	○
	힙	$O(n\log n)$	×	○
값의 분포 기반	계수	$O(n)$	○	×
	기수	$O(n)$	○	×

과제 안내

과제

- **힙 정렬, 계수 정렬 문제 구현하기**
  - e-Class 업로드
- 양식 (한글, 워드,PDF -> 자유)
- 파일명 (이름_학번_전공)
  - 예) 최희석_2014182009_게임공학

- 질의 응답은 e-Class 질의응답 게시판에 남겨 주시길 바랍니다.