# Chapter 6-2

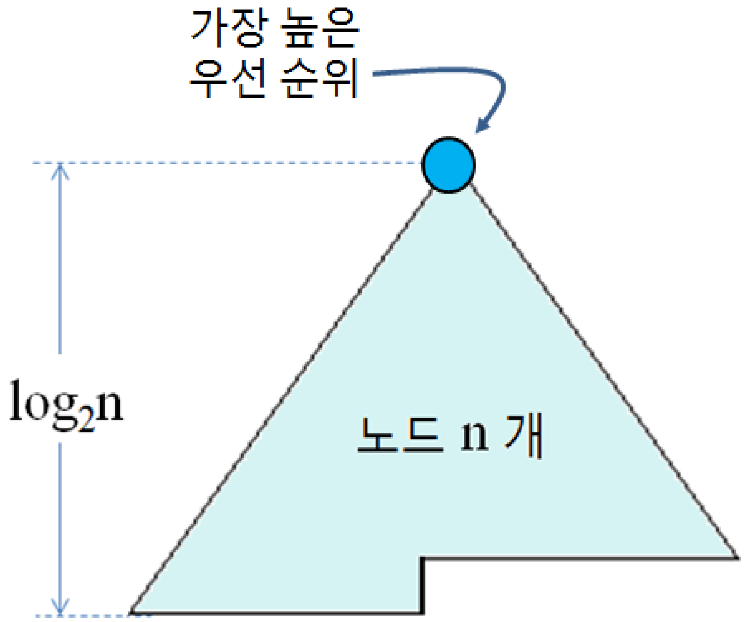
## Sorting Algorithms

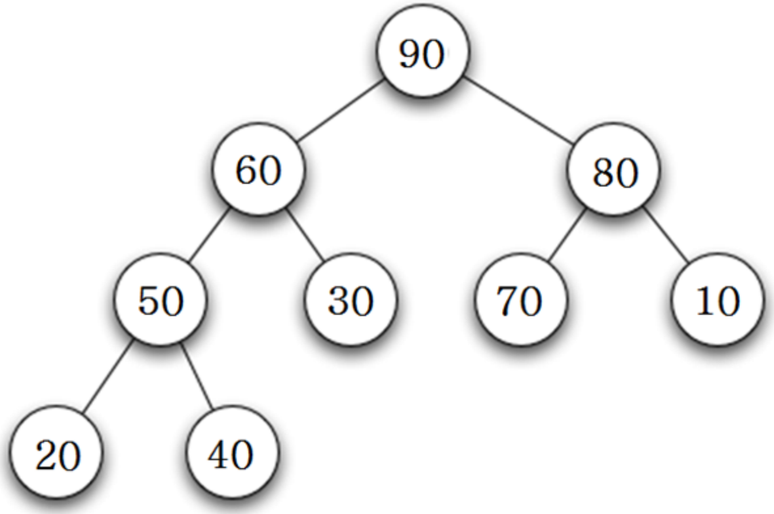
# Heap Sort

- 이진 힙 (Binary Heap)
  - 힙 조건을 만족하는 완전 이진 트리 (Complete Binary Tree)
  - 힙 조건: 각 노드의 우선 순위 (priority)가 자식 노드의 우선 순위보다 높음
  - 최대 힙 (Maximum Heap): 가장 큰 값이 root에 저장
  - 최소 힙 (Minimum Heap): 가장 작은 값이 root에 저장

# Heap Sort

- 이진 힙 (Binary Heap)
  - n개의 노드를 가진 힙
    - 완전 이진 트리이므로, 힙의 높이가 $\log_2 n$이며, 노드들을 빈 공간 없이 배열에 저장




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| – | 90 | 60 | 80 | 50 | 30 | 70 | 10 | 20 | 40 |

# Heap Sort

- Heap의 노드 관계
  - 힙에서 부모와 자식의 관계
    - A[i]의 부모 = A[i/2]
      - 단, i가 홀수이면 i/2에서 정수 부분만
    - A[i]의 왼쪽 자식 = A[2i]
    - A[i]의 오른쪽 자식 = A[2i+1]

# Heap Sort

- Heap Sort
  - 정렬할 입력으로 최대 힙 (Max heap)을 만듦
  - 힙 루트에 가장 큰 수가 있으므로, 루트와 힙의 가장 마지막 노드를 교환함
    - 가장 큰 수를 배열의 맨 뒤로 옮긴 것
  - 힙 크기를 1개 줄임
  - 루트에 새로 저장된 숫자로 인해 위배된 힙 조건을 해결하여 힙 조건을 만족시킴
  - 이 과정을 반복하여 정렬함

# Heap Sort

- 알고리즘

HeapSort

입력: 입력이 A[1]부터 A[n]까지 저장된 배열 A

출력: 정렬된 배열 A

```
1. A의 숫자에 대해서 최대 힙을 구성
2. heapSize = n        // 힙의 크기를 조절하는 변수
3. for i = 1 to n-1
4.     A[i] ↔ A[heapSize]      // 루트와 힙의 마지막 노드 교환
5.     heapSize = heapSize – 1     // 힙의 크기 1 감소
6.     DownHeap()                  // 위배된 힙 조건을 만족시킴
7. return A
```

# Heap Sort

- BuildHeap()

```
BuildHeap (A) {
   for (i= floor[n/2] to 1)
     DownHeap(i)
}
```

- Downheap()
  - 루트로부터 자식들 중에서 큰 값을 가진 자식과 비교하여 힙 속성이 만족될 때까지 숫자를 교환하며 잎의 방향으로 진행

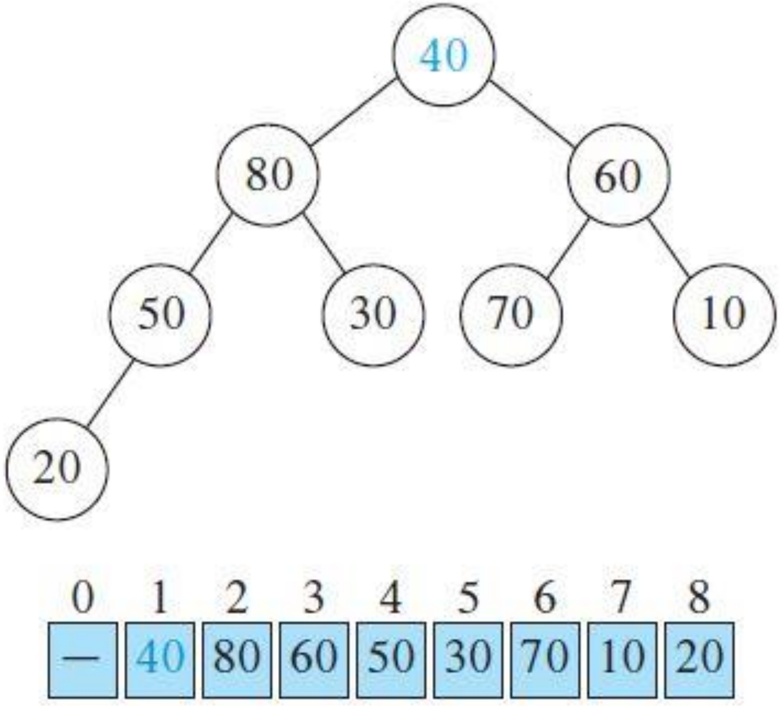
# Heap Sort

- DownHeap(i)

```
leftChild = 2i          // i의 왼쪽 자식 노드
rightChild = 2i+1       // i의 오른쪽 자식 노드
If ((leftChild <= n) and (A[leftChild] > A[i])
        bigger = leftChild
else
        bigger = i
If ((rightChild <= n) and (A[rightChild] > A[bigger])
        bigger = rightChild
If (bigger != i) {
        A[i] ↔ A[bigger]
        DownHeap(bigger)
}
```

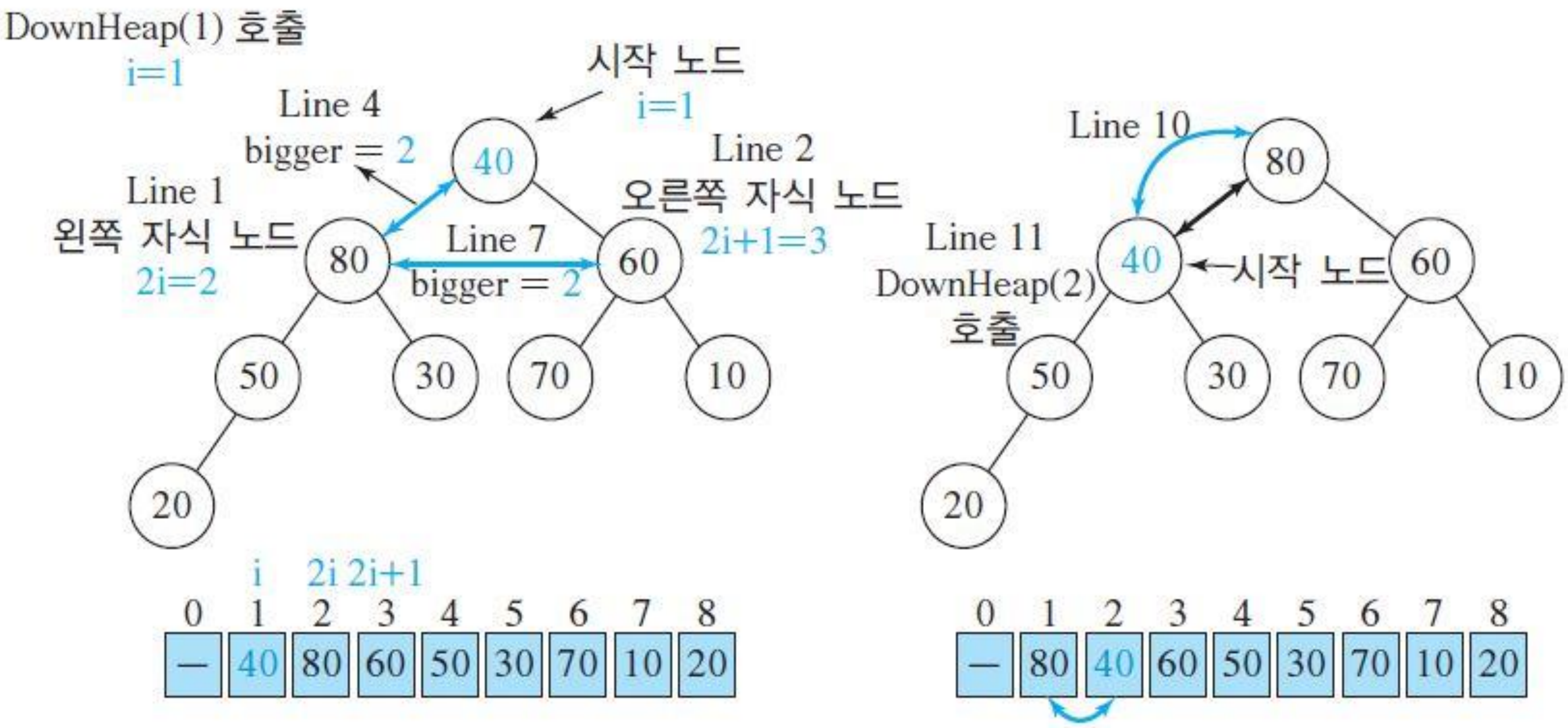
# Heap Sort

- Downheap() 예제



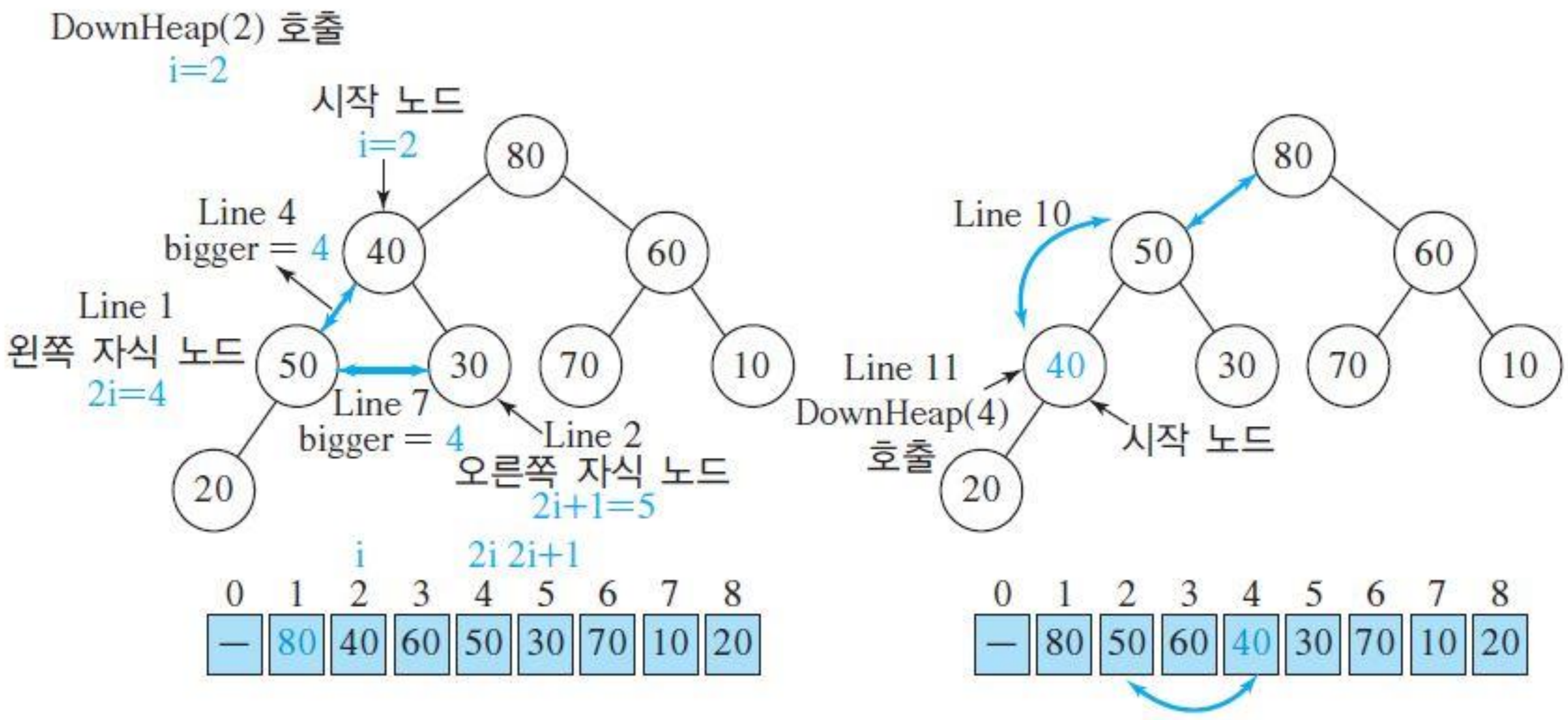
# Heap Sort

- Downheap() 예제



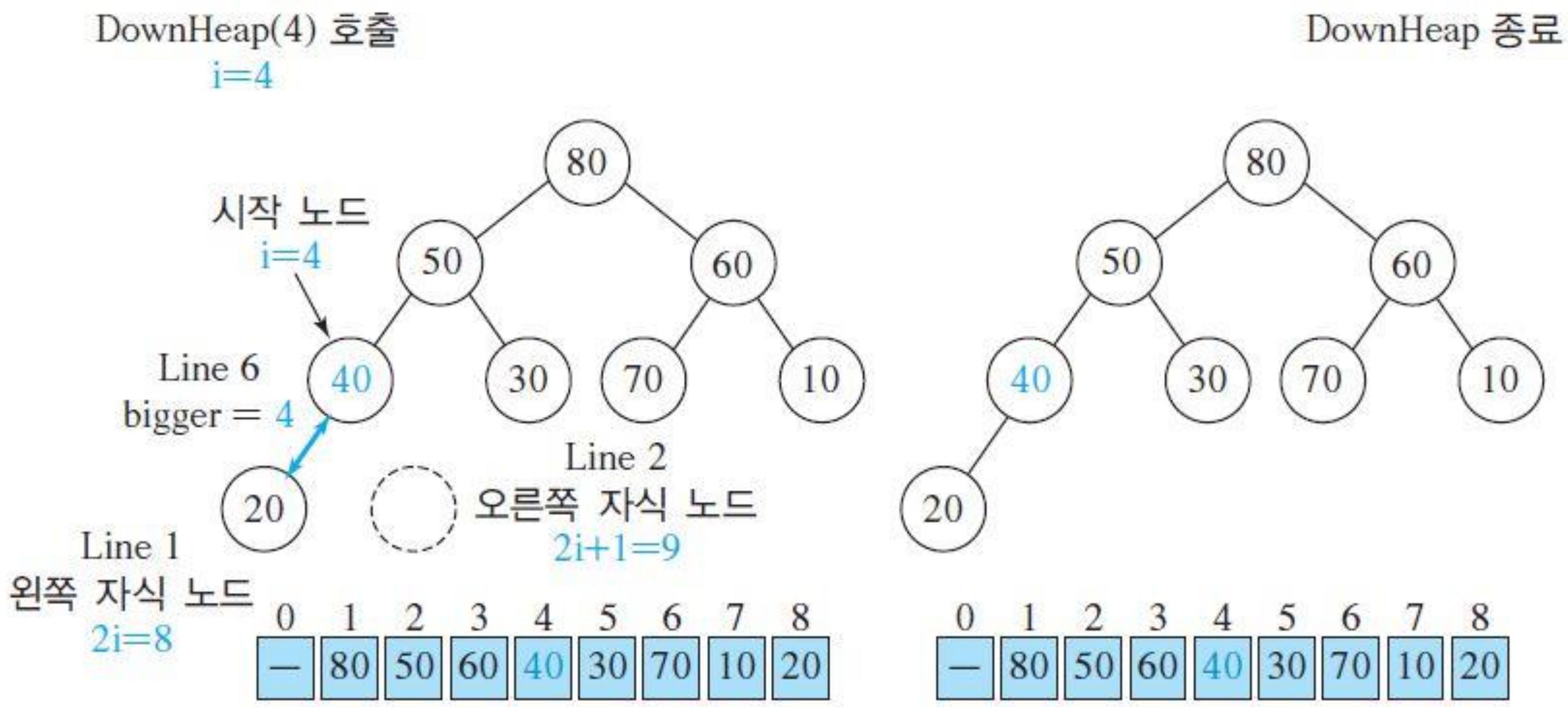
# Heap Sort

- Downheap() 예제



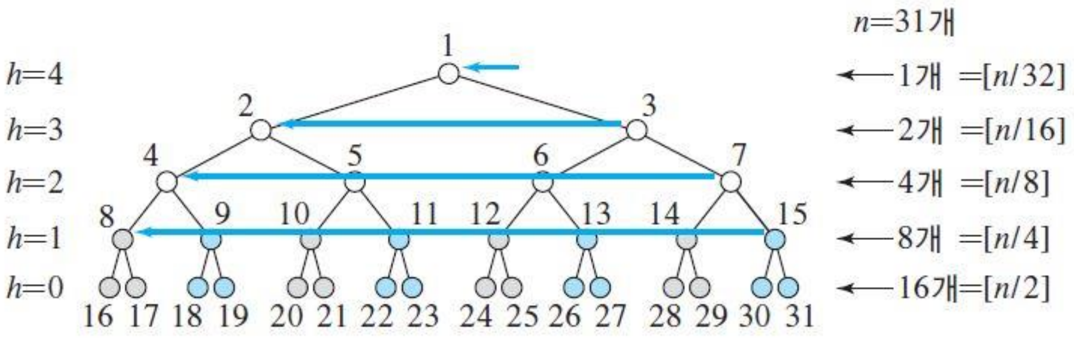
# Heap Sort

- Downheap() 예제



# Heap Sort

- Time Complexity of BuildHeap




[부록 그림-3]

- Time complexity = (각 층에 있는 노드 수) X (층 높이)

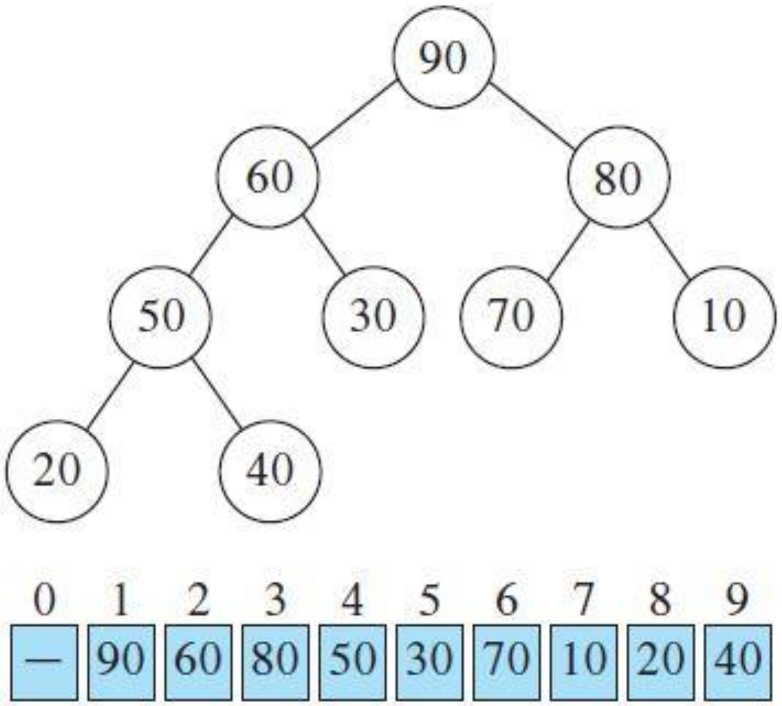
$$= \sum_{h=0}^{\log n} h \frac{n}{2^{h+1}} = \mathrm{n} \sum_{h=0}^{\log n} h \frac{1}{2^{h+1}} \leq \frac{n}{4} \sum_{h=0}^{\infty} h \frac{1}{2^{h-1}}$$

$$= \frac{n}{4} \sum_{h=0}^{\infty} h x^{h-1} \text{ (x=1/2) } = \frac{n}{4} \frac{d}{dh} \sum_{h=0}^{\infty} x^{h} = \frac{n}{4} \frac{d}{dh} \left( \frac{1}{1-x} \right)$$

$$= \frac{n}{4} \frac{1}{(1-x)^2} = \frac{n}{4} \frac{1}{(1-1/2)^2} = n \Rightarrow \mathrm{O(n)}$$

# Heap Sort

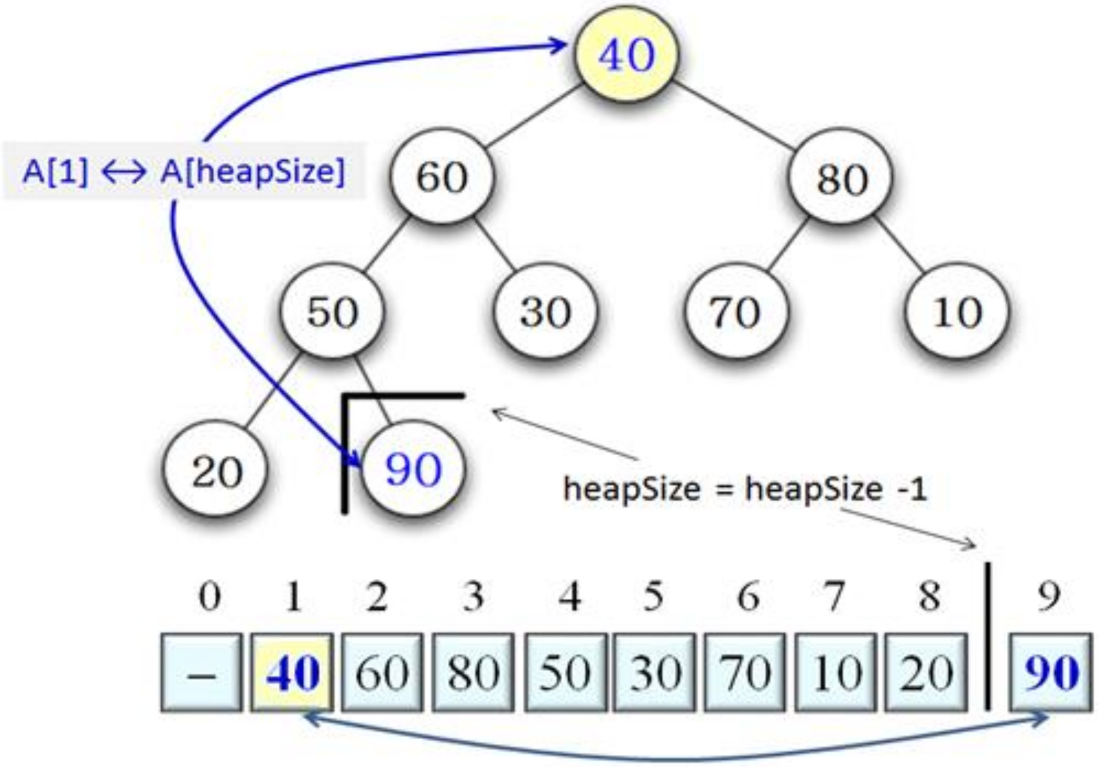
- Heap Sort 알고리즘 수행 과정




[그림 6-2]

# Heap Sort

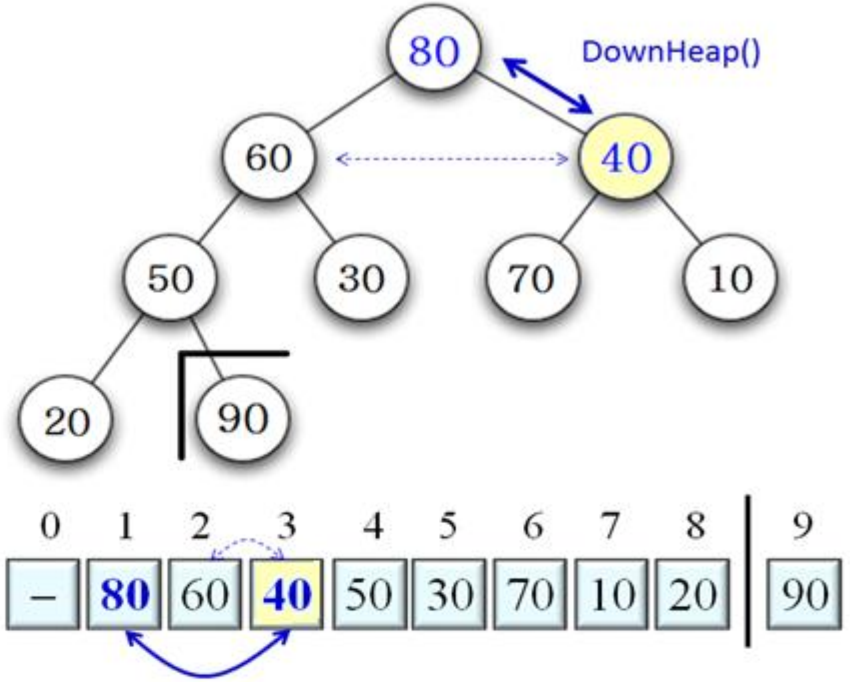
- Heap Sort 알고리즘 수행 과정




- 힙의 마지막 노드 40과 루트 90을 바꾸고, 힙의 노드수(heapSize) 1 감소

# Heap Sort

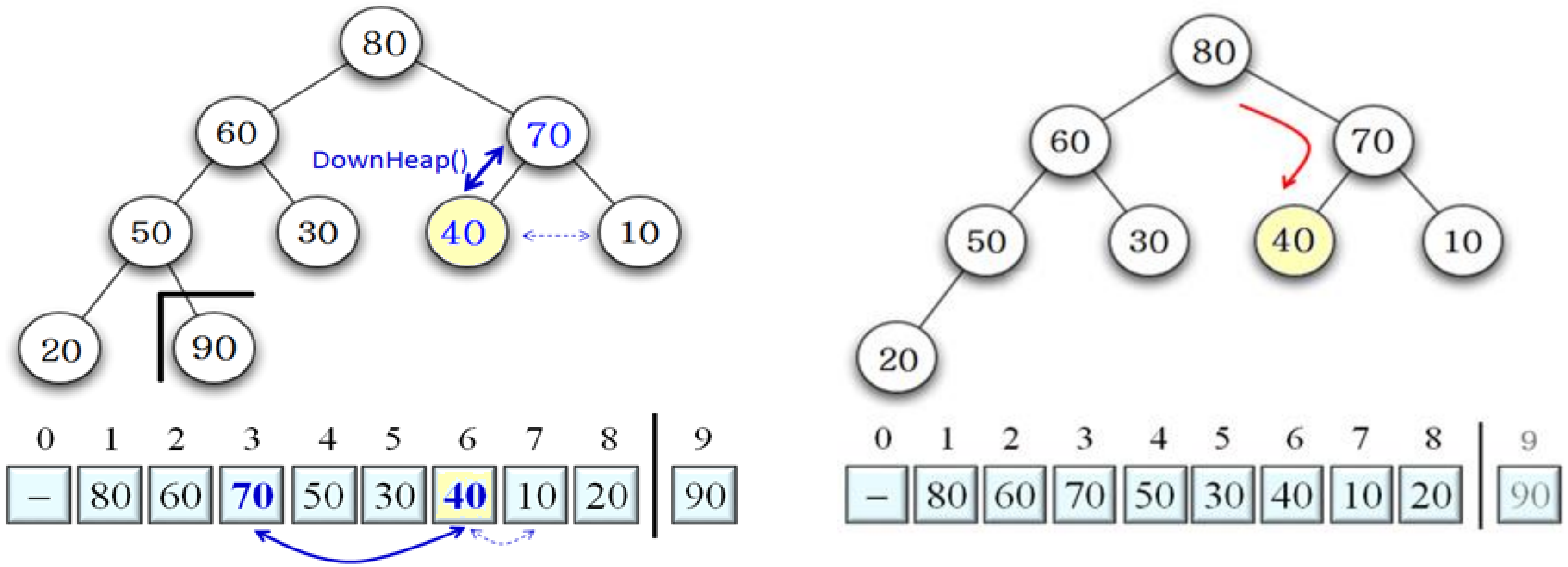
- Heap Sort 알고리즘 수행 과정
  - DownHeap()




  - 새롭게 루트에 저장된 40이 루트의 자식 노드 60과 80보다 작으므로 자식 중에서 큰 자식 80과 루트 40을 교환

# Heap Sort

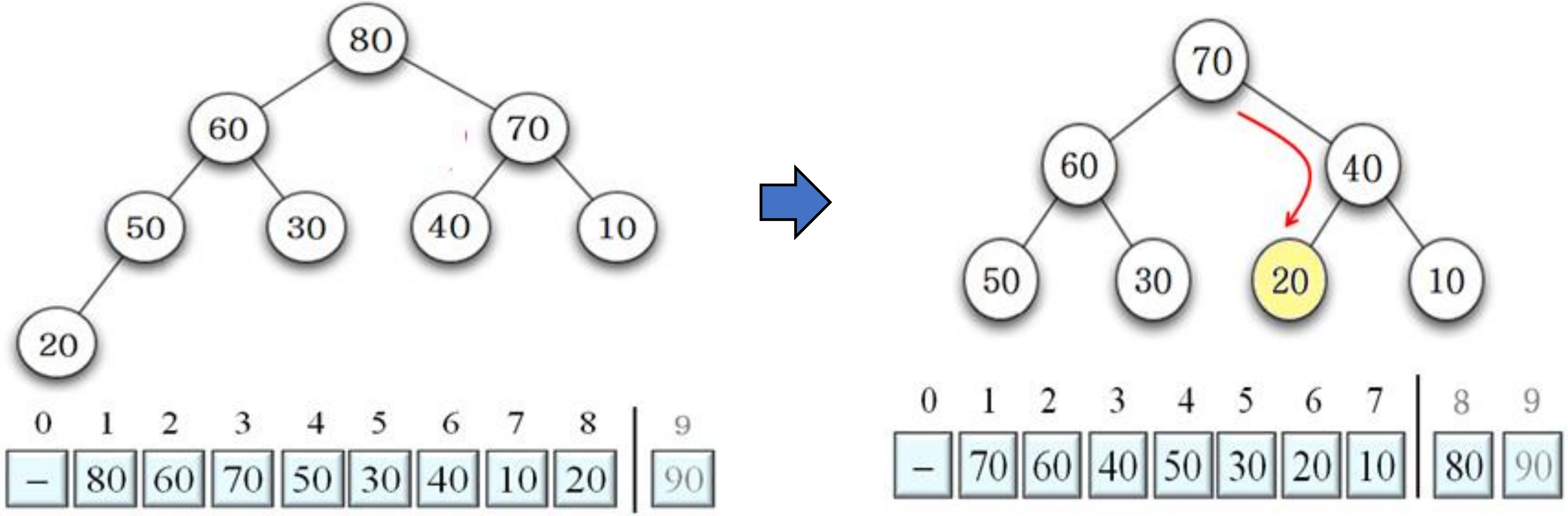
- Heap Sort 알고리즘 수행 과정
  - DownHeap()




  - 40은 다시 자식 노드 70과 10 중에서 큰 자식 70과 비교하여, 70과 40을 교환

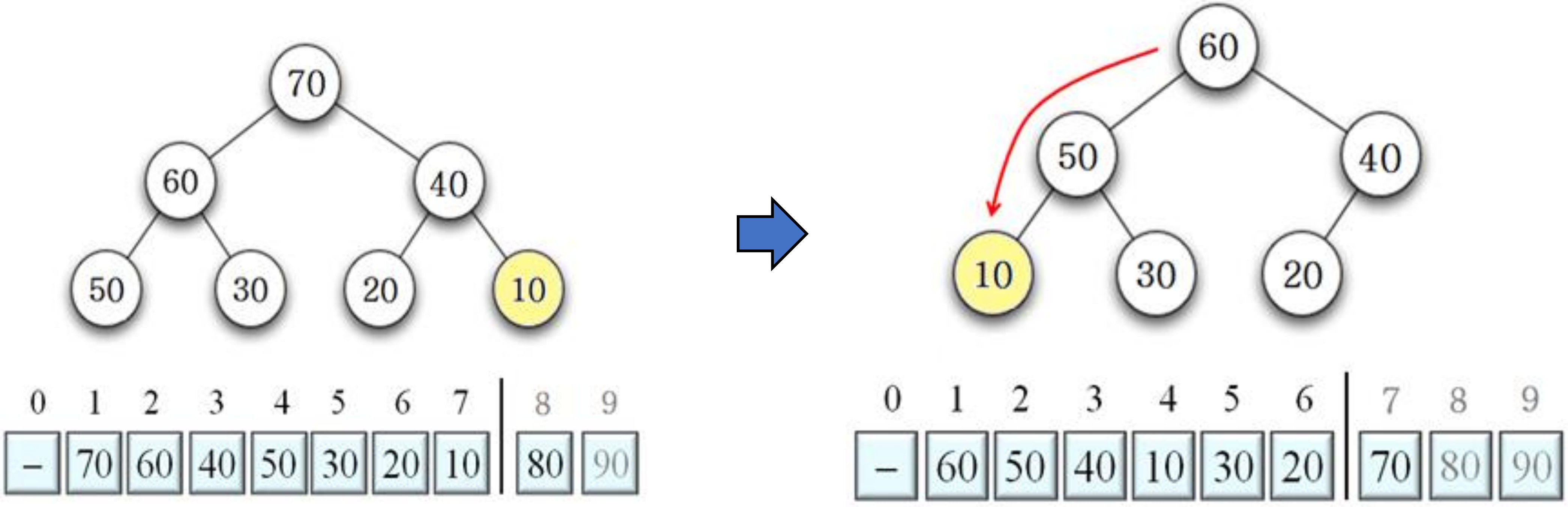
# Heap Sort

- Heap Sort 알고리즘 수행 과정
  - 80 ↔ 20 교환 후 DownHeap 수행 결과



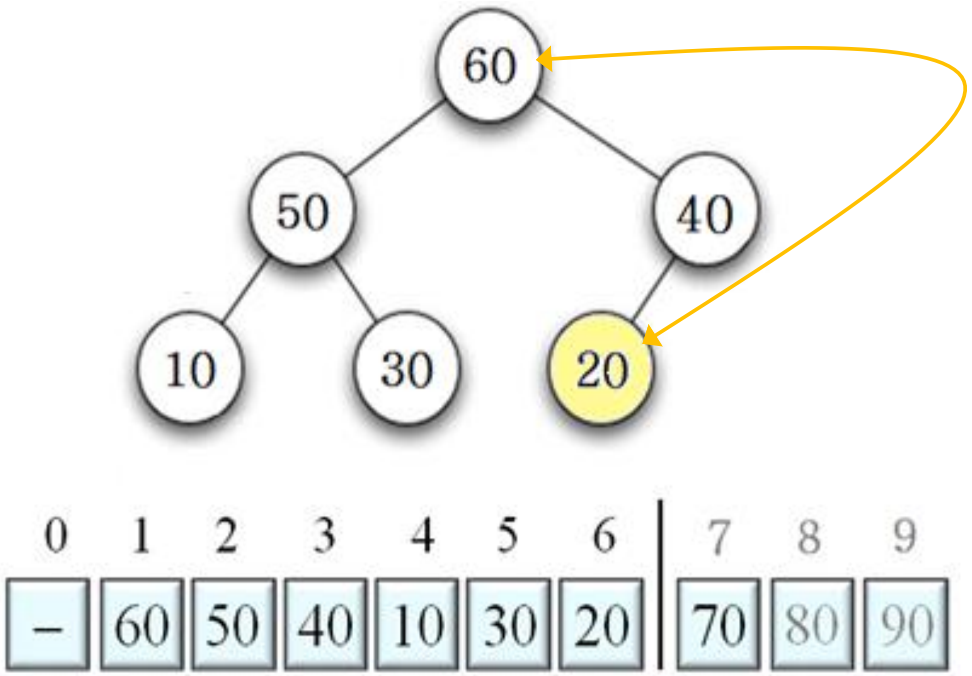
# Heap Sort

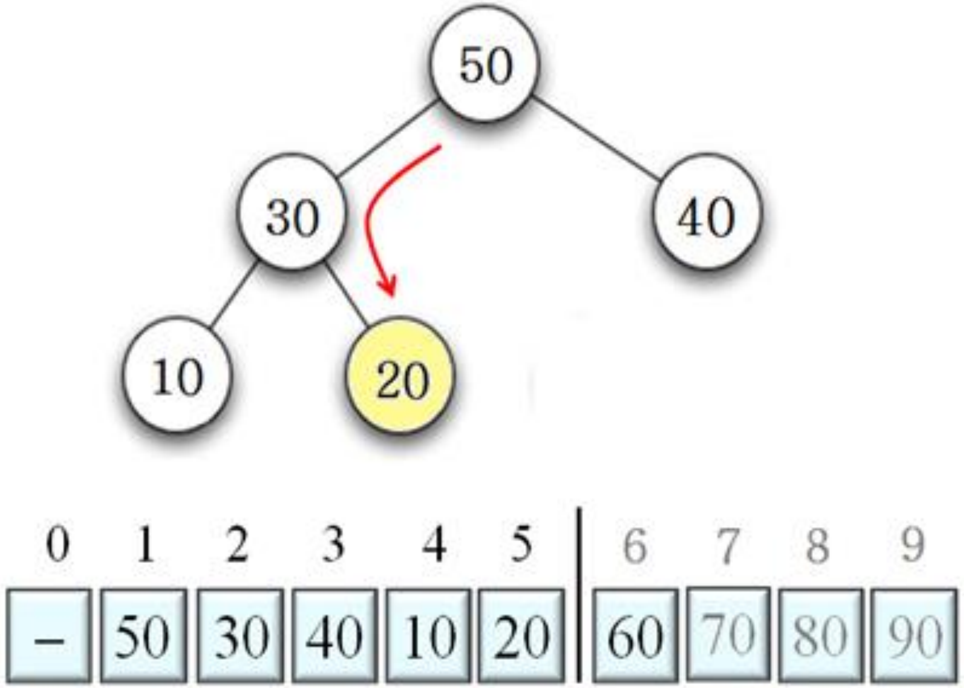
- Heap Sort 알고리즘 수행 과정
  - 70 ↔ 10 교환 후 DownHeap 수행 결과



# Heap Sort

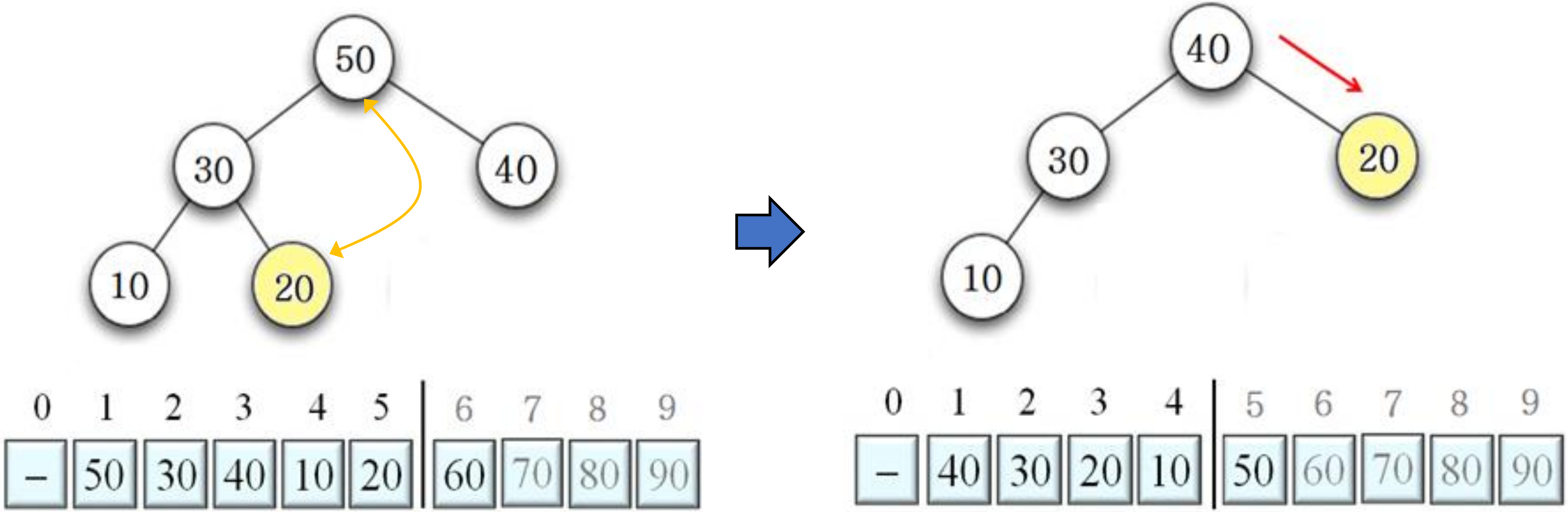
- Heap Sort 알고리즘 수행 과정
  - 60 ↔ 20 교환 후 DownHeap 수행 결과



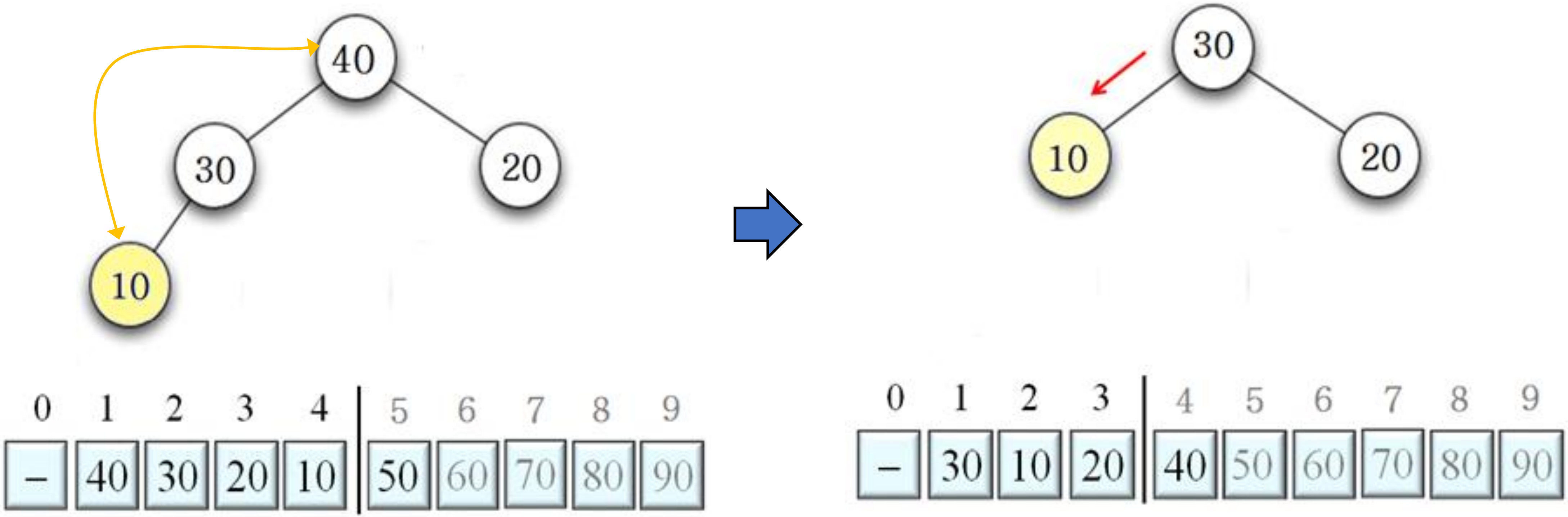
# Heap Sort

- Heap Sort 알고리즘 수행 과정
  - 50 ↔ 20 교환 후 DownHeap 수행 결과



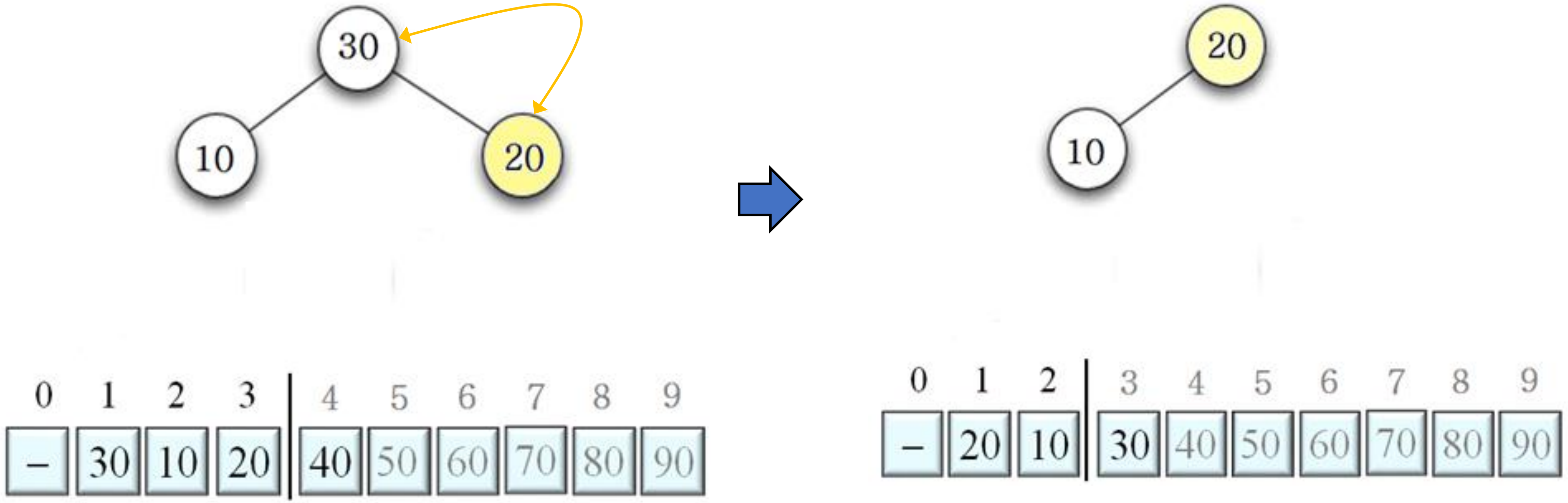
# Heap Sort

- Heap Sort 알고리즘 수행 과정
  - 40 ↔ 10 교환 후 DownHeap 수행 결과



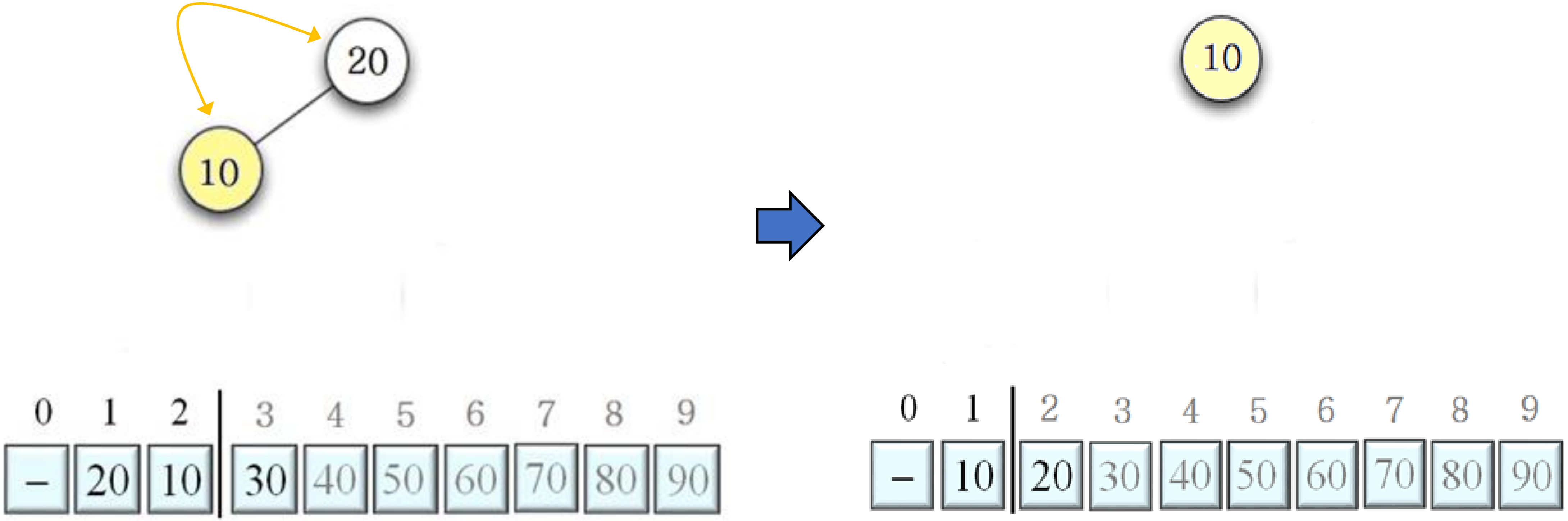
# Heap Sort

- Heap Sort 알고리즘 수행 과정
  - 30 ↔ 20 교환 후 DownHeap 수행 결과



# Heap Sort

- Heap Sort 알고리즘 수행 과정
  - 20 ↔ 10 교환 후 DownHeap 수행 결과



# Heap Sort

- Time Complexity
  - Heap 만드는 데 O(n) 시간
  - for-loop는 (n-1)번 수행
    - 루프 내부는 O(1) 시간
  - DownHeap은 O(logn) 시간
  - O(n) + (n-1) X O(logn) = O(nlogn)

# Heap Sort

- Heap Sort의 특성
  - 큰 입력에 대해 DownHeap()을 수행할 때 자식을 찾아야 하므로 너무 많은 캐시 미스로 인해 페이지 부재 (page fault)를 야기시킴
  - 최선, 최악, 평균 시간 복잡도가 동일 O(nlogn)

# Heap Sort

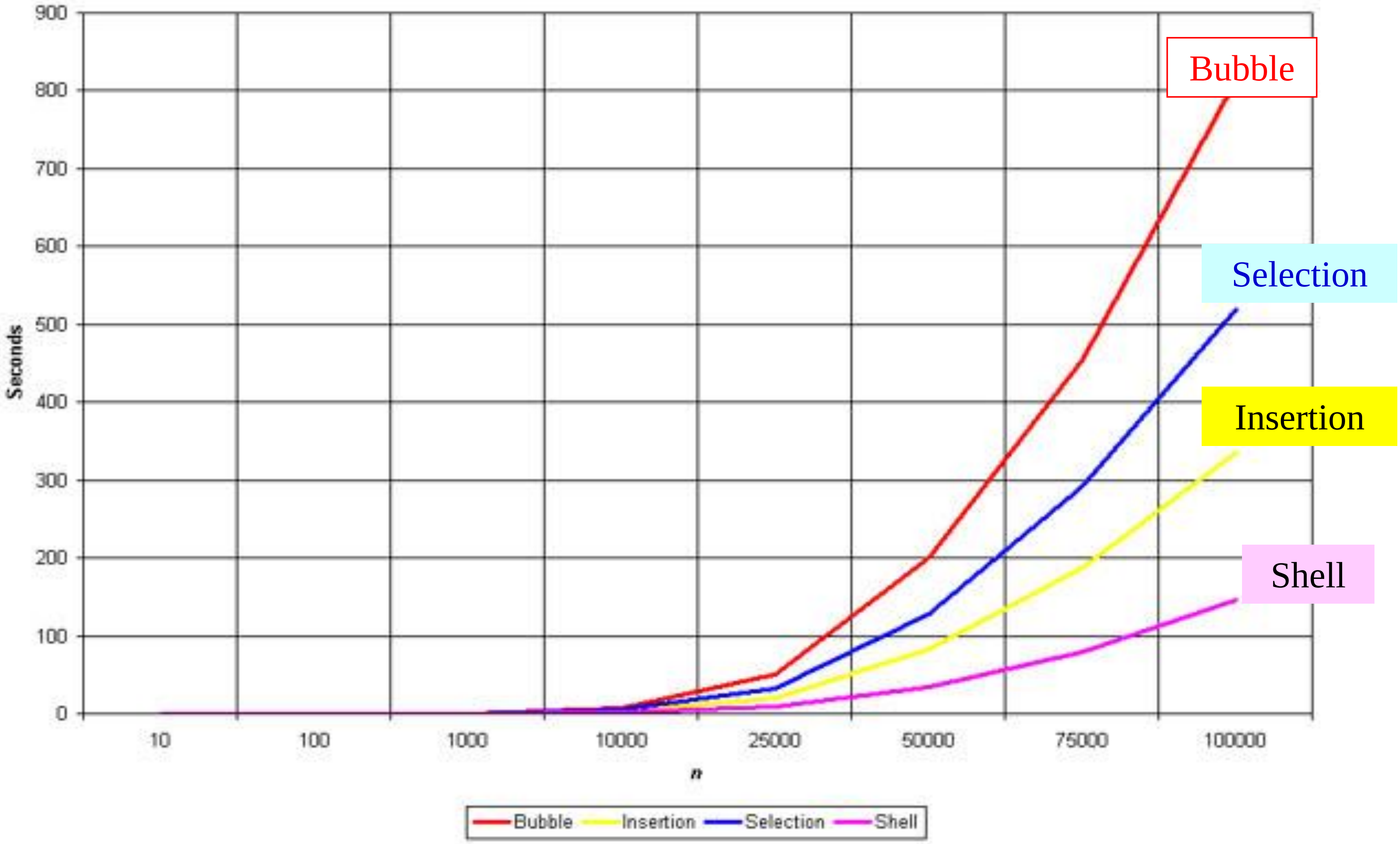
- 내부 정렬 알고리즘 성능 비교

| | 최선 경우 | 평균 경우 | 최악 경우 | 추가 공간 | 안정성 |
|---|---|---|---|---|---|
| 선택 정렬 | $n^2$ | $n^2$ | $n^2$ | $O(1)$ | X |
| 삽입 정렬 | $n$ | $n^2$ | $n^2$ | $O(1)$ | ○ |
| 쉘 정렬 | $nlogn$ | ? | $n^{1.5}$ | $O(1)$ | X |
| 힙 정렬 | $nlogn$ | $nlogn$ | $nlogn$ | $O(1)$ | X |
| 합병 정렬 | $nlogn$ | $nlogn$ | $nlogn$ | $n$ | ○ |
| 퀵 정렬† | $nlogn$ | $nlogn$ | $n^2$ | $O(1)$* | X |
| Tim Sort | $n$ | $nlogn$ | $nlogn$ | $n$ | ○ |

* 퀵 정렬에서 수행되는 순환 호출까지 고려한 추가 공간은 $O(\log n)$
† 이중 피벗 퀵 정렬의 이론적인 성능은 퀵 정렬과 같다.

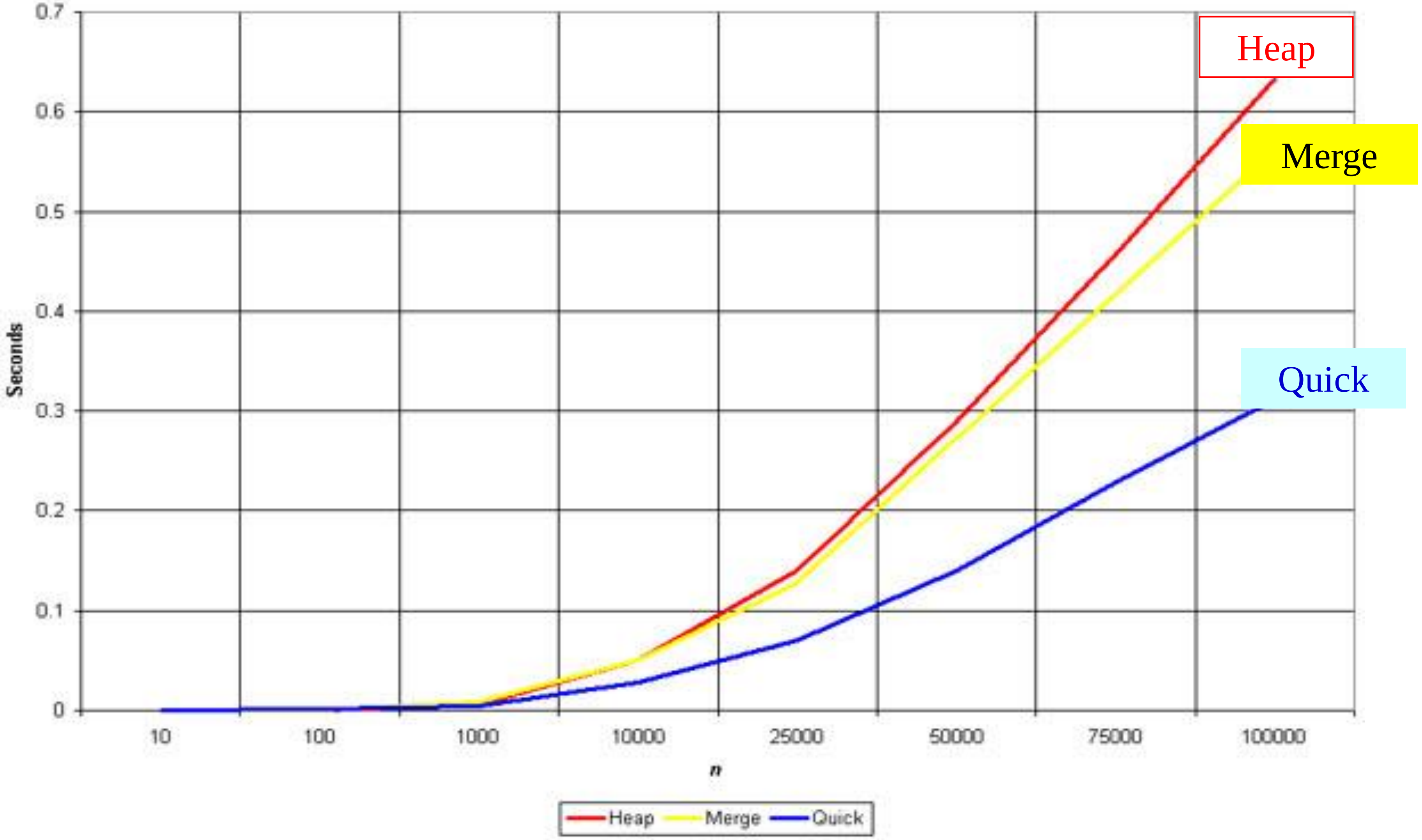
# Heap Sort

- 내부 정렬 알고리즘 성능 비교



# Heap Sort

- 내부 정렬 알고리즘 성능 비교



# 정렬 문제의 하한

- 비교 정렬 (Comparison Sort)
  - Bubble Sort, Selection Sort, Insertion Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort의 공통점은 비교가 부분적이 아닌 숫자 대 숫자로 이루어짐

- 기수 정렬 (Radix sort)은 비교 정렬이 아님
  - 숫자들을 한 자리씩 부분적으로 비교

# 정렬 문제의 하한

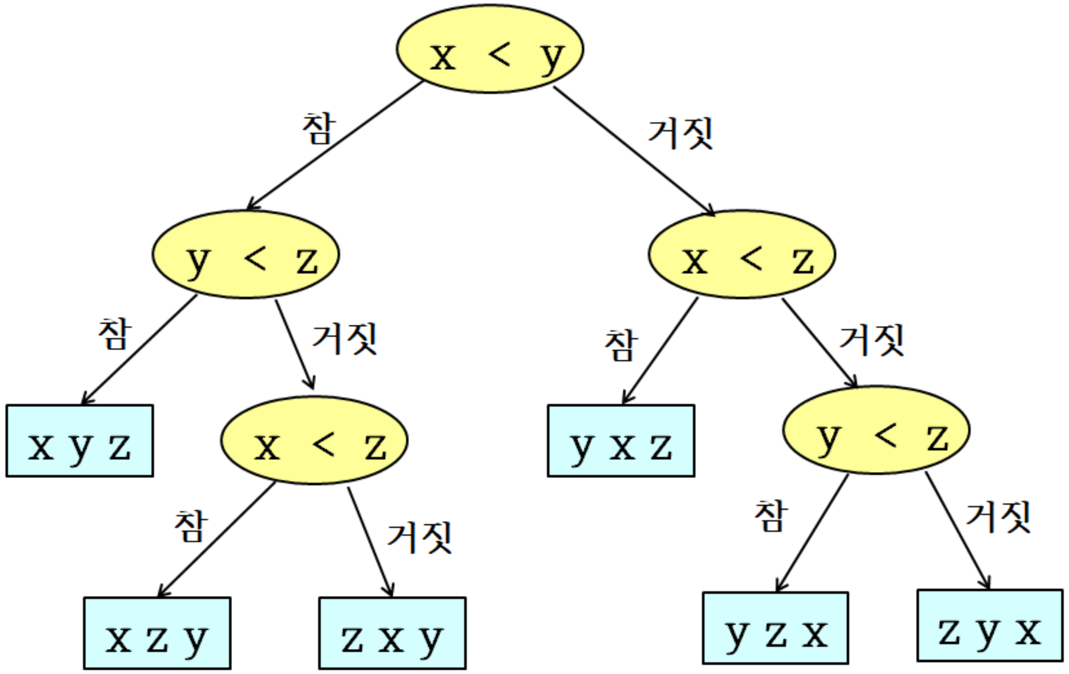
- 어떤 주어진 문제에 대해 Time Complexity의 하한 (lower bound)이라 함은 어떠한 알고리즘도 문제의 하한보다 빠르게 해를 구할 수 없음을 의미
  - 문제의 하한은 어떤 특정 알고리즘에 대한 시간 복잡도의 하한을 뜻하는 것은 아님
  - 문제가 지닌 고유한 특성 때문에 어떠한 알고리즘일지라도 해를 찾으려면 적어도 하한의 time complexity만큼 시간이 걸린다는 뜻

# 정렬 문제의 하한

- 최댓값을 찾는 문제의 하한
  - 최댓값을 찾기 위해 숫자들을 적어도 몇 번 비교해야 하는가?
  - 어떤 방식으로 탐색하든지 적어도 (n-1)번의 비교가 필요
    - 왜냐하면 어떤 방식이라도 각 숫자를 적어도 한 번 비교해야 됨
    - (n-1)보다 작은 비교 횟수가 의미하는 것은 n개의 숫자 중에서 적어도 1개의 숫자는 비교되지 않았다는 것
    - 비교 안 된 숫자가 가장 큰 수일 수도 있기 때문에, (n-1)보다 적은 비교 횟수로는 최댓값을 항상 찾을 수는 없음

# 정렬 문제의 하한

- 정렬 문제의 하한
  - 3개의 서로 다른 숫자 x, y, z에 대해서, 정렬에 필요한 모든 경우의 숫자 대 숫자 비교




결정 트리 (Decision Tree)

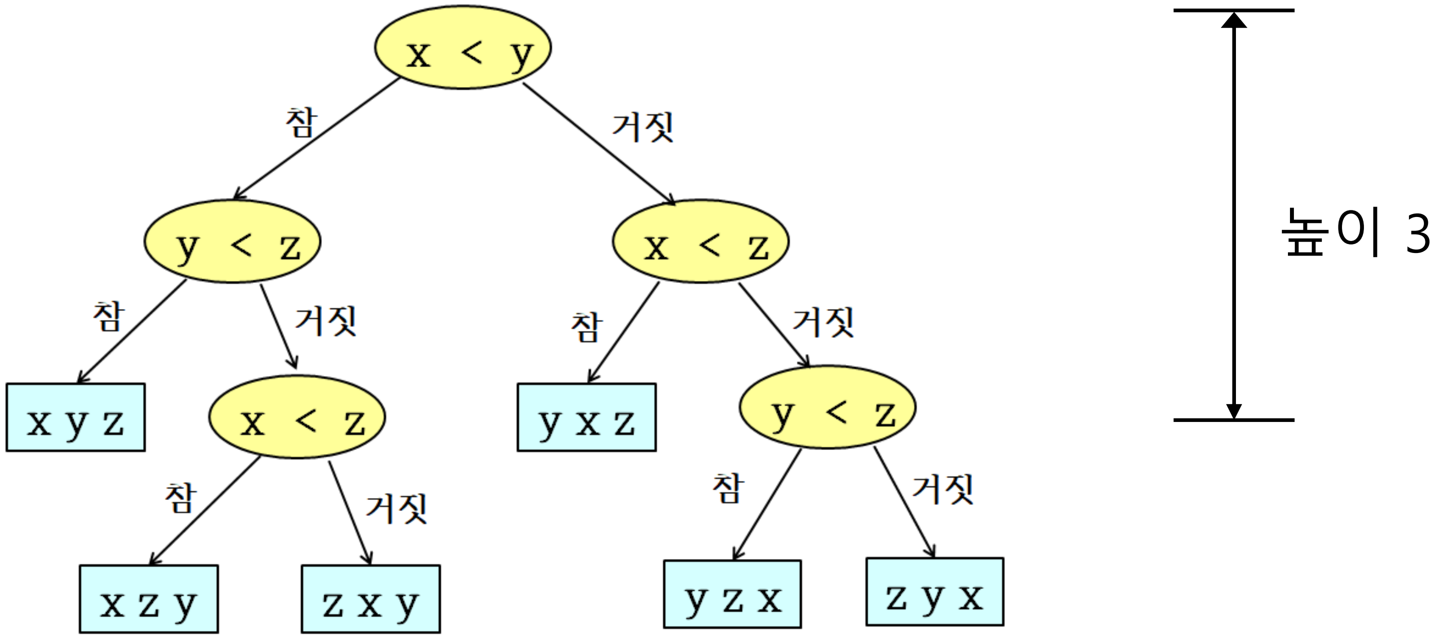
- 각 내부 노드에서는 2개의 숫자가 비교
- 비교 결과가 참이면 왼쪽으로 거짓이면 오른쪽으로 분기
- 각 잎에서는 정렬된 결과 저장

# 정렬 문제의 하한

- 결정 트리의 특징
  - 잎의 수는 3! = 6
  - 결정 트리는 이진 트리 (binary tree)임
  - 결정 트리에는 정렬을 하는데 불필요한 내부 노드가 없음
    - 중복 비교를 하는 노드들이 있으나, 이들은 root로부터 각 잎 노드의 정렬된 결과를 얻기 위해서 반드시 필요한 노드들임

# 정렬 문제의 하한

- 정렬 문제의 하한
  - 어느 경우에도 서로 다른 3개의 숫자가 정렬되기 위해서는 적어도 3번의 비교가 필요함
    - 3번의 횟수는 앞의 결정 트리의 높이



# 정렬 문제의 하한

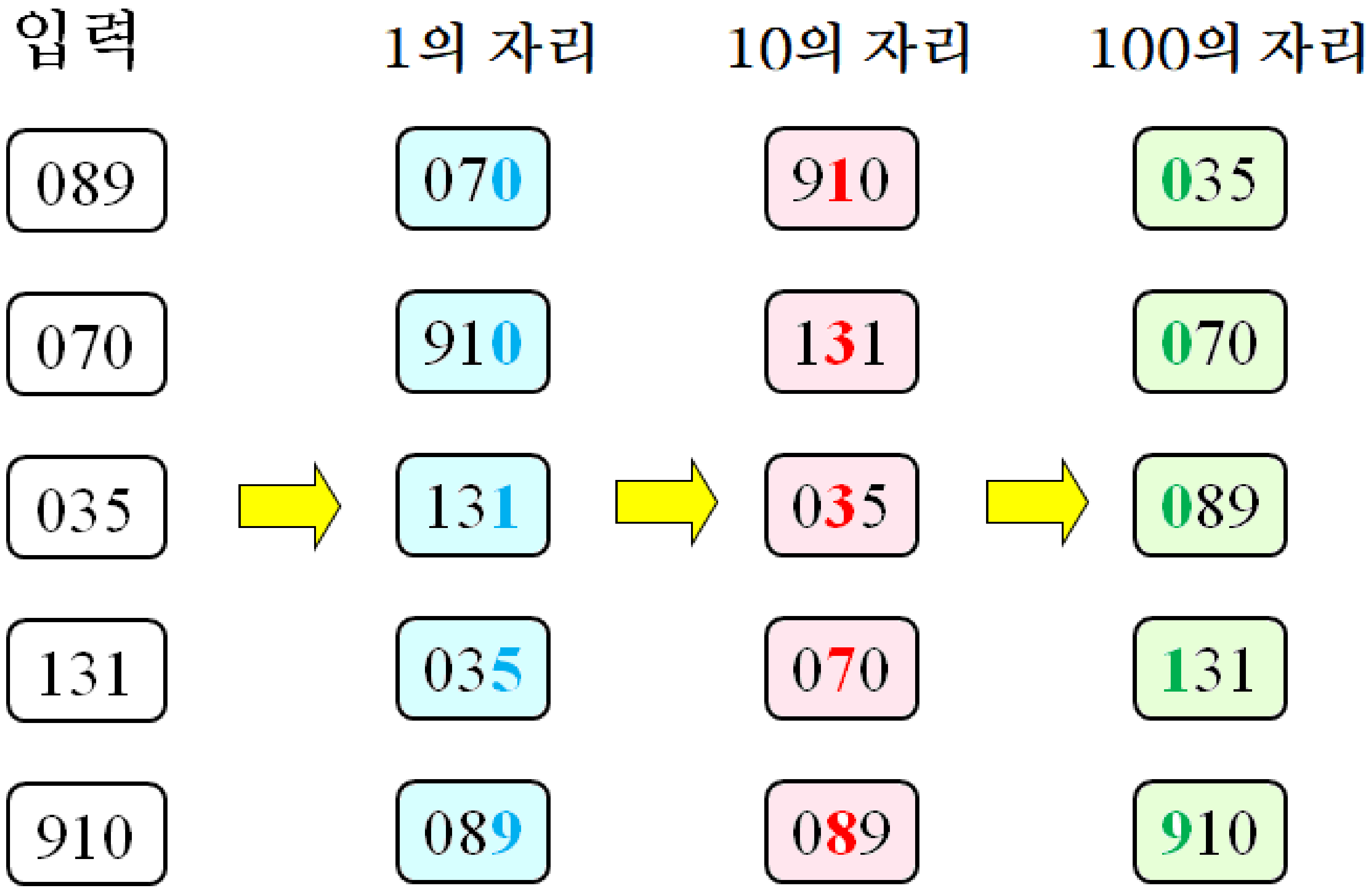
- 정렬 문제의 하한
  - n개의 서로 다른 숫자를 비교 정렬하는 결정 트리의 높이가 비교 정렬의 하한임
  - k개의 잎이 있는 이진 트리의 높이는 logk보다 큼
  - 따라서 n!개의 잎을 가진 결정 트리의 높이는 log(n!)보다 큼
  - log(n!) = O(nlogn)이므로, 비교 정렬의 하한은 O(nlogn)
    - $n! >= (n/2)^{n/2}$ 이므로 $\log(n!) >= \log(n/2)^{n/2} = (n/2)\log(n/2) = O(n\log n)$
  - 즉, O(nlogn)보다 빠른 time complexity를 가진 비교 정렬 알고리즘은 존재하지 않음
  - 점근적 표기 방식으로 하한을 표기하면 Ω(nlogn)

# Radix Sort

- 기수 정렬 (Radix Sort)
  - 숫자를 부분적으로 비교하며 정렬
  - 기(radix)는 특정 진수를 나타내는 숫자들
    - 10진수의 기는 0, 1, 2, ..., 9
    - 2진수의 기는 0, 1
  - Radix sort는 제한적인 범위 내에 있는 숫자에 대해서 각 자릿수 별로 정렬하는 알고리즘임

# Radix Sort

• 기수 정렬 (Radix Sort)



# Radix Sort

- 안정성 (Stability)
  - 입력에 중복된 숫자가 있을 때, 정렬된 후에도 중복된 숫자의 순서가 입력에서의 순서와 동일하면 정렬 알고리즘이 안정성 (stability)을 가진다고 함

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 정렬 전 | 90 A | 10 B | 35 C | 13 D | 10 E | 35 F | 31 G | 08H |
| 안정한 정렬 | 08H | 10 B | 10 E | 13 D | 31 G | 35 C | 35 F | 90 A |
| 불안정한 정렬 | 08H | 10 E | 10 B | 13 D | 31 G | 35 F | 35 C | 90 A |

# Radix Sort

- 알고리즘

  RadixSort

  입력: n개의 r진수의 k자리 숫자

  출력: 정렬된 숫자

1. for i = 1 to k
2. 각 숫자의 i자리 숫자에 대해 안정한 정렬을 수행
3. return A

# Radix Sort

- LSD 기수 정렬
  - RadixSort는 1의 자리부터 k자리로 진행하는 경우, Least Significant Digit(LSD) 기수 정렬 또는 RL (Right-to-Left) 기수 정렬이라고 부름

# Radix Sort

- LSD 기수 정렬

| A | | | temp | | | A | | | temp | | | A | | | temp | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c | f | b | e | d | a | e | d | a | d | a | b | d | a | b | a | a | c |
| e | d | a | f | e | a | f | e | a | e | a | b | e | a | b | a | b | a |
| d | a | b | a | b | a | a | b | a | a | a | c | a | a | c | a | c | c |
| f | e | a | c | f | b | c | f | b | c | a | e | c | a | e | b | c | e |
| f | f | b | d | a | b | d | a | b | a | b | a | a | b | a | b | b | f |
| e | a | b | f | f | b | f | f | b | b | b | f | b | b | f | c | a | e |
| a | a | c | e | a | b | e | a | b | a | c | c | a | c | c | c | f | b |
| a | b | a | a | a | c | a | a | c | b | c | e | b | c | e | d | a | b |
| b | c | e | a | c | c | a | c | c | e | d | a | e | d | a | e | a | b |
| a | c | c | b | c | e | b | c | e | f | e | a | f | e | a | e | d | a |
| c | a | e | c | a | e | c | a | e | c | f | b | c | f | b | f | e | a |
| b | b | f | b | b | f | b | b | f | f | f | b | f | f | b | f | f | b |

# Radix Sort

- MSD 기수 정렬
  - k자리로 부터 1의 자리로 진행하는 방식은 Most Significant Digit (MSD) 기수 정렬 또는 LR (Left-to-right) 기수 정렬이라고 부름

# Radix Sort

- MSD 기수 정렬

| A | | |
|---|---|---|
| c | f | b |
| e | d | a |
| d | a | b |
| f | e | a |
| f | f | b |
| e | a | b |
| a | a | c |
| a | b | a |
| b | c | e |
| a | c | c |
| c | a | e |
| b | b | f |

| temp | | |
|---|---|---|
| a | a | c |
| a | b | a |
| a | c | c |
| b | c | e |
| b | b | f |
| c | f | b |
| c | a | e |
| d | a | b |
| e | d | a |
| e | a | b |
| f | e | a |
| f | f | b |

| A | | |
|---|---|---|
| a | a | c |
| a | b | a |
| a | c | c |
| b | c | e |
| b | b | f |
| c | f | b |
| c | a | e |
| d | a | b |
| e | d | a |
| e | a | b |
| f | e | a |
| f | f | b |

| temp | | |
|---|---|---|
| a | a | c |
| a | b | a |
| a | c | c |
| b | b | f |
| b | c | e |
| c | a | e |
| c | f | b |
| d | a | b |
| e | a | b |
| e | d | a |
| f | e | a |
| f | f | b |

| A | | |
|---|---|---|
| a | a | c |
| a | b | a |
| a | c | c |
| b | b | f |
| b | c | e |
| c | a | e |
| c | f | b |
| d | a | b |
| e | a | b |
| e | d | a |
| f | e | a |
| f | f | b |

| temp | | |
|---|---|---|
| a | a | c |
| a | b | a |
| a | c | c |
| b | b | f |
| b | c | e |
| c | a | e |
| c | f | b |
| d | a | b |
| e | a | b |
| e | d | a |
| f | e | a |
| f | f | b |

# Radix Sort

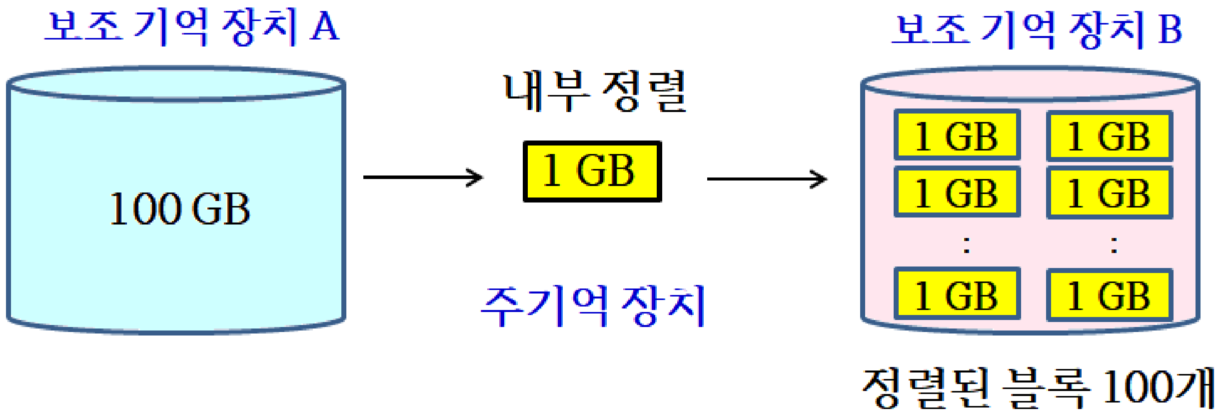
- Time Complexity
  - for-loop가 k번 반복
    - k는 입력의 최대 자릿수
    - Loop가 1회 수행될 때 n개의 숫자의 i자리 수를 읽으며, r개로 분류하여 개수를 세고, 그 결과에 따라 숫자가 이동하므로 O(n+r) 시간 소요
  - Time Complexity는 O(k(n+r))
    - k나 r이 입력 크기인 n보다 크지 않으면, time complexity는 O(n)

# External Sort

- 내부 정렬 (Internal Sort)
  - 입력이 주기억 장치 (내부 메모리)에 있는 상태에서 정렬이 수행되는 정렬
- 외부 정렬 (External Sort)
  - 입력 크기가 매우 커서 읽고 쓰는 시간이 오래 걸리는 보조 기억 장치에 입력을 저장할 수밖에 없는 상태에서 수행되는 정렬
  - 주기억 장치의 용량이 1GB이고, 입력 크기가 100GB라면, 어떤 내부 정렬 알고리즘으로도 직접 정렬할 수 없음

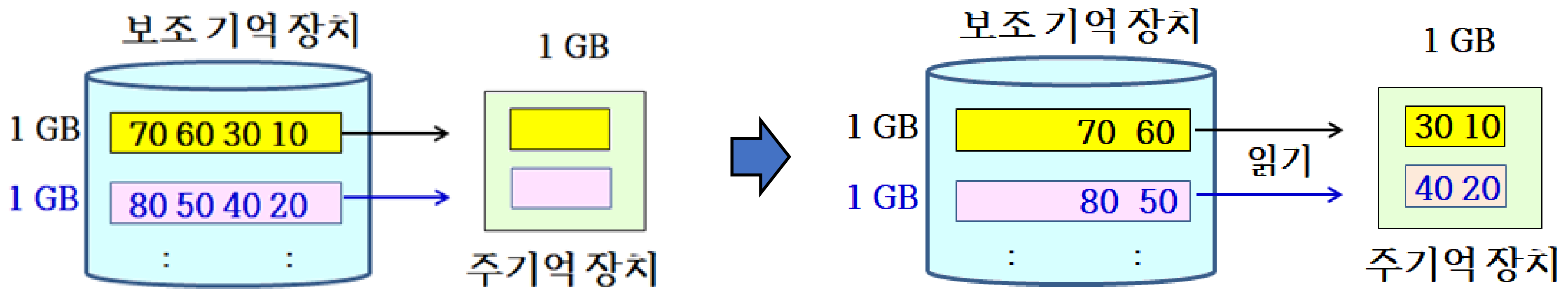
# External Sort

- 주기억 장치에 수용할 만큼 Read/Sort
  - 외부 정렬은 입력은 분할하여 주기억 장치에 수용할 만큼의 데이터에 대해서만 내부 정렬을 수행하고, 그 결과를 보조 기억 장치에 일단 다시 저장
    - 100GB의 데이터를 1GB 만큼씩 주기억 장치로 읽어 들이고, quick sort와 같은 내부 정렬 알고리즘을 통해 정렬한 후, 다른 보조 기억 장치에 저장
  - 이를 반복하면, 원래의 입력이 100개의 정렬된 블록으로 분할되어 보조 기억 장치에 저장됨



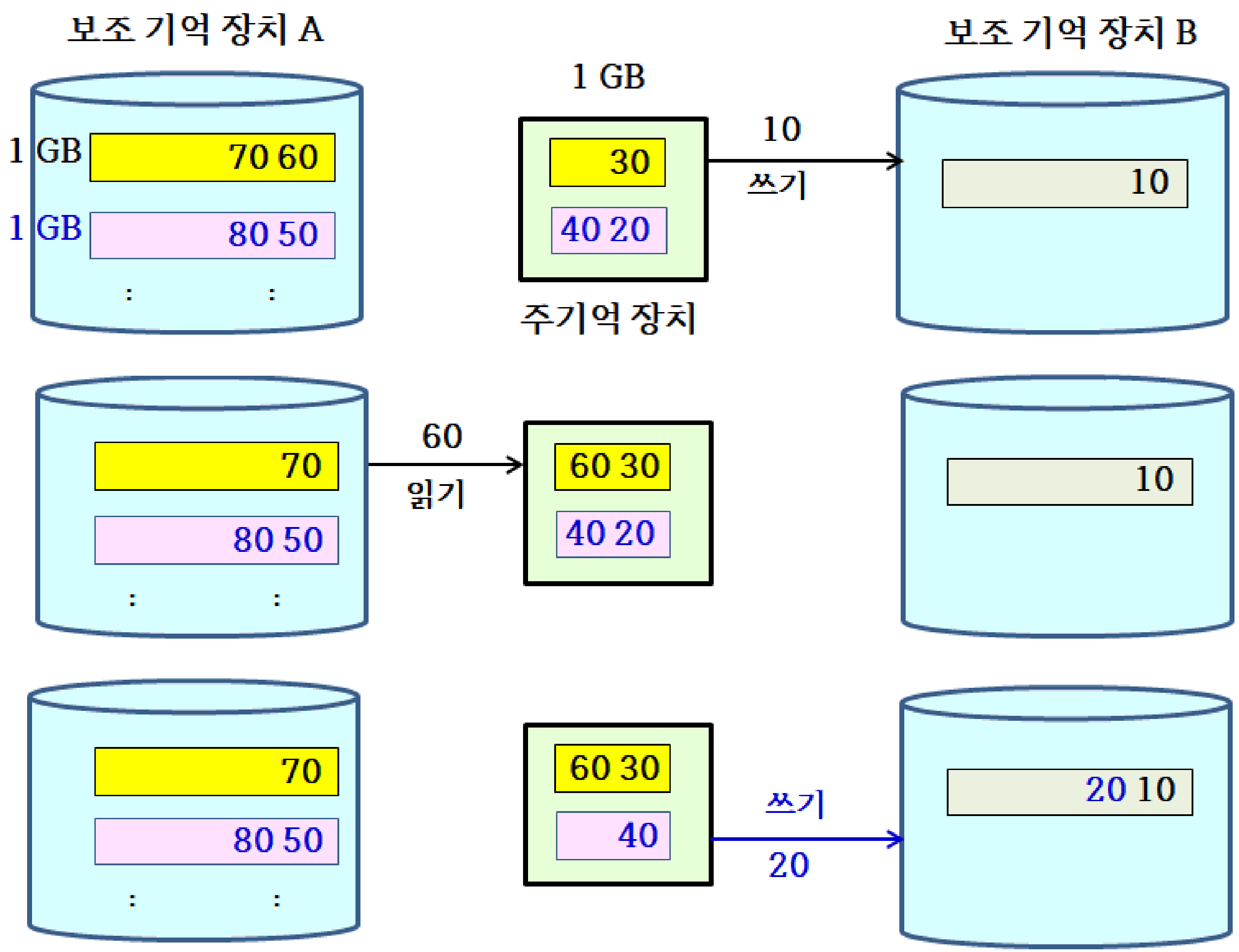
# External Sort

- 정렬된 블록의 합병
  - 정렬된 블록들을 반복적인 합병(merge)을 통해서 하나의 정렬된 거대한 (100GB 크기) 블록으로 만듦
    - 블록들을 부분적으로 주기억 장치에 읽어 들여서, 합병을 수행하여 부분적으로 보조 기억 장치에 쓰는 과정 반복
  - 블록을 부분적으로 읽어 들인 상황



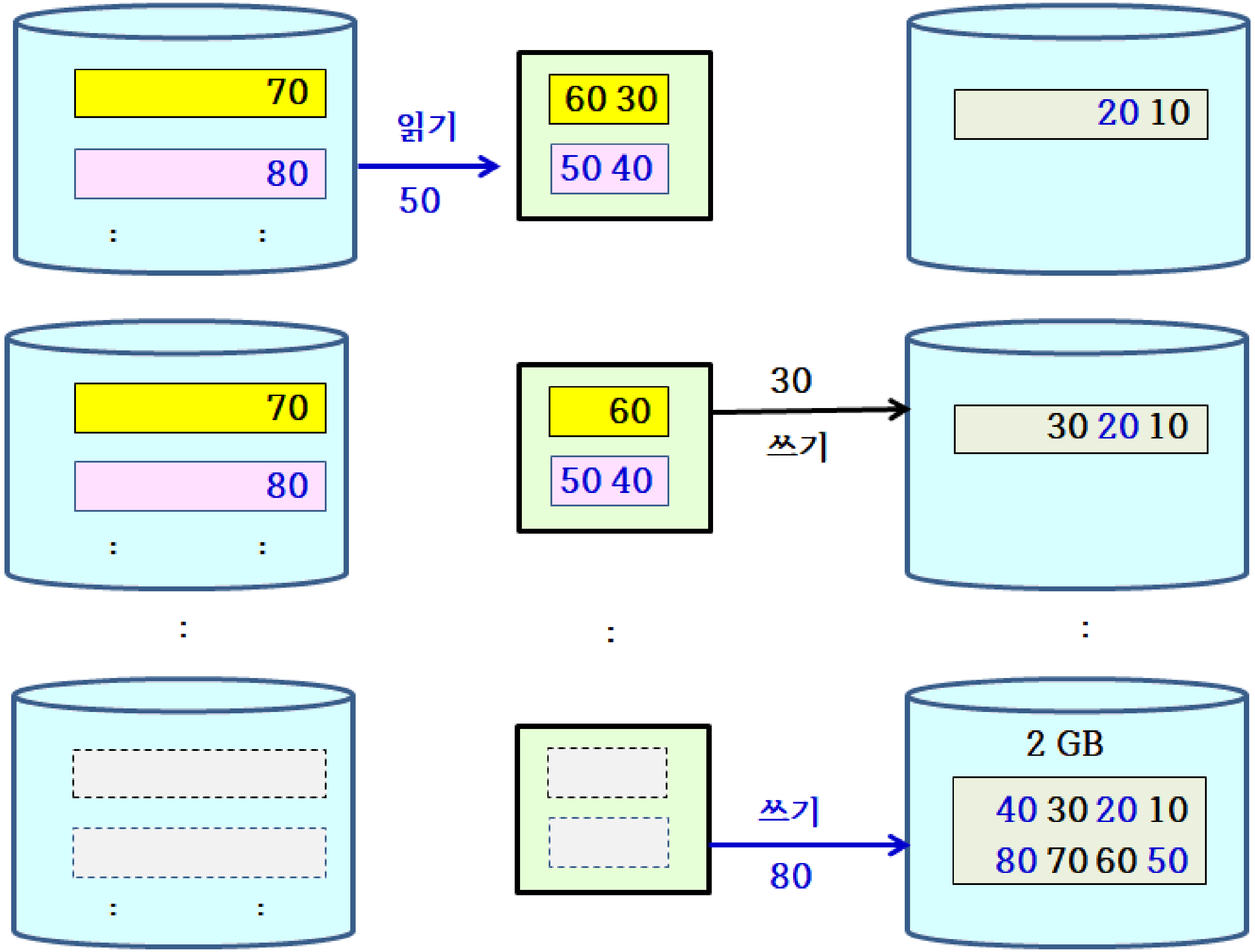
# External Sort

- 1GB 블록 2개가 합병되는 과정



# External Sort

- 1GB 블록 2개가 합병되는 과정



# External Sort

- 1GB 블록 2개가 합병되는 과정
  - 나머지 98개의 블록에 대해서 이와 같이 49회를 반복하면, 2GB 블록이 총 50개 만들어지고
  - 그 다음엔 2GB 블록 2개씩 짝을 지워 합병하는 과정을 총 25회 수행하면, 4GB 블록 25개가 만들어짐
  - 이러한 방식으로 계속 합병을 진행하면, 블록 크기가 2배로 커지고 블록의 수는 1/2로 줄어들게 되어 결국에는 100GB 블록 1개만 남음

# External Sort

- 1GB 블록 2개가 합병되는 과정
  - 외부 정렬 알고리즘은 보조 기억 장치에서의 읽고 쓰기를 최소화하는 것이 매우 중요
    - 왜냐하면 보조 기억 장치의 접근 시간이 주기억 장치의 접근 시간보다 매우 오래 걸리기 때문
  - ExternalSort()
    - M : 주기억 장치의 용량
    - 외부 정렬 알고리즘은 입력이 저장된 보조 기억 장치 외에 별도의 보조 기억 장치 사용
    - 알고리즘에서 보조 기억 장치는 'HDD'임

# External Sort

- 알고리즘

  ExternalSort

  입력: 입력 데이터에 저장된 입력 HDD

  출력: 정렬된 데이터가 저장된 출력 HDD

1. 입력 HDD에 저장된 입력을 M만큼씩 주기억 장치에 읽어 들인 후 내부 정렬 알고리즘으로 정렬하여 별도의 HDD에 저장함. 다음 단계에서 별도의 HDD는 입력 HDD로 사용되고, 입력 HDD는 출력 HDD로 사용
2. while 입력 HDD에 저장된 블록 수 > 1
3. 입력 HDD에 저장된 블록을 2개씩 선택하여, 각각의 블록으로부터 데이터를 부분적으로 주기억 장치에 읽어 들여서 합병을 수행함. 이때 합병된 결과는 출력 HDD에 저장함. 단, 입력 HDD에 저장된 블록 수가 홀수일 때에는 마지막 블록은 그대로 출력 HDD에 저장
4. 입력과 출력 HDD의 역할을 바꾼다
5. return 출력 HDD

# External Sort

- External Sort의 수행 과정
  - 128GB 입력과 1GB의 주기억 장치에 대한 ExternalSort의 수행 과정
    - 1GB의 정렬된 블록 128개를 별도의 HDD에 저장
    - 2GB의 정렬된 블록 64개가 출력 HDD에 만들어짐
    - 4GB의 정렬된 블록 32개가 출력 HDD에 만들어짐
    - 8GB의 정렬된 블록 16개가 출력 HDD에 만들어짐

⋮

    - 64GB의 정렬된 블록 2개가 출력 HDD에 만들어짐
    - 128GB의 정렬된 블록 1개가 출력 HDD에 만들어짐
    - 출력 HDD를 return

# External Sort

- Time Complexity
  - 외부 정렬은 전체 데이터를 몇 번 처리(읽고 쓰기)하는가를 가지고 시간 복잡도를 측정
  - 패스(pass): 전체 데이터를 1회 처리하는 것
  - External Sort 알고리즘에서는 line 3에서 전체 데이터를 입력 HDD에서 읽고 합병하여 출력 HDD에 저장함
    - 즉, 1 패스가 수행됨
  - while-loop가 수행된 횟수가 알고리즘의 time complexity

# External Sort

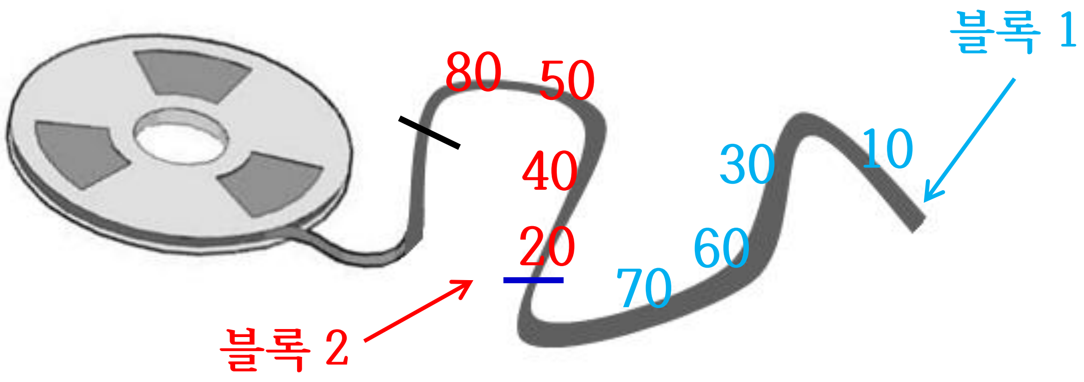
- Time Complexity
  - 입력 크기가 N이고, 메모리 크기가 M이라고 하면, line 3이 수행될 때마다 블록 크기가 2M, 4M, ..., $2^k$M으로 (2배) 증가
  - 마지막에 만들어진 1개의 블록 크기가 $2^k$M이면 이 블록은 입력 전체가 합병된 결과를 가지므로, $2^k M = N$
    - k는 while-loop가 수행된 횟수
    - $2^k = N/M$
    - $k = \log_2(N/M)$
  - Time complexity: $O(\log(N/M))$

# External Sort

- 2-way 합병
  - ExternalSort 알고리즘에서는 하나의 보조 기억 장치에서 2개의 블록을 동시에 주기억 장치로 읽어 들일 수 있다는 가정
  - 2개의 블록이 각각 다른 보조 기억 장치에서 읽어 들여야 하는 경우도 있음
  - 테이프 드라이브 저장 장치는 블록을 순차적으로만 읽고 쓰는 장치이므로, 2개의 블록을 동시에 주기억 장치로 읽어 들일 수 없음

# External Sort

- 2-way 합병
  - 블록 1이 [10 30 60 70], 블록 2가 [20 40 50 80], 이 두 블록을 합병하려면 블록 1의 10을 읽고, 블록 2의 20을 읽어서 비교해야 함




  - 테이프를 한쪽 방향으로만 테이프가 감기므로, 합병하려면 블록 2의 20을 읽은 후 다시 되감아 블록 1의 두 번째 숫자인 30을 읽을 수 없음

# External Sort

- 테이프 드라이브에서 ExternalSort 알고리즘
  - ExternalSort 알고리즘의 line 3에서 2개의 블록을 읽어 들여 합병하면서 만들어지는 블록들을 2개의 저장 장치에 번갈아 저장

# External Sort

- 2-way 합병 수행 과정

| | | | | |
|---|---|---|---|---|
| TD 0 | 블록 1 | 블록 3 | 블록 5 | 블록 7 |
| TD 1 | 블록 2 | 블록 4 | 블록 6 | 블록 8 |
| TD 2 | | | | |
| TD 3 | | | | |

| | | |
|---|---|---|
| TD 0 | | |
| TD 1 | | |
| TD 2 | 블록 1 + 블록 2 | 블록 5 + 블록 6 |
| TD 3 | 블록 3 + 블록 4 | 블록 7 + 블록 8 |

pass 1

# External Sort

- 2-way 합병 수행 과정