Sorting Algorithms

- Running times of sorting algorithms

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

- Comparison Sorts
  - Insertion Sort
    - In-place sorting
    - worst-case running time : $\Theta(n^2)$
  - Merge Sort
    - Out-of-place sorting
    - running time : $\Theta(n \lg n)$
  - Heap Sort
    - In-place sorting
    - running time : $\Theta(n \lg n)$
  - Quick Sort
    - In-place sorting
    - worst-case running time : $\Theta(n^2)$
    - expected running time : $\Theta(n \lg n)$

Heap Sort

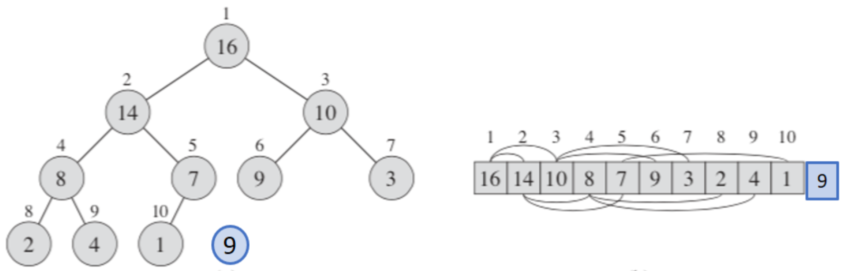
Heap

- (Binary) Heap : heap property, such as A[Parent(i)] >= A[i],
  를 만족하는 완전 이진 트리를 배열에 순서대로 저장한 것
- Length of a heap : 배열에 저장된 **모든 원소의 개수**
- Size of a heap : 배열에 저장된 원소 중 **Heap에 속하는 원소의 갯수**

PARENT(i)
return $\lfloor i/2 \rfloor$

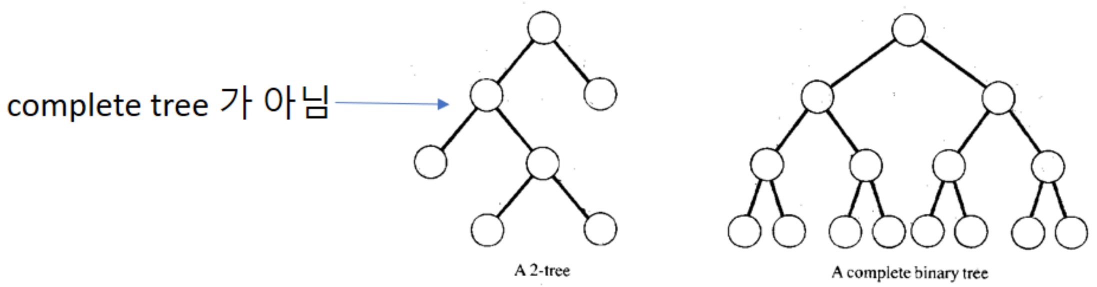
LEFT(i)
return $2i$

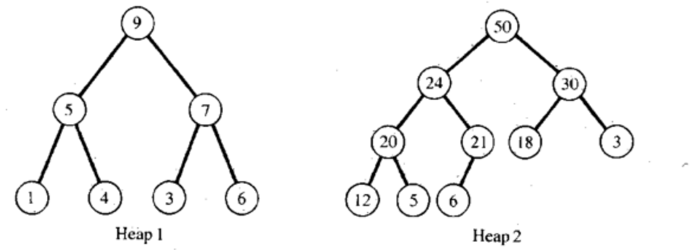
RIGHT(i)
return $2i + 1$



(a) (b)

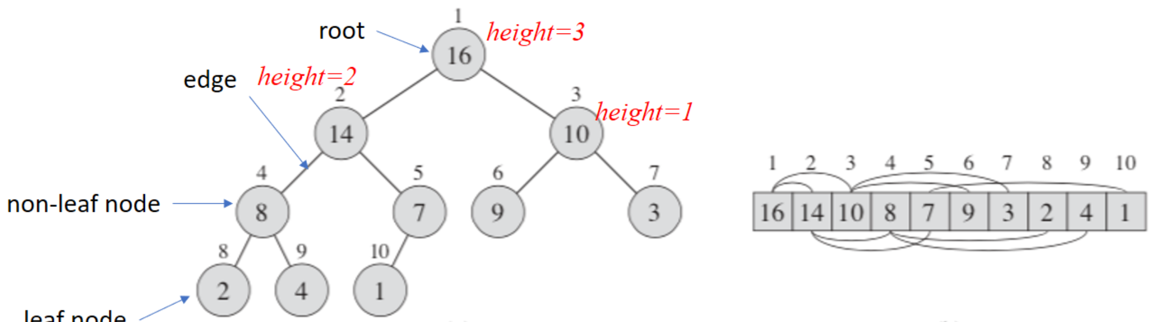
- Examples





Max Heap (최대 힙)

- Heap Property : A[Parent(i)] >= A[i]
- Height of a node : 노드에서 리프에 이르는 **하향 경로 중 가장 긴 것의 edge 개수**
- Height of a heap : height of a root node = $\Theta(\lg n)$, where n = size of heap

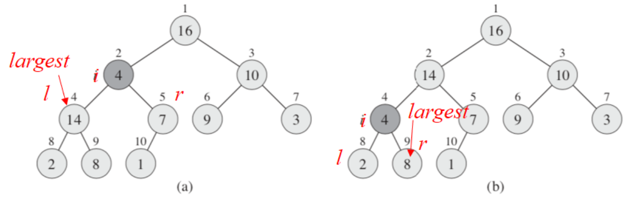


(a) (b)

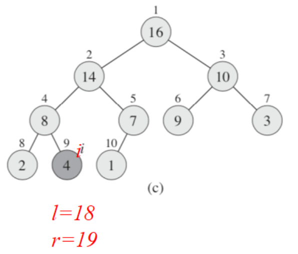
Min Heap (최소 힙)

- Heap Property : A[Parent(i)] <= A[i]

Max Heapify



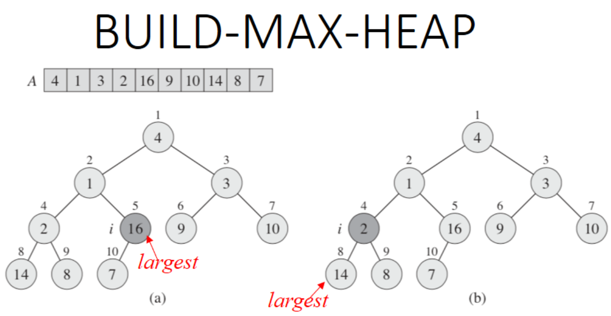

```
MAX-HEAPIFY(A, i)
 l = LEFT(i)
 r = RIGHT(i)
 if l ≤ A.heap-size and A[l] > A[i]
     largest = l
 else largest = i
 if r ≤ A.heap-size and A[r] > A[largest]
     largest = r
 if largest ≠ i
     exchange A[i] with A[largest]
     MAX-HEAPIFY(A, largest)
```



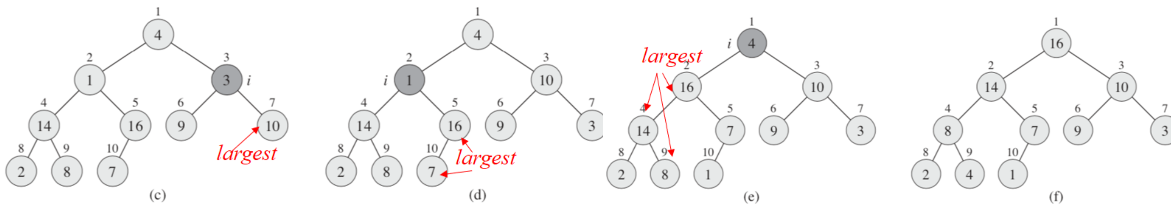

```
PARENT(i)
 return ⌊i/2⌋

LEFT(i)
 return 2i

RIGHT(i)
 return 2i + 1
```

- Running Time : $T(n) = O(h) = O(\lg n)$

Build-Max-Heap




```
BUILD-MAX-HEAP(A)
 A.heap-size = A.length
 for i = ⌊A.length/2⌋ downto 1
     MAX-HEAPIFY(A, i)
```

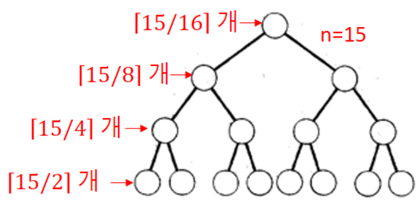


- Running Time
  - Heap Size가 n인 heap에서 height가 h인 노드들의 갯수 ≤ $\lceil n/2^{h+1} \rceil$
  - Height가 h인 실행시간 : O(h)

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

by formula A.8 →

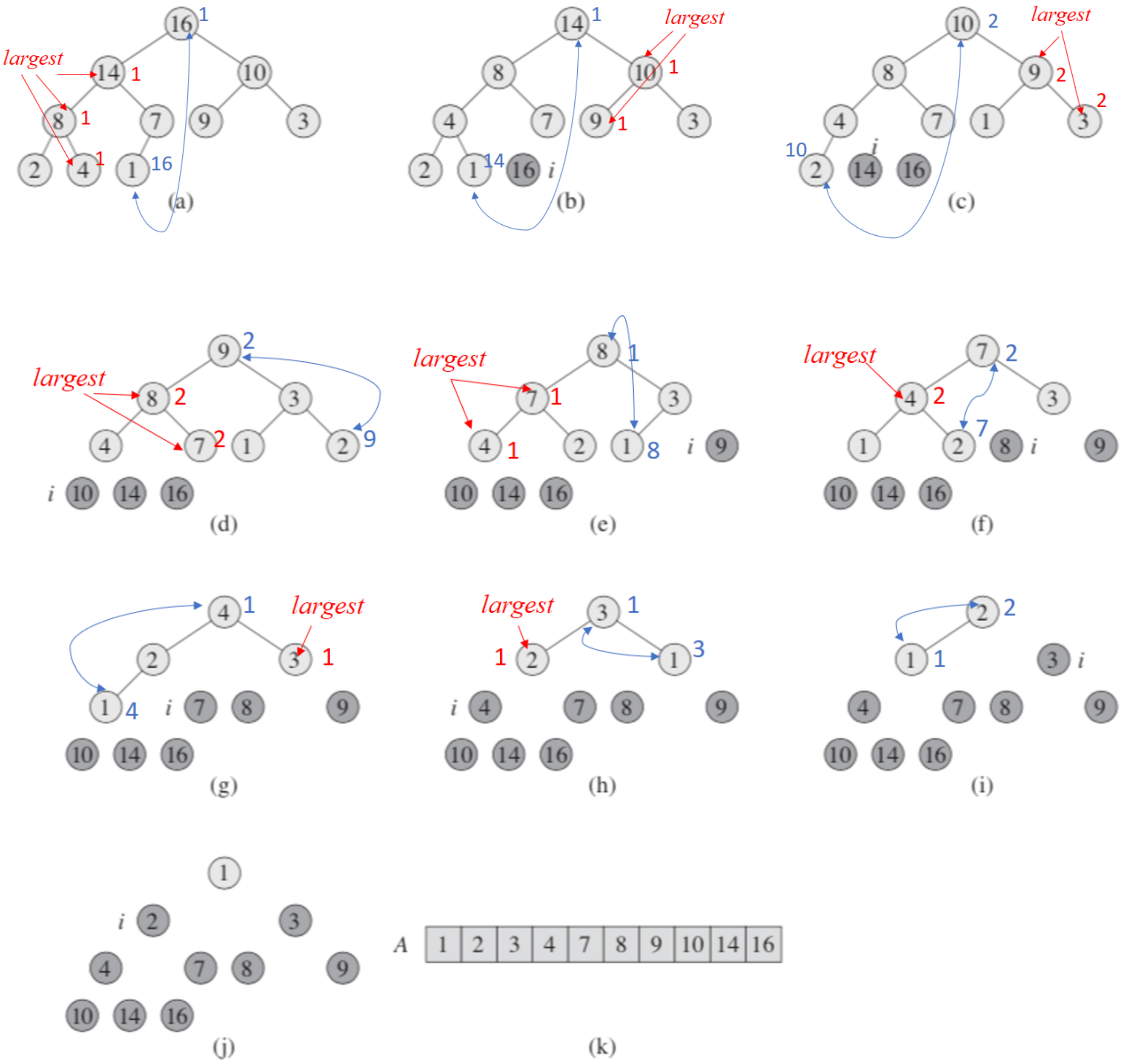


$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n) .$$

HEAPSORT

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

MAX-HEAP



- Running Time

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)                          : O(n)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size - 1
5      MAX-HEAPIFY(A, 1)                      : O(lg i)
```

running time of for loop (line 2-5) : $O(\sum_{i=2}^{n} \lg i) = O(\lg 2 + \lg 3 + \ldots + \lg n)$

$= O(\lg (n!)) = O(\Theta(n \lg n))$ by Stirling's approximation

$T(n) = O(n) + O(n \lg n) = O(n \lg n)$

Factorials

Stirling's approximation : $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$

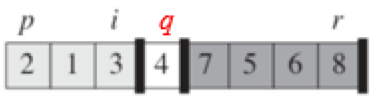
$$\rightarrow \quad \begin{aligned} n! &= o(n^n)\ , \\ n! &= \omega(2^n)\ , \\ \lg(n!) &= \Theta(n \lg n) \end{aligned}$$

Quick Sort

Divide and Conquer

- Divide : 배열 A[p..r]을 두 개의 부분 배열 A[p..q-1]과 A[q+1..r]로 분할
- Conquer : 두 개의 부분 배열을 Quicksort
- Combine : 없음

partition



Recursive Implementation

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Running Time
    - Partition이 Balance 되거나 Unbalance 되거나에 따라서 달라짐
        - Balanced : merge sort와 비슷한 속도가 나옴
        - Unbalanced : insertion sort와 비슷한 속도가 나옴
    - Worst Case Partitioning
        - Partition이 항상 0개 원소의 subarray와 n-1개 원소의 subarray로 나눠질 때
        - input이 sort되어 있을 때

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) . \end{aligned}$$

    - Best Case Partitioning
        - Partition이 항상 n/2개의 원소를 가진 2 개의 subarray로 나눠질 때

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) . \end{aligned}$$

    - Average Case
        - best case 수행시간에 가까움
        - partition이 항상 9:1로 나눠진다면

$$\begin{aligned} T(n) &\leq T(9n/10) + T(n/10) + \Theta(n) \\ &= O(n \lg n) . \end{aligned}$$

랜덤버전 퀵소트

RANDOMIZED-QUICKSORT(A, p, r)

```
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

RANDOMIZED-PARTITION(A, p, r)

```
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

PARTITION(A, p, r)

```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```