3주차

알 고 리 즘

2017. 9. 21.

충남대학교 컴퓨터공학과 분산이동컴퓨팅 연구실
TA 이정진

Contents

Quick Sort

▶ Quick Sort란?

Divide-and-conquer를 기반으로 하는 **내부 정렬** 알고리즘.
분할, 정복, 결합을 수행하는 2개의 프로시저가 필요하다.

▷ 퀵 정렬이 구현된 재귀 프로시저

```
QUICKSORT(A, p, r)
  if p < r
    then q ← PARTITION(A, p, r)
      QUICKSORT(A, p, q–1)
      QUICKSORT(A, q+1, r)
```

Quick Sort

▶ Quick Sort의 내부 정렬 과정

내부 정렬을 담당하는 **PARTITION 프로시저**는
다음과 같은 순서로 동작한다. (교재에 기술된 알고리즘)

1) input parameter = A[p .. r], p, r

2) X = A[r]

3) 정렬 : **{Value ≤ X} {X} {Value > X}**

4) 반환 : **{X}의 인덱스**

Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

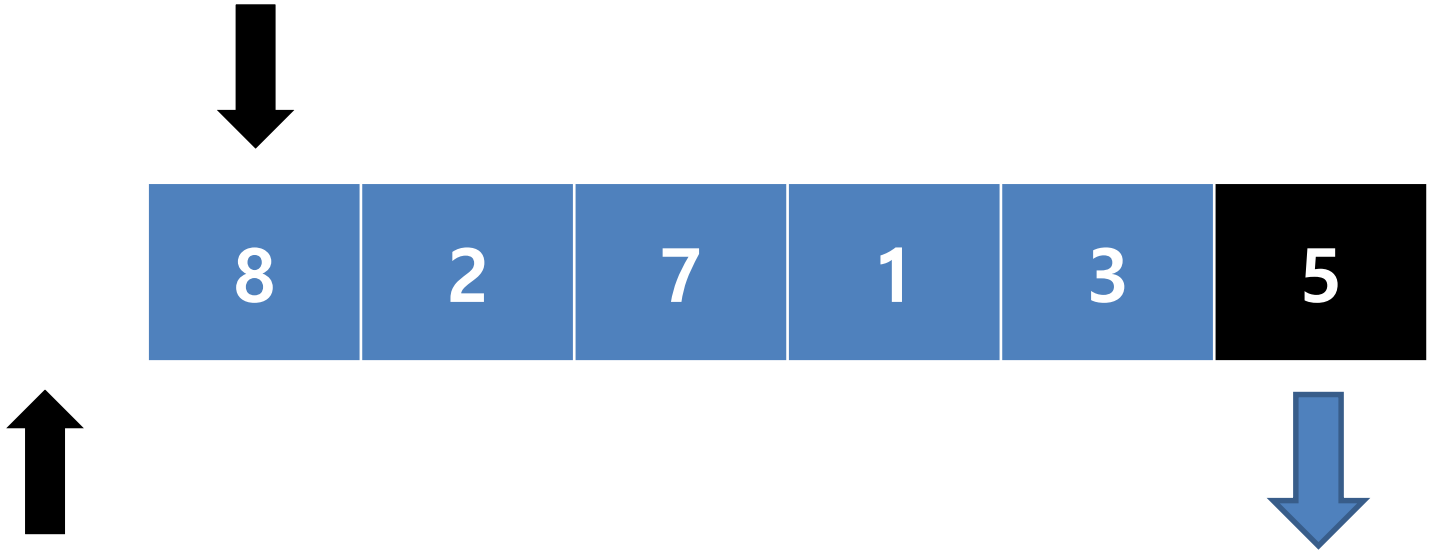
  PARTITION 프로시저의 초기 설정은 다음과 같다.

Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

  반복문 안에는 하나의 조건문 **if A[j] ≤ X**만 존재한다.

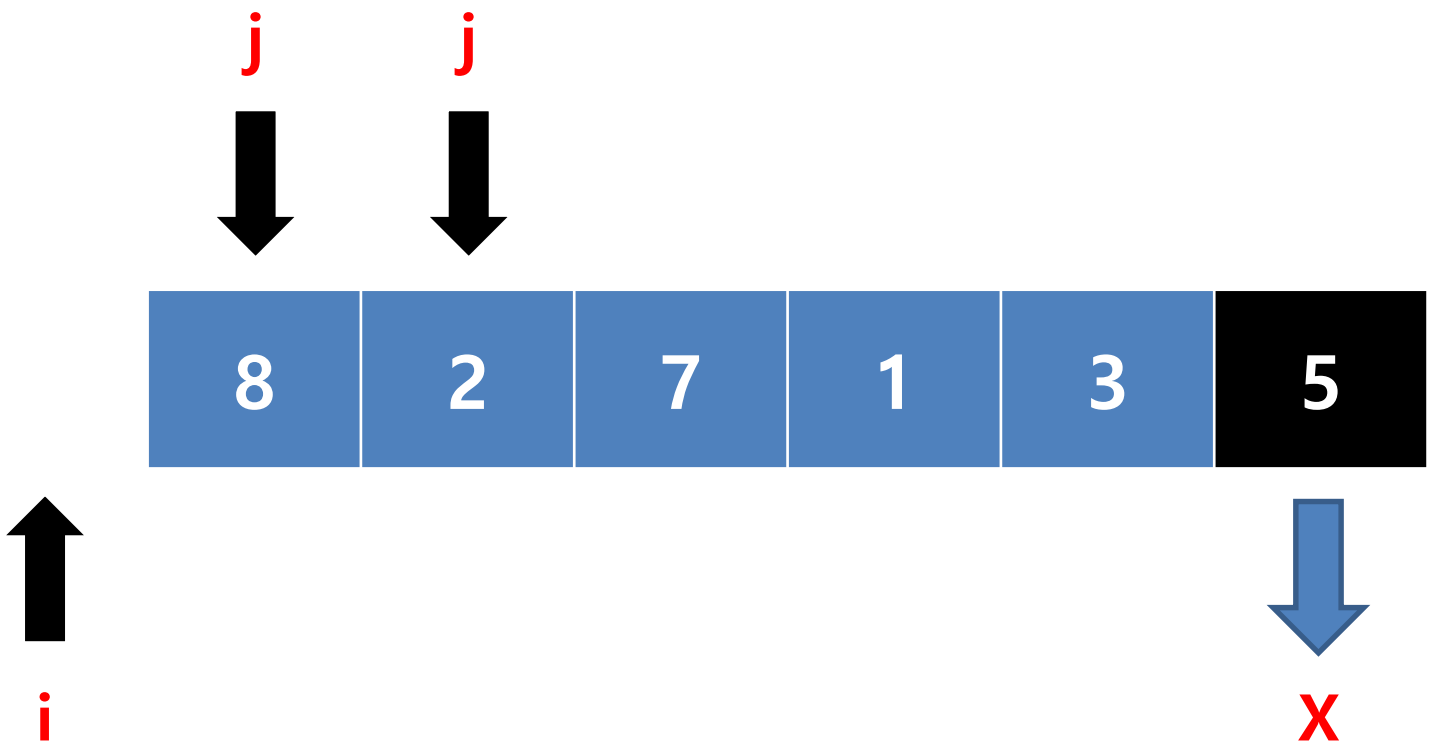
for j = p to r-1



Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

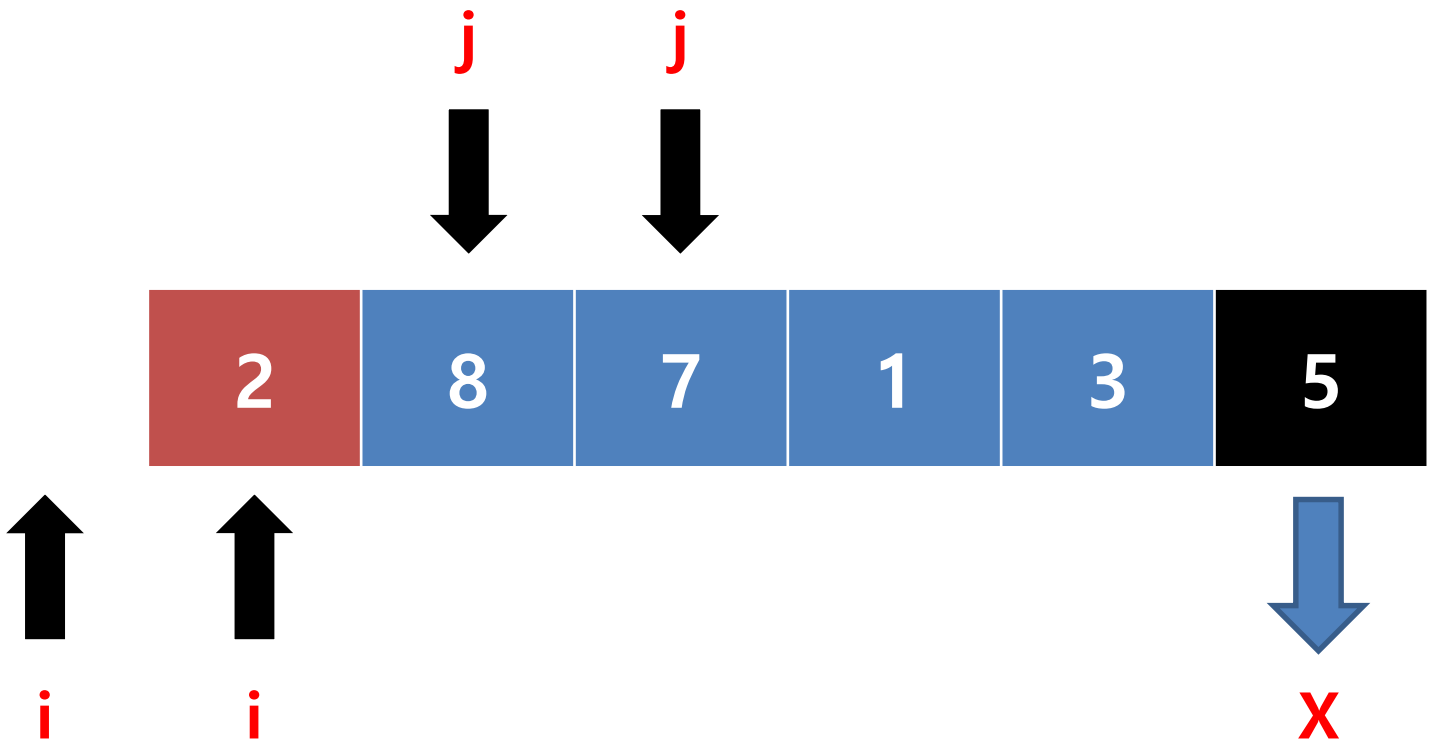
  조건문이 참일 때만 **i를 +1** 시키고, **A[i] ↔ A[j]**를 수행한다.



Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

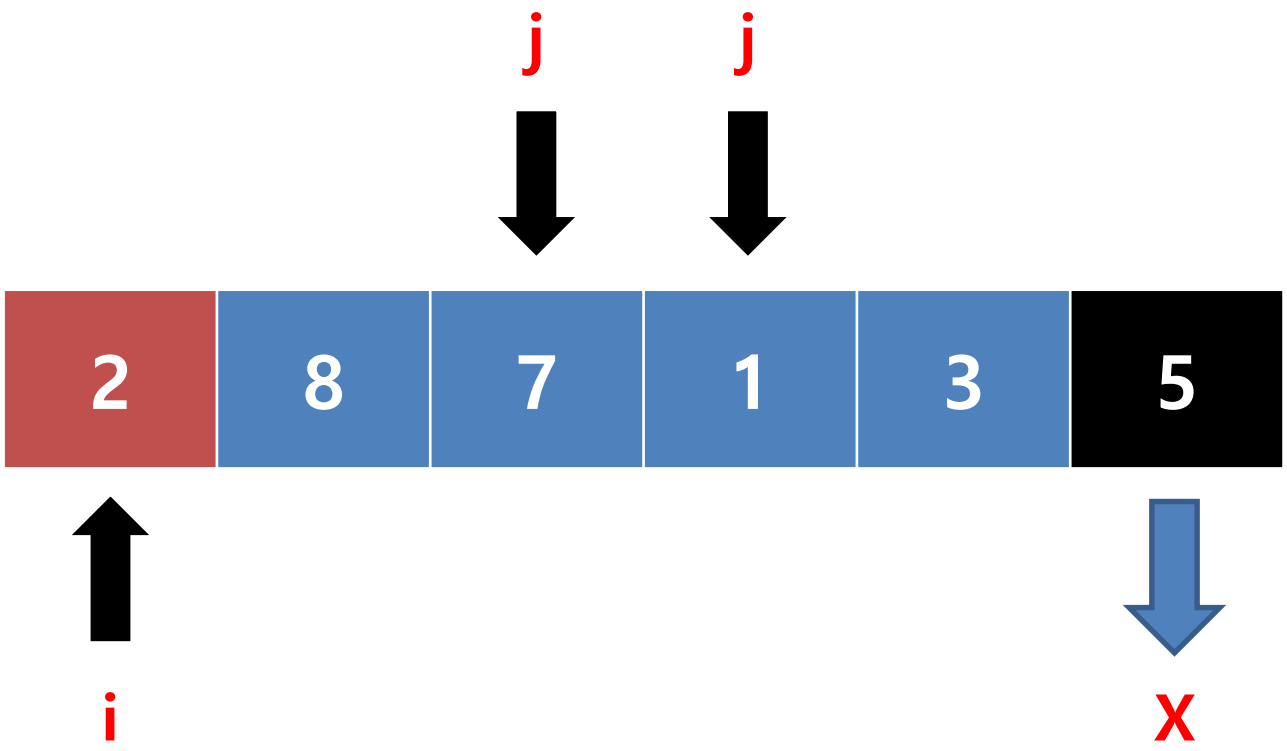
  조건문이 참일 때만 **i를 +1** 시키고, **A[i] ↔ A[j]**를 수행한다.



Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

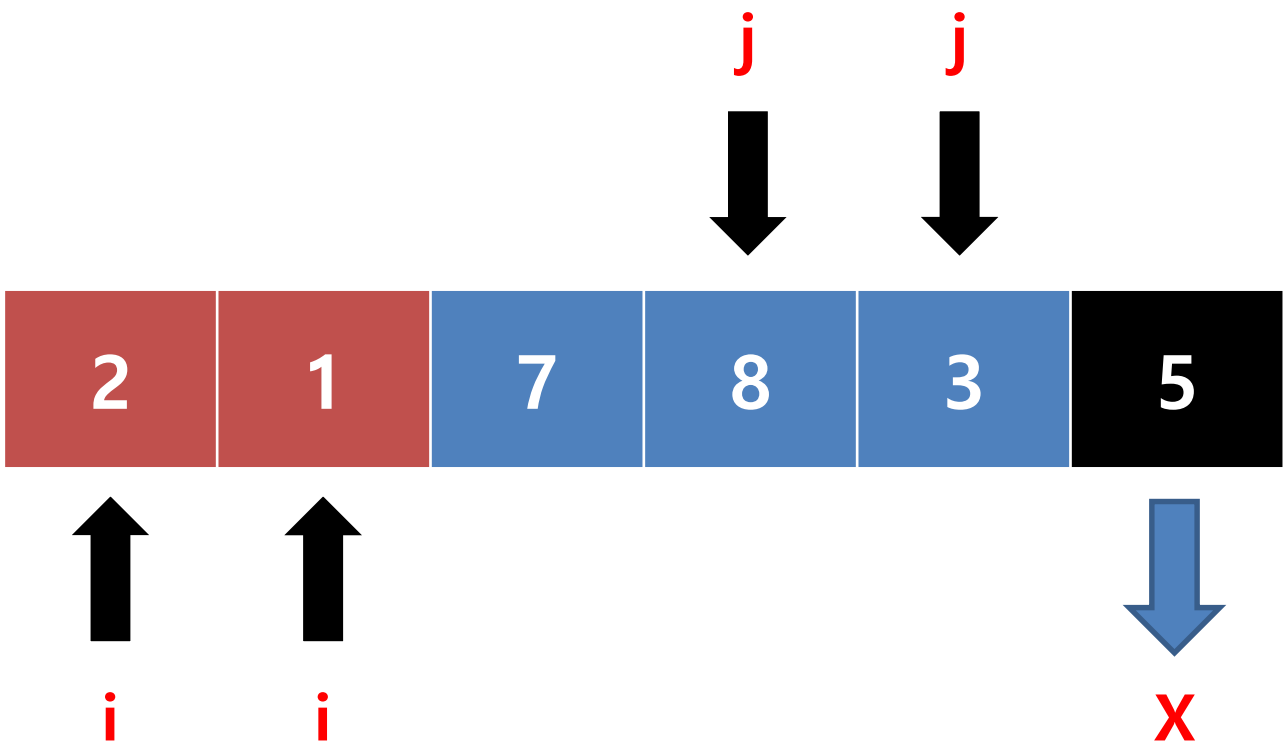
  조건문이 참일 때만 **i를 +1** 시키고, **A[i] ↔ A[j]**를 수행한다.



Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

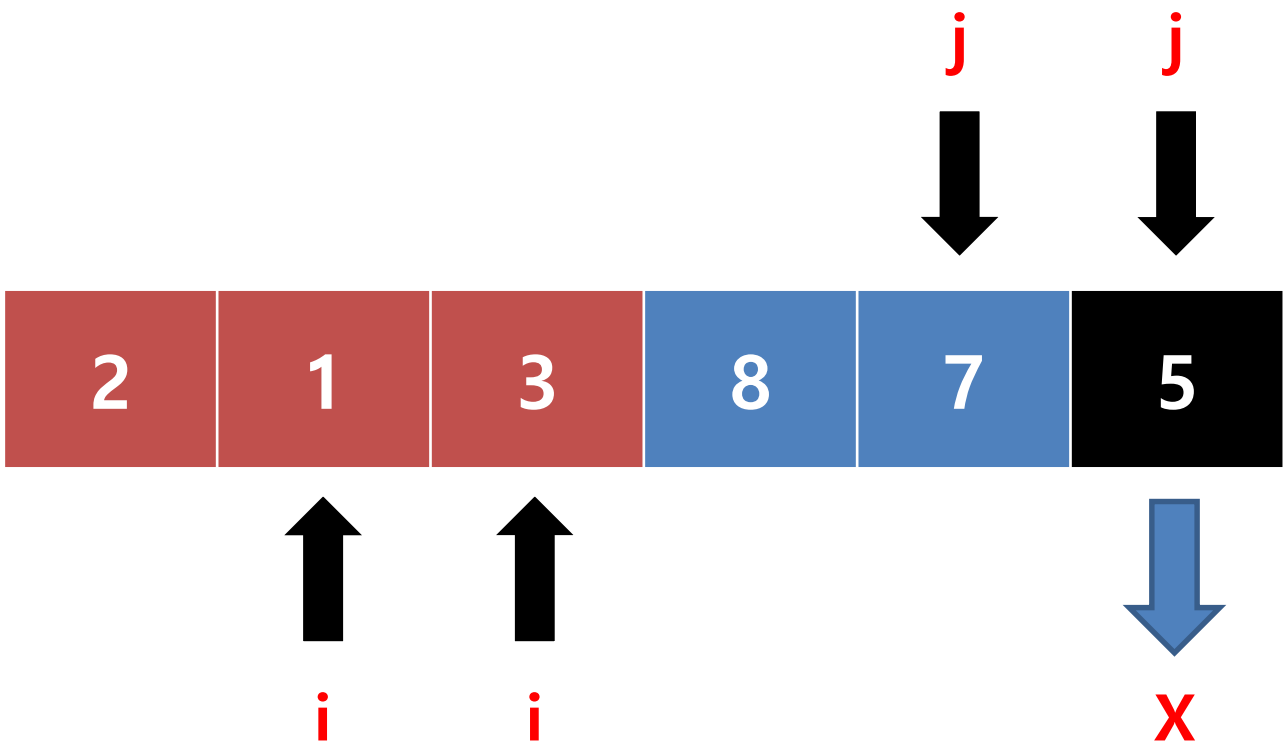
  조건문이 참일 때만 **i를 +1** 시키고, **A[i] ↔ A[j]**를 수행한다.



Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

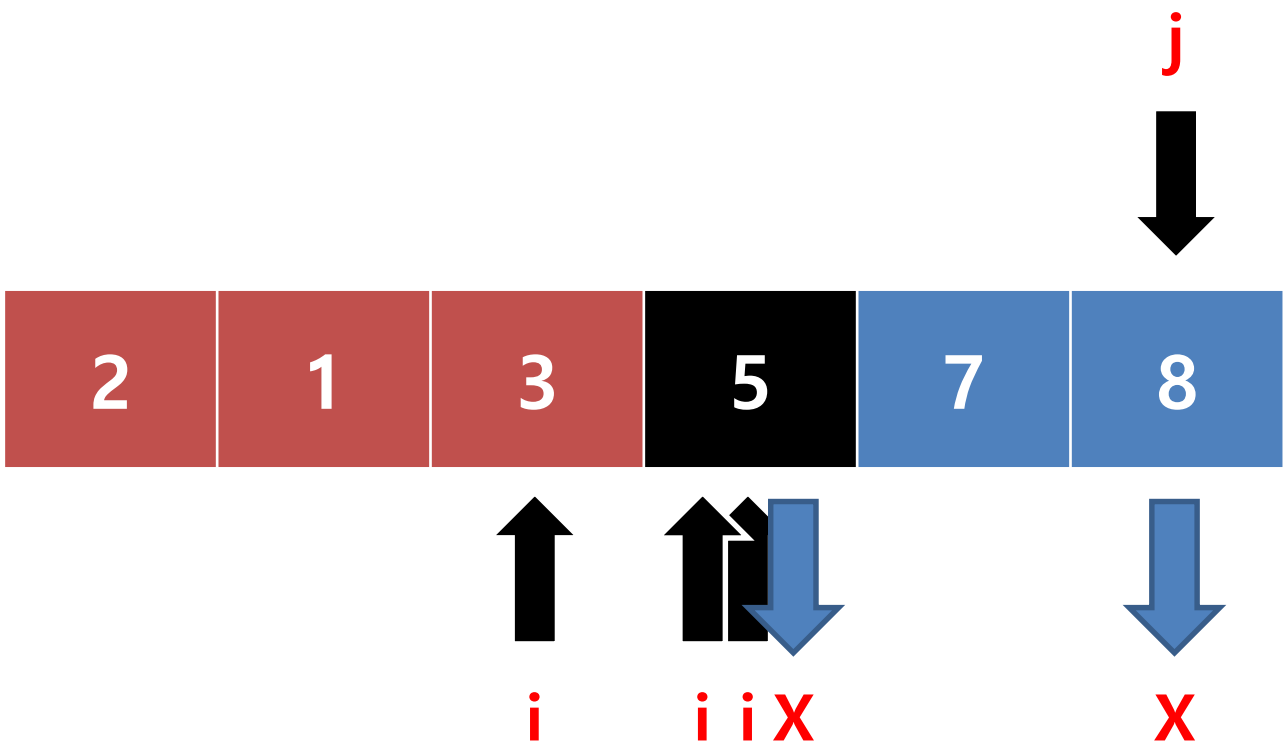
  조건문이 참일 때만 **i를 +1** 시키고, **A[i] ↔ A[j]**를 수행한다.



Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

  반복문이 종료되면 **i를 +1** 시키고, **A[i] ↔ A[j]**를 수행한다.



Partition

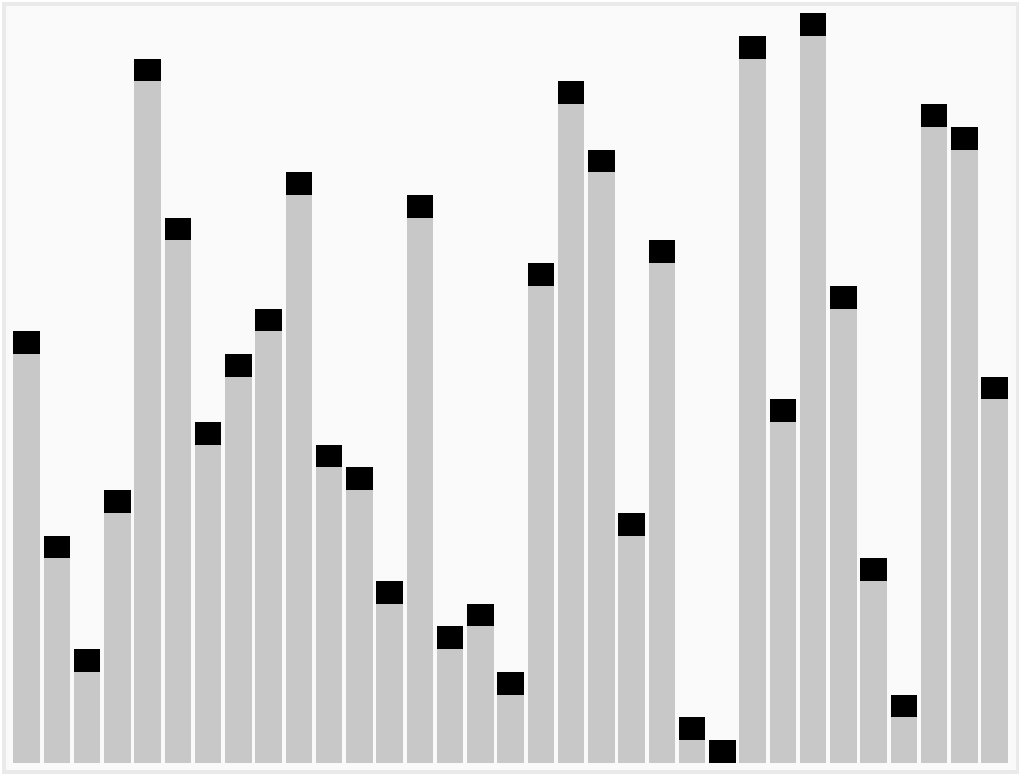
- **Quick Sort의 내부 정렬 과정 (Partition)**

  최종적으로 X가 위치한 곳의 인덱스 **i를 반환**한다.

  QUICKSORT 프로시저는 이를 반환 받은 후
  A[p .. i-1], A[i+1 .. r]로 각각 **재귀 호출**한다.

Partition

- Quick Sort 과정 GIF



Quick Sort

▶ Quick Sort의 내부 정렬 과정

내부 정렬을 담당하는 **PARTITION 프로시저**는
다음과 같은 순서로 동작한다. (교수님 설명 알고리즘)

1) input parameter = A[p .. r], p, r

2) X = A[r]

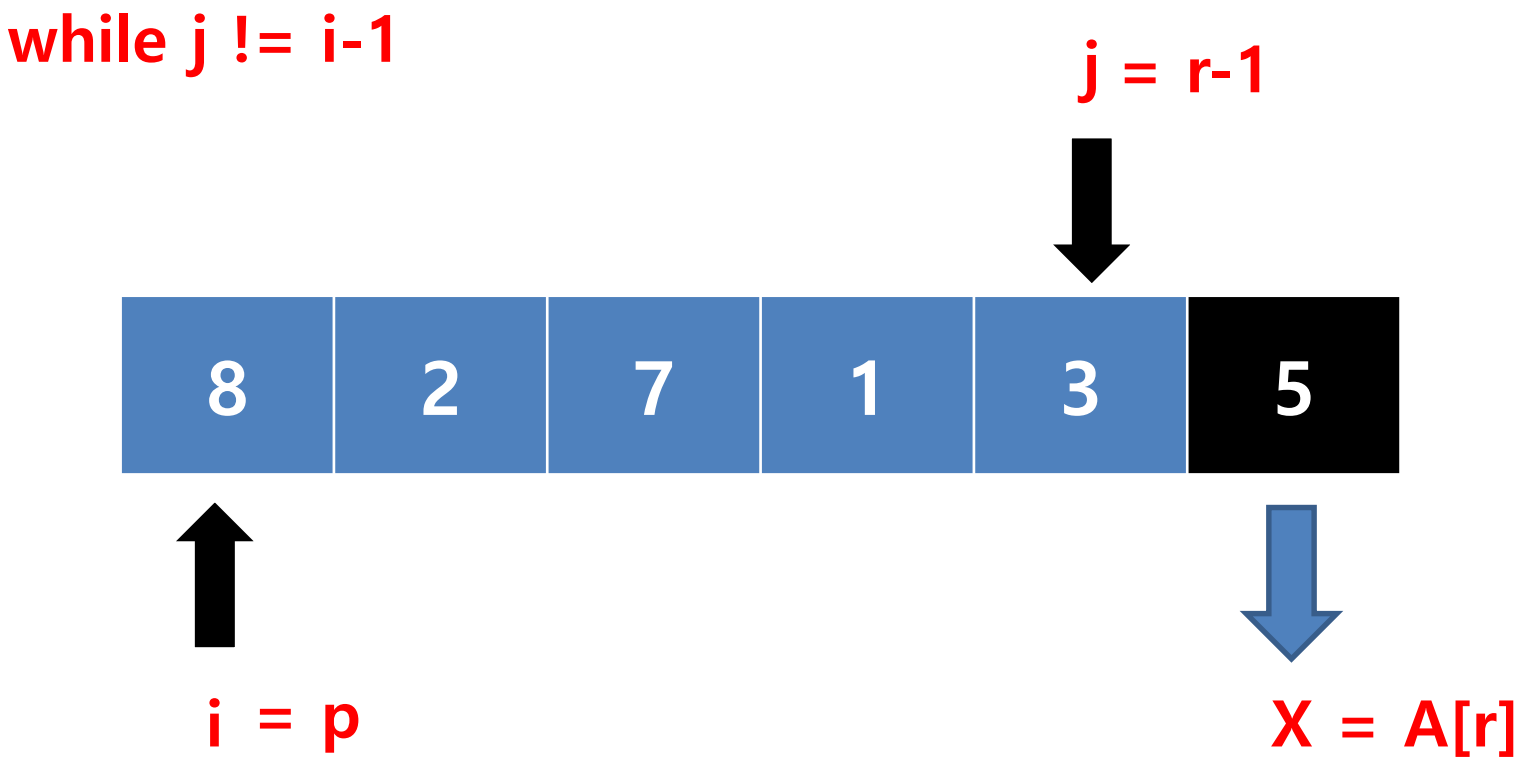
3) 정렬 : **{Value ≤ X} {X} {Value > X}**

4) 반환 : **{X}의 인덱스**

Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

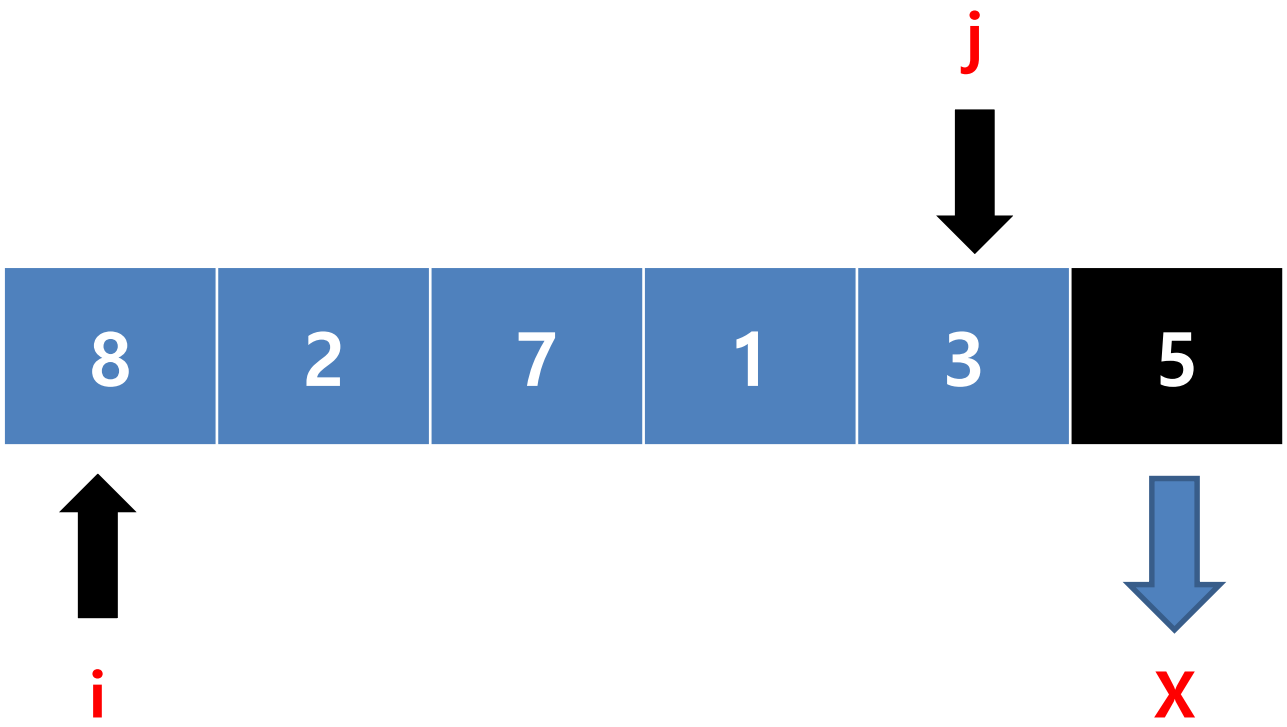
  PARTITION 프로시저의 초기 설정은 다음과 같다.



Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

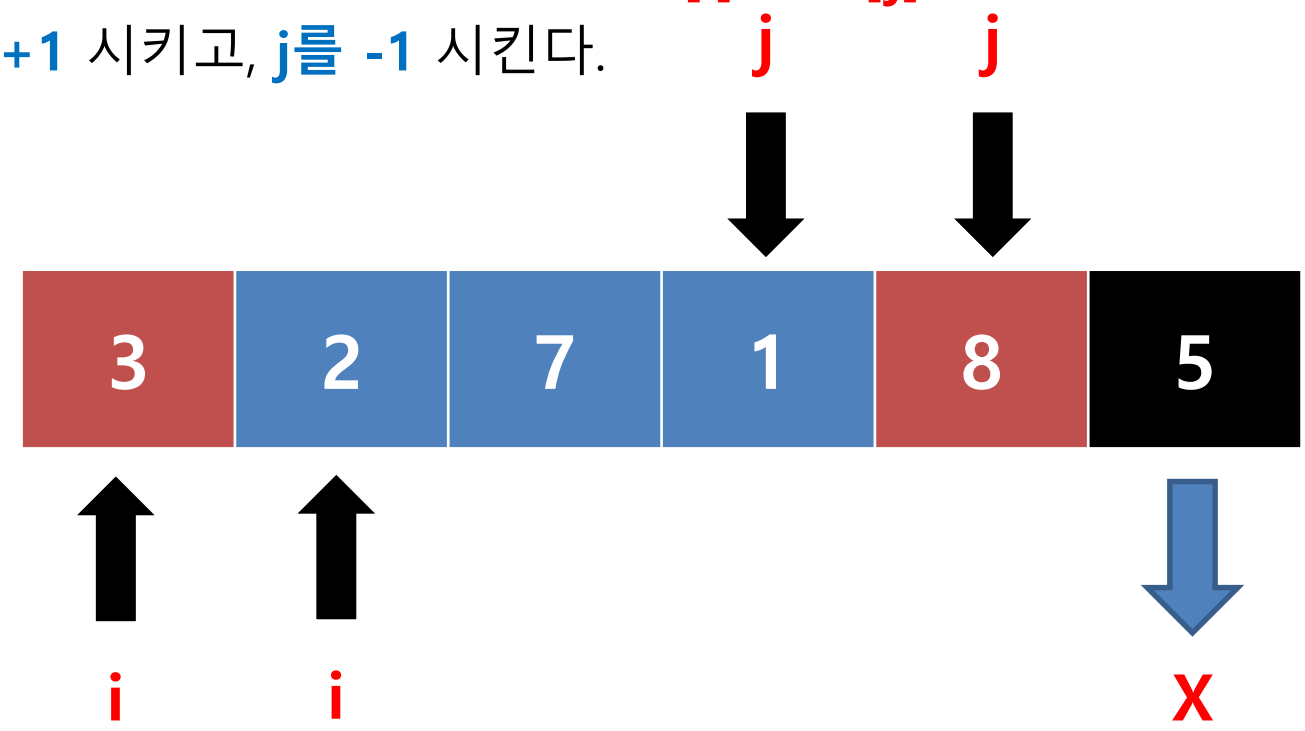
반복문 안에는 조건문 **if A[i] ≤ X**
else if X < A[j]
else 가 존재한다.



Partition

▶ **Quick Sort의 내부 정렬 과정 (Partition)**

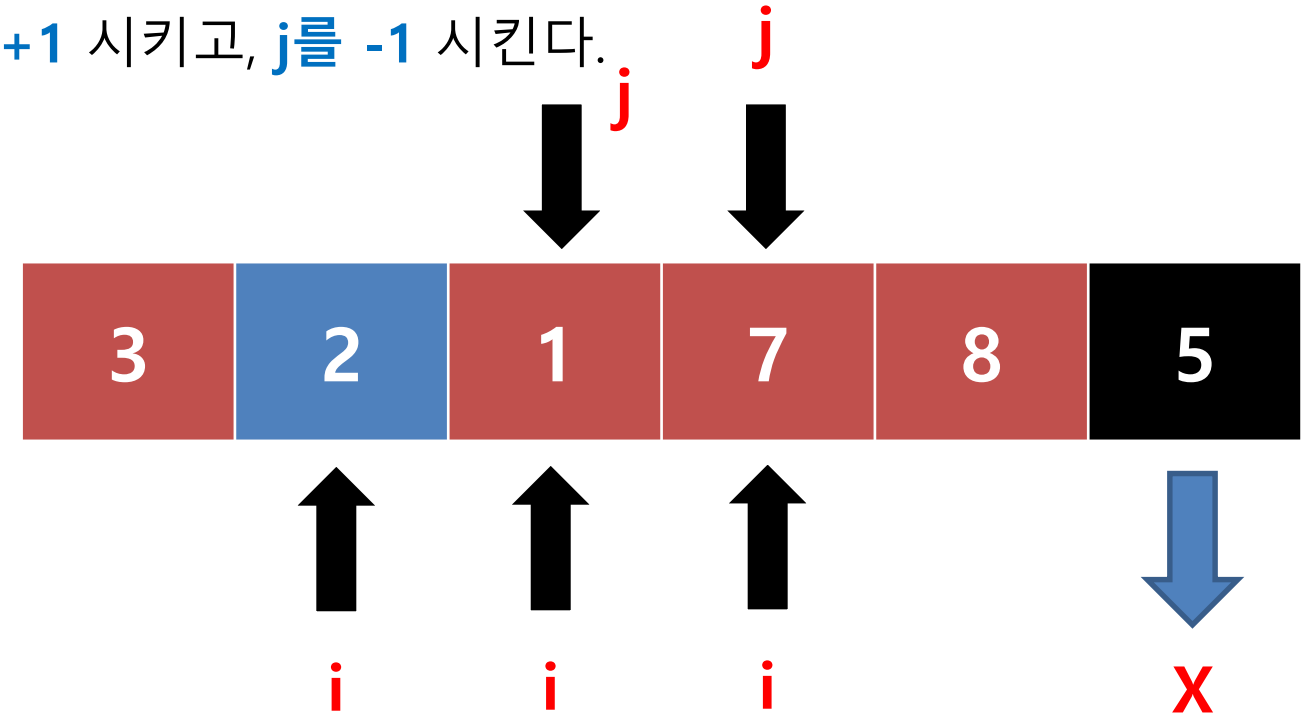
if A[i] ≤ X 조건문이 참인 경우 **i를 +1** 시킨다.
else if X < A[j] 조건문이 참인 경우 **j를 -1** 시킨다.
두 조건에 해당되지 않는 경우 **A[i] ↔ A[j]**를 수행한 후
i를 +1 시키고, **j를 -1** 시킨다.



Partition

▶ **Quick Sort의 내부 정렬 과정 (Partition)**

if A[i] ≤ X 조건문이 참인 경우 **i를 +1** 시킨다.
else if X < A[j] 조건문이 참인 경우 **j를 -1** 시킨다.
두 조건에 해당되지 않는 경우 **A[i] ↔ A[j]**를 수행한 후
i를 +1 시키고, **j를 -1** 시킨다.

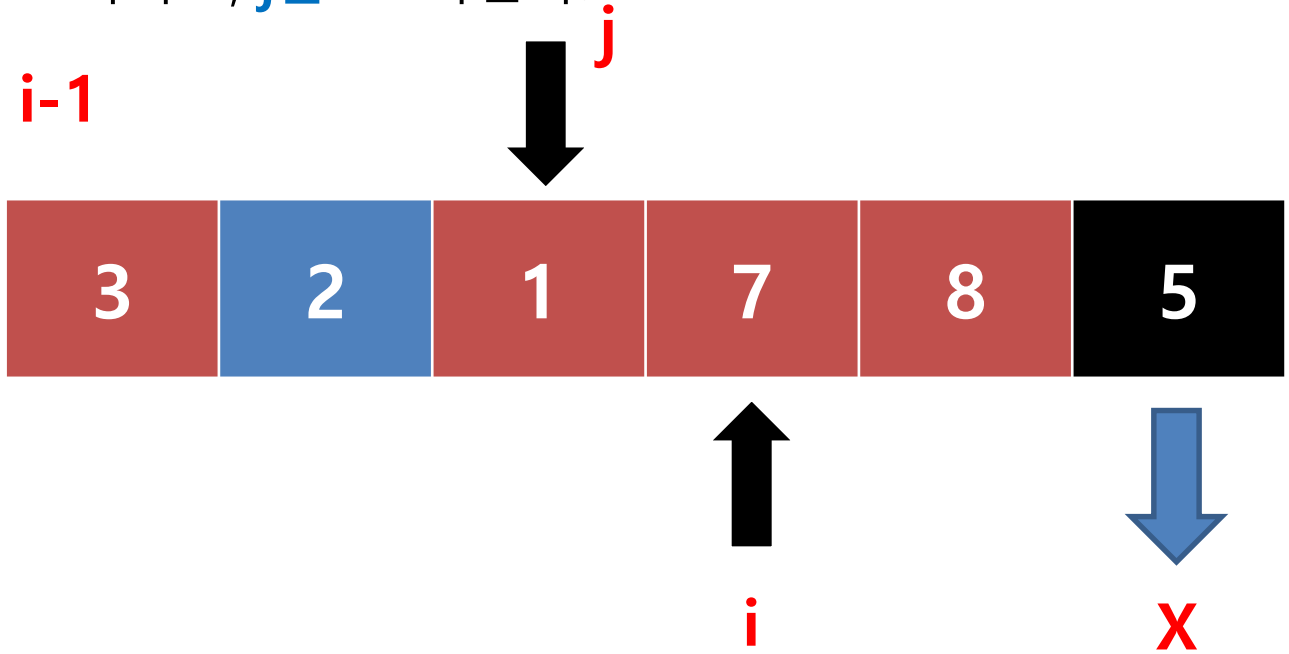


Partition

▶ **Quick Sort의 내부 정렬 과정 (Partition)**

if A[i] ≤ X 조건문이 참인 경우 **i를 +1** 시킨다.
else if X < A[j] 조건문이 참인 경우 **j를 -1** 시킨다.
두 조건에 해당되지 않는 경우 **A[i] ↔ A[j]**를 수행한 후
i를 +1 시키고, **j를 -1** 시킨다.

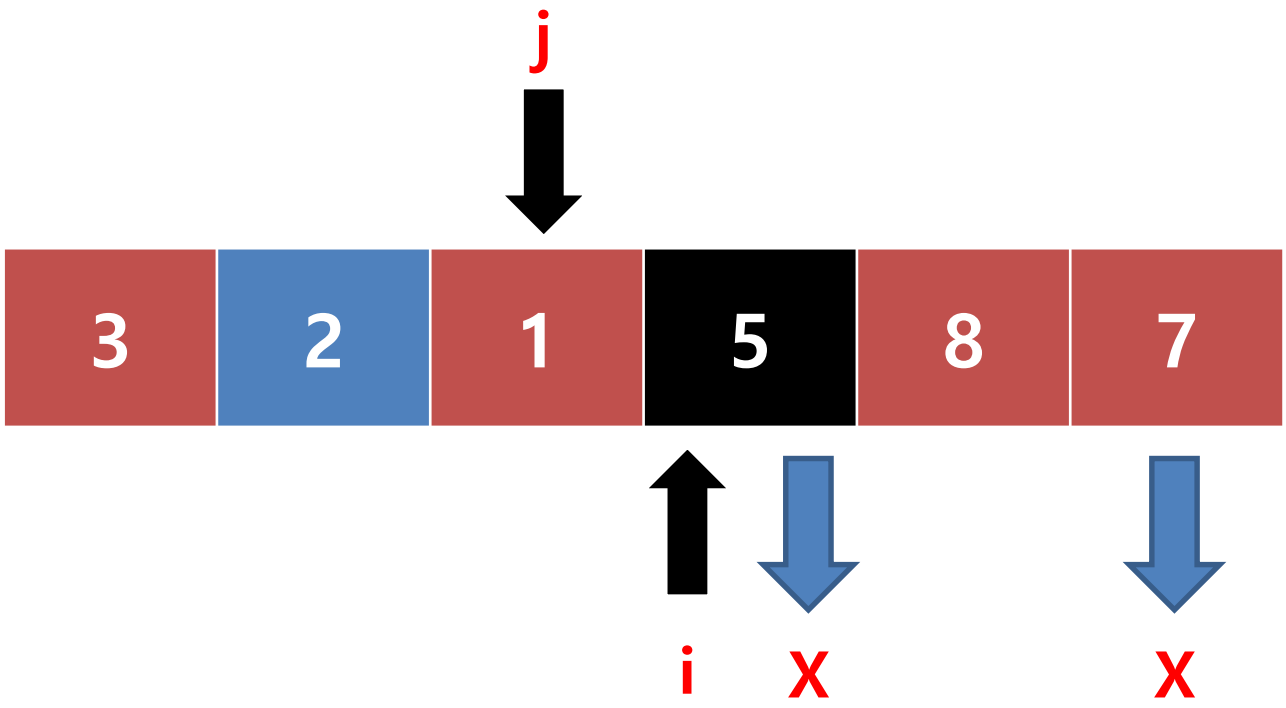
while j != i-1



Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

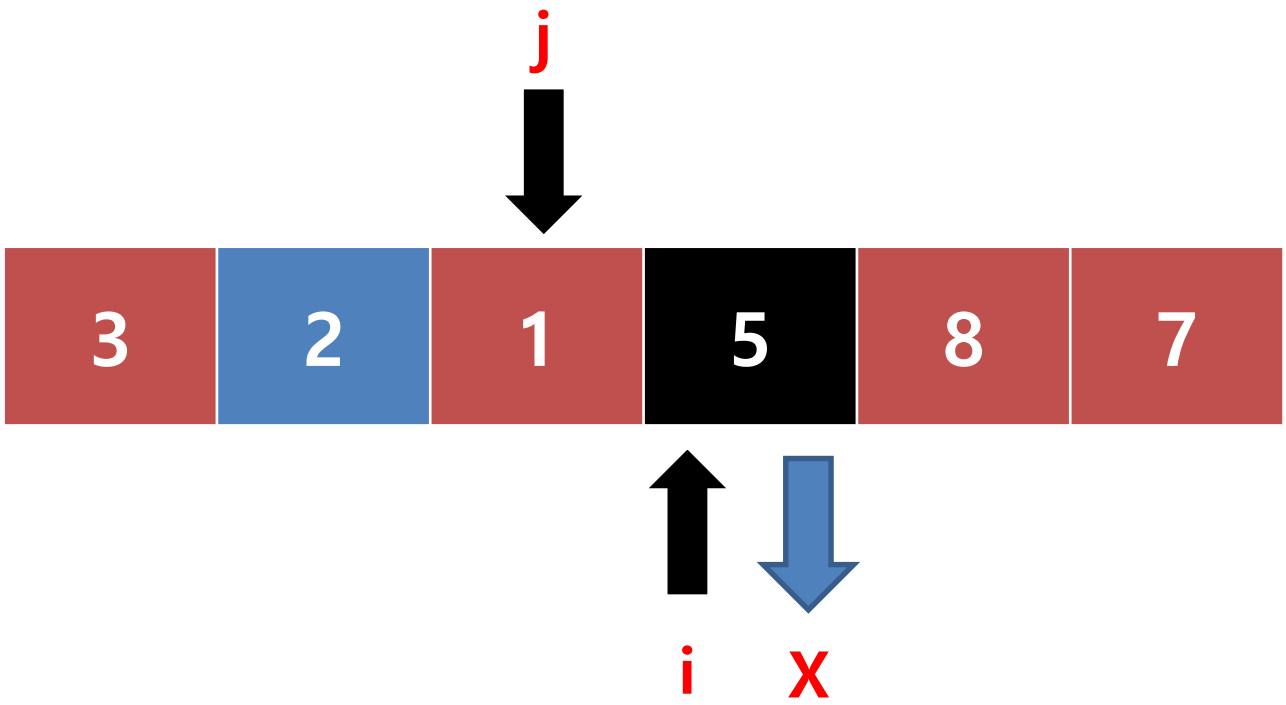
  반복문이 종료되면 **A[i] ↔ A[r]**를 수행한다.



Partition

- **Quick Sort의 내부 정렬 과정 (Partition)**

  최종적으로 X가 위치한 곳의 인덱스 **i를 반환**한다.
  QUICKSORT 프로시저는 이를 반환 받은 후
  A[p .. i-1], A[i+1 .. r]로 각각 **재귀 호출**한다.



Partition

```
PARTITION (A, p, r)
    x ← A[ r ]
    i ← p – 1
    for j ← p to r – 1
        do if A[ j ] ≤ x
            then i ← i + 1
                 A[ i ] ↔ A[ j ]
    i ← i + 1
    A[ i ] ↔ A[ r ]
    return i
```

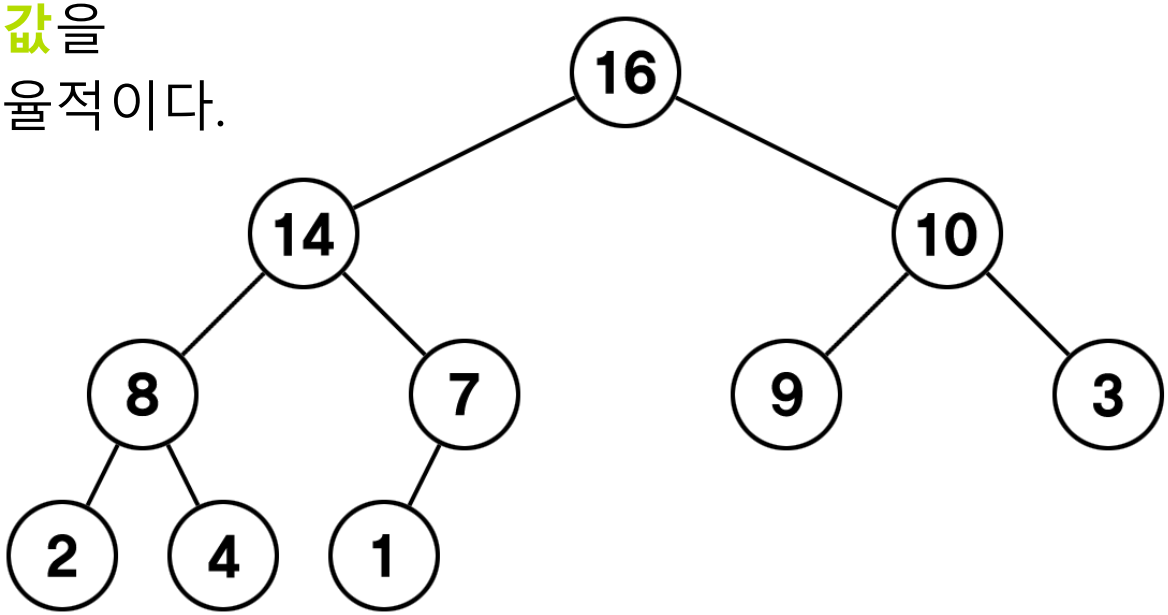
```
PARTITION (A, p, r)
    x ← A[ r ]
    i ← p
    j ← r – 1
    while j ≠ i – 1
        do if A[ r ] ≥ A[ i ]
            then i ← i + 1
        else if A[ r ] < A[ j ]
            then j ← j – 1
        else
            then A[ i ] ↔ A[ j ]
                 i ← i + 1
                 j ← j – 1
    A[ i ] ↔ A[ r ]
    return i
```

Heap

▶ Heap이란? (1/3)

완전 이진 트리를 기반으로 하는 배열형 자료구조.

최대값 또는 **최소값**을
빠르게 찾는데 효율적이다.



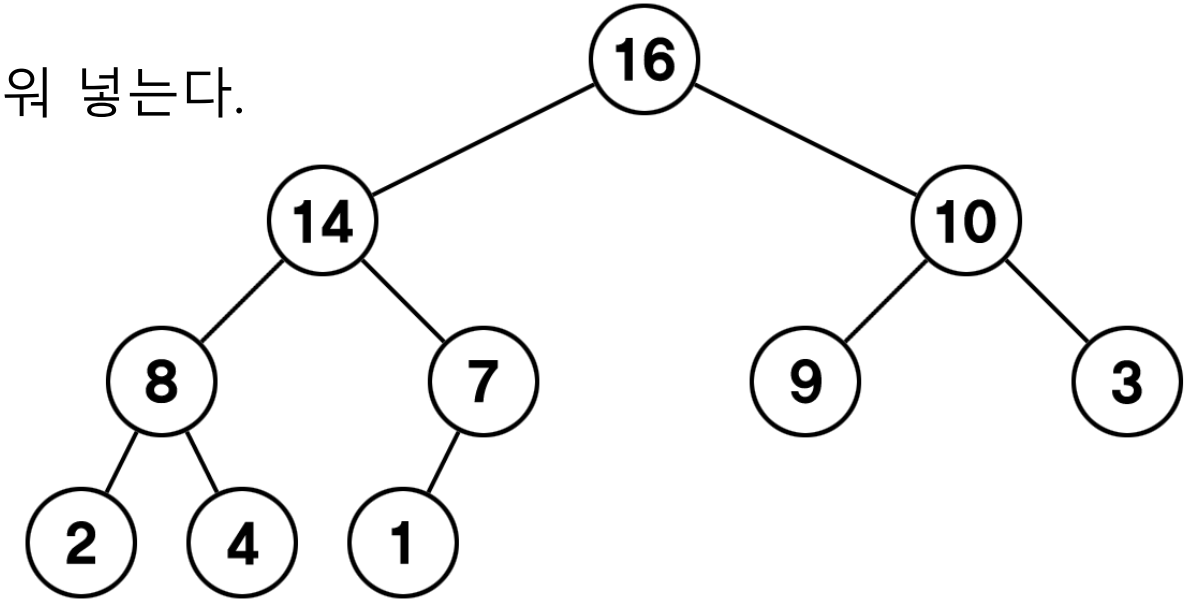
16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

Heap

▶ Heap이란? (2/3)

이 트리는 가장 낮은 층을 제외하고는 완전히 차 있고,

가장 낮은 층은
왼쪽에서부터 채워 넣는다.



16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

Heap

▶ Heap이란? (3/3)

어떤 노드가 저장된 인덱스를 i라 할 때,

PARENT*(i)*

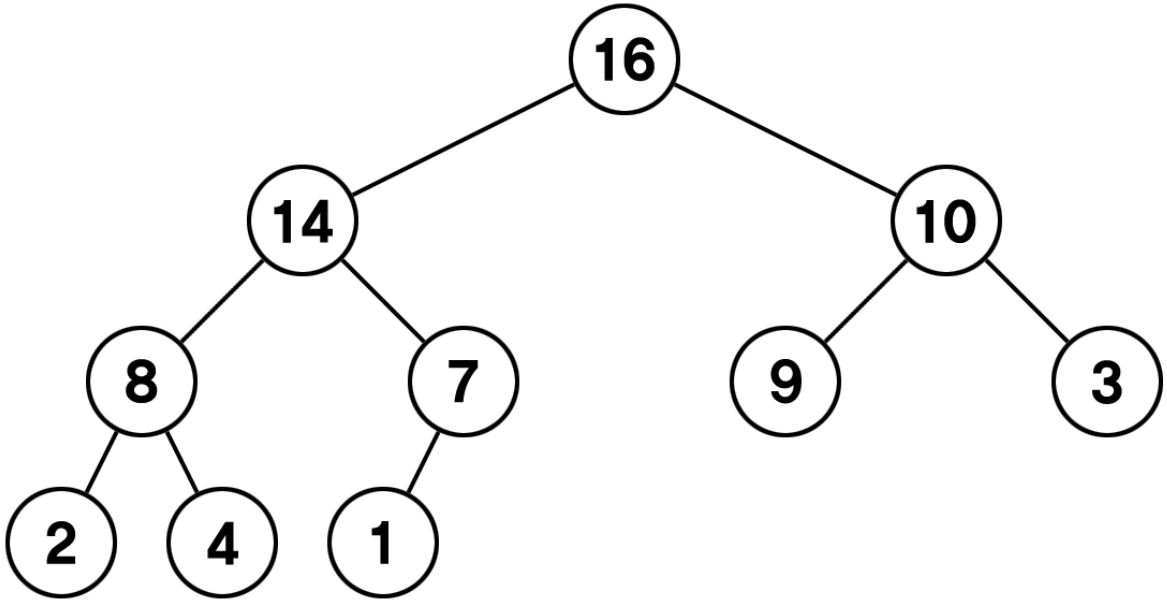
return $\lfloor i/2 \rfloor$

LEFT-CHILD*(i)*

return $2i$

RIGHT-CHILD*(i)*

return $2i+1$



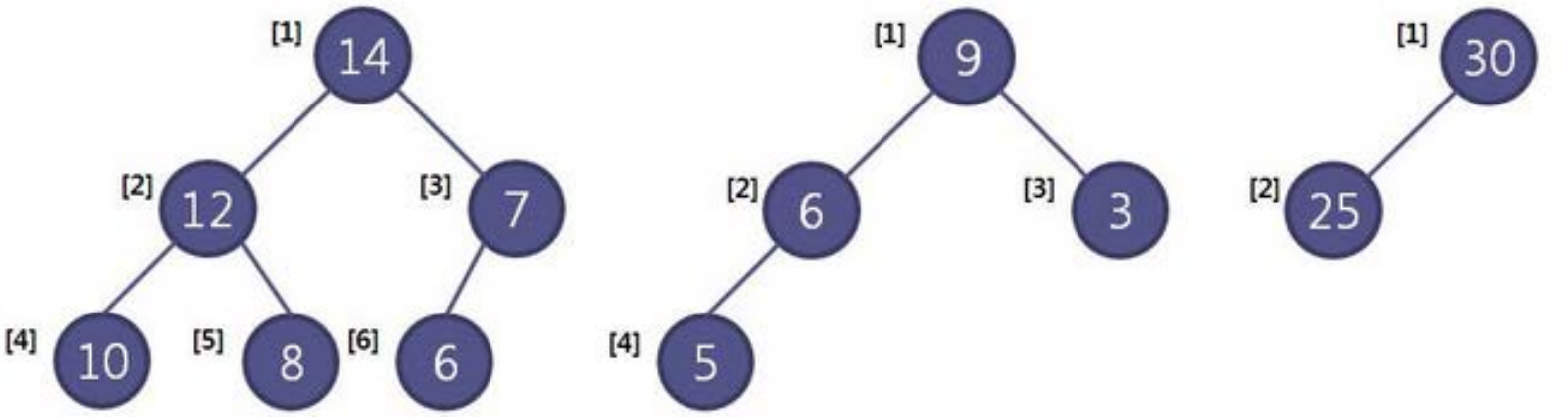
16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

Heap

▶ 최대 힙과 최소 힙

1) 최대 힙 (Max Heap) : **루트에 최대값이 저장**된다.

부모의 키 값 ≥ 자식의 키 값



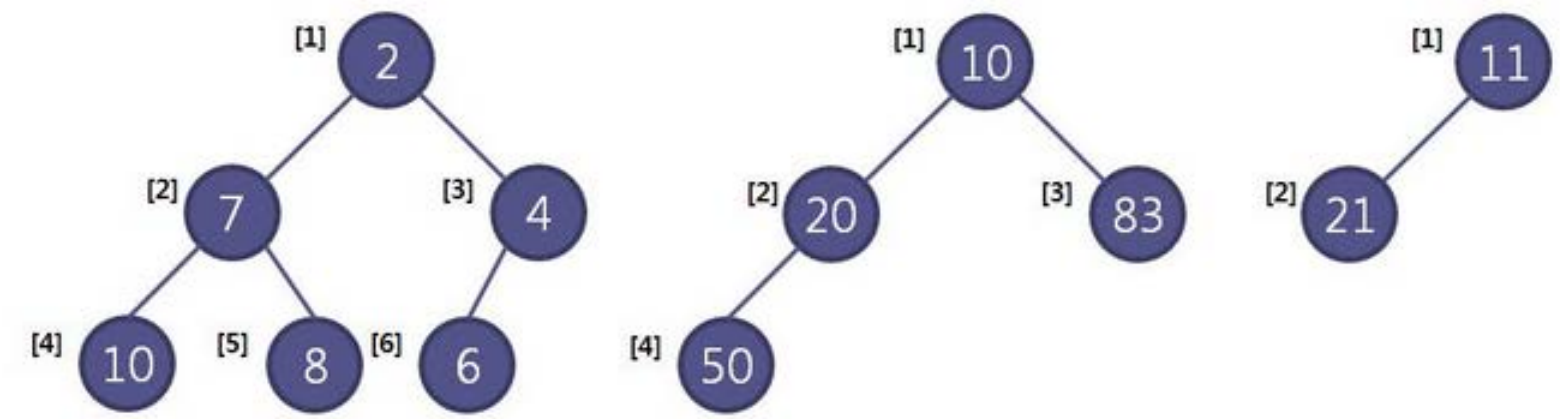
[그림 1] 최대 히프의 예

Heap

▶ 최대 힙과 최소 힙

2) 최소 힙 (Min Heap) : **루트에 최소값이 저장**된다.

부모의 키 값 ≤ 자식의 키 값



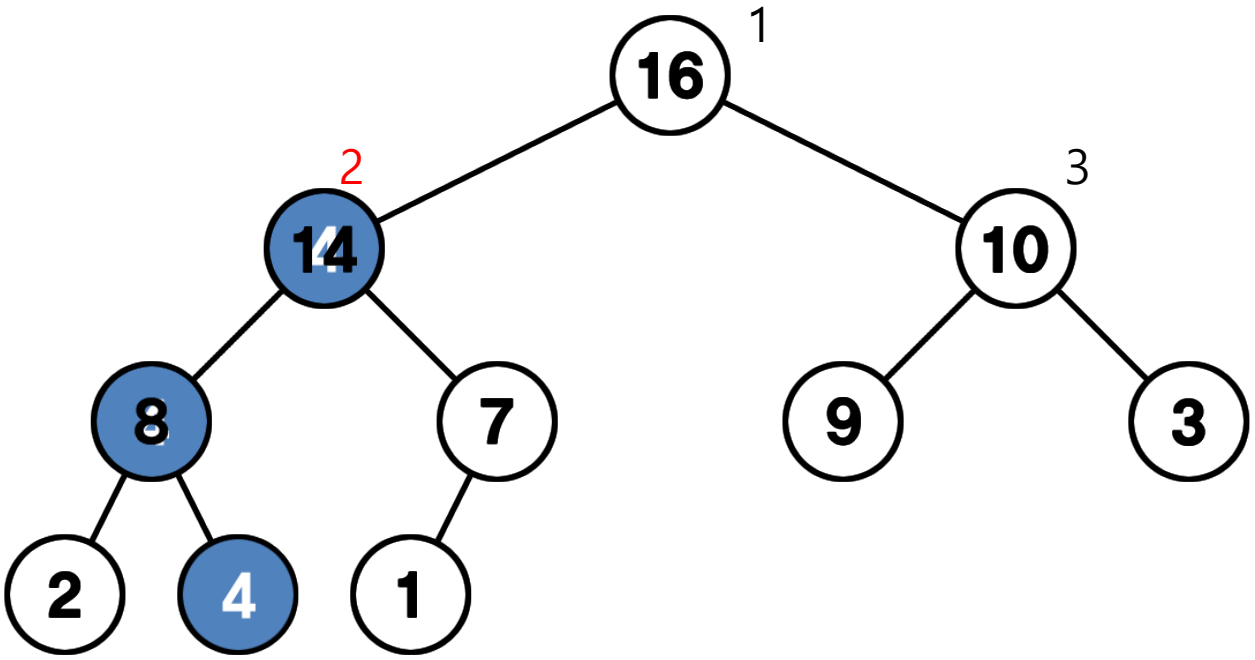
[그림 2] 최소 히프의 예

Heap

▶ 힙의 구현

MAX-HEAPIFY (S,2)

특정 노드를 기점으로, 트리를 내려가며 최대 힙이 되도록 함



Heap

▶ 힙의 구현

```
MAX-HEAPIFY(A,i)
    L ← LEFT-CHILD(i)
    R ← RIGHT-CHILD(i)
    if L ≤ heap_size[A] and A[L] > A[i]
        then largest ← L
        else largest ← i
    if R ≤ heap_size[A] and A[R] > A[largest]
        then largest ← R
    if largest ≠ i
        then A[i] ↔ A[largest]
            MAX-HEAPIFY(A,largest)
```

Heap

▶ 힙의 구현

BUILD-MAX-HEAP(A)

입력 받은 **배열 A를 최대 힙으로 만드는 함수**.

자식을 갖는 마지막 노드부터 루트 노드까지
순서대로 검사하며 MAX-HEAPIFY 함수를 실행한다.

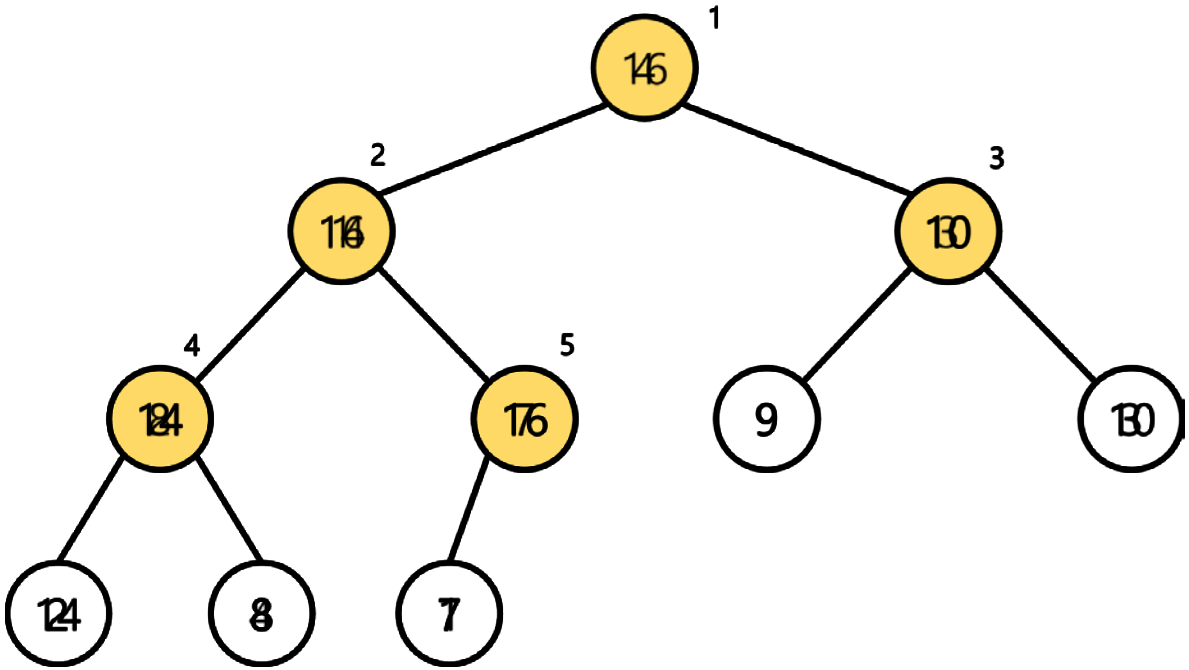
```
BUILD-MAX-HEAP(A)
    heap_size[A] ← length[A]
    for i ← ⌊length[A]/2⌋ downto 1
        do MAX-HEAPIFY(A,i)
```

Heap

- 힙의 구현

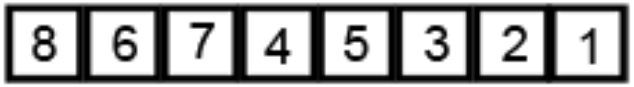
BUILD-MAX-HEAP(A)

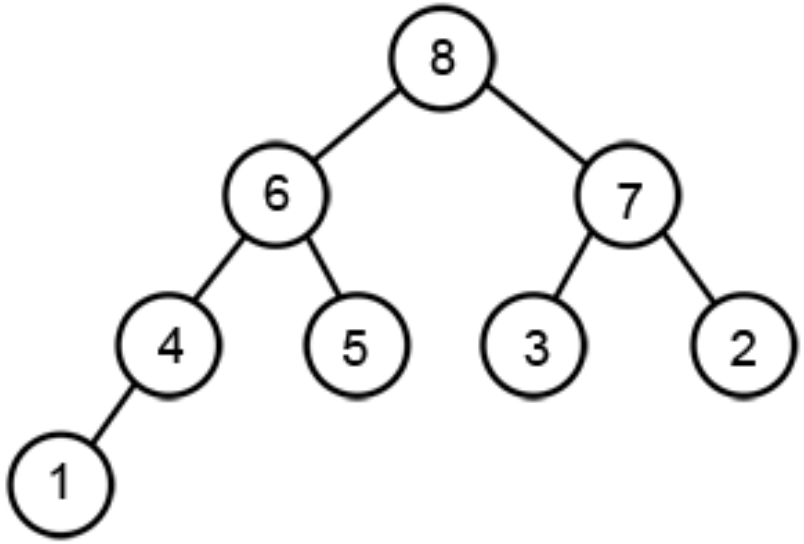
4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10



Heap

- **Complete Heap Sort**
  loop Extract_max





Priority Queue

▶ Priority Queue란?

힙 자료구조를 이용한 효율적인 큐(Queue) 알고리즘.

노드에 저장되는 키 값을 우선순위로 하고
우선순위가 최대/최소인 작업을 먼저 꺼내서 처리하는 방법으로,

작업 스케줄링, 네트워크 트래픽 제어 등에 활용된다.

힙 자료구조에 삽입, 추출 프로세스를 추가하는 것만으로
간단하게 구현할 수 있으며,
힙과 마찬가지로 최대/최소 우선 순위 큐로 나누어진다.

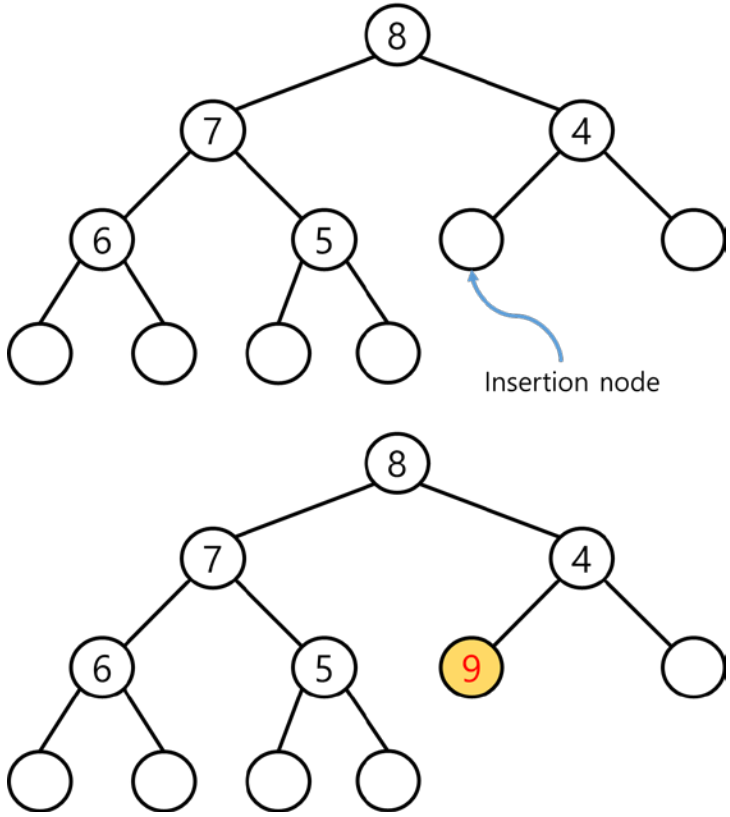
Priority Queue

▶ 최대 우선 순위 큐의 삽입 과정

1) 트리의 가장 마지막 부분에
새로운 노드를 삽입한다.

2) 삽입한 노드와 부모 노드의
크기를 비교한다.

3) 부모 노드가 더 크면
삽입 프로세스를 종료한다.

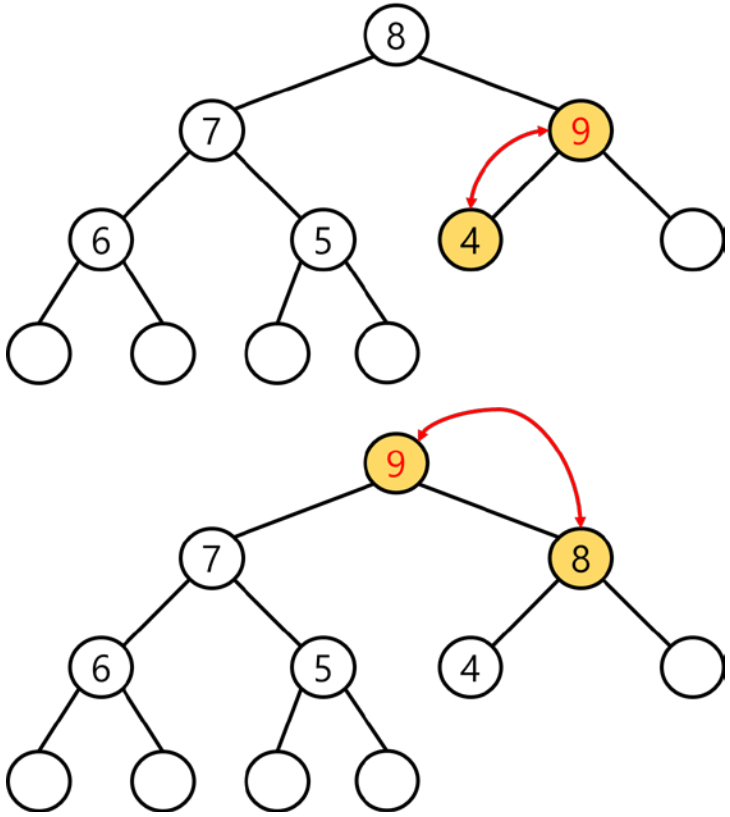


Priority Queue

▶ 최대 우선 순위 큐의 삽입 과정

4) 삽입한 노드가 더 크면
이를 부모 노드와 교체한다.

5) 프로세스가 종료되거나
루트에 도달할 때까지
상방향으로 연산을 반복한다.



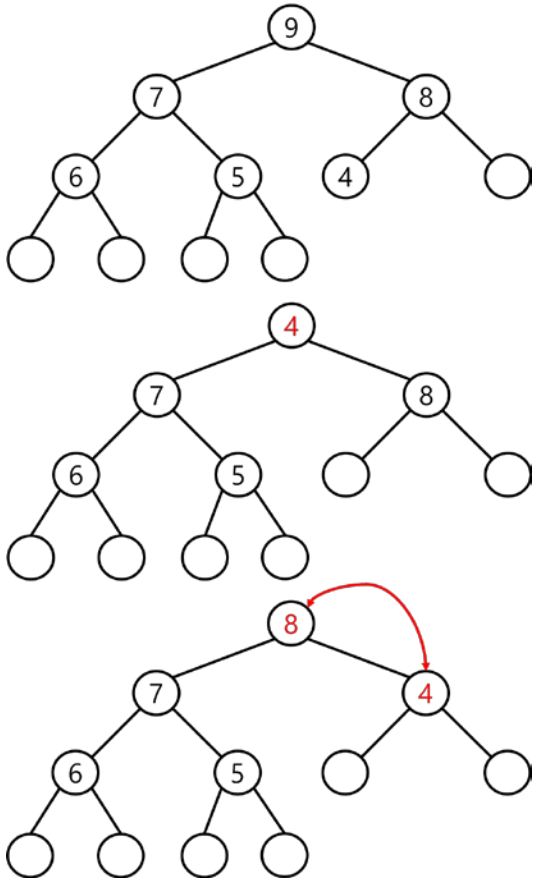
Priority Queue

▶ 최대 우선 순위 큐의 추출 과정

1) 루트 노드를 추출한다.

2) 가장 마지막에 위치한 노드를
   루트 위치로 이동시킨다.

3) 새로운 루트 노드를 기준으로
   MAX-HEAPIFY를 수행한다.



Practice / Homework

1. 수업 PPT에 제시된 Partition 알고리즘을 사용하여 첨부된 데이터를 퀵 정렬하고 출력하는 1개의 프로그램을 구현하라.

a . 정렬 결과를 파일로 출력할 때, 파일명은 다음과 같이 한다.

"학번_quick.txt"

b . 정렬 후 출력된 파일은,

Best-case에 있는 파일과 완전히 같은 파일이 되어야 한다.

즉, 파일 쓰기를 할 때 **동일한 양식**으로 작성하여야 한다.

Practice / Homework

1. 수업 PPT에 제시된 Partition 알고리즘을 사용하여 첨부된 데이터를 퀵 정렬하고 출력하는 1개의 프로그램을 구현하라.

c . 다음 두 가지 함수가 **반드시** 구현되어 있어야 한다.

```
partition(A, p, r);
quickSort(A, p, r);
```

d . 2주차 때 진행했던 insertion sort, merge sort 알고리즘과 비교, 분석하여 보고서 작성

Practice / Homework

2. data_heap.txt 파일의 데이터로 Max Heap을 만들고
삽입, 추출 프로세스를 추가하여 Max Priority Queue를 구현하라.
삽입, 추출을 수행할 때마다 남은 노드 목록을 출력하여야 한다.

1) 하나의 노드와, 이를 배열로 갖는 힙 구조체를 정의한다.

2) 힙을 생성하고 공간을 할당한 후에,
파일 읽기가 수행되는 순서대로 각 데이터 정보를 저장한다.

3) MAX-HEAPIFY와 BUILD-MAX-HEAP을 구현 및 사용하여,
2번에서 얻은 배열(힙)을 Max Heap으로 바꾼다.

Practice / Homework

2. data_heap.txt 파일의 데이터로 Max Heap을 만들고
삽입, 추출 프로세스를 추가하여 Max Priority Queue를 구현하라.
삽입, 추출을 수행할 때마다 남은 노드 목록을 출력하여야 한다.

4) 삽입 함수 insert()와 추출 함수 extractMax()를 구현한다.

5) 완성된 프로그램을 구동하고, 파일을 큐에 읽어 들인 후에도 사용자가 노드를 추가하거나 우선도가 가장 높은 노드를 추출(삭제) 등 다음 페이지에 나와있는 기능들이 동작 할 수 있도록 메인 함수를 완성한다.

Practice / Homework

2. data_heap.txt 파일의 데이터로 Max Heap을 만들고
삽입, 추출 프로세스를 추가하여 Max Priority Queue를 구현하라.
삽입, 추출을 수행할 때마다 남은 노드 목록을 출력하여야 한다.
아래 기능의 함수들은 전부 구현되어야 한다.

a. insert(S, x) - S에 원소 x를 새로 넣는다.

b. max(S) - S에서 키값이 최대인 원소를 리턴한다.

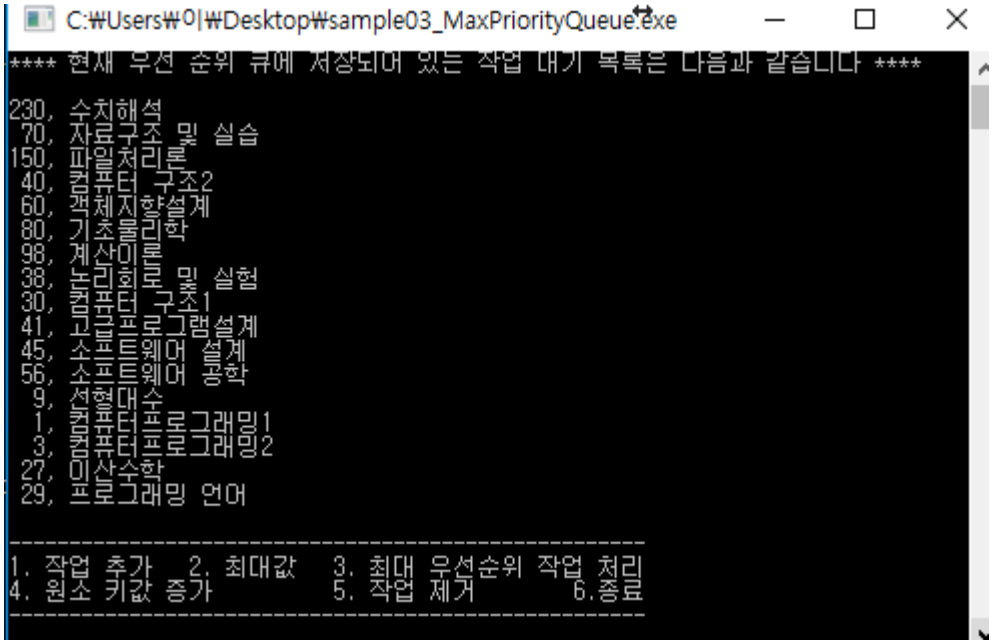
c. extract_max(S) - S에서 키값이 최대인 원소를 제거한다.

d. increase_key(S, x, k) - 원소 x의 키값을 k로 증가시킨다. 이때 k는 x의 현재 키값보다 작아지지 않는다고 가정한다.

e. delete(S, x) - S에서 노드 x를 제거한다. 제거 후 Max heap 유지

Practice / Homework

▶ **결과 화면 예시 : sample03_MaxPriorityQueue.exe**



- 명령어 입력에 따라 작업을 수행한 결과를 화면에 출력

Practice / Homework

※ 그 외 실습 과제 수행 중 유의 사항

a . 과제 제출은 사이버 캠퍼스에 과제 제출,
구현한 소스파일(.java)과 보고서를 zip으로 압축하여 보낼 것.
(**프로젝트 폴더채로 압축하지 않도록**)

b . 과제 하나당 하나의 파일만 생성할 것.

c . **과제 평가는 별도의 input data를 사용**함. (양식은 동일)

d . 코드에 대한 설명은 보고서에 적을 것.

Practice / Homework

과제 제출 안내	
제출 파일	Quick Sort, Priority Queue JAVA코드 보고서 파일 (1개)
제출 기한	9월 28일 (목) **실습 수업 시간 전까지**

과제 평가 감점 사항	
제출 지연 (수업 시작부터)	- 50% / 1주
요구 사항 누락 / 결과값 불일치	- 10 ~ 20% / 1개
코드 Error	- 50 ~ 100%
과제 Copy	0점