Sorting and Order Statistics (Chapters 6~9)

김동진
(NHN NEXT)

목 차

- Heapsort (Chapter 6)
- Quicksort (Chapter 7)
- Sorting in Linear Time (Chapter 8)
- Medians and Order Statistics (Chapter 9)

Sort가 중요한 이유

- 정보 정렬이 필요한 응용 분야가 많음
  - 검색 결과를 ranking 순서로 사용자에게 제공
  - 고객 id를 번호순으로 정렬
  - 기록을 날짜순으로 정렬 등
- 비교 기반 정렬의 lower bound가 증명되어 있음
  - 다른 문제의 lower bound를 증명하는데 사용 가능
  - 새로 개발한 비교 기반의 알고리즘으로 sorting을 하는데 time complexity가 $O(n \lg n)$보다 빠르다면 개발한 알고리즘은 잘못된 알고리즘임.

In-place Sort

- 주어진 입력 배열 외에 O(1)의 추가 메모리를 사용하여 정렬하는 sort algorithm

- Insertion sort
  - In-place
  - O(1)의 보조 메모리 사용

- Merge sort
  - in-place sort 알고리즘이 아님
  - 입력 배열 크기의 보조 메모리 사용

Heap Sort 특징

- Heap 자료 구조를 사용하여 정렬하는 알고리즘
- Time Complexity
  - $O(n \lg n)$
- In-place 알고리즘
  - O(1)의 보조 메모리만 사용

Tree 특성 확인 Problem

- Binary tree가 주어져 있다.
- 주어진 binary tree가 다음 특성을 만족하는지 검사하려고 한다.
  - 각 node의 값은 자신의 child node들의 값보다 크거나 같다.
- 주어진 특성을 만족하면 1을 return하고 아니면 0을 return하는 함수를 코딩하시오.
  - C-stype code
    - int checkTree(node_t *node);

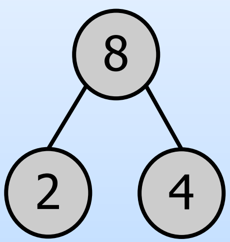
Complete Binary Tree Problem

- Complete binary tree는 배열에 저장할 수 있다.
- Node개수가 n개인 complete binary tree가 배열에 저장되어 있다.
  - 각 노드는 integer 값을 갖는다.
  - 각 노드가 저장된 배열에서의 위치를 node의 id로 정의하자.
  - tree의 root는 배열의 1번에 저장한다. 즉, root node의 id는 1이다.
- 주어진 노드의 child에 저장된 값을 출력하는 코드를 작성하시오.
  - C-style API
    - void printChildren(int *treeArr, int nodeNum, int givenNodeId);

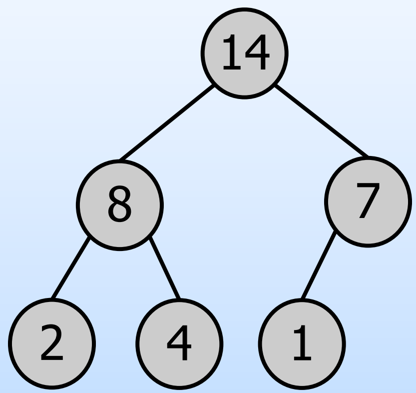
Heap의 정의

- Max heap은 각 노드의 값이 children의 값보다 크거나 같은 complete binary tree이다.
- Min heap은 각 노드의 값이 children의 값보다 작거나 같은 complete binary tree이다.

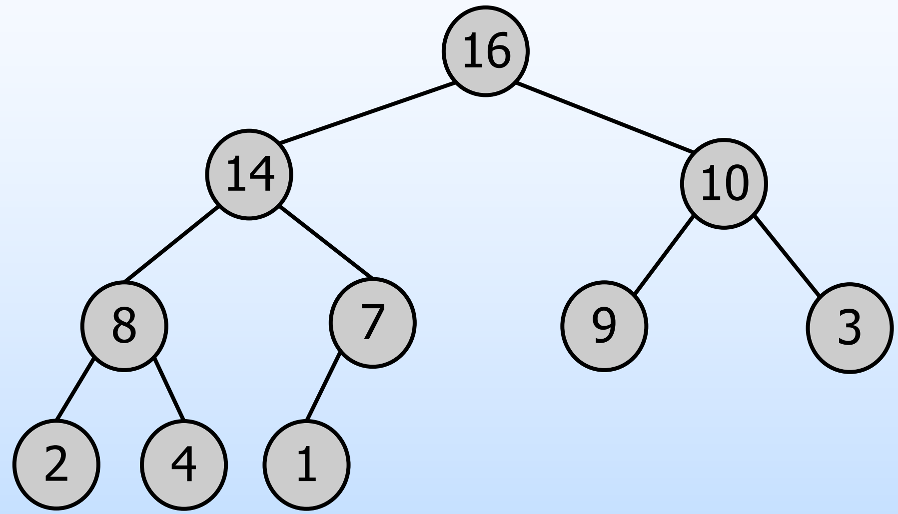
Max Heap 예제



예제 1

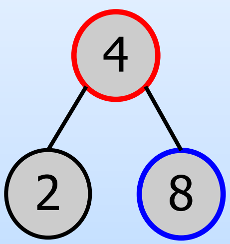


예제 2



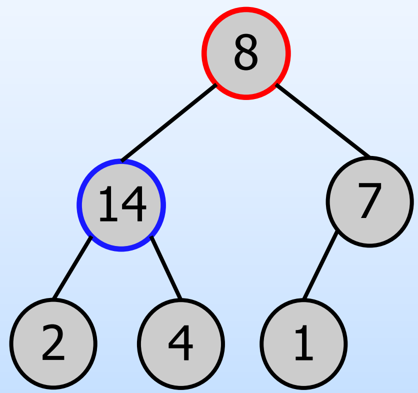
예제 3

Max Heap이 아닌 예제



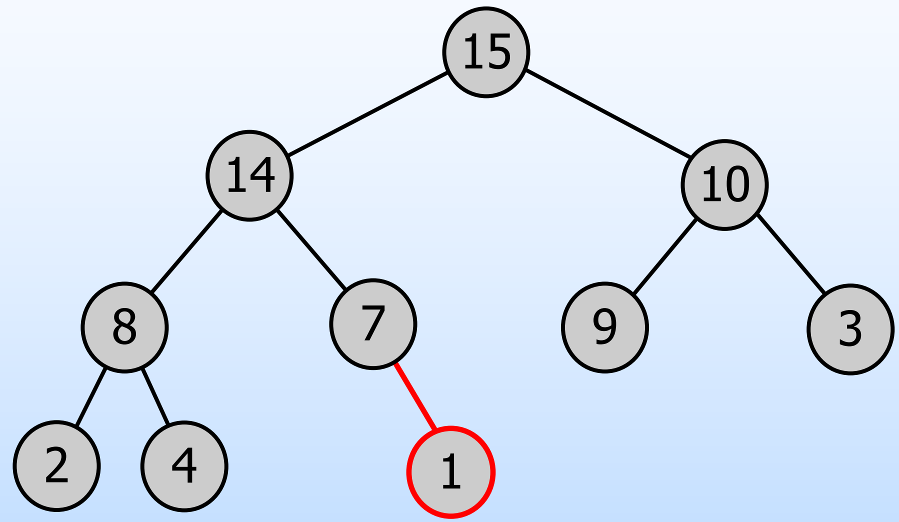
예제 1

4의 right child가
4보다 커서
max heap이 아님



예제 2

8의 left child가
8보다 커서
max heap이 아님



예제 3

Complete binary tree가 아님.
7의 left child가 없고
right child만 있어서 heap이 아님

Max Heap의 특징

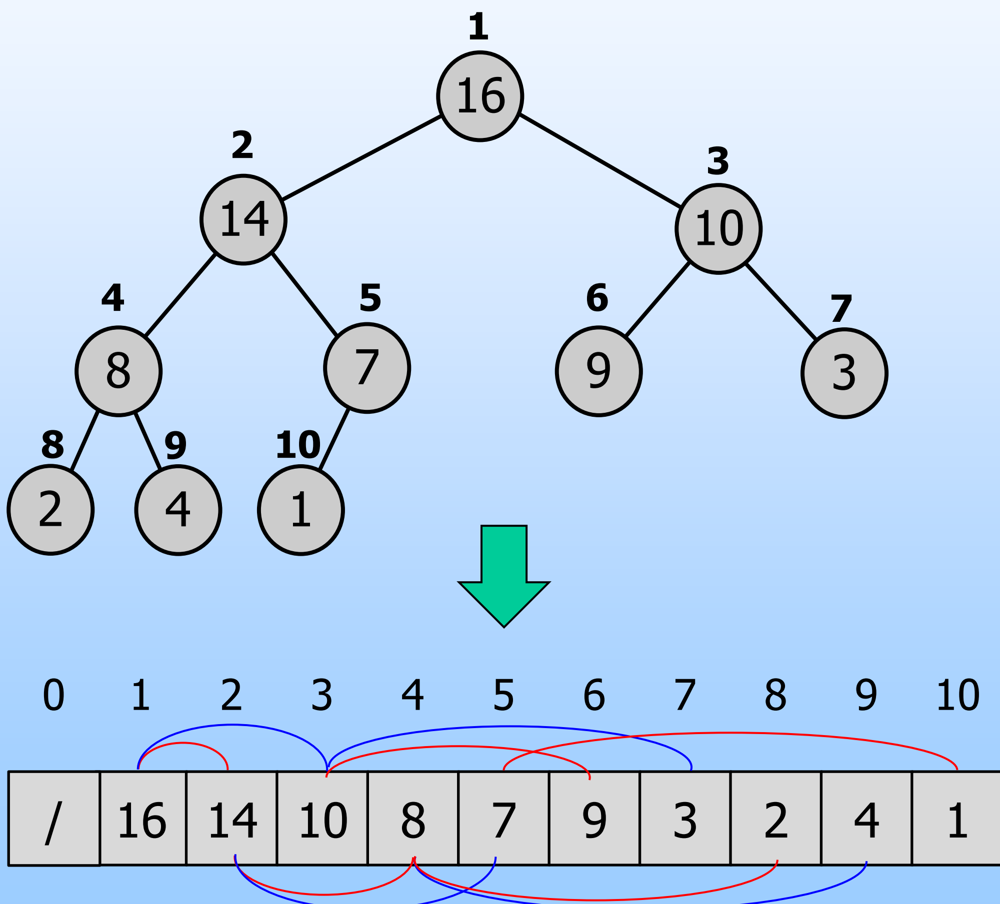
- 최대값 검색이 빠름
  - Root node에 있는 값이 제일 큼
  - 따라서, 최대값을 찾는 데 소요되는 시간이 O(1)

- 자료 구조 관리의 편리함
  - Complete binary tree라서 배열을 사용하여 효율적으로 관리할 수 있음

- Complete binary tree이므로 max-heap의 height는?
  - $\Theta(\lg n)$

Max Heap의 배열 기반 표현

- 구성 노드 개수가 n개일 때
  - 노드의 값을 배열의 1번 위치부터 n번 위치에 저장



i 번째 노드에 대해서

parent(i) = $\lfloor i/2 \rfloor$

left_child(i) = $2i$

right_child(i) = $2i + 1$

단, i 가 1이면 i 가 root

결과가 n보다 크면 child 없음

빨간선: left child

파란선: right child

Max Heap 연산 종류

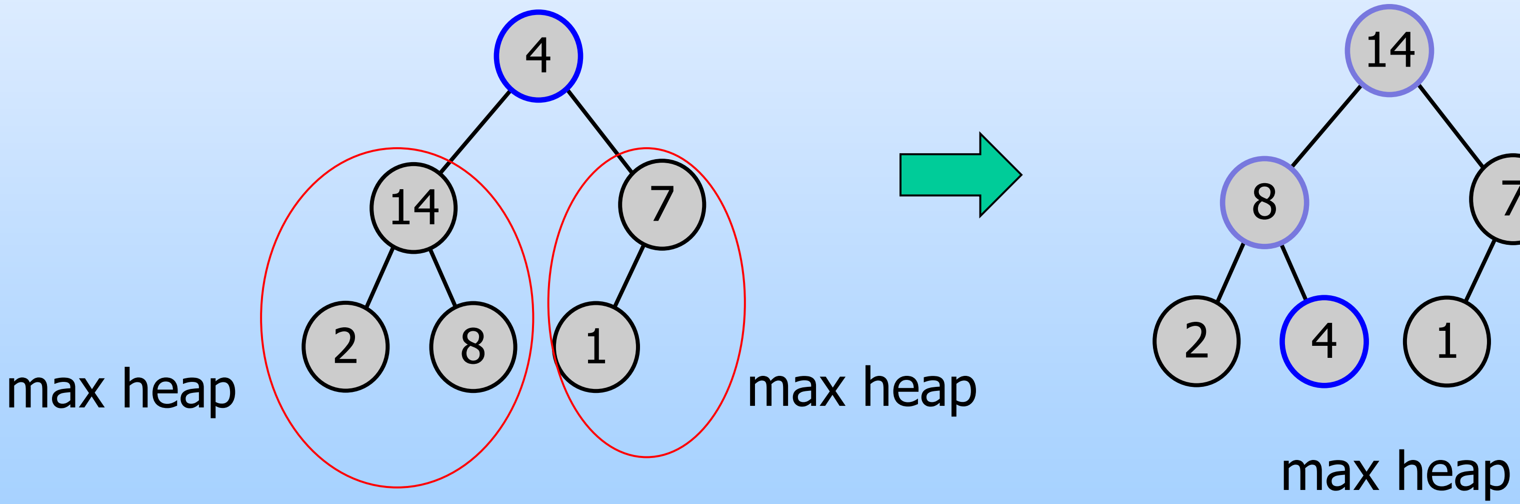
- Max heap 생성
  - 배열이 주어지면 linear time에 max heap을 생성

- Heap 갱신
  - 최대값 노드를 제거
  - 새로운 노드를 heap에 추가
  - 특정 노드의 key 값 증가

- 보조 연산
  - Complete binary tree에서 주어진 노드의 두 개 subtrees가 각각 max-heap일 때 주어진 노드를 root로 하는 tree가 heap이 되도록 위치 조정

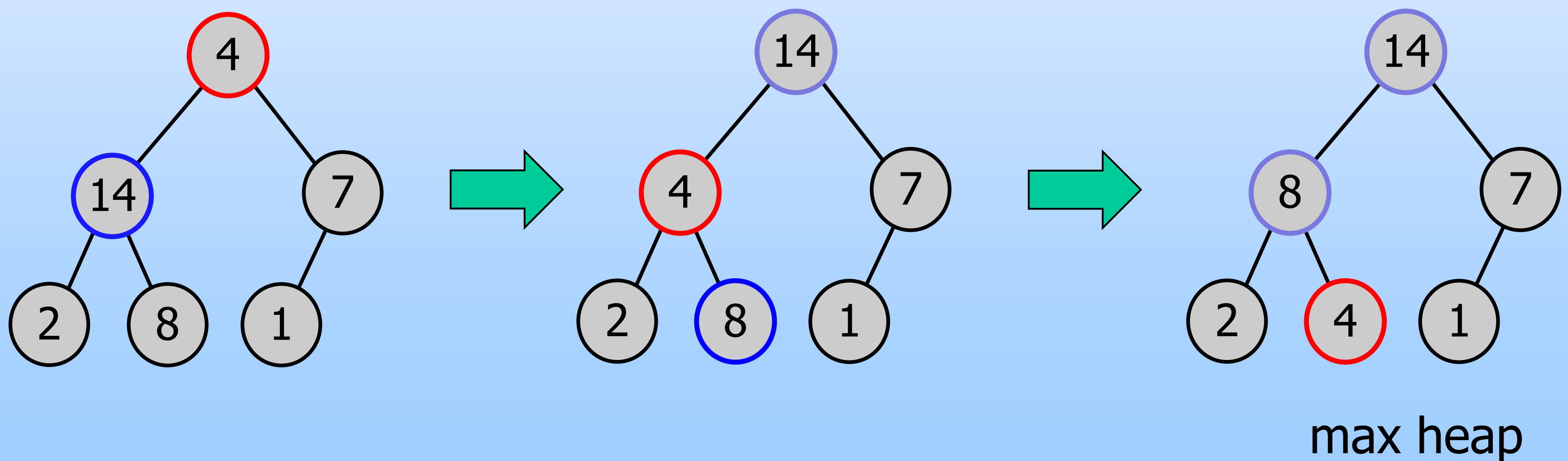
Heapify (1)

- 기능
  - 두 개의 subtree가 max-heap일 때 root를 포함한 전체가 heap되도록 위치 조정
  - Float-down(작은 값이 흘러내려가면서 처리되는) 방식 사용



Heapify (2)

- 방법
  - Root가 child보다 값이 작으면
    - max heap 조건에 위배되므로
    - 두 개의 child 중 값이 큰 노드와 root를 교체
  - 교체되었다면 교체된 노드에 대해서 위 작업 반복
    - 교체가 없거나 leaf node일 때까지 반복



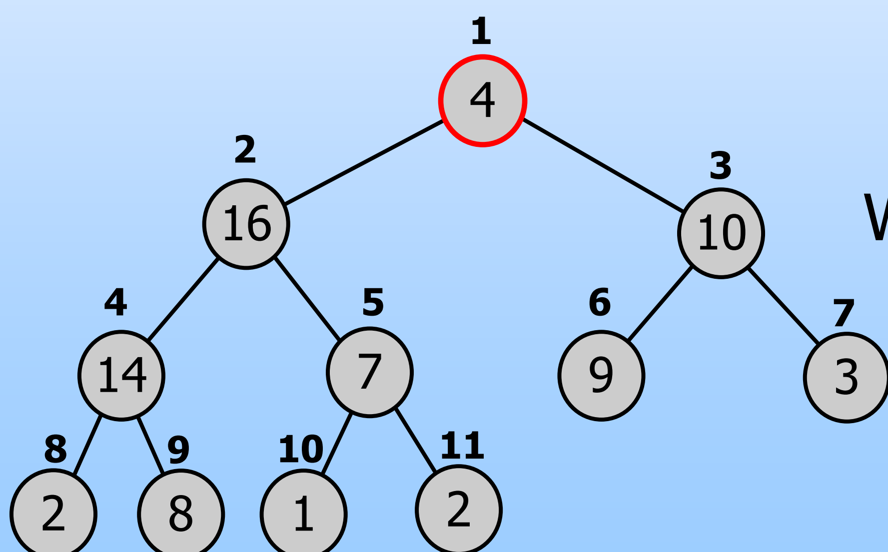
Heapify (3)

- Psuedo code
  - 배열로 표현된 트리를 입력으로 사용한다.

```
MAX-HEAPIFY(A, i) // i번 노드를 root로 하는 sub-tree를 heap으로 구성
    l = LEFT(i)                 // i번 노드 left child의 위치, 2i
    r = RIGHT(i)                // i번 노드 right child의 위치, 2i + 1
    if l ≤ A.heap_size and A[l] > A[i]
        largest = l             // left child가 node i보다 값이 크다.
    else largest = i
    if r ≤ A.heap_size and A[r] > A[largest]
        largest = r             // right child가 최대
    if largest ≠ i              // node i가 최대가 아닌 경우 아래 실행
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

Heapify (4)

- Time Complexity
  - $T(n) = T(2n/3) + \Theta(1)$
    - 자신과 두 개의 child 중 제일 큰 노드 찾기 : $\Theta(1)$
    - Subtree의 최대 노드 개수: $2n/3$ (subtree가 full binary tree인 경우가 worst case임.)
  - Master Theorem의 case 2: $\Theta(\lg n)$



Worst case

실습 주제 (1)

◆ 연습 문제 6.2-1

6.2-1

Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

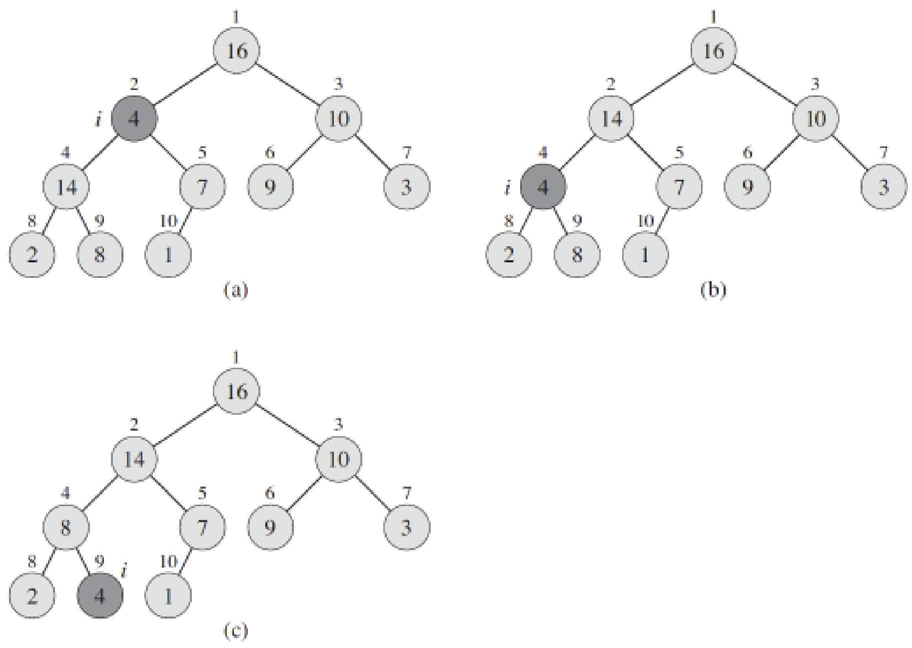


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

실습 주제 (2)

- Max_heapify() 함수 손코딩
  - C-style 구조체 예제

```
typedef struct heap {
    int     size;       // node 개수
    int     capacity; // 배열 최대 크기
    int     *element;
} heap_t;
```

  - C-style 함수 예제
    - void max_heapify(heap_t * heap, int pos);
    - 가정: heap->element[pos] node의 left child subtree와 right child subtree 모두 max heap이라고 가정
    - 출력: heap->element[pos]를 root로 하는 subtree가 max-heap이 되도록 한다.

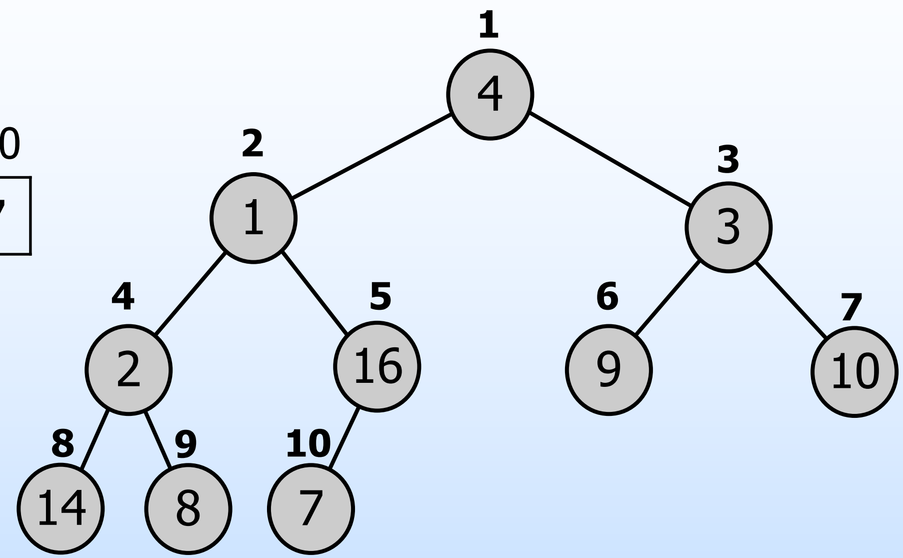
코딩 과제 HW#2.C (1)

- HW#2.C1 (실습 주제 2에 해당)
  - max_heapify() 함수 구현
    - 노드의 subtrees는 모두 max heap라고 가정
  - test_heapify() 함수 구현
    - 아래의 함수들을 활용해서 max_heapify() 함수 동작 여부 검증하는 함수
  - makeSampleHeap(int n)
    - 테스트 용도의 max heap 생성 함수
  - isMaxHeap(heap_t *heap)
    - 주어진 heap이 max heap인지 여부를 판단하는 함수
  - printHeap() 함수 구현
    - heap의 내용을 출력하는 함수

Max Heap 구성 (1)

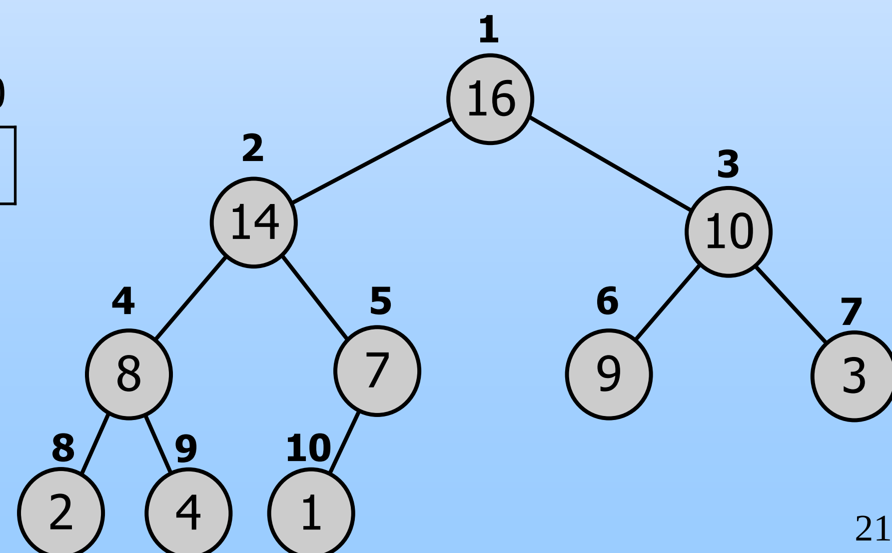
◆ 입력: 배열 A[1..n]

0	1	2	3	4	5	6	7	8	9	10
/	4	1	3	2	16	9	10	14	8	7



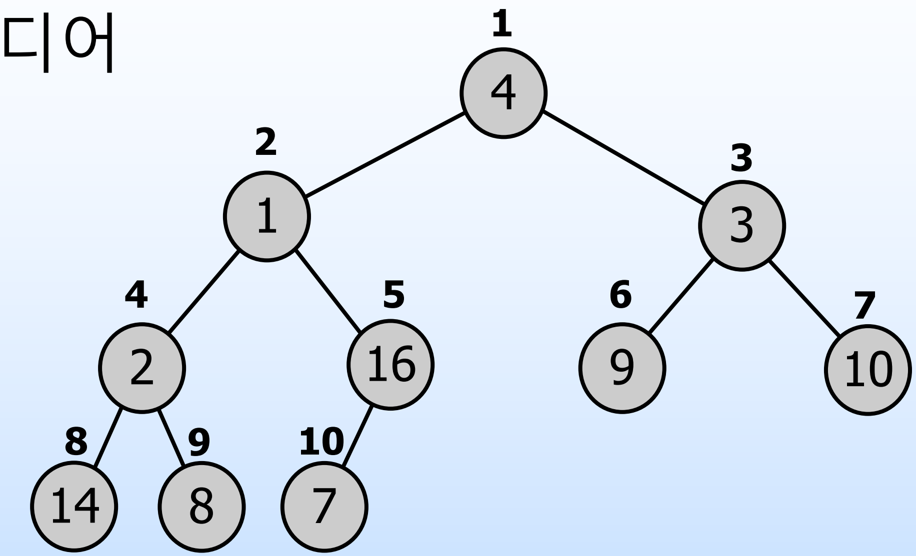
◆ 출력: Max-heap 배열 A[1..n]

0	1	2	3	4	5	6	7	8	9	10
/	16	14	10	8	7	9	3	2	4	1



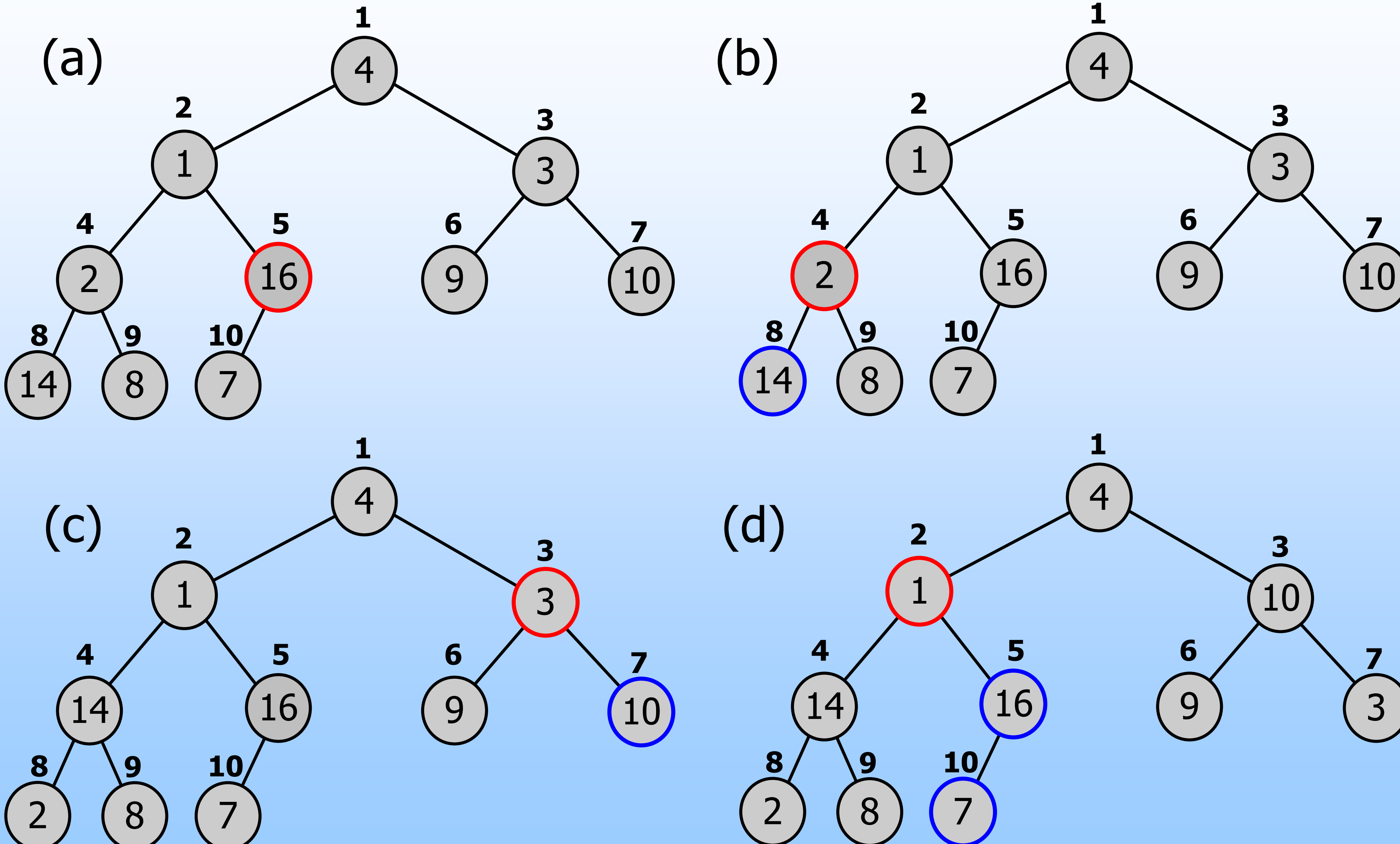
Max Heap 구성 (2)

- 아이디어

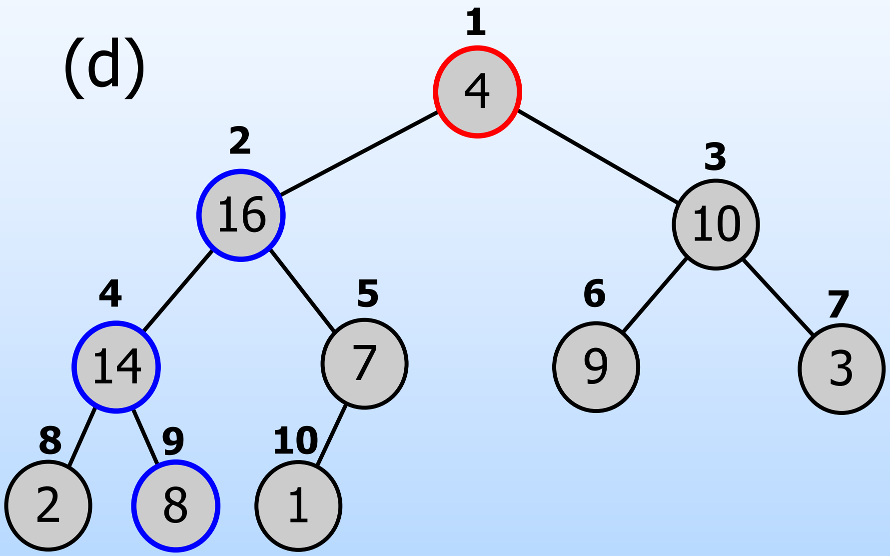


- Buttom-up 방법으로 heapify 적용
  - Level이 높은 노드부터 처리
  - 즉, 배열의 뒤에서부터 차례대로 처리
- Leaf node는 이미 max-heap이므로 5번 노드부터 진행
  - 첫 번째 처리할 노드의 위치: $\lfloor n/2 \rfloor$
- Heapify하는 노드 순서: 5 → 4 → 3 → 2 → 1

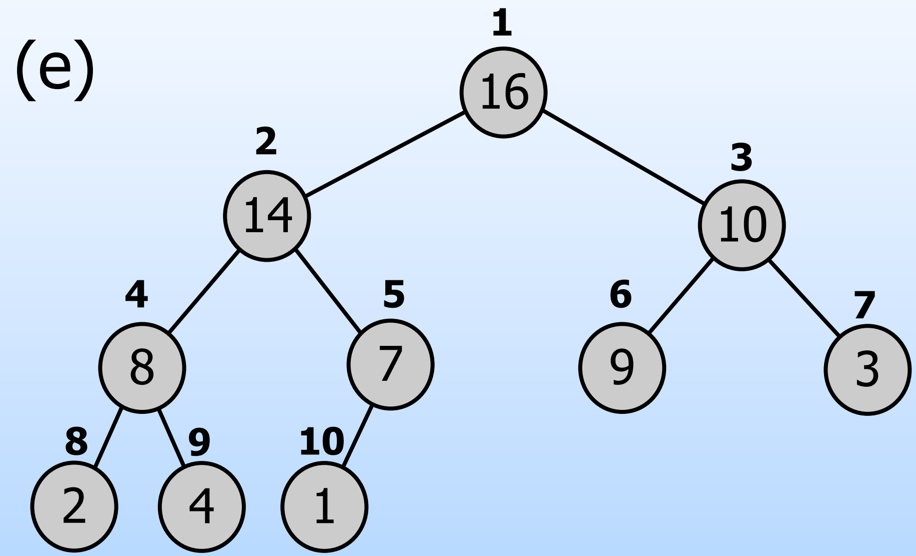
Max Heap 구성 (3)



Max Heap 구성 (4)



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



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

Max Heap 구성 (5)

- Pseudo Code

BUILD-MAX-HEAP (A) // 배열 A를 heap이 되도록 재구성

$A.heap\_size = A.length$

for $i = \lfloor A.length/2 \rfloor$ down to 1

MAX-HEAPIFY(A, i)

Max Heap 구성 (6)

- Loop Invariant
  - 2, 3번째 line 시작하기 전에 노드 i+1, i+2, ..., n 각각은 max-heap의 root이다.

BUILD-MAX-HEAP (A)

1 $A.heap\_size = A.length$

2 for $i = \lfloor A.length/2 \rfloor$ downto 1

3 MAX-HEAPIFY(A, i)

Max Heap 구성 (7)

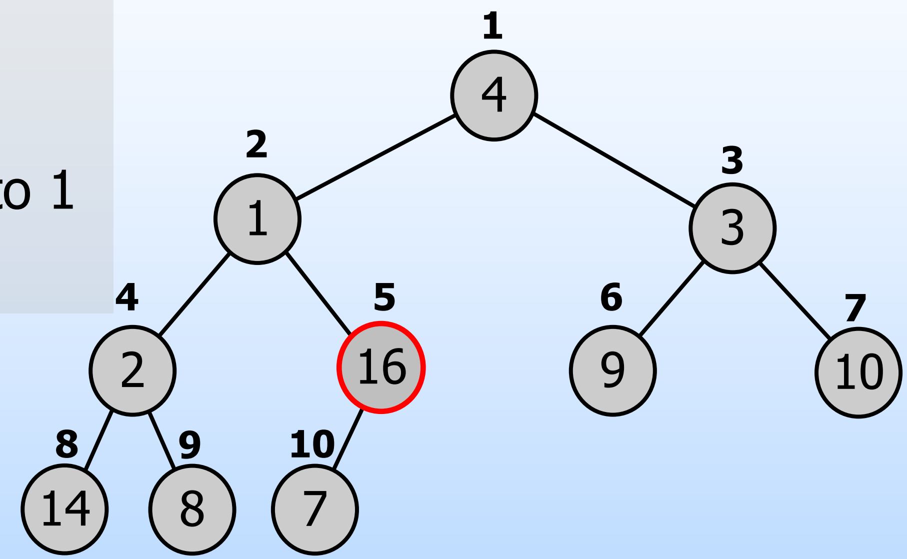
- Loop Invariant 증명: Initialization

BUILD-MAX-HEAP (A)

$A.heap\_size = A.length$

for $i = \lfloor A.length/2 \rfloor$ downto 1

MAX-HEAPIFY(A, i)

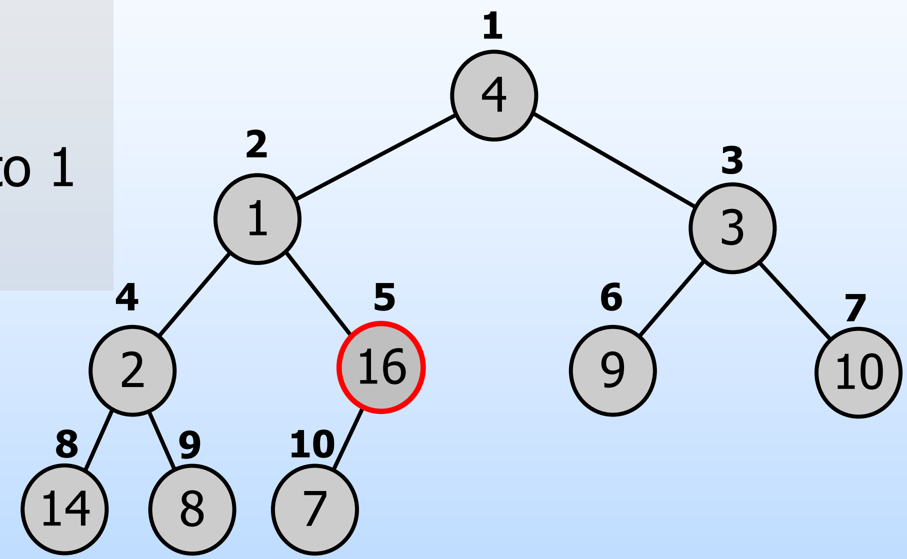


- 첫 번째로 loop 수행하기 전에 $i = \lfloor n/2 \rfloor$이다.
- $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, …, n 노드들은 leaf node이므로 각각 max-heap이다.
- 따라서, loop invariant는 초기 상태에 TRUE이다.

Max Heap 구성 (8)

- Loop Invariant 증명: Maintenance

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

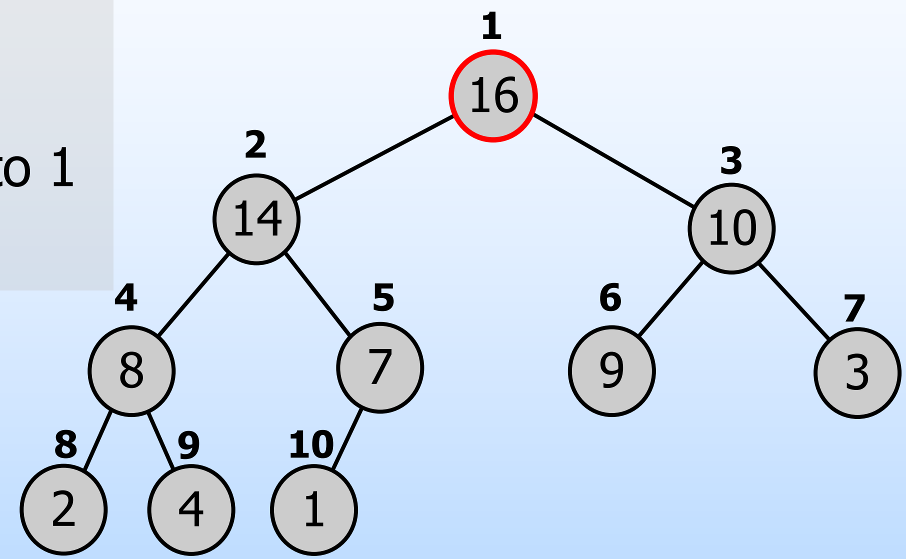


- 노드 번호 i는 감소되는 순서로 처리된다.
- 따라서, 노드 i를 수행하기 전에 번호가 큰 노드는 모두 max-heap의 root이다.
- 이 조건은 MAX-HEAPIFY()의 입력 조건에 해당되므로 MAX-HEAPIFY() 수행 완료 후에 노드 i는 max heap의 root가 된다.
- 따라서, i가 1 감소한 다음 번 loop 수행 시작 시점에 loop invariant는 TRUE이다.

Max Heap 구성 (9)

- Loop Invariant 증명: Termination

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



- 종료 시점에 i는 0이 된다.
- Loop invariant에 의해서 노드 1, 2, ..., n는 각각 max-heap의 root이다.
- 특히, node 1은 전체 노드들로 구성된 max-heap의 root이다.
- 따라서, BUILD-MAX-HEAP은 max-heap을 생성한다.

실습 주제 (3)

- Build-Max-Heap 함수 손코딩

Max Heap 구성 (10)

- Time Complexity: Simple Upper Bound

BUILD-MAX-HEAP (A)

$A.heap\_size = A.length$

for $i = \lfloor A.length/2 \rfloor$ downto 1

MAX-HEAPIFY(A, i)

- MAX-HEAPIFY 수행 시간: $O(\lg n)$
- MAX-HEAPIFY 호출 횟수: $n/2$
- 따라서, $O(n \lg n)$
- 그러나, Tight upper bound가 아님

Max Heap 구성 (11)

- Time Complexity: Tight Upper Bound
  - n개의 원소로 구성된 heap의 height = $\lfloor \lg n \rfloor$
  - 자신을 root로 하는 subtree의 height가 h인 노드 개수
    - $\lceil n/2^{h+1} \rceil$, h는 height (HW#2.P5: 연습 문제 6.3-3)
  - MAX-HEAPIFY 수행 시간: $O(h)$
  - 전체 수행 시간: $O(n)$ ← why?
    - 계산 방법: height x (height 별 노드 개수)의 총합

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

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

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

$$= O(n)$$

이 수식 풀이는
다음 슬라이드에...

Max Heap 구성 (12)

- 수식 풀이

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

  - 아래의 식 1에서 시작
  - 식 1의 양변을 x에 대해서 미분해서 식 2를 구하고
  - 양변에 x 를 곱하면 식 3을 구할 수 있다.
  - 식 3의 x에 1/2를 대입하면 $\sum_{h=0}^{\infty}\frac{h}{2^h}=2$가 됨

$$\sum_{h=0}^{\infty}x^h=\frac{1}{1-x}$$

(식 1)

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

(식 2)

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

(식 3)

  - 참고 자료
    - 알고리즘 교재 1,148페이지 수식 A.8
    - 알고리즘 교재 1,147페이지 수식 A.6

HW#2.P (1)

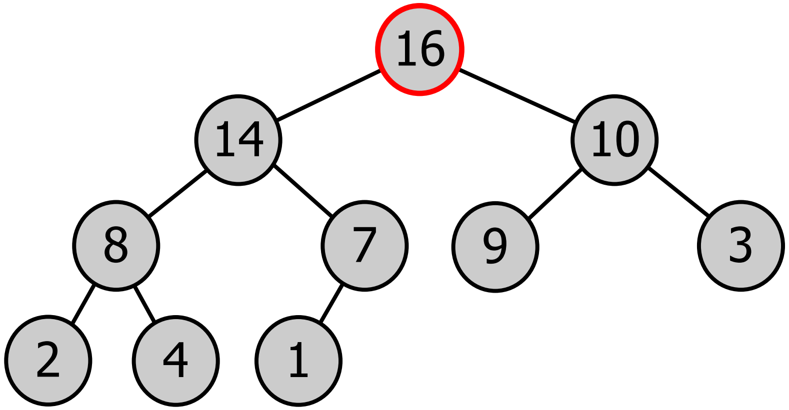
- HW#2.P1
  - 알고리즘 교재 연습 문제 6.1-1
- HW#2.P2
  - 알고리즘 교재 연습 문제 6.1-2
- HW#2.P3
  - 알고리즘 교재 연습 문제 6.2-1
- HW#2.P4
  - 알고리즘 교재 연습 문제 6.2-6
- HW#2.P5
  - 알고리즘 교재 연습 문제 6.3-3

주의: 3rd Edition 문제 번호임
2nd edition은 다를 수 있음

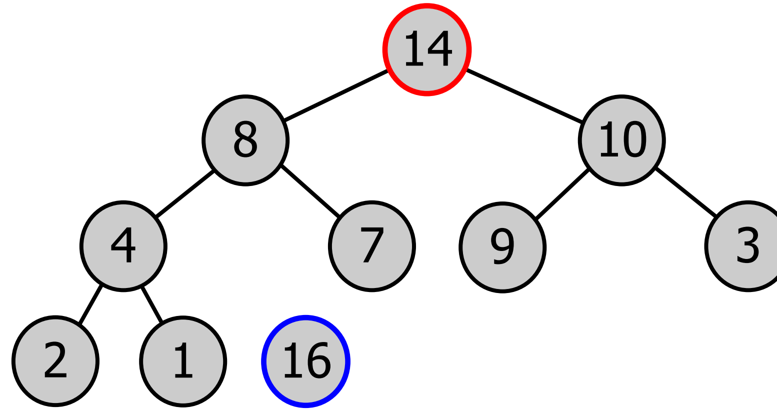
Heap Sort 알고리즘

- 배열이 주어지면 max-heap을 구성한다.
- Max-heap에서 최대값을 제거하고 heap 재구성한다.
  - Root를 배열의 마지막 원소와 교체하고
  - Heap을 구성하는 원소 개수를 1 줄인 상태에서
  - Root를 흘러내리는 방식으로 heapify한다.
  - 원소 개수가 1 줄었으므로 마지막 원소는 영향을 받지 않는다.
- 제거하는 작업을 heap을 구성하는 원소 개수가 0이 될 때까지 반복한다.

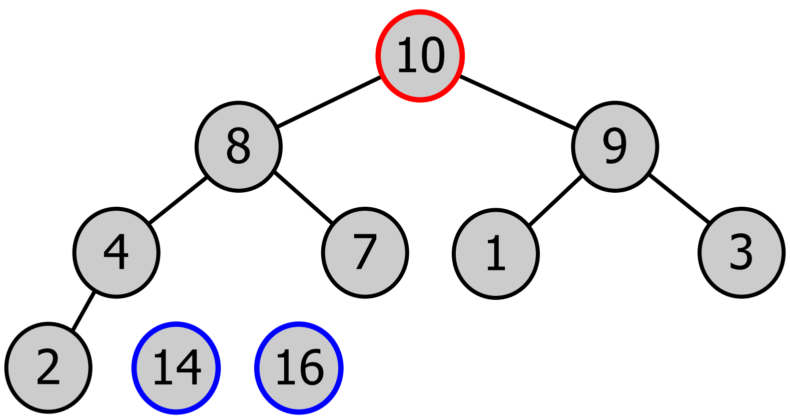
최대값 제거 및 Heap 재구성 과정 (1)



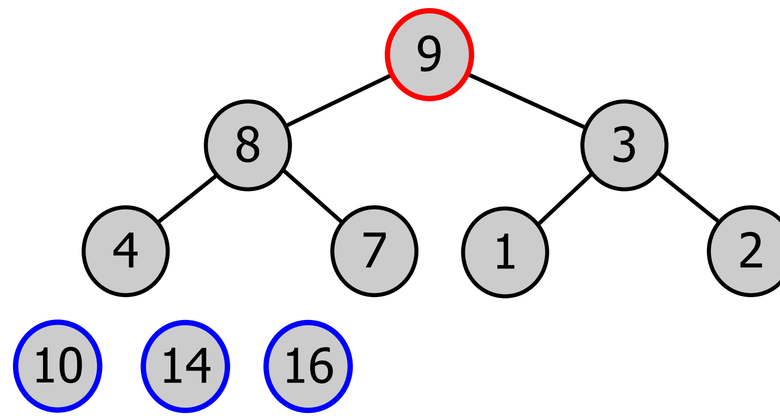
/	16	14	10	8	7	9	3	2	4	1



/	14	8	10	4	7	9	3	2	1	**16**

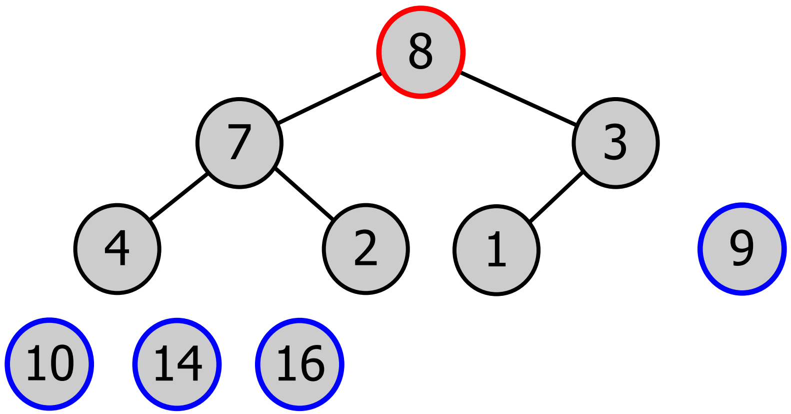


/	10	8	9	4	7	1	3	2	**14**	**16**

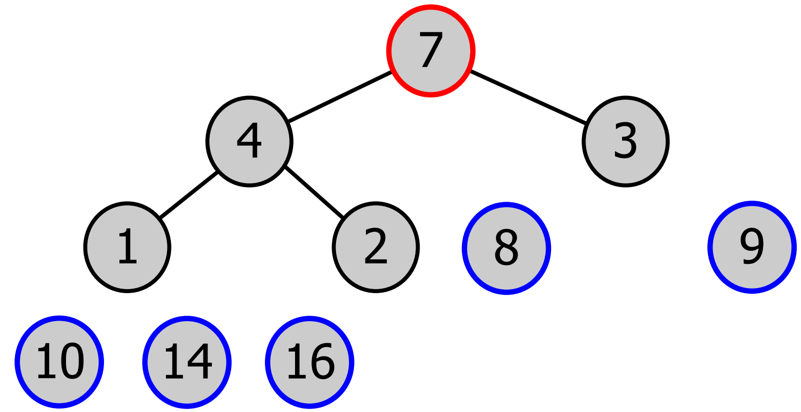


/	9	8	3	4	7	1	2	**10**	**14**	**16**

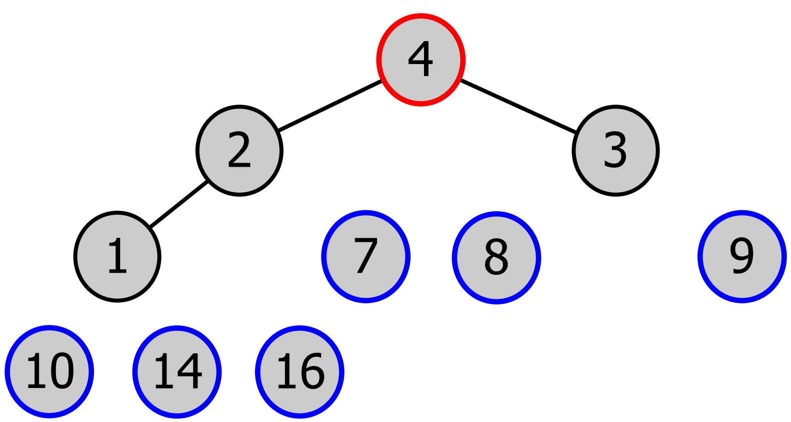
최대값 제거 및 Heap 재구성 과정 (2)



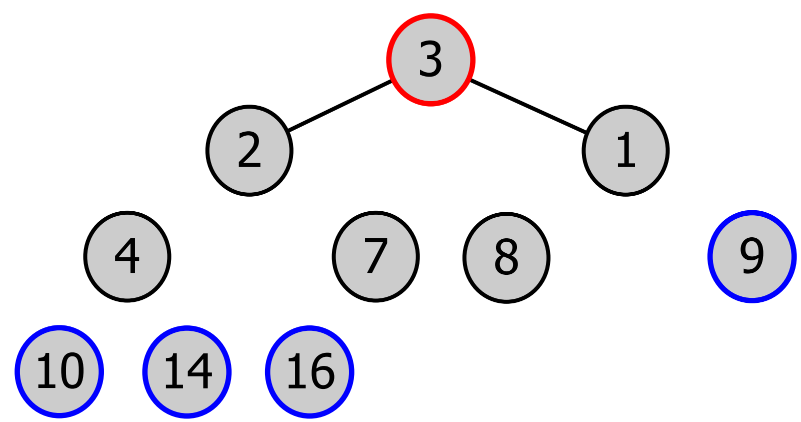
/	8	7	3	4	2	1	**9**	**10**	**14**	**16**



/	7	4	3	1	2	**8**	**9**	**10**	**14**	**16**

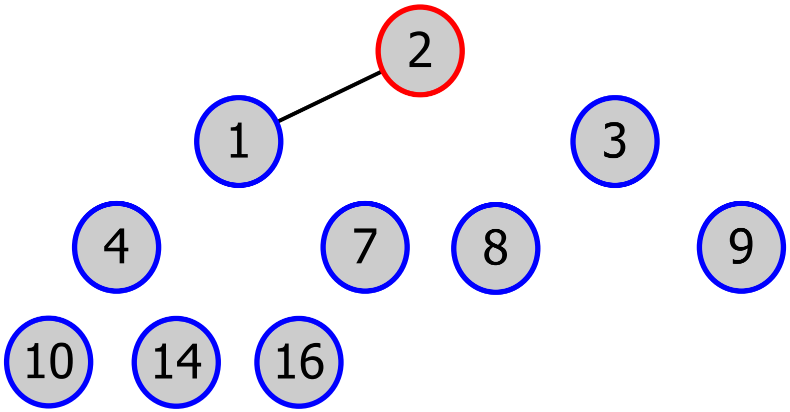


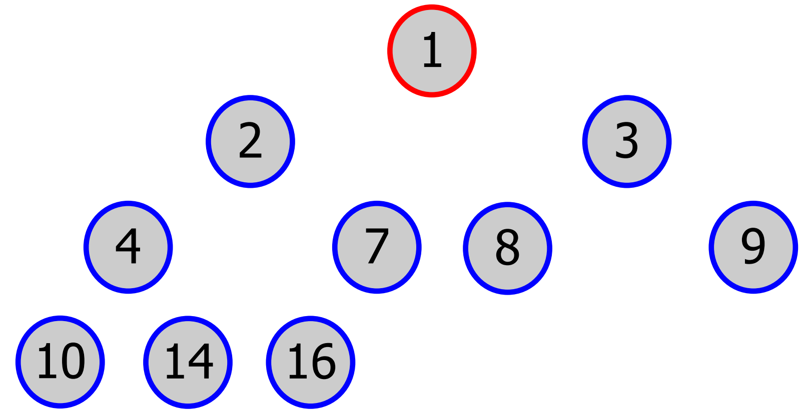
/	4	2	3	1	**7**	**8**	**9**	**10**	**14**	**16**



/	3	2	1	**4**	**7**	**8**	**9**	**10**	**14**	**16**

최대값 제거 및 Heap 재구성 과정 (3)





/	2	1	**3**	**4**	**7**	**8**	**9**	**10**	**14**	**16**

/	1	**2**	**3**	**4**	**7**	**8**	**9**	**10**	**14**	**16**

Heap Sort Pseudo Code

◆ Pseudo Code

```
HEAPSORT (A) // 배열의 1번부터 n번까지 원소가 있는 배열
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]   // root와 맨 마지막 노드를 교체
        A.heap-size = A.heap-size – 1
        MAX-HEAPIFY(A, 1)         // root node를 heapify
```

Heap Sort 시간 복잡도

- 구성 요소
  - Max-heap 생성: $O(n)$
  - MAX-HEAPIFY 호출 횟수: $O(n)$
  - MAX-HEAPIFY 시간 복잡도: $O(\lg n)$
- 총 소요 시간
  - $O(n \lg n)$

```
HEAPSORT (A) // 배열의 1번부터 n번까지 원소가 있는 배열
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]   // root와 맨 마지막 노드를 교체
        A.heap_size = A.heap_size – 1
        MAX-HEAPIFY(A, 1)         // root node를 heapify
```

실습 주제 (4)

- Heap sort 손코딩

Priority Queues (1)

- Queue
  - FIFO (선입선출) 자료구조

- Priority Queue
  - 삽입 순서와 관계없이 우선순위가 높은 것부터 출력(삭제)하는 자료구조
  - 삽입/삭제가 빈번하게 발생해도 우선순위가 높은 원소를 효율적으로 삭제할 수 있는 자료구조가 필요함
  - Heap 자료구조 사용

Priority Queues (2)

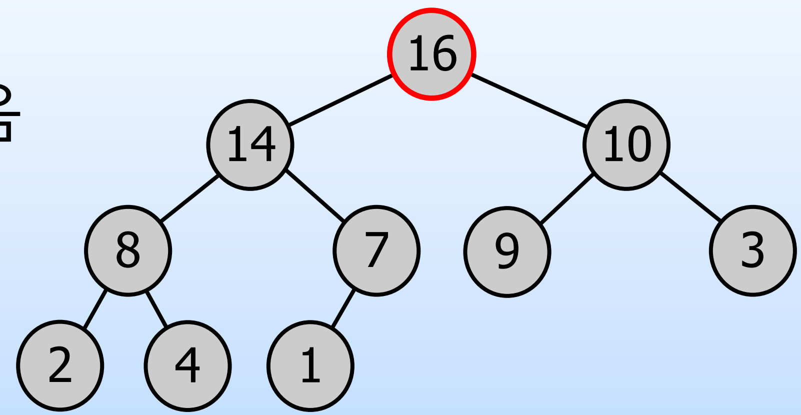
- Max-priority queue
  - 제일 큰 값부터 삭제
  - Job-scheduling
    - OS에서 우선순위가 높은 job부터 처리
    - 처리 중인 작업이 마무리되면 남아 있는 작업 중 우선 순위가 제일 높은 작업 진행
  - Max-heap 사용
- Min-priority queue
  - 제일 작은 값부터 삭제
  - Event-driven simulator
    - 시간순으로 향후 발생할 event들을 queue에 저장
    - 제일 먼저 발생할 event들부터 처리
    - Event 처리 후 향후 발생할 새로운 event들을 queue에 추가
  - Min-heap 사용

Max-Priority Queue 연산 종류

- INSERT(S, x)
  - Set S에 element x를 추가
- MAXIMUM(S)
  - S에서 값이 제일 큰 element return
- EXTRACT-MAX(S)
  - S에서 값이 제일 큰 element를 삭제하고 return
- INCREASE(S, x, k)
  - Element x에 저장된 값을 k로 증가 (k는 x의 현재값보다 크거나 같다고 가정)

Max-Heap에서 최대값 읽기

- 방법
  - Max-heap의 root에 최대값이 저장되어 있음
  - Root node의 값을 읽음
  - 배열 구조에서 A[1]을 읽음
- Time complexity
  - $\Theta(1)$

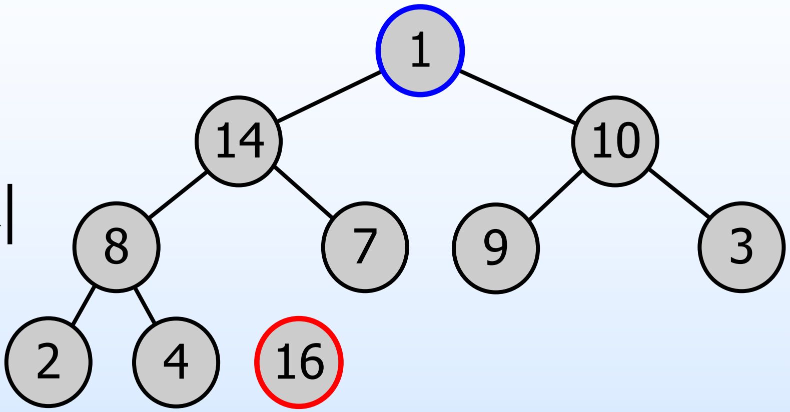


/	16	14	10	8	7	9	3	2	4	1

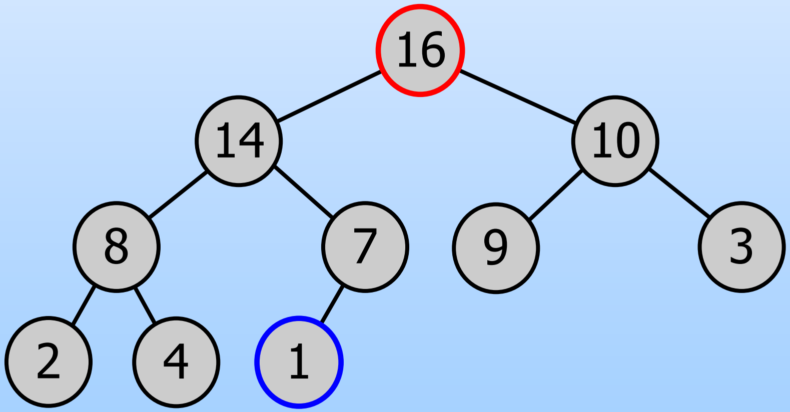
```
HEAP-MAXIMUM (A)
1      return A[1]
```

Max-Heap에서 최대값 제거 (1)

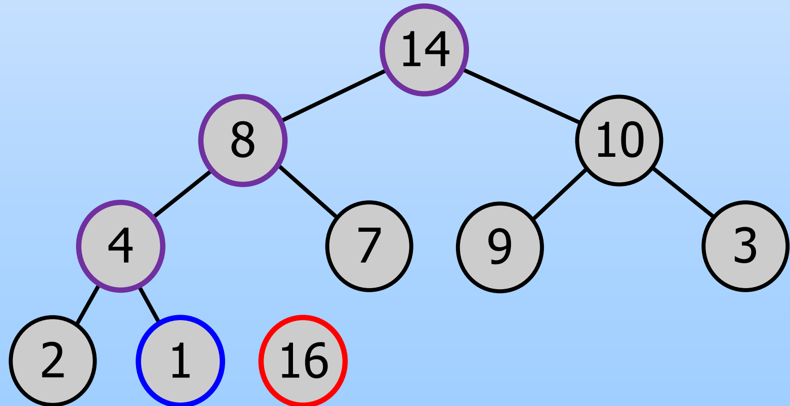
- 방법
  - Root node를 제거
  - 마지막 노드를 root에 배치
  - 흘러 내려가는 방식으로 위치 보정 (Heapify)



/	1	14	10	8	7	9	3	2	4	16



/	16	14	10	8	7	9	3	2	4	1



/	14	8	10	4	7	9	3	2	1	16

Max-Heap에서 최대값 제거 (2)

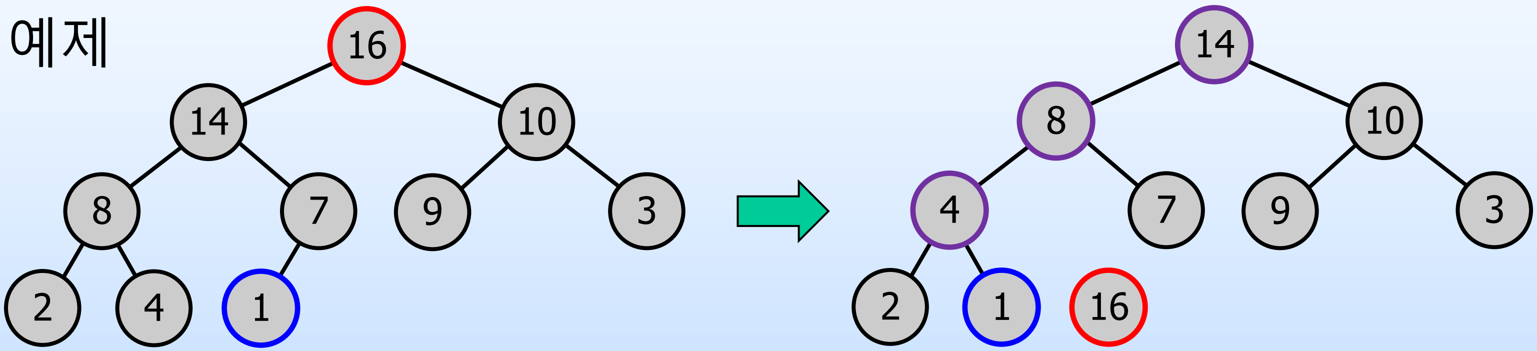
- 배열에서의 처리 방법
  - 배열의 마지막 원소와 A[1]와 swap
  - 흘러 내려가는 방식으로 A[1]을 위치 조정 (heapify)

- Time complexity
  - $\Theta(\lg n)$

```
HEAP-EXTRACT-MAXIMUM (A)
if A.heap_size < 1
        error "heap underflow"
max = A[1]
A[1] = A[A.heap_size]
A.heap_size = A.heap_size – 1
MAX-HEAPIFY(A, 1)
return max
```

실습 주제 (5)

- Max heap에서 최대값 제거 함수 손코딩
  - 입력: max heap을 저장한 배열
  - 출력: 최대값이 제거된 max heap과 최대값
  - 예제



/	16	14	10	8	7	9	3	2	4	1

/	14	8	10	4	7	9	3	2	1	16

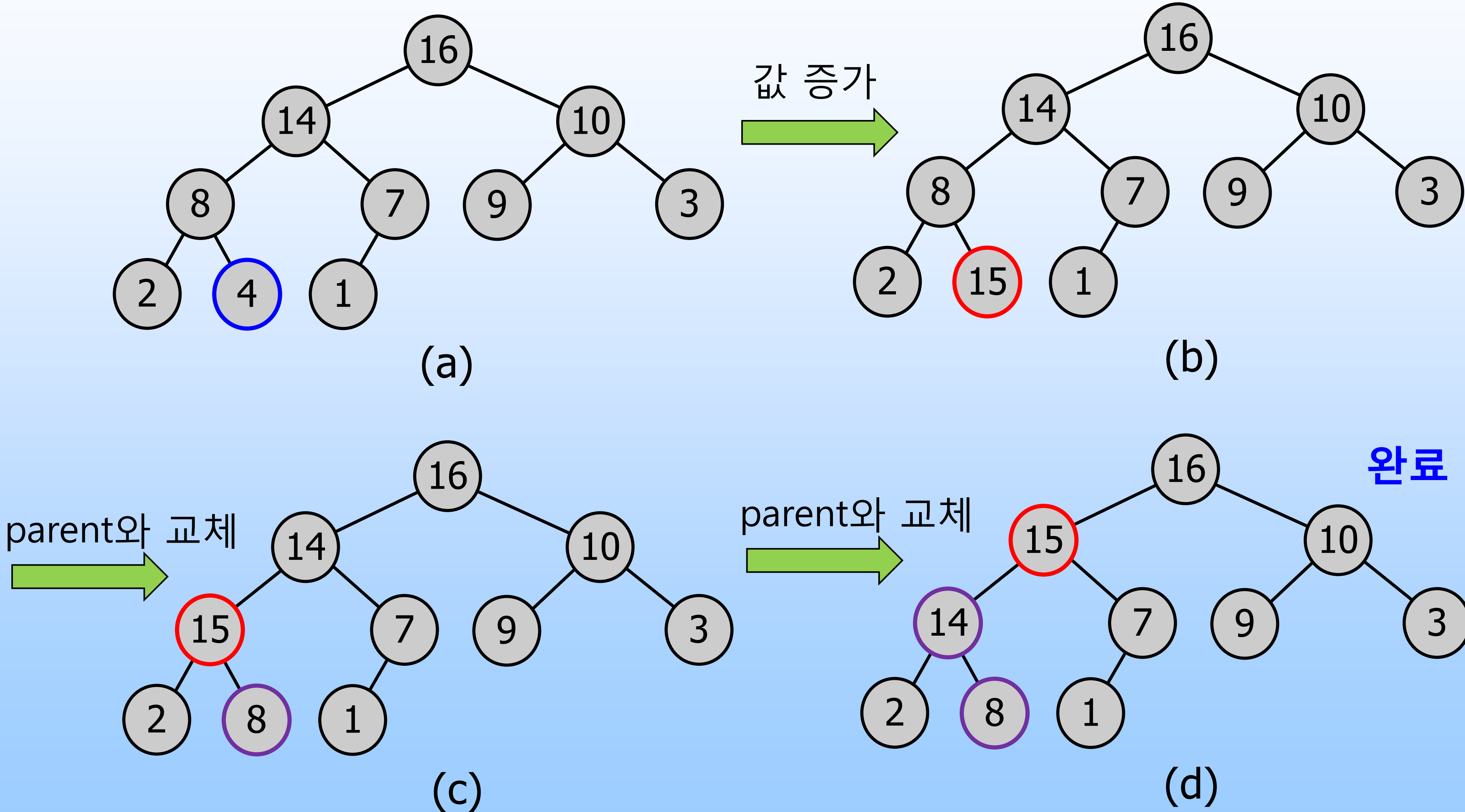
  - C-style API
    - int extractMax(heap_t *heap, int *maxValue);
    - void maxHeapify(heap_t *heap, int nodeId); 는 주어진다고 가정

```
typedef struct heap {
    int     size;          // node 개수
    int     capacity; // 배열 최대 크기
    int     *element;
} heap_t;
```

Max-Heap에서 원소의 값 증가 (1)

- 목적
  - 원소의 값 변경 시 삭제 후 추가하지 않고 값을 증가시키는 방법을 사용하여 효율적으로 처리
- 아이디어
  - 값이 증가해서 parent보다 값이 더 크게 되면 max-heap 조건에 위배됨
  - Max-heap 조건에 위배되지 않도록 위치 조정
- 방법
  - Root 방향으로 거슬러 올라가는 방식으로 위치 조정
    - Parent와 값을 비교해서 더 크면 parent와 교체
    - 이 작업을 parent보다 값이 작거나 parent가 없을 때까지 즉 root에 도달할 때까지 반복 진행

Max-Heap에서 값 증가 (2)



Max-Heap에서 값 증가 (3)

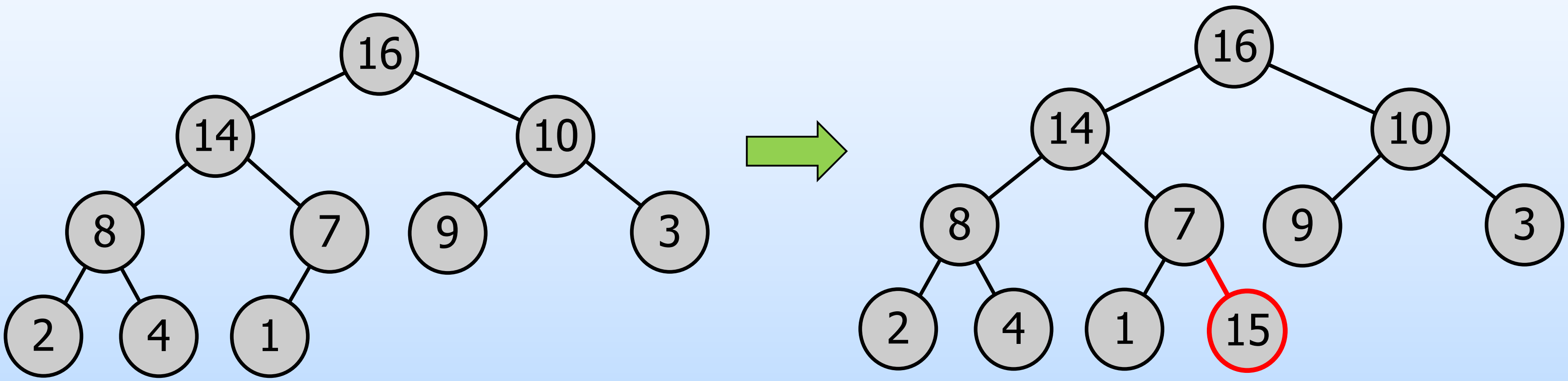
- Pseudo code

```
HEAP-INCREASE-KEY (A, i, key)
if key < A[i]
        error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

- Time complexity
  - $O(\lg n)$

Max-Heap Insertion (1)

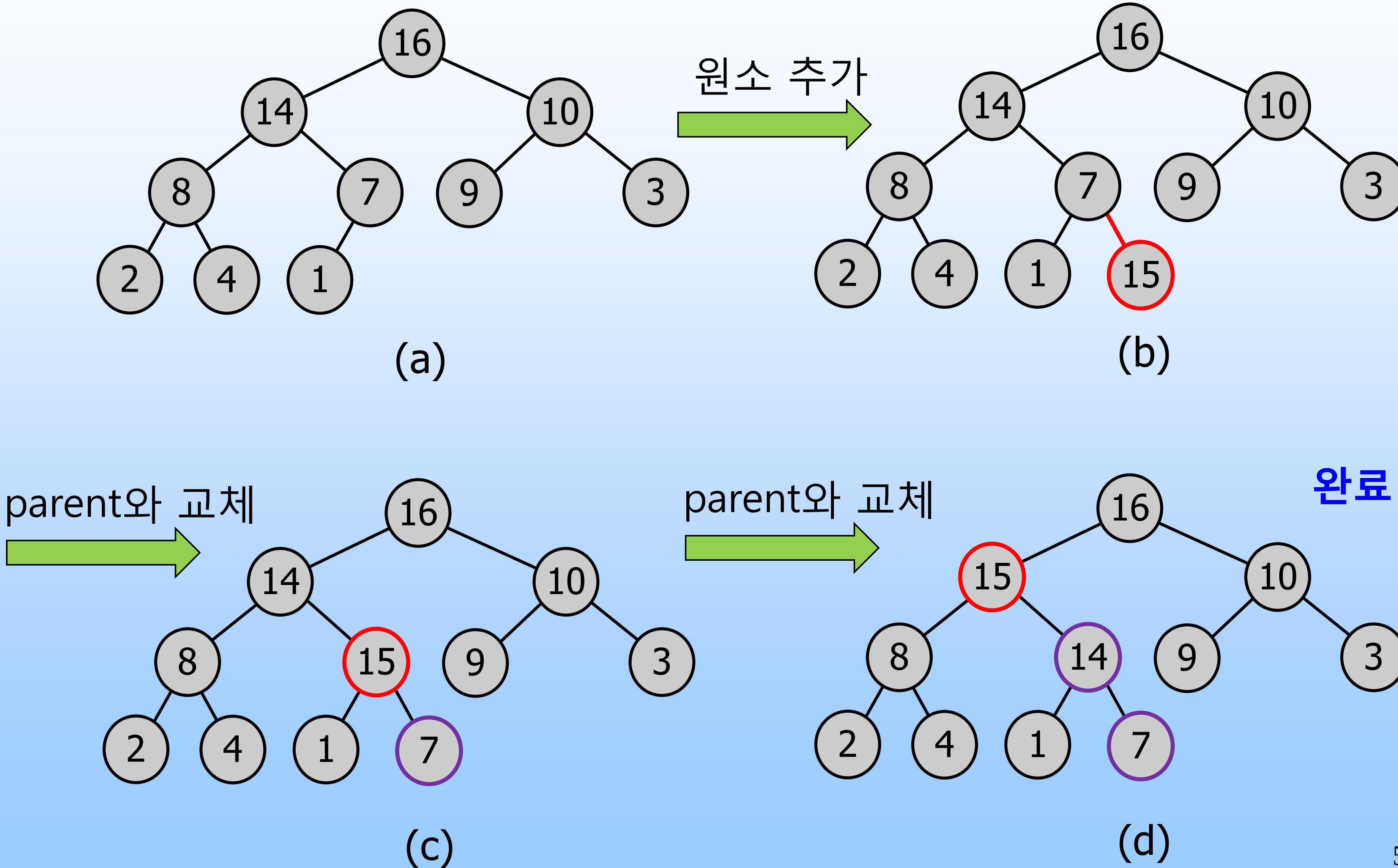
- 방법
  - Complete binary tree의 맨 마지막에 노드 추가



(a) (b)

  - Max-heap 조건에 위배되지 않도록 Root 방향으로 거슬러 올라가는 방식으로 위치 조정
    - Parent와 비교해서 값이 더 크면 parent와 교체
    - 이 작업을 parent보다 값이 작거나 parent가 없을 때까지 즉 root에 도달할 때까지 반복 진행

Max-Heap에 원소 추가 (2)



Max-Heap Insertion (3)

Pseudo Code

```
MAX-HEAP-INSERT (A, key)
    A.heap_size = A.heap_size + 1
    A[A.heap_size] = −∞
    HEAP-INCREASE-KEY(A, A.heap_size, key)
```

- 마지막에 $-\infty$값을 갖는 노드 삽입 후 값 증가시키는 방법 사용
- 실재 code에서 $-\infty$를 사용할 수 없는 경우 HEAP-INCREASE-KEY 함수를 변형해서 구현해야 함.

Time Complexity

- $O(\lg n)$
  - 왜냐하면 HEAP-INCREASE-KEY 함수의 time complexity가 $O(\lg n)$이기 때문이다.

실습 주제 (6)

- Max heap에 새로운 값 추가 손코딩

코딩 과제 HW#2.C (2)

- HW#2.C2
  - 실습 주제 3, 4, 5, 6 구현

HW#2.P (2)

- HW#2.P6
  - 알고리즘 교재 연습 문제 6.4-2
- HW#2.P7
  - 알고리즘 교재 연습 문제 6.5-2
- HW#2.P8
  - 알고리즘 교재 연습 문제 6.5-7
- HW#2.P9
  - 알고리즘 교재 연습 문제 6.5-9
- HW#2.P10
  - 알고리즘 교재 Chapter 6 Problems 6-1
- HW#2.P11
  - 알고리즘 교재 Chapter 6 Problems 6-3

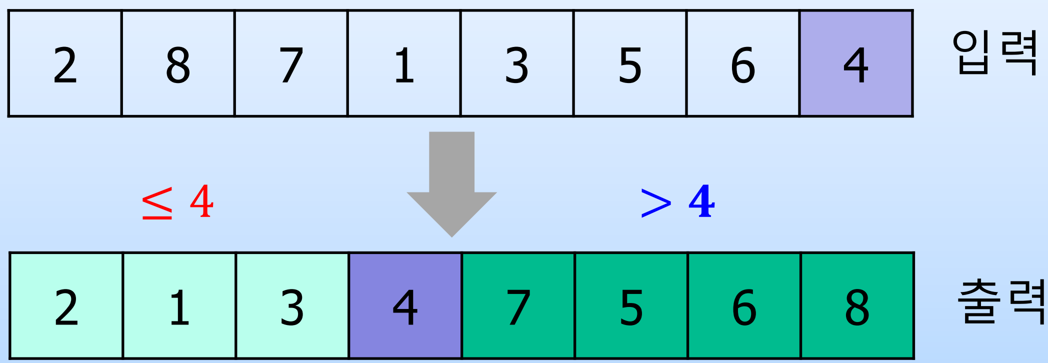
주의: 3rd Edition 문제 번호임
2nd edition은 다를 수 있음

목 차

- Heapsort (Chapter 6)
- Quicksort (Chapter 7)
- Sorting in Linear Time (Chapter 8)
- Medians and Order Statistics (Chapter 9)

배열 Partition Problem

- 입력
  - 정수 배열
- 출력
  - 입력의 맨 마지막 원소를 기준으로 작은 원소들과 큰 원소들을 분리된 배열과 기준이 되는 원소의 위치



(입력의 맨 마지막 원소 기준으로 왼쪽은 작은 숫자들, 오른쪽은 큰 숫자들이 오도록 배치)

- 손코딩 (C-style API)
  - int partition(int *arr, int size);
  - return 값은 입력에서 맨 마지막 원소의 출력 배열에서의 위치
    - 위 예제에서는 3이 return 값

Quicksort 특징

- Divide-and-Conquer 기법의 알고리즘
- 가장 실용적인 sort 기법
  - Sorting 기법 중 가장 빠르다고 해서 quick이라는 이름이 들어감
- Time Complexity
  - Worst case: $\Theta(n^2)$
  - Expected running time: $\Theta(n \lg n)$
  - Time complexity에 들어가는 constant factor가 작아서 $\Theta(n^2)$인데도 속도가 빠름
- In-place 알고리즘
  - $O(1)$의 추가 메모리 사용

Quicksort 아이디어

- 입력 배열을 기준 (pivot) 값을 중심으로 두 개로 분할(partition)
  - 왼쪽은 기준(pivot) 값보다 작거나 같음
  - 오른쪽은 기준(pivot) 값보다 큼

2	8	7	1	3	5	6	4

≤ 4 $\qquad$ $4 \leq$

2	1	3	4	7	5	6	8

  - Pivot을 정하고 분할하는 방법은 뒤에서 설명
- 분할(partition)된 각각을 Quicksort로 정렬
- 분할(partition)된 각각이 정렬 완료되면 전체 정렬 완료

Quicksort 알고리즘

- Divide
  - 입력 배열 A[p..r]을 A[p..q-1]과 A[q+1..r]로 분할 (partition)
  - 이 때 아래 두 가지의 조건을 만족하도록 함
    - A[p..q-1]의 모든 값은 A[q]보다 작거나 같음
    - A[q+1..r]의 모든 값은 A[q]보다 큼
- Conquer
  - A[p..q-1]과 A[q+1..r] 각각을 quicksort
- Combine
  - 각각의 sub array는 이미 정렬되어 있으므로 별도 작업 없음

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)   // 다음 슬라이드에서 설명
    QUICKSORT(A, p, q – 1)
    QUICKSORT(A, q + 1, r)
```

Quicksort의 Issue

- Partition할 때 기준(pivot)을 어떻게 정할 것인가?
  - 일정 위치에 있는 원소 사용
    - 배열의 맨 뒤에 있는 원소
    - 배열의 맨 앞에 있는 원소
  - 배열에 있는 원소 중 임의로 선택된 원소

배열 Partition (1)

- 배열의 맨 마지막 원소를 pivot으로 사용하는 방법 (1)
  - 목표

2	8	7	1	3	5	6	4	입력

≤ 4 $\quad$ $> \mathbf{4}$

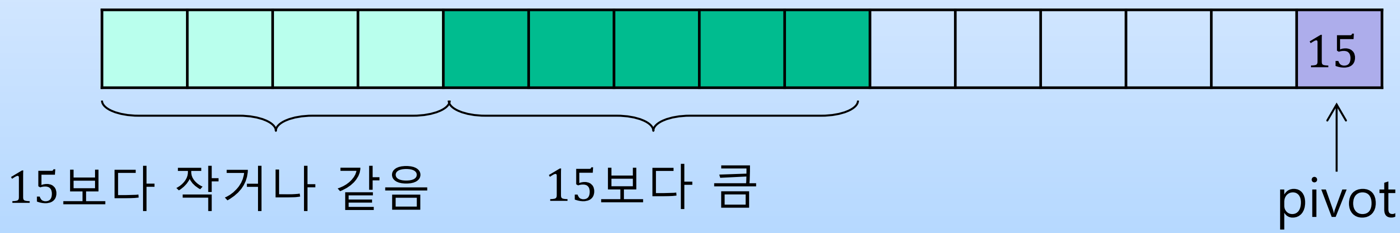
2	1	3	8	7	5	6	4	맨 마지막 원소를 pivot으로 사용해서 두 개로 분할

≤ 4 $\quad$ $> \mathbf{4}$

2	1	3	4	7	5	6	8	Pivot(맨 마지막 원소)이 중간에 오도록 swap

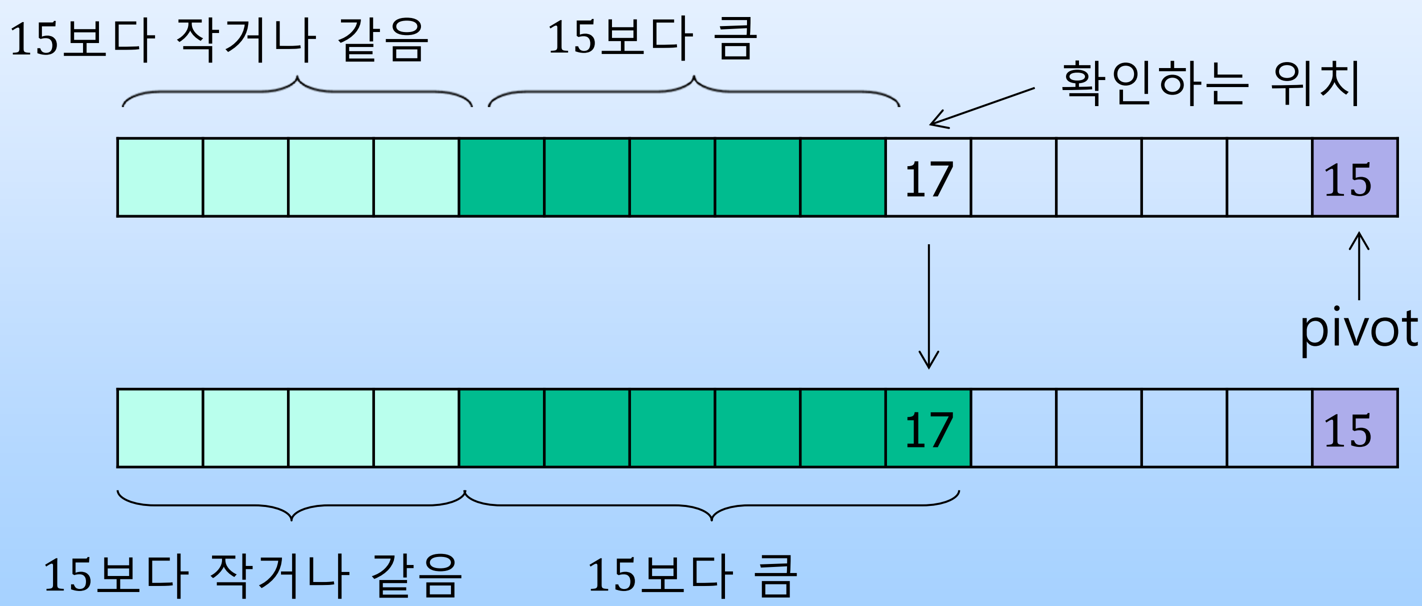
배열 Partition (2)

- 배열의 맨 마지막 원소를 pivot으로 사용하는 방법 (2)
  - 아이디어
    - 배열의 맨 앞부터 차례로 확인하면서
    - Pivot보다 작거나 같은 값을 앞부분에 배치하고
    - Pivot보다 큰 값을 뒷부분에 배치



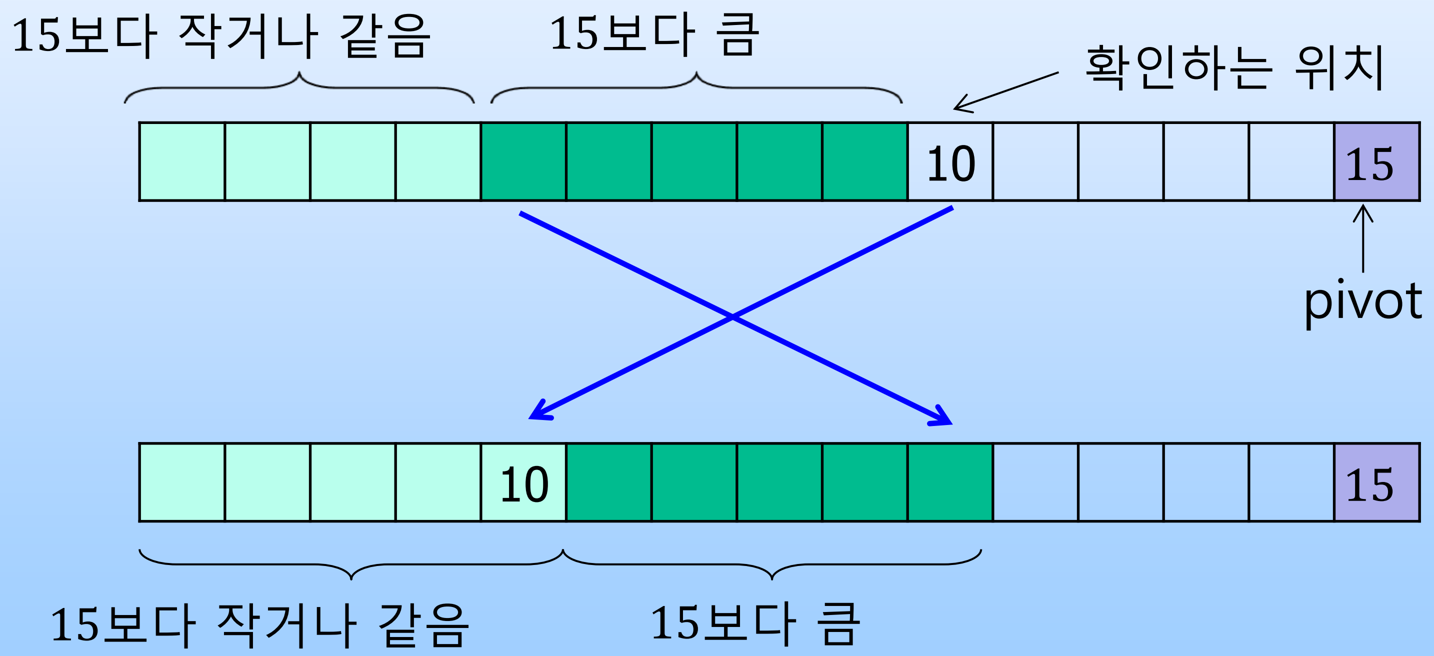
배열 Partition (3)

- 배열의 맨 마지막 원소를 pivot으로 사용하는 방법 (3)
  - 확인한 숫자가 pivot보다 큰 경우
    - 큰 쪽 영역에 추가



배열 Partition (4)

- 배열의 맨 마지막 원소를 pivot으로 사용하는 방법 (4)
  - 확인한 숫자가 pivot보다 작은 경우
    - 작은 영역의 마지막 원소 다음 원소와 교체



배열 Partition (5)

◆ 예제

배열 Partition (6)

PARTITION $(A, start, end)$ // 배열과 처리할 위치의 시작과 끝

$pivotValue = A[end]$ // 끝 원소를 $pivotValue$로 지정

$endOfLowBlock = start - 1$ // $endOfLowBlock$은 empty로 시작

for $posToBeChecked = start$ to $end - 1$ // incremental method 사용

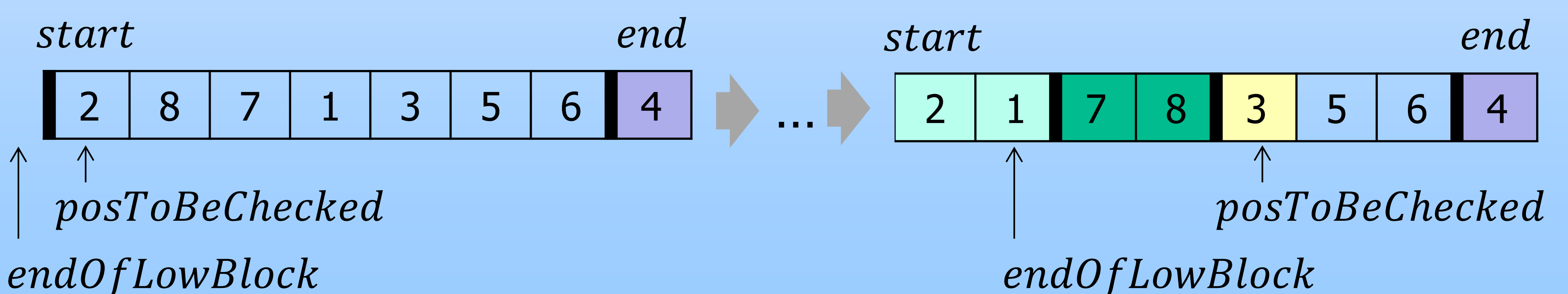
  if $A[posToBeChecked] \leq pivotValue$

    $endOfLowBlock = endOfLowBlock + 1$

    exchange $A[endOfLowBlock]$ with $A[posToBeChecked]$

exchange $A[endOfLowBlock + 1]$ with $A[end]$ // 값이 큰 블록의 첫 번째 원소와 입력 배열의 마지막 원소 교체

return $endOfLowBlock + 1$ // 값이 큰 블록의 첫 번째 원소 위치를 return



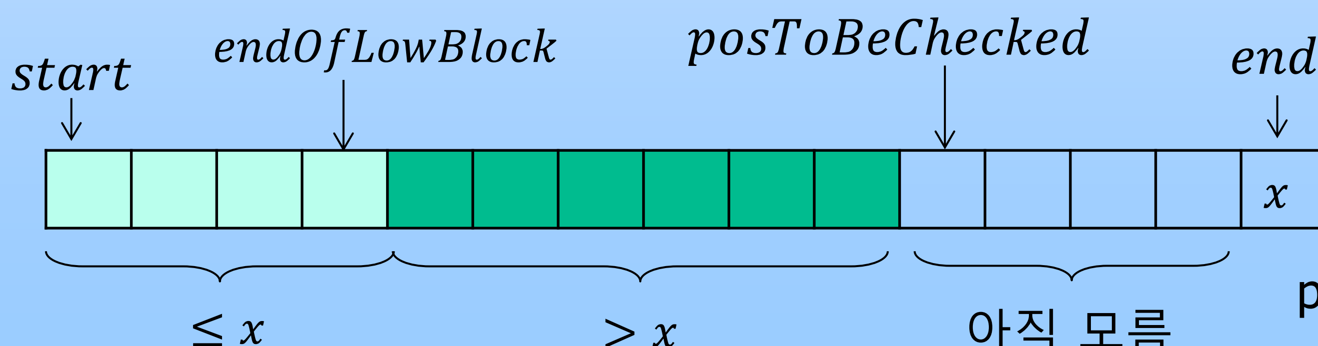
배열 Partition (7)

- Time Complexity
  - 원소들을 한번씩 처리했으므로 $\Theta(n)$

배열 Partition (8)

- Loop Invariant
  - Lines 3-6 시작 시점 각각에 array index k 에 대해서
    1. If $start \le\ k\ \le endOfLowBlock$, then $A[k] \le pivotValue$
    2. If $endOfLowBlock + 1\ \le\ k\ \le\ posToBeChecked\ - 1$, then $A[k] > pivotValue$
    3. If $k = end$, then $A[k] = pivotValue$

3 for $posToBeChecked = start$ to $end - 1$ // incremental method 사용
4 if $A[posToBeChecked] \le pivotValue$
5 $endOfLowBlock = endOfLowBlock + 1$
6 exchange $A[endOfLowBlock]$ with $A[posToBeChecked]$



pivot value $= x$

배열 Partition (9)

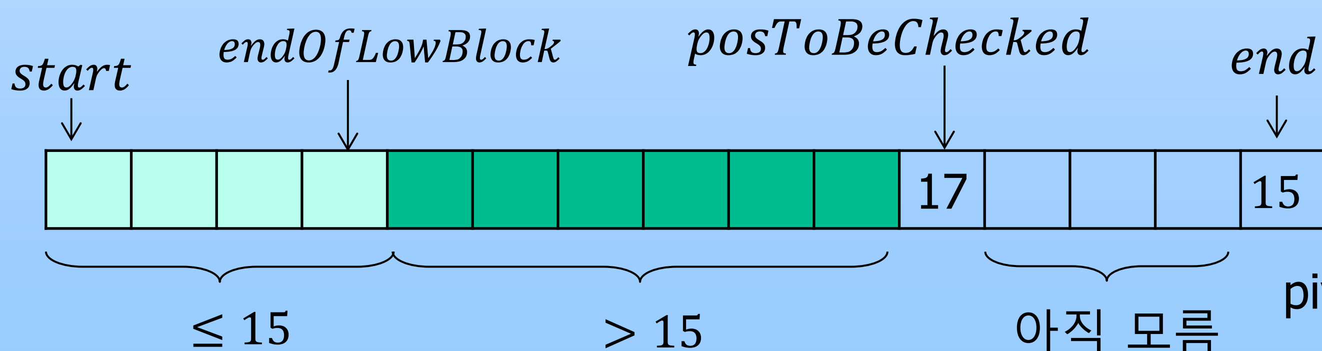
- Loop Invariant: Initialization
  - $endOfLowBlock < start$이므로 $[start,\ endOfLowBlock]$가 empty
    → 조건 1 성립 ($p \rightarrow q \equiv \sim p \vee q$이므로 p가 FALSE이면 항상 TRUE)
  - $(endOfLowBlock + 1) == \ posToBeChecked$이므로
    $[endOfLowBlock + 1, posToBeChecked - 1]$가 empty → 조건 2 성립
  - Line 1에 의해서 조건 3 성립

PARTITION $(A, start, end)$ // 배열과 처리할 위치의 시작과 끝
1 $pivotValue = A[end]$ // 끝 원소를 $pivotValue$로 지정
2 $endOfLowBlock = start - 1$ // $endOfLowBlock$은 empty로 시작
3 for $posToBeChecked = start$ to $end - 1$ // incremental method 사용

배열 Partition (10)

- Loop Invariant: Maintenance
  - $A[posToBeChecked]$ 가 pivot value보다
    - 작으면 $endOfLowBlock$이 1증가하고 $A[posToBeChecked]$와 $A[endOfLowBlock + 1]$ 교체 → 조건 1 만족
    - 크면 $posToBeChecked$만 1 증가하므로 조건 2 만족
  - $A[end]$ 는 변하지 않았으므로 조건 3 만족

```
for posToBeChecked = start to end − 1     // incremental method 사용
    if A[posToBeChecked] ≤ pivotValue
        endOfLowBlock = endOfLowBlock + 1
        exchange A[endOfLowBlock] with A[posToBeChecked]
```

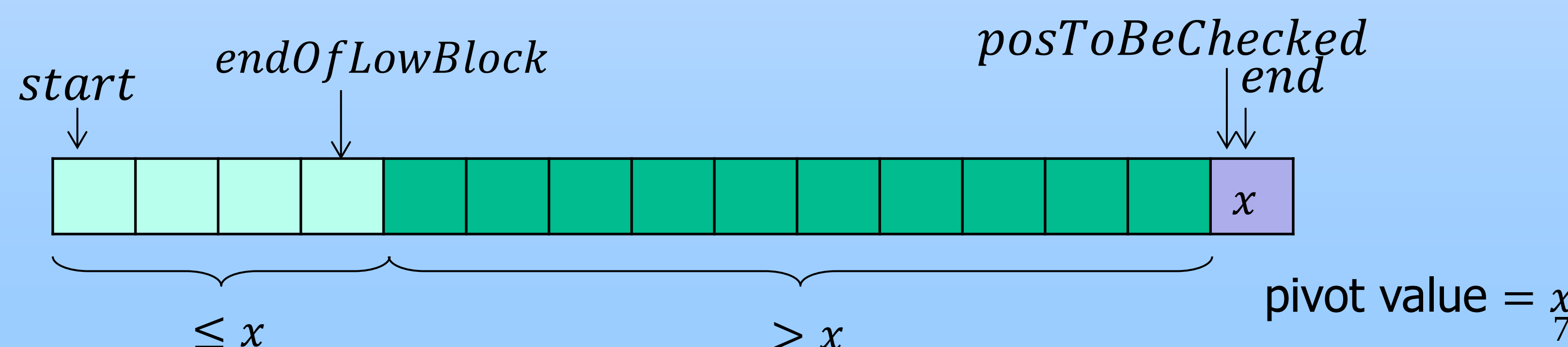


pivot value = 15

배열 Partition (11)

- Loop Invariant: Termination
  - $posToBeChecked = end$ 에서 종료
  - 입력 배열은 세 개의 집합으로 구성
    - pivotValue보다 작은 원소 집합, 큰 원소 집합, pivot 자체
  - 각각은 loop invariant의 각 조건을 만족함.

```
for posToBeChecked = start to end − 1     // incremental method 사용
    if A[posToBeChecked] ≤ pivotValue
        endOfLowBlock = endOfLowBlock + 1
        exchange A[endOfLowBlock] with A[posToBeChecked]
```



Quicksort Performance (1)

- Time complexity에 가장 크게 영향을 주는 요소
  - Partitioning 시 왼쪽과 오른쪽의 균형 정도
  - 즉, subproblem에서의 문제 크기인 n_1과 n_2의 균형

$$T(n) = T(n_1) + T(n_2) + \Theta(n)$$

Quicksort Performance (2)

- Worst-case Partitioning
  - Partition했을 때 n-1개가 한쪽에 들어가는 경우
  - 즉, 제일 큰 원소 혹은 제일 작은 원소를 pivot으로 지정한 경우

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

  - $T(n) = \Theta(n^2)$

Quicksort Performance (3)

- Best-case Partitioning
  - 두 개의 subproblems 크기가 동일한 경우

$$T(n) = 2T(n/2) + \Theta(n)$$

$$\Downarrow$$

$$T(n) = \Theta(n \lg n)$$

Quicksort Performance (4)

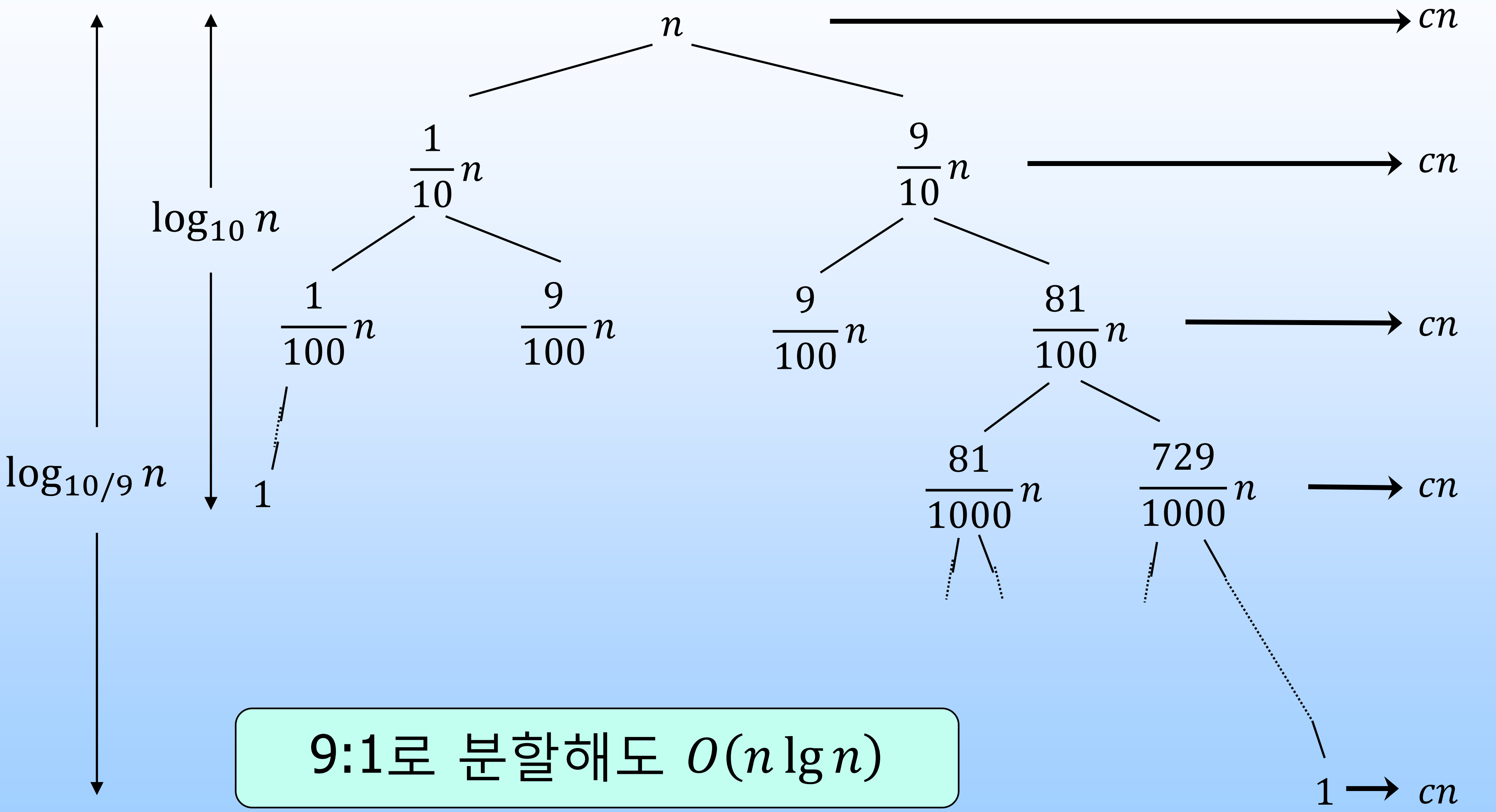
- Balanced Partitioning
  - 두 개 subproblem의 크기가 균형을 맞추어서 time complexity가 $O(n \lg n)$가 되는 경우
  - 어느 정도 되어야 balanced일까?
    - 문제) 9:1로 분할된 경우는 balanced일까 아닐까?

$$T(n) = T(9n/10) + T(n/10) + cn$$

$$\Downarrow$$

$T(n) = O(n \lg n)$ 혹은 $O(n^2)$?

$T(n) = T(9n/10) + T(n/10) + cn$의 Recursion Tree



9:1로 분할해도 $O(n \lg n)$

Total : $O(n \lg n)$

Quicksort Performance (5)

- Worst-case
  - $\Theta(n^2)$
- Average-case
  - $\Theta(n \lg n)$
  - 분할 과정에서 well-balanced 분할과 bad-balanced 분할 과정이 섞여 있어도 average-case time complexity는 $\Theta(n \lg n)$이다.

Quicksort 손코딩

- C-style API
  - void quicksort(int *arr, int start, int end);

실습 주제 (7)

- Quicksort에서 사용할 partition 함수 구현
  - 기능
    - Partition하고 pivot의 위치를 return한다.
  - API를 정하고 구현한다.
  - 테스트 방법
    - void test_partition() 함수 구현
    - 이 함수의 동작 과정은 다음과 같다.
    - 1) 길이가 1~32인 배열을 차례로 생성하고 각 배열에 임의의 값을 저장한 후 partition() 함수를 호출하고 그 결과를 확인한다. (확인하는 방법은 아래 2번 참고)
    - 2) 호출 결과가 정상적인지 확인하는 함수를 구현해서 테스트한다. 즉, pivot의 위치보다 왼쪽에 있는 값들은 pivot 값보다 작거나 같고 오른쪽에 있는 값들은 큰 지를 확인하는 함수를 구현해서 테스트한다.
    - 3) 증가순서인 입력 데이터에 대해서 partition() 함수 결과를 확인한다.
    - 4) 감소순서인 입력 데이터에 대해서 partition() 함수 결과를 확인한다.

실습 주제 (8)

- Quicksort 함수 구현
  - 기능
    - Quicksort를 수행한다.
  - API를 정하고 구현한다.
  - 테스트 방법
    - void test_quicksort() 함수 구현
    - 이 함수의 동작 과정은 다음과 같다.
    - 1) 길이가 1~16인 배열을 차례로 생성하고 각 배열에 임의의 값을 저장한 후 quicksort() 함수를 호출하고 그 결과를 확인한다. (확인하는 방법은 아래 2번 참고)
    - 2) 호출 결과가 정상적인지 확인하는 함수를 구현해서 테스트한다. 즉, pivot의 위치보다 왼쪽에 있는 값들은 pivot 값보다 작거나 같고 오른쪽에 있는 값들은 큰 지를 확인하는 함수를 구현해서 테스트한다.
    - 3) 증가순서인 입력 데이터에 대해서 quicksort() 함수 결과를 확인한다. 기존 실습주제에서 구현한 정렬 결과 확인 함수를 이용해서 정렬 여부를 검사한다.
    - 4) 감소순서인 입력 데이터에 대해서 quicksort() 함수 결과를 확인한다. 기존 실습주제에서 구현한 정렬 결과 확인 함수를 이용해서 정렬 여부를 검사한다.

코딩 과제 HW#2.C (3)

- HW#2.C3
  - 실습 주제 7 구현
- HW#2.C4
  - 실습 주제 8 구현

Randomized Quicksort

- 방법
  - Partition할 때 pivot을 고르는 방법으로 배열의 원소 중 하나를 random하게 선택
  - 평균적으로 well-balanced partition이 이루어짐

RANDOMIZED-PARTITION $(A, start, end)$

$i =$ RANDOM$(start, end)$ // A[start..end]에서 한 원소를 random 선택
exchange $A[end]$ with $A[i]$
return PARTITION(A, p, r) // random 선택된 원소와 마지막 원소를 교체

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 Algorithm (1)

- 알고리즘의 동작 과정이 입력 뿐만 아니라 random number generator에 의해서 만들어진 값들에 영향을 받는 알고리즘
- 예제
  - Randomized quicksort 등
- Time Complexity 계산 방법
  - Expected running time 사용
- Expected running time vs Average-case running time
  - Average case running time
    - 입력의 확률 분포 고려
  - Expected running time
    - 알고리즘 자체 내에서 random 선택을 하는 경우에 사용

Randomized Algorithm (2)

- 장점
  - 입력에 관계없이 일정한 수준의 성능을 얻을 수 있음
  - 악의적인 입력 형태에 의한 서비스 성능 저하를 막을 수 있음
  - 예를 들어
    - 소팅 서비스를 제공 중이라고 가정
    - 악의적인 사용자가 서비스에 사용 중인 알고리즘 특성 파악
    - 알고리즘의 최악의 경우를 입력하여 부하 가중
    - 일반 사용자들의 요청 처리가 지연
    - Randomized algorithm 사용 시 이러한 피해 방지 가능

Indicator Random Variable

- Indicator random variable의 용도
  - 확률과 기대값 사이의 변환 방법
- Indicator random variable I{A}의 정의
  - I{A} = 1 if A occurs.
  - I{A} = 0 if A does not occur.
- 동전 던지기
  - 동전을 한 번 던졌을 때
    - 앞면(H)이 나올 확률 = 뒷면(T)이 나올 확률 = 1/2
  - 앞면이 나올 경우의 indicator random variable을 $X_H = I\{앞면\}$라 하면
    - 한번 던졌을 때 동전 앞면이 나올 기대값, $E[X_H]$

      $E[X_H]$ = E[I{앞면}]

      = 1 x Pr{앞면} + 0 x Pr{앞면이 아님}

      = 1 x (1/2) + 0 x (1/2)

      = 1/2

Quicksort: Expected Running Time (1)

- X를 quicksort 수행 시 4번째 비교문의 총 수행 횟수라 하면
- Quicksort의 time complexity = O($n + X$)
  - n: PARTITION 호출 횟수. 호출 시 1개의 pivot 지정됨.

PARTITION $(A, start, end)$ // 배열과 처리할 위치의 시작과 끝

1 $pivotValue = A[end]$ // 끝 원소를 $pivotValue$로 지정

2 $endOfLowBlock = start - 1$ // $endOfLowBlock$은 empty로 시작

3 for $posToBeChecked = start$ to $end - 1$ // incremental method 사용

4 if $A[posToBeChecked] \leq pivotValue$ // pivot과 다른 원소 대소 비교

5 $endOfLowBlock = endOfLowBlock + 1$

6 exchange $A[endOfLowBlock]$ with $A[posToBeChecked]$

7 exchange $A[endOfLowBlock + 1]$ with $A[end]$ // 값이 큰 블록의 첫 번째 원소와 입력 배열의 마지막 원소 교체

8 return $endOfLowBlock + 1$ // 값이 큰 블록의 첫 번째 원소 위치를 return

Quicksort: Expected Running Time (2)

- $X_{ij} = I\{$원소 z_i와 원소 z_j를 비교$\}$로 정의하자.
  - 즉, 두 개 원소가 비교되면 $X_{ij} = 1$
  - 아니면 $X_{ij} = 0$
  - 여기서 z_i는 배열에서 i번째로 작은 원소를 의미한다.
- X는 비교문 수행 횟수라서 모든 원소들 사이의 비교 횟수 총합임.

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

- X의 기대값

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{와 } z_j \text{ 를 비교}\}$$

Quicksort: Expected Running Time (3)

- z_i가 z_j와 비교될 확률
  - z_i 와 z_j 사이의 다른 값이 먼저 pivot이 되면 두 개는 비교되지 않음.
  - 둘 중 하나가 pivot으로 선택된 경우에만 비교됨
  - Pivot으로 사용되면 더 이상 비교되지 않으므로 한 번 비교되면 더 이상 비교되지 않음.
  - 따라서, z_i가 z_j와 비교될 확률(기대값)
    - z_i와 z_j 사이의 원소 중 z_i 혹은 z_j가 첫 번째로 pivot이 될 확률
    - 즉, $j-i+1$개의 원소 중 z_i와 z_j를 선택할 확률
  - 결론

$$\Pr\{z_i\text{가 } z_j\text{와 비교}\} = \frac{2}{j-i+1}$$

    - 예) 인접한 두 원소 z_i 와 z_{i+1} 비교 확률(기대값) $= 1$

Quicksort: Expected Running Time (4)

- 따라서,

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{와 } z_j \text{ 를 비교}\}$$

$$\begin{aligned}
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) \, .
\end{aligned}$$

↓ 알고리즘 교재 식 A.7 참고

Quicksort: Expected Running Time (5)

- 결론
  - Randomized quicksort의 expected running time은 $O(n \lg n)$

HW#2.P (3)

- HW#2.P12
  - 알고리즘 교재 연습 문제 7.1-1
- HW#2.P13
  - 알고리즘 교재 연습 문제 7.2-3
- HW#2.P14
  - 알고리즘 교재 연습 문제 7.4-5
- HW#2.P15
  - 알고리즘 교재 연습 문제 5.3-7

목 차

- Heapsort (Chapter 6)
- Quicksort (Chapter 7)
- Sorting in Linear Time (Chapter 8)
- Medians and Order Statistics (Chapter 9)

Sort 알고리즘의 구분

- Comparisons(비교) 방식 알고리즘

- 비교 방식이 아닌 정렬 알고리즘

비교 방식의 정렬 알고리즘

- 원소들의 순서를 정하기 위해서 대소를 비교하는 방법을 사용하는 정렬 알고리즘
- Insertion sort, quicksort, heapsort, mergesort 등

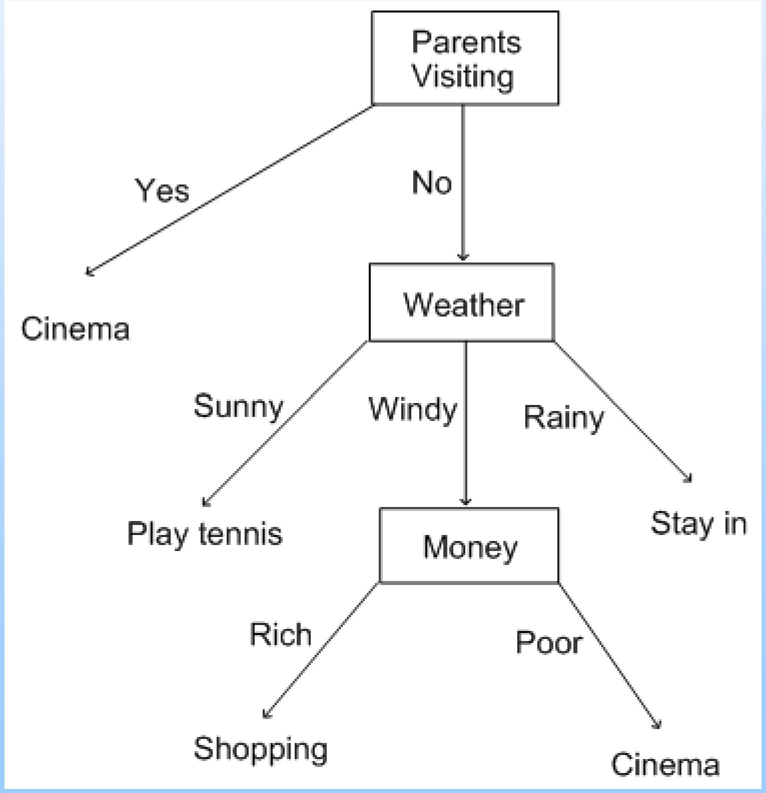
Worst case의 lower bound는?

$$\Omega(n \lg n)$$

Decision Tree Model을 사용하여 증명

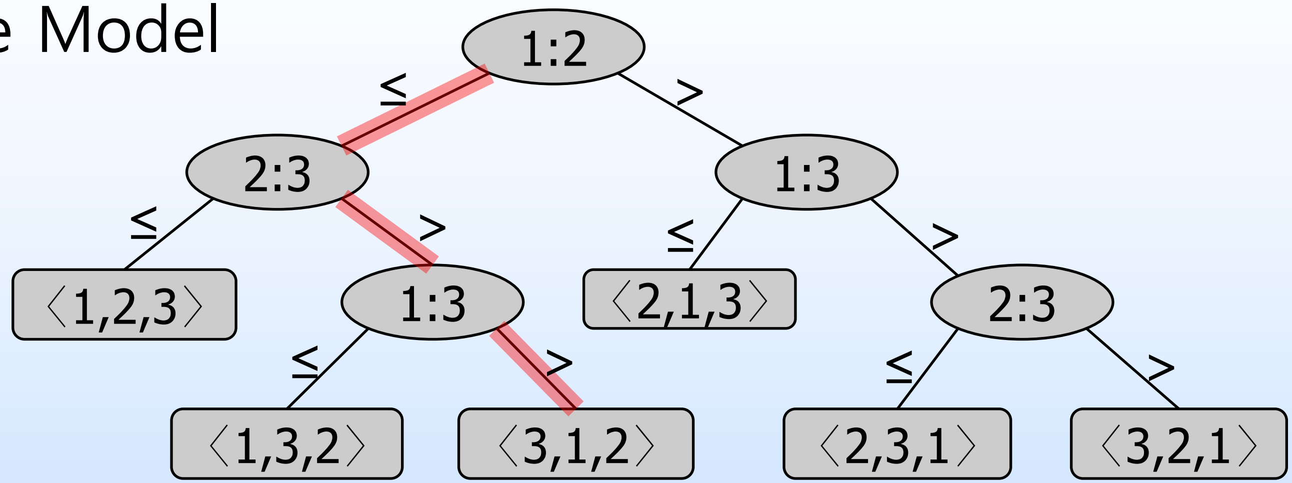
Decision Trees

- 각 노드는 질문에 해당
- 질문의 결과에 따라서 edge를 따라 다음 질문으로 이동
- Leaf node에 도달하면 결과를 얻음
- 예제



Lower Bounds for Sorting (1)

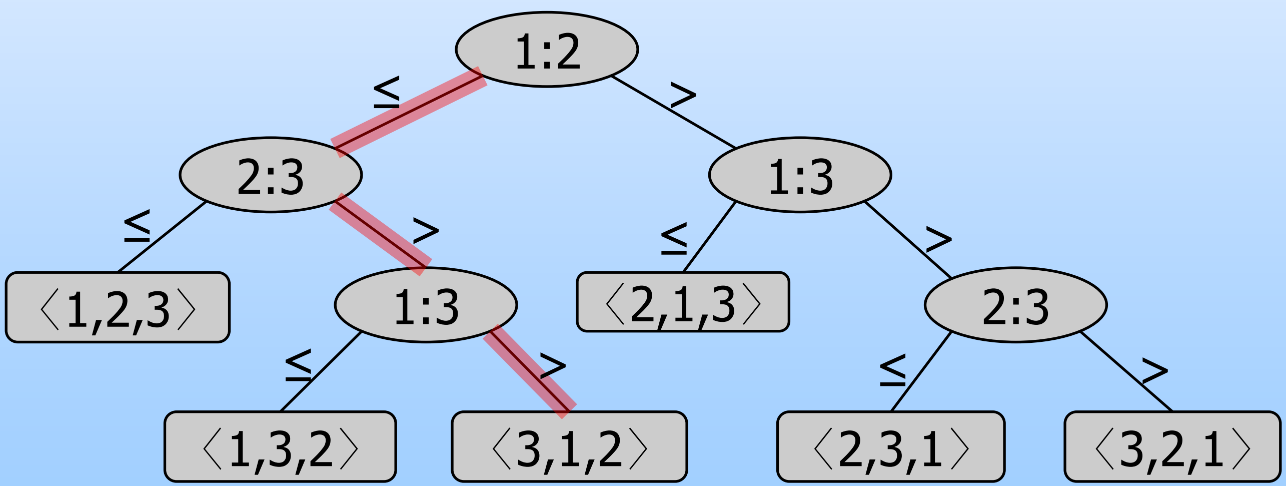
◆ Decision Tree Model



- 주어진 크기의 입력 배열을 sort할 때 sort algorithm이 수행되는 과정을 decision tree로 표현하는 방법
- 각 노드는 비교하는 두 개의 element들을 표시
- Leaf node는 정렬된 결과를 표시
  - 배열의 크기가 n일 때 leaf node의 개수는 n!
- 예제 그림) 3개의 원소로 insertion sort할 때의 decision tree
  - 3! = 6개의 leaf node 존재
  - <3, 1, 2>는 입력이 $a_1 = 6, a_2 = 8, a_3 = 5$인 경우에 해당함.

Lower Bounds for Sorting (2)

- Worst Case의 Lower Bound
  - Decision tree의 height
    - 주어진 sort algorithm의 worst case에서의 비교 횟수와 같음
    - 아래 그림(insertion sort)의 worst case에서의 비교 횟수는 3임.
  - 모든 종류의 decision trees 중 tree height가 제일 낮은 decision tree가 비교 방식 정렬 알고리즘의 lower bound가 됨



Lower Bounds for Sorting (3)

- Theorem 8.1
  - Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.
  - 즉, worst case에도 $O(n \lg n)$보다 빠른 알고리즘은 존재하지 않는다.
- Proofs
  - 원소 개수가 n개일 때 나올 수 있는 순서 종류의 수 $n!$
  - Leaf node 개수를 l 이라 할 때 $l = n!$
  - Height가 h인 binary tree의 최대 leaf node 개수는 2^h보다 작거나 같음
  - 따라서, leaf node 개수가 n!인 binary tree의 height h는?
    - $2^h \geq l$ 이고 $l = n!$ 이므로 $2^h \geq n!$ 를 만족해야 함
  - 따라서, $h \geq \lg(n!)$

$= \Omega(n \lg n)$ by Stirling's approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \quad \text{(교재 식 3.18)}$$

$$\lg(n!) = \Theta(n \lg n) \quad \text{(교재 식 3.19)}$$

또는 자구알 강의자료 basic_concepts 48페이지

비교 방식이 아닌 정렬 알고리즘

- 종류
  - Counting sort
  - Radix sort
  - Bucket sort
- 특징
  - $\Omega(n \lg n)$가 적용되지 않음
  - Linear time, O(n), algorithms

순위 추출 Problem

- 입력
  - N명 학생들의 점수(0점~100점)를 저장한 정수 배열
- 출력
  - O(n)에 각 학생의 등수를 계산
- 예제
  - 입력 배열: 10, 90, 80, 85 (점수)
  - 출력 배열: 4, 1, 3, 2 (순위)
- 손코딩 (C-style API)
  - int calculateRanks (int *score, int *rank, int totalNum);

Counting Sort (1)

- 0부터 k 사이의 정수 n개가 입력으로 주어져 있고 $k = O(n)$ 일 때 $\Theta(n)$ 시간으로 정렬하는 방법

- 아이디어
  - 값이 x인 원소에 대해서
  - x 보다 작은 원소의 개수를 센 후
  - x 가 출력 배열에 들어갈 위치에 바로 저장한다.

Counting Sort (2)

- 입력/출력
  - 입력: 크기 n인 배열 A[1..n]
  - 출력: 크기 n인 배열 R[1..n]
- 방법
  - 0부터 k사이의 값을 저장하는 배열 C[0..k]선언하고 모든 값들을 0으로 초기화
  - 입력 배열을 scan하면서 그 값이 m이면 C[m] 값을 1 증가
    - C[m]: 값은 입력 배열에서 값이 m인 원소 개수
  - 배열 C[0..k]를 scan하면서 각 원소에 대해서 자신보다 작거나 같은 값의 개수를 누적
    - C[m]: 입력 배열에서 값이 m보다 작거나 같은 원소 개수
  - 입력 배열을 뒤에서부터 앞으로 scan하면서 원소의 값이 m이면 출력 배열의 C[m] 위치에 저장하고 C[m]값을 1 줄인다.
    - C[m] 값을 1 줄여야 값이 m인 다음 원소를 정렬 결과의 올바른 위치에 넣을 수 있음.

Counting Sort (3)

입력 A								
	2	5	3	0	2	3	0	3

임시 C

0	1	2	3	4	5
0	0	0	0	0	0

초기화

0	1	2	3	4	5
2	0	2	3	0	1

개수 측정

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

개수 누적

입력 A

2	5	3	0	2	3	0	3

임시 배열 C

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

출력 B

1	2	3	4	5	6	7	8
						3	

입력 A

2	5	3	0	2	3	0	3

임시 배열 C

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

출력 B

1	2	3	4	5	6	7	8
	0					3	

Counting Sort (4)

입력 A

2	5	3	0	2	3	0	3

임시 배열 C

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

↓

1	2	4	5	7	8

출력 B

1	2	3	4	5	6	7	8
	0				3	3	

입력 A

2	5	3	0	2	3	0	3

임시 배열 C

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

↓

2	2	3	5	7	8

출력 B

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

- - - - - - - - - - - - - - - - - - - -

입력 A

2	5	3	0	2	3	0	3

임시 배열 C

0	1	2	3	4	5
0	2	3	4	7	7

↓

0	2	2	4	7	7

출력 B

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

Counting Sort (5)

- Pseudo Code

```
COUNTING-SORT(A, B, k)                    // B는 출력 배열
    let C[0..k] be a new array
    for i = 0 to k
        C[i] = 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
    // C[i] now contains the number of elements equal to i.
    for i = 1 to k
        C[i] = C[i] + C[i-1]
    // C[i] now contains the number of elements less than or equal to i
    for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

Counting Sort (6)

- Time Complexity
  - $\Theta(n+k)$
    - 입력 배열을 읽고 출력 배열에 저장하는 작업: $\Theta(n)$
    - 숫자 k보다 작거나 같은 원소 개수 찾는 작업: $\Theta(k)$
  - $k = O(n)$인 경우: $\Theta(n)$
  - $\Omega(n \lg n)$보다 빠름

- Stable sort 기법에 해당됨
  - 값이 같은 두 개의 원소 중 입력 배열에서 앞에 있던 원소가 출력 배열에서도 앞에 있음
  - 증명
    - HW#2.P17 (알고리즘 교재 8.2-2)

Counting Sort 손코딩

- C-style API
  - void count_sort(int *arr, int size);

Radix Sort (1)

- Radix의 의미
  - [수학] 근, 기수
  - 10진수의 radix는 10
    - $315 = 3 \times 10^2 + 1 \times 10^1 + 5 \times 10^0$
- 문제
  - d개 자리(digit)로 표현되는 숫자들을 정렬하라.
- 예제
  - d=3인 10진수 집합 정렬
    - 329, 457, 657, 839, 436, 720, 355
  - (년도, 달, 일)의 tuple 정렬
    - tuple은 세 개 keys로 표현되는 record를 의미
    - (2013, 5, 9), (2012, 12, 10), (2010, 3, 8), (2013, 1, 3)

Radix Sort (2)

- 직관적인 방법
  - 가장 상위 자리의 수부터 비교해 순서를 정한다
  - 가장 상위 자리의 수가 같은 숫자들끼리 모아서 그 다음 상위 자리의 수를 비교한다.
  - 이 과정을 맨 하위 자리까지 반복한다.

329		329		329		329
437		355		355		355
657		437		437		436
839	→	436	→	436	→	437
436		657		657		657
820		839		820		820
355		820		839		839

  - 단점: 주어진 자리의 수 기준으로 정렬한 상태에서 하위 자리의 수를 추가 비교하기 위해서 자리의 수가 같은 값끼리 취합해야 함.

Radix Sort (3)

- 가장 낮은 자리의 수부터 비교하는 방법
  - 가장 낮은 자리의 수를 기준으로 stable sort 적용
    - 예) counting sort
  - 두 번째 낮은 자리의 수를 기준으로 stable sort 적용
  - 최상위 자리 기준으로 정렬할 때까지 위 작업을 반복한다.

329	720	720	329
437	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

- 증명: HW#2.P19 (알고리즘 교재 연습 문제 8.3-3)

Radix Sort (4)

- Pseudo code

```
RADIX-SORT(A, d)
    for i = 1 to d
        use a stable sort to sort array A on digit i
```

Radix Sort (5)

- Lemma 8.3
  - Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time if the stable sort it uses takes $\Theta(n+k)$ time.
- 의미
  - k: 10진수인 경우 10
  - d-digit numbers: 100은 3자리 숫자임
- 증명
  - Stable sort(예: counting sort)를 1번 수행하는데 $\Theta(n+k)$ 소요되고 이 작업을 d번 반복하므로 $\Theta(d(n+k))$이다.

Radix Sort (6)

- Lemma 8.4
  - Given n b-bit numbers and any positive integers $r \le b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n + 2^r))$ time if the stable sort it uses takes $\Theta(n + k)$ time for inputs in the range 0 to k.
- 의미
  - 32bit word를 8bit 자리의 수 4개로 되어 있다고 볼 경우
    - b=32, r=8, $k = 2^r = 256$ (256진수)
    - $d = b/r = 4$ (256진수의 4자리 숫자)
  - 2^r진수에서 b/r 자리수 숫자들의 RADIX-SORT 시간
- 증명
  - Lemma 8.3과 거의 동일함.

Radix Sort (7)

- Time complexity
  - $\Theta((b/r)(n+2^r))$, $r \le b$일 때
    - $b = O(\lg n)$이면
    - $r \approx \lg n$로 지정한 경우에 $2^{\lg n} = n$이므로
    - $\Theta\left(\left(\frac{b}{r}\right)(n+2^r)\right) = \Theta(n)$
  - Quicksort(average-time complexity $\Theta(n \lg n)$) 보다 좋음
  - 반면에 constant factor가 quicksort 대비 매우 큼
  - Quicksort의 경우 hardware cache를 좀 더 효율적으로 사용
- Space complexity
  - Counting sort 사용 시 in-place 알고리즘이 아님
  - 메모리 용량이 부족한 경우 quicksort 등의 in-place 알고리즘을 사용하는 것이 좋음

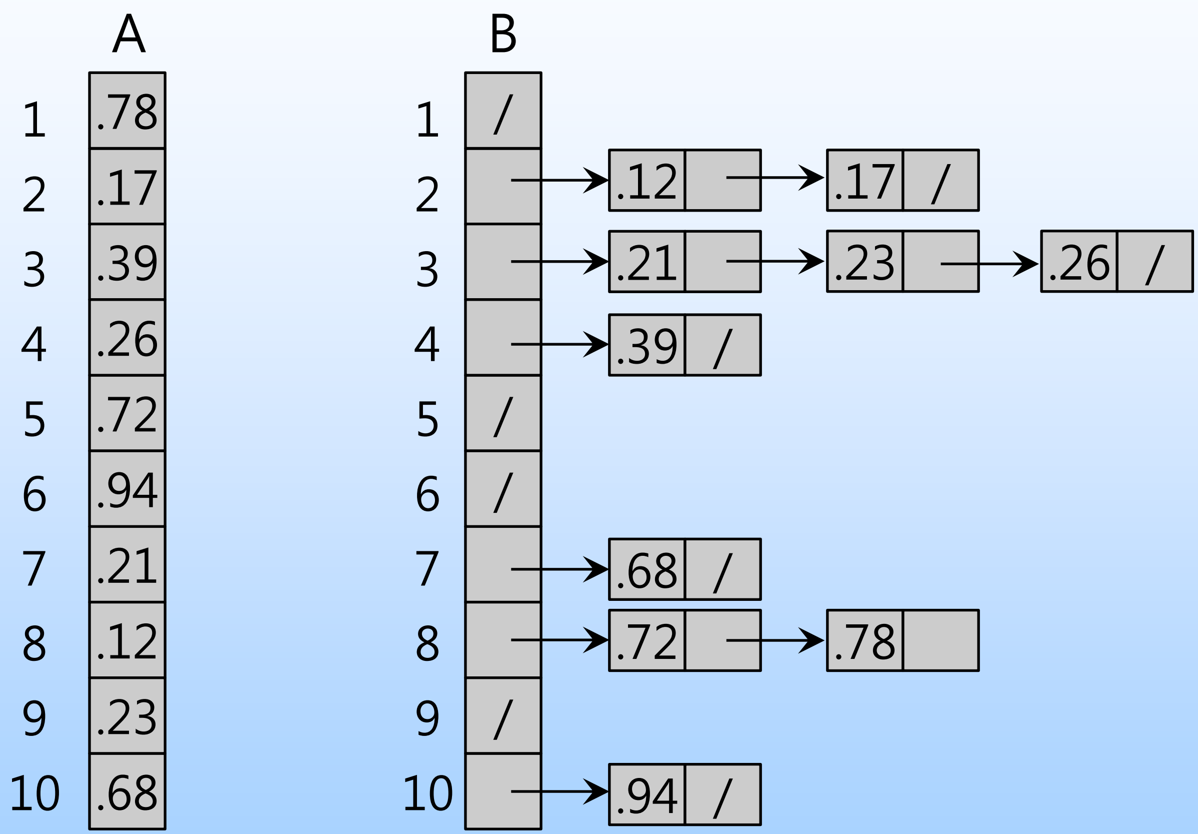
Radix Sort 손코딩

- C-style API
  - void radix_sort(int *arr, int size, int digitNum);

Bucket Sort (1)

- 가정
  - Input이 [0, 1) 의 영역에서 uniform하게 분포되어 있음
- 방법
  - [0, 1) 사이의 영역을 동일한 크기로 여러 개의 sub-interval로 분할
    - 각각의 sub-interval을 bucket이라고 부름
  - 각 원소들을 buckets에 넣어줌
  - 각 bucket 내에서 원소들을 정렬
    - Linked list 기반의 insertion sort를 사용

Bucket Sort (2)



(a) (b)

Bucket Sort (3)

- Pseudo code

BUCKET-SORT(A)

let $B[0..n-1]$ be a new array   // bucket 용도의 array 선언

$n = A.length$

for $i = 0$ to $n-1$

    make $B[i]$ an empty list   // bucket 초기화

for $i = 1$ to n

    insert $A[i]$ into list $B[nA[i]]$ // [0..1) → [0..n) 변환 후 배열에 저장

for $i = 0$ to $n-1$

    sort list $B[i]$ with insertion sort

concatenate the list $B[0]$, $B[1]$, …, $B[n-1]$ together in order

Bucket Sort (4)

- Time Complexity

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

- Average-case running time
  - 입력 분포에 대한 기대값: $E[T(n)]$

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$,여기서 $E[n_i^2]$는 constant

$$= \Theta(n)$$

코딩 과제 HW#2.C (4)

- HW#2.C5
  - Counting sort 구현
- HW#2.C6
  - Radix sort 구현

HW#2.P (4)

- HW#2.P16
  - 알고리즘 교재 연습 문제 8.2-1
- HW#2.P17
  - 알고리즘 교재 연습 문제 8.2-2
- HW#2.P18
  - 알고리즘 교재 연습 문제 8.3-1
- HW#2.P19
  - 알고리즘 교재 연습 문제 8.3-3
- HW#2.P20
  - 알고리즘 교재 연습 문제 8.4-4

목 차

용어 정리

- k-th order statistic of a set of n elements
  - k-th 번째 작은 원소
- minimum
  - The first order statistic ($k = 1$)
- maximum
  - The n-th order statistic ($k = n$)
- median
  - 중간점에 해당
  - Lower median = $\left\lfloor \frac{n+1}{2} \right\rfloor$
  - Upper median = $\left\lceil \frac{n+1}{2} \right\rceil$

Selection Problem 정의

- Input
  - A set of n (distinct) numbers and an integer k, with $1 \le k \le n$
- Output
  - The element $x \in A$ that is larger than exactly $k-1$ other elements of A.

Minimum & Maximum (1)

- Minimum을 찾기 위한 최소 비교 횟수
  - n – 1
- Maximum을 찾기 위한 최소 비교 횟수
  - n – 1
- Minimum과 maximum을 동시에 찾기 위한 비교 횟수
  - $3\lfloor n/2 \rfloor$ 이내
  - 핵심 아이디어
    - 비교 과정에서 중간 상태까지의 최대값과 최소값을 동시에 유지
    - 다른 2개 원소를 비교해서 큰 값과 작은 값 추출
    - 작은 값과 최소값 비교, 큰 값과 최대값 비교로 최소값, 최대값 갱신

Minimum & Maximum (2)

- $3\lfloor n/2 \rfloor$ 이내의 비교로 minimum/maximum을 동시에 찾는 방법
  - n 이 홀수일 때
    - 1개 값을 최대/최소값으로 동시 지정
    - 나머지 원소들을 2개씩 묶어서 처리.
    - 총 $\lfloor n/2 \rfloor$번 수행 → 매번 3번의 비교 → $3\lfloor n/2 \rfloor$
  - n 이 짝수일 때
    - 원소 2개를 비교해서 (최소, 최대)를 구함
    - 나머지 원소들을 2개씩 묶어서 처리
    - 초기에 두 개 원소를 1번 비교 후 2개씩 처리 작업을 $(n-2)/2$번 수행했으므로 $1+3(n-2)/2=3n/2-2$
  - 따라서, 최대 $3\lfloor n/2 \rfloor$ 이내로 최소값, 최대값 추출

두 번째로 작은 값 찾기

- Theorem
  - n 개의 원소 중 두 번째로 작은 값은 worst case에 $n + \lceil \lg n \rceil - 2$ 횟수 비교해서 찾을 수 있다.
- 증명
  - HW#2.P23 (알고리즘 교재 연습문제 9.1-1)
  - 힌트1: 최소값 찾기
  - 힌트2: 토너먼트 트리

Selection in expected linear time (1)

- 목표
  - 주어진 n 개의 서로 다른 원소 중 k번째로 작은 원소를 expected running time $\Theta(n)$에 찾는 방법

- 아이디어
  - Randomized quicksort를 응용
  - Pivot의 최종 위치와 k를 비교
    - k 가 pivot의 최종 위치보다 작다면 pivot보다 작은 값 중 하나가 k번째 원소가 됨
    - k가 pivot의 최종 위치보다 크다면 pivot보다 큰 값 중 하나가 k번째 원소가 됨

k번째 작은 원소 찾기 손코딩

- C-style API
  - void selectKth(int *arr, int first, int last, int kth);
    - randomized_partition()함수는 있다고 가정

Selection in expected linear time (2)

◆ Pseudo Code

```
RANDOMIZED-SELECT(A, first, last, k) // [first..last] 에서 k번째 찾기. k ≥ 1.
    if first == last             // 이 코드에서는 first ≤ k ≤ last 검사 누락되어 있음.
        return A[first]          // i 번째를 찾았으므로 return.
    posOfPivot = RANDOMIZED-PARTITION(A, first, last)
    pivotOrder = posOfPivot – first + 1 // first로부터의 거리 계산. 입력이
    //A[1..n]라서 "+1"이 있음. 예를 들어 posOfPivot == first라면 pivotOrder = 1.
    if k == pivotOrder
        return A[pivotOrder]                 // k 번째를 찾았으므로 return
    else if k < pivotOrder                   // 작은쪽에서 찾기
        return RANDOMIZED-SELECT(A, first, posOfPivot – 1, pivotOrder)
    else                                     // 큰 쪽에서 찾기. 찾고자 하는 순위 감소.
        return RANDOMIZED-SELECT(A, posOfPivot + 1, last, k – pivotOrder)

// 호출하는 곳
    value = RANDOMIZED-SELECT(A, 1, n, k) // A[1..n]에서 k(≥ 1)번째 찾기.
```

Selection in expected linear time (3)

- Time Complexity
  - Expected running time: $\Theta(n)$
  - 상세한 증명은 알고리즘 교재 217~219 페이지
- 직관적인 증명
  - RANDOMIZED-PARTITION 소요 시간: $\Theta(n)$
  - Pivot이 정해졌을 때
    - k번째 원소를 찾았으면 종료
    - k번째 원소를 찾지 못했다면 작은 쪽 그룹 혹은 큰 쪽 그룹에서 k번째 원소를 찾으므로 나머지 한 쪽은 더 이상 볼 필요 없음
  - Partition할 때마다 평균적으로 $n/2$ 만큼 고려 대상에서 제외된다면 partition에서의 비교 횟수는 $n \to \frac{n}{2} \to \frac{n}{4} \ldots \to n/2^k$로 줄어든다.
  - 총 비교 횟수:

$$\sum_{k=0}^{\lg n} n/2^k < \sum_{k=0}^{\infty} n/2^k = 2n = \Theta(n)$$

코딩 과제 HW#2.C (5)

- HW#2.C7
  - k번째 원소를 찾는 O(n) expected running time 방법 구현

Selection in worst-case linear time (1)

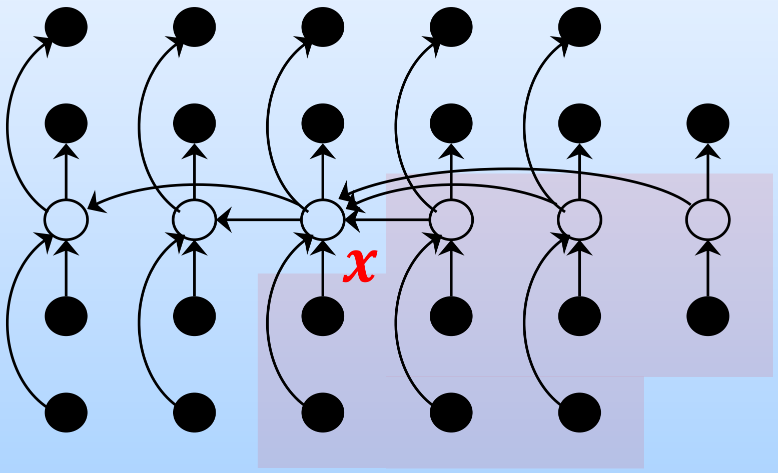
- 목표
  - 주어진 n 개의 서로 다른 원소 중 k번째로 작은 원소를 worst case에 $O(n)$ 시간에 찾는 방법

Selection in worst-case linear time (1)

- 방법
  - 5개 원소로 이루어진 그룹 구성
  - 각각의 그룹에서 median 추출
  - 추출된 median들에서 다시 median of medians 추출 (그림에서 x)
  - 화살표 b ← a는 b < a를 의미
    - 그림에서 x 왼쪽 위 < x
    - 그림에서 x 오른쪽 아래 > x
    - 그 외는 x 와의 대소 관계 모름
  - x 오른쪽 아래 원소 최소 개수

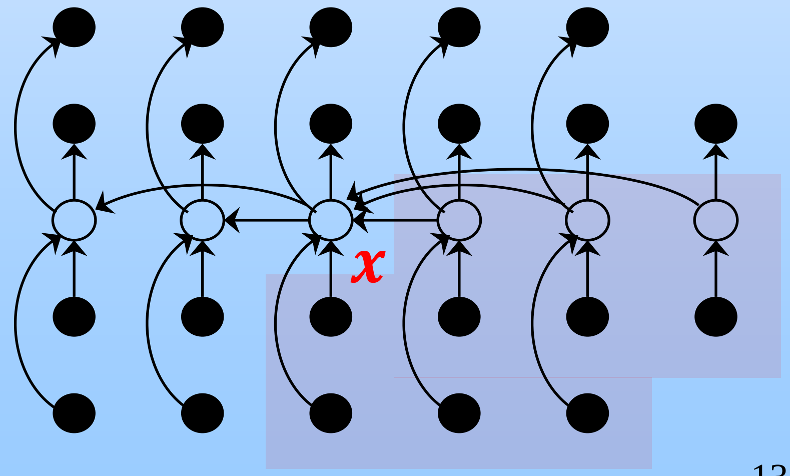
$$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil-2\right)\geq\frac{3n}{10}-6$$

    - -2는 x 를 포함하는 그룹과 맨 마지막 그룹을 포함하지 않음을 의미
  - x 왼쪽 위 원소 개수
    - $3n/10-6$보다 많음



Selection in worst-case linear time (2)

- 방법 (cont)
  - Median of medians x 를 기준으로 partition 수행해서 median of medians x 가 몇 번째 값인지 확인
  - Median of medians x가 k번째보다 크다면
    - x의 오른쪽 아래 영역에 있는 숫자는 k번째일 수 없음
  - Median of medians이 k번째보다 작다면
    - x의 왼쪽 위 영역에 있는 숫자는 k번째일 수 없음
  - k번째일 수 없는 원소들을 제외하고 나머지 원소들을 대상으로 recursive 방법으로 진행



Selection in worst-case linear time (3)

- Time Complexity
  - $T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$
    - $T(\lceil n/5 \rceil)$: median of medians 찾는 시간
    - $T(7n/10 + 6)$: $n - (3n/10 - 6)$에서 k번째 원소 찾는 시간
    - $O(n)$: partition 시간
  - Substitution method로 풀면 $T(n) = O(n)$
    - 풀이 과정은 교재 참고

- 의미
  - k번째 원소를 구하기 위해서 비교 기반의 sort를 사용할 경우
    - $\Omega(n \lg n)$
  - Linear-time selection algorithms을 사용할 경우
    - $O(n)$
  - k번째 원소를 구하기 위해서 sort를 사용하는 것은 비효율적임

HW#2.P (5)

- HW#2.P21
  - 알고리즘 교재 연습 문제 9.3-8
- HW#2.P22
  - 알고리즘 교재 연습 문제 9.3-9
- HW#2.P23
  - 알고리즘 교재 연습 문제 9.1-1