정렬

- 간단한 정렬 알고리즘
  - 선택정렬, 버블정렬, 삽입정렬
- 분할과 정복 (Divide and Conquer) 방법
- 분할과 정복에 의한 알고리즘
  - 병합정렬과 퀵정렬
- 힙정렬 (Heapsort)
- 정렬 하한계

힙(Heap)과 힙정렬 (Heap Sort)

힙(Heap)

◆ Heap(힙) is a binary tree with the following properties:

(1) Structure property (구조적 성질):

- A binary tree is complete (완전이진트리) if
  - Every level except the last is full
  - In the last level, every leaf is as far to the left as possible (마지막 level에서는 노드들이 왼쪽부터 오른쪽으로 꽉 채워져 있다)

(2) Heap (Heap Order) property:

For every node x, the value (key) in the node x is greater than or equal to the values (keys) in its children

⇒ 최대힙 (Max Heap)

Heap의 성질 (Max and Min)

- 최대힙 (Max-Heap) 정렬에서는 최대힙 사용
  - For every node excluding the root, value is at most that of its parent.
  - Largest element is stored at the root.
  - In any subtree, no values are larger than the value stored at subtree root.

- 최소힙 (Min-Heap)
  - For every node excluding the root, value is at least that of its parent.
  - Smallest element is stored at the root.
  - In any subtree, no values are smaller than the value stored at subtree root

Heap

- Height of a heap with n nodes:
  $\lfloor \log_2 n \rfloor$

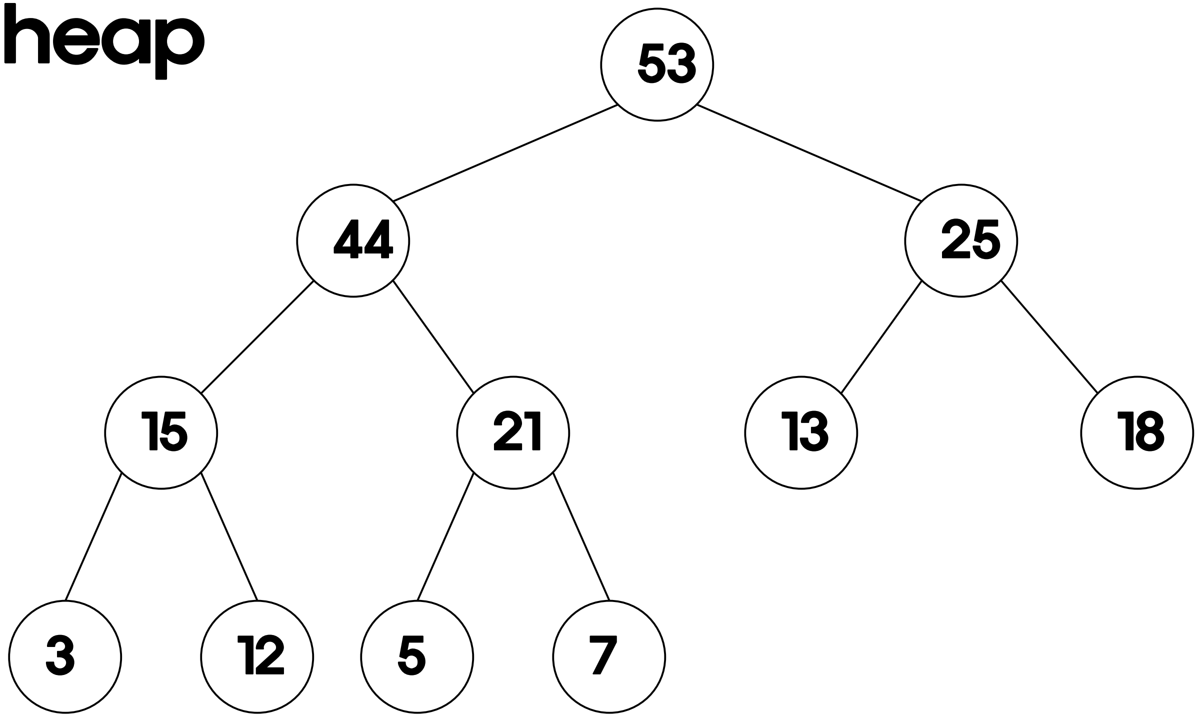
n개 노드를 가진 힙의 높이 $\lfloor \log n \rfloor$

- Number of leaves of a complete binary tree with n nodes: $\lceil n/2 \rceil$

n개 노드를 가진 힙의노드수 $= \lceil n/2 \rceil$

Example:

max heap



배열 구현

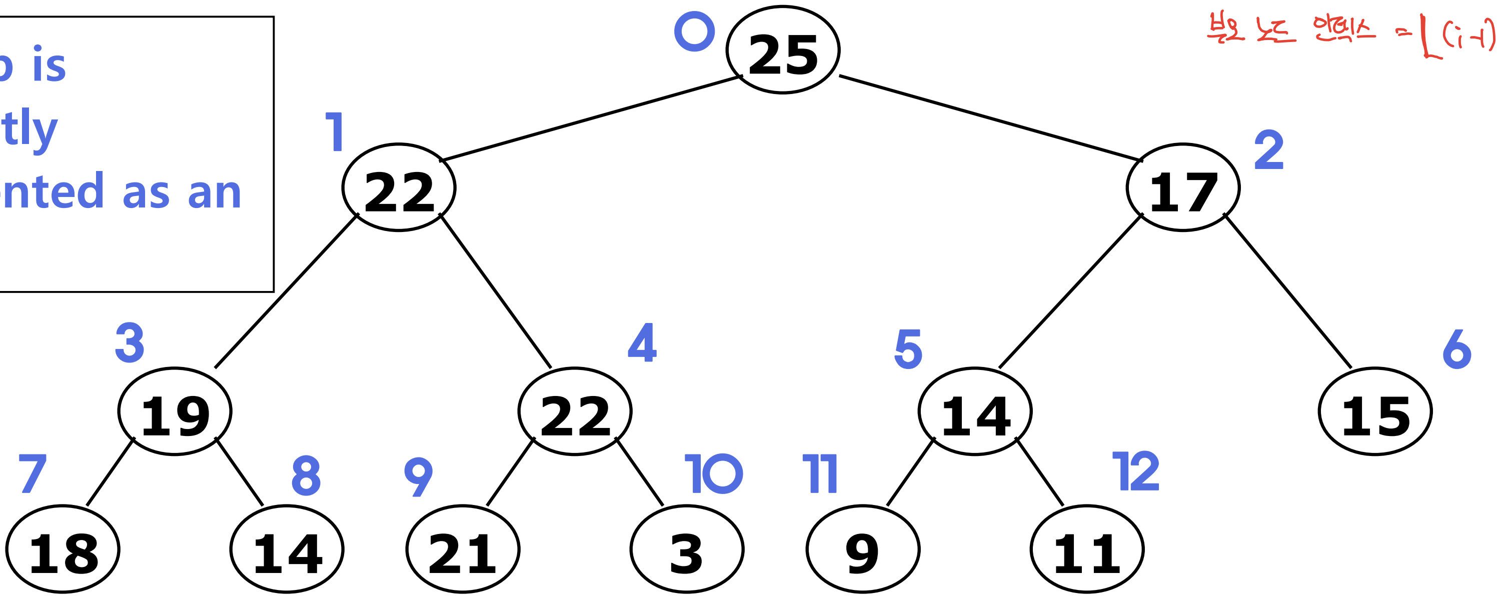
i가 특정 노드의 인덱스일때

왼쪽 자식 노드 인덱스 = $2*i+1$

오른쪽 자식 노드 인덱스 = $2*i+2$

부모 노드 인덱스 = $\lfloor (i-1)/2 \rfloor$

- A heap is efficiently represented as an array.



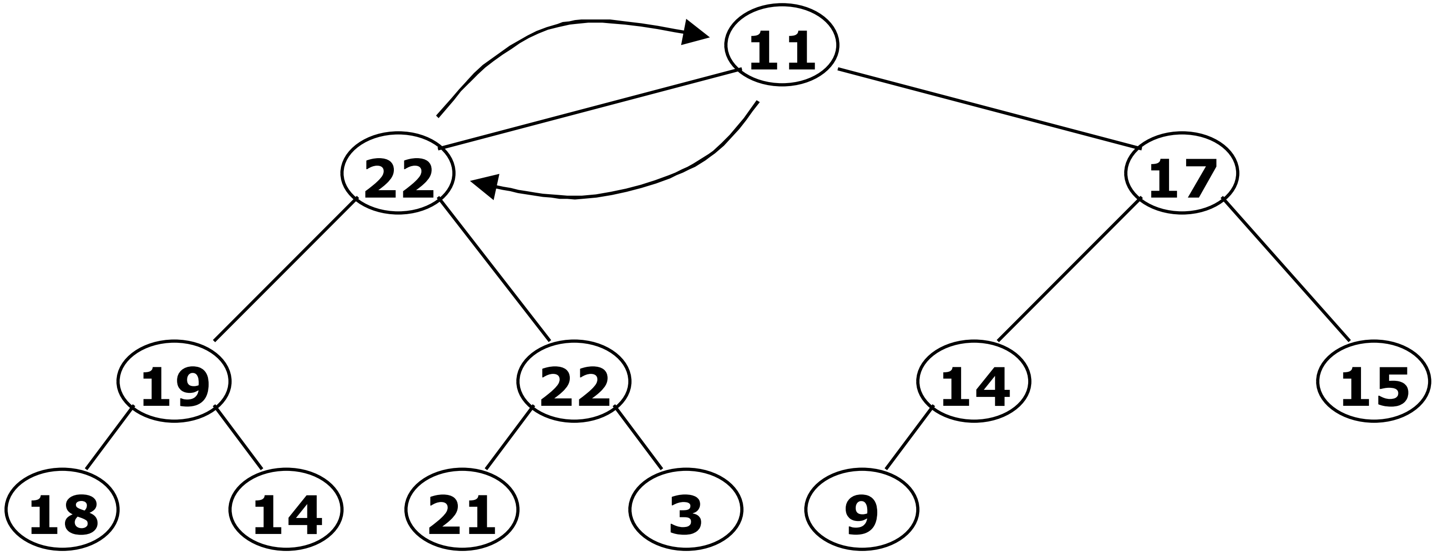
	0	1	2	3	4	5	6	7	8	9	10	11	12
A	25	22	17	19	22	14	15	18	14	21	3	9	11

◆ n is heap size (the number of nodes)

- The left child of index i is at index $2*i+1$ if $2*i+1 < n$
- The right child of index i is at index $2*i+2$ if $2*i+2 < n$
- The parent of index i is at $\lfloor (i-1)/2 \rfloor$ if $i \neq 0$

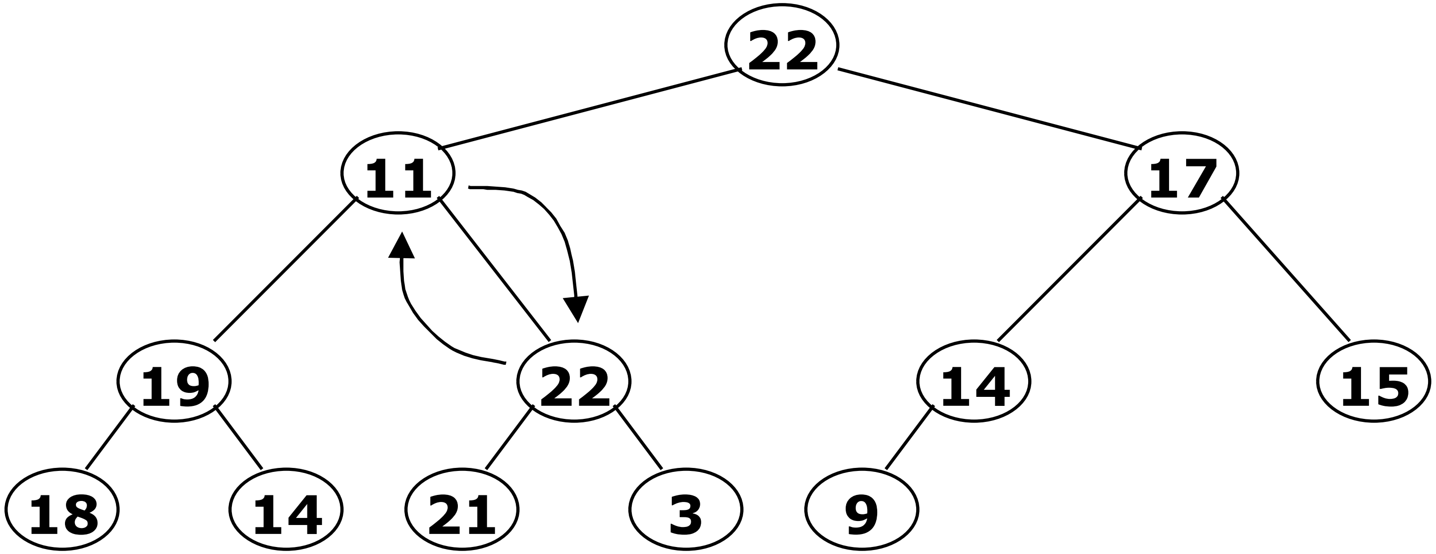
rebuildHeap

- 힙 정렬에서 유용한 연산
- 루트의 왼쪽 부트리와 오른쪽 부트리가 최대힙일 경우, 루트를 포함한 전체 트리를 힙으로 만듬



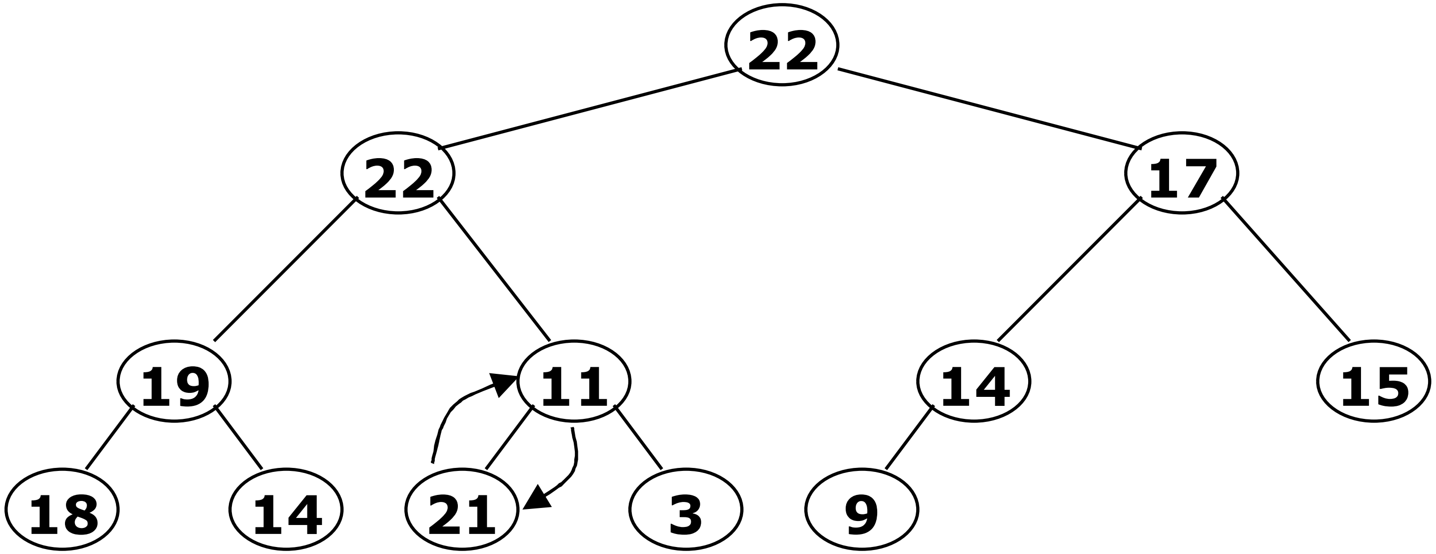
rebuildHeap 예

- Now the left child of the root (still the number 11) lacks the heap property



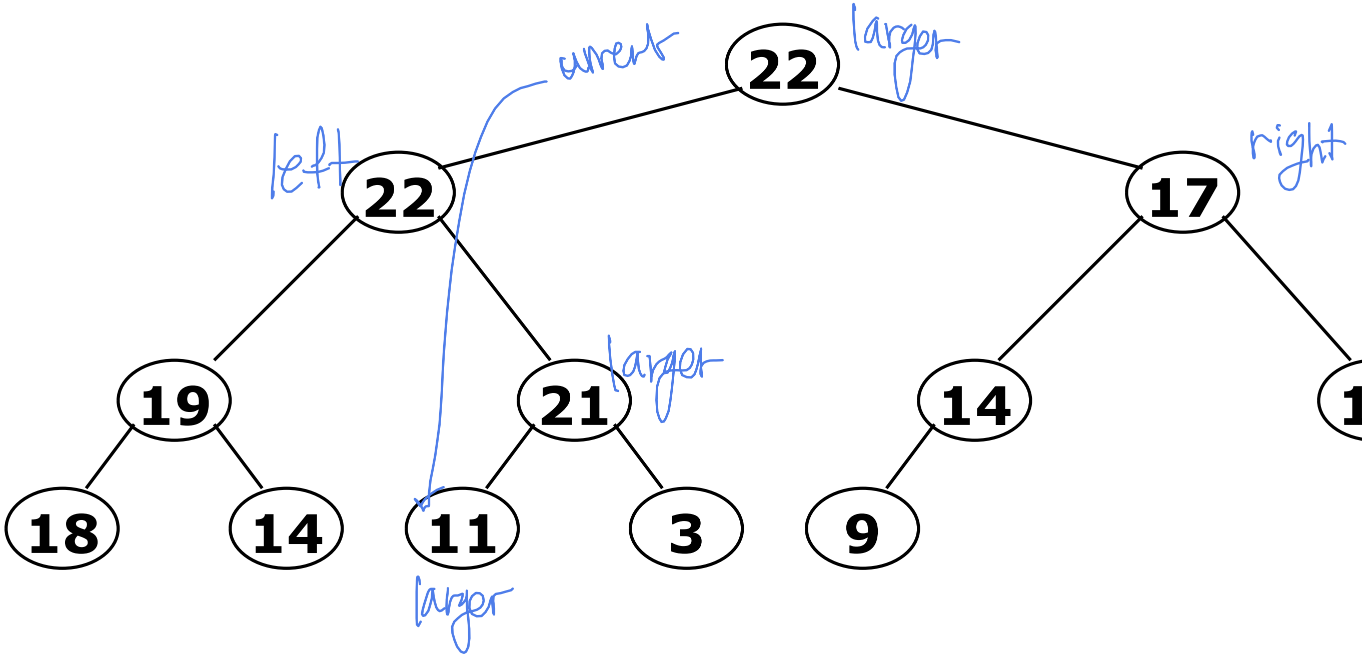
rebuildHeap 예

- Now the right child of the left child of the root (still the number `11`) lacks the heap property:



rebuildHeap 예

- Our tree is once again a heap, because every node in it has the heap property

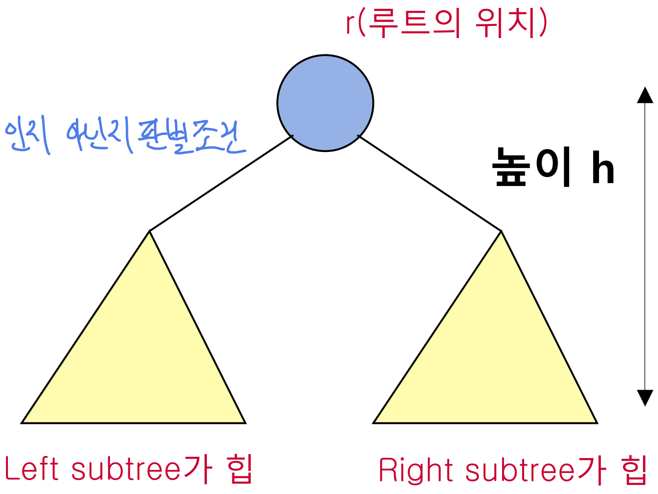


rebuildHeap

```
def rebuildHeap(A, r, n):
# r의 left subtree와 right subtree가 최대힙일 때 루트가 r인 최대힙을 만듬
# r은 리스트에서 root의 위치임
    current = r
    value = A[r] # 루트의 값을 value가 참조(value에 저장)
    while (2*current+1 < n):   # current가 leaf가 아니면
        leftChild = 2*current + 1
        rightChild = leftChild + 1
        # 두 자식 노드 중 큰 값의 노드를 largerChild로 둠
        if rightChild < n and A[rightChild] > A[leftChild]:
            largerChild = rightChild
        else:
            largerChild = leftChild

        if value < A[largerChild]: # largerChild의 값이 크면
            A[current] = A[largerChild]
            current = largerChild     # current를 largerChild로 내림
        else:
            break
    A[current] = value
```

leaf인지 아닌지 판별조건



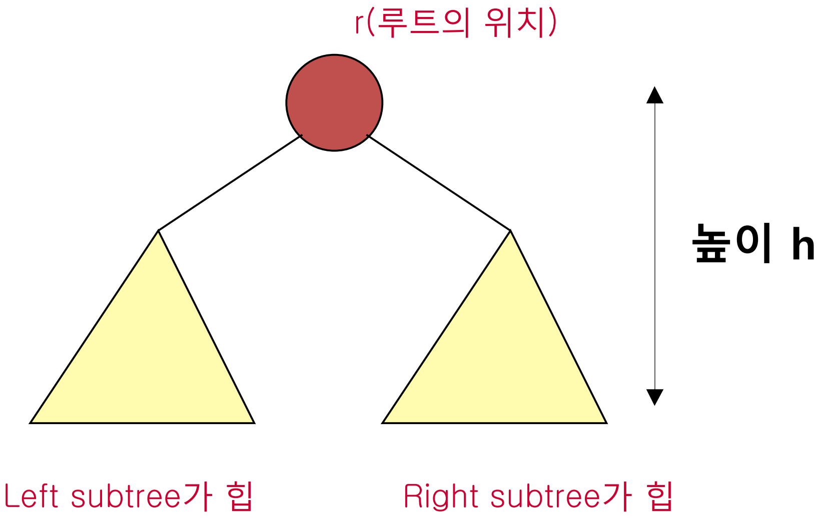
수행시간 : O(h)

rebuildHeap - Maintaining the heap property

rebuildHeap은 root r의 left subtree와 right subtree가 max heap일 때 r까지 포함하여 heap을 만듬

// r은 배열에서 루트의 위치이고, n은 힙의 크기(전체 원소 개수)임

```
void  rebuildHeap(ItemType A[], int r,  int n)
{   int i, largerChild;
    ItemType value = A[r];
    int current = r;
    while (2*current + 1 < n) {
       int  leftChild = 2*current + 1;
       int rightChild = leftChild + 1;
       int largerChild;
       if(rightChild < n && (A[rightChild ] >  A[leftChild]))
           largerChild = rightChild;
       else
           largerChild = leftChild;
       if(value < A[largerChild]) {
           A[current] = A[largerChild];
           current = largerChild;
       }
       else
          break;
    }
    A[current] = value;
}
```



수행시간 : O(h)

Heap Sort

최대힙을 이용한 정렬

단계 1: 배열(리스트) A를 최대 힙으로 만듬 (rebuildHeap을 이용)

단계 2: 단계 1에서 만든 최대힙(배열 A[0..n-1])으로부터 다음 과정을 반복하여 정렬함 (n은 원소 개수)

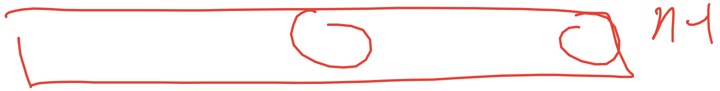
```
heap_size = n
for last = n – 1 downto 1
    // 힙에 있는 최대 원소를 마지막으로 옮긴다.
       A[0]와 A[last]를 교환
    // A[0..last-1]을 힙으로 만든다.
       heap_size -= 1
       rebuildHeap(A, 0, heap_size)
```

Heap Sort

단계 1: 최대 힙을 만듬 (rebuildHeap을 이용)

```
// 마지막 노드의 부모 노드부터 시작하는 rebuildHeap
for root  =  n/2 – 1 ~~to 0 by -1~~  down to 0
    // transform a semiheap with the given root into a heap
        rebuildHeap( A,  root,  n )
```

$$\frac{n-1-1}{2} = \boxed{\frac{n}{2}-1}$$

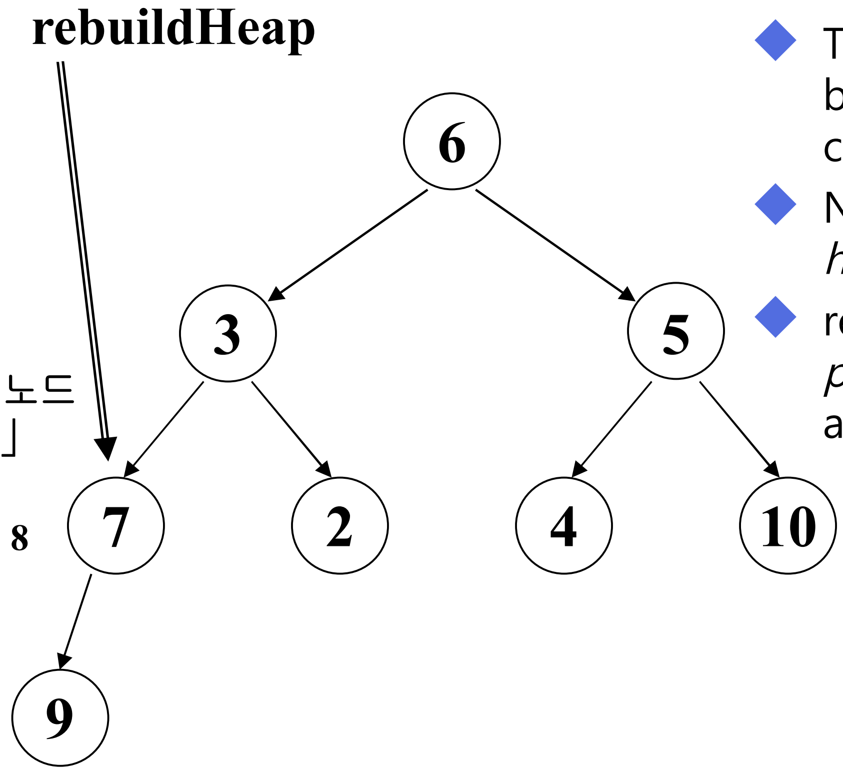


```
# Python
n = len(heap)
for i in range(n//2-1, -1, -1):
    rebuildHeap(A, i, n)
```

```
// C
for (int root = n/2 - 1; root >= 0; root--) {
  rebuildHeap(A,  root,  n);
}
```

단계 1: rebuildHeap을 이용한 최대힙 만들기 - 예

A[ ]:	6	3	5	7	2	4	10	9
	0	1	2	3	4	5	6	7



마지막 노드의 부모 노드 위치: $\lfloor (n-1-1)/2 \rfloor$

$= \lfloor (n-2)/2 \rfloor$

$= \lfloor n/2 \rfloor - 1 = 8$

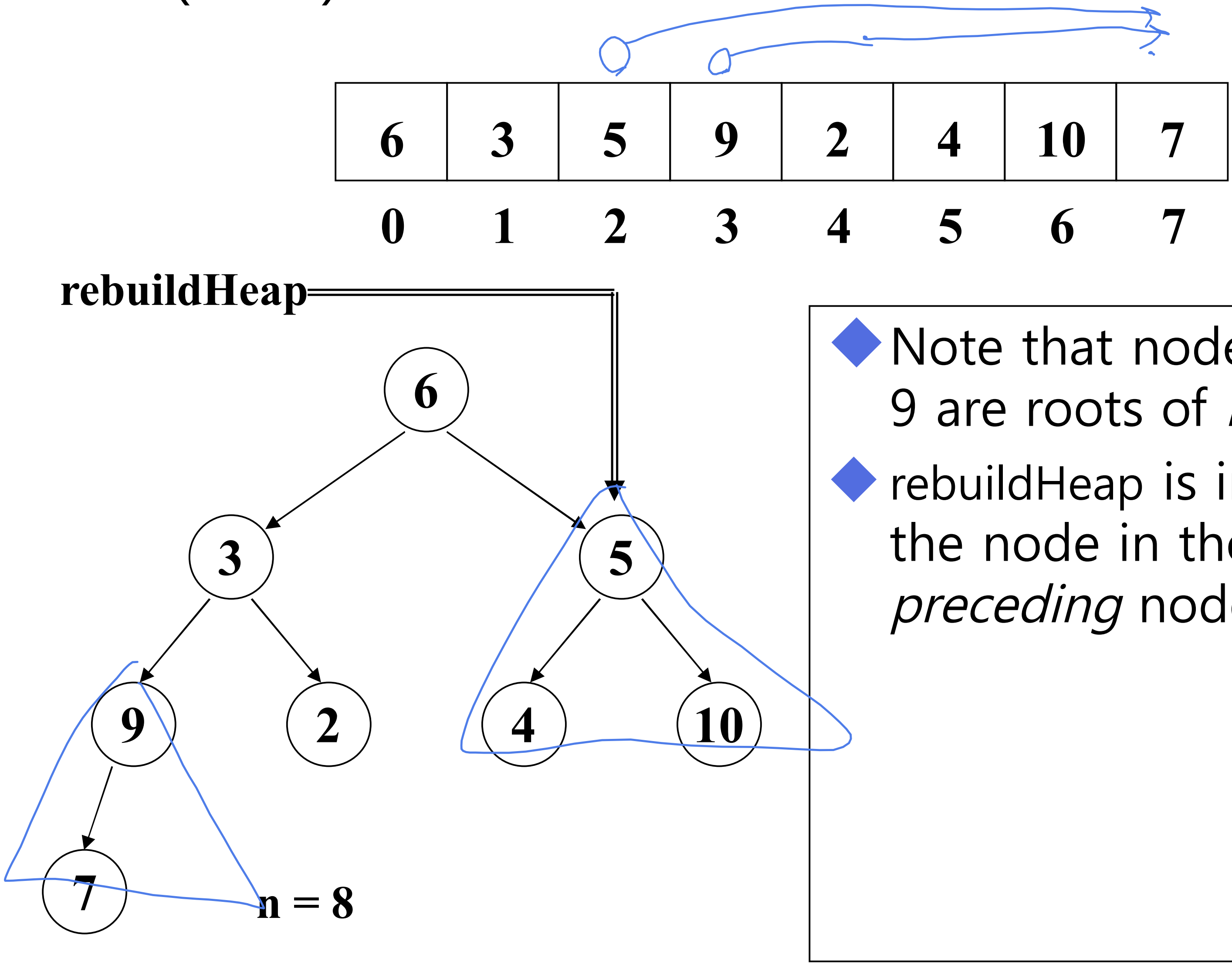
마지막 노드 위치:

n -1 = 8

n = 8

- The items in the array, above, can be considered to be stored in the complete binary tree shown at left.
- Note that leaves 9, 10, 4, and 2 are *heaps.*
- rebuildHeap is invoked on the *parent* of the last node (=9) in the array.

단계 1: rebuildHeap을 이용한 최대힙 만들기 – 예 (계속)

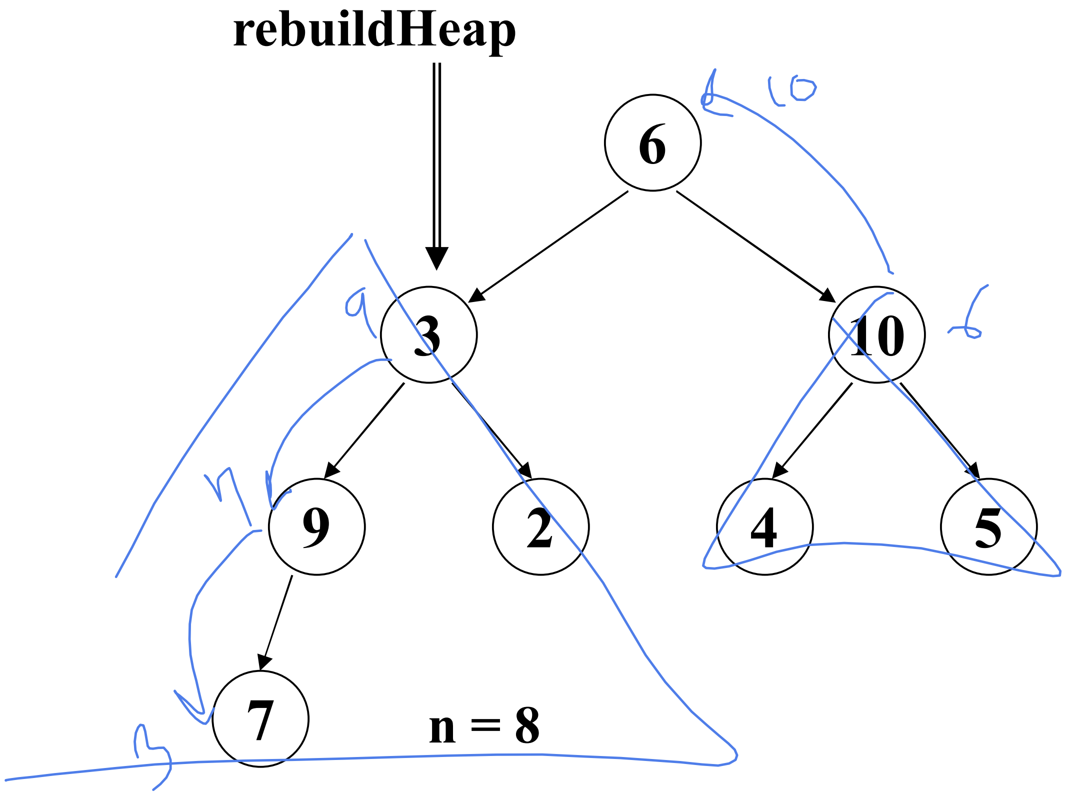


6	3	5	9	2	4	10	7
0	1	2	3	4	5	6	7

- Note that nodes 10, 4, 2, 9 are roots of *heaps*.
- rebuildHeap is invoked on the node in the array *preceding* node 9.

단계 1: rebuildHeap을 이용한 최대힙 만들기 – 예 (계속)

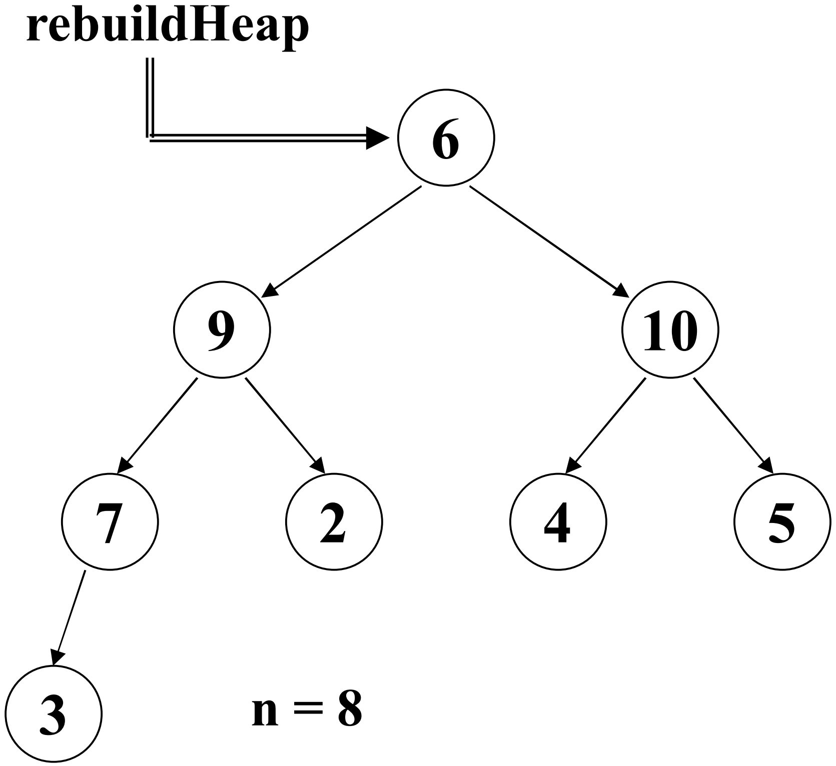
6	3	10	9	2	4	5	7
0	1	2	3	4	5	6	7



- Note that nodes 2, 9, 10 are roots of *heaps*.
- *rebuildHeap* is invoked on the node in the array *preceding* node 10.

단계 1: rebuildHeap을 이용한 최대힙 만들기 – 예 (계속)

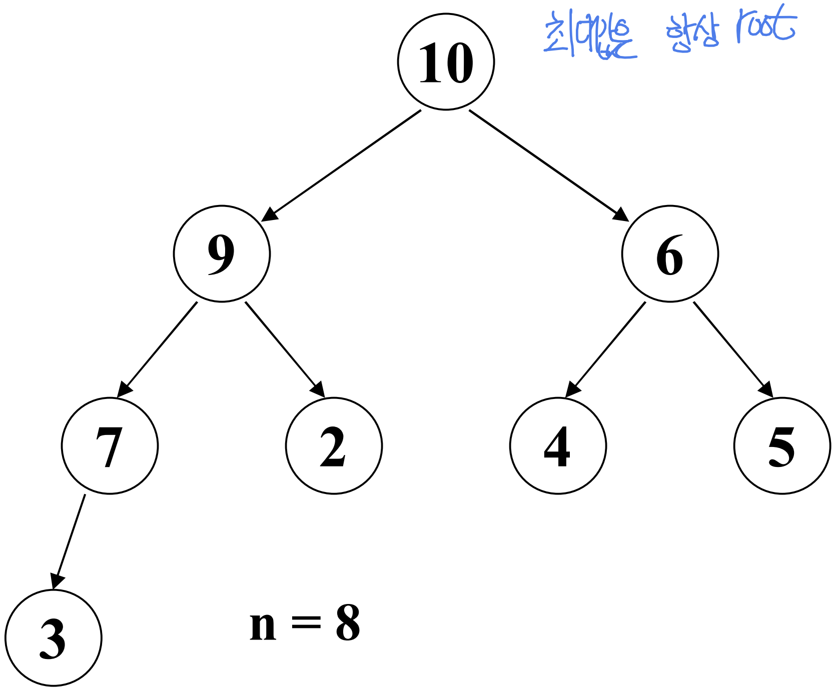
6	**9**	**10**	**7**	**2**	**4**	**5**	**3**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**



- Note that nodes 9, 10 are roots of *heaps.*
- *rebuildHeap* is invoked on the node in the array *preceding* node 9.

단계 1: rebuildHeap을 이용한 최대힙 만들기 – 예 (계속)

10	**9**	**6**	**7**	**2**	**4**	**5**	**3**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**



n = 8

최대값은 항상 root

- Note that node 10 is now the root of a *heap*.
- The transformation of the *array* into a *heap* is complete.

list

주어진 heap을 가지고 rebuild heap을 만드는 과정을 설명하시오.

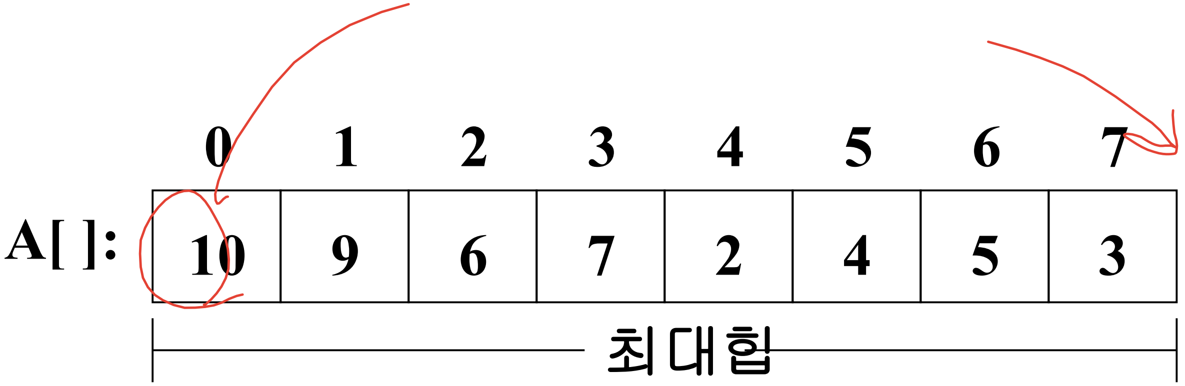
단계 2: 단계 1에서 만든 최대힙(배열 A[0..n-1])으로부터 다음 과정을 반복하여 정렬함 (n은 원소 개수)

```
heap_size = n
for last = n – 1 downto 1
    // 힙에 있는 최대 원소를 마지막으로 옮긴다.
       A[0]와 A[last]를 교환
    // A[0..last-1]을 힙으로 만든다.
       heap_size -= 1
       rebuildHeap(A,  0,  heap_size)
```

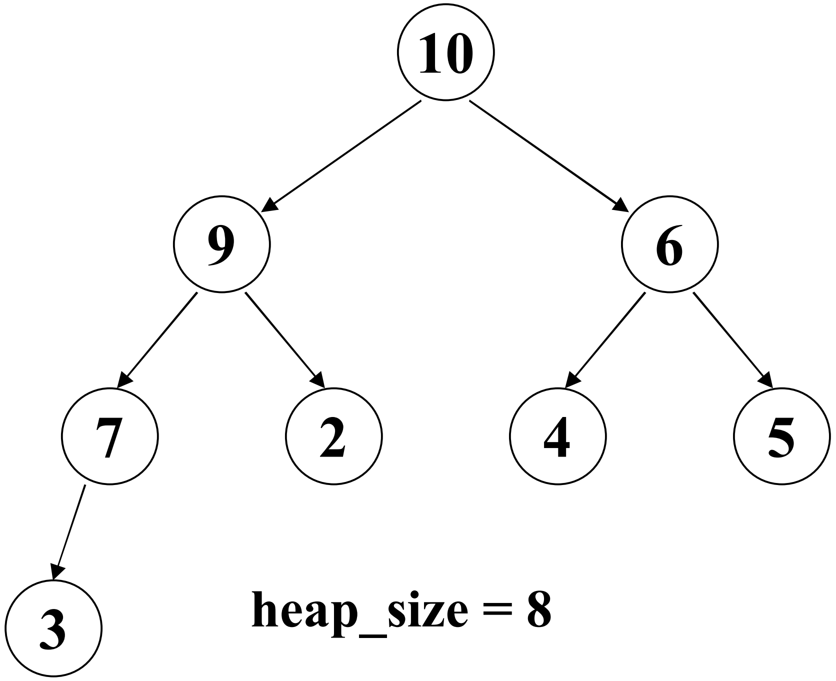
```
# Python
heap_size = n
for last in range(n-1, 0, -1):
    A[0], A[last] = A[last], A[0]
#     temp = A[last]
#     A[last] = A[0]
#     A[0] = temp
    heap_size -= 1
    rebuildHeap(A, 0, heap_size)
```

```
// C
int heap_size = n;
for( int  last = n – 1;  last  >  0;  last--) {
        // move the largest item in the heap, A[0 .. last], to the
        //  beginning of the sorted region, A[last+1 .. n–1], and
        //  increase the sorted region.
        // That is,  swap A[0] and  A[last].
        ItemType temp = A[0];
        A[0] = A[last];
        A[last] = temp;
        // transform the semiheap in A[0 .. last-1] into a heap.
        heap_size--;
        rebuildHeap(A,  0,  heap_size )
}
```

단계 2: 최대힙을 정렬된 리스트(배열)로 만듬 - 예



	0	1	2	3	4	5	6	7
A[ ]:	10	9	6	7	2	4	5	3

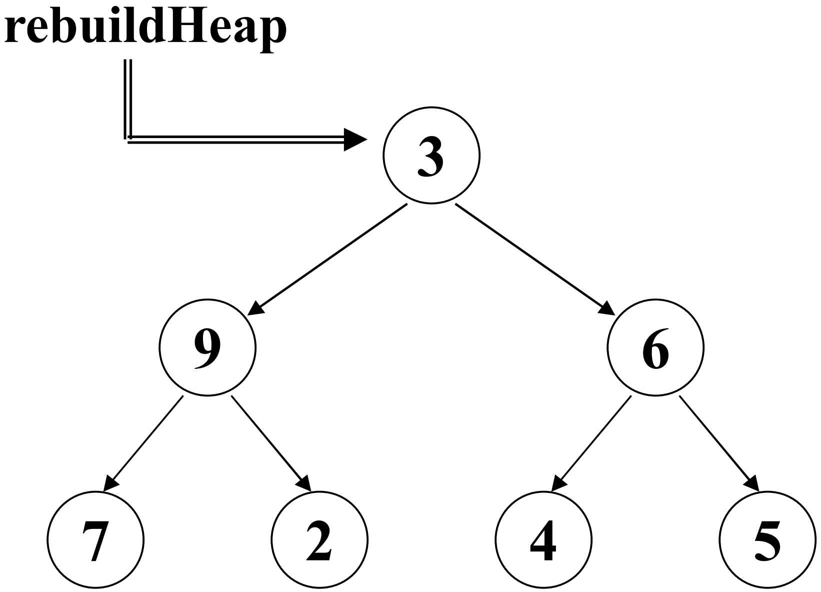


heap_size = 8

- We start with the heap that we formed from an unsorted array.
- The heap is in A[0..7] and the sorted region is empty.
- We move the largest item in the heap to the beginning of the sorted region by swapping A[0] with A[7].

단계 2: 최대힙을 정렬된 리스트(배열)로 만듬 – 예 (계속)

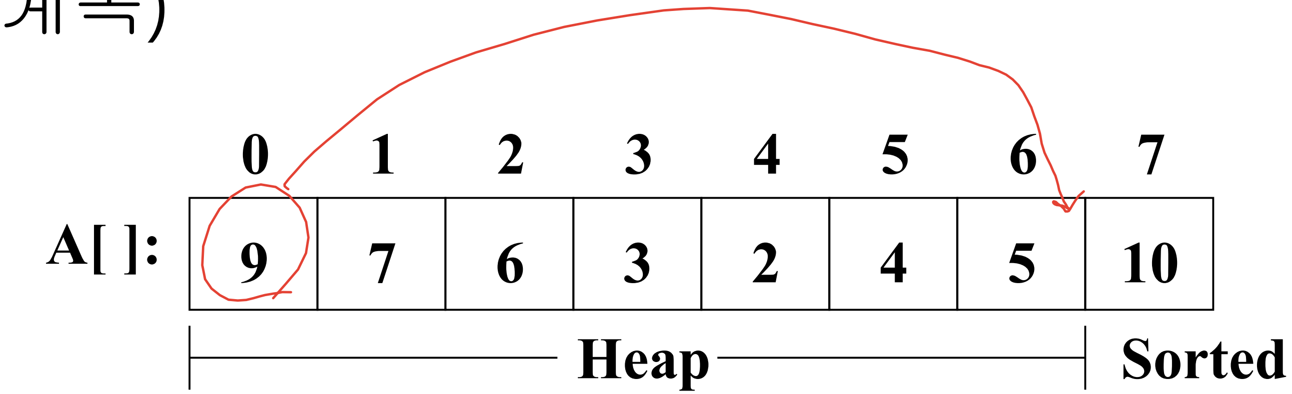
	0	1	2	3	4	5	6	7
A[ ]:	**3**	**9**	**6**	**7**	**2**	**4**	**5**	**10**
	Semiheap							**Sorted**

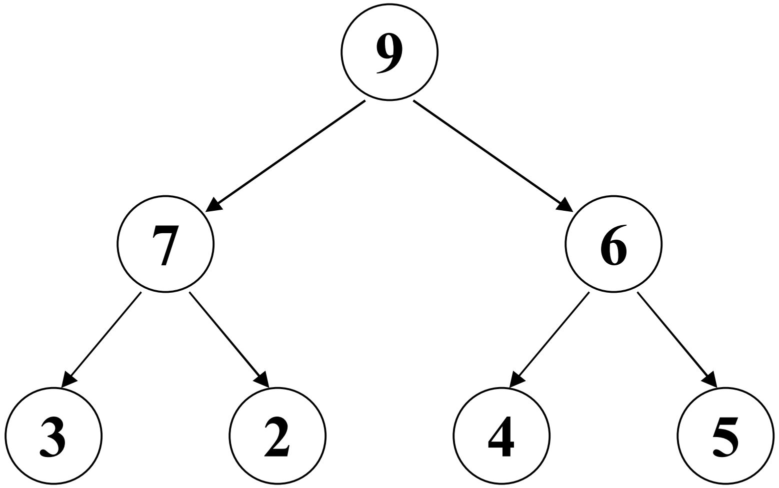


heap_size = 7

- A[0..6] now represents a semiheap.
- A[7] is the sorted region.
- Invoke rebuildHeap on the semiheap rooted at A[0].

단계 2: 최대힙을 정렬된 리스트(배열)로 만듬 – 예 (계속)



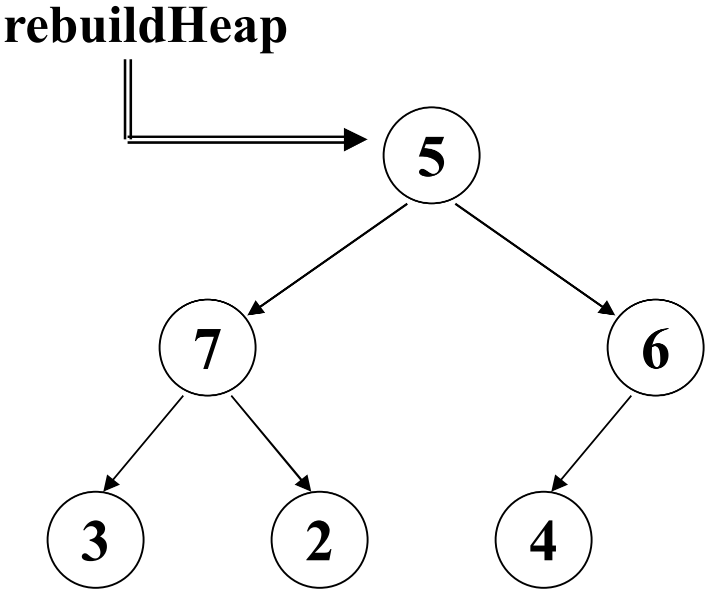


heap_size = 7

- A[0] is now the root of a heap in A[0..6].
- We move the largest item in the heap to the beginning of the sorted region by swapping A[0] with A[6].

단계 2: 최대힙을 정렬된 리스트(배열)로 만듬 – 예 (계속)

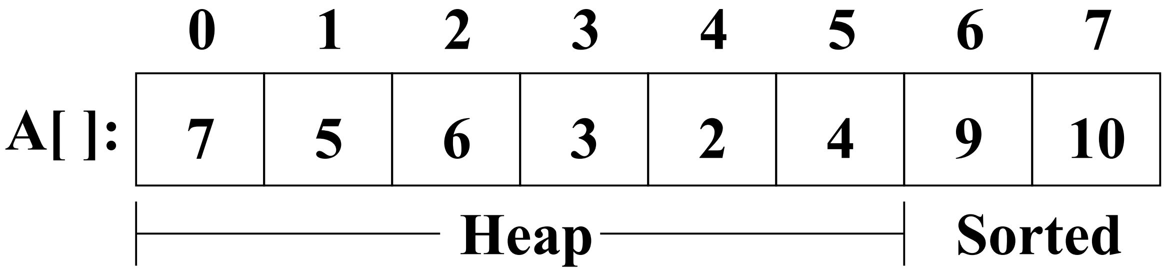
	0	1	2	3	4	5	6	7
A[ ]:	**5**	**7**	**6**	**3**	**2**	**4**	**9**	**10**
	Semiheap						**Sorted**	



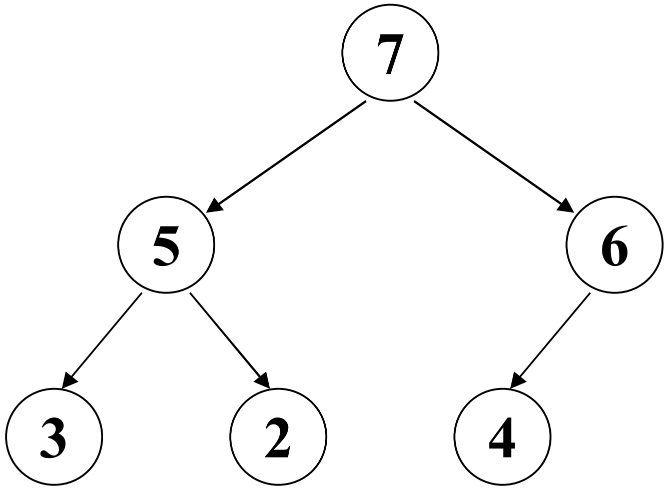
heap_size = 6

- A[0..5] now represents a semiheap.
- A[6..7] is the sorted region.
- Invoke rebuildHeap on the semiheap rooted at A[0].

단계 2: 최대힙을 정렬된 리스트(배열)로 만듬 – 예 (계속)



	0	1	2	3	4	5	6	7
A[ ]:	7	5	6	3	2	4	9	10



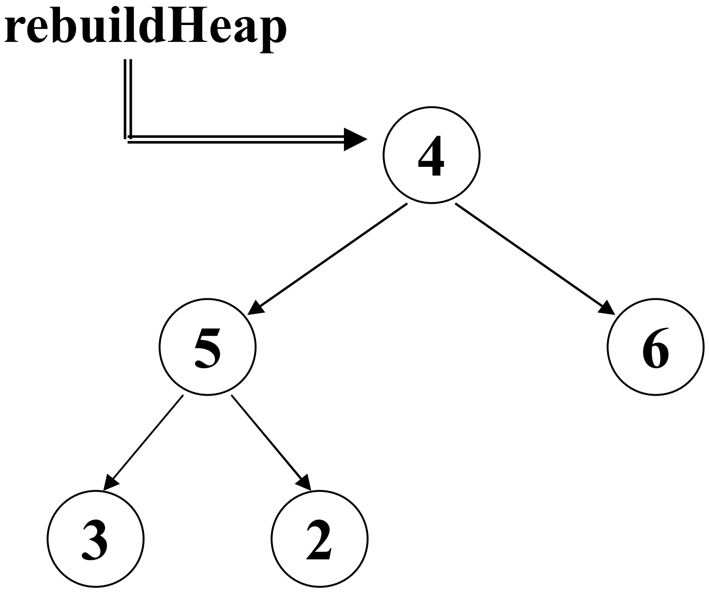
heap_size = 6

- A[0] is now the root of a heap in A[0..5].
- We move the largest item in the heap to the beginning of the sorted region by swapping A[0] with A[5].

단계 2: 최대힙을 정렬된 리스트(배열)로 만듬 – 예 (계속)

	0	1	2	3	4	5	6	7
A[ ]:	**4**	**5**	**6**	**3**	**2**	**7**	**9**	**10**

Semiheap (0–4) **Sorted** (5–7)



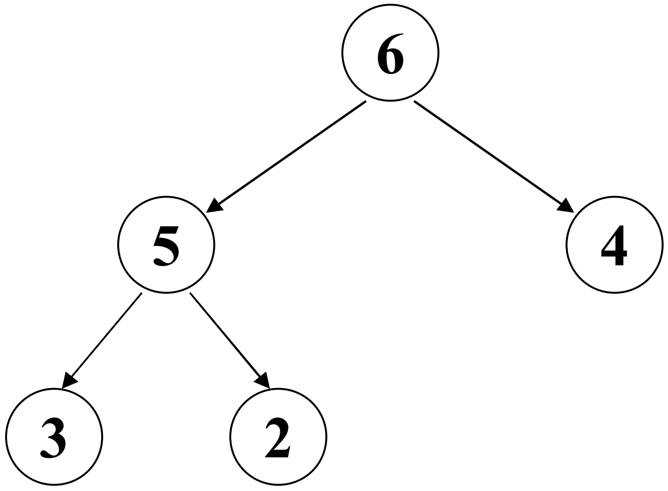
heap_size = 5

- A[0..4] now represents a semiheap.
- A[5..7] is the sorted region.
- Invoke rebuildHeap on the semiheap rooted at A[0].

단계 2: 최대힙을 정렬된 리스트(배열)로 만듬 – 예 (계속)

	0	1	2	3	4	5	6	7
A[ ]:	6	5	4	3	2	7	9	10

Heap (A[0..4]) | **Sorted** (A[5..7])



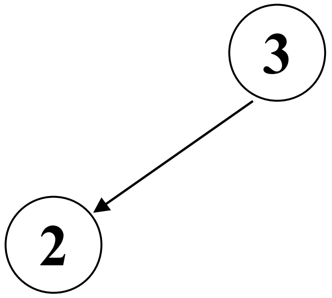
heap_size = 5

- A[0] is now the root of a heap in A[0..4].
- We move the largest item in the heap to the beginning of the sorted region by swapping A[0] with A[4].

단계 2: 최대힙을 정렬된 리스트(배열)로 만듬 – 예 (계속)

	0	1	2	3	4	5	6	7
A[ ]:	**3**	**2**	**4**	**5**	**6**	**7**	**9**	**10**

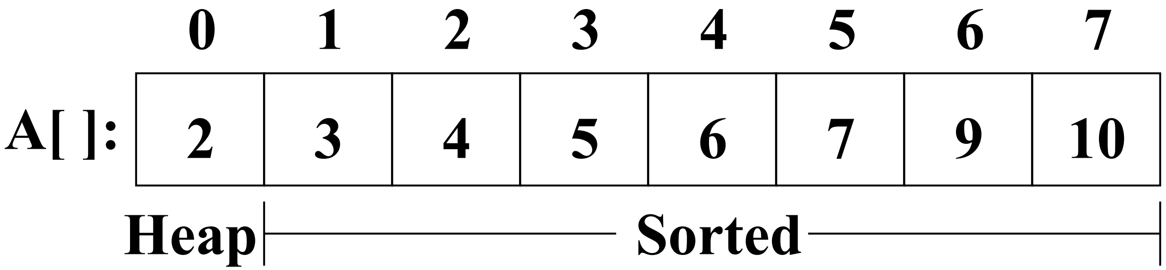
Heap **Sorted**



heap_size = 2

- A[0] is now the root of a heap in A[0..1].
- We move the largest item in the heap to the beginning of the sorted region by swapping A[0] with A[1].

단계 2: 최대힙을 정렬된 리스트(배열)로 만듬 – 예 (계속)



	0	1	2	3	4	5	6	7
A[ ]:	2	3	4	5	6	7	9	10

heap_size = 1

- A[1..7] is the sorted region.
- Since A[0] is a heap, we are done.

단계 2: 최대힙을 정렬된 리스트(배열)로 만듬 – 예 (계속)

	0	1	2	3	4	5	6	7
A[ ]:	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**10**

Sorted

- ◆ Since the sorted region contains all the items in the array, we are done.

Heap sort (in Python)

```
def  heapsort(A)
  // transform A[ ] into a heap
    n = len(A)
    for i in range(n//2-1, -1, -1): // step 1
        rebuildHeap(A,  i,  n);

    heap_size = n
    for last in range(n-1, 0, -1)  // step 2
    # move the largest item in the A[0 .. last], to the
    #  beginning of the sorted region, A[last+1 .. n–1], and
    #  increase the sorted region. That is,  swap A[0] and A[ last ].
        A[0], A[last] = A[last], hA[0]
    # transform the semiheap in A[0 .. Last-1] into a heap.
        heap_size -= 1;
        rebuildHeap(A,  0,  heap_size );
```

Heap sort (in C)

```
void  heapsort( ItemType  A[ ],  int  n )
{  // transform array A[ ] into a heap
    for( int  root  =  n/2 – 1;   root  >=  0;   root – – ) // step 1
        rebuildHeap(A,  root,  n);

    int heap_size = n;
    for( int  last = n – 1;  last  >  0;  last--)  { // step 2
        // move the largest item in the A[0 .. last], to the
        //  beginning of the sorted region, A[last+1 .. n–1], and
        //  increase the sorted region.
        // That is,  swap A[0] and A[last].
        ItemType temp = A[0];
        A[0] = A[last];
        A[last] = temp;
        // transform the semiheap in A[0 .. Last-1] into a heap.
        heap_size--;
        rebuildHeap(A,  0,  heap_size);
    }
}
```

Heap Sort의 수행시간 분석

단계1 n/2 호출

rebuild log n

n log n

단계2는 n-1번 호출.

rebuild log n

n log n

◆ 단계 1의 수행시간

- RebuildHeap takes *O*(*h*) time where *h* is the height of the subtree rooted at node where rebuildHeap is applied
- *rebuildHeap( )* is invoked $\lfloor n / 2 \rfloor$ times
- The running time of step 1 is O(*n* lg *n*) : This is not tight bound

Tight bound on the running time of step 1 is O(*n*)

Heap Sort의 수행시간 분석 (계속)

◆ 단계 1의 tight한 수행시간 분석

- The level of a node is its distance from the root
- h (the height of a heap A with n nodes) = ⌊lg n⌋

$$\text{running time} = \sum_{i=0}^{h} \#node(i) \cdot O(h-i)$$

where $\#node(i)$ is the number of nodes at level i and $h = height(A)$.

#node(i) ≤ 2^{i-1}
h = ⌊lg n⌋

$$\sum_{i=1}^{h-1} 2^{i-1} \cdot O(h-i) = O\left(\sum_{i=1}^{h-1} 2^{i-1}(h-i)\right) = O(2^{h-1}\sum_{i=1}^{h-1}\frac{(h-i)}{2^{h-i}}) = O(n)$$

because

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

$$2^{h\text{-}1} = O(n).$$

Heap Sort의 수행시간 분석 (계속)

- 단계 2의 수행 시간 분석
  - rebuildHeap(A, 0, heap_size) takes *O*(log n) time
  - rebuildHeap(A, 0, hezp_size) is invoked (*n*-*1*) times
  - step 2의 수행시간은 O(*n* log *n*)

단계 1 + 단계 2: heap sort의 수행시간은 O(n log n)이다.

- The time complexity of heap sort in the *best*, *average* and *worst* cases is O(n log n)

- Knuth's analysis shows that, in the *average case*, *Heapsort* requires about twice as much time as *Quicksort*

우선순위 큐(Priority Queues)

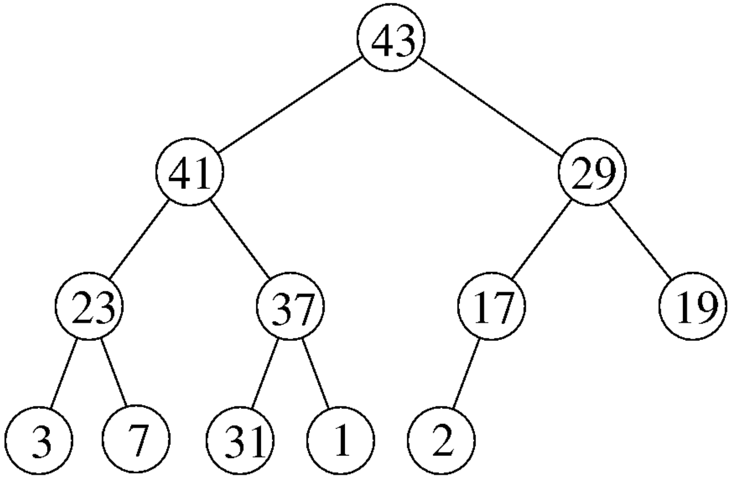
자료구조 시간 때 했으므로
넘어감

- *Priority queue is* popular & important application of heaps
  - A data structure for maintaining a dynamic set *S* of elements, each with an associated value *key*
  - Supports the operations **insert**, **findMax**, and **deleteMax efficiently.**
    - insert(*S*, *x*) - inserts the element *x* into the set *S*
      - $S \leftarrow S \cup \{x\}$.
    - findMax (*S*) - returns the element of *S* with the largest key.
    - deleteMax(*S*) - removes and returns the element of *S* with the largest key.

- Applications:
  - Ready list of processes in operating systems by their priorities – the list is highly dynamic
  - In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence.

Insertion Operation

- Place item to be inserted into leftmost open array slot
- If item is greater than parent, swap and, and repeat it.
- Number of comparisons in the worst case?

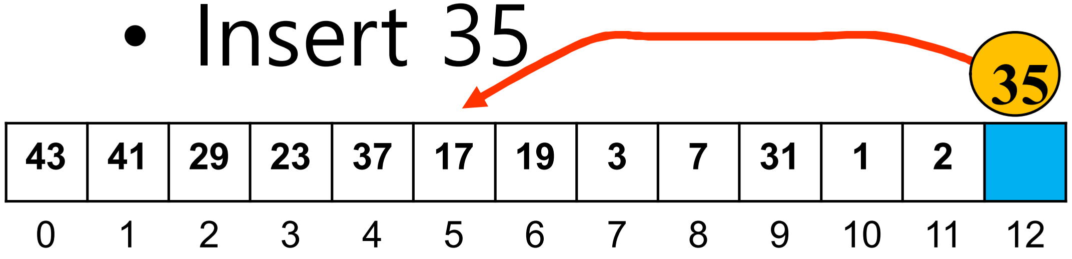


43	**41**	**29**	**23**	**37**	**17**	**19**	**3**	**7**	**31**	**1**	**2**	
0	1	2	3	4	5	6	7	8	9	10	11	12

n (heap_size)= 12

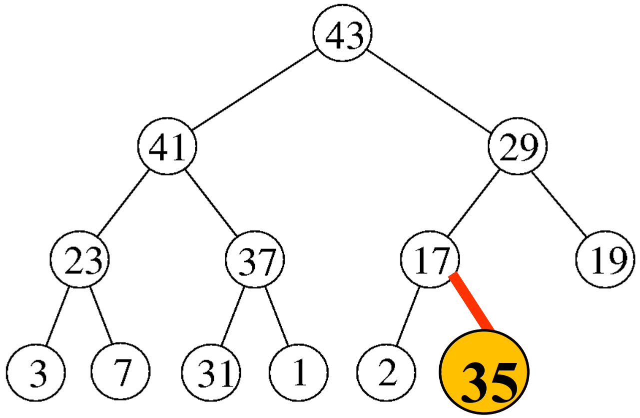
Insertion Operation - Example

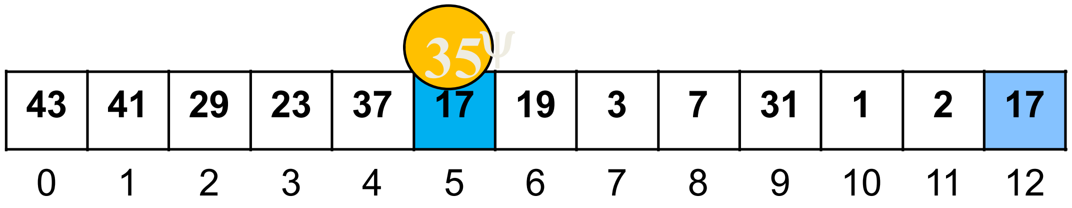
- Insert 35



43	41	29	23	37	17	19	3	7	31	1	2	
0	1	2	3	4	5	6	7	8	9	10	11	12

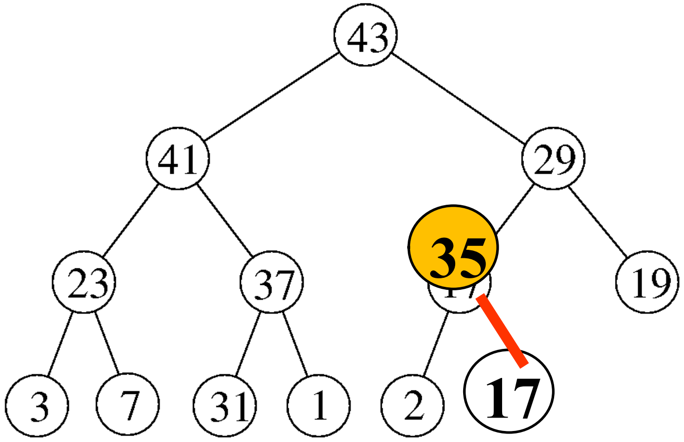
n = 13





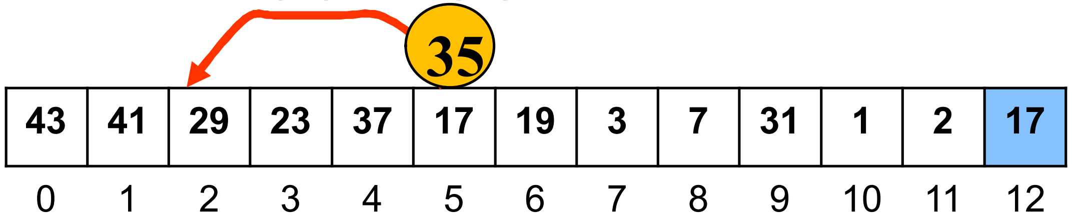
43	41	29	23	37	17	19	3	7	31	1	2	17
0	1	2	3	4	5	6	7	8	9	10	11	12

n = 13



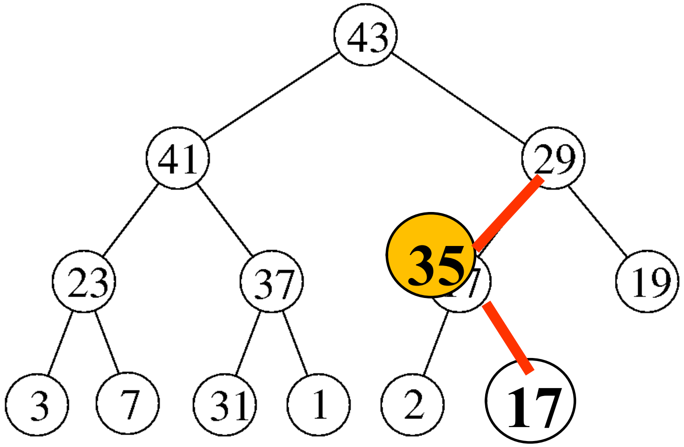
Insertion Operation – Example (Cont'd)

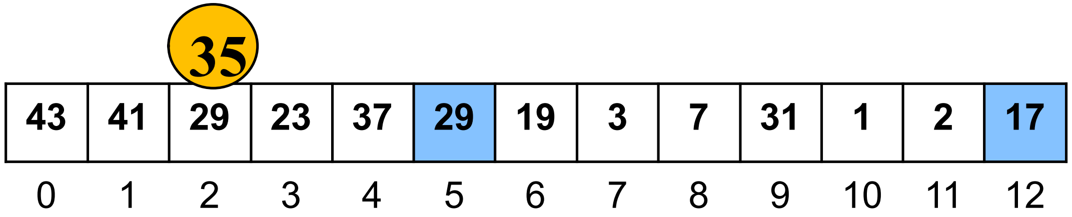
- Insert 35



43	**41**	**29**	**23**	**37**	**17**	**19**	**3**	**7**	**31**	**1**	**2**	**17**
0	1	2	3	4	5	6	7	8	9	10	11	12

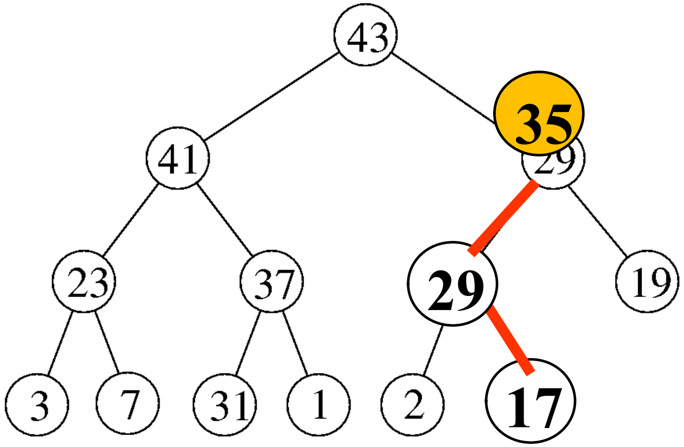
n = 13





43	**41**	**29**	**23**	**37**	**29**	**19**	**3**	**7**	**31**	**1**	**2**	**17**
0	1	2	3	4	5	6	7	8	9	10	11	12

n = 13



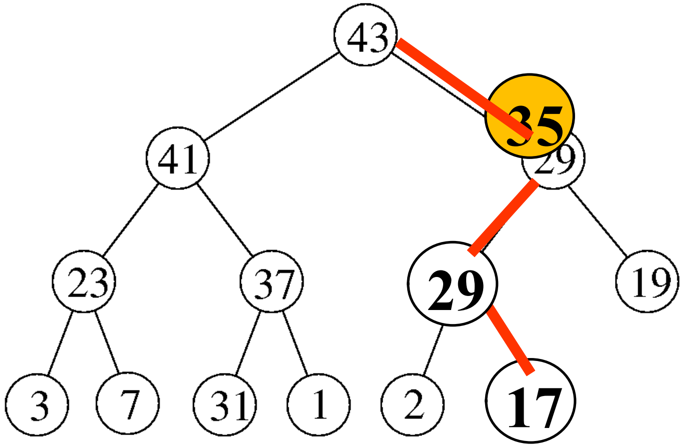
Insertion Operation – Example (Cont'd)

- Insert 35

35

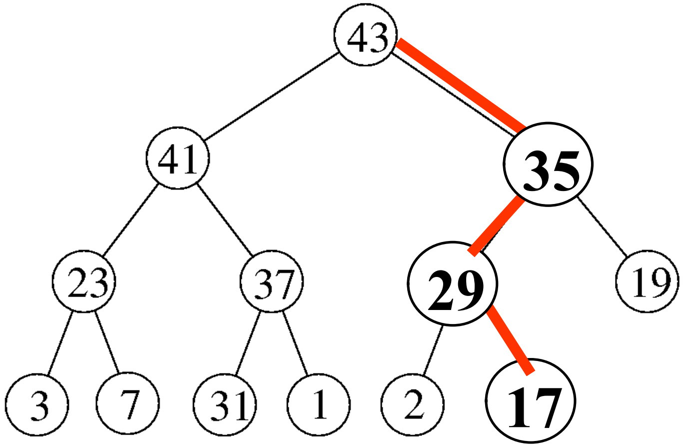
43	41	29	23	37	29	19	3	7	31	1	2	17
0	1	2	3	4	5	6	7	8	9	10	11	12

n = 13



43	41	35	23	37	29	19	3	7	31	1	2	17
0	1	2	3	4	5	6	7	8	9	10	11	12

n = 13



insert(A, key) // A is a heap

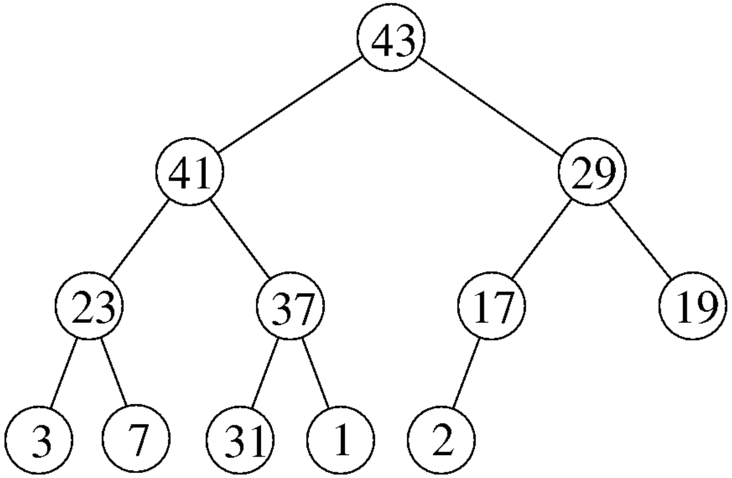
```
Algorithm insert(A, key)
// n is heap size
n ← n + 1
i ← n - 1
while i > 0 and A[Parent(i)] < key
    L[i] ← L[Parent(i)]
    i ← Parent(i)
L[i] ← key
```

Running time is $O(\log n)$

The path traced from the new leaf to the root has length $O(\log n)$

deleteMax Operation

- Copy root value to be returned
- Move rightmost entry to root
- Perform **rebuldHeap** to fix up heap



43	**41**	**29**	**23**	**37**	**17**	**19**	**3**	**7**	**31**	**1**	**2**	
0	1	2	3	4	5	6	7	8	9	10	11	12

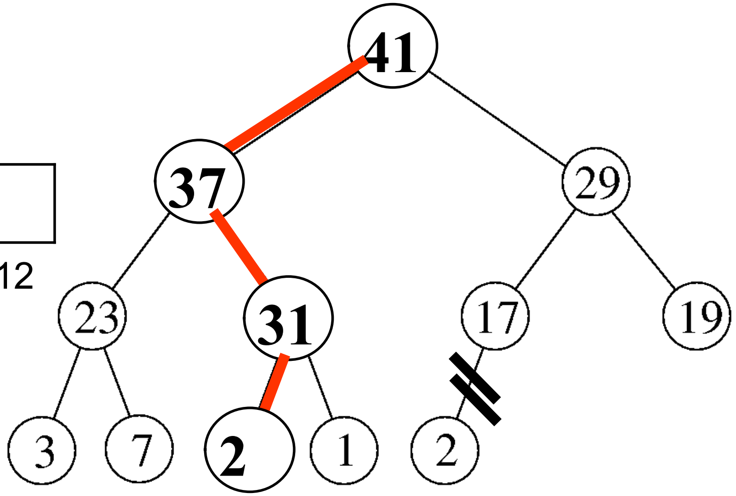
n (heap_size) = 12

deleteMax Operation – 예 (계속)

- max = 43

41	37	29	23	31	17	19	3	7	2	1		
0	1	2	3	4	5	6	7	8	9	10	11	12

n = 11



deleteMax(A)

Implements the deleteMax operation.

```
Algorithm deleteMax(A)
// n is the heap size
if n < 1
    then error “heap underflow”
maximum $\leftarrow$ A[0]
A[0] $\leftarrow$ A[n-1]
n $\leftarrow$ n - 1
rebuildHeap (A, 0, n)
return maximum
```

수행시간 : **Dominated by the running time of rebuildHeap**

= $O(\log n)$

파이썬 heapq

파이썬은 우선순위큐를 위한 heapq를 라이브러리로 제공

연산들: insert, deleteMin, …

import heapq

heapq에 선언된 메소드

- **heapq.heappush(*heap, item*)** # insert() 메소드와 동일
- **heapq.heappop(*heap*)** # deleteMin() 메소드와 동일
- **heapq.heappushpop(*heap, item*)** # item 삽입 후 deleteMin() 수행
- **heapq.heapify(*h*)** # 리스트 h를 힙으로 만듬

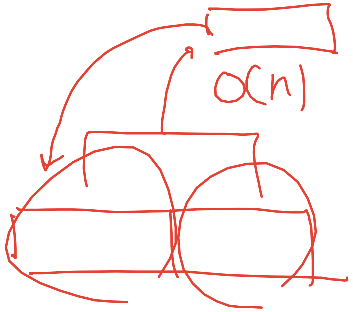
C++ STL priority_queue

C++는 우선순위큐를 위한 STL (Standard Template Library)
prioity_queue를 제공

정렬 알고리즘들의 비교

정렬하는데 사용되는 메모리 공간 (입력받은 리스트안에 정렬)
추가적으로 사용되는 것이 상수임.

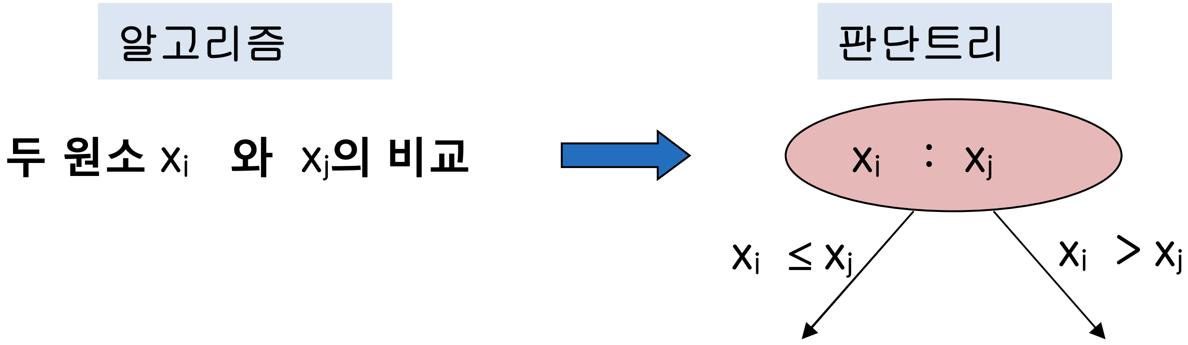
알고리즘	Worst Case	Average case	In place 여부	Stable 여부
Insertion Sort (Bubble Sort)	$\Theta(n^2)$	$\Theta(n^2)$	In place $O(1)$	○
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$	recursion을 쓰므로 스택 메모리 씀. 스택 메모리 $O(\log n)$	X
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(n)$ – for merging	○
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	In place $O(1)$	X



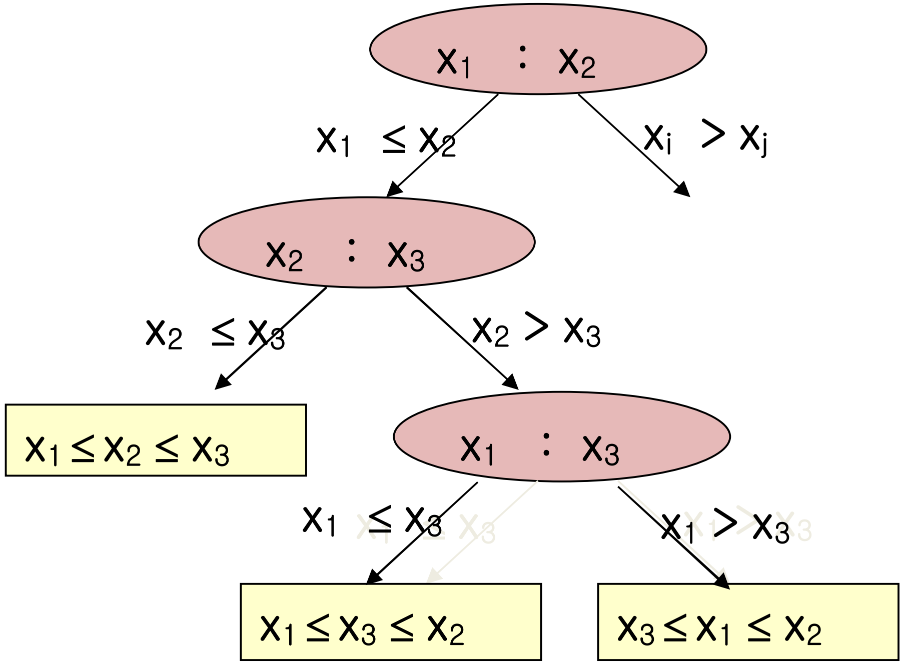
- Merge Sort나 Heap Sort가 최악의 경우 최적인 정렬 알고리즘인가? 즉, 최악의 경우 Merge Sort나 Heap Sort보다 저 좋은 알고리즘이 있는가?

정렬 알고리즘의 하한계 (lower bound)

- 키 비교에 의한 정렬알고리즘은 판단트리(decision tree)로 나타낼 수 있다.



- 삽입정렬의 판단트리 표현 예: n = 3, (x_1, x_2, x_3)



정렬 알고리즘의 하한계 (lower bound)

- 정리: n개의 자료(원소)를 키 비교에 의하여 정렬하는 알고리즘은 최악의 경우 적어도 $\lceil \log_2 n! \rceil$ ($\approx \lceil n \log_2 n - 1.443n \rceil$)의 키 비교를 한다.

(증명) A: 키 비교에 의한 임의의 정렬알고리즘

T: A를 표현한 판단트리

T는 n!개의 leaf를 가진다.

A의 최악의 경우 비교횟수 = T의 높이

T의 높이 $\geq \lceil \log_2 n! \rceil \geq \lceil \log_2 (1 \times 2 \times 3 \cdots \times n) \rceil$ ($\approx \lceil n \lg n - 1.443n \rceil$)

※ T의 높이 $\geq \log_2 (n/2 \times n/2 \times \cdots \times n/2)$

n/2을 n/2개 곱한 것

T의 높이 $\geq n/2 \log_2 n/2$

따라서 T의 높이 = $\Omega(n \log n)$

→ 키정렬 알고리즘은 n!개 결과를 낼 수 있다

이진트리는 높이가 h일때 최대 2^h의 리프 노드를 가질 수 있다.

즉, 2^h >= n!을 만족해야한다. 양변 로그 h>= log n! 이다.

최악의 경우 정렬 알고리즘은 h번 비교하므로 최소 log n!번 비교 연산함.

$h \geq \log n! > \log \left\{ \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \right\} > \log \left(\frac{n}{e}\right)^n$

$\Rightarrow h > \log \left(\frac{n}{e}\right)^n$

$h > O(n \log n)$

- 키 비교에 의한 정렬알고리즘의 최악의 경우 시간복잡도는 $\Omega(n \log n)$이다 ⇒ 키 비교에 의한 정렬알고리즘으로 Merge Sort나 Heap Sort는 최악의 경우 최적인 알고리즘이다.

↗ $O(n \log n)$보다 효율적인 최악의 경우 키 비교 알고리즘은 있을수 X