# COMP319 Algorithms 1
# Lecture 8
# Heap

Instructor: Gil-Jin Jang

Max-Min Heapify

Heap Building

Heap Sort

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

Max heap and min heap

Heapify operations

Build heaps

# HEAP

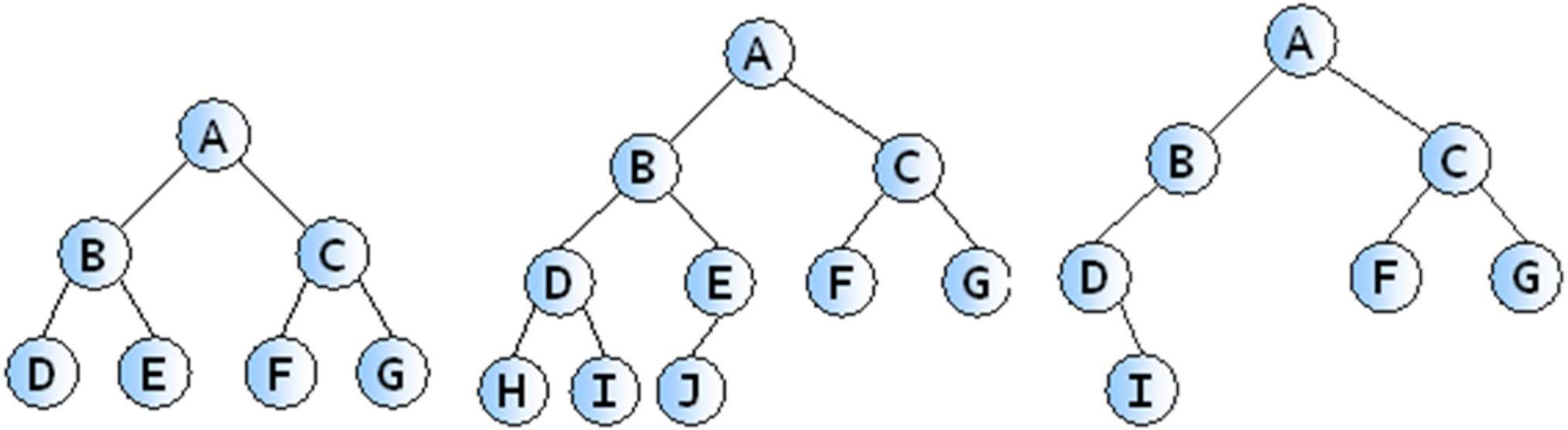
# Review: Comparing Sorting Methods

- Insertion/selection/bubble sort
  - Advantages: using less extra memory
  - Disadvantages: $T(n) = T(n\text{-}1)+cn \rightarrow O(n\text{^}2)$
- Merge sort
  - Advantages: $T(n) = 2T(n/2)+cn \rightarrow O(n \lg n)$
  - Disadvantages: extra memory of $O(n)$
- Quicksort
  - $O(n \lg n)$ without extra memory
  - Disadvantages: in worst case, $O(n\text{^}2)$
- *Heapsort*
  - Combines advantages of the previous algorithms

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Review: 이진 트리의 분류

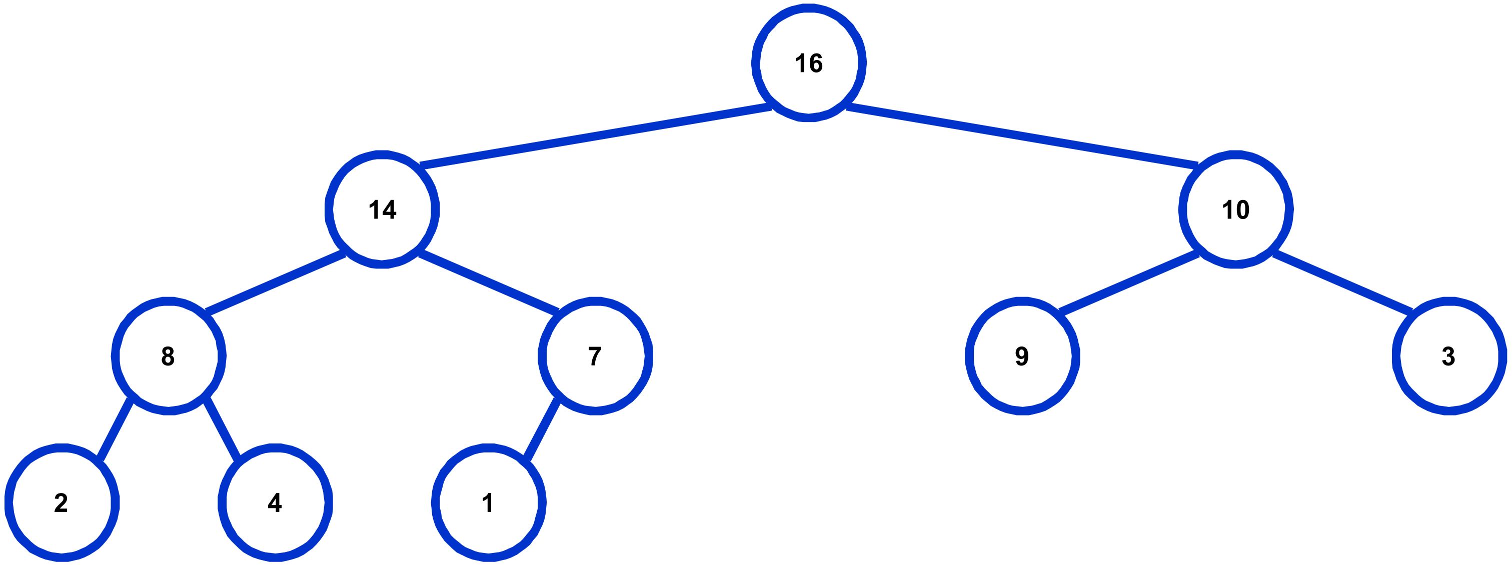
- 포화 이진 트리(full binary tree)
- 완전 이진 트리(complete binary tree)
- 기타 이진 트리




(a) 포화이진트리 (b) 완전이진트리 (c) 기타이진트리

# Heaps as Binary Trees

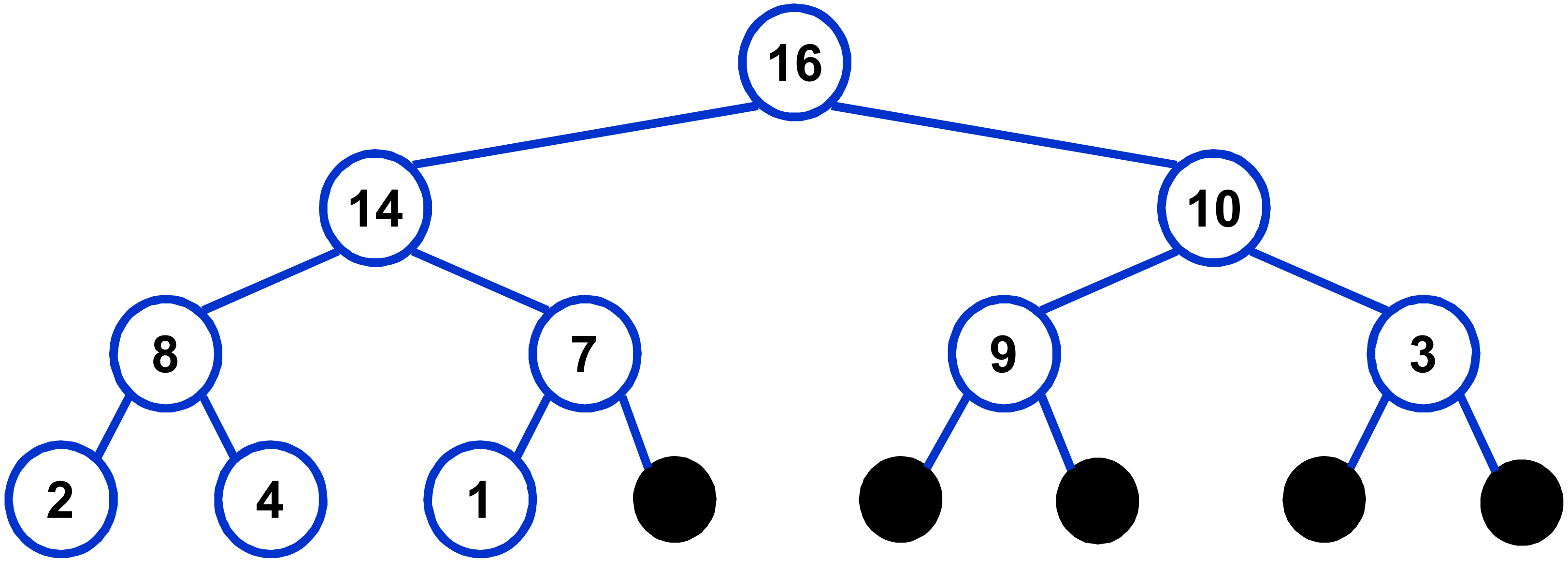
- A *heap* can be seen as a complete binary tree:
  - *What makes a binary tree complete?*
  - *Is the example below complete?*




*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Heaps as Complete Binary Trees

- A heap can be seen as a complete binary tree:
  - Or as **NEARLY FULL** binary trees
    - Unfilled slots are represented as **NULL** pointers (filling dummy values in)




*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Heap Implementation as Arrays

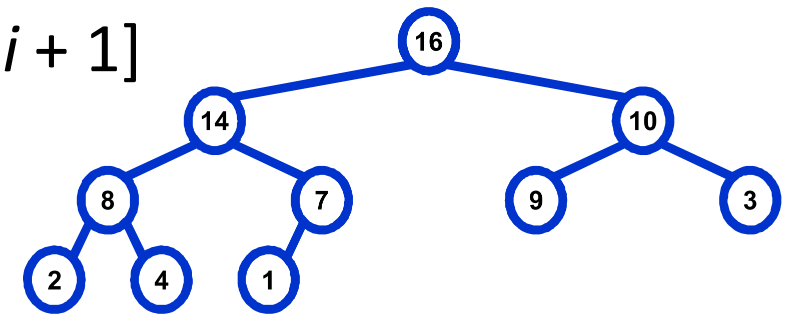
- In practice, heaps (*complete binary trees*) are usually implemented as arrays:
  - The root node is A[1] (*note: not A[0]*)
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2]
    - note: integer division, quotient only
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

*A* =

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| - | *16* | *14* | *10* | *8* | *7* | *9* | *3* | *2* | *4* | *1* |

=

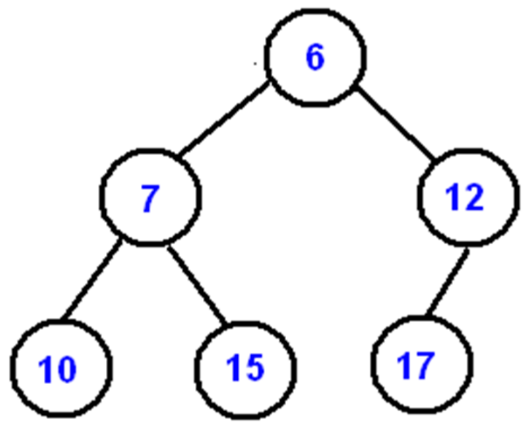



*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# The Heap Ordering Properties

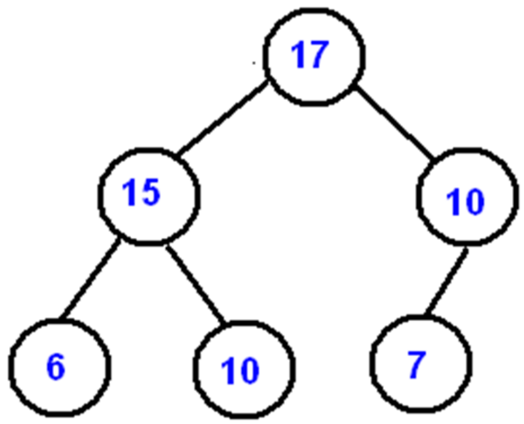
- *min-heap*:
  - the value of each node is greater than or equal to the value of its parent, resulting in minimum-value at the root.
  - 각 노드의 값은 자신의 children의 값보다 크지 않다

  A[*Parent*(*i*)] ≤ A[*i*] for all nodes *i* > 1




- *max-heap*:
  - the value of each node is less than or equal to the value of its parent, resulting in maximum-value at the root.
  - 각 노드의 값은 자신의 children의 값보다 작지 않다

  A[*Parent*(*i*)] ≥ A[*i*] for all nodes *i* > 1




Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

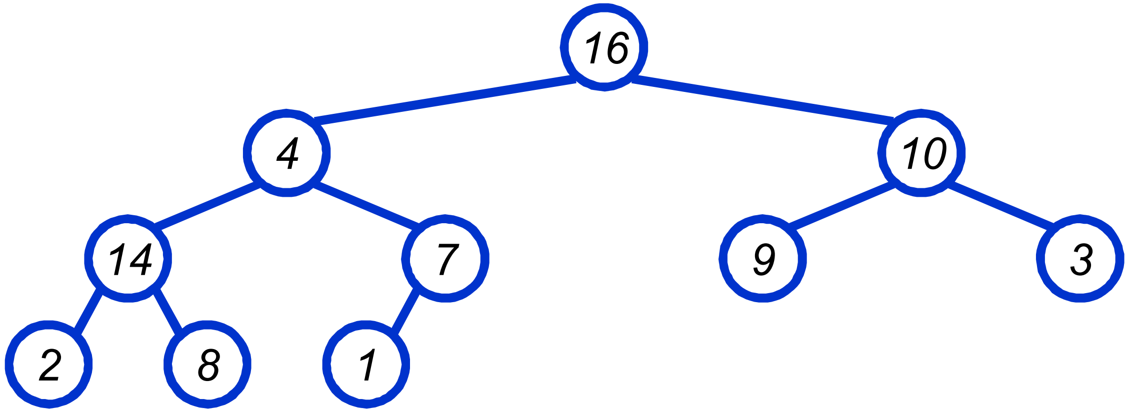
# Heap Height

- Definition of HEAP HEIGHT:
  - The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
  - The height of a tree = the height of its root
- *What is the height of an n-element heap?*
  - **`Ceiling(log2(n)): a smallest integer greater than log2(n)`**
- Basic heap operations take at most time proportional to the height of the heap

# Heap Operations: MaxHeapify()

- **`MaxHeapify()`**: to keep the max-heap property
  - Inputs: Array A (or binary tree), index *i* in array
  - Precondition (input): the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps
    - Note: A[*i*] may be smaller than its children, A[*i**2] and A[*i**2+1] are larger than or equal to ALL OF THEIR CHILDREN
  - Postcondition (output): The subtree rooted at index *i* is a max-heap
    - A[*i*] is larger than or equal to ALL OF ITS CHILDREN
  - Action: let the value of the parent node FLOAT-DOWN so subtree at *i* satisfies the max-heap property

# MaxHeapify() Example

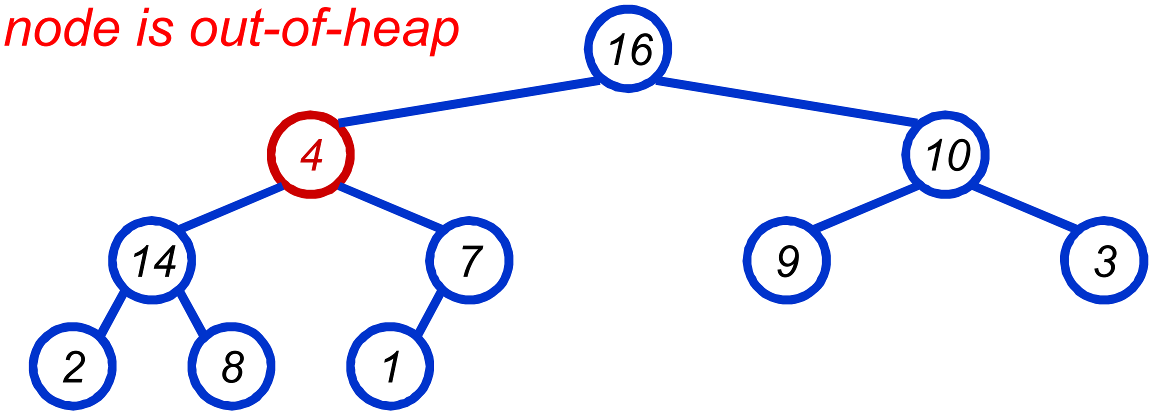



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *A* = | *16* | *4* | *10* | *14* | *7* | *9* | *3* | *2* | *8* | *1* |

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# MaxHeapify() Example

*Only a single node is out-of-heap*

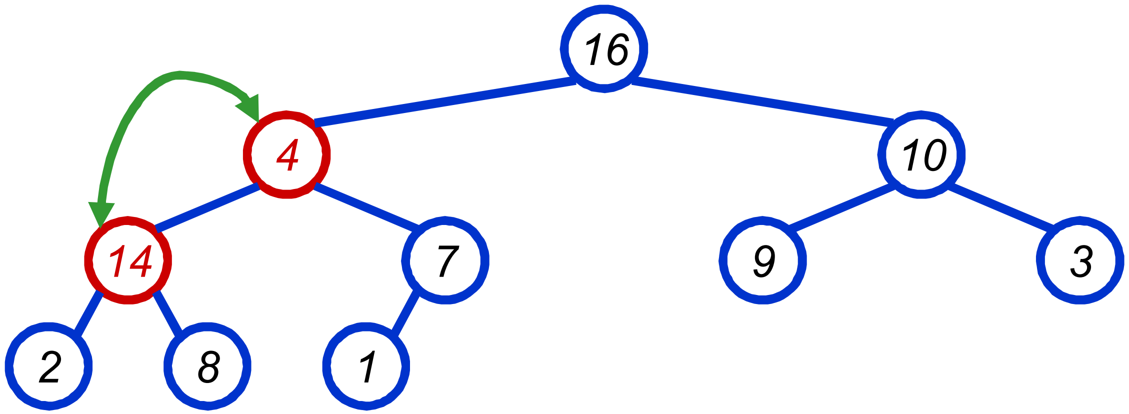



A =

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 4 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |

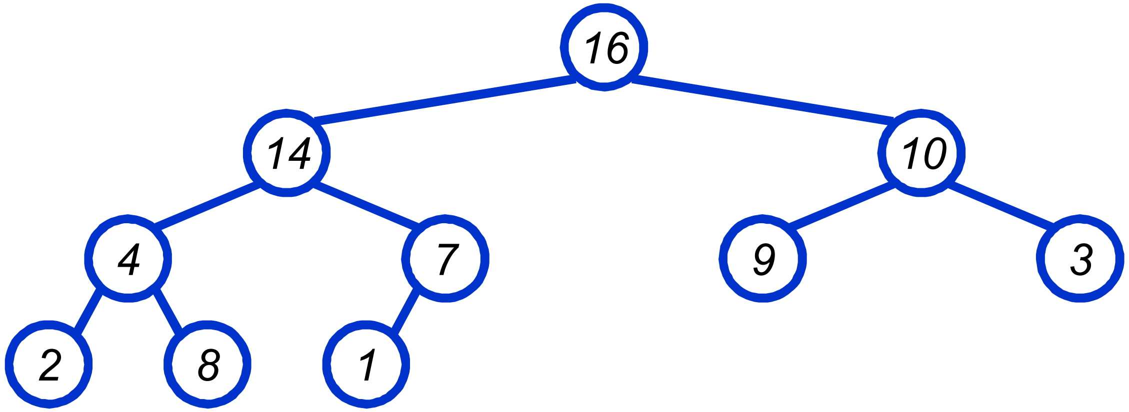
*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# MaxHeapify() Example




| A = | 16 | 4 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|

# MaxHeapify() Example

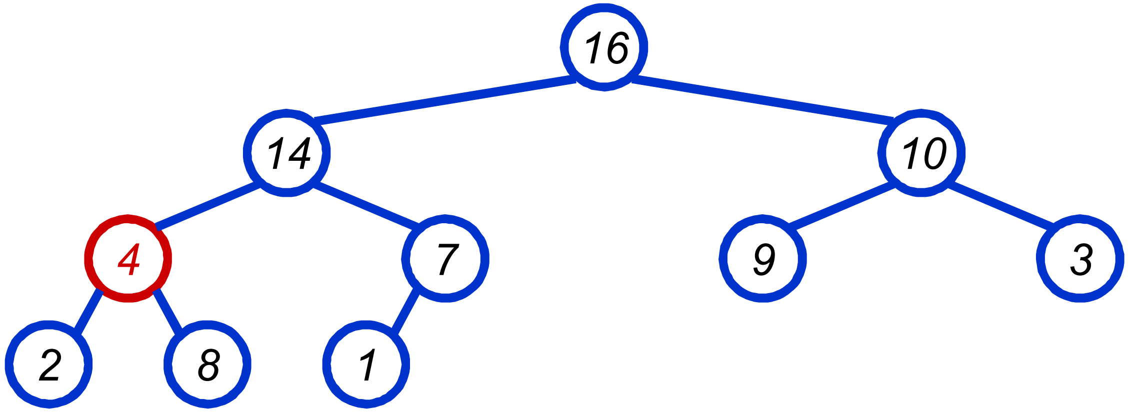



*A* =

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 4 | 7 | 9 | 3 | 2 | 8 | 1 |

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# MaxHeapify() Example

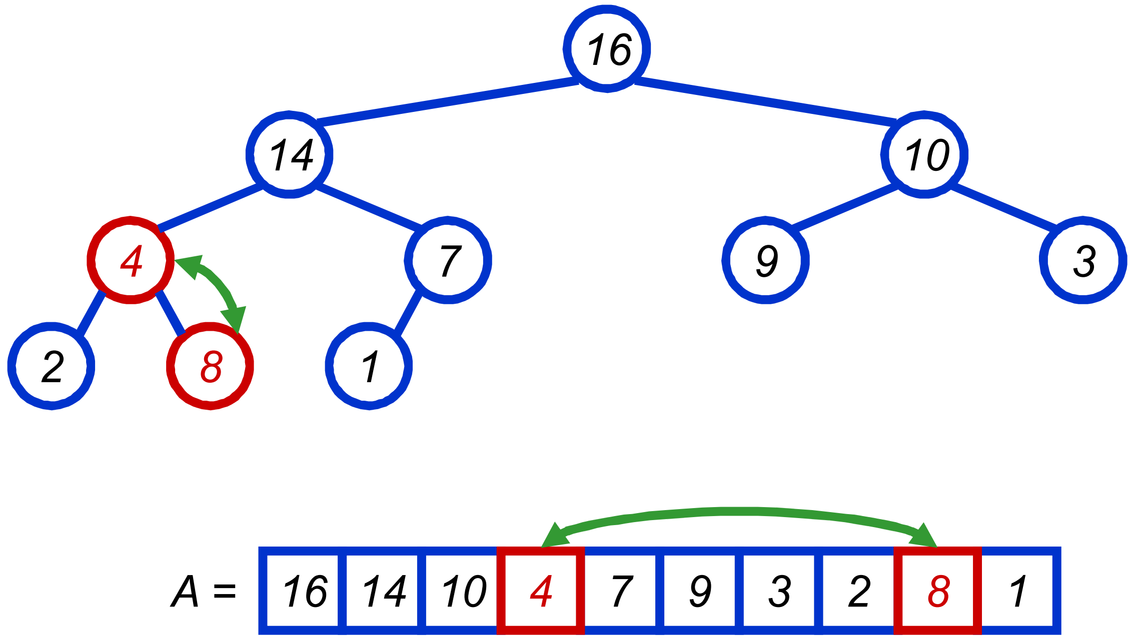



*A* =

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *16* | *14* | *10* | *4* | *7* | *9* | *3* | *2* | *8* | *1* |

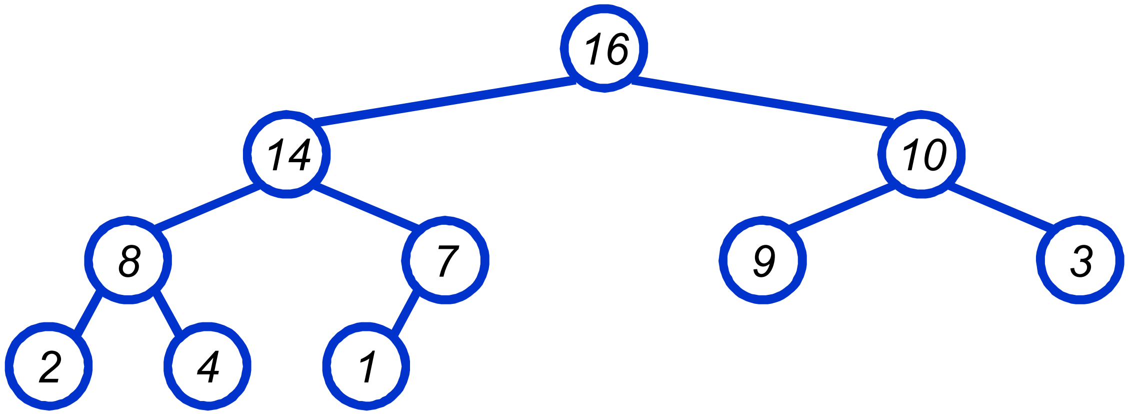
*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# MaxHeapify() Example




*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

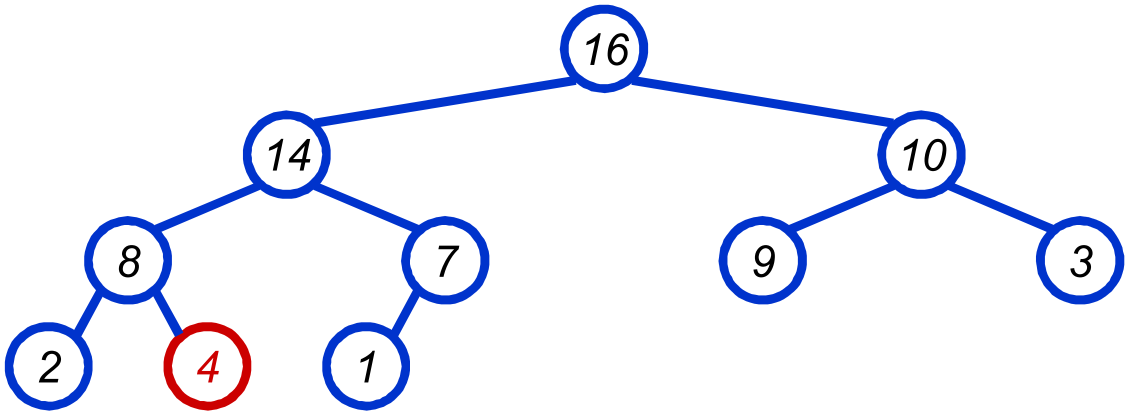
# MaxHeapify() Example




| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *A* = | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# MaxHeapify() Example

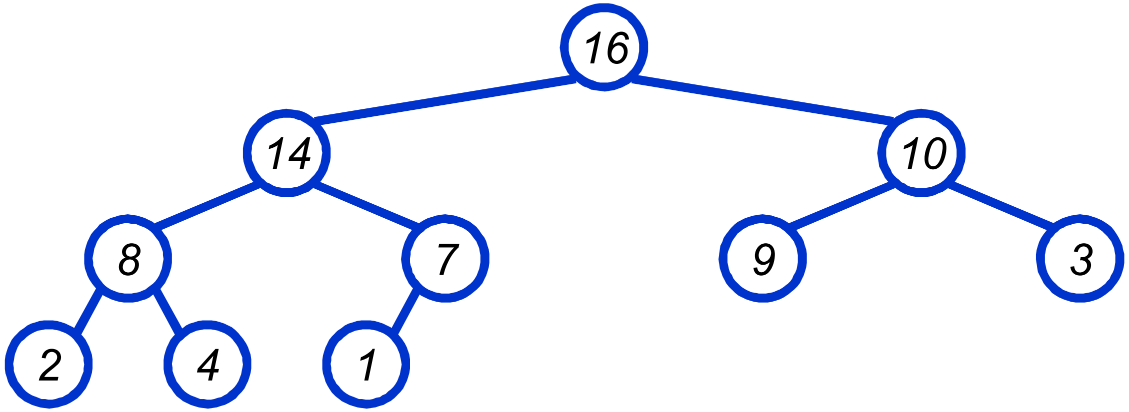



*A* =

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# MaxHeapify() Example




A =

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Heap Operations: MaxHeapify()

_i_: *index of a node that is out-of-heap*
→ *May occur when the value of node _i_ changes*

```
MaxHeapify(A,i)
 l = LEFT(i)
 r = RIGHT(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 swap(A[i],A[largest])
         MaxHeapify(A,largest)    // sub-root is changed
```

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# MaxHeapify() time complexity

- *Aside from the recursive call, what is the running time of* ***`MaxHeapify()`****?*
  - Each call to MaxHeapify takes some constant $c$ steps, because root is compared with direct children of left and right subtrees
- *How many times can* ***`MaxHeapify()`*** *recursively call itself?*
  - MaxHeapify is recursively called "**AT MOST**" $h$ times, where $h$ is the height of the subtree starting at $i$
    - o Why is it not 2**h* times? – only one of left and right subtree is changed
- *Worst-case running time of* ***`MaxHeapify()`*** *on a heap of size n?*
  - for all inputs, "**AT MOST**" $ch$ steps are needed
  - the worst case time complexity is $O(h) = O(\log n)$ where $h$ is the subtree height with root $i$

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Analyzing MaxHeapify(): Formal

- Fixing up relationships between $i$, $l$, and $r$ takes $\Theta(1)$ time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
  - Draw it
  - Note: for a complete binary tree, #(leaf nodes) = #(non-leaf nodes)+1
- Answer: 2$n$/3 (worst case: bottom row 1/2 full)
  - If the left tree is selected, #(nodes in the left) = 2*#(nodes in the right)
- So time taken by **`MaxHeapify()`** is given by

  $T(n) \leq T(2n/3) + \Theta(1)$

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Analyzing MaxHeapify(): Formal

- So we have

$$T(n) \leq T(2n/3) + \Theta(1)$$

- By case 2 of the Master Theorem,

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

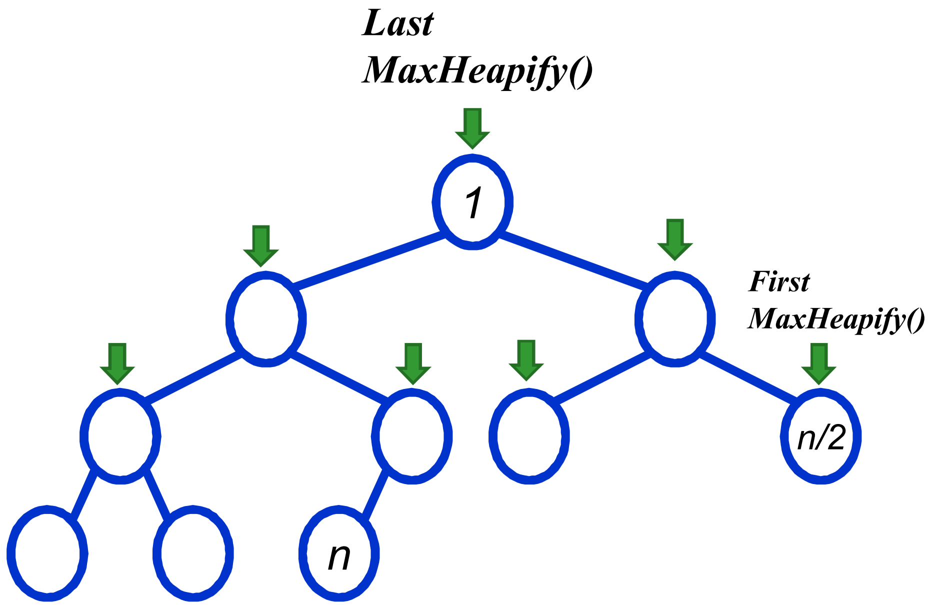
- Thus, **`MaxHeapify()`** takes logarithmic time

# Heap Operations: BuildMaxHeap()

- Question: How efficiently can we build a heap?
- Idea:
  - FIRST create a binary tree (stick each element into a node of the tree) OR put all the elements in an array
  - THEN use MaxHeapify on non-leaf nodes
- We can build a heap in a bottom-up manner by running **`MaxHeapify()`** on successive subarrays

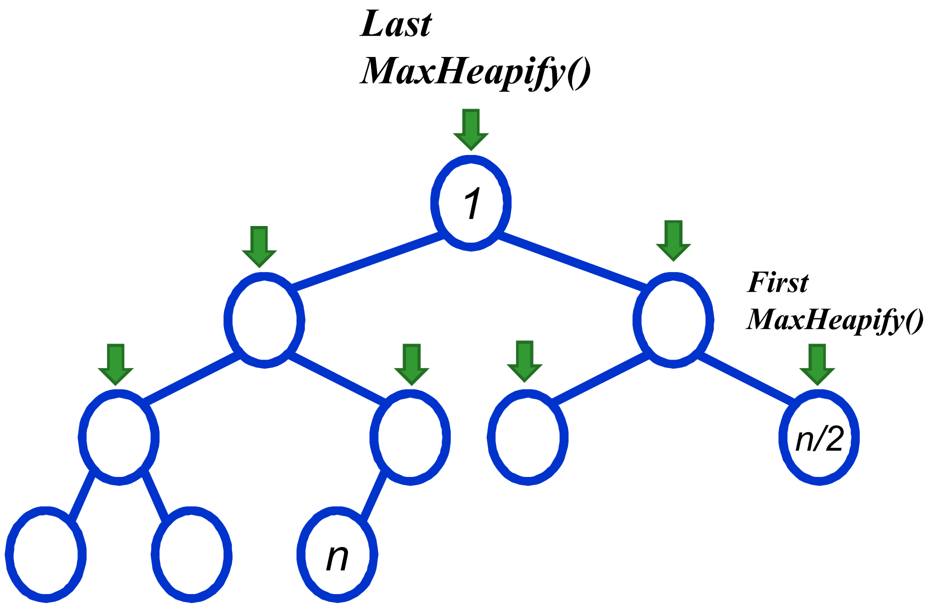
# Heap Operations: BuildMaxHeap()

- Leaf nodes
  - Fact: for array of length *n*, all elements in range A[⌊n/2⌋ + 1 .. n] are heaps
    - o *Why? – a leaf node has no child*
- BuildMaxHeap() in a bottom-up manner:
  - Walk <u>**BACKWARDS**</u> through the array from n/2 to 1, calling **`MaxHeapify()`** on each node.
  - Order of processing guarantees that the children of node *i* are heaps when *i* is processed



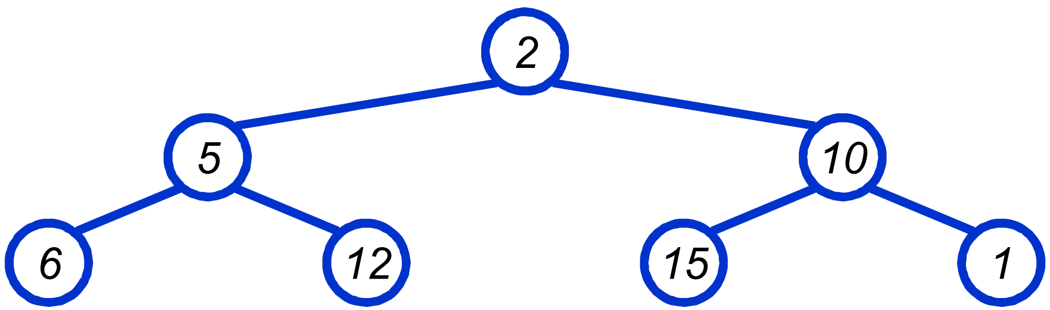
# BuildMaxHeap()

```
// given an unsorted array A
// make A a heap
BuildMaxHeap(A)
{
   heap_size(A) = length(A);
   for (i = ⌊length[A]/2⌋
        downto 1)
        MaxHeapify(A, i);
}
```



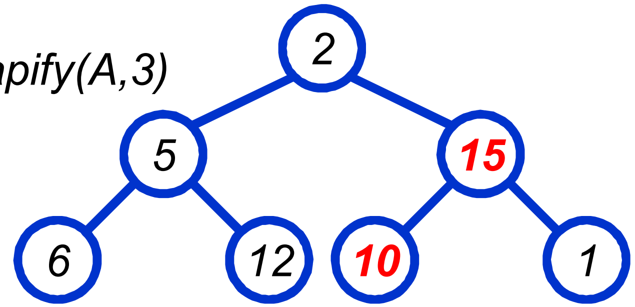
# BuildMaxHeap() Example

- Appy BuildMaxHeap() to the binary tree below
  A = {2, 5, 10, 6, 12, 15, 1}
  - Note: Since length(A) =7, $\lfloor$length[A]/2$\rfloor$ = $\lfloor$3.5$\rfloor$ = 3

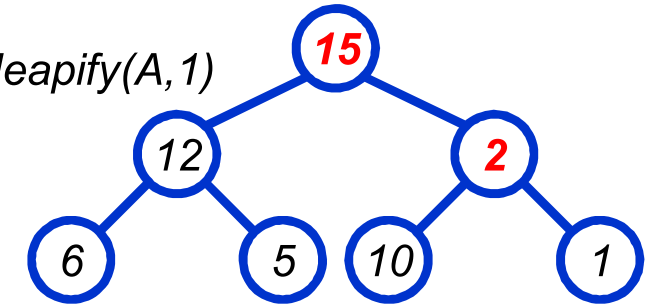


# BuildMaxHeap() Example

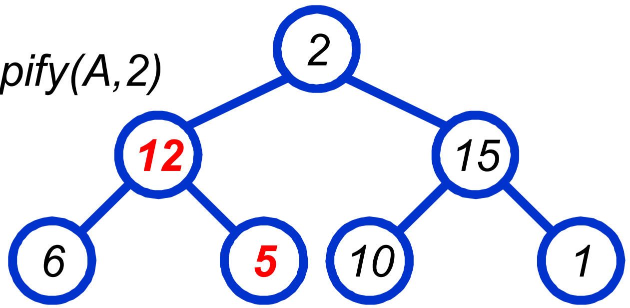
*MaxHeapify(A,3)*



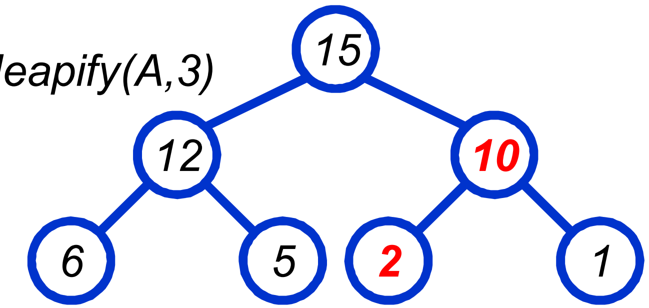

*MaxHeapify(A,1)*




*MaxHeapify(A,2)*



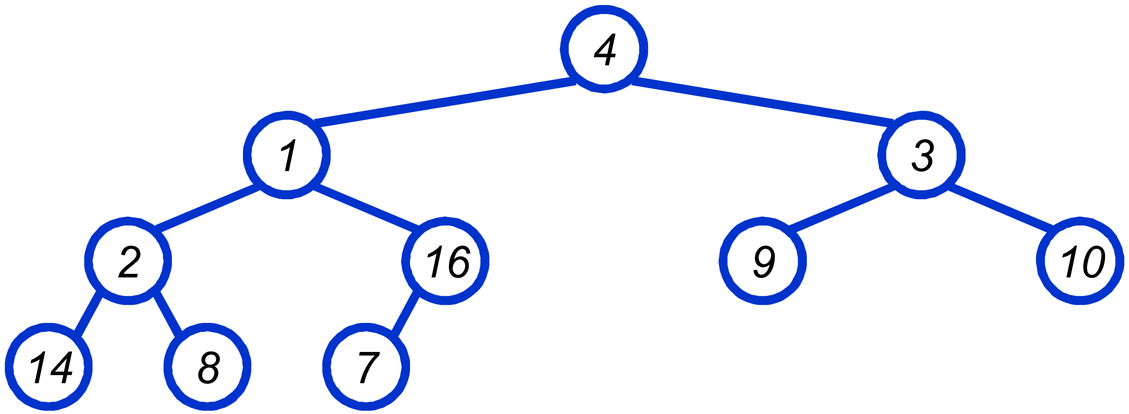

*MaxHeapify(A,3)*




*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# BuildMaxHeap() Example

- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}




*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# BuildMaxHeap Worst Case

- Show that the worst-case running time is O(*n*):
  - MaxHeapify() has a worst-case running time of O(log *n*), and there are at most *n* calls to MaxHeapify()
  - So, the worst-case time complexity is O(*n* log *n*) (*REALLY*?)
  - *Not a correct asymptotic upper bound*

# BuildMaxHeap: Better Analysis

- the running time needed at each level of the tree
  - For a node at height $h$, the worst running time is $(c*h)$
  - There are at most $\lceil n/2^{h+1} \rceil$ nodes of height $h$ in the tree

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 15 | | | | | height = 2 |
| | 12 | | | | 10 | | height = 1 |
| 6 | | 5 | | 2 | | 1 | height = 0 |

  - So, worst-case running time of all nodes at height $h$ is **$c * h * \lceil n/2^{h+1} \rceil$**

- The height varies from **0 to $\lfloor \log2(n) \rfloor$**

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

# BuildMaxHeap: Better Analysis

- Sum this over all the nodes in the tree:

$$T(n) = \sum_{h=0}^{tree\ height} ch\lceil n/2^{h+1}\rceil \le \sum_{h=0}^{\lfloor \log_2 n\rfloor} ch\lceil n/(2\cdot 2^h)\rceil$$

$$\le \frac{cn}{2}\sum_{h=0}^{\infty}\frac{h}{2^h}$$

$$= \frac{cn}{2}\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\frac{5}{32}+\cdots\right) \le \frac{cc_2}{2}n$$

  - Proof: http://courses.washington.edu/css343/zander/NotesProbs/heapcomplexity

    https://math.stackexchange.com/questions/1755708/summation-of-an-expression-sum-h-0-ln-n-frach2h

- So, the worst case time complexity in O(*n*)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

MaxHeap을 이용한 오름차순 정렬

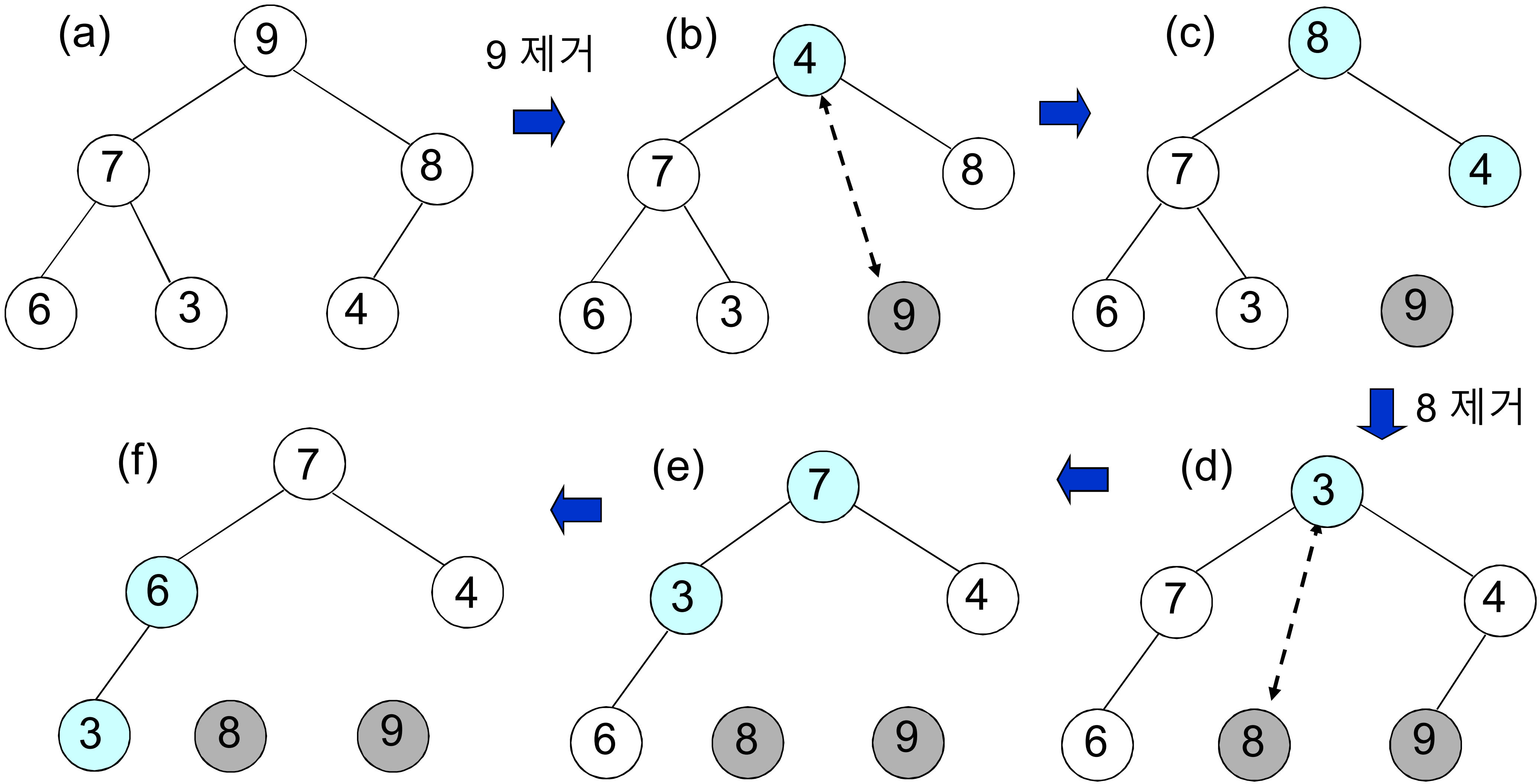
# ASCENDING HEAP SORT

# Heapsort

- 주어진 배열을 힙으로 만든 다음, 차례로 하나씩 힙에서 제거함으로써 정렬한다
- Given **`BuildHeap()`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, always swapping A[1] for A[heap_size(A)]

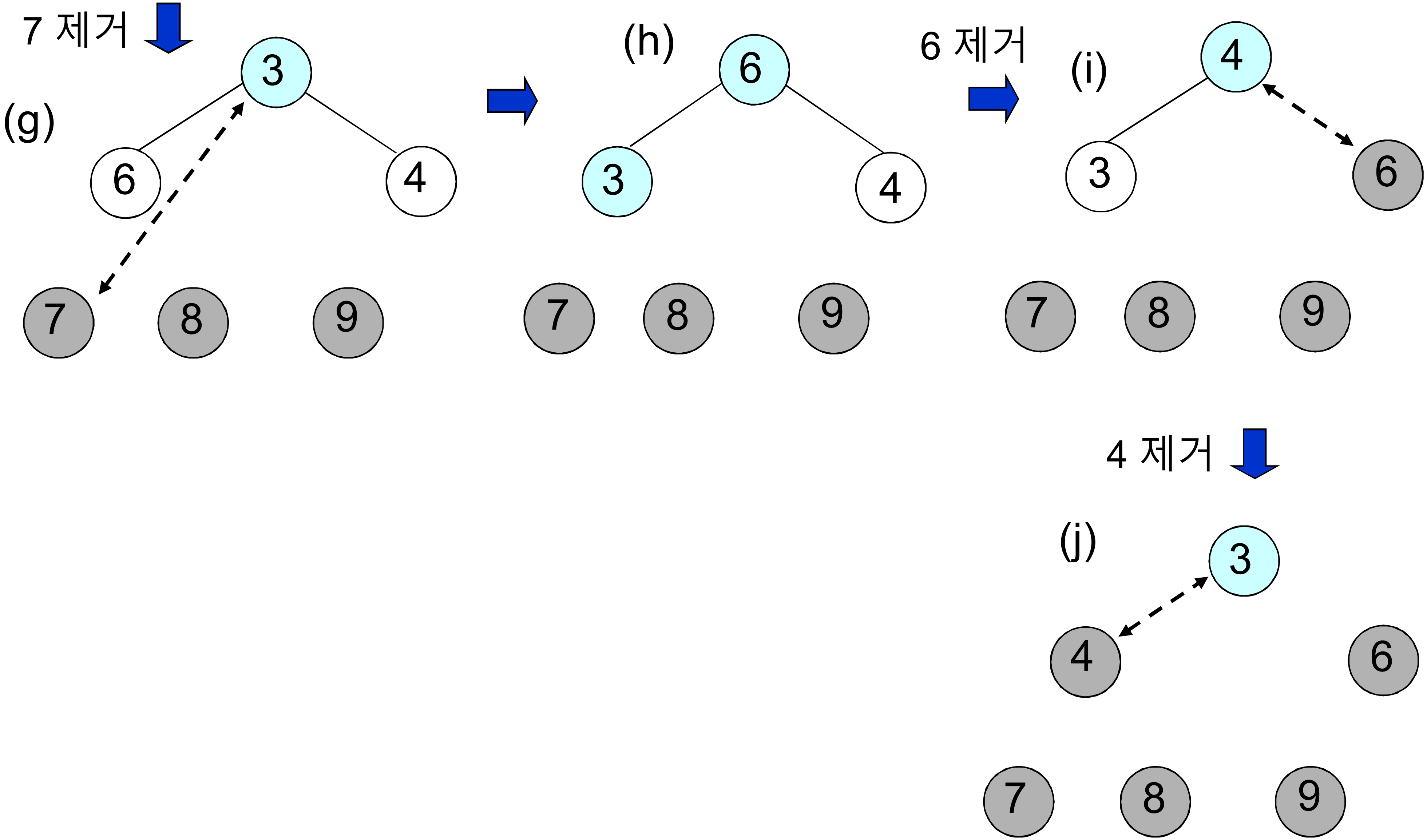
*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Heapsort, Ascending




*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Heapsort, Ascending




*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Heapsort, Ascending

```
HeapsortAscending(A)
/* 오름차순 정렬 */
{
    BuildMaxHeap(A);      /* 최대힙 만들기 */
    for (i = length(A) downto 2)
    {
        Swap(A[1], A[i]);  /* 최대값교환 */
        heap_size(A) -= 1;
        MaxHeapify(A, 1);
    }
}
```

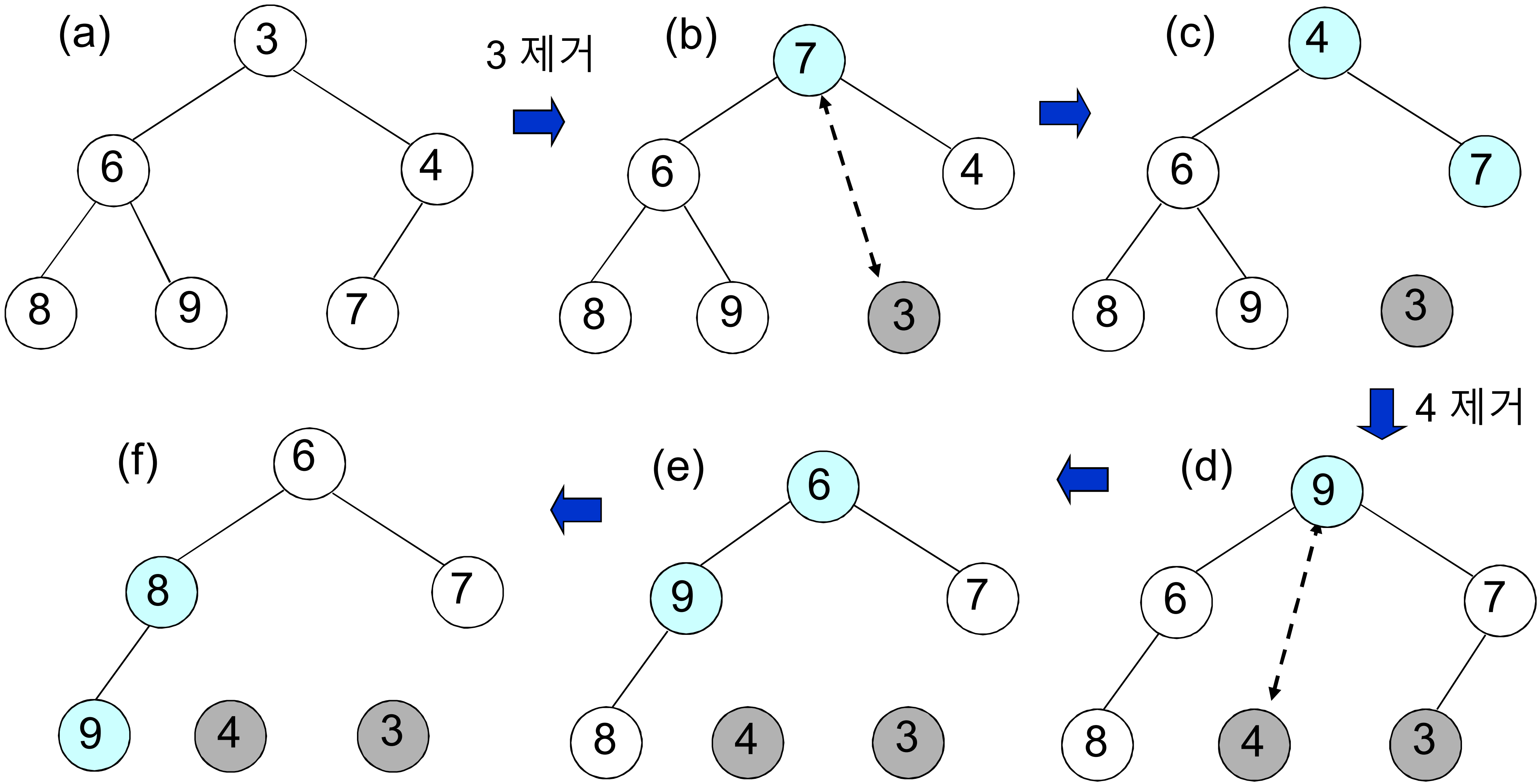
*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

MinHeap을 이용한 내림차순 정렬

# DESCENDING HEAP SORT

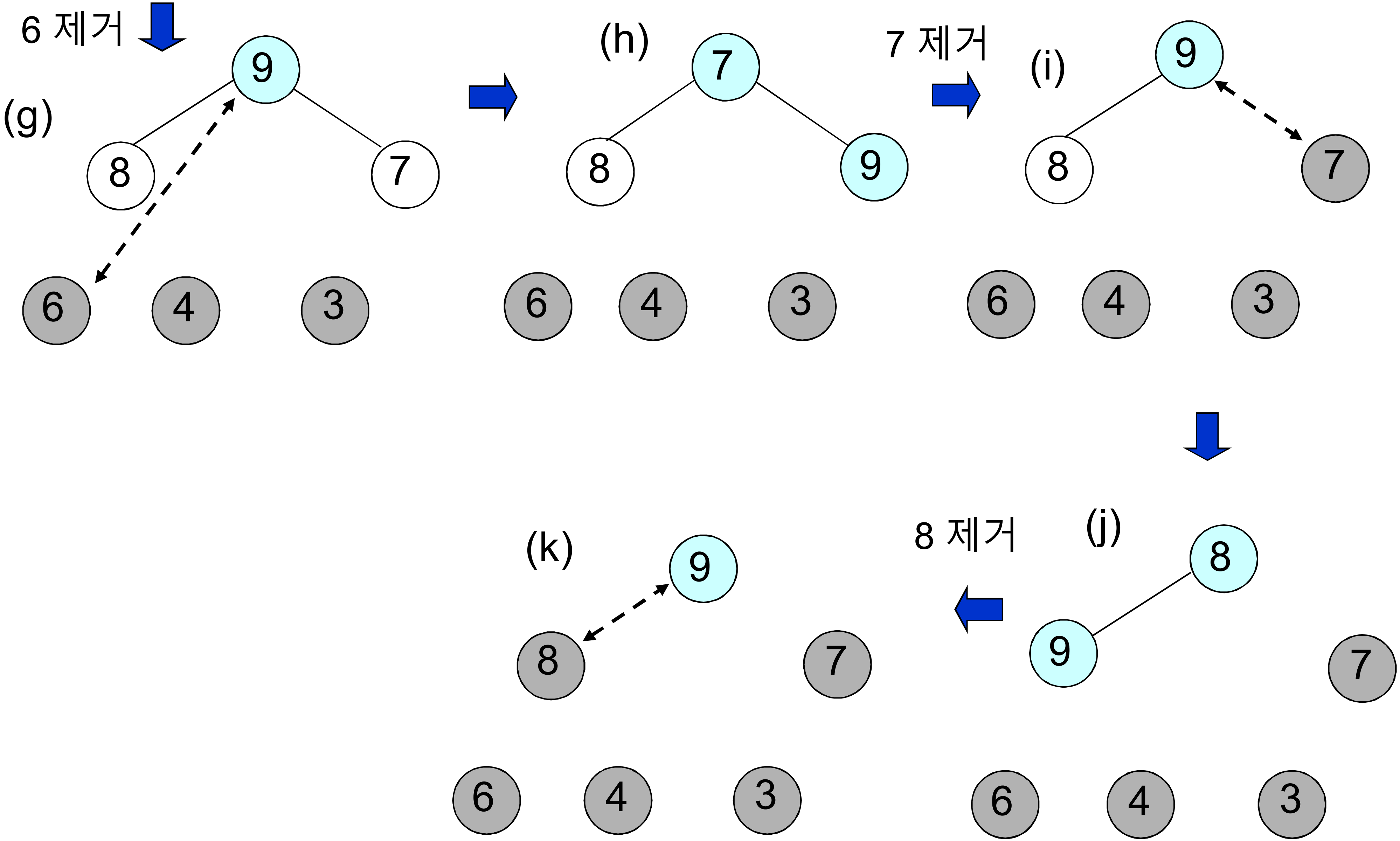
*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Heapsort, Descending




Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

# Heapsort, Descending




*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Heapsort, Descending

```
HeapsortDescending(A)
/* 내림차순 정렬 */
{
    BuildMinHeap(A);     /* 최소힙 만들기 */
    for (i = length(A) downto 2)
    {
        Swap(A[1], A[i]);  /* 최소값교환 */
        heap_size(A) -= 1;
        MinHeapify(A, 1);
    }
}
```

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Analyzing Heapsort

- The initial call to **`BuildHeap()`** takes O($n$) time
- Each of the $n$ - 1 calls to **`Heapify()`** takes O(lg $n$) time
- Thus the total time taken by **`HeapSort()`**
  = O($n$) + ($n$ - 1) O(lg $n$)
  = O($n$) + O($n$ lg $n$)
  = O($n$ lg $n$)
  - 최악의 경우에도 O($n$ lg $n$) 시간 소요!

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# 효율성 비교

| | Worst Case | Average Case |
|---|---|---|
| Selection Sort | $n^2$ | $n^2$ |
| Bubble Sort | $n^2$ | $n^2$ |
| Insertion Sort | $n^2$ | $n^2$ |
| Mergesort (*O(n) extra space) | $n\log n$ | $n\log n$ |
| Quicksort | $n^2$ | $n\log n$ |
| **Heapsort** | $\boldsymbol{n\log n}$ | $\boldsymbol{n\log n}$ |

*Heapsort is an efficient algorithm, but in practice Quicksort usually wins*

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

Heap을 이용한 우선순위 큐

# PRIORITY QUEUES BY HEAPS

# Priority Queues

- The heap data structure is useful in implementing ***priority queues***
  - A data structure for maintaining a set *S* of elements, each associated with ***key***
  - Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**
  - ***What might a priority queue be useful for?***

# Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key

- *How could we implement these operations using a heap?*

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

# Implementing Priority Queues

```
HeapMaximum(A)    // This one is really tricky:
{    return A[1];    }

HeapExtractMax(A)   {
    if (heap_size[A] < 1) { error; }
    max = A[1];
    A[1] = A[heap_size[A]];      // fill the empty root with the last item
    heap_size[A] --;             // reduce the heap size
                                         // now the last item's old
   location is removed
    Heapify(A, 1);               // since A[1] is changed, adjust the heap
    return max;
}
```

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*

# Implementing Priority Queues

```
HeapInsert(A, key)    // what is the running time?
{
    heap_size[A] ++;
    i = heap_size[A];
    while (i > 1  AND  A[Parent(i)] < key) {
        A[i] = A[Parent(i)];
        i = Parent(i);
    }
    A[i] = key;
       // No Heapify()?
}
```

Next topics:

Binary search tree (BST)

# END OF LECTURE 8

*Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I*