Heapsort

Heejin Park
Hanyang University

Contents

- Heaps
- Building a heap
- The heapsort algorithm
- Priority queues

Heapsort

- Like merge sort
  - Running time is $O(n\lg n)$
- Like insertion sort
  - Heapsort sorts in place.

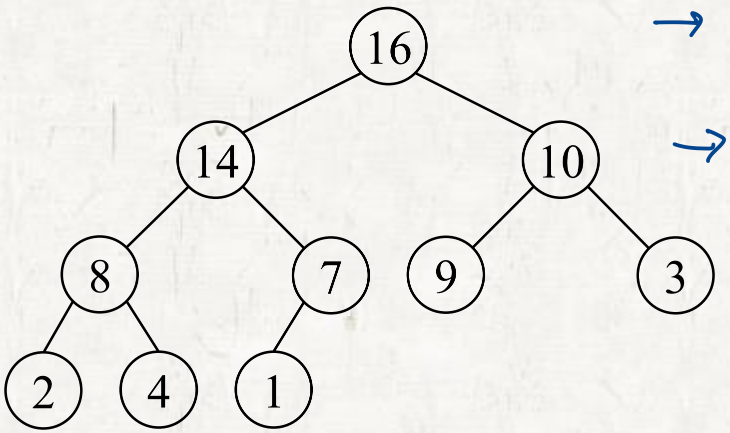
No an additional memory B

Heaps

- The shape of a (binary) heap
  - A nearly *complete binary tree*.
  - Complete binary tree is in which all leaves have the same depth and all internal nodes have degree 2.

→ 마지막 level 제외 모두 채워져 있음

$n = 2^h - 1$



$\rightarrow 1 = 2^1 - 1$

$\rightarrow 3 = 2^2 - 1$

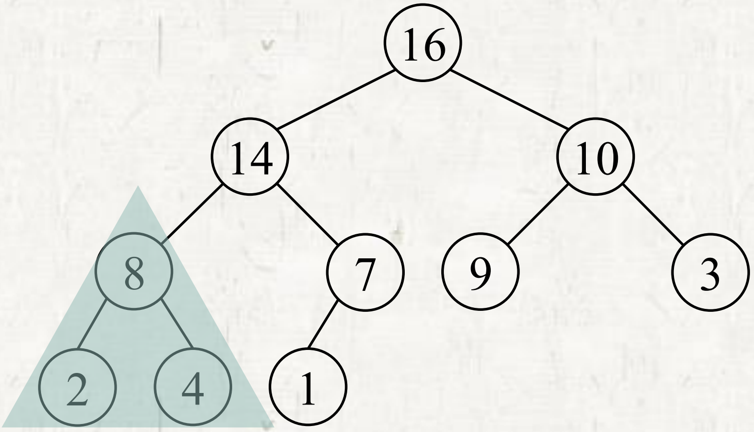
$\rightarrow 7 = 2^3 - 1$

Heaps

- Heap property
  - 2 kinds of binary heaps
    - *max-heaps* and *min-heaps*

Heaps

- ***max-heap property***
  - A[PARENT(i)] $\geq$ A[i]
    - The parent is bigger than or equal to its child.
    - The root node has the largest element.
    - The root of any subtree has the largest element among the subtree.

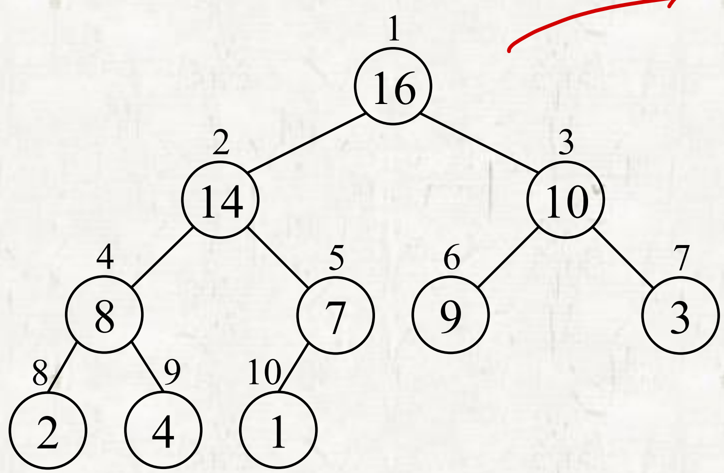


Heaps

- ***min-heap property***
  - A[PARENT(i)] ≤ A[i]
    - A child is bigger than or equal to its parent.
    - The root node has the smallest element.

Heaps

- A heap can be stored in an array.
  - The root is stored in $A[1]$.
  - All elements are stored in level order.



Struct {
* element
* parent
* left child
* right child }

4times more space
↳ If we array
→ don't need additional memory!!

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

Heaps

i = level (depth)

- PARENT(i) $\left\lfloor \frac{i}{2} \right\rfloor$

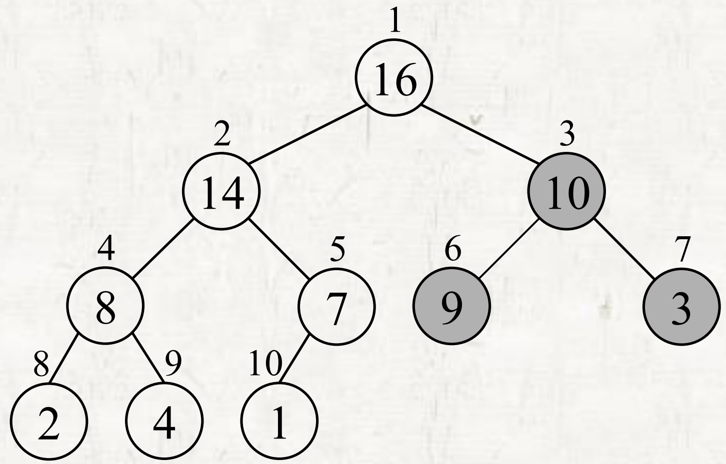
  return

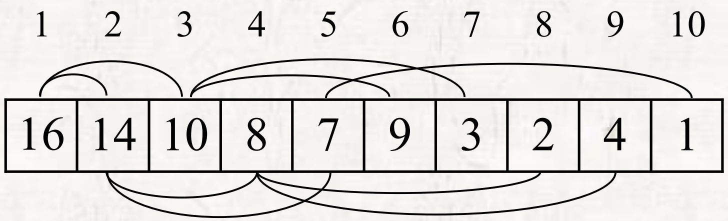
- LEFT(i)

  return $2i$

- RIGHT(i)

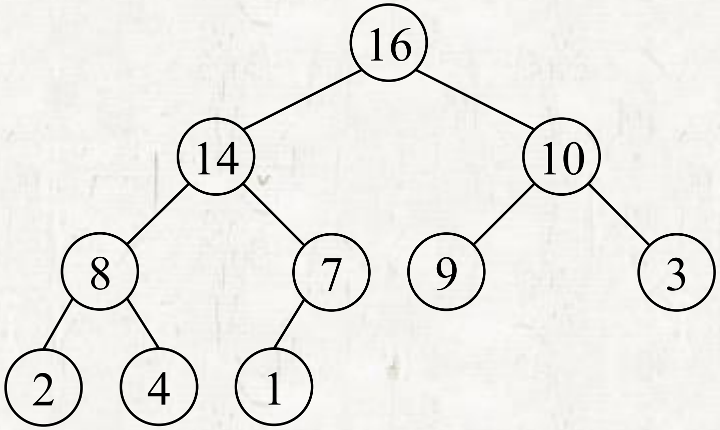
  return $2i + 1$





Heaps

- The height of a node
  - The number of edges on the longest simple downward path from the node to a leaf.

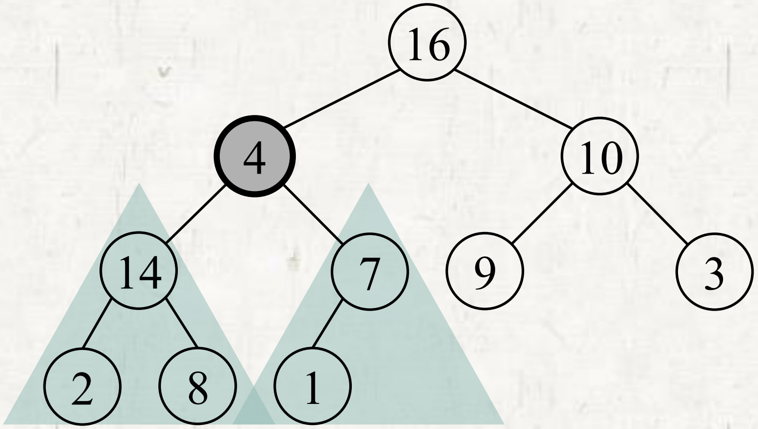


Heaps

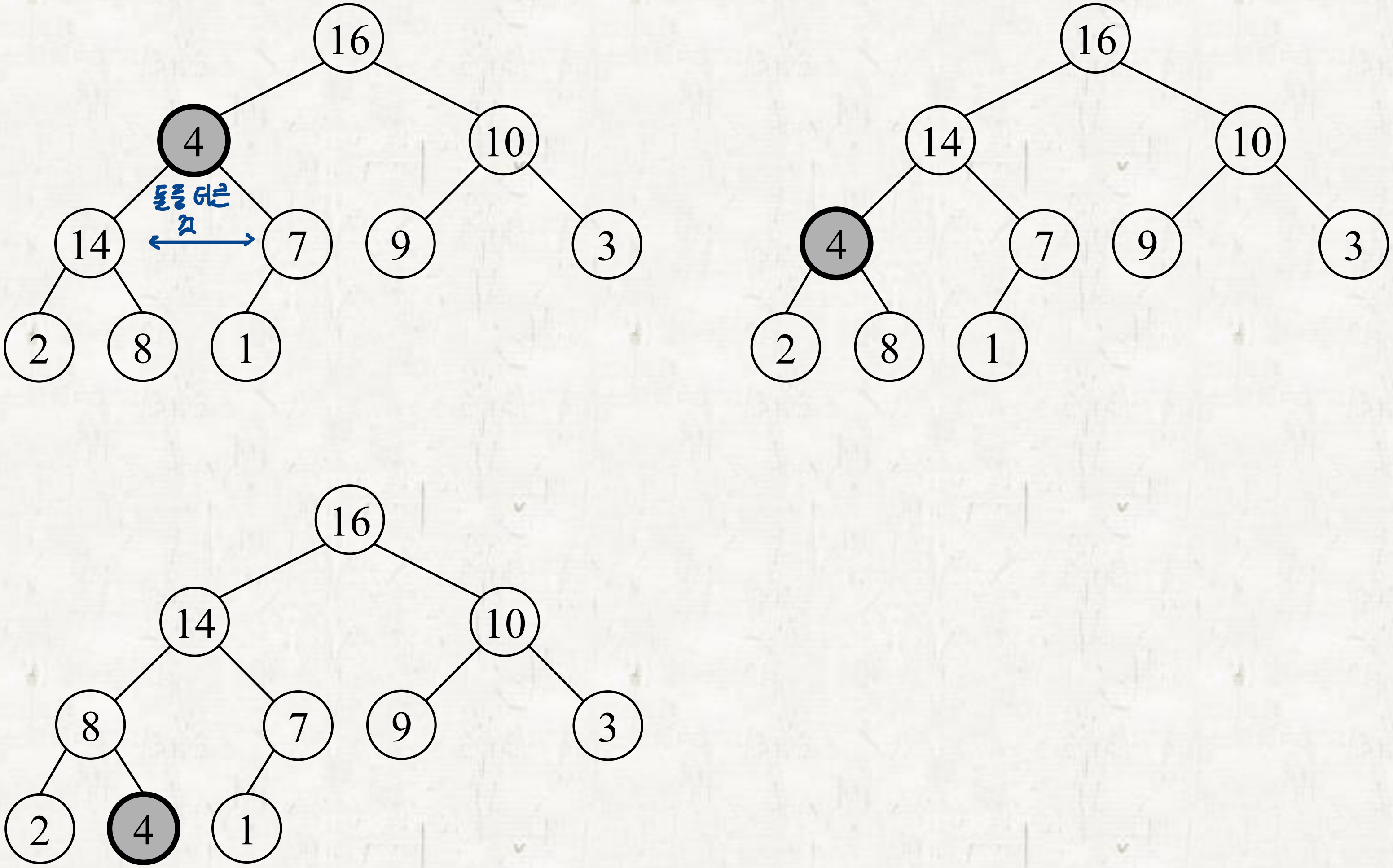
- The height of a heap
  - The height of the root.
  - $\Theta(\lg n)$
  - Since a heap of n elements is based on a complete binary tree.

Maintaining the heap property

- Max-Heapify procedure
  - Input : A node whose left and right subtrees are max-heaps, but the value at the node may be smaller than those of its children, thus violating the max-heap property.
  - Let the value at the node "float down" in the max-heap so that the subtree rooted at the node becomes a max-heap.



Maintaining the heap property



Maintaining the heap property

- The running time of MAX-HEAPIFY
  - T(n) where n is the number of nodes in the subtree.
    - $\Theta(1)$ time to exchange values
    - $O(h) = O(\lg n)$ time in total → worse case: height of tree

not $\Theta(\log n)$ → why?

In best case, one exchange or no exchange can max max heap

Maintaining the heap property

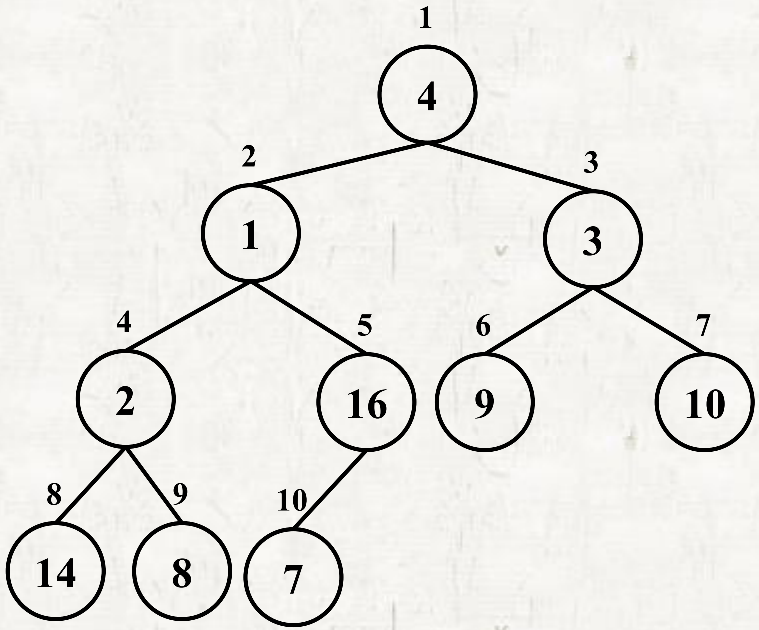
- BUILD-MAX-HEAP

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*)

Building a heap

- BUILD-MAX-HEAP
  - The input array with 10 elements and its binary tree representation.

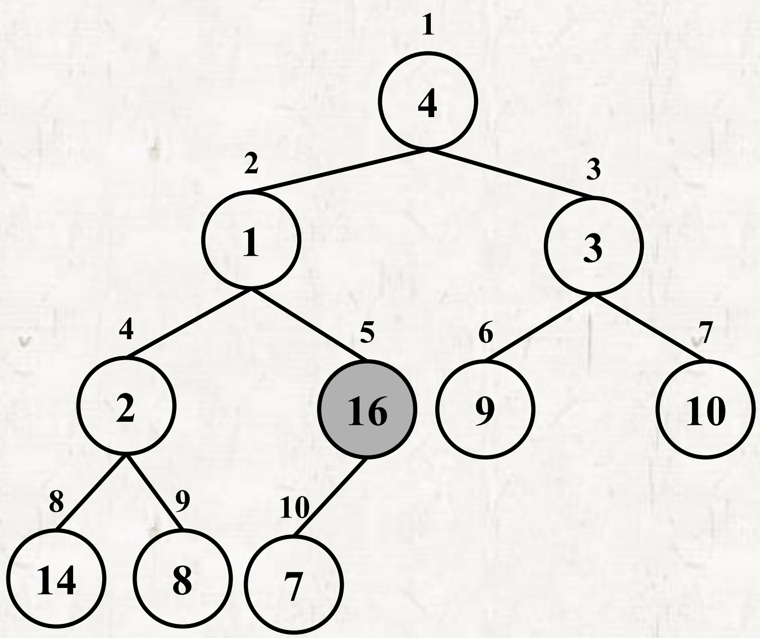
4	1	3	2	16	9	10	14	8	7

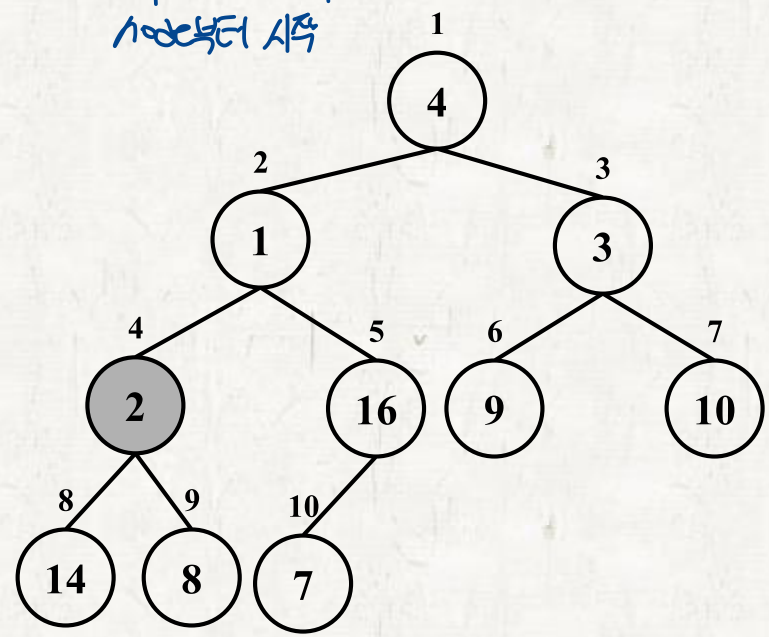


Building a heap

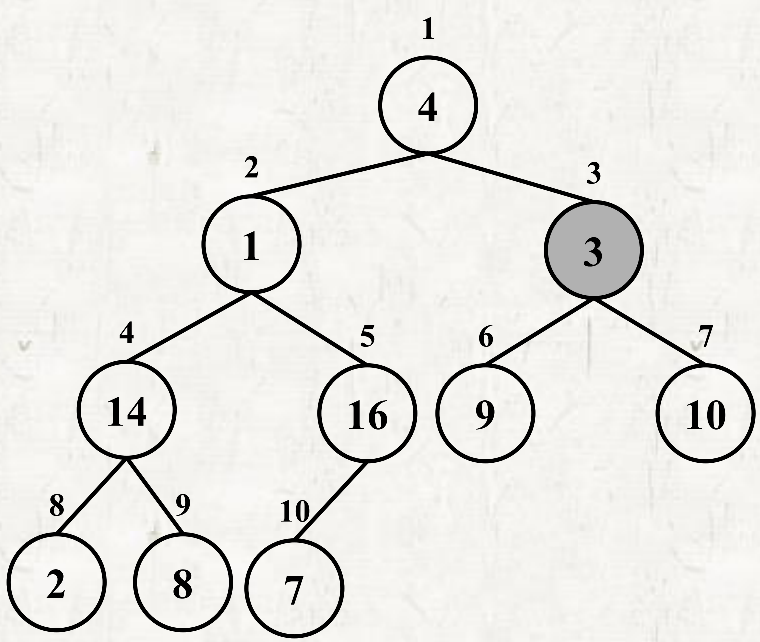
- Call MAX-HEAPIFY(A, i) at the rightmost node that has the child from the bottom. $i = \lfloor A.length / 2 \rfloor$

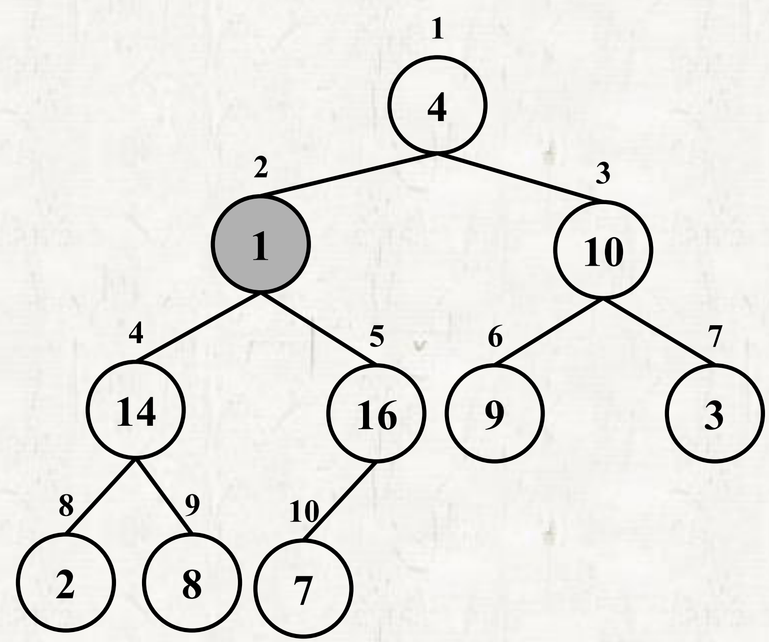
자식이 있는 가장 큰 index의 node부터 시작



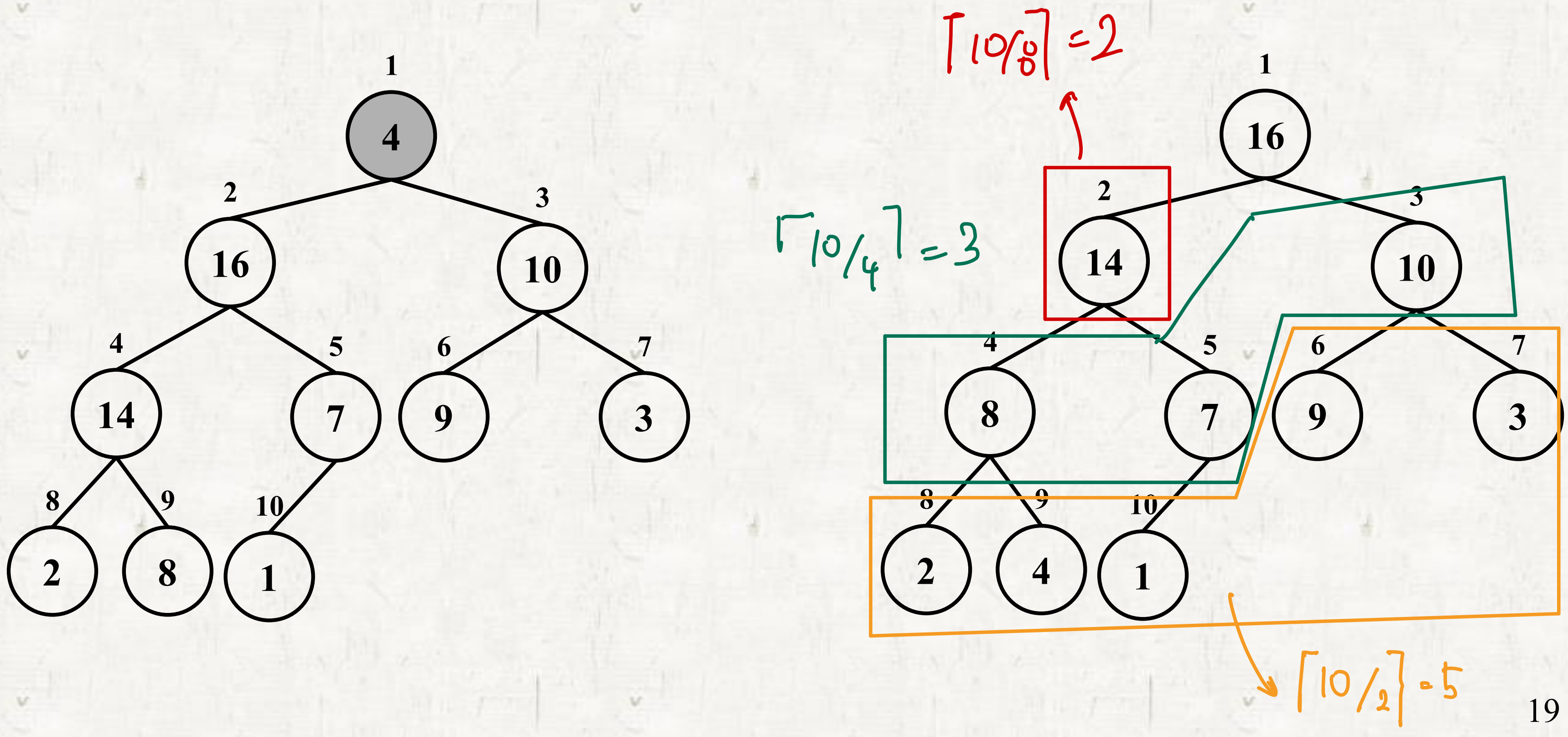


Building a heap





Building a heap



Building a heap

- Running time
  - Upper bound
    - Each call to MAX-HEAPIFY costs $O(\lg n)$ time, and there are $O(n)$ such calls, Thus, the running time is $O(n \lg n)$.

Building a heap

- Running time
  - Tighter upper bound
    - The time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.
    - Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.

Building a heap

- Tighter bound
  - The running time of MAX-HEAPIFY on a node of height h is $O(h)$, so the total cost of BUILD-MAX-HEAP is

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

  - The last summation can be evaluated as follows.

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

Building a heap

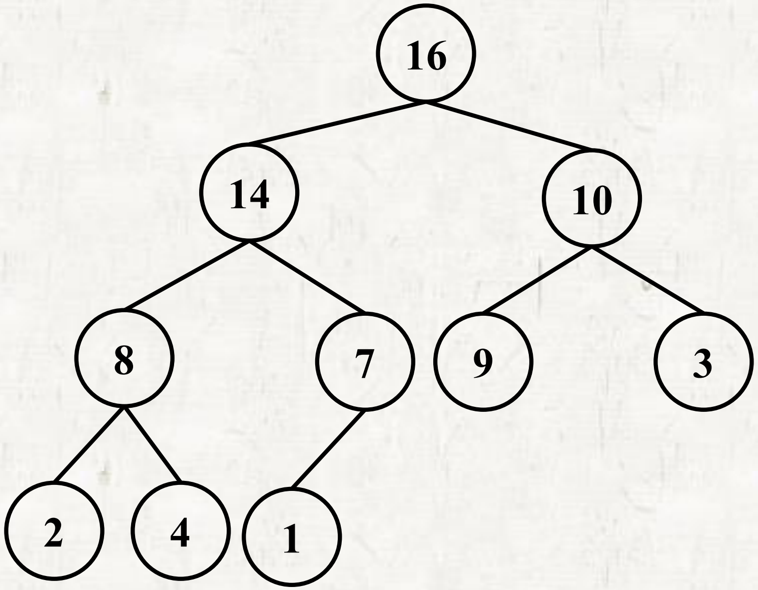
- Thus, the running time of BUILD-MAX-HEAP can be bounded as

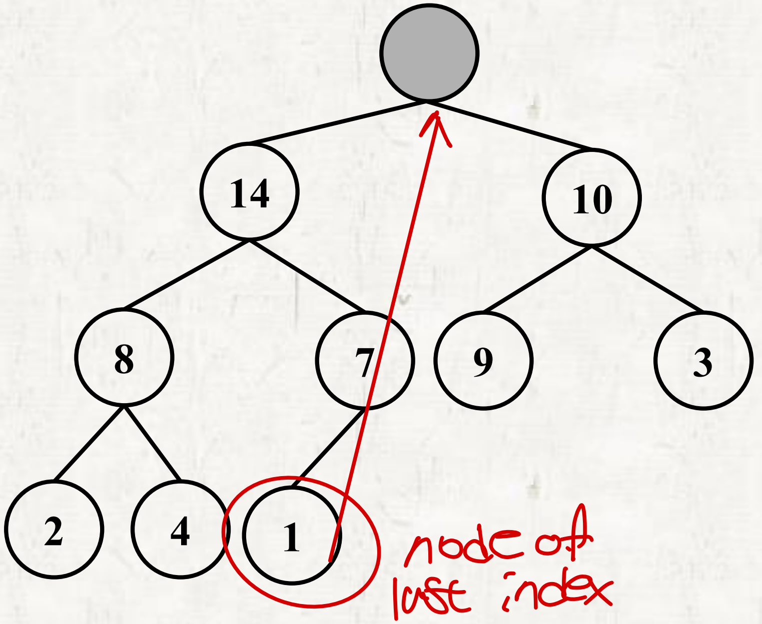
$$O(n\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(n\sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

- Hence, we can build a max-heap in linear time.

Extract-Max

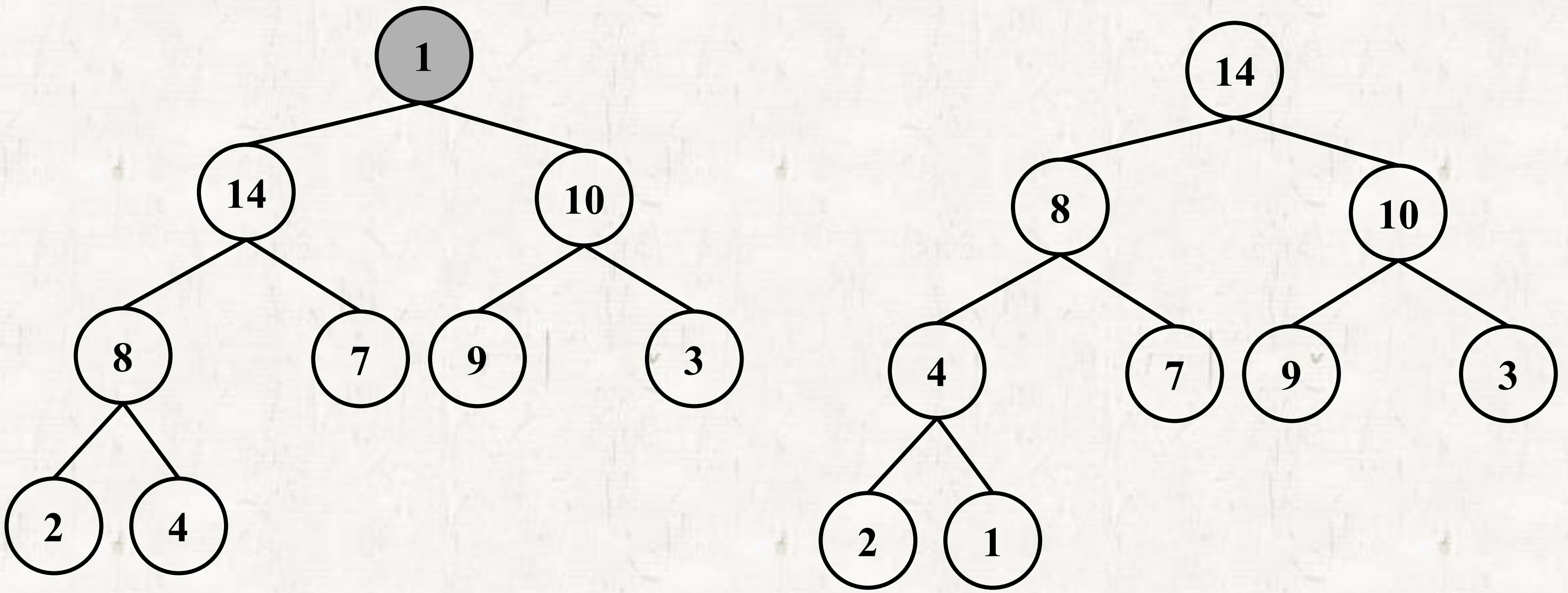
- Extract-Max
  - Remove the maximum element from a heap
  - Restore the structure to a heap





Extract-Max

- Extract-Max
  - Restore the structure to a heap

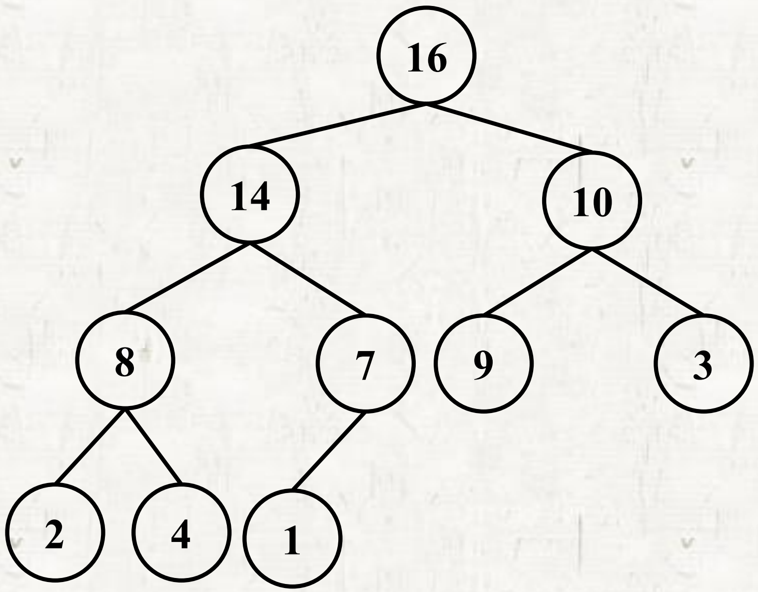


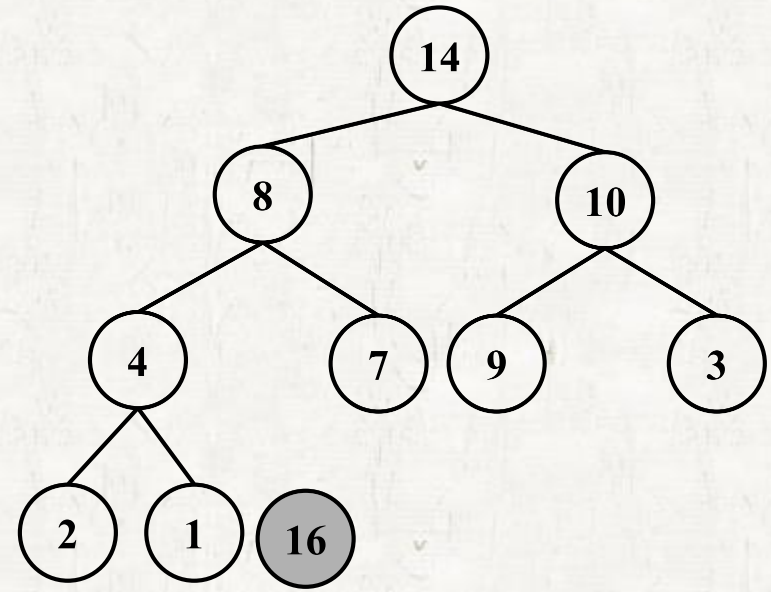
  - $O(\lg n)$

Heapsort

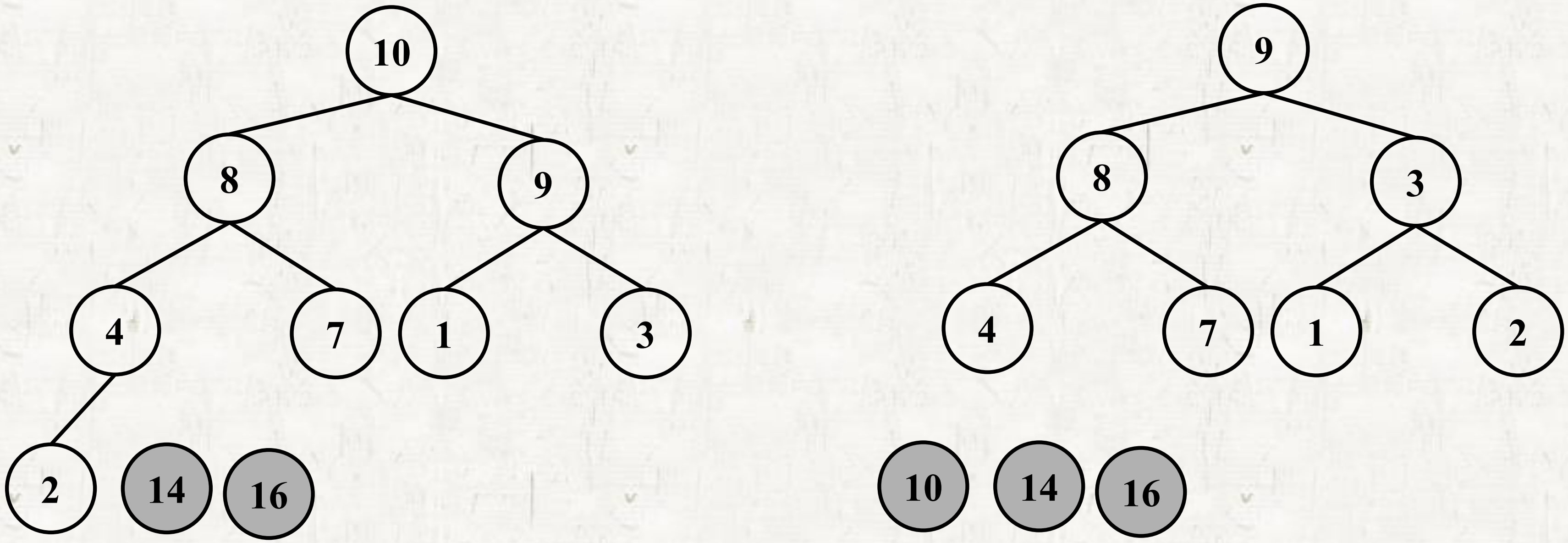
- The Heapsort algorithm
  - BUILD-MAX-HEAP on $A[1..n]$
    - $O(n)$ time.
  - Extract Max for n times
    - $O(n \lg n)$ time.

Heapsort

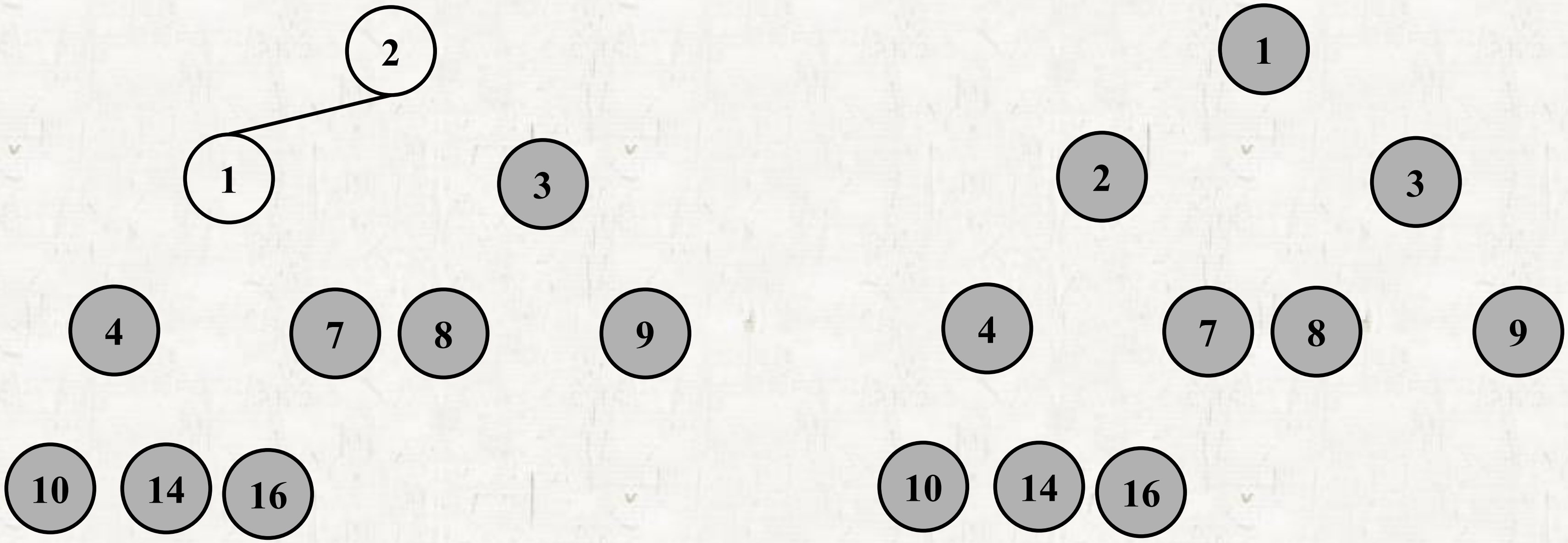




Heapsort



Heapsort



1	2	3	4	7	8	9	10	14	16

Heapsort

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

Heapsort

- The running time of Heapsort
  - $O(n\lg n)$
    - BUILD-MAX-HEAP: $O(n)$
    - MAX-HEAPFY: $O(\lg n)$

Priority queues

- Priority Queue
  - A data structure for maintaining a set S of elements, each with an associated value called a key.
  - A max-priority queue operations.
    - INSERT(S, x) inserts the element x into the set S.
    - MAXIMUM(S) returns the element of S with the largest key.
    - EXTRACT-MAX(S) removes and returns the element of S with the largest key.
    - INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k,

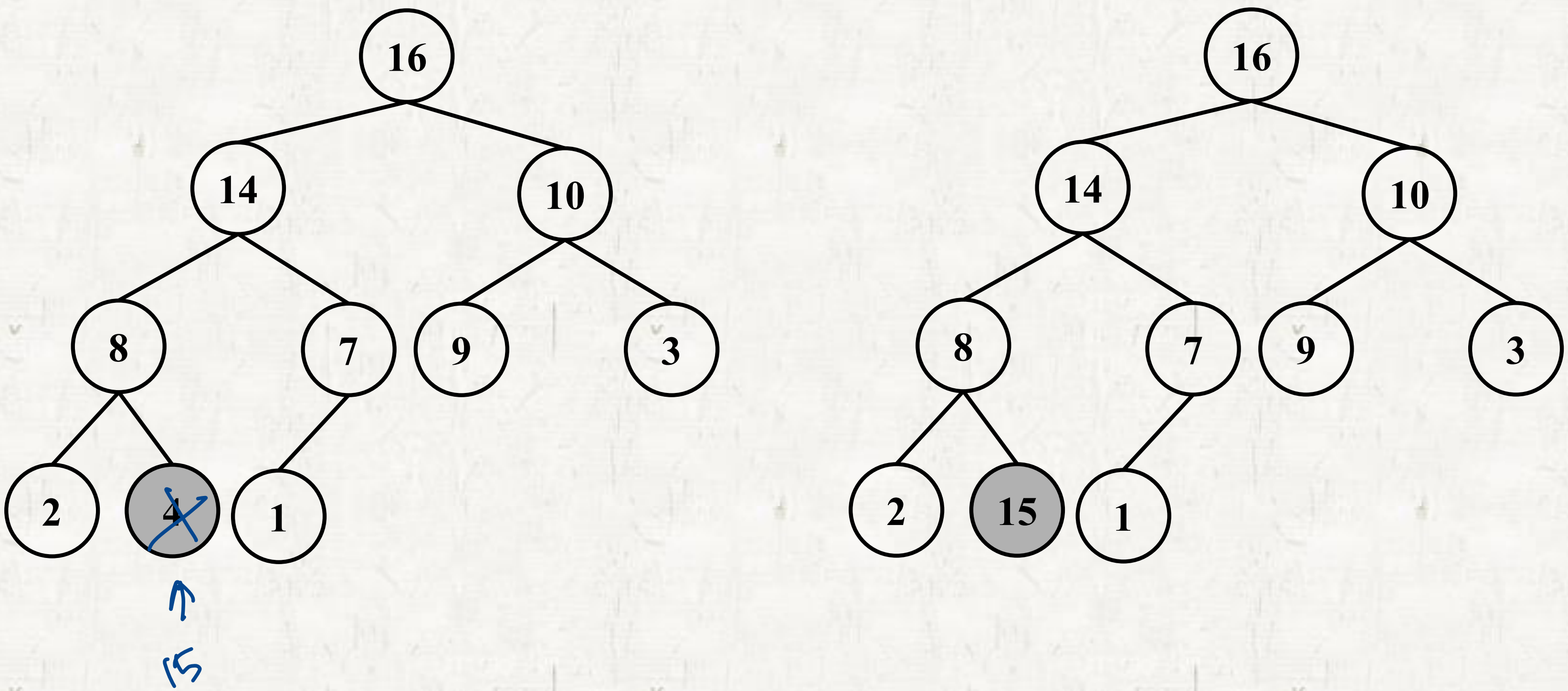
Priority queues

- MAXIMUM
  - Read the max value → array[1]
  - $O(1)$ time

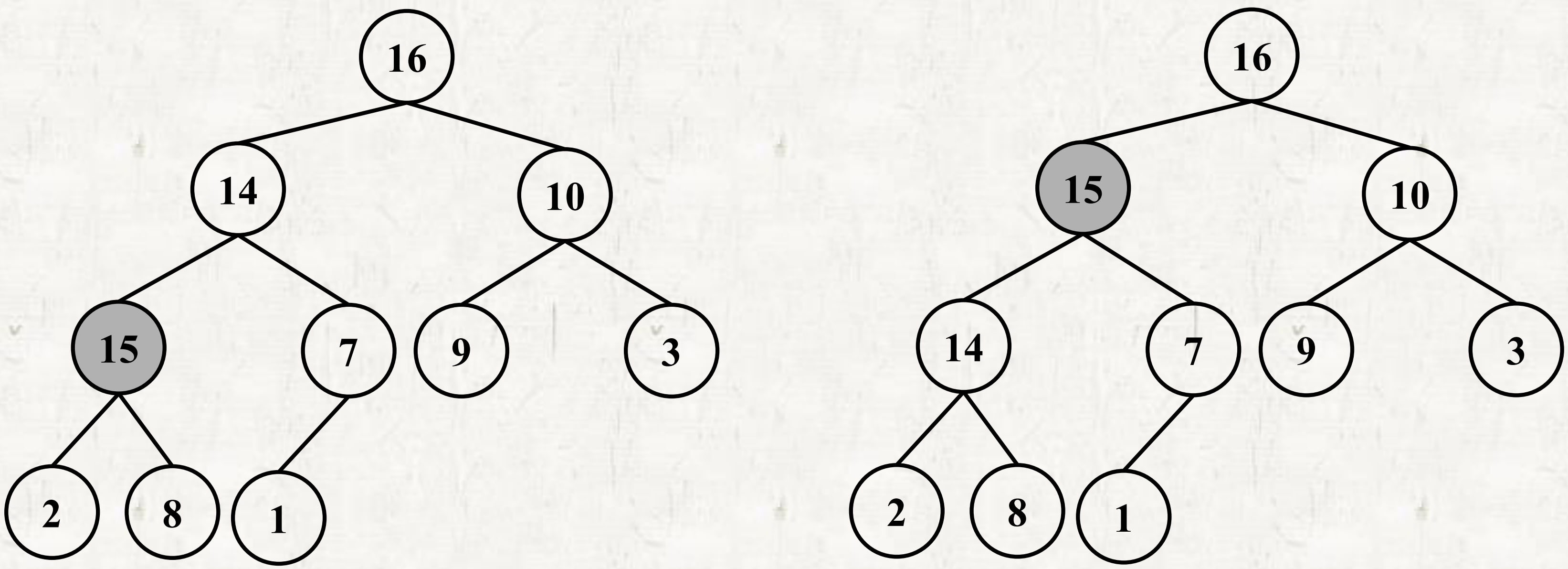
- EXTRACT-MAX
  - Remove the max value + MAX-HEAPIFY
  - $O(\lg n)$

Priority queues

- INCREASE-KEY



Priority queues



Priority queues

- HEAP-INCREASE-KEY
  - $O(\lg n)$ time.

Priority queues

- INSERT
  - $O(\lg n)$ time.

MAX-HEAP-INSERT(*A*, *key*)
1. *A.heap-size* = A.*heap-size* + 1
2. *A*[*A.heap-size*] = $-\infty$ → negative infinity
3. HEAP-INCREASE-KEY(*A*, A.*heap-size*, *key*)

Self-study

- **Exercise 6.3-1**
  - BUILD-MAX-HEAP on A = <5, 3, 17, 10, 84, 19, 6, 22, 9>
- **Exercise 6.4-1**
  - HEAPSORT on A = <5, 13, 2, 25, 7, 17, 20, 8, 4>
- **Exercise 6.5-8 (6.5-7 in the 2nd ed.)**
  - Give an algorithm for HEAP-DELETE(A,i) in $O(\lg n)$ time.