Heaps

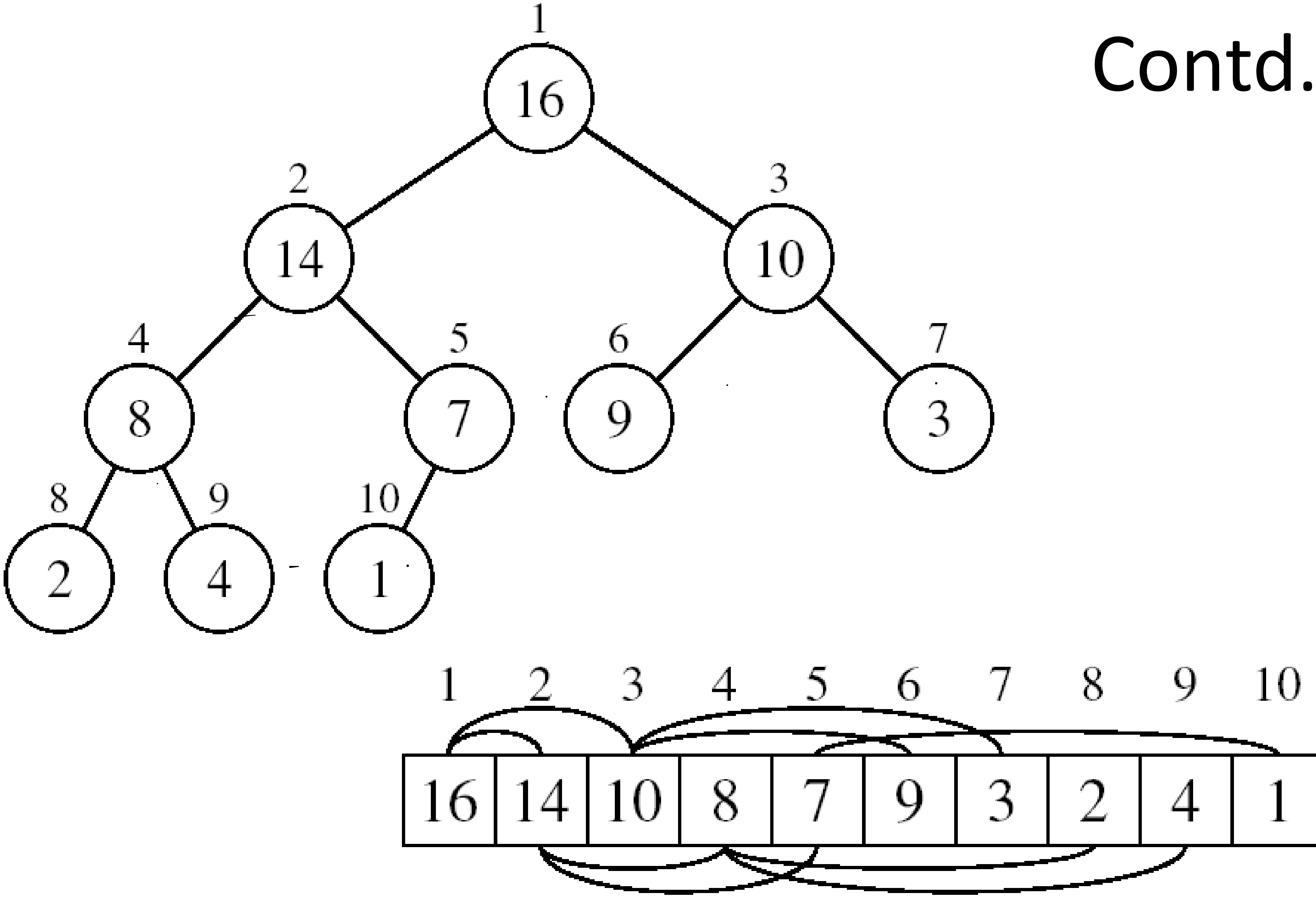
Heap Data Structure

- It is a nearly complete binary tree.
  - All levels are full, except possibly the last one, which is filled from left to right.
  - Due to this, they are commonly stored using arrays as there is no memory wastage.
- Values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap.

Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Number of elements in the array = $A.length$
  - Number of elements in the heap which are stored within array A = $A.heap\text{-}size$
  - $0 \le A.heap\text{-}size \le A.length$
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves

Contd…



Types of Heaps

- **Max-heaps** (largest element at root), satisfy the *max-heap property:*
  - for every node i other than the root,

$$A[\mathrm{PARENT}(i)] \geq A[i]$$

- **Min-heaps** (smallest element at root), satisfy the *min-heap property:*
  - for every node i other than the root,

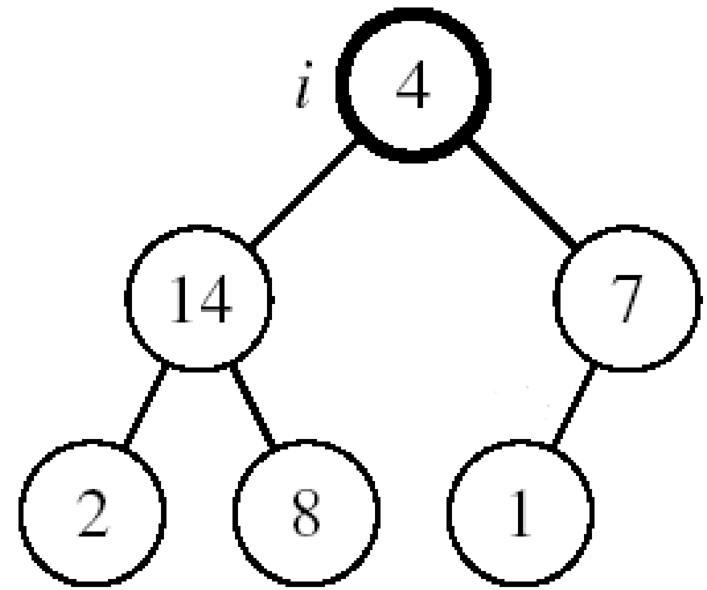
$$A[\mathrm{PARENT}(i)] \leq A[i]$$

Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queues
  - MAX-HEAP-INSERT,
  - HEAP-EXTRACT-MAX,
  - HEAP-INCREASE-KEY, and
  - HEAP-MAXIMUM

Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Left and Right subtrees of *i* are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children

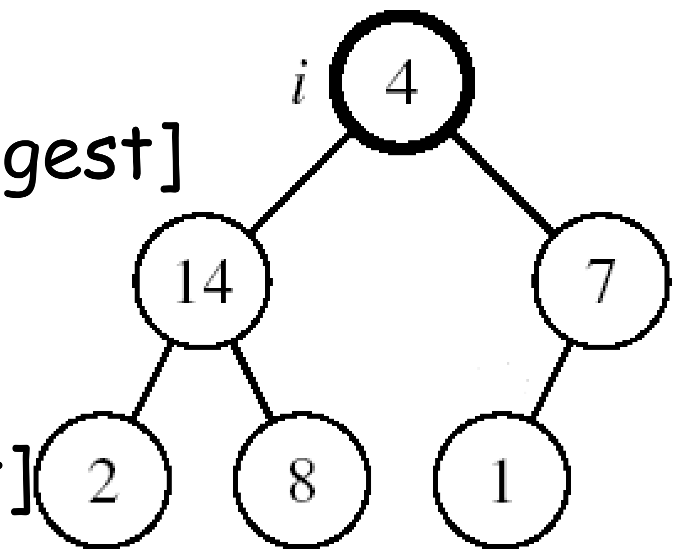


Maintaining the Heap Property

MAX-HEAPIFY(*A*, i)

l = LEFT(i)
r = RIGHT(i)
if l ≤ *A*.heap-size and *A*[l] > *A*[i]
    then largest = l
    else largest = i
if r ≤ *A*.heap-size and *A*[r] > *A*[largest]
    then largest = r
if largest ≠ i
    then exchange *A*[i] $\boxed{\leftrightarrow}$ *A*[largest]
MAX-HEAPIFY(*A*, largest)

Assumptions:
- Left and Right subtrees of i are max-heaps
- *A*[i] may be smaller than its children

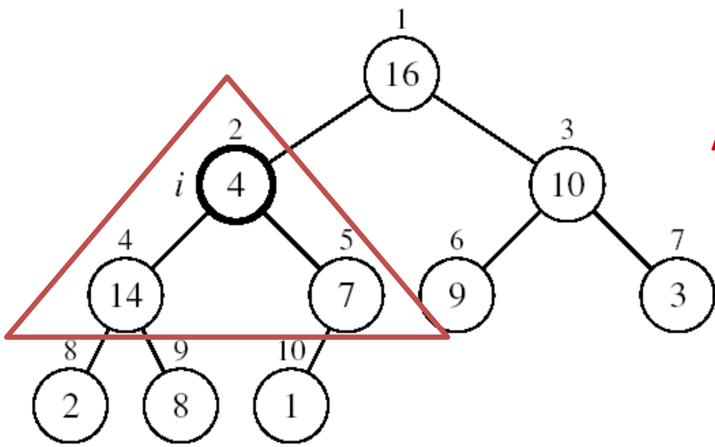


Example

A.heap-size = 10

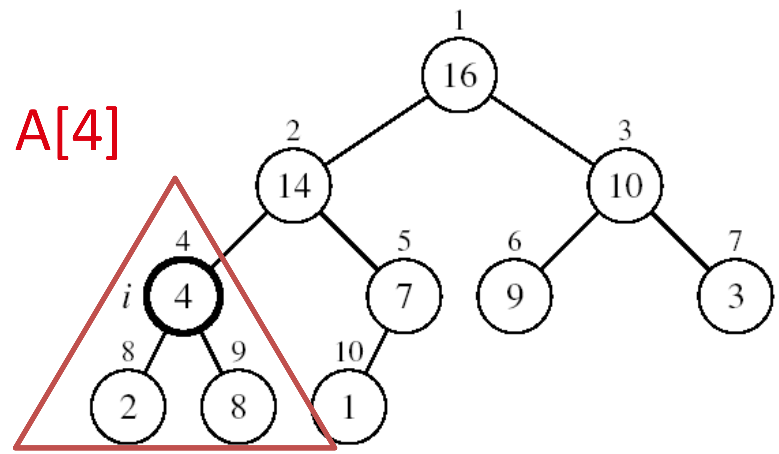
A.length = 10

<u>MAX-HEAPIFY(A, 2)</u>



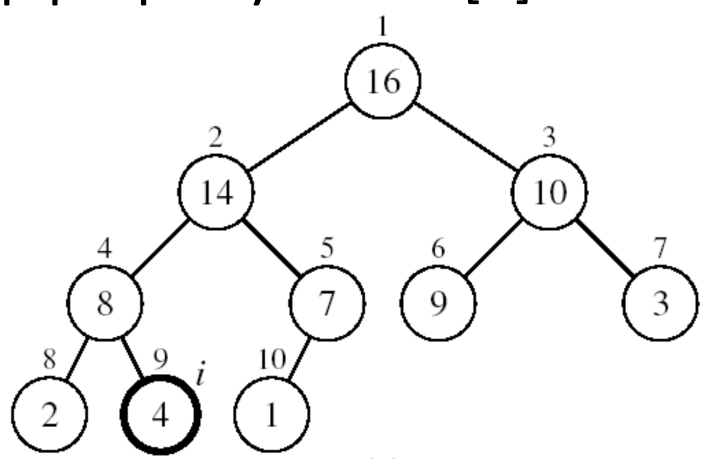
A[2] violates the heap property

A[2] $\leftrightarrow$ A[4]



A[4] violates the heap property

A[4] $\leftrightarrow$ A[9]



Heap property restored

MAX-HEAPIFY Running Time

- It traces path from root to a leaf.
- In worst case length of length path is h.
- Running time of MAX-HEAPIFY is $O(h)$ or $O(\lg n)$
  - Since the height of the heap is $\lg n$.

OR

- In the worst case (last level is exactly half-filled), the children's subtrees each have size at most 2n/3.
- Thus, running time of MAX-HEAPIFY

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

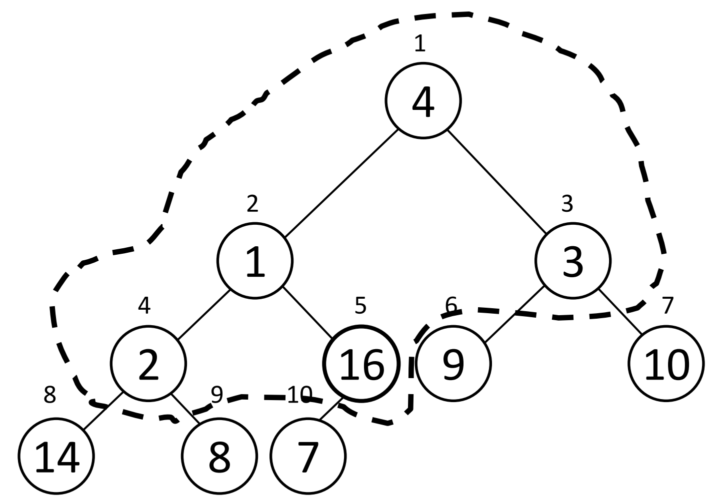
- Case 2 of master theorem. $T(n) = O(\lg n)$

Building a Heap

- Convert an array $A[1 \ldots n]$, where $n = A.length$ into a max-heap.
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) \,..\, n]$ are leaves.
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$.

BUILD-MAX-HEAP(A)

1. $A.heap\text{-}size = A.length$
2. **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3. MAX-HEAPIFY(A, i)

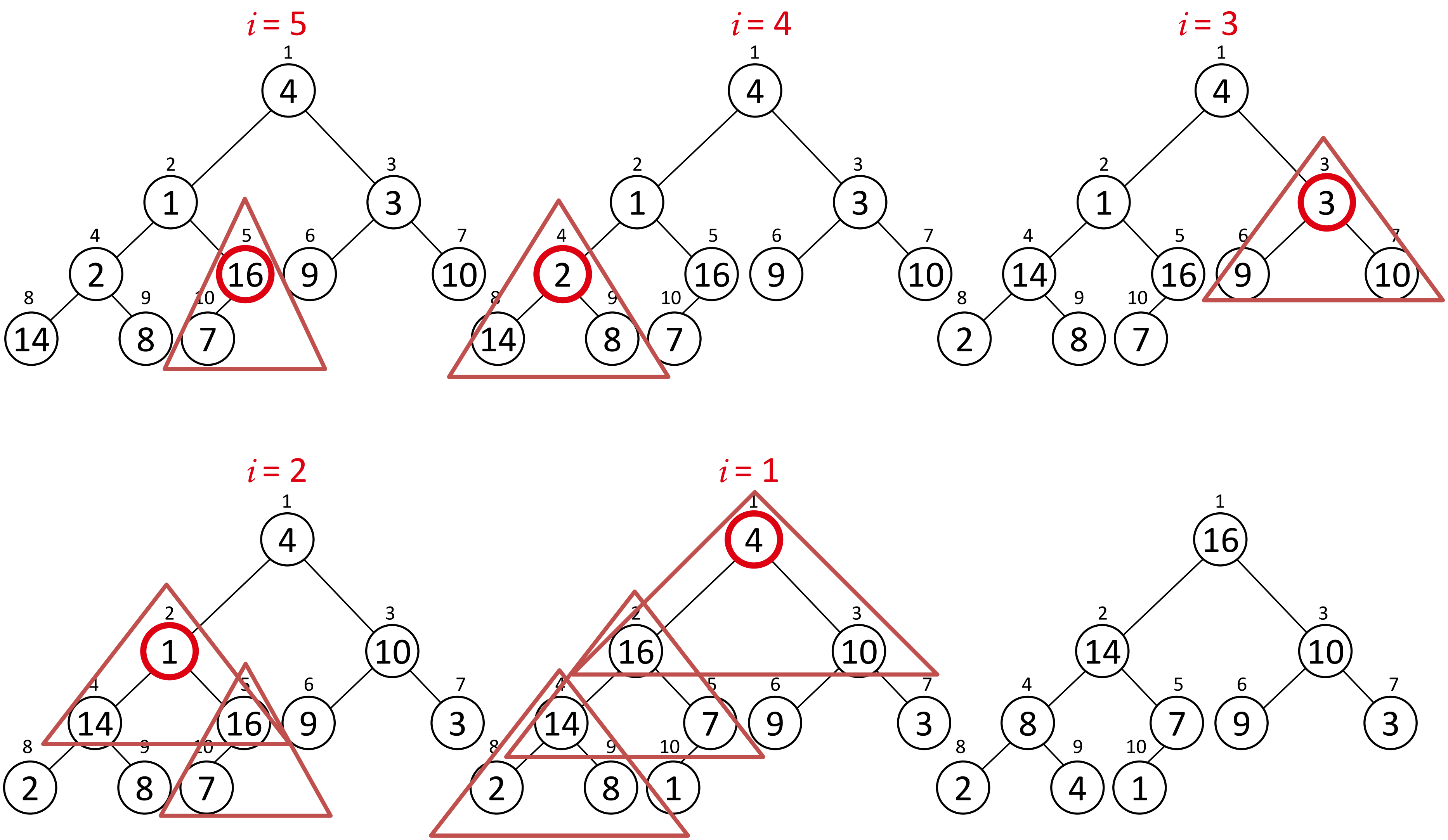


A:

4	1	3	2	16	9	10	14	8	7

Example:

A										
	4	1	3	2	16	9	10	14	8	7



Running Time of BUILD MAX HEAP

BUILD-MAX-HEAP(A)

1. $A.heap\text{-}size = A.length$
2. **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1 $\quad O(n)$
3. MAX-HEAPIFY(A, i) $\quad O(\lg n)$

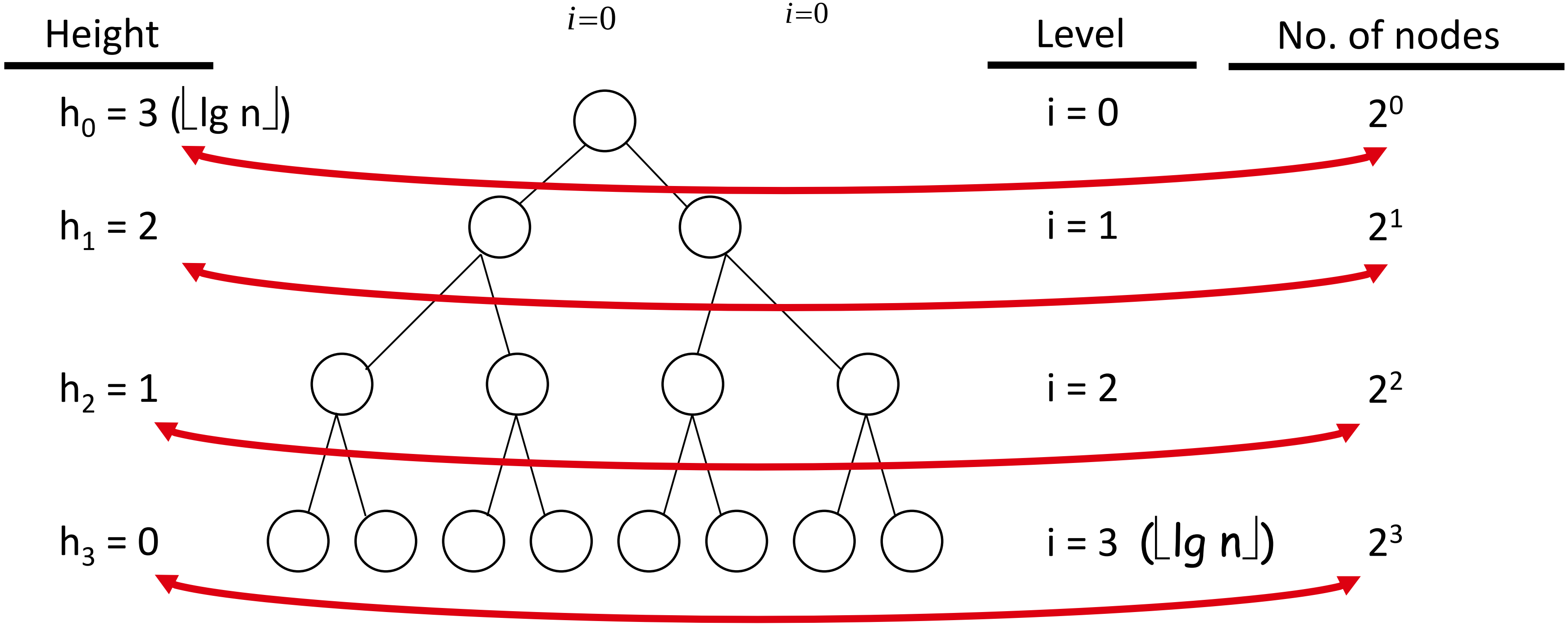
$\Rightarrow$ Running time: $O(n \lg n)$

- This is not an asymptotically tight upper bound

Running Time of BUILD MAX HEAP

- HEAPIFY takes $O(h)$ $\Rightarrow$ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

Heapsort

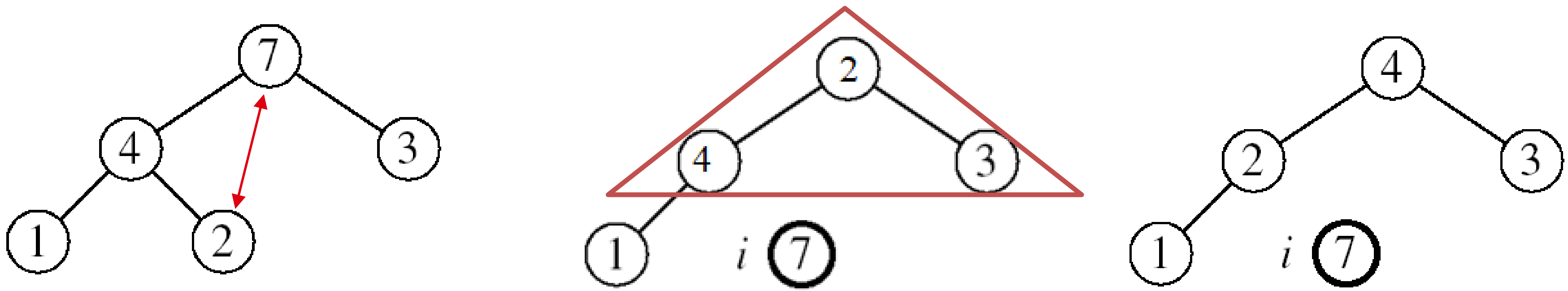
- Goal:
  - Sort an array using heaps.
- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - “Discard” this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

HEAPSORT(*A*)

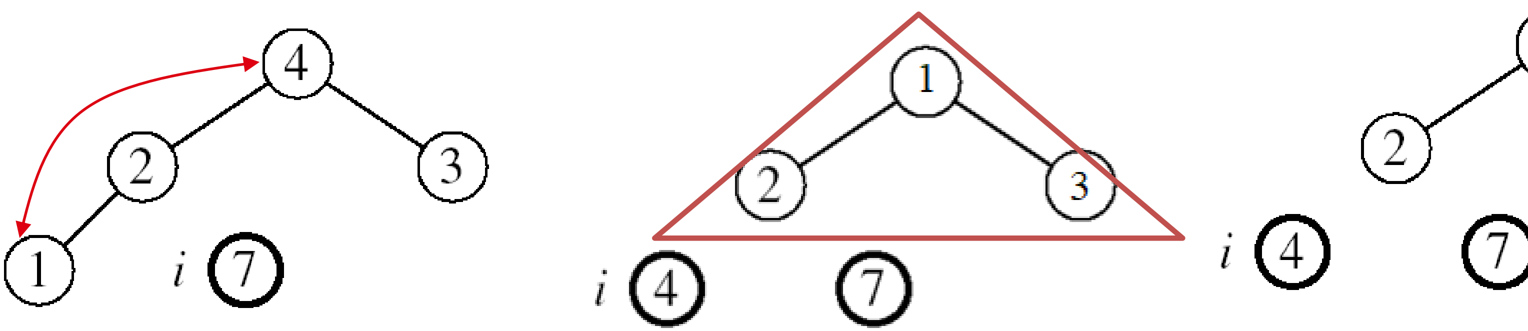
1.	BUILD-MAX-HEAP(*A*)	$O(n)$
2.	**for** i = *A.length* **downto** 2	n – 1 times
3.	**do** exchange *A*[1] with *A*[i]	constant
4.	*A.heap-size* = *A.heap-size* – 1	constant
5.	MAX-HEAPIFY(*A*, 1)	$O(\lg n)$

- Running time: $O(n \lg n)$

Sort: 4, 7, 3, 1, 2 A=[7, 4, 3, 1, 2]



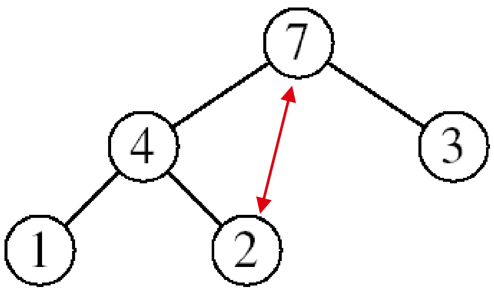
MAX-HEAPIFY(A, 1)



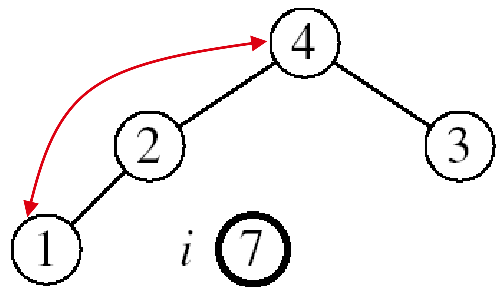
MAX-HEAPIFY(A, 1)

Sort: 4, 7, 3, 1, 2

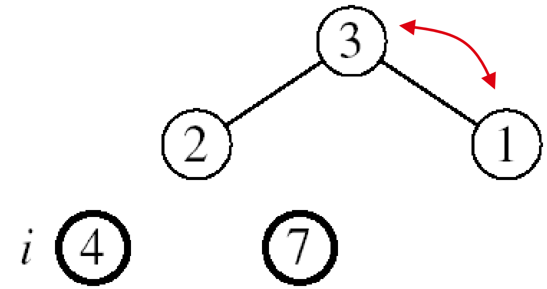
A=[7, 4, 3, 1, 2]



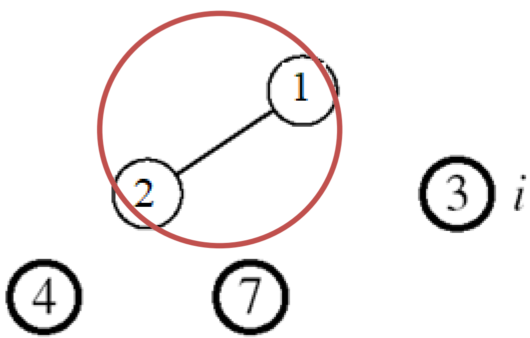
MAX-HEAPIFY(A, 1)

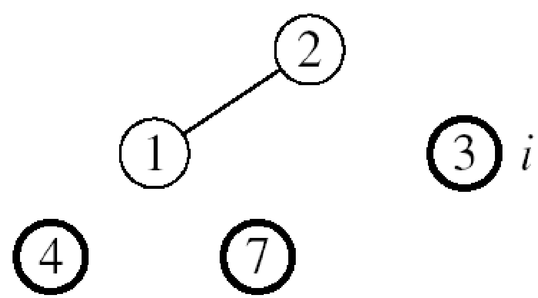


MAX-HEAPIFY(A, 1)

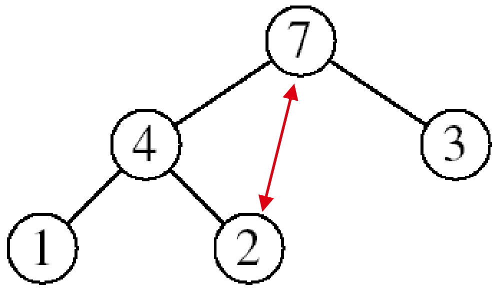


MAX-HEAPIFY(A, 1)

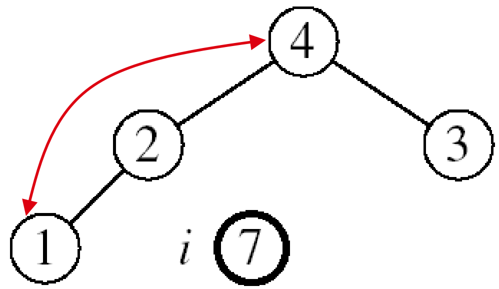




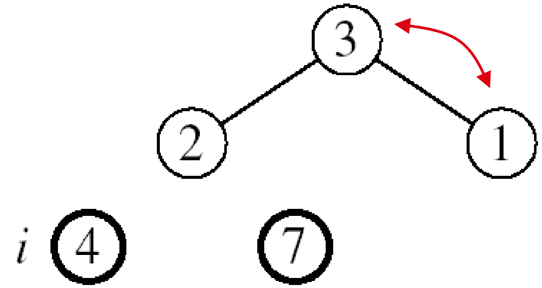
Sort: 4, 7, 3, 1, 2 A=[7, 4, 3, 1, 2]



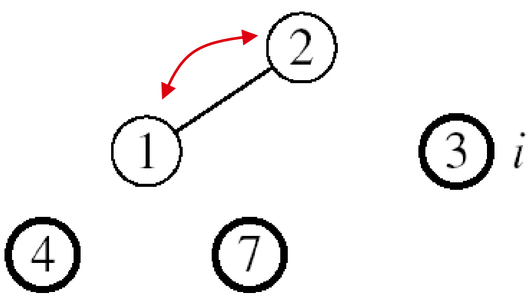
MAX-HEAPIFY(A, 1)



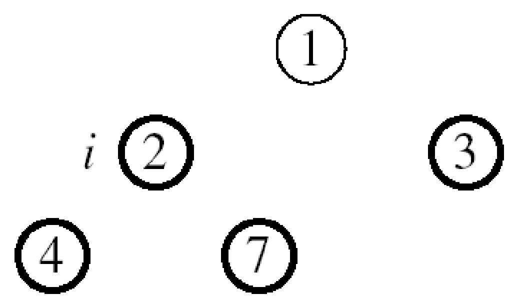
MAX-HEAPIFY(A, 1)



MAX-HEAPIFY(A, 1)



MAX-HEAPIFY(A, 1)



A	1	2	3	4	7

Priority Queues

- Data structure for maintaining a set S of elements, each with an associated value called a key.
- Max-priority queues support the following operations:
  - INSERT(S, x): <u>inserts</u> element x into set S
  - EXTRACT-MAX(S): <u>removes and returns</u> element of S with largest key
  - MAXIMUM(S): <u>returns</u> element of S with largest key
  - INCREASE-KEY(S, x, k): <u>increases</u> value of element x's key to k (Assume $k \geq x$'s current key value)

HEAP-MAXIMUM

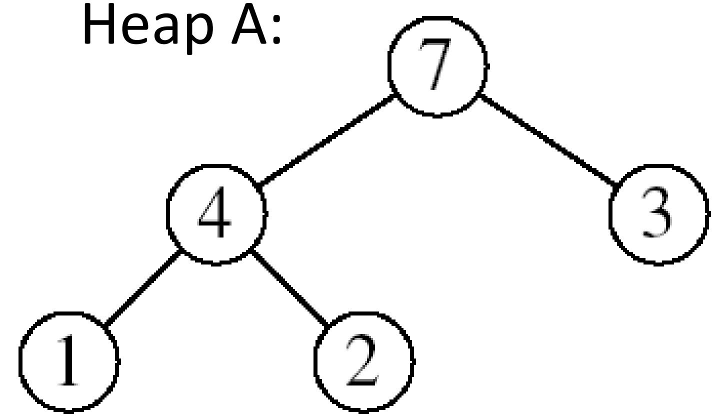
Goal:

- Return the largest element of the heap

HEAP-MAXIMUM(A)

1. **return** $A[1]$

Heap A:



Running time: $O(1)$

Heap-Maximum(A) returns 7

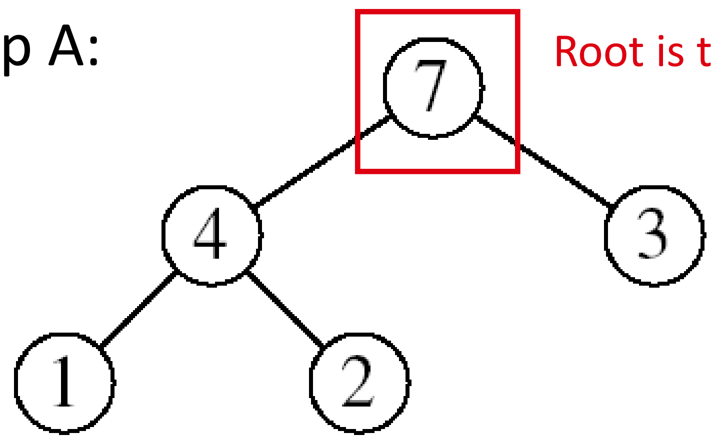
HEAP-EXTRACT-MAX

Goal: Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

Idea:

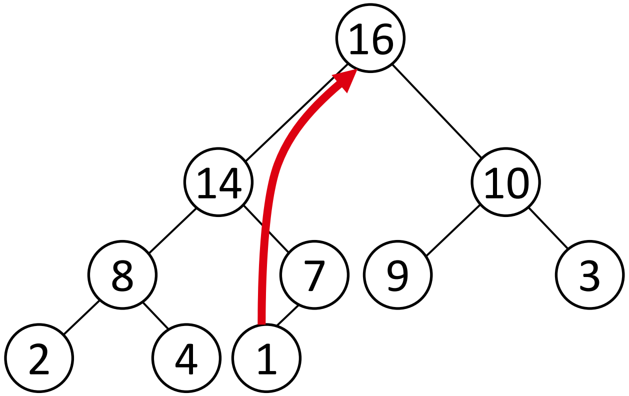
- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root

Heap A:

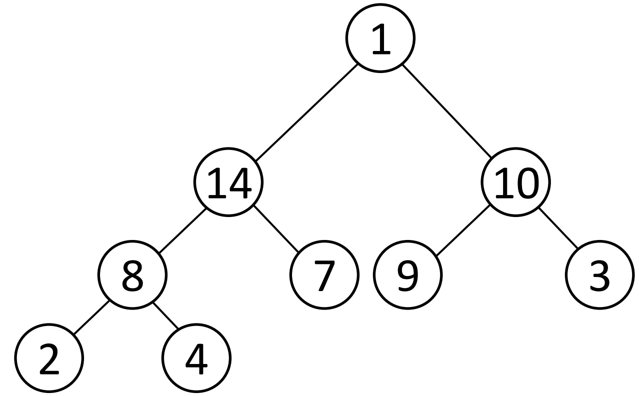


Root is the largest element

Example: HEAP-EXTRACT-MAX

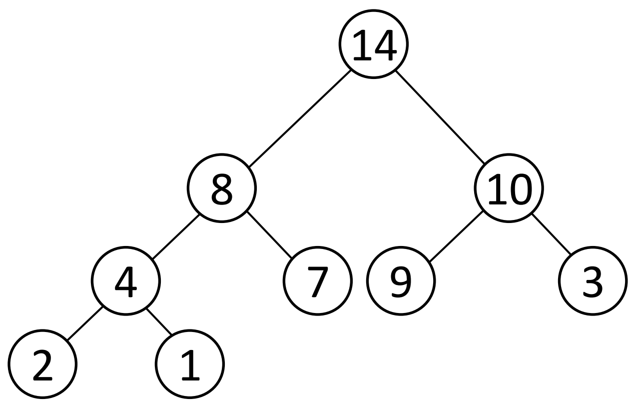


max = 16



Heap size decreased with 1

Call MAX-HEAPIFY$(A, 1)$



HEAP-EXTRACT-MAX

HEAP-EXTRACT-MAX(A)

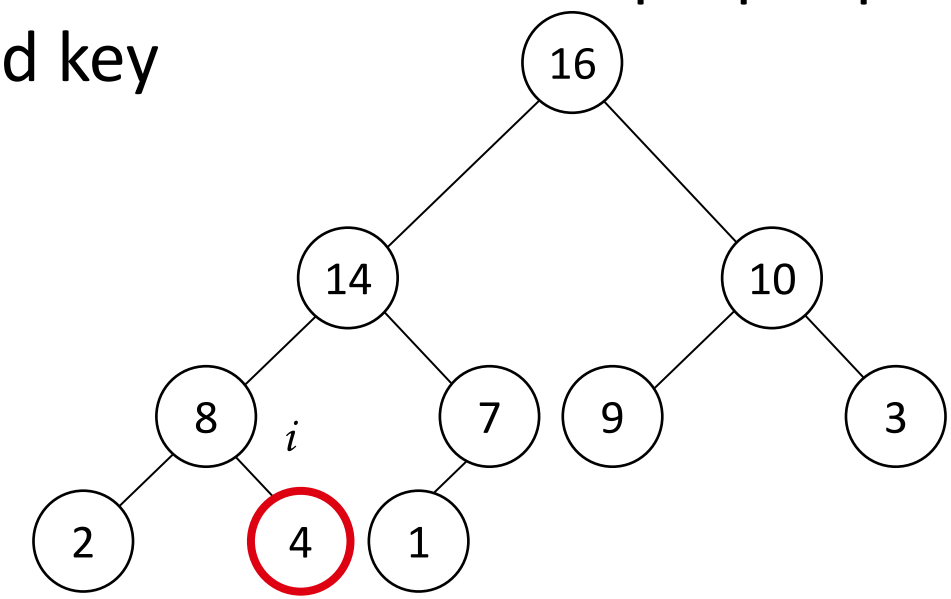
1. **if** A.heap-size < 1
2.   **then error** “heap underflow”
3. $max = A[1]$
4. $A[1] = A[A.\text{heap-size}]$
5. $A.\text{heap-size} = A.\text{heap-size} - 1$
6. MAX-HEAPIFY(A, 1)
7. **return** max

Running time: $O(\lg n)$

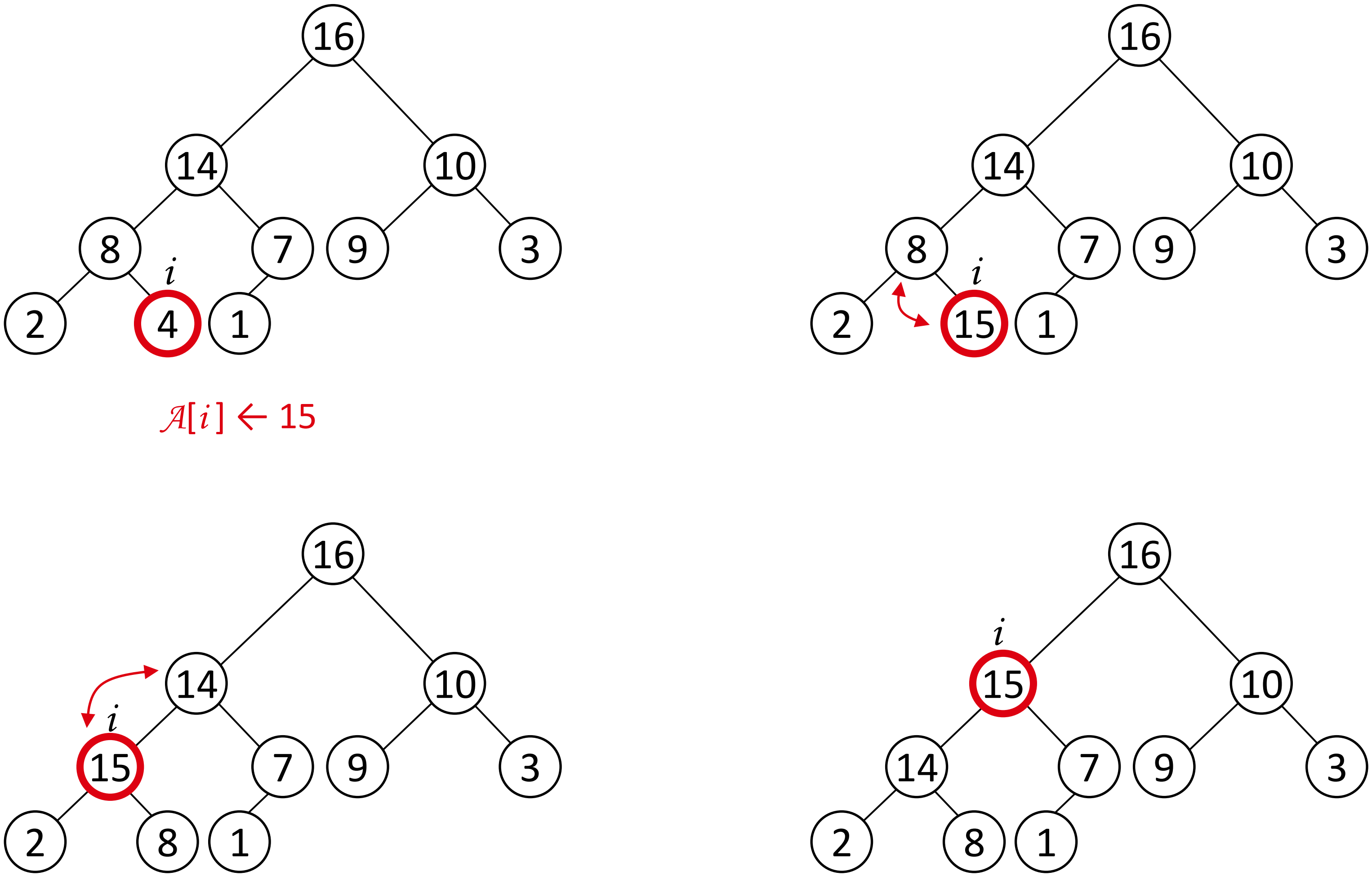
HEAP-INCREASE-KEY

- Goal: Increases the key of an element i in the heap
- Idea:
  - Increment the key of $A[i]$ to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key

$A[i] \leftarrow 15$



Example: HEAP-INCREASE-KEY



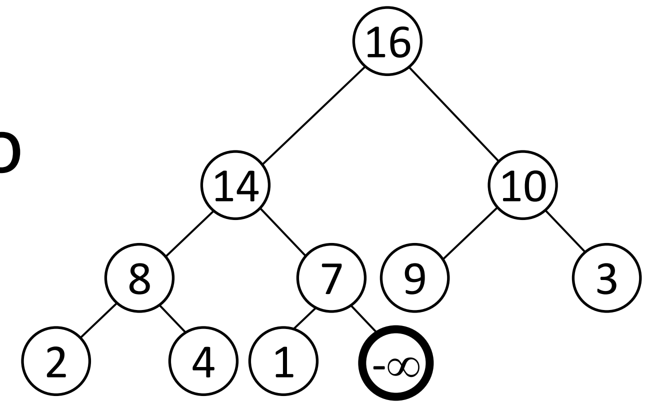
HEAP-INCREASE-KEY

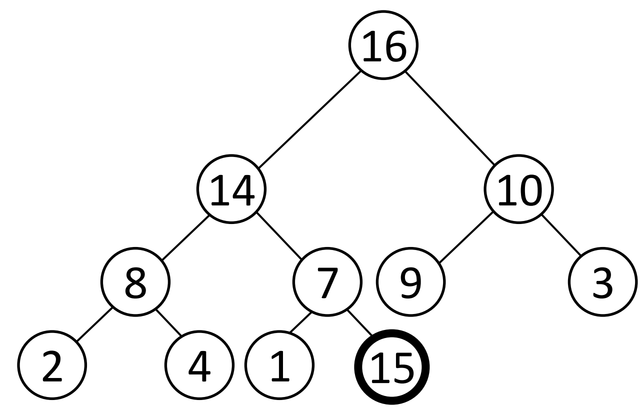
```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
   then error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] with A[PARENT(i)]
         i = PARENT(i)
```

- Running time: $O(\lg n)$

MAX-HEAP-INSERT

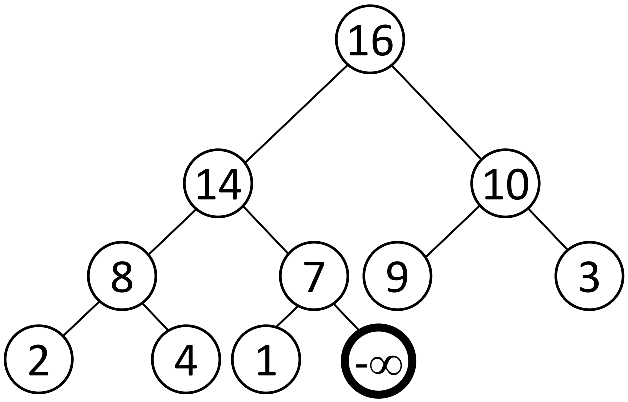
- Goal: Inserts a new element into a max-heap
- Idea:
  - Expand the max-heap with a new element whose key is $-\infty$
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property



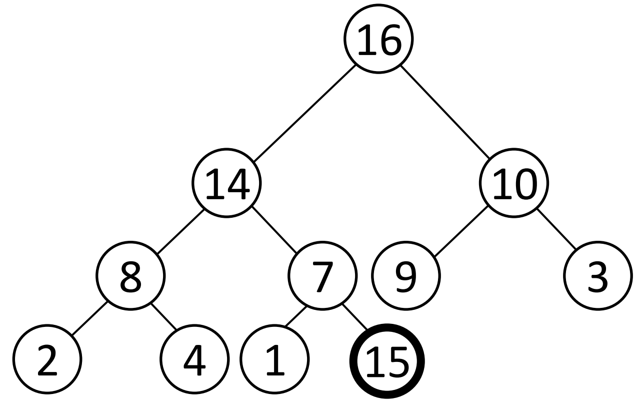


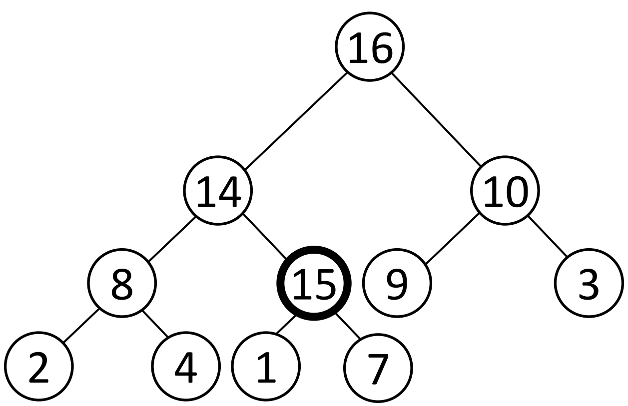
Example: MAX-HEAP-INSERT

Insert value 15:
- Start by inserting -∞

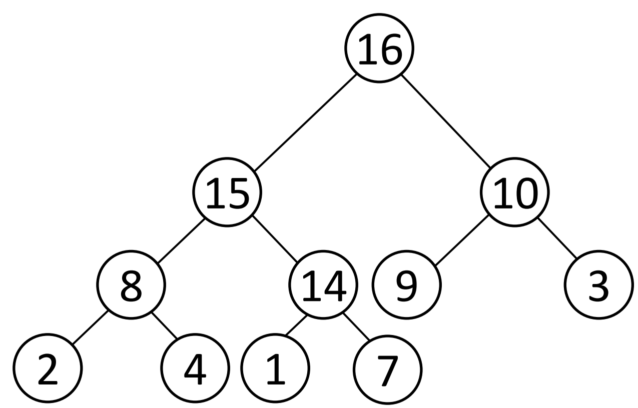


Increase the key to 15
Call HEAP-INCREASE-KEY on A[11] = 15





The restored heap containing
the newly added element



MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, key)

1. $A.heap\text{-}size = A.heap\text{-}size + 1$
2. $A[A.heap\text{-}size] = -\infty$
3. HEAP-INCREASE-KEY(A, $A.heap\text{-}size$, key)

Running time: $O(\lg n)$

Applications

- Heap sort
- Priority queues: Query for minimum or maximum value in a dynamic collection of values.
- Dijkstra's algorithm for finding the shortest path between a pair of nodes uses heap to pick the closest unexplored node at each iteration to continue the search from it.
  Example: routing of network packets between two nodes.
- Prim's algorithm for finding the Minimum Spanning Tree uses heap to select a new minimum-cost edge that expands your current minimum spanning tree.
  Example: wire layout for a service network, such as electricity or cable. Aim is to provide service coverage to an entire area with the minimum wiring cost possible.
- Huffman encoding (data compression).
- Used by an operating system for dynamic memory allocation.

Thank you