

Heaps

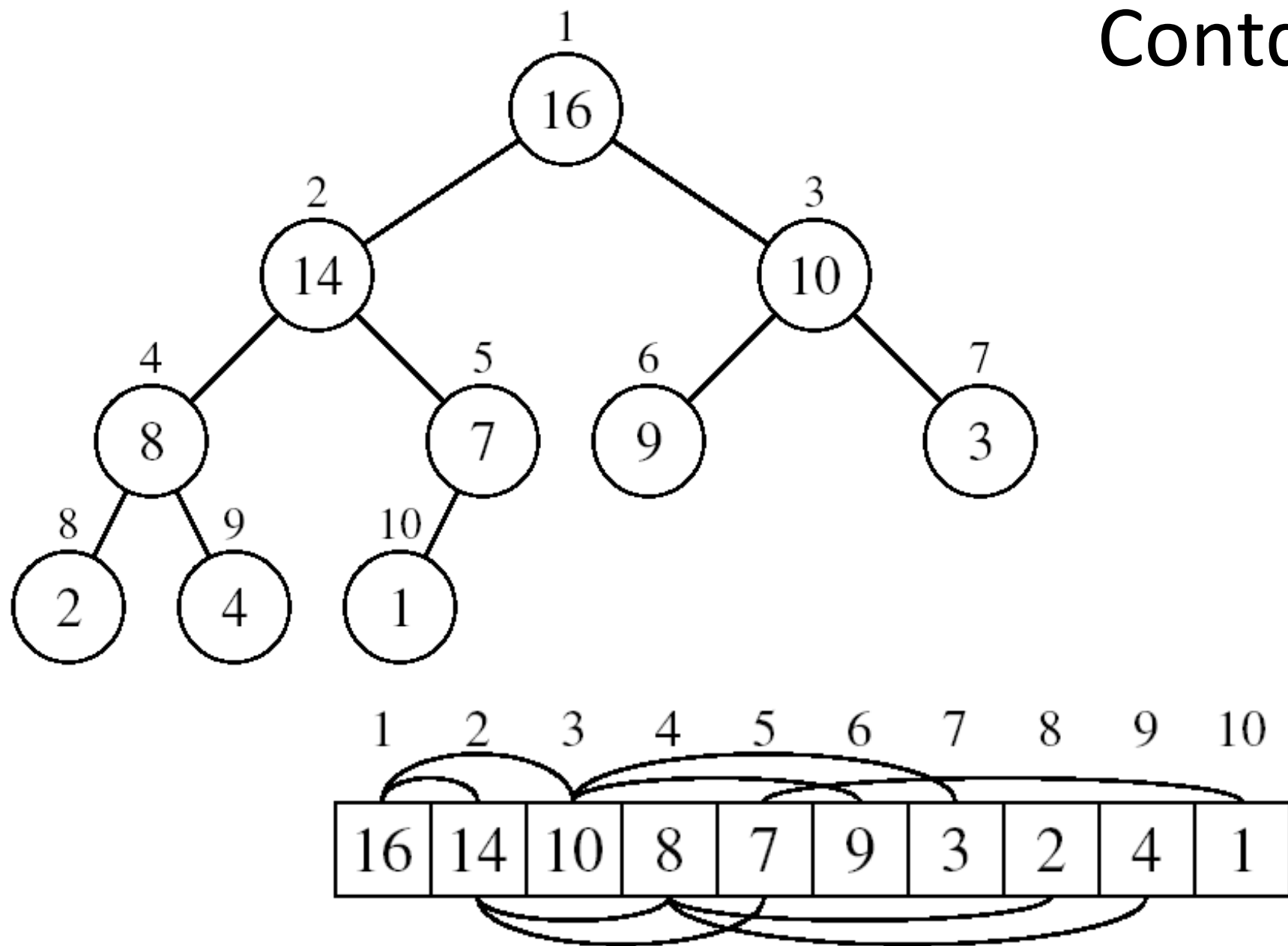
Heap Data Structure

- 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 $A = A.heap-size$
 - $0 \leq A.heap-size \leq A.length$
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves

Contd...

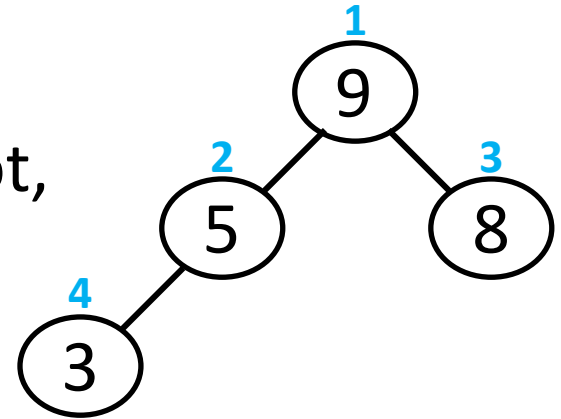


Types of Heaps

- **Max-heaps** (largest element at root), satisfy the *max-heap property*:

- for every node i other than the root,

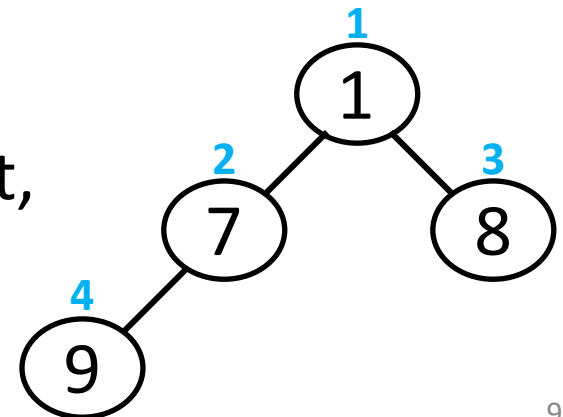
$$A[\text{PARENT}(i)] \geq A[i]$$



- **Min-heaps** (smallest element at root), satisfy the *min-heap property*:

- for every node i other than the root,

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

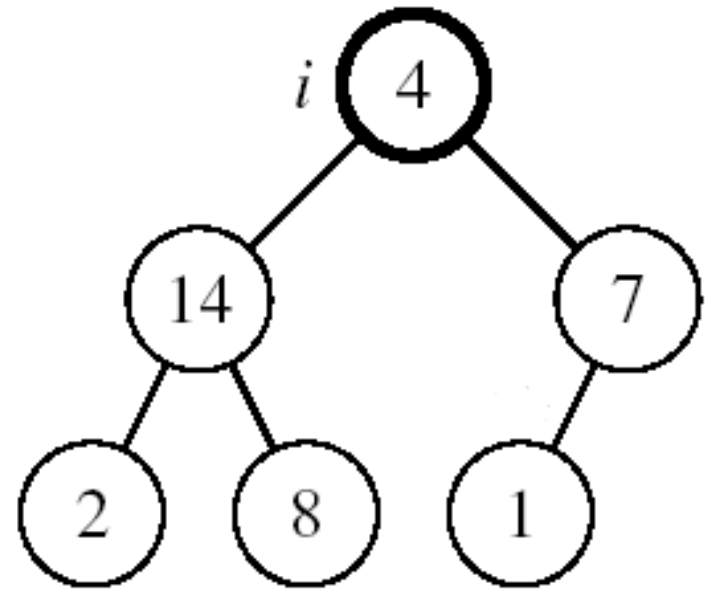


Operations on Heaps

- 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



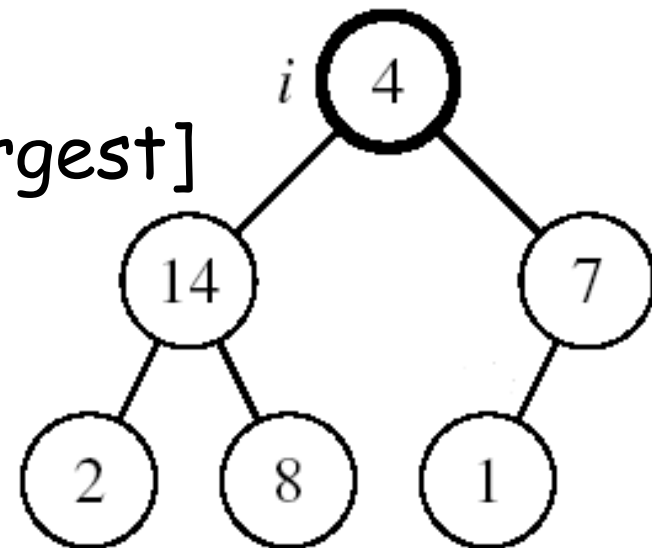
Maintaining the Heap Property

MAX-HEAPIFY(A, i)

1. $l = \text{LEFT}(i)$
2. $r = \text{RIGHT}(i)$
3. if $l \leq A.\text{heap-size}$ and $A[l] > A[i]$
4. then $\text{largest} = l$
5. else $\text{largest} = i$
6. if $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$
7. then $\text{largest} = r$
8. if $\text{largest} \neq i$
9. then exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}$)

Assumptions:

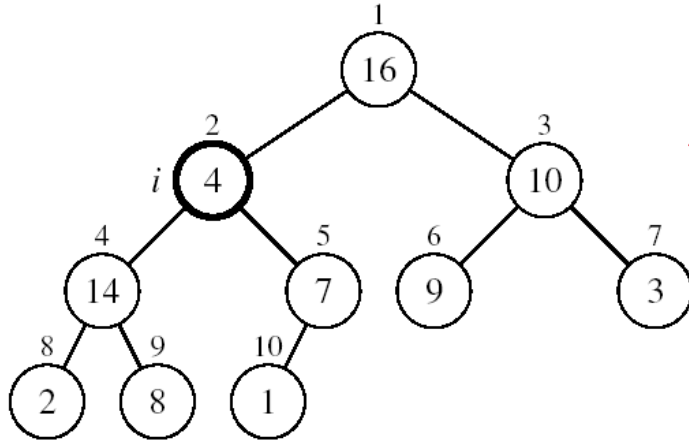
- Left and Right subtrees of i are max-heaps
- $A[i]$ may be smaller than its children



Example

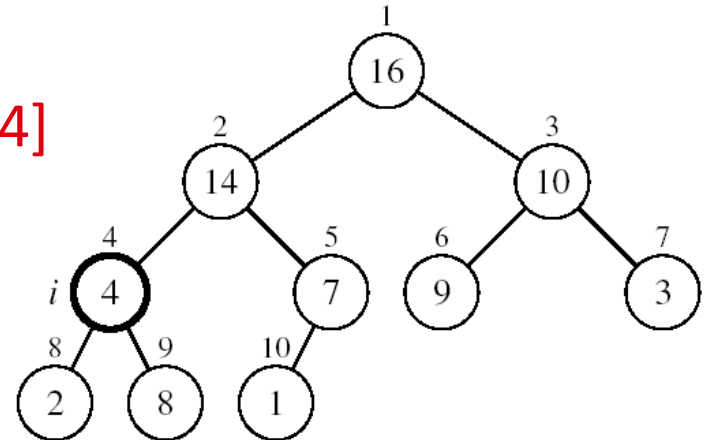
$A.\text{heap-size} = 10$
 $A.\text{length} = 10$

MAX-HEAPIFY(A, 2)



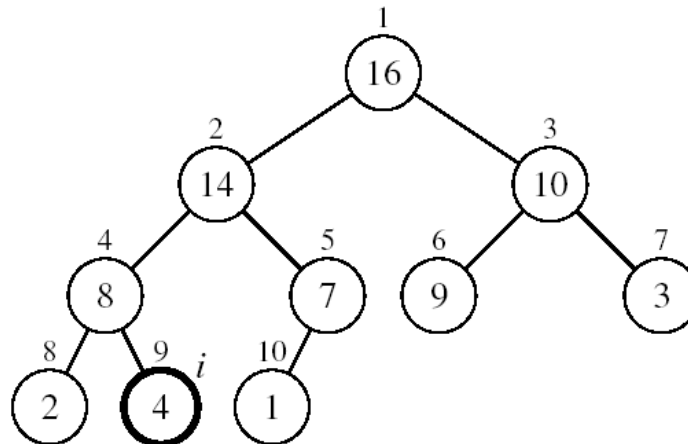
$A[2]$ violates the heap property

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



$A[4]$ violates the heap property

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



Heap property restored

MAX-HEAPIFY Running Time

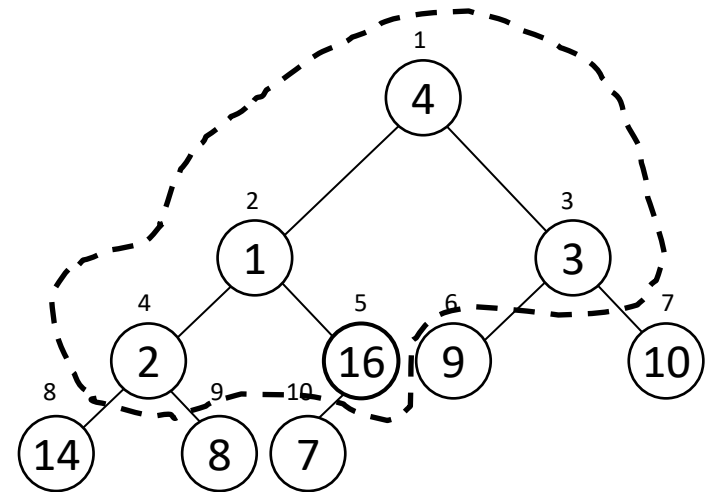
- It traces path from root to a leaf.
- In worst case length of path is h .
- Running time of MAX-HEAPIFY is $O(h)$ or $O(\lg n)$
 - Since the height of the heap is $\lg n$.

Building a Heap

- Convert an array $A[1 \dots n]$, where $n = A.length$ into a max-heap.
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \dots n]$ 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)



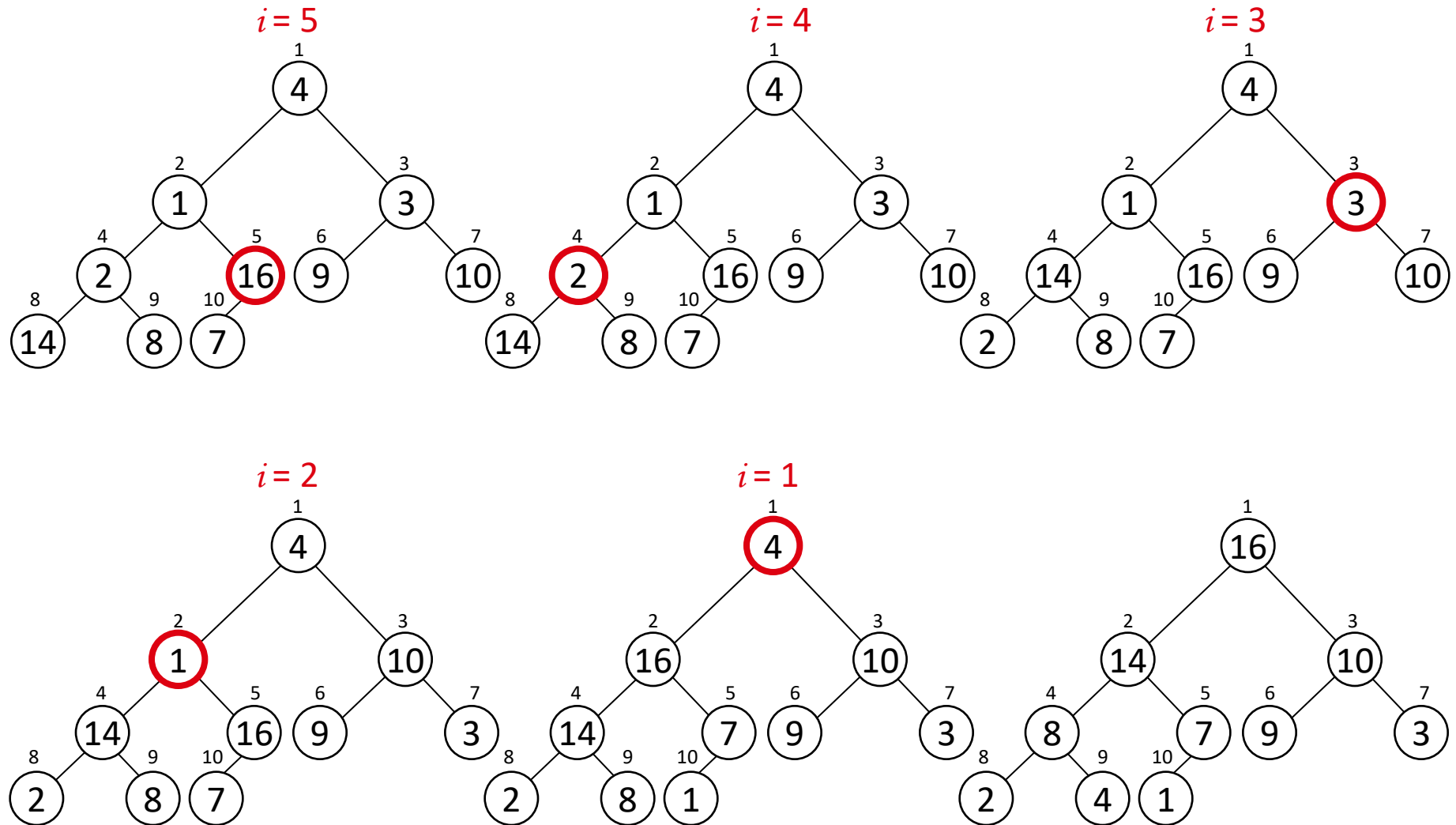
A:

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

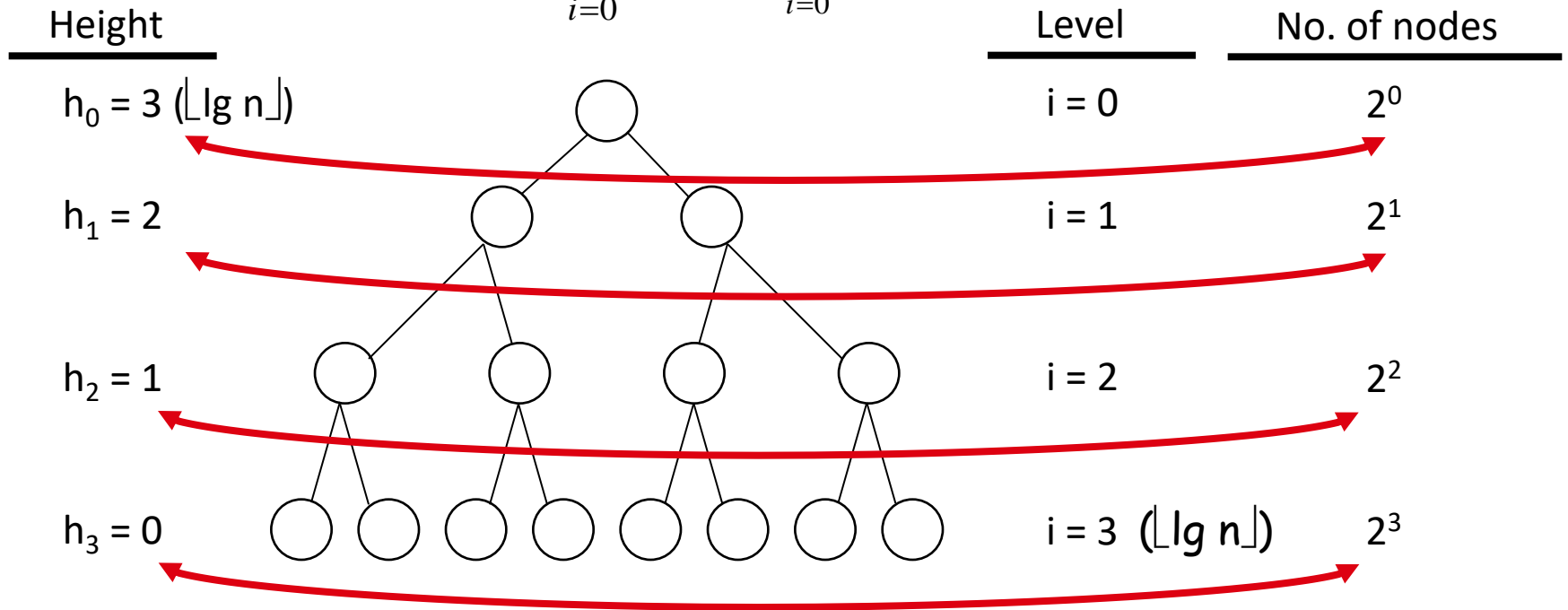
\Rightarrow Running time: $O(n \lg n)$

- 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

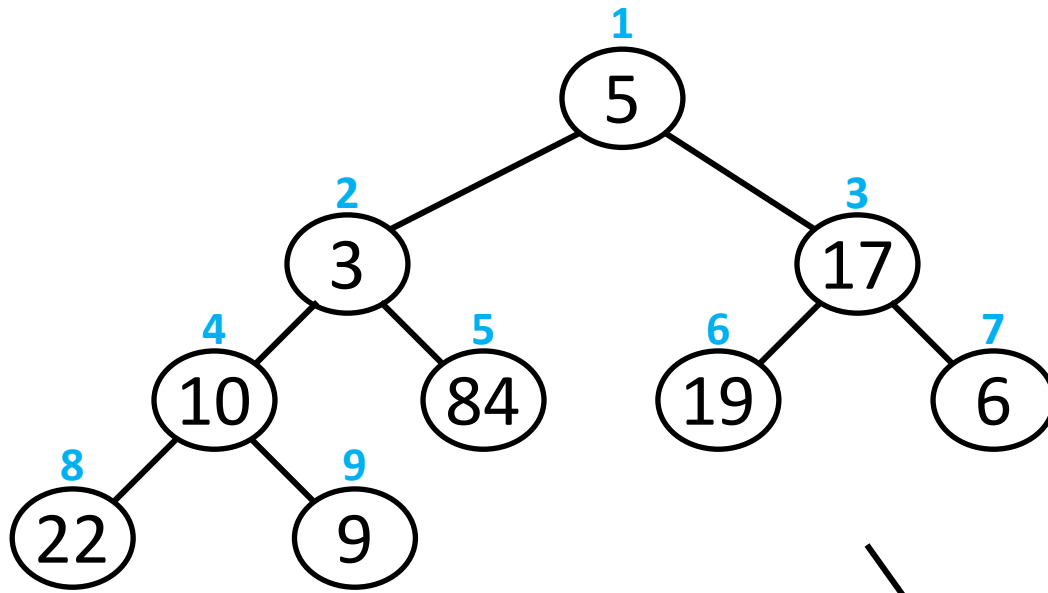
Running Time of BUILD MAX HEAP

$$\begin{aligned} T(n) &= \sum_{i=0}^h n_i h_i && \text{Cost of HEAPIFY at level } i * \text{number of nodes at that level} \\ &= \sum_{i=0}^h 2^i (h - i) && \text{Replace the values of } n_i \text{ and } h_i \text{ computed before} \\ &= \sum_{i=0}^h \frac{h-i}{2^{h-i}} 2^h && \text{Multiply by } 2^h \text{ both at the nominator and denominator and} \\ &&& \text{write } 2^i \text{ as } \frac{1}{2^{-i}} \\ &= 2^h \sum_{k=0}^h \frac{k}{2^k} && \text{Change variables: } k = h - i \\ &\leq n \sum_{k=0}^{\infty} \frac{k}{2^k} && \text{The sum above is smaller than the sum of all elements to } \infty \\ &&& \text{and } h = \lg n \\ &= O(n) && \text{The sum above is smaller than 2} \end{aligned}$$

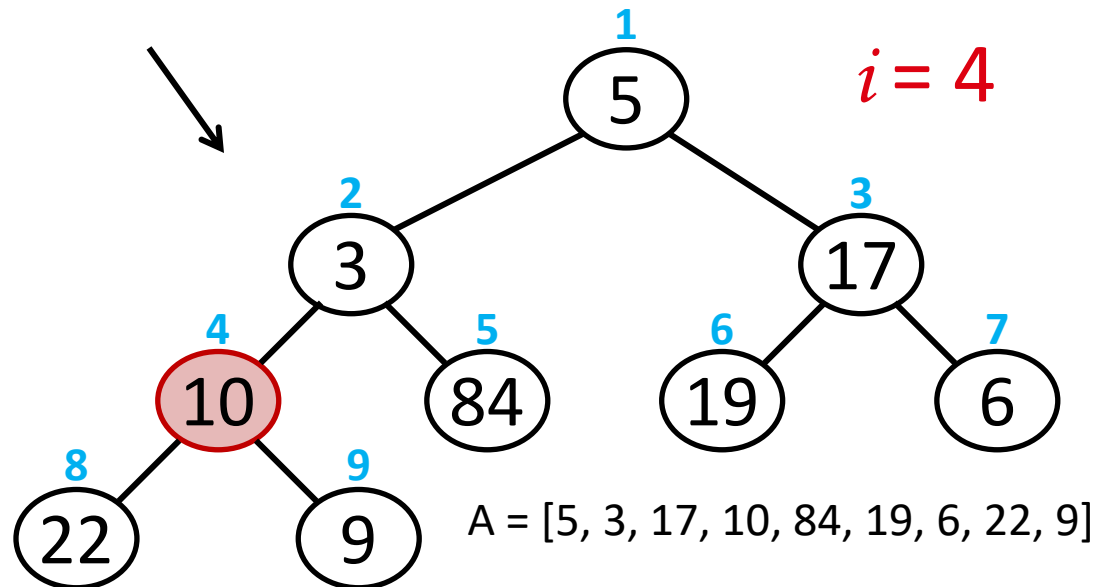
Running time of BUILD-MAX-HEAP: $T(n) = O(n)$

Build Max-heap

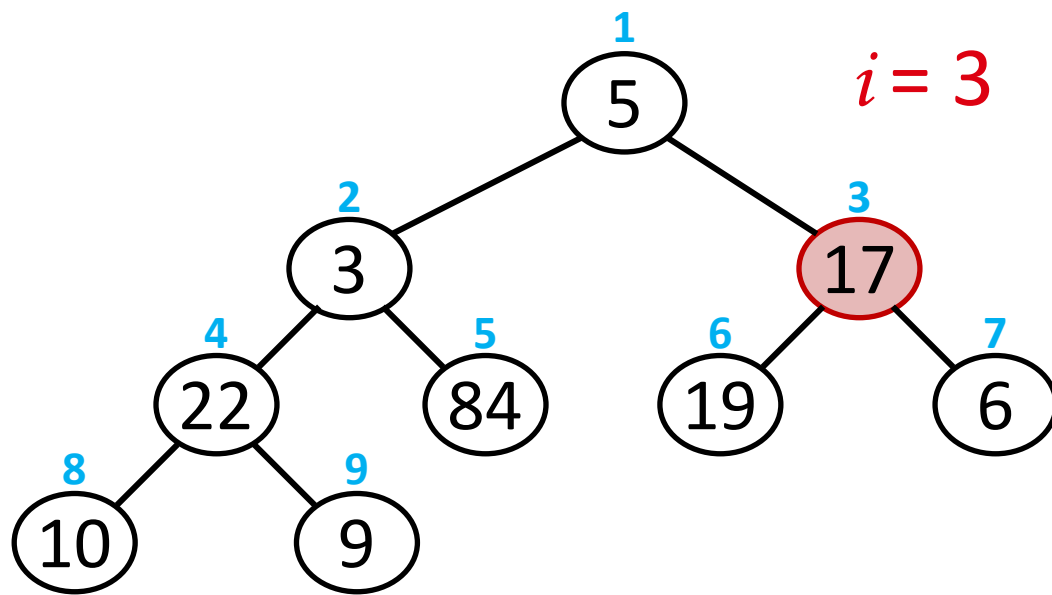
$A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$



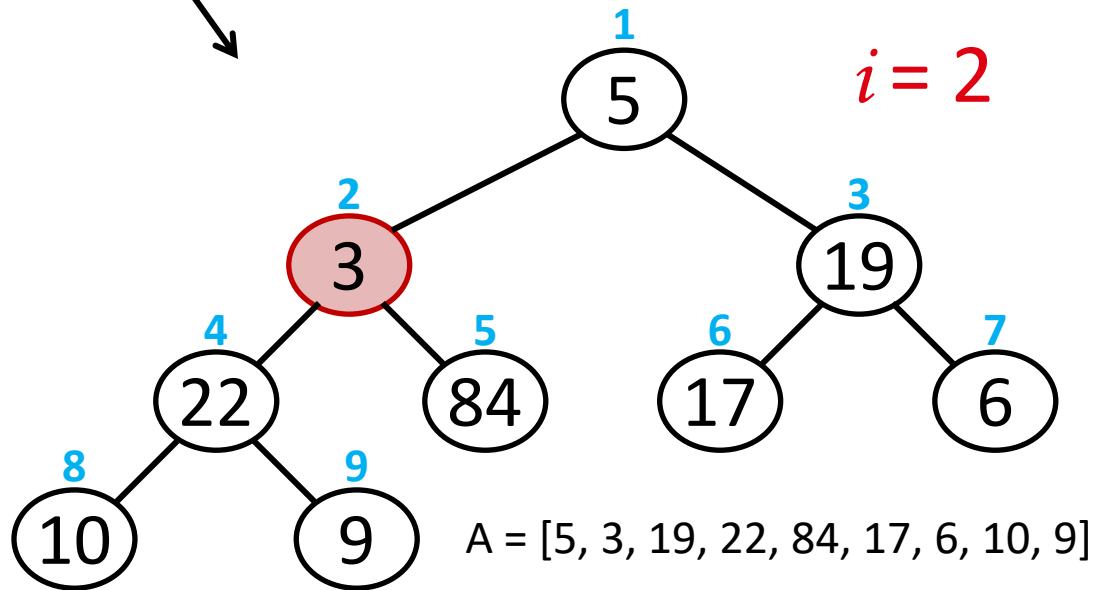
$A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$



Contd...

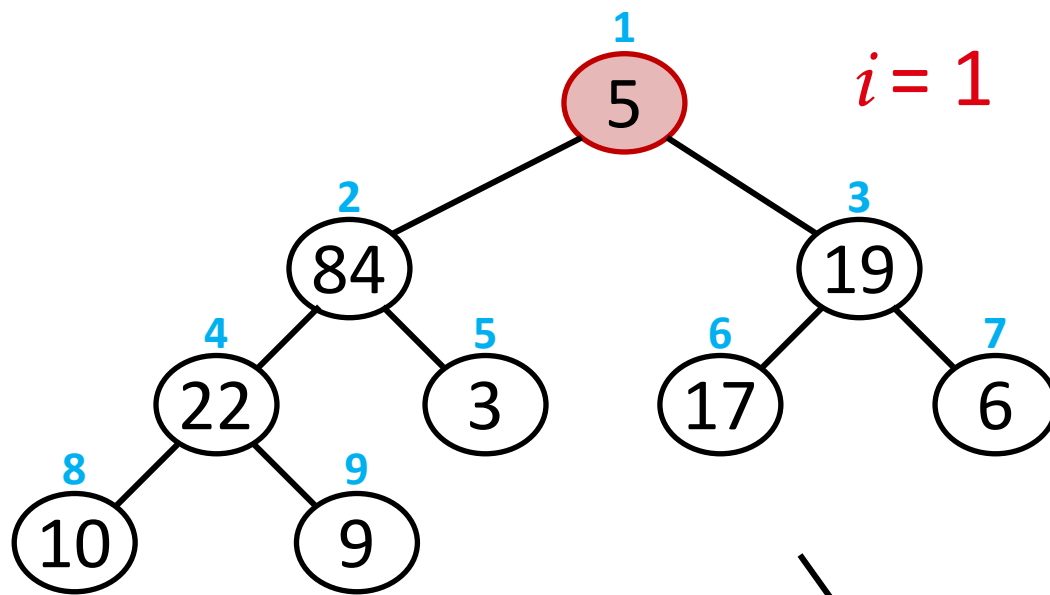


$A = [5, 3, 17, 22, 84, 19, 6, 10, 9]$

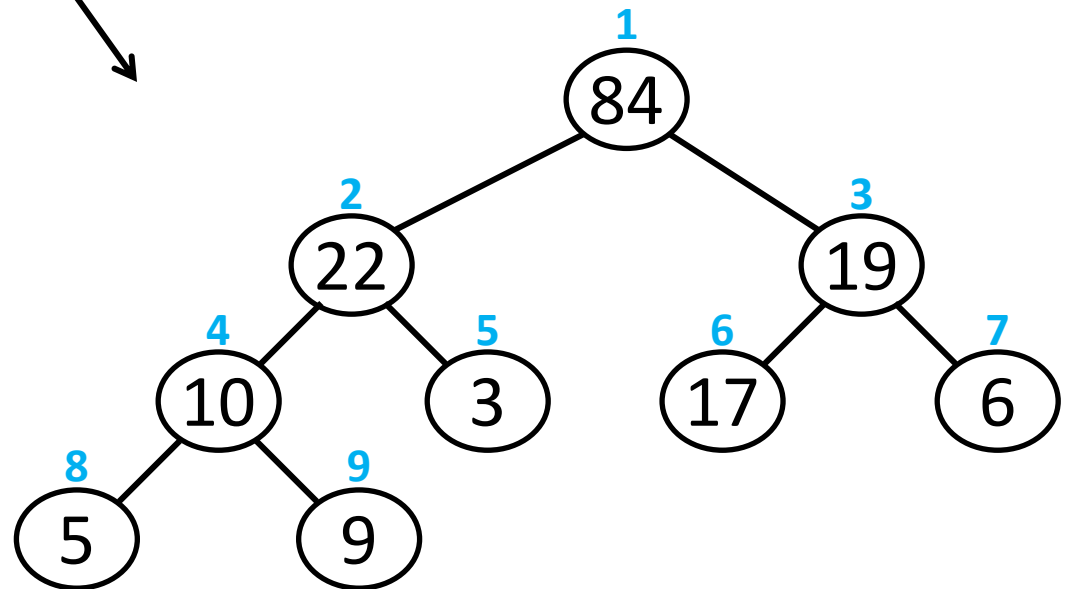


$A = [5, 3, 19, 22, 84, 17, 6, 10, 9]$

Contd...



$A = [5, 84, 19, 22, 3, 17, 6, 10, 9]$



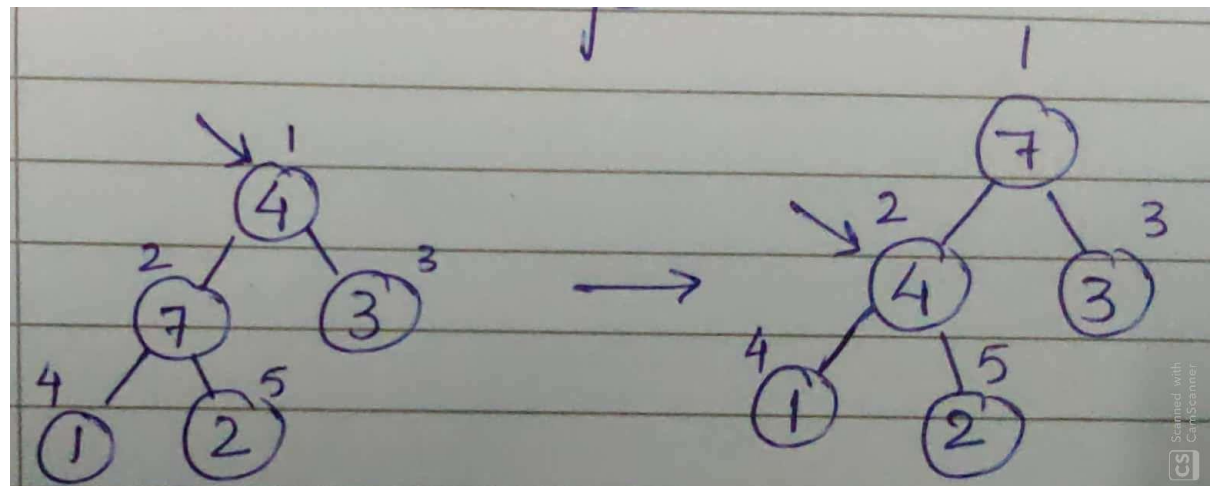
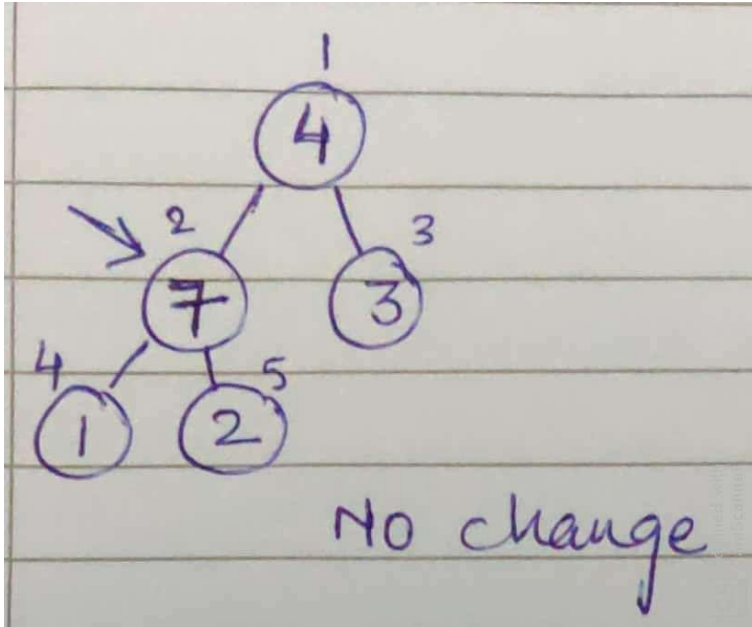
$A = [84, 22, 19, 10, 3, 17, 6, 5, 9]$

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

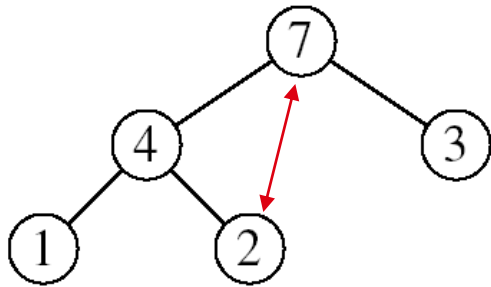
Sort: 4, 7, 3, 1, 2

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

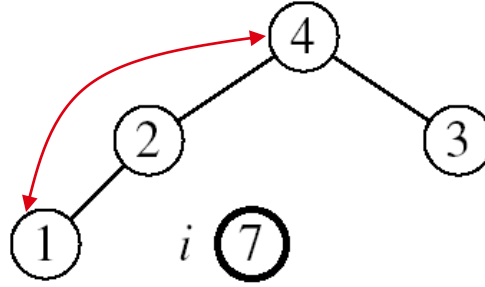


Sort: 4, 7, 3, 1, 2

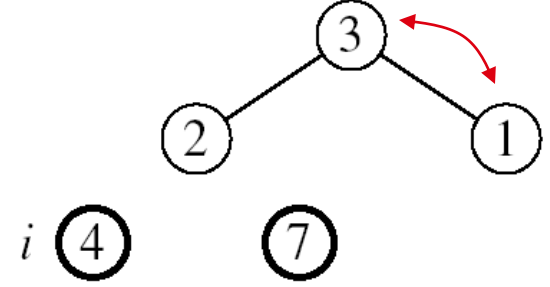
$A=[7, 4, 3, 1, 2]$



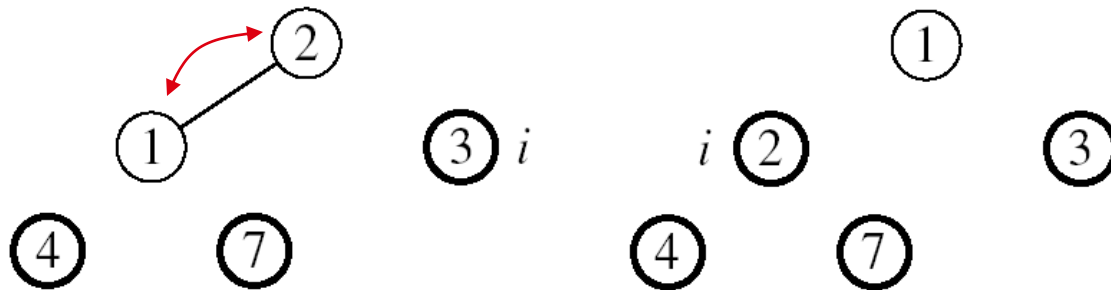
MAX-HEAPIFY(A, 1)



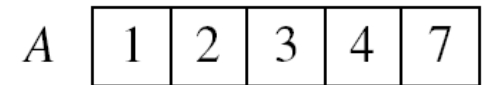
MAX-HEAPIFY(A, 1)



MAX-HEAPIFY(A, 1)



MAX-HEAPIFY(A, 1)



HEAPSORT(*A*)

- | | | |
|----|---|---------------|
| 1. | BUILD-MAX-HEAP(<i>A</i>) | $O(n)$ |
| 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(<i>A</i> , 1) | $O(\lg n)$ |
-
- Running time: $O(n \lg n)$

Priority Queues

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:
 - $\text{INSERT}(S, x)$: inserts element x into set S
 - $\text{EXTRACT-MAX}(S)$: removes and returns element of S with the largest key
 - $\text{MAXIMUM}(S)$: returns element of S with the largest key
 - $\text{INCREASE-KEY}(S, x, k)$: increases value of element x 's key to k (Assume $k \geq x$'s current key value)

HEAP-MAXIMUM

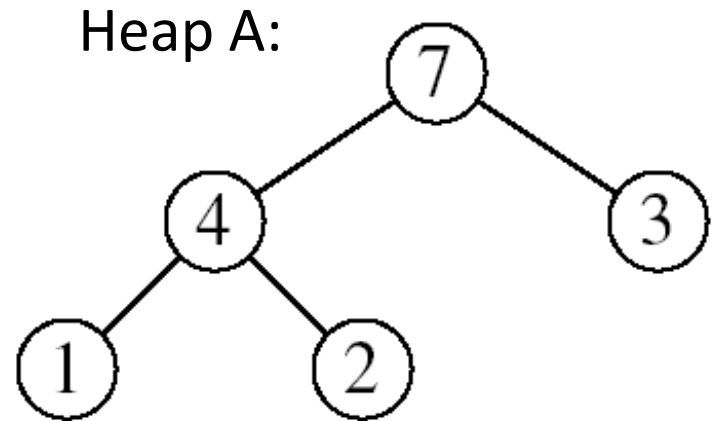
Goal:

- Return the largest element of the heap

HEAP-MAXIMUM(A)

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

Running time: $O(1)$



Heap-Maximum(A) returns 7

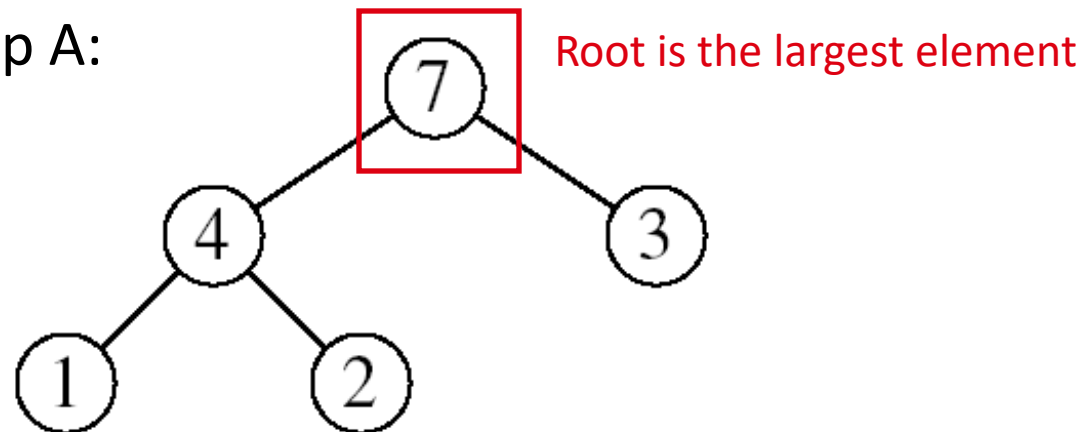
HEAP-EXTRACT-MAX

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:



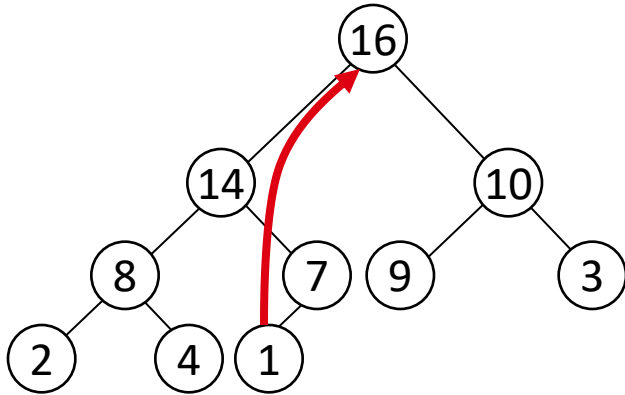
HEAP-EXTRACT-MAX

HEAP-EXTRACT-MAX(A)

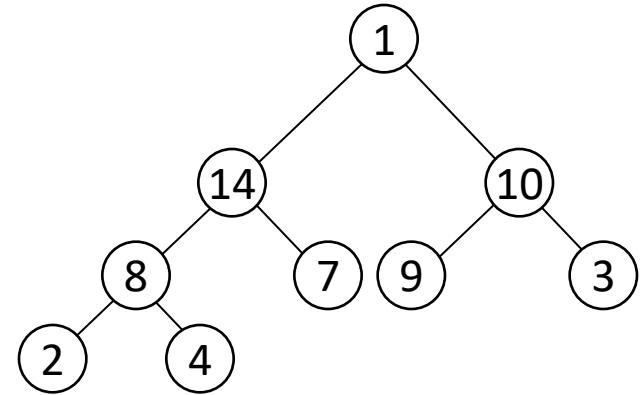
1. **if** $A.\text{heap-size} < 1$
2. **then error** "heap underflow"
3. $\text{max} = A[1]$
4. $A[1] = A[A.\text{heap-size}]$
5. $A.\text{heap-size} = A.\text{heap-size} - 1$
6. $\text{MAX-HEAPIFY}(A, 1)$
7. **return** max

Running time: $O(\lg n)$

Example: HEAP-EXTRACT-MAX

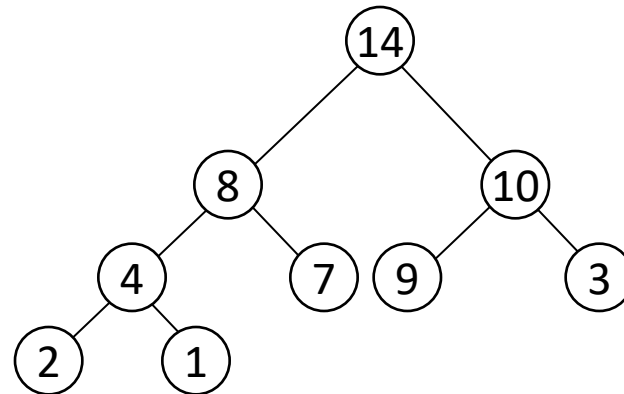


max = 16



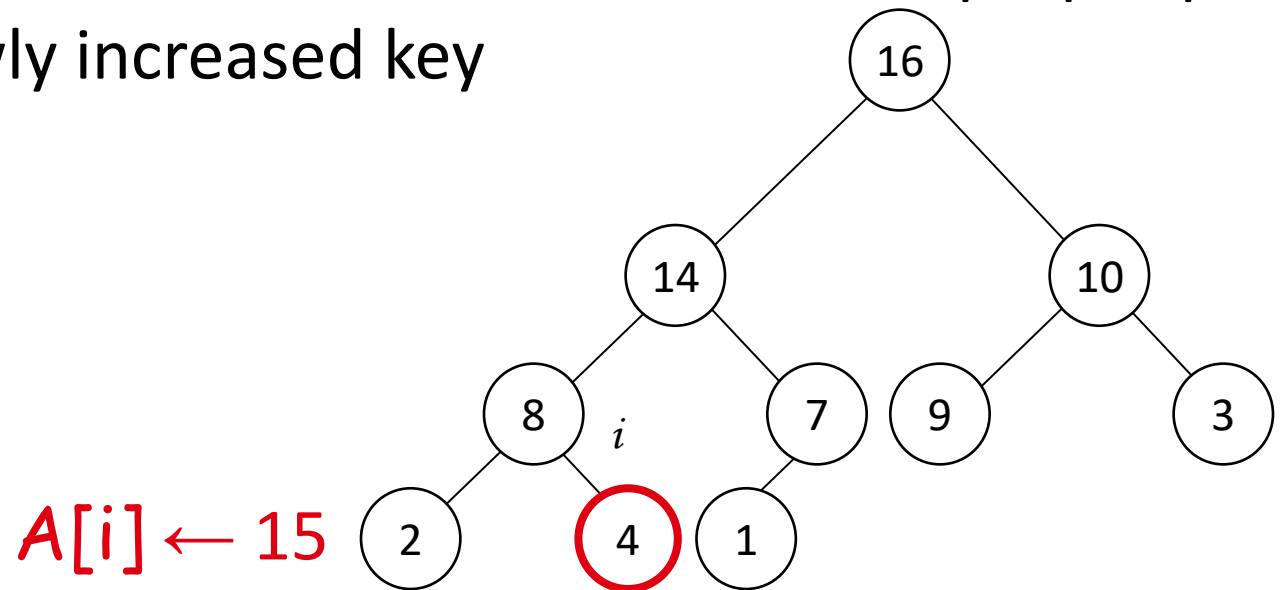
Heap size decreased with 1

Call MAX-HEAPIFY(A, 1)



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

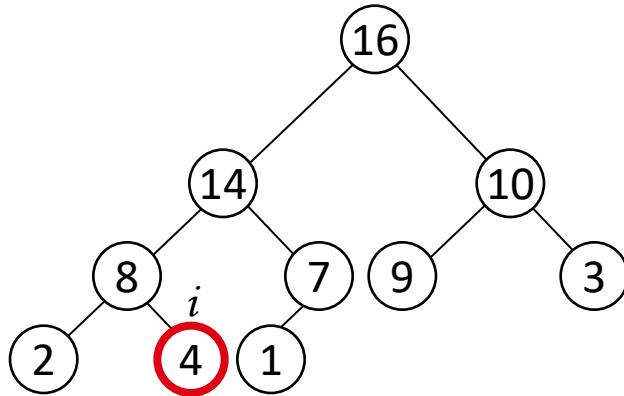


HEAP-INCREASE-KEY

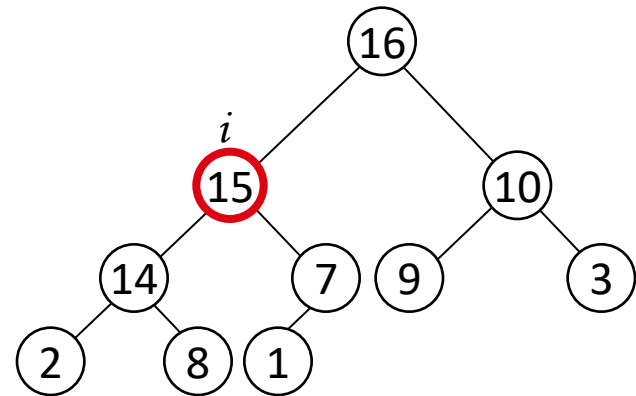
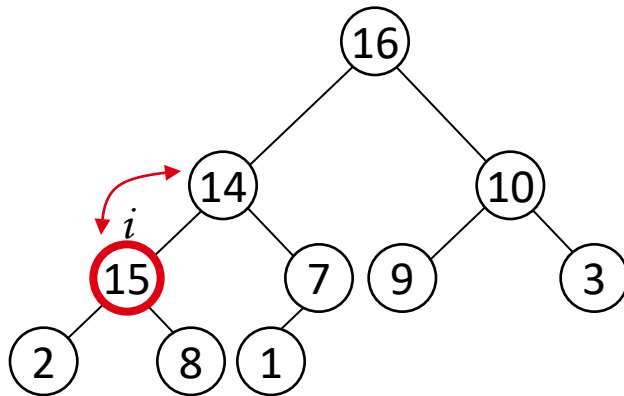
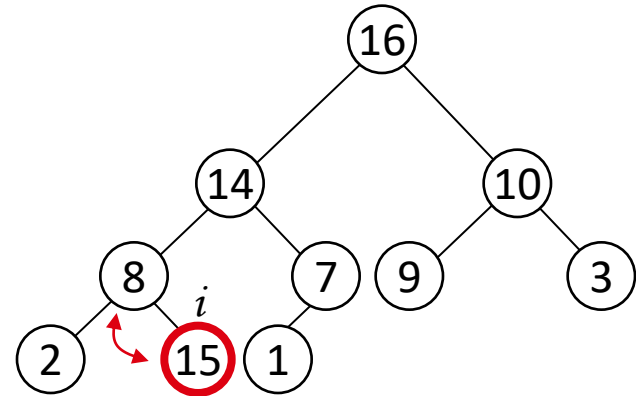
HEAP-INCREASE-KEY(A, i, key)

1. **if** $\text{key} < A[i]$
 2. **then error** “new key is smaller than current key”
 3. $A[i] = \text{key}$
 4. **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
 5. **do** exchange $A[i]$ with $A[\text{PARENT}(i)]$
 6. $i = \text{PARENT}(i)$
- Running time: $O(\lg n)$

Example: HEAP-INCREASE-KEY

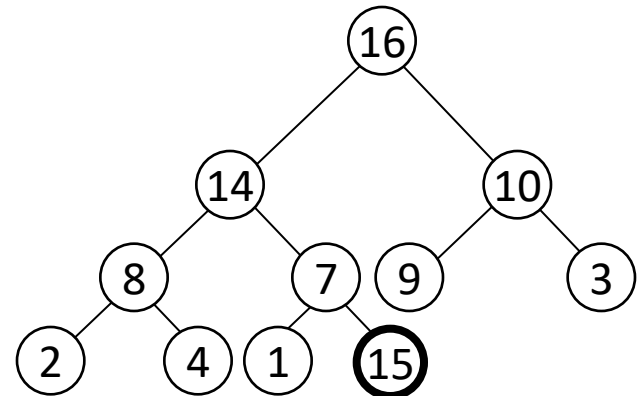
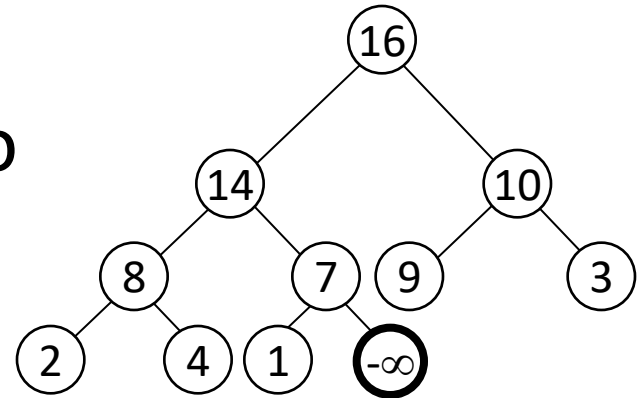


$\mathcal{A}[i] \leftarrow 15$



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



MAX-HEAP-INSERT

MAX-HEAP-INSERT(A , key)

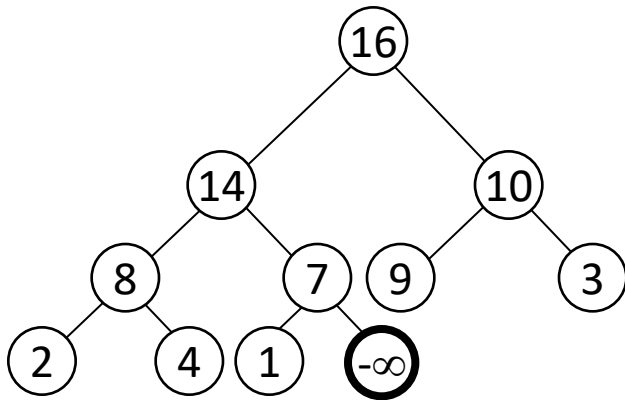
1. $A.heap-size = A.heap-size + 1$
2. $A[A.heap-size] = -\infty$
3. HEAP-INCREASE-KEY(A , $A.heap-size$, key)

Running time: $O(\lg n)$

Example: MAX-HEAP-INSERT

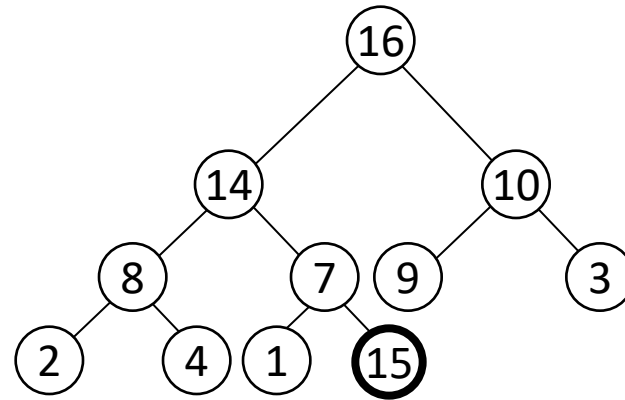
Insert value 15:

- Start by inserting $-\infty$

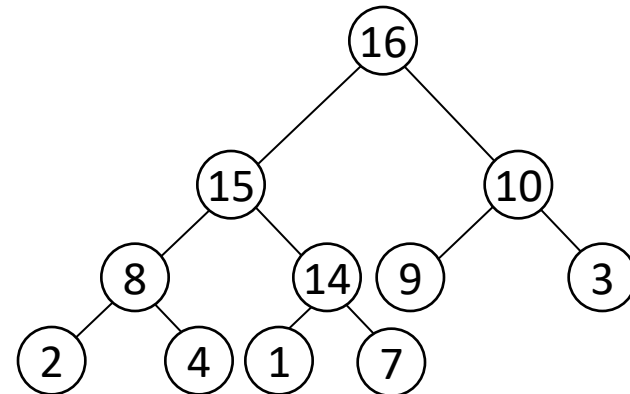
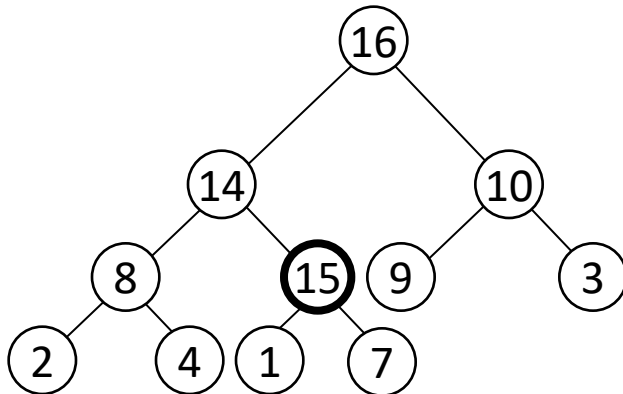


Increase the key to 15

Call HEAP-INCREASE-KEY on $A[11] = 15$

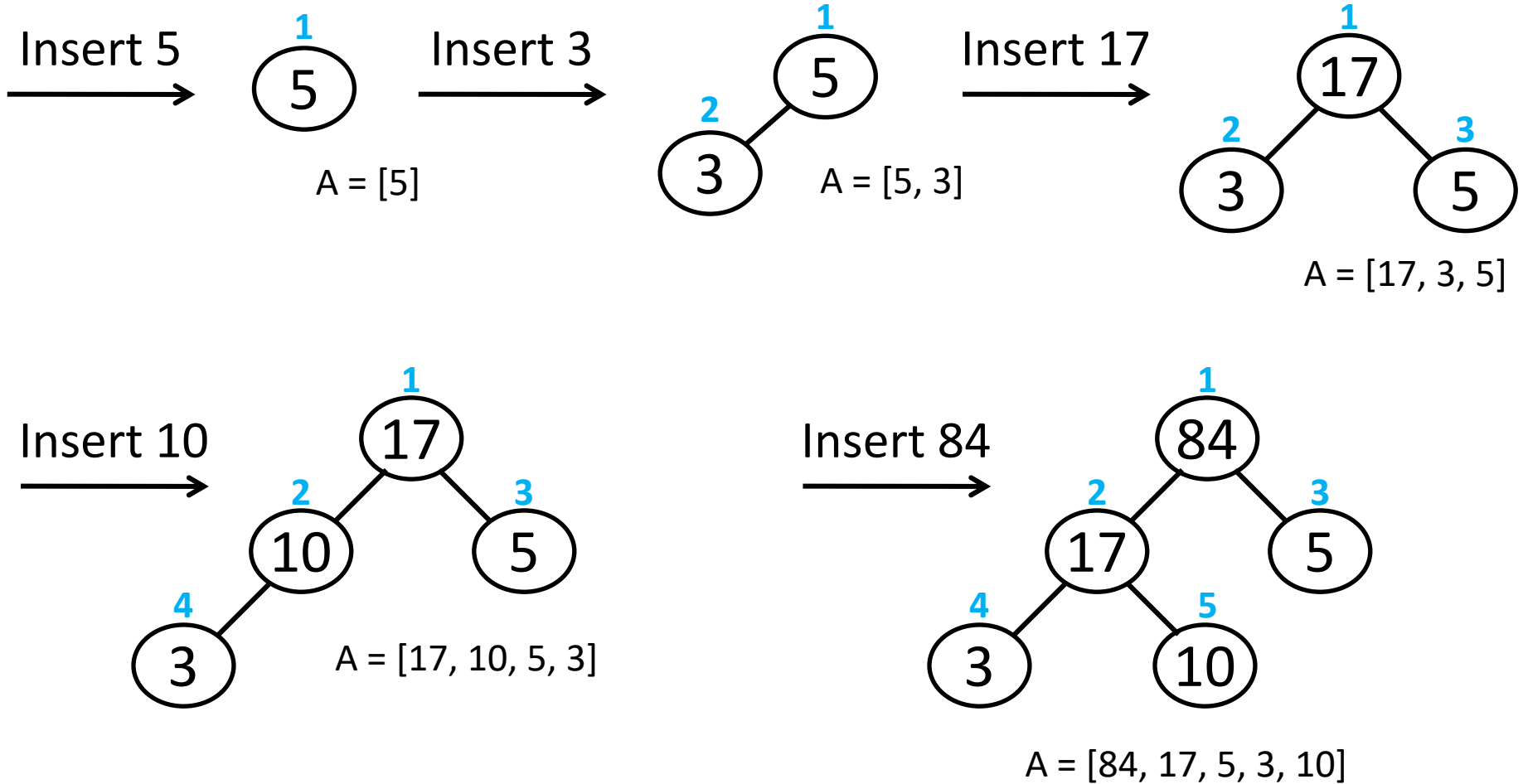


The restored heap containing the newly added element

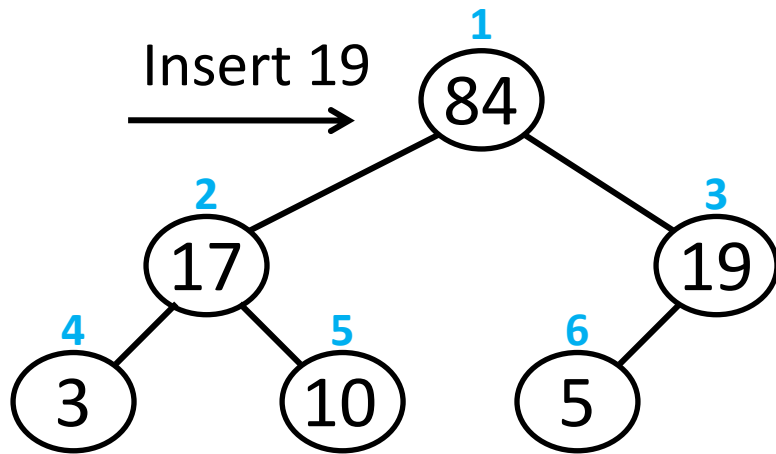


Create Max-heap using Max-Heap-Insert

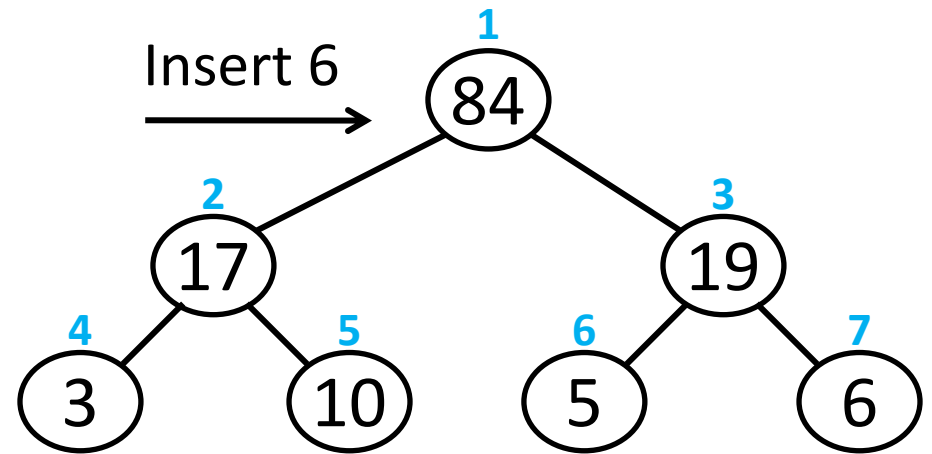
$A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$



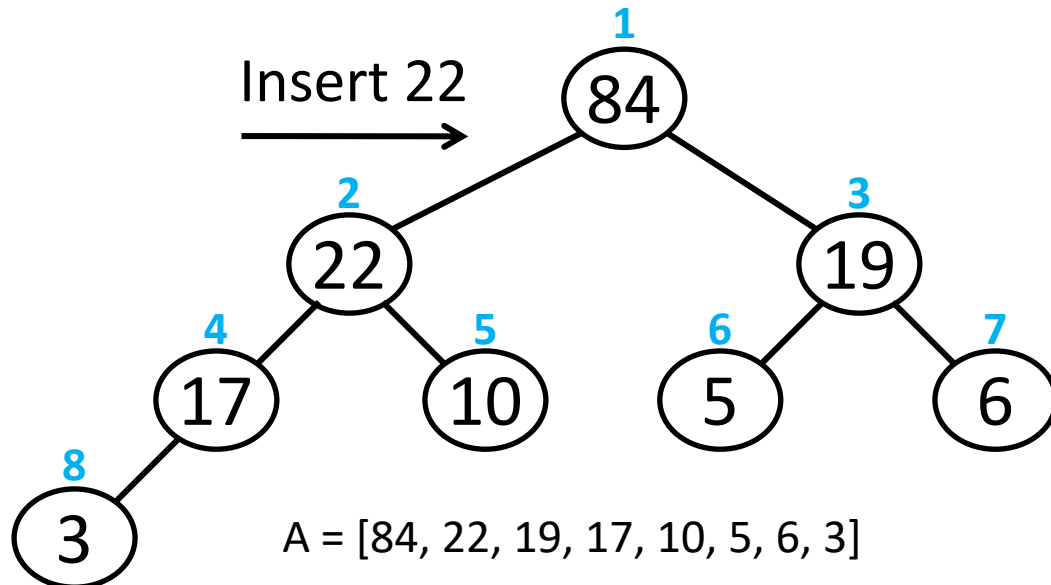
$A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$ Contd...



$A = [84, 17, 19, 3, 10, 5]$

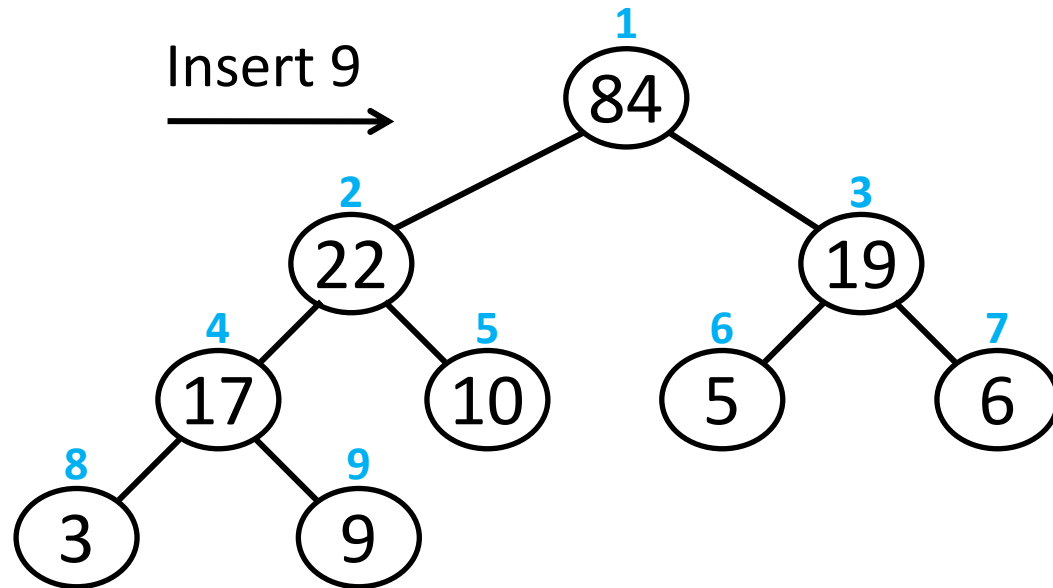


$A = [84, 17, 19, 3, 10, 5, 6]$



$A = [84, 22, 19, 17, 10, 5, 6, 3]$

$A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$ Contd...

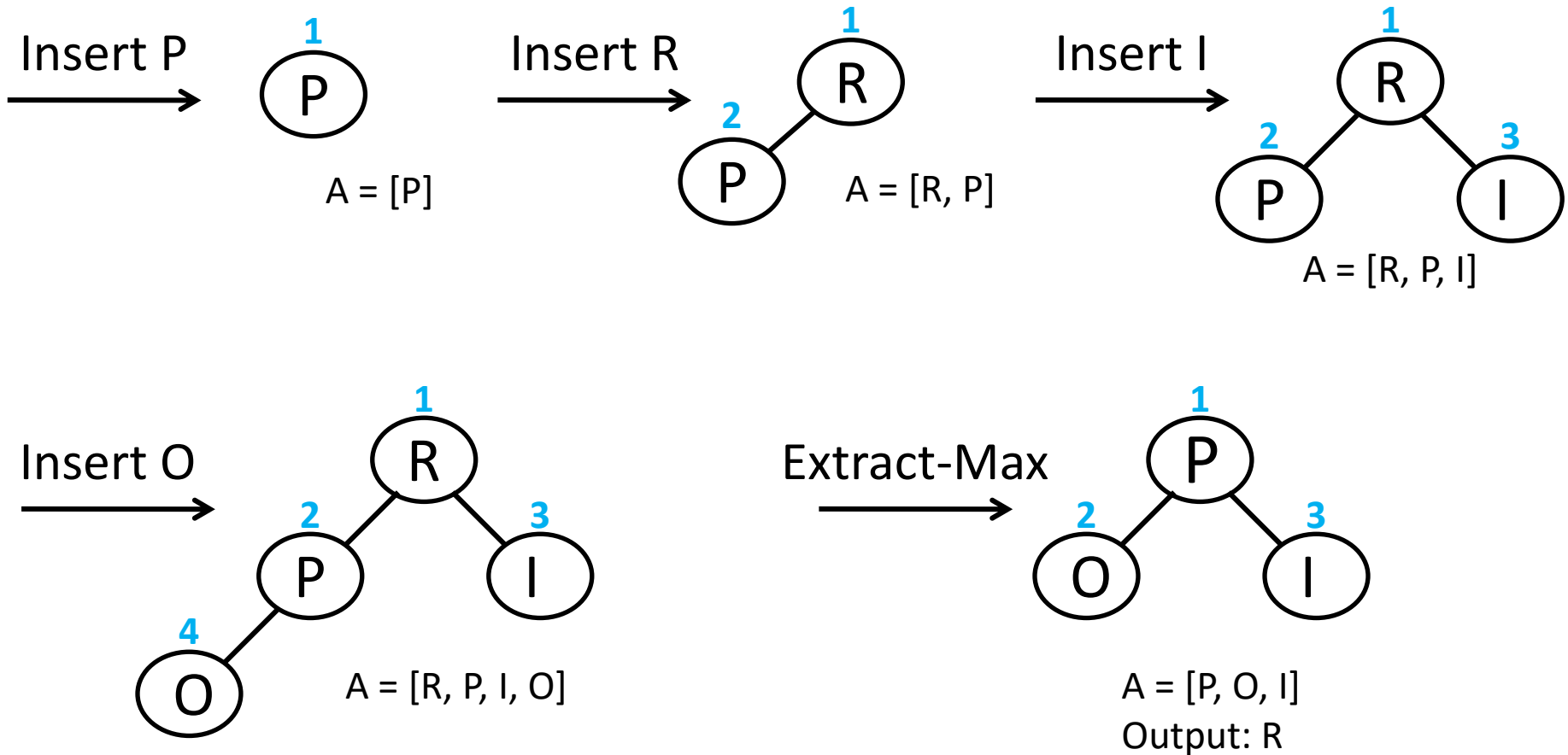


$A = [84, 22, 19, 17, 10, 5, 6, 3, 9]$

P R I O * R * * I * T * Y * * * Q U E * * * U * E

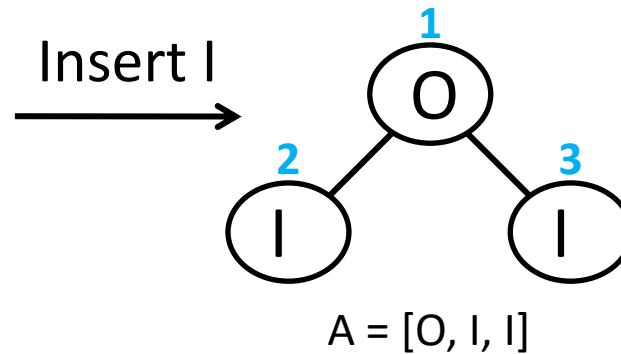
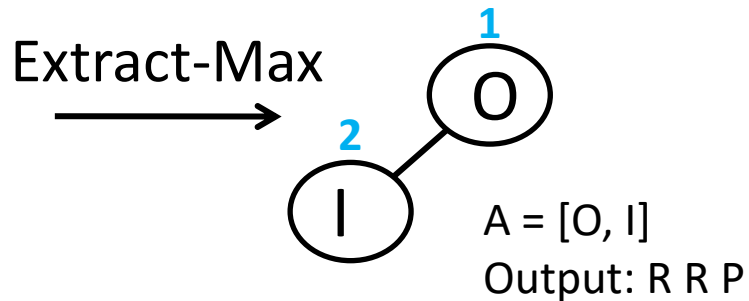
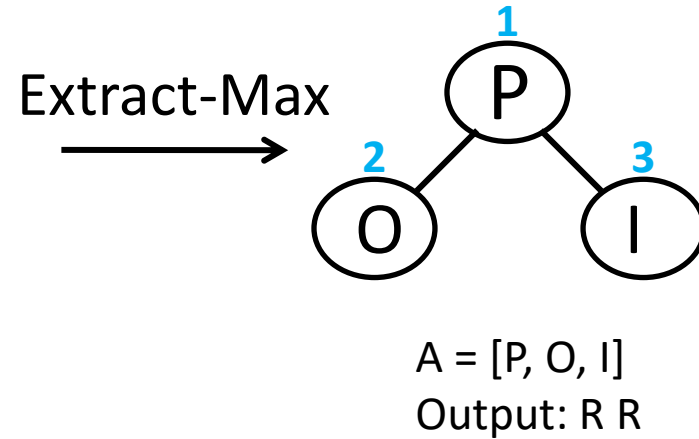
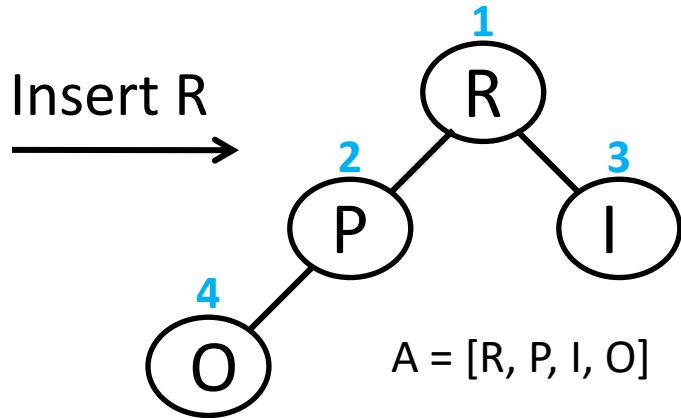
Character \rightarrow Max-Heap-Insert.

* \rightarrow Extract-Max.



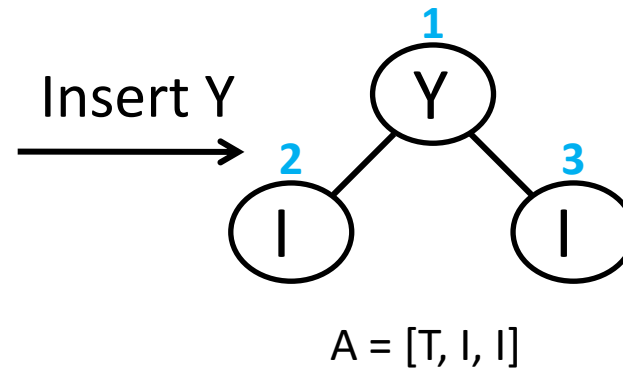
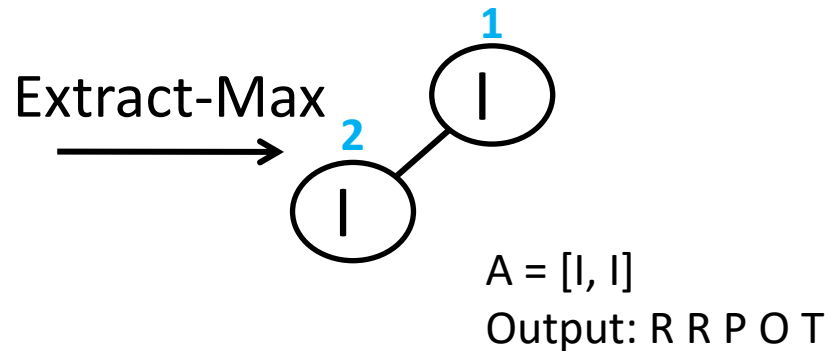
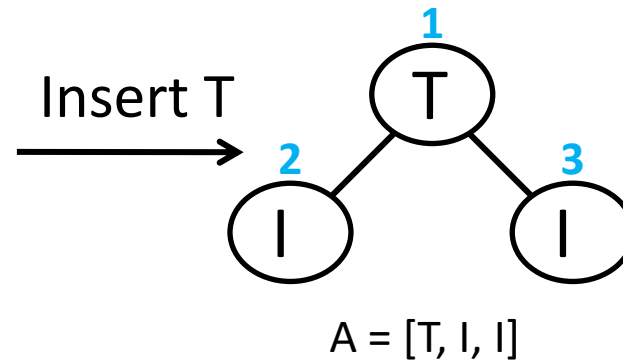
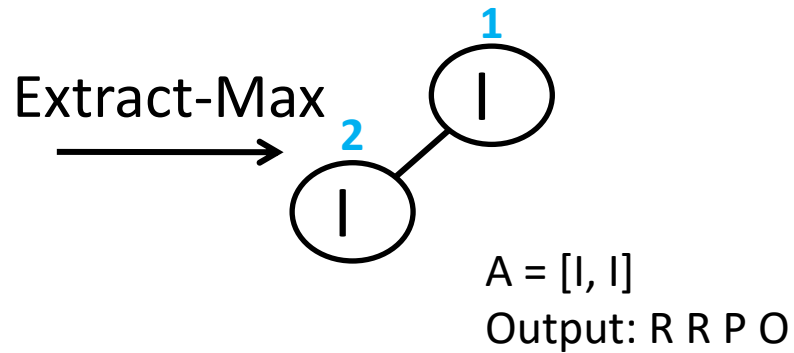
P R I O * R * * I * T * Y * * * Q U E * * * U * E

Contd...



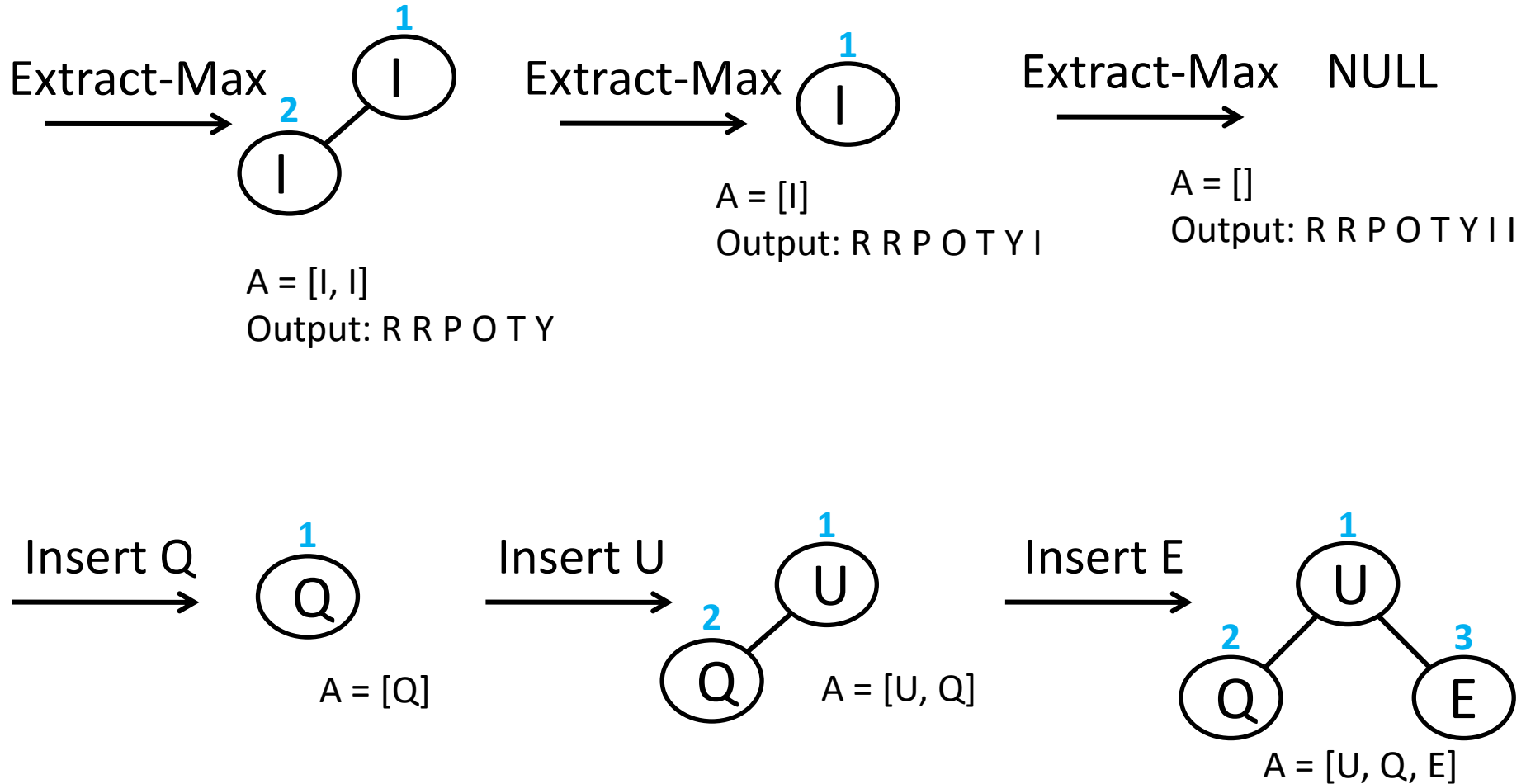
P R I O * R * * I * T * Y * * * Q U E * * * U * E

Contd...



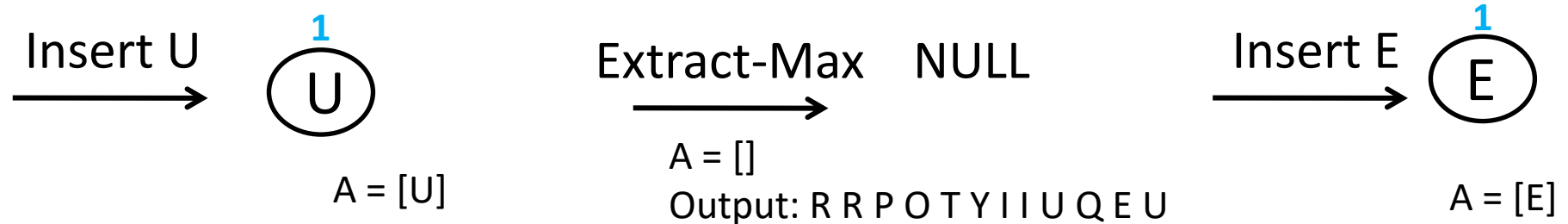
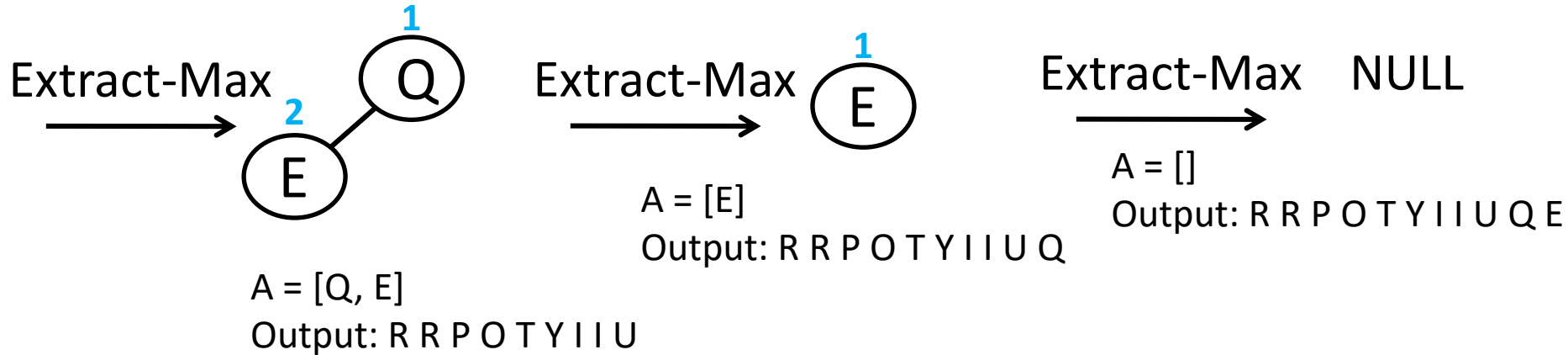
P R I O * R * * I * T * Y * * * Q U E * * * U * E

Contd...



P R I O * R * * I * T * Y * * * Q U E * * * U * E

Contd...



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.