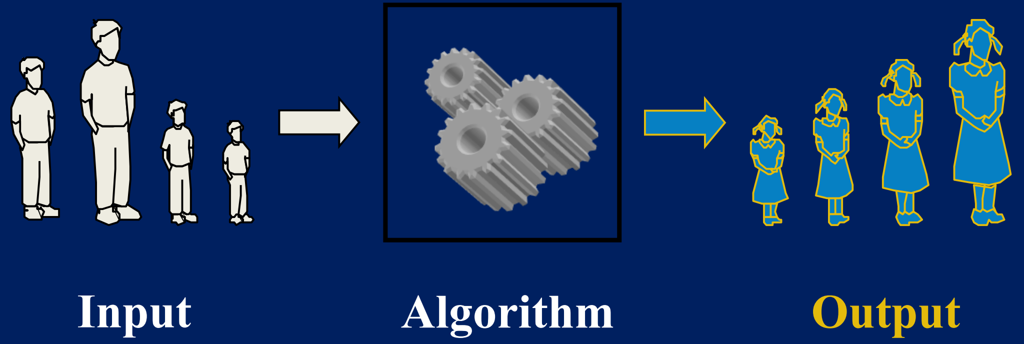
DESIGN & ANALYSIS OF ALGORITHM (BCSC0012)

Chapter 5: Divide and Conquer

Heap Sort



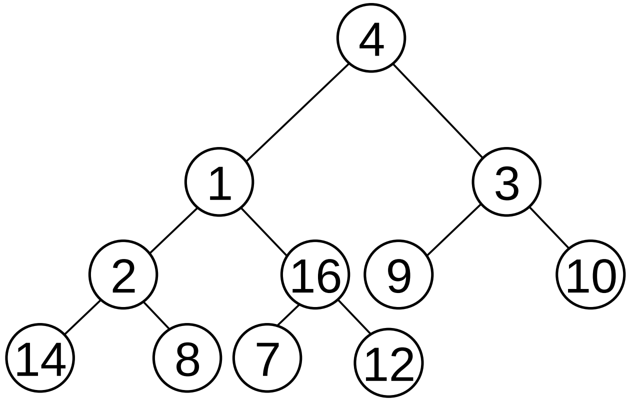
Prof. Anand Singh Jalal

Department of Computer Engineering & Applications

Trees: Overview

- Full Binary Tree:

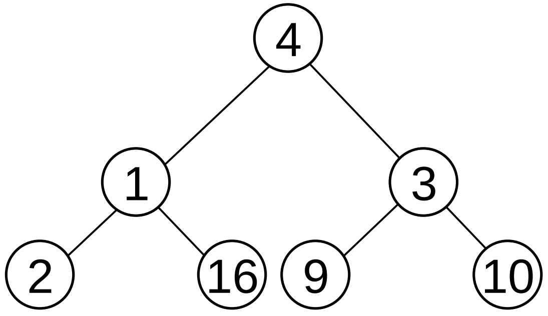
  A binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

- Complete Binary Tree:

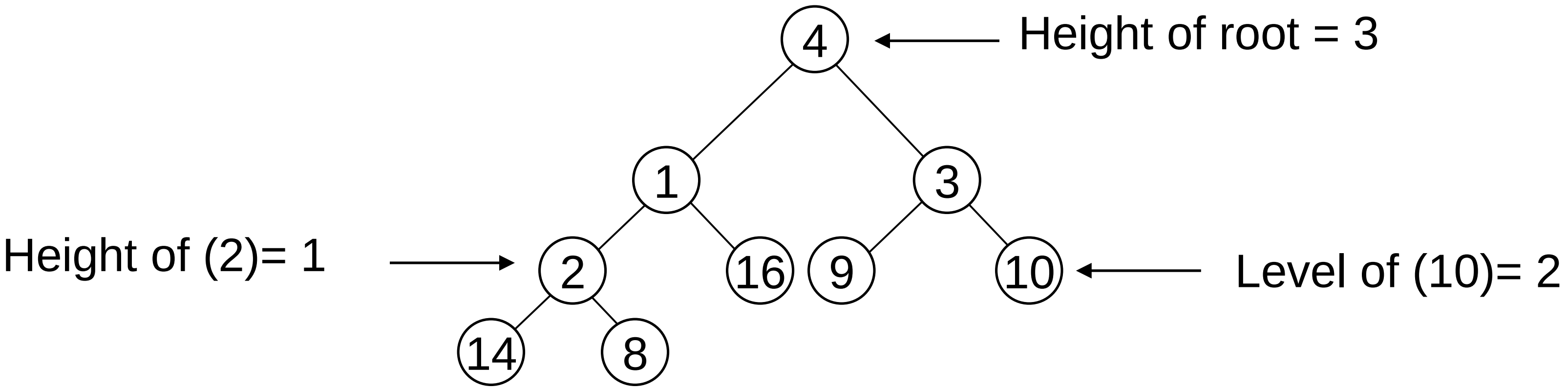
  A binary tree in which all leaves are on the same level and all internal nodes have degree 2.



Complete binary tree

Trees: Overview ...

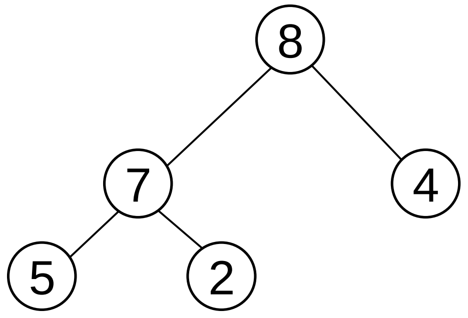
- **Height of a node** = the number of edges on the longest simple path from the node down to a leaf
- **Level of a node** = the length of a path from the root to the node
- **Height of tree** = height of root node



The Heap Data Structure

- *Def:* A **heap** is a *nearly complete* binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node X,

    Parent(X) ≥ X (in case of MaxHeap)



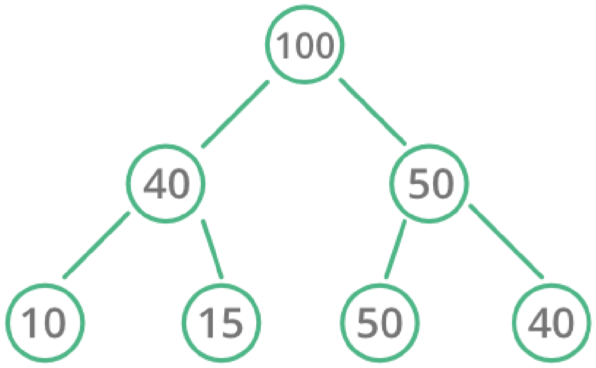
Heap

From the max heap property, it follows that:
"The root is the maximum element of the heap!"

Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

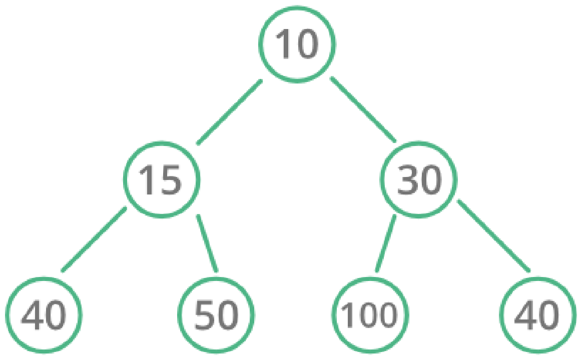
$$A[\mathrm{PARENT}(i)] \geq A[i]$$



Max Heap

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

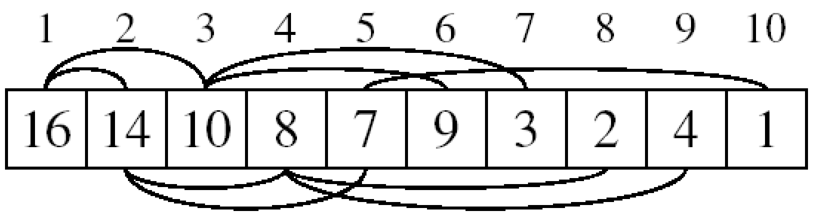
$$A[\mathrm{PARENT}(i)] \leq A[i]$$

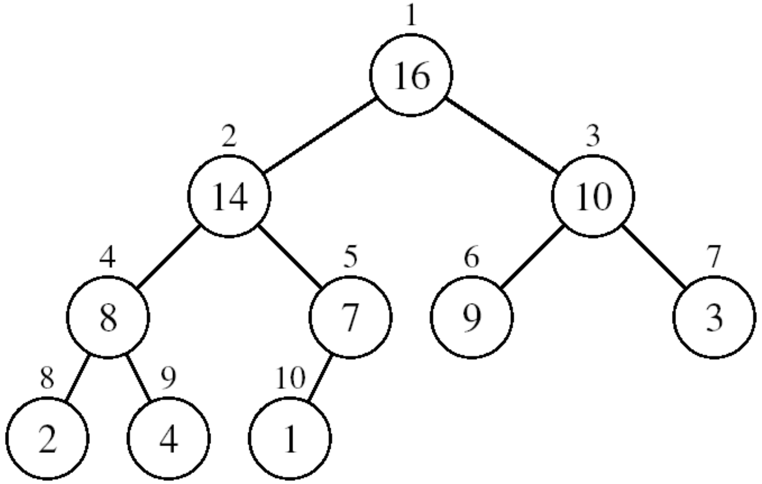


Min Heap

Array Representation of Heaps

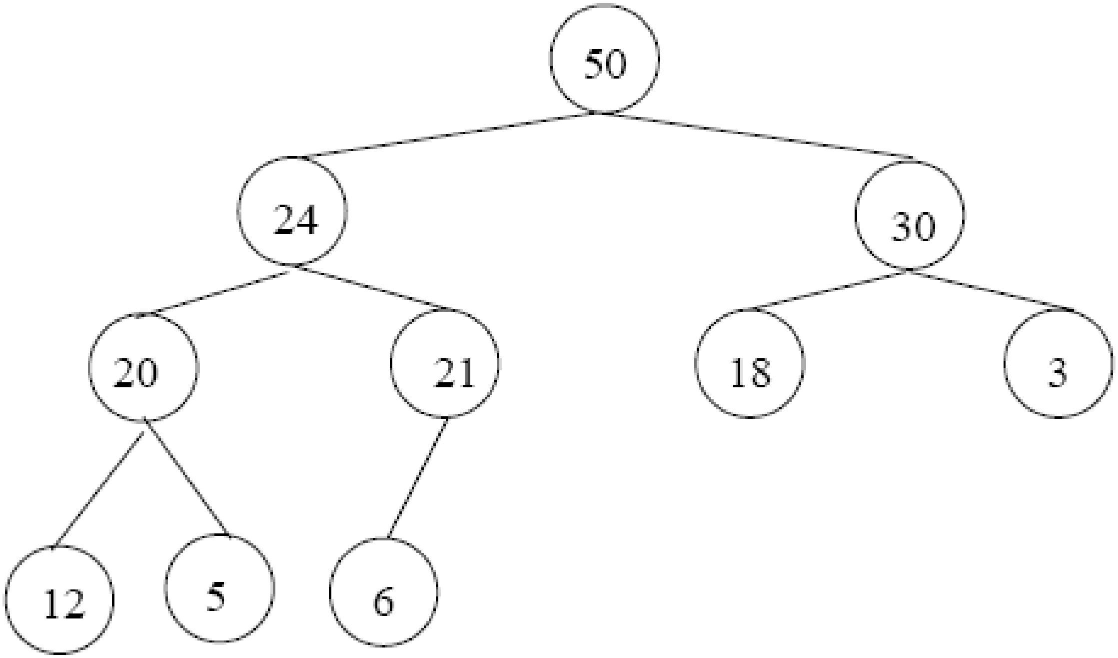
- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves





Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

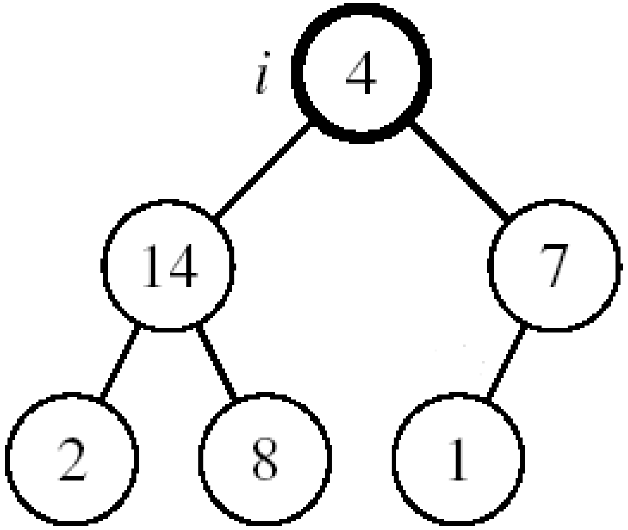


Operations on Heaps

- 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

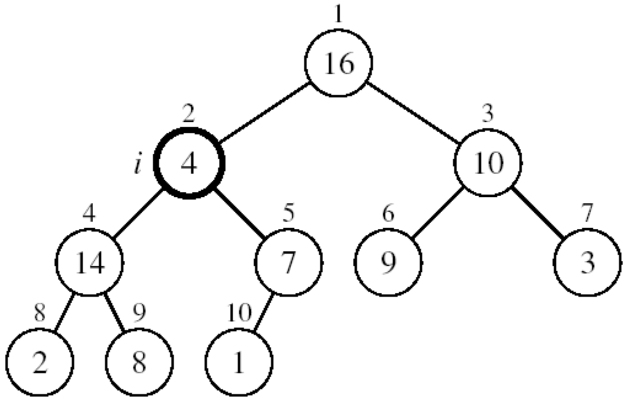
Maintaining the Heap Property

- **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



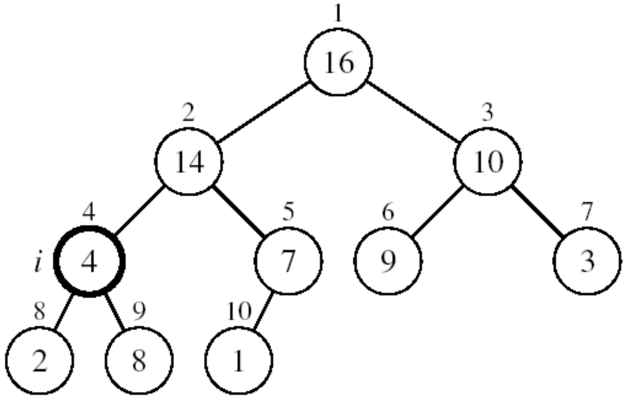
Example

<u>**MAX-HEAPIFY(A, 2, 10)**</u>



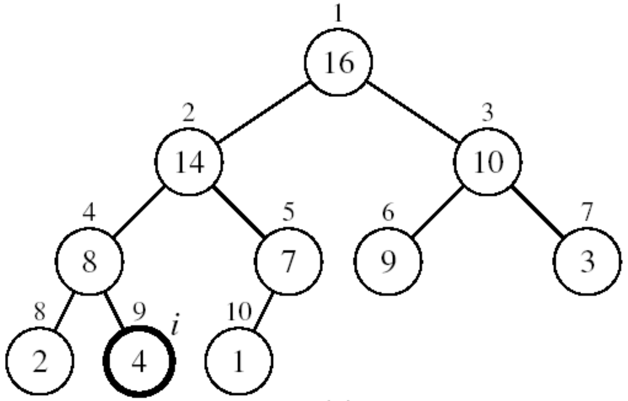
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

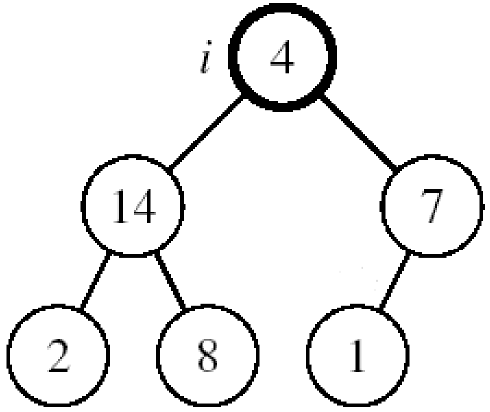
A[4] ↔ A[9]



Heap property restored

Maintaining the Heap Property

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



Alg: MAX-HEAPIFY(A, i, n)

```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
    then largest ←l
Else
    largest ←i;
if r ≤ n and A[r] > A[largest]
    then largest ←r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
        MAX-HEAPIFY(A, largest, n)
```

MAX-HEAPIFY Running Time

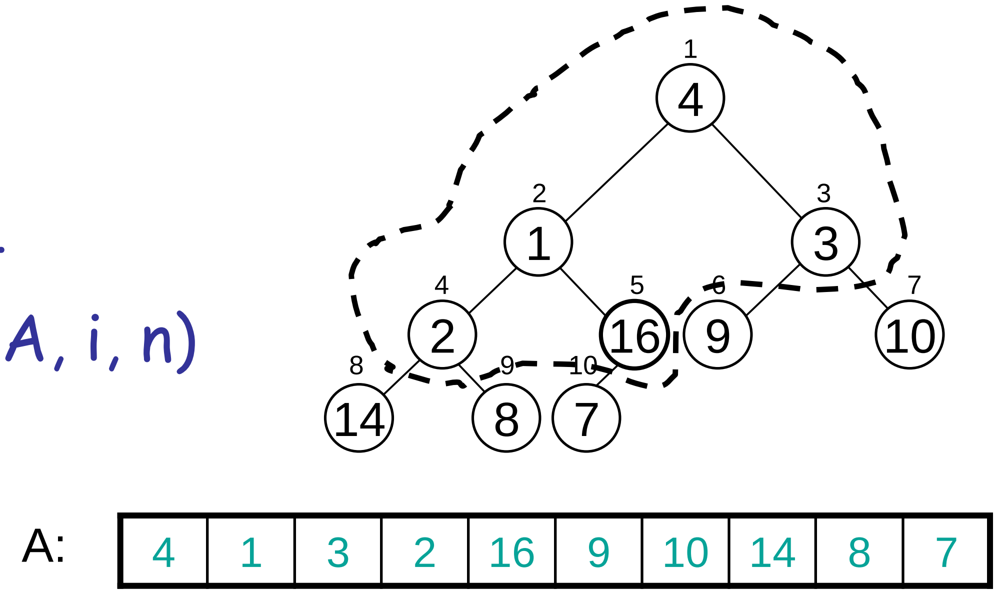
- Intuitively:
  - It traces a path from the root to a leaf (longest path length: h)
  - At each level, it makes exactly 2 comparisons
  - Total number of comparisons is 2h
  - Running time is O(h) or $O(lgn)$
- Running time of MAX-HEAPIFY is $O(lgn)$
- Can be written in terms of the height of the heap, as being $O(h)$
  – Since the height of the heap is $\lfloor lgn \rfloor$

Building a Heap

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

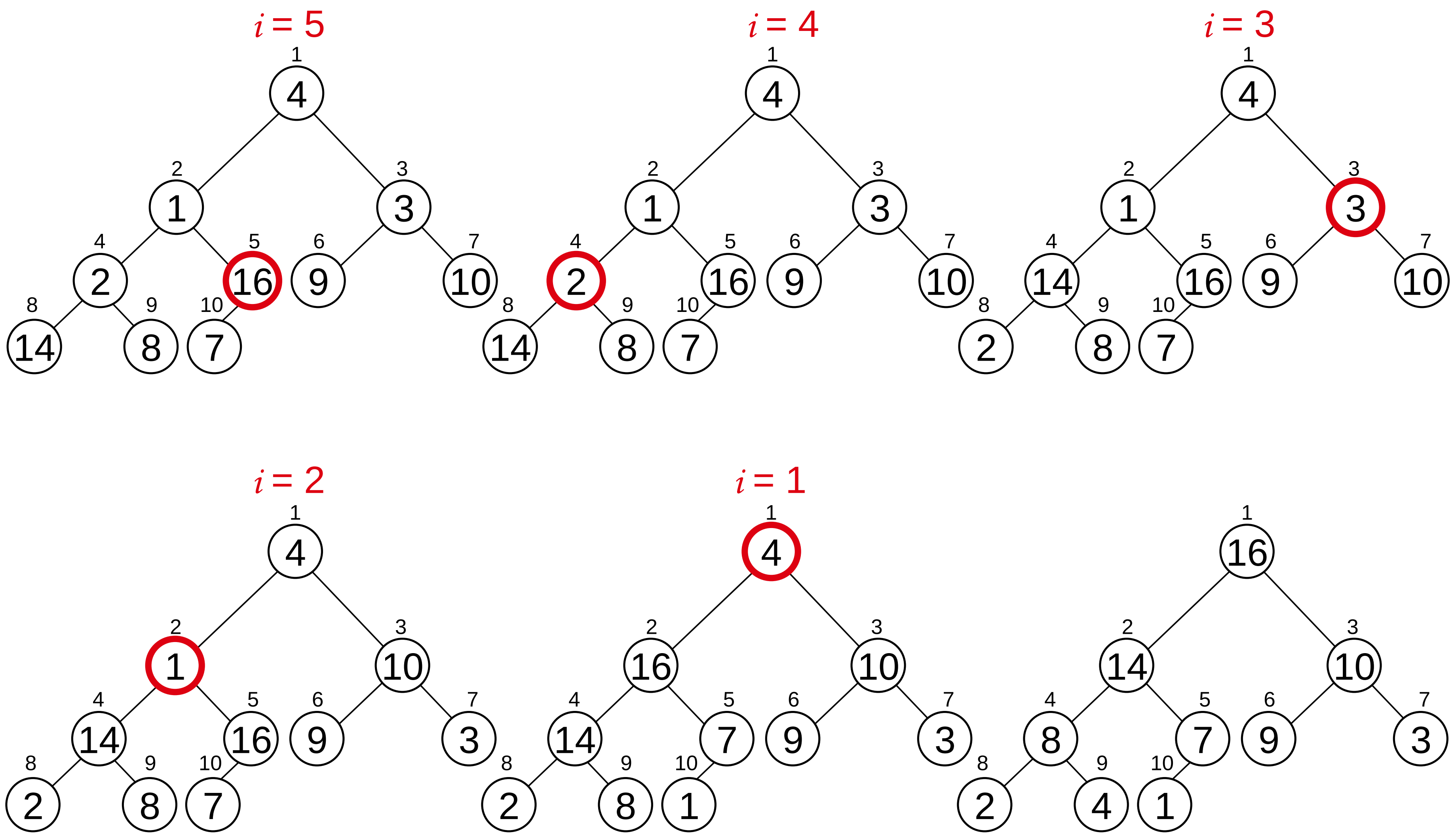
Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)



Example:

A	4	1	3	2	16	9	10	14	8	7



Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)

$O(\lg n)$ $\Big\}$ $O(n)$

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

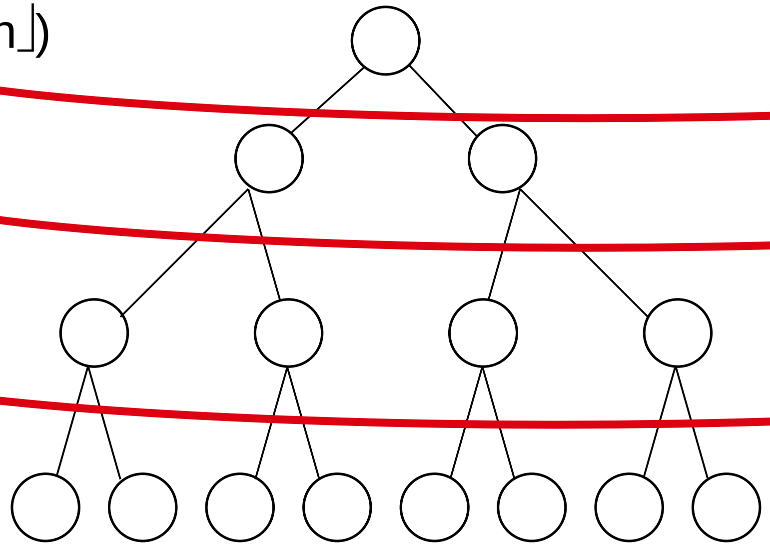
- This is not an asymptotically tight upper bound

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

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

Height	Level	No. of nodes
$h_0 = 3$ ($\lfloor \lg n \rfloor$)	$i = 0$	2^0
$h_1 = 2$	$i = 1$	2^1
$h_2 = 1$	$i = 2$	2^2
$h_3 = 0$	$i = 3$ ($\lfloor \lg n \rfloor$)	2^3



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

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

Running Time of BUILD MAX HEAP

$$T(n) = \sum_{i=0}^{h} n_i h_i$$ Cost of HEAPIFY at level i ∗ number of nodes at that level

$$= \sum_{i=0}^{h} 2^i (h-i)$$ Replace the values of n_i and h_i computed before

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$ Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$ Change variables: k = h - i

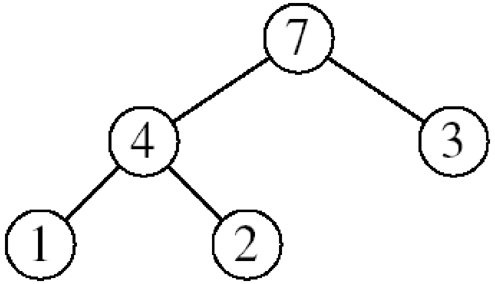
$$\le n \sum_{k=0}^{\infty} \frac{k}{2^k}$$ The sum above is smaller than the sum of all elements to ∞ and h = lgn

$$= O(n)$$ The sum above is smaller than 2

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

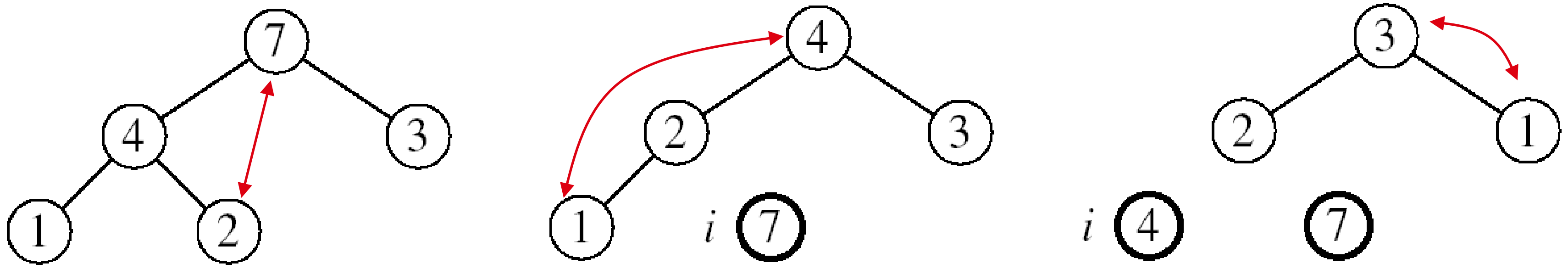
Heapsort

- Goal:
  - Sort an array using heap representations
- 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

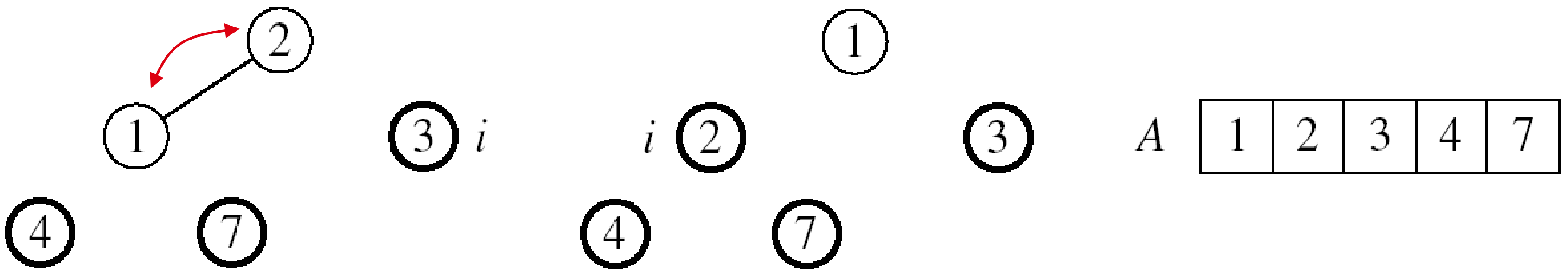


Heapsort ...

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



MAX-HEAPIFY(A, 1, 4) MAX-HEAPIFY(A, 1, 3) MAX-HEAPIFY(A, 1, 2)



MAX-HEAPIFY(A, 1, 1)

Heapsort …

1. BUILD-MAX-HEAP(A) $O(n)$
2. **for** i ← *length*[A] **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. MAX-HEAPIFY(A, 1, i - 1) $O(\lg n)$

n-1 times

- Running time: $O(n\lg n)$ --- Can be shown to be $\Theta(n\lg n)$

Heapsort : HEAP-EXTRACT-MAX

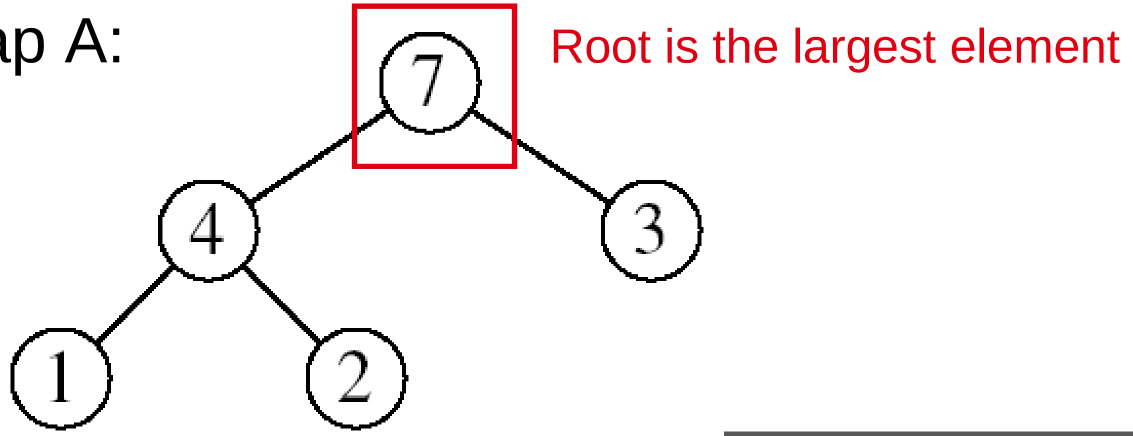
Goal:

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

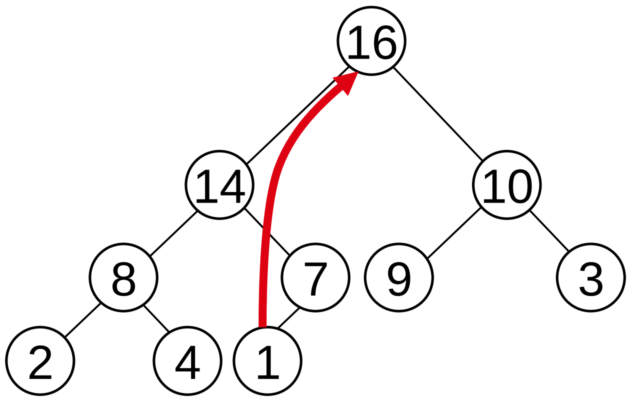
Idea:

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

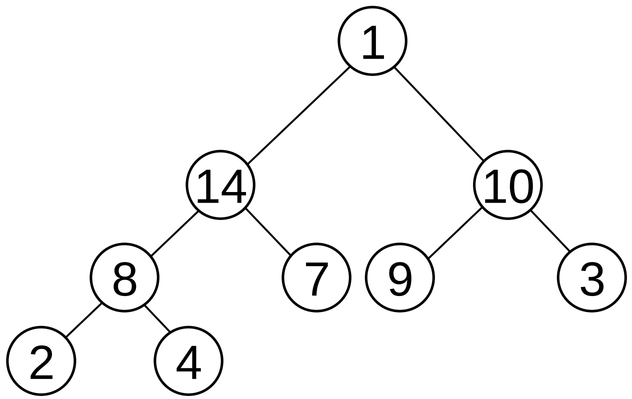
Heap A:



Heapsort : HEAP-EXTRACT-MAX …

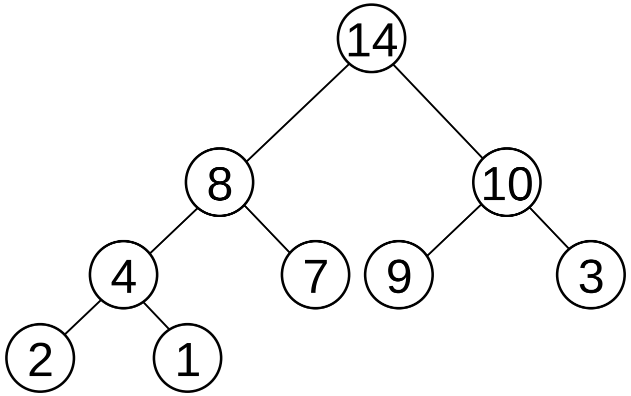


max = 16



Heap size decreased with 1

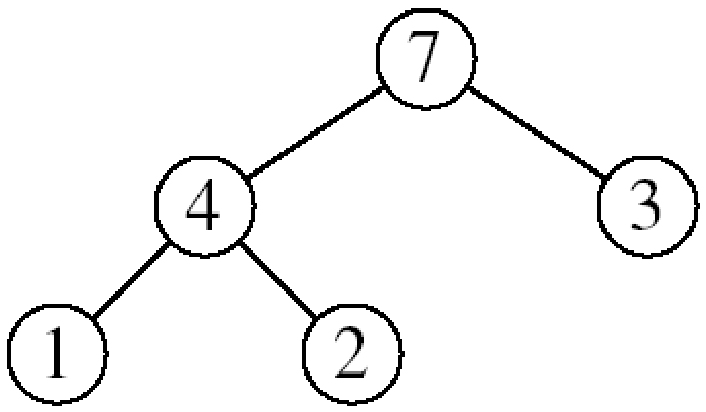
Call MAX-HEAPIFY(A, 1, n-1)



Heapsort : HEAP-EXTRACT-MAX …

Alg: HEAP-EXTRACT-MAX(A, n)

1. **if** $n < 1$
2. **then error** "heap underflow"
3. $max \leftarrow A[1]$
4. $A[1] \leftarrow A[n]$
5. MAX-HEAPIFY(A, 1, n-1) ▷ remakes heap
6. **return** max



Running time: $O(\lg n)$

Heapsort

- We can perform the following operations on heaps:
  - MAX-HEAPIFY $O(\lg n)$
  - BUILD-MAX-HEAP $O(n)$
  - HEAP-SORT $O(n\lg n)$
  - MAX-HEAP-INSERT $O(\lg n)$
  - HEAP-EXTRACT-MAX $O(\lg n)$
  - HEAP-INCREASE-KEY $O(\lg n)$
  - HEAP-MAXIMUM $O(1)$

Average $O(\lg n)$

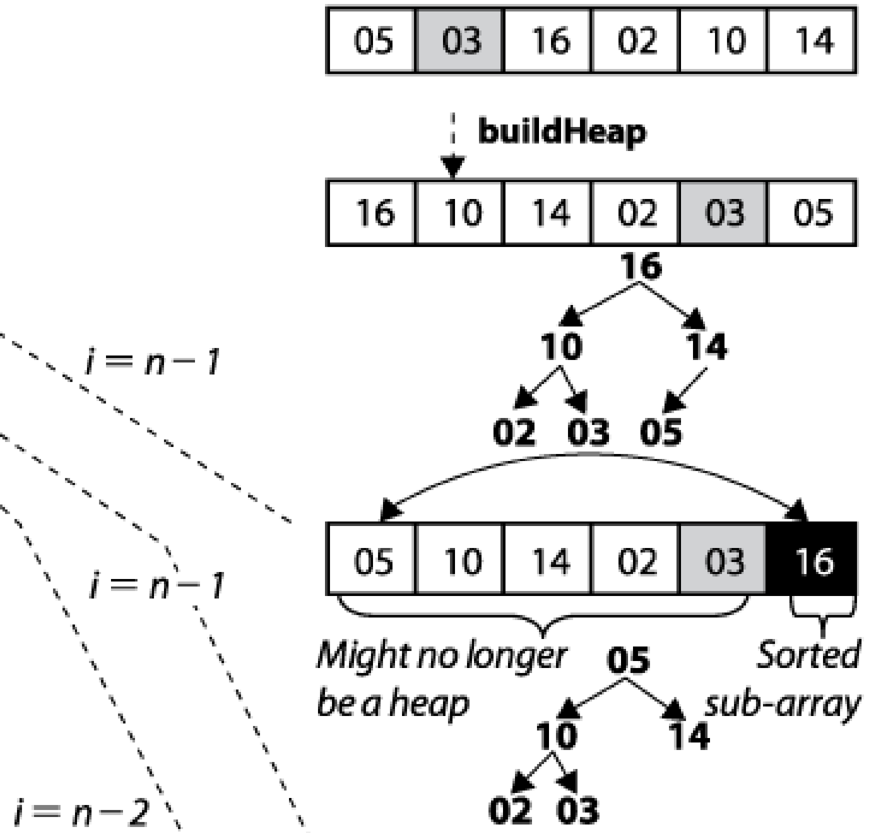
Heapsort

HEAP SORT

Best	Average	Worst
O(n log n)	O(n log n)	O(n log n)

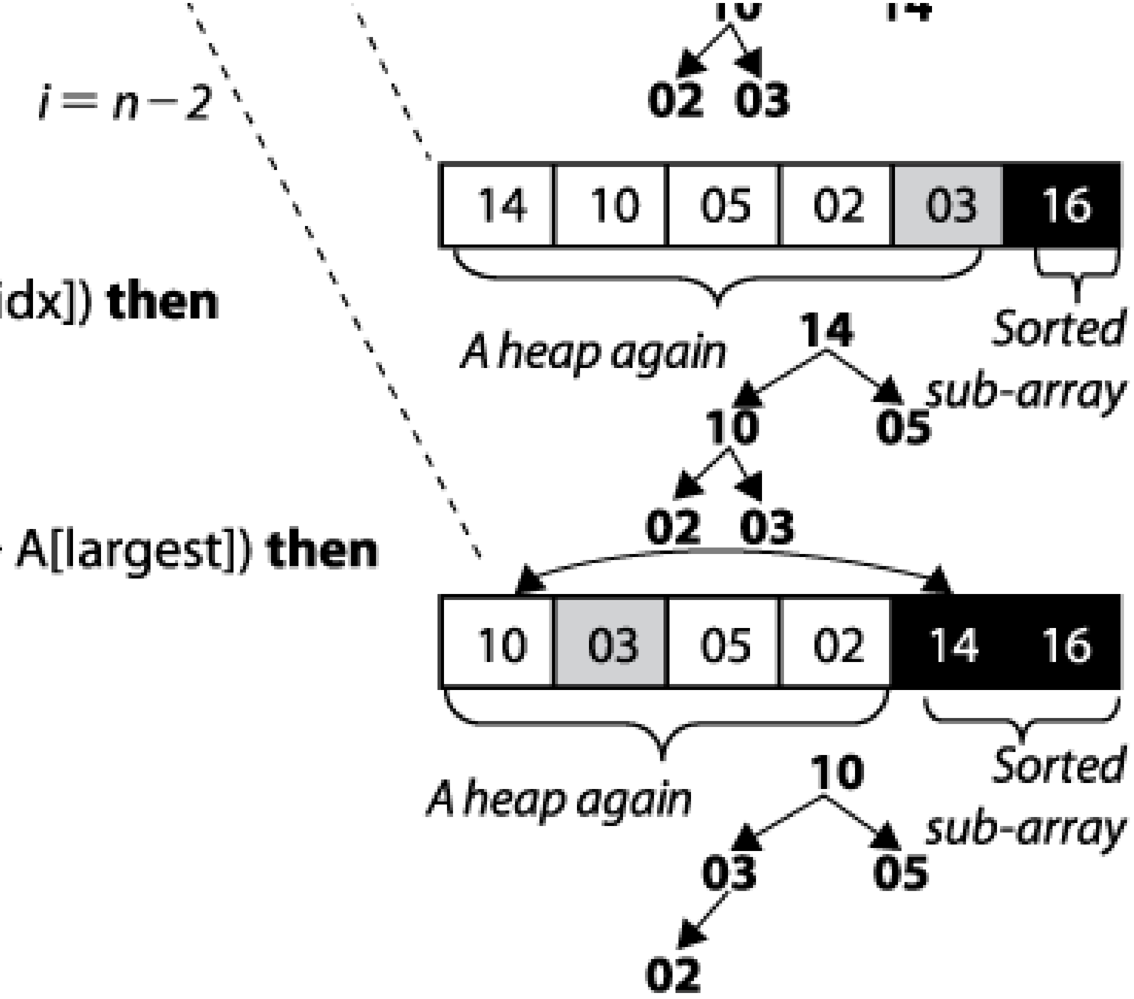
Array · Recursion · Binary Heap

```
sort (A)
1.    buildHeap(A)
2.    for i = n − 1  downto 1 do
3.       swap A[0] with A[i]
4.       heapify (A, 0, i)
end

buildHeap (A)
1.    for i = ⌊n/2⌋ − 1 downto 0 do
2.       heapify (A, i, n)
end

heapify (A, idx, max)
1.    left = 2*idx + 1
```



Heapsort ...

```
heapify (A, idx, max)
    left = 2*idx + 1
    right = 2*idx + 2
    if (left < max and A[left] > A[idx]) then
        largest = left
    else largest = idx
    if (right < max and A[right] > A[largest]) then
        largest = right
    if (largest ≠ idx) then
        swap A[i] and A[largest]
        heapify (A, largest, max)
end
```



Any Questions ?

Dr. Anand Singh Jalal
Professor
Email: asjalal@gla.ac.in