CS473 - Algorithms I

Lecture 8
Heapsort

View in slide-show mode

Cevdet Aykanat and Mustafa Ozdal
Computer Engineering Department, Bilkent University

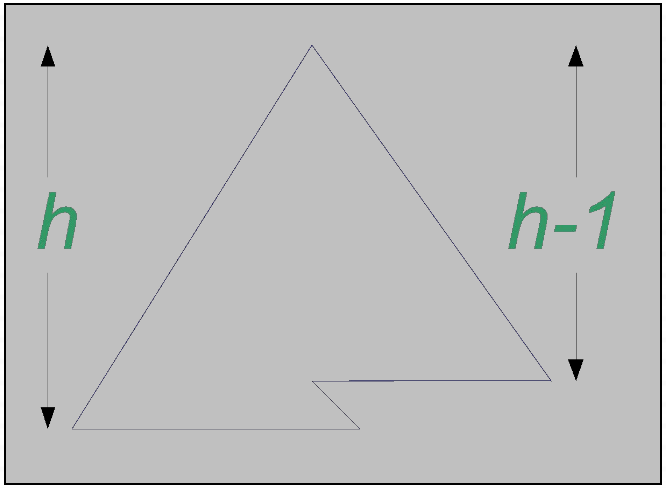
Heapsort

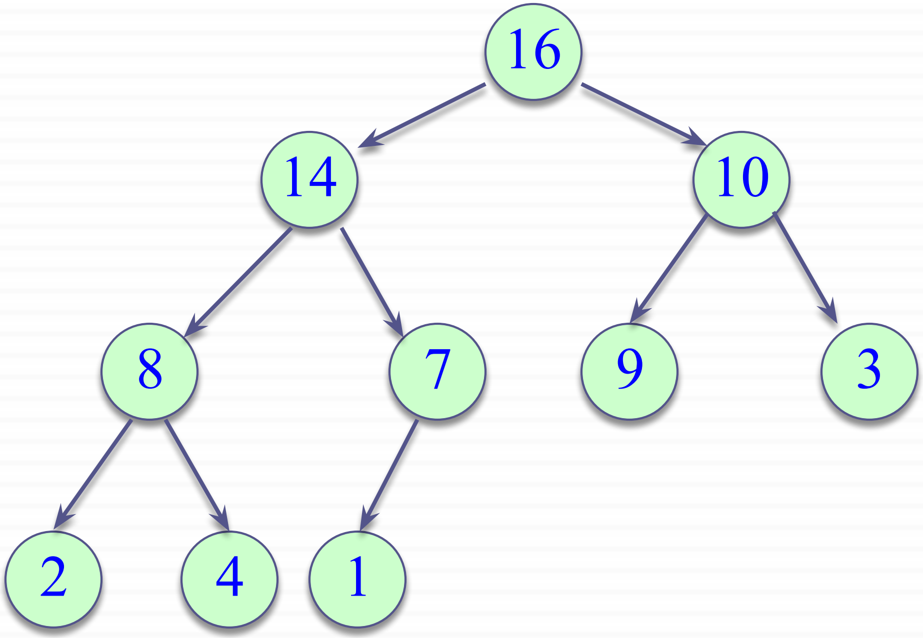
- Worst-case runtime: O(nlgn)

- Sorts in-place

- Uses a special data structure (heap) to manage information during execution of the algorithm
  - Another design paradigm

Heap Data Structure

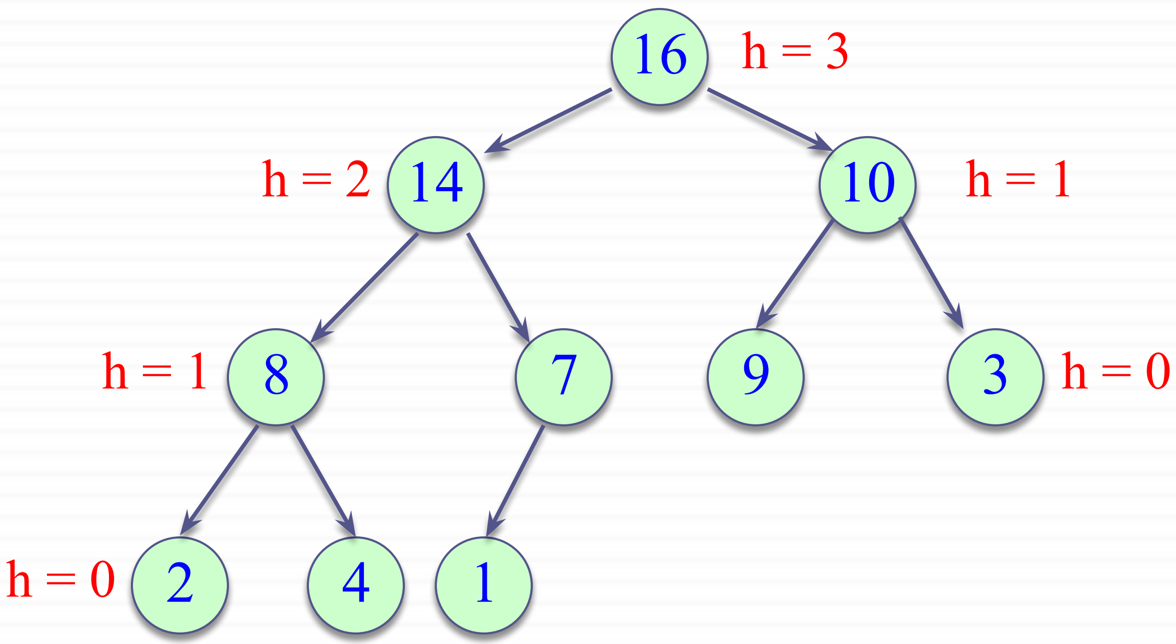




Nearly complete binary tree

- Completely filled on all levels except possibly the lowest level

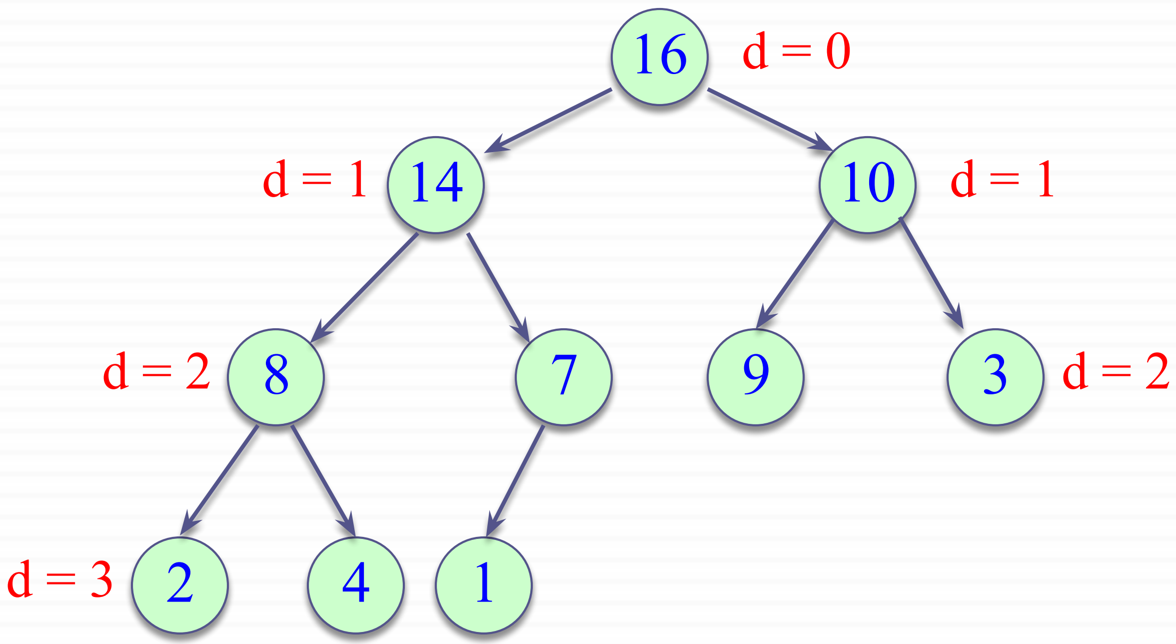
Heap Data Structures



Height of node i: Length of the longest simple downward path from i to a leaf

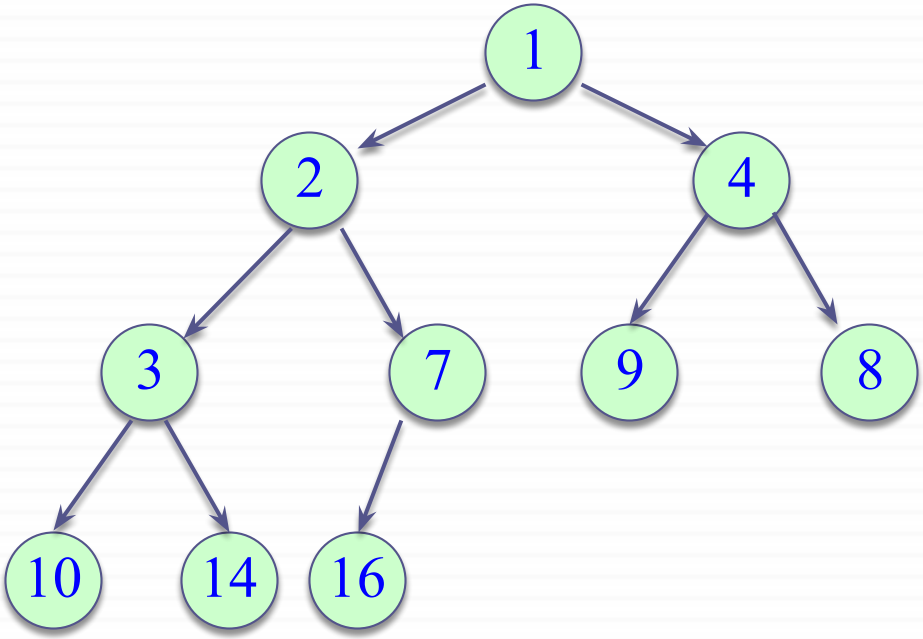
Height of the tree: height of the root

Heap Data Structures



Depth of node i: Length of the simple downward path from the root to node i

Heap Property: Min-Heap

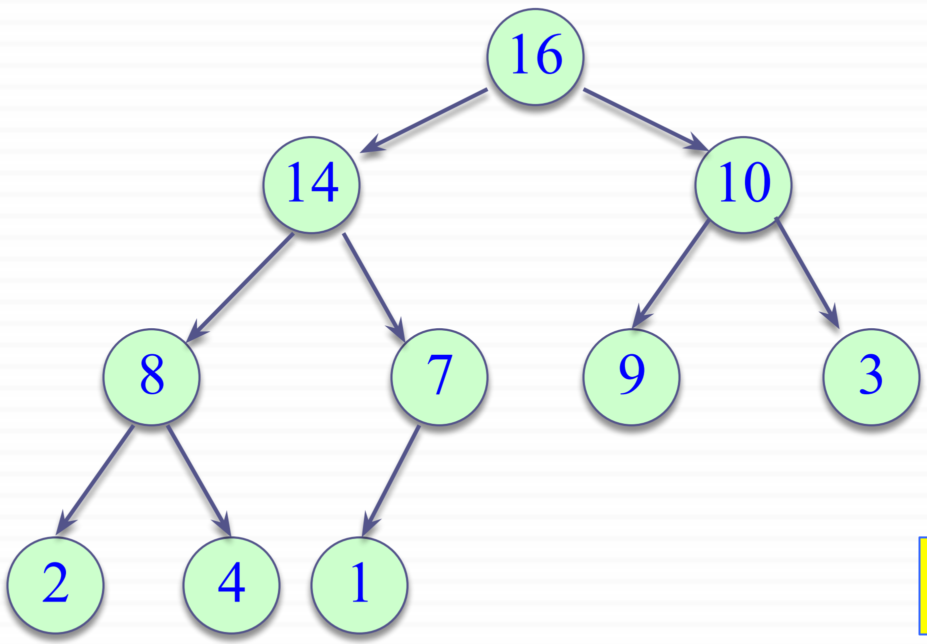


The smallest element in any subtree is the root element in a min-heap

Min heap: For every node i other than root, A[parent(i)] ≤ A[i]

- Parent node is always smaller than the child nodes

Heap Property: Max-Heap



The largest element in any subtree is the root element in a max-heap

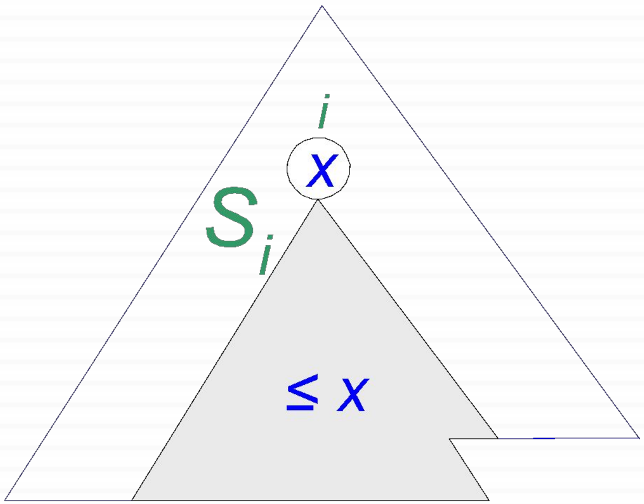
We will focus on max-heaps

Max heap: For every node i other than root, A[parent(i)] ≥ A[i]

- Parent node is always larger than the child nodes

Heap Property: Max-Heap

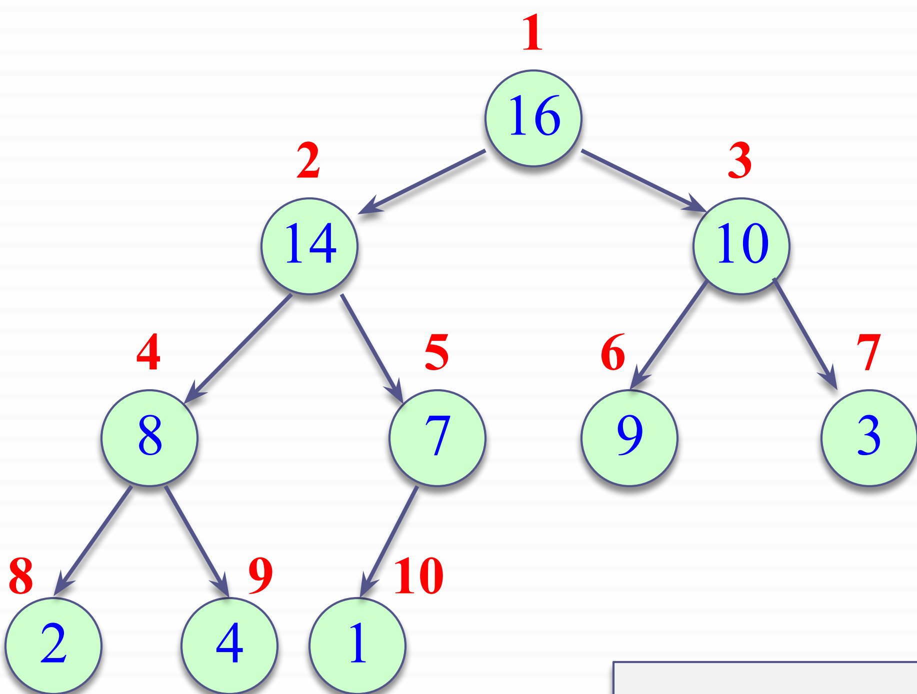
The largest element in any subtree is the root element in a max-heap



Max heap: For every node i other than root, A[parent(i)] ≥ A[i]

- Parent node is always larger than the child nodes

Heap Data Structure

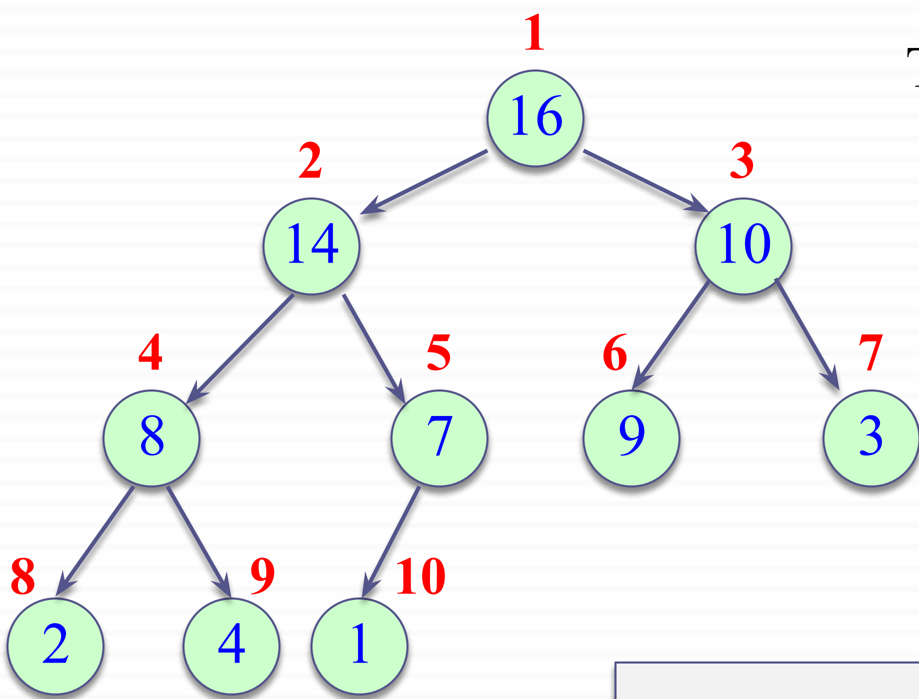


Heap can be stored in a linear array

Storage

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

Heap Data Structure



The links in the heap are implicit:

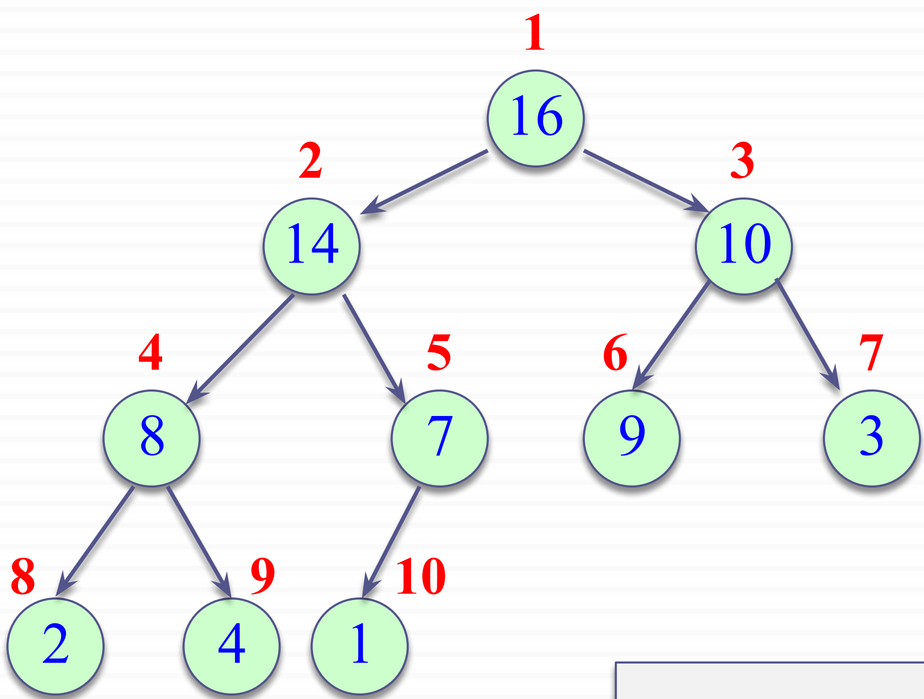
$$left(i) = 2i$$

$$right(i) = 2i + 1$$

$$parent(i) = \lfloor i/2 \rfloor$$

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

Heap Data Structure



$$left(i) = 2i$$

e.g. Left child of node 4 has index 8

$$right(i) = 2i + 1$$

e.g. Right child of node 2 has index 5

$$parent(i) = \lfloor i / 2 \rfloor$$

e.g. Parent of node 7 has index 3

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

Heap Data Structures

- Computing left child, right child, and parent indices very fast
  - left(i) = 2i binary left shift
  - right(i) = 2i+1 binary left shift, then set the lowest bit to 1
  - parent(i) = floor(i/2) right shift in binary

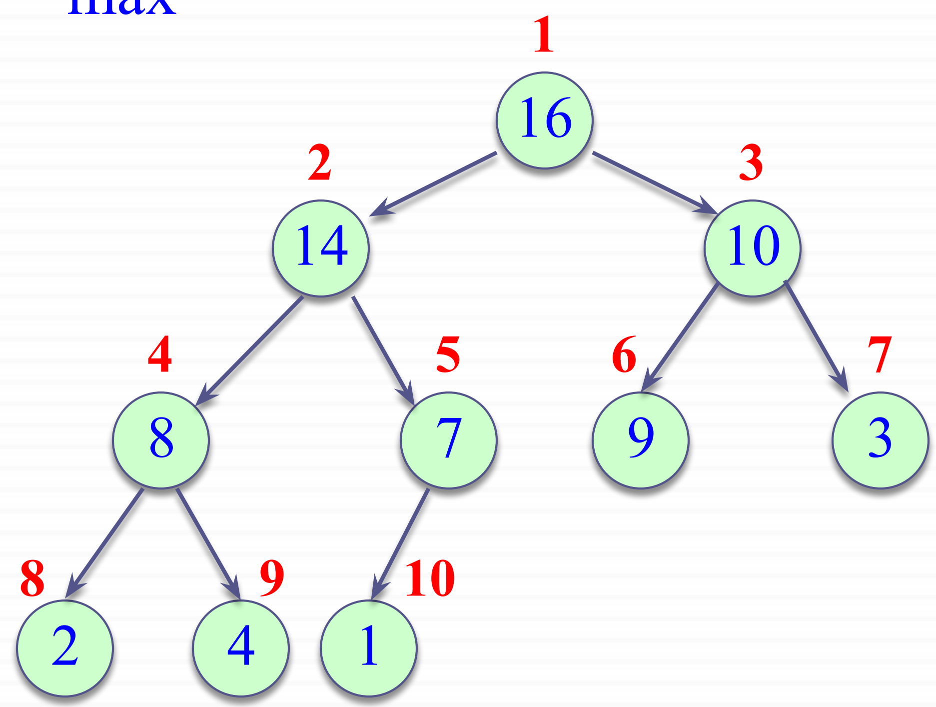
- A[1] is always the root element

- Array A has two attributes:
  - length(A): The number of elements in A
  - n = heap-size(A): The number elements in heap

$$n \leq \text{length}(A)$$

Heap Operations: Extract-Max

max=



EXTRACT-MAX(A, *n*)

max ← A[1]

A[1] ← A[n]

$n \leftarrow n - 1$

HEAPIFY(A, 1, n)

return max

Return the max element, and reorganize the heap to maintain heap property

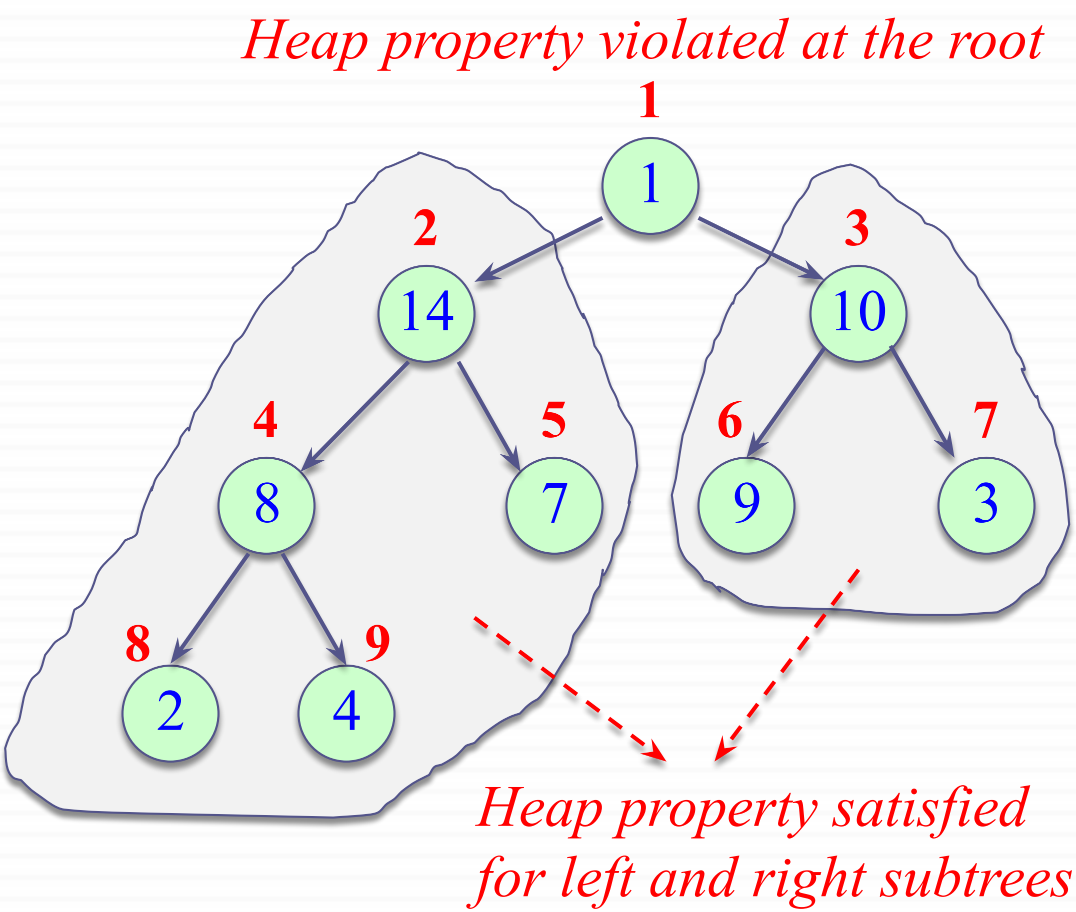
Heap Operations: HEAPIFY

Maintaining heap property:

Subtrees rooted at left[i] and right[i] are already heaps.

But, A[i] may violate the heap property (i.e., may be smaller than its children)

Idea: Float down the value at A[i] in the heap so that subtree rooted at i becomes a heap.



Heap Operations: HEAPIFY

HEAPIFY(A, i, n)

largest $\leftarrow i$

if $2i \leq n$ **and** $A[2i] > A[i]$
then largest $\leftarrow 2i$

if $2i + 1 \leq n$ **and** $A[2i+1] > A[\text{largest}]$
then largest $\leftarrow 2i + 1$

if largest $\neq i$ **then**
exchange $A[i] \leftrightarrow A[\text{largest}]$
HEAPIFY(A, largest, n)

initialize *largest* to be the *node i*

check the *left child* of *node i*

check the *right child* of *node i*

compute the largest of:

1) node i
2) left child of node i
3) right child of node i

exchange the *largest* of the 3 with *node i*

recursive call on the subtree

Heap Operations: HEAPIFY

HEAPIFY(A, i, n)

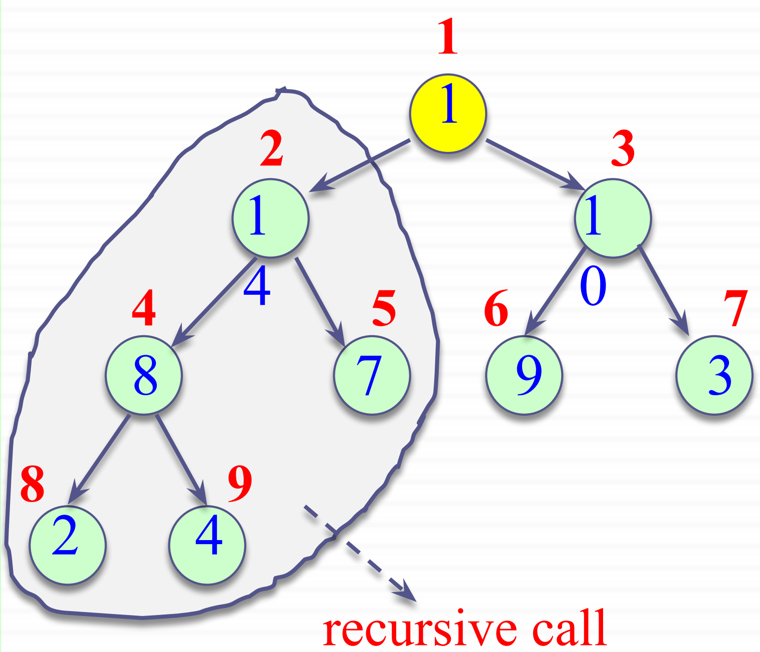
largest $\leftarrow i$

if $2i \le n$ **and** $A[2i] > A[i]$
then largest $\leftarrow 2i$

if $2i + 1 \le n$ **and** $A[2i+1] > A[\text{largest}]$
then largest $\leftarrow 2i + 1$

if largest $\ne i$ **then**
exchange $A[i] \leftrightarrow A[\text{largest}]$
HEAPIFY(A, largest, n)

HEAPIFY(A, 1, 9)



Heap Operations: HEAPIFY

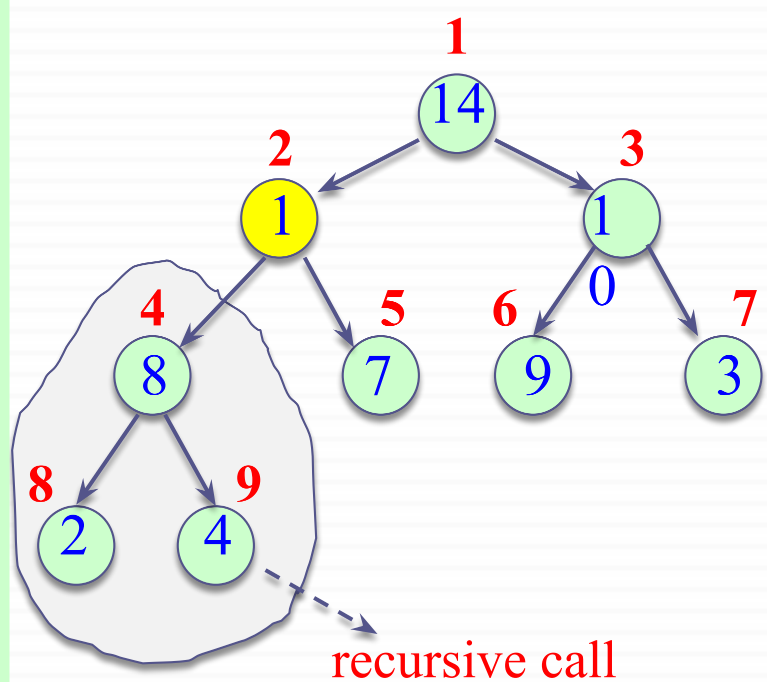
HEAPIFY(A, i, n)

largest ← i

if $2i \le n$ **and** $A[2i] > A[i]$
then largest ← $2i$

if $2i + 1 \le n$ **and** $A[2i+1] > A[\text{largest}]$
then largest ← $2i + 1$

if largest $\ne i$ **then**
exchange $A[i] \leftrightarrow A[\text{largest}]$
HEAPIFY(A, largest, n)



Heap Operations: HEAPIFY

HEAPIFY(A, i, n)

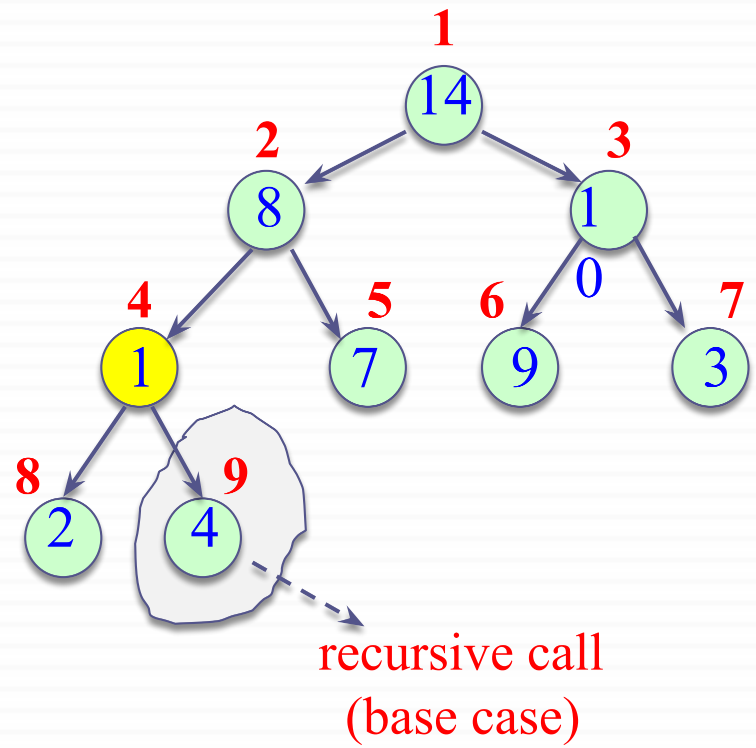
largest ← i

if $2i \le n$ **and** $A[2i] > A[i]$
then largest ← $2i$

if $2i + 1 \le n$ **and** $A[2i+1] > A[\text{largest}]$
then largest ← $2i + 1$

if largest $\neq i$ **then**
exchange $A[i] \leftrightarrow A[\text{largest}]$
HEAPIFY(A, largest, n)

recursive call:
HEAPIFY(A, 4, 9)



HEAPIFY: Summary (Floating Down the Value)

HEAPIFY(A, i, n)

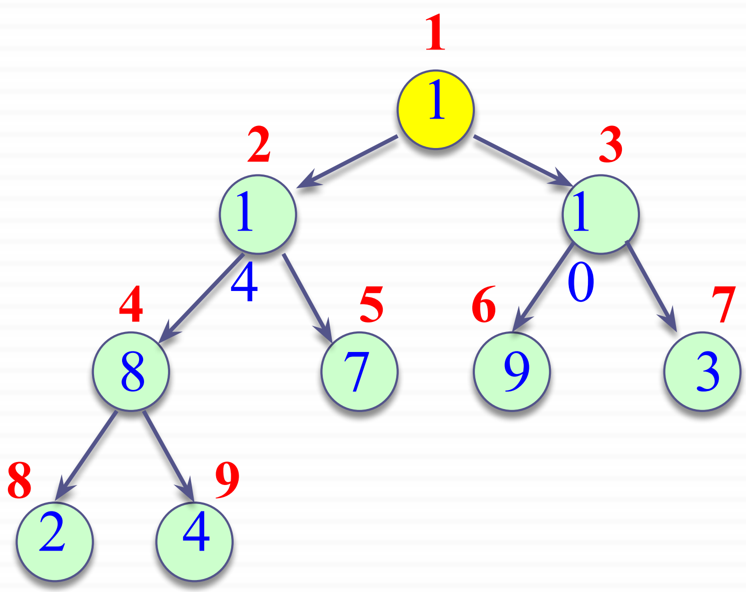
largest $\leftarrow i$

if $2i \le n$ **and** $A[2i] > A[i]$
then largest $\leftarrow 2i$

if $2i+1 \le n$ **and** $A[2i+1] > A[\text{largest}]$
then largest $\leftarrow 2i+1$

if largest $\ne i$ **then**
  exchange $A[i] \leftrightarrow A[\text{largest}]$
  HEAPIFY(A, largest, n)

HEAPIFY(A, 1, 9)



Heap Operations: HEAPIFY

HEAPIFY(A, i, n)

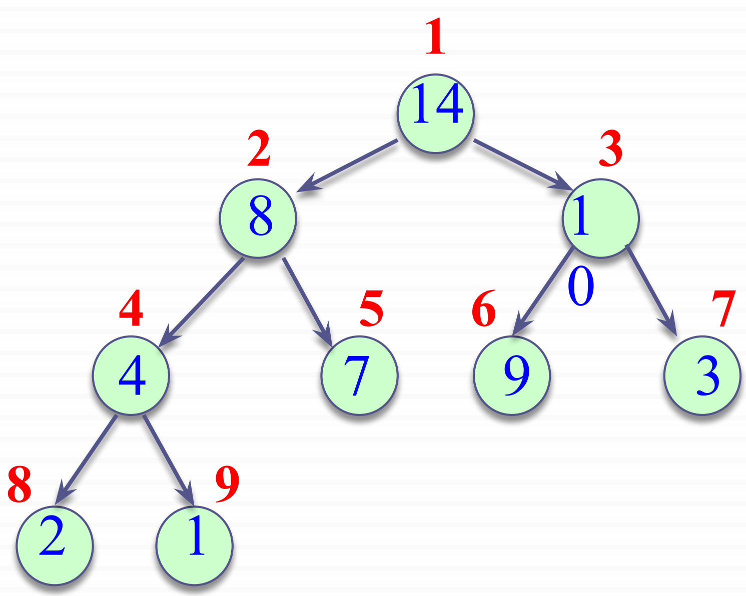
largest ← i

if $2i \le n$ **and** $A[2i] > A[i]$
then largest ← $2i$

if $2i + 1 \le n$ **and** $A[2i+1] >$ A[largest]
then largest ← $2i + 1$

if largest $\neq i$ **then**
exchange $A[i] \leftrightarrow$ A[largest]
HEAPIFY(A, largest, n)

after HEAPIFY:



Intuitive Analysis of HEAPIFY

- Consider HEAPIFY(A, i, n)
  - let $h(i)$ be the height of node i
  - at most $h(i)$ recursion levels
    - Constant work at each level: $\Theta(1)$
  - Therefore $T(i) = O(h(i))$
- Heap is almost-complete binary tree

  > $h(n) = O(\lg n)$
- Thus $\boxed{T(n) = O(\lg n)}$

Formal Analysis of HEAPIFY

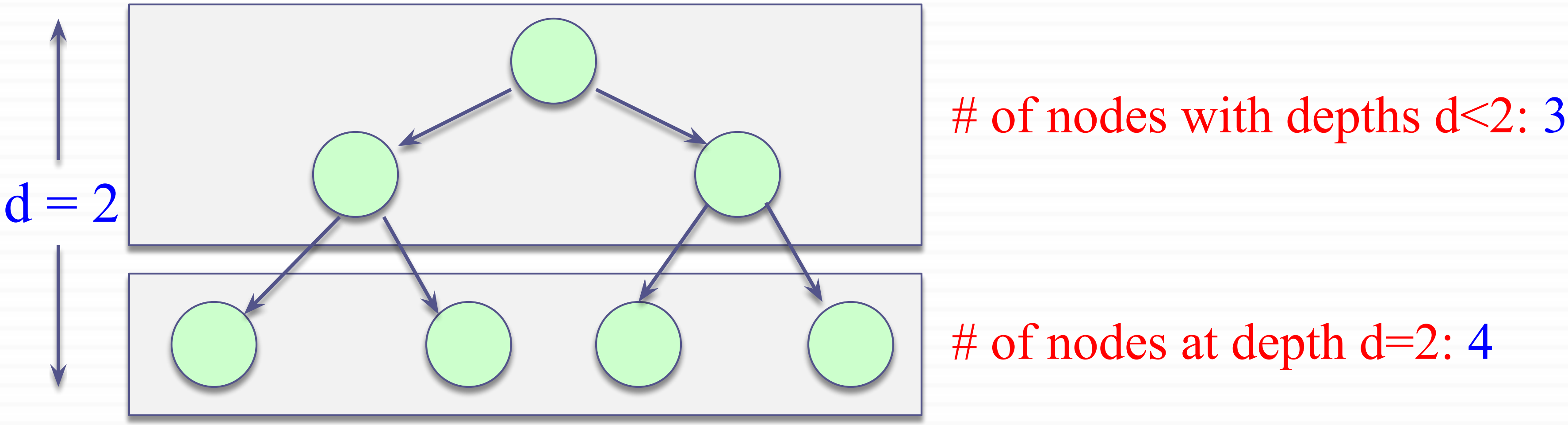
- What is the recurrence?
  - Depends on the size of the subtree on which recursive call is made
  - In the next couple of slides, we try to compute an upper bound for this subtree.

Reminder: Binary trees

For a *complete* binary tree:

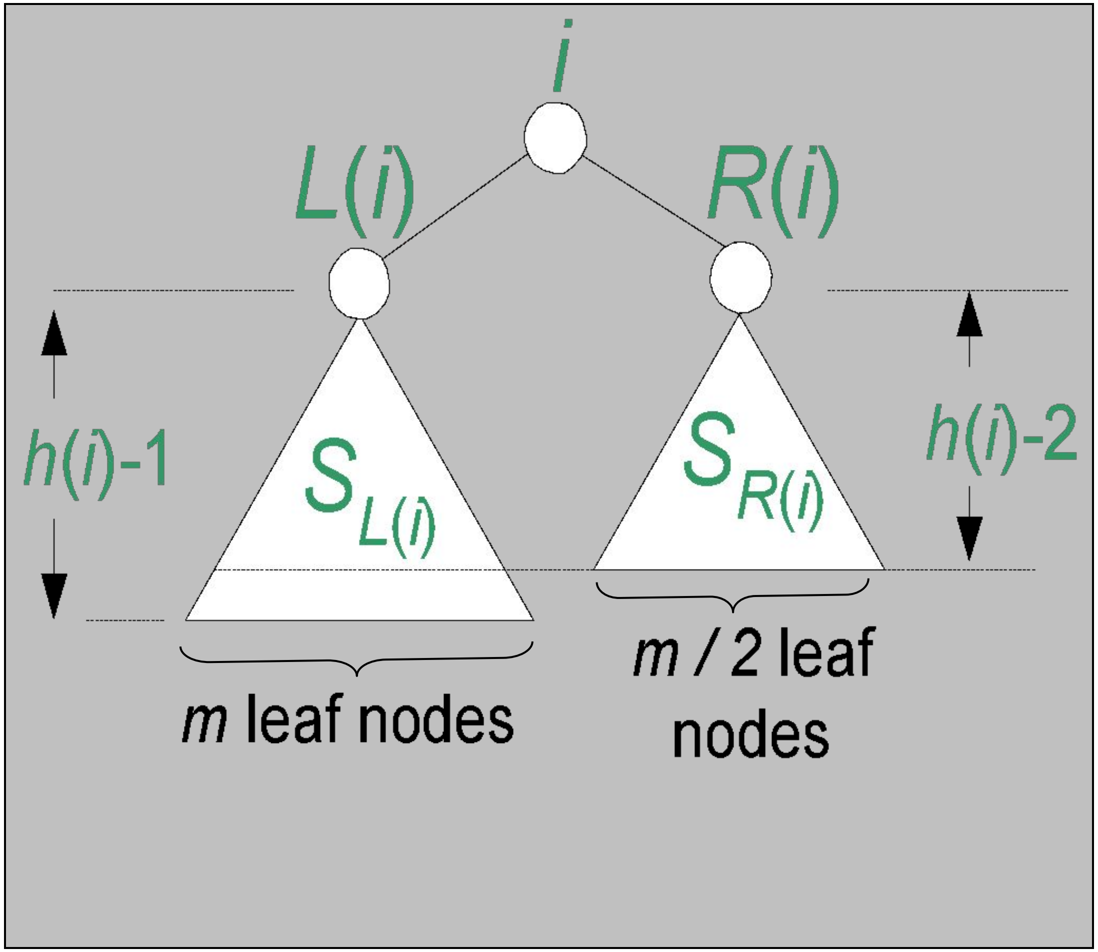
- # of nodes at depth d: 2^d
- # of nodes with depths less than d: 2^d-1

Example:



Formal Analysis of HEAPIFY

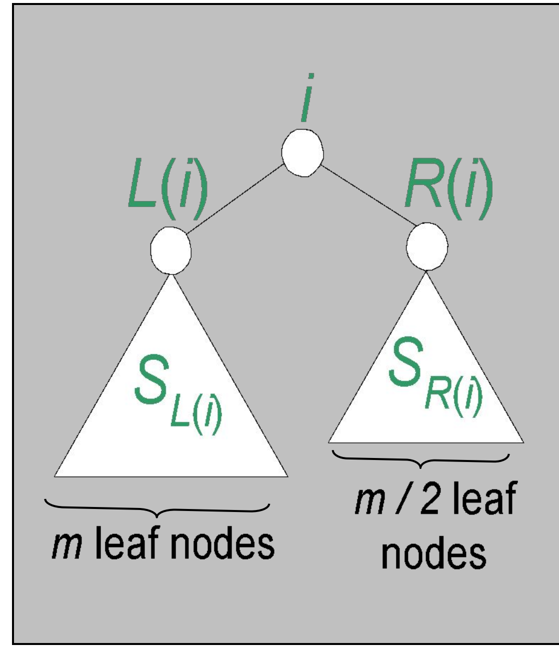
- Worst case occurs when last row of the subtree S_i rooted at node i is half full



- $T(n) \leq T(|S_{L(i)}|) + \Theta(1)$
- $S_{L(i)}$ and $S_{R(i)}$ are complete binary trees of heights $h(i) - 1$ and $h(i) - 2$, respectively

Formal Analysis of HEAPIFY

- Let m be the number of leaf nodes in $S_{L(i)}$
- $|S_{L(i)}| = \underbrace{m}_{\text{ext}} + \underbrace{(m-1)}_{\text{int}} = 2m - 1$;
- $|S_{R(i)}| = \overbrace{m/2} + \overbrace{(m/2 - 1)} = m - 1$



- $|S_{L(i)}| + |S_{R(i)}| + 1 = n$

  $(2m - 1) + (m - 1) + 1 = n \Rightarrow m = (n+1)/3$

  $|S_{L(i)}| = 2m - 1 = 2(n+1)/3 - 1 = (2n/3 + 2/3) - 1 = 2n/3 - 1/3 \leq 2n/3$

- $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow \boxed{T(n) = O(\lg n)}$

By case 2 of Master Thm

HEAPIFY: Efficiency Issues

- Recursion vs iteration:
  - In the absence of tail recursion, iterative version is in general more efficient
    - because of the pop/push operations to/from stack at each level of recursion.

Heap Operations: HEAPIFY

Recursive:

HEAPIFY(A, i, n)

largest ← i

if $2i \le n$ **and** $A[2i] > A[i]$
then largest ← $2i$

if $2i+1 \le n$ **and** $A[2i+1] > A[\text{largest}]$
then largest ← $2i+1$

if largest $\ne i$ **then**
exchange $A[i] \leftrightarrow A[\text{largest}]$
HEAPIFY(A, largest, n)

Iterative:

HEAPIFY(A, i, n)

j ← i
while (true) **do**
largest ← j

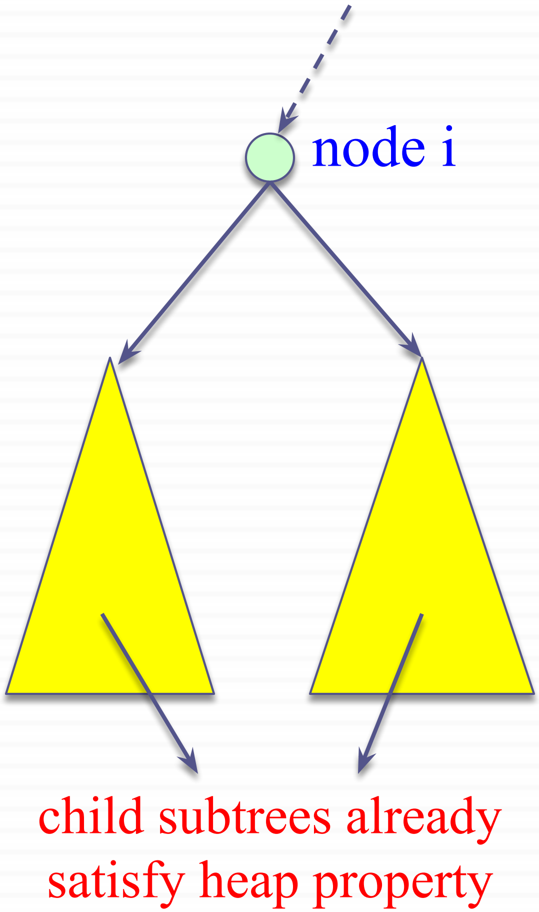
if $2j \le n$ **and** $A[2j] > A[j]$
then largest ← $2j$

if $2j+1 \le n$ **and** $A[2j+1] > A[\text{largest}]$
then largest ← $2j+1$

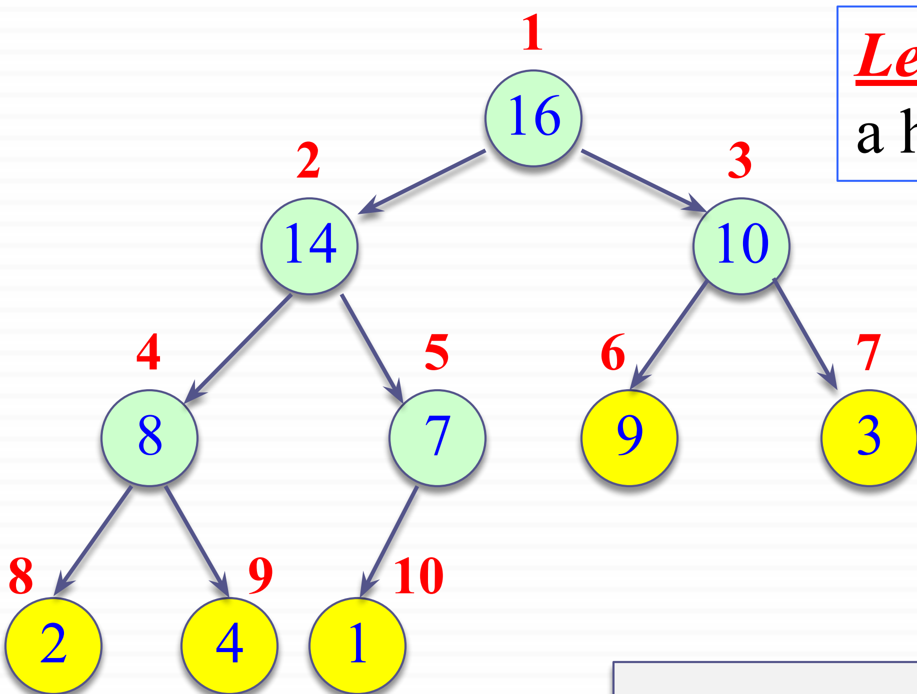
if largest $\ne j$ **then**
exchange $A[j] \leftrightarrow A[\text{largest}]$
j ← largest
else return

Heap Operations: Building Heap

- Given an arbitrary array, how to build a heap from scratch?
- Basic idea: Call HEAPIFY on each node bottom up
  - Start from the leaves (which trivially satisfy the heap property)
  - Process nodes in bottom up order.
  - When HEAPIFY is called on node i, the subtrees connected to the left and right subtrees already satisfy the heap property.



Where are the leaves stored?



Lemma: The last $\lceil n/2 \rceil$ nodes of a heap are *all leaves*

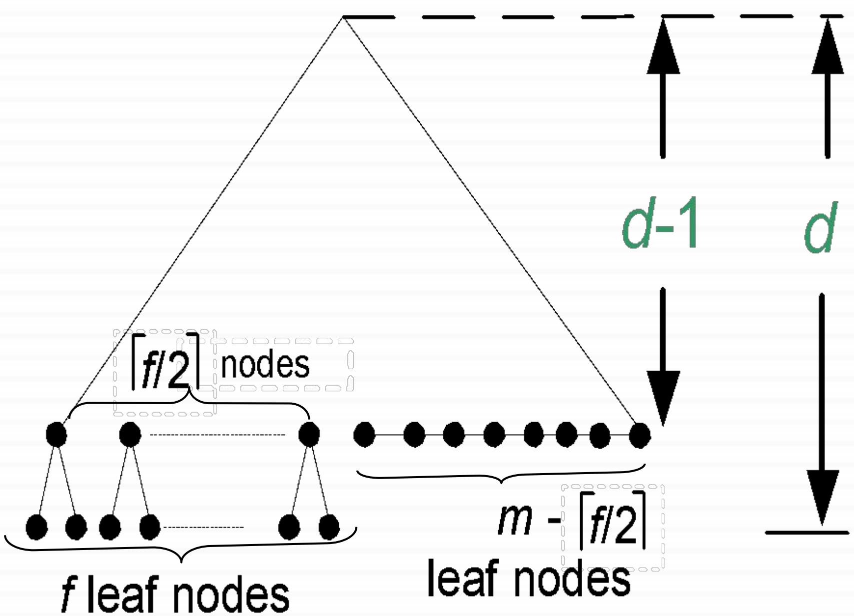
Storage

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

Proof of Lemma

Lemma: last $\lceil n/2 \rceil$ nodes of a heap are all leaves

Proof:



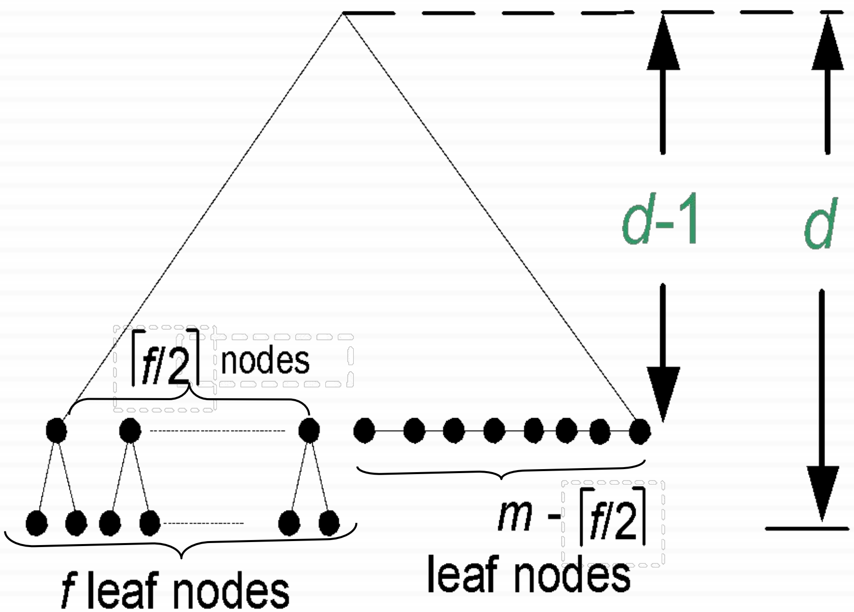
$m = 2^{d-1}$: # nodes at level $d - 1$

f : # nodes at level d (last level)

# of nodes with depth d-1: m
# of nodes with depth < d-1: m-1
# of nodes with depth d: f
Total # of nodes: n = f + 2m-1

Proof of Lemma (cont'd)

$f = n - 2m + 1$



# of leaves: $f + m - \lceil f/2 \rceil$

$= m + \lfloor f/2 \rfloor$

$= m + \lfloor (n-2m+1)/2 \rfloor$

$= \lfloor (n+1)/2 \rfloor$

$= \lceil n/2 \rceil$

Proof complete

Heap Operations: Building Heap

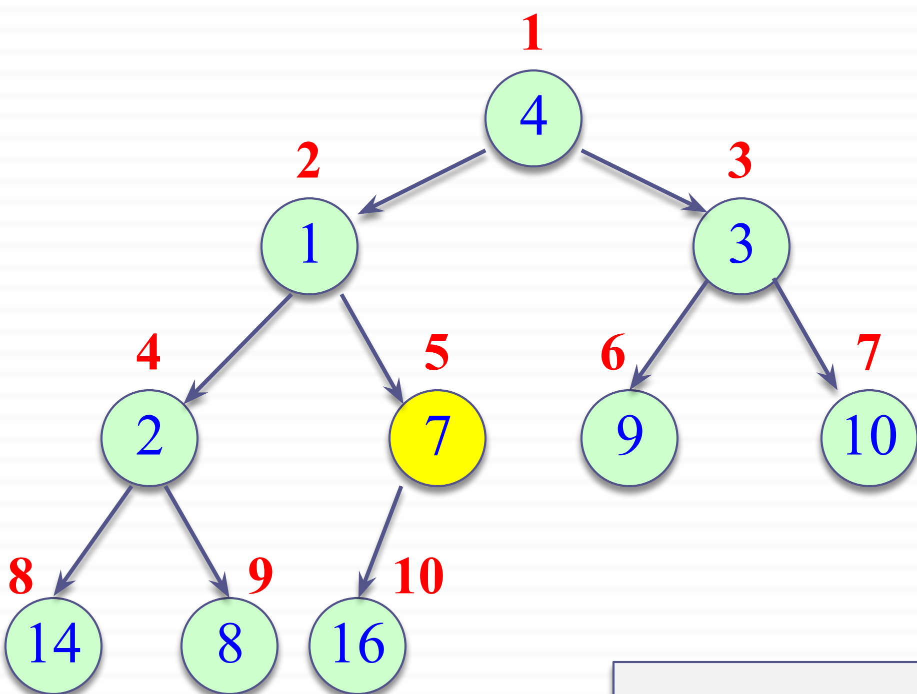
BUILD-HEAP (A, n)

for i = $\lfloor n/2 \rfloor$ **downto** 1 **do**

HEAPIFY(A, i, n)

Reminder: The last $\lceil n/2 \rceil$ nodes of a heap are *all leaves*, which trivially satisfy the heap property

Build-Heap: Example

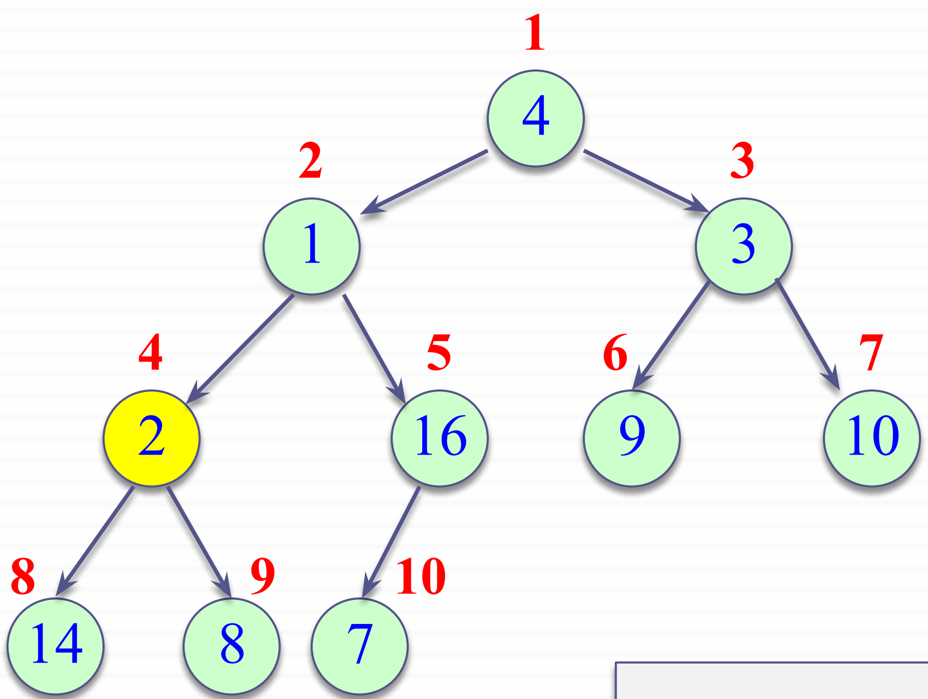


i=5

HEAPIFY(A, 5, 10)

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

Build-Heap: Example



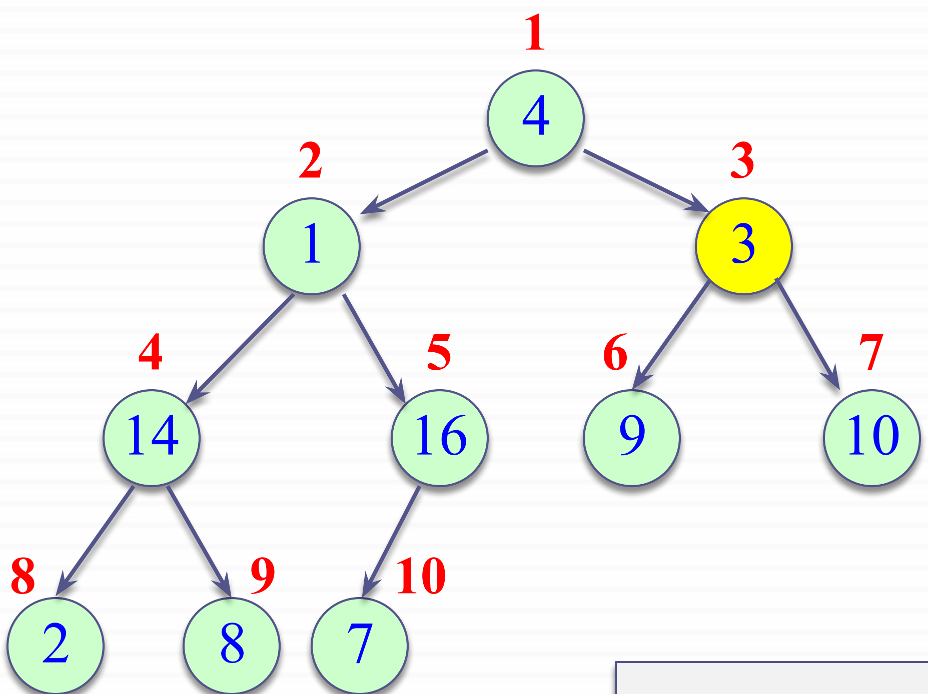
i=4

HEAPIFY(A, 4, 10)

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

Cevdet Aykanat and Mustafa Ozdal
Computer Engineering Department, Bilkent University

Build-Heap: Example

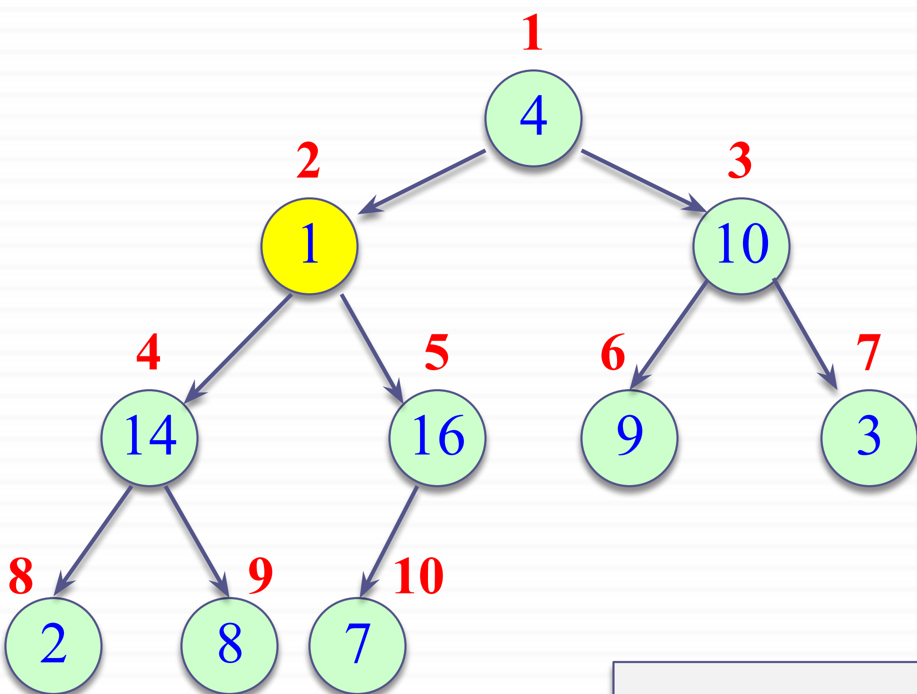


i=3

HEAPIFY(A, 3, 10)

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

Build-Heap: Example

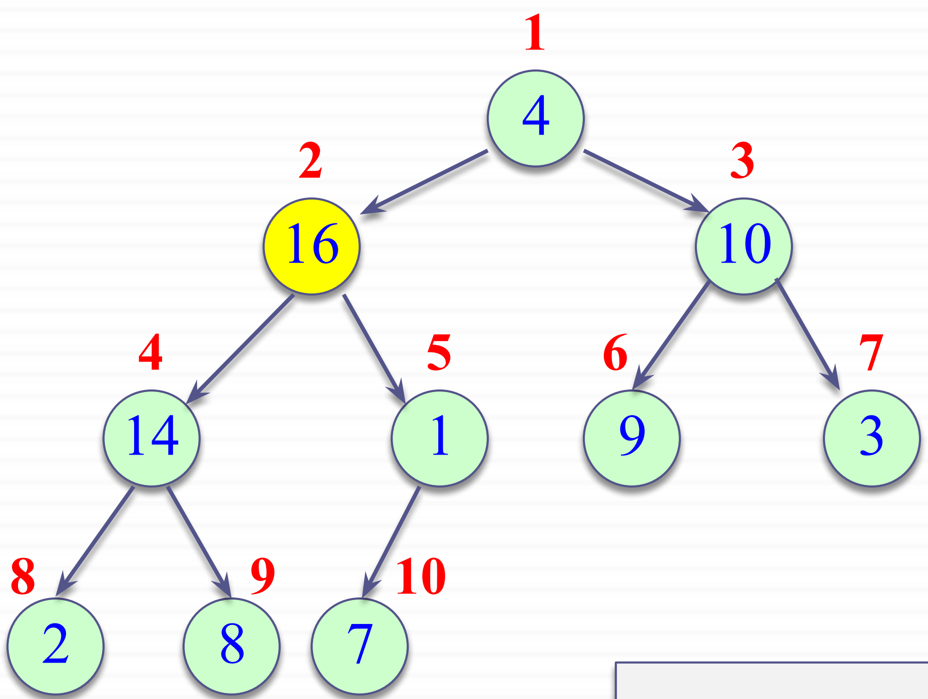


i=2

HEAPIFY(A, 2, 10)

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

Build-Heap: Example

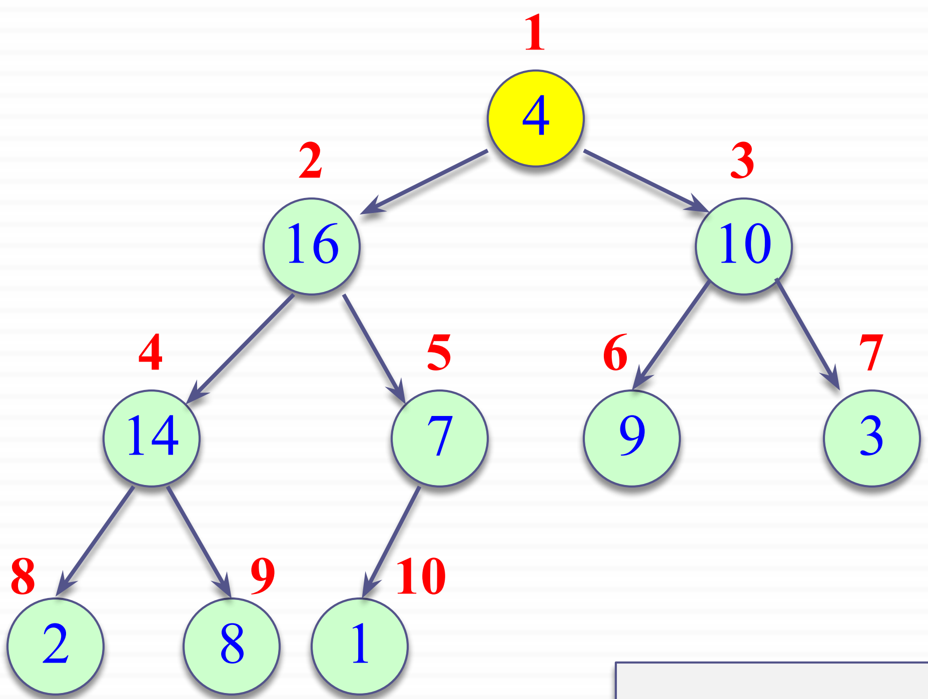


i=2 (cont'd)

HEAPIFY(A, 2, 10)

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

Build-Heap: Example

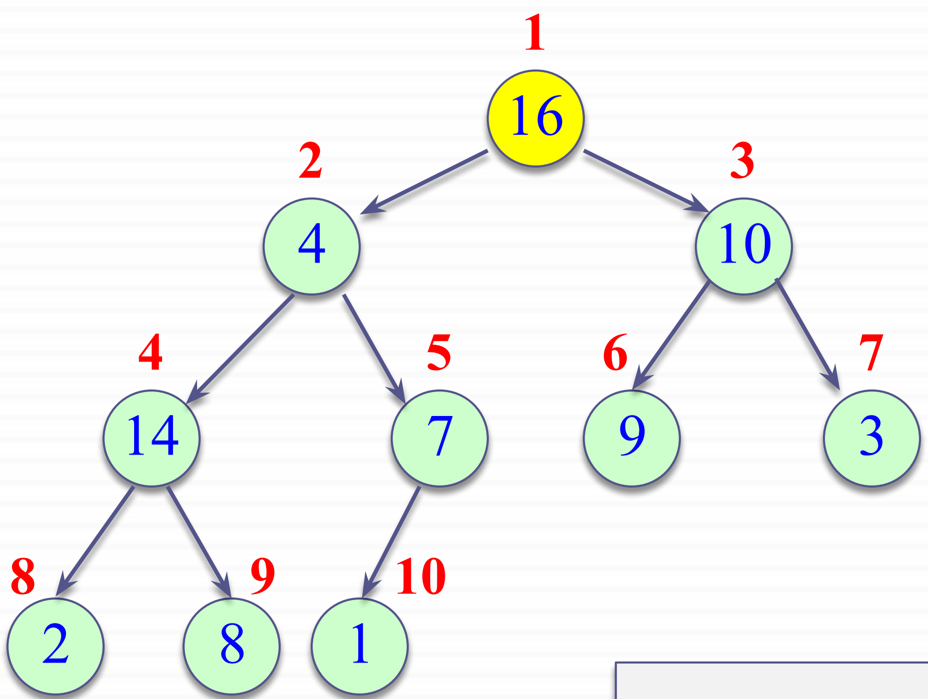


i=1

HEAPIFY(A, 1, 10)

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

Build-Heap: Example

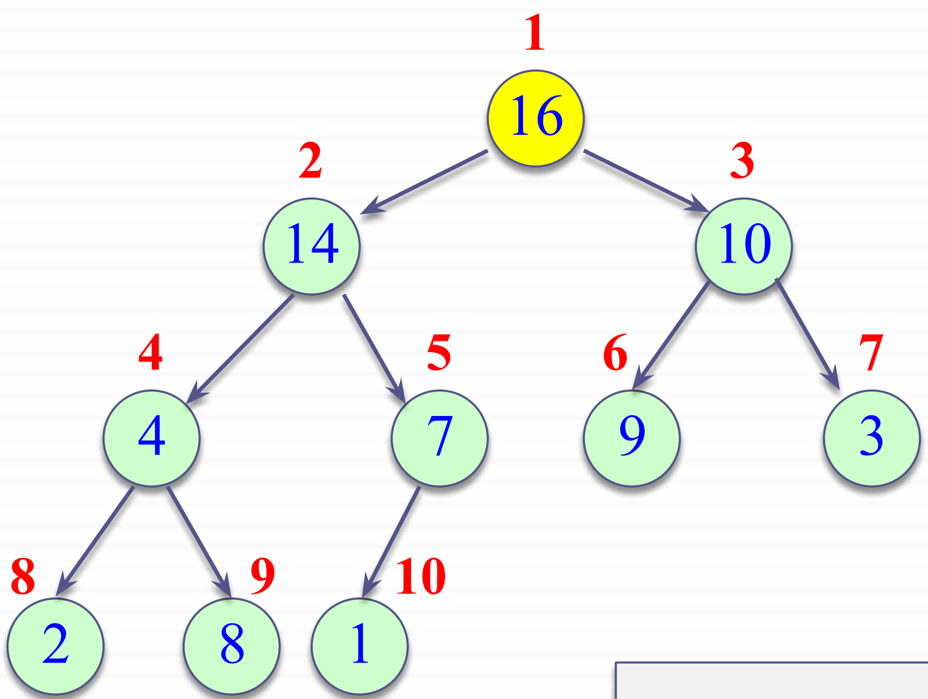


i=1 (cont'd)

HEAPIFY(A, 1, 10)

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

Build-Heap: Example



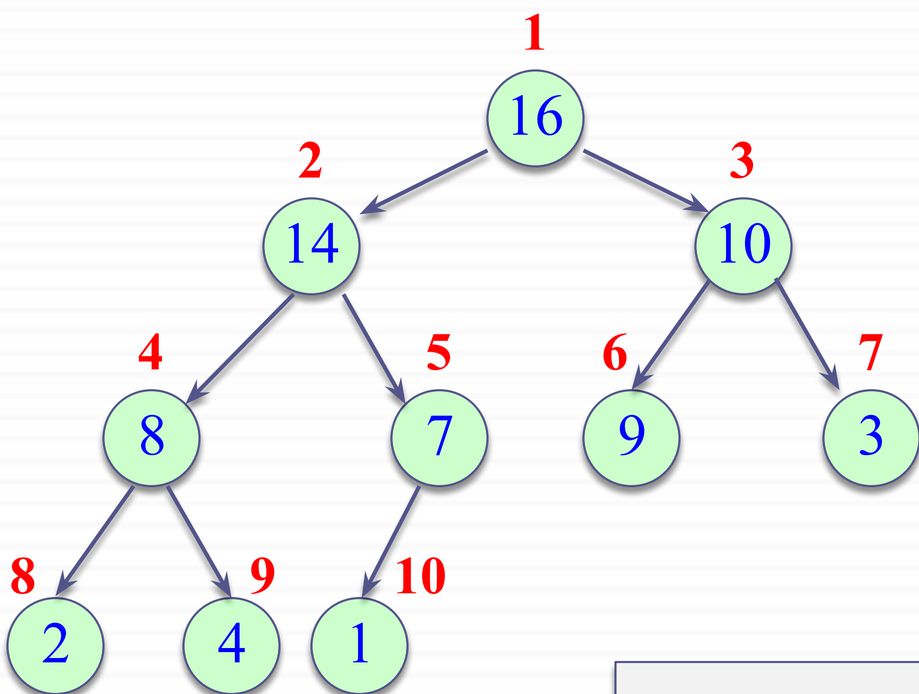
i=1 (cont'd)

HEAPIFY(A, 1, 10)

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

Build-Heap: Example

After Build-Heap



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

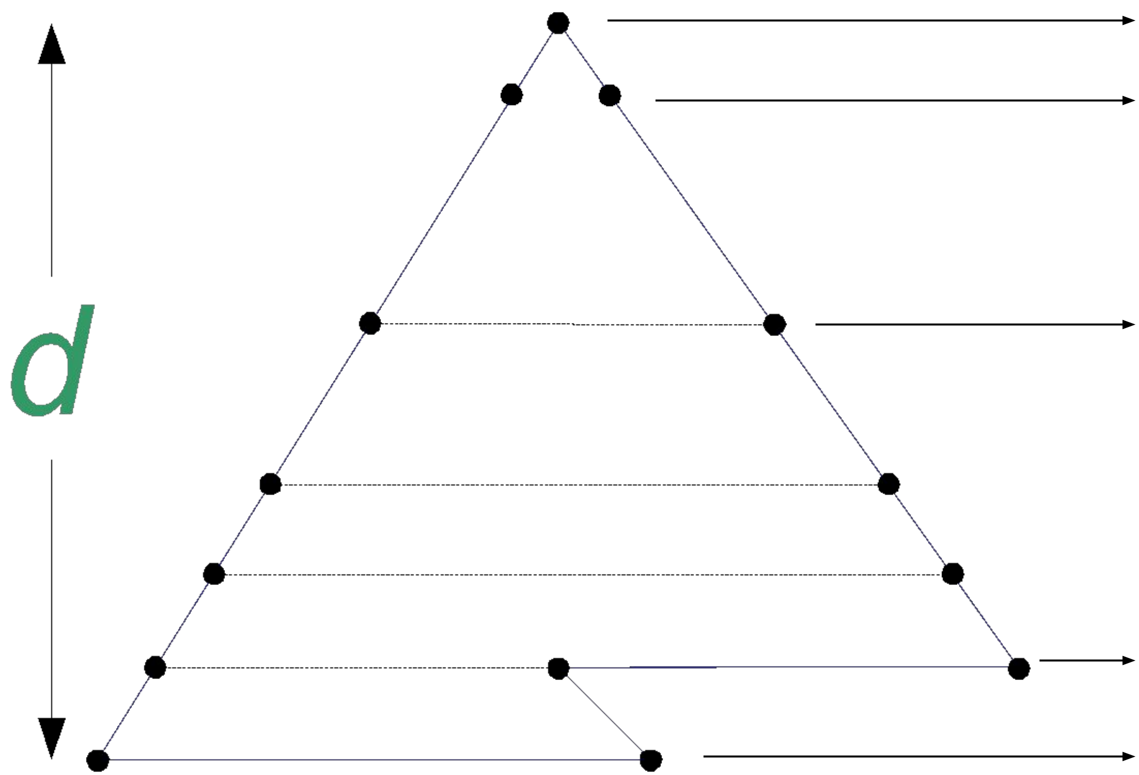
Build-Heap: Runtime Analysis

- Simple analysis:
  - O(n) calls to HEAPIFY, each of which takes O(lgn) time
  - O(nlgn) → loose bound

- In general, a good approach:
  - Start by proving an easy bound
  - Then, try to tighten it

- Is there a tighter bound?

Build-Heap: tighter running time analysis



$l = 0,\ h_0 = d$

$l = 1,\ d - 2 \le h_1 \le d - 1$

$l,\ d - l - 1 \le h_l \le d - l$

$l = d - 1,\ 0 \le h_{d-1} \le 1$

$l = d,\ h_d = 0$

If the heap is complete binary tree then $h_l = d - l$

Otherwise, nodes at a given level do not all have the same height

But we have $d - l - 1 \le h_l \le d - l$

Build-Heap: **tighter** running time analysis

Assume that all nodes at level $l = d - 1$ are processed

$$T(n) = \sum_{l=0}^{d-1} n_l \, O(h_l) = O\left(\sum_{l=0}^{d-1} n_l h_l\right) \qquad \begin{cases} n_l = 2^l = \text{\# of nodes at level } l \\ h_l = \text{height of nodes at level } l \end{cases}$$

$$\therefore T(n) = O\left(\sum_{l=0}^{d-1} 2^l (d - l)\right)$$

Let $h = d - l \Rightarrow l = d - h$ (change of variables)

$$T(n) = O\left(\sum_{h=1}^{d} h\, 2^{d-h}\right) = O\left(\sum_{h=1}^{d} h\, 2^d/2^h\right) = O\left(2^d \sum_{h=1}^{d} h\, (1/2)^h\right)$$

but $2^d = \Theta(n) \Rightarrow T(n) = O\left(n \sum_{h=1}^{d} h\, (1/2)^h\right)$

Build-Heap: **tighter** running time analysis

$$\sum_{h=1}^{d} h(1/2)^h \leq \sum_{h=0}^{d} h(1/2)^h \leq \sum_{h=0}^{\infty} h(1/2)^h$$

recall infinite decreasing geometric series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \text{ where } |x| < 1$$

differentiate both sides

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

Build-Heap: **tighter** running time analysis

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

then, multiply both sides by x

$$\sum_{k=0}^{\infty} kx^{k} = \frac{x}{(1-x)^2}$$

in our case: $x = 1/2$ and $k = h$

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

$$\therefore T(n) = O(n\sum_{h=1}^{d} h(1/2)^h) = O(n)$$

Heapsort Algorithm

The HEAPSORT algorithm

(1) Build a heap on array $A[1\ldots n]$ by calling BUILD-HEAP(A, n)

(2) The largest element is stored at the root $A[1]$

Put it into its correct final position $A[n]$ by $A[1] \leftrightarrow A[n]$

(3) Discard node n from the heap

(4) Subtrees (S_2 & S_3) rooted at children of root remain as heaps

but the new root element may violate the heap property

Make $A[1\ldots n-1]$ a heap by calling HEAPIFY(A, 1, $n-1$)

(5) $n \leftarrow n - 1$

(6) Repeat steps 2–4 until $n = 2$

Heapsort Algorithm

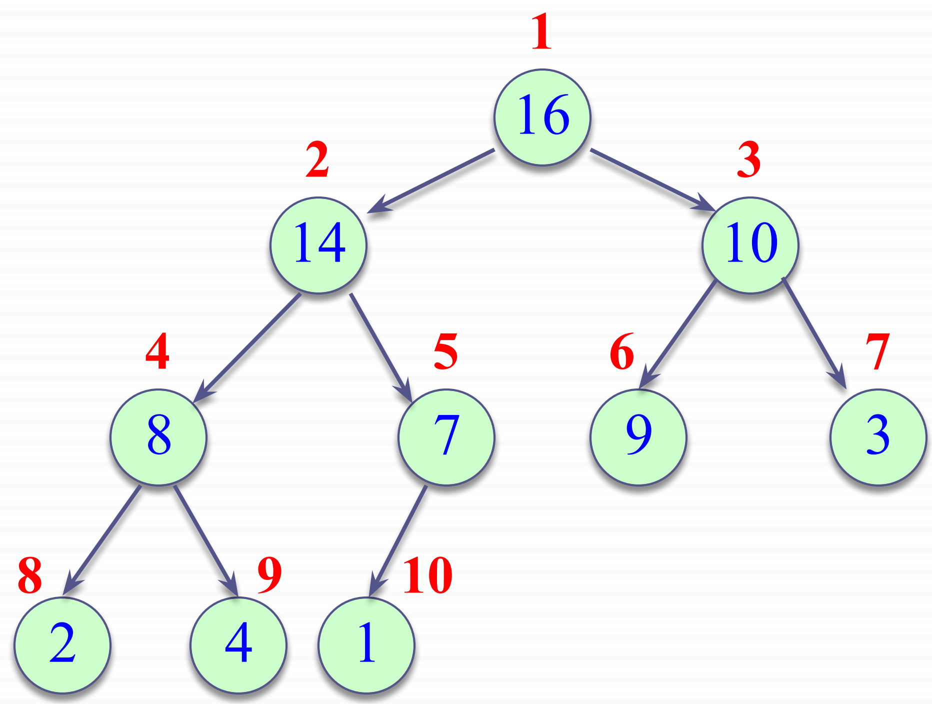
HEAPSORT(A, n)

BUILD-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**

exchange A[1] $\leftrightarrow$ A[i]

HEAPIFY(A, 1, $i-1$)



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

Heapsort Algorithm

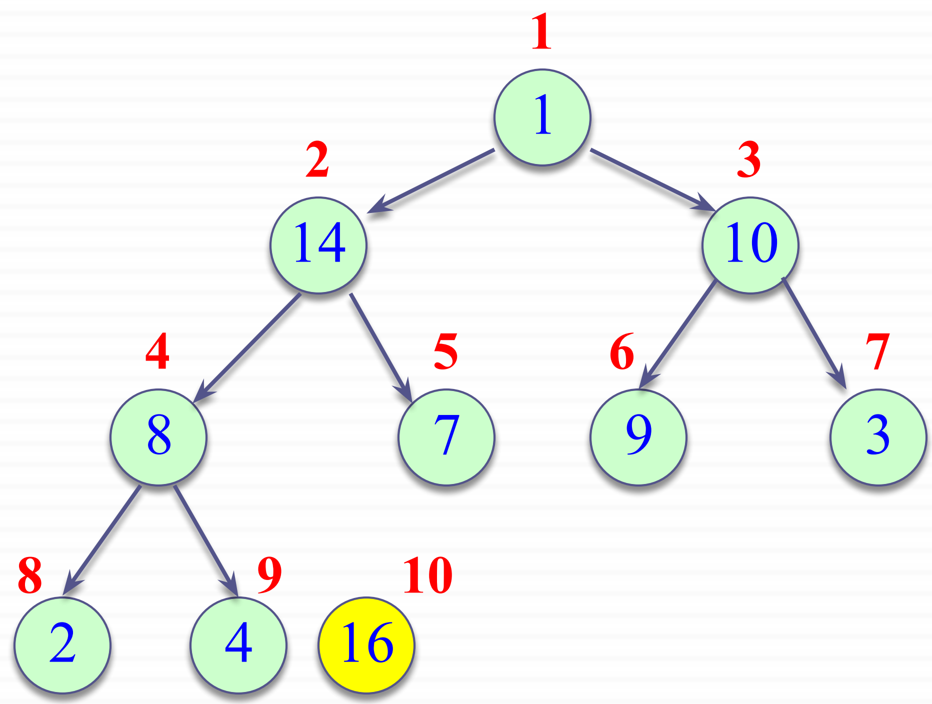
HEAPSORT(A, n)

BUILD-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**

exchange A[1] $\leftrightarrow$ A[i]

HEAPIFY(A, 1, $i - 1$)



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

Heapsort Algorithm

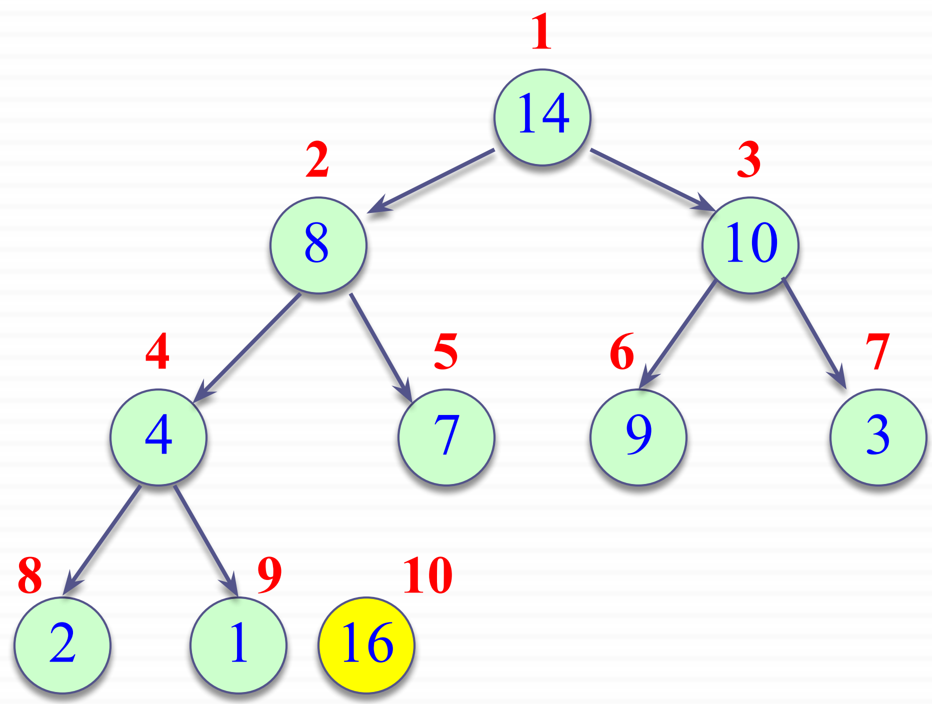
HEAPSORT(A, n)

BUILD-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**

exchange A[1] $\leftrightarrow$ A[i]

HEAPIFY(A, 1, $i - 1$)



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

Heapsort Algorithm

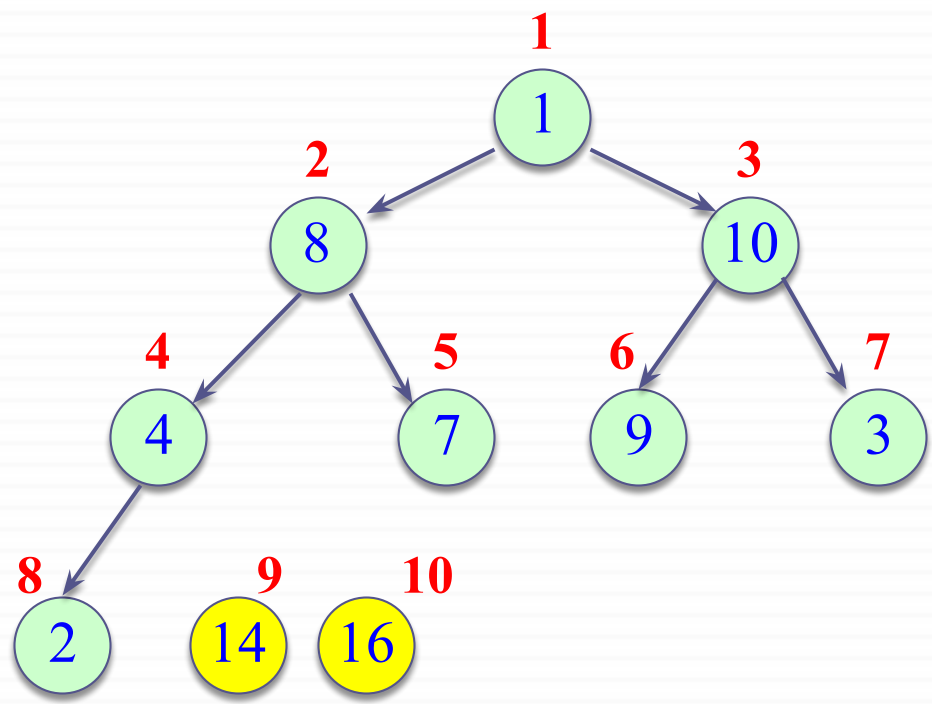
HEAPSORT(A, n)

BUILD-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**

exchange A[1] $\leftrightarrow$ A[i]

HEAPIFY(A, 1, i −1)



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

Heapsort Algorithm

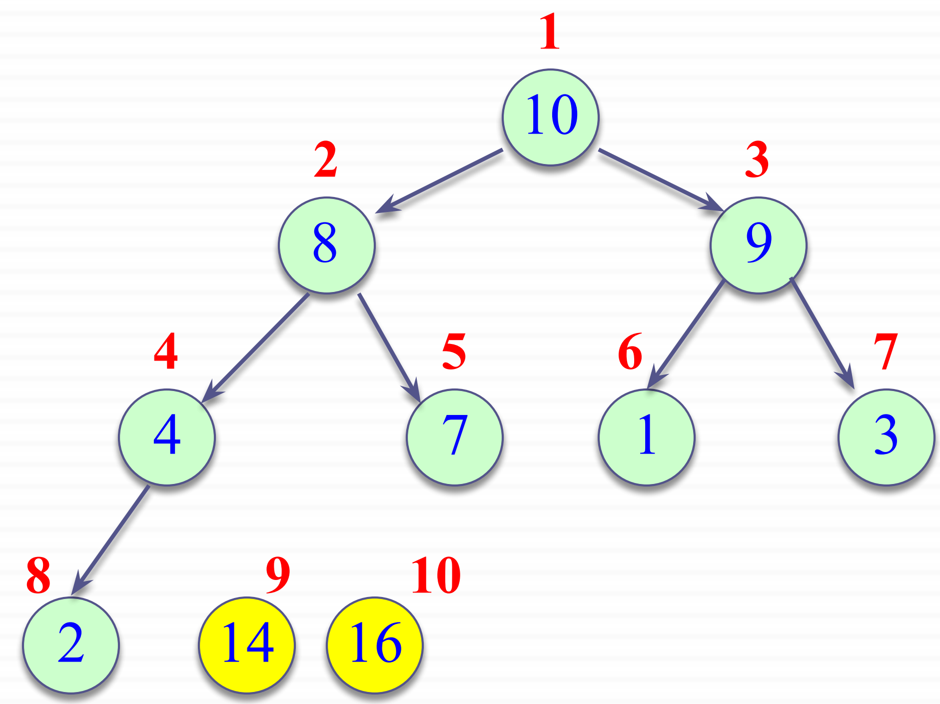
HEAPSORT(A, n)

BUILD-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**

exchange A[1] $\leftrightarrow$ A[i]

HEAPIFY(A, 1, i −1)



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

Heapsort Algorithm

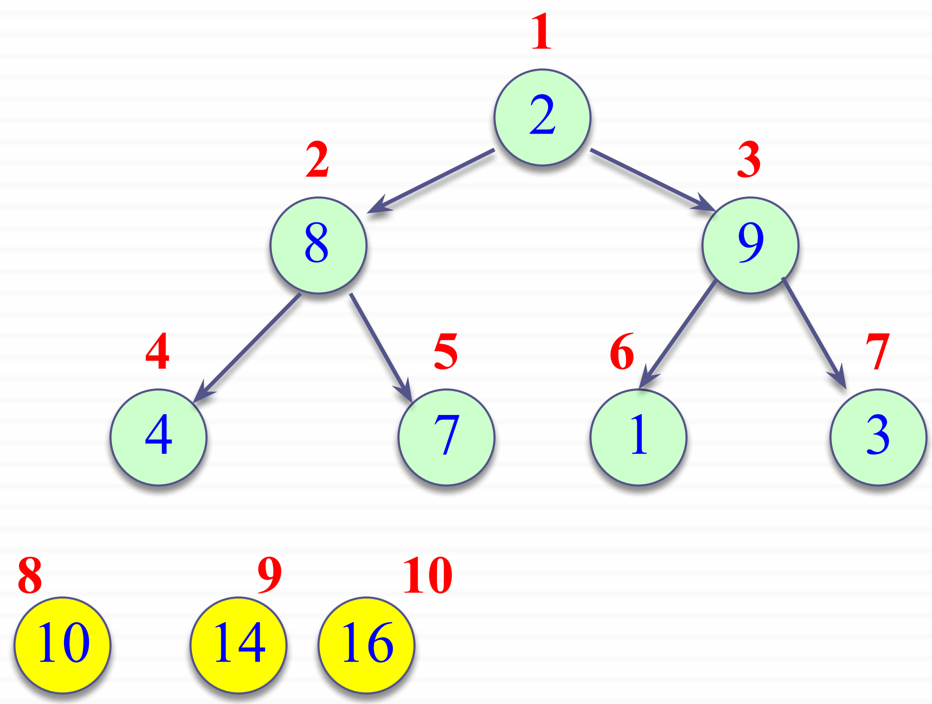
HEAPSORT(A, n)

BUILD-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**

exchange A[1] $\leftrightarrow$ A[i]

HEAPIFY(A, 1, i −1)



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

Heapsort Algorithm

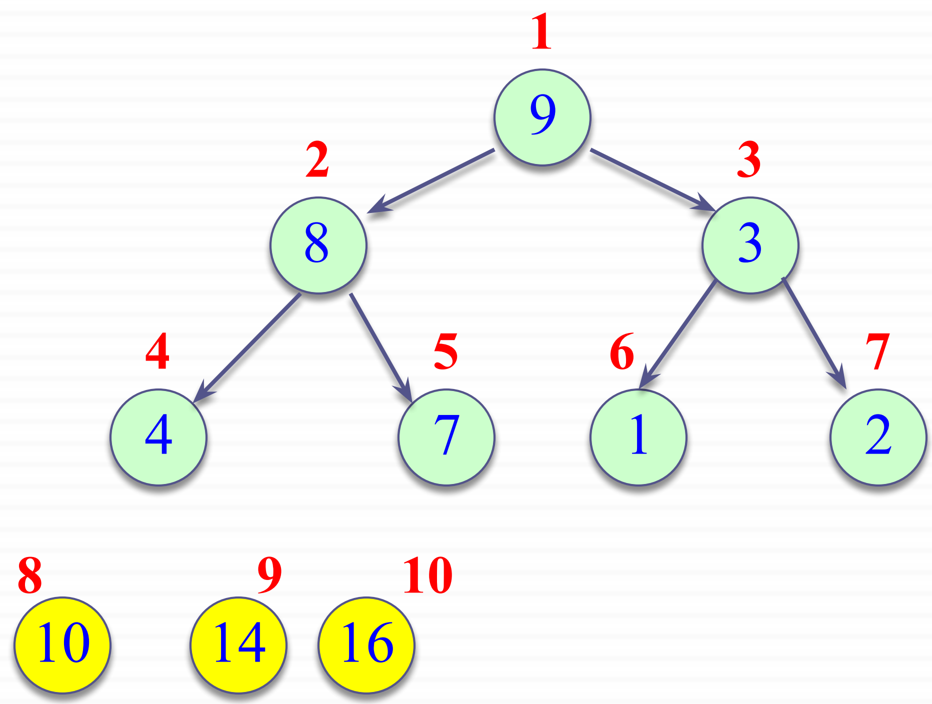
HEAPSORT(A, n)

BUILD-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**

exchange A[1] $\leftrightarrow$ A[i]

HEAPIFY(A, 1, i −1)



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

Heapsort Algorithm

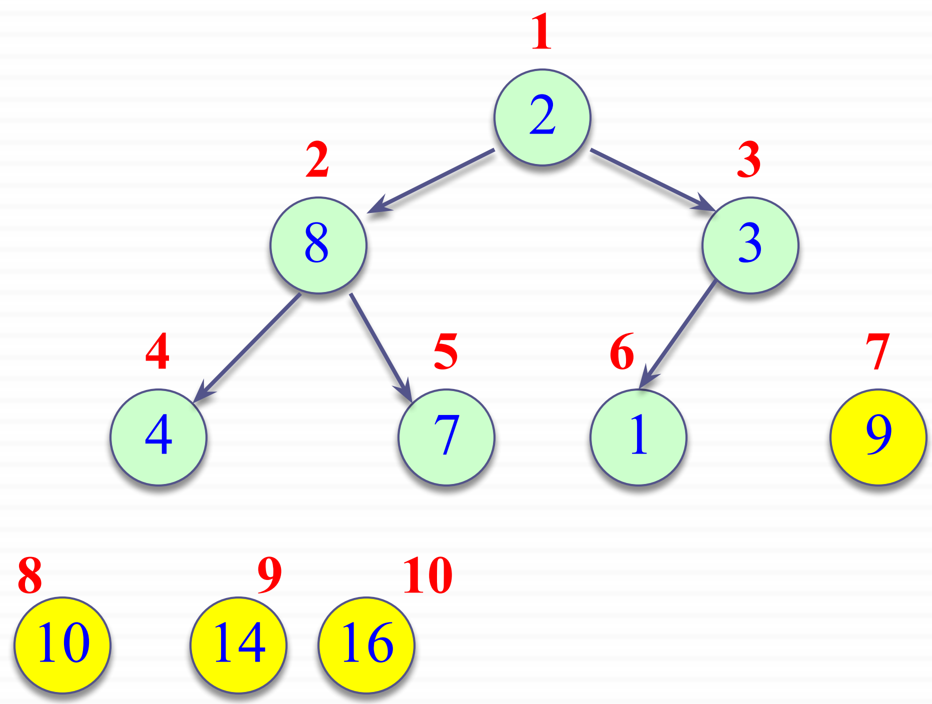
HEAPSORT(A, n)

BUILD-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**

exchange A[1] $\leftrightarrow$ A[i]

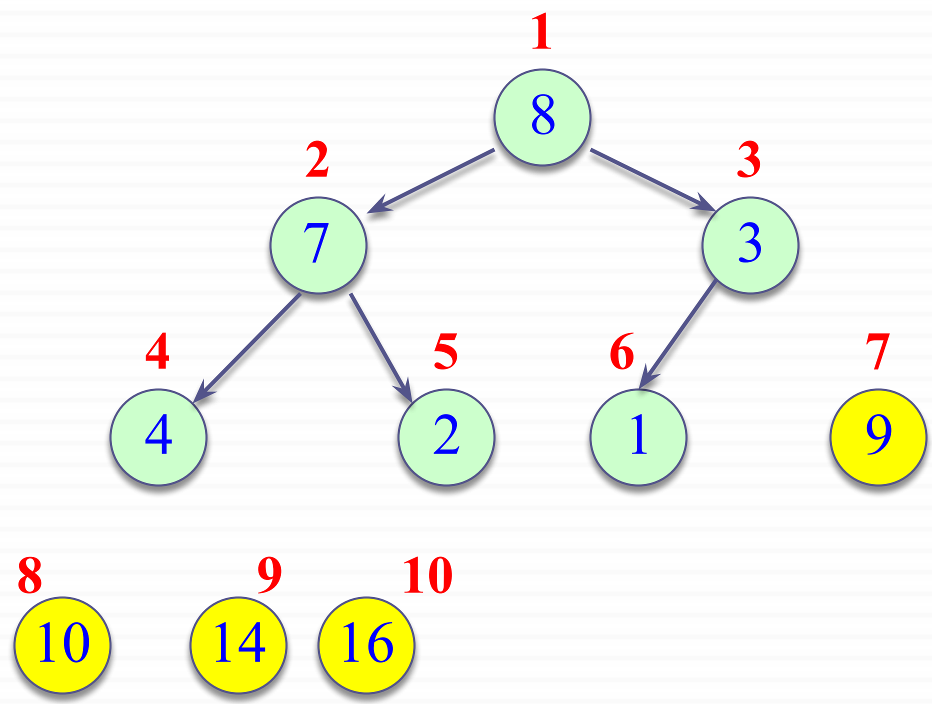
HEAPIFY(A, 1, i −1)



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

Heapsort Algorithm

HEAPSORT(A, n)
BUILD-HEAP(A, n)
for $i \leftarrow n$ **downto** 2 **do**
exchange A[1] $\leftrightarrow$ A[i]
HEAPIFY(A, 1, i −1)

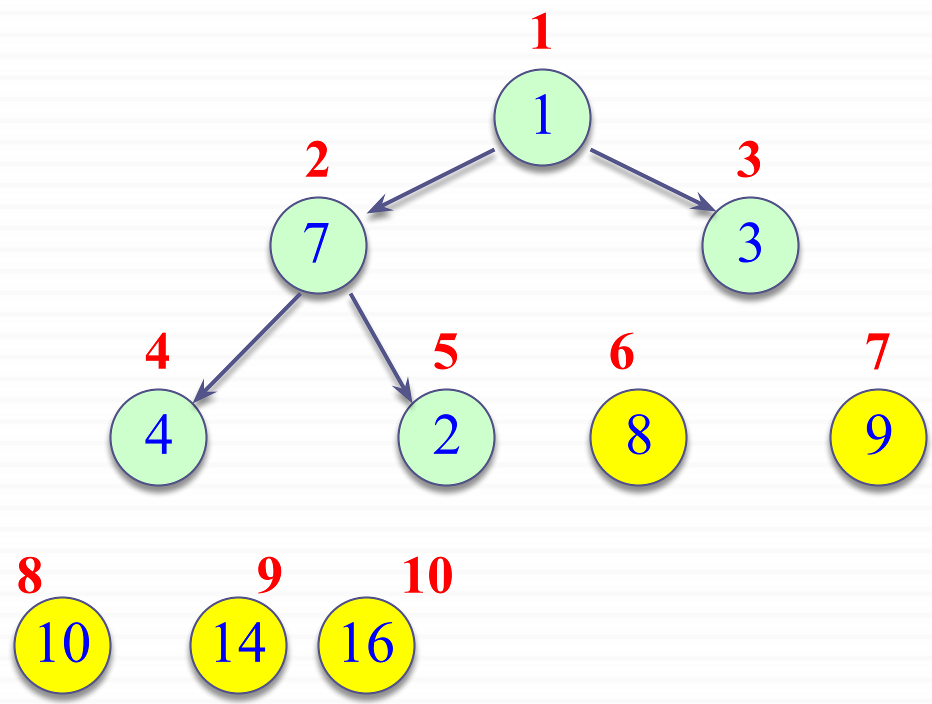


	1	2	3	4	5	6	7	8	9	10
A	8	7	3	4	2	1	9	10	14	16

Heapsort Algorithm

HEAPSORT(A, n)

```
BUILD-HEAP(A, n)
for i ← n downto 2 do
    exchange A[1] ↔ A[i]
    HEAPIFY(A, 1, i −1)
```



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

Heapsort Algorithm

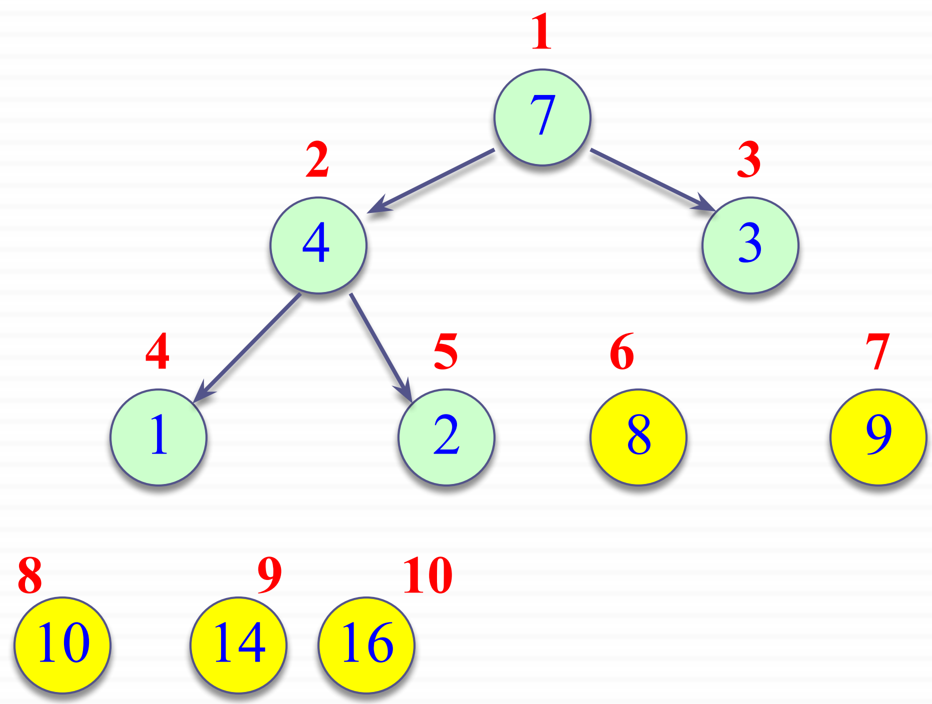
HEAPSORT(A, n)

BUILD-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**

  exchange A[1] $\leftrightarrow$ A[i]

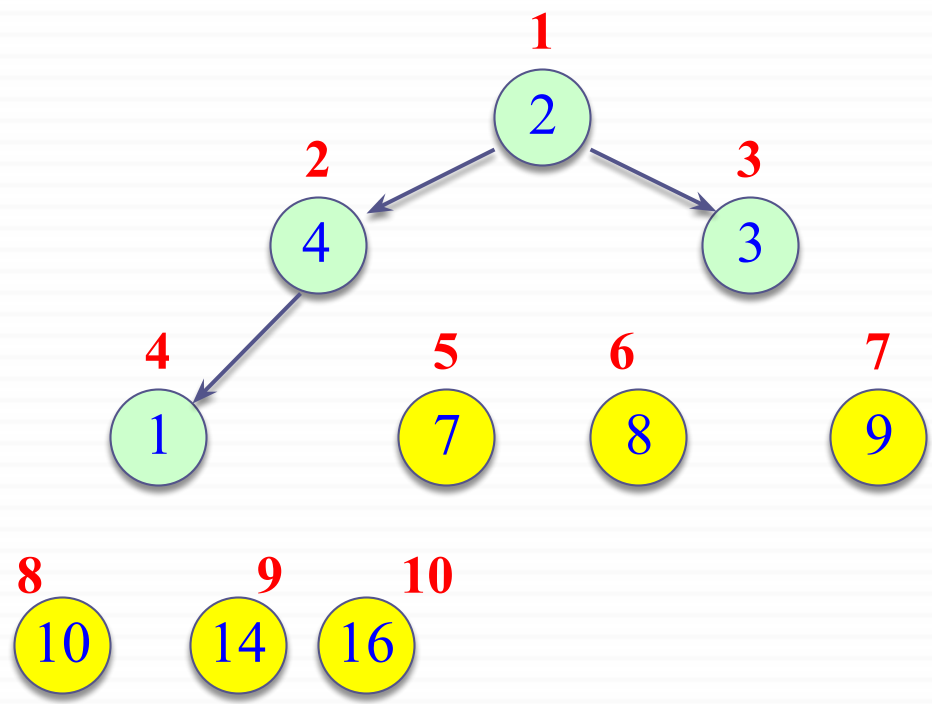
  HEAPIFY(A, 1, $i-1$)



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

Heapsort Algorithm

HEAPSORT(A, n)
BUILD-HEAP(A, n)
for $i \leftarrow n$ **downto** 2 **do**
exchange A[1] $\leftrightarrow$ A[i]
HEAPIFY(A, 1, $i-1$)



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

Heapsort Algorithm

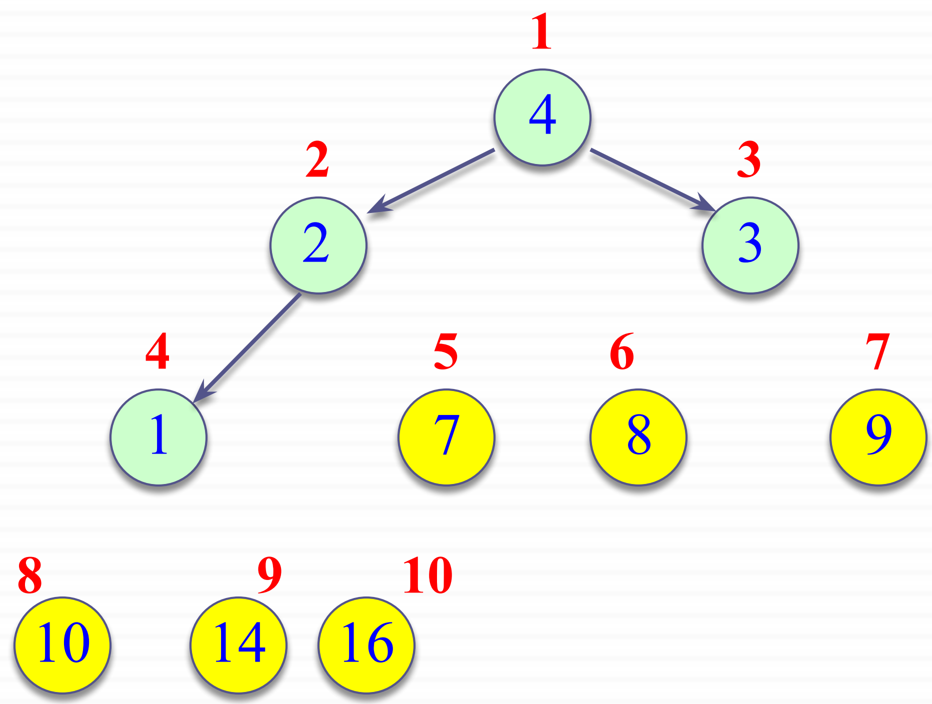
HEAPSORT(A, n)

BUILD-HEAP(A, *n*)

for *i* ← *n* **downto** 2 **do**

exchange A[1] ↔ A[*i*]

HEAPIFY(A, 1, *i* −1)



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

Heapsort Algorithm

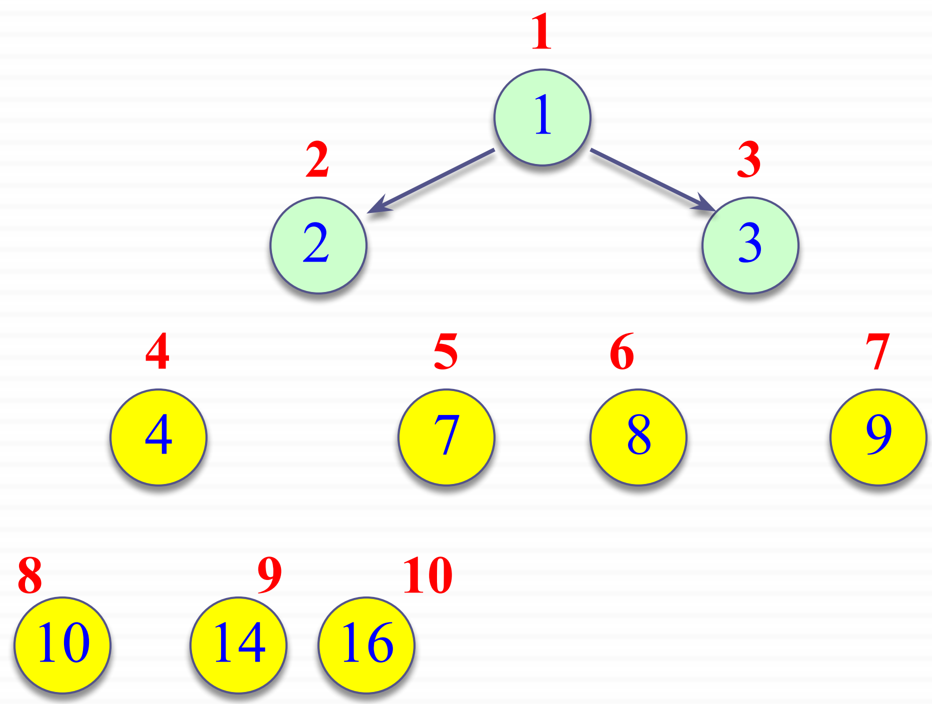
HEAPSORT(A, n)

BUILD-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**

exchange A[1] ↔ A[i]

HEAPIFY(A, 1, $i - 1$)

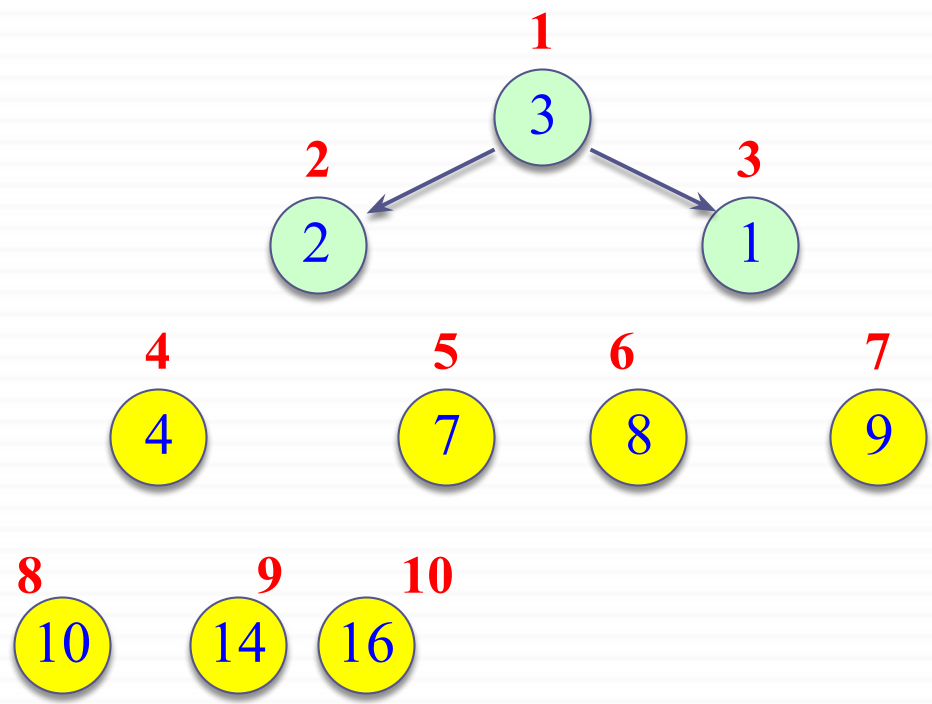


	1	2	3	4	5	6	7	8	9	10
A	1	2	3	4	7	8	9	10	14	16

Heapsort Algorithm

HEAPSORT(A, n)

> BUILD-HEAP(A, n)
> **for** $i \leftarrow n$ **downto** 2 **do**
>   **exchange** A[1] $\leftrightarrow$ A[i]
>   HEAPIFY(A, 1, $i-1$)

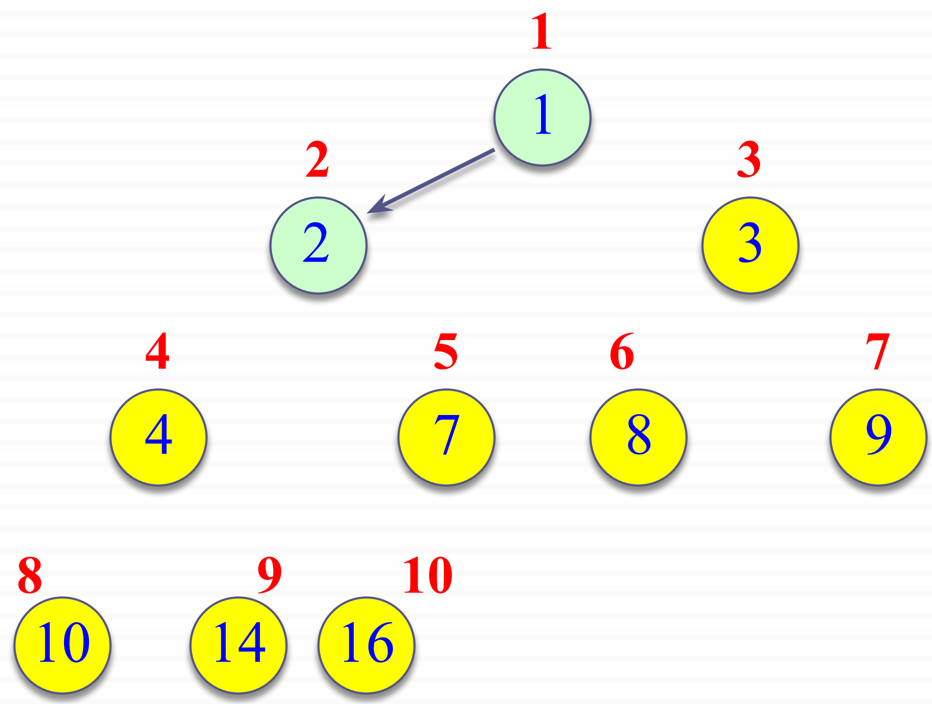


	1	2	3	4	5	6	7	8	9	10
A	3	2	1	4	7	8	9	10	14	16

Heapsort Algorithm

HEAPSORT(A, n)

```
BUILD-HEAP(A, n)
for i ← n downto 2 do
    exchange A[1] ↔ A[i]
    HEAPIFY(A, 1, i −1)
```



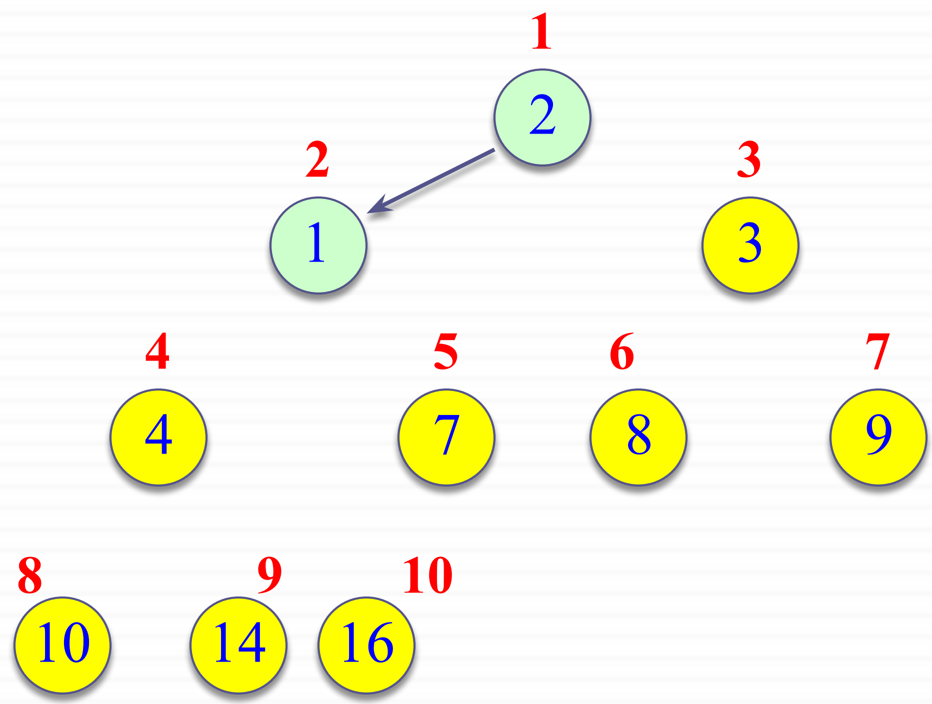
	1	2	3	4	5	6	7	8	9	10
A	1	2	3	4	7	8	9	10	14	16

Cevdet Aykanat and Mustafa Ozdal
Computer Engineering Department, Bilkent University

Heapsort Algorithm

HEAPSORT(A, n)

BUILD-HEAP(A, n)
for $i \leftarrow n$ **downto** 2 **do**
  exchange A[1] $\leftrightarrow$ A[i]
  HEAPIFY(A, 1, i −1)

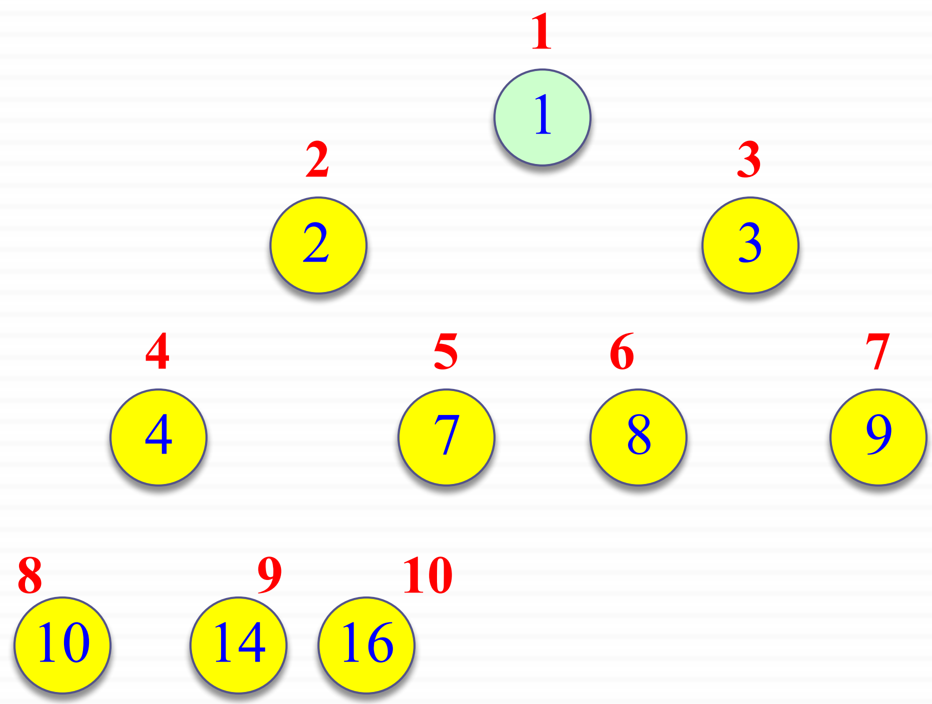


	1	2	3	4	5	6	7	8	9	10
A	2	1	3	4	7	8	9	10	14	16

Heapsort Algorithm

HEAPSORT(A, n)

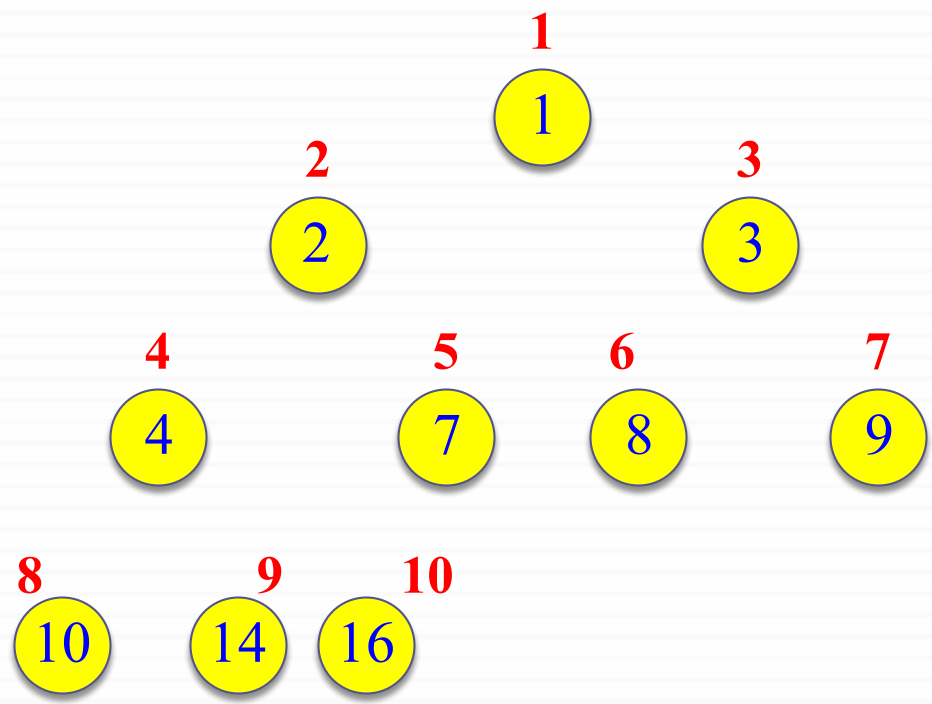
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ **downto** 2 **do**
        exchange $A[1] \leftrightarrow A[i]$
        HEAPIFY(A, 1, $i-1$)



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

Heapsort Algorithm

HEAPSORT(A, n)
BUILD-HEAP(A, n)
for $i \leftarrow n$ **downto** 2 **do**
exchange A[1] $\leftrightarrow$ A[i]
HEAPIFY(A, 1, $i - 1$)



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

Heapsort Algorithm: Runtime Analysis

HEAPSORT(A, n)
  BUILD-HEAP(A, n) ---------------- $\Theta(n)$
  for $i \leftarrow n$ **downto** 2 **do**
    exchange A[1] $\leftrightarrow$ A[i] ---------------- $\Theta(1)$
    HEAPIFY(A, 1, $i-1$) ---------------- O(lg(i-1))

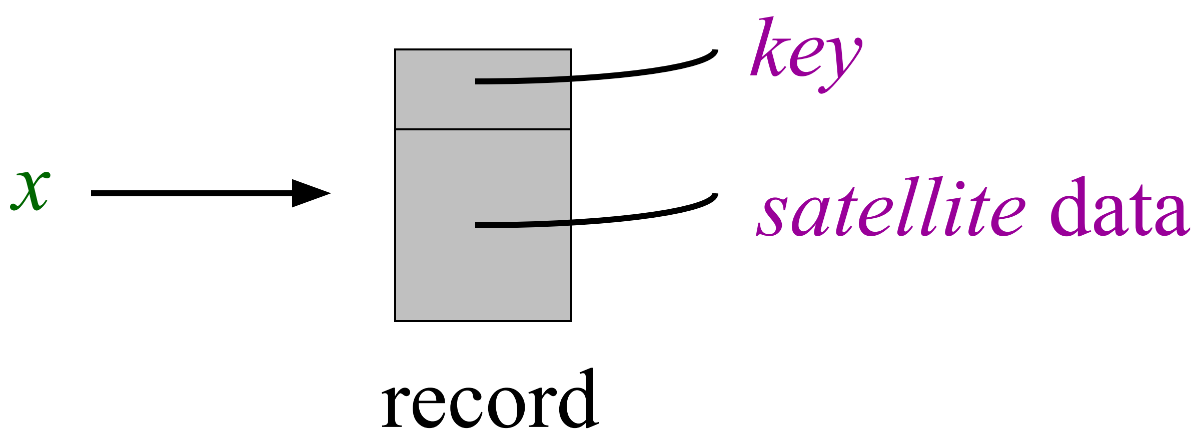
$$T(n) = \Theta(n) + \sum_{i=2}^{n} O(\lg i) = \Theta(n) + O\left(\sum_{i=2}^{n} O(\lg n)\right) = O(n \lg n)$$

Heapsort - Notes

- Heapsort is a very good algorithm but, a good implementation of quicksort always beats heapsort in practice
- However, heap data structure has many popular applications, and it can be efficiently used for implementing priority queues

Data structures for Dynamic Sets

- Consider sets of records having *key* and *satellite* data



Operations on Dynamic Sets

- Queries: Simply return info; Modifying operations: Change the set

  - INSERT(S, x): (Modifying) $S \leftarrow S \cup \{x\}$
  - DELETE(S, x): (Modifying) $S \leftarrow S - \{x\}$
  - MAX(S) / MIN(S): (Query) return $x \in S$ with the largest/smallest *key*
  - EXTRACT-MAX(S) / EXTRACT-MIN(S) : (Modifying) return and delete $x \in S$ with the largest/smallest *key*
  - SEARCH(S, k): (Query) return $x \in S$ with $key[x] = k$
  - SUCCESSOR(S, x) / PREDECESSOR(S, x) : (Query) return $y \in S$ which is the next larger/smaller element after x

- Different data structures support/optimize different operations

Priority Queues (*PQ*)

- Supports
  - INSERT
  - MAX / MIN
  - EXTRACT-MAX / EXTRACT-MIN
- One application: Schedule jobs on a shared resource
  - PQ keeps track of jobs and their relative priorities
  - When a job is finished or interrupted, highest priority job is selected from those pending using EXTRACT-MAX
  - A new job can be added at any time using INSERT

Priority Queues

- Another application: Event-driven simulation
  - Events to be simulated are the items in the PQ
  - Each event is associated with a time of occurrence which serves as a *key*
  - Simulation of an event can cause other events to be simulated in the future
  - Use EXTRACT-MIN at each step to choose the next event to simulate
  - As new events are produced insert them into the PQ using INSERT

Implementation of Priority Queue

- Sorted linked list: Simplest implementation
  - INSERT
    - O(n) time
    - Scan the list to find place and splice in the new item
  - EXTRACT-MAX
    - O(1) time
    - Take the first element

> Fast extraction but slow insertion.

Implementation of Priority Queue

- Unsorted linked list: Simplest implementation
  - INSERT
    - O(1) time
    - Put the new item at front
  - EXTRACT-MAX
    - O(n) time
    - Scan the whole list

> Fast insertion but slow extraction

Sorted linked list is better on the average
- Sorted list: on the average, scans $n/2$ elem. per insertion
- Unsorted list: always scans n elem. at each extraction

Heap Implementation of PQ

- INSERT and EXTRACT-MAX are both $O(\lg n)$
  - good compromise between fast insertion but slow extraction and vice versa
- EXTRACT-MAX: already discussed HEAP-EXTRACT-MAX

INSERT: Insertion is like that of Insertion-Sort.

Traverses $O(\lg n)$ nodes, as HEAPIFY does but makes fewer comparisons and assignments

– HEAPIFY: compares parent with both children

– HEAP-INSERT: with only one

```
HEAP-INSERT(A, key, n)
    n ← n + 1
    i ← n
    while i > 1 and A[⌊i/2⌋] < key do
        A[i] ← A[⌊i/2⌋]
        i ← ⌊i/2⌋
    A[i] ← key
```

Example: HEAP-INSERT(A, 15)

HEAP-INSERT(A, *key*, *n*)

$n \leftarrow n + 1$

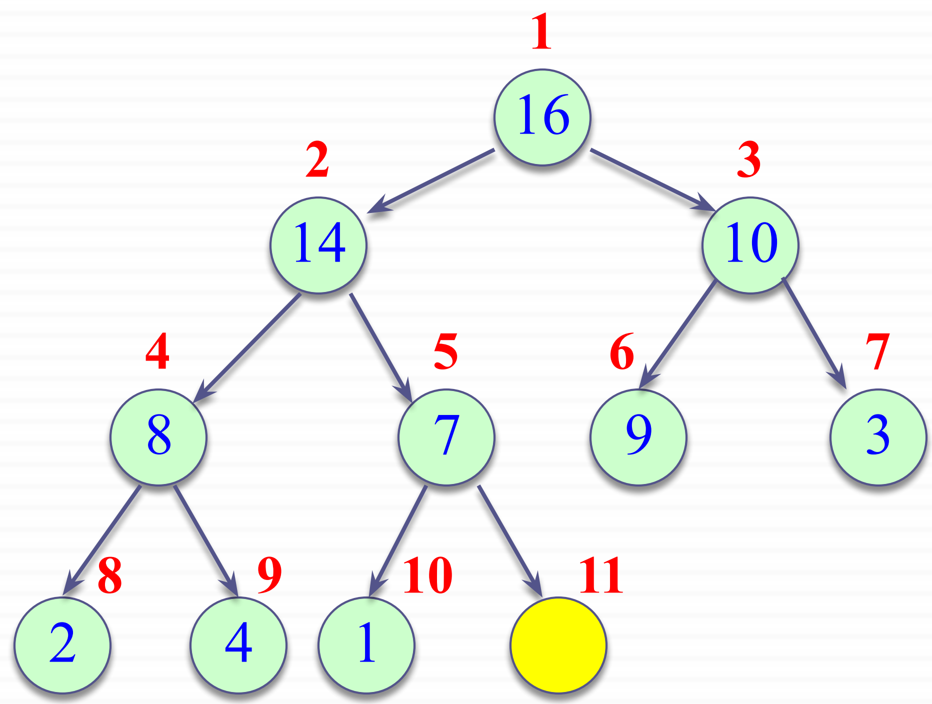
$i \leftarrow n$

while $i > 1$ **and** $A[\lfloor i/2 \rfloor] < key$ **do**

$A[i] \leftarrow A[\lfloor i/2 \rfloor]$

$i \leftarrow \lfloor i/2 \rfloor$

$A[i] \leftarrow key$



key = 15

Example: HEAP-INSERT(A, 15)

HEAP-INSERT(A, *key*, *n*)

$n \leftarrow n + 1$

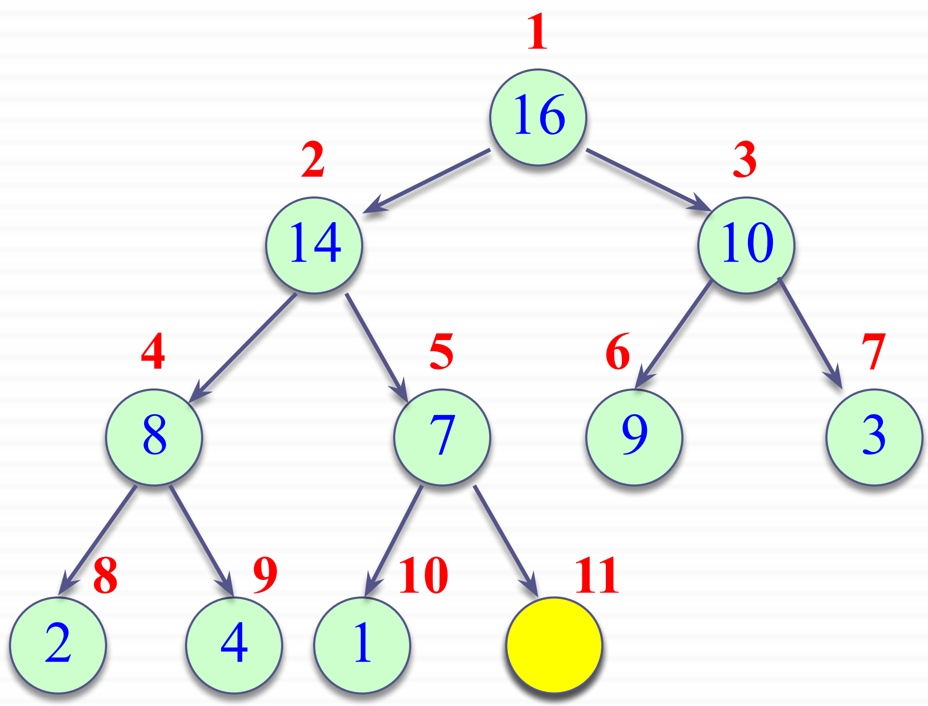
$i \leftarrow n$

while $i > 1$ **and** $A[\lfloor i/2 \rfloor] < key$ **do**

$A[i] \leftarrow A[\lfloor i/2 \rfloor]$

$i \leftarrow \lfloor i/2 \rfloor$

$A[i] \leftarrow key$



key = 15

Example: HEAP-INSERT(A, 15)

HEAP-INSERT(A, key, n)

$n \leftarrow n + 1$

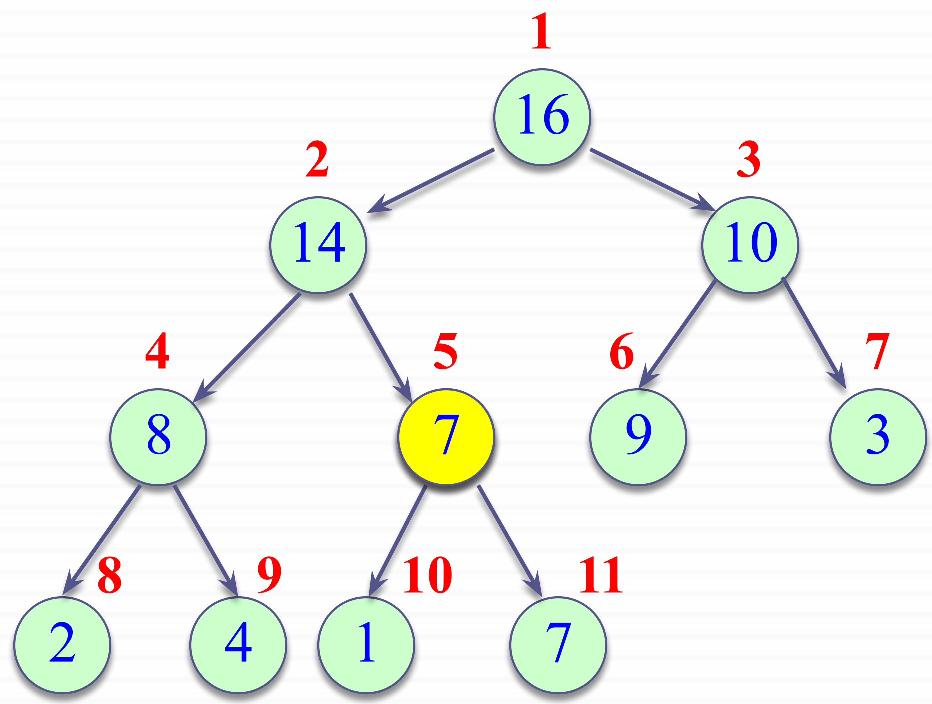
$i \leftarrow n$

while $i > 1$ **and** $A[\lfloor i/2 \rfloor] < key$ **do**

$A[i] \leftarrow A\lfloor i/2 \rfloor$

$i \leftarrow \lfloor i/2 \rfloor$

$A[i] \leftarrow key$



key = 15

Example: HEAP-INSERT(A, 15)

HEAP-INSERT(A, *key*, *n*)

$n \leftarrow n + 1$

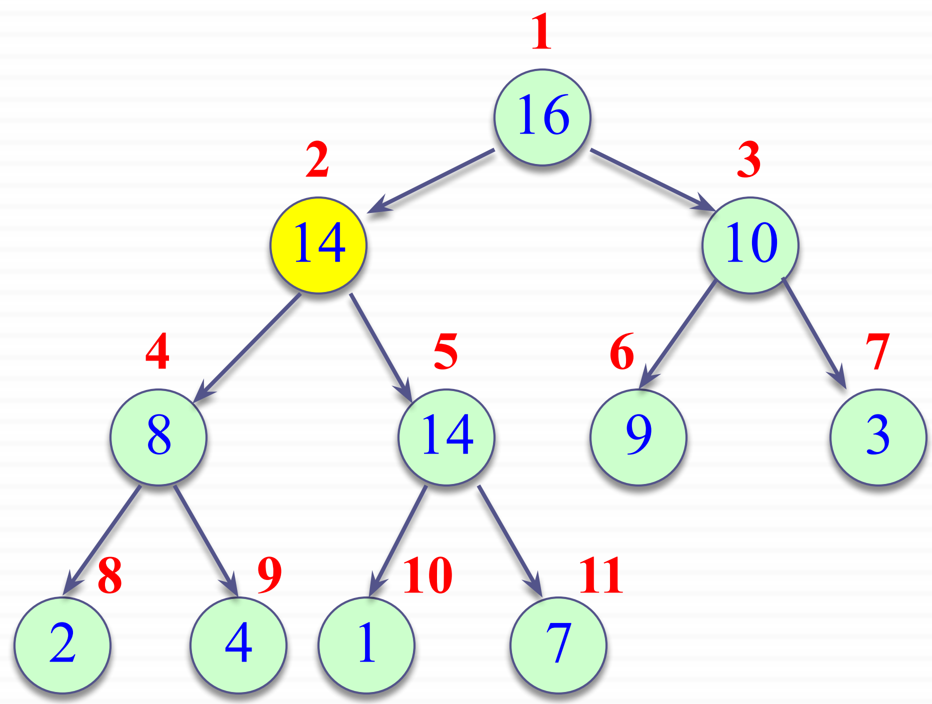
$i \leftarrow n$

while $i > 1$ **and** $A[\lfloor i/2 \rfloor] < key$ **do**

$A[i] \leftarrow A\lfloor i/2 \rfloor$

$i \leftarrow \lfloor i/2 \rfloor$

$A[i] \leftarrow key$



key = 15

Example: HEAP-INSERT(A, 15)

HEAP-INSERT(A, *key*, *n*)

$n \leftarrow n + 1$

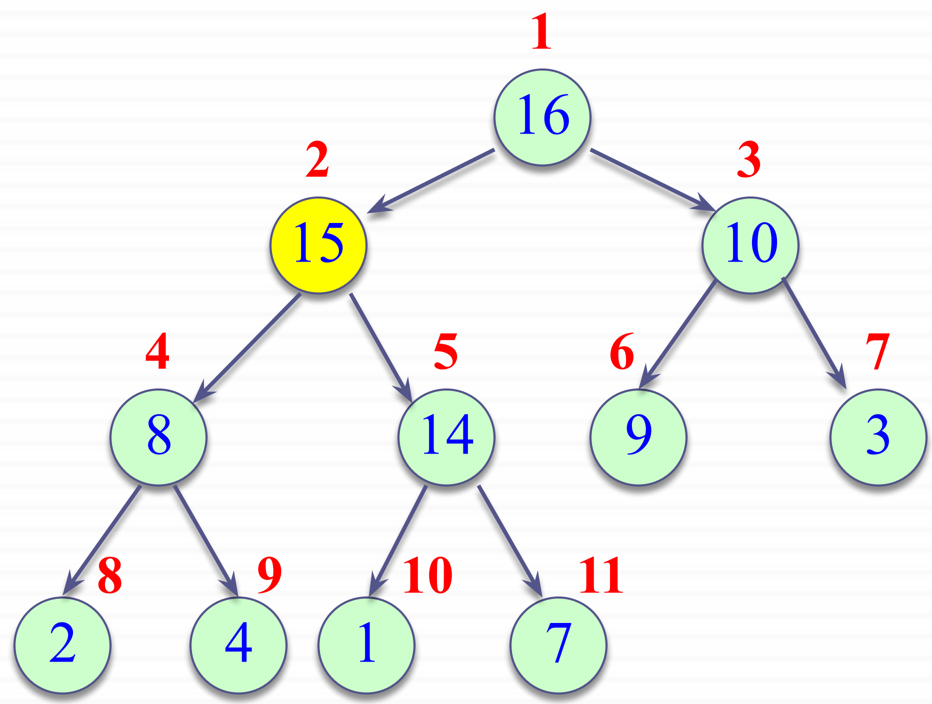
$i \leftarrow n$

while $i > 1$ **and** $A[\lfloor i/2 \rfloor] < key$ **do**

$A[i] \leftarrow A[\lfloor i/2 \rfloor]$

$i \leftarrow \lfloor i/2 \rfloor$

$A[i] \leftarrow key$



key = 15

Heap Increase Key

- Key value of i-th element of heap is increased from A[i] to *key*

HEAP-INCREASE-KEY(A, i, *key*)
    if *key* < A[i] **then**
        return error
    while $i > 1$ **and** A[$\lfloor i/2 \rfloor$] < key **do**
        A[i] $\leftarrow$ A[$\lfloor i/2 \rfloor$]
        $i \leftarrow \lfloor i/2 \rfloor$
    A[i] $\leftarrow$ *key*

Example: HEAP-INCREASE-KEY(A, 9, 15)

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

if $key < A[i]$ **then**

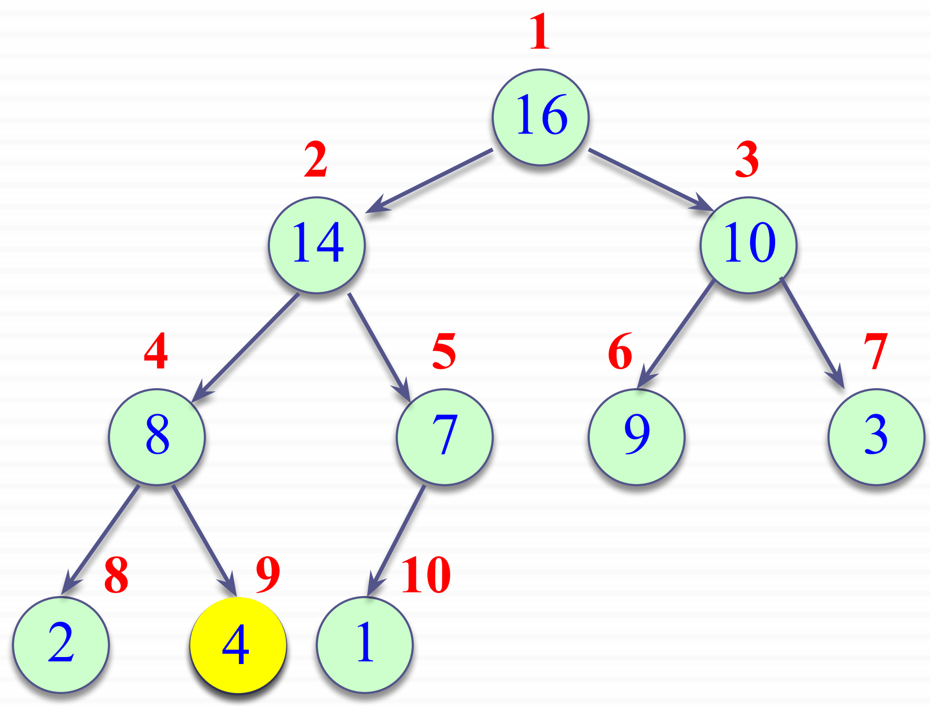
return error

while $i > 1$ **and** $A[\lfloor i/2 \rfloor] < \text{key}$ **do**

$A[i] \leftarrow A[\lfloor i/2 \rfloor]$

$i \leftarrow \lfloor i/2 \rfloor$

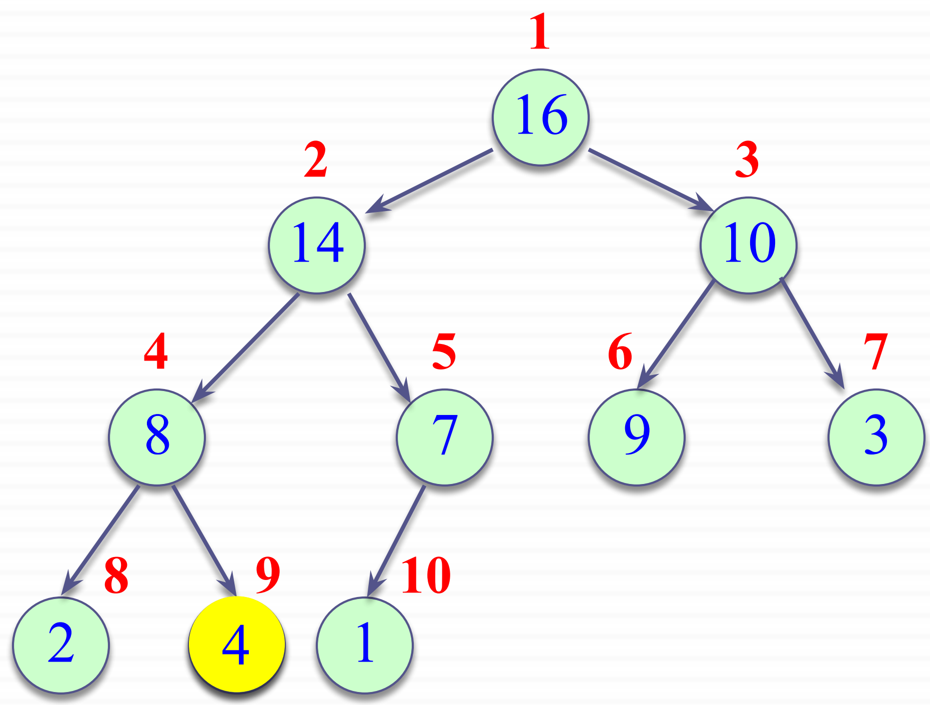
$A[i] \leftarrow key$



key = 15

Example: HEAP-INCREASE-KEY(A, 9, 15)

HEAP-INCREASE-KEY(A, *i*, *key*)
  if *key* < A[*i*] **then**
    return error
  while *i* >1 **and** A[⌊*i*/2⌋] < key
do
    A[*i*] ← A[⌊*i*/2⌋]
    i ← ⌊*i*/2⌋
  A[*i*] ← *key*



key = 15

Example: HEAP-INCREASE-KEY(A, 9, 15)

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

if $key < A[i]$ **then**

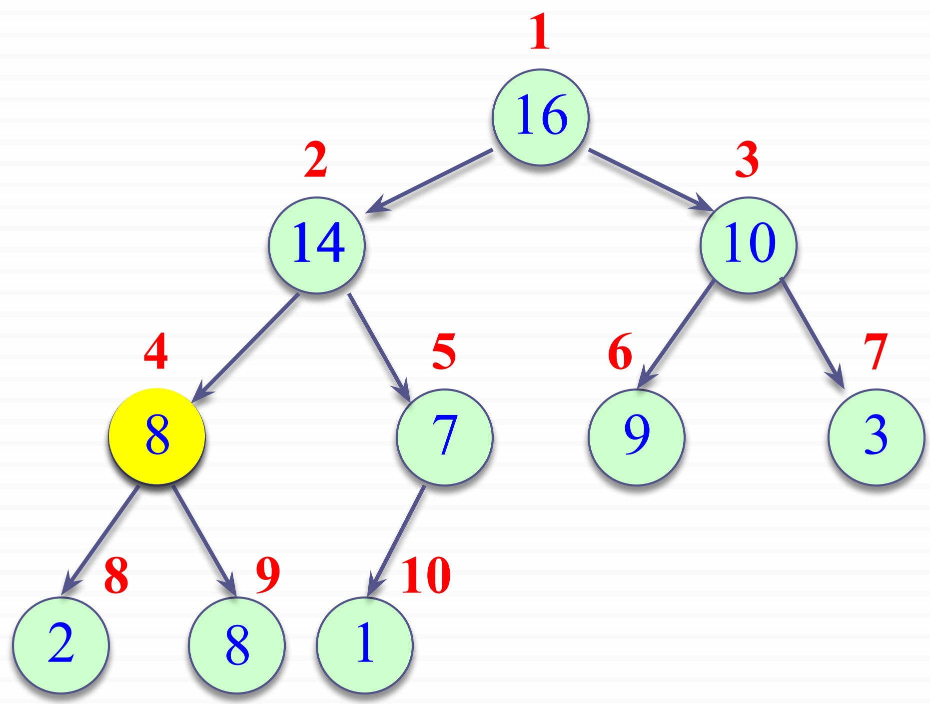
return error

while $i > 1$ **and** $A[\lfloor i/2 \rfloor] < \text{key}$ **do**

$A[i] \leftarrow A[\lfloor i/2 \rfloor]$

$i \leftarrow \lfloor i/2 \rfloor$

$A[i] \leftarrow key$



key = 15

Example: HEAP-INCREASE-KEY(A, 9, 15)

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

if *key* < A[*i*] **then**

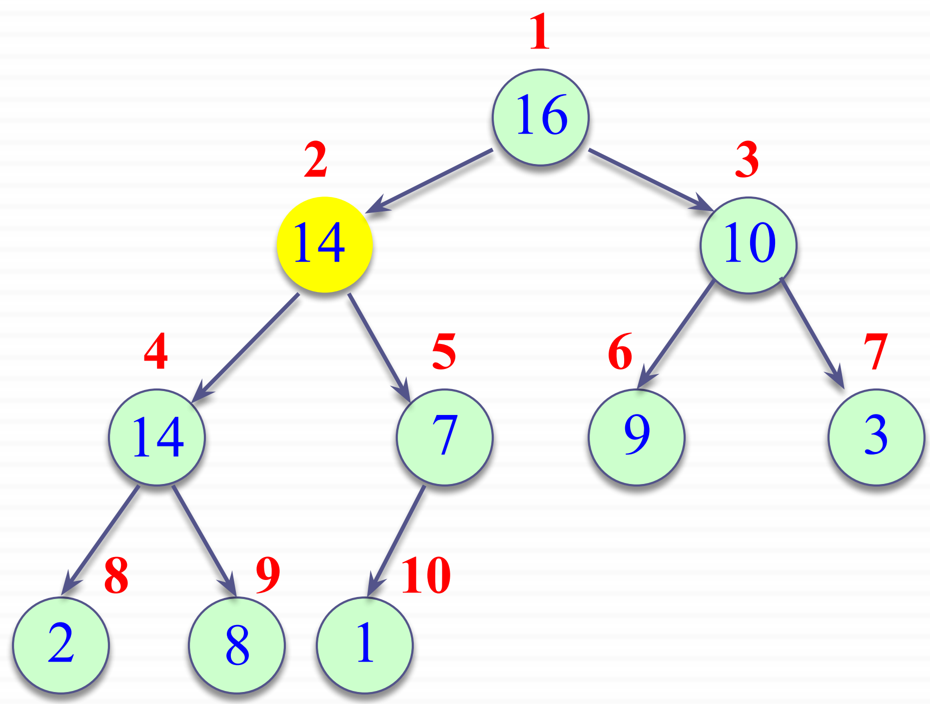
  return error

while *i* >1 **and** A[⌊*i*/2⌋] < key **do**

  A[*i*] ← A[⌊*i*/2⌋]

  i ← ⌊*i*/2⌋

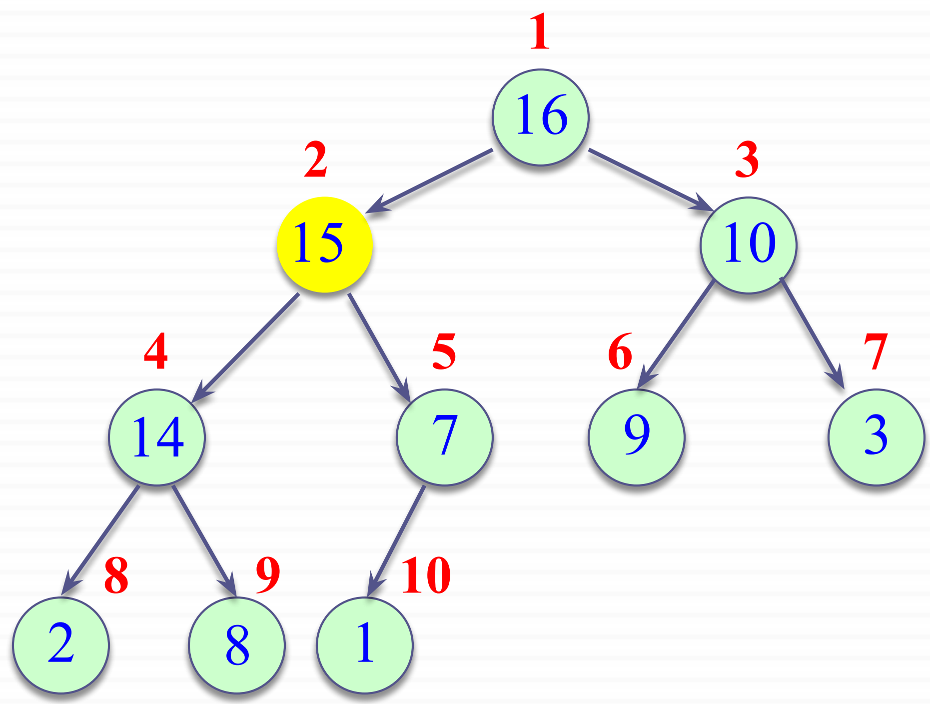
A[*i*] ← *key*



key = 15

Example: HEAP-INCREASE-KEY(A, 9, 15)

HEAP-INCREASE-KEY(A, *i*, *key*)
    if *key* < A[*i*] **then**
        return error
    while *i* >1 **and** A[⌊*i*/2⌋] < key
do
        A[*i*] ← A[⌊*i*/2⌋]
        i ← ⌊*i*/2⌋
    A[*i*] ← *key*



key = 15

Heap Implementation of PQ

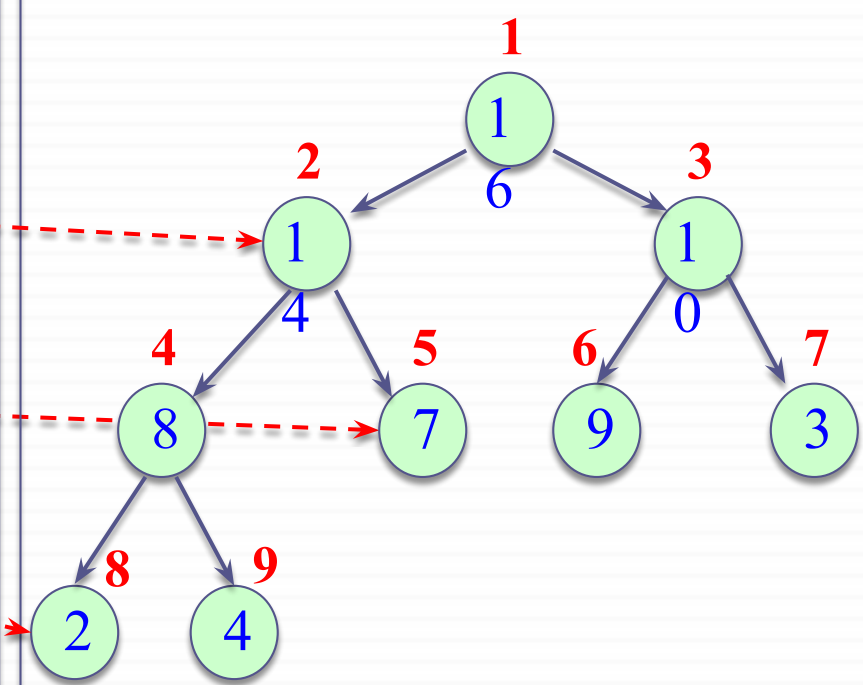
Storage in Application

	key	data	H-index
a	14		2
b	3		7
c	7		5
d	10		3
e	*		--
f	4		9
g	8		4
h	*		--
i	9		6
j	16		1
k	2		8

Heap Storage

	handle
1	j
2	a
3	d
4	g
5	c
6	i
7	b
8	k
9	f

Abstract Heap Representation



Cevdet Aykanat and Mustafa Ozdal
Computer Engineering Department, Bilkent University

Summary: Max Heap

Heapify(A, i)

Works when both child subtrees of node i are heaps

"*Floats down*" node i to satisfy the heap property

Runtime: O(lgn)

Max (A, n)

Returns the max element of the heap (no modification)

Runtime: O(1)

Extract-Max (A, n)

Returns and removes the max element of the heap

Fills the gap in A[1] with A[n], then calls Heapify(A,1)

Runtime: O(lgn)

Summary: Max Heap

Build-Heap(A, n)

Given an arbitrary array, builds a heap from scratch

Runtime: O(n)

Min(A, n)

How to return the min element in a *max-heap*?

Worst case runtime: O(n)

because ~half of the heap elements are leaf nodes

Instead, use a *min-heap* for efficient min operations

Search(A, x)

For an arbitrary x value, the worst-case runtime: O(n)

Use a sorted array instead for efficient search operations

Summary: Max Heap

Increase-Key(A, i, x)

Increase the key of node i (from A[i] to x)

“Float up” x until heap property is satisfied

Runtime: O(lgn)

Decrease-Key(A, i, x)

Decrease the key of node i (from A[i] to x)

Call Heapify(A, i)

Runtime: O(lgn)

Example Problem: Phone Operator

A phone operator answering n phones

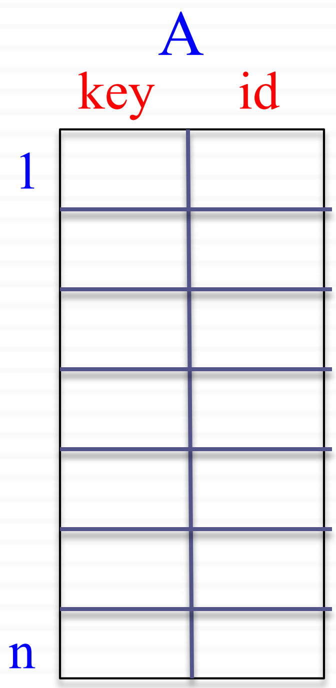
Each phone i has x_i people waiting in line for their calls to be answered.

Phone operator needs to answer the phone with the largest number of people waiting in line.

New calls come continuously, and some people hang up after waiting.

Solution

Step 1: Define the following array:



A[i]: the i^{th} element in heap

A[i].id: the index of the corresponding phone

A[i].key: # of people waiting in line for phone with index A[i].id

Cevdet Aykanat and Mustafa Ozdal
Computer Engineering Department, Bilkent University

Solution

Step 2: Build-Max-Heap (A, n)

Execution:

When the operator wants to answer a phone:

id = A[1].id

Decrease-Key(A, 1, A[1].key-1)

answer phone with index id

When a new call comes in to phone i:

Increase-Key(A, i, A[i].key+1)

When a call drops from phone i:

Decrease-Key(A, i, A[i].key-1)