CSE214 – Analysis of Algorithms

PhD Furkan Gözükara, Toros University

https://github.com/FurkanGozukara/Analysis-of-Algorithms-2019

Lecture 7

Heapsort

Based on Ching-Chi Lin's Lecture Notes - National Taiwan Ocean University

Based on Cevdet Aykanat's and Mustafa Ozdal's Lecture Notes - Bilkent

Outline

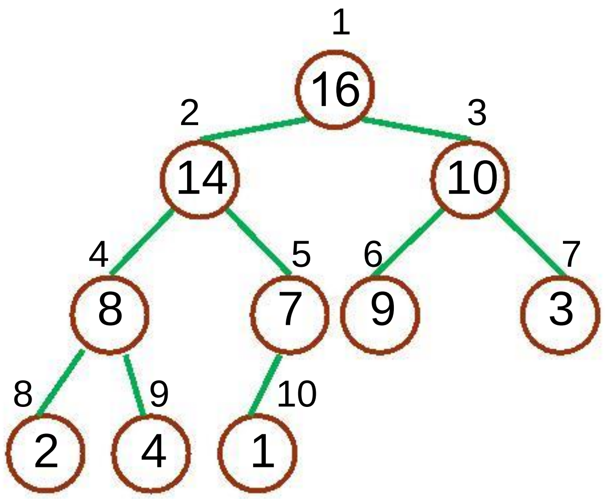
- **Heaps**
- Maintaining the heap property
- Building a heap
- The heapsort algorithm
- Priority queues

The purpose of this chapter

- In this chapter, we introduce the **heapsort** algorithm.
  - with worst case running time $O(n\lg n)$
  - an **in-place** sorting algorithm: only a constant number of array elements are stored outside the input array at any time.
  - thus, require at most $O(1)$ additional memory
- We also introduce the **heap** data structure.
  - an useful data structure for heapsort
  - makes an efficient priority queue

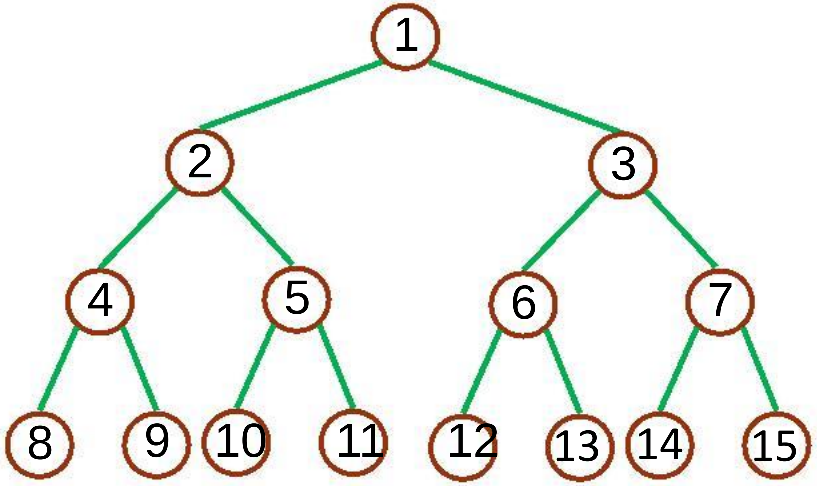
Heaps

- The (**Binary**) **heap** data structure is an **array** object that can be viewed as a nearly complete binary tree.
  - A binary tree with *n* nodes and depth *k* is **complete** if its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.

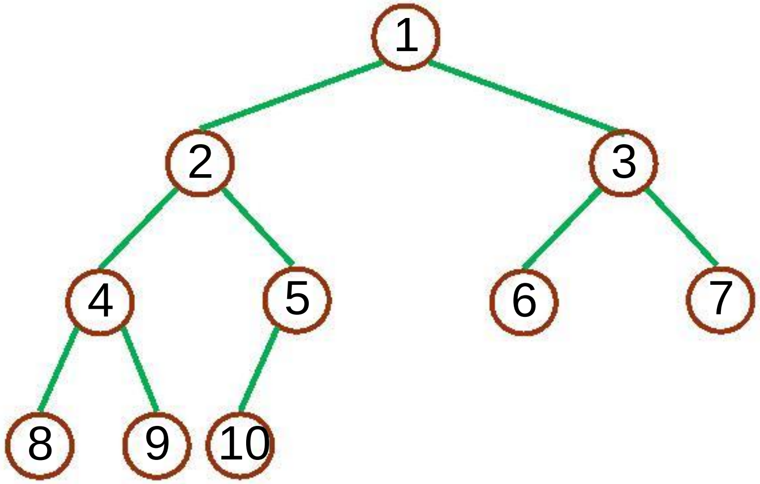


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Binary tree representations



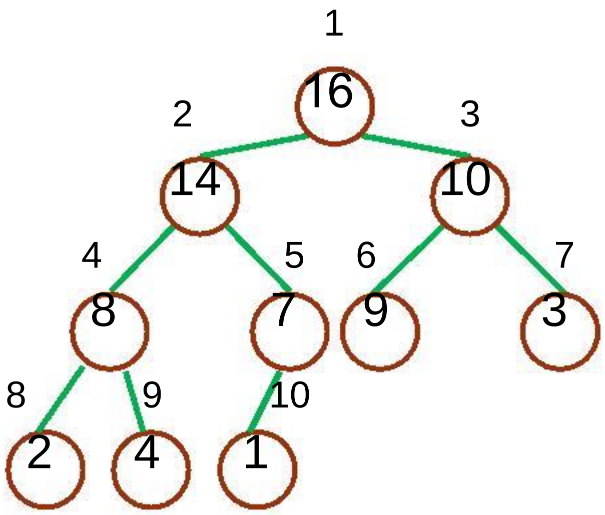
A full binary tree of height 3.



A complete binary tree with 10 nodes and height 3.

Attributes of a Heap

- An array A that presents a heap with two attributes:
  - **length[A]:** the number of elements in the array.
  - **heap-size[A]:** the number of elements in the heap stored with array A.
  - **length[A] ≥ heap-size[A]**

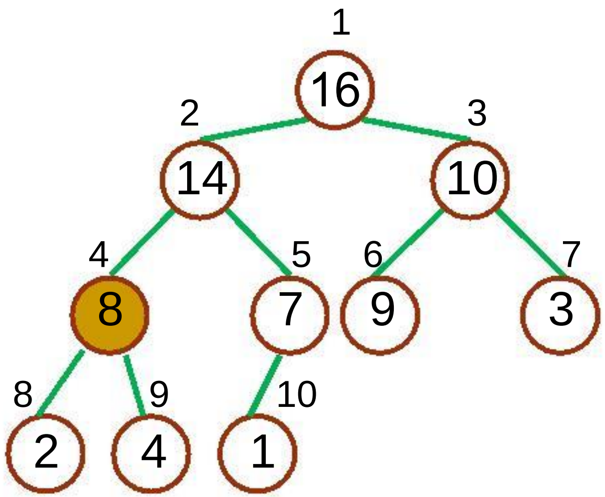


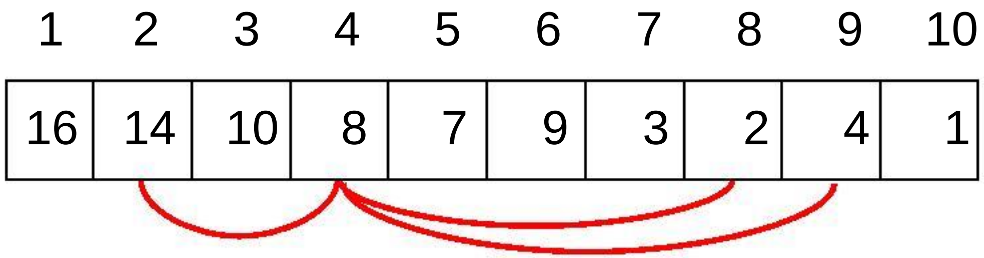
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

length[A]=heapsize[A]=10

Basic procedures 1/2

- If a complete binary tree with *n* nodes is represented sequentially, then for any node with index *i*, 1 □ *i* □ n, we have
  - A[1] is the **root** of the tree
  - the parent **PARENT(*i*)** is at ⌊*i*/2⌋ if *i* □ 1
  - the left child **LEFT(*i*)** is at 2*i*
  - the right child **RIGHT(*i*)** is at 2*i*+1





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

Basic procedures 2/2

- The **LEFT** procedure can compute 2*i* in one instruction by simply shifting the binary representation of *i* left one bit position.
- Similarly, the **RIGHT** procedure can quickly compute 2*i*+1 by shifting the binary representation of *i* left one bit position and adding in a 1 as the low-order bit.
- The **PARENT** procedure can compute $\lfloor i/2 \rfloor$ by shifting *i* right one bit position.

Heap properties

- There are two kind of binary heaps: max-heaps and min-heaps.
  - In a **max-heap**, the **max-heap property** is that for every node *i* other than the root,

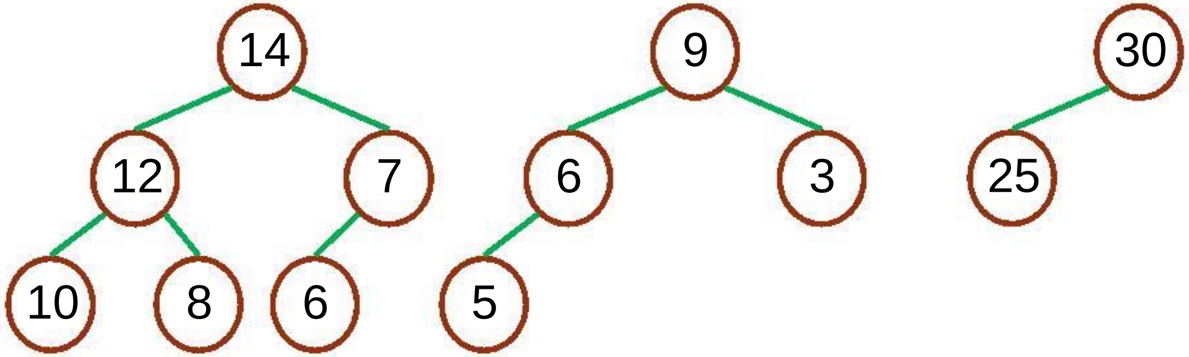
$$A[\text{P}{\small\text{ARENT}}(i)] \geq A[i] .$$

    - the largest element in a max-heap is stored at the root
    - the subtree rooted at a node contains values no larger than that contained at the node itself
  - In a **min-heap**, the **min-heap property** is that for every node *i* other than the root,

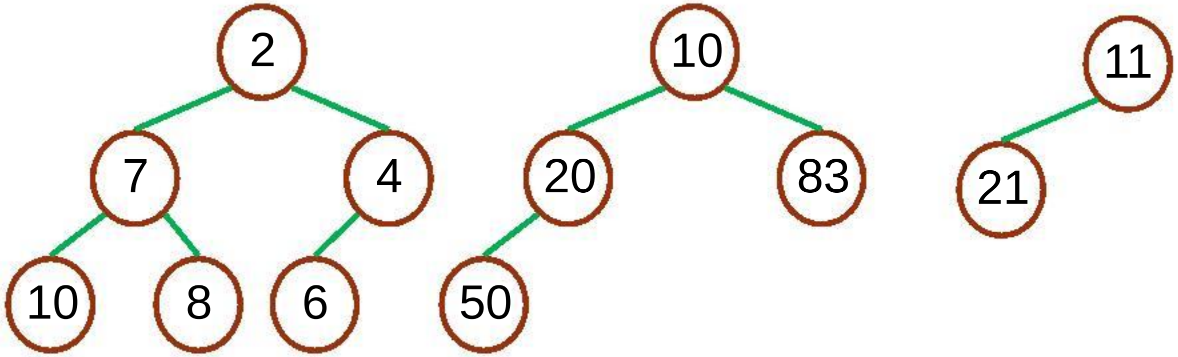
$$A[\text{P}{\small\text{ARENT}}(i)] \leq A[i] .$$

    - the smallest element in a min-heap is at the root
    - the subtree rooted at a node contains values no smaller than that contained at the node itself

Max and min heaps



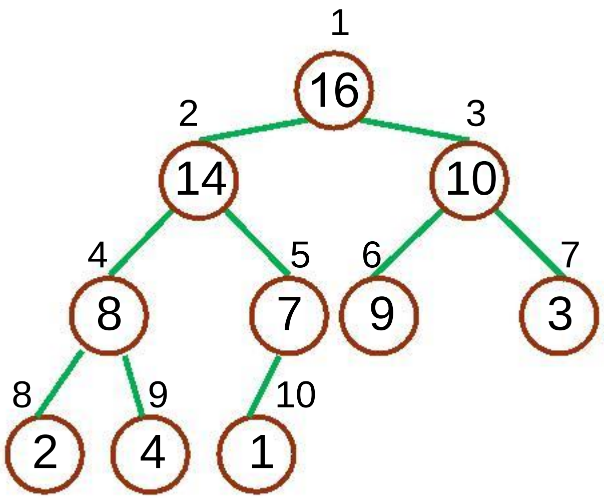
Max Heaps



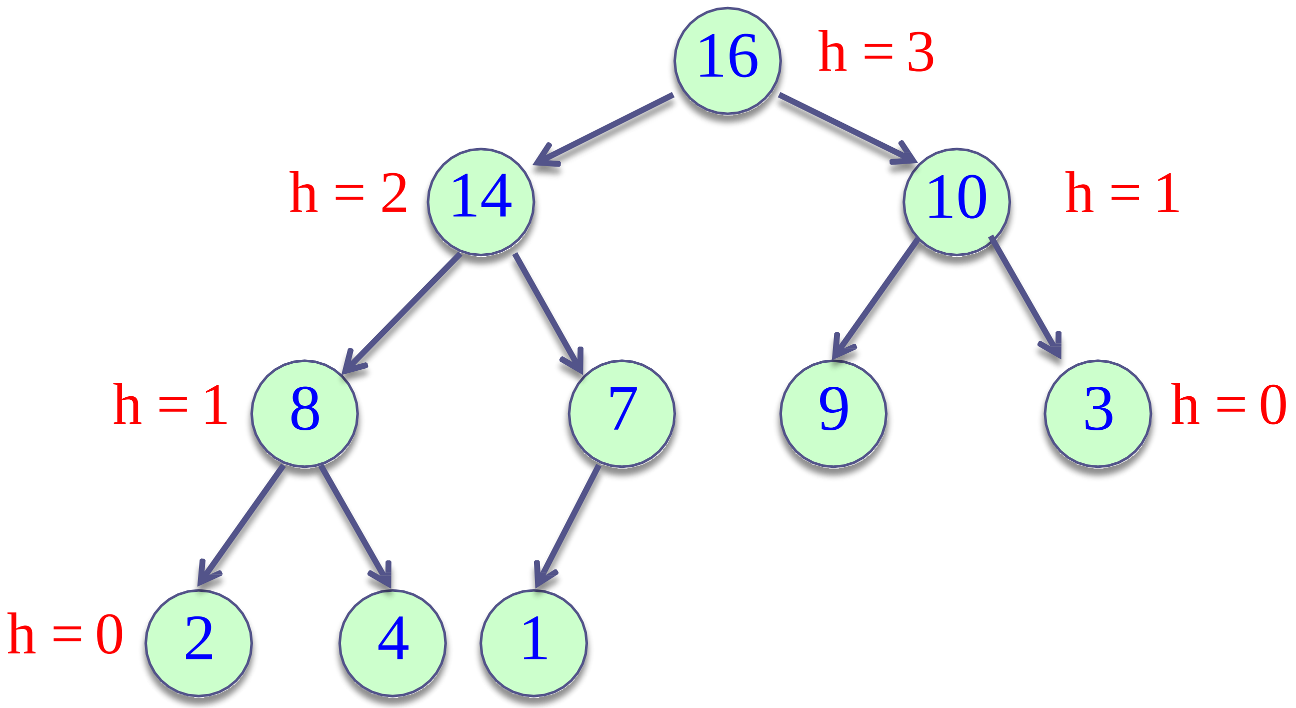
Min Heaps

The height of a heap

- The **height** of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf, and the height of the heap to be the height of the root, that is □ (lg*n*).
- For example:
  - the height of node 2 is 2
  - the height of the heap is 3



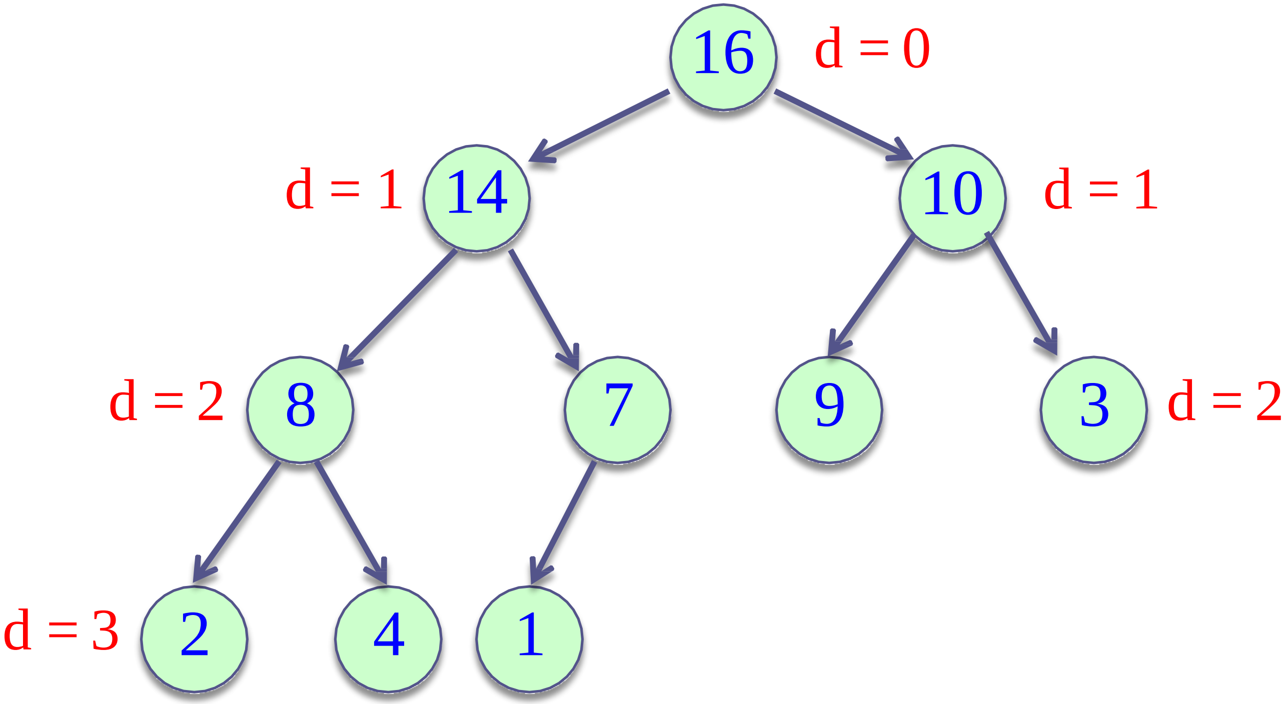
Heap Data Structures



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

The remainder of this chapter

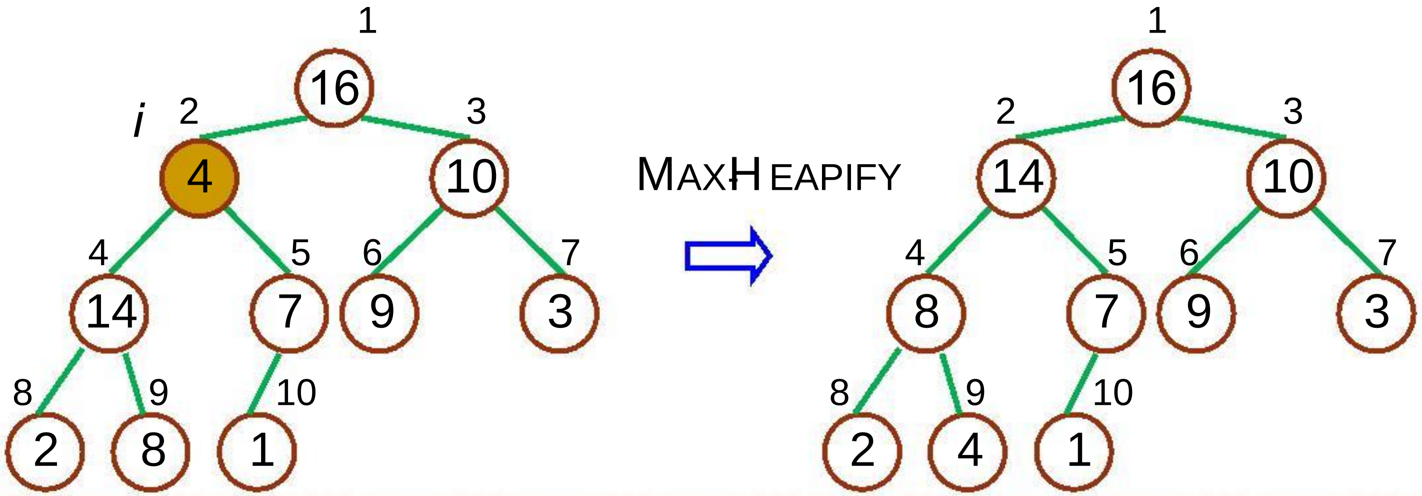
- We shall presents some basic procedures in the remainder of this chapter.
  - The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
  - The **BUILD-MAX-HEAP** procedure, which runs in $O(n)$ time, produces a max-heap from an unordered input array.
  - The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
  - The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to be used as a priority queue.

Outline

- Heaps
- **Maintaining the heap property**
- Building a heap
- The heapsort algorithm
- Priority queues

The MAX-HEAPIFY procedure1/2

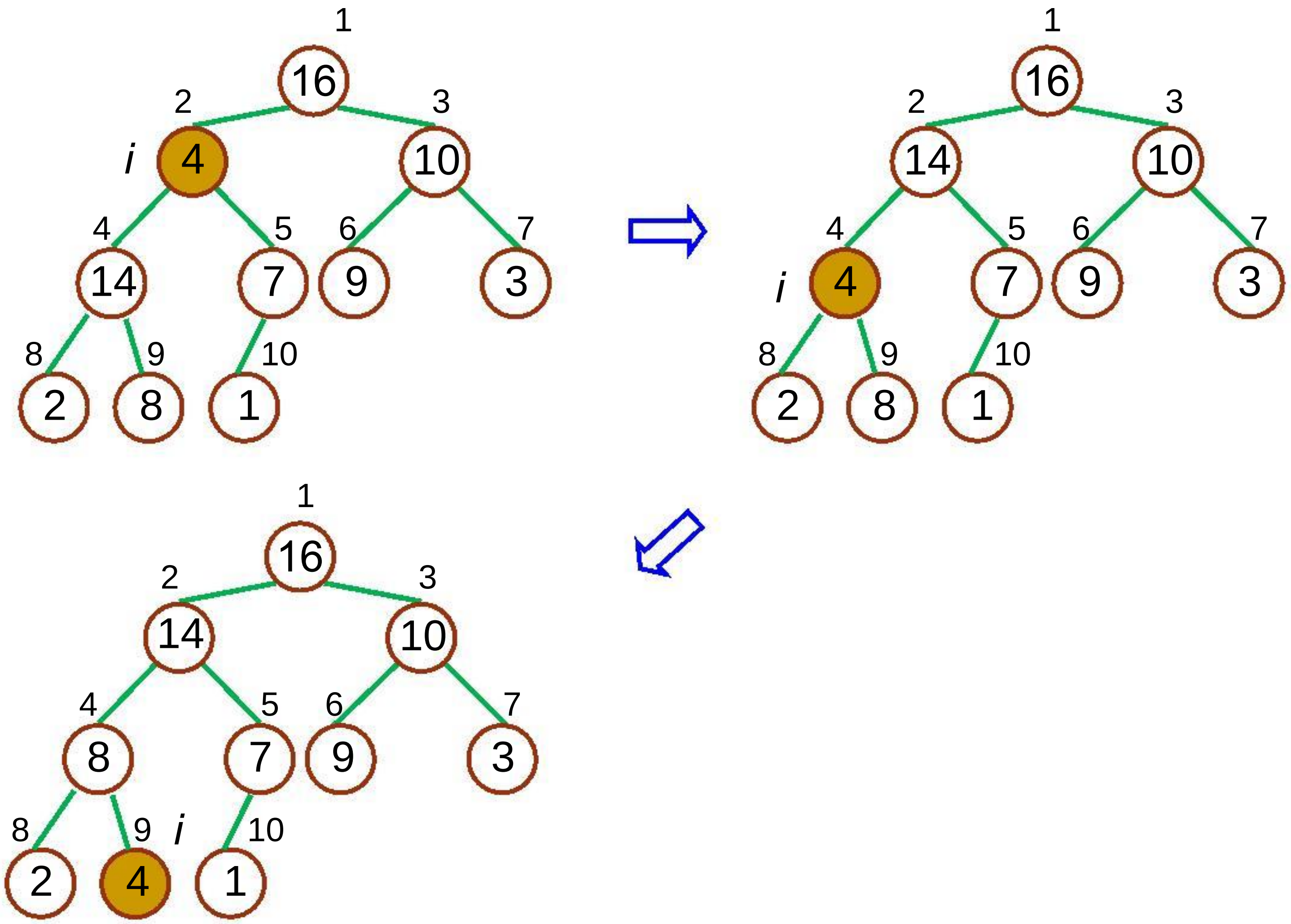
- MAX-HEAPIFY is an important subroutine for manipulating max heaps.
  - **Input**: an array A and an index *i*
  - **Output**: the subtree rooted at index *i* becomes a max heap
  - **Assume**: the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps, but A[i] may be smaller than its children
  - **Method**: let the value at A[i] "float down" in the max-heap



The MAX-HEAPIFY procedure2/2

```
MAX-HEAPIFY(A, i)
    $l \gets$ LEFT(i)
    $r \gets$ RIGHT(i)
    if $l \leq$ heap-size[A] and $A[l] > A[i]$
        then largest $\gets l$
           else largest $\gets i$
    if $r \leq$ heap-size[A] and $a[r] > A[largest]$
        then largest $\gets r$
    if largest $\neq i$
           then exchange $A[i] \leftrightarrow A[largest]$
                MAX-HEAPIFY (A, largest)
```

An example of MAX-HEAPIFY procedure



The time complexity

- It takes $\Theta(1)$ time to fix up the relationships among the elements A[*i*], A[LEFT(*i*)], and A[RIGHT(*i*)].
- Also, we need to run MAX-HEAPIFY on a subtree rooted at one of the children of node *i*.
- The children's subtrees each have size at most 2*n*/3
  - worst case occurs when the last row of the tree is exactly half full
- The running time of MAX-HEAPIFY is

$$
\begin{aligned}
T(n) &= T(2n/3) + \Theta(1) \\
&= O(\lg n)
\end{aligned}
$$

  - solve it by case 2 of the master theorem
- Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height *h* as *O*(*h*).

Master Theorem: Reminder

T(n) = aT(n/b) + f(n)

Case 1: $\frac{n^{\log_b a}}{f(n)} = \Omega(n^{\varepsilon})$ $\Rightarrow$ $T(n) = \Theta(n^{\log_b a})$

Case 2: $\frac{f(n)}{n^{\log_b a}} = \Theta(\lg^k n)$ $\Rightarrow$ $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

Case 3: $\frac{f(n)}{n^{\log_b a}} = \Omega(n^{\varepsilon})$

and $a\, f(n/b) \leq c\, f(n)$ for $c < 1$ $\Rightarrow$ $T(n) = \Theta(f(n))$

Outline

- Heaps
- Maintaining the heap property
- **Building a heap**
- The heapsort algorithm
- Priority queues

Building a Heap

- We can use the MAX-HEAPIFY procedure to convert an array A=[1..n] into a max-heap in a **bottom-up** manner.
- The elements in the subarray A[(⌊*n*/2⌋+1)…*n*] are all **leaves** of the tree, and so each is a 1-element heap.
- The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

```
BUILD-MAX-HEAP(A)
    heap-size[A] □ length[A]
    for i □ ⌊length[A]/2⌋ downto 1
        do MAX-HEAPIFY(A,i)
```

Time Complexity 1/2

- **Analysis 1**:
  - Each call to MAX-HEAPIFY costs $O(\lg n)$, and there are $O(n)$ such calls.
  - Thus, the running time is $O(n \lg n)$. This upper bound, through correct, is **not asymptotically tight**.
- **Analysis 2**:
  - For an n-element heap, height is $\lfloor \lg n \rfloor$ and at most $\lceil n / 2^{h+1} \rceil$ nodes of any height h.
  - The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$.
- The total cost is $$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right).$$

Time Complexity 2/2

- The last summation yields

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

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

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

- We can build a max-heap from an unordered array in **linear time**.

Outline

- Heaps
- Maintaining the heap property
- Building a heap
- **The heapsort algorithm**
- Priority queues

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

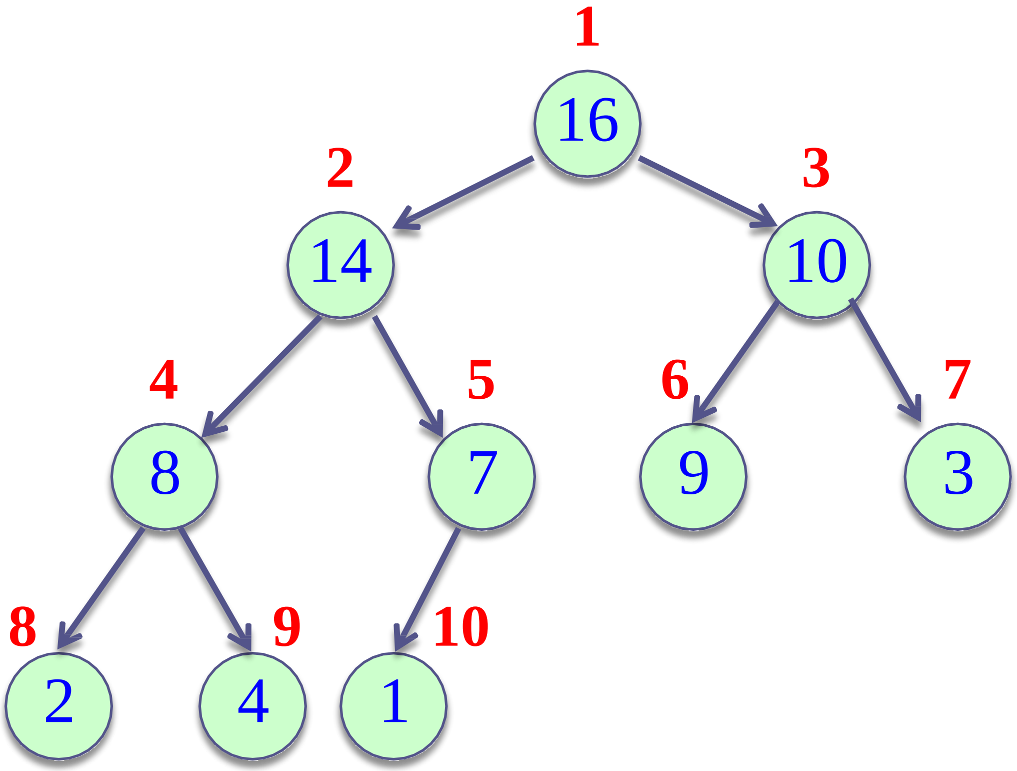
Make $A[1 \ldots n-1]$ a heap by calling HEAPIFY(A, 1, $n-1$)

(5) $n \leftarrow n - 1$

(6) Repeat steps 2–4 until $n = 2$

Heapsort Algorithm

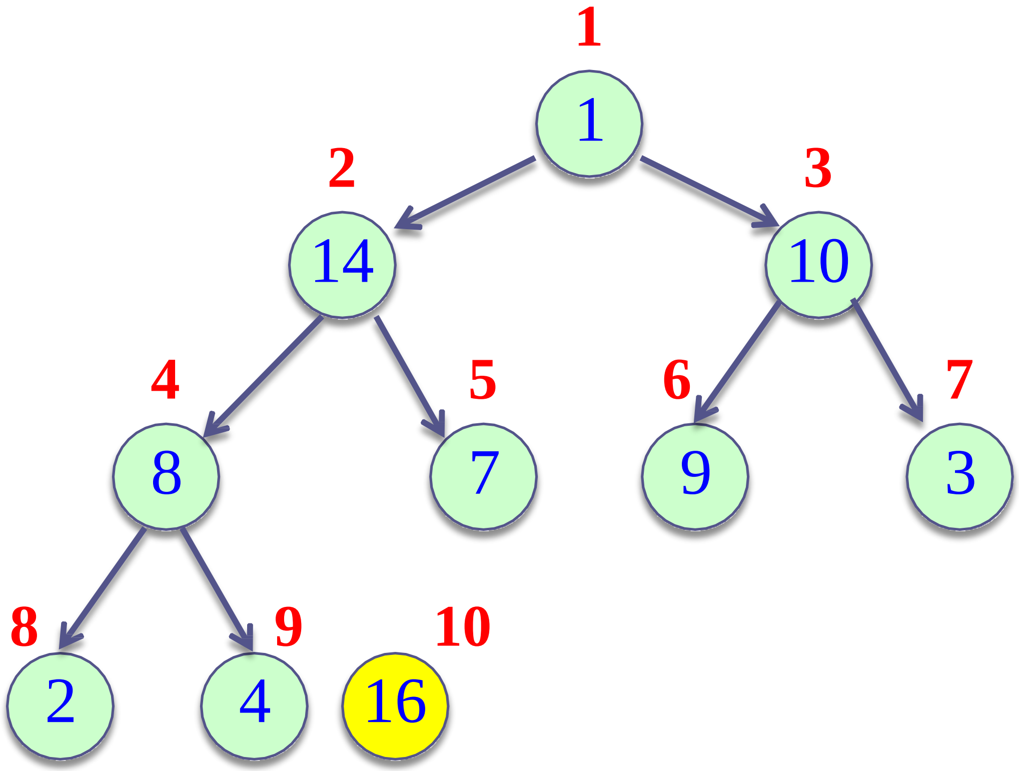
```
HEAPSORT(A, n)
    BUILD-MAX-HEAP(A, n)
    for i ← n downto 2 do
        exchange A[1] ↔A[i]
        MAX-HEAPIFY(A, 1, i–1)
```



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

Heapsort Algorithm

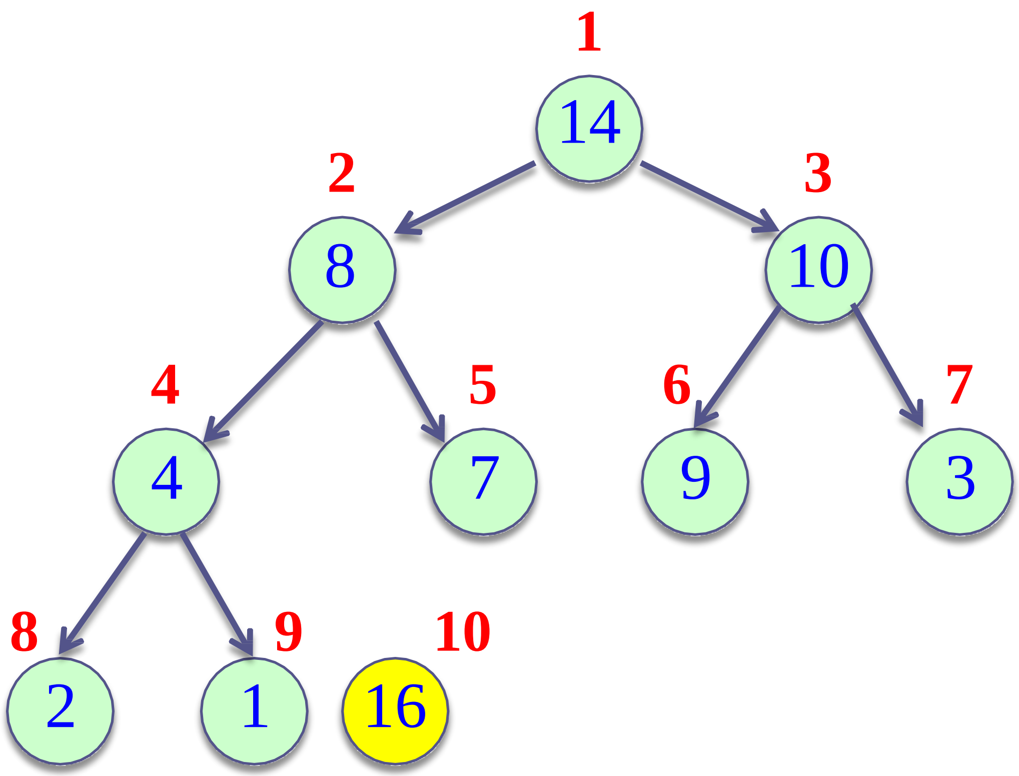
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ downto 2 do
        exchange $A[1] \leftrightarrow A[i]$
        HEAPIFY(A, 1, $i-1$)
```



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

Heapsort Algorithm

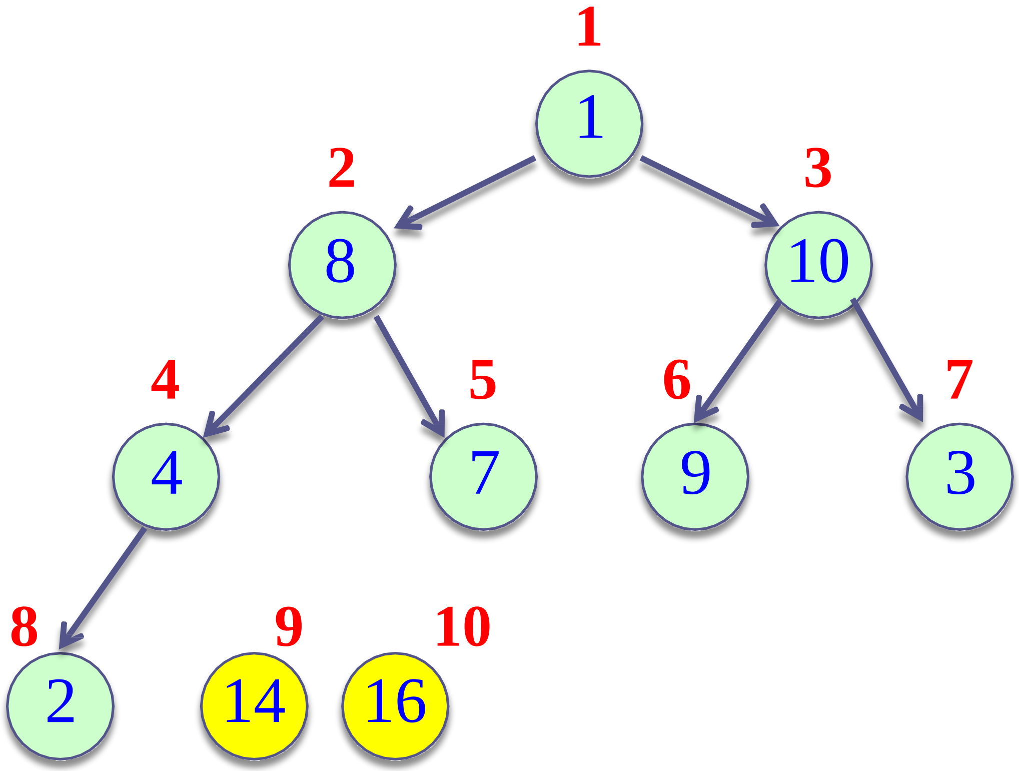
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for i ← n downto 2 do
        exchange A[1] ↔ A[i]
        HEAPIFY(A, 1, i–1)
```



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

Heapsort Algorithm

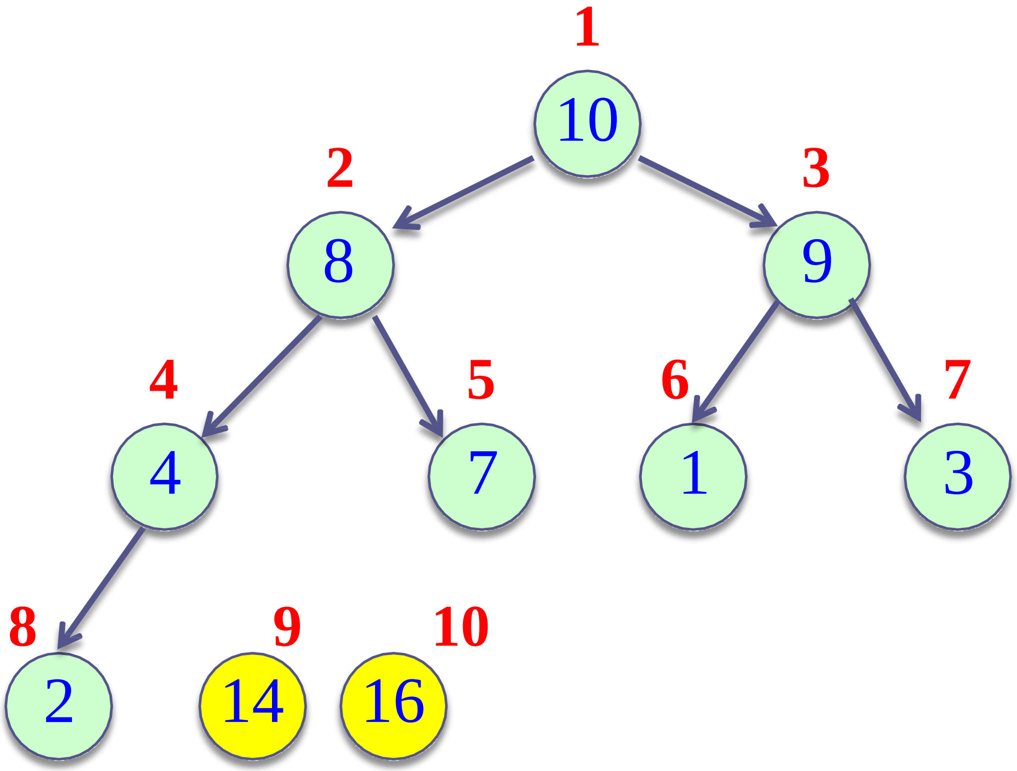
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for i ← n downto 2 do
        exchange A[1] ↔ A[i]
        HEAPIFY(A, 1, i–1)
```



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

Heapsort Algorithm

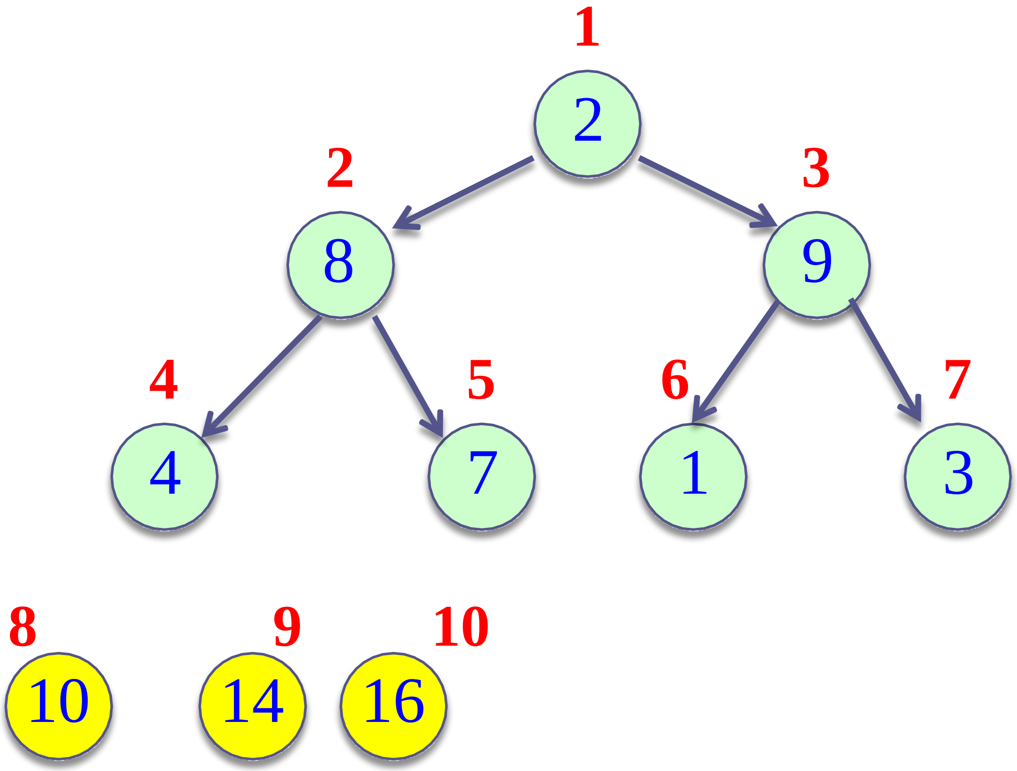
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ downto 2 do
        exchange A[1] $\leftrightarrow$ A[$i$]
        HEAPIFY(A, 1, $i$–1)
```



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

Heapsort Algorithm

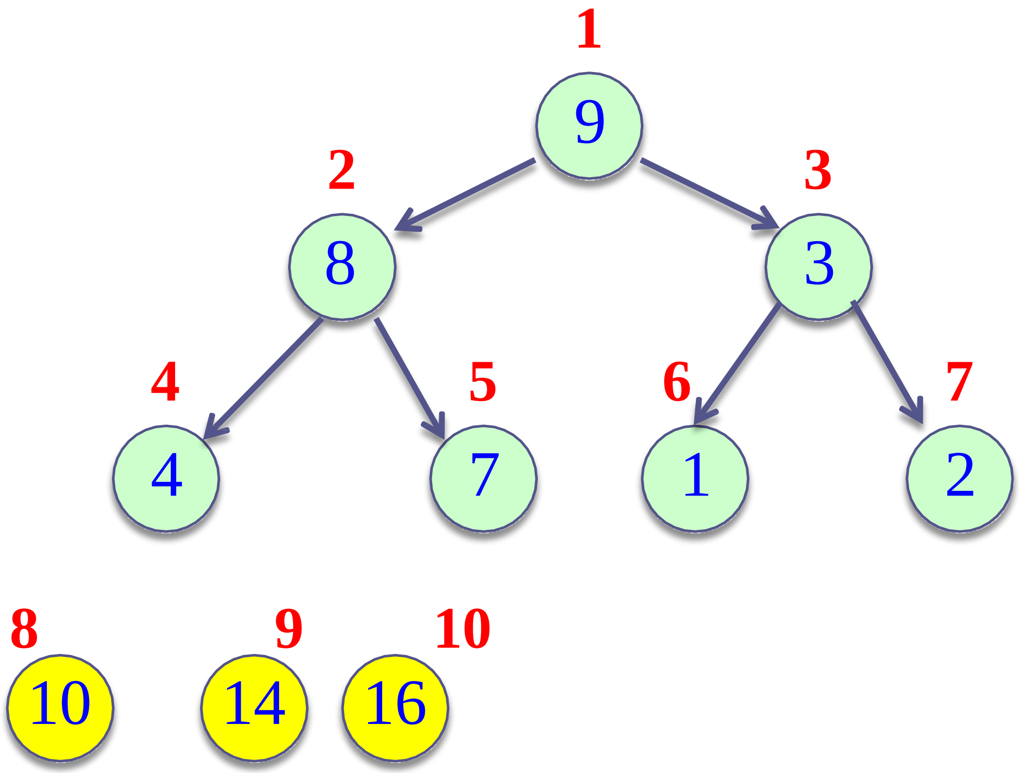
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ downto 2 do
        exchange A[1] ↔ A[i]
        HEAPIFY(A, 1, i–1)
```



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

Heapsort Algorithm

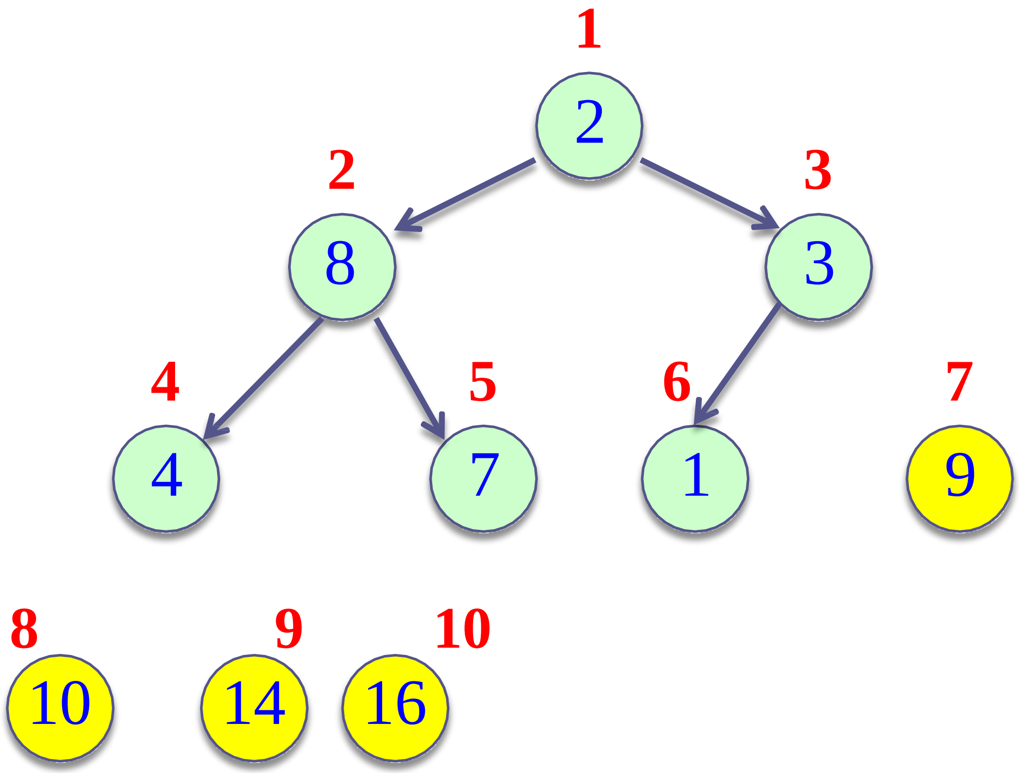
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i$ ← $n$ downto 2 do
        exchange A[1] ↔ A[$i$]
        HEAPIFY(A, 1, $i$–1)
```



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

Heapsort Algorithm

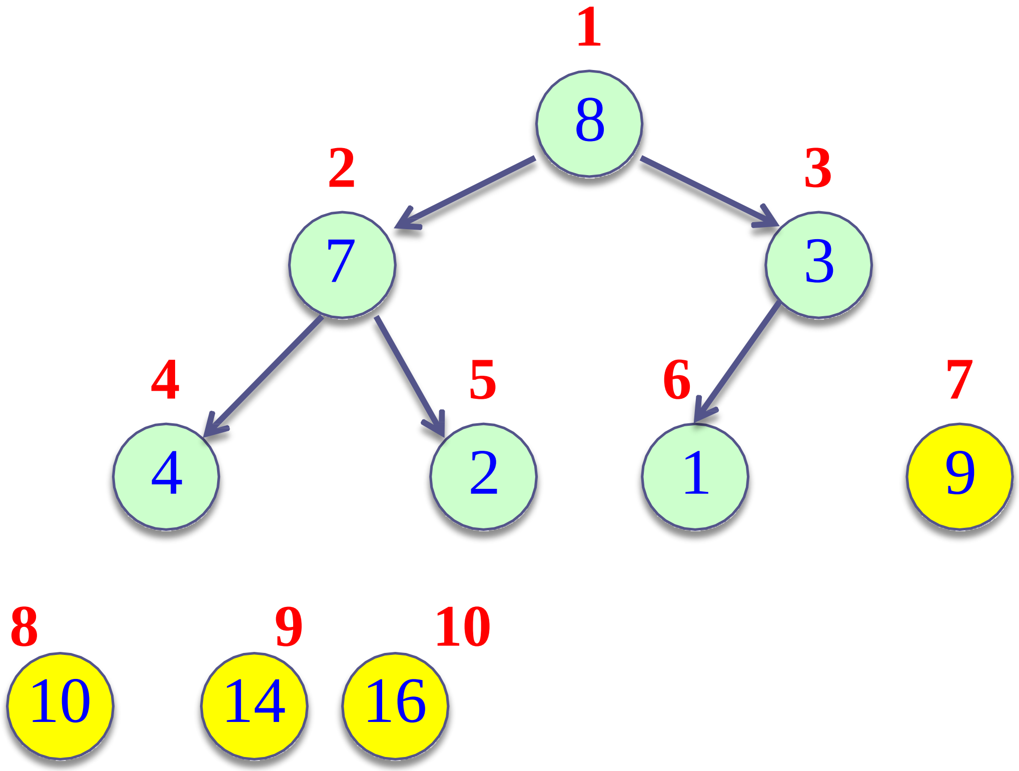
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ downto 2 do
        exchange A[1] ↔ A[i]
        HEAPIFY(A, 1, i–1)
```



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

Heapsort Algorithm

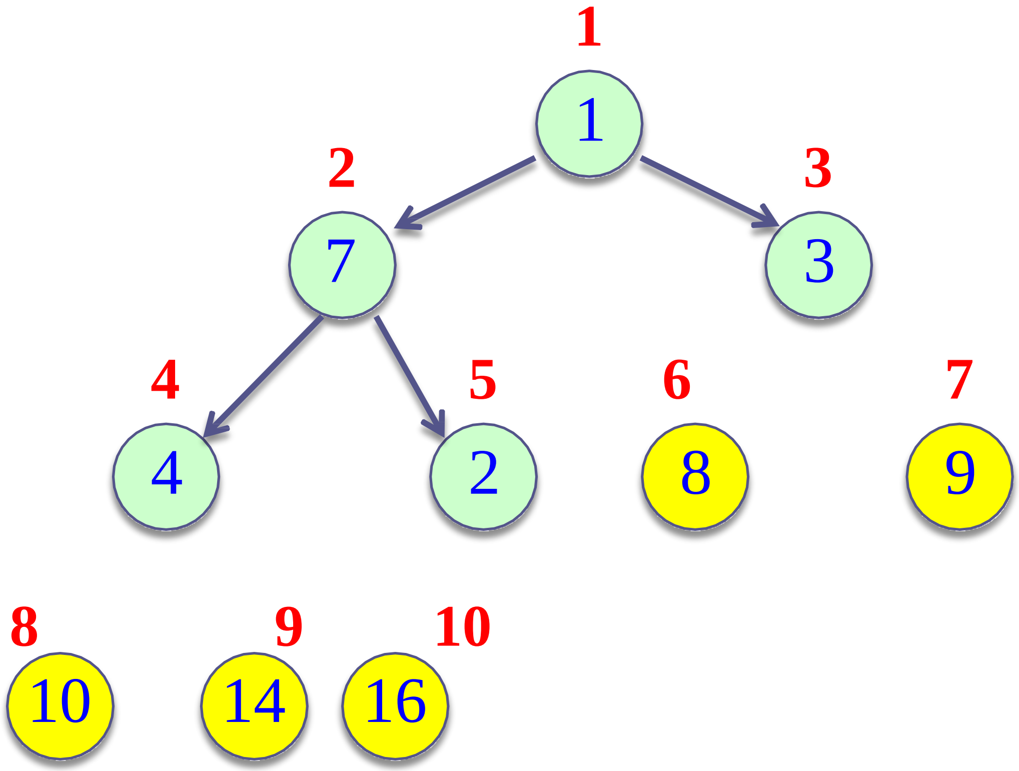
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for i ← n downto 2 do
        exchange A[1] ↔ A[i]
        HEAPIFY(A, 1, i–1)
```



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

Heapsort Algorithm

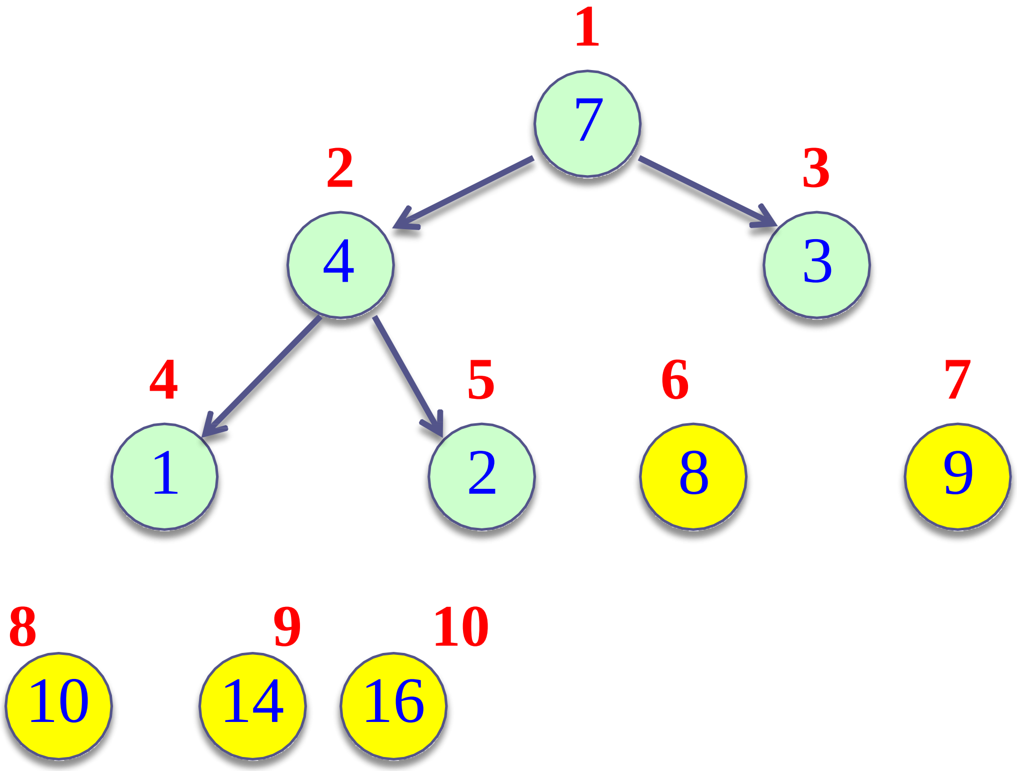
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for i ← n downto 2 do
        exchange A[1] ↔ A[i]
        HEAPIFY(A, 1, i–1)
```



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

Heapsort Algorithm

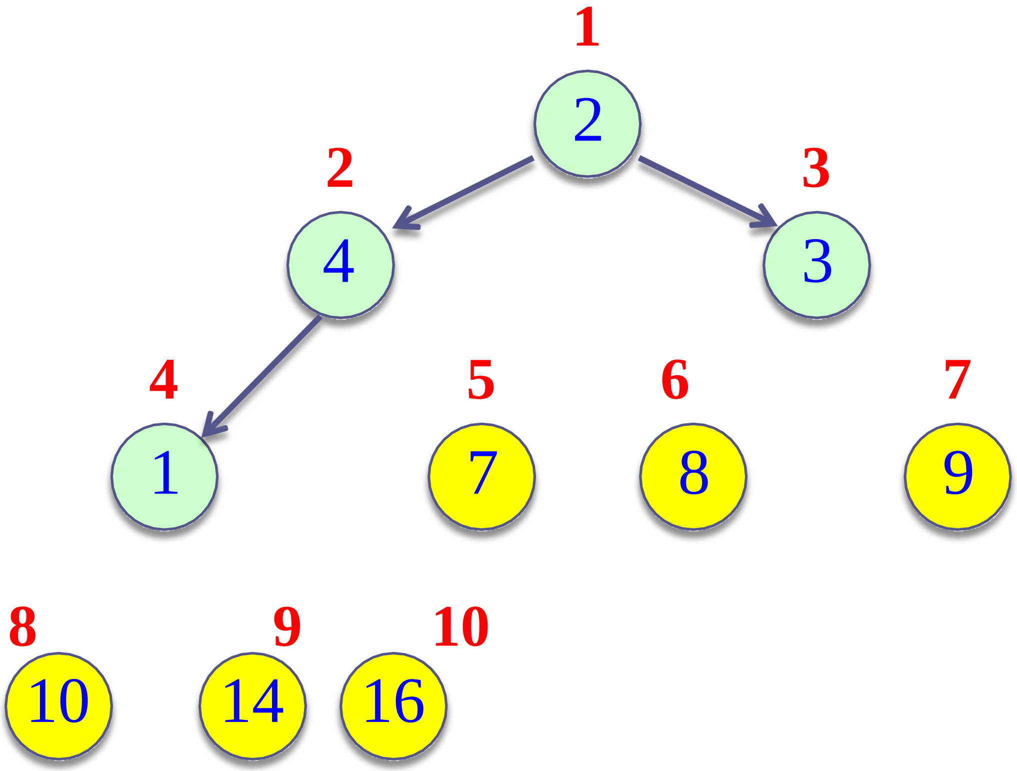
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for i ← n downto 2 do
        exchange A[1] ↔ A[i]
        HEAPIFY(A, 1, i–1)
```



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

Heapsort Algorithm

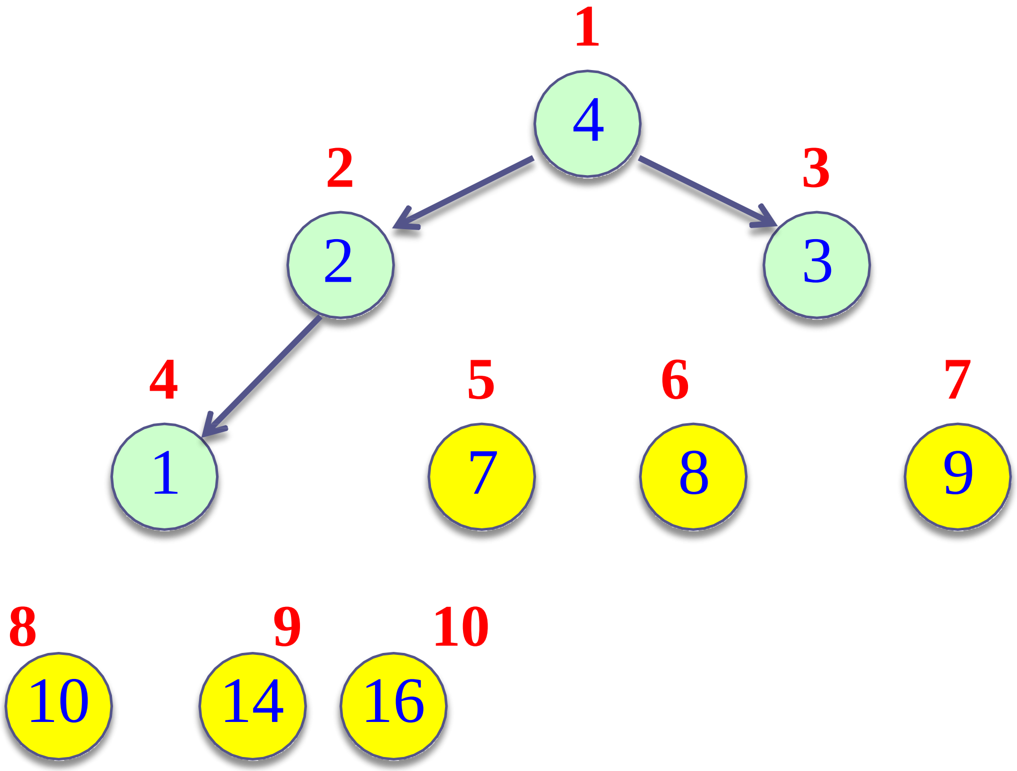
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ downto 2 do
        exchange A[1] ↔ A[$i$]
        HEAPIFY(A, 1, $i$–1)
```



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

Heapsort Algorithm

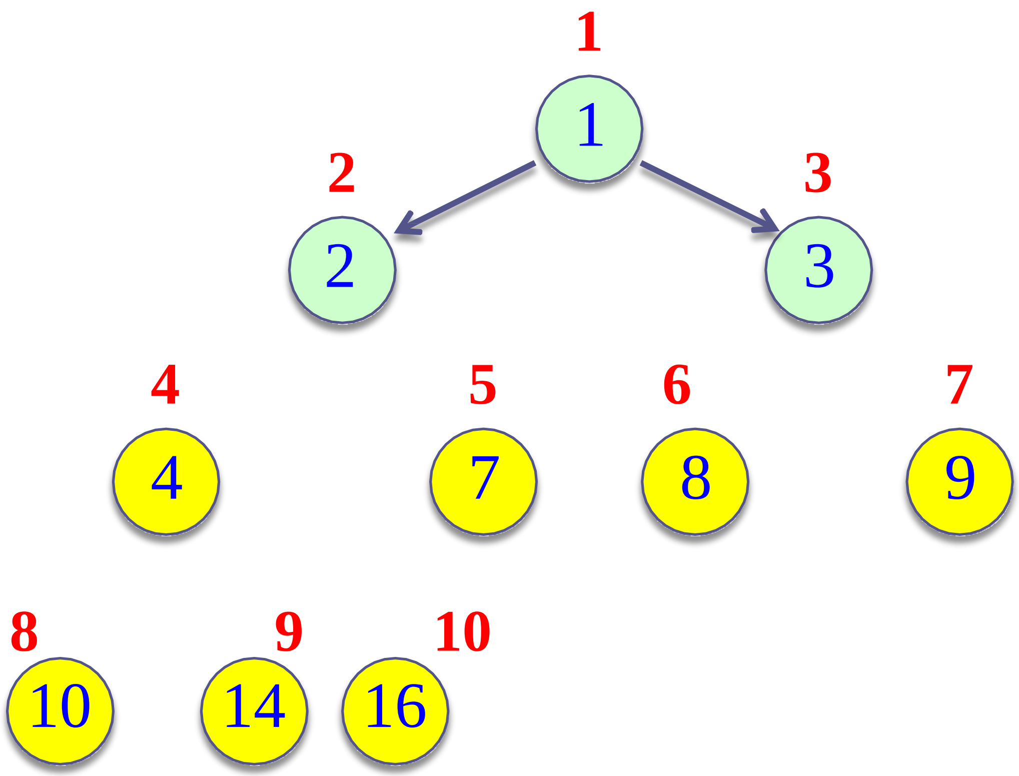
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for i ← n downto 2 do
        exchange A[1] ↔ A[i]
        HEAPIFY(A, 1, i–1)
```



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

Heapsort Algorithm

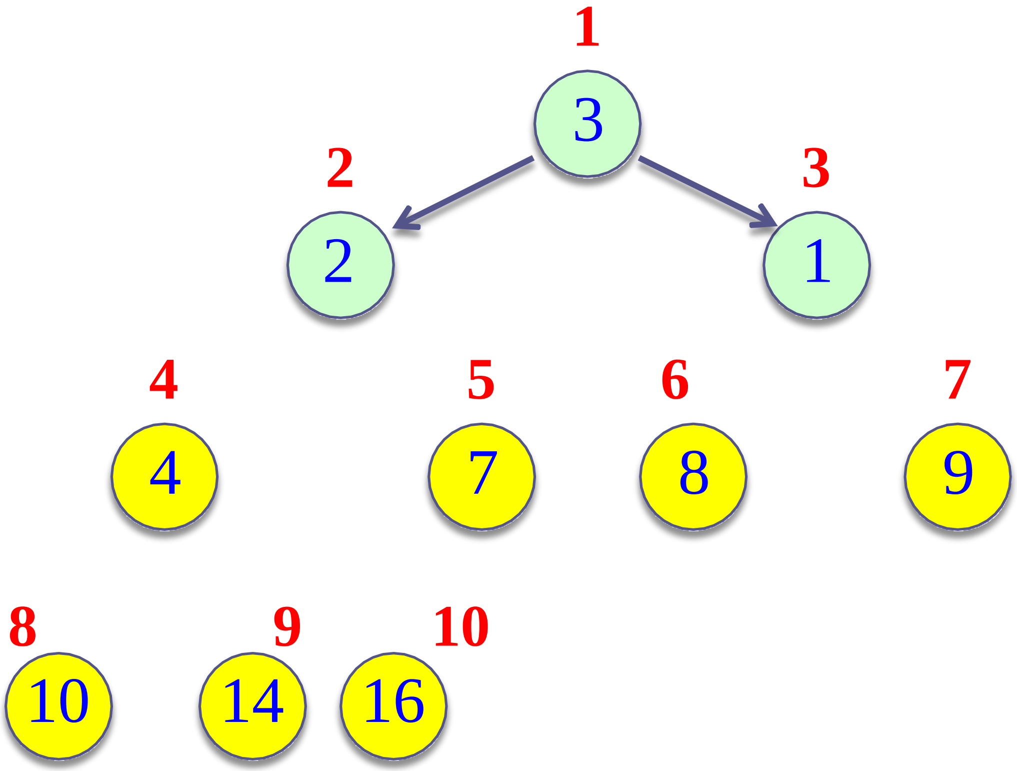
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ downto 2 do
        exchange $A[1] \leftrightarrow A[i]$
        HEAPIFY(A, 1, $i-1$)
```



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

Heapsort Algorithm

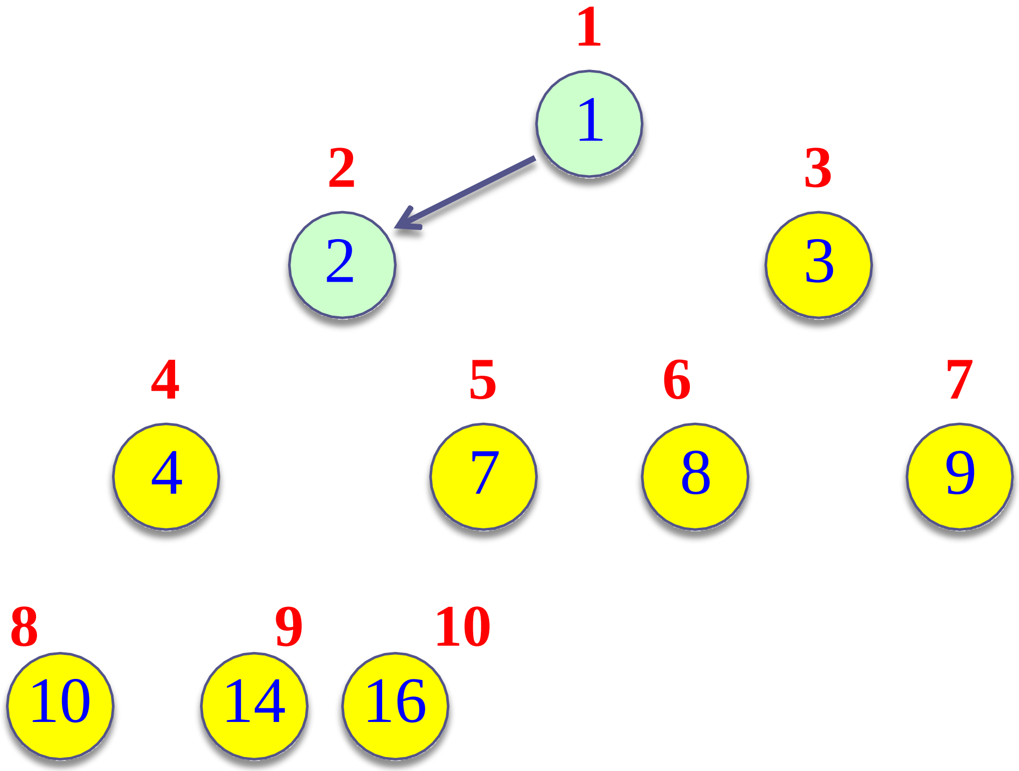
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ downto 2 do
        exchange A[1] ↔ A[$i$]
        HEAPIFY(A, 1, $i$–1)
```



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

Heapsort Algorithm

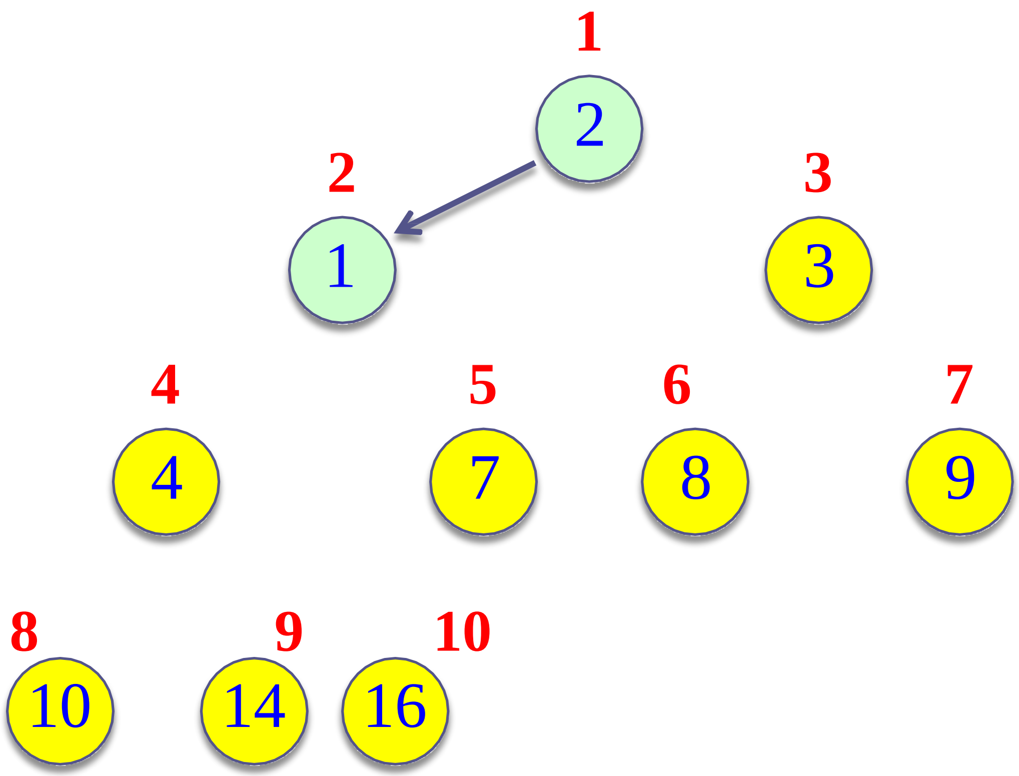
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ downto 2 do
        exchange $A[1] \leftrightarrow A[i]$
        HEAPIFY(A, 1, $i-1$)
```



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

Heapsort Algorithm

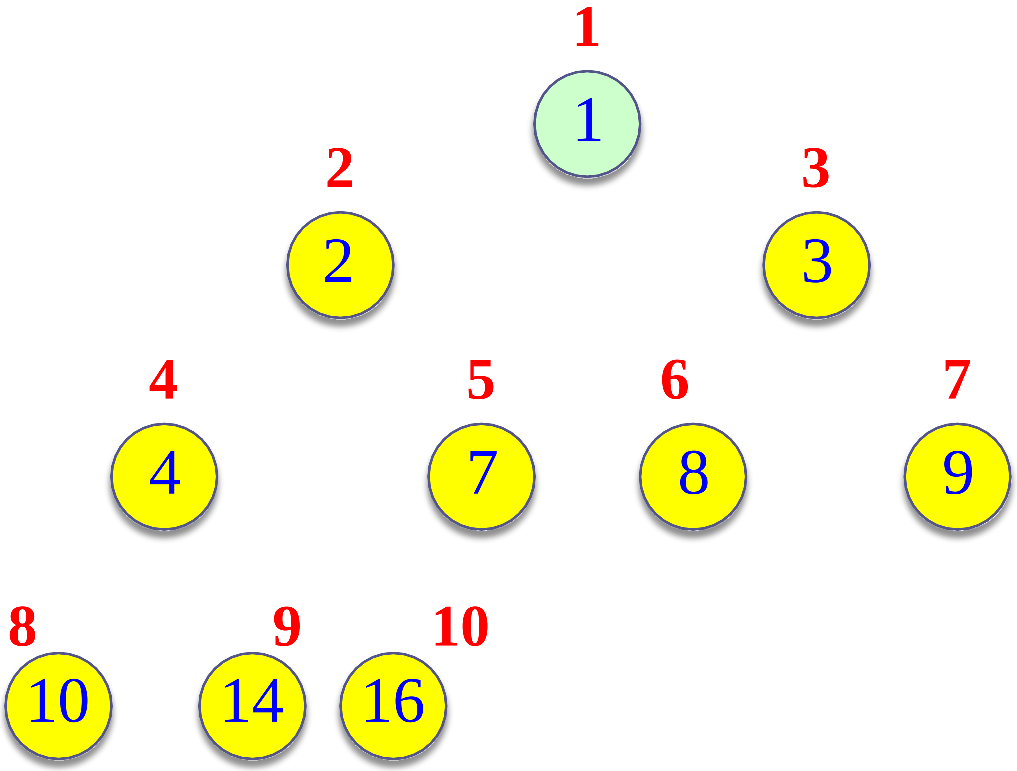
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ downto 2 do
        exchange $A[1] \leftrightarrow A[i]$
        HEAPIFY(A, 1, $i-1$)
```



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

Heapsort Algorithm

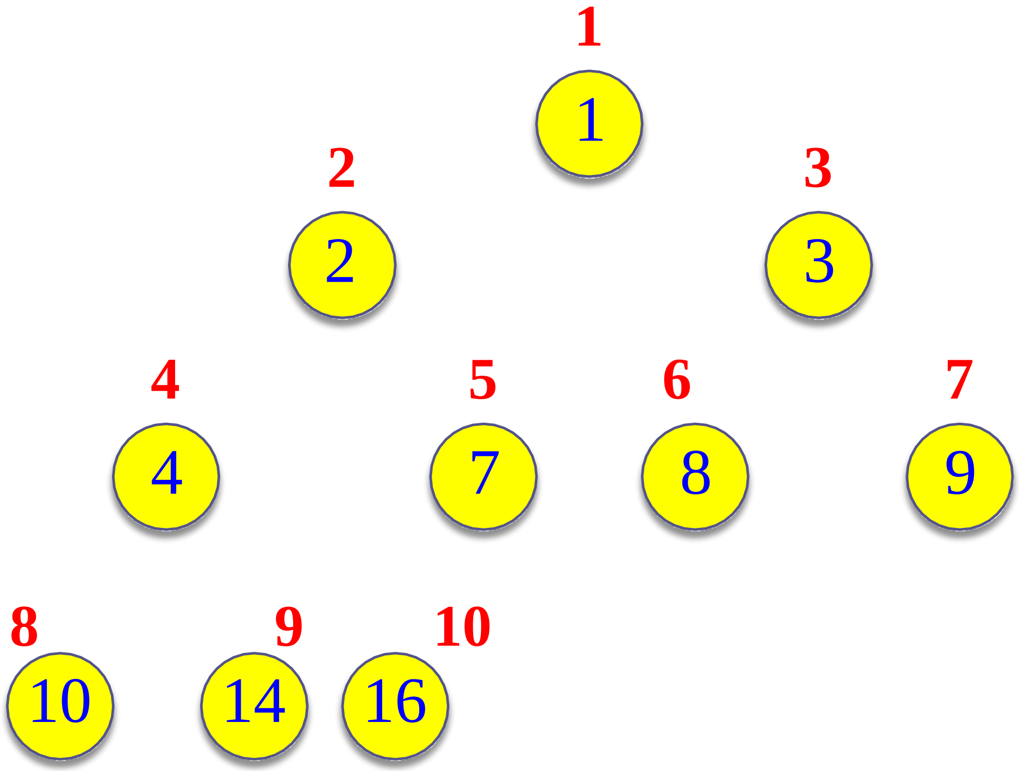
```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for $i \leftarrow n$ downto 2 do
        exchange A[1] ↔ A[$i$]
        HEAPIFY(A, 1, $i$–1)
```



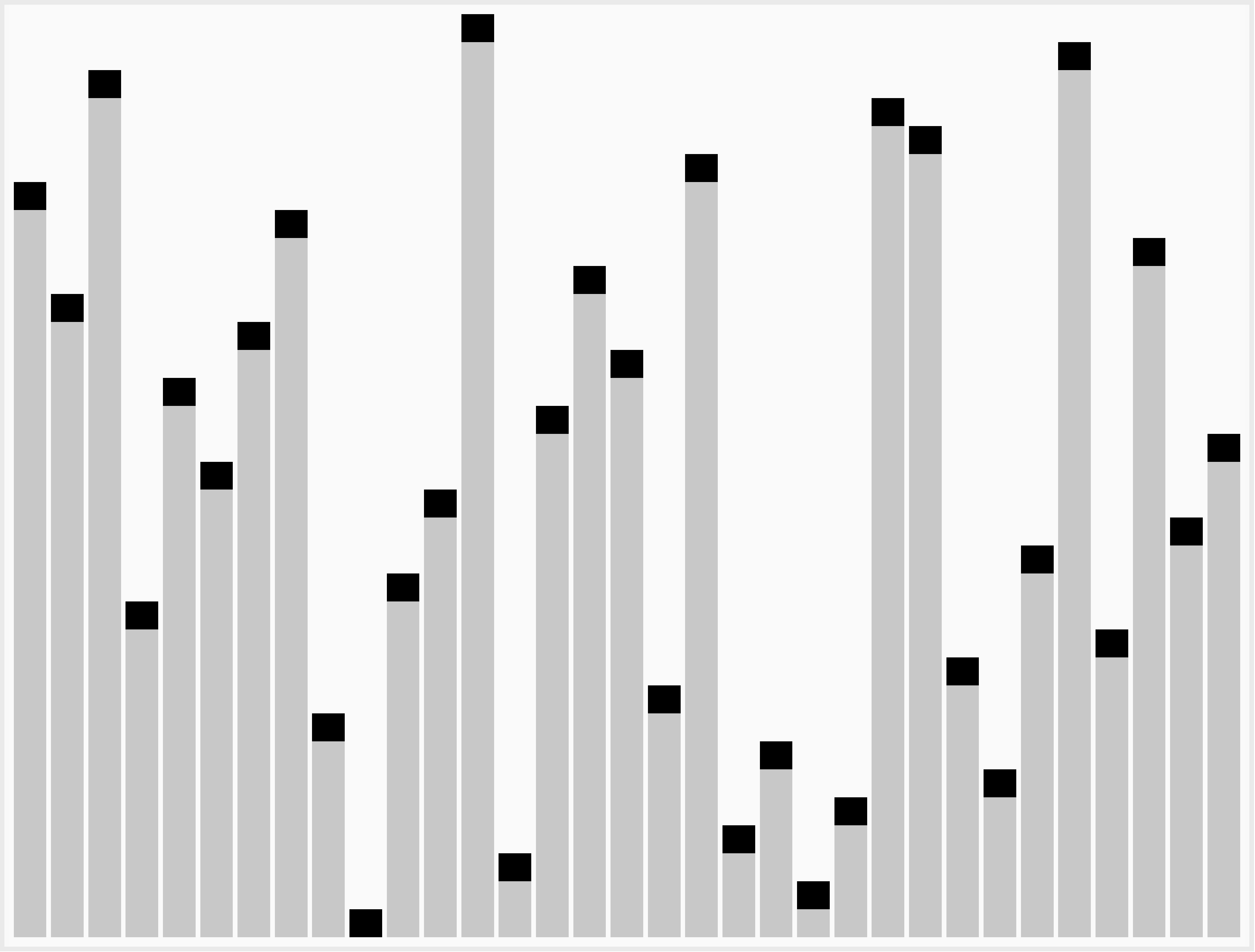
	1	2	3	4	5	6	7	8	9	10
A	1	2	3	4	7	8	9	10	14	16

Heapsort Algorithm

```
HEAPSORT(A, n)
    BUILD-HEAP(A, n)
    for i ← 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: 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

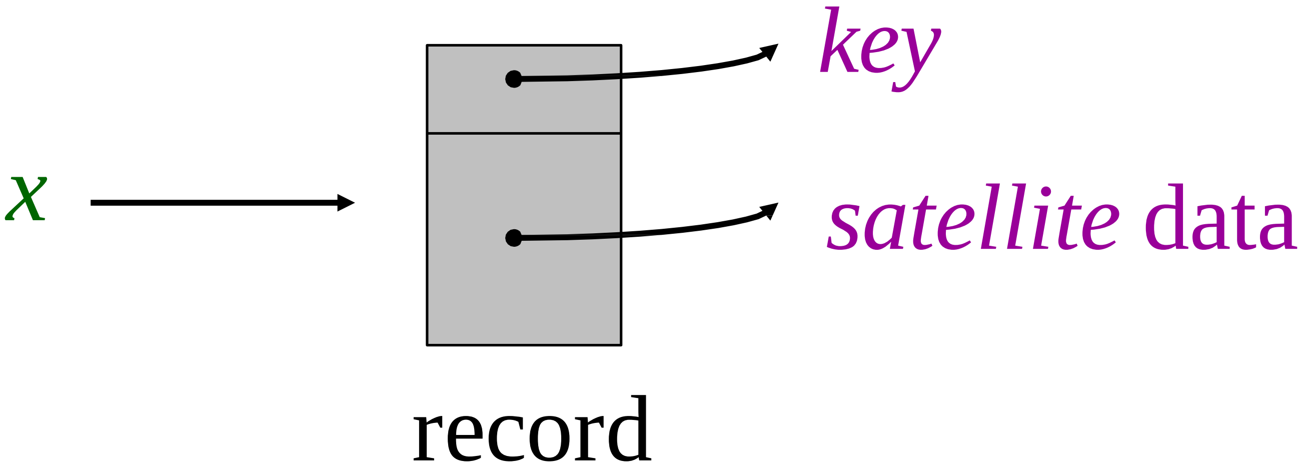
- Heapsort is a very good algorithm but, a good implementation of quicksort always beats heapsort in practice
- However, heap data structure has many popular applications, and it can be efficiently used for implementing priority queues

Outline

- Heaps
- Maintaining the heap property
- Building a heap
- The heapsort algorithm
- **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

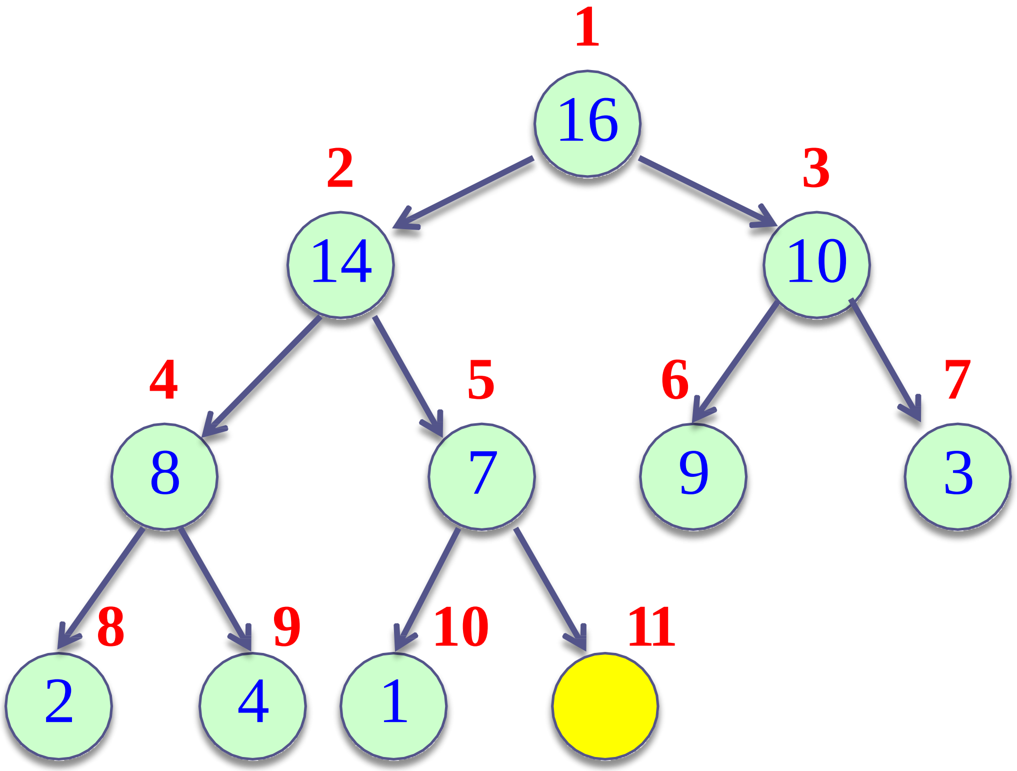
–HEAPIFY: compares parent with both children

–HEAP-INSERT: with only one

```
HEAP-INSERT(A, key, n)
    $n \leftarrow n + 1$
    $i \leftarrow n$
    while $i > 1$ and $A[\lfloor i/2 \rfloor] < key$ do
        $A[i] \leftarrow A[\lfloor i/2 \rfloor]$
        $i \leftarrow \lfloor i/2 \rfloor$
    $A[i] \leftarrow key$
```

Example: HEAP-INSERT(A, 15)

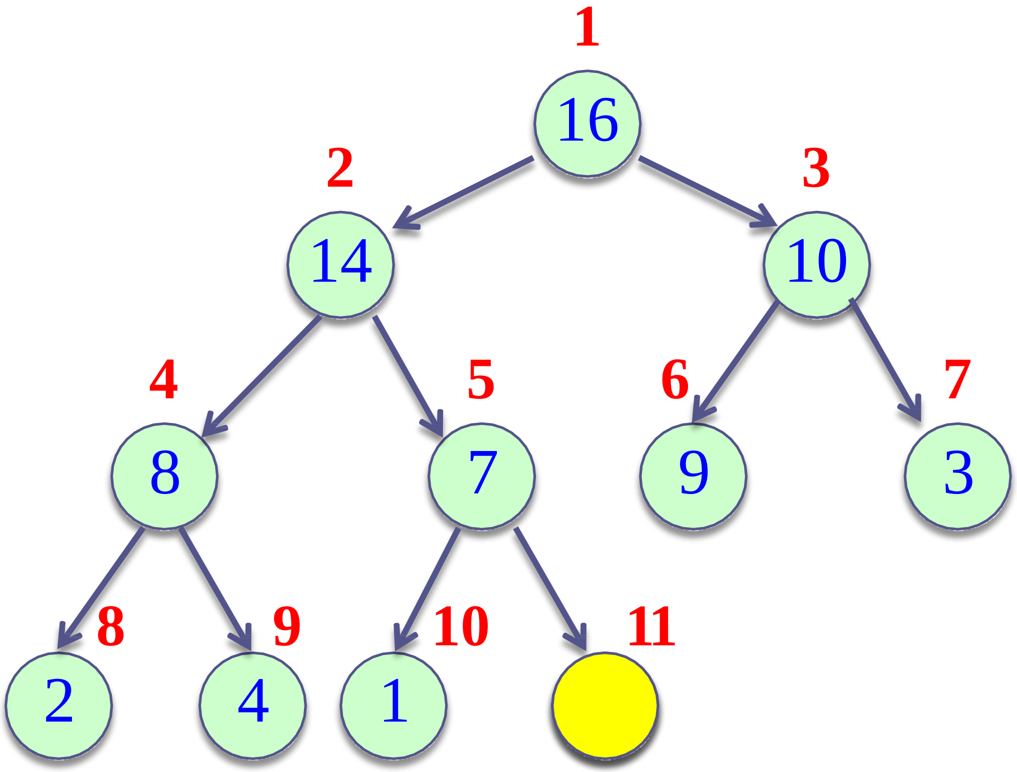
```
HEAP-INSERT(A, key, n)
    $n \leftarrow n + 1$
    $i \leftarrow n$
    while $i > 1$ and $A[i/2] < key$ do
        $A[i] \leftarrow A[i/2]$
        $i \leftarrow [i/2]$
    $A[i] \leftarrow key$
```



key = 15

Example: HEAP-INSERT(A, 15)

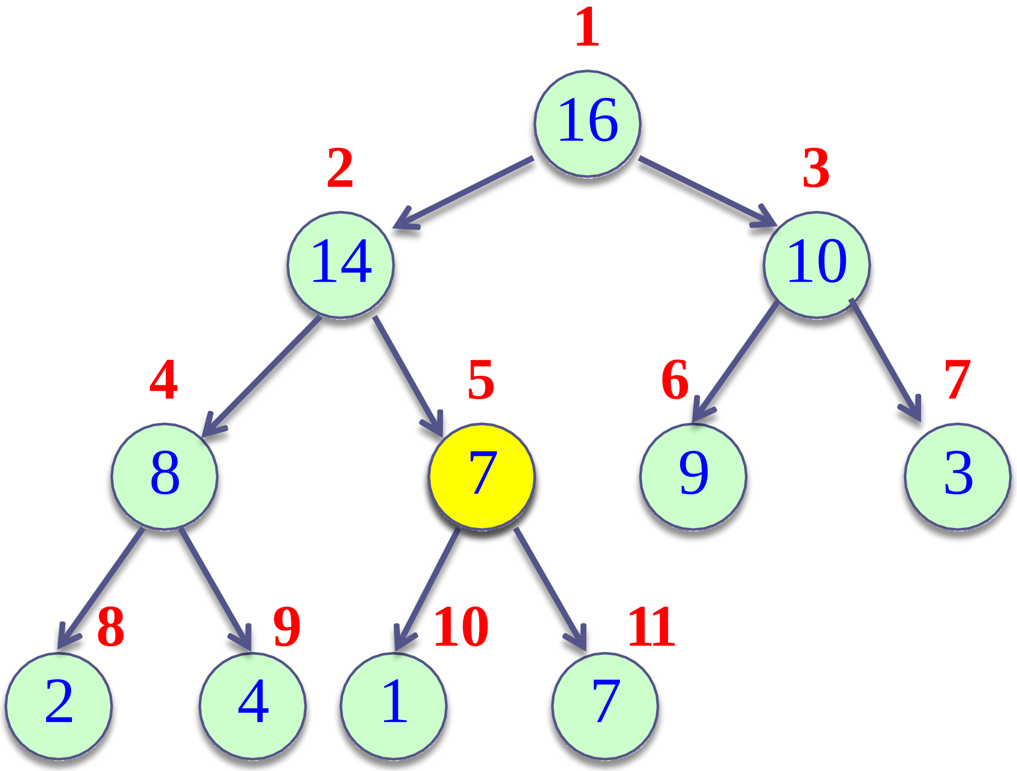
```
HEAP-INSERT(A, key, n)
    n ← n +1
    i ← n
    while i >1 and A[i/2] < key do
        A[i] ← A[i/2 ]
        i ← [i/2 ]
    A[i] ← key
```



key = 15

Example: HEAP-INSERT(A, 15)

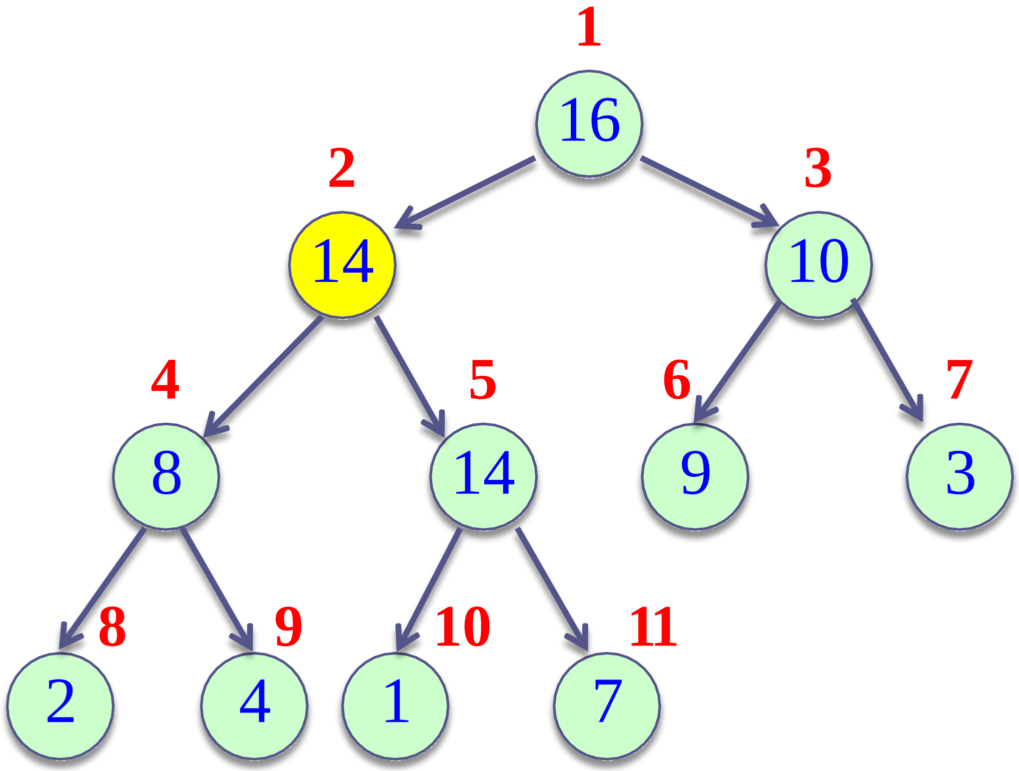
```
HEAP-INSERT(A, key, n)
    $n \leftarrow n + 1$
    $i \leftarrow n$
    while $i > 1$ and $A[i/2] < key$ do
        $A[i] \leftarrow A[i/2]$
        $i \leftarrow [i/2]$

    $A[i] \leftarrow key$
```



key = 15

Example: HEAP-INSERT(A, 15)

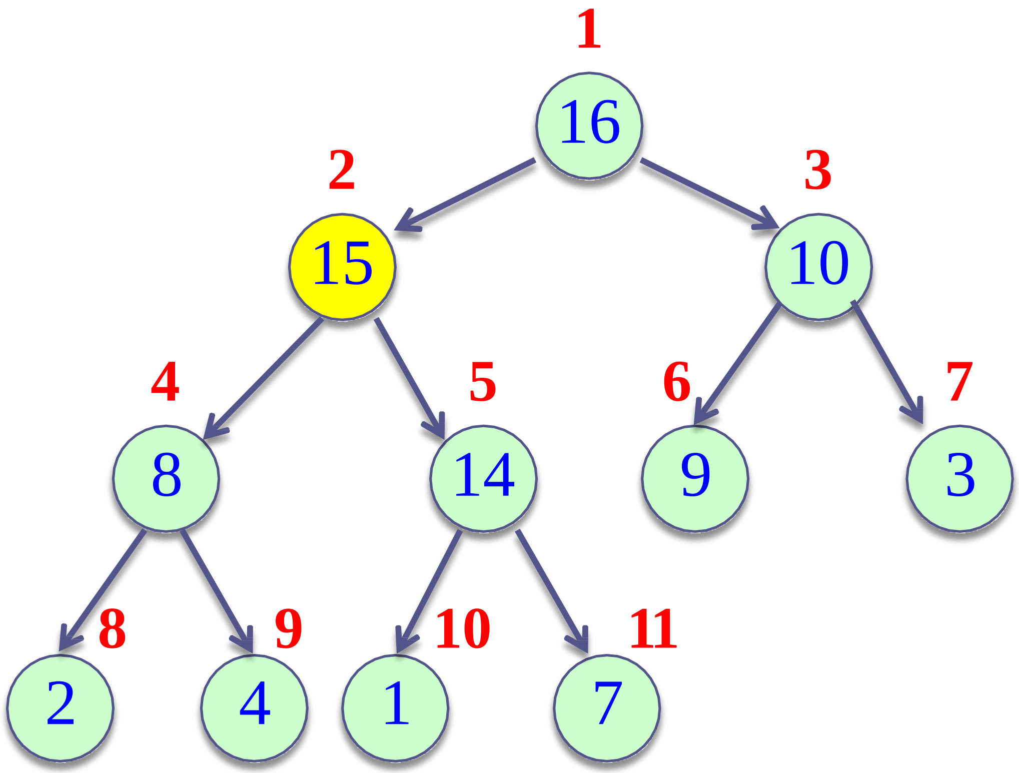
```
HEAP-INSERT(A, key, n)
    $n \leftarrow n + 1$
    $i \leftarrow n$
    while $i > 1$ and $A[i/2] < key$ do
        $A[i] \leftarrow A[i/2]$
        $i \leftarrow [i/2]$
    $A[i] \leftarrow key$
```



key = 15

Example: HEAP-INSERT(A, 15)

```
HEAP-INSERT(A, key, n)
    $n \leftarrow n + 1$
    $i \leftarrow n$
    while $i > 1$ and $A[i/2] < key$ do
        $A[i] \leftarrow A[i/2]$
        $i \leftarrow [i/2]$
➡   $A[i] \leftarrow key$
```



key = 15

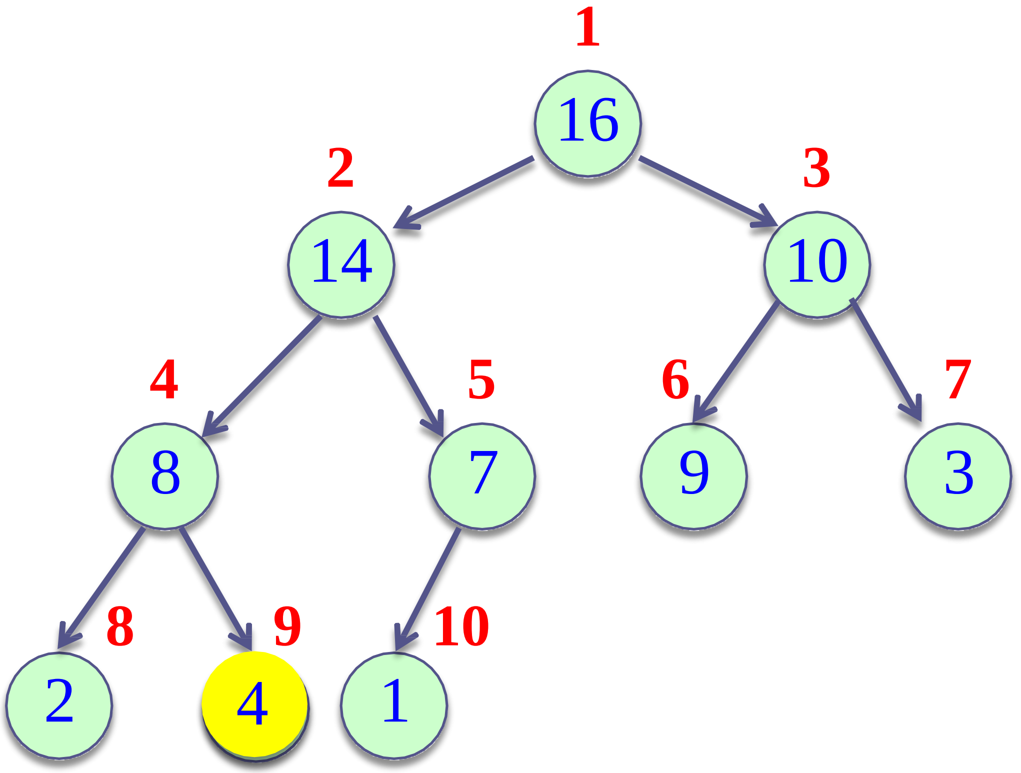
Heap Increase Key

- 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)

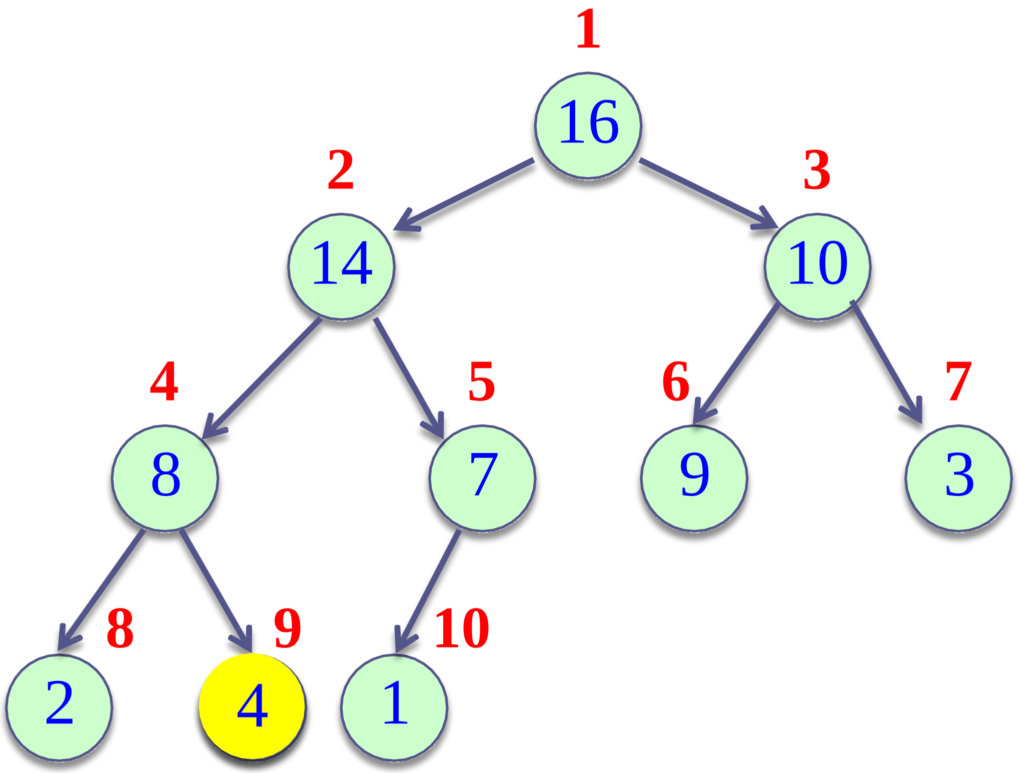
```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i] then
        return error
    while i >1 and A[i/2]< key do
        A[i] ← A[i/2]
        i ← [i/2]
    A[i] ← key
```



key = 15

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

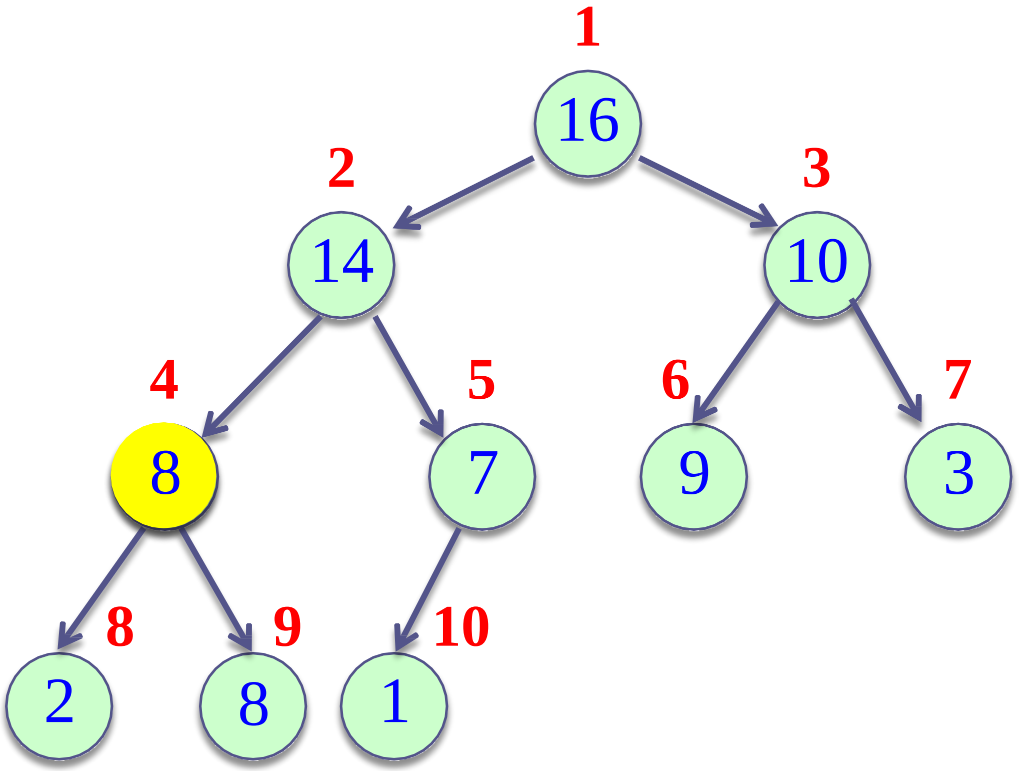
```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i] then
        return error
    while i >1 and A[i/2]< key do
        A[i] ← A[i/2]
        i ← [i/2]
    A[i] ← key
```



key = 15

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

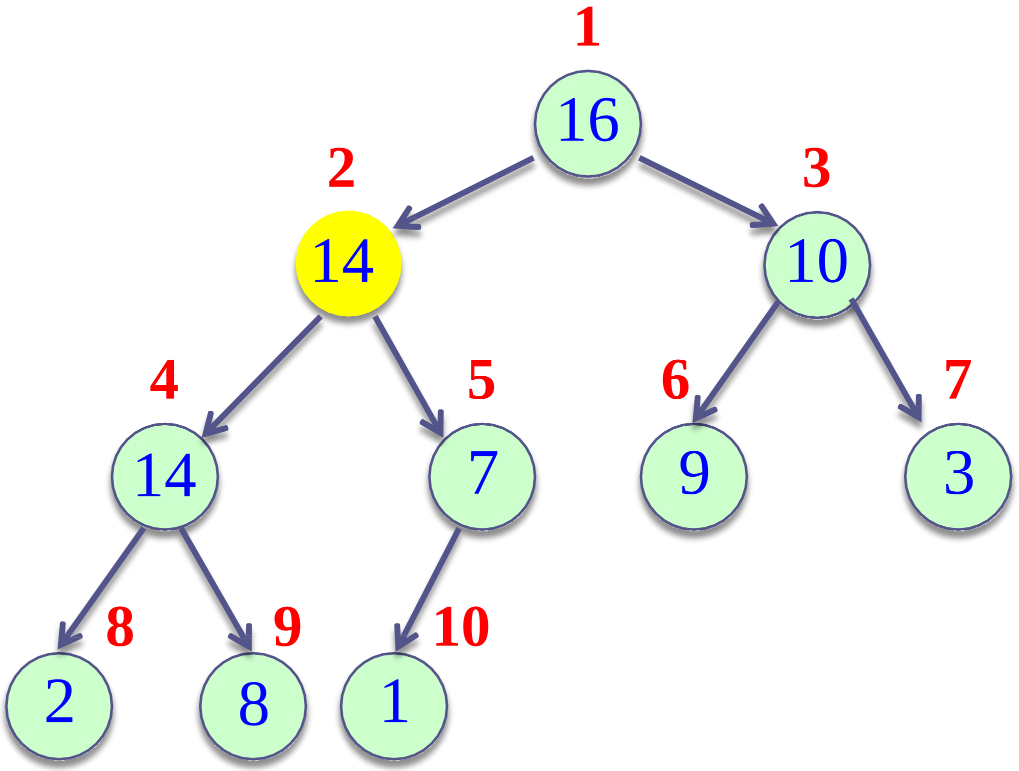
```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i] then
        return error
    while i >1 and A[i/2] < key do
        A[i] ← A[i/2]
        i ← [i/2]
    A[i] ← key
```



key = 15

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

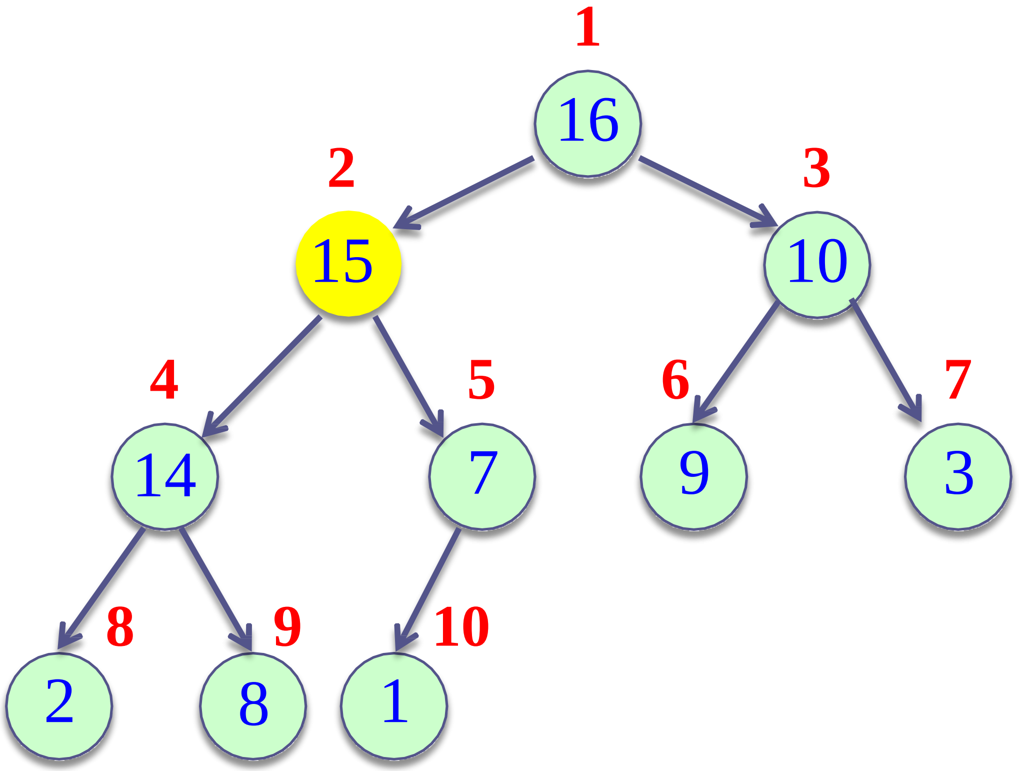
```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i] then
        return error
    while i >1 and A[i/2] < key do
        A[i] ← A[i/2]
        i ← [i/2]
    A[i] ← key
```



key = 15

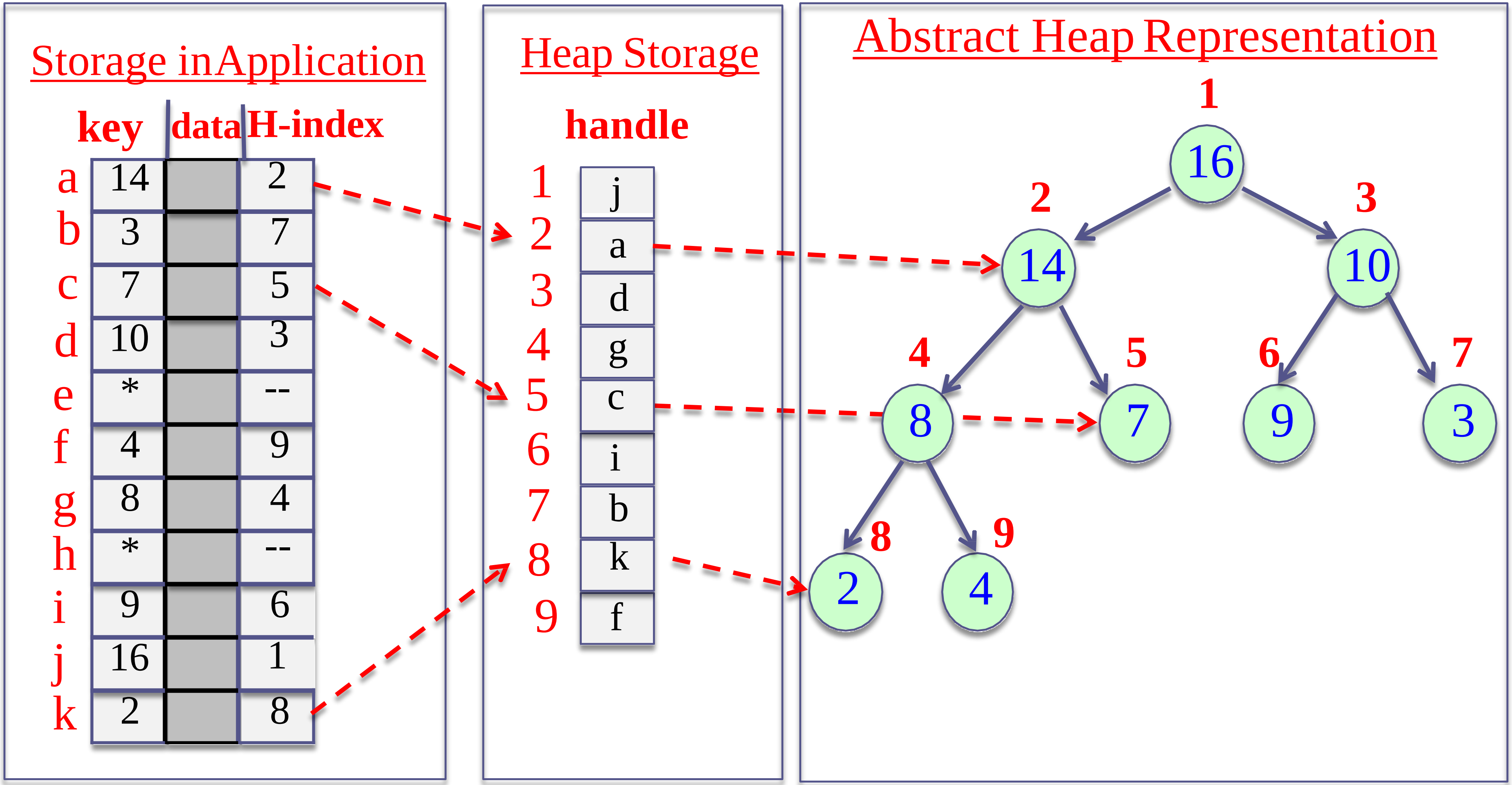
Example: HEAP-INCREASE-KEY(A, 9, 15)

```
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 inApplication

	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

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 (fromA[i] to x)

Call Heapify(A, i)

Runtime: O(lgn)

Example Problem: Phone Operator

A phone operator answering n phones

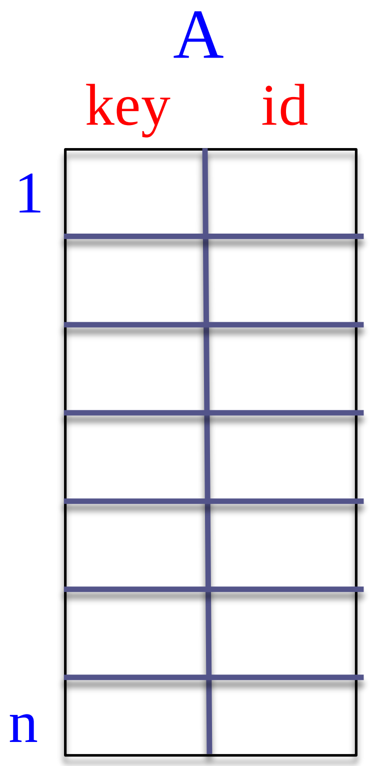
Each phone i has x_i people waiting in line for their calls to be answered.

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

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)
```