Design and Analysis of Algorithms

Heap Sort

Fall 2022

National University of Computer and Emerging Sciences,
Islamabad

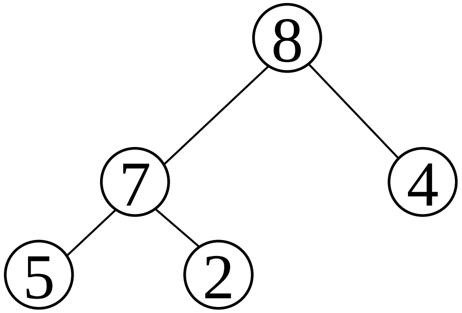
Heapsort

- Running time of heapsort is **$O(n \log_2 n)$**
- It sorts **in place**
- It uses a data structure called a *heap*
- The heap data structure is also used to implement a priority queue efficiently

The Heap Data Structure

- *Def:* **A heap is a <u>nearly complete</u> 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**

$$\text{Parent}(x) \geq x$$



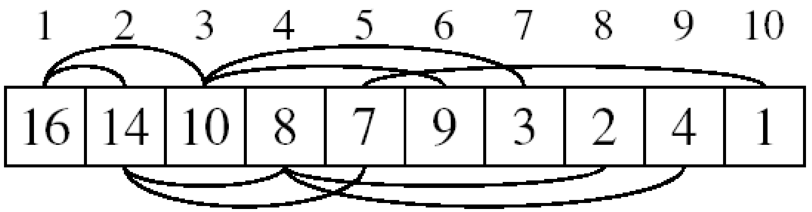
Heap

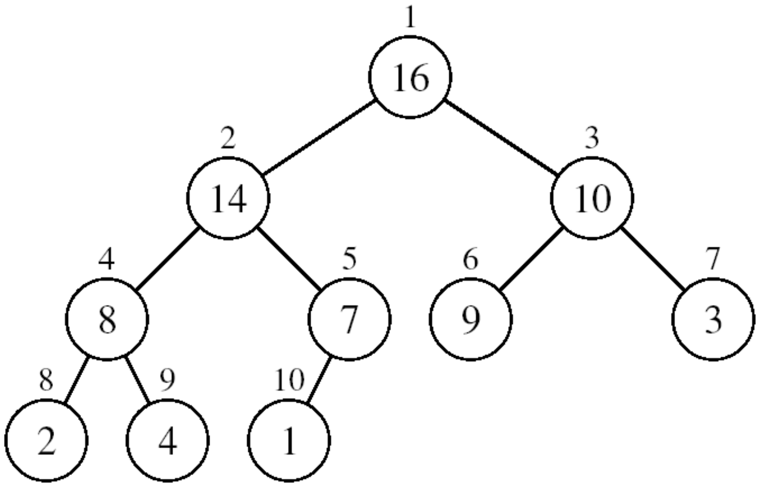
From the heap property, it follows that:
"The root is the maximum element of the heap!"

A heap is a binary tree that is filled in order

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





Heap Types

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

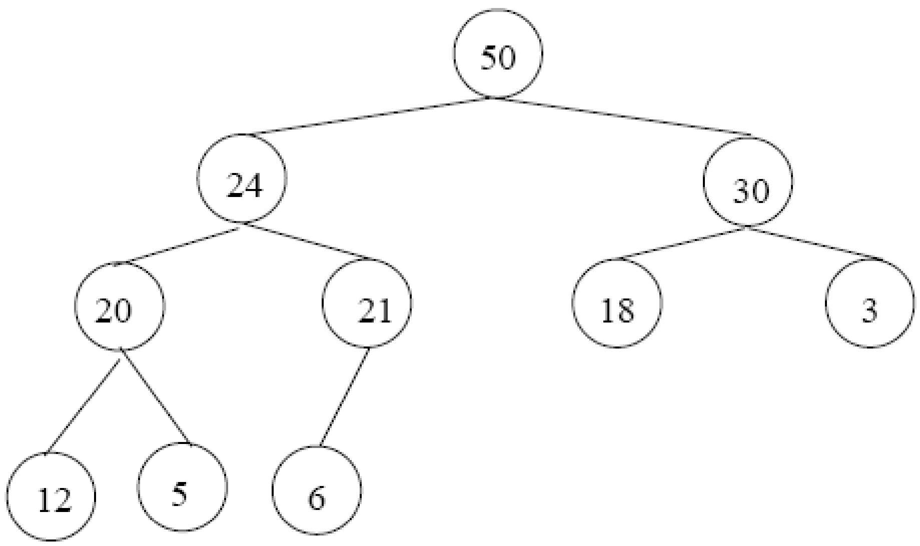
$$A[\mathrm{PARENT}(i)] \geq A[i]$$

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

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

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)**

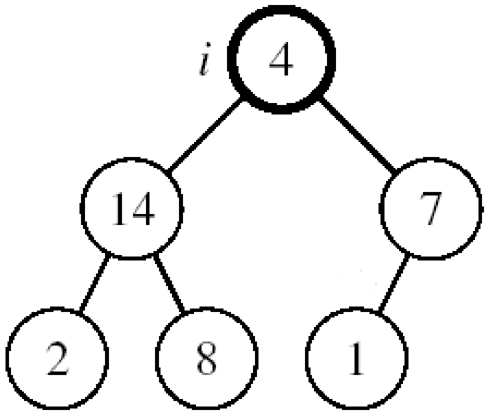


Operations on Heaps

- **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**
- **Priority queues**

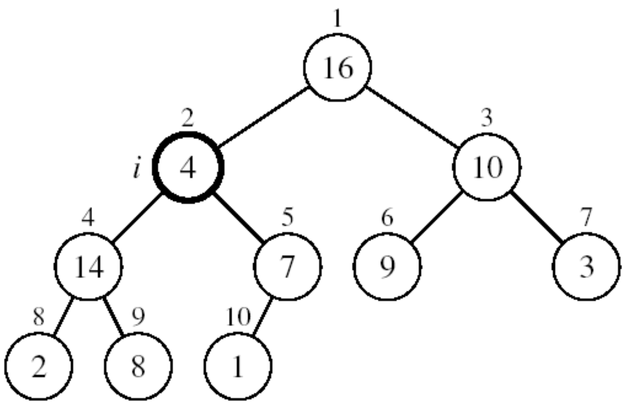
Maintaining the Heap Property

- 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



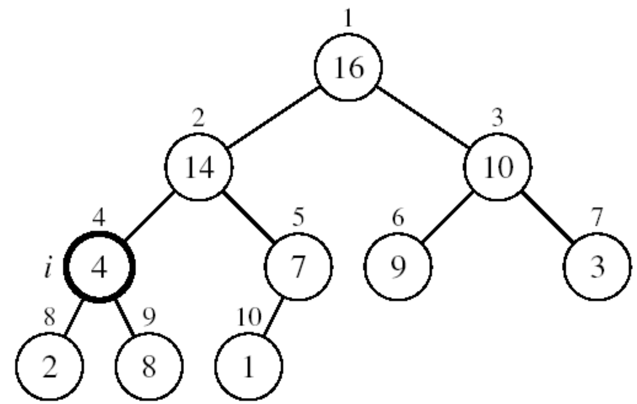
Example

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



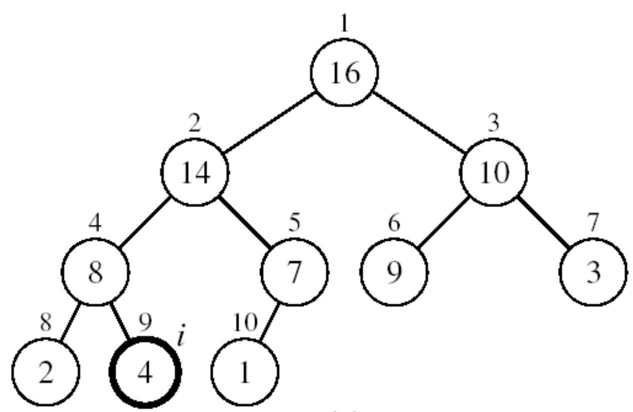
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

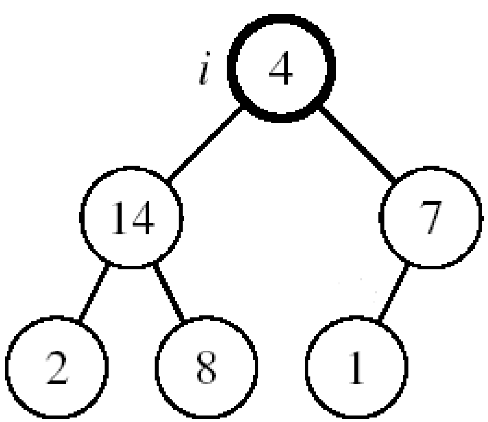
A[4] ↔ A[9]



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

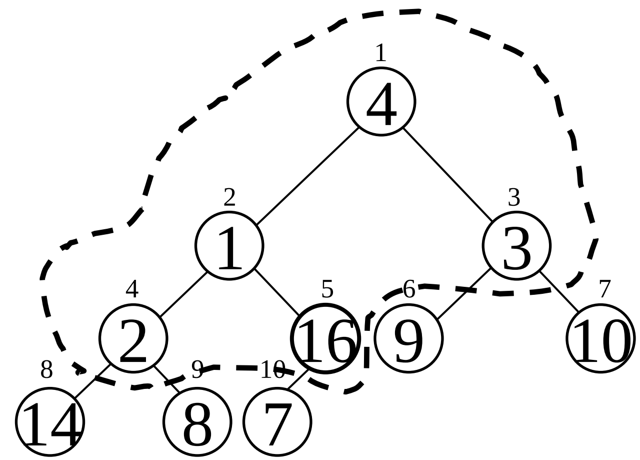
- **Intuitively:**
  - It traces a path from the root to a leaf (longest path length h)
  - At each level, it makes exactly 2 comparisons
  - Total number of comparisons is 2h
  - Running time is O(h) or $O(lgn)$
- **Running time of MAX-HEAPIFY is *O*(lg*n*)**
- **Can be written in terms of the height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

Building a Heap

- Convert an array $A[1 \ldots n]$ into a max-heap (n = length[A])
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \ldots n]$ 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 ← $\lfloor n/2 \rfloor$ downto 1
3. do MAX-HEAPIFY(*A*, *i*, *n*)

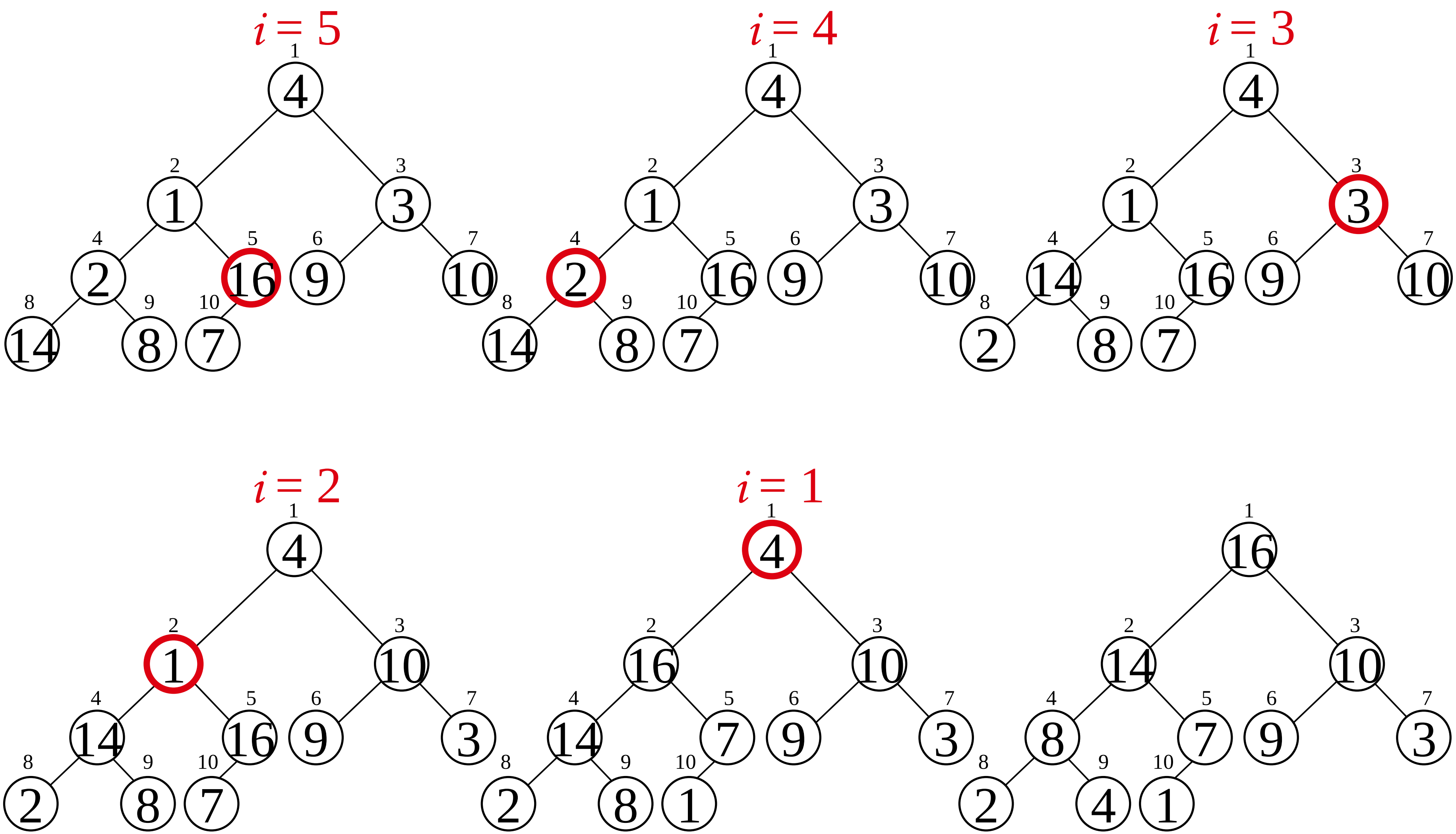


A:

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

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) $\quad O(\lg n)$

$$\left.\begin{array}{l} \text{lines 2–3} \end{array}\right\} O(n)$$

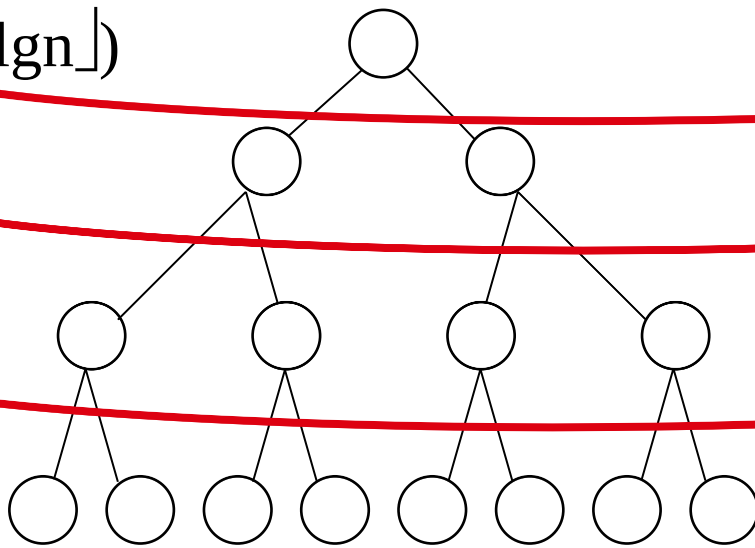
$\Rightarrow$ **Running time: $O(n\lg n)$**

- **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**

$$T(n)=O(n)$$

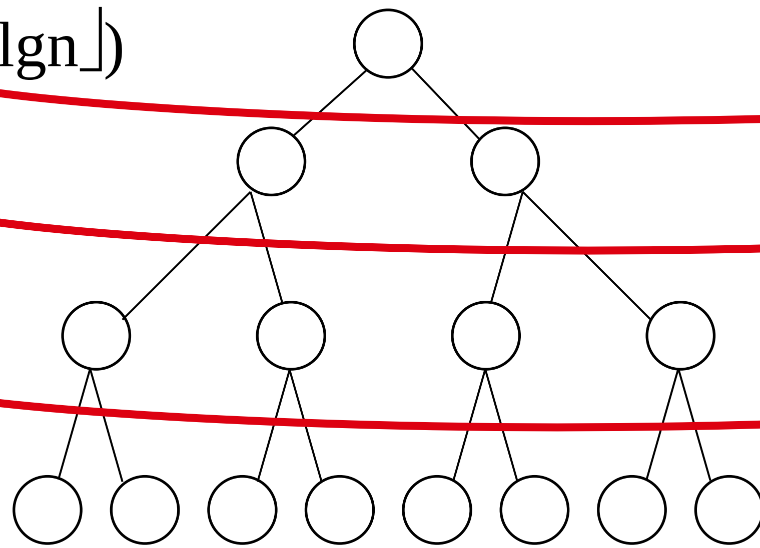


Height	Level	No. of nodes
$h_0 = 3$ ($\lfloor \lg n \rfloor$)	$i = 0$	2^0
$h_1 = 2$	$i = 1$	2^1
$h_2 = 1$	$i = 2$	2^2
$h_3 = 0$	$i = 3$ ($\lfloor \lg n \rfloor$)	2^3

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)**

Running Time of BUILD MAX HEAP

- We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.
- n-element heap has height $\lfloor \lg n \rfloor$
- 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)

Running Time of BUILD MAX HEAP

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

We evalaute the last summation by substituting $x = 1/2$ in the formula (A.8) yielding

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2. \end{aligned}$$

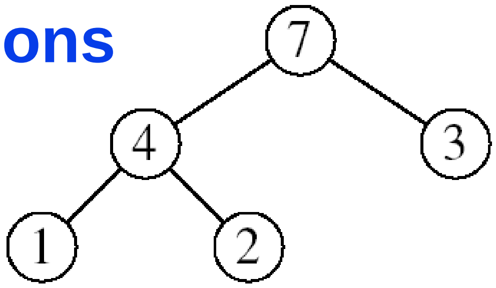
Thus, we can bound the running time of BUILD-MAX-HEAP as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n). \end{aligned}$$

Hence, we can build a max-heap from an unordered array in linear time.

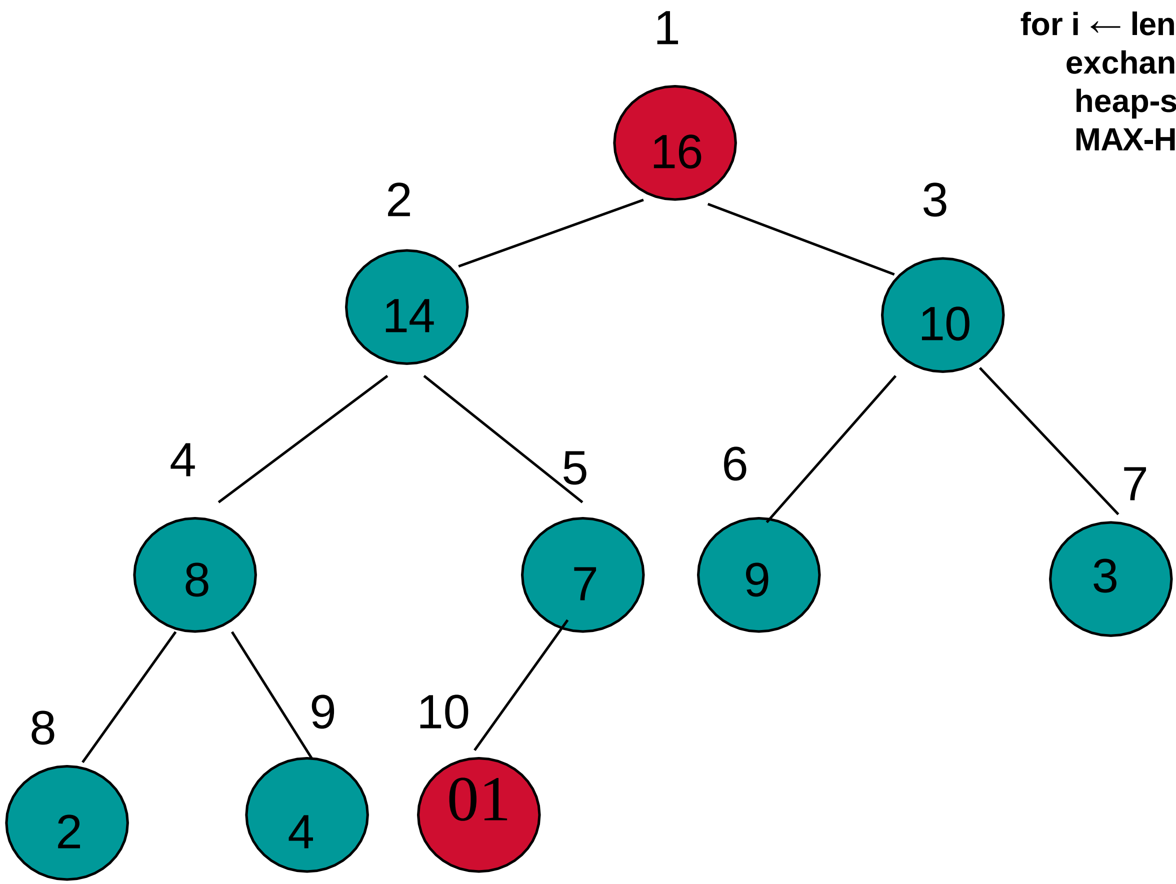
Heapsort

- **Goal:**
  - **Sort an array using heap representations**



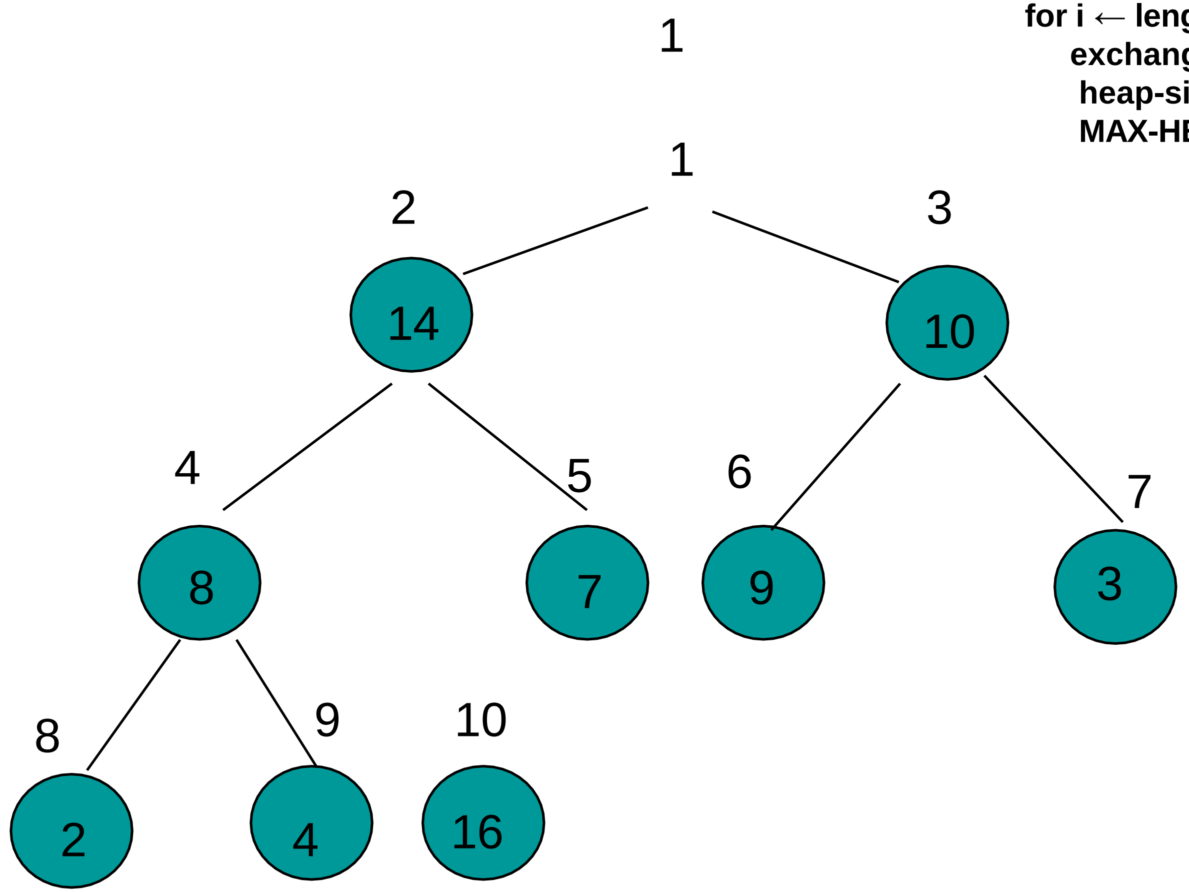
- **Idea:**
  - Build a max-heap from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

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



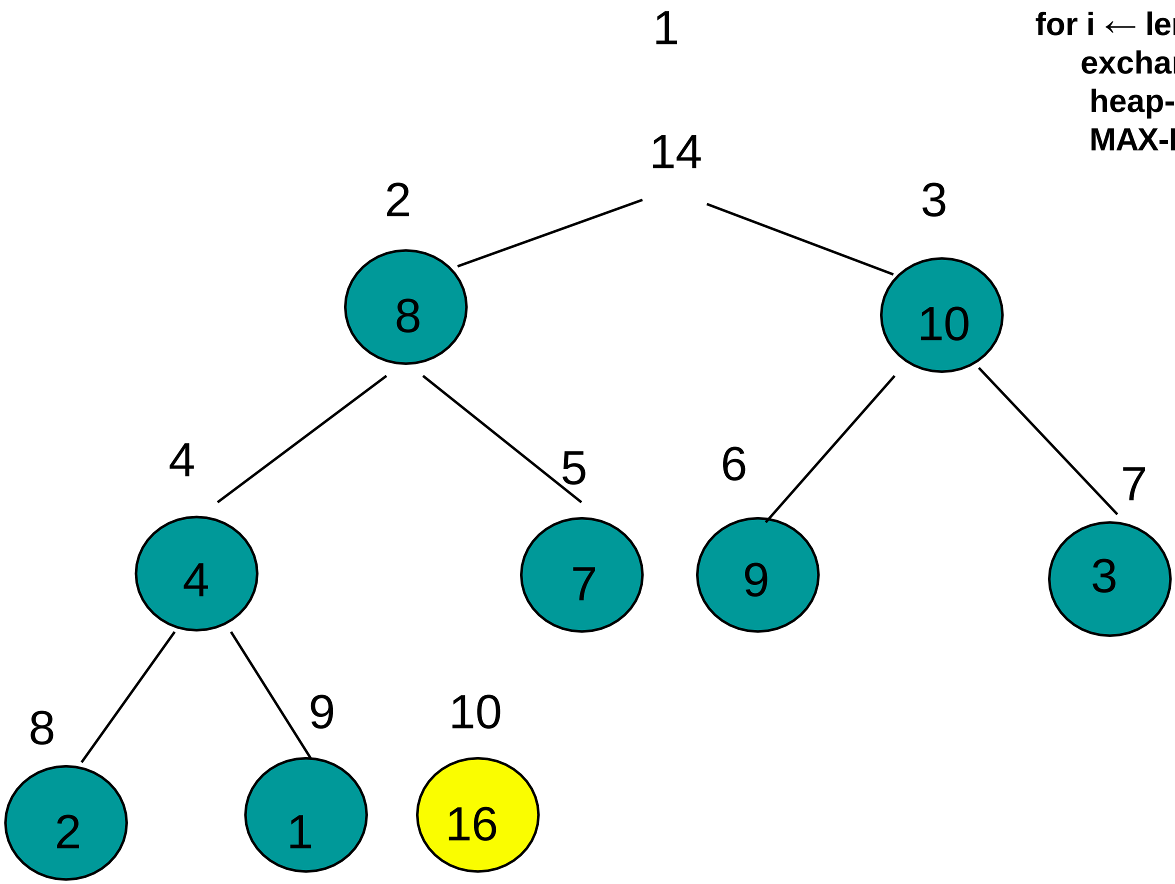
16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



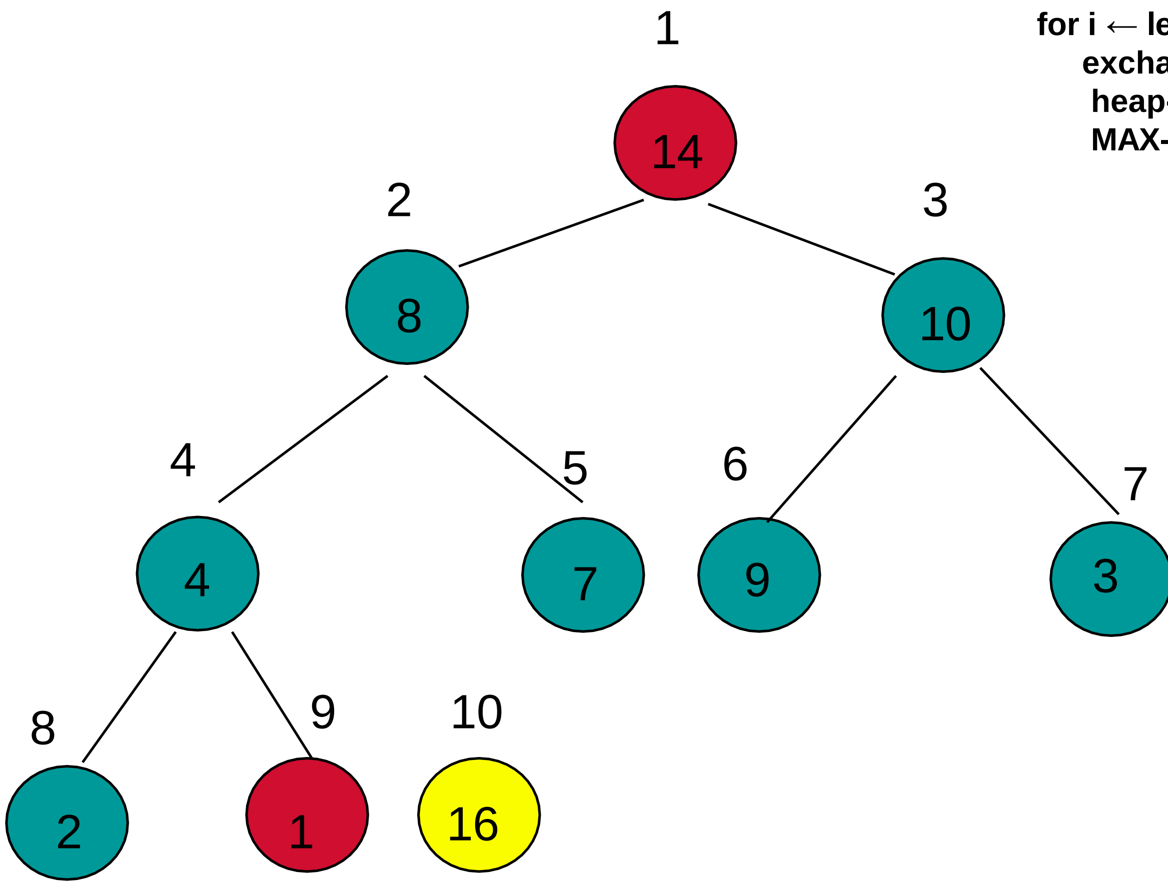
1	14	10	8	7	9	3	2	4	16
1	2	3	4	5	6	7	8	9	10

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



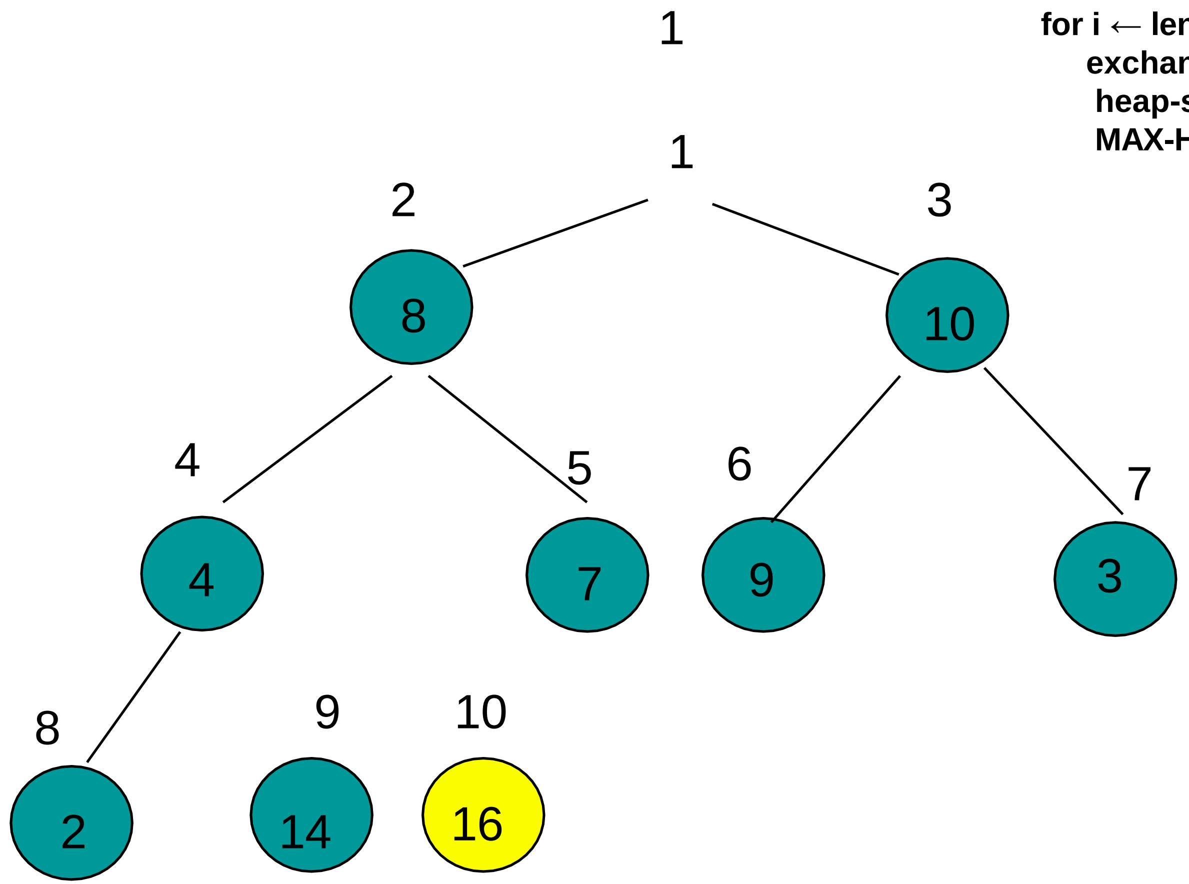
14	8	10	4	7	9	3	2	1	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



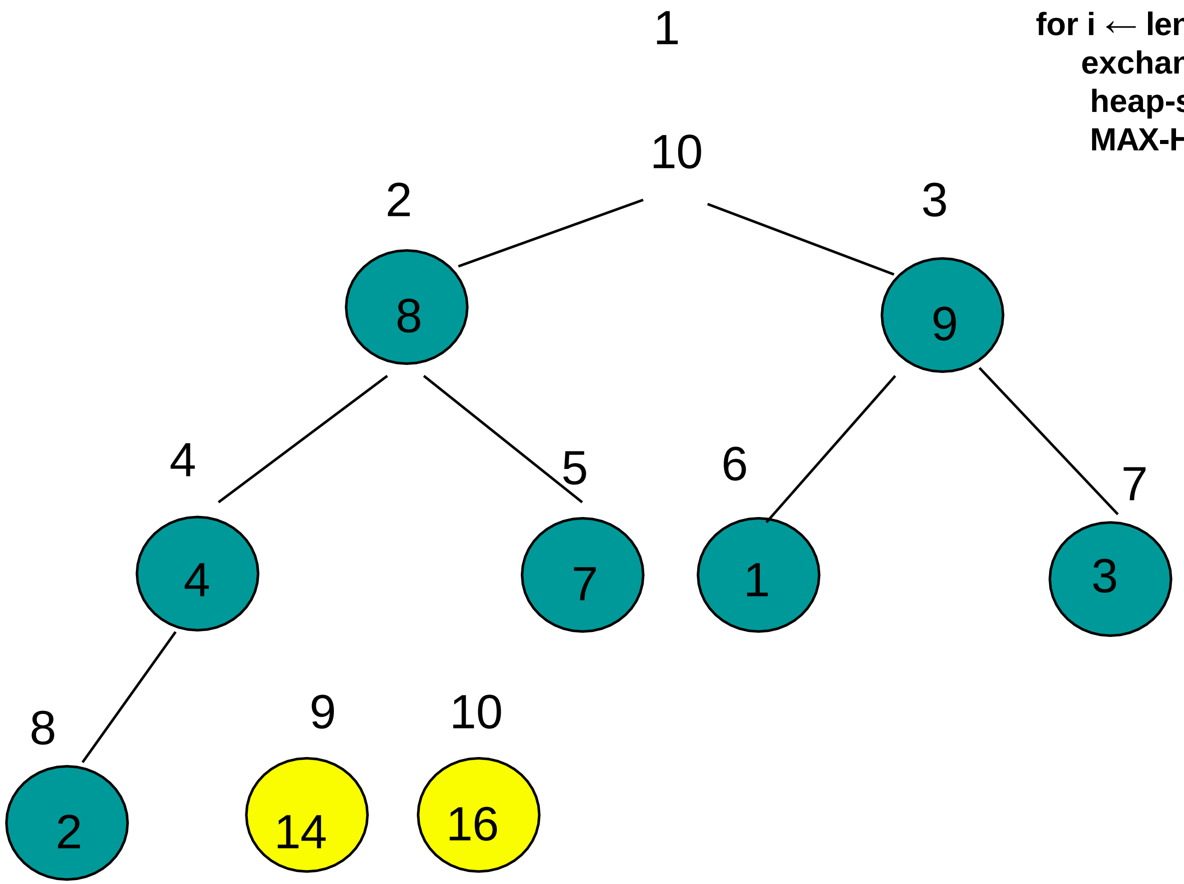
14	8	10	4	7	9	3	2	1	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



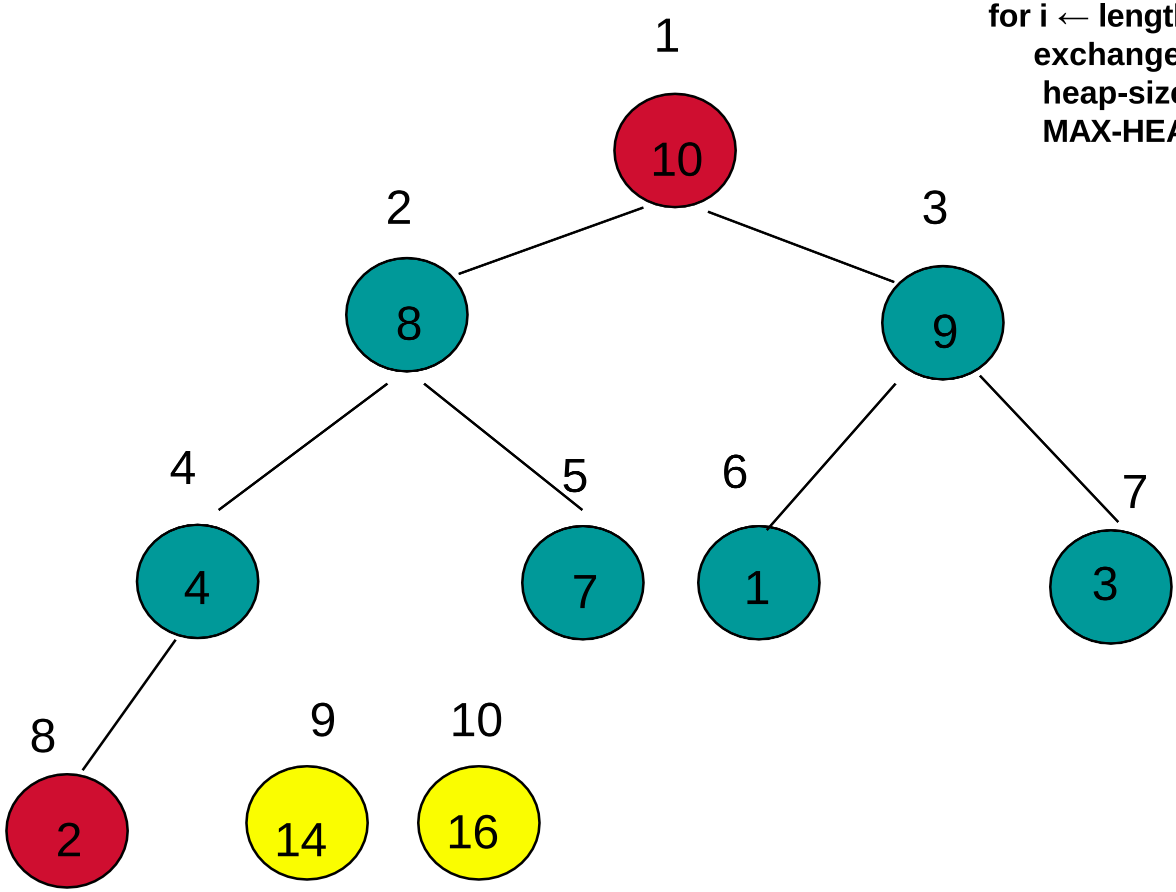
1	8	10	4	7	9	3	2	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



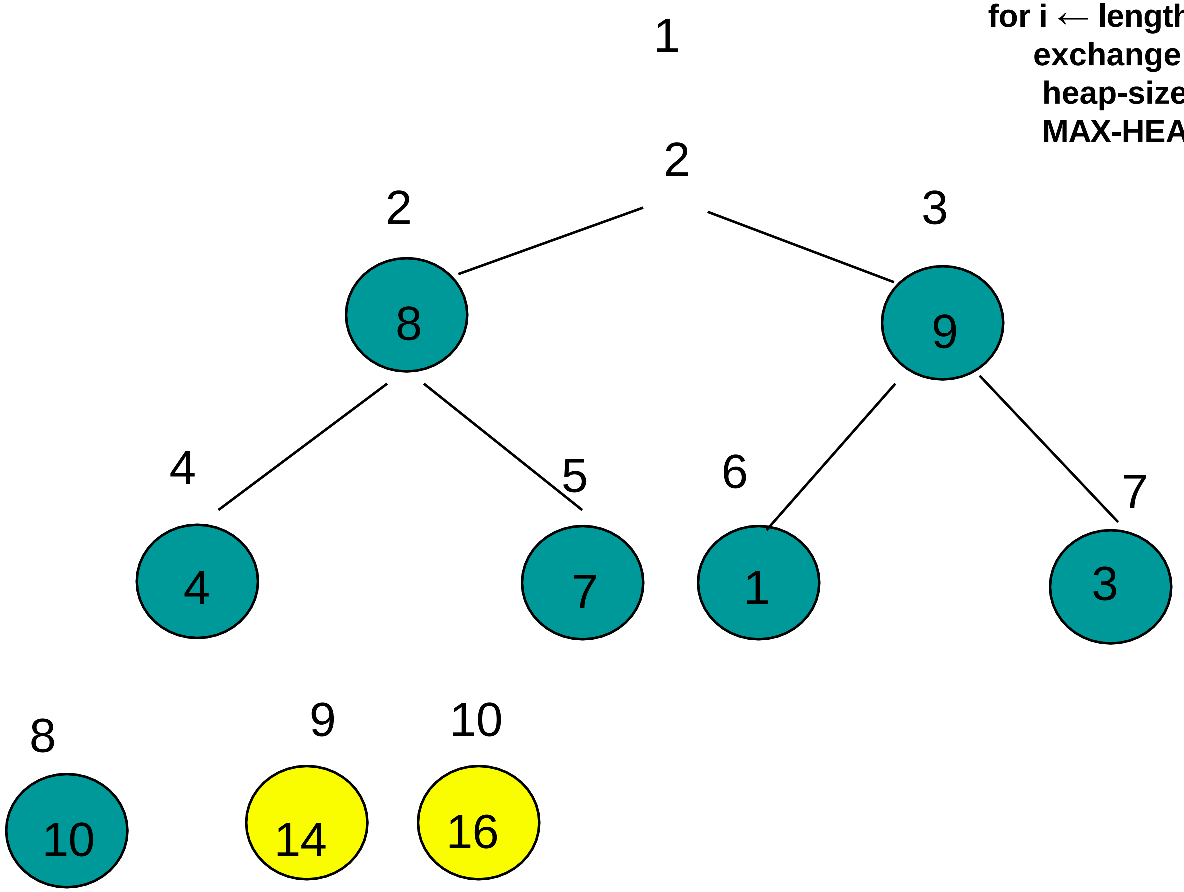
10	8	9	4	7	1	3	2	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



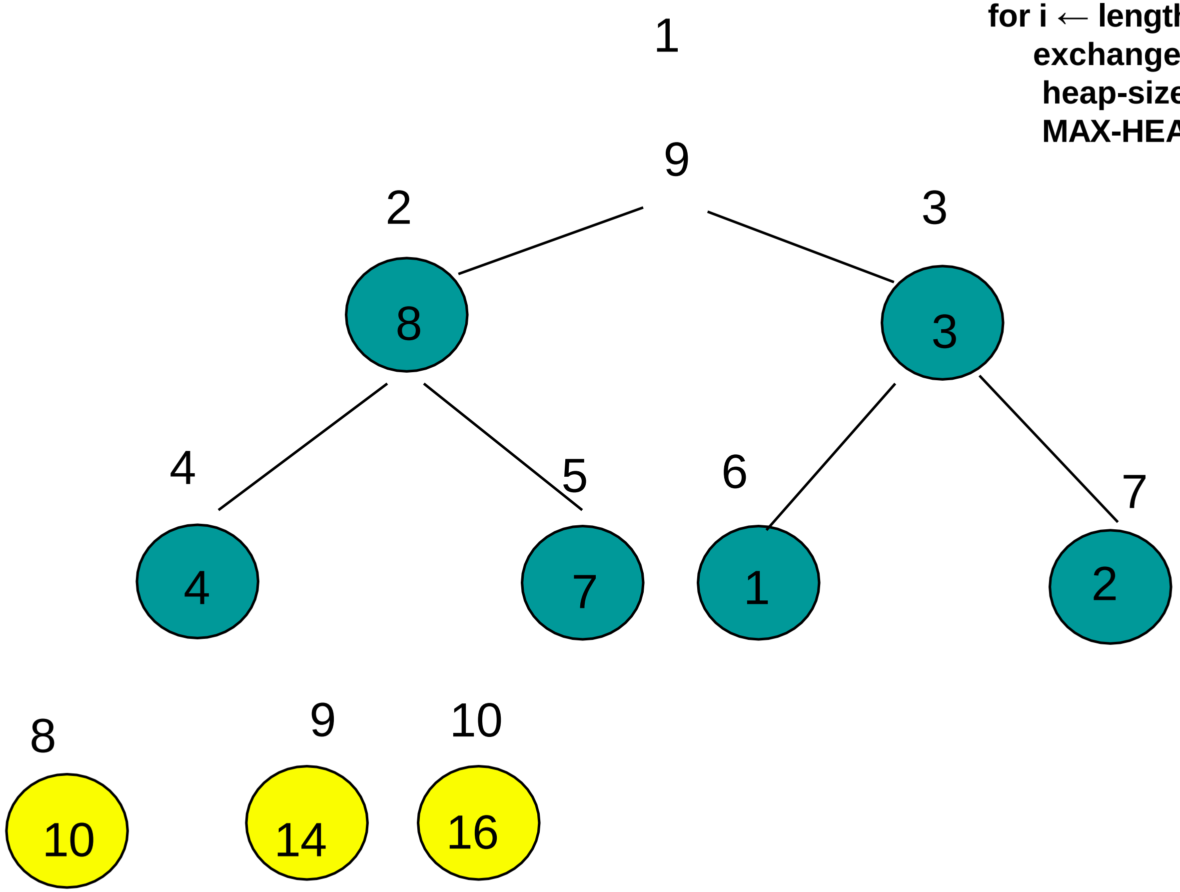
10	8	9	4	7	1	3	2	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



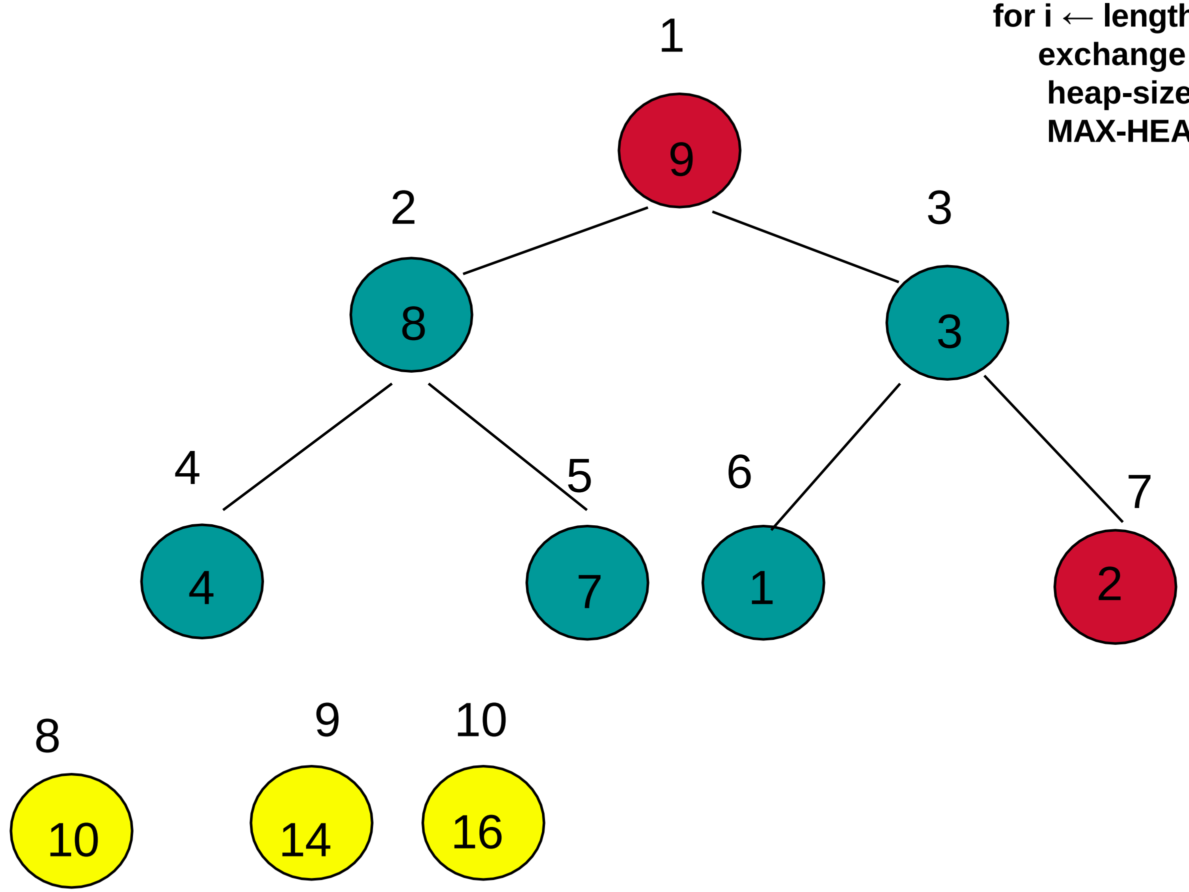
2	8	9	4	7	1	3	10	14	16
1	2	3	4	5	6	7	8	9	10

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



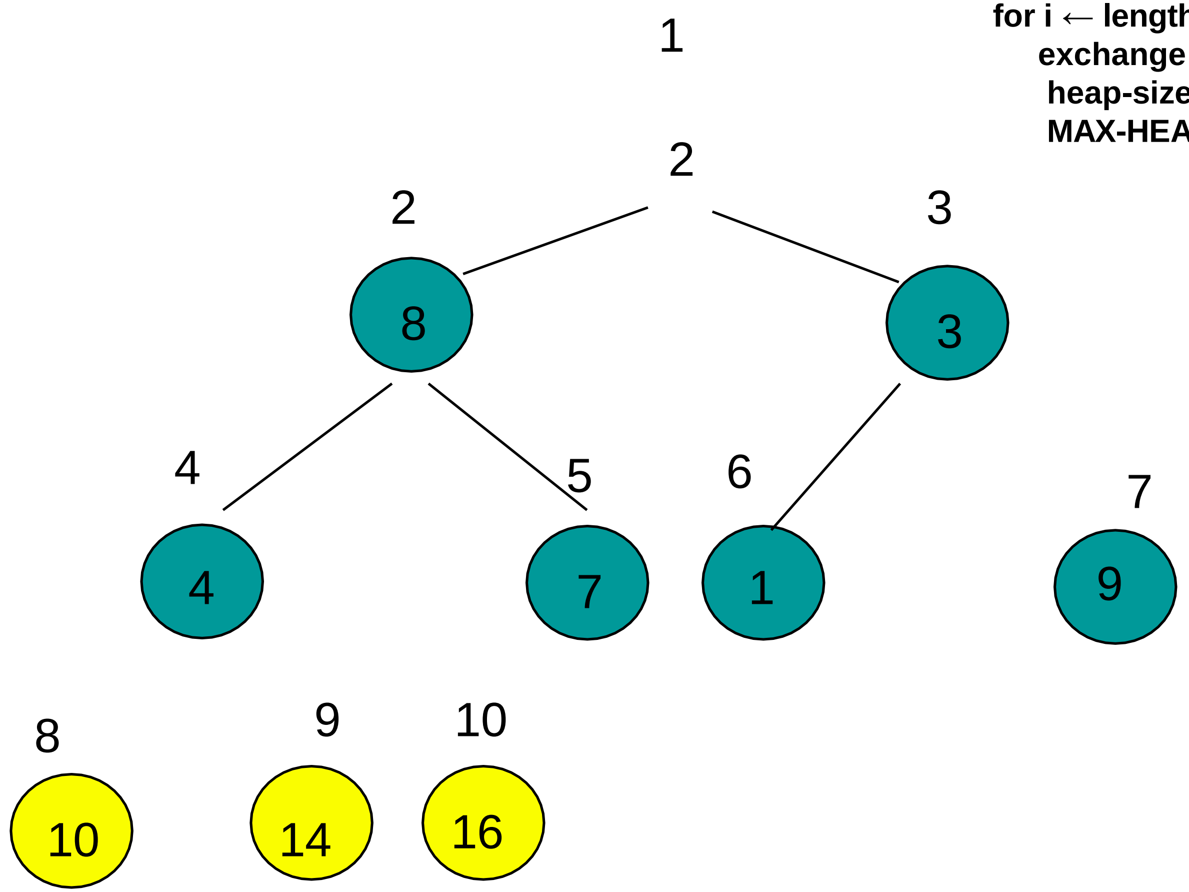
9	8	3	4	7	1	2	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
    heap-size[A] ← heap-size[A] – 1
    MAX-HEAPIFY(A, 1)



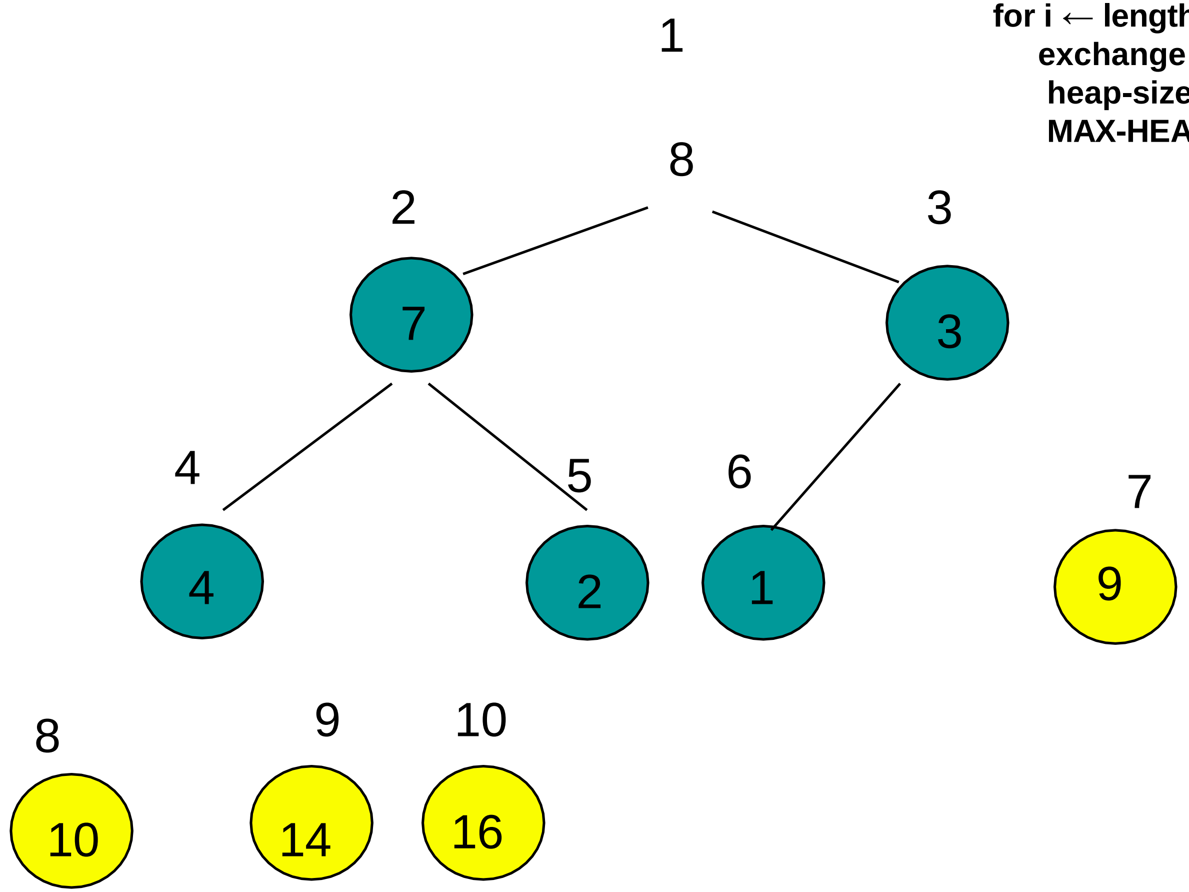
9	8	3	4	7	1	2	10	14	16
1	2	3	4	5	6	7	8	9	10

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



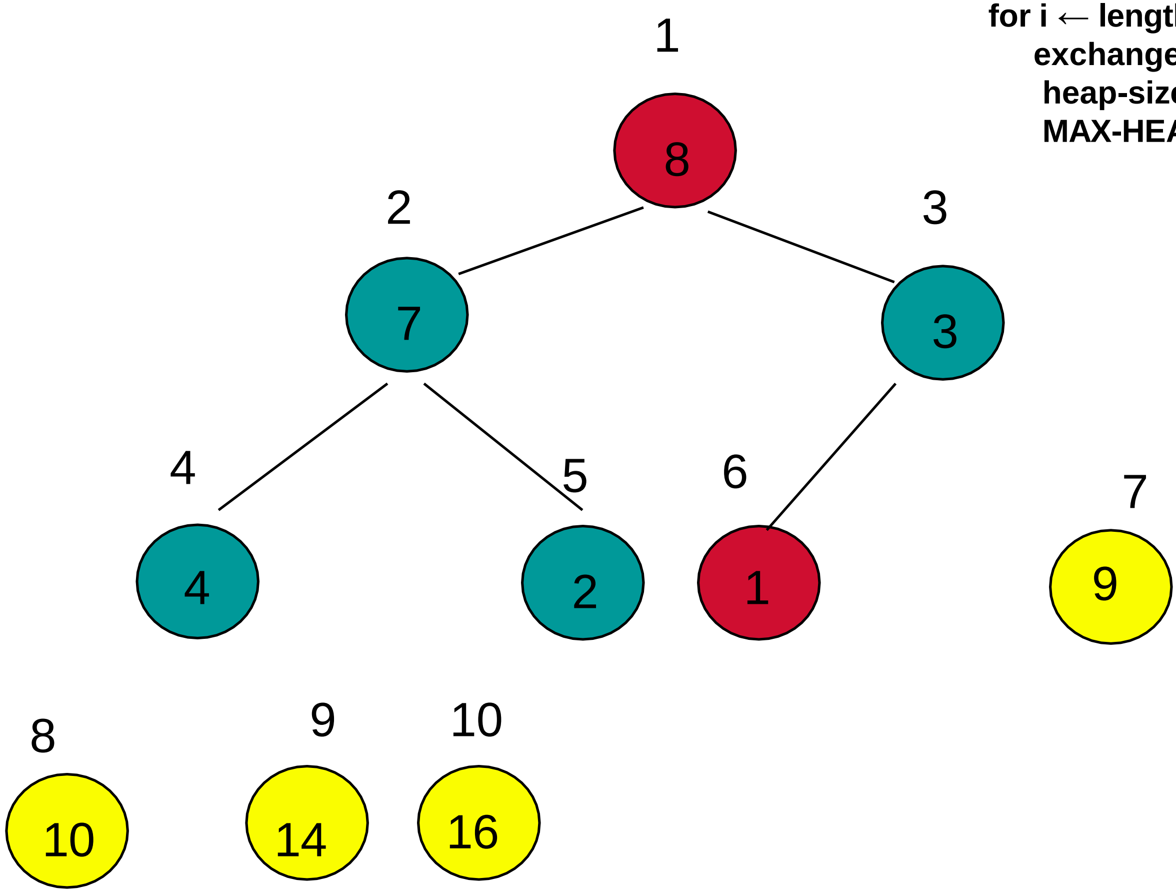
2	8	3	4	7	1	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



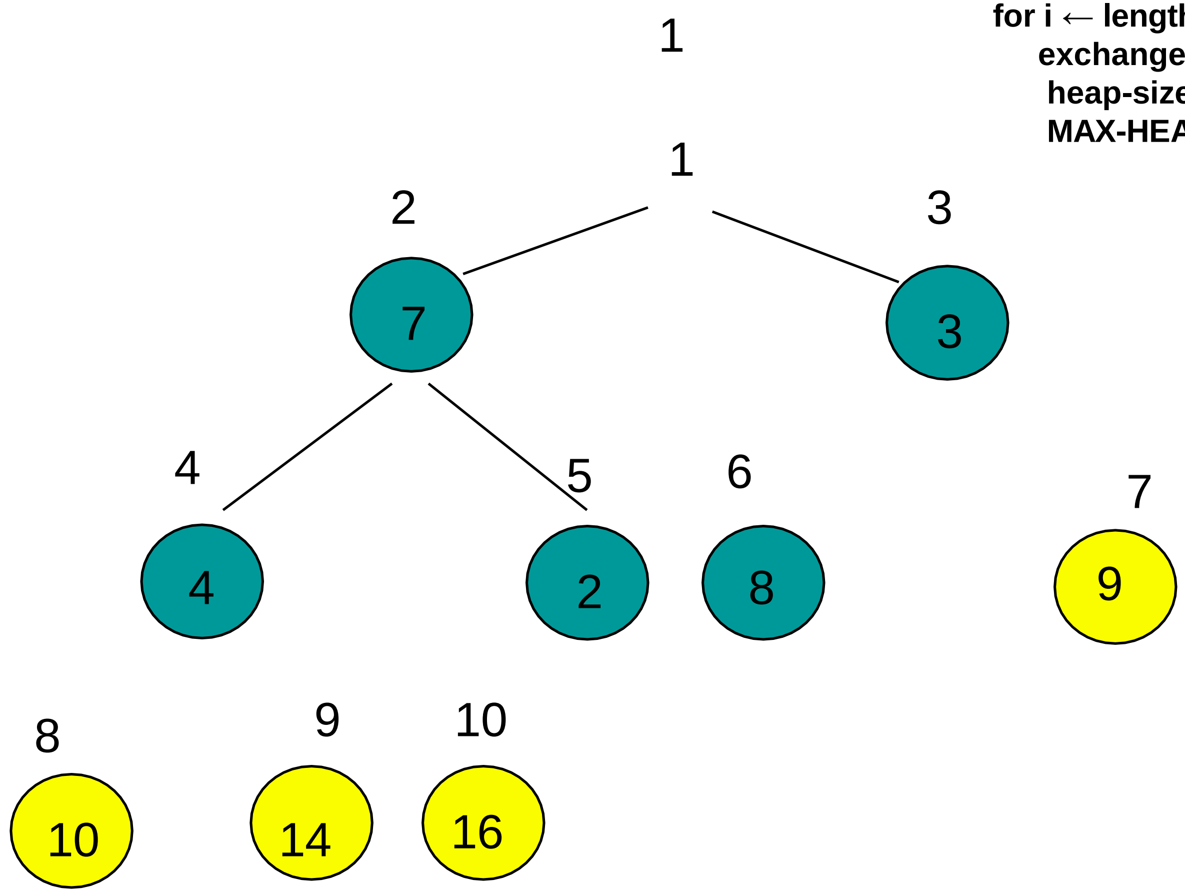
8	7	3	4	2	1	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



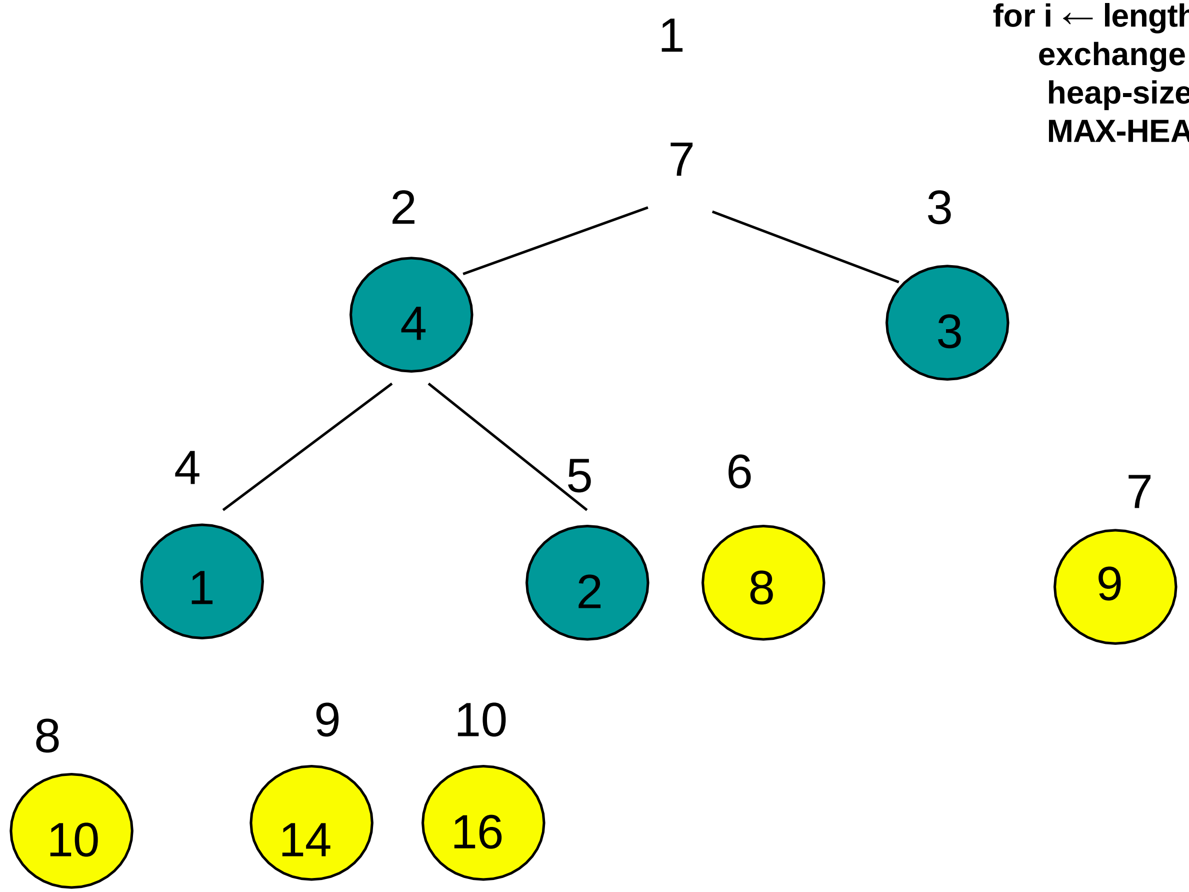
1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16

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



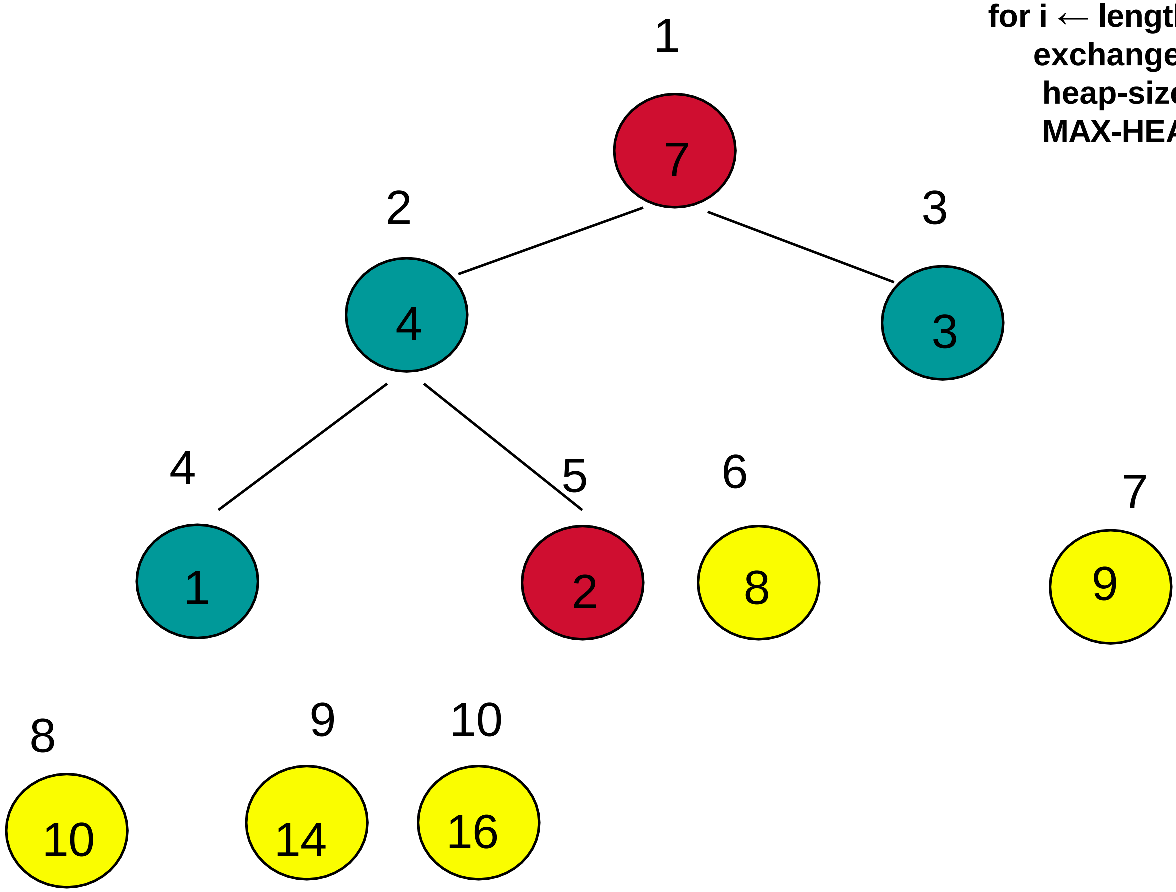
1	7	3	4	2	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



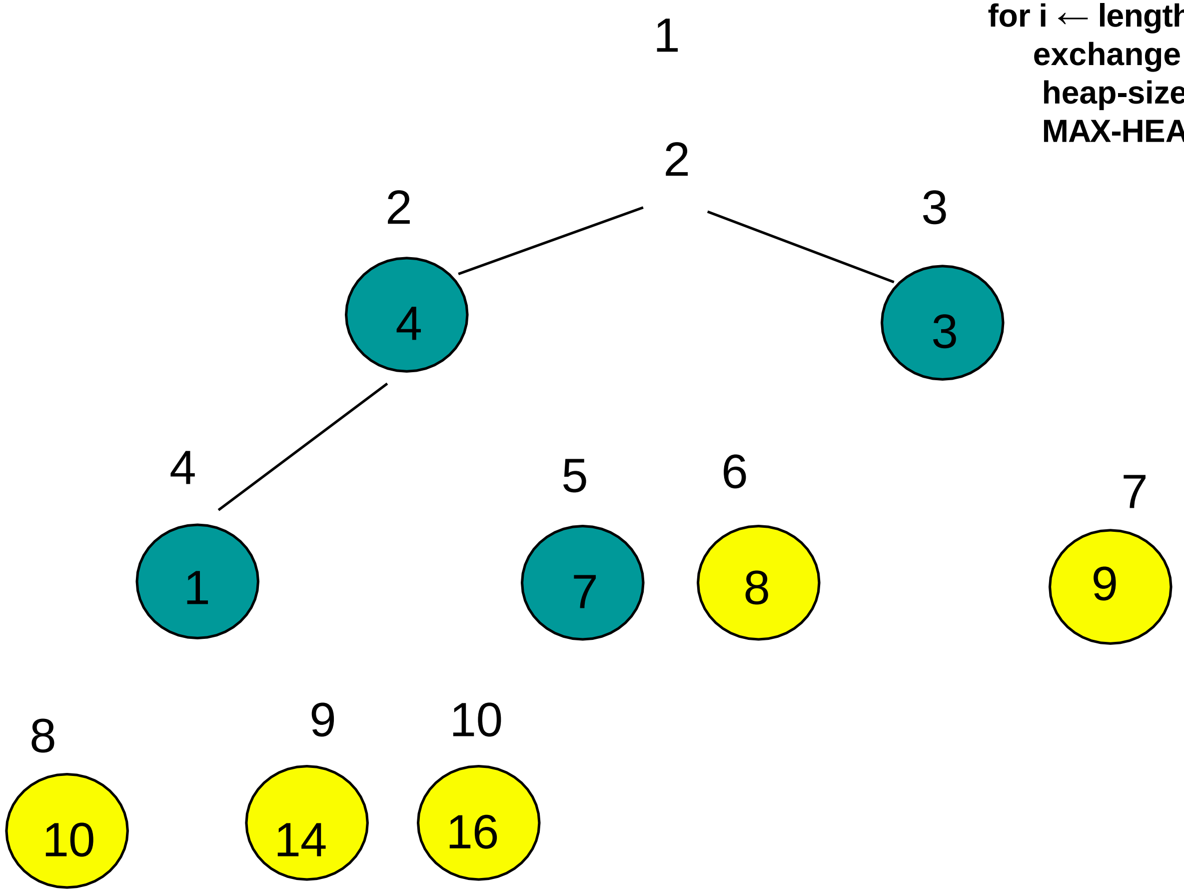
7	4	3	1	2	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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



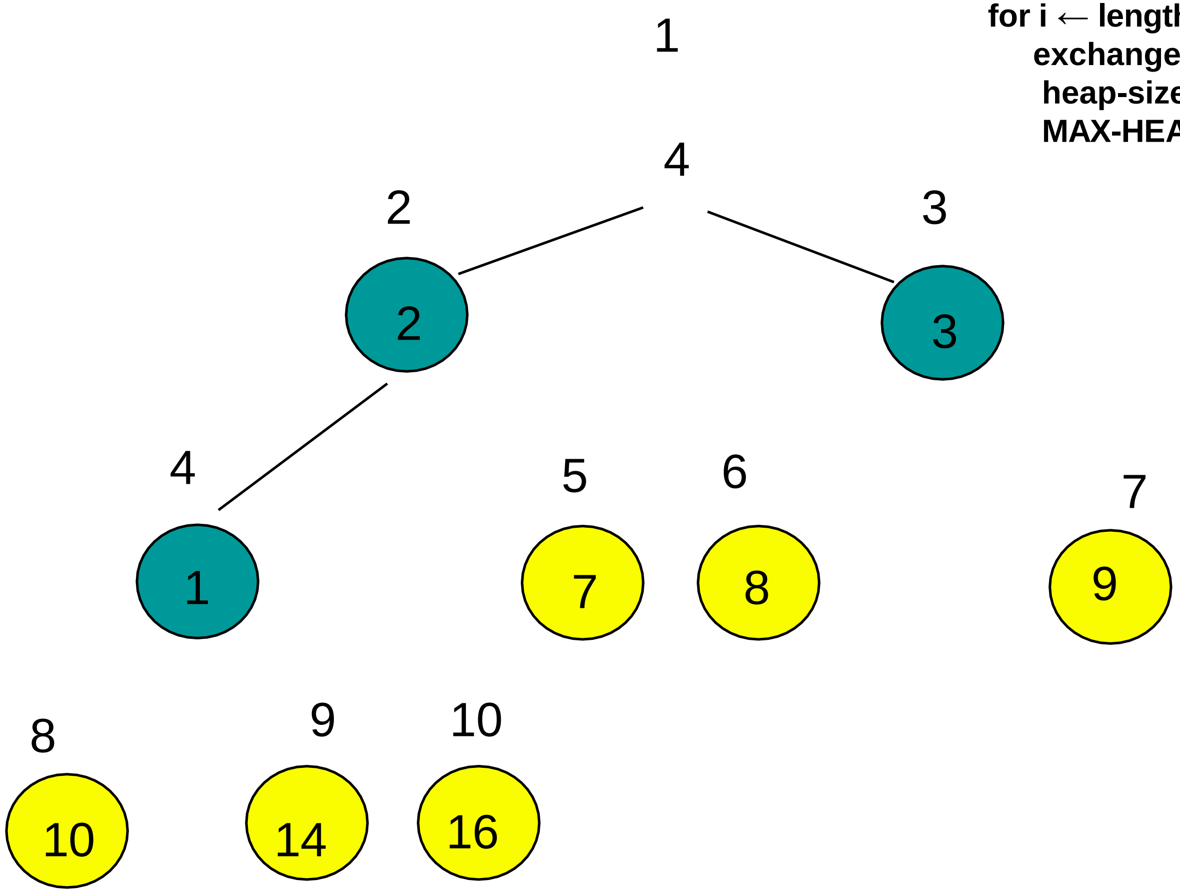
7	4	3	1	2	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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



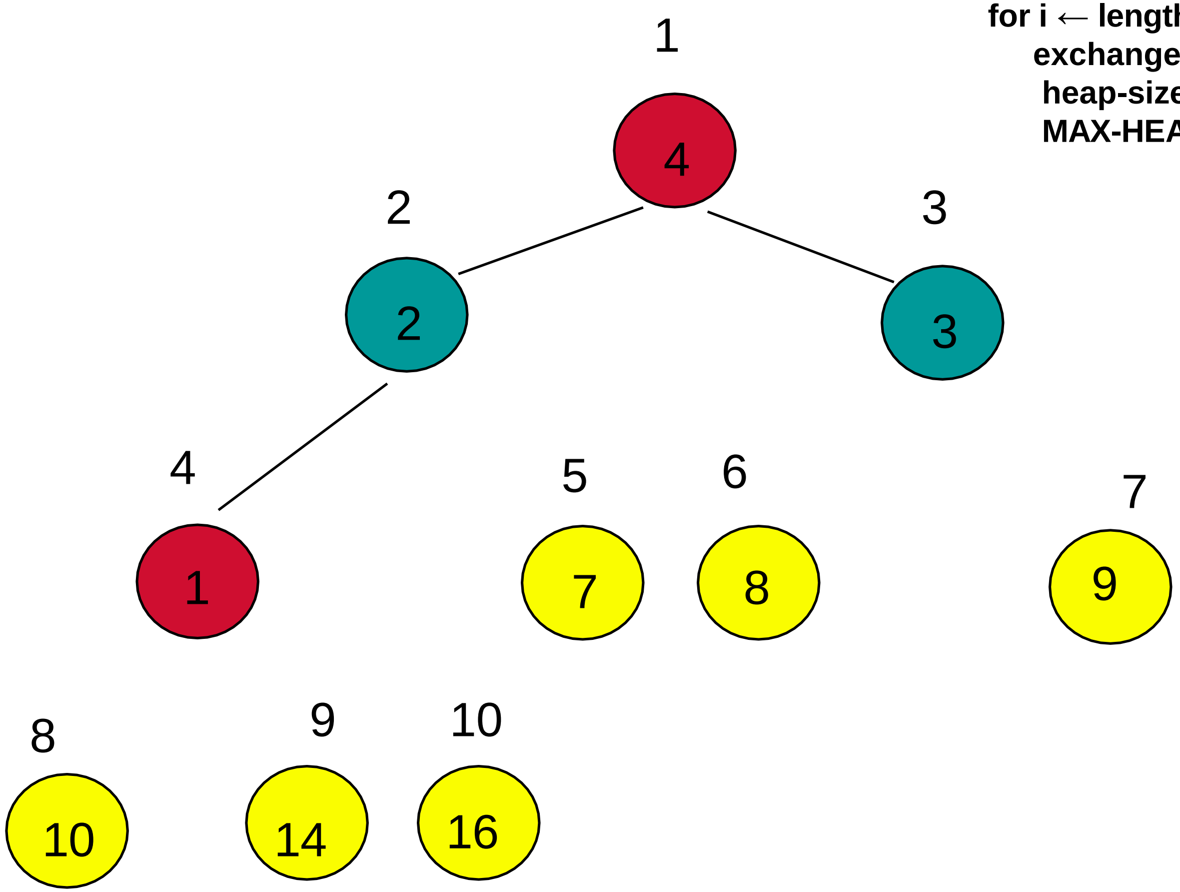
2	4	3	1	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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



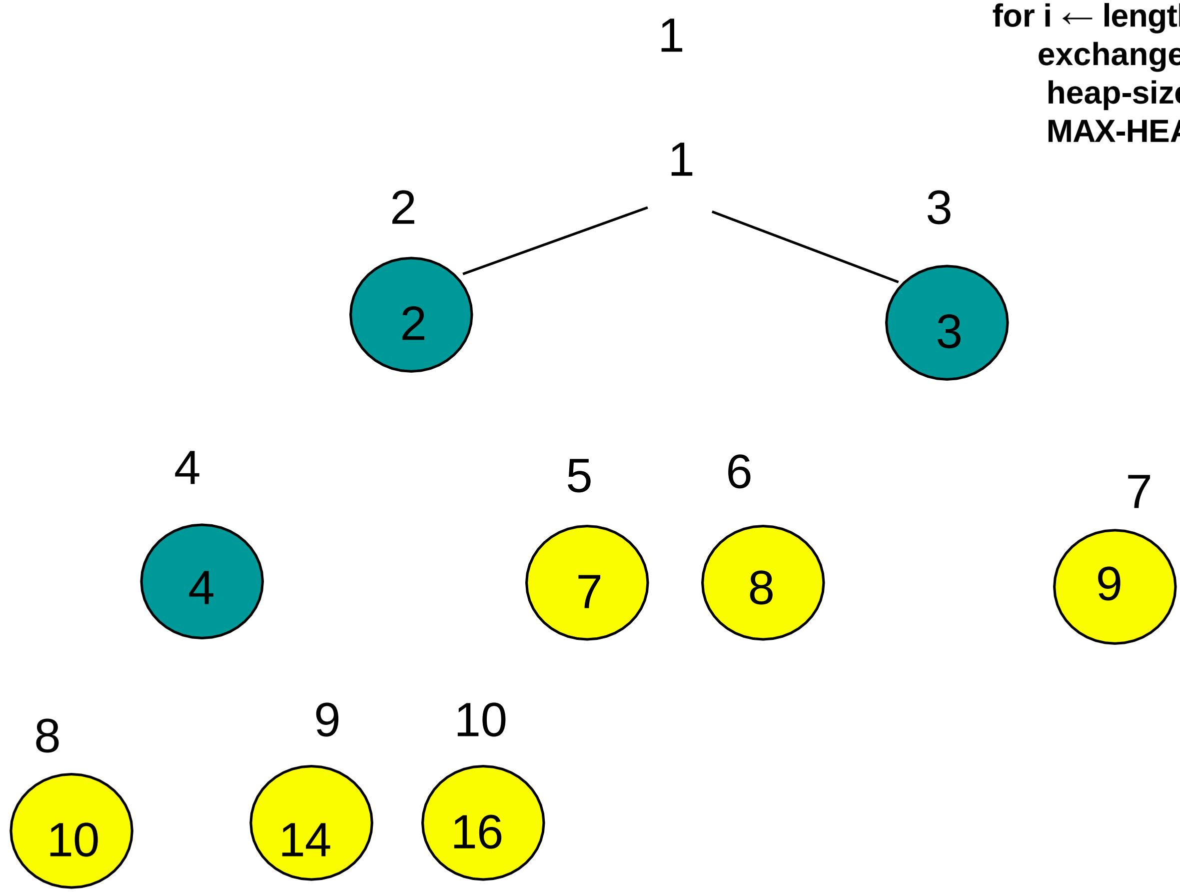
4	2	3	1	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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



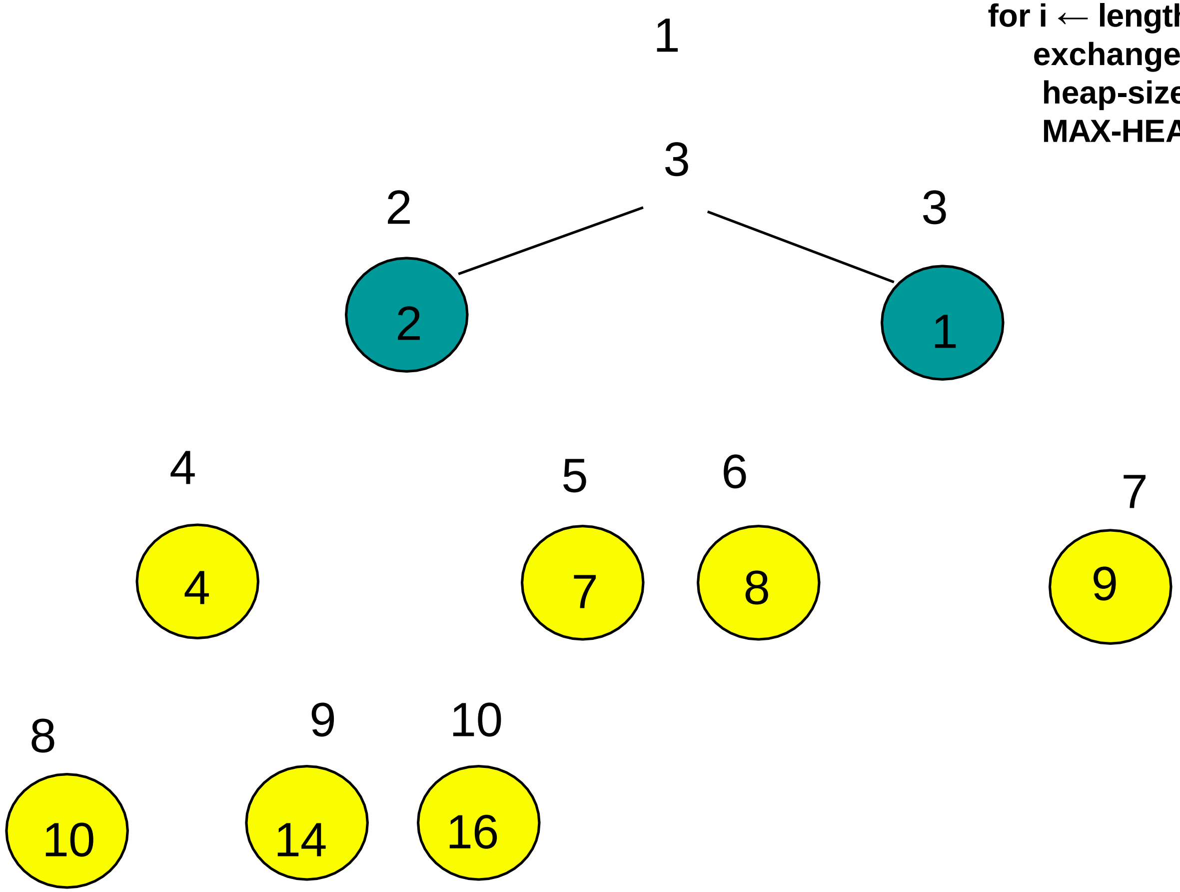
4	2	3	1	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



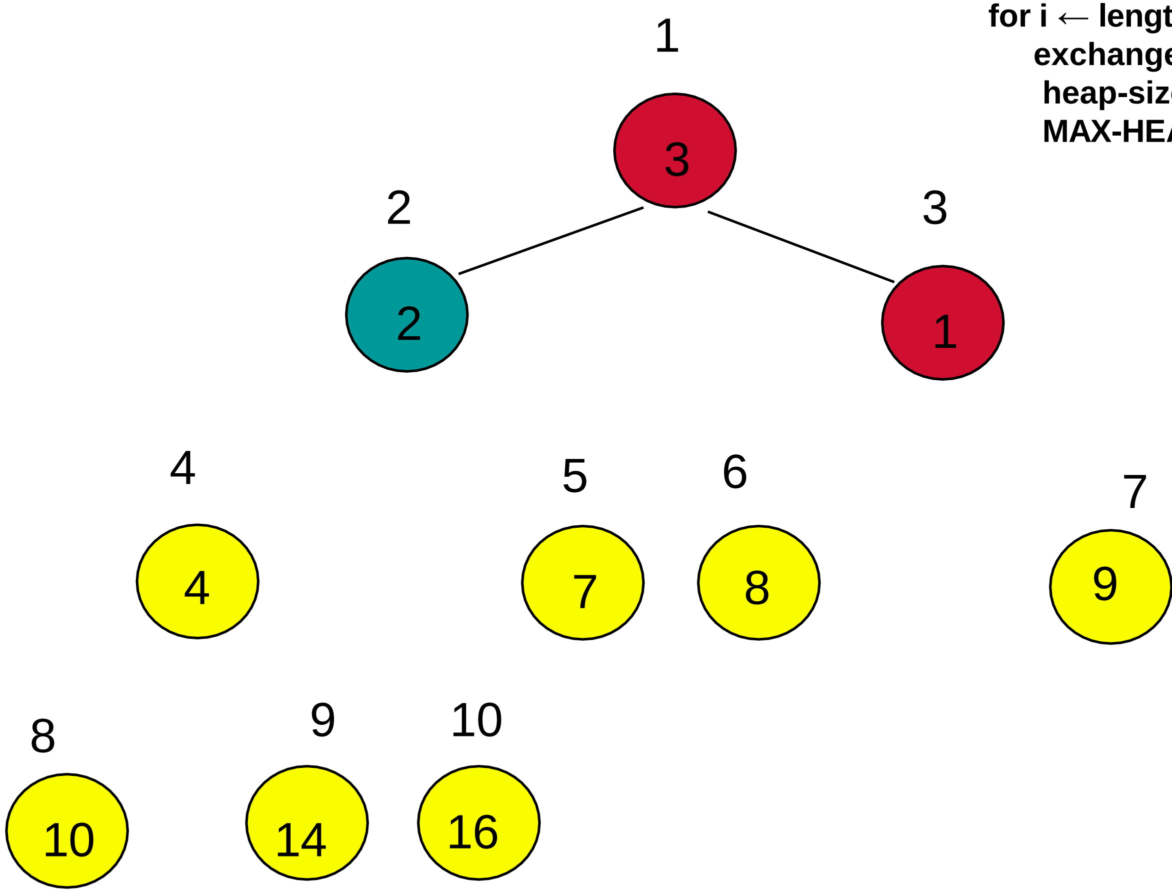
1	2	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



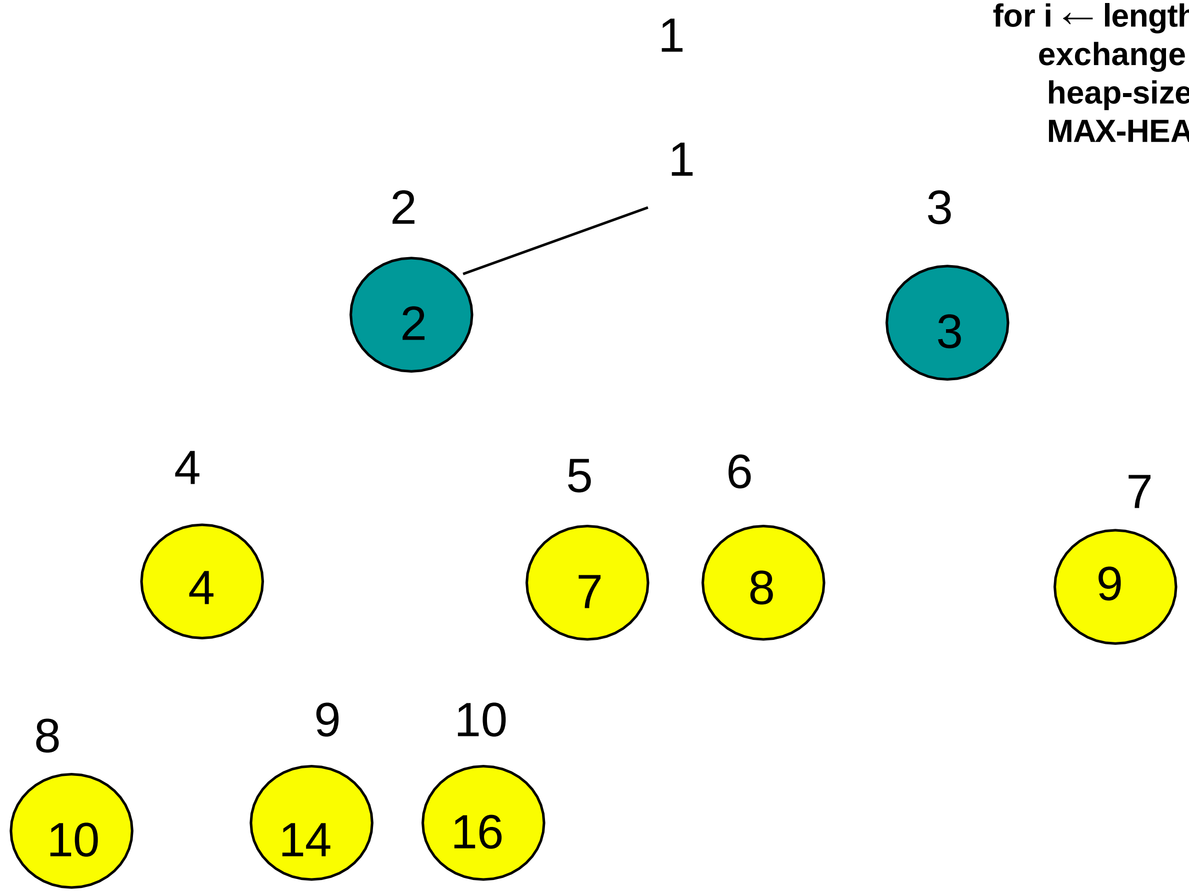
3	2	1	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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



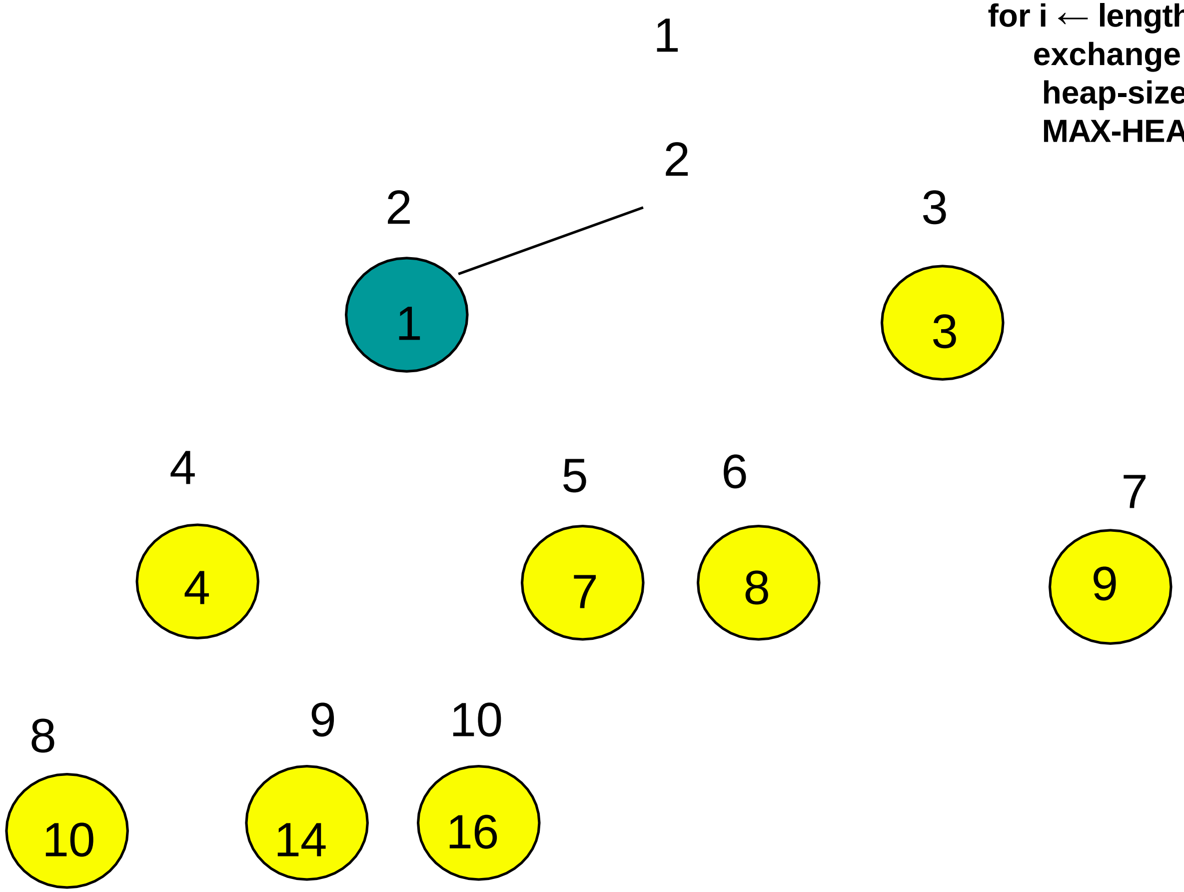
3	2	1	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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



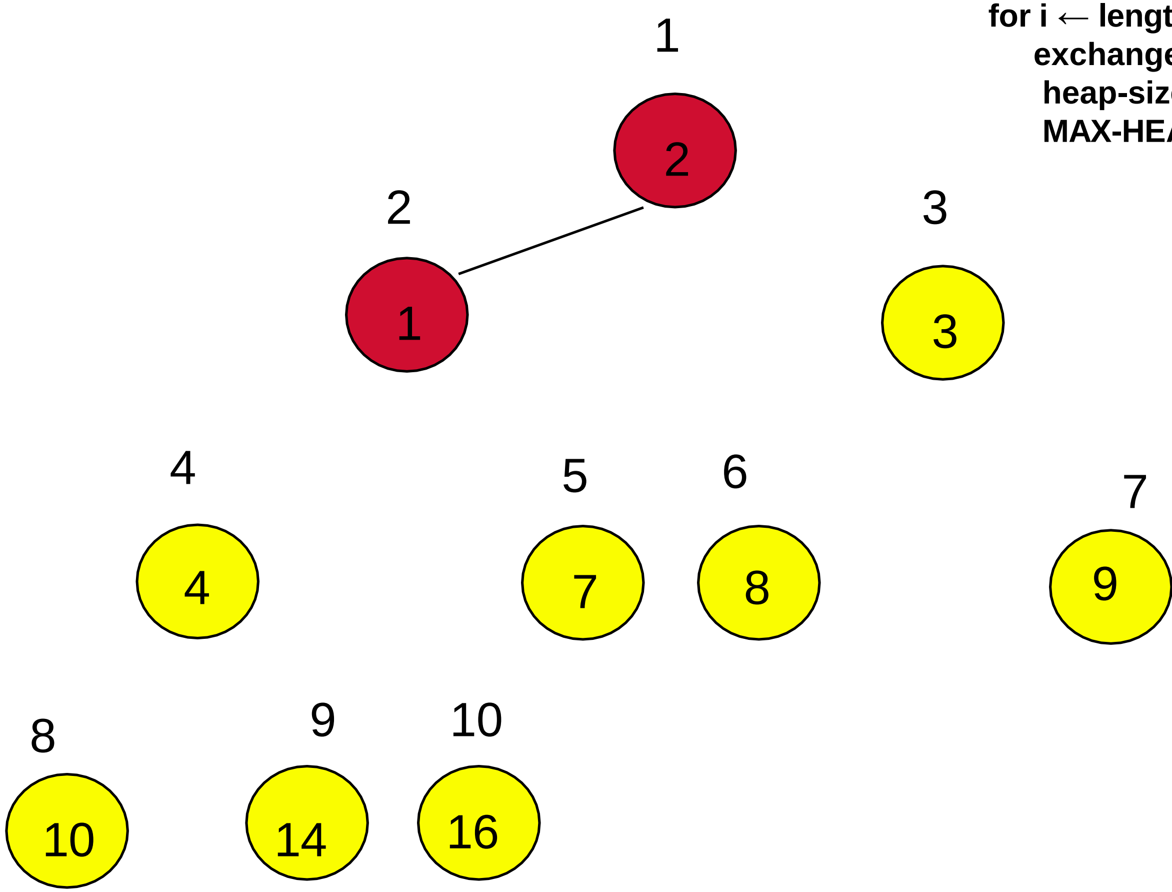
1	2	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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



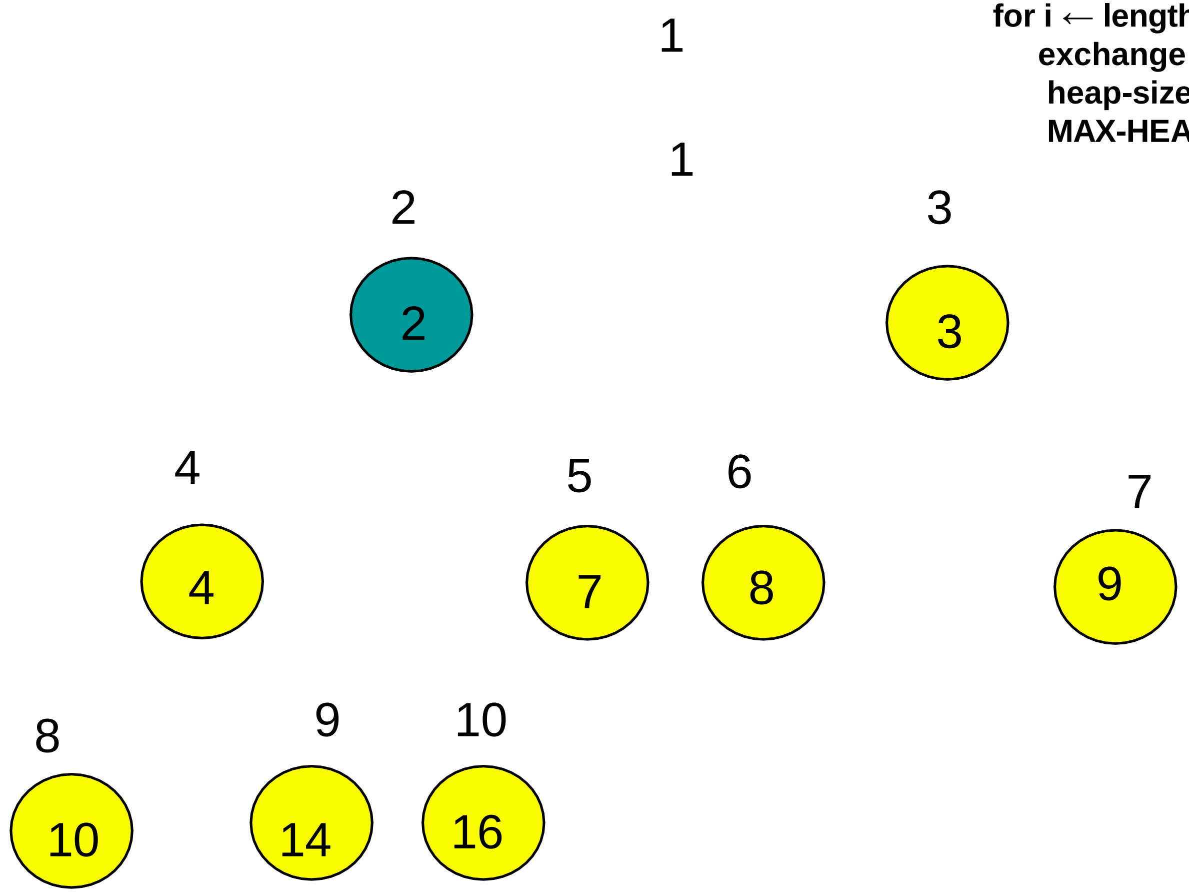
2	1	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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



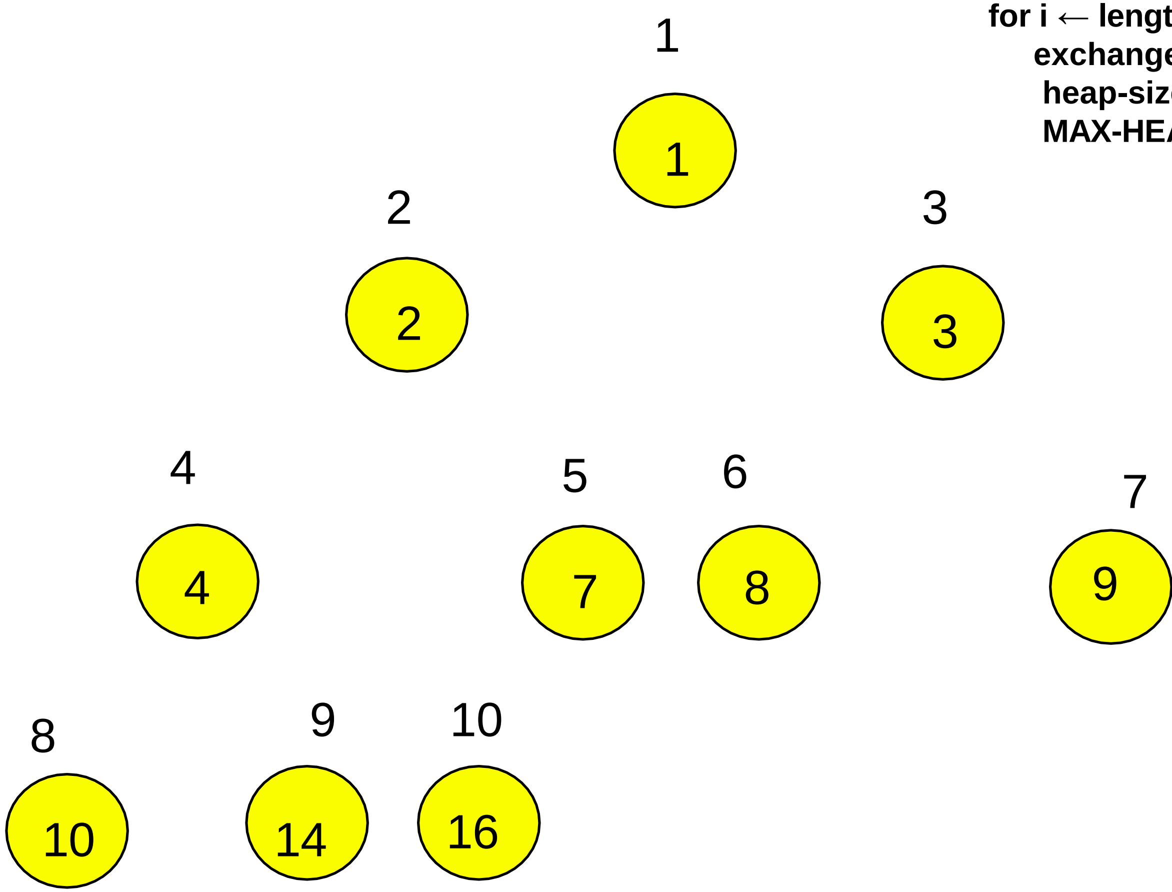
2	1	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



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

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



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

Running time of Heapsort

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)                              O(n)
  for i ← length[A] downto 2 do                  O(n-1)
    exchange A[1] ↔ A[i]                         O(1)
    heap-size[A] ← heap-size[A] - 1              O(1)
    MAX-HEAPIFY(A, 1)                            O(lg n)
```

Total time is:

$$O(n) + O(n\text{-}1) * [\ O(1) + O(1) + O(\lg n)\ ]$$

which is approximately

$$O(n) + O(n \lg n)$$

or just $\quad O(n \lg n)$

Alg: ***HEAPSORT(A)***

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

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

Reference

- **Introduction to Algorithms**
    - **Chapter # 6**
    - **Thomas H. Cormen**
    - **3^{rd} Edition**