Chapter 6
Heapsort

The slides for this course are based on the course textbook: Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms, 3rd edition, The MIT Press, McGraw-Hill, 2010.

Many of the slides were provided by the publisher for use with the textbook. They are copyrighted, 2010.

These slides are for classroom use only, and may be used only by students in this specific course this semester. They are NOT a substitute for reading the textbook!

Chapter 6 Topics

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

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

Full and Complete Binary Trees

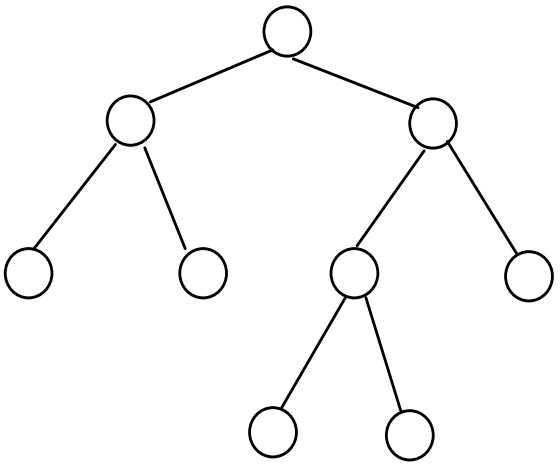
A **full binary tree** is a binary tree in which each node is either a leaf node or has degree 2 (i.e., has exactly 2 children).

A **complete binary tree** is a full binary tree in which all leaves have the same depth.

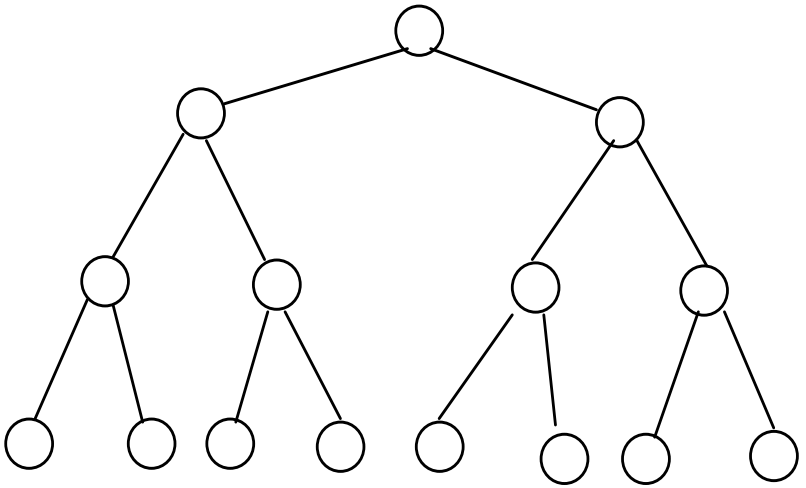
A **nearly complete binary tree** is completely filled on all levels except possibly the lowest, which is filled *from the left* up to a point.

Examples

Full binary tree:



Complete binary tree:



Representation of Nearly Complete Binary Tree

A nearly complete binary tree may be represented as an array (i.e., no pointers):

Number the nodes, beginning with the root node and moving from level to level, left to right within a level.

The number assigned to a node is its index in the array.

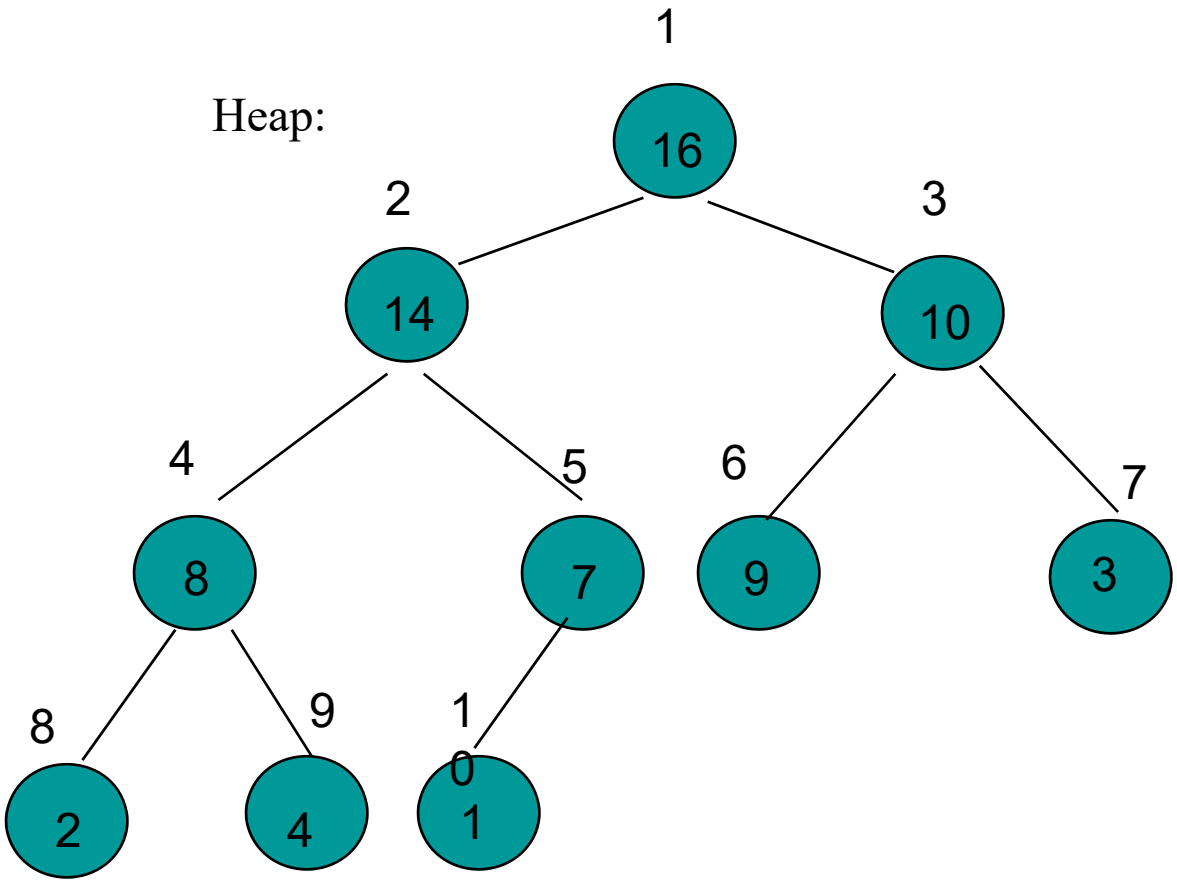
Additional Properties of Nearly Complete Binary Trees

- The root of the tree is A[1].
- If a node has index i, we can easily compute the indices of its:
  - parent $\lfloor i/2 \rfloor$
  - left child $2i$
  - right child $2i + 1$

Numbering

Array:

16	14	10	8	7	9	3	2	4	1



Heap

- Implemented as an array object, A[ ]
- Array A that implements the heap has two attributes
  - length(A)
  - heap-size(A)

Heap

A binary tree with n nodes and of height *h* is **almost complete** iff its nodes correspond to the nodes which are numbered 1 to *n* in the complete binary tree of height *h*.

A **heap** is an *almost complete binary tree* that satisfies the **heap property**:

max-heap: For every node *i* other than the root:

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

min-heap: For every node i other than the root:

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

Max-Heap

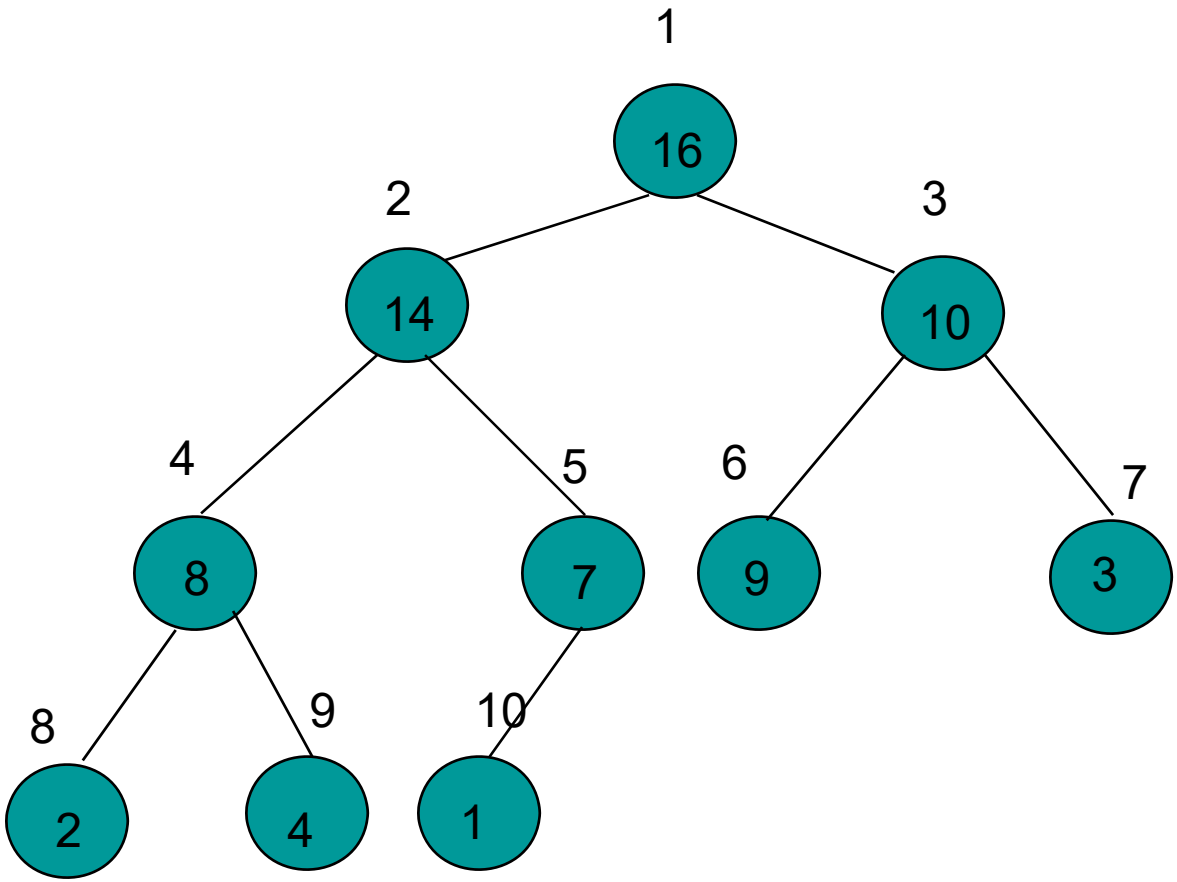
A **max-heap** is an *almost complete binary tree* that satisfies the **heap property**:

For every node i other than the root,

$$A[PARENT(i)] \geq A[i]$$

What does this mean?

- the value of a node is at most the value of its parent
- the largest element in the heap is stored in the root
- subtrees rooted at a node contain smaller values than the node itself



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

Height of a node in 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.

The height of a heap is the height of its root.

Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$.

Heaps have 5 basic procedures

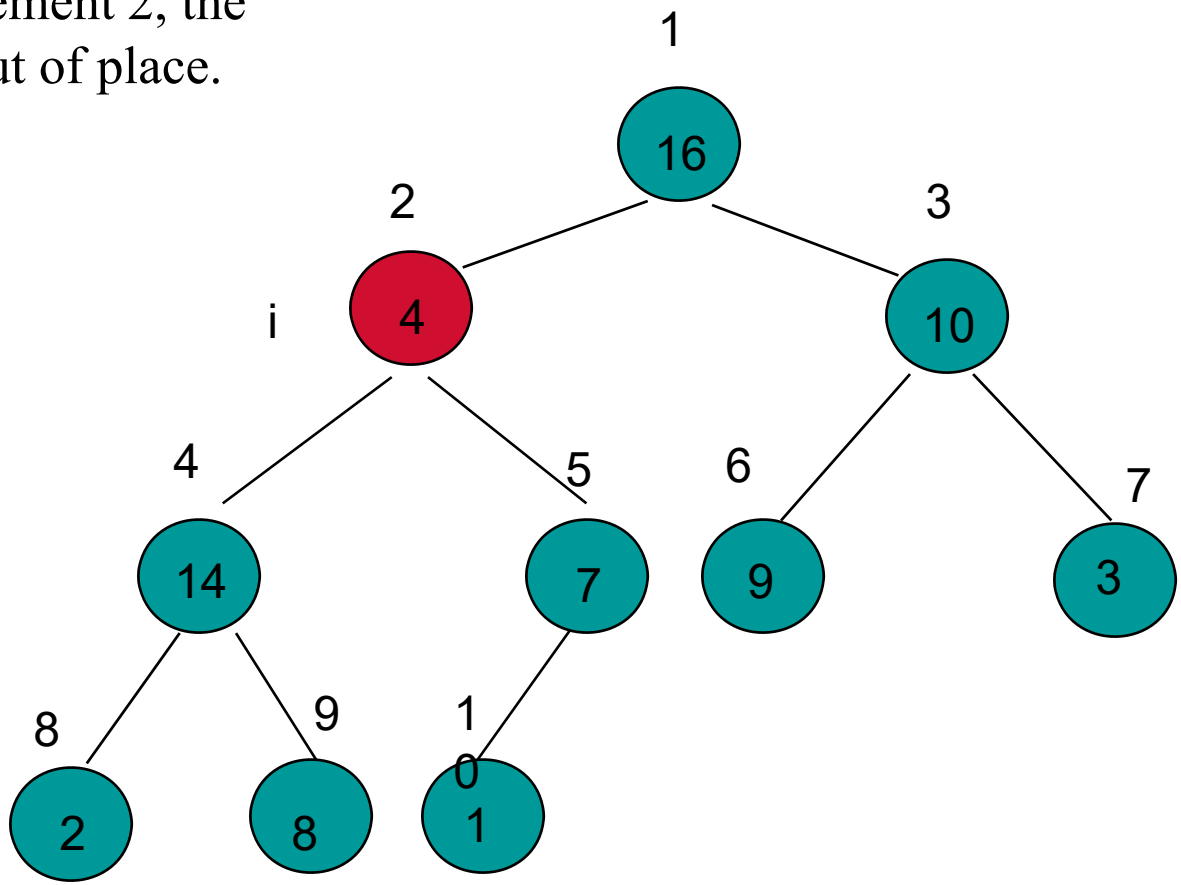
- HEAPIFY: maintains the heap property
- BUILD-HEAP: builds a heap from an unordered array
- HEAPSORT: sorts an array in place
- EXTRACT-MAX: selects max element
- INSERT: inserts a new element

We'll work with MAX heaps

MAX-HEAPIFY(A,i)

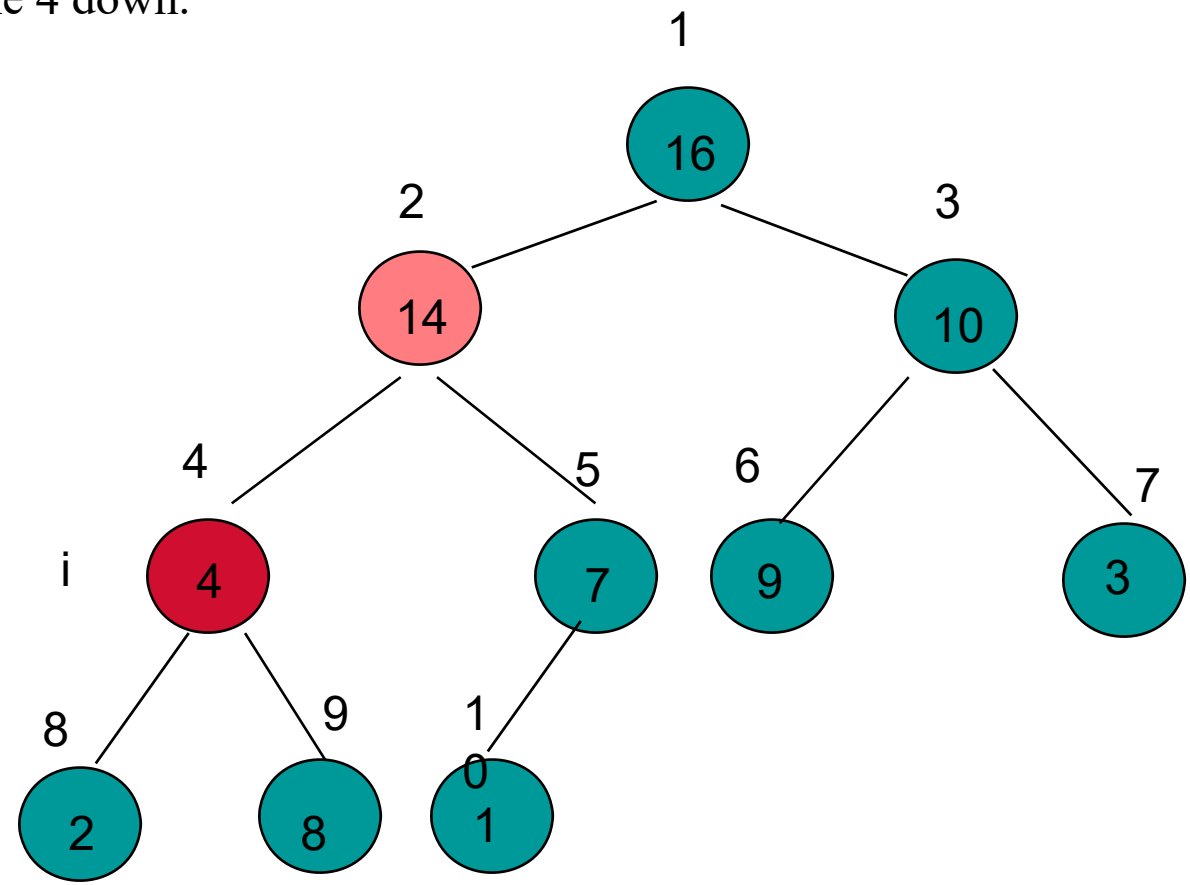
- Goal is to put the i^{th} element in the correct place in a portion of the array that "almost" has the heap property.
- The only element with index of *i* or greater that is out of place is A[*i*].
- Assume that left and right subtrees of A[*i*] have the heap property.
- "Sift" A[*i*] down to the right position.

Array element 2, the "4", is out of place.



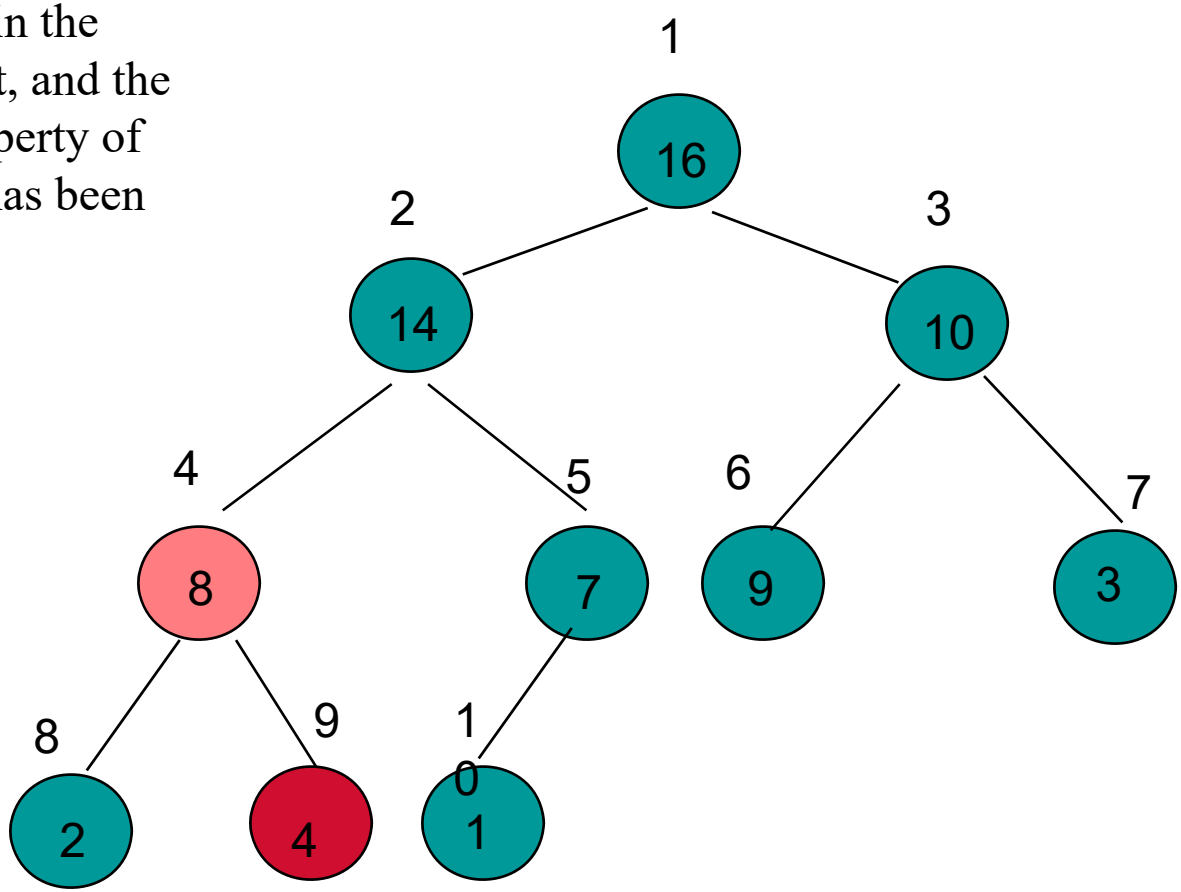
MAX-HEAPIFY(A,2) heap-size[A] = 10

Moving the 4 down.



MAX-HEAPIFY(A,4) heap-size[A] = 10

The 4 is in the right spot, and the heap property of the tree has been restored.



MAX-HEAPIFY(A,9) heap-size[A] = 10

MAX-HEAPIFY

```
MAX-HEAPIFY(A, i)
1   l ← LEFT(i)
2   r ← RIGHT(i)
3   if l ≤ heap-size[A] and A[l] > A[i]
4        then largest ← l
5        else largest ← i
6   if r ≤ heap-size[A] and A[r] > A[largest]
7        then largest ← r
8   if largest ≠ i
9        then exchange A[i] ↔ A[largest]
10             MAX-HEAPIFY(A, largest)
```

Running time of MAX-HEAPIFY

The recursive call to MAX-HEAPIFY in line 10 implies a recurrence relation.

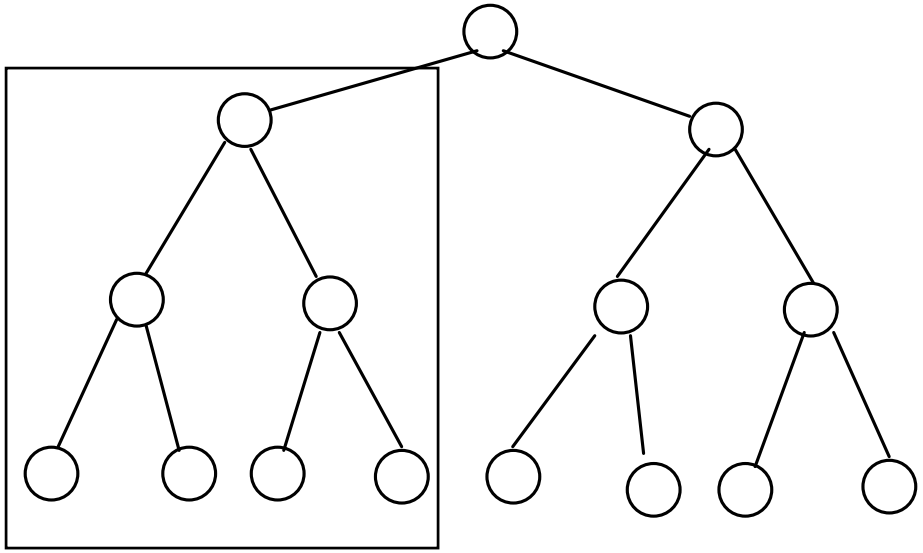
Lines 1-9 cost $\Theta(1)$ steps.

But we may need to call MAX-HEAPIFY on a subtree rooted at one of the children of the current node, so we have to add the cost of doing that.

Running time of MAX-HEAPIFY

How many nodes might be involved?

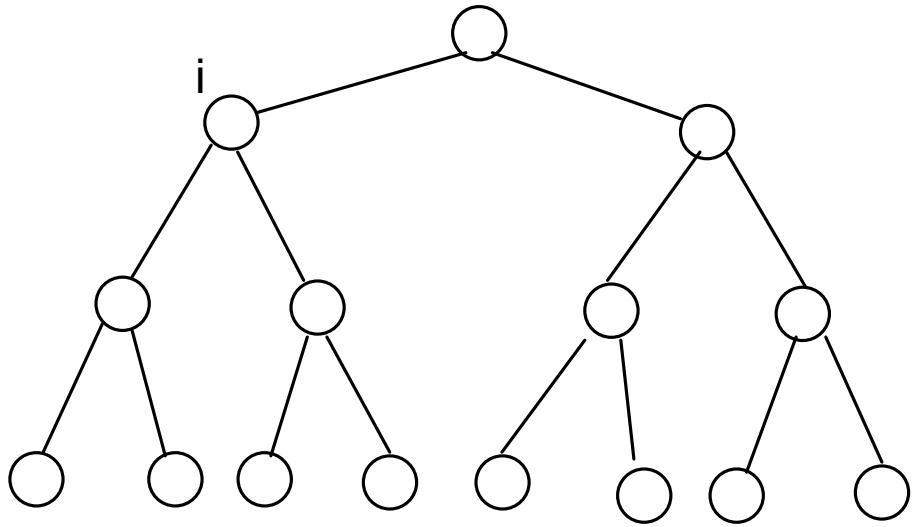
In the case of a full binary tree, about half of the tree might be involved.



Running time of MAX-HEAPIFY

In a complete binary tree with 15 nodes, 8 of those nodes are leaves at the bottom level.

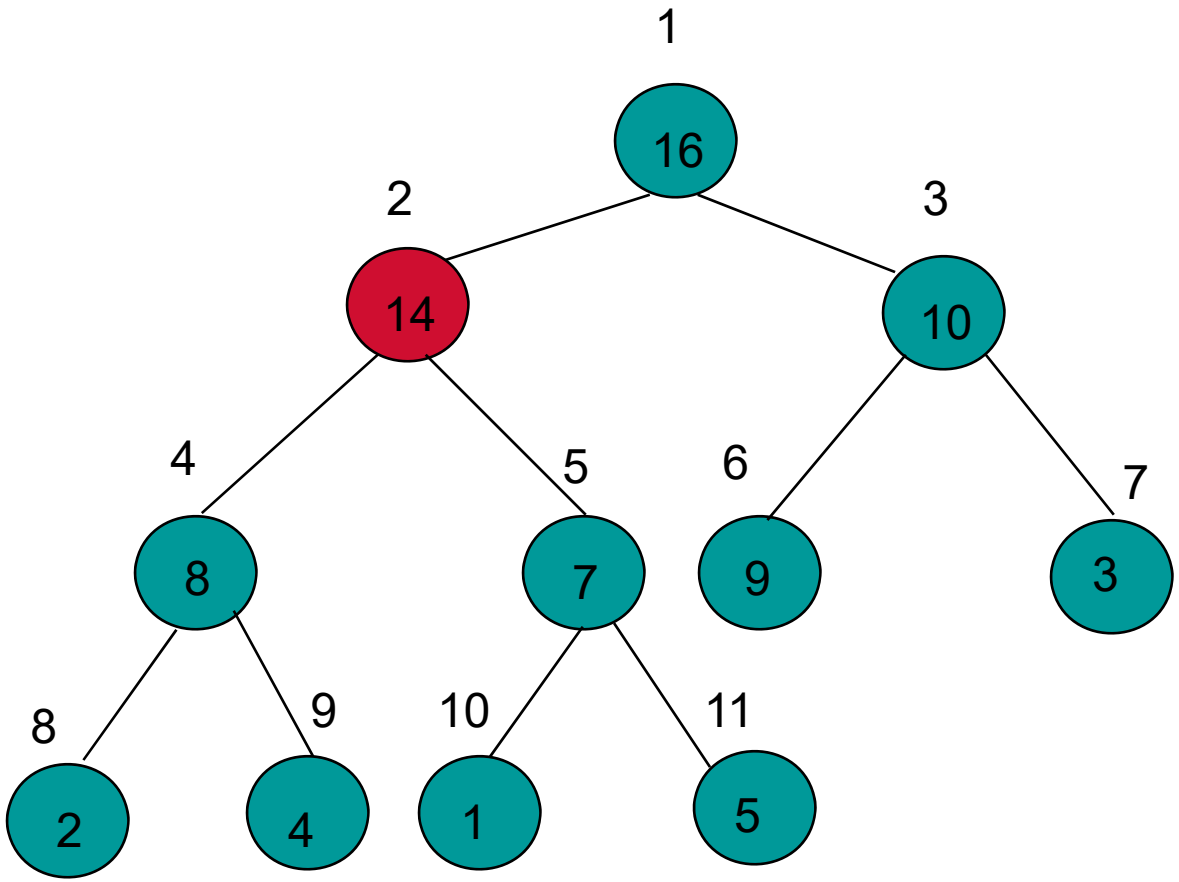
If we perform MAX-HEAPIFY on node i, 7 of the 15 nodes will be involved – about ½ of the nodes.



Running time of MAX-HEAPIFY

What is the worst case?

When the last row of the tree is half full.



Here 7 out of 11 nodes are involved.

In general, $\leq 2/3^{rds}$ of the tree might be involved in the worst case.

Running time of MAX-HEAPIFY

Remember that, in a complete binary tree, *more than half* of the nodes in the entire tree are the leaf nodes on the bottom level of the tree.

But the only nodes involved in MAX-HEAPIFY are the descendants of A[i], which must be in A[i]'s half of the tree.

So worst case is when the last row of the tree is half full on the left side and A[i] is their ancestor.

Running time of MAX-HEAPIFY

The subtrees of the children of our current node have size at most $2n/3$.

The running time of MAX_HEAPIFY can be described by the recurrence:

$$T(n) \leq T(2n/3) + \Theta(1)$$

This is Case 2 by the master method, so:

$$T(n) = O(\lg n)$$

Running time of MAX-HEAPIFY

We could also describe the running time of MAX-HEAPIFY for a node of height h as $O(h)$. (This is useful only if we know the height of a specific node.)

BUILD-MAX-HEAP

- Use MAX-HEAPIFY in a bottom-up manner to convert an array A[1..n] into a heap.
- Each leaf is initially a one-element heap. Elements A[$\lfloor n/2 \rfloor + 1..n$] are leaves.
- MAX-HEAPIFY is called on all interior nodes.

BUILD-MAX-HEAP

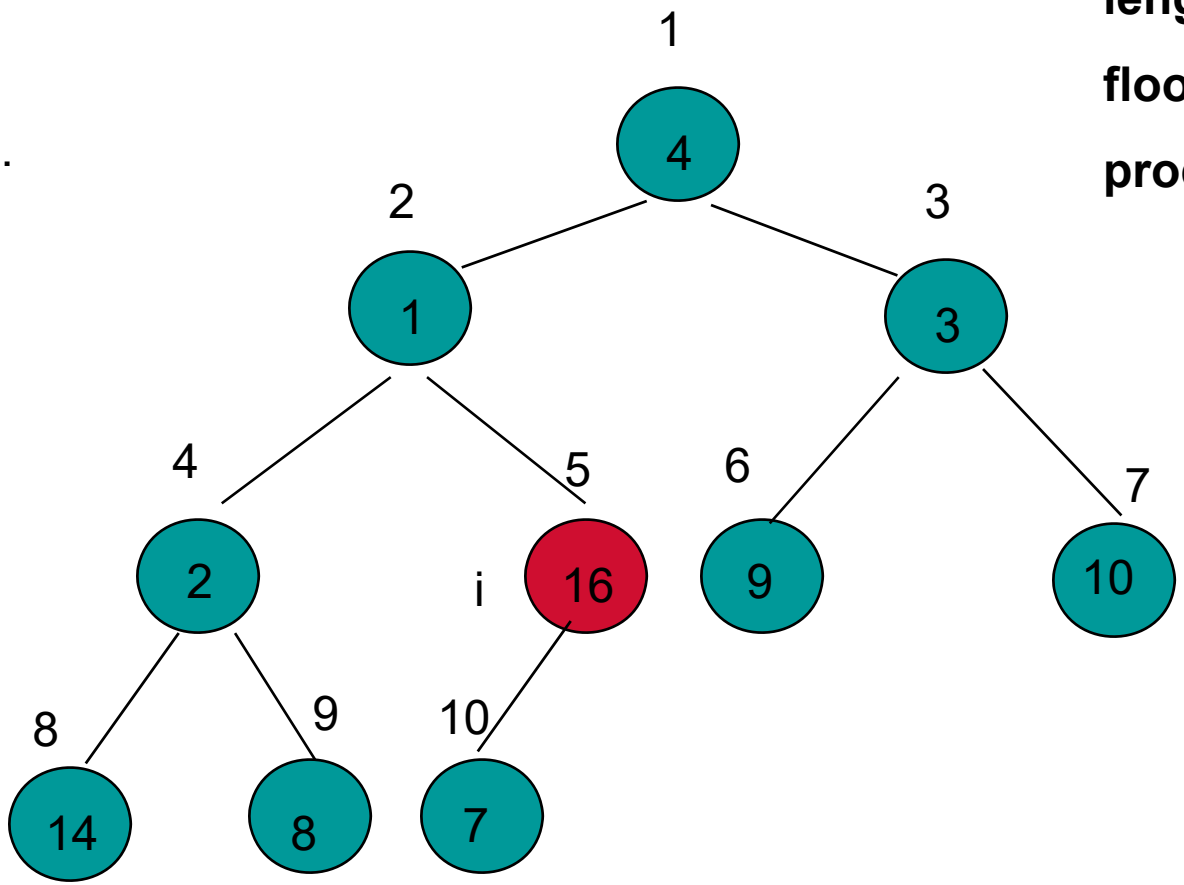
```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← floor(length[A]/2) downto 1 do
3      MAX-HEAPIFY(A, i)
```

length(A) = 10

floor(length(A)/2) = 5

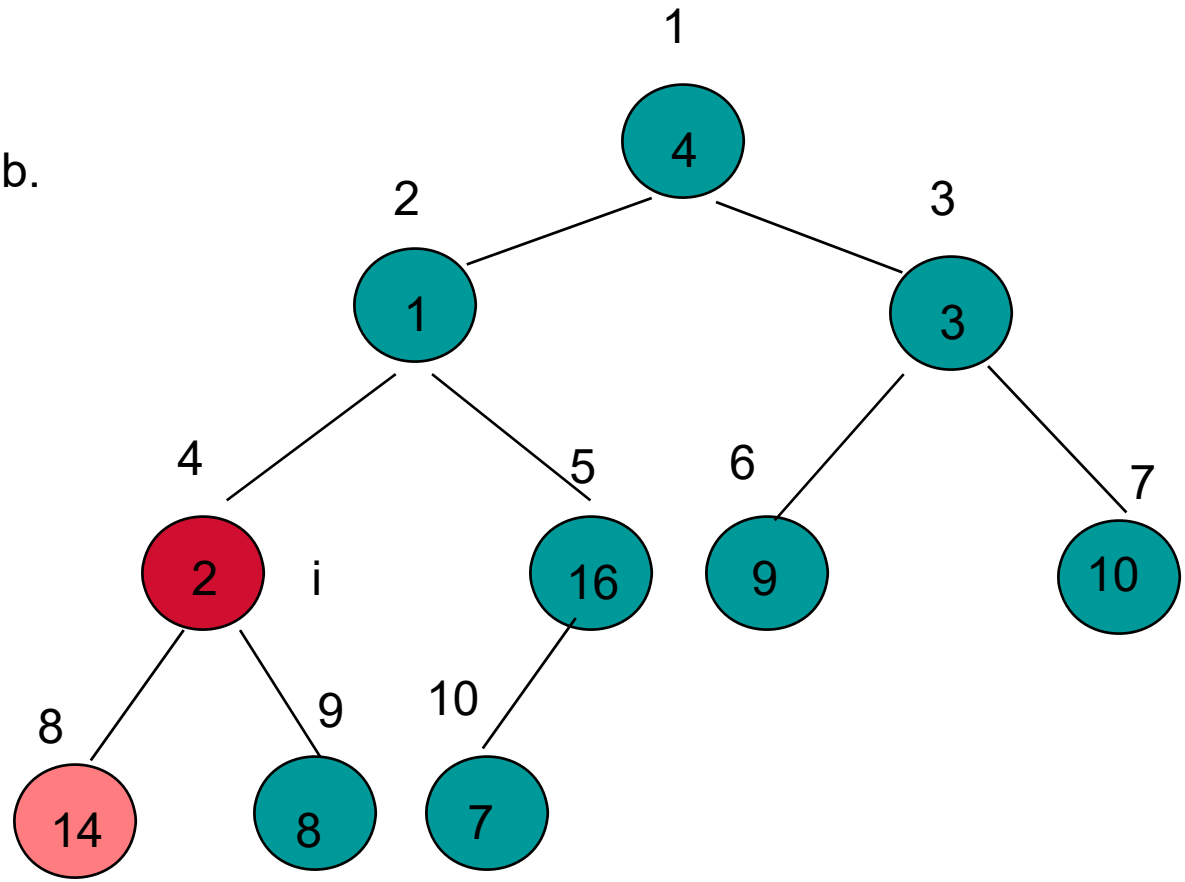
process from 5 to 1

a.

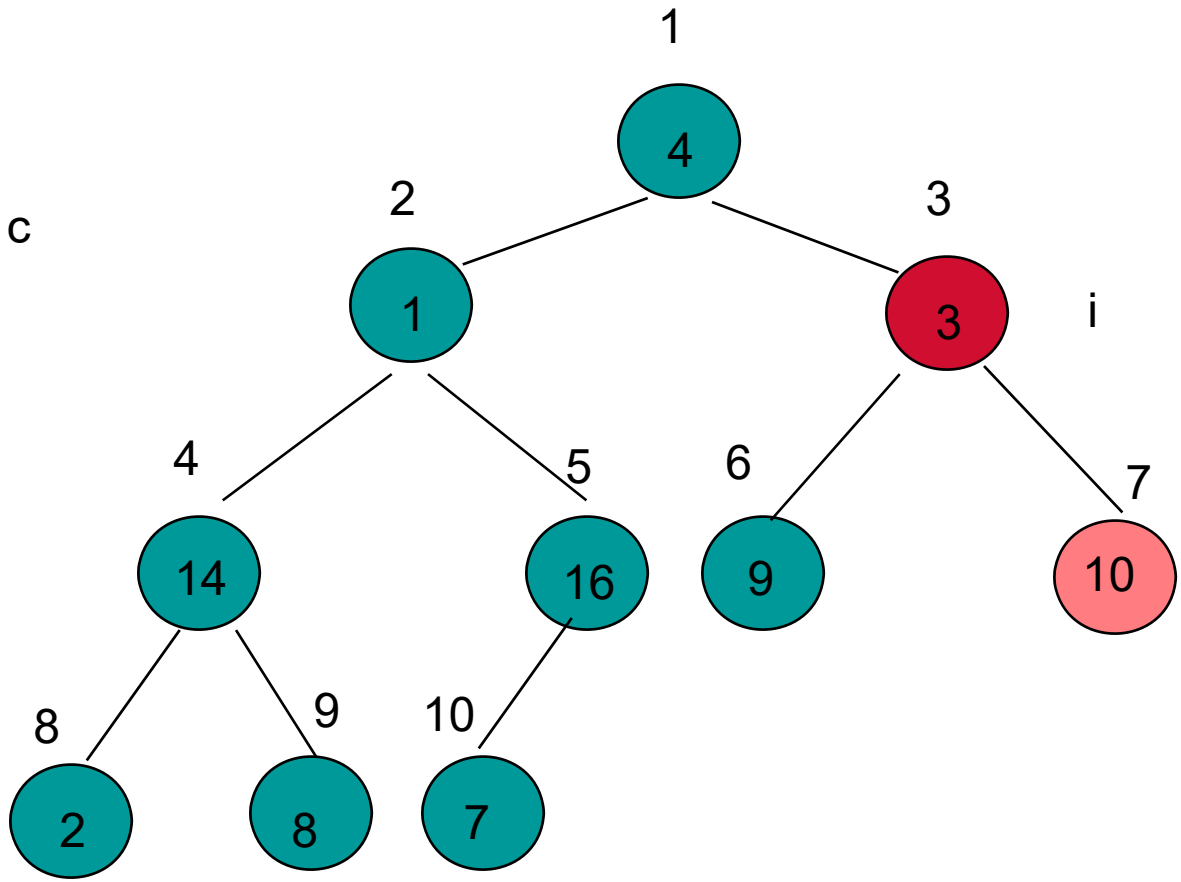


4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

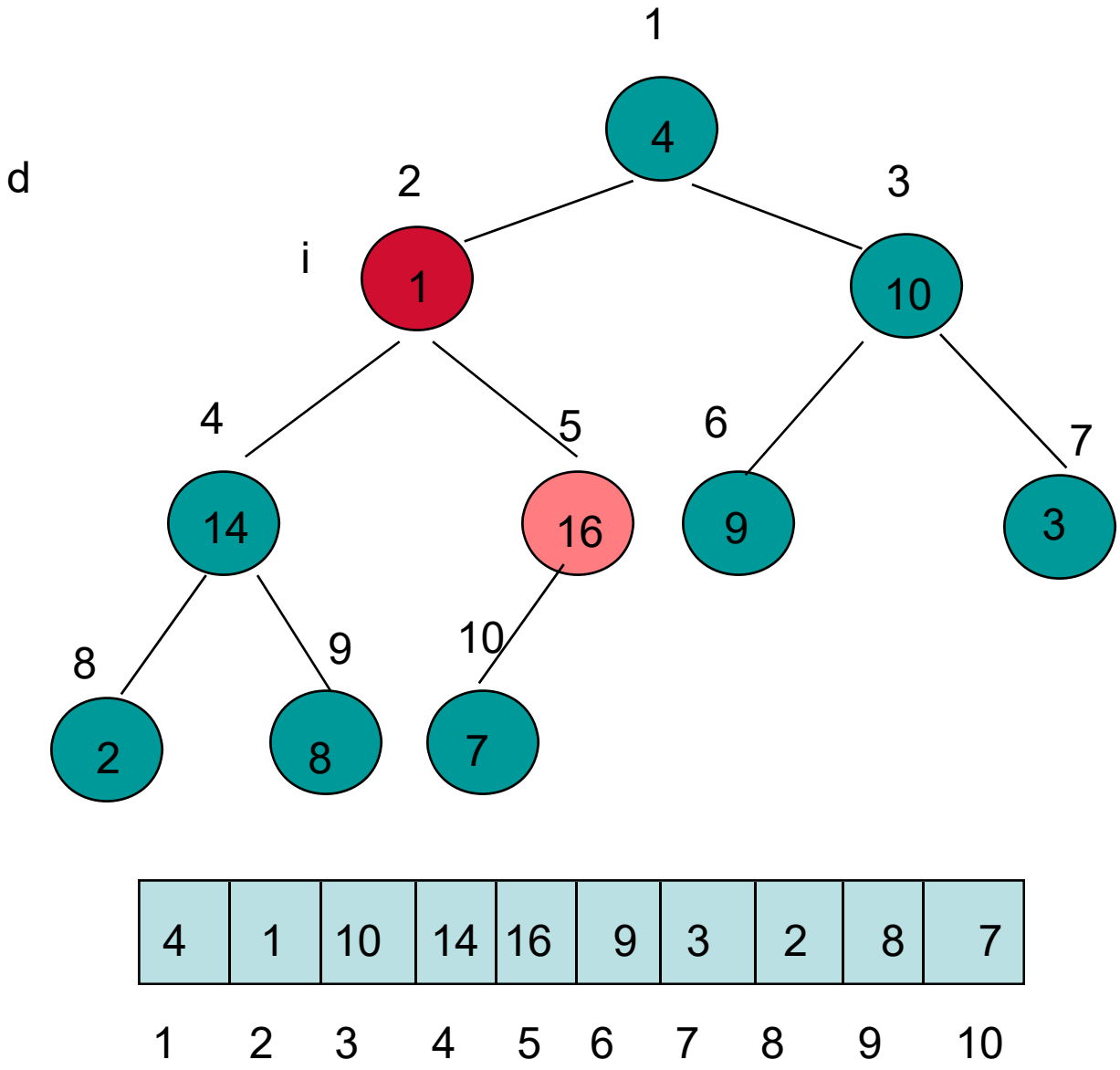
b.

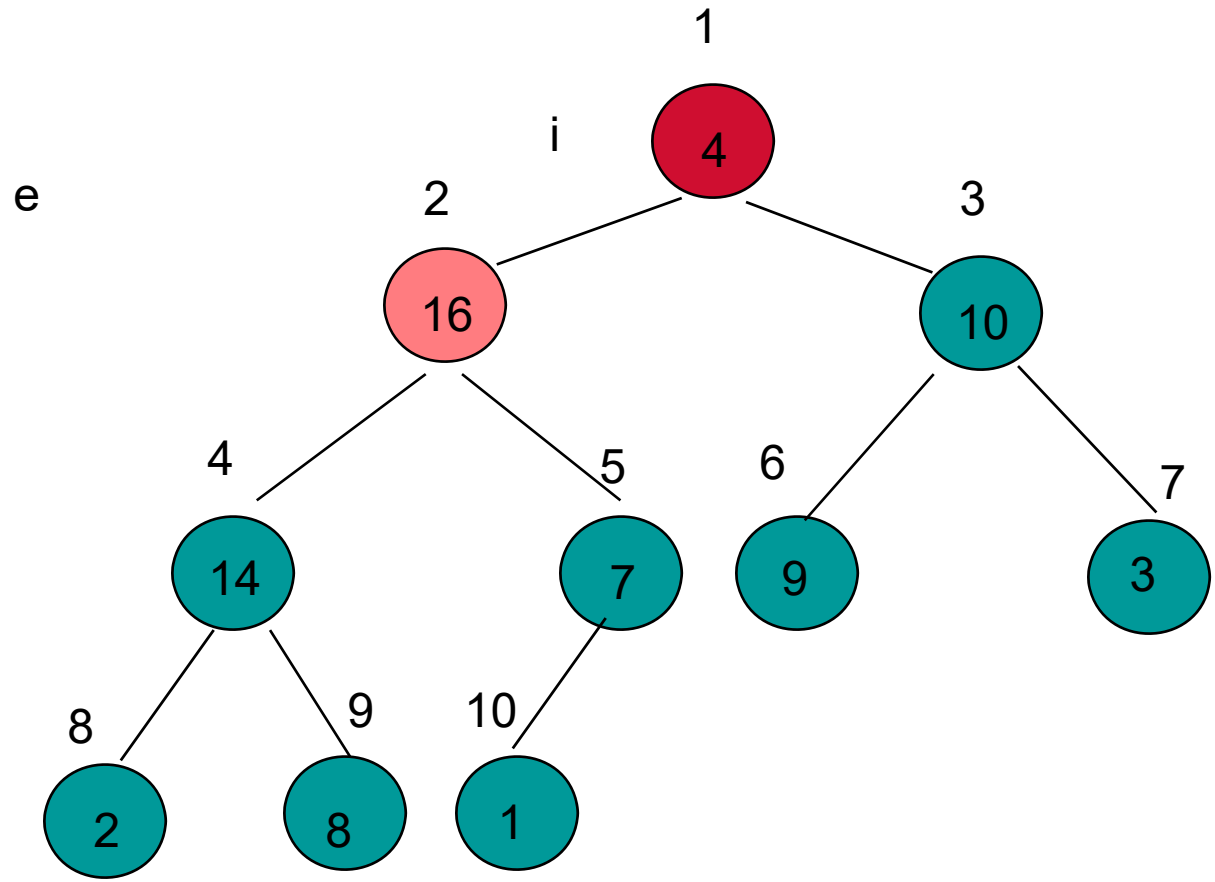


4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

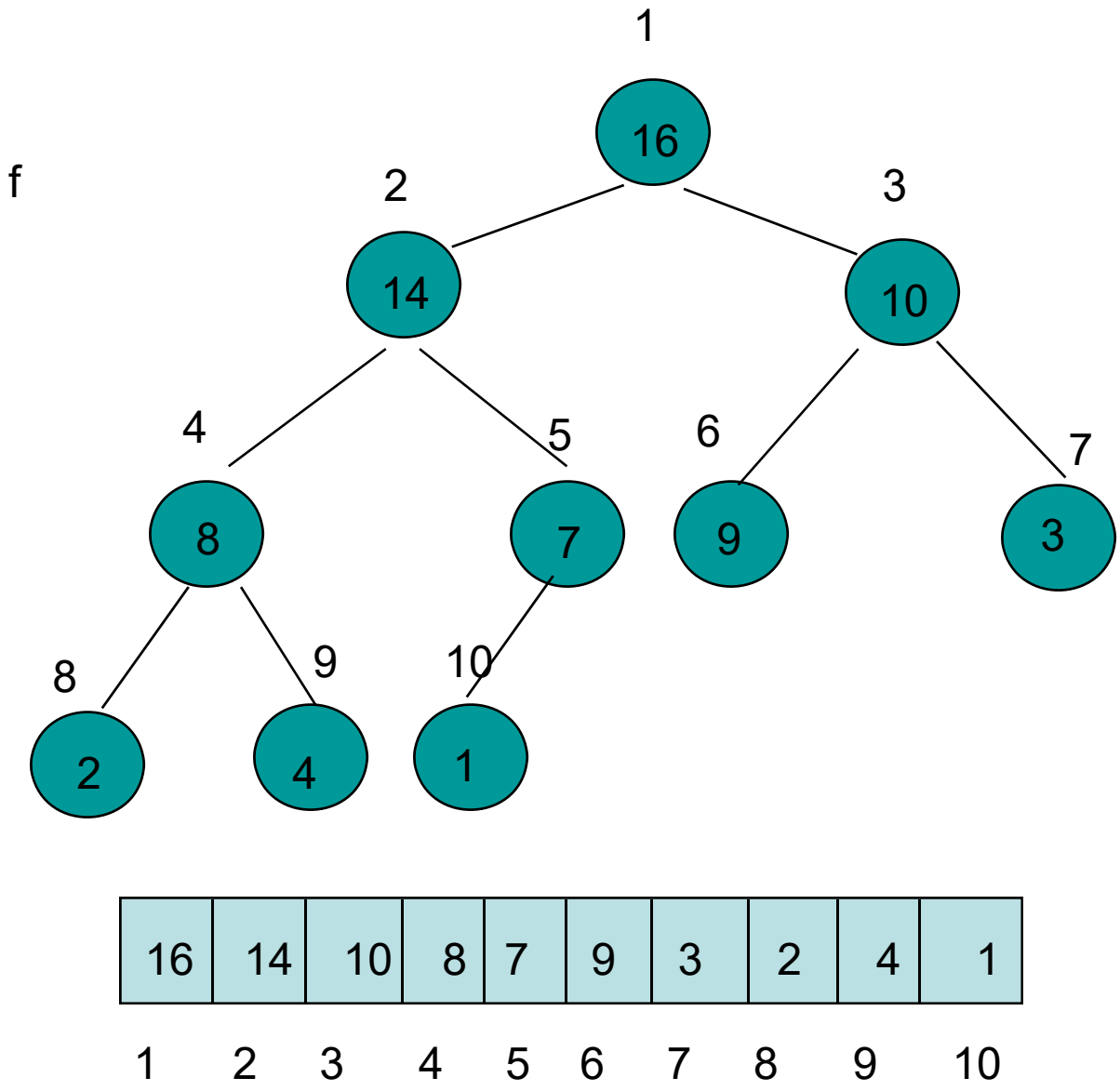


4	1	3	14	16	9	10	2	8	7
1	2	3	4	5	6	7	8	9	10


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



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



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

Running Time of BUILD-MAX-HEAP

- Simple upper bound:
  - each call to MAX-HEAPIFY costs $O(\lg n)$
  - $O(n)$ such calls
  - running time at most $O(n \lg n)$
- Previous bound is not tight:
  - lots of the elements are leaves
  - most elements are near leaves (small height)

Tighter Bound for BUILD-MAX-HEAP

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

By substituting x = ½ in the formula for differentiating infinite geometric series, we have:

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

Tighter Bound for BUILD-MAX-HEAP (continued)

Thus the running time is bounded by:

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

Therefore, we can build a heap from an unordered array in linear time.

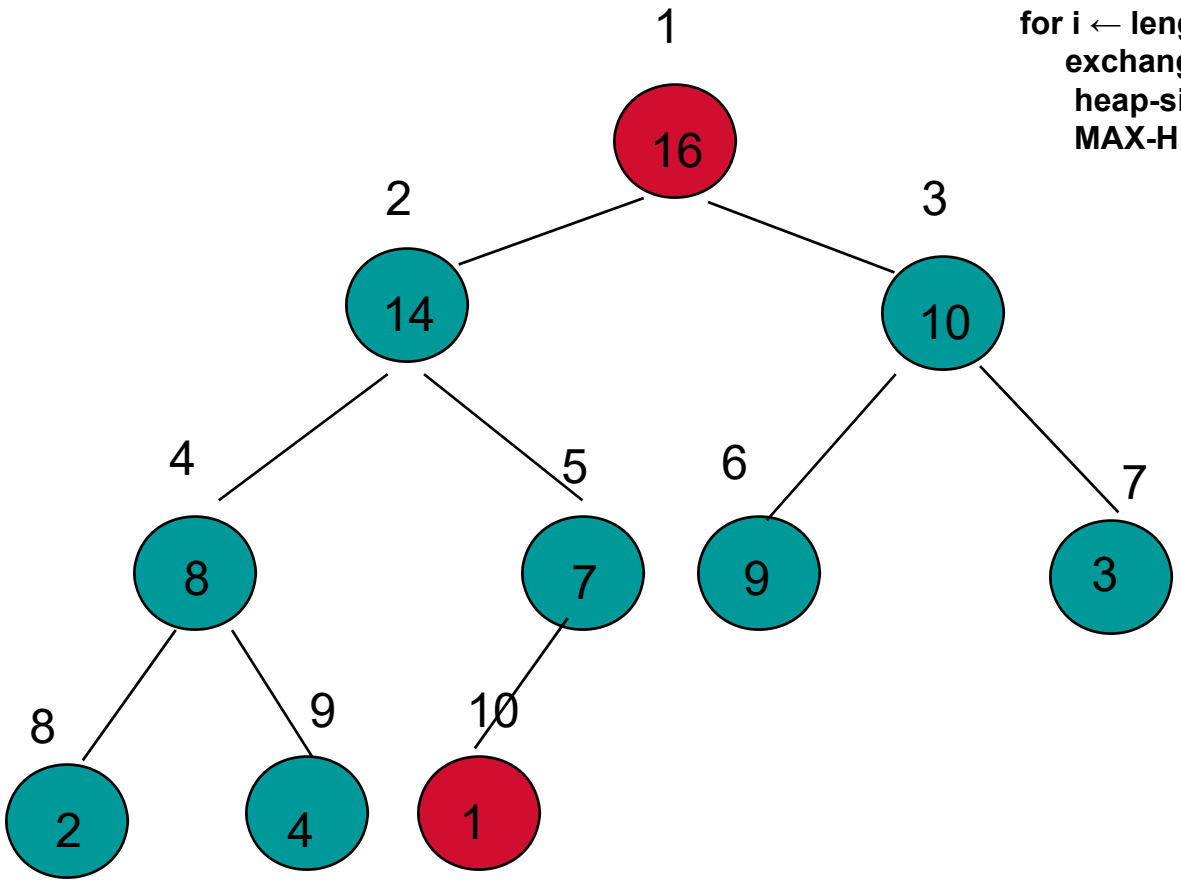
Heapsort

- First build a heap.
- Then successively remove the biggest element from the heap and move it to the first position in the sorted array.
- The element currently in that position is then placed at the top of the heap and sifted to the proper position.

HEAPSORT

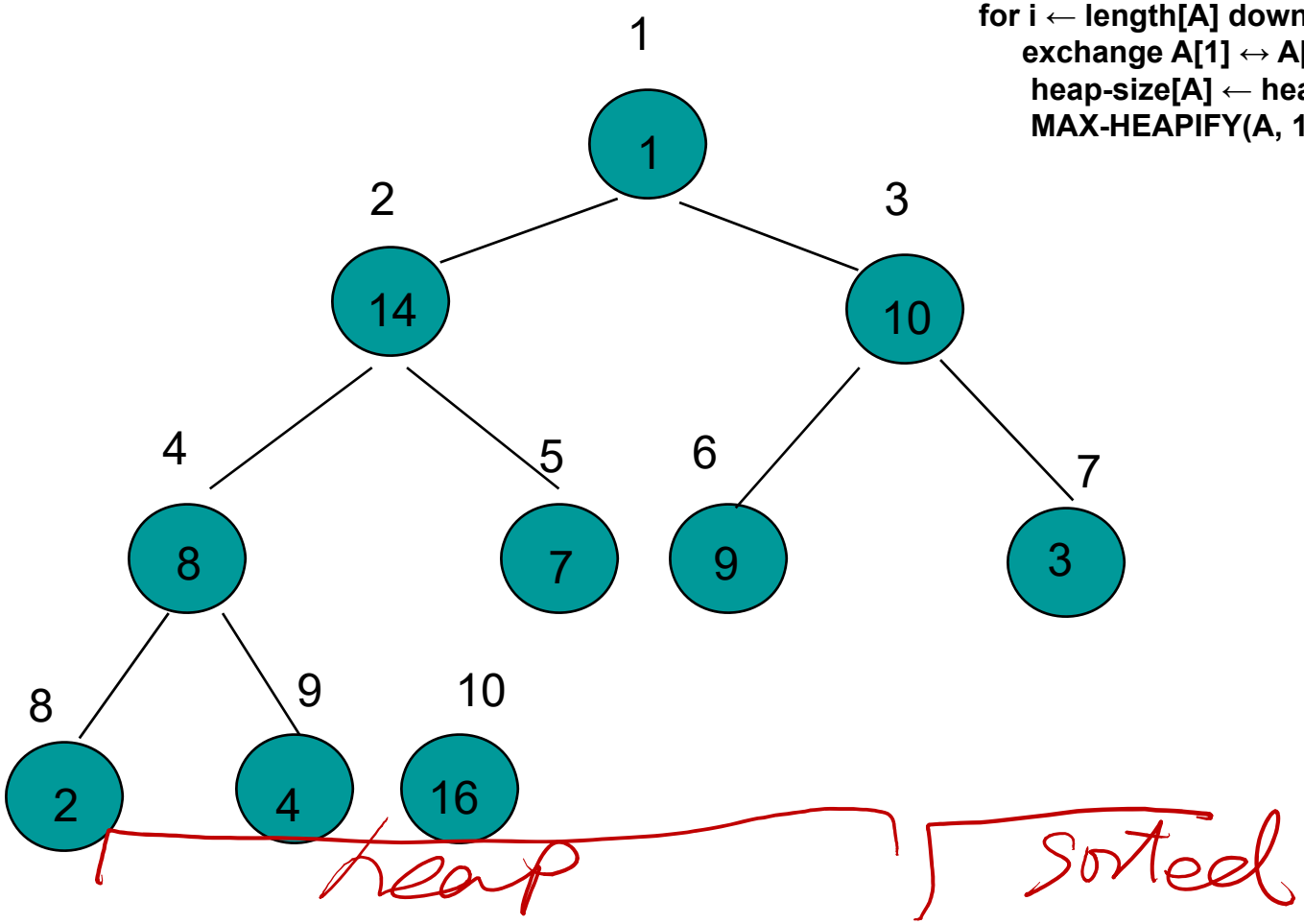
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2 do
3    exchange A[1] ↔ A[i]
4    heap-size[A] ← heap-size[A] – 1
5    MAX-HEAPIFY(A, 1)
```

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)



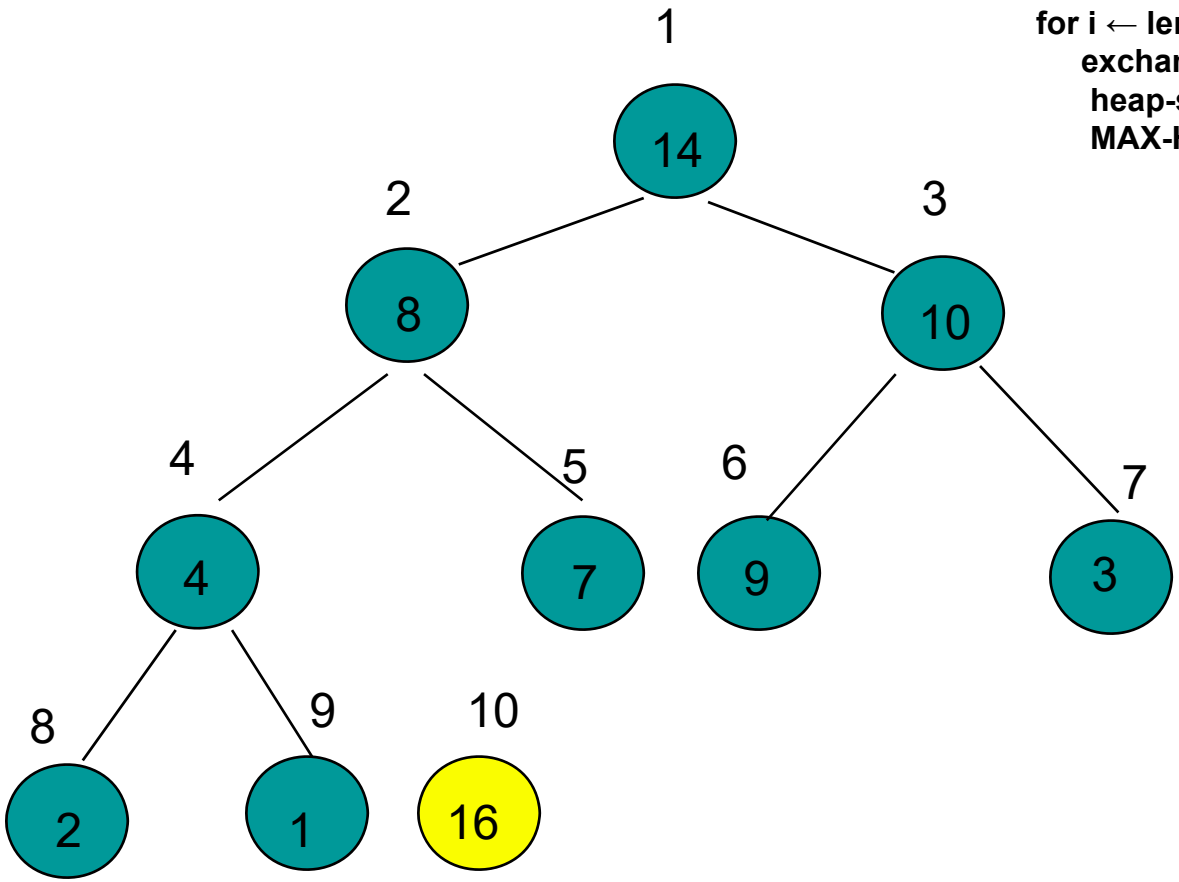
16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

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)



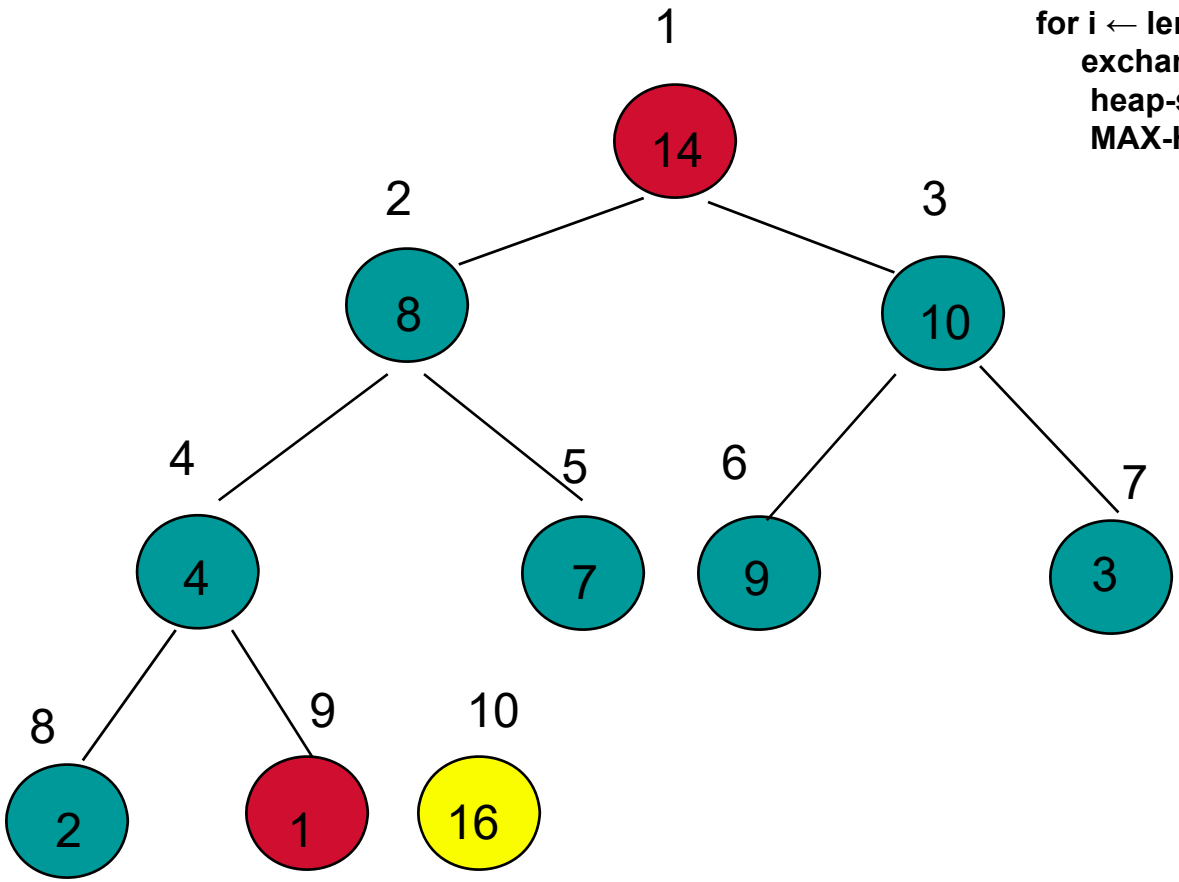
1	2	3	4	5	6	7	8	9	10
1	14	10	8	7	9	3	2	4	16

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)



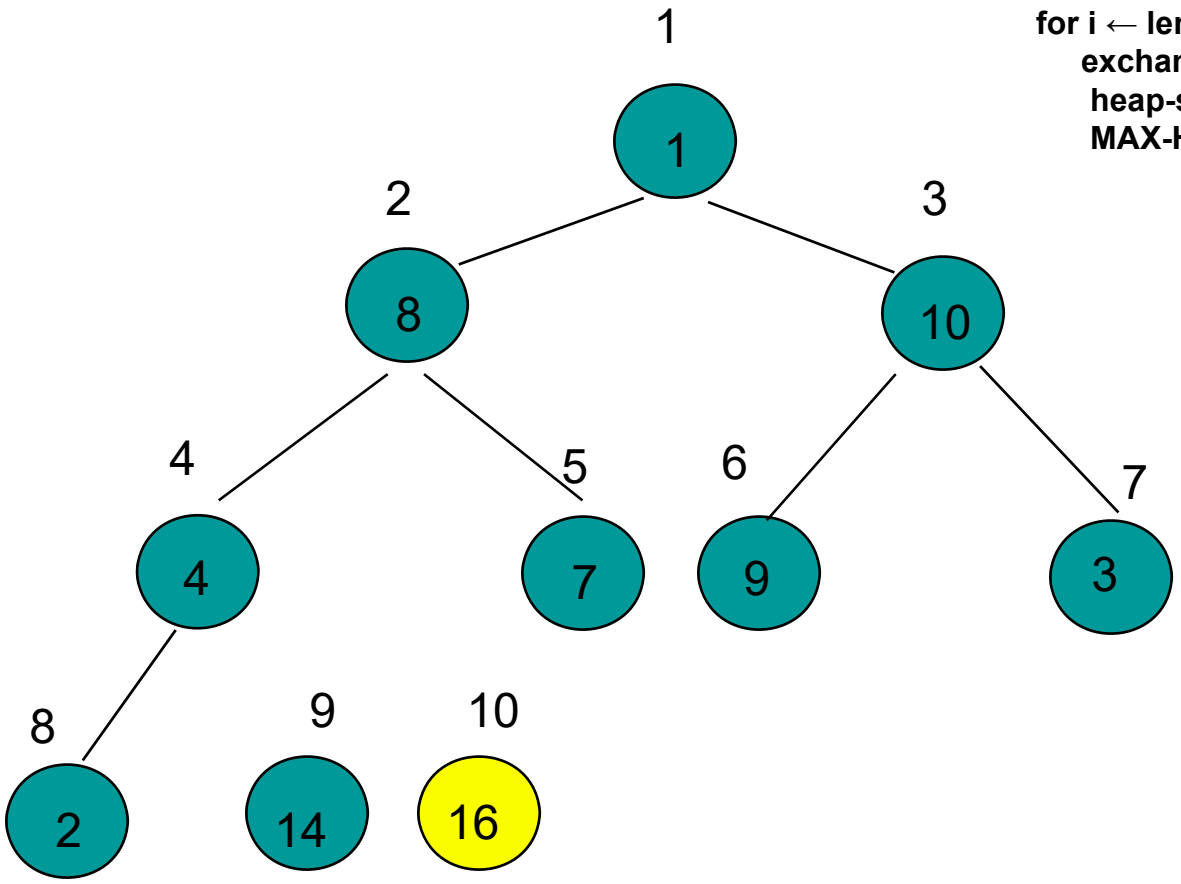
14	8	10	4	7	9	3	2	1	16
1	2	3	4	5	6	7	8	9	10

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)



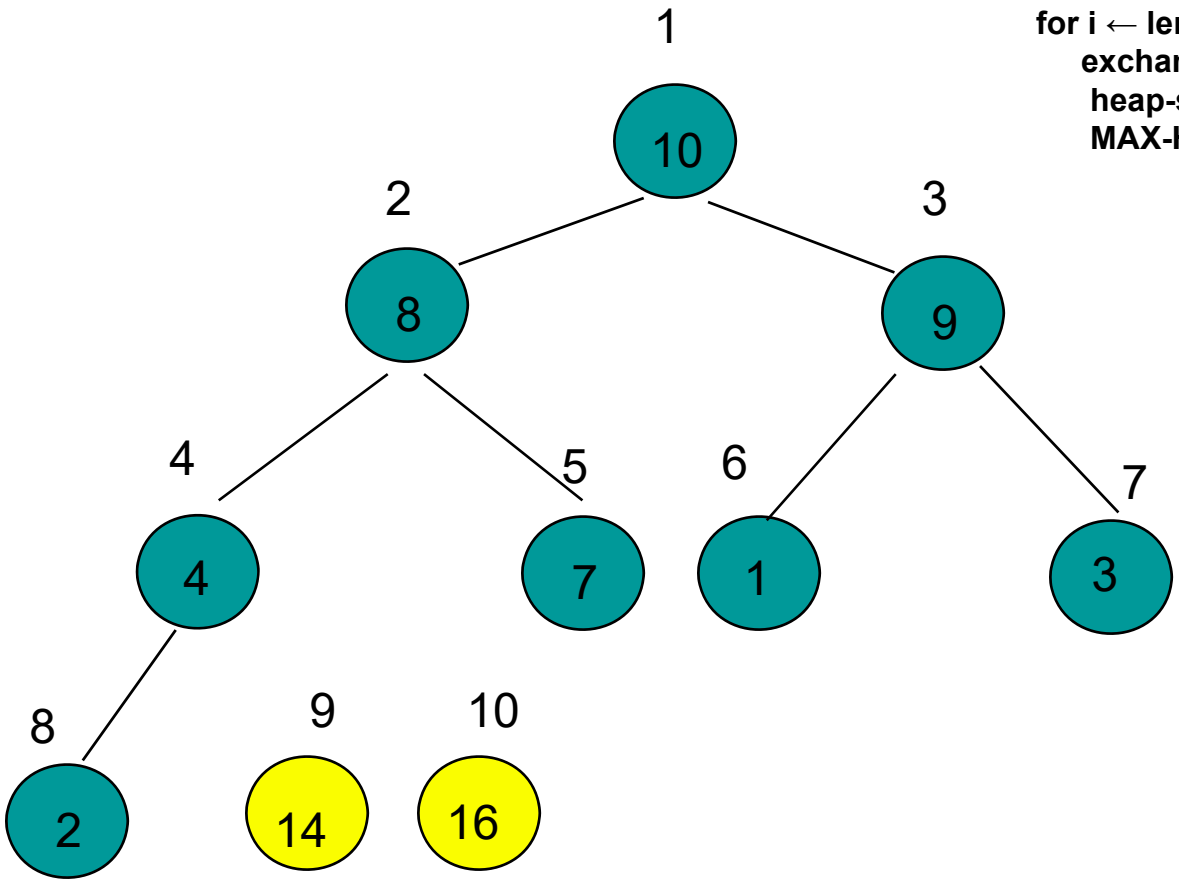
14	8	10	4	7	9	3	2	1	16
1	2	3	4	5	6	7	8	9	10

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)



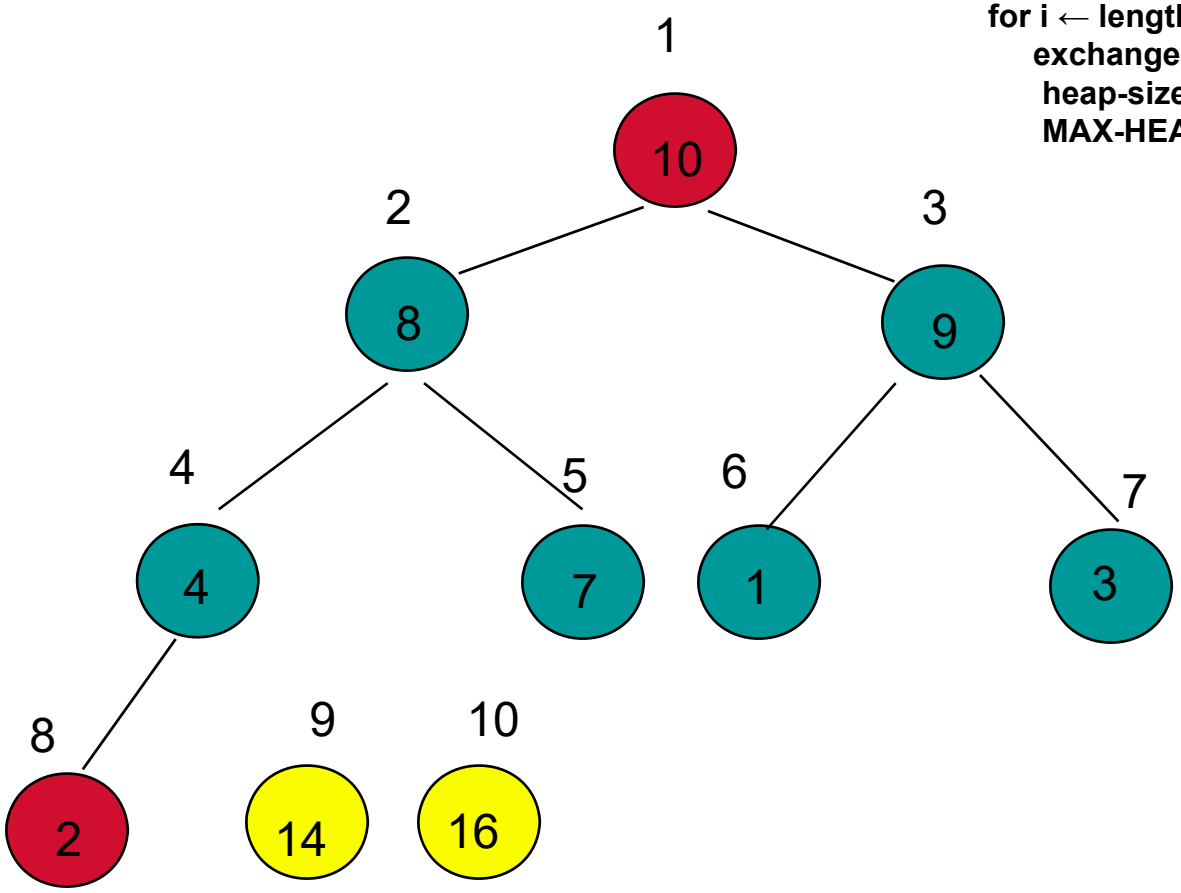
1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	3	2	14	16

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)



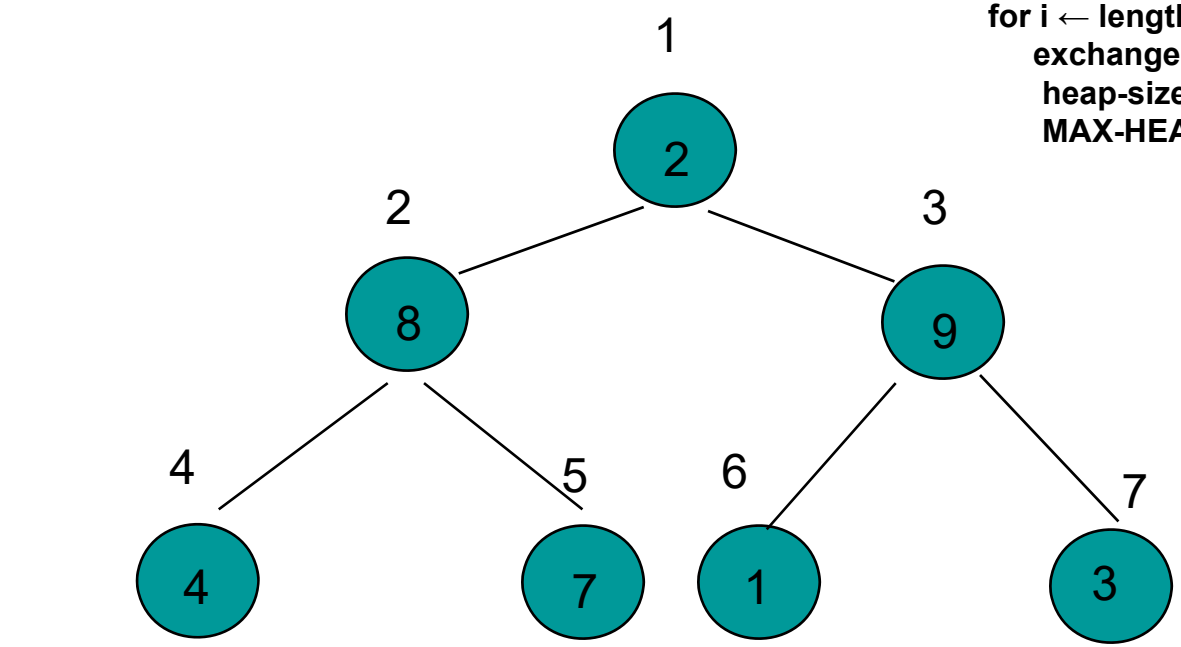
10	8	9	4	7	1	3	2	14	16
1	2	3	4	5	6	7	8	9	10

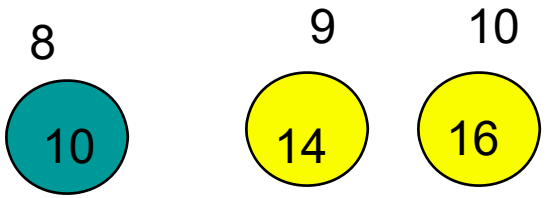
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



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

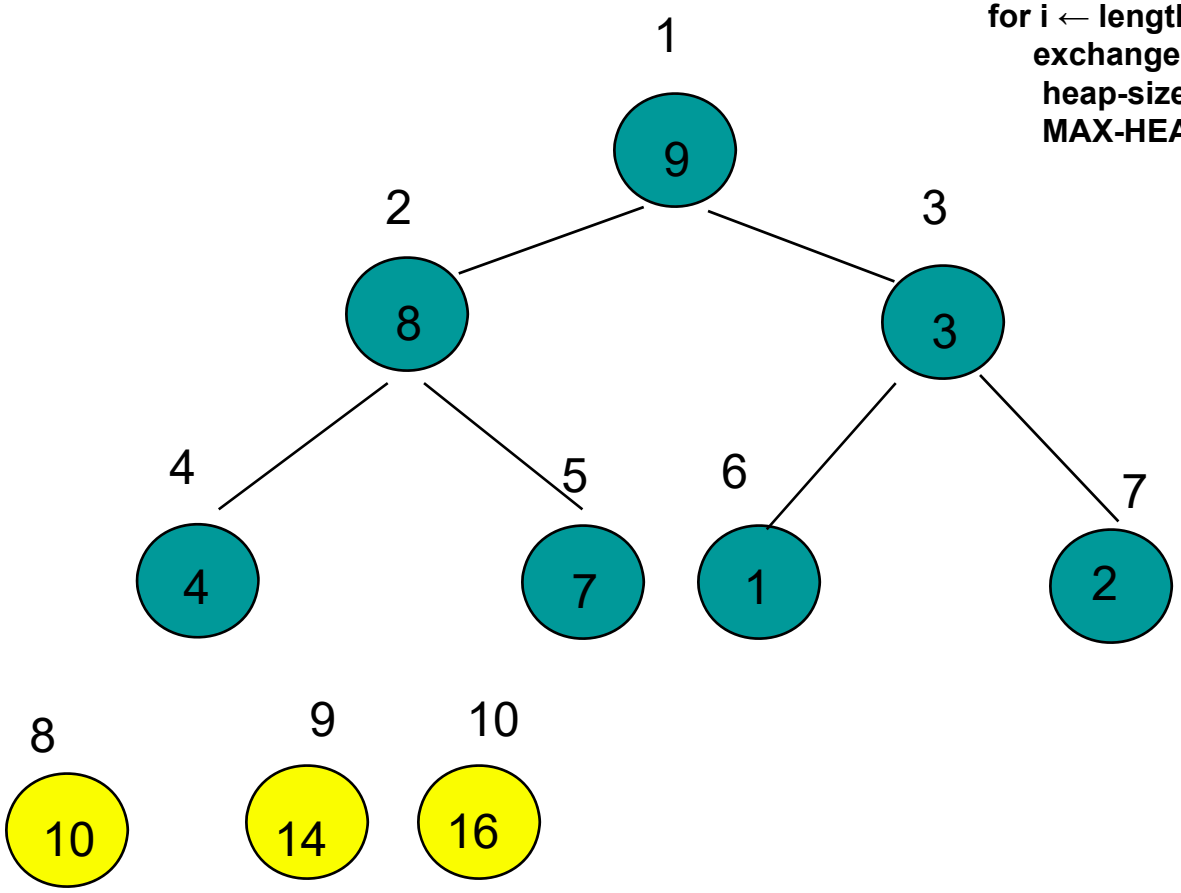
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)





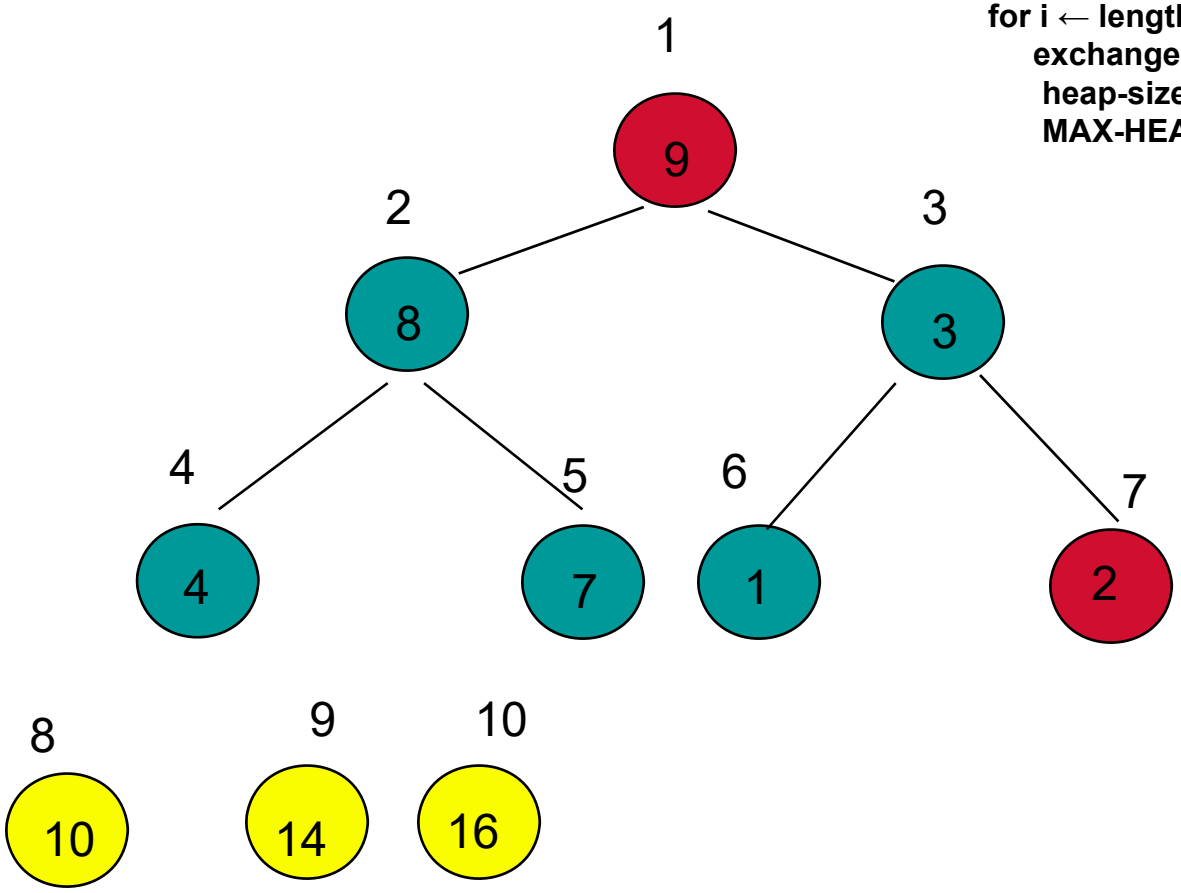
2	8	9	4	7	1	3	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



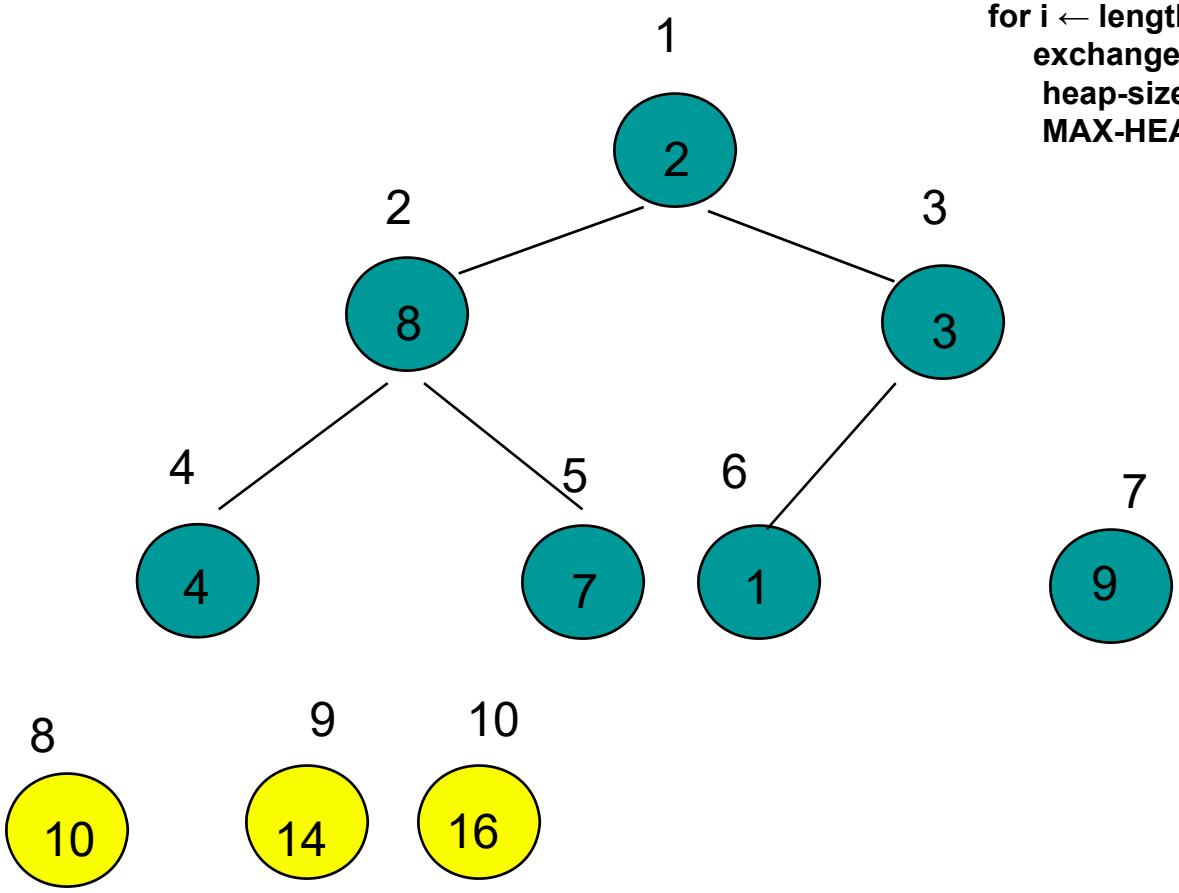
9	8	3	4	7	1	2	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



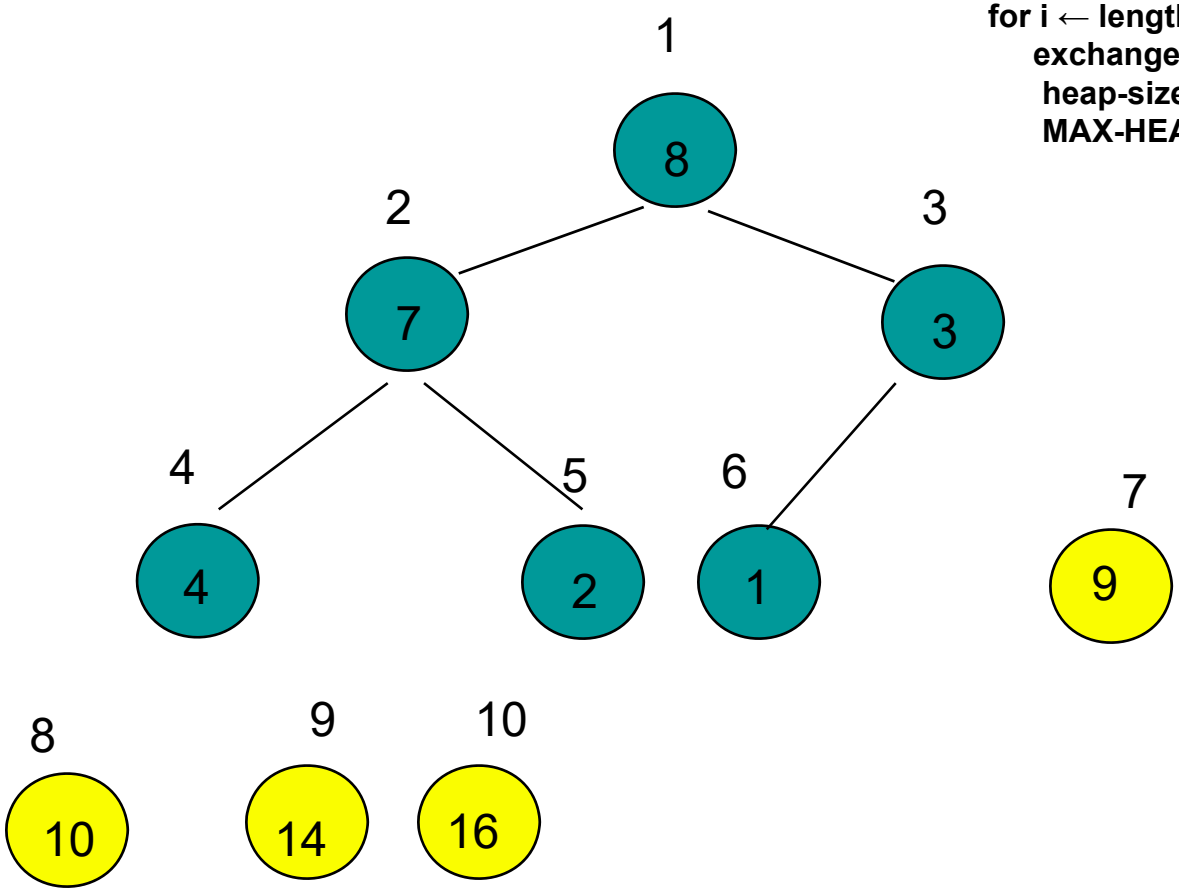
9	8	3	4	7	1	2	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



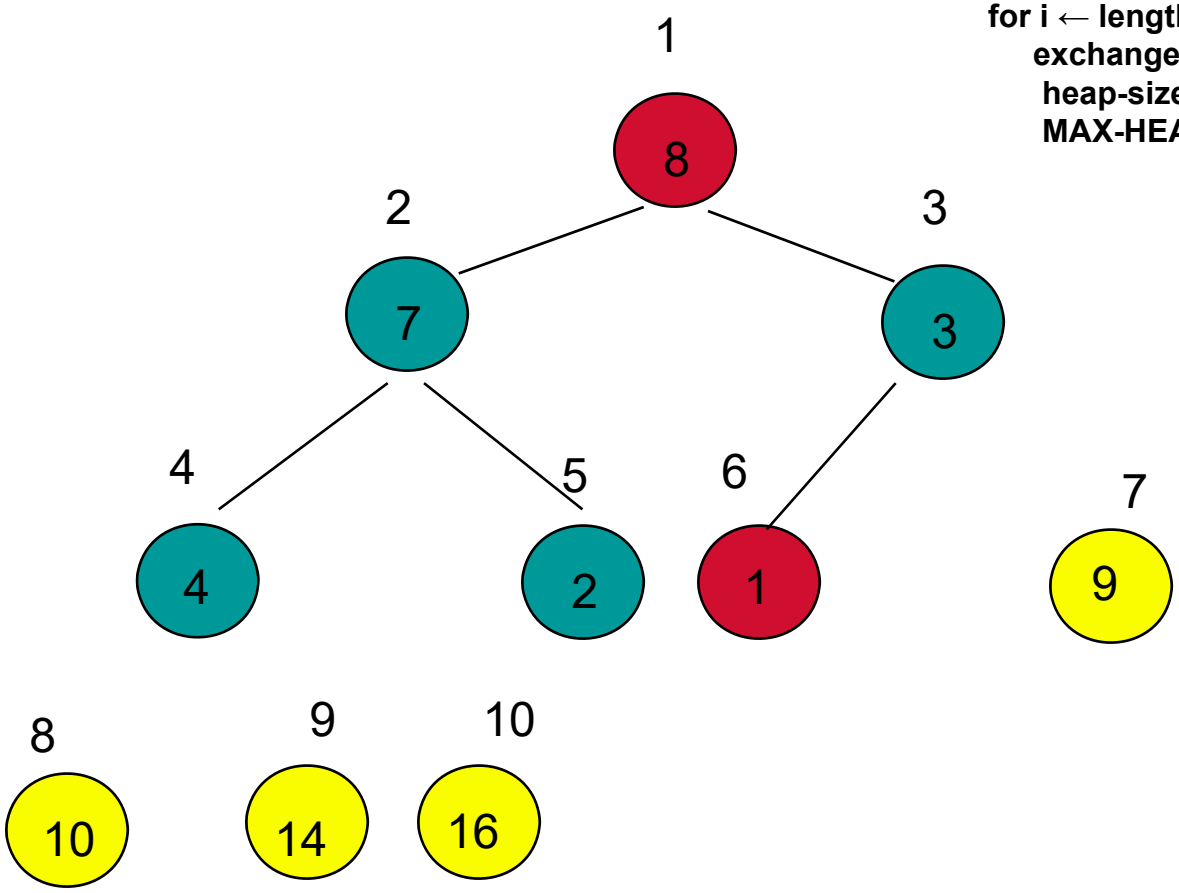
2	8	3	4	7	1	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
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)
```



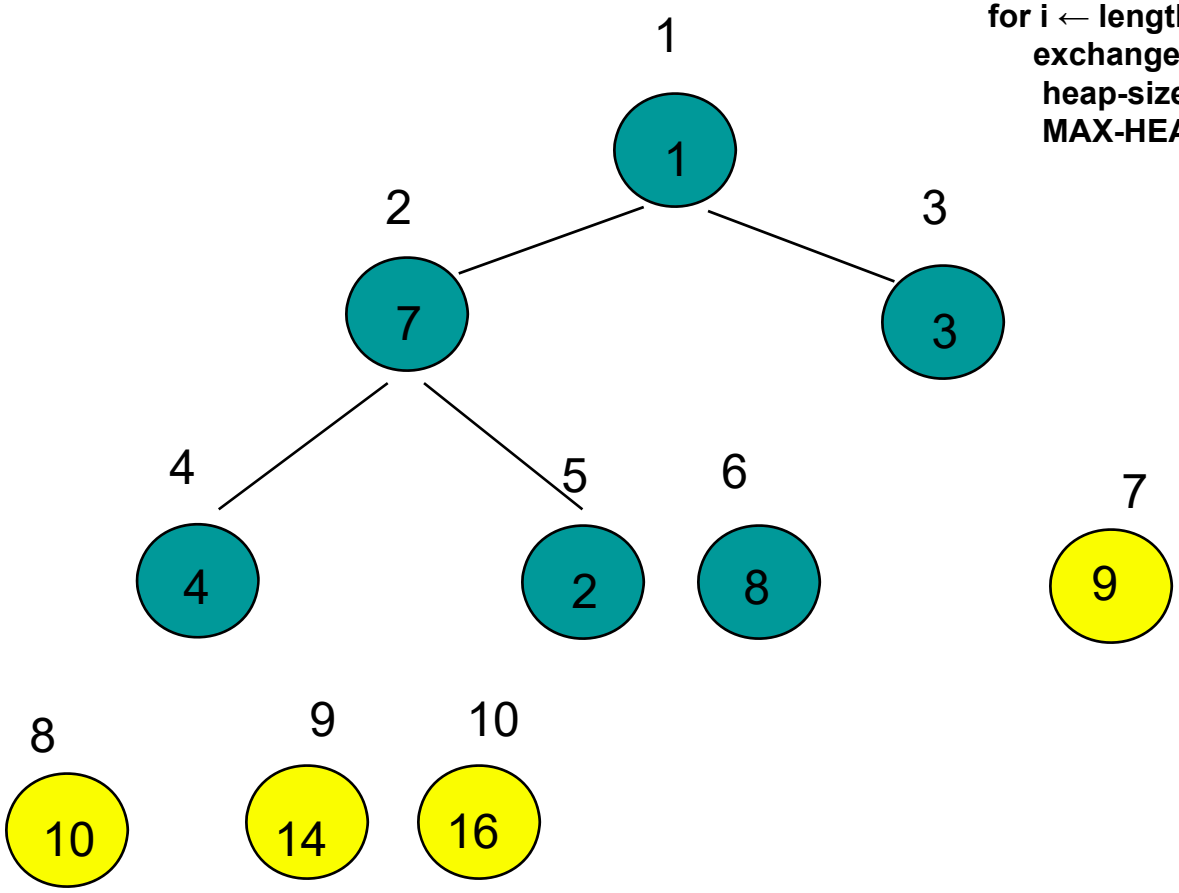
8	7	3	4	2	1	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



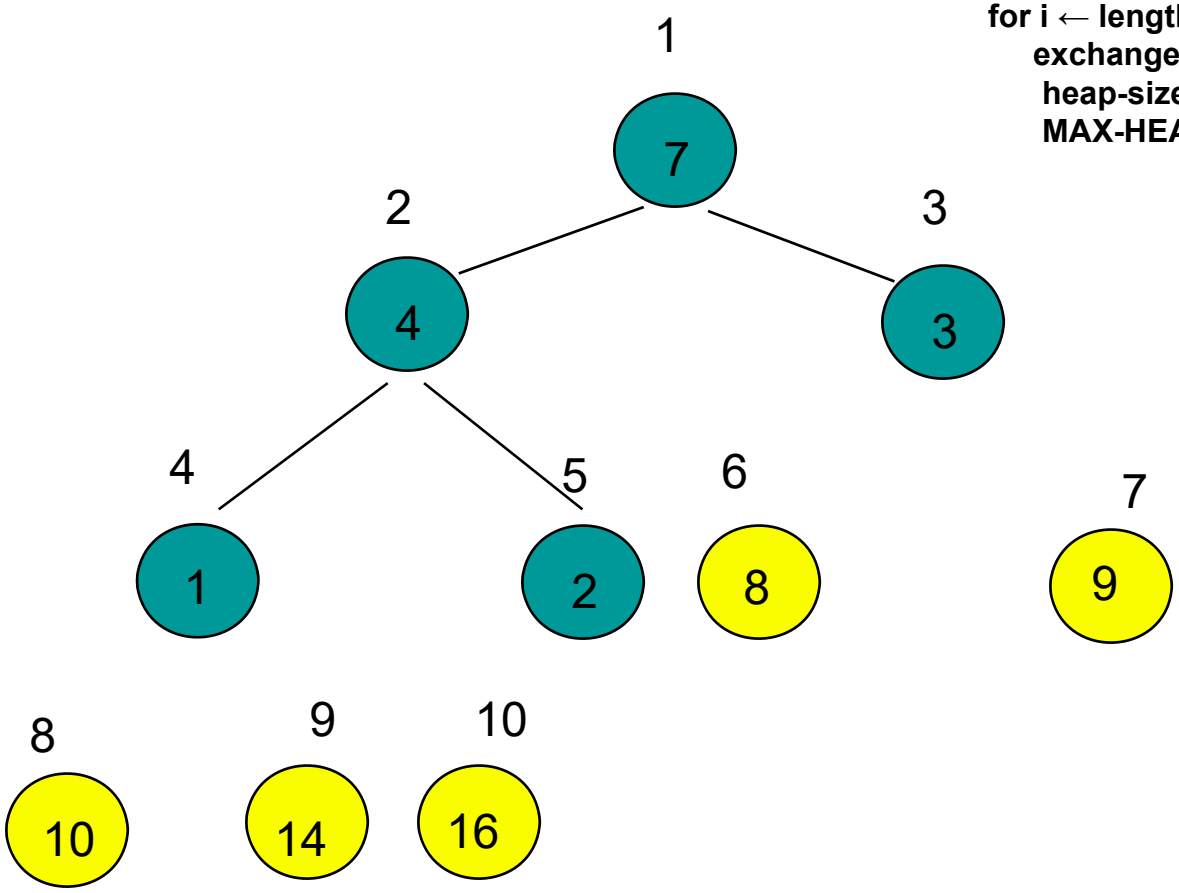
8	7	3	4	2	1	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



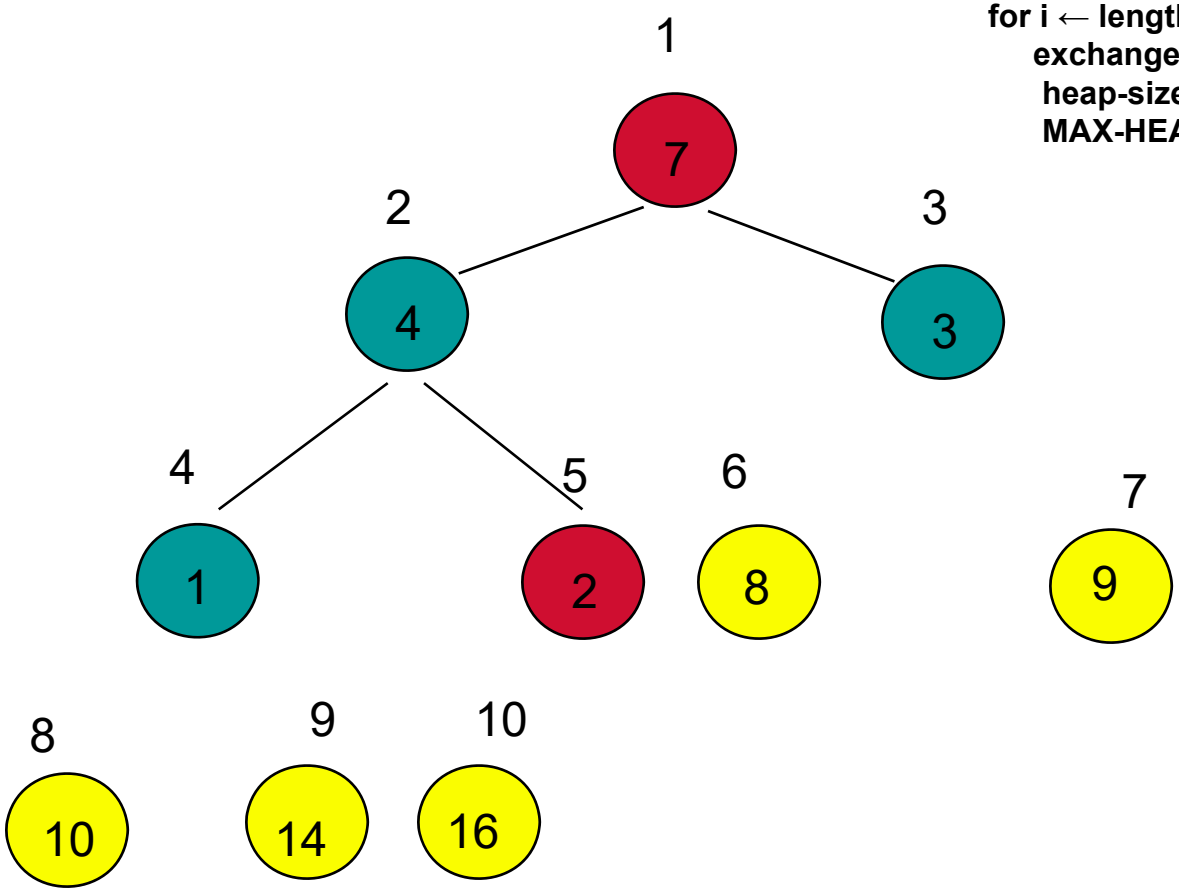
1	2	3	4	5	6	7	8	9	10
1	7	3	4	2	8	9	10	14	16

```
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)
```



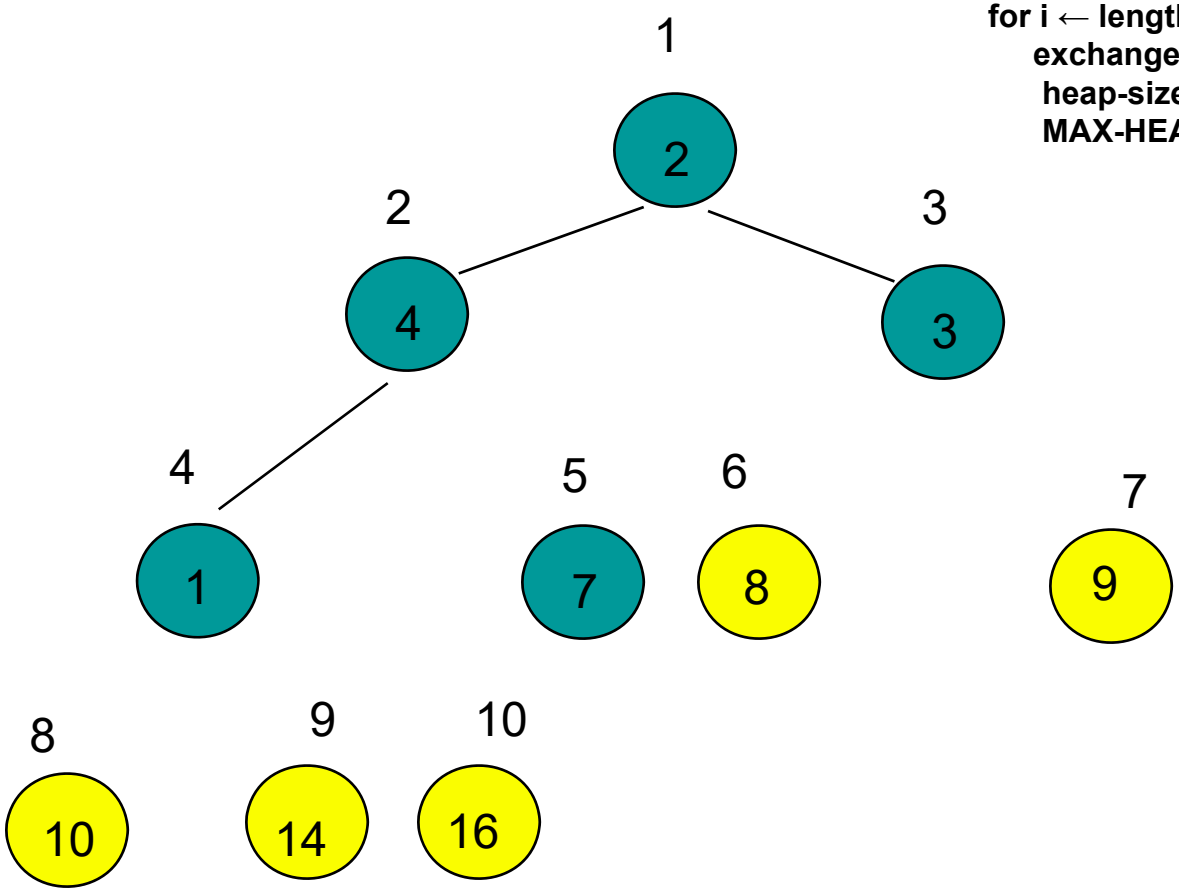
7	4	3	1	2	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



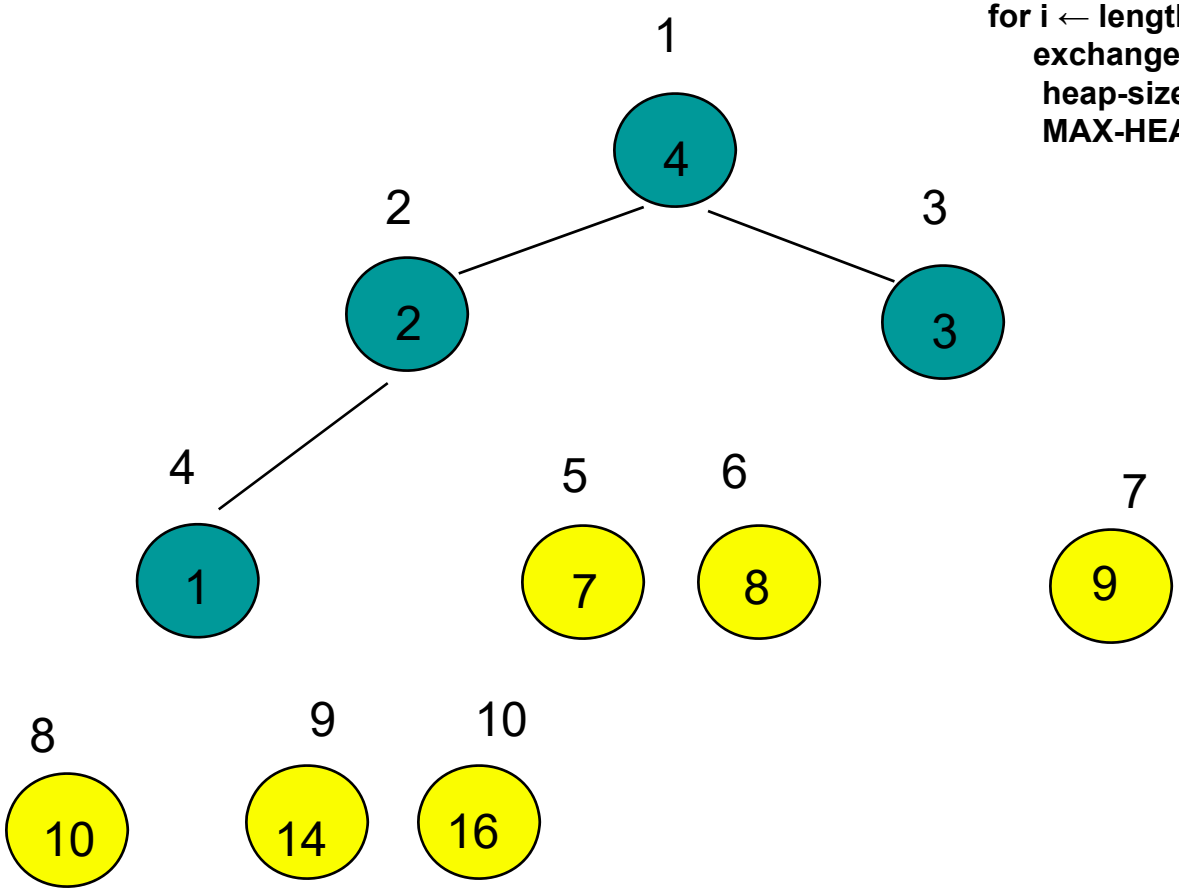
7	4	3	1	2	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
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)
```



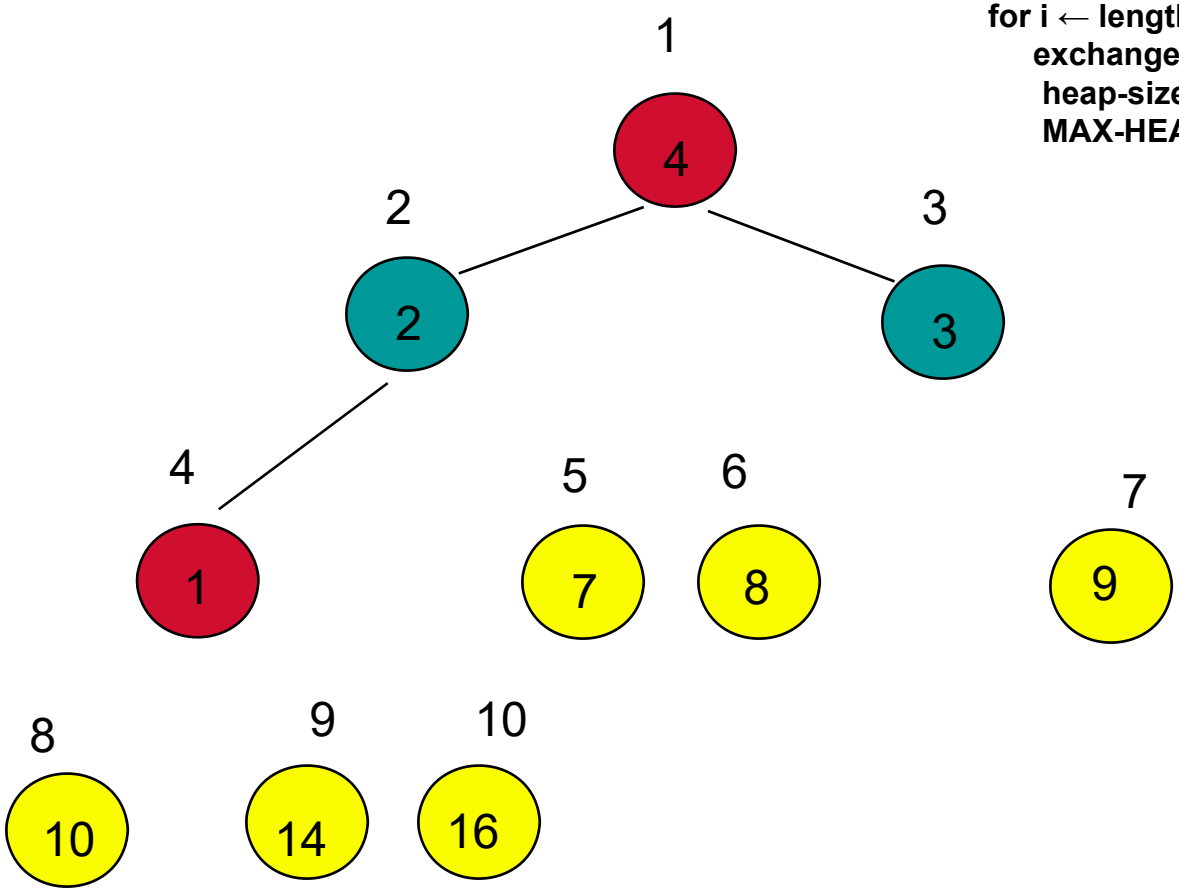
2	4	3	1	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
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)
```



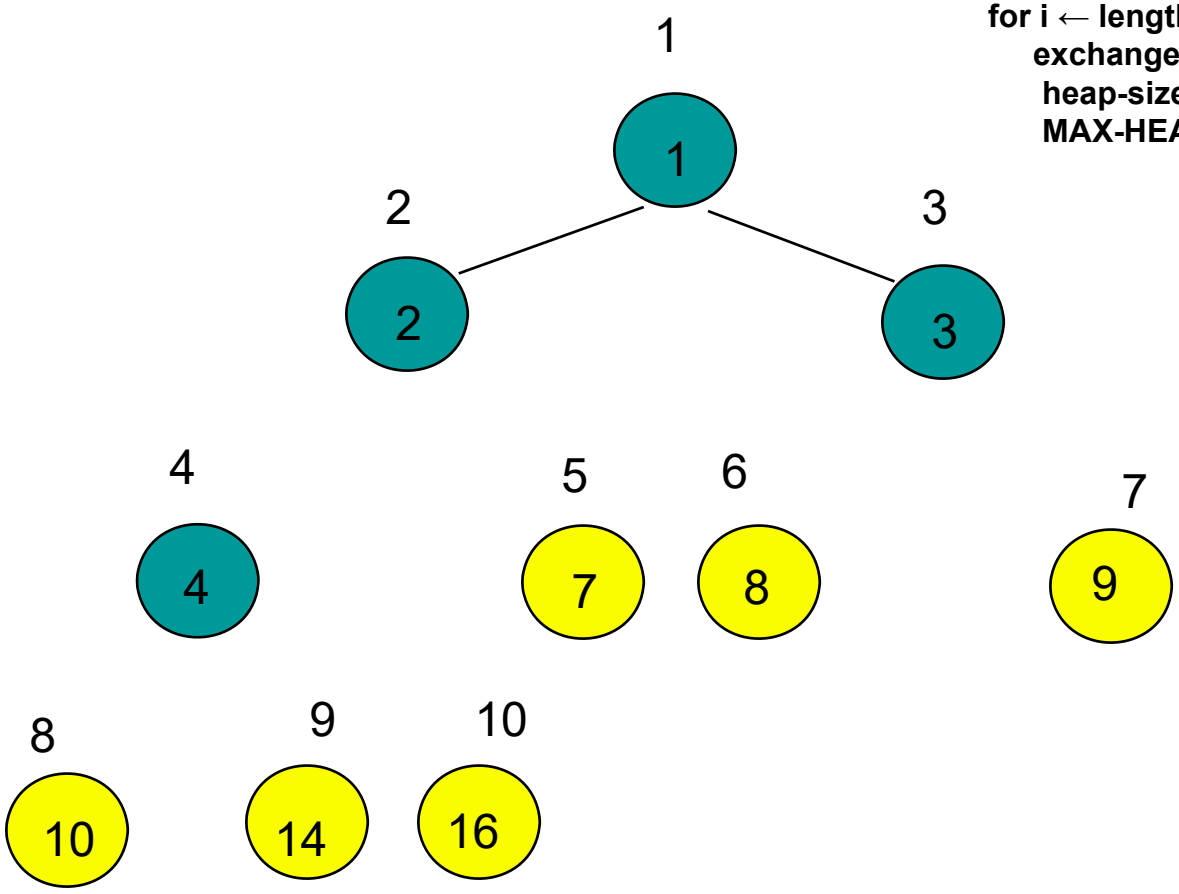
4	2	3	1	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



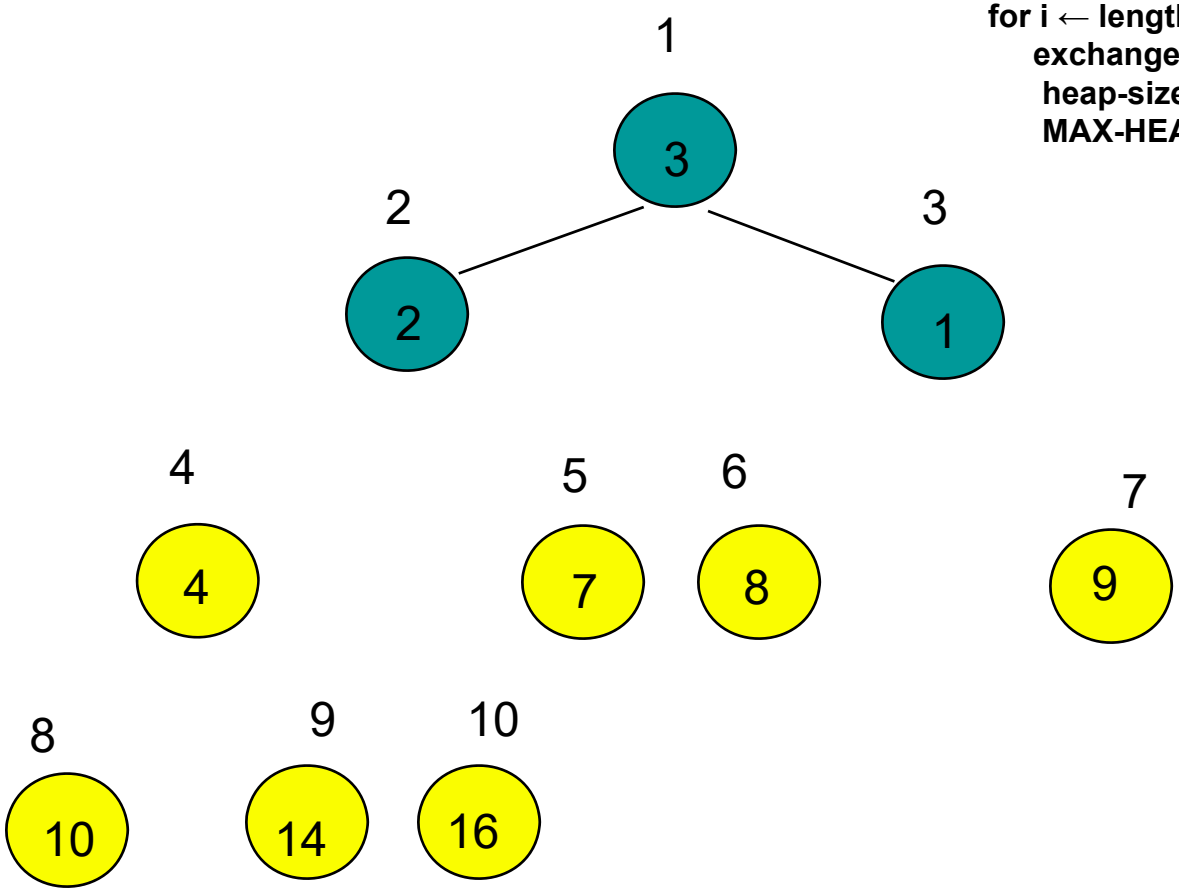
4	2	3	1	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
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)
```



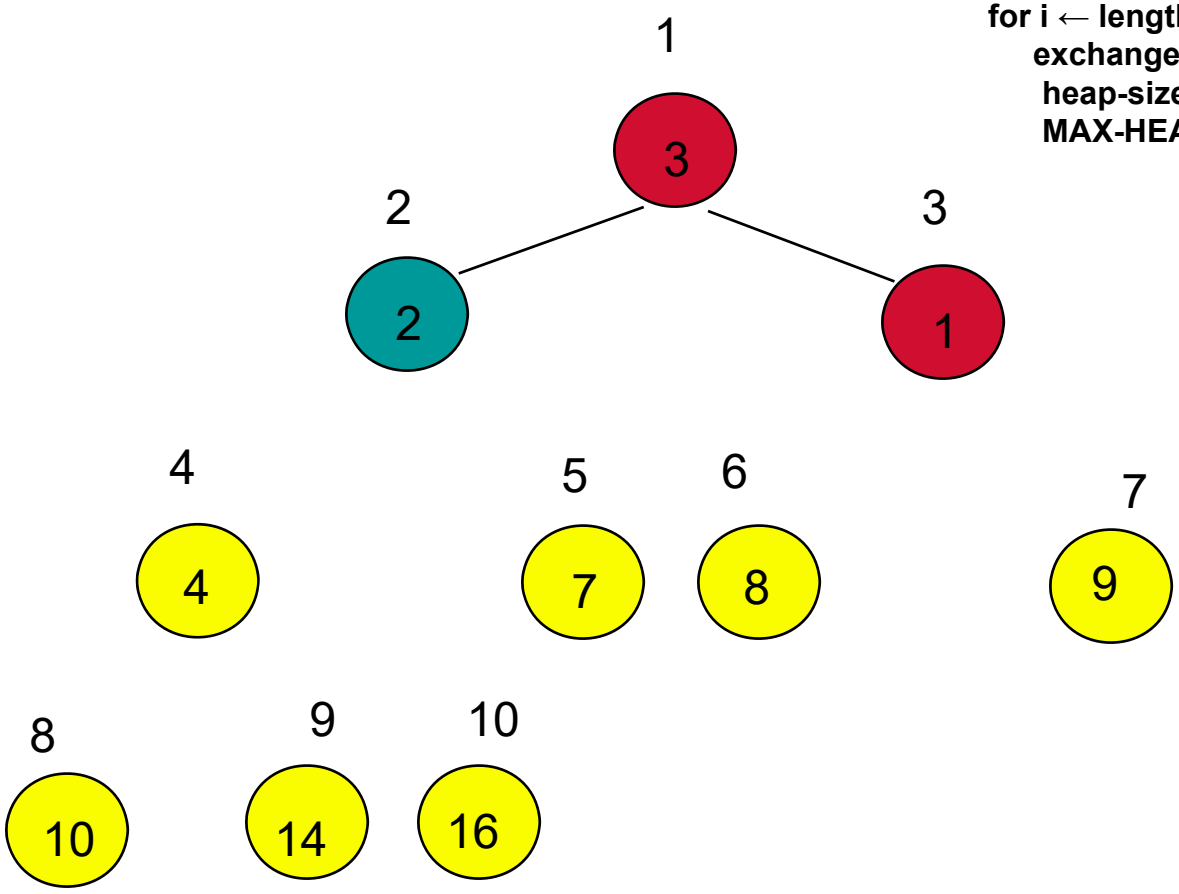
1	2	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



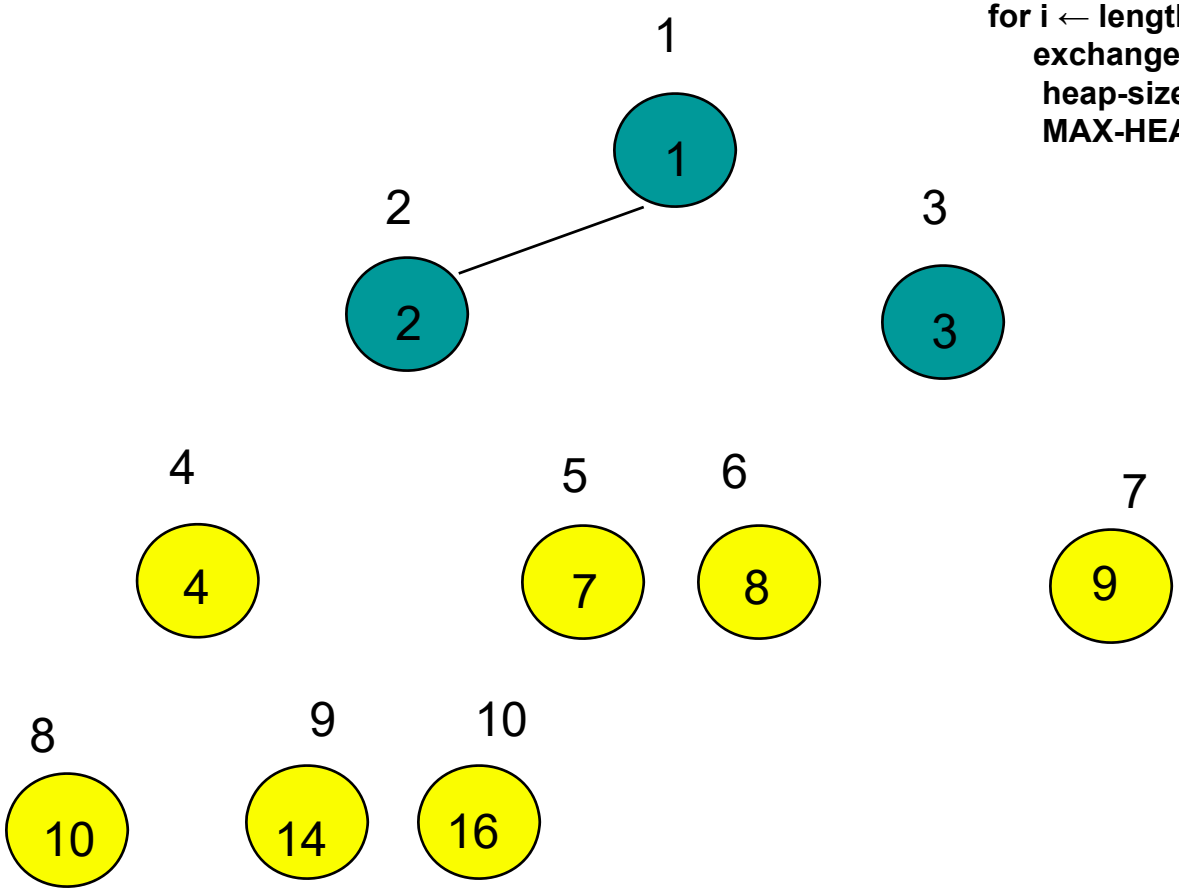
3	2	1	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



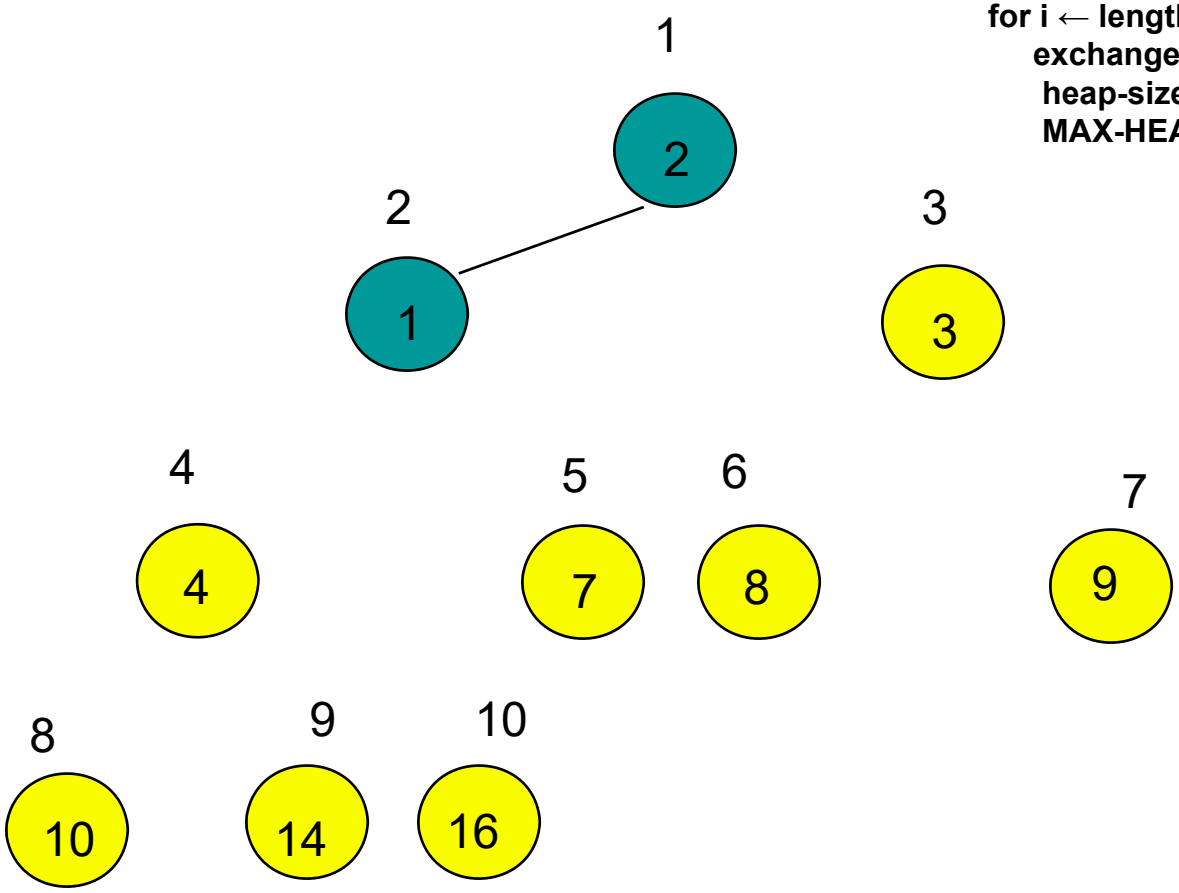
3	2	1	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
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)
```



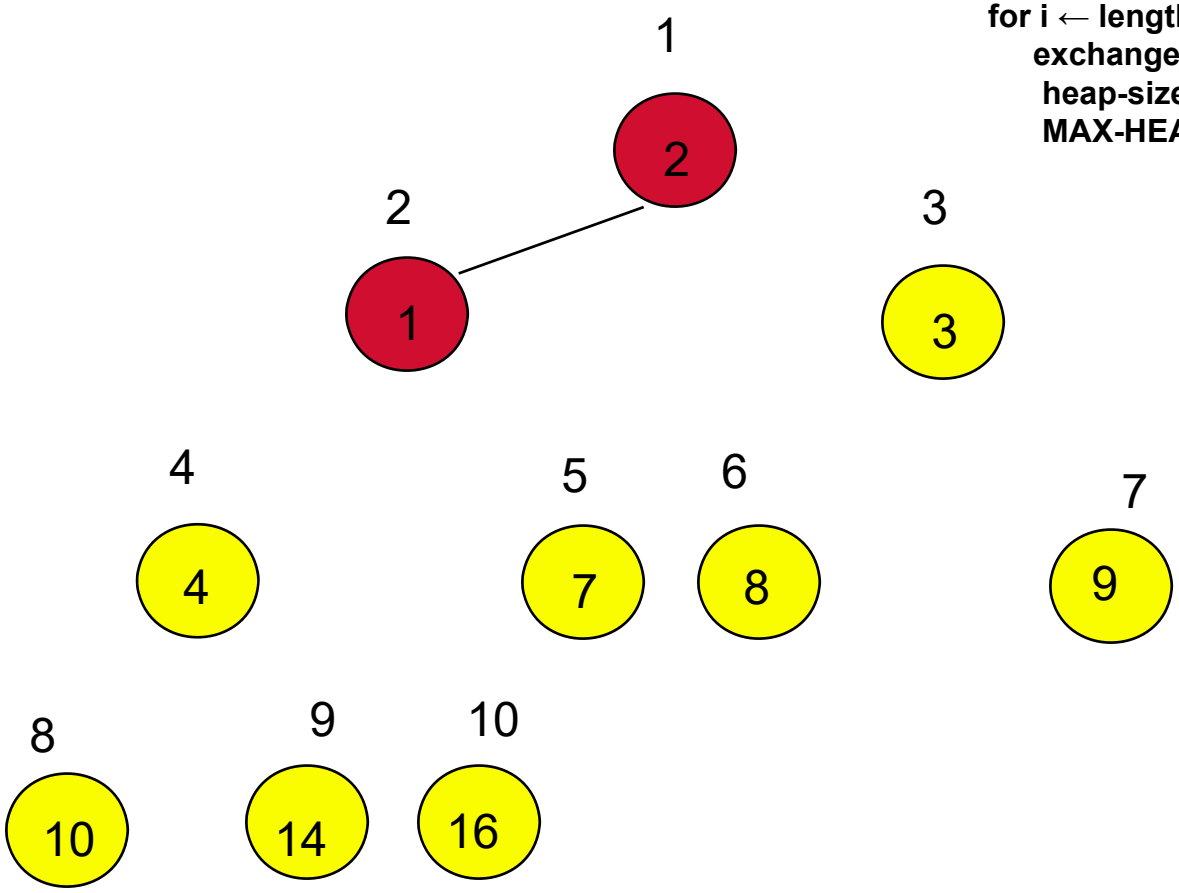
1	2	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



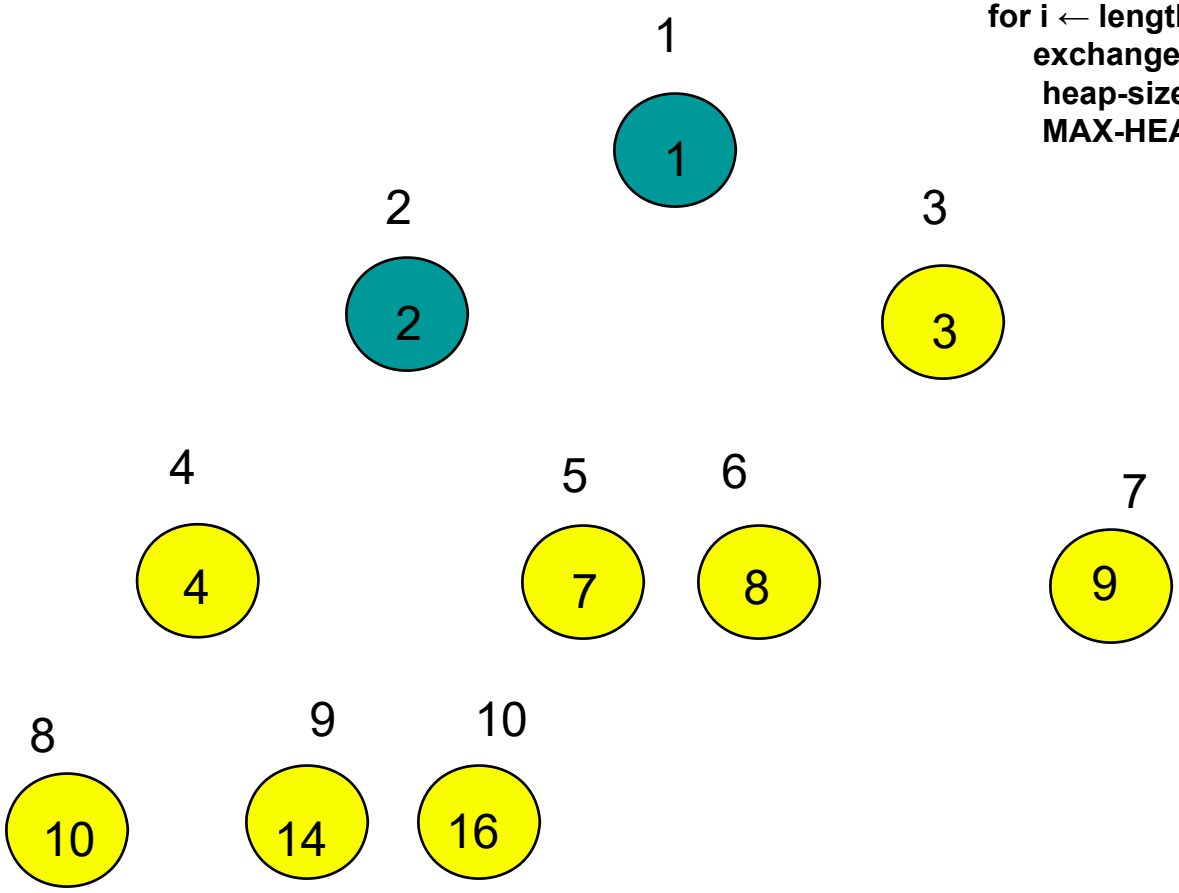
2	1	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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)



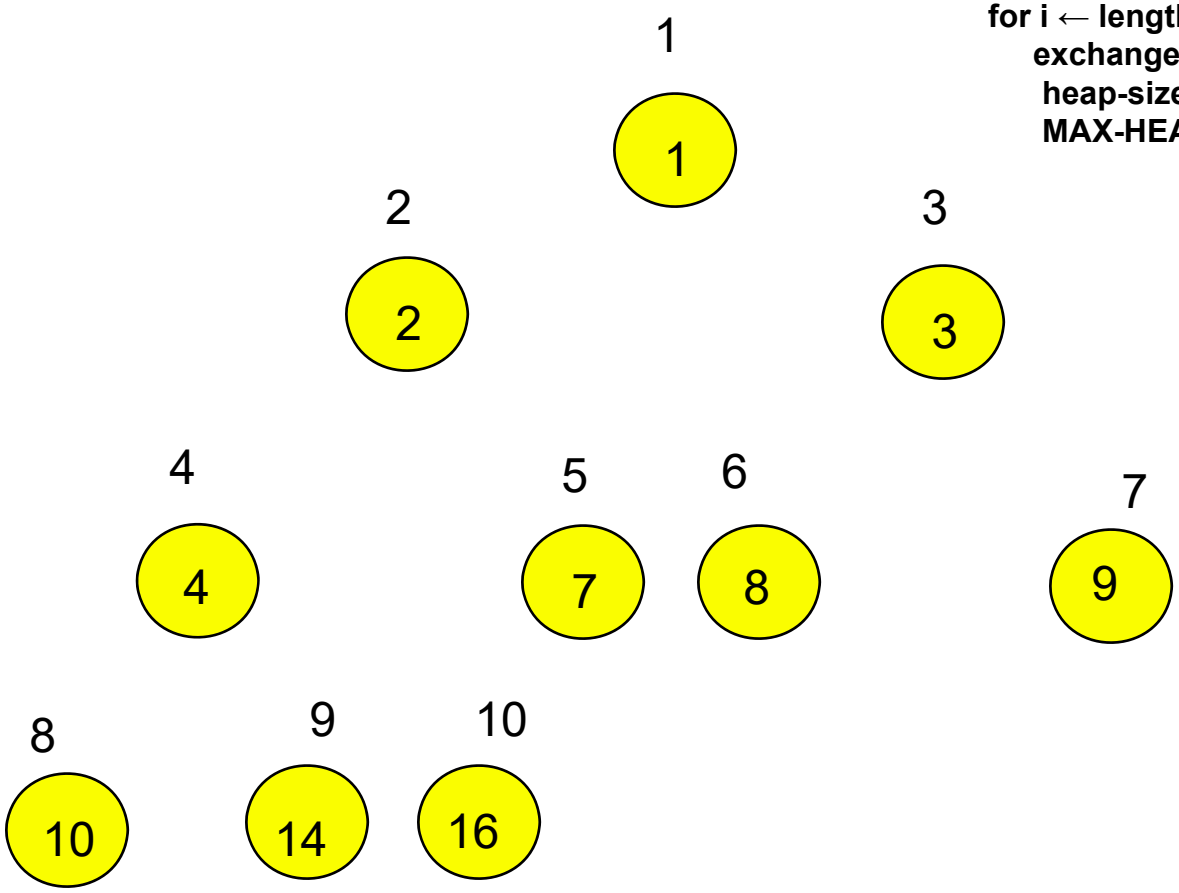
2	1	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

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

Is there a loop? If so, how many times will it execute? What is the cost of one iteration of the loop?

Running time of Heapsort

```
HEAPSORT(A)
1   BUILD-MAX-HEAP(A)                          O(n)
2   for i ← length[A] downto 2 do              O(n-1)
3    exchange A[1] ↔ A[i]                      O(1)
4    heap-size[A] ← heap-size[A] - 1           O(1)
5    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 $O(n \lg n)$

Running time of Heapsort

- BUILD-MAX-HEAP takes O(n).
- We have a loop. Each of the n-1 calls to MAX-HEAPIFY takes O(lg n) time.
- Total time is O(n lg n).
- Will heap sort always take O(n lg n) time? Is there a best-case scenario? Is there a worst-case scenario? Why or why not?

Space requirements of Heapsort

- Heapsort uses an array as its data structure.
- Heapsort sorts “in place”.
- Any extra storage needed?
- Only a negligible amount – one extra storage location is needed as temporary storage when swapping two array elements.

Priority Queues

- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key.
- Applications include
  - scheduling jobs on a shared computer (max-priority queue)
  - event-driven simulators (min-priority queue)

Max-Priority Queue Operations

- INSERT(S,x): insert element x into set S
- MAXIMUM(S): return element of S with the largest key
- EXTRACT-MAX(S): remove and return element of S with the largest key
- INCREASE-KEY(S, x, k): increase value of x's key to k, where k is at least as large as x's current key value

Min-Priority Queue Operations

- INSERT(S,x): insert element x into set S
- MINIMUM(S): return element of S with the smallest key
- EXTRACT-MIN(S): remove and return element of S with the smallest key
- DECREASE-KEY(S, x, k): decrease value of x's key to k, where k is at least as small as x's current key value

Priority Queue Operations

- All operations can be done on a set of size n in $O(\lg n)$ time

HEAP-MAXIMUM

```
HEAP-MAXIMUM(A)
1  return A[1]
```

- Returns the item at the top of the heap
- Runs in $\Theta(1)$ time

HEAP-EXTRACT-MAX

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2     then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] - 1
6  MAX-HEAPIFY(A,1)
7  return max
```

Running time of HEAP-EXTRACT-MAX

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1                          O(1)
2     then error "heap underflow"               O(1)
3  max ← A[1]                                   O(1)
4  A[1] ← A[heap-size[A]]                       O(1)
5  heap-size[A] ← heap-size[A] - 1              O(1)
6  MAX-HEAPIFY(A,1)                             O(lg n)
7  return max                                   O(1)
```

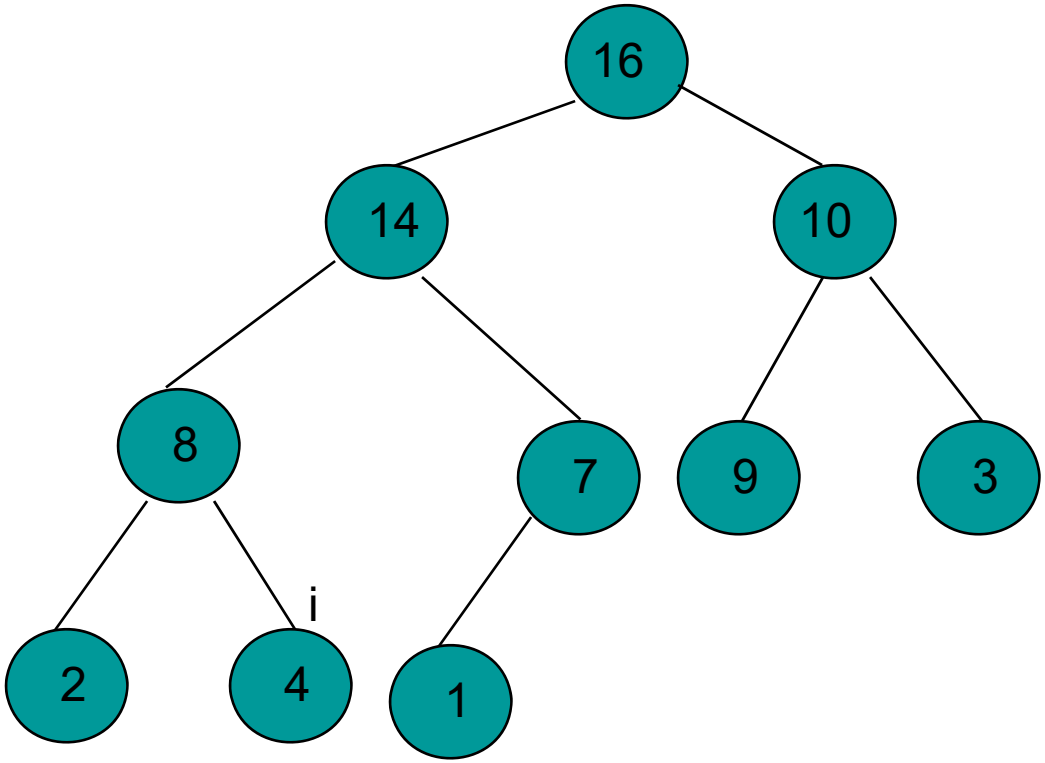
Any loops? No. So just sum up the times: $O(6) + O(\lg n)$

The dominant term is $O(\lg n)$.

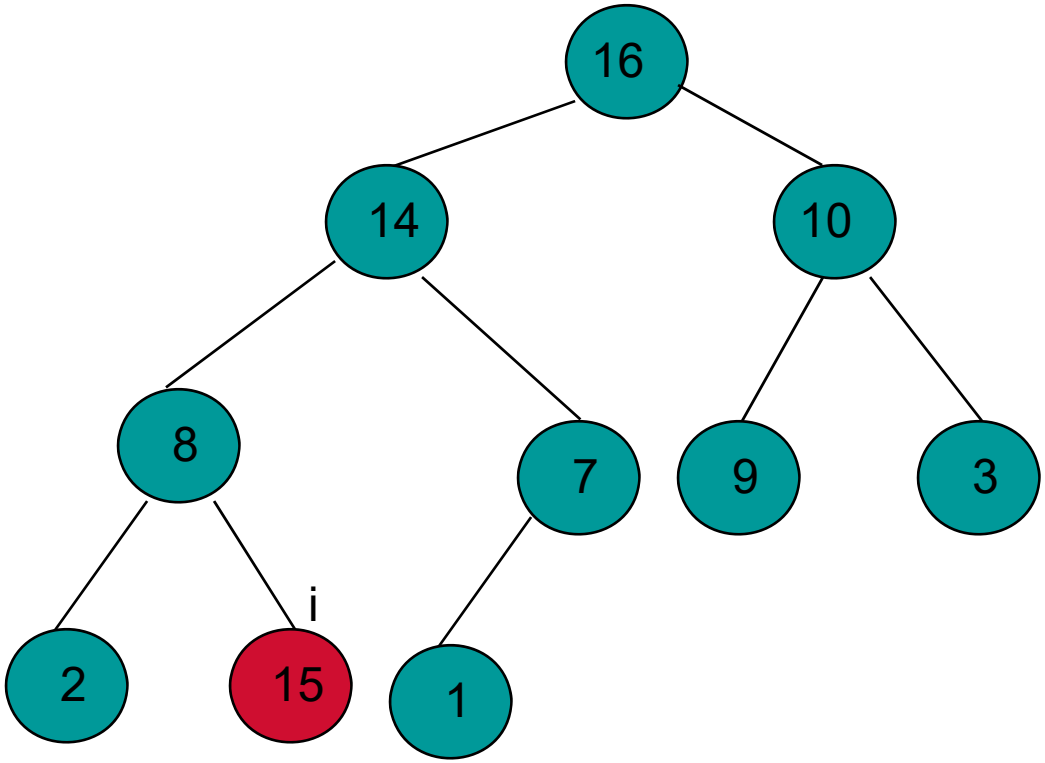
HEAP-INCREASE-KEY

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      then error "new key is smaller
         than current key"
3  A[i] ← key
4  while i > 1 and A[PARENT(i)] < A[i] do
5       exchange A[i] ↔ A[PARENT(i)]
6       i ← PARENT(i)
```

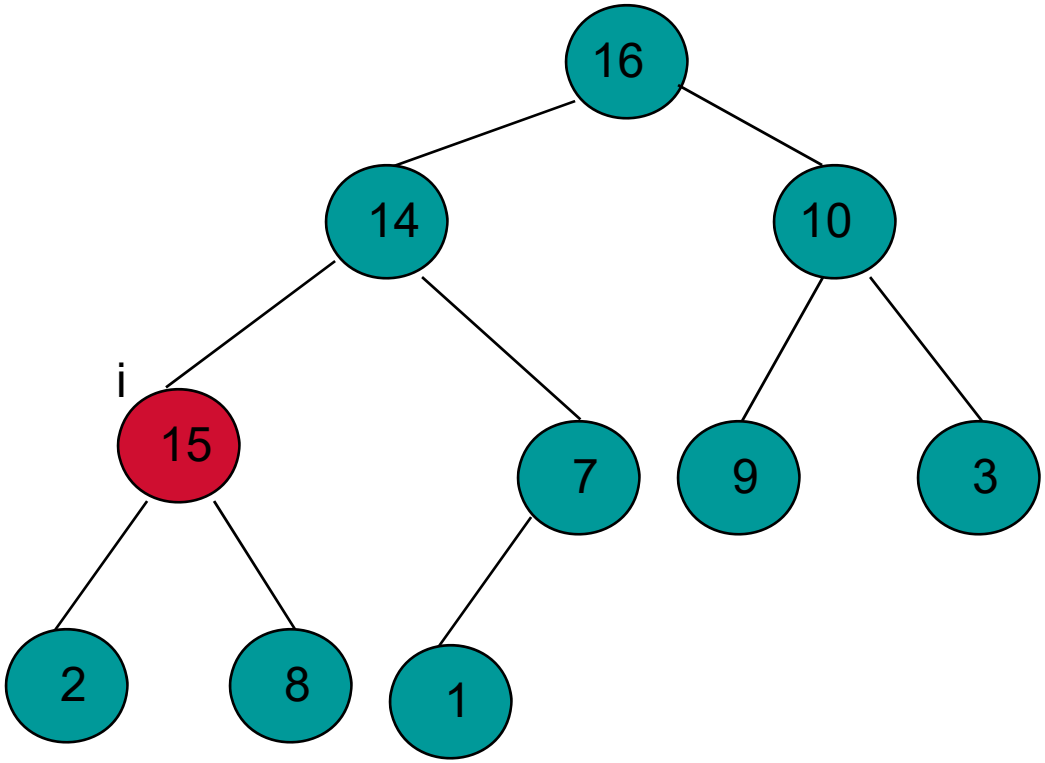
Example of HEAP-INCREASE-KEY



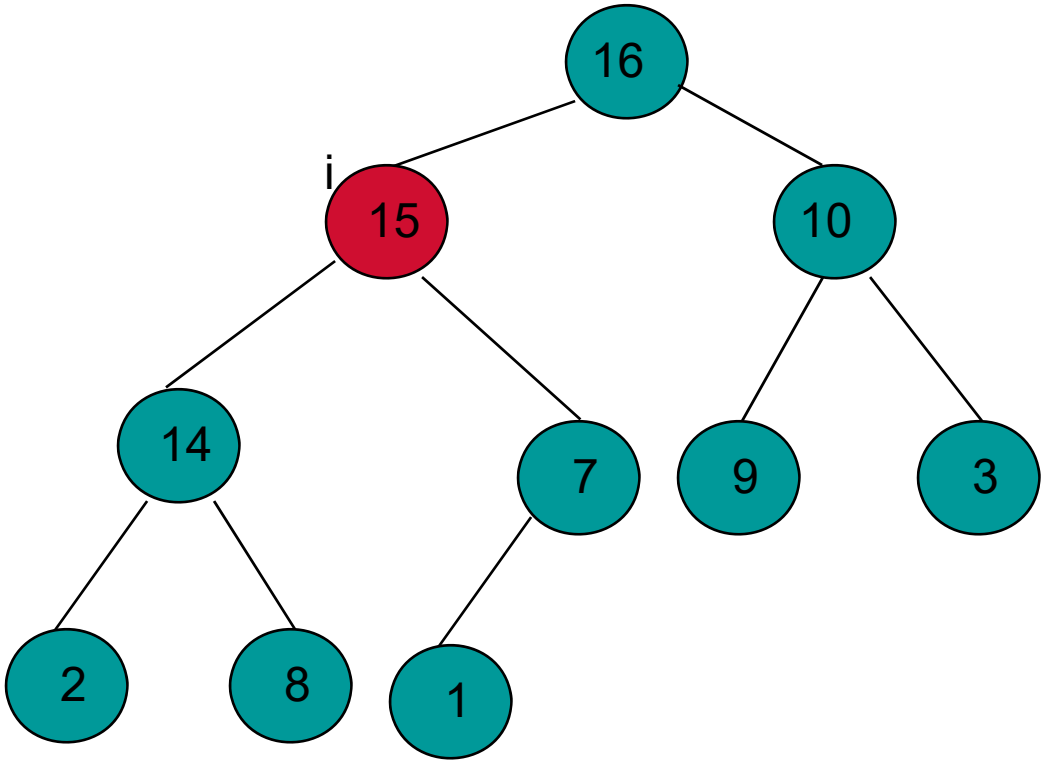
Example of HEAP-INCREASE-KEY (continued)



Example of HEAP-INCREASE-KEY (continued)



Example of HEAP-INCREASE-KEY (continued)



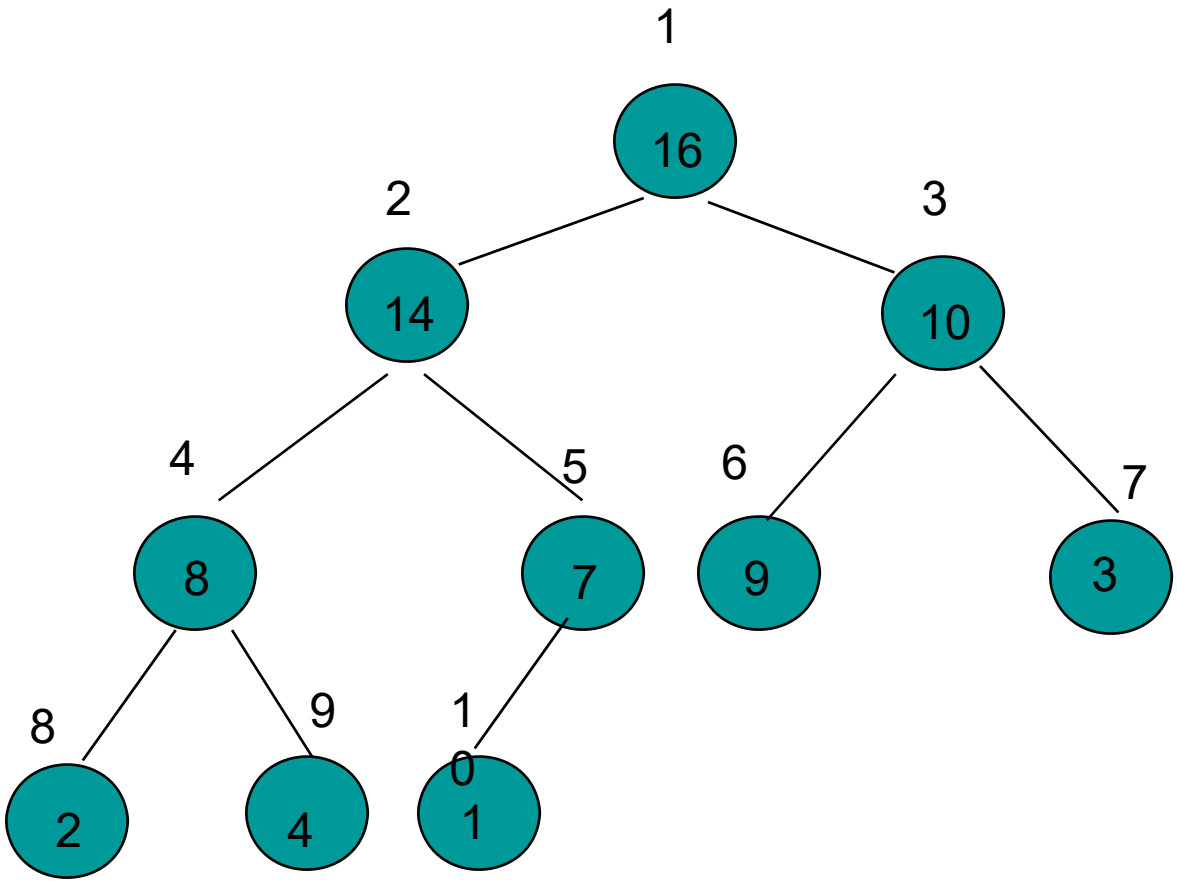
Running time of HEAP-INCREASE-KEY

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]                                          O(1)
2     then error                                          O(1)
           "new key is smaller than current key"
3  A[i] ← key                                             O(1)
4  while i > 1 and A[PARENT(i)] < A[i] do                 O(lg n)
5       exchange A[i] ↔ A[PARENT(i)]                      O(3)
6       i ← PARENT(i)                                     O(1)
```

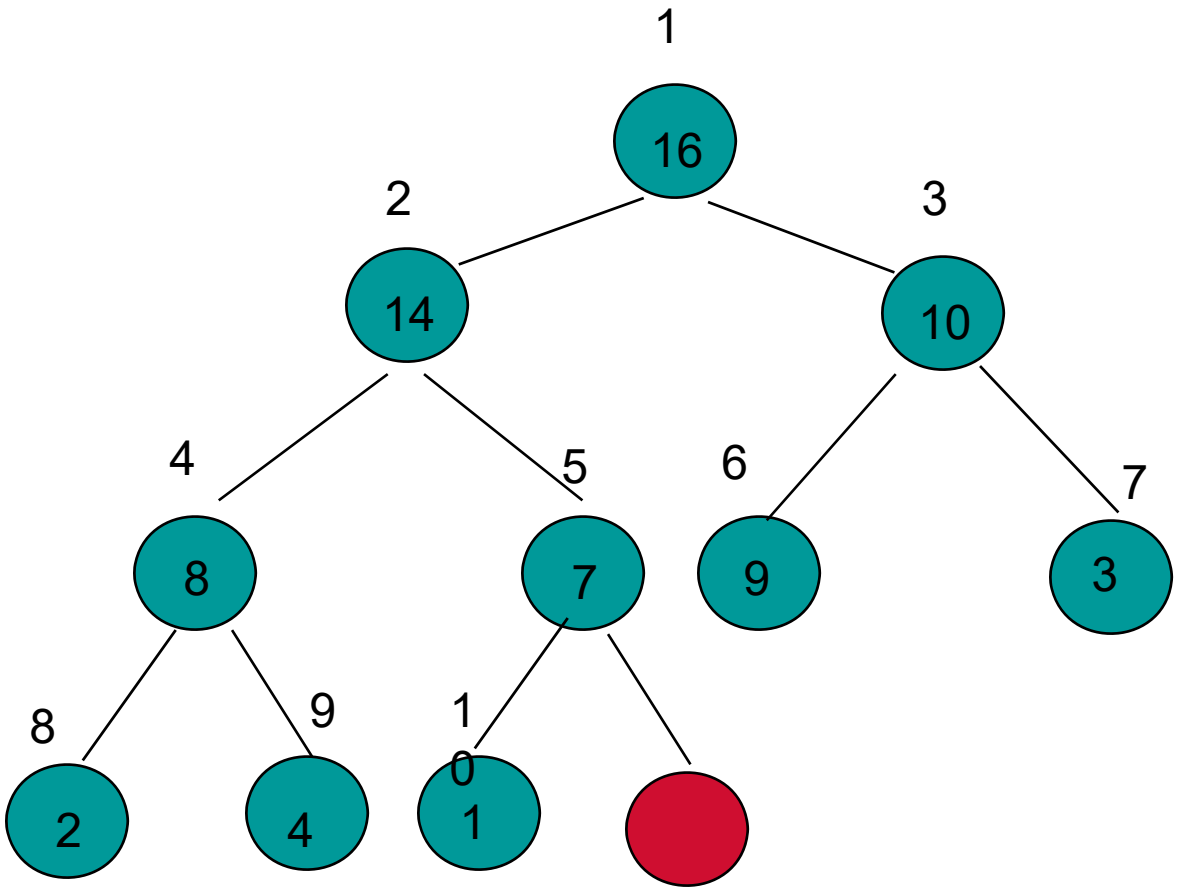
Any loops? Yes. How many times will the loop execute? As many times as node i has ancestors, which = the depth of the tree. The depth of a binary tree is O(lg n). We do a constant amount of work in the loop. Cost is: O(3) + O(4 lg n), or just O(lg n)

MAX-HEAP-INSERT

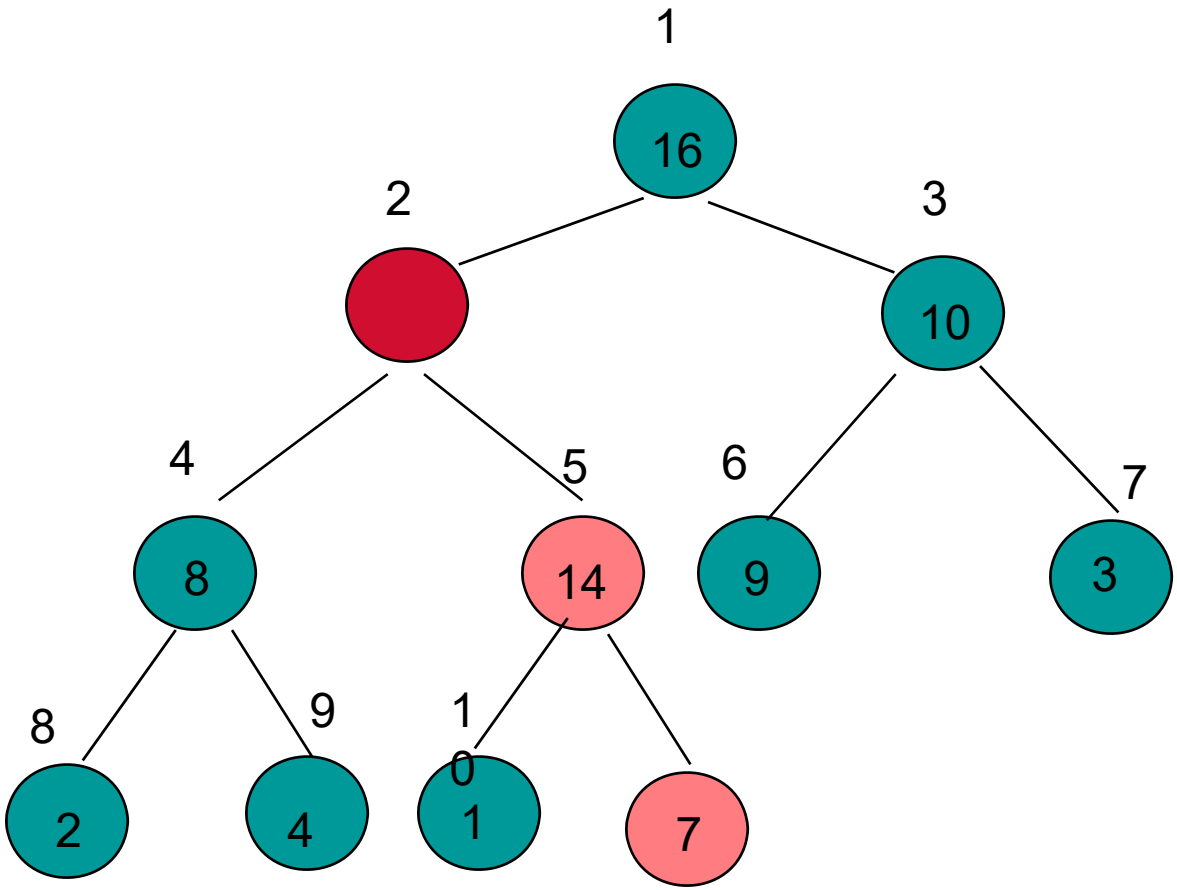
```
MAX-HEAP-INSERT(A,key)
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size] ← -∞
3  HEAP-INCREASE-KEY(A, heap-size[A], key)
```



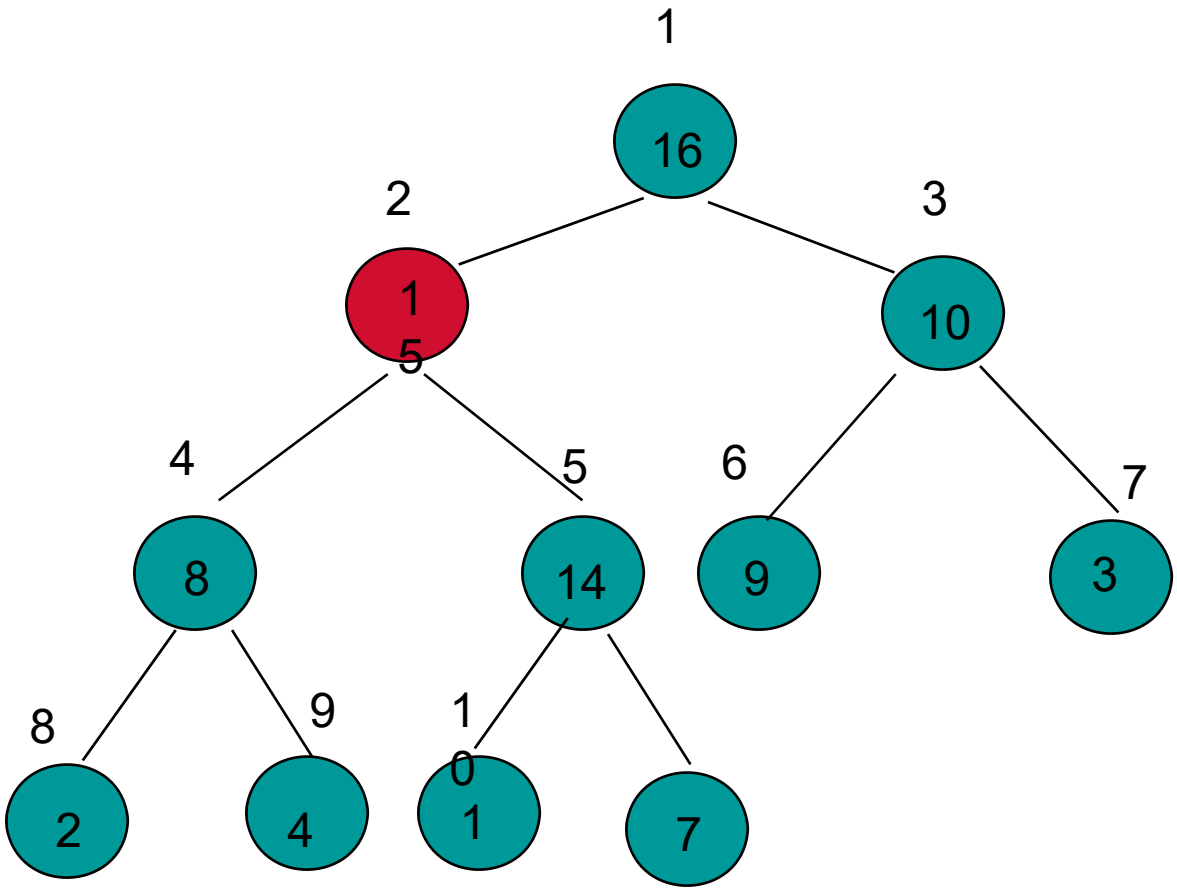
MAX-HEAP-INSERT(A,15)



HEAP-INSERT(A,15)



MAX-HEAP-INSERT(A,15)



MAX-HEAP-INSERT(A,15)

MAX-HEAP-INSERT

```
MAX-HEAP-INSERT(A,key)
1  heap-size[A] ← heap-size[A] + 1                  O(1)
2  A[heap-size] ← -∞                                O(1)
3  HEAP-INCREASE-KEY(A, heap-size[A], key)          O(lg n)
```

Any loops? No.

Add up the times: O(1) + O(1) + O(lg n) = O(2) + O(lg *n*)

Dominant term is O(lg *n*), so running time is just O(lg *n*).

Conclusion

We have seen:

- what a heap is
- how to build a heap
- how to use a heap for sorting
- how to analyze heapsort's running time
- how to use a heap for priority queues