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

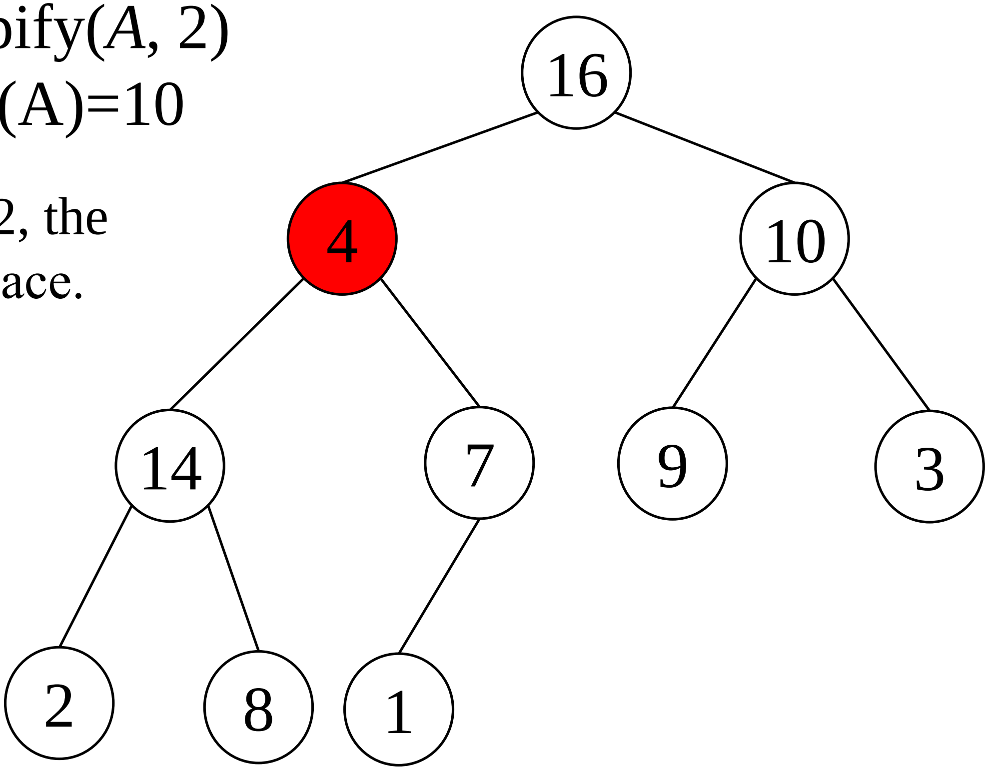
MaxHeapify(A,i)

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

MaxHeapify – Example

MaxHeapify(*A*, 2)
Heapsize(A)=10

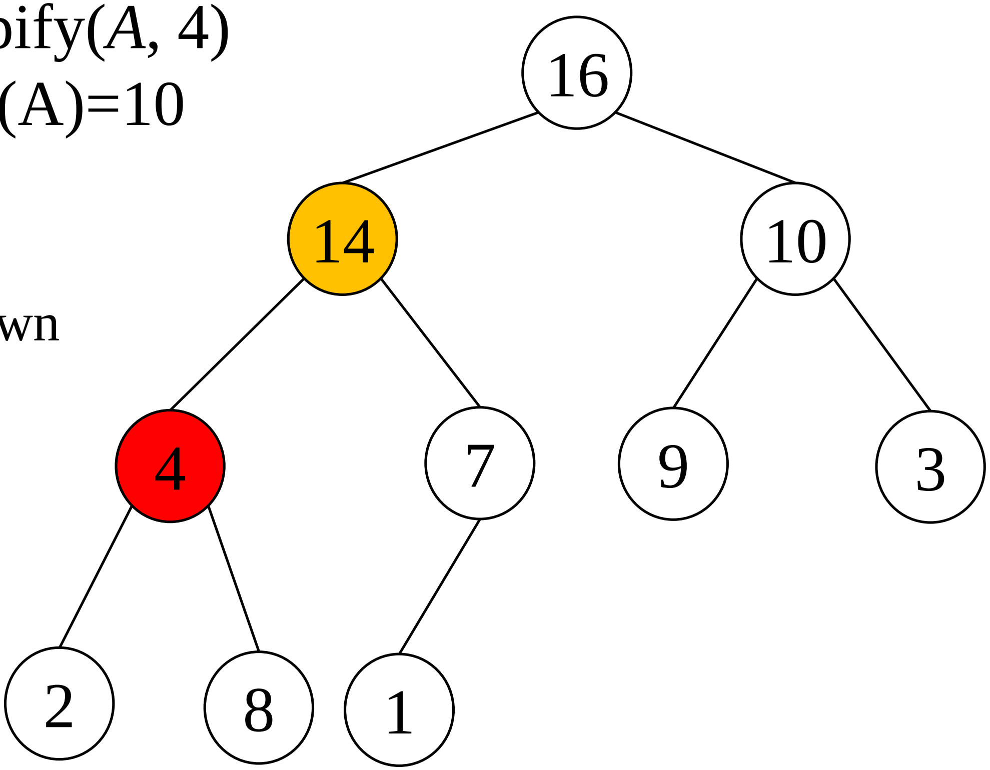
Array element 2, the "4", is out of place.



MaxHeapify – Example

MaxHeapify(A, 4)
Heapsize(A)=10

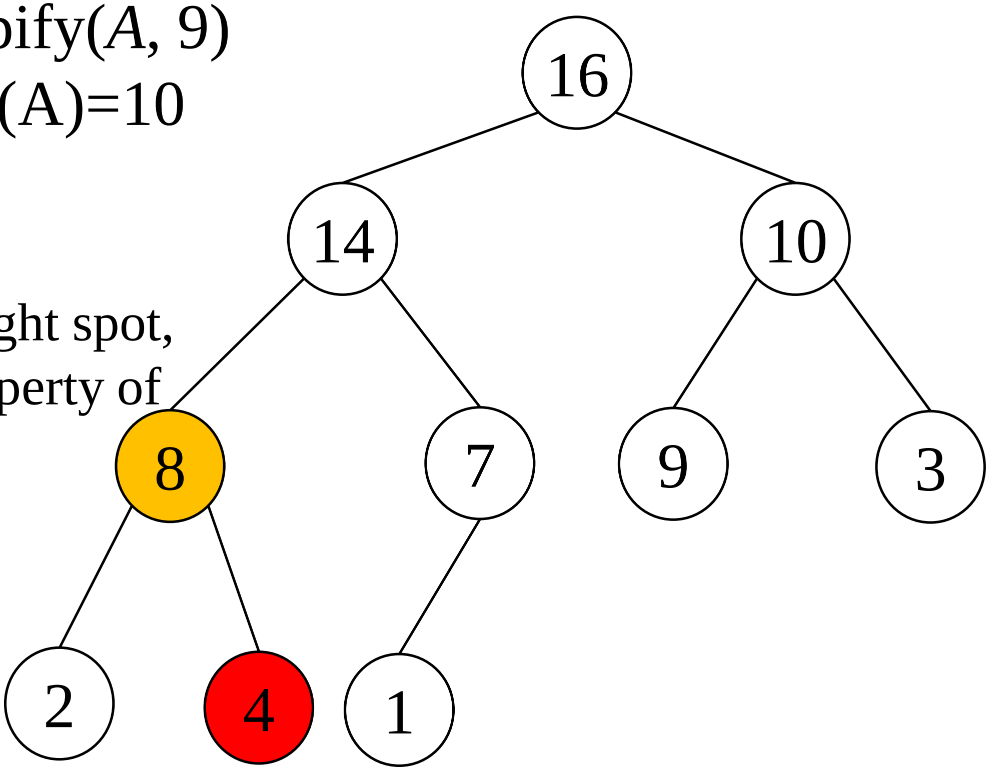
Moving the 4 down



MaxHeapify – Example

MaxHeapify(A, 9)
Heapsize(A)=10

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



MaxHeapify

MaxHeapify(*A*, *i*)

$l \leftarrow$ left(i)
$r \leftarrow$ right(i)
if $l \leq$ *heap-size*[A] and $A[l] > A[i]$
    then *largest* $\leftarrow l$
    else *largest* $\leftarrow i$
if $r \leq$ *heap-size*[A] **and** $A[r] > A[$*largest*$]$
    then *largest* $\leftarrow r$
if *largest* $\neq i$
    then exchange $A[i] \leftrightarrow A[$*largest*$]$
        MaxHeapify(*A*, *largest*)

Assumption:
Left(i) and Right(i) are max-heaps.

Running time of MaxHeapify

<u>*MaxHeapify*(*A*, *i*)</u>

```
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.             MaxHeapify(A, largest)
```

Line1~Line9:
Time to fix node *i* and its children = ***Θ(1)***

+

let h(i) be the height of node i
at most h(i) recursion levels ⇨ ***O(logn)***

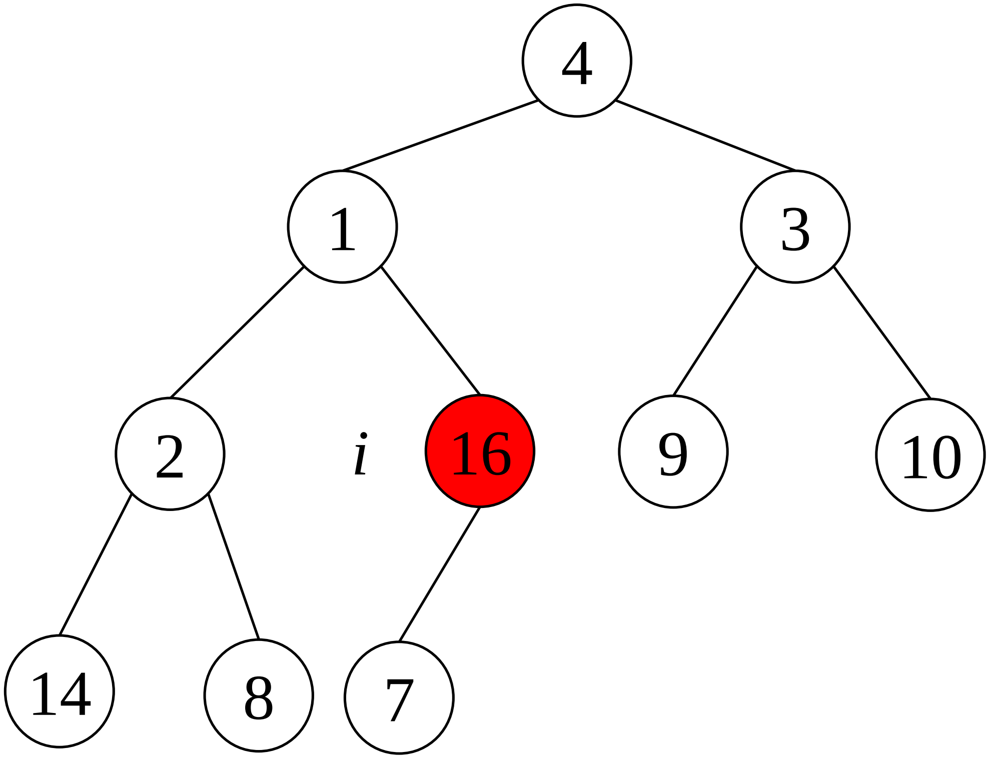
Running time of MaxHeapify is O(*logn*) or O(*h*)

BUILD-MAX-HEAP

- Use MaxHeapify 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.
- MaxHeapify is called on all internal nodes.

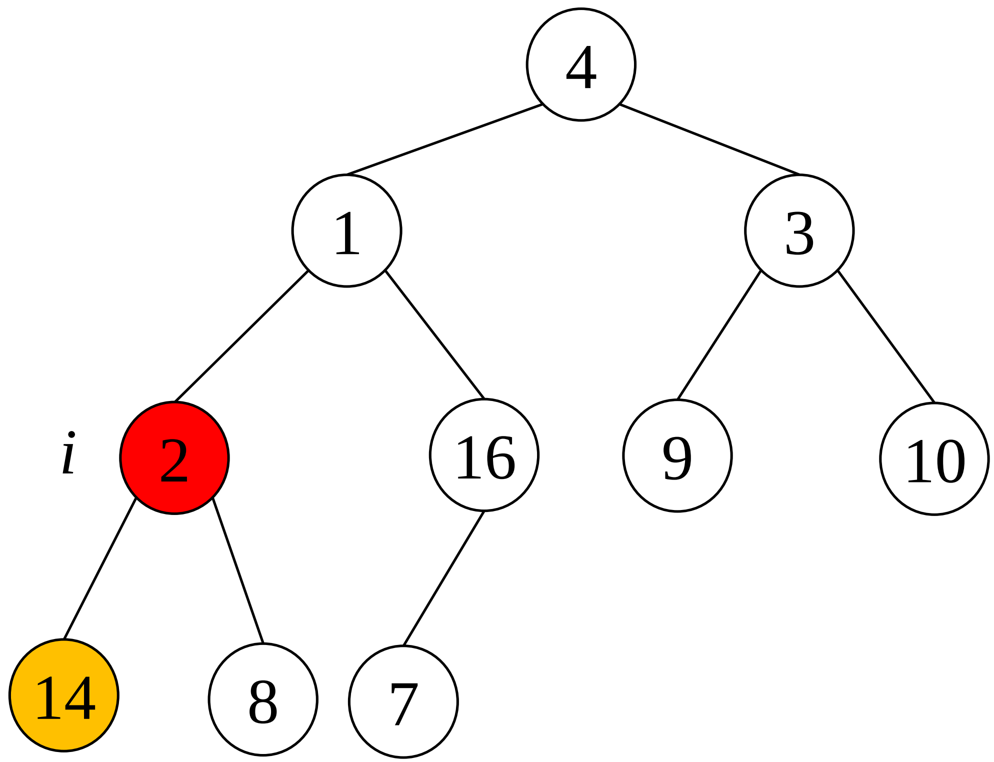
BUILD-MAX-HEAP

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

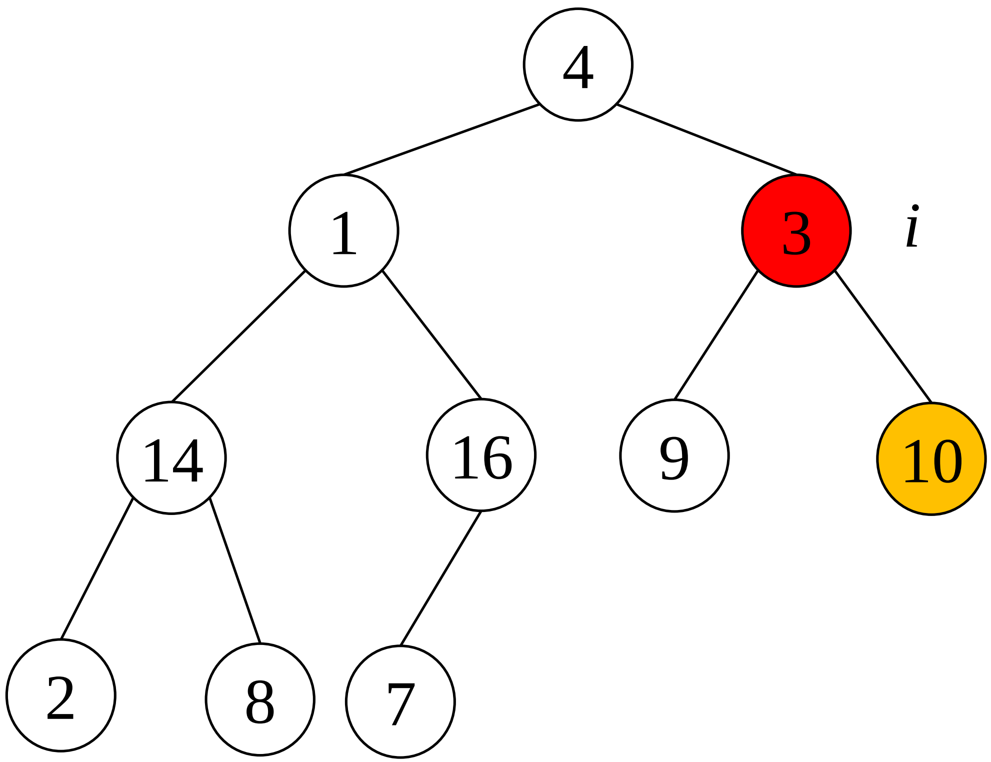


length(A) = 10
floor(length(A)/2) = 5
process from 5 to 1

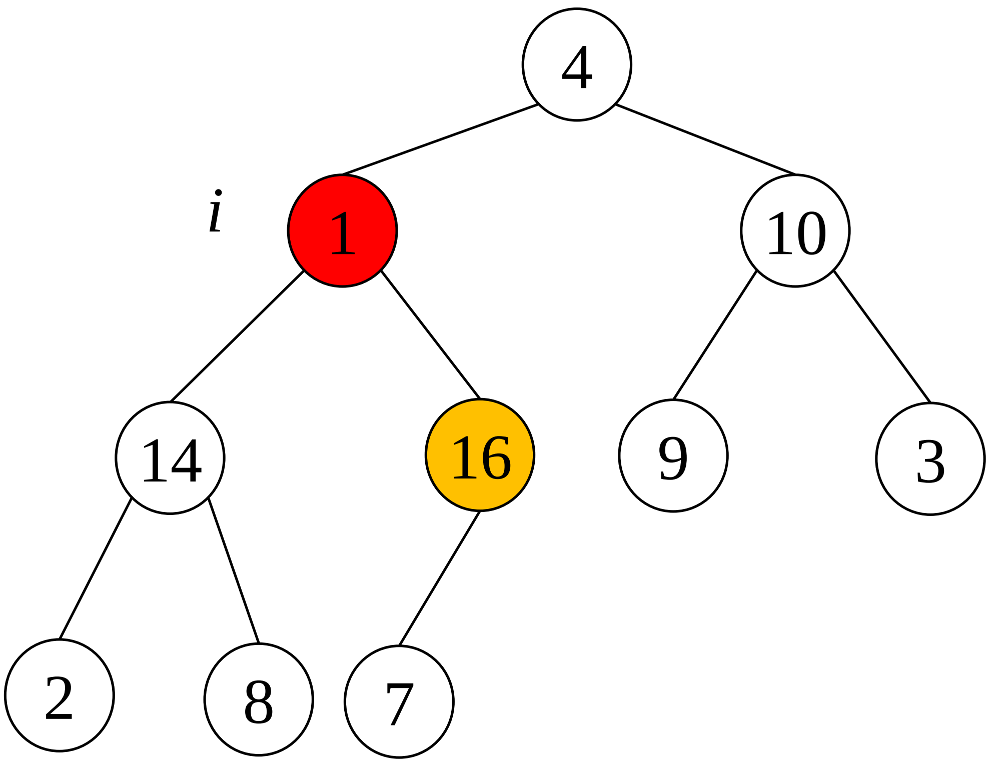
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



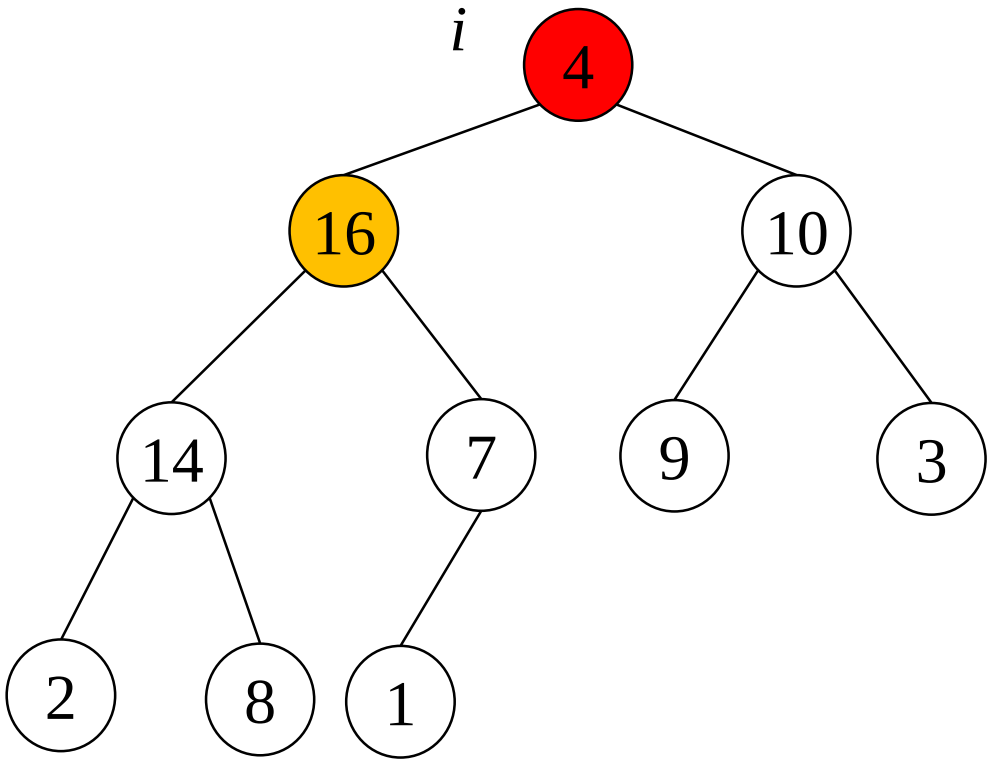
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



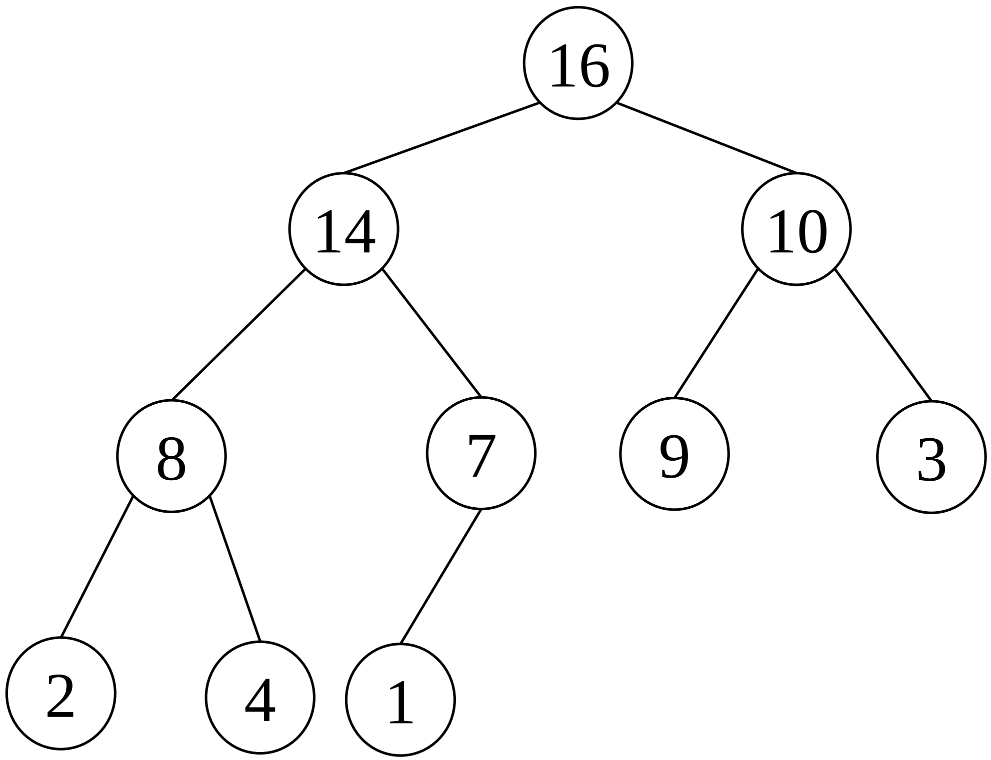
1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7



1	2	3	4	5	6	7	8	9	10
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

Running Time of BUILD-MAX-HEAP

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

Tighter Bound for BUILD-MAX-HEAP

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

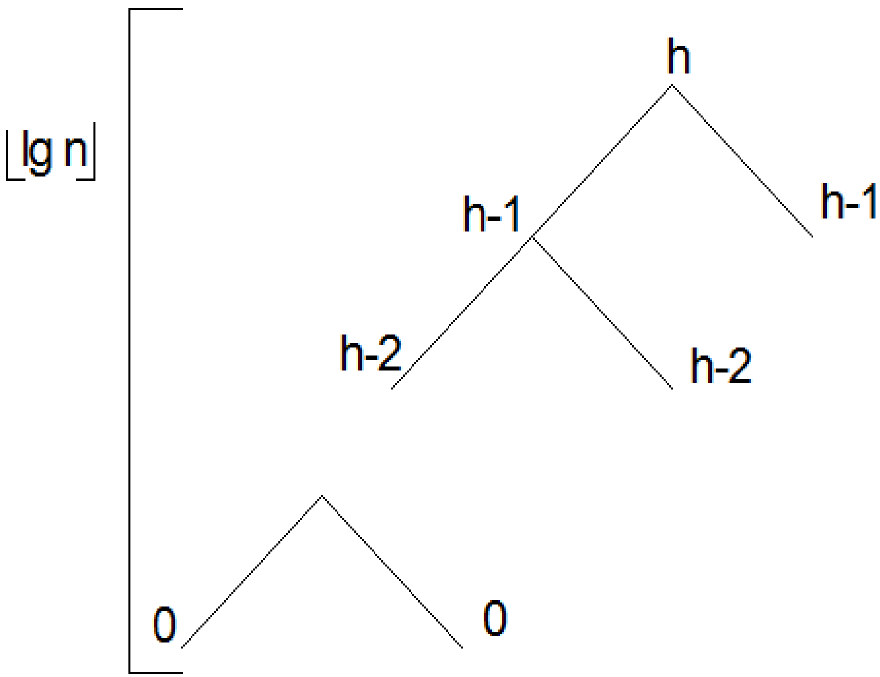
$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \le \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

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

Thus the running time is bounded by <u>O(*n*)</u>

<u>Therefore, we can build a heap from an unordered array in linear time</u>

Recursion tree method for calculating Build-Max-Heap cost



Height	Nodes	Cost
h	2^0	$h*2^0$
h-1	2^1	$(h-1)*2^1$
h-2	2^2	$(h-2)*2^2$
0	2^h	$0*2^h$

$$\sum_{h=0}^{\lfloor \lg n \rfloor} h * 2^{\lfloor \lg n \rfloor - h} = \sum_{h=0}^{\lfloor \lg n \rfloor} h * \frac{2^{\lfloor \lg n \rfloor}}{2^h} \le \sum_{h=0}^{\lfloor \lg n \rfloor} h * \frac{2^{\lg n}}{2^h} = \sum_{h=0}^{\lfloor \lg n \rfloor} h * \frac{n^{\lg 2}}{2^h}$$

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

Heap Sort

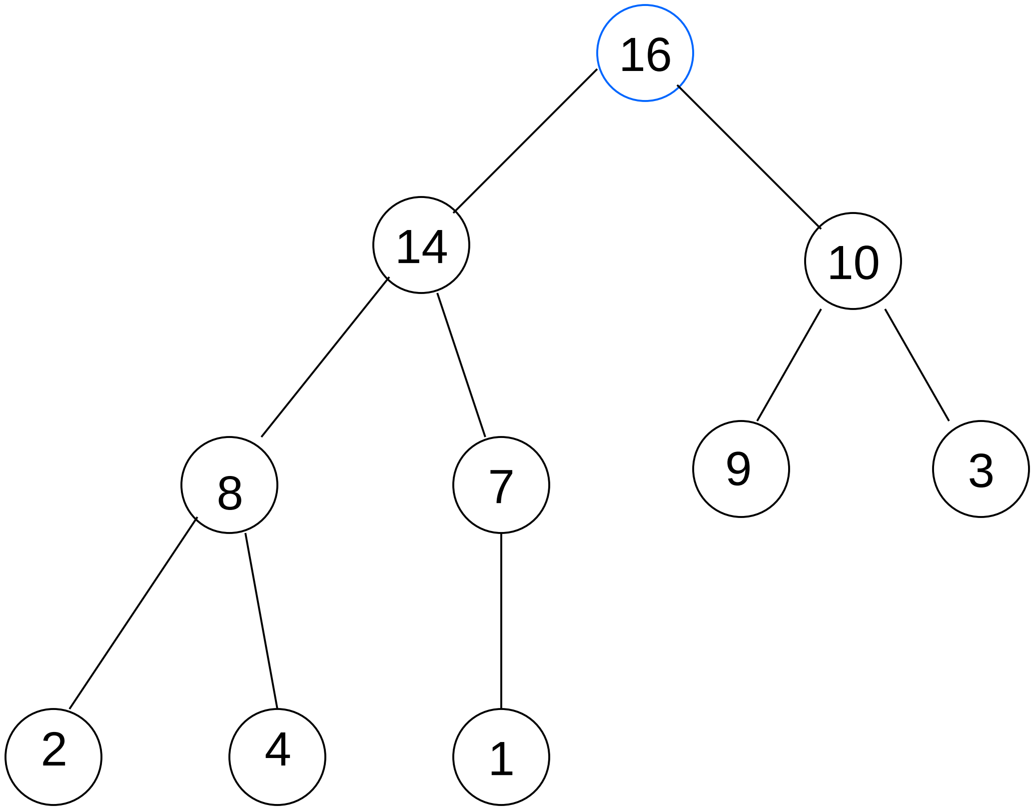
- First build a heap.
- Swap the root (the maximum element) with the last element in the array
- “Discard” this last node by decreasing the heap size
- Call MaxHeapify on the new root
- Repeat this process until only one node remains

Heap Sort

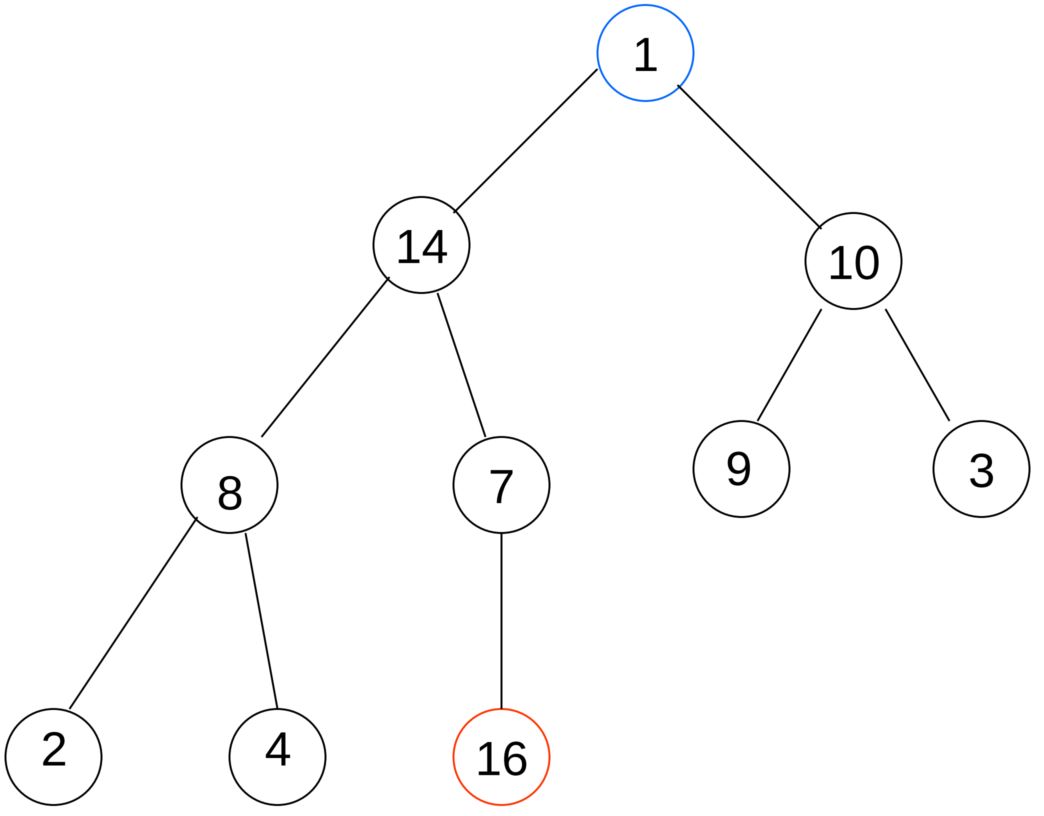
```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
        exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] – 1
        MaxHeapify(A, 1)
```

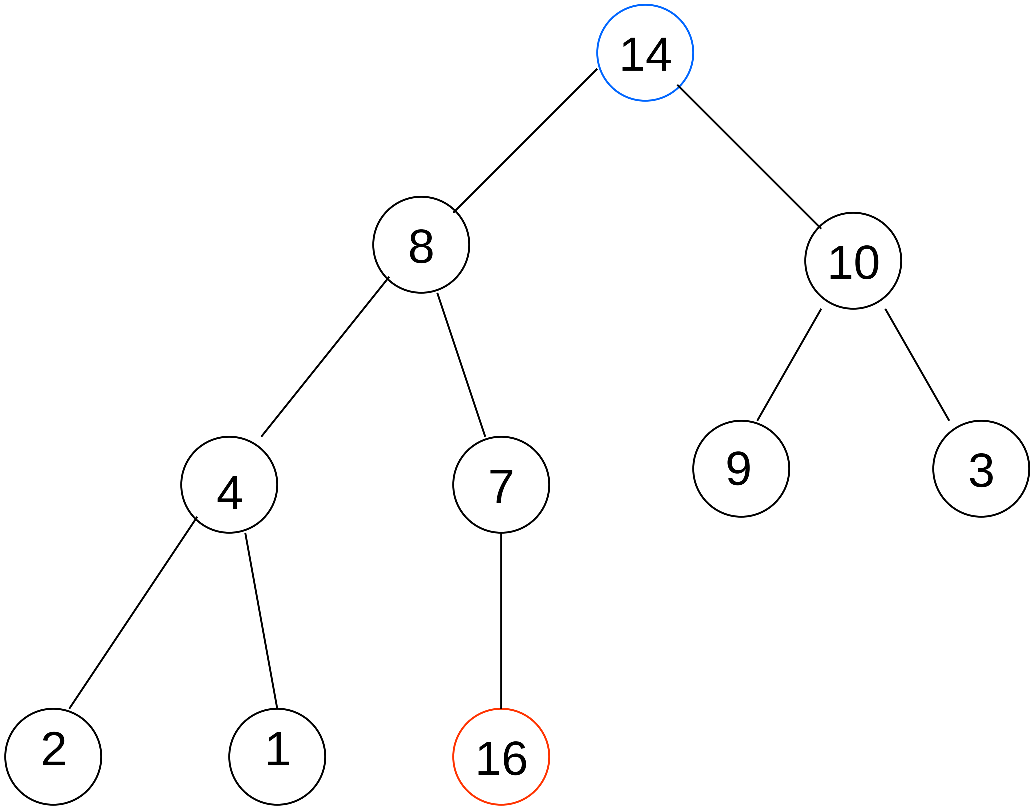
Input: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

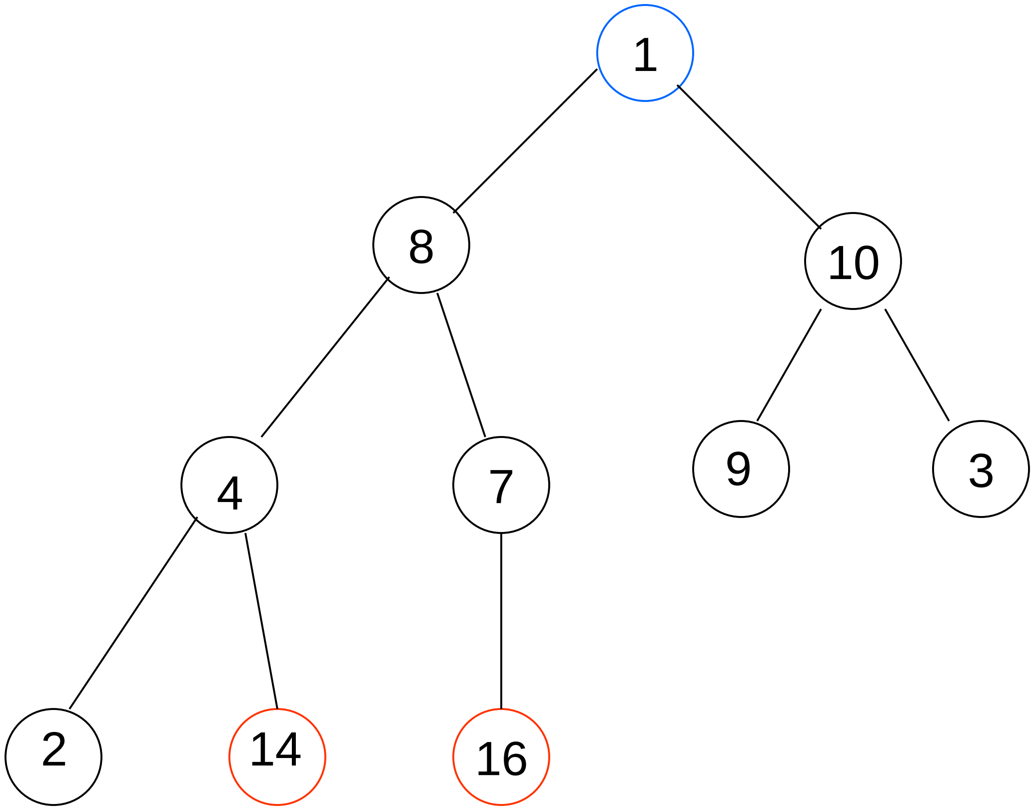
Build a max-heap

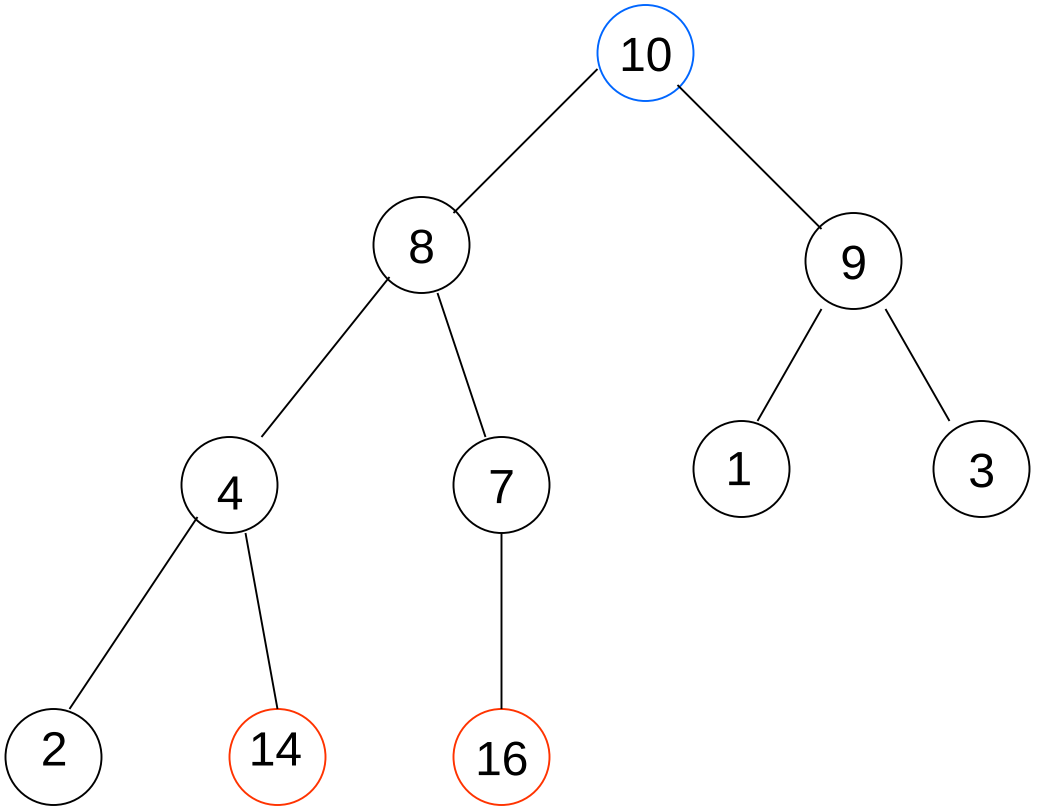


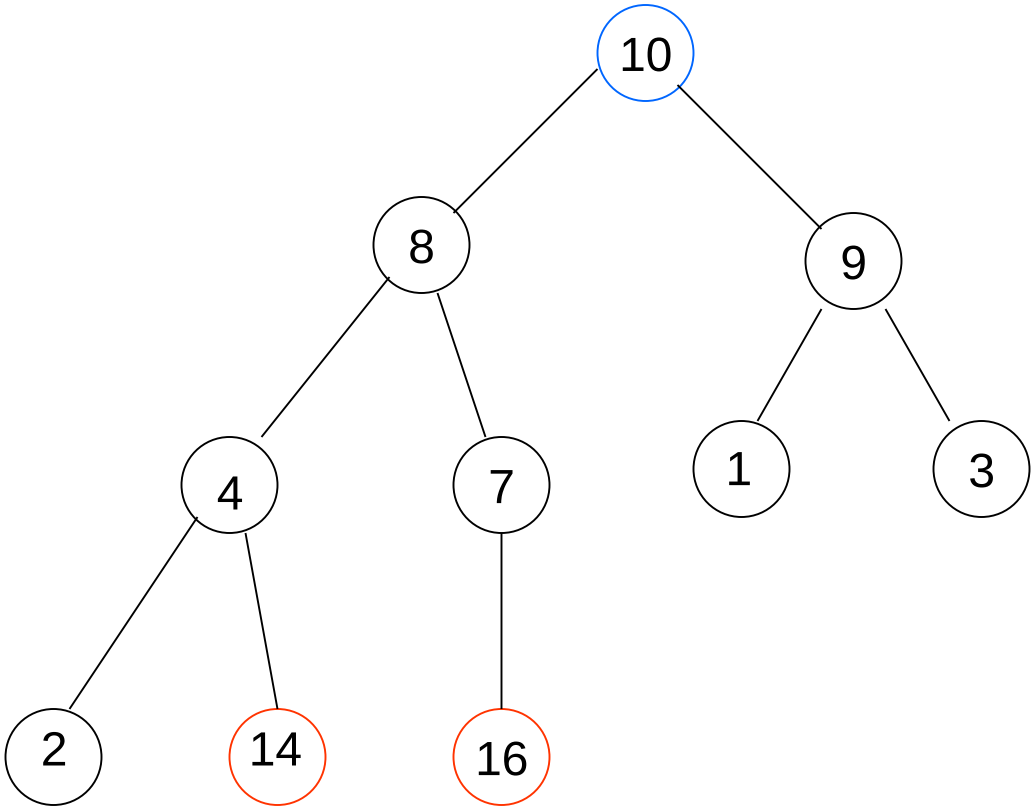
16, 14, 10, 8, 7, 9, 3, 2, 4, 1


1
14
10
8
7
9
3
2
4
16

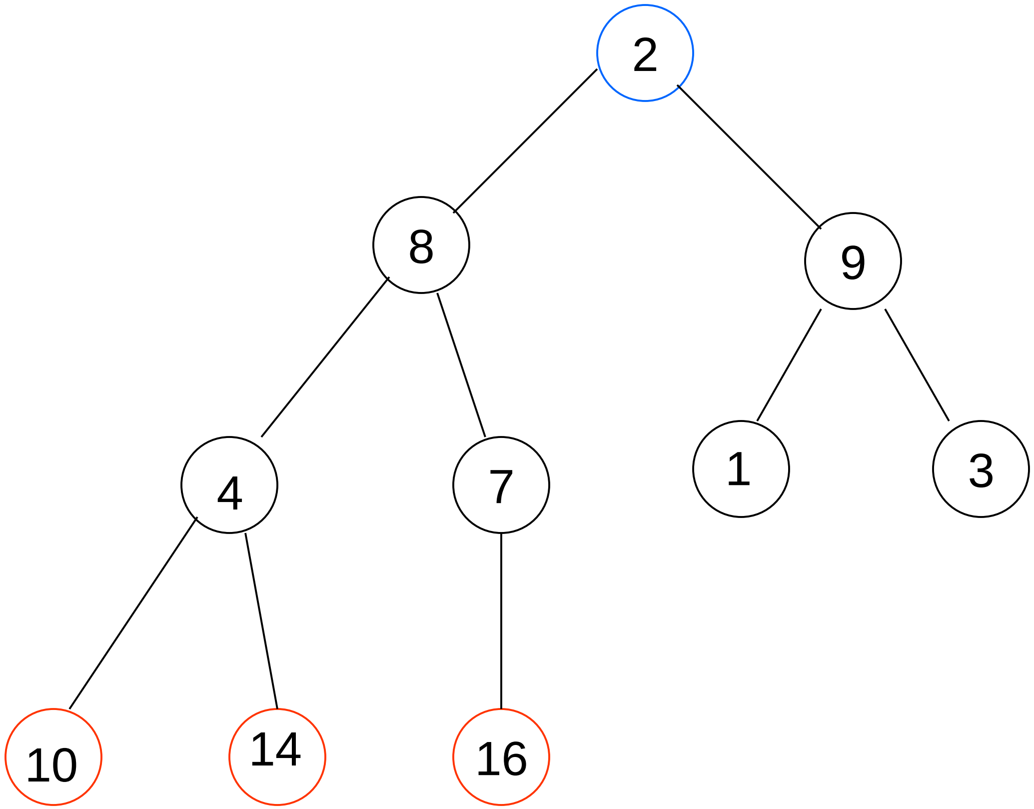

14
8
10
4
7
9
3
2
1
16

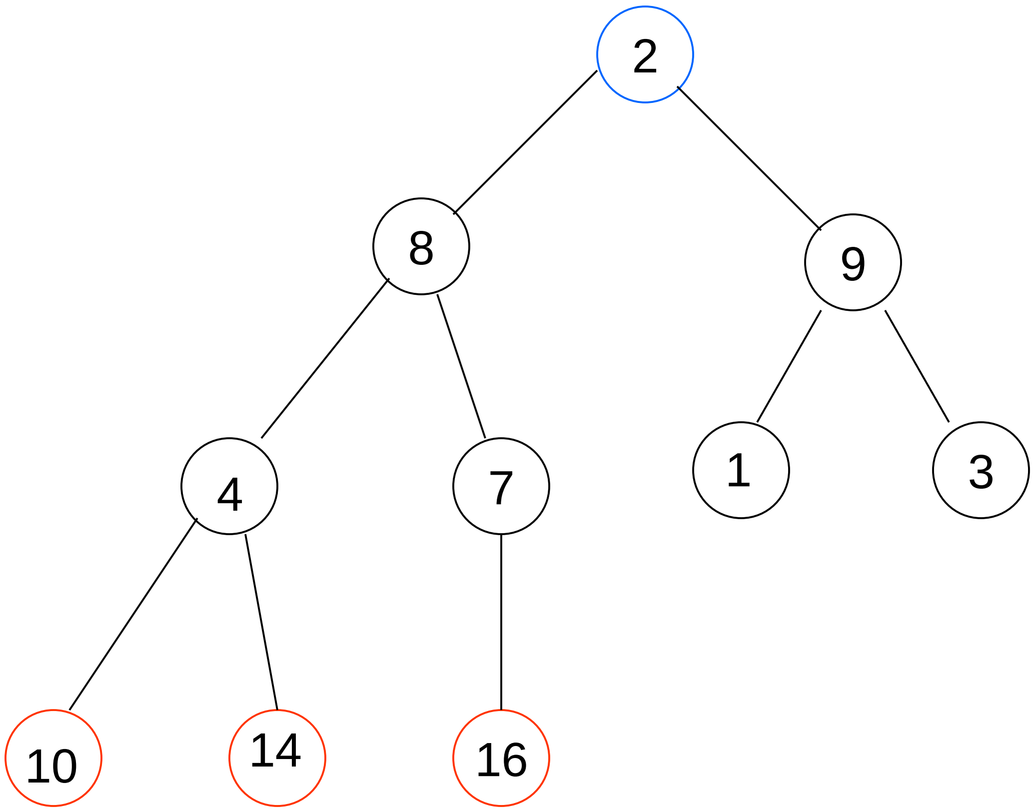

1
8
10
4
7
9
3
2
14
16


10
8
9
4
7
1
3
2
14
16



10, 8, 9, 4, 7, 1, 3, 2, 14, 16


2
8
9
4
7
1
3
10
14
16



2, 8, 9, 4, 7, 1, 3, 10, 14, 16

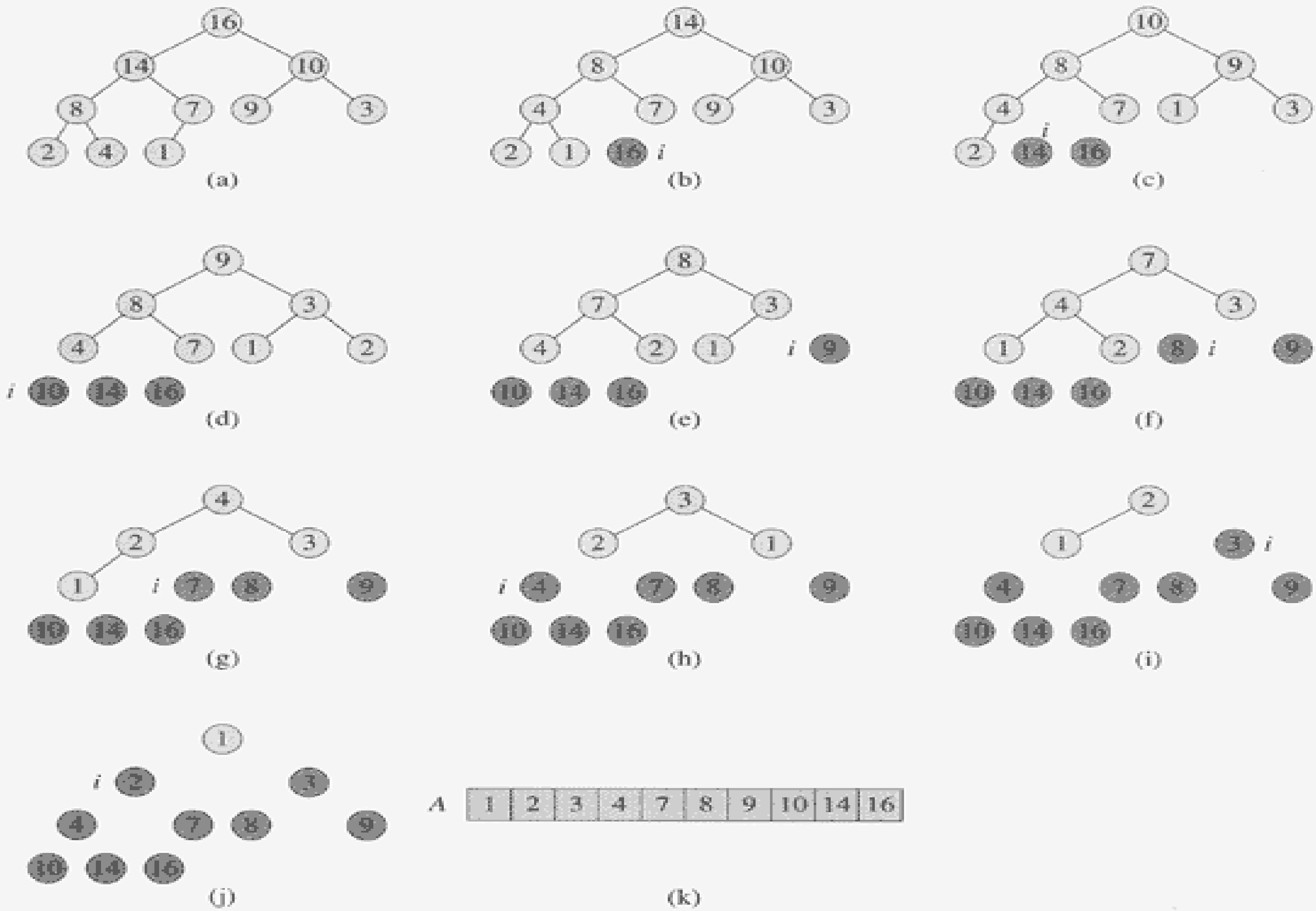


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after it has been built by BUILD-MAX-HEAP. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5. The value of i at that time is shown. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

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(log n)
```

- Total time is:
  O(n) + O(n-1) * [O(1) + O(1) + O(log n)]
  which is approximately O(*n) + O(nlogn)*
  or just <u>O(*nlogn)*</u>

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(log *n*) time*.*
- Total time is O(*nlogn*)*.*

Space requirements of Heapsort

- Heapsort uses heap 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