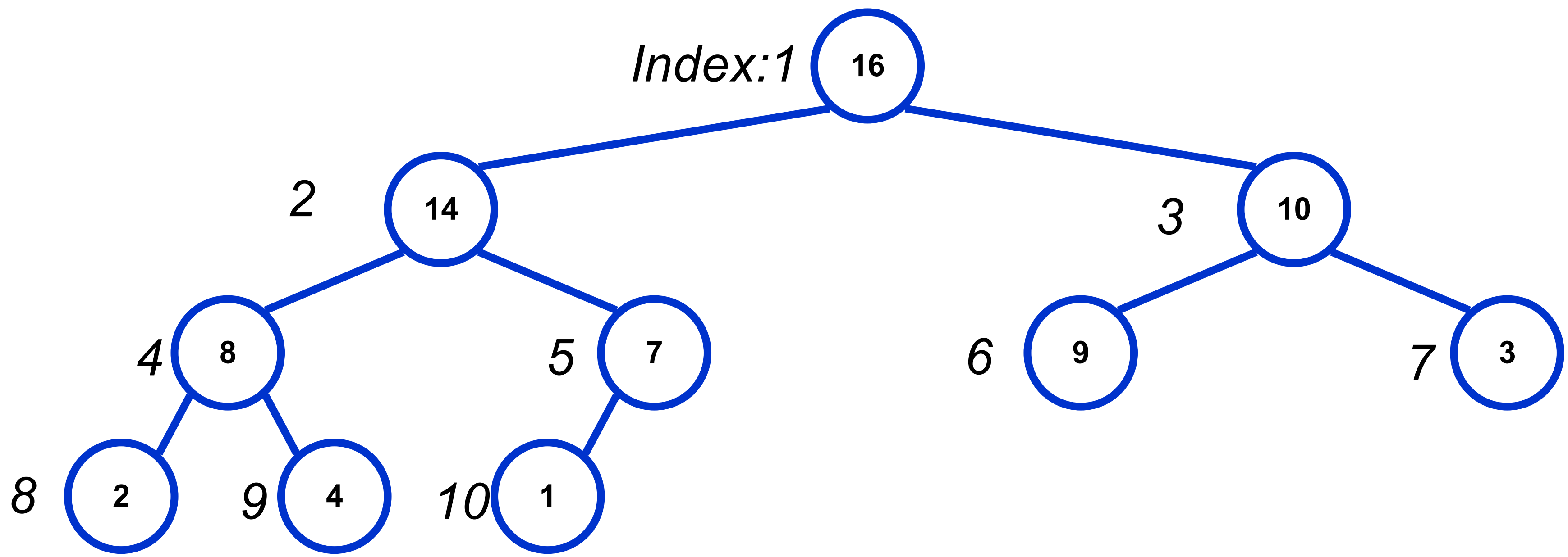
Heap and Heapsort

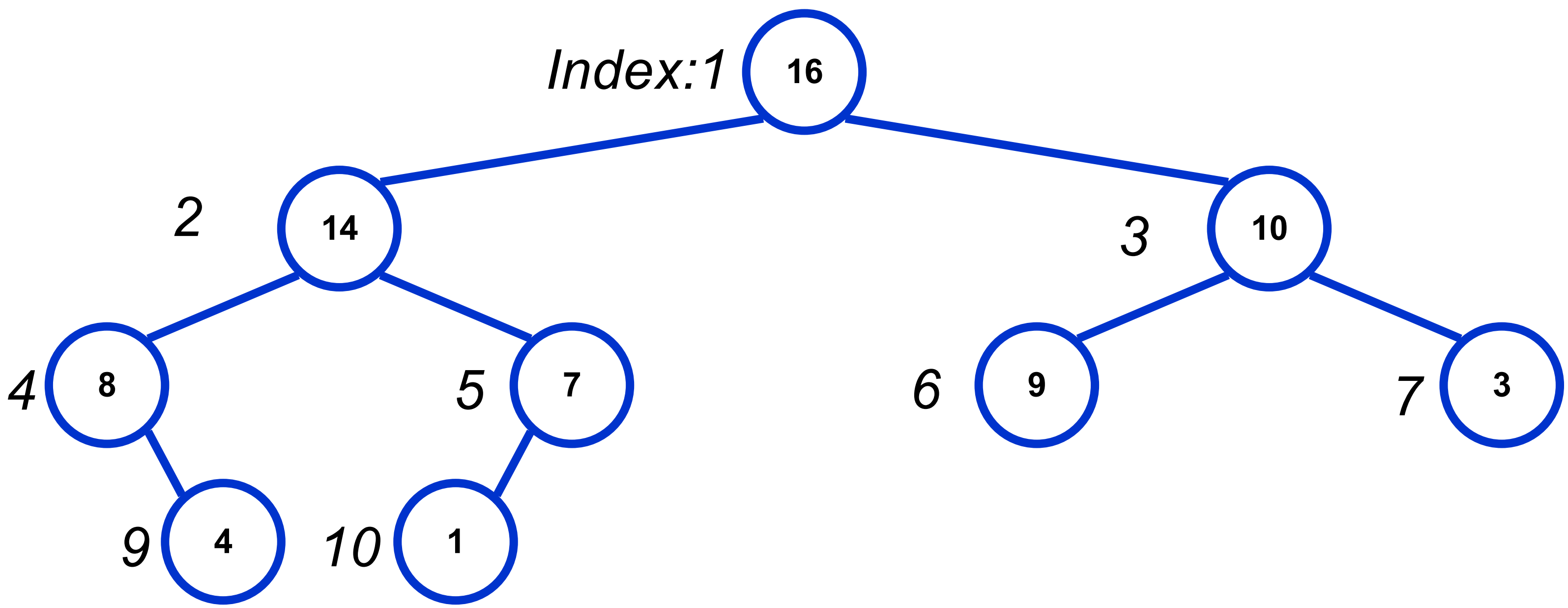
Heaps

- A *heap* can be seen as a complete binary tree:

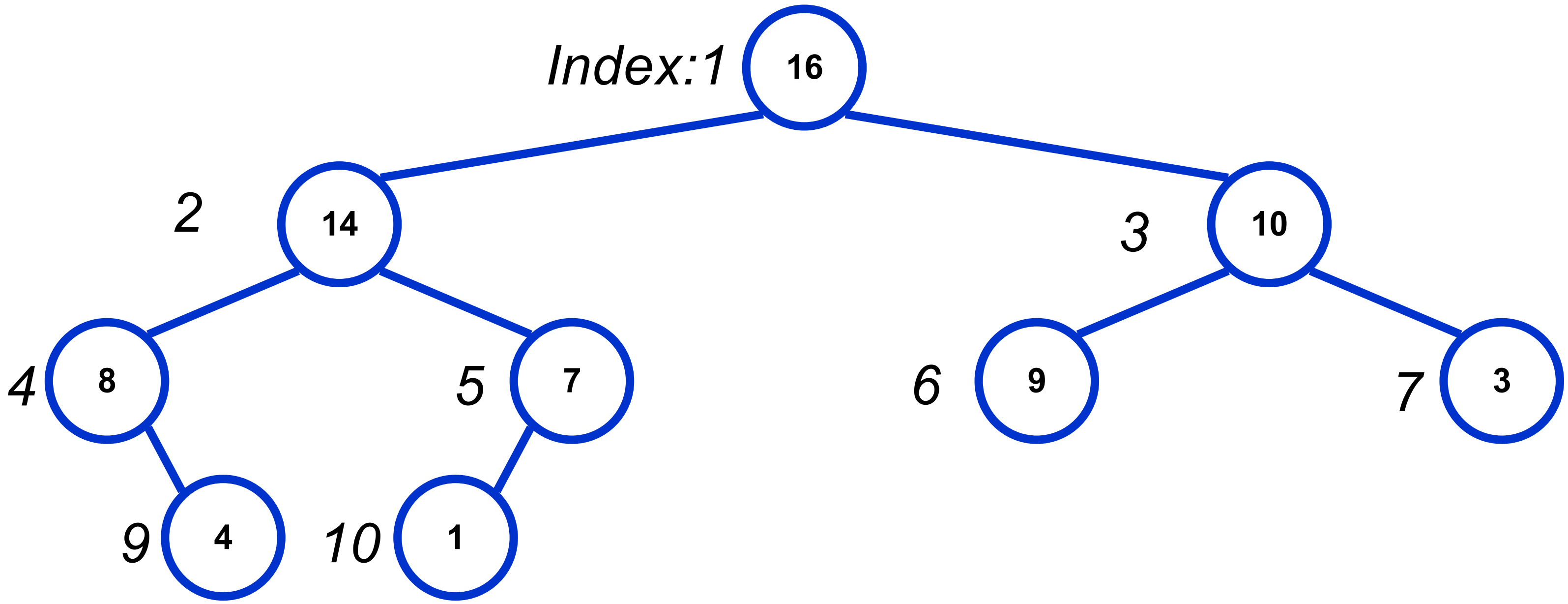


  - *What makes a binary tree complete?* Top-down left to right indexing: no index will be missing
  - *Is the example above complete?*

Is this a complete binary tree?

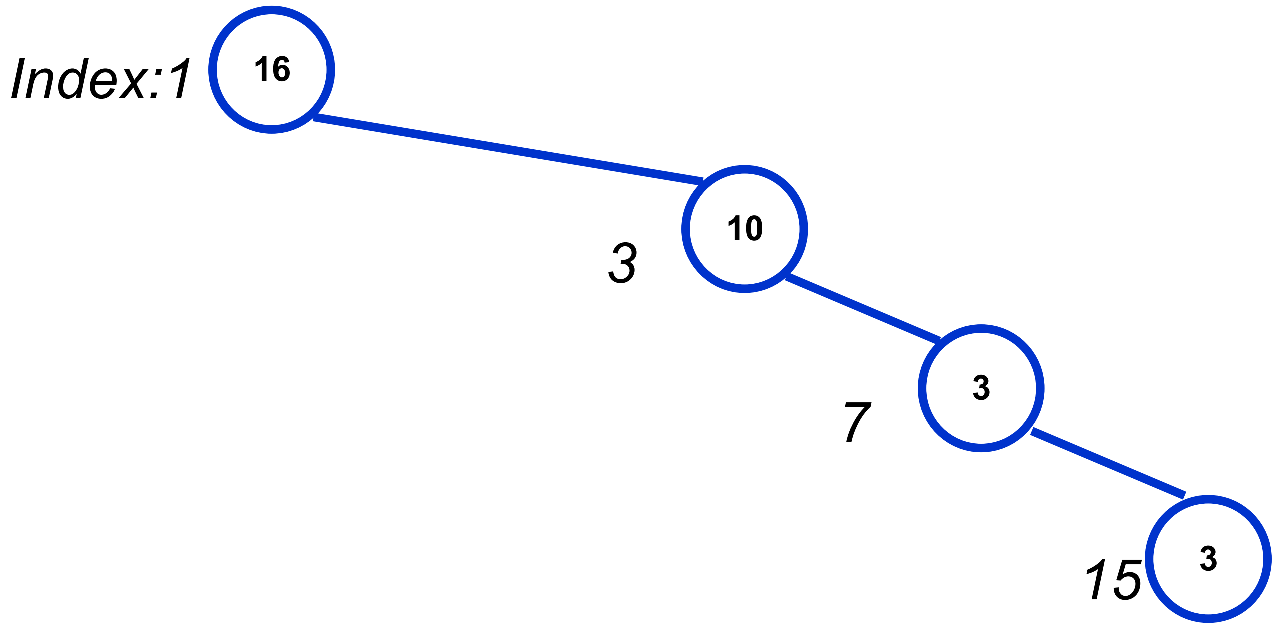


Is this a complete binary tree?



- *No*
- *Index 8 is missing*

Is this a Complete Binary Tree (n=4)?



Indices to put in an array: 1, 3, 7, 15 (2^4-1)

Array needs O(2^n) spaces if not complete binary tree -->Use pointer/list to represent

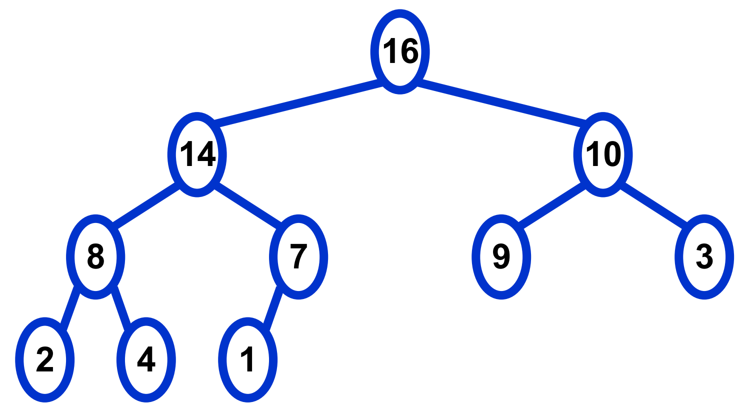
Heaps

- In practice, heaps are usually implemented as arrays as we need *n* spaces (due to complete binary tree property).

$A =$

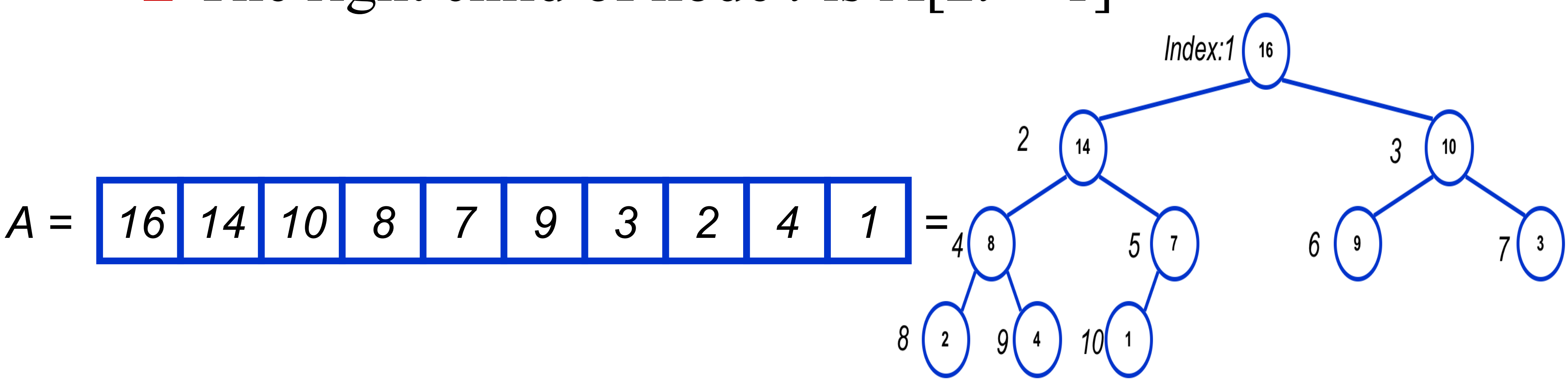
16	14	10	8	7	9	3	2	4	1

$=$



Heaps

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node i is A[i]
  - The parent of node i is A[i/2] (note: integer divide)
  - The left child of node i is A[2i]
  - The right child of node i is A[2i + 1]



Referencing Heap Elements

- So…

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

The Heap Property

- Heaps also satisfy the *heap property*:

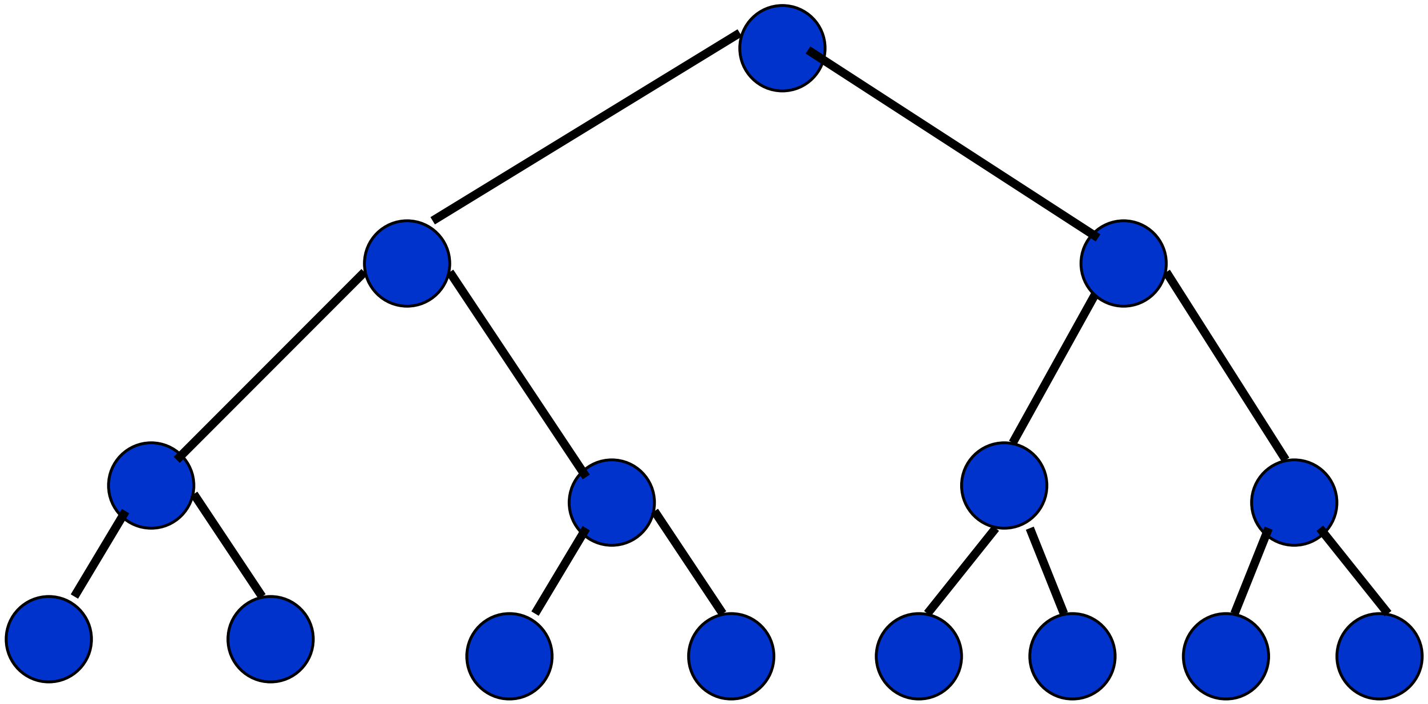
  $A[Parent(i)] \geq A[i]$ for all nodes $i > 1$

  - In other words, the value of a node is at most the value of its parent, for Max Heap.
  - Min heap maintains $\leq$ relation.
  - *Where is the largest element in a (max)heap stored?*

Heap Height

- Definitions:
  - The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
  - The height of a tree = the height of its root
- *What are the minimum and maximum numbers of nodes in a heap of height h?*
- *What is the height of an n-element heap? Why?*
  - Basic heap operations take at most time proportional to the height of the heap

Maximum Number of Nodes



Maximum number of nodes (start with h=0)

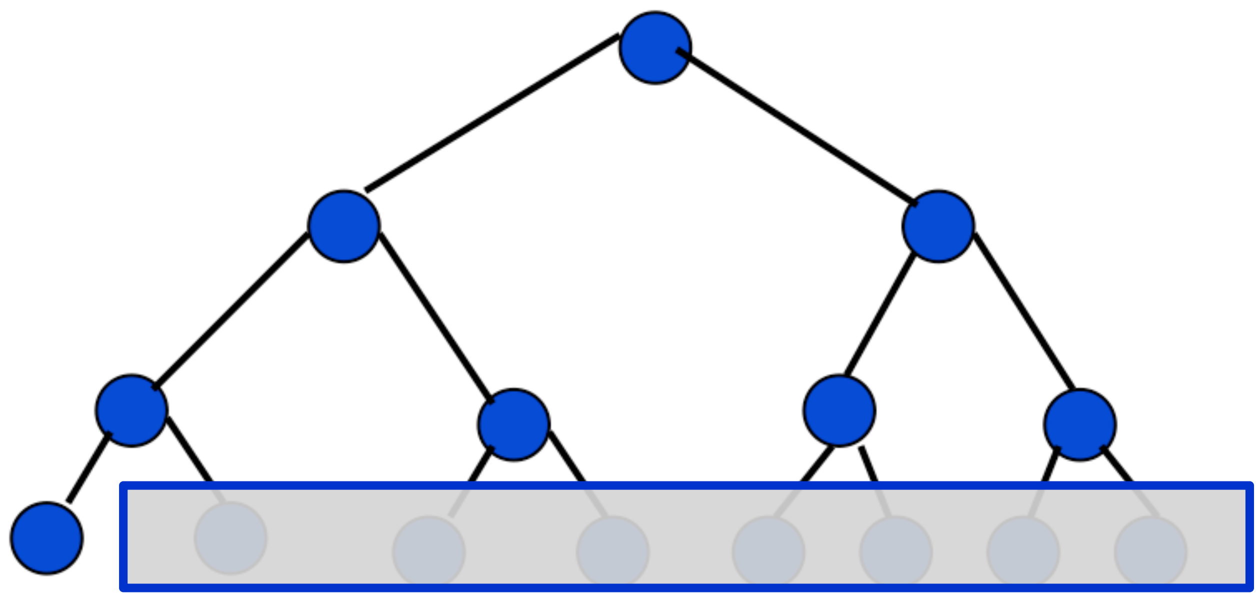
$= 1 + 2 + 4 + 8 + \ldots + 2^h$

$= 2^{h+1} - 1.$ (if start with h=1, then $2^h - 1$)

Minimum Number of Nodes

- For the minimum, the last level should contain a single node.
- Which means it is full up to a height of h-1

$$(2^h - 1) + 1$$
$$= 2^h$$



Height of Heap

Number of nodes = n.

Min num. of nodes $\leq$ n $\leq$ Max num. of nodes

$2^h \leq n \leq (2^{h+1} - 1)$

$2^h \leq n < 2^{h+1}$

$h \leq \log n < h+1$

$\log n - 1 < h \leq \log n$

h must be an integer → $h = \lfloor \log n \rfloor$

Question

1. Is a sorted array a heap?

1. Is a reverse sorted array a heap?

2. Why for Q1 and Q2?

Question

1. Is a sorted array a heap?

   -- It's a min heap. Not a max heap.

1. Is a reverse sorted array a heap?

   -- It's a max heap

1. Why for Q1 and Q2?

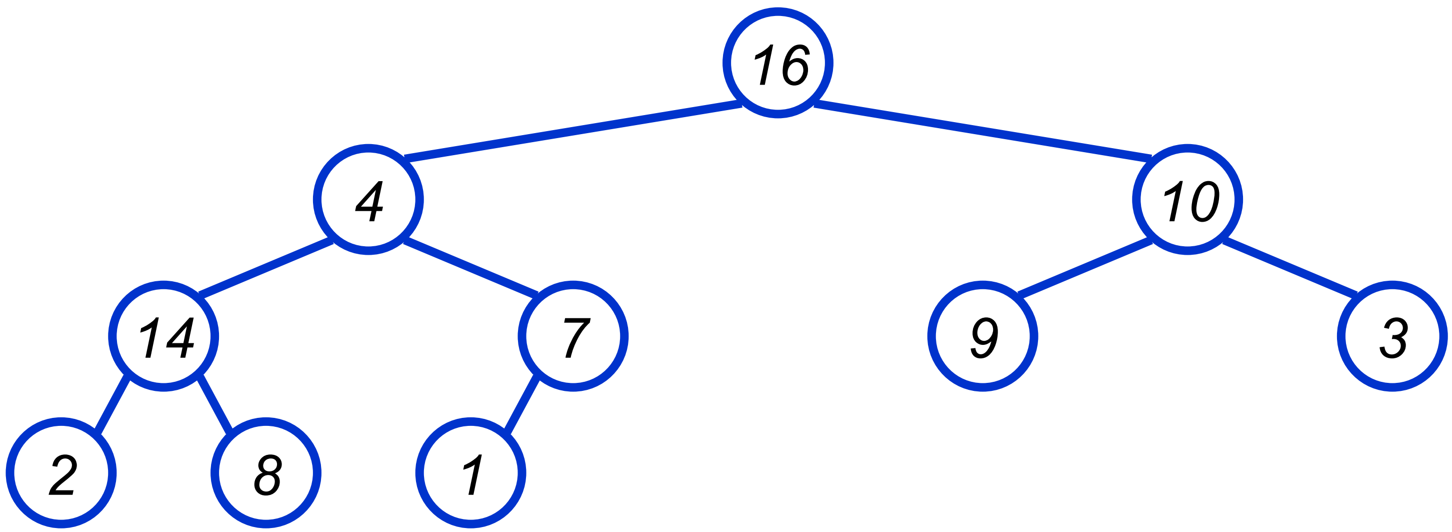
Heap Operations: Heapify()

- **`Heapify()`**: restore the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Problem: The subtree rooted at *i* may violate the heap property (*How?*)
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - *What do you suppose will be the basic operation between i, l, and r?*

Heap Operations: MaxHeapify()

```
MaxHeapify(A, i)
{
   l = Left(i); r = Right(i);
   if (l <= heap_size(A) && A[l] > A[i])
        largest = l;
   else
        largest = i;
   if (r <= heap_size(A) && A[r] > A[largest])
        largest = r;
   if (largest != i)
        Swap(A, i, largest);
        MaxHeapify(A, largest);
}
```

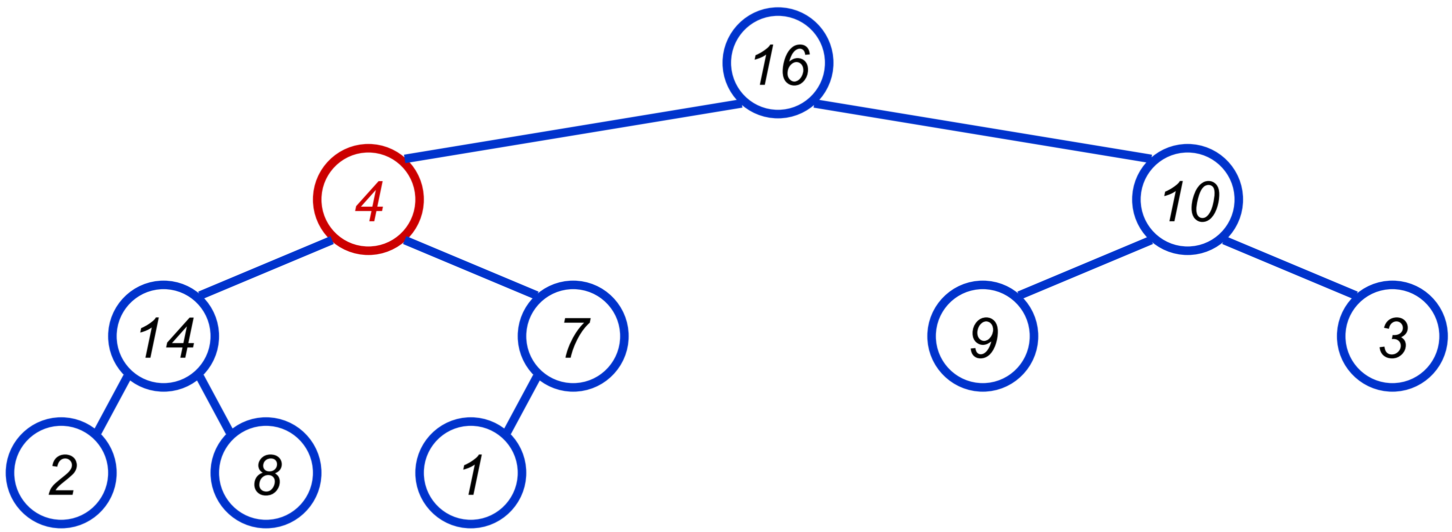
MaxHeapify() Example



$A =$

16	4	10	14	7	9	3	2	8	1

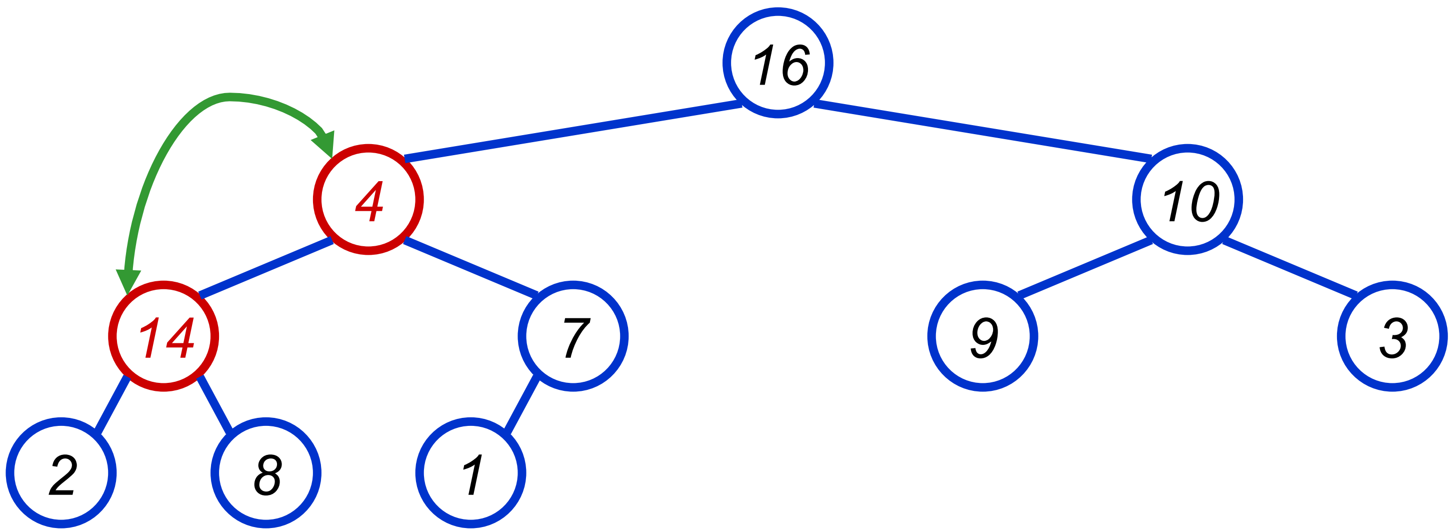
MaxHeapify() Example



$A =$

16	4	10	14	7	9	3	2	8	1

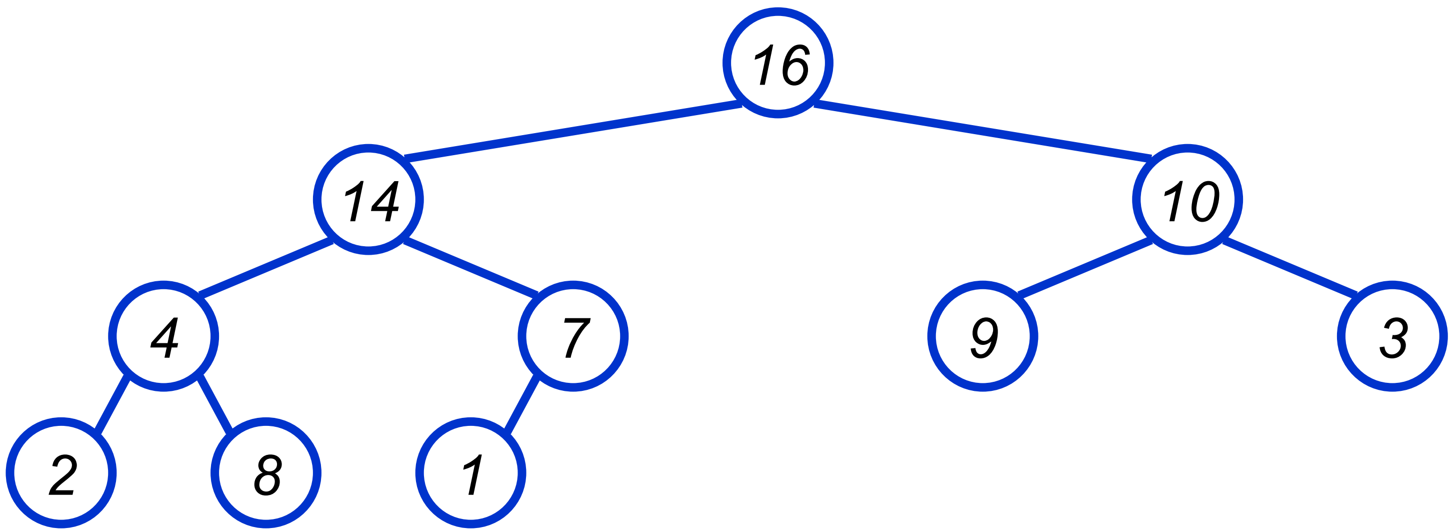
MaxHeapify() Example



A =

16	4	10	14	7	9	3	2	8	1

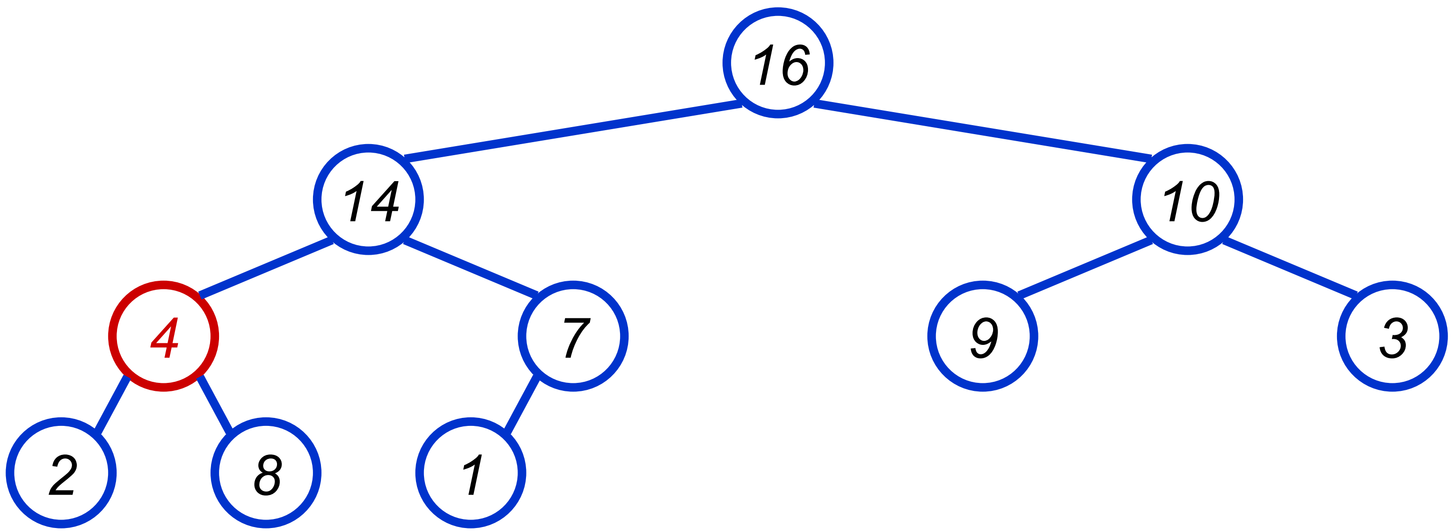
MaxHeapify() Example



$A =$

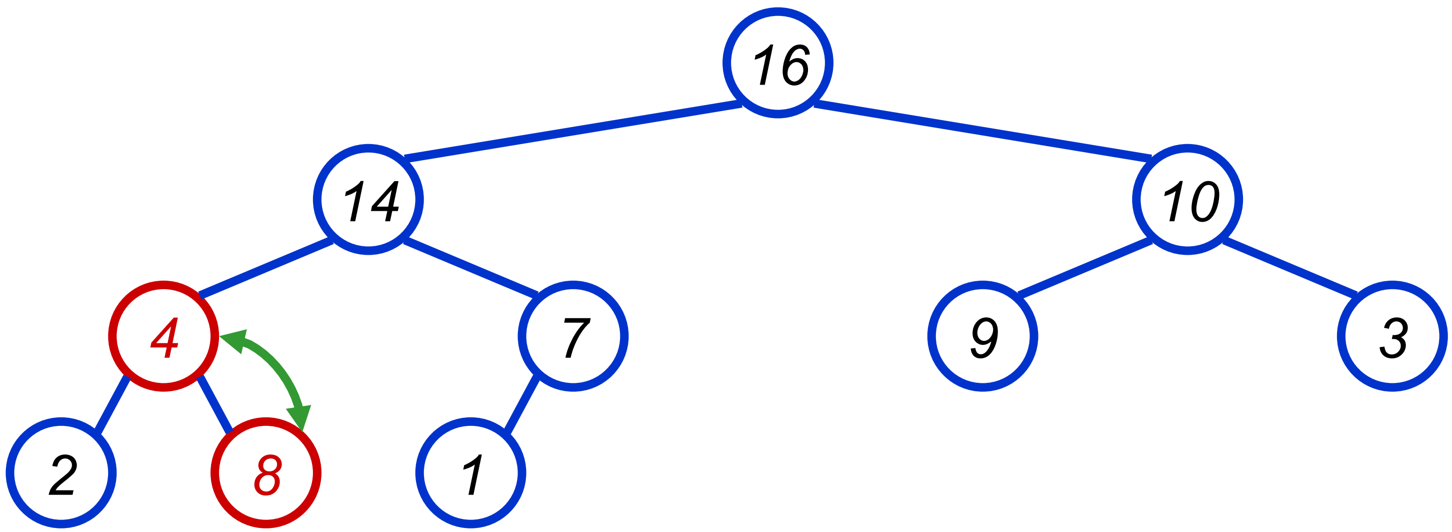
16	14	10	4	7	9	3	2	8	1

MaxHeapify() Example



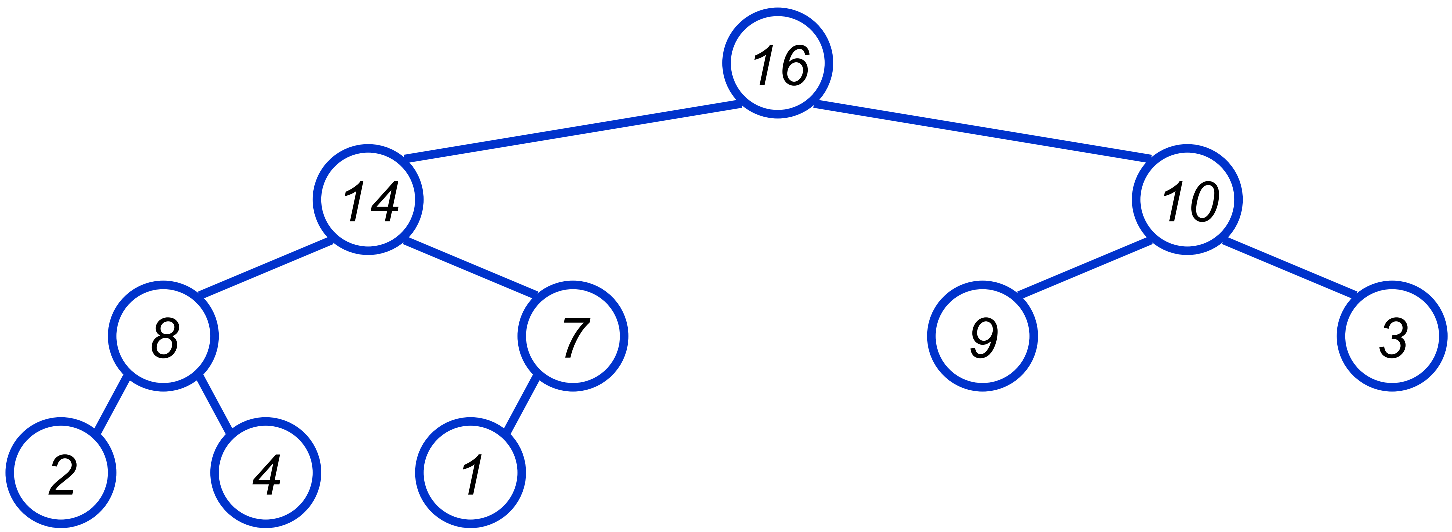
$A =$	16	14	10	4	7	9	3	2	8	1

MaxHeapify() Example



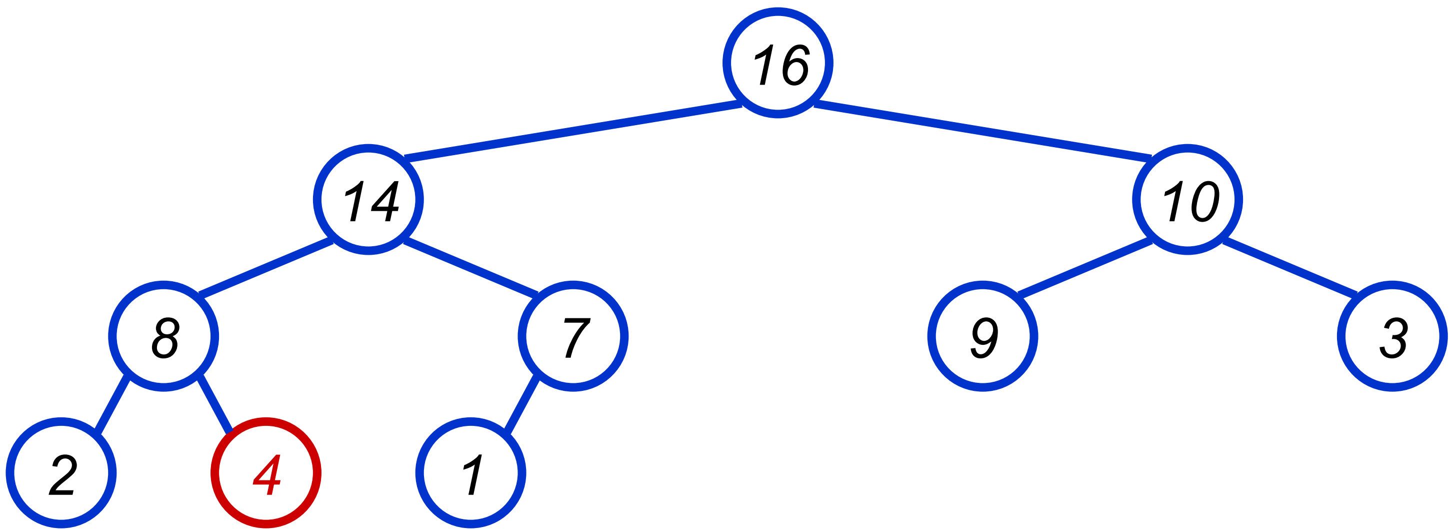
A =	16	14	10	4	7	9	3	2	8	1

MaxHeapify() Example



A =	16	14	10	8	7	9	3	2	4	1

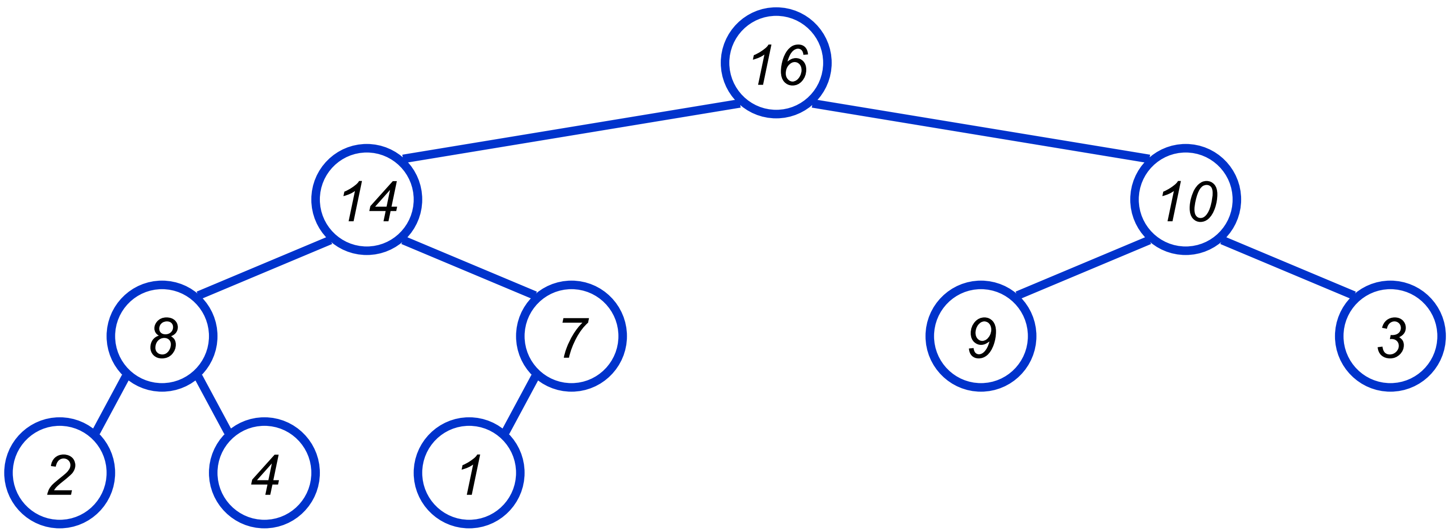
MaxHeapify() Example



A =

16	14	10	8	7	9	3	2	4	1

MaxHeapify() Example



A =

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

Analyzing Heapify()

- *Aside from the recursive call, what is the running time of* **`Heapify()`***?*
- *How many times can* **`Heapify()`** *recursively call itself?*
- *What is the worst-case running time of* **`Heapify()`** *on a heap of size n?*

Ans. O(height) = O(lg n)

Analyzing Heapify()

Can we derive a recursive form of the complexity?

Analyzing Heapify()

Can we derive a recursive form of the complexity?

$$T(n) \le T(2n/3) + \Theta(1) = O(\lg n)$$

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - We can avoid this part of the array: A[$\lfloor n/2 \rfloor$ + 1 .. n] are heaps (*Why?*)

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - We can avoid this part of the array: $A[\lfloor n/2 \rfloor + 1 .. n]$ are heaps (*Why?*)
  - Subtree rooted at each of this node is already a heap. (*Why?*)

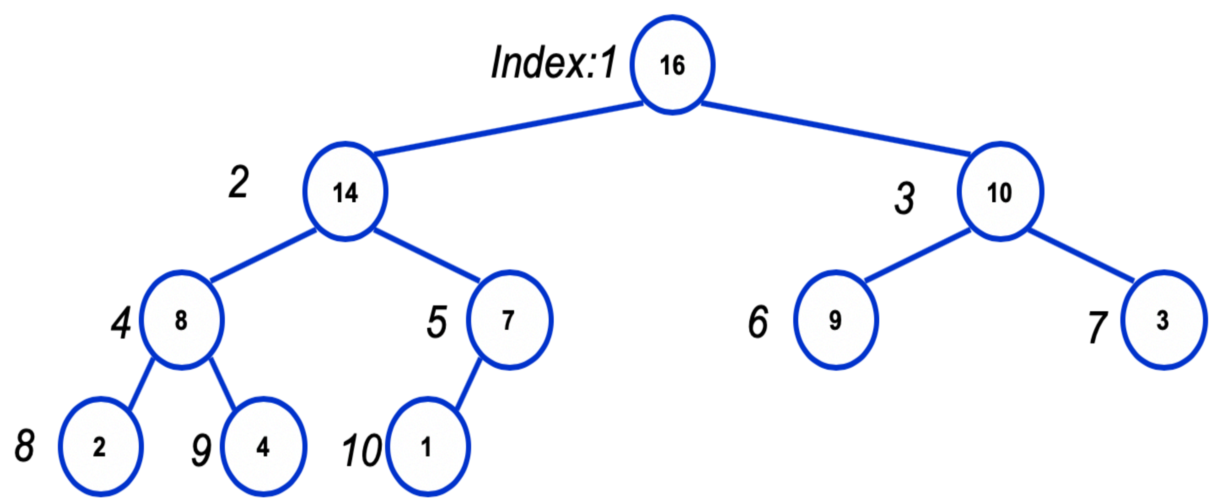
Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - We can avoid this part of the array: A[$\lfloor n/2 \rfloor$ + 1 .. n] are heaps (*Why?*)
  - Subtree rooted at each of this node is already a heap. (*Why?*)

Ans: A[$\lfloor n/2 \rfloor$ + 1 .. n]

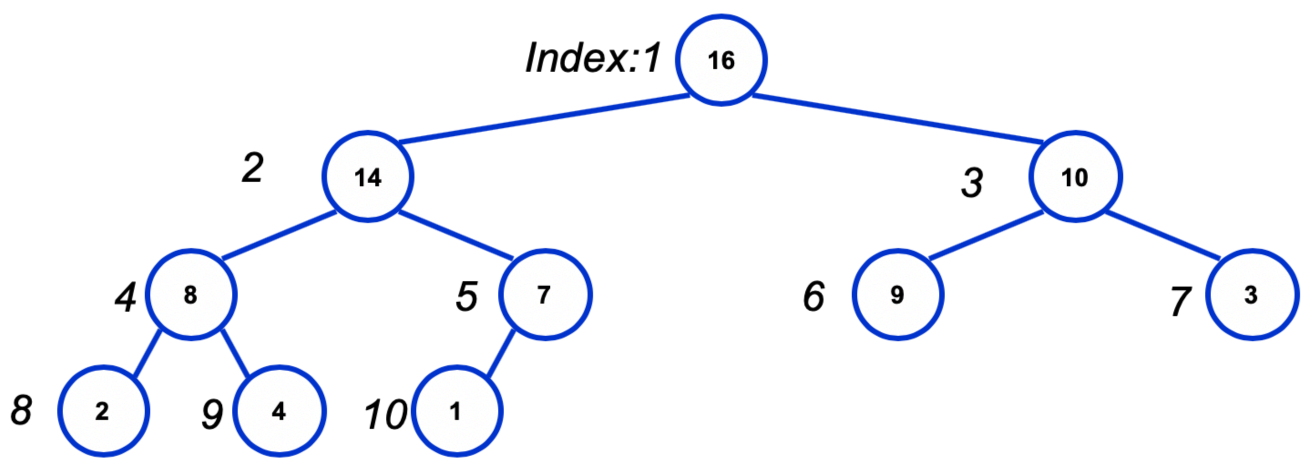
are leaves→

1 element heap.



Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Skip A[⌊n/2⌋ + 1 .. n]
  - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
  - Order of processing guarantees that subtrees rooted at the children of node *i* are heaps when *i* is processed

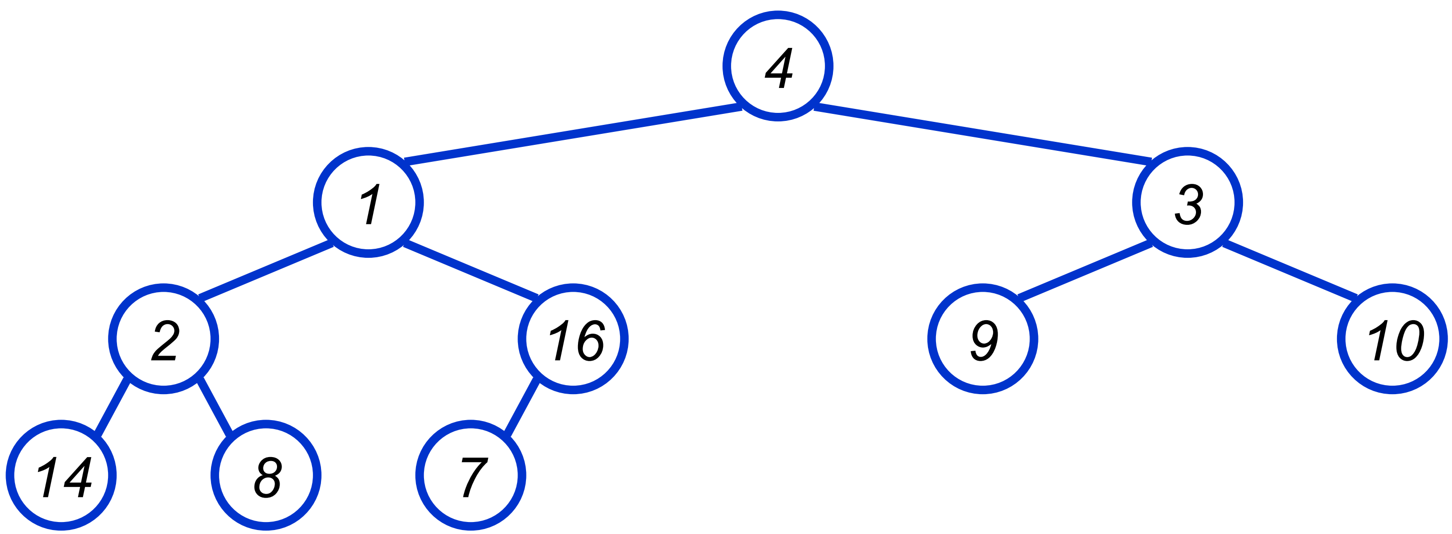


BuildHeap()

```
// given an unsorted array A, make A a heap
BuildHeap(A)
{
  heap_size(A) = length(A);
  for (i = ⌊length[A]/2⌋ downto 1)
     Heapify(A, i);
}
```

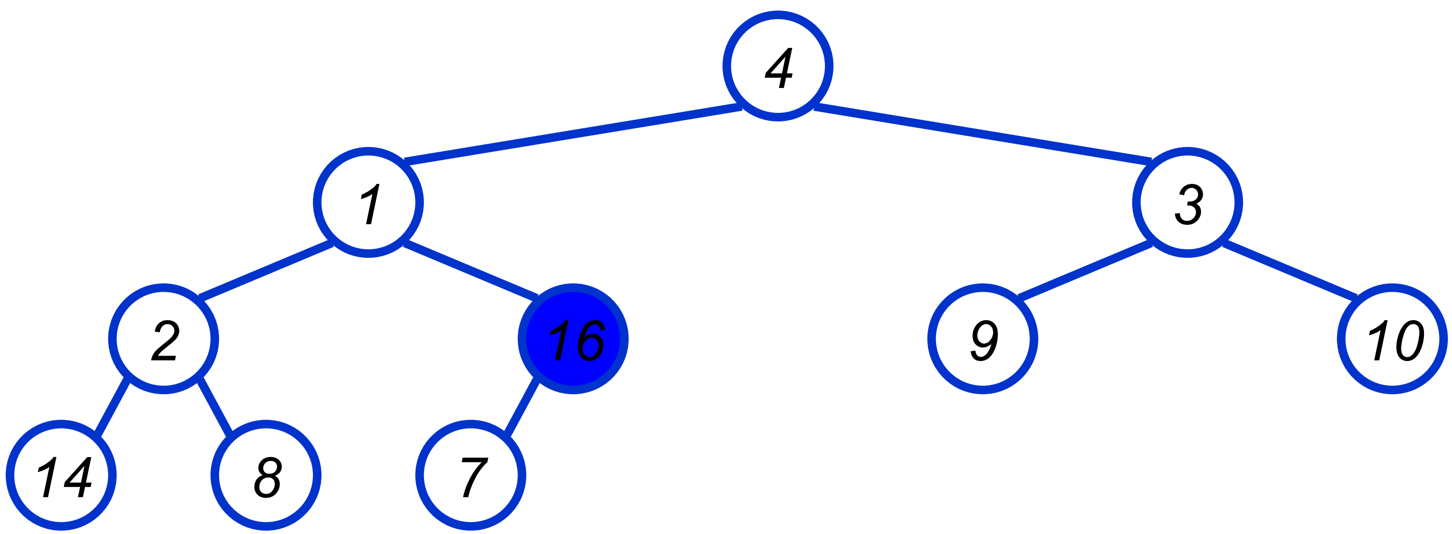
BuildHeap() Example

- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



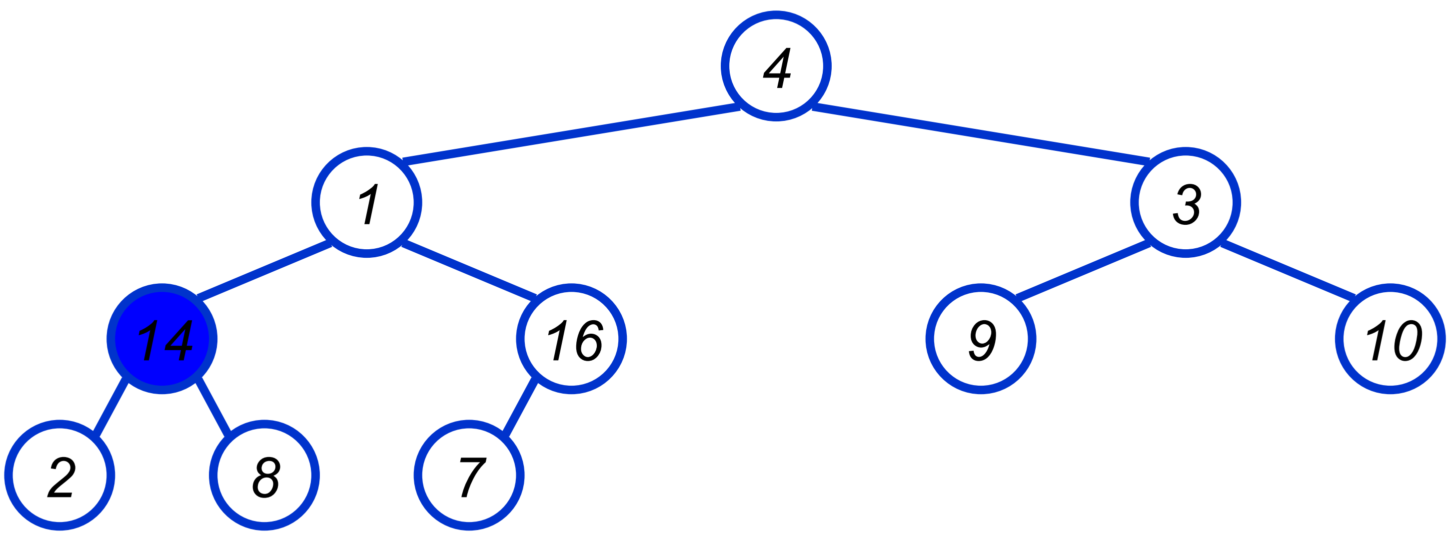
BuildHeap() Example

- Heapify (A, 5) A[5] =16
- Then will be called Heapify (A, 4) A[4] =2



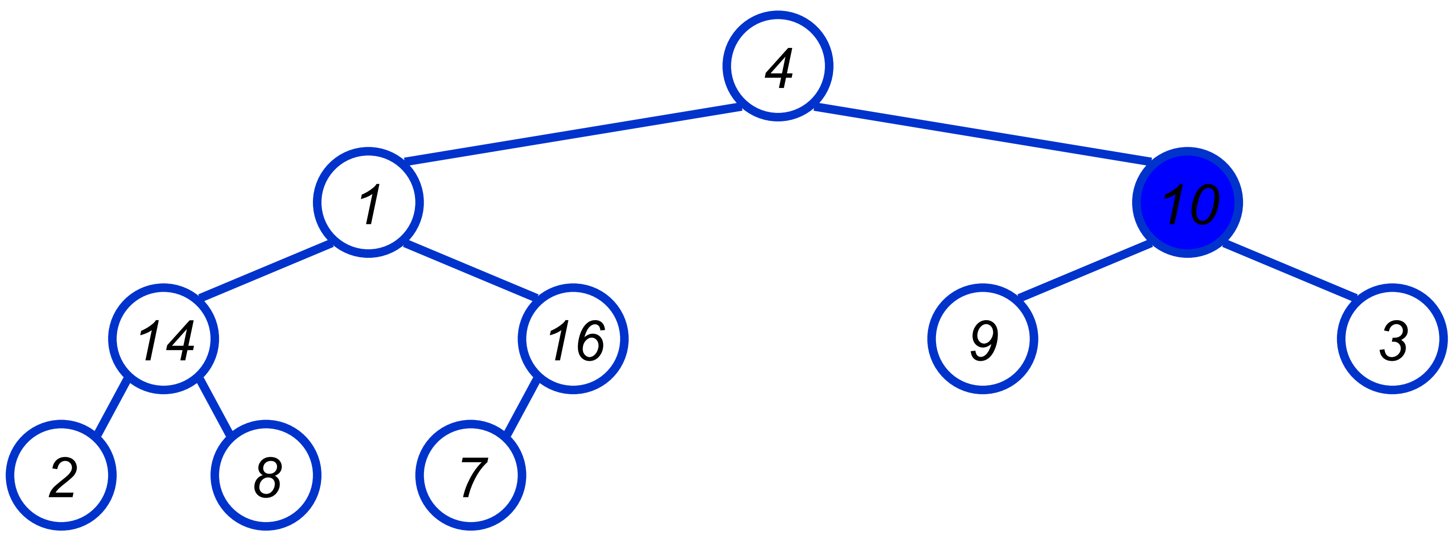
BuildHeap() Example

- Heapify (A, 4) A[4] =2
- Then will be called Heapify (A, 3) A[3] =3



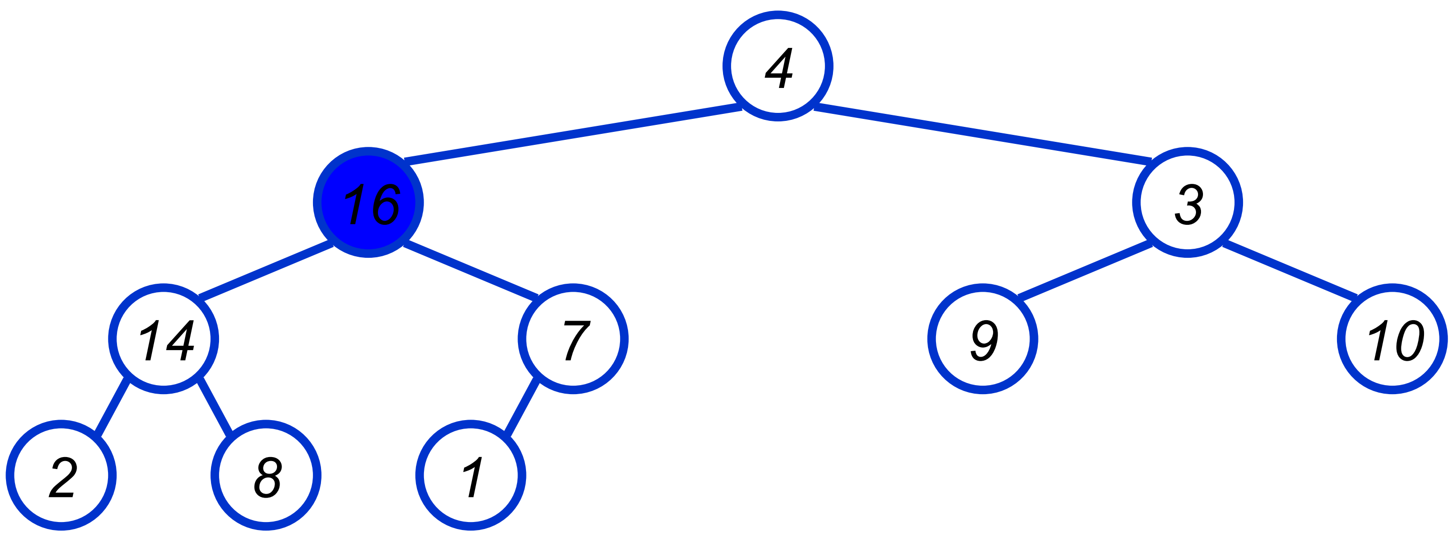
BuildHeap() Example

- Heapify (A, 3) A[3] =3
- Then will be called Heapify (A, 2) A[2] =1



BuildHeap() Example

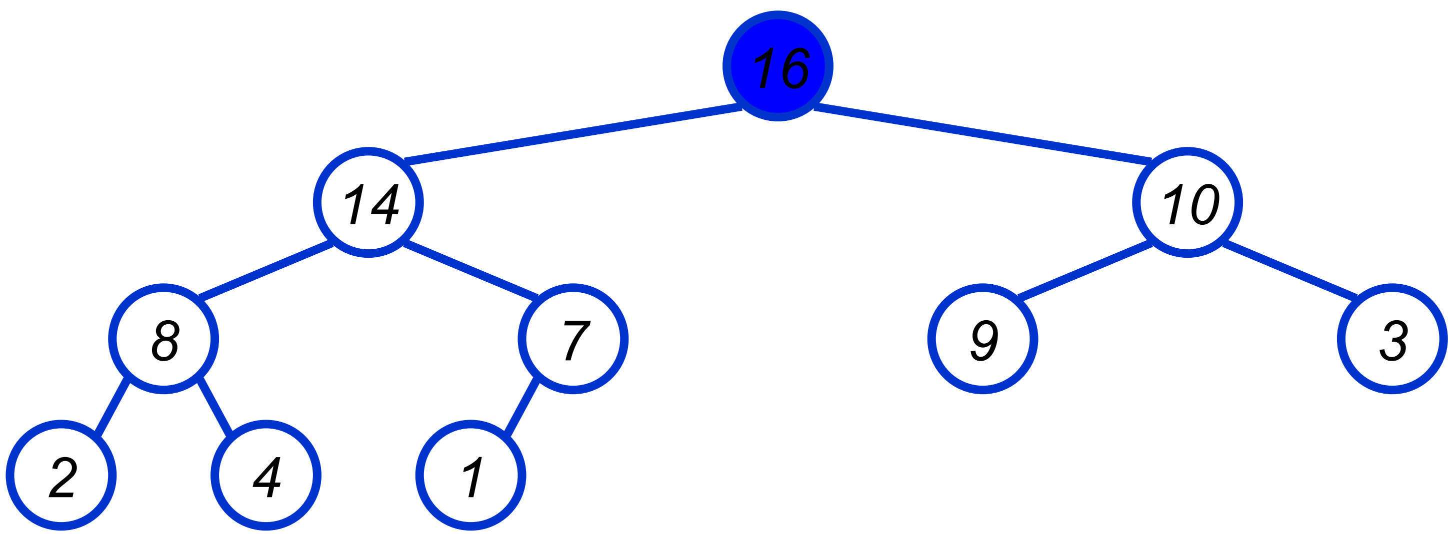
- Heapify (A, 2) A[2] =1
- Then will be called Heapify (A, 1) A[1] =4



BuildHeap() Example

- Heapify (A, 1)
- Done!

A[1] =4



Analyzing BuildHeap()

- Each call to **`Heapify()`** takes O(lg n) time
- There are O(n) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(n lg n)
  - *Is this a correct asymptotic upper bound?*
  - *Is this an asymptotically tight bound?*
- A tighter bound is O(n)
  - By taking the fact that an n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h

Analyzing BuildHeap()

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

We evalaute the last summation by substituting $x = 1/2$ in the formula 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

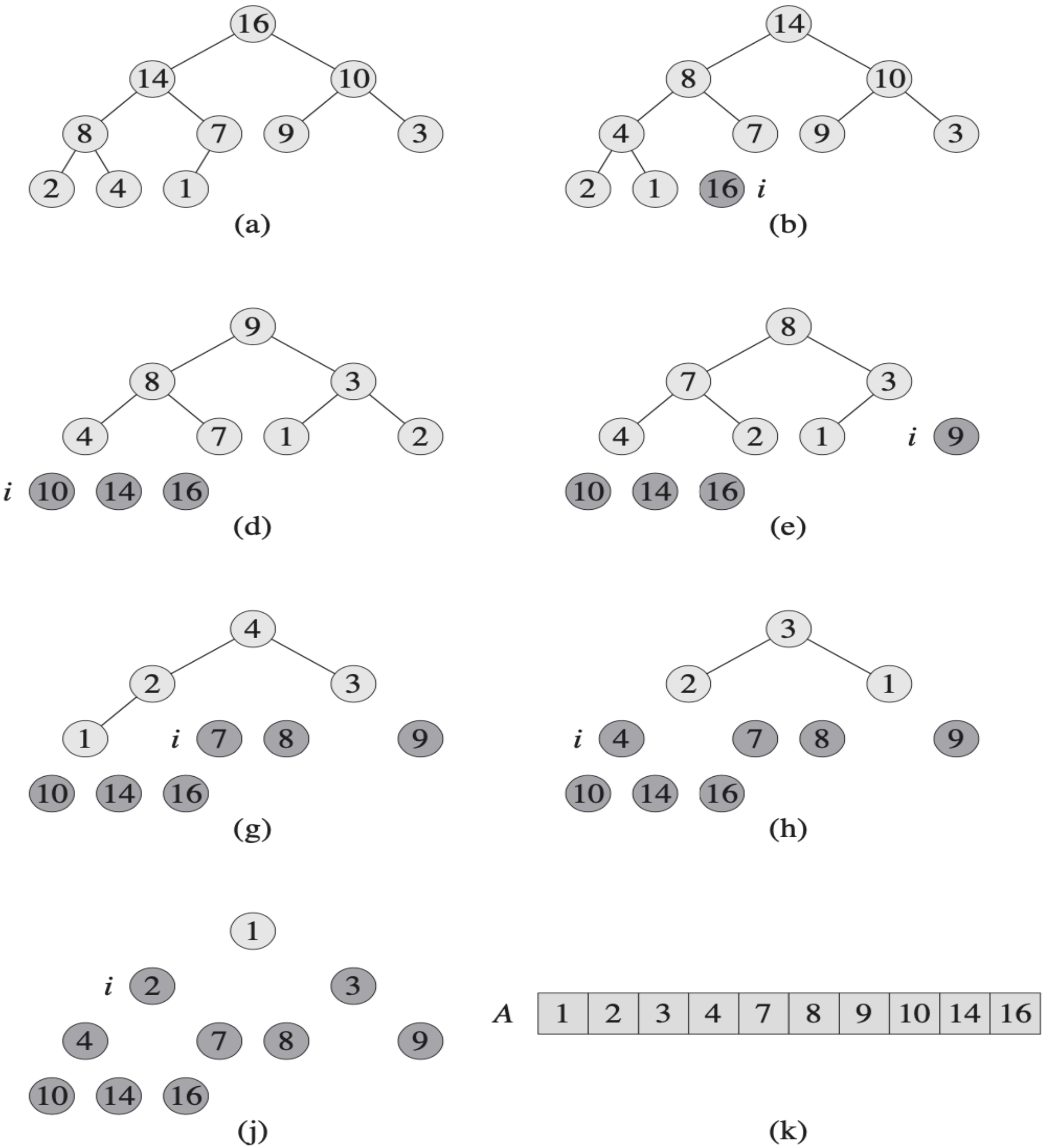
$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n) . \end{aligned}$$

Heapsort

- Given **`BuildHeap()`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains the right value in terms of sorting.
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, always swapping A[1] for A[heap_size(A)]

Heapsort

```
Heapsort(A)
{
      BuildHeap(A);
      for (i = length(A) downto 2)
      {
            Swap(A[1], A[i]);
            heap_size(A) -= 1;
            Heapify(A, 1);
      }
}
```



An example after BuildHeap

Analyzing Heapsort

- The call to **`BuildHeap()`** takes $O(n)$ time
- Each of the $n - 1$ calls to **`Heapify()`** takes $O(\lg n)$ time
- Thus the total time taken by **`HeapSort()`**
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$

Priority Queues

The heap data structure is incredibly useful for implementing *priority queues*

- A data structure for maintaining a set *S* of elements, each with an associated value or *key*
- Enables to easily extract an element with the highest priority (min or max)
  - **`Max Priority Queue: using Max Heap`**
  - **`Min Priority Queue: using Min Heap`**
- ***What might a priority queue be useful for?***