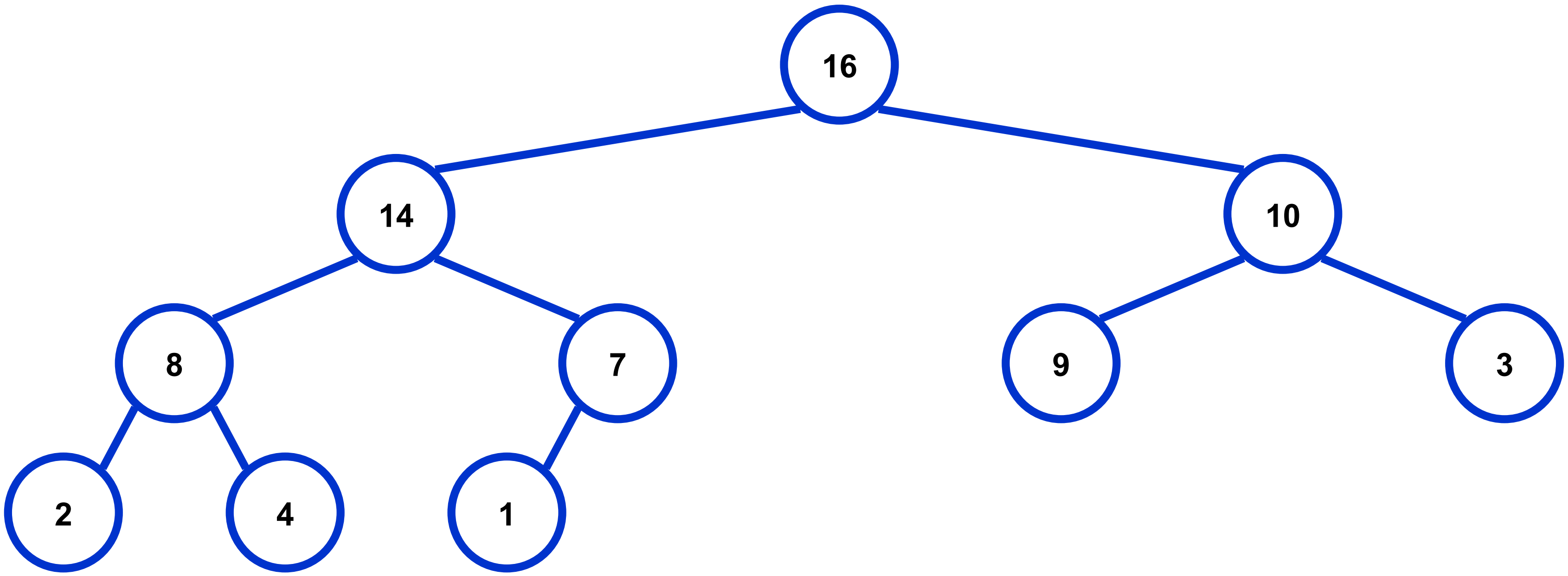
Heaps(Priority Queue)

Heaps

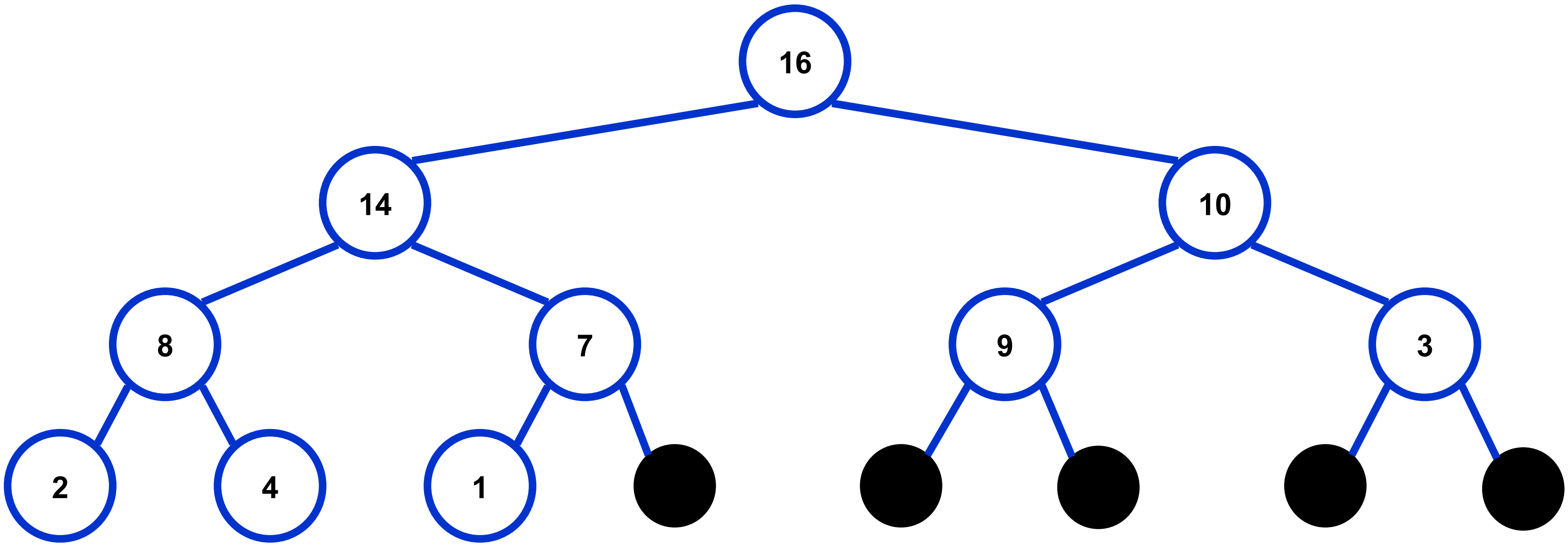
- A *heap* can be seen as a complete binary tree:



  - *What makes a binary tree complete?*
  - *Is the example above complete?*

Heaps

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



  - Can think of unfilled slots as null pointers

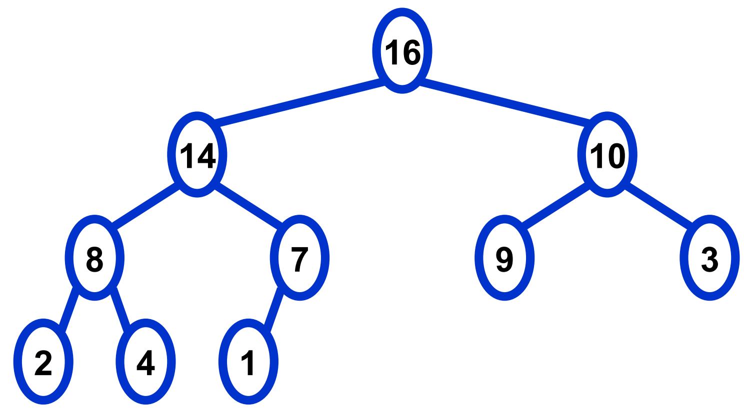
Heaps

- In practice, heaps are usually implemented as arrays:

A =

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

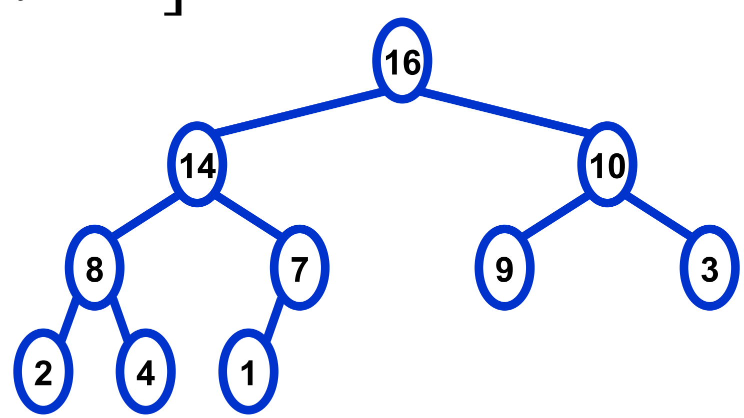
=



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]

A = | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 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
  - *Where is the largest element in a heap stored?*
- 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

Heap Height

- *What is the height of an n-element heap? Why?*
- This is nice: basic heap operations take at most time proportional to the height of the heap

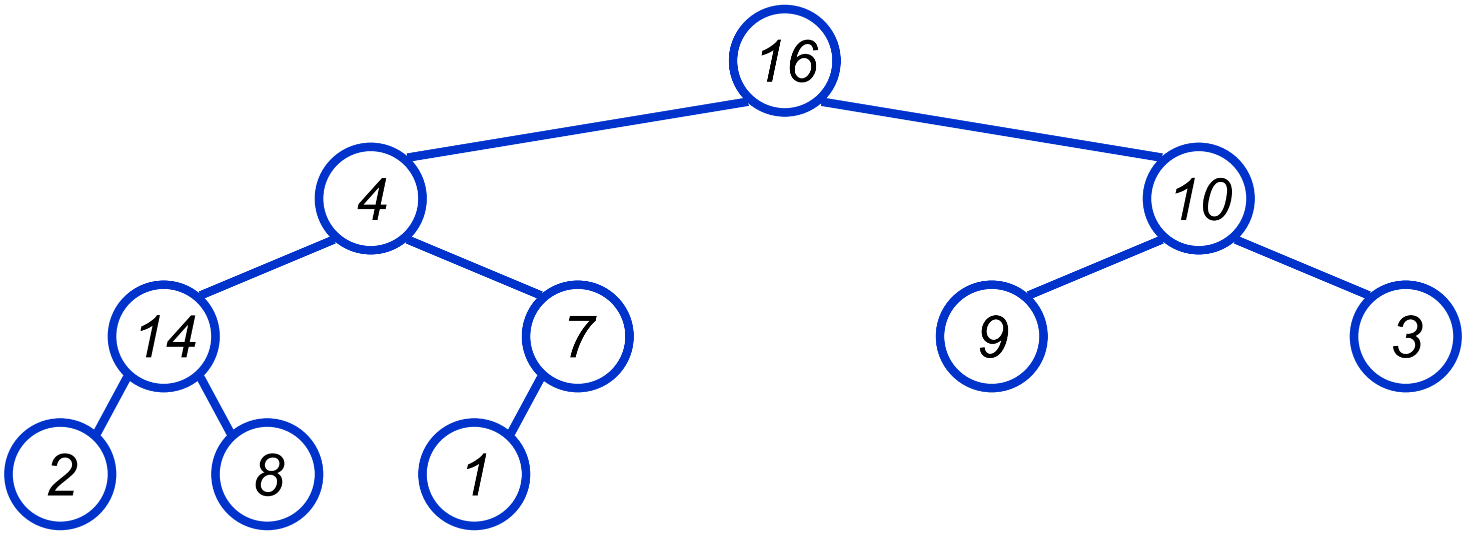
Heap Operations: Heapify()

- **`Heapify()`**: maintain 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: Heapify()

```
Heapify(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);
      Heapify(A, largest);
}
```

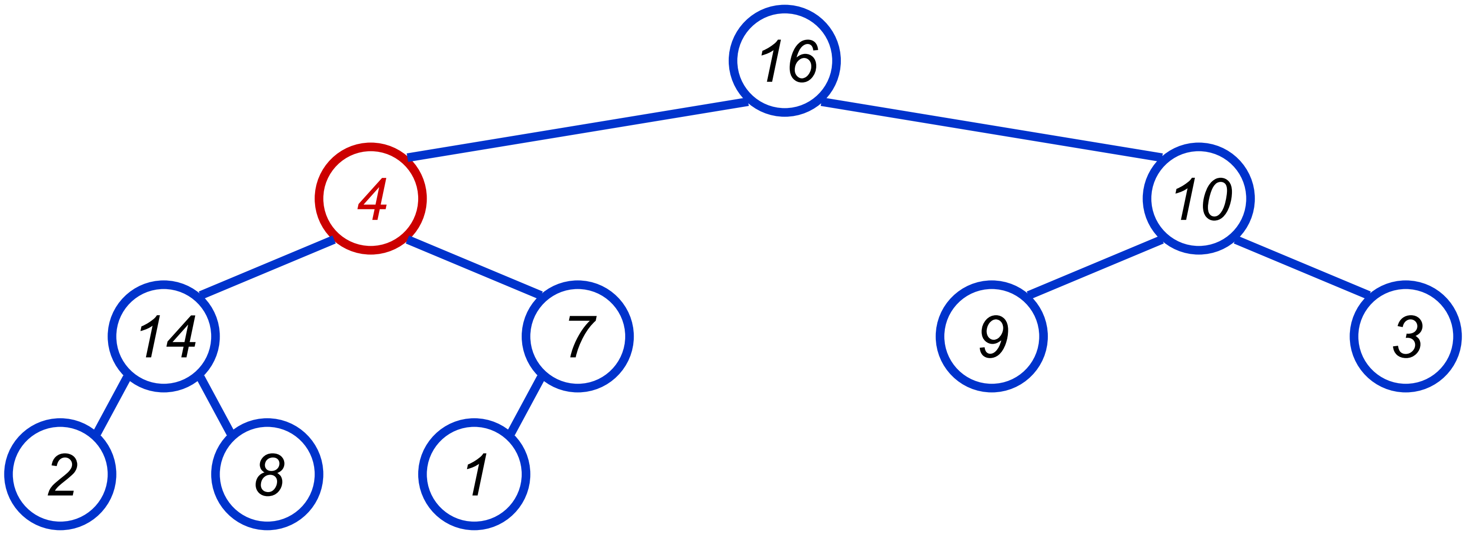
Heapify() Example



A =

16	4	10	14	7	9	3	2	8	1

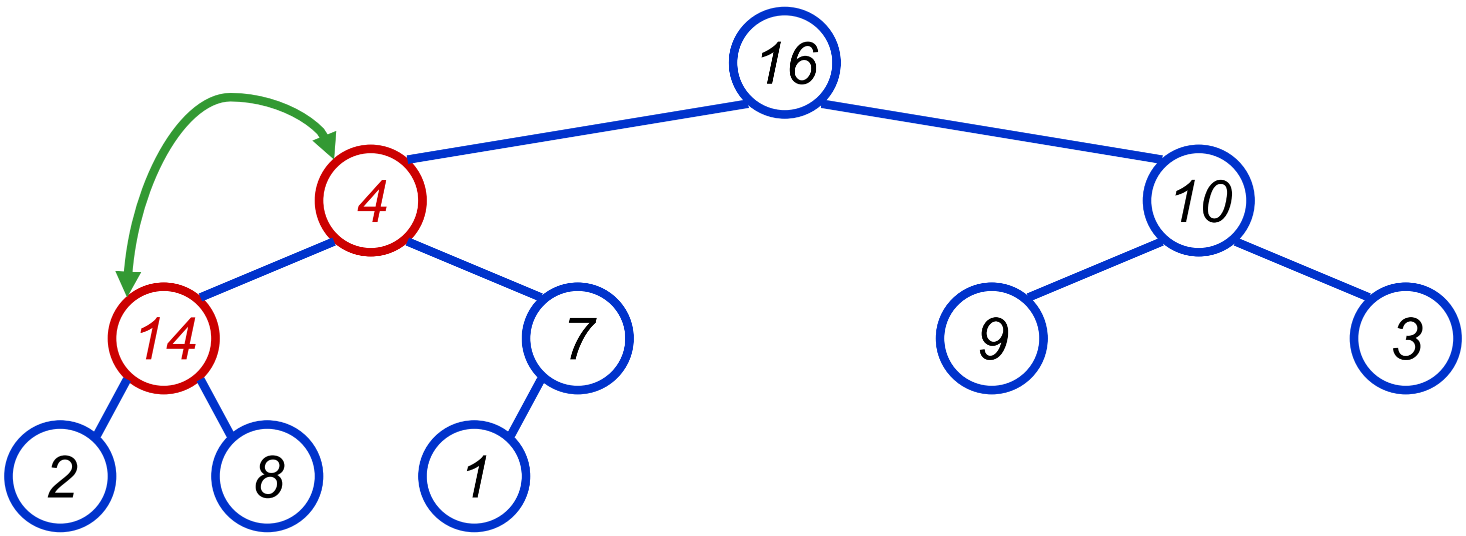
Heapify() Example



A =

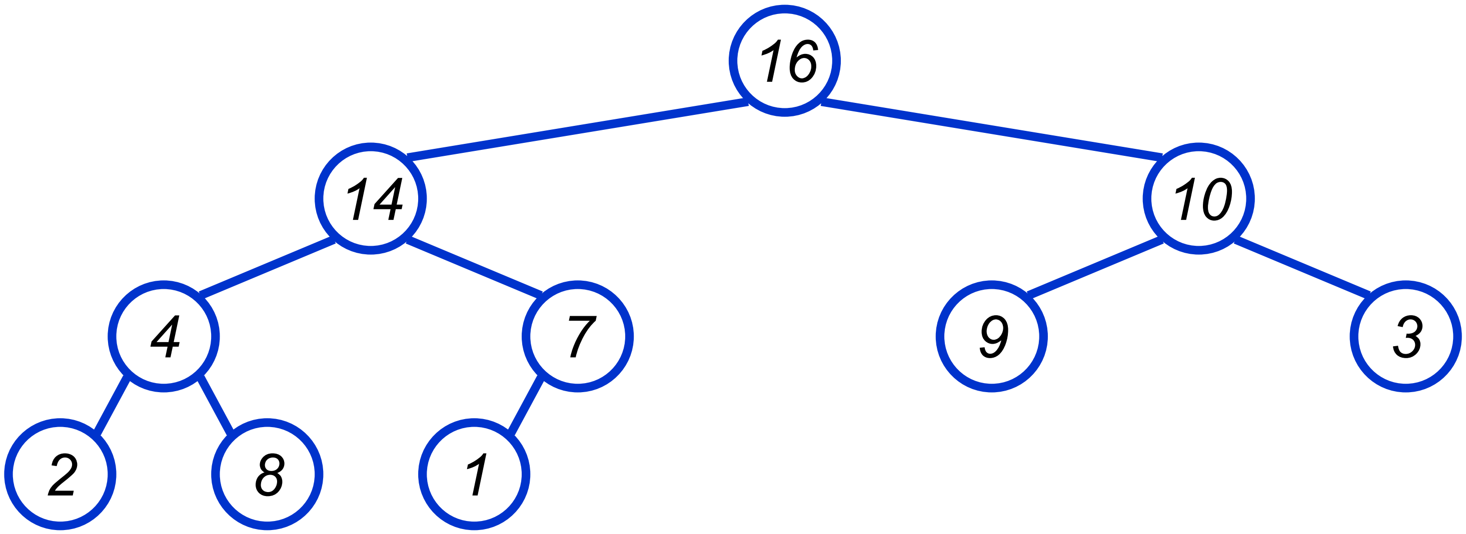
16	4	10	14	7	9	3	2	8	1

Heapify() Example



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

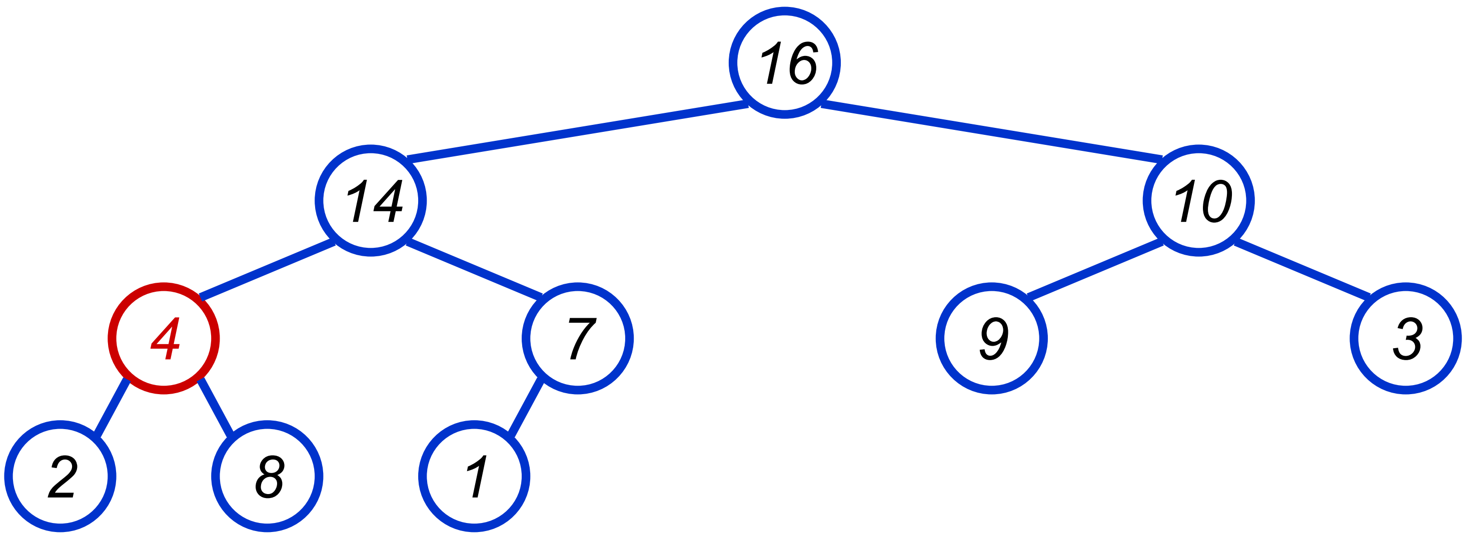
Heapify() Example



A =

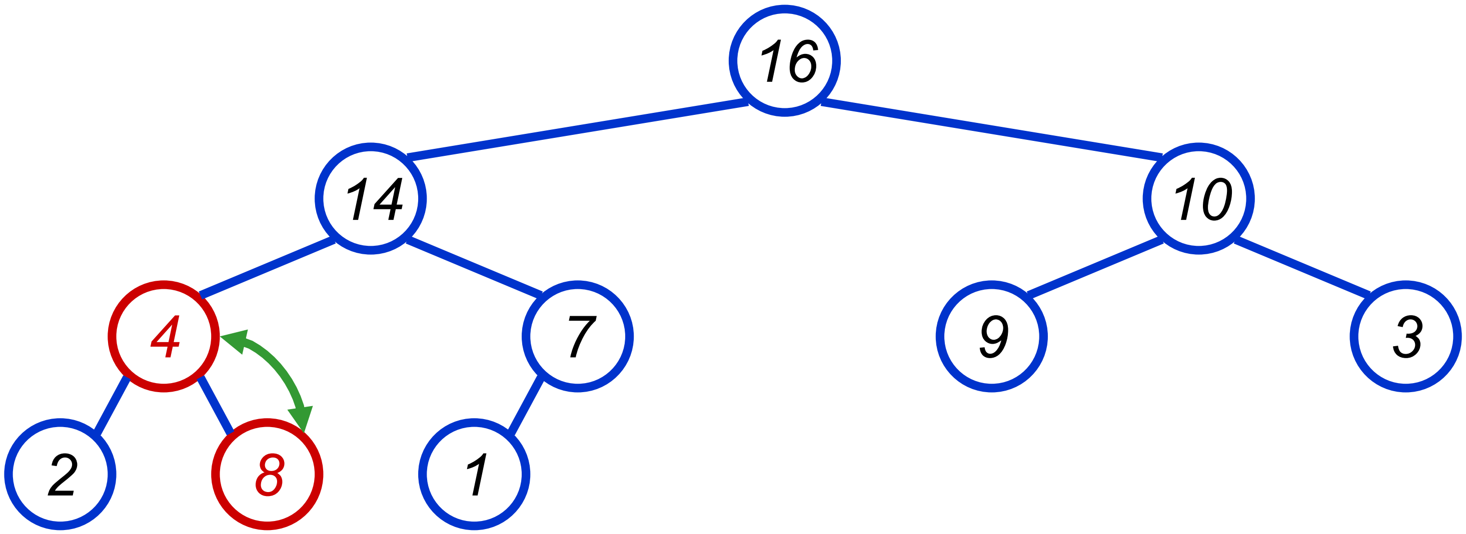
16	*14*	*10*	*4*	*7*	*9*	*3*	*2*	*8*	*1*

Heapify() Example



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

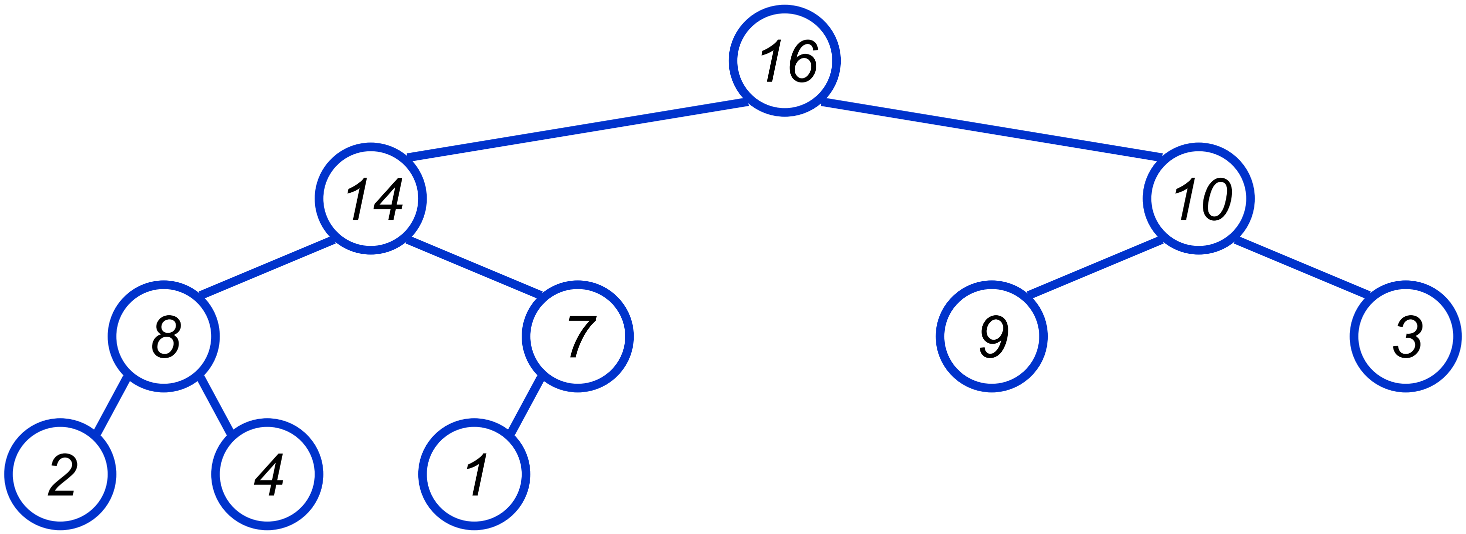
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

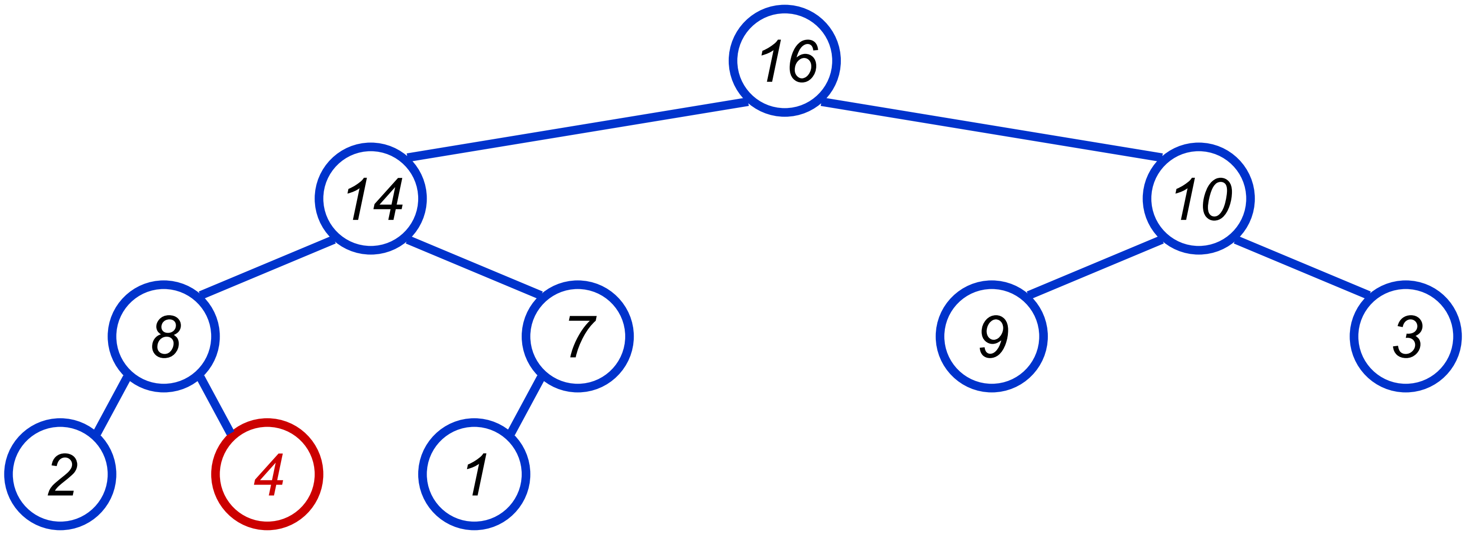
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

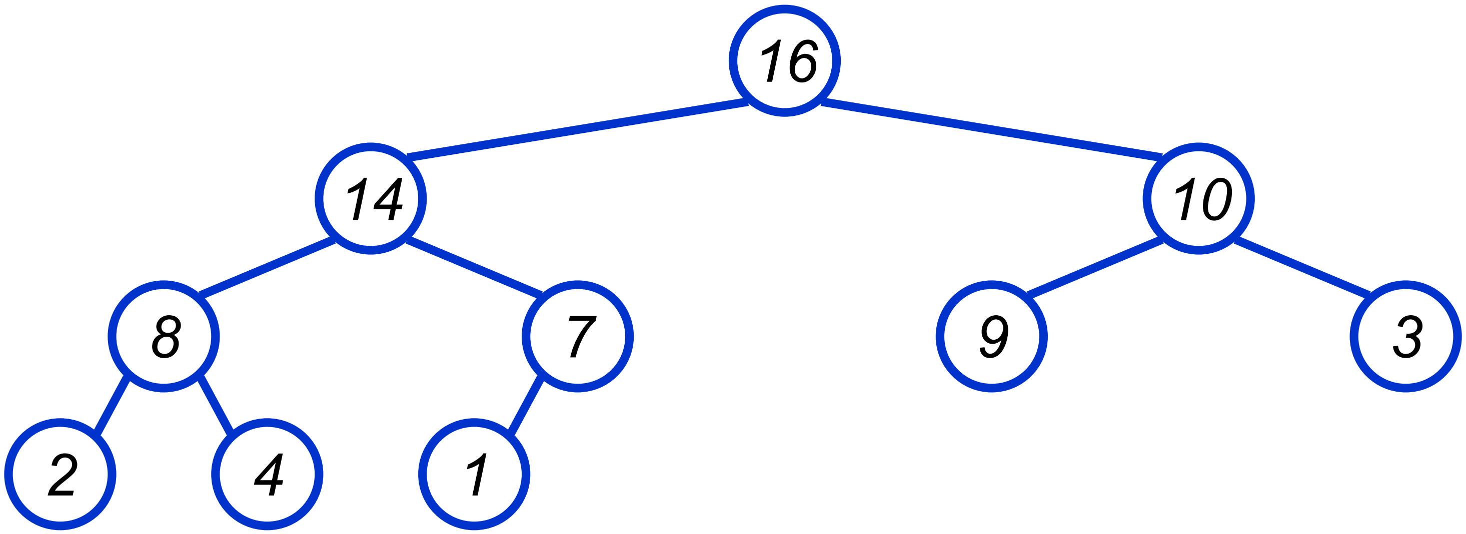
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

Heapify() Example



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

Analyzing Heapify(): Formal

- Fixing up relationships between i, l, and r takes $\Theta(1)$ time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
  - Draw it
- Answer: $2n/3$ (worst case: bottom row 1/2 full)
- So time taken by **`Heapify()`** is given by

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

Analyzing Heapify(): Formal

- So we have

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

- By case 2 of the Master Theorem,

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

- Thus, **`Heapify()`** takes less than linear time

Heap Operations: BuildHeap()

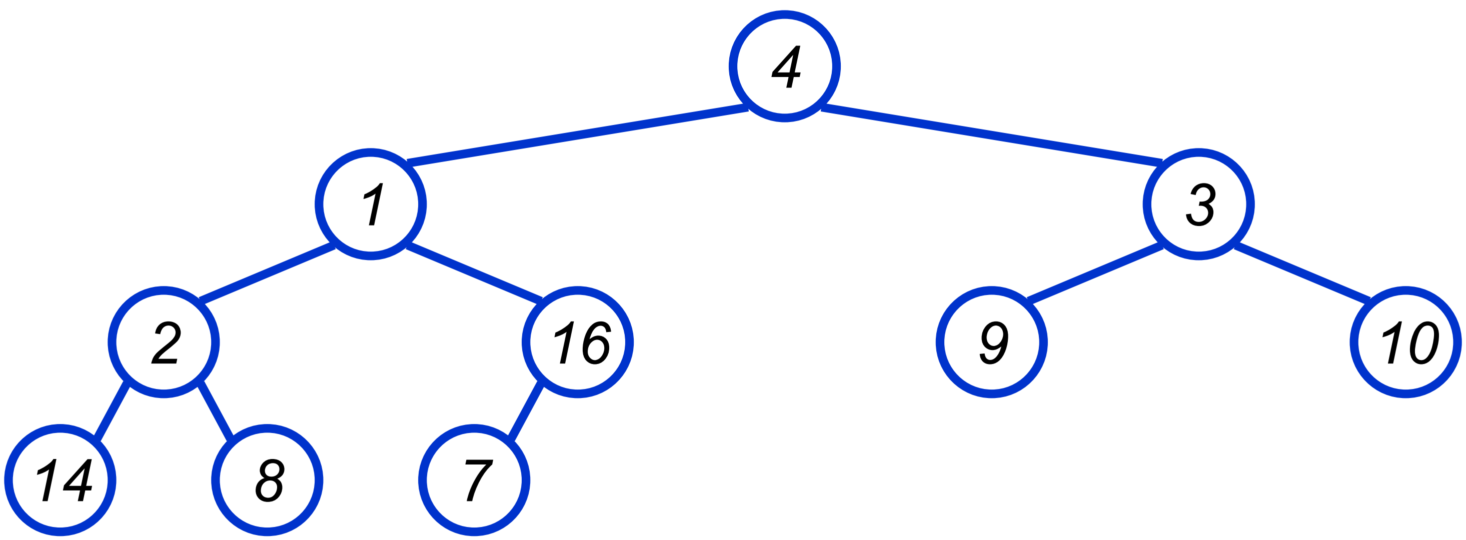
- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length *n*, all elements in range A[$\lfloor n/2 \rfloor$ + 1 .. n] are heaps (*Why?*)
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that 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}



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 correct value
  - 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);
    }
}
```

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

- Heapsort is a nice algorithm, but in practice Quicksort usually wins
- But 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*
  - Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**
  - ***What might a priority queue be useful for?***

Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key
- *How could we implement these operations using a heap?*