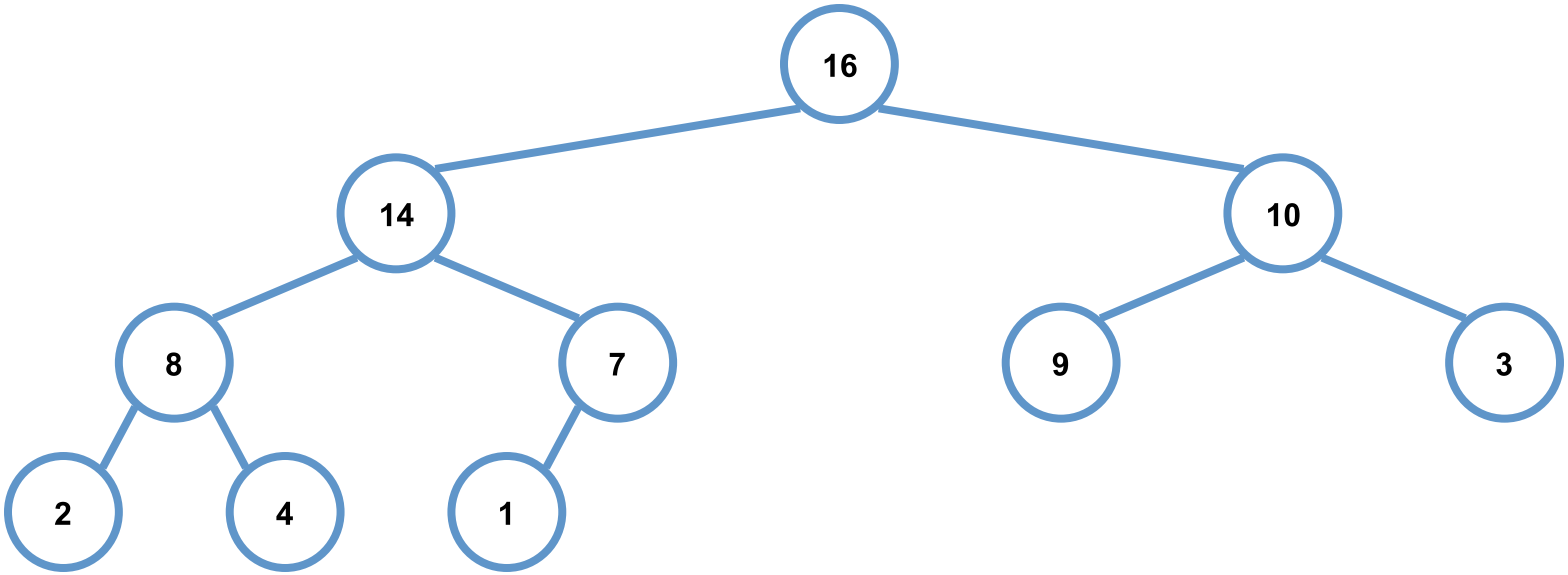
Algorithm and Data Structure Analysis (ADSA)

Lecture 8: Priority Queues

(Book Chapter 6)

Heaps

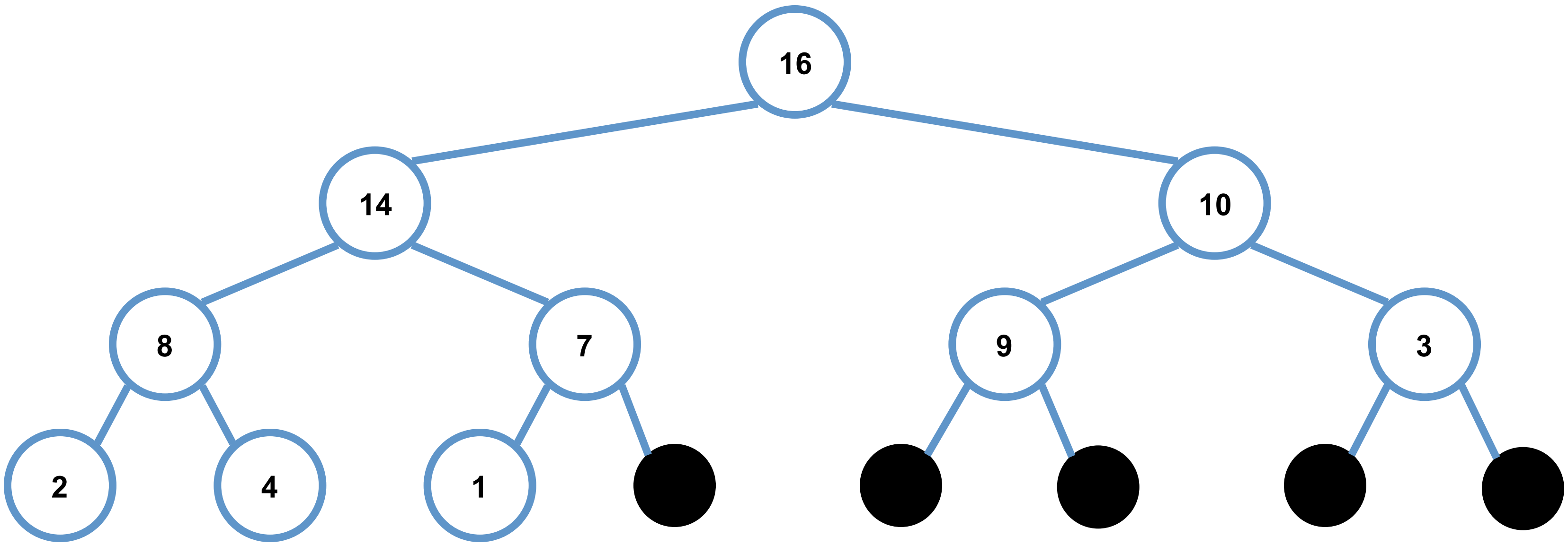
- A *heap* can be seen as a complete binary tree:



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

Heaps

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



  - The book calls them "nearly complete" binary trees; can think of unfilled slots as null pointers

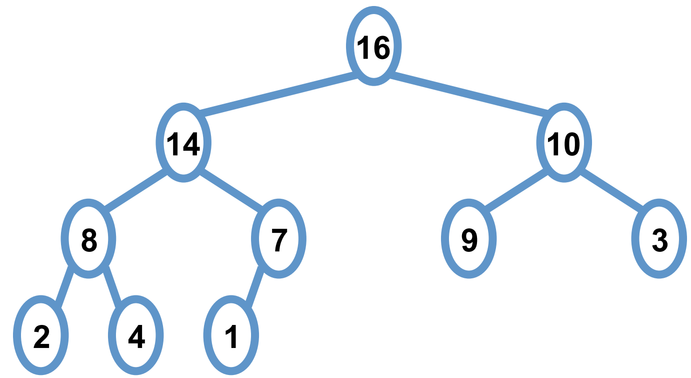
Heaps

- In practice, heaps are usually implemented as arrays:

A =

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

=



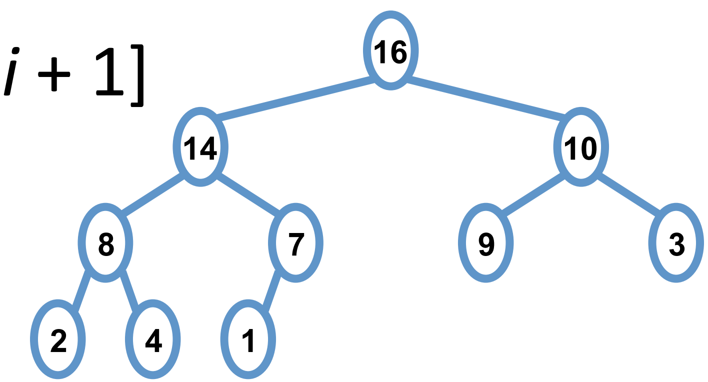
Heaps

- 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[2*i*]
  - The right child of node *i* is A[2*i* + 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; }
```

- An aside: *How would you implement this most efficiently?*
  - Trick question, I was looking for "`i << 1`", etc.
  - But, any modern compiler is smart enough to do this for you (and it makes the code hard to follow)

The Heap Property

- Heaps also satisfy the *heap property*:

  A[*Parent*(*i*)] ≥ 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?*

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 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

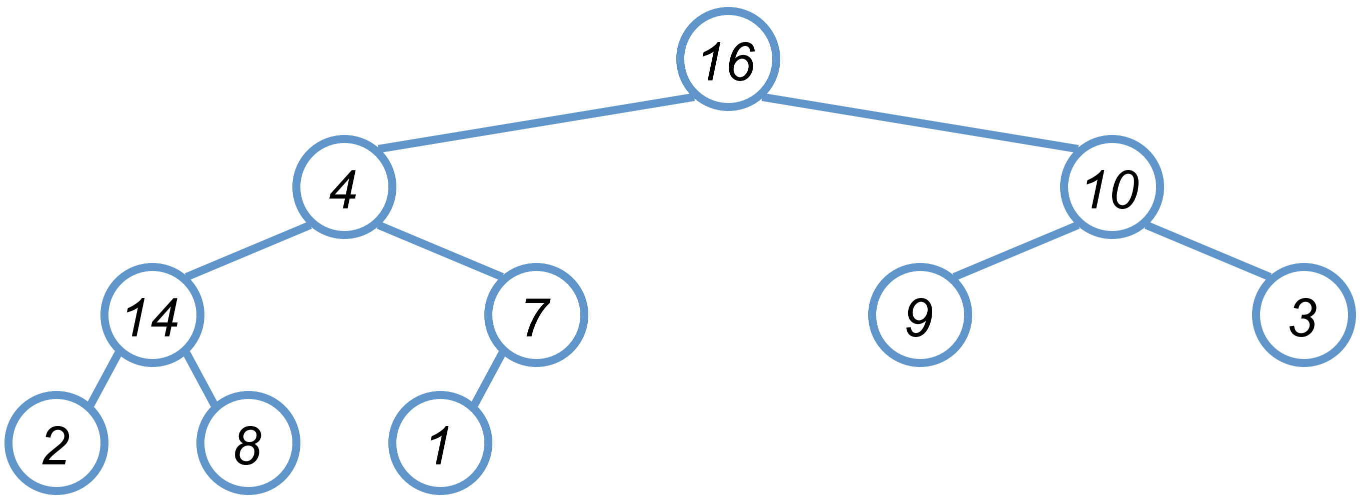
Heap Operations: Heapify

- **`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);
}
```

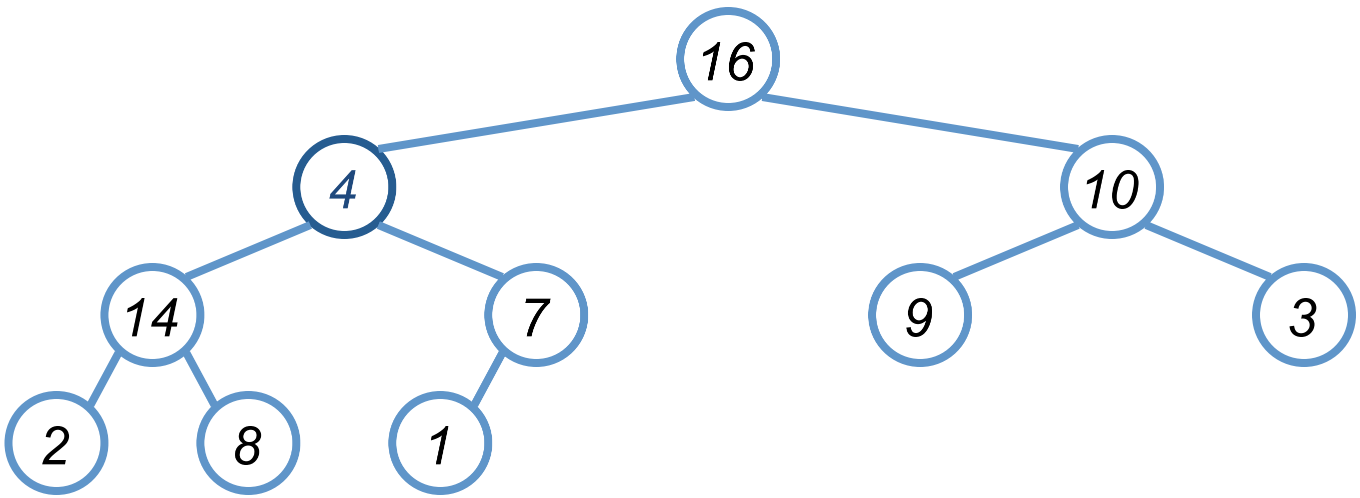
Heapify Example



A =

16	4	10	14	7	9	3	2	8	1

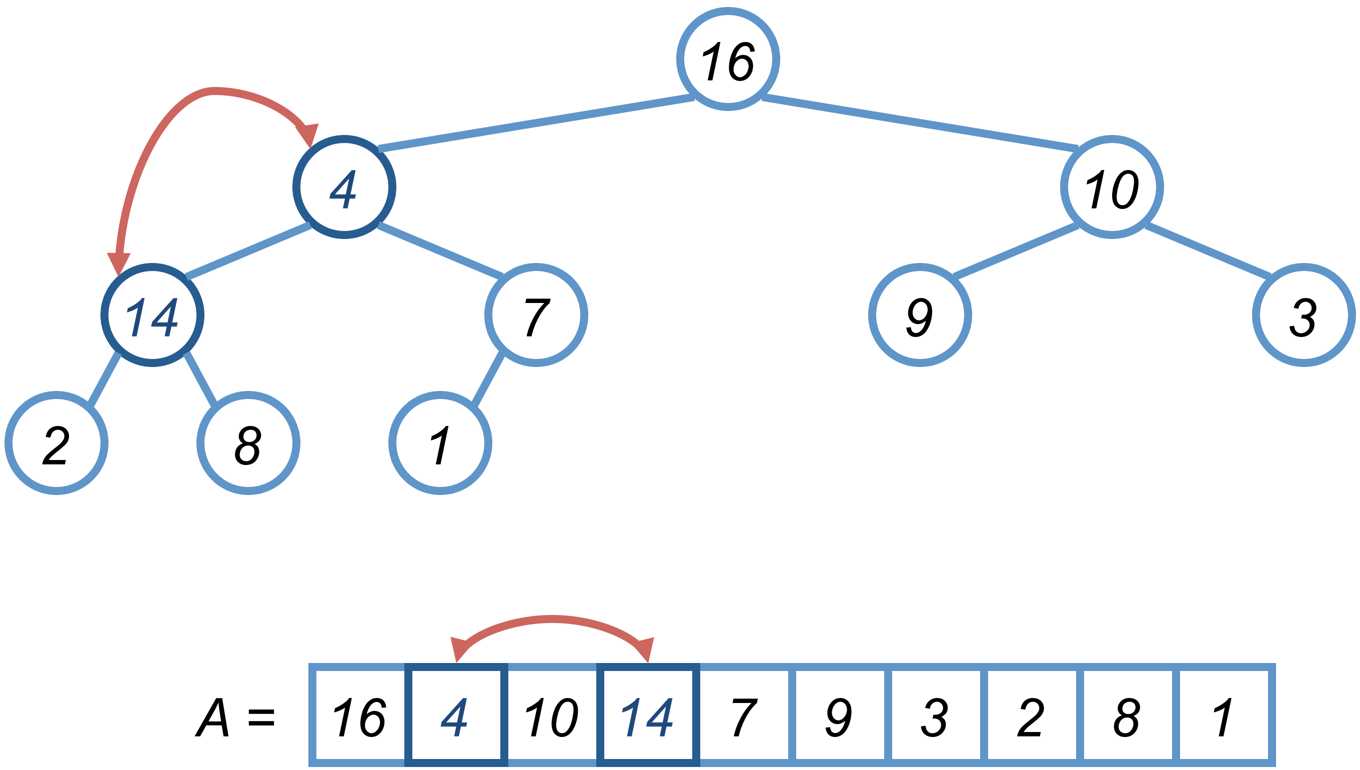
Heapify Example



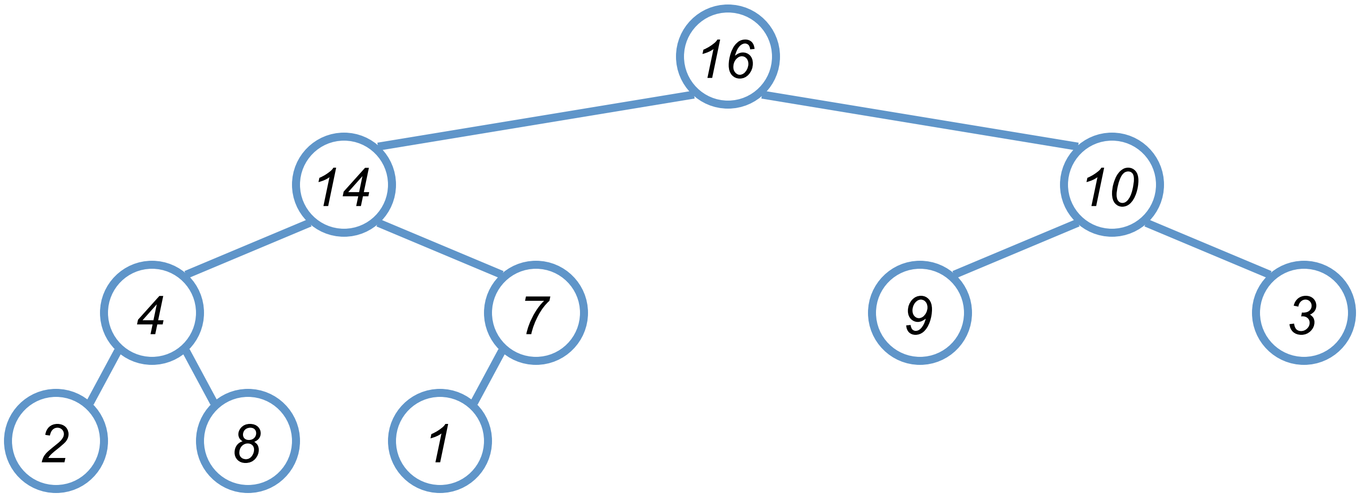
A =

16	4	10	14	7	9	3	2	8	1

Heapify Example



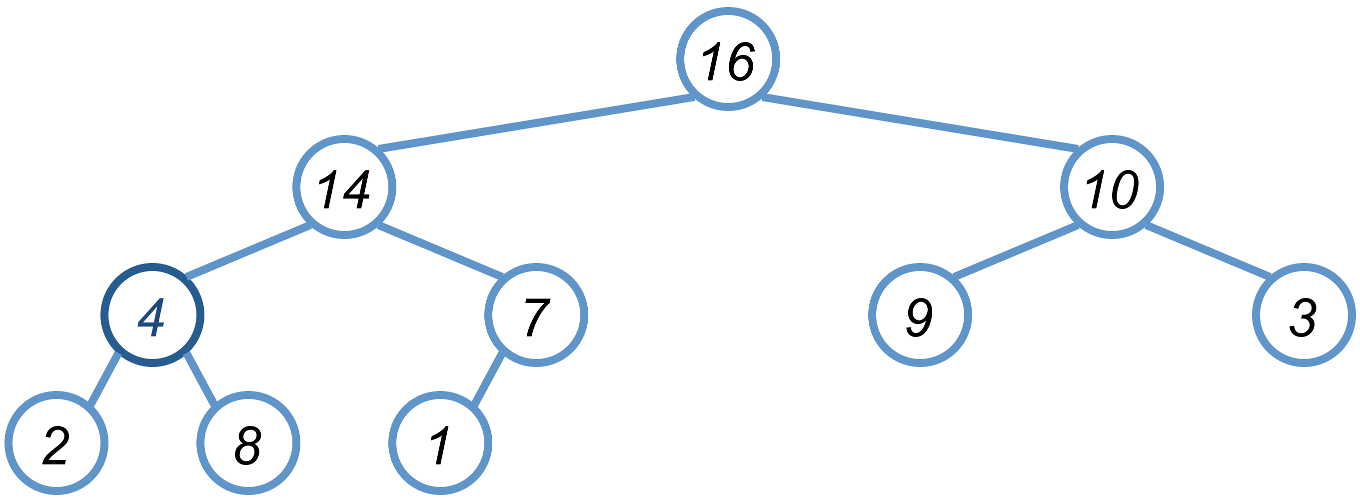
Heapify Example



A =

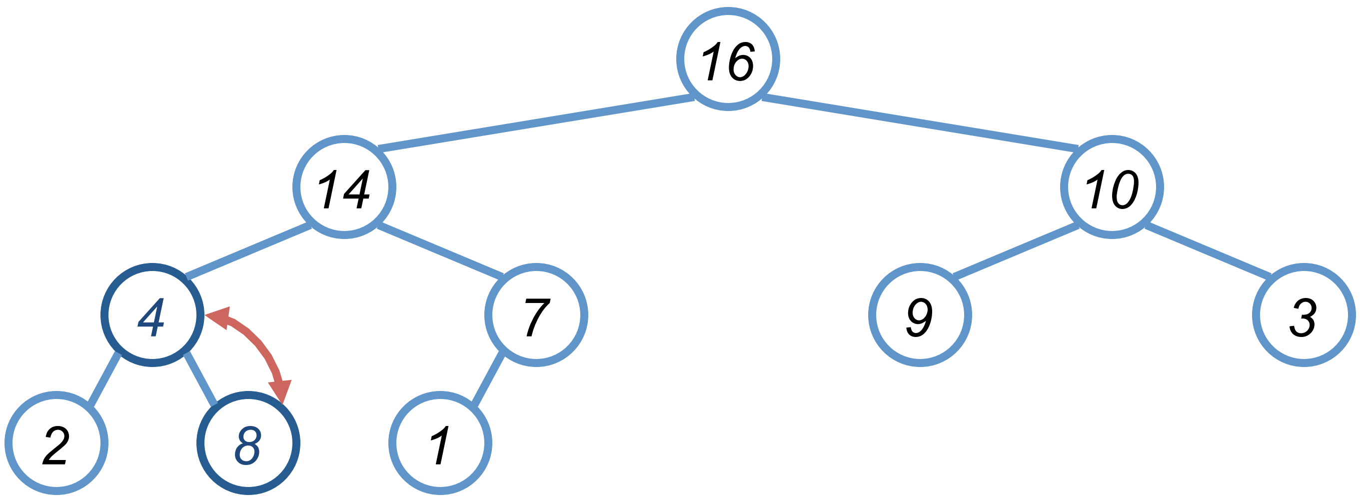
16	14	10	4	7	9	3	2	8	1

Heapify Example



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

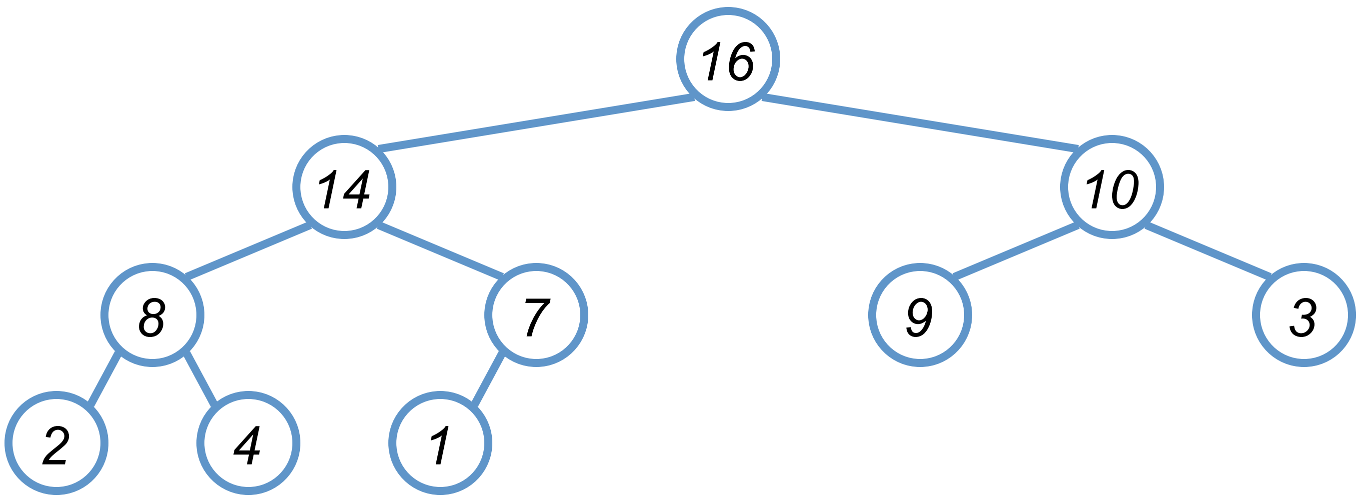
Heapify Example



A =

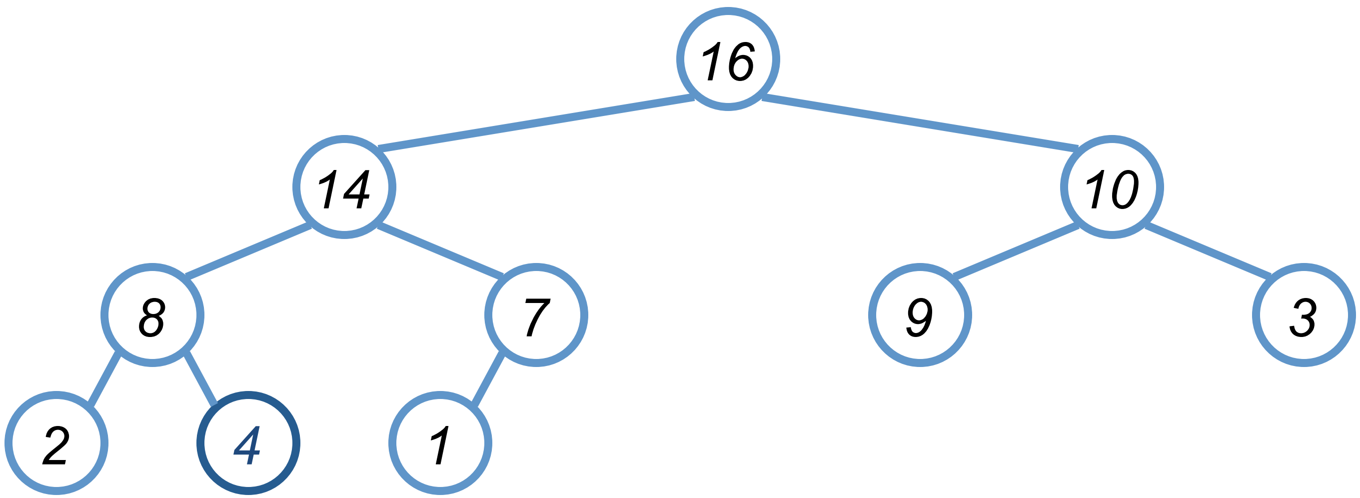
16	14	10	4	7	9	3	2	8	1

Heapify Example



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

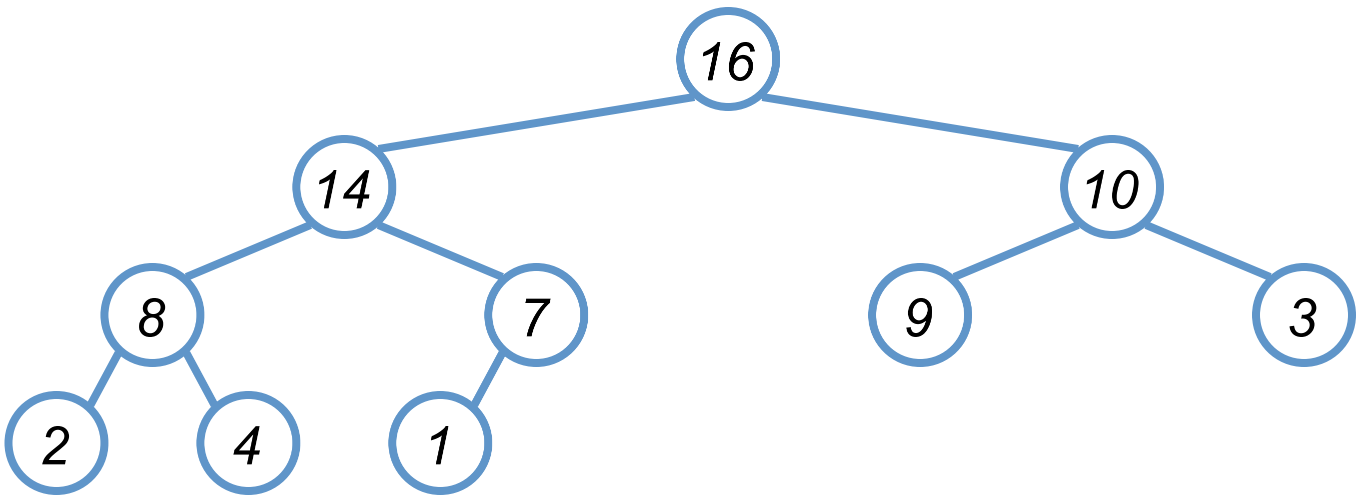
Heapify Example



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: Informal

- *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?*

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

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

- By case 2 of the Master Theorem,

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

- Thus, **`Heapify`** takes logarithmic time