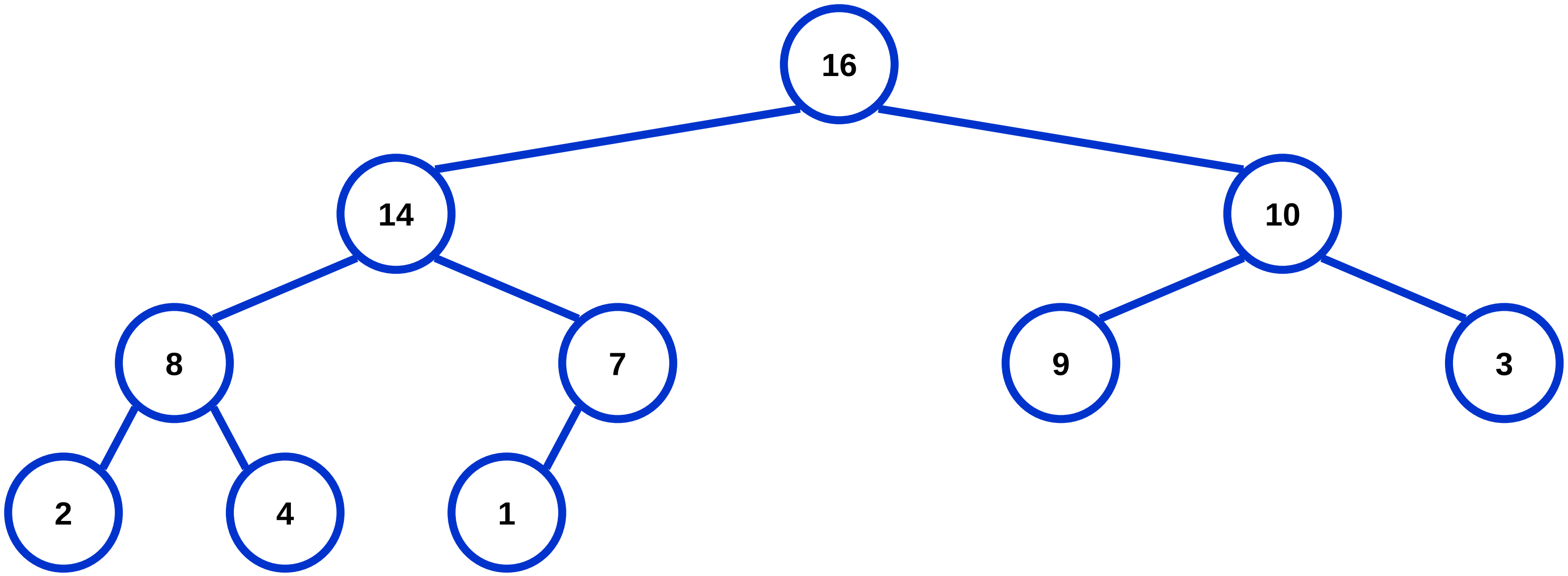
Heapsort

Heaps

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

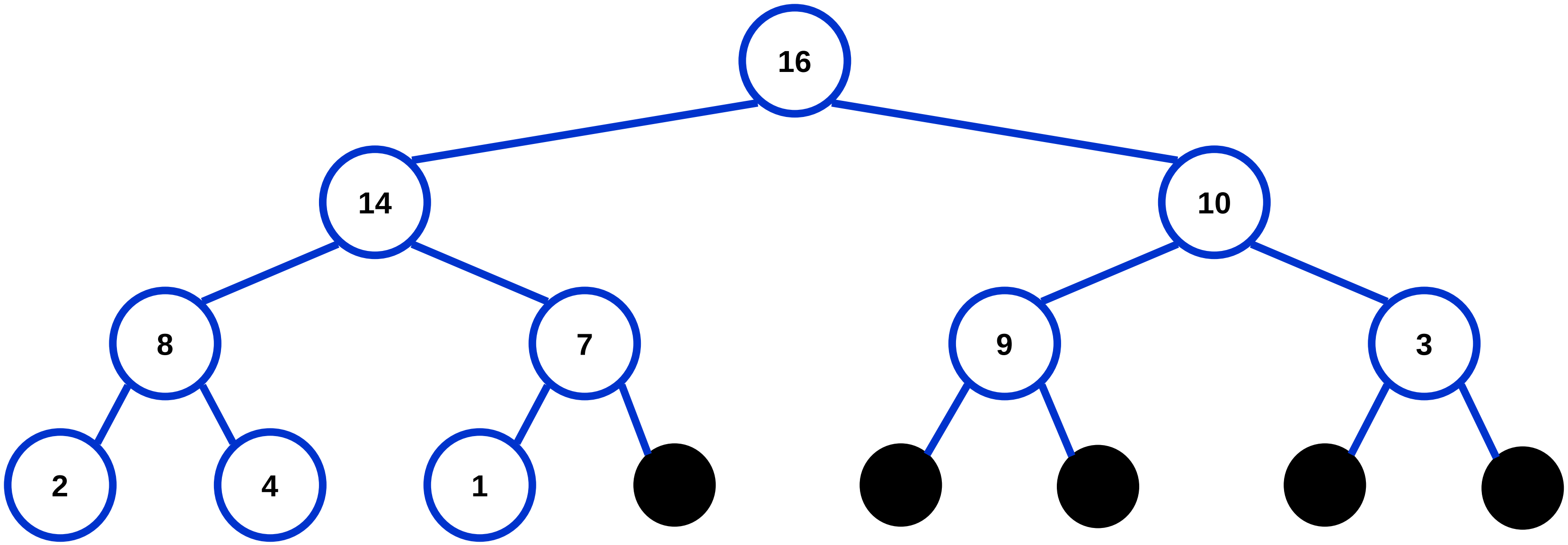


What makes a binary tree complete?

Is the example above complete?

Heaps

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

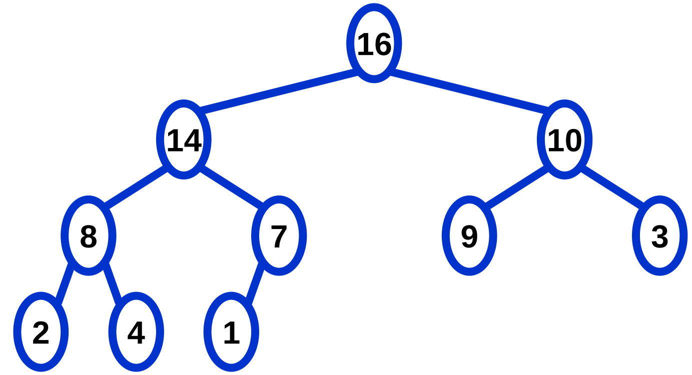


The book calls them "nearly complete" binary trees; can think of unfilled slots as null pointers

Heaps

In practice, heaps are usually implemented as arrays:

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



Heaps

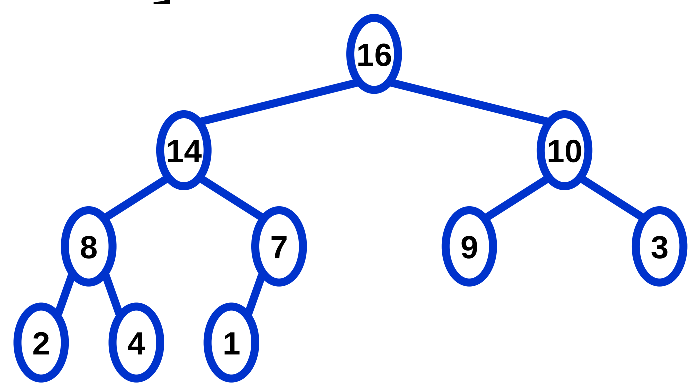
To represent a complete binary tree as an array:

- 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(A[i]) { return A[⌊i/2⌋]; }
Left(A[i]) { return A[2*i]; }
right(A[i]) { return A[2*i + 1]; }
```

The Heap Property

Heaps also satisfy the *heap property*:

A[*Parent*(A[*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?

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?*)

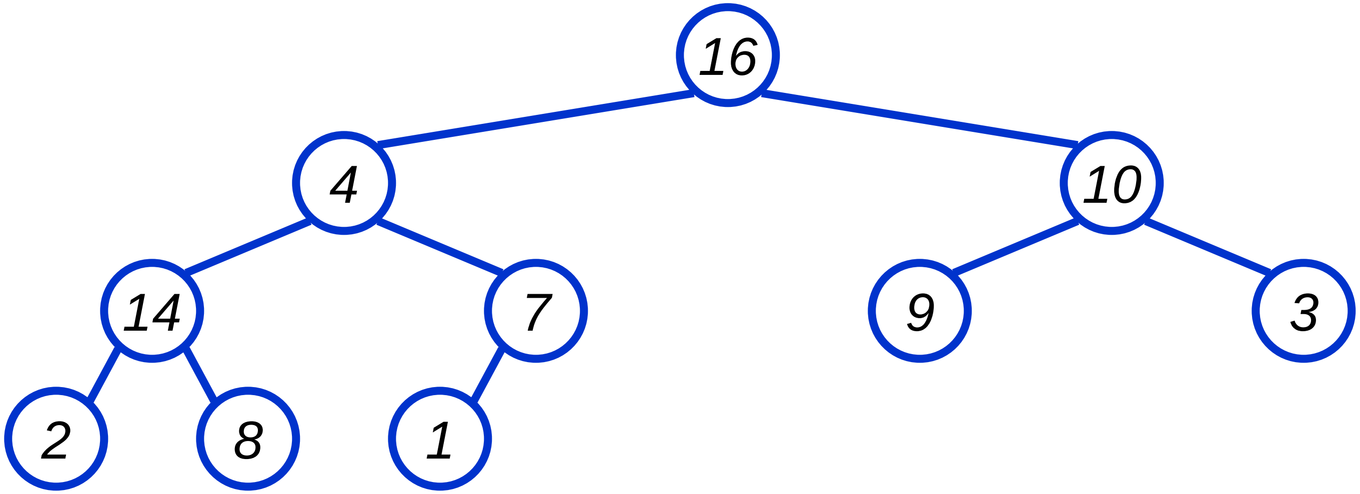
Action: let the value of the parent node “float down” so subtree at *i* satisfies the heap property

What do you suppose will be the basic operation between i, l, and r?

Heap Operations: Heapify()

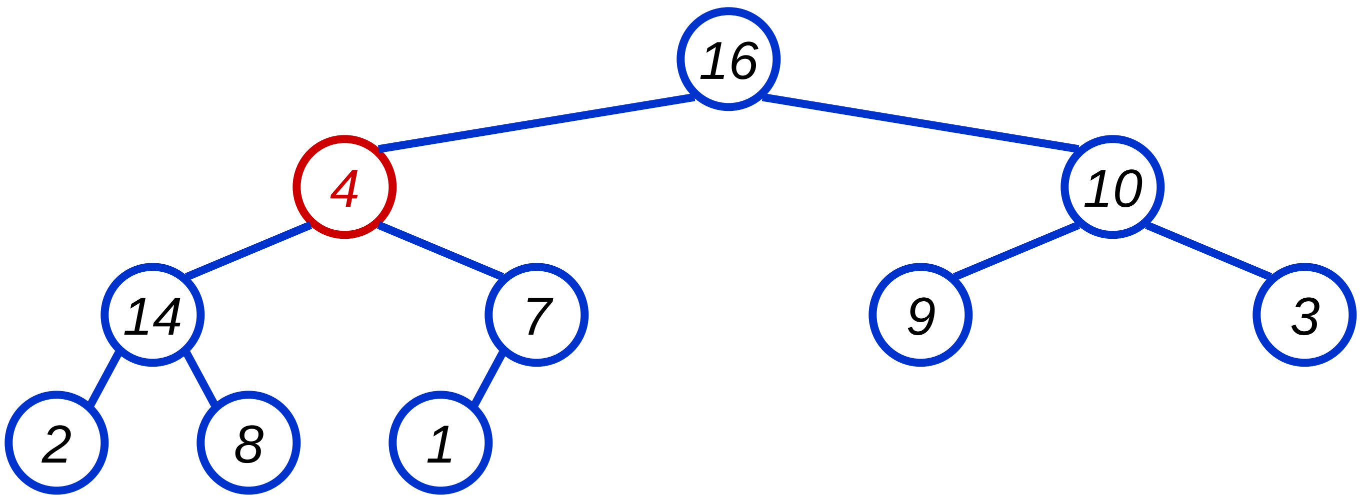
```
Heapify(A, i)
{
  l = Left(i); r = Right(i);
  if (l <= heap_size(A) && A[l] > A[i])
      largest = l;
  else
      largest = i;
  if (r <= heap_size(A) && A[r] > A[largest])
      largest = r;
  if (largest != i)
      Swap(A, i, largest);
      Heapify(A, largest);
}
```

Heapify() Example



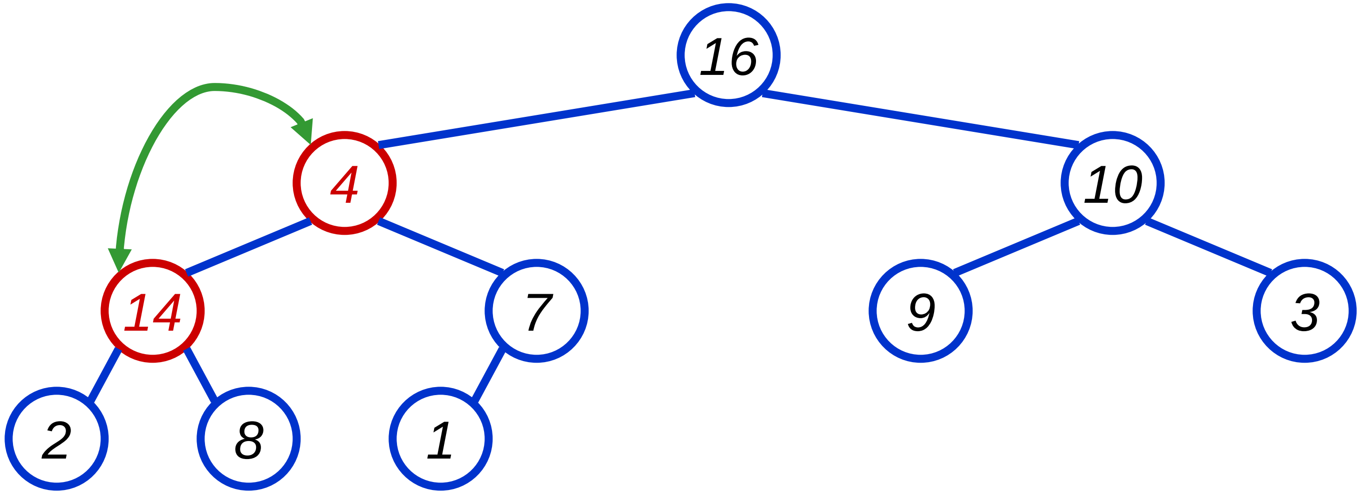
A =										
	16	4	10	14	7	9	3	2	8	1

Heapify() Example



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

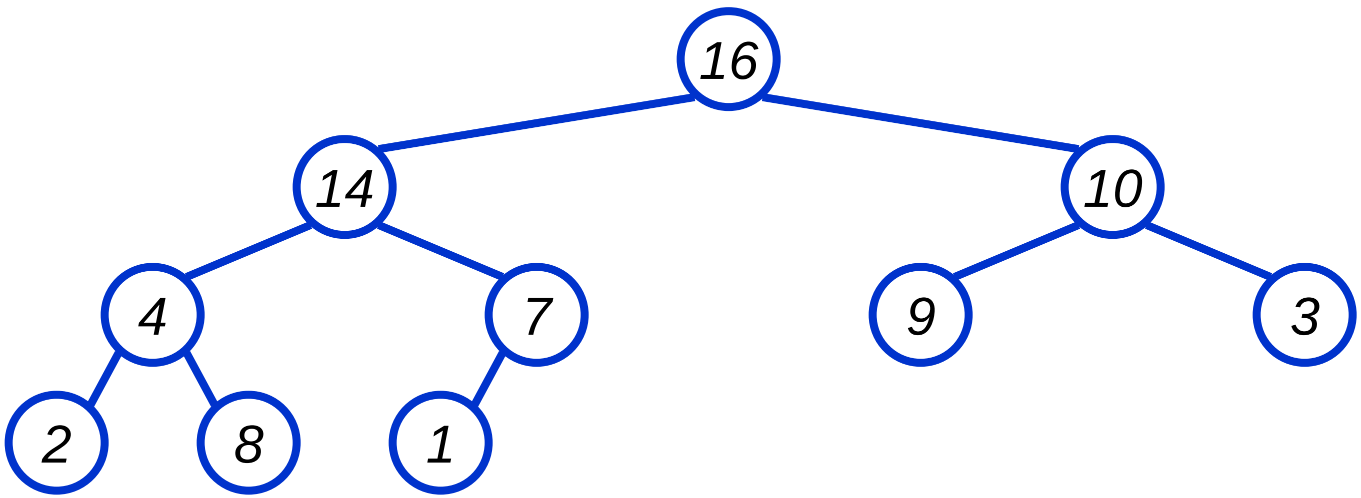
Heapify() Example



A =

16	4	10	14	7	9	3	2	8	1

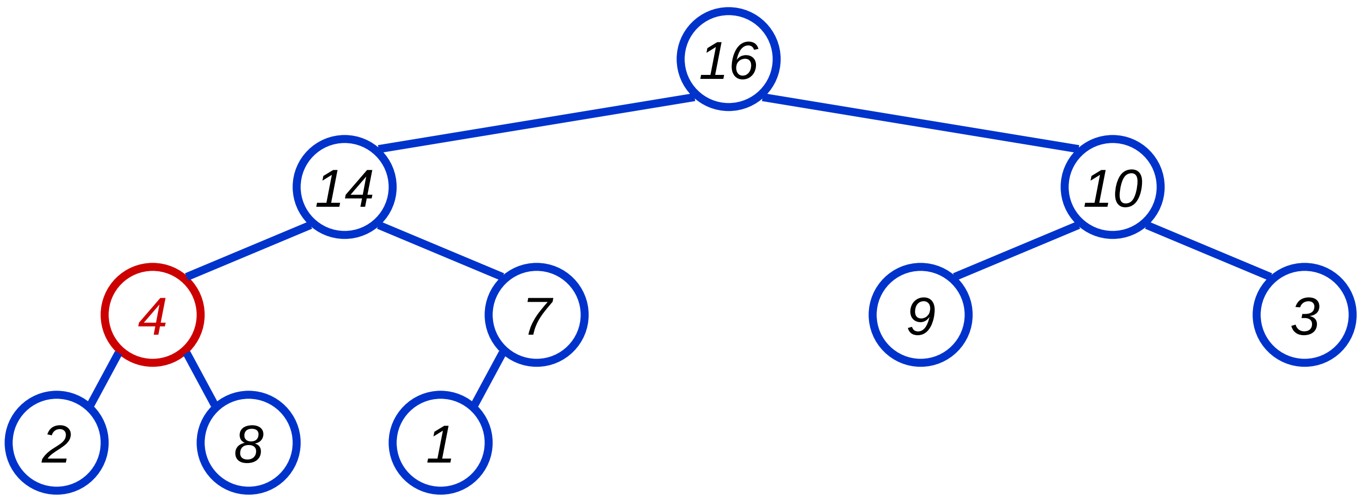
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

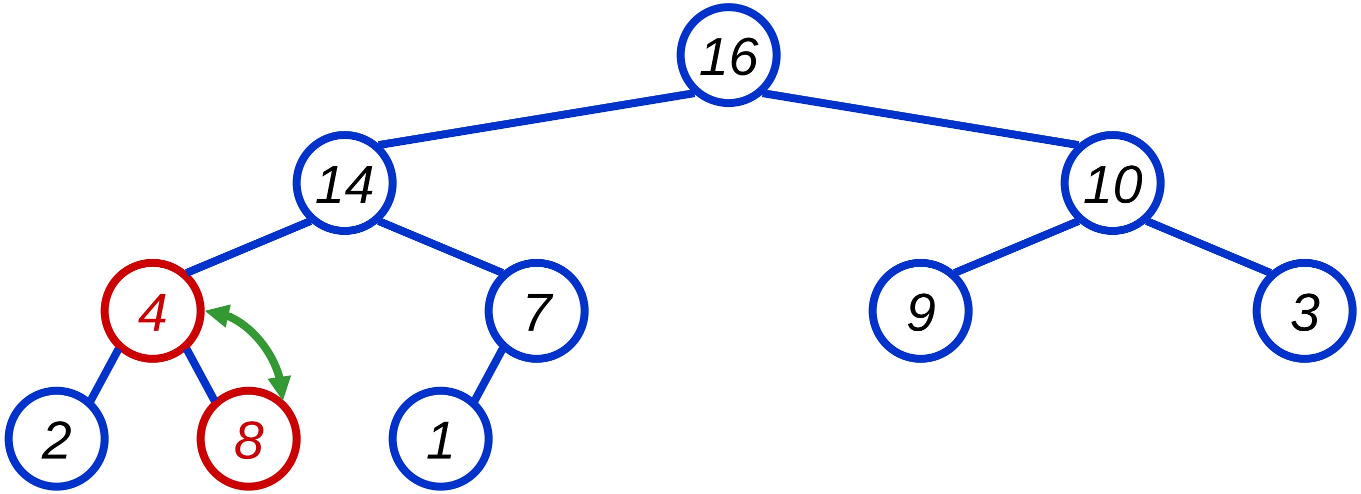
Heapify() Example



$A =$

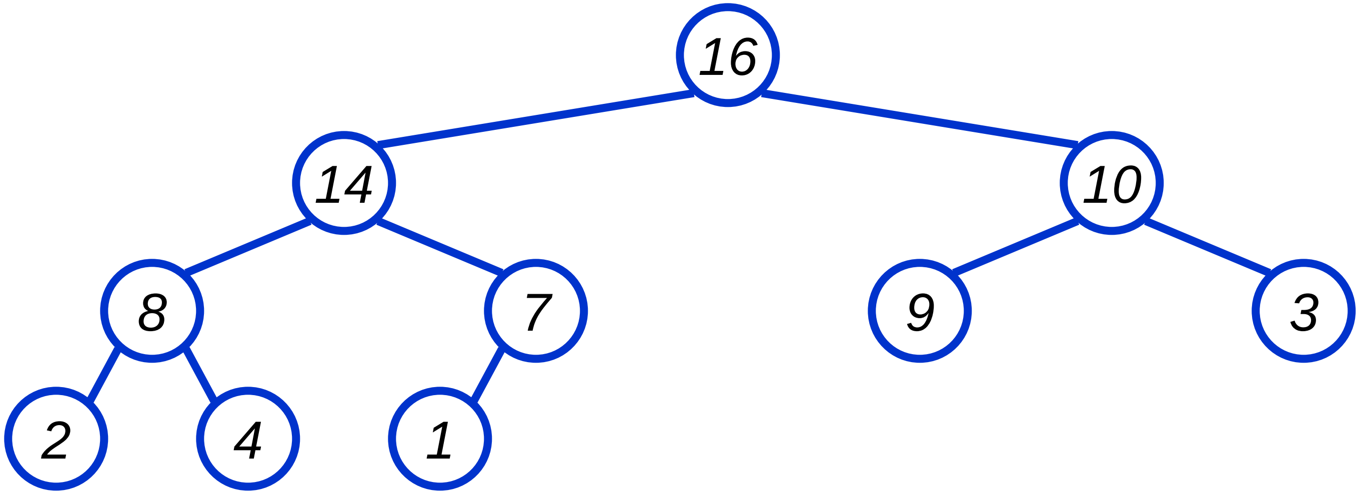
16	14	10	4	7	9	3	2	8	1

Heapify() Example



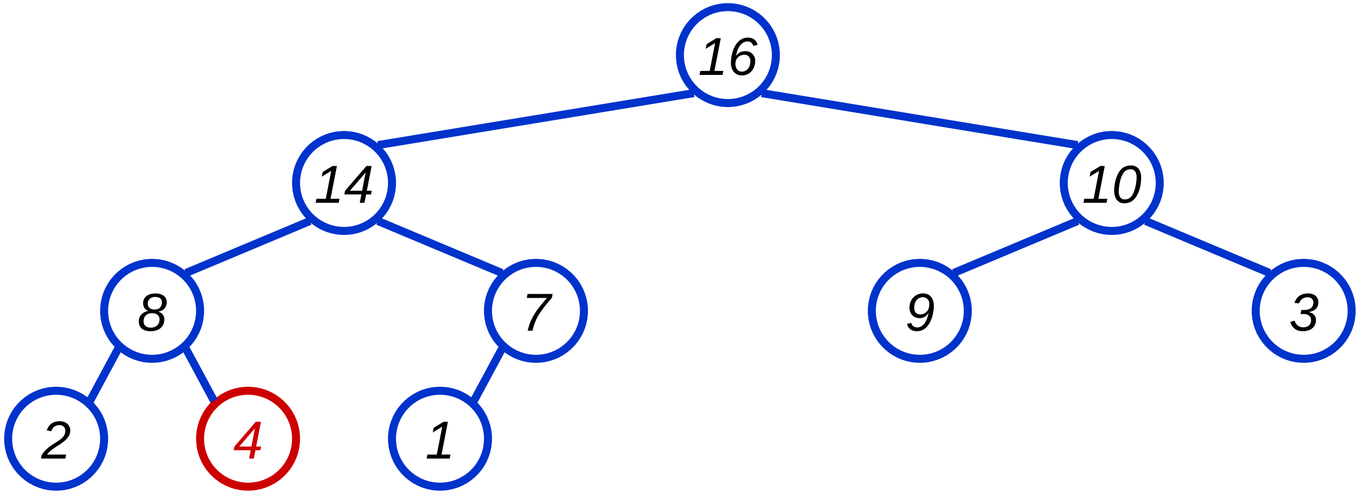
A =	16	14	10	4	7	9	3	2	8	1

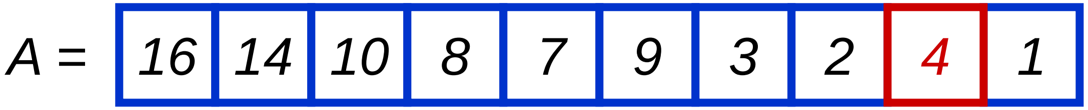
Heapify() Example



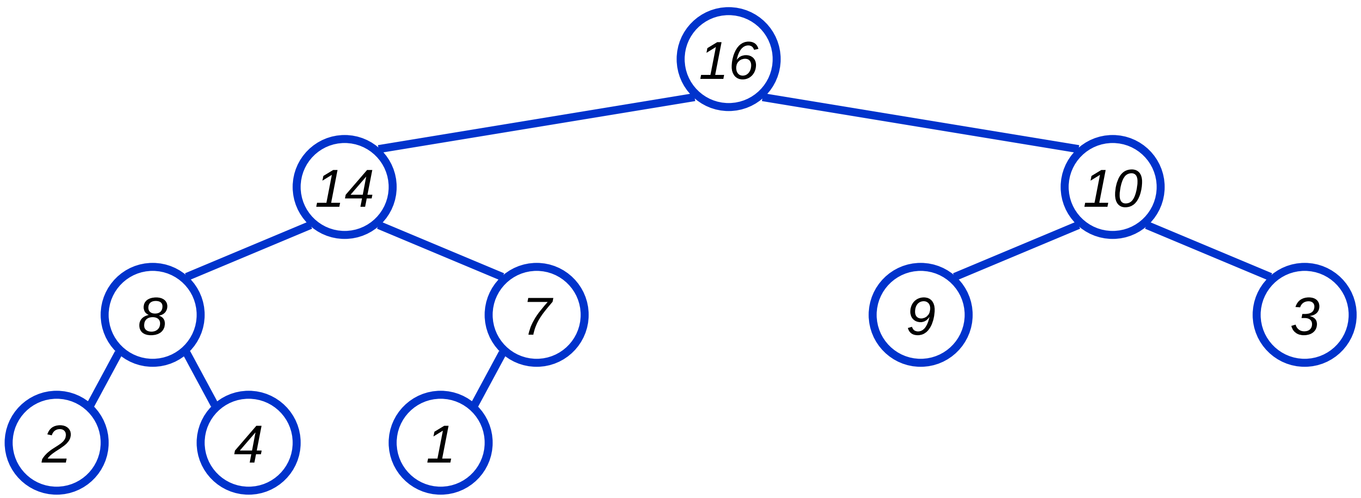
A =	*16*	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

Heapify() Example





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

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

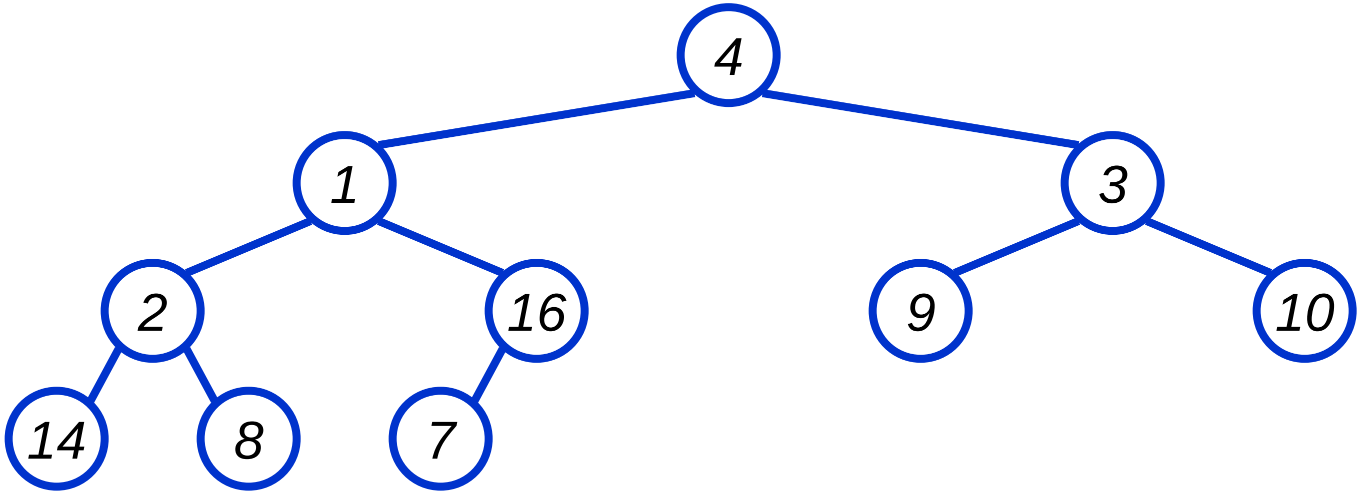
BuildHeap()

```
// 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}



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)

- *How can this be? Is there a flaw in the above reasoning?*

Analyzing BuildHeap(): Tight

To **`Heapify()`** a subtree takes O(*h*) time where *h* is the height of the subtree

- *h* = O(lg *m*), m = # nodes in subtree
- The height of most subtrees is small

Fact: an *n*-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height *h*

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 (coming up) 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?***

Implementing Priority Queues with Heaps

The heap data structure with the `Heapify-down` and `Heapify-up` operations can efficiently implement a priority queue that is constrained to hold at most N elements at any point in time. Here we summarize the operations we will use.

- `StartHeap`(N) returns an empty heap H that is set up to store at most N elements. This operation takes $O(N)$ time, as it involves initializing the array that will hold the heap.
- `Insert`(H, v) inserts the item v into heap H. If the heap currently has n elements, this takes $O(\log n)$ time.
- `FindMin`(H) identifies the minimum element in the heap H but does not remove it. This takes $O(1)$ time.
- `Delete`(H, i) deletes the element in heap position i. This is implemented in $O(\log n)$ time for heaps that have n elements.
- `ExtractMin`(H) identifies and deletes an element with minimum key value from a heap. This is a combination of the preceding two operations, and so it takes $O(\log n)$ time.