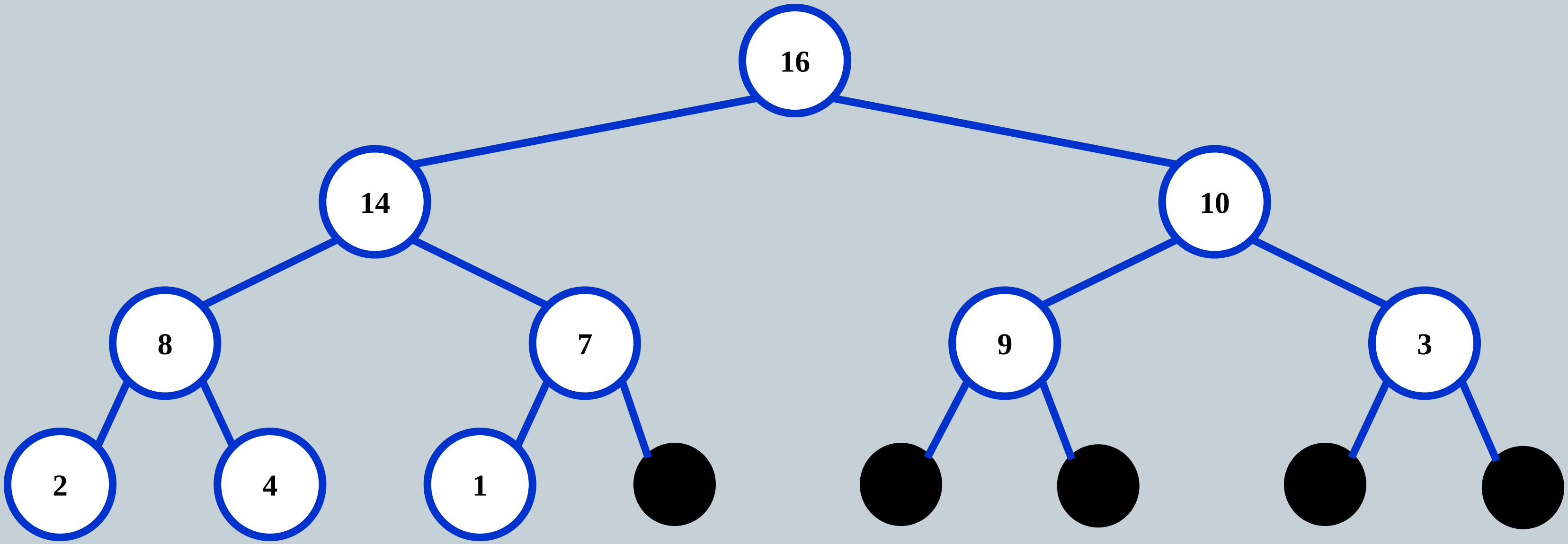
ALGORITHMS & ADVANCED DATA STRUCTURES (#5)

HEAPS-HEAPSORT

ADAPTED FROM
CS 146 SJSU (KATERINA POTIKA)

Heaps (chapter 6)

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

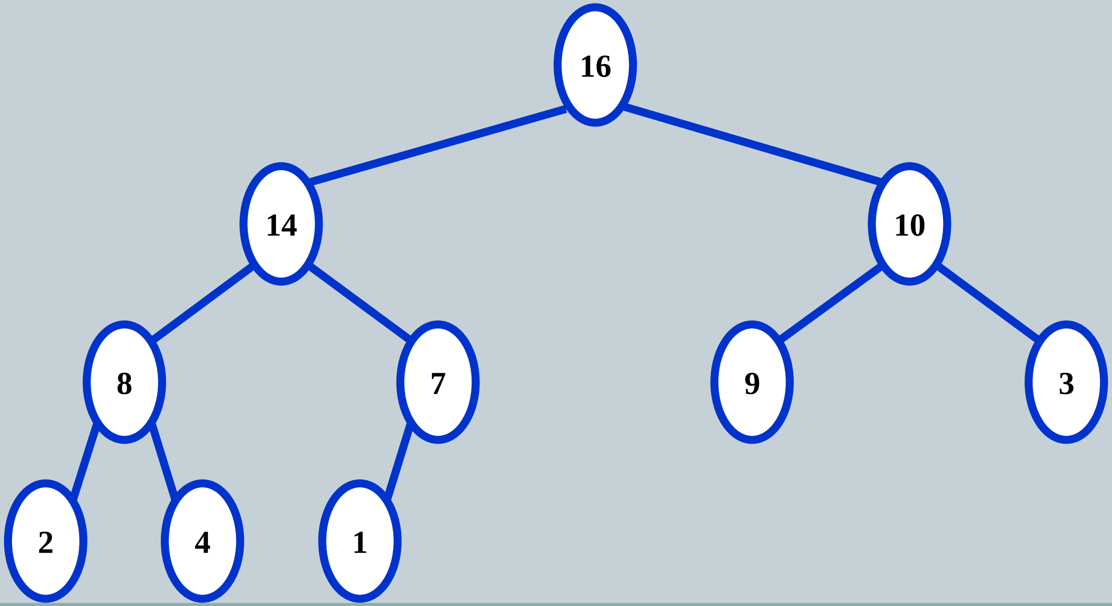


“nearly complete” binary trees: you can think of unfilled slots as null pointers

Heaps

In practice, heaps are usually implemented as arrays:

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	
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*]

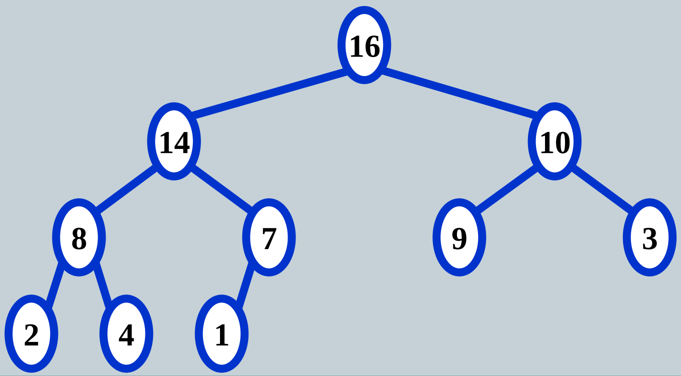
The parent of node *i* is A[*i*/2] (note: integer divide)

The left child of node *i* is A[2*i*]

The right child of node *i* is A[2*i* + 1]

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
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; }
```

How would you implement this most efficiently?

The Heap Property

Heaps also satisfy the ***heap property***:

A[*parent*(*i*)] ≥ A[*i*]   for all nodes $i > 1$

i.e. , the value of a node is at most the value of its parent

In case of **Max Heap**

Where is the largest element in a heap stored?

- 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

Self test

- Is the array with values 23; 17; 14; 6; 13; 10; 1; 5; 7; 12 a max-heap?

Heap Height

What is the height of an n-element heap?

This is nice property: all basic heap operations take at most time proportional to the height of the heap!

Heap Operations: Heapify()

Heapify(): maintain the heap property

- *Input: a node i in the heap with children l and r*
  - *& two subtrees rooted at l and r, assumed to be heaps*

The subtree rooted at *i* may violate the heap property (*Give an example…*)

- *Output: the tree rooted at i is a heap*

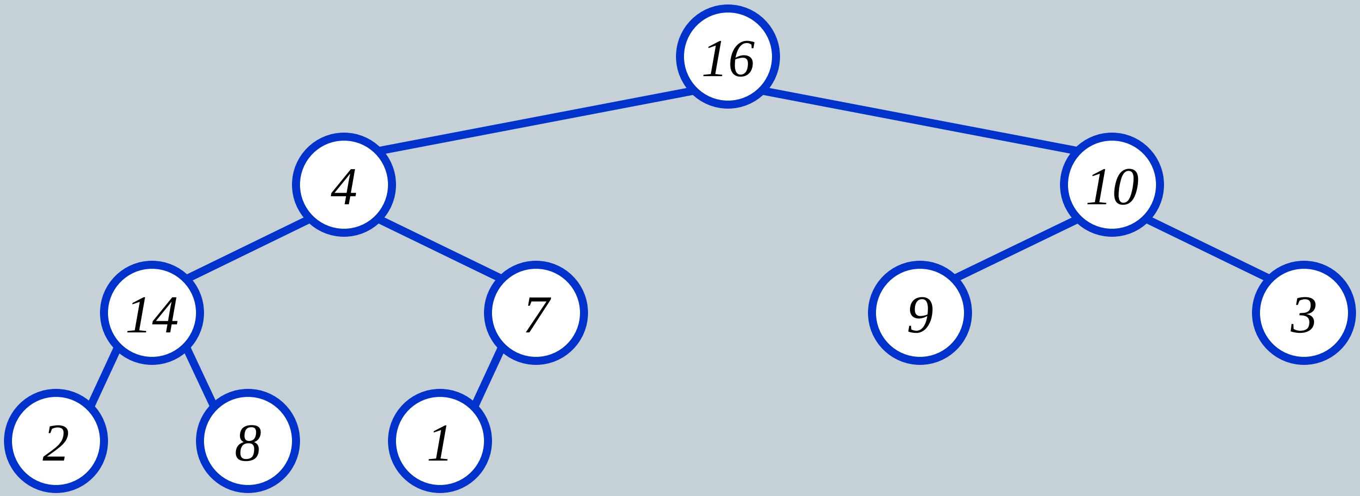
Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property

- *What basic operation between i, l, and r must be used?*

Heap Operations: Heapify()

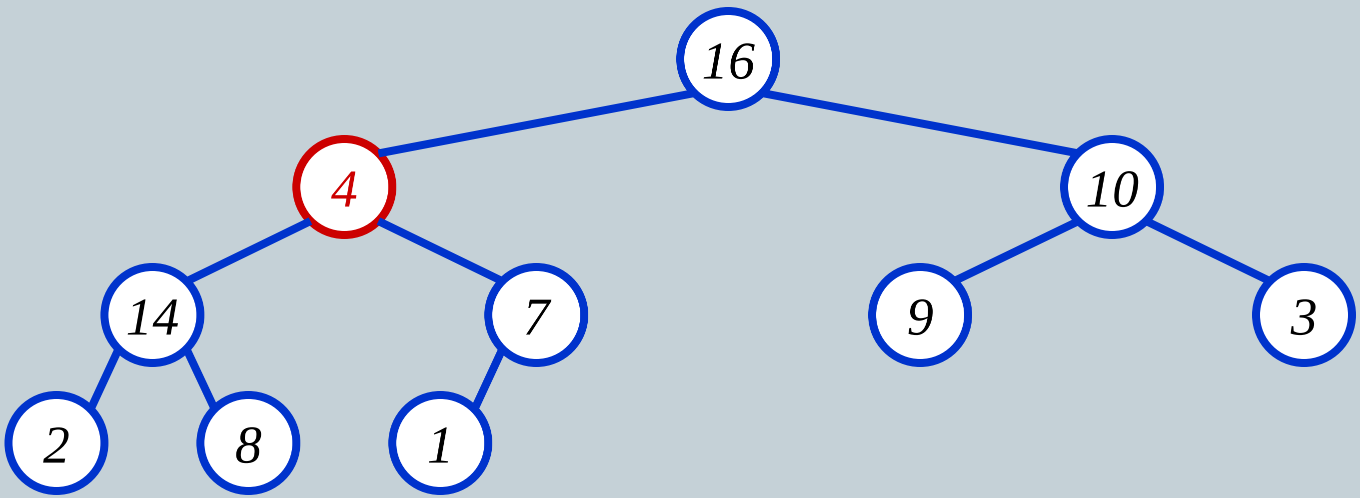

```
Heapify(A, i)

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

Heapify() Example



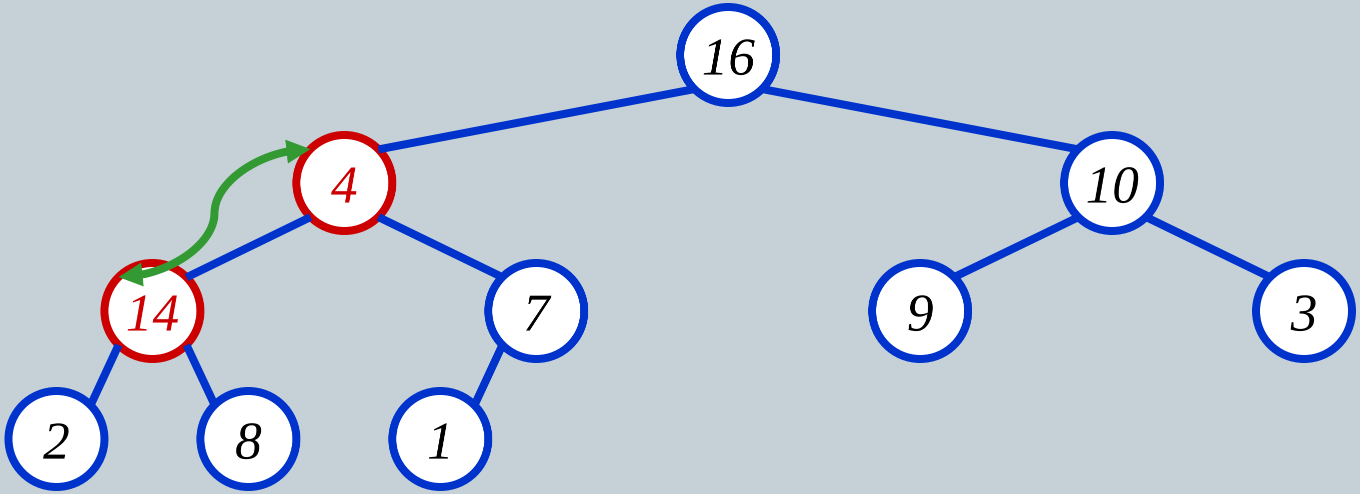
	1	2	3	4	5	6	7	8	9	10
$A =$	16	4	10	14	7	9	3	2	8	1

Heapify() Example



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

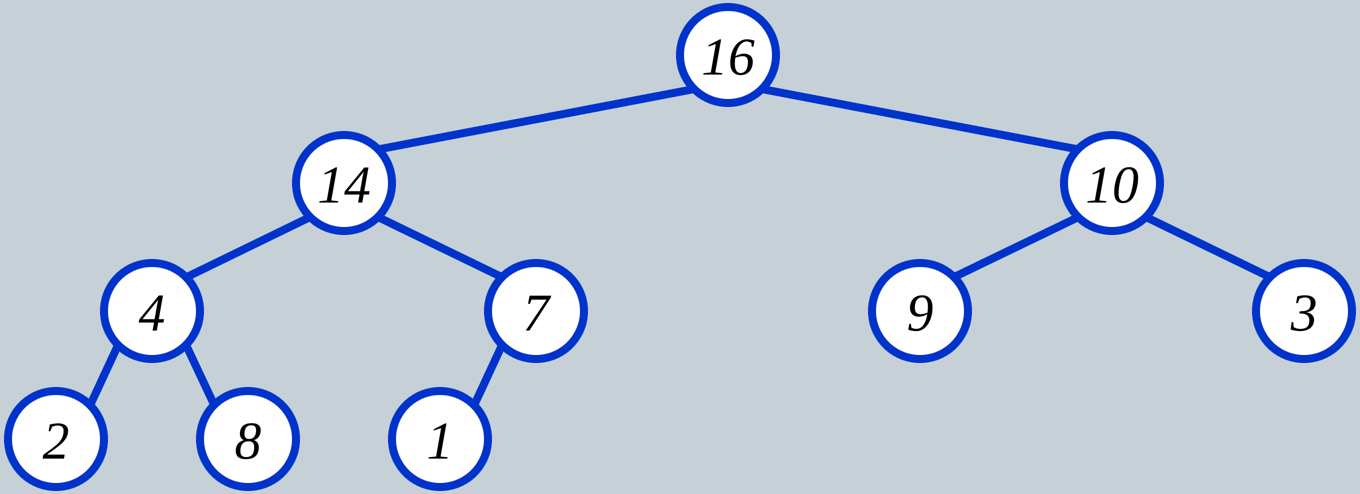
Heapify() Example



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

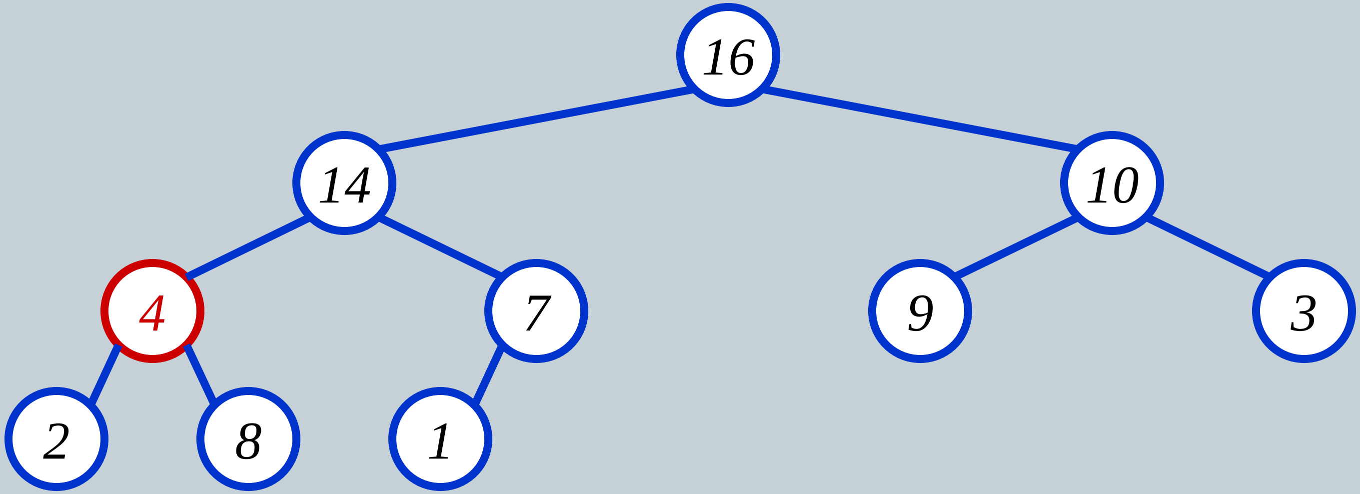
A =

Heapify() Example



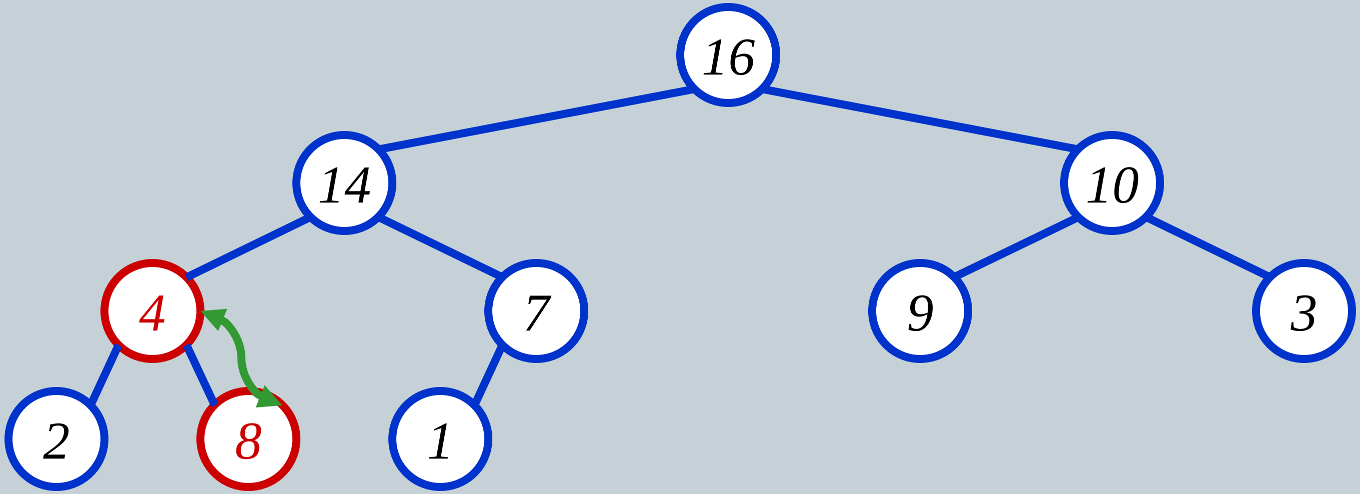
	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	4	7	9	3	2	8	1

Heapify() Example



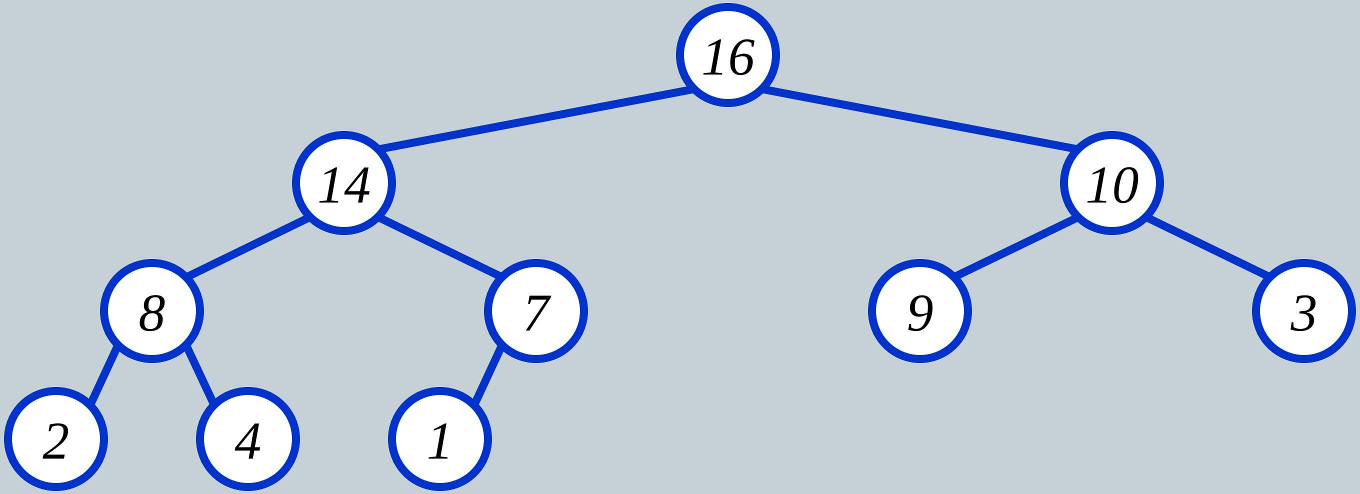
	1	2	3	4	5	6	7	8	9	10
$A =$	16	14	10	4	7	9	3	2	8	1

Heapify() Example



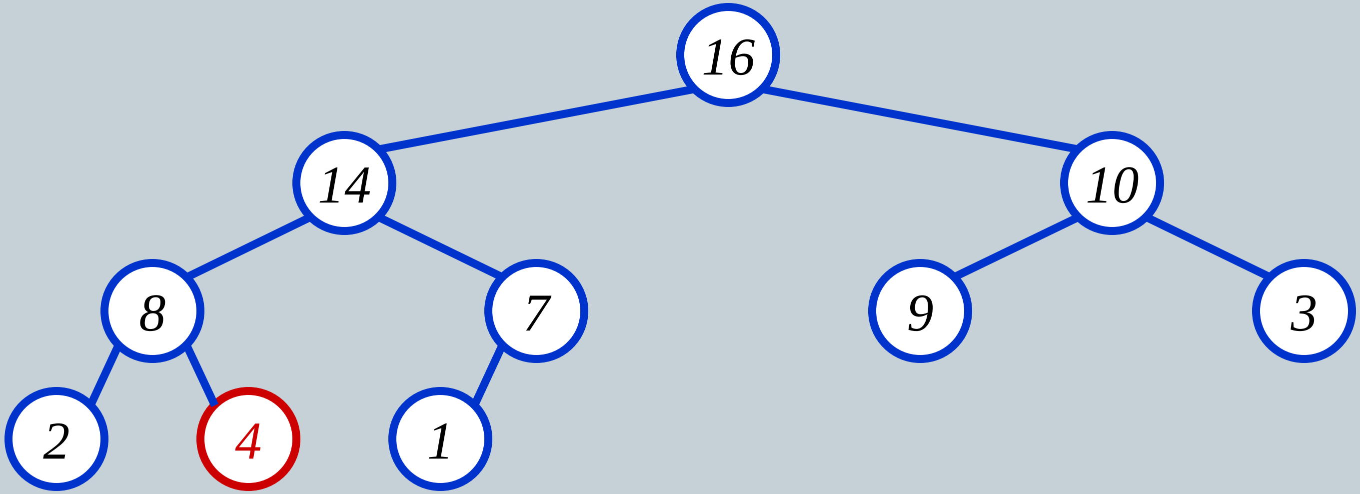
	1	2	3	4	5	6	7	8	9	10
$A =$	16	14	10	4	7	9	3	2	8	1

Heapify() Example



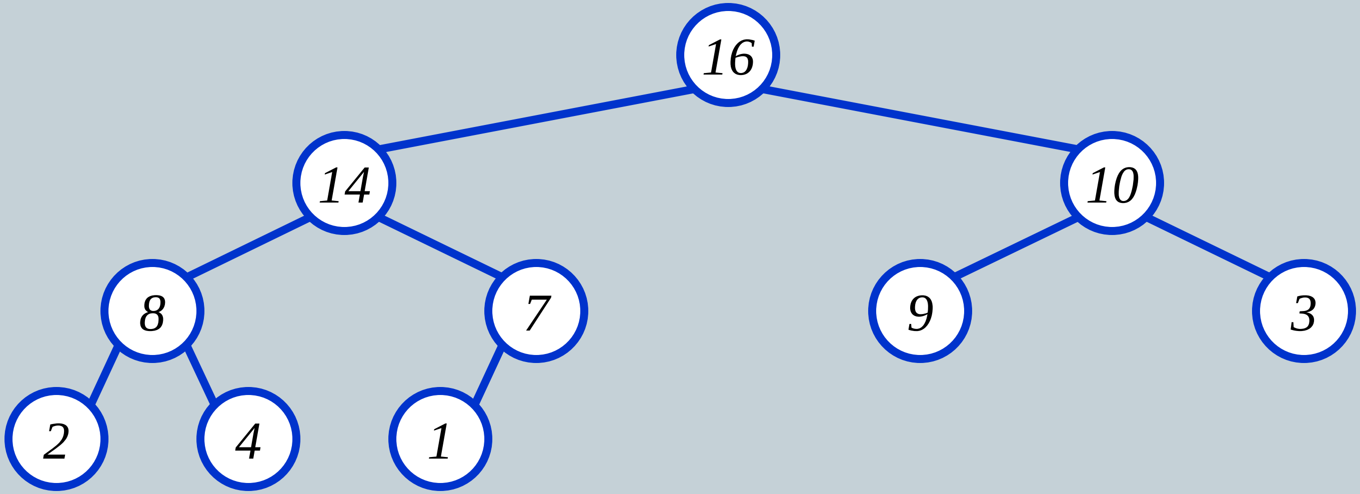
	1	2	3	4	5	6	7	8	9	10
$A =$	16	14	10	8	7	9	3	2	4	1

Heapify() Example



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

Heapify() Example



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

Analyzing Heapify()

Except 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

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

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 logarthmic 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 is that?*)

Key idea:

- 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

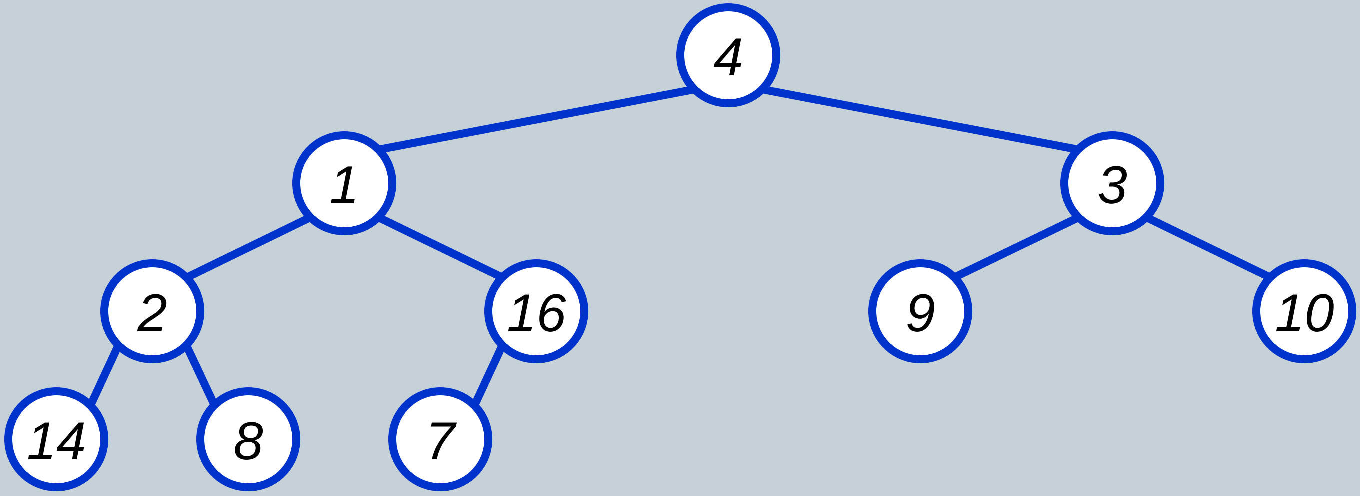
BuildHeap()


```
// given an unsorted array A, make A a heap
BuildHeap(A)
{
  A.heap_size = A.length;
  for (i = ⌊A.length/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(log *n*) time

There are O(*n*) such calls ($\lfloor n/2 \rfloor$)

Thus the running time is O(*n* log *n*)

Is this a correct asymptotic upper bound?

Is this an asymptotically tight bound?

Can we do better?

A tighter bound is O(*n*)

Is there a flaw in the above reasoning? No more careful analysis

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

Uses this fact to prove that **`BuildHeap()`** takes O(n) time

BuildHeap() is O(n)!

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

- Due X=1/2 it is

(see A.8 textbook)

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

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

Self Test

- illustrate the operation of BUILD-MAX-HEAP on the array 5; 3; 17; 10; 84; 19; 6; 22; 9.

Heapsort!

Given **BuildHeap()**, an in-place sorting algorithm is easily constructed, key idea:

- 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 = A.length downto 2)
   Swap(A[1], A[i]);
   A.heap_size(A) -= 1;
   Heapify(A, 1);
```

Example

- 21, 14, 16, 12, 10, 4, 8
- HeapSort using min heap

Analyzing Heapsort

The call to **BuildHeap()** takes O(n) time

Each of the n - 1 calls to **Heapify()** takes O(log 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?***

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?

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?

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?

Maximum and ExtractMaximum


```
HeapMaximum(A)
   Return A[1]

HeapExtractMax(A)
{
    if (A.heap_size < 1) { error }
    max = A[1]
    A[1] = A[A.heap_size]
    A.heap_size --
    Heapify(A, 1)
    return max
}
```

HeapExtractMax

O(lgn) (same as Heapify)

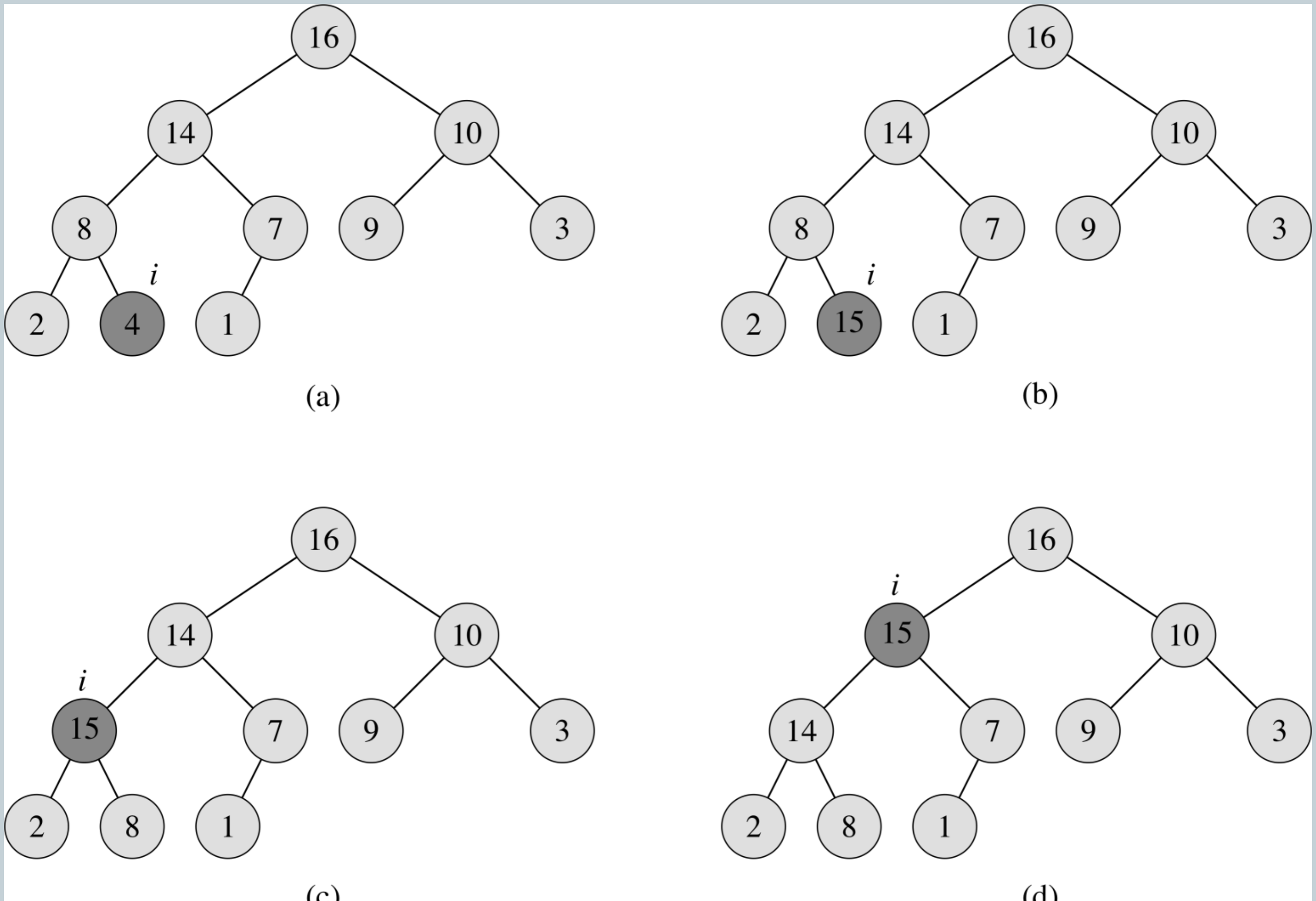
IncreaseKey


```
IncreaseKey(A,i,key)
{
  if key<A[i] error "new key is smaller than current"
  A[i]=key
  while (i>1 && A[parent(i)]<A[i])
    swap(A[i],A[parent(i)])
    i=parent(i)
}
```

IncreaseKey complexity

O(lgn) why?

example



(a) (b) (c) (d)

Insert


```
Insert(A,key)
{
   A.heap_size++
    A[A.heap_size]=-infinity
    IncreaseKey(A,A.heap_size, key)
}
```

O(lgn) complexity