403: Algorithms and Data Structures

Heaps

Fall 2016

UAlbany

Computer Science

Some slides borrowed by [David Luebke]

Birdseye view plan

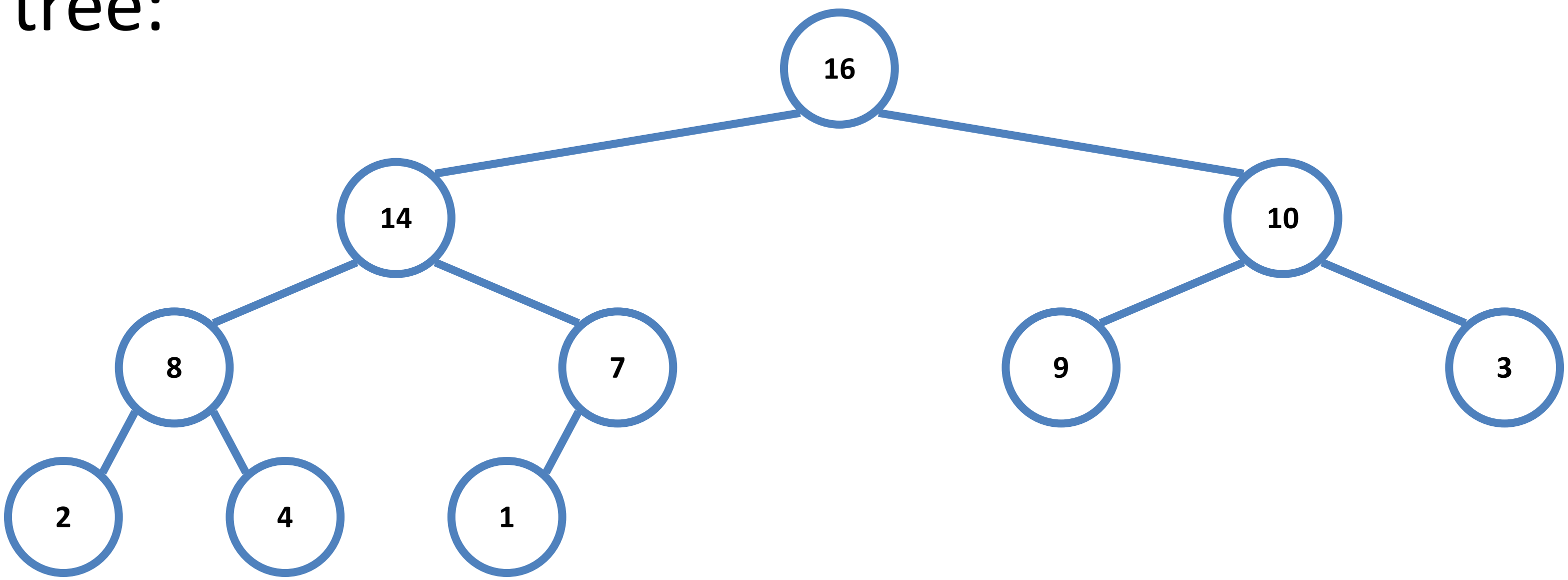
- For the next several lectures we will be looking at sorting and related problems
- Assume:
  – Input is a sequence of n numbers
  – Note: practical cases of other data than numbers can also be handled

Sorting Revisited

- So far we've talked about two algorithms to sort an array of numbers
  - What is the advantage of merge sort?
  - What is the advantage of insertion sort?
- Next on the agenda: *Heapsort*
  - Combines advantages of both previous algorithms
    - In-place
    - $O(n \log n)$
  - Uses a new data structure: Binary Heap

Heaps

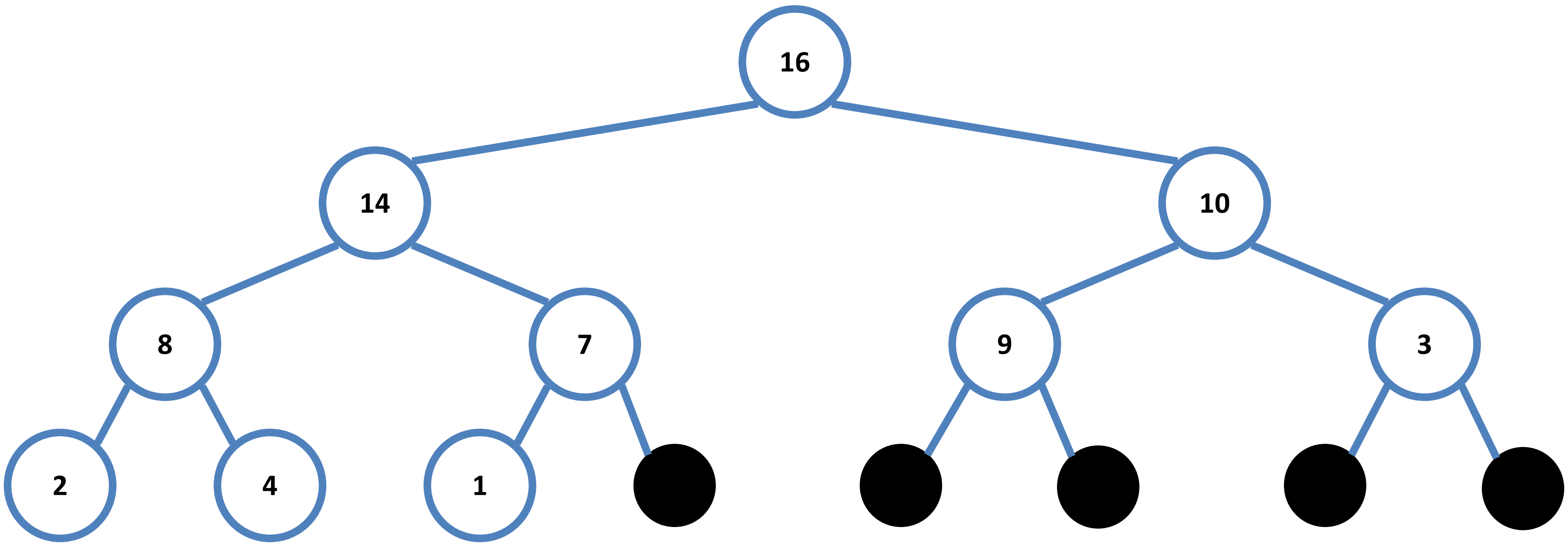
- A (binary) *heap* can be seen as a complete binary tree:



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

Heaps

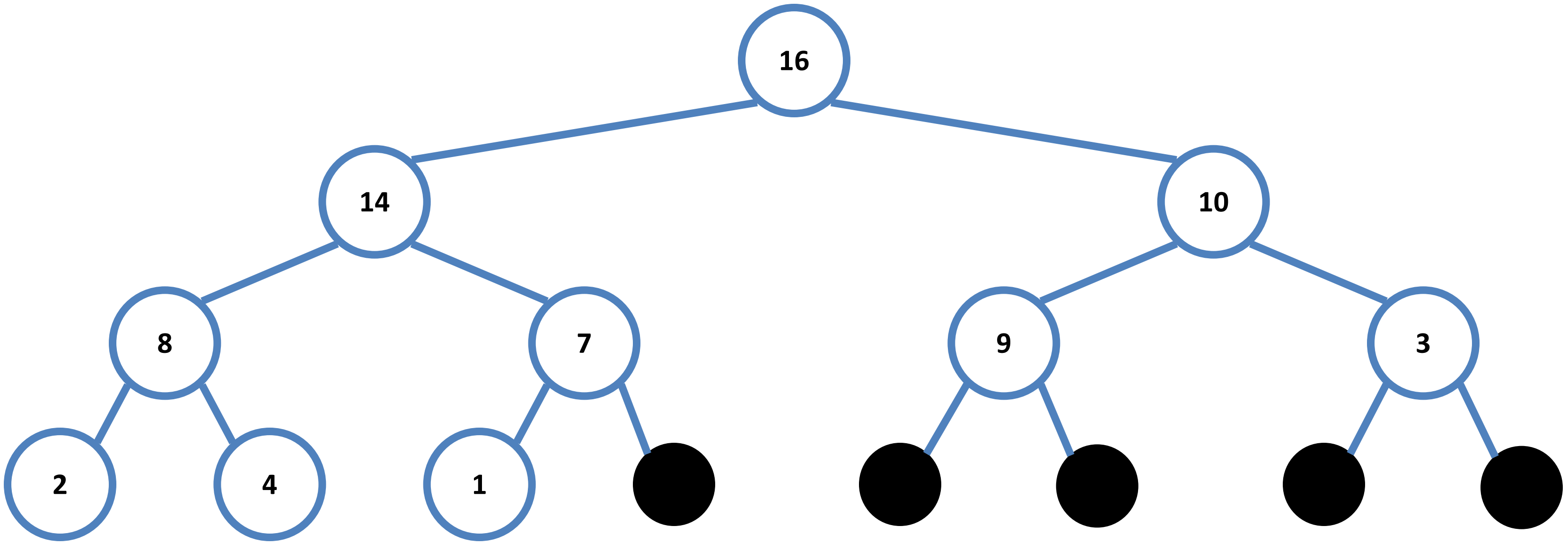
- A *heap* can be seen as a complete binary tree:



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

Heaps

- The lowest level is filled left to right



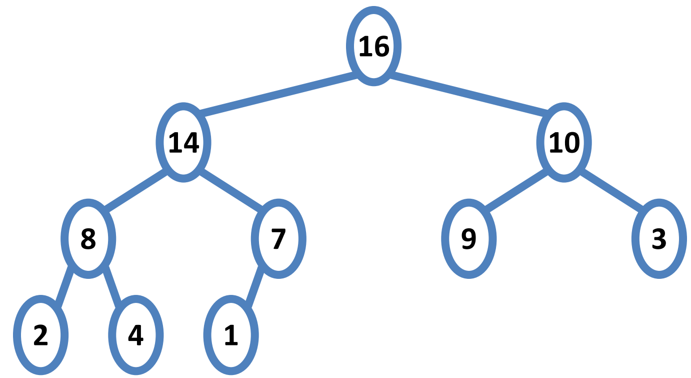
Heaps

- In practice, heaps are usually implemented as arrays:

A =

16	14	10	8	7	9	3	2	4	1

=



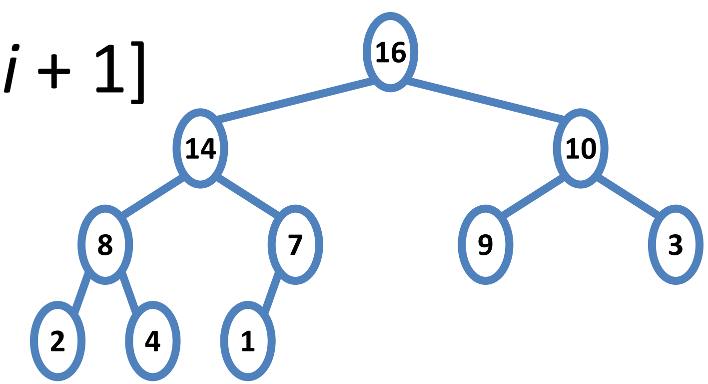
Heaps

- 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 $A[\lfloor i/2 \rfloor]$
  - 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

=

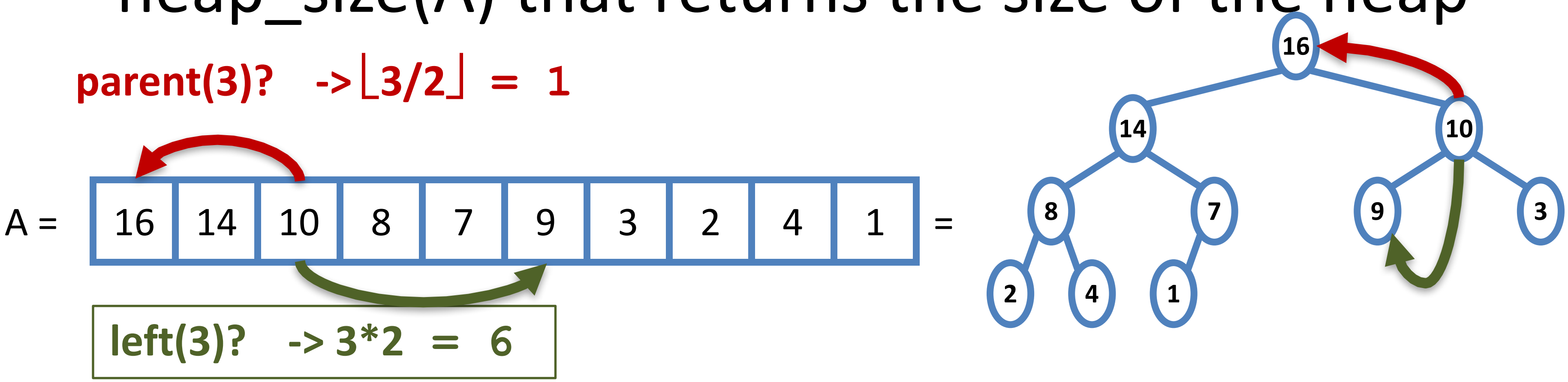


Referencing Heap Elements

- So…

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

- We will also assume we have a function heap_size(A) that returns the size of the heap



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?*

- [Refresh] of tree Definitions:
  – The *height* of a node in the tree = the number of edges on the longest downward path to a leaf

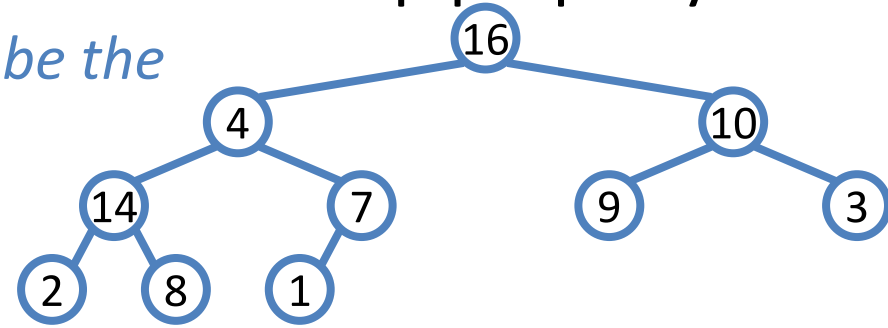
  – The height of a tree = the height of its root

Heap Height

- *What is the height of an n-element heap? Why?*
- Basic heap operations take at most time proportional to the height of the heap
- THIS IS NICE!

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?*



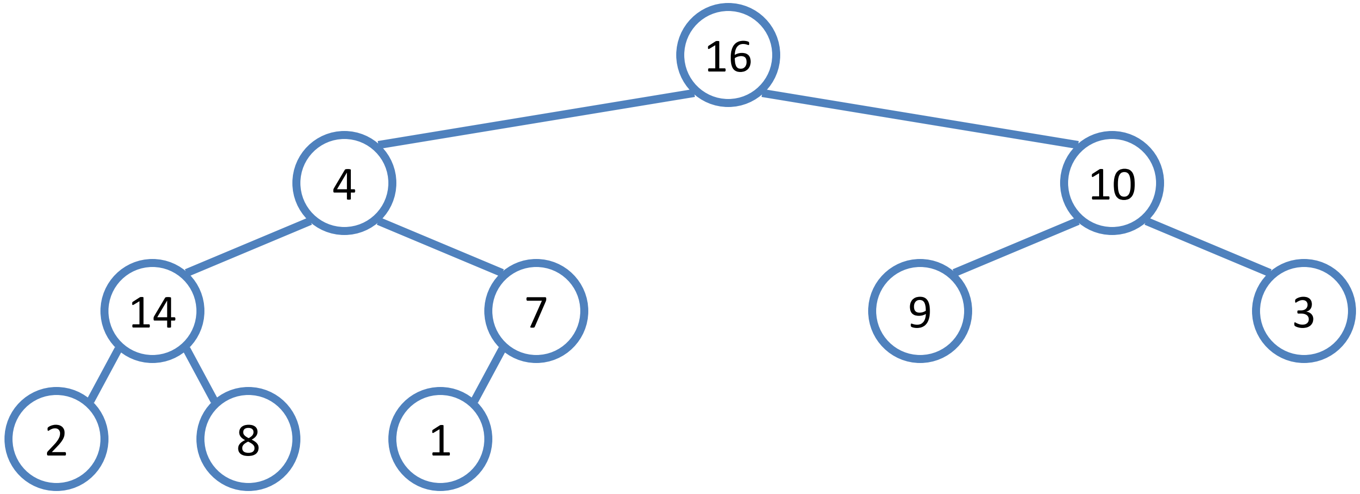
Heap Operations: Heapify()

```
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);
}
```

Carefully check boundary conditions

Finds the index of max(A[i], A[l],A[r])

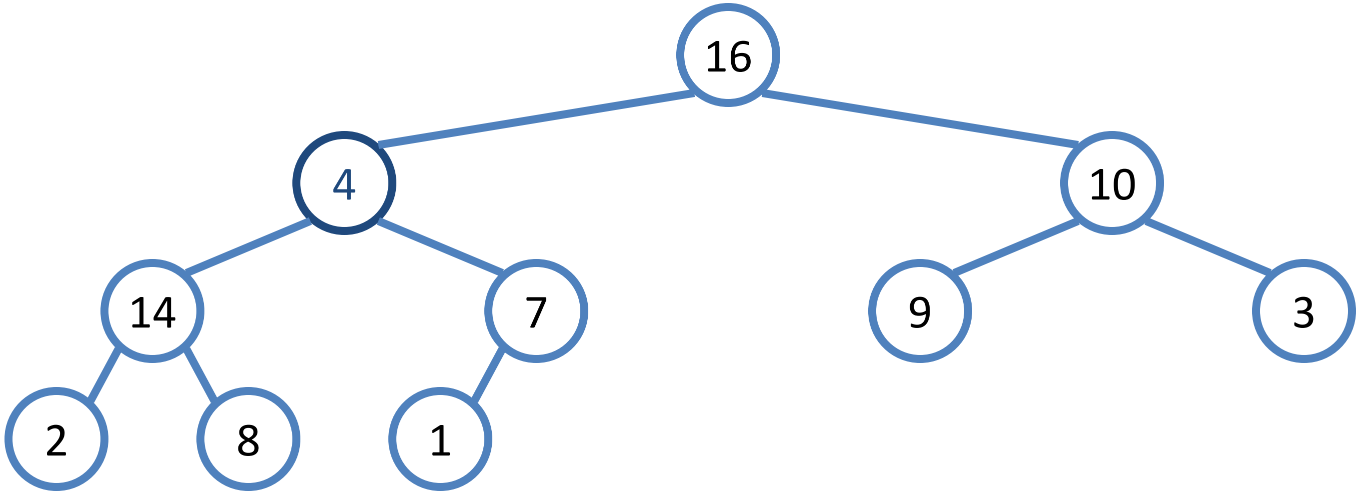
Heapify() Example



A =

16	4	10	14	7	9	3	2	8	1

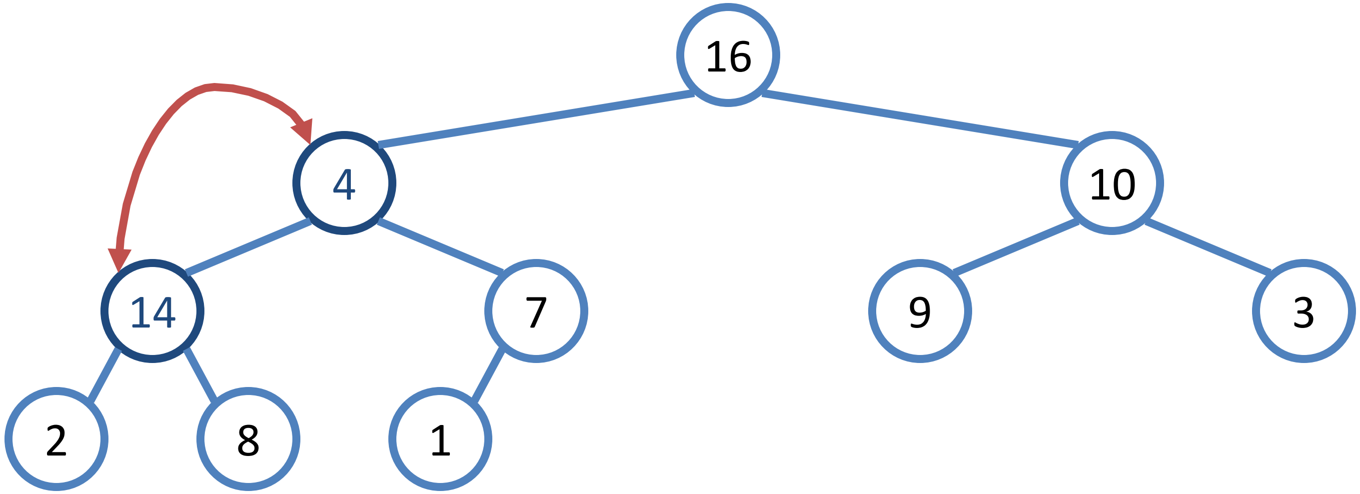
Heapify() Example



A =

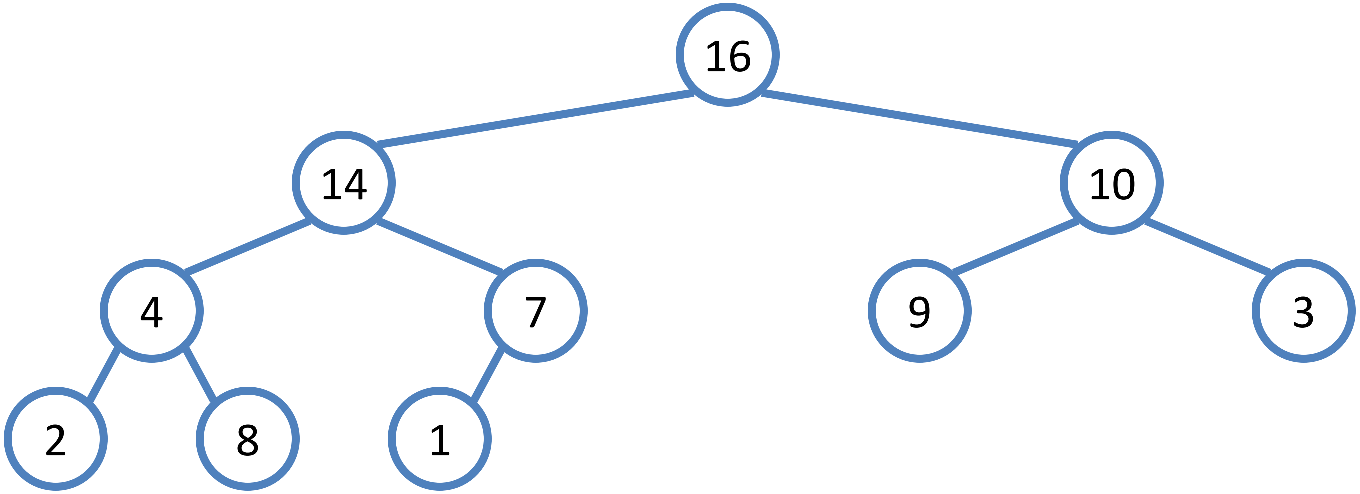
16	4	10	14	7	9	3	2	8	1

Heapify() Example



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

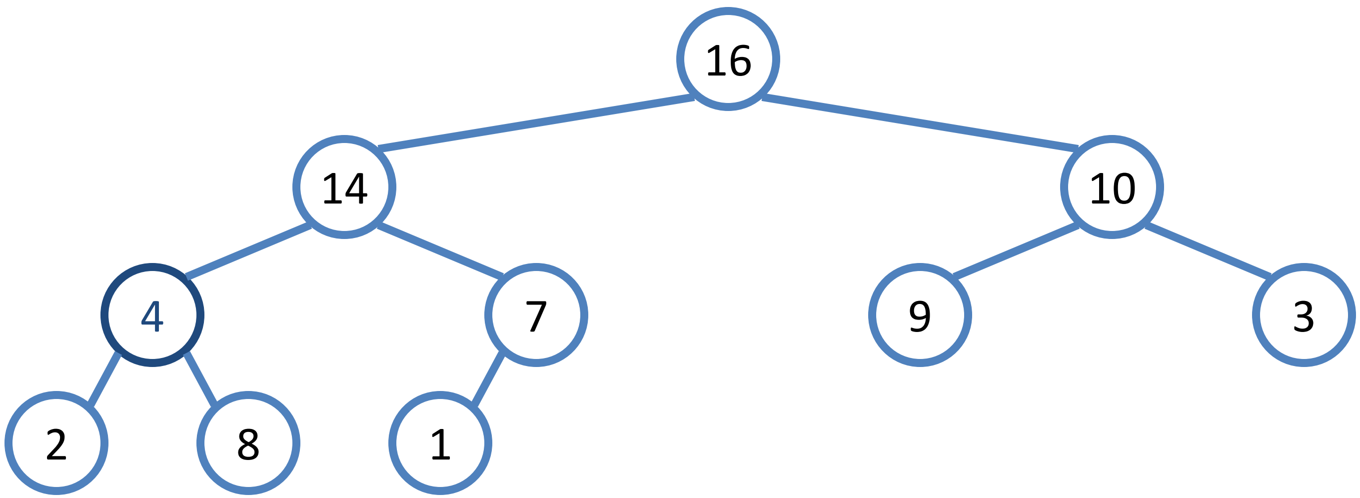
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

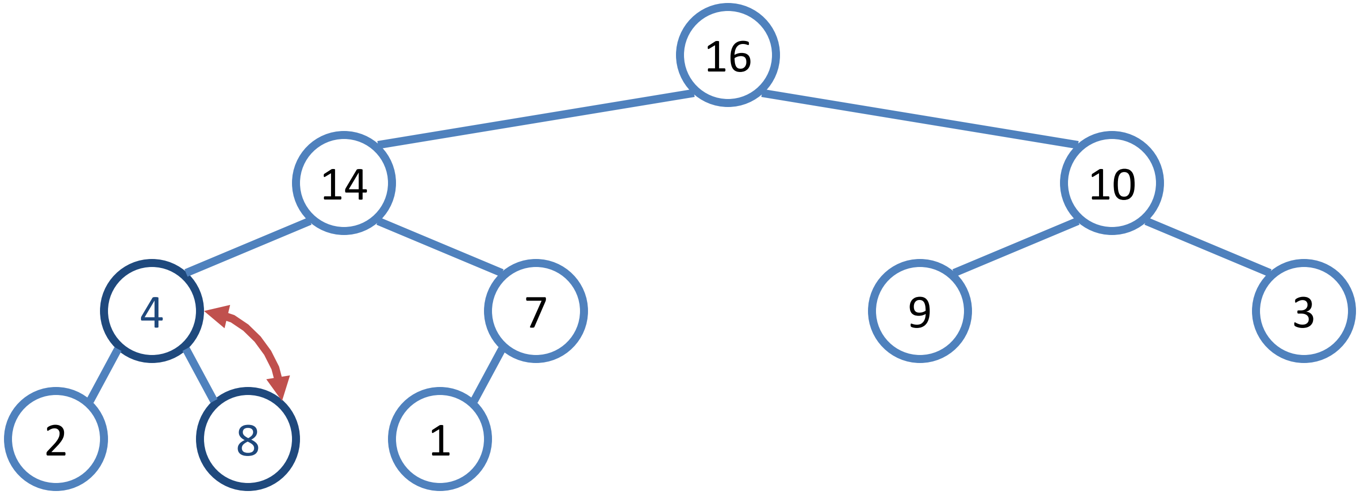
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

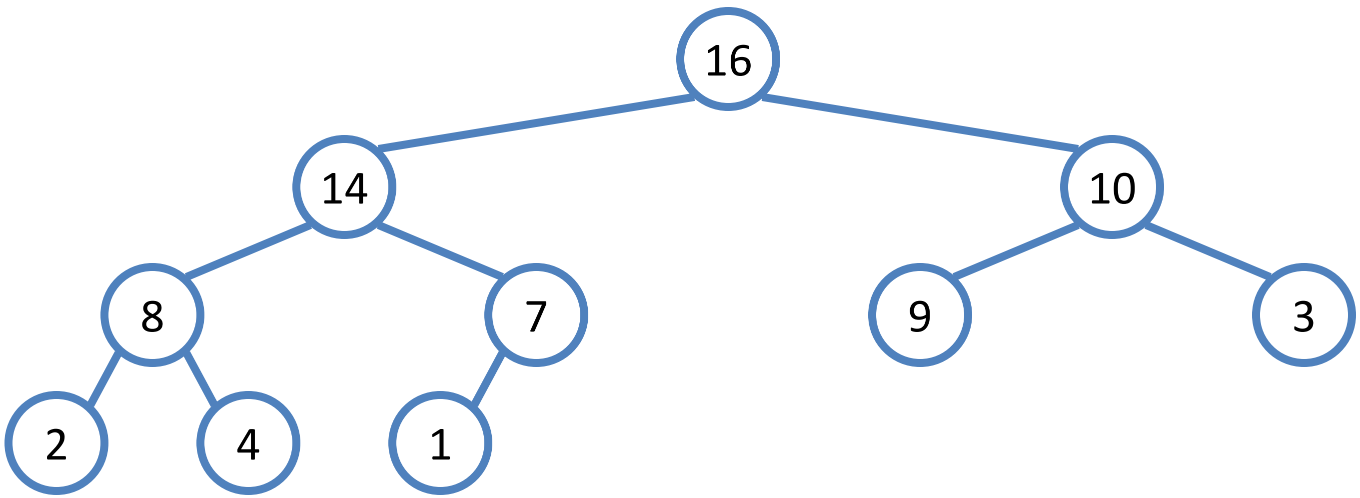
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

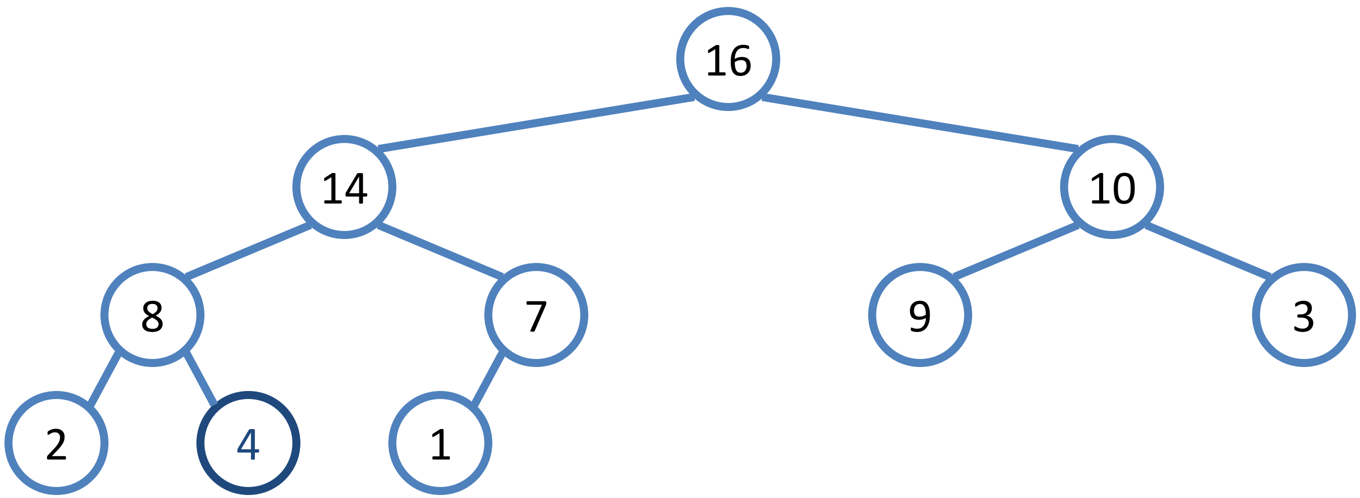
Heapify() Example



A =

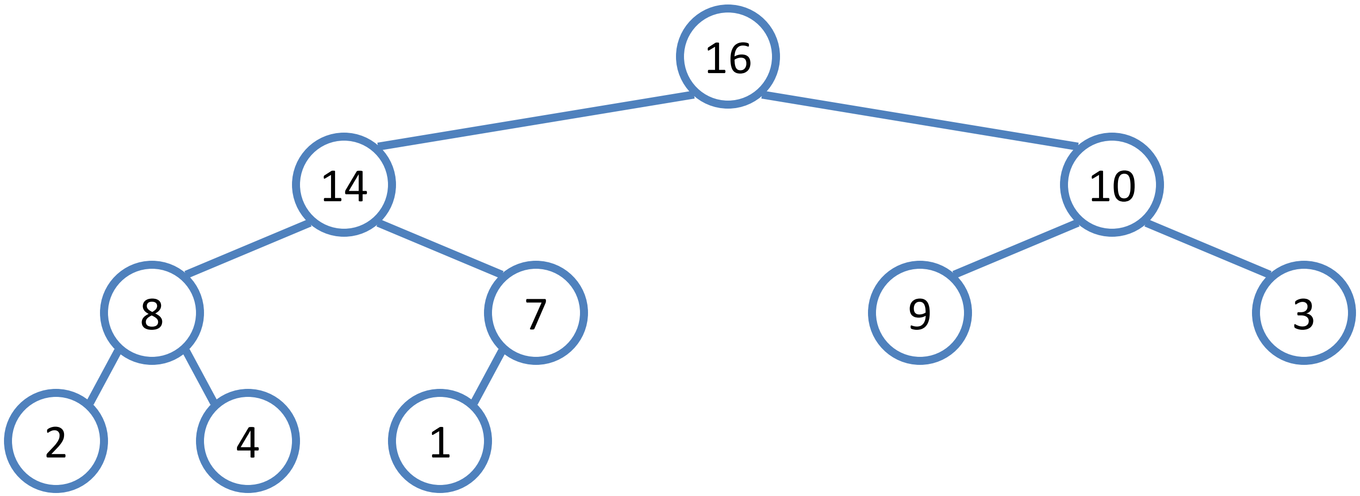
16	14	10	8	7	9	3	2	4	1

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(): Informal

- *Aside from the recursive call, what is the running time of* ***`Heapify()`****?*
  - *Spends O(1) time at any node i. Why?*
- *How many times can* ***`Heapify()`*** *recursively call itself?*
  - *Height = O(log n)*
- *What is the worst-case running time of* ***`Heapify()`*** *on a heap of size n?*
  - *O(log n)*

Analyzing Heapify(): Formal

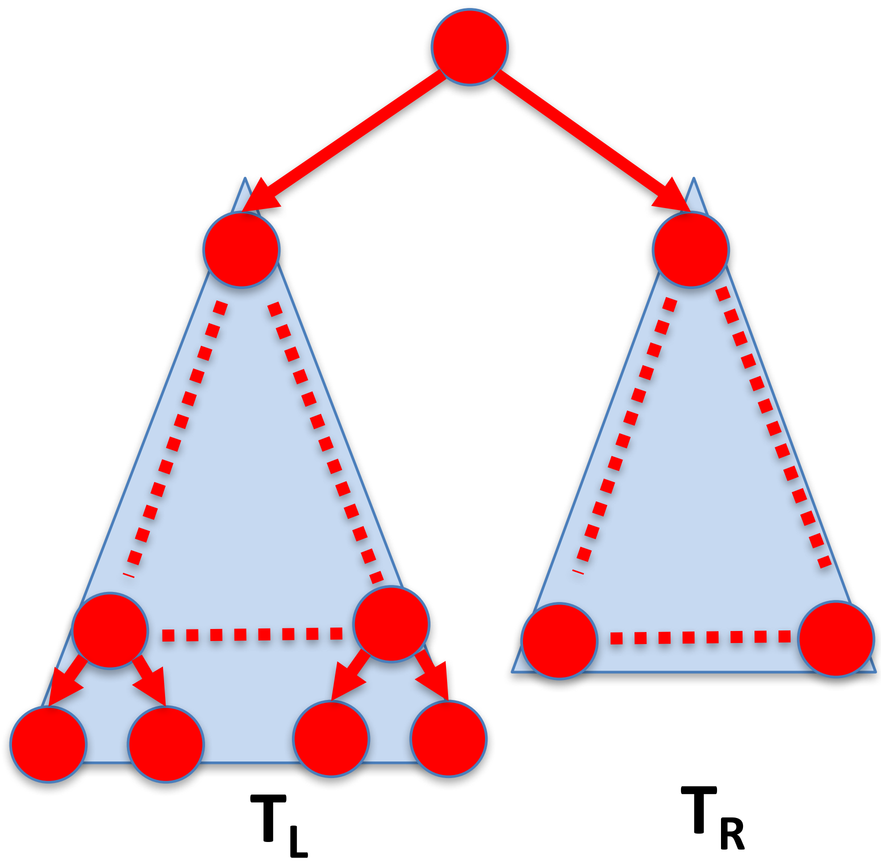
- Recursive algorithm -> need to derive the *recurrence*
- Easy part: Fixing up relationships between *i*, *l*, and *r* takes O(1) time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*

… but before this

- What is this a recipe for?
  - O(# Emmy awards) = O(# Major dead characters)

Bound the largest of two subtrees in terms of total nodes n

- T_L and T_R are "full" in all levels except for last
- Last level – left to right
- Largest possible fraction of nodes T_L?
  - All lowest level in T_L
- $|T_L| = 2^{h-1} - 1 + 2^{h-1} = 2^h - 1$
- $|T_R| = 2^{h-1} - 1$
- So $n = |T_L| + |T_R| + 1 =$
  $= 3 * 2^{h-1} - 1$
  - $2/3\ n = 2^h - 1$ as big as T_L can get



Analyzing Heapify(): Formal

- Recursive algorithm -> need to derive the *recurrence*
- Easy part: Fixing up relationships between *i*, *l*, and *r* takes O(1) time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
- **2n/3** (worst case: bottom row 1/2 full)
- So time taken by **`Heapify()`** is given by the *recurrence:*

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

Analyzing Heapify(): Formal

- So we have

$$
\begin{aligned}
T(n) &\le T((2/3)*n) + O(1) \\
&\le T((2/3)*(2/3)*n) + O(1) + O(1) \\
&= T((2/3)^2 n) + 2O(1) \\
&\dots \\
&\le T((2/3)^r n) + rO(1)
\end{aligned}
$$

- The recursion ends when $(2/3)^r n = 1$
  - Or $r = \log_{2/3} 1/n = \log_{3/2} n = \log_2(3/2) \log_2 n$.
  - Side note: *base of log does not matter for growth rate as long as it is O(1)*
- Thus, **`Heapify()`** takes logarithmic time:
  - $T(n) \le T(1) + c_1 \log_2(3/2) \log_2 n = c_2 + c_1' \log_2 n = O(\log n)$

Announcements

- Read through Chapter 6
  - Next class: Build heap, Heap Sort, Priority Queues
- HW2 available on BB after class