CSE 373: Data Structures and Algorithms

More Asymptotic Analysis; More Heaps

Riley Porter

Winter 2017

Course Logistics

- HW 1 posted. Due next Tuesday, January 17^{th} at 11 pm. Dropbox not on catalyst, will be through the Canvas for the course.

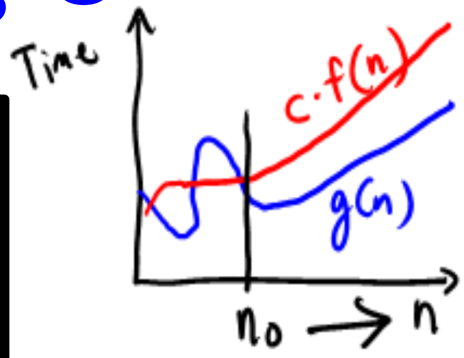
- TA office hour rooms and times are all posted and finalized. Please go visit the TAs so they aren't lonely.

- Java Review Session materials from yesterday posted in the Announcements section of the website.

Review from last time (what did we learn?)

- Analyze algorithms without specific implementations through space and time (what we focused on).
- We only care about asymptotic runtimes, we want to know what will happen to the runtime proportionally as the size of input increases
- Big-O is an upper bound and you can prove that a runtime has a Big-O upper bound by computing two values: c and n_0

Review: Formally Big-O



Definition:

g(n) is in O(f(n)) if there exist constants c and n_0 such that g(n) $\leq$ c f(n) for all $n \geq n_0$

- To show g(n) is in O(f(n)), pick a c large enough to "cover the constant factors" and n_0 large enough to "cover the lower-order terms"
  - Example: Let g(n) = $3n^2$+17 and f(n) = n^2
    c=5 and n_0=10 is more than good enough

- This is "less than or equal to"
  - So $3n^2$+17 is also $O(n^5)$ and $O(2^n)$ etc.

Big-O: Common Names

$O(1)$	constant (same as $O(k)$ for constant k)
$O(\mathbf{log}\ n)$	logarithmic
$O(n)$	linear
$\mathrm{O}(\mathrm{n}\ \mathbf{log}\ n)$	"n **log** n"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is any constant: linear, quadratic and cubic all fit here too.)
$O(k^n)$	exponential (where k is any constant > 1)

Note: "exponential" does not mean "grows really fast", it means "grows at rate proportional to k^n for some k>1". Example: a savings account accrues interest exponentially (k=1.01?).

More Asymptotic Notation

- Big-O Upper bound: $O(\, f(n)\,)$ is the set of all functions asymptotically less than or equal to $f(n)$
  - $g(n)$ is in $O(\, f(n)\,)$ if there exist constants c and n_0 such that
  $g(n) \leq c\, f(n)$ for all $n \geq n_0$

- Big-Omega Lower bound: $\Omega(\, f(n)\,)$ is the set of all functions asymptotically greater than or equal to $f(n)$
  - $g(n)$ is in $\Omega(\, f(n)\,)$ if there exist constants c and n_0 such that
  $g(n) \geq c\, f(n)$ for all $n \geq n_0$

- Big-Theta Tight bound: $\theta(\, f(n)\,)$ is the set of all functions asymptotically equal to $f(n)$
  - Intersection of $O(\, f(n)\,)$ and $\Omega(\, f(n)\,)$ (use *different* c values)

A Note on Big-O Terms

- **A common error is to say O(function) when you mean θ(function):**
  - People often say Big-O to mean a tight bound
  - Say we have f(n)=n; we could say f(n) is in O(n), which is true, but only conveys the upper-bound
  - Since f(n)=n is also $O(n^5)$, it's tempting to say "this algorithm is exactly O(n)"
  - Somewhat incomplete; instead say it is $\theta(n)$
  - That means that it is not, for example O(log n)

What We're Analyzing

- The most common thing to do is give an O or θ bound to the worst-case running time of an algorithm
- Example: True statements about binary-search algorithm
  - Common: $\theta(\log n)$ running-time in the worst-case
  - Less common: $\theta(1)$ in the best-case (item is in the middle)
  - Less common (but very good to know): the find-in-sorted array problem is $\Omega(\log n)$ in the worst-case
    - No algorithm can do better (without parallelism)

Intuition / Math on O(logN)

- If you're dividing your input in half (or any other constant) each iteration of an algorithm, that's O(logN).
- Binary Search Example:

If you divide your input in half each time and discard half the values, to figure out the worst-case runtime you need to figure out how many "halves" you have in your input. So you're solving:

$$N \,/\, 2^x = 1$$

where N is size of input, X is "number of halves", because 1 is the desired number of elements you're trying to get to.

$$\log(2^x) = X*\log(2) = \log(N)$$

$$X = \log(N) \,/\, \log(2)$$

$$X = \log_2(N)$$

Other things to analyze

- Remember we can often use space to gain time
- Average case
  - Sometimes only if you assume something about the *probability distribution* of inputs
  - Usually the way we think about Hashing
    - Will discuss in two weeks
  - Sometimes uses randomization in the algorithm
    - Will see an example with sorting
  - Sometimes an *amortized guarantee*
    - Average time over any sequence of operations
    - Will discuss next week

Usually asymptotic is valuable

- Asymptotic complexity focuses on behavior for large *n* and is independent of any computer / coding trick
- But you can "abuse" it to be misled about trade-offs
- Example: $n^{1/10}$ vs. `log` *n*
  - Asymptotically $n^{1/10}$ grows more quickly
  - But the "cross-over" point is around $5 * 10^{17}$
  - So if you have input size less than 2^{58}, prefer $n^{1/10}$
- For *small n*, an algorithm with worse asymptotic complexity might be faster
  - Here the constant factors can matter, if you care about performance for small *n*

Summary of Asymptotic Analysis

Analysis can be about:

- The problem or the algorithm (usually algorithm)
- Time or space (usually time)
- Best-, worst-, or average-case (usually worst)
- Upper-, lower-, or tight-bound (usually upper)

- The most common thing we will do is give an O upper bound to the worst-case running time of an algorithm.

Let's use our new skills!

Here's a picture of a kitten as a segue to analyzing an ADT

Analysis of Priority Queue ADT

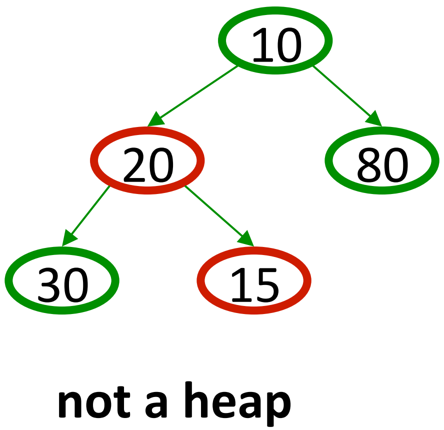
Let's compare some options for implementing Priority Queues. All runtimes worst-case, but assume arrays have room for new elements. We'll look at the binary search tree operations and runtimes more on Friday.

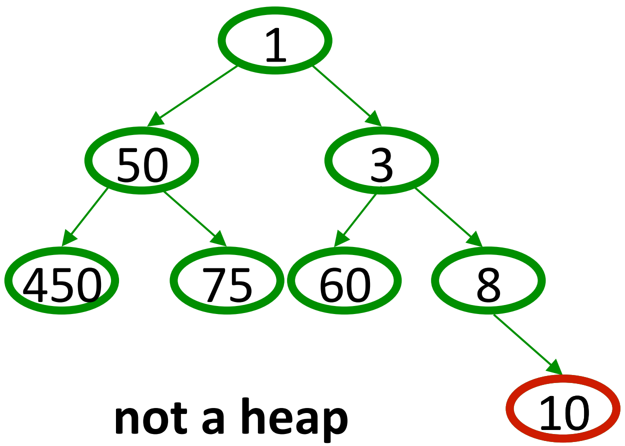
data structure	**insert**		**deleteMin**
unsorted array	add at end	$O(1)$	search $O(n)$
unsorted linked list	add at front	$O(1)$	search $O(n)$
sorted array	search / shift	$O(n)$	stored in reverse $O(1)$
sorted linked list	put in right place $O(n)$		remove at front $O(1)$
binary search tree	put in right place $O(n)$		leftmost $O(n)$
heaps	**???**		**???**

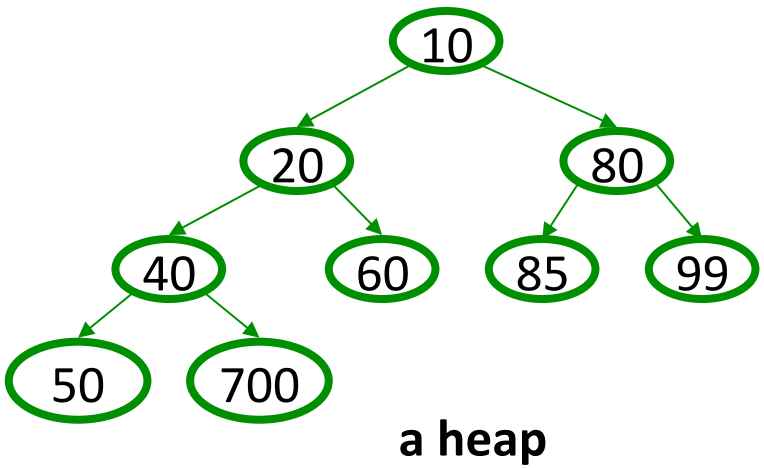
Review of last time: Heaps

Heaps follow the following two properties:

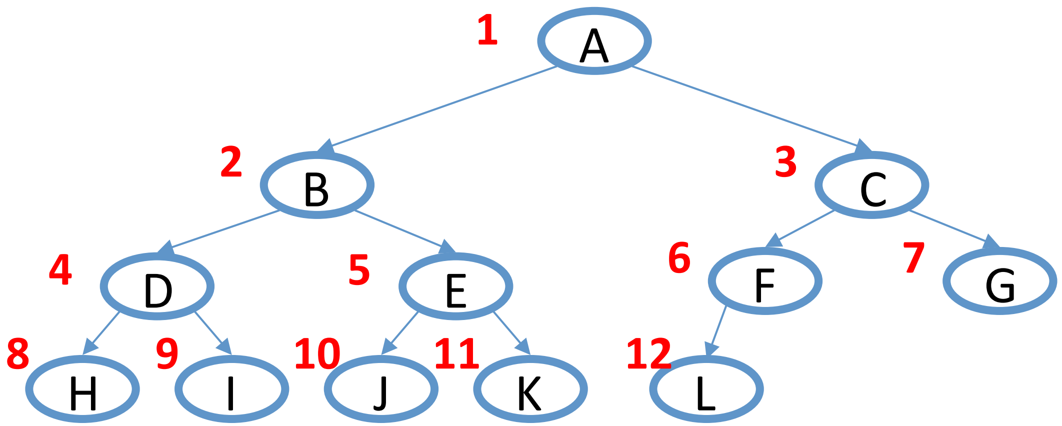
- Structure property: A *complete* binary tree
- Heap order property: The priority of the children is always a greater value than the parents (greater value means less priority / less importance)







Array Representation of Heaps (or any tree structure)



Starting at node `i`

left child: `i*2`
right child: `i*2+1`
parent: `i/2`

(wasting index 0 is convenient for the index arithmetic)

implicit (array) implementation:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13
		parent		i = 4				left	right				

Judging the array implementation

Positives:

- Non-data space is minimized: just index 0 and unused space on right
  - In conventional tree representation, one edge per node (except for root), so *n*-1 wasted space (like linked lists)
  - Array would waste more space if tree were not complete

- Multiplying and dividing by 2 is very fast (shift operations in hardware)
- Last used position is just index **`size`**

Negatives:

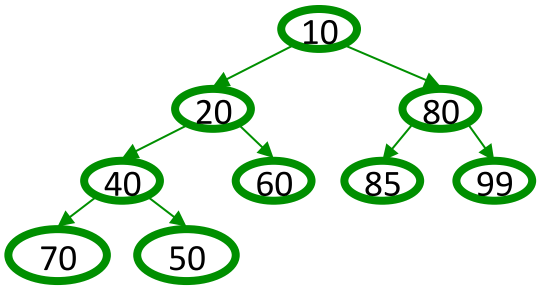
- Same might-by-empty or might-get-full problems we saw with stacks and queues (resize by doubling as necessary)

Plusses outweigh minuses: “this is how people do it”

Pseudocode: insert

```
void insert(int val) {
   if(size == arr.length-1)
     resize();
   size++;
   i=percolateUp(size,val);
   arr[i] = val;
}
```

```
int percolateUp(int hole,int val) {
  while(hole > 1 &&
         val < arr[hole/2])
    arr[hole] = arr[hole/2];
    hole = hole / 2;
  }
  return hole;
}
```

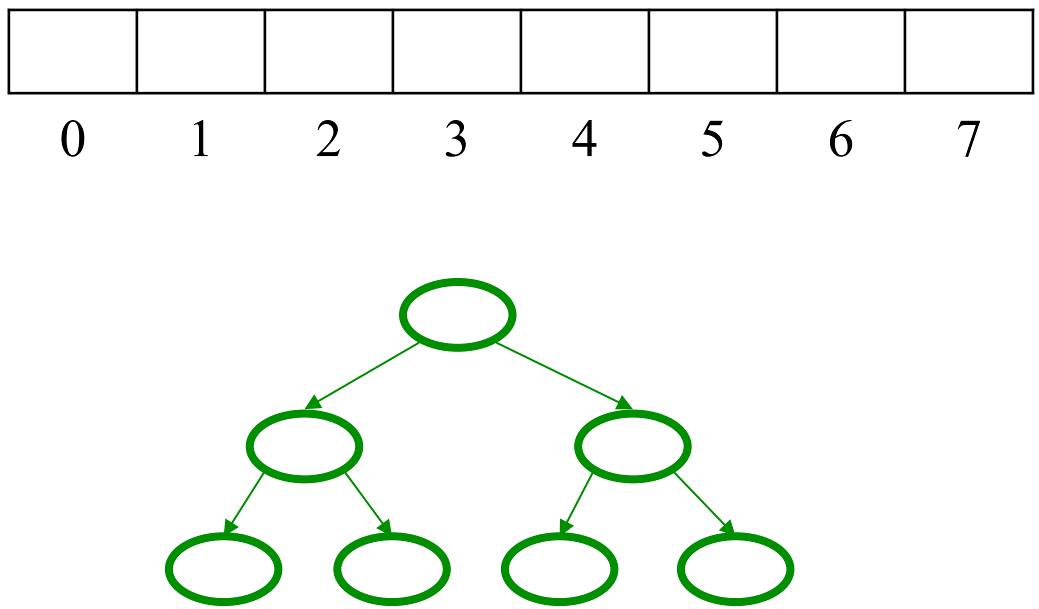


This pseudocode uses ints. Since not all data types are comparable, you could instead have data nodes with priorities.

	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

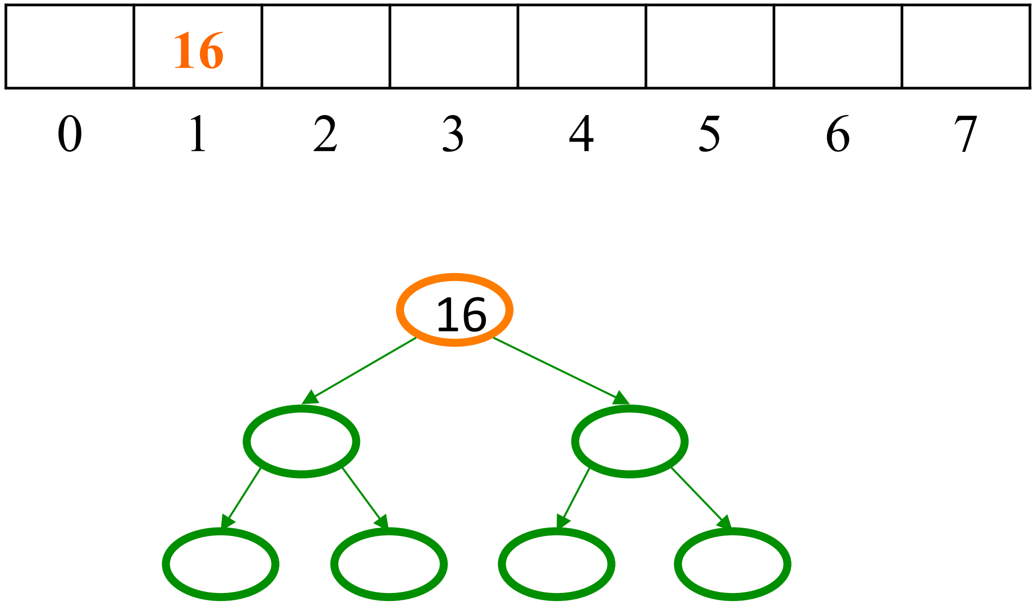
Example

1. insert: 16, 32, 4, 69, 105, 43, 2



Example

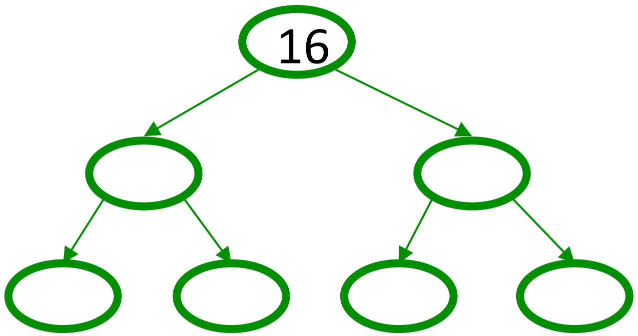
1. insert: 16, 32, 4, 69, 105, 43, 2



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

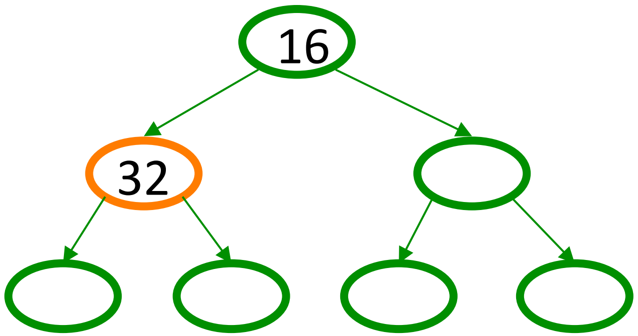
	16						
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

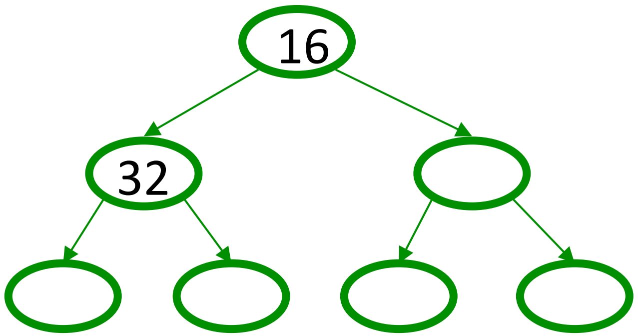
	16	**32**					
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

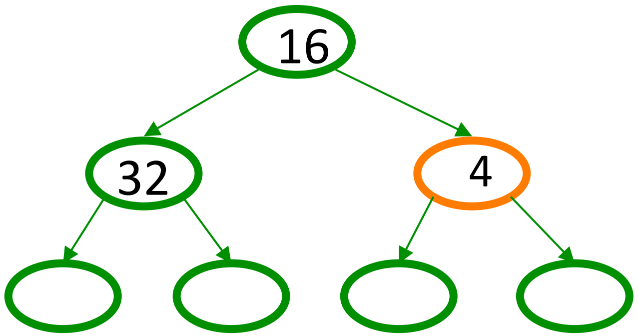
	16	**32**					
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

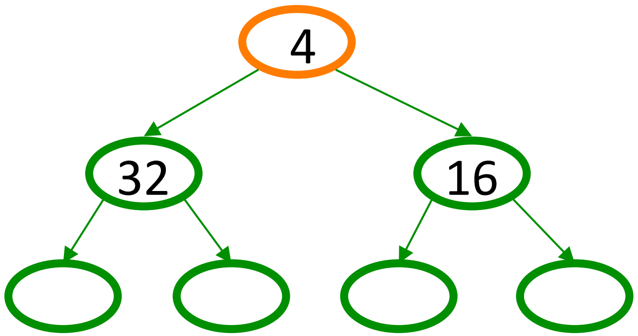
	16	**32**	**4**				
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

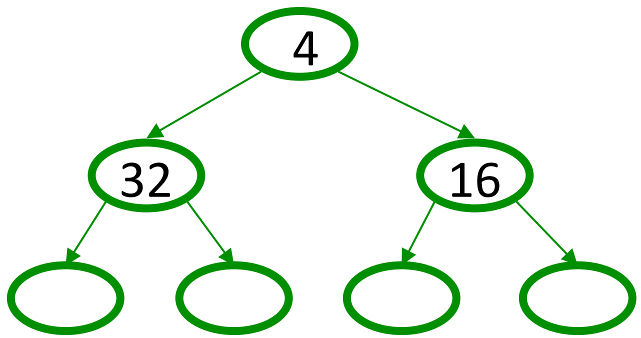
	4	**32**	**16**				
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

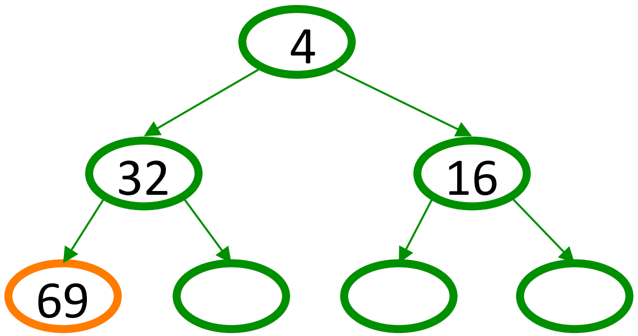
	4	**32**	**16**				
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

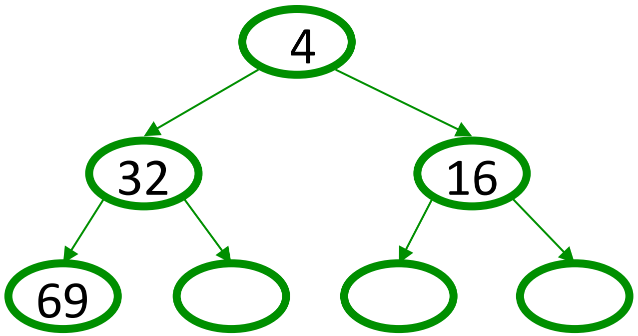
0	1	2	3	4	5	6	7
	4	**32**	**16**	**69**			



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

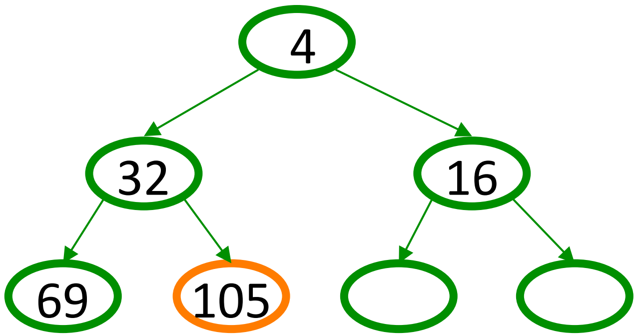
	4	**32**	**16**	**69**			
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

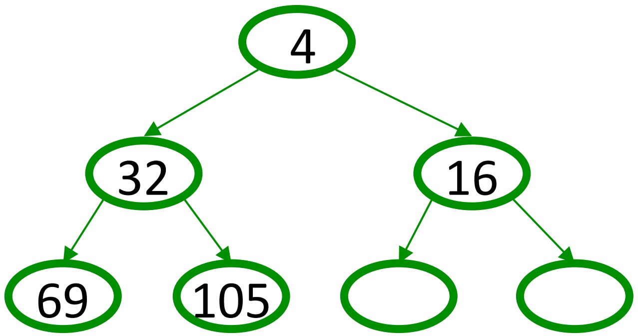
	4	**32**	**16**	**69**	**105**		
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

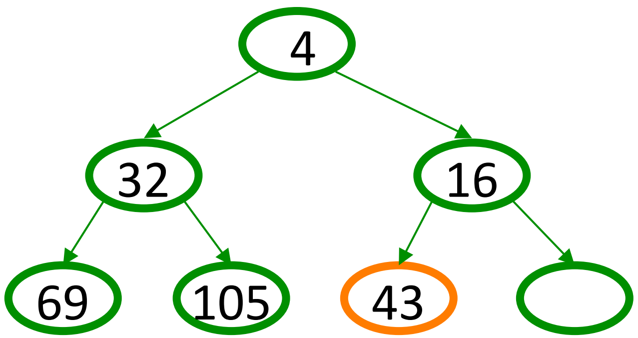
	4	**32**	**16**	**69**	**105**		
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

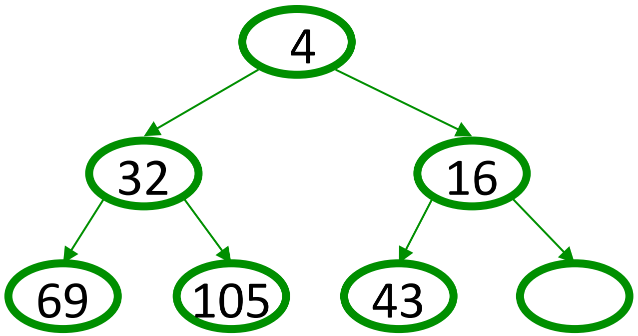
	4	32	16	69	105	43	
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

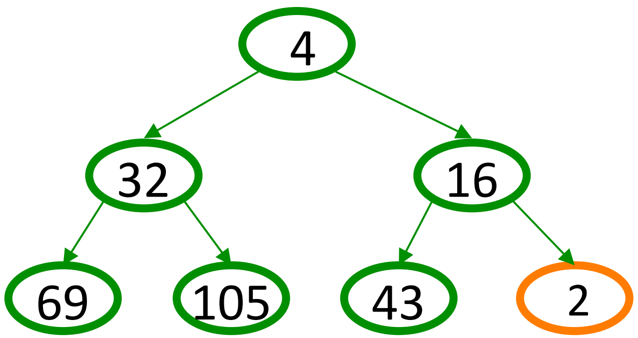
	4	**32**	**16**	**69**	**105**	**43**	
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

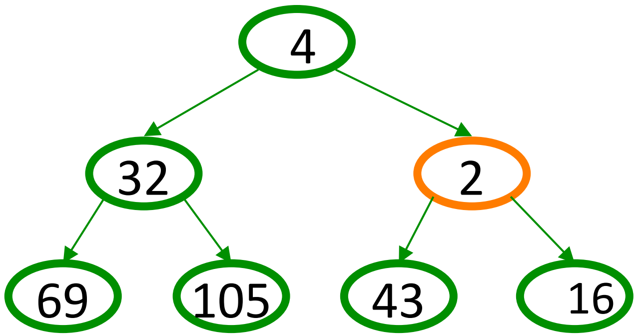
	4	**32**	**16**	**69**	**105**	**43**	**2**
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

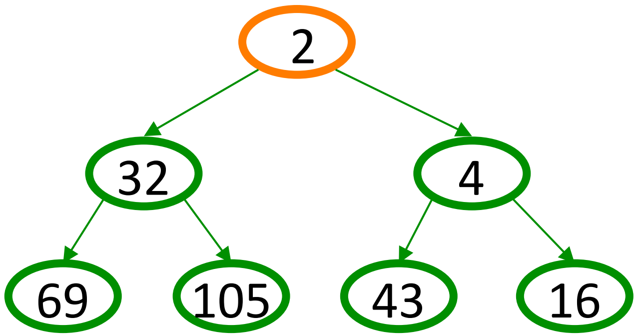
	4	**32**	**2**	**69**	**105**	**43**	**16**
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

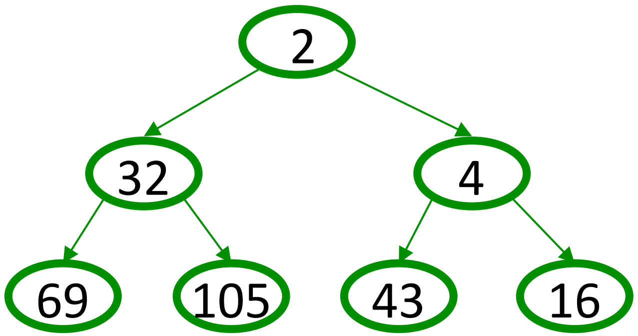
	2	32	4	69	105	43	16
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 69, 105, 43, 2

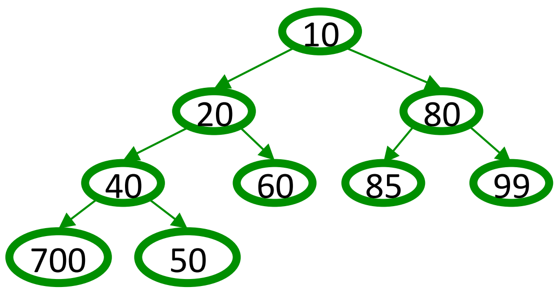
	2	**32**	**4**	**69**	**105**	**43**	**16**
0	1	2	3	4	5	6	7



Pseudocode: deleteMin

```
int deleteMin() {
  if(isEmpty()) throw…
  ans = arr[1];
  hole = percolateDown
          (1,arr[size]);
  arr[hole] = arr[size];
  size--;
  return ans;
}
```

```
int percolateDown(int hole,int val){
 while(2*hole <= size) {
   left  = 2*hole;
   right = left + 1;
   if(arr[left] < arr[right]
      || right > size)
     target = left;
   else
     target = right;
   if(arr[target] < val) {
     arr[hole] = arr[target];
     hole = target;
   } else
     break;
 }
 return hole;
}
```

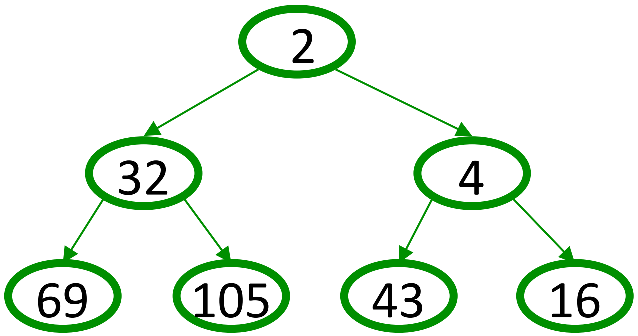


	10	**20**	**80**	**40**	**60**	**85**	**99**	**700**	**50**				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Example

1. deleteMin

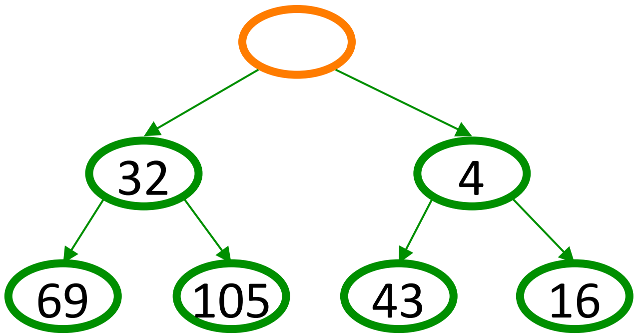
	2	**32**	**4**	**69**	**105**	**43**	**16**
0	1	2	3	4	5	6	7



Example

1. deleteMin

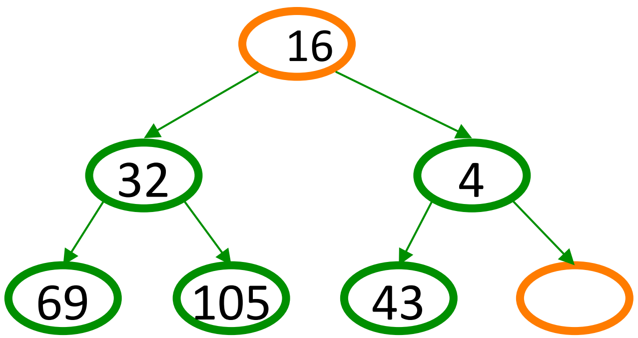
		32	**4**	**69**	**105**	**43**	**16**
0	1	2	3	4	5	6	7



Example

1. deleteMin

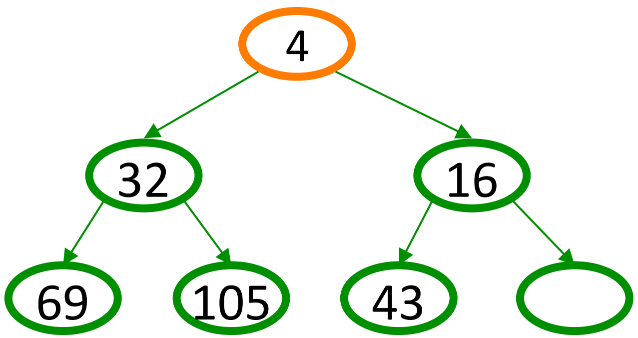
0	1	2	3	4	5	6	7
	16	32	4	69	105	43	



Example

1. deleteMin

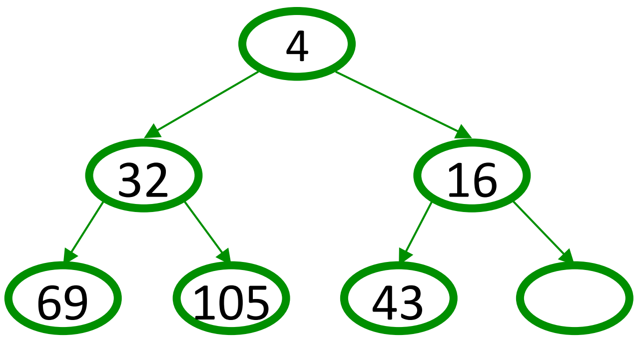
	4	32	16	69	105	43	
0	1	2	3	4	5	6	7



Example

1. deleteMin

	4	**32**	**16**	**69**	**105**	**43**	
0	1	2	3	4	5	6	7



DeleteMin: Run Time Analysis

- We will percolate down at most (height of heap) times
  - So run time is *O*(height of heap)

- A heap is a complete binary tree

- Height of a complete binary tree of *n* nodes?
  - height = $\lfloor \texttt{log}_2(n) \rfloor$

- Run time of **`deleteMin`** is *O*(**`log`** *n*)

Insert: Run Time Analysis

- Same as **`deleteMin`** worst-case time proportional to tree height *O*(**`log`** *n*)
- **`deleteMin`** needs the "last used" complete-tree position and **`insert`** needs the "next to use" complete-tree position
  - If "keep a reference to there" then **`insert`** and **`deleteMin`** have to adjust that reference: *O*(**`log`** *n*) in worst case
  - Could calculate how to find it in *O*(**`log`** *n*) from the root given the size of the heap

Other operations

- **`decreaseKey`**: given pointer to object in priority queue (e.g., its array index), lower its priority value. *Remember **lower priority value** is *better** (higher in tree).
  - Change priority and percolate up
- **`increaseKey`**: given pointer to object in priority queue (e.g., its array index), raise its priority value.
  - Change priority and percolate down
- **`remove`**: given pointer to object in priority queue (e.g., its array index), remove it from the queue.
  - Percolate up to top and removeMin
- **`buildHeap:`** given a list of elements, construct a heap with those values.
  - Floyd's Method will be seen on Friday

Revisit: Analysis of Priority Queue ADT

Let's compare some options for implementing Priority Queues. All runtimes worst-case, but assume arrays have room for new elements. We'll look at the binary search tree operations and runtimes more on Friday.

data structure	**insert**		**deleteMin**
unsorted array	add at end	$O(1)$	search $O(n)$
unsorted linked list	add at front	$O(1)$	search $O(n)$
sorted array	search / shift	$O(n)$	stored in reverse $O(1)$
sorted linked list	put in right place $O(n)$		remove at front $O(1)$
binary search tree	put in right place $O(n)$		leftmost $O(n)$
heaps	**O(logn)**		**O(logn)**

Today's Takeaways

- Understand Big-O, Big-theta, and Big-Omega definitions and how to find them for a given runtime.
- Understand how Heap operations are implemented with the array representation and be able to analyze their runtimes.