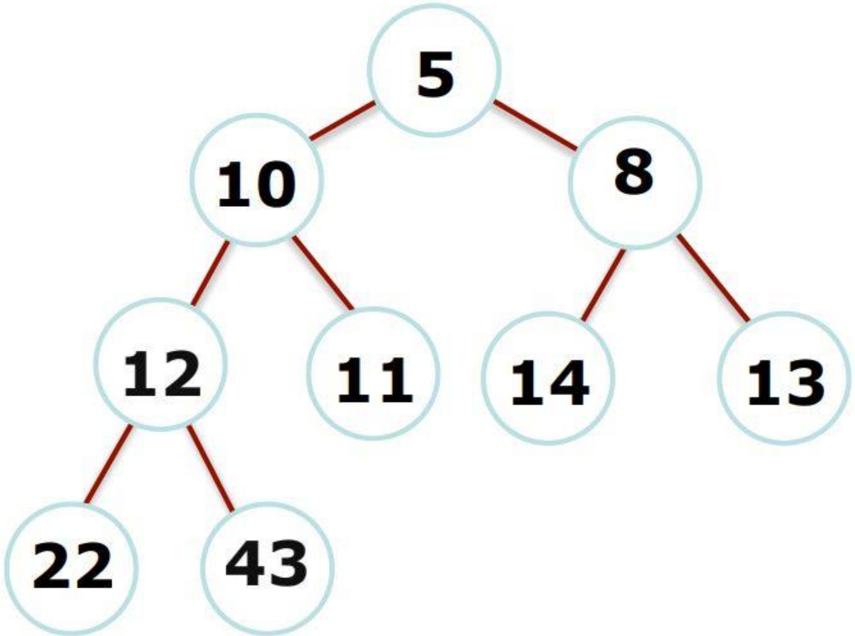
HEAP

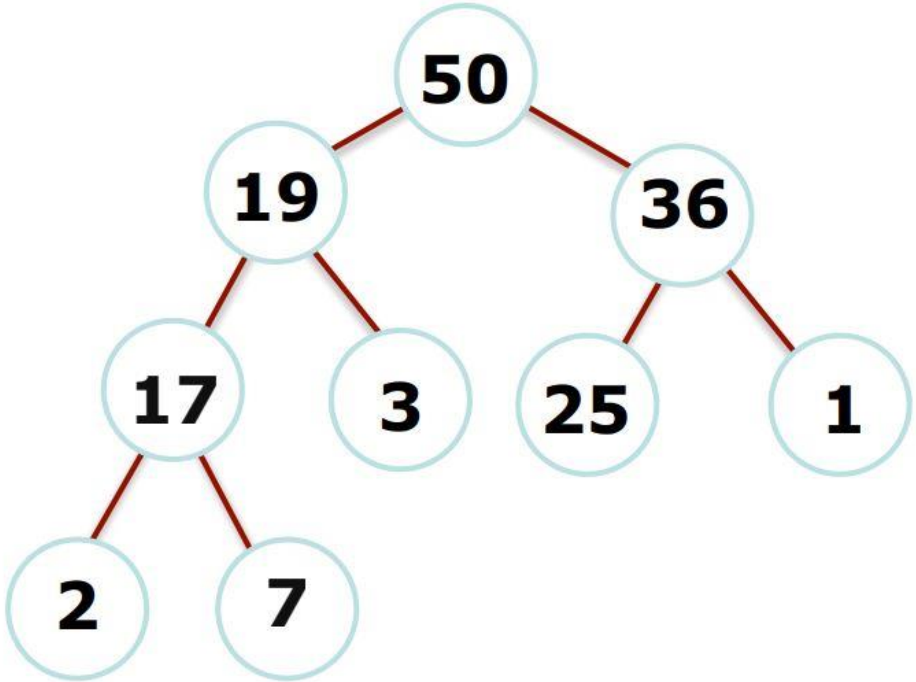
Binary Heap

- There are two types of heaps:

Min Heap
(root is the smallest element)



Max Heap
(root is the largest element)



Binary Heap

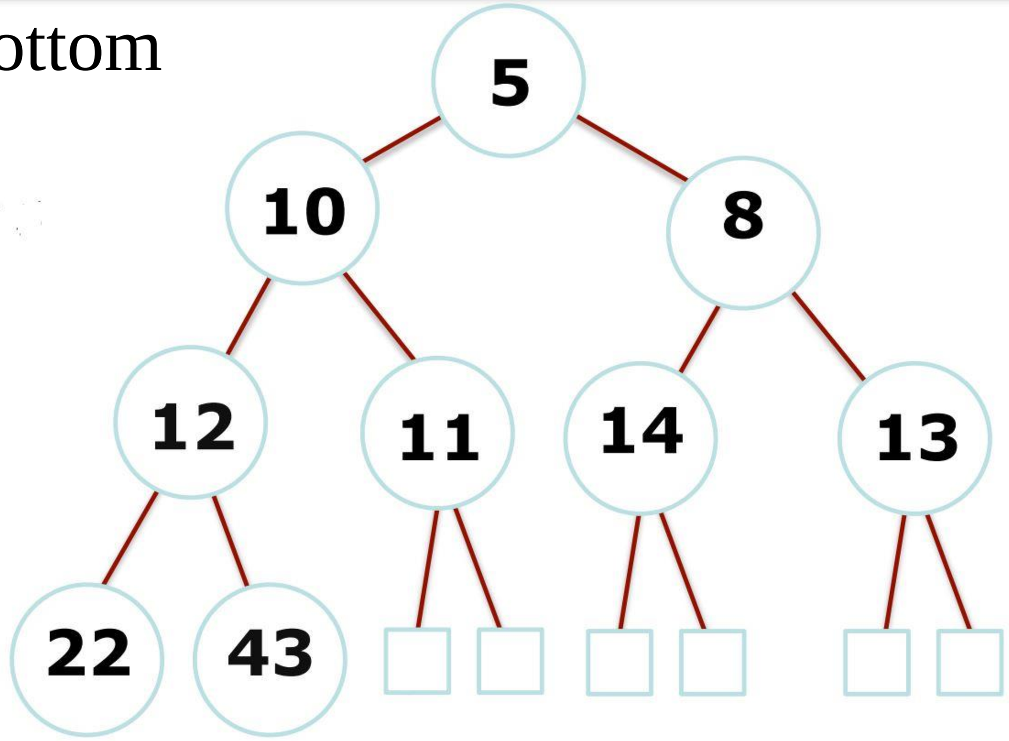
- There are no implied orderings between siblings, so both of the trees below are min-heaps:

Binary Heap

Heaps are **completely filled**, with the exception of the bottom level.
They are, therefore, "complete binary trees":

complete: all levels filled except the bottom

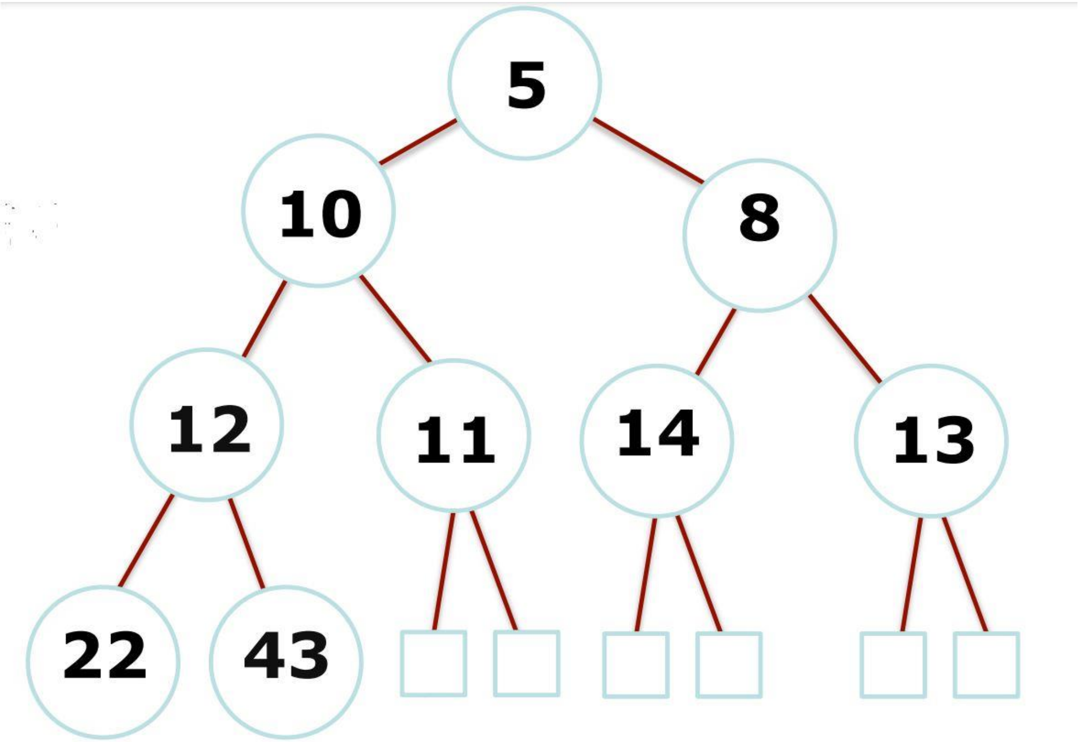
binary: two children per node (parent)



- Maximum number of nodes
- Filled from left to right

Binary Heap

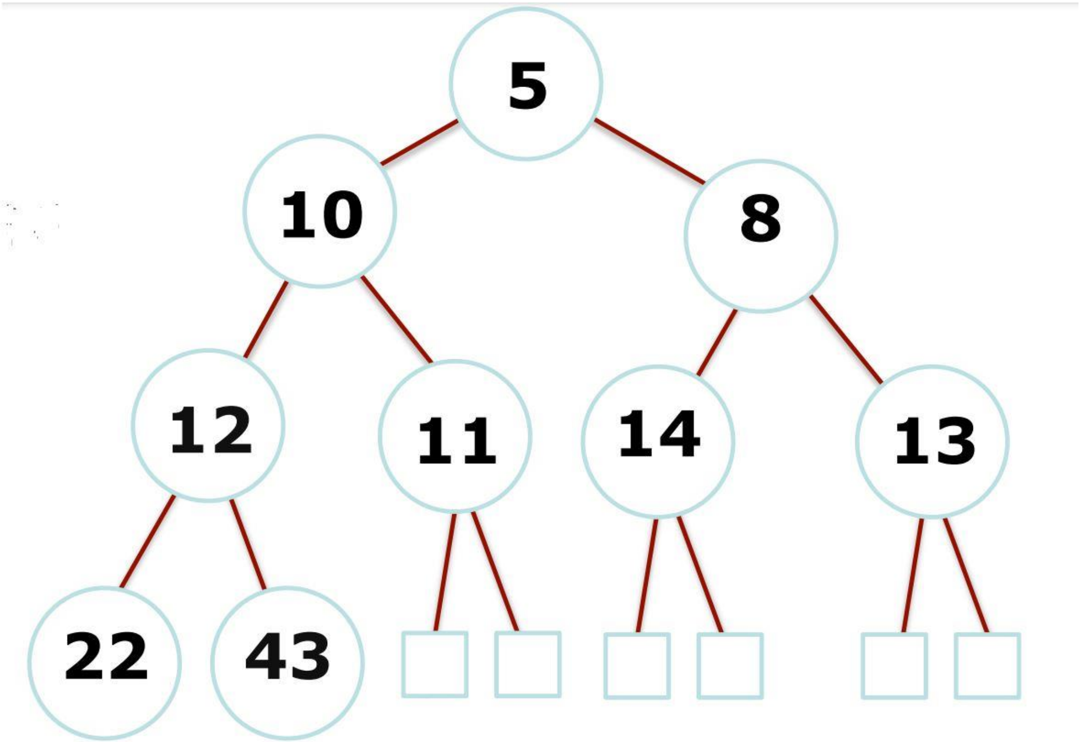
What is the best way to store a heap?



Binary Heap

ARRAY!!!

	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

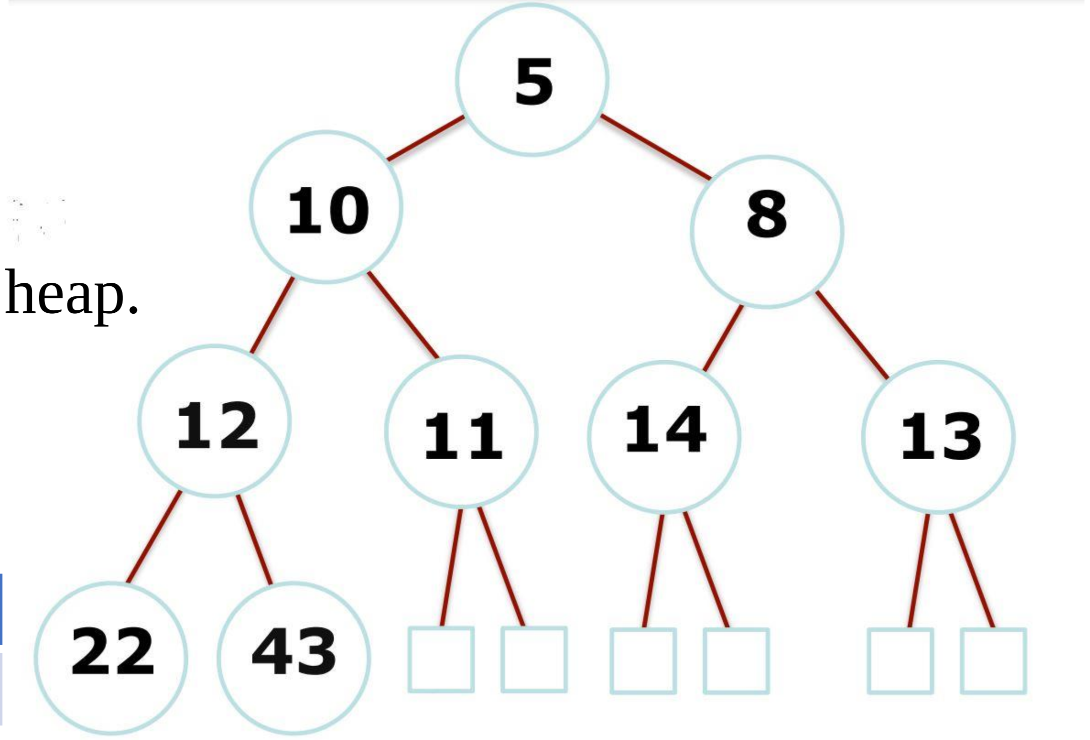


Binary Heap

The array representation makes determining parents and children a matter of simple arithmetic: •For an element at position i:

- •left child is at 2i
- •right child is at 2i+1
- •parent is at $\lfloor i/2 \rfloor$
- •heapSize: the number of elements in the heap.

	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

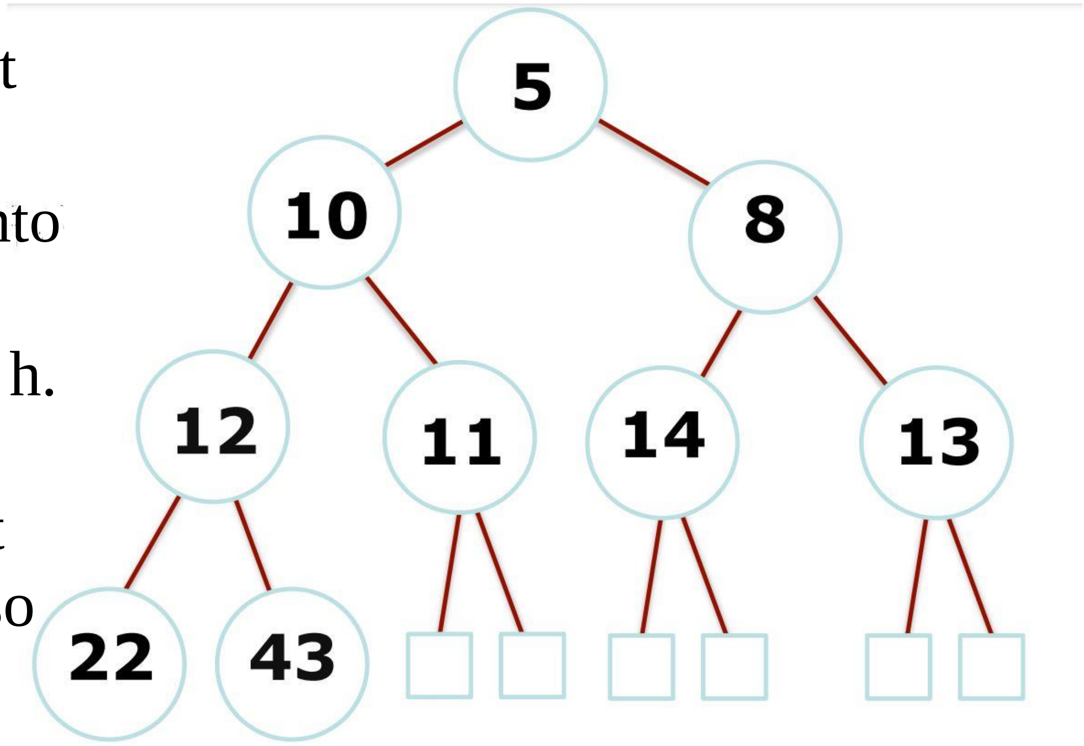


Heap Operation

Remember that there are three important priority queue operations:

1.**peek():** return an element of h with the smallest key.
2.**enqueue(k,e)**: insert an element e with key k into the heap.
3.**dequeue()**: removes the smallest element from h.

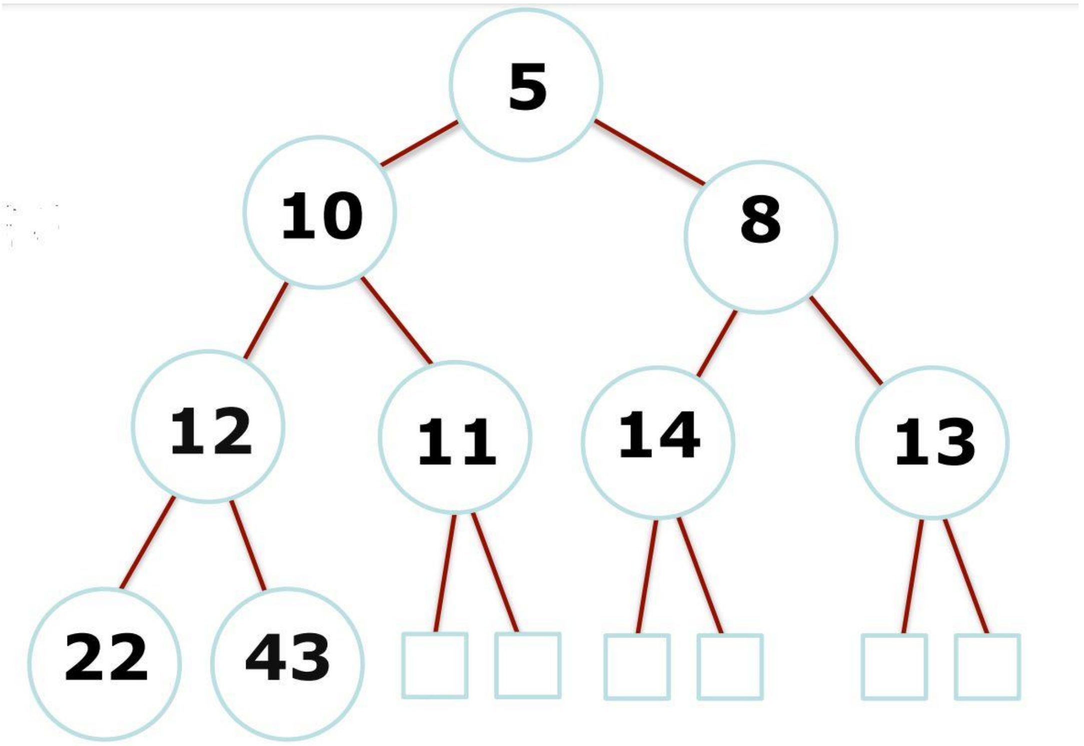
We can accomplish this with a heap! We will just look at keys for now -- just know that we will also store a value with the key



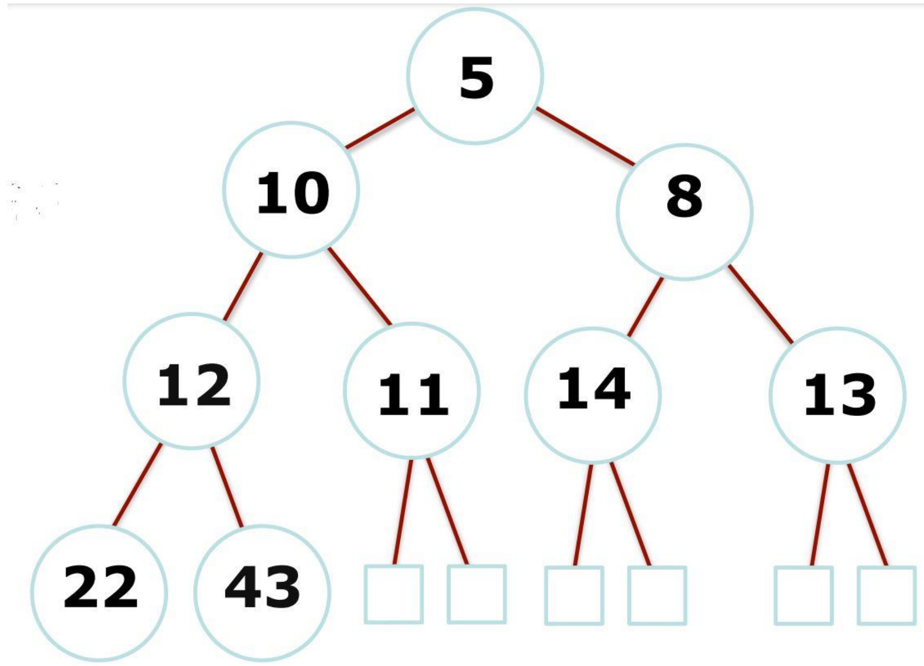
Heap Operation

See animation at:

https://www.cs.usfca.edu/~galles/visualization/Heap.html



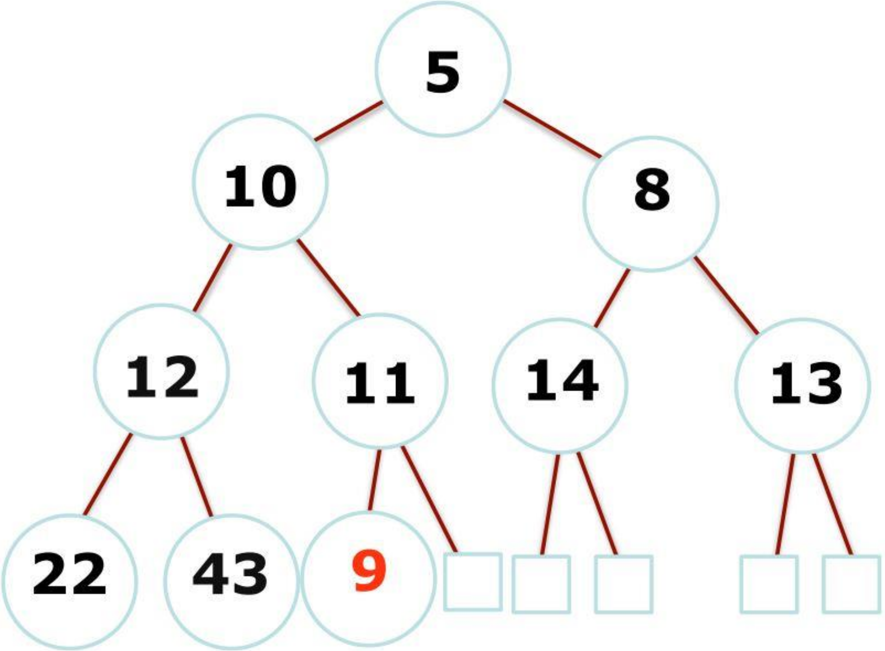
Heap Operation: enqueue(9)



	5	10	8	12	11	14	13	22	43	9	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Start by inserting the key at the first empty position. 9 This is always at index **heap.size()+1**.

Heap Operation: enqueue(9)

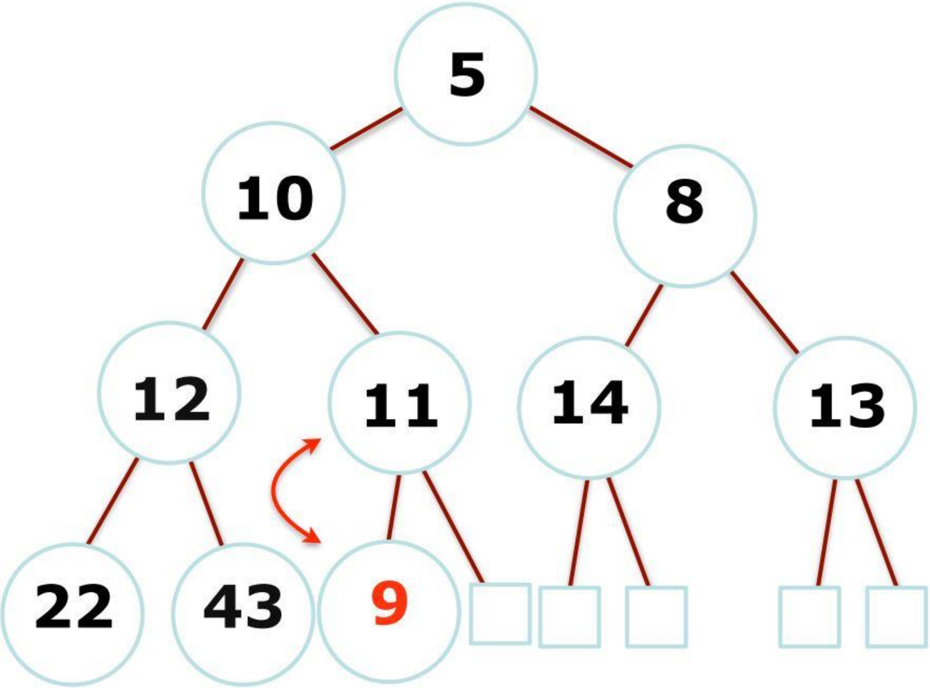


	5	10	8	12	11	14	13	22	43	9	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Look at parent of index 10, and compare: do we meet the heap property requirement?

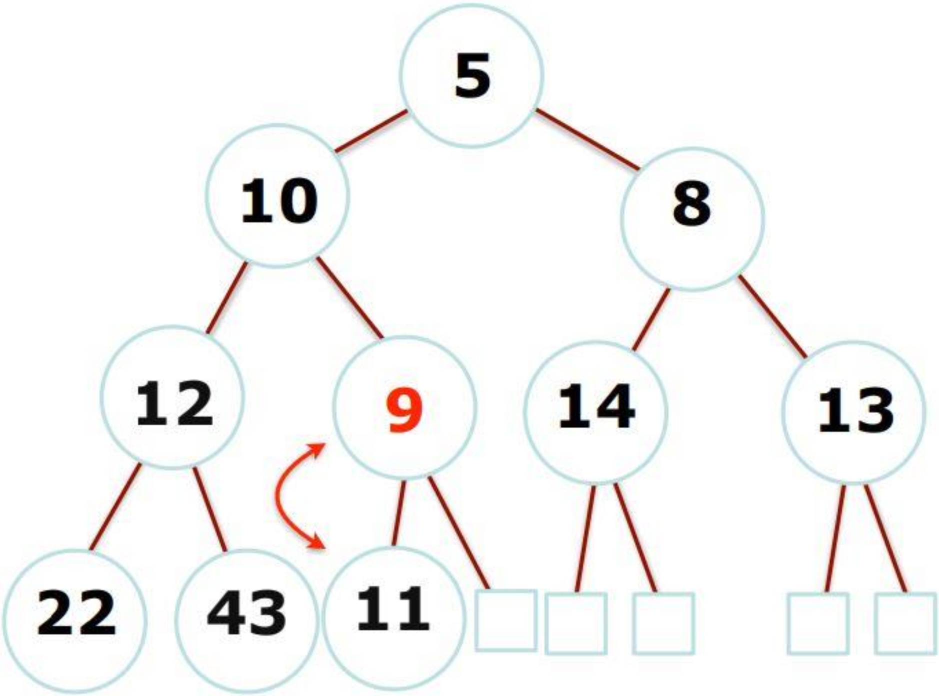
No -- we must swap.

Heap Operation: enqueue(9)



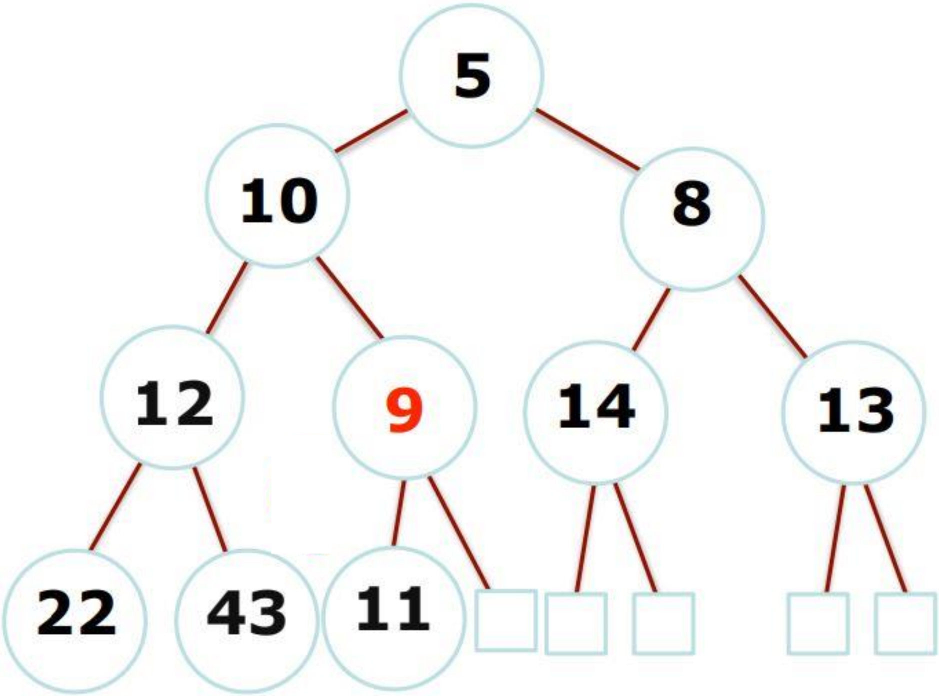
	5	10	8	12	11	14	13	22	43	9	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operation: enqueue(9)



	5	10	8	12	9	14	13	22	43	11	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operation: enqueue(9)

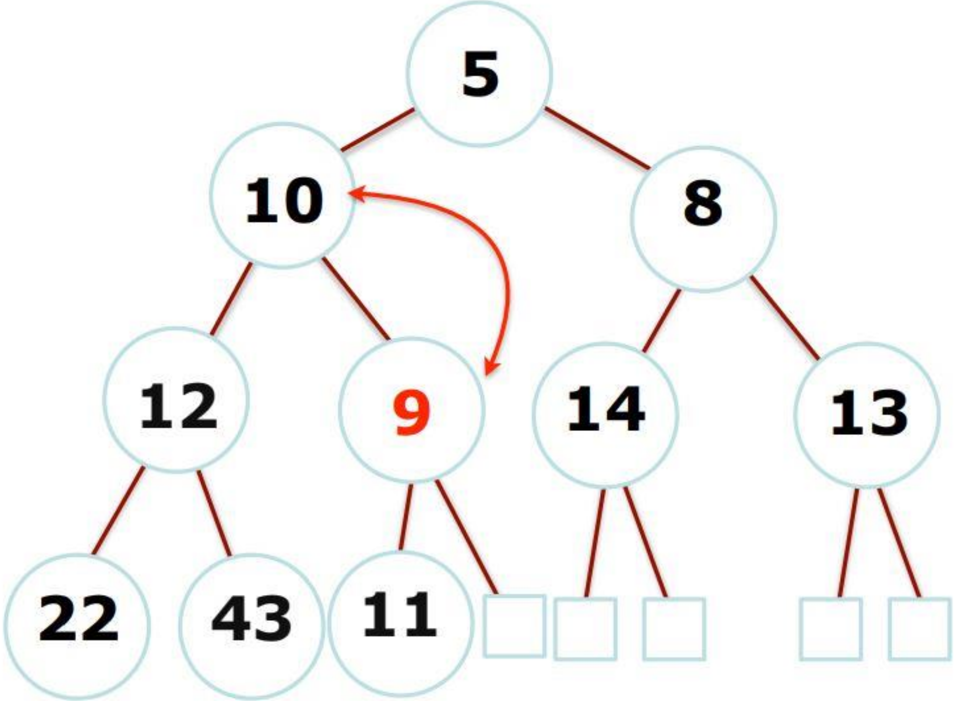


	5	10	8	12	9	14	13	22	43	11	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Look at parent of index 5, and compare: do we meet the heap property requirement?

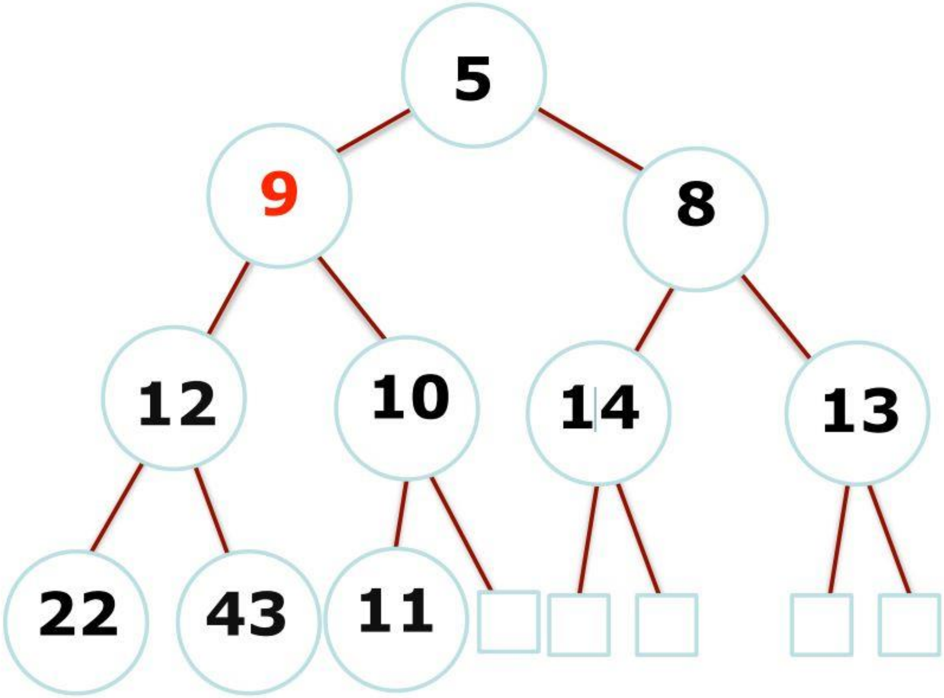
No -- we must swap. This "bubbling up" won't ever be a problem if the heap is "already a heap" (i.e., already meets heap property for all nodes)

Heap Operation: enqueue(9)



	5	9	8	12	10	14	13	22	43	11	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operation: enqueue(9)



No swap necessary between index 2 and its parent. We're done bubbling up!

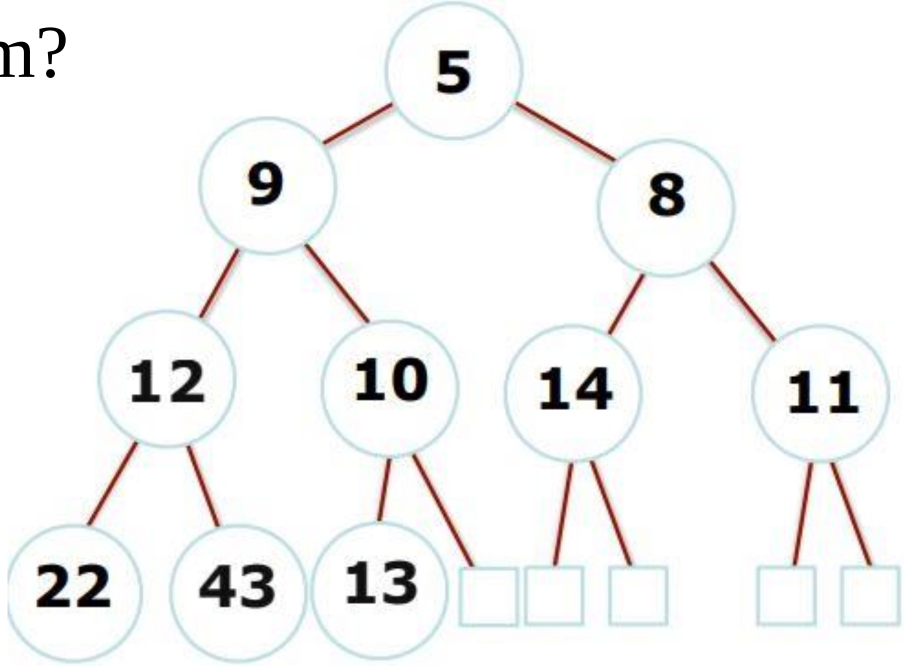
	5	9	8	12	10	14	13	22	43	11	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Complexity? O(log n) - yay!
Average complexity for random inserts: O(1),
see:
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6312854

Heap Operation: dequeue()

- How might we go about removing the minimum?

dequeue()



	5	9	8	12	10	14	11	22	43	13	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operation: dequeue()

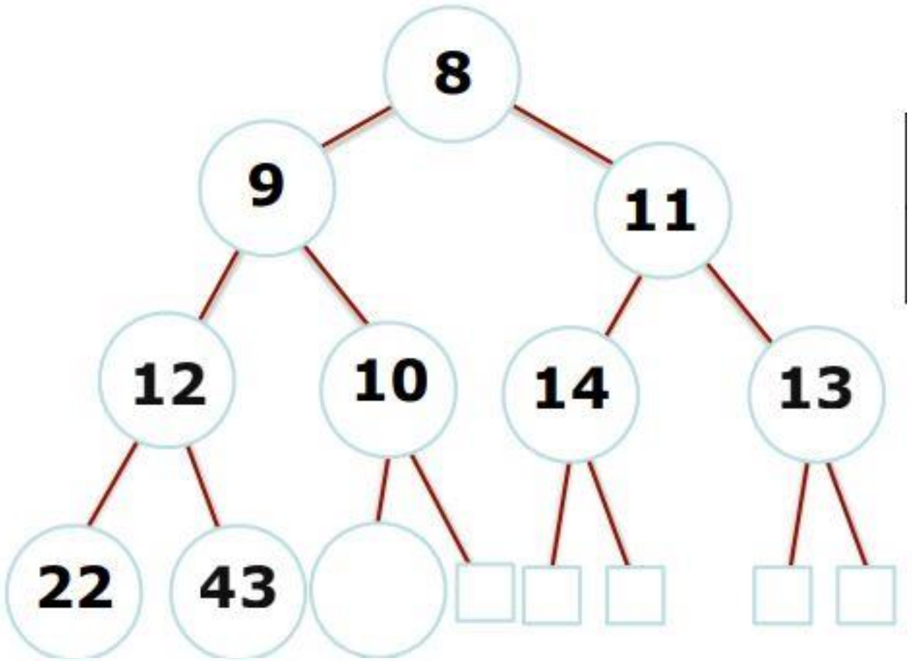
How?!!

Explain in the class

Heap Operation: dequeue()

- How might we go about removing the minimu

dequeue()

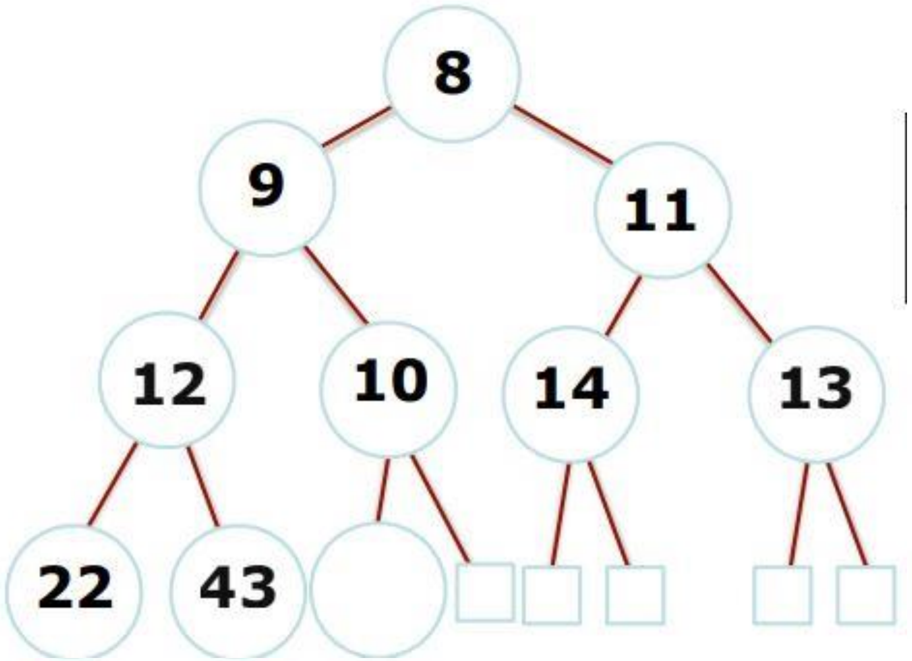


	8	9	11	12	10	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operation: dequeue()

- How might we go about removing the minimu

  dequeue()



	8	9	11	12	10	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Complexity? O(log n) - yay