Lecture 24

Binary Heap

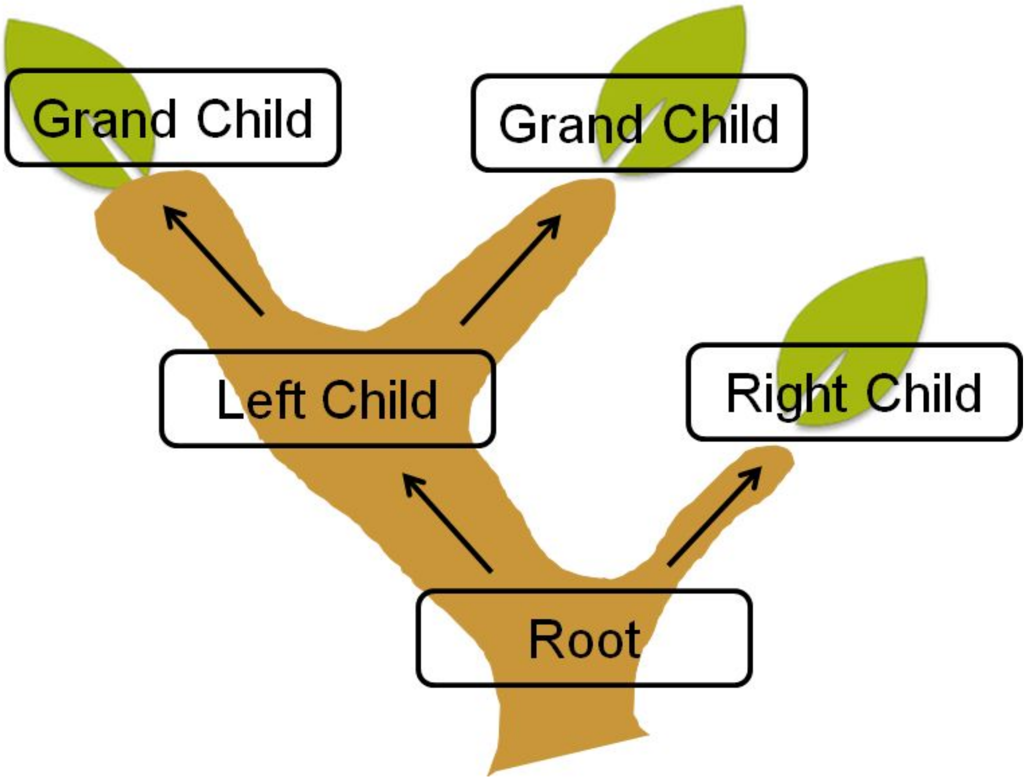
November 08, 2021
Monday

A Priority Queue can easily be implemented using a Binary Heap.

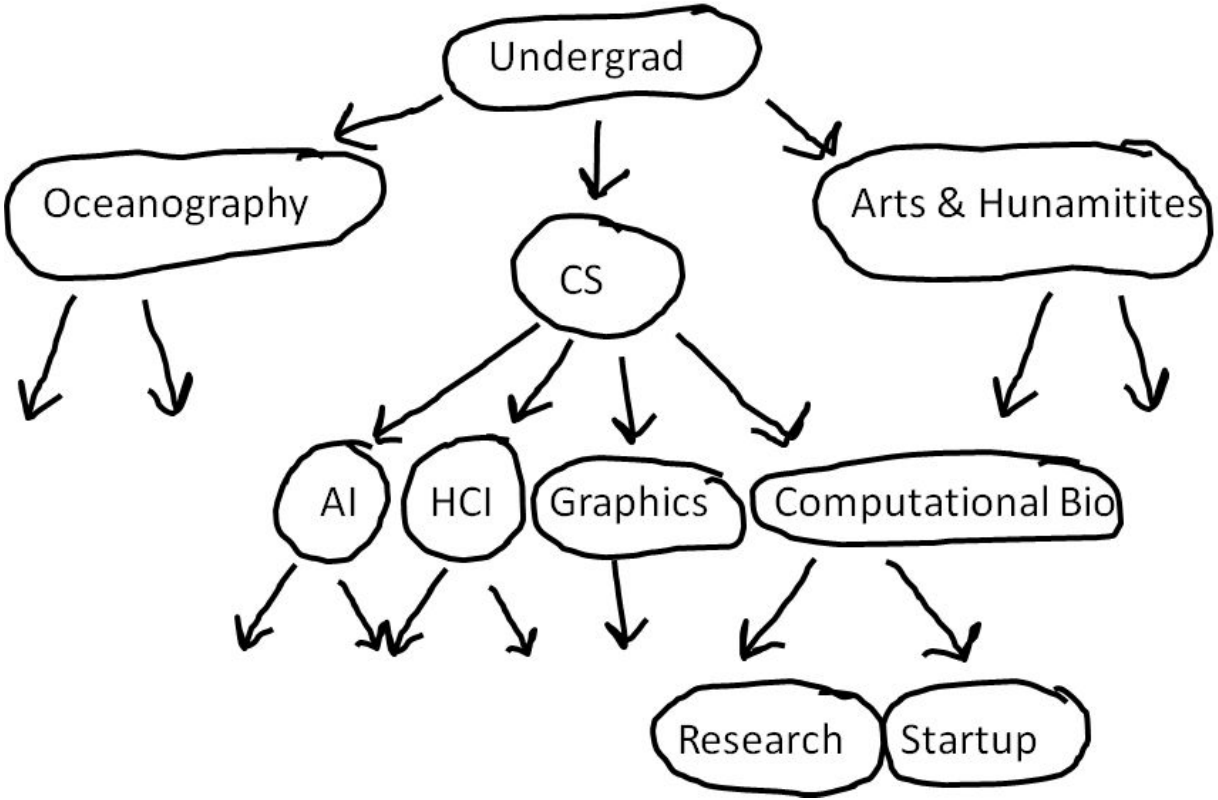
LINEAR DATA STRUCTURES

- So far we have only studied linear data structures
  - Arrays
  - Linked List
  - Stacks
  - Queues
- Each node had the information about at most single node.
- Linear Data Structures cannot represent non-linear relationships.
  - Family Tree
  - Decision Tree

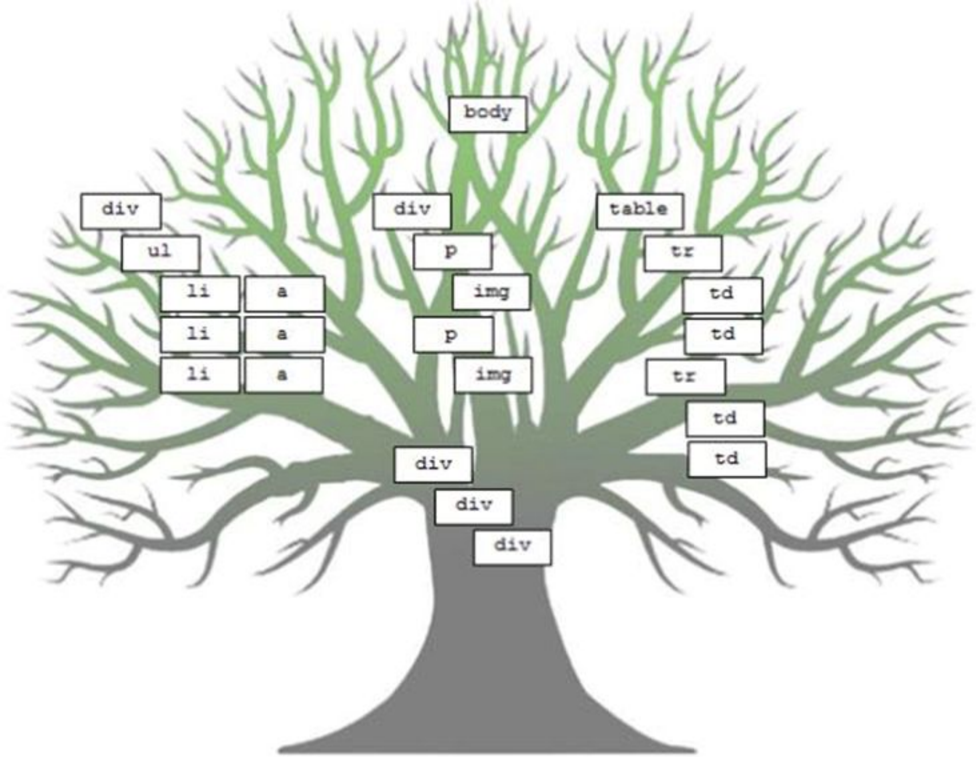
TREE DATA STRUCTURES



DECISION TREE



WEBPAGE


body
div
ul
li
a
li
a
li
a
div
p
img
p
img
table
tr
td
td
tr
td
td
div
div
div

TREE DATA STRUCTURE

- A hierarchical structure of nodes with parent child relationships.
- The starting node is called Root, it is the only node in the tree which does not have a parent node.
- Except root all nodes have parent node.
- Each parent can have many child nodes.
- In Binary Tree, the policy is to restrict each parent to have **at most 2 childs.**

SOME TERMS TO REMEMBER

- **Root:** The top most parent, or the starting node of the tree.
- **Parent:** The upward nodes which leads to the current node from the root.
- **Child:** The nodes descending the current node are childs of this node.
- **Siblings:** Childs of same parent, nodes having common parent node.
- **Leaf:** Node which does not have any child nodes.
- **Subtree:** All descendants of the current node forming a tree.
- **Level:** A complete generation of the nodes, all children of a parent are on the same level. Root is on level 0, its child nodes are on level 1, and grand child is on level 2.

SOME TERMS TO REMEMBER

- **Degree of a Node:** The number of children of that node, how many pointers current node have, or how many outgoing edges it has.
- **Degree of Tree:** The degree of a tree is the maximum degree of nodes in given tree.
- **Path:** The sequence of nodes from root to the target node.
- **Height of a Node:** Maximum path length from current node to the leaf.
- **Height of a Tree:** The height of root node.
- **Depth of a Tree:** Max level of any leaf node of the tree.

SOME TERMS TO REMEMBER

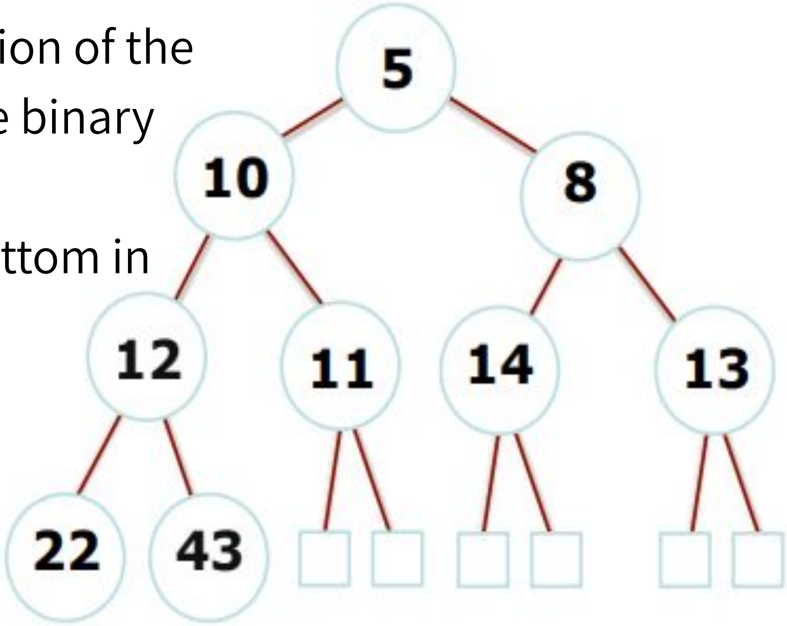
- **Complete Binary Tree:** A tree completely filled with the exception of the last level, and all nodes are as far left as possible.

Heap is a tree based structure that satisfies the **Heap Property**

Parents have a higher priority key than any of their children

BINARY HEAPS

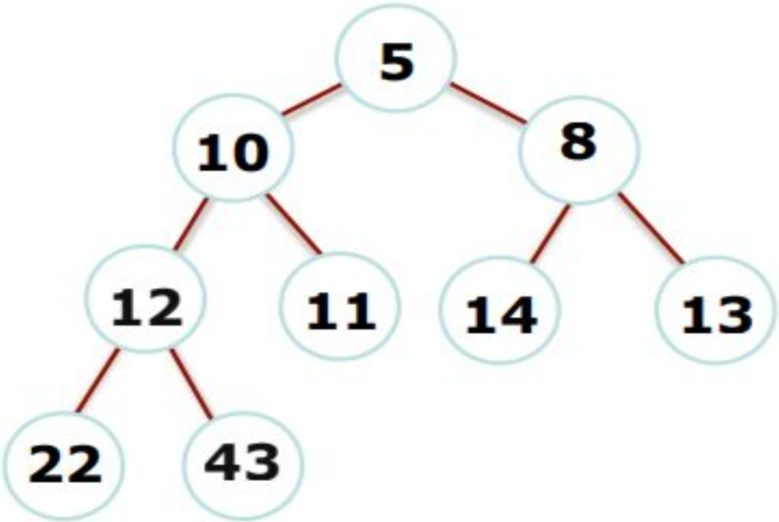
- Heaps are completely filled, with the exception of the bottom level. They are, therefore, "complete binary trees":
  - complete: all levels filled except the bottom in the leftmost positions.
  - binary: two children per node (parent)
- Maximum number of nodes
- Filled from left to right



TWO TYPES OF BINARY HEAP

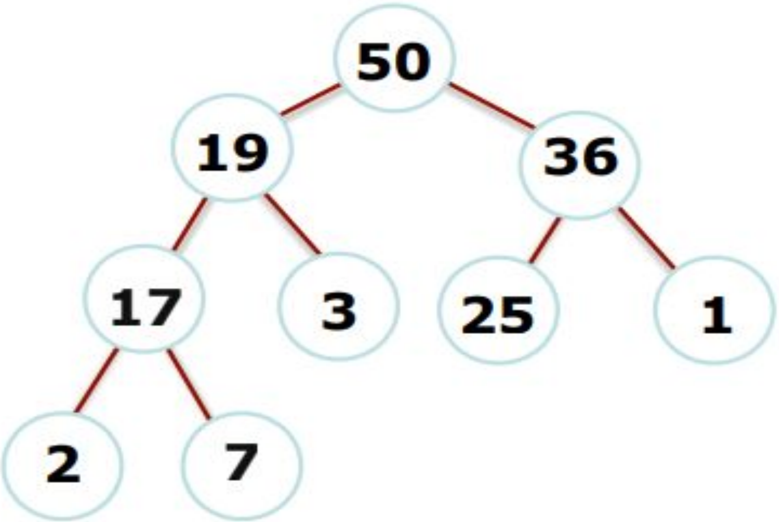
Min Heap

(root is the smallest element)



Max Heap

(root is the largest element)

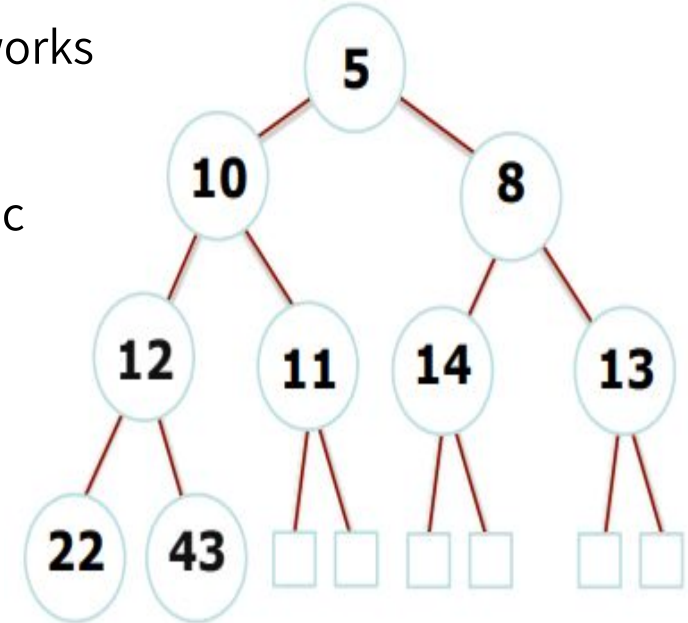


HEAPS

- The root in a max heap is the largest element.
- The root in a min heap is the smallest element.
- This is ensured by the Heap Property.
- The tree is perfectly balanced and the leaves in last level are all in the leftmost positions.
    - The number of levels in the tree is ***O (n)***.

ARRAYS OR LINKED LIST FOR HEAPS

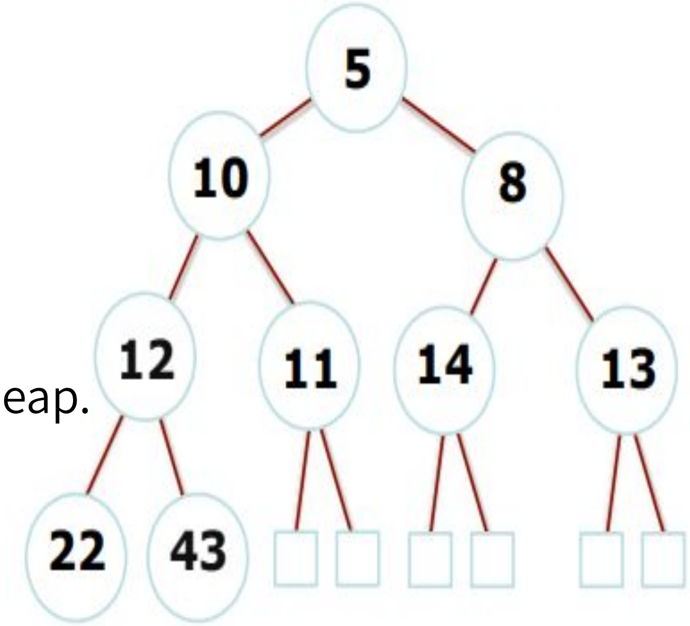
- We can use Linked list, but it turns out an Array works really well for storing a binary heap.
- If we start from index 1 instead of 0, the arithmetic becomes very clean.



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

ARRAYS OR LINKED LIST FOR HEAPS

- Determining the parent and children of a node becomes simple arithmetic.
  - Left child is at ***2i***
  - Right Child is at ***2i + 1***
  - parent is at $\lfloor i / 2 \rfloor$
  - Heap size is the number of elements in the heap.

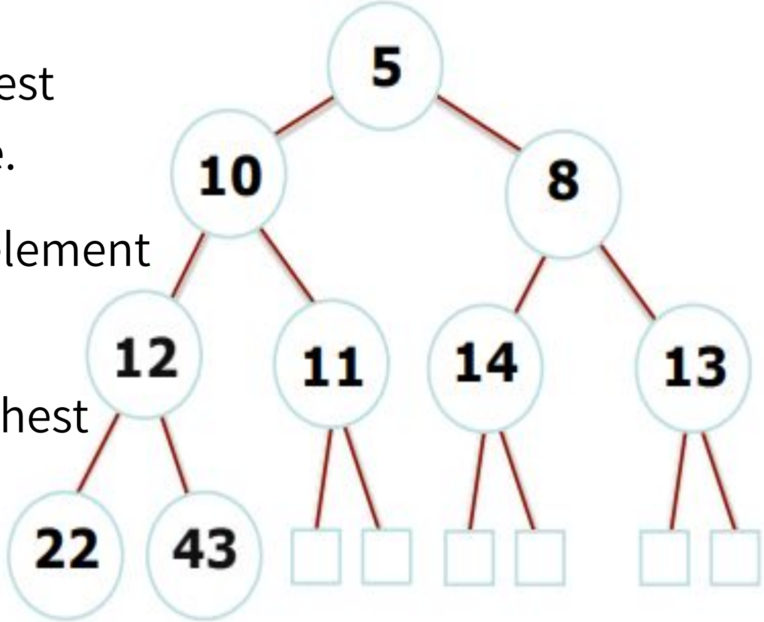


	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

HEAP OPERATIONS

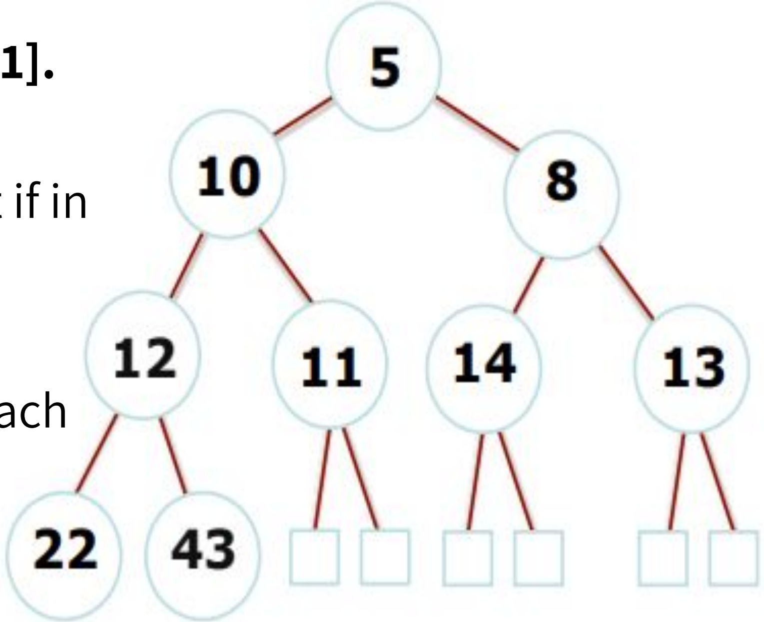
Recalling the Operations of Priority Queue

- **peek ( )** - returns the element with the highest priority without removing it from the queue.
- **enqueue (priority, element)** - inserts an element with priority p into the queue.
- **dequeue ( )** - removes the element with highest priority from the queue.



HEAP OPERATIONS : enqueue( )

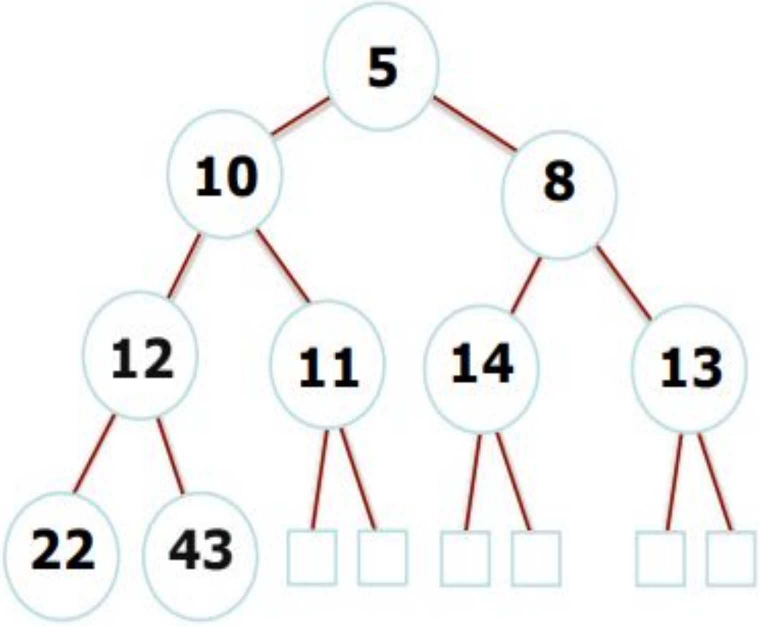
- Insert item at element **heap [heap.size ( ) + 1].**
- Perform a bubble up or up-heap operation
  - Compare the new element with it parent if in correct order stop.
  - If not in order, swap parent with the new element and repeat the step until you reach root.



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

HEAP OPERATIONS : peek ( )

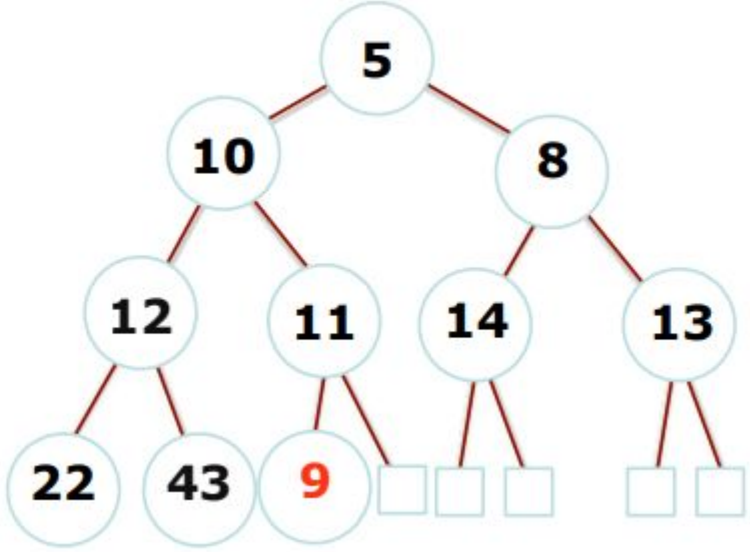
```
peek ( ) {
    return heap [ 1 ];
}
```



Complexity Time: ***O (1)***

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

HEAP OPERATIONS : enqueue(9)



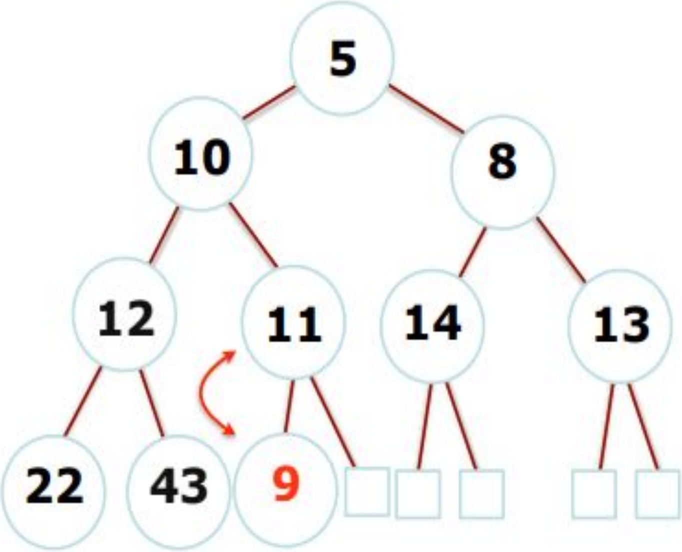
- Start by inserting the element 9 at the empty position.
- The empty position is always at
  - heap.size ( ) + 1.

Is the Heap Property Satisfied...?

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

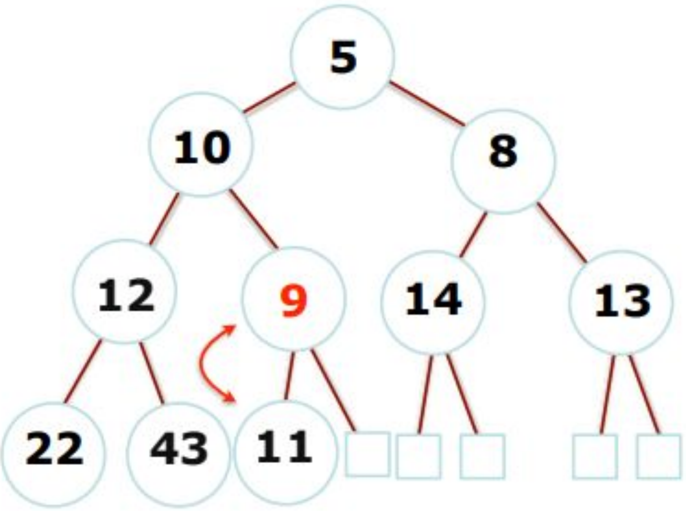
VISUALIZATION

HEAP OPERATIONS : 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 OPERATIONS : enqueue(9)

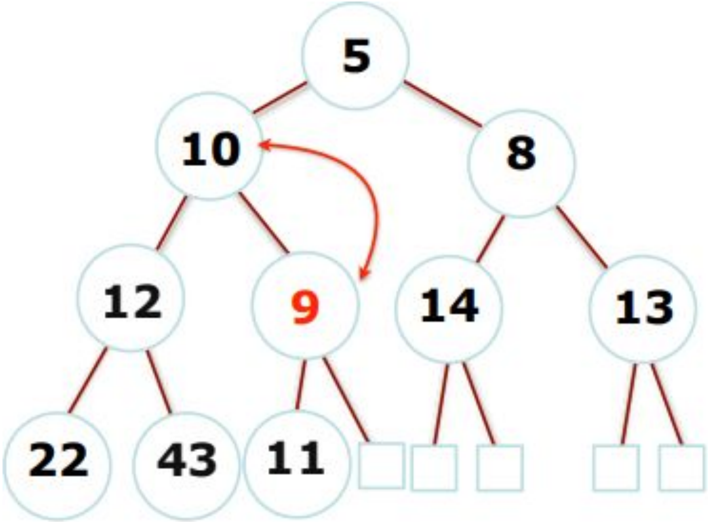


	5	10	8	12	9	14	13	22	43	11	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Will this bubbling up create any problems if the data is already following heap property.???

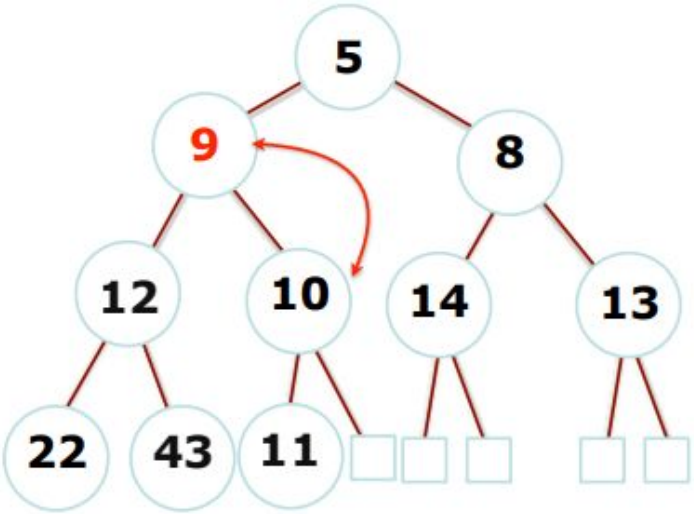
- Now we look at the parent of index 5, is it greater than its parent.
- NO…!
  - We need to swap again

HEAP OPERATIONS : 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 OPERATIONS : 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 Property attained, no more swaps.

Time Complexity *O (log n).*

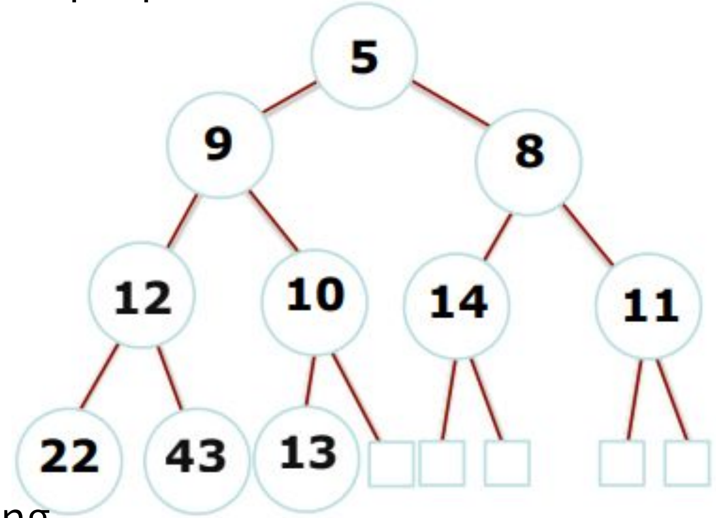
Average Time Complexity *O (1).*

Why O (1) ? Have a read..!
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6312854

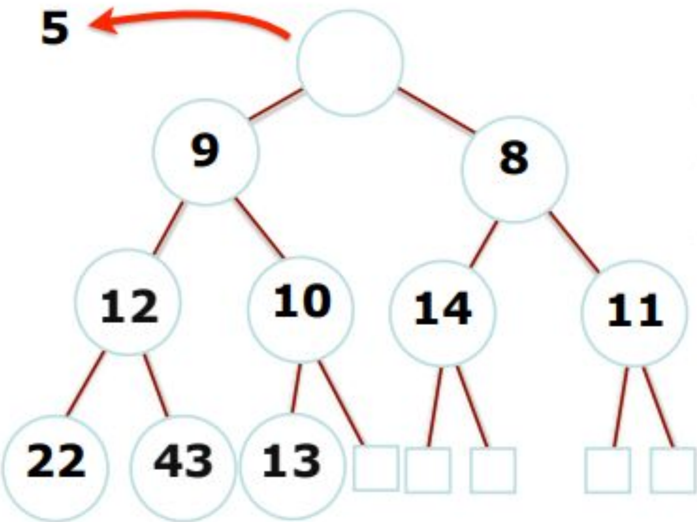
HEAP OPERATIONS : dequeue ( )

what about Dequeue ???

- We are removing the root, we need to retain the two properties of our heap.
  - Smallest element on top.
  - Complete Binary Tree.
- Replace root with the last element.
- Bubble down or down-heap the new root
  - Compare it with its children.
  - If it is in the correct position stop
  - If not swap the smallest child, and keep repeating.
  - It is essential to check if the current node have any children or not.



HEAP OPERATIONS : dequeue ( )

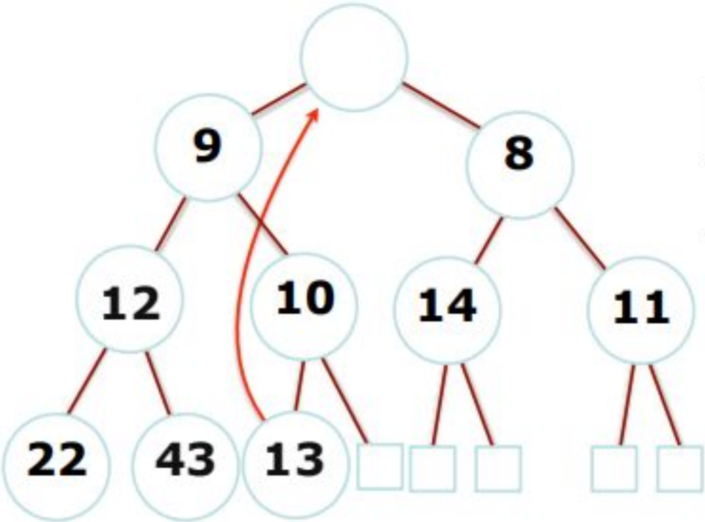


	5	9	8	12	10	14	11	22	43	13	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

HEAP OPERATIONS : dequeue ( )

Move last element (heap [heap.size ( )]) to the root heap [1].

Start the bubble down or down heap process.



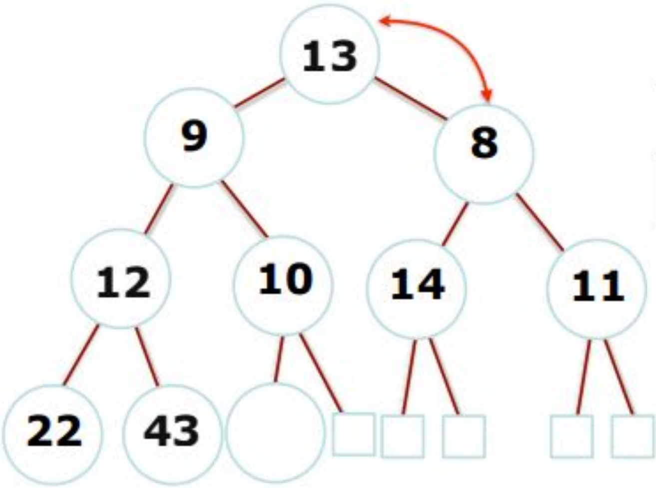
	5	9	8	12	10	14	11	22	43	13	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Don't forget to decrease heap size!

HEAP OPERATIONS : dequeue ( )

Compare the root with children

Swap with the smaller child. Have any thoughts why???

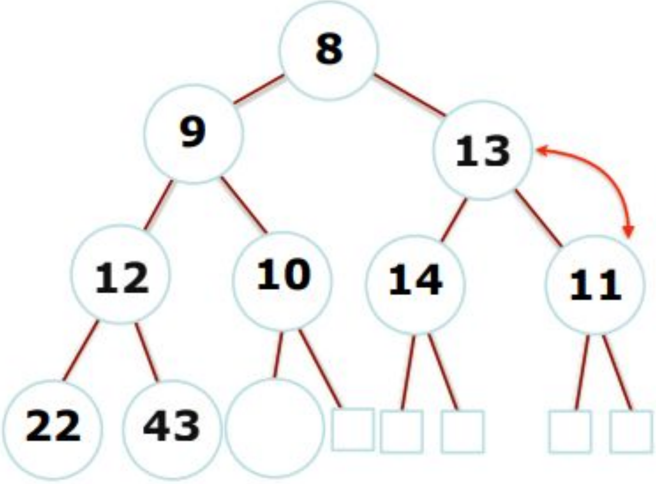


	13	9	8	12	10	14	11	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

HEAP OPERATIONS : dequeue ( )

Keep Swapping if necessary,

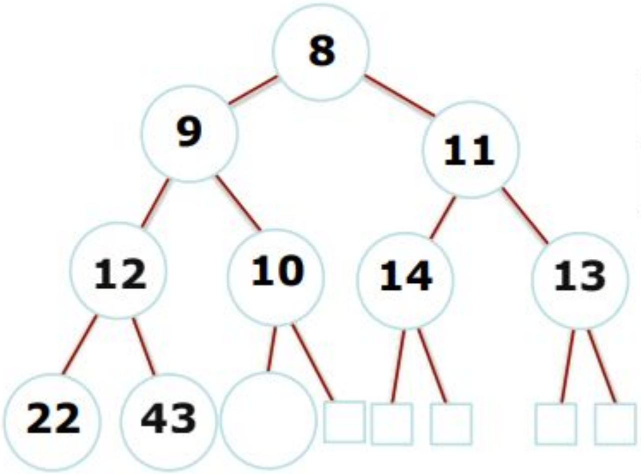
Now we will compare it with 14 & 11 and swap with 11.



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

HEAP OPERATIONS : dequeue ( )

13 has reached its proper position.



	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!

HEAPS IN REAL LIFE

- Heap Sort (Will discuss in next Lecture).
- Google Maps
  - Finding the shortest path between places
- All priority queue situations
- Huffman coding.
- Kernel processing scheduling.

BUILDING HEAP FROM SCRATCH

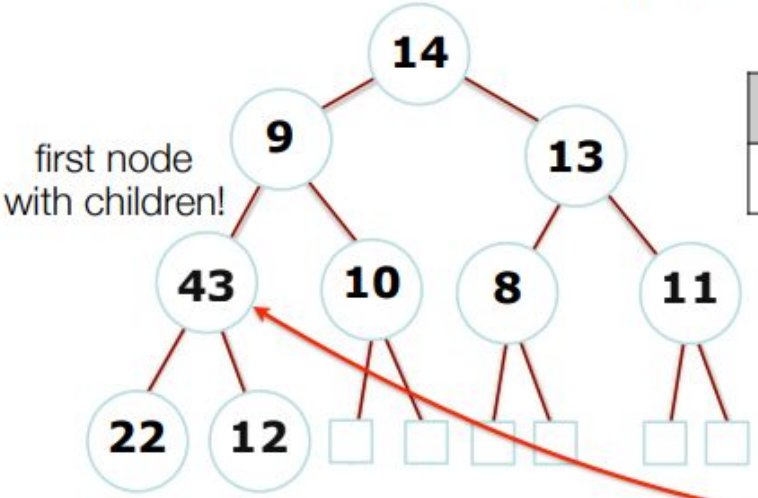
14, 9, 13, 43, 10, 8, 11, 22, 12

- We could insert each in turn.
  - Insertion takes *O (log n)* and we have to insert n items *O (n log n).*
- There is a better way
  - **Heapify ( )**
  - Start from the lowest completely filled level at the first node with children.
  - Down heap each element

```
for (int i = heapSize/2; i > 0; i--) {
      downheap ( i );
 }
```

BUILDING HEAP FROM SCRATCH

14, 9, 13, 43, 10, 8, 11, 22, 12

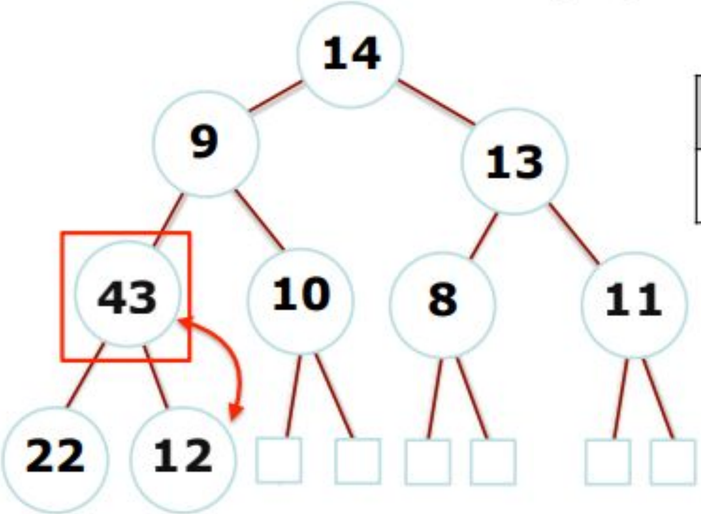


	14	9	13	43	10	8	11	22	12		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

loop down:
i=heapSize/2
heapSize==9,
i==4

BUILDING HEAP FROM SCRATCH

14, 9, 13, 43, 10, 8, 11, 22, 12

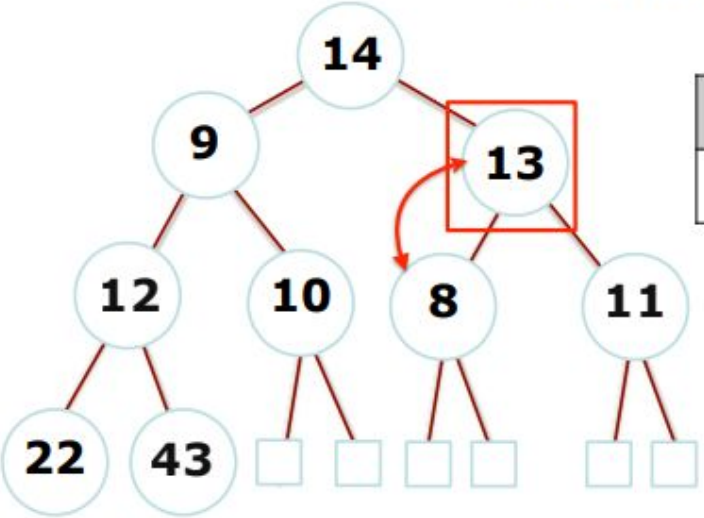


	14	9	13	43	10	8	11	22	12		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

i==4

BUILDING HEAP FROM SCRATCH

14, 9, 13, 43, 10, 8, 11, 22, 12

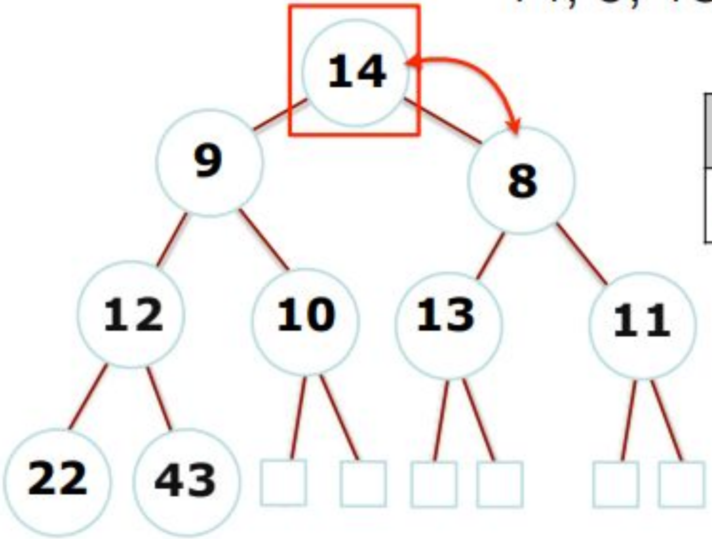


	14	9	13	12	10	8	11	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

i==3

BUILDING HEAP FROM SCRATCH

14, 9, 13, 43, 10, 8, 11, 22, 12

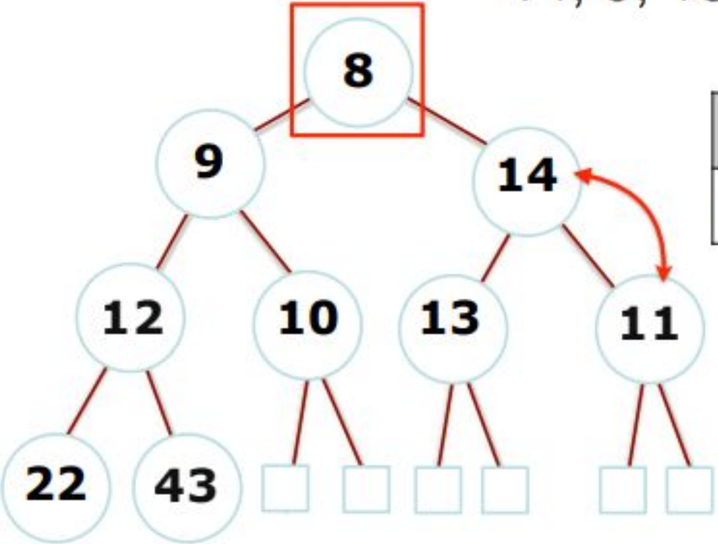


	14	9	8	12	10	13	11	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

i==1

BUILDING HEAP FROM SCRATCH

14, 9, 13, 43, 10, 8, 11, 22, 12



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

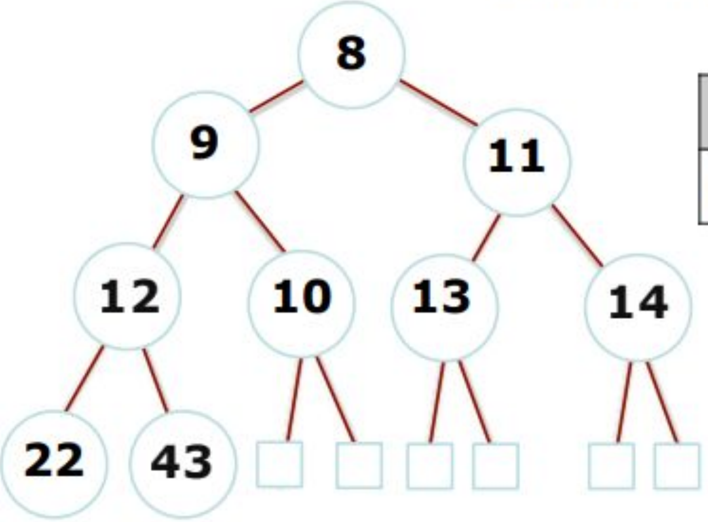
must keep down-heaping

BUILDING HEAP FROM SCRATCH

Don't think it should take *O (n)* ??? Have a look!

http://www.cs.umd.edu/~meesh/351/mount/lectures/lect14-heapsort-analysis-part.pdf

14, 9, 13, 43, 10, 8, 11, 22, 12



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

Done!
We now have a proper min-heap.
Asymptotic complexity — not trivial to determine, but turns out to be O(n).