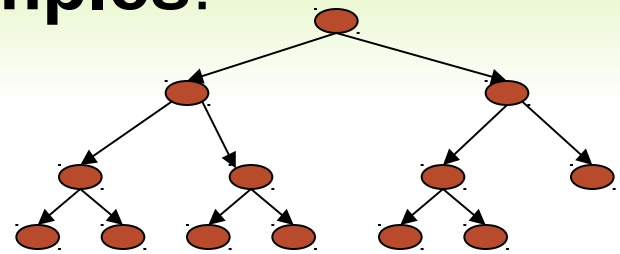
Heap Structure

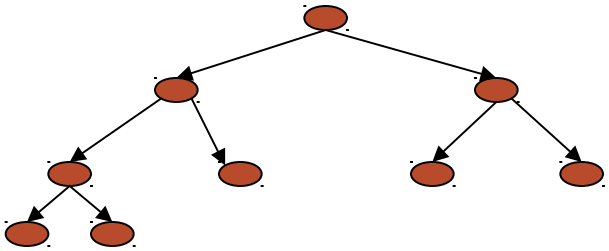
Heap **Structure** Property

A binary heap is a ***complete*** binary tree.

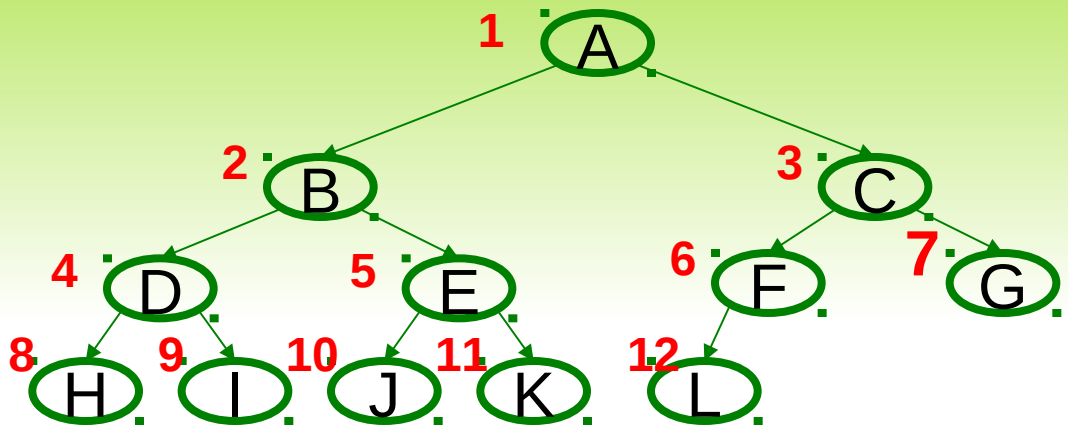
Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:





Representing Complete Binary Trees in an Array

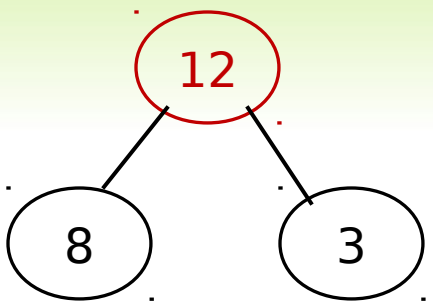


implicit (array) implementation:

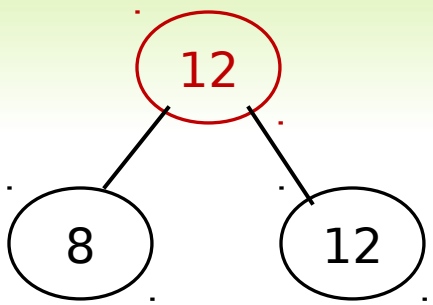
	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

The heap property

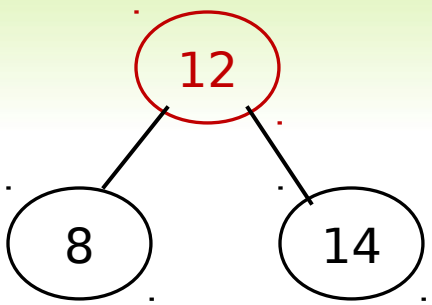
- A node has the heap property if the value in the node is as large as or larger than the values in its children



Red node has heap property



Red node has heap property

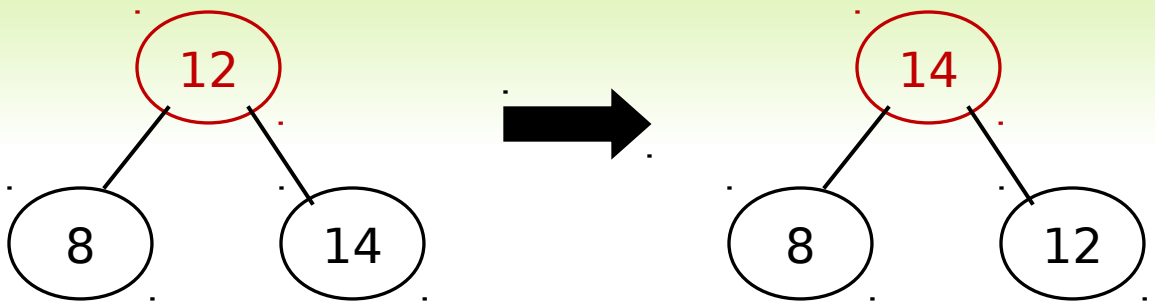


Red node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

Shift Up

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



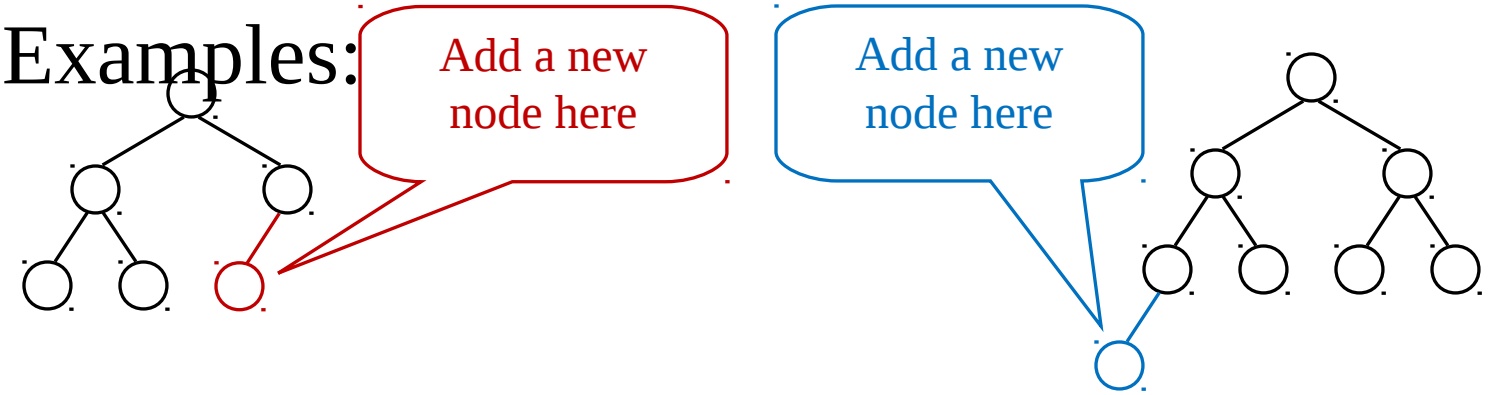
Red node does not have heap property

Red node has heap property

- This is sometimes called shifting up
- Notice that the child may have *lost* the heap property

Constructing a heap I

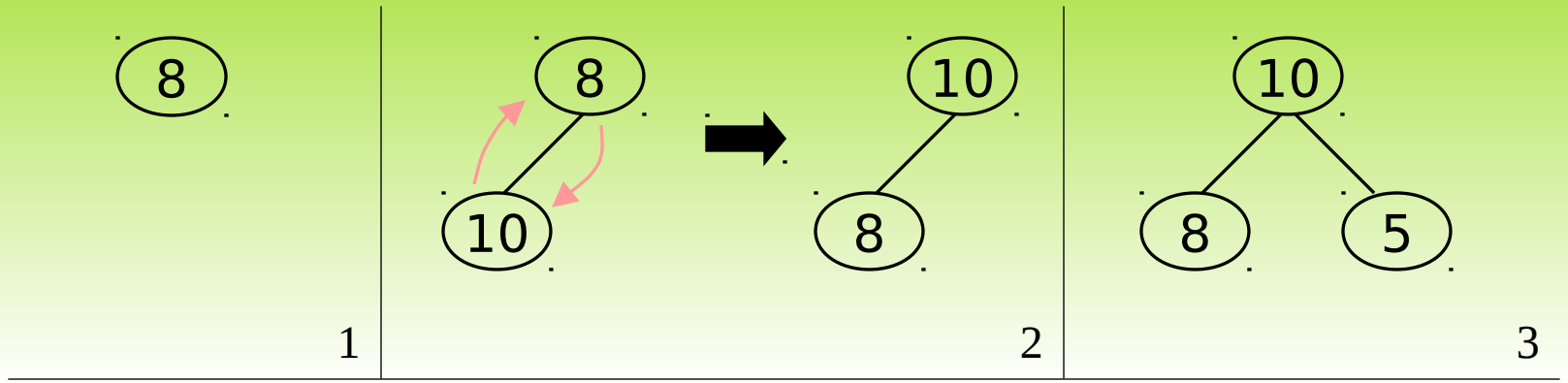
- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:

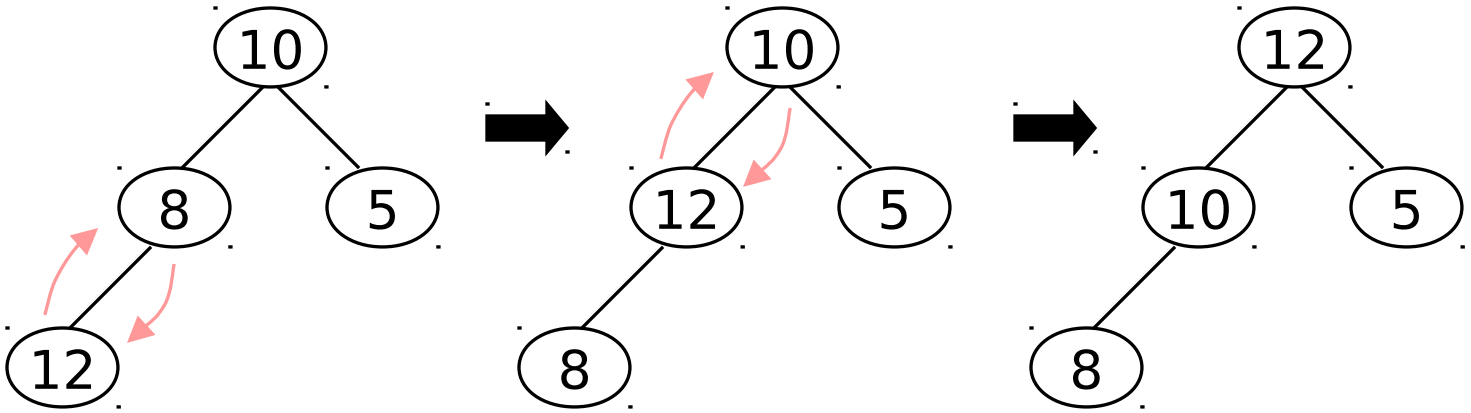


Constructing a heap II

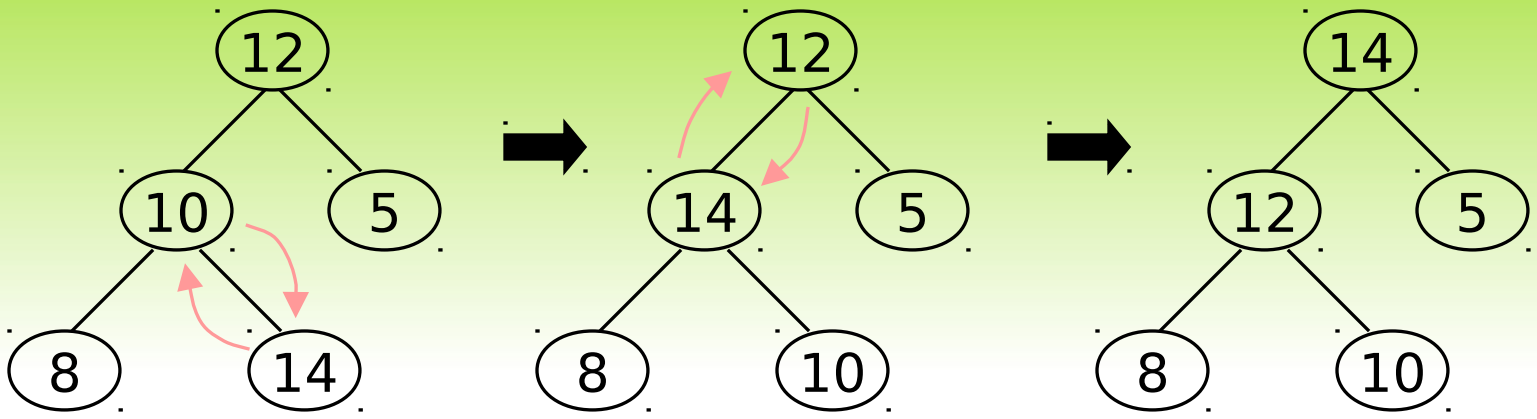
- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we shift up
- But each time we shift up, the value of the topmost node in the shift may increase, and this may destroy the heap property of *its* parent node
- We repeat the shifting up process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root

Constructing a heap III





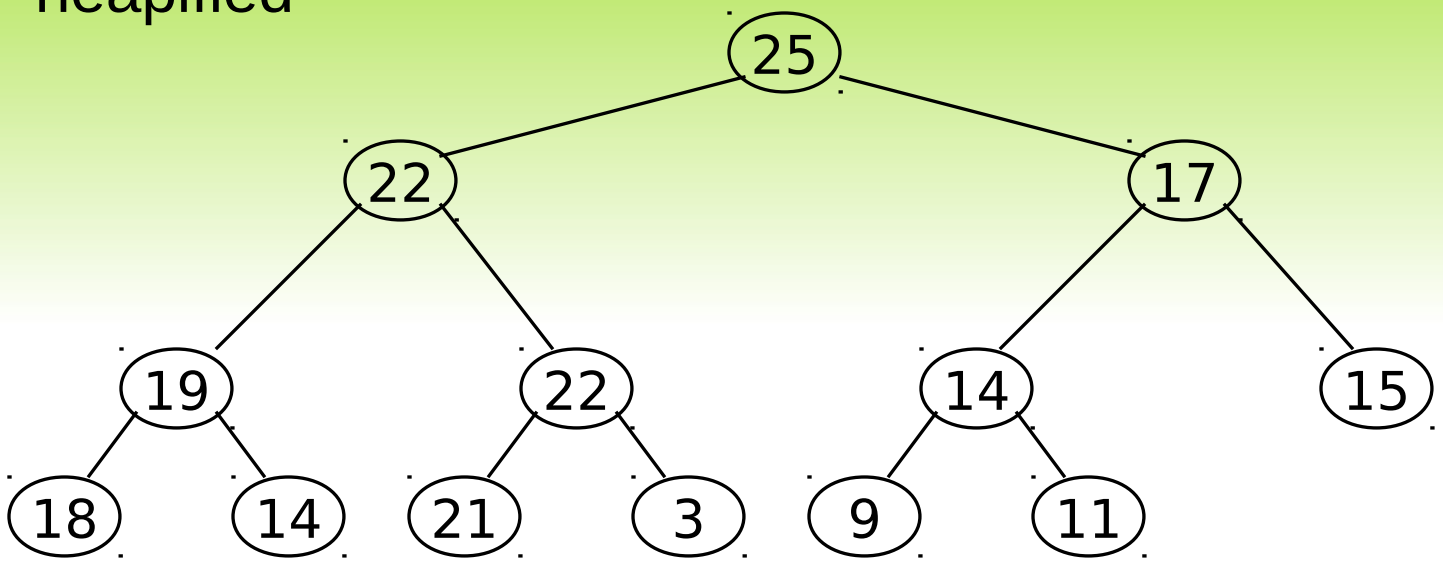
Other children are not affected



- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

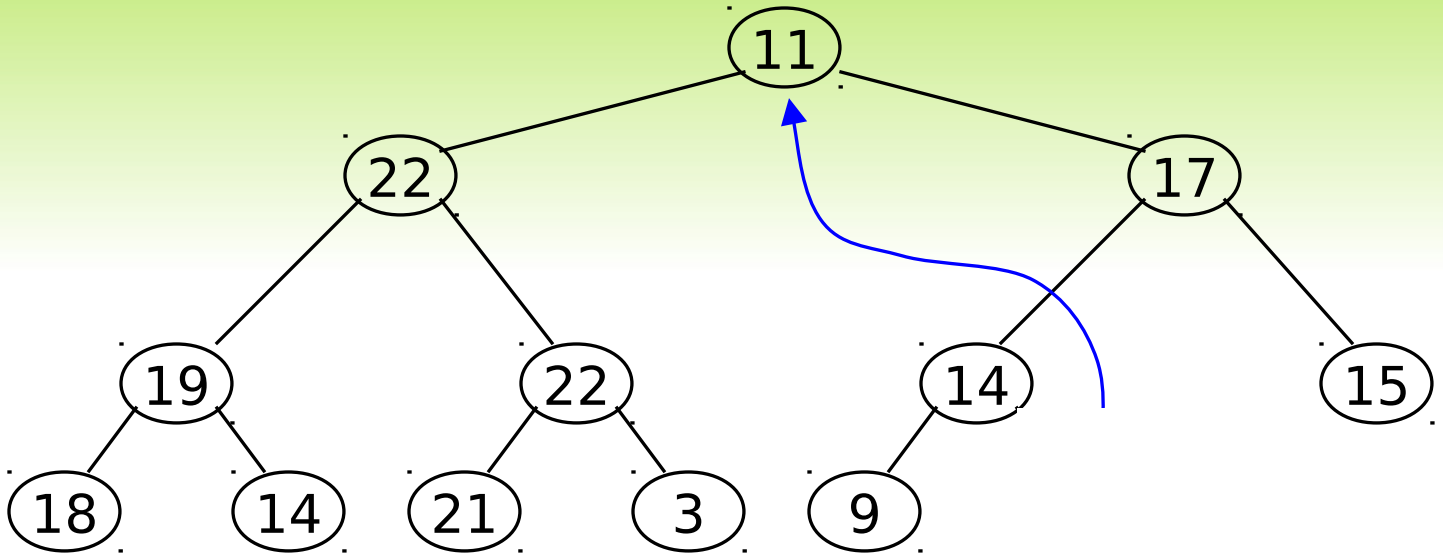
- Here's a sample binary tree after it has been heapified



- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Removing the root

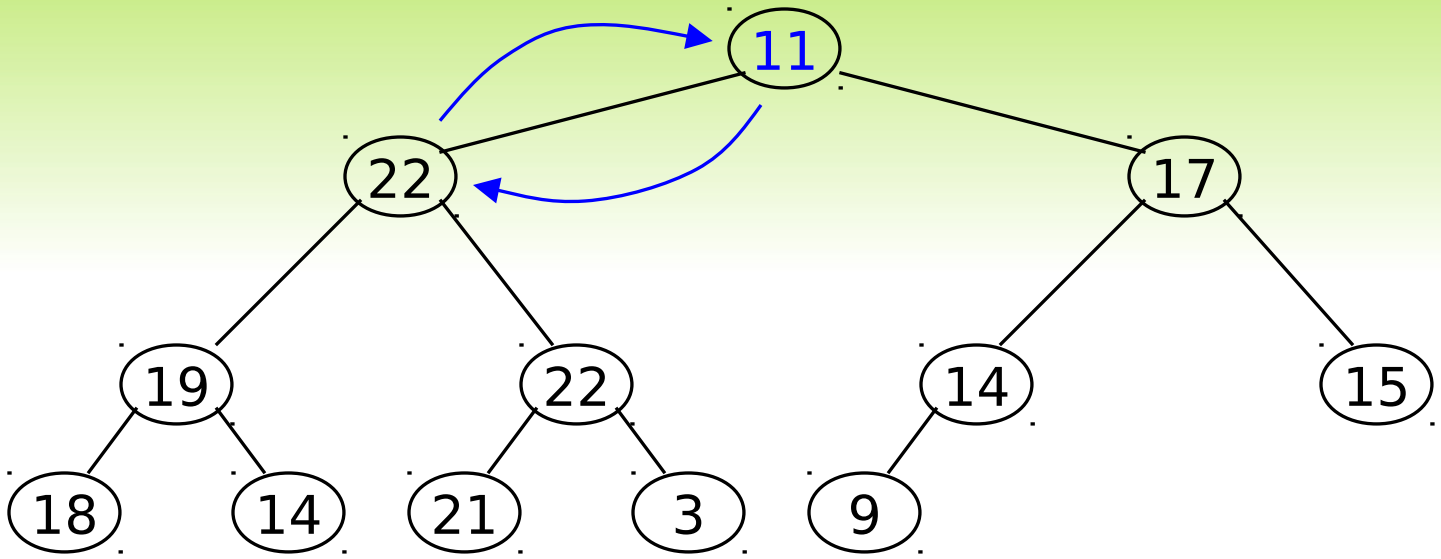
- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- How can we fix the binary tree so it is once again *balanced and left-justified?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

The ReHeap method I

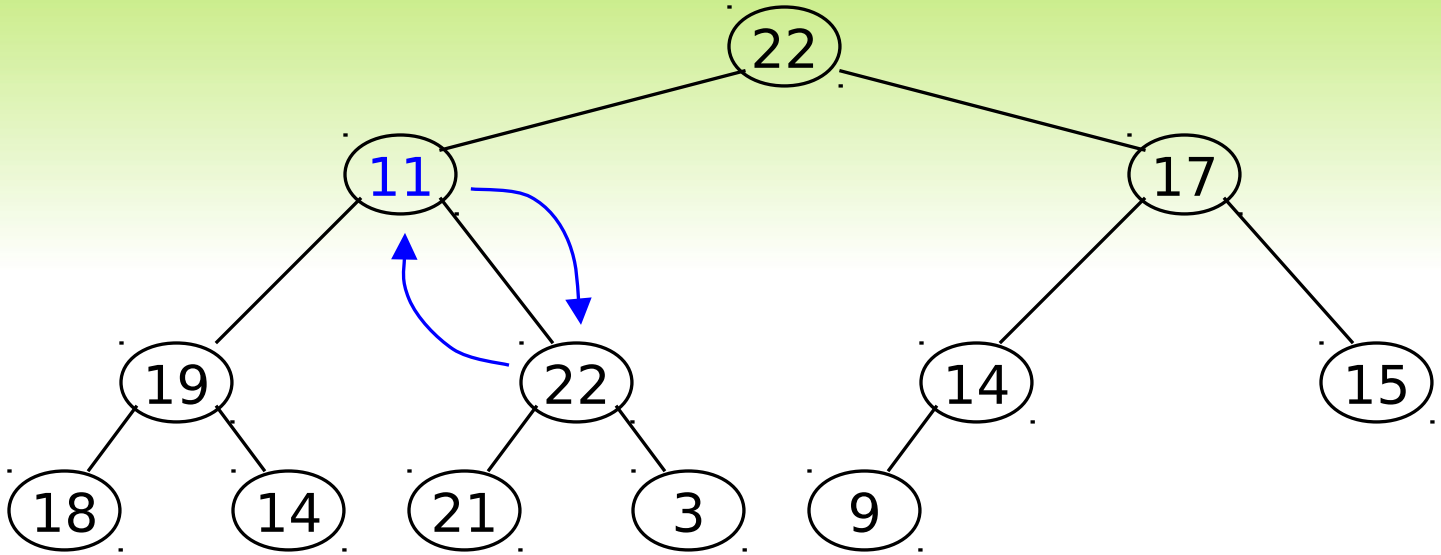
- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property



- We can `shiftUp()` the root
- After doing this, one and only one of its children may have lost the heap property

The reHeap method II

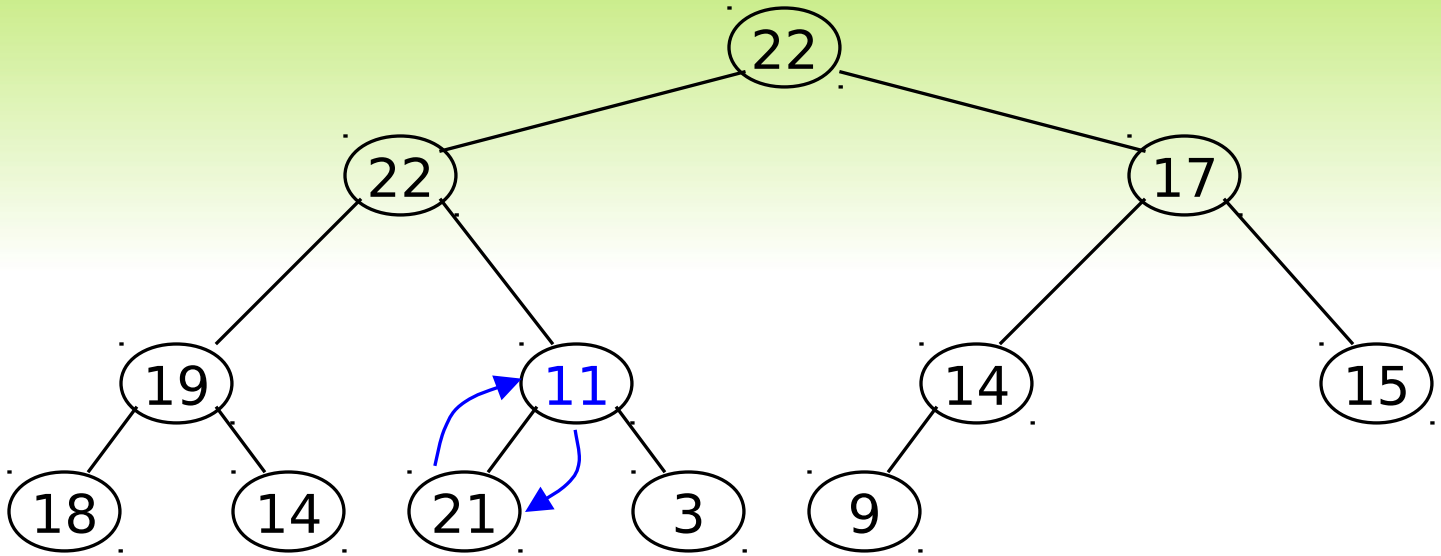
- Now the left child of the root (still the number 11) lacks the heap property



- We can shiftUp() this node
- After doing this, one and only one of its children may have lost the heap property

The reHeap method III

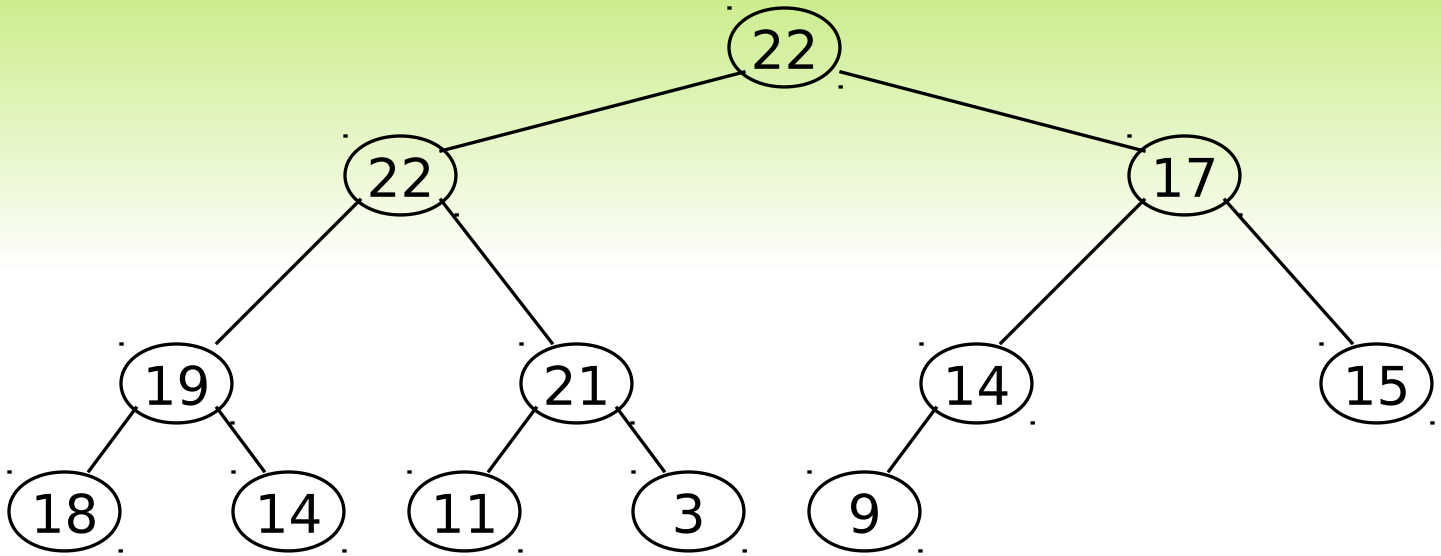
- Now the right child of the left child of the root (still the number 11) lacks the heap property:



- We can shiftUp() this node
- After doing this, one and only one of its children may have lost the heap property —but it doesn't, because it's a leaf

The reHeap method IV

- Our tree is once again a heap, because every node in it has the heap property



- Once again, the largest (or *a* largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest

Sample Run

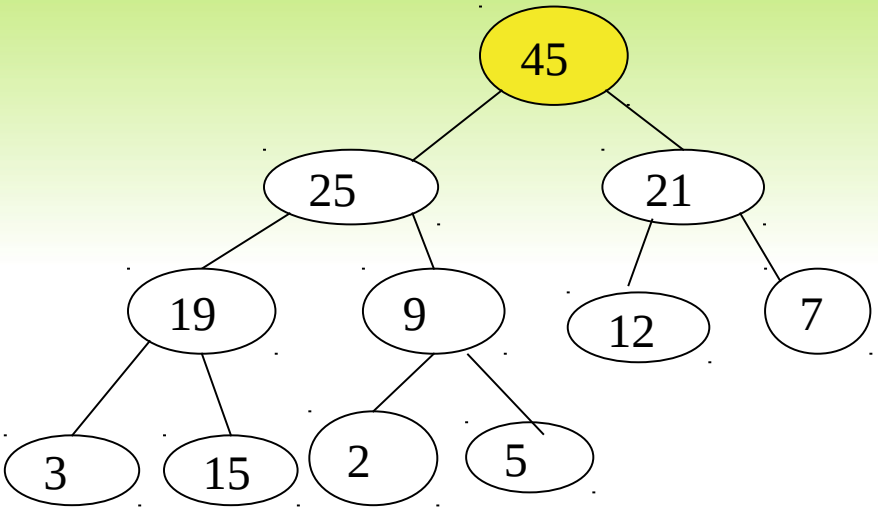
- Start with unordered array of data

Array representation:

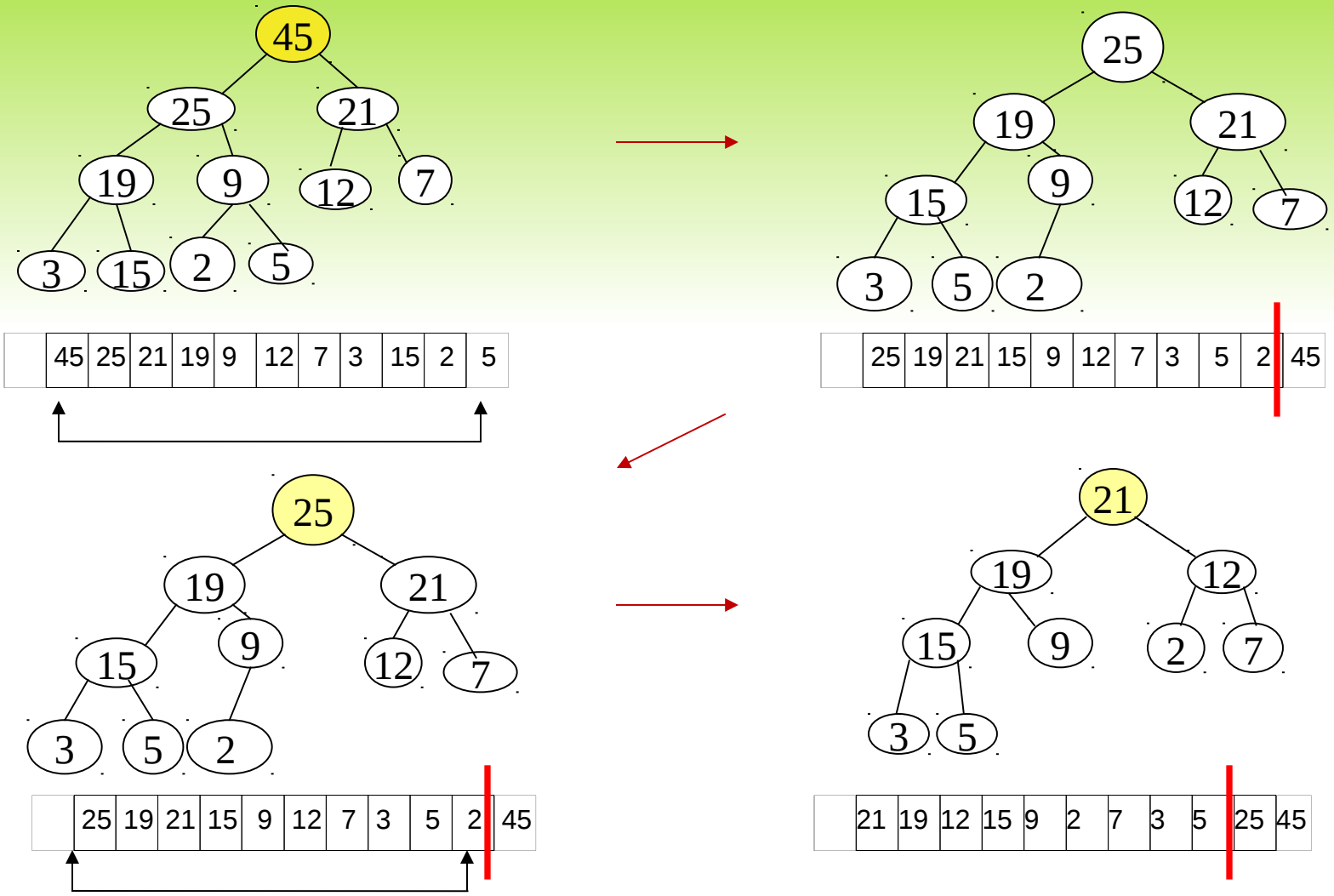
	21	15	25	3	5	12	7	19	45	2	9

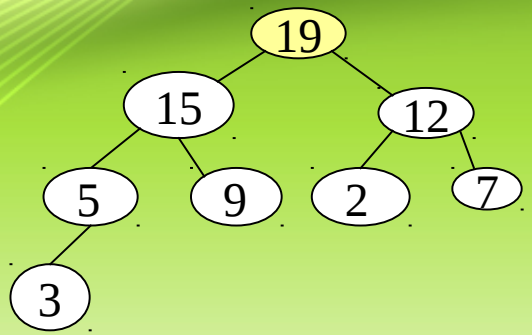
Sample Run

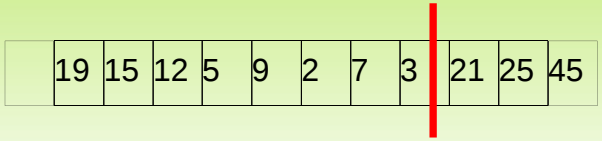
- Heapify the binary tree -

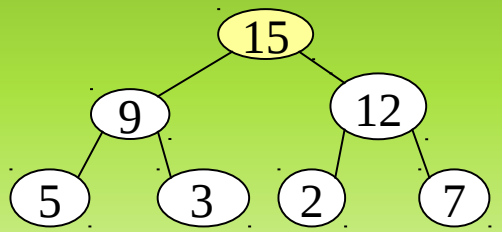


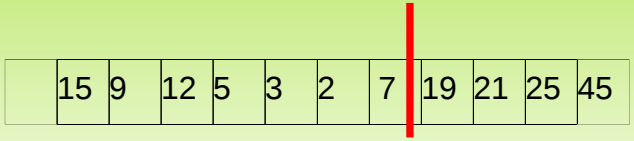
Step 2 – perform n – 1 deleteMax(es), and replace last element in heap with first, then re-heapify. Place deleted element in the last nodes position.

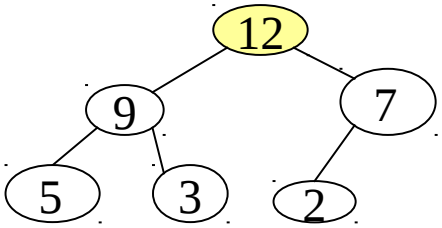


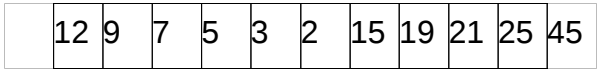

19
15
12
5
9
2
7
3


19 15 12 5 9 2 7 3 21 25 45


15
9
12
5
3
2
7

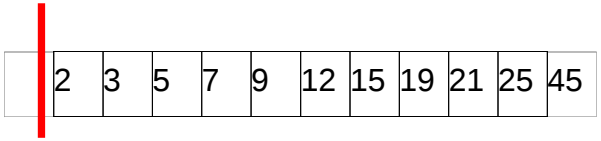

15 9 12 5 3 2 7 19 21 25 45


12
9
7
5
3
2


12 9 7 5 3 2 15 19 21 25 45

…and finally


2


2 3 5 7 9 12 15 19 21 25 45

Exercise:

Sort the given numbers using heap tree approach:

- 20, 35, 9, 26, 49, 78, 2, 46
- 25, 29, 36, 32, 38, 44, 40, 54