

Heapsort

Compiled from Internet

Thanks to the internet community to share resources

Why study Heapsort?

- It is a well-known, traditional sorting algorithm you will be expected to know
- Heapsort is *always* $O(n \log n)$
 - Quicksort is usually $O(n \log n)$ but in the worst case slows to $O(n^2)$
 - Quicksort is generally faster, but Heapsort is better in time-critical applications

What is a “heap”?

Definitions of **heap**:

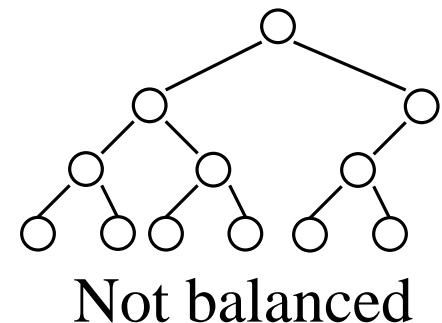
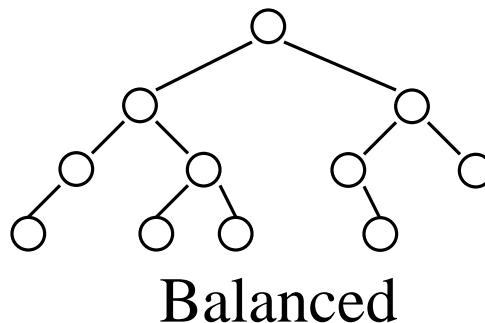
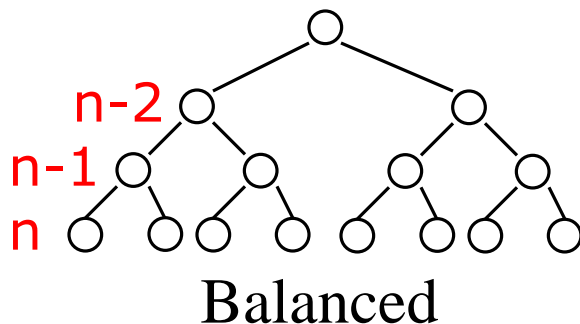
A balanced, left-justified binary tree in which no node has a value greater than the value in its parent

Binary tree:

A tree node has at most two children

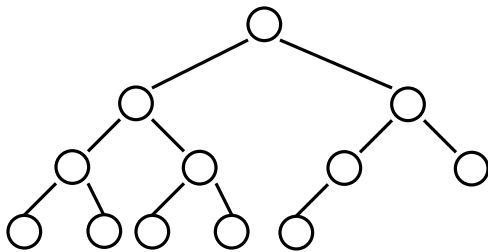
Balanced binary trees

- Recall:
 - The **depth** of a node is its distance from the root
 - The **depth** of a tree is the depth of the deepest node
- A binary tree of depth **n** is **balanced** if all the nodes at depths **0** through **$n-2$** have two children

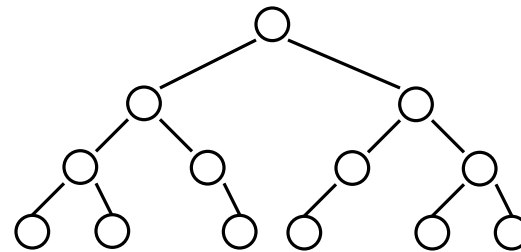


Left-justified binary trees

- A balanced binary tree is **left-justified** if:
 - all the leaves are at the same depth, or
 - all the leaves at depth $n+1$ are to the left of all the nodes at depth n



Left-justified



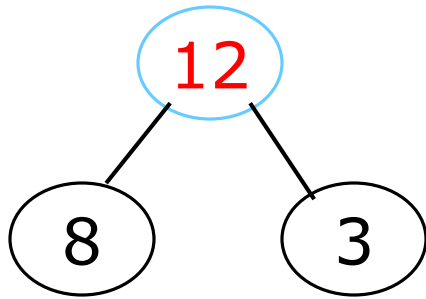
Not left-justified

Use heap to sort

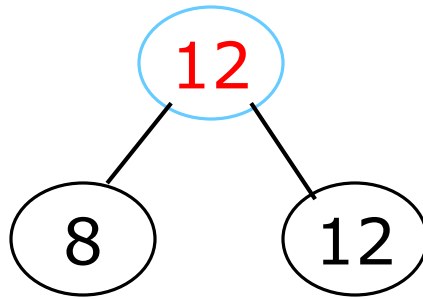
- How to turn a binary tree into a heap
- How to turn a binary tree *back* into a heap after it has been changed in a certain way
- How to use these ideas to sort an *array*

The heap property

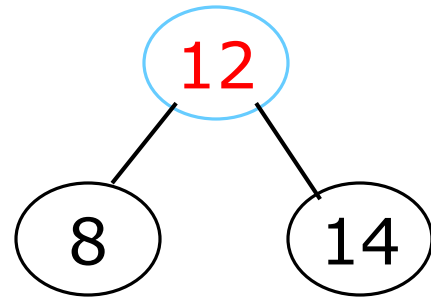
- A node has the **heap property** if the value in the node is as large as or larger than the values in its children



Blue node has
heap property



Blue node has
heap property

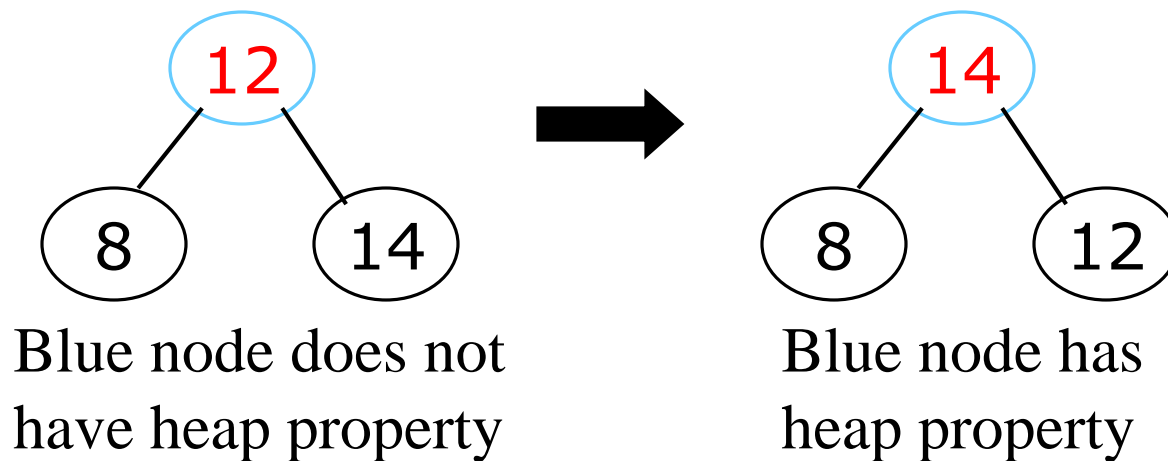


Blue node does not
have heap property

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

siftUp

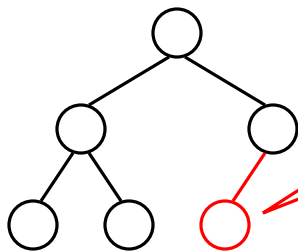
- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



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

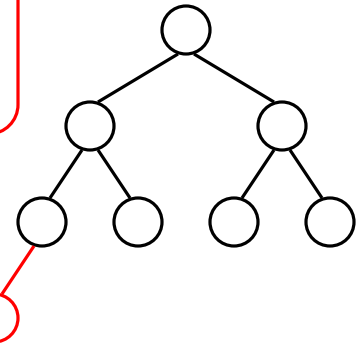
Constructing a heap I

- 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:



Add a new
node here

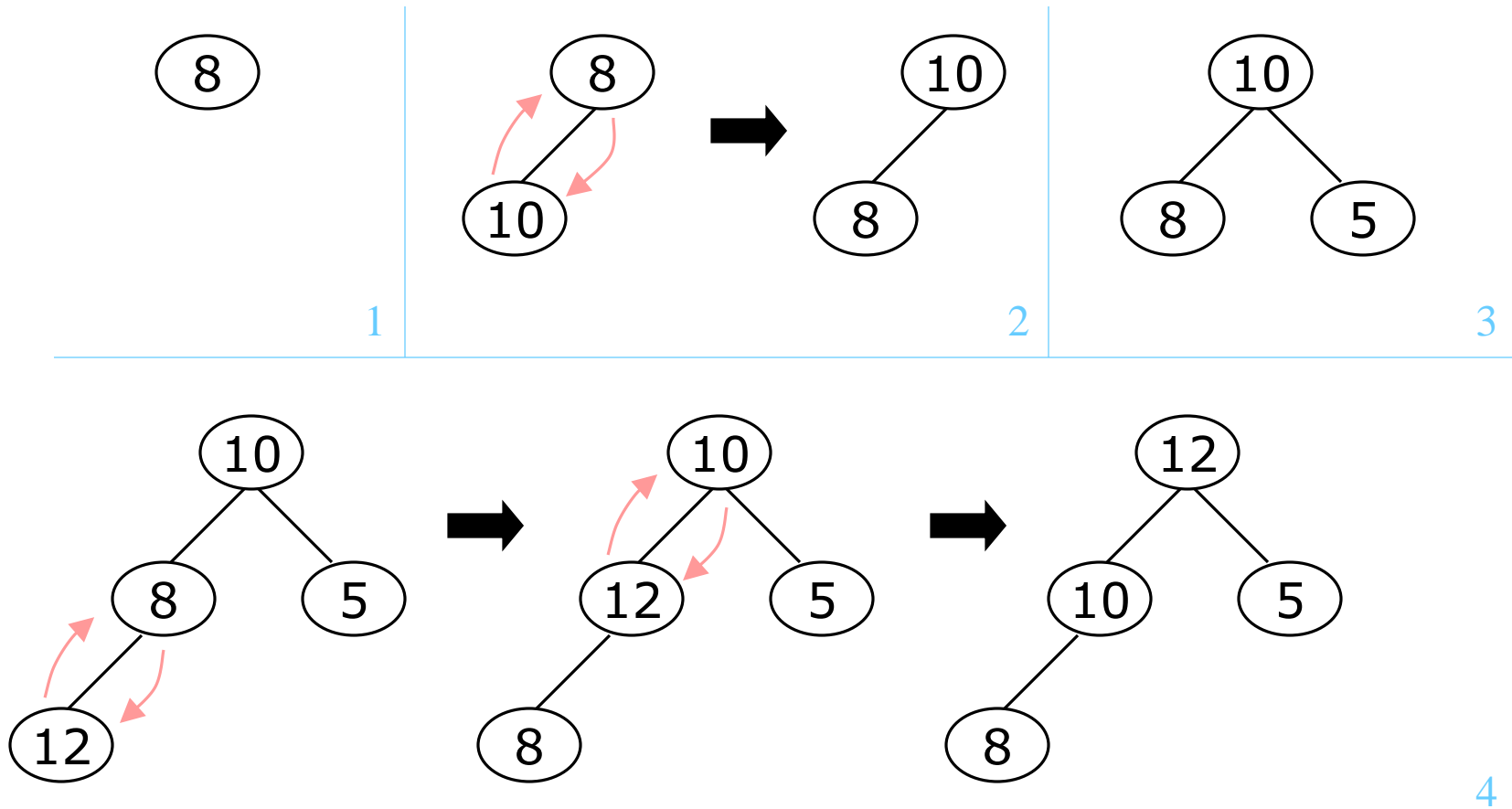
Add a new
node here



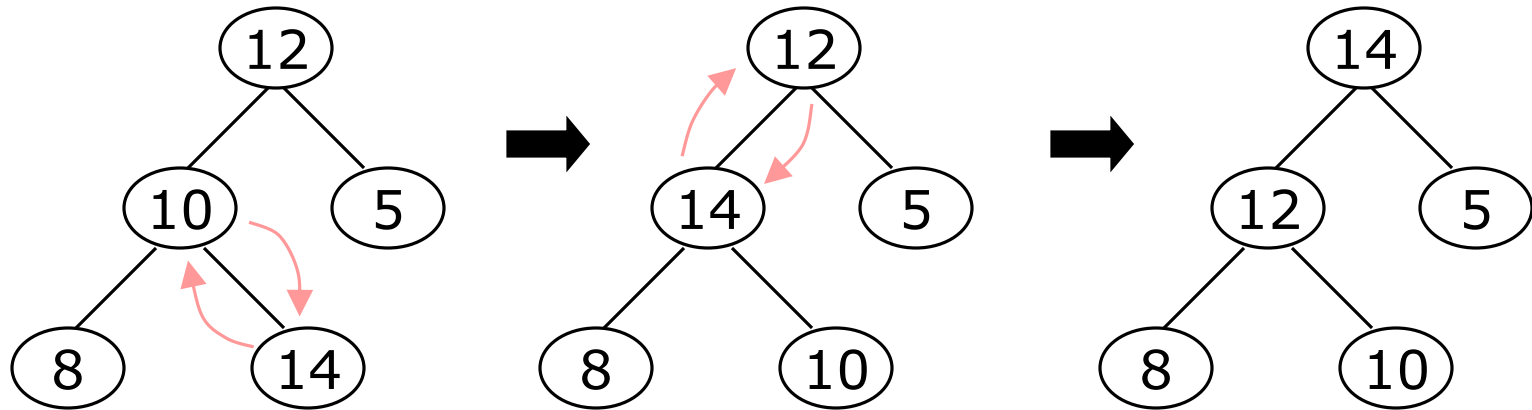
Constructing a heap II

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we **sift up**
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node
- We **repeat** the **sifting 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



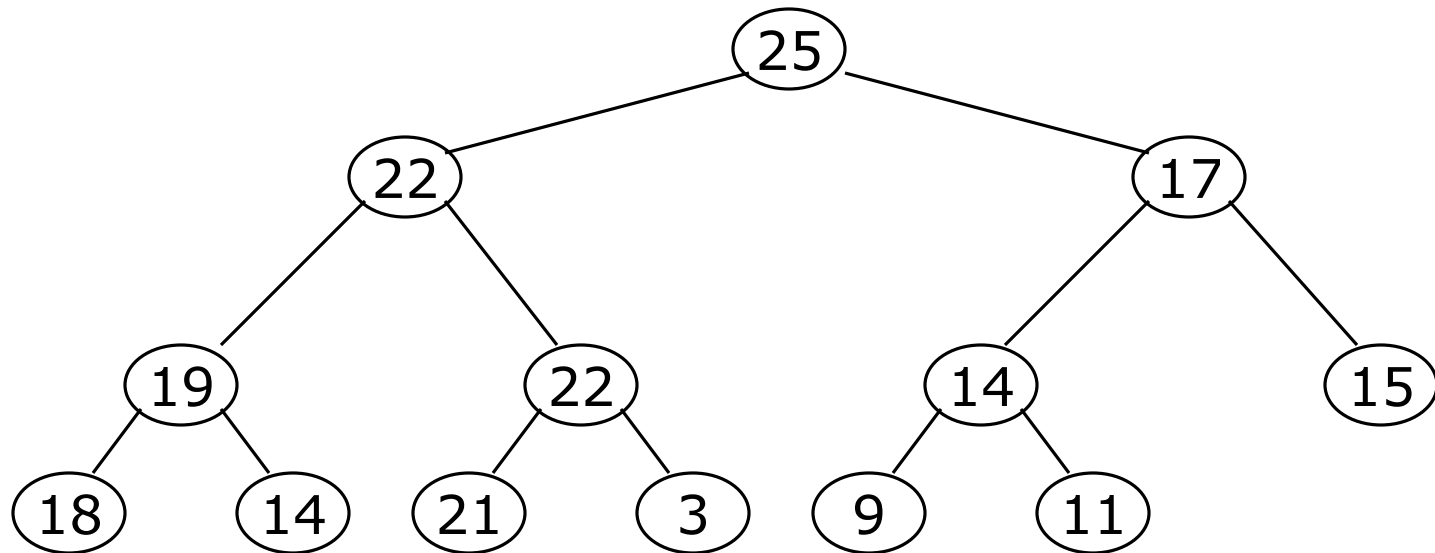
Other children are not affected



- 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

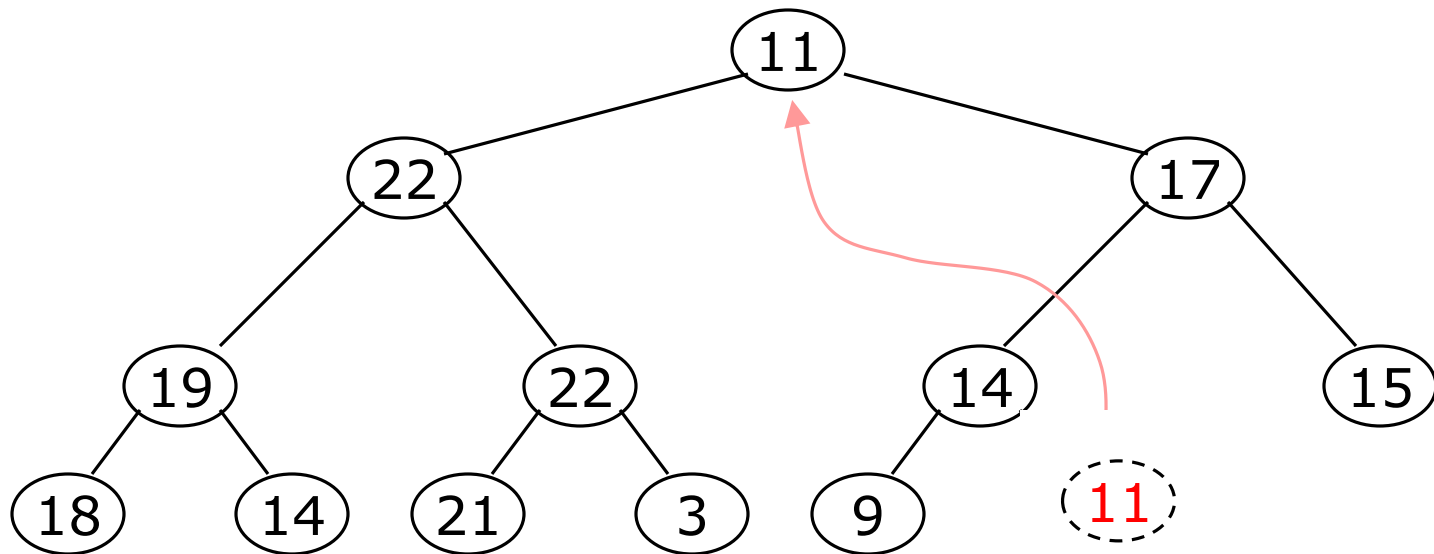
- Here's a sample binary tree after it has been heapified



- 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

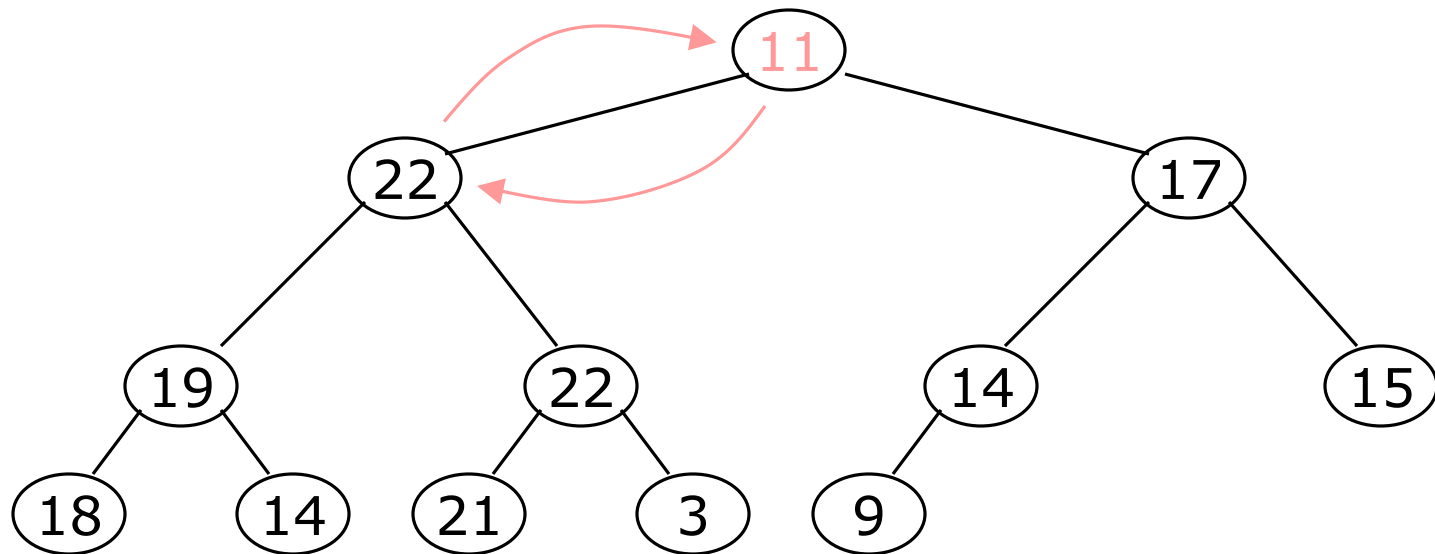
- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- 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. In our case it is node with value **11**

The **reHeap** method I

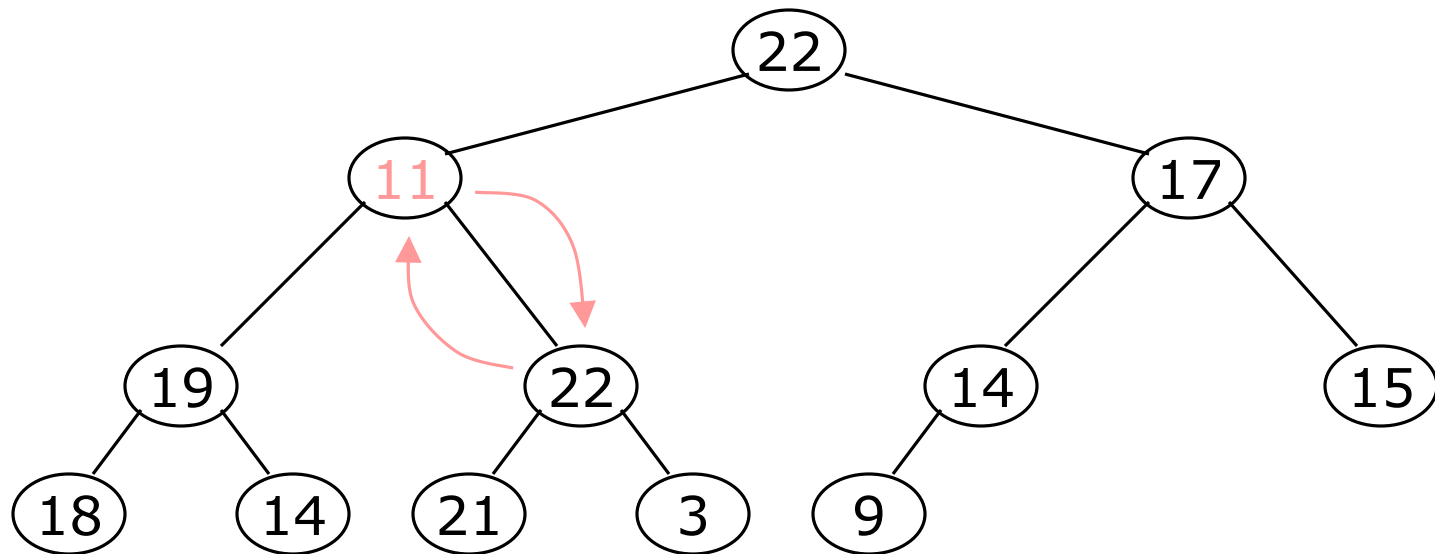
- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property



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

The **reHeap** method II

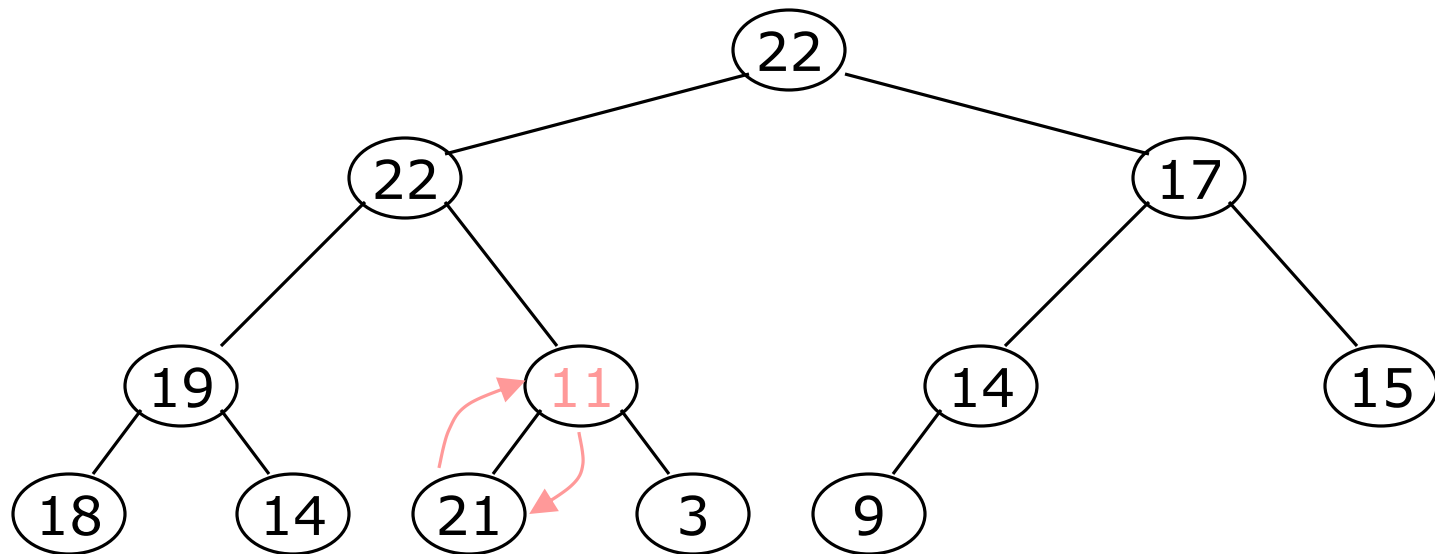
- Now the left child of the root (still the number **11**) lacks the heap property



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

The **reHeap** method III

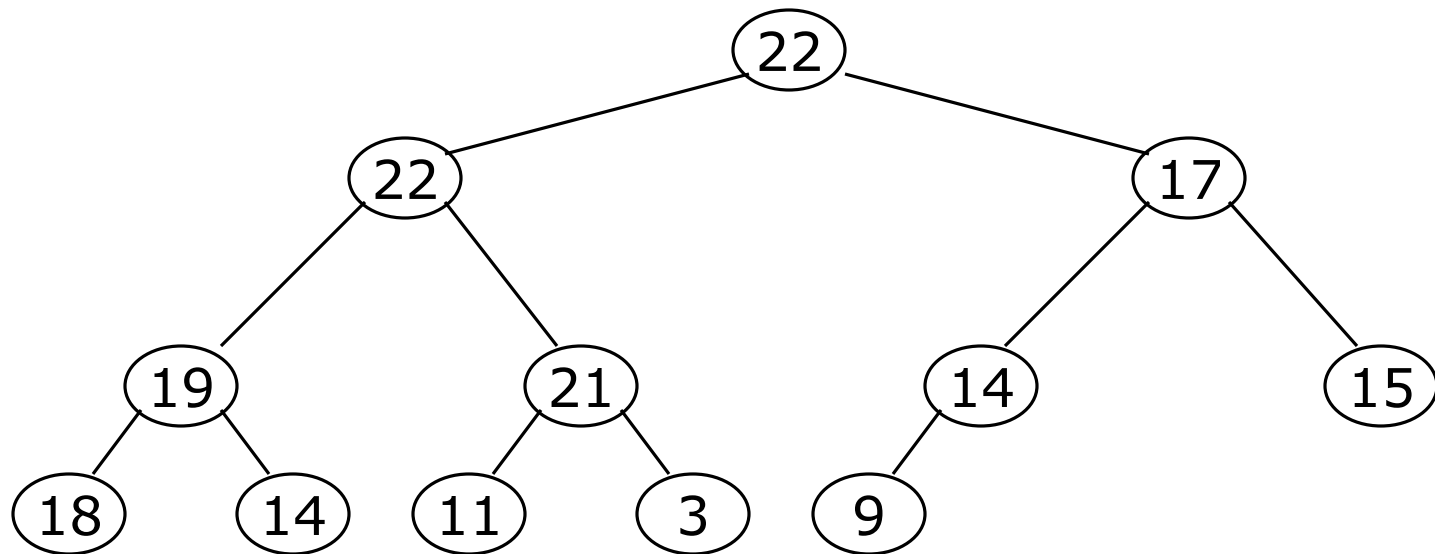
- Now the right child of the left child of the root (still the number **11**) lacks the heap property:



- We can **siftUp()** 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



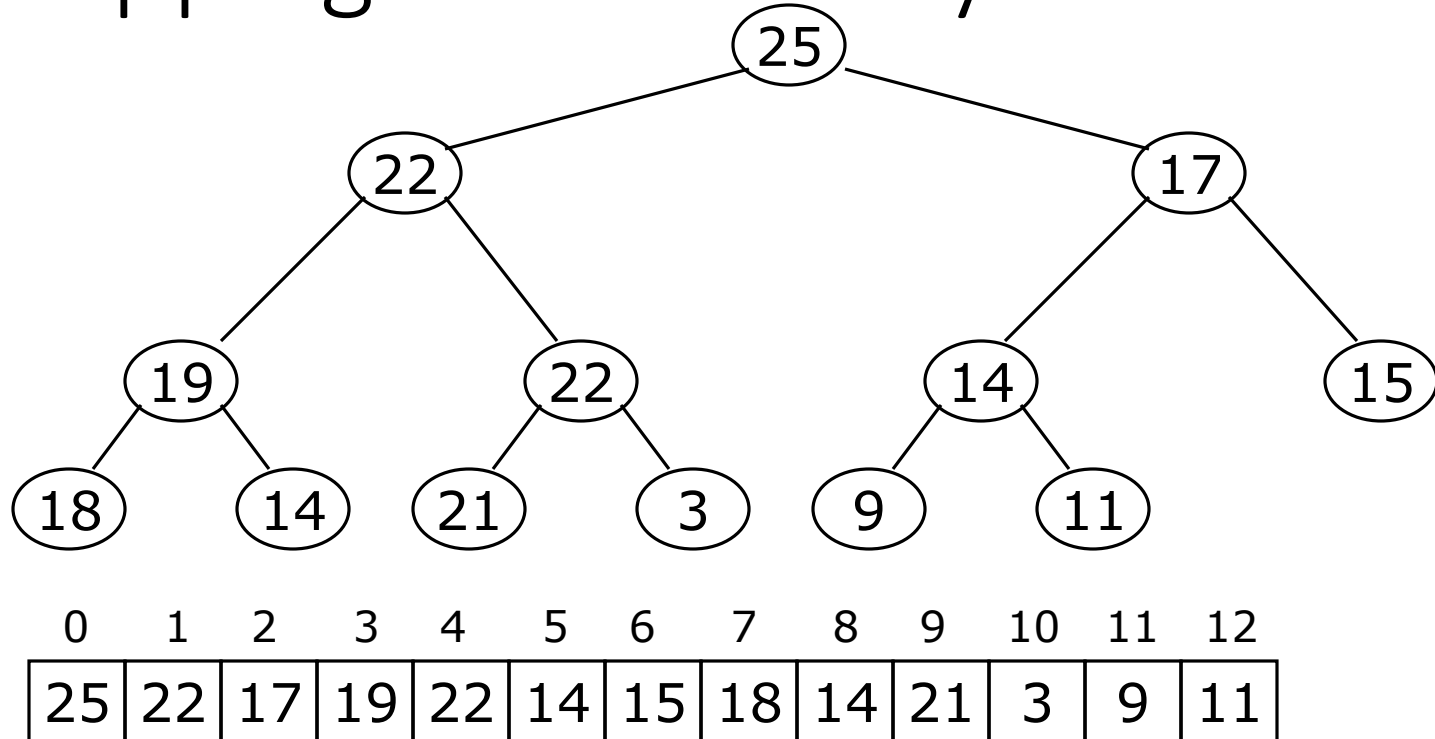
- Once again, the largest (or *a* largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest

Sorting

- What do heaps have to do with sorting an array?
- Here's the neat part:
 - Because the binary tree is *balanced* and *left justified*, it can be represented as an array
 - All our operations on binary trees can be represented as operations on *arrays*
 - To sort:

```
heapify the array;  
while the array isn't empty {  
    remove and replace the root;  
    reheap the new root node;  
}
```

Mapping into an array

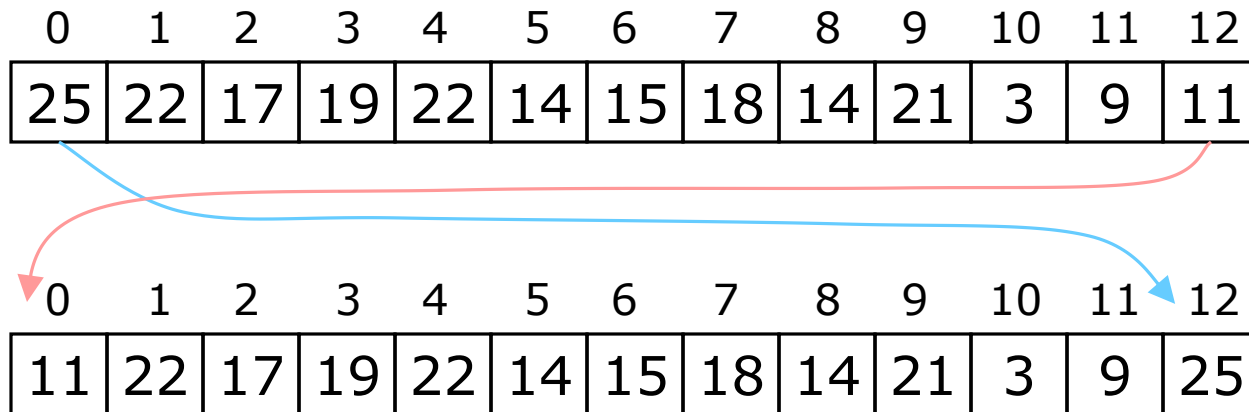


- Notice:

- The left child of index i is at index $2*i+1$
- The right child of index i is at index $2*i+2$
- Example: the children of node 3 (19) are 7 (18) and 8 (14)

Removing and replacing the root

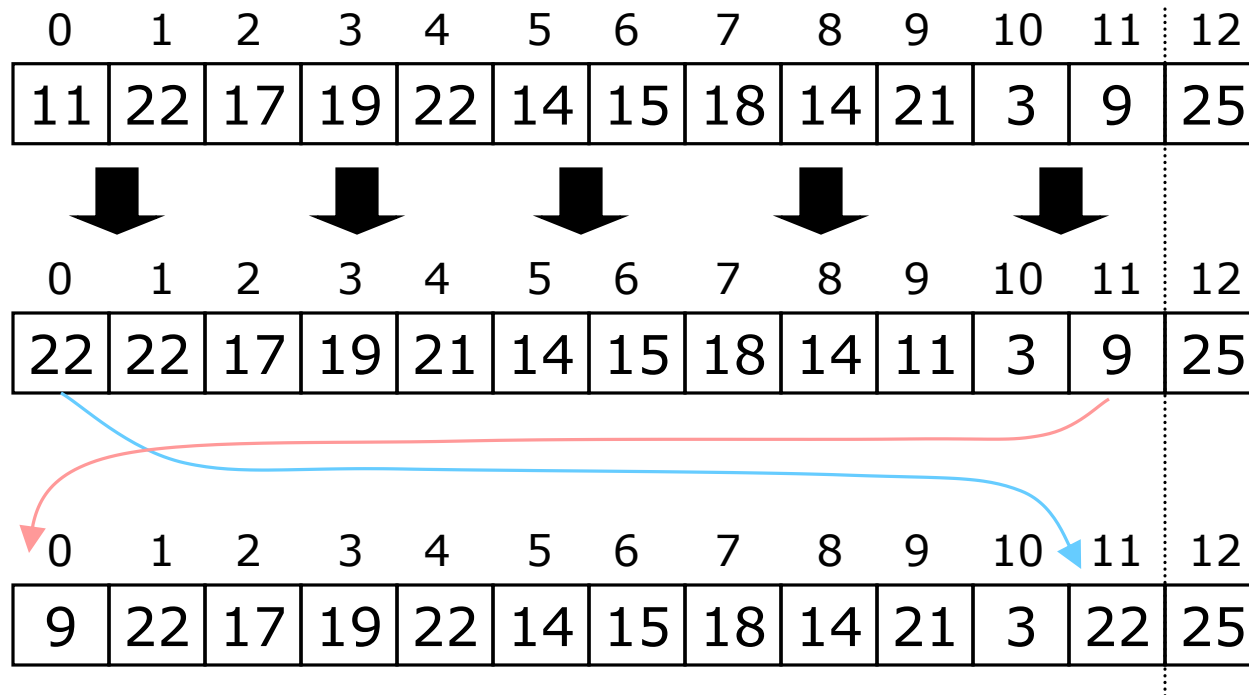
- The “root” is the first element in the array
- The “rightmost node at the deepest level” is the last element
- Swap them...



- ...And pretend that the last element in the array no longer exists—that is, the “last index” is **11** (9)

Reheap and repeat

- Reheap the root node (index 0, containing **11**)...



- ...And again, remove and replace the root node
- Remember, though, that the “last” array index is changed
- Repeat until the last becomes first, and the array is sorted!

Analysis I

- Here's how the algorithm starts:
 heapify the array;
- Heapifying the array: we add each of n nodes
 - Each node has to be sifted up, possibly as far as the root
 - Since the binary tree is perfectly balanced, sifting up a single node takes $O(\log n)$ time
 - Since we do this n times, heapifying takes $n * O(\log n)$ time, that is, $O(n \log n)$ time

Analysis II

- Here's the rest of the algorithm:

```
while the array isn't empty {  
    remove and replace the root;  
    reheap the new root node;  
}
```

- We do the while loop n times (actually, $n-1$ times), because we remove one of the n nodes each time
- Removing and replacing the root takes $O(1)$ time
- Therefore, the total time is n times however long it takes the `reheap` method

Analysis III

- To reheap the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
- The binary tree is perfectly balanced
- Therefore, this path is $O(\log n)$ long
 - And we only do $O(1)$ operations at each node
 - Therefore, reheaping takes $O(\log n)$ times
- Since we reheap inside a while loop that we do n times, the total time for the while loop is $n * O(\log n)$, or $O(n \log n)$

Analysis IV

- Here's the algorithm again:

```
heapify the array;  
while the array isn't empty {  
    remove and replace the root;  
    reheap the new root node;  
}
```

- We have seen that heapifying takes $O(n \log n)$ time
- The while loop takes $O(n \log n)$ time
- The total time is therefore $O(n \log n) + O(n \log n)$
- This is the same as $O(n \log n)$ time