11 - HEAPS

- **print jobs:** CSE lab printers constantly accept and complete jobs from all over the building. We want to print faculty jobs before staff before student jobs, and grad students before undergrad, etc.

- **ER scheduling:** Scheduling patients for treatment in the ER. A gunshot victim should be treated sooner than a guy with a cold, regardless of arrival time. How do we always choose the most urgent case when new patients continue to arrive?

- *key operations we want:*
  - ***add*** *an element (print job, patient, etc.)*
  - ***get/remove*** *the* ***most "important"*** *or "urgent" element*

Topics

- Priority Queue and its implementation
  - Heap implementation
  - Introduction to Heaps
    - Construction of a heap (Insertion of a node)
    - Removal of root node from a heap
- Heap and Heapsort Algorithm

4

Priority Queue

Implementations

Priority queue – (RECALL)

- A stack is first in, last out
- A queue is first in, first out
- A priority queue is *largest-first-out*
  - The "largest" element is the first one removed
    - (You could also define a *least-first-out* priority queue)
  - The definition of "largest" is up to the programmer (for example, you might define it by implementing Comparator or Comparable)
  - If there are several "largest" elements, the implementer must decide which to remove first
    - Remove any "largest" element (don't care which)
    - Remove the first one added

A priority queue ADT

- Here is one possible ADT:
  - PriorityQueue(): a constructor
  - void add(Comparable o): inserts o into the priority queue
  - Comparable removeLargest(): removes and returns the largest element
  - Comparable getLargest(): returns (but does not remove) the largest element
  - boolean isEmpty(): returns true if empty
  - int size(): returns the number of elements
  - void clear(): discards all elements

Evaluating implementations

- When we choose a data structure, it is important to look at usage patterns
  - If we load an array once and do thousands of searches on it, we want to make searching fast—so we would probably sort the array
  - If we load a huge array and expect to do only a few searches, we probably *don't* want to spend time sorting the array
- For almost all uses of a queue (including a priority queue), we eventually remove everything that we add
- Hence, when we analyze a priority queue, neither "add" nor "remove" is more important—we need to look at the timing for "add + remove"

Array implementations

- A priority queue could be implemented as an *unsorted array* (with a count of elements)
  - Adding an element would take O(1) time (why?)
  - Removing an element would take O(n) time (why?)
  - Hence, adding *and* removing an element takes O(n) time
  - This is an inefficient representation
- A priority queue could be implemented as a *sorted array in descending order* (again, with a count of elements)
  - Adding an element would take O(n) time (why?)
  - Removing an element would take O(1) time (why?)
  - Hence, adding *and* removing an element takes O(n) time
  - Again, this is inefficient

Linked list implementations

- A priority queue could be implemented as an *unsorted* linked list
  - Adding an element would take O(1) time (why?)
  - Removing an element would take O(n) time (why?)
- A priority queue could be implemented as a *sorted* linked list in a descending order
  - Adding an element would take O(n) time (why?)
  - Removing an element would take O(1) time (why?)
- As with array representations, adding *and* removing an element takes O(n) time
  - Again, these are inefficient implementations

Binary tree implementations

- A priority queue could be represented as a (not necessarily balanced) binary search tree
  - Insertion times would range from O(log n) to O(n) (why?)
  - Removal times would range from O(log n) to O(n) (why?)
- A priority queue could be represented as a *balanced* binary search tree
  - Insertion and removal could destroy the balance
  - We need an algorithm to *rebalance* the binary tree
  - Good rebalancing algorithms require only O(log n) time, but are complicated

Heap implementation – Solution ☺

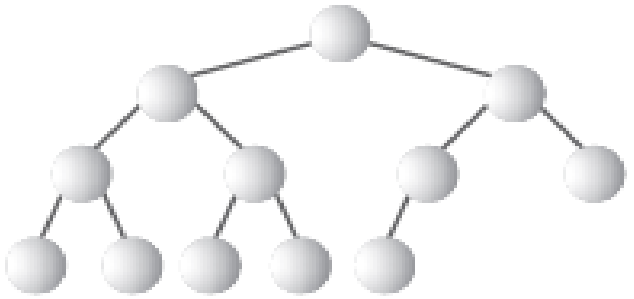
- A priority queue can be implemented as a heap
  - It is a kind of tree. Removal and insertion are both in O(logN) time.
    - It's the method of choice for implementing priority queues where speed is important and there will be many insertions.

12

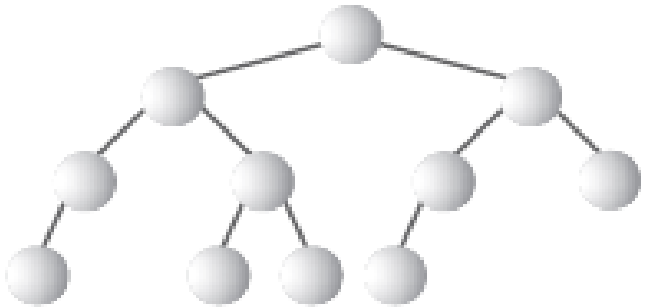
Introduction to Heap

Introduction to Heaps

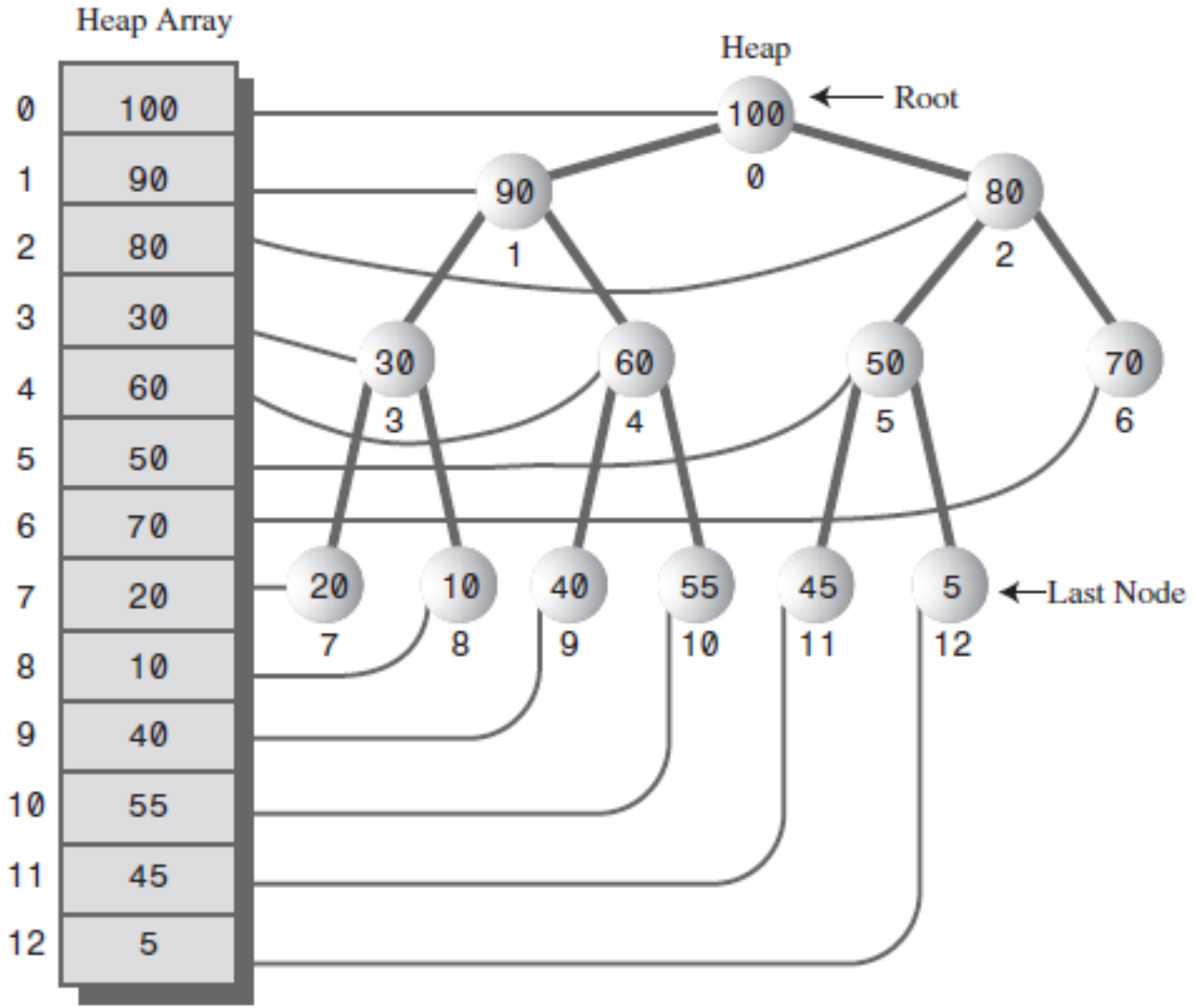
- A heap is a binary tree, that is:
  - Complete: completely filled in, reading from left to right across each row, although the last row need not be full.
  - Implemented as an array.
  - Each node in a heap satisfies the *heap condition*, which states that every node's key is larger than (or equal to) the keys of its children.



a) Complete



b) Incomplete



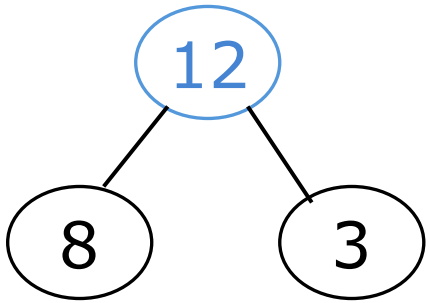
A heap and its underlying array.

Array implementation of a Heap

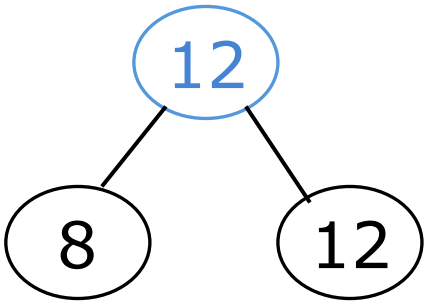
- The fact that a heap is a complete binary tree implies that there are no "holes" in the array used to represent it.
- Every cell is filled, from 0 to N-1. (N is 13 in Figure of previous slide)
- We'll assume that the maximum key is in the root.
- A priority queue based on such a heap is a *descending-priority* queue.

The heap property

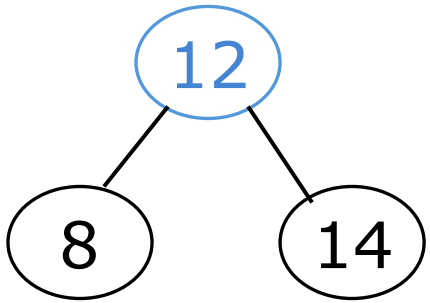
- A node has the heap property if the value in the node is greater than or equal to the values in its children



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

How to restore Heap property of a node?

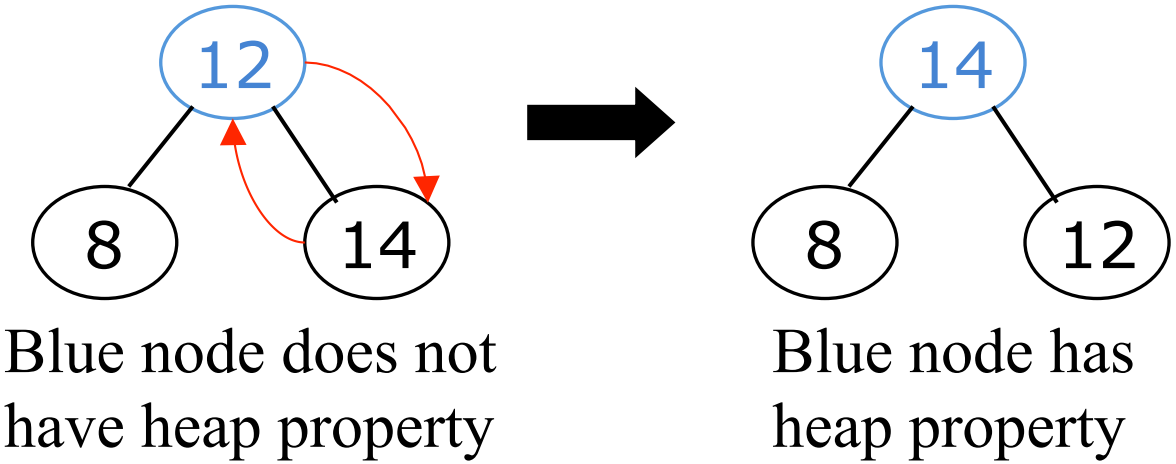
- Trickle Down or Trickle Up !!!!

Trickle Down/Trickle Up

- To *trickle* (the terms *bubble* or *percolate* are also used) a node up or down means to move it along a path step by step, either top-down or bottom-up, swapping it with the node ahead of it, checking at each step to see whether it's in its proper position.

Trickle Down/Trickle 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



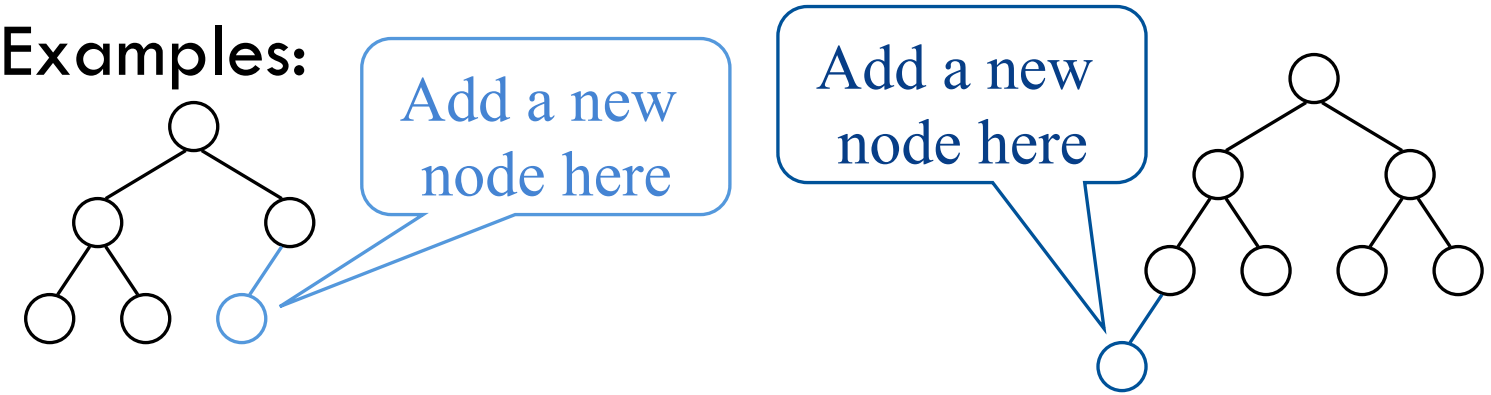
- This is called trickle down
- Notice that the child may have *lost* the heap property

Heap Operations

- We will learn how to:
  - Construct a heap from an unordered array.
    - Insert a node in a heap (O (log N))
      - Insertion in a priority queue
  - Remove root from a heap (i.e. remove largest element – O(log N))
    - Removal from a priority queue
- Finally we will see how to use these ideas to sort an array, implementing *heapsort* algorithm
- (See Heap WorkShop applet)

Constructing a heap I

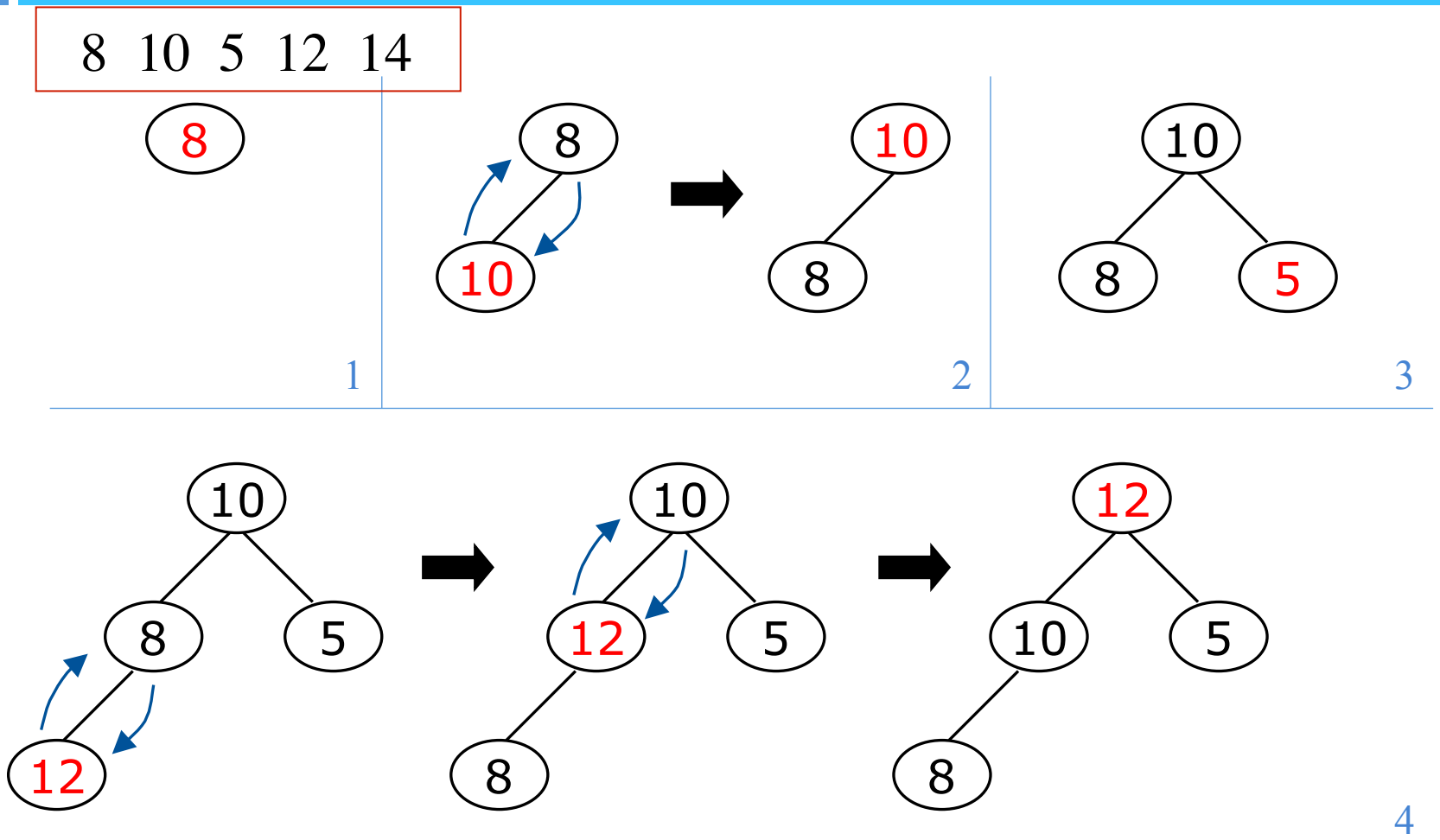
- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:



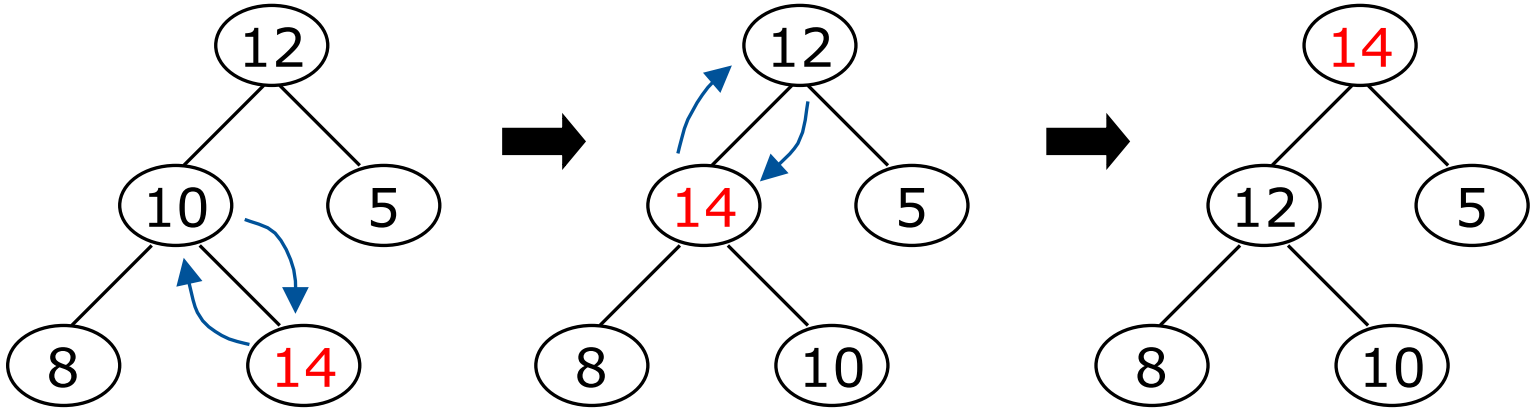
Constructing a heap II

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we trickle up (opposite of trickle down)
- But each time we trickle up, the value of the top node may increase, and this may destroy the heap property of *its* parent node
- We repeat the trickle 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



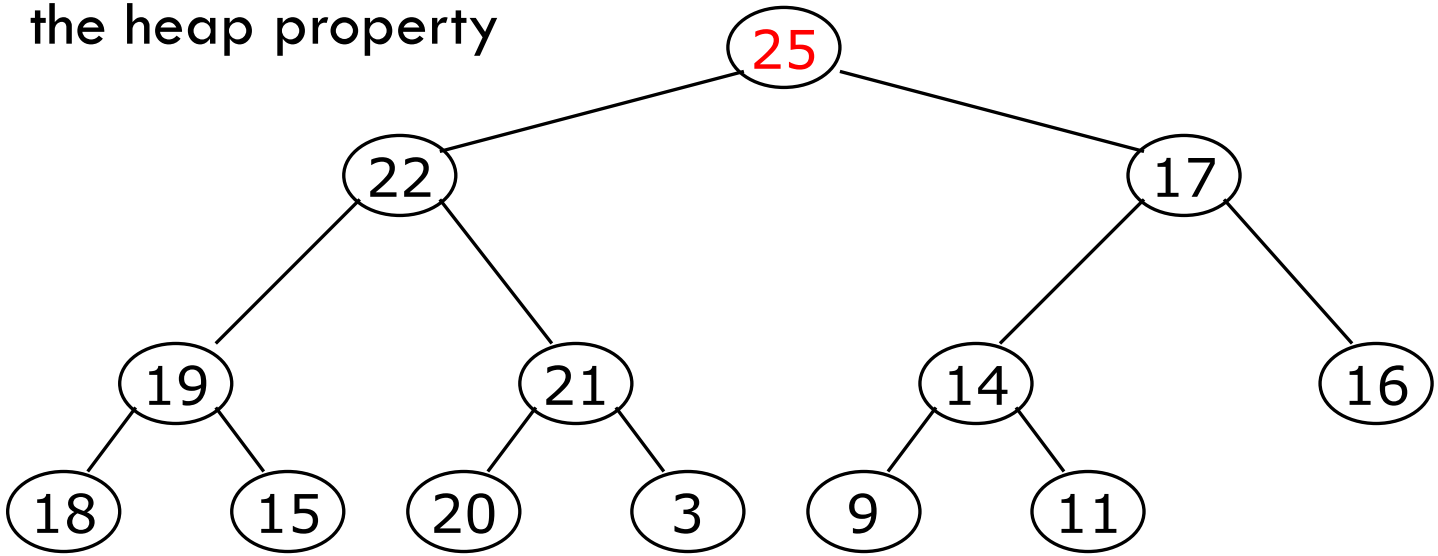
Other children are not affected



- 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

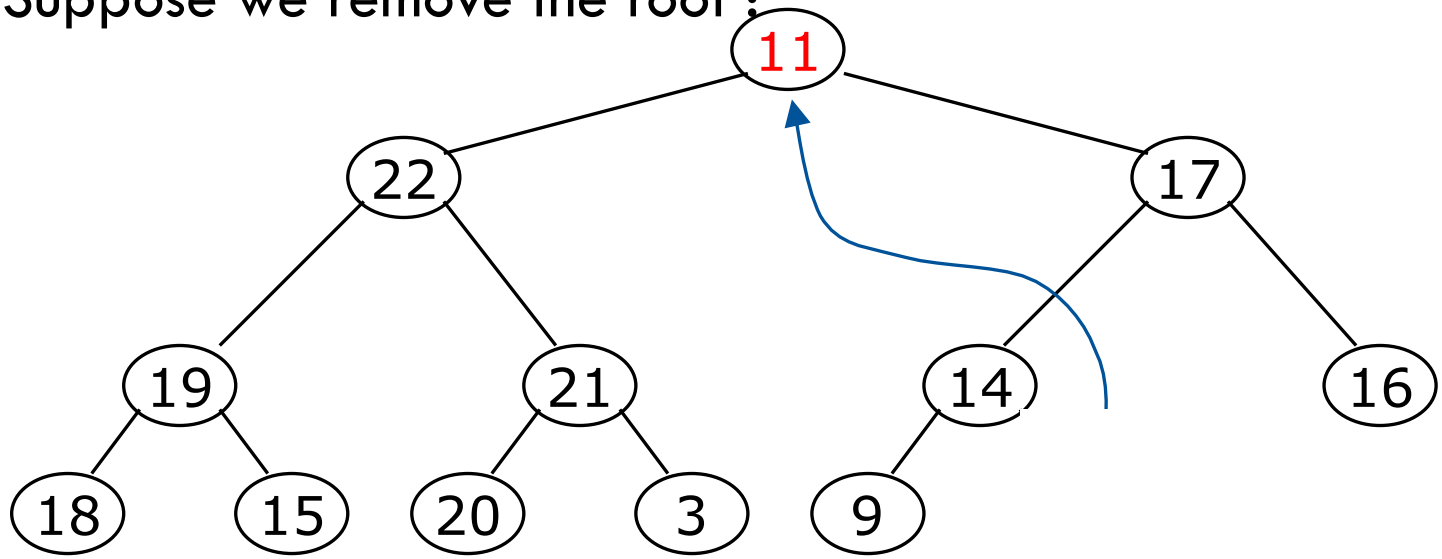
- Here's a sample heap (i.e. a binary tree whose nodes satisfy the heap property



- Notice that heap does *not* mean sorted, but the root always has the maximum value
- This binary tree is complete (and of course balanced) because it started out that way

Removing the root (animated)

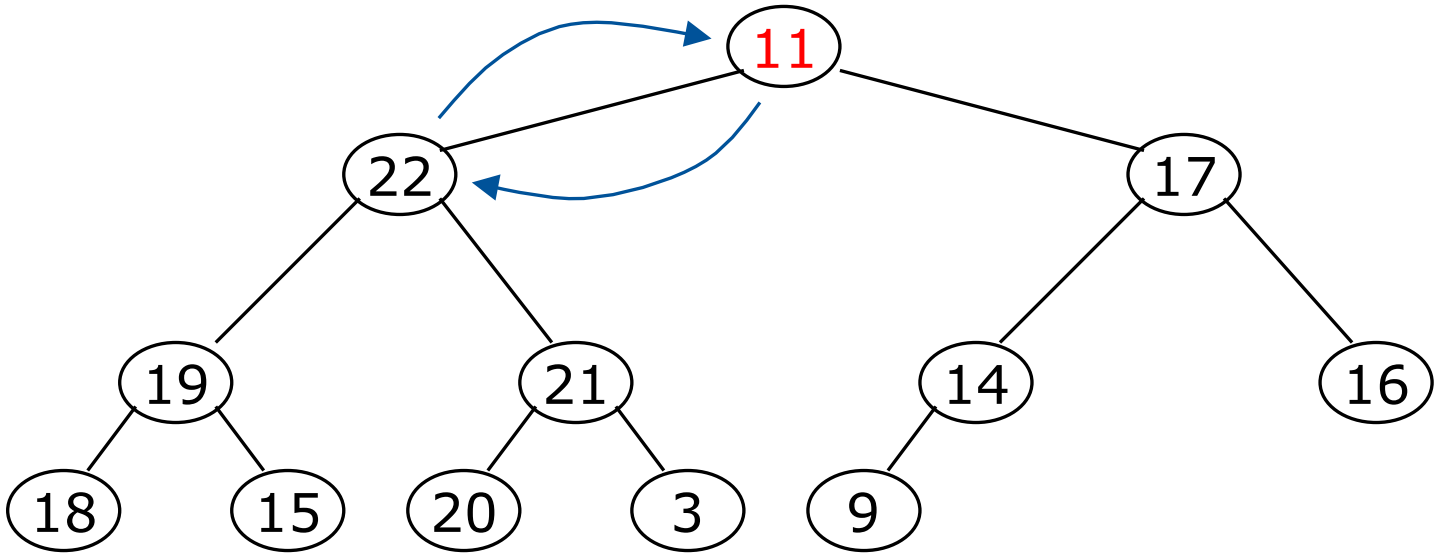
- Notice that the largest number is now in the root
- Suppose we remove the root :



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

The reHeap method I

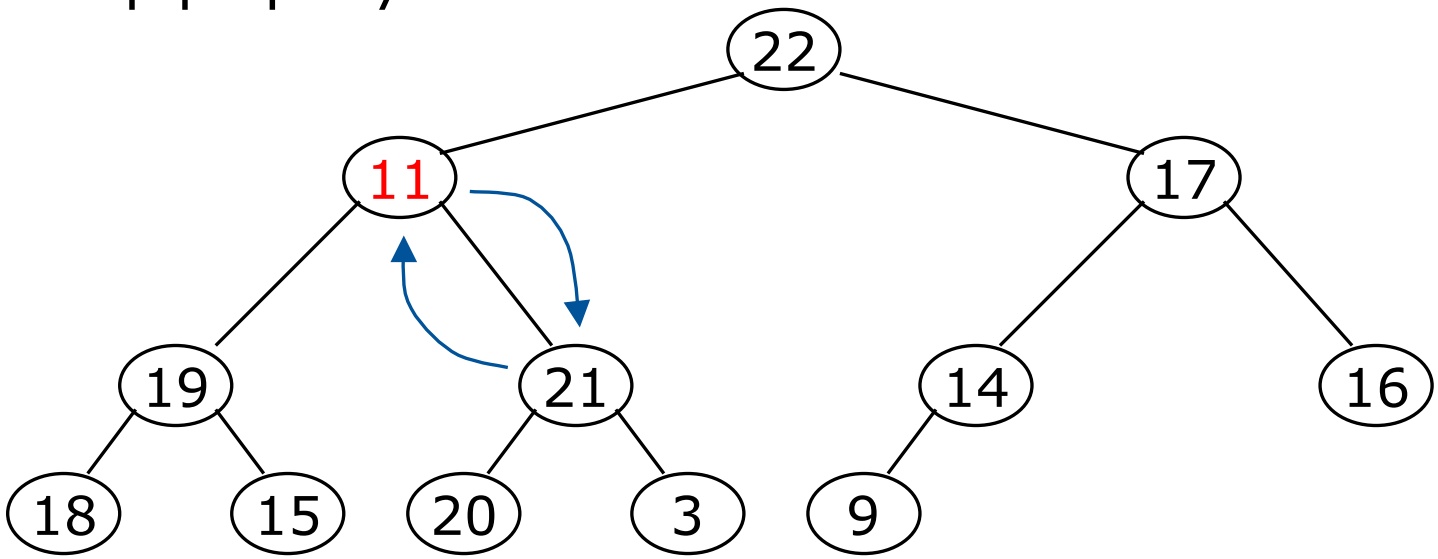
- Our tree is complete, but no longer a heap
- However, *only the root* lacks the heap property



- We can trickle down the root, by swapping with the largest child.
- After doing this, one and only one of its children may have lost the heap property

The reHeap method II

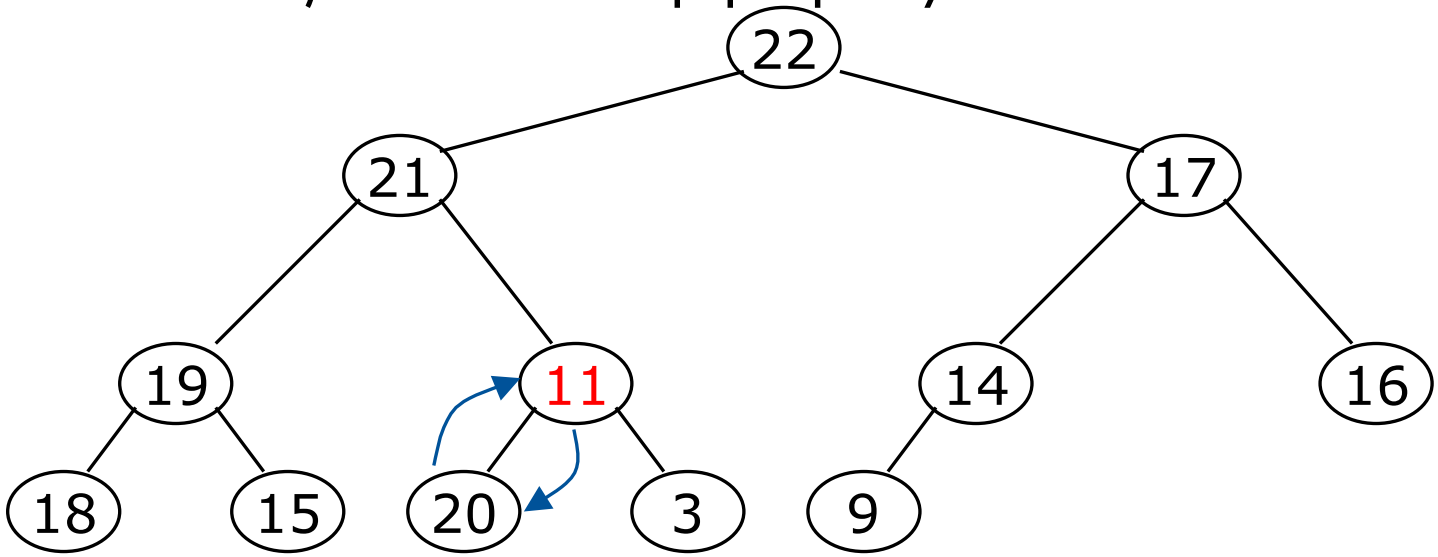
- Now the left child of the root (still the number 11) lacks the heap property



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

The reHeap method III

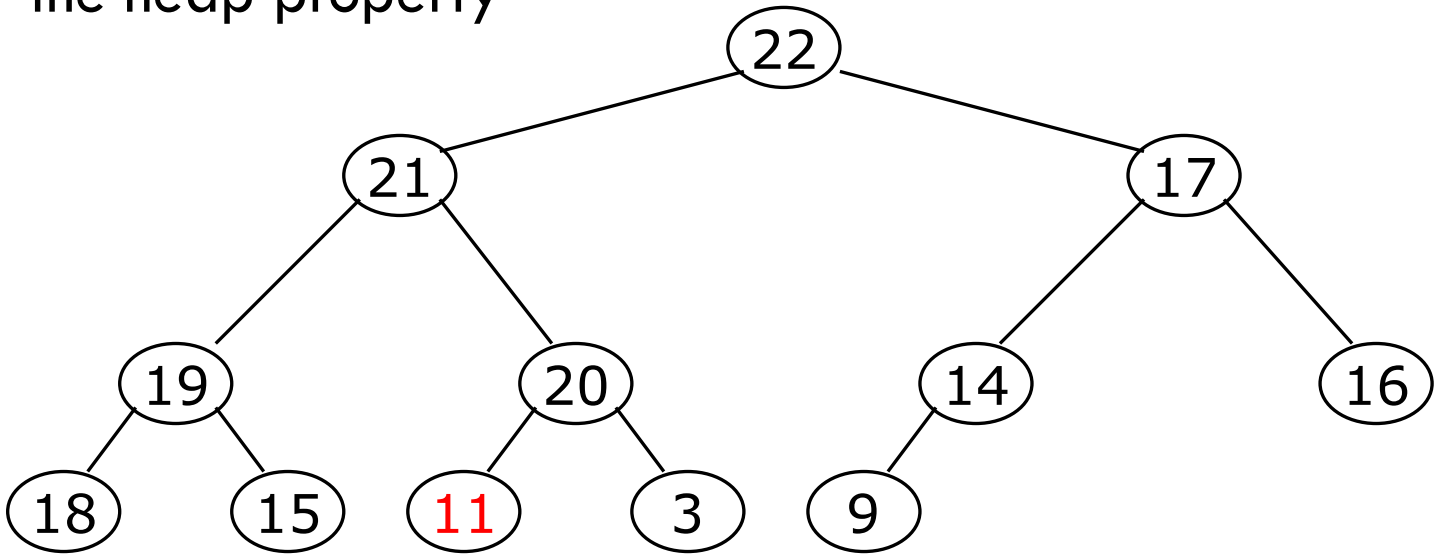
- Now the right child of the left child of the root (still the number 11) lacks the heap property:



- We can trickle down 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

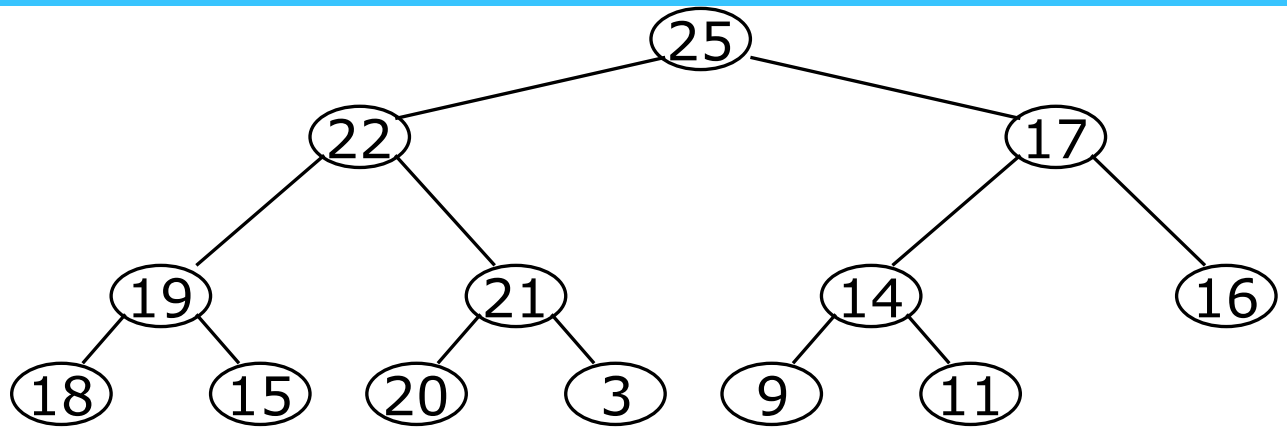
The reHeap method IV

- 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

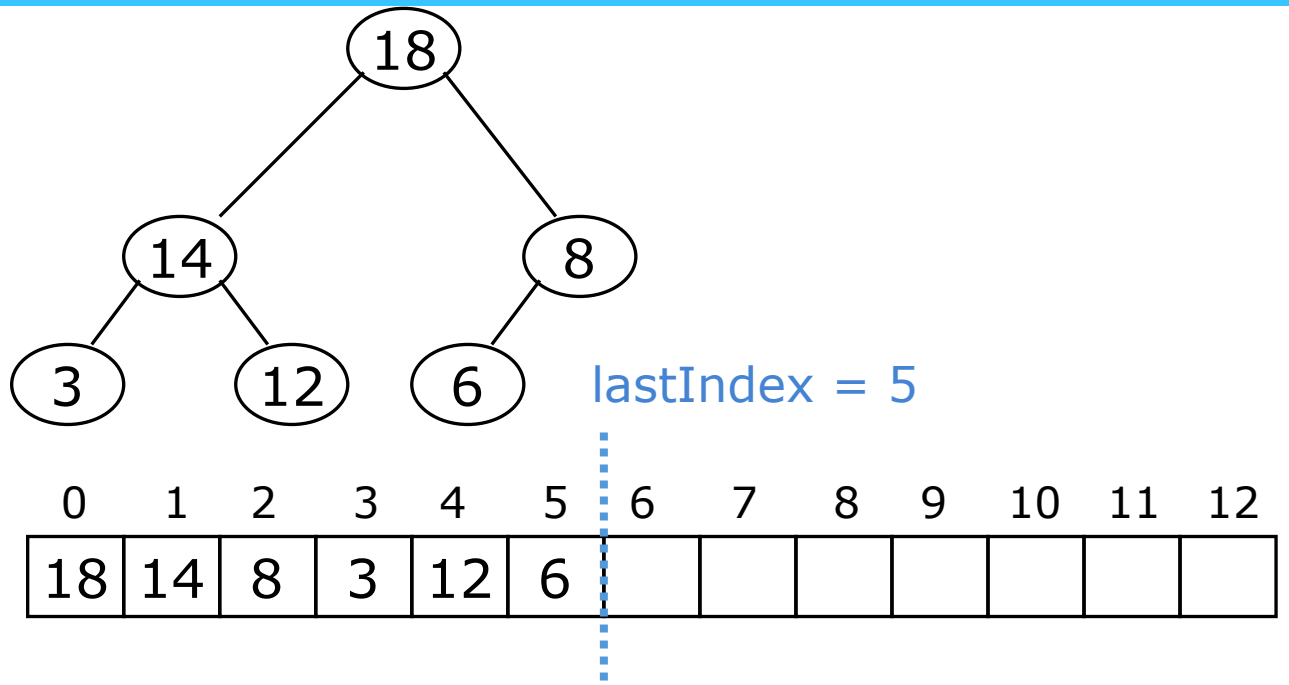
Array representation of a Heap



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	21	14	16	18	15	20	3	9	11

- Notice:
  - The left child of index i is at index $2*i+1$
  - he right child of index i is at index $2*i+2$
  - Parent of node k is $(k - 1)/2$
    - Unless $k == 0$
  - Example: the children of node 3 (19) are 7 (18) and 8 (15), and parent of node 9 (20) is 4 (21)

Heap representation of a Priority Queue



Using the heap as a Priority Queue

- See Listing 12.1, heap.java, page 592.
- To add an element:
  - Increase lastIndex and put the new value there
  - Reheap the newly added node
    - This is called trickle up (or up-heap bubbling or percolating up)
    - Trickle up requires O(log n) time
- To remove an element:
  - Remove the element at location 0
  - Move the element at location lastIndex to location 0, and decrement lastIndex
  - Reheap the new root node (the one now at location 0)
    - This is called trickle down (or down-heap bubbling or percolating down)
    - Trickle down requires O(log n) time
- Thus, it requires O(log n) time to add *and* remove an element

Comments

- A priority queue is a data structure that is designed to return elements in order of priority
- Efficiency is usually measured as the *sum* of the time it takes to add and to remove an element
  - Simple implementations take O(n) time
  - Heap implementations take O(log n) time
  - Balanced binary tree implementations take O(log n) time
  - Binary tree implementations, without regard to balance, can take O(n) (linear) time
- Thus, for any sort of heavy-duty use, heap or balanced binary tree implementations are better

Heap and Heapsort

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
- Heapsort is a *really cool* algorithm!

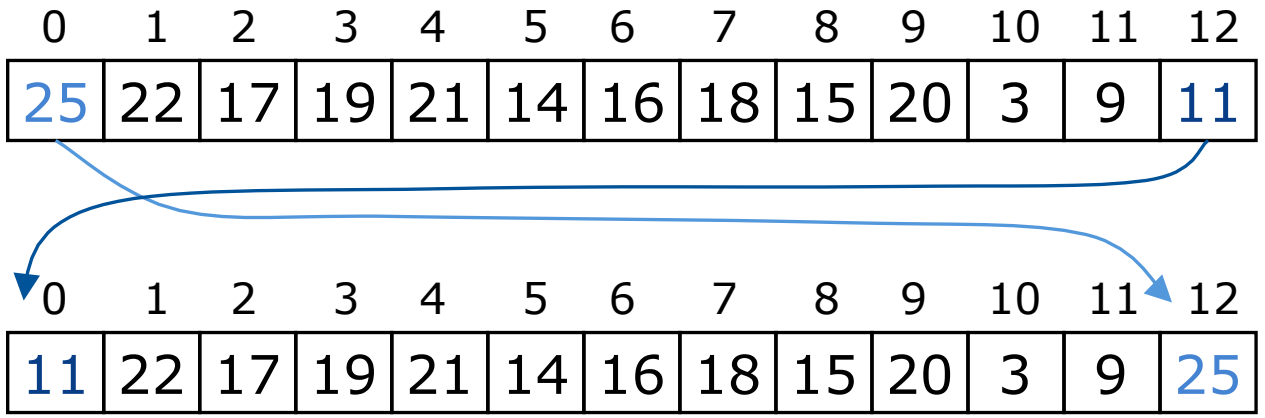
Sorting

- What do heaps have to do with sorting an array?
- Here's the neat part:
  - Because the binary tree is *balanced* and *left justified (i.e. Complete)*, it can be represented as an array
    - ***Please Note that:*** This representation works well ***only*** with ***balanced, left-justified*** binary trees
  - 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;
}
```

Removing and replacing the root

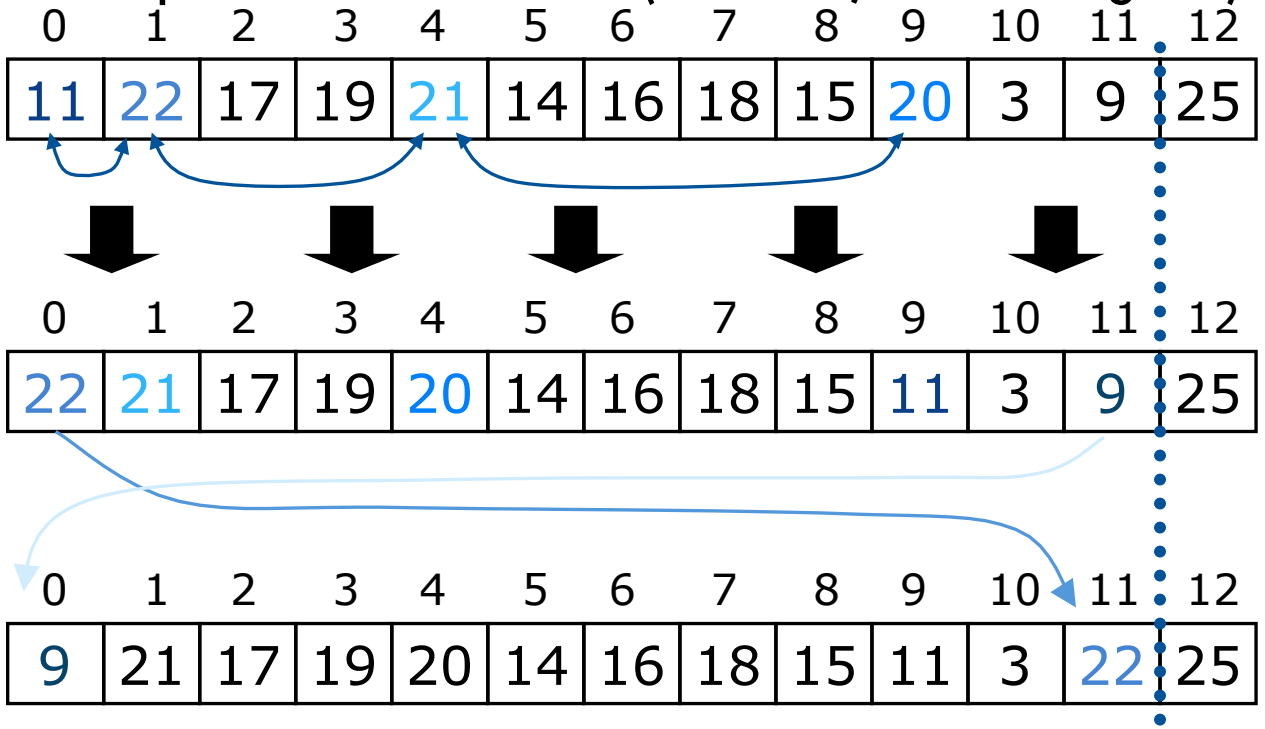
- The "root" is the first element in the array
- The "rightmost node at the deepest level" is the last element
- Swap them...



- ...And pretend that the last element in the array no longer exists —that is, the "last index" is 11 (containing the value 9)

Reheap and repeat

- Reheap of 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 trickled up, possibly as far as the root
    - Since the binary tree is perfectly balanced, trickling 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
- Listing 12.2, HeapSort.java, page 605

The End