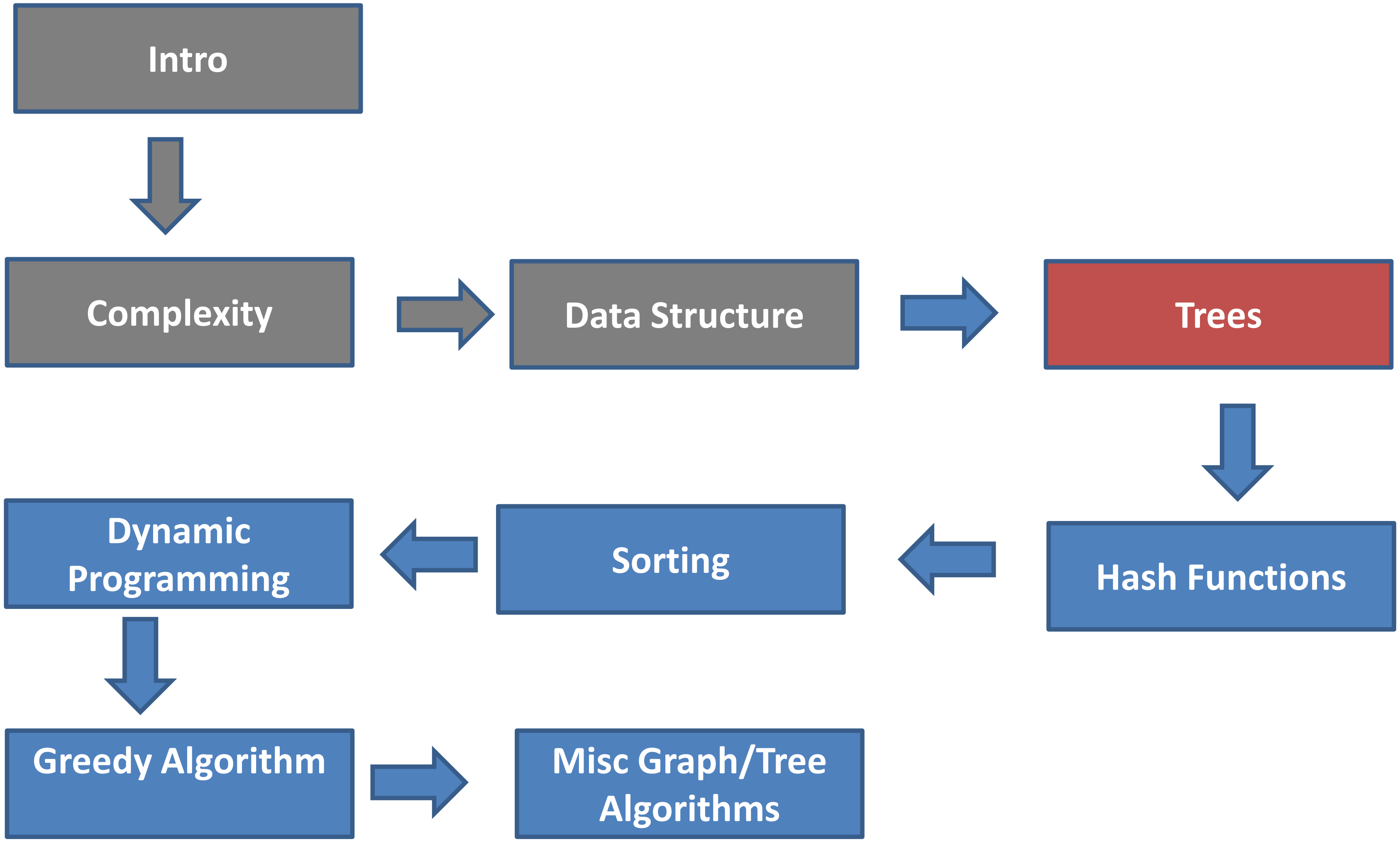
An Introduction to Algorithms

By

Hossein Rahmani

h_rahmani@iust.ac.ir
http://webpages.iust.ac.ir/h_rahmani/


Intro
Complexity
Data Structure
Trees
Hash Functions
Sorting
Dynamic Programming
Greedy Algorithm
Misc Graph/Tree Algorithms

Heaps

A **heap** is a certain kind of complete binary tree.

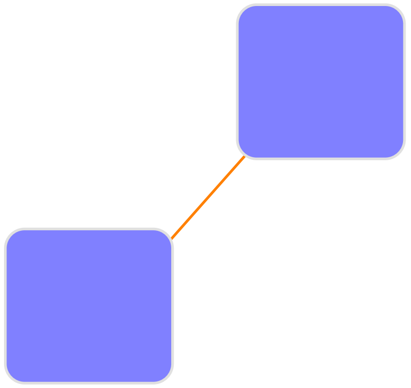
Heaps

A **heap** is a certain kind of complete binary tree.

When a complete binary tree is built, its first node must be the root.

Heaps

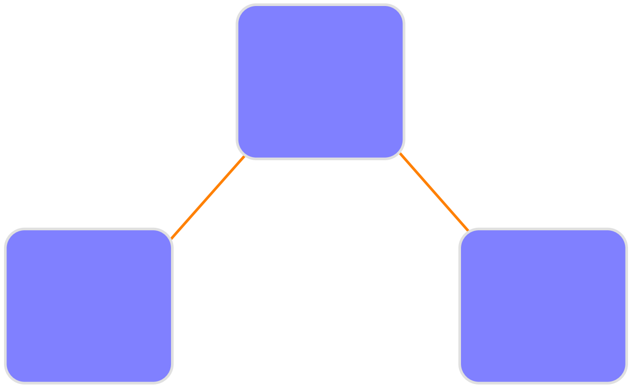
Complete binary tree.



The second node is always the left child of the root.

Heaps

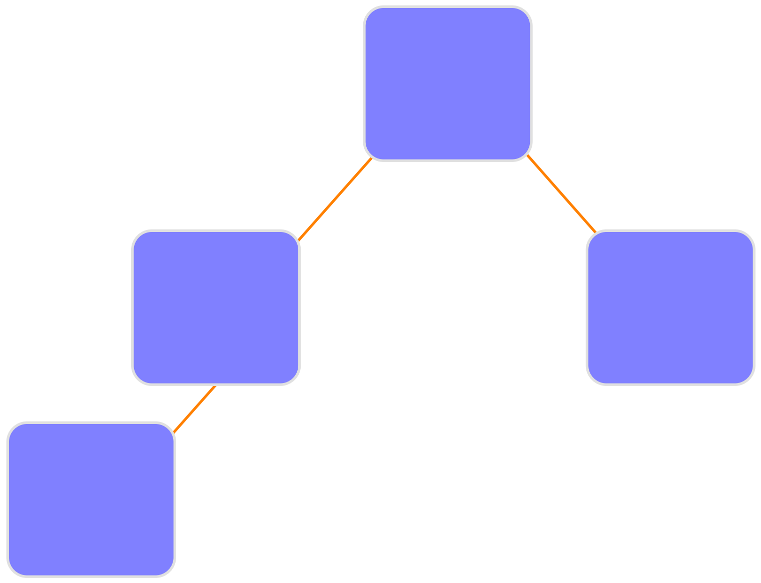
Complete binary tree.



The third node is always the right child of the root.

Heaps

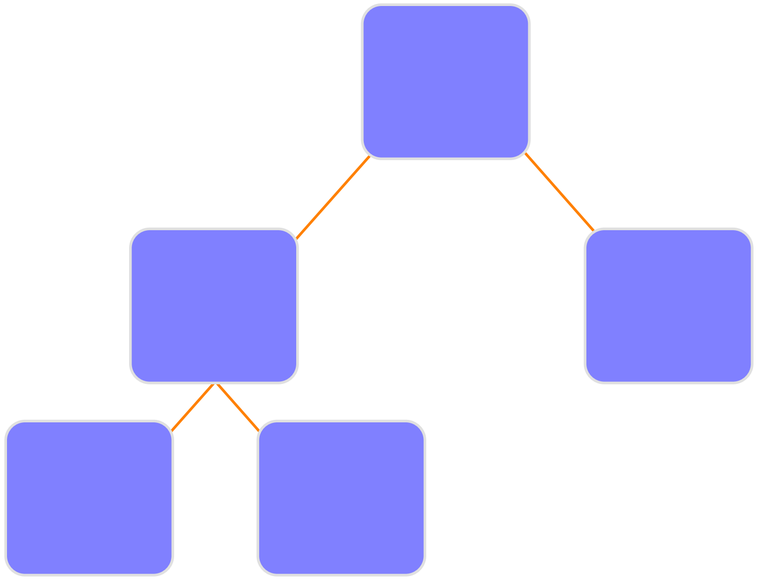
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

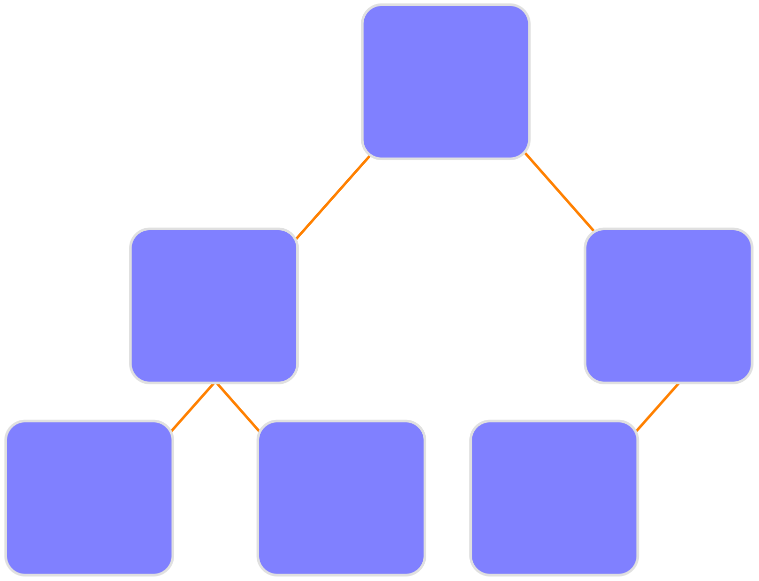
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

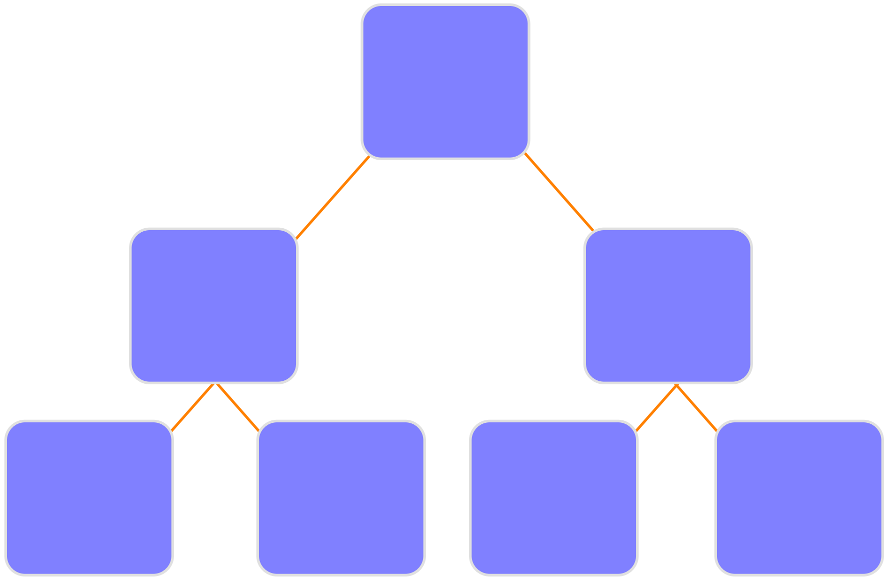
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

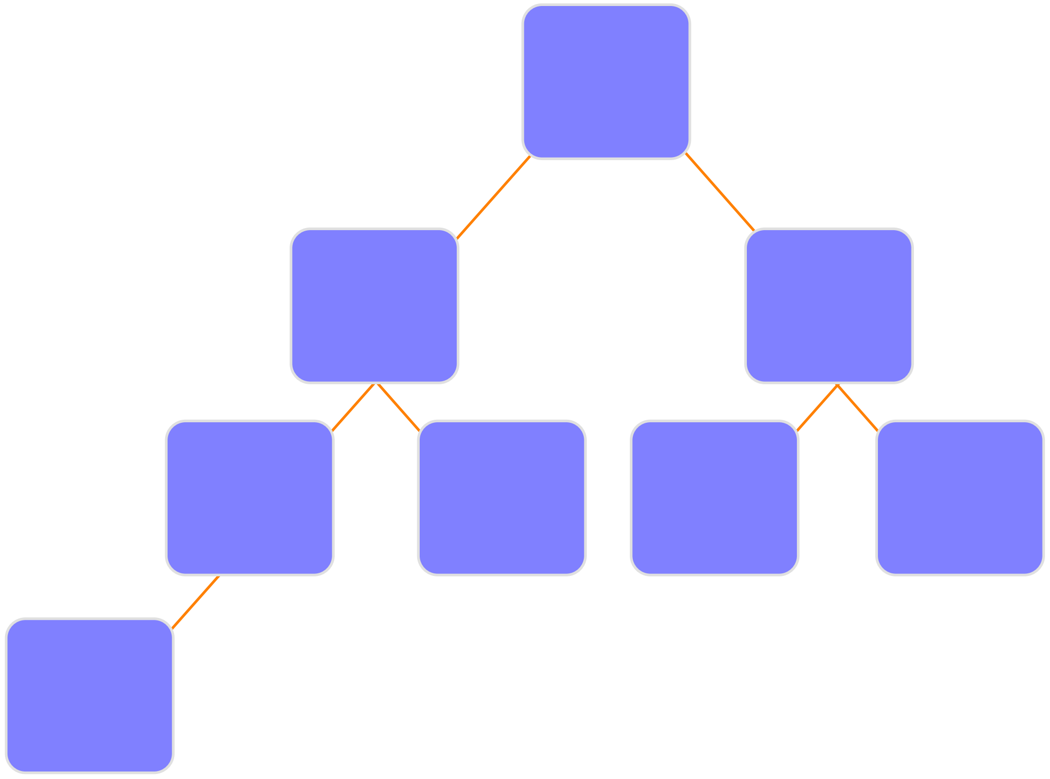
Complete binary tree.



The next nodes always fill the next level from left-to-right.

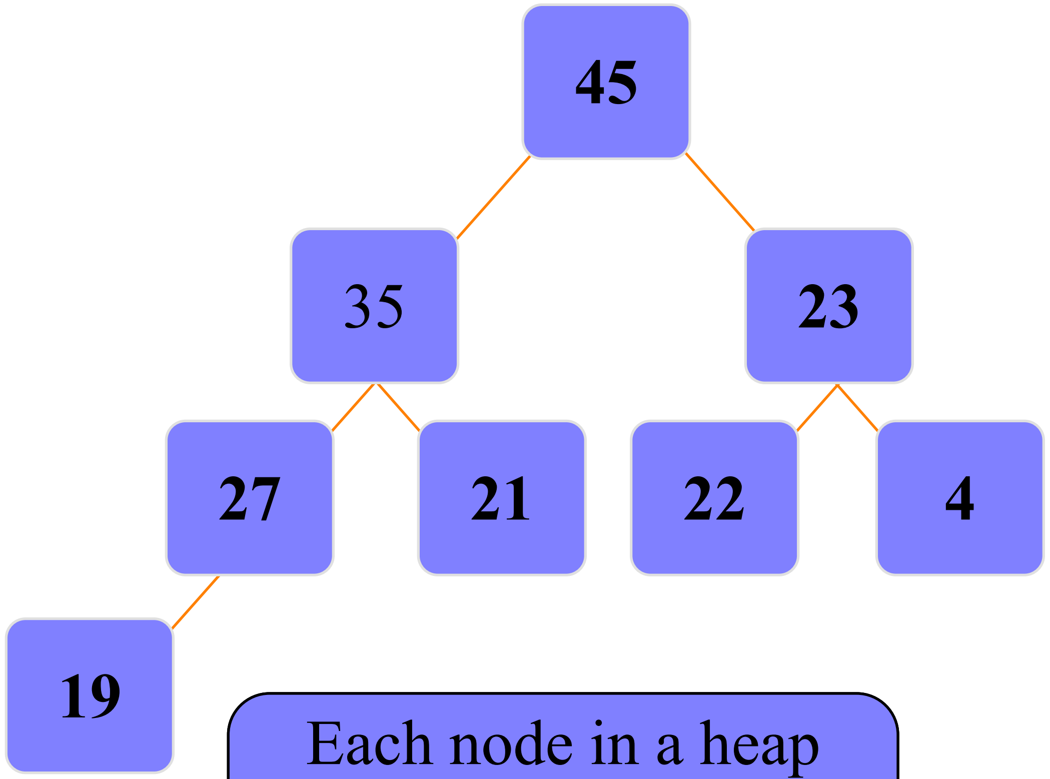
Heaps

Complete binary tree.



Heaps

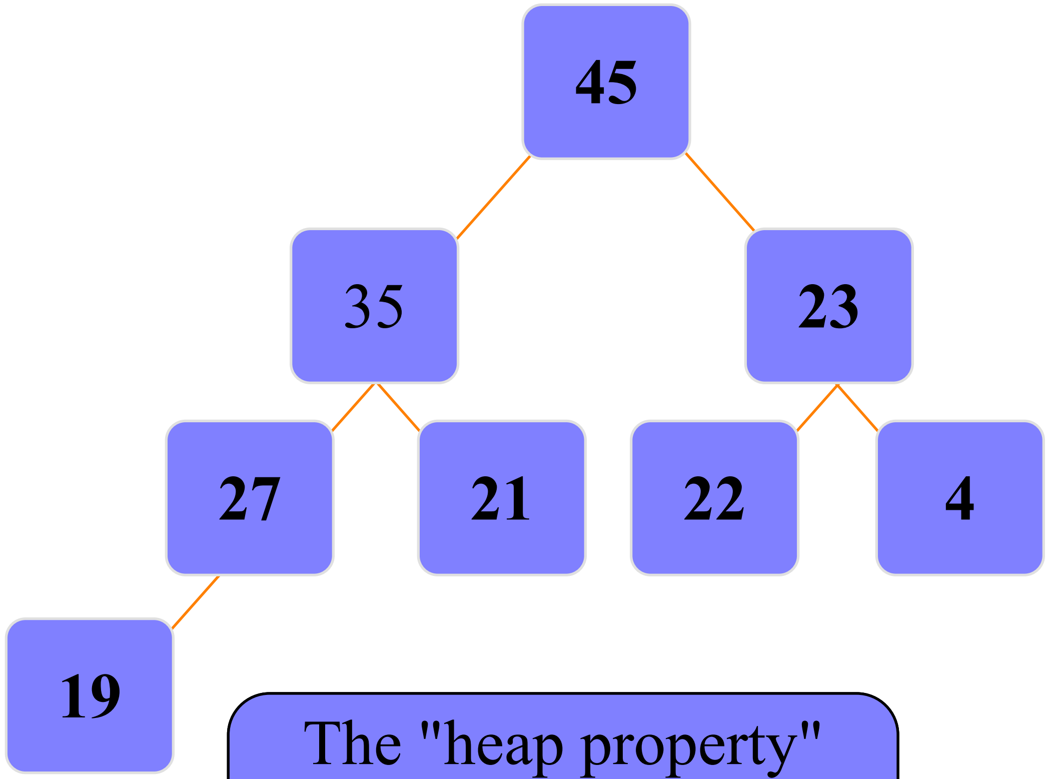
A heap is a **certain** kind of complete binary tree.



Each node in a heap contains a key that can be compared to other nodes' keys.

Heaps

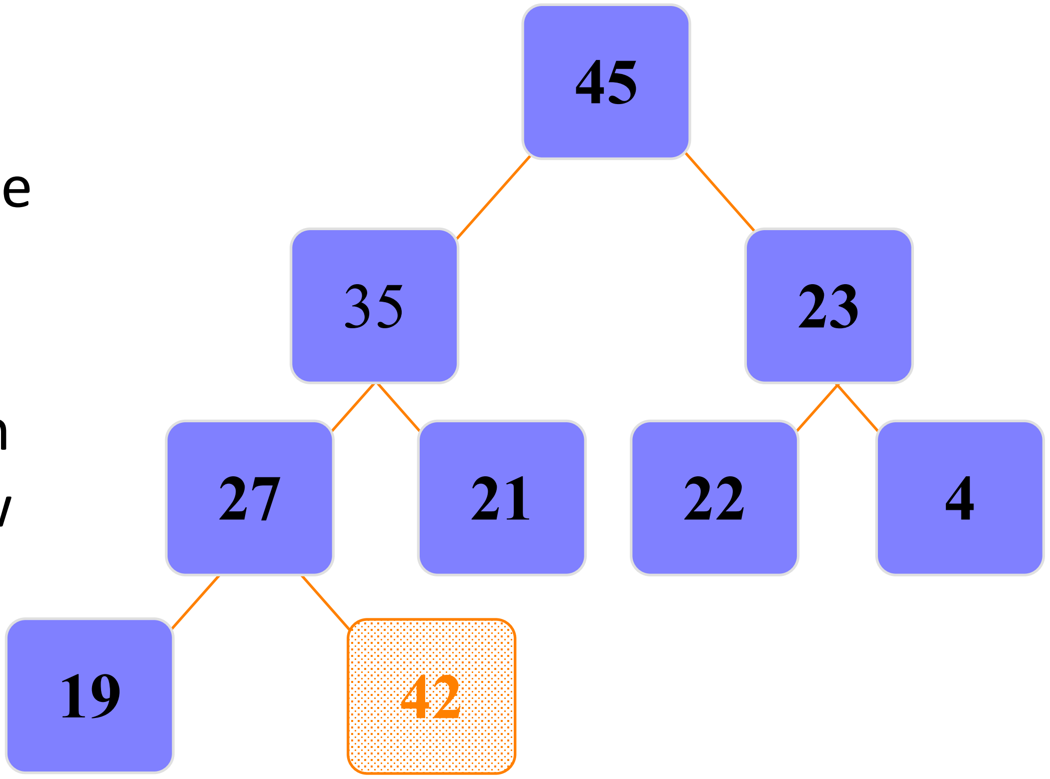
A heap is a **certain** kind of complete binary tree.



The "heap property" requires that each node's key is >= the keys of its children

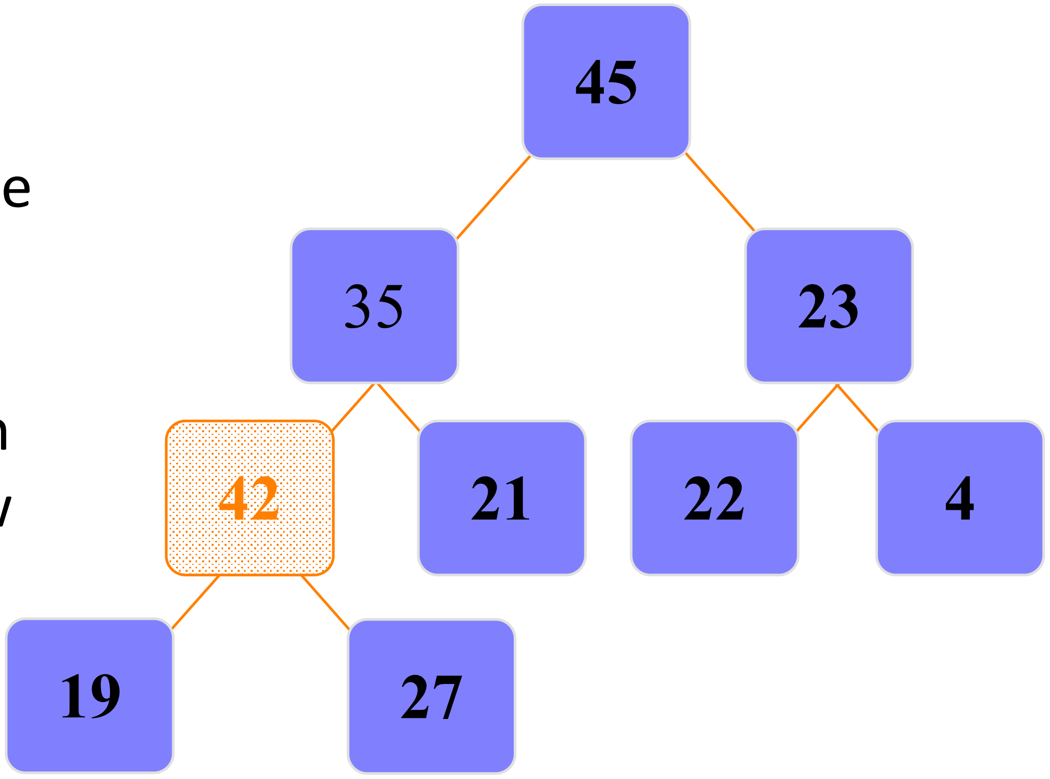
Adding a Node to a Heap

- ❑ Put the new node in the next available spot.
- ❑ Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



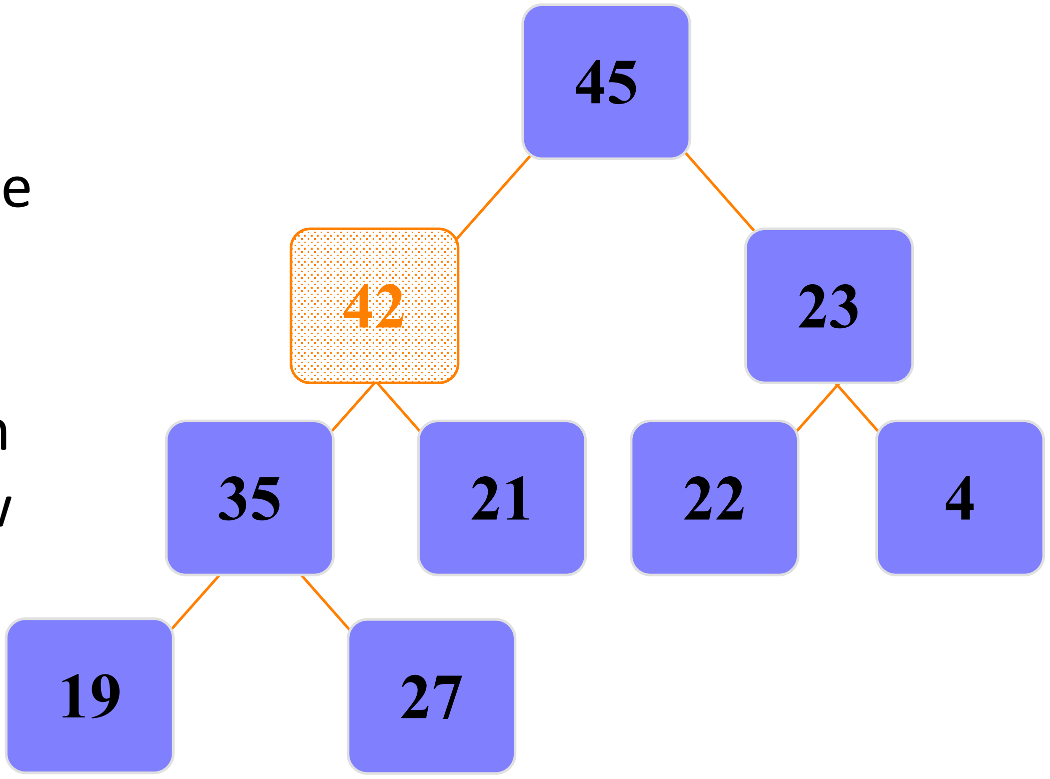
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



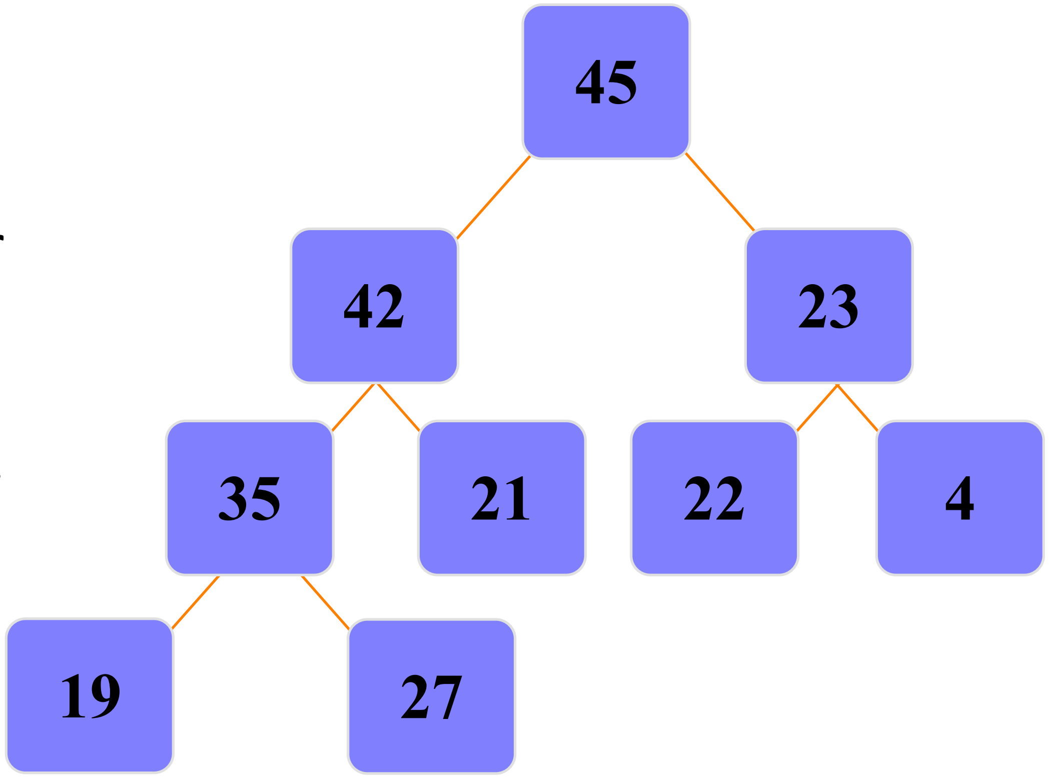
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



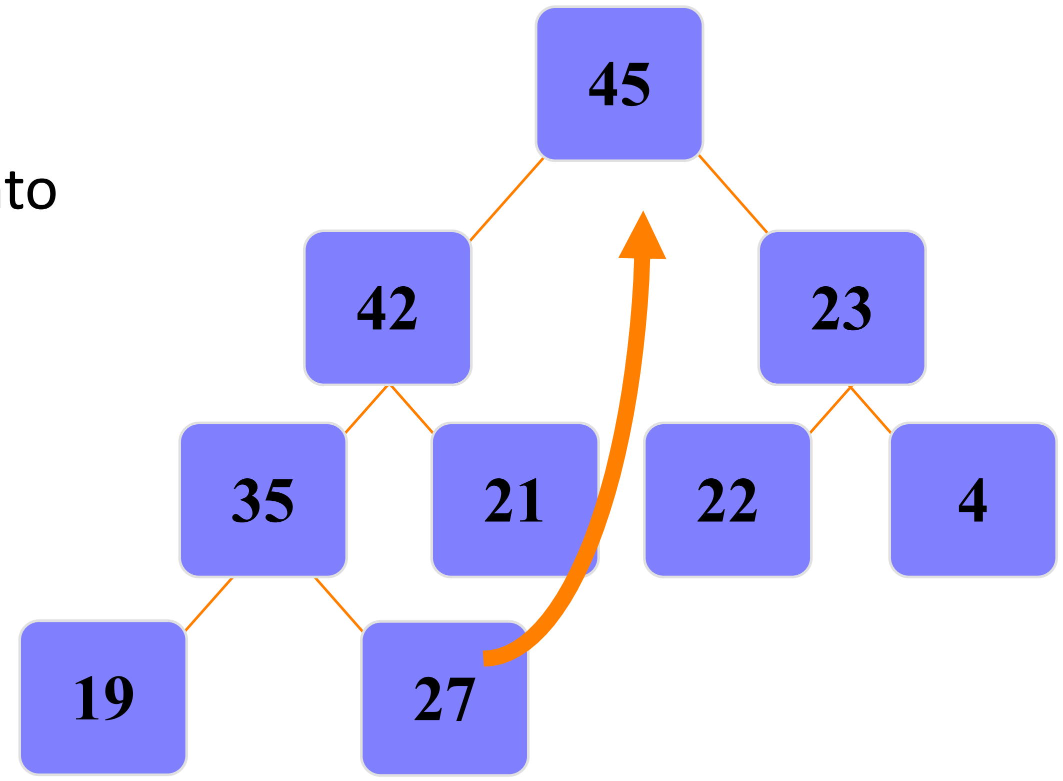
Adding a Node to a Heap

- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.



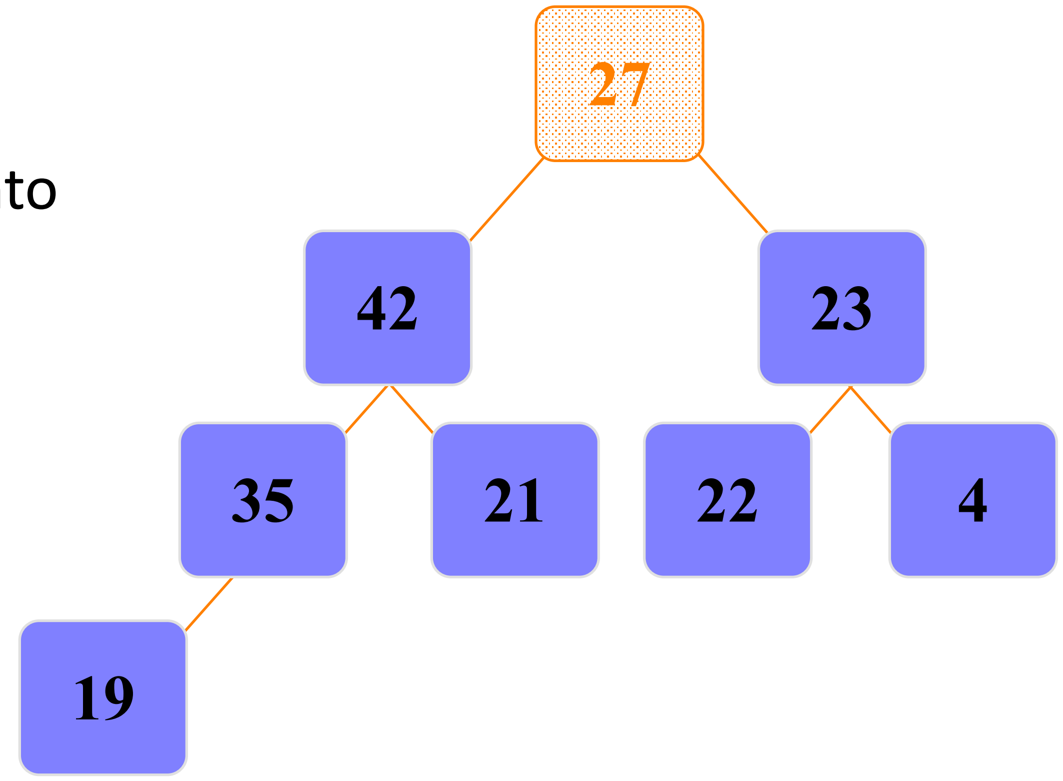
Removing the Top of a Heap

- Move the last node onto the root.



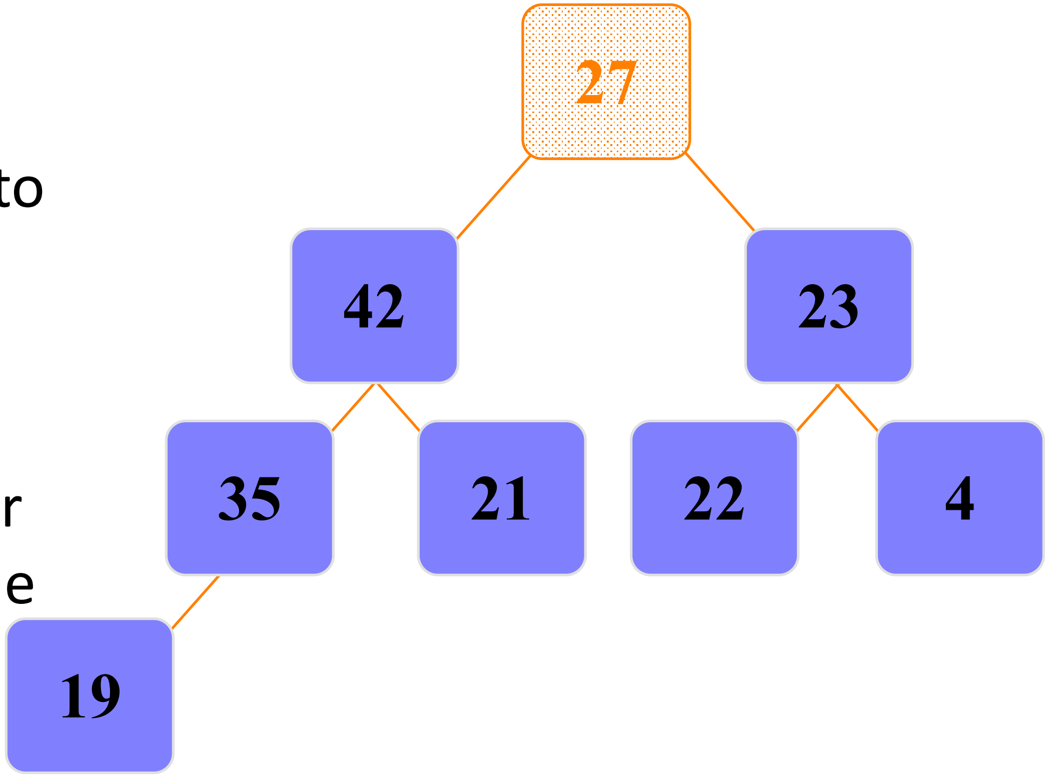
Removing the Top of a Heap

- Move the last node onto the root.



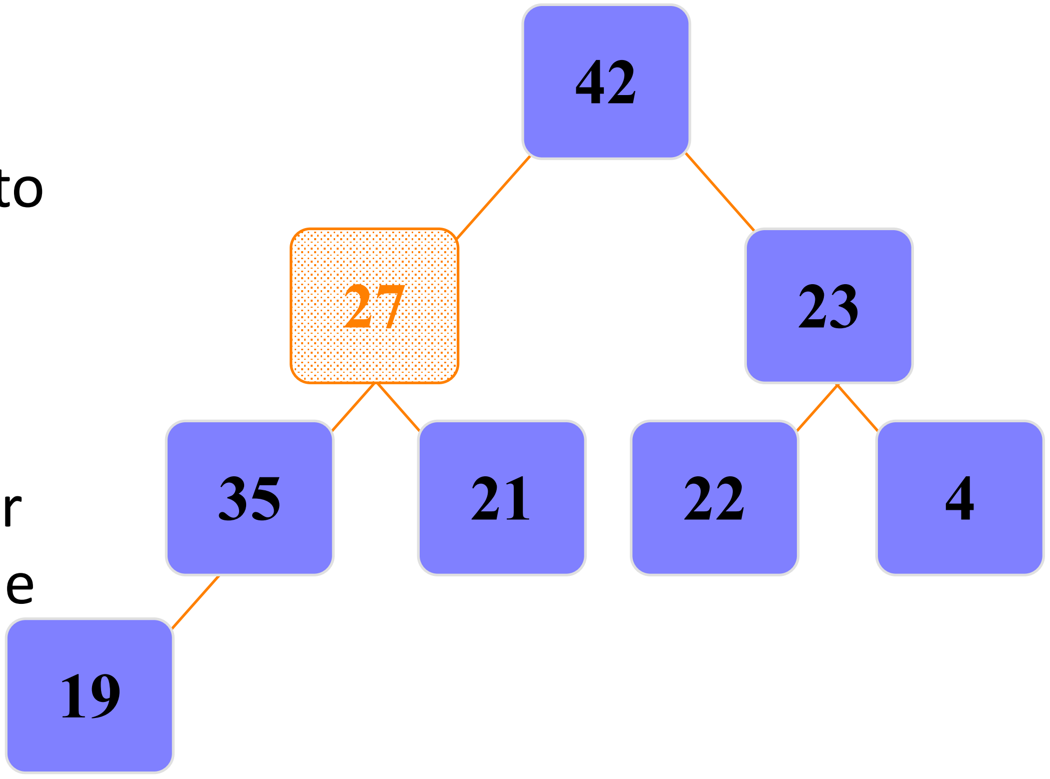
Removing the Top of a Heap

- ❑ Move the last node onto the root.
- ❑ Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



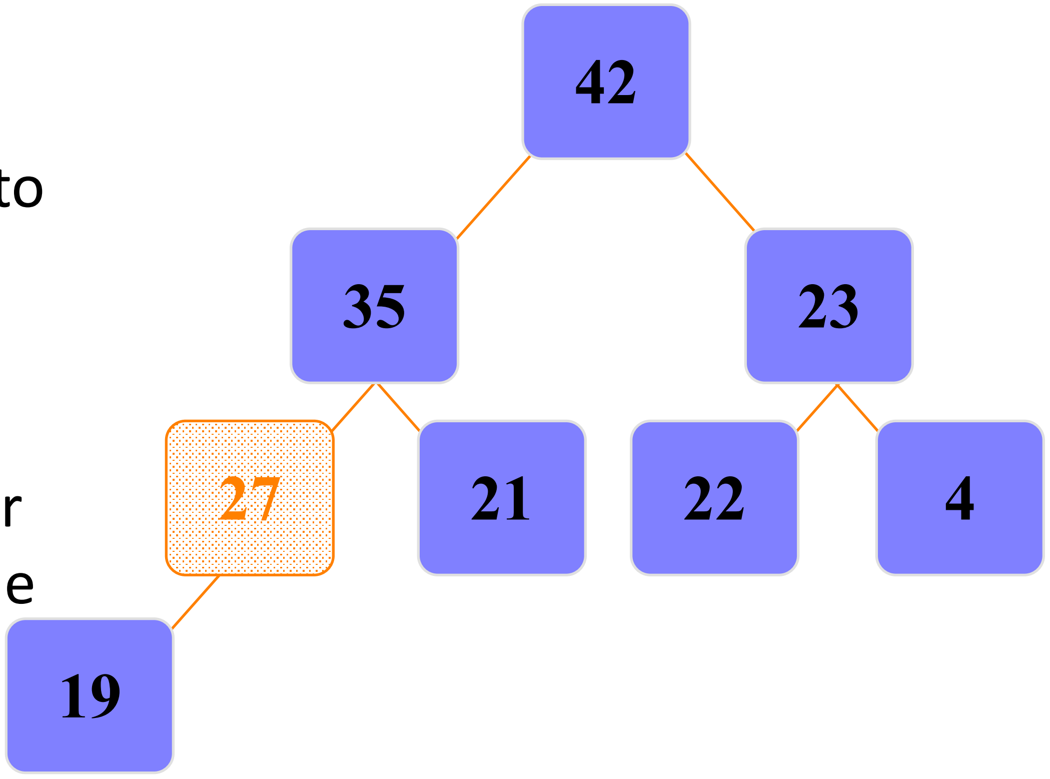
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



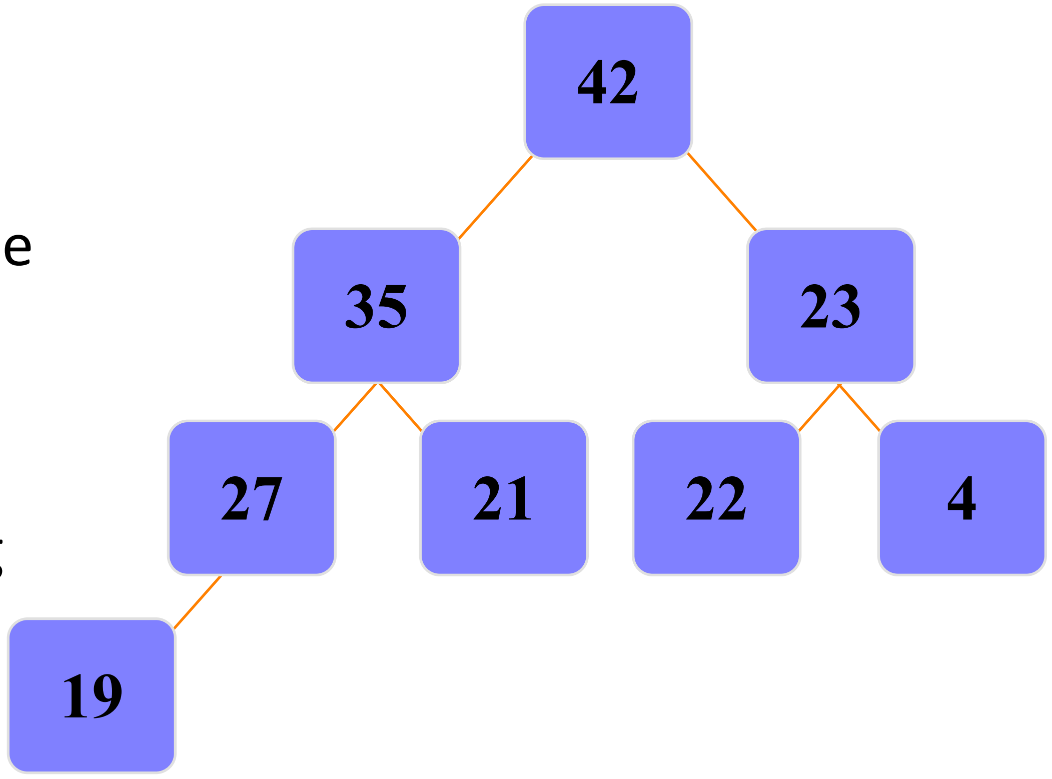
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



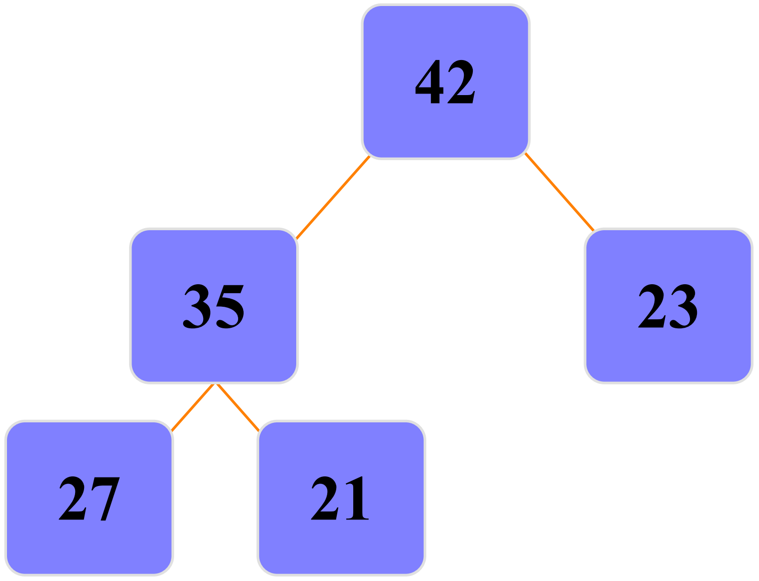
Removing the Top of a Heap

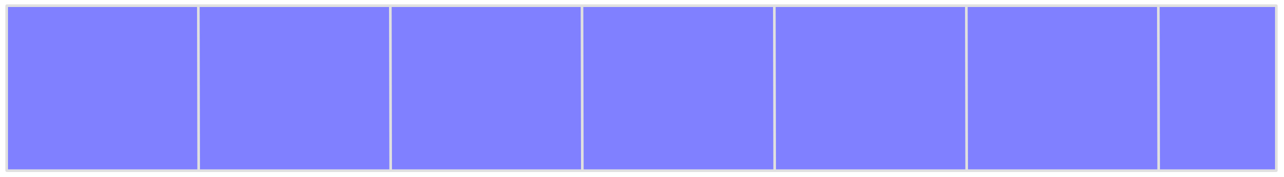
- The children all have keys <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.



Implementing a Heap

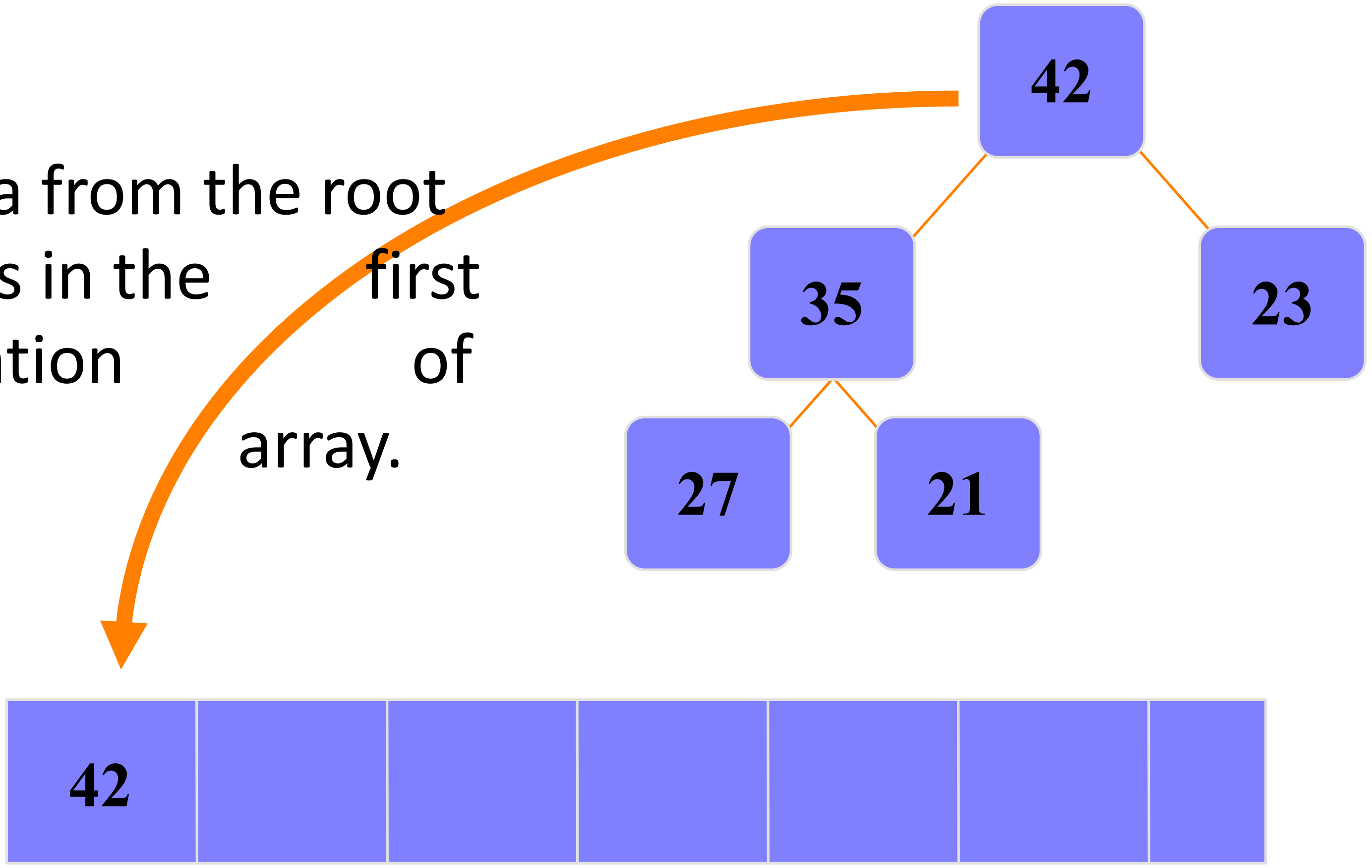
- We will store the data from the nodes in a partially-filled array.





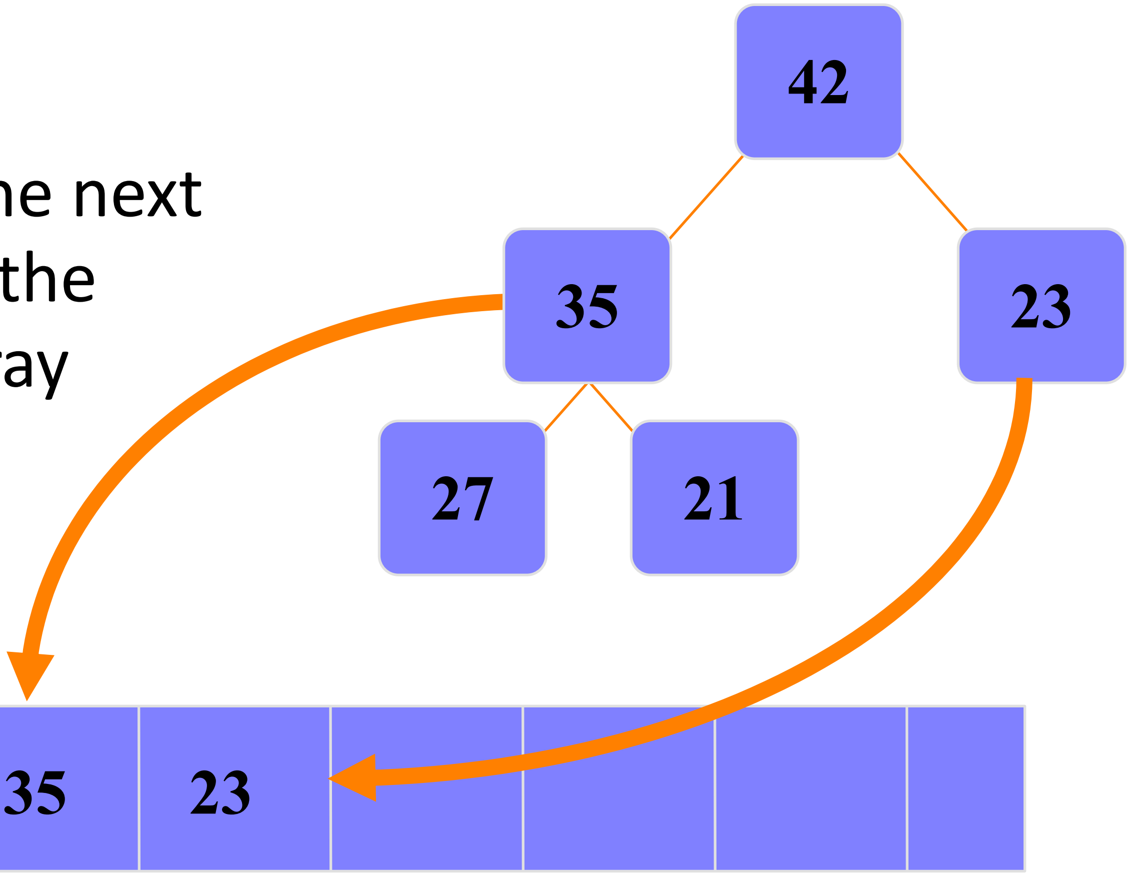
Implementing a Heap

- Data from the root goes in the first location of the array.



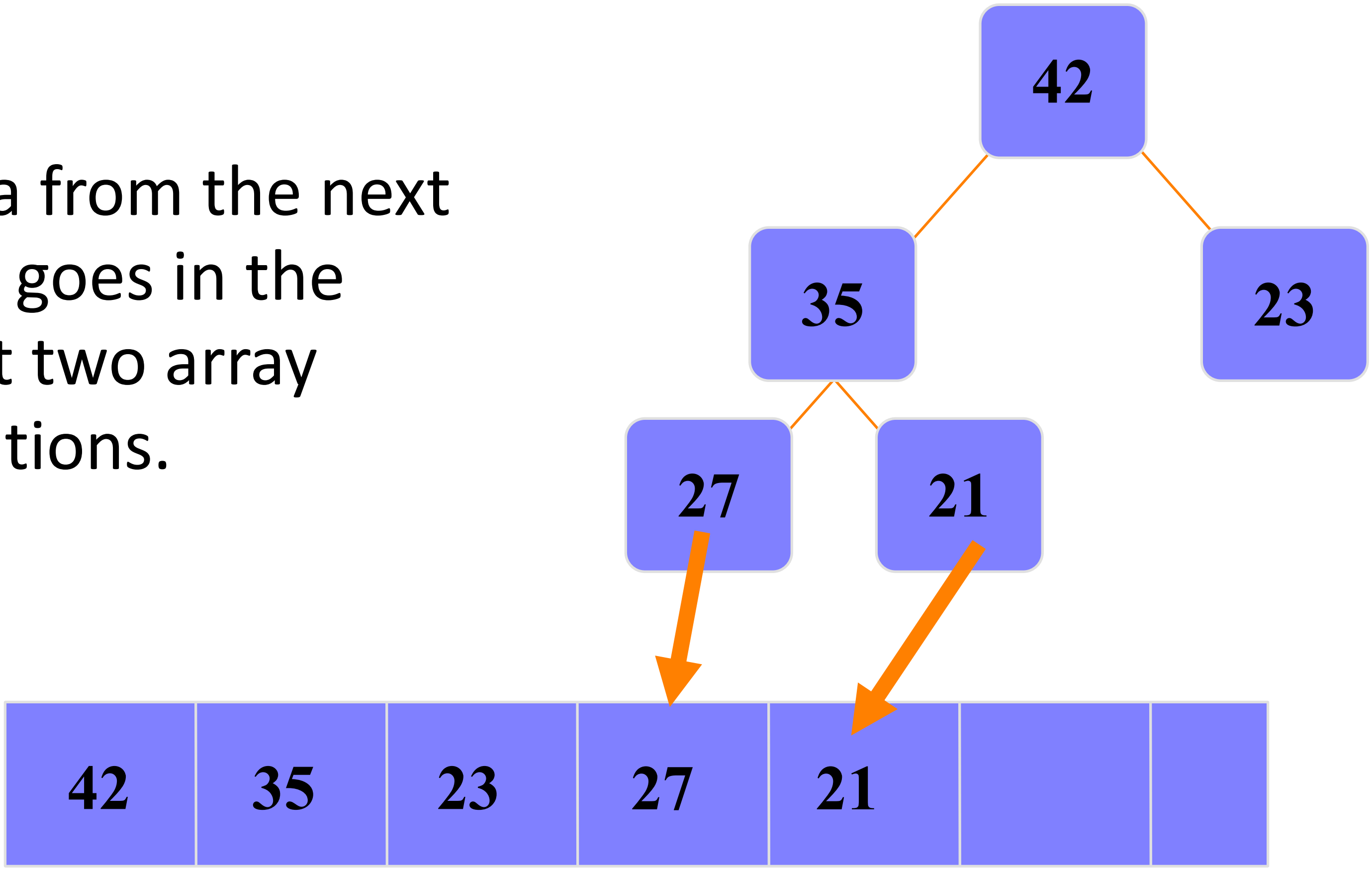
Implementing a Heap

- Data from the next row goes in the next two array locations.



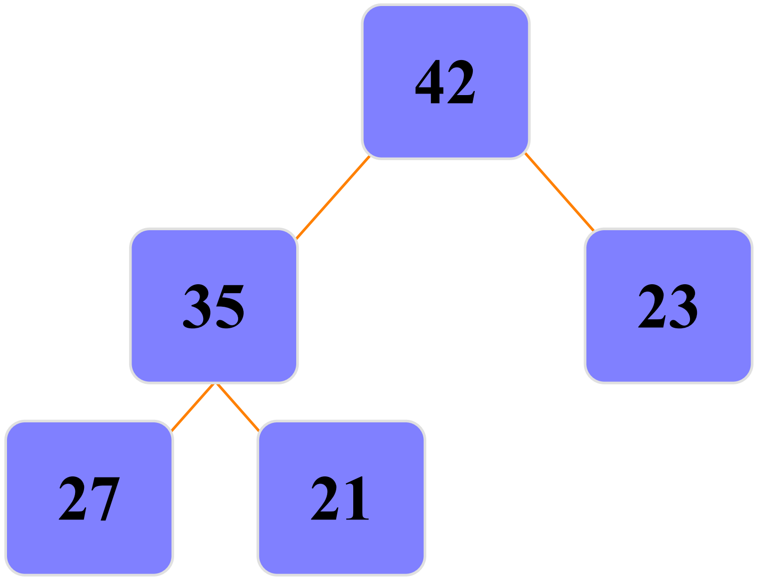
Implementing a Heap

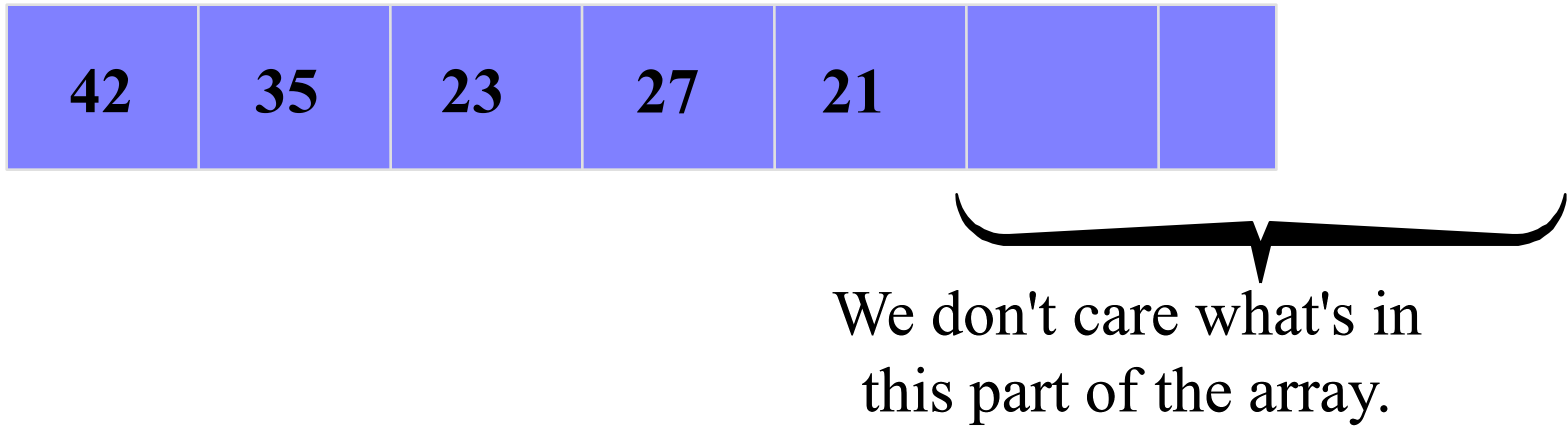
- Data from the next row goes in the next two array locations.



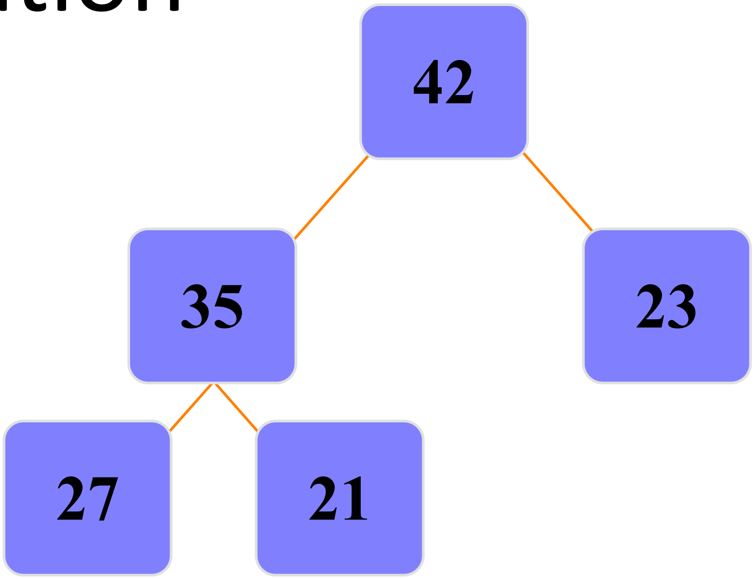
Implementing a Heap

- Data from the next row goes in the next two array locations.

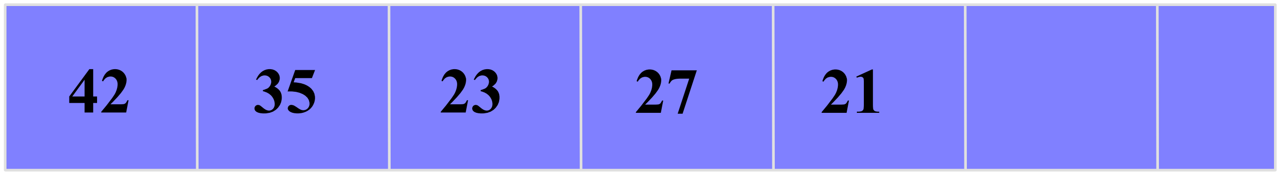




Important Points about the Implementation

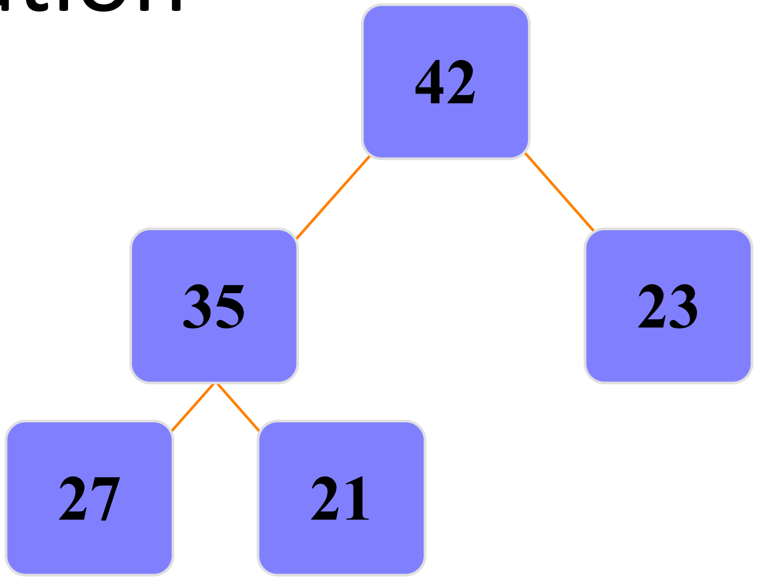


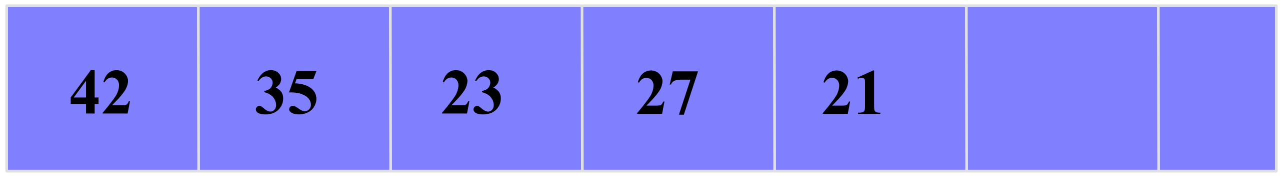
- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.



Important Points about the Implementation

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children. Formulas are given in the book.





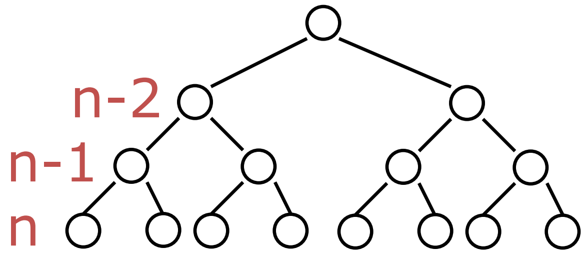
Heap Sort

Heap data structure

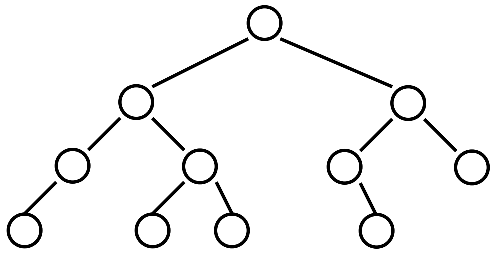
- Binary tree
- Balanced
- Left-justified or Complete
- (Max) Heap property: no node has a value greater than the value in its parent

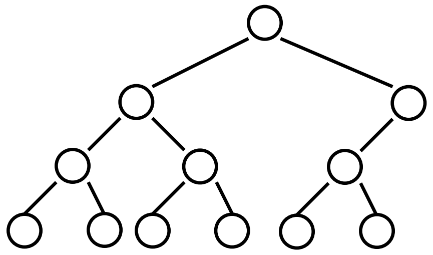
Balanced binary trees

- 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



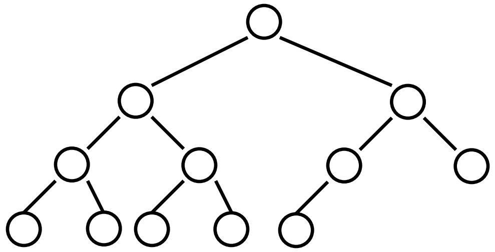
Balanced

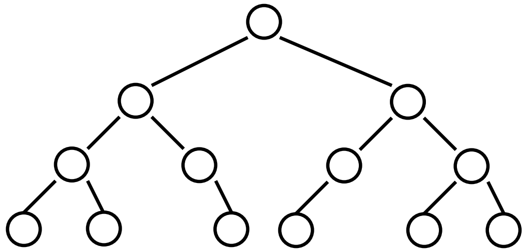

Balanced


Not balanced

Left-justified binary trees

- A balanced binary tree of depth n is left-justified if:
  - it has 2^n nodes at depth n (the tree is “full”), or
  - it has 2^k nodes at depth k, for all $k < n$, *and* all the leaves at depth n are as far left as possible


Left-justified

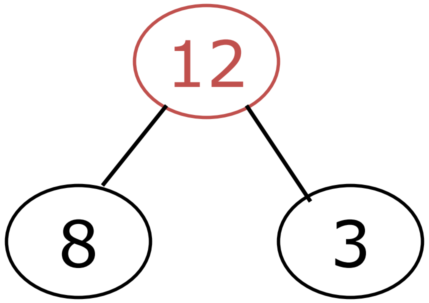

Not left-justified

Building up to heap sort

- How to build a heap
- How to maintain a heap
- How to use a heap to sort data

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

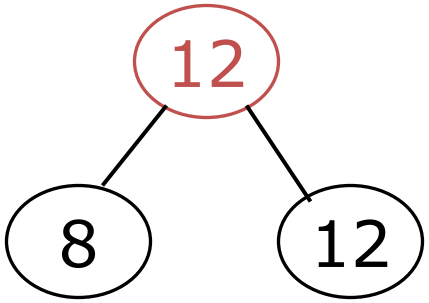
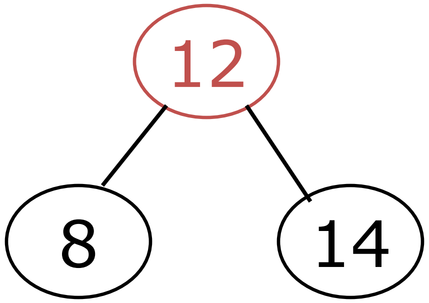


has heap property

has heap property

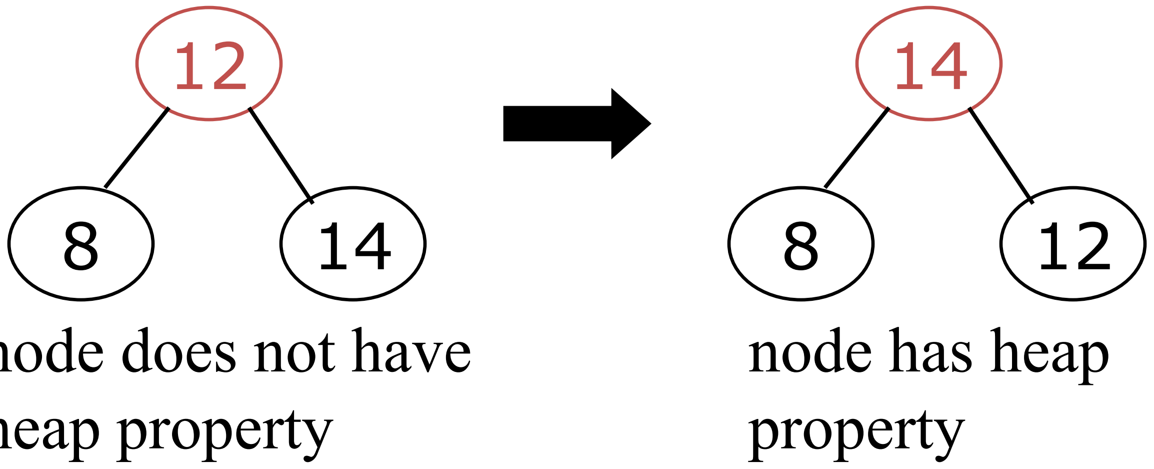
does not have heap property

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

siftUp

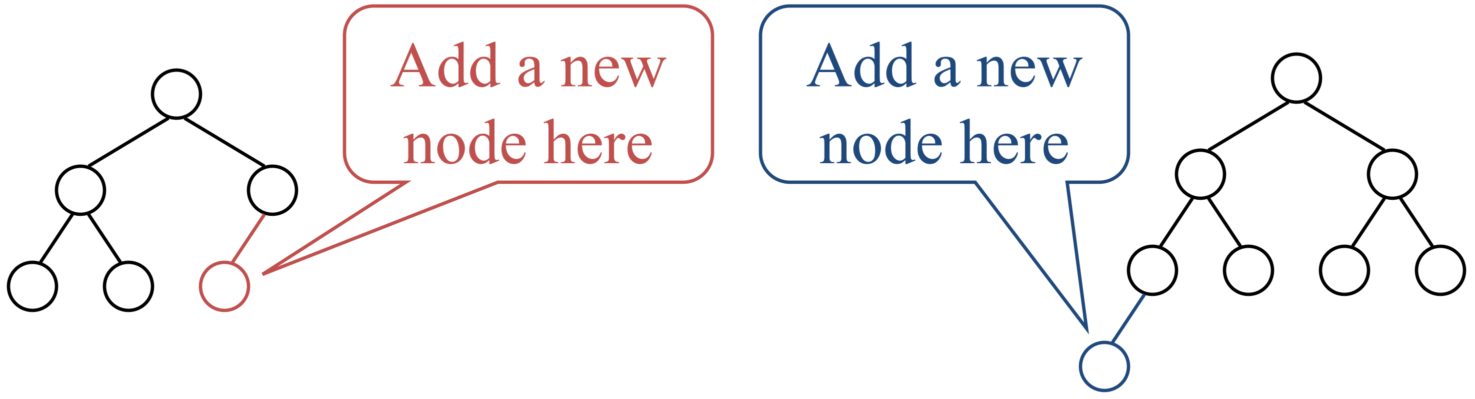
- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



- This is sometimes called sifting up

Constructing a heap I

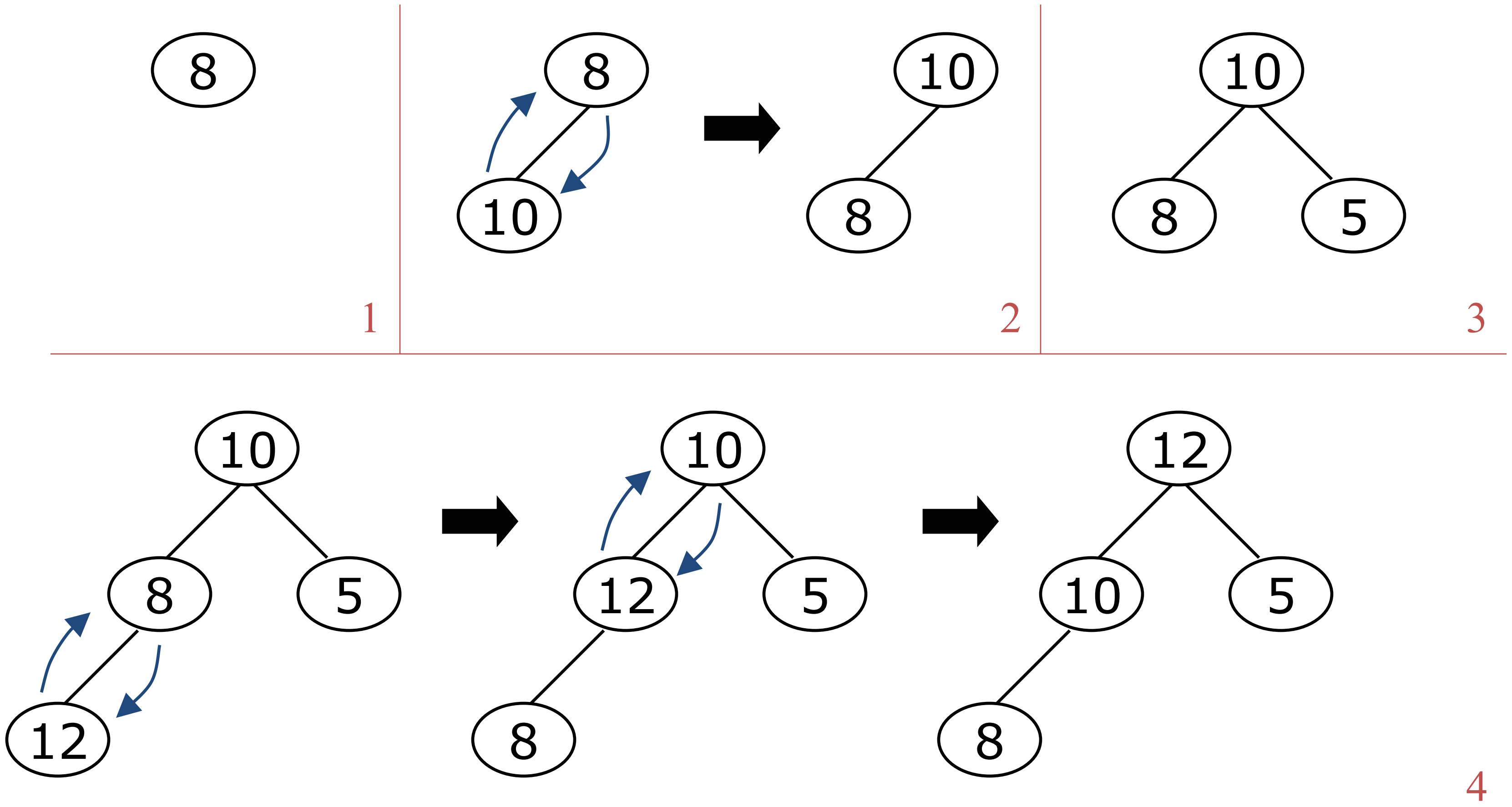
- A tree consisting of a <u>single node</u> is automatically a <u>heap</u>
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the <u>deepest level</u>
  - If the deepest level is full, start a new level
- Examples:



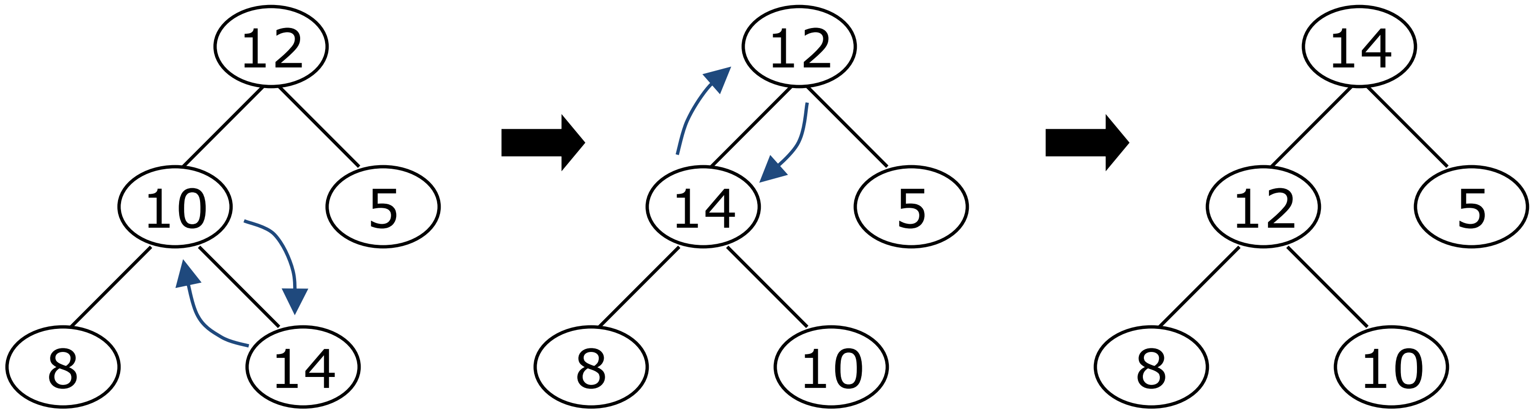
Constructing a heap II

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we 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



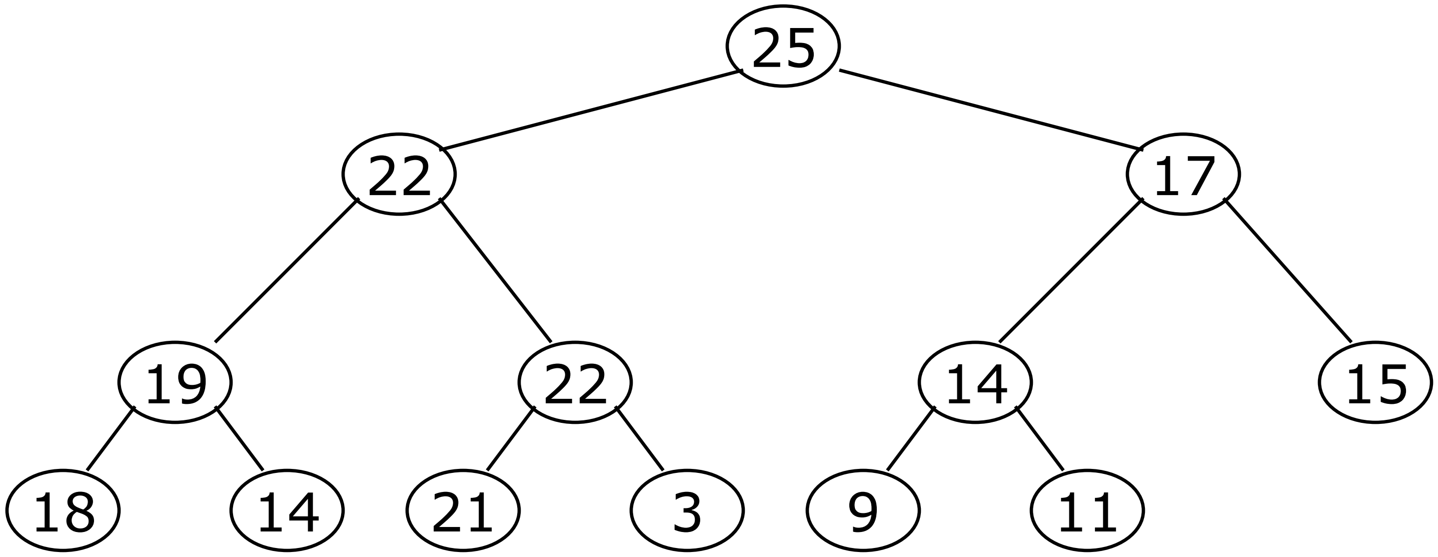
Other children are not affected



- 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

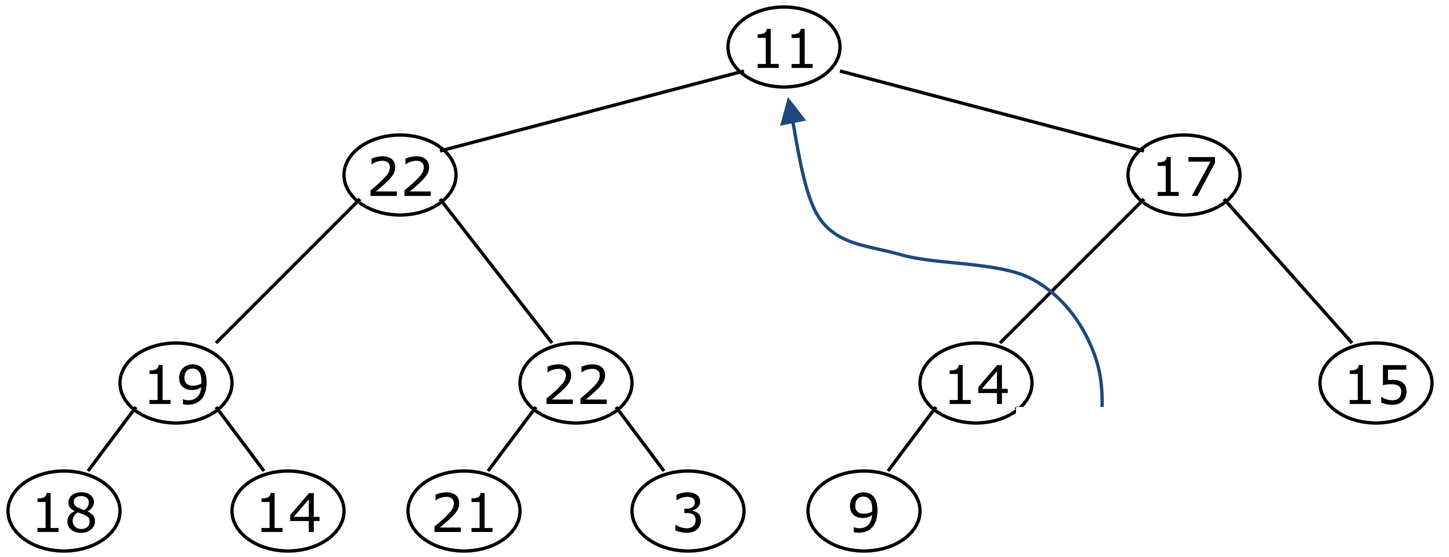
- Here’s a sample binary tree after it has been heapified



- Notice that heapified <u>does *not* mean sorted</u>
- 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 (animated)

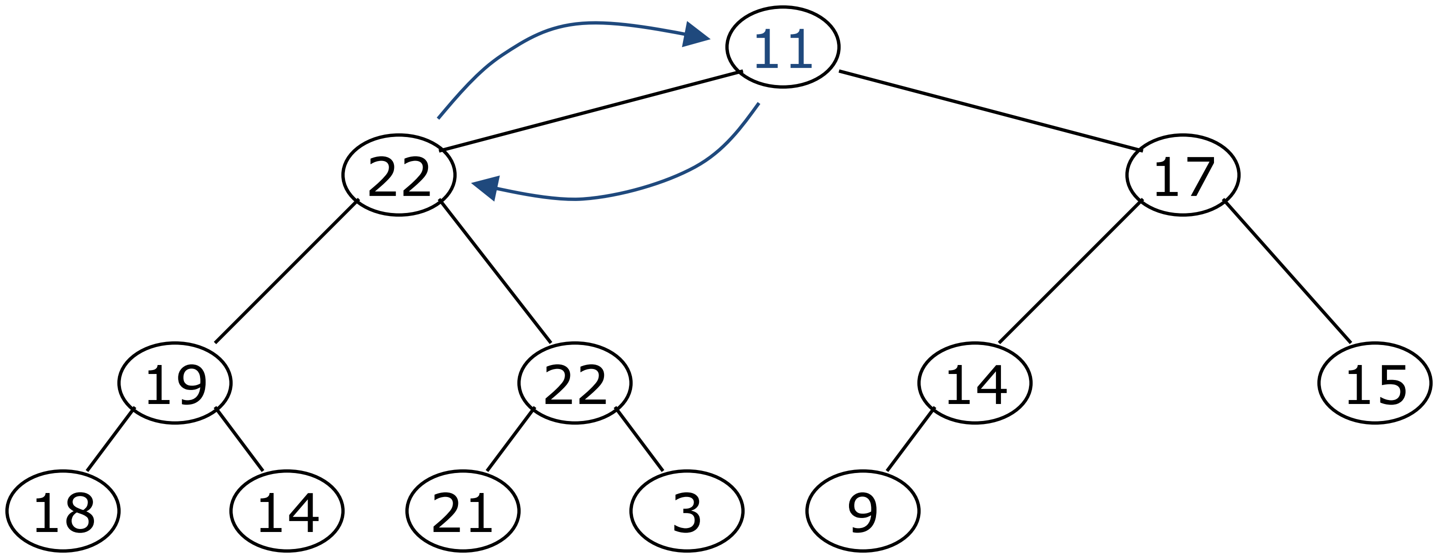
- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- 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

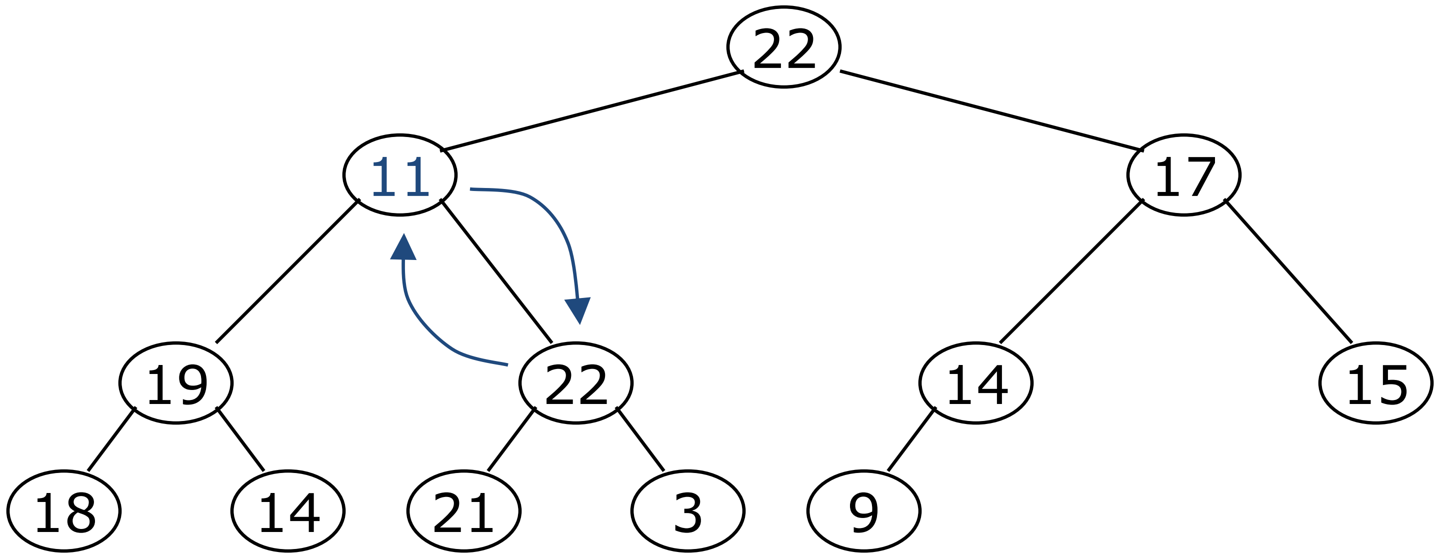
- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property



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

The reHeap method II

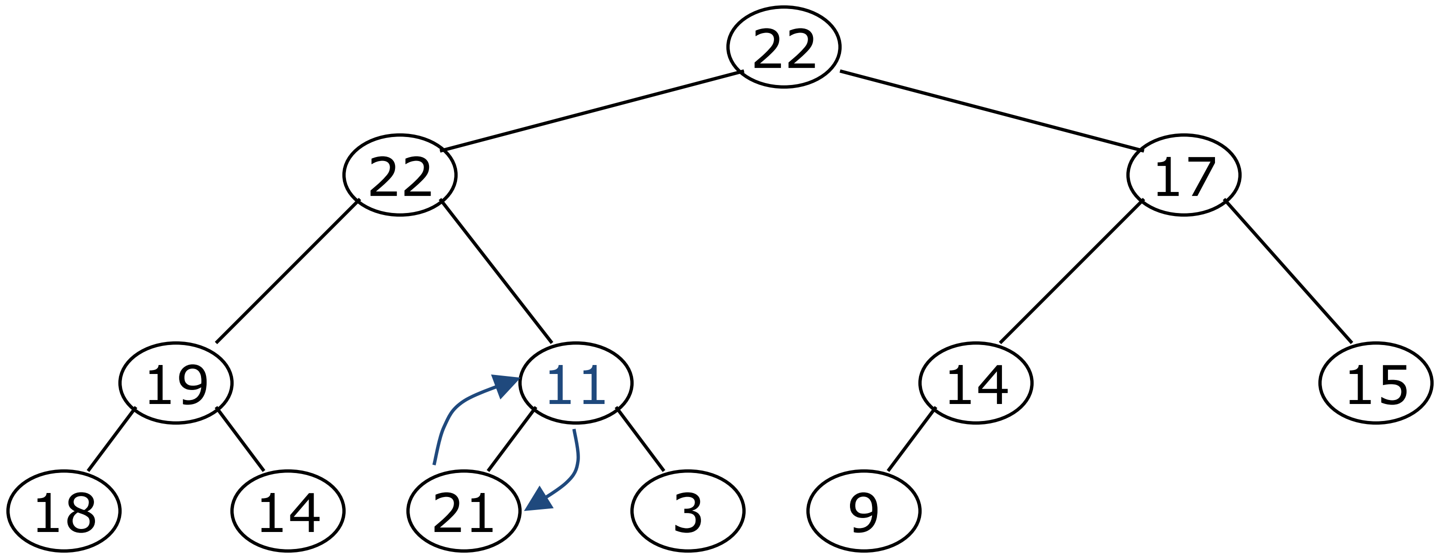
- Now the left child of the root (still the number 11) lacks the heap property



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

The reHeap method III

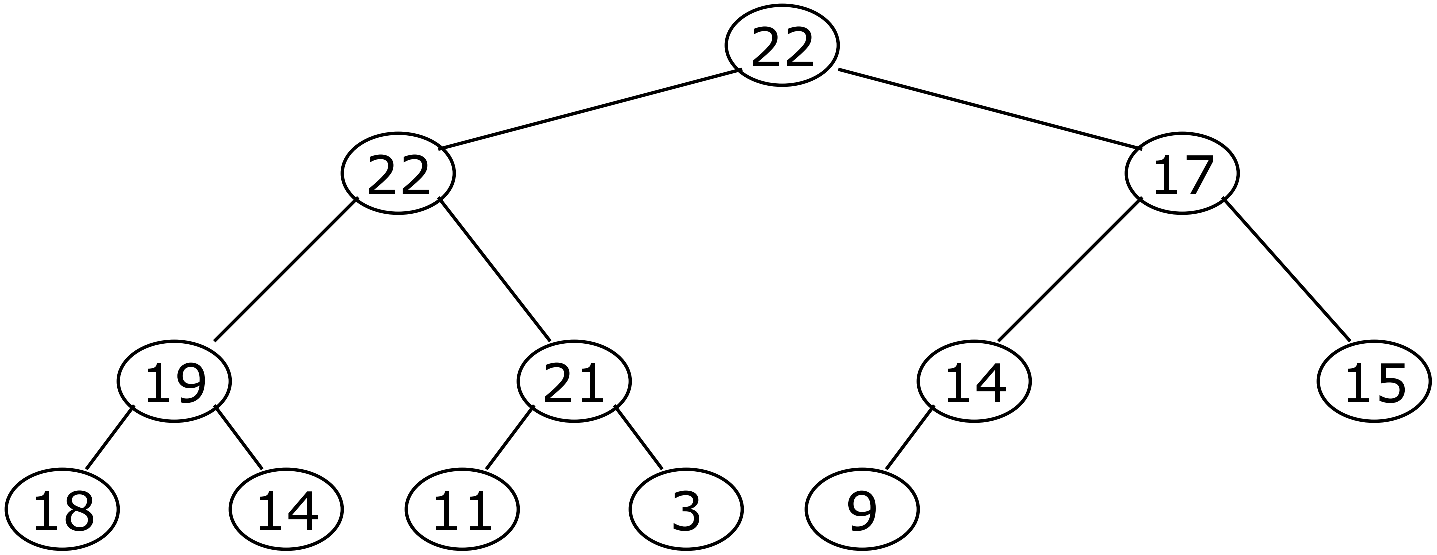
- Now the right child of the left child of the root (still the number 11) lacks the heap property:



- We can siftDown() 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

- 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;
}
```

Key properties

- Determining location of root and “last node” take constant time

- Remove n elements, re-heap each time

Analysis

- 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

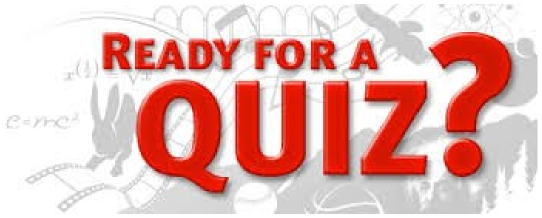
- Construct the heap O(n log n)

- Remove and re-heap O(log n)
  - Do this n times O(n log n)

- Total time O(n log n) + O(n log n)

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!



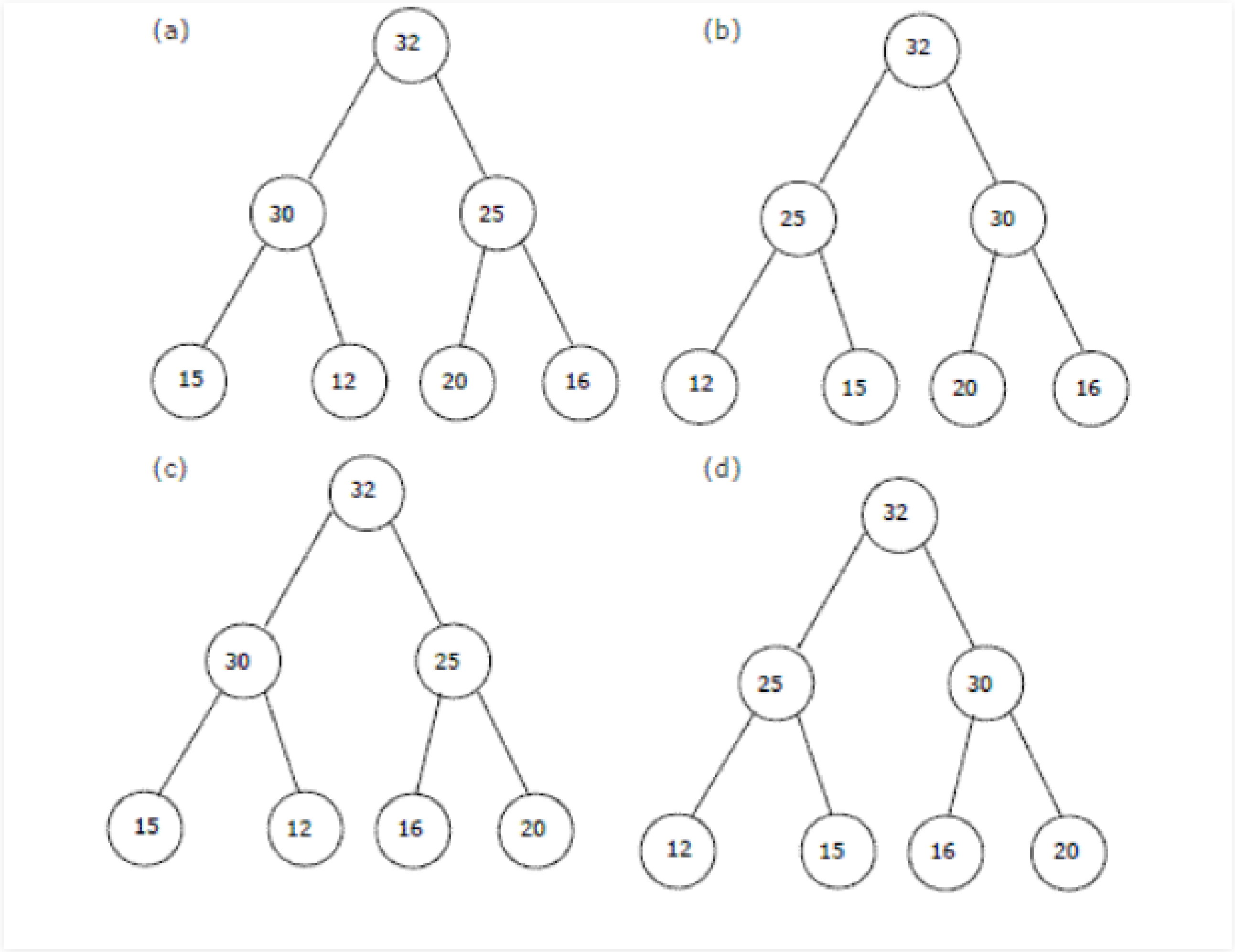
Quiz 1

- Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap?

A	25,12,16,13,10,8,14
B	25,12,16,13,10,8,14
C	25,14,16,13,10,8,12
D	25,14,12,13,10,8,16

Quiz 2

- The elements 32, 15, 20, 30, 12, 25, 16 are inserted one by one in the given order into a Max Heap. The resultant Max Heap is.



Quiz 3

- In a min-heap with n elements with the smallest element at the root, the 7th smallest element can be found in time
- a) Θ(n log n)
- b) Θ(n)
- c) Θ(log n)
- d) Θ(1)
- Discuss in group, two cases that we have/ have not duplicates in Min-Heap?

Quiz 4

- In a binary max heap containing n numbers, the smallest element can be found in time

A	0(n)
B	O(logn)
C	0(loglogn)
D	0(1)

Quiz 5

- Write a pseudocode for heapification upward in max-heap.