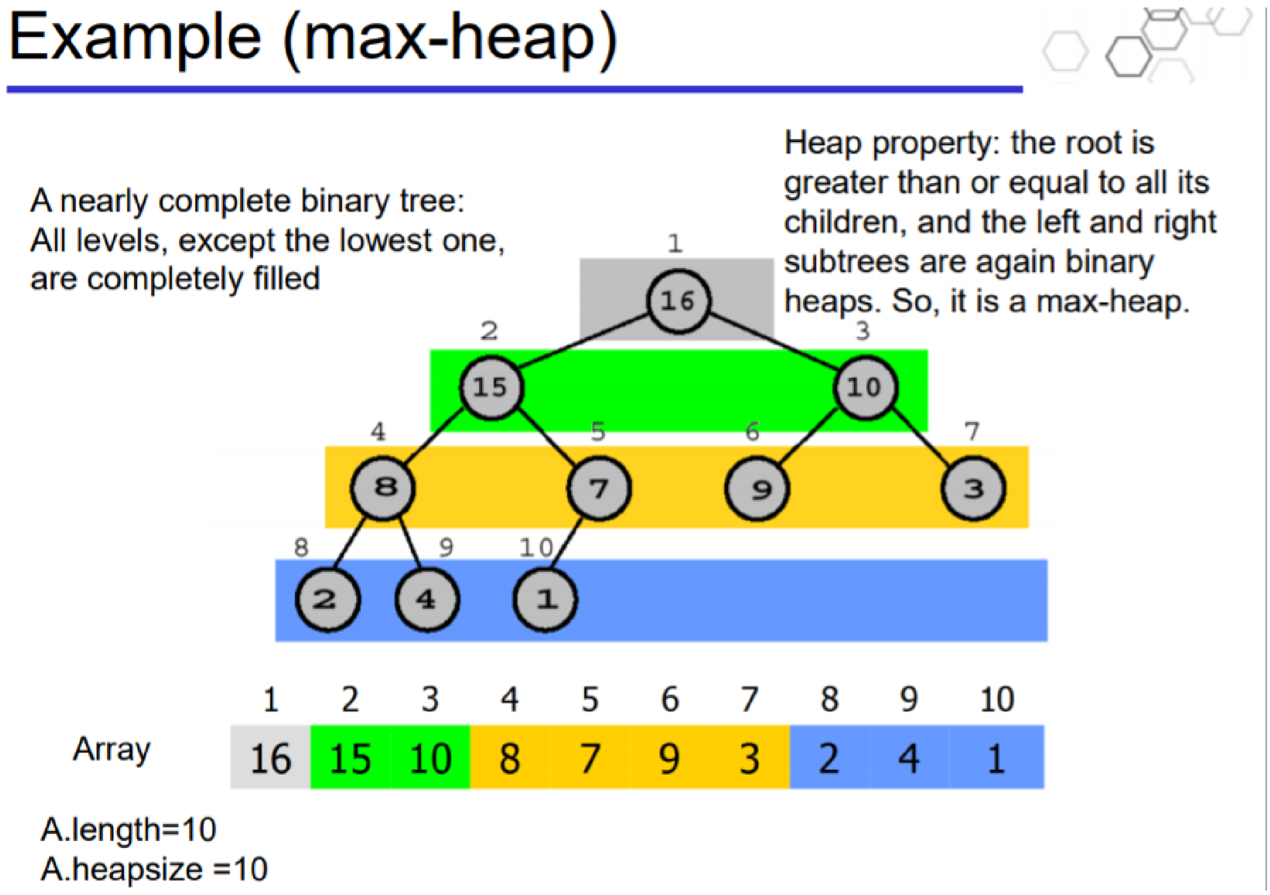
Heapsort and ADTs: Heapify.

Heap

- Binary heap data structure A
  - Array
  - Can be viewed as a nerly complete binary tree
    - All levels, except the lowest one, are completely filled
  - Heap property
    - The root is greater than or equal to all its children, and the left and right subtrees are again binary heaps (max-heap)
    - The root is less than or equal to all its children, and the left and right subtrees are again binary heaps (min-heap)
- Two attributes
  - A.length: the number of elements in the array
  - A.heapsize: the number of elements in the heap that is stored in the array
    - 1 <= A.heapsize <= A.length
    - A[1..A.length] may contain many elements, but only the elements in A[1..A.heap-size] are valid elements of the heap



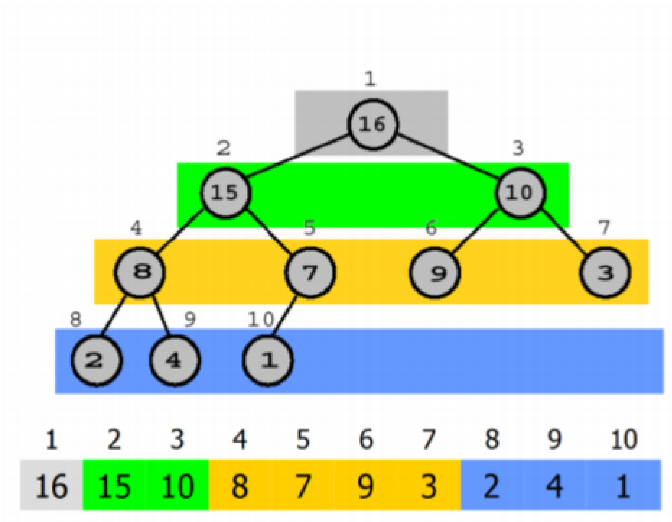
Parent, left child, and right child

All levels, except the lowest one, are completely filled

Parent (i)
  return $\lfloor i/2 \rfloor$

Left (i)
  return $2i$

Right (i)
  return $2i+1$



Heap propertiy:

$A[\text{Parent}(i)] \geq A[i]$

The value of a node is at most the value of its parent.

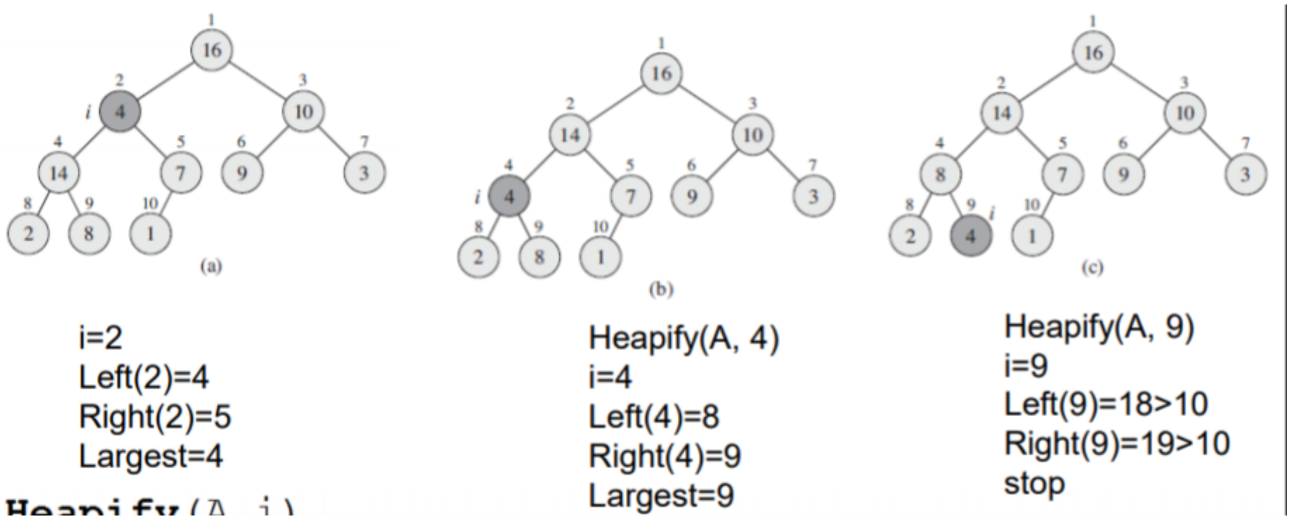
Remember! Left = left child!

Maintaining the heap property

- Heapify
  - Input: Array A and an index i into the array.
  - Assume:
    - Binary trees rooted at Left(i) and Right(i) are heaps
  - But, A[i] might be smaller than its two children, thus violating the heap property.
  - The method Heapify makes A a heap by moving A[i] down the heap until the heap property is satisfied again.

```
Heapify(A,i)
l  = LEFT(i)
r  = RIGHT(i)
largest = index of the largest among A[i], A[l], A[r]
if largest != i then
   exchange A[i]↔A[largest]
   Heapify(A, largest)
```

Example: Heapify(A,2)



Analysis of Heapify

- We need to ask ourselves the following questions.
    - Is Heapify a recursive algorithm or not?
    - If yes, write down the recurrence and solve the recurrence.
    - If not, use the RAM model.

- Identifying the recurrence for heapify
    - Dividing (lines 1-3)
        - Figuring out the relationship among the elemtents A[i], A[l], and A[r].
        - Can be done in constant time, i.e. $\Theta(1)$.
    - Conquer (lines 4-6)
        - Case 1: if A[i] is the largest among A[i], A[l], A[r] already, do nothing.
        - Case 2: Otherwise, conquer the same problem (i.e., Heapify) on one of the subtree of node i.
        - How many sub-problems are you going to solve for each case?
            - Case 1: 0
            - Case 2: 1
            - We at most need to solve only one sub-problem.
    - Dividing: $\Theta(1)$
    - Conquer:
        - In case there is one sub-problem, what is the size of the sub-problem?
            - It depends on which sub-tree you need to continue to Heapify.
            - It depends on the size of the biggest sub-tree of node i.
        - A sub-tree's size is at most 2n/3.
            - It happens when the bottom level of the tree is half full.
    - Combining: nothing.

Worst case is where one side has 2/3 of elements and the other has 1/3 of it

- Recurrence for Heapify (A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.)

    - $T(n) \leq T(2n/3) + \Theta(1)$
    - Let's solve $T(n) = T(2n/3) + \Theta(1)$ first
        - Matter method: a = 1, b = 3/2 = 1.5, $lg_b a = 0$, $n^0 = 1$, case 2 applies.
        - $T(n) = \Theta(n^0 lgn) = \Theta(lgn)$
    - Since the real recurrence uses "<=" but not "=".
        - $T(n) = O(lgn)$

What is the intuition here? In the worst case, heapify needs to traverse from the root to a leave. Since heap is a binary tree, the depth of the tree is at most lgn.

Some Notes

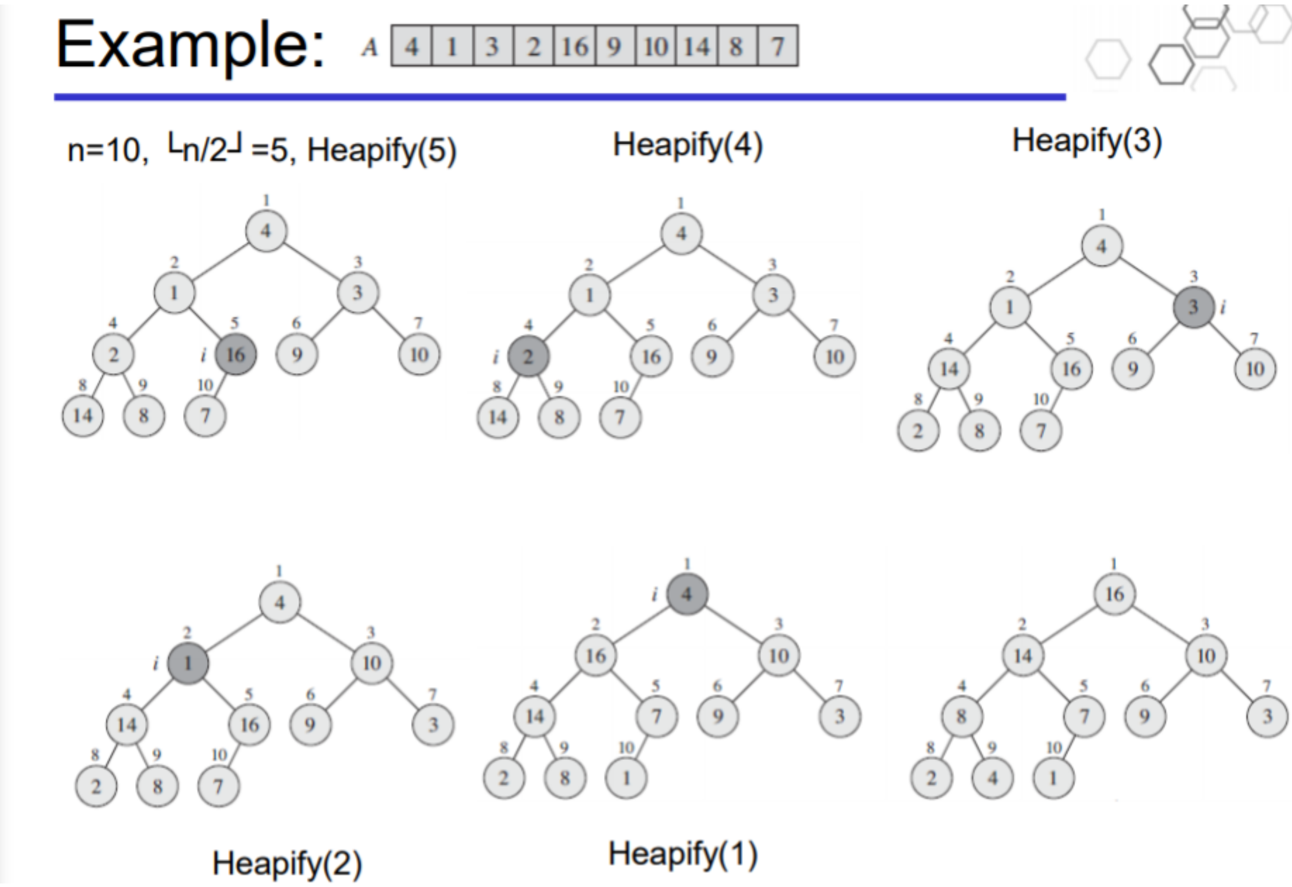
- Sometimes, it is more important to write down a correct recurrence than solving the recurrence
- How many sub-problems and what is the size of each sub-problem
    - Quick sort: best case vs. worst case
    - Heapify: the largest size sub-problem
- What is the cost of dividing and combining
    - Quick sort: partition, dividing phase.
    - Merge sort: merge, combining phase.

The leafs in the tree is already/always heaps. So call heap on the first element with children.

Building a heap from an array

- Convert an array A[1..n] into a heap
- Notice that the elements in the subarray $A[(FLOOR\, n/2\; FLOOR + 1) ... n]$ are already 1-element heaps to begin with!
    - In other words, these elements do not have any children
- Call heapify from the $FLOOR\, n/2\; FLOOR$ -th element down to the first element.

> ```
> Build-Heap(A)
> for i = ⌊n/2⌋ downto 1 do
>    Heapify(A, i)
> ```



Analysis of building a heap

```
Build-Heap(A)
for i =⌊n/2⌋ downto 1 do
    Heapify(A, i)
```

- A non-tight analysis
  - The for loop makes O(n) iterations.
  - Each iteration is a Heapify which takes O(lgn).
  - Thus, O(nlgn) for building a heap with n elements.
- This analysis is not wrong, but not tight.
  - Why? Because not every single heapify need to take O(lgn), but only the heapify on the root takes O(lgn).
  - Precisely, heapify on a node takes O(h), where h is the height of the node.
- A tight analysis
  - For each height h, we count how mayn nodes are there in height h, say z_h nodes.
  - The largest height of a heap with n elements is lgn. This means h is 0 to lgn.
  - Then, we sum $z_0, z_1, z_2, \ldots, z_{lgn}$
  - Equivalently, we compute $\sum_{h=0}^{lgn} h * z_h$, which is then the total run time.
- Let's identify $z_h = 2^{lgn-h}$ and compute $\sum_{h=0}^{lgn} h * z_h$
  - Proof can also be found in CLRS page 157-159;