CSC 383 – Data Structures and Algorithms in Java

Heaps

and Priority Queue

Definition

The binary heap data structure is an array object that can be viewed as a nearly complete binary tree. Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

Max and Min

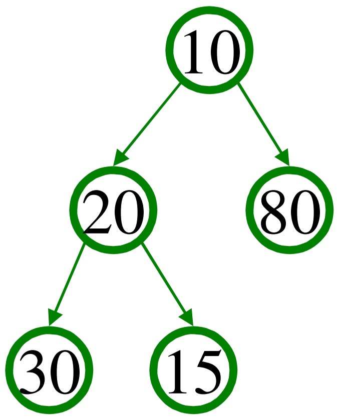
There are two kinds of heaps – max-heaps and min-heaps

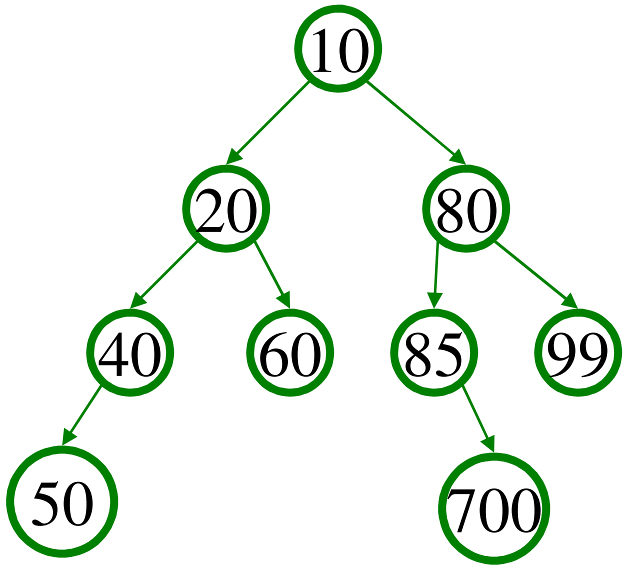
- Max-Heaps: Must satisfy the property that for every node *i* other than the root, the value of node *i* is at most the value of its parent.
- Min-Heaps: Must satisfy the property that for every node *i* other than the root, the value of node *i* is at least the value of its parent.

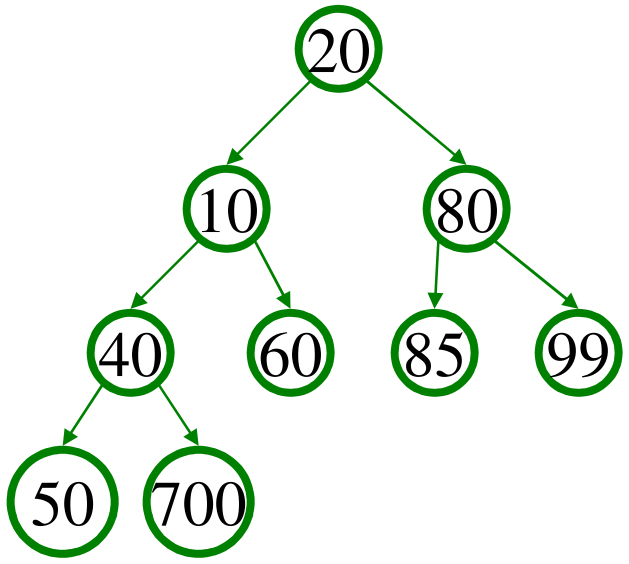
Heap Property

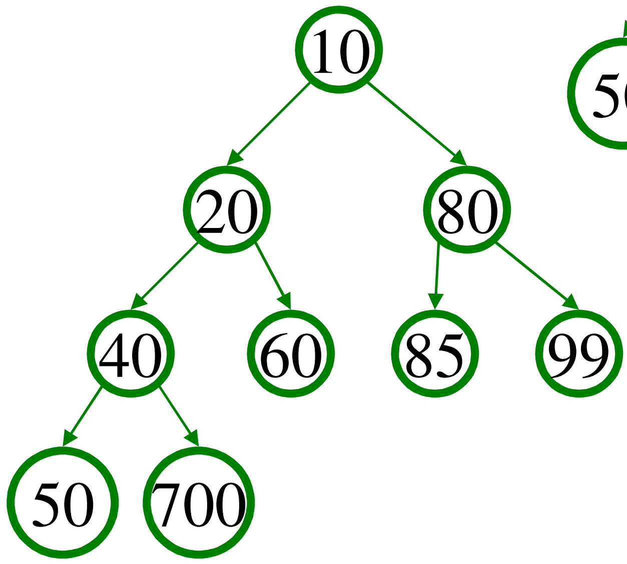
- So, with heaps, either the minimum or maximum element is at the root of the tree. Why not use an AVL tree and keep these elements at the far left/far right respectively?
- Well, to find it would cost logarithmic time (instead of constant) for any node that is along the path from the root to a leaf.

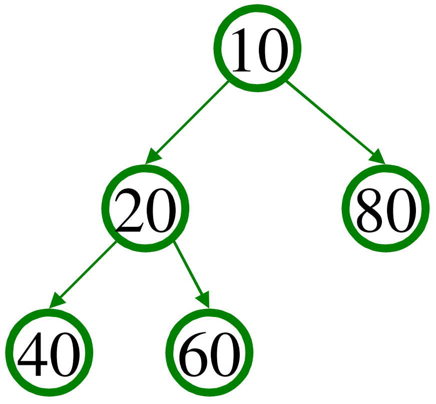
Which are min-heaps?

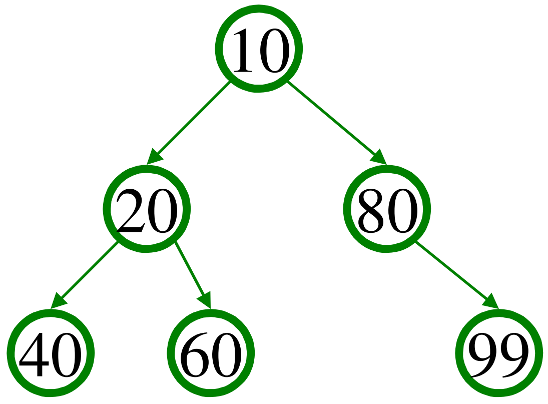




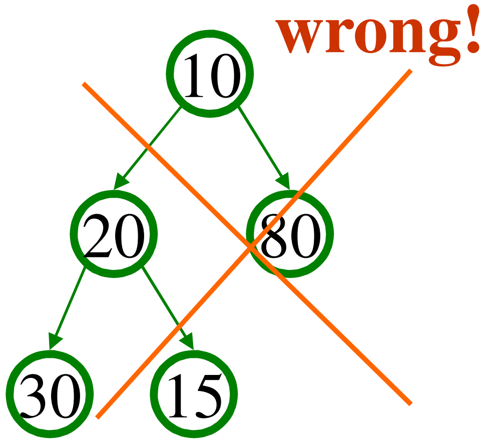


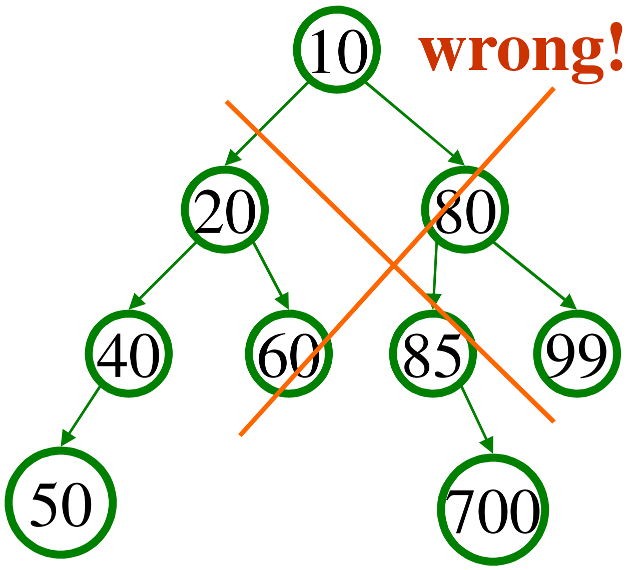


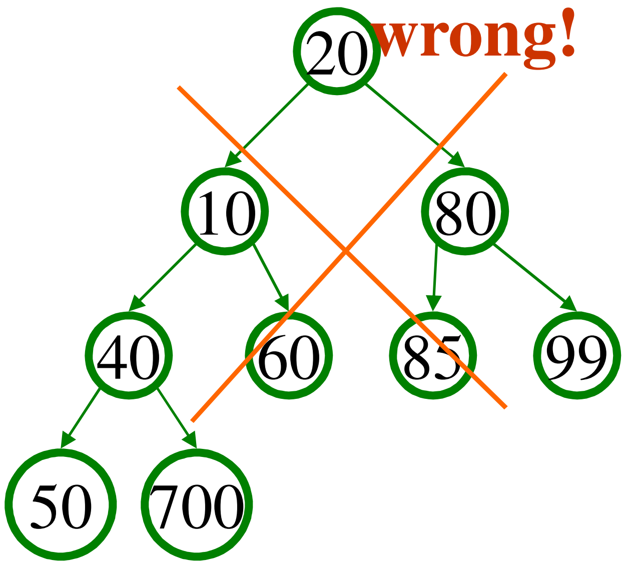


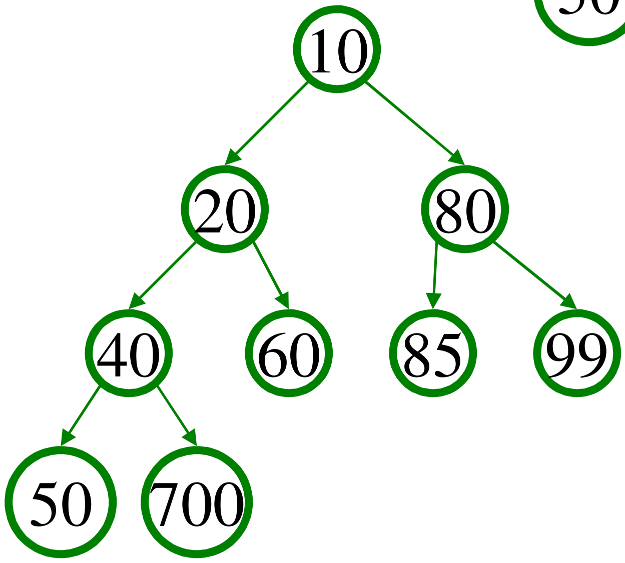


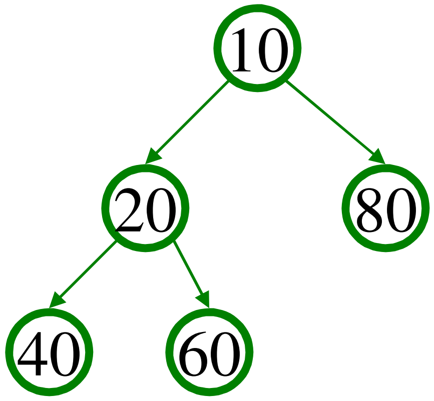
Which are min-heaps?

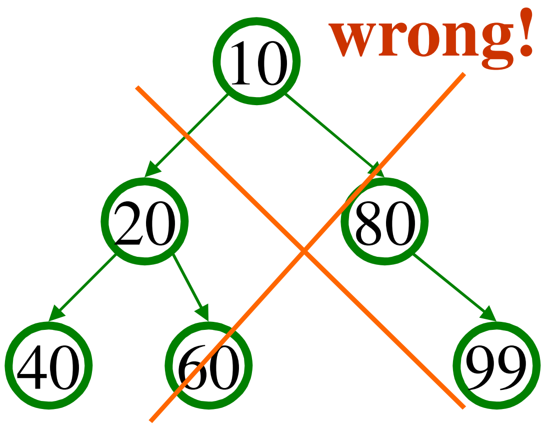




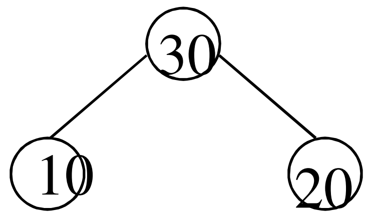


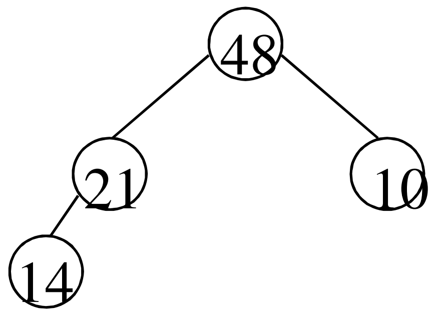


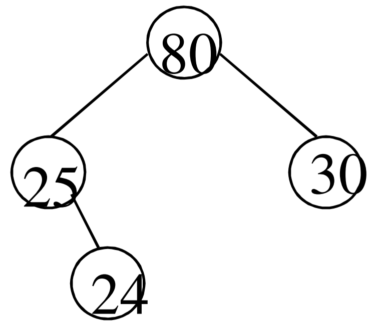


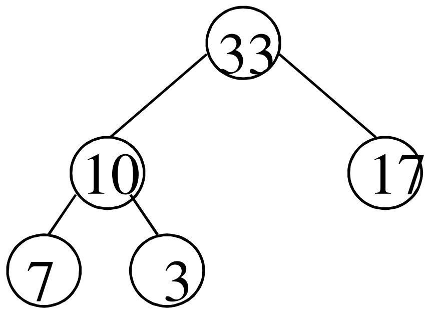


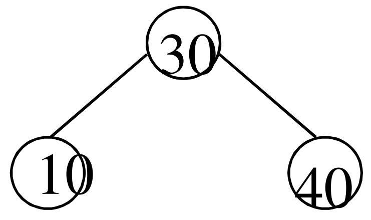
Which are Max-Heaps?

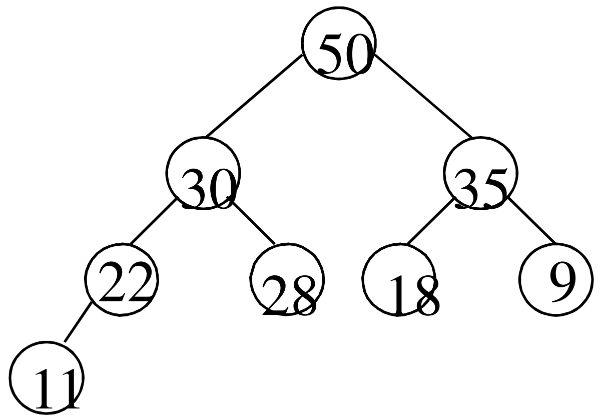




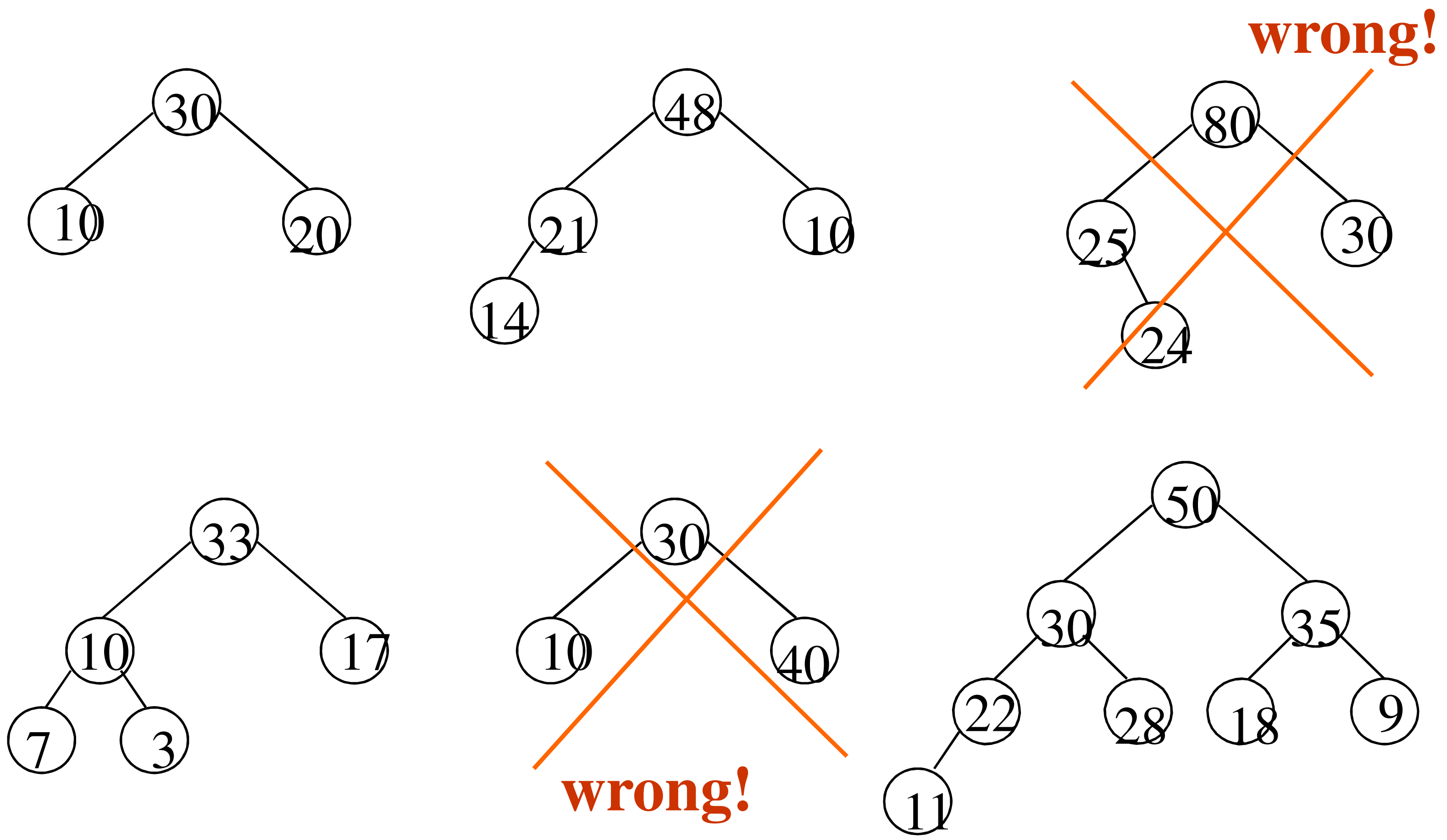






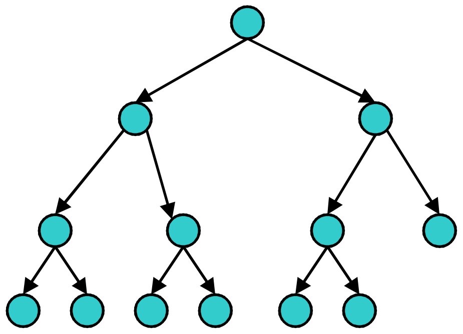


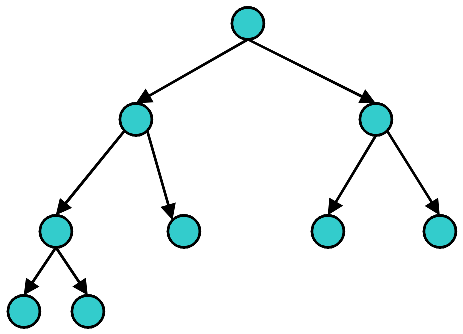
Which are Max-Heaps?



Heap height and runtime

- height of a complete tree is always log *n*, because it is always balanced
  - this suggests that searches, adds, and removes will have O(log *n*) worst-case runtime
  - because of this, if we implement a priority queue using a heap, we can provide the O(log *n*) runtime required for the `add` and `remove` operations



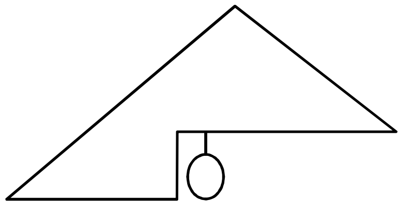


n-node complete tree of height h:

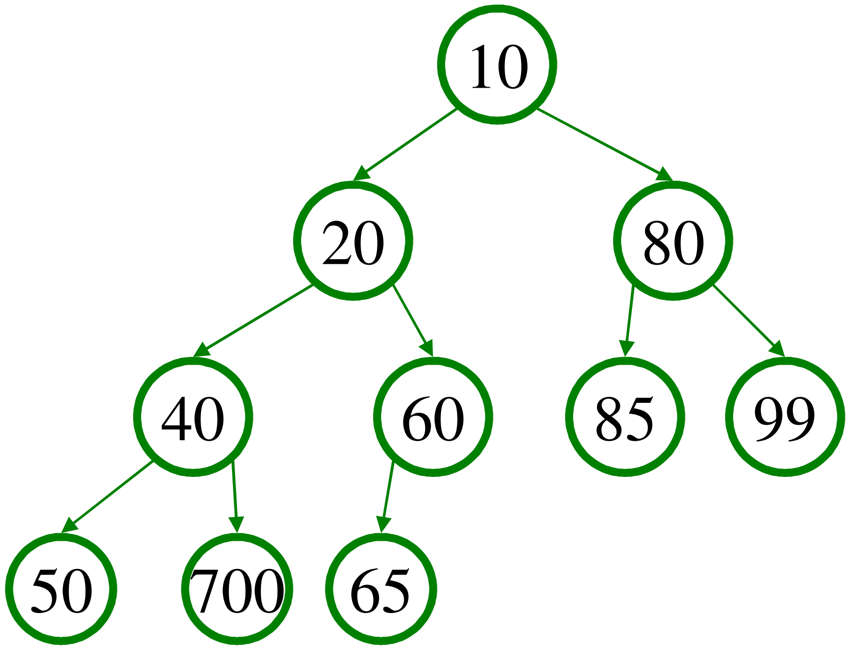
$h = \lfloor \log n \rfloor$

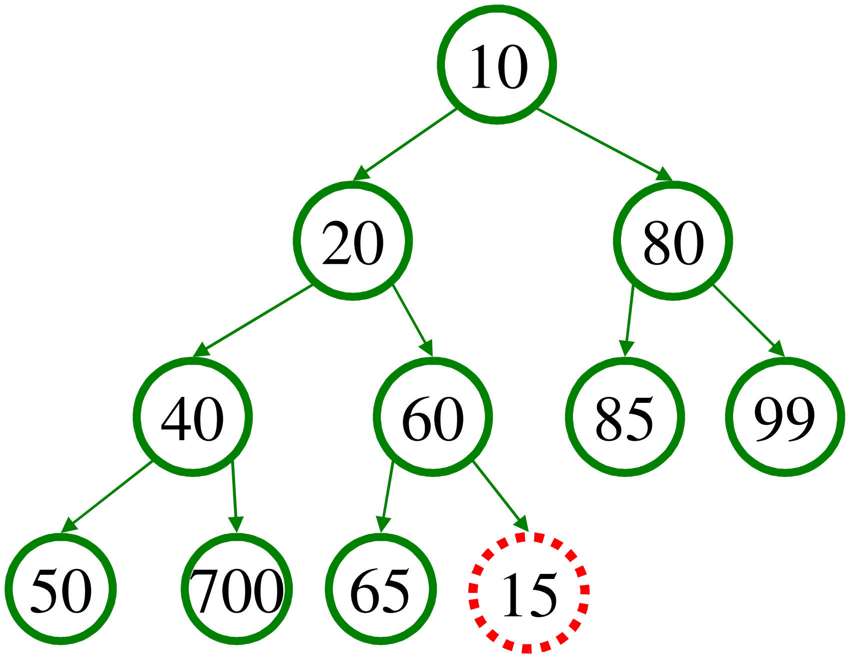
$2^h \leq n \leq 2^{h+1} - 1$

Adding to a heap



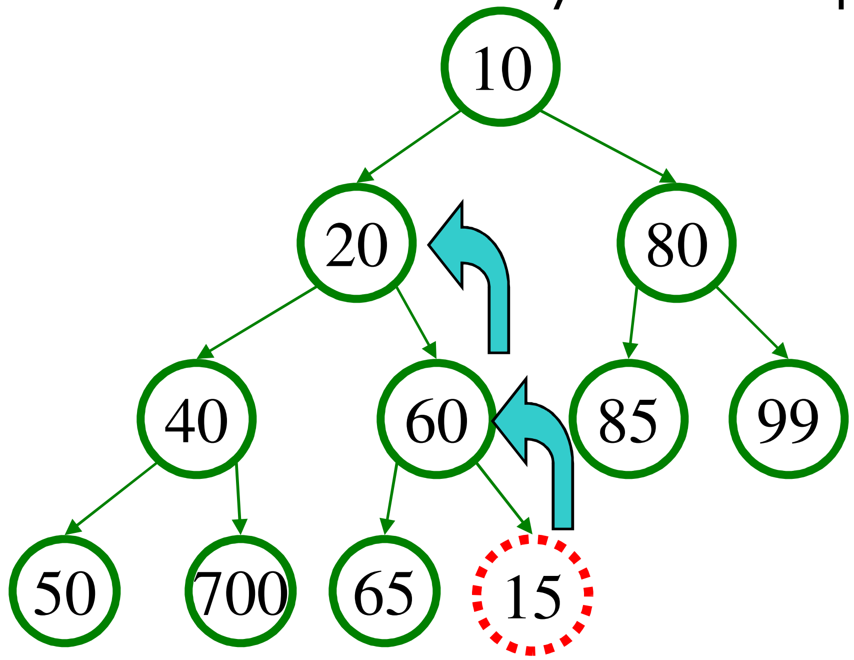
- when an element is added to a heap, it should be initially placed as the rightmost leaf (to maintain the completeness property)
  - heap ordering property becomes broken!

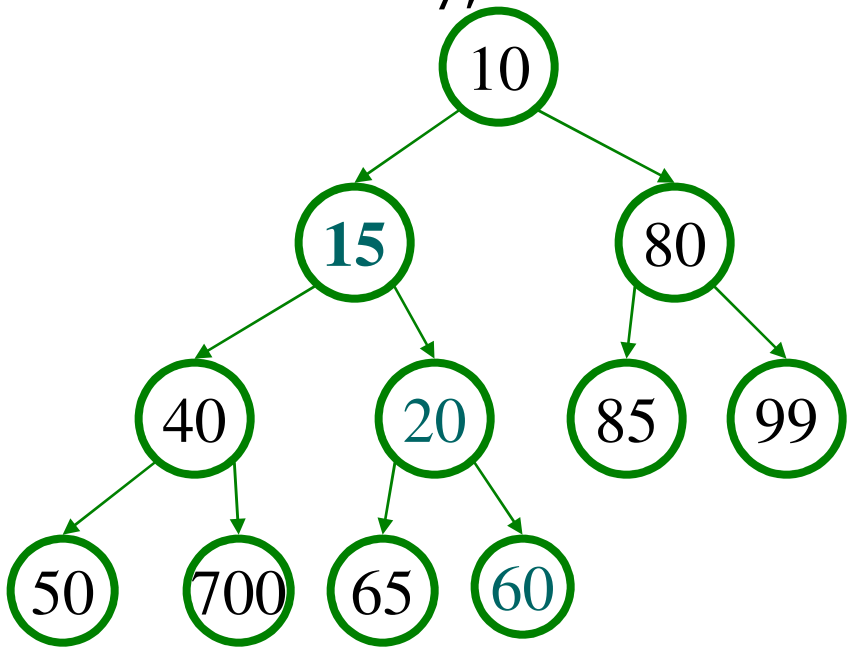




Adding to a heap, cont'd.

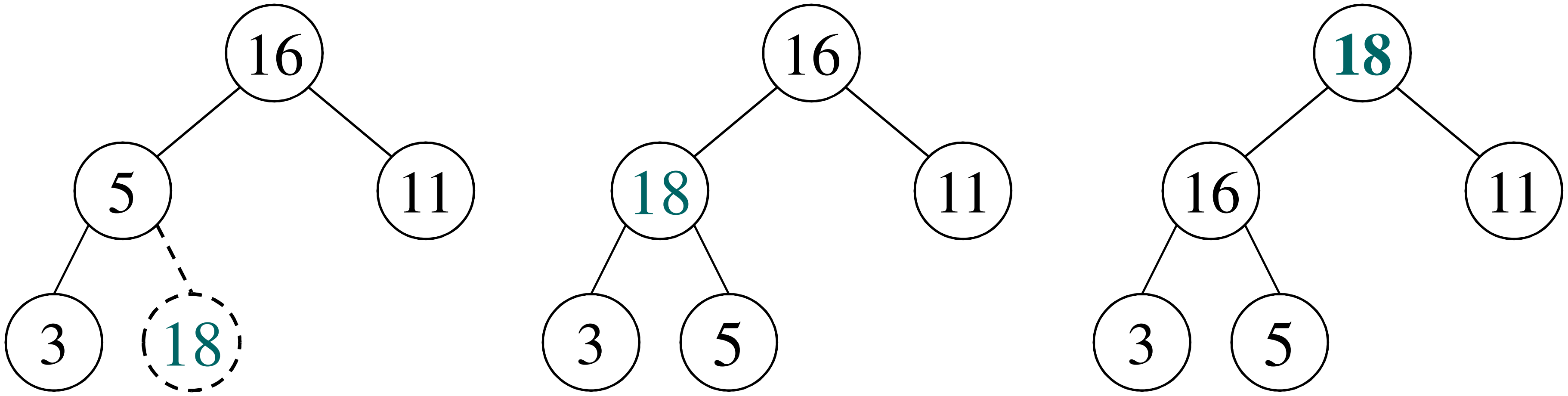
- to restore heap ordering property, the newly added element must be shifted upward ("bubbled up") until it reaches its proper place
  - bubble up (aka: "percolate up") by swapping with parent
  - how many bubble-ups could be necessary, at most?





Adding to a max-heap

- same operations, but must bubble up *larger* values to top

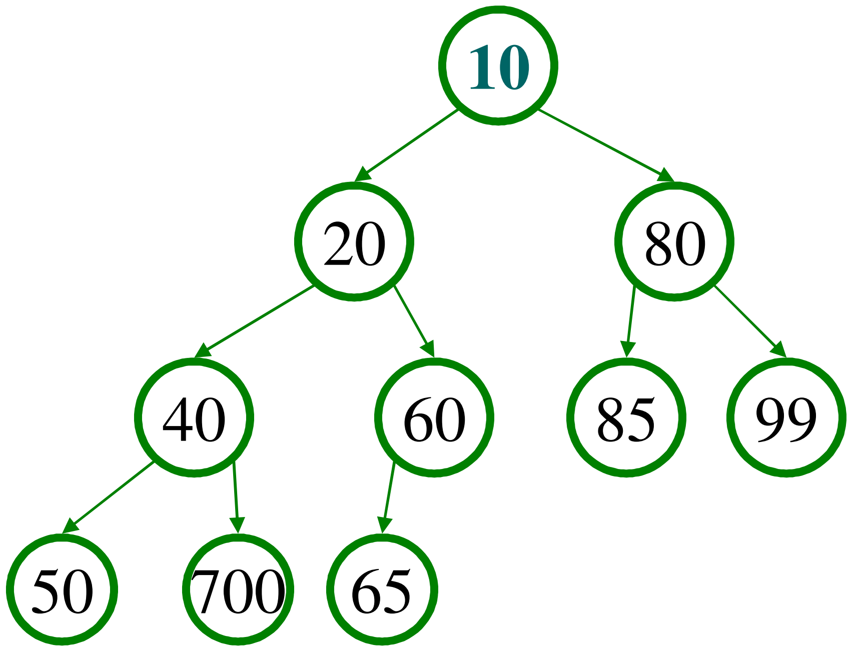


Heap practice problem

- Draw the state of the min-heap tree after adding the following elements to it:

  6, 50, 11, 25, 42, 20, 104, 76, 19, 55, 88, 2

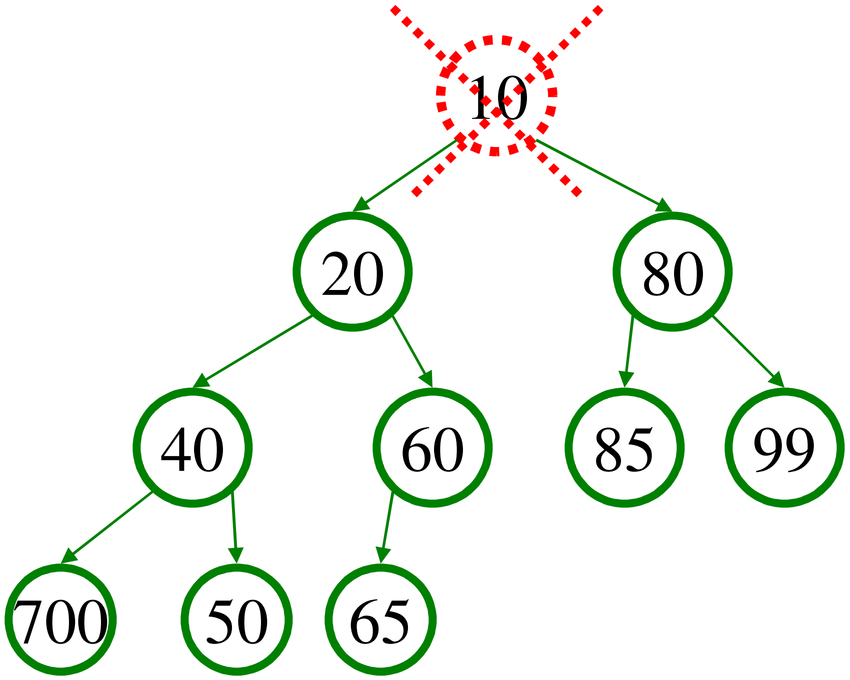
The `peek` operation

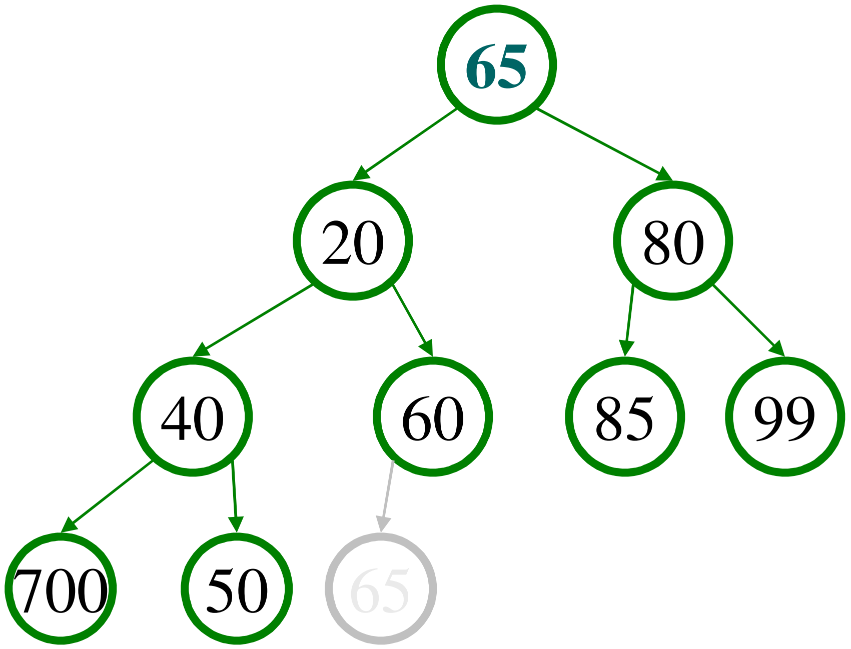


- `peek` on a min-heap is trivial; because of the heap properties, the minimum element is always the root
  - `peek` is O(1)
  - `peek` on a max-heap would be O(1) as well, but would return you the maximum element and not the minimum one

Removing from a min-heap

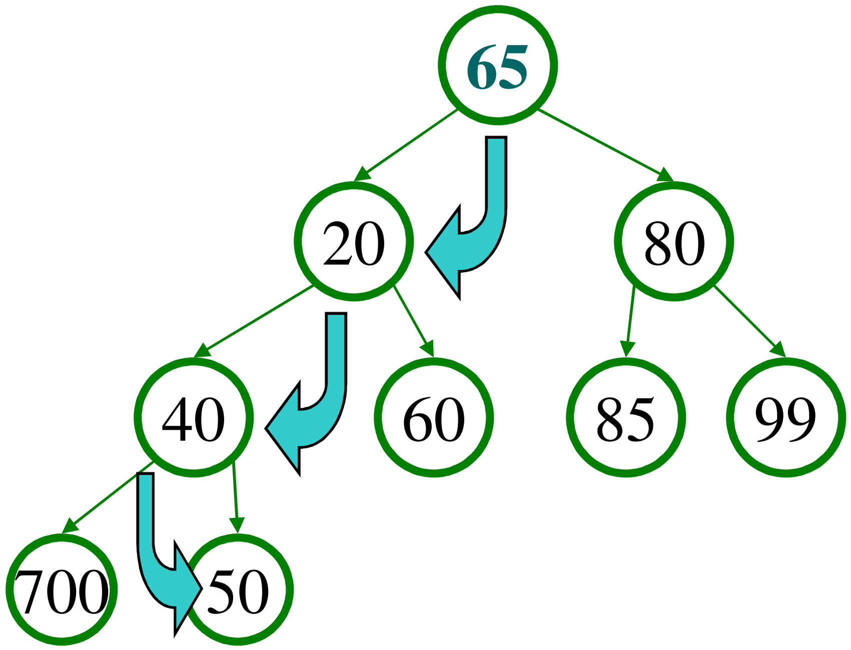
- min-heaps only support `remove` of the min element (the root)
  - must remove the root while maintaining heap completeness and ordering properties
  - intuitively, the last leaf must disappear to keep it a heap
  - initially, just swap root with last leaf (we'll fix it)

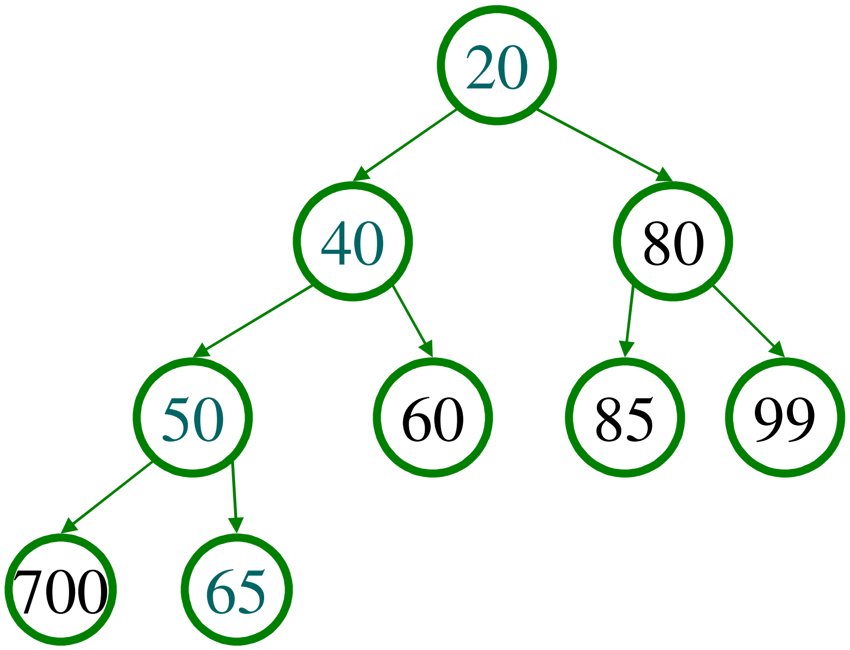




Removing from heap, cont'd.

- must fix heap-ordering property; root is out of order
  - shift the root downward ("bubble down") until it's in place
  - swap it with its smaller child each time
    - What happens if we don't always swap with the smaller child?





Heap practice problem

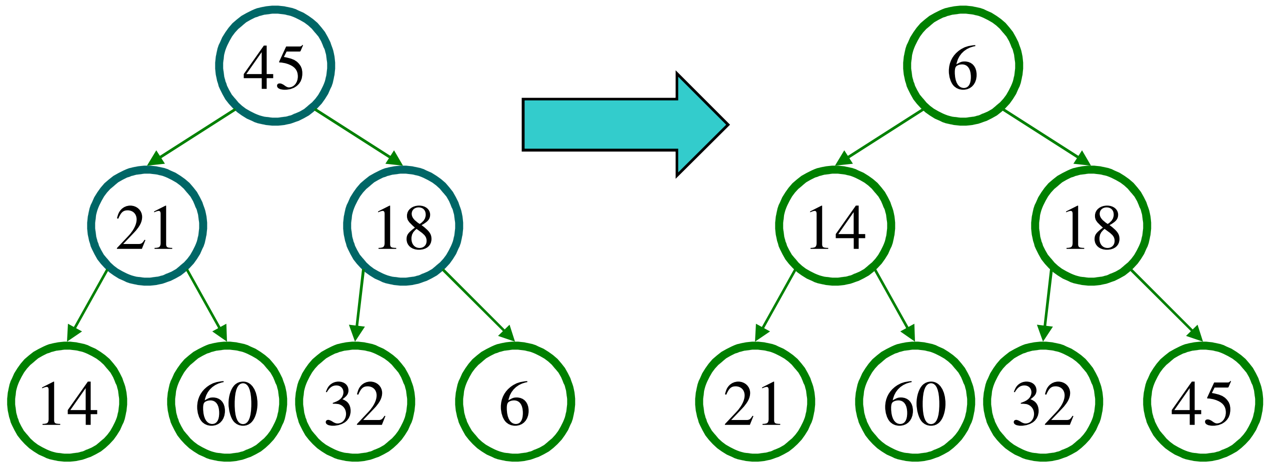
- Assuming that you have a heap with the following elements to it (from the last question):

  6, 50, 11, 25, 42, 20, 104, 76, 19, 55, 88, 2

- Show the state of the heap after remove() has been executed on it 3 times.

Turning any input into a heap

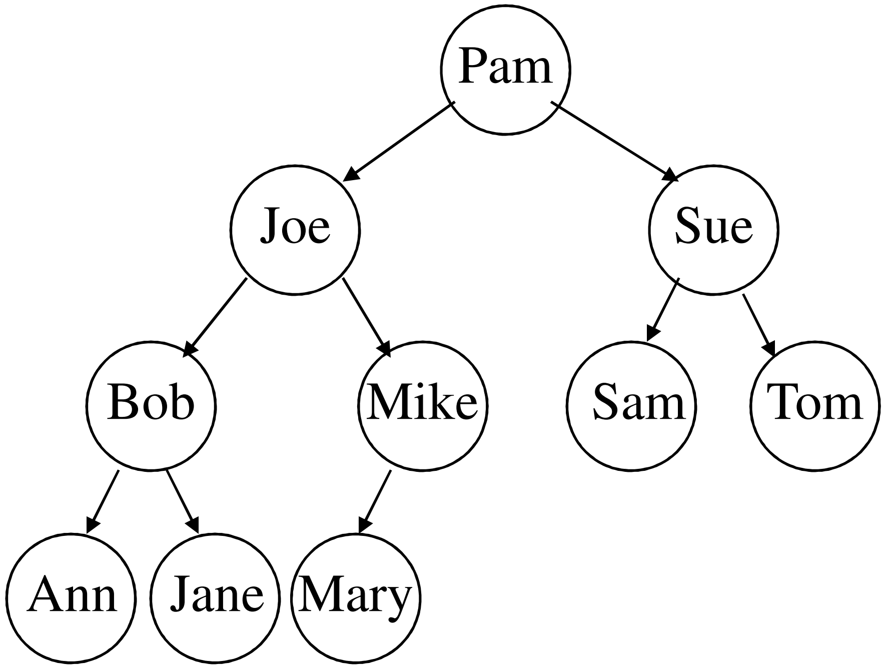
- we can quickly turn any complete tree of comparable elements into a heap with a `buildHeap` algorithm
- simply perform a "bubble down" operation on every node that is not a leaf, starting from the rightmost internal node and working back to the root
  - why does this `buildHeap` operation work?
  - how long does it take to finish? (big-Oh)



Array tree implementation

- corollary: a complete binary tree can be implemented using an array (the example tree shown is *not* a heap)

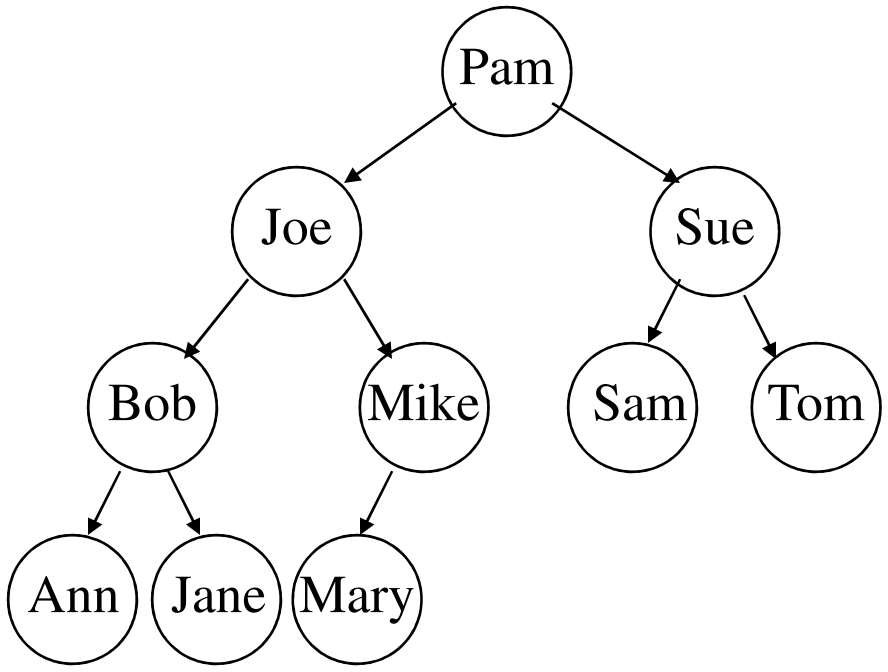
i	Item	Left Child	2*i	Right Child
1	Pam	2	2	3
2	Joe	4	4	5
3	Sue	6	6	7
4	Bob	8	8	9
5	Mike	10	10	-1
6	Sam	-1	11	-1
7	Tom	-1	13	-1
8	Ann	-1	15	-1
9	Jane	-1	17	-1
10	Mary	-1	19	-1



- LeftChild(i) = 2*i
- RightChild(i) = 2*i + 1

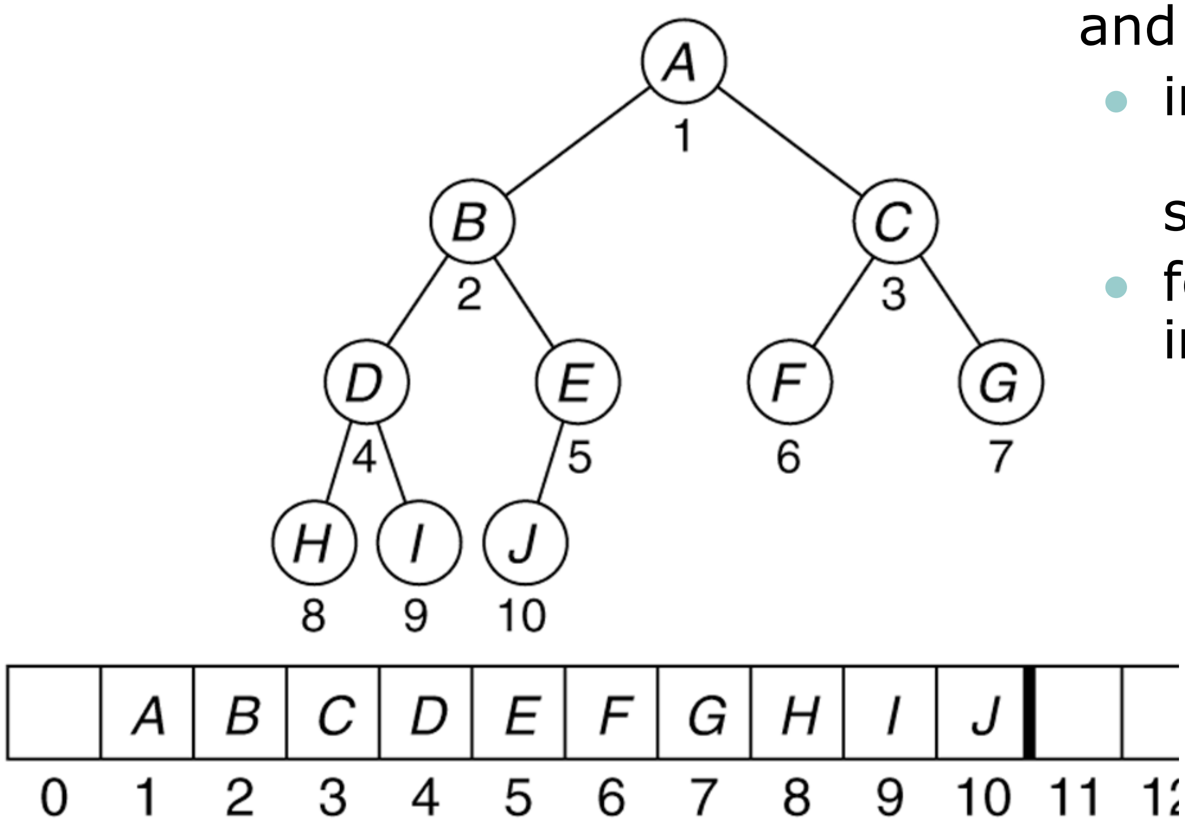
Array binary tree - parent

i	Item	Parent	i / 2
1	Pam	-1	0
2	Joe	1	1
3	Sue	1	1
4	Bob	2	2
5	Mike	2	2
6	Sam	3	3
7	Tom	3	3
8	Ann	4	4
9	Jane	4	4
10	Mary	5	5



- Parent(i) = $\lfloor i / 2 \rfloor$

Implementation of a heap



when implementing a complete binary tree, we actually can "cheat" and just use an array

- index of root = 1 (leave 0 empty for simplicity)
- for any node *n* at index *i*,
  - index of *n*.left = 2*i*
  - index of *n*.right = 2*i* + 1

Advantages of array heap

the "implicit representation" of a heap in an array makes several operations very fast

- add a new node at the end (O(1))
- from a node, find its parent (O(1))
- swap parent and child (O(1))
- a lot of dynamic memory allocation of tree nodes is avoided
- the algorithms shown usually have elegant solutions