ADT and Data structures.
Priority Queue

[Review 02.02]

by Marina Barsky

[Example 3. Priority Queue ADT]

- A ***priority queue*** is a generalization of a *queue* where each element is assigned a priority and elements come out in order of priority
- If the priority is the earliest time they were added to the queue then priority queue becomes a regular queue
- We are interested in a case when priority of each element is not related to the time when the element was added to the queue

Specification of Priority Queue ADT

Priority Queue is an **Abstract Data Type** supporting the following main operations:

- ➔ ***top()*** **-** get an element with the highest priority
- ➔ ***enqueue(e,p)**** - adds a new element with priority *p*
- ➔ ***dequeue()*** **-** removes and returns the element with the highest priority

---

*To simplify the discussion we use *enqueue(e),* where *e* is a number which reflects the priority

Priority Queue: possible Data Structures

	enqueue	dequeue
Unsorted array/list	O(1)	O(n)
Sorted array/list	O(n)	O(1)

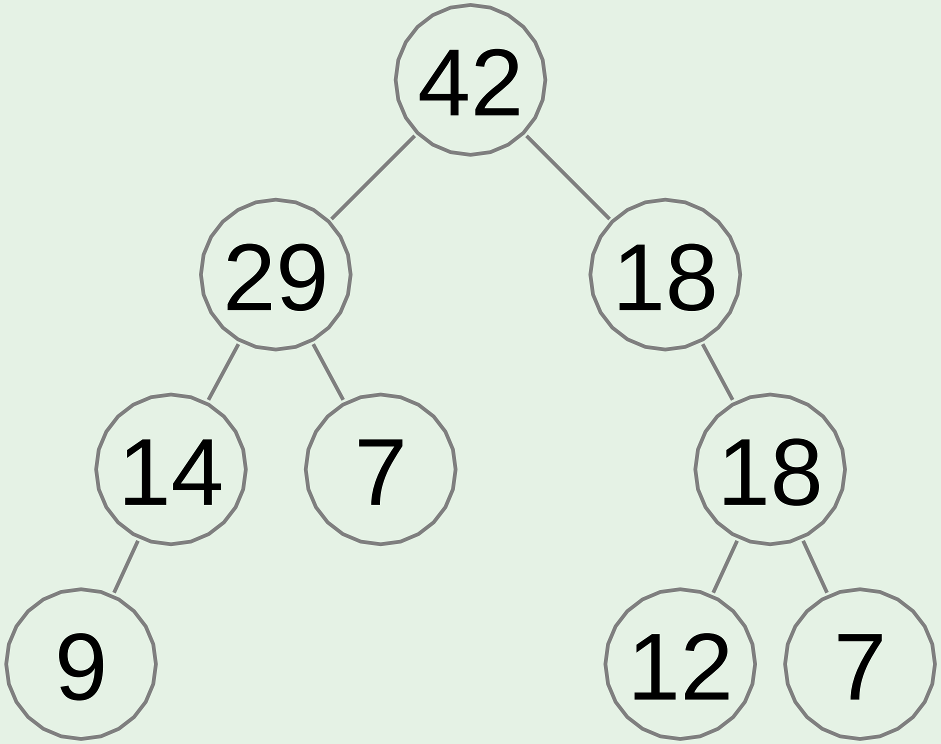
[Binary max-heap]

Definition

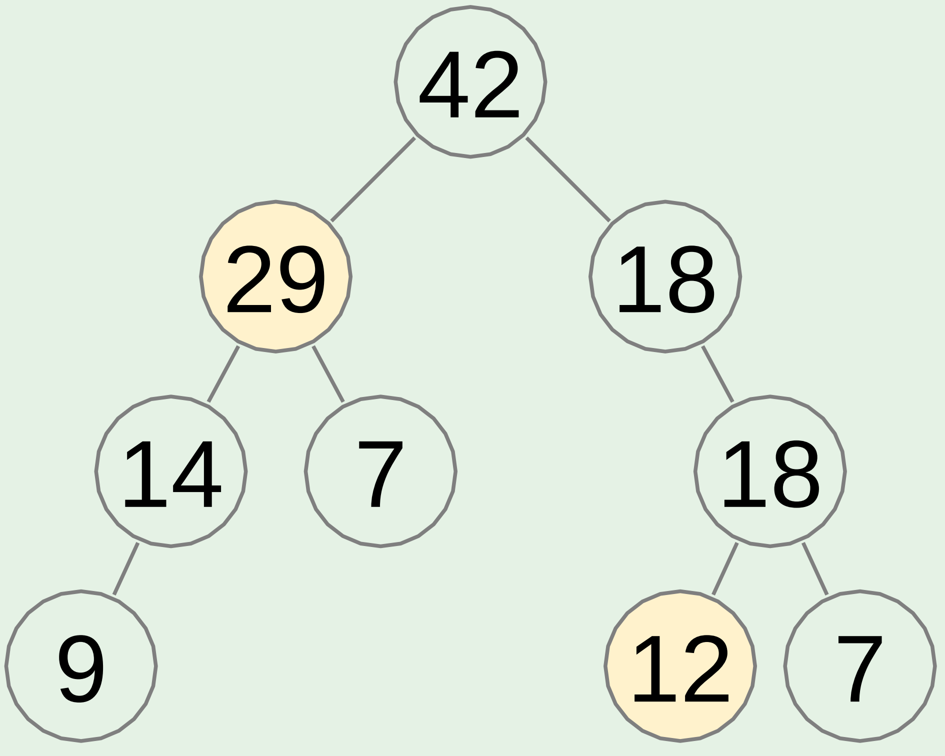
Binary max-heap is a **binary** tree (each node has zero, one, or **two** children) where the value of each node is at least the values of its children.

https://visualgo.net/en/heap?slide=1

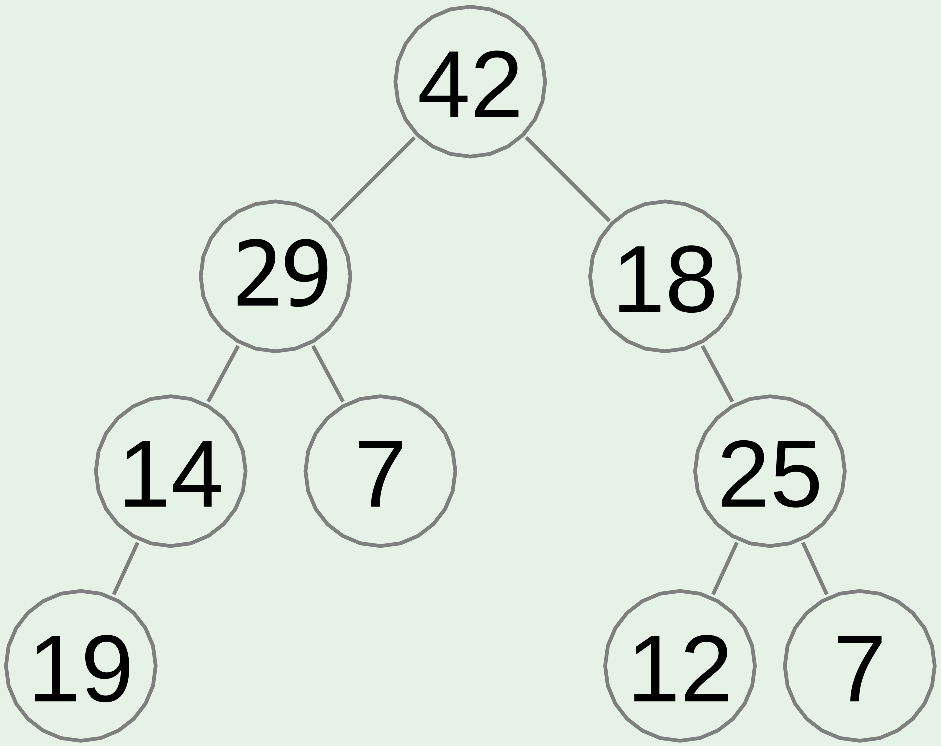
Heap?



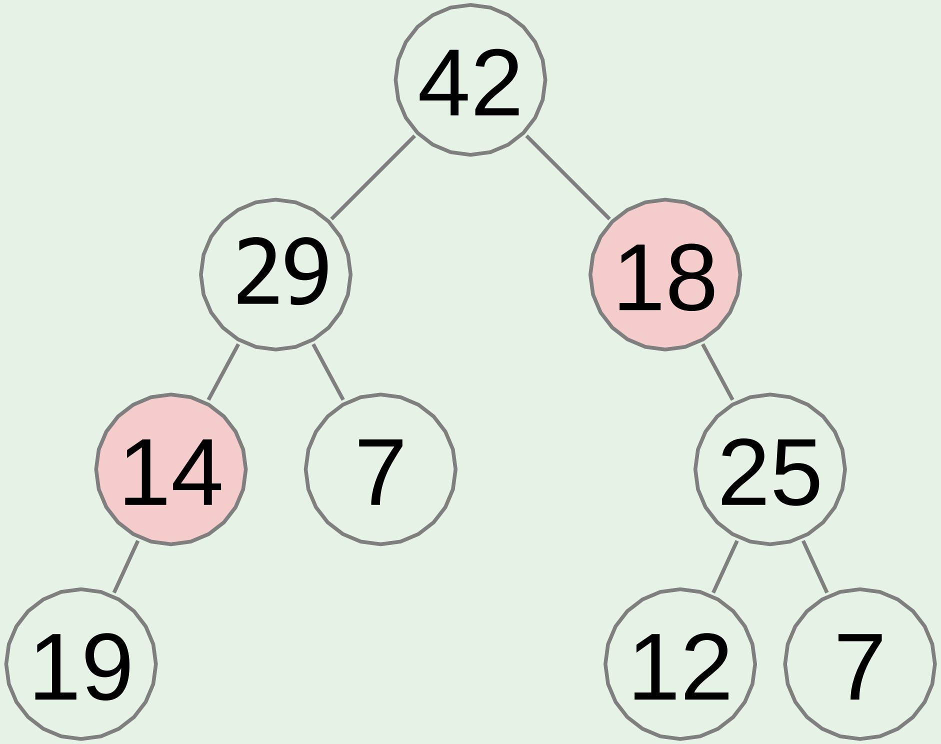
Heap? Yes



Heap?

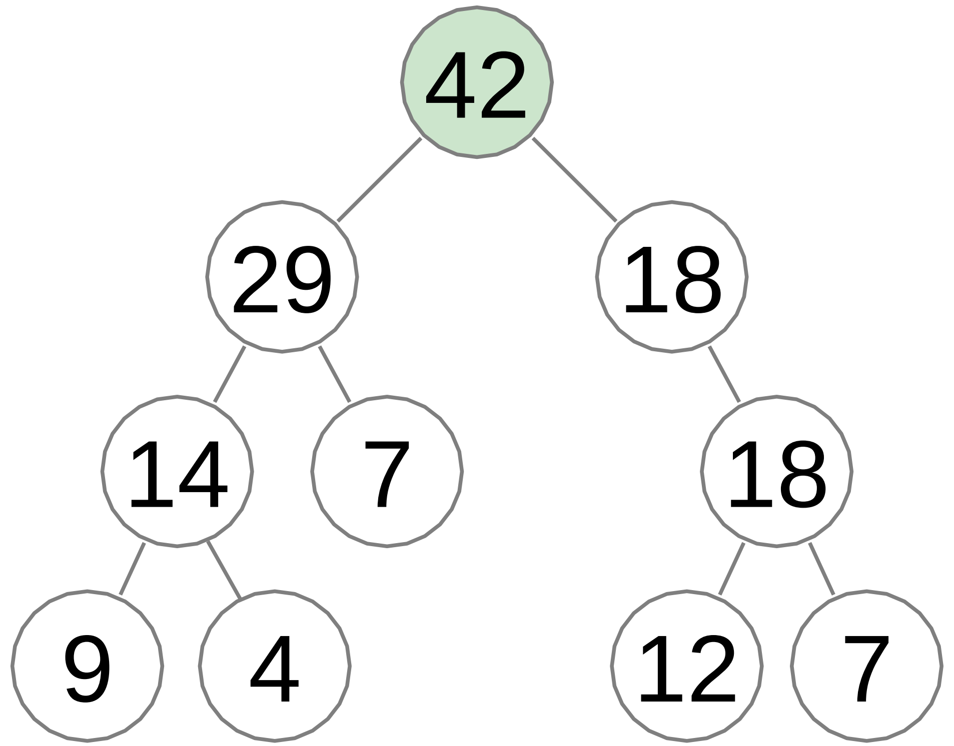


Heap? No



Heap operations: `top`

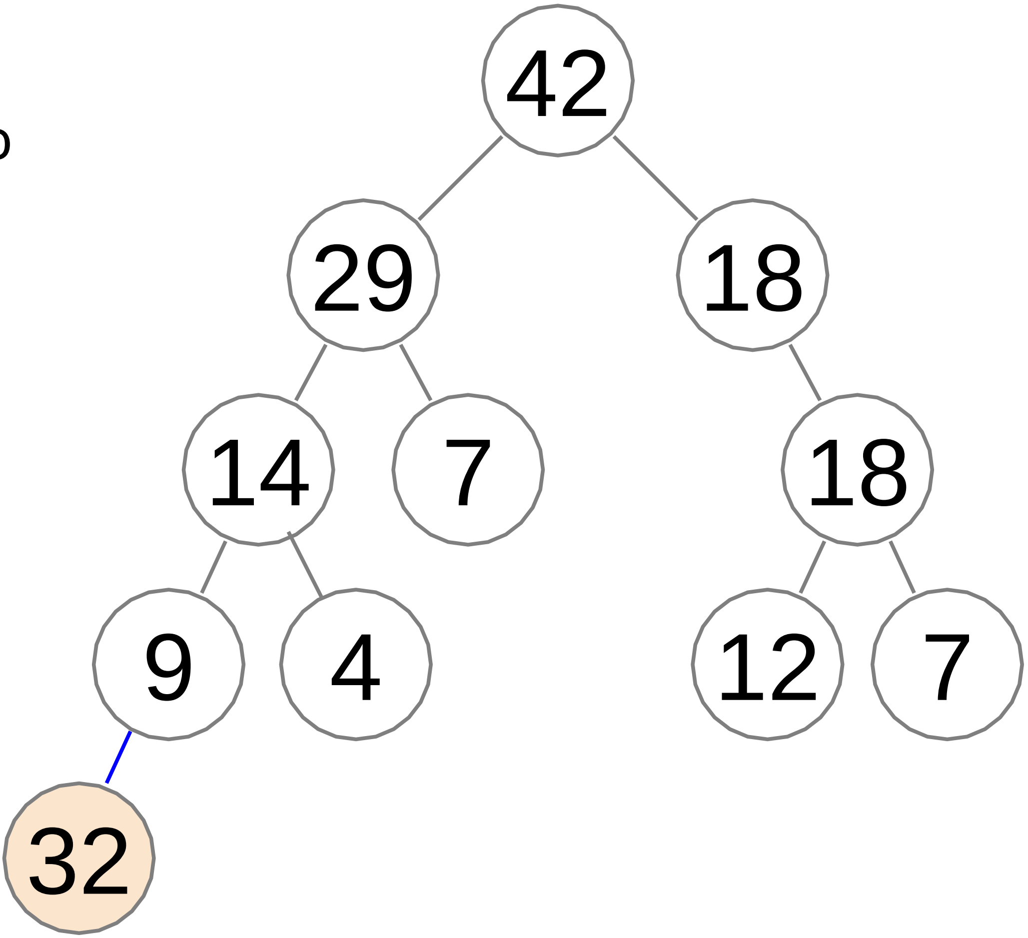
return the root value



Run-time: O(1)

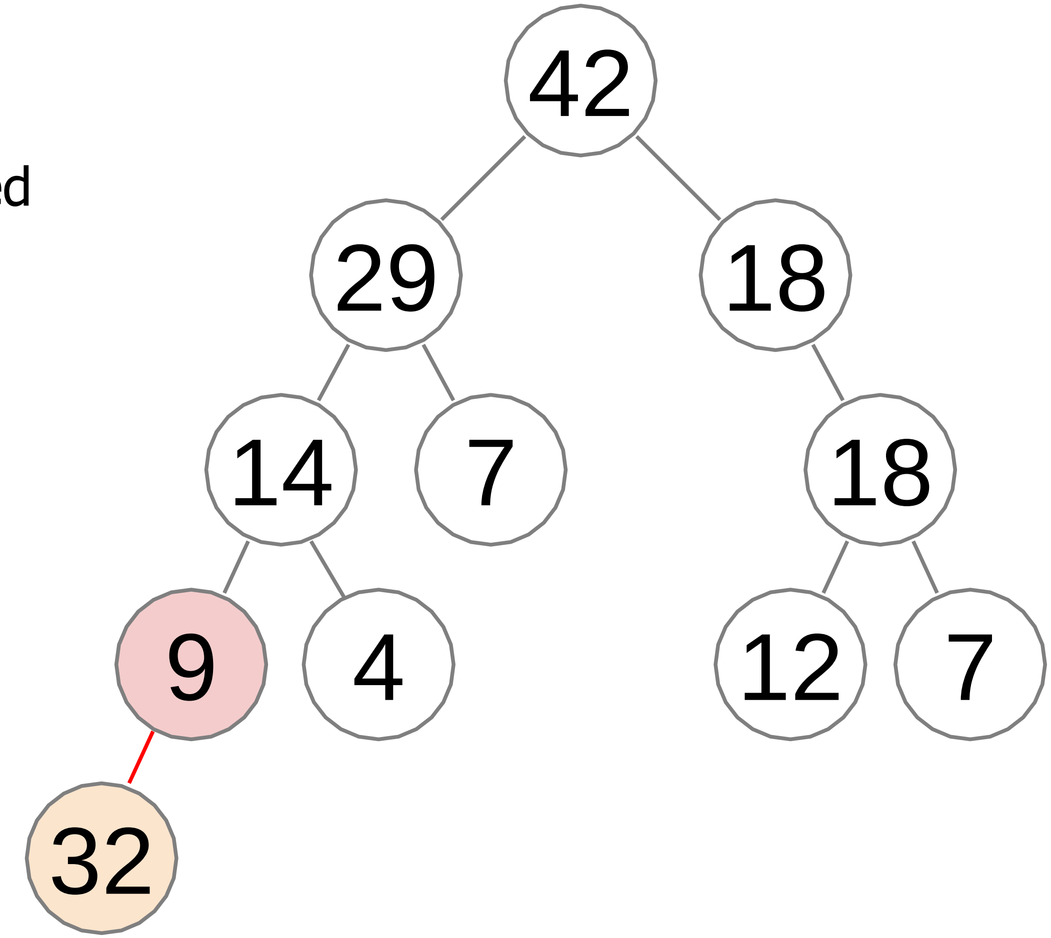
Heap operations: ***enqueue(e)***

attach a new node to
any leaf



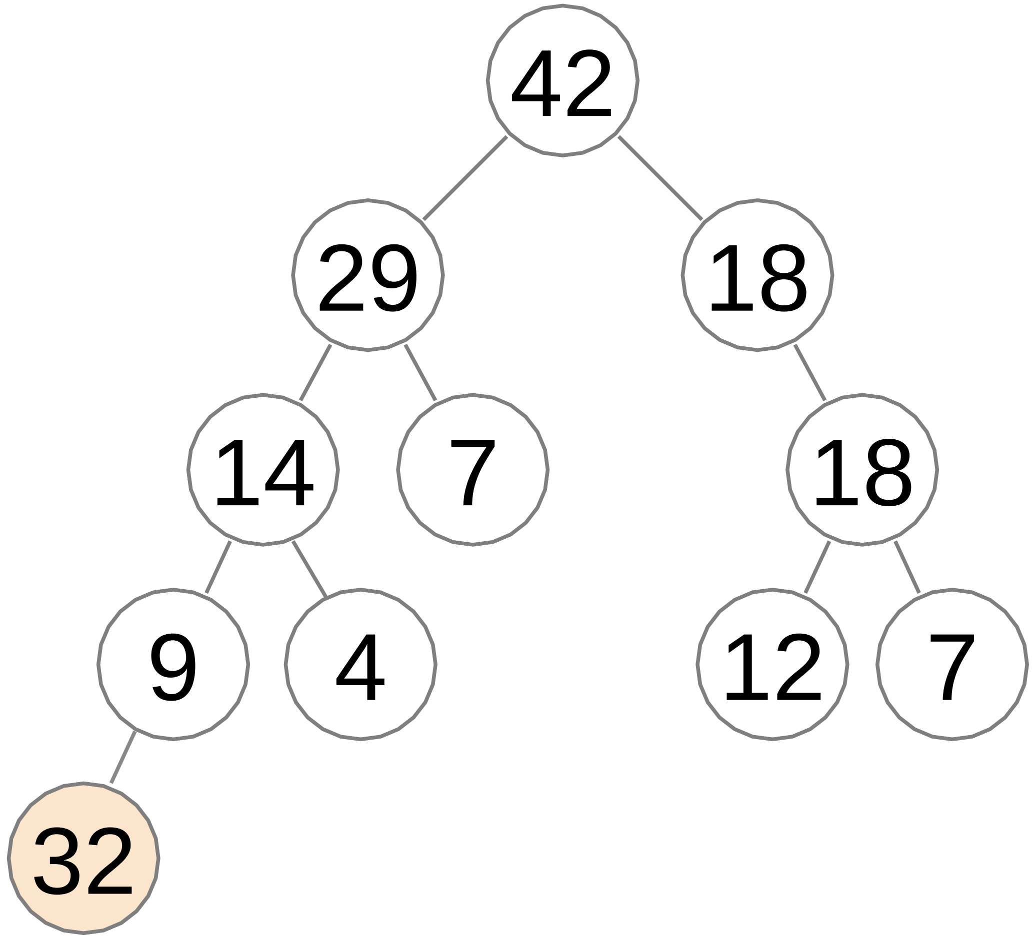
Heap operations: `enqueue(e)`

the heap property
may become violated



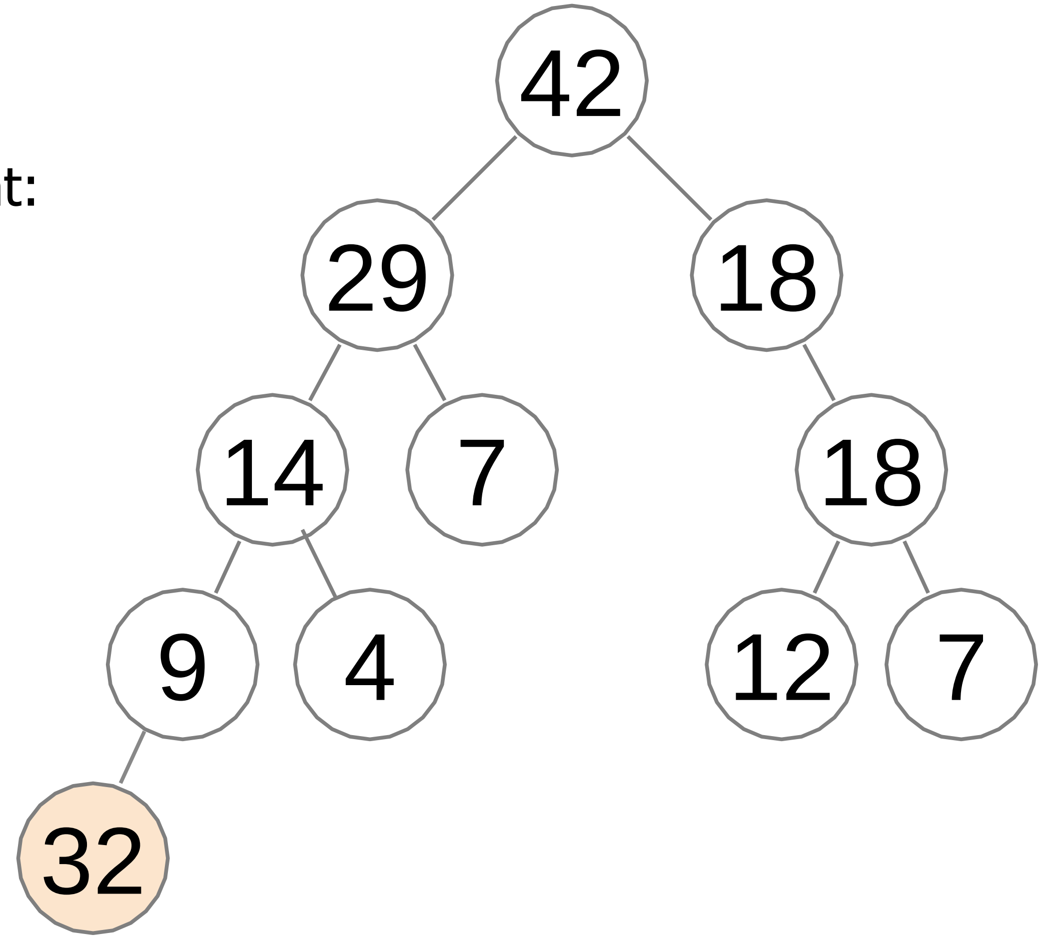
Heap operations: *enqueue(e)*

to fix that we let the
new node *sift up*



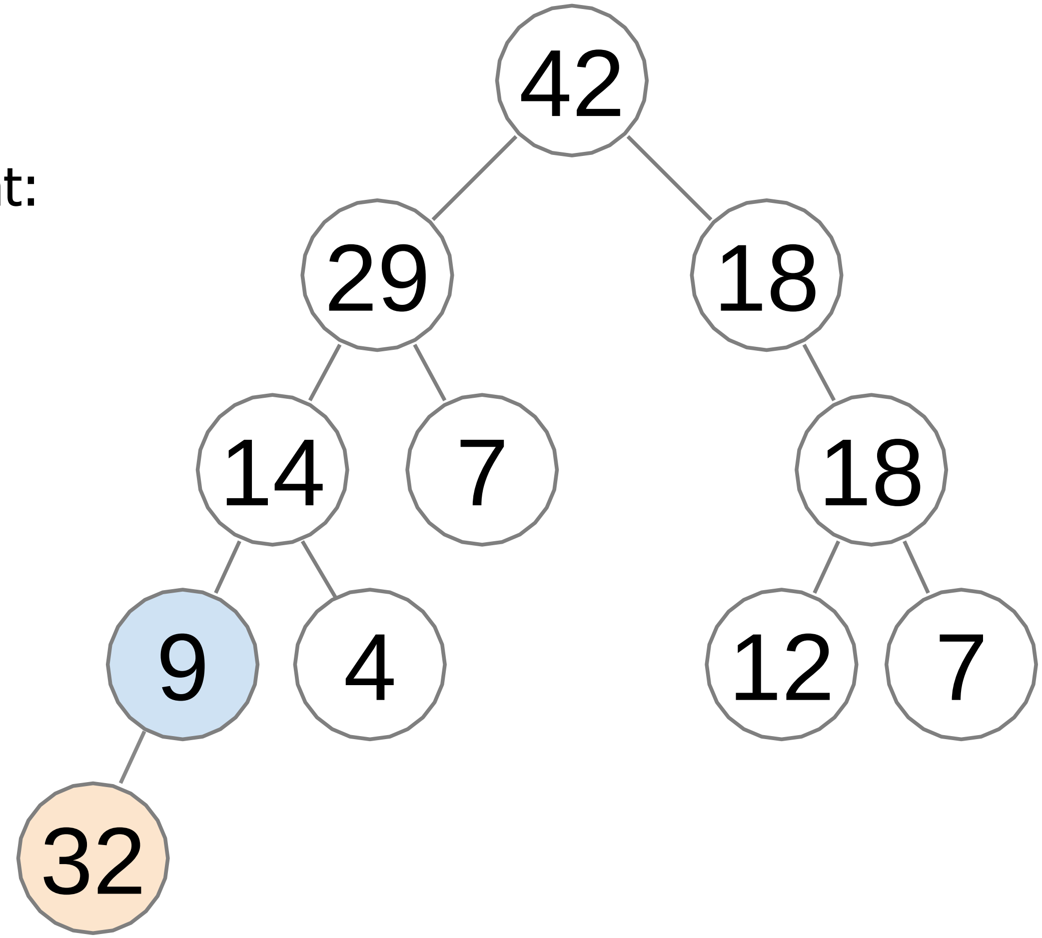
Heap operations: `sift_up(e)`

if current element is
bigger than the parent:
swap



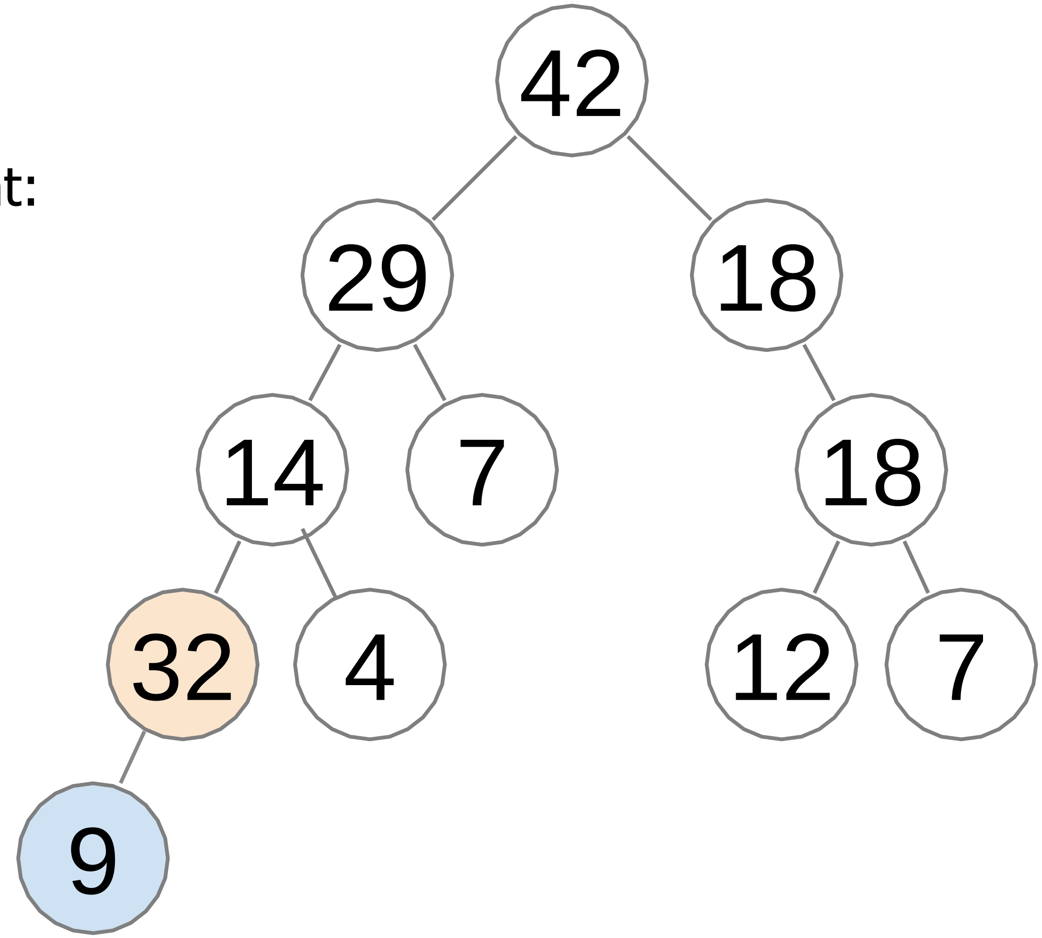
Heap operations: *sift_up(e)*

if current element is
bigger than the parent:
swap



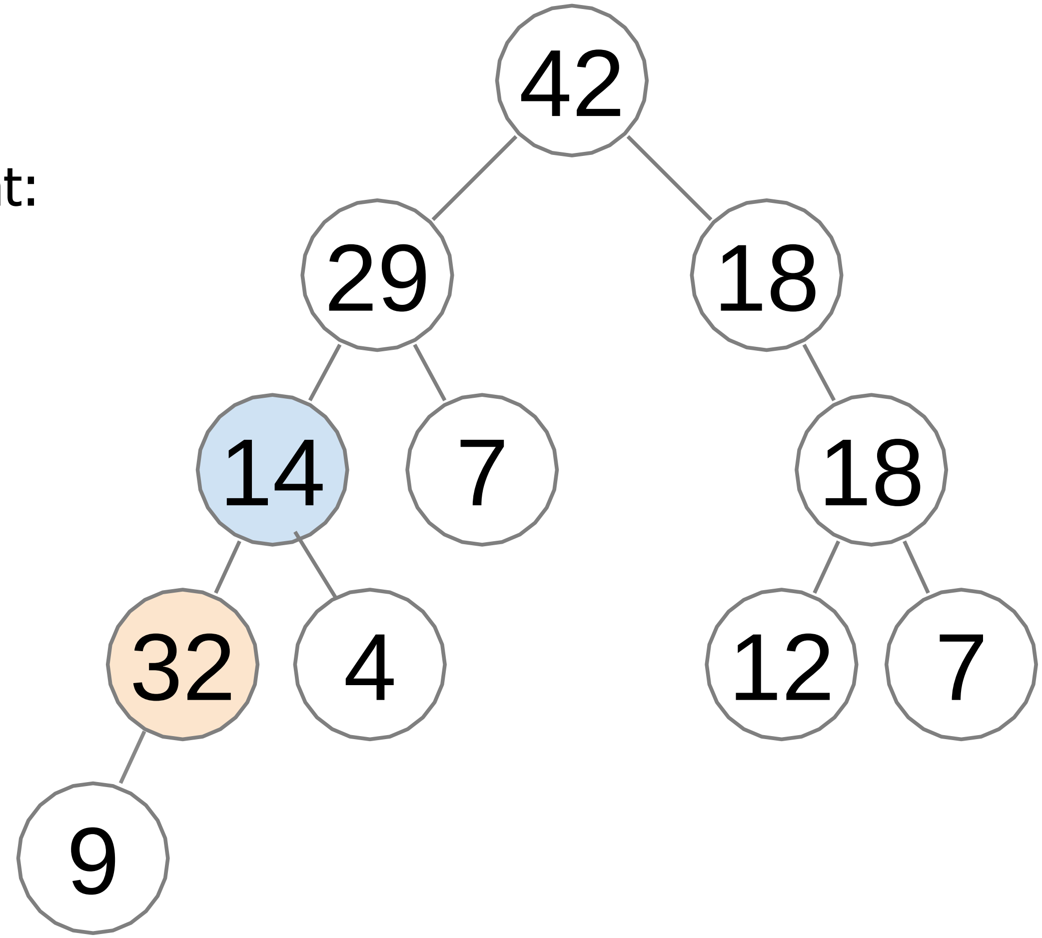
Heap operations: ***sift_up(e)***

if current element is bigger than the parent: *swap*



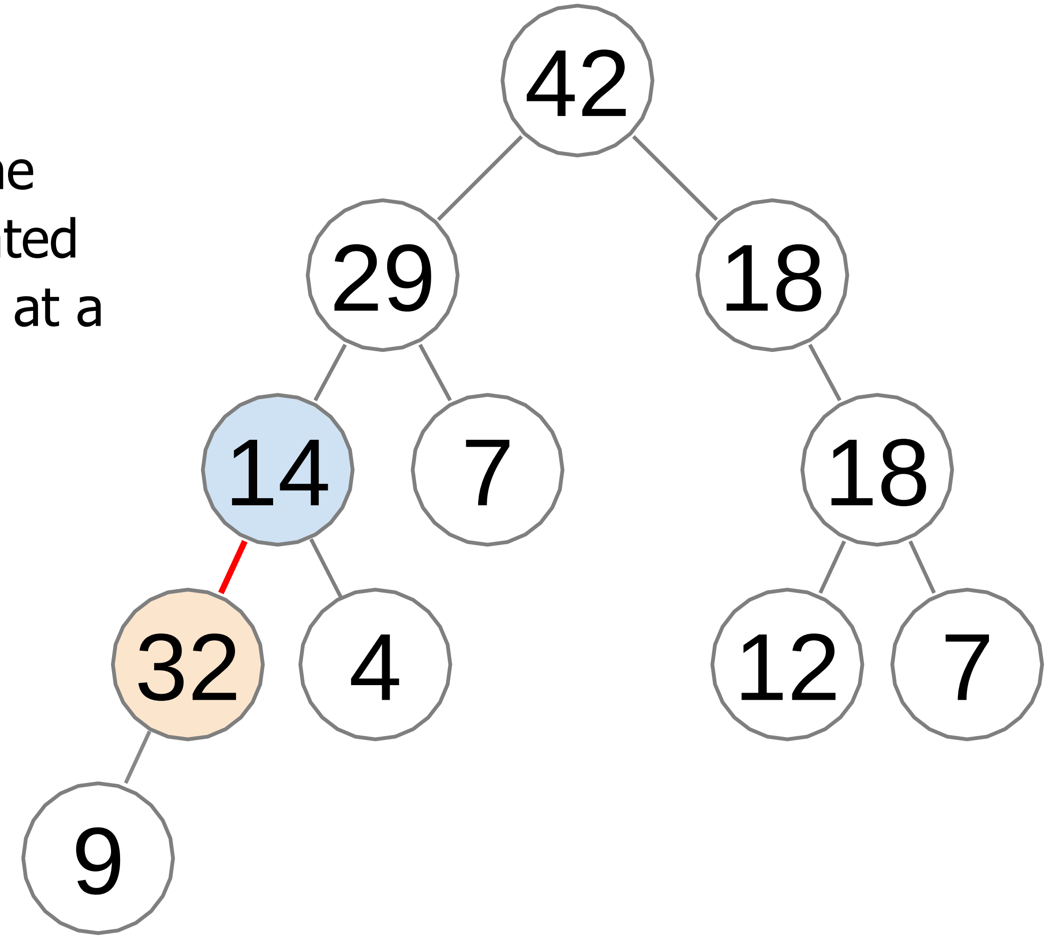
Heap operations: ***sift_up(e)***

if current element is
bigger than the parent:
swap



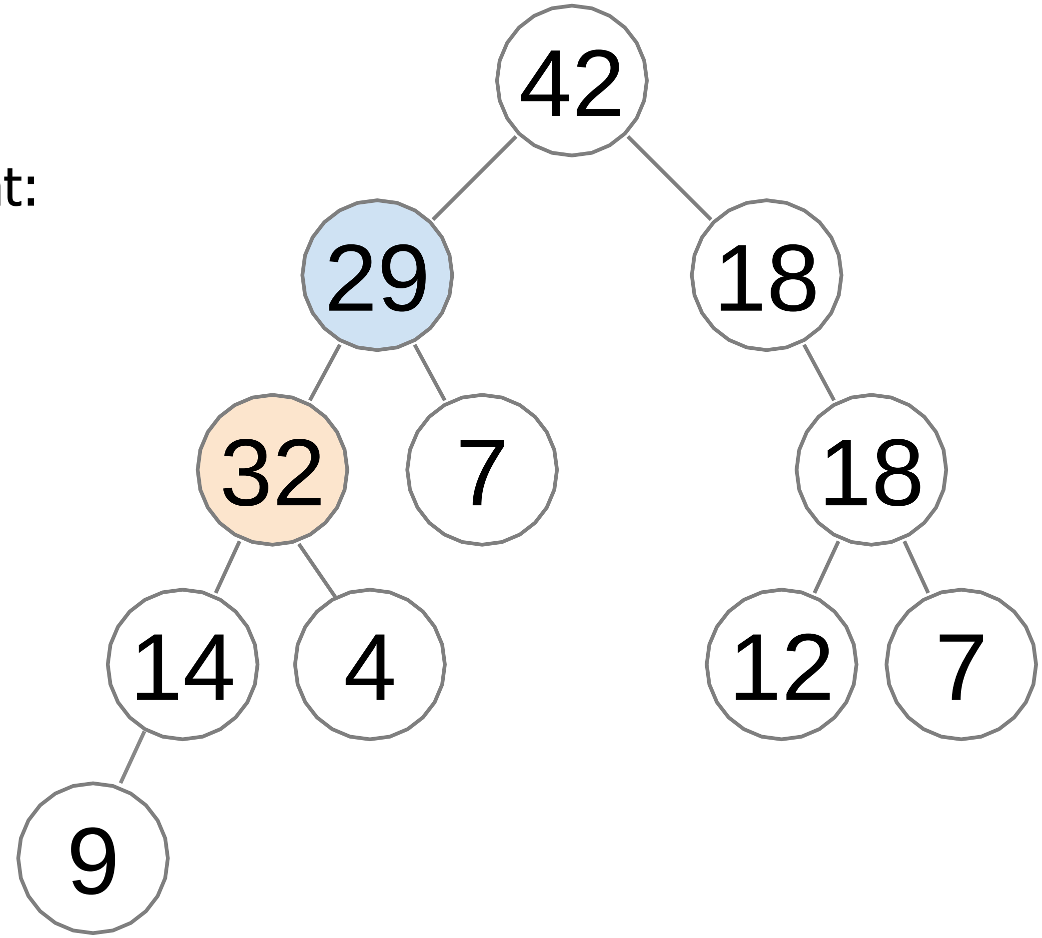
Heap operations: ***sift_up(e)***

this works because the heap property is violated only on a single edge at a time



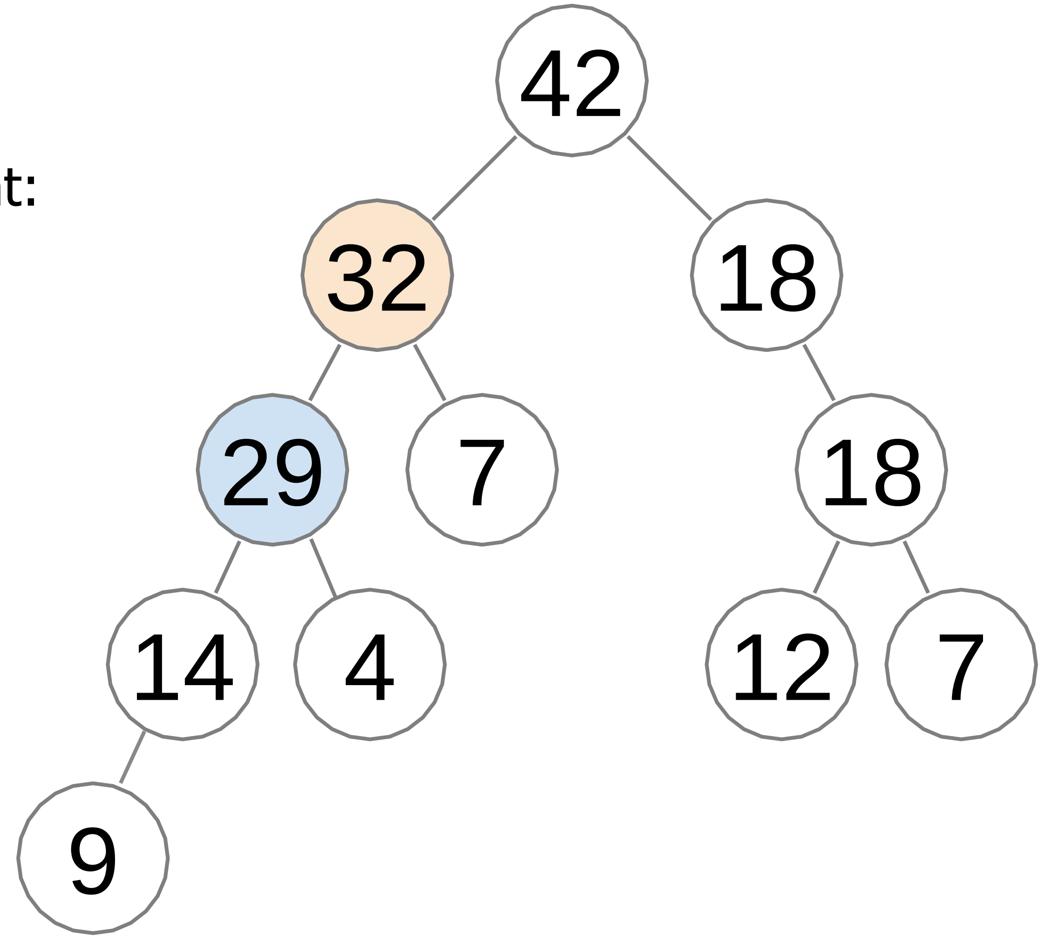
Heap operations: `sift_up(e)`

if current element is
bigger than the parent:
swap



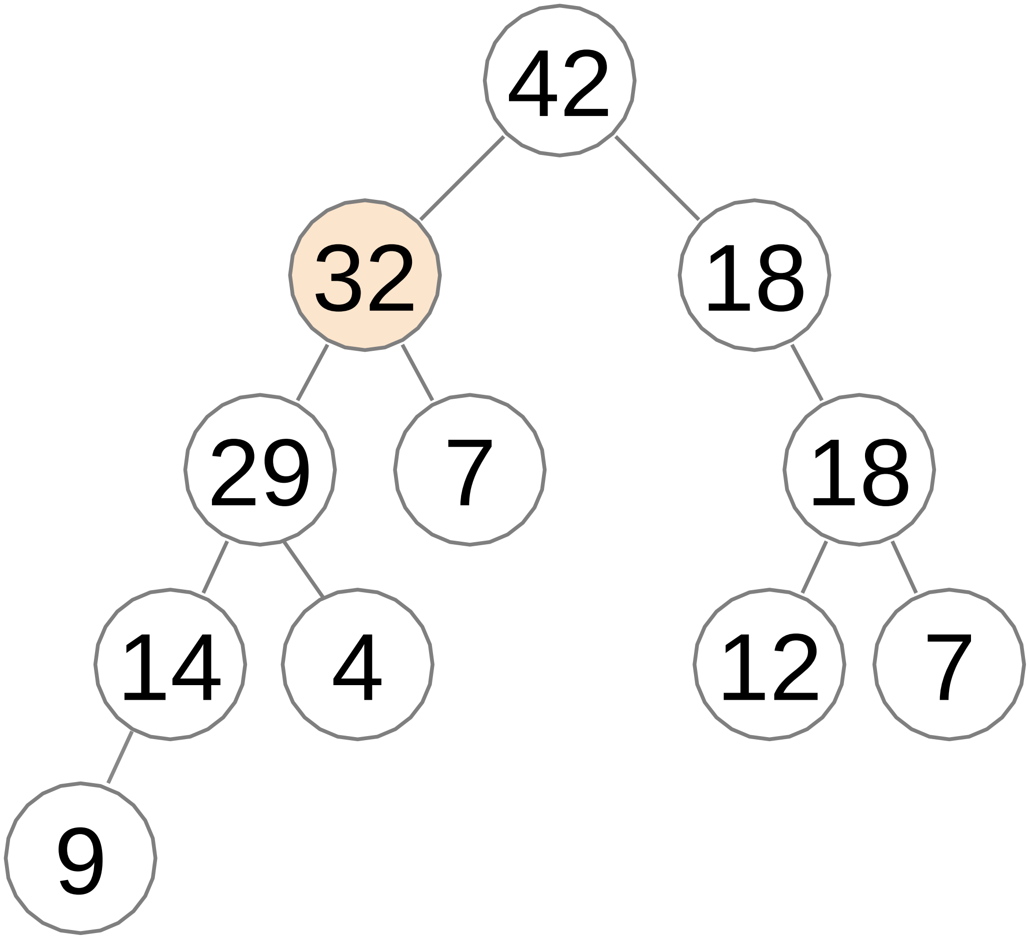
Heap operations: ***`sift_up(e)`***

if current element is
bigger than the parent:
swap



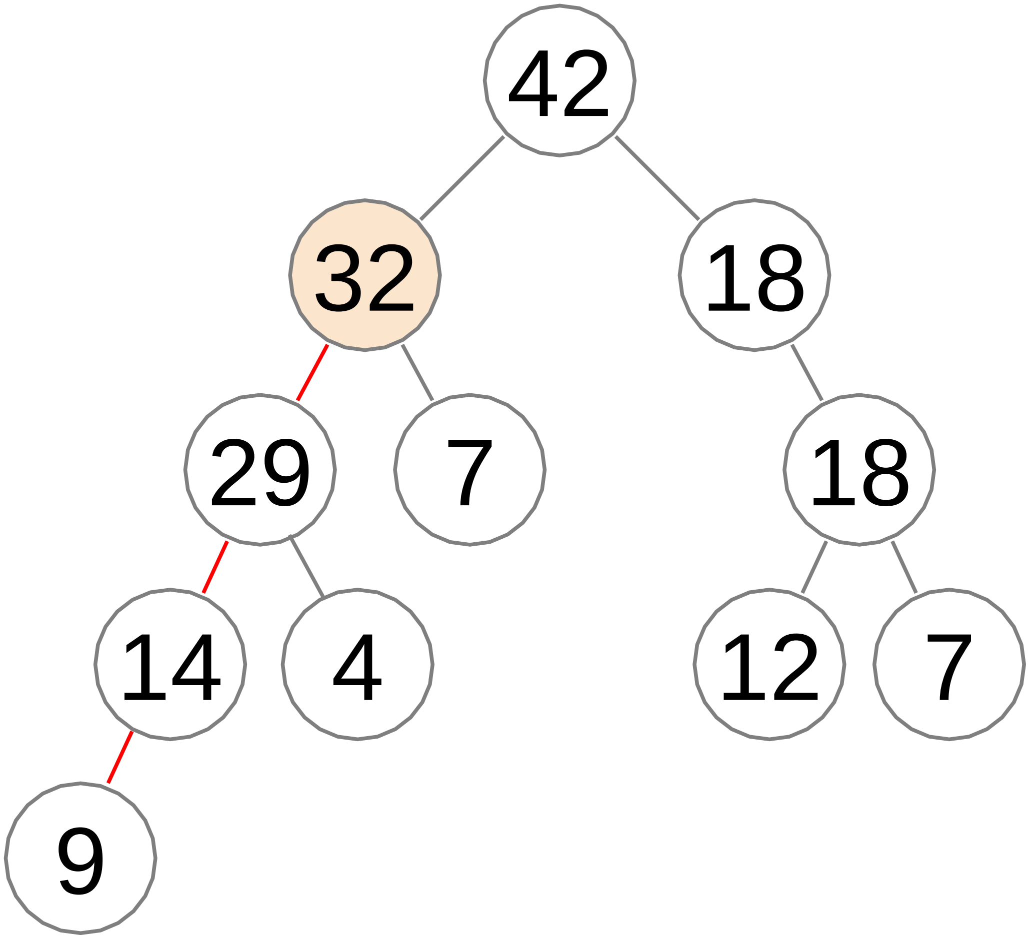
Heap operations: `sift_up(e)`

heap property is restored



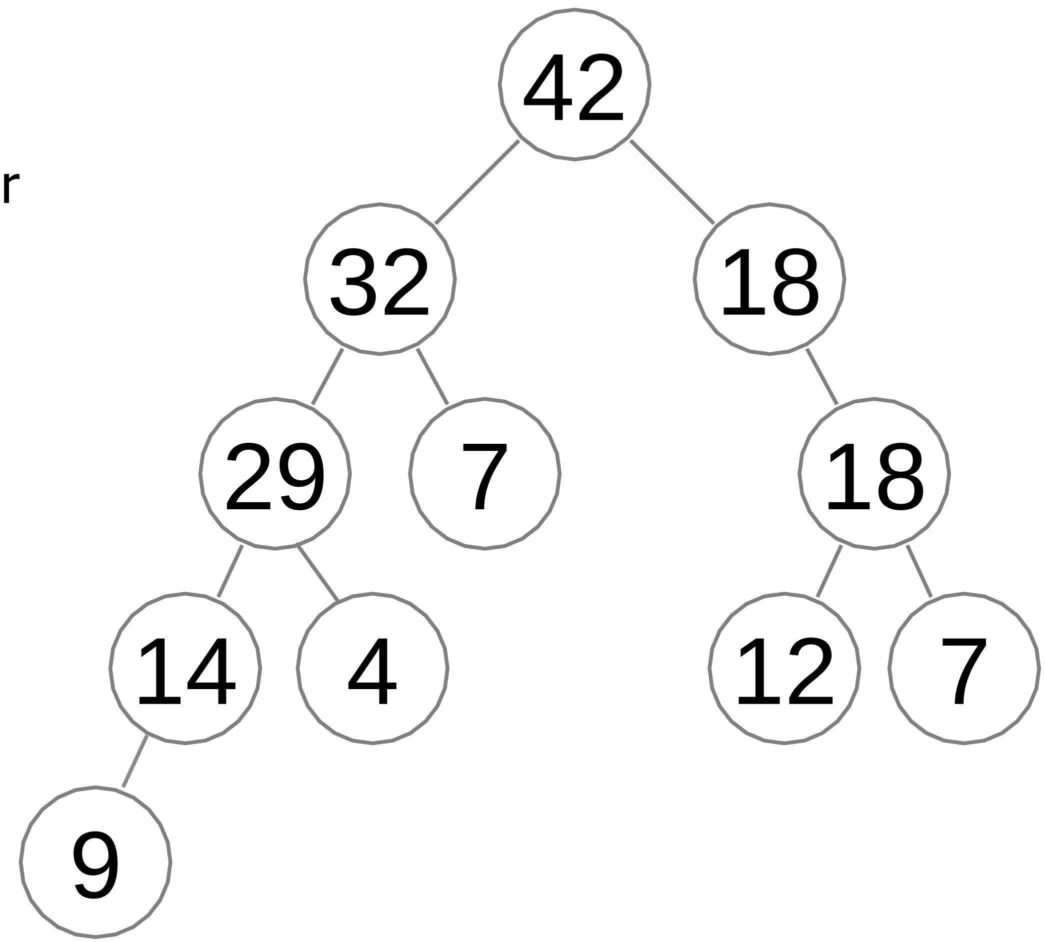
Heap operations: ***enqueue (e)***

running time of *enqueue* depends on how many times we need to *swap*



Heap operations: ***enqueue (e)***

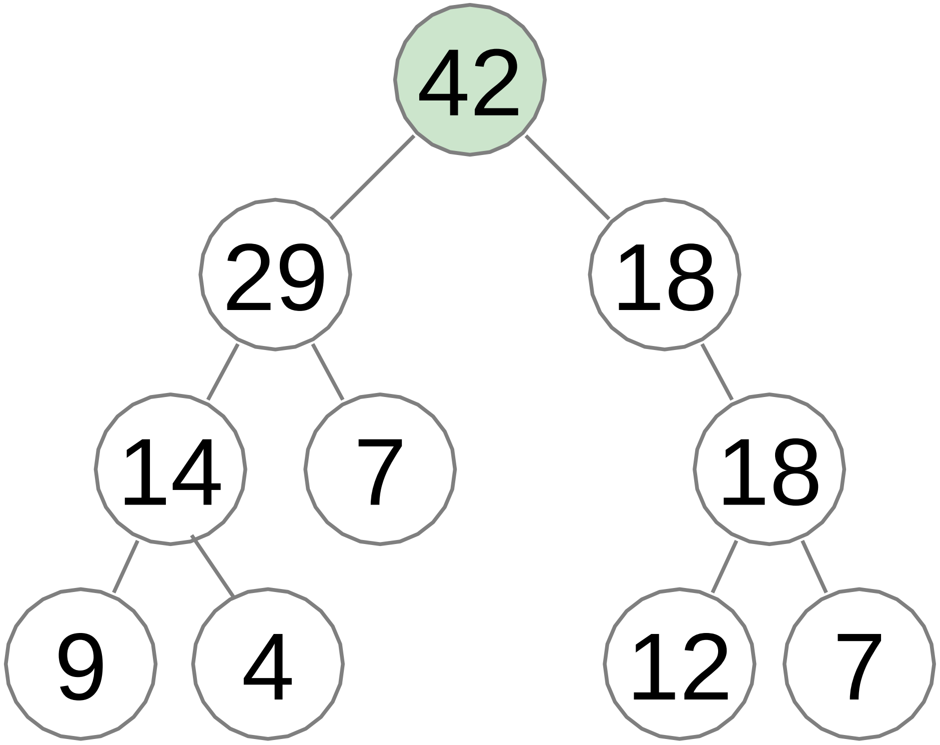
with each swap, the problematic node moves one node closer to the root



running time: *O*(tree height)

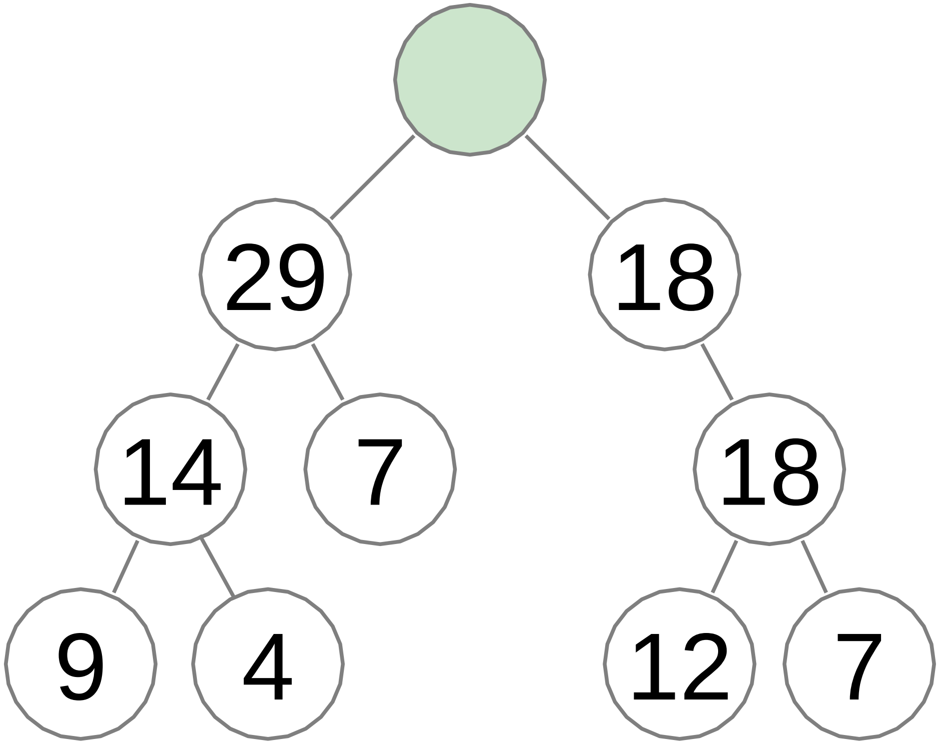
Heap operations: *dequeue*

remove and return the root value



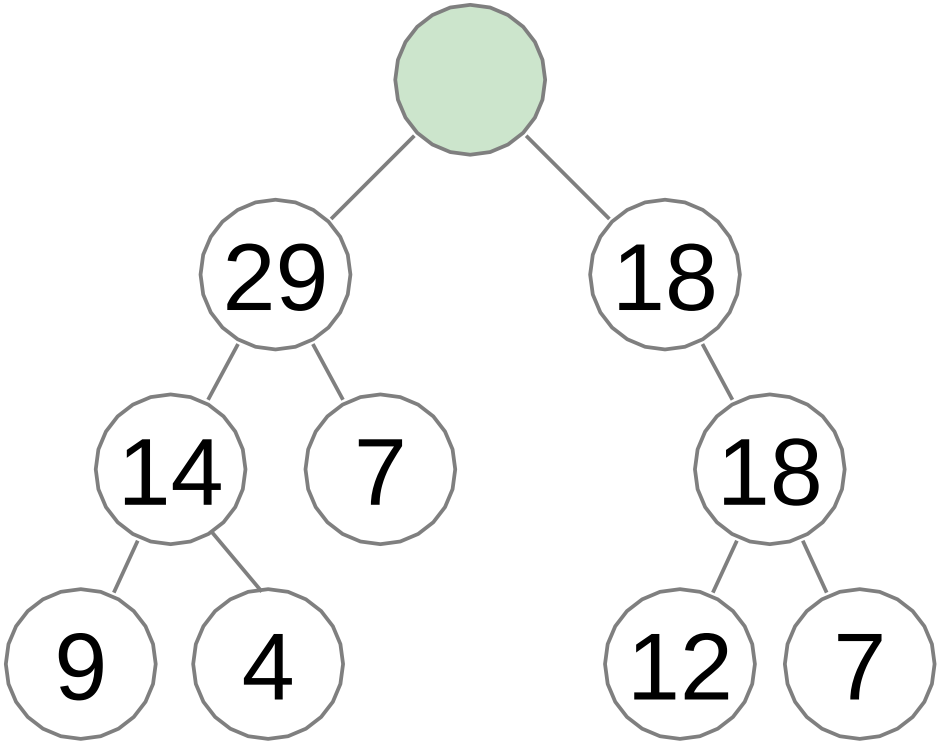
Heap operations: ***dequeue***

remove the root value



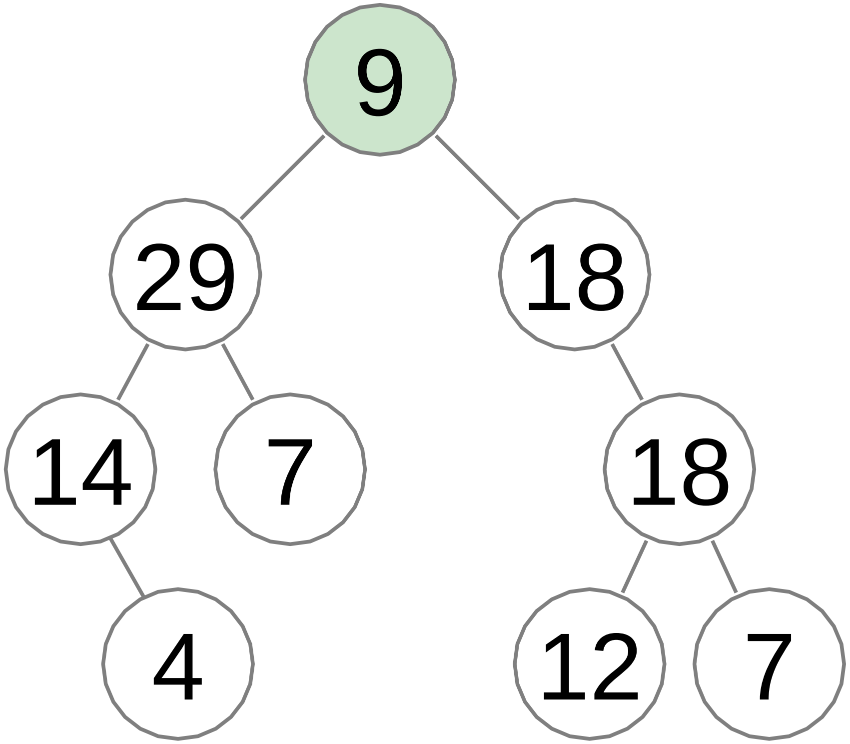
Heap operations: ***dequeue***

replace the empty node value with any leaf node value and remove the leaf



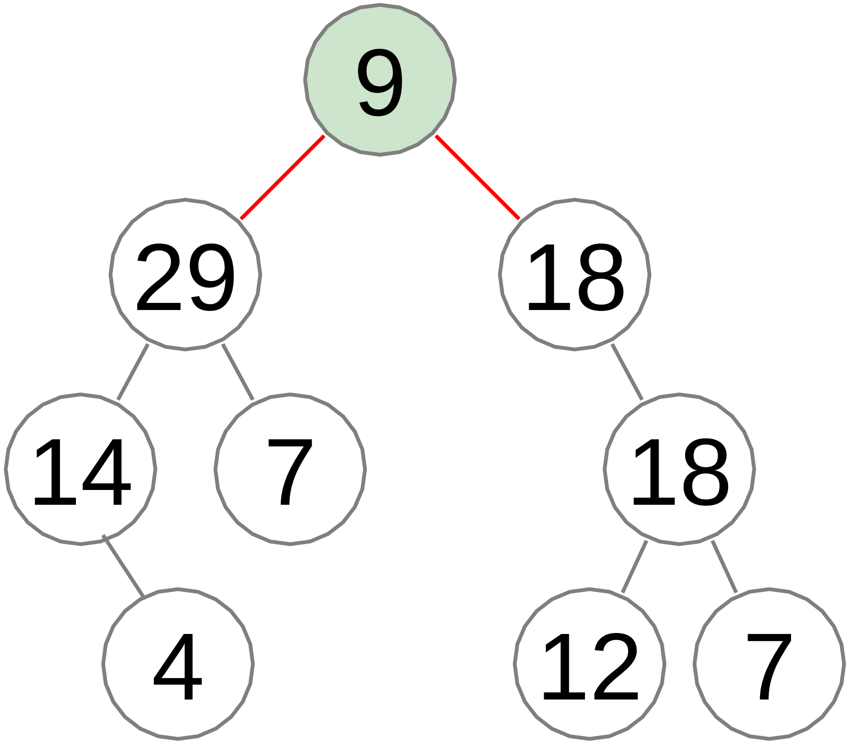
Heap operations: ***dequeue***

replace the empty node value with any leaf node value and remove the leaf



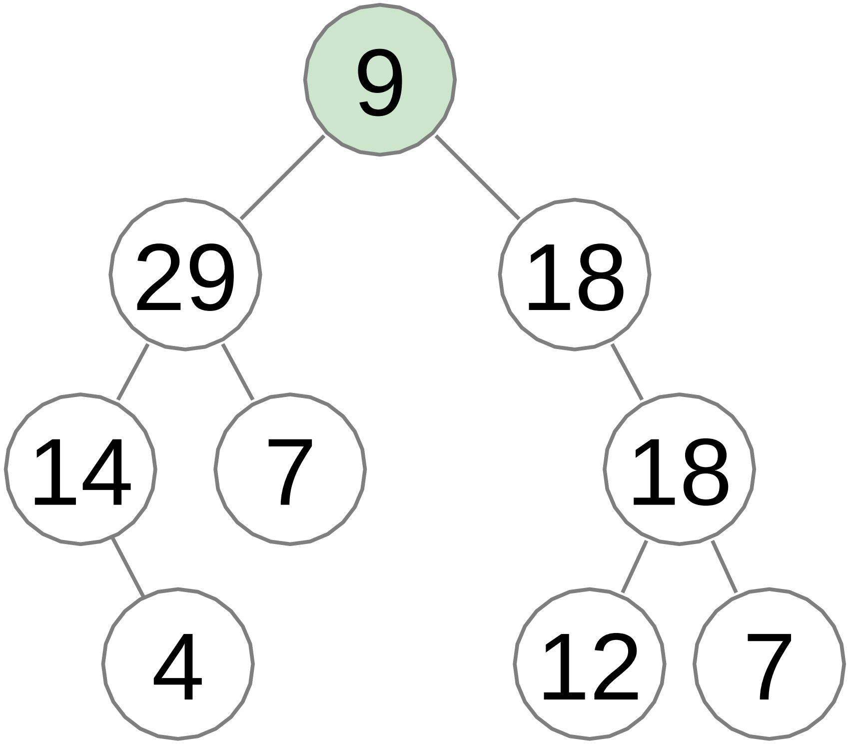
Heap operations: ***dequeue***

again, this may violate the heap property



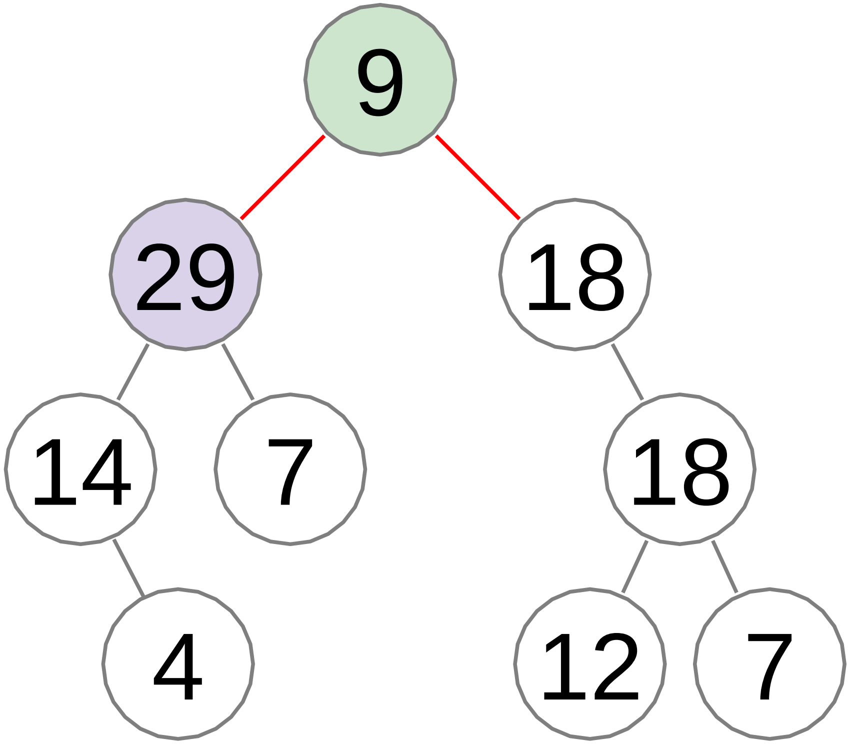
Heap operations: ***dequeue***

to fix it we let the problematic node *sift down*



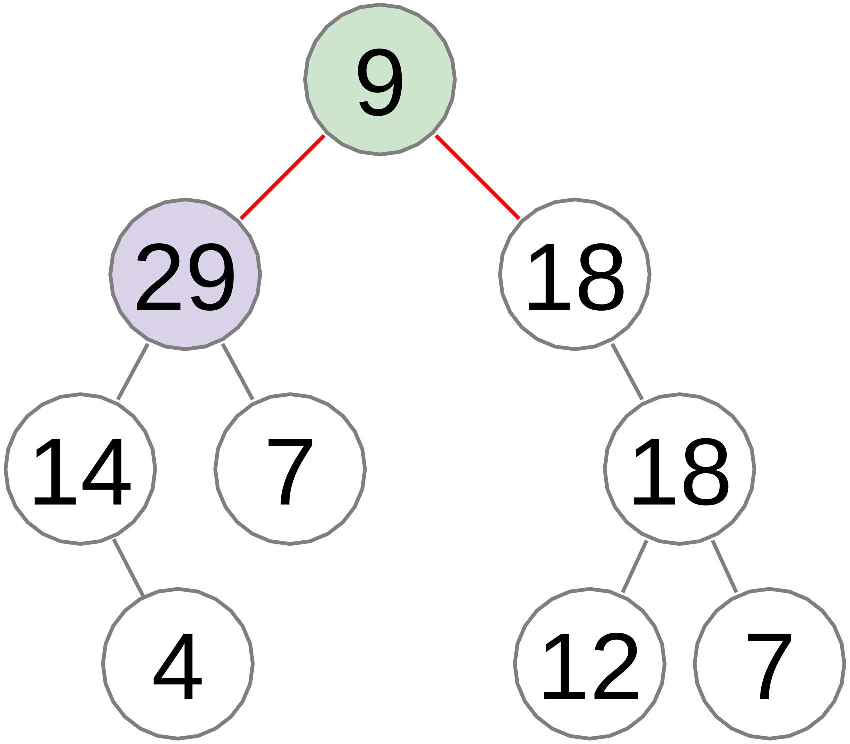
Heap operations: `sift_down(e)`

if current node is smaller than one of its children, swap it with the largest child



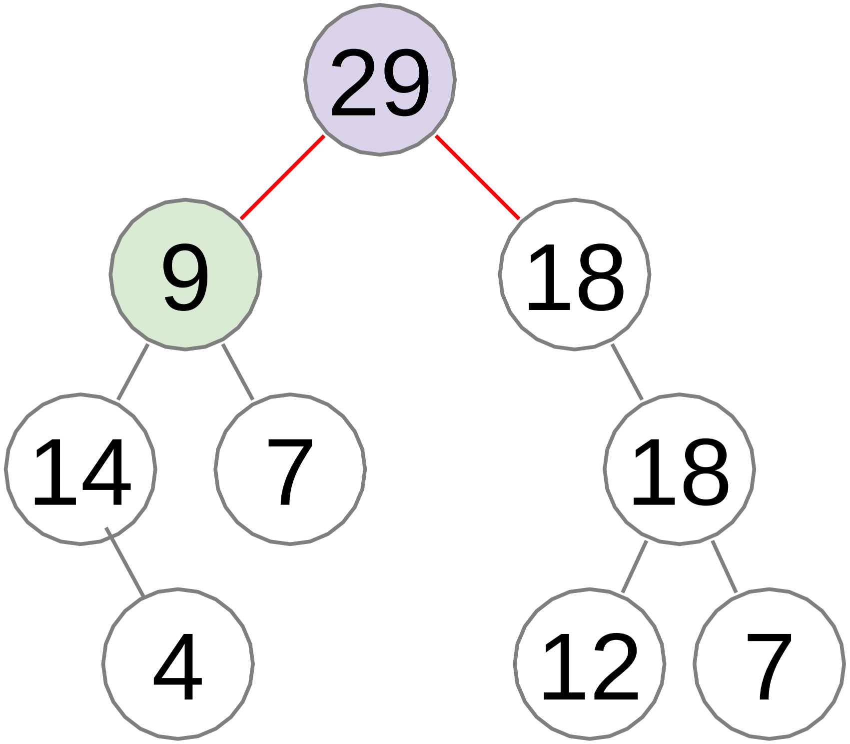
Heap operations: ***sift_down(e)***

swapping with the largest child automatically restores both broken edges



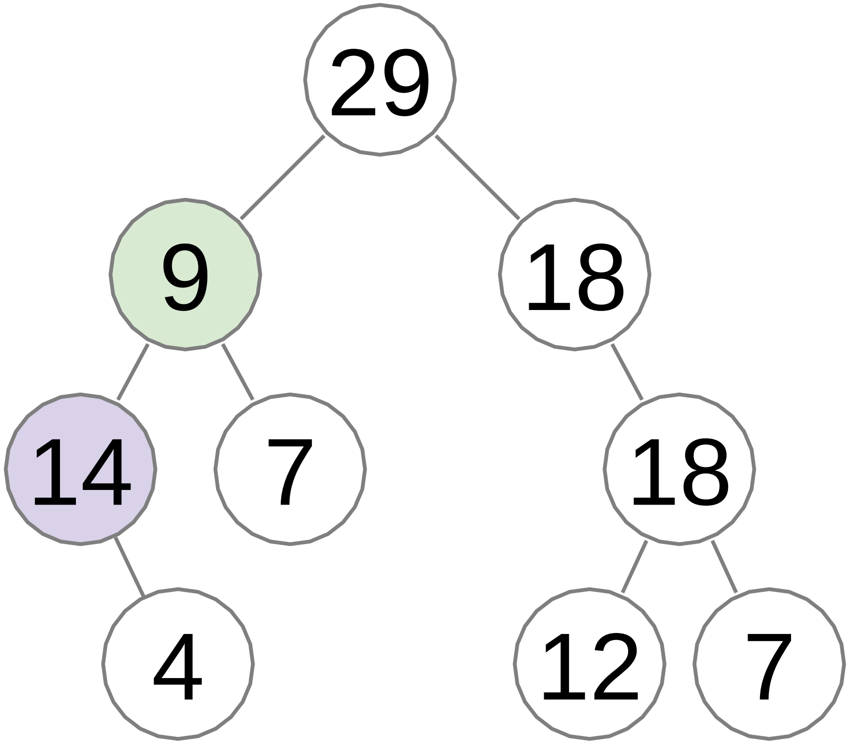
Heap operations: ***sift_down(e)***

swapping with the largest child automatically restores both broken edges



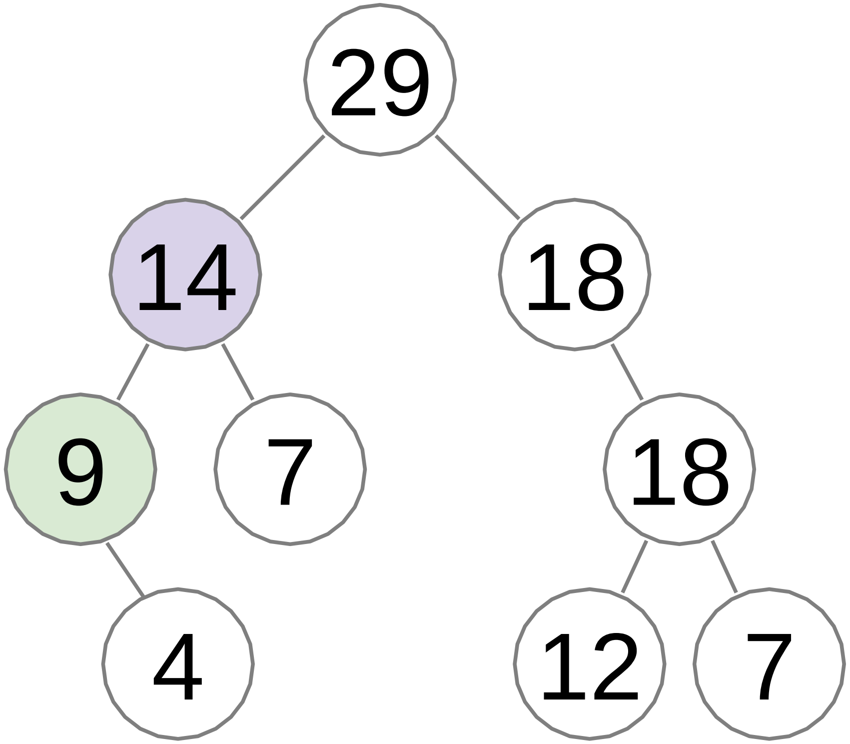
Heap operations: ***sift_down(e)***

if current node is smaller than one of its children, swap it with the largest child



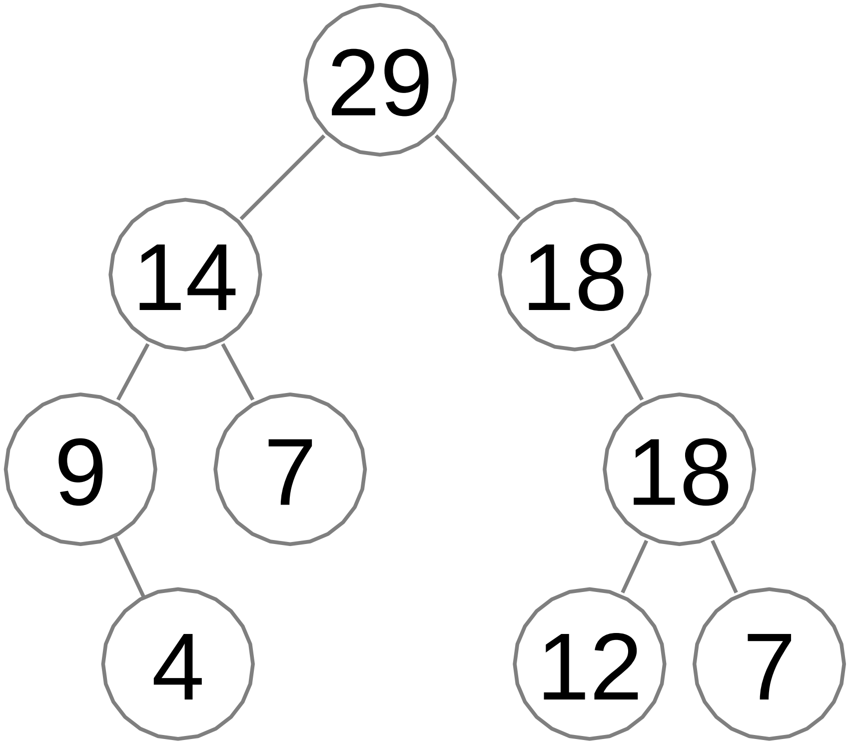
Heap operations: `sift_down(e)`

if current node is smaller than one of its children, swap it with the largest child



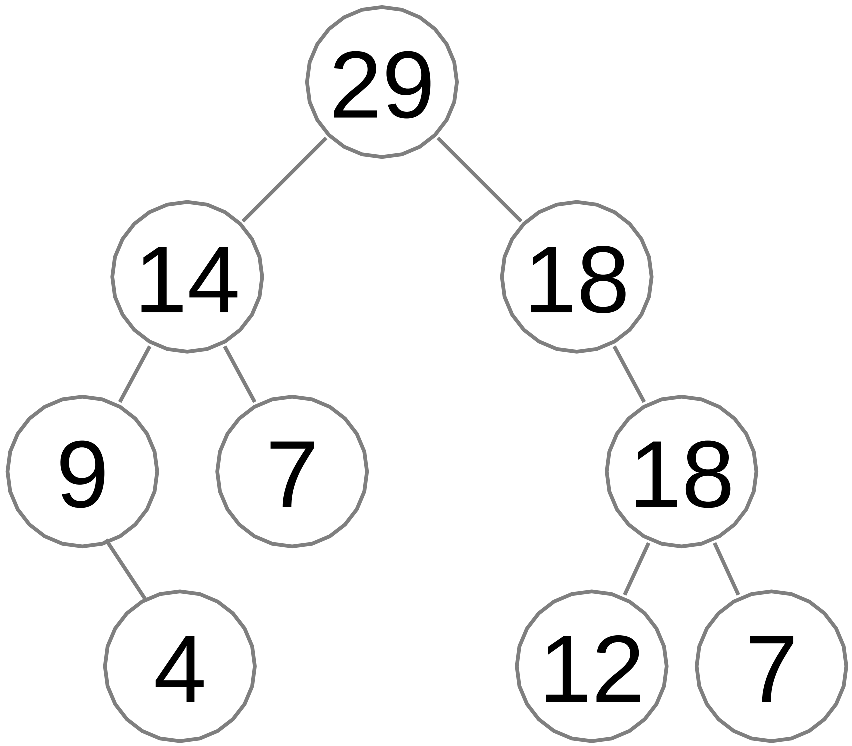
Heap operations: `sift_down(e)`

the heap property is restored



Heap operations: ***dequeue***

depends on how many times the *swap* is performed to restore the heap



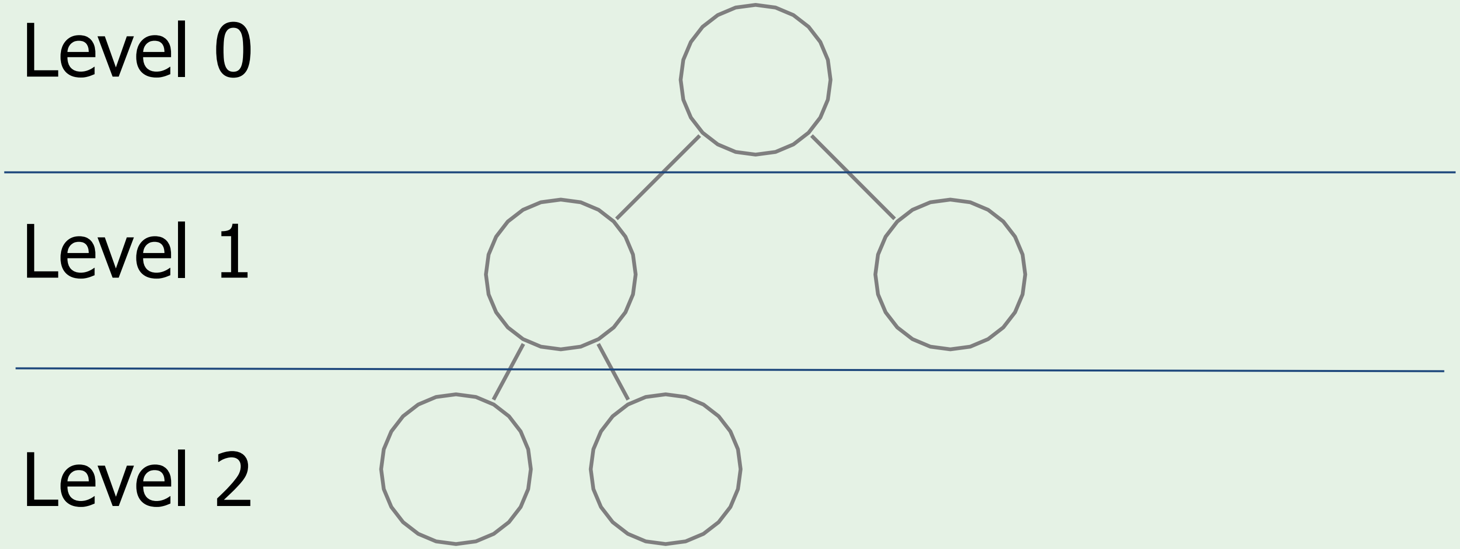
running time: *O*(tree height)

We want a tree with min height
How to Keep a Tree Shallow?

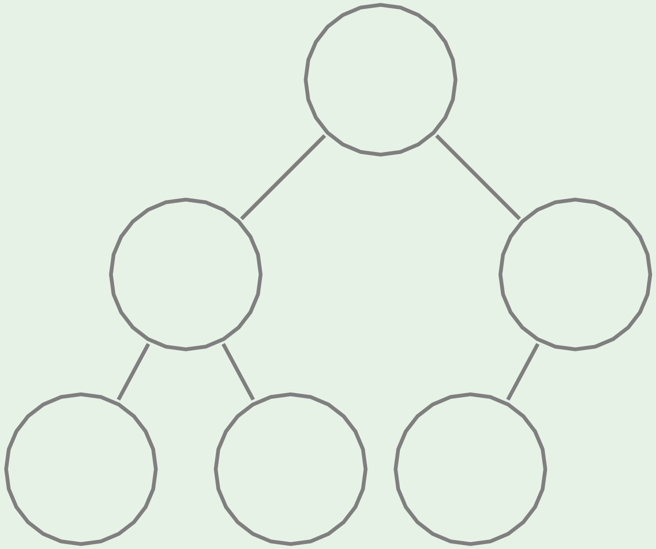
Definition

A binary tree is ***complete*** if all its levels are full except possibly the last one which is filled from left to right.

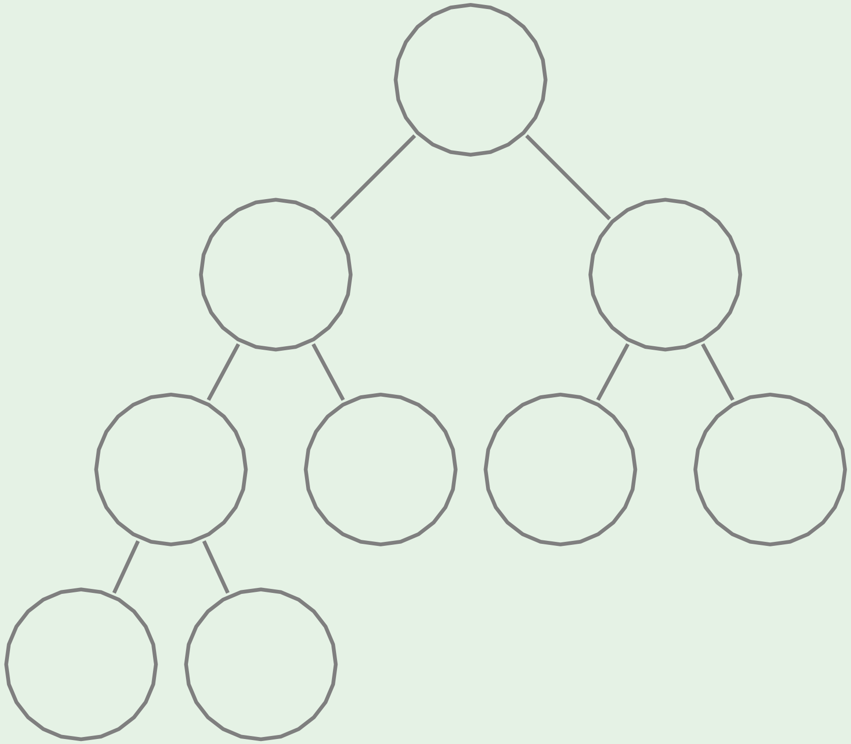
Example: complete binary tree



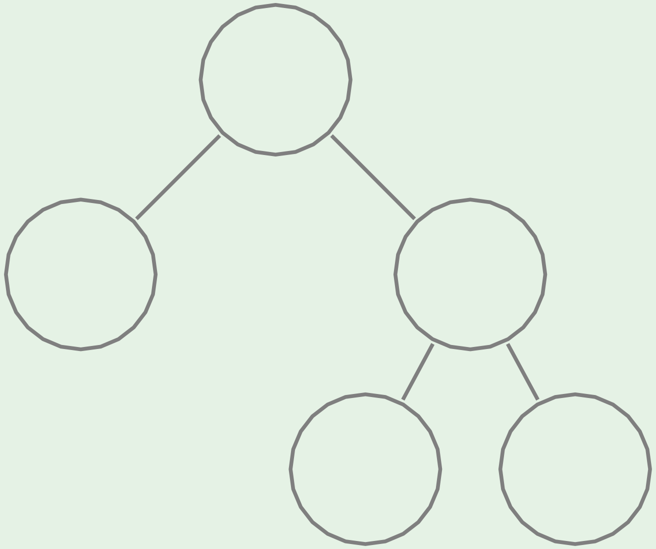
Complete binary tree ?



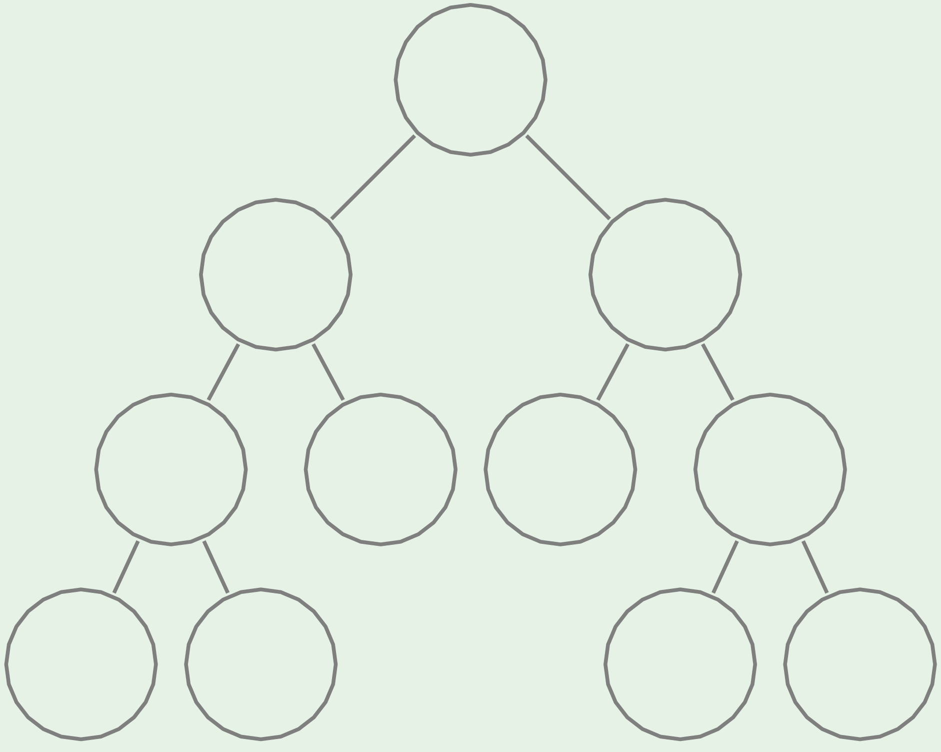
Complete binary tree ?



Complete binary tree ?



Complete binary tree ?



Advantage of Complete Binary Trees: low height

Theorem

A complete binary tree with n total nodes has height at most $O(\log n)$.

Proof

- Complete the last level of the tree if it is not full to get a **full** binary tree.

- This full tree has $n' \geq n$ nodes and the same height h.

- At level 0 we have $2^0=1$ node, at the first level: $2^1=2$ nodes, at level k: 2^k nodes, and the total number of levels is h-1. Then the total number of nodes:

$$n' = 1 + 2^1 + 2^2 + \ldots 2^{h-1} = \frac{2^{(h-1)+1} - 1}{2-1} = 2^h - 1$$

(sum of geom. series)

- Note that $n' \leq 2n$, because the actual total number of nodes n is between $2^{h-2+1} - 1 + 1 = 2^{h-1}$ and $2^h - 1$

- Then $n' = 2^h - 1$ and hence:
  $h = \log_2(n' + 1) \leq \log_2(2n + 1) = O(\log n)$. ■

If we store Heap as Complete Binary Tree:

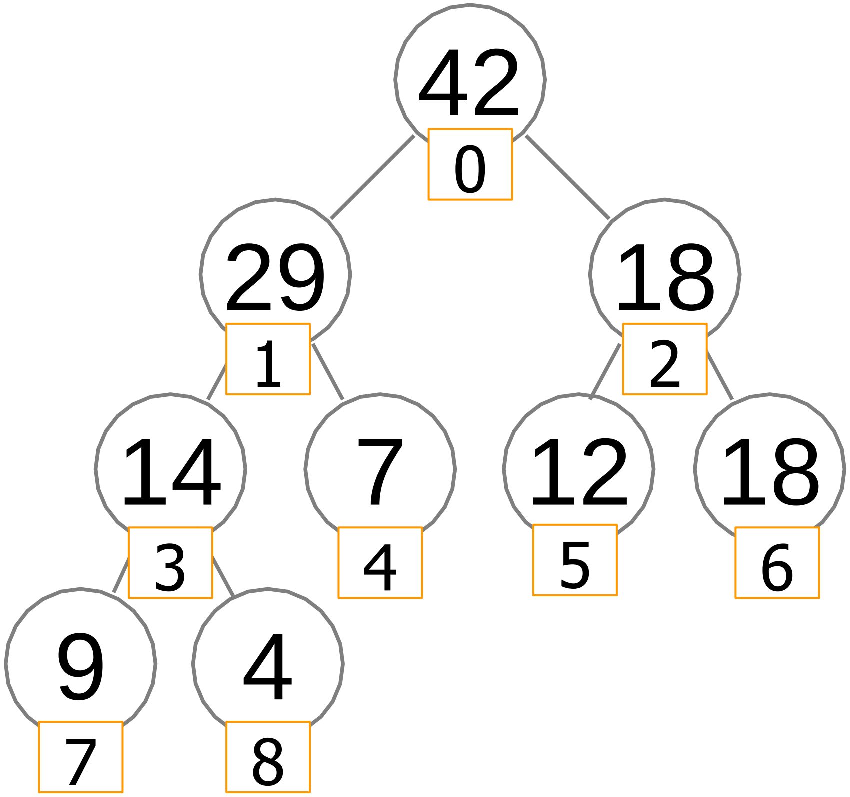
→ *Top* in time O(1)

→ *Dequeue* in time O(log *n*)

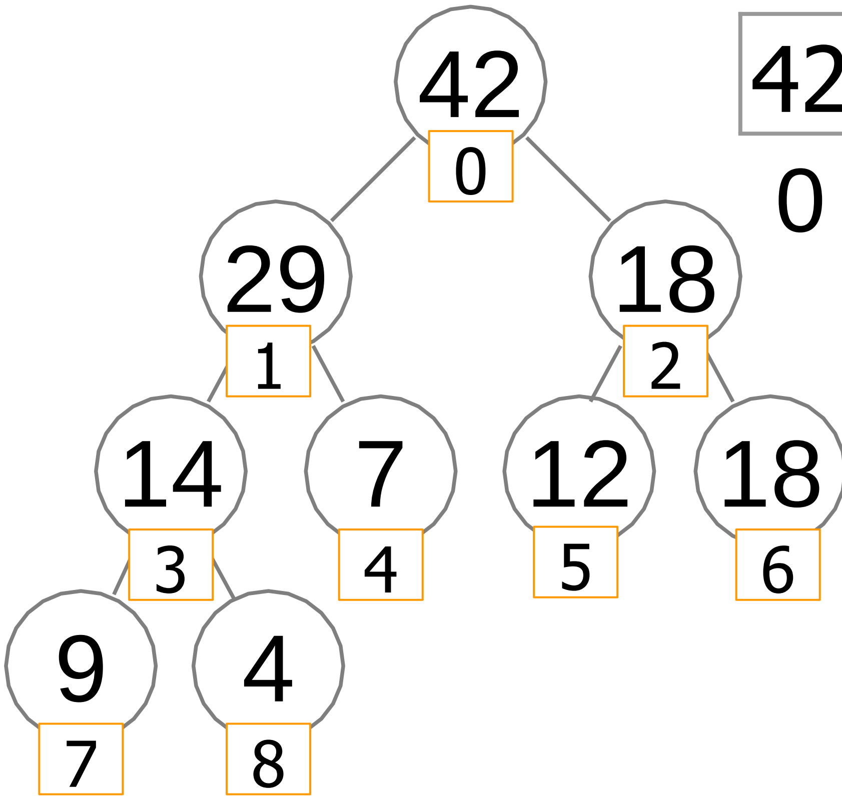
→ *Enqueue* in time O(log *n*)

As long as we keep the tree complete

The Complete Binary Tree can be stored in an Array



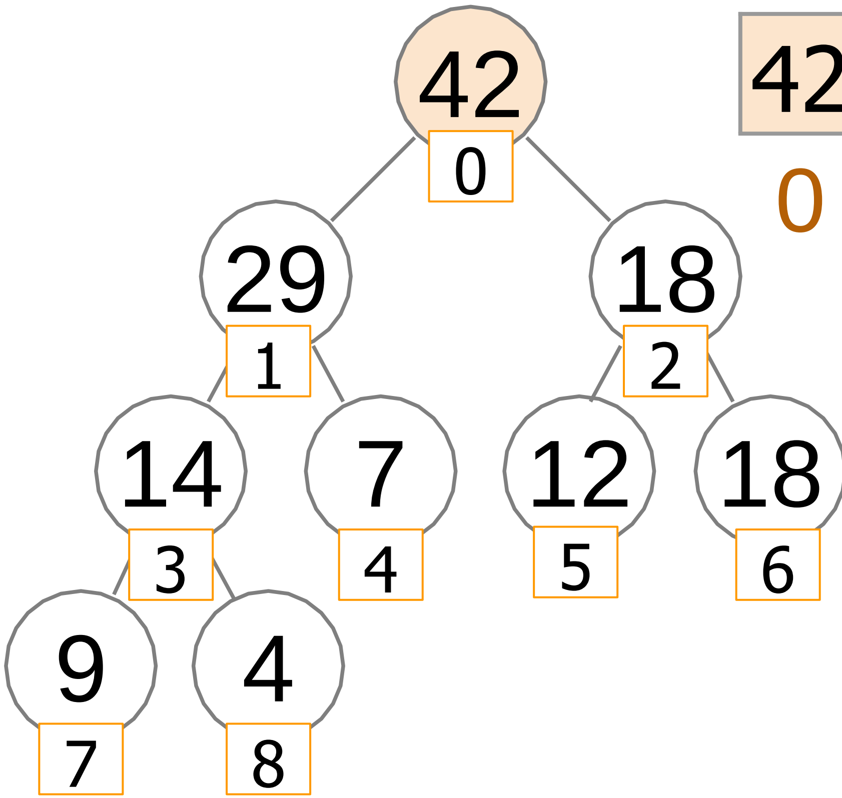
The Complete Binary Tree can be stored in an Array



42	29	18	14	7	12	18	9	4
0	1	2	3	4	5	6	7	8

Zero-based array!

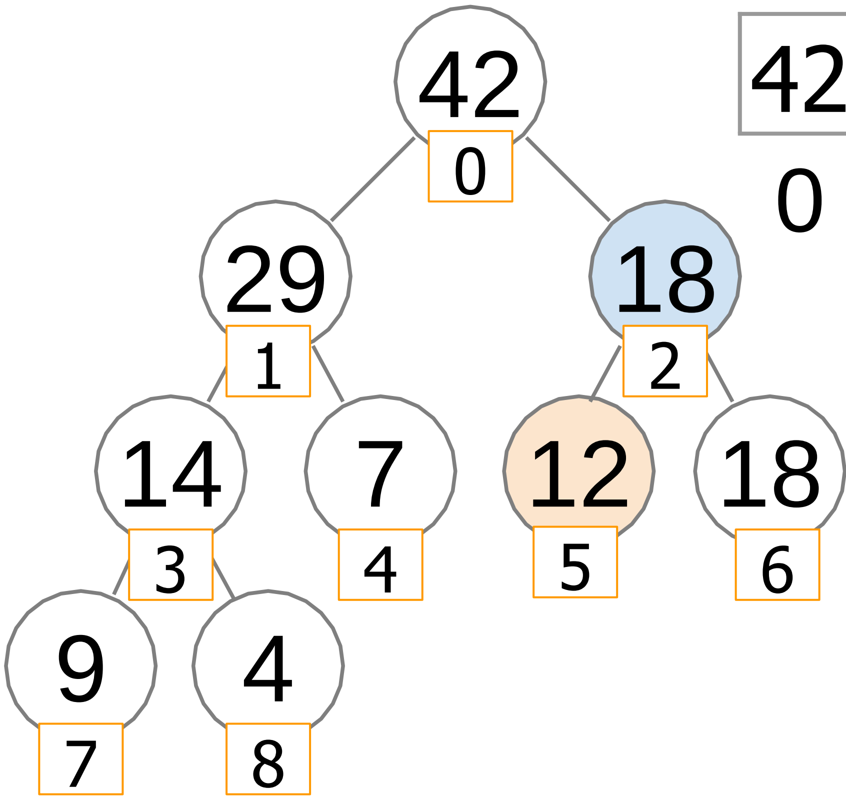
[The Complete Binary Tree can be stored in an Array]



42	29	18	14	7	12	18	9	4
0	1	2	3	4	5	6	7	8

top: A[0]

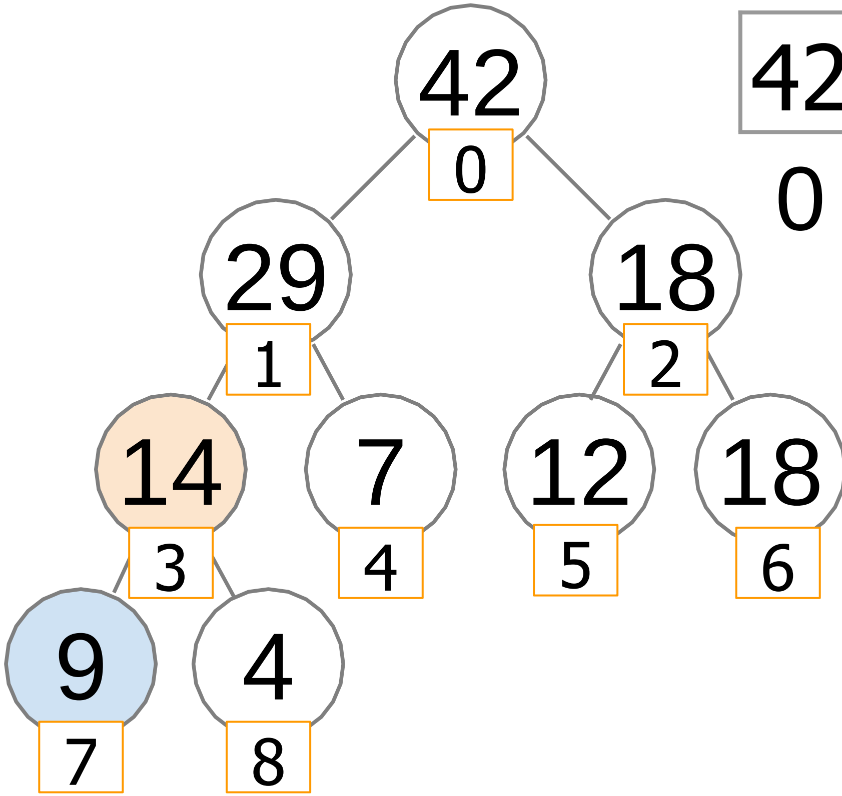
Tree operations in a heap array



42	29	18	14	7	12	18	9	4
0	1	2	3	4	5	6	7	8

$$\text{parent}(A[i]) = A[\lfloor (i-1)/2 \rfloor]$$

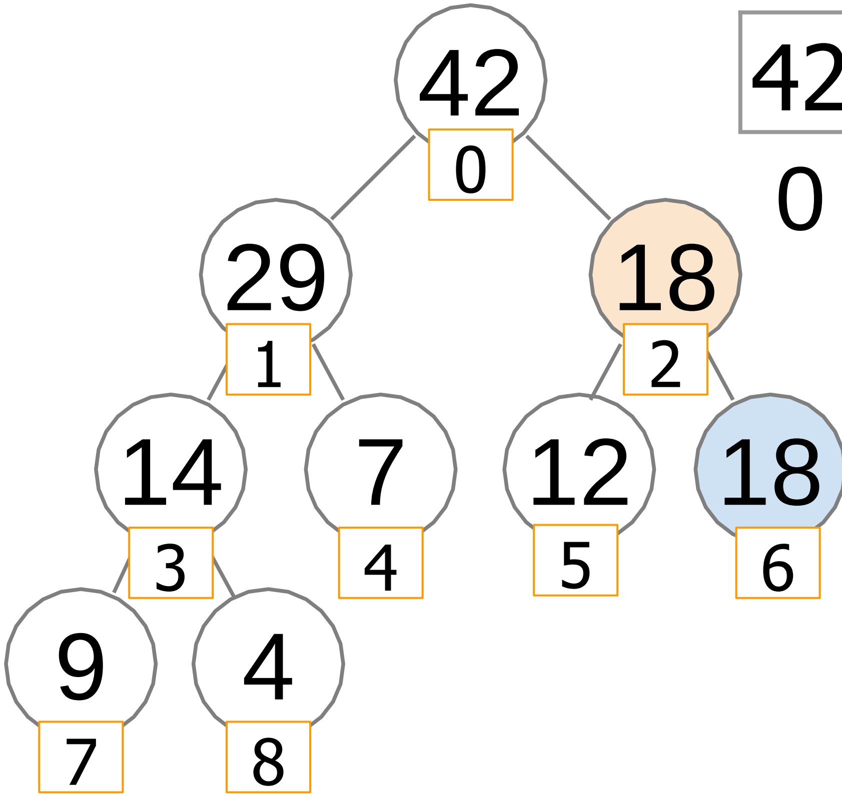
Tree operations in a heap array



42	29	18	14	7	12	18	9	4
0	1	2	3	4	5	6	7	8

left_child(A[i]) = A[$2i + 1$]

Tree operations in a heap array

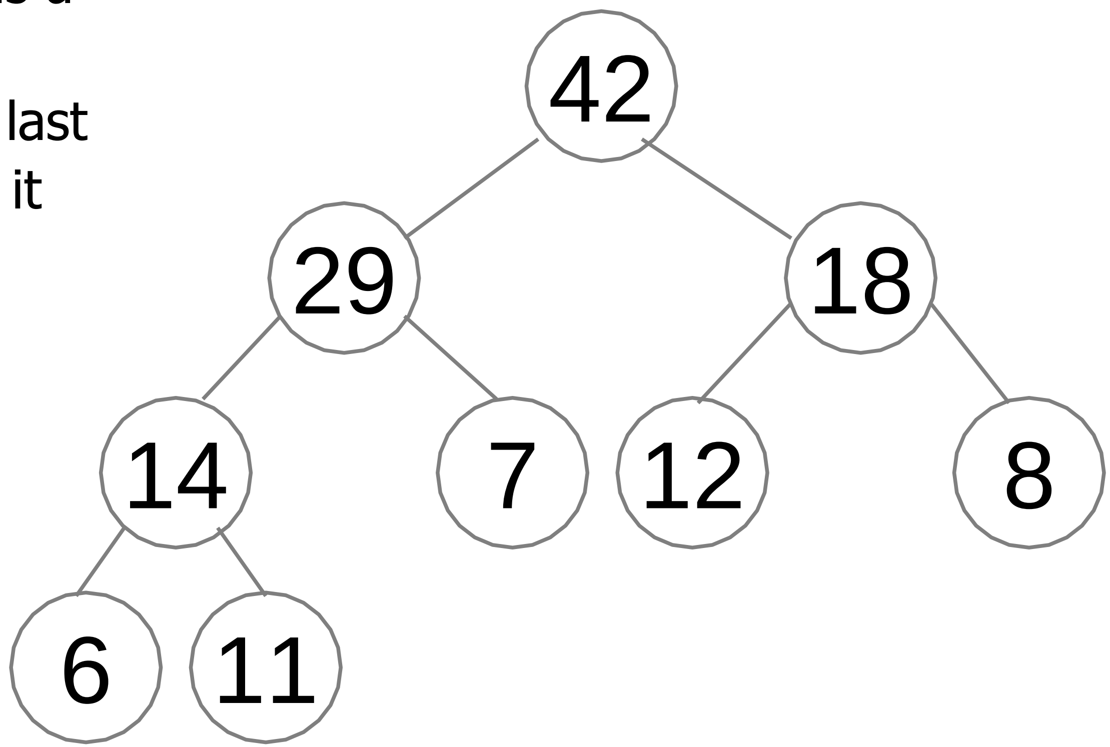


42	29	18	14	7	12	18	9	4
0	1	2	3	4	5	6	7	8

right_child(A[i]) = A[$2i + 2$]

Heap array: ***enqueue(33)***

to add an element, insert it as a leaf in the leftmost vacant position in the last level (the last position of the array) and let it *sift up*



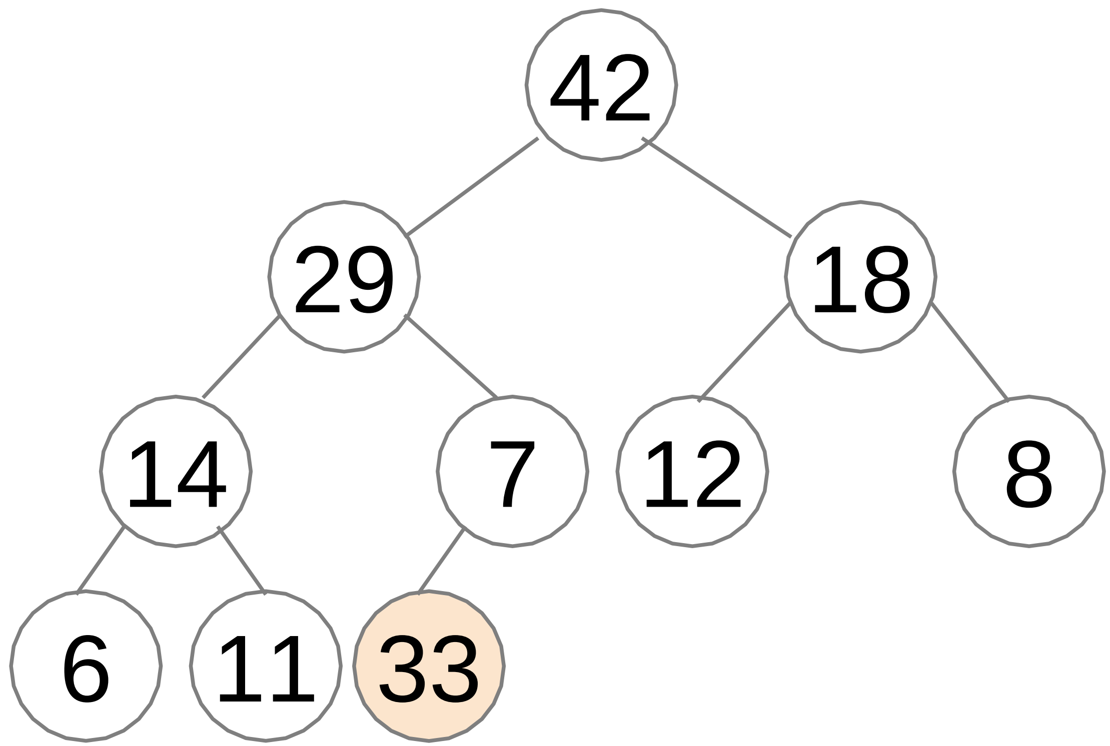
0	1	2	3	4	5	6	7	8	9
42	29	18	14	7	12	8	6	11	

Heap array: ***`enqueue(33)`***

parent(9) = 4
swap(A[9],A[4])

parent(4) = 1
swap(A[4],A[1])

parent(1) = 0 OK
stop



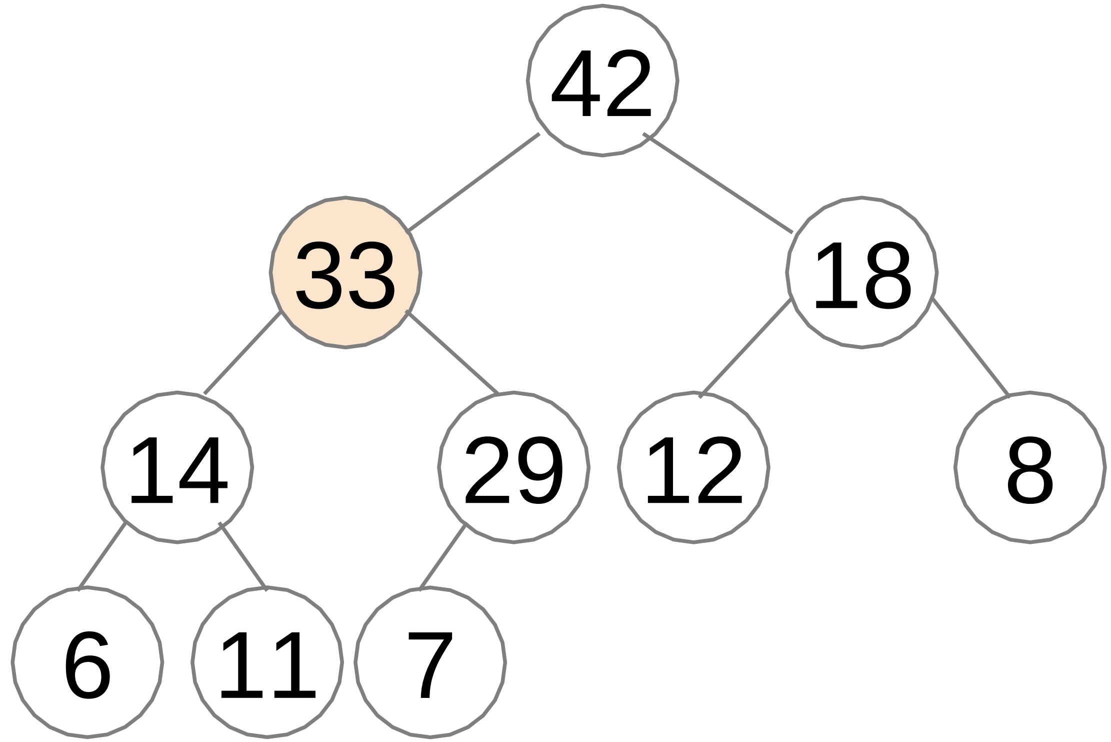
0	1	2	3	4	5	6	7	8	9
42	29	18	14	7	12	8	6	11	33

Heap array: ***enqueue(33)***

parent(9) = 4
swap(A[9],A[4])

parent(4) = 1
swap(A[4],A[1])

parent(1) = 0 OK
stop



42	33	18	14	29	12	8	6	11	7
0	1	2	3	4	5	6	7	8	9

Heap array: **`dequeue()`**

Similarly, to extract the maximum value, replace the root by the last leaf and let it *sift down*

Binary **Min**-Heap

Definition

Binary **min**-heap is a binary tree where the value of each node is **at most** the values of its children.

Can be implemented similarly to max-heap

Priority Queue: possible Data Structures

	enqueue	dequeue
Unsorted array/list	O(1)	O(n)
Sorted array/list	O(n)	O(1)
Binary heap	O(log n)	O(log n)

- Binary heap can be used to implement *Priority Queue* ***ADT***
- Heap implementation is very efficient: all required operations work in time ***O*(log *n*)**
- Heap implementation as an array is also **space efficient**: we only store an array of priorities. Parent-child relationships are not stored, but are implied by the positions in the array

Common implementations of Priority Queues using Heaps

- C++: *priority_queue* in *std* library
- Java: *PriorityQueue* in *java.util* package
- Python: *heapq* (separate module)

Underneath is a dynamic array

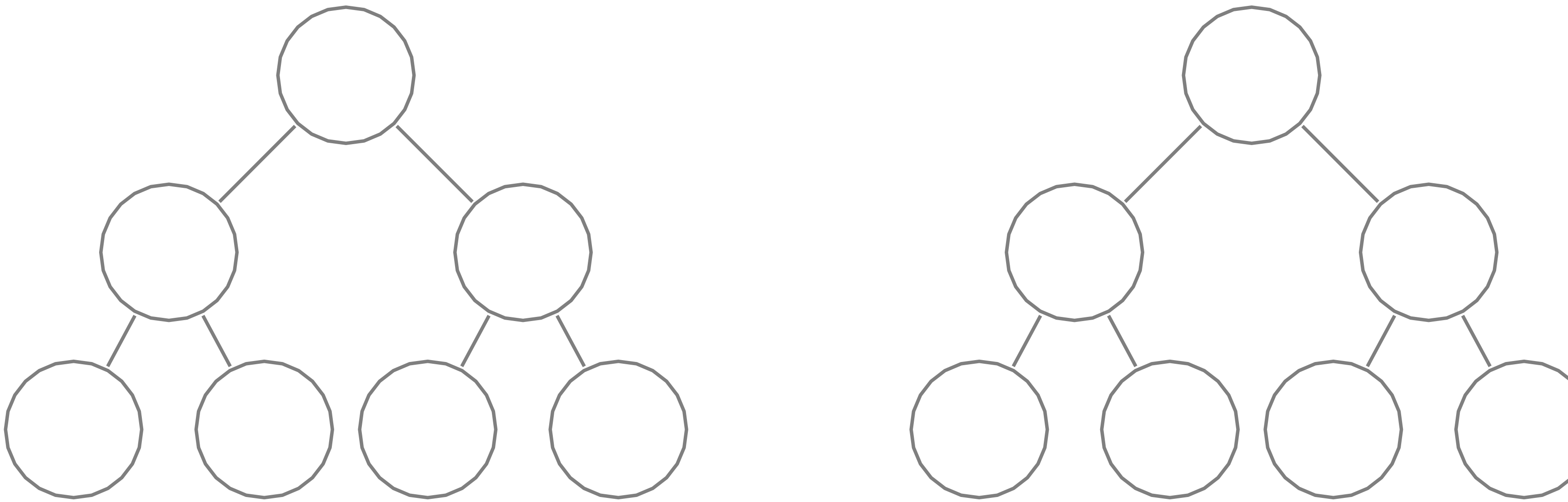
Problem 3

Maintaining median

Input: Array A of n elements with dynamic maintenance in time O(log n)

Output: Median - the middle value of elements in A in time O(1)

Median in time O(1)



0, 1, 2, 3, 5, 7, 8, 9