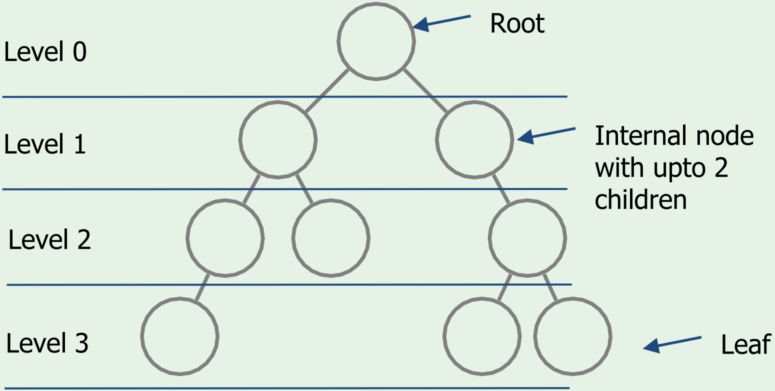
Binary Heaps

Lecture 02.06
by Marina Barsky

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

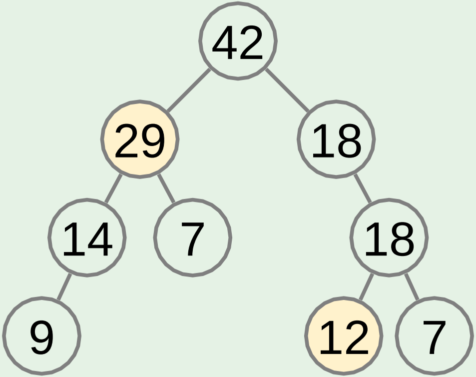
Example: Binary Tree



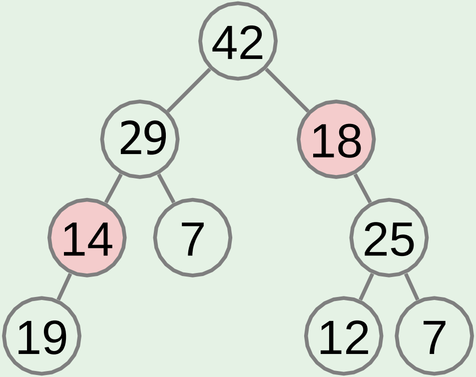
Definition

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.

Example: heap

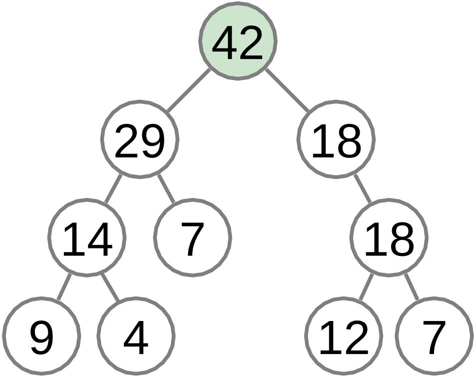


Example: not a heap



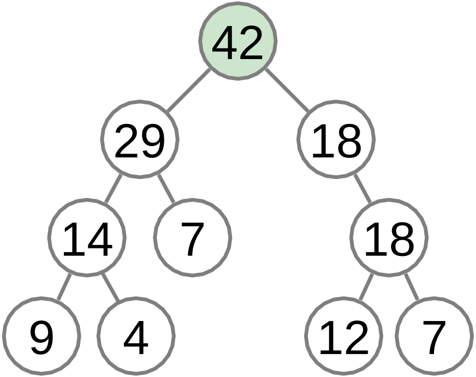
Heap operations: **`get max`**

return the
root value



Heap operations: **`get max`**

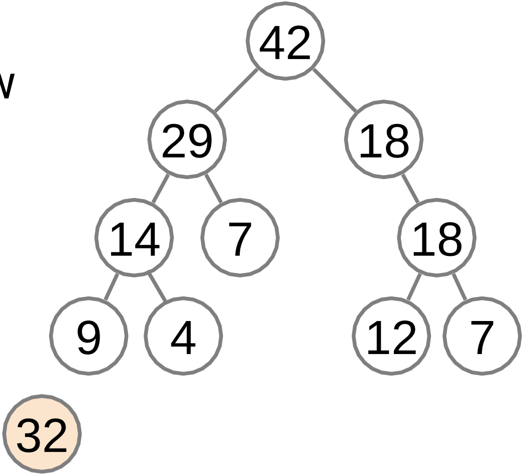
return the root value



Run-time: O(1)

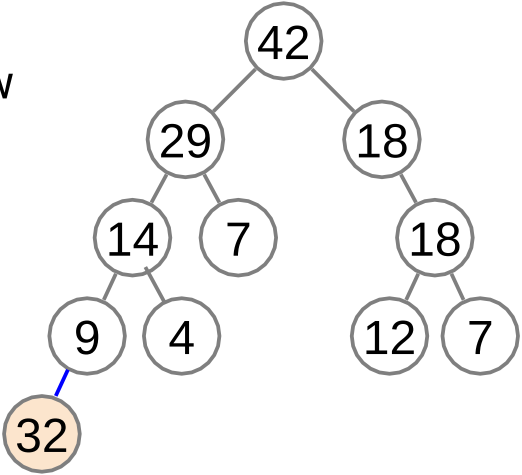
Heap operations: ***insert(e)***

create a new node



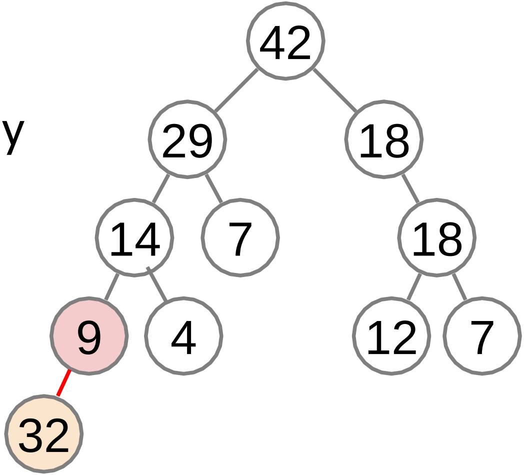
Heap operations: ***`insert(e)`***

attach a new node to any leaf



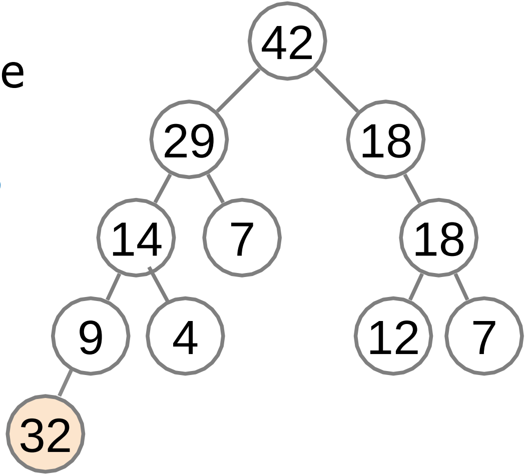
Heap operations: ***`insert(e)`***

the heap property may become violated



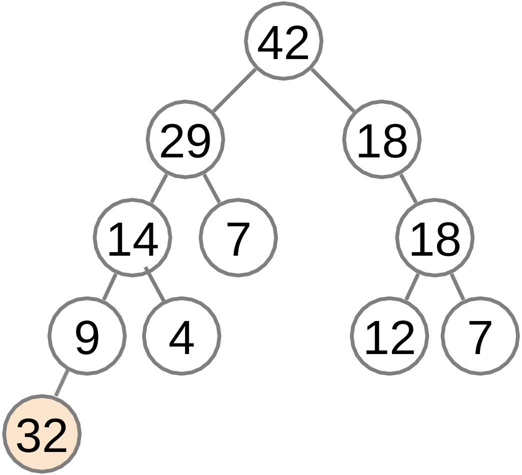
Heap operations: ***`insert(e)`***

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



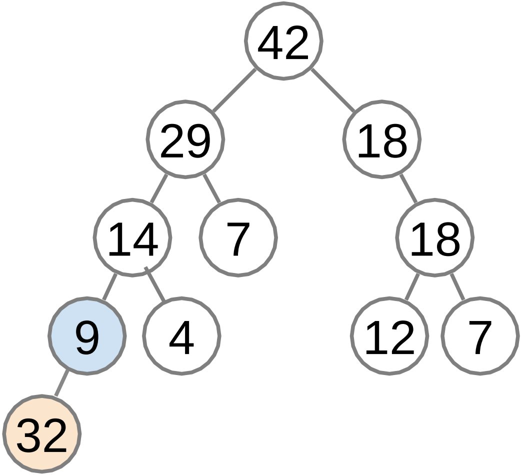
Heap operations: `sift_up(e)`

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



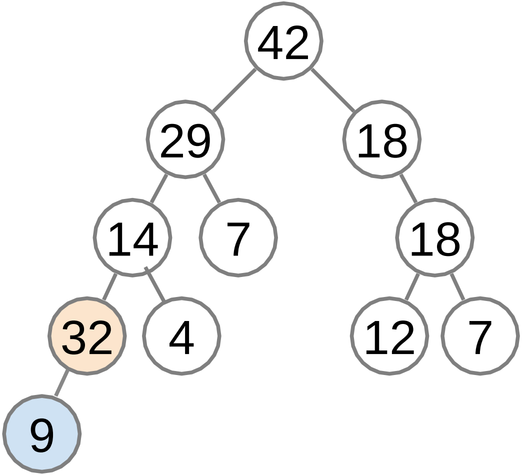
Heap operations: ***sift_up(e)***

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



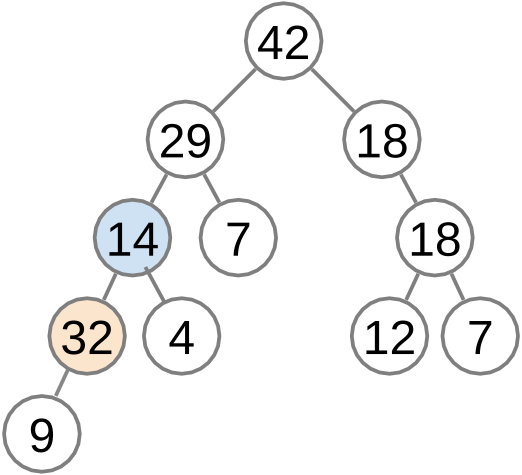
Heap operations: `sift_up(e)`

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



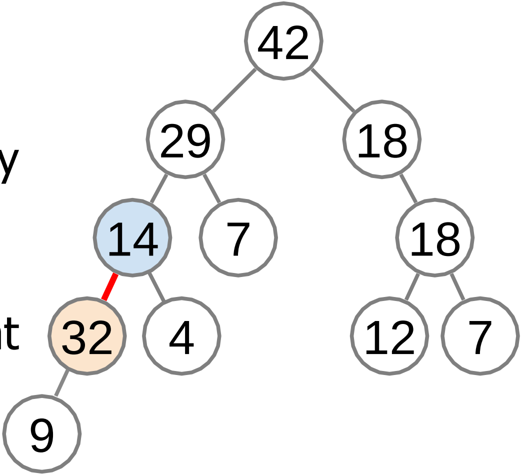
Heap operations: `sift_up(e)`

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



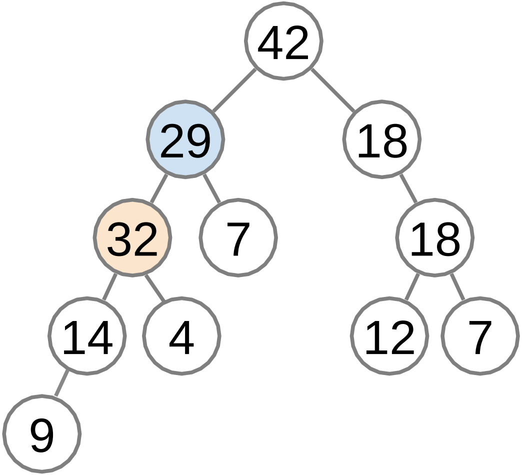
Heap operations: ***sift_up(e)***

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



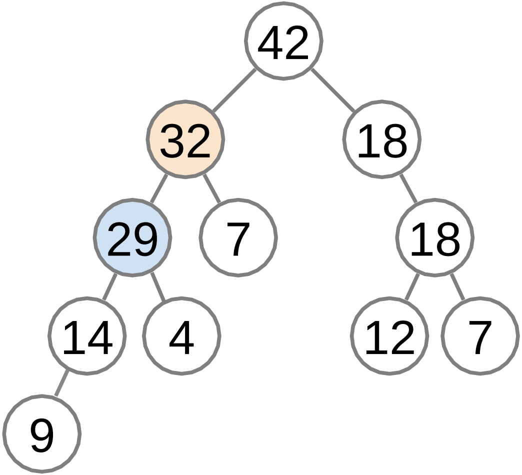
Heap operations: ***sift_up(e)***

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



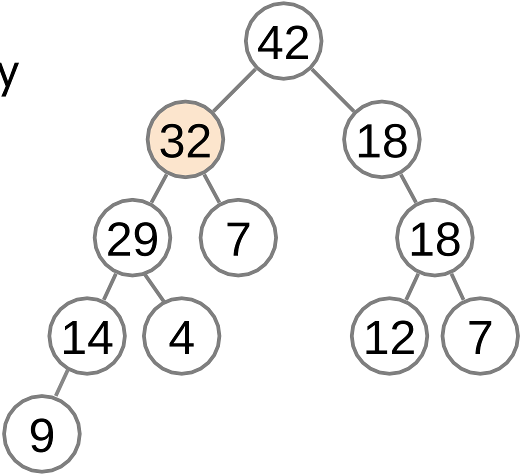
Heap operations: `sift_up(e)`

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



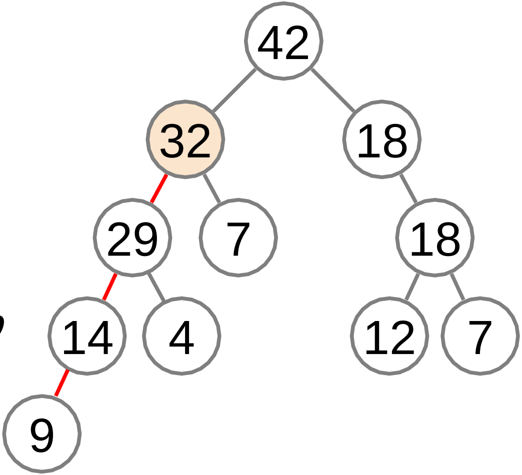
Heap operations: ***sift_up(e)***

heap property
is restored



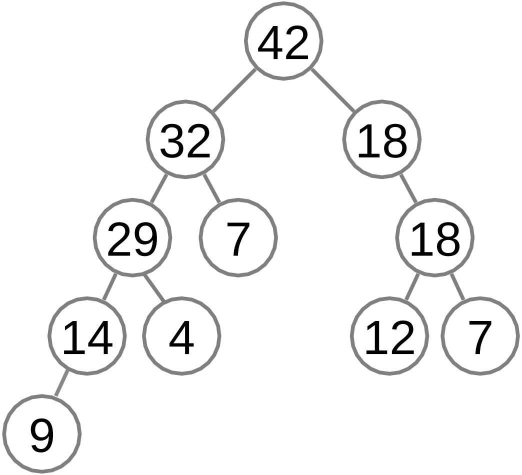
Heap operations: ***`insert(e)`***

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



Heap operations: ***insert(e)***

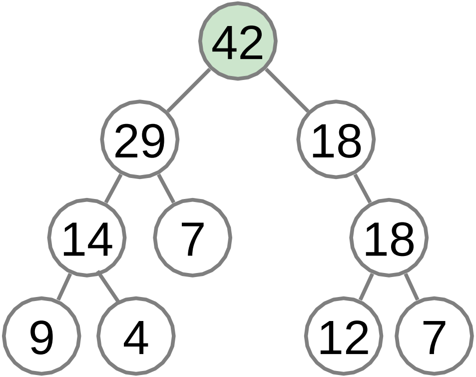
the problematic node gets closer to the root with each swap



running time: *O*(tree height)

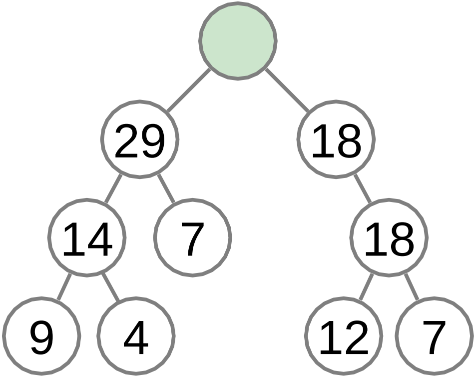
Heap operations: ***extract max***

remove and
return the
root value



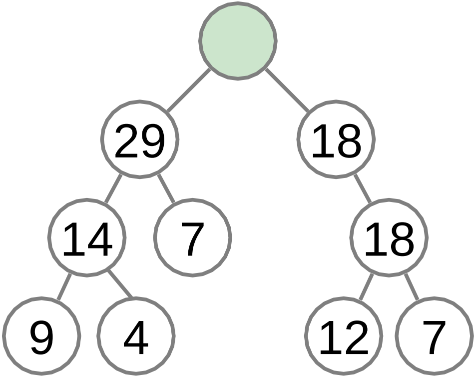
Heap operations: **`extract max`**

remove the root value



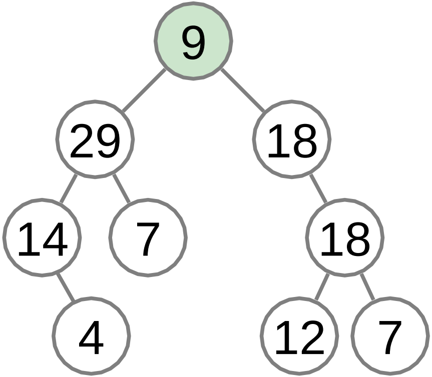
Heap operations: **`extract max`**

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



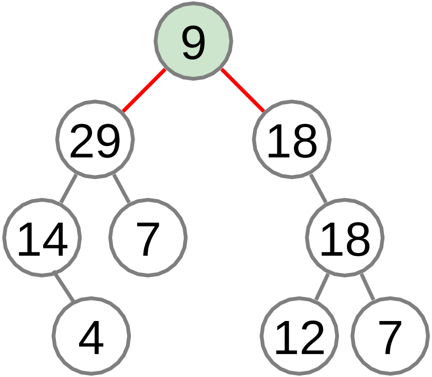
Heap operations: ***extract max***

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



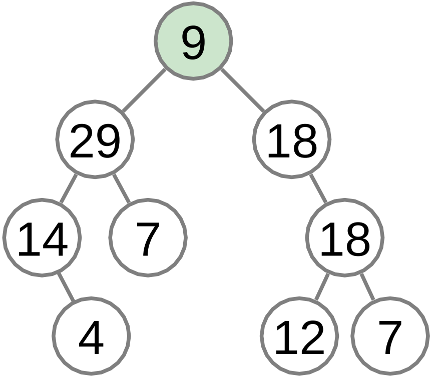
Heap operations: ***extract max***

again, this may violate the heap property



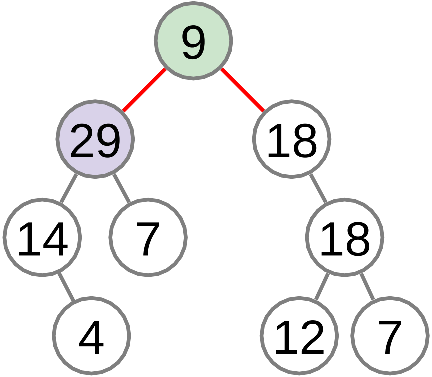
Heap operations: **`extract max`**

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



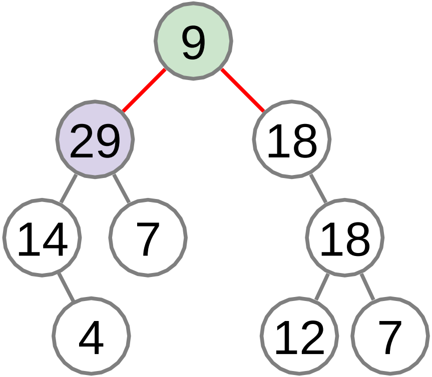
Heap operations: *sift_down(e)*

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



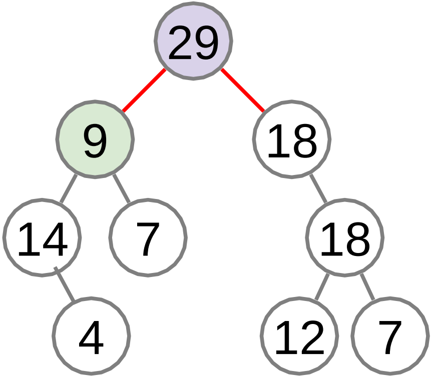
Heap operations: ***sift_down(e)***

swapping with the largest child automatically restores both broken edges



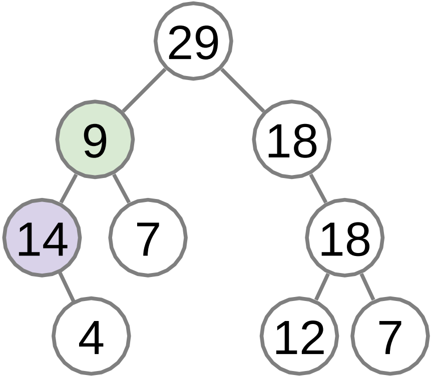
Heap operations: `sift_down(e)`

swapping with the largest child automatically restores both broken edges



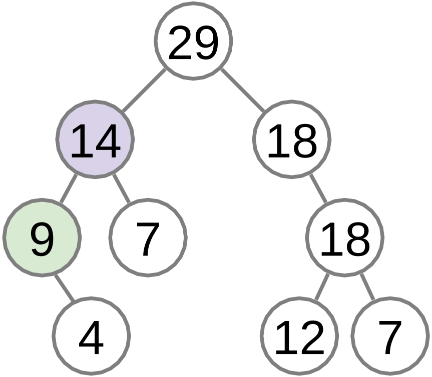
Heap operations: ***sift_down(e)***

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



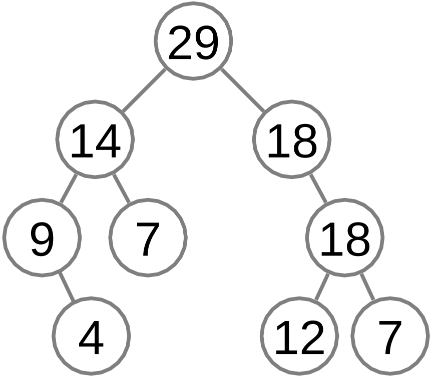
Heap operations: *sift_down(e)*

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



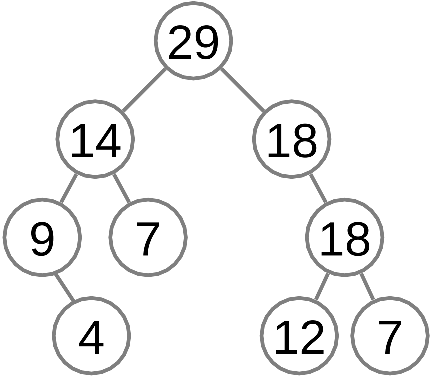
Heap operations: ***sift_down(e)***

the heap property is restored



Heap operations: ***extract max***

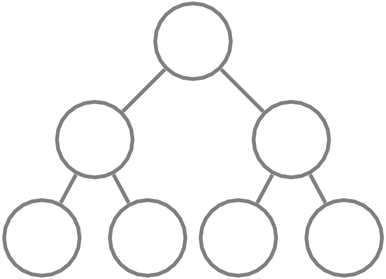
depends on how many times the *swap* is performed to restore the heap

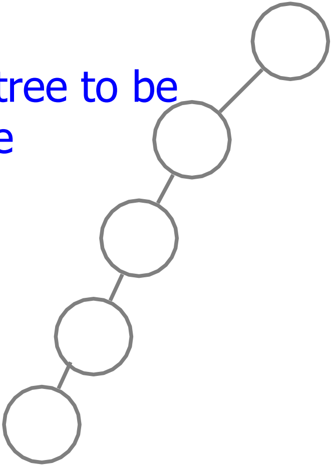


running time: O(tree height)

Summary so far

- **`get_max`** works in time $O(1)$
- all other operations work in time $O(\text{tree height})$
- we definitely want a tree to be as shallow as possible



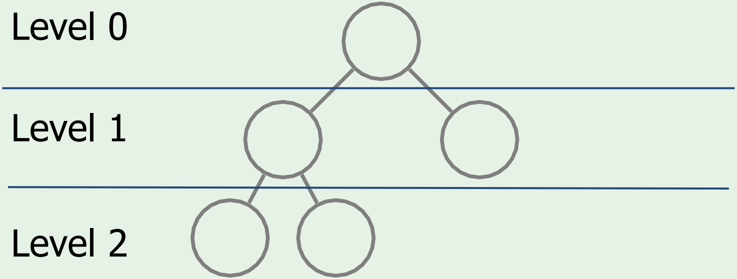


How to Keep a Tree Shallow?

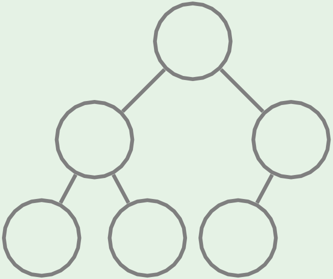
Definition

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

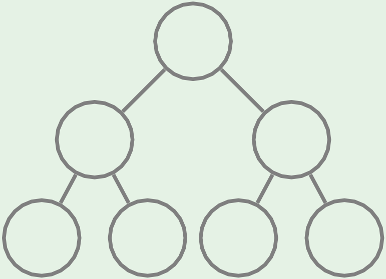
Example: complete binary tree



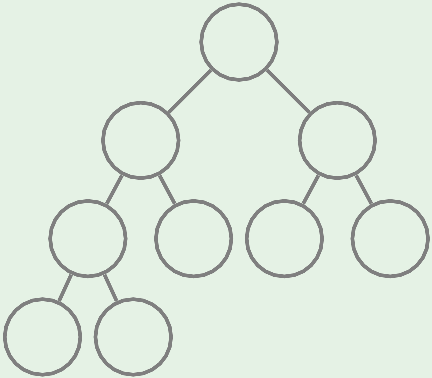
Example: complete binary tree



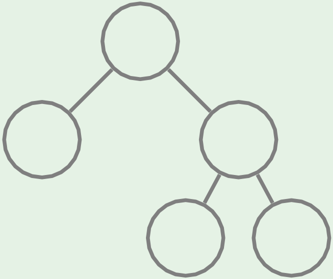
Example: complete binary tree



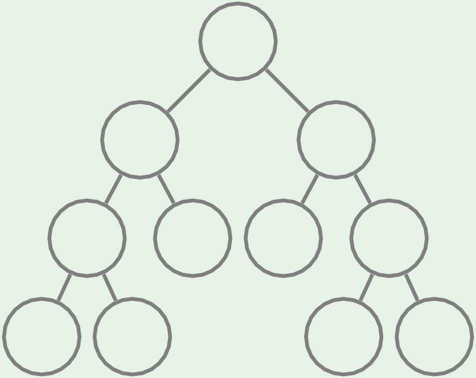
Example: complete binary tree



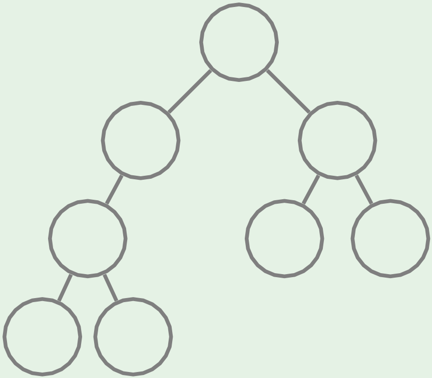
Example: not complete binary tree



Example: not complete binary tree



Example: not complete binary tree



Advantage of Complete Binary Trees: Low Height

Lemma

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 number of levels with the last level marked as ℓ.
- Note that $n' \leq 2n$, because the total number of nodes n is between $2^{\ell-1} - 1$ and $2^{\ell} - 1$
- Then $n' = 2^{\ell} - 1$ (sum of geometric series) and hence:

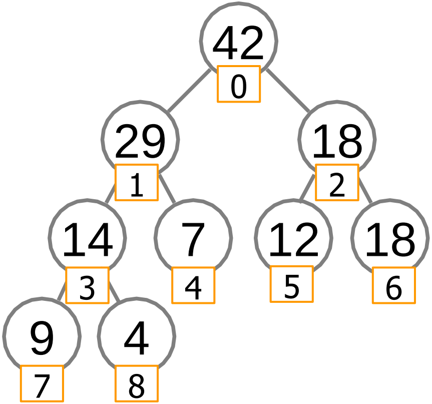
  $\ell = \log_2(n' + 1) \leq \log_2(2n + 1) = O(\log n)$. ∎

If we store Heap as a Complete Binary Tree we can:

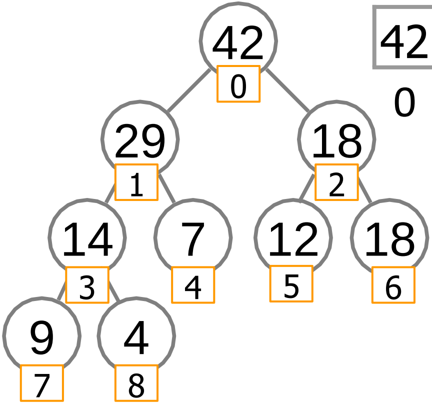
- ➔ *Get max* in time O(1)
- ➔ *Extract max* in time O(log n)
- ➔ *Insert(e)* in time O(log n)

As long as we keep the tree complete

More advantages:
The Complete Binary Tree can be stored in an Array!

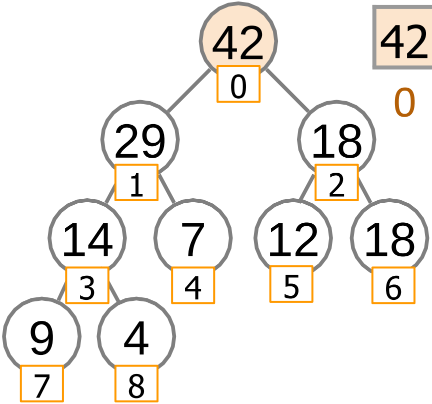


The Complete Binary Tree can be stored in an Array



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

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

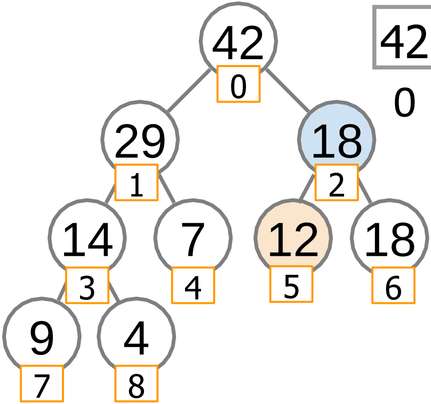
max value: A[0]

Tree operations in a heap array

But how do we perform heap operations that require traversing the tree?

- → *Insert(e)*
- → *Extract max*

Tree operations in a heap array

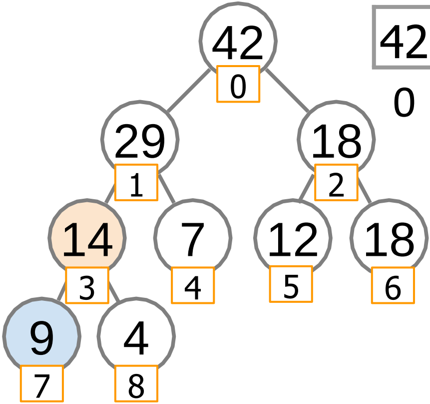


42	29	18	14	7	12	18	9	4
0	1	2	3	4	5	6	7	8

max value: $A[0]$

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

Tree operations in a heap array



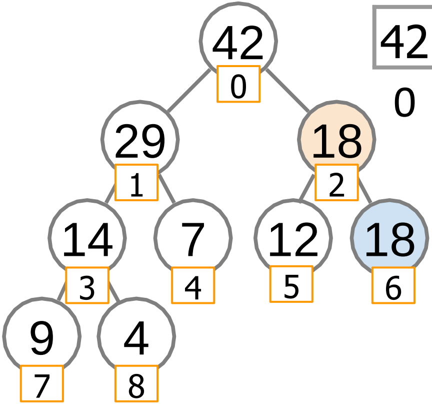
42	29	18	14	7	12	18	9	4
0	1	2	3	4	5	6	7	8

max value: A[0]

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

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

max value: A[0]

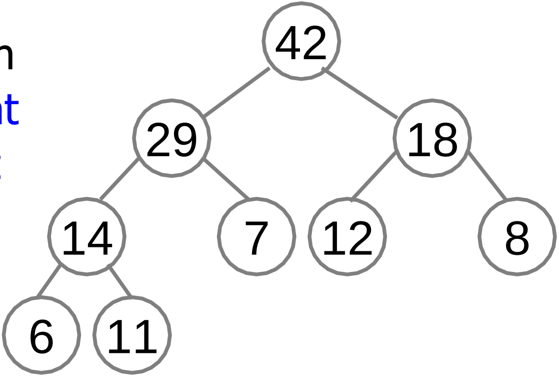
parent(A[i]) = A[$\lfloor(i-1)/2\rfloor$]

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

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

Heap array: ***insert(e)***

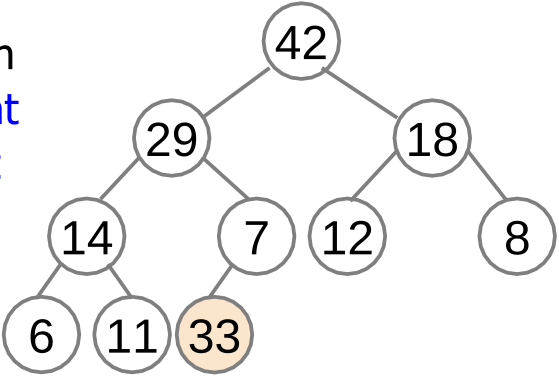
to insert element, insert it as a leaf in the leftmost vacant position in the last level (the last position of the array) and let it *sift up*



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

Heap array: ***insert(e)***

to insert 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*



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

Heap array: ***sift_up(e)***

parent(A[*i*]) = A[⌊(*i* -1)/2⌋]

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

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

parent(9) = 4

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

swap(9,4)

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

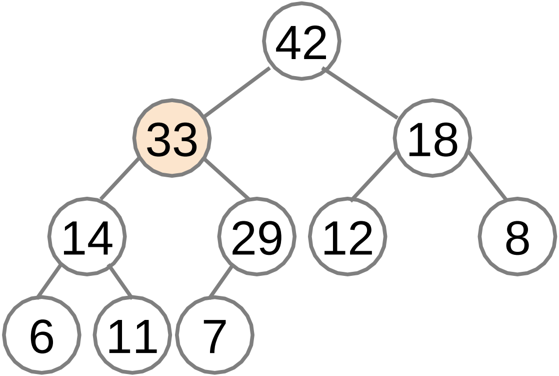
parent(4) = 1

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

swap(4,1)

parent(1) = 0 OK

Heap array: `insert(e)`

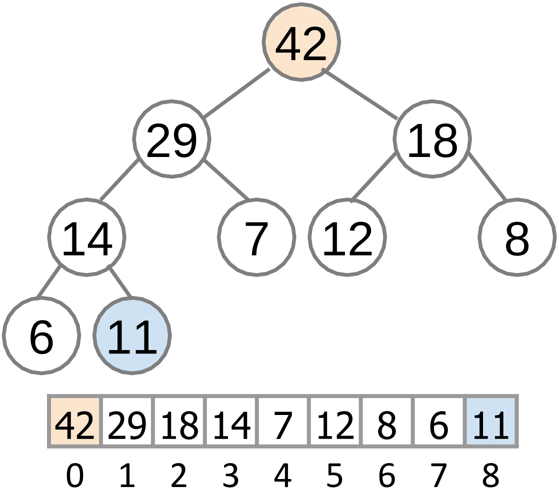


42	33	18	14	29	12	8	6	11	7
0	1	2	3	4	5	6	7	8	9

running time: $O(\log n)$

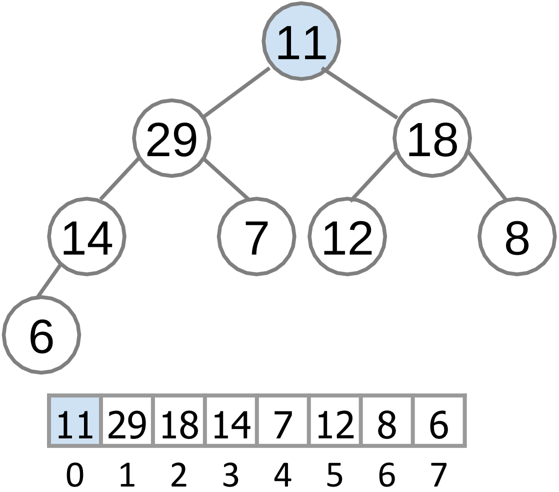
Heap array: ***`extract max()`***

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



Heap array: ***extract max()***

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



Heap array: `sift_down()`

left_child(A[*i*]) = A[2*i* + 1]
right_child(A[*i*]) = A[2*i* + 2]

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

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

left_child(0) = 1
right_child(0) = 2

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

swap with max
swap(0,1)

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

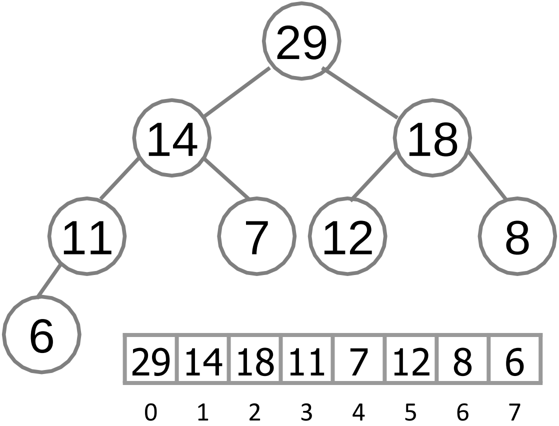
left_child(1) = 3
right_child(1) = 4

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

swap with max
swap(1,3)

heap restored

Heap array: `extract max()`

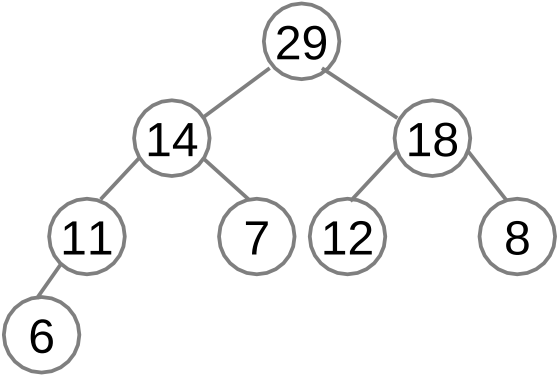


running time: $O(\log n)$

Implementing Heap with Array

- ➔ Maintain *capacity* - the maximum number of elements in the heap
- ➔ Maintain *size* - the (current) number of heap elements
- ➔ *H* [1 . . . *capacity*] is an array which occupies space *capacity* where the heap elements occupy the first *size* positions of this array

Example



capacity: 14
size: 8

29	14	18	11	7	12	8	6						
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Summary

- We learned a new **data structure**: ***binary heap***
- 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
- It is also **easy to implement**

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