September 30, 2020

Sorting algorithms

- bubbles sort
- insertion sort
- selection sort

$\left.\begin{array}{l} \\ \\ \\ \end{array}\right\}$ quadratic time algorithms

- merge sort $\longrightarrow$ the order of magnitude of the cost is $n \log n$

Asymptotic notation

O, Ω, Θ

We say that $f(n) = O(g(n))$ for some functions $f(n)$ and $g(n)$ defined on the set of natural numbers if there is a universal constant $c > 0$ and a positive integer n_0 such that

$$f(n) \leq c\, g(n)$$

for each $n \geq n_0$

Example

Bubble sort

number of comparisions for $A[1:n]$ is

$$\frac{n(n-1)}{2}$$

With this O notation we can write the cost as $O(n^2)$

It means that $\frac{n(n-1)}{2} \leq c n^2$ with a suitable $c>0$ if n in big enough

$$\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$$ for each natural number n

$$\underset{c = \frac{1}{2}}{\uparrow}$$

$$\frac{n(n+1)}{2} = O(n^2) \quad \text{also}$$

$$\frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n \leq \frac{1}{2}n^2 + \frac{1}{2}n^2 = 1 \cdot n^2$$

$$\underset{c = 1}{\uparrow}$$

Some important order of magnitude

$O(1)$	constant time	the best one
$O(\log n)$	logarithmic time	very good
$O(n)$	linear time	
$O(n \log n)$		
$O(n^2)$	quadratic time	
$\vdots$		
$O(n^k)$	with a fixed k $\rightarrow$ polynomial time	

$O(2^n)$

$O(n!)$

$O(n^n)$

algorithms with such complexities are inefficient algorithms

Ω, Θ

We say that $f(n) = \Omega(g(n))$ if there is a universal constant $c > 0$ and a natural number n_0 such that

$f(n) \geq c\, g(n)$ for each $n \geq n_0$

We say that $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ both

---

Sortings

A disadvantage of merge sort is the using of a huge amount of additional space (in memory)

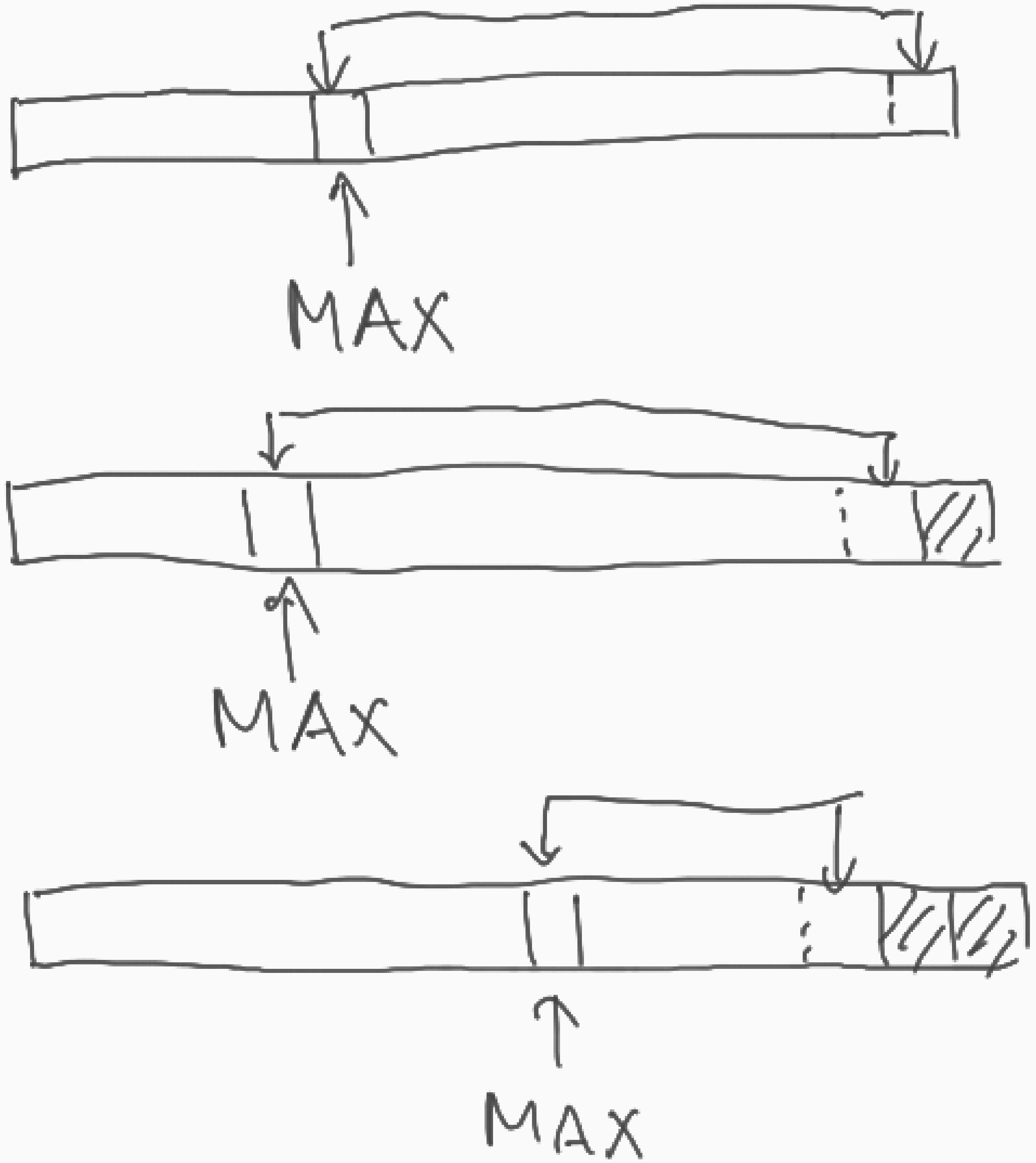
The next algorithm will be an in-place sorting algorithm (we don't need a huge amount of additional memory, only a few auxiliary variables)

The time complexity is also $O(n \log n)$

HEAP SORT

It is a special max selection sort

Max selection Sort $\rightarrow O(n^2)$



$A[1:n] \rightarrow n-1$ comp's

$A[1:n-1] \rightarrow n-2$ comp's

$A[1:n-2] \rightarrow n-3$ comp's

$\vdots$

$\frac{n(n-1)}{2}$ comp's

We select the max in each iteration among almost the same set of numbers

Can we do this faster? YES

We can do this much faster!

A quite usual technique to do first some "preprocessing"

Here preprocessing is some initial rearrangement

After this preprocessing the max selections will be much faster!

We introduce the concept of heap property (maxheap property) for arrays (of numbers)

$A[1:n]$ satisfies the maxheap property if

$$A[1] \geq A[2], A[3]$$

$$A[2] \geq A[4], A[5]$$

$$A[3] \geq A[6], A[7]$$

$$\vdots$$

$$A[i] \geq A[2i], A[2i+1]$$

Note that a sorted array satisfies the heap property but not only the sorted arrays do this

Example

9	3	6	2	1	5	4

maxheap

What we can do with a maxheap?

Note that the first element is the biggest one

9	3	6	2	1	5	4

$A[1:n]$ SWAP$(A[1], A[n])$

4	3	6	2	1	5	9

$A[1:n-1]$

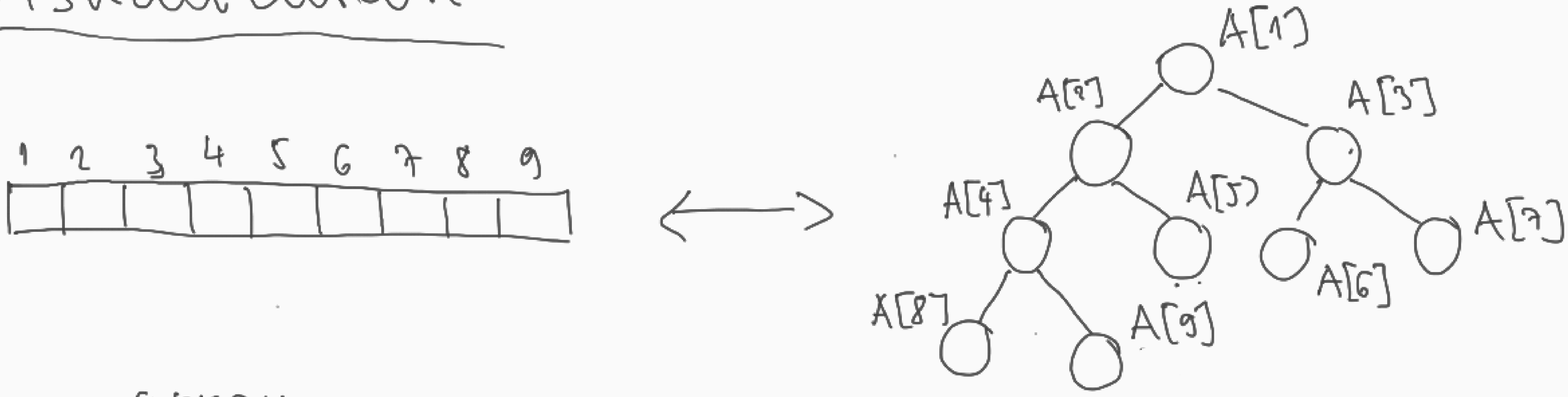
Is it heap?

No, but almost

The only problem here is that the heap property is violated at $A[1]$

We can restore the heap property quite easily here

Visualization



array

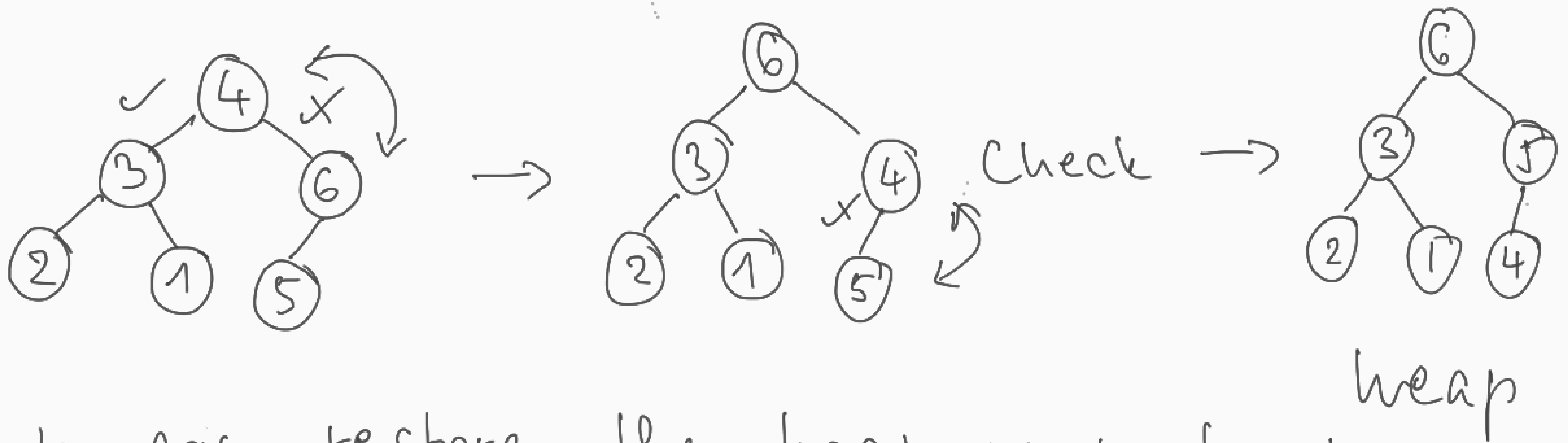
"pseudo" complete binary tree

In the tree representati the max heap property can be read as follows:

the number in a vertex in bigger than or equal to the numbers in the children of this vertex

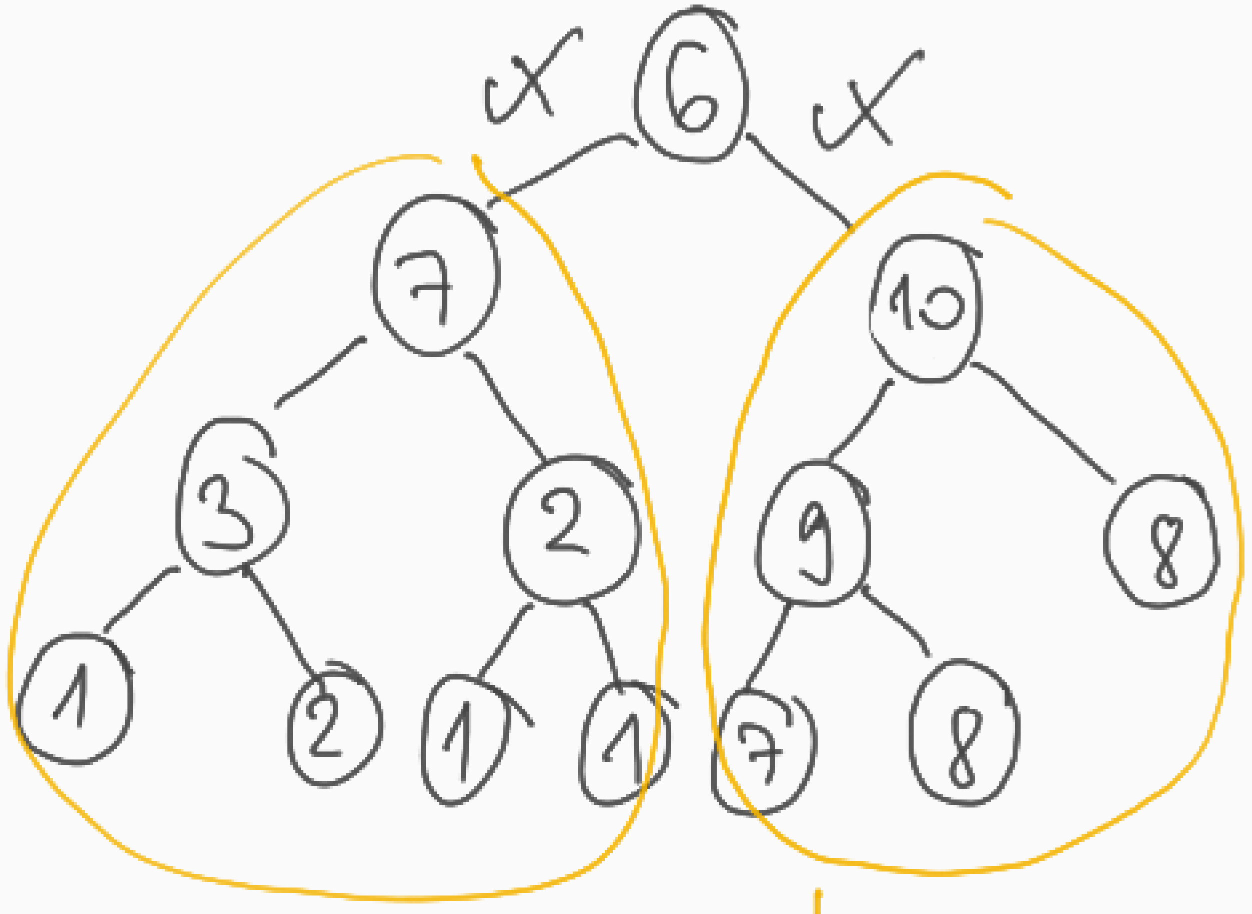
How can we restore the heap property in the tree representation

4	3	6	2	1	5

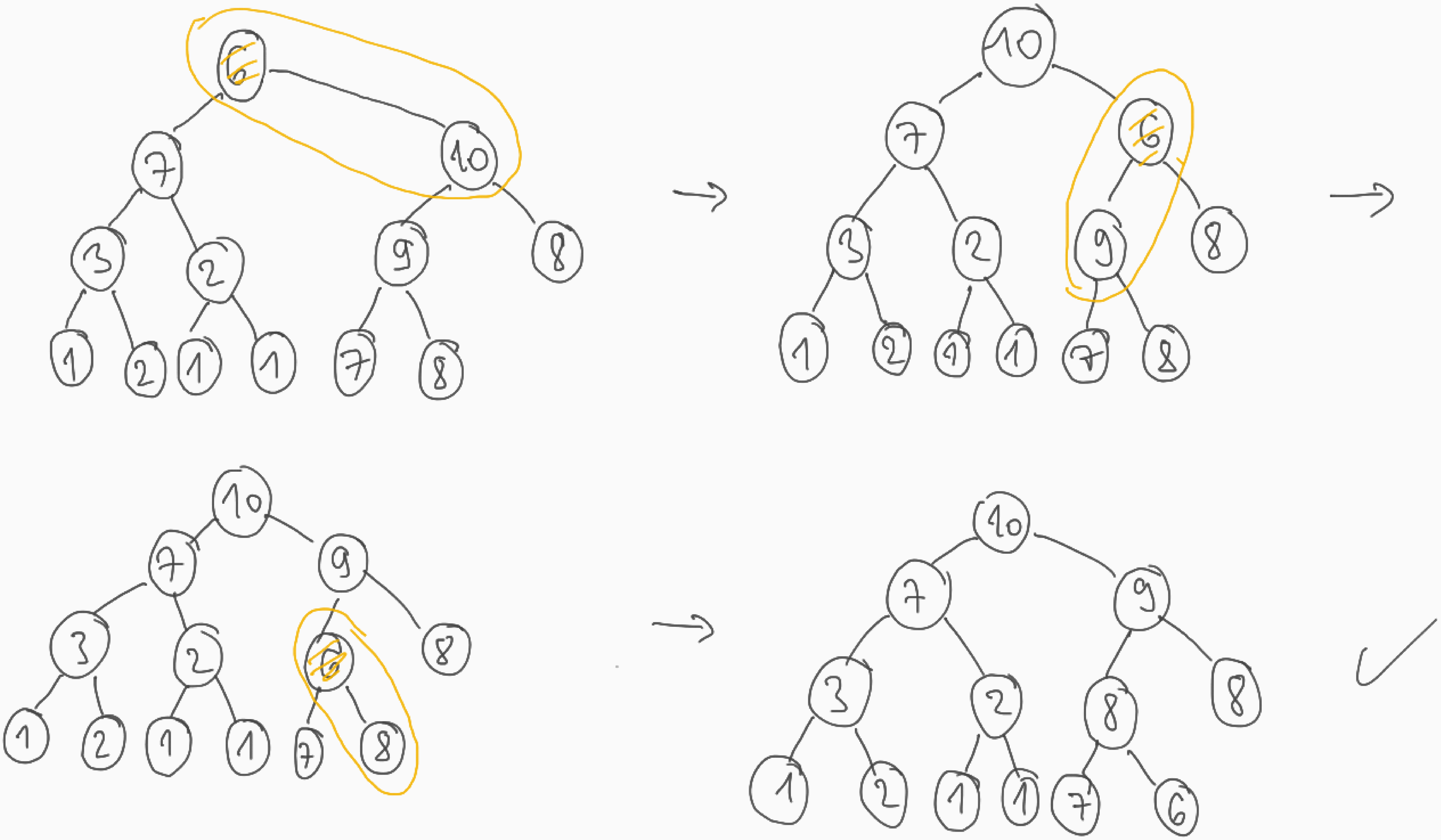


We can restore the heap property by swapping parent-child pairs along ONE root - leaf path

One more example



if the number (6) in a vertex is smaller than both numbers (7) (10) in its children then we swap it with the bigger child



In the array

6	7	10	3	2	9	8	1	2	11	7	8

How many comparisons are needed to restore the heap property and to identify the max $\longrightarrow$ $O(\log n)$

[instead of $O(n)$ as in the simple max selection]

After preprocessing (which takes a linear time as we will see in the next lecture) we can find the next maximum in $O(\log n)$ time after the swap of the previous max with the last element of the array [instead of $O(n)$ time]