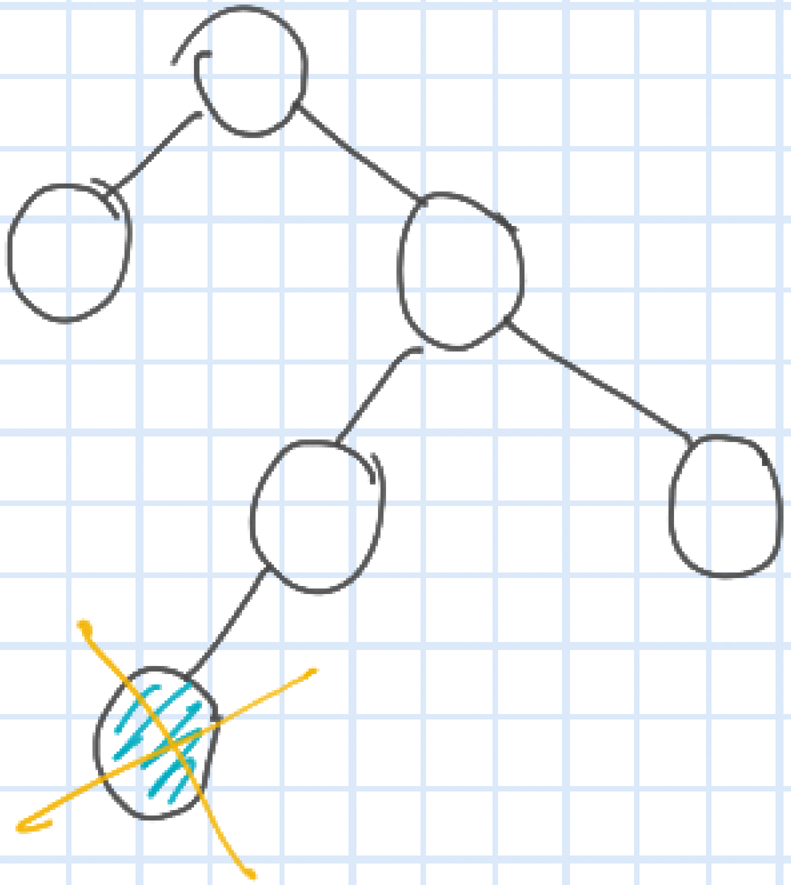
November 26, 2020

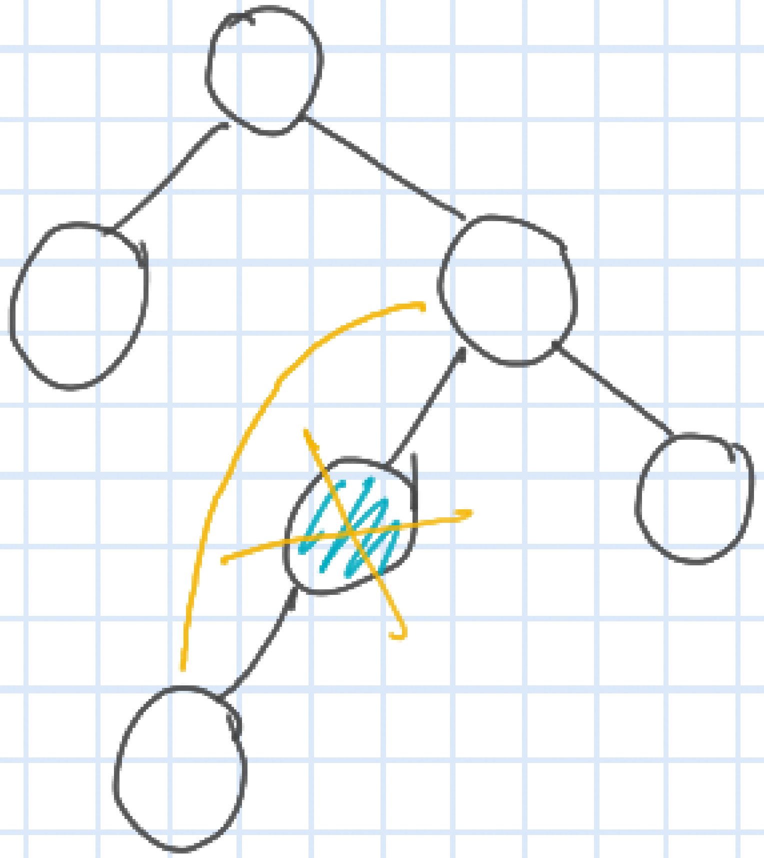
Deletion in binary search trees

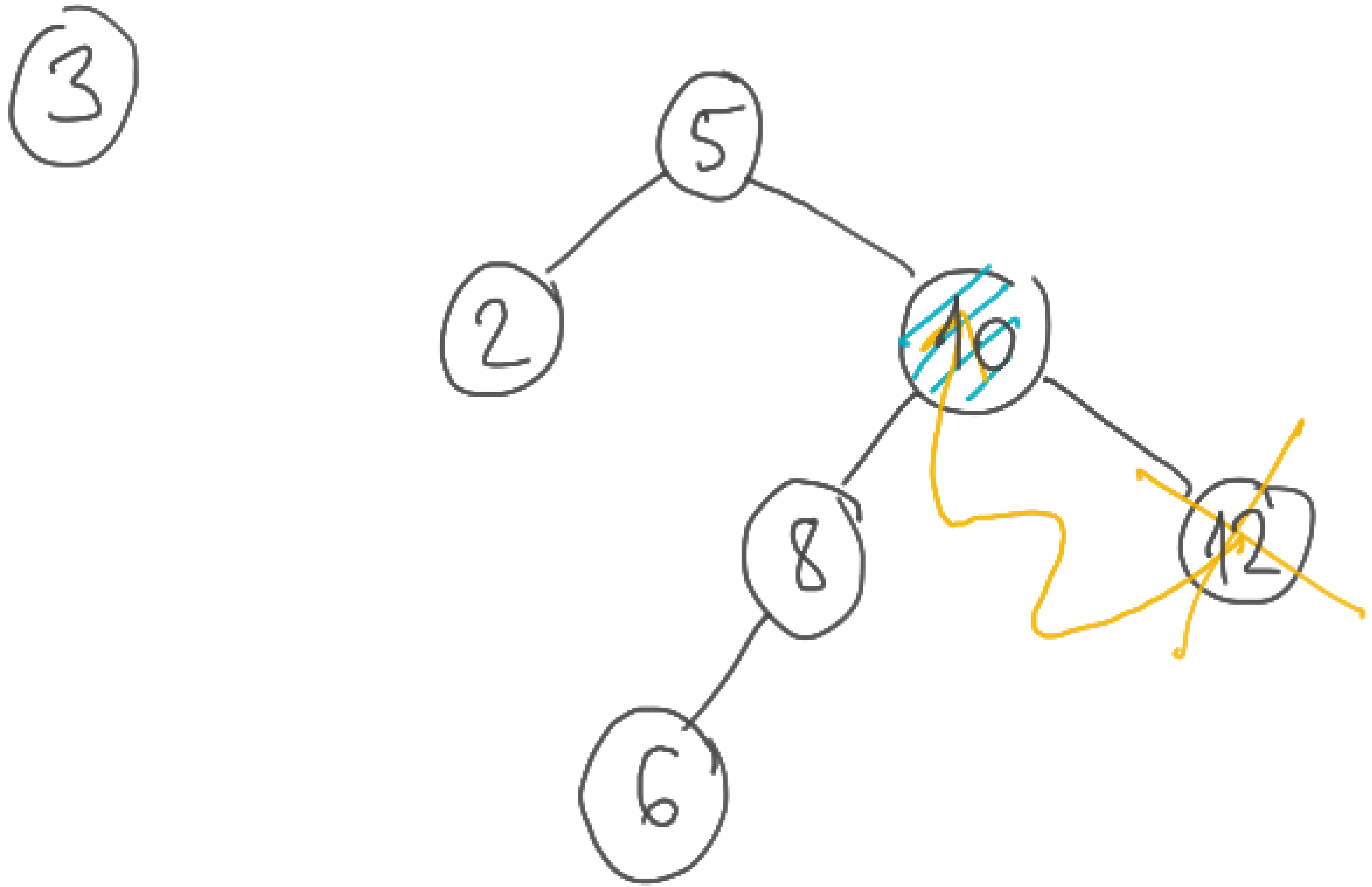
Three types

1



2





Next (it has 0 or 1 child)

```
Delete (T, z)
if z→left = nil or z→right = nil
  then y := z
  else y := Next(T, z)
/* y is the vertex we
   remove from T
if y→left ≠ nil
  then x := y→left
  else x := y→right
/* x is a not nil child of y, if such a child exists
```

z is the vertex we want to remove

in the third type we will actually remove another vertex → y

```
if x ≠ nil then
   x -> parent := y -> parent
if y -> parent = nil
   then
      root[T] := x
   else
      if y = y -> parent -> left
         then y -> parent -> left := x
         else y -> parent -> right := x
```

if $y \neq z$ then

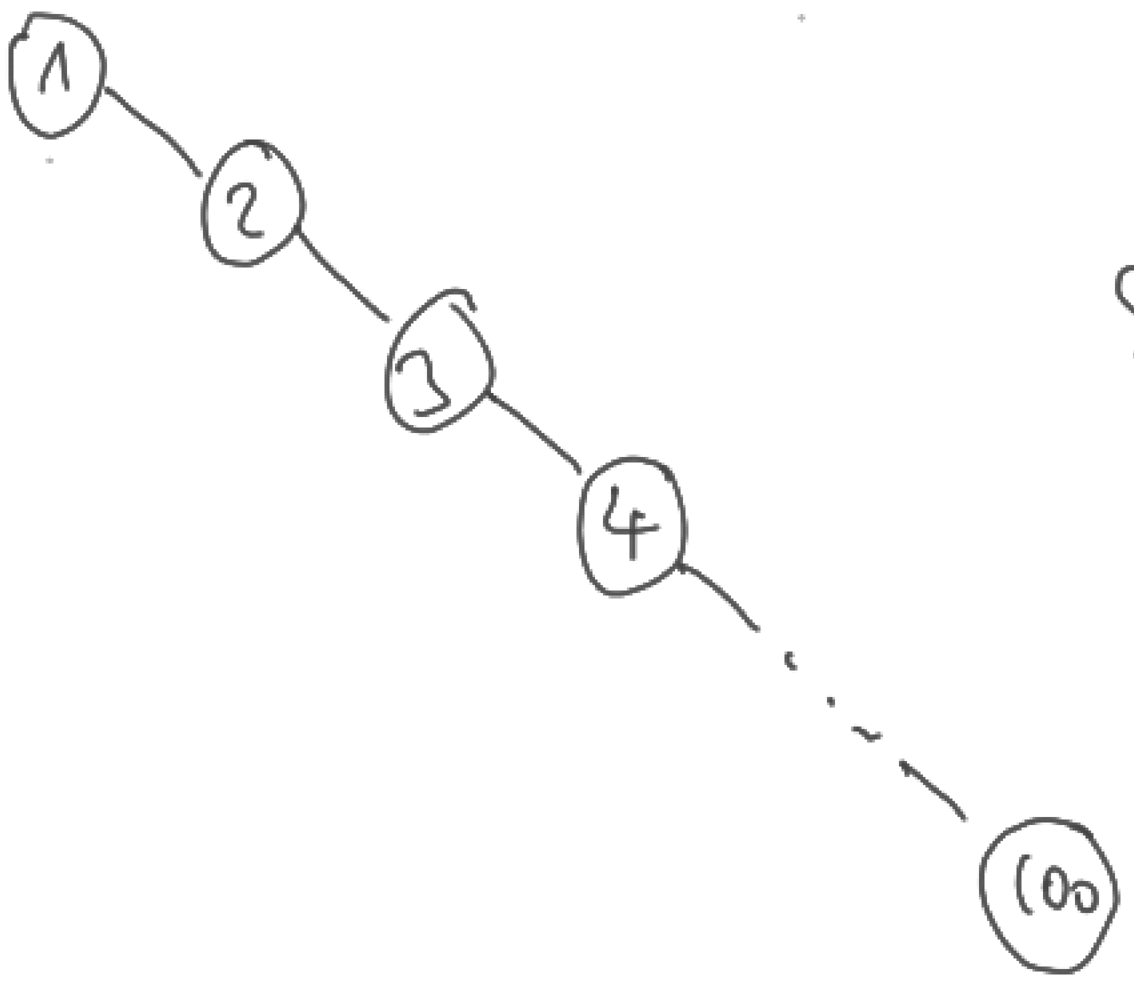
  $z \to key := y \to key$

return y

Cost: $O(h)$ [h is the depth of the BST]

Is a binary search tree an efficient data structure?

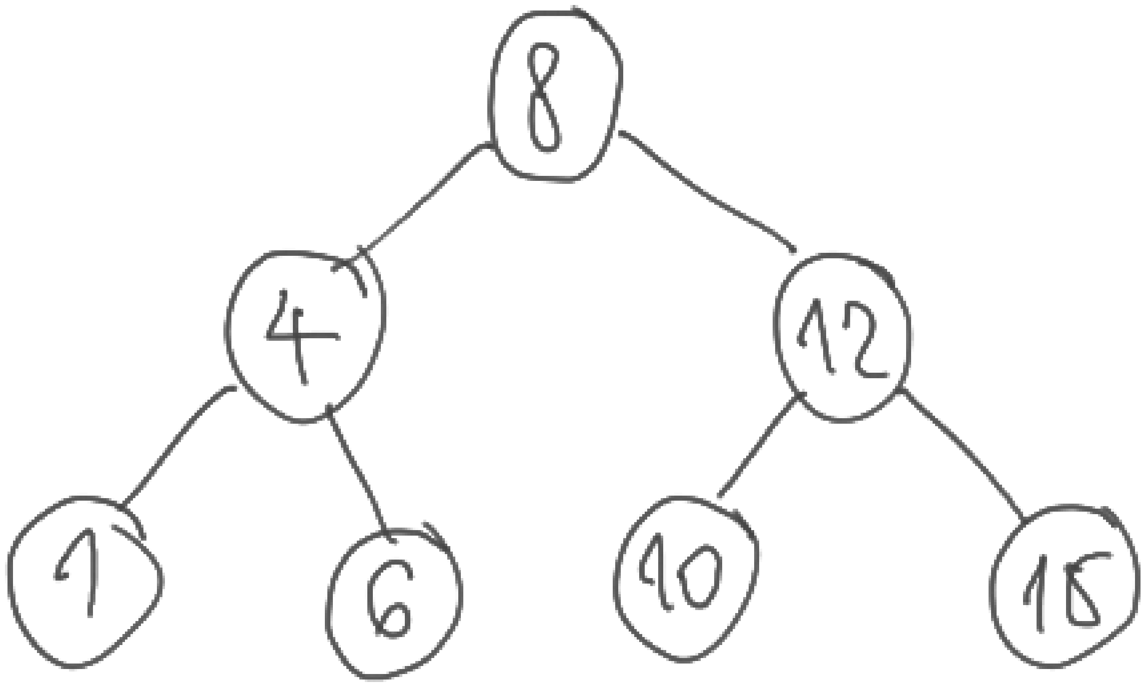
Not necessarily:



search cost:

$O(\#\text{keys})$

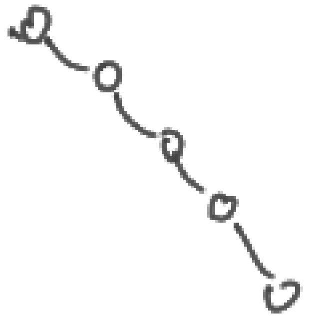
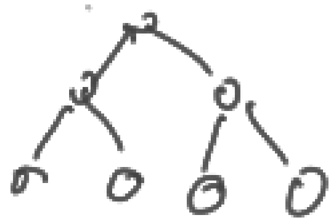
In the same time yes ;



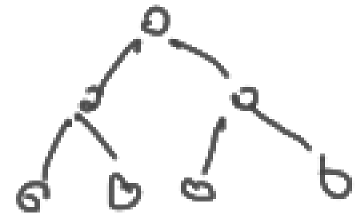
$O(h) = O(\log(\#\text{keys}))$

Problem

For a given set of data elements one can easily build such a balanced binary search tree.

However if you have to perform a huge number of insertions and deletions (dynamic data set) then after these operations the shape of the BST might be closer to  than to 

Question:

Can we guarantee somehow to remain close to the shape  after a lot of insertions and deletions?

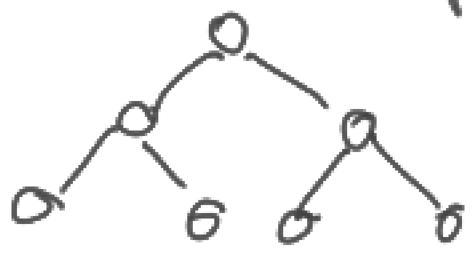
Yes, if we are allowed to reorganize a little the structure of the tree after a deletion or insertion.

What kind of reorganization can be imagined here?

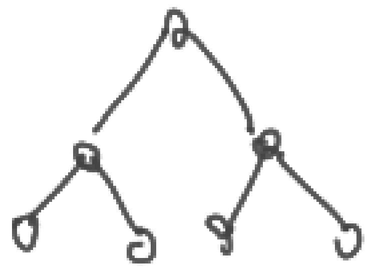
→ takes a little time (logarithmic)

→ keeps the binary search tree property

→ the shape will be close to



Next question

What "close to " means?

Natural approaches

(1) for any vertex the number of vertices in its left subtree and right subtree differ by a small number ☹

Use ratio instead of difference

$$\frac{1}{\alpha} \leq \frac{\#\text{ vertices of the left subtree}}{\#\text{ vertices of the right subtree}} \leq \alpha$$

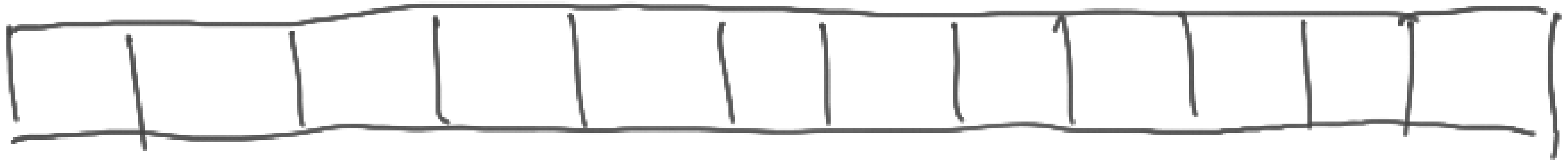
for appropriate α's

$\boxed{2}$ for any vertex the heights of its subtrees (left and right) differs by 0 or 1 $\longrightarrow$ AVL-trees

$$h = O(\log(\#\text{ vertices}))$$

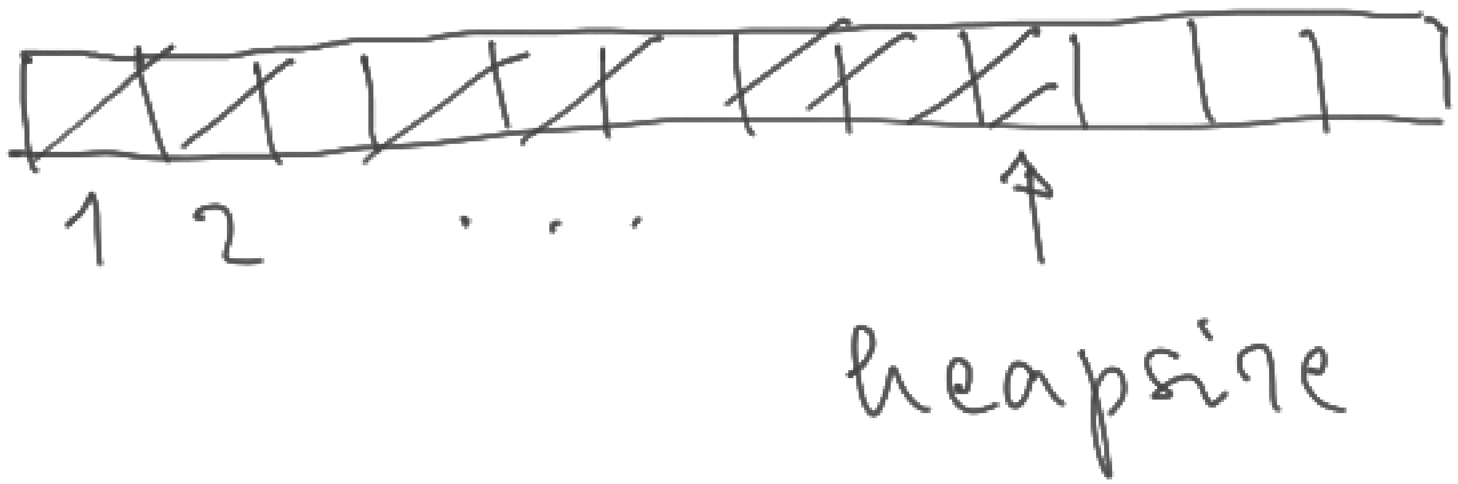
<u>Heap</u> (priority queue)

it is closely related to the heapsort



$A[1:n]$

We will store the data element here.
We won't use the whole array; n is an upper bound for the number of elements we can store in the heap.



Operations supported here

- insertion $\rightarrow O(\log(\text{heapsize}))$
- delete the smallest element from the heap $\rightarrow O(\log(\text{heapsize}))$

[there is a version where we can delete the max element]

<u>Trick</u>

We store the elements in the array in such a way that the min-heap property holds

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

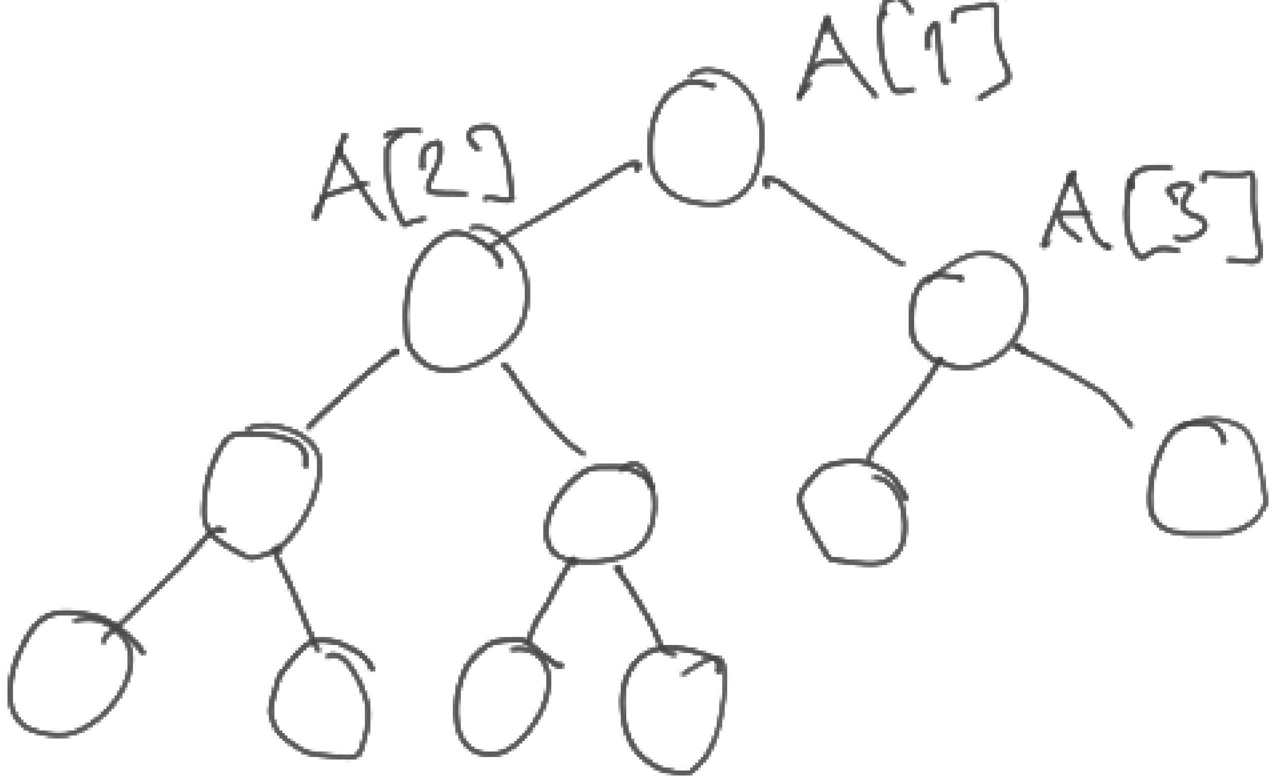
$A[2] \leq A[4], A[5]$

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

$\vdots$

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

Visual description (cf. heapsort)



A[i] is smaller than or equal to all of the element of its subtree