

## QuickSelect

Cormen 9.1 and 9.2

Median and order statistics

Input: An array  $A[1, n]$  and integer  $i$

Output: The element  $x \in A$  that is smallest than exactly  $i-1$  elements in  $A$

$$i = \frac{n}{2} \text{ median}$$

$$i = n \text{ min}$$

$$i = 1 \text{ max}$$

Mean, variance, Max, min can be computed in  $\Theta(n)$  time

What about  $i$ th element?

- Sort  $A$  and return  $A[i]$

$$\Theta(n \log n) \text{ time}$$

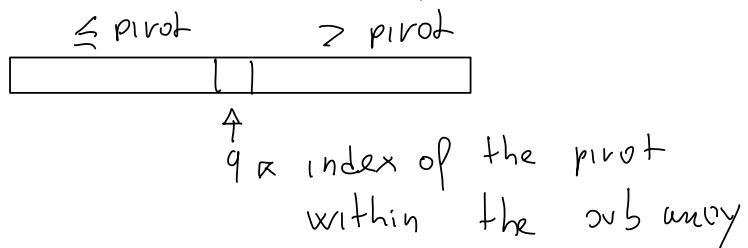
Can we do better? yes,  $\Theta(n)$  time algorithm  
(in the worst case)

QuickSelect ( $A, i$ ) use an approach similar to

Q Sort BUT it recursively go in just one of the two parts.

the one that we know contains the element we are looking for

**Divide** Choose (at random) a pivot and partition the sub array



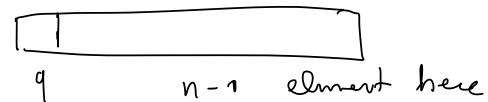
**Conquer** if  $i = q$  return pivot

if  $i < q$  recursively search  $i$ -th element in the first part

if  $i > q$  recursively search  $(i-q)+1$  element in the second part

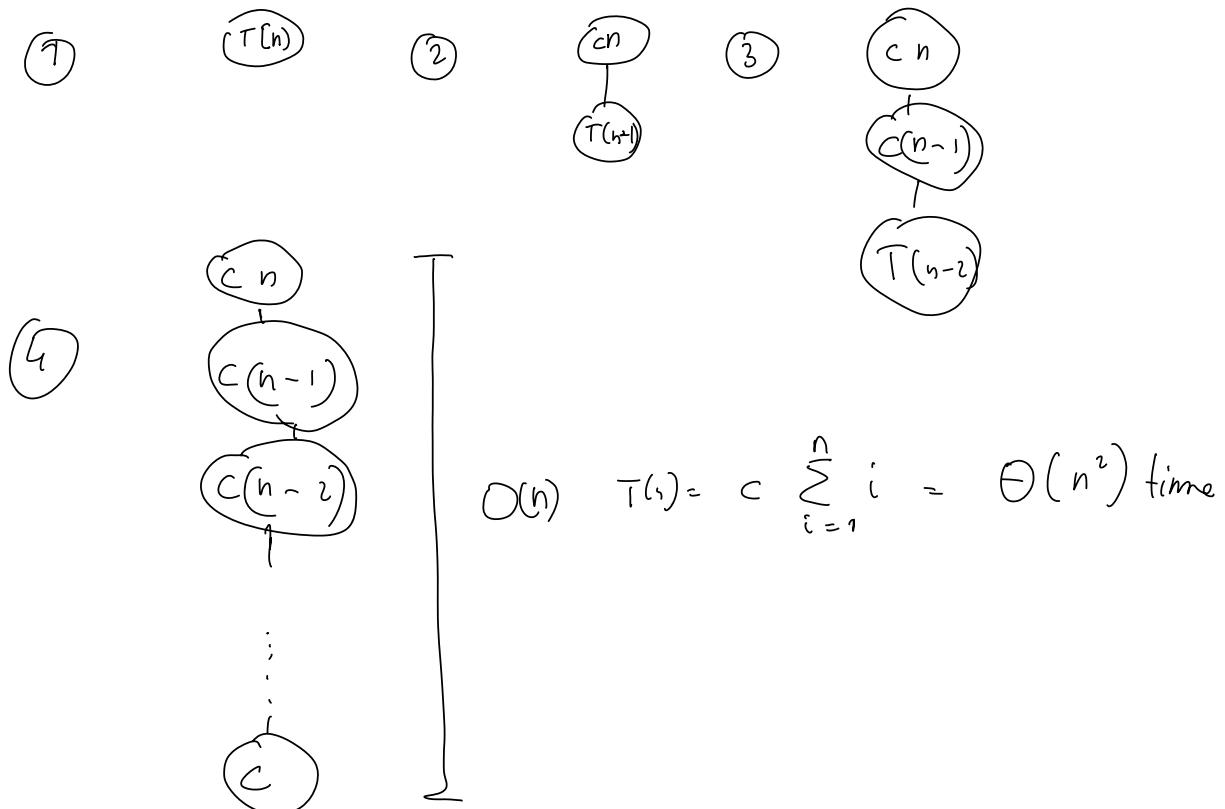
**Combine** nothing to do

Worst case

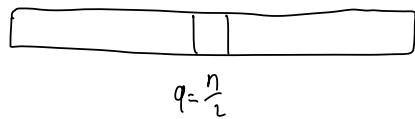


$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{otherwise} \end{cases}$$

$$T(n) = \Theta(n^2)$$



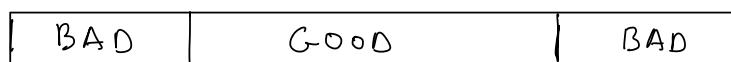
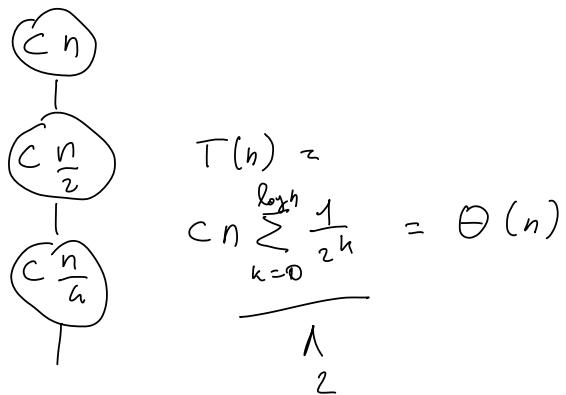
Best case



$$T(n) = \begin{cases} O(1) & \text{if } n \leq 1 \\ T\left(\frac{n}{2}\right) + \Theta(n) & \text{else} \end{cases}$$

$\uparrow$   
 $a = 1$  just one recursive call

$$T(n) \approx \Theta(n) + \text{time}$$



$$\frac{n}{4}$$

$$\frac{3}{4}n$$

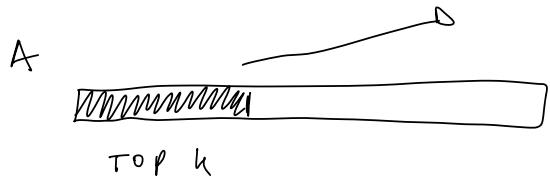
$$\frac{n}{2}$$

$$2n = \Theta(n)$$

$$\frac{n}{2}$$

$$n = \Theta(n)$$

$$\frac{n}{2}$$



A 8 6 9 1 2 4 3 5 10

We want 6 smallest element in A

1 2 4 3 5 6

Top k in  $\Theta(n)$  time but unsorted

$\Theta(n) + \Theta(k \log k)$  time if you want them sorted

## Priority Queue Cormen Ch 6

Problem (Max Priority Queue)

Build a Data Structure for maintaining a set  $S$  of keys (keys are the priorities of our items) to support the following operations

$\text{Insert}(S, x)$  insert the key  $x$  in  $S$   
 $S = S \cup \{x\}$

$\text{Maximum}(S)$  return the largest key in  $S$

$\text{Extract Max}(S)$  remove (and return) the largest key in  $S$

Several Applications eg Sorting

First trivial Solution

Store  $S$  in a unordered array  $A$

- $\text{Insert}(S, x)$   $O(1)$  just  $A.append(x)$
- $\text{Max}(S)$   $>$  Scan  $A$  to compute max
- $\text{Extract Max}(S)$   $\Theta(n)$

## Second trivial Solution

Store the keys in a sorted array A

- Insert ( $S, x$ )
  - 1  $A.append(x)$
  - 2 move  $x$  to the left until it finds its correct position
- Max ( $S$ )  $> O(1)$
- Extract Max ( $S$ )

	FTS	STS	Heap
Insert	$O(1)$	$\Theta(n)$	$\Theta(\log n)$
Max	$\Theta(n)$	$O(1)$	$O(1)$
Extract Max	$\Theta(n)$	$O(1)$	$\Theta(\log n)$

with FTS and STS Sorting takes  $\Theta(n^2)$  time

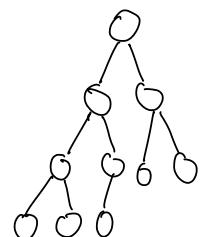
with heap Sorting takes  $\Theta(n \log n)$  time  
 (Heap Sort)

## Heap

A (binary) heap data structure is an array that can be seen as a complete binary tree

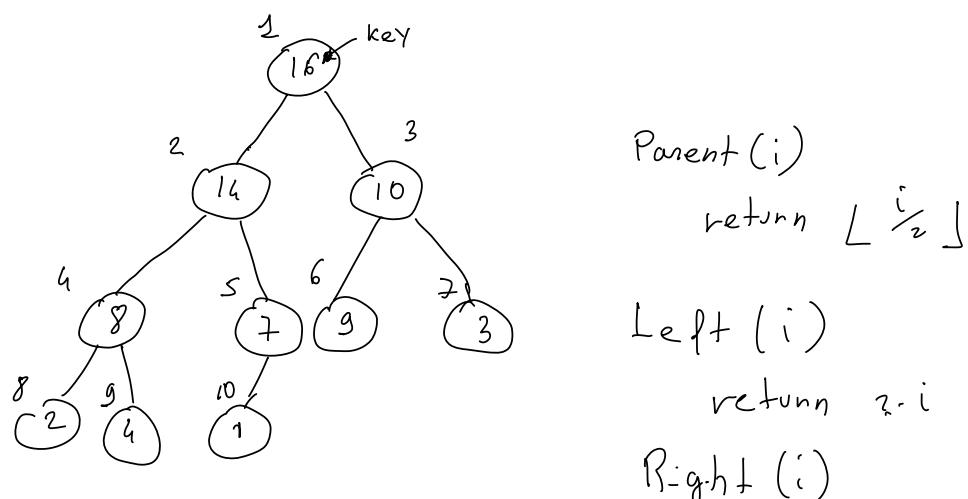
complete binary tree

tree is completely filled on all the levels except possibly at the lowest one which is filled from left



$\leftarrow$  1st level       $2^0$  nodes  
 $\leftarrow$  2nd level       $2^1$  nodes  
 $\leftarrow$  3rd level       $2^2$  nodes  
 $\leftarrow$  4th level       $\leq 2^3$  nodes

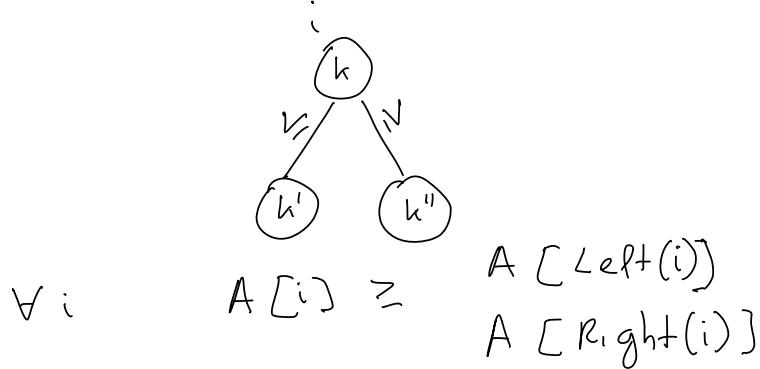
height of a tree of  $n$  nodes?  $\Theta(\log n)$



A	16	14	10	8	7	9	3	2	4	1	return $2 \cdot i + 1$
	1	2	3	4	5	6	7	8	9	10	

Main property : (Max) - Heap Property

$$\forall i \quad (i \neq 1) \quad A[\text{parent}(i)] \geq A[i]$$



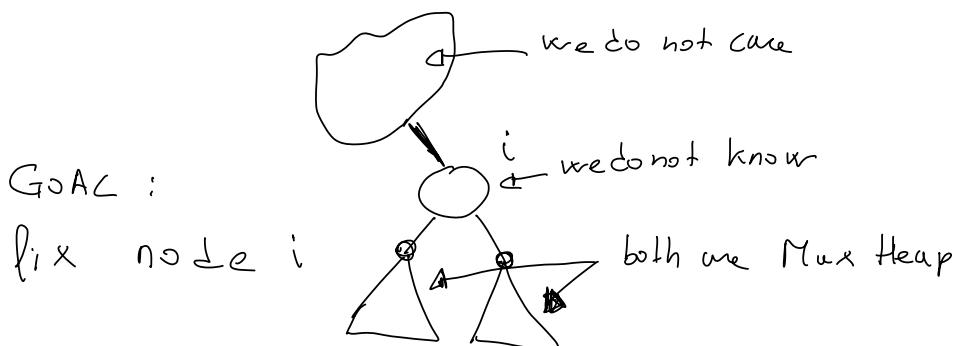
Maximum (S)  
return  $A[1]$   $O(1)$  time

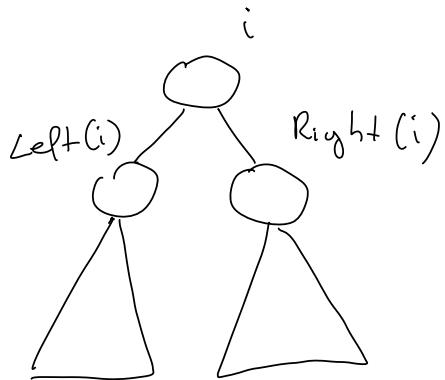
Maintaining the heap property

Max-Heapify ( $A, i$ )

called at node  $i$  and it assumes  
that both subtrees rooted at  $\text{Left}(i)$  and  
 $\text{Right}(i)$  are Max-heap

but  $A[i]$  may be smaller than its  
children. (violating the property)





- $A[i] \geq A[\text{Left}(i)]$

$$A[i] \geq A[\text{Right}(i)]$$

Ok nothing to do

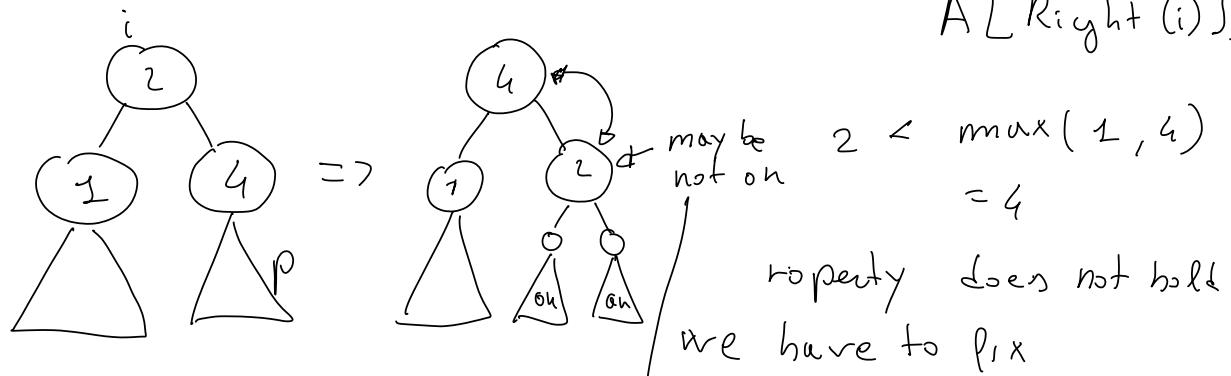
Heapsify stops

- $A[i] < A[\text{Left}(i)]$

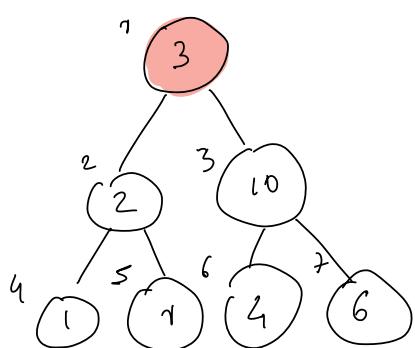
or

$$A[i] < A[\text{Right}(i)]$$

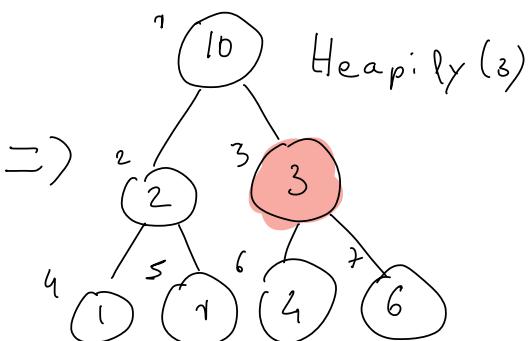
$$\Rightarrow A[i] < \max(A[\text{Left}(i)], A[\text{Right}(i)])$$

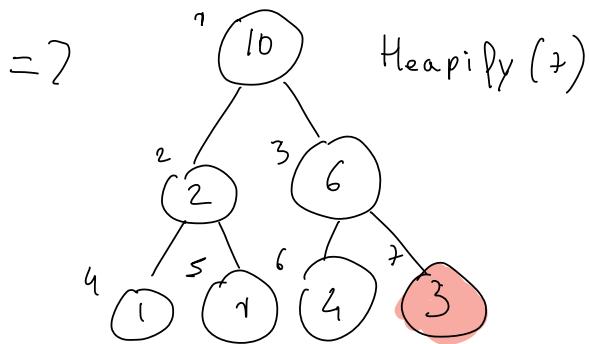


Heapsify( $i$ )

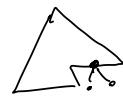


$\Rightarrow$  Heapsify( $3$ )





Max-Heapify ( $A, i$ )



1  $l = \text{Left}(i)$

2  $r = \text{Right}(i)$

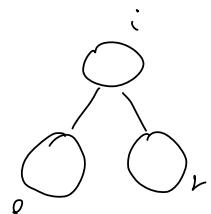
3 if  $l \leq A.\text{heap\_size}$  and  $A[l] > A[i]$ :

4      largest =  $l$

5 else     largest =  $i$

6 if  $r \leq A.\text{heap\_size}$

and  $A[r] > A[\text{largest}]$



7      largest =  $r$

8 if  $\text{largest} \neq i$

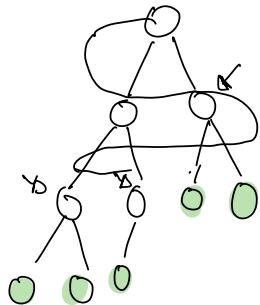
9      swap ( $A[i], A[\text{largest}]$ )

10     Max-Heapify ( $A, \text{largest}$ )

Complexity  $\hookrightarrow \Theta(h) = \Theta(\log n)$

## Build a Heap

We start from an unordered array A



Build ~ Max \_ Heap( A )

1 A.heap\_size = A.length

2 For i =  $\lfloor A.length/2 \rfloor$  down to 1

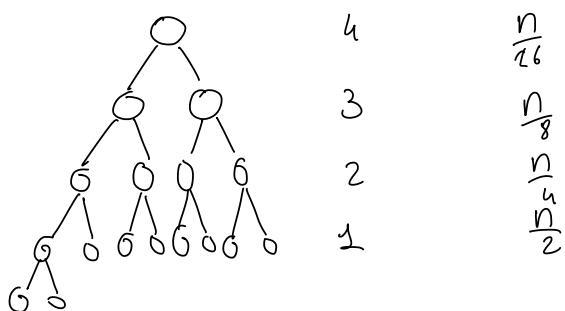
3 Max\_heapify ( A , i )

Complexity trial analysis

$\Theta(n)$  iterations each costs  $O(\log n)$   
 $\Rightarrow O(n \log n)$  time

Better analysis

Cost heapify # nodes

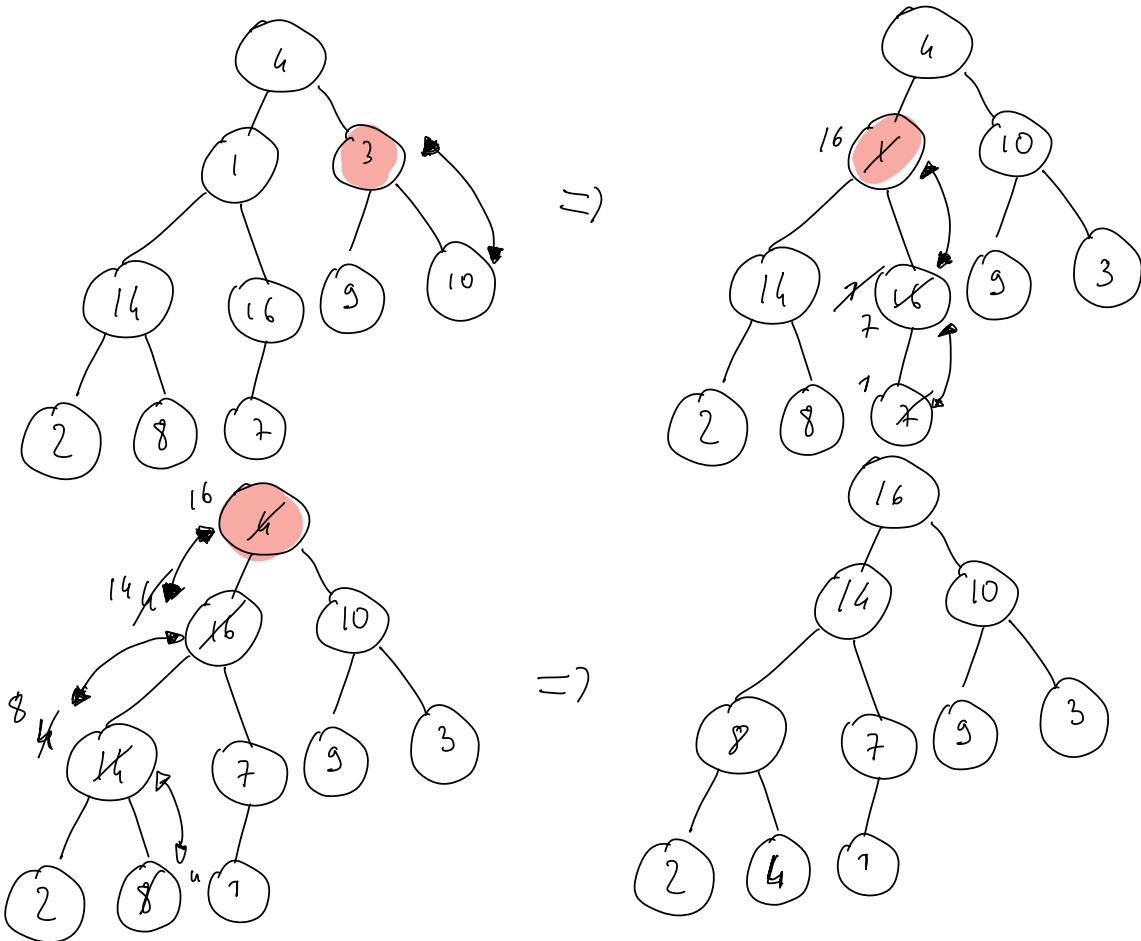
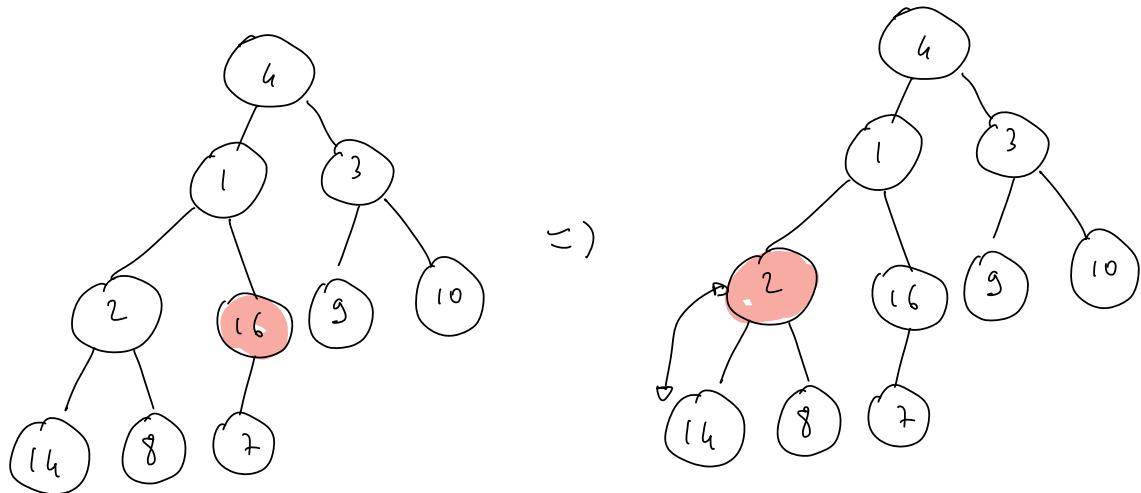


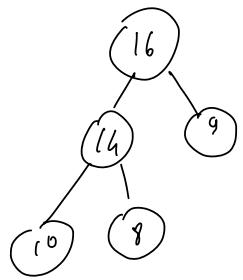
$$\begin{aligned}
 T(n) &= \sum_{h=1}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h-1}} \right\rceil O(h^{-1}) \\
 &= O\left(n \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^{h-1}}\right) = O(n)
 \end{aligned}$$

$$\begin{aligned}
 0 &+ \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} \\
 &\quad \overbrace{\qquad\qquad\qquad}^{\frac{1}{2}} + \overbrace{\frac{1}{4}}^{\frac{1}{2}} + \overbrace{\frac{1}{8}}^{\frac{1}{4}}
 \end{aligned}$$

A

4	1	3	2	16	9	10	14	8	7
7	2	3	4	5	6	7	8	9	10





2nd largest value could be  
at 3rd level

## Extract - Max

Idea:

- Swap  $A[1]$  with  $A[A.\text{heapsize}]$
- remove  $A[A.\text{heapsize}]$
- Run Heapsify ( $A, 1$ )

### Max Extract ( $A$ )

- 1 if  $A.\text{heapsize} < 1$
- 2 return "Error"
- 3 max =  $A[1]$
- 4  $A[1] = A[A.\text{heapsize}]$
- 5  $A.\text{heapsize} = A.\text{heapsize} - 1$
- 6 Max-Heapsify ( $A, 1$ )
- 7 return max

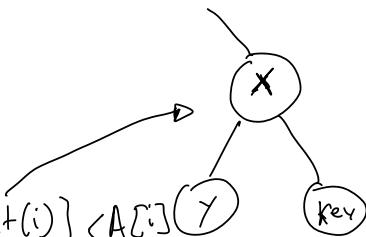
Complexity =  $\Theta(\frac{\log n}{\text{time}})$

## Increase key

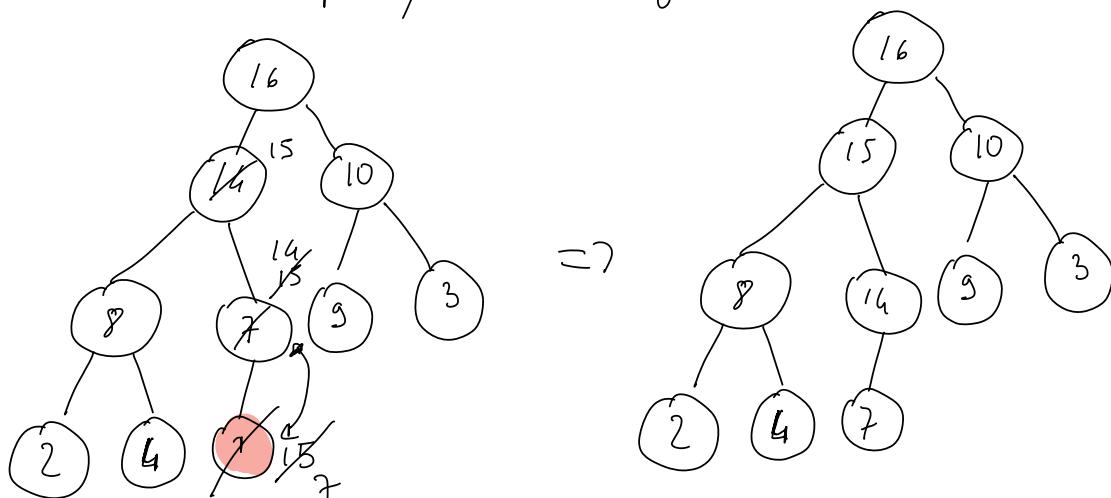
We want to increase an existing key in A

Increasekey (A, i, key)

- 1 if key < A[i]
- 2 return "Error"
- 3 A[i] = key
- 4 while i > 1 and A[parent(i)] < A[i]  
5 swap (A[parent(i)], A[i])  
6 i = parent(i)



Complexity :  $\Theta(\log n)$  time



## Insert a key

Insert (A, key)

Complexity :  $\Theta(\log n)$   
time

- 1 A.heap size = A.heap size + 1
- 2 A[A.heap size] = -∞
- 3 Increase key (A, A.heap size, key)

Decrease key (A, i, key)

(Assume  $A[i] \geq \text{key}$ )

Decrease key (A, i, key)

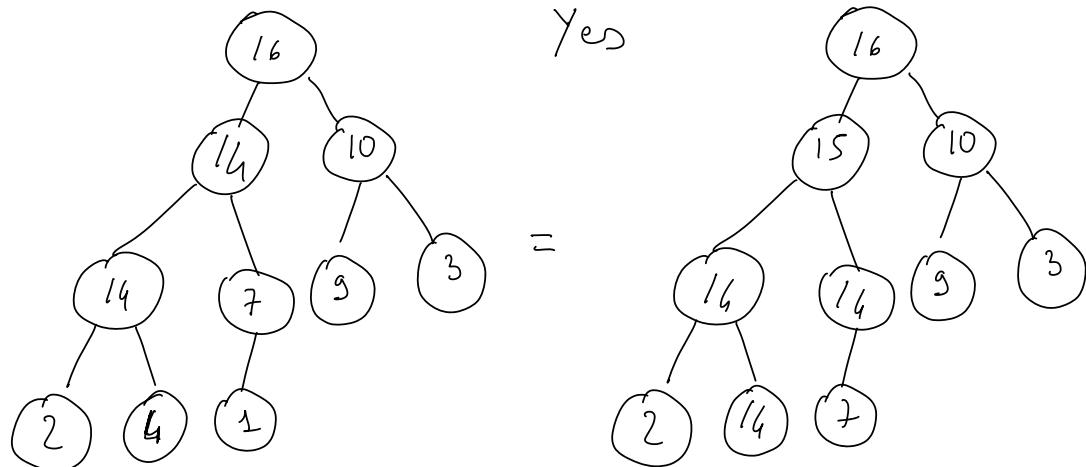
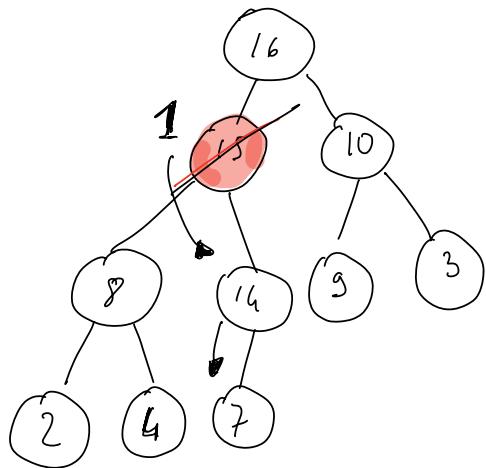
1 If  $\text{key} > A[i]$

2 return "Error"

3  $A[i] = \text{key}$

4 MaxHeapify (A, i)

Complexity  $\Theta(\log n)$   
time

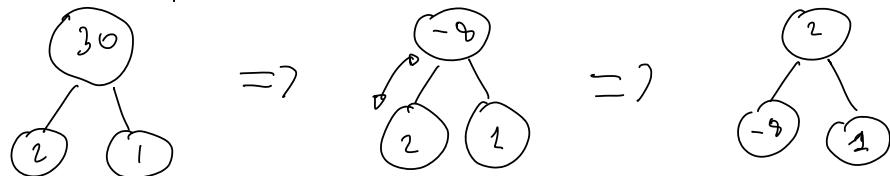


Delete ( $A, i$ )

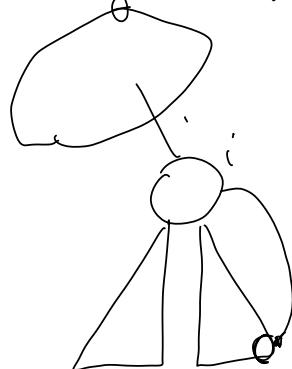
- 1 DecreaseKey( $A, i, -\infty$ )
- 2  $A.\text{heapsize} = A.\text{heapsize} - 1$

INCORRECT

Counterexample



Correct Algorithm for Delete ( $A, i$ )



Delete ( $A, i$ )

- 1 if  $A.\text{heapsize} < i$
- 2 return "Error"
- 3  $A[i] = A[A.\text{heapsize}]$
- 4  $A.\text{heapsize} = A.\text{heapsize} - 1$
- 5 Max\_Heapify ( $A, i$ )

Complexity  $\Theta(\log n)$   
time

Exercise

Write the pseudocode of all the ops for Min Heap