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

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

$i = n$ min

$i = 1$ max

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

What about ith element?

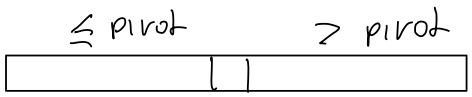
- Sort A and return $A[i]$

  $\Theta(n \log n)$ time

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

QuickSelect (A, i) use an approach similar to Q Sort <u>BUT</u> 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



q ← index of the pivot within the sub array

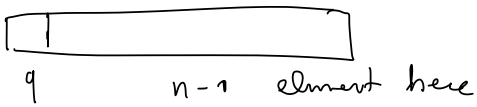
Conquer if $i = q$ return pivot

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

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

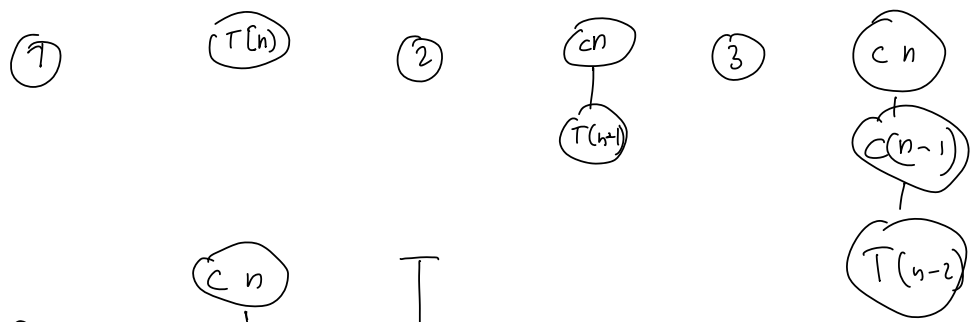
Combine nothing to do

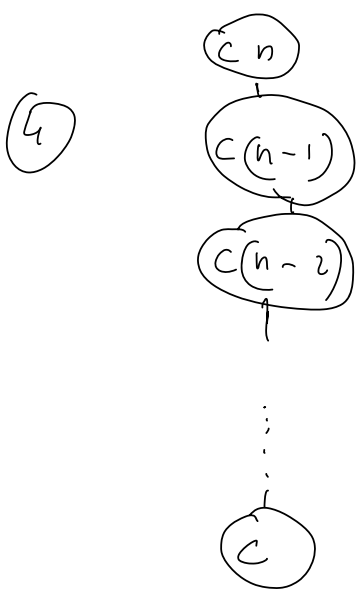
Worst case



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

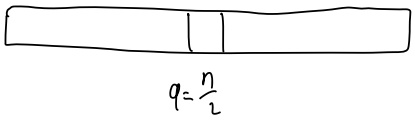
$$T(n) = \Theta(n^2)$$





$O(n)$ $T(n) = c \sum_{i=1}^{n} i = \Theta(n^2)$ time

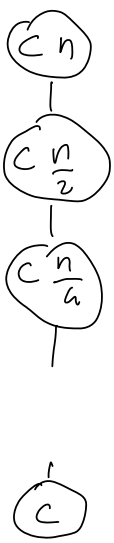
Best case



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

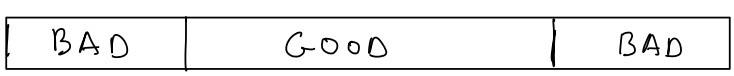
$a = 1$ just one recursive call

$T(n) = \Theta(n)$ time



$$T(n) = cn \sum_{k=0}^{\log n} \frac{1}{2^k} = \Theta(n)$$

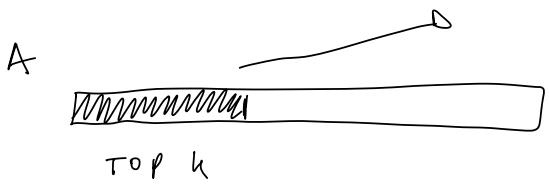
$\sum \leq 2$



n
- BAD
- GOOD $\leq \frac{n}{2}$

$2n = \Theta(n)$

q GOOD $\leq \frac{n}{2}$ $\quad n = \Theta(n)$



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

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

Maximum (S) return the largest key in S

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

- Insert (S, x) $O(1)$ just A.append(x)
- Max (S)
- Extract Max (S)

> Scan A to compute max
> $\Theta(n)$

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 it's correct position
- Max (S)
- ExtractMax (S)

> $O(1)$

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

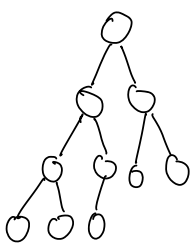
with FTS and STS Sorting takes $\Theta(n^2)$ time

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

Heap

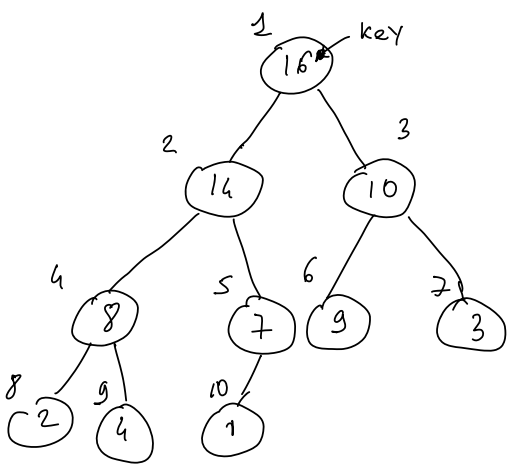
A (binary) heap data structure is an array that can be seen as a complete binary tree

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



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

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



Parent(i)
return $\lfloor i/2 \rfloor$

Left(i)
return $2 \cdot i$

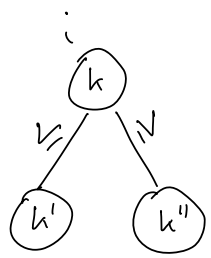
Right(i)
return $2 \cdot i + 1$

A

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

Main property : (Max) - Heap Property

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



$\forall i \quad A[i] \geq A[\text{Left}(i)]$, $A[\text{Right}(i)]$

Maximum (S)

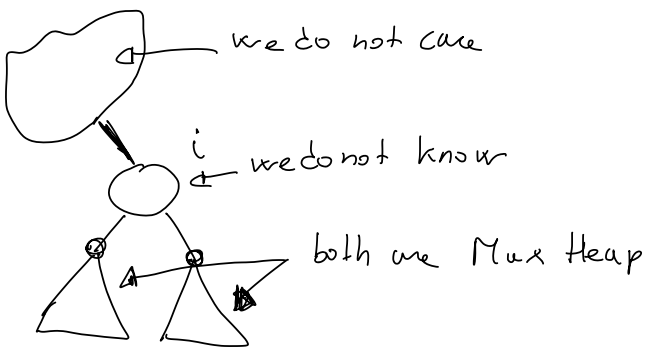
return A[1]

O(1) time :)

Maintaining the heap property

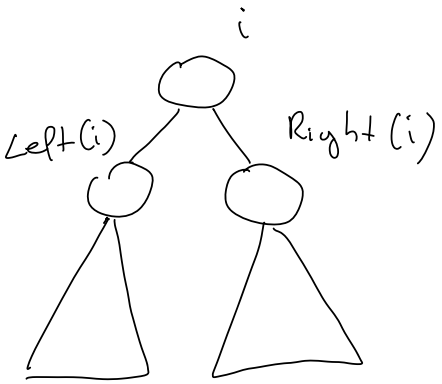
Max-Heapify (A, i)

called at node i and it assumes that both subtrees rooted at Left(i) and Right(i) are Max-Heap but A[i] may be smaller than its children. (violating the Property)



GOAL :

fix node i



- $A[i] \geq A[Left(i)]$
  $A[i] \geq A[Right(i)]$

  Ok nothing to do

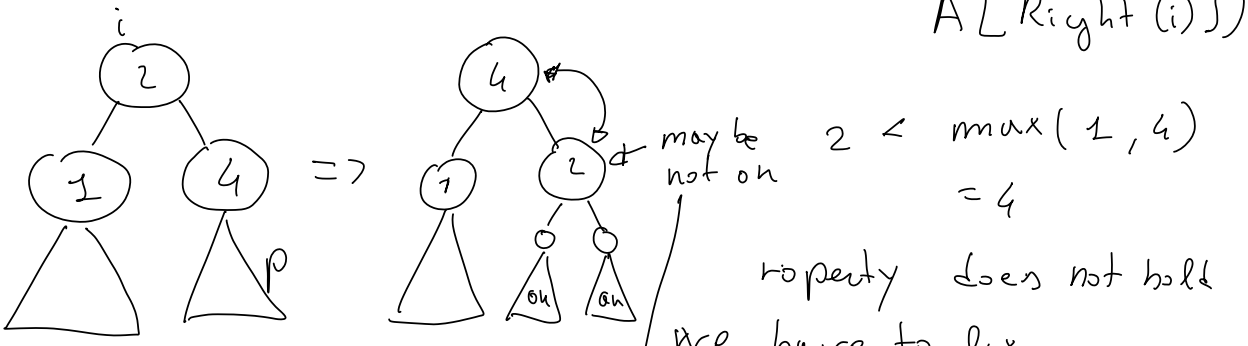
  Heapify Stops

- $A[i] < A[Left(i)]$

  or

  $A[i] < A[Right(i)]$

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



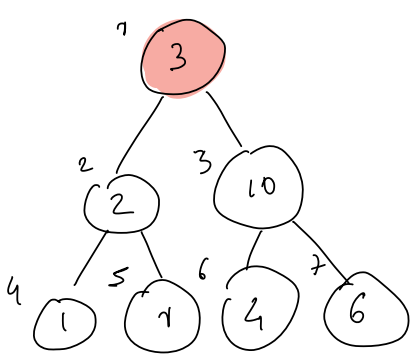
$2 < \max(1, 4) = 4$

property does not hold

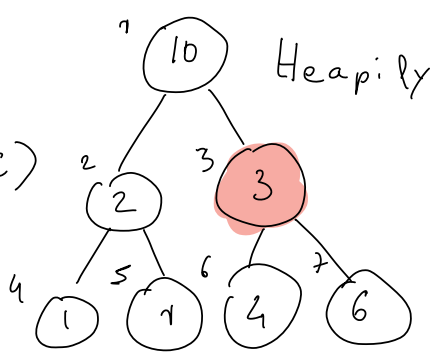
we have to fix

Heapify(1)

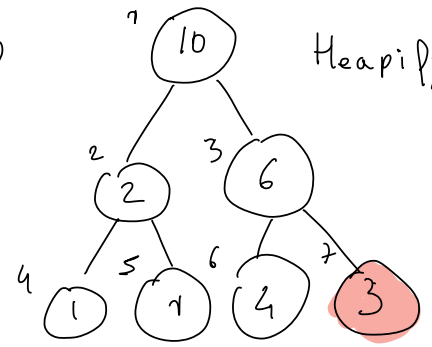
. call Heapify(Right(i))



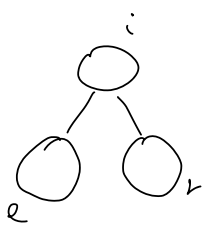
=>



=>



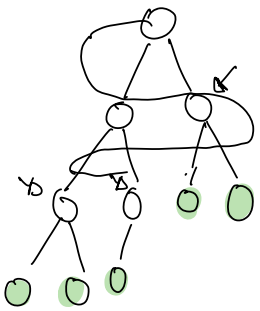
```
Max-Heapify (A, i)
1   l = Leaf(i)
2   r = Right(i)
3   if l ≤ A.heap_size and A[l] > A[i]:
4       largest = l
5   else largest = i
6   if r ≤ A.heap_size
        and A[r] > A[largest]
7       largest = r
8   if largest =/= i
9       swap(A[i], A[largest])
10      Max_Heapify(A, largest)
```



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

Build a Heap

We start from an unordered array A



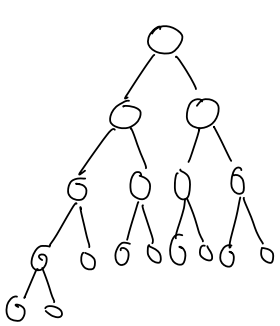
```
Build_Max_Heap(A)
1  A.heap_size = A.length
2  for i = ⌊A.length/2⌋ down to 1
3      Max_Heapify(A, i)
```

Complexity trivial analysis

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

$\Rightarrow O(n \log n)$ time

Better analysis



Cost Heapify	# nodes
4	$\frac{n}{16}$
3	$\frac{n}{8}$
2	$\frac{n}{4}$
1	$\frac{n}{2}$

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

$$= O\left(n \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^{h-1}} \right) = O(n)$$

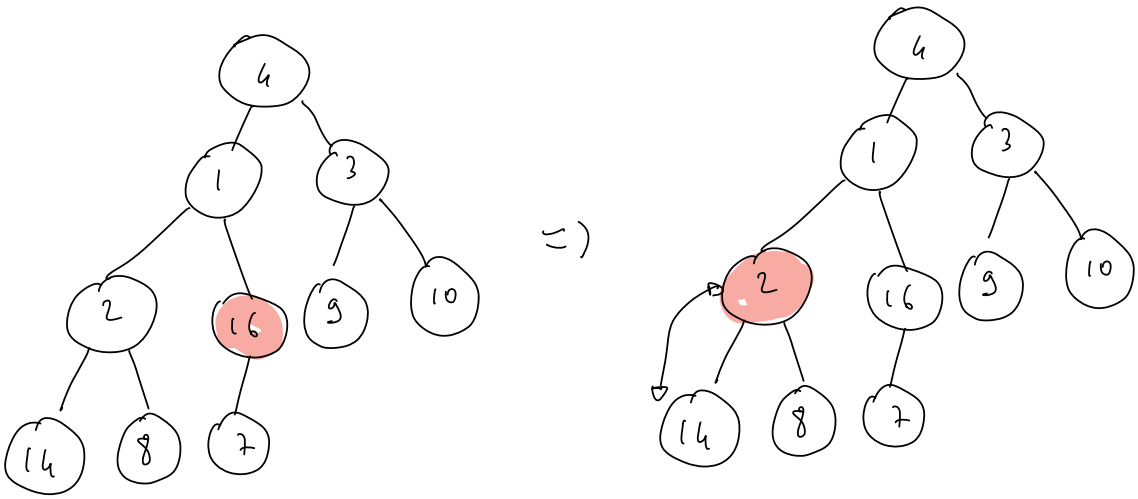
$\sum \frac{h}{2^{h-1}} \Rightarrow O(1)$

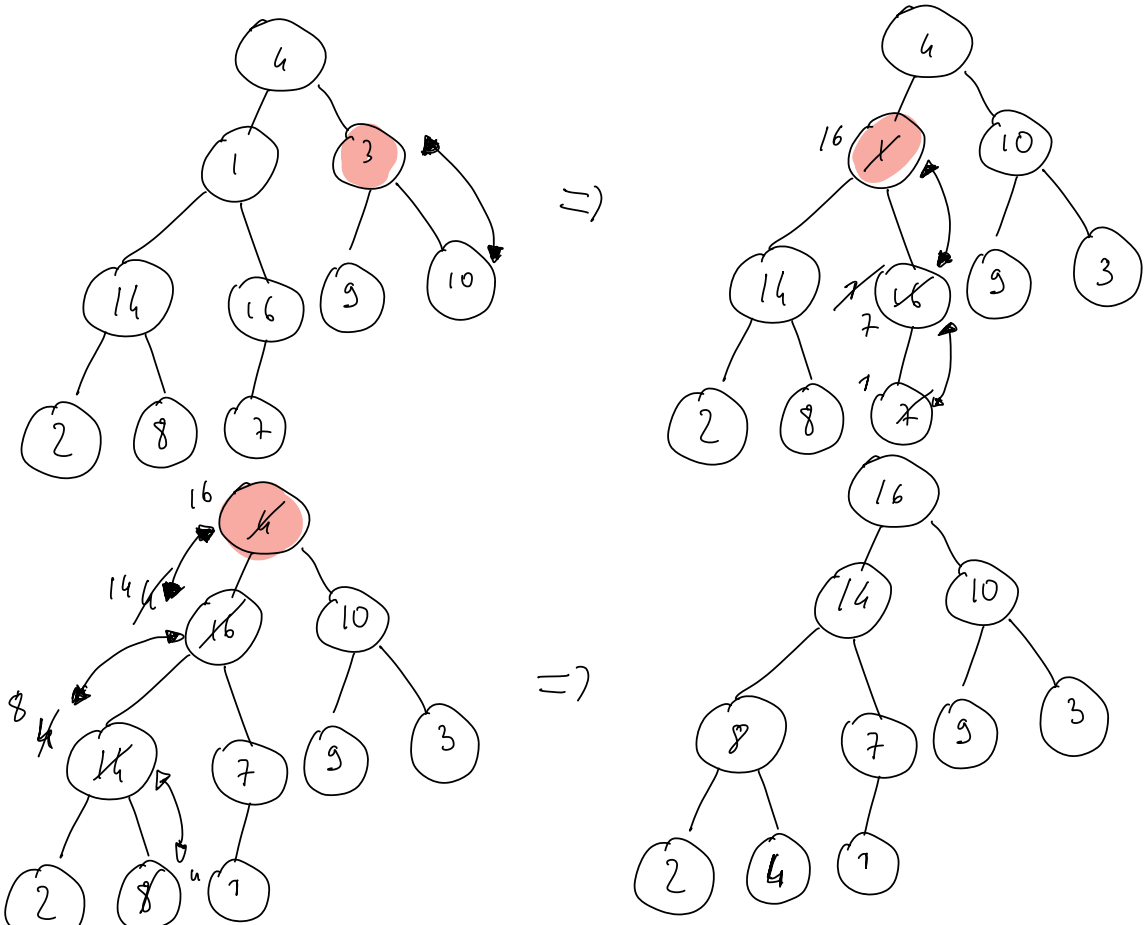
$$0 + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16}$$

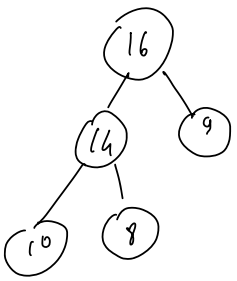
$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8}$$

A

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







3rd largest value could be at 3rd level

Extract - Max

Idea:
- Swap A[1] with A[A.heapsize]
- remove A[A.heapsize]
- Run Heapify (A, 1)

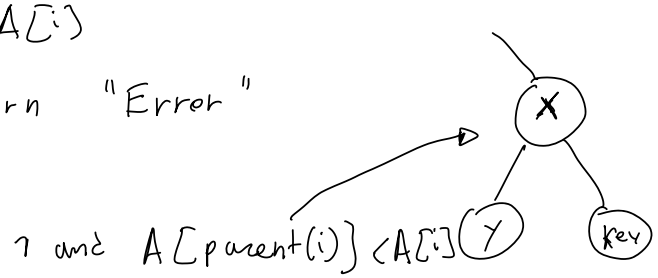
```
Max Extract (A)
1  if A.heapsize < 1
2        return "Error"
3  max = A[1]
4  A[1] = A[A.heapsize]
5  A.heapsize = A.heapsize - 1
6  Max-Heapify (A, 1)
7  return max
```

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

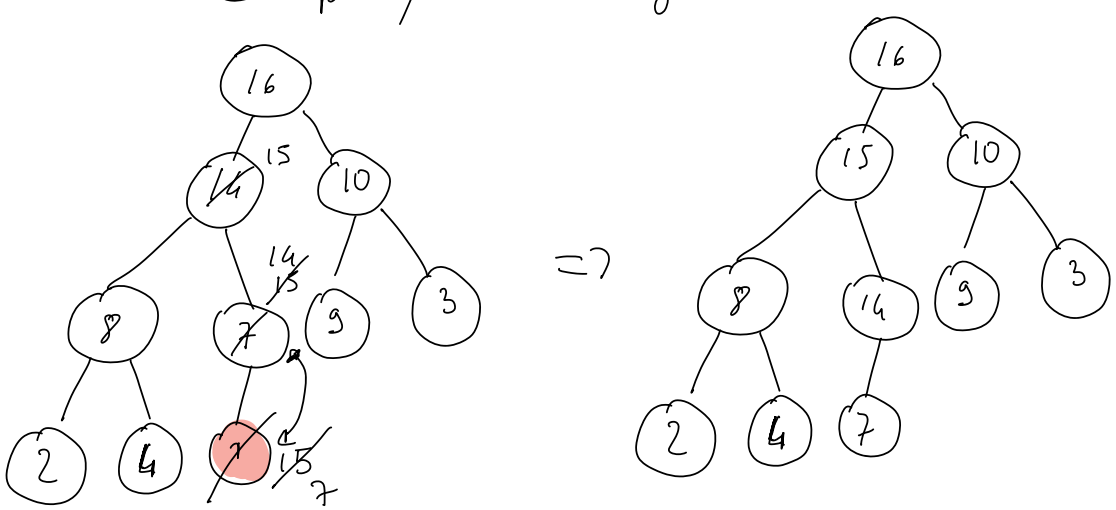
Increase key

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

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

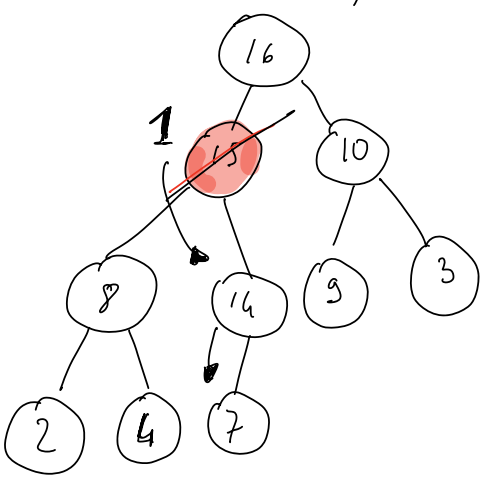
```
Insert ( A, key)
1   A.heapsize = A.heapsize + 1
2   A[A.heapsize] = - ∞
3   Increase key ( A, A.heapsize, key)
```

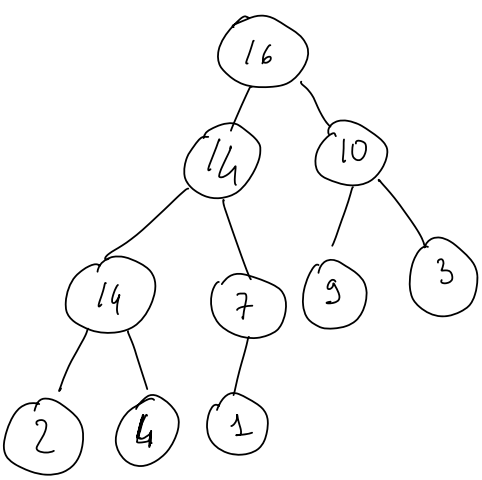
Decrease key (A, i, key)

(Assume $A[i] \geq key$)

```
Decrease key (A, i, key)
1  if key > A[i]
2        return "Error"
3  A[i] = key
4  Max Heapify (A, i)
```

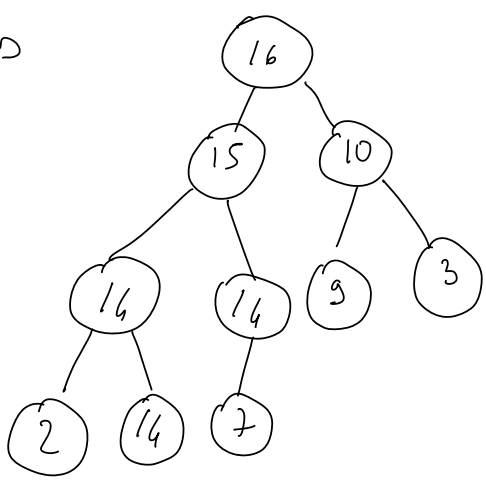
Complexity $\Theta(\log n)$ time





Yes

=



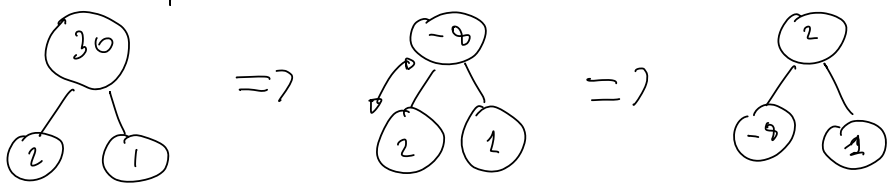
Delete (A, i)

1 DecreaseKey(A, i, $-\infty$)

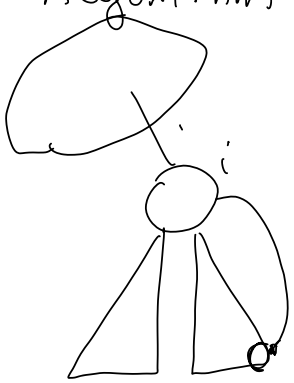
2 A.heapsize = A.heapsize - 1

INCORRECT

Counterexample



Correct Algorithm for Delete (A, i)



Delete (A, i)

1 if A.heapsize < i

2 return "Error"

3 A[i] = A[A.heapsize]

4 A.heapsize = A.heapsize - 1

5 Max Heapify (A, i)

Complexity $\Theta(\log n)$ time

Exercise

Write the pseudocode of all the ops for Min Heap