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 larger than exactly $i-1$ other elements of A

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

$i = 1$ Maximum

$i = n$ Minimum

Mean, Variance, Max, Min can be computed in $\Theta(n)$

What about other i?

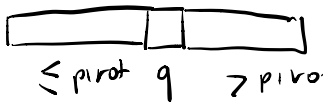
- Sort A and return $A[\frac{n}{2}]$

$\Theta(n \log n)$ time

Can we do better?

QuickSelect (A, i) uses an approach similar to QuickSort BUT it recursively goes just on 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 in



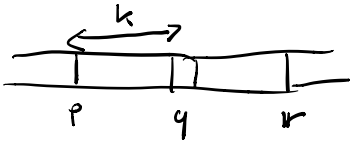
Conquer if $i == q$ return pivot and stop

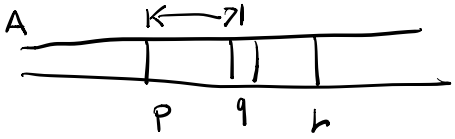
if $i < q$ recursively search i-th element in the first part

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

Combine nothing to do!

```
QuickSelect (A, i, p, r)
    if p == r
        return A[p]
    q = Partition (A, p, r)
    k = q - p + 1
    if i == k
        return A[q]
    if i < k
        QuickSelect (A, i, p, q-1)
    else
        QuickSelect (A, i-k, q+1, r)
```





Use QS to find $i = 4$ on

	1	2	3	4	5	6	7	8	9	10
A =	3	2	9	0	7	5	4	8	6	1

partition: 0 | 1 | 3 2 9 7 5 4 8 6

p=1 q=2 r=10

$i = 4 - 2$

3 2 5 4 | 6 | 9 7 8

q=7 k=5 r=10

$i = 2$

3 2 | 4 | 5

q=5 k=3

$i = 2$

| 2 | 3

q=3 k=1 p=4 r=4

$i = 2 - 1 = 1$

worst case

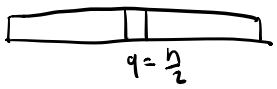
q

search here !!!

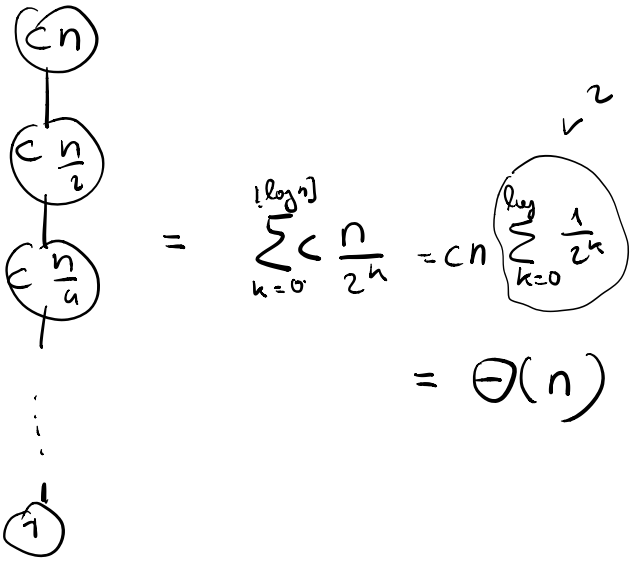
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{otherwise} \end{cases}$$

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

"Best Case"



$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \underbrace{T\left(\frac{n}{2}\right)}_{\uparrow} + \Theta(n) & \text{otherwise} \end{cases}$$



QS runs in $\Theta(n)$ expected time

Analysis is very similar to the one for Quicksort

Priority Queue Cormen Ch 6

Problem (Max priority queue)

Build a <u>data structure</u> 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 into the set S
i.e., $S = S \cup \{x\}$

Maximum(S) returns the largest key in S

Extract-Max(S) removes and returns the largest key in S

Several applications e.g. Sorting

Trade offs between the operations

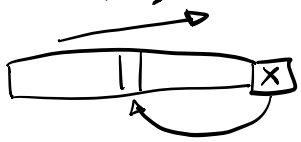
Trivial trade off

First trivial solution

- Store S in an unordered array A
- Insert(S, x) → A.append(x) $\Theta(1)$
- Maximum(S) → Scan A $\Theta(n)$
- Extract-Max(S) → Scan A and move keys

Second trivial solution

- Store S in a sorted array A
- Insert (S, x) → A.append(x) and keep A sorted // $\Theta(n)$



- Maximum(S) → returns $A[n]$
- Extract-Max(S) → remove last element of A // $O(1)$

	FTS	STS	HEAP
Insert	$O(1)$	$\Theta(n)$	$\Theta(\log n)$
Max	$\Theta(n)$	$O(1)$	$O(1)$
E Max	$\Theta(n)$	$O(1)$	$\Theta(\log n)$

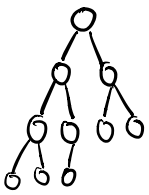
with FTS or STS Sorting in $\Theta(n^2)$

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

Heap : an efficient priority queue

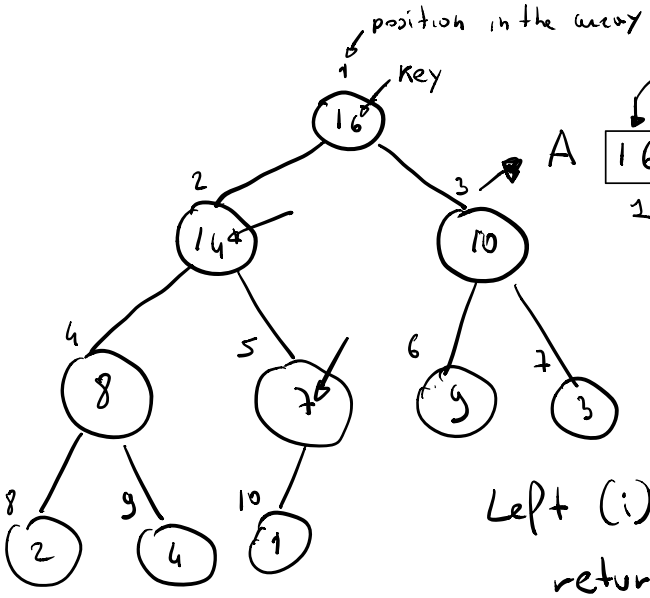
A (binary) heap data structure is an array that can be view as a complete binary tree

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



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

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



A

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

```
Parent(i)
    return ⌊i/2⌋
```

```
Left(i)
    return 2·i
```

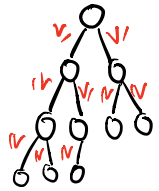
```
Right(i)
    return 2·i + 1
```

(Max)-heap property

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

$$= \quad \forall_i \qquad A[i] \geq \begin{matrix} A[Left(i)] \\ A[Right(i)] \end{matrix}$$

```
Maximum (S)
    return A[1]
```

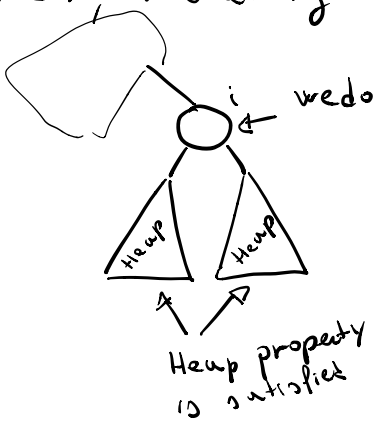


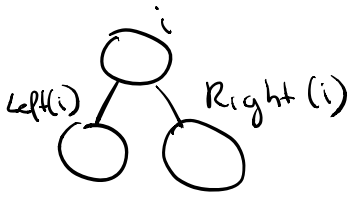
Complexity: $O(1)$ 

Maintaining the Heap Property

```
Max-Heapify (A, i)
```

when this op is called at position i it assumes that subtrees rooted at Left(i) and Right(i) are Max-Heaps but A[i] may be smaller than its children, violating Heap property



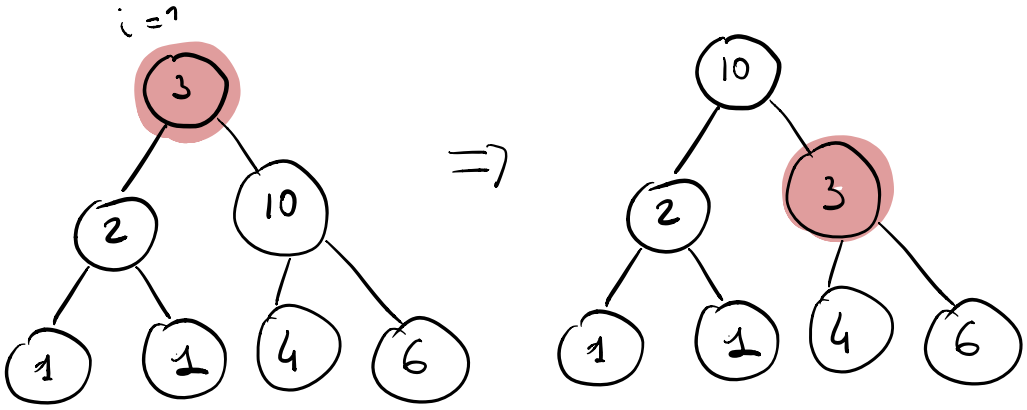


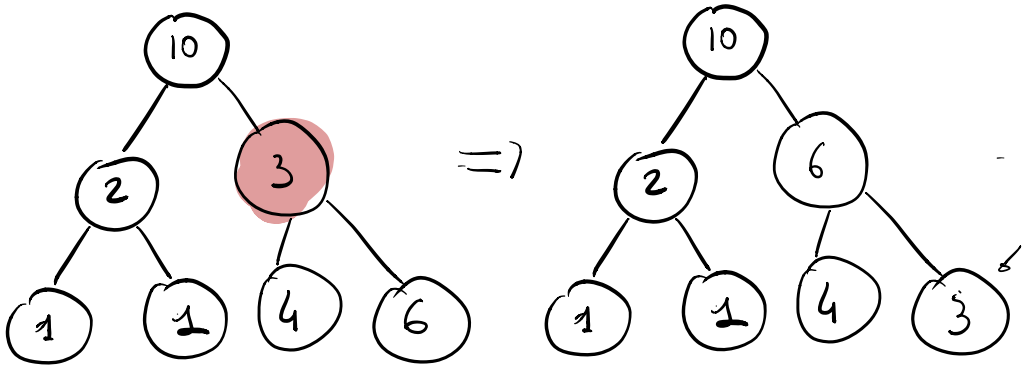
- $A[i] \geq A[Left(i)]$ AND $A[i] \geq A[Right(i)]$ $\Rightarrow$ Ok, nothing to do

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

  ① swap $(A[i], A[Right(i)])$

  ② go recursively to fix subtree rooted at $Right(i)$





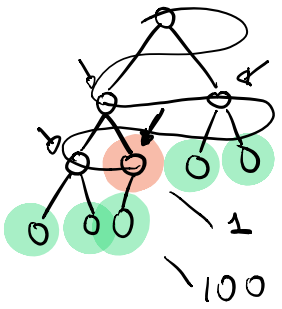
Complexity is $\Theta(h) = \Theta(\log n)$ time

```
Max-Heapify (A, i)
1   l = Leaf(i)                    the size of the heap
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)
```

Build a heap

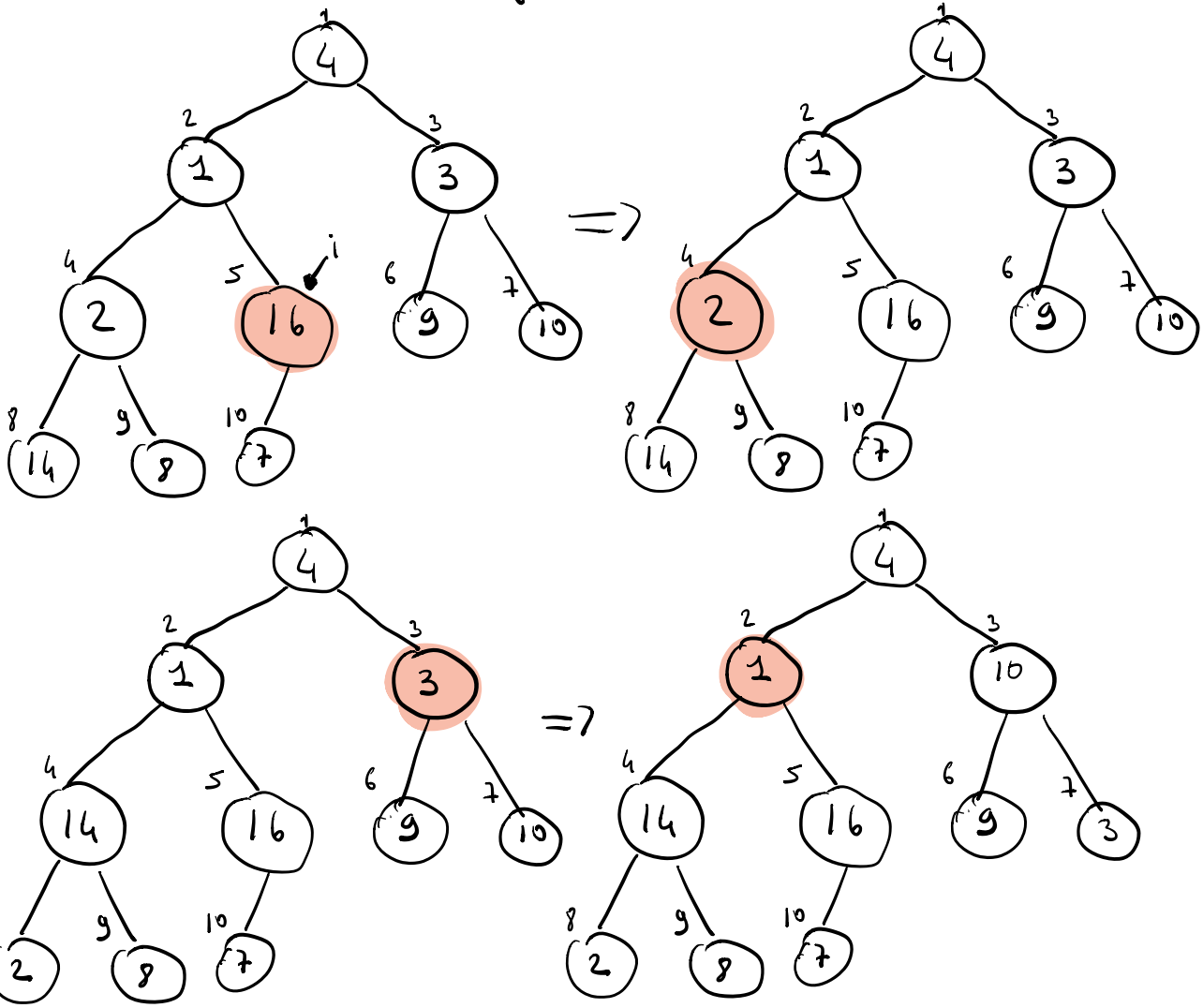
We start from an unordered array A and we would like to move keys to get a Max Heap

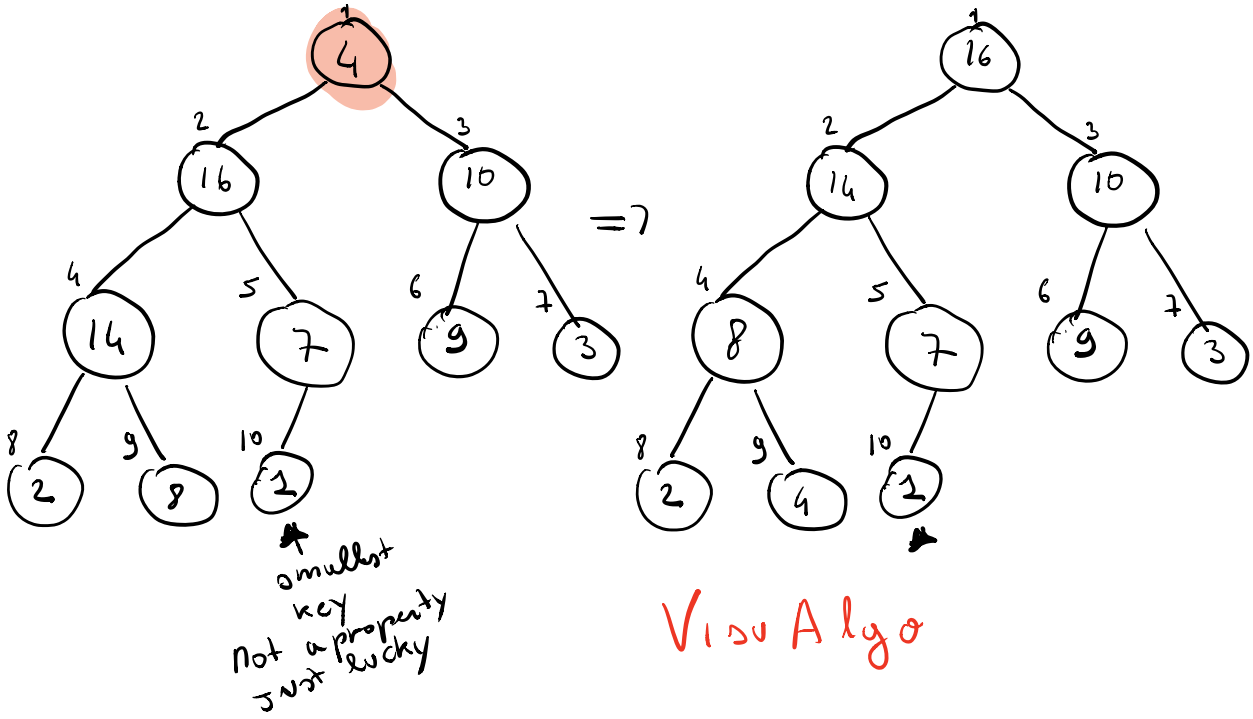
A = [ ]



```
Build-Max-Heap (A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ down to 1
3      Max-Heapify(A, i)
```

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

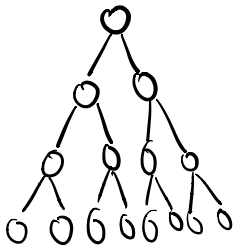




Complexity

- $\leq \frac{n}{2}$ calls to Max heapify()
- Each call costs $O(\log n)$ time
- overall cost is $O(n \log n)$ time

Better analysis



$\simeq \frac{n}{8}$ nodes cost 2

$\simeq \frac{n}{4}$ nodes cost 1

$\leq \frac{n}{2}$ nodes cost 0

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h-1}} \right\rceil \cdot O(h)$$

$$= O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = \Theta(n)$$

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

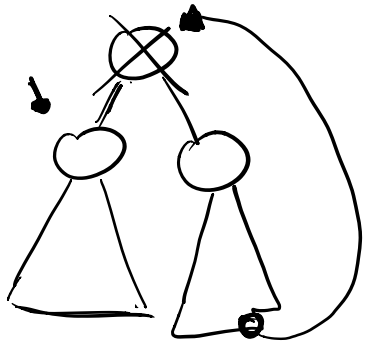
$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} = \frac{1/2}{\left(1 - \frac{1}{2}\right)^2} = 2$$

Extract - Max

Idea: • Replace A[1] with last element A[A.heap-size]
• Run Max-Heapify(A, 1)

Max-Extract(A)

```
1  if A.heap_size < 1
2      return "Error"
3  max = A[1]
4  A[1] = A[A.heap_size]
5  A.heap_size = A.heap_size - 1
6  Max_heapify(A, 1)
7  return max
```

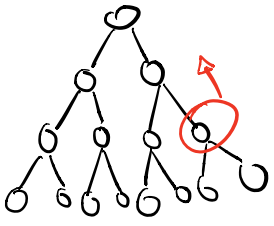


Complexity: $\Theta(\log n)$ time

Increase key

You want to increase an existing key

Idea:



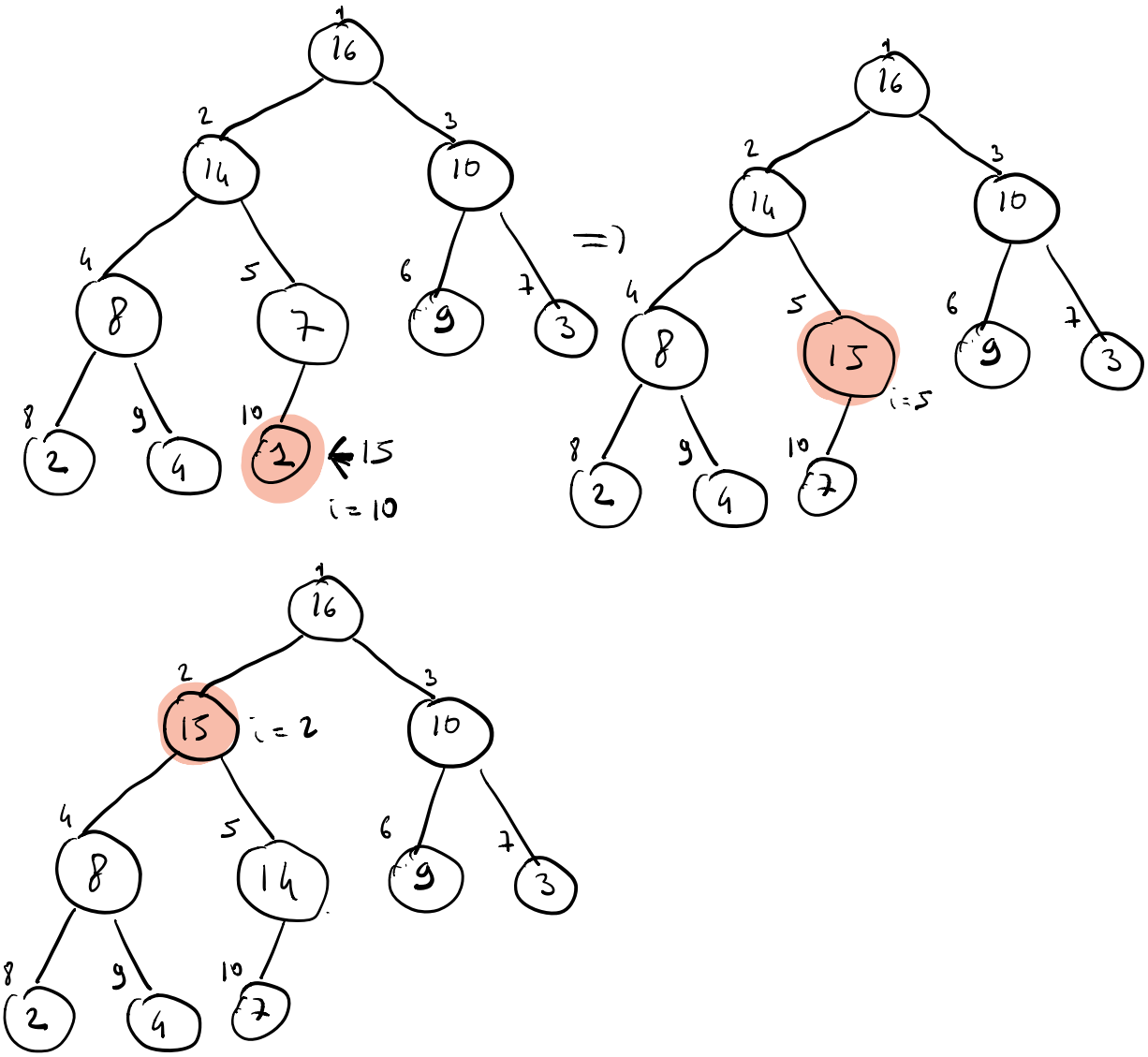
traverse the path from the node to the root until needed

```
Increase key (A, i, key)
1  if key < A[i]
2        return "Error: older key is bigger"
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

i=

Increase key (A, 10, key = 15)


1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
10
2
15
i = 10
=>
1
16
2
14
3
10
4
8
5
15
i = 5
6
9
7
3
8
2
9
4
10
7

1
16
2
15
i = 2
3
10
4
8
5
14
6
9
7
3
8
2
9
4
10
7

Inserting a key

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

Insert (A, 18)

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

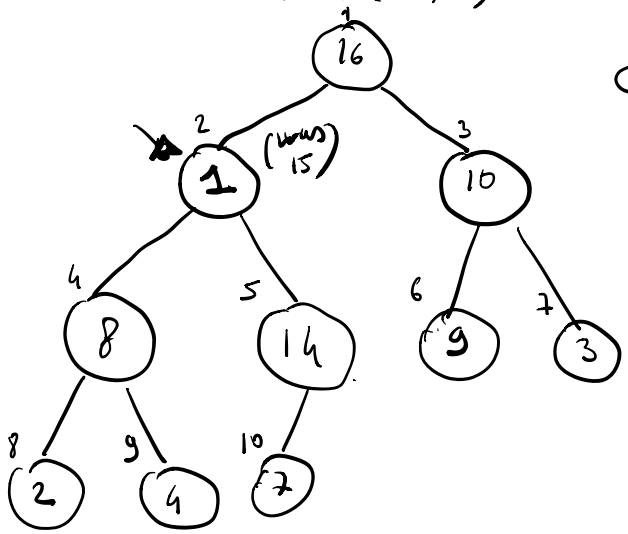
Exercises

(1) Decrease-key (A, i, key) in $\Theta(\log n)$ time

Delete (A, i) in $\Theta(\log n)$ time

Decrease_key(A, i, key)

1 if key > A[i]
2 return "Error"
3 A[i] = key
4 Max_Heapify(A, i)

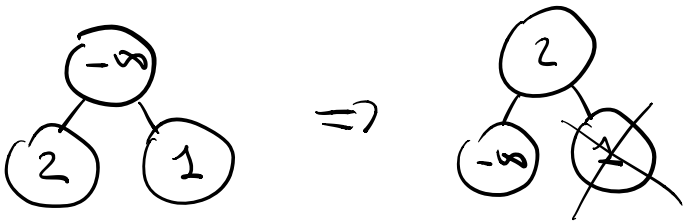


Complexity: $\Theta(\log n)$

Delete(A, i) INCORRECT

1 Decrease(A, i, $-\infty$)
2 A.heap_size = A.heap_size − 1

counterexample



Delete (A, i)

```
1  if A.heap_size < i
2        return "error"
3  A[i] = A[A.heap_size]
4  A.heap_size = A.heap_size - 1
5  Max-Heapify (A, i)
```

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

Exercise

Write the pseudocode of all the ops we have seen for a MIN-Heap