

# COMP20007 Workshop Week 8

## Preparation:

- use the powerpoint or PDF slides for note taking if you like, but
- ***have papers and pens (or drawing tools) ready***, and/or
- *ready to work on whiteboards*

1 Binary Heap: Operations, Heapsort, Questions 8.2, 8.3

2 Sorting Algorithms - quicksort: Question 8.1  
Quickselect (Question 8.4)

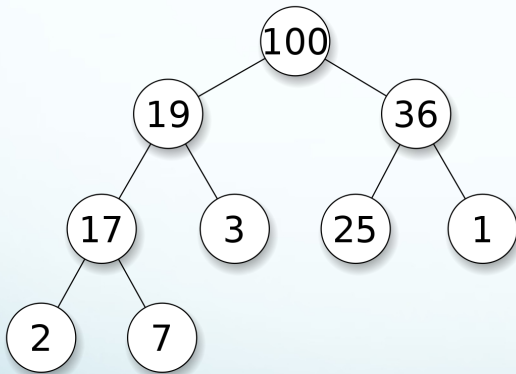
LAB Lab: Sorting algs, follow the given instructions

# A Priority Queue: Binary Heap = ?

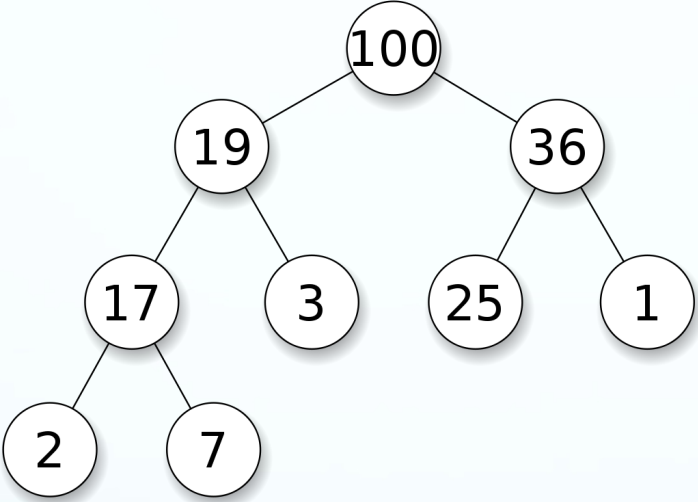
Binary Heap as a concrete data type (implementation) for PQ.

min heap, max heap = ?

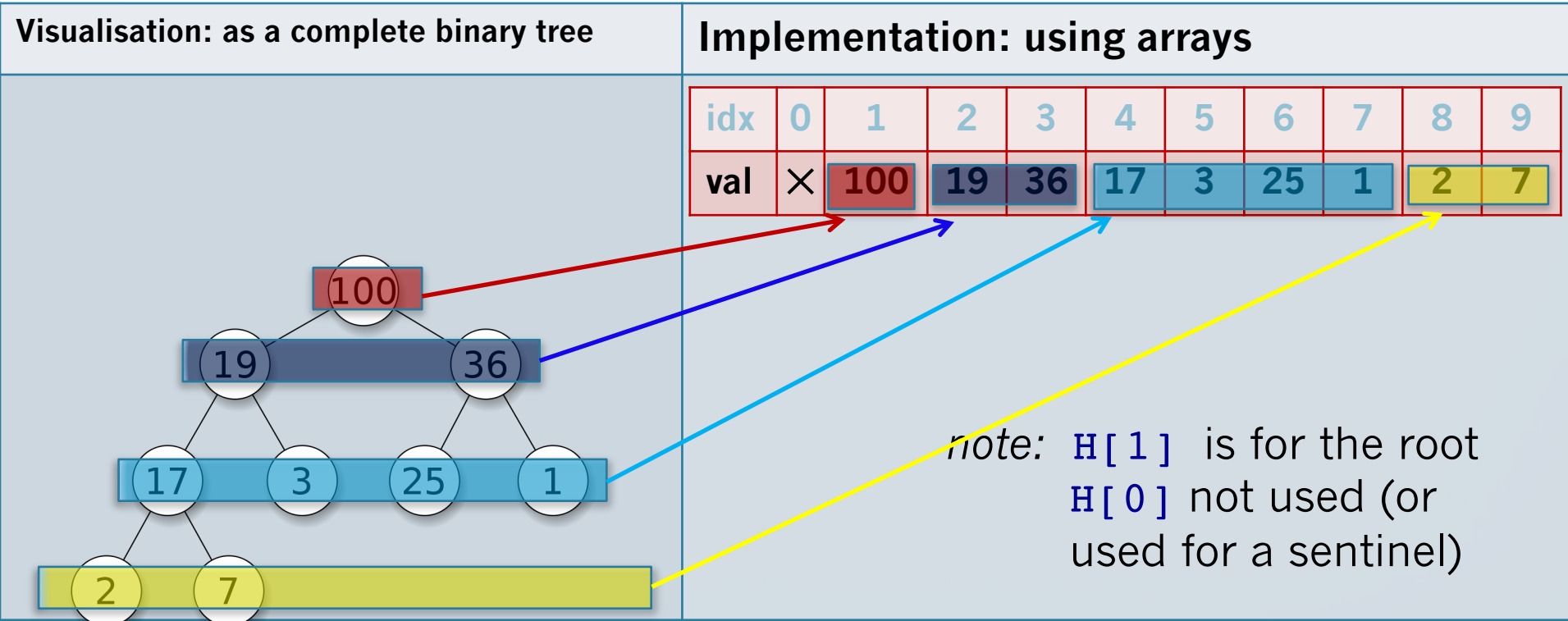
What is a, say, max heap?  
How is it implemented?



# Binary Heap: conceptually, is a binary tree

Example	Conditions
 <pre>graph TD; 100((100)) --- 19((19)); 100 --- 36((36)); 19 --- 17((17)); 19 --- 3((3)); 17 --- 2((2)); 17 --- 7((7)); 36 --- 25((25)); 36 --- 1((1));</pre>	<ol style="list-style-type: none"><li data-bbox="1020 339 1870 646">1. The tree is <i>complete</i>:<ul style="list-style-type: none"><li data-bbox="1116 404 1792 518">• all levels, except for the last, are full</li><li data-bbox="1116 532 1870 646">• the last level is filled from left to right</li></ul></li><li data-bbox="1020 782 1870 1089">2. The <i>heap property</i>: each node has a higher priority (here, is larger) than any of its descendants (or equivalently, just its children).</li></ol>

# Binary Heap: is implemented as an array!



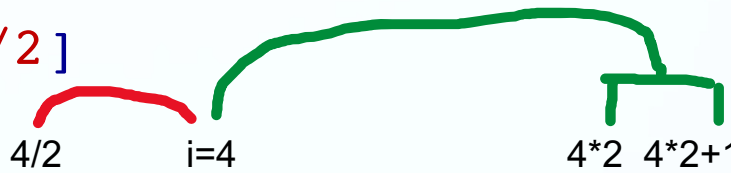
Heap is  $H[1..n]$

- level  $i$  occupies  $2^i$  cells in array  $H[1..n]$   
(except for the last level)
- if root is level 1, then level  $i$  starts from  $H[i]$

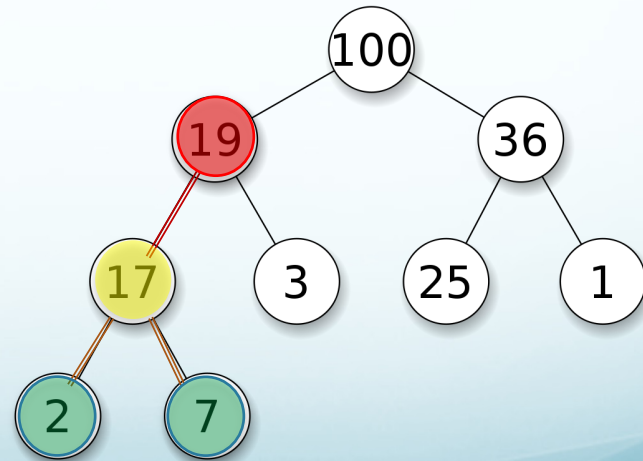
# Binary Heap: parent and children of a node

- left child of  $H[i]$  is  $H[2*i]$
- right child of  $H[i]$  is  $H[2*i+1]$

parent of  $H[i]$  is  $H[i/2]$

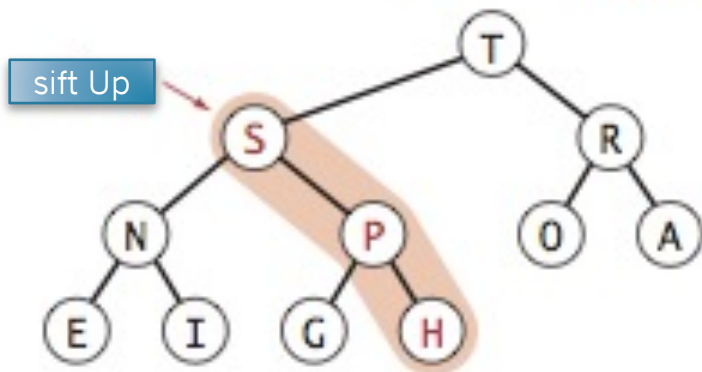
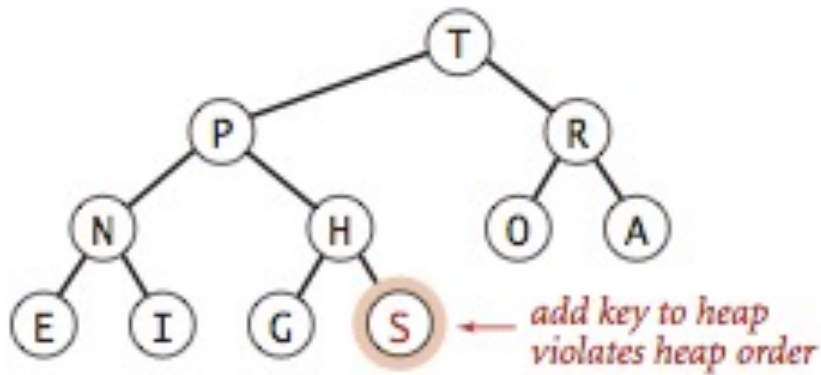
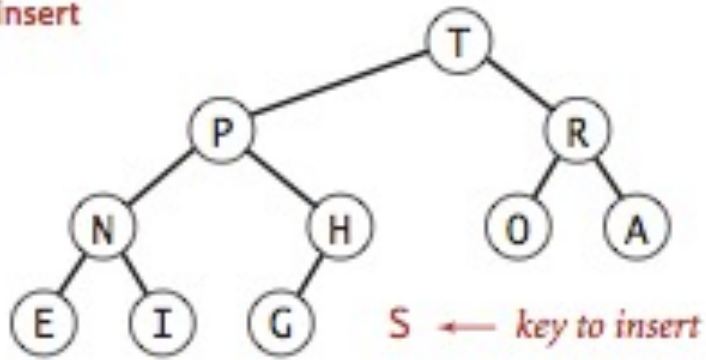


idx	0	1	2	3	4	5	6	7	8	9
val	×	100	19	36	17	3	25	1	2	7



# inject = enPQ = Insert a new elem into a heap

insert



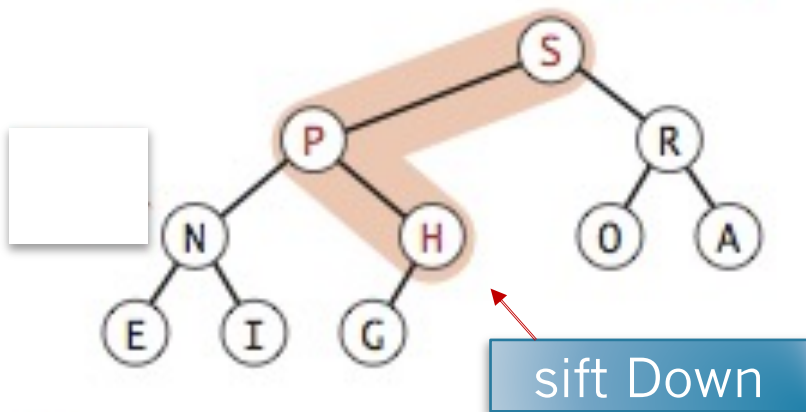
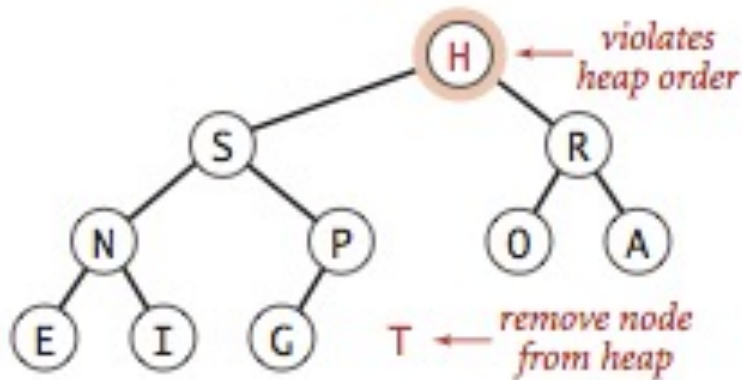
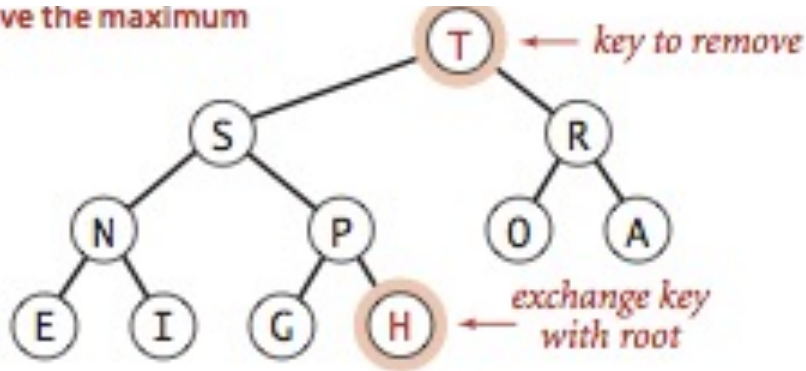
Notes:

Sift Up

while (has parent and parent has lower priority): swap up with the parent

# eject = delete (and returns) the heaviest

remove the maximum



Notes:

Sift Down:

while (has children and the heavier child has higher priority): swap down with the *heavier* child



## Heapify: Turning an array $H[1..n]$ into a heap

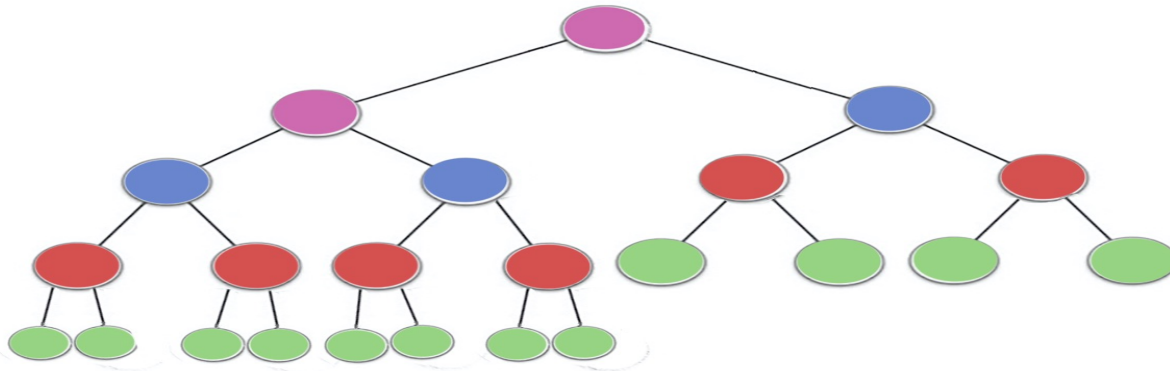
```
function Heapify(H[1..n])
  for i ← n/2 downto 1 do
    downheap(H, i)
```

=  **$\Theta(n)$**  (see lectures and/or ask Google for a proof)

The operation is aka. **Heapify**/Makeheap/ Bottom-Up Heap Construction

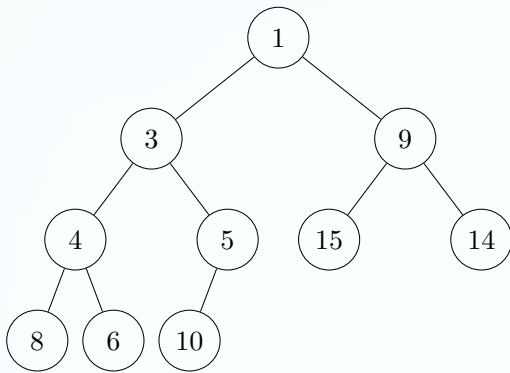
Example: build maxheap for keys E X A M P

Notes:





## Q 8.2



- a) Show how this heap would be stored in an array as discussed in lectures (root is at index 1; node at index  $i$  has children at indices  $2i$  and  $2i+1$ )
- b) Run the `RemoveRootFromHeap` (`eject`) algorithm from lectures on this heap by hand (i.e., swap the root and the “last” element and remove it. To maintain the heap property we then `SiftDown` from the root).
- c) Run the `InsertIntoHeap` (`inject`) algorithm and insert the value 2 into the heap

Your answers:

- a) array is: [ ??? ]
- b) Run the `RemoveRootFromHeap`:

- c) Run the `InsertIntoHeap(2)`

## Q 8.3 [opt]: k-smallest using min-heap

- The *k-th smallest* problem:
  - Given an array  $A[ ]$  of  $n$  elements, and an integer  $k$
  - Find the  $k$ -th smallest value (suppose that  $k$  is zero-origin, that is,  $k$  can be any of  $0, 1, 2, \dots, n-1$ )
- How can we use a min-heap data structure to solve the  $k$ th-smallest element problem? What is the time-complexity of this algorithm?

**Your answer:**

Algorithm	Complexity
<pre>function HeapkthSmallest(A[0..n-1],k)</pre>	

# Basic Sorting Algorithms

Know how to run by hand the following algorithms:

- Selection Sort
- Insertion Sort
- Quick Sort with Lomuto's Partitioning
- Quick Sort with Hoare's Partitioning
- Merge Sort
- Heap Sort

Run example with keys:

**E X A M P**

**LAB:** follow instructions in workshop sheet

## Question 8.1

We saw the following sorting algorithms,

- (a) Selection Sort
- (b) Insertion Sort
- (c) Quicksort (with Lomuto partitioning)

For each algorithm:

- (i) Run the algorithm on the array:

[A N A L Y S I S]

- (ii) time complexity of the algorithm=?
- (iii) Is the sorting algorithm stable?
- (iv) Does the algorithm sort in-place?
- (v) Is the algorithm input sensitive?
- (vi) What is the strongest point of the algorithm (when should it be used)?

If you get time, try to answer these questions for

- (d) Quicksort (with Hoare partitioning), and
- (e) Merge Sort
- (f) Heap Sort

# Quicksort for $A[l..r]$

```
function quicksort( $A[l..r]$ )
```

```
  if  $l \geq r$  then return  
   $s \leftarrow$  do partitioning, ie:           # Lomuto or Hoare or ...  
    pivot =  $A[l]$   
    rearrange  $A[]$  into  $A[l..s-1]$   $A[s]$   $A[s+1..r]$  so that  
     $A[s] =$  pivot  
    all elements in  $A[l..s-1]$  are  $\leq$  pivot  
    all elements in  $A[s+1..r]$  are  $\geq$  pivot  
  quicksort( $A[l..s-1]$ )  
  quicksort( $A[s+1..r]$ )
```

## Notes:

- We can choose any element in  $A[l..r]$  as **pivot**: just swap it with  $A[l]$  at the start of partitioning
- $A[l..s-1]$  or  $A[s+1..r]$  could be empty
- Complexity depends on the relative lengths of  $A[l..s-1]$  and  $A[s+1..r]$

function LOMUTOPARTITION(A[l..r])

# loop init:

$P \leftarrow A[l]$   
 $s \leftarrow l, i \leftarrow l+1$

# loop invariant:

#  $A[l+1..s] < P$      $A[s+1..i-1] \geq P$

#  $A[i..r]$  not yet examined

while  $i \leq r$  do

  # move  $i++$  until  $A[i] < P$

  if  $A[i] < P$  then

    #extend the yellow area

$s \leftarrow s + 1$

    SWAP( $A[s], A[i]$ )

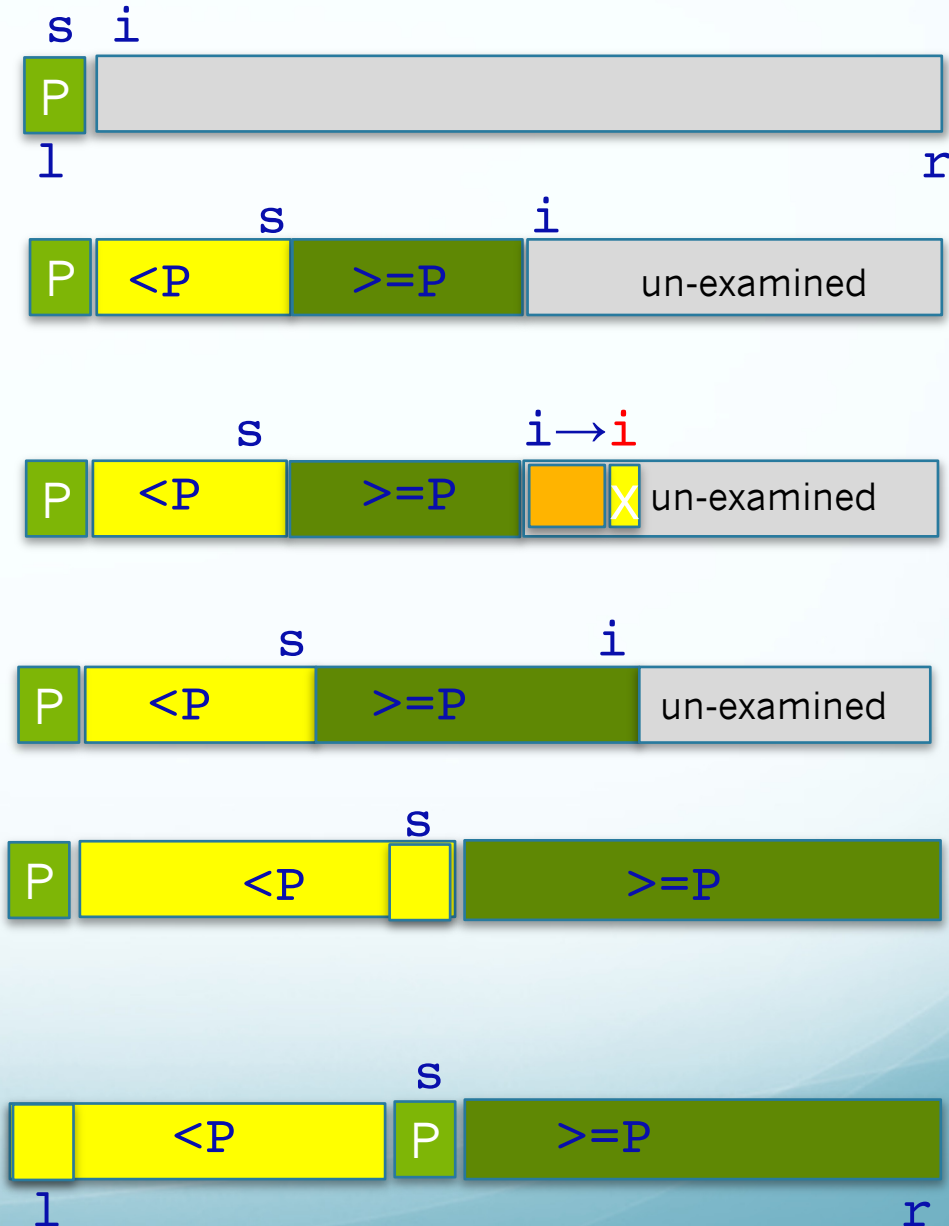
$i \leftarrow i+1$

# at loop exit ( $i=r+1$ ) ----->

#  $A[s]$  is the last yellow (OR  $s=l$ )

SWAP( $A[l], A[s]$ )

return  $s$



# Example: Run quicksort with Lomuto's for [ E X A M P ]

```
# loop init:
P ← A[l]
s ← l, i ← l+1
# loop invariant:
# A[l+1..s] < P    A[s+1..i-1] ≥ P
# A[i..r] not yet examined

while i ≤ r do
    # move i++ until A[i] < P
    if A[i] < P then

        #extend the yellow area
        s ← s + 1
        SWAP(A[s], A[i])

    i ← i+1
# at loop exit (i=r+1) ----->
# A[s] is the last yellow (OR s=l)

SWAP(A[l], A[s])
return s
```

Partitioning [ E X A M P ]

E<sub>s</sub> X<sub>i</sub> A M P  
E<sub>s</sub> X A<sub>i</sub> M P  
E A<sub>s</sub> X<sub>i</sub> M P  
E A<sub>s</sub> X M P<sub>i</sub>

[A] E<sub>s</sub> [X M P]

Partitioning [ X M P ]

X<sub>s</sub> M<sub>i</sub> P  
X M<sub>si</sub> P  
X M<sub>s</sub> P<sub>i</sub>  
X M P<sub>si</sub>  
X M P<sub>s</sub> i

[P M] X<sub>s</sub>

Partitioning [ P M ]

P<sub>s</sub> M<sub>i</sub>  
P M<sub>si</sub>  
P M<sub>s</sub> i

[M] P<sub>s</sub>

Final sorted: [ A E M P X ]

To simplify,  
we will write  
out the  
sequence  
only: at the  
beginning  
and end of  
the loop,  
and after a  
swap.

The  
algorithm is  
quite slow  
on small  
arrays!

# qsort with Lomuto's Partitioning

Q8.1c): Run quicksort with Lomuto's for [ A N A L Y S I S ]. Here we rewrite the sequence after each swap and at the start/end of the loop.

1. Partitioning of  
A N A L Y S I S

A<sub>s</sub> N<sub>i</sub> A L Y S I S  
???

2. Partitioning of  
???

???

3. Partitioning of  
???

???

4. Partitioning of  
???

???

5. Partitioning of  
???

???

Final sorted sequence  
???



```
function HOAREPARTITION(A[l..r])
```

# loop init:

$P \leftarrow A[l]$

$i \leftarrow l; j \leftarrow r + 1$

# loop invariant:

#  $A[l+1..i] \leq P$      $A[j..r] \geq P$

#  $A[i+1..j-1]$  not yet examined

repeat

# move  $i$  forward until  $A[i] \geq P$

repeat  $i \leftarrow i+1$  until  $A[i] \geq P$

# same as do  $i \leftarrow i+1$  while  $A[i] < P$

# move  $j$  backward until  $A[j] \leq P$

repeat  $j \leftarrow j-1$  until  $A[j] \leq P$

# extend yellow and green area

# at the same time by swapping ---->

if ( $i < j$ ) then SWAP( $A[i], A[j]$ )

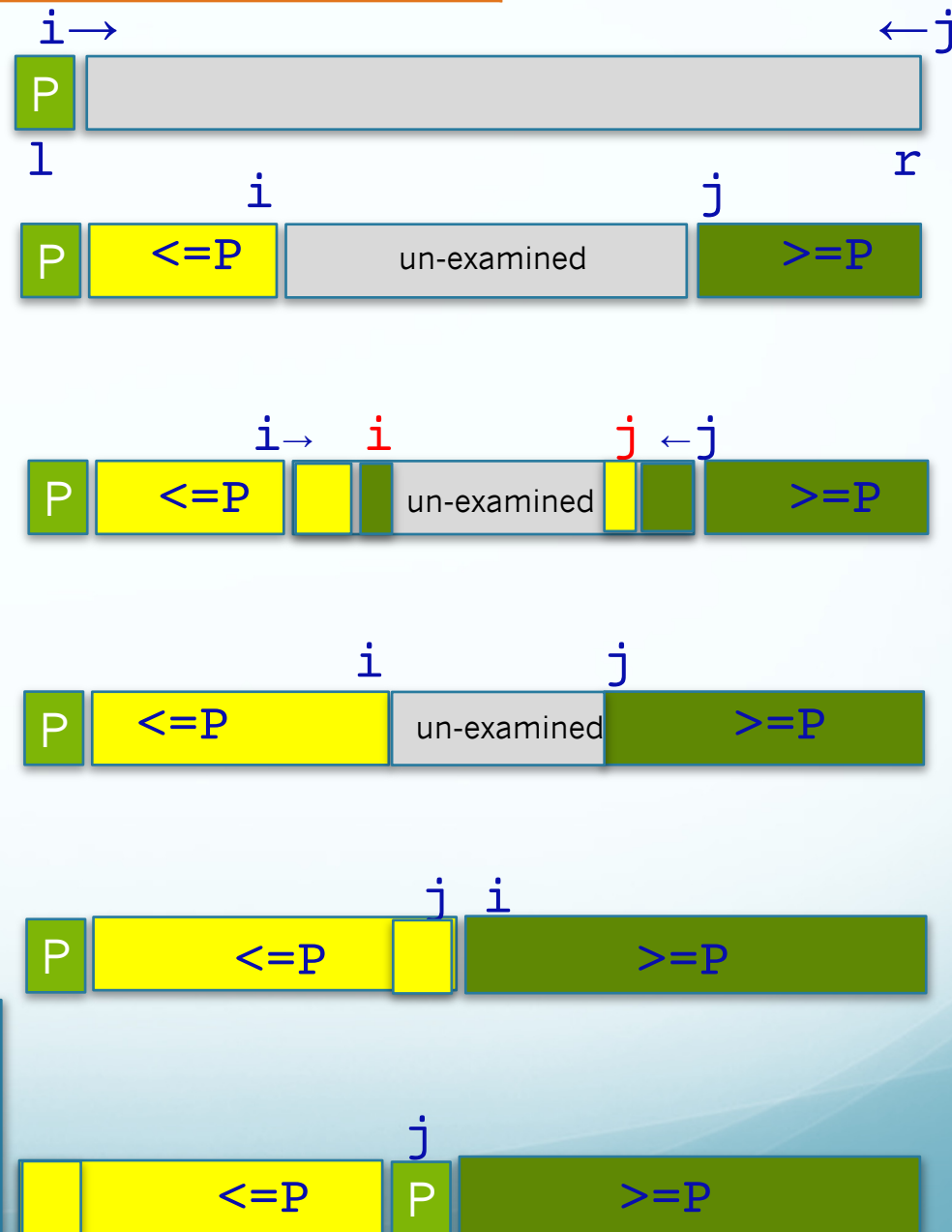
until  $i \geq j$

# at loop's exit:

SWAP( $A[l], A[j]$ )

return  $j$

Run example  
with  
EXAMPLE



# qsort with Hoare's Partitioning

**Q8.1d)** Run quicksort with Hoare's for [ A N A L Y S I S ]. Here we rewrite the sequence after each swap and at the start/end of the loop.

1. Partitioning of  
A N A L Y S I S

$A_i$  N A L Y S I S  $j$   
???

2. Partitioning of  
???

???

3. Partitioning of  
???

???

4. Partitioning of  
???

???

Final sorted sequence  
???

## Also

Make sure you can run (by hand) Merge Sort and HeapSort for

- [E X A M P]
- [A N A L Y S I S]

And, review the lectures for the remaining questions of problem 1. For each sorting algorithm, think:

- which is the best situations when we want to employ that algorithm?
- in which situations when we definitely don't want that algorithm?

## Q 7.4 [opt]

- a) Design an algorithm Quickselect based on Quicksort which uses the **Partition** algorithm to find the **k**-th smallest element in an array **A**.
- b) Show how you can run your algorithm to find the **k**-th smallest element where **k** = 4 and **A** = [9, 3, 2, 15, 10, 29, 7].
- c) What is the best-case time-complexity of your algorithm? What type of input will give this time-complexity?
- d) What is the worst-case time-complexity of your algorithm? What type of input will give this time-complexity?
- e) What is the expected-case (i.e., average) time-complexity of your algorithm?
- f) When would we use this algorithm instead of the heap based algorithm from Question 3

# partitionning & qselect

```
partition( A[lo..hi]):  
    ...  
    return m
```

```
11 function qselect( A[lo..hi], k)  
12     m= partition(A[lo..hi])  
13  
14  
15  
16  
17
```

```
21 function ksmallest(A[0..n-1], k)  
22     if (k>=0 && k<n)  
23         return qselect(A[0..n-1], k)
```

# Check: partitionning & qselect

```
partition( A[lo..hi]):  
    ...  
    return m
```

```
11 function qselect( A[lo..hi], k)  
12     m= partition(A[lo..hi])  
13     if (k==m) then return A[m]  
14     if (k<m) then  
15         qselect(A[lo..m-1], k)  
16     else  
17         qselect(A[m+1..hi], k)
```

```
21 function ksmallest(A[0..n-1], k)  
22     if (k>=0 && k<n)  
23         return qselect(A[0..n-1], k)
```

# Problem 4

- a) Design an algorithm based on Quicksort which uses the Partition algorithm to find the  $k$ -th smallest element in an array  $A$ .
- b) Show how you can run your algorithm to find the  $k$ -th smallest element where  $k = 4$  and  $A = \{9, 3, 2, 15, 10, 29, 7\}$ .
- c) What is the best-case time-complexity of your algorithm? What type of input will give this time-complexity?
- d) What is the worst-case time-complexity of your algorithm? What type of input will give this time-complexity?
- e) What is the expected-case (i.e., average) time-complexity of your algorithm?



# LAB