Algorithms and Datastructures

Lecture 5

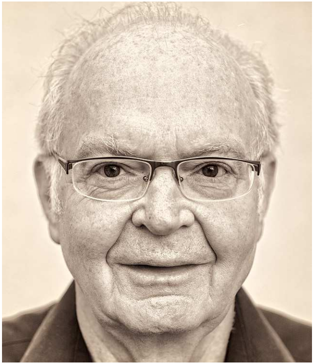
Manfred Jaeger

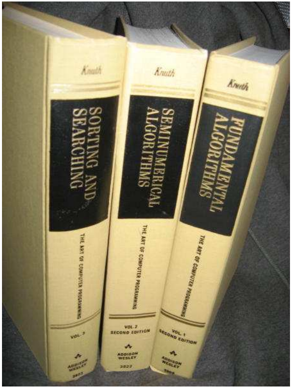
AALBORG UNIVERSITET

Sorting

Donald E. Knuth *1938

Author of *The Art of Computer Programming*, and co-founder of the science of algorithms.

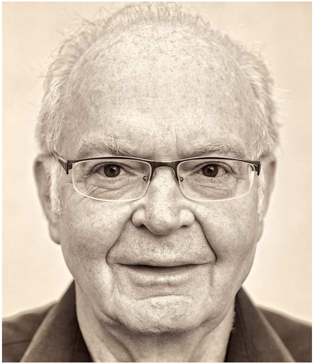


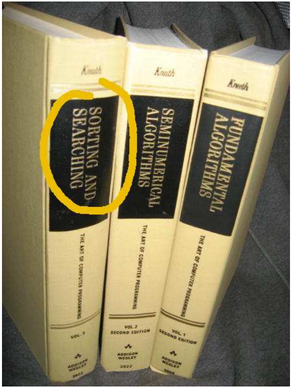


Also: inventor of TeX, recipient of the Turing Award (1974).

Donald E. Knuth *1938

Author of *The Art of Computer Programming*, and co-founder of the science of algorithms.





Also: inventor of TeX, recipient of the Turing Award (1974).

Algorithm	Worst case	Average case	In place
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	no
Bubble sort	$\Theta(n^2)$		yes

In place: the algorithm only requires a constant amount of memory besides the array being sorted (independent of size of array).

Merge sort needs $\Theta(n)$ extra space to store the sorted sub-arrays before merging.

Ordered Sets

The only property of integers we use in sorting is the ordering relation:

$$i < j\,? \quad i > j\,? \quad i = j\,?$$

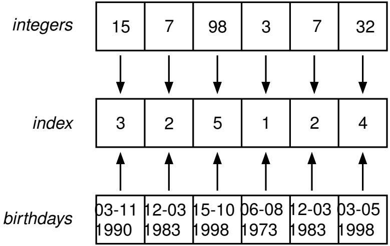
☞ Our algorithms work for arrays that contain any kind of *keys* with values in an *ordered set*.

Order Index

For key k in array A let

$$index(k) = |\{k' \in A | k' < k\}|$$

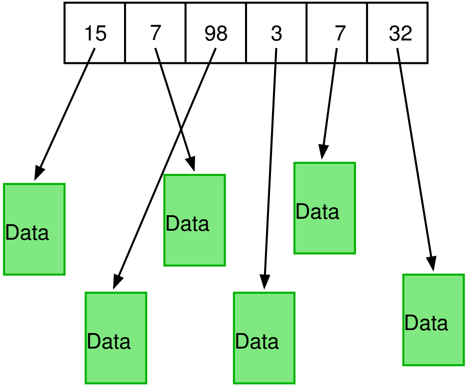
(number of elements in A that are smaller than k)



☞ The operations and runtime of the algorithms only depend on the order indices of the array elements

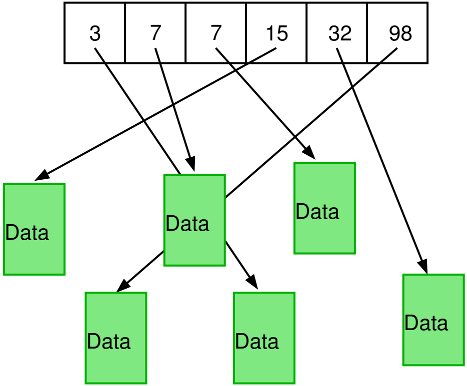
Usually, we sort **keys** that are associated with larger amounts of **(satellite) data**.

Need to sort keys, and references to the associated data:



Usually, we sort **keys** that are associated with larger amounts of **(satellite) data**.

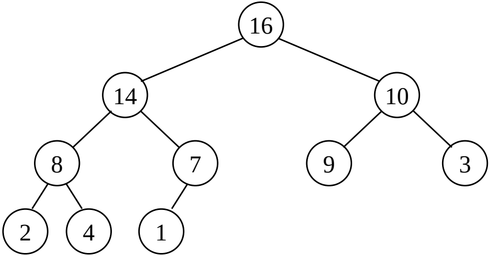
Need to sort keys, and references to the associated data:



Heapsort

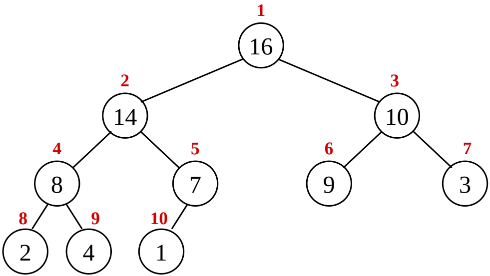
Binary tree with values at the nodes, and

- all levels except last one are completely filled
- the **max-heap property**: the value at a node is greater equal the values of its (at most) two children.



Binary tree with values at the nodes, and

- all levels except last one are completely filled
- the **max-heap property**: the value at a node is greater equal the values of its (at most) two children.



The nodes indexed in top-to-bottom, left-to-right order

Using the node-indexing, the data can be stored in an array *A*:

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
16	14	10	8	7	9	3	2	4	1	...	...	...	...	...

$A.heap\!-\!size$ $\qquad$ $A.length$

- The indexing must start with 1!
- The array may be longer than the actual heap content. The attribute *A.heap-size* denotes the last position in the array that contains a heap element.

Using the node-indexing, the data can be stored in an array *A*:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	14	10	8	7	9	3	2	4	1	...	...	...	...	...

$A.heap\!-\!size$ (index 10) $\qquad$ $A.length$ (index 15)

- The indexing must start with 1!
- The array may be longer than the actual heap content. The attribute *A.heap-size* denotes the last position in the array that contains a heap element.

Given a heap element with index i, the indices of its parent and left and right children can be computed as follows:

$$
\begin{array}{ll}
\mathit{Parent}(i) = \lfloor i/2 \rfloor & (\text{if } i \neq 1) \\
\mathit{Left}(i) = 2i & (\text{if } 2i \leq \mathit{A.heap\text{-}size})) \\
\mathit{Right}(i) = 2i + 1 & (\text{if } 2i + 1 \leq \mathit{A.heap\text{-}size}))
\end{array}
$$

A **Max-Heap** is an (integer) array with

- an additional attribute *A.heap-size* with $0 \leq A.heap\text{-}size \leq A.length$
- the **max-heap property**: $A[Parent(i)] \geq A[i]$ for all $2 \leq i \leq A.heap\text{-}size$.

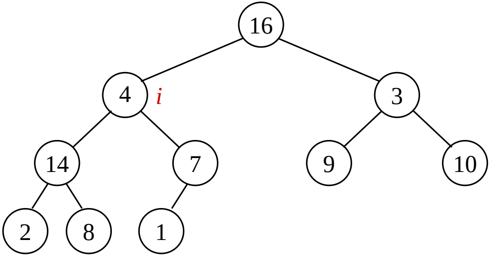
Similarly: **Min-Heap**

Why arrays?

☞ the arithmetic operations $i \mapsto 2i$, $i \mapsto 2i + 1$, $i \mapsto \lfloor i/2 \rfloor$ are more efficient than following pointers in a tree representation using linked nodes.

Procedure MAX-HEAPIFY(A, i), where

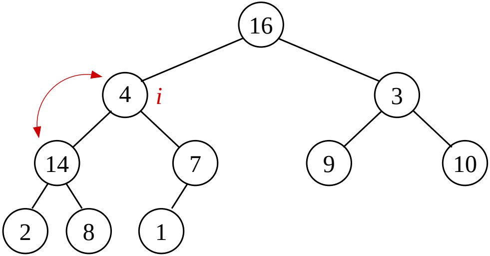
- *A* is an array with attribute *A.heap-size*
- $1 \leq i \leq$ *A.heap-size*, and the two sub-trees rooted at *i* satisfy the max-heap property



If max-heap property violated at node *i*:

Procedure MAX-HEAPIFY(A, i), where

- A is an array with attribute *A.heap-size*
- $1 \leq i \leq$ *A.heap-size*, and the two sub-trees rooted at i satisfy the max-heap property

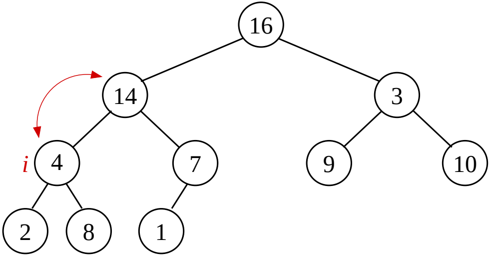


If max-heap property violated at node i:

- exchange the value of i with the larger value of its two children

Procedure MAX-HEAPIFY(A, i), where

- A is an array with attribute *A.heap-size*
- $1 \leq i \leq$ *A.heap-size*, and the two sub-trees rooted at i satisfy the max-heap property

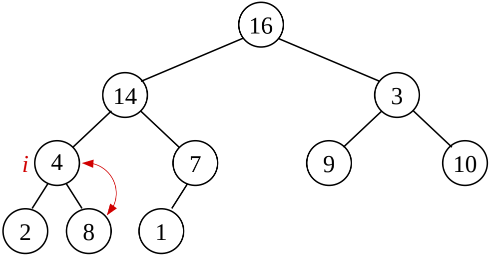


If max-heap property violated at node i:

- exchange the value of i with the larger value of its two children
- call MAX-HEAPIFY for the child with which value was exchanged

Procedure MAX-HEAPIFY(A, i), where

- A is an array with attribute *A.heap-size*
- $1 \leq i \leq$ *A.heap-size*, and the two sub-trees rooted at i satisfy the max-heap property

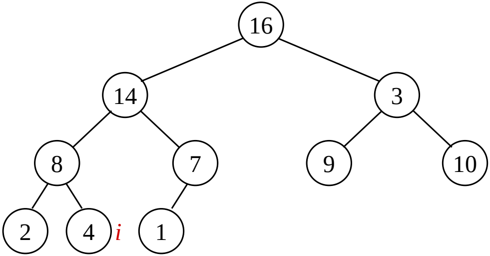


If max-heap property violated at node i:

- exchange the value of i with the larger value of its two children
- call MAX-HEAPIFY for the child with which value was exchanged

Procedure MAX-HEAPIFY(A, i), where

- *A* is an array with attribute *A.heap-size*
- $1 \leq i \leq$ *A.heap-size*, and the two sub-trees rooted at *i* satisfy the max-heap property

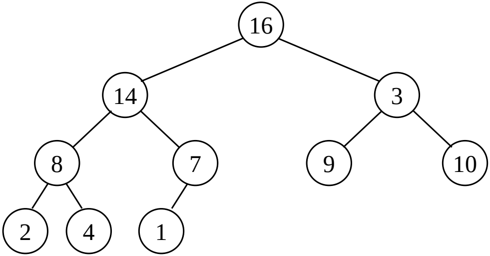


If max-heap property violated at node *i*:

- exchange the value of *i* with the larger value of its two children
- call MAX-HEAPIFY for the child with which value was exchanged

Procedure MAX-HEAPIFY(A, i), where

- A is an array with attribute *A.heap-size*
- $1 \leq i \leq A.heap\text{-}size$, and the two sub-trees rooted at i satisfy the max-heap property



If max-heap property violated at node i:

- exchange the value of i with the larger value of its two children
- call MAX-HEAPIFY for the child with which value was exchanged

Correctness

After execution of MAX-HEAPIFY(A, i) the subtree rooted at i satisfies the max-heap property.

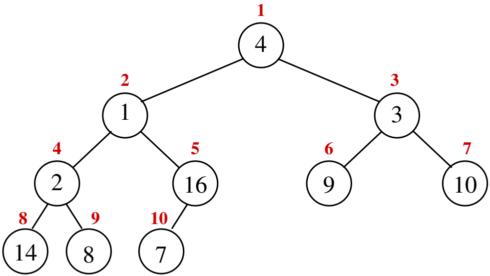
Complexity

If the subtree rooted at i contains n nodes, then the time complexity is given by the recurrence:

$$T(n) \leq T(2n/3) + \Theta(1),$$

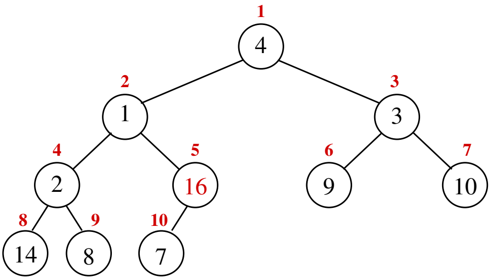
and so $T(n) = O(\lg n)$ (Master Theorem, case 2).

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



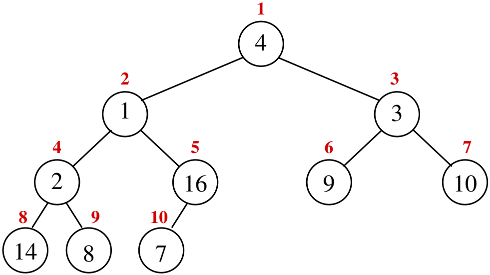
Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



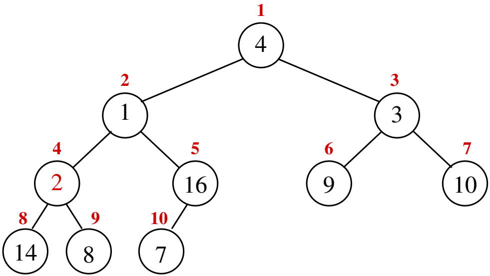
Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



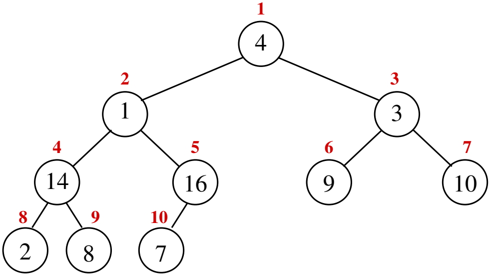
Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



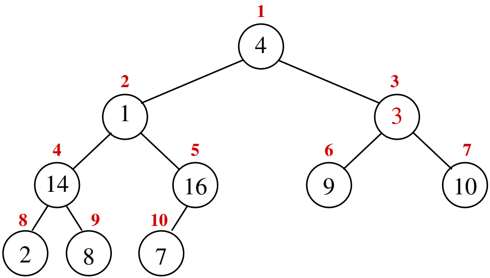
Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



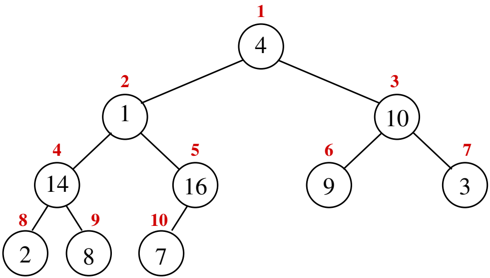
Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



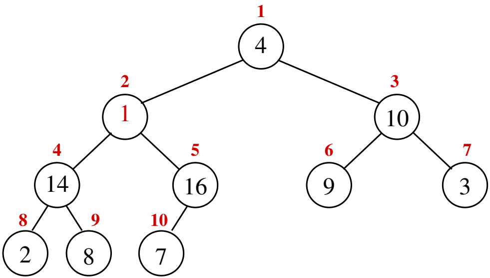
Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



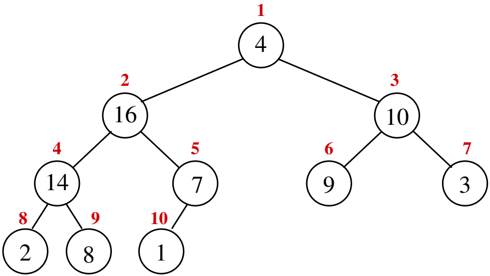
Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



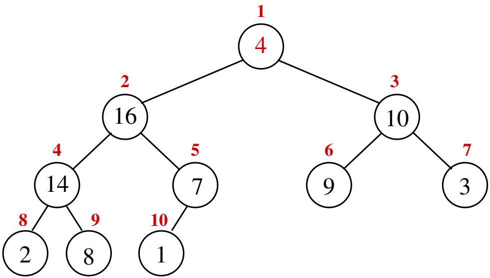
Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



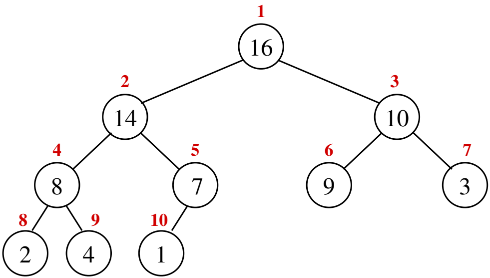
Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

BUILD-MAX-HEAP(A) transforms the array A into a max-heap.



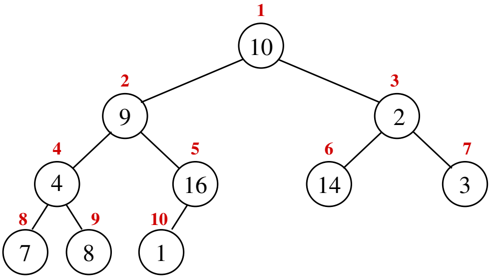
Start at the highest index that is not a leaf, and call MAX-HEAPIFY for all positions in descending order.

Correctness: key observation: when MAX-HEAPIFY(i) is called, then all children of i are roots of max-heaps.

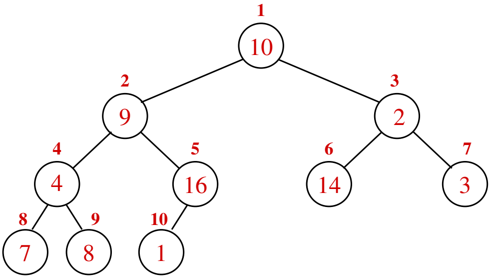
Complexity:

For an array of length n:

- Rough analysis: $O(n)$ calls of $O(\lg n)$ MAX-HEAPIFY procedure $\rightsquigarrow$ $O(n \lg n)$ upper bound
- Better: use that most of the MAX-HEAPIFY calls are for small sub-trees (size much smaller than n). Gives $O(n)$ bound.

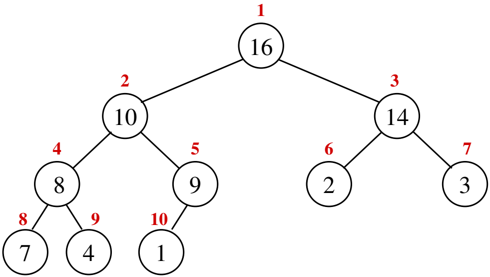


A an array:



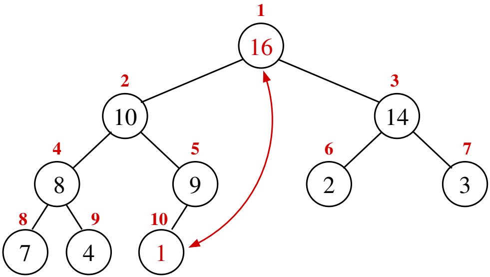
A an array:

- 1. Turn *A* into a heap with BUILD-MAX-HEAP



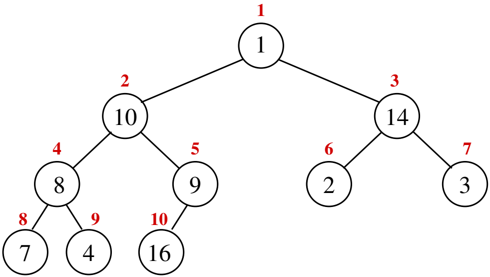
A an array:

- 1. Turn *A* into a heap with BUILD-MAX-HEAP



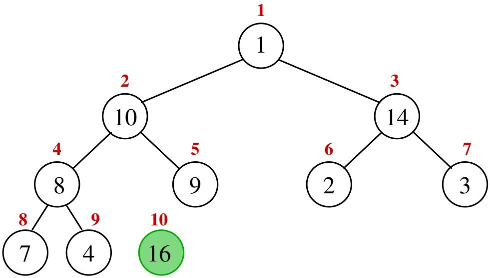
A an array:

- 1. Turn *A* into a heap with BUILD-MAX-HEAP
- 2. Exchange root value with last leaf value



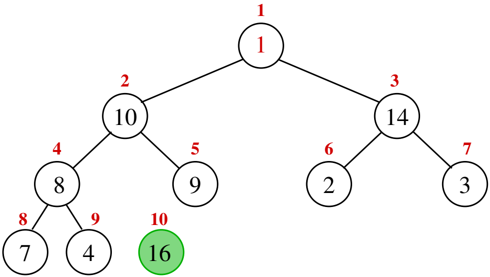
A an array:

- 1. Turn *A* into a heap with BUILD-MAX-HEAP
- 2. Exchange root value with last leaf value



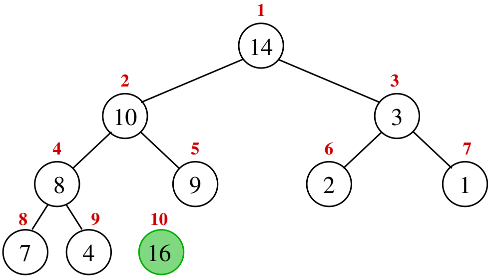
A an array:

- 1. Turn *A* into a heap with BUILD-MAX-HEAP
- 2. Exchange root value with last leaf value
- 3. Detach last leaf from heap by setting *A.heap-size*=*A.heap-size*-1



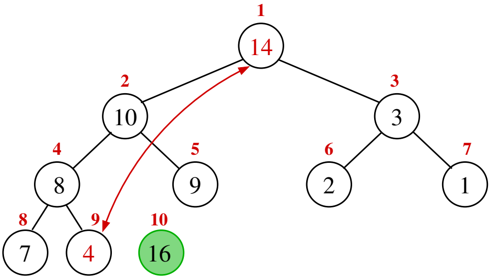
A an array:

- 1. Turn *A* into a heap with BUILD-MAX-HEAP
- 2. Exchange root value with last leaf value
- 3. Detach last leaf from heap by setting *A.heap-size*=*A.heap-size*-1
- 4. Call MAX-HEAPIFY$(A, 1)$



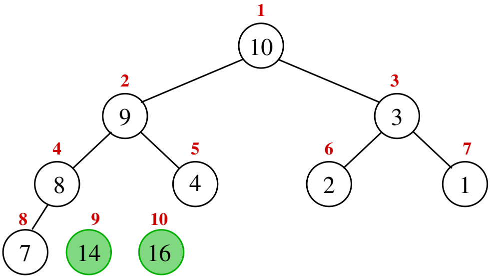
A an array:

- 1. Turn *A* into a heap with BUILD-MAX-HEAP
- 2. Exchange root value with last leaf value
- 3. Detach last leaf from heap by setting *A.heap-size*=*A.heap-size*-1
- 4. Call MAX-HEAPIFY$(A, 1)$



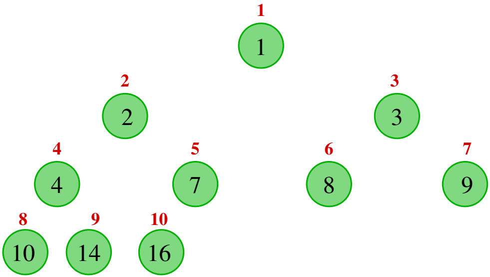
A an array:

- 1. Turn *A* into a heap with BUILD-MAX-HEAP
- 2. Exchange root value with last leaf value
- 3. Detach last leaf from heap by setting *A.heap-size*=*A.heap-size*-1
- 4. Call MAX-HEAPIFY$(A, 1)$
- Iterate 2.-4. ...



A an array:

- 1. Turn *A* into a heap with BUILD-MAX-HEAP
- 2. Exchange root value with last leaf value
- 3. Detach last leaf from heap by setting *A.heap-size*=*A.heap-size*-1
- 4. Call MAX-HEAPIFY(A, 1)
- Iterate 2.-4. ...



A an array:

- 1. Turn *A* into a heap with BUILD-MAX-HEAP
- 2. Exchange root value with last leaf value
- 3. Detach last leaf from heap by setting *A.heap-size*=*A.heap-size*-1
- 4. Call MAX-HEAPIFY(A, 1)
- Iterate 2.-4. ...

Correctness: key observation: after swapping the values in the root and the final leaf, and detaching the final leaf, the max-heap property can only be violated at the root.

Complexity:

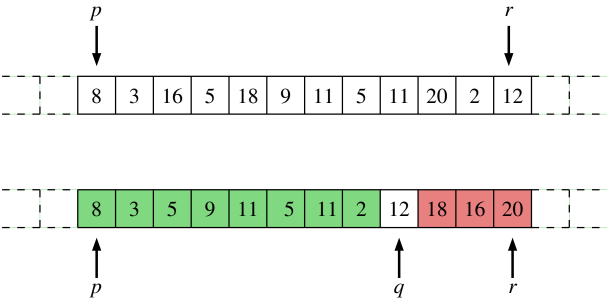
- Initial BUILD-MAX-HEAP: $O(n)$
- $n - 1$ calls of MAX-HEAPIFY: $O(n \lg n)$ (in this case: most of the calls are for still large sub-trees).
- In total: $O(n \lg n)$.

Algorithm	Worst case	Average case	In place
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	no
Bubble sort	$\Theta(n^2)$		yes
Heap sort	$O(n \lg n)$		yes

For QUICKSORT the correct row is ...

Row	Worst case	Average case	In place
A	$\Theta(n \lg n)$	$\Theta(n \lg n)$	yes
B	$\Theta(n \lg n)$	$\Theta(n \lg n)$	no
C	$\Theta(n^2)$	$\Theta(n \lg n)$	yes
D	$\Theta(n^2)$	$\Theta(n \lg n)$	no
E	$\Theta(n \lg n)$	$\Theta(n)$	no

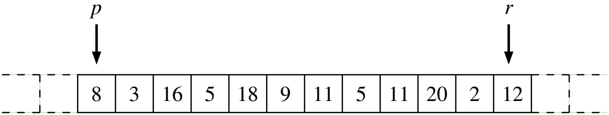
Quicksort



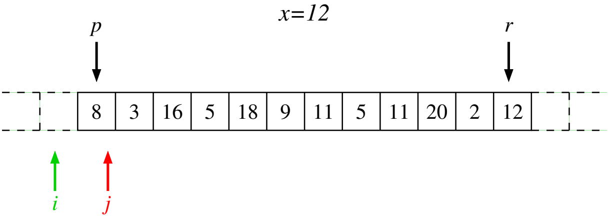
Quicksort Divide and Conquer

- Given (sub-) array with start/end indices p, r
- Select **pivot** element x from $A[p..r]$
- Re-distribute elements, so that for some index $p \leq q \leq r$:
  - $A[q] = x$
  - $A[h] \leq x$ for $p \leq h \leq q - 1$
  - $A[h] \geq x$ for $q + 1 \leq h \leq r$
- Recursively sort the arrays $A[p \,..\, q-1]$ and $A[q+1 \,..\, r]$

```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

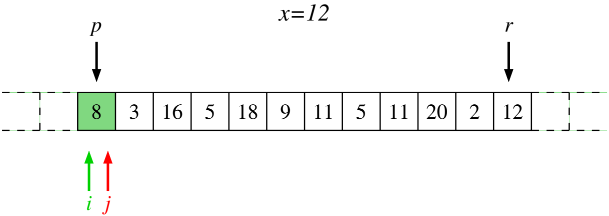


```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

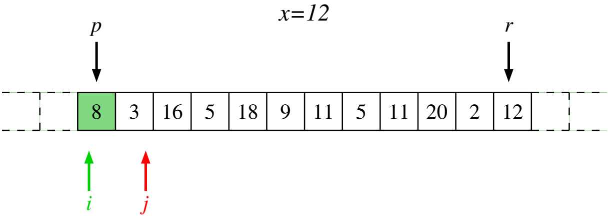


PARTITION(A, p, r)

```
// Use last element as pivot
x = A[r]
i = p - 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

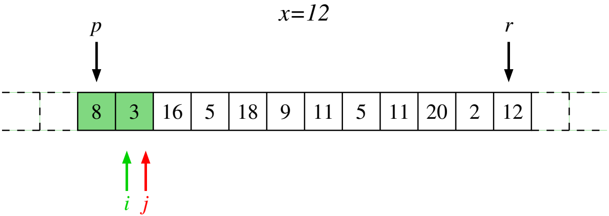


```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

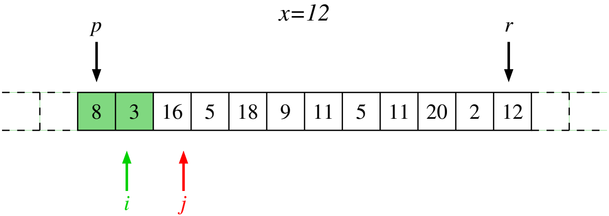


PARTITION(A, p, r)

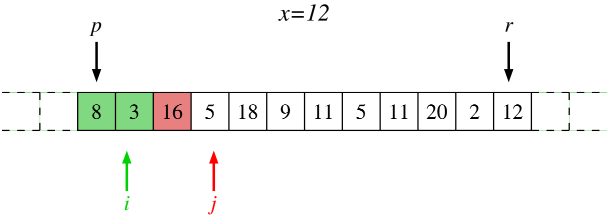
```
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```



```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

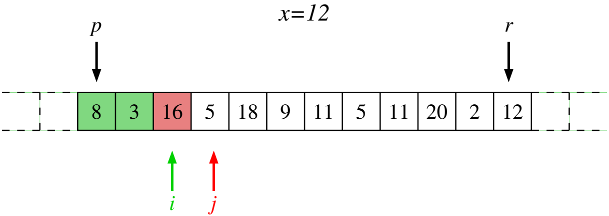


```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```



PARTITION(A, p, r)

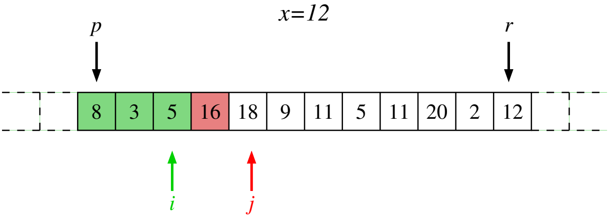
```
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```



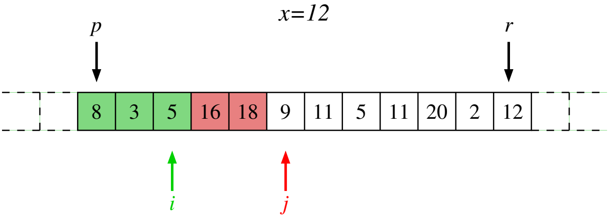
PARTITION(A, p, r)

```
// Use last element as pivot
```

$x = A[r]$
$i = p - 1$
for *j=p to r-1* **do**
    if $A[j] \leq x$ **then**
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
return *i+1*



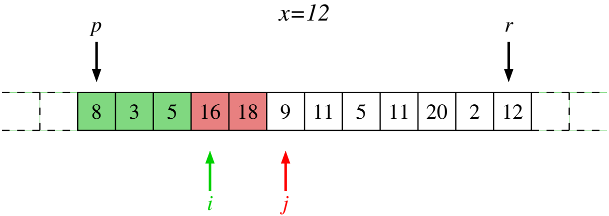
```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```



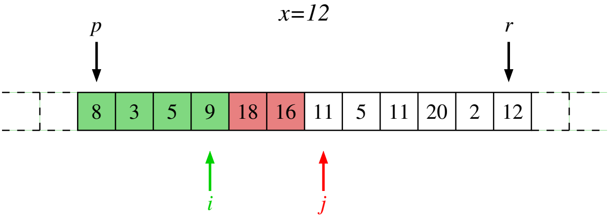
PARTITION(A, p, r)

```
// Use last element as pivot
```

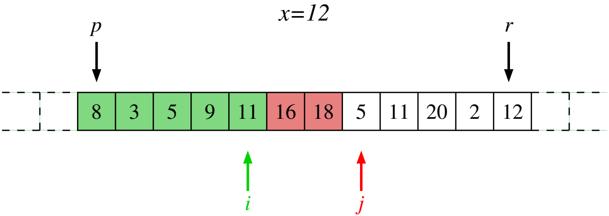
$x = A[r]$
$i = p - 1$
for *j=p to r-1* **do**
    if $A[j] \leq x$ **then**
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
return *i+1*



```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

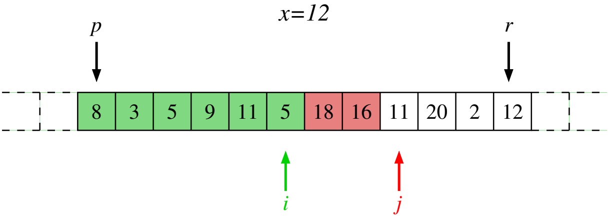


```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

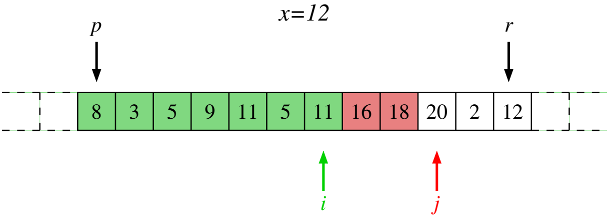


PARTITION(A, p, r)

```
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

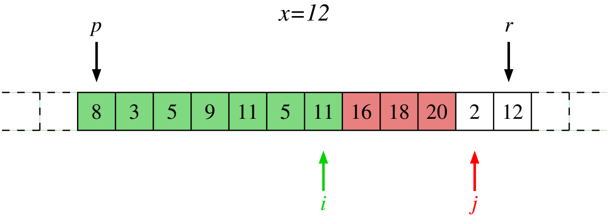


```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

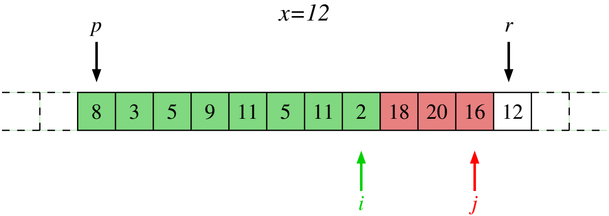


PARTITION(A, p, r)

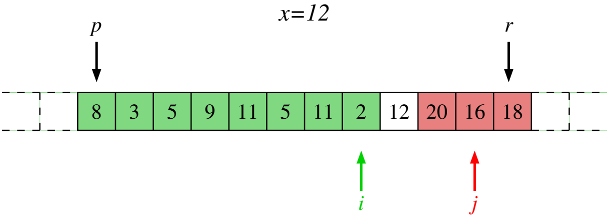
```
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```



```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```



```
PARTITION(A, p, r)
// Use last element as pivot
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```



Correctness: At the end of PARTITION:

- $A[i+1] = x$
- $A[h] \leq x$ for $p \leq h \leq i$
- $A[h] \geq x$ for $i+2 \leq h \leq r$

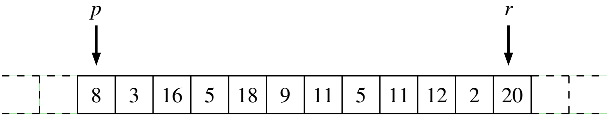
where x is the element originally located in $A[r]$.

This is proven using the loop invariant for lines 3-6 :

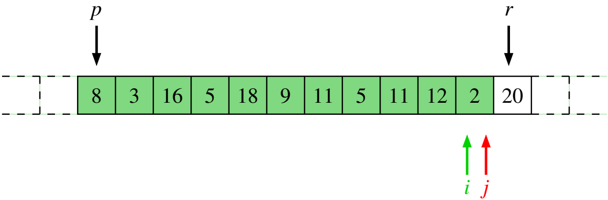
- $A[r] = x$
- $A[h] \leq x$ for $p \leq h \leq i$
- $A[h] > x$ for $i+1 \leq h \leq j-1$

Complexity: $\Theta(n)$ for $n = r - p$.

If $A[r]$ contains the maximal element of the array:



If $A[r]$ contains the maximal element of the array:



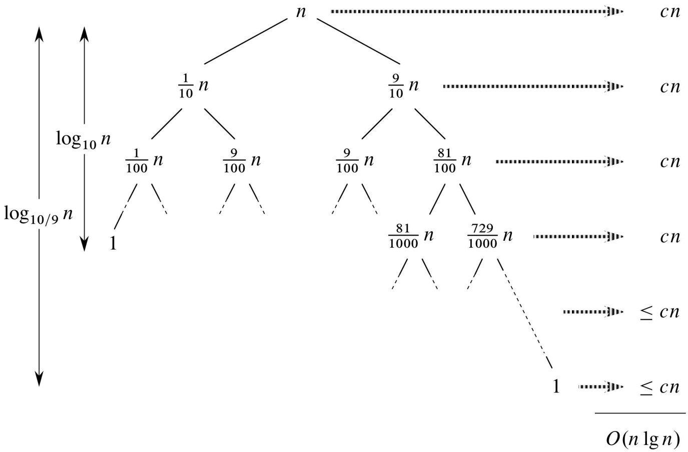
☞ Recurrence *for the case of already sorted arrays*: $T(n) = T(n-1) + \Theta(n)$

☞ $T(n) = \Theta(n^2)$

One can show that this is indeed the worst case, and therefore:

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

Recursion tree for the case that array always is split in a 1:9 ratio ([ItoA], Fig. 7.4):



Let c be a constant so that the PARTITION step is bounded by cn.

- No matter how the recursion tree is structured by the divisions in the PARTITION step: the *cost per level* in the recursion tree is bounded by cn.
- If the height of the recursion tree is $O(\lg n)$, then the total computation time is $O(n \lg n)$

Let c be a constant so that the PARTITION step is bounded by cn.

- No matter how the recursion tree is structured by the divisions in the PARTITION step: the *cost per level* in the recursion tree is bounded by cn.
- If the height of the recursion tree is $O(\lg n)$, then the total computation time is $O(n \lg n)$

R-balanced partitions

- Let $0.5 \leq R < 1$ be some constant
- Call a partitioning of $A[p \mathinner{..} r]$ **R-balanced**, if the larger of the created sub-arrays has size at most $R \cdot (r - p + 1)$
- If all partionings in the recursion tree are R-balanced, then the height of the tree is bounded by $\log_{1/R} n = O(\lg n)$.

Let c be a constant so that the PARTITION step is bounded by cn.

- No matter how the recursion tree is structured by the divisions in the PARTITION step: the *cost per level* in the recursion tree is bounded by cn.
- If the height of the recursion tree is $O(\lg n)$, then the total computation time is $O(n \lg n)$

R-balanced partitions

- Let $0.5 \leq R < 1$ be some constant
- Call a partitioning of $A[p \mathinner{..} r]$ **R-balanced**, if the larger of the created sub-arrays has size at most $R \cdot (r - p + 1)$
- If all partionings in the recursion tree are R-balanced, then the height of the tree is bounded by $\log_{1/R} n = O(\lg n)$.

Allowing bad splits

If there exists constants $0.5 \leq R < 1$ and $0 < K \leq 1$, such that

- on every path from the root to a leaf of the tree, the ratio of R-balanced vs. not R-balanced nodes is at least K

then the height of the tree is still $O(\lg n)$ (the height is increased at most by a factor of $1/K$ relative to an R-balanced tree).

Uniform input distribution

Assumption: for the input array *A* all orderings (permutations) of its elements are equally likely to occur.

☞ not always realistic, because arrays may more likely be partially sorted already

Randomized-Quicksort

- In PARTITION: first exchange $A[r]$ with a *randomly selected* element from $A[p\,..\,r]$.

Results

Two average runtimes:

- The average (or *expected*) runtime of RANDOMIZED-QUICKSORT: average number of steps taken for *any input* (average taken over different possible executions for one input)
- The average runtime of QUICKSORT under the uniform input distribution assumption (average taken over different inputs)

☞ both averages are identical, and equal to $\Theta(n \lg n)$.

Algorithm	Worst case	Average case	In place
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	yes
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	no
Bubble sort	$\Theta(n^2)$		yes
Heap sort	$O(n \lg n)$		yes
Quicksort	$\Theta(n^2)$	$O(n \lg n)$	yes