

# **Grundlæggende Algoritmer og Datastrukturer**

**Merge-Sort [CLRS, kapitel 2.3]**

**Heaps [CLRS, kapitel 6]**

# Merge-Sort

(Eksempel på Del-og-kombiner)

MERGE-SORT( $A, p, r$ )

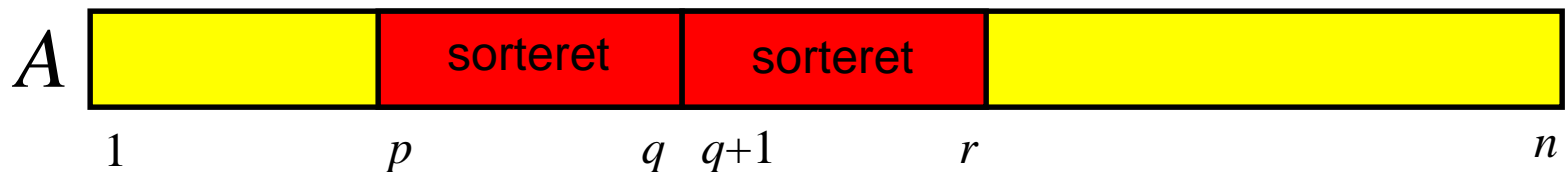
1 **if**  $p < r$

2      $q = \lfloor (p + r) / 2 \rfloor$

3     MERGE-SORT( $A, p, q$ )

4     MERGE-SORT( $A, q + 1, r$ )

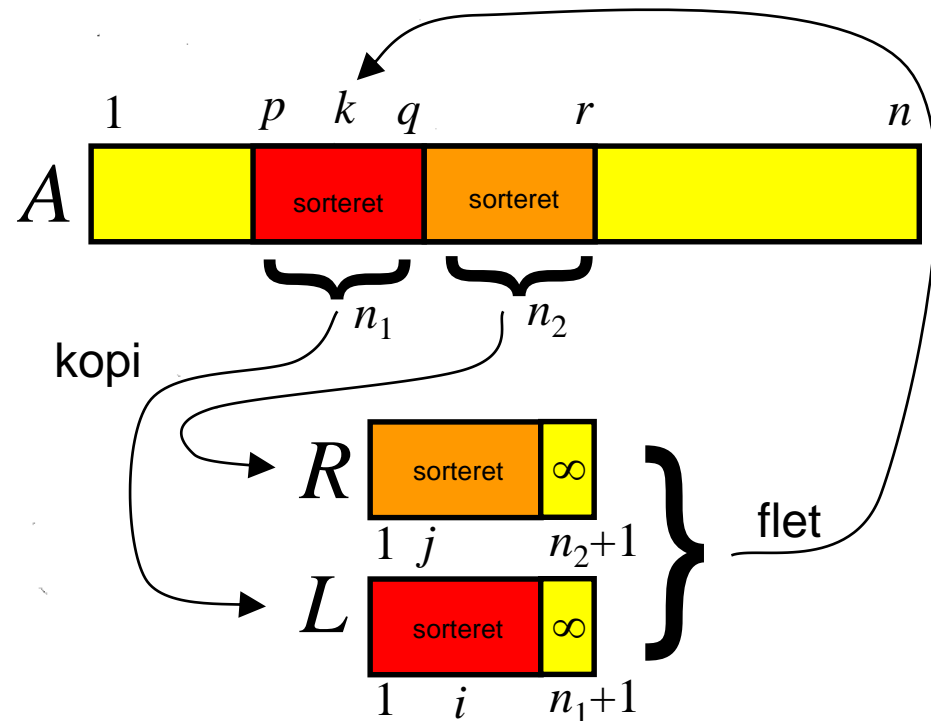
5     MERGE( $A, p, q, r$ )



I starten kaldes MERGE-SORT( $A, 1, n$ )

# MERGE( $A, p, q, r$ )

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```



# Hvor mange gange kan et element $y$ blive sammenlignet ? (worst-case)

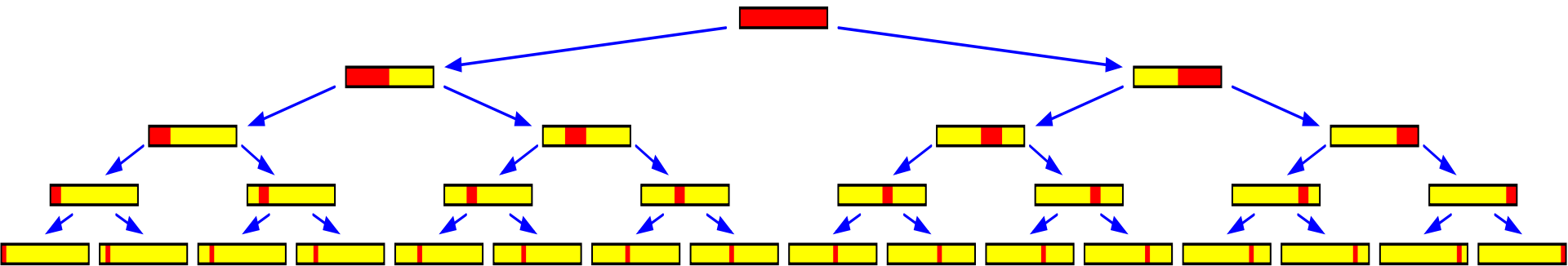
- a)  $O(\log n)$
- b)  $O(n)$
- c)  $O(n \log n)$
- d)  $O(n^2)$
- e) Ved ikke

```
MERGE-SORT( $A, p, r$ )  
1  if  $p < r$   
2       $q = \lfloor (p + r) / 2 \rfloor$   
3      MERGE-SORT( $A, p, q$ )  
4      MERGE-SORT( $A, q + 1, r$ )  
5      MERGE( $A, p, q, r$ )
```



# Merge-Sort : Analyse

## Rekursionstræet



## Observation

Samlet arbejde per lag er  $O(n)$

## Arbejde

$$O(n \cdot \# \text{ lag}) = O(n \cdot \log_2 n)$$

# MERGE-SORT( $A, p, r$ )

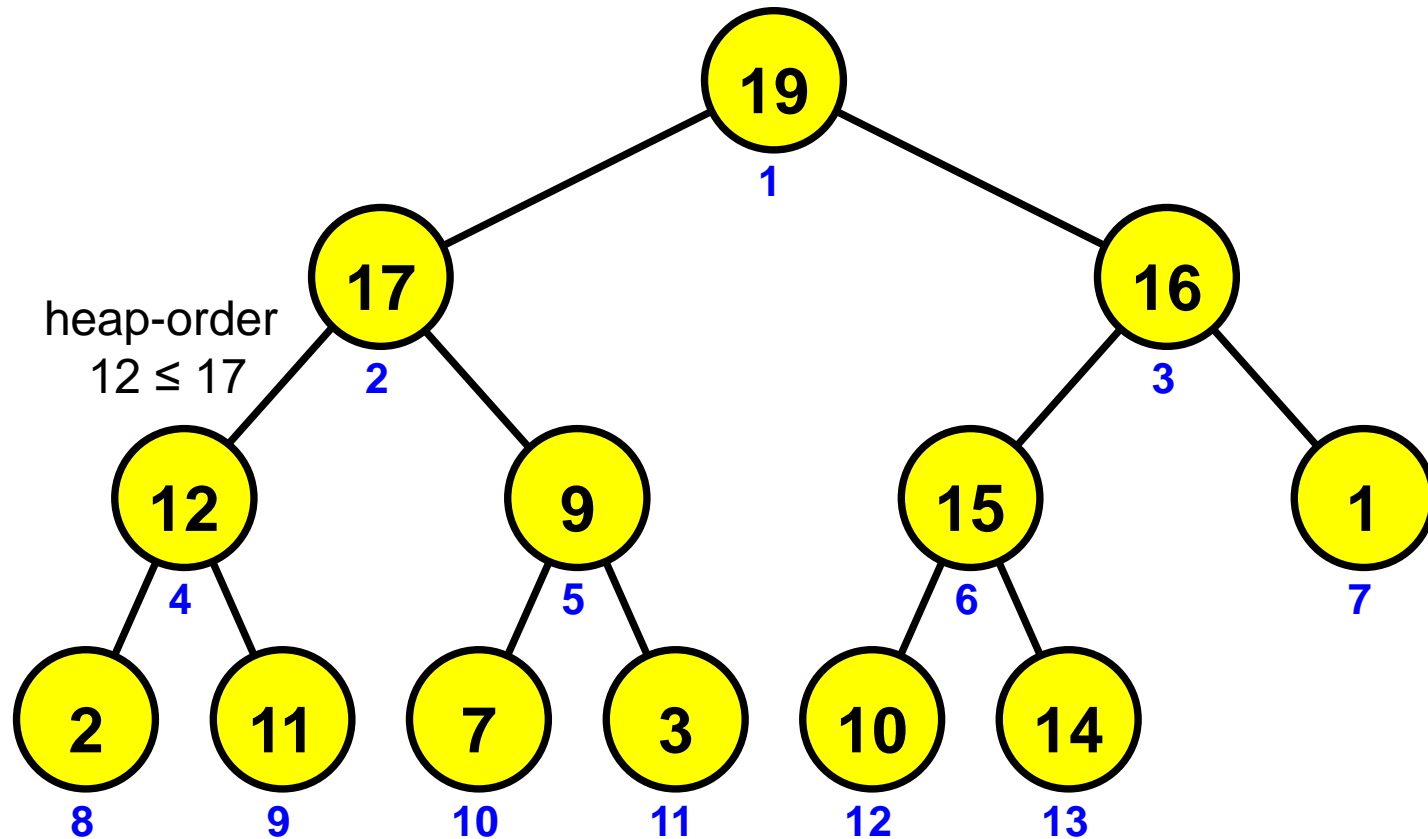
```
1  if  $p < r$ 
2       $q = \lfloor (p + r) / 2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

	# procedure- kald	# element- flytninger	# sammen- ligninger
a)	$O(\log n)$	$O(n)$	$O(n \log n)$
b)	$O(\log n)$	$O(n \log n)$	$O(n \log n)$
c)	$O(n)$	$O(n)$	$O(n \log n)$
d)	$O(n)$	$O(n \log n)$	$O(n \log n)$
e)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
f)		Ved ikke	



# Heap-Sort

# Binær (Max-)Heap



19	17	16	12	9	15	1	2	11	7	3	10	14
1	2	3	4	5	6	7	8	9	10	11	12	13



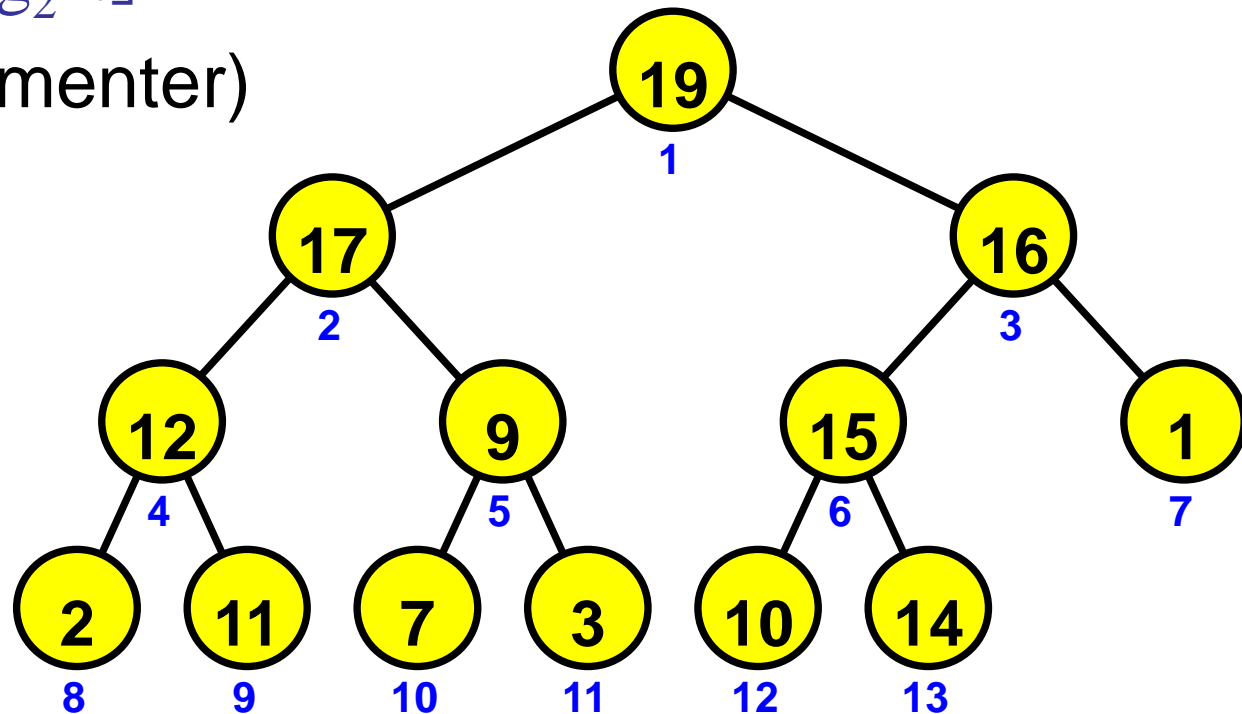
# Lovlig Max-Heap ?

- a) Ja
- b) Nej – 1 element opfylder ikke heap-order
- c) Nej – 2 elementer opfylder ikke heap-order
- d) Nej – 3 elementer opfylder ikke heap-order
- e) Nej – 4 elementer opfylder ikke heap-order
- f) Ved ikke

8	9	7	2	6	5	4	3	1
1	2	3	4	5	6	7	8	9

# Max-heap : Egenskaber

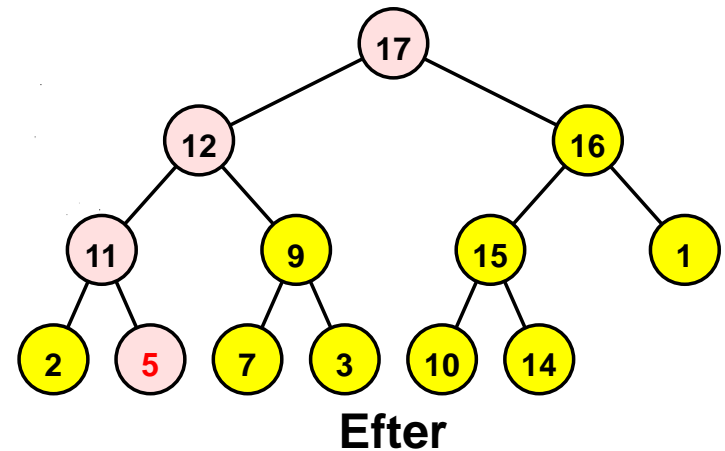
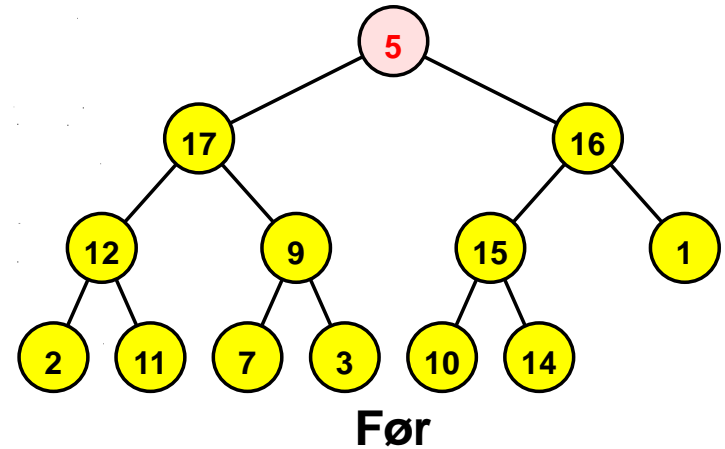
- Roden : knude 1
- Børn til knude  $i$  :  $2i$  og  $2i+1$
- Faren til knude  $i$  :  $\lfloor i / 2 \rfloor$
- Dybde :  $1 + \lfloor \log_2 n \rfloor$   
(  $n$  = antal elementer)



# Max-Heapify

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      $\text{largest} = l$
- 5 **else**  $\text{largest} = i$
- 6 **if**  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$
- 7      $\text{largest} = r$
- 8 **if**  $\text{largest} \neq i$
- 9     exchange  $A[i]$  with  $A[\text{largest}]$
- 10     MAX-HEAPIFY( $A, \text{largest}$ )



**Tid**  $O(\log n)$

# Heap-Sort

BUILD-MAX-HEAP( $A$ )

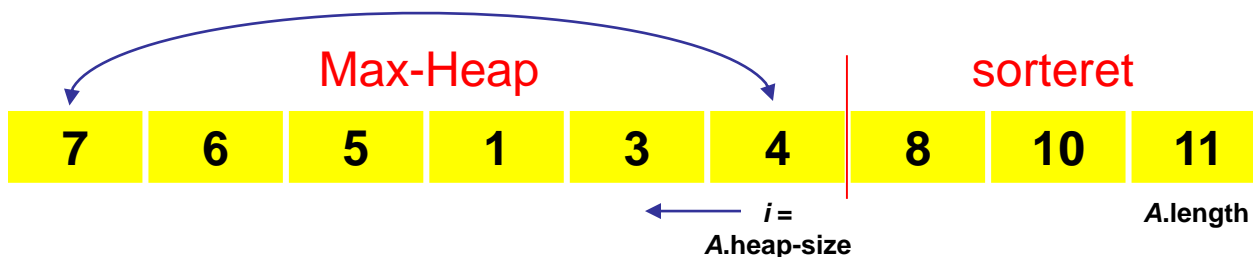
- 1  $A.heap\text{-}size = A.length$
- 2 **for**  $i = \lfloor A.length/2 \rfloor$  **downto** 1
- 3     MAX-HEAPIFY( $A, i$ )

Floyd, 1964

HEAPSORT( $A$ )

- 1 BUILD-MAX-HEAP( $A$ )
- 2 **for**  $i = A.length$  **downto** 2
- 3     exchange  $A[1]$  with  $A[i]$
- 4      $A.heap\text{-}size = A.heap\text{-}size - 1$
- 5     MAX-HEAPIFY( $A, 1$ )

Williams, 1964



Tid

$O(n \cdot \log n)$

# Resultatet af Build-Max-Heap ?

Input

1	2	3	4	5	6
1	2	3	4	5	6

a)

6	5	4	3	2	1
1	2	3	4	5	6

b)

6	5	3	4	2	1
1	2	3	4	5	6

c)

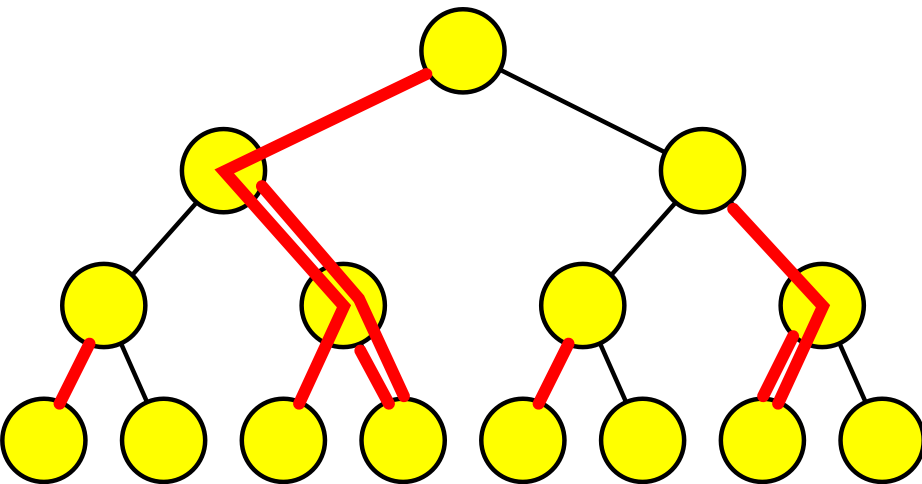
6	5	4	1	2	3
1	2	3	4	5	6

d)

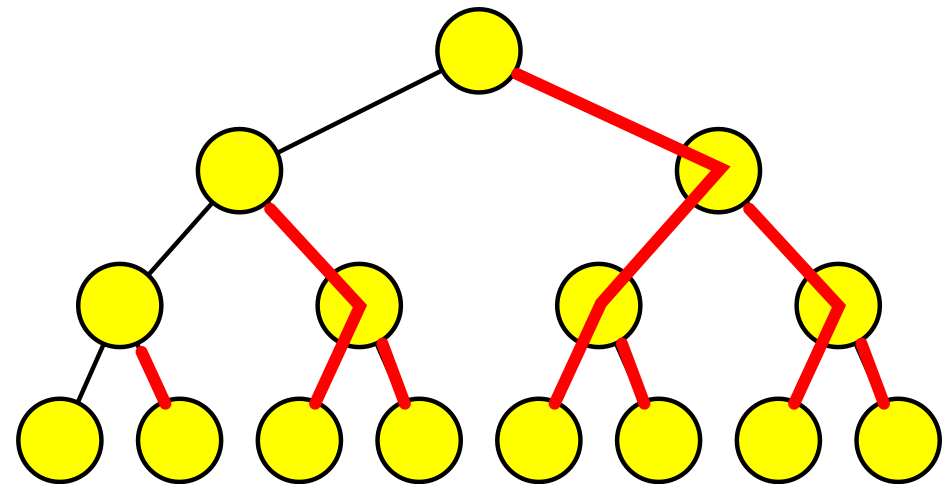
6	5	1	4	2	3
1	2	3	4	5	6

e) Ved ikke

# Build-Max-Heap



Max-Heapify stierne (eksempel)



Ikke-overlappende stier med samme #kanter (højre, venstre, venstre...)

Tid for Build-Max-Heap  
=  $\sum$  tid for Max-Heapify  
= **# røde kanter**

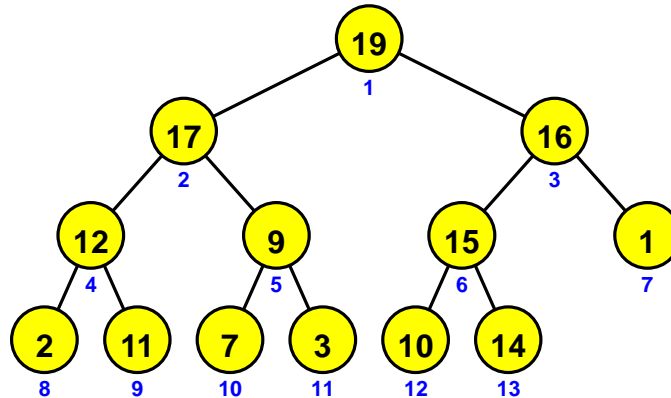
$\leq$  **# røde kanter**  
=  $n$  - dybde  
=  $O(n)$

**Tid  $O(n)$**

# Sorterings-algoritmer

Algoritme	Worst-Case Tid
Heap-Sort	$O(n \cdot \log n)$
Merge-Sort	
Insertion-Sort	$O(n^2)$

# Max-Heap operationer



HEAP-MAXIMUM( $A$ )

1 **return**  $A[1]$

HEAP-EXTRACT-MAX( $A$ )

```
1 if  $A.heap-size < 1$ 
2   error "heap underflow"
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  $max$ 
```

MAX-HEAP-INSERT( $A, key$ )

```
1  $A.heap-size = A.heap-size + 1$ 
2  $A[A.heap-size] = -\infty$ 
3 HEAP-INCREASE-KEY( $A, A.heap-size, key$ )
```

HEAP-INCREASE-KEY( $A, i, key$ )

```
1 if  $key < A[i]$ 
2   error "new key is smaller than current key"
3  $A[i] = key$ 
4 while  $i > 1$  and  $A[PARENT(i)] < A[i]$ 
5   exchange  $A[i]$  with  $A[PARENT(i)]$ 
6    $i = PARENT(i)$ 
```



# Max-Heap operation

Operation	Worst-Case Tid
Max-Heap-Insert	$O(\log n)$
Heap-Extract-Max	
Max-Increase-Key	
Heap-Maximum	$O(1)$

$n$  = aktuelle antal elementer i heapen

# Prioritetskø

En **prioritetskø** er en **abstrakt datastruktur** der gemmer en mængde af **elementer** med tilknyttet **nøgle** og understøtter operationerne:

- **Insert**( $S, x$ )
- **Maximum**( $S$ )
- **Extract-Max**( $S$ )

Maximum er med hensyn til de tilknyttede nøgler.

En mulig **implementation** af en prioritetskø er en **heap**.