Algoritmer og Datastrukturer 1

Gerth Stølting Brodal

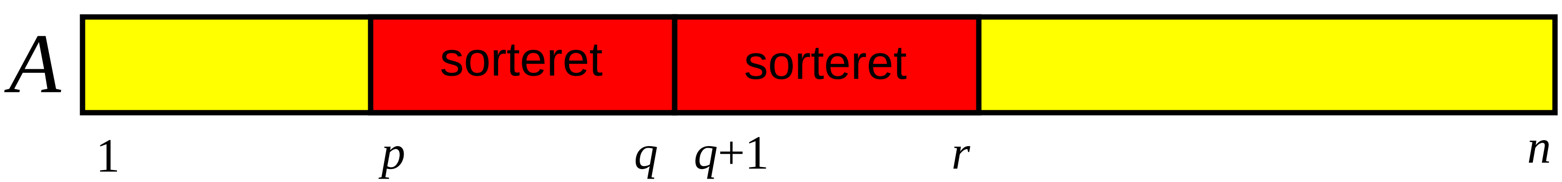
Merge-Sort [CLRS, kapitel 2.3]
Heaps [CLRS, kapitel 6]

AARHUS UNIVERSITET

Merge-Sort

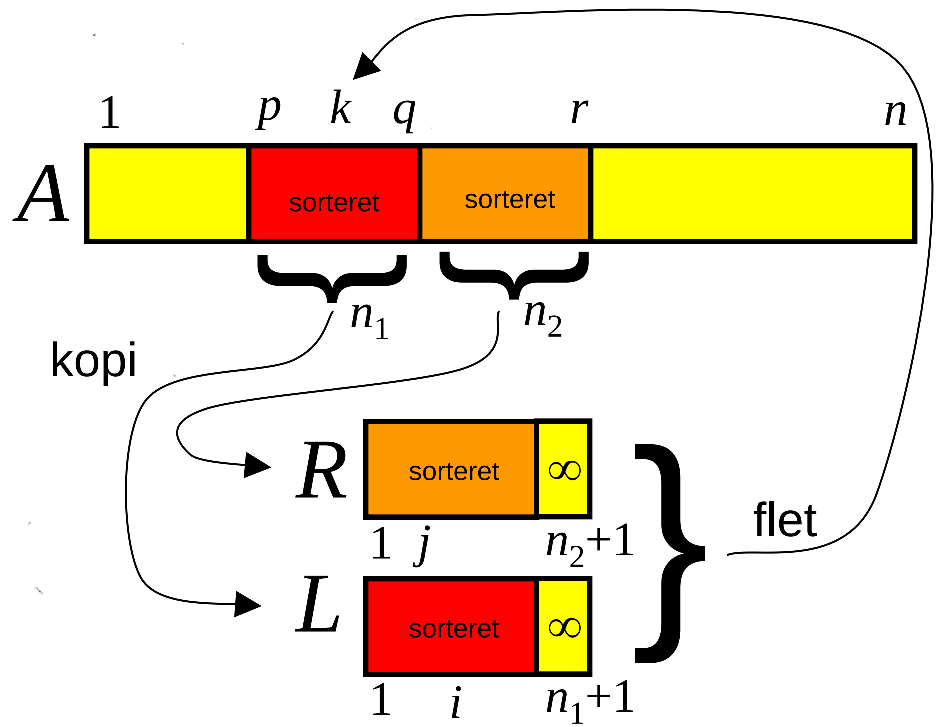
(Eksempel på Del-og-kombiner)

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```



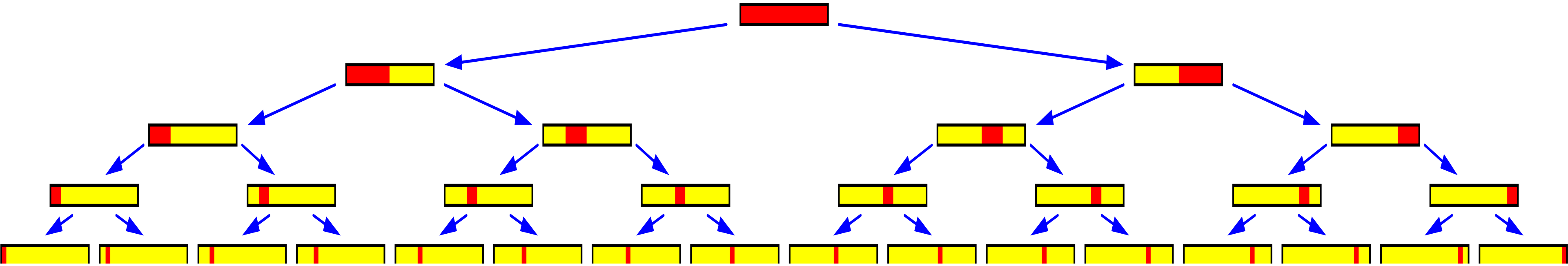
I starten kaldes MERGE-SORT(A,1,n)

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
        L[i] = A[p + i - 1]
for j = 1 to n2
        R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else A[k] = R[j]
                j = j + 1
```



Merge-Sort : Analyse

Rekursionstræet



Observation

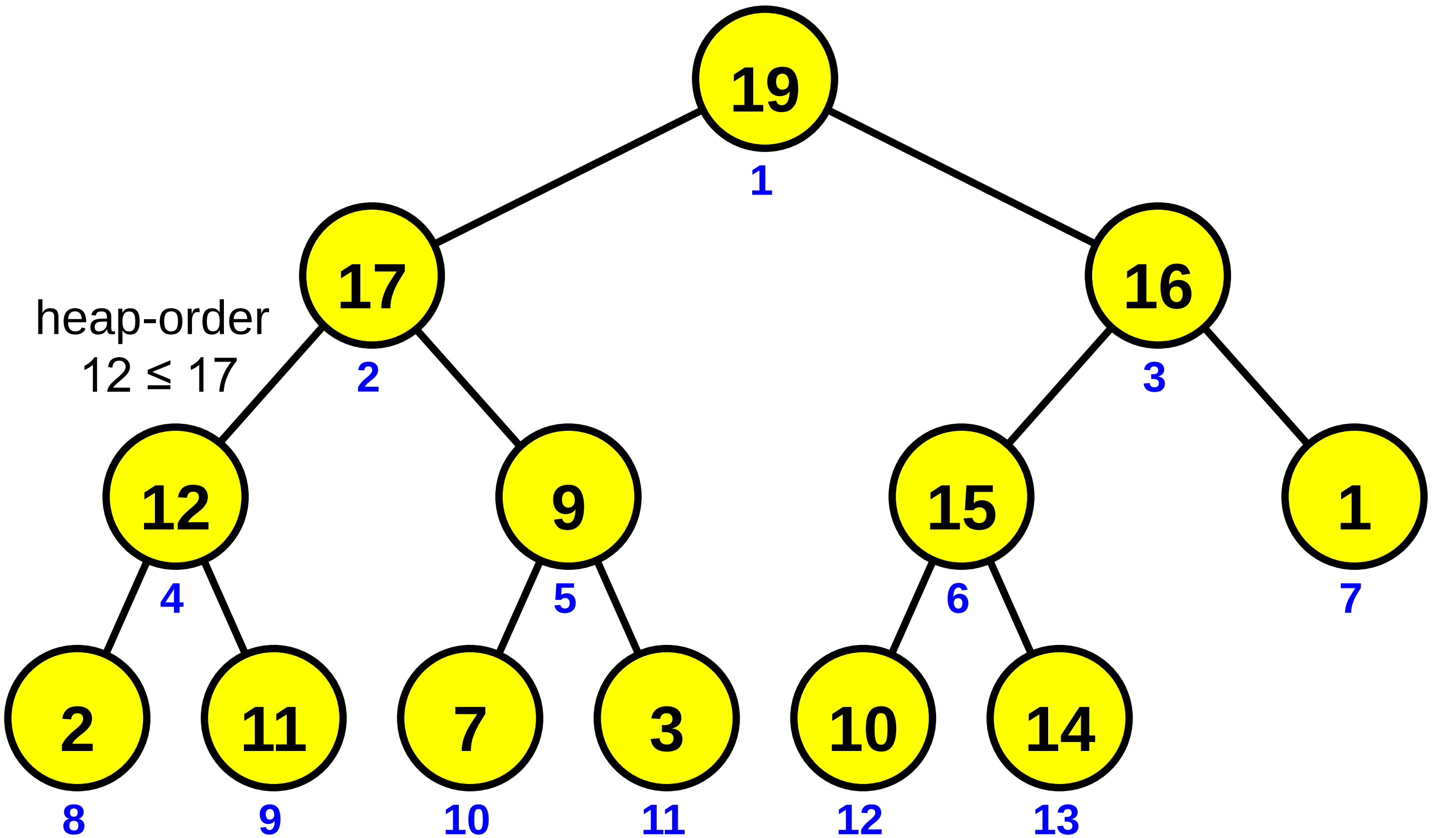
Samlet arbejde per lag er $O(n)$

Arbejde

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

Heap-Sort

Binær (Max-)Heap



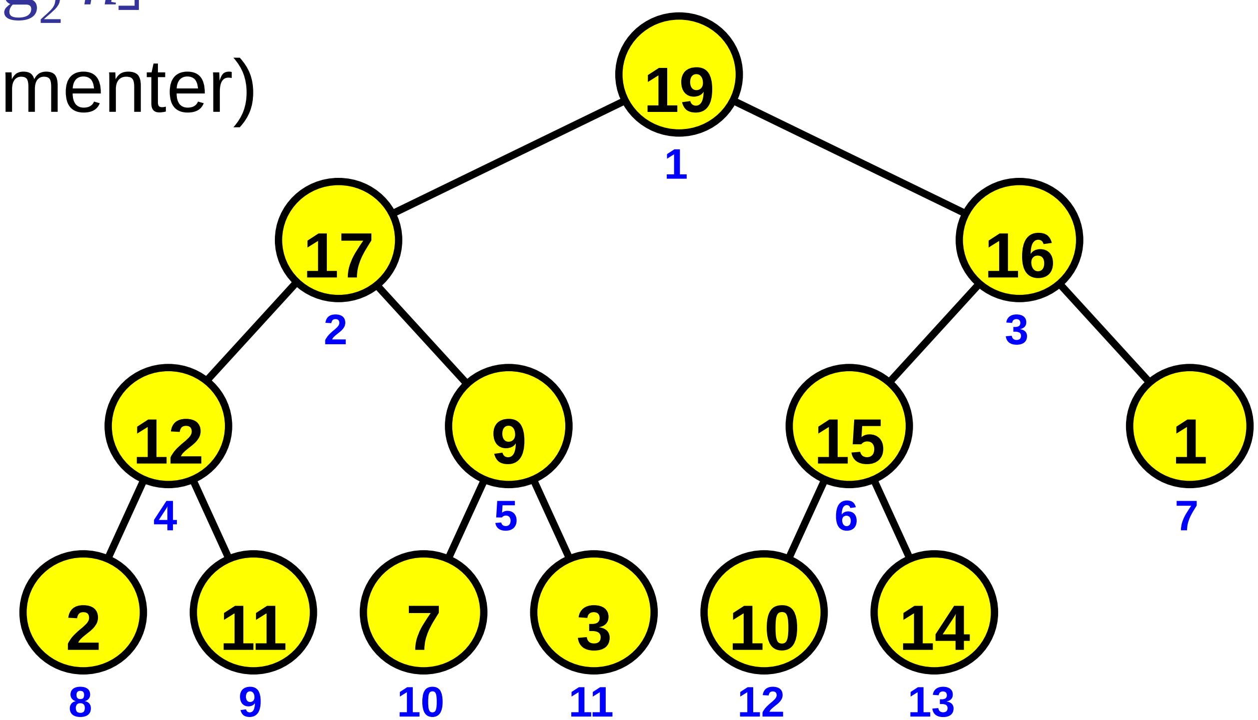
19	17	16	12	9	15	1	2	11	7	3	10	14
1	2	3	4	5	6	7	8	9	10	11	12	13

Williams, 1964

Max-heap : Egenskaber

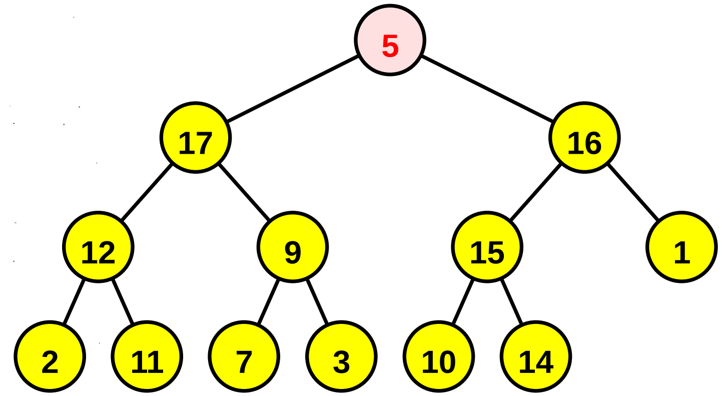
- Roden : knude 1
- Børn til knude i : $2i$ og $2i+1$
- Faren til knude i : $\lfloor i / 2 \rfloor$
- Dybde : $1+\lfloor \log_2 n \rfloor$

  (n = antal elementer)

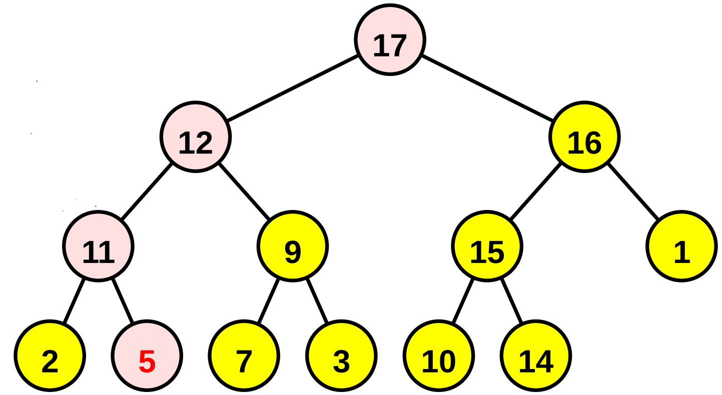


Max-Heapify

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```



Før



Efter

Tid O(log *n*)

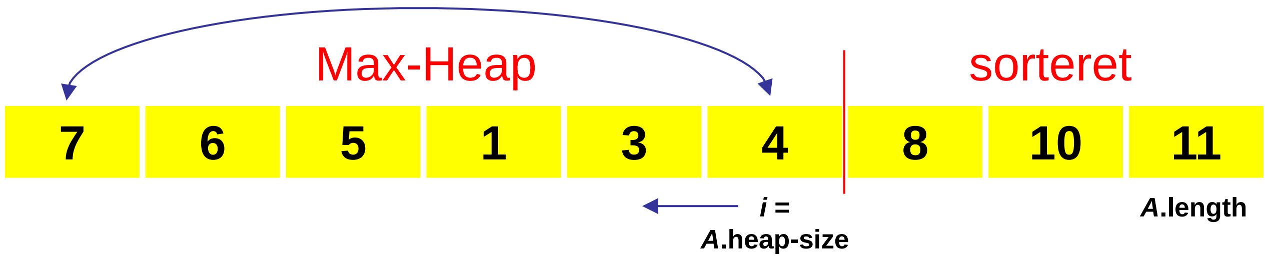
Heap-Sort

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

Floyd, 1964

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

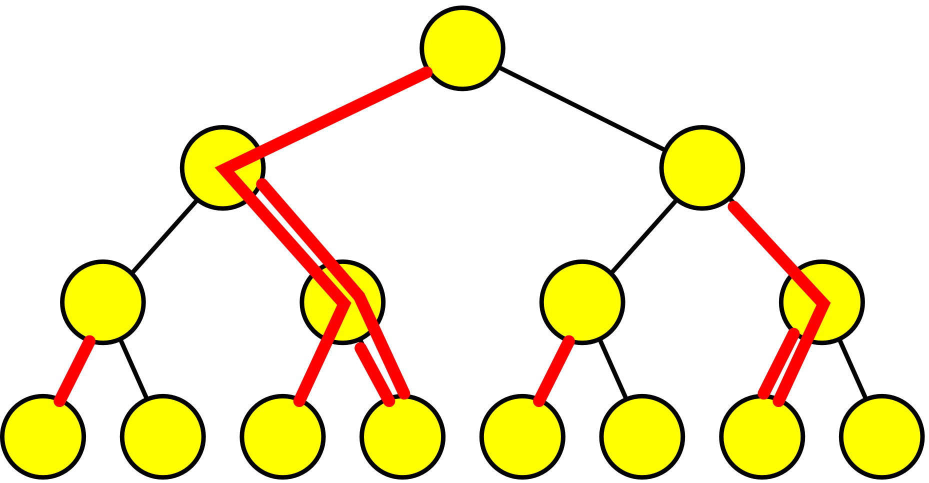
Williams, 1964



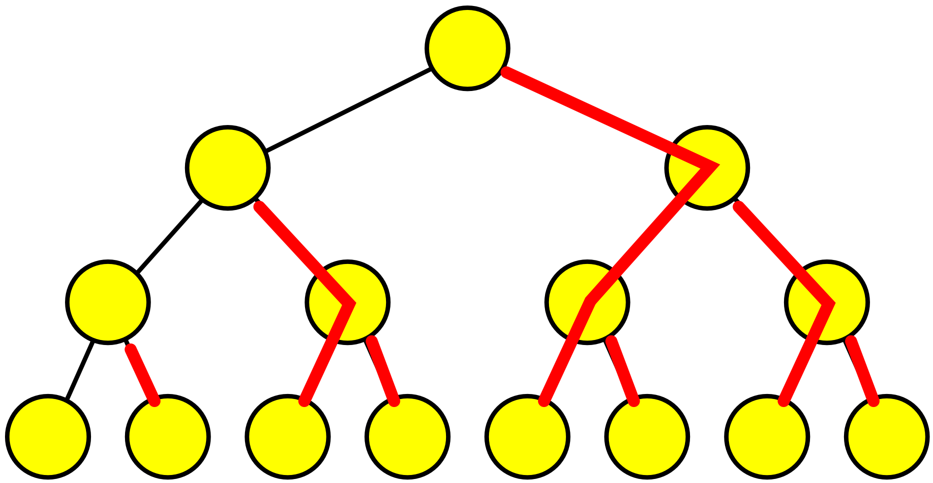
Tid

$O(n \cdot \log n)$

Build-Max-Heap



Max-Heapify stierne (eksempel)



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

Tid for Build-Max-Heap

= Σ tid for Max-Heapify

= # røde kanter

≤ # røde kanter

= *n* - dybde

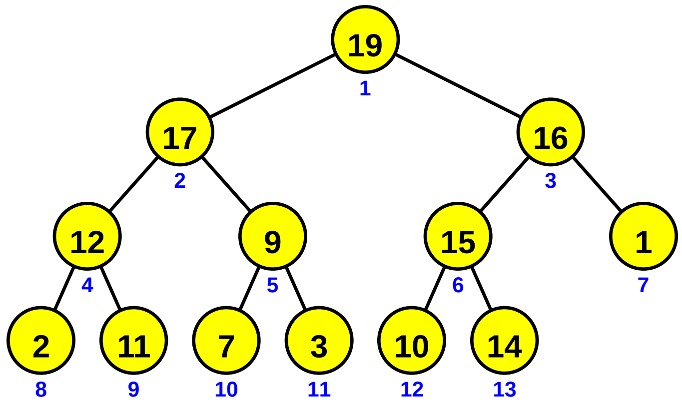
= O(n)

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)
return A[1]

HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max

MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
HEAP-INCREASE-KEY(A, A.heap-size, key)

HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    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.