HUSK EKSAMENSTILMELDING

Frist: 15. februar

Algoritmer og Datastrukturer 1

Gerth Stølting Brodal

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

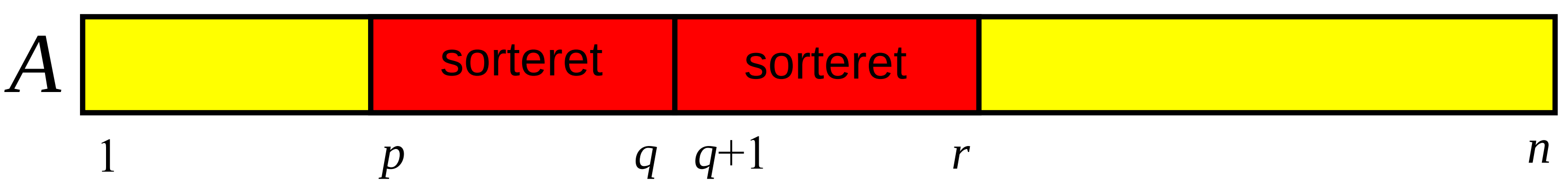
AARHUS UNIVERSITET

Merge-Sort

(Eksempel på Del-og-kombiner)

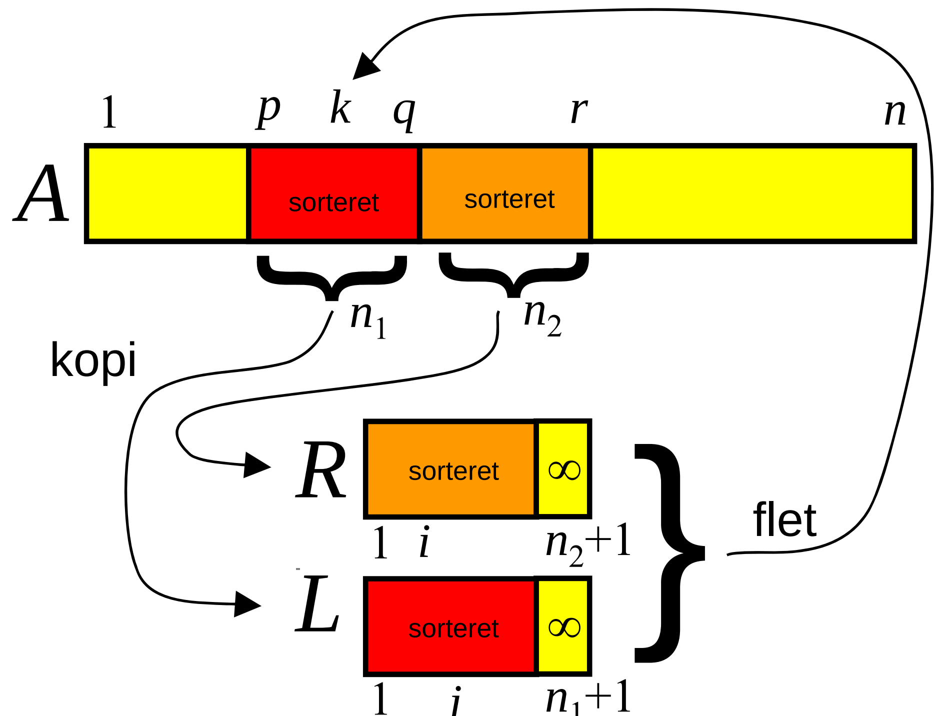
MERGE-SORT(A, p, r)

if $p < r$
    then $q \leftarrow \lfloor (p + r)/2 \rfloor$
        MERGE-SORT(A, p, q)
        MERGE-SORT$(A, q + 1, r)$
        MERGE(A, p, q, r)



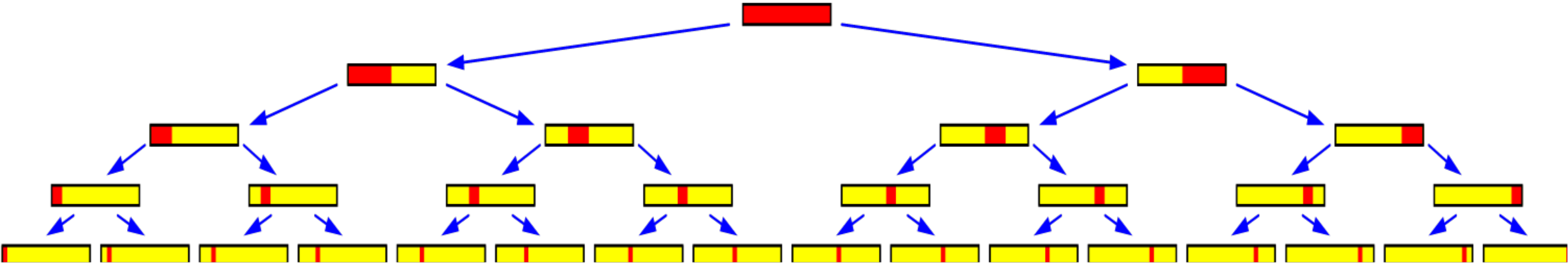
I starten kaldes MERGE-SORT$(A,1,n)$

```
MERGE(A, p, q, r)
    n1 ← q − p + 1
    n2 ← r − q
    create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
    for i ← 1 to n1
         do L[i] ← A[p + i − 1]
    for j ← 1 to n2
         do R[j] ← A[q + j]
    L[n1 + 1] ← ∞
    R[n2 + 1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
         do if L[i] ≤ R[j]
               then A[k] ← L[i]
                    i ← i + 1
               else A[k] ← R[j]
                    j ← j + 1
```



Merge-Sort : Analyse

Rekursionstræet



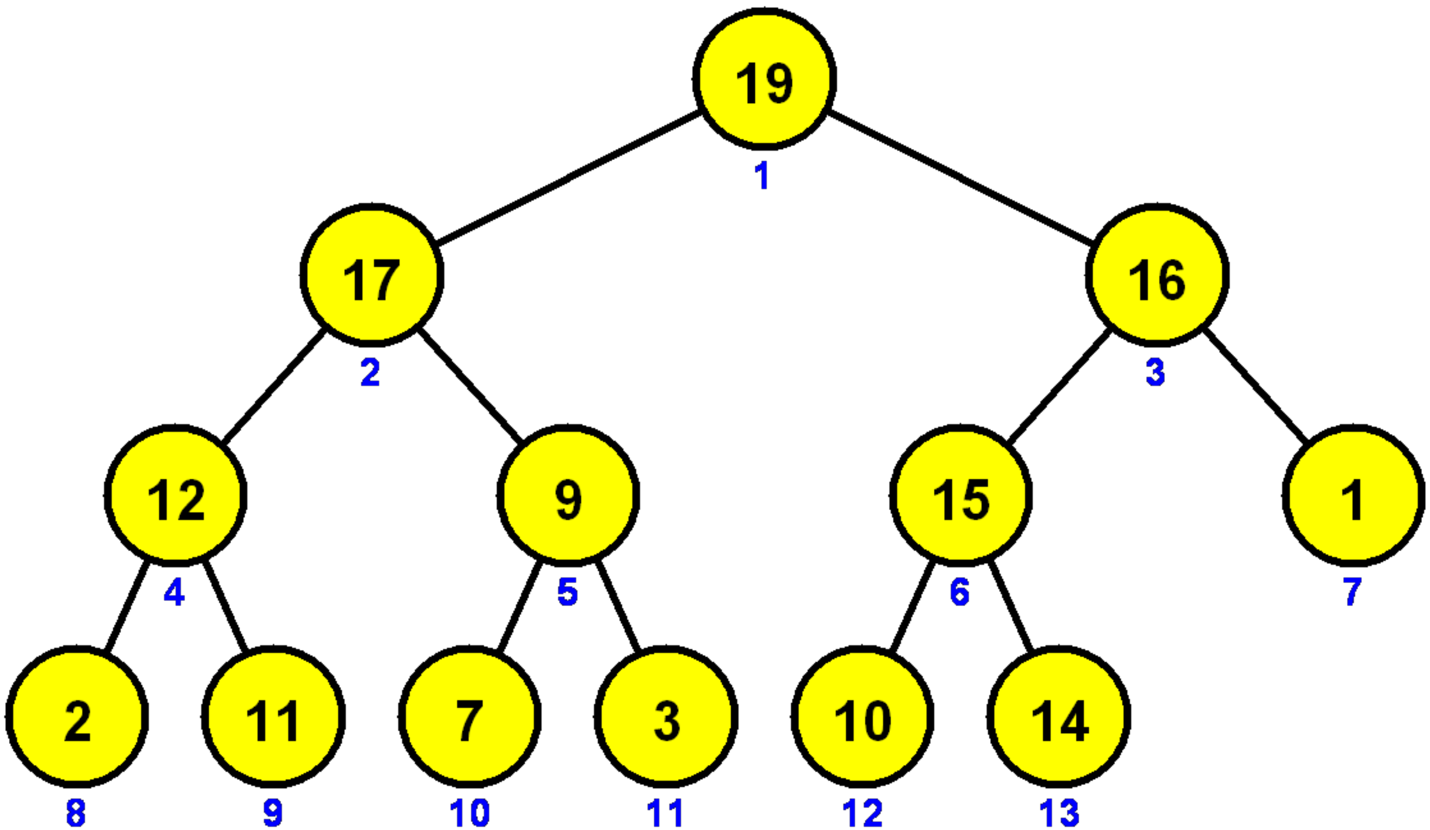
Observation

Samlet arbejde per lag er $O(n)$

Arbejde

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

Binær (Max-)Heap

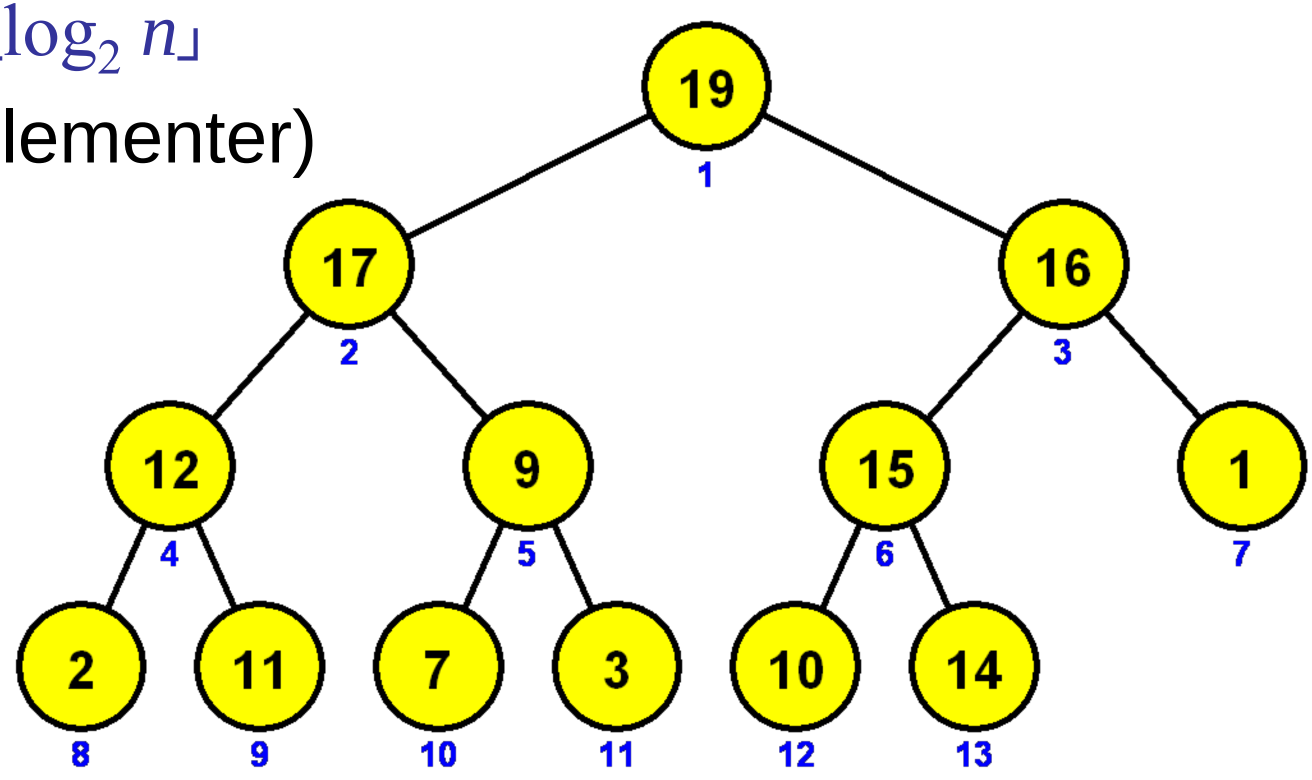


19	17	16	12	9	15	1	2	11	7	3	10	14
1	2	3	4	5	6	7	8	9	10	11	12	13

Williams, 1964

Max-heap : Egenskaber

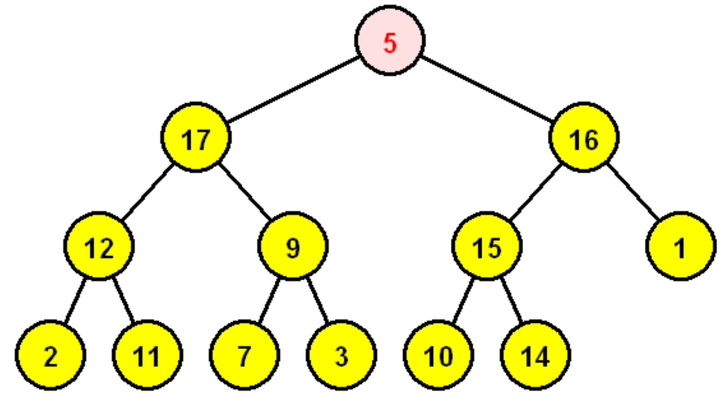
- Roden : knude 1
- Børn til knude i : $2i$ og $2i+1$
- Faren til knude i : $\lfloor i / 2 \rfloor$
- Dybde : $1+\lfloor \log_2 n \rfloor$
  (n = antal elementer)



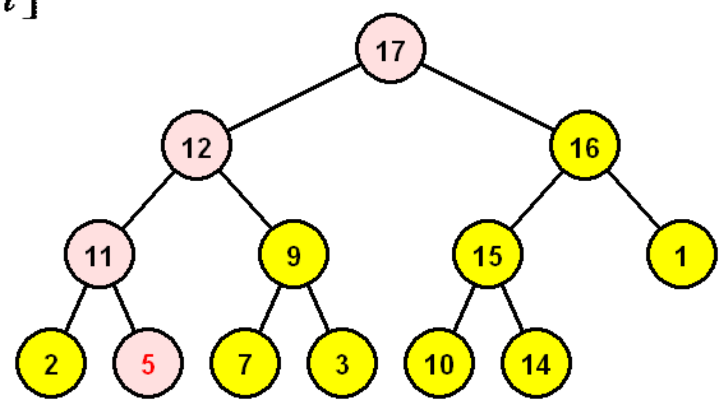
Max-Heapify

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

Tid O(log n)



Før



Efter

Heap-Sort

```
BUILD-MAX-HEAP(A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

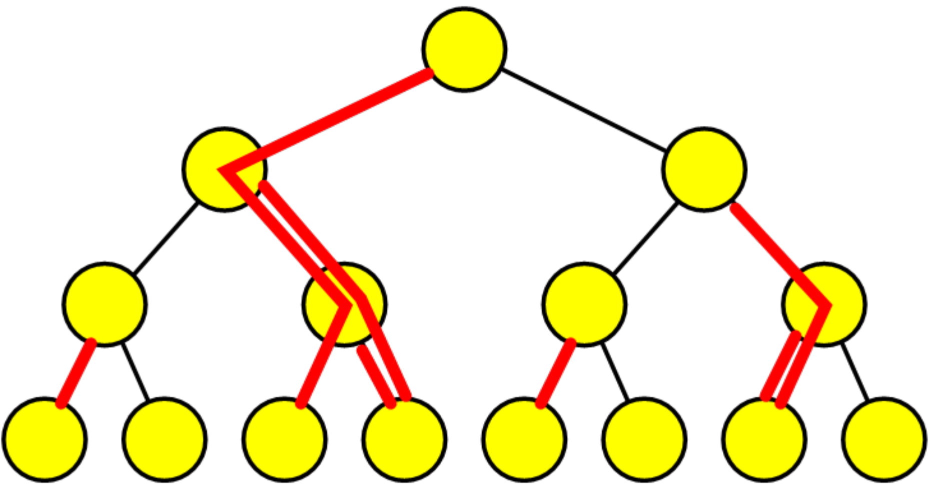
Floyd, 1964

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2
        do exchange A[1] ↔ A[i]
            heap-size[A] ← heap-size[A] − 1
            MAX-HEAPIFY(A, 1)
```

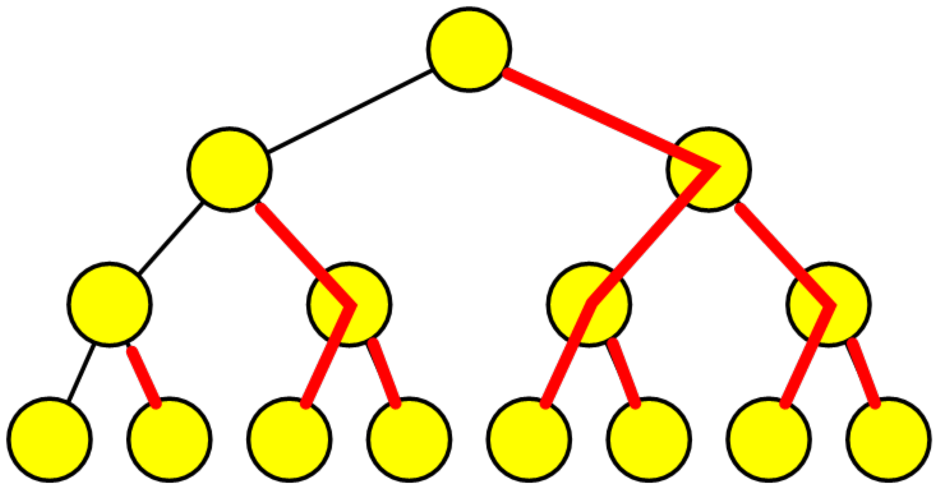
Williams, 1964

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	$O(n^2)$

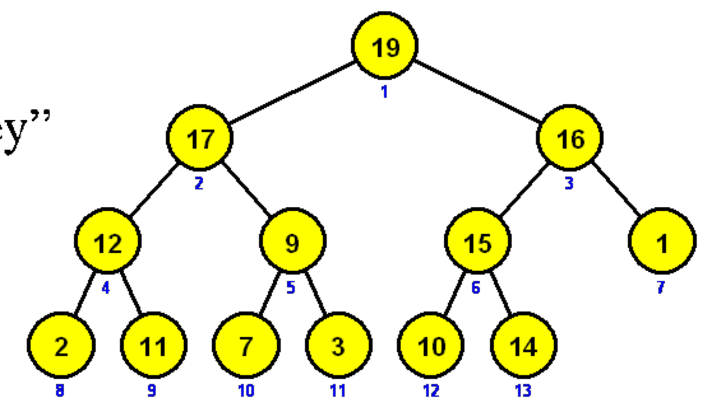
Max-Heap operationer

```
HEAP-MAXIMUM(A)
return A[1]
```

```
MAX-HEAP-INSERT(A, key)
heap-size[A] ← heap-size[A] + 1
A[heap-size[A]] ← −∞
HEAP-INCREASE-KEY(A, heap-size[A], key)
```

```
HEAP-EXTRACT-MAX(A)
if heap-size[A] < 1
   then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] − 1
MAX-HEAPIFY(A, 1)
return max
```

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
   then error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] ↔ A[PARENT(i)]
       i ← PARENT(i)
```



Max-Heap operation

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