

# Sortering

# Sortering

**Input:**  $n$  tal

**Output:** De  $n$  tal i sorteret orden

Eksempel:

6, 2, 9, 4, 5, 1, 4, 3  $\rightarrow$  1, 2, 3, 4, 4, 5, 6, 9

Mange opgaver er hurtigere i sorterede data (tænk på ordbøger, adresselister i telefoner, ...). Dette gælder både for mennesker og for computere. Sortering er ofte en byggesten i algoritmer for andre problemer.

Sortering er en fundamental og central opgave.

Mange algoritmer er udviklet: Insertionsort, Selectionsort, Bubblesort, Mergesort, Quicksort, Heapsort, Radixsort, Countingsort, ...

Vi skal møde alle ovenstående i dette kursus.

# Sortering

Kommentarer:

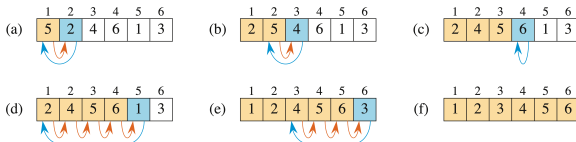
- ▶ Sorteret orden kan være stigende eller faldende. Vi vil i dette kursus altid bruge stigende (mere præcist: ikke-faldende). Skal man sortere faldende, skal alle sammenligninger bare vendes.
- ▶ Vi vil antage, at input ligger i en liste (Python) / et array (Java).
- ▶ Man sorterer ofte elementer sammensat af en sorteringsnøgle samt yderligere information. Sorteringsnøglen kan være et tal, kommatal, eller andet der kan sammenlignes (f.eks. strenge/ord). Vi viser i dette kursus blot elementer som heltal.

# Insertionsort

Bruges af mange, når man sorterer en hånd i kort:



Samme idé udført på tal i en liste / et array:



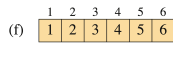
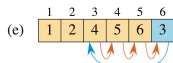
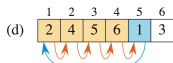
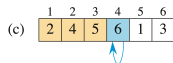
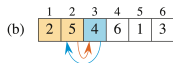
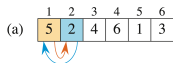
Argument for korrekthed: Den gule del af array er altid sorteret. Denne del udvides med én hele tiden ( $\Rightarrow$  algoritmen stopper, og når den stopper er alle elementer sorteret).

# Insertionsort

Som pseudo-kode:

INSERTION-SORT( $A, n$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3      // Insert  $A[i]$  into the sorted subarray  $A[1 : i - 1]$ .
4       $j = i - 1$ 
5      while  $j > 0$  and  $A[j] > key$ 
6           $A[j + 1] = A[j]$ 
7           $j = j - 1$ 
8       $A[j + 1] = key$ 
```



# Analyse af køretid for Insertionsort

| INSERTION-SORT( $A, n$ )   | <i>cost</i> | <i>times</i>             |
|--|-------------|--------------------------|
| 1 <b>for</b> $i = 2$ <b>to</b> $n$                               | $c_1$       | $n$                      |
| 2 $key = A[i]$   | $c_2$       | $n - 1$                  |
| 3     // Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$ . | 0           | $n - 1$                  |
| 4 $j = i - 1$  | $c_4$       | $n - 1$                  |
| 5 <b>while</b> $j > 0$ and $A[j] > key$                          | $c_5$       | $\sum_{i=2}^n t_i$       |
| 6 $A[j + 1] = A[j]$  | $c_6$       | $\sum_{i=2}^n (t_i - 1)$ |
| 7 $j = j - 1$  | $c_7$       | $\sum_{i=2}^n (t_i - 1)$ |
| 8 $A[j + 1] = key$   | $c_8$       | $n - 1$                  |

Her er  $t_i$  hvor mange gange testen i den indre **while**-løkke udføres. Dvs.  $t_i - 1$  er hvor mange gange denne løkke kører (hvilket er hvor mange elementer det  $i$ 'te element skal forbi under indsættelsen). Bemærk, at  $1 \leq t_i \leq i$ . Sæt  $c = c_1 + c_2 + \dots + c_8$ .

**Best case:**  $t_i = 1$  for alle  $i$ . Samlet tid  $\leq c \cdot n$ .

**Worst case:**  $t_i = i$  for alle  $i$ . Samlet tid  $\leq c \cdot n^2$ , eftersom

$$\sum_{i=2}^n i \leq (1 + 2 + 3 + \dots + n) = \frac{(n+1)n}{2} = \frac{n^2 + n}{2} \leq \frac{2n^2}{2} = n^2.$$

# Selectionsort

En anden simpel og naturlig sorteringsalgoritme:

```
indListe = input
udListe = tom liste
while indListe ikke tom:
    find mindste element  $x$  i indListe
    flyt  $x$  fra indListe til enden af udListe
```

Klart **korrekt**, dvs. giver sorteret output (hvert element, som udtages, er mindre eller lig alle efterfølgende, der udtages).

## Køretid?

I alt  $n$  gange findes mindste element i `IndListe`.

En simpel metode til at finde mindste element er lineær søgning, som kigger én gang på hvert tilbageværende element.

Derved bliver tiden  $\leq c \cdot (n + (n - 1) + (n - 2) + \dots + 1) \leq c \cdot n^2$ .

# Merge

**Input:** To **sorterede** rækker  $A$  og  $B$

**Output:** De samme elementer i én sorteret række  $C$

Eksempel:

$$\begin{array}{l} A = 2,4,5,7,8 \\ B = 1,2,3,6 \end{array} \Rightarrow C = 1,2,2,3,4,5,6,7,8$$

Vi kan naturligvis sortere  $A \cup B$ .

Men det er hurtigere at **flette** (**merge**):

**Repeat:**

Flyt det mindste af de to forreste elementer i  $A$  og  $B$  til enden af  $C$

**Køretid:**  $\leq c \cdot n$ , hvor  $n$  = antal elementer i alt i  $A$  og  $B$ .

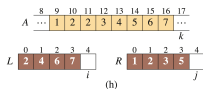
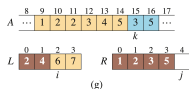
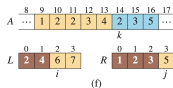
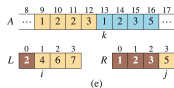
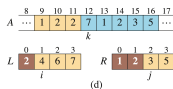
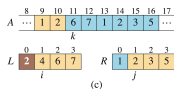
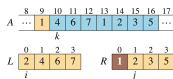
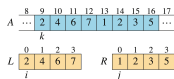
**Korrekthed:** Merge kan ses som en udgave af Selectionsort, der udnytter at  $A$  og  $B$  er sorterede, så den mindste i (resten af)  $A \cup B$  kan findes ved kun at se på de to forreste i  $A$  og  $B$ , hvilket tager konstant tid.



## Eksempel på merge

Her antages, at de to input-lister er nabodele af samme liste/array  $A$ , nemlig  $A[p \dots q]$  og  $A[q + 1 \dots r]$ . De flyttes først over i  $L$  og  $R$ .

(Bemærk, at i bogen er  $A[p \dots q]$  lig elementerne  $A[p]$ ,  $A[p + 1]$ , ...,  $A[q]$ , hvilket er ét element mere end næsten samme notation i Python.)

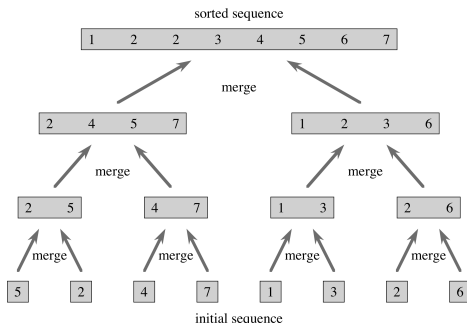


# Pseudo-kode for Merge

```
MERGE( $A, p, q, r$ )
1   $n_L = q - p + 1$       // length of  $A[p : q]$ 
2   $n_R = r - q$           // length of  $A[q + 1 : r]$ 
3  let  $L[0 : n_L - 1]$  and  $R[0 : n_R - 1]$  be new arrays
4  for  $i = 0$  to  $n_L - 1$  // copy  $A[p : q]$  into  $L[0 : n_L - 1]$ 
5       $L[i] = A[p + i]$ 
6  for  $j = 0$  to  $n_R - 1$  // copy  $A[q + 1 : r]$  into  $R[0 : n_R - 1]$ 
7       $R[j] = A[q + j + 1]$ 
8   $i = 0$                 //  $i$  indexes the smallest remaining element in  $L$ 
9   $j = 0$                 //  $j$  indexes the smallest remaining element in  $R$ 
10  $k = p$                //  $k$  indexes the location in  $A$  to fill
11 // As long as each of the arrays  $L$  and  $R$  contains an unmerged element,
    // copy the smallest unmerged element back into  $A[p : r]$ .
12 while  $i < n_L$  and  $j < n_R$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
18      $k = k + 1$ 
19 // Having gone through one of  $L$  and  $R$  entirely, copy the
    // remainder of the other to the end of  $A[p : r]$ .
20 while  $i < n_L$ 
21      $A[k] = L[i]$ 
22      $i = i + 1$ 
23      $k = k + 1$ 
24 while  $j < n_R$ 
25      $A[k] = R[j]$ 
26      $j = j + 1$ 
27      $k = k + 1$ 
```

# Mergesort

Mergesort: opbyg længere og længere sorterede dele af input ved gentagen brug af merge:

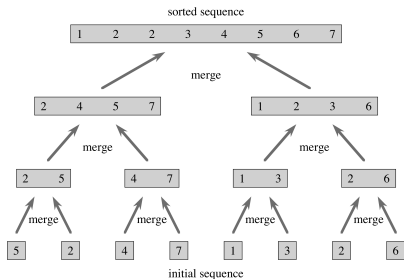


Tid: Hver merge bruger højst  $c \cdot n_1$  tid, hvor  $n_1$  er antal elementer der merges. Så alle merge-operationer i ét lag bruger tilsammen højst  $c \cdot (n_1 + n_2 + \dots) = c \cdot n$ . Dette gælder alle lagene. Der er i alt  $\log_2 n$  lag, så den samlede tid er højst  $c \cdot n \cdot \log_2 n$ .

# Mergesort

Hvorfor er der  $\log_2 n$  merge-lag?

Antal sorterede lister efter  $k$  merge-lag (antag at  $n$  er en potens af 2):



| $k$      | Antal lister |
|----------|--------------|
| $\vdots$ | $\vdots$     |
| $k$      | $n/2^k$      |
| $\vdots$ | $\vdots$     |
| 3        | $n/2^3$      |
| 2        | $n/2^2$      |
| 1        | $n/2$        |
| 0        | $n$          |

Algoritmen stopper, når der er én sorteret liste:

$$n/2^k = 1 \Leftrightarrow n = 2^k \Leftrightarrow \log_2 n = k$$

## Mergesort hvis $n$ ikke er en potens af 2?

Algoritmen merger i hvert lag så mange par, den kan, og der bliver evt. én liste, som ikke merges (denne er med som liste på næste lag).

F.eks. bliver 12 lister til  $12/2 = 6$  lister mens 13 (= 12 + 1) lister bliver til  $12/2 + 1 = 6 + 1 = 7$  lister.

Generelt: Hvis der er  $x$  lister før et merge-lag, er der  $\lceil x/2 \rceil$  lister efter.

Se på to input størrelse  $n$  og  $n'$ , med  $n \leq n'$ . Da  $\lceil x/2 \rceil$  er en voksende funktion af  $x$ , kan antallet af lister i hvert lag ikke være mindre for  $n'$  end for  $n$ . Derfor kan antallet af lag (før algoritmen når ned på én liste) ikke være mindre for  $n'$  end for  $n$ .

Sæt  $n'$  til mindste potens af to, som er større end eller lig  $n$ . Der er præcis  $\log_2 n'$  lag for  $n'$ , og dermed højst så mange lag for  $n$ .

Omvendt er det nemt at se, at for  $n = 2^k + 1$  er der  $k + 1 = \lceil \log_2 n \rceil$  lag. Så der er  $\lceil \log_2 n \rceil$  lag for generelt  $n$ .

|             |       |           |       |       |       |       |       |       |       |            |       |
|-------------|-------|-----------|-------|-------|-------|-------|-------|-------|-------|------------|-------|
| $n$         | 7     | $8 = 2^3$ | 9     | 10    | 11    | 12    | 13    | 14    | 15    | $16 = 2^4$ | 17    |
| $\log_2(n)$ | 2.807 | 3         | 3.169 | 3.321 | 3.459 | 3.584 | 3.700 | 3.807 | 3.906 | 4          | 4.087 |
| Antal lag   | 3     | 3         | 4     | 4     | 4     | 4     | 4     | 4     | 4     | 4          | 5     |

# Mergesort

Mergesort som pseudo-kode, i en variant formuleret med rekursion:

```
MERGE-SORT( $A, p, r$ )
1  if  $p \geq r$                                // zero or one element?
2      return
3   $q = \lfloor (p + r)/2 \rfloor$                    // midpoint of  $A[p : r]$ 
4  MERGE-SORT( $A, p, q$ )                         // recursively sort  $A[p : q]$ 
5  MERGE-SORT( $A, q + 1, r$ )                     // recursively sort  $A[q + 1 : r]$ 
6  // Merge  $A[p : q]$  and  $A[q + 1 : r]$  into  $A[p : r]$ .
7  MERGE( $A, p, q, r$ )
```

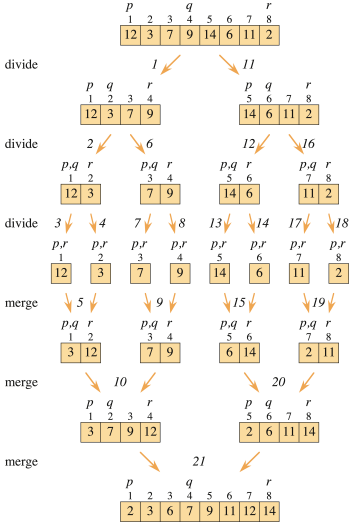
Et kald  $\text{MERGE-SORT}(A, p, r)$  har til opgave at stille elementerne i  $A[p \dots r]$  i sorteret orden.

Første kald er  $\text{MERGE-SORT}(A, 1, n)$ , som har til opgave at sortere hele  $A$ .

Et kald  $\text{MERGE}(A, p, q, r)$  merger de to sorterede del-arrays/lister  $A[p \dots q]$  og  $A[q + 1 \dots r]$  sammen til  $A[p \dots r]$ .

# Mergesort

Eksempel på kørsel:



# Quicksort

## Mergesort:

- ▶ Del input op i to dele  $X$  og  $Y$  (trivielt)
- ▶ Sorter hver del for sig (rekursion)
- ▶ Merge de to sorterede dele til én sorteret del (reelt arbejde)

Basistilfælde:  $n \leq 1$  (allerede sorteret, gør intet)

## Quicksort:

- ▶ Del input op i to dele  $X$  og  $Y$  så  $X \leq Y$  (reelt arbejde)
- ▶ Sorter hver del for sig (rekursion)
- ▶ Returner  $X$  efterfulgt af  $Y$  (trivielt)

Basistilfælde:  $n \leq 1$  (allerede sorteret, gør intet)

[Hoare, 1960]



# Quicksort

Som pseudo-kode:

```
QUICKSORT( $A, p, r$ )
1  if  $p < r$ 
2      // Partition the subarray around the pivot, which ends up in  $A[q]$ .
3       $q = \text{PARTITION}(A, p, r)$ 
4      QUICKSORT( $A, p, q - 1$ ) // recursively sort the low side
5      QUICKSORT( $A, q + 1, r$ ) // recursively sort the high side
```

Et kald  $\text{QUICKSORT}(A, p, r)$  har til opgave at stille elementerne i  $A[p \dots r]$  i sorteret orden.

Første kald er  $\text{QUICKSORT}(A, 1, n)$ , som har til opgave at sortere hele  $A$ .

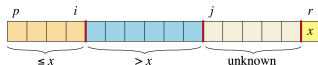
Et kald  $\text{PARTITION}(A, p, r)$  vælger et element  $x \in A$  og opdeler  $A[p \dots r]$  således at:

$$A[q] = x \quad A[p \dots q - 1] \leq x \quad A[q + 1 \dots r] > x$$

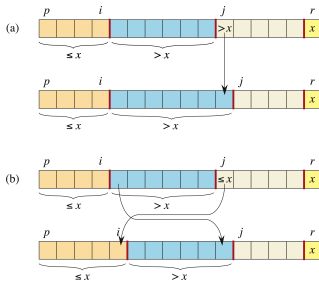
# Partition

Hvordan lave PARTITION?

**Idé:** Vælg et element  $x$  fra input at opdele efter (her sidste element i array-del). Opbyg de to dele under et gennemløb af array ud fra flg. princip:

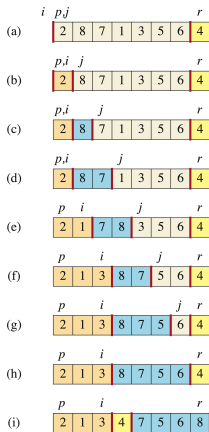


Hvordan tage et skridt under gennemløb?



# Partition

Et eksempel på gennemløb:

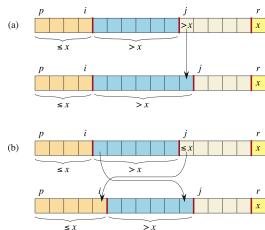


Tid:  $O(n)$ , hvor  $n$  er antal elementer i  $A[p \dots r]$ .

# Partition

Som pseudo-kode:

```
PARTITION( $A, p, r$ )  
1  $x = A[r]$  // the pivot  
2  $i = p - 1$  // highest index into the low side  
3 for  $j = p$  to  $r - 1$  // process each element other than the pivot  
4     if  $A[j] \leq x$  // does this element belong on the low side?  
5          $i = i + 1$  // index of a new slot in the low side  
6         exchange  $A[i]$  with  $A[j]$  // put this element there  
7 exchange  $A[i + 1]$  with  $A[r]$  // pivot goes just to the right of the low side  
8 return  $i + 1$  // new index of the pivot
```



## Quicksort køretid

Hænger på, hvordan partitions gennem rekursionen deler input.

To ekstremer af størrelser på rekursive kald:

- ▶ Helt ubalanceret: 0 og  $n - 1$
- ▶ Helt balanceret:  $\lceil (n - 1)/2 \rceil$  og  $\lfloor (n - 1)/2 \rfloor$
- ▶ Hvis alle partitions er helt balancerede:  $O(n \log n)$  (ca. samme analyse som for Mergesort).
- ▶ Hvis alle partitions er helt ubalancerede:  
 $O(n + (n - 1) + (n - 2) + \dots + 2 + 1) = O(n^2)$ .

Man kan vise at dette er henholdsvis best case og worst case for Quicksort.

## Quicksort køretid

- ▶ I praksis: køretid  $O(n \log n)$  for næsten alle input.
- ▶ Bemærk dog: allerede sorteret input giver køretid  $\Theta(n^2)$ , hvis opdelingsselement  $x$  i Partition vælges som sidste element (hvad bogen gør). Brug **ikke** det valg i praksis.
- ▶ Forslag til mere robuste valg af opdelingsselement  $x$ : enten som midterelementet, som medianen af flere elementer, som et tilfældigt element, eller som medianen af flere tilfældigt valgte elementer.
- ▶ Quicksort er *inplace*: bruger ikke mere plads end input-array'et.
- ▶ Kode er meget effektiv i praksis. En godt implementeret Quicksort er ofte bedste all-round sorteringsalgoritme (og valgt i mange biblioteker, f.eks. Java og C++/STL).

# Heapsort

En **Heap** er:

1. et binært træ
2. med heap-orden
3. og heap-facon
4. udlagt i et array

(Note: “heap” bruges også om et hukommelsesområde brugt til allokering af objekter under et programs udførelse. De to anvendelser er urelaterede.)

[Williams, 1964]

# 1) Binært træ

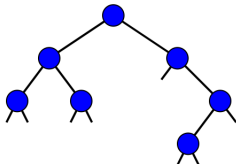
Et binært træ er enten

- ▶ det tomme træ

eller

- ▶ en knude  $v$  samt to undertræer (et højre og et venstre).

Visualisering:

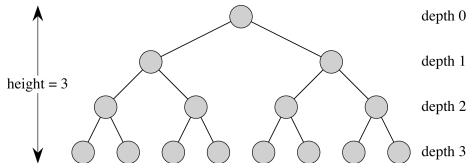


Knuden  $v$  kaldes også **rod** for træet. Hvis  $v$  har et ikke-tomt undertræ, kaldes roden  $u$  af dette for et **barn** af  $v$ , og  $v$  kaldes  $u$ 's **forælder**. Hvis begge  $v$ 's undertræer er tomme, kaldes  $v$  et **blad**. Stregerne mellem børn og forældre kaldes for **kanter**. Forældre/barn-begrebet generaliserer på naturlig måde til **forfader** og **efterkommer**.



# 1) Binært træ

- ▶ Dybde af knude = antal kanter til rod
- ▶ Højde af knude = max antal kanter til blad
- ▶ Højde af træ = højde af dets rod
- ▶ Fuldt (complete) binært træ = træ hvor alle lag er helt fyldte.



Et fuldt binært træ af højde  $h$  har

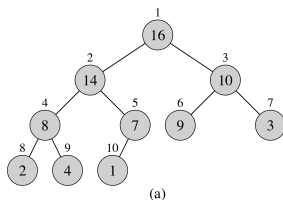
$$1 + 2 + 4 + 8 + \dots + 2^h = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$

knuder (formel A.5 side 1147), heraf  $2^h$  blade.

## 2) Heap-orden

Et binært træ med værdier i alle knuder er **max-heap-ordnet** hvis det for enhver knude  $v$  med barn  $u$  gælder at

$$\text{værdi i } v \geq \text{værdi i } u$$



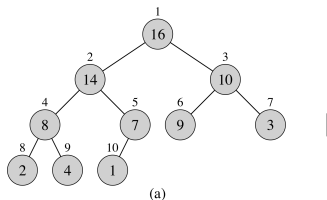
[Bemærk: en ækvivalent definition er, at det for enhver knude  $v$  med **efterkommer**  $u$  gælder, at værdi i  $v \geq$  værdi i  $u$ .]

I et max-heap-ordnet træ vil roden indeholde den største værdi i hele heapen.

Et træ er **min-heapordnet**, hvis ovenstående gælder med  $\leq$  i stedet for  $\geq$ .

### 3) Heapfacon

Et binært træ har heapfacon hvis alle lag i træet er helt fyldte, undtagen det sidste lag, hvor alle knuder findes længst til venstre. (Specielt har et fuldt træ heapfacon).



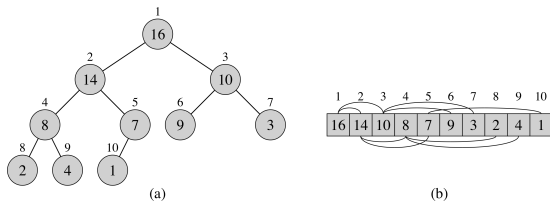
For et træ af heapfacon af højde  $h$  med  $n$  knuder:

$$n > \text{antal knuder i fuldt træ af højde } h - 1 = 2^h - 1$$

$$n > 2^h - 1 \Leftrightarrow n + 1 > 2^h \Leftrightarrow \log_2(n + 1) > h$$

## 4) Heap udlagt i et array

Et binært træ i heapfacon kan naturligt udlægges i et array ved at tildele array-indeksler til knuder ved et top-down, venstre-til-højre gennemløb af træets lag:



Navigering mellem børn og forældre i array-versionen kan udføres ved simple beregninger: Knuden på plads  $i$  har

- ▶ Forælder på plads  $\lfloor i/2 \rfloor$
- ▶ Børn på plads  $2i$  og  $2i + 1$

(Se figur ovenfor. Formelt bevis til øvelsestimer.)

# Operationer på en heap

Vi ønsker at lave følgende operationer på en heap:

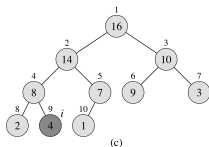
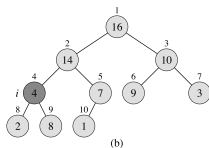
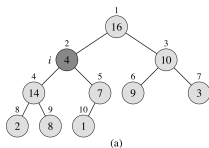
- ▶ **MAX-HEAPIFY**: Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens undertræ til at overholde heap-orden.
- ▶ **BUILD-MAX-HEAP**: Lav  $n$  input elementer (uordnede) til en heap.

[Navnene ovenfor er for en max-heap. For en min-heap findes de samme operationer med “min-” i stedet for “max-” i navnet.]

# Max-Heapify

Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens træ til at overholde heap-orden.

- ▶ Problem: knudens værdi mindre end et eller begge børns værdier.
- ▶ Løsning: byt plads med barnet med den største værdi, kør derefter MAX-HEAPIFY på dette barn.

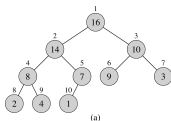


Tid:  $O(\text{højde af knude})$ .

# Max-Heapify

Som pseudo-kode (med indarbejdet check for at man ikke kigger “for langt” i arrayet, dvs. længere end plads  $n$ ):

```
MAX-HEAPIFY( $A, i, n$ )  
   $l = \text{LEFT}(i)$   
   $r = \text{RIGHT}(i)$   
  if  $l \leq n$  and  $A[l] > A[i]$   
     $largest = l$   
  else  $largest = i$   
  if  $r \leq n$  and  $A[r] > A[largest]$   
     $largest = r$   
  if  $largest \neq i$   
    exchange  $A[i]$  with  $A[largest]$   
    MAX-HEAPIFY( $A, largest, n$ )
```

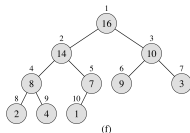
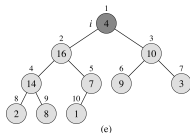
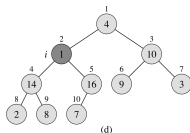
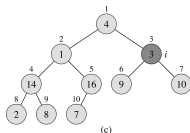
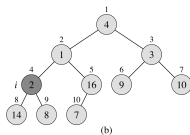
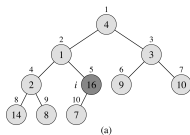


# Build-Heap

Lav  $n$  input elementer (uordnede) til en heap.

- ▶ Ide: arranger elementerne i heap-facon, bring derefter træet i heap-orden nedefra og op.
- ▶ Observation: et træ af størrelse én overholder altid heaporden.

A [ 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 ]

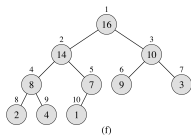
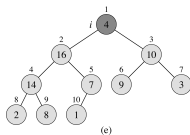
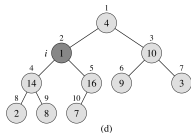
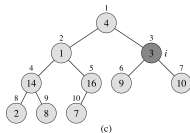
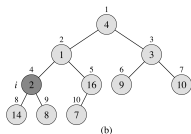
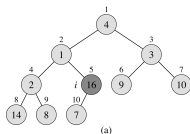




# Build-Heap

A 

|   |   |   |   |    |   |    |    |   |   |
|---|---|---|---|----|---|----|----|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|----|---|----|----|---|---|



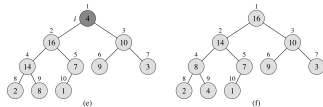
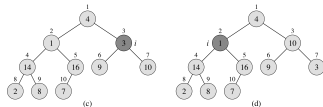
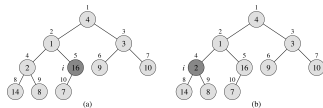
Tid:  $O(n \log_2 n)$  klart. Bedre analyse giver  $O(n)$ .

# Build-Heap

Som pseudo-kode:

**BUILD-MAX-HEAP**( $A, n$ )  
**for**  $i = \lfloor n/2 \rfloor$  **downto** 1  
  **MAX-HEAPIFY**( $A, i, n$ )

A [ 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 ]



# Heapsort

En form for selectionsort hvor der bruges en heap til hele tiden at udtage det største tilbageværende element:

byg en heap

**gentag** til heap er tom:

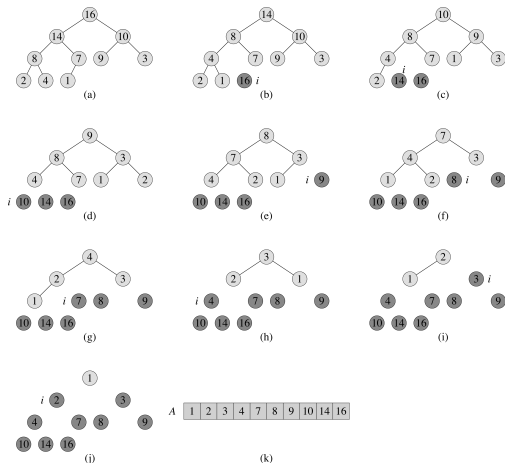
udtag rod (største element i heapen)

sæt sidste element op som ny rod

genskab heap-struktur ved MAX-HEAPIFY på ny rod.

# Heapsort

Eksempel:



# Heapsort

Som pseudo-kode:

```
HEAPSORT( $A, n$ )  
  BUILD-MAX-HEAP( $A, n$ )  
  for  $i = n$  downto 2  
    exchange  $A[1]$  with  $A[i]$   
    MAX-HEAPIFY( $A, 1, i - 1$ )
```

Tid:  $O(n) + O(n \log n) = O(n \log n)$

## Tre $n \log n$ sorteringsalgoritmer

|           | Worstcase | Inplace |
|-----------|-----------|---------|
| QuickSort |           | ✓       |
| MergeSort | ✓         |         |
| HeapSort  | ✓         | ✓       |

Heapsort kører dog langsommere end Mergesort og Quicksort pga. ineffektiv brug af hukommelse (random access).

**Introsort** [Musser, 1997]: brug Quicksort, men skift under rekursionen til heapsort hvis rekursionen bliver for dyb. Dette giver en inplace, worst case  $O(n \log n)$  algoritme, med god køretid i praksis (dette er sorteringsalgoritmen i standardbiblioteket STL for C++).