

# **Grundlæggende Algoritmer og Datastrukturer**

**Elementære Datastrukturer [CLRS, kapitel 10]**

# [CLRS, Del 3] : Datastrukturer

Oprethold en struktur for en  
**dynamisk** mængde data

# Abstrakte Datastrukturer for Mængder

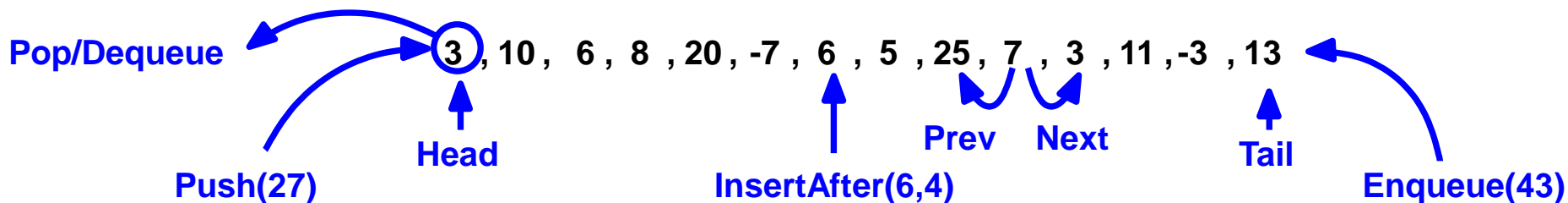
*-Min-prioritetskø  
-Max-prioritetskø  
- Ordbog*

Forespørgsel	Minimum( $S$ )	pointer til element	●		
	Maximum( $S$ )	pointer til element		●	
	Search( $S, x$ )	pointer til element			●
	Member( $S, x$ )	TRUE eller FALSE			
	Successor( $S, x$ )	pointer til element			
	Predecessor( $S, x$ )	pointer til element			
Opdateringer	Insert( $S, x$ )	pointer til element	●	●	●
	Delete( $S, x$ )	-			●
	DeleteMin( $S$ )	element	●		
	DeleteMax( $S$ )	element		●	
	Join( $S_1, S_2$ )	mængde $S$			
	Split( $S, x$ )	mængder $S_1$ og $S_2$			

# Abstrakte Datastrukturer for Lister

-Stak -Kø

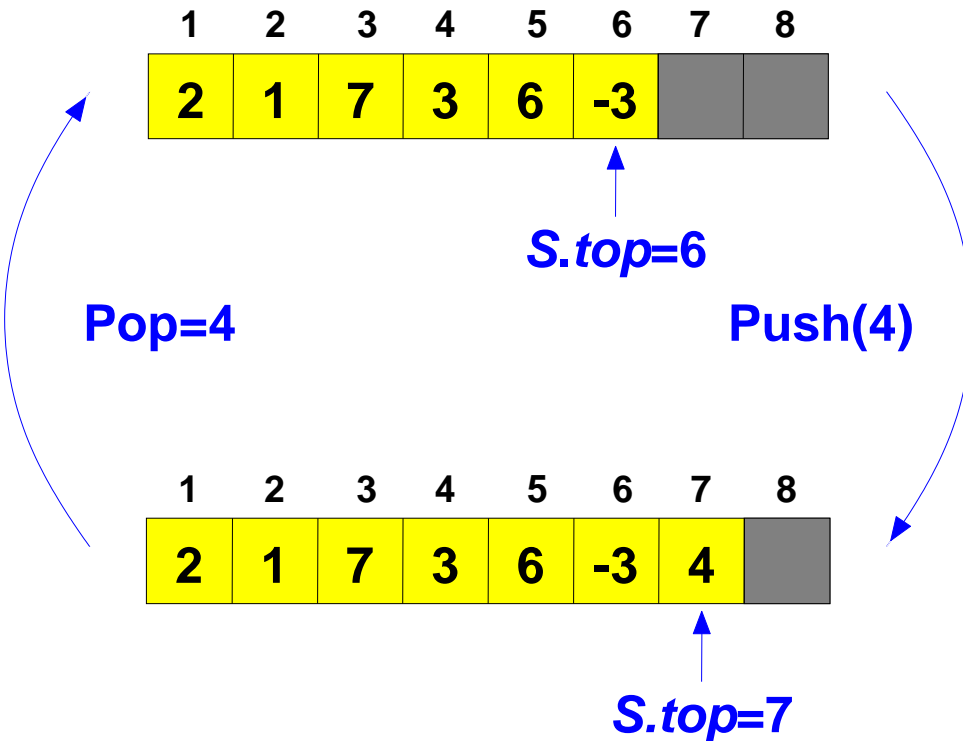
Forespørgsel	Empty( $S$ )	TRUE eller FALSE	●	●
	Head( $S$ ), Tail( $S$ )	pointer til element		
	Next( $S, x$ ), Prev( $S, x$ )	pointer til element		
	Search( $S, x$ )	pointer til element		
Opdateringer	Push( $S, x$ )	-	●	
	Pop/Dequeue( $S$ )	element	●	●
	Enqueue( $S, x$ )	-		●
	Delete( $S, x$ )	Element		
	InsertAfter( $S, x, y$ )	pointer til element		





**Stak**

# Stak : Array Implementation



STACK-EMPTY( $S$ )

```
1  if  $S.top == 0$ 
2      return TRUE
3  else return FALSE
```

PUSH( $S, x$ )

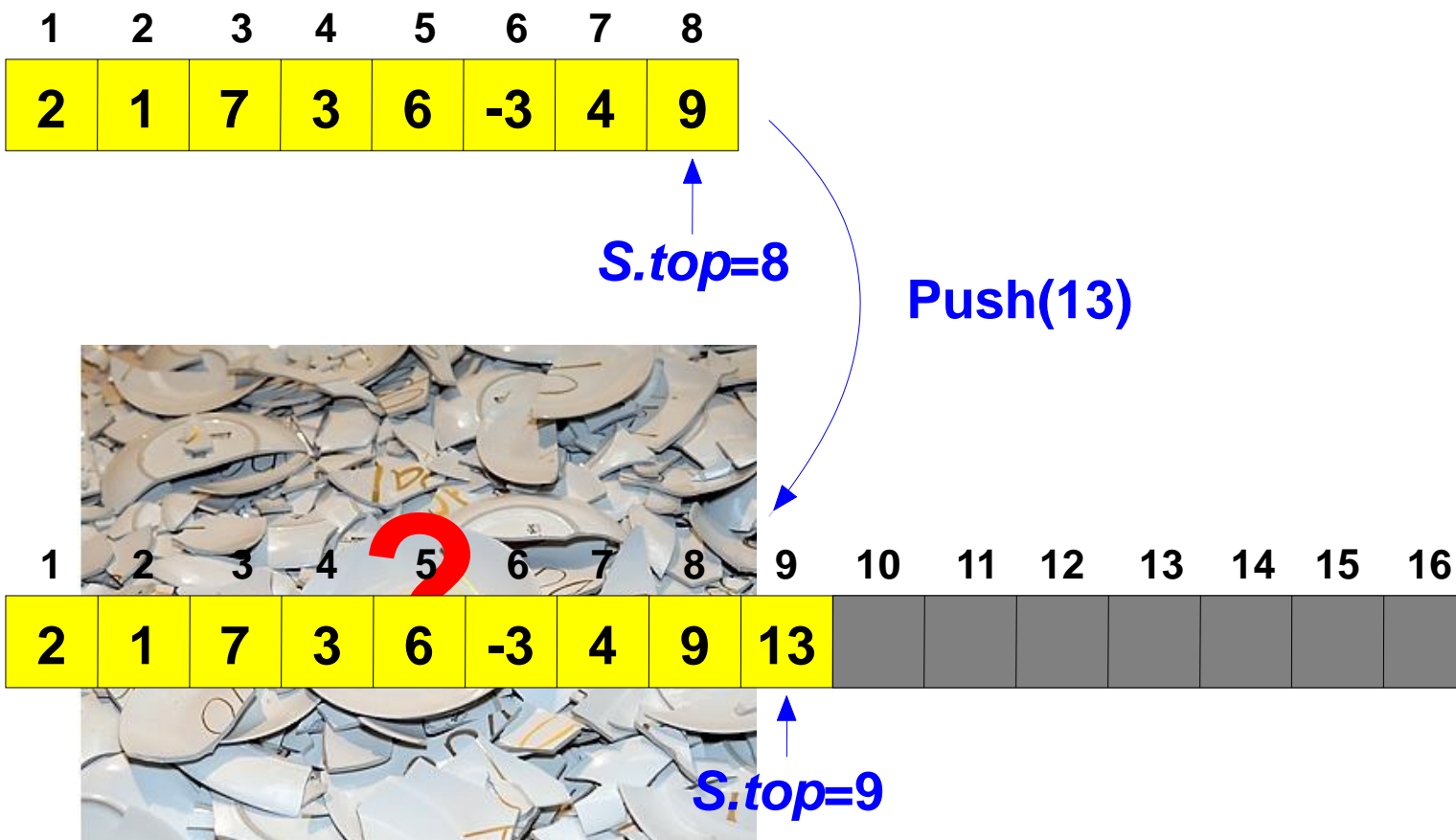
```
1   $S.top = S.top + 1$ 
2   $S[S.top] = x$ 
```

POP( $S$ )

```
1  if STACK-EMPTY( $S$ )
2      error "underflow"
3  else  $S.top = S.top - 1$ 
4      return  $S[S.top + 1]$ 
```

**Stack-Empty, Push, Pop :  $O(1)$  tid**

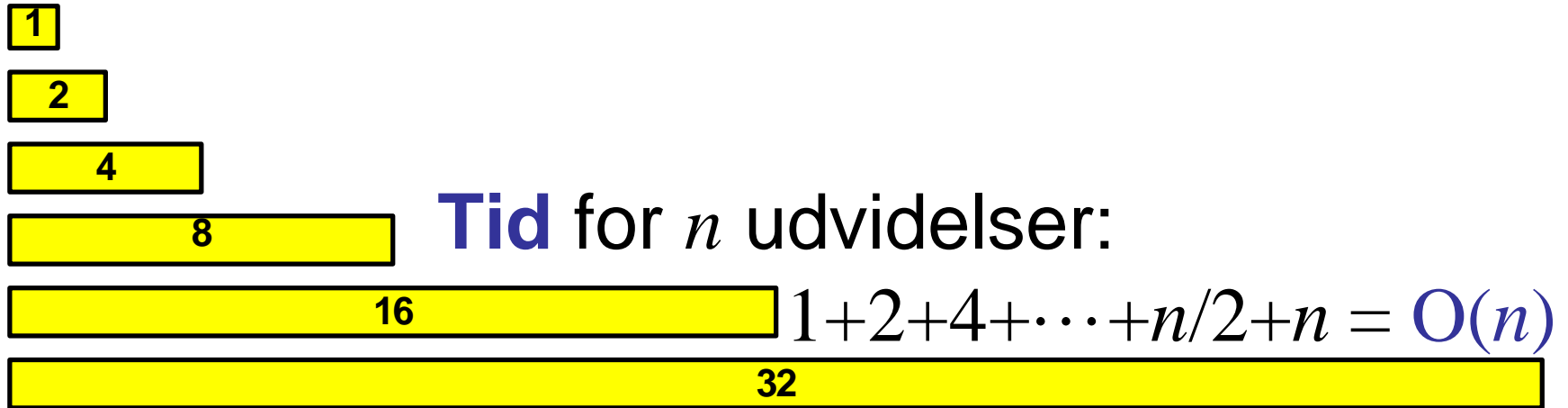
# Stak : Overløb



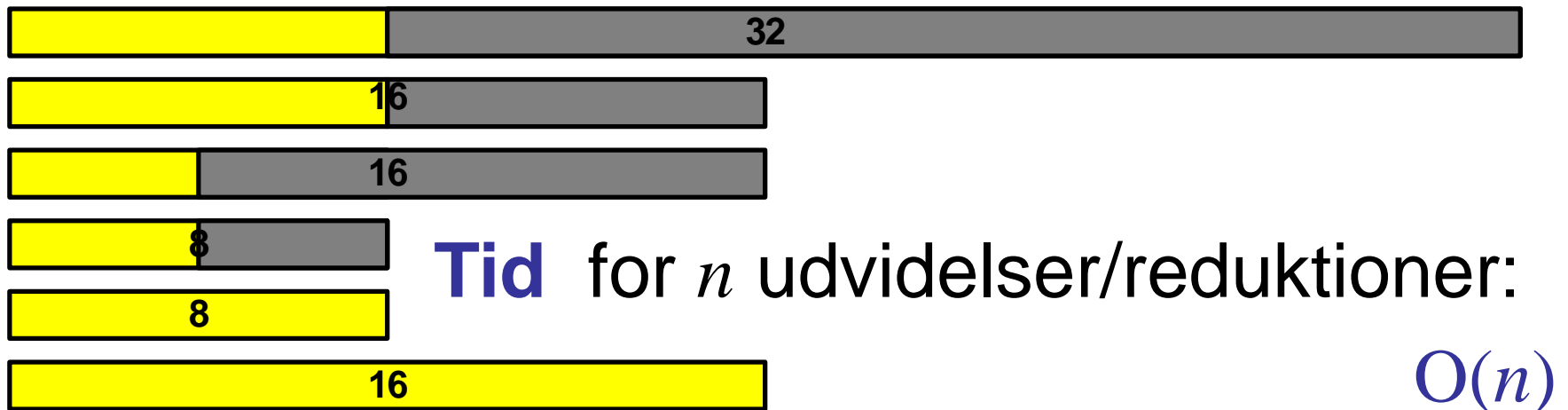
Array fordobling :  $O(n)$  tid

# Array Fordobling

**Fordoble** arrayet når det er fuld



**Halver** arrayet når det er  $<1/4$  fyldt





# Array Fordobling + Halvering

– en generel teknik

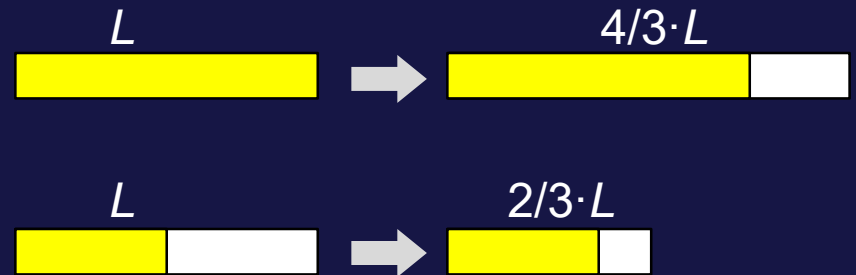
**Tid** for  $n$  udvidelser/reduktioner er  $O(n)$

**Plads**  $\leq 4 \cdot$  aktuelle antal elementer

**Array implementation af Stak:**  
 $n$  push og pop operationer tager  $O(n)$  tid

# 4/3 Forøgelse

Ved overløb lad den nye længde være  $4/3$  af den oprindelige, og når arrayet bliver halv fuldt reducer det til  $2/3$  længde.



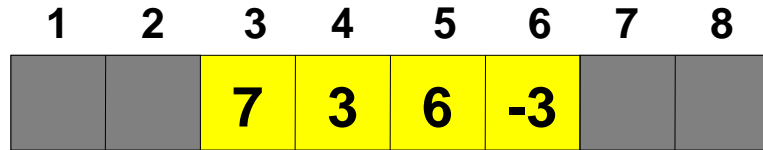
Hvor langt kan arrayet maksimalt være når det gemmer  $n$  elementer ?

- a)  $3/2 \cdot n$
- b)  $4/3 \cdot n$
- c)  $2 \cdot n$
- d)  $8/3 \cdot n$
- e) ved ikke



**Kø**

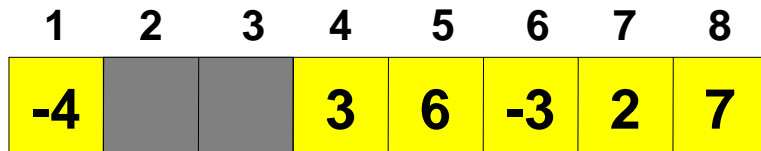
# Kø : Array Implementation



$Q.head=3$

$Q.tail=7$

Enqueue(2)  
Enqueue(7)  
Enqueue(-4)  
Dequeue = 7



$Q.tail=2$

$Q.head=4$

ENQUEUE( $Q, x$ )

```
1  $Q[Q.tail] = x$   
2 if  $Q.tail == Q.length$   
3      $Q.tail = 1$   
4 else  $Q.tail = Q.tail + 1$ 
```

DEQUEUE( $Q$ )

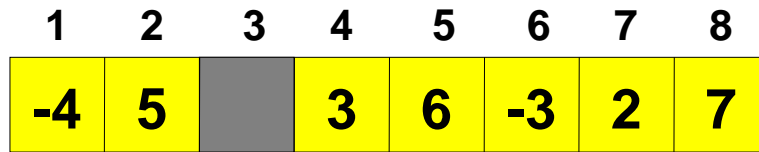
```
1  $x = Q[Q.head]$   
2 if  $Q.head == Q.length$   
3      $Q.head = 1$   
4 else  $Q.head = Q.head + 1$   
5 return  $x$ 
```

Enqueue, dequeue :  $O(1)$  tid

Maximal kapacitet af en kø  
implementeret i et array af længde  $n$  ?

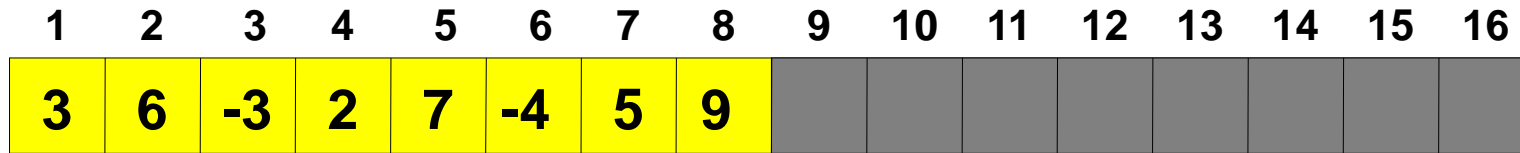
- a)  $n$
- b)  $n - 1$
- c)  $n - 2$
- d)  $n / 2$
- e) ved ikke

# Kø : Array Implementation



$Q.tail=3$       $Q.head=4$

Enqueue(9)



$Q.head=1$

$Q.tail=9$

Empty :  $Q.tail=Q.head$  ?

**Overløb** : array fordobling/  
halvering

Array implementation af Kø:

$n$  enqueue og dequeue operationer tager  $O(n)$  tid

# Arrays (med Fordobling/Halvering)

<b>Stak</b>	<b>Push(<math>S, x</math>)</b>	$O(1)^*$
	<b>Pop(<math>S</math>)</b>	$O(1)^*$
<b>Kø</b>	<b>Enqueue(<math>S, x</math>)</b>	$O(1)^*$
	<b>Dequeue(<math>S</math>)</b>	$O(1)^*$

\* Worst-case uden fordobling/halvering  
Amortiseret ([CLRS, Kap. 17]) med fordobling/halvering

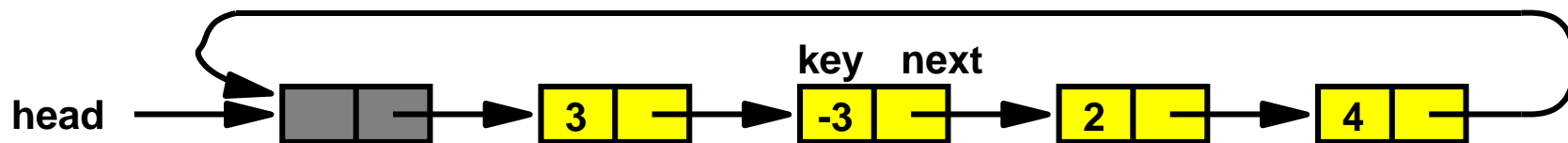
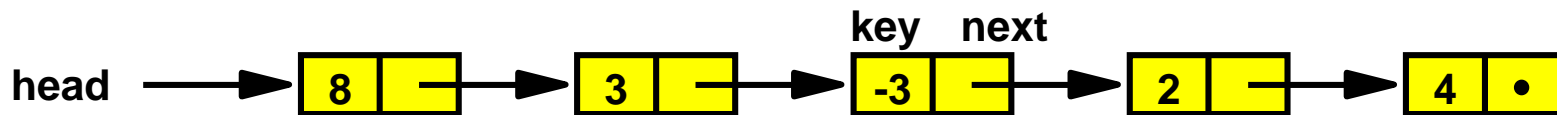


**Kædede lister**

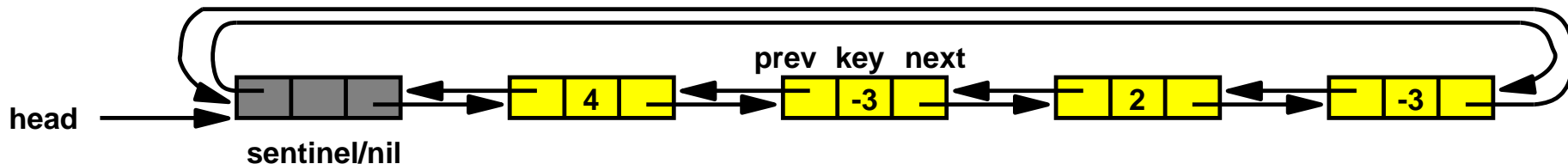
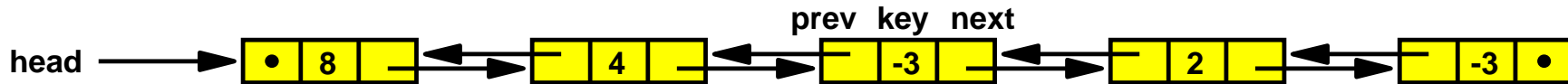


# Kædede Lister

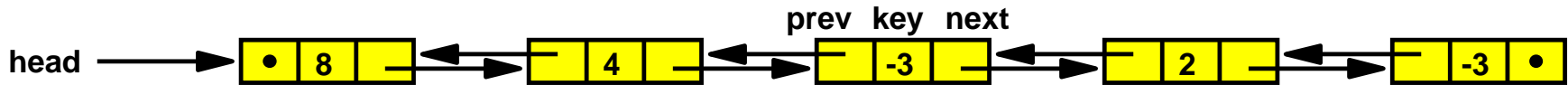
## Enkelt kædede (ikke-cyklisk og cyklisk)



## Dobbelt kædede (ikke-cyklisk og cyklisk)



# Dobbelt Kædede Lister



LIST-SEARCH( $L, k$ )

```
1  $x = L.head$ 
2 while  $x \neq \text{NIL}$  and  $x.key \neq k$ 
3      $x = x.next$ 
4 return  $x$ 
```

LIST-INSERT( $L, x$ )

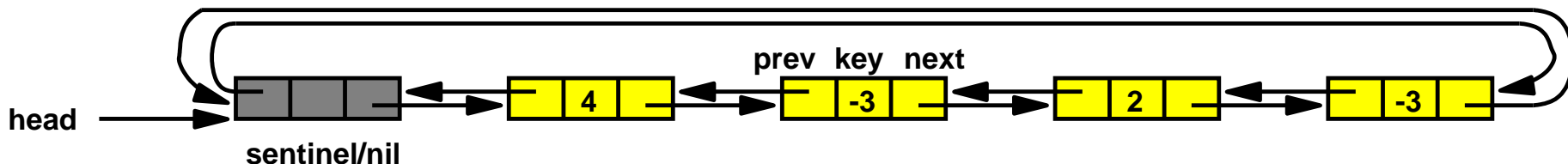
```
1  $x.next = L.head$ 
2 if  $L.head \neq \text{NIL}$ 
3      $L.head.prev = x$ 
4  $L.head = x$ 
5  $x.prev = \text{NIL}$ 
```

LIST-DELETE( $L, x$ )

```
1 if  $x.prev \neq \text{NIL}$ 
2      $x.prev.next = x.next$ 
3 else  $L.head = x.next$ 
4 if  $x.next \neq \text{NIL}$ 
5      $x.next.prev = x.prev$ 
```

List-Search	$O(n)$
List-Insert	$O(1)$
List-Delete	$O(1)$

# Dobbelt Kædede Cykliske Lister



LIST-SEARCH'( $L, k$ )

- 1  $x = L.nil.next$
- 2 **while**  $x \neq L.nil$  and  $x.key \neq k$
- 3      $x = x.next$
- 4 **return**  $x$

LIST-INSERT'( $L, x$ )

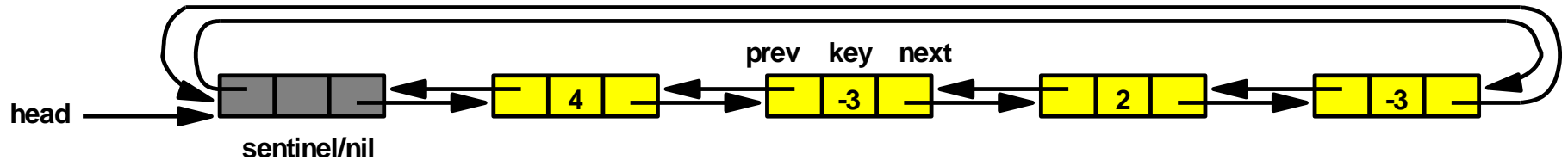
- 1  $x.next = L.nil.next$
- 2  $L.nil.next.prev = x$
- 3  $L.nil.next = x$
- 4  $x.prev = L.nil$

LIST-DELETE'( $L, x$ )

- 1  $x.prev.next = x.next$
- 2  $x.next.prev = x.prev$

<b>List-Search'</b>	$O(n)$
<b>List-Insert'</b>	$O(1)$
<b>List-Delete'</b>	$O(1)$

# Dobbelt Kædede Cykliske Lister



<b>Stak</b>	<b>Push(<math>S, x</math>)</b>	$O(1)$
	<b>Pop(<math>S</math>)</b>	$O(1)$
<b>Kø</b>	<b>Enqueue(<math>S, x</math>)</b>	$O(1)$
	<b>Dequeue(<math>S</math>)</b>	$O(1)$

## Dancing Links

*Donald E. Knuth, Stanford University*

My purpose is to discuss an extremely simple technique that deserves to be better known. Suppose  $x$  points to an element of a doubly linked list; let  $L[x]$  and  $R[x]$  point to the predecessor and successor of that element. Then the operations

$$L[R[x]] \leftarrow L[x], \quad R[L[x]] \leftarrow R[x] \quad (1)$$

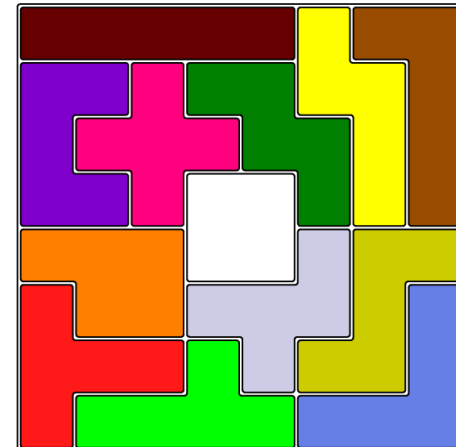
remove  $x$  from the list; every programmer knows this. But comparatively few programmers have realized that the subsequent operations

$$L[R[x]] \leftarrow x, \quad R[L[x]] \leftarrow x \quad (2)$$

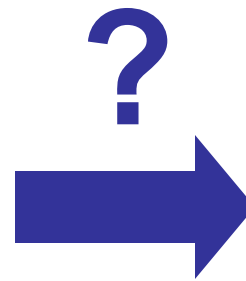
will put  $x$  back into the list again.



Donald E. Knuth (1938-)



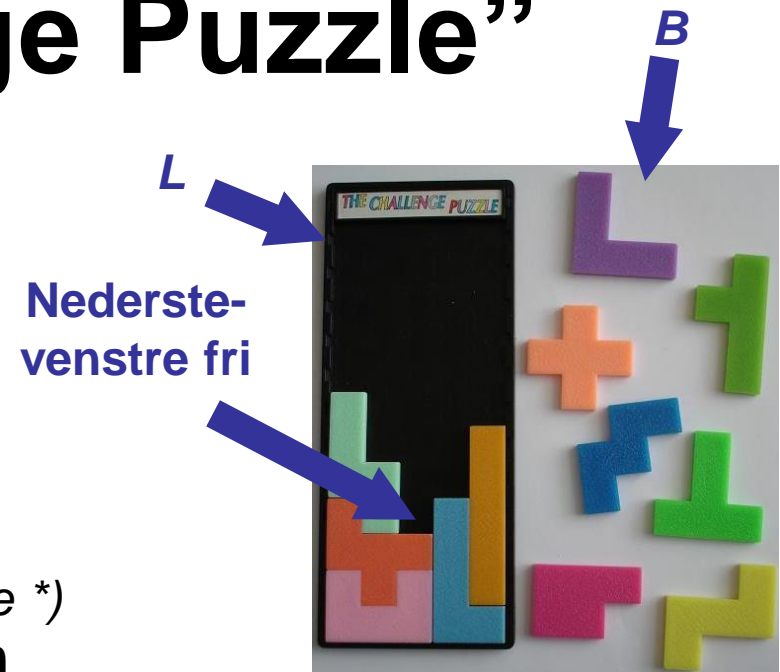
# “The Challenge Puzzle”



# ”The Challenge Puzzle”

$L :=$  Tomt bræt  
 $B :=$  Alle brikker  
Solve( $L, B$ )

```
procedure Solve(Delløsning  $L$ , Brikker  $B$ )  
  for alle  $b$  i  $B$   
    for alle orienteringer af  $b$  (* max 8 forskellige *)  
      if  $b$  kan placeres i nederste venstre fri then  
        fjern  $b$  fra  $B$   
        indsæt  $b$  i  $L$   
        if  $|B|=0$  then  
          rapporter  $L$  er en løsning  
        else  
          Solve( $L, B$ )  
      fi  
    slet  $b$  fra  $L$   
  genindsæt  $b$  i  $B$   
fi
```



Før



Efter

# ”The Challenge Puzzle”

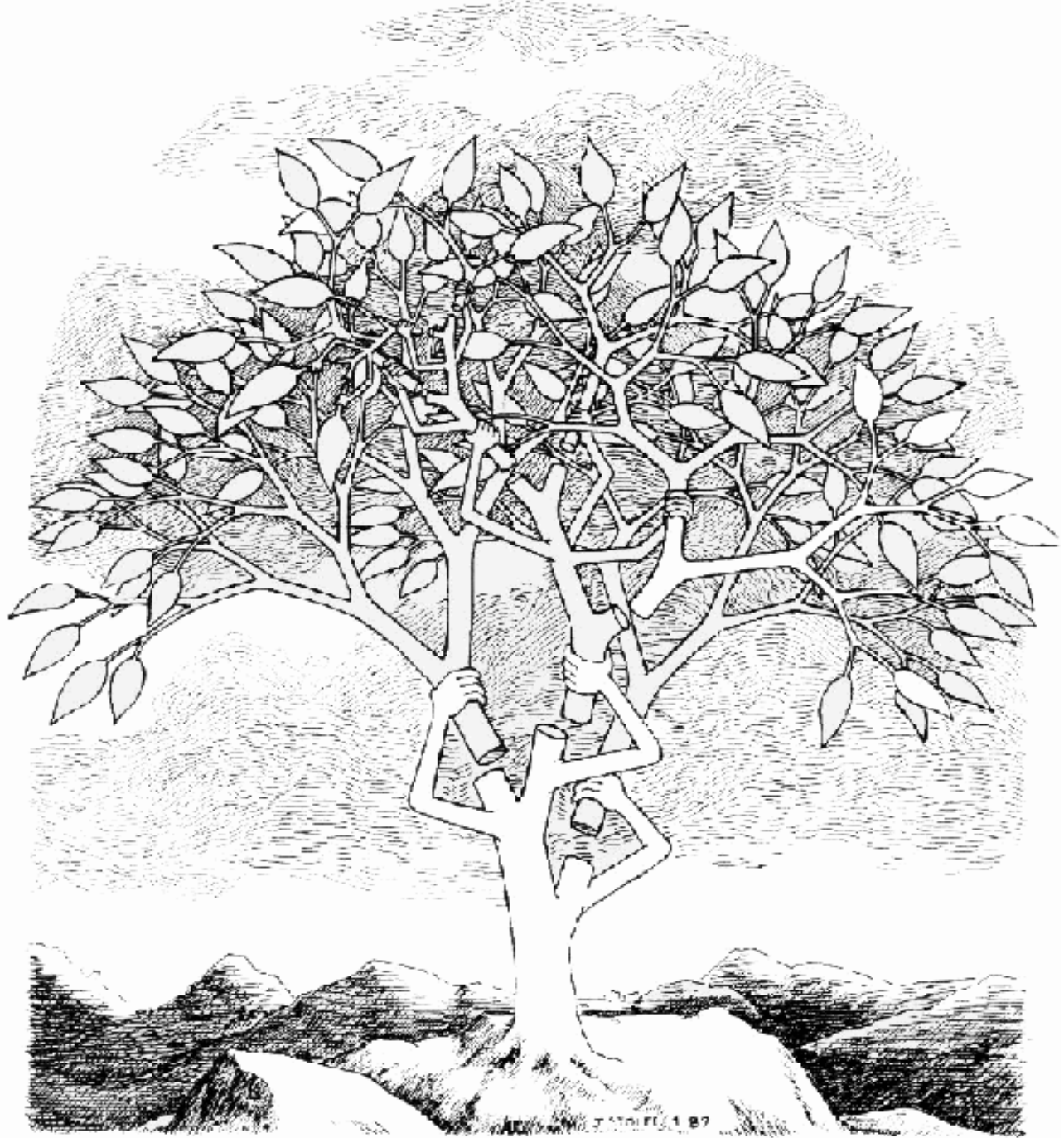


**4.040 løsninger**

**Solve placerer**

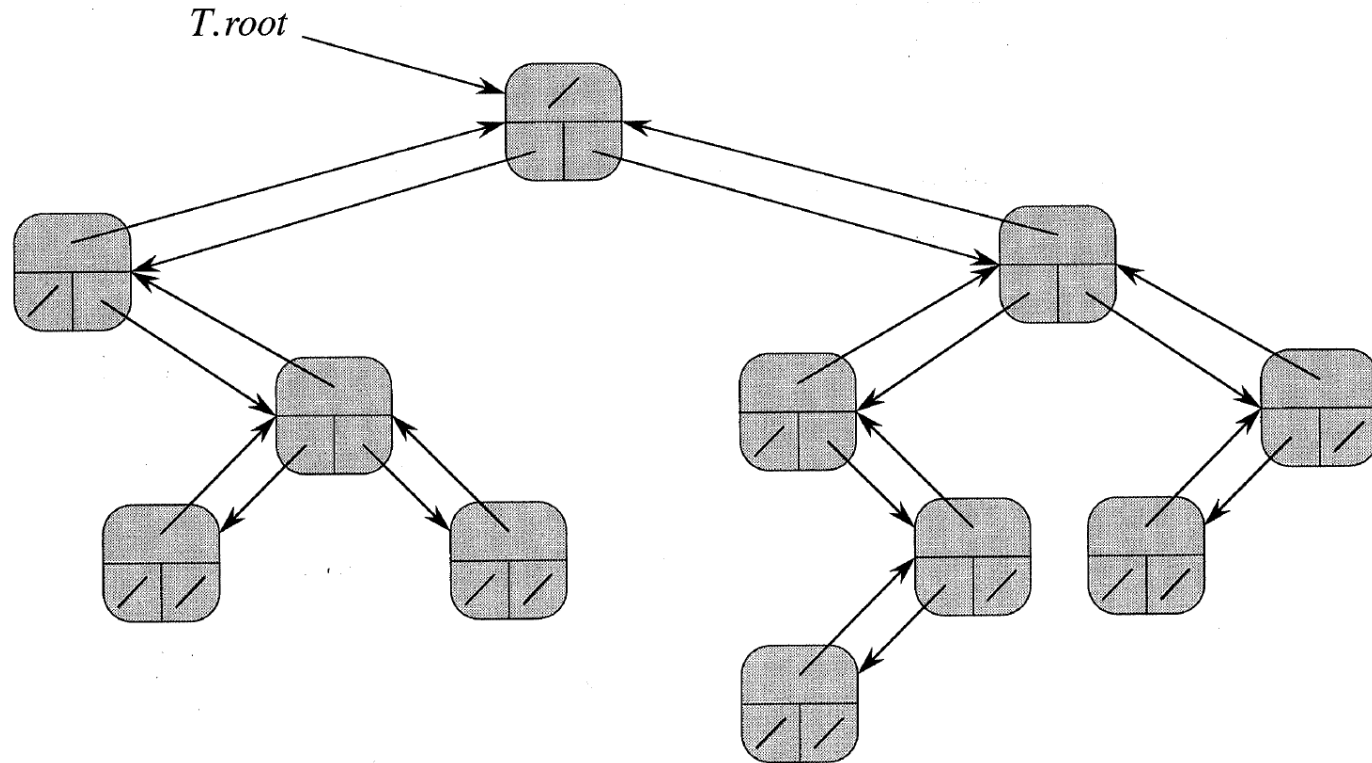
**8.387.259 brikker**





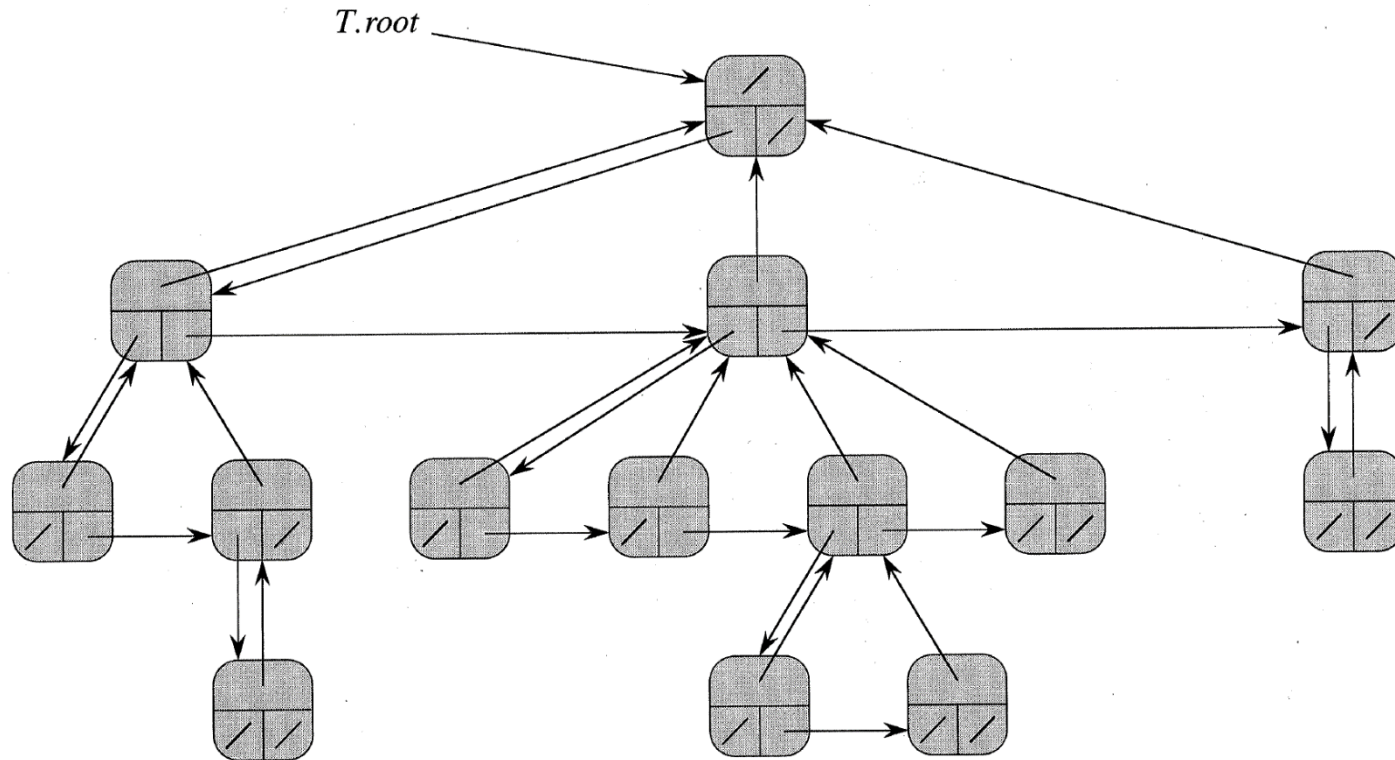
(Jorge Stolfi)

# Binær Træ Repræsentation



Felter: **Left, right, parent**

# Træ Repræsentation



Felter: **Left-child, right-sibling, parent**

Hvor lang tid tager det at tilgå  
det  $i$ -te barn til en knude ?

- a)  $O(1)$
- b)  $O(i)$
- c)  $O(\log i)$
- d) ved ikke