Algoritmer og Datastrukturer 1

Gerth Stølting Brodal

Elementære Datastrukturer [CLRS, kapitel 10]

AARHUS UNIVERSITET

[CLRS, Del 3] : Datastrukturer

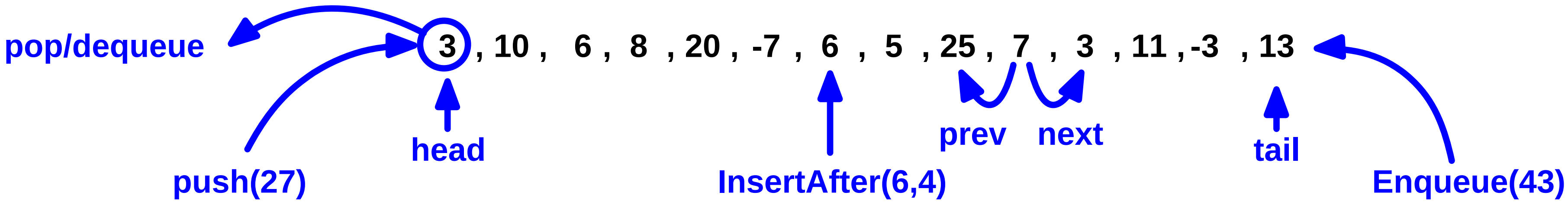
Oprethold en struktur for en **dynamisk** mængde data

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			

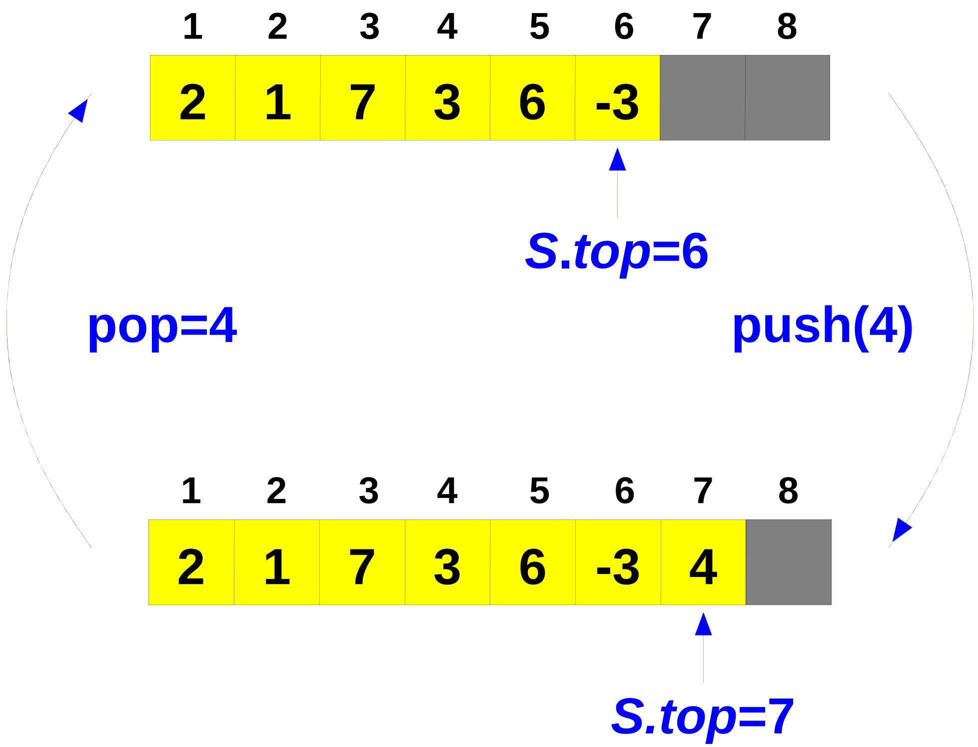
Abstrakte Datastrukturer for Lister

			-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)
    if S.top == 0
        return TRUE
    else return FALSE

PUSH(S, x)
    S.top = S.top + 1
    S[S.top] = x

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

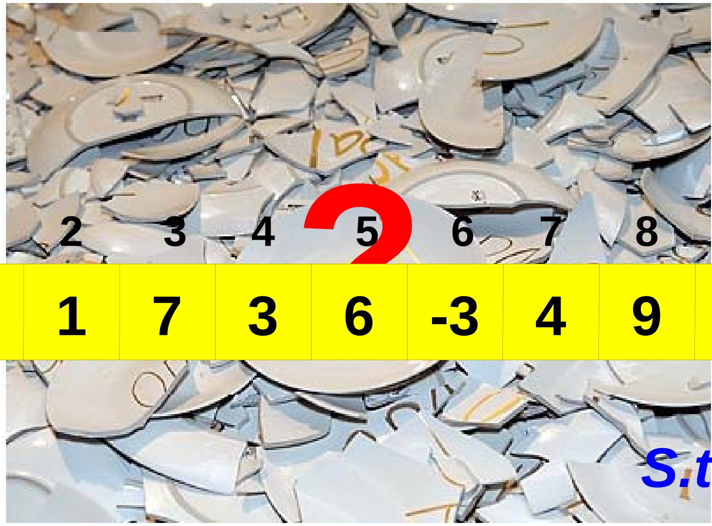
Stack-Empty, Push, Pop : O(1) tid

Stak : Overløb

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

S.top=8

Push(13)



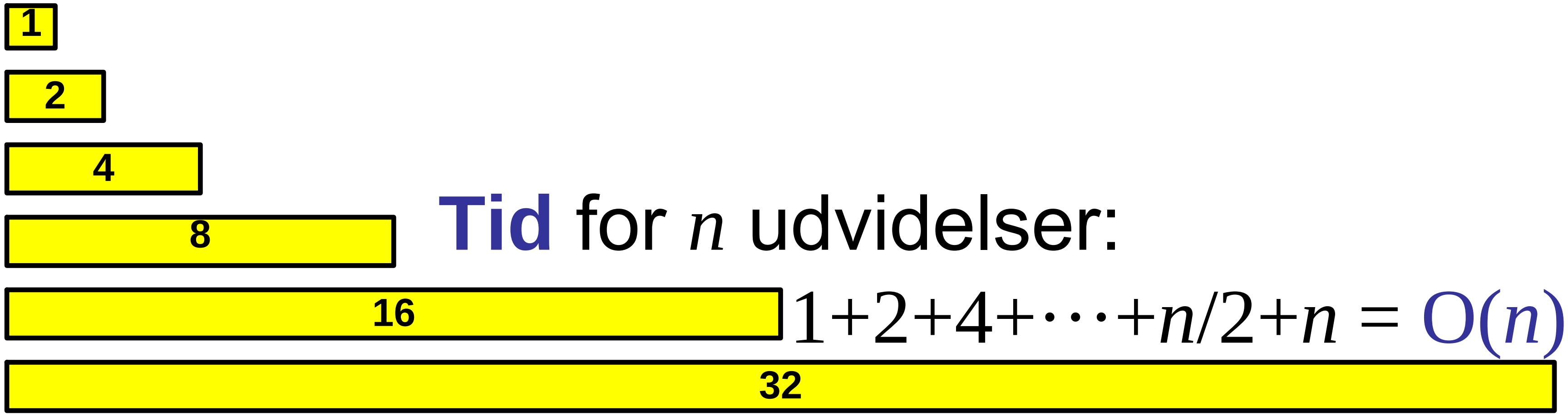
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	1	7	3	6	-3	4	9	13							

S.top=9

Array fordobling : O(n) tid

Array Fordobling

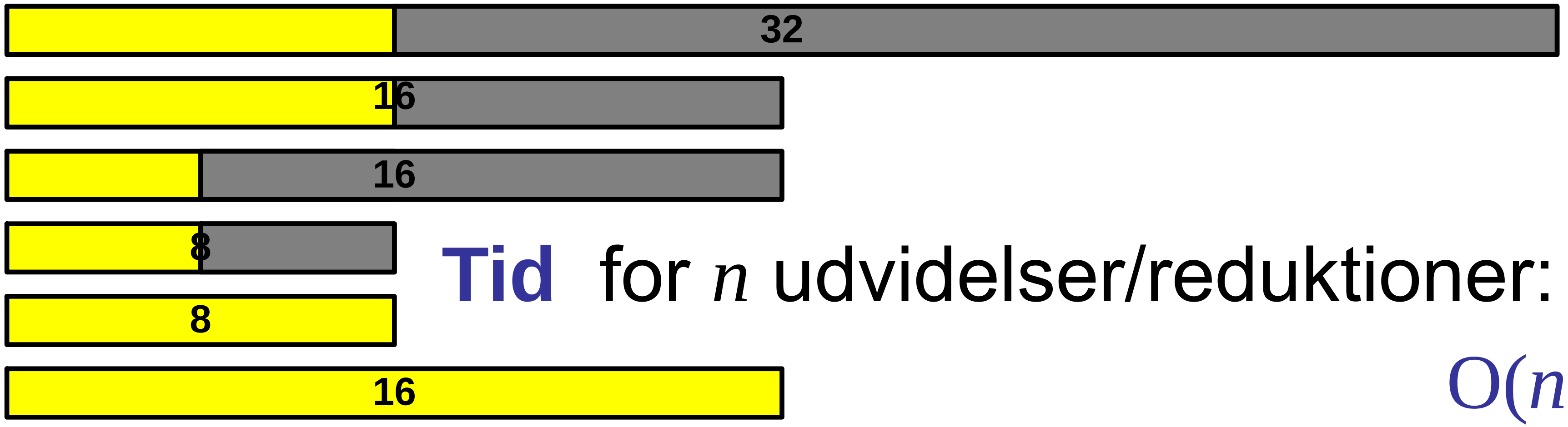
Fordoble arrayet når det er fuld



Tid for n udvidelser:

$$1+2+4+\cdots+n/2+n = O(n)$$

Halver arrayet når det er <1/4 fyldt



Tid for n udvidelser/reduktioner:

$$O(n)$$

Array Fordobling + Halvering

– en generel teknik

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

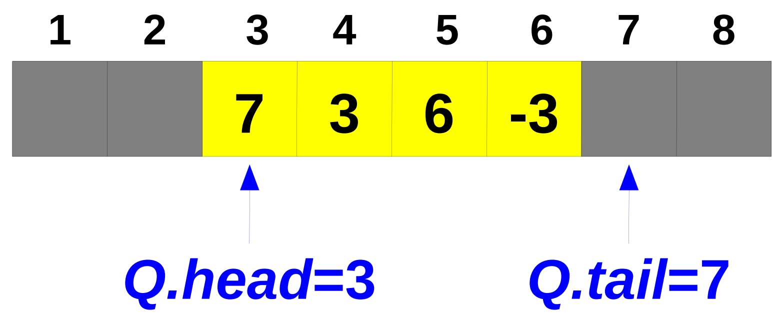
Plads $\leq$ 4 · aktuelle antal elementer

Array implementation af Stak:

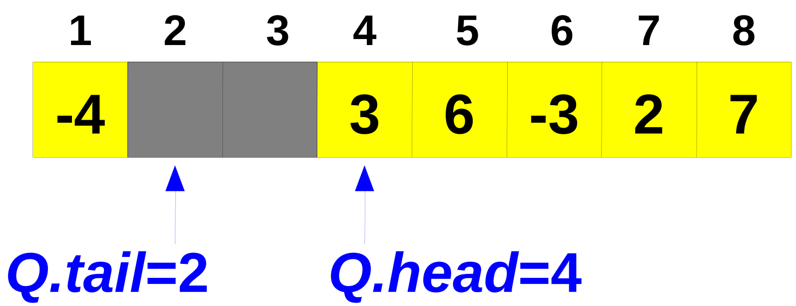
n **push og pop operationer tager** $O(n)$ **tid**

Kø

Kø : Array Implementation



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

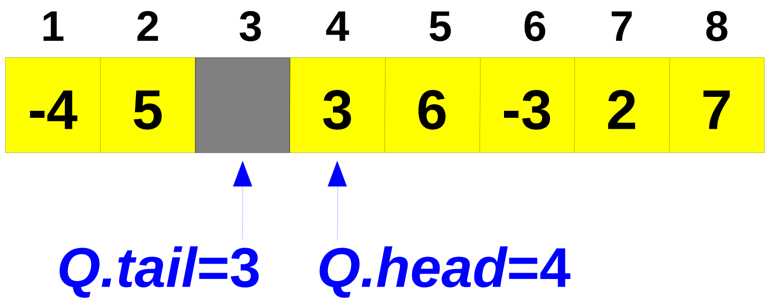



```
ENQUEUE(Q, x)
Q[Q.tail] = x
if Q.tail == Q.length
        Q.tail = 1
else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
x = Q[Q.head]
if Q.head == Q.length
        Q.head = 1
else Q.head = Q.head + 1
return x
```

Enqueue, dequeue : $O(1)$ **tid**

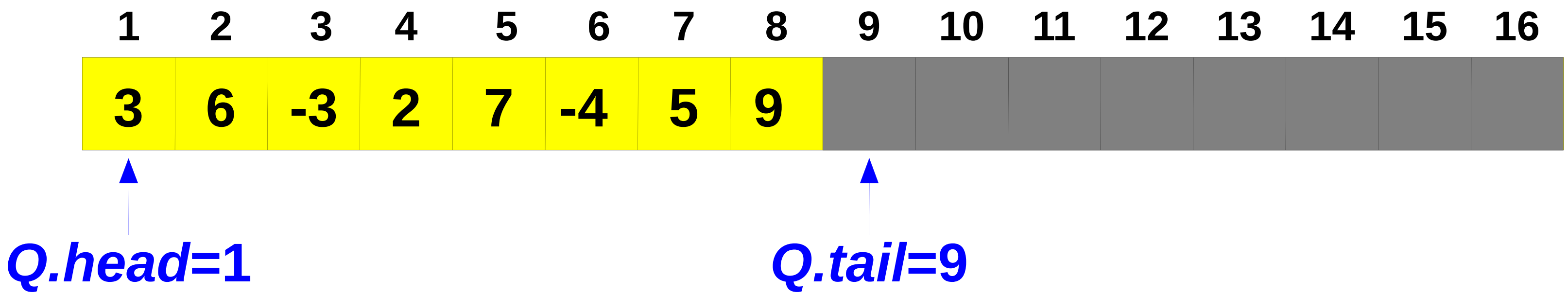
Kø : Array Implementation



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

Overløb : array fordobling/ halvering

Enqueue(9)



Array implementation af Kø:
n enqueue og dequeue operationer tager $O(n)$ tid

Arrays (med Fordobling/Halvering)

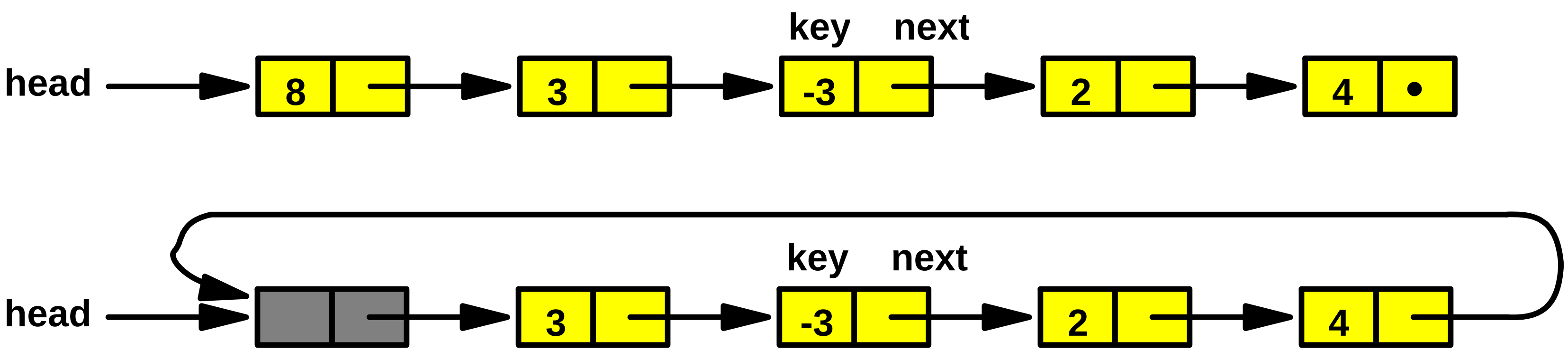
Stak	**Push(*S*, *x*)**	O(1)*
	Pop(*S*)	O(1)*
Kø	**Enqueue(*S*, *x*)**	O(1)*
	Dequeue(*S*)	O(1)*

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

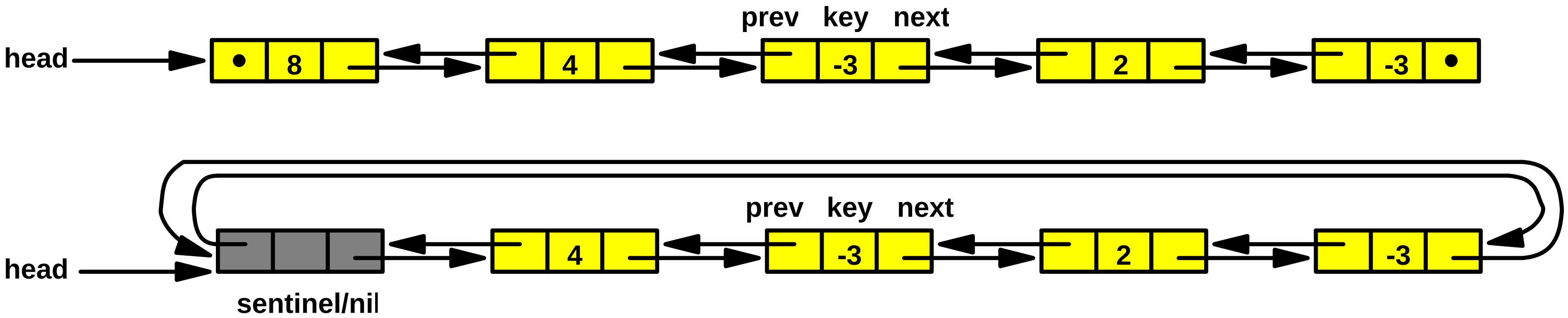
Kædede lister

Kædede Lister

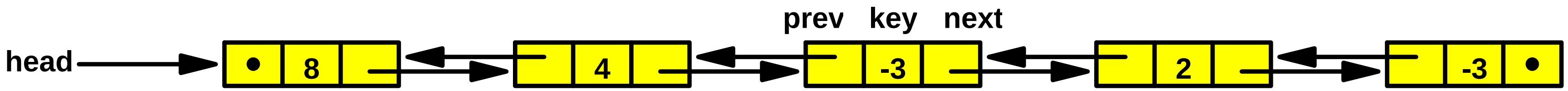
Enkelt kædede (ikke-cyklisk og cyklisk)



Dobbelt kædede (ikke-cyklisk og cyklisk)



Dobbelt Kædede Lister



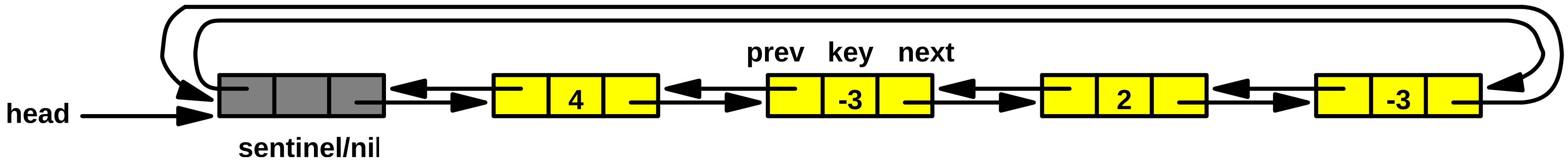

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

```
LIST-DELETE(L, x)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    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)

```
x = L.nil.next
while x ≠ L.nil and x.key ≠ k
    x = x.next
return x
```

LIST-INSERT′(L, x)

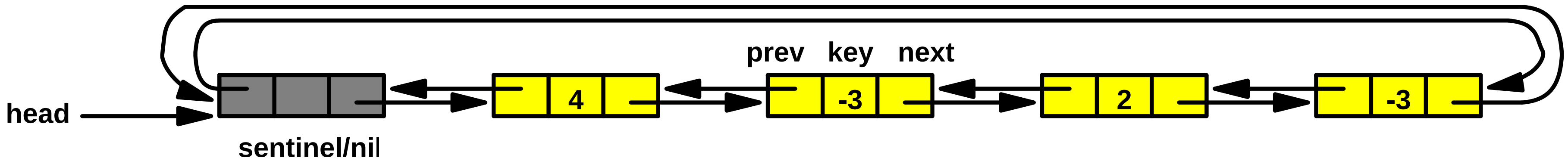
```
x.next = L.nil.next
L.nil.next.prev = x
L.nil.next = x
x.prev = L.nil
```

LIST-DELETE′(L, x)

```
x.prev.next = x.next
x.next.prev = x.prev
```

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

Dobbelt Kædede Cykliske Lister



Stak	**Push(S, x)**	O(1)
	Pop(S)	O(1)
Kø	**Enqueue(S, x)**	O(1)
	Dequeue(S)	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

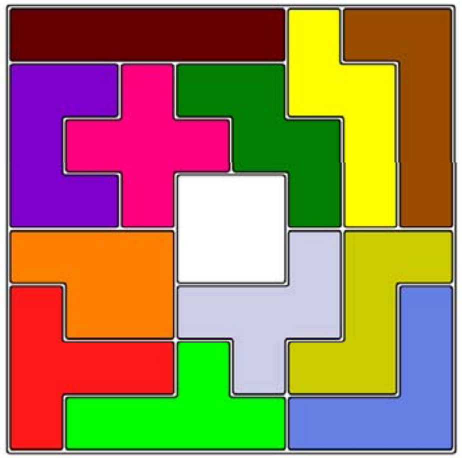
$$L\big[R[x]\big] \leftarrow L[x], \qquad R\big[L[x]\big] \leftarrow R[x] \tag{1}$$

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

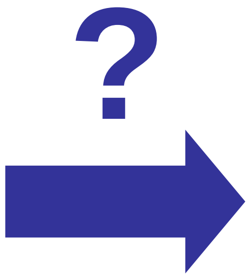
$$L\big[R[x]\big] \leftarrow x, \qquad R\big[L[x]\big] \leftarrow x \tag{2}$$

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 fa 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
```

B

L

Nederste-
venstre fri

Før

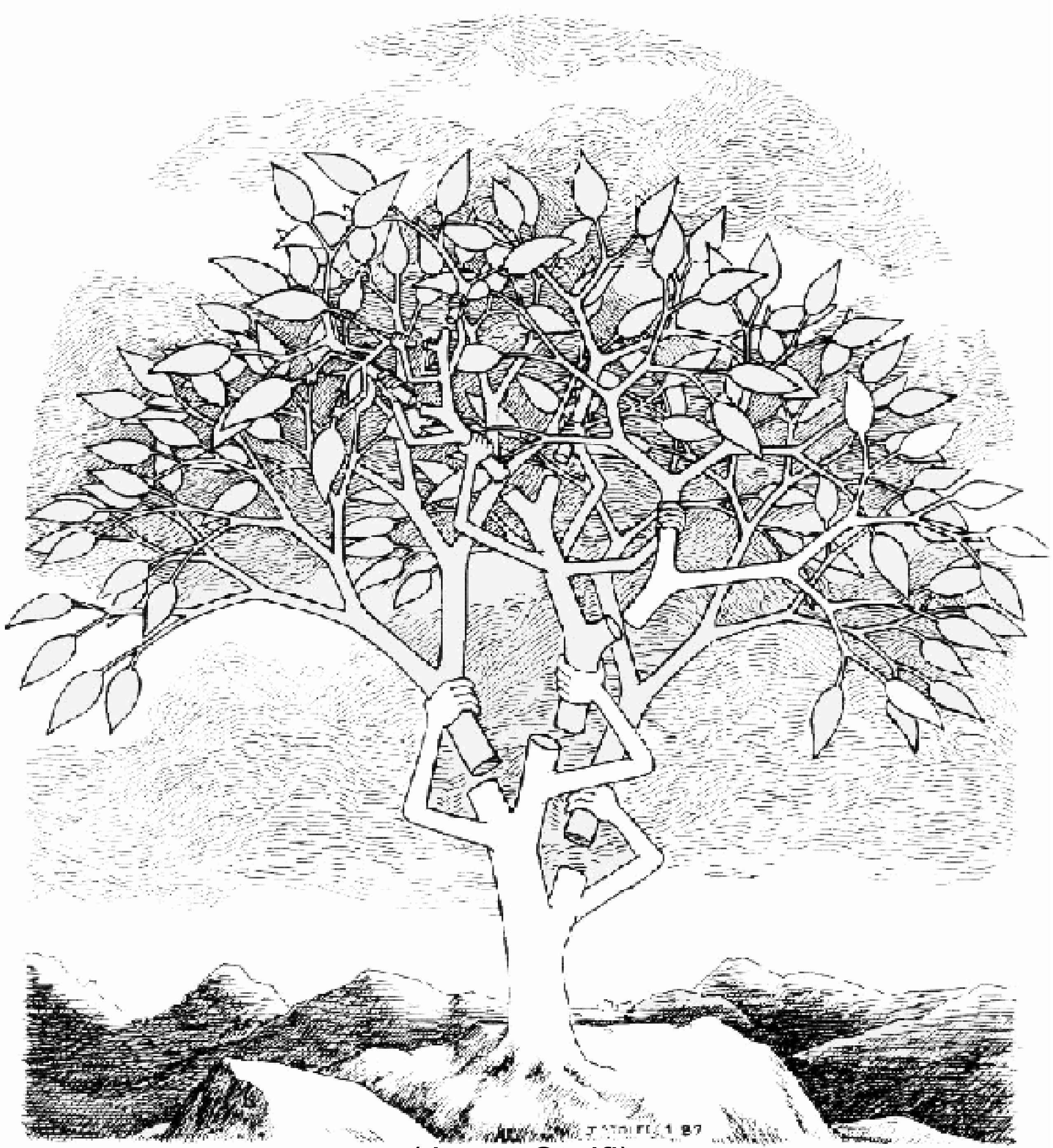
Efter

”The Challenge Puzzle”

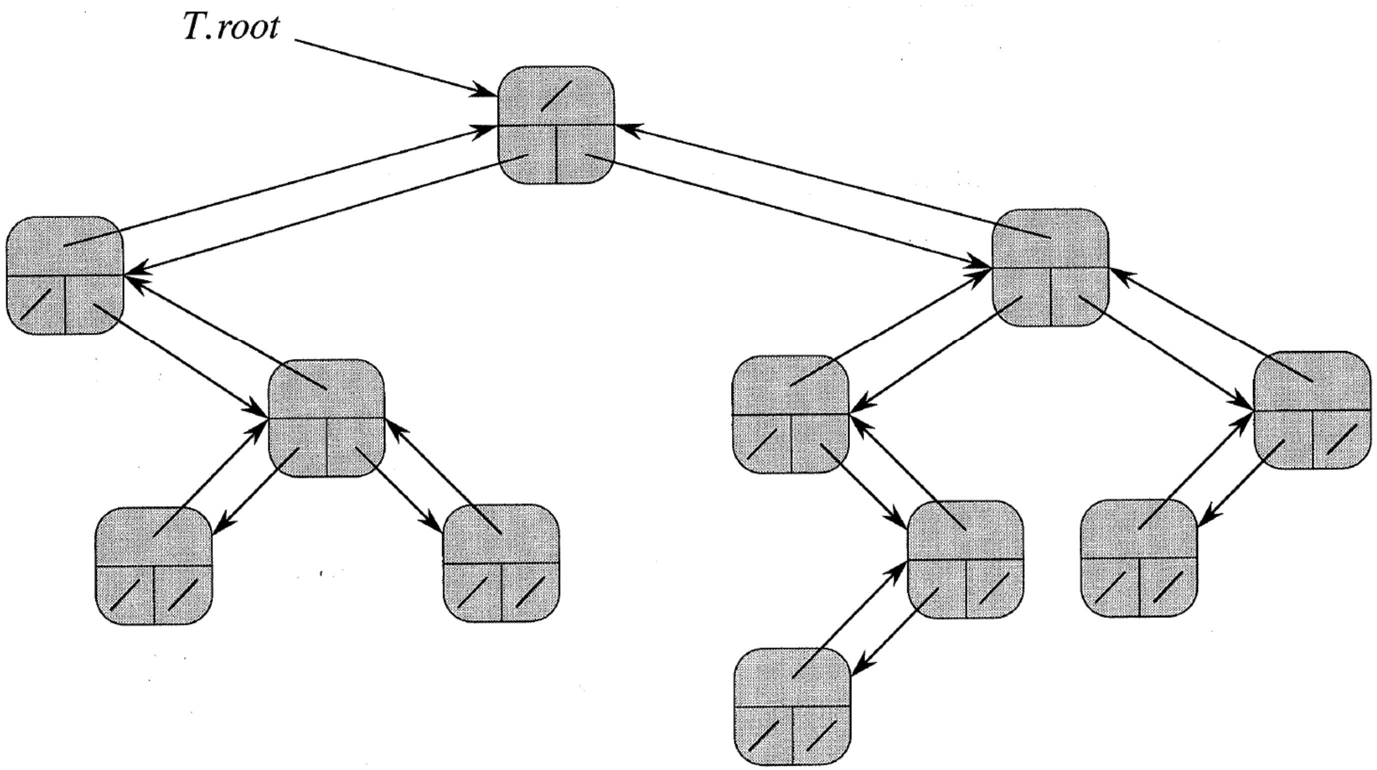
4.040 løsninger

Solve placerer

8.387.259 brikker

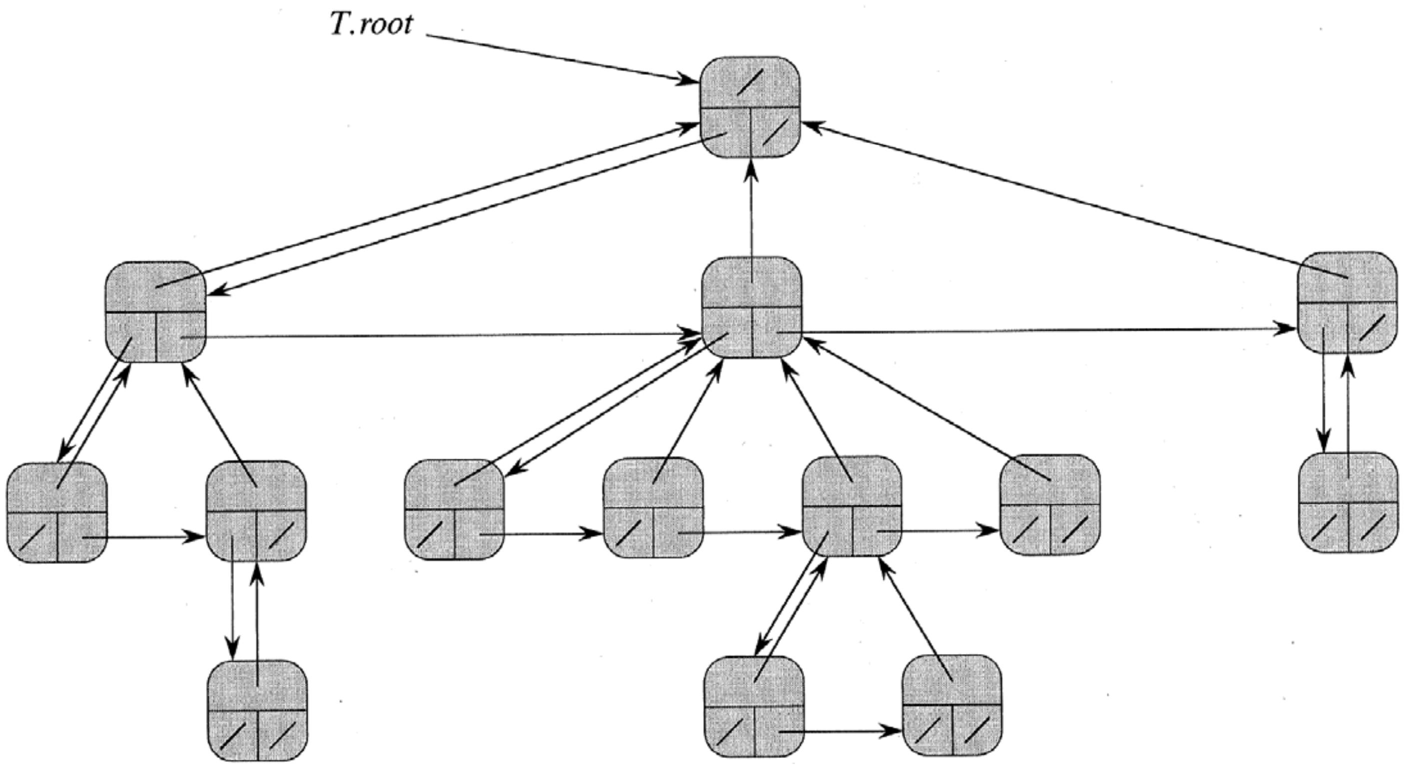

(Jorge Stolfi)

Binær Træ Repræsentation



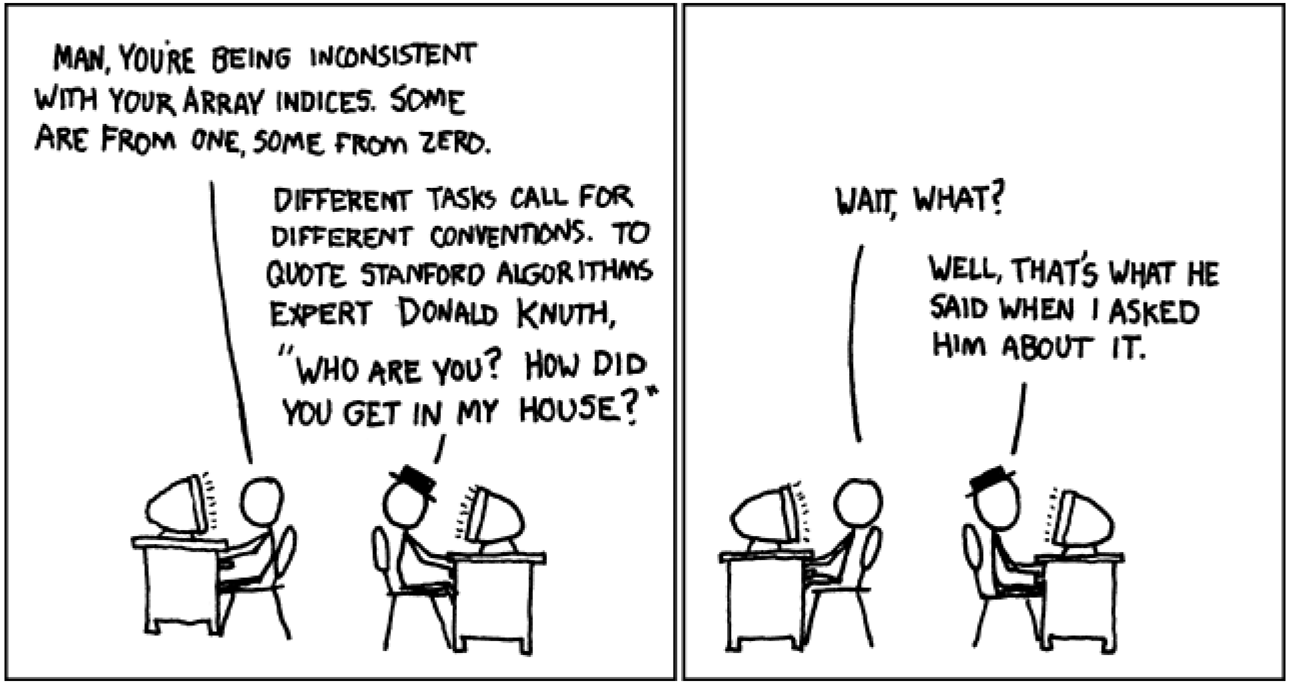
Felter: **Left**, **right**, **parent**

Træ Repræsentation



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

Donald Knuth



xkcd.com/163