

# Hoofdstuk 4

## Lineaire ADTs

# Lineaire Structuren vs. Lineaire ADT's

*Vorige hoofdstuk: Data is lineair gestructureerd in het geheugen.*

We onderzoeken tijd en geheugenverbruik.

*Dit hoofdstuk: ADT's die een "lineair gedrag" vertonen.*

*De **implementatie** ervan gebruikt **niet noodzakelijk** een lineaire datastructuur*

# Inhoud

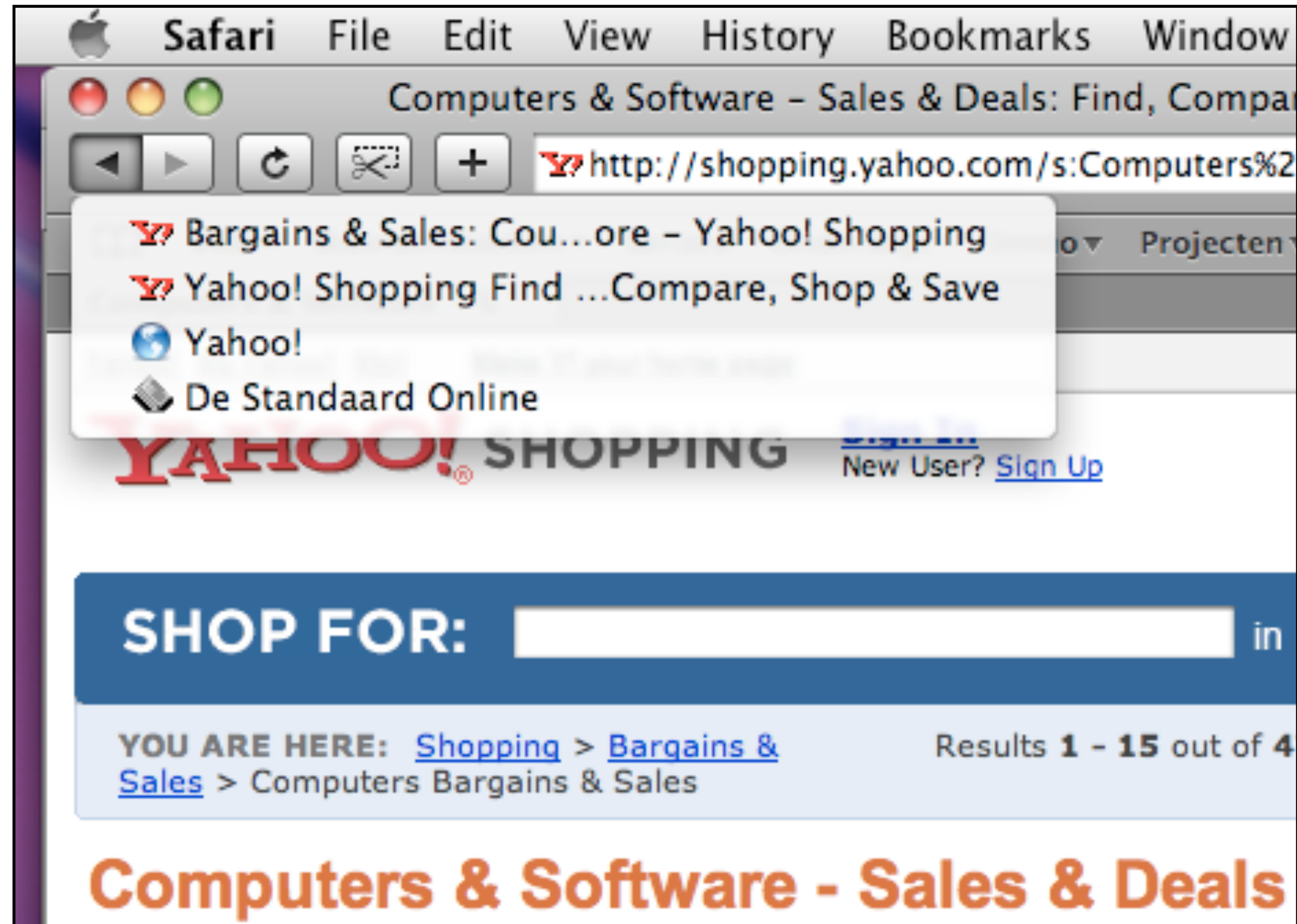
1. Stacks
2. Queues
3. Prioriteitenqueues
4. Heaps

# 4.1 Stacks

# ADT #1: Stacks

“stapels” in het  
Nederlands

*LIFO* gedrag = *Last In First Out*



# Het Stack-ADT

top leest de top maar  
verwijdert hem niet

```
ADT stack

new
    (  $\emptyset$   $\rightarrow$  stack )
stack?
    ( any  $\rightarrow$  boolean )
push!
    ( stack any  $\rightarrow$  stack )
top
    ( stack  $\rightarrow$  any )
pop!
    ( stack  $\rightarrow$  any )
empty?
    ( stack  $\rightarrow$  boolean )
full?
    ( stack  $\rightarrow$  boolean )
```

pop! leest de top  
en verwijdert hem

# Vectoriële Stack Implementatie

O(1)

representatie 

```
(define stack-size 10)

(define-record-type stack
  (make f v)
  stack?
  (f first-free first-free!)
  (v storage))

(define (new)
  (make 0 (make-vector stack-size ())))
```

*Zeer lage flexibiliteit  
wat betreft grootte*

verificatie 

```
(define (empty? stack)
  (= (first-free stack) 0))

(define (full? stack)
  (= (first-free stack)
     (vector-length (storage stack))))
```

```
(define (push! stack val)
  (define vector (storage stack))
  (define ff (first-free stack))
  (if (= ff (vector-length vector))
      (error "stack full (push!)" stack))
  (vector-set! vector ff val)
  (first-free! stack (+ ff 1))
  stack)
```

manipulatie 

```
(define (pop! stack)
  (define vector (storage stack))
  (define ff (first-free stack))
  (if (= ff 0)
      (error "stack empty (pop!)" stack))
  (let ((val (vector-ref vector (- ff 1))))
    (first-free! stack (- ff 1))
    val))
```

```
(define (top stack)
  (define vector (storage stack))
  (define ff (first-free stack))
  (if (= ff 0)
      (error "stack empty (top)" stack))
  (vector-ref vector (- ff 1)))
```

# Gelinkte Implementatie

O(1)

representatie 

```
(define-record-type stack  
  (make 1)  
  stack?  
  (1 scheme-list scheme-list!))
```

```
(define (new)  
  (make ()))
```

*Zeer hoge flexibiliteit wat  
betreft grootte*

verificatie 

```
(define (empty? stack)  
  (define slst (scheme-list stack))  
  (null? slst))
```

```
(define (full? stack)  
  #f)
```

manipulatie 

```
(define (push! stack val)  
  (define slst (scheme-list stack))  
  (scheme-list! stack (cons val slst))  
  stack)
```

```
(define (top stack)  
  (define slst (scheme-list stack))  
  (if (null? slst)  
      (error "stack empty (top)" stack)  
      (car slst)))
```

```
(define (pop! stack)  
  (define slst (scheme-list stack))  
  (if (null? slst)  
      (error "stack empty (pop!)" stack)  
      (let ((val (car slst)))  
        (scheme-list! stack (cdr slst))  
        val)))
```



# Stacks: Performantie van beide Implementaties

	vectorieel	gelinkt
new	$O(1)$	$O(1)$
empty?	$O(1)$	$O(1)$
full?	$O(1)$	$O(1)$
stack?	$O(1)$	$O(1)$
top	$O(1)$	$O(1)$
push!	$O(1)$	$O(1)$
pop!	$O(1)$	$O(1)$

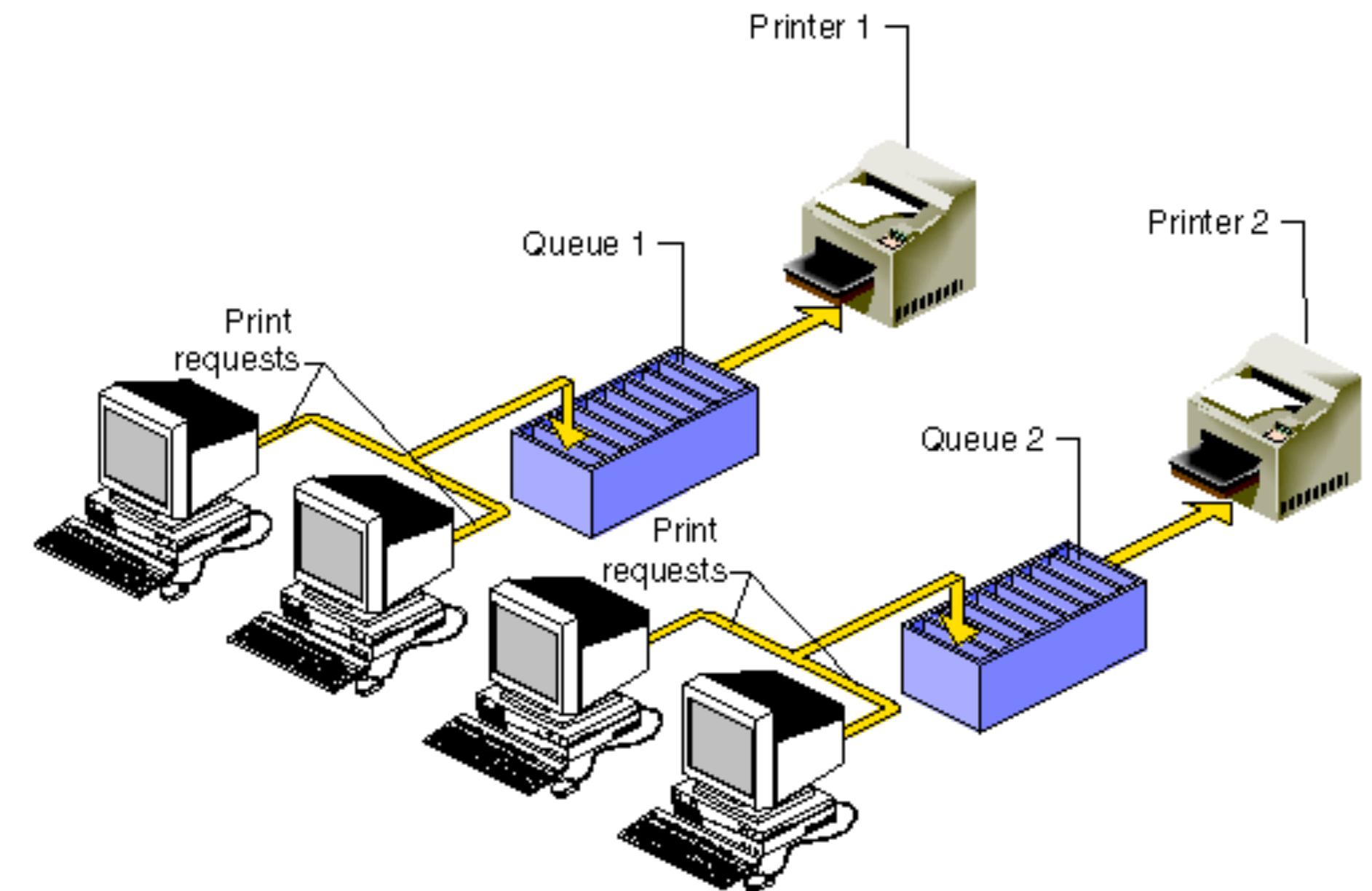
*Laat je keuze dus afhangen van de criteria “flexibiliteit” versus “geheugenverbruik”.*

## 4.2 Queues

“wachtrijen” in het Nederlands

# ADT #2: Queues

*FIFO* gedrag = *First In First Out*



# Het Queue-ADT

ADT queue

new

(  $\emptyset \rightarrow \text{queue}$  )

queue?

( any  $\rightarrow$  boolean )

enqueue!

( queue any  $\rightarrow$  queue )

peek

( queue  $\rightarrow$  any )

serve!

( queue  $\rightarrow$  any )

empty?

( queue  $\rightarrow$  boolean )

full?

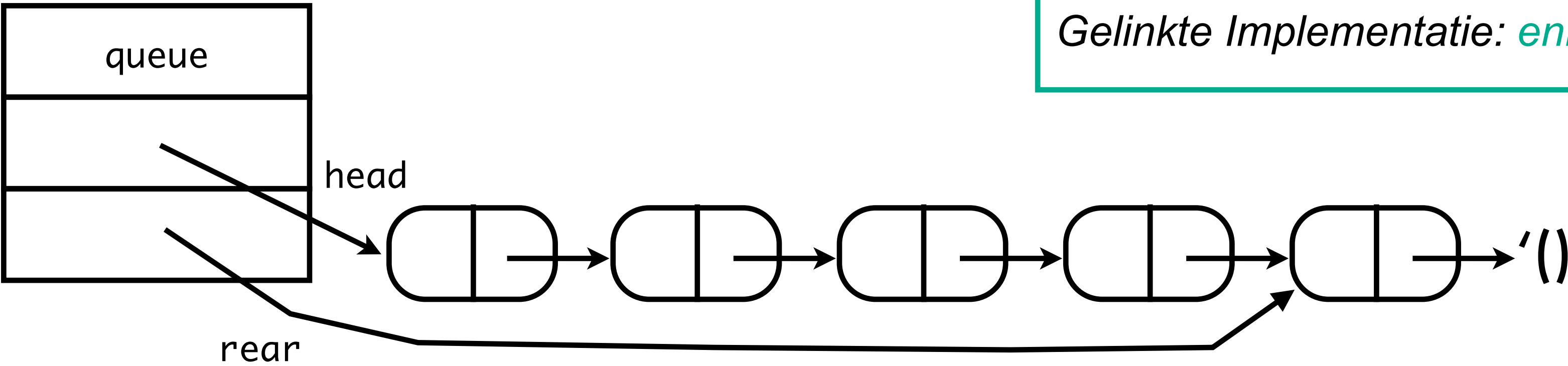
( queue  $\rightarrow$  boolean )

peek leest de kop maar  
verwijdert hem niet

serve! leest de kop  
en verwijdert hem

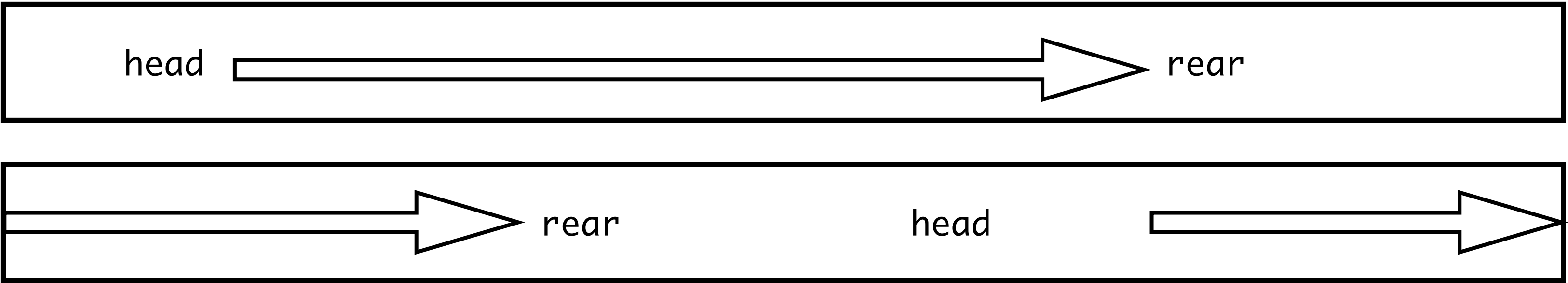


# Queues: Implementatiestrategieën



*Gelinkte Implementatie: enkel gelinkt met rear*

*Vectoriële Implementatie: met circulaire vector*



# Gelinkte Queue Implementatie

## representatie

```
(define-record-type queue
  (make h r)
  queue?
  (h head head!)
  (r rear rear!))
```

```
(define (new)
  (make '() '()))
```

## verificatie

```
(define (empty? q)
  (null? (head q)))
```

```
(define (full? q)
  #f)
```

O(1)

## manipulatie

```
(define (enqueue! q val)
  (define last (rear q))
  (define node (cons val '()))
  (if (null? (head q))
      (head! q node)
      (set-cdr! last node))
  (rear! q node)
  q)
```

```
(define (peek q)
  (if (null? (head q))
      (error "empty queue (peek)" q)
      (car (head q))))
```

```
(define (serve! q)
  (define first (head q))
  (if (null? first)
      (error "empty queue (serve!)" q))
  (head! q (cdr first))
  (if (null? (head q))
      (rear! q '()))
  (car first))
```

# Even tussendoor...

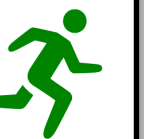
`(modulo x y)` = *rest bij deling van x door y*

```
(define max 15)

(define (modulo-series ctr n)
  (display (list "mod" ctr n " = " (modulo ctr n)))
  (newline)
  (if (< ctr max)
      (modulo-series (+ ctr 1) n)))
```



```
Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512 MB.
> (modulo-series 0 5)
(mod 0 5 = 0)
(mod 1 5 = 1)
(mod 2 5 = 2)
(mod 3 5 = 3)
(mod 4 5 = 4)
(mod 5 5 = 0)
(mod 6 5 = 1)
(mod 7 5 = 2)
(mod 8 5 = 3)
(mod 9 5 = 4)
(mod 10 5 = 0)
(mod 11 5 = 1)
(mod 12 5 = 2)
(mod 13 5 = 3)
(mod 14 5 = 4)
(mod 15 5 = 0)
```



# Vectoriële Queue Implementatie

representatie 

```
(define capacity 5)

(define-record-type queue
  (make s h r)
  queue?
  (s storage)
  (h head head!)
  (r rear rear!))

(define (new)
  (make (make-vector capacity) 0 0))
```

verificatie 

```
(define (empty? q)
  (= (head q)
     (rear q)))

(define (full? q)
  (= (modulo (+ (rear q) 1) capacity)
     (head q)))
```

manipulatie 

```
(define (enqueue! q val)
  (if (full? q)
      (error "full queue (enqueue!)" q))
      (let ((new-rear (modulo (+ (rear q) 1) capacity)))
        (vector-set! (storage q) (rear q) val)
        (rear! q new-rear)))
  q)

(define (peek q)
  (if (empty? q)
      (error "empty queue (peek)" q))
      (vector-ref (storage q) (head q)))

(define (serve! q)
  (if (empty? q)
      (error "empty queue (peek)" q))
      (let ((result (vector-ref (storage q) (head q))))
        (head! q (modulo (+ (head q) 1) capacity))
        result))
```

O(1)



# Queues: Performantie van beide Implementaties

	vectorieel	gelinkt
new	$O(1)$	$O(1)$
empty?	$O(1)$	$O(1)$
full?	$O(1)$	$O(1)$
queue?	$O(1)$	$O(1)$
peek	$O(1)$	$O(1)$
enqueue!	$O(1)$	$O(1)$
serve!	$O(1)$	$O(1)$

*Laat je keuze dus afhangen van de criteria “flexibiliteit” vs. “geheugenverbruik”.*

## 4.3 Prioriteitenqueues

# ADT #3: Prioriteitenqueues

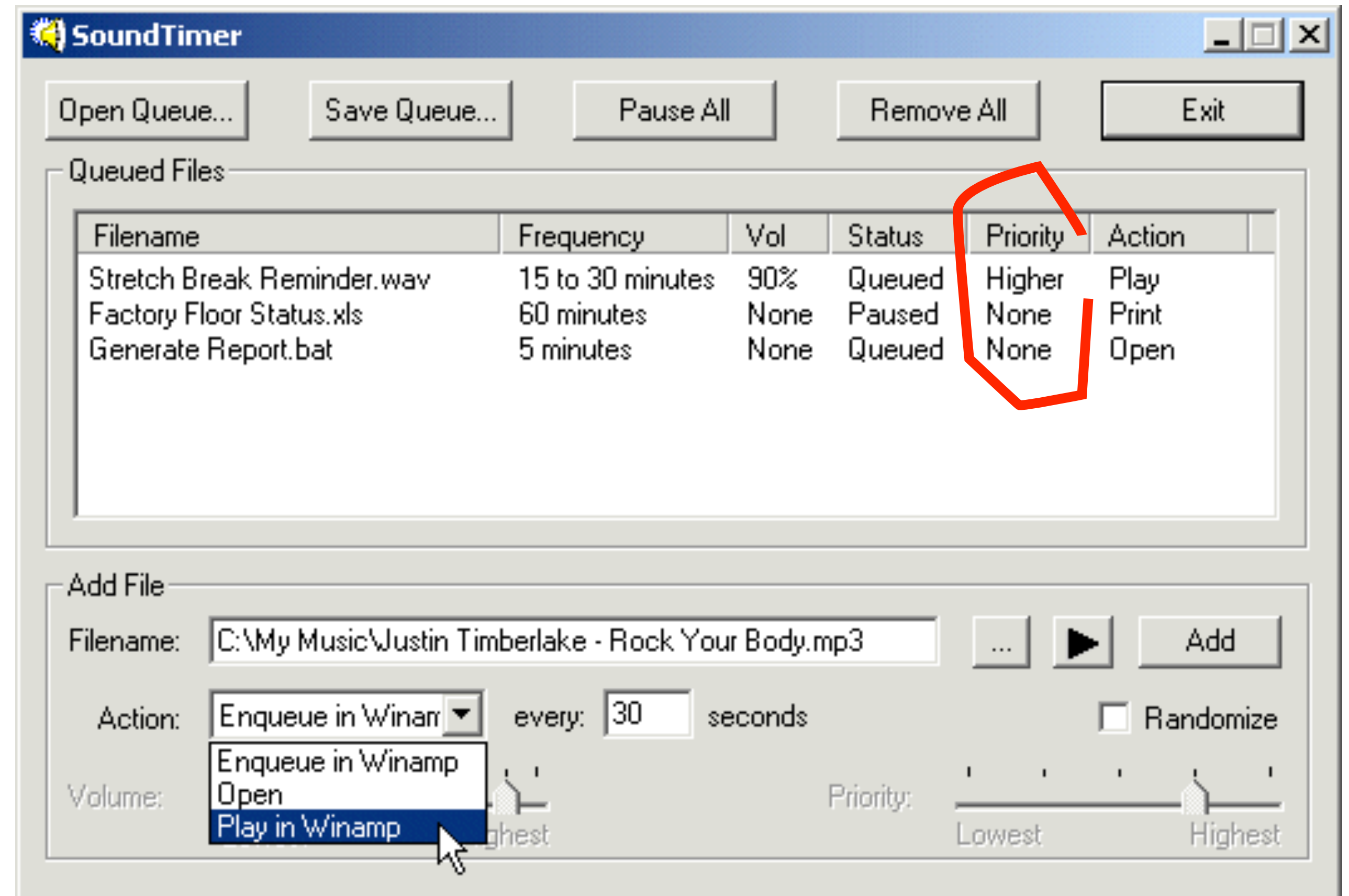
*HPFO* gedrag = *Highest Priority First Out*

“prioriteitenwachtrijen” in  
het Nederlands

© Original Artist  
Reproduction rights obtainable from  
[www.CartoonStock.com](http://www.CartoonStock.com)



“It's death and taxes to see you - taxes wants to  
come in first.”



# Het Priority Queue ADT

new verwacht >>?

peek leest het element met de hoogste prioriteit maar verwijdert het niet

ADT priority-queue< P >

new

( ( P P → boolean ) → priority-queue< P > )

priority-queue?

( any → boolean )

enqueue!

( priority-queue< P > any P → priority-queue< P > )

peek

( priority-queue< P > → any )

serve!

( priority-queue< P > → any )

full?

( priority-queue< P > → boolean )

empty?

( priority-queue< P > → boolean )

serve! leest het element met de hoogste prioriteit en verwijdert het.

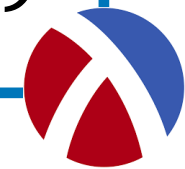
# Priority Queue Items

*Priority queue items* vormen de opgeslagen “values”. Een priority queue item bestaat uit een *data element* samen met de erbij horende *prioriteit*.

Zal de rol van “V” spelen in het gebruik van bestaande ADTs

```
(define pq-item-make cons)
(define pq-item-val car)
(define pq-item-priority cdr)

(define (lift func)
  (lambda (item1 item2)
    (func (pq-item-priority item1)
          (pq-item-priority item2)))))
```



```
Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512 MB.
> (define item1 (pq-item-make "Wolf" 1))
> item1
("Wolf" . 1)
> (define item2 (pq-item-make "Vivi" 2))
> (> item1 item2)
✗✗ >: contract violation
  expected: real?
  given: (cons "Vivi" 2)
> (define >> (lift >))
> (>> item1 item2)
#f
```



# Implementaties: 2 Naïeve Pogingen

Basisidee: gooi alles gedisciplineerd in een gesorteerde lijst, ook al kost dat tijd. Nadien nemen we het element met de hoogste prioriteit gewoon vooraan.

*Een eerste implementatie is gebaseerd op sorted-lists*

Basisidee: gooi alles gewoon in een lijst, zo snel mogelijk. Nadien zoeken we wel de hoogste prioriteit.

*Een tweede implementatie is gebaseerd op position-lists*



# Sorted List Implementatie

manipulatie 

$O(n)$

```
(define-record-type priority-queue
  (make s)
  priority-queue?
  (s slist))
```

```
(define (new >>?)
  (make (slist:new (lift >>?)
                  (lift eq?))))
```

representatie 

```
(define (enqueue! pq val pty)
  (slist:add! (slist pq) (pq-item-make val pty))
  pq)
```

$O(1)$

```
(define (serve! pq)
  (define slst (slist pq))
  (if (empty? pq)
      (error "empty priority queue (serve!)" pq))
  (slist:set-current-to-first! slst)
  (let ((served-item (slist:peek slst)))
    (slist:delete! slst)
    (pq-item-val served-item))))
```

$O(1)$

```
(define (peek pq)
  (define slst (slist pq))
  (if (empty? pq)
      (error "empty priority queue (peek)" pq))
  (slist:set-current-to-first! slst)
  (pq-item-val (slist:peek slst))))
```

# Positional List Implementatie

$O(n)$

```
(define-record-type priority-queue
  (make l g)
  priority-queue?
  (l plist)
  (g greater))
```

representatie 

```
(define (new >>?)
  (make (plist:new eq?) (lift >>?)))
```

$O(1)$

```
(define (enqueue! pq val pty)
  (plist:add-before! (plist pq) (pq-item-make val pty))
  pq)
```

manipulatie 

```
(define (serve! pq)
  (define plist (plist pq))
  (define >>? (greater pq))
  (if (empty? pq)
      (error "priority queue empty (serve!)" pq))
  (let*
    ((highest-priority-position
      (let loop
        ((current-pos (plist:first plist))
         (maximum-pos (plist:first plist))
         (if (plist:has-next? plist current-pos)
             (loop (plist:next plist current-pos)
                   (if (>>? (plist:peek plist current-pos)
                           (plist:peek plist maximum-pos))
                       current-pos
                       maximum-pos)))
         (if (>>? (plist:peek plist current-pos)
                 (plist:peek plist maximum-pos))
             current-pos
             maximum-pos))))
     (served-item (plist:peek plist highest-priority-position)))
    (plist:delete! plist highest-priority-position)
    (pq-item-val served-item)))
```





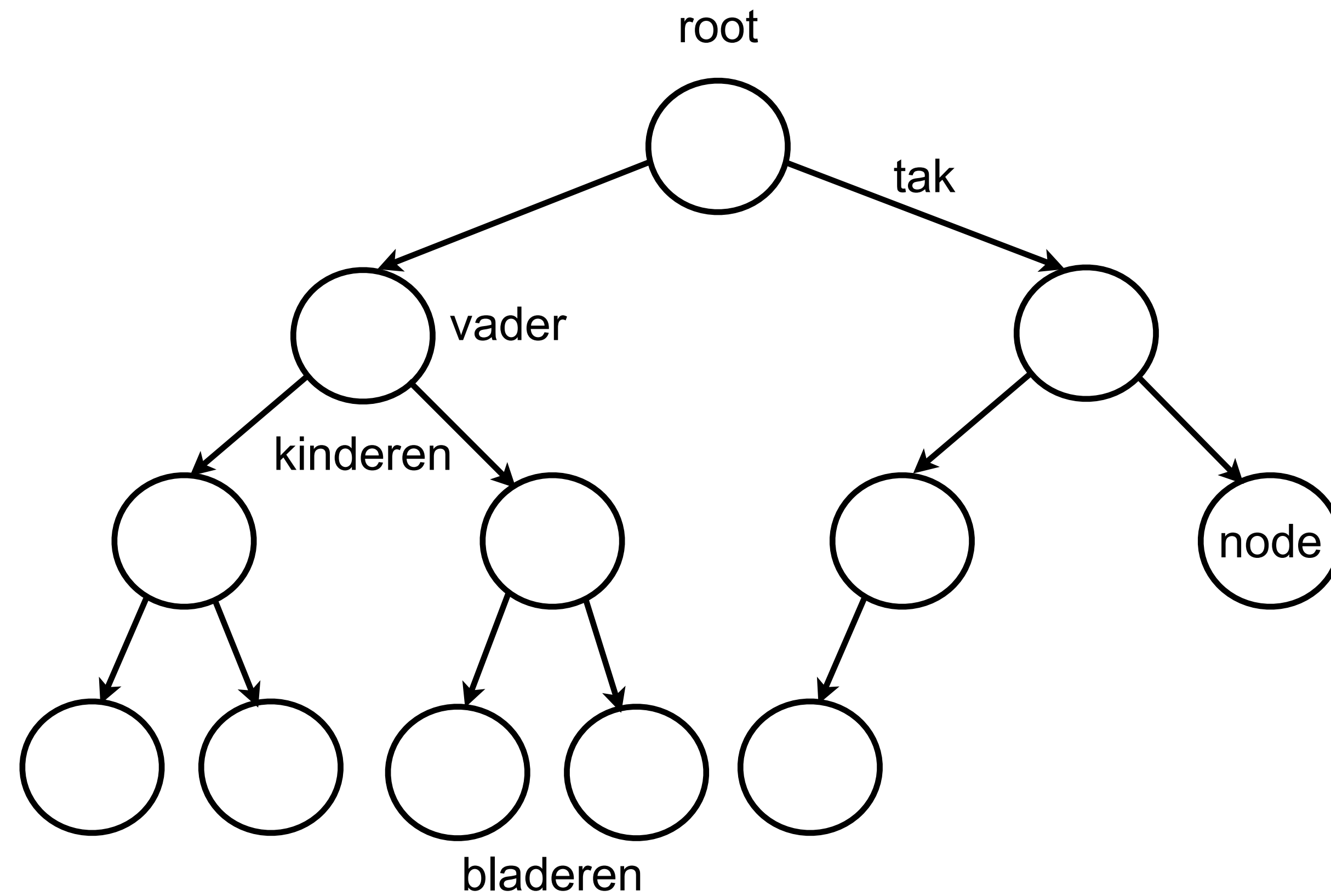
# Priority Queues: Overzicht Naïeve Implementaties

	sorted-list	position-list
new	$O(1)$	$O(1)$
empty?	$O(1)$	$O(1)$
full?	$O(1)$	$O(1)$
priority-queue?	$O(1)$	$O(1)$
peek	$O(1)$	$O(n)$
enqueue!	$O(n)$	$O(1)$
serve!	$O(1)$	$O(n)$

*Dit is **heel slecht nieuws**: een priority queue is geen stabiele datastructuur: ieder element moet er 1 in én 1 keer uit gaan.*

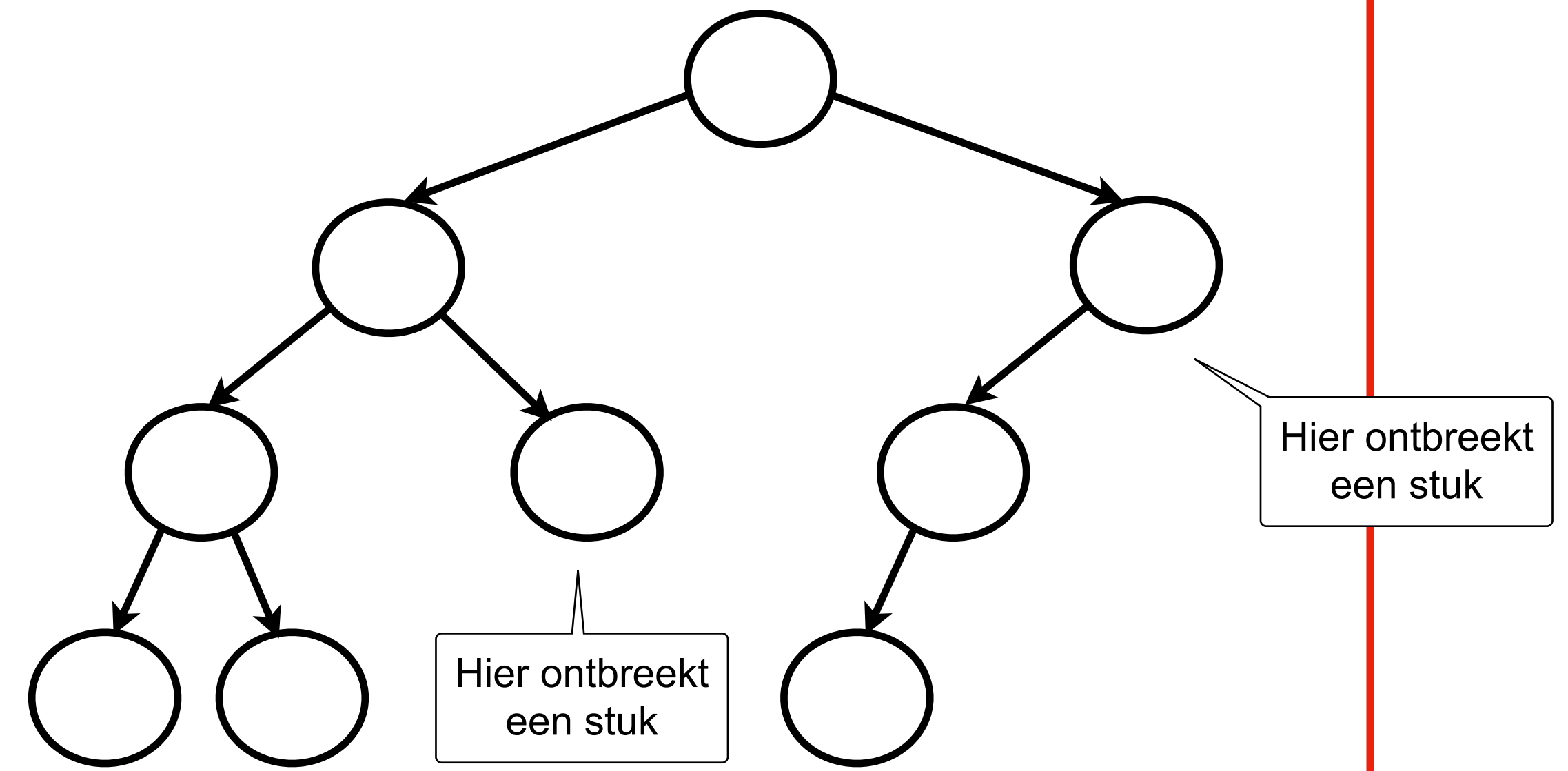
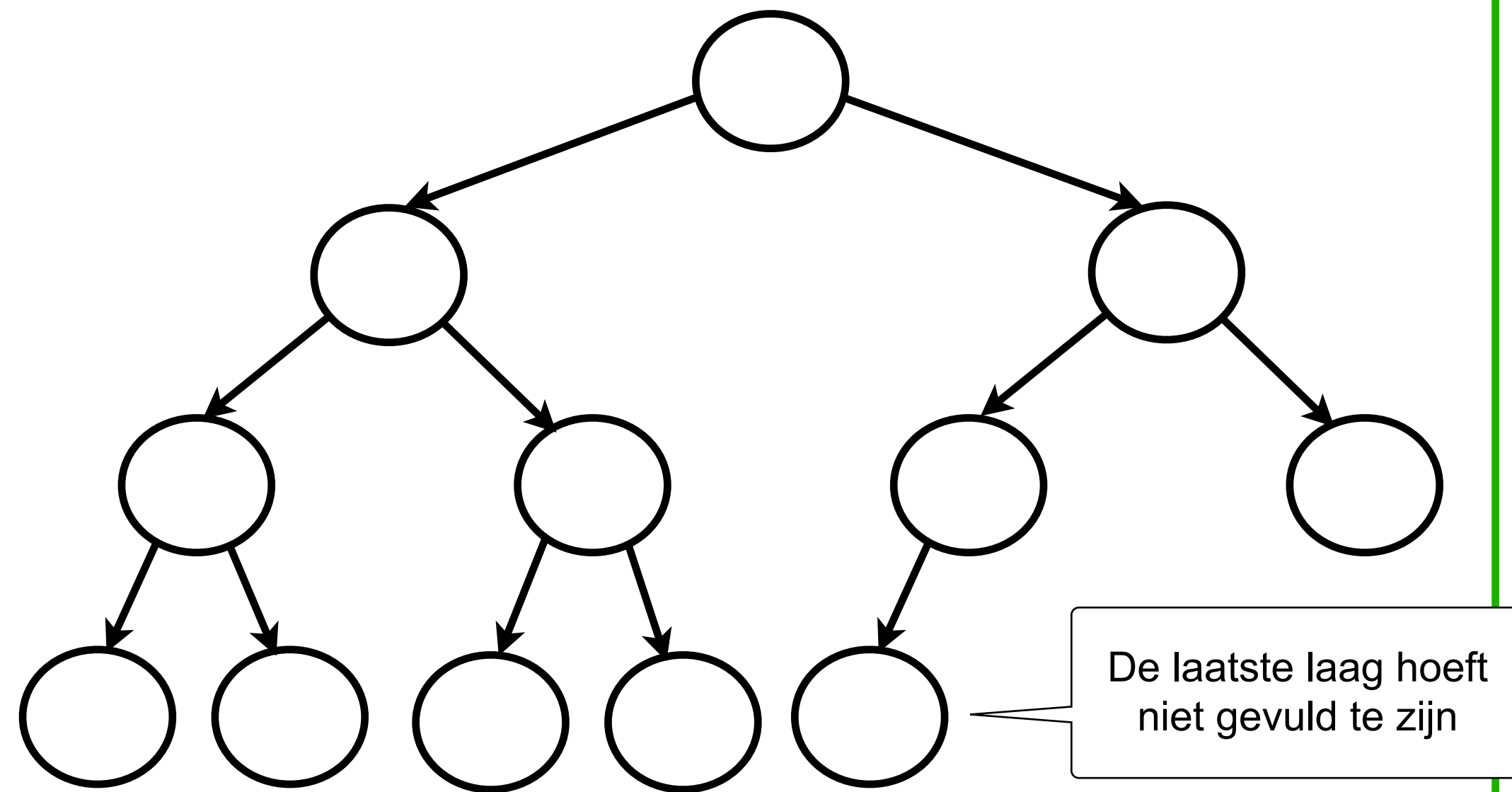
## 4.3 Heaps

# Interludium: Bomenterminologie



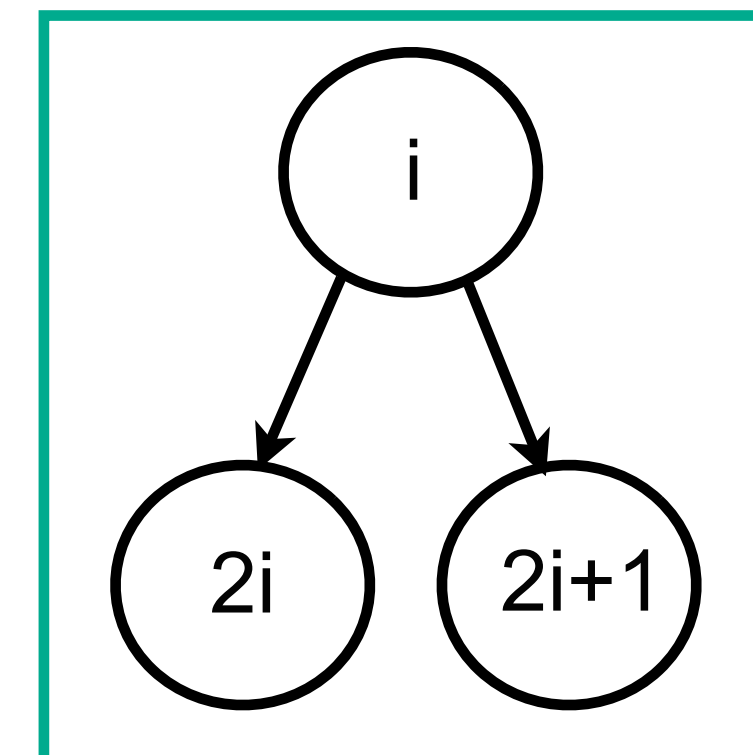
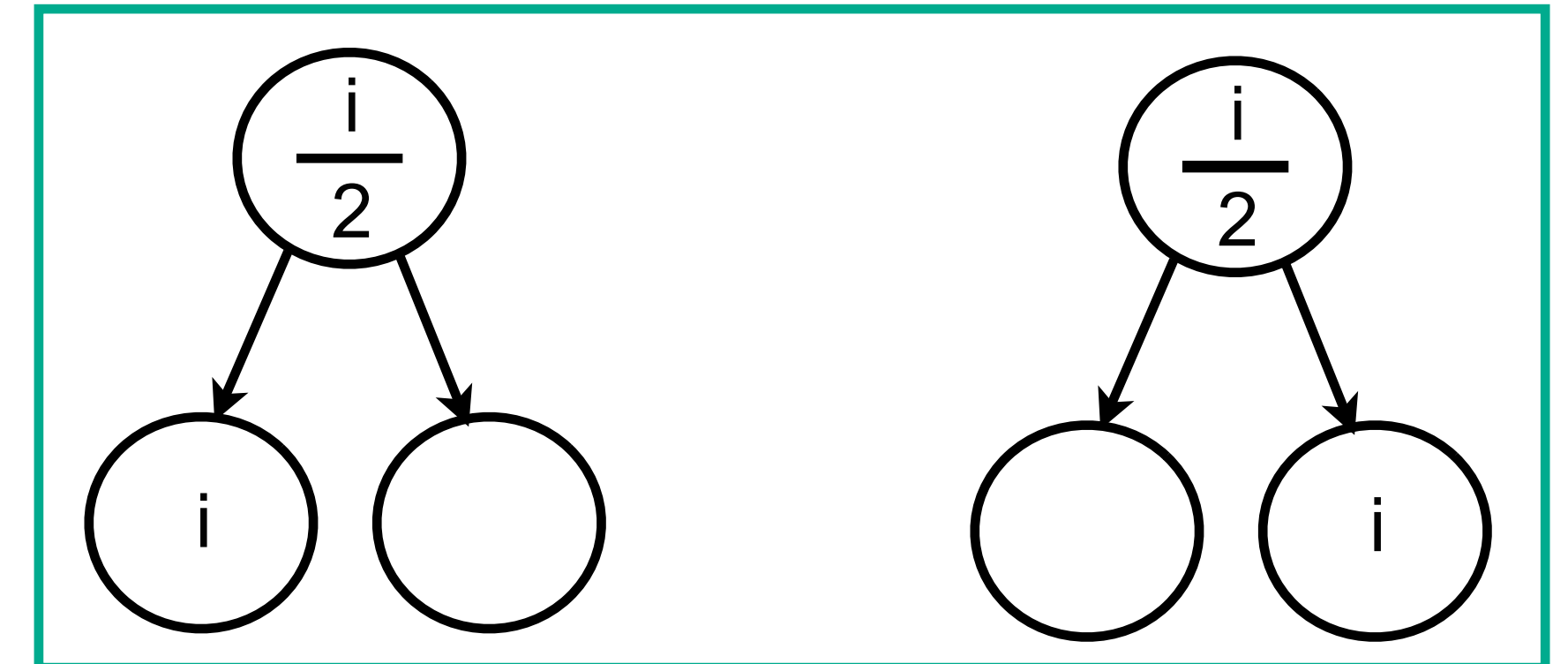
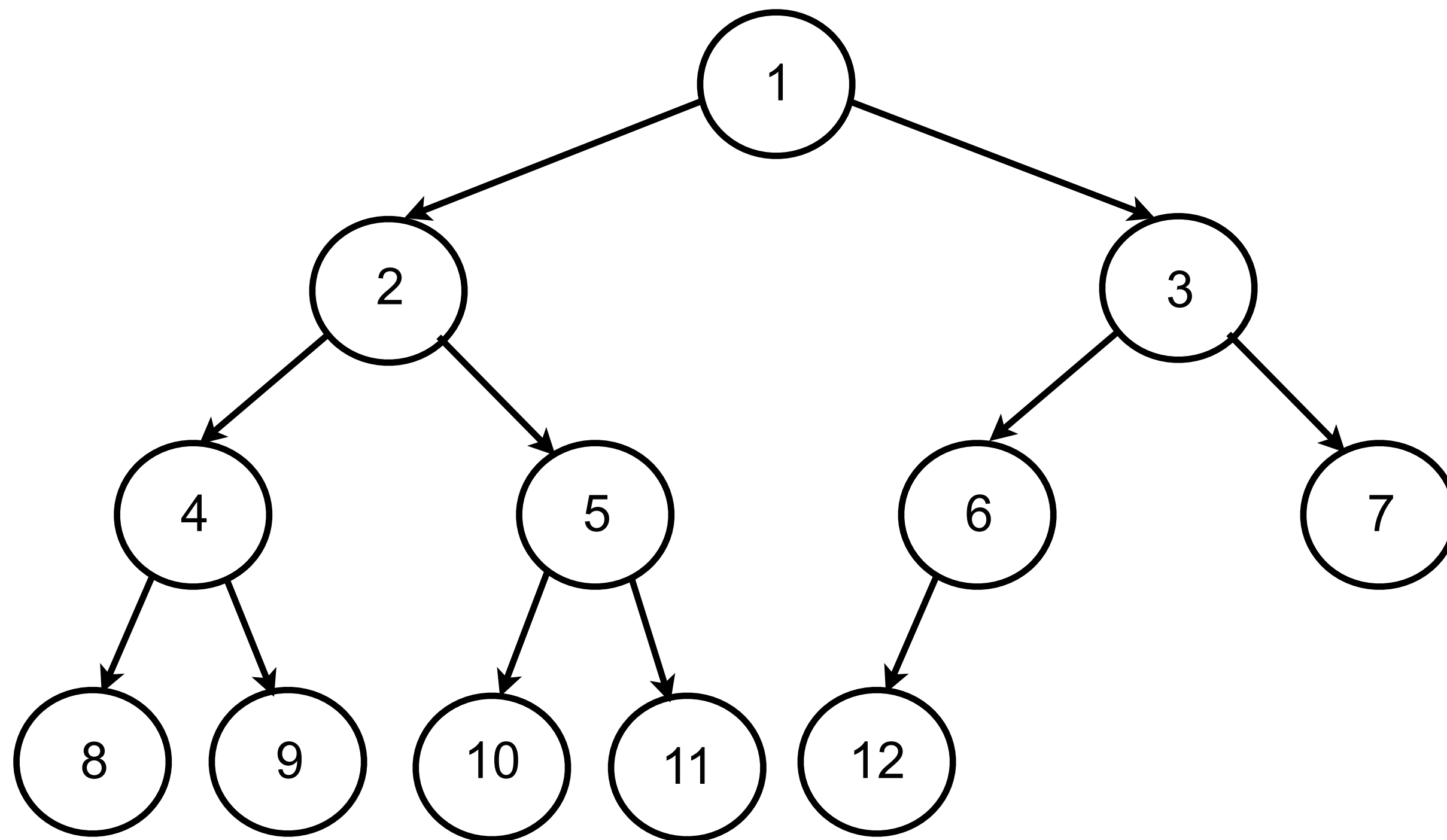
# Volledige Bomen

Een *volledige boom* heeft geen gaten wanneer we hem van links naar rechts lezen, laag per laag



# Volledige Bomen: Eigenschap

De knopen van een volledige boom kan je **eenduidig** nummeren van 1 tot  $n$ . Een volledige boom kan je dus in een **vector** opslaan.



Je kan de index van je vader uitrekenen.

Je kan de index van je beide kinderen uitrekenen.



# Een nieuw ADT: Heaps



Een *heap* is een rij elementen  $e_1, e_2, \dots, e_n$  zodat de *heapvoorwaarde* waar is:

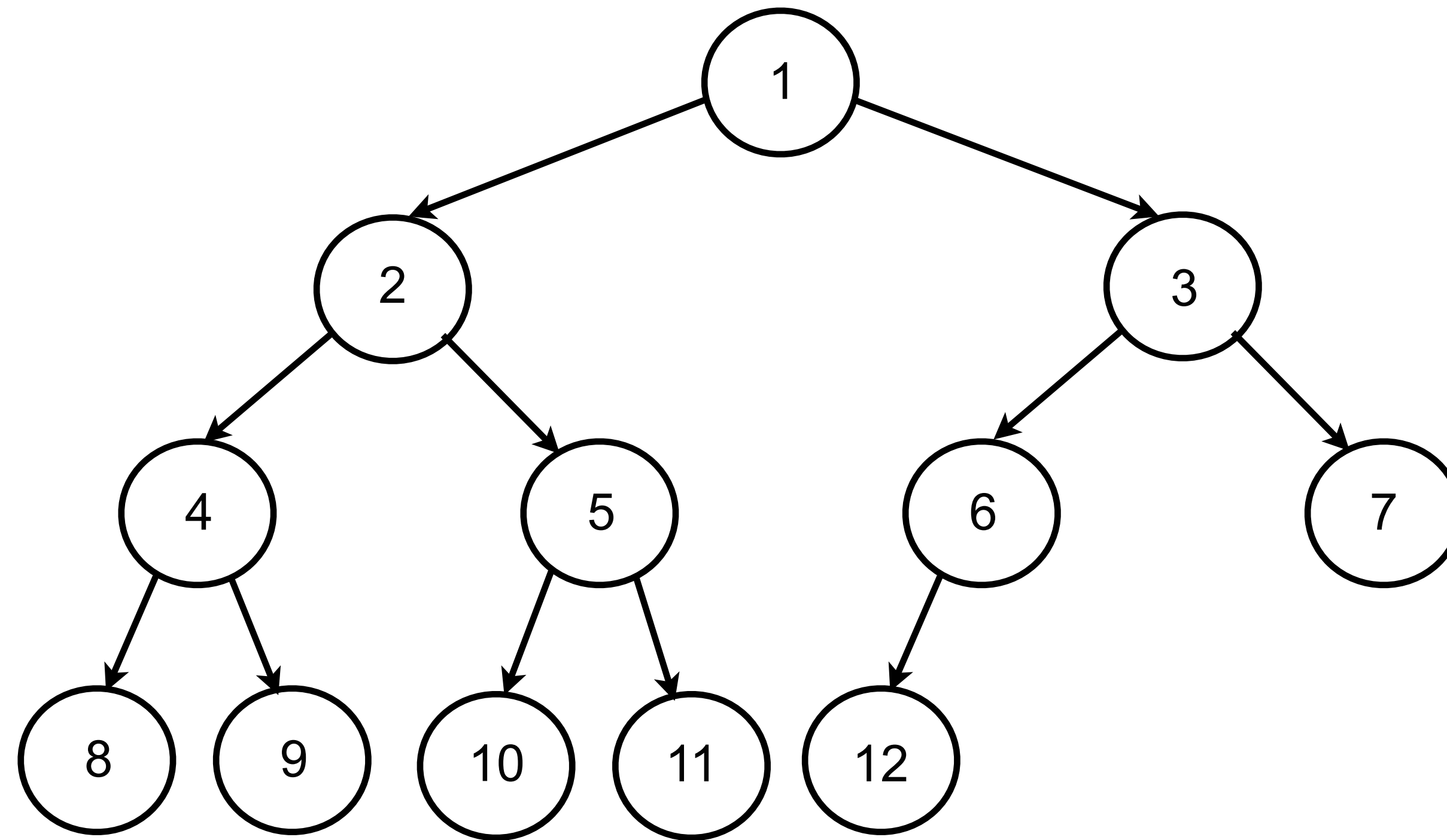
$$e_i < e_{2i} \quad \text{en} \quad e_i < e_{2i+1}$$

*Let op in Scheme:  $e_0 < e_{2.0}$  kan niet*

*Een heap is niet noodzakelijk gesorteerd. Een gesorteerde rij is wel een altijd een heap.*



# Een Heap als Volledige Boom



$$e_i < e_{2i} \quad \text{en} \quad e_i < e_{2i+1}$$

$\Rightarrow$

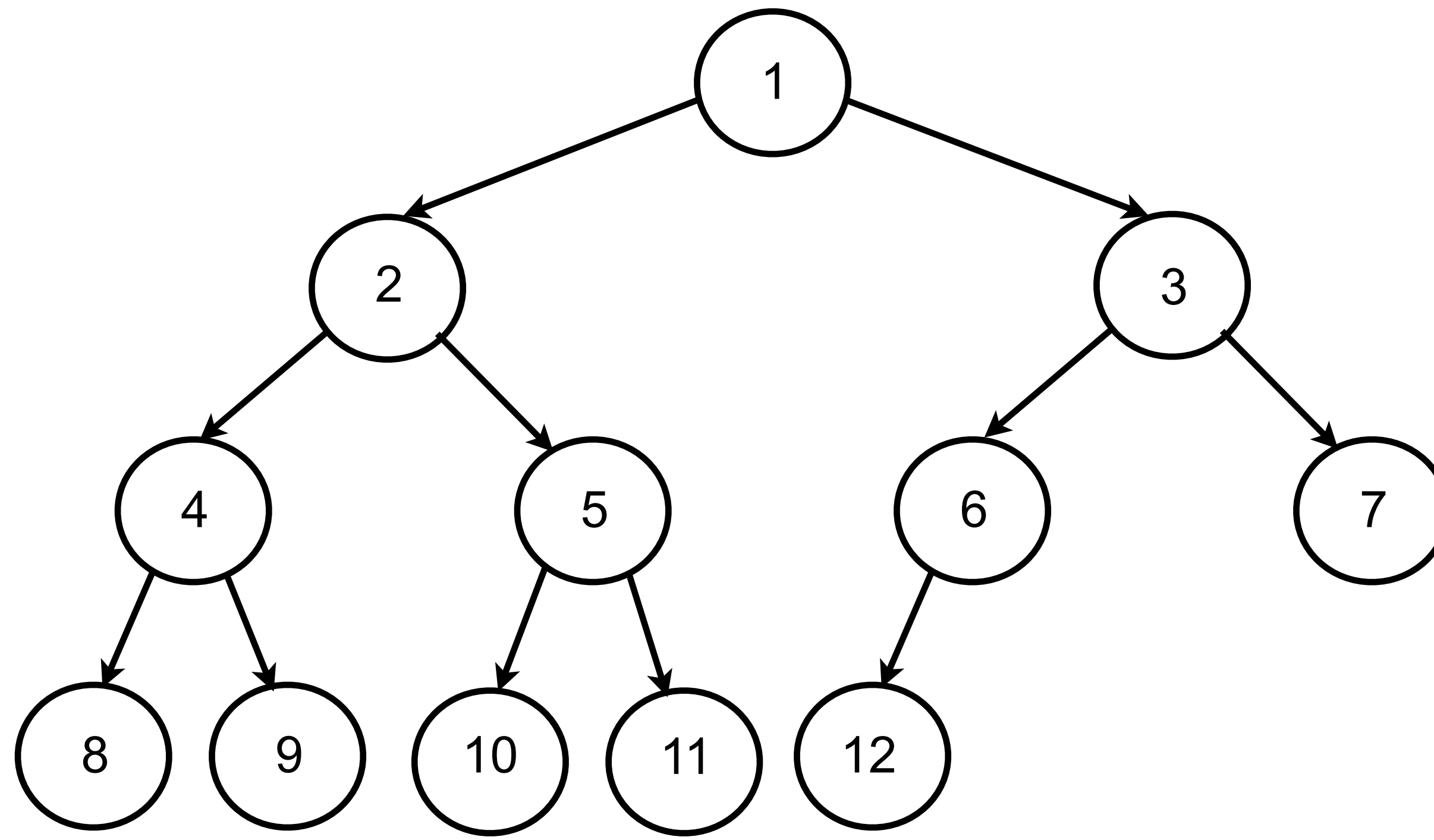
$e_1$  is het kleinste element

# Eigenschappen van Heaps

Niveau = aantal stappen verwijderd van de root

Hoogte = maximum aantal takken van een leaf naar de root

$h=3$



niveau 0 =  $2^0$

niveau 1 =  $2^1$

niveau 2 =  $2^2$

niveau 3 =  $2^3$

Eigenschap 1

*De hoogte van een heap met  $n$  elementen is  $h = \lfloor \log_2(n) \rfloor$*

Eigenschap 2

*In een heap met  $n$  nodes zijn er  $\lceil \frac{n}{2} \rceil$  bladeren*

Eigenschap 3

*Er bevinden zich maximum  $\lceil \frac{n}{2^{h+1}} \rceil$  knopen op niveau  $h$*



# Het Heap ADT

Merk op: er is geen find operatie.  
Een heap dient niet om in te zoeken!

ADT heap< V >

new verwacht <<?

from-scheme-vector

( vector< V > ( V V → boolean ) → heap< V > )

new

( number ( V V → boolean ) → heap< V > )

full?

( heap< V > → boolean )

empty?

( heap< V > → boolean )

insert!

( heap< V > V → heap< V > )

delete!

( heap< V > → V )

peek

( heap< V > → V )

size

( heap< V > → number )

peek leest de top maar  
verwijdert hem niet

insert! gooit een nieuw element  
op de hoop en herorganiseert hem

delete! leest de top en verwijdert  
hem, en herorganiseert de boel

# Heap: Implementatie

## representatie

```
(define-record-type heap
  (make v s l)
  heap?
  (v storage storage!)
  (s size size!)
  (l lesser))

(define (new capacity <<?)
  (make (make-vector capacity) 0 <<?))
```

## verificatie

```
(define (full? heap)
  (= (vector-length (storage heap))
     (size heap)))

(define (empty? heap)
  (= (size heap) 0))

(define (peek heap)
  (if (empty? heap)
      (error "heap empty" heap)
      (vector-ref (storage heap) 0)))
```

# Implementatie: Toevoegen en verwijderen

*Herinner de heap conditie:*

$$e_i < e_{2i} \text{ en } e_i < e_{2i+1}$$

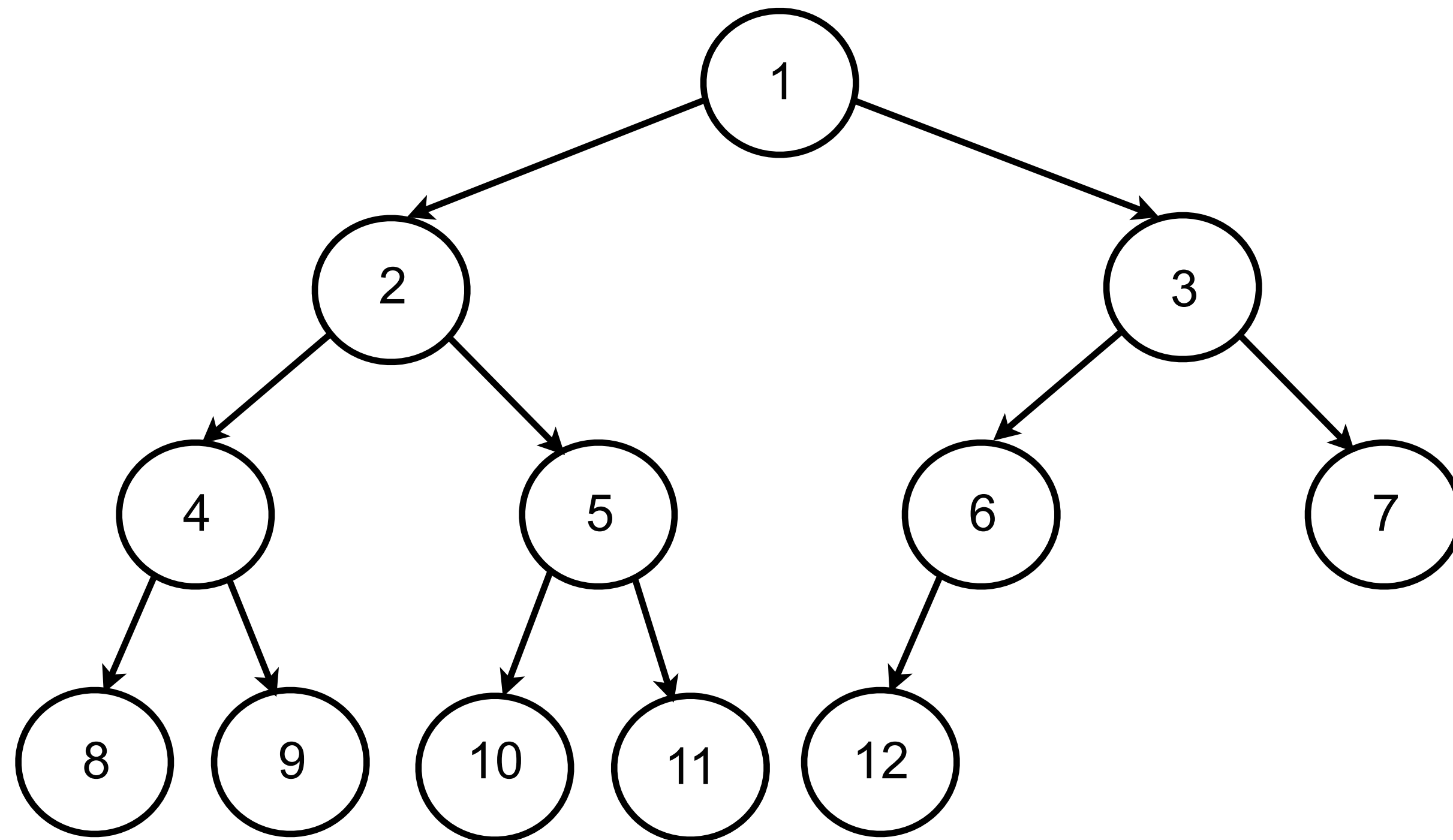
*die altijd waar moet zijn.*

⇒ Je kan *niet zomaar* ergens een element *invoegen* in de heap +

Je kan *niet zomaar* ergens een element *wegknippen* uit de heap

⇒ *Gedisciplineerd toevoegen en verwijderen*

# Implementatie: Gedisciplineerd Toevoegen

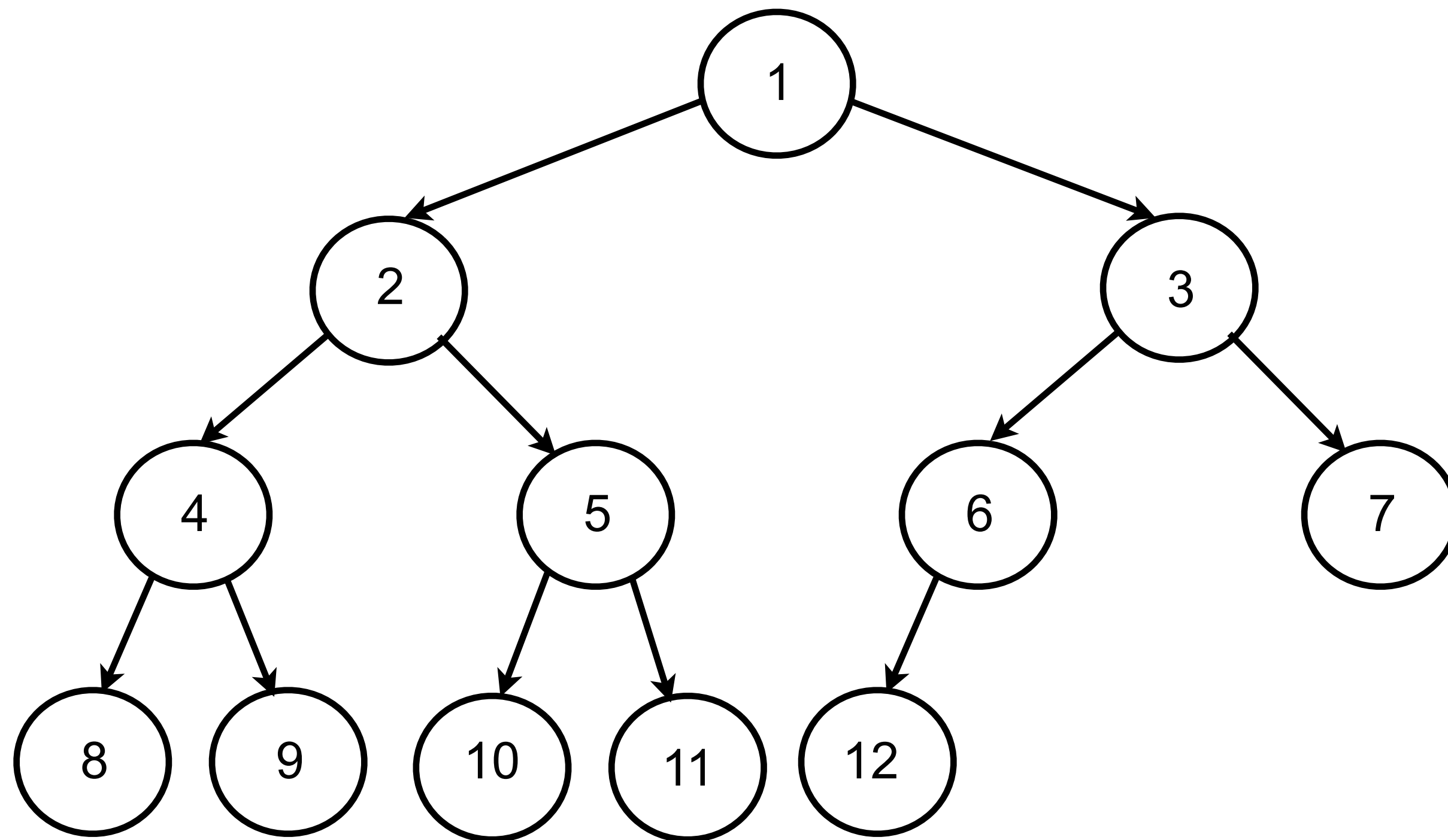


```
(define (insert! heap item)
  (if (full? heap)
      (error "heap full" heap))
  (let* ((vector (storage heap))
         (size (size heap)))
    (vector-set! vector size item)
    (if (> size 0)
        (sift-up heap (+ size 1)))
    (size! heap (+ size 1))))
```

Zet een nieuw element op de laatste plaats (13)  
en “zift” het naar boven tot op de juiste plaats



# Implementatie: Gedisciplineerd Verwijderen

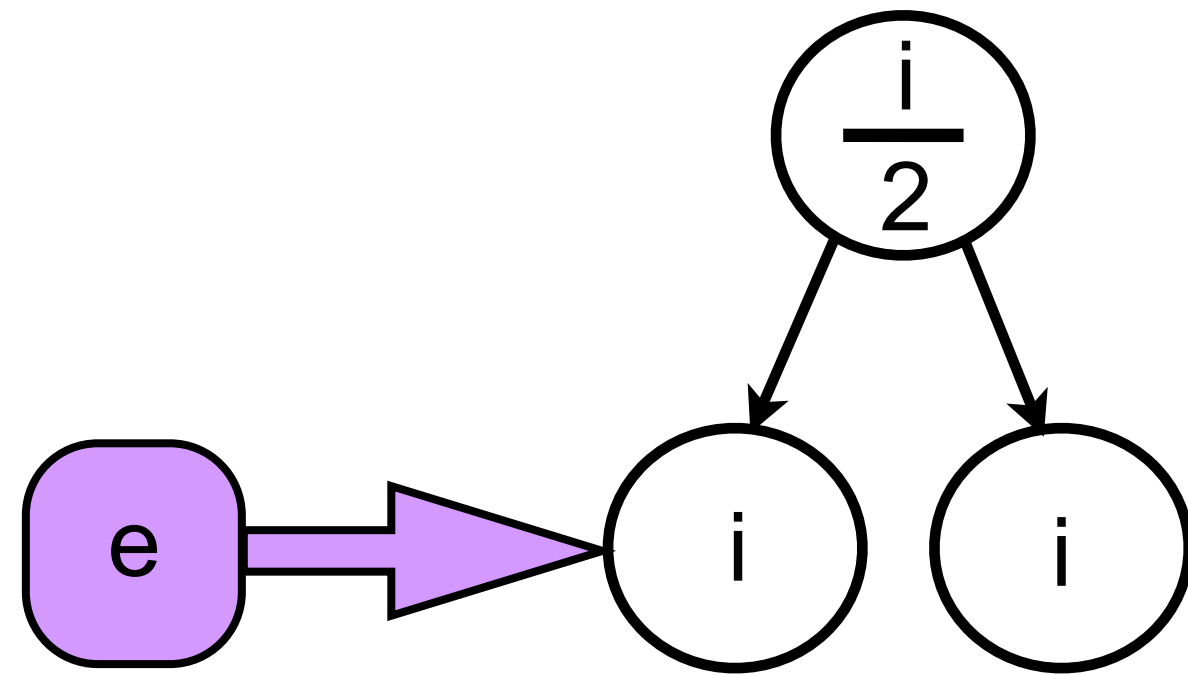


Knip de top weg (1) en vervang hem door het laatste element (12). *Zift het vervolgens naar beneden* tot op de juiste plaats

```
(define (delete! heap)
  (if (empty? heap)
      (error "heap empty" heap))
  (let* ((vector (storage heap))
         (size (size heap))
         (first (vector-ref vector 0))
         (last (vector-ref vector (- size 1))))
    (size! heap (- size 1))
    (if (> size 1)
        (begin
          (vector-set! vector 0 last)
          (sift-down heap 1)))
    first))
```



# Implementatie: Opwaarts ziften



*Als  $i$  de root is zijn we klaar. Stop “e” in de root.*

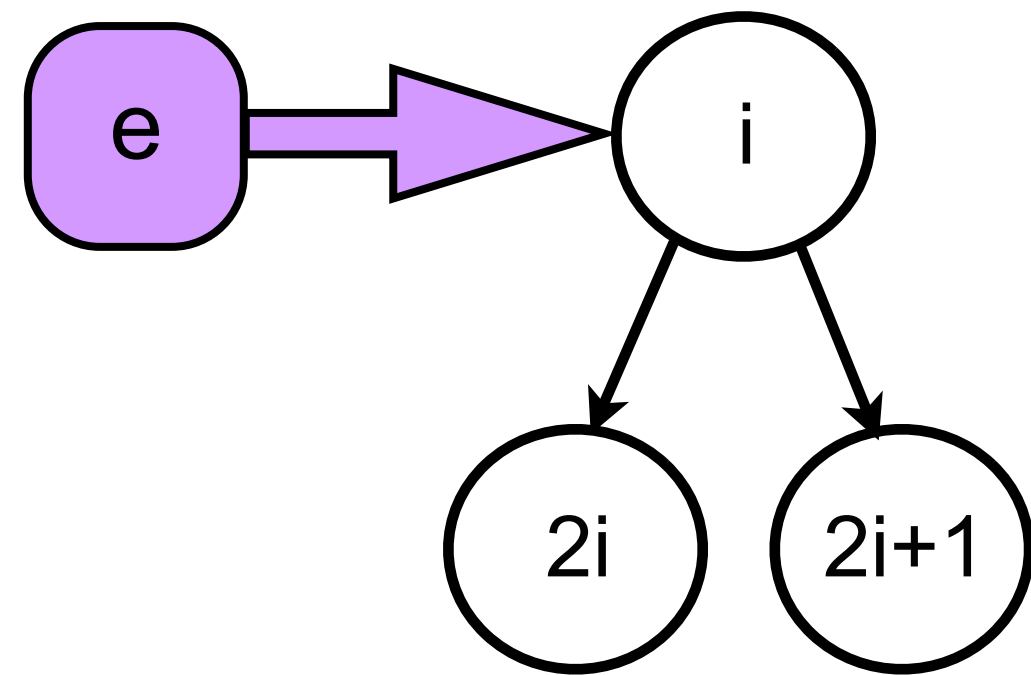
*Als element “e” kleiner is dan het element op positie  $\frac{i}{2}$ , stop dat van  $\frac{i}{2}$  dan in  $i$  en zeef “e” hoger in  $\frac{i}{2}$ .*

*Anders stop je e in  $i$  en zijn we klaar.*

```
(define (sift-up heap idx)
  (let
    ((vector-ref
      (lambda (v i)
        (vector-ref v (- i 1)))))
    (vector-set!
      (lambda (v i a)
        (vector-set! v (- i 1) a)))
    (vector (storage heap))
    (size (size heap))
    (<<? (lesser heap)))
    (let sift-iter
      ((child idx)
       (element (vector-ref vector idx)))
      (let ((parent (div child 2)))
        (cond ((= parent 0)
              (vector-set! vector child element))
              ((<<? element (vector-ref vector parent))
               (vector-set! vector child (vector-ref vector parent))
               (sift-iter parent element))
              (else
               (vector-set! vector child element))))))))
```



# Implementatie: Neerwaarts ziften



*Als  $e$  kleiner is dan de twee kinderen zijn we klaar. Stop “ $e$ ” op positie  $i$ .*

*Bepaal anders de index  $j$  die het kleinste element van de twee bevat. Stop het element op  $j$  in positie  $i$  en zeef “ $e$ ” dieper in  $j$*

```
(define (sift-down heap idx)
  (let
    (...)
    (let sift-iter
      ((parent idx)
       (element (vector-ref vector idx)))
      (let* ((childL (* 2 parent))
              (childR (+ (* 2 parent) 1))
              (smallest
               (cond ((< childL size)
                      (if (<=? (vector-ref vector childL)
                                (vector-ref vector childR))
                          (if (<=? element (vector-ref vector childL))
                              parent
                              childL)
                          (if (<=? element (vector-ref vector childR))
                              parent
                              childR)))
                      ((= childL size)
                       (if (<=? element (vector-ref vector childL))
                           parent
                           childL))
                      (else parent))))
              (if (not (= smallest parent))
                  (begin (vector-set! vector parent (vector-ref vector smallest))
                          (sift-iter smallest element))
                  (vector-set! vector parent element)))))))
```





# Worst-case Performantie

*Opwaarts ziften: vertrek van het laatste element. Hoe dikwijls kan je  $n$  halveren eer je op 1 uitkomt:  $\log_2(n)$  keer*

*Neerwaarts ziften: vertrek van het eerste element. Hoe dikwijls kan je 1 verdubbelen eer je op  $n$  uitkomt:  $\log_2(n)$  keer*

## Conclusie:

*In beide procedures wordt sift-iter maximaal  $\log_2(n)$  keer uitgevoerd.*

*Dus zijn delete! en insert! in  $O(\log(n))$*



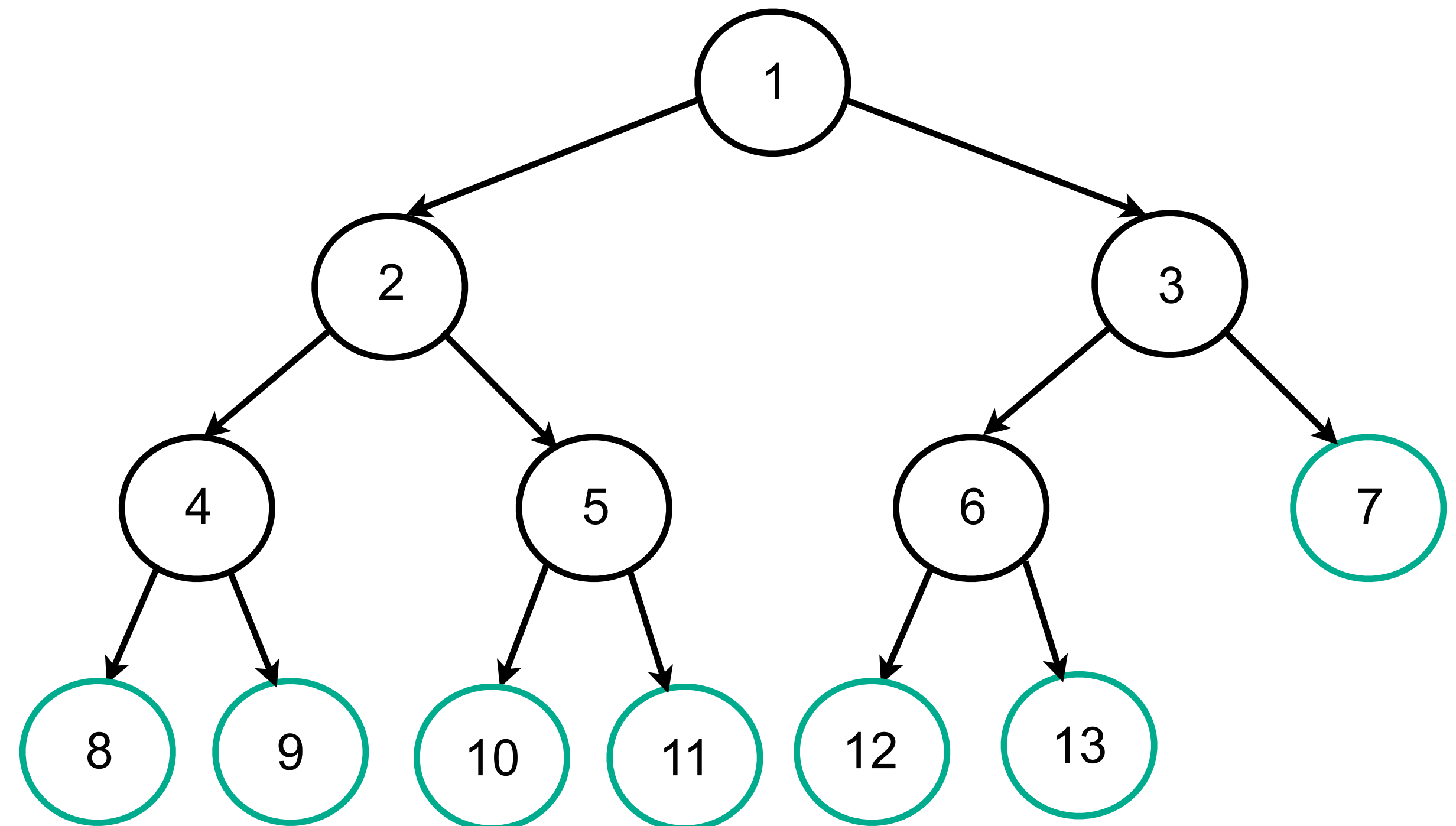
# Een heap bouwen uit een vector: Heapify

```
(define (from-scheme-vector vector <<?)  
  (define size (vector-length vector))  
  (define heap (make vector size <<?))  
  (define (iter index)  
    (sift-down heap index)  
    (if (> index 1)  
        (iter (- index 1))))  
  (iter (div size 2))  
  heap)
```



De *laatste [helft]* van de vector zijn bladeren: triviale heaps van grootte 1

Hebben we  
bewezen



# Performantie van Heapify

```
(define (from-scheme-vector vector <<?)  
  (define size (vector-length vector))  
  (define heap (make vector size <<?))  
  (define (iter index)  
    (sift-down heap index)  
    (if (> index 1)  
        (iter (- index 1))))  
  (iter (div size 2))  
  heap)
```



*Eerste Poging*

sift-down is in  $O(\log(n))$

*Dat doen we  $n/2$  keer.*

*Dus  $O(n \cdot \log(n))$ .*

*Tweede Poging*

*Op hoogte  $h$  kost sift-down  $O(h)$  werk.*

*Op hoogte  $h$  zijn er maximaal  $\frac{n}{2^{h+1}}$  knopen.*

*Dus op hoogte  $h$  is er  $O(\frac{h \cdot n}{2^{h+1}})$  werk te doen.*

*Dat moet gebeuren voor alle hoogten  $h$  tussen 0 en  $\log_2 n$ .*

$$\text{Dus } O\left(n \cdot \sum_{h=0}^{\log_2(n)} \frac{h}{2^{h+1}}\right).$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

*Dus  $O(n)$*

# Conclusie Heaps

	heap
new	$O(1)$
empty?	$O(1)$
full?	$O(1)$
from-vector	$O(n)$
insert!	$O(\log(n))$
delete!	$O(\log(n))$
peek	$O(1)$
size	$O(1)$

# Revenons à nos moutons

Ongewijzigd

```
(define pq-item-make cons)
(define pq-item-val car)
(define pq-item-priority cdr)
(define (lift func)
  (lambda (item1 item2)
    (func (pq-item-priority item1)
          (pq-item-priority item2)))))
```



*We waren priority queues aan het implementeren...*

$O(\log(n))$

```
(define (serve! pq)
  (if (empty? pq)
      (error "empty priority queue (serve!)" pq))
      (pq-item-val (heap:delete! (heap pq)))))
```

```
(define (peek pq)
  (if (empty? pq)
      (error "empty priority queue (peek)" pq))
      (pq-item-val (heap:peek (heap pq)))))
```

```
(define (enqueue! pq value pty)
  (heap:insert! (heap pq) (pq-item-make value pty)
                pq))
```

$O(\log(n))$



```
(define-record-type priority-queue
  (make h)
  priority-queue?
  (h heap))
```

Slechts 1  
veldje: de heap

```
(define default-size 50)
```

```
(define (new >>?)
  (make (heap:new default-size (lift >>?))))
```



# Priority Queues: Conclusie

	sorted-list	position-list	heap
new	$O(1)$	$O(1)$	$O(1)$
empty?	$O(1)$	$O(1)$	$O(1)$
full?	$O(1)$	$O(1)$	$O(1)$
priority-queue?	$O(1)$	$O(1)$	$O(1)$
peek	$O(1)$	$O(n)$	$O(1)$
enqueue!	$O(n)$	$O(1)$	$O(\log(n))$
serve!	$O(1)$	$O(n)$	$O(\log(n))$

*Dit is heel goed nieuws. (Denk terug aan de tabel met getallen van hoofdstuk 1!)*



# Hoofdstuk 4

## 4.1 Stacks

- 4.1.1 Het stack ADT
- 4.1.2 De vector implementatie
- 4.1.3 De gelinkte implementatie

## 4.2 Queues

- 4.2.1 Het Queue ADT
- 4.2.3 De Gelinkte Implementatie
- 4.2.4 De Vector Implementatie

## 4.3 Prioriteitenqueues

- 4.3.1 Het Priority Queue ADT
- 4.3.2 De Sorted List Implementatie
- 4.3.3 De Position List Implementatie

## 4.4 Heaps

- 4.4.1 Wat is een heap?
- 4.4.2 Eigenschappen van heaps
- 4.4.3 Het Heap ADT
- 4.4.7 De heap bouwen
- 4.4.8 Priority Queues en Heaps

