



Hoofdstuk 4

Lineaire ADT's

Lineaire Structuren vs. ADT's

Vorige hoofdstuk: Data is lineair gestructureerd in het geheugen

Dit hoofdstuk: ADT's die een lineair gedrag vertonen

De implementatie ervan gebruikt niet noodzakelijk een lineaire datastructuur

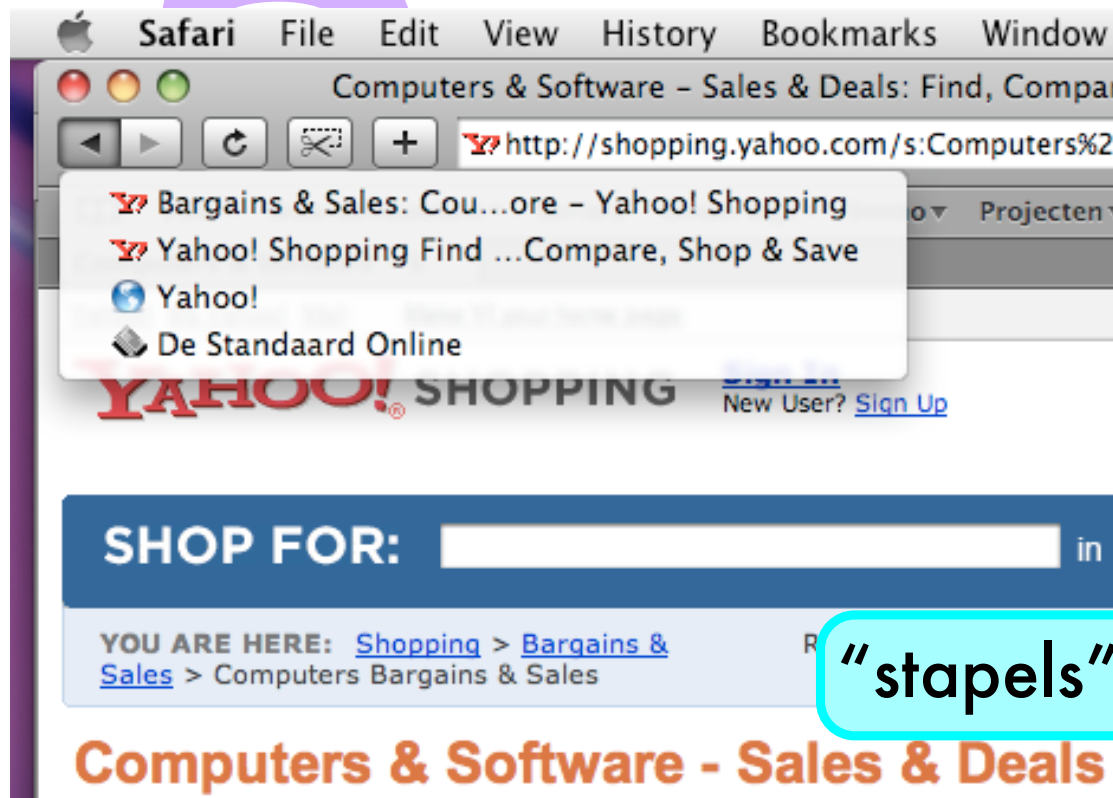
Stacks

Queues

Prioriteitenqueues

ADT #1: Stacks

LIFO Gedrag



“stapels” in het Nederlands

Het Stack-ADT

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)

top leest de top maar
verwijdert hem niet

pop! leest de top en
verwijdert hem

Vectoriële Stack Implementatie

representatie

```
(library
  (stack)
  (export new stack? push! pop! top empty? full?)
  (import (rnrs base (6))
    (prefix (a-d positional-list adt) plist:))

  (define stack-tag 'vector-stack)
  (define (make)
    (list stack-tag (plist:new eq?)))
  (define (plist stack)
    (cadr stack))

  (define (new)
    (make))

  (define (stack? any)
    (and (pair? any)
      (eq? (car any) stack-tag)))

  ...)
```

verificatie

```
(define (empty? stack)
  (plist:empty? (plist stack)))

(define (full? stack)
  (plist:full? (plist stack)))
```

Vectoriële Stack Implementatie

manipulatie

```
(define (push! stack val)
  (plist:add-after! (plist stack) val)
  stack)

(define (top stack)
  (define plst (plist stack))
  (if (= (plist:length plst) 0)
      (error "stack empty (top)" stack)
      (plist:peek plst (plist:last plst))))

(define (pop! stack)
  (define plst (plist stack))
  (if (= (plist:length plst) 0)
      (error "stack empty (pop)" stack)
      (let ((val (plist:peek
                  plst
                  (plist:last plst)))))
        (plist:delete!
         plst (plist:last plst))
        val)))
```

Performantie

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

**Betrouw op best-case
analyse van add-after!
en delete!**

**Zeer lage
flexibiliteit**

Gelinkte Implementatie

representatie

```
(library
  (stack)
  (export new stack? push! pop! top empty? full?)
  (import (except (rnrs base (6)) map list length for-each)
    (rnrs io simple)
    (rnrs mutable-pairs)
    (prefix (a-d positional-list adt) plist:))

  (define stack-tag 'linked-stack)
  (define (make)
    (cons stack-tag (plist:new eq?)))
  (define (plist stack)
    (cdr stack))

  (define (new)
    (make))

  (define (stack? any)
    (and (pair? any)
         (eq? (car any) stack-tag)))

  ...)
```

verificatie

```
(define (empty? stack)
  (define plst (plist stack))
  (plist:empty? plst))

(define (full? stck)
  (define plst (plist stck))
  (plist:full? plst))
```


Gelinkte Implementatie

manipulatie

```
(define (push! stack val)
  (plist:add-before! (plist stack) val)
  stack)

(define (top stack)
  (define plst (plist stack))
  (if (= (plist:length plst) 0)
      (error "stack empty (top)" stack)
      (plist:peek plst (plist:first plst))))

(define (pop! stack)
  (define plst (plist stack))
  (define first-position (plist:first plst))
  (if (= (plist:length plst) 0)
      (error "stack empty (pop)" stack)
      (let ((val (plist:peek plst first-position)))
        (plist:delete! plst first-position)
        val)))
```

Performantie

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

**Betrouw op best-case
analyse van add-before!
en van delete!**

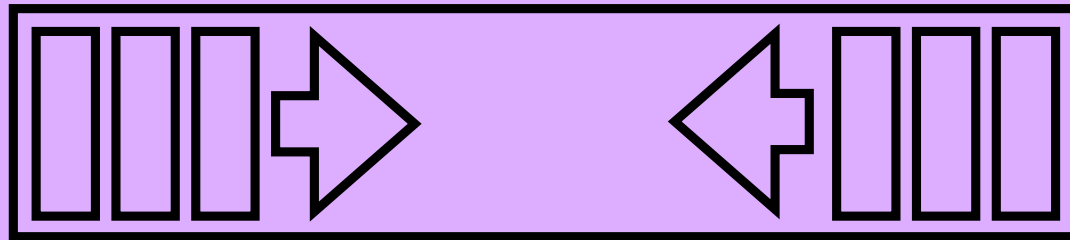
**Zeer hoge
flexibiliteit**

Stacks: Overzicht

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”.

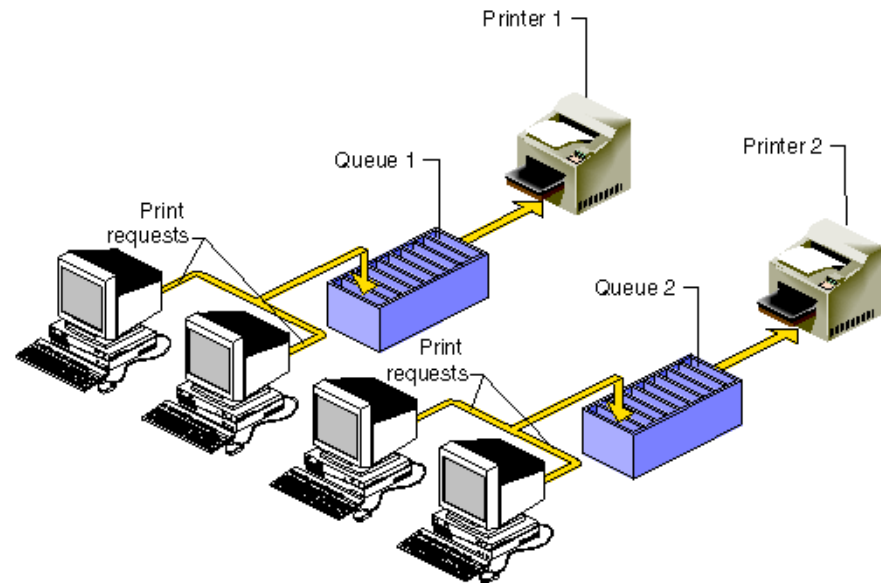
Variante: Stackparen



Kan de slechte "flexibiliteit" van vector-gebaseerde stacks enigszins counteren.

Een implementatie op basis van positionele lijsten is niet meer mogelijk.

ADT #2: Queues



FIFO Gedrag

“wachtrijen” in het Nederlands

Het Queue-ADT

ADT queue

new

($\emptyset \rightarrow$ 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

Positionele lijsten kunnen niet meer gebruikt worden.

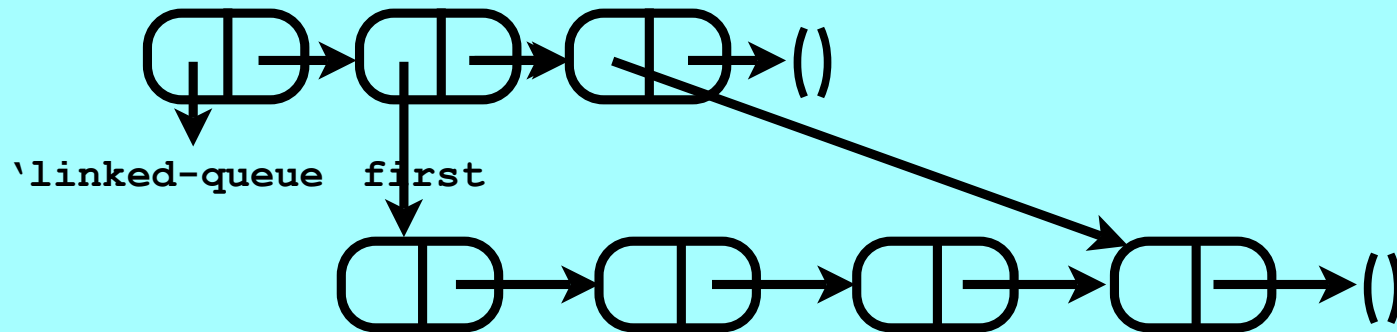
In de vector implementatie wordt ofwel `serve!` ofwel `enqueue!` een procedure in $O(n)$

In de gelinkte implementaties krijgen we hetzelfde fenomeen om een andere reden.

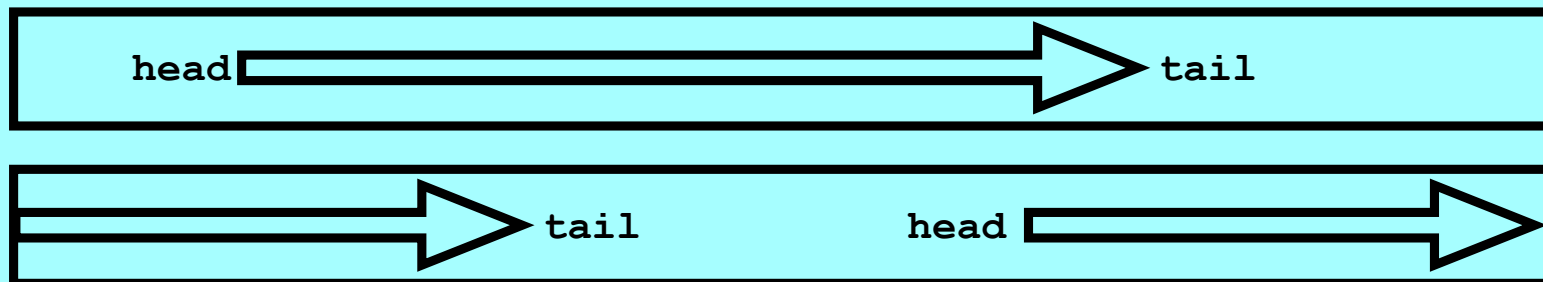
De enhanced dubbelgelinkte implementatie geeft $O(1)$ maar dit is omwille van de `last`.

Queues: Implementatiestrategieën

Gelinkte Implementatie: enkel gelinkt met rear



Vectoriële Implementatie: met circulaire vector



Gelinkte Queue Implementatie

```
(library
  (queue)
  (export new queue? serve! enqueue!
          peek full? empty?)
  (import (rnrs base (6))
          (rnrs mutable-pairs)))

(define queue-tag 'linked-queue)
(define (make)
  (list queue-tag '() '()))
(define (head q)
  (cadr q))
(define (head! q node)
  (set-car! (cdr q) node))
(define (rear q)
  (caddr q))
(define (rear! q node)
  (set-car! (cddr q) node))

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

representatie

```
(define (make-queue-node val next)
  (cons val next))
(define (queue-node-val node)
  (car node))
(define (queue-node-next node)
  (cdr node))
(define (queue-node-next! node next)
  (set-cdr! node next))
```

Gelinkte Queue Implementatie

manipulatie

```
(define (enqueue! q val)
  (define last (rear q))
  (define node (make-queue-node val '()))
  (if (null? (head q))
      (head! q node)
      (queue-node-next! last node))
  (rear! q node)
  q)

(define (peek q)
  (if (null? (head q))
      (error "empty queue (peek)" q)
      (queue-node-val (head q))))

(define (serve! q)
  (define first (head q))
  (if (null? first)
      (error "empty queue (serve!)" q)
      (head! q (queue-node-next first)))
  (if (null? (head q))
      (rear! q '()))
  (queue-node-val first))
```

Even tussendoor...

De rest bij deling
van x door y

$(\text{mod } x \ y)$

$(\text{mod } 0 \ 5)$
 $(\text{mod } 1 \ 5)$
 $(\text{mod } 2 \ 5)$
 $(\text{mod } 3 \ 5)$
 $(\text{mod } 4 \ 5)$
 $(\text{mod } 5 \ 5)$
 $(\text{mod } 6 \ 5)$
 $(\text{mod } 7 \ 5)$
 $(\text{mod } 8 \ 5)$
 $(\text{mod } 9 \ 5)$
 $(\text{mod } 10 \ 5)$
 $(\text{mod } 11 \ 5)$
 $(\text{mod } 12 \ 5)$
 $(\text{mod } 13 \ 5)$
...

0
1
2
3
4
0
1
2
3
4
0
1
2
3
...

Vectoriële Queue Implementatie

```
(library (queue)
  (export new queue? enqueue! serve!
    peek full? empty?)
  (import (rnrs base (6))
    (rnrs mutable-pairs)))

(define queue-tag 'vector-queue)
(define default-size 5)
(define (make size)
  (list queue-tag (make-vector size) 0 0))
(define (storage q)
  (cadr q))
(define (head q)
  (caddr q))
(define (head! q head)
  (set-car! (caddr q) head))
(define (tail q)
  (caddr q))
(define (tail! q tail)
  (set-car! (caddr q) tail))

(define (new)
  (make default-size))
...)
```

representatie

verificatie

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

(define (full? q)
  (= (mod (+ (tail q) 1) default-size)
    (head q)))
```

Vectoriële Queue Implementatie

manipulatie

```
(define (enqueue! q val)
  (if (full? q)
      (error "full queue (enqueue!)" q))
      (let ((new-tail (mod (+ (tail q) 1) default-size)))
        (vector-set! (storage q) (tail q) val)
        (tail! q new-tail))
      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 (mod (+ (head q) 1) default-size))
        result))
```

Queue Implementaties: Overzicht

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” en “geheugenverbruik”.

ADT #3: Prioriteitenqueues

© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com



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

PATIENTS
WAITING ROOM

3 AILMENTS
OR LESS

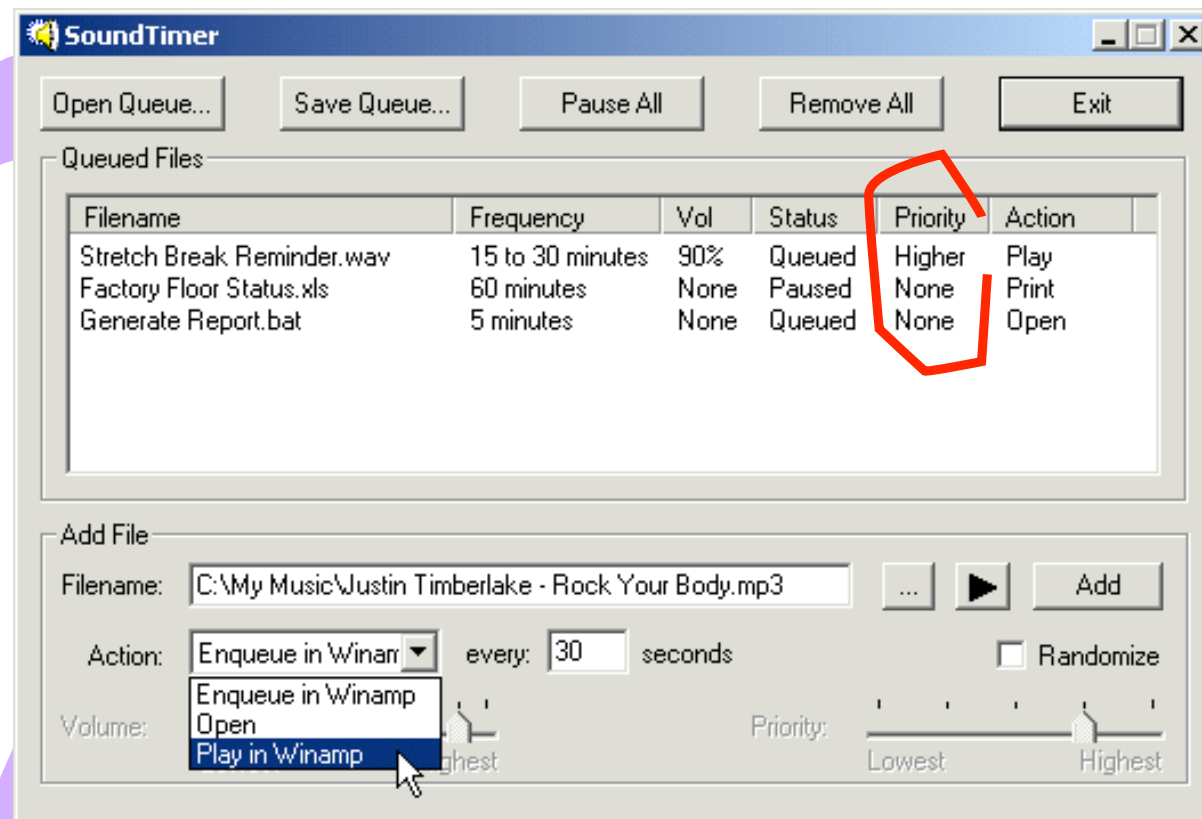


© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com

HPFO Gedrag

"prioriteitenwachtrijen" in het Nederlands

Voorbeeld: Task Scheduling



Het Priority Queue ADT

```
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 )
```

peek leest de kop maar
verwijdert hem niet

serve! leest de kop en
verwijdert hem

new verwacht >>?

Implementaties

Een eerste implementatie is gebaseerd op `sorted-lists`

Een tweede implementatie is gebaseerd op `position-lists`

De opgeslagen "values" zijn "priority queue items"

= waarde + prioriteit

Priority Queue Items

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

(define (pq-item-specialize-for-priorities func)
  (lambda (item1 item2)
    (func (pq-item-priority item1)
          (pq-item-priority item2))))
```

```
(define << (pq-item-specialize-for-priorities >))
```

```
(define item1 (pq-item-make "Wolf" 1))
(define item2 (pq-item-make "Vivi" 2))
(<< item1 item2)
```

Sorted List Implementatie

representatie

```
(library (priority-queue)
  (export new priority-queue? enqueue! serve! peek full? empty?)
  (import (except (rnrs base) list)
    (prefix (a-d sorted-list linked) slist:))

  (define pq-tag 'sorted-list-pq)
  (define (make >>?)
    (cons pq-tag
      (slist:new (pq-item-specialize-for-priorities >>?)
        (pq-item-specialize-for-priorities eq?))))
  (define (slist pq)
    (cdr pq))

  (define (new >>?)
    (make >>?))
  ...)
```

Sorted List Implementatie

$O(n)$

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

manipulatie

```
(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)))
```

$O(1)$

Position List Implementatie

```
(library (priority-queue)
  (export new priority-queue? enqueue! peek serve! full? empty?)
  (import (except (rnrs base (6)) list)
    (prefix (a-d positional-list adt) plist:)))

(define pq-tag 'positional-list-pq)
(define (make >>?)
  (cons pq-tag
    (cons (plist:new eq?)
      (pq-item-specialize-for-priorities >>?))))
(define (greater pq)
  (cddr pq))
(define (plist pq)
  (cadr pq))

(define (new >>?)
  (make >>?))
...)
```

representatie

$O(1)$

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

manipulatie


Position List Implementatie

manipulatie

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

$O(n)$

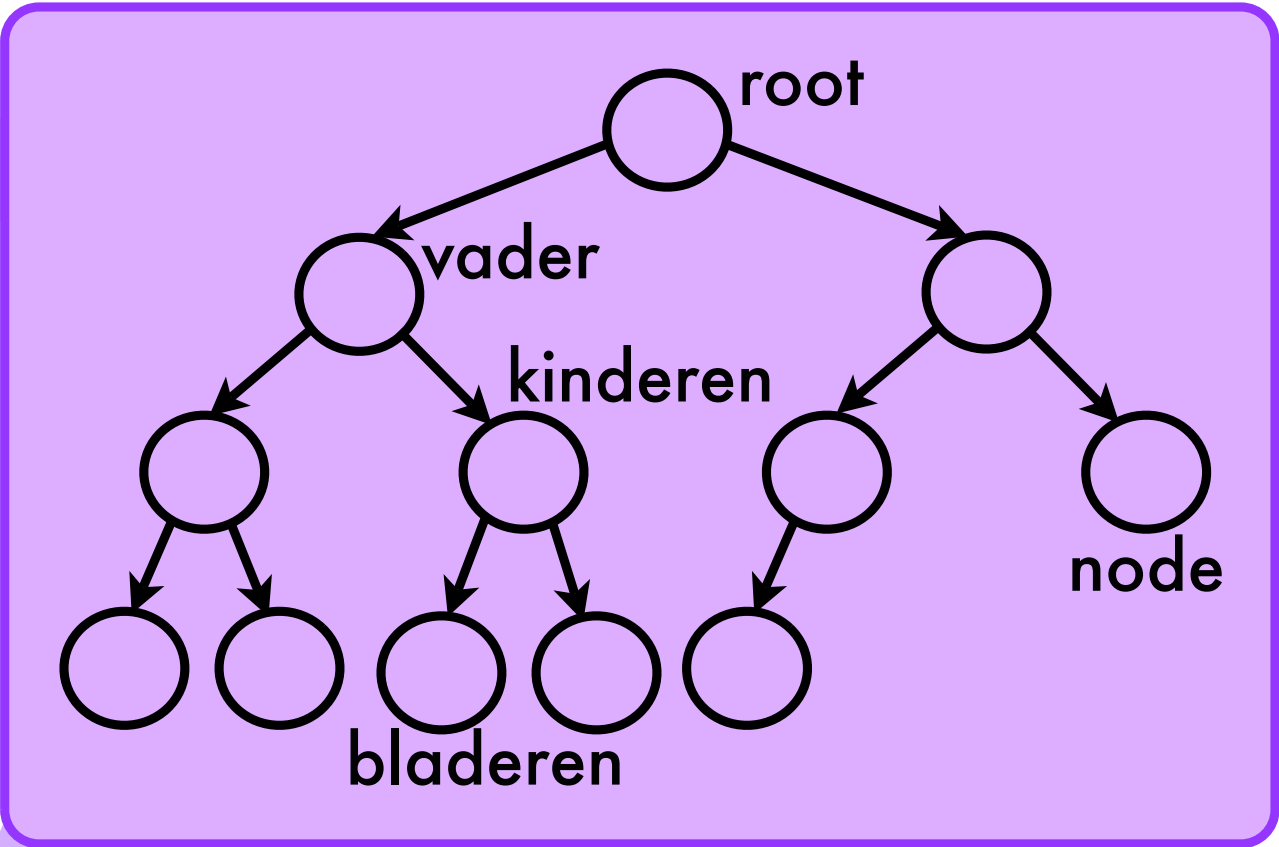
Priority Queues: Overzicht



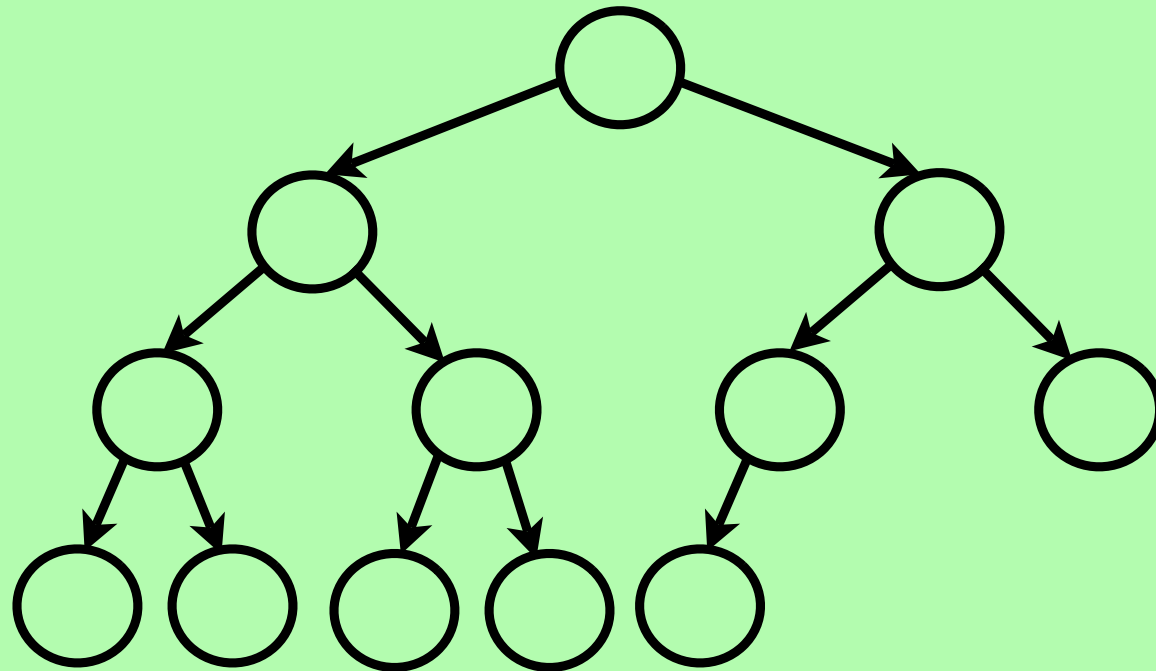
	sorted-list	position-list
new	$O(1)$	$O(1)$
empty?	$O(1)$	$O(1)$
full?	$O(1)$	$O(1)$
priority-queue?	$O(1)$	$O(1)$
enqueue!	$O(n)$	$O(1)$
serve!	$O(1)$	$O(n)$
peek	$O(1)$	$O(n)$

Dit is heel slecht nieuws: een priority queue is geen stabiele datastructuur.

Bomen: Terminologie

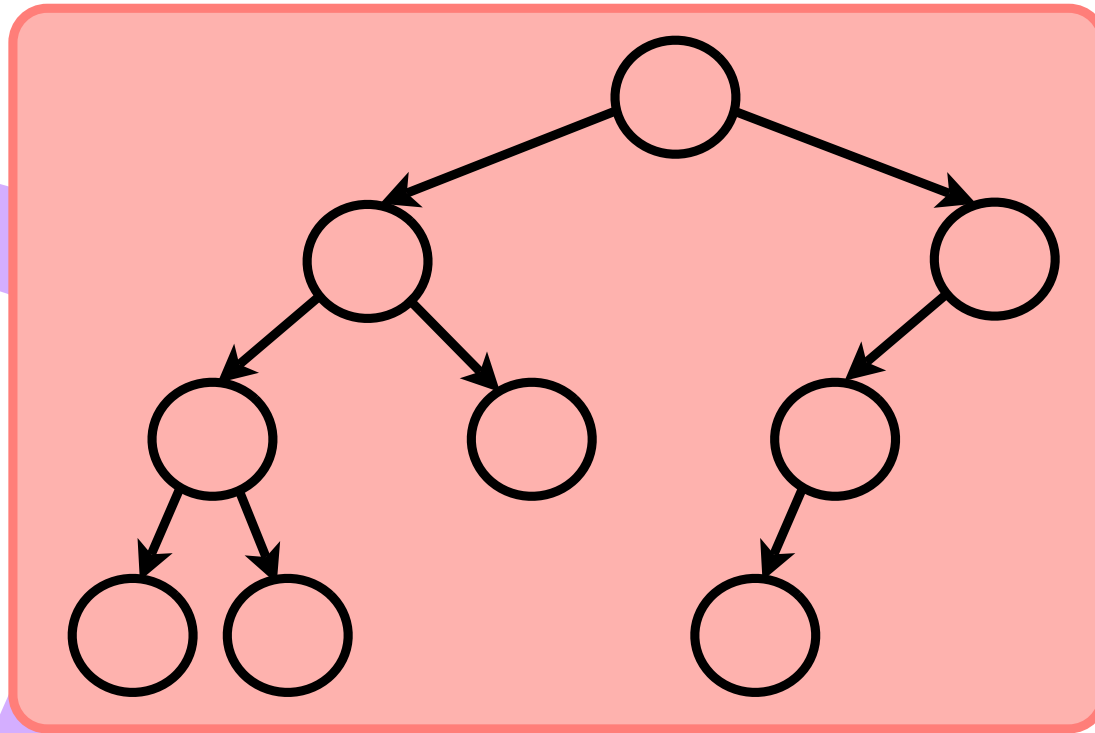


Volledige Bomen



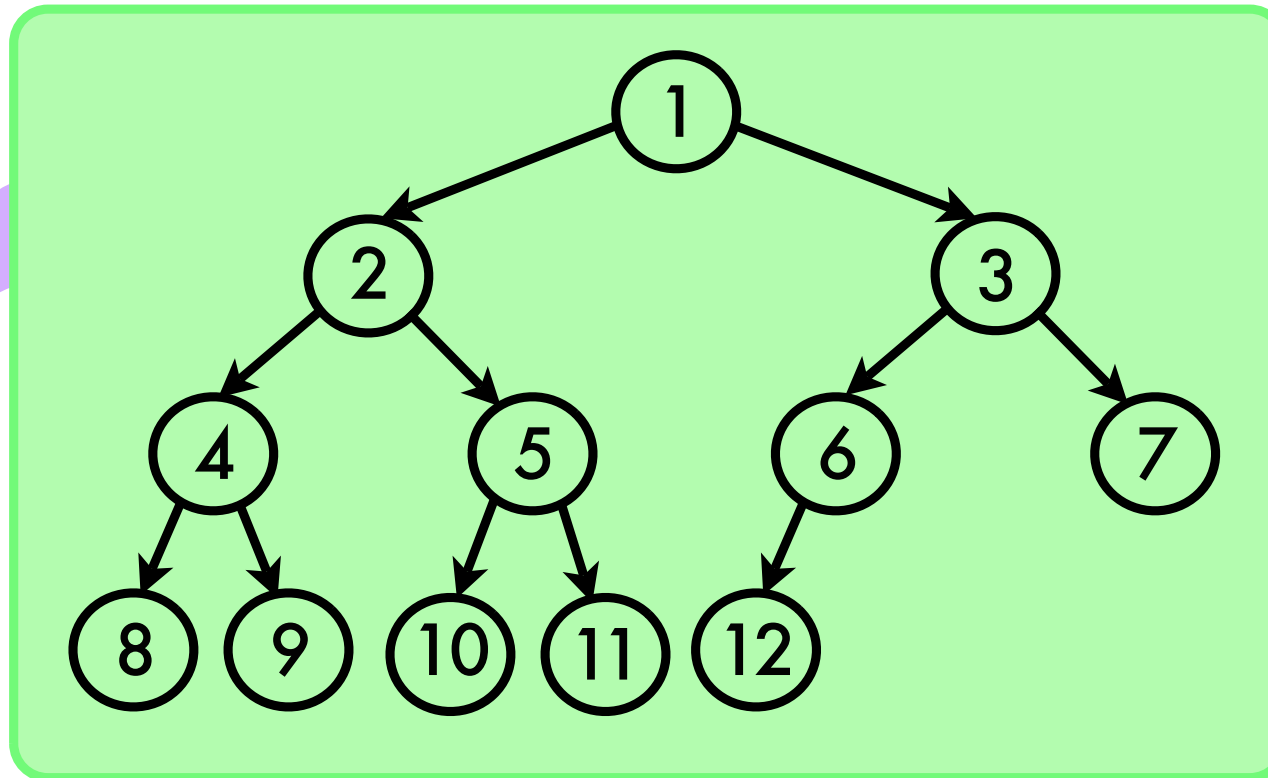
Een volledige boom heeft geen gaten wanneer we hem van links naar rechts lezen, laag per laag

Volledige Bomen



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

Volledige Bomen: Eigenschap



Een volledige boom kan je eenduidig nummeren van 1 tot n

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

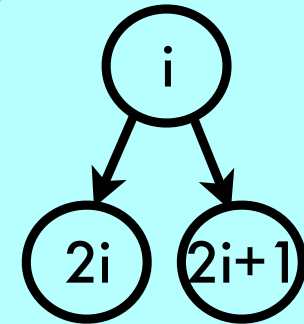
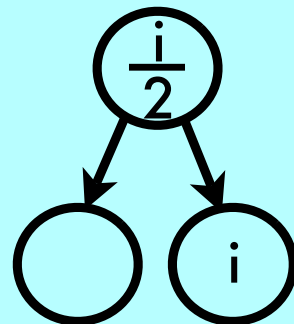
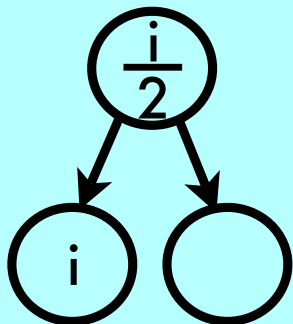
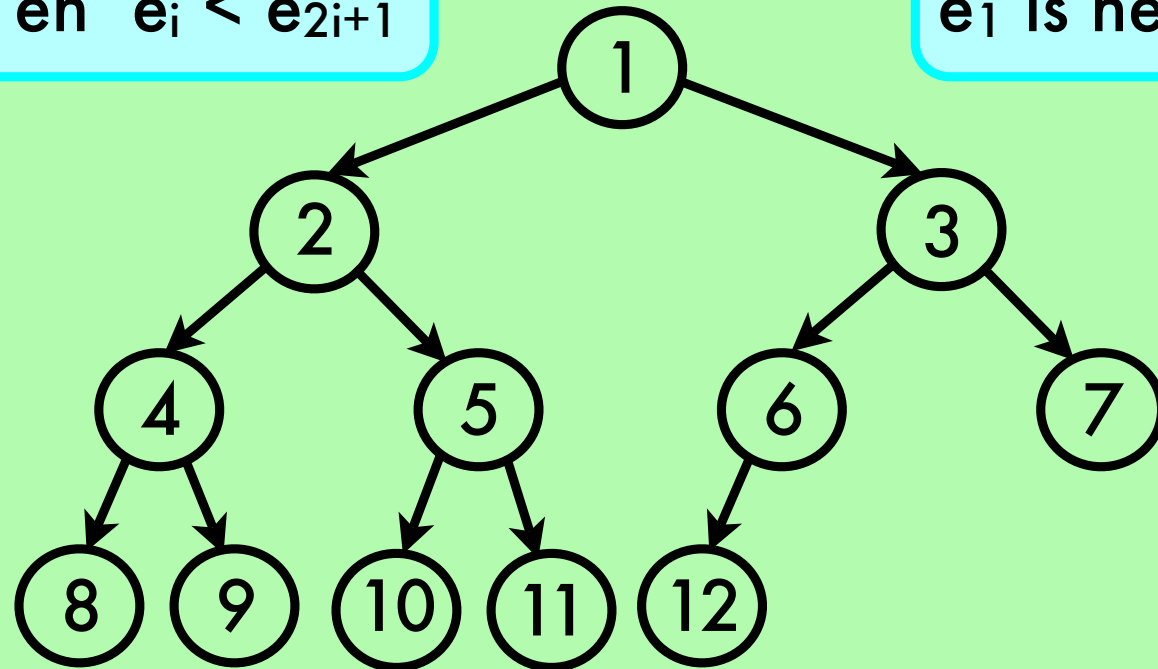
Niet noodzakelijk gesorteerd

Een gesorteerde rij is wel een heap natuurlijk

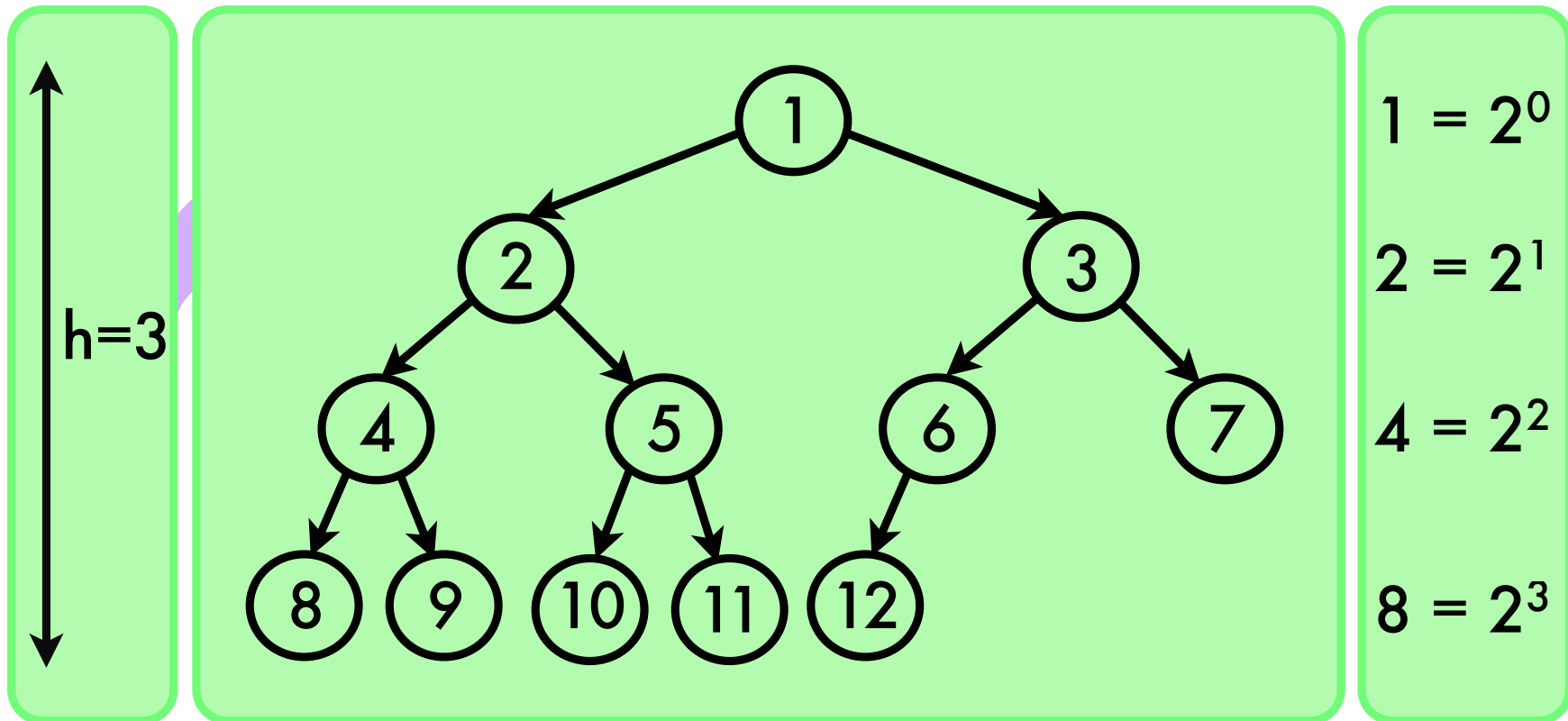
Een Heap als Volledige Boom

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

e_1 is het kleinste



Hoogte vs. aantal elementen



$$h = \lfloor \log_2(n) \rfloor$$

Er bevinden zich maximum $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ knopen op hoogte h

Het Heap ADT

ADT heap < V >

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

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

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

new verwacht <<?

Heap: Implementatie

representatie

```
(library (heap)
  (export new from-scheme-vector full? empty?
          insert! delete! peek length)
  (import (except (rnrs base (6)) length)
          (rnrs mutable-pairs))

  (define (make vector size <<?)
    (list 'heap vector size <<?))
  (define (storage heap)
    (cadr heap))
  (define (storage! heap vector)
    (set-car! (cdr heap) vector))
  (define (size heap)
    (caddr heap))
  (define (size! heap s)
    (set-car! (cddr heap) s))
  (define (lesser heap)
    (caddr heap))

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

Heap: Implementatie

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

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

(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))))

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

Het eerste element is steeds het kleinste in de heap

Heap: Toevoegen en verwijderen

Herinner de heap conditie:

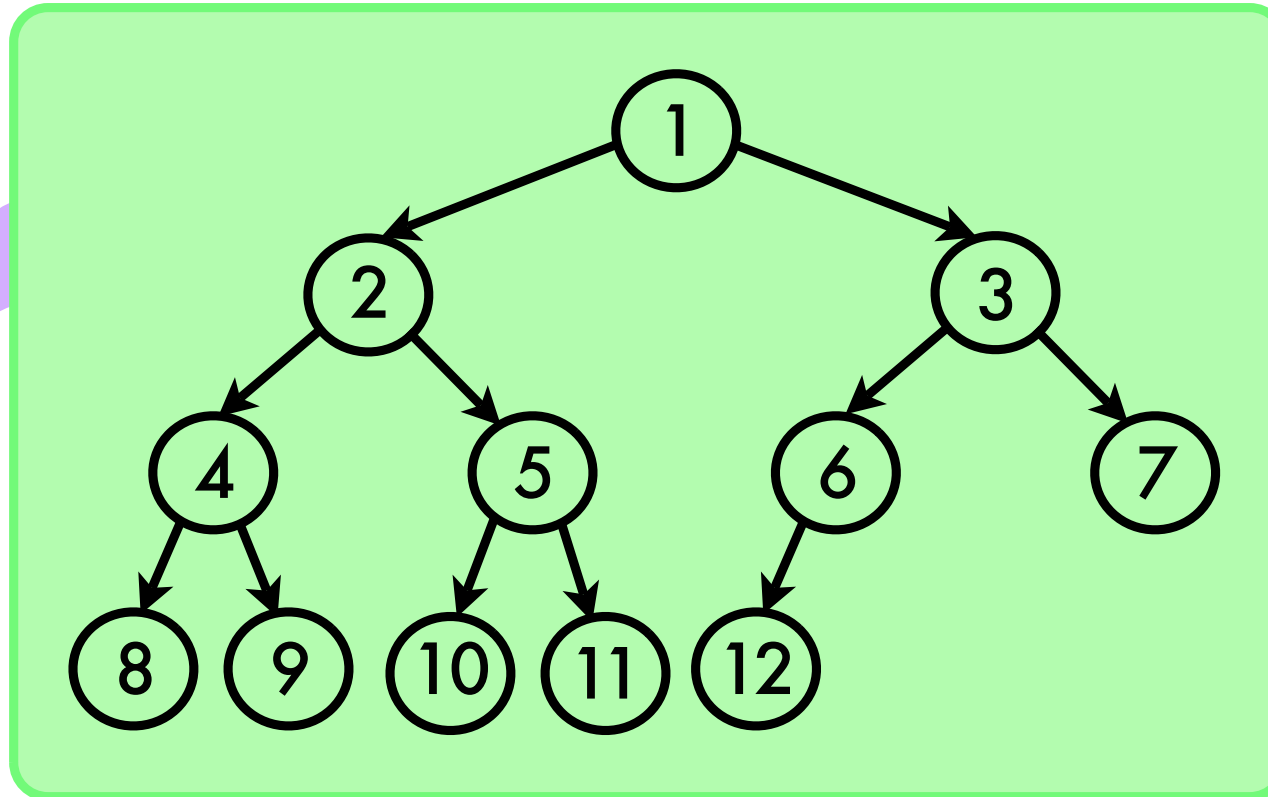
$$e_i < e_{2i} \text{ en } e_i < e_{2i+1}$$

Je kan niet zomaar een element invoegen in de heap

Je kan niet zomaar een element wegknippen uit de heap

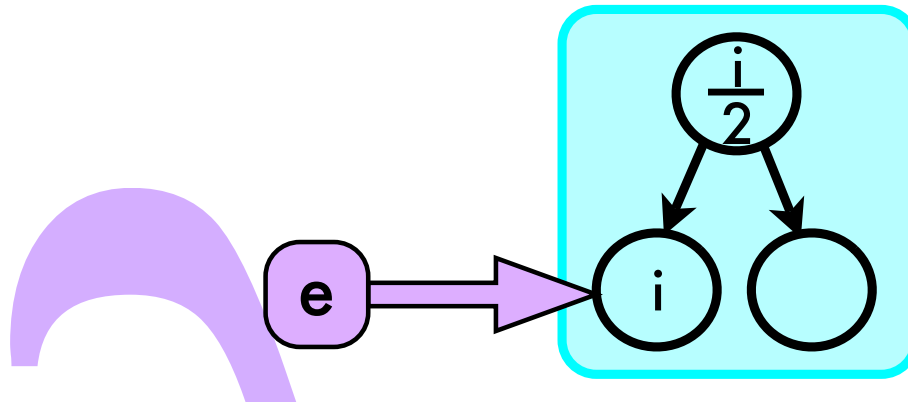
Gedisciplineerd toevoegen en verwijderen

Gedisciplineerd Toevoegen



Zet een nieuw element op de laatste plaats (13) en "zeef" dat tot op de juiste plaats

Opwaarts zeven van "e"

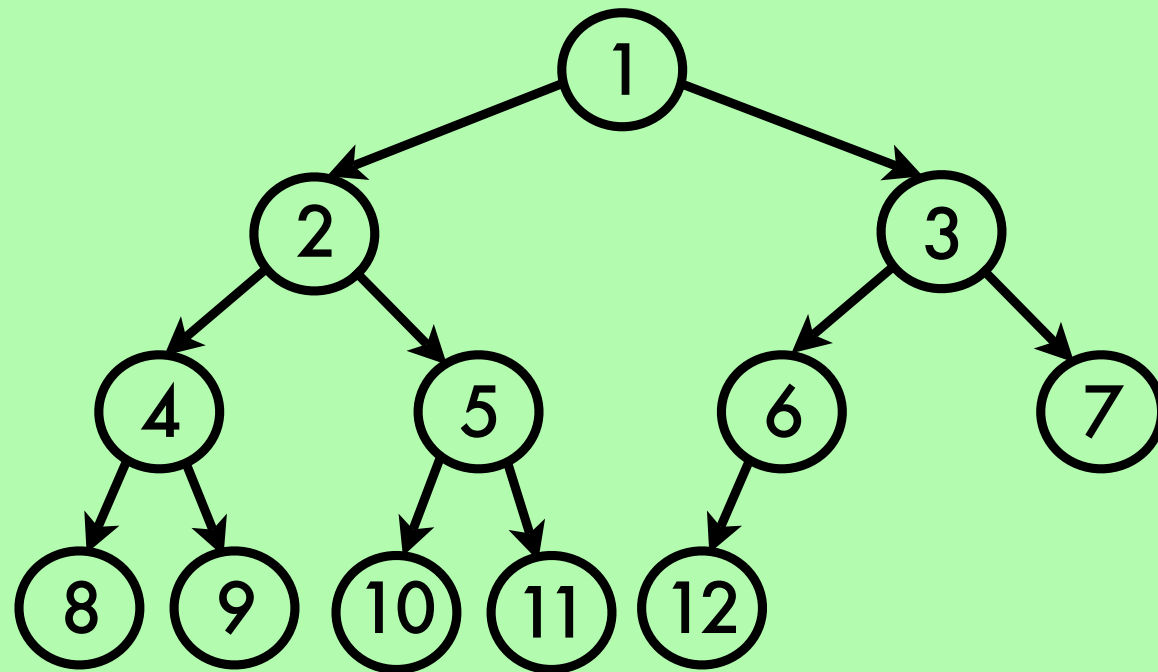


Als element i de root is zijn we klaar. Stop "e" in de root

Als element "e" kleiner is dan $\frac{i}{2}$, stop $\frac{i}{2}$ dan in i en zeef "e" in $\frac{i}{2}$

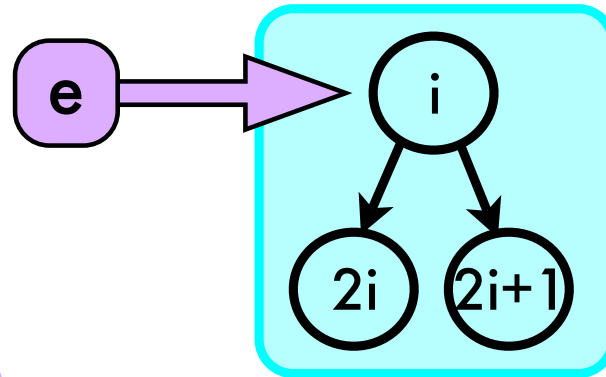
Anders stop je e in i

Gedisciplineerd Verwijderen



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

Neerwaarts zeven van "e"



Als element i kleiner is dan de twee andere zijn we klaar. Stop "e" in i

Bepaal anders de index j die het kleinste element van de drie bevat. Wissel i met j en zeef "e" in j

Heap: Implementatie

manipulatie

```
(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))))

(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))
```


Opwaarts zeven

```
(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)))))))
```

Neerwaarts zeven

```
(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))))))
```

Performantie: Worst-case

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

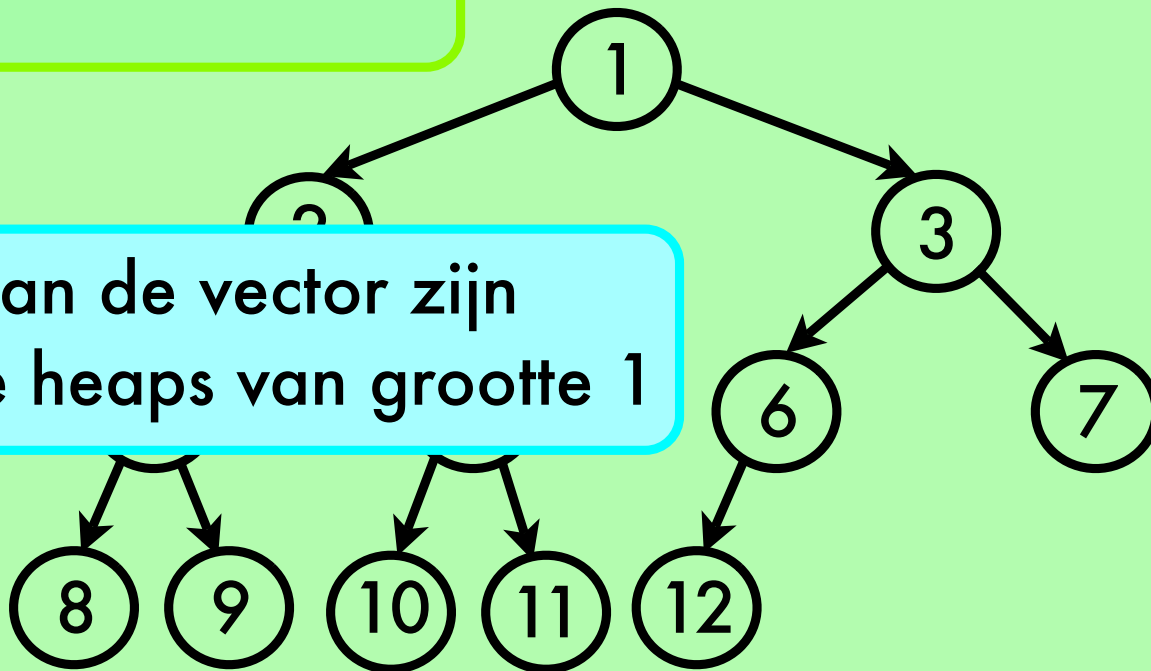
Neerwaarts zeven: vertrek van het eerste element. Hoe dikwijls kan je 1 verdubbelen eer je op n uitkomt eer je op nul uitkomt: $\log_2(n)$ keer

Gevolg: sift-iter wordt maximaal $\log_2(n)$ keer uitgevoerd. Dus zijn delete! en insert! in $O(\log(n))$

Een heap bouwen uit een vector

```
(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



Eerste schatting

```
(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)
```

sift-down is in $O(\log(n))$ en dat
doen we $n/2$ keer. Dus $O(n \cdot \log(n))$.

Een slimmere schatting

```
(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)
```

Op hoogte h kost sift-down $O(h)$ werk. Op hoogte h zijn er maximaal $n/2^{h+1}$ knopen. Dus op hoogte h is er $O(h \cdot n/2^{h+1})$ werk te doen.

Je moet dat werk doen voor alle hoogten h tussen 0 en $\log_2 n$. Dus $O(n \sum^{\log n} h/2^h)$.

$$\sum^{\infty} h/2^h = 2$$

Dus $O(n)$

Conclusie Heaps

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

Implementatie voor priority queues...

```
(library (priority-queue)
  (export new priority-queue? enqueue! peek serve! full? empty?)
  (import (rnrs base (6))
    (prefix (a-d heap standard) heap:))

  (define default-size 50)
  (define pq-tag 'heap-pq)
  (define (make >>?)
    (list pq-tag (heap:new default-size >>?)))
  (define (heap pq)
    (cadr pq))

  (define (new >>?)
    (make (pq-item-specialize-for-priorities >>?)))
  ...)
```

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


De PQ-Operaties in $O(\log(n))$

```
(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))
```

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?	$O(1)$	$O(1)$	$O(1)$
enqueue!	$O(n)$	$O(1)$	$O(\log(n))$
serve!	$O(1)$	$O(n)$	$O(\log(n))$
peek	$O(1)$	$O(n)$	$O(1)$

Dit is heel goed nieuws. Denk terug aan de tabel met waarden 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

