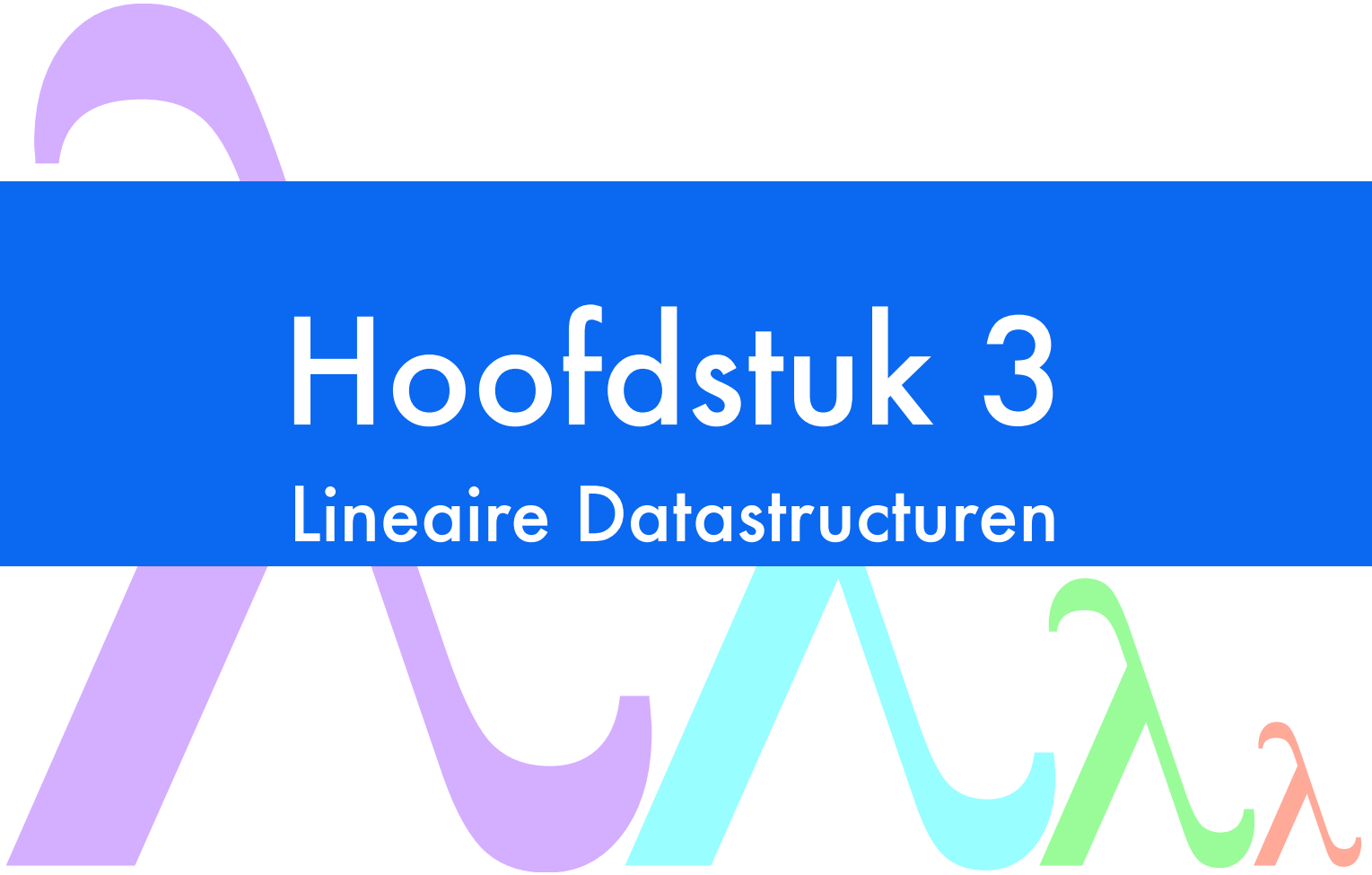
Hoofdstuk 3

Lineaire Datastructuren



Lineariteit

Lineariteit is een zeer natuurlijk begrip

Een rij van boeken op een rek

Een wachtrij in de supermarkt

Een stapel dossiers die te verwerken zijn

Lineariteit

Ook in computertoepassingen

Je ‘contacts’ lijst op MSN

Je playlist in iTunes

De menu-opties onder “edit”

Een lijst van kandidaten op een kieslijst

Lineariteit in Scheme

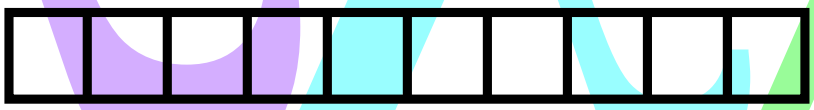
In Scheme kunnen we zeer eenvoudig een lineaire datastructuur bouwen door een reeks data elementen in een lijst te hangen of door ze in een vector te stoppen.

Maar:
- wat zijn de performantiekarakteristieken van de operaties?
- er zijn heel wat nadelen verbonden aan het gebruik van “naakte” lijsten en “naakte” vectoren

Naakte vectoren: problemen

Grootte moet a-priori gekend zijn: veel verspilling + extra variabele nodig per vector in de applicatie: vervuiling van de omgeving

Element toevoegen op (grootte+1) resulteert in een fout of een storage-move



Naakte Scheme lijsten: problemen

"aanconsen" kan enkel vooraan. Op andere plaatsen toevoegen is traag.

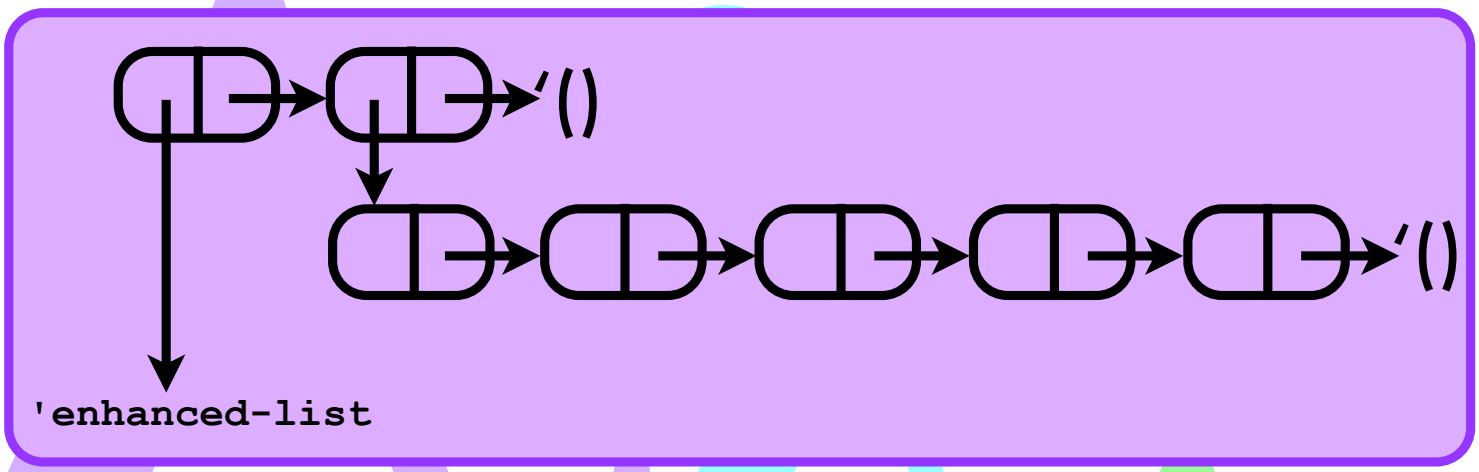
call-by-value steekt stokken in de wielen:

```
(define (add-to-first l e)
  (set! l (cons e l)))
```

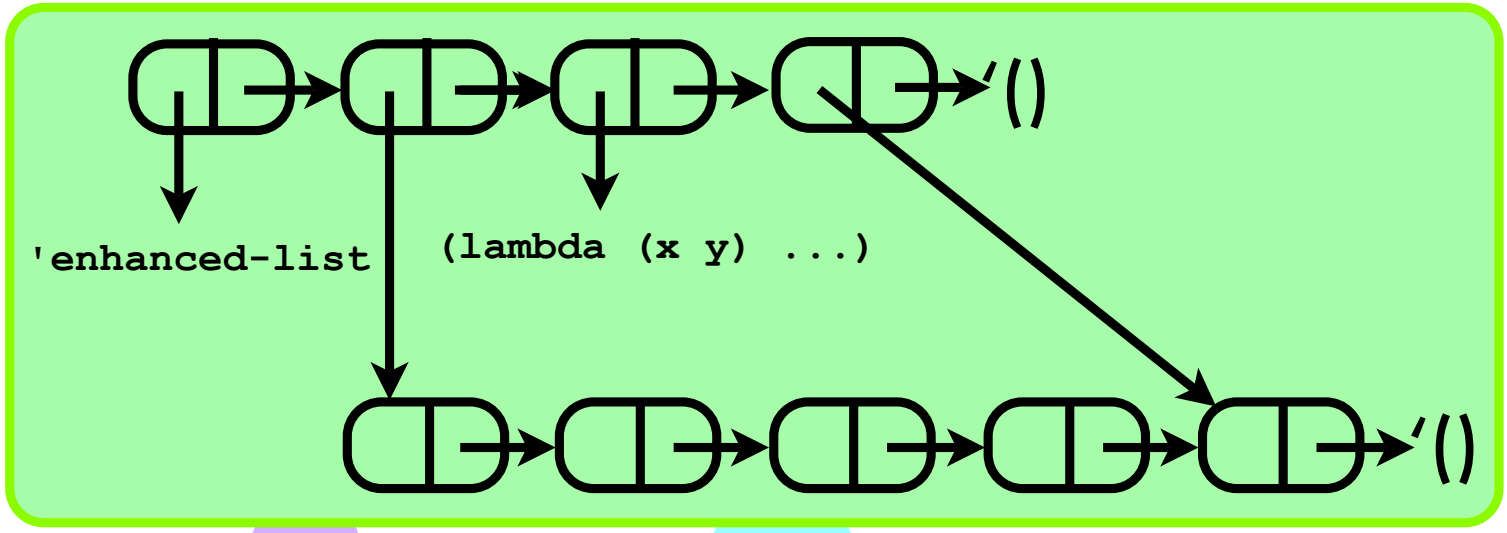
Scheme lijsten zijn niet generisch. member? memv? en memq? gebruiken een vaste "gelijkheid". Hoe zoek je de persoon met een zekere naam in een Scheme lijst?

Oplossing: Enhanced Lists

Een enhanced list is een datastructuur die bestaat uit een header die een verwijzing heeft naar het eerste pair van de eigenlijke lijst.



Wat kan de header zoal doen?



extra informatie bijhouden (snelheid)

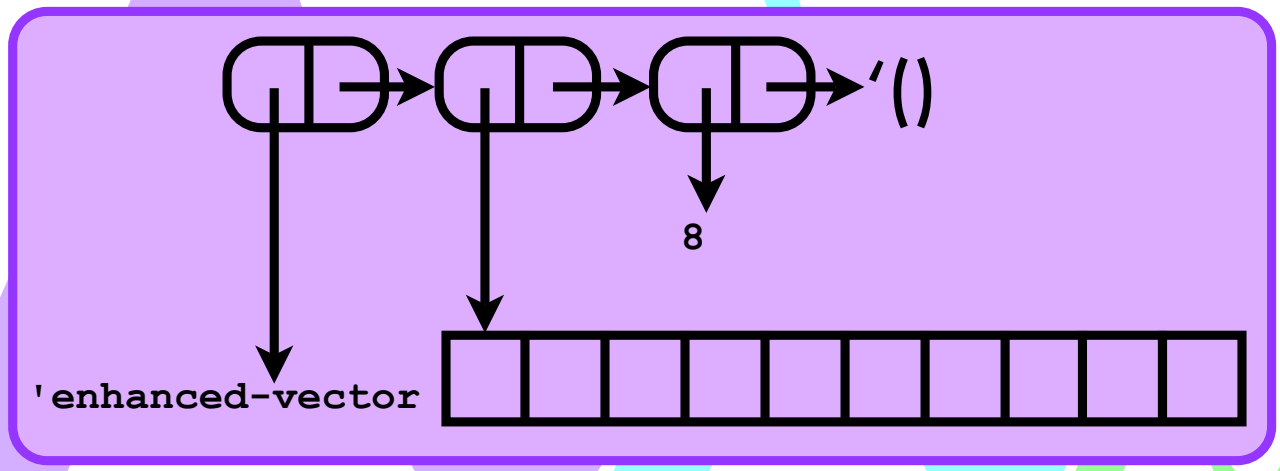
genericiteit garanderen

problemen met wijzigen van het eerste element vermijden

Analoog: enhanced vectoren

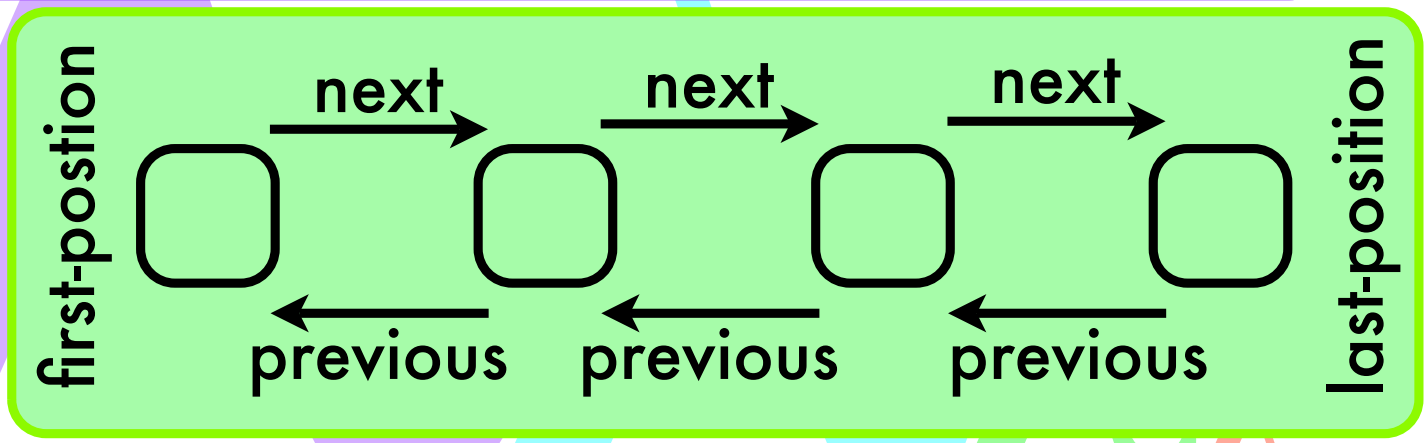
Een enhanced list is een datastructuur die bestaat uit een header die een verwijzing heeft naar het eerste pair van de eigenlijke lijst.

Een enhanced vector is een datastructuur die bestaat uit een header die een verwijzing heeft naar de eigenlijke vector.



Lineariteit: Positionele Lijsten

Een lineaire datastructuur is een verzameling data elementen waarbij elk data element geassocieerd is met een positie. Elke positie heeft een unieke volgende positie en een unieke vorige positie. Er zijn twee speciale posities: de laatste positie heeft geen volgende positie en de eerste positie heeft geen vorige positie.



Positionele Lijsten: Het ADT

```
ADT position-list<V P>

new
    ( (V V → boolean) → position-list<V P> )
from-scheme-list
    ( pair (V V → boolean) → position-list<V P>) )
position-list?
    ( any → boolean)
length
    ( position-list<V P> → number )
full?
    ( position-list<V P> → boolean )
empty?
    ( position-list<V P> → boolean )
map
    ( position-list<V P> (V → V') (V' V' → boolean) → position-list<V' P> )
for-each
    ( position-list<V P>  (V → any) → position-list<V' P> )
first
    ( position-list<V P> → P )
last
    ( position-list<V P> → P )
```

posities zijn niet noodzakelijk numeriek

Positionele Lijsten: Het ADT

```
has-next?
    ( position-list<V P> P → boolean )
has-previous?
    ( position-list<V P> P → boolean )
next
    ( position-list<V P> P → P )
previous
    ( position-list<V P> P → P )
find
    ( position-list<V P> V → P U {#f} )
update!
    ( position-list<V P> P V → position-list<V P> )
delete!
    ( position-list<V P> P → position-list<V P> )
peek
    ( position-list<V P> P → V )
add-before!
    ( position-list<V P> V . P → position-list<V P> )
add-after!
    ( position-list<V P> V . P → position-list<V P> )
```

Positionele Lijsten: Voorbeeld

```
(define entry-tag 'entry)
(define (make-entry day month note)
  (cons (cons entry-tag day) (cons month note)))
(define day cdar)
(define month cadr)
(define note cddr)
```

```
(define todo-list-entry-1 (make-entry 5 10 "Give Lecture on Strings"))
(define todo-list-entry-2 (make-entry 12 10 "Give Lecture on Linearity"))
(define todo-list-entry-3 (make-entry 19 10 "Give Lecture Sorting"))
```

```
(define entry-eq? (lambda (entry1 entry2)
                    (and (eq? (day entry1) (day entry2))
                         (eq? (month entry1) (month entry2)))))
(define todo-list (new entry-eq?))
```

```
todo-list ∈ positional-list<entry number>
```

```
todo-list ∈ positional-list<entry pair>
```

Positionele Lijsten: Voorbeeld

```
(add-after! todo-list todo-list-entry-1)
(add-after! todo-list todo-list-entry-2)
(add-after! todo-list todo-list-entry-3)
```

```
(define lecture-2 (find todo-list (make-entry 12 10 '())))
(add-before! todo-list
             (make-entry 8 10 "Prepare Lecture on Linearity")
             lecture-2)

(define prepare-lecture (find todo-list (make-entry 8 10 '())))
(add-after! todo-list (make-entry 9 10 "Have a Rest") prepare-lecture)

(define lecture-3 (find todo-list (make-entry 19 10 '())))
(update! todo-list
         lecture-3
         (make-entry 19 10 "Give Lecture on Linear ADTs"))
```

```
(for-each
   todo-list
   (lambda (entry)
     (display (list "On " (day entry) "/"
                    (month entry) ": " (note entry)))
     (newline)))
```

Implementaties: algemene structuur

Libraries in Scheme

```
(library
 (positional-list adt)
 (export new from-scheme-list map for-each
         next previous first last has-next? has-previous?
         find delete! peek update! add-before! add-after!
         length empty? full?)
 ...)
```

```
(library
 (vector-position-list)
```

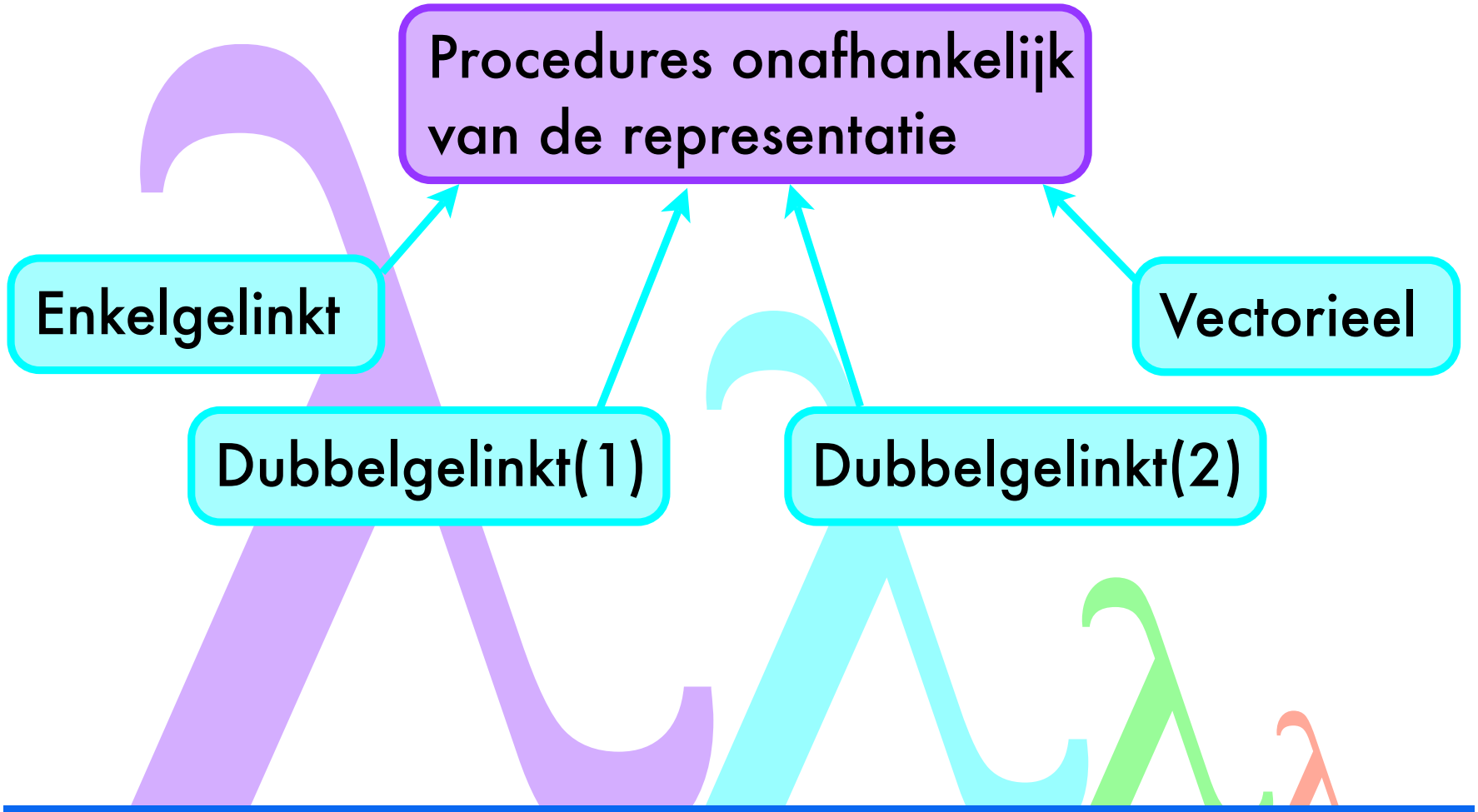
```
 (library
  (linked-position-list)
```

```
  (library
   (double-position-list)
```

```
   (library
    (augmented-double-linked-positional-list)
    (export new positional-list-tag equality
            attach-first! attach-last! attach-middle!
            detach-first! detach-last! detach-middle!
            length empty? full? update! peek
            first last has-next? has-previous? next previous)
```

Laat ons toe stukken code te hergebruiken

Implementaties: algemene structuur



Herbruikbare Procedures

```
(define (for-each plst f)
  (if (not (empty? plst))
    (let for-all
      ((curr (first plst)))
      (f (peek plst curr))
      (if (has-next? plst curr)
        (for-all (next plst curr)))))
  plst)

(define (map plst f ==?)
  (define result (new ==?))
  (if (empty? plst)
    result
    (let for-all
      ((orig (first plst))
       (curr (first
              (add-after! result (f (peek plst (first plst)))))))
      (if (has-next? plst orig)
        (for-all (next plst orig)
                 (next (add-after! result
                                   (f (peek plst (next plst orig)))
                                   curr)
                       curr))
        result))))
```

Herbruikbare Procedures

```
(define (from-scheme-list slst ==?)
  (define result (new ==?))
  (if (null? slst)
    result
    (let for-all
      ((orig (cdr slst))
       (curr (first (add-after! result (car slst)))))
      (cond
        ((not (null? orig))
         (add-after! result (car orig) curr)
         (for-all (cdr orig) (next result curr)))
        (else
         result)))))

(define (delete! plst pos)
  (cond
    ((eq? pos (first plst))
     (detach-first! plst))
    ((not (has-next? plst pos))
     (detach-last! plst pos))
    (else
     (detach-middle! plst pos)))
  plst)
```

Herbruikbare Procedures

```
(define (add-before! plst val . pos)
  (cond
    ((and (empty? plst)
          (not (null? pos)))
     (error "illegal position (add-before!)" plst))
    ((or (null? pos)
         (eq? (car pos) (first plst)))
     (attach-first! plst val))
    (else
     (attach-middle! plst val (previous plst (car pos)))))
  plst)

(define (add-after! plst val . pos)
  (cond
    ((and (empty? plst)
          (not (null? pos)))
     (error "illegal position (add-after!)" plst))
    ((null? pos)
     (attach-last! plst val))
    (else
     (attach-middle! plst val (car pos))))
  plst)
```

Herbruikbare Procedures

```
(define (positional-list? any)
  (and (pair? any)
       (eq? (car any) positional-list-tag)))

(define (find plst key)
  (define ==? (equality plst))
  (if (empty? plst)
    #f
    (let sequential-search
      ((curr (first plst)))
      (cond
        ((==? key (peek plst curr))
         curr)
        ((not (has-next? plst curr))
         #f)
        (else
         (sequential-search (next plst curr)))))))
```

Performantie

`from-scheme-list`	$O(n)$
`map`	$O(n)$
`for-each`	$O(n)$
`delete!`	$O(\max(f_{\text{detach-first!}}, f_{\text{detach-last!}}, f_{\text{detach-middle!}}))$
`find`	$O(n)$
`add-before!`	$O(\max(f_{\text{attach-first!}}, f_{\text{previous}}, f_{\text{attach-middle!}}))$
`add-after!`	$O(\max(f_{\text{attach-middle!}} f_{\text{attach-last!}} **))$

Op te vullen gaten

representatie

```
(define position-list-tag ...)
(define (new ==?) ...)
```

verificatie

```
(define (length plst) ...)
(define (full? plst) ...)
(define (empty? plst) ...)
```

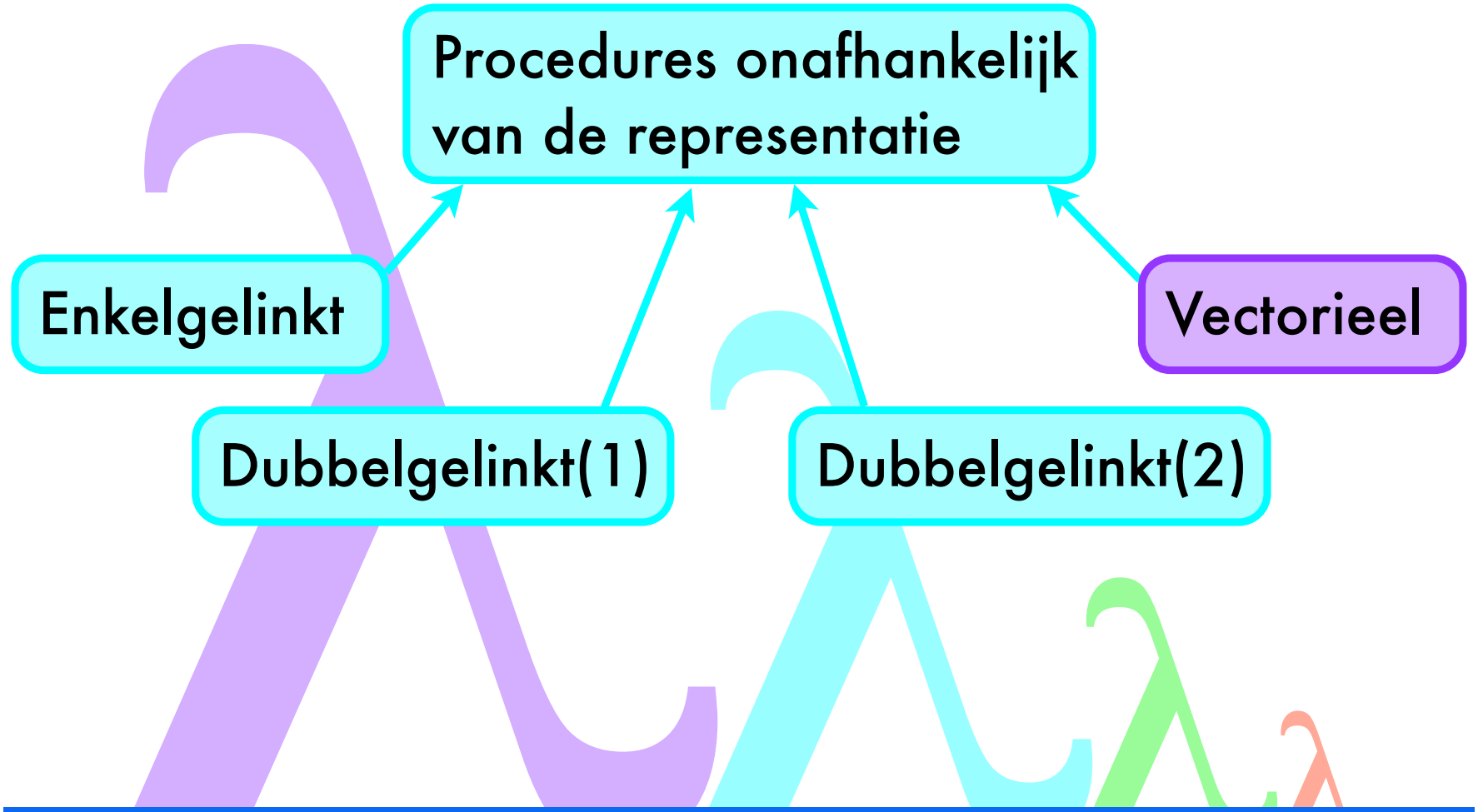
navigatie

```
(define (first plst) ...)
(define (last plst) ...)
(define (has-next? plst pos) ...)
(define (has-previous? plst pos) ...)
(define (next plst pos) ...)
(define (previous plst pos) ...)
```

manipulatie

```
(define (update! plst pos val) ...)
(define (peek plst pos) ...)
(define (attach-first! plst val) ...)
(define (attach-middle! plst val pos) ...)
(define (attach-last! plst val) ...)
(define (detach-first! plst) ...)
(define (detach-middle! plst pos) ...)
(define (detach-last! plst pos) ...)
```

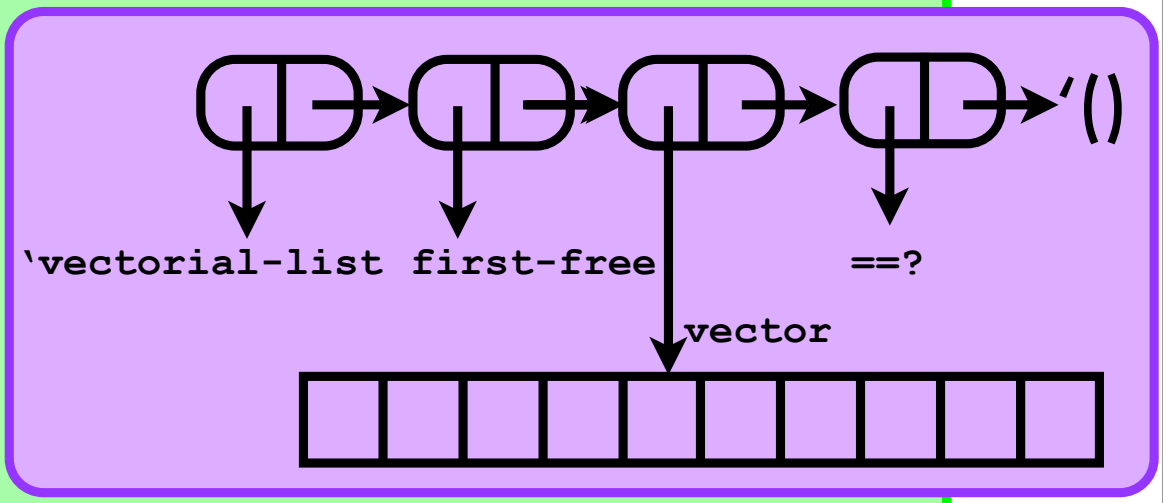
Implementaties: algemene structuur



De vectoriële implementatie

representatie

```
(define positional-list-tag 'vectorial-list)
(define positional-list-size 50)

(define (make first-free size ==?)
  (list positional-list-tag first-free (make-vector size) ==?))
(define (storage plst)
  (caddr plst))
(define (storage! plst vector)
  (set-car! (cddr plst) vector))
(define (size plst)
  (cadr plst))
(define (size! plst s)
  (set-car! (cdr plst) s))
(define (equality plst)
  (cadddr plst))



(define (new ==?)
  (make 0 positional-list-size ==?))
```



De vectoriële implementatie

```
(define (storage-move-right vector i j)
  (define (iter idx)
    (vector-set! vector (+ idx 1) (vector-ref vector idx))
    (if (> idx i)
      (iter (- idx 1))))
  (iter j))

(define (storage-move-left vector i j)
  (define (iter idx)
    (vector-set! vector (- idx 1) (vector-ref vector idx))
    (if (< idx j)
      (iter (+ idx 1))))
  (iter i))
```

De vectoriële implementatie

verificatie

```
(define (length plst)
  (size plst))

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

(define (full? plst)
  (= (+ (size plst) 1)
     (vector-length (storage plst))))
```

De vectoriële implementatie

```
(define (peek plst pos)
  (if (> pos (size plst))
    (error "illegal position (peek)" plst)
    (vector-ref (storage plst) pos)))

(define (update! plst pos val)
  (if (> pos (size plst))
    (error "illegal position (update!)" plst)
    (vector-set! (storage plst) pos val))))
```

manipulatie

```
(define (detach-first! plst)
  (detach-middle! plst 0))

(define (detach-last! plst pos)
  (define first-free (size plst))
  (size! plst (- first-free 1)))

(define (detach-middle! plst pos)
  (define list-storage (storage plst))
  (define first-free (size plst))
  (storage-move-left list-storage (+ pos 1) (- first-free 1))
  (size! plst (- first-free 1)))
```

De vectoriële implementatie

```
(define (attach-first! plst val)
  (attach-middle! plst val -1))

(define (attach-last! plst val)
  (define list-storage (storage plst))
  (define first-free (size plst))
  (vector-set! list-storage first-free val)
  (size! plst (+ first-free 1)))

(define (attach-middle! plst val pos)
  (define list-storage (storage plst))
  (define first-free (size plst))
  (storage-move-right list-storage (+ pos 1) first-free)
  (vector-set! list-storage (+ pos 1) val)
  (size! plst (+ first-free 1)))
```

De vectoriële implementatie

navigatie

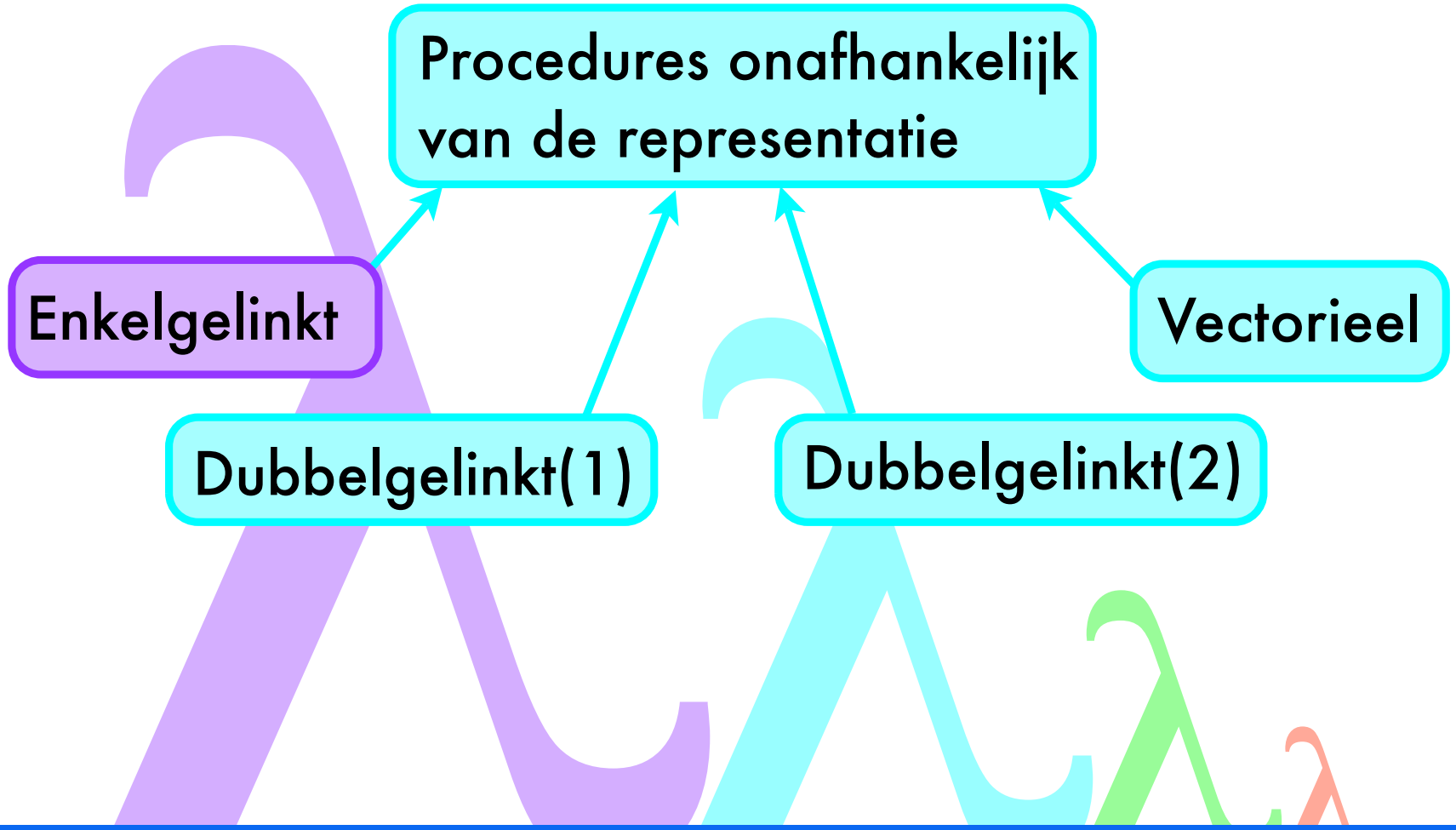
```
(define (first plst)
  (if (= 0 (size plst))
    (error "empty list (first)" plst)
    0))

(define (last plst)
  (if (= 0 (size plst))
    (error "empty list (last)" plst)
    (- (size plst) 1)))

(define (has-next? plst pos)
  (< (+ pos 1) (size plst)))

(define (has-previous? plst pos)
  (< 0 pos))

(define (next plst pos)
  (if (not (has-next? plst pos))
    (error "list has no next (next)" plst)
    (+ pos 1)))

(define (previous plst pos)
  (if (not (has-previous? plst pos))
    (error "list has no previous (previous)" plst)
    (- pos 1)))
```

Vectorieel: Performantie

`length`	`O(1)`
`first`	`O(1)`
`last`	`O(1)`
`has-next?`	`O(1)`
`has-previous?`	`O(1)`
`next`	`O(1)`
`previous`	`O(1)`
`peek`	`O(1)`
`update!`	`O(1)`
`delete!`	`O(n)`
`add-before!`	`O(n)`
`add-after!`	`O(n)`

Navigatie is zeer snel. Toevoegen en weglaten traag. Beperkte flexibiliteit.

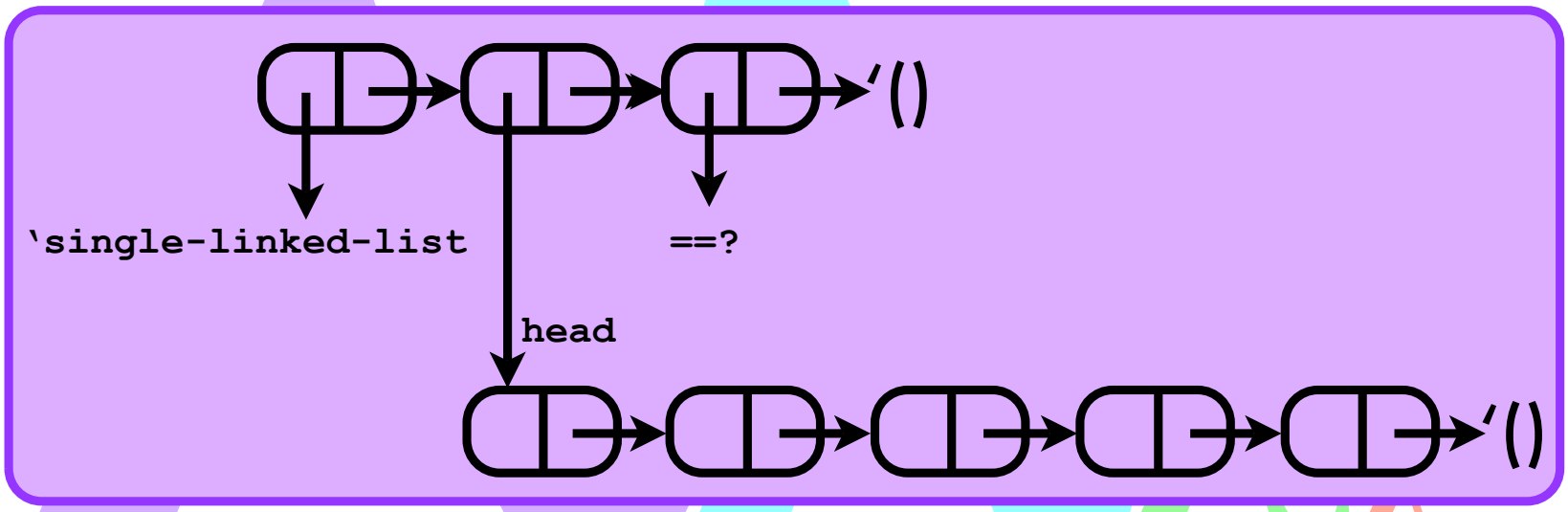
Implementaties: algemene structuur



De enkelgelinkte implementatie

```
(define positional-list-tag 'single-linked-list)
(define (make head ==?)
  (list positional-list-tag head ==?))
(define (head plst)
  (cadr plst))
(define (head! plst node)
  (set-car! (cdr plst) node))
(define (equality plst)
  (caddr plst))

(define (new ==?)
  (make '() ==?))
```

representatie



De enkelgelinkte implementatie

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

Een privaat “list-node” ADTtje

De enkelgelinkte implementatie

```
(define (length plst)
  (let length-iter
    ((curr (head plst))
     (size 0))
    (if (null? curr)
      size
      (length-iter (list-node-next curr) (+ size 1)))))

(define (full? plst)
  #f)

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

verificatie

```
(define (update! plst pos val)
  (list-node-val! pos val)
  plst)

(define (peek plst pos)
  (list-node-val pos))
```

manipulatie

De enkelgelinkte implementatie

```
(define (attach-first! plst val)
  (define first (head plst))
  (define node (make-list-node val first))
  (head! plst node))

(define (attach-last! plst val)
  (define first (head plst))
  (define node (make-list-node val '()))
  (if (null? first)
    (head! plst node) ; last is also first
    (list-node-next!
     (let iter-to-last
       ((curr first))
       (if (null? (list-node-next curr))
         curr
         (iter-to-last (list-node-next curr))))
     node)))

(define (attach-middle! plst val pos)
  (define next (list-node-next pos))
  (define node (make-list-node val next))
  (list-node-next! pos node))
```

De enkelgelinkte implementatie

```
(define (detach-middle! plst pos)
   (define next (list-node-next pos))
   (define prev (let iter-to-previous
                  ((curr (head plst)))
                  (if (eq? pos (list-node-next curr))
                    curr
                    (iter-to-previous (list-node-next curr)))))
   (list-node-next! prev next))

 (define (detach-last! plst pos)
   (define first (head plst))
   (if (null? (list-node-next first)) ; last is also first
     (head! plst '())
     (list-node-next!
      (let iter-to-penultimate
        ((penu first)
         (last (next plst first)))
        (if (not (has-next? plst last))
          penu
          (iter-to-penultimate last (next plst last))))
      '())))
```

De enkelgelinkte implementatie

```
(define (detach-first! plst)
  (define first (head plst))
  (define second (list-node-next first))
  (head! plst second))
```

navigatie

```
(define (previous plst pos)
  (if (not (has-previous? plst pos))
    (error "list has no previous (previous)" plst))
  (let iter-to-previous
    ((curr (head plst)))
    (if (eq? (list-node-next curr) pos)
      curr
      (iter-to-previous (list-node-next curr)))))
```

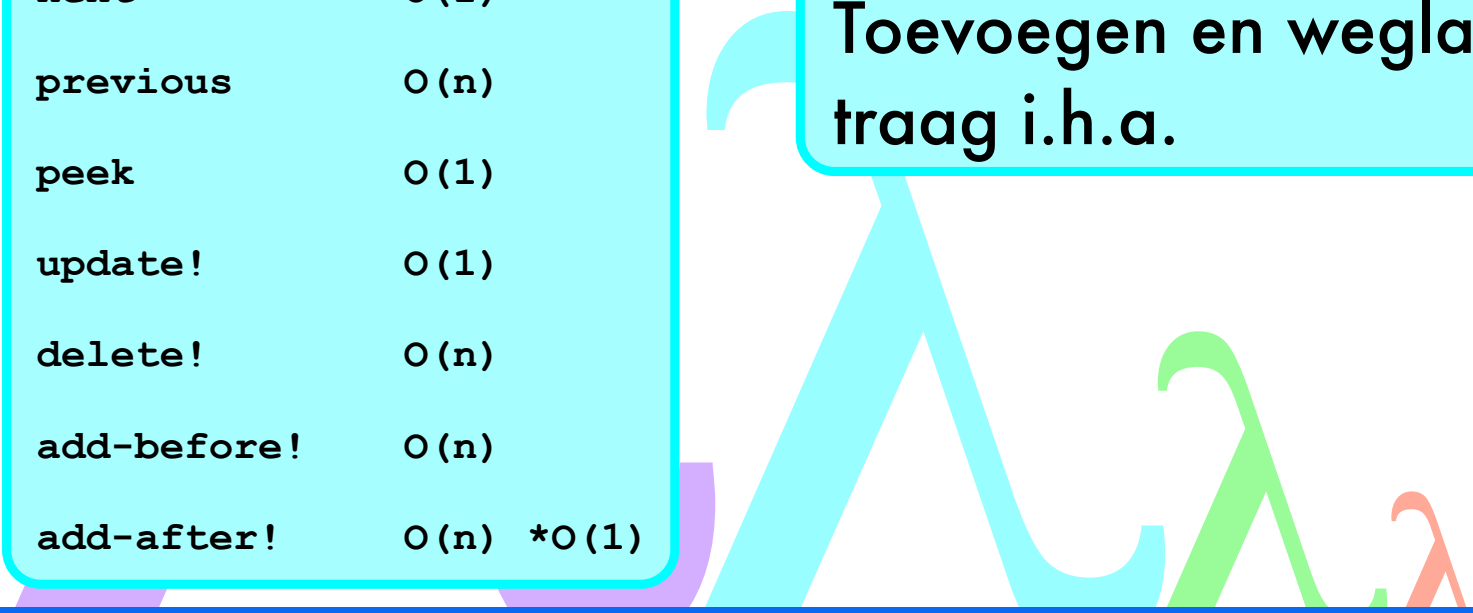
De enkelgelinkte implementatie

```
(define (first plst)
  (if (null? (head plst))
    (error "list empty (first)" plst)
    (head plst)))

(define (last plst)
  (if (null? (head plst))
    (error "list empty (last)" plst))
  (let iter-to-last
    ((curr (head plst)))
    (if (null? (list-node-next curr))
      curr
      (iter-to-last (list-node-next curr)))))

(define (has-next? plst pos)
  (not (null? (list-node-next pos))))

(define (has-previous? plst pos)
  (not (eq? pos (head plst))))

(define (next plst pos)
  (if (not (has-next? plst pos))
    (error "list has no next (next)" plst)
    (list-node-next pos)))
```

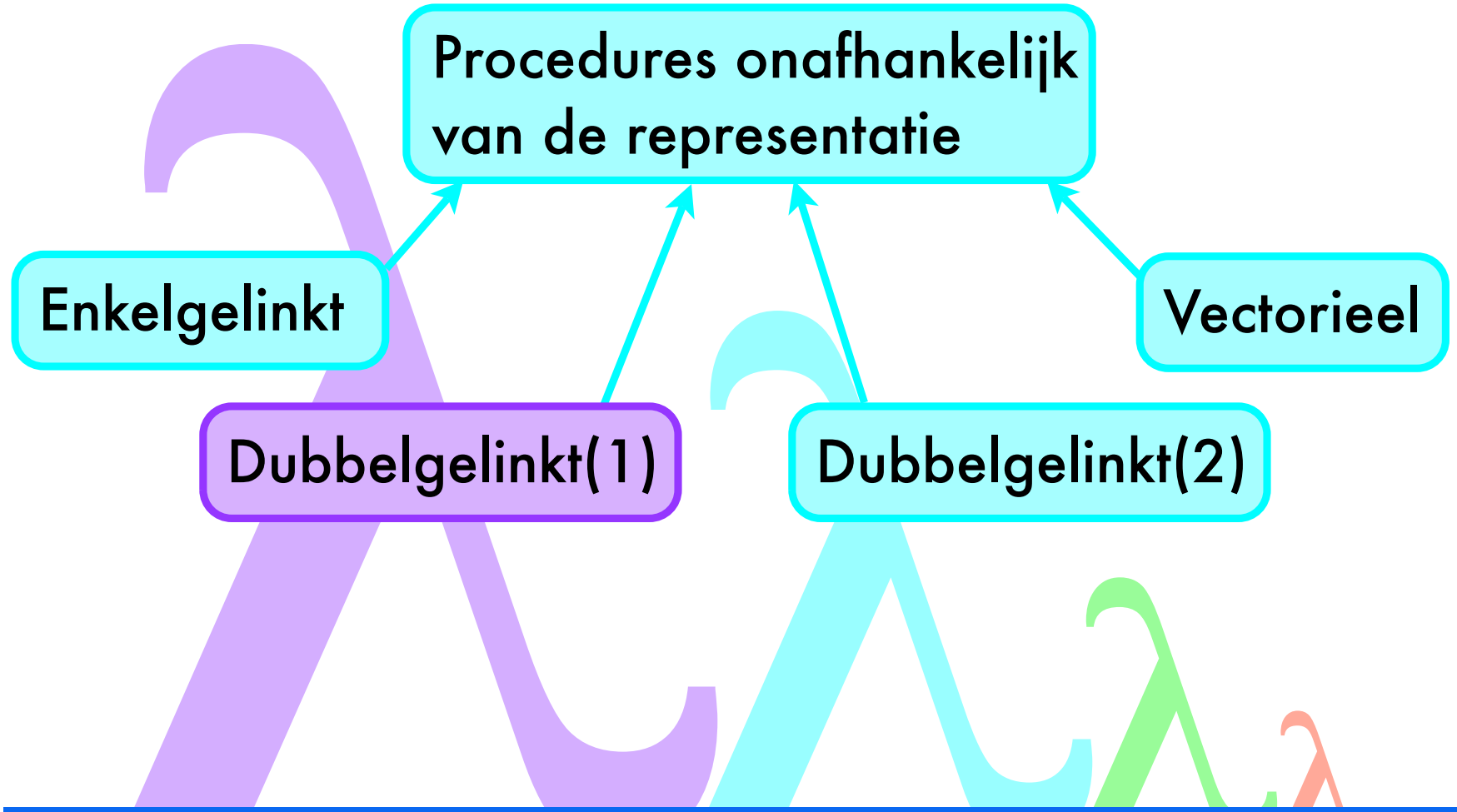
Enkelgelinkt: Performantie

`length`	`O(n)`	
`first`	`O(1)`	
`last`	`O(n)`	
`has-next?`	`O(1)`	
`has-previous?`	`O(1)`	
`next`	`O(1)`	
`previous`	`O(n)`	
`peek`	`O(1)`	
`update!`	`O(1)`	
`delete!`	`O(n)`	
`add-before!`	`O(n)`	
`add-after!`	`O(n)`	`*O(1)`

Achterwaartse navigatie traag. Grote Flexibiliteit. Toevoegen en weglaten traag i.h.a.



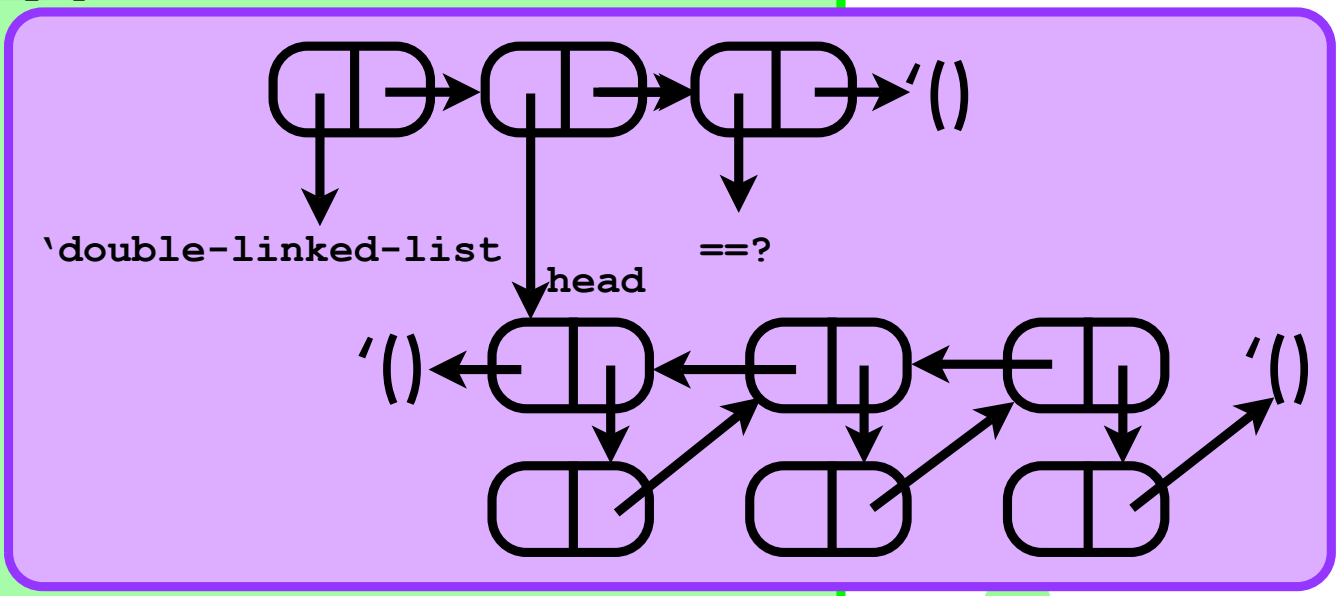
Implementaties: algemene structuur



De dubbelgelinkte implementatie

```
(define positional-list-tag 'double-linked-list)

(define (make head ==?)
  (list positional-list-tag head ==?))
(define (head plst)
  (cadr plst))
(define (head! plst node)
  (set-car! (cdr plst) node))
(define (equality plst)
  (caddr plst))
```

representatie




```
(define (new ==?)
  (make '() ==?))
```

De dubbelgelinkte implementatie

```
(define (make-list-node val prev next)
  (cons prev (cons val next)))
(define (list-node-val node)
  (cadr node))
(define (list-node-val! node val)
  (set-car! (cdr node) val))
(define (list-node-next node)
  (cddr node))
(define (list-node-next! node next)
  (set-cdr! (cdr node) next))
(define (list-node-prev node)
  (car node))
(define (list-node-prev! node prev)
  (set-car! node prev))
```

Een privaat "list-node" ADTtje

De dubbelgelinkte implementatie

verificatie

```
...
```

navigatie

```
(define (next plst pos)
  ...)

(define (previous plst pos)
  (if (not (has-previous? plst pos))
    (error "list has no previous (previous)" plst)
    (list-node-prev pos)))

(define (first plst)
  ...)

(define (last plst)
  ...)

(define (has-next? plst pos)
  ...)

(define (has-previous? plst pos)
  ...)
```

De dubbelgelinkte implementatie

```
(define (update! plst pos val)
  ... )

(define (peek plst pos)
  ... )

(define (attach-first! plst val)
  (define head (first plst))
  (define node (make-list-node val '() head))
  (if (not (null? head))
    (list-node-prev! head node))
  (head! plst node))

(define (attach-middle! plst val pos)
  (define next (list-node-next pos))
  (define node (make-list-node val pos next))
  (list-node-next! pos node)
  (if (not (null? next))
    (list-node-prev! next node)))
```

manipulatie

De dubbelgelinkte implementatie

```
(define (attach-last! plst val)
  (define first (head plst))
  (define last
    (if (null? first)
      first
      (let iter-to-last
        ((curr first))
        (if (not (has-next? plst curr))
          curr
          (iter-to-last (next plst curr))))))
  (define node (make-list-node val last '()))
  (if (null? first) ; last will be first as well
    (head! plst node)
    (list-node-next! last node)))
```

De dubbelgelinkte implementatie

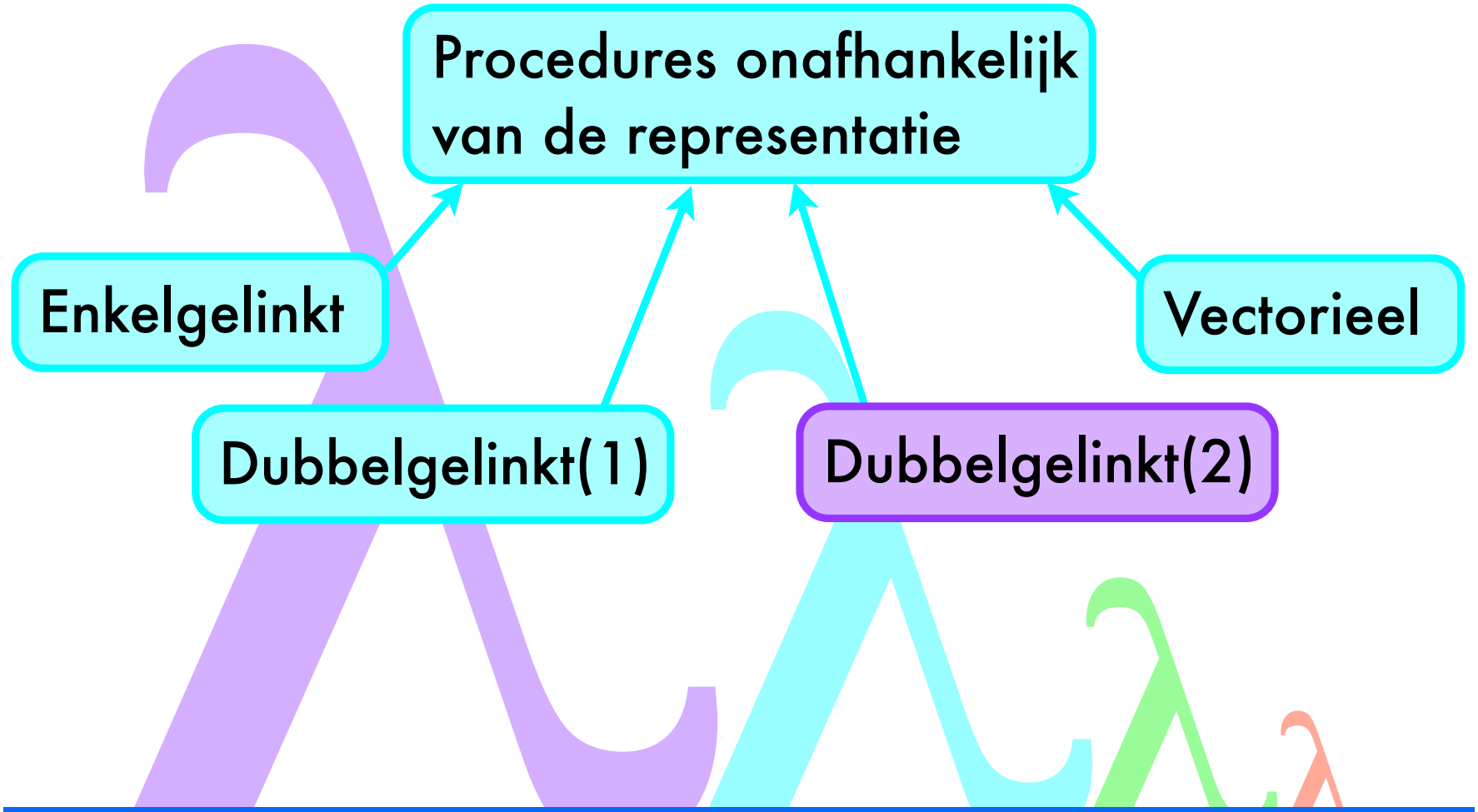
```
(define (detach-first! plst)
  (define first (head plst))
  (define next (list-node-next first))
  (head! plst next)
  (if (not (null? next))
    (list-node-prev! next '())))

(define (detach-last! plst pos)
  (define first (head plst))
  (if (null? (list-node-next first)) ; last is the first
    (head! plst '())
    (list-node-next! (list-node-prev pos) '())))

(define (detach-middle! plst pos)
  (define next (list-node-next pos))
  (define prev (list-node-prev pos))
  (list-node-prev! next prev)
  (list-node-next! prev next))
```

Dubbelgelinkt: Performantie

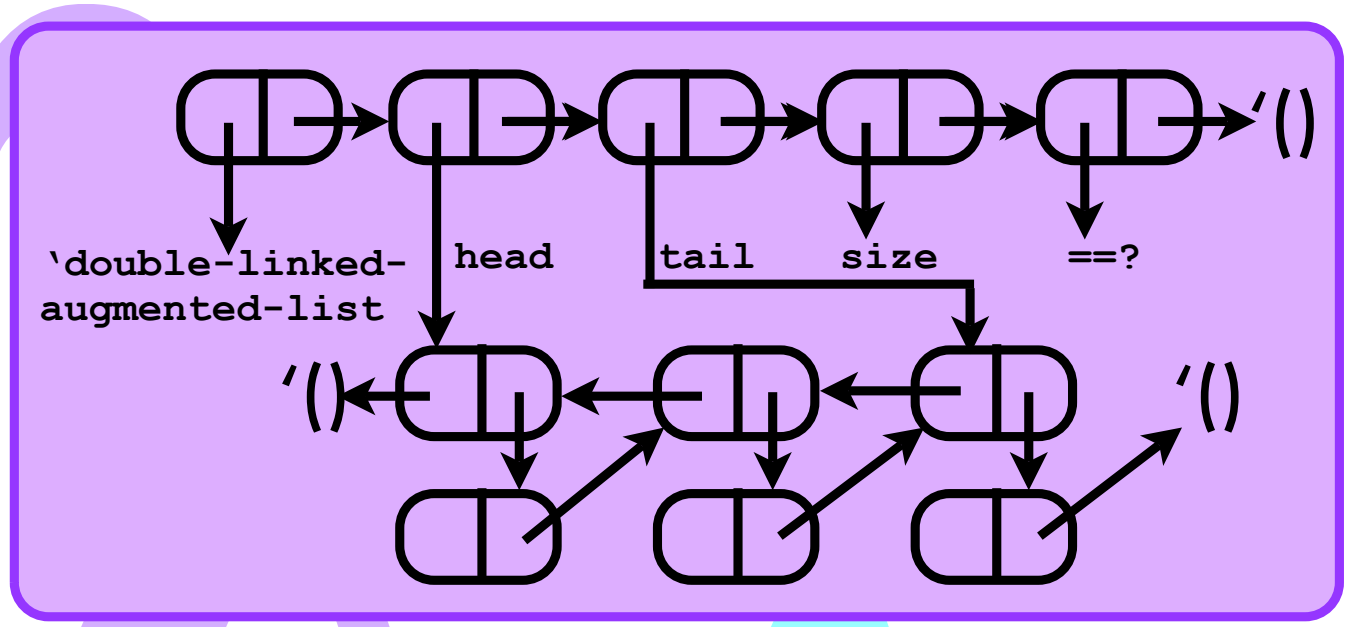
`length`	`O(n)`
`first`	`O(1)`
`last`	`O(n)`
`has-next?`	`O(1)`
`has-previous?`	`O(1)`
`next`	`O(1)`
`previous`	`O(1)`
`peek`	`O(1)`
`update!`	`O(1)`
`delete!`	`O(1)`
`add-before!`	`O(1)`
`add-after!`	`O(n)` `*O(1)`

Navigatie zeer snel.
Grote Flexibiliteit.
Bijna alles O(1)

Implementaties: algemene structuur



Dubbelgelinkt + Grote Header



Dubbelgelinkt + Grote Header

```
(define positional-list-tag 'double-linked-augmented-list)
(define (make head tail size ==?)
  (list positional-list-tag head tail size ==?))
(define (head plst)
  (cadr plst))
(define (head! plst node)
  (set-car! (cdr plst) node))
(define (tail plst)
  (caddr plst))
(define (tail! plst node)
  (set-car! (cddr plst) node))
(define (size plst)
  (cadddr plst))
(define (size! plst s)
  (set-car! (cdddr plst) s))
(define (equality plst)
  (caddr (cddr plst)))

(define (new ==?)
  (make '() '() 0 ==?))
```

representatie

Het privaat "list-node" ADTtje blijft onveranderd

```
(define (length plst)
  (size plst))
...
```

verificatie

Dubbelgelinkt + Grote Header

```
(define (last plst)
  (if (null? (tail plst))
    (error "list empty (last)" plst)
    (tail plst)))
```

navigatie

```
(define (attach-first! plst val)
  (define first (head plst))
  (define node (make-list-node val '() first))
  (head! plst node)
  (if (null? (tail plst)) ; last is first
    (tail! plst node)
    (list-node-prev! first node))
  (size! plst (+ 1 (size plst))))

(define (attach-last! plst val)
  (define last (tail plst))
  (define node (make-list-node val last '()))
  (tail! plst node)
  (if (null? (head plst)) ; first is last
    (head! plst node)
    (list-node-next! last node))
  (size! plst (+ 1 (size plst))))
```

manipulatie

Dubbelgelinkt + Grote Header

```
(define (attach-middle! plst val pos)
  (define next (list-node-next pos))
  (define node (make-list-node val pos next))
  (list-node-next! pos node)
  (if (eq? pos (tail plst))
    (tail! plst node)
    (list-node-prev! next node))
  (size! plst (+ 1 (size plst))))

(define (detach-first! plst)
  (define first (head plst))
  (define second (list-node-next first))
  (if (null? second)                    ; first is the only one
    (tail! plst second))
  (head! plst second)
  (size! plst (- (size plst) 1)))
```

Dubbelgelinkt + Grote Header

```
(define (detach-last! plst pos)
  (define first (head plst))
  (define second (list-node-next first))
  (define last (tail plst))
  (define penultimate (list-node-prev last))
  (if (null? second)                    ; last is the first?
    (head! plst second)
    (list-node-next! penultimate '()))
  (tail! plst penultimate)
  (size! plst (- (size plst) 1)))

(define (detach-middle! plst pos)
  (define next (list-node-next pos))
  (define prev (list-node-prev pos))
  (list-node-next! prev next)
  (list-node-prev! next prev)
  (size! plst (- (size plst) 1)))
```

Dubbelgelinkt + GH: Performantie

`length`	O(1)
`first`	O(1)
`last`	O(1)
`has-next?`	O(1)
`has-previous?`	O(1)
`next`	O(1)
`previous`	O(1)
`peek`	O(1)
`update!`	O(1)
`delete!`	O(1)
`add-before!`	O(1)
`add-after!`	O(1)

Grote Flexibiliteit.
Alles in O(1).

Conclusie position-list ADT

`length`	`O(n)`	`O(n)`	`O(n)`	`O(1)`
`first`	`O(1)`	`O(1)`	`O(1)`	`O(1)`
`last`	`O(1)`	`O(n)`	`O(1)`	`O(1)`
`has-next?`	`O(1)`	`O(1)`	`O(1)`	`O(1)`
`has-previous?`	`O(1)`	`O(1)`	`O(1)`	`O(1)`
`next`	`O(1)`	`O(1)`	`O(1)`	`O(1)`
`previous`	`O(1)`	`O(n)`	`O(1)`	`O(1)`
`peek`	`O(1)`	`O(1)`	`O(1)`	`O(1)`
`update!`	`O(1)`	`O(1)`	`O(1)`	`O(1)`
`delete!`	`O(n)`	`O(n)`	`O(1)`	`O(1)`
`add-before!`	`O(n)`	`O(n)`	`O(1)`	`O(1)`
`add-after!`	`O(n)`	`O(n)` `*O(1)`	`O(n)` `*O(1)`	`O(1)`

`find` blijft in O(n)

De "performantie trade-off"

Een lijst van n elementen =
vectorieel : n geheugencellen
enkelgelinkt: 2n geheugencellen
dubbelgelinkt: 3n of 4n geheugencellen

Lijstimplementaties leveren een mooie illustratie van het principe:

Geheugen $\leftrightharpoons$ Uitvoeringstijd

Positionele Lijsten : Probleem

```
(define lectures (list (find todo-list (make-entry 5 10 '()))
                       (find todo-list (make-entry 12 10 '()))
                       (find todo-list (make-entry 19 10 '()))))
```

```
(define rest (find todo-list (make-entry 9 10 '())))
(add-after! todo-list (make-entry 11 10 "Go out with friends") rest)

(map (lambda (pos)
       (display (note (peek todo-list pos)))
       (newline))
     lectures)
```

```
Give Lecture on Strings
Go out with friends
Give Lecture on Linearity
```

Conceptueel: relatief

Technisch: absoluut

3 ≠ Oplossingen

Alle operaties "vertalen" hun positie parameters naar het heden. Onhaalbaar.

Posities worden nooit uit de lijst vrijgegeven: list-with-current

Posities hebben geen betekenis meer eens uit de lijst vrijgegeven: ranked-list

Lijsten met een "current"

```
ADT list-with-current<V>

new
    ( (V V → boolean) → list-with-current<V> )
from-scheme-list
    ( pair (V V → boolean) → list-with-current<V>) )
list-with-current?
    ( any → boolean)
length
    ( list-with-current<V> → number )
full?
    ( list-with-current<V> → boolean )
empty?
    ( list-with-current<V> → boolean )
set-current-to-first!
    ( list-with-current<V> → list-with-current<V> )
set-current-to-last!
    ( list-with-current<V> → list-with-current<V> )
current-has-next?
    ( list-with-current<V> → boolean )
```

Lijsten met een "current"

```
current-has-previous?
    ( list-with-current<V> → boolean)
set-current-to-next!
    ( list-with-current<V> → list-with-current<V> )
set-current-to-previous!
    ( list-with-current<V> → list-with-current<V> )
has-current?
    ( list-with-current<V> → boolean )
find!
    ( list-with-current<V> V → list-with-current<V> )
update!
    ( list-with-current<V> V → list-with-current<V> )
peek
    ( list-with-current<V> → V )
delete!
    ( list-with-current<V> → list-with-current<V> )
add-before!
    ( list-with-current<V> V → list-with-current<V> )
add-after!
    ( list-with-current<V> V → list-with-current<V> )
```

Gerankte lijsten

```
ADT ranked-list<V>

new
    ( (V V -> boolean) → ranked-list<V> )
from-scheme-list
    ( any (V V → boolean) → ranked-list<V>) )
ranked-list?
    ( any → boolean)
length
    ( ranked-list<V> → number )
full?
    ( ranked-list<V> → boolean )
empty?
    ( ranked-list<V> → boolean )
find
    ( ranked-list<V> V → number U {#f} )
peek-at-rank
    ( ranked-list<V> number → V )
update-at-rank!
    ( ranked-list<V> number V → ranked-list<V> )
delete-at-rank!
    ( ranked-list<V> number → ranked-list<V> )
add-at-rank!
    ( ranked-list<V> V . number → ranked-list<V> )
```

Zoeken in lineaire structuren

Het algoritme voor `find` in de implementaties van positionele lijsten is in O(n).

Sequentieel Zoeken

We bestuderen 3 verbeteringen:

- Sentinel Search
- Zoeken in gesorteerde lijsten
- Binair Zoeken

Sentinel Search

`attach-last!` in O(1) !

```
(define (find plst key)
  (if (empty? plst)
    #f
    (let
        ((==? (equality plst)))
      (attach-last! plst key)
      (let*
          ((pos (let search-sentinel
                  ((curr (first plst)))
                  (if (==? (peek plst curr) key)
                    curr
                    (search-sentinel (next plst curr)))))
           (res (if (has-next? plst pos)
                  pos
                  #f)))
        (detach-last! plst (last plst))
        res)))))
```

Zoeken in lineaire structuren

Het algoritme voor `find` in de implementaties van positionele lijsten is in O(n).

Sequentieel Zoeken

We bestuderen 3 verbeteringen:

- Sentinel Search
- Zoeken in gesorteerde lijsten
- Binair Zoeken

Gesorteerde Lijsten

```
new
    ( (V V → boolean)
      (V V → boolean) → sorted-list<V> )
from-scheme-list
    ( pair
      (V V → boolean)
      (V V → boolean) → sorted-list<V>) )
sorted-list?
    ( any → boolean)
length
    ( sorted-list<V> → number )
empty?
    ( sorted-list<V> → boolean )
full?
    ( sorted-list<V> → boolean )
find!
    ( sorted-list<V> V → sorted-list<V> )
delete!
    ( sorted-list<V> → sorted-list<V> )
```

De gebruiker krijgt minder controle!

```
peek
    ( sorted-list<V> → V )
add!
    ( sorted-list<V> V → sorted-
list<V> )
set-current-to-first!
    ( sorted-list<V> → sorted-list<V> )
set-current-to-next!
    ( sorted-list<V> → sorted-list<V> )
has-current?
    ( sorted-list<V> → boolean )
current-has-next?
    ( sorted-list<V> → boolean )
```

Gesorteerde Lijsten

representatie

```
(define sorted-list-tag 'vector-sorted-list)
(define default-size 20)
(define (make len <<? ==?)
  (list sorted-list-tag
        len -1
        (make-vector (max default-size len)) ==? <<?))
(define (size slst)
  (cadr slst))
(define (size! slst s)
  (set-car! (cdr slst) s))
(define (current slst)
  (caddr slst))
(define (current! slst idx)
  (set-car! (cddr slst) idx))
(define (storage slst)
  (cadddr slst))
(define (storage! slst vector)
  (set-car! (cdddr slst) vector))
(define (equality slst)
  (caddr (cddr slst)))
(define (lesser slst)
  (caddr (cdddr slst)))
```

```
(define (new <<? ==?)
  (make 0 <<? ==?))

(define (from-scheme-list slst <<? ==?)
  (let loop
    ((lst slst)
     (idx 0))
    (if (null? lst)
      (make idx <<? ==?)
      (add! (loop (cdr lst) (+ idx 0))
            (car lst)))))
```

Gesorteerde Lijsten

navigatie

```
(define (set-current-to-first! slst)
  (current! slst 0))

(define (set-current-to-next! slst)
  (if (not (has-current? slst))
    (error "current has no meaningful value (set-current-to-next!" slst)
    (current! slst (+ 1 (current slst)))))

(define (has-current? slst)
  (not (= -1 (current slst))))

(define (current-has-next? slst)
  (if (not (has-current? slst))
    (error "no Current (current-has-next?)" slst)
    (< (+ (current slst) 1) (length slst))))
```

manipulatie

```
(define (peek slst)
  (if (not (has-current? slst))
    (error "no current (peek)" slst)
    (vector-ref (storage slst) (current slst))))
```

Gesorteerde Lijsten

```
(define (add! slst val)
   (define ==? (equality slst))
   (define <<? (lesser slst))
   (define vector (storage slst))
   (define length (size slst))
   (define (vector-iter idx)
     (cond
       ((= idx 0)
        (vector-set! vector idx val))
       ((<<? val (vector-ref vector (- idx 1)))
        (vector-set! vector idx (vector-ref vector (- idx 1)))
        (vector-iter (- idx 1)))
       (else
        (vector-set! vector idx val))))
   (if (= length (vector-length vector))
     (error "list full (add!)" slst))
   (vector-iter length)
   (size! slst (+ length 1))
   slst)
```

```
(define (delete! slst)
  (define vect (storage slst))
  (define last (size slst))
  (define curr (current slst))
  (if (not (has-current? slst))
    (error "no current (delete!)" slst))
  (if (< (+ curr 1) last)
    (storage-move-left vect (+ curr 1) last))
  (size! slst (- last 1))
  (current! slst -1)
  slst)
```

Gesorteerde Lijsten

```
(define (find! slst key)
  (define ==? (equality slst))
  (define <<? (lesser slst))
  (define vector (storage slst))
  (define length (size slst))
  (define (vector-iter idx)
    (cond
      ((>= idx length)
       (current! slst -1))
      ((==? key (vector-ref vector idx))
       (current! slst idx))
      ((<<? (vector-ref vector idx) key)
       (vector-iter (+ idx 1)))
      (else
       (current! slst -1))))
  (vector-iter 0)
  slst)
```

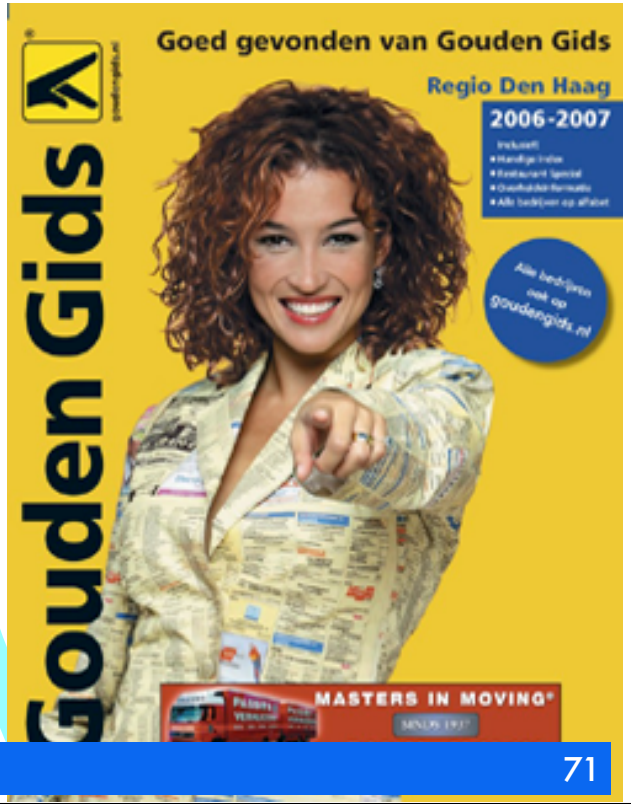
zoeken

Zoeken in lineaire structuren

Het algoritme voor `find` in de implementaties van positionele lijsten is in O(n).

Sequentieel Zoeken

We bestuderen 3 verbeteringen:
- Sentinel Search
- Zoeken in gesorteerde lijsten
- Binair Zoeken



Binair Zoeken

```
(define (find! slst key)
  (define ==? (equality slst))
  (define <<? (lesser slst))
  (define vector (storage slst))
  (define length (size slst))
  (let binary-search
    ((left 0)
     (right (- length 1)))
    (if (<= left right)
      (let ((mid (div (+ left right 1) 2)))
        (cond
          ((==? (vector-ref vector mid) key)
           (current! slst mid))
          ((<<? (vector-ref vector mid) key)
           (binary-search (+ mid 1) right))
          (else
           (binary-search left (- mid 1)))))
      (current! slst -1)))
  slst)
```

O(log(n))

Vereist O(1) indexering!

Ringen

```
ADT ring<V>

new
    ( ∅ → ring<V> )
from-scheme-list
    (pair → ring<V>) )
add-after!
    ( ring<V> V → ring<V> )
add-before
    ( ring<V> V → ring<V> )
shift-forward!
    ( ring<V> → ring<V> )
shift-backward!
    ( ring<V> → ring<V> )
delete!
    ( ring<V> → ring<V> )
update!
    ( ring<V> V → ring<V> )
peek
    ( ring<V> → V )
length
    ( ring<V> → number )
```

Een “haasje-over” task scheduler zit in quasi ieder besturingssysteem

In sommige programma’s bestaat de menu-optie “cycle through windows”

Ringen

representatie

```
(define ring-tag 'ring)
(define (make current)
  (list ring-tag current))
(define (current ring)
  (cadr ring))
(define (current! ring node)
  (set-car! (cdr ring) node))

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

(define (from-scheme-list slst)
  (let loop
    ((scml slst)
     (ring (new)))
    (if (null? scml)
      ring
      (loop (cdr scml) (add-after! ring (car scml))))))
```

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

Ringen

navigatie

```
(define (shift-forward! ring)
  (define curr (current ring))
  (if (null? curr)
    (error "empty ring (shift-forward!)" ring))
  (current! ring (ring-node-next curr))
  ring)

(define (shift-backward! ring)
  (define curr (current ring))
  (if (null? curr)
    (error "empty ring (shift-backward!)" ring)
    (current! ring (let iter-to-previous
                       ((prev curr)
                        (next (ring-node-next curr)))
                       (if (eq? curr next)
                         prev
                         (iter-to-previous next (ring-node-next next))))))
  ring)
```

Kan sneller dubbelgelinkt

Ringen

```
(define (length ring)
  (define curr (current ring))
  (if (null? curr)
    0
    (let loop
      ((pointer (ring-node-next curr))
       (acc 1))
      (if (eq? pointer curr)
        acc
        (loop (ring-node-next pointer) (+ acc 1))))))

(define (update! ring val)
  (define curr (current ring))
  (if (null? curr)
    (error "empty ring (update!)"ring)
    (ring-node-val! curr val)))

(define (peek ring)
  (define curr (current ring))
  (if (null? curr)
    (error "empty ring (peek)" ring)
    (ring-node-val curr)))
```

verificatie

manipulatie

Ringen

```
(define (add-before! ring val)
  (define curr (current ring))
  (define node (make-ring-node val curr))
  (ring-node-next!
   (if (null? curr)
     node
     (let iter-to-previous
       ((prev curr)
        (next (ring-node-next curr)))
       (if (eq? curr next)
         prev
         (iter-to-previous next (ring-node-next next)))))
   node)
  (current! ring node)
  ring)
```

Kan sneller dubbelgelinkt

```
(define (add-after! ring val)
  (define curr (current ring))
  (define node (make-ring-node val '()))
  (ring-node-next! node
                   (if (null? curr)
                     node
                     (ring-node-next curr)))
  (if (not (null? curr))
    (ring-node-next! curr node))
  (current! ring node)
  ring)
```

Ringen

Kan sneller dubbelgelinkt

```
(define (delete! ring)
  (define curr (current ring))
  (if (null? curr)
    (error "empty ring (delete!)" ring))
  (ring-node-next!
   (let iter-to-previous
     ((prev curr)
      (next (ring-node-next curr)))
     (if (eq? curr next)
       prev
       (iter-to-previous next (ring-node-next next))))
   (ring-node-next curr))
  (if (eq? curr (ring-node-next curr))
    (current! ring '())
    (current! ring (ring-node-next curr)))
  ring)
```

Hoofdstuk 3