Hoofdstuk 3

Lineaire Datastructuren

Inhoud

1. Lineaire Datastructuren
2. Positionele Lijsten
3. Variaties op Positionele Lijsten
4. Zoeken in Lineaire Datastructuren

3.1 Lineaire Datastructuren

Lineariteit

Een rij van boeken op een rek

Een wachtrij in de supermarkt

Lineariteit is een zeer natuurlijk begrip

Een stapel dossiers die te verwerken zijn

De menu-opties onder “edit”

Je ‘friends’ lijst op FB

Ook in computertoepassingen

Je playlist in Spotify

Een lijst van kandidaten op een kieslijst

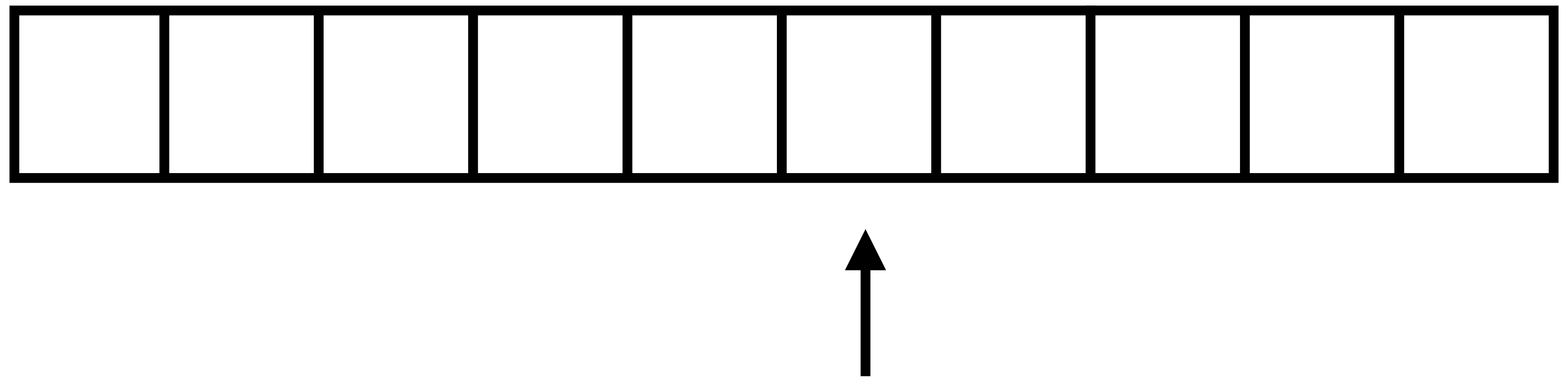
Er zijn heel wat nadelen verbonden aan het gebruik van “naakte” lijsten en “naakte” vectoren

Wat zijn de performantiekarakteristieken van de operaties?

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.

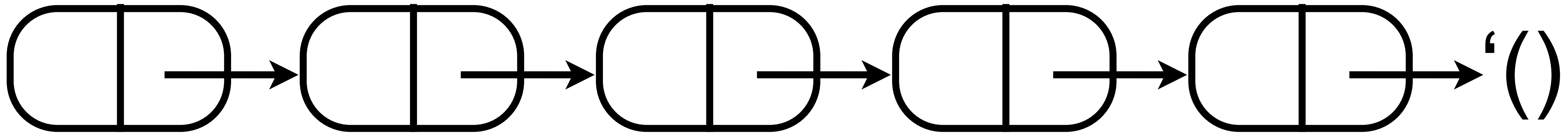
Problemen met “naakte” vectoren en lijsten

Grootte moet a-priori gekend zijn ⇒ mogelijke verspilling of tekort



extra variabele nodig per vector om “tot hier zit nuttige info” aan te geven: vervuiling van de omgeving

“aanconsen” kan enkel vooraan. Op andere plaatsen toevoegen is traag.



Scheme lijsten zijn niet generisch. `member` `memv` en `memq` gebruiken een vaste “gelijkheid”. Hoe zoek je de persoon met een zeker salaris in een Scheme lijst?

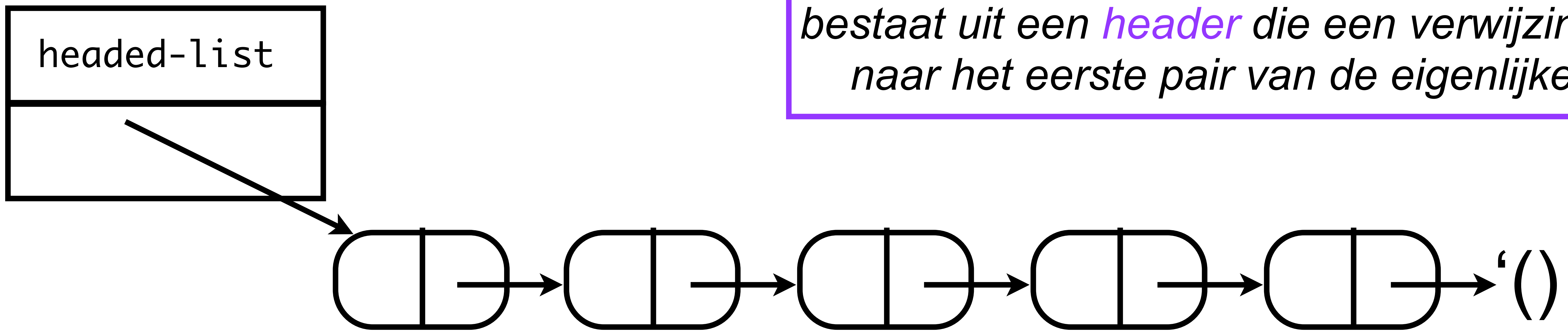
call-by-value steekt stokken in de wielen:

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

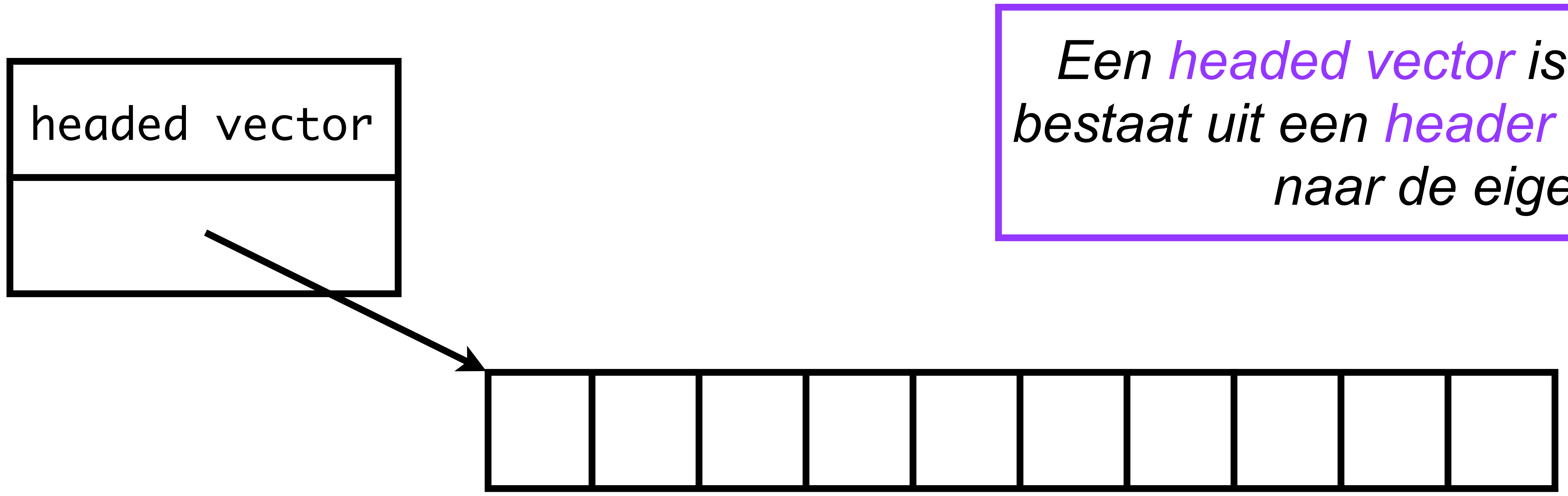
Oplossing: Headed Lists en Headed Vectoren

De header kan een gelijk wat zijn: pair, vector, record

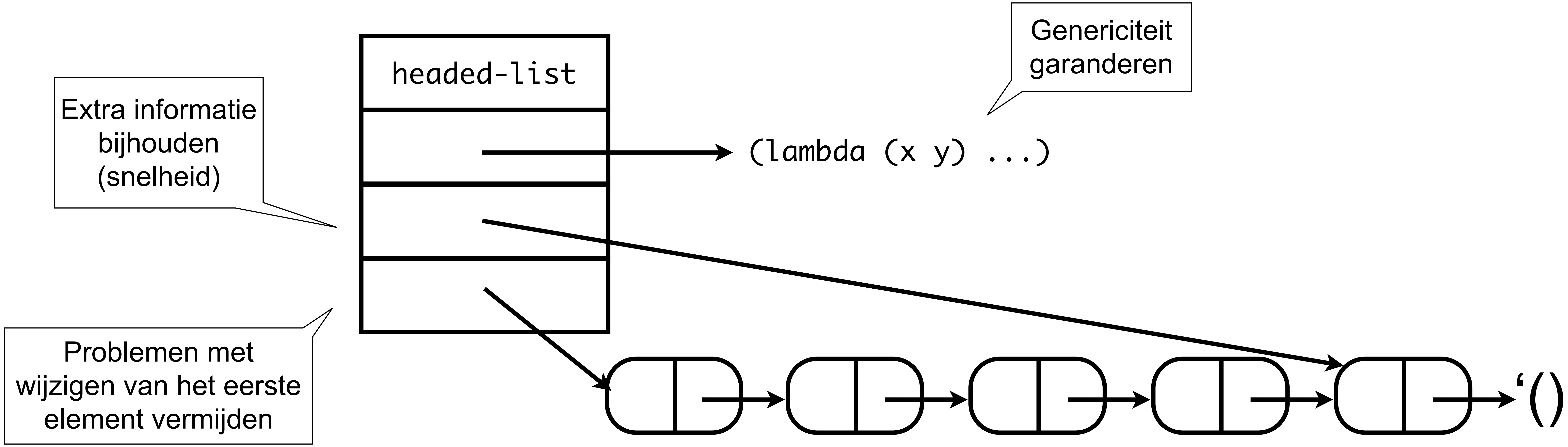
Een headed list is een datastructuur die bestaat uit een header die een verwijzing heeft naar het eerste pair van de eigenlijke lijst.



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



Wat kan de header zoal doen?



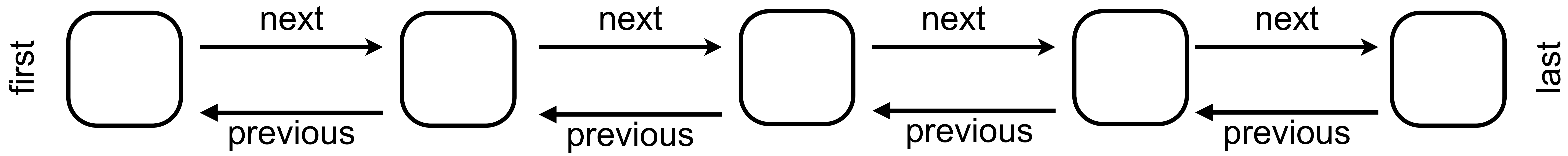
Genericiteit garanderen

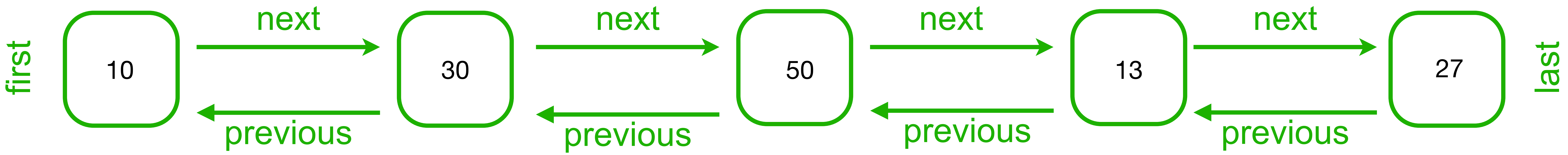
Extra informatie bijhouden (snelheid)

Problemen met wijzigen van het eerste element vermijden

Lineaire Datastructuur: Abstracte Definitie

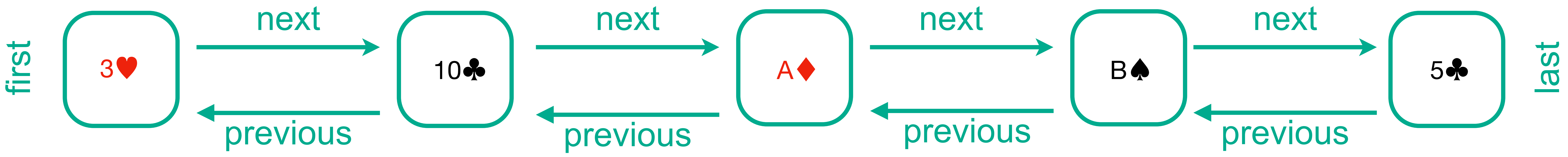
Een lineaire datastructuur is een verzameling data elementen waarbij elk data element geassocieerd is met een positie. Elke positie heeft een unieke volgende positie en een unieke vorige positie. Er zijn twee speciale posities: de laatste positie heeft geen volgende positie en de eerste positie heeft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





Voorbeeld: De data elementen zij speelkaarten

Voorbeeld: De data elementen zij getallen



3.2 Positionele Lijsten

Positionele Lijsten: Het ADT

```
ADT positional-list<V P>

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

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

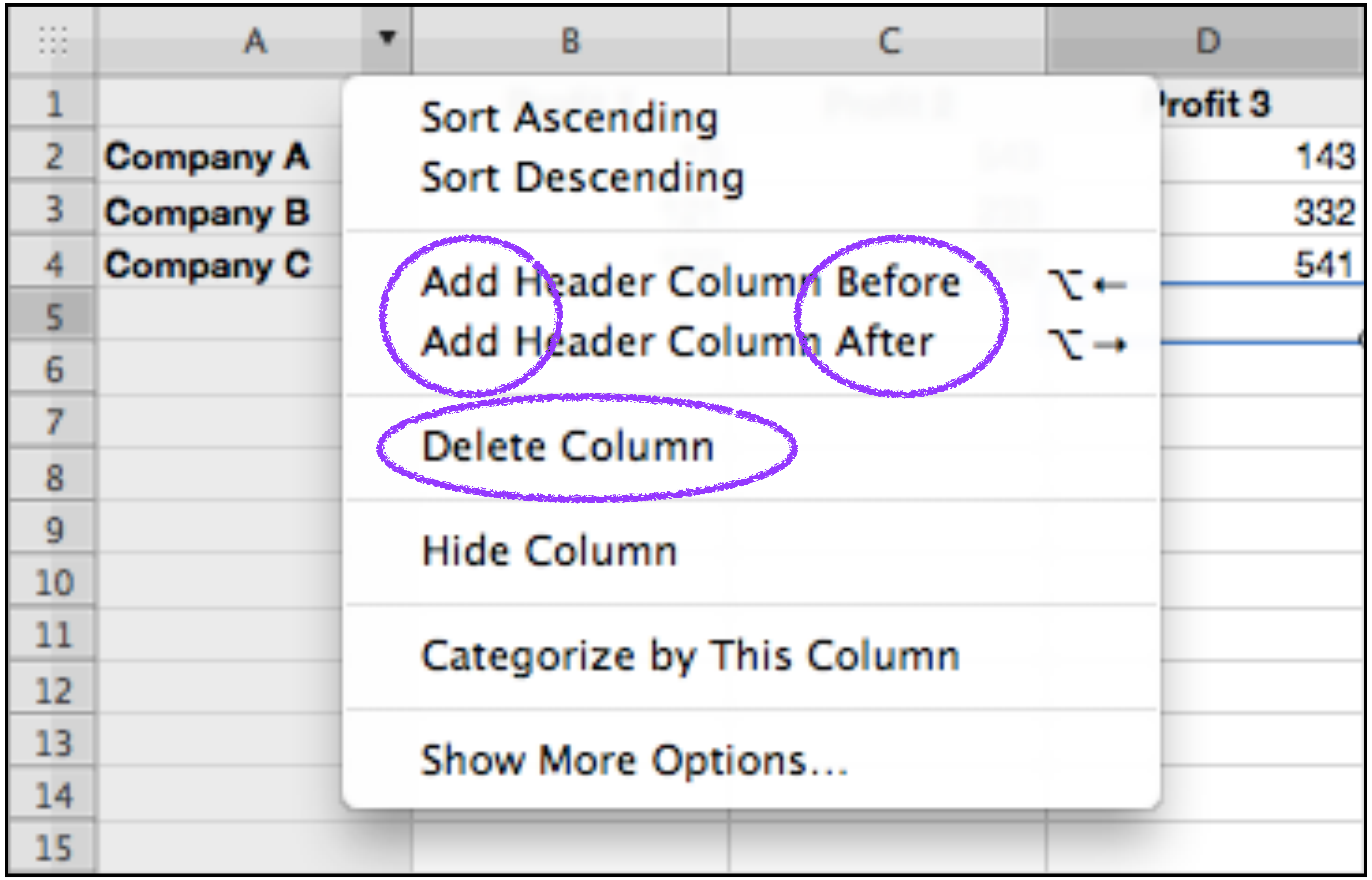
V = type van de bijgehouden data elementen

P = type van de posities

posities zijn niet noodzakelijk numeriek

Geest van de add&delete Operaties

Spreadsheetprogramma



Positionele Lijsten: Voorbeeld

```
(define-record-type event
  (make-event d m n)
  event?
  (d day)
  (m month)
  (n note))

(define event-eq?
  (lambda (event1 event2)
    (and (eq? (day event1) (day event2))
         (eq? (month event1) (month event2)))))
```

P hangt af van de versie die geïmporteerde is

positional-list<event number>

of

positional-list<event pair>

```
(define todo-list (plist:new event-eq?))

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

(plist:add-after! todo-list todo-list-event-1)
(plist:add-after! todo-list todo-list-event-2)
(plist:add-after! todo-list todo-list-event-3)
```

Positionele Lijsten: Voorbeeld (2)

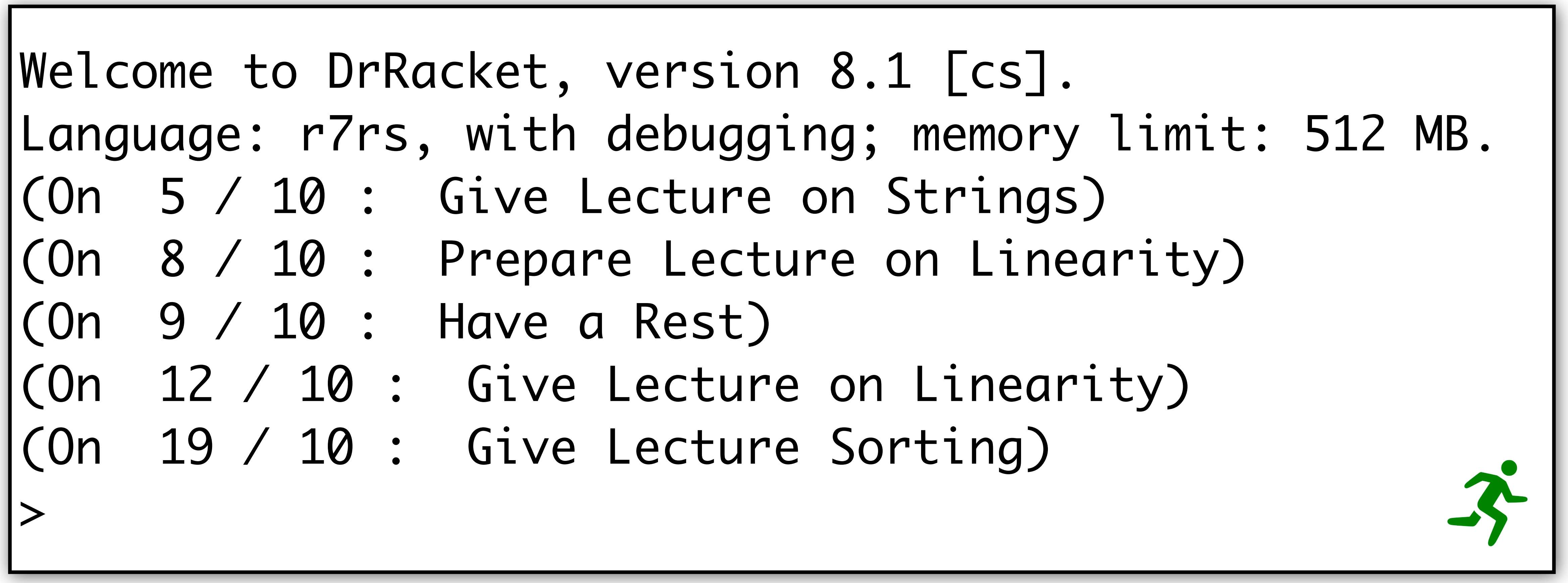
```scheme
(define lecture-2 (plist:find todo-list (make-event 12 10 ‘())))

(plist:add-before! todo-list (make-event 8 10 "Prepare Lecture on Linearity") lecture-2)

(define prepare-lecture (plist:find todo-list (make-event 8 10 ‘())))

(plist:add-after! todo-list (make-event 9 10 "Have a Rest") prepare-lecture)

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



```
Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512 MB.
(On  5 / 10 :  Give Lecture on Strings)
(On  8 / 10 :  Prepare Lecture on Linearity)
(On  9 / 10 :  Have a Rest)
(On  12 / 10 :  Give Lecture on Linearity)
(On  19 / 10 :  Give Lecture Sorting)
>
```

Structuur van de Libraries

We proberen zoveel mogelijk code zo hoog mogelijk te zetten in deze “import boom”. Hoe hoger hoe algemener.

positional-list importeert exact een implementatie: with-sentinel of without-sentinel

```
(define-library (positional-list-adt)
  (export new from-scheme-list positional-list?
          next previous
          map for-each
          find delete! peek update! add-before! add-after!
          first last has-next? has-previous?
          length empty? full?)
  (import (except (scheme base) length list? map for-each)
          ;(a-d positional-list with-sentinel))
          (a-d positional-list without-sentinel))
```

positional-list-with-sentinel importeert exact een implementatie

```
(define-library (positional-list-with-sentinel)
 (export new positional-list? find
         attach-first! attach-last! attach-middl
         detach-first! detach-last! detach-middl
         length empty? full? update! peek
         first last has-next? has-previous? next previous)
(import
 (except (scheme base) length map for-each)
;(a-d positional-list vectorial))
 (a-d positional-list augmented-double-linked))
(begin
```

positional-list-without-sentinel importeert exact een implementatie

```
(define-library (positional-list-without-sentinel)
 (export new positional-list? find
         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)
 (import (except (scheme base) length list? map for-each)
         (a-d positional-list single-linked))
         ;(a-d positional-list double-linked))
(begin
```

```
(define-library (vector-positional-list)
  (expo



  (impo
  (begi
```

```
(define-library (augmented-double-linked-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

```
(define-library (linked-positional-list)
  (expor



  (impor
  (begin
```

```
(define-library (double-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

Structuur van de Libraries

Door de commentaren in de imports te veranderen bepalen we welke implementatie de gebruiker van het ADT zal zien

We proberen zoveel mogelijk code zo hoog mogelijk te zetten in deze "import boom". Hoe hoger hoe algemener

Eerst de operaties die onafhankelijk van de representatie zijn

```
(define-library (positional-list-adt)
  (export new from-scheme-list positional-list?
          next previous
          map for-each
          find delete! peek update! add-before! add-after!
          first last has-next? has-previous?
          length empty? full?)
  (import (except (scheme base) length list? map for-each)
          ;(a-d positional-list with-sentinel))
          (a-d positional-list without-sentinel))
```

```
(define-library (positional-list-with-sentinel)
  (export new positional-list? find
          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)
  (import
   (except (scheme base) length map for-each)
   ;(a-d positional-list vectorial))
   (a-d positional-list augmented-double-linked))
  (begin
```

```
(define-library (vector-positional-list)
  (expo


  (impo
  (begi
```

```
(define-library (augmented-double-linked-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

```
(define-library (positional-list-without-sentinel)
  (export new positional-list? find
          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)
  (import (except (scheme base) length list? map for-each)
          (a-d positional-list single-linked))
          ;(a-d positional-list double-linked))
 (begin
```

```
(define-library (linked-positional-list)
  (expor


  (impor
  (begin
```

```
(define-library (double-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

Algemene Procedures

O(n)

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

O(n)

```
(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)
```

O(n)

```
(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))))
```

Algemene Procedures: Sequentieel Zoekalgoritme

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

O(n)

Algemene Procedures

O(?)

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

O(?)

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

O(?)

```
(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)
```

Performantie

	gemeenschappelijk
`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 per implementatie

representatie

```
(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) ...)
```

Structuur van de Libraries

We proberen zoveel mogelijk code zo hoog mogelijk te zetten in deze “import boom”. Hoe hoger hoe algemener

Door de commentaren in de imports te veranderen bepalen we welke implementatie de gebruiker van het ADT zal zien

Implementatie #1: De vectoriële representatie

```
(define-library (positional-list-adt)
  (export new from-scheme-list positional-list?
          next previous
          map for-each
          find delete! peek update! add-before! add-after!
          first last has-next? has-previous?
          length empty? full?)
  (import (except (scheme base) length list? map for-each)
          ;(a-d positional-list with-sentinel))
          (a-d positional-list without-sentinel))
```

```
(define-library (positional-list-with-sentinel)
  (export new positional-list? find
          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)
  (import
   (except (scheme base) length map for-each)
   ;(a-d positional-list vectorial))
   (a-d positional-list augmented-double-linked))
  (begin
```

```
(define-library (positional-list-without-sentinel)
  (export new positional-list? find
          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)
  (import (except (scheme base) length list? map for-each)
          ;(a-d positional-list single-linked))
          (a-d positional-list vectorial))
  ;(a-d positional-list double-linked))
  ;(a-d positional-list augmented-double-linked))
  (begin
```

```
(define-library (vector-positional-list)
  (expo
  (impo
  (begi
```

```
(define-library (augmented-double-linked-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

```
(define-library (linked-positional-list)
  (expor
  (impor
  (begin
```

```
(define-library (double-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

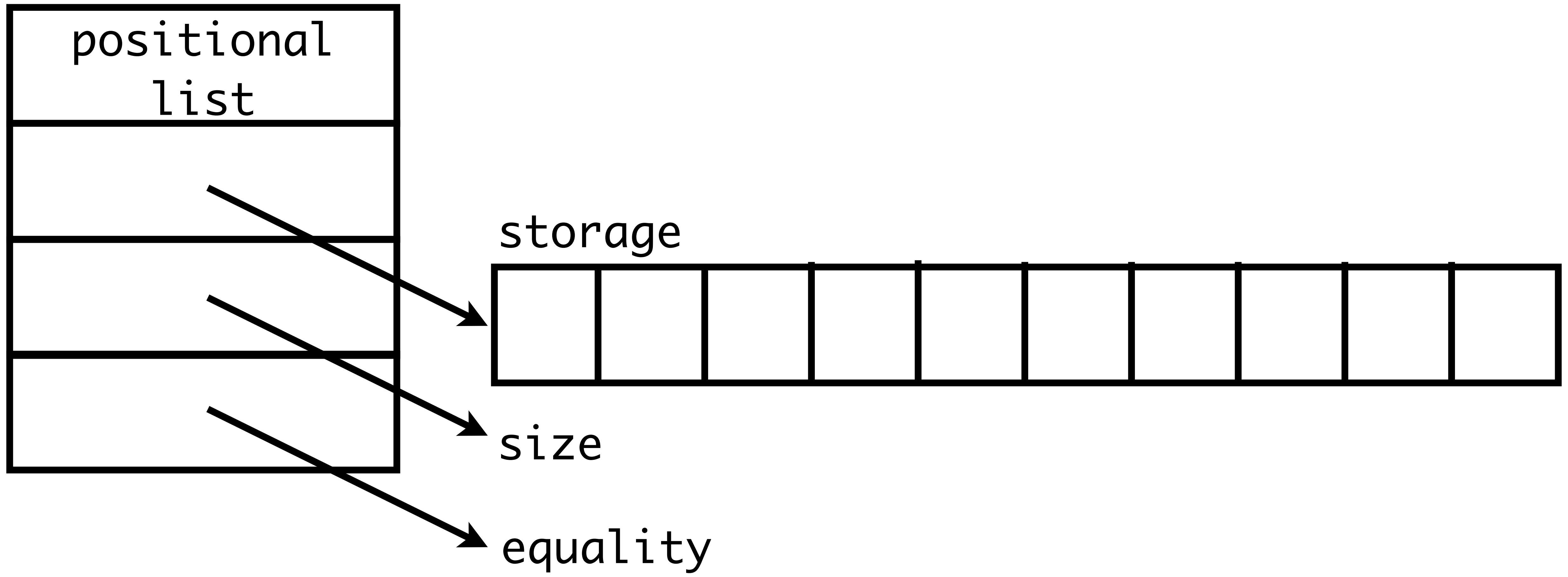
De Vectoriële Implementatie

representatie

```
(define capacity 10)

(define-record-type positional-list
  (make v s e)
  positional-list?
  (v storage storage!)
  (s size size!)
  (e equality))

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



De Vectoriële Implementatie

O(1)

verificatie

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

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

(define (full? plst)
  (= (size plst) capacity))
```

navigatie

O(1)

```
(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)))
```

De Vectoriële Implementatie: Storage Moving

O(n)

```
(define (storage-move-right vector i j)
  (do ((idx j (- idx 1)))
    ((< idx i))
    (vector-set! vector (+ idx 1) (vector-ref vector idx))))

(define (storage-move-left vector i j)
  (do ((idx i (+ idx 1)))
    ((> idx j))
    (vector-set! vector (- idx 1) (vector-ref vector idx))))
```

De Vectoriële Implementatie

O(1)

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

O(1)

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

manipulatie

O(n)

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

O(1)

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

O(n)

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

O(n)

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

O(n)

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

O(1)

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

De Vectoriële Implementatie: Conclusie

	vectorieel	enkelgelinkt	dubbelgelinkt	dubbelgelinkt-2
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)			$O(max(f_{\text{detach-first!}}, f_{\text{detach-last!}}, f_{\text{detach-middle!}}))$
add-before!	O(n)			$O(max(f_{\text{attach-first!}}, f_{\text{previous}}, f_{\text{attach-middle!}}))$
add-after!	O(n)			$O(max(f_{\text{attach-middle!}} f_{\text{attach-last!}}))$

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

Structuur van de Libraries

We proberen zoveel mogelijk code zo hoog mogelijk te zetten in deze “import boom”. Hoe hoger hoe algemener

Implementatie #2: De enkelgelinkte representatie

Door de commentaren in de imports te veranderen bepalen we welke implementatie de gebruiker van het ADT zal zien

```
(define-library (positional-list-adt)
  (export new from-scheme-list positional-list?
          next previous
          map for-each
          find delete! peek update! add-before! add-after!
          first last has-next? has-previous?
          length empty? full?)
  (import (except (scheme base) length list? map for-each)
          ;(a-d positional-list with-sentinel))
          (a-d positional-list without-sentinel))
```

```
(define-library (positional-list-with-sentinel)
  (export new positional-list? find
          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)
  (import
   (except (scheme base) length map for-each)
   ;(a-d positional-list vectorial))
   (a-d positional-list augmented-double-linked))
  (begin
```

```
(define-library (positional-list-without-sentinel)
  (export new positional-list? find
          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)
  (import (except (scheme base) length list? map for-each)
          ;(a-d positional-list single-linked))
          (a-d positional-list vectorial))
  ;(a-d positional-list double-linked))
  ;(a-d positional-list augmented-double-linked))
  (begin
```

```
(define-library (vector-positional-list)
  (expo
  (impo
  (begi
```

```
(define-library (augmented-double-linked-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

```
(define-library (linked-positional-list)
  (expo
  (impo
  (begi
```

```
(define-library (double-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

De Enkelgelinkte Implementatie

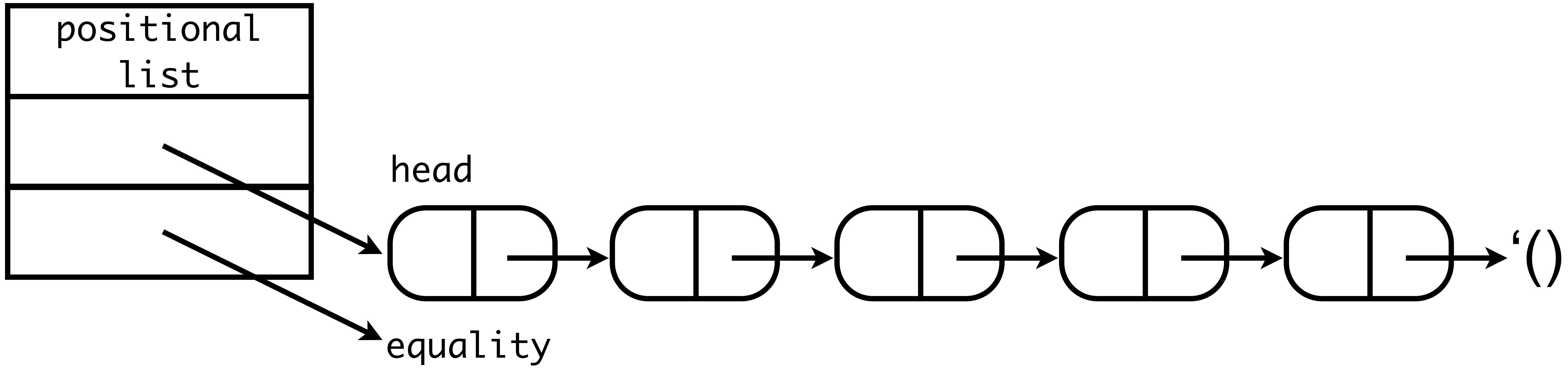
representatie

```
(define-record-type positional-list
  (make h e)
  positional-list?
  (h head head!)
  (e equality))

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

list-node abstracties

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



De Enkelgelinkte Implementatie

verificatie

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

O(n)

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

O(1)

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

O(1)

manipulatie

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

O(1)

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

O(1)

De Enkelgelinkte Implementatie

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

(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)))
```

O(1)

navigatie

```
(define (iter-from-head-until plst stop?)
  (define frst (head plst))
  (let chasing-pointers
    ((prev '())
     (next frst))
    (if (stop? next)
        prev
        (chasing-pointers
         next
         (list-node-next next)))))
```

O(n)

```
(define (last plst)
  (if (null? (head plst))
      (error "list empty (last)" plst)
      (iter-from-head-until plst null?)))

(define (previous plst pos)
  (if (not (has-previous? plst pos))
      (error "list has no previous (previous)" plst)
      (iter-from-head-until plst (lambda (node) (eq? pos node)))))
```

O(n)

O(n)

De Enkelgelinkte Implementatie

manipulatie

O(1)

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

O(1)

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

O(n)

```
(define (attach-last! plst val)
  (define last (iter-from-head-until plst null?))
  (define node (make-list-node val '()))
  (define frst (head plst))
  (if (null? frst)
      (head! plst node) ; last is also first
      (list-node-next! last node)))
```

O(1)

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

O(n)

```
(define (detach-middle! plst pos)
  (define next (list-node-next pos))
  (define prev (iter-from-head-until
                plst
                (lambda (node) (eq? pos node))))
  (list-node-next! prev next))
```

O(n)

```
(define (detach-last! plst pos)
  (define frst (head plst))
  (define scnd (list-node-next frst))
  (if (null? scnd) ; last is also first
      (head! plst '())
      (list-node-next! (iter-from-head-until
                        plst
                        (lambda (last) (not (has-next? plst last))))
                       '())))
```

De Enkelgelinkte Implementatie: Conclusie

	vectorieel	enkelgelinkt	dubbelgelinkt	dubbelgelinkt-2
`length`	O(1)	O(n)		
`first`	O(1)	O(1)		
`last`	O(1)	O(n)		
`has-next?`	O(1)	O(1)		
`has-previous?`	O(1)	O(1)		
`next`	O(1)	O(1)		
`previous`	O(1)	O(n)		
`peek`	O(1)	O(1)		
`update!`	O(1)	O(1)		
`delete!`	O(n)	O(n)		$O(\mathtt{max}(f_{\mathtt{detach-first!}},\ f_{\mathtt{detach-last!}},\ f_{\mathtt{detach-middle!}}))$
`add-before!`	O(n)	O(n)		$O(\mathtt{max}(f_{\mathtt{attach-first!}},\ f_{\mathtt{previous}},\ f_{\mathtt{attach-middle!}}))$
`add-after!`	O(n)	O(1)		$O(\mathtt{max}(f_{\mathtt{attach-middle!}}\ f_{\mathtt{attach-last!)}}))$

*O(n)

Achterwaartse navigatie traag. Grote flexibiliteit qua grootte.
Toevoegen en weglaten traag i.h.a.

Structuur van de Libraries

We proberen zoveel mogelijk code zo hoog mogelijk te zetten in deze “import boom”. Hoe hoger hoe algemener

Implementatie #3: De dubbelgelinkte representatie

Door de commentaren in de imports te veranderen bepalen we welke implementatie de gebruiker van het ADT zal zien

```
(define-library (positional-list-adt)
  (export new from-scheme-list positional-list?
          next previous
          map for-each
          find delete! peek update! add-before! add-after!
          first last has-next? has-previous?
          length empty? full?)
  (import (except (scheme base) length list? map for-each)
          ;(a-d positional-list with-sentinel))
          (a-d positional-list without-sentinel))
```

```
(define-library (positional-list-with-sentinel)
  (export new positional-list? find
          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)
  (import
   (except (scheme base) length map for-each)
   ;(a-d positional-list vectorial))
   (a-d positional-list augmented-double-linked))
  (begin
```

```
(define-library (positional-list-without-sentinel)
  (export new positional-list? find
          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)
  (import (except (scheme base) length list? map for-each)
          ;(a-d positional-list single-linked))
          (a-d positional-list vectorial))
  ;(a-d positional-list double-linked))
  ;(a-d positional-list augmented-double-linked))
  (begin
```

```
(define-library (vector-positional-list)
  (expo
  (impo
  (begi
```

```
(define-library (augmented-double-linked-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

```
(define-library (linked-positional-list)
  (export
  (import
  (begin
```

```
(define-library (double-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

De Dubbelgelinkte Implementatie

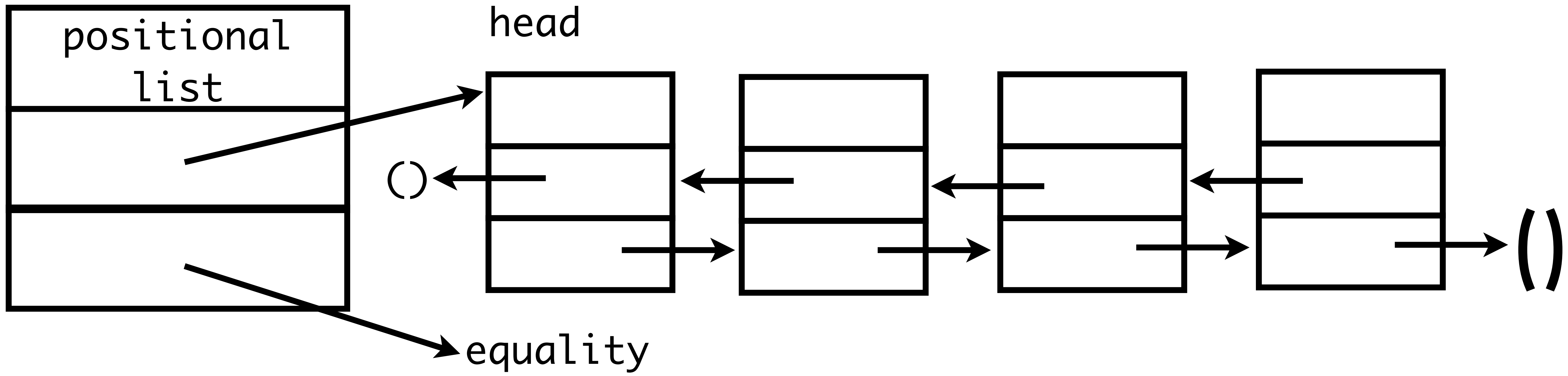
representatie

```
(define-record-type positional-list
  (make h e)
  positional-list?
  (h head head!)
  (e equality))

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

Een lokale abstractie voor list nodes

```
(define-record-type list-node
  (make-list-node v p n)
  list-node?
  (v list-node-val list-node-val!)
  (p list-node-prev list-node-prev!)
  (n list-node-next list-node-next!))
```



De Dubbelgelinkte Implementatie

verificatie

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

(define (full? plst)
  ...)

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

manipulatie

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

(define (peek plst pos)
  ...)
```

navigatie

```
(define (first plst)
  ...)

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

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

(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 (last plst)
  ...)
```

O(1)

```
(define (previous plst pos)
  (if (not (has-previous? plst pos))
    (error "list has no previous (previous)" plst)
    (iter-from-head-until plst (lambda (node) (eq? pos node)))))
```

De Dubbelgelinkte Implementatie

manipulatie

previous pointer goed zetten

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

(define (attach-last! plst val)
  (define last (iter-from-head-until plst null?))
  (define node (make-list-node val '()))
  (define frst (head plst))
  (if (null? frst)
      (head! plst node) ; last is also first
      (list-node-next! last node)))
```

O(1)

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

O(1)

```
(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)))
```

O(n)

```
(define (attach-last! plst val)
  (define last (iter-from-head-until plst null?))
  (define node (make-list-node val last '()))
  (define frst (head plst))
  (if (null? frst)
      (head! plst node) ; last is also first
      (list-node-next! last node)))
```

De Dubbelgelinkte Implementatie

manipulatie

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

(define (detach-middle! plst pos)
  (define next (list-node-next pos))
  (define prev (iter-from-head-until
                plst
                (lambda (node) (eq? pos node))))
  (list-node-next! prev next))

(define (detach-last! plst pos)
  (define frst (head plst))
  (define scnd (list-node-next frst))
  (if (null? scnd) ; last is also first
      (head! plst '())
      (list-node-next! (iter-from-head-until
                        plst
                        (lambda (last) (not (has-next? plst last)))
                        '()))))
```

previous pointer goed zetten

O(1)

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

(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))

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

previous pointer gebruiken

O(1)

previous pointer gebruiken

De Dubbelgelinkte Implementatie

	vectorieel	enkelgelinkt	dubbelgelinkt	dubbelgelinkt-2
length	O(1)	O(n)	O(n)	
first	O(1)	O(1)	O(1)	
last	O(1)	O(n)	O(n)	
has-next?	O(1)	O(1)	O(1)	
has-previous?	O(1)	O(1)	O(1)	
next	O(1)	O(1)	O(1)	
previous	O(1)	O(n)	O(1)	
peek	O(1)	O(1)	O(1)	
update!	O(1)	O(1)	O(1)	
delete!	O(n)	O(n)	O(1)	$O(max(f_{\text{detach-first!}},\ f_{\text{detach-last!}},\ f_{\text{detach-middle!}}))$
add-before!	O(n)	O(n)	O(1)	$O(max(f_{\text{attach-first!}},\ f_{\text{previous}},\ f_{\text{attach-middle!}}))$
add-after!	O(n)	O(1)	O(1)	$O(max(f_{\text{attach-middle!}}\ f_{\text{attach-last!)}}))$

*O(n)

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

Structuur van de Libraries

We proberen zoveel mogelijk code zo hoog mogelijk te zetten in deze “import boom”. Hoe hoger hoe algemener

Implementatie #4: De 2’de dubbelgelinkte representatie

Door de commentaren in de imports te veranderen bepalen we welke implementatie de gebruiker van het ADT zal zien

```
(define-library (positional-list-adt)
  (export new from-scheme-list positional-list?
          next previous
          map for-each
          find delete! peek update! add-before! add-after!
          first last has-next? has-previous?
          length empty? full?)
  (import (except (scheme base) length list? map for-each)
          ;(a-d positional-list with-sentinel))
          (a-d positional-list without-sentinel))
```

```
(define-library (positional-list-with-sentinel)
  (export new positional-list? find
          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)
  (import
   (except (scheme base) length map for-each)
   ;(a-d positional-list vectorial))
   (a-d positional-list augmented-double-linked))
  (begin
```

```
(define-library (positional-list-without-sentinel)
  (export new positional-list? find
          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)
  (import (except (scheme base) length list? map for-each)
          ;(a-d positional-list single-linked))
          (a-d positional-list vectorial))
  ;(a-d positional-list double-linked))
  ;(a-d positional-list augmented-double-linked))
  (begin
```

```
(define-library (vector-positional-list)
  (expo
  (impo
  (begi
```

```
(define-library (augmented-double-linked-positional-list)
  (export new positional-list? 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)
  (import (except (scheme base) length))
  (begin
```

```
(define-library (linked-positional-list)
  (expor
  (impor
  (begin
```

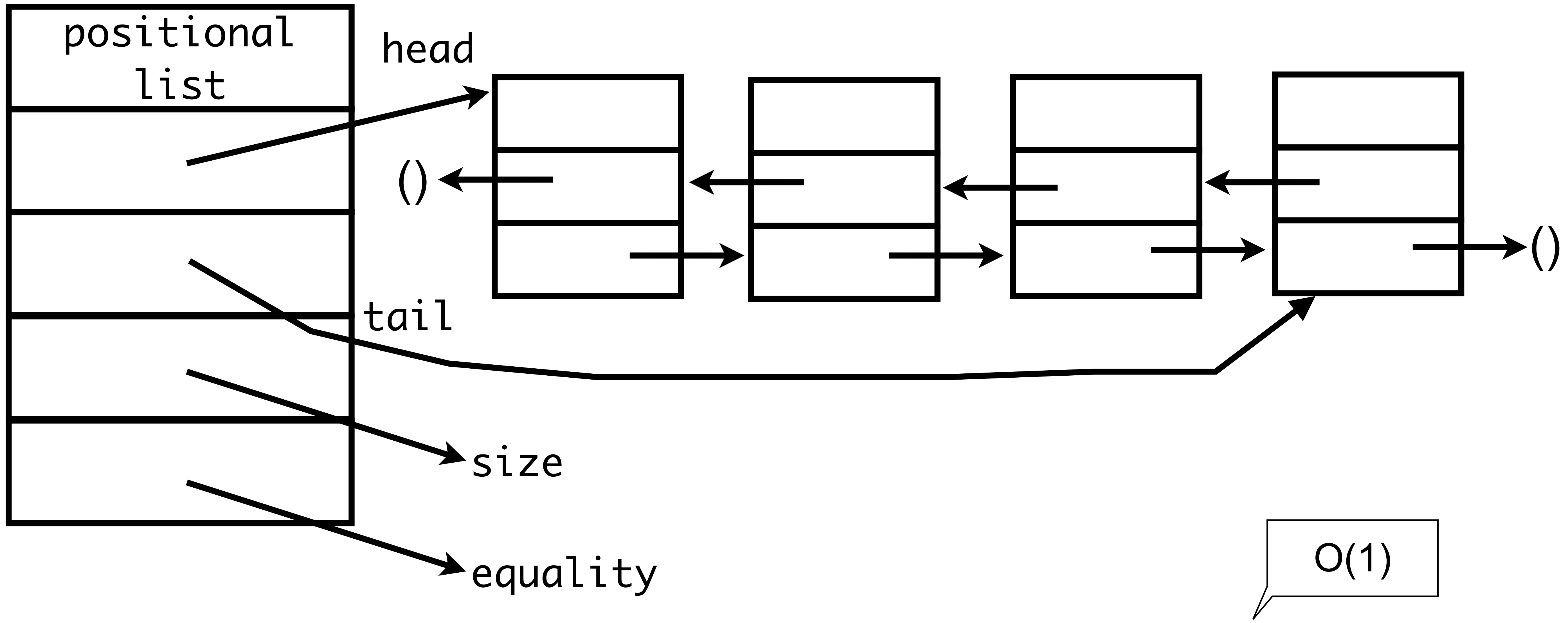
```
(define-library (double-positional-list)
  (export new positional-list? equality
          attach-first! attach-last! attach-middle!
          detach-first! detach-last! detach-middle!
          length empty? full? update! peek
          first last has-next? has-previous? next previous)
  (import (except (scheme base) length))
  (begin
```

De Verbeterde Dubbelgelinkte Implementatie

Abstracties voor list nodes blijven identiek

representatie

```
(define-record-type positional-list
  (make h t s e)
  positional-list?
  (h head head!)
  (t tail tail!)
  (s size size!)
  (e equality))
```



O(1)

navigatie

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

O(1)

verificatie

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

manipulatie

De Verbeterde Dubbelgelin

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

(define (attach-last! plst val)
  (define last (iter-from-head-until plst null?))
  (define node (make-list-node val last '()))
  (define frst (head plst))
  (if (null? frst)
      (head! plst node) ; last is also first
      (list-node-next! last node)))
```

Header up-to-date houden!

```
(define (attach-first! plst val)
  (define frst (head plst))
  (define node (make-list-node val '() frst))
  (head! plst node)
  (if (not (null? frst))
      (list-node-prev! frst node)
      (tail! plst node)) ; first null => last null
  (size! plst (+ 1 (size plst))))

(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)
      (tail! plst node)); next null => new last
  (size! plst (+ 1 (size plst))))

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

O(1)

O(1)

O(1)

De Verbeterde Dubbelgelinkte

manipulatie

Header up-to-date houden!

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

(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))

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

last pointer gebruiken

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

(define (detach-last! plst pos)
  (define frst (head plst))
  (define scnd (list-node-next frst))
  (define last (tail plst))
  (define penu (list-node-prev last))
  (if (null? scnd) ; last is also first
      (head! plst '())
      (list-node-next! penu '()))
  (tail! plst penu)
  (size! plst (- (size plst) 1)))
```

O(1)

first is the only one

O(1)

O(1)

De Verbeterde Dubbelgelinkte Implementatie

	vectorieel	enkelgelinkt	dubbelgelinkt	dubbelgelinkt-2
`length`	O(1)	O(n)	O(n)	O(1)
`first`	O(1)	O(1)	O(1)	O(1)
`last`	O(1)	O(n)	O(n)	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(1)	O(1)	O(1)

*O(n) *O(n)

Grote Flexibiliteit en alles is in O(1)!

Conclusie: positional-list ADT Implementaties

	vectorieel	enkelgelinkt	dubbelgelinkt	dubbelgelinkt-2
`length`	O(1)	O(n)	O(n)	O(1)
`first`	O(1)	O(1)	O(1)	O(1)
`last`	O(1)	O(n)	O(n)	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(1)	O(1)	O(1)

*O(n)

*O(n)

`find` *blijft in O(n) in elke implementatie!*

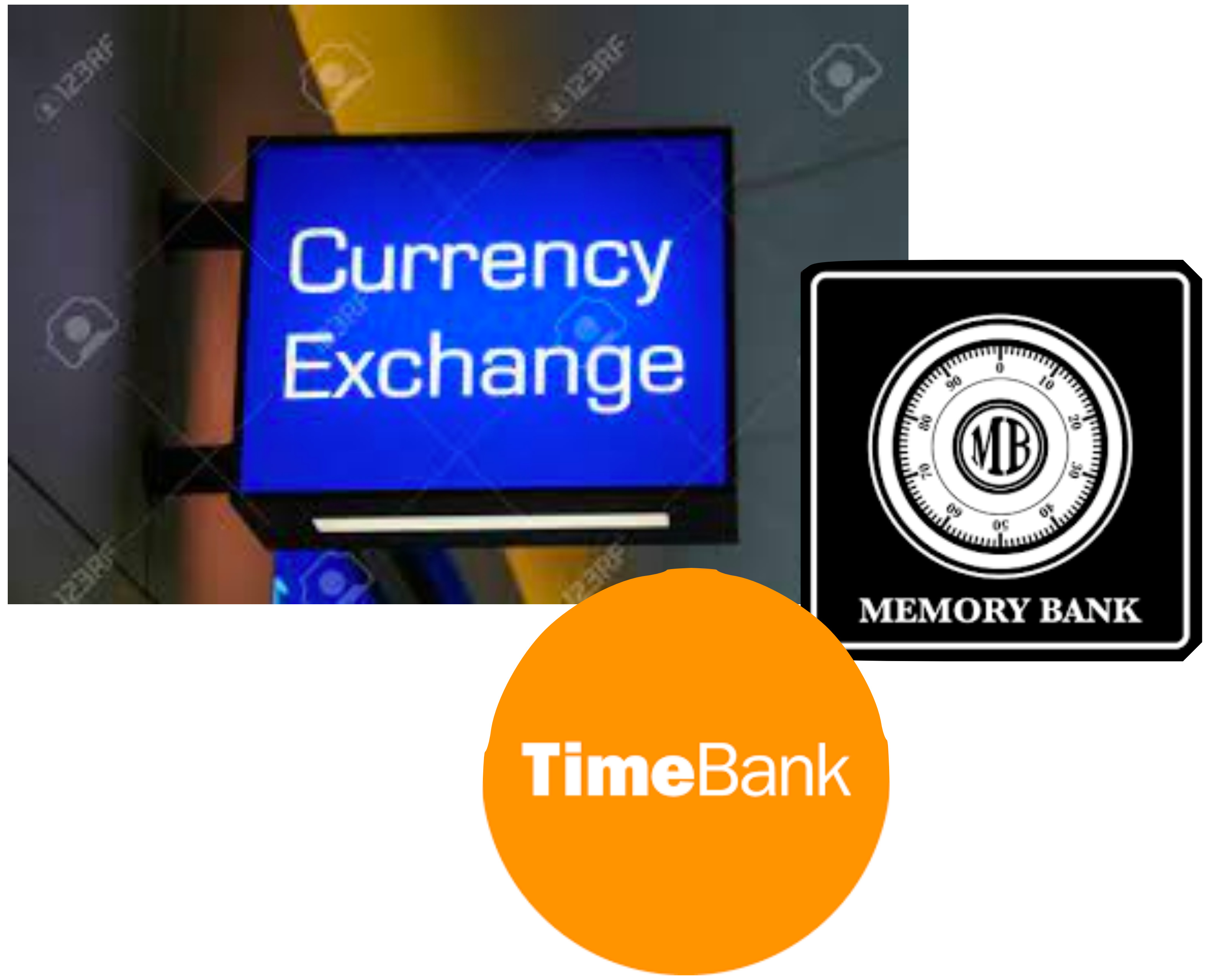
De “Performance Trade-Off”

Een lijst van n datawaarden =

vectorieel : $\Theta(n)$ geheugencellen

enkelgelinkt: $\Theta(2n)$ geheugencellen

dubbelgelinkt: $\Theta(3n)$ geheugencellen



Lijstimplementaties leveren een mooie illustratie van het principe:

Geheugen ⇆ Uitvoeringstijd

3.3 Variaties op Positionele Lijsten

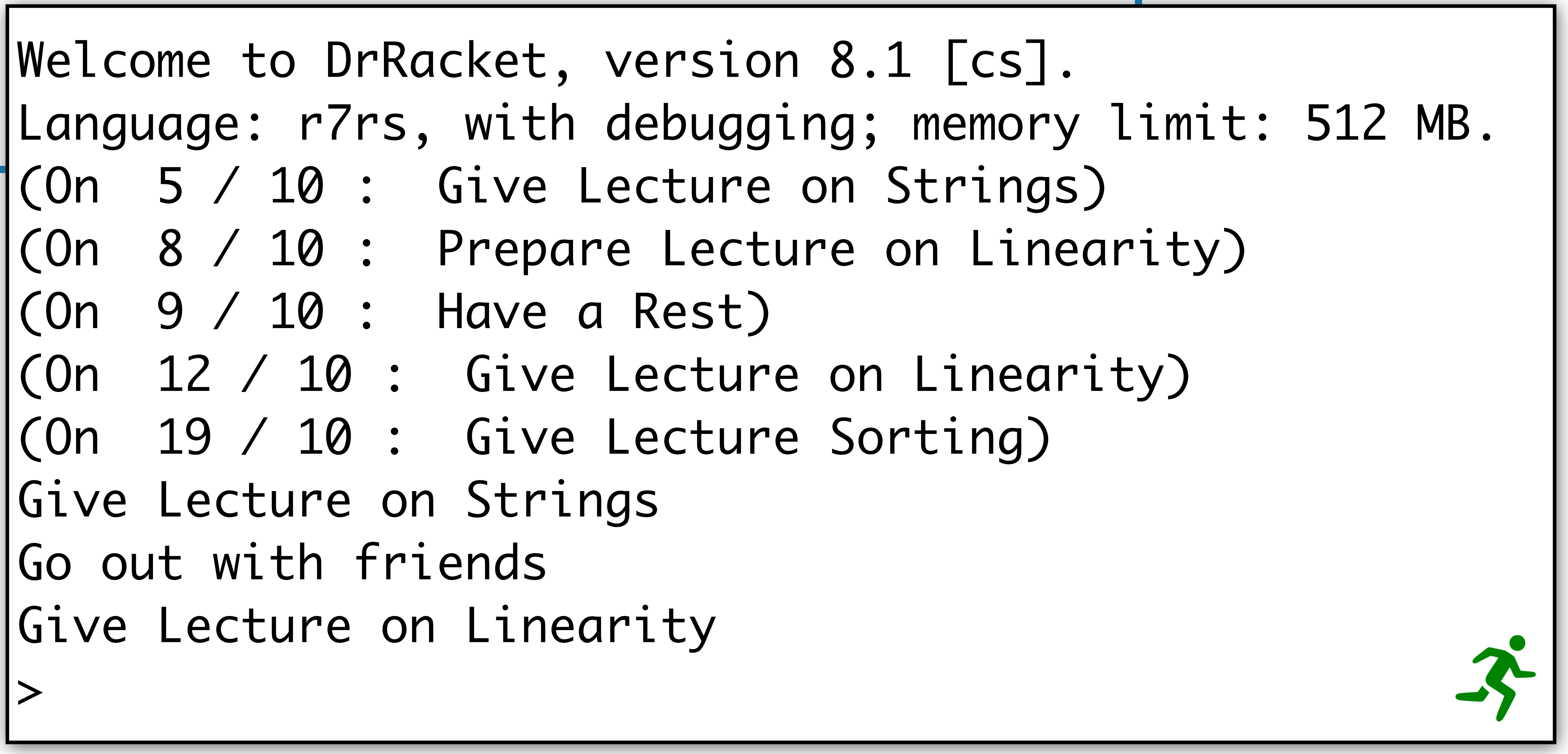
Positionele Lijsten: Constructiefout i/h ADT

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

(define lectures (list (plist:find todo-list (make-event 5 10 '()))
                       (plist:find todo-list (make-event 12 10 '()))
                       (plist:find todo-list (make-event 19 10 '()))))

(define rest (plist:find todo-list (make-event 9 10 '())))
(plist:add-after! todo-list (make-event 11 10 "Go out with friends") rest)
(for-each (lambda (pos)
            (display (note (plist:peek todo-list pos)))
            (newline))
     lectures)
```

Het probleem is dat posities volgens het ADT enkel een relatieve betekenis hebben (d.w.z. next, previous) maar als absolute Scheme waarden toch 'lekken' naar andere datastructuren



```
Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512 MB.
(On  5 / 10 :  Give Lecture on Strings)
(On  8 / 10 :  Prepare Lecture on Linearity)
(On  9 / 10 :  Have a Rest)
(On  12 / 10 :  Give Lecture on Linearity)
(On  19 / 10 :  Give Lecture Sorting)
Give Lecture on Strings
Go out with friends
Give Lecture on Linearity
>
```

2 ≠ Oplossingen

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

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

Men kan in beide gevallen de 4 implementatiestrategieën toepassen: oefening.

Variatie#1: 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 )
```

P is verdwenen uit dit ADT!

De ‘current’ is soms geïnvalideerd

```
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> )
```

De ‘current’ (die ingekapseld is) laat ons toe om verschillende ‘posities’ te manipuleren)

Variatie#2: Gerankte Lijsten

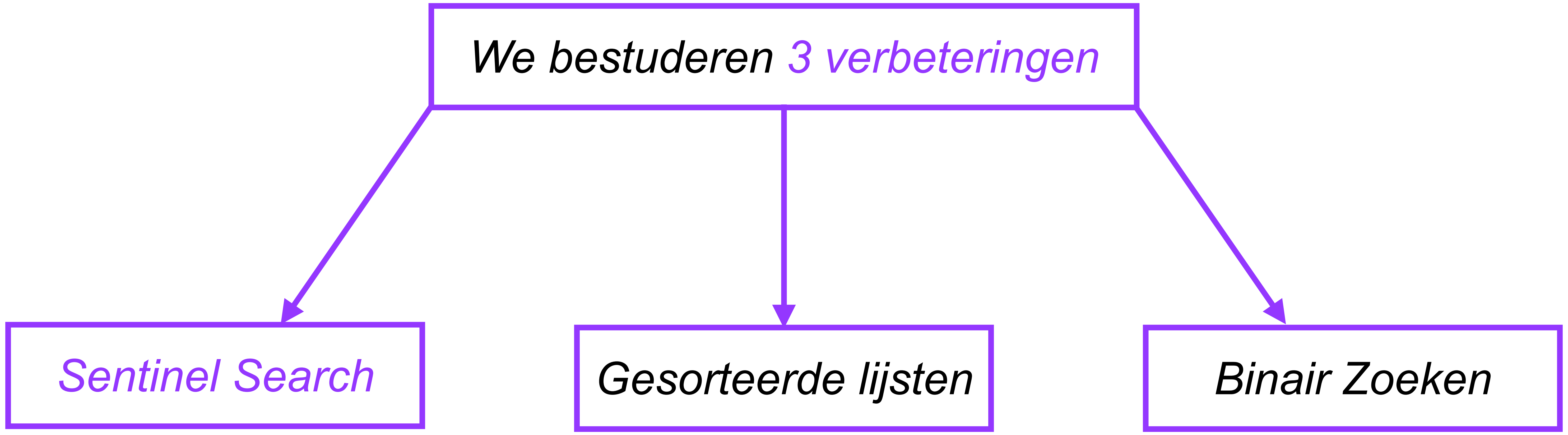
```
ADT ranked-list<V>

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 ∪ {#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> )
```

3.4 Zoeken in Lineaire Datastructuren

Zoeken in Lineaire Datastructuren

Het sequentieel zoekalgoritme voor `find` in de 4 implementaties van positionele lijsten is in O(n).



We bestuderen 3 verbeteringen

Sentinel Search

Gesorteerde lijsten

Binair Zoeken

#1 Sentinel Search

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

```
(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))))
```

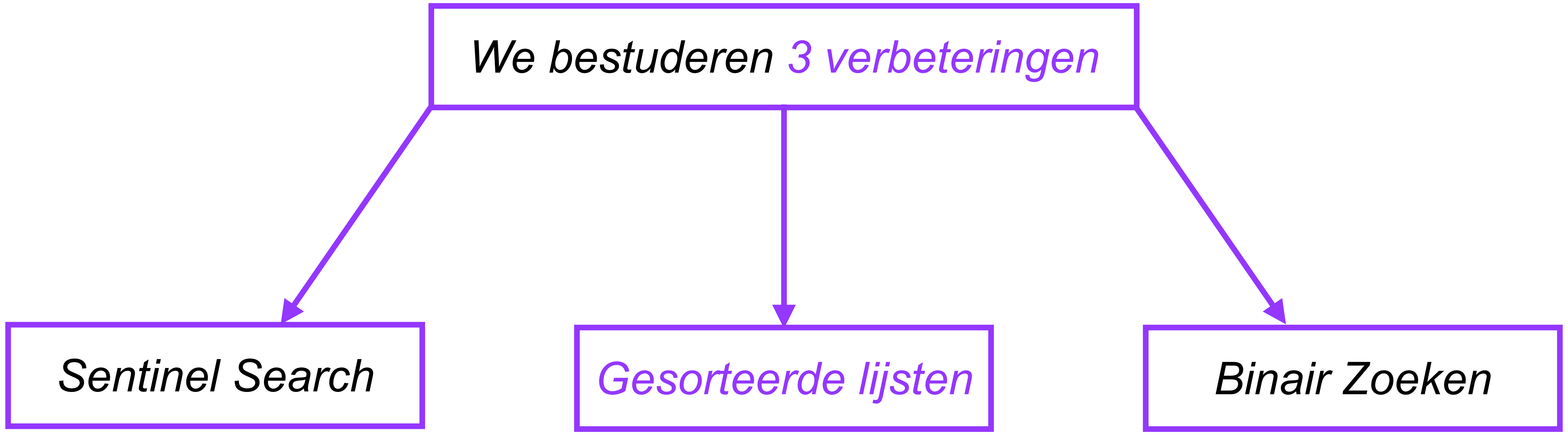
O(n)

attach-last!
moet in O(1) zijn!

“Schildwacht”

Zoeken in Lineaire Datastructuren

Het sequentieel zoekalgoritme voor `find` in de 4 implementaties van positionele lijsten is in O(n).



We bestuderen 3 verbeteringen

Sentinel Search

Gesorteerde lijsten

Binair Zoeken

#2 Gesorteerde Lijsten (1/5)

```
ADT sorted-list<V>

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> )
```

```
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 )
```

De gebruiker krijgt minder controle!

Gesorteerde Lijsten (2/5)

```
(define default-capacity 20)

(define-record-type sorted-list
  (make-sorted-list s c v l e)
  sorted-list?
  (s size size!)
  (c current current!)
  (v storage)
  (l lesser)
  (e equality))

(define (make len <<? ==?)
  (make-sorted-list 0 -1 (make-vector (max default-capacity len)) <<? ==?))
```

representatie

```
(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 1)) (car lst)))))
```

manipulatie

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

Gesorteerde Lijsten (3/5)

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

O(n)

manipulatie

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

O(n)

Gesorteerde Lijsten (4/5)

O(1)

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))))
```

Gesorteerde Lijsten (5/5): Zoeken

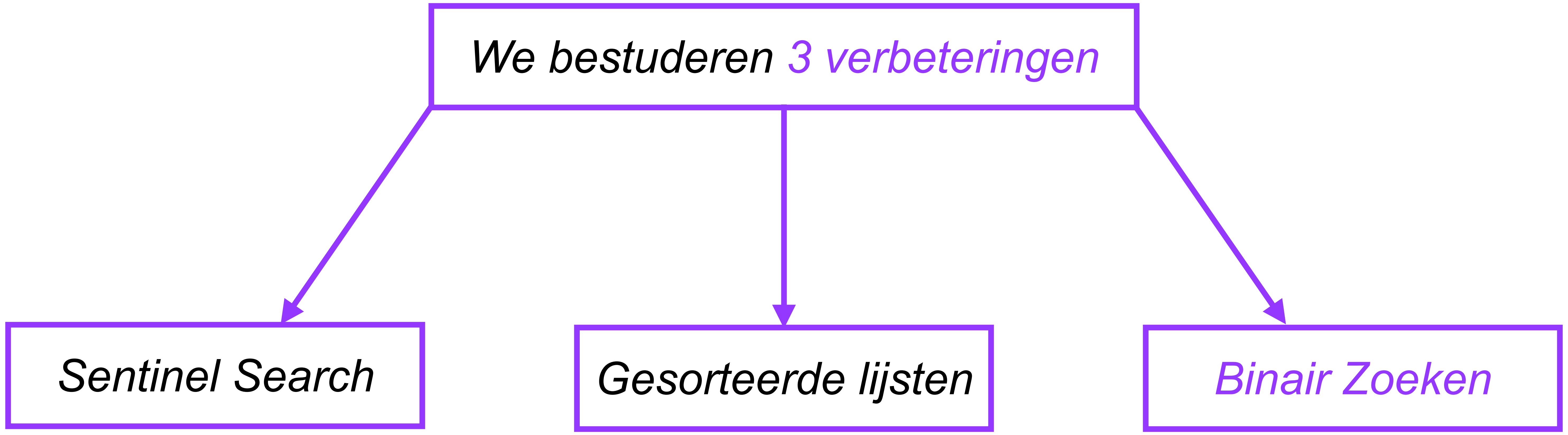
```
(define (find plst key)
  (define ==? (equality plst))
  (if (empty? plst)
      #f
      (let sequential-search
        ((curr (first plst)))
        (cond
          ((==? key (peek plst curr))
           curr)
          ((not (has-next? plst curr))
           #f)
          (else
           (sequential-search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

O(n)

```
(define (find! slst key)
  (define ==? (equality slst))
  (define <<? (lesser slst))
  (define vect (storage slst))
  (define free (size slst))
  (let sequential-search
    ((curr 0))
    (cond ((>= curr free)
           (current! slst -1))
          ((==? key (vector-ref vect curr))
           (current! slst curr))
          ((<<? (vector-ref vect curr) key)
           (sequential-search (+ curr 1)))
          (else
           (current! slst -1))))
  slst)
```

Zoeken in Lineaire Datastructuren

Het sequentieel zoekalgoritme voor `find` in de 4 implementaties van positionele lijsten is in O(n).



We bestuderen 3 verbeteringen

Sentinel Search

Gesorteerde lijsten

Binair Zoeken

Vereist O(1) indexering!

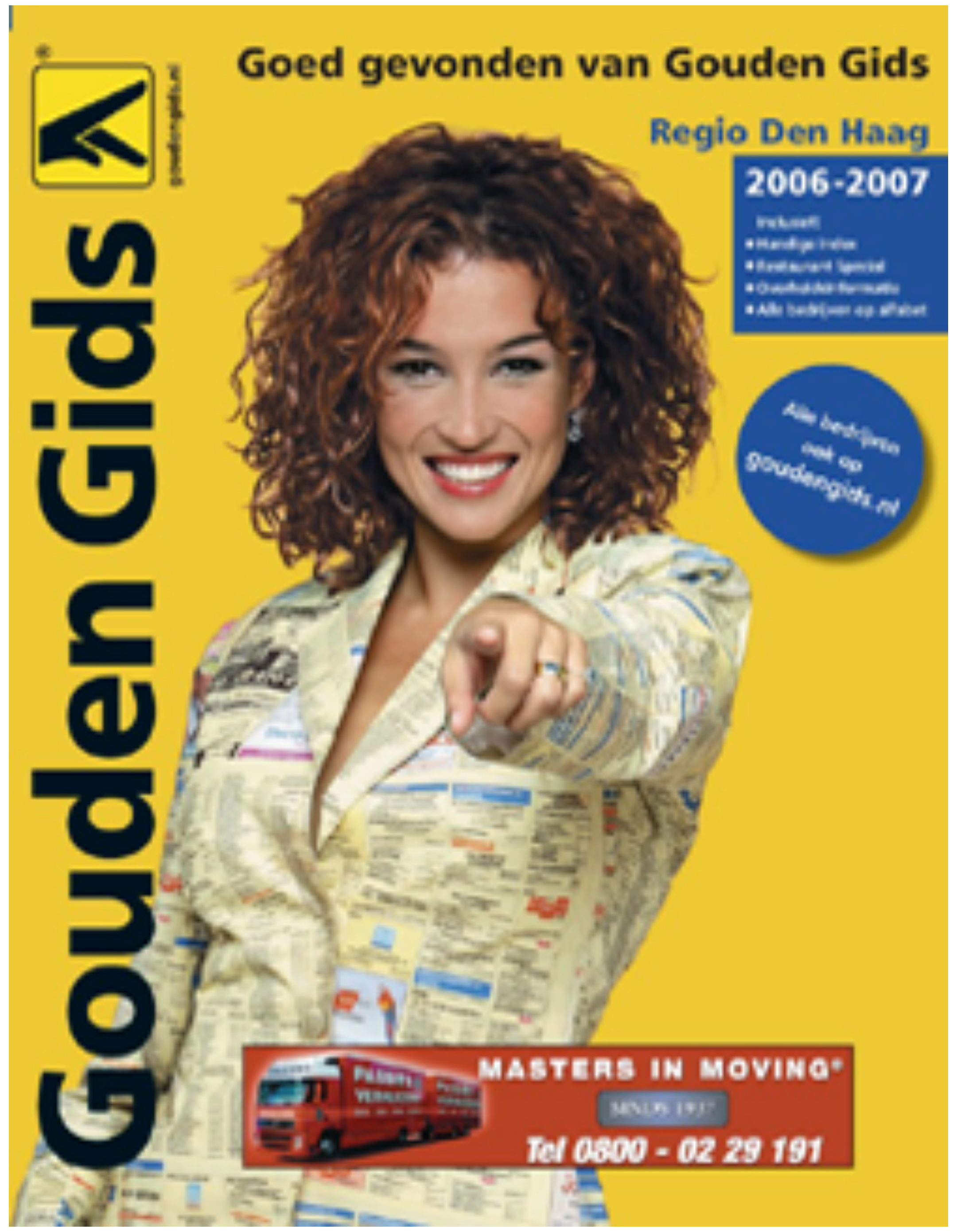
#3 Binair Zoeken

Vereist een gesorteerde rij

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

$O(\log_2(n))$

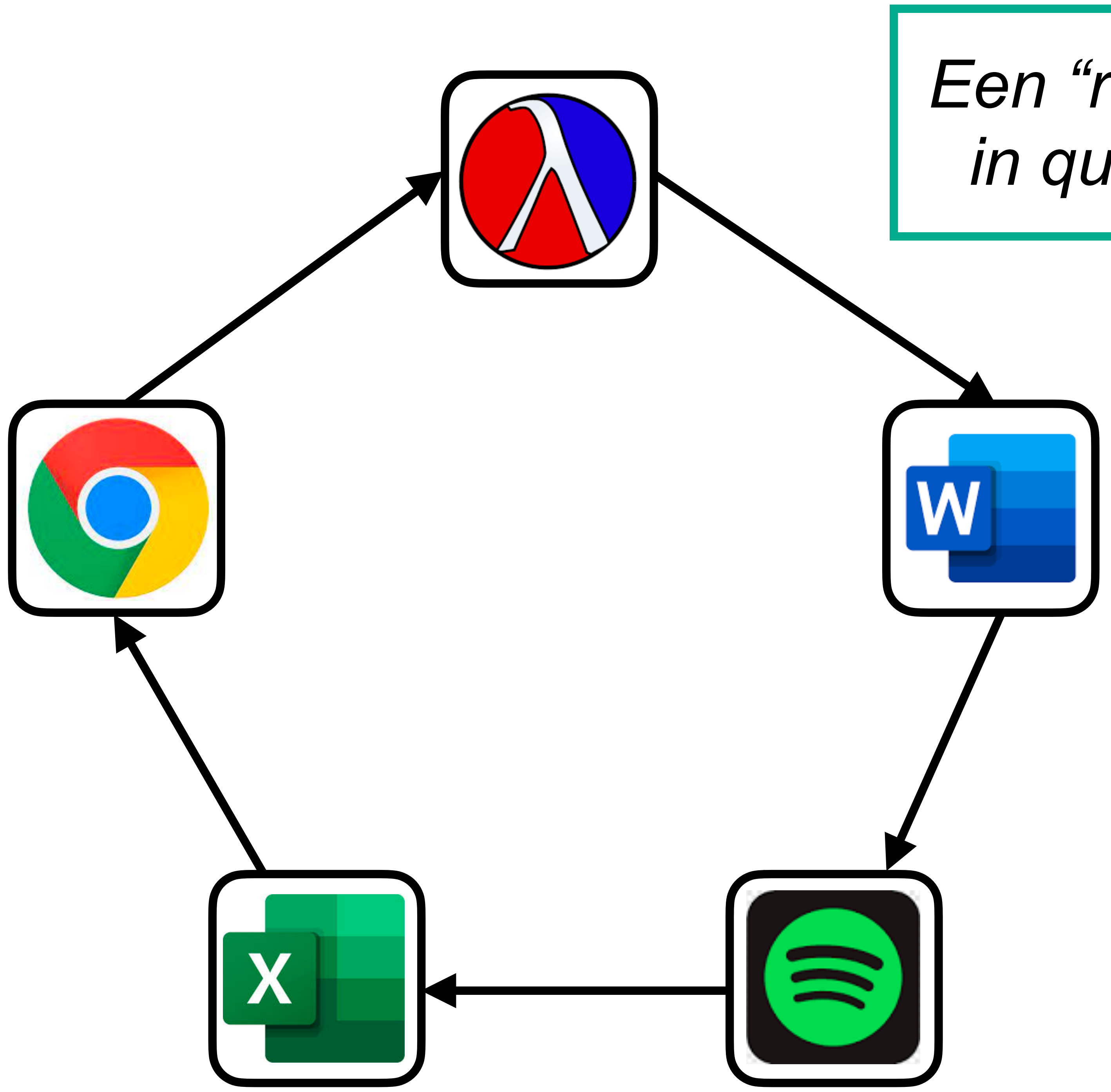
Hoe dikwijls kan je n delen door 2 voor je bij 1 uitkomt? $\log_2(n)$



Ringen

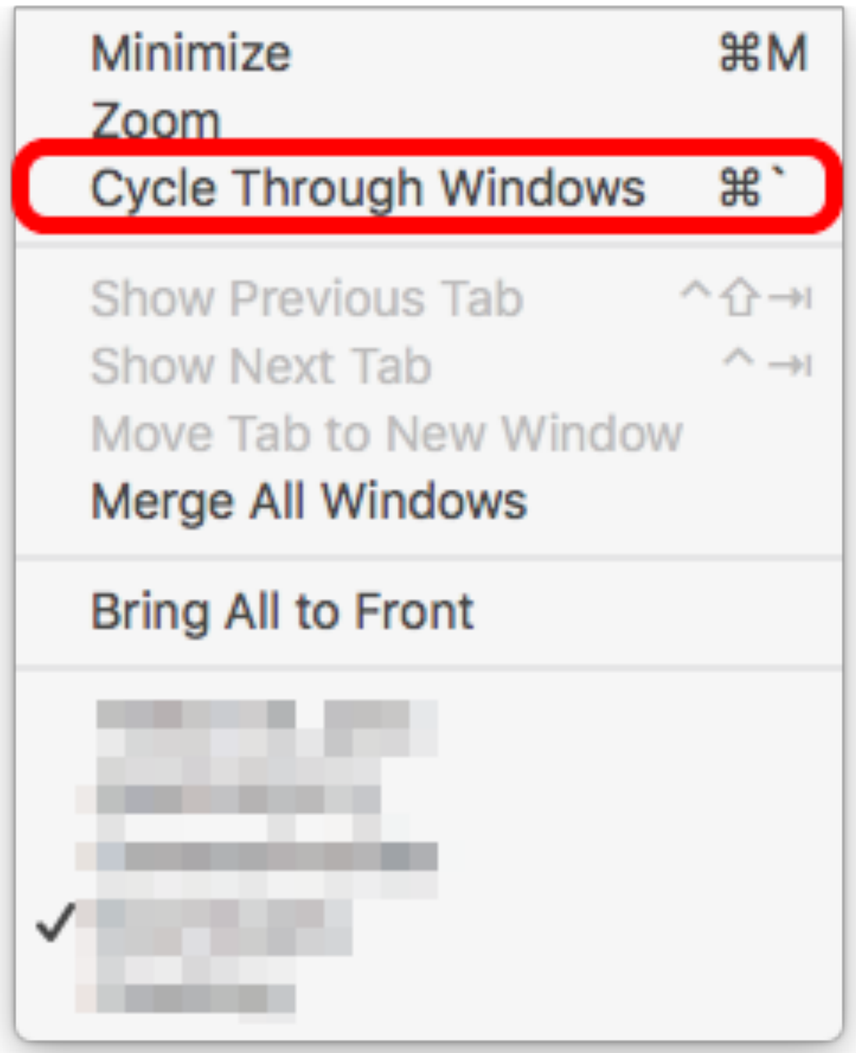
```
ADT ring

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



Een "round robin" task scheduler zit in quasi ieder besturingssysteem

In sommige programma's bestaat de menu-optie "cycle through windows"



Ringen

representatie

```
(define-record-type ring
  (make-ring c)
  ring?
  (c current current!))

(define (new)
  (make-ring '()))

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

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

```
(define (iter-to-previous node)
  (let chasing-pointers
    ((prev node)
     (next (ring-node-next node)))
    (if (eq? node next)
        prev
        (chasing-pointers next (ring-node-next next)))))
```

Kan sneller dubbelgelinkt

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 (iter-to-previous curr)))
  ring)
```

Ringen

manipulatie

```
(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

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

Ringen

```
(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)
```

O(1)

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

O(n)

Kan sneller dubbelgelinkt

manipulatie

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

O(n)

Hoofdstuk 3

3.1 Lineaire Datastructuren uit Scheme

3.2 Positionele Lijsten
3.2.1 Abstracte Definities
3.2.2 Het Positioneel Lijst ADT
3.2.4 Vector Implementatie
3.2.5 Enkelgelinkte Lijsten
3.2.6 Dubbelgelinkte Lijsten
3.2.7 Augmented Dubbelgelinkte Lijsten

3.3 Variaties op Positionele Lijsten
3.3.1 Het Probleem
3.3.2 Lijsten met een Current
3.3.3 Gerankte Lijsten

3.4 Zoeken in Lineaire Datastructuren
3.4.1 Sentinelzoeken
3.4.2 Zoeken in Gesorteerde Lijsten
3.4.3 Binair Zoeken

3.5 Ringen