Hfst 1.1: Terminologie

dinsdag 14 december 2021 18:03

Data en data constructoren

- **Primitieve data**: kan niet worden onderverdeeld in kleinere data
- **Data types**: indicator voor een set van procedures dat toepasbaar zijn op data-values van dat data-type (number, boolean, symbol, character, procedure,…)
  - Data type = naam + operaties
- **Samengestelde datatypes**: datatypes van samengestelde datawaarden (vb. vector, pair, string)
- **Data values**: elementen dat behoren bij een data type (vb. 3, #t,…)
- **Samengestelde data**: onder te verdelen in primitieve data /data die met behulp van een **dataconstructor** geconstrueerd wordt.
  - **Dataconstructor** = geheugen reserveren + onderdelen initialiseren
    - **Letterlijk constrcroren**: vb ', " ", …
    - **Procedurele constructoren**: vb (make-vector), (make-string)
  - **Accessors/getters**: om data elementen uit de data-structuur te lezen (cdr)
  - **Mutators/setters**: data-elementen in de data-structuur aan te passen (set-cdr)
    - **Immutable data-structuur**: hebben geen mutators (strings)
  - **Operations**: operaties op de data-structuur zonder de elementen in de data-structuur weer te geven (member, reverse, …)
- **Data constructor**: procedures die een nieuwe data structuur creëren
  - **Procedurele Data Constructor**: moet opgeroepen worden door de programmeur om een nieuwe datastructuur te creëren in de computergeheugen (cons, make-vector, …)
  - **Literal Data Constructor**: creëren van nieuwe data structuren zonder procedures op te roepen (" ", #())

Algoritmes en algoritmen constructors

- **Algoritme**: alle mechanische procedures om problemen op te lossen of uitvoeren van computational taken (oorspronkelijk voor regels van het decimaal systeem)
- **Primitieve algoritmen**: kan niet onderverdeeld worden in primitieve stappen (in dezelfde programmeertaal)
- **Samengestelde algoritmen**: gecreëerd door meerdere algoritme constructoren te gebruiken (let, let*, lambda, if, …)
- **Proceduretype van de procedure**: Iedere Scheme procedure werkt op **invoer/argumenten** van een zeker datatype en produceert **uitvoer** van een zeker datatype. De combinatie van deze types is het proceduretype van de procedure. -> (argtype-1 … argtype-n -> result-type)

"define" is geen algoritme constructor, maar geeft een naam aan algoritmes

Elementaire computational steps:

- **Generality**: toepasbaar op alle mogelijke inputs van een bepaald data-type
- **Computability**: nummer van duidelijk gespecifieerde computationale stappen (moet programmeerbaar zijn)

Hfst 1.2: Abstractie

dinsdag 14 december 2021 19:37

Data-abstractie: naam geven aan samengestelde data en aan alle operaties die op die data inwerken
Procedurele abstractie: naam geven aan een samengestelde procedure
2 groepen van programmeurs: **gebruikers** & **implementator**

Procedurele abstractie en procedure types

- **Readability**: Gemakkelijker te lezen
- **Adaptability**: Algoritme niet hoeven copy-pasten, gewoon oproepen bij naam
  Vermijden van duplicatie van code
- **Algoritme type/ procedure type**: beschrijving van de verwachte input en output van de data-types van een procedure (number -> number)

Data Abstractie: Abstracte Data Types (ADT's)

- **ADT**: bestaat uit een **naam voor een datatype** en een **reeks proceduretypes** die de procedures beschrijven waarmee data elementen van het ADT gemanipuleerd kunnen worden. Dit zijn proceduretypes voor de constructoren, de accessoren, de mutators en de operaties.
- Een ADT is een abstract concept.
  Gegeven een programmeertaal, dan kiest een implementator voor een **representatie** en een **implementatie**. De implementator probeert de representatie zo goed mogelijk weg te stoppen voor de gebruikers. (bvb met libraries of met objecten)

	Data	Algoritmen
Primitief	Primitieve Data	Ingebouwde Procedures
Samengesteld	Datastructuur	Zelf geschreven Procedures
Abstractie	ADTs	Procedurenamen en Proceduretypes

Implementeren van ADT's in Scheme

Procedure stijl

```
(define-library (name)
  (export functies)
  (import (functies))
  (begin
    ;maak de functies"
    ))
```

Procedure stijl met records

```
(define-record-type naam
  (new a b c ...)
  naam?   :test
  (a naam-a)
  (b naam-b)
  ...)
```

- **Records**: data waarde dat bestaat uit benoemde "fields"

Object georiënteerde stijl: Encapsulation

```
(define (object a b c ...)
  ;maak functies
  (define (dispatch message)
    (cond ((eq? message 'naam-functie) naam-functie)
          ...
          ))
  dispatch)
```

- **Dispatcher**: Encapsulatie van de ADT data waarden in een scheme procedure

Voordelen

- **Encapsulation**: onderhoud implementatie en code
- **Environment management**: alleen de constructor van het ADT's data value is opgeroepen in de globale omgeving
- **Code size**: minder code nodig dan in procedurele stijl, minder accessors nodig door de lexicale scoping regels

Nadelen

- **Encapsulation Breached**: noodzakelijke omleiding door encapsulatie van het object om de voorstelling van het object te krijgen
  Soms meer accessors nodig dat niet deel uit hoeven te maken tot het ADT
  -> encapsulation is niet gegarandeerd
- **Code Complexity**: complexer dan procedurele stijl
  kennis nodig van scheme's scoping rules
  Trager door de cond van de dispatch functie
- **Space Ineffeciency**: als we de aanmaak procedure van het ADT aanroepen worden de locale procedures worden ook aangeroepen

Hfst 1.2: Abstractie (2)

dinsdag 14 december 2021 21:02

Oplsagdatastructuur: Een datastructuur die dient om datawaarden op te slaan met als bedoeling die later weer op te kunnen vragen

Generositeit voor ADTs en Data structuren

- Generic ADT's: opslag data structuren dat onafhankelijk zijn van het data type van de waarden dat men hoort op te slaan
- Generic data structuren: data structuren dat onafhankelijk zijn van het data type van de gegevens dat men opslaat. In Scheme wordt dit met een hogere-orde constructor bewerkstelligd.

Vb. ADT max-o-mem?

Waardes door te geven aan max-o-mem maar enkel de grootste waarde behoud

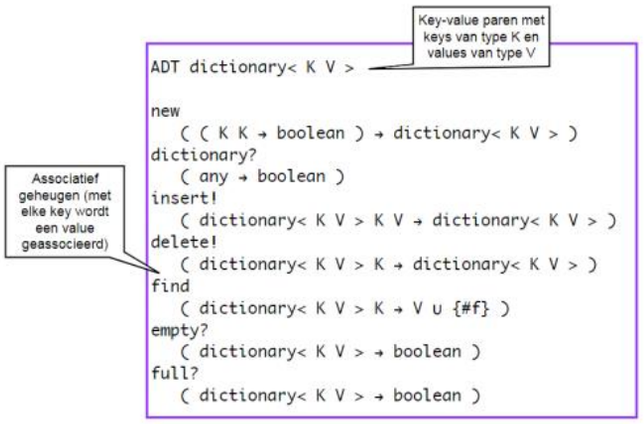
```
ADTmax-o-mem<T>

new          ( ( T T → boolean ) T → max-o-mem<T> )
max-o-mem?   ( any → boolean )
write!       ( max-o-mem<T> T → max-o-mem<T> )
read         ( max-o-mem<T> → T )
```

- write! Schrijft een waarde naar max-o-mem
- read leest de huidige grootste waarde

Dictionary ADT

Dictionary: een datastructuur die "associaties" beheert. Elke associatie bestaat uit een key en een value. De belangrijkste operaties zijn het toevoegen, verwijderen en opzoeken van een associatie. Het snel implementeren van deze 3 operaties is één van de centrale vraagstukken van de cursus.

- Associatief geheugen: Sleutels met waarden associëren



- Records: elementen dat behoren tot storage data structuren (fields + key)
- Fields: individuele data dat de records maken (vb. phone number field, naam field,…)
- We sorteren "records" die bestaan uit "velden". De velden waarop we sorteren heten sleutelvelden of sorteervelden. De andere noemen we satellietvelden.

Hfst 1.3: Performance

woensdag 15 december 2021 11:53

Performance Measure 1: Speed

- Experimentele benadering: (time-it algoritme)

Nadelen

- Non Generality: cijfers i/e tabel van de experimenten is niet algemeen genoeg
- Absoluteness: afhankelijk van de hardware van de computer

The Big Oh, Big Omega en Big Theta Notatie

Analytische techniek: aantal computationele stappen tellen

- Dit levert een functie op, fA(n), de performantiekarakteristiek van algoritme A
- Worst-case Analyse: wanneer de input zo is gevormd dat het algoritme meer stappen moet doen (vb. de laatste tak van de cond)
- Best-case Analyse: meest optimistische uitvoering van het algoritme (vb. telkens de eerste tak van de if/cond)
- Average-case Analyse: gemiddelde van de input genomen -> vaak dicht bij worst-case

n	$log(n)$	$\sqrt{n}$	n	$n.log(n)$	n^2	n^3	2^n
2	1	2	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	3	8	24	64	512	256
16	4	4	16	64	256	4096	65536
32	5	6	32	160	1024	32768	4294967296
64	6	8	64	384	4096	262144	1.85×10^{19}
128	7	12	128	896	16384	2097152	3.40×10^{38}
256	8	16	256	2048	65536	16777216	1.16×10^{77}
512	9	23	512	4608	262144	134217728	1.34×10^{154}
1024	10	32	1024	10240	1048576	1073741824	1.79×10^{308}

Figure 1.5: Growth of Functions of n

We zijn enkel geïnteresseerd in grote n'en en we zijn enkel geïnteresseerd in de kolom waarin een algoritme zich bevindt.

Big Theta

$$\Theta(g(n)) = \{f | \exists c_1, c_2 > 0, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)\}$$

- g(n) is asymtotisch aan f(n)
- Meest precies: houd rekening met upper-bound en lower-bound

Big Oh

$$O(g(n)) = \{f | \exists c, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)\}$$

- Assymtotische upper-bound (minder precies)
- Meest gebruikt zodat O zo klein mogelijk is voor ons algoritme
- Geen rekening met constanten
- Worst case
- Regels
  - $O(c.n) = O(n)$
  - $O(c) = O(1)$
  - $O(log_a(n)) = O(log(n))$

Big Omega

$$\Omega(g(n)) = \{f | \exists c, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)\}$$

- Assymptotische lower-bound
- Best case

Relaties

Voor alle functies f en g geldt:

$f \in \Theta(g)$ als en slechts als $f \in \Omega(g)$ én $f \in O(g)$

Idem voor Ω

$O(c.f) = O(f)$ voor alle constanten c (kies gewoon een andere constante in de definitie)

- Gevolg: $O(c) = O(1)$
- Gevolg: $O(log_a(n)) = O(log_b(n))$

Enkel dominante termen tellen mee:

$O(t_1(n) + t_2(n)) = O(max(t_1(n), t_2(n)))$
$\Omega(t_1(n) + t_2(n)) = \Omega(min(t_1(n), t_2(n)))$

waarbij:

$1 < log(n) < \sqrt{n} < n < n.log(n) < n^{k-1} < n^k < 2^n < n!$

Let soms op:

- In zeldzame gevallen zitten er heel grote constanten in Θ, Ω en O-expressies verborgen
- Θ, Ω en O zijn enkel nuttig om functies van verschillende orde te vergelijken. Eén algoritme van O(n) kan toch véél sneller zijn dan een ander van O(n).

Hfst 1.3 Performance

dinsdag 28 december 2021 12:56

Rekenregels

Vervang overal max door min voor Ω

$O(f_{(define\ n\ E)}(n))$	$= O(1) + O(f_E(n)) = O(f_E(n))$
$O(f_{(set!\ v\ E)}(n))$	$= O(1) + O(f_E(n)) = O(f_E(n))$
$O(f_{(set\text{-}car!\ c\ E)}(n))$	$= O(f_{(set\text{-}cdr!\ c\ E)}(n)) = O(f_E(n))$
$O(f_{(lambda\ args\ body)}(n))$	$= O(1)$
$O(f_{(let\ ((v1\ e1)\ (v2\ e2)\ \ldots\ (vn\ en)\ b1\ b2\ ..\ bk)}(n))$	$= O(max_i(f_{ei}(n)) + max_j(f_{bj}(n))) = O(max_{\{i,j\}}\{ f_{ei}(n), f_{bj}(n) \})$
$O(f_{(begin\ e1\ \ldots\ en)}(n))$	$= O(f_{e1}(n) + \ldots + f_{en}(n)) = O(max_i(f_{ei}(n)))$
$O(f_{(cond\ ((c1\ a1)\ \ldots\ (cn\ an)))}(n))$	$= O(max_i\ \{f_{ci}(n), f_{ai}\}(n)\})$
$O(f_{(if\ c\ t\ e)}(n))$	$= O(max\{f_c(n), f_t(n), f_e(n)\})$
$O(f_{(p\ a1\ \ldots\ an)}(n))$ voor $p \in \{ +, -, eq?, vector\text{-}ref, vector\text{-}set, car, cdr, \ldots \}$	$= O(1) + O(f_{ai}(n)) + \ldots + O(f_{an}(n))$
$O(f_{(f\ a1\ \ldots\ an)}(n))$	$= O(f_f(n)+f_{a1}(n)+ \ldots + f_{an}(n))$

niet-recursieve Scheme procedures:

- Bepaal O/Ω voor alle deelexpressies van de body en neem de som, t.t.z. het maximum/minimum.

Vuistregel: voor een recursieve Scheme procedure

- Versie 1: bepaal $O(b(n))$ voor de body en bepaal je $O(r(n))$ voor het aantal recursieve oproepen. De procedure is dan $O(b(n).r(n))$
- Versie 2: bepaal je $O(r(n))$ voor het aantal recursieve oproepen. Verder bepaal je $O(b(i))$ voor de i'de keer dat de body wordt uitgevoerd. De volledige procedure is dan $O(\sum_{i=1}^{r(n)} b(i))$

De eerste versie is een speciaal geval:
$b(i) = b(n)\ \forall\ i$

Dus:
$O(\sum_{i=1}^{r(n)} b(i)) = O(b(n) \sum_{i=1}^{r(n)} 1) = b(n).r(n)$

Addendum #1: Named Let

performantiekarakteristiek van een named let zoeken is equivalent aan de analyse van een recursieve procedure.

Addenum #2: Meerdere argumenten

de performantiekarakteristiek van een procedure van meerdere argumenten zal afhangen van elk van die argumenten.

Analyse:

- De meeste algoritmen uit deze cursus zijn "in-place": ze consumeren geen extra geheugen naast het geheugen ingenomen door de invoer.
- Als algoritmen tóch extra geheugen vragen, meten we dat op dezelfde manier als tijd: Θ, Ω en O

Hfst 2.1 Strings in Scheme

donderdag 16 december 2021 10:08

- String: een eindige sequentie van karakters
- Conversion Operations: string->list en list->string (vb. list #\a #\b ...) van O(n) met n lengte van string/list
- Comparison Operations: vergelijken van strings
  - Lexicographic order: korte strings voor de langere, volgerde ASCII code
  - Case sensitive comparison: verschil tussen hoodletter en kleine letter
  - Case insensitive comparison: geen verschil tss hoofdletter & kleine letter

String comparison operation	Functionality
(string=? s1 s2)	String equality
(string-ci=? s1 s2)	String equality, case insensitive
(string<? s1 s2)	String before
(string>? s1 s2)	String after
(string<=? s1 s2)	String before-equal
(string>=? s1 s2)	String after-equal
(string-ci<? s1 s2)	String before, case insensitive
(string-ci>? s1 s2)	String after, case insensitive
(string-ci<=? s1 s2)	String before-equal, case insensitive
(string-ci>=? s1 s2)	String after-equal, case insensitive

- String addition & substraction: samensmelten van 2 string met string-append en substraction mbv een substring: (string index index -> string) - past niet de oorspronkelijke string aan!

De tekst noteren we met t. Het patroon met p. De lengte van het patroon **n-p** of n_p. De lengte van de tekst **n-t** of n_t

De lengte van een string s is het aantal karakters in s. Notatie: |s|

Bij een string s is s_k het k'de karakter. $s_{i \to j}$ is de deelstring van i tot en met j

Hfst 2.2 The Pattern Matching problem

- Geen standaard oplossing voor het probleem

Notatie: s = e.u, $s_{0->k-1}$ (lengte van s is k),
$s_{|s|-k->|s|-1}$ (s heeft suffix van lengte k)

- Prefix: e is een prefix van s als (string=? (append e u) s)
- Suffix: u suffix van s
- Proper Prefix: e niet lege string en e != s dan is e een proper prefix van s
- Proper Suffix: u niet lege string en u!=s dan is u een proper suffix van s

Conclusie probleem:

- Patroonherkenning is **moeilijk en tijdrovend**. Het is één van de dingen waarcomputers zeer slecht in zijn en mensen heel goed in zijn. Het is één van de centrale vraagstukken van de artificiële intelligentie, zowel wat betreft tekst, beeld als geluid.
- Strings aan mekaar **plakken is niet moeilijk**. Strings correct weer **uit mekaar halen is moeilijk en tijdrovend**. Strings dienen bijgevolg **enkel om met gebruikers te communiceren**! Strings vormen de armste datastructuur die er bestaat.

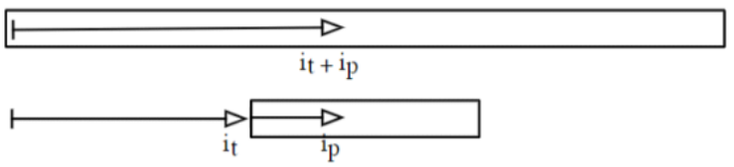
Hfst 2.3 The Brute-Force Algoritme

donderdag 16 december 2021 10:25

Bekijkt alle mogelijk matches, 1 per 1 tot het een match heeft gevonden

(tekst pattern -> #f U number) -> number is de index waar de string in voor komt

```
(define (match t p) ; t is string & p is pattern
      (define n-t (string-length t)) ; n-t = lengte van de string
      (define n-p (string-length p)) ; n-p = lengte van het pattern
      (let loop  ;soort recursie
        ((i-t 0) ; loopt string af
         (i-p 0) ; loopt pattern af
        (cond
          ((> i-p (- n-p 1)) ; i-p > lengte-pattern - 1 => i-t terug geven
           i-t)

          ((> i-t (- n-t n-p)) ; i-t < lengte string - string pattern => false
           #f)

          ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
           (loop i-t (+ i-p 1)))
          ; letter op plaats (i-t + i-p) van string = letter op plaats i-p van pattern
          ; loop i-p+1 maar i-t blijft zelfde

          (else ; miss match
           (loop (+ i-t 1) 0 ))))))
          ; loop i-t+1 en i-p terug op 0
```



Performance: O(t*p) -> kwadratisch

- als p klein is heeft het een lineair gedrag is
- als t en p groot zijn heeft het bruto force een verschrikkelijke performantie

De loop wordt uitgevoerd voor i_t gaande van 0 tot n_t-n_p.
Voor elke i_t gaat i_p in het slechtste geval van 0 tot n_p.

Dus:
$b_{match(nt,np)} \in O(1)$
$r_{match(nt,np)} \in O(n_t.n_p)$

En Dus:
$f_{match(nt,np)} \in O(n_t.n_p)$

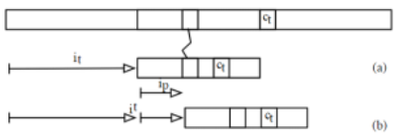
Hfst 2.4 The QuickSearch Algorithm

donderdag 16 december 2021 10:54

- Mismatch -> pattern opnieuw beginnen zoeken op de plaats waar de mismatch zich voordeed
- Nadeel: overshooting wanneer het patroon meerdere keren voorkomt -> kiezen voor het voorkomen het meest rechtse van de tekst

```
(define (compute-shift-function p)
  (define n-p (string-length p)) ;n-p lengte van pattern
  (define min-ascii (char->integer (string-ref p 0)))
  ; min-ascii: 1ste letter van pattern omgezet naar integer
  (define max-ascii min-ascii)

  (define (create-table index)
    (if (< index n-p) ;index < lengte pattern
        (begin
          (set! min-ascii (min min-ascii (char->integer (string-ref p index))))
          ;kleinste waarde van pattern bijhouden
          (set! max-ascii (max max-ascii (char->integer (string-ref p index))))
          ;maximum waarde van pattern bijhouden
          (create-table (+ index 1)))
        ;als alle indexen van pattern zijn overlopen
        ;vector van lengte (max-asci - min-asci + 1) opgevuld met (lengte pattern + 1)
        (make-vector (- max-ascii min-ascii -1) (+ n-p 1))))

  (define (fill-table index)
    (if (< index n-p) ;index < lengte pattern
        ;ascii = asci waarde van de index van het pattern
        (let ((ascii (char->integer (string-ref p index))))
          ;pas vector aan op plaats (ascii - min-ascii) naar (lengte pattern - index)
          (vector-set! shift-table (- ascii min-ascii) (- n-p index))
          (fill-table (+ index 1)))))

  ;shift-table = vector
  (define shift-table (create-table 0))
  (fill-table 0)
  (lambda (c)
    (let ((ascii (char->integer c)))
      ; max-ascii >= ascii >= min-ascii
      (if (>= max-ascii ascii min-ascii)
          ;element op plaats (ascii - min-ascii) halen
          (vector-ref shift-table (- ascii min-ascii))
          ;else lengte pattern + 1
          (+ n-p 1)))))


(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (define shift (compute-shift-function p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ;positie pattern > lengte p - 1 => positie tekst
      ((> i-p (- n-p 1))
       i-t)
      ;positie tekst > positie tekst - pattern => #f
      ((> i-t (- n-t n-p))
       #f)
      ;letter pattern = letter tekst => recursie
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
       (loop i-t (+ i-p 1)))
      (else
       ;c-t = letter op positie rest (+ positie tekst lengte pattern) en lengte tekst
       (let ((c-t (string-ref t (modulo (+ i-t n-p) n-t))))
         ;recursie tekst + rest en positie pattern 0
         (loop (+ i-t (shift c-t)) 0))))))
```

Worst-case:
$f_{match}(n_t,n_p) \in O(n_t.n_p)$



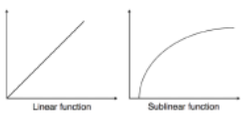
- *In de praktijk lineair en sneller dan al de rest (op Engelse tekst).*
- *Vertoont soms sublineair gedrag:* $O(N + \frac{n_t}{n_p+1})$ $N = \max(\delta_p, n_p)$

Het algoritme is dus niet bruikbaar als je alle characters in de tekst gezien wil hebben

De "spanwijdte" van het "alfabet" van het patroon (c.f. opvullen vector)

Performance

- Worst case zelfde als Brute-Force
- Best case: sublineair door het overslaan van karakters bij mismatch



Hfst 2.5 The Knuth-Morris-Pratt Algoritme

donderdag 16 december 2021 17:11

karakter pattern samenleggen met de eerste mismatch -> overschooting

Skips prefix van het pattern

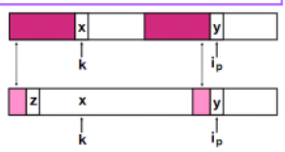
Algoritme:

```
(define (compute-failure-function p)
  (define n-p (string-length p))
  ;vector lengte pattren
  (define sigma-table (make-vector n-p 0))
  (let loop
  ;index pattern starten op pos 2
    ((i-p 2)
     (k 0))
  ;index-pattern < lengte pattern
    (when (< i-p n-p)
      (cond
  ;lettern in pattern = letter in tekst?
        ((eq? (string-ref p k)
              (string-ref p (- i-p 1)))
  ;vector aanpassen op pos index pattern naar k+1
         (vector-set! sigma-table i-p (+ k 1))
  ;recursie alles + 1
         (loop (+ i-p 1) (+ k 1)))
  ; k > 0 => loop met kde element uit vector als nieuwe k
        ((> k 0)
         (loop i-p (vector-ref sigma-table k)))
        (else ; k=0
  ;vector aanpassen op index pattern naar 0 => recursie index pattern + 1
         (vector-set! sigma-table i-p 0)
         (loop (+ i-p 1) k))))
  ;vector aanpassen op pos 0 naar -1
    (vector-set! sigma-table 0 -1)
    (lambda (q)
  ;element uit vector halen op pos q
      (vector-ref sigma-table q))))
```

Opstellen van σ: Vergelijk patroon met zichzelf

*Voor elke i_p zoeken we de k=σ(i_p), t.t.z. het langste **roze** stuk dat zowel prefix als suffix-tot-i_p is:*

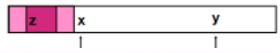
Is x=y? Zoja, schuif k en i_p op en beschouw de volgende x en y om misschien een nóg langer roze stuk te vinden



Zonee, is er dan misschien een korter prefix/suffix-tot-i_p dat y (samen met een zekere z) dan misschien langer maakt?

Maar zo'n prefix/suffix-van-i_p moet noodzakelijkerwijs ook een prefix/suffix-tot-i_p van het roze stuk zélf zijn (zie beide stippellijnen)

Dus zoeken we in dat geval het langste prefix/suffix-tot-k (= reeds berekend toen i_p nog k was). En dat kennen we al, namelijk σ(k). Dus zoeken we verder met σ(k) en i_p



∀ i_p van 2 tot n_p: zoek de langste k

Performance Compute-failure

Beschouw i_p - k:

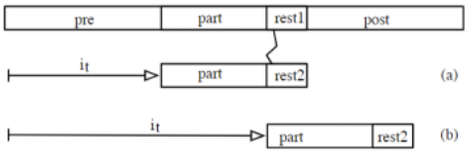
ip ging omhoog

- *in de eerste tak: van i_p en k naar i_p+1 en k+1 dus is i_p - k constant*
- *in de tweede tak: i_p blijft en k naar σ(k) dus is i_p - k verhoogt* (σ(X) < X)
- *in de derde tak: ip naar i_p+1 en k blijft dus is i_p - k verhoogt*

i_p - k ≤ i_p en i_p ≤ n_p en dus wordt de lus maximaal 2n_p keer uitgevoerd

Conclusie: het bepalen van σ is in O(n_p)

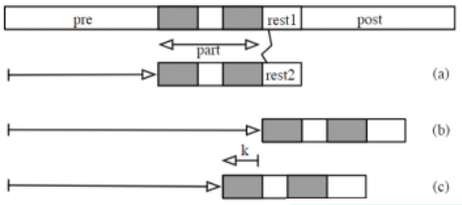
```
(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (define sigma (compute-failure-function p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ;index pattern > lengte pattern -1
      ((> i-p (- n-p 1))
      ;index tekst terug geven -
       i-t)
      ;index tekst > lengte tekst - lengte patroon
      ((> i-t (- n-t n-p))
      ;false teruggeven
       #f)
      ;letter tekst = letter pattern => loop index pattern + 1
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
       (loop i-t (+ i-p 1)))
      (else
      ;loop (+ index tekst (- index pattern compute index pattern))
      ;als index pattern > 0 -> compute index pattern, else 0
       (loop (+ i-t (- i-p (sigma i-p))) (if (> i-p 0)
                                             (sigma i-p)
                                             0))))))
```



Structuur van KMP: stel vóór het matchproces op basis van het patroon een functie σ op zodat k=σ(i_p). Dit heet preprocessing.

σ heet de failure function

Bij herhaling van sequences van het patroon



Bij het matchen verschuift je het patroon ter waarde van i_p - σ(i_p). De eerste σ(i_p) moeten we niet meer checken. Per conventie is σ(0) = -1 zodat we bij een eerste mismatch 1 opschuiven.

Performance match:

De evolutie van i_t + i_p in de loop: een maatstaaf voor de vooruitgang.

- *Ofwel match: i_t + i_p ⇒ i_t + (i_p + 1)*
- *Ofwel geen match: i_t + i_p ⇒ (i_t + i_p - σ(i_p)) + σ(i_p) = i_t + i_p*

σ(X) < X

De som i_t + i_p verhoogt of stagneert (maar dan gaat i_t omhoog) in elke slag van de lus. Dus wordt de lus maximaal 2n_t keer uitgevoerd.

Conclusie: het matchen zélf is in O(n_t)

Performance

- Worst case O(t + p) -> beter dan Brute Force en Quicksearch

Hfst 3.1 Using Scheme's Linear Data Structures

vrijdag 17 december 2021 16:37

"Naked" Vectors

Voordeel

- Fast random access property: Indexen => vector-ref, vector-set! In O (1)

Nadelen

- Fixed lenght: je kan een vector niet langer/korter maken
  - Op voorhand weten hoeveel data je wilt opslaan in vector
  - Wat als vector vol zit? Vector laten groeien => O(n), error, ...
  - Upper bound voor een vector als je nog niet weet hoe lang de vector moet zijn
    - Meer opslag
    - Counter bijhouden van waar het laatst ingevuld element zich bevindt in de vector => error prone

Scheme's Built-in Lists (Naked Scheme List)

Voordeel

- Hoeft niet te weten hoeveel elementen je wilt opslaan

Nadelen

- Alleen elementen vooraan lijst toe te voegen als 1ste element in O(1)
- Geen indexen => geen referentie naar laatste pair
- Operaties op lijsten wat niet met het 1ste element in de lijst te maken hebben => O(n) (vb append, add-to-end,...)
- Parameter Passeer Mechanisme: Aanpassen van parameters met een procedure gaan de cons-cellen buiten de functie niet aanpassen

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

  Vb (define lst '(5 4 3 2 1)) => (add-to-first! lst 4) gaat lst niet aangepast hebben
- Not generic: voorbeeld member?, memq?,... werken niet op alle data-types

Headed Lists and Headed Vectors


```
(define-record-type headed-list
  (make l i)
  headed-list?
  (l scheme-list scheme-list!)
  (i info info!)))
```

De header kan nu info bijhouden => O(1)

bv. Een directe reference naar het laatste element in de list, lengte van list,...

Oplossing voor add-to-first!

```
(define (add-to-front! hl e)
  (scheme-list! hl (cons e (scheme-list hl))))
```

- Headed vectors

  Vb. opslaan van een counter in de header

Lineaire Datastructuur: 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.

Hfst 3.2 Positional Lists

vrijdag 17 december 2021 17:36

Eigenschappen:

- Elementen hebben previous positie en een next positie
- 1ste element enkel een next positie
- Last element enkel een previous positie
- Niet numeriek maar abstract (volgende, vorige)

Positional-list 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> )
```

V = type van de bijgehouden data elementen

P = type van de posities

posities zijn niet noodzakelijk numeriek

```
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 U {#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> )
```

7

Uitleg sommige minder vanzelfsprekende functies

- **From-scheme-list**: maakt een positionele lijst mbv van een scheme-list als input
- **Map**: loopt over de positionele lijst, voort er iets op uit en geeft een nieuwe positionele lijst weer
- **For-each**: loopt door de positionele lijst
- **Update!**: een element op een bepaalde positie veranderen
- **Peek**: geeft een element weer vanin de positionele lijst

ADT implementatie

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

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


(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 (from-scheme-list slst ==?)
  ;slst is een list & ==? is ???
  (define result (new ==?))
  (if (null? slst)
      result
      (let for-all
        ((orig (cdr slst))
         ;orig = rest lijst
         (curr (first (add-after! result (car slst)))))
         ;curr = eerste element in pos lijst
        (cond
          ((not (null? orig)) ;cdr van lijst niet leeg
           (add-after! result (car orig) curr)
           ;achter current plaats je de car van de rest
           (for-all (cdr orig) (next result curr)))
           ;recursie met cdr rest, arg na current in pos-lijst
          (else
           result)))))
```

Hfst 3.2 Positional Lists (2)

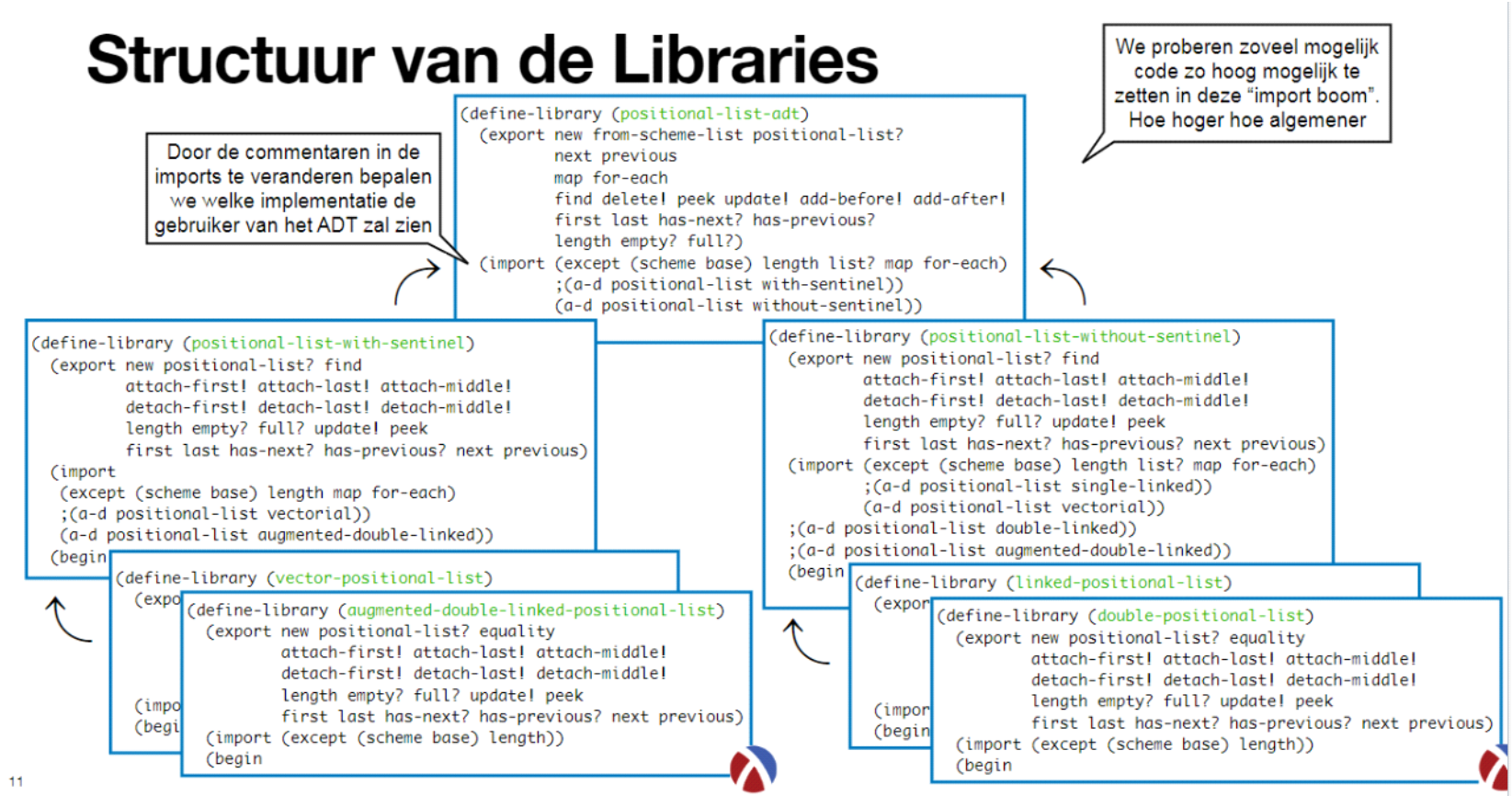
vrijdag 17 december 2021 22:43

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

(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)))

(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)
      (iter-from-head-until plst null?)))

(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)
  ;plst = pos-list & pos = positie
  (if (not (has-next? plst pos))
      (error "list has no next (next)" plst)
      (list-node-next pos)))

(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)))))

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

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

```
(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))))

(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 (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)

(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)

(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))
```

Hfst 3.2 Positional Lists (3)

dinsdag 28 december 2021 13:33



Eerst de operaties die onafhankelijk zijn van de representatie:

- Positional-list-adt
- Positional-list-without-sentinel

Hfst 3.2 Positional Lists (3)

vrijdag 17 december 2021 22:55

Performance

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

The vectorial implementation

- P (positie) heeft een index -> mag niet op vetrouwd worden, geen onderdeel van het ADT (implementie heeft nog steeds "next" en "previous"

```
(define positional-list-size 10) ;representatie

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

(define (new ==?) ;representatie
  (make (make-vector positional-list-size) 0 ==?))

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

(define (storage-move-left vector i j) ;O(n)
  ;idx van i -> i + 1
  (do ((idx i (+ idx 1)))
    ;test: idx > j?
    ((> idx j))
    ;vector aanpassen elemnent 1 plaats naar links verplaatsen
    (vector-set! vector (- idx 1) (vector-ref vector idx))))
```

```
;manipulatie O(n)
(define (attach-first! plst val)
  (attach-middle! plst val -1))

(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)))

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

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

;manipulatie O(n)
(define (detach-last! plst pos)
  (define free (size plst))
  (size! plst (- free 1)))

(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)))
```

```
(define (length plst) ;verificatie
  (size plst))

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

(define (full? plst) ;verificatie
  (= (size plst)
     (vector-length (storage plst))))
```

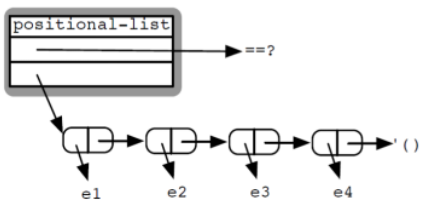
```
(define (peek plst pos) ;manipulatie in O(1)
  (if (> pos (size plst))
      (error "illegal position (peek)" plst)
      (vector-ref (storage plst) pos)))

(define (update! plst pos val);manipulatie in O(1)
  (if (> pos (size plst))
      (error "illegal position (update!)" plst)
      (vector-set! (storage plst) pos val)))

            irst plst) ;navigatie
             (size plst))
(define (first plst) ;navigatie
  (if (= 0 (size plst))
      (error "empty list (first)" plst)
      0))

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

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

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

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

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

Hfst 3.2 Positional Lists (4)

zaterdag 18 december 2021 13:28

The Single Linked Implementation

- Op basis van positional-list ADT
  - positie functie NIET -> nieuwe implementatie
- Header -> nodes: link naar successor ???




```
;representatie
(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!)

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

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


;verificatie O(n)
(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)))
```

```
;O(n)
(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)))))
```

```
;navigatie
;alle functie die gebruik maken van:
;iter-from-head-until -> O(n)
(define (first plst) ;O(1)
  (if (null? (head plst))
      (error "list empty (first)" plst)
      (head plst)))

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

(define (has-next? plst pos) ;O(1)
  (not (null? (list-node-next pos))))

(define (has-previous? plst pos) ;O(1)
  (not (eq? pos (head plst))))

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

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

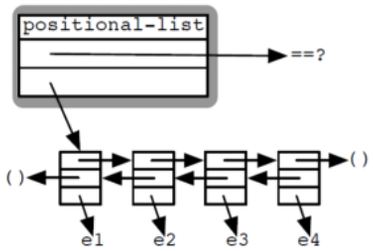
```
;manipulatie O(1)
(define (update! plst pos val)
  (list-node-val! pos val)
  plst)

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

```
;manipulatie
(define (attach-first! plst val) ;O(1)
  (define frst (head plst))
  (define node (make-list-node val frst))
  (head! plst node))

(define (attach-middle! plst val pos) ;O(1)
  (define next (list-node-next pos))
  (define node (make-list-node val next))
  (list-node-next! pos node))

(define (attach-last! plst val) ;O(n)
  (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)))

(define (detach-first! plst) ;O(1)
  (define frst (head plst))
  (define scnd (list-node-next frst))
  (head! plst scnd))

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

Hfst 3.2 Postional Lists (5)

zaterdag 18 december 2021 13:51

A double linked implementation



- Betere performantie dan de vectoriele en single linked implementation
- In node extra back pointer toevoegen (pointer naar previous position)
- Extra info opslaan in de header (vb lenght)
- Verifiactie functies zelfde als bij single linked lists

Double linked

```
;representatie
(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!))

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

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


; navigatie O(n) -> O(1)
(define (previous plst pos)
  (if (not (has-previous? plst pos))
      (error "list has no previous (previous)" plst)
      (list-node-prev pos)))
```

Alle onbrekende functies, zelfde als bij linked lists

Augmented Double linked implementation

- Add-after & last van O(n) -> O(1)
- Record-type list node behouden

Augmented Double linked

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

(define (new ==?)
  (make '() '() 0 ==?))
;verificatie
(define (length plst)
  (size plst))

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

Double linked

```
;manipulatie
(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) ;O(n)
  (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)))

(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) ;O(n) -> O(1)
  (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)
                       '())))
```

Augmented Double linked

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

Hfst 3.2 Positional lists (6)

zaterdag 18 december 2021 14:27

Operation	Vector	Linked	Double Linked 1	Double Linked 2
`new`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`from-scheme-list`	$O(n)$	$O(1)$	$O(n)$	$O(n)$
`length`	$O(1)$	$O(n)$	$O(n)$	$O(1)$
`full?`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`empty?`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`map`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`for-each`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`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`	$O(1)$	$O(n)$	$O(1)$	$O(1)$
`find`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`update!`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`delete!`	$O(n)$	$O(n)$	$O(1)$	$O(1)$
`peek`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`add-before!`	$O(n)$	$O(n)$	$O(1)$	$O(1)$
`add-after!`	$O(n)$	$O(n)$	$O(n)$	$O(1)$

Performantie

Linkes > vectorieel > Double linked > Augmented Double Linked

Geheugen

Een lijst van n datawaarden =

- vectorieel : $\Theta(n)$ geheugencellen
- enkelgelinkt: $\Theta(2n)$ geheugencellen
- dubbelgelinkt: $\Theta(3n)$ geheugencellen

Positionele lijsten: constructiefout in het ADT

Het probleem is dat posities volgens het ADT enkel een relatieve betekenis hebben (t.t.z. next, previous) maar als absolute Scheme waarden toch 'lekken' naar andere datastructuren

Oplossingen:

- Posities worden nooit uit de lijst vrijgegeven: list-with-current
- Posities hebben geen betekenis meer eens uit de lijst vrijgegeven: ranked-list

Hfst 3.3 Variations on Positional Lists

zaterdag 18 december 2021 14:30

The Problem

Posítites zijn relatief geimplemmenteerd (afhankelijk van buren), deze posities worden niet gesyncroniseerd met de absolute posities in de positionele list.

Oplsossingen:

- Ranked lists: verandwoordelijkheid bij gebruiken, poitners naar buren zijn niet permanent (kan elk moment in de tijd veranderd worden) -> men kan de lijst zodanig manipuleren dat de list met 1 bewerking volledig veranderd ????
- List with current: 1 hoofdpositie (current position) kiezen en alle andere posities afhankelijk maken van deze hoofpositie -> internalizing positions

Relative Positions: List with 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 geinvalideerd

```
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)

Realtive Positions: Ranked Lists

```
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> )
```

Hfst 3.4 Searching in Lineair Data Structures

zaterdag 18 december 2021 15:56

Sentinel Searching

- Find in positional-list zijn lineair of sequential zoek algoritmes
- Toevoegen van key om 2de cond "if einde list -> #f" weg te laten
- Toevoegen van de key "sentinal" aan het einde van de list => juist controleren of het element dat we vast hebben om te controleren het laatste element is (key terug verwijderen achter het zoeken ????
- O(n) maar k keer sneller dan "naive" find want maar 1 test uit te voeren , MAAR enkel als attach-last! En detach-last in O(1) zijn!

```
(define (find plst key)
  (if (empty? plst)
      #f
      (let
          ((==? (equality plst)))
        ;key op einde lijst toevoegen
        (attach-last! plst key)
        (let*
            ((pos (let search-sentinel
                    ;curr = element dat we vast hebben
                    ((curr (first plst)))
                    (if (==? (peek plst curr) key)
                        curr
                        ;volgende element vergelijken
                        (search-sentinel (next plst curr)))))
             ;res = #f of positie
             (res (if (has-next? plst pos)
                      pos
                      #f)))
          (detach-last! plst (last plst))
          res))))
```

Sorted Lists

```
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 )
```

Hfst 3.4 Searching in Lineair Data (2)

zaterdag 18 december 2021 17:55

Sorted lists

Implementatie

```
(define default-size 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-size len)) <<? ==?))

(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)))))

(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))

(define (length slst)
  (size slst))

(define (empty? slst)
  (= (length slst) 0))

(define (full? slst)
  (= (length slst)
     (vector-length (storage slst))))
```

```
(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))))
```

```
(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)

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

(define (add! slst val)
  (define <<? (lesser slst))
  (define vect (storage slst))
  (define leng (size slst))
  (if (= leng (vector-length vect))
      (error "list full (add!)" slst))
  (let vector-iter
    ((idx leng))
    (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 (+ leng 1))
  slst)
```

```
;belangerijkste functie voor dit ADT
;O(n) worst case
(define (find-sequential! slst key)
  (define ==? (equality slst))
  (define <<? (lesser slst))
  (define vect (storage slst))
  (define leng (size slst))
  (let sequential-search
    ((curr 0))
    (cond ((>= curr leng)
           (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)
```

Hfst 3.4 Searching in Lineair Data (3)

zaterdag 18 december 2021 18:05

Binary Search

- O(log(n))
- Vector-implementatie

```
(define (find! slst key)
  (define ==? (equality slst))
  (define <<? (lesser slst))
  (define vect (storage slst))
  (define leng (size slst))
  (let binary-search
    ((left 0)
     (right (- leng 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)
```

Methode

- Vector opsplitsen in 2 delen
- Key in welk deel
- Dit deel weer in 2 verdelen
- …

Hfst 3.5 Rings

zaterdag 18 december 2021 18:13

ALLE elementen hebben een previous en next positie

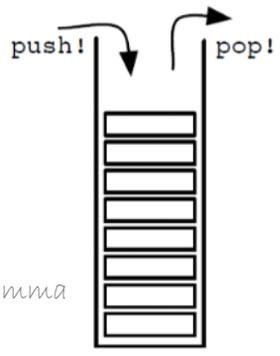
```
ADT ring

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 )
```

```
(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 (add-after! ring val) ;O(1)
  (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)

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

```
(define (add-before! ring val)
  ;O(n) want iter-to-previous nodig
  ;double linked -> O(1)
  (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)

(define (shift-forward! ring) ;O(1)
  (define curr (current ring))
  (if (null? curr)
      (error "empty ring (shift-forward!)" ring))
  (current! ring (ring-node-next curr))
  ring)

(define (shift-backward! ring) ;O(n)
  ;doucle-linked -> O(1)
  (define curr (current ring))
  (if (null? curr)
      (error "empty ring (shift-backward!)" ring)
      (current! ring (iter-to-previous curr)))
  ring)

(define (delete! ring) ;O(n)
  ;double-linked -> O(1)
  (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)

(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)))

(define (length ring) ;O(n)
  (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))))))
```

Hfst 4.1 Lineair Abstract Data Types - Stacks

maandag 20 december 2021 10:59




```
ADT stack

new
   ( ø → stack )
stack?
   ( any → boolean)
push!
   ( stack any → stack )
top
   ( stack → any)
pop!
   ( stack → any )
empty?
   ( stack → boolean )
full?
   ( stack → boolean )
```

LIFO: Last in, First out

Vb'den

- Internet Browsers:
  - Pushed! : clicken op een link
  - Pop! : back knop in browser
- Undo:
  - Pushed! : elke actie uitgevoerd in een programma
  - Pop! : undo-knop
  - 'stack depth' fixed -> n most recent actions
  - Geen echte stack, want length is fixed ???

ADT

- V is the data type of the values stored (any data type)
- top: kijkt in de stack en geeft het meest recente element weer (zonder het de pop!'en)
- Performantie: O(1)
- Vectorial Implementatie: fixed length
- Linked Implementatie: 2 keer zoveel opslag doordat we de pointers moeten opslaan

Vector Implementatie

```
(define stack-size 10)

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

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

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

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

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

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

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

Linked Implementatie

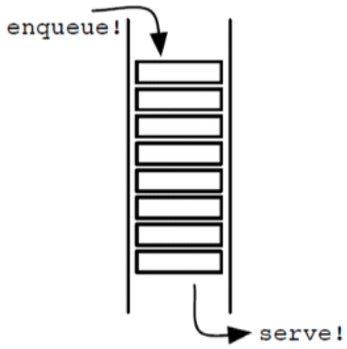
```
(define-record-type stack
  (make l)
  stack?
  (l scheme-list scheme-list!))

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

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

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

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

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

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

Stack-pair ADT

- 2 stacks op hetzelfde moment
- Vector Implementatie -> wasted opslag van vector wordt gebruikt voor 2de stack
- Linker stack grows naar rechts
- Rechter stack grows naar links

Hfst 4.2 Queues

maandag 20 december 2021 11:37



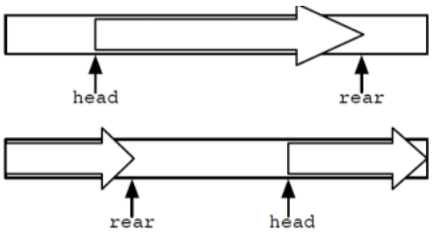
- FIFO: first in, first out
- Enqueue!: voegt een element toe aan de "rear"
- Serve!: haalt het element op de "head" uit de queue

Vb'den

- Print Queues:
  - Enqueue!: nieuw document is naar de printer gestuurd
  - Serve!: 1ste document op de queue wordt geprint
- Outgoing Mail (geen internet):
  - Enqueue!: Nieuwe mail wordt in wachtlijst gezet om te versturen wanneer internet terug is
  - Serve!: Mails worden in volgorde waarin geschreven verstuurd wanneer men terug internet heeft

ADT

- Linked Implementation: Bij elke serve! Worden de elementen dan 1 p opgeschoven naar links in de vector => O(n) (als we de vector omdraaien wordt Enqueue! O(n)
- Circulaire vector: O(1) performantie (ook bij enqueue! of serve!)




```
ADT queue

new
   ( ø → queue )
queue?
   ( any → boolean )
enqueue!
   ( queue any → queue )
peek
   ( queue → any )
serve!
   ( queue → any )
empty?
   ( queue → boolean )
full?
   ( queue → boolean )
```

Linked Implementatie

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

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

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

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

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

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

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

Vector Implementatie

```
(define default-size 5)
(define-record-type queue
  (make s h r)
  queue?
  (s storage)
  (h head head!)
  (r rear rear!))

(define (new)
  (make (make-vector default-size) 0 0))

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

(define (full? q)
  (= (remainder (+ (rear q) 1) default-size)
     (head q)))

(define (enqueue! q val)
  (if (full? q)
      (error "full queue (enqueue!)" q))
  (let ((new-rear (remainder (+ (rear q) 1) default-size)))
    (vector-set! (storage q) (rear q) val)
    (rear! q new-rear))
  q)

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

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

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

Hfst 4.3 Priority Queues

maandag 20 december 2021 16:11

HPFO: Highest priority first out - geen stabiele datastructuur
Priority queue items vormen de opgeslagen "values".
Een priority queue item bestaat uit een data element samen met de erbij horende prioriteit.
Vb'den

- Todo-lists: taken die belangerijker zijn of eerder af moeten worden bovenaan in de todo-list getoond
- Emergencies: mensen met de ergste symptomen worden eerst behandeld

ADT

ADT priority−queue < P >

new
((P P → boolean) → priority−queue < P >)
priority−queue?
(any → boolean)
enqueue!
(priority−queue < P > any P → priority−queue < P >)
peek
(priority−queue < P > → any)
serve!
(priority−queue < P > → any)
full?
(priority−queue < P > → boolean)
empty?
(priority−queue < P > → boolean)

Sorted List Implementatie

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

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

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

(define (empty? pq)
  (slist:empty? (slist pq)))

(define (full? pq)
  (slist:full? (slist pq)))
```

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

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

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

Implementation With Positional Lists

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

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

(define (full? pq)
  (plist:full? (plist pq)))

(define (empty? pq)
  (plist:empty? (plist pq)))

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

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

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

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

Hfst 4.4 Heaps

maandag 20 december 2021 16:40

Heap: Voorgesteld als een Complete Binairy Tree

- Indices: positie van de nodes (links -> rechts, level per level)
- Element in de parent > elementen children
- Hoogte van de node: langste afstand van node naar de leaf node
- Parent: i
- Left Child: $2i$
- Right Child: $2i + 1$

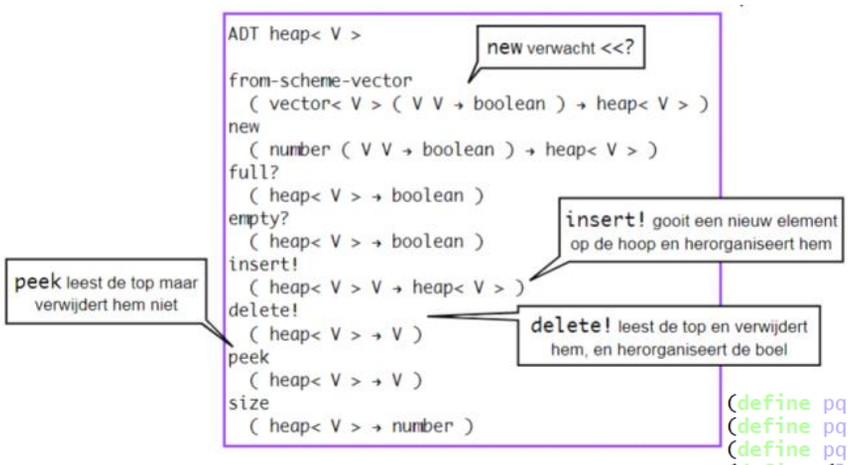
Binary Tree

- Nodes: alle punten in de tree
- Root Node: Is geen kind
- Leaf Node: heeft geen kinderen
- Complete binary Tree: geen gaten wanneer gelezen wordt van link naar rechts, level per level

Properties of Heaps

- Hoogte Heap met n elementen: $h = \lfloor log_2(n) \rfloor$
- #leafs in een heap met n elementen: $\lceil \frac{n}{2} \rceil$
- #knopen op niveau h: $\lceil \frac{n}{2^{h+1}} \rceil$

ADT



Priority Queues en Heaps

Operation	Sorted List	Positional List	Heap
enqueue!	$O(n)$	$O(1)$	$O(log(n))$
serve!	$O(1)$	$O(n)$	$O(log(n))$

Priority Queue en Heap ADT

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

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

(define default-size 50)

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

(define (empty? pq)
  (heap:empty? (heap pq)))

(define (full? pq)
  (heap:full? (heap pq)))

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

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

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

Hfst 4.4 Heaps (2)

maandag 20 december 2021 17:38

Implementatie

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

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

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

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

(define (length heap)
  (size heap))))
```

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

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

```
(define (sift-down heap idx) ;maintaining
  (let
      ((vector-ref
        (lambda (v i)
          (vector-ref v (- i 1))))
       (vector-set!
        (lambda (v i a)
          (vector-set! v (- i 1) a)))
       (vector (storage heap))
       (size (size heap))
       (<<? (lesser heap)))
    (let sift-iter
      ((parent idx)
       (element (vector-ref vector idx)))
      (let*
          ((childL (* 2 parent))
           (childR (+ (* 2 parent) 1))
           (smallest
            (cond
              ((< childL size)
               (if (<<? (vector-ref vector childL)
                        (vector-ref vector childR))
                   (if (<<? element (vector-ref vector childL))
                       parent
                       childL)
                   (if (<<? element (vector-ref vector childR))
                       parent
                       childR)))
              ((= childL size)
               (if (<<? element (vector-ref vector childL))
                   parent
                   childL))
              (else parent))))
        (if (not (= smallest parent))
            (begin (vector-set! vector parent (vector-ref vector smallest))
                   (sift-iter smallest element))
            (vector-set! vector parent element))))))
```

Builiding Heap - Heapify

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

(define (new capacity <<?) ;representatie
  (make (make-vector capacity) 0 <<?))

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

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

Hfst 5.1 Sorting Terminology

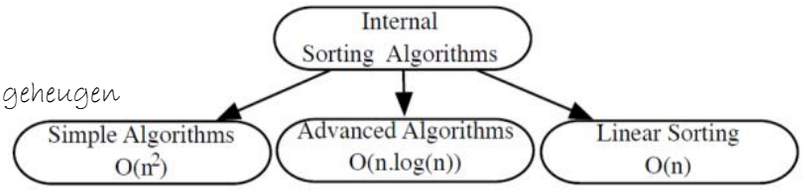
maandag 20 december 2021 18:26

Internal sorting Algoritmes:

- Data past in het centrale geheugen van de computer

External sorting Algoritmes:

- Data past niet in het centrale geheugen
- Data moet opgeslagen worden op extern geheugen



Performatie sorting algoritmes is minimum O(n)

Sort field: key field -> Key voor zoeken = key voor sorteren

Delen Algoritme

- Compare: mbv "ordering" procedure <<?
- Move:
  - Swapping: van i-de positiie naar j-de positie (en omgekeerd)
  - Nieuwe index-vector waar de indexen van vorige vector worden gesorteed (orignele vector blijft hetzelfde)

Soorten sorting algoritmes

- Comparatieve sorteeralgoritmen: ingewikkelde combinaties van "vergelijken van data" en "verplaatsen van data". De sleutelvelden worden vergeleken. Alle velden worden verplaatst. Verplaatsen neemt meestal de vorm van "swap" aan: verwisselen
- Niet-comparatieve sorteeralgoritmen: De velden wordt in detail beschouwd en soms ontleed in onderdelen. Er wordt uitgerekend (bvb op basis van de onderdelen) waar ze moeten staan, of er wordt geturfd hoeveel records er groter of kleiner zijn
- Stable sorting algoritmes: respect the original order of records that have identical keys
  Stabiel als het de relatieve orde van velden met gelijke sleutel bewaart
- In-place sorting algoritmes: no additional memory, only memory that is already occupied by input data -> algoritme moet in O(1)
- Simple algorithms:
  - kwadratisch performantie
  - Bestaat uit inner-loop en outer-loop
  - Gemakkelijk te lezen
- Advanced algorithms:
  - Gecompliceerde algoritmes
  - Performantie: O(n log(n))

Hfst 5.2 Simple Sorting Algorithms

dinsdag 21 december 2021 17:34

Bubble Sort (exchange sort)

- **Slechte performantie**
- Eigenschappen: Stabiel als je strikte ongelijkheid kiest, Makkelijk herschreven voor gelinkte lijsten, In-place
- Einde vector -> begin vector
- Kies index, vergelijk die index voorgaande elementen en swapped deze elementen in de vector (begin vector -> index)
- **Index**: verdeeld vector in een gesorteerd deel en een nog te sorteren deel

```
(define (bubble-sort vector <<?)
  (define (bubble-swap vector idx1 idx2)
    (let ((keep (vector-ref vector idx1)))
      (vector-set! vector idx1 (vector-ref vector idx2))
      (vector-set! vector idx2 keep)
      #t))
  (let outer-loop
    ((unsorted-idx (- (vector-length vector) 2)))
    (if (>= unsorted-idx 0)
        (if (let inner-loop
              ((inner-idx 0)
               (has-changed? #f))
              (if (> inner-idx unsorted-idx)
                  has-changed?
                  (inner-loop (+ inner-idx 1)
                              (if (<<? (vector-ref vector (+ inner-idx 1))
                                       (vector-ref vector inner-idx))
                                  (bubble-swap vector inner-idx (+ inner-idx 1))
                                  has-changed?))))
            (outer-loop (- unsorted-idx 1))))))

(define sort bubble-sort)))
```

∀ unsorted-idx *van n-2 naar 0*

∀ inner-idx *van 0 tot* unsorted-idx

Als elementen op inner-idx en inner-idx+1 fout staan: swap ze

Als er in de inner loop niets geswapt werd is het meteen gedaan!

Performantie: de i'de inner loop doet n-1 werk. Dus $b(i) = n-i$

- worst case - $O(n^2)$: inner loop wordt n keer uitgevoerd en has-changed? wordt telkens #t. Dus $r(n) = n$.

  $\sum_{i=1}^{r(n)} b(i) = \frac{n(n-1)}{2} \in O(n^2)$ n(n-1)/2 compares en swaps
- best case - $O(n)$: inner loop wordt slechts 1 keer uitgevoerd en has-changed? blijft #f. De outer loop stopt dus meteen. Dus $r(n) = 1$.

  $\sum_{i=1}^{r(n)} b(i) = n - 1 \in \Omega(n)$ n-1 compares en 0 swaps

Insertion Sort

vb kaarten sorteren

- Eigenschappen: gemakkelijk herschreven voor gelinkte llijsten, stabiel als strikte ongelijkheid, in-place
- Gesorteerd en ongesorteerd deel vector -> effecient op lijsten die al deels zijn gesorteerd (beste simple sorting techniek)
- 1 per 1 elementen in het gesorteerde deel op zijn plaats steken
- Gelijke elementen "springen" nooit over elkaar in de vector
- Vaak op linked lists (kan nieuwe elementen toevoegen in de gesorteerde lijst)

∀ current op outer-idx van *n-2 naar 0*
Verwissel element current *met het element op de volgende positie:*

∀ inner-idx van outer-idx tot n:

- Als current *kleiner* is dan het element op inner-idx of einde vector bereikt: inner-idx-1 gevonden en stop.
- Anders: schuif element op inner-idx naar inner-idx-1 en doe verder.


```
(define (insertion-sort vector <<?)
  (define (>=? x y) (not (<<? x y)))
  (let outer-loop
    ((outer-idx (- (vector-length vector) 2)))
    (let
        ((current (vector-ref vector outer-idx)))
      (vector-set!
       vector
       (let inner-loop
         ((inner-idx (+ 1 outer-idx)))
         (cond
           ((or (>= inner-idx (vector-length vector))
                (>=? (vector-ref vector inner-idx)
                     current))
            (- inner-idx 1))
           (else
            (vector-set! vector (- inner-idx 1) (vector-ref vector inner-idx))
            (inner-loop (+ inner-idx 1)))))
       current)
      (if (> outer-idx 0)
          (outer-loop (- outer-idx 1))))))

(define sort insertion-sort)))
```

Average Case: n-1 keer met in de i'de keer $\frac{i}{2}$ werk.
Dus $\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4}$ stappen. Dit geeft O(n²).

Hoeveel keer wordt de inner loop uitgevoerd?

Best Case (gesorteerde vector): 1 keer.
Dit geeft Ω(n)

Worst Case(ongesorteerde vector): n-1 keer met de i'de keer i werk. Dus $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$.
Dit geeft O(n²)

Het aantal compares is altijd gelijk aan het aantal moves bij insertion sort

Performantie:

- Best case: $O(n)$
- Worst case: $O(n^2)$

Hfst 5.2 Simple Sorting Algorithms (2)

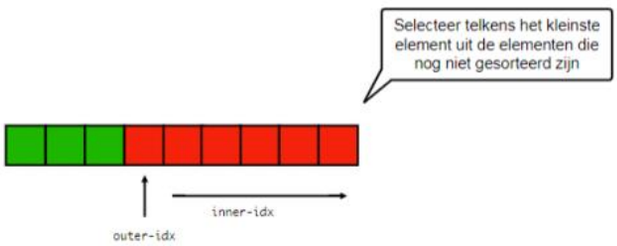
woensdag 22 december 2021 15:27

Selection Sort

- Beter dan insertion sort als er heel veel elementen moeten worden gesorteerd
- Eigenschappen: niet stabiel, in-place, makkelijk herschrijven voor gelinkte lijsten
- Omgekeerde werkwijze dan insertion sort
  - Outer loop: houd de gesorteerde lijst bij en groeit bij elke iteratie
  - Inner loop: neemt het kleinste element uit de ongesorteerde lijst en zet dat elment aan de rear van de gesorteerde lijst

∀ `outer-idx` *van 0 naar n-1.*
Verwissel dit element met het element dat zit op:

> `smallest-id` *is de locatie het kleinste element "tot nu toe".*
> *Voor elke* `inner-idx` *van* `outer-idx+1` *tot n:*
>
> > *Als het element op* `inner-idx` *kleiner is dan het element op* `smallest-idx` *dan wordt* `inner-idx` *de nieuwe waarde van* `smallest-idx`




```
(define (selection-sort vector <<?)
  (define (swap vector i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (let outer-loop
    ((outer-idx 0))
    (swap vector
          outer-idx
          (let inner-loop
            ((inner-idx (+ outer-idx 1))
             (smallest-idx outer-idx))
            (cond
              ((>= inner-idx (vector-length vector))
               smallest-idx)
              ((<<? (vector-ref vector inner-idx)
                    (vector-ref vector smallest-idx))
               (inner-loop (+ inner-idx 1) inner-idx))
              (else
               (inner-loop (+ inner-idx 1) smallest-idx)))))
    (if (< outer-idx (- (vector-length vector) 1))
        (outer-loop (+ outer-idx 1)))))

(define sort selection-sort)))
```

Performantie: big theta (n^2)

De outer loop werkt van 0 naar n-1. De inner loop werkt van outer-idx +1 naar n-1. Dus r(n) = n en b(i) = n-i. Altijd!

$$\sum_{i=1}^{r(n)} b(i) = \sum_{i=1}^{n} (n-i) = (n^2\text{-}n) = O(n^2).$$ *Altijd!*

Maar hiervan zijn er slechts n-1 swaps

	worst-case	best-case
bubble-sort	$O(n^2)$	$\Omega(n)$
insertion-sort	$O(n^2)$	$\Omega(n)$
selection-sort	$\Theta(n^2)$	$\Theta(n^2)$

Hfst 5.3 Advanced Sorting Algorithms

woensdag 22 december 2021 16:02

Quicksort

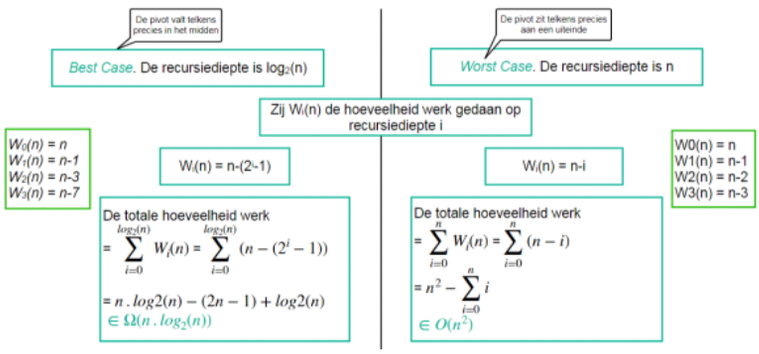
- Meest gebruikt voor general purpose
- Eigenschappen: niet stabiel, goed op dubbel gelinkte lijsten (niet op andere soorten), niet in place min log(n) geheugen
- Relatief gemakkelijk program voor een best-case performantie van O(nlog(n))
- Recursief
- Kan ontaarden: pivot komt toevallig bij een uiteinde -> degenereert
- Divide and conquer algoritme: 3 fases
  - **Divide**: probleem p is onderverdeeld in 2 kleinere dezelfde problemen p_1,p_2 (subproblemen)
  - **Conquer**: algoritme recursief toepassen op de subproblemen, elk subprobleem wordt dan opgedeeld in andere subproblemen s_1,....s_k
  - **Combine**: s_1,....s_k wordt gecombined tot de oplossing S voor P

Algoritme

- **Input**: vector lengte n
- Pivot: gekozen element uit de vector (first phase), zo wordt de vector verdeeld in 2 delen - Divide
- Partitioning phase (secondary phase): alle elementen kleiner dan de pivot worden links ervan geplaats, alle groter rechts, algoritme terug toepassen op linker- en rechterdeel van de pivot - Conquer
- Third phase: zelfde algoritme word recursief toegepast : en wordt zo de oplossing opgesteld - combine

Om de elementen van l tot r te sorteren	
*Kies het element op plaats l (= de **pivot**)*	
*Ga van l naar r en tegelijk van r naar l. Vergelijk ieder element met de pivot. Gooi grotere elementen naar rechts en kleinere naar links. Stop de pivot in het **midden** m.*	
Quicksort van l tot m-1	Quicksort van m+1 tot r

```
(define (quicksort vector <<?)
  (define (swap i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (define (shift-to-right i x)
    (if (<<? (vector-ref vector i) x)
        (shift-to-right (+ i 1) x)
        i))
  (define (shift-to-left j x)
    (if (<<? x (vector-ref vector j))
        (shift-to-left (- j 1) x)
        j))
  (define (partition pivot i j)
    (let ((shifted-i (shift-to-right i pivot))
          (shifted-j (shift-to-left j pivot)))
      (cond ((< shifted-i shifted-j)
             (swap shifted-i shifted-j)
             (partition pivot shifted-i (- shifted-j 1)))
            (else
             shifted-j))))
  (define (quicksort-main l r)
    (if (< l r)
        (begin
          (if (<<? (vector-ref vector r)
                   (vector-ref vector l))
              (swap l r))
          (let ((m (partition (vector-ref vector l) (
            (swap l m)
            (quicksort-main l (- m 1))
            (quicksort-main (+ m 1) r)))))
  (quicksort-main 0 (- (vector-length vector) 1)))
```



Performantie:

- Best case: O(nlog(n))
- Worst case: $O(n^2)$
- Gemiddeld: Een goede na een slechte pivot
  Levert 3 delen:
  - 2 delen: lengte (n-1)/2
  - 1 deel lengte 1

  => recursieboom die dubbel zo diep is als best-case (onafhankelijk van n)
  Praktijk: 80% goed en 20% slecht => 1,5 keer dieper

Nadelen

- Input vector al gesorteerd (ook als vector omgekeerd gesorteerd is) -> $O(n^2)$ want pivot op plaats 0 (pivot op plaats n-1)
- Beste performantie wanneer pivot in het midden van de vector ligt

Hfst 5.3 Advanced Sorting Algorithms

woensdag 22 december 2021 16:57

Randomized Quicksort:

- Elementen input vector randomizen zodat de kans op worst-case wordt verminderd

```
(define (quicksort vector <<?)
  (define (swap i j)
    (let ((temp (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j temp)))
  (define (shift-to-right i x)
    (if (<<? (vector-ref vector i) x)
        (shift-to-right (+ i 1) x)
        i))
  (define (shift-to-left j x)
    (if (<<? x (vector-ref vector j))
        (shift-to-left (- j 1) x)
        j))
  (define (partition pivot i j)
    (let ((shifted-i (shift-to-right i pivot))
          (shifted-j (shift-to-left j pivot)))
      (cond ((< shifted-i shifted-j)
             (swap shifted-i shifted-j)
             (partition pivot shifted-i (- shifted-j 1)))
            (else
             shifted-j))))
  (define (randomized-partition l r)
    (swap l (random-inbetween l r))
    (if (<<? (vector-ref vector r)
             (vector-ref vector l))
        (swap l r))
    (partition (vector-ref vector l) (+ l 1) (- r 1)))
  (define (quicksort-main vector l r)
    (if (< l r)
        (let ((m (randomized-partition l r)))
          (swap l m)
          (quicksort-main vector l (- m 1))
          (quicksort-main vector (+ m 1) r))))
  (quicksort-main vector 0 (- (vector-length vector) 1)))
```

Median of Three Quicksort

- Kans worst case nog meer verkleinen dan Randomize Quicksort
- Meer gecontroleerd de pivot kiezen
- Mediaan te nemen van het linkste en rechtse element zodat de pivot in het midden ligt

```
(define (quicksort vector <<?)
  (define (swap i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (define (shift-to-right i x)
    (if (<<? (vector-ref vector i) x)
        (shift-to-right (+ i 1) x)
        i))
  (define (shift-to-left j x)
    (if (<<? x (vector-ref vector j))
        (shift-to-left (- j 1) x)
        j))
  (define (partition pivot i j)
    (let ((shifted-i (shift-to-right i pivot))
          (shifted-j (shift-to-left j pivot)))
      (cond ((< shifted-i shifted-j)
             (swap shifted-i shifted-j)
             (partition pivot shifted-i (- shifted-j 1)))
            (else
             shifted-j))))
```

```
(define (m3-partition l r)
  (let ((middle (quotient (+ l r) 2)))
    (swap middle (+ l 1))
    (if(<<? (vector-ref vector l)
            (vector-ref vector (+ l 1)))
       (swap l (+ l 1)))
    (if (<<? (vector-ref vector r)
             (vector-ref vector (+ l 1)))
        (swap r (+ l 1)))
    (if (<<? (vector-ref vector r)
             (vector-ref vector l))
        (swap l r))
    (partition (vector-ref vector l) (+ l 1) (- r 1))))
(define (quicksort-main vector l r)
  (if (< l r)
      (let ((m (m3-partition l r)))
        (swap l m)
        (quicksort-main vector l (- m 1))
        (quicksort-main vector (+ m 1) r))))
(quicksort-main vector 0 (- (vector-length vector) 1)))
```

Beschouw l, middle en r.
Het kleinste gaat in l+1, mediaan in l, het grootste in r en de overschot in middle.

Hfst 5.3 Advances Sorting Algorithms

woensdag 22 december 2021 17:06

De "computationele overhead" van Quicksort is zeer groot. Recursie, bindingen, procedure oproepen, etc.

A third improvement

- Nog beter dan de median of three Quicksort
- Als lengte (sub)vector kleiner is dan een constante (meestal 10) -> overschakelen naar een simpler sorting algoritme (vb insertion sort)

```
(define (quicksort vector amount <<?)
  (define (swap i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (define (shift-to-right i x)
    (if (<<? (vector-ref vector i) x)
        (shift-to-right (+ i 1) x)
        i))
  (define (shift-to-left j x)
    (if (<<? x (vector-ref vector j))
        (shift-to-left (- j 1) x)
        j))
  (define (partition pivot i j)
    (let ((shifted-i (shift-to-right i pivot))
          (shifted-j (shift-to-left j pivot)))
      (cond ((< shifted-i shifted-j)
             (swap shifted-i shifted-j)
             (partition pivot shifted-i (- shifted-j 1)))
            (else
             shifted-j))))
```

```
(define (m3-partition l r)
  (let ((middle (quotient (+ l r) 2)))
    (swap middle (+ l 1))
    (if(<<? (vector-ref vector l)
            (vector-ref vector (+ l 1)))
       (swap l (+ l 1)))
    (if (<<? (vector-ref vector r)
             (vector-ref vector (+ l 1)))
        (swap r (+ l 1)))
    (if (<<? (vector-ref vector r)
             (vector-ref vector l))
        (swap l r))
    (partition (vector-ref vector l) (+ l 1) (- r 1))))
(define (quicksort-main vector l r)
  (if (< l r)
      (let ((m (m3-partition l r)))
        (swap l m)
        (quicksort-main vector l (- m 1))
        (quicksort-main vector (+ m 1) r))))
(quicksort-main vector 0 (- amount 1)))
```

Mergesort

- Nadeel: not in-place en moeilijk om naar in-place te maken
- Voordeel: stabiel algoritme

$W_i(n)$ = de hoeveelheid werk op backtrackniveau i

$i \in [1 .. \log_2(n)]$

Algoritme

$W_i(n) = 2^{\log(n)-i} M_i(n)$

- Regions: Vector in 2 helften verdelen, recursief de functie op de 2 helften oproepen (Calling Phase). Deze helften zijn regions
- Eindconditie recursie: length regions = 1
- Na eindconditie -> backtracking en het sorteerwerk (mergen van regions)

Dus is de totale hoeveelheid werk

$$= \sum_{i=1}^{\log_2(n)} W_i(n)$$

$M_i(n)$ = Hoeveelheid werk om 2 "regio's" van lengte 2^{i-1} te mergen naar 1 van lengte 2^i

```
(define (merge vector p q r)
  (let ((working-vector (make-vector (+ (- r p) 1))))
    (define (copy-back a b)
      (vector-set! vector b (vector-ref working-vector a))
      (if (< a (- (vector-length working-vector) 1))
          (copy-back (+ a 1) (+ b 1))))
    (define (flush-remaining k i until)
      (vector-set! working-vector k (vector-ref vector i))
      (if (< i until)
          (flush-remaining (+ k 1) (+ i 1) until)
          (copy-back 0 p)))
    (define (merge-iter k i j)
      (cond ((and (<= i q) (<= j r))
             (let ((low1 (vector-ref vector i))
                   (low2 (vector-ref vector j)))
               (if (<<? low1 low2)
                   (begin
                     (vector-set! working-vector k low1)
                     (merge-iter (+ k 1) (+ i 1) j))
                   (begin
                     (vector-set! working-vector k low2)
                     (merge-iter (+ k 1) i (+ j 1))))))
            ((<= i q)
             (flush-remaining k i q))
            (else
             (flush-remaining k j r))))
    (merge-iter 0 p (+ q 1))))
(define (merge-sort-rec vector p r)
  (if (< p r)
      (let ((q (quotient (+ r p) 2)))
        (merge-sort-rec vector p q)
        (merge-sort-rec vector (+ q 1) r)
        (merge vector p q r))))
(merge-sort-rec vector 0 (- (vector-length vector) 1))
vector)
```

Mergen van 2 regio's

Tussen 2^{i-1} en 2^i-1 compares

Elk element moet moven: dus 2^{i+1} moves

Dus het aantal stappen van $M_i(n)$

$\leq 2^{i+1} + 2^i - 1$

$\leq 2^{i+1} + 2^i$

$\leq 2^{i+1} + 2^{i+1}$

$= 2^{i+2}$

Dus is de totale hoeveelheid werk

$$= \sum_{i=1}^{\log_2(n)} W_i(n)$$

$$= \sum_{i=1}^{\log_2(n)} 2^{\log_2(n)-i} M_i(n)$$

$$\leq \sum_{i=1}^{\log_2(n)} 2^{\log_2(n)-i} 2^{i+2}$$

$\in O(n \,.\, \log_2(n))$

Heapsort

- Vector omzetten naar heap, en dan kleinste element telkens verwijderen uit de heap tot als de heap leeg is
- Niet stabiel

Gooi alle elementen op een heap

↓

Pas n keer `delete!` toe op de heap en vul zo de vector opnieuw Pas n keer `delete!` toe op de heap en vul zo de vector opnieuw

```
(define (heapsort vector <<?)
  (define >>? (lambda (x y) (not (<<? x y))))
  (define heap (from-scheme-vector vector >>?))
  (define (extract idx)
    (vector-set! vector idx (delete! heap))
    (if (> idx 0)
        (extract (- idx 1))))
  (extract (- (vector-length vector) 1)))
```

Performance: Heapify $O(n)$ + extract n keer delete $O(\log(n))$ => $O(n) + O(n\log(n)) = O(.\log(n))$

Hfst 5.4 Limitations of Comparative Sorting

woensdag 22 december 2021 18:29

Kan niet sneller dan $\Omega(n.\log(n))$ want:

- Sequence: $s_0, s_1, \ldots s_{n-1}$ (n data elementen die moeten gesorteerd worden)
- Sequence -> boom, met elke node een comparison van 2 elementen
- Pad in de boom vormt de gesorteerde oplossing
- De boom toont alle mogelijke manieren voor uitvoeren van het sorting algorithm => n! permutaties => n! leaves
- Hoogte boom: > log(n!) => log(n!) $\in \Omega(n.\log(n)) \Rightarrow h > n.\log(n) \Rightarrow$ performantie: $\Omega(n.\log(n))$

Hfst 5.5 Comparing Comparative Algorithms

- **Bubble sort**: traagste algoritme, enkel handig als **data al gesorteerd is**
- **Selection sort**: trager dan **insertion sort** maar beter als er **niet veel data moet gesorteerd worden**, in alle andere gevallen is **insertion sort** beter
- **Quicksort**: niet effecient voor vectors kleiner dan 10 (computational overhead), als data < 10 => **insertion sort**, niet veel data gesorteerd moet worden => **selection sort**
- **Quicksort & Heapsort**: goede **algeme sorteer algoritmes** in O(nlog(n))
  - Voordeel **Heapsort: in-place** (Quicksort niet in-place door recursie)
  - **Veel data te sorteren is heapsort** beter dan quicksort want quicksort spendeerd veel tijd om data-elementen rond the pivot te bewegen
- Mergesort:
  - Goed om **linked lists** te sorteren omdat mergesort telkens het linkse element neemt
  - Kan gebruikt worden als **external sorting algorithm** (want goede performantie op heel veel data) want geen O(1) access op vectoren nodig
  - 40% **minder elementen te vergelijken** dan bij Quicksort
  - **Minder moves** dan quicksort
  - Merge-sort **niet in-place** (en moeilijk om in-place te maken)

Hfst 5.6 Sorting in Lineair time

woensdag 22 december 2021 19:59

Assumptions about the structure key -> lineair behaviour

Radix Sort

- Vb. Punch Card (Colruyt) -> wordt gesorteerd door in bakken te gooien, de locatie van een gat in de kaart bepaald in welke bak de kaart terecht komt, deze volgorde behouden en kijken naar positie volgende gat, ….
- Starten bij de minst significate digit (vb 54321 -> starten bij 1)
- It is applicable to any kind of keys
  that consist of **k symbols $s_k s_{k-1} \ldots s_1$** and in which each symbol has **N** possible values. N is called the **radix** of the keys.
- Lexicographical ordering:
  - 2 even lange keys (K & K') van lengte **i**
  - $K = d_i d_{i-1} \ldots d_1$ & $K' = d'_i d'_{i-1} \ldots d'_1$ met K de key, en d_i een element uit K
  - $d_i \leq d'_i$ & $d_{i-1} \ldots d_1 \leq d'_{i-1} \ldots d'_1 \Rightarrow K \leq K'$

```
(define (radix-sort slst key key-size)
  (define sort-bins (make-vector 10 '()))
  (define (spread lst digit-k)
    (define (digit item)
      (remainder (quotient item (expt 10 digit-k)) 10))
    (define (spread-iter lst)
      (let ((idx (digit (key (car lst)))))
        (vector-set! sort-bins
                     idx
                     (cons (car lst)
                           (vector-ref sort-bins idx)))
        (if (not (null? (cdr lst)))
            (spread-iter (cdr lst)))))
    (spread-iter lst))
  (define (collect)
    (define (collect-iter index acc)
      (define (collect-list lst acc)
        (if (null? lst)
            (if (> index 0)
                (collect-iter (- index 1) acc)
                acc)
            (collect-list (cdr lst) (cons (car lst) acc))))
      (let ((l-index (vector-ref sort-bins index)))
        (vector-set! sort-bins index '())
        (collect-list l-index acc)))
    (collect-iter 9 '()))
  (define (radix-iter digit-k slst)
    (spread slst digit-k)
    (if (= digit-k key-size)
        (collect)
        (radix-iter (+ 1 digit-k) (collect))))
  (radix-iter 0 slst))
```

Performantie

- Radix-iter k keer herhaald
- Elke iteratie spread en collect => O(n)
- n data elementen met k digits => O(k.n) (k verwaarloosbaar) => O(n)
- Stabiel
- Grote n => snelste sorteer algoritme maar enkel toepasbaar bij specifieke gevallen
  (vb niet toepasbaar op komma-getallen want plaats van de komma is essentieel)
- Radix moet op voorhand gekend zijn
- Aantal mogelijke bins per digit moet klein genoeg zijn (vb 10 cijfers, 26 letters)
- Not in-place: bins nodig en lijsten
- Best op linked lists: linked lists per bin en dan terug aan elkaar zetten (mogelijk (her) gebruik van cdr-pointers, hier nog niet toegepast)

Hfst 5.6 Sorting in Lineair time (2)

woensdag 22 december 2021 21:47

Bucket Sort

- Buckets: bins of radix sort
- Bucket vector: linked-list in de buckets
- 1-fase voor de k-fases in radix sort

Performance: 1 pass van de Radix sort => O(n)

Wanneer handig om te gebruiken?

- **Imperfect sort**: ordening in de buckets is onbelangerijk
- **Buckets zijn klein** genoeg om een ander sorteer algoritme erop toe te passen

Counting Sort

- Alle **keys zijn integers** in een domein van 0 tot max-key
- telt hoeveel keer iedere key voorkomt en stopt de getallen in een vector entry per key. Dan worden de accumulatieve sommen berekend van die getallen die dan nadien per key als index dienen.
- Methode: Neem data element met key k => het #elementen tellen waarvan k' < k => #elementen is de positie voor dit data element
- Performantie: O(n + max-key); max-key te groot => counting sort ineffecient
- **Nadeel**: keys integers moeten zijn
- Not in-place maar wel stabiel

```
(define (counting-sort in out max-key key)
  (let ((count (make-vector max-key 0))
        (size (vector-length in)))
    (define (fill-count-vector i)
      (let ((k (key (vector-ref in i))))
        (vector-set! count
                     k (+ (vector-ref count k) 1))
        (if (< (+ i 1) size)
            (fill-count-vector (+ i 1)))))
    (define (sum-vector i)
      (vector-set! count
                   i
                   (+ (vector-ref count (- i 1))
                      (vector-ref count i)))
      (if (< (+ i 1) max-key)
          (sum-vector (+ i 1))
          count))
    (define (spread-out-again i)
      (let* ((data (vector-ref in i))
             (k (- (vector-ref count (key data)) 1)))
        (vector-set! out k data)
        (vector-set! count (key data) k)
        (if (<= i 0)
            out
            (spread-out-again (- i 1)))))
    (fill-count-vector 0)
    (sum-vector 1)
    (spread-out-again (- size 1))))
```

3 fases:

- **Fase 1**
  - Alle keys in de input-vector worden 1 per 1 in fill-count-vector gestoken
  - Elk voorkomen van key k => counter in (vector-ref count k) + 1
  - => count-vector bevat het aantal voorkomen van elke key
- **Fase 2**
  - Voor index i worden alle element op index ≤ i in count opgeteld met sum-vector, dit geeft dan het aantal voorkomens weer van de elementen kleiner of = i
- **Fase 3**
  - In elke iteratie: k gelezen in count om de index van de uitkomst vector te weten
  - Waarde van index (vector-ref in k) in count is de index voor de oplossing vector oorspronkelijk in (vector-ref in k) ????
  - Niet alle elementen zijn distinct => waarde op (vector-ref in k) in count verminderen
  - Spread-out-again zet de elementen in de oplossing vector

Performance: Tellen O(n) + Sommen berekenen O(k) + Spreiden O(n) = O(n + k)

Hfst 6.1 The structure of Trees

donderdag 23 december 2021 20:59

Structuur: Hiërarchisch

- Nodes: alle elementen in de tree
- Internal Nodes: nodes die children hebben

Vb. Folders & menu's (in programma's)

Terminologie

Tree: collectie van data in nodes

- Empty tree: geen nodes
- Non-empty tree: unieke root node
- Parent node: direct reference naar children nodes
- Leaf nodes: geen children
- k-ary nodes: k = #children van de nodes (De ariteit van een node is het aantal kinderen)
- k-ary tree: alle parents hebben max k children
- Subtree: elk kind n_i bestaat er een tree T_i met n_i als root node (T_i heeft min 1 node)
- Descendant of n: als n een node en n_i een direct children is van n of een descentand van de children van n
- Ancestors: alle descendats van n
- Siblings: hebben dezelfde parent node
- Height tree: lengte van langste pad in de tree naar de leaf-node
- Treaded trees: pointer dat zou wijzen naar een "empty" child (wasted pointer) wordt gebruikt om naar de volgende node te wijze, deze pointer is een tread
- Heigth complete k-ary tree: $\lfloor log_k(n) \rfloor$

Binary Trees

```
ADT binary-tree

null-tree
  binary-tree
new
  ( any binary-tree binary-tree → binary-tree )
null-tree?
  ( binary-tree → boolean )
left
  ( binary-tree → binary-tree )
left!
  ( binary-tree binary-tree → binary-tree )
right
  ( binary-tree → binary-tree )
right!
  ( binary-tree binary-tree → binary-tree )
value
  ( binary-tree → any )
value!
  ( binary-tree any → binary-tree )
```

A linked implementation

- Elke node is gerepresenteerd door een record dat de waarde van de node opslaat en de pointer naar de linker en rechter sub-tree
- Linked: pointers naar children
- Zoeken naar parent node in O(n)

```
(define-record-type tree
  (new v l r)
  tree?
  (v value value!)
  (l left left!)
  (r right right!))

(define null-tree '())

(define (null-tree? node)
  (eq? node null-tree))))
```

Vector Representatie -> nadeel: the max capaciteit van de tree moet op voorhand gekend zijn

Alternatieve linked implementatie: double linked

- Back pointer naar parent node
- Zoeken naar parent node in O(1)
- Nadeel: extra pointer per node op te slaan

Hfst 6.2 Tree Traversals

donderdag 23 december 2021 11:08

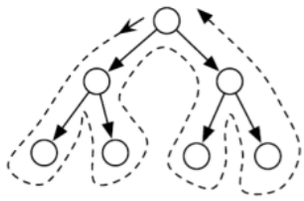


Figure 6.9: Depth First Traversal

Depth-First Traversal (on binary trees)

- children of a node priority over the siblings of the node
- Descends from root to leaves
- Subtrees of a node are traversed -> siblings are considerd

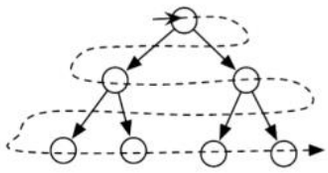
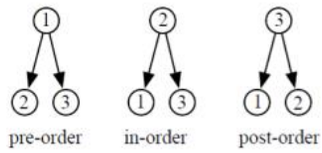


Figure 6.10: Breadth First Traversal



Recursive Traversal Implementation

```
(define (in-order tree proc)
(define (do-traverse current)
  (when (not (null-tree? current))
    (do-traverse (left current))
    (proc (value current))
    (do-traverse (right current))))
  (do-traverse tree))

(define (pre-order tree proc)
  (define (do-traverse current)
    (when (not (null-tree? current))
      (proc (value current))
      (do-traverse (left current))
      (do-traverse (right current)))
  (do-traverse tree))

(define (post-order tree proc)
  (define (do-traverse current)
    (when (not (null-tree? current))
      (do-traverse (left current))
      (do-traverse (right current))
      (proc (value current))))
  (do-traverse tree))
```

Iterative Traversal Implementation

- Runtime stack: Requires stack to store nodes found along the path during the descend -> reached the leaf -> backtracking: popping the topmost element from the stack
- Pre-order traversal

```
(define (iterative-pre-order tree proc)
  (define stack (stack:new))
  (define (loop)
    (if (not (stack:empty? stack))
        (let ((node (stack:pop! stack)))
          (proc (value node))
          (if (not (null-tree? (right node)))
              (stack:push! stack (right node)))
          (if (not (null-tree? (left node)))
              (stack:push! stack (left node)))
          (loop))))
  (stack:push! stack tree)
  (loop))
```

- In-order traversal

```
(define (iterative-in-order tree proc)
  (define stack (stack:new))
  (define (loop-up)
    (let ((node (stack:pop! stack)))
      (proc (value node))
      (if (not (null-tree? (right node)))
          (begin (stack:push! stack (right node))
                 (loop-down))
          (if (not (stack:empty? stack))
              (loop-up)))))
  (define (loop-down)
    (let ((node (stack:top stack)))
      (if (not (null-tree? (left node)))
          (begin (stack:push! stack (left node))
                 (loop-down))
          (loop-up))))
  (stack:push! stack tree)
  (loop-down))
```

Hfst 6.2 Tree Traversals (2)

vrijdag 24 december 2021 13:46

- Post-order traversal
  - Complexer: moeten telkens weten of het de 1ste/2de of 3de keer is dat men zich bevindt in deze node

```
(define (iterative-post-order tree proc)
  (define stack (stack:new))
  (define (loop-up-right)
    (let ((node (stack:pop! stack)))
      (proc (value node))
      (cond ((and (not (stack:empty? stack))
                  (eq? (right (stack:top stack)) node))
             (loop-up-right))
            ((not (stack:empty? stack))
             (loop-up-left)))))
  (define (loop-up-left)
    (let ((node (stack:pop! stack)))
      (cond ((not (null-tree? (right node)))
             (stack:push! stack node)
             (stack:push! stack (right node))
             (loop-down))
            ((and (not (stack:empty? stack))
                  (eq? (right (stack:top stack)) node))
             (proc (value node))
             (loop-up-right))
            ((not (stack:empty? stack))
             (proc (value node))
             (loop-up-left)))))
  (define (loop-down)
    (if (not (stack:empty? stack))
        (let ((node (stack:top stack)))
          (if (null-tree? (left node))
              (loop-up-left)
              (begin
                (stack:push! stack (left node))
                (loop-down))))))
  (stack:push! stack tree)
  (loop-down))
```

Breath-first traversals

- Siblings of a node priority over the children
- Queue: children are added to the end of the queue -> sibling in front of queue then children
- Iteratieve versie (most easy to implement vs recursive)

```
(define (breadth-first tree proc)
  (define q (queue:new))
  (define (loop)
    (let ((node (queue:serve! q)))
      (proc (value node))
      (if (not (null-tree? (left node)))
          (queue:enqueue! q (left node)))
      (if (not (null-tree? (right node)))
          (queue:enqueue! q (right node)))
      (if (not (queue:empty? q))
          (loop))))
  (queue:enqueue! q tree)
  (loop))
```

Vb game tree:

- Nodes: situations in a game played by a player and the computer
- Leaves: game situations in which either the player or the computer has won
- Combinatorial explosion with millions of tree nodes -> too large for computer memory
- Not generate complete tree but only few levels for every move

Hfst 6.3 Binary Search Trees

vrijdag 24 december 2021 13:58

A List-based Implementation of Dictionaries

- As sorted lists: more effecient to use "find"

```
;store key value pairs
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)
;lift take a procedure that works on keys
;output procedure that works on associations
(define (lift proc)
  (lambda (assoc1 assoc2)
    (proc (assoc-key assoc1)
          (assoc-key assoc2))))

(define (new ==? <<?)
  (slist:new
   (lift <<?)
   (lift ==?)))

(define (insert! dct key val)
  (slist:add! dct (make-assoc key val)))

(define (delete! dct key) ;O(n)
  (slist:find! dct (make-assoc key 'ignored))
  (if (slist:has-current? dct)
      (slist:delete! dct)))


(define (find dct key) ;O(log(n) -> binairy chearch
  (slist:find! dct (make-assoc key 'ignored))
  (if (slist:has-current? dct)
      (assoc-value (slist:peek dct))
      #f))
```

- Vectorial implementation:
  - size know upfront
  - Binary search -> O(log(n))
- Linked version of sorted list:
  - doesn't allow binary search -> find O(n)

Binary Search Trees (BST)

- BST voorwaarde: Een binaire zoekboom is een binaire boom zodat voor elke node n geldt: de elementen in de linkerdeelboom van n zijn allen kleiner dan n en de elementen in de rechterdeelboom van n zijn allen groter dan n.
- If n has left child
  -> all data element of left child are <n
- If n has right child
  -> all data elements of right child are >n
- If key = root node -> key has found
- If key > root -> Searching right of the tr
- If key < root -> searching left of the tree

```
ADT BST<V>

new
   ( ( V V → boolean) ( V V → boolean) → BST<V> )
tree?
   ( any → boolean )
find
   ( BST<V> V → V ∪ { #f } )
insert!
   ( BST<V> V → BST<V> )
delete!
   ( BST<V> V → BST<V> )
```

```
(define (find bst key)
  (define <<? (lesser bst))
  (define ==? (equality bst))
  (let find-key
    ((node (root bst)))
    (if (tree:null-tree? node)
        #f
        (let
            ((node-value (tree:value node)))
          (cond
            ((==? node-value key)
             node-value)
            ((<<? node-value key)
             (find-key (tree:right node)))
            ((<<? key node-value)
             (find-key (tree:left node))))))))
```

Performantie find:

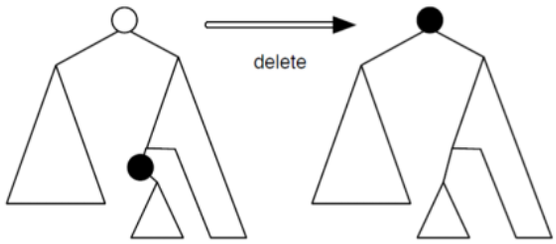
- best case O(log(n))
- Worst case O(n)

```
(define-record-type bst
  (make r e l)
  bst?
  (r root root!)
  (e equality)
  (l lesser))

(define (new ==? <<?)
  (make tree:null-tree ==? <<?))
```

Hfst 6.3 Binary Search Trees (2)

vrijdag 24 december 2021 15:29

```
(define (insert! bst val)
  (define <<? (lesser bst))
  (let insert-iter
    ((parent tree:null-tree)
     (child! (lambda (ignore child) (root! bst child)))
     (child (root bst)))
    (cond
      ((tree:null-tree? child)
       (child! parent
               (tree:new val
                         tree:null-tree
                         tree:null-tree)))
      ((<<? (tree:value child) val)
       (insert-iter child tree:right!
                    (tree:right child)))
      ((<<? val (tree:value child))
       (insert-iter child tree:left!
                    (tree:left child)))
      (else
       (tree:value! child val)))))
```



- Leaf node deleted: use child! To make the parent refer to the empty tree
- Node with 1 subtree: parent to node refer to subtree
- Node with 2 subtrees: replace node by another to satisfy BST condition
    - Node not really deleted but replaced with leftmost node of the right subtree
    - That leftmost node is deleted by making its parent refer to its right subtree
    - Garbage collect the node

```
(define (delete! bst val)
  (define <<? (lesser bst))
  (define ==? (equality bst))
  (define (find-leftmost deleted parent child! child)
    (if (tree:null-tree? (tree:left child))
        (begin
          (tree:value! deleted (tree:value child))
          (child! parent (tree:right child)))
        (find-leftmost deleted child
                       tree:left!
                       (tree:left child))))
  (define (delete-node parent child! child)
    (cond
      ((tree:null-tree? (tree:left child))
       (child! parent (tree:right child)))
      ((tree:null-tree? (tree:right child))
       (child! parent (tree:left child)))
      (else
       (find-leftmost child
                      child
                      tree:right!
                      (tree:right child)))))
  (let find-node
    ((parent tree:null-tree)
     (child! (lambda (ignore child) (root! bst child)))
     (child (root bst)))
    (cond
      ((tree:null-tree? child)
       #f)
      ((==? (tree:value child) val)
       (delete-node parent child! child)
       (tree:value child))
      ((<<? (tree:value child) val)
       (find-node child tree:right! (tree:right child)))
      ((<<? val (tree:value child))
       (find-node child tree:left! (tree:left child))))))
```

Hfst 6.3 Binary Search Trees (3)

vrijdag 24 december 2021 15:40

A BST-based Implementation of Dictionaries

```
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)
(define (lift proc)
  (lambda (assoc1 assoc2)
    (proc (assoc-key assoc1)
          (assoc-key assoc2))))
(define (new ==? <<?)
  (bst:new
   (lift ==?)
   (lift <<?)))
(define (insert! dct key val)
  (bst:insert! dct (make-assoc key val))
  dct)
(define (delete! dct key)
  (bst:delete! dct (make-assoc key 'ignored))
  dct)
(define (find dct key)
  (define assoc (bst:find dct (make-assoc key 'ignored)))
  (if assoc
      (assoc-value assoc)
      assoc))
```

Discussion

Performance:

- Best case: $O(\log(n))$
- Worst case (only 1 child -> defenerated): $O(n)$

Possibilities degenerated tree

- Insereted in ascending order -> degenerated to the right
- Inserted in descending order -> degenerated to the left
- Inserted alternatingly outside-in -> tree linked list with alternating left brances and right branches

Degenerated trees worse than plain linked lists

- Unused pointers -> more memory
- Relative small trees -> avg BST $O(\log(n))$

	sorted list			
	gelinkte implementatie	vector implementatie	BST implementatie (best case)	BST implementatie (worst case)
new	O(1)	O(1)	O(1)	O(1)
find	O(n)	O(log(n))	Ω(log(n))	O(n)
insert!	O(n)	O(n)	Ω(log(n))	O(n)
delete!	O(n)	O(n)	Ω(log(n))	O(n)

Hfst 6.4 AVL Trees

vrijdag 24 december 2021 15:51

Guaranteeing complete tree -> costly: every insertion/deletion might cause tree to be restructed
Solution : "reasonably balanced" trees -> height subtree 1 - subtree 2 = max(1)

- Not calculate heigth every time but using tags: balanced, left high or right high

Node Representation

```
(define balanced 'balanced)
(define Lhigh 'Lhigh)
(define Rhigh 'Rhigh)

(define-record-type AVL-node
  (make-AVL-node v l r b)
  AVL-node?
  (v value value!)
  (l left left!)
  (r right right!)
  (b balance balance!))

(define-record-type bst
  (make r e l)
  bst?
  (r root root!)
  (e equality)
  (l lesser))

(define (new ==? <<?)
  (make null-tree ==? <<?))
```

Rebalancing by rotation

```
(define (single-rotate-left! black)
  (define white (right black))
  (define tree (left white))
  (right! black tree)
  (left! white black)
  white)

(define (single-rotate-right! black)
  (define white (left black))
  (define tree (right white))
  (left! black tree)
  (right! white black)
  white)

(define (double-rotate-left-then-right! black)
  (define white (left black))
  (left! black (single-rotate-left! white))
  (single-rotate-right! black))

(define (double-rotate-right-then-left! black)
  (define white (right black))
  (right! black (single-rotate-right! white))
  (single-rotate-left! black))
```

Insertion

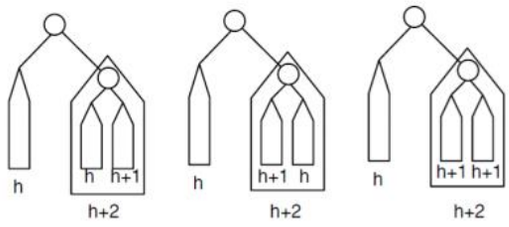
```
(define (insert! avl val)
  (define <<? (lesser avl))
  (define ==? (equality avl))
  (let insert-rec
    ((parent null-tree)
     (child! (lambda (ignore child) (root! avl child)))
     (child (root avl)))
    (cond
      ((null-tree? child)
       (child! parent (make-AVL-node null-tree val balanced null-tree))
       #t)
      ((<<? (AVL-node-value child) val)
       (if (insert-rec child AVL-node-right! (AVL-node-right child))
           (check-after-insert-right parent child! child)
           #f))
      ((<<? val (AVL-node-value child))
       (if (insert-rec child AVL-node-left! (AVL-node-left child))
           (check-after-insert-left parent child! child)
           #f))
      (else ; value = (AVL-node-value node)
       (AVL-node-value! child val)
       #f)))
  avl)
```

Hfst 6.4 AVL Trees (2)

vrijdag 24 december 2021 16:05

```
(define (check-after-insert-right parent child! child)
  (cond
    ((eq? (AVL-node-balance child) Lhigh)
     (AVL-node-balance! child balanced)
     #f)
    ((eq? (AVL-node-balance child) balanced)
     (AVL-node-balance! child Rhigh)
     #t)
    (else ; child already was right-high
     (let* ((right (AVL-node-right child))
            (left (AVL-node-left right)))
           (if (eq? (AVL-node-balance right) Rhigh)
               (begin
                 (child! parent (single-rotate-left! child))
                 (single-rotate-left-update! child right))
               (begin
                 (child! parent (double-rotate-right-then-left! child))
                 (double-rotate-right-then-left-update! child right left)))
           #f))))
```

Grey Node:	Lhigh	balanced	Rhigh
White Node:	Rhigh $(h, h-1)$	balanced (h, h)	balanced $(h-1, h)$
Black Node:	balanced $(h, h-1)$	balanced (h, h)	Lhigh $(h-1, h)$
Grey Node:	balanced	balanced	balanced




```
(define (single-rotate-left-update! black white)
  (cond ((eq? (balance white) Rhigh)
         (balance! black balanced)
         (balance! white balanced))
        (else
         (balance! black Rhigh)
         (balance! white Lhigh))))
(define (double-rotate-right-then-left-update! black white grey)
  (cond ((eq? (AVL-node-balance grey) Lhigh)
         (AVL-node-balance! white Rhigh)
         (AVL-node-balance! black balanced)
         (AVL-node-balance! grey balanced))
        ((eq? (AVL-node-balance grey) balanced)
         (AVL-node-balance! white balanced)
         (AVL-node-balance! black balanced)
         (AVL-node-balance! grey balanced))
        (else
         (AVL-node-balance! white balanced)
         (AVL-node-balance! black Lhigh)
         (AVL-node-balance! grey balanced))))
```

Deletion

```
(define (delete! avl val)
  (define ==? (equality avl))
  (define <<? (lesser avl))
  (define (find-leftmost deleted parent child! child)
    (if (null-tree? (AVL-node-left child))
        (begin
          (AVL-node-value! deleted (AVL-node-value child))
          (child! parent (AVL-node-right child))
          #t)
        (if (find-leftmost deleted child AVL-node-left! (AVL-node-left child))
            (check-after-delete-left parent child! child)
            #f)))
  (define (delete-node parent child! child)
    (cond
      ((null-tree? (AVL-node-left child))
       (child! parent (AVL-node-right child))
       #t)
      ((null-tree? (AVL-node-right child))
       (child! parent (AVL-node-left child))
       #t)
      (else
       (if (find-leftmost child child AVL-node-right! (AVL-node-right child))
           (check-after-delete-right parent child! child)
           #f))))
```

Hfst 6.4 AVL Trees (3)

vrijdag 24 december 2021 16:14

```
  (let find-node
    ((parent null-tree)
     (child! (lambda (ignore child) (root! avl child)))
     (child (root avl)))
    (cond
      ((null-tree? child)
       #f)
      ((==? (AVL-node-value child) val)
       (delete-node parent child! child))
      ((<<? (AVL-node-value child) val)
       (if (find-node child AVL-node-right! (AVL-node-right child))
           (check-after-delete-right parent child! child)
           #f))
      ((<<? val (AVL-node-value child))
       (if (find-node child AVL-node-left! (AVL-node-left child))
           (check-after-delete-left parent child! child)
           #f))))
  avl)
(define (check-after-delete-right parent child! child)
  (cond
    ((eq? (AVL-node-balance child) Rhigh)
     (AVL-node-balance! child balanced)
     #t)
    ((eq? (AVL-node-balance child) balanced)
     (AVL-node-balance! child Lhigh)
     #f)
    (else
     (let← ((left (AVL-node-left child))
            (left-bal (AVL-node-balance left))
            (right (AVL-node-right left)))
           (if (or (eq? left-bal Lhigh)
                   (eq? left-bal balanced))
               (begin
                 (child! parent (single-rotate-right! child))
                 (single-rotate-right-update! child left))
               (begin
                 (child! parent (double-rotate-left-then-right! child))
                 (double-rotate-left-then-right-update! child left right)))
           (not (eq? left-bal balanced))))))
```

Finding

= ordinary BTS find in $O(log(n))$

Hfst 6.5 Comparing Dictionary

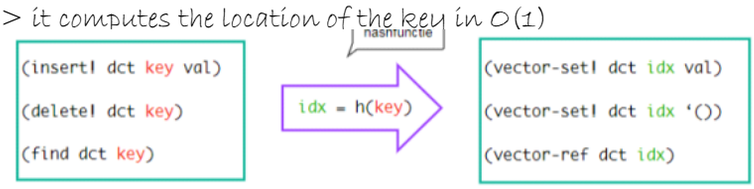
Operation	Sorted List (Vector)	Sorted List (Linked)	Double Linked List	BST	AVL
`insert!`					
(worst)	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$
(average)	$O(n)$	$O(n)$	$O(1)$	$O(log(n))$	$O(log(n))$
`delete!`					
(worst)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(log(n))$
(average)	$O(n)$	$O(n)$	$O(n)$	$O(log(n))$	$O(log(n))$
`find`					
(worst)	$O(log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(log(n))$
(average)	$O(log(n)))$	$O(n)$	$O(n)$	$O(log(n))$	$O(log(n))$

Hfst 7.1 Hashing: Basic Idea

zondag 26 december 2021 12:26

- function h to the key -> to compute the location of the key in the vector-based data structure.
- The data structure: **hash table** & h: **hash function**
- The technique: **hashing**
- try to design the hash function -> it computes the location of the key in O(1)

Basic Idea: Performance O(1)



- Keys: between 0 & M-1
- Values: stored in vector "dict"
- Index vector: keys
- Data value: value of key-value pair
- Inserting key-value pair (k,v): store value in vector on index key -> (vector-set! Dict k v)
- Find: (vector-ref dict k)
- Delete!: storing '()

Hash tables: 2 steps

- Hash function (h): transforms key into number in O(1)
- **Number key** has to be **small** to fit into the vector -> (module i M) i number ordinary key -> M amount of enteries in the vector

Home address of k: number that comes out the 2 steps

Funneling: een deel van de key in een deel van de tabel komt

Issues to be resolved:

- Collision: keys with the same home adress
  - Solution: conflict resolting strategy
- Perfect Hash function: no collisions
  -> generally impossible
  - Load factor: hash table of size M contains n key-value pairs
    -> load factor $\alpha = \frac{n}{M} = \frac{\text{number of elements contained in the hash table}}{\text{the number of entries it supplies}}$
  - (hash table filled: α . 100 %)
  - Control changes of collision: load factor < 1 (even beter 0.75)
  - When factor > 1: hash table grow by rehashing into new table (mostly 2x as big)

Hfst 7.2 Collision Resolution Strategies

External Chaining

- External Chaining: bestaat erin van gelinkte lijsten bij te houden als meerdere keys op hetzelfde home adres terecht komen. Deze lijsten noemen we buckets => alfa kan groter zijn dan 1
- If **2 keys hash to the same home address** in the hash table, a **linked list** is stored in the table. This turns the hash table into a vector of linked lists, also called buckets

```
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)
(define-record-type external-chaining
  (make s h e)
  dictionary?
  (s storage)
  (h hash-function)
  (e equality))
;assume h O(1)
(define (new ==? M h) ;M = size, h = hash function
  (make (make-vector M ()) (lambda (k) (modulo (h k) M)) ==?))

(define (insert! table key val) ;O(n)
  (define vector (storage table))
  (define h (hash-fct table))
  (define ==? (equality table))
  (define home-address (h key))
  (define assoc (make-assoc key val))
  (let insert-in-bucket
    ((prev '())
     (next! (lambda (ignore next)
              (vector-set! vector home-address next)))
     (next (vector-ref vector home-address)))
    (cond
      ((null? next)
       (next! prev (cons assoc next)))
      ((==? (assoc-key (car next)) key)
       (set-car! next assoc))
      (else
       (insert-in-bucket next set-cdr! (cdr next)))))
  table)
```

```
(define (find table key) ;O(n)
  (define vector (storage table))
  (define h (hash-fct table))
  (define ==? (equality table))
  (define home-address (h key))
  (let find-in-bucket
    ((next (vector-ref vector home-address)))
    (cond
      ((null? next)
       #f)
      ((==? (assoc-key (car next)) key)
       (assoc-value (car next)))
      (else
       (find-in-bucket (cdr next))))))
```

Worst-case: alle keys hashen naar hetzelfde home adres. Dat geeft O(n) voor alle operaties.

```
(define (delete! table key) ;O(n)
  (define vector (storage table))
  (define h (hash-fct table))
  (define ==? (equality table))
  (define home-address (h key))
  (let delete-from-bucket
    ((prev '())
     (next! (lambda (ignore next)
              (vector-set! vector home-address next)))
     (next (vector-ref vector home-address)))
    (cond
      ((null? next)
       #f)
      ((==? (assoc-key (car next)) key)
       (next! prev (cdr next))
       table)
      (else
       (delete-from-bucket next set-cdr! (cdr next)))))
  table)
```

Best-case: als de hash functie de keys uniform spreidt, heeft elke bucket 1/M kans. Met n aanwezige sleutels, geeft dat lijsten met een gemiddelde lengte α. Dus O(α).

Hfst 7.2 Collision Resolution Strategies

zondag 26 december 2021 14:44

Performantie (avarage case)

- probability of a key to end up in one particular bucket is 1/M
- Lenght $\alpha = n / M$ => linked lists avg time $\alpha/2$
- Find best case: $O(1 + \alpha/2)$

Operation	Performance
`find`	$O(1+\alpha)$
`insert!`	$O(1+\alpha)$
`delete!`	$O(1+\alpha)$

The table size

$f(k) = h(k) mod (M)$ met $h(k)$ hash functie

Funneling: enteries of the table remain unused

<-> **goal**: uniformly distribute the key-value pairs as much as possible over the entire table in order to make collisions as unlikely as possible

- $h(k)$ and M have a multiplicative factor lambda in common
  => assume: $h(k) = \gamma.r_{h(k)}$ & $M = \gamma.r_M$
- Defenitie: $a = \beta N + b \Rightarrow a = b \ mod \ N$
- => $f(k) = \beta M + h(k) = \beta\gamma r_M + \gamma.r_{h(k)} = \gamma X$

How do we choose M that has no factors in commen?

- M = prime that are not too close to exact powers of 2
- M and h(k) have **no factors is common** is to make sure that 2 is the only factor of M and that h(k) does not have 2 as a factor

Open Addressing Methods

collects **colliding elements** in linked lists, open addressing methods store such **elements elsewhere in the same hash table** (rehashing)

- $\alpha < 1$: cannot store more key-value pairs in the table than the number of entries it provides
- Probe: new location that is obtained after rehashing is occupied => the rehashing algorithm
  => probe sequence: the result
  - The longer the probe sequence, the worst the performance

Lineair probing

$h'(k,i) = (h(k) + i.c) \ mod \ M$ where $i \in \{1, 2, 3, 4, ...\}$

- C = constante (often 1)
- Single-slot stepping: C is added to the home address in order to come up with the next probe

Asume c and M have a factor lambda in common:

$$\begin{aligned} h'(k,i) &= (h(k) + i.c) \ mod \ M \\ &= \beta.M + (h(k) + i.c) \\ &= \beta.\gamma.r_M + h(k) + i.\gamma.r_c \\ &= h(k) + \gamma(\beta.r_M + i.r_c) \end{aligned}$$

for all i, h'(k, i) will end up at the home address h(k) + lambda − fold.

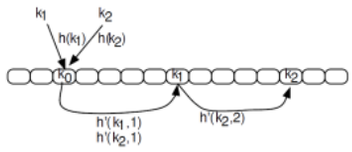
All table entries which are unreachable by adding lambda − folds to the home address remain unvisited => repetitively **jumping around at the same locations** in the table

Solution:

- Tombstone
- Nonrepetitious complete coverage: c & M no common factors => c & M realtive prime

Hfst 7.2 Collision Resolution Strategies

zondag 26 december 2021 15:43




```
(define-record-type linear-rehashing
  (make s h e)
  dictionary?
  (s storage)
  (h hash-function)
  (e equality))

(define (new ==? M h)
  (make (make-vector M 'empty) (lambda (x) (modulo (h x) M)) ==?))

(define c 1)
(define (rehash address M)
  (modulo (+ address c) M))

(define (insert! table key val)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h (hash-function table))
  (define ==? (equality table))
  (define new-assoc (make-assoc key val))
  (let rehash-iter
    ((address (h key)))
    (let ((assoc (vector-ref vector address)))
      (cond ((or (eq? assoc 'empty)
                 (eq? assoc 'deleted))
             (vector-set! vector address new-assoc))
            ((==? (assoc-key assoc) key)
             (vector-set! vector address new-assoc))
            (else
             (rehash-iter (rehash address M))))))
  table)

(define (find table key)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h (hash-function table))
  (define ==? (equality table))
  (let rehash-iter
    ((address (h key)))
    (let ((assoc (vector-ref vector address)))
      (cond
        ((eq? assoc 'empty)
         #f)
        ((eq? assoc 'deleted)
         (rehash-iter (rehash address M)))
        ((==? (assoc-key assoc) key)
         (assoc-value assoc))
        (else
         (rehash-iter (rehash address M)))))))
```

```
(define (delete! table key) ;O(n)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h (hash-function table))
  (define ==? (equality table))
  (let rehash-iter
    ((address (h key)))
    (let ((assoc (vector-ref vector address)))
      (cond
        ((eq? assoc 'empty)
         #f)
        ((eq? assoc 'deleted)
         (rehash-iter (rehash address M)))
        ((==? (assoc-key assoc) key)
         (vector-set! vector address 'deleted))
        (else
         (rehash-iter (rehash address M))))))
  table)

;for linear probing with single-slot stepping (c=1)
(define (delete! table key)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h (hash-function table))
  (define ==? (equality table))
  (define (between x <<1 y <<2 z)
    (and (<<1 x y)
         (<<2 y z)))
  (define (storage-move prev next)
    (if (eq? (vector-ref vector next) 'empty)
        prev
        (let ((home (h (assoc-key (vector-ref vector next)))))
          (if (or (between prev < home <= next)
                  (between home <= next < prev)
                  (between next < prev < home))
              (storage-move prev (rehash next M))
              (begin (vector-set! vector prev (vector-ref vector next))
                     (storage-move next (rehash next M)))))))
  (let rehash-iter
    ((address (h key)))
    (let ((assoc (vector-ref vector address)))
      (cond
        ((eq? assoc 'empty)
         #f)
        ((==? (assoc-key assoc) key)
         (vector-set!
          vector
          (storage-move address (rehash address M)) 'empty))
        (else
         (rehash-iter (rehash address M))))))
  table)
```

The Clustering Phenomenon

Clustering: elements occurring in one probe sequence can also be part of other probe Sequence

- **Primary Clustering**: once 2 keys rehash to the same adress somewhere along there probe sequence => rest of their ptobe sequence will be identical
  - **Lineair Probing**: 2 differnt probe sequences add togheter because 1 rehashing adress in common
- **Secondary Clustering**: merging probe sequences -> the probe sequences of the keys cluster: for all i, we have h'(k1, i) = h(k2, i)

Indien
$h'(k_1,i) = h'(k_2,j)$ impliceert dat $\forall\ i'>i,\ j'>j : h'(k_1,i') = h'(k_2,j')$
dan
spreekt men van primary clustering

Indien
$h'(k_1,0) = h'(k_2,0)$ impliceert dat $\forall\ i: h'(k_1,i) = h'(k_2,i)$
dan
spreekt men van secondary clustering

Lineair Probing suffers for primary & secondary clustering

Nadeel: probe sequeences get longer and longer

Quadratic Probing/ rehashing

- the step size increase for every rehash -> grows quadratically
- eliminates the problem of primary clustering

$$h'(k,i) = (h(k) + c_1.i + c_2.i^2)\ mod\ M$$

Make sure every enteries are visited:

c1 = 0, c2 = 1

M a prime number of the form 4k + 3

(other choses also exists but this works the best)

Problem primary clustering fixed but not 2ary:

rehash to different locations in the next iteration

(both keys are rehashed using a different step size)

```
(define (find table key)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h (hash-function table))
  (define ==? (equality table))
  (let rehash-iter
    ((address (h key))
     (odd 1))
    (let ((assoc (vector-ref vector address)))
      (cond
        ((eq? assoc 'empty)
         #f)
        ((eq? assoc 'deleted)
         (rehash-iter (rehash address odd M) (+ odd 2)))
        ((==? (assoc-key assoc) key)
         (assoc-value assoc))
        (else
         (rehash-iter (rehash address odd M) (+ odd 2)))))))

(define (rehash address j M)
  (modulo (+ address j) M))
```

Hfst 7.2 Collision Resolution Strategies

zondag 26 december 2021 16:42

Double Hashing

Avoids Primary & Secondary clustering: different step sizes even for keys that end up at the same home address (c of the linear probing method is replaced by a value that depends on the key)

$h'(k,i) = (h_1(k) + i.h_2(k))\ mod\ M$ where $i \in \{1,2,3,4,...\}$

which are the good values for this technique to cover the entire hash table?

M and h2(k) are relative prime

- M to be a power of two (such that 2 is its only factor) & h such that it always returns an odd number (such that 2 is not a factor)
- M to be prime & $h_2(k)$ to be smaller than M (such that it does not share a factor with M)

Ex. h1(k) = k mod M and h2(k) = 1+(k mod M0) where M' = M − 1 or M = M − 2.

```
(define-record-type double-rehashing
  (make s h1 h2 e)
  dictionary?
  (s storage)
  (h1 hash-function1)
  (h2 hash-function2)
  (e equality))

(define (new ==? M h1 h2)
  (make (make-vector M 'empty)
        (lambda (x) (modulo (h1 x) M)) h2 ==?))

(define (rehash key address h2 M)
  (modulo (+ address (h2 key)) M))

(define (find table key)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h1 (hash-function1 table))
  (define h2 (hash-function2 table))
  (define ==? (equality table))
  (let rehash-iter
    ((address (h1 key)))
    (let ((assoc (vector-ref vector address)))
      (cond
        ((eq? assoc 'empty)
         #f)
        ((eq? assoc 'deleted)
         (rehash-iter (rehash key address h2 M)))
        ((==? (assoc-key assoc) key)
         (assoc-value assoc))
        (else
         (rehash-iter (rehash key address h2 M)))))))
```

	Unsuccesful	Succesful
Linear rehashing	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$
Double rehashing Quadratic rehashing	$\frac{1}{1-\alpha}$	$-\left(\frac{1}{\alpha}\right) \times log(1-\alpha)$
External Chaining	$1+\alpha$	$1+\frac{1}{2}\alpha$

OA vs Extern Chainen:

- Tombstone genereers veel afvl die probe sequences langer maakt => performantie niet afhankelijk van alfa
- Lijsten kunnen wel de overhand krijgen, andere datastructuren wegen niet op
- Extern chainen kan alfa > 1 => volledig rehashen van de tabel is nooit nodig

Hfst 7.3 Hash Functions

zondag 26 december 2021 16:53

Perfect Hash Functions

A perfect hash function is a hash function that yields no collisions => h(k1) = h(k2) implies k1 = k2
Olmost impossible

Good Hash Functions

2 problems

- Funneling: entire parts of the table never to be considered for the keys. Whenever hashing a key, the possibility arises that we keep on stepping in circles in the table even though there is plenty of free space in the table
- Clustering: causes probe sequences of distinct keys to merge in some cases

Hash functions has to be:

- Simple: with complex functions it is harder to study its funnelling and clustering characteristics
- Fast: entire idea of hashing is to mimic the O(1) direct access performance characteristic of vectors
- Strong: uniformly distributes the keys over the entries of the table (entery should be 1/M)

Assume that keys consists of a (possibly large) integer number. The following hash functions provide us with ways to convert them into smaller numbers that can be used as indexes in the hash table

- Folding Method: take all the digits of a key and to combine them
  - performs quite poorly: order between the digits plays no role whatsoever in the final value returned by the hash function. All keys the digits of which are permutations of each other hash to the same home address
- Digit Selection Method: take a key that consist of n digits, $k = d_1d_2d_3...d_n$ and to select a combination of digits $h(k) = d_id_jd_k$ that results in a good uniform distribution
  - digit analysis for a large number of concrete keys that can occur in a given situation: ultimate goal of a digit analysis is to select digits which achieve avalanche.
    Avalanche: a difference of one single bit in the input guarantees thathalf of the bits in the output are different.
    If avalanche is achieved, it means that every bit (which can take two different states, namely 0 or 1) in the input is used to its full extent & has a large effect on the output
- Division Method: key k and "truncate" it by considering the remainder of division by M, the size of the hash table -> h(k) = k mod M
  - simple and performs quite well
  - Drawback: prime numbers are not easily computed
- Multiplication Method: multiply the key k with some constant C (with $0 < C < 1$). The fractional part of this real number is then multiplied by M: $h(k) = \lfloor M.(kC \bmod 1) \rfloor$

Hfst 7.3 Hash Functions

Search-Based implementations: traverses the dictionary and compares the search key with the keys that the algorithm encounters during the traversal

- Requires an ==? Operator & additional ordering operator <<?

Hashed implementations: avoid searching for keys by requiring the keys to map themselves to their home address in the storage data structure

- not need the ordering <<?

Bewijzen - Heaps

dinsdag 28 december 2021 10:54

Eigenschap: Heap van n elementen: hoogte $h = \lfloor log_2 n \rfloor$ (aantal takken van langste pad tot leaf)
Bewijs:

- Formule meetkundige rij: $\sum_{i=0}^{h} a^i = \frac{a^{h+1}-1}{a-1}$
- Op niveau i<h : 2^i elementen
- Op niveau h:
  - min(h) = 1
  - max(h) = 2^h

$\sum_{i=0}^{h-1} 2^i + 1 \leq n \leq \sum_{i=0}^{h} 2^i$
min # nodes = #nodes op level h-1 + 1 node ≤ n (#nodes) ≤ max # nodes
$\ll\Rightarrow \frac{2^{h-1+1}-1}{2-1} + 1 \leq n \leq \frac{2^{h+1}-1}{2-1}$ (formule meetkundige rij)
$\ll\Rightarrow \frac{2^h-1}{1} + 1 \leq n \leq \frac{2^{h+1}-1}{1}$
$\ll\Rightarrow 2^h - 1 + 1 \leq n \leq 2^{h+1} - 1$
$\ll\Rightarrow 2^h \leq n \leq 2^{h+1} - 1$
$\ll\Rightarrow 2^h \leq n < 2^{h+1}$ (bovengrens groter maken)
$\ll\Rightarrow log_2(2^h) \leq log_2(n) < log_2(2^{h+1})$ (van alles log_2 nemen)
$\ll\Rightarrow h \leq log_2 n \leq h+1$ (uitweken van log)
$\ll\Rightarrow h = \lfloor log_2 n \rfloor$ (haakjes staan voor afgerond naar beneden)

Eigenschap 2: heap met n elementen heeft $\lceil \frac{n}{2} \rceil$ bladeren
Bewijs: Uit het ongerijmde
Stel: #bladeren heap < $\lceil \frac{n}{2} \rceil$
Neem het laatste element dat geen leaf is: index element i > $\lceil \frac{n}{2} \rceil$ (defenitie heap)
=> leaf heeft min 1 child op index 2i > $2\lceil \frac{n}{2} \rceil$ <=> 2i > n
=> onmogelijk want het totaal aantal elementen van de heap is n dus je kan nooit een child hebben op positie >n
-> contractie
Stel: #bladeren heap > $\lceil \frac{n}{2} \rceil$
Neel het laatste element dat een blad is: index leaf = n dan is parent leaf $\frac{n}{2}$ dit zou ook een leaf moeten zijn want #bladeren heap > $\lceil \frac{n}{2} \rceil$ dit kan niet want een leaf kan geen kinderen hebben
-> contradictie
Stelling is bewezen door insluiting.

Eigenschap 3: heap van n elementen heeft op hoogte h maximum $\lceil \frac{n}{2^{h+1}} \rceil$ elementen
Bewijs: per inductie

- Op hoogte 0 (laag met alle leaves) => #bladeren $\lceil \frac{n}{2} \rceil$
- Op hoogte 1 (laag met parents van de leaves) => #elementen $\left\lceil \frac{\frac{n}{2}}{2} \right\rceil = \lceil \frac{n}{4} \rceil = \lceil \frac{n}{2^2} \rceil$
- Op hoogte 2 => $\left\lceil \frac{\frac{n}{4}}{2} \right\rceil = \lceil \frac{n}{8} \rceil = \lceil \frac{n}{2^3} \rceil$
- Op hoogte h => $\lceil \frac{n}{2^{h+1}} \rceil$

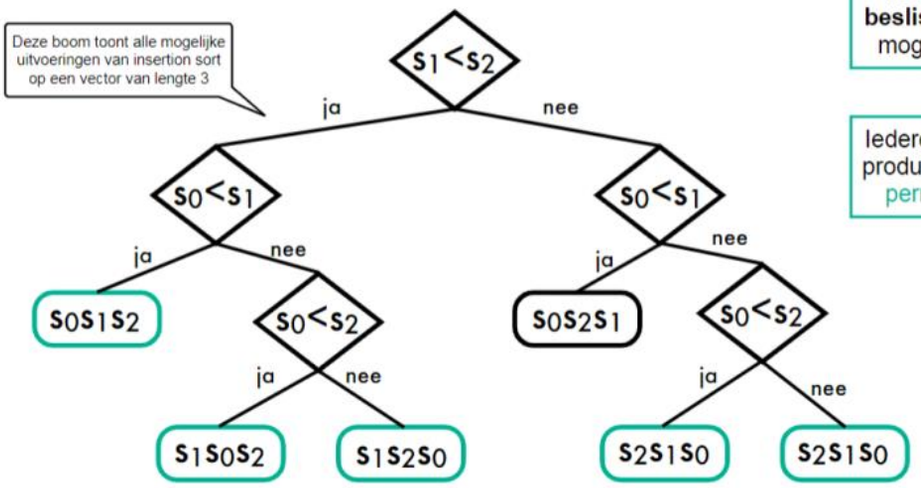
Bewijzen - Heapify

dinsdag 28 december 2021 12:18

Stelling: geen enkel algoritme dat gebaseerd is op vergelijkingen alleen kan sneller dan n.log(n)

Voor ieder algoritme kan je een **beslissingsboom** tekenen die alle mogelijke executies visualiseert.

Iedere uitvoering van het algoritme produceert één van de n! mogelijke permutaties van de inputvector



Bewijs

Stelling: geen enkel algoritme dat gebaseerd is op vergelijkingen alleen kan sneller dan n.log(n)

Bewijs:

n! permutaties

Dus: n! leaves

Dus: minstens n! knopen

Dus: hoogte h > log(n!)

Dus h > n.log(n) want log(n!) ∈ Ω(n.log(n))

h is de hoogte van de beslissingsboom. Dus minstens Ω(n.log(n)) vergelijkingen!

Lemma: log(n!) ∈ Ω(n.log(n))

Bewijs:

log(n!)

$$= log(1) + log(2) + \ldots + log(\frac{n}{2}) + \ldots + log(n)$$

$$\geq log(\frac{n}{2}) + \ldots + log(n)$$

$$\geq log(\frac{n}{2}) + \ldots + log(\frac{n}{2})$$

$$= \frac{n}{2} log(\frac{n}{2})$$

$$= \frac{n}{2} log(n-1)$$

$$> \frac{n}{2} log(n) - \frac{n}{6} log(n) \; voor \; n > 8 = 2^3 \; (= \; n_0)$$

$$Dus \; log(n!) > \frac{1}{3} n \, . \, log(n) \; voor \; n > 8$$