Chapter 7

Hashing

Inhoud

- Basisidee en concepten
  - Hashfunctie, hashtabel, load factor
  - Funneling
- External chaining
- Open addressing (linear & quadratic probing, double hashing)
  - Clustering
  - Tombstones
- Hashfuncties

Herinner ons doel

```
ADT dictionary< K V >

new
   ( ( K K → boolean ) → dictionary< K V > )
dictionary?
   ( any → boolean )
insert!
   ( dictionary< K V > K V → dictionary< K V > )
delete!
   ( dictionary< K V > K → dictionary< K V > )
find
   ( dictionary< K V > K → V ∪ {#f} )
empty?
   ( dictionary< K V > → boolean )
full?
   ( dictionary< K V > → boolean )
```

Kunnen we sneller dan O(log(n))?

Bomen (en binair zoeken)

Vergelijk: dictionaries vs. vectoren

```
(insert! dct key val)

(delete! dct key)

(find dct key)
```

```
(vector-set! dct idx val)

(vector-set! dct idx '())

(vector-ref dct idx)
```

```
insert!    O(1)

delete!    O(1)

find       O(1)
```

Vectoren zijn eigenlijk dictionaries met getallen (= indices) als sleutels. Kunnen we dat idee veralgemenen?

Veralgemening naar andere keys

```
(insert! dct key val)

(delete! dct key)

(find dct key)
```

hashfunctie

idx = h(key)

```
(vector-set! dct idx val)

(vector-set! dct idx '())

(vector-ref dct idx)
```

Een hashfunctie beeldt sleutels af op vector-indices in O(1)

2 stappen:
1. berekening van i door h
2. binnen bereik brengen van vector: `(modulo i M)`

Het resultaat heet het home address van de sleutel

Hashfuncties

Twee voorbeelden

Een collision doet zich voor als h(k) = h(k')

Een collision resolution strategie is nodig om een collision op te lossen

Een perfecte hashfunctie is een hashfunctie die geen collisions veroorzaakt

Perfecte hash functies bestaan nagenoeg niet (zie later)

We onderzoeken dan maar volop collision resolution strategieën

folding

digit selection

```
Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512 MB.
> (apply + (map char->integer
                (string->list "hashing is fun")))
1351
> (apply + (map char->integer
                (string->list "dancing is fun")))
1337
> (let* ((s "hashing is fun")
         (l (string-length s)))
    (+ (* 26 (char->integer (string-ref s (- l 2))))
       (char->integer (string-ref s (- l 1)))))
3152
> (let* ((s "dancing is fun")
         (l (string-length s)))
    (+ (* 26 (char->integer (string-ref s (- l 2))))
       (char->integer (string-ref s (- l 1)))))
3152
>
```

De load factor

Indien we een hash tabel met tabelgrootte M hebben die op een gegeven moment n elementen bevat, dan definieert men de load factor als

$$\alpha = \frac{n}{M}$$

Hou het aantal botsingen onder controle door $\alpha < 0{,}75$

De hashtabel is $100 \times \alpha$ percent vol

*De tabel vergroten betekent **alle** keys herhashen want de locatie hangt van M af.*

Het Funneling Fenomeen

Beschouw het gedrag van: f(k) = h(k) mod M

We kiezen h(k) = k en M = 25

We hashen achtereenvolgens 0, 5, 10, 15, 20, 25, 30, ..., 100

Alle keys komen terecht in 0, 5, 10, 15 en 20!

Funneling is het fenomeen waarbij een deel van de keys in een deel van de tabel komt.

De andere locaties worden nooit bezocht voor die keys

Oorzaak van Funneling

Bvb. 3 = 18 mod 5
want 3 = (-3).5+18

Beschouw het gedrag van: f(k) = h(k) mod M

Definitie: a = b mod N
⇔ a = β.N + b voor zekere β

Veronderstel dat h(k) en M een factor γ gemeen hebben.

M.a.w., $\exists r_{h(k)}, r_M : h(k) = \gamma r_{h(k)}$ en $M = \gamma r_M$

$$
\begin{aligned}
\Rightarrow f(k) &= h(k) \mod M \\
&= h(k) + \beta M \\
&= \gamma r_{h(k)} + \beta \gamma r_M \\
&= \gamma (r_{h(k)} + \beta r_M)
\end{aligned}
$$

⇒ alle f(k) zijn γ-vouden en komen dus op dezelfde plekken.

Kies M priem

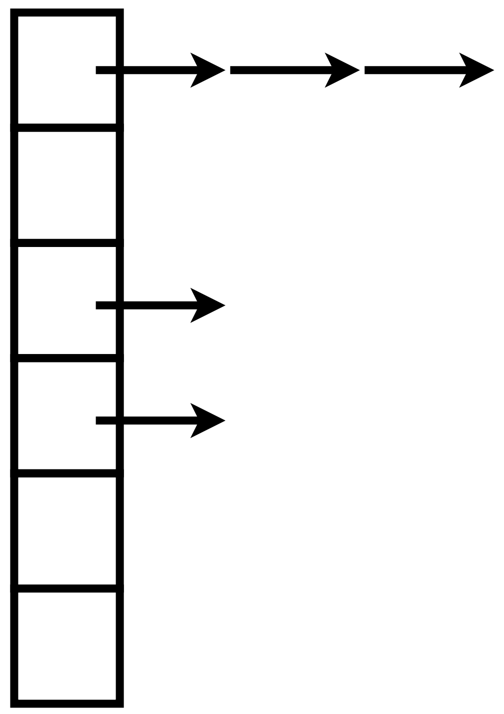
Kies M perfecte 2-macht en zorg dat h(k) oneven is.

Maar de hele tabel gebruiken (zonder funneling) is geen garantie voor geen botsingen

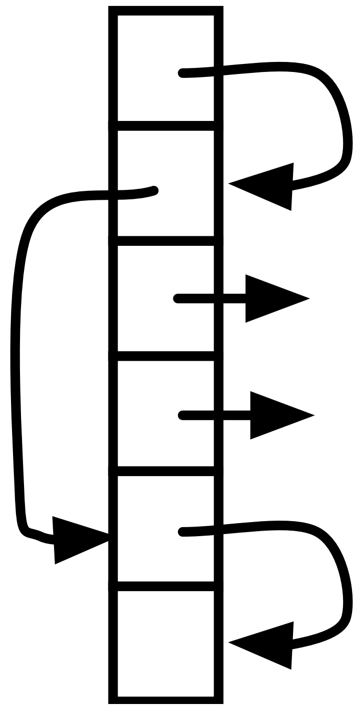
Collision Resolution Strategieën

Eerst bestuderen we collision resolution strategieën

External Chaining



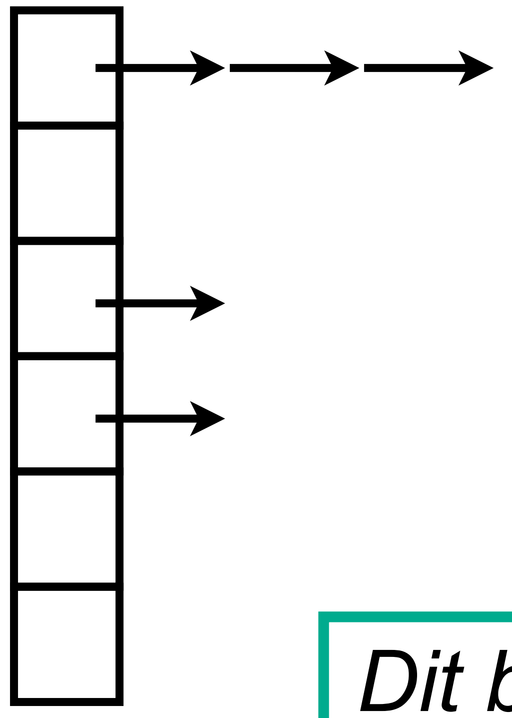
Open addressing:
- *Linear Probing*
- *Quadratic Probing*
- *Double Rehashing*



Daarna bestuderen we enkele hashfuncties

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*.



Dit betekent dat $\alpha > 1$ kan worden

External Chaining: Representatie

```scheme
(define-record-type external-chaining
  (make s h e)
  dictionary?
  (s storage)
  (h hash-function)
  (e equality))

(define (new ==? M h)
  (make (make-vector M '())
        (lambda (k)
          (modulo (h k) M))
        ==?))
```

```scheme
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)
```

Sequentieel zoeken

```scheme
(define (find table key)
  (define vector (storage table))
  (define h (hash-function 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))))))
```

External Chaining: Toevoegen

```
(define (insert! table key val)
  (define vector (storage table))
  (define h (hash-function 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)
```

Chasing Pointers

next! plakt de knoop aan de previous

External Chaining: Verwijderen

```
(define (delete! table key)
  (define vector (storage table))
  (define h (hash-function 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)
```

Chasing Pointers

next! plakt de knoop aan de previous

External Chaining: Eigenschappen

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

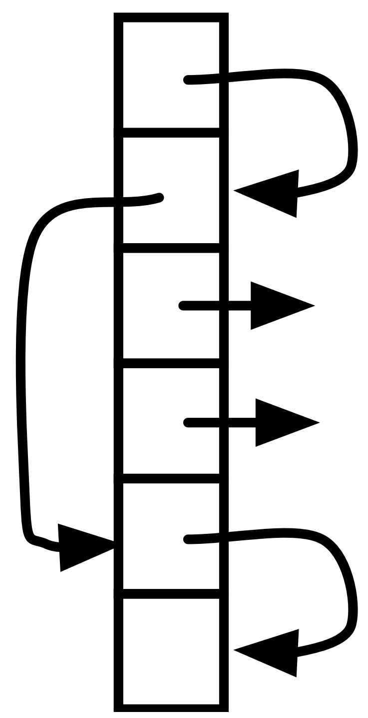
Best-case: als de hash functie de keys uniform spreidt, heeft elke bucket 1/M kans. Met n aanwezige sleutels, geeft dat lijsten met een gemiddelde lengte α. Dus O(α).

Open addressing

Open addressing bestaat erin van botsende keys te herhashen in de hoop een vrije plaats te vinden.

De rij van probeersels die leidt naar een vrije plaats heet een probe sequence.

Elk probeersel noemen we een probe.



$\alpha \leq 1$

OA #1: Lineaire Probing

Opeenvolgende probes

h'(k,i) = (h(k) +i.c) mod M met i = 1, 2, 3, 4, ...

Stel dat c en M een factor γ gemeen hebben.

M.a.w., $\exists r_c, r_M : c = \gamma r_c$ en $M = \gamma r_M$

$$\begin{aligned} \Rightarrow h'(k,i) &= (h(k) + ic) \mod M \\ &= h(k) + ic + \beta M \\ &= h(k) + i\gamma r_c + \beta\gamma r_m \\ &= h(k) + \gamma(ir_c + \beta r_m) \end{aligned}$$

Dus we "springen rond" in γ-vouden en slaan andere locaties over!

Kies c en M relatief priem

Linear Probing: Representatie

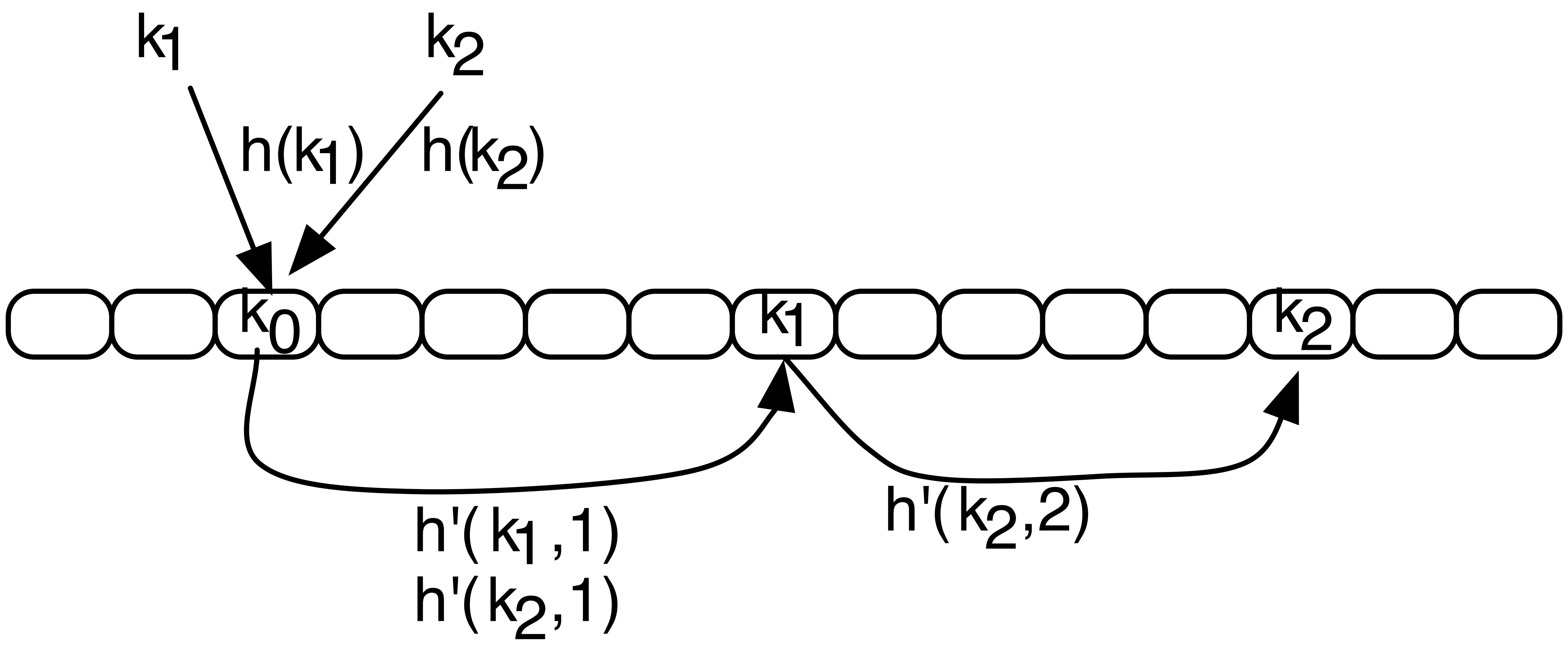
```
(define-record-type linear-rehashing
  (make s h e)
  dictionary?
  (s storage)
  (h hash-function)
  (e equality))

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

```
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)
```

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

Bereken h’(k,i+1) op basis van h’(k,i)

Open Addressing: Verwijderen



Als we k_0 verwijderen zijn we k_1 en k_2 “kwijt”!

Oplossing: de Tombstone


```
(define (delete! 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)
         (vector-set! vector address 'deleted))
        (else
         (rehash-iter (rehash address M))))))
  table)
```

'deleted

Oplossing: de Tombstone

```
(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)
```

Zowel `'empty` *als* `'deleted` *zijn lege vakjes*

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

`'deleted` *is deel van probe sequences. Dus verder zoeken!*

Clustering

Indien
$h'(k_1,i) = h'(k_2,j)$ impliceert dat $\forall$ $m>0$: $h'(k_1,i+m) = h'(k_2,j+m)$
dan
spreekt men van primary clustering

De probe sequences verstrengelen vanaf een bepaalde combinatie van i en j.

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

De probe sequences verstrengelen vanaf het begin.

Primary clustering impliceert secondary clustering maar niet omgekeerd

Primary clustering krijg je weg door verschillende stapgrootte te nemen als $i \neq j$

Kwadratische Probing

Lineaire probing lijdt aan beide vormen van clustering

Secondary clustering krijg je weg door verschillende stapgrootte te nemen als $k_1 \neq k_2$

Dubbel Rehashen

OA #2: Kwadratische Probing

$h'(k,i) = (h(k) + c_1 i + c_2 i^2) \bmod M$ met $i = 1, 2, 3, 4, \ldots$

Primary clustering wordt vermeden

Merk op (voor c_1=0, c_2=1):

$h'(k,i) = h'(k,i-1) + kwadraat_i$

$kwadraat_i = kwadraat_{i-1} + oneven_i$

$0 \rightsquigarrow (+1) \rightsquigarrow 1 \rightsquigarrow (+3) \rightsquigarrow 4 \rightsquigarrow (+5) \rightsquigarrow 9 \rightsquigarrow (+7) \rightsquigarrow 16 \rightsquigarrow (+9) \rightsquigarrow 25 \ldots$

Kwadratische Probing

```
(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))
```

OA #3: Dubbel Rehashing

2 Hashfuncties

$h'(k,i) = (h_1(k) + h_2(k)i) \bmod M$ *met* $i = 1, 2, 3, 4, \ldots$

Secondary clustering wordt vermeden

M en $h_2(k)$ moeten relatief priem zijn. Ofwel M priem, ofwel M 2-macht en $h_2(k)$ oneven

Is eigenlijk een variante van linear rehashen, met niet-constante c

Dubbel Rehashing

```
(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 make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)
```

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

Dubbel Rehashing

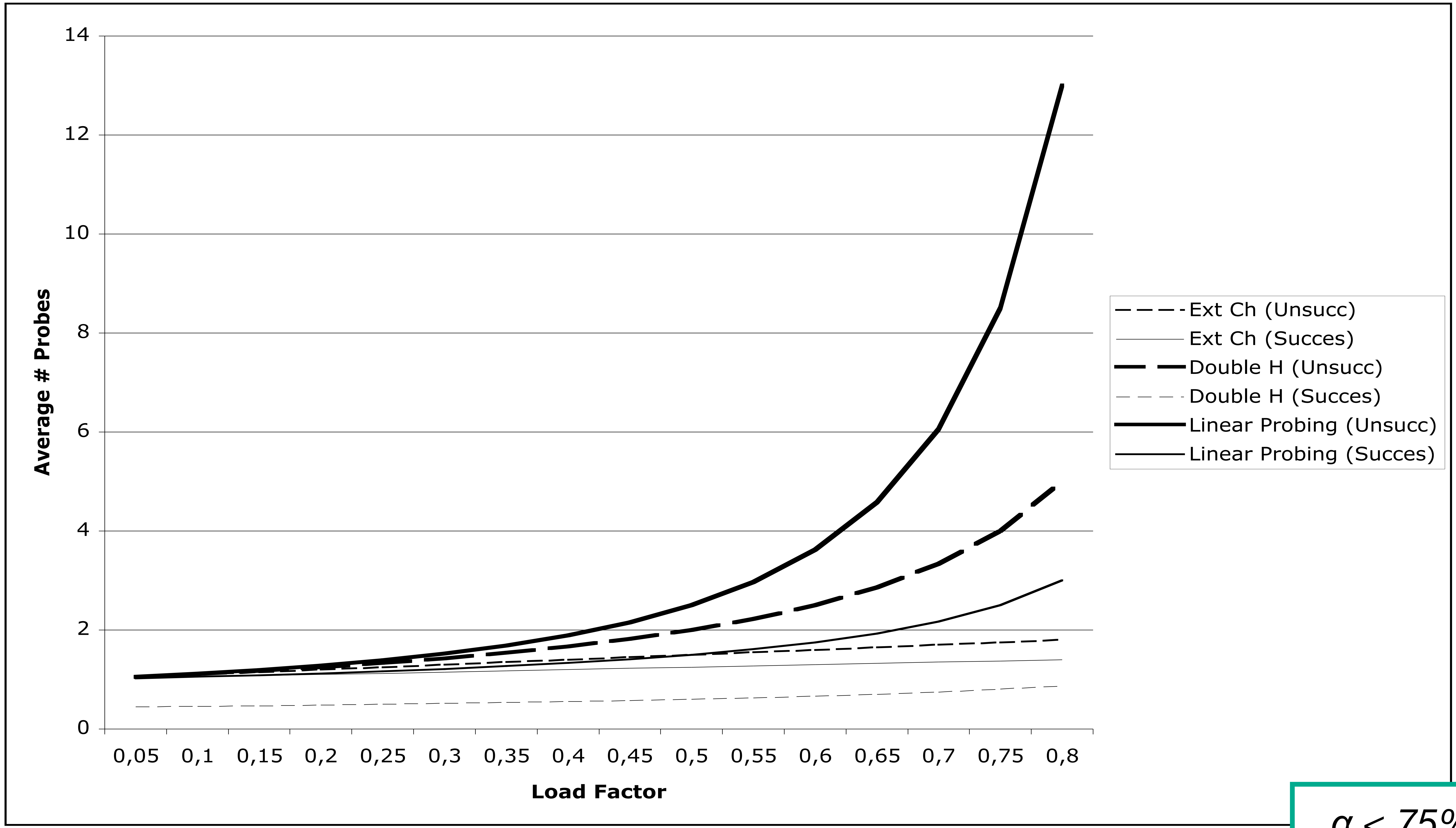
```
(define (find table key)
  (define vector (storage table))
  (define M (vector-length vector))
  (define h1 (hash-function1 table))
  (define h2 (hash-function2 table))
  (define ==? (equality table))
  (let rehash-iter
    ((address (h1 key)))
    (let ((assoc (vector-ref vector address)))
      (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)))))))
```

Performantie Hashing

	onsuccesvol	succesvol
lineair rehashen	$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$	$\frac{1}{2}(1 + \frac{1}{(1-\alpha)})$
dubbel / kwadratisch rehashen	$\frac{1}{1-\alpha}$	$\frac{-1}{a}(\log(1-a))$
extern chainen	$1 + \alpha$	$1 + \frac{1}{2}\alpha$

`delete!` *is een succesvolle opzoeking.*
`insert!` *een onsuccesvolle*

Betekenis van de formules



$\alpha < 75\%$

OA vs. Extern Chainen: Bedenkingen

De tombstone genereert veel afval die probe sequences langer maakt! Dus is de performantie niet meer enkel afhankelijk van α

Lijsten kunnen wel de overhand krijgen. Andere datastructuren wegen niet op.

Bij extern chainen kan α>1 worden. Het volledig herhashen van de tabel is nooit nodig

Perfecte Hashfuncties

h(k) = h(k') impliceert k=k'

Zelfs bij uniforme distributie van keys: heel moeilijk te vinden.

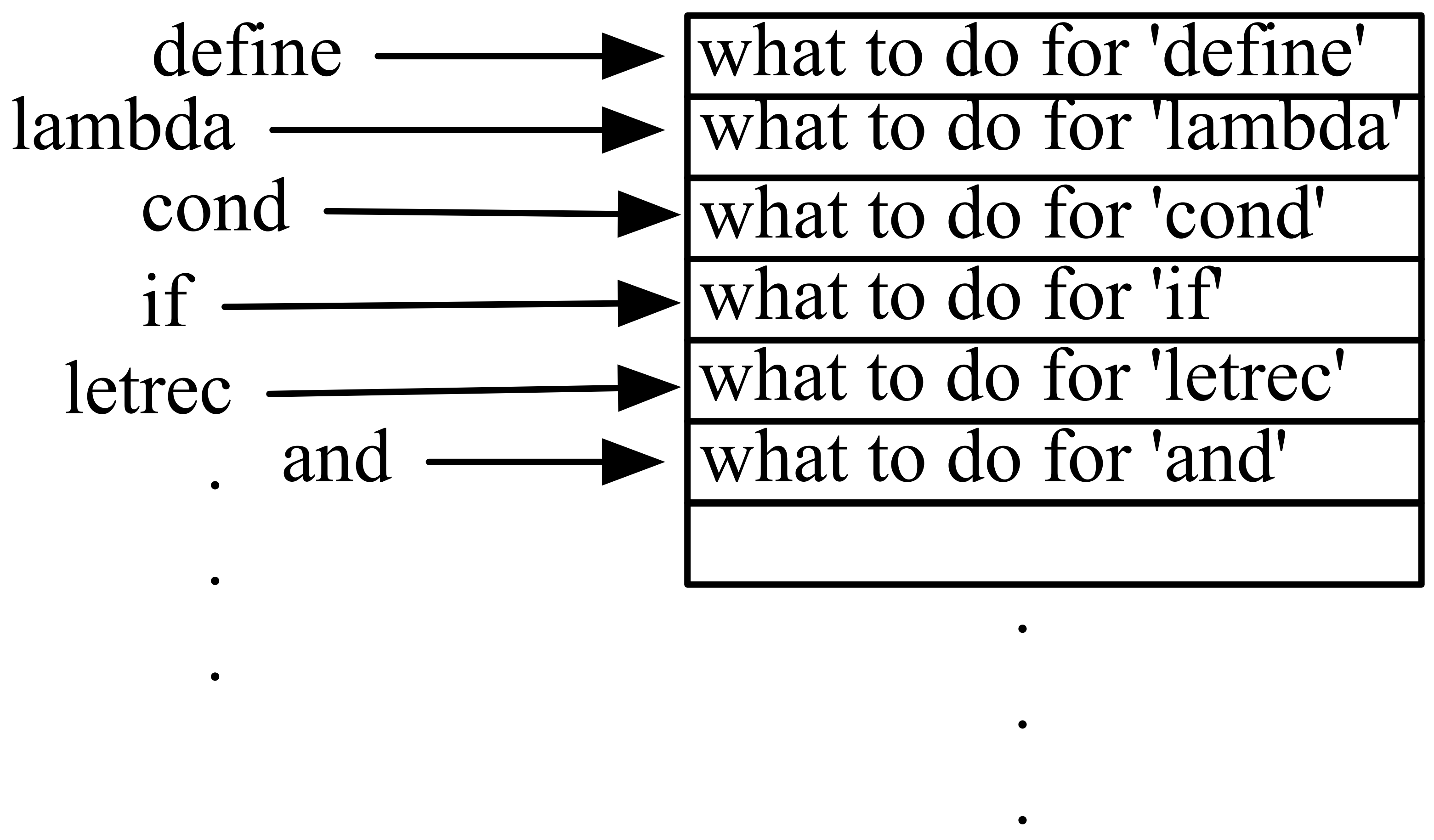
Verjaardagsparadox: in een klas met 23 personen is de kans 50% op een botsing van jarigen. Vanaf 60 personen 99%.

Het zoeken naar perfecte hashfuncties heeft alleen zin als we het aantal keys en de manier waarop ze eruit zien kennen.

Je kan maar beter met botsingen leren leven

Perfecte Hashfuncties : Voorbeeld

Om `(F a1 a2 ... an)` *snel te kunnen evalueren*



Een "symbol table" voor Scheme

Eigenschappen van “goede” hashfuncties

Een goede hashfunctie is simpel zodat je het funneling en clustering gedrag kan onderzoeken.

Een goede hashfunctie is snel zodat de hashfunctie niet de overhand krijgt boven de hashalgoritmiek.

Een goede hashfunctie is sterk wat wil zeggen dat elke locatie evenveel kans maakt.

Hashfuncties voor getallen volstaan

Bedoeling is een (mogelijk groot) geheel getal te verkrijgen voor een gegeven data waarde.

Bijvoorbeeld

`(to-number “bar”)`

=

$256^2 \times 98 + 256^1 \times 97 + 256^0 \times 114$

Daarna wordt de hashfunctie toegepast op dit getal.

We concentreren ons dus op hashfuncties die op gehele getallen werken.

Voorbeeld 1: Folding

$k = d_1d_2d_3d_4d_5$

$h(k) = d_1+d_2+d_3+d_4+d_5$

h(12345) = h(32451) = h(54321) = ...

Zeer populair maar geeft slechte resultaten.

Voorbeeld 2: Digit Selection

$k = d_1d_2d_3d_4d_5$

$h(k) = d_i+d_j+d_k$

Let op welke digits je selecteert. Doe een digit analyse op een groot stuk random data.

*Natte droom = "achieve avalanche". Elke gekozen digit met waarden [0..N] deelt **alle** sleutels in N groepen.*

Voorbeeld 3: Division

h(k) = k mod M

Zeer populair. Let op dat M priem is of een 2-macht is!

Nadeel 2-machten: je selecteert enkel de minst significante bits van de binaire voorstelling.

Nadeel priemgetallen: hersizen van de tabel wordt lastig omdat priemgetallen vinden lastig is.

Voorbeeld 4: Multiplication

h(k) = M.(kC mod 1) met $0 < C < 1$

Populaire C = 0.618034

Een goede C genereert de distributie.

De keuze van M wordt minder kritiek. 2-machten doen het dus goed.

Conclusie Hashing

Is nogal “artisanaal”

Zeer kleine wijzigingen in de hashfunctie kunnen grote gevolgen hebben voor funneling en clustering.

Goede hashfuncties vind je op het net. Maar ook zeer veel slechte. Let op!

Een Allerlaatste Inzicht

Alle niet-gehashte implementaties eisten een extra <<?

```
ADT dictionary< K V >

new
   ( ( K K → boolean ) → dictionary< K V > )
dictionary?
   ( any → boolean )
insert!
   ( dictionary< K V > K V → dictionary< K V > )
delete!
   ( dictionary< K V > K → dictionary< K V > )
find
   ( dictionary< K V > K → V ∪ {#f} )
empty?
   ( dictionary< K V > → boolean )
full?
   ( dictionary< K V > → boolean )
```

Bomen

Hashing

Geordende dictionaries ↔ Niet-geordende dictionaries

Hoofdstuk 7