

datastructuren en algoritmen

2011 09 15

college 4

totnutoe

- **lineaire datastructuren**
stacks, queues, vectoren, lijsten, rijtjes
- **hierarchische datastructuren**
bomen, binaire bomen, heaps
- **algoritmen**
sorteren, heap-sort

schema

- recap
- priority queues
- dictionaries
- hashing
- materiaal

schema

- recap
- **priority queues**
 - priority queue
 - sorteren met een priority queue
- dictionaries
- hashing
- materiaal

priority queue: eigenschappen

- belangrijkste item staat vooraan in de queue
- voorbeeld: rij bij de eerste-hulp

priority queue: setting

- we werken met **items** bestaande uit een **key** en een **element**
- totale ordening op de keys
- belangrijker naarmate de key kleiner is
- een key mag meer dan eens voorkomen
bijvoorbeeld zowel $(1, a)$ als $(1, b)$ in de verzameling items

priority queue: ADT

- insertItem(k, e)
- removeMin() (verwijder en return)
- minElement() (niet verwijderen)
- minKey() (niet verwijderen)
- size()
- isEmpty()

priority queue sorteer-schema

idee van het algoritme:

- 1 haal items uit de te sorteren rij
en stop ze met insertItem in de priority queue
- 2 haal ze met removeMin uit de priority queue
en stop ze een voor een achteraan in de rij

priority queue sort: algoritme

Algorithm PriorityQueueSort(A, C):

Input: List A , Comparator C

Output: List A sorted in ascending order

P = new priority queue with comparator C

while $\neg A.isEmpty()$ **do**

$e = A.remove(A.first())$

$P.insertItem(e, e)$

while $\neg P.isEmpty()$ **do**

$A.insertLast(P.removeMin())$

tijdscomplexiteit hangt af van implementatie PQ

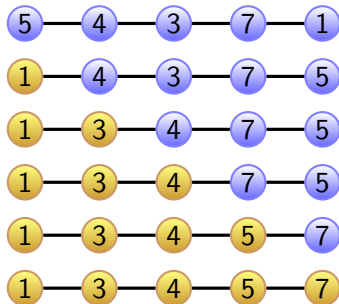
priority queue met niet-gesorteerde lijst



- in $\mathcal{O}(1)$: insertItem
- in $\mathcal{O}(n)$: removeMin, minKey, minElement
- voor PQ-sort:
 - stap 1 is makkelijk (constant)
 - stap 2 is moeilijk (lineair)
- PQ-sort in $\mathcal{O}(n^2)$
 - want $n + (n - 1) + \dots + 1 = \frac{n^2+n}{2}$
 - Gauss
- selection sort

sorteren: voorbeeld selection sort

- zoek een kleinste key in het rijtje en verwissel dat item met het eerste item
- itereer op de rest van het rijtje



Unsorted Part

Sorted Part

Minimal Element

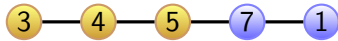
priority queue met gesorteerde lijst



- in $\mathcal{O}(1)$: removeMin, minKey, minElement
- in $\mathcal{O}(n)$: insertItem
(zowel array-based als singly/doubly linked list implementatie)
- voor PQ sort:
stap 1 is moeilijk (linear)
stap 2 is makkelijk (constant)
- PQ-sort in $\mathcal{O}(n^2)$
want $1 + 2 + \dots + n = \frac{n^2+n}{2}$
- insertion sort

insertion sort

- het rijtje bestaat uit een gesorteerd deel en een niet-gesorteerd deel
- neem het eerste item uit het niet-gesorteerde deel en voeg het op de juiste plek toe aan het gesorteerde deel



Unsorted Part

Sorted Part

Minimal Element

priority queue met een min-heap

- toevoegen en verwijderen alletwee in $\mathcal{O}(n \log(n))$
- heap-sort (bijna; want met max-heap)

schema

- recap
- priority queues
- dictionaries
- hashing
- materiaal

dictionaries: voorbeelden

database-achtige structuur waarin we 'elementen' zoeken via 'keys'

- namen en adressen, of namen en telefoonnummers
- credit card nummers en pincodes
- host-names en internet-adressen

dictionaries: eigenschappen

- een collectie van key-element items
- je kan er in zoeken met de key
- keys niet perse geordend

dictionary: eigenschappen

- nu: ongeordende dictionary
- hier: verschillende items kunnen dezelfde key hebben
- soms: key is hetzelfde als element
(voorbeeld: lijst namen)

dictionary: ADT

operaties:

- `findElement(k)`
retourneert het element behorend bij key k
retourneert `NoSuchKey` als geen item met key k
- `insertElement(k, e)`
voegt het item (k, e) toe
- `removeElement(k)`
verwijdert het item met key k en retourneert het element
retourneert `NoSuchKey` als zo'n item niet bestaat
- `size()`, `isEmpty()`, `keys`, `elements`

log file

ongeordende dictionary geïmplementeerd met ongesorteerde lijst

- in $\mathcal{O}(1)$: insertElement(k , e)
- in $\mathcal{O}(n)$: findElement(k), removeElement(k)
- handig alleen bij kleine database,
of: vaak toevoegen maar zelden zoeken of verwijderen (bijvoorbeeld access log)

schema

- recap
- priority queues
- dictionaries
- **hashing**
- materiaal

hashing: idee

- waar staat informatie e met key k ?
- in plaats van zoeken reken met de key uit waar de informatie staat
- we willen: operaties in $\mathcal{O}(1)$ verwachte tijdscomplexiteit
(soms is worst-case slechter)

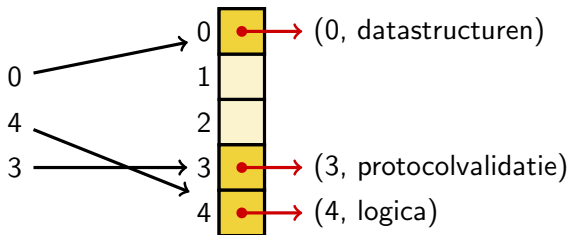
hashing: toepassingen

- geheugen adres
zie het geheugen adres van de key als integer (in Java)
- zie bits van de key als integer
voor korte keys
- in compilers

hashing: voorbeeld bucket array

- **keys:** integers in $[0, N - 1]$
- **elementen:** (in het voorbeeld: vakken)
- item $i = (k, e)$ wordt opgeslagen in bucket array ter lengte N :
 $A[k] := e$

voorbeeld: array ter lengte 5 met vakken als elementen

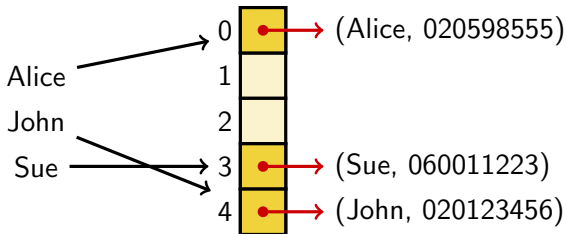


nadelen naieve aanpak met bucket array

- als lengte array N veel groter dan aantal gebruikte keys:
hoge ruimtecomplexiteit
- als keys geen integers zijn
dan werkt het niet
- daarom iets minder naief:
we gaan de indexen berekenen

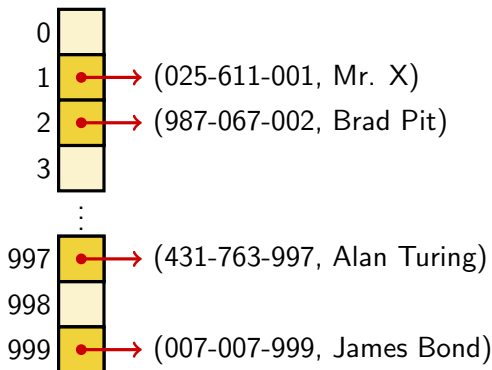
hashing: voorbeeld telefoonboek

- keys: namen
- hash functie berekent index met $h(w) = \text{length}(w) \bmod 5$
- elementen: telefoonnummers
- dus: item $i = (w, n)$ wordt opgeslagen in array: $A[h(w)] := n$ ('vaak' dezelfde index voor verschillende namen)



hashing: voorbeeld database

- **keys:** social security numbers
- **hash functie** berekent indexen:
 $h(\dots pqr) = pqr$ (laatste drie digits)
- **elementen:** namen
- dus: item $i = (\dots pqr, w)$ wordt opgeslagen in array: $A[pqr] := w$



hash table: algemeen

- bucket array ter lengte N
- hash functie $h : \text{keys} \rightarrow [0, N - 1]$
- item $i = (k, e)$ wordt opgeslagen in array: $A[h(k)] := e$
- gewenst: zoeken in $\mathcal{O}(1)$ ('verwacht')

hash functie in twee stappen

- van keys naar integers
(hash code)
- van integers naar integers in $[0, \dots, N - 1]$
(compression map)

hash functies: voorbeelden

- $h(k) = k$

- $h(k) = k \bmod N$

N doet ertoe: bekijk voorbeeld $N = 100$ en $N = 101$,
en voeg toe $0, 50, 100, 150, 200, \dots$

- $h(k) = a \cdot k + b \bmod N$
voorbeeld: $3k + 7 \bmod 101$

- ...

collisions

collision als $h(k) = h(k')$

oftewel: items met verschillende keys worden op dezelfde index gezet

collisions: wat combinatoriek

we hebben n items en een hash table ter grootte N

- er zijn N^n mogelijkheden voor de hash functie h
bij $n = 8$ en $N = 10$: 10^8 mogelijkheden
- er zijn $\frac{N!}{(N-n)!}$ mogelijkheden voor h zonder collision
bij $n = 8$ en $N = 10$ zijn dat er $3 \cdot 4 \cdot \dots \cdot 10$

collisions: wat kansen

- meestal: (veel) meer keys dan geheugenruimte dus collisions
- verjaardagsparadox:
bij 23 mensen is de kans dat iedereen een andere verjaardag heeft $< \frac{1}{2}$
- oftewel: bij $N = 365$ en $n = 23$ is de kans op collision $\geq \frac{1}{2}$

hash functies: wanneer vinden we ze goed?

- zoeken in $\mathcal{O}(1)$
- weinig collision
items worden netjes verdeeld; het lijkt random

Example (Hash Function for Strings in Python)

Python hash values modulo 997:

$h('a')$	$= 535$	$h('b')$	$= 80$	$h('c')$	$= 618$	$h('d')$	$= 163$
$h('ab')$	$= 354$	$h('ba')$	$= 979$...			

hash functies: wat maakt ze efficient?

- hash functie makkelijk te berekenen
- het type van de keys
- de distributie van de keys die echt gebruikt worden
- hoeveel ruimte er over is in de table
- wat doen we met collision

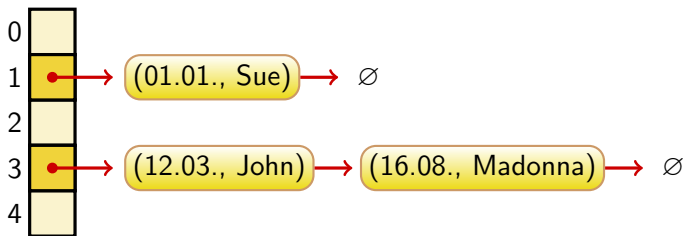
De *load factor* λ van een hash table is de ratio n/N

dus: aantal items gedeeld door de grootte van de table

load factor klein (liefst < 1) is beter

collision handling: chaining

stop in elk vakje niet een ding maar een rijtje
voorbeeld: hash functie is geboortemaand modulo 5



worst case: we gebruiken slechts 1 cel

collision handling:

collision handling: linear probing

als vol: probeer steeds de volgende cel, dus probeer

$$h(k) + 0$$

$$h(k) + 1$$

$$h(k) + 2$$

...

$$h(k) + N - 1$$

linear probing: zoeken is makkelijk

findElement(k):

$i = h(k)$

$p = 0$

while $p < N$ **do**

$c = A[i]$

if $c == \emptyset$ **then return** *NoSuchKey*

if $c.key == k$ **then return** $c.element$

$i = (i + 1) \bmod N$

$p = p + 1$

return *NoSuchKey*

linear probing: verwijderen is moeilijk

- de items 'erna' moeten opschuiven
- in plaats van opschuiven: markeren met 'available'
- van tijd tot tijd opschoonen

linear probing: toevoegen

begin bij $h(k)$, voeg toe bij eerste lege of available cel
(NB: zoeken gaat door bij available)

linear probing

- voornamelijk handig bij zelden verwijderen
- in plaats van +1 kun je op een andere manier een volgende cel zoeken

double hashing

gebruik twee hash functies h en h'
probeer dan

$$h(k) + 0 \cdot h'(k) \bmod N$$

$$h(k) + 1 \cdot h'(k) \bmod N$$

$$h(k) + 2 \cdot h'(k) \bmod N$$

\vdots

voorbeeld

$$h(k) = k \bmod N$$

$$h'(k) = 1 + (k \bmod (N - 2))$$

double hashing: voorbeeld

$$N = 13, h(k) = k \bmod 14, h'(k) = 7 - (k \bmod 7)$$

k	$h(k)$	$h'(k)$	probeer
18	5	3	5
41	2	1	2
22	9	6	9
44	5	5	5, 10
59	7	4	7
32	6	3	6
31	5	4	5,9,0
73	8	4	8

hashing: efficiency

- worst case: zoeken, toevoegen, verwijderen in $\mathcal{O}(n)$
- in de praktijk: erg snel zolang $n/N < 0.85$
verwachte running time: niet in $\mathcal{O}(n)$ maar in $\mathcal{O}(1)$
- gebruikt voor:
kleine databases, compilers, browser caches

universal hashing

voorlichting voor nieuwe studenten

- wil je voorlichting geven aan nieuwe studenten informatica of imm?
- 27 september training
- 8,5 euro per uur
- neem contact op met Judith Oost Lieveense:
j.f.oostlievense@vu.nl

schema

- recap
- priority queues
 - priority queue
 - sorteren met een priority queue
- dictionaries
- hashing
- **materiaal**

materiaal

- boek 2.4.1, 2.4.2
- boek 2.5

extra materiaal

- hier iets