INF2220 - Algoritmer og datastrukturer

HØSTEN 2007

Institutt for informatikk, Universitetet i Oslo

INF2220, forelesning 3:
Hashing

Hashing

- Hashtabeller (kapittel 5.1)
- Hash-funksjoner (kapittel 5.2)
- Kollisjonshåndtering
  - Åpen hashing (kapittel 5.3)
  - Lukket hashing (kapittel 5.4)
- Rehashing (kapittel 5.5)
- Utvidbar hashing (kapittel 5.6)

HASHING

Anta at en bilforhandler har 50 ulike modeller han ønsker å lagre data om.

Hvis hver modell har et entydig nummer mellom 0 og 49 kan vi enkelt lagre dataene i en array som er 50 lang.

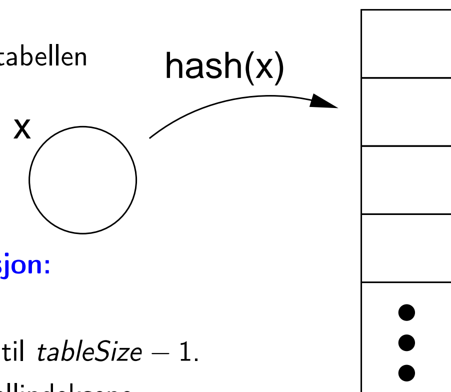
Hva hvis numrene ligger mellom 0 og 49 999?

- Array som er 50 000 lang:
  - sløsing med plass!
- Array som er 50 lang:
  - søking tar lineær tid...

Hashtabeller

- En hashtabell tilbyr innsetting, sletting og søking med **konstant** gjennomsnittstid.
- Men: operasjoner som finnMinste og skrivSortert har **ingen** garantier.
- Brukes gjerne når vi først og fremst ønsker et raskt svar på om et gitt element finnes i datastrukturen eller ikke.
- **Eksempler:**
  - kompilatorer: Er variabel y deklarert?
  - stavekontroller: Finnes ord x i ordlisten?
  - spill: Har jeg allerede vurdert denne stillingen via en annen trekkrekkefølge?

- Ideen i hashing er å lagre elementene i en array (hashtabell), og la verdien til elementet x (eller en del av x, kalt **nøkkelen** til x) bestemme plasseringen (indeksen) til x i hashtabellen.
- **tableSize** = størrelsen på hashtabellen
- **Hash-funksjon**: Fra nøkkelverdier til tabellindekser



Egenskaper til en god hash-funksjon:

- Rask å beregne
- Kan gi alle mulige verdier fra 0 til $tableSize - 1$.
- Gir en god fordeling utover tabellindeksene.

Idealsituasjonen

Perfekt situasjon:

- n elementer
- tabell med n plasser
- hash-funksjon slik at
  - den er lett (rask) å beregne
  - forskjellige nøkkelverdier gir forskjellige indekser

Eksempel:
Hvis modellene er nummerert

$$0,\ 1\,000,\ 2\,000,\ \ldots,\ 48\,000,\ 49\,000$$

kan data om modell i lagres på indeks $i/1\,000$ i en tabell som er 50 stor.

Problem: Hva hvis modell 4 000 får nytt nummer 3 999?

Hashtabell er en abstrakt datatype (ADT)

- Hashing er en **teknikk** for å implementere denne ADTen.
- Størrelsen `tableSize` er en del av ADTen!

Hovedproblemstillinger ved hashing

- Hvordan velge **hash-funksjon**?
  - ofte er nøklene strenger
- Hvordan håndtere **kollisjoner**?
- Hvor **stor** bør hashtabellen være?

I praksis

- Hash-funksjonen kan ikke mappe alle nøkler til ulike indekser.
- I stedet: hash-funksjonen bør gi en så *jevn fordeling* som mulig.

Hash-funksjoner

Eksempel

- Nøkler er heltall
- Begrenset antall tabellindekser

La hash-funksjonen være $hash(key) = key \mod tableSize$

Gir *jevn fordeling* for tilfeldige tall.

Hint:

- Pass på at ikke nøklene har spesielle egenskaper: Hvis $tableSize = 10$ og alle nøklene slutter på 0 vil alle elementene havne på samme indeks!

Huskeregel:

- La alltid tabellstørrelsen være et primtall.

Strenger som nøkler

Vanlig strategi: ta utgangspunkt i ascii/unicode-verdiene til hver bokstav og "gjør noe lurt".

Funksjon 1: Summer verdiene til hver bokstav.

```
public static int hash1(String key, int tableSize) {
    int hashVal = 0;

    for (int i = 0; i < key.length(); i++) {
        hashVal += key.charAt(i);
    }

    return (hashVal % tableSize);
}
```

- **Fordel**: Enkel å implementere og beregne.
- **Ulempe**: Dårlig fordeling hvis tabellstørrelsen er stor.

Funksjon 2: Bruk bare de tre første bokstavene og vekt disse.

```
public static int hash2(String key, int tableSize) {

    return (key.charAt(0) + 27 * key.charAt(1) +
               729 * key.charAt(2)) % tableSize;

}
```

- **Fordel**: Grei fordeling for tilfeldige strenger.
- **Ulempe**: Vanlig språk er ikke tilfeldig!

Funksjon 3: $\sum_{i=0}^{keySize-1} key[keySize - i - 1] \cdot 37^i$

```
public static int hash3(String key, int tableSize) {
    int hashVal = 0;

    for (int i = 0; i < key.length(); i++) {
        hashVal = 37 * hashVal + key.charAt(i);
    }

    hashVal = hashVal % tableSize;
    if (hashVal < 0) {
        hashVal += tableSize;
    return hashVal;
}
```

- **Fordel**: Enkel og relativt rask å beregne. Stort sett bra nok fordeling.
- **Ulempe**: Beregningen tar lang tid for lange nøkler.

Hash-funksjoner: oppsummering

- Må (i hvert fall teoretisk) kunne gi alle mulige verdier fra 0 til $tableSize - 1$.
- Må gi en god fordeling utover tabellindeksene.
- Tenk på hva slags data som skal brukes til nøkler.
  - Fødselsår kan gi god fordeling i persondatabaser, men ikke for en skoleklasse!

Eksempel: Ideell situasjon

Input:
0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Hash-funksjon: $hash(x, tableSize) = \sqrt{x}$

Hva hvis hash-funksjonen hadde vært

$hash(x, tableSize) = x \ mod \ tableSize$

istedenfor?

0	0	0	
1	1	1	
2	4	2	
3	9	3	
4	16	4	
5	25	5	
6	36	6	
7	49	7	
8	64	8	
9	81	9	

Kollisjonshåndtering

Hva gjør vi hvis to elementer hashes til den samme indeksen?

- Åpen hashing: Elementer med samme hashverdi samles i en liste (eller annen passende struktur).
- Lukket hashing: Dersom en indeks er opptatt, prøver vi en annen indeks inntil vi finner en som er ledig.

  Det finnes flere strategier for å velge *hvilken* annen indeks vi skal prøve.

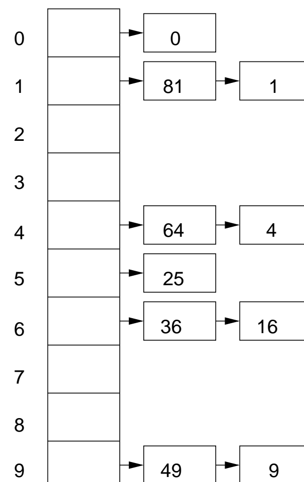
Åpen hashing ("Separate chaining")

Elementer med samme hashverdi samles i en liste (eller annen passende struktur)

Vi forventer at hash-funksjonen er god, slik at alle listene blir korte.

Vi definerer load-faktoren, λ, til en hashtabell til å være antall elementer i tabellen i forhold til tabellstørrelsen.

For åpen hashing ønsker vi $\lambda \approx 1.0$.



Lukket hashing — åpen adressering

Hvis hash-verdien allerede er brukt:

Prøver alternative indekser $h_0(x)$, $h_1(x)$, $h_2(x)$, ...
inntil vi finner en som er ledig.

h_i er gitt ved:

$$h_i(x) = (hash(x) + f(i)) \mod tableSize$$

og slik at $f(0) = 0$.

Merk at vi trenger en større tabell enn for åpen hashing
— generelt ønsker vi her $\lambda < 0.5$.

Skal se på tre mulige strategier (valg av f):

- Lineær prøving
- Kvadratisk prøving
- Dobbel hashing

Lineær prøving

Velger f til å være en **lineær** funksjon av i, typisk $f(i) = i$.

Input:
89, 18, 49, 58, 69

Hash-funksjon:
$hash(x, tableSize) = x \mod tableSize$

Kvadratisk prøving: La f være $f(i) = i^2$.

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Dobbel hashing

Bruker en *ny hash-funksjon* for å løse kollisjonene, typisk

$$f(i) = i \cdot hash_2(x),$$

med

$$hash_2(x) = R - (x \mod R)$$

hvor R er et primtall mindre enn *tableSize*.

Eksempel.

Input: 89, 18, 49, 58, 69

Andre hash-funksjon: $hash_2(x) = 7 - (x \mod 7)$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Rehashing

Hvis tabellen blir for full, begynner operasjonene å ta *veldig lang tid.*

Mulig løsning:

- Lag en ny hashtabell som er omtrent dobbelt så stor (men fortsatt primtall!).
- Gå gjennom hvert element i den opprinnelige tabellen, beregn den nye hash-verdien og sett inn på rett plass i den nye hashtabellen.

Dette er en dyr operasjon, $O(n)$, men opptrer relativt sjelden (må ha hatt $n/2$ innsettinger siden forrige rehashing).

Java's HashTable

Klassen `java.util.HashTable`:

- Implementerer en hashtabell som mapper nøkler til verdier.
- Bruker åpen hashing — ved kollisjon lagres verdiene i en bøtte (som blir gjennomsøkt sekvensielt)
- Default load faktor $\lambda < 0.75$, tradeoff tid/plass.
  - Når load faktor passeres, flyttes alle elemntene over i en større hashtabell ved rehashing

Når internminnet blir for lite

- En lese-/skriveoperasjon på en harddisk (aksesstid 7-12 millisekunder) tar nesten 1 000 000 ganger lenger tid enn tilsvarende operasjon i internminnet (aksesstid ca. 10 nanosekunder).
- Dramatisk forskjell — fanges dårlig opp av $\mathcal{O}$–modellen...
- Dersom *ikke* hele datastrukturen får plass i interminnet (hurtiglageret), lønner det seg nesten alltid å bruke datastrukturer som minimaliserer antall diskoperasjoner.

Utvidbar hashing

Anta at vi

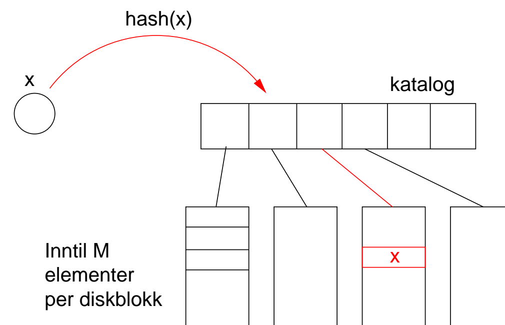
- skal lagre N dataelementer og at N varierer med tiden.
- kan lagre maksimalt M elementer i en diskblokk (= den delen av harddisken som kan hentes inn i *en* enkel leseoperasjon).

Problemet med å bruke vanlig hashing er at

- kollisjoner kan føre til at find(x) må undersøke mange diskblokker selv om hashfunksjonen distribuerer elementene godt.
- "rehashing" blir fryktelig kostbart.

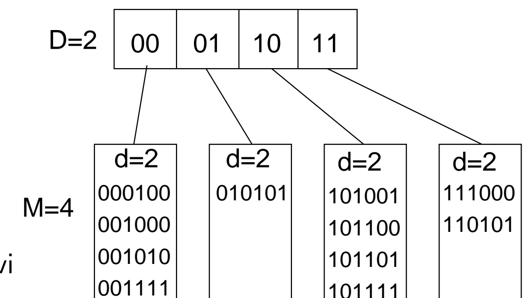
Løsning:

- Vi lar hashfunksjonen — via en katalog — angi hvilken diskblokk et element x befinner seg i (hvis det finnes).
- Dermed trenger find(x) bare to diskaksesser (og bare en aksess dersom katalogen kan lagres i internminnet).



I diskblokken lagrer vi for hvert element

- den binære strengen som koder hashverdien til elementet.
- "innmaten" i elementet eller en peker til denne.
- Vi benytter de *D* første bitene av hashverdien til oppslag i katalogen.
- I eksemplet til høyre forenkler vi litt og lagrer for hvert element bare hashverdien til elementet (som 6-sifret binærtall).



- Katalogen har 2^D indekser.
- Hver diskblokk har plass til M elementer.
- For hver diskblokk L lagrer vi et tall $d_L \leq D$.

Invariant:

Det garanteres at alle elementer i L har minst de d_L første bitene felles.

- Hvis $d_L = D$: akkurat én indeks i katalogen peker på diskblokk L.
- Hvis $d_L < D$: to eller flere indekser i katalogen peker på L.

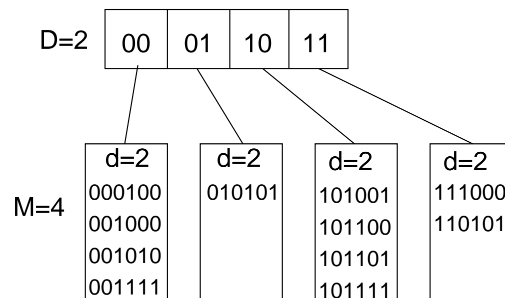
Innsettingsalgoritme

1. Beregn **hash(x)** og finn riktig diskblokk L ved å slå opp i katalogen på de D første sifrene i hashverdien.
2. Hvis det er færre enn M elementer i L, så sett x inn i L.
3. Hvis diskblokken derimot er full, så sammenlign d_L med D:
   - 3.1 Hvis $d_L < D$: "splitt" L i to blokker L_1 og L_2:
     - 3.1.1 Sett $d_{L_1} = d_{L_2} = d_L + 1$.
     - 3.1.2 Gå gjennom elementene i L og plasser dem i L_1 eller L_2 avhengig av verdien på de $d_L + 1$ første sifrene.
     - 3.1.3 Prøv på nytt å sette inn x (gå til punkt 2).
   - 3.2 Hvis $d_L = D$:
     - 3.2.1 Doble katalogstørrelsen ved å øke D med 1.
     - 3.2.2 Fortsett som ovenfor (splitt L i to blokker osv.).

Eksempel

$M = 4$ (maks. 4 elementer i en diskblokk)

Antall bits i hashverdiene = 6



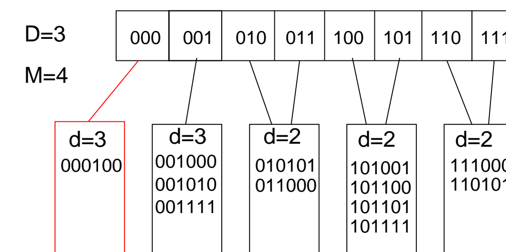
Sett inn element med hashverdi '011000'

Det er ledig plass i diskblokken, så vi setter rett inn.

Sett inn '000011'

Diskblokk '00' er full og $d_{00} = D$:

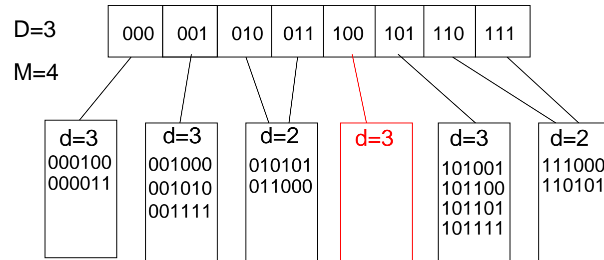
Vi utvider katalogen, splitter deretter diskblokken og flytter elementer til ny blokk. Dermed blir det plass.



De andre oppslagene i katalogen *deler* de andre blokkene parvis.

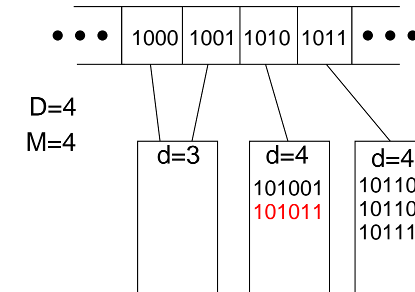
Sett inn '101011'

Diskblokk '101' er full og $d_{101} < D$, så vi splitter diskblokken:



Men: Vi får ikke flyttet noen elementer til den nye blokken!

Diskblokk '101' er fortsatt full, så vi må øke D til 4 og splitte en gang til.



Merk: Dupliserte hash-verdier
Algoritmen virker ikke hvis mer enn M elementer hasher til samme verdi.

Fyllingsgrad til diskblokkene

- Veldig stor katalog hvis elementer har mange ledende sifre felles.
- For bedre spredning bruker vi hashfunksjon (ikke nøkkelen til elementet direkte).
- Hvis vi antar *uniform fordeling av bitmønstrene*, blir antall blokker

$$\frac{N}{M}\log_2 e = \frac{N}{M}\cdot\frac{1}{\ln 2}$$

- Forventet antall elementer pr. diskblokk blir da:

$$\frac{N}{\frac{N}{M}\cdot\frac{1}{\ln 2}} = N\frac{M}{N}\cdot\frac{\ln 2}{1} = M\ln 2 = 0.693M$$

  dvs. at i det "lange løp" vil ca. 69% av hver diskblokk være fylt opp.
- Forventet størrelse på katalogen blir da

$$\frac{N^{1+\frac{1}{M}}}{M}$$

  dvs. at liten M gir stor katalog.

Hashing — Oppsummering

- Hashtabell gir innsetting, sletting, og søking med konstant gjennomsnittstid
- Hvor stor bør hash-tabellen være: *tableSize*
- Hash-funksjon: *jevn fordeling* over indeksene 0 til *tableSize* − 1.
- Kollisjoner: åpen vs. lukket hashing
- Utvidbar hashing: optimert for rehash-funksjonen