

# INF2220 - Algoritmer og datastrukturer

HØSTEN 2008

Institutt for informatikk, Universitetet i Oslo

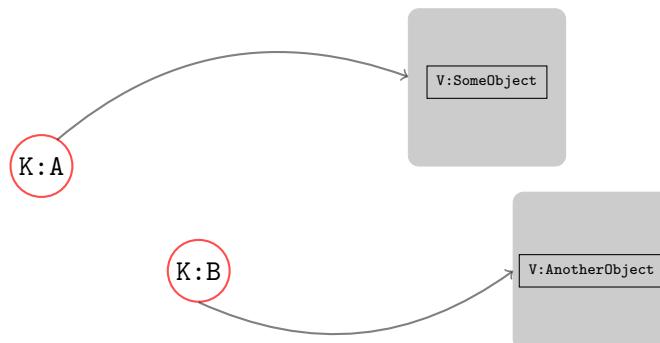
INF2220, forelesning 3:  
**Maps og Hashing**

## Maps og Hashing

- ▶ Map - Abstrakt Data Type (kapittel 9.1)
- ▶ Hash-funksjoner (kapittel 9.2.2)
- ▶ hashCode (kapittel 9.2.3)
- ▶ Kollisjonshåndtering (kapittel 9.2.5)
  - ▶ Åpen hashing
  - ▶ Lukket hashing (== Åpen addressering)
- ▶ Rehashing (kapittel 9.2.7)
- ▶ Kapittel 9.3 - 9.5 ikke pensum

## Map - ADT

- ▶ Ideen bak **interfacet Map<K, V>**



## Map vs Array

- ▶ Array - Nøkler er heltall
- ▶ Maps - Nøkler er objekter
- ▶ Array - Raske oppslag
- ▶ Maps - Vi må lage raske oppslag

# 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?

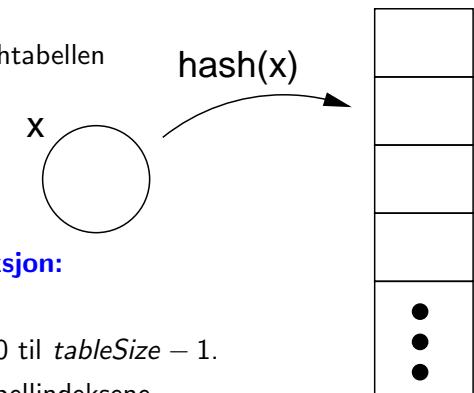
- ▶ Array som er 50 000 lang:
  - ▶ sløsing med plass!
- ▶ Array som er 50 lang:
  - ▶ søking tar lineær tid...

- ▶ 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 tabellindeks



## 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 indeks

### Eksempel:

Hvis modellene er nummerert

0, 1 000, 2 000, ..., 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?

## 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

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

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

## 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 nøklene til ulike indeks.
- ▶ I stedet: hash-funksjonen bør gi en så *jevn fordeling* som mulig.

## Når bruker vi Hashtabeller?

### Eksempler:

- ▶ **kompilatorer**: Er variabel y deklarert?
- ▶ **stavekontroller**: Finnes ord x i ordlisten?
- ▶ **database spørninger**: Nøkler mapper til kolonnenavn, verdi til innhold
- ▶ **XML parsing**: Nøkler blir attributtnavn, verdi blir innhold

Nesten alle scriptespråk har **hash** som del av språket.  
(Perl, Python, Ruby, PHP...)

```
myHash = {};  
myHash['key'] = "value";
```

## Hashtabeller

En hashtabell tilbyr:

- ▶ **innsetting**
- ▶ **sletting**
- ▶ **søking**

med **konstant** gjennomsnittstid.

Vi får raske svar på om et element er med i datastrukturen

## Hash-funksjoner

### Eksempel

- ▶ Nøkler er heltall
- ▶ Begrenset antall tabellindeks

La hash-funksjonen være  $\text{hash}(\text{key}) = \text{key} \bmod \text{tableSize}$

Gir *jevn fordeling* for tilfeldige tall.

### Hint:

- ▶ Pass på at ikke nøklene har *spesielle egenskaper*: Hvis  $\text{tableSize} = 10$  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!

**Generelt:** Bør være mange ganger  $tableSize$  før man gjør mod-operasjonen.

Ikke ta utgangspunkt i deler av nøklen som bare kan ta få av mulige verdier, for eksempel første siffer i datoens/månedsnr.

## hashCode

Alle objekter i Java utvider klassen `java.lang.Object`

- ▶ `Object` inneholder metoden: `int hashCode()`
- ▶ Trenger vi da tenke på implementasjon av hash funksjoner?
- ▶ `hashCode()` returnerer typisk minneadresse konvertert til `int`
- ▶ Objekter vi anser for like må hashe til samme verdi
- ▶ De vil aldri ha samme minneadresse
- ▶ Vi må også re-implementere `equals` i klassen vi bruker som nøkkel

## 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.

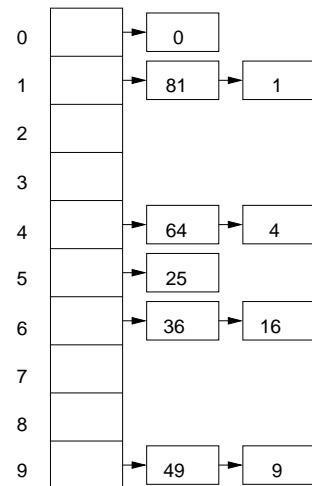
## Åpen hashing ("Separate chaining")

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,  $\lambda$ , 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  $h_0(x), h_1(x), h_2(x), \dots$  inntil vi finner en som er ledig.

$h_i$  er gitt ved:

$$h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{tableSize}$$

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:

$\text{hash}(x, \text{tableSize}) = x \bmod \text{tableSize}$

**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

$$f(i) = i \cdot \text{hash}_2(x),$$

med

$$\text{hash}_2(x) = R - (x \bmod R)$$

hvor  $R$  er et primtall mindre enn  $\text{tableSize}$ .

### Eksempel.

Input: 89, 18, 49, 58, 69

Andre hash-funksjon:  $\text{hash}_2(x) = 7 - (x \bmod 7)$

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 omrent 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 sjeldent (må ha hatt  $n/2$  innsetninger 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 elementene over i en større hashtabell ved rehashing

## Hashing — Oppsummering

- ▶ Hashtabell gir **insetting**, **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