

Hashing og hashtabeller

Teknik til at repræsentere mængder

- Konstant tid for finde og indsætte
- men ingen “sortering” af elementerne

Specielt udviklet til

- Meget stort “udfaldsrum”
- ... hvor faktiske mængder er *relativt* små i forhold til udfaldsrum

Standardeksempel nr. 1:

Udfaldsrum = mængden af alle mulige navne på variable, klasser ...

Faktisk mængde = mængden af alle navne i et bestemt Javaprogram

Dagens program

- Indledningsøvelse: repræsentation af mængder over små udfaldsrum
- Princippet i Hashing: Om hashingfunktioner
- “Kollisioner”, dvs. to forskellige objekter “hasher” til samme tal
- Løsning på problemet 0: Hægtede lister
- Løsning på problemet 1: Lineær afprøvning
- Løsning på problemet 2: Kvadratisk afprøvning
- Kort om implementation i Javas API
 - Med en lille afstikker om at heltal i Java er skøre

“Hashing” ???

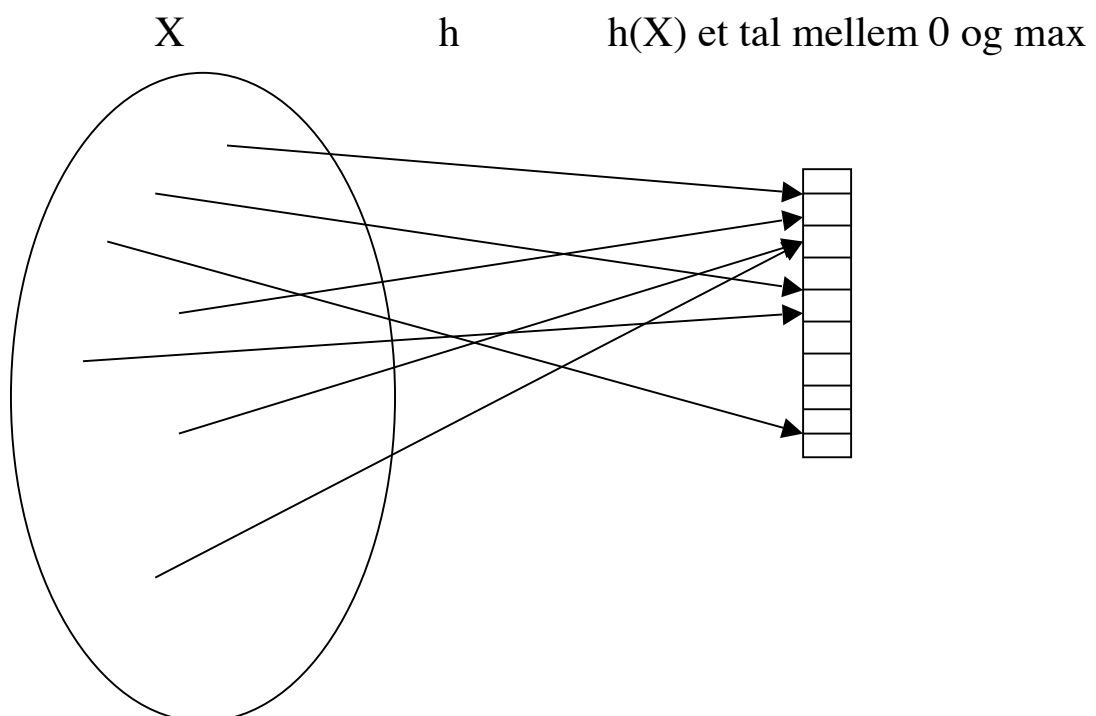
Ordbogsopslag:

hash v.t. (VP 1, 10 with up) chop or cut up (meat) into small pieces. **n.** 1. [U] (dish of) cooked meat, ~ed and re-cooked. 2. (fig.) *make a ~ of sth.*, do it very badly, make a mess of it. *settle sb.’s ~*, deal with him in such a way that he gives no more trouble.

Teknik til repræsentation af mængder over

- Meget stort “udfaldsrum”
- ... hvor faktiske mængder er *relativt* små i forhold til udfaldsrum

Ideen: at afbilde stort adresserum over i et lille:



Indledende øvelse: Repræsentation af mængder over små udfaldsrum

Pascal (*Wirth, ca. 1974*)

```
type fisk = (torsk, sild, rødspætte, kuller, kulmule, fjæsing, gråsej,  
            pighvar, slethvar, laks, ørred, tobis);  
var nogle_fisk, flere_fisk, endnu_flere_fisk : set of fisk;  
nogle_fisk := [torsk..kuller, tobis, slethvar];  
flere_fisk := [torsk, rødspætte..fjæsing, laks, slethvar];  
endnu_flere_fisk := nogle_fisk + flere_fisk; // foreningsmgd nogle_fisk U flere_fisk
```

Compiler tildeler hvert symbol et nummer

```
torsk -> 0  
sild -> 1  
...  
tobis -> 11
```

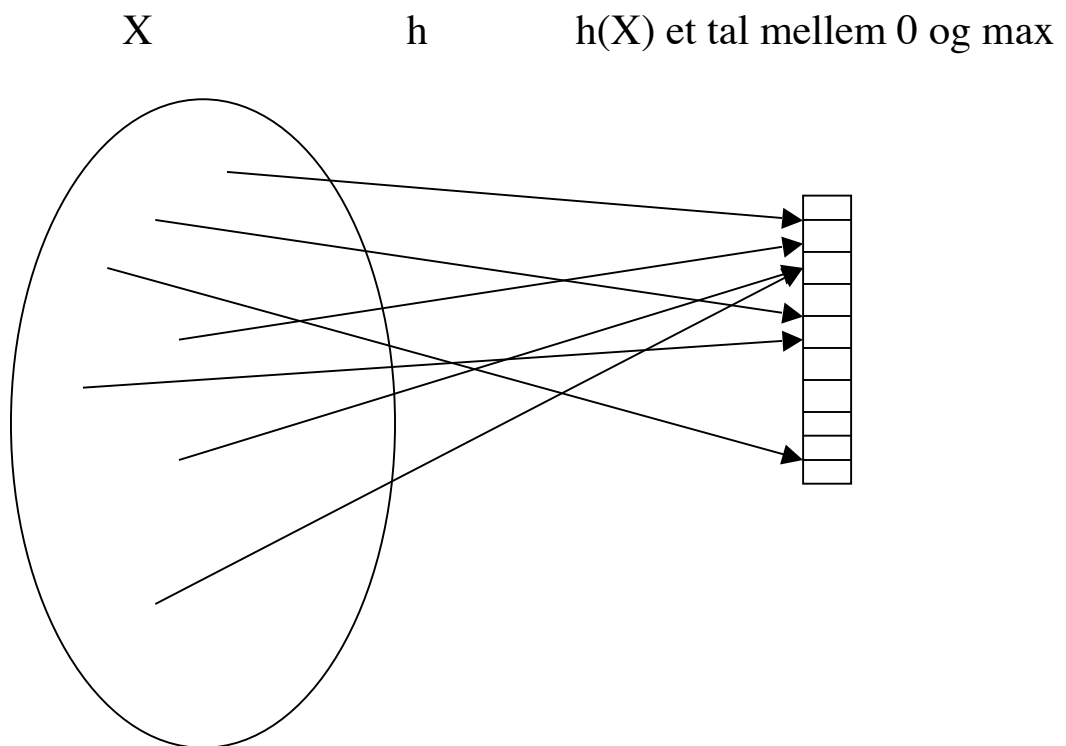
En mængde er en bitvektor, f.eks.

```
nogle_fisk      = 111100001001  
flere_fisk      = 101111000100  
endnu_flere_fisk = 111111001101 “+” eller “U” implementeret ved  
bitvis logisk “eller”
```

OBS: Java understøtter bitvise operationer, så princippet kan eftergøres — men man må selv lege compiler!

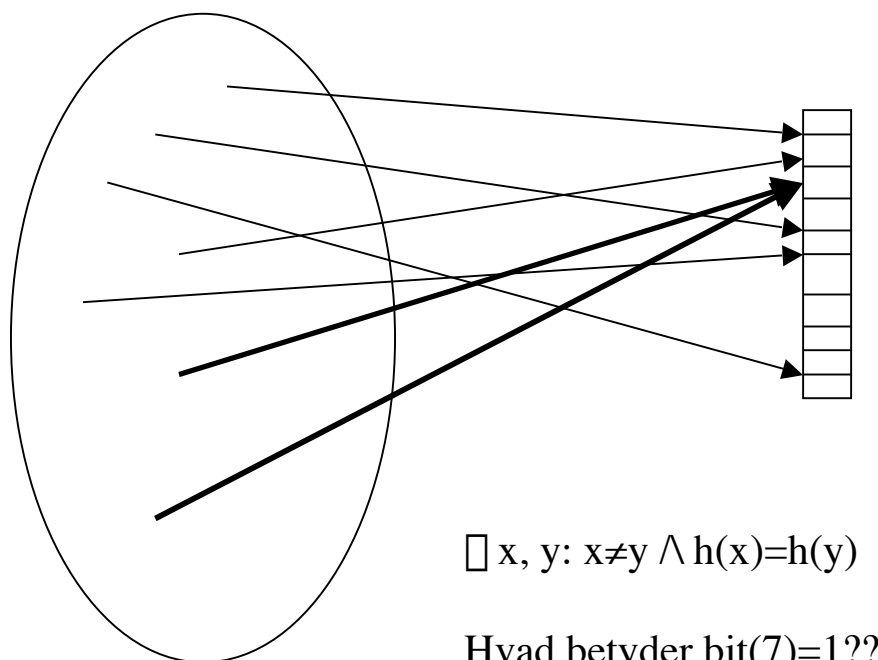
```
Forslag: static final int torsk = 1;  
            static final int sild = 2;  
            static final int rødspætte = 4;  
            static final int kuller = 8;  
  
int nogle_fisk = torsk+sild+rødspætte+kuller+tobis+slethvar;  
  
endnu_flere_fisk = nogle_fisk | flere_fisk; //bitvis “eller”
```

Princippet i Hashing: Komprimere adresserum vha hashfunktion



Og bruge princippet fra bitvektoren: 1 \approx jeg er med; 0 \approx jeg er ikke med.

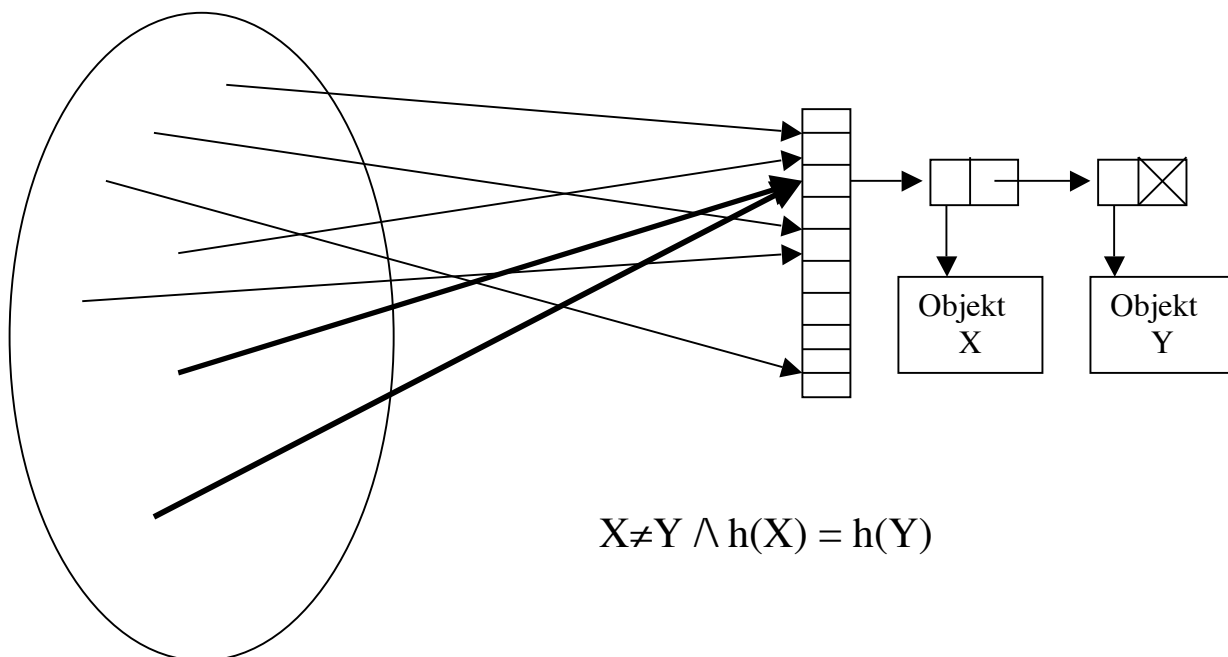
Problem: Kollision, når to elementer hasher til samme værdi



$$\exists x, y: x \neq y \wedge h(x) = h(y)$$

Hvad betyder bit(7)=1??

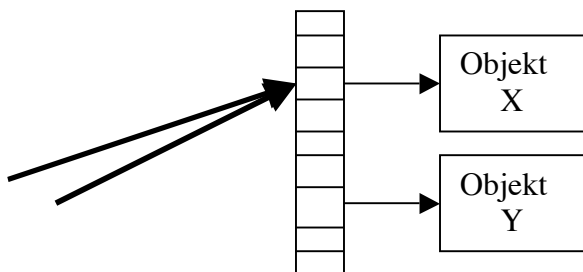
En mulig løsning på problemet, hæftet liste af objekter:



Nu ikke bitvektor men array 0..max af hægtede lister af objekter (dvs. repræsentation af objekter i den store mængde (udfaldsrummet)).

Ikke populær da det giver ekstra arbejde med at forfølge referencer hver gang, selv når tabellen er stort set tom!

De sædvanlige løsninger på problemet: Hvis pladsen er optaget så find en anden plads i tabellen (efter fast algoritme):



Af finde Y i tabellen:

Hvis tabel(h(Y)) tom, så er Y ikke med;

Hvis tabel(h(Y)) refererer til objekt Y så OK;

Hvis tabel(h(Y)) refererer til objekt X ($\neq Y$) så prøv andet steds...;

Kunsten at finde en anden plads i tabellen

Den oplagte metode: "Linear probing"

Prøv næste ubrugte celle:

Antag

$$h(X) = h(Y) = 117$$

$$h(Z) = 118$$

$$h(\text{Æ}) = 117$$


117: X

118:

119:

120:

...

117: X 


118: Y

119:

120:

...


117: X


118: Y 


119: Z

120:

...

117: X 

118: Y 

119: Z 

120: Æ 

...

Fordele: Virker altid indtil tabellen er helt fuld

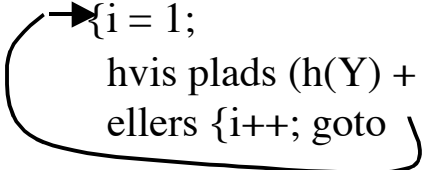
Ulemper: Der dannes klumper.

Dvs. for mange indsætte og finde kolliderer flere gange!

(Kræver et matematisk argument, som vi springer over)

Den mest udbredte strategi: “Quadratic probing”

hvis plads $h(Y)$ er ledig placér Y der;

ellers 

```
{ i = 1;
  hvis plads  $(h(Y) + i^2) \% \text{tabelstørrelse}$  ledig, så placér  $Y$  der;
  ellers { i++; goto } ; }
```

Hvad får vi ud af det? En matematiker kan bevise:

- Klumpningsgraden mindre
- Hvis tabelstørrelse er et primtal, så besøges halvdelen af alle celler i værste fald
- Dvs. fyldningsgrad på 50% eller derover er ikke godt!

Tag det som et kuriøst faktum; ikke eksamenspensum at redegøre for det!!

NB: Bogen gør et nummer ud af at **optimere beregningen** så multiplikation og modulo operation undgås

$$\begin{array}{l} 0^2 = 0 \\ 1^2 = 0 + 1 = 1 \\ 2^2 = \blacktriangleleft + 3 = 4 \\ 3^2 = \blacktriangleleft + 5 = 9 \\ 4^2 = \blacktriangleleft + 7 = 16 \\ \dots \end{array}$$

dvs. nu følgende kode:

```
currentPos += 2 * ++collisionNum - 1;
if (currentPos >= array.length) currentPos -= array.length;
```

OBS-OBS-OBS:

- En god compiler oversætter “2 *” til højreskift
- En god compiler oversætter “-1” til én maskininstruktion
DEC <register>

Om hashfunktionen

Faktum vi må leve med: Kollisioner kan ikke undgås

Krav til god hashfunktion

- skal sprede sig så godt som muligt over hele intervallet
 - ramme alle celler
 - og ca. lige meget
 - og det for en *typisk* mængde af objekter.
- skal kunne beregnes effektivt

Eksempel på dårlig hashfunktion; tableSize = to-talspotens

```
int hash(String key, int tableSize) {  
    int hashVal = 0;  
    for (int i = 0; i < key.length(); i++)  
        hashVal = (hashVal * 256 + key.charAt(i)) % tableSize;  
    return hashVal}
```

Læs i bogen i afsnit 20.2 om at primtal er gode, men der er meget meget mere i historien om, hvad der gør en hashfunktion god!

For at forstå det, skal vi lige have en foredrag om heltal i Java — som er uden kontrol for overflow!!!

Hvad er overflow?

Eksempel: Regning med tal op til 4 cifre

$$\begin{array}{r} 0020 \\ + 0099 \\ \hline 0119 \end{array} \qquad \begin{array}{r} 0500 \\ + 0499 \\ \hline 9999 \end{array}$$

$$\begin{array}{r} 0502 \\ + 0503 \\ \hline ???? \end{array}$$

En elektronisk dims kunne måske vise give

$$\begin{array}{r} 0502 \\ + 0503 \\ \hline EEEE \end{array}$$

Svarer til OVERFLOW kontrol

En dum mekaniske dims med fire hjul med hver 10 cifre vil sandsynligvis vise

$$\begin{array}{r} 0502 \\ + 0503 \\ \hline \text{⊕} 0005 \end{array}$$

Uden OVERFLOW kontrol

Eksempel med 4-bits aritmetik hvor bit længst til venstre er fortegn

f.eks. 0100 $\approx 4_{10}$
 1001 $\approx 1_{10}$

1)	2)
0010	0111
+ 0011	+ 0111
=====	=====
0101	0 1110

Med overflowkontrol: Eksempel 2 giver anledning til fejlmeddelelse
Uden overflowkontrol: Eksempel 2 regner forkert:

$$7 + 7 = -6 \quad \text{!!!! (ralle)}$$

Java's heltal opfører sig ca. på den måde:

```
public final class OverFlowTest
{ public static void main( String [ ] args )
  { int n = 0;int i;
    n = 2147483647; // maximal int
    System.out.println("n " + n);
    n = n+17;
    System.out.println("n " + n);
    for(i=0; i< 2147483647; i=i+100000) {n=n+1;}
    System.out.println("SLUT); } }
```

n 2147483647

n -2147483632

^C Programmet gik i løkke pga. overflow og manglende overflowcheck

Om hashing i Java API

Standardklasser for

HashSet ~ svarende til de hashede mængder vi har set

HashMap ~ afbildning repræsenteret som hashede mængder,
hvor vi for en objekt X, i tabellen finder ref. til objekt
[fra X, til Værdi≠]

Bemærk at den generelle klasse Object har metode hashCode() som returnerer bare-et-eller-andet-heltal.

For at kunne bruge HashMap og HashSet på sine egne objekter må man også huske at skrive en hashCode() metode

Eksempel på indbygget hashCode() for tekststreng:

```
public final class HaCo
{ public static void printHash(String s)
  {System.out.println(s + ": " + s.hashCode());}

public static void main( String [ ] args )
  { printHash("a");
    printHash("b");
    printHash("c");
    printHash("aa");
    printHash("ab");
    printHash("ac");
    printHash("ba");
    printHash("Der var en gang en mand");
    printHash("Han boede i en spand");
    printHash("Hsdjazzd noperun fgkdfgo krpot, krpot, poylgkrs okerpil mkdfg!");
  }}
```

a: 97

b: 98

c: 99

aa: 3104

ab: 3105

ac: 3106

ba: 3135

Der var en gang en mand: -904611645

Han boede i en spand: -1964416372

Hsdjazzd noperun fgkdfgo krpot, krpot, poylgkrs okerpil mkdfg!: -505073700

Både positive og negative tal returneres

Dvs. implementation af hashtabel skal teste ≥ 0 og tage modulo (%) tabelstørrelse

Implementation af Java.util's HashSet og HashSet:

Baseret på

- “quadratic probing” med primtal som tabellængde
- “Re-hashing”, hvis tabel bliver halvt fuld, så fordobles tabellen (rundet op til næste primtal)

Sletning foretages ved at indsætte en boolean i hver celle

active = true ~ som vi forventer

active = false ~ pågældende felt har været indsat,

men slettet igen (og skal ignoreres)

Nødvendig pga. “probing” da ægte sletning for bøvlet!

Ellers ingen overraskelser, kig på koden i afsnit 20.4.1.

~ Tak for Deres opmærksomhed ~