Hashing

The term "hash" comes by way of analogy with its non-technical meaning, to "chop and mix". Indeed, typical hash functions, like the mod operation, "chop" the input domain into many sub-domains that get "mixed" into the output range to improve the uniformity of the key distribution.

```
public interface Collection<V> {

    public default boolean contains(V value) {
     return false;
    }

    public void add(V value);

    public default V remove(V value) {
     return null;
    }

}
```

```
public interface Map<K, V> {

    public default V get(K key) {
     return null;
    }

    public void put(K key, V value);

    public default V remove(K key) {
     return null;
    }

}
```

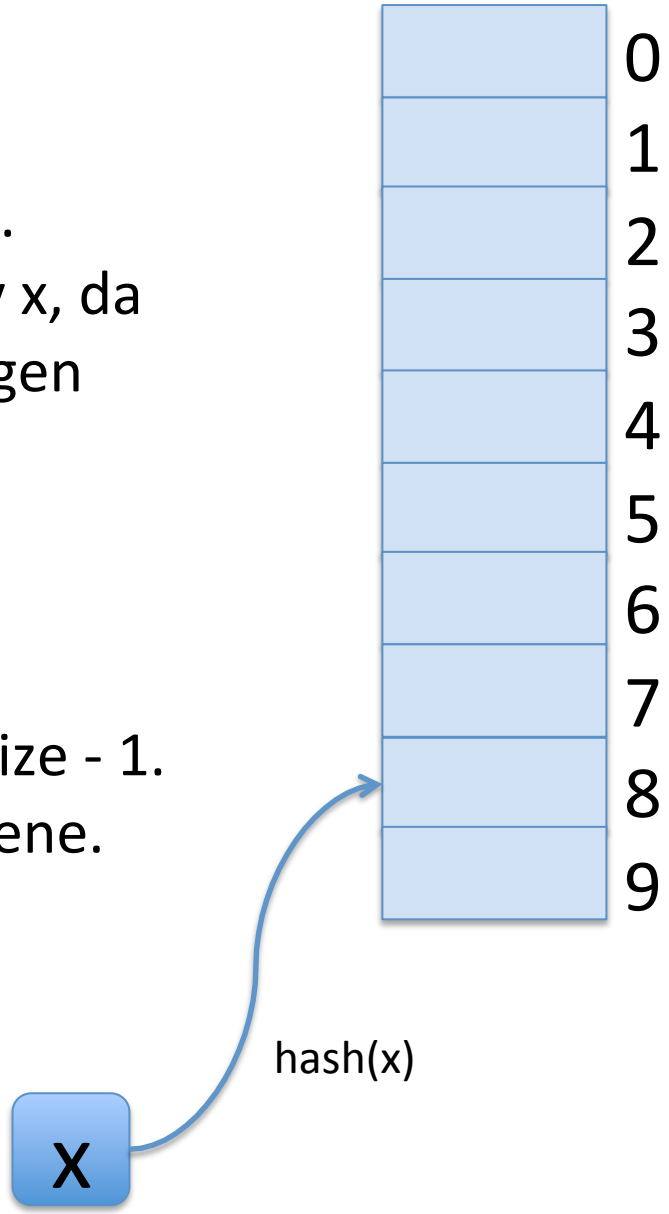
Hashtabeller

Ideen i hashing er å

- lagre elementene i en array (hashtabell).
- la verdien til elementet x (eller en del av x, da kalt nøkkelen til x), bestemme plasseringen (indeksen) til x i hashtabellen.

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.

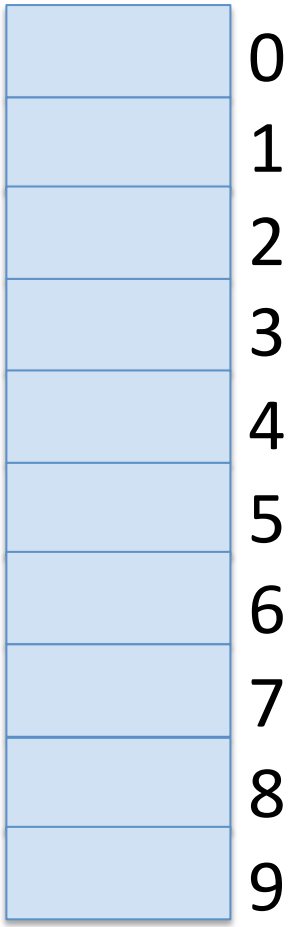


Idealsituasjonen

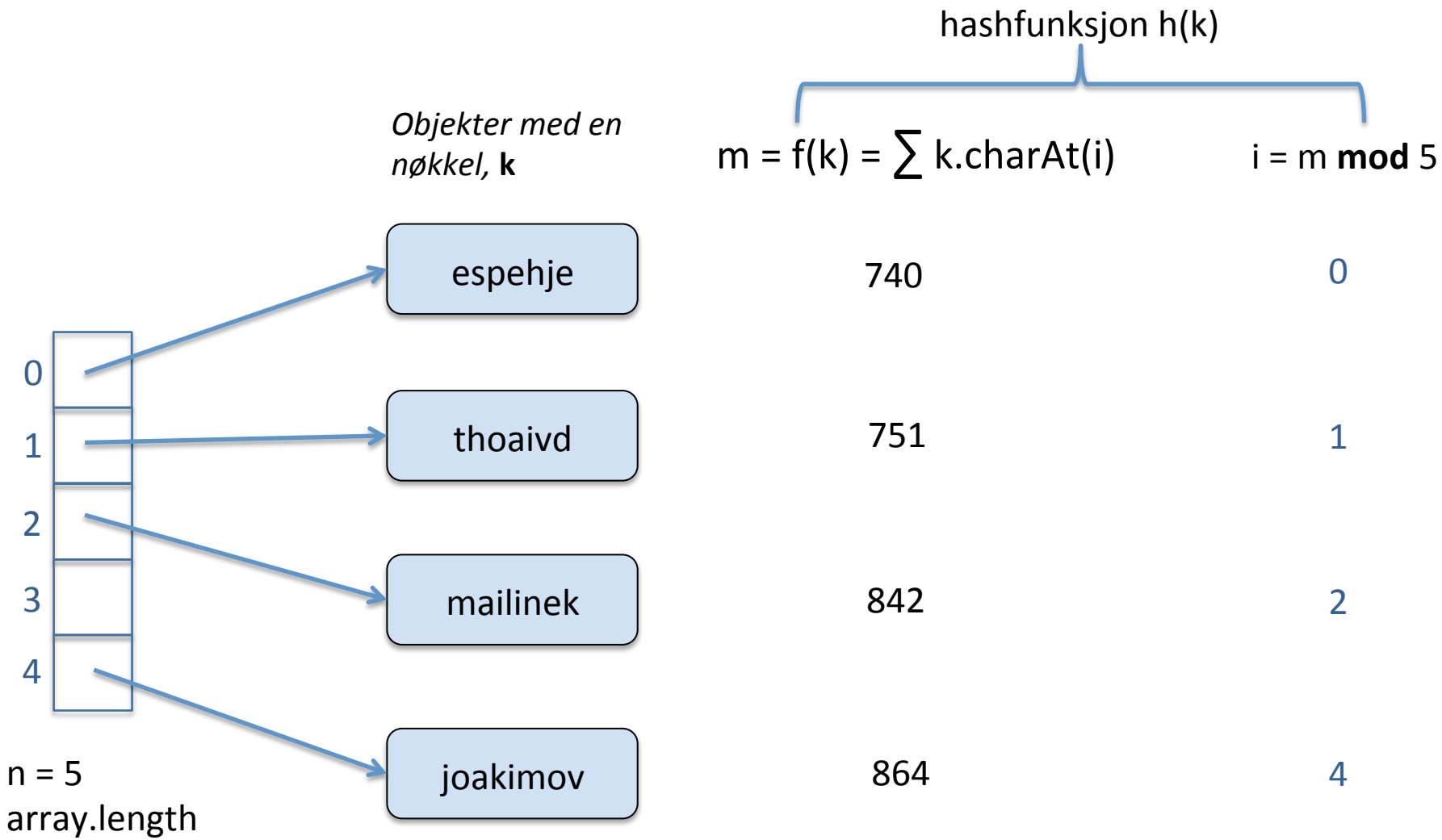
- antall elementer = tableSize
- hashfunksjon som gir forskjellige indekser for alle elementene

For eksempel 10 elementer med verdiene
0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Gå sammen to og to og finn den perfekte hashfunksjonen for denne situasjonen.



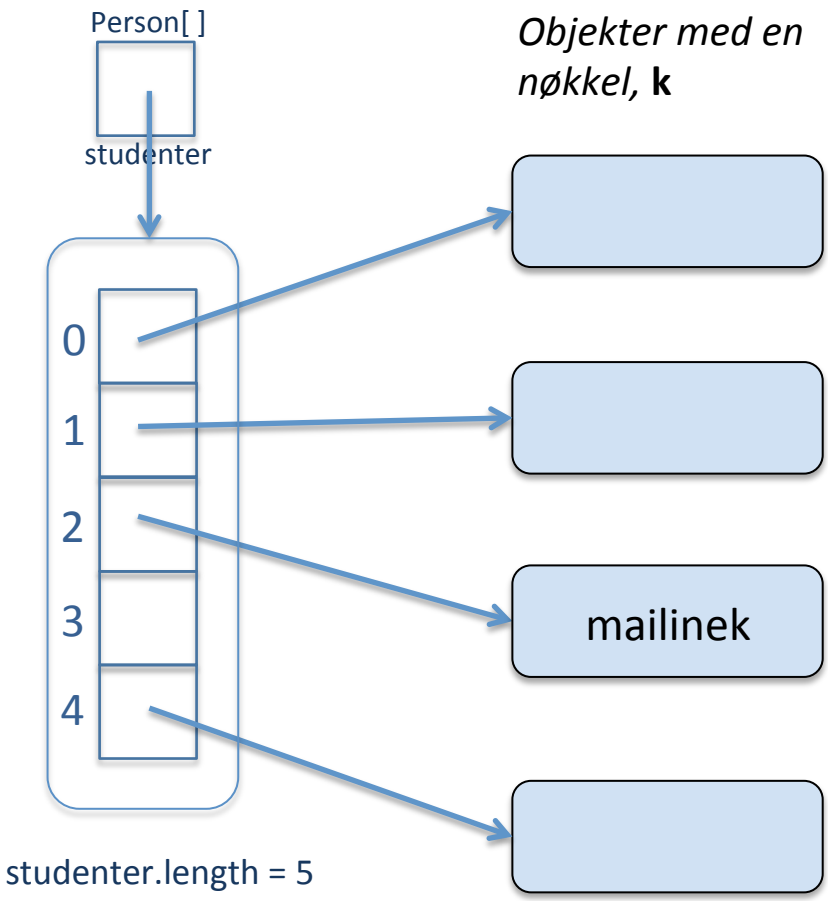
X


hashfunksjon h(k)
Objekter med en nøkkel, **k**
m = f(k) = ∑ k.charAt(i)
i = m **mod** 5
espehje
740
0
thoaivd
751
1
mailinek
842
2
joakimov
864
4
0
1
2
3
4
n = 5
array.length

hashfunksjon h(k)

$$h(k) = \left(\sum k.charAt(i) \right) \textbf{ mod } tableSize$$

```
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);
    }
```

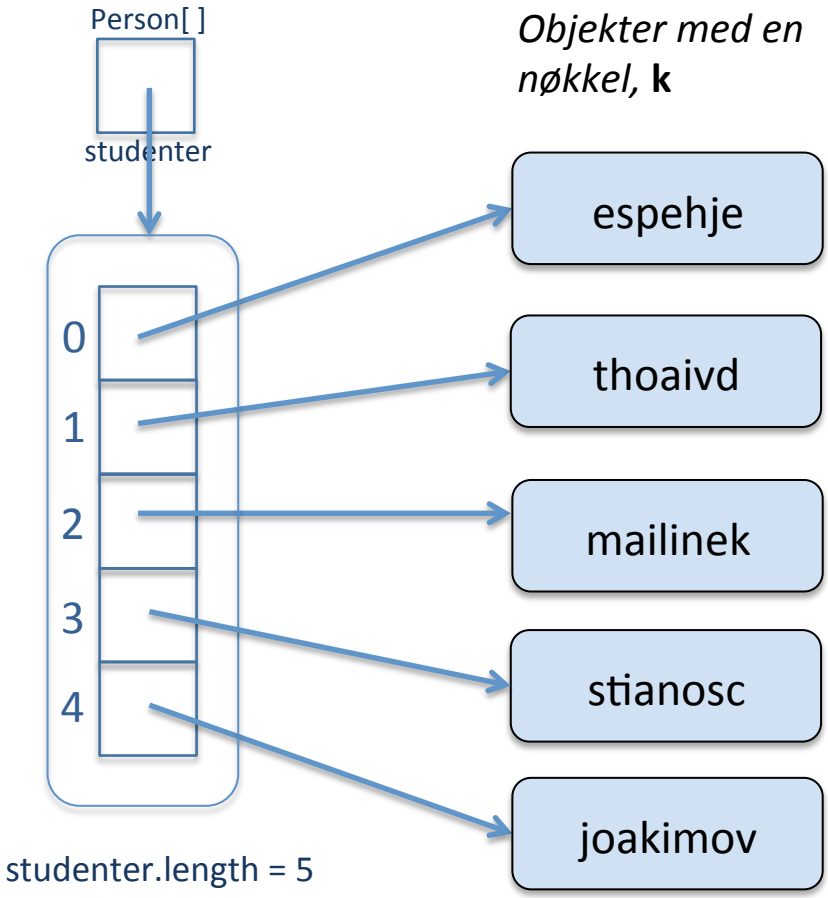


Oppgave:

Hvordan finne ut om et objekt med nøkkel “malinek” finnes i arrayen eller ikke?

Vi bruker den samme hashfunksjonen som vi brukte, og finner at når k = “malinek”, så er $\sum$ k.charAt(i) = 737. 737 mod 5 = 2.

Så «slår vi opp» i arrayen på indeks 2:

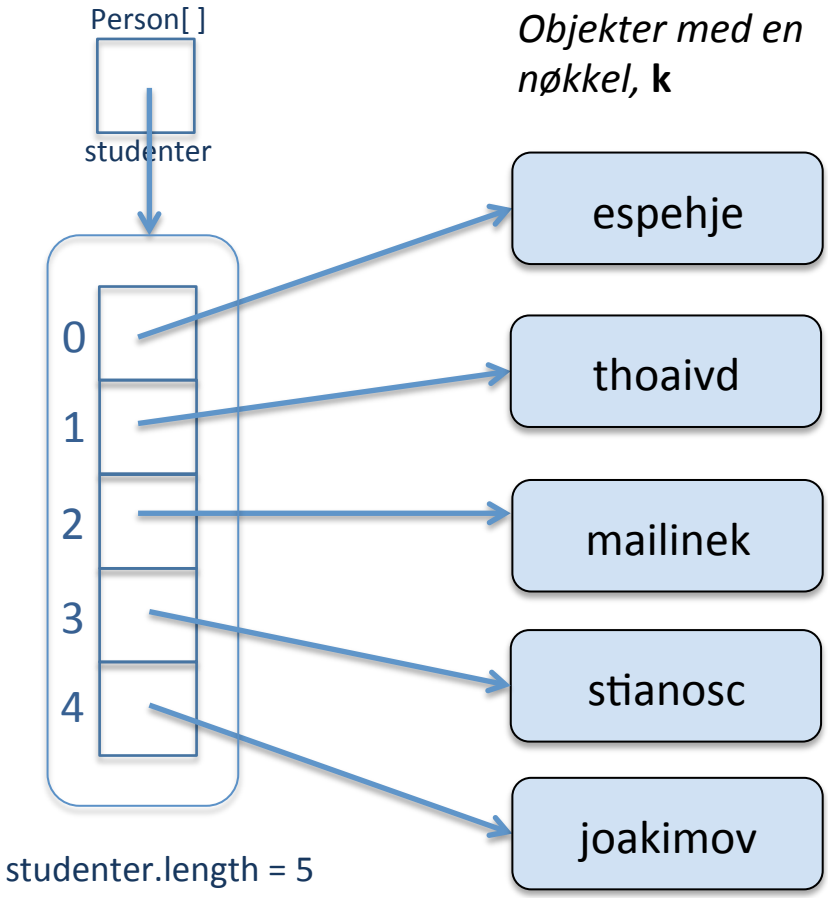


Oppgave:

Sett inn et Person-objekt med k = “stianosc”.

$\sum$ k.charAt(i) = 868

868 mod 5 = 3



Oppgave:

Sett inn et Person-objekt med k = “philipnh”.

1. Bestemme (tilordne) nøkkel. Denne må være unik slik at to forskjellige elementer ikke kan ha/få samme nøkkel.

2. Bestemme (initiell) tabellengde.

3. Bestemme hashfunksjon(er).

4. Hvordan håndtere kollisjoner?

Når bruker vi hashtabeller?

Brukes gjerne når vi har ganske mange elementer og 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?
- XML parsering: Nøkler blir attributtnavn, verdi blir innhold.
- Rutere: Bestemme hvilken linje en datapakke skal sendes til (IP-adresse er nøkkel).

Nesten alle scriptspråk has hash som del av språket. (Perl, Python, Ruby, PHP, …)

1. Map is an interface in Java and its concrete implementations are called HashMaps and Hashtables, among other

2. Python - Dictionaries

3. C++ - hash_map, unordered_map and so on are different implementations

Hashtabeller

Ideen i hashing er å

- lagre elementene i en array (hashtabell).
- la verdien til elementet x (eller en del av x, da kalt nøkkelen til x), bestemme plasseringen (indeksen) til x i hashtabellen.

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

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

Eksempel:

Hvis objektene har nummer (nøkkel)

0, 1 000, 2 000, …, 48 000, 49 000

kan objektet lagres på indeks i/1000 i en tabell som er 50 stor.

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

Tidsforbruk

En hashtabell tilbyr

- innsetting
- sletting
- søking (gjenfinning)

med ***konstant*** gjennomsnittstid.

Men operasjoner som *finnMinste* og *skrivSortert* har ***ingen garantier.***

Hash-funksjoner

Eksempel:

- Heltall som nøkler
- Begrenset antall tabellindekser
- La hash-funksjonen være

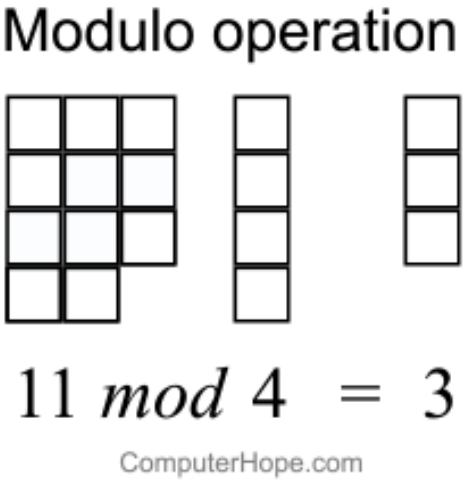
  hash(x,tableSize) = x mod tableSize

  Dette gir jevn fordeling for tilfeldige tall.

Pass på at nøklene ikke 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.

Modulo-operatoren



Strenger som nøkler

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

Strenger som nøkler

Funksjon 1: **Summer verdiene** til hver bokstav.

```
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.

Strenger som nøkler

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

h(k) = (k.charAt(0) + k.charAt(1)*27 + k.charAt(2)*729) **mod** tableSize

```
static int hash2(String key, int tableSize) {
    int hashVal = key.charAt(0) +
        27 * key.charAt(1) +
        729* key.charAt(2);
    return (hashVal % tableSize);
}
```

Fordel: Grei fordeling for tilfeldige strenger.

Ulempe: Vanlig språk er ikke tilfeldig!

Strenger som nøkler

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

```
static int hash3(String key, int tableSize) {
    int hashVal = 0;
    for (int i = 0; i < key.length(); i++) {
        hashVal = 37*hashVal + key.charAt(i);
    }
    return Math.abs(hashVal % tableSize);
}
```

Fordel: Enkel og relativt rask å beregne. Stort sett bra nok fordeling.

Ulempe: Vanlig språk er ikke tilfeldig!

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 modulo-operasjonen.

hashCode i Java

int hashCode()

Returns a hash code value for the object

Typisk en minneadresse konvertert til int.
Forskjellige objekter har alltid forskjellig hashCode()

Men husk at i vår anvendelse kan det være slik at to forskjellige objekter skal regnes som like og derfor må få samme hashverdi. Dette kan vi løse med @Override

string_hash i Python

```
01 arguments: string object
02 returns: hash
03 function string_hash:
04     if hash cached:
05         return it
06     set len to string's length
07     initialize var p pointing to 1st char of string object
08     set x to value pointed by p left shifted by 7 bits
09     while len >= 0:
10         set var x to (1000003 * x) xor value pointed by p
11         increment pointer p
12     set x to x xor length of string object
13     cache x as the hash so we don't need to calculate it again
14     return x as the hash
```

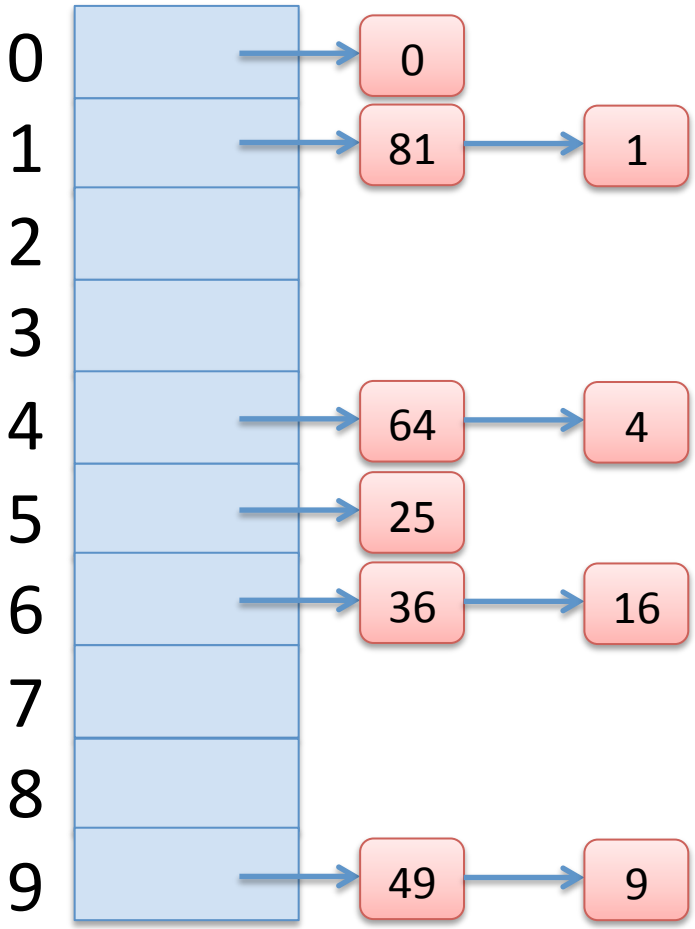
Kollisjonshåndtering

Hva gjør vi når 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.

Åpen hashing (Separate chaining)



Forventer at hashfunksjonen er god, slik at alle listene blir korte.

Load-faktoren λ er antall elementer i hashtabellen i forhold til tabellstørrelsen. For åpen hashing ønsker vi $\lambda \approx 1$.

Lukket hashing – åpen adressering

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

h_i er gitt ved

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

slik at $f(0) = 0$.

Merk at vi trenger en større tabell enn for åpen hashing – generelt ønsker vi her $\lambda \approx 0{,}5$. Skal se på tre mulige strategier (valg av f):

- Lineær prøving
- Kvadratisk prøving
- Dobbel hashing

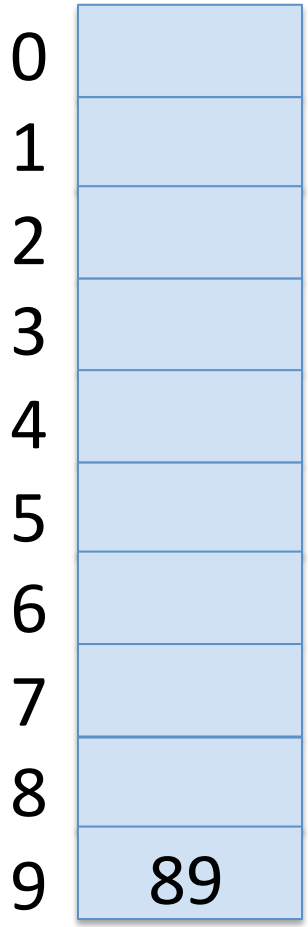
Lineær prøving

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

Eksempel:
Input: **89**, 18, 49, 58, 69
Hash-funksjon:
h(x,tableSize) = x **mod** tableSize

hash(i,x) = (h(x,tableSize) + f(i)) mod tableSize

hash(0 , 89) = (89 + 0) **mod** 10 = 9



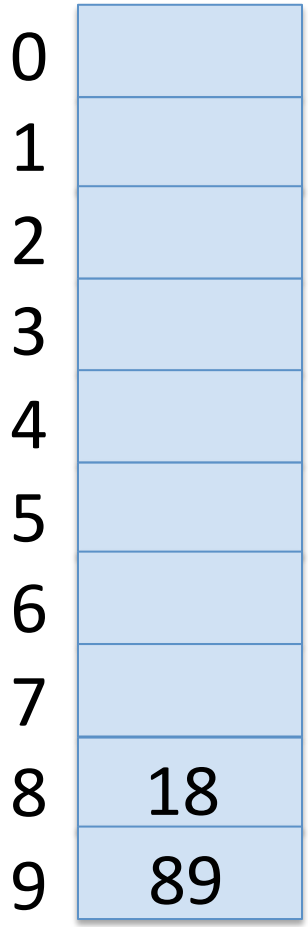
Lineær prøving

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

Eksempel:
Input: 89, **18**, 49, 58, 69
Hash-funksjon:
h(x,tableSize) = x **mod** tableSize

hash(i,x) = (h(x,tableSize) + f(i)) mod tableSize

hash(0 , 18) = (18 + 0) mod 10 = 8



Lineær prøving

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

Eksempel:
Input: 89, 18, **49**, 58, 69
Hash-funksjon:
h(x,tableSize) = x **mod** tableSize

hash(i,x) = (h(x,tableSize) + f(i)) mod tableSize

hash(1 , 49) = (49 mod 10 + 1) mod 10
= (9+1) mod 10 = 10 mod 10 = 0

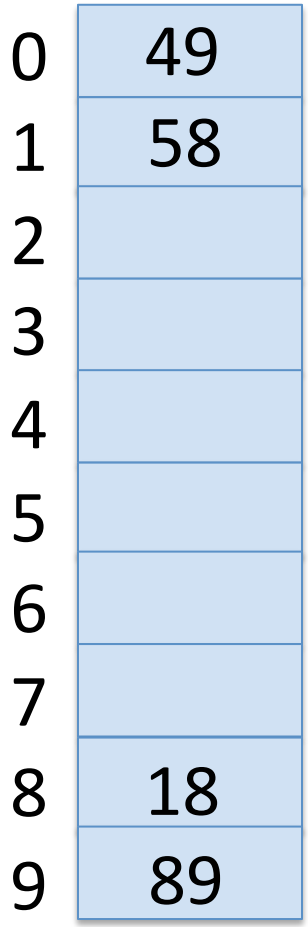
Lineær prøving

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

Eksempel:
Input: 89, 18, 49, **58**, 69
Hash-funksjon:
h(x,tableSize) = x **mod** tableSize

hash(i,x) = (h(x,tableSize) + f(i)) mod tableSize

hash(3, 58) = (58 mod 10 + 3) mod 10
= (8+3) mod 10 = 11 mod 10 = 1



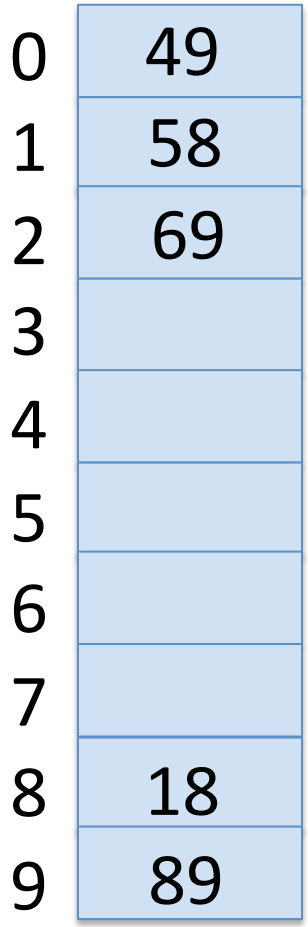
Lineær prøving

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

Eksempel:
Input: 89, 18, 49, 58, **69**
Hash-funksjon:
h(x,tableSize) = x **mod** tableSize

hash(i,x) = (h(x,tableSize) + f(i)) mod tableSize

hash(3, 69) = (69 mod 10 + 3) mod 10
= (9+3) mod 10 = 12 mod 10 = 2



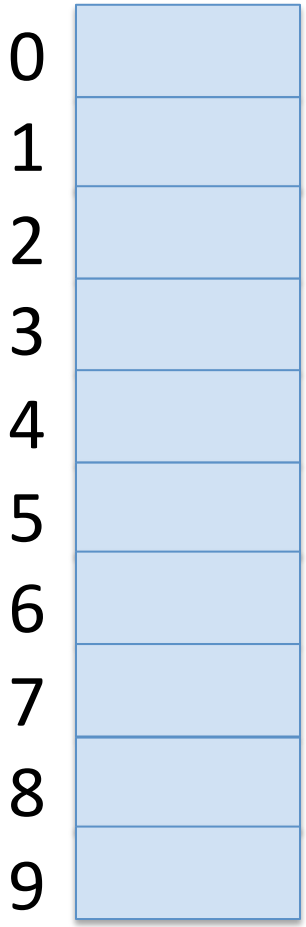
Lineær prøving – ny input

Velger f til å være f(i) = i

Oppgave: Hvordan blir arrayen med denne input: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Hash-funksjon:

hash(i, x) = (x **mod** 10 + f(i)) **mod** 10



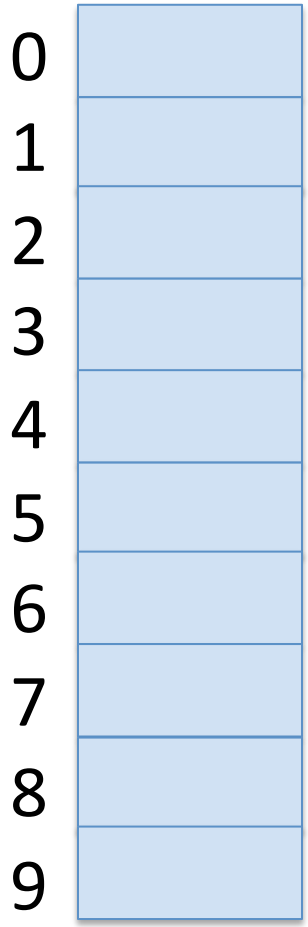
Lukket hashing med kvadratisk prøving

Velger f til å være en **kvadratisk** funksjon av i, som oftest $f(i) = i^2$

Eksempel:
Input: 89, 18, 49, 58, 69

Hash-funksjon:

hash(i, key) = (key **mod** 10 + f(i)) **mod** 10,
der $f(i) = i^2$



Lukket hashing – dobbel hashing

Velger f *til å være funksjon som bruker en ny hashfunksjon, typisk*

f(i) = i × [R − (x **mod** R)]

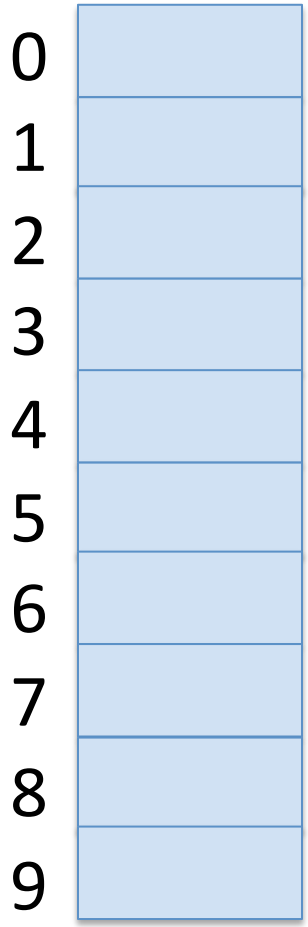
hvor R *er et primtall mindre enn* tableSize

Eksempel med input: 89, 18, 49, 58, 69

Hash-funksjoner:

hash(0,key) = key **mod** 10

hash(i,key) = i × [7 − (key **mod** 7)] *(for* i=*1,2,…)*



Sekundær opphopning

Selv om man lager sofistikerte hash-funksjoner ved åpen adressering, risikerer man at selv om det finnes ledige indekser, vil de aldri finnes. Dette skjer når hashfunksjonen gir indeksverdier som «går i ring».

Dette kan ikke skje ved enkel lineær prøving.

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 hver 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).*

Javas HashMap

Klassen java.util.HashMap<K,V> :

- implementerer en tabell som gitt et objekt av type K (key) gir (mapper) et objekt av type V (value)
- implementerer interfacet Map
- bruker åpen hashing
- default tabellstørrelse 16
- loadfaktor $\lambda = 0{,}75$
- rehashing hvis $\lambda \geq 0{,}75$

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Python Dictionaries

Fra https://www.laurentluce.com/posts/python-dictionary-implementation/

- bruker lukket hashing
- ved kollisjon brukes $f(i) = 2^i$
- default tabellstørrelse 8
- loadfaktor $\lambda = 0{,}75$
- rehashing hvis $\lambda \geq 0{,}75$
- ny tabellstørrelse er $4 \times b$, der b er opptatte indekser i den gamle tabellen. (Første gang er b lik 6, siden 6/8 = 0,75)