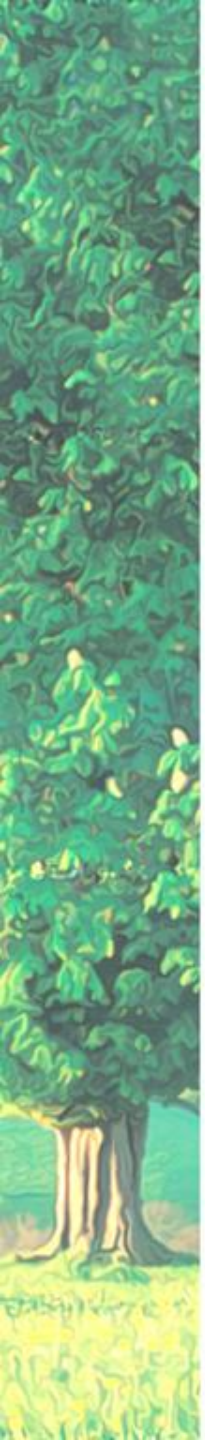
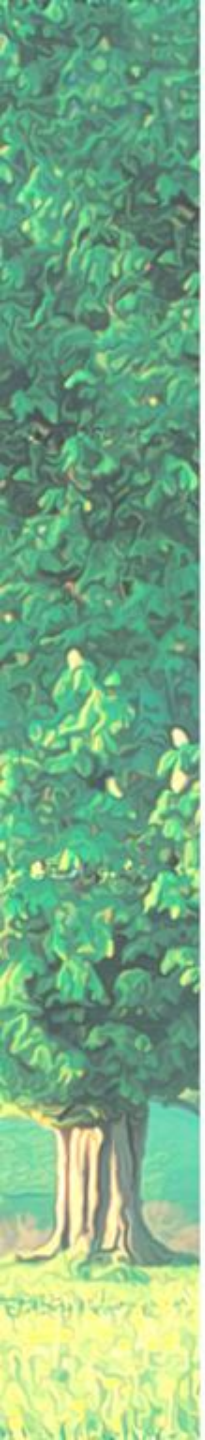


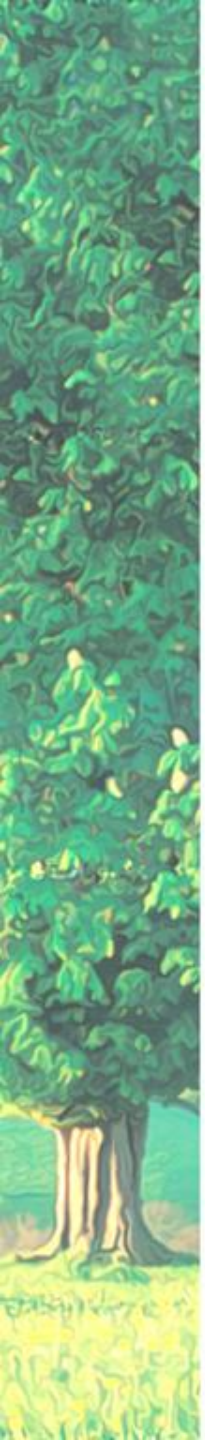
Kapittel 14, Hashing

- Tema
  - Definere hashing
  - Studere ulike hashfunksjoner
  - Studere kollisjonsproblemet



Hashing

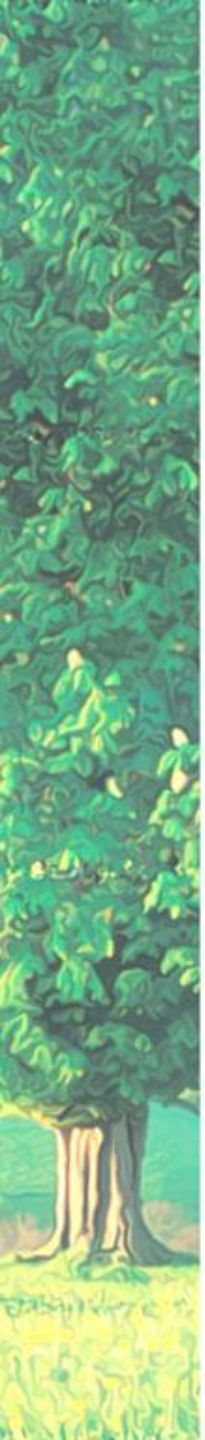
- Hashing er en effektiv metode ved lagring og gjenfinning (søking) av informasjon
- Søkemetoder hittil:
  - Sekvensiell søking
  - Binærsøking



Hashing

- I hashing er elementene lagret i en *hashtabell.* Lokasjonene i tabellen er bestemt av en hashfunksjon.

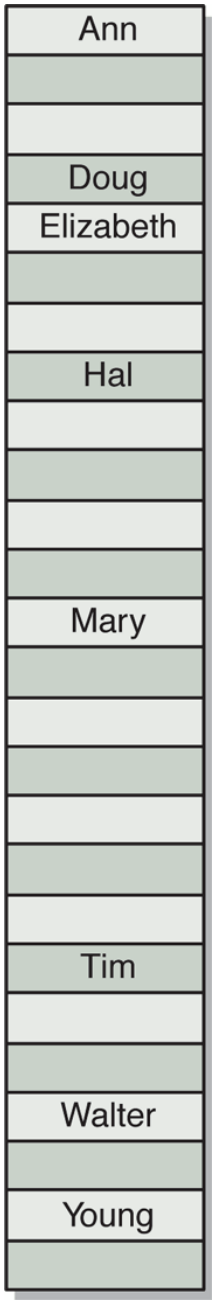
- Hver lokasjon i hashtabellen er kalt en cellle eller en bøtte.

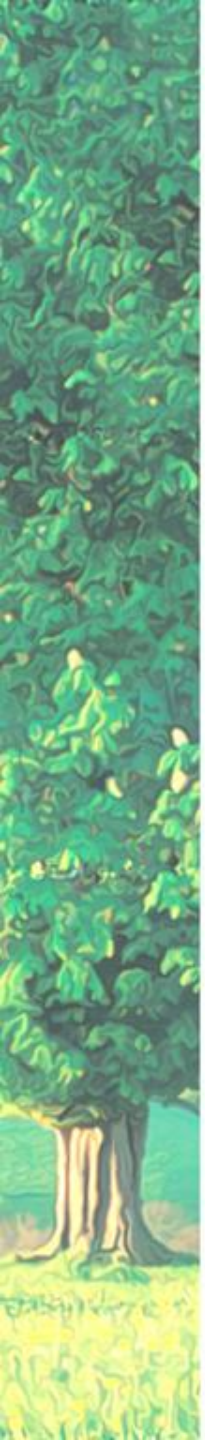


Hashing

- Studerer et eksempel med en tabell som kan inneholde 26 elementer
- Vi ønsker å lagre navn i tabellen.
- Lager en hashfunksjon, *hash* som bruker første bokstaven i navnet,
- hash(navn) → 0…25
  Verdimengden til hash gir **lokasjonen**.

FIGUR 14.1
et enkelt eksempel

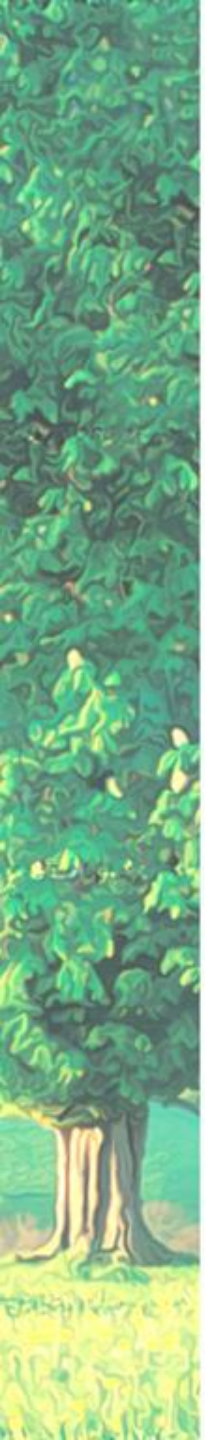




Hashing

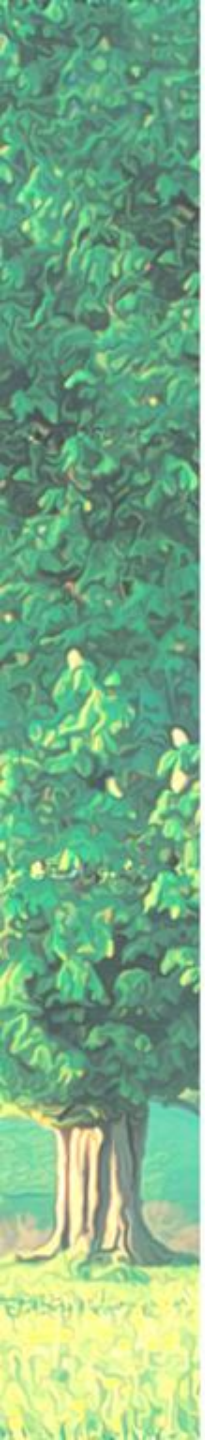
- Tiden det tar å søke etter et bestemt element kalles søketiden (aksesstiden).

- Søketiden er **O(1)** dersom hvert element avbildes til en entydig posisjon i tabellen kalt hashadresse.



Vanlige hashoperasjoner

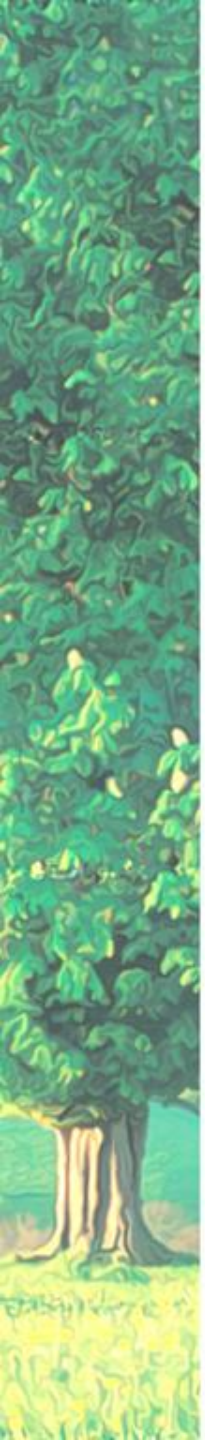
- Opprette en tom tabell med tomme innganger
- Avgjøre om tabellen er tom
- Sette en ny nøkkel inn i tabellen
- Slette en nøkkel i tabellen (obs! delproblem)
- Lese informasjonen fra en tabellinngang
- Oppdatere en tabellinngang



Hashing

- En **kollisjon** oppstår når to eller flere elementer avbildes på den samme lokasjonen.
  Tidligere eks: To navn som starter på samme bokstav.

- En *perfekt* hashfunksjon avbilder hvert element på en entydig lokasjon. (hashadresse).



Hashing

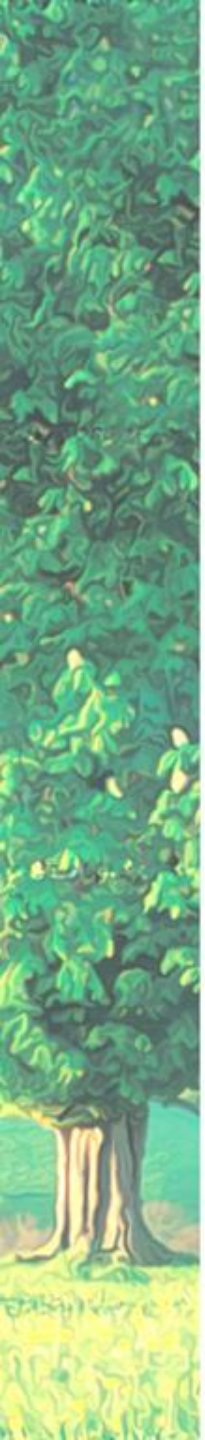
- En må ta stilling til hvor stor hash-tabellen bør være.

- Hvis vi ikke kjenner størrelsen på datamengden, må vi kunne vurdere å utvide tabellen og videre utvide verdimengden til hashfunsjonen.

Hashing

- Etter hvert som hashtabellen fylles vil søketiden øke. **En god hashfunksjon vil fordele elementene uniformt** (med lik sannsynlighet) over hele hashtabellen.

- Når antall elementer øker til et bestemt nivå (feks. fyllingsgrad lik 0.9), er det ofte vanlig å utvide hashtabellen med utviding av verdimengden.



Valg av hashfunksjon
1) Uttrekking

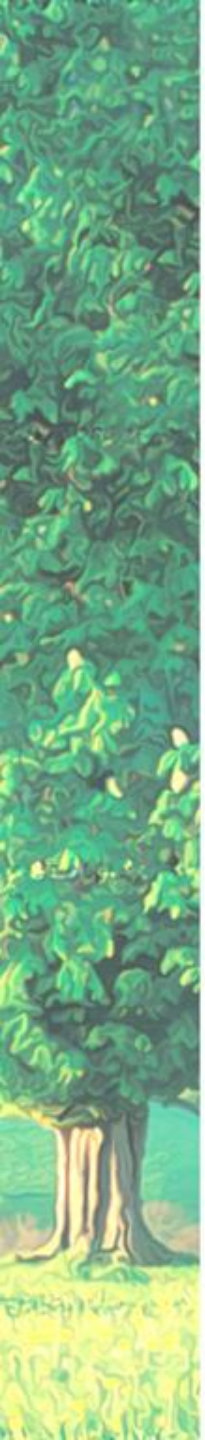
- Bruker en del av elementverdien (nøkkelen) til å beregne lokasjonen der elementet skal lagres.

- I vårt tidligere eksempel trakk vi ut første bokstav i en streng og beregnet bokstav-verdien relativ til bokstaven A.

Valg av hashfunksjon
1) Uttrekking

- Eks:

```
public int hash( String navn )
  {
   int tall =
  (int)navn.charAt(0);
   return tall % tabellLengde;
}
```



Valg av hashfunksjon
2) Divisjon

- Bruker resten ved heltallsdivisjon.

  hash(nøkkel) = Math.abs(nøkkel)%p;

  der p er et passende positvt tall

  hash(nøkkel) → 0…p-1.

- Hvis p = hashTabell.length, så får vi :

  hash(nøkkel) → 0…hashTabell.length-1
  som altså er alle tillatte indekser.

- Primtalsstørrelse på tabellen gir best fordeling av nøkler

Velg av hashfunksjon
3) – Folding (slå sammen)

- Nøkkelen er delt i deler og er deretter slått sammen for å gi en indeks i hashtabellen
- Eks: SSN 987-65-4321 kan vi dele i tre deler 987, 654, 321 og deretter legge disse tallene sammen som gir 1962.
- Til slutt kan vi enten bruke divisjon eller uttrekking basert på tallet 1962 for å beregne indeksen (lokasjonen).

Valg av hashfunksjon
4) – Kvadrere midten

- Nøkkelen kvadreres først.
  Deretter brukes uttrekking av passende antall sifre i midten.

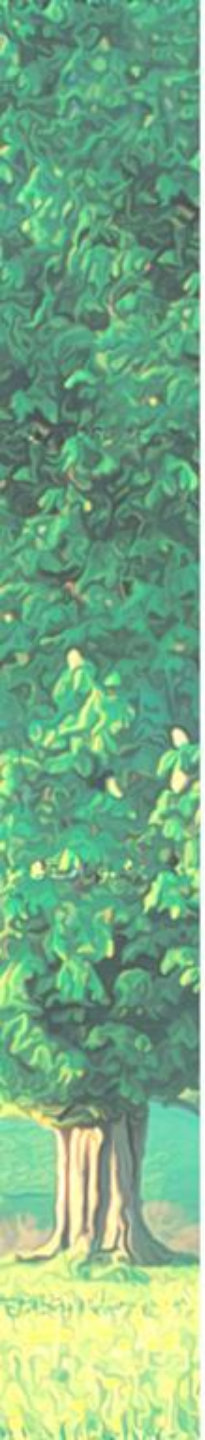
- Eks, hvis nøkkelen er 4321
  - $4321^2 = 18671041$
  - Uttrekking av tre sifre

Valg av hashfunksjon
5) – Radix Transformasjon

- Nøkkelen transformereres til en annet tallsystem.

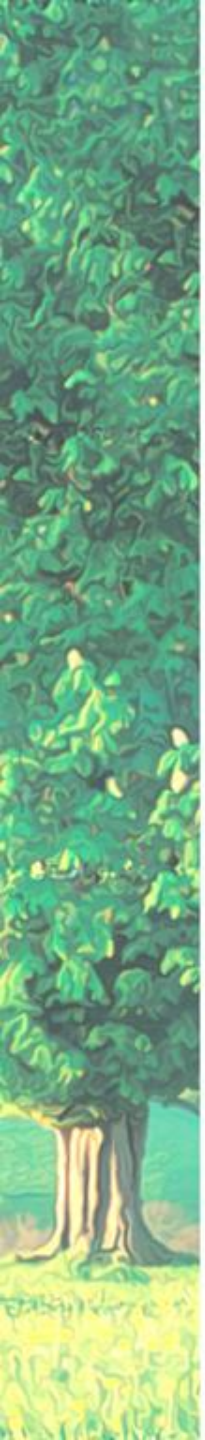
- Eks: Nøkkelen er 23 i 10- tallsystemet Vi kan konverterere tallet til 32 i 7-tallystemet. Deretter kan vi bruke divisjonsmetoden ved:

  32 % tabellstørrelsen som gir indeksen



Hashing – Håndtering av kollisjoner

- La K_1 og K_2 være to ulike nøkler
- K_1 og K_2 er **synonymer** mhp hash() dersom hash(K_1)== hash(K_2)
  (adresserer samme celle)
- **Kollisjon** får vi dersom hash(K_1) == hash(K_2),

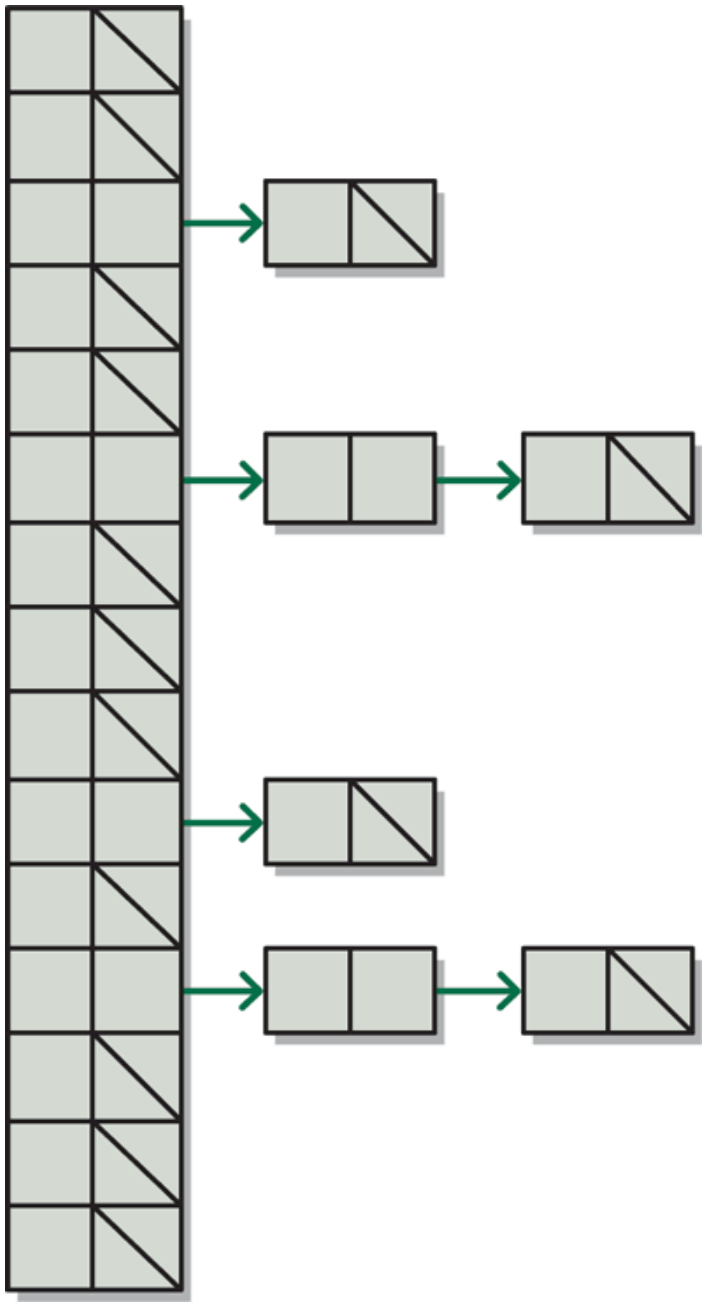


- Håndtering av kollisjoner
1) Kjedet overløp

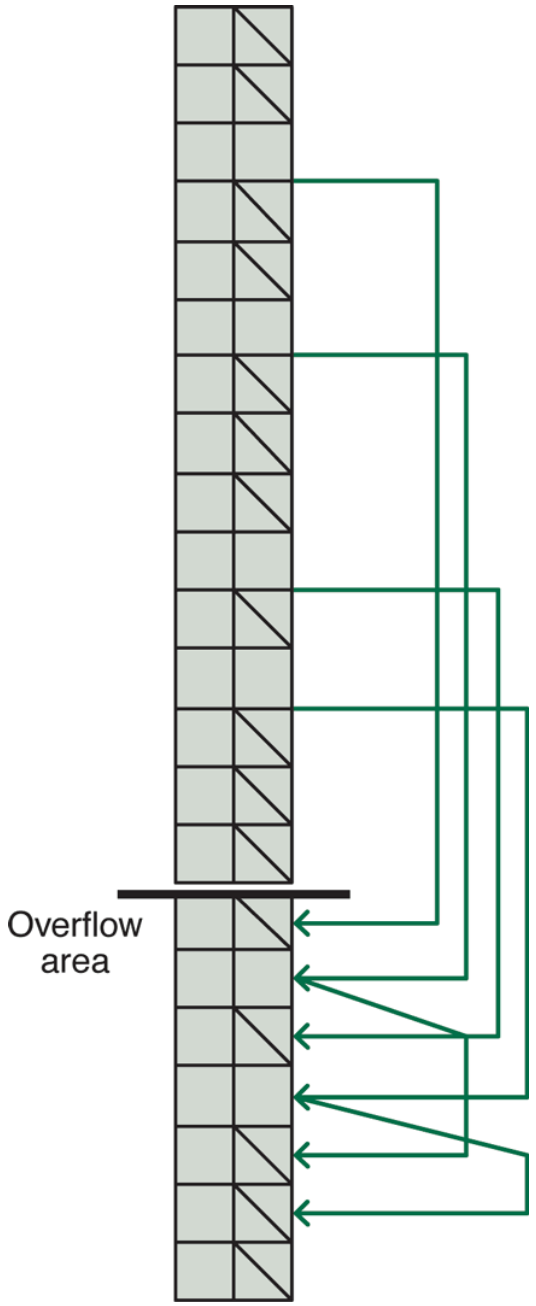
- Benytter en liste pr. celle (bøtte) slik at hver liste inneholder alle synonymer for denne bøtten.

  En søking vil medføre beregning av hash-adressen hash(K) og deretter en lineær søking i listen til hash(K).

FIGUR 14.2
Kjedet overløp



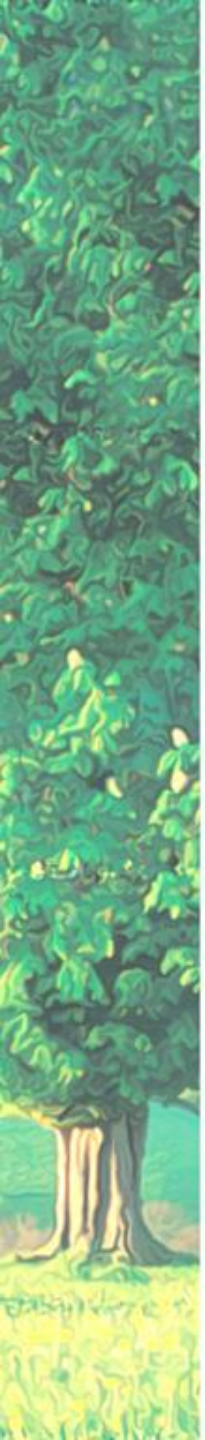
FIGUR14.3
Kjeding ved bruk av et overflytområde



2) Åpen adressering med lineær probing

- Tar neste ledige inngang der det er plass.

- Går **syklisk** gjennom hashtabellen
  Når alle inngangene over hash(K) er fulle, så går vi til starten av tabellen og søker etter ledig plass inntil posisjonen hash(K) (maks en runde).



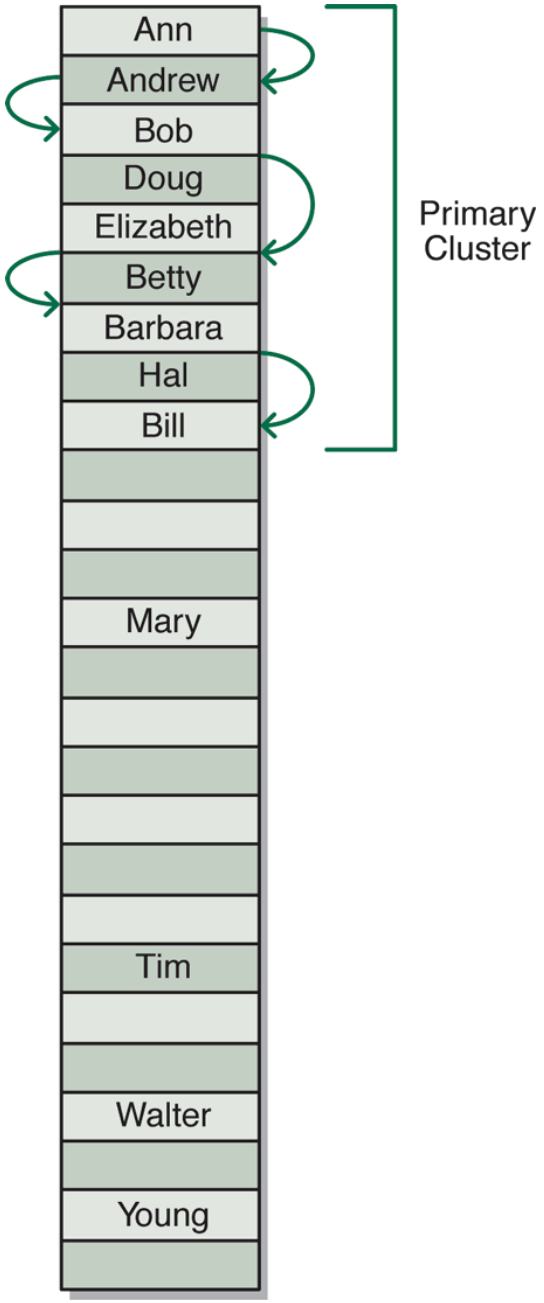
2) Åpen adressering med lineær probing

- Fordelen er at søkingen etter ledig plass ikke medfører større utregninger

- En ulempe er at hash-adressene kan ”klynge” seg sammen (”primær-cluster”) Når flere nøkkelverdier blir avbildet, vil dette fenomenet forsterkes.

  En annen ulempe er at ved søking etter ledig plass kan det medføre et stort antall sml. av nøkkelverdier med ulike hash-adresser.

FIGUR 147.4
Åpen adressing med lineær probing



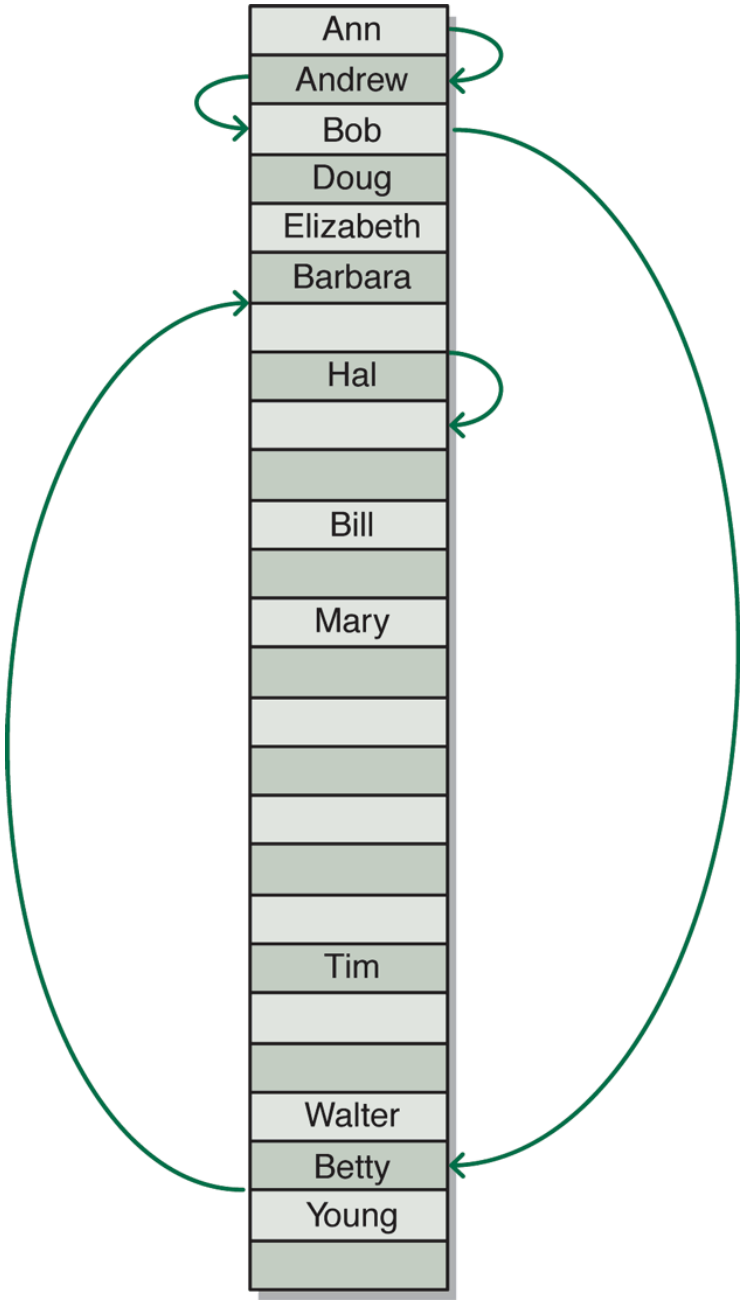
Håndtering av kollisjoner
Andre metoder

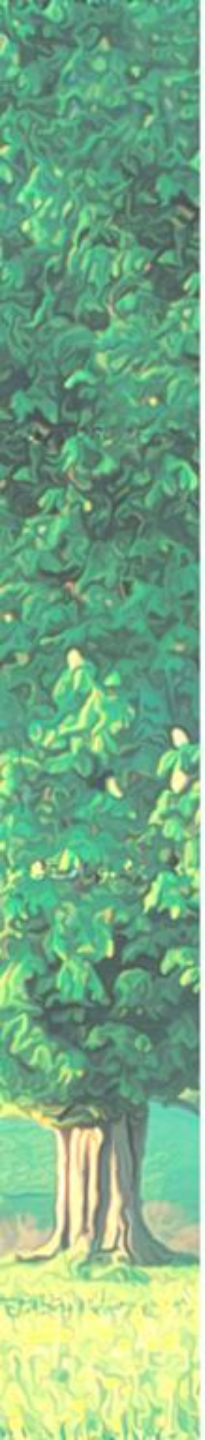
- **Kvadratisk probing**
  nyhash(K) = hash(K) + $(-1)^{i-1}((i+1)/2)^2$
  Eliminerer problem med primær-cluster (oppnår bedre fordeling)

  i = 1..hashTabell.length-1

  Benytter så divisjonsmetoden for å få hashkoden innenfor tabell-indeksområdet.
  nyhash(K) % hashTabell.length
  Søke sekvens p, p+1, p-1, p+4, p-4, p+9, p-9, …

FIGUR 14.5
Åpen adressing med kvadratisk probing

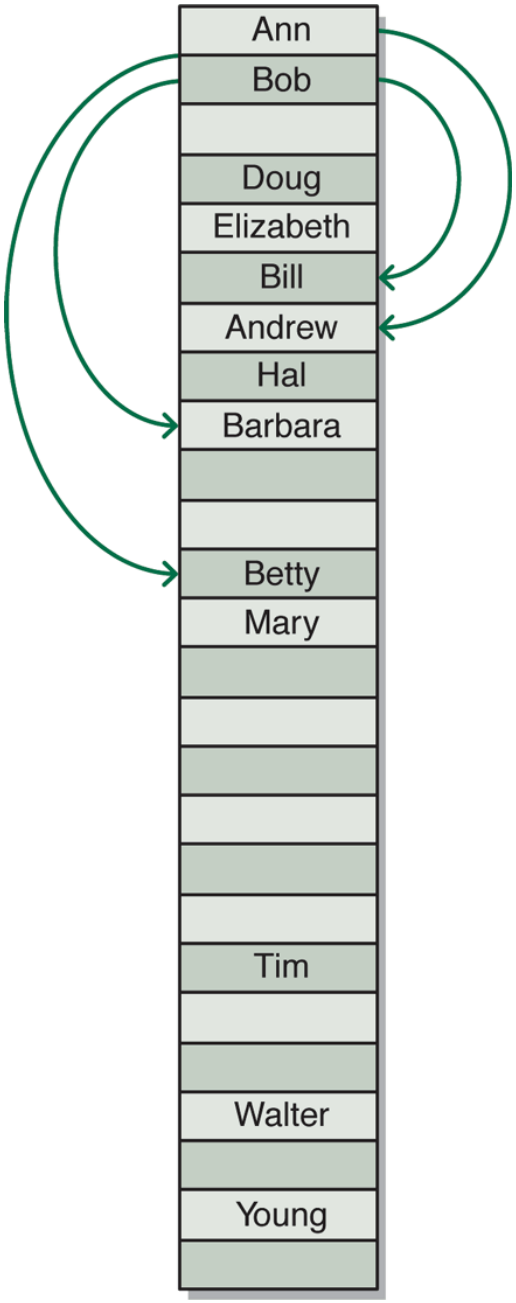


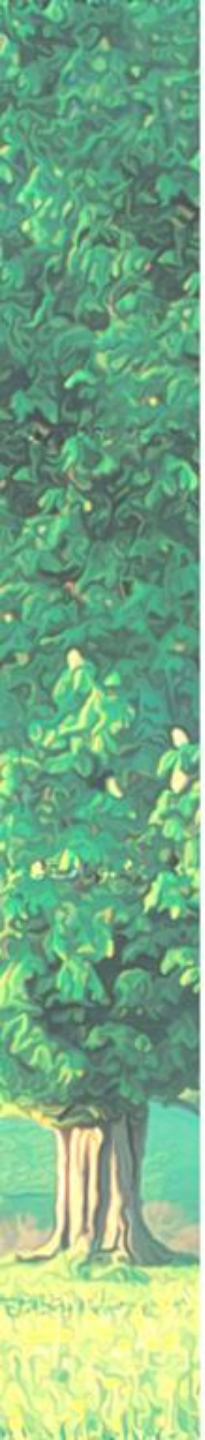


Håndtering av kollisjoner
Andre metoder

- Dobellhashing
- Anta p = hash(K) er en opptatt adresse,
- Forsøker da andreHash(K)
  p’ = p + andreHash(K)
- Hvis denne posisjonen også er opptatt:
  p”= p + 2*andreHash(K),
  fortsetter inntil en ledig plass fins

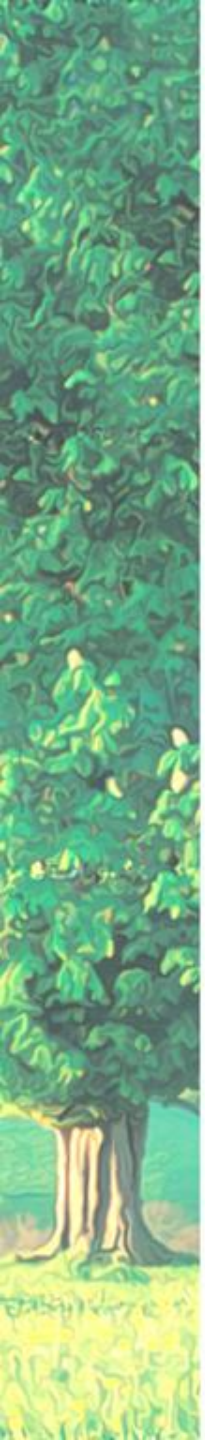
FIGURE 14.6
Åpen adressing ved dobbellhashing





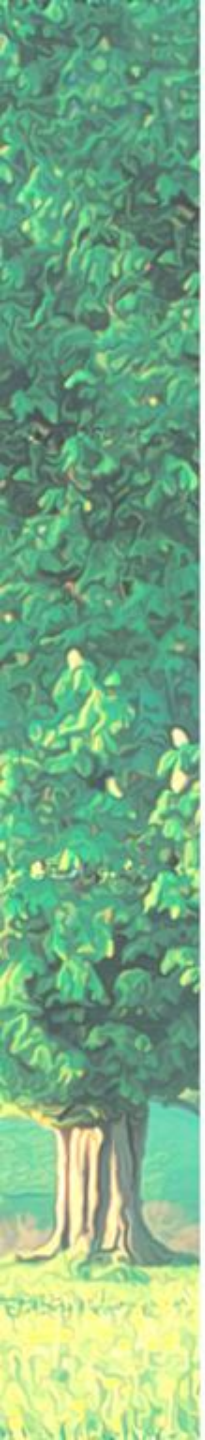
Slette elementer fra en hashtabell

- Slette i kjedet overløp medfører: slette aktuell node.
- Slette i kjeding med overflytområde: Tilsvarer sletting i kjedet liste.
- **Slette i åpen adressering**: Slette innholdet.
  OBS! Må ha et "flagg" som markerer om inngangen er slettet ellers brytes probesekvensen og vi må da søke gjennom alle innganger som er tungvint!



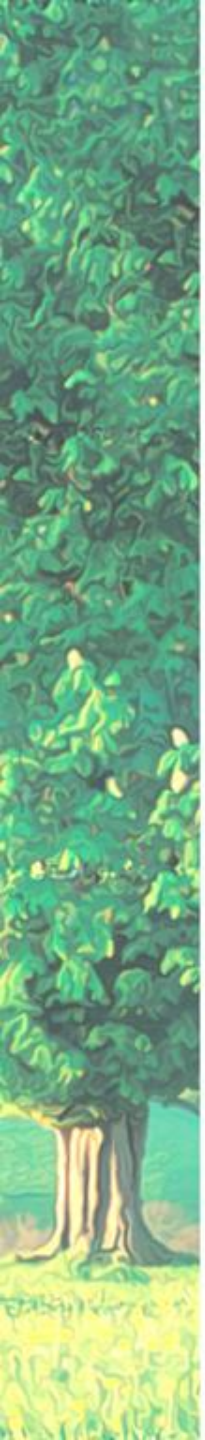
Et eksempel (1)

```
class TabellInngang {
   //nøkkel er -1 hvis inngangen er slettet
    public int nøkkel;
    public InfoType  info;
   }
class InfoType  {
    public Object dataFelt1;
    public Object dataFelt2;
}
```



(2)

```
class HashTabell {
   private int   M;   // Tabellstørelsen
  //Kapasiteten er M -1, dvs. en tom
                                  /reservert plass
  private int  antall;   // antall innganger
  private  TabellInngang[] T;  //
                                       HashTabellen
  …
```



(2) Basert på en tom/reservert plass

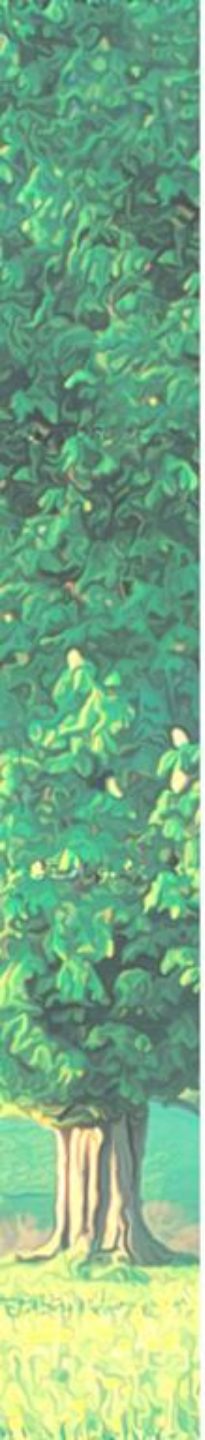
```
// Bruker en tom plass for at løkken ved
   søk ikke skal gå uendelig, ellers må vi
   telle.

public HashTabell(int tabellStor){
    M = tabellStor;
    antall = 0;
    T = new TabellInngang[M];

}//
```

(3)

```
private int h(int K){ // hashfunksjon
    return K%M;
 }//
  //OBS! Forutsetter at  probefunksjonen
  dekker alle innganger. Se lysark 40.
  private int p(int K){// probefunksjon
  int probe = (K/M)%M;
    if(probe < 1)
        probe =1;
  return probe;
}//
```



(4)

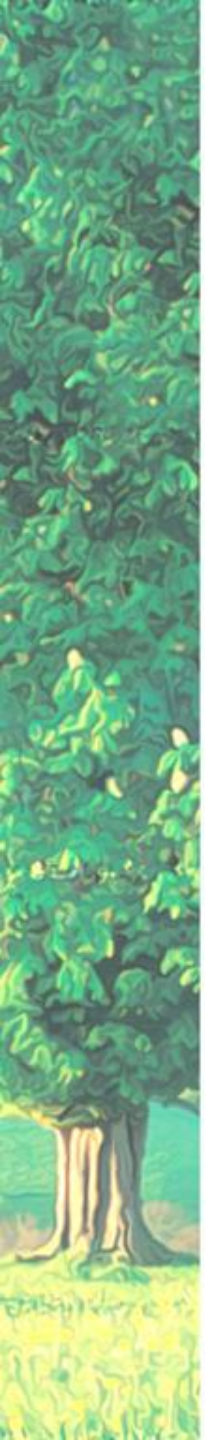
```
void hashInn(int K, InfoType I){
        int i = h(K);
        int probeDekrement = p(K);

      if(antall < M -1){ // Husk en tom plass
         // finner ledig plass
        while(T[i] != null && T[i].nøkkel != -1)
            i = (i - probeDekrement)%M;
        }
```

(5)

```
        // setter inn
        T[i].nøkkel = K;// Innsett
        T[i].info = I;
        antall++;
 }else{
    System.out.println(”Tabellen er full”);

}//hashInn
```



(6) Vi må kanskje gå flere runder.

```
int hashSøk(int K){
    int i = h(K);
    int probeDekrement = p(K);
```

(6) Vi må kanskje gå flere runder i tabellen!.

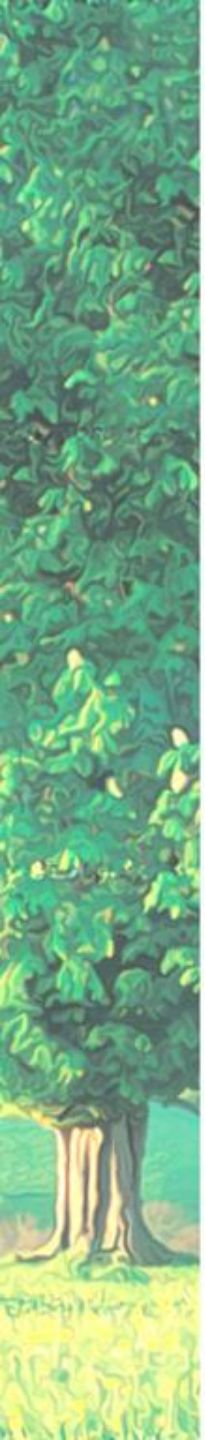
```
    while(T[i] != null) {
        int probeNøkkel = T[i].nøkkel;
        if(T[i].nøkkel == K)
                return i;
        i = (i - probeDekrement)%M;
    }// while

    return -1;// ulovlig indeks

}//class
```

(7)

```
public  void slett(int K){ //Husk en tom plass
  boolean slettet = false;
  int   = h(K);
  int probeDekrement = p(K);
  int probeNøkkel = T[i].nøkkel;
  while(T[i] != null && !slettet){
     if(T[i].nøkkel == K){
            T[i].nøkkel = -1;
            slettet = true;
      }//if
```



(7)

```
 i = (i - probeDekrement)%M;
}//while

if(slettet)
     System.out.println (”slettet”)
else
    System.out.println(”fins ikke”)

}//slett
```

Probesekvensen

- Det er viktig at probesekvensen dekker hele tabellen på en eller flere runder; K er nøkkel; M er tabellstørrelsen $1 \leq p(K) \leq M$
- Dette får vi til ved (uten bevis) :

1) **Velg tabellstørrelsen som et primtall,**

   – For eksempel M=97. eller

2) **Velg M som potens av 2 og p(K) som odde tall**

   – For eksempel. M = 2^3,

   – og p(K) $\in$ {1, 3, 5, 7}

Sammenligninger av søkemetoder

Repr	Initialisere	Avgjøre full tabell	**Søke/Hente / Oppdatere**	Sette inn	Slette
Sortert tabell	O(n)	O(1)	**O(log n)**[1]	O(n)	O(n)
Hashtabell	O(n)	O(1)	**O(1)**	O(1)	O(1)

1 Binærsøking for en allerede sortert tabell sml også søking i et BSTre som er mest mulig balansert (for eksempel komplett)

I hver iterasjon av løkken halveres størrelsen på listen (deltabell) som skal undersøkes .
I verste tilfelle vil vi undersøke $\log_2 n$ elementer i listen.