19
Hashtabeller.

Hashing problemet.

Hashfunktioner.

Kollision.

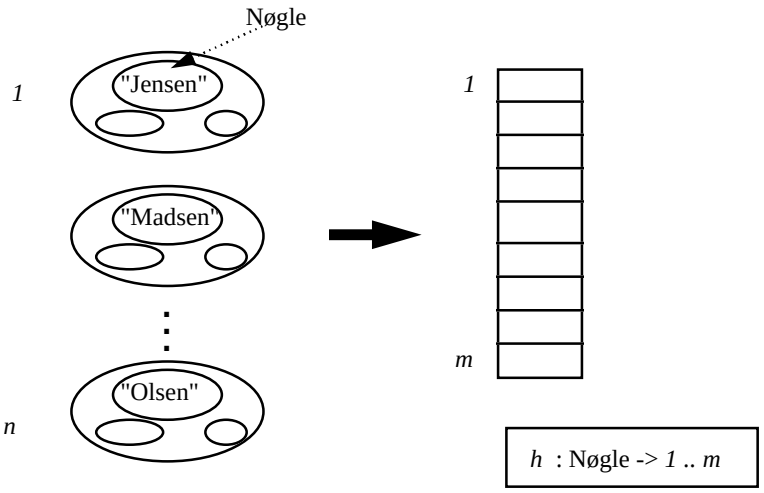
Søgning og indsættelse.

Sammenligning af hashtabeller og søgetræer.

Noter

Hashing-problemet (1).

Vi ønsker at afbilde *n* objekter på en tabel med *m* indgange.



PS1 -- Hashtabeller

Noter

Lad os antage, at vi har et antal objekter, hver med en **nøgle**, som vi ønsker at gemme i en tabel. Objekternes placering i tabellen skal ene og alene bestemmes på baggrund af nøglens værdi. En nøgle antages at identificere et objekt entydigt.

Vi ønsker med andre ord at definere en funktion h, som afbilder nøgler ind i intervallet af indeces af tabellen. En sådan funktion kaldes en **hashfunktion.**

"Hashing" betyder "at hakke i stykker" eller "at rode sammen". Navnet er taget i anvendelse, fordi hashing kan opfattes som om man "hakker nøglen i stykker" og benytter dele af nøglen som basis for at organisere objekter i en "rodet" samling i en tabel.

"Nøgle transformation" ("key transformation") er en alternativ betegnelse for "hashing".

Hvis en hashfunktion kan beregnes meget effektivt (kører meget hurtigt), kan man, givet en nøgle N, meget hurtigt finde objektet med nøglen N. Dette er attraktivt; og tilmed udgør dette et søgeprincip, der er let at forstå.

Hashtabellen antages at være et array, hvor der er direkte tilgang til elementer med givne indeces.

Ovenfor er objekterne data om personer, og nøglerne er valgt som personernes efternavne.

Hashing kan både benyttes på tabeller, som er lagret i **primært lager** (RAM), og i tabeller på **sekundært lager** (disk).

Hashing problemet (2).

1.	Konstant, lille antal objekter (= *n*) med kendte, faste nøgler	Tabel med *m* indgange, *n Š m.*
2.	Variabelt, lille antal objekter (= *n*) med ikke-kendte nøgler	Tabel med *m* indgange, *n Š m.*
3.	Variabelt, stort antal objekter (= *n*) med ikke-kendte nøgler	Forholdsvis lille tabel med *m* indgange, *n >> m.*

Hvordan finder man en god hashing-funktion?

Hvordan håndterer man kollision?

Noter

Tilfælde 1: I tilfælde 1 er vi f.eks. i den situation, at vi skal indplacere 31 objekter i en tabel, som har 41 indgange. Vores opgave går således ud på, at lave en funktion fra de 31 kendte nøgler (så som strenge, der repræsenterer 31 efternavne på personer) ind i intervallet [1..41], således at intet objekt lageres oven i et andet. Sådanne funktioner er sjældne! Der er 41^31 mulige funktioner (hvert af de 31 elementer kan afbildes over på enhver af de 41 pladser). Men der er kun 41*40*...*11 funktioner, der afbilder de 31 objekter over i indbyrdes forskellige pladser i tabellen. Hvis man regner på dette, finder man ud af, at ca. kun 1 ud af hver 10 million funktion er brugbar. (Dette eksempel er taget fra Knuth, *Sorting and Searching).*

Tilfælde 2: Dette er en mere realistisk situation, men det er desværre også helt håbløst at forestille sig, at man kan afbilde ukendte objekter med ukendte nøgler over i en tabel, således at der ikke opstår sammenfald. Hvis eller når der opstår en situation, hvor to objekter afbildes over i samme indgang taler vi om **kollision.**

Tilfælde 3: Dette er den typiske og realistiske situation. Universet af objekter kunne være alle danskere, tabellen kunne være på 700 elementer, og problemet kunne være at gemme information om alle studerende på Afdeling for Matematik og Datalogi. Nøglen vil sikkert være personnummeret. Man kunne i princippet allokere en tabel med indeces fra minimum-personnummer til maximum_personnummer, altså en tabel med én plads pr. dansker. Men da vi kun vil gemme information om en meget lille del af danskerne, bliver udnyttelsen af en sådan tabel meget dårlig.

Opgave. Denne opgave går ud på at kaste lys over tilfælde 1. Antag vi har en *mængde* af personer {marie, arne, anders, lars, martin, hanne}. Led efter en funktion, der afbilder disse seks navne ind i forskellige indgange i en tabel af størrelse 10. Hvor sjælden er en sådan funktion? Det er vigtigt at indse de konkrete såvel som de principielle problemer i at finde en sådan funktion.

Hash-funktioner (1).

h : Nøgle -> 0 .. m-1

En hashfunktion bør fordele objekterne jævnt på tabellens indices.

Eksempel på en simpel hashfunktion:

h: Tekststreng -> 0 .. m-1

h(s) = ((ord(c1) + ... ord(ck))
mod m)

"c1 c2 .... ck"

- En mængde af tekststrenge er ofte "sammenklumpede".
- Det må tilstræbes, at "sammenklumpede" strenge ikke afbildes i samme indeks af hashfunktionen.
- Tallet m vælges ofte som et primtal.

PS1 -- Hashtabeller

Noter

Angående den simple hash-funktion:

For at gøre hash-funktionen hurtig kan man være fristet til kun at tage et vist antal tegn i betragtning først (eller sidst) i strengen, i beregningen af summen af ordinal-værdier.

Det betyder, at alle strenge, der starter (eller slutter) med de samme bogstaver, har en risiko for at blive afbildet i det samme indeks af hash-funktionen.

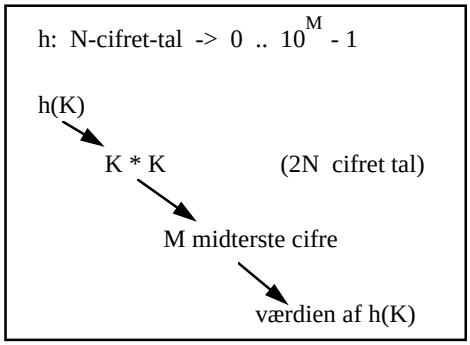
Alle strenge, som er en permutation af de samme bogstaver, afbildes under alle omstændigheder i samme indeks med ovenstående hashfunktion.

Når vi siger, at en mængde af tekststregne ofte er "sammenklumpede" mener vi, at teksstrengene ofte fordeler sig i delmængder, der har et stort sammenfald af bogstaver og delstrenge. Som et eksempel på en sådan delmængde kan man tænke på alle afledninger af det sammen ord.

Der henvises iøvrigt til afsnit 5.2 af Weiss, som indeholder en vældig god diskussion af hashfunktioner.

Hash-funktioner (2).

Eksempel på en hashfunktion, hvor nøglerne er heltal:



Noter

Opgave: Lad os antage, at nøglen i vore dataobjekter er danske person-numre, og at hashtabellen har en størrelse på 10.000 elementer (0..9999). Diskuter problemerne i at vælge forskellige, sammenhængende sekvenser på fire cifre af et personnummer som værdien af hash-funktionen.

Hvad bliver udnyttelsesgraden af en hashtabel, hvis de første fire cifre i personnummeret anvendes som værdien af hashtabellen.

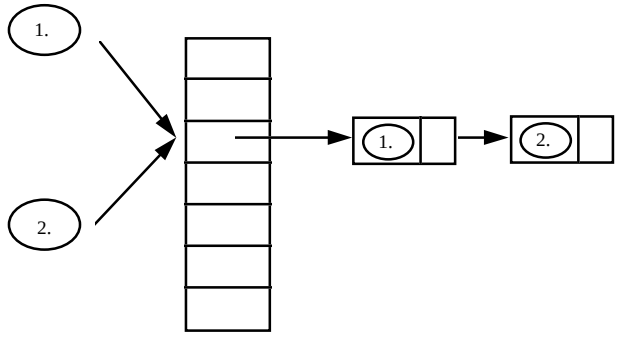
Vurder også udnyttelsesgraden, hvis de sidste fire cifre i personnummeret anvendes.

Vurder hvordan ovenstående hashfunktion vil virke på personnumre.

Problemerne skal bl.a. ses i lyset af, at hashfunktioner skal kunne bruges på f.eks. alle personer i en given population, såsom alle børn i en skoleklasse, eller alle personer i en husmoderforening (og give en passende spredning i tabellen).

Håndtering af kollision (1).

Åben hashing med direkte kædning.



To objekter, der hashes på samme indeks, lagres i en kædet liste, som har udgangspunkt i tabellen.

Noter

Kollision forekommer oftere, end man tror.

Betragt kollisionsproblemet: "N mennesker er samlet til fest. Hvad er det mindste N, således at sandsynligheden for at to af festdeltagerne har samme fødselsdag (dag og måned) er større end 0.5".

Det kan vises, at det mindste sådanne N er 23.

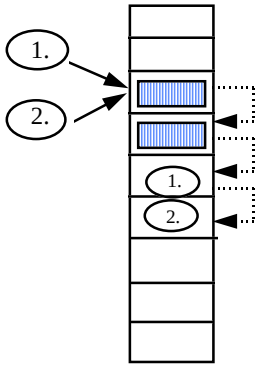
I vores sammenhæng betyder det, at hvis 23 eller flere tilfældige elementer (personer) afbildes i en tabel med 365 indgange via en hashfunktion, som afspejler deres fødselsddag, er sandsynligheden for, at "det går godt" (at der ikke sker en kollision) mindre end 0.5.

Ovenstående analogi er lånt fra Knuth (som igen har lånt det fra en anden, angivet kilde).

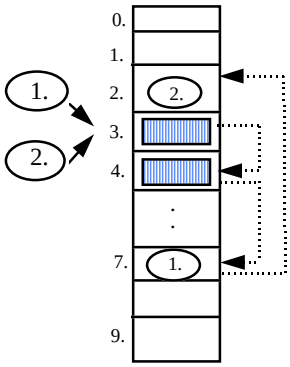
Håndtering af kollision (2).

Lukket hashing.

"Linear probing":



"Quadratic probing":



- Alle pladser i tabellen udnyttes.
- Tendens til sammenklumpning.

- Tabellen udnyttes ikke fuldt ud.
- Større spredning.

PS1 -- Hashtabeller

Noter

De stribede pladser i tabellen indeholder eksisterende objekter, som er indsat på det tidspunkt, hvor vi indsætter vore objekter 1 og 2. Det antages, at objekt 1 indsættes før objekt 2.

De stiplede pile angiver mønstret af"probes", altså hvike undersøgelser der foregår, for at finde en tom plads i tabellen.

Lineær probing:

Ved lineær probing leder man under indsættelse cyklisk efter en tom plads.

Tilsvarende, når man søger efter et objekt med en nøgle N, hvor man via hashing-funktionen rammer det tredie felt i tabellen, søger man cyklisk efter et objekt, som har nøglen N. Hvis man finder et tomt felt i tabellen, kan man konkludere, at objektet ikke er tilstede.

Det fremgår klart, at der sker en (uønsket) sammenklumpning af objekter i tabellen, som har samme hashværdi på deres nøgler.

Quadratic probing:

Vi antager i eksemplet ovenfor, at to objekter begge hashes til 3. position i tabellen. Ved kvadratisk probing leder man ved indsættelse efter 1., 4., 9.,16. ... plads efter den tredie plads, cyklisk i tabellen. I eksemplet, hvor tabelstørrelsen er 10, vil vi oprindelig via hashing prøve plads nummer 3. Dernæst vil vi via probing prøve plads 4, 7 og 2.

Man kan observere, at man måske her vil komme i den situation, hvor man ikke kan finde en tom plads, selv om en sådan eksisterer. I Weiss vises det, at hvis tabellens størrelse er et primtal, vil mindst halvdelen af tabellen blive afsøgt for en tom plads, inden vi møder en plads, vi allerede har afprøvet.

Søgning, indsættelses og sletning i en lukket hashtabel.

Søg(N: Nøgle, H: Hashtabel): Søgeresultat :
Præbetingelse: H må ikke være fyldt til grænsen.

```
i := hash(N);
while  H[i] er optaget  and then  H[i].nøgle ° N
do   i := next(i) ;
if  H[i] er optaget   and then  H[i].nøgle = N
then  returner dataobjektet i H[i] fundet på i-te plads.
else  { H[i] er fri  eller slettet }
      returner "ikke fundet" og  indeks i er brugbar ved indsættelse
```

Indsæt (D: Dataobjekt, H: Hashtabel):
Præbetingelse: H må ikke være fyldt til grænsen.

```
let søgeresultat be Søg(D.nøgle, H) in
  if søgeresultat er "ikke fundet" og indeks i er fri
  then Indsæt  D på i-te plads i  H
```

Funktionen
next: index -> index

- linear probing
- quadratic probing

Slet(N: Nøgle, H: Hashtabel):
Præbetingelse: H må ikke være fyldt til grænsen.

```
let søgeresultat be Søg(N, H) in
  if søgeresultat er et dataobjekt med nøgle  N på i-te plads
  then Marker  i-te plads i H som slettet.
```

PS1 -- Hashtabeller

Noter

Ovenstående operationer viser principperne i hhv. søgning, indsættelse og sletning i en lukket hashtabel. Funktionen Søg returnerer enten (i tilfældet fundet) det eftersøgte dataobjekt, eller (i tilfældet ikke fundet) det indeks hvori et dataobjekt med den givne nøgle kan indsættes. Derved slipper vi for at gennemføre den samme probing to gange i træk i det tilfældet, hvor vi skal indsætte et nyt dataobjekt.

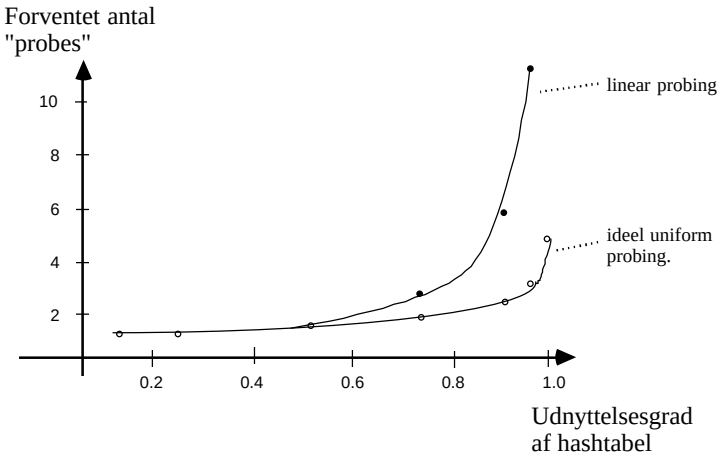
En indgang i tabellen kan enten være *optaget*, *fri*, eller markeret som *slettet*. Sidstnævnte er resultatet af det som Weiss kalder for "doven sletning". Det er fatalt at fjerne et element fra en lukket hashtabel, idet det ville umuliggøre at genfinde objekter, der benytter det slettede objekt som "mellemstation" under probing.

Prebetingelsen af Søg (og dermed de andre operationer) udtrykker, at tabellen ikke må være fyldt til grænsen. Grænsen afhænger af hvilken probingteknik vi bruger. Det essentielle er at vi i tilfælde af at objektet med nøglen N ikke findes i tabellen kan returnere en tom plads, hvor objektet kan indsættes. Alternativt skulle vi holde styr på, hvornår vi havde været hele turen rundt i tabellen (hvilket er enkelt med lineær probing).

Vi antager også ved indsættelse, at der er plads til det nye dataobjekt. Ligeledes ved sletning, idet vi ellers ville komme i problemer når vi kalder Søg.

Funktionen next afhænger af, om vi arbejder med lineær probing eller kvadratisk probing. Next er formuleret som en funktion af det tidligere afprøvede index. For at få dette til at virke med kvadratisk probing, må funktionen huske på, hvor mange gange den er blevet kaldt (eller bedre: man må overføre en parameter, som holder styr på dette).

Effektivitet af hashing ift. tabellens opfyldning.



Noter

Ovenstående grafer viser sammenhængen mellem tabellens opfyldningsgrad og det forventede antal gange, man skal forsøge at finde en tom plads i tabellen.

I den nederste kurve forudsættes der en kollisionshåndtering, hvor det nye forsøg på at ramme et tomt felt rammer tilfældig på tilbageblevne pladser i tabellen.

I den øverste kurve forudsættes der linear probing.

Benyt kun graferne til at vise tendenser. De er ikke tegnede nøjagtig nok til egentlig aflæsning af sammenhørende værdier.

Pro og cons af hashing.

Hurtig indsættelse og søgning.

Fast øvre grænse på antallet af objekter i tabellen.

Der bør afsættes mere lager til hashtabellen, end der aktuelt udnyttes.

Det er problematisk at slette elementer fra en lukket hashtabel.

Sorteret udtræk af objekter fra en hashtabel er kostbart.

Noter

Om sletning af data fra en hashtabel:

I tilfælde af en åben hashtabel, er det ikke et problem at slette data.

I tilfælde af en lukket hashtabel, vil sletning af et dataobjekt fra en hashtabel give et "hul" i tabellen, som kan forhindre os i at genfinde tilbageblevne data. Derfor kan vi kun markere dataobjekter i tabellen som slettede, og ikke rent fysisk slette dem fra en lukket hashtabel. Sidstnævnte teknik kaldes som omtalt "doven sletning" i Weiss.

Sammenligning af hashing med andre søgeteknikker.

Forventet køretid:

	Balancerede søgetræer	Hashtabeller
indsættelse af element	O(log n)	O(1)
søgning efter element	O(log n)	O(1)
sletning af element	O(log n)	-
sorteret udskrivning	O(n)	O(n log n)

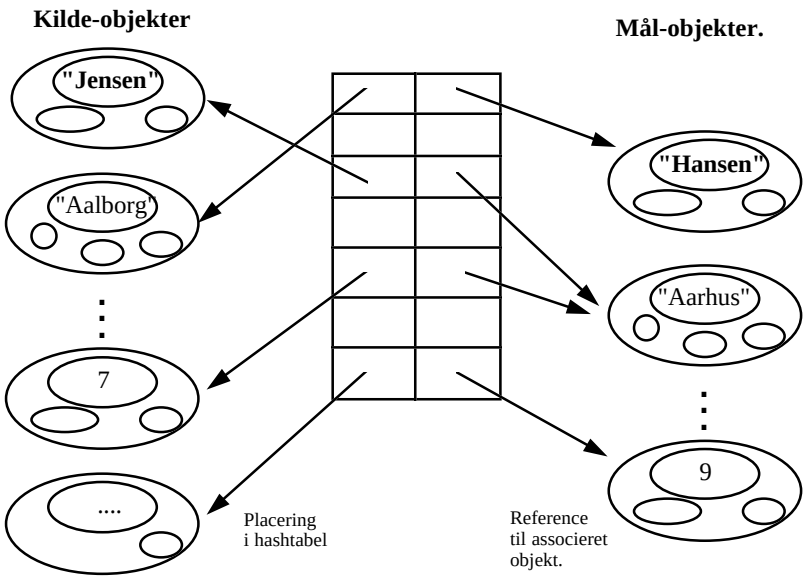
PS1 -- Hashtabeller

Noter

De vurderinger af køretiden, som for hashtabeller er O(1) ovenfor, er udtryk for **"average case" analyse.** Derimod er vurderingen af køretiden for operationerne på balancerede søgertræer udtryk for **"worst case" analyse.**

De gode konstante tidskompleksiteter for hashing kræver, at hashtabellen ikke er for tæt på at være fyldt op. Hvis hashtabellen næsten er fyldt op, kan indsættelse og søgning have så høj køretid som O(n). Ultimativt vil vi ikke kunne indsætte flere elementer i hashtabellen, fordi tabellen er fuld.

Association mellem objekter via hashing.



PS1 -- Hashtabeller

Noter

Hidtil har vi anvendt hashtabeller som en datastruktur til indsættelse af og søgning efter objekter. På denne slide vil vi illustrere en anden anvendelse, nemlig associering/relatering af objekter.

Hvis alle klasser understøtter en hash funktion, er det let at realisere en mekanisme, som muliggør at associere et vilkårligt objekt med et andet vilkårligt objekt (uanset deres type (klasse)) på en meget effektiv måde.

Man kan forestille sig, alle klasser af objekter er forpligtede til at stille en hashfunktion til rådighed. Man kan også forestille sig, at den mest generelle klasse er i stand til at lave den (f.eks. ved at returnere en objektets-adresse i lageret (eller en funktion deraf) modulo tabel-størrelsen som hashværdi).

I tabellen ovenfor afspejler pilene til venstre den aktuelle placering af objekterne i hashtabellen (efter probing). Første søjle anvendes til referencer til kilde-objekterne (således at Søg operationen kan erkende, om et objekt er i tabellen eller ej). Anden søjle indeholder referencer til mål-objekteterne.

Man kan have meget glæde af at kunne knytte vilkårlige objekter sammen på en effektiv måde. Mange Lisp-programmer benytter sig af denne mulighed.