Hashing

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Høsten 2021

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Oversikt uke 44

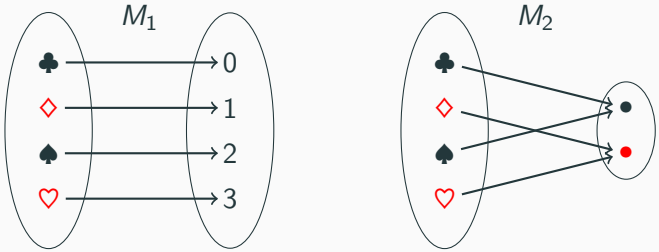
Oversikt uke 44

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hash maps
  - ofte kalt hashtabeller eller dictionaries
- Hash maps brukes for å assosiere en nøkkel med en verdi
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer
  1. gjøre en vilkårlig verdi om til et tall som brukes som en indeks
  2. hvordan håndtere to verdier som får samme indeks
  3. opprettholde en ideell størrelse på arrayet
- Målet er å få effektiv innsetting, oppslag og sletting

Map

Map

- Et map, helt generelt, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, •)
  - slå opp: M_2.`get`(□) ↦ •
  - slette: M_2.`remove`(□)
    - M_2.`get`(□) ↦ ?

Hashmap

- Et hashmap er en måte å materialisere den abstrakte datatypen map
- Vi bruker kun et enkelt array A, sammen med en *hashfunksjon h*
- Hashfunksjonen konverterer en nøkkel k til et tall i der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall
  - uansett hva størrelsen på A er
- Derfor vil vi få *kollisjoner*
  - Altså at to ulike nøkler blir konvertert til det samme tallet i
- Vi skal se på to måter å håndtere kollisjoner på

Hashfunksjoner

Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon h*
  - får en nøkkel k og et positivt heltall N som input
  - og returnerer et positivt heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)
  - Samme input gir *alltid* samme output
- Den må gi få *kollisjoner* (altså være godt distribuert)
  - Ulike input bør hashe til ulike output så ofte som mulig
- Ulike datatyper krever ulike hashfunksjoner

Hashfunksjoner – Eksempler på dårlige hashfunksjoner

Algorithm 1: En inkonsistent hashfunksjon

Input: En nøkkel k og et positivt heltall N

Output: Et heltall i slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2     return Random(0, N − 1)
```

Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

Algorithm 2: En dårlig distribuert hashfunksjon

Input: En nøkkel k og et positivt heltall N

Output: Et heltall i slik at $0 \leq i < N$

```
1 Procedure Collider(k, N)
2     return 0
```

Denne hashfunksjonen er konsistent, men alt kolliderer med alt!

Vi ønsker funksjoner som ikke lider av noen av disse problemene

Hashfunksjoner – Eksempel: strenger

For å hashe en streng kan vi se på hver bokstav som et tall

Algorithm 3: En litt dårlig hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     end
6     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene
- Til slutt returnerer vi denne summen *modulo N*
- Denne er konsistent og kan distribuere ganske godt
  - Hvorfor er den allikevel ikke god i praksis?
  - Hint: $a + b = b + a$

Hashfunksjoner – Eksempel: strenger

Vi fortsetter med samme idé, men introduserer litt mer kaos!

Algorithm 4: En god hashfunksjon på strenger

Input: En streng s og et positivt heltall N
Output: Et heltall h slik at $0 \leq h < N$

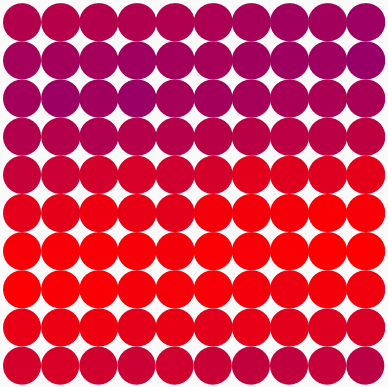
```
1 Procedure HashString(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← 31 · h + charToInt(c)
5     end
6     return h mod N
```

- Denne er konsistent og distribuerer godt!

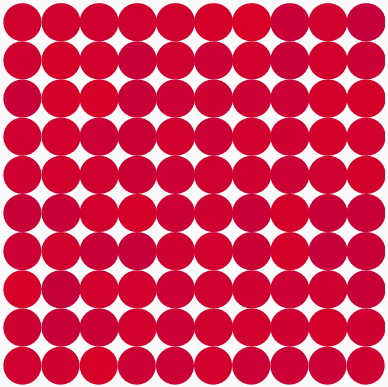
Hashfunksjoner – Eksempel: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder 235886 ord
- Vi kan teste hvor mange kollisjoner vi får for ulike verdier av N
- Hvis vi lar $N = 235886$ så får vi at
  - `HashStringBad`("*algorithm*", N) hasher til 967, med 577 kollisjoner
  - `HashString`("*algorithm*", N) hasher til 184369, med 1 kollisjoner

Hashfunksjoner – Eksempel: HashStringBad vs HashString (N=100)

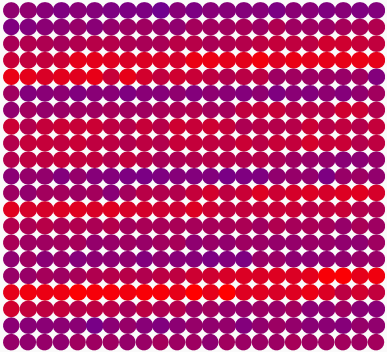


HashStringBad

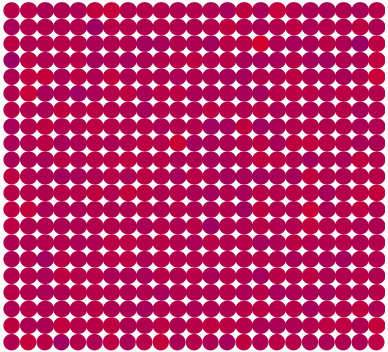


HashString

Hashfunksjoner – Eksempel: HashStringBad vs HashString (N=500)

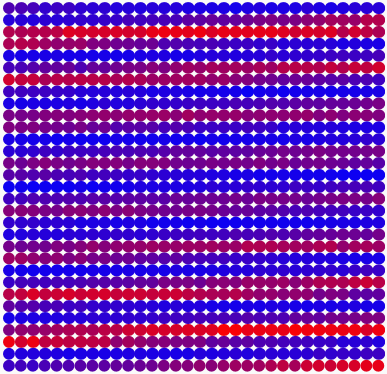


HashStringBad

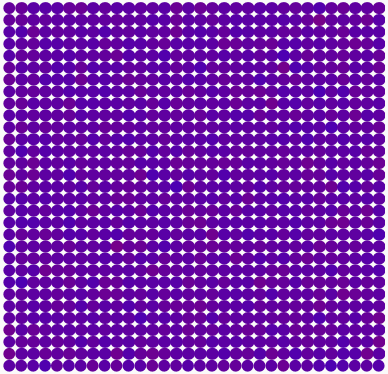


HashString

Hashfunksjoner – Eksempel: HashStringBad vs HashString (N=1000)

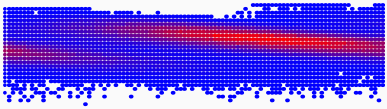


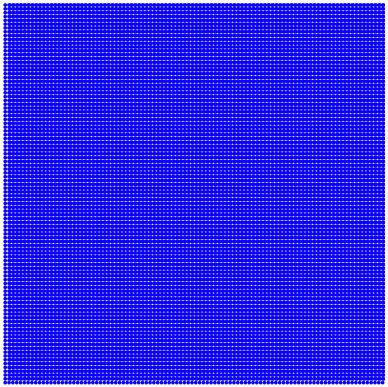
HashStringBad



HashString

Hashfunksjoner – Eksempel: HashStringBad vs HashString (N=10000)





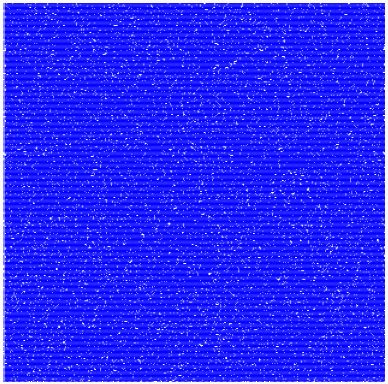
HashStringBad

HashString

Hashfunksjoner – Eksempel: HashStringBad vs HashString (N=100000)



HashStringBad

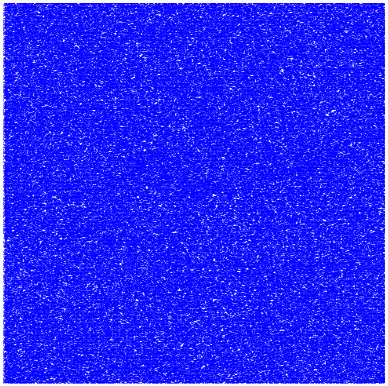


HashString

Hashfunksjoner – Eksempel: HashStringBad vs HashString (N=235886)



HashStringBad



HashString

Kollisjonshåndtering

Kollisjonshåndtering

Vi dekker kun idéene for kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte være en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet
  - Ved kollisjoner ser vi etter neste ledige plass i arrayet
  - Dette er enkelt, men ved sletting må vi tenke oss litt om

Anta at vi har et array A med størrelse N
og inneholder mindre enn N elementer

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. La B $\leftarrow$ A[i]
  2. Hvis B er **null**, opprett en liste og sett inn (k, v)
  3. Ellers setter vi inn (k, v) på slutten av B

     Hvis vi finner en node med nøkkel k på veien vi verdien med v
- Oppslag: gitt en nøkkel k som hasher til i
  1. La B $\leftarrow$ A[i]
  2. Hvis B er **null**, returner **null**
  3. Hvis ikke, slå opp på nøkkelen k i B
- Sletting: gitt en nøkkel k som hasher til i
  1. La B $\leftarrow$ A[i]
  2. Hvis B er **null**, returner
  3. Hvis ikke, slett noden med nøkkelen k i B

Kollisjonshåndtering – Linear probing

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. Hvis A[i] er **null**, sett inn (k, v) på plass i og returner
  2. Hvis nøkkelen til A[i] er k, sett inn (k, v) på plass i og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Oppslag: gitt en nøkkel k som hasher til i
  1. Hvis A[i] er **null**, returner **null**
  2. Hvis nøkkelen til A[i] er k, returner verdien på A[i]
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- Sletting: gitt en nøkkel k som hasher til i
  1. Hvis A[i] er **null**, returner **null**
  2. Hvis nøkkelen på A[i] er ulik k, gå til steg 1 med ($i \leftarrow i + 1 \bmod N$)
  3. Hvis nøkkelen på A[i] er lik k, sett A[i] til **null**
  4. *Tett hullet*

Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Flaggene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass i, tett hullet (uten juks)
     - Søk etter en nøkkel som hasher til i eller tidligere
     - Hvis treffer en **null**, kan vi avslutte
     - Finner vi en slik nøkkel flyttes den (sammen med verdien) til plass i
     - Så må vi tette det nye hullet på samme måte

Effektivitet

Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for separate chaining få lange lister, som bruker lineær tid på alle operasjoner
  - for linear probing få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir
  - forholdet mellom antall elementer n i hashmappen
  - og størrelsen på arrayet N
  - altså er load factor gitt ved $\frac{n}{N}$
- Å finne en ideell load factor bør avgjøres eksperimentelt
  - men antageligvis ligger den mellom 0.5 og 0.75
  - altså kan hashmappen bare være litt mer en halvfull

Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr bare å
  1. lage et større array
  2. sette inn alle elementene fra det forrige arrayet
- Det samme kan man gjøre dersom hashmappen blir for tomt

Effektivitet

- Hash maps er utrolig raske i praksis, og ekstremt anvendelige
- I en verste tilfelle analyse har de $O(n)$ på alle operasjoner
- Men dersom man antar en god hashfunksjon får vi $O(1)$ *forventet* kjøretid
  - Å analysere forventet kjøretid er ikke pensum
- For separate chaining vil dette si at
  - den lengste listen i en hashmap med få elementer
  - er ikke mindre enn den lengste listen i en hashmap med mange
  - fordi vi øker størrelsen på arrayet ved behov
- For linear probing vil det si at
  - det lengste segmentet i en hashmap med få elementer
  - er ikke mindre enn segmentet i en hashmap med mange
- Rehashing tar $O(n)$ tid, men det skjer sjeldent