Oversikt

- Hashing er den underliggende teknikken som gir oss hash maps(hashtabeller eller dictionaries),
- Hash maps brukes for å assosiere en nøkkel med en verdi,
- Vi ønsker å bruke arrayer som underliggende datastrukturer.

Problemer ved Hashing:

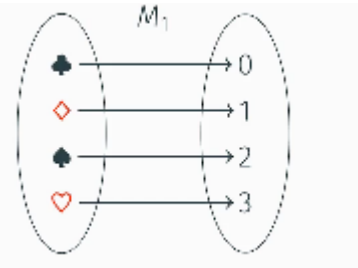
- Gjøre en vilkårlig verdi om til et tall som brukes som en indeks.
- Hvordan håndtere to verdier som får samme indeks(kollisjon).
- Opprettholde en ideell størrelse på arrayet.

Målet er å få effektiv innsetting, oppslag og sletting ved Hashing

Maps og hashmaps

Maps

- Et map, helt generelt, assosierer en nøkkel med nøyaktig en verdi, kan forklares gjennom bildet vist under:



(Kløver assosieres med “0” f.eks)

Hashmaps

- 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 <= i < |A| - 1(størrelsen på arrayet)
    - Vi kaller denne konverteringen for “å hashe”
- Som regel finnes det utrolig mange mulige nøkler
    - Det finnes f.eks uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til |A| tall
    - Uavhengig av hva størrelsen på “A” er
- Derfor vil vi få kollisjoner
    - Altså at to ulike nøkler blir konvertert til det samme tallet “i”

Hashfunksjoner

- En hashfunksjon h
    - får en nøkkel k og et positivt heltall(eller 0) “N” som input
    - og returnerer et positivt heltall(eller 0) slik at 0 <= h(k,N) < N
- Den må være konsistent(altså oppføre seg som en funksjon)
    - Samme input gir alltid samme output, ikke noe annet output
- Det må forekomme få kollisjoner (Altså være godt distribuert i forhold til “i” og nøklene assosiert med “i”)
    - Ulike input bør hashe til ulike output så ofte som mulig

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: $a + b = b + a$

6

ASCII-tabellen er relevant her da vi kan henvise til ASCII-tabellen for å angi hver bokstav som et tall, f.eks a = 97(i likhet med ASCII). Hensikten med funksjonen er kun å hashe strengen til en gitt bokstav. Denne funksjonen er dårlig ettersom at den er kommutativ, altså rekkefølgen til (a+b = b+a) spiller ingen rolle. Så dermed vil den kollidere hvor vi vil få tilfeller hvor bokstavene hashes til samme "i".

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

Kollisjonshåndtering

Ideene bak kollisjonshåndtering - tilfeller hvor to nøkler referer til samme "i", så har vi to ideer bak slik håndtering.

Ene er ved "Separate chaining" hvor vi lar hver plass i arrayet peke til en "bøtte"

- Vi tar utgangspunkt i at hver bøtte er en lenket liste

Andre er 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

Separate Chaining

Kollisjonshåndtering – Separate chaining

- Innsetting: gitt en nøkkel k som hasher til i, og en verdi v
  1. La B ← 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 ← 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 ← 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 – 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