Föreläsning
Datastrukturer (DAT037)

Nils Anders Danielsson

2015-11-16

Idag

- Mängder, avbildningar.
- Hashtabeller.
- Sortering.

Pseudokod

- Blandning av programmeringsspråk, matematisk notation och naturligt språk.
- Mål: fokusera på det viktiga, undvik onödiga detaljer.
- Ibland kan det vara bra med fler detaljer, ibland färre.

Mängder, avbildningar

Mängder

ADT:

- Konstruerare för tom mängd.
- `insert`(x): Lägger till x till mängden.
  Mängder innehåller ej dubbletter.
- `member`(x): Avgör om x finns i mängden.
- `delete`(x): Tar bort x från mängden.

Avbildningar/maps

ADT:

- Konstruerare för tom avbildning.
- `insert(`k`, `v`)`: Lägger till bindningen $k \mapsto v$. Om det finns en gammal bindning $k \mapsto v'$ skrivs den (t ex) över.
- `lookup(`k`)`: Om det finns en bindning $k \mapsto v$ så ges v som svar.
- `delete(`k`)`: Tar bort bindningen $k \mapsto v$ (om det finns en sådan bindning).

Krav på nyckeltypen

Olika mängd-/avbildningsdatastrukturer har olika krav på nyckeltypen:

- Likhetstest.
- Olikhetstest/komparator.
- Hashfunktion.

Hashtabeller

Hashtabeller

- Implementerar mängd- eller avbildnings-ADTn.
- Idé: Array plus funktion som säger i vilken "hink" man ska lägga elementen/bindningarna.

Hashfunktioner

- Tar värden till heltal (t ex).
- Krav: Om $x = y$ ska $h(x) = h(y)$.
- Dock kan $h(x) = h(y)$ för $x \neq y$: kollision.
- Finns olika metoder för att hantera kollisioner.

Hashtabeller, separat kedjning

```
class HashTable<A>:
  private int         size
  private List<A> [] table

  HashTable(int capacity):
    initialise(capacity)

  private initialise(int capacity):
    if capacity <= 0 then
      raise error

    size  = 0
    table = new array of size capacity
    for each position i in table do
      table[i] = new LinkedList<A>()
```

Hashtabeller, separat kedjning

```
member(A x):
  List<A> bucket = table[x.hash() mod table.length()]
  return bucket.contains(x)
```

Obs:

- Jag antar att $n \texttt{ mod } s \in \{ 0, 1, ..., s - 1 \}$ (för $s > 0$).
- Javas % uppfyller inte det kravet, t ex för $n = -5$ och $s = 107$.

Hashtabeller, separat kedjning

```
delete(A x):
  List<A> bucket = table[x.hash() mod table.length()]

  if bucket.contains(x) then
    bucket.remove(x)
    size--
```

Hashtabeller, separat kedjning

```
insert(A x):
  List<A> bucket = table[x.hash() mod table.length()]

  if bucket.contains(x) then
    bucket.remove(x)
    bucket.add(x)
  else
    bucket.add(x)
    size++
    if size is "too large" then
      rehash
```

Hashtabeller, separat kedjning

```
private rehash:
  oldtable = table

  initialise("suitable" capacity)

  for each position i in oldtable do
    for each element x in oldtable[i] do
      insert(x)
```

Kan undvika att beräkna hashkoder igen
genom att spara dem tillsammans med elementen.

Skapa en hashtabell med kapacitet 5, och stoppa in följande värden: $3, 7, 8, 2, 9, 11$.
Använd hashfunktionen $h(x) = x$.
Hur många kollisioner inträffar?
(Tabellens kapacitet ändras inte.)

- 0.
- 1.
- 2.
- 3.
- 4.
- 5.

Hashtabeller

“För stor”, “lämplig” kapacitet:

- Lastfaktor: storlek/kapacitet.
- Hög lastfaktor ger ev fler kollisioner.
- Låg lastfaktor $\Rightarrow$ många tomma hinkar.
- JDK 7 `HashMap` (använder separat kedjning): lastfaktor max 0.75 (kan ändras).

Hashtabeller

- Kursbokens rekommendation:
  kapacitet primtal.
- Skydd mot vissa dåligt designade hashfunktioner.
- Säg att alla hashkoder har formen $im + n$ (för $i = 0, 1, 2, ...$):
  - Om kapaciteten är km så används som mest k hinkar.
  - Om kapaciteten och m är relativt prima så kan alla hinkar användas.
- Kapacitet 2^k leder till kollision om sista k bitarna i $h(x)$ och $h(y)$ är lika:
  övriga bitar ignoreras.

Hashtabeller

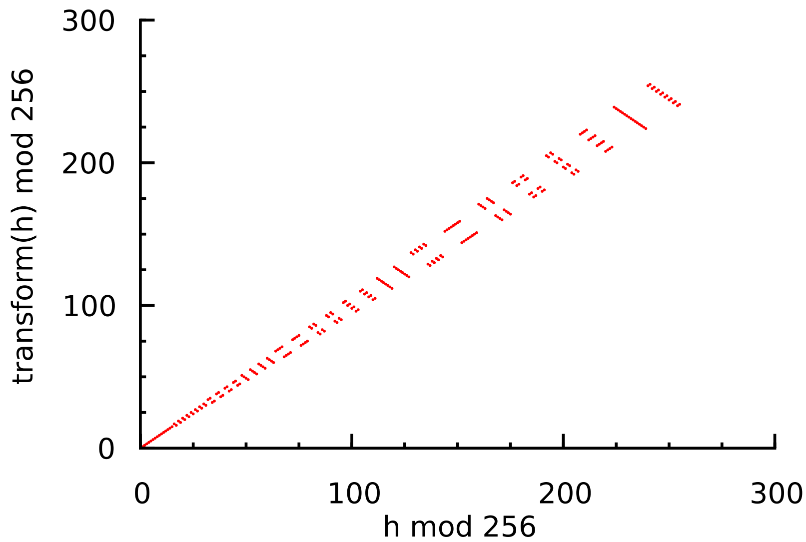
- JDK 6–8 `HashMap`: kapacitet 2^k.
- För att undvika problem transformeras hashkoderna med en andra hashfunktion. I JDK 6:

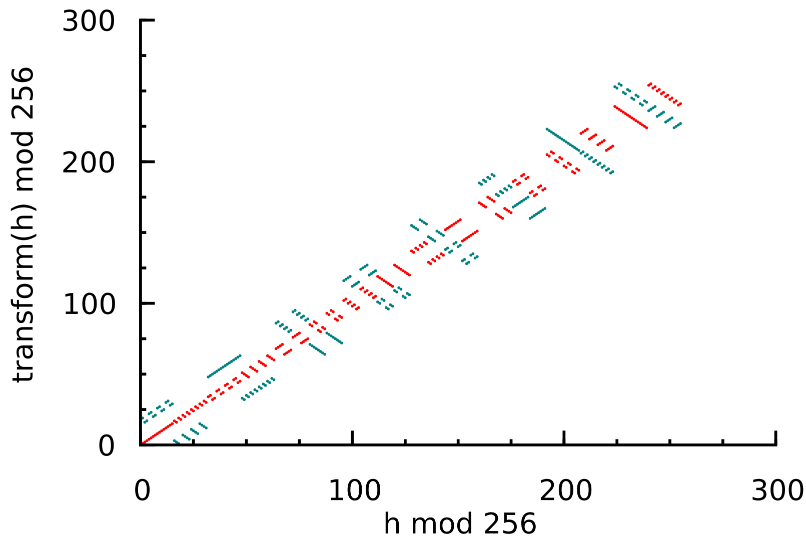
```
h ^= (h >>> 20) ^ (h >>> 12);
return h ^ (h >>> 7) ^ (h >>> 4);
```

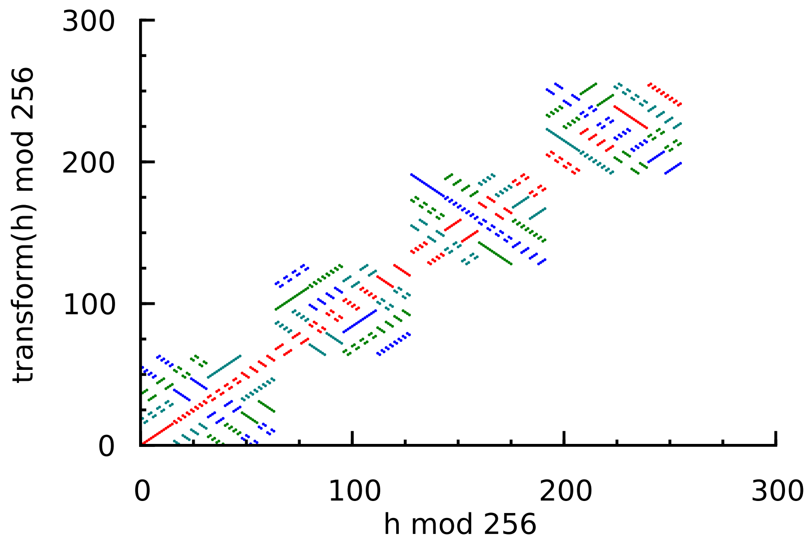
  (`^` är xor, n `>>>` k är $n/2^k$.)
- I JDK 8:

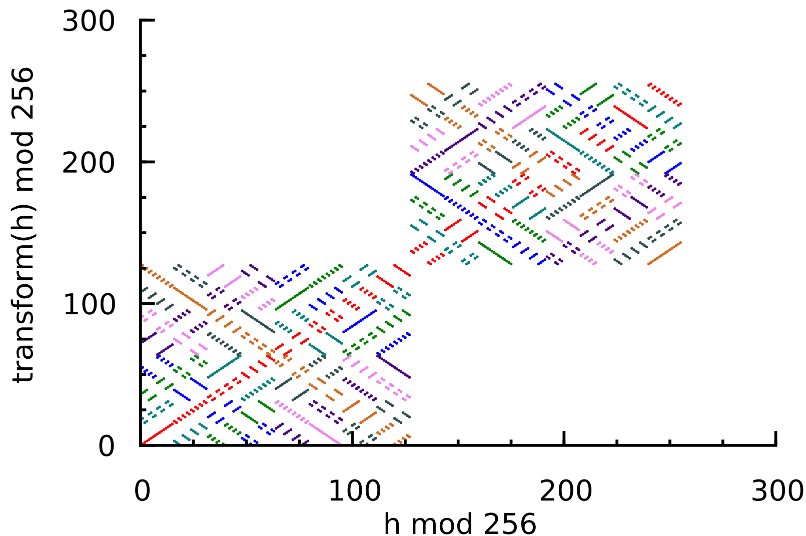
```
h = h ^ (h >>> 16)
```

  Dessutom: Hinkar med många element använder (kanske) balanserade sökträd.

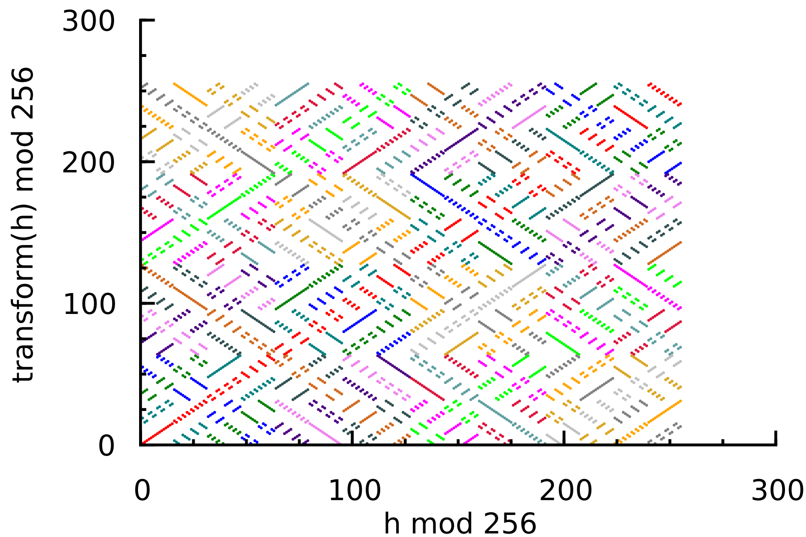

0–255
300
200
100
0
transform(h) mod 256
0
100
200
300
h mod 256
0–255


0–511
300
200
100
0
transform(h) mod 256
0
100
200
300
h mod 256
0–255
256–511

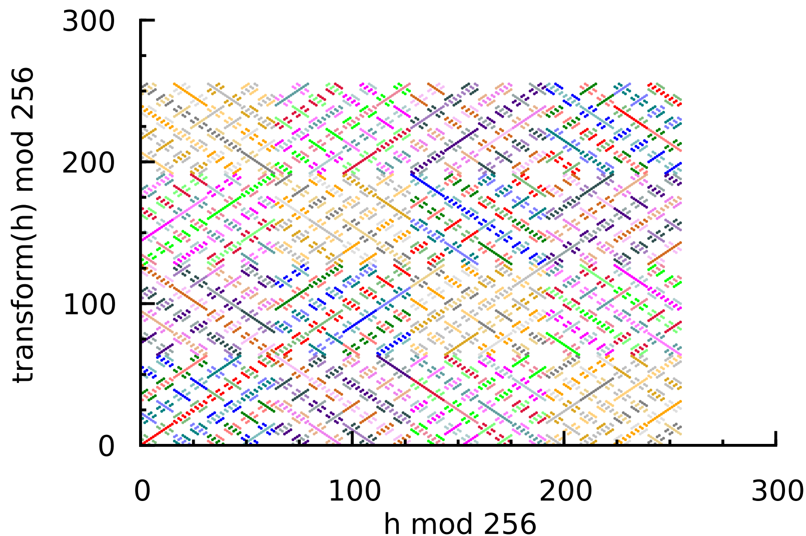

0–1023
300
200
100
0
transform(h) mod 256
0
100
200
300
h mod 256
0–255
256–511
512–767
768–1023

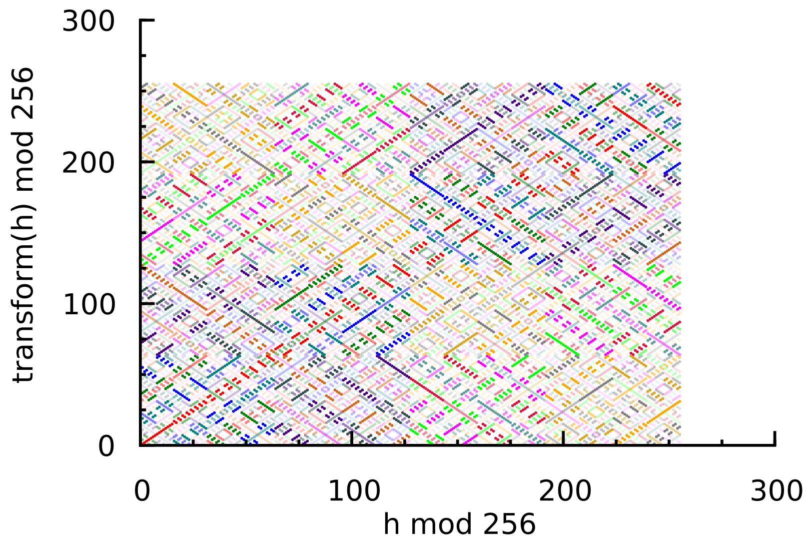


0–2047
300
200
100
0
transform(h) mod 256
0
100
200
300
h mod 256
0–255
256–511
512–767
768–1023



0–4095
300
200
100
0
transform(h) mod 256
0
100
200
300
h mod 256
0–255
256–511
512–767
768–1023


0–8191
300
200
100
0
transform(h) mod 256
0
100
200
300
h mod 256
0–255
256–511
512–767
768–1023


0–16383
300
200
100
0
transform(h) mod 256
0
100
200
300
h mod 256
0–255
256–511
512–767
768–1023

Hashfunktioner

- Bra hashfunktion:
  snabb, liten risk för kollisioner.
- Lätt att skriva dålig hashfunktion.
- Svårt att designa bra hashfunktion.
- Kursbokens ”bra” hashfunktion för strängar
  är kanske inte jättebra
  (men fungerar nog OK för hashtabeller;
  Java använder en liknande definition).
- Hur skriver man en bra hashfunktion?

Hashfunktioner

- Finns ett antal hashfunktioner som påstås fungera bra:
  - MurmurHash.
  - CityHash.
  - SpookyHash.
  - ...

  Kanske är bra att använda någon av dem.
- Kan vara lämpligt att testa hashfunktionen.

Hashfunktioner

Finns bibliotek som kan vara till hjälp vid definition av hashfunktion för egendefinierad klass.

Exempel:

- JDK 8: `java.util.Objects.hash`.
  Enkel hashfunktion, liknar den för strängar.

  ```
  public int hashCode() {
    return Objects.hash(field1, field2, field3);
  }
  ```

  (Glöm inte $x = y \Rightarrow h(x) = h(y)$!)
- `com.google.common.hash`.
  Flera olika hashfunktioner.

Hashtabeller, separat kedjning

Tidskomplexitet med
$O(1)$ perfekt hashfunktion (inga kollisioner),
lastfaktor ≤ 1,
$O(1)$ likhetstest,
och kapacitet 2^k:

- Tom hashtabell: $O(kapacitet)$.
- `insert`: $O(1)$ (amorterat).
- `member`: $O(1)$.
- `delete`: $O(1)$.

Hashtabeller, separat kedjning

Tidskomplexitet med
$O(1)$ mycket dålig hashfunktion (bara kollisioner),
lastfaktor ≤ 1,
$O(1)$ likhetstest,
och kapacitet 2^k:

- Tom hashtabell: $O(kapacitet)$.
- `insert`: $O(n)$.
- `member`: $O(n)$.
- `delete`: $O(n)$.

Hashtabeller, separat kedjning

Tidskomplexitet med
$O(1)$ mycket dålig hashfunktion (bara kollisioner),
lastfaktor ≤ 1,
$O(1)$ likhetstest,
och kapacitet 2^k:

- Tom hashtabell: $O(\mathit{kapacitet})$.
- `insert`: $O(n)$.
- `member`: $O(n)$.
- `delete`: $O(n)$.

Notera: En angripare kanske kan välja indata på ett sådant sätt att man får många kollisioner.

Vad är tidskomplexiteten för borttagande av n strängar, var och en med n tecken, från en hashtabell som innehåller alla strängarna? Anta att hashfunktionen är perfekt, och linjär i strängarnas storlek.

- $\Theta(1)$.
- $\Theta(n)$.
- $\Theta(n^2)$.
- $\Theta(n^3)$.

Hashtabeller, öppen adressering

- Inga länkade listor.
- Vid kollision:
  element sparas på annan position i arrayen.
- `delete` lite krånglig (gravstenar).
- Kan i vissa fall ge bättre cacheutnyttjande.

Sortering

Insättningssortering

```
// Precondition för insert x xs: xs är sorterad.
insert :: Ord a => a -> [a] -> [a]
insert x []         = [x]
insert x (y : ys)
  | x <= y     = x : y : ys
  | otherwise = y : insert x ys

insertionSort :: Ord a => [a] -> [a]
insertionSort []         = []
insertionSort (x : xs) = insert x (insertionSort xs)
```

Insättningssortering

- *Stabil* sorteringsalgoritm:
  bevarar inbördes ordning mellan “lika” element.
- Anta att `(x, _) <= (y, _) = x <= y`:
  `insertionSort [(1, 3), (1, 2), (2, 0), (1, 4)] =`
  `[(1, 3), (1, 2), (1, 4), (2, 0)]`

Vad är bästa- och värstafalls-tidskomplexiteten för `insertionSort`, givet att jämförelser tar konstant tid?

- $\Theta(n)$, $\Theta(n)$.
- $\Theta(n)$, $\Theta(n^2)$.
- $\Theta(n^2)$, $\Theta(n^2)$.

Mergesort

```
// Precondition: Listorna är sorterade.
merge :: Ord a => [a] -> [a] -> [a]
merge []        ys        = ys
merge xs        []        = xs
merge (x : xs) (y : ys)
  | x <= y    = x : merge xs (y : ys)
  | otherwise = y : merge (x : xs) ys
```

Mergesort

```
mergeSort :: Ord a => [a] -> [a]
mergeSort xs
  | |xs| <= 1 = xs
  | otherwise = merge (mergeSort ys) (mergeSort zs)
  where ys = first ⌈|xs|/2⌉ elements from xs
        zs = last  ⌊|xs|/2⌋ elements from xs
```

Mergesort

```
mergeSort :: Ord a => [a] -> [a]
mergeSort xs
  | |xs| <= 1 = xs
  | otherwise = merge (mergeSort ys) (mergeSort zs)
  where ys = first ⌈|xs|/2⌉ elements from xs
        zs = last  ⌊|xs|/2⌋ elements from xs
```

- I kursboken: mergesort för arrayer.

Är `mergeSort` stabil?

- Ja.
- Nej.

Mergesort

Värstafallstidskomplexitet (om <= tar konstant tid):

- `merge`: Linjär i listornas längd: $O(|\mathtt{xs}| + |\mathtt{ys}|)$.
- `mergeSort`:

$$
\begin{aligned}
T(0) &= 1 \\
T(1) &= 1 \\
T(n) &= n + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right), \text{ om } n \geq 2
\end{aligned}
$$

Mergesort

Anta att $n = 2^k$, $k \in \mathbb{N}$.

$$\begin{aligned}
T(n) &= n + 2T(n/2) \\
&= n + 2(n/2 + 2T(n/4)) \\
&= 2n + 4T(n/4) \\
&= 2n + 4(n/4 + 2T(n/8)) \\
&= 3n + 8T(n/8) \\
&= \ldots \\
&= \log_2 n \cdot n + nT(n/n) \\
&= n \log_2 n + n \\
&= \Theta(n \log n)
\end{aligned}$$

Divide and conquer

- Algoritmteknik: söndra och härska (divide and conquer).
- Om man skapar två delproblem av halv storlek på linjär tid, och dessutom slår ihop delresultaten på linjär tid, så får man en $O(n \log n)$-algoritm.

Sammanfattning

Idag:

- Mängder, avbildningar.
- Hashtabeller.
- Sortering.

Nästa gång:

- Grafer.