Föreläsning 5
Datastrukturer (DAT037)

Fredrik Lindblad[1]

13 november 2017

[1]Slides skapade av Nils Anders Danielsson har använts som utgångspunkt. Se
`http://www.cse.chalmers.se/edu/year/2015/course/DAT037`

Innehåll

- Hashtabeller

Hashtabeller

Hashtabeller

- Implementerar mängd- eller avbildnings-ADTn.
- Använder array för att lagring av alla element/bindningar.
- Nyckeltypen kan vara vad som helst, t.ex. stora heltal eller strängar.
- Kan inte ha en plats för varje tänkbar nyckel (de är alldelas för många).
- Idé: Funktion som säger i vilken "hink" man ska lägga elementen/bindningarna.
- `hashCode()` finns i Javas `Object`.

Hashfunktioner

- $h(k)$ funktion från nyckel till heltal.
- Dessa funktioner är kopplade till nyckel-typen och känner inte till hashtabellens storlek.
- Låt index vara $f(k) = h(k) \bmod n \in \{0 \ldots n-1\}$ där n är arrayens storlek.
- Krav: Om $x = y$ ska $h(x) = h(y)$.
- Dock kan $h(x) = h(y)$ och $f(x) = f(y)$ för $x \neq y$: kollision.
- Finns olika metoder för att hantera kollisioner, separat kedjning (chaining), öppen adressering

Rehashing

- När hashtabellen fylls blir så småningom antalet kollisioner för många.
- Likt dynamisk array, allokera (multiplikativt) större array.
- Nycklarna har samma hash-kod men p.g.a. att arrayens storlek ändras kommer de hamna på andra platser och fördelas över hela nya arrayen.

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.

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.

Svar: 2

Hashtabeller, separat kedjning

Varje hink kan lagra en mängd element, traditionellt i en länkad lista. På nästföljande slides följer pseudo-kod för en implementering.

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)

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)
```

Hashtabeller, öppen adressering

- Inga länkade listor.
- Vid kollision:
  element sparas på annan position i arrayen.
- Måste hantera det faktum att borttagna element en gång fanns där, annars kan man misslyckas att hitta andra element.
- Vid försök i tittar man på plats $(f(k) + p(i)) \bmod n$, där $p(0) = 0$ (första försöket).
- Klassiska val av $p(i)$ är linear probing, quadratic probing.
- Annat alternativ är dubbelhashning.

Olika val av stegfunktion

- Linear probing – fortsätt på nästa plats ($p(i) = i$).
  Risk för hopklumpning (clustering). Effektivt m.a.p. på minnescachning.
- Quadratic probing – $p(i) = i^2$
  Mindre risk för clustering för sökning för element som vars hashkod pekar på olika index hoppar iväg i olika banor. Dåligt för cachning.
- Dubbelhashning – hoppar ett antal steg som bestäms av andra hashfunktion
  ($p(i) = i \cdot h_2(k)$, $h_2(k) \neq 0$)
  Ännu mindre clustering. Mer tidskrävande att beräkna två hashfunktioner.

Borttagna element

Standardsättet att hantera borttagna element vid öppen adressering är att varje plats i arrayen har tre tillstånd, tom, upptagen och borttagen. Från början är alla platser tomma. När element läggs till blir en plats upptagen. När element tas bort blir en plats borttagen. För att hitta element hanteras en borttagen plats som upptagen; man fortsätter leta. När man sätter in nya element så hanteras en borttagen plats som tom; man sätter in elementet där.

Borttagna element

Detta leder efter upprepad insättning och borttagning till många oanvända platser med tillståndet borttagen och det ökar söktiden. Rehashing kan behövas bara för att blir av med alla borttagna platser. Lazy deletion är en metod som minskar problemet något. Det innebär att man vid sökningar flyttar element till den första platsen markerad borttagen som man passerade innan man hittade rätt. Ett sätt att helt slippa borttagna platser, som fungerar för linjär probning, är att vid borttagning starta en kedja av framflytt av element för att fylla ut den tomma platsen som skapats.

Hashtabellers storlek

Lämplig kapacitet på arrayen:

- 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).
- Vid öppen adressering är en lägre lastfaktor lämplig (t.ex. 0.5)

Hashtabellers storlek

- Kursbokens rekommendation: kapacitet primtal.
- Skyddar 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.

Hashfunktioner

- Bra hashfunktion: snabb, liten risk för kollisioner.
- Svårt att designa bra hashfunktion.
- Bra hashfunktion: bra fördelning över heltalen för de instansen av nyckeltypen som faktiskt förekommer.
- Bra hashfunktion: inte bara en liten del av nyckelvärdet ska påverka hashkoden.
- Kursbokens "bra" hashfunktion för strängar: $h(x) = x_0 + 37x_1 + 37^2x_2 + ...$
  Java använder en liknande definition.

Hashfunktioner

- För icke-muterbar data (t.ex. strängar) kan man undvika att beräkna hashkoder igen genom att spara dem tillsammans med elementen.
- 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.

Komplexitet

Tidskomplexitet med
$O(1)$ perfekt hashfunktion (inga kollisioner),
lastfaktor ≤ 1:

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

Komplexitet

Tidskomplexitet med
$O(1)$ mycket dålig hashfunktion (bara kollisioner),
lastfaktor ≤ 1:

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

Förväntad tidsåtgång

Genomstittsligt antal jämförelser för lyckade sökning som funktion av lastfaktorn:

- Öppen adressering:
  $c = \frac{1}{2}(1 + \frac{1}{1-L})$
- Separat kedjning:
  $c = 1 + \frac{L}{2}$

Förväntad tidsåtgång

L	öppen adr.	kedjning
0	1	1
0.5	1.5	1.25
0.75	2.5	1.38
0.9	5.5	1.45
0.95	10.5	1.48
1	-	1.5
2	-	2